SUNDB数据库管理系统
V5.0-SQL手册

SQL Manual

目录

1.SQL Elements

1.1 Syntax Elements（语法元素）

Identifiers（标识符）

标识符分为常规标识符（ordinary identifier）与分隔标识符（delimited identifier）

常规标识符由字母或字母和数字组成在内部将所有字符替换为大写字母使用因此常规标识符不区分大小写

以下为常规标识符的示例

```
SUNDB

Sundb
```

分隔标识符由带有双引号（“）的字母或字母和数字组成在内部使用所记述的字符因此使用分隔标识符时需要区分大小写

以下为分隔标识符的示例

```
"SUNDB"

"Sundb"
```

字面量（Literals）

字面量（Literals）是non-null值的表示法

字符串字面量（Text Literals）

字符串字面量是字符串（string ）与二进制字符串（binary string）的表示法

字符串的字符串字面量的表示方法为在字符串的开头和结尾使用单引号（'）不仅是双引号(")除单引号（'）外的所有字符串都可以成为写入单引号（'）中的字符串若需要在字符串中使用单引号(')则需要无空格的连续使用两次单引号(')一个字符串中最多可以写入4000个字符

以下为字符串的字符串字面量的表示示例

```
'SUNDB'
'Csii''s DBMS'
```

二进制字符串的字符串字面量以x'或X'开头并以'结束的16进制字符串表示16进制的字符串仅可写入对应0~9, A(a)~F(f)的字符两位数的字符表示一个字节因此十六进制字符串的长度应为偶数二进制字符串的最大长度为4000字符

以下为二进制字符串的字符串字面量的表示示例

```
x'001f'
X'FF0A'
x'aF37BBc013'
```

数值字面量（Numeric Literals）

Numeric字面量（numeric literals）是数值字面量的表示形式可以写入整数或带小数点的数字数值字面量的相关语法如下

```
[ + | - ] <digits> [ . <digits> ] [ E | e [ + | - ] <digits> ] [ f | F | d | D ]
```

- 最前面的+- 表示整个数字为正数或负数+-可以省略省略则默认为+正数
- <digits>可以无空格的列出0~9之间的数字可通过小数点(.)表示带小数点的数字
- 第一个<digits>后面可记述指数（Exponent）形式为此记述表示指数（Exponent）的E或e在其后记述Exponent的符号+-之后记述<digits>Exponent的符号也可以省略省略则默认为+正数
- 最后数字后可出现f, F, d, D等字符该字符表示其为BINARY_FLOAT, BINARY_DOUBLE的数字如果省略相应字符则视为NUMBER类型的数字

以下为数值字面量的表示示例

```
20

+123.45

0.03

+1.23E-02

-1.5


10f

+123.45F
```

```
1.2E-3F

-22d

123.45D

-1.23E+05D
```

日期时间字面量（Datetime Literals）

日期时间字面量（Datetime Literals）是日期/时间类型的字面量形式datetime value可以使用string字面量指定或使用TO_*函数（TO_DATE等）转换character或numeric value指定

日期/时间类型有DATE, TIME, TIME WITH TIME ZONE, TIMESTAMP, TIMESTAMP WITH TIME ZONE

日期字面量（Date Literals）

日期字面量（date literals）可表示为DATE'string literal'或TO_DATE(string_literal [, format])的形式

- DATE'string literal'
  - 日期类型的格式为'SYYYY-MM-DD'
  - DATE'2002-07-15'
- TO_DATE(string_literal [, format])
  - 未指定格式时日期类型的格式默认为NLS_DATE_FORMAT
  - 指定格式时使用指定的格式
- 日期类型包含年月日时分秒（fractional seconds(小数点之后的秒)除外）
- 如果省略日期则指定为当前月份的第一天

- 如果省略时分秒则默认指定为零点
  - HH24格式时是'00:00:00'
  - HH12格式时是'12:00:00'
- 可使用TRUNC（date）将包含时分秒的DATA值的时分秒设置为默认值零点
  - 例如TRUNC(SYSDATE)：SYSDATE是包含年月日时分秒的值
- 比较除时分秒外的DATE值的年月日值时要使用TRUNC函数将时分秒值设置为零点

详细内容参考**TO_DATE**, **日期时间格式字符串**, **NLS_DATE_FORMAT**

以下为日期类型字面量的表示示例

```
DATE'2002-07-15'
```

- 未指定格式的NLS_DATE_FORMAT = 'YYYY-MM-DD' 时

```
TO_DATE( '2002-07-15' )
```

- 指定格式时

```
TO_DATE( '15-JUL-02', 'DD-MON-RR' )
TO_DATE( '2002-07-15 00:00:00', 'YYYY-MM-DD HH24:MI:SS' )
TO_DATE( '2002-07-15 13:25:30', 'YYYY-MM-DD HH24:MI:SS' )
```

- 日期类型中未写入日期时(存储为当前月份的第一天)

```
gSQL> SELECT TO_DATE( '2000-07', 'YYYY-MM' ) FROM DUAL;
  TO_DATE( '2000-07', 'YYYY-MM' )
  -------------------------------
```

```
2000-07-01
```

- 日期类型中未写入时分秒时(时分秒存储为零点)

```
gSQL> SELECT
        TO_CHAR( DATE'2002-07-15', 'YYYY-MM-DD HH24:MI:SS' ) AS RESULT
        FROM DUAL;
  RESULT
  -------------------
  2002-07-15 00:00:00
```

- 将包含时分秒的DATE value(SYSDATE)的时分秒设置为默认值零点时

```
gSQL> SELECT
        TO_CHAR( SYSDATE,
                 'YYYY-MM-DD HH24:MI:SS' ) AS RESULT_SYSDATE,
        TO_CHAR( TRUNC( SYSDATE ),
                 'YYYY-MM-DD HH24:MI:SS' ) AS RESULT_TRUNC_SYSDATE
        FROM DUAL;
  RESULT_SYSDATE      RESULT_TRUNC_SYSDATE
  ------------------- -------------------
  2014-08-19 10:06:49 2014-08-19 00:00:00
```

- 比较date value中除时分秒外的年月日时

```
gSQL> SELECT
        TO_DATE( '2002-08-12' ) =
```

```
       TRUNC( TO_DATE( '2002-08-12 23:59:59', 'YYYY-MM-DD HH24:MI:SS' ) )
       AS RESULT FROM DUAL;

RESULT
------
TRUE
```

时间字面量（Time Literals）

时间字面量（time literals）可表示为TIME'string literal'或TO_TIME(string_literal [, format])的形式

- TIME'string literal'
  - 时间类型的格式为'HH24:MI:SS[.[FF6]]'
  - TIME'15:30:59.999999'
- TO_TIME(string_literal [, format])
  - 未指定格式时时间类型的格式为NLS_TIME_FORMAT
  - 指定格式时使用指定的格式

时间类型包含时分秒小数点后面的秒（fractional seconds）

fractional seconds最长可指定6 digit的数字格式

详细内容参考**TO_TIME**，**日期时间格式字符串NLS_TIME_FORMAT**

以下为时间类型字面量的表示示例

```
TIME'15:30:59.999999'
```

- 未指定格式的 NLS_TIME_FORMAT = 'HH24:MI:SS.FF6' 时

```
TO_TIME( '15:30:59.999999' )
```

- 指定格式时

```
TO_TIME( '09.45.03.546873 AM', 'HH12.MI.SS.FF6 AM' )
TO_TIME( '09:45:03', 'HH12:MI:SS' )
```

有时区的时间字面量（Time with time zone Literals）

带时区的时间字面量（Time with time zone Literals）可表示为TIME'string literal', TIME WITH TIME ZONE'string literal'或TO_TIME_WITH_TIME_ZONE(string_literal [, format]) , TO_TIME_TZ(string_literal [, format])的形式

- TIME'string literal'或TIME WITH TIME ZONE'string literal'
  - 带时区的时间类型的格式为 'HH24:MI:SS[.[FF6]] TZH:TZM'
  - TIME'15:30:59.999999 +09:00'
  - TIME WITH TIME ZONE'15:30:59.999999 +09:00'
- TO_TIME_WITH_TIME_ZONE(string_literal [, format])
  - 未指定格式时带时区的时间类型的格式为NLS_TIME_WITH_TIME_ZONE_FORMAT
  - 指定格式时使用指定的格式

带时区的时间类型包含时分秒小数点后面的秒（fractional seconds）时区偏移（时区小时时区分钟）

小数点后面的秒(fractional seconds)最长可指定6 digit的数字格式

详细内容参考**TO_TIME_WITH_TIME_ZONE**，**日期时间格式字符串**，

NLS_TIME_WITH_TIME_ZONE_FORMAT

以下为带时区的时间字面量的表示示例

```
TIME'15:30:59.999999 +09:00'

TIME WITH TIME ZONE'15:30:59.999999 +09:00'
```

- 未指定格式的NLS_TIME_WITH_TIME_ZONE_FORMAT = 'HH24:MI:SS.FF6 TZH:TZM' 时

```
TO_TIME_WITH_TIME_ZONE( '15:30:59.999999 +09:00' )

TO_TIME_TZ( '15:30:59.999999 +09:00' )
```

- 指定格式时

```
TO_TIME_WITH_TIME_ZONE( '09.45.03.546873 +09:00 AM',

                        'HH12.MI.SS.FF6 TZH:TZM AM' )
```

时间戳字面量（Timestamp Literals）

时间戳字面量（timestamp literals）表示为 TIMESTAMP'string literal'或 TO_TIMESTAMP(string_literal [, format]) 的形式

- TIMESTAMP'string literal'
  - 时间戳类型的格式为'SYYYY-MM-DD HH24:MI:SS[.[FF6]]'
  - TIMESTAMP'2002-07-15 15:39:59.999999'
- TO_TIMESTAMP(string_literal [, format])
  - 未指定格式时时间戳类型的格式为NLS_TIMESTAMP_FORMAT

- 指定格式时使用指定的格式

时间戳类型包含年月日时分秒小数点后面的秒（fractional seconds）

小数点后面的秒最长可指定6 digit的数字格式

详细内容参考**TO_TIMESTAMP**，**日期时间格式字符串**, **NLS_TIMESTAMP_FORMAT**

以下为时间戳字面量的表示示例

```
TIMESTAMP'2002-07-15 15:39:59.999999'
```

- 未指定格式的NLS_TIMESTAMP_FORMAT = 'YYYY-MM-DD HH24:MI:SS.FF6'时

```
TO_TIMESTAMP( '2002-07-15 15:39:59.999999' )
```

- 指定格式时

```
TO_TIMESTAMP( '15-JUL-02 11.06.30.123456 AM',
              'DD-MON-RR HH12.MI.SS.FF6 AM' )
```

有时区的时间戳字面量（Timestamp with time zone Literals）

带时区的时间戳字面量（timestamp with time zone literals）表示为TIMESTAMP'string literal', TIMESTAMP WITH TIME ZONE'string literal'或TO_TIMESTAMP_WITH_TIME_ZONE(string_literal [, formt]), TO_TIMESTAMP_TZ(string_literal [, format])的形式

- TIMESTAMP'string literal' 或 TIMESTAMP WITH TIME ZONE'string literal'
  - 带时区的时间戳格式为'SYYYY-MM-DD HH24:MI:SS[.[FF6]] TZH:TZM'

  - TIMESTAMP'2002-07-15 15:39:59.999999 +09:00'
  - TIMESTAMP WITH TIME ZONE'2002-07-15 15:39:59.999999 +09:00'
- TO_TIMESTAMP_WITH_TIME_ZONE(string_literal [, formt])
  - 未指定格式时有时区的时间戳的格式为NLS_TIMESTAMP_WITH_TIME_ZONE_FORMAT
  - 指定格式时使用指定的格式

带时区的时间戳类型包含年月日时分秒小数点后面的秒（fractional seconds）时区偏移（时区小时,时区分钟)

fractional seconds最长可指定为6 digit的数字格式

详细内容参考**TO_TIMESTAMP_WITH_TIME_ZONE**，**日期时间格式字符串**，**NLS_TIMESTAMP_WITH_TIME_ZONE_FORMAT**

以下为带时区的时间戳字面量的表示示例

```
TIMESTAMP'2002-07-15 15:39:59.999999 +09:00'
TIMESTAMP WITH TIME ZONE'2002-07-15 15:39:59.999999 +09:00'
```

- 未指定格式的NLS_TIMESTAMP_WITH_TIME_ZONE_FORMAT = 'YYYY-MM-DD HH24:MI:SS.FF6 TZH:TZM'时

```
TO_TIMESTAMP_WITH_TIME_ZONE( '2002-07-15 15:39:59.999999 +09:00' )
TO_TIMESTAMP_TZ( '2002-07-15 15:39:59.999999 +09:00' )
```

- 指定格式时

```
TO_TIMESTAMP_WITH_TIME_ZONE( '15-JUL-02 11.06.30.123456 +09:00 AM',
                             'DD-MON-RR HH12.MI.SS.FF6 TZH:TZM AM' )
```

```
TO_TIMESTAMP_TZ( '15-JUL-02 11.06.30.123456 +09:00 AM',
                 'DD-MON-RR HH12.MI.SS.FF6 TZH:TZM AM' )
```

区间字面量（Interval Literals）

区间字面量（Interval literals）指定时间的间隔

Interval如下分为两大类

- Year-month INTERVAL values
  - 包含YEAR和MONTH
  - Display string表达：'year-month'
  - 可以写为INTERVAL 'string literal' YEAR[leading precision] TO MONTH或NUMTOYMINTERVAL(num, interval_indicator)形式
- Day-time INTERVAL values
  - 包含DAYHOURMINUTESECOND (包含fractional seconds)
  - Display string表达：'day hour:minute:second.fractional_seconds'
  - 可以写为INTERVAL 'string literal' DAY[leading precision] TO SECOND[fractional seconds precision]或NUMTODSINTERVAL(num, interval_indicator)形式
- Interval value的符号只能在string表达的最前面指定一次

  例：INTERVAL'+3 11:22:33.999999'DAY TO SECOND (O)

  INTERVAL'-3 +11:22:33.999999'DAY TO SECOND (X)

Interval类型目录如下

- INTERVAL YEAR (leading precision)

- INTERVAL MONTH (leading precision)
- INTERVAL YEAR (leading precision) TO MONTH
- INTERVAL DAY (leading precision)
- INTERVAL HOUR (leading precision)
- INTERVAL MINUTE (leading precision)
- INTERVAL SECOND (leading precision[, fractional seconds precision])
- INTERVAL DAY (leading precision) TO HOUR
- INTERVAL DAY (leading precision) TO MINUTE
- INTERVAL DAY (leading precision) TO SECOND (fractional seconds precision)
- INTERVAL HOUR (leading precision) TO MINUTE
- INTERVAL HOUR (leading precision) TO SECOND (fractional seconds precision)
- INTERVAL MINUTE (leading precision) TO SECOND (fractional seconds precision)

前导精度（leading precision）

- 该字段的位数可指定2 ~ 6未指定时默认值为2
- 如果前导字段值超过前导精度则报错

小数秒精度（ fractional seconds precision）

- 是fractional seconds的位数可指定0 ~ 6未指定时默认值为6
- 如果小数秒字段值超过fractional seconds precision时将四舍五入

详细内容参考**INTERVAL表 16-1 INTERVAL * TO * 中第二个之后field的precision与值的范围**

NUMTODSINTERVAL NUMTOYMINTERVAL

Interval字面量的使用示例

以下为使用区间字面量的示例

Interval YEAR

以下为interval YEAR literals的使用示例

示例	说明	字符串表示
INTERVAL'1'YEAR INTERVAL'01-00'YEAR	1 year	+01-00
INTERVAL'100'YEAR	leading precision超过2因此报错	-
INTERVAL'100'YEAR(3)	100 year	+100-00
INTERVAL'+999999'YEAR(6)	999999 year	+999999-00
INTERVAL'-999999'YEAR(6)	-(999999 year)	-999999-00

Interval MONTH

以下为interval MONTH literals的使用示例

示例	说明	字符串表示
INTERVAL'1'MONTH INTERVAL'00-01'MONTH	1 month	+00-01
INTERVAL'100'MONTH	leading precision超过2因此报错	-
INTERVAL'100'MONTH(3)	8 year 4 month	+008-04
INTERVAL'+999999'MONTH(6)	83333 year 3 month	+083333-03
INTERVAL'-999999'MONTH(6)	-(83333 year 3 month)	-083333-03

Interval YEAR TO MONTH

以下为interval YEAR TO MONTH literals的使用示例

示例	说明	字符串表示
INTERVAL'1-06'YEAR TO MONTH	1 year 6 month	+01-06
INTERVAL'1-12'YEAR TO MONTH	月份值超过11因此报错	-
INTERVAL'100-11'YEAR TO MONTH	leading precision超过2因此报错	-
INTERVAL'100-11'YEAR(3) TO MONTH	100 year 11 month	+100-11
INTERVAL'+999999-11'YEAR(6) TO MONTH	999999 year 11 month	+999999-11
INTERVAL'-999999-11'YEAR(6) TO MONTH	-(999999 year 11 month)	-999999-11

Interval DAY

以下为interval DAY literals的使用示例

示例	说明	字符串表示
INTERVAL'1'DAY INTERVAL'01 00:00:00'DAY	1 day	+01 00:00:00
INTERVAL'100'DAY	leading precision超过2因此报错	-
INTERVAL'100'DAY(3)	100 day	+100 00:00:00
INTERVAL'+999999'DAY(6)	999999 day	+999999 00:00:00

示例	说明	字符串表示
INTERVAL'-999999'DAY(6)	-(999999 day)	-999999 00:00:00

Interval HOUR

以下为interval HOUR literals的使用示例

示例	说明	字符串表示
INTERVAL'1'HOUR INTERVAL'00 01:00:00'HOUR	1 hour	+00 01:00:00
INTERVAL'1000'HOUR(3)	超过leading precision3因此报错	-
INTERVAL'1000'HOUR(4)	41 day 16 hour	+0041 16:00:00
INTERVAL'+999999'HOUR(6)	41666 day 15 hour	+041666 15:00:00
INTERVAL'-999999'HOUR(6)	-(41666 day 15 hour)	-041666 15:00:00

Interval MINUTE

以下为interval MINUTE literals的使用示例

示例	说明	字符串表示
INTERVAL'1'MINUTE INTERVAL'00 00:01:00'MINUTE	1 minute	+00 00:01:00
INTERVAL'12345'MINUTE(4)	超过leading precision4因此报错	-

示例	说明	字符串表示
INTERVAL'12345'MINUTE(5)	8 day 13 hour 45 minute	+00008 13:45:00
INTERVAL'+999999'MINUTE(6)	694 day 10 hour 39 minute	+000694 10:39:00
INTERVAL'-999999'MINUTE(6)	-(694 day 10 hour 39 minute)	-000694 10:39:00

Interval SECOND

以下为interval SECOND literals的使用示例

示例	说明	字符串表示
INTERVAL'1'SECOND INTERVAL'00 00:00:01.000000'SECOND	1 second	+00 00:00:01.000000
INTERVAL'100'SECOND	leading precision超过2 因此报错	-
INTERVAL'99.9999999'SECOND INTERVAL'99.9999999'SECOND(2,6)	小数秒（fractional seconds）四舍五入为 100 秒leading precision 超过2因此报错	-
INTERVAL'99.9999999'SECOND(3)	1 minute 40 second	+000 00:01:40.000000
INTERVAL'29.506167'SECOND(2, 2)	29.51 second	+00 00:00:29.51
INTERVAL'999999.999999'SECOND(6,6)	11day 13 hour 46 minute 39.999999 second	+000011 13:46:39.999999

示例	说明	字符串表示
INTERVAL'-999999.999999'SECOND(6,6)	-(11day 13 hour 46 minute 39.999999 second)	-000011 13:46:39.999999

Interval DAY TO HOUR

以下为interval DAY TO HOUR literals的使用示例

示例	说明	字符串表示
INTERVAL'1 23'DAY TO HOUR INTERVAL'01 23:00:00'DAY TO HOUR	1 day 23 hour	+01 23:00:00
INTERVAL'1 24'DAY TO HOUR	小时值为超过23的无效值因此报错	-
INTERVAL'100 23'DAY TO HOUR	leading precision超过2因此报错	-
INTERVAL'100 23'DAY(3) TO HOUR	100 day 23 hour	+100 23:00:00
INTERVAL'+999999 23'DAY(6) TO HOUR	999999 day 23 hour	+999999 23:00:00
INTERVAL'-999999 23'DAY(6) TO HOUR	-(999999 day 23 hour)	-999999 23:00:00
INTERVAL'-999999 +23'DAY(6) TO HOUR	因符号指定错误而报错	-

Interval DAY TO MINUTE

以下为interval DAY TO MINUTE literals的使用示例

示例	说明	字符串表示
INTERVAL'1 23:59'DAY TO MINUTE INTERVAL'01 23:59:00'DAY TO MINUTE	1 day 23 hour 59 second	+01 23:59:00
INTERVAL'1 24:59'DAY TO MINUTE	小时值为超过23的无效值因此报错	-
INTERVAL'1 23:60'DAY TO MINUTE	分钟值为超过59的无效值因此报错	-
INTERVAL'100 23:59'DAY TO MINUTE	leading precision超过2因此报错	-
INTERVAL'100 23:59'DAY(3) TO MINUTE	100 day 23 hour 59 minute	+100 23:59:00
INTERVAL'+999999 23:59'DAY(6) TO MINUTE	999999 day 23 hour 59 minute	+999999 23:59:00
INTERVAL'-999999 23:59'DAY(6) TO MINUTE	-(999999 day 23 hour 59 minute)	-999999 23:59:00

Interval DAY TO SECOND

以下为interval DAY TO SECOND literals的使用示例

示例	说明	字符串表示
INTERVAL '1 23:59:59.999999'DAY TO SECOND	1 day 23 hour 59 minute 59.999999 second	+01 23:59:59.999999

示例	说明	字符串表示
INTERVAL '1 24:59:59.999999'DAY TO SECOND	小时值超过23因此报错	-
INTERVAL '1 23:60:59.999999'DAY TO SECOND	分钟值超过59因此报错	-
INTERVAL '1 23:59:60.999999'DAY TO SECOND	秒值超过60因此报错	-
INTERVAL '99 23:59:59.9999999'DAY TO SECOND	fractional seconds四舍五入后为100dayleading precision 超过2因此报错	-
INTERVAL '99 23:59:59.9999999'DAY(3) TO SECOND	100 day	+100 00:00:00.000000
INTERVAL '1 11:22:33.567890'DAY(2) TO SECOND(2)	1 day 11 hour 22 minute 33.57 second	+01 11:22:33.57
INTERVAL '+999999 23:59:59.999999'DAY(6) TO SECOND(6)	999999 day 23 hour 59 minute 59.999999 hour	+999999 23:59:59.999999
INTERVAL '-999999 23:59:59.999999'DAY(6) TO SECOND(6)	-(999999 day 23 hour 59 minute 59.999999 hour)	-999999 23:59:59.999999

Interval HOUR TO MINUTE

以下为interval HOUR TO MINUTE literals的使用示例

示例	说明	字符串表示
INTERVAL'23:59'HOUR TO MINUTE INTERVAL'00 23:59:00'HOUR TO MINUTE	23 hour 59 minute	+00 23:59:00
INTERVAL'23:60'HOUR TO MINUTE	分钟值超过59因此报错	-
INTERVAL'100:59'HOUR TO MINUTE	leading precision超过2 因此报错	-
INTERVAL'100:59'HOUR(3) TO MINUTE	4 day 4 hour 59 minute	+004 04:59:00
INTERVAL'+999999:59'HOUR(6) TO MINUTE	41666 day 15 hour 59 minute	+041666 15:59:00
INTERVAL'-999999:59'HOUR(6) TO MINUTE	-(41666 day 15 hour 59 minute)	-041666 15:59:00

Interval HOUR TO SECOND

以下为interval HOUR TO SECOND literals 的使用示例

示例	说明	字符串表示
INTERVAL '23:59:59.999999'HOUR TO SECOND INTERVAL '00 23:59:59.999999'HOUR TO SECOND	23 hour 59 minute 59.999999 second	+00 23:59:59.999999
INTERVAL '23:60:59.999999'HOUR TO SECOND	分钟值超过59因此报错	-

示例	说明	字符串表示
INTERVAL '23:59:60.999999'HOUR TO SECOND	秒值超过59因此报错	-
INTERVAL '99:59:59.9999999'HOUR TO SECOND	fractional seconds四舍五入为100 hourleading precision 超过2因此报错	-
INTERVAL '99:59:59.9999999'HOUR(3) TO SECOND	4 day 4 hour	+004 04:00:00.000000
INTERVAL '11:22:29.569'HOUR(3) TO SECOND(1)	11 hour 22 minute 29.6 second	+000 11:22:29.6
INTERVAL '+999999:59:59.999999'HOUR(6) TO SECOND(6)	41666 day 15 hour 59 minute 59.999999 second	+041666 15:59:59.999999
INTERVAL '-999999:59:59.999999'HOUR(6) TO SECOND(6)	-(41666 day 15 hour 59 minute 59.999999 second)	-041666 15:59:59.999999

Interval MINUTE TO SECOND

以下为interval MINUTE TO SECOND literals 的使用示例

示例	说明	字符串表示
INTERVAL '15:23.123456'MINUTE TO SECOND INTERVAL '00 00:15:23.123456'MINUTE TO SECOND	15 minute 23.123456 second	+00 00:15:23.123456
INTERVAL '15:60.123456'MINUTE TO SECOND	秒值超过59因此报错	-
INTERVAL '99:59.999999'MINUTE TO SECOND(2)	fractional seconds四舍五入后为100 minutesleading precision 超过2因此报错	-
INTERVAL '99:59.999999'MINUTE(3) TO SECOND(2)	1 hour 40 minute	+000 01:40:00.00
INTERVAL '+999999:59.999999'MINUTE(6) TO SECOND(6)	694 day 10 hour 39 minute 59.999999 second	+000694 10:39:59.999999
INTERVAL '-999999:59.999999'MINUTE(6) TO SECOND(6)	-(694 day 10 hour 39 minute 59.999999 second)	-000694 10:39:59.999999

空值（Null Value）

空值（null value）指未知的值或未定义的值所有数据类型值均可成为空值

Boolean类型的未知值表示为null value

空值定义为关键字（keyword）且不区分大小写

以下为空值（null value）的表示示例

```
NULL

Null
```

注释（Comments）

单行注释（Single Line Comments）

单行注释指以--或//开头的注释单行注释将其注释符号后面开始到行尾处理为注释

以下为单行注释的使用示例

```
gSQL> SELECT I1, -- I2, I3,

2 I4, I5

3 FROM T1;


I1        I4        I5

--------- --------- ---------
```

```
column i1 column i4 column i5


1 row selected.



gSQL> SELECT I1, // I2, I3,
2 I4, I5
3 FROM T1;

I1        I4        I5
--------- --------- ---------
column i1 column i4 column i5

1 row selected.
```

多行注释（Multiple Line Comments）

多行注释指以/*开头以*/结束的注释多行注释将从/*到*/指定为注释并且可以用多行来表示注释

以下为多行注释的使用示例

```
gSQL> SELECT I1, I2, I3, I4, I5
2 /* 对于TABLE T1
3    输出所有COLUMN */
4 FROM T1;
```

```
I1        I2        I3        I4        I5
--------- --------- --------- --------- ---------
column i1 column i2 column i3 column i4 column i5

1 row selected.
```

提示注释（Hint Comments）

提示注释指以/*+开头并以*/结尾的注释提示注释类似于多行注释但区别在于提示注释在开头符号中有+提示注释的开头符号*与+之间有空格时会处理为多行注释因此需注意

提示注释与其他注释不同仅可在SELECT关键字后面使用提示注释中说明了用户指定SUNDB优化程序的处理方法等内容详细内容参考**SQL Hint**

以下为提示注释的使用示例

```
gSQL> SELECT /*+ FULL(T1) */ * FROM T1;

I1        I2        I3        I4        I5
--------- --------- --------- --------- ---------
column i1 column i2 column i3 column i4 column i5

1 row selected.
```

SQL保留字和关键字（SQL Reserved Words and

Keywords）

SQL保留字（SQL Reserved Words）

SUNDB中有指定为SQL保留字的保留字该SQL保留字除了在其保留字的使用位置外在其他位置使用时必须要带单引号(')另外不推荐使用带单引号的SQL保留字

以下为SUNDB的SQL保留字用*表示的是标准SQL中指定的保留字更多目录相关信息请参照

V$RESERVED_WORDS view

ABSOLUTE

ACCESS

ALL *

ALLOCATE *

ALTER *

AND *

ANY *

ARE *

AS *

ASYMMETRIC *

AT *

AUTHORIZATION *

BEGIN *

BETWEEN *

BOTH *

BY *

CALL *

CASE *

CHECK *

CLOSE *

COLUMN *

COMMENT

COMMIT *

CONNECT *

CONSTRAINT *

CREATE *

CROSS *

CURRENT *

CURRENT_CATALOG *

CURRENT_DATE *

CURRENT_DEFAULT_TRANSFORM_GROUP *

CURRENT_PATH *

CURRENT_ROLE *

CURRENT_ROW *

CURRENT_SCHEMA *

CURRENT_TIME *

CURRENT_TIMESTAMP *

CURRENT_TRANSFORM_GROUP_FOR_TYPE *

CURRENT_USER *

DATABASE

DEALLOCATE *

DECLARE *

DEFAULT *

DELETE *

DEREF *

DESCRIBE *

DETERMINISTIC *

DISCONNECT *

DISTINCT *

DROP *

ELSE *

END *

END_EXEC *

ESCAPE *

EXCEPT *

EXEC *

EXECUTE *

EXISTS *

FALSE *

FETCH *

FILTER *

FIRST

FOR *

FOREIGN *

FREE *

FROM *

FULL *

FUNCTION *

GET *

GLOBAL *

GRANT *

GROUP *

HAVING *

HOLD *

IDENTIFIED

IF

IMMEDIATE

IN *

INDICATOR *

INNER *

INOUT *

INSERT *

INTERSECT *

INTO *

IS *

JOIN *

LAST

LEADING *

LEFT *

LIKE *

LIMIT

LOCAL *

LOCALTIME *

LOCALTIMESTAMP *

MATCH *

MEMBER *

MERGE *

MINUS

NATURAL *

NEW *

NEXT

NOT *

NULL *

OF *

OFFSET *

OLD *

ON *

OPEN *

OR *

ORDER *

OUT *

PREPARE *

PRIMARY *

PRIOR

PROCEDURE *

PROFILE

REF *

REFERENCES *

RELATIVE

RELEASE *

RENAME

RETURN *

RETURNING

RETURNS *

REVOKE *

RIGHT *

ROLLBACK *

ROW *

ROWID

ROWS *

ROW_NUMBER *

SAVEPOINT *

SELECT *

SESSION_USER *

SET *

SOME *

SQL *

SQLEXCEPTION *

SQLSTATE *

SQLWARNING *

START *

SYMMETRIC *

SYNONYM

SYSDATE

SYSTEM *

SYSTEM_USER *

SYSTIME

SYSTIMESTAMP

TABLE *

THEN *

TO *

TRAILING *

TRIGGER *

TRUE *

TRUNCATE *

UNION *

UNIQUE *

UNKNOWN *

UPDATE *

UPPER *

USER *

USING *

VALUES *

VIEW

WHEN *

WHENEVER *

WHERE *

WINDOW *

WITH *

WITHOUT *

SQL关键字（SQL Keywords）

SUNDB的SQL关键字并非保留字但由于是在SUNDB内部使用的关键字因此使用SUNDB的SQL关键字时会影响可读性因此不推荐使用

更多有关SUNDB的SQL关键字目录信息可通过**V$KEYWORDS** view进行查询

兼容性

语法元素SQL标准兼容性如下

Feature ID	说明	是否支持
E021-03	Character literals	O
E131	Null value support (nulls in lieu of values)	O
E161	SQL comments using leading double minus	O
F051-01	DATE data type (including support of DATE literal)	O
F051-02	TIME data type (including support of TIME literal) with fractional seconds precision of at least 0	O
F051-03	TIMESTAMP data type (including support of TIMESTAMP literal) with fractional seconds precision of at least 0 and 6	O
F271	Compound character literals	X

Feature ID	说明	是否支持
F383	Set column not null clause	O
F391	Long identifiers	X
F392	Unicode escapes in identifiers	X
F393	Unicode escapes in literals	X
T023	Compound binary literals	X
T024	Spaces in binary literals	X
T101	Enhanced nullability determination	X
T351	Bracketed comments	X
T591	UNIQUE constraints of possibly null columns	O
X041	Basic table mapping: null absent	X
X042	Basic table mapping: null as nil	X
X051	Advanced table mapping: null absent	X
X052	Advanced table mapping: null as nil	X
X170	XML null handling options	X
X400	Name and identifier mapping	X

Table 1-1 SQL标准兼容性

1.2 数据类型

数字类型

数字类型可根据存储方式和小数的表达方式进行分类

- 根据存储方式的分类
  - 十进制数字类型
    - 以100进制形式存储十进制数字的方法
    - 类型示例: NUMBER, NUMERIC, FLOAT
  - 二进制数字类型
    - 直接存储C语言的数字
    - 类型示例: NATIVE_INTEGER, NATIVE_DOUBLE
- 根据小数的表达方式进行分类
  - 定点数(exact numeric)
    - 固定scale的数字类型
    - 类型示例: NUMERIC(precision, scale), NATIVE_INTEGER
  - 浮点数 (approximate numeric)
    - 未指定scale的数字类型
    - 类型示例: FLOAT(precision), NATIVE_DOUBLE

十进制数字类型

表示有效数字精确度的precision及表示小数点范围的scale以十进制（decimal）为基础

十进制定点数类型

十进制定点数是由SQL定义的类型

类型	十进制精度	十进制范围	参考
NUMBER(p)	p	0	**NUMBER**
NUMBER(p, s)	p	s	**NUMBER**
NUMERIC(p)	p	0	**NUMERIC**
NUMERIC(p, s)	p	s	**NUMERIC**
DECIMAL(p)	p	0	**NUMERIC**类型别名
DECIMAL(p, s）	p	s	**NUMERIC**类型别名
DEC(p)	p	0	**NUMERIC**类型别名
DEC(p, s)	p	s	**NUMERIC**类型别名
SMALLINT	5	0	**NUMBER**类型别名
INTEGER	10	0	**NUMBER**类型别名
BIGINT	19	0	**NUMBER**类型别名
INT2	5	0	**NUMBER**类型别名
INT4	10	0	**NUMBER**类型别名
INT8	19	0	**NUMBER**类型别名

Table 1-2 十进制定点数类型

十进制浮点数类型

十进制浮点数是SQL定义的类型

类型	十进制精度	十进制范围	参考
NUMBER	38	N/A	**NUMBER**
FLOAT(p)	ceil($\log_{10}2^{p}$)	N/A	**FLOAT**
REAL	ceil($\log_{10} 2^{24}$) = 8	N/A	**FLOAT**类型别名
DOUBLE	ceil($\log_{10} 2^{53}$) = 16	N/A	**FLOAT**类型别名
FLOAT4	ceil($\log_{10} 2^{24}$) = 8	N/A	**FLOAT**类型别名
FLOAT8	ceil($\log_{10} 2^{53}$) = 16	N/A	**FLOAT**类型别名

Table 1-3 十进制浮点数类型

二进制数字类型

表示有效数字精确度的precision及表示小数点范围的scale以二进制（binary）为基础

二进制定点数类型

二进制定点数参考C语言的signed integer的类型

除了用于表示sign bit的1bit外其他bit均用于表示精度（precision）表示scale时不使用bit

类型	二进制精度	二进制范围	参考
NATIVE_SMALLINT	15	0	**NATIVE_SMALLINT**
NATIVE_INTEGER	31	0	**NATIVE_INTEGER**
NATIVE_BIGINT	63	0	**NATIVE_BIGINT**

Table 1-4 二进制定点数类型

二进制浮点数类型

二进制浮点数参考C语言中的float与double类型

使用1bit表示sign bit其余bit用于表示precision与scale

类型	二进制精度	二进制范围	参考
NATIVE_REAL	23	8	**NATIVE_REAL**
NATIVE_DOUBLE	52	11	**NATIVE_DOUBLE**

Table 1-5 二进制浮点数类型

> Note:
>
> 二进制浮点数的precision与scale可随着操作系统与编译环境发生变动

CHARACTER STRING 类型

CHARACTER STRING类型可根据是否为可变长度字符串与字符串的最大长度进行分类

- 根据是否为可变长度字符串进行分类
  - 固定长度字符串
    - **CHARACTER**
  - 可变长度字符串
    - **CHARACTER VARYING**, **CHARACTER LONG VARYING**
- 根据字符串最大长度进行分类
  - 2000 [characters或字节]
    - **CHARACTER**
  - 4000 [characters或字节]
    - **CHARACTER VARYING**
  - 100 Mega Bytes
    - **CHARACTER LONG VARYING**

二进制字符串（BINARY STRING）类型

二进制字符串可根据是否为可变长度二进制字符串与二进制字符串的最大长度进行分类

- 根据是否为可变长度二进制字符串进行分类
  - 固定长度二进制字符串
    - **BINARY**

    - 可变长度二进制字符串
        - **BINARY VARYING**, **BINARY LONG VARYING**
- 根据二进制字符串的最大长度进行分类
    - 2000
        - **BINARY**
    - 4000
        - **BINARY VARYING**
    - 100 mega bytes
        - **BINARY LONG VARYING**

日期/ 时间类型

日期/时间类型以各类型的表示方法指定年月日时分秒时区偏移

日期/时间类型包括**DATETIMETIMESTAMP**类型

INTERVAL类型

INTERVAL类型指定时间的间隔

以各类型的表示方式指定年月日时分秒的时间间隔

INTERVAL类型根据值的表示范围分为YEAR TO MONTH系列与DAY TO SECOND系列

BOOLEAN类型

Boolean类型存储TRUEFALSEUNKNOWN的truth值其中UNKNOWN值表示空（null）值

用作条件（condition）的所有表达式（expression）均返回boolean值并且定义为boolean类型的column或值均可用作条件

以下为可存储为boolean类型的literal

- TRUE值
  - Keyword：TRUE
  - 字符： 't', 'true' , 'y', 'yes' , 'on' ,'1'
- FALSE值
  - Keyword：FALSE
  - 字符： 'f', 'false', 'n', 'no', 'off', '0'
- UNKNOWN值
  - Keyword：UNKNOWN, NULL

详细内容参考**BOOLEAN**

ROWID类型

存储于数据库的所有记录都有唯一的位置信息为了区分各个记录使用记录标识符（ROWID）

ROWID类型用于存储和管理记录标识符（ROWID)

通过使用ROWID pseudo column的查询可获得记录标识符（ROWID）

详细内容参考**ROWID**

类型间比较（type comparison）

两个类型间的比较以一个类型为基准来进行比较对象类型与代表类型不同时通过类型变换执行比较

类型间用于比较的代表类型定义两个类型间用于对比的代表类型

以下表格说明为了按照各个代表类型进行比较而转换对象类型

- **用于在VC进行比较的类型转换**
- **用于在LC进行比较的类型转换**
- **用于在VB进行比较的类型转换**
- **用于在LB进行比较的类型转换**
- **用于在NB进行比较的类型转换**
- **用于在ND进行比较的类型转换**
- **用于在NU进行比较的类型转换**
- **用于在DA进行比较的类型转换**
- **用于在TI进行比较的类型转换**
- **用于在TZ进行比较的类型转换**
- **用于在TS进行比较的类型转换**
- **用于在SZ进行比较的类型转换**
- **用于在YM进行比较的类型转换**
- **用于在DS进行比较的类型转换**
- **用于在BO进行比较的类型转换**
- **用于在RI进行比较的类型转换**

> Tip:

以下是用于类型间比较的缩写

* "VC" : CHARACTER VARYING

* "LC" : CHARACTER LONG VARYING

* "VB" : BINARY VARYING

* "LB" : BINARY LONG VARYING

* "NB" : NATIVE_BIGINT

* "ND" : NATIVE_DOUBLE

* "NU" : NUMBER

* "DA" : DATE

* "TI" : TIME

* "TZ" : TIME WITH TIMEZONE

* "TS" : TIMESTAMP

* "SZ" : TIMESTAMP WITH TIMEZONE

* "YM" : INTERVAL YEAR TO MONTH

* "DS" : INTERVAL DAY TO SECOND

* "BO" : BOOLEAN

* "RI" : ROWID

Note:

类型间对比表中built-in数据类型用引号("") 表示缩写词

* "CHAR": CHARACTER

* "VARCHAR": CHARACTER VARYING

* "LONG VARCHAR": CHARACTER LONG VARYING

* "VARBINARY": BINARY VARYING

* "LONG VARBINARY": BINARY LONG VARYING

* "TIME_TZ": TIME WITH TIMEZONE

* "TIMESTAMP_TZ": TIMESTAMP WITH TIMEZONE

* "INTERVAL_YM": INTERVAL YEAR TO MONTH

* "INTERVAL_DS": INTERVAL DAY TO SECOND

Data type	CHAR	VARCHAR	LONG VARCHAR	BINARY	VARBINARY	LONG VARBINARY	NATIVE SMALLINT	NATIVE INTEGER	NATIVE BIGINT	NATIVE REAL	NATIVE DOUBLE	NUMBER	NUMERIC	FLOAT	DATE	TIME
CHAR	VC	VC	LC				NU	NU	NU	NU	ND	NU	NU	NU	DA	TI
VARCHAR	VC	VC	LC				NU	NU	NU	NU	ND	NU	NU	NU	DA	TI

Data type	CHAR	VARCHAR	LONG VARCHAR	BINARY	VARBINARY	LONG VARBINARY	NATIVE_SMALLINT	NATIVE_INTEGER	NATIVE_BIGINT	NATIVE_REAL	NATIVE_DOUBLE	NUMBER	NUMERIC	FLOAT	DATE	TIME
LONG VARCHAR	LC	LC	LC				NU	NU	NU	NU	ND	NU	NU	NU	DA	TI
BINARY				VB	VB	LB										
VARBINARY				VB	VB	LB										
LONG VARBINARY				LB	LB	LB										
NATIVE_SMALLINT	NU	NU	NU				NB	NB	NB	ND	ND	NU	NU	NU		
NATIVE_INTEGER	NU	NU	NU				NB	NB	NB	ND	ND	NU	NU	NU		
NATIVE_BIGINT	NU	NU	NU				NB	NB	NB	ND	ND	NU	NU	NU		

Data type	CHAR	VARCHAR	LONG VARCHAR	BINARY	VARBINARY	LONG VARBINARY	NATIVE SMALLINT	NATIVE INTEGER	NATIVE BIGINT	NATIVE REAL	NATIVE DOUBLE	NUMBER	NUMERIC	FLOAT	DATE	TIME
NATIVE_REAL	NU	NU	NU				ND	ND	ND	ND	ND	NU	NU	NU		
NATIVE_DOUBLE	ND	ND	ND				ND	ND	ND	ND	ND	ND	ND	ND		
NUMBER	NU	NU	NU				NU	NU	NU	NU	ND	NU	NU	NU		
NUMERIC	NU	NU	NU				NU	NU	NU	NU	ND	NU	NU	NU		
FLOAT	NU	NU	NU				NU	NU	NU	NU	ND	NU	NU	NU		
DATE	DA	DA	DA												DA	
TIME	TI	TI	TI													TI

Data type	CHAR	VARCHAR	LONG VARCHAR	BINARY	VARBINARY	LONG VARBINARY	NATIVE SMALLINT	NATIVE INTEGER	NATIVE BIGINT	NATIVE REAL	NATIVE DOUBLE	NUMBER	NUMERIC	FLOAT	DATE	TIME
TIME_TZ	TZ	TZ	TZ													TZ
TIMESTAMP	TS	TS	TS												TS	
TIMESTAMP_TZ	SZ	SZ	SZ												SZ	
INTERVAL_YM	YM	YM	YM				YM	YM	YM			YM	YM	YM		
INTERVAL_DS	DS	DS	DS				DS	DS	DS			DS	DS	DS		
BOOLEAN	BO	BO	BO													
ROWID	RI	RI	RI													

Table 1-6 类型间用于比较的代表类型

原本类型	转换类型
CHAR	CHAR（没有转换）
VARCHAR	VARCHAR（没有转换）

Table 1-7 用于在VC进行比较的类型转换

原本类型	转换类型
CHAR	CHAR（没有转换）
VARCHAR	VARCHAR（没有转换）
LONG VARCHAR	LONG VARCHAR（没有转换）

Table 1-8 用于在LC进行比较的类型转换

原本类型	转换类型
BINARY	BINARY（没有转换）
VARBINARY	VARBINARY（没有转换）

Table 1-9 用于在VB进行比较的类型转换

原本类型	转换类型
BINARY	BINARY（没有转换）
VARBINARY	VARBINARY（没有转换）

原本类型	转换类型
LONG VARBINARY	LONG VARBINARY（没有转换）

Table 1-10 用于在LB进行比较的类型转换

原本类型	转换类型
CHAR	NATIVE_BIGINT
VARCHAR	NATIVE_BIGINT
LONG VARCHAR	NATIVE_BIGINT
NATIVE_SMALLINT	NATIVE_SMALLINT（没有转换）
NATIVE_INTEGER	NATIVE_INTEGER（没有转换）
NATIVE_BIGINT	NATIVE_BIGINT（没有转换）

Table 1-11 用于在NB进行比较的类型转换

原本类型	转换类型
CHAR	NATIVE_DOUBLE
VARCHAR	NATIVE_DOUBLE
LONG VARCHAR	NATIVE_DOUBLE
NATIVE_SMALLINT	NATIVE_SMALLINT（没有转换）
NATIVE_INTEGER	NATIVE_INTEGER（没有转换）
NATIVE_BIGINT	NATIVE_BIGINT（没有转换）

原本类型	转换类型
NATIVE_REAL	NATIVE_REAL（没有转换）
NATIVE_DOUBLE	NATIVE_DOUBLE（没有转换）
NUMBER	NUMBER（没有转换）
NUMERIC	NUMERIC（没有转换）
FLOAT	FLOAT（没有转换）

Table 1-12 用于在ND进行比较的类型转换

原本类型	转换类型
CHAR	NUMBER
VARCHAR	NUMBER
LONG VARCHAR	NUMBER
NATIVE_SMALLINT	NATIVE_SMALLINT（没有转换）
NATIVE_INTEGER	NATIVE_INTEGER（没有转换）
NATIVE_BIGINT	NATIVE_BIGINT（没有转换）
NATIVE_REAL	NATIVE_REAL（没有转换）
NATIVE_DOUBLE	NATIVE_DOUBLE（没有转换）
NUMBER	NUMBER（没有转换）
NUMERIC	NUMERIC（没有转换）

原本类型	转换类型
FLOAT	FLOAT（没有转换）

Table 1-13 用于在NU进行比较的类型转换

原本类型	转换类型
CHAR	DATE
VARCHAR	DATE
LONG VARCHAR	DATE
DATE	DATE（没有转换）

Table 1-14 用于在DA进行比较的类型转换

原本类型	转换类型
CHAR	TIME
VARCHAR	TIME
LONG VARCHAR	TIME
TIME	TIME（没有转换）

Table 1-15 用于在TI进行比较的类型转换

原本类型	转换类型
CHAR	TIME_TZ

原本类型	转换类型
VARCHAR	TIME_TZ
LONG VARCHAR	TIME_TZ
TIME	TIME_TZ
TIME_TZ	TIME_TZ（没有转换）

Table 1-16 用于在TZ进行比较的类型转换

原本类型	转换类型
CHAR	TIMESTAMP
VARCHAR	TIMESTAMP
LONG VARCHAR	TIMESTAMP
DATE	DATE（没有转换）
TIMESTAMP	TIMESTAMP（没有转换）

Table 1-17 用于在TS进行比较的类型转换

原本类型	转换类型
CHAR	TIMESTAMP_TZ
VARCHAR	TIMESTAMP_TZ
LONG VARCHAR	TIMESTAMP_TZ
DATE	TIMESTAMP_TZ

原本类型	转换类型
TIMESTAMP	TIMESTAMP_TZ
TIMESTAMP_TZ	TIMESTAMP_TZ（没有转换）

Table 1-18 用于在SZ进行比较的类型转换

原本类型	转换类型
CHAR	INTERVAL_YM
VARCHAR	INTERVAL_YM
LONG VARCHAR	INTERVAL_YM
NATIVE_SMALLINT	INTERVAL_YM
NATIVE_INTEGER	INTERVAL_YM
NATIVE_BIGINT	INTERVAL_YM
NUMBER	INTERVAL_YM
NUMERIC	INTERVAL_YM
FLOAT	INTERVAL_YM
INTERVAL_YM	INTERVAL_YM（没有转换）

Table 1-19 用于在YM进行比较的类型转换

原本类型	转换类型
CHAR	INTERVAL_DS

原本类型	转换类型
VARCHAR	INTERVAL_DS
LONG VARCHAR	INTERVAL_DS
NATIVE_SMALLINT	INTERVAL_DS
NATIVE_INTEGER	INTERVAL_DS
NATIVE_BIGINT	INTERVAL_DS
NUMBER	INTERVAL_DS
NUMERIC	INTERVAL_DS
FLOAT	INTERVAL_DS
INTERVAL_DS	INTERVAL_DS（没有转换）

Table 1-20 用于在DS进行比较的类型转换

原本类型	转换类型
CHAR	BOOLEAN
VARCHAR	BOOLEAN
LONG VARCHAR	BOOLEAN
BOOLEAN	BOOLEAN（没有转换）

Table 1-21 用于在BO进行比较的类型转换

原本类型	转换类型
CHAR	ROWID
VARCHAR	ROWID
LONG VARCHAR	ROWID
ROWID	ROWID（没有转换）

Table 1-22 用于在RI进行比较的类型转换

类型间转换（type conversion）

类型间转换分为内部转换（Implicit Type Conversion）与外部转换（Explicit Type Conversion）

- 内部转换在选择（select） 插入（insert） 删除（delete）更新（update）的表达式，运算符函数，条件中产生
- 外部转换通过CAST operator实现

类型间转换（type conversion）说明是否可以从一个数据类型转换到其他数据类型

> Note:
>
> 在类型间转换表中built-in数据类型用引号("") 表示缩写的字符串
>
> * "CHAR": CHARACTER
>
> * "VARCHAR": CHARACTER VARYING
>
> * "LONG VARCHAR": CHARACTER LONG VARYING

* "VARBINARY": BINARY VARYING

* "LONG VARBINARY": BINARY LONG VARYING

* "TIME_TZ": TIME WITH TIMEZONE

* "TIMESTAMP_TZ": TIMESTAMP WITH TIMEZONE

* "INTERVAL_YM": INTERVAL YEAR TO MONTH

* "INTERVAL_DS": INTERVAL DAY TO SECOND

Data type	CHAR	VARCHAR	LONG VARCHAR	BINARY	VARBINARY	LONG VARBINARY	NATIVE INTEGER	NATIVE SMALLINT	NATIVE BIGINT	NATIVE REAL	NATIVE DOUBLE	NUMBER	NUMERIC	FLOAT	DATE	TIME	TIME WITH TIMEZONE	TIMESTAMP
CHAR	0	0	0				0	0	0	0	0	0	0	0	0	0	0	0
VARCHAR	0	0	0				0	0	0	0	0	0	0	0	0	0	0	0

Data type	CHAR	VARCHAR	LONG VARCHAR	BINARY	VARBINARY	LONG VARBINARY	NATIVE_SMALLINT	NATIVE_INTEGER	NATIVE_BIGINT	NATIVE_REAL	NATIVE_DOUBLE	NUMBER	NUMERIC	FLOAT	DATE	TIME	TIMEZ	TIMESTAMP
LONG VARCHAR	O	O	O				O	O	O	O	O	O	O	O	O	O	O	O
BINARY				O	O	O												
VARBINARY				O	O	O												
LONG VARBINARY				O	O	O												
NATIVE_SMALLINT	O	O	O				O	O	O	O	O	O	O	O				
NATIVE_INTEGER	O	O	O				O	O	O	O	O	O	O	O				
NATIVE_BIGINT	O	O	O				O	O	O	O	O	O	O	O				

Data type	CHAR	VARCHAR	LONG	BINARY	VARBINARY	LONG VARBINARY	NATIVE SMALLINT	NATIVE INTEGER	NATIVE BIGINT	NATIVE REAL	NATIVE DOUBLE	NUMBER	NUMERIC	FLOAT	DATE	TIME	TIMEZ	TIMESTAMP
NATIVE_REAL	O	O	O				O	O	O	O	O	O	O	O				
NATIVE_DOUBLE	O	O	O				O	O	O	O	O	O	O	O				
NUMBER	O	O	O				O	O	O	O	O	O	O	O				
NUMERIC	O	O	O				O	O	O	O	O	O	O	O				
FLOAT	O	O	O				O	O	O	O	O	O	O	O				
DATE	O	O	O												O			O
TIME	O	O	O													O	O	

Data type	CHAR	VARCHAR	LONG	BINARY	VARBINARY	LONG VARBINARY	NATIVE SMALLINT	NATIVE INTEGER	NATIVE BIGINT	NATIVE REAL	NATIVE DOUBLE	NUMBER	NUMERIC	FLOAT	DATE	TIME	TIME TZ	TIMESTAMP
TIME_TZ	O	O	O													O	O	
TIMESTAMP	O	O	O												O	O		O
TIMESTAMP_TZ	O	O	O												O	O	O	O
INTERVAL_YM	O	O	O				O	O	O			O	O	O				
INTERVAL_DS	O	O	O				O	O	O			O	O	O				
BOOLEAN	O	O	O															
ROWID	O	O	O															

Table 1-23 类型间转换

- 转换为CHARACTER类型
  - 原本类型为CHARACTER类型的情况
    - 原本类型的精确度大于CHARACTER类型的精确度时报错
    - 原本类型的精确度等于CHARACTER类型的精确度时不更新字符串
    - 原本类型的精确度小于CHARACTER类型的精确度时字符串后面追加与精确度差异相同数量的空格字符
  - 原本类型为CHARACTER VARYING类型或CHARACTER LONG VARYING类型的情况
    - 原本类型的字符串长度大于CHARACTER类型的精确度时报错
    - 原本类型的字符串长度等于CHARACTER类型的精确度时不更新字符串
    - 原本类型的字符串长度小于CHARACTER类型的精确度时在字符串后面追加与精确度差异相同数量的空格字符
  - 原本类型为数字型类型日期/时间类型时间段（interval）类型BOOLEAN类型ROWID类型的情况
    - 原本类型的转换字符串长度大于CHARACTER类型的精确度时报错
    - 原本类型的转换字符串长度等于CHARACTER类型的精确度时不更新字符串
    - 原本类型的转换字符串长度小于CHARACTER类型的精确度时在字符串后面追加与精确度差异相同数量的空格字符
- 转换为CHARACTER VARYING类型
  - 原本类型为CHARACTER类型的情况
    - 原本类型的精确度大于CHARACTER VARYING类型的精确度时报错
    - 原本类型的精确度小于或等于CHARACTER VARYING类型的精确度时不更新字符串
  - 原本类型为CHARACTER VARYING类型或CHARACTER LONG VARYING类型的情况

        - 原本类型的字符串长度大于CHARACTER VARYING类型的精确度时报错
        - 原本类型的字符串长度小于或等于CHARACTER VARYING类型的精确度时不更新字符串
    - 原本类型为数字型类型日期/时间类型时间段类型BOOLEAN类型ROWID类型的情况
        - 原本类型的转换字符串长度大于CHARACTER VARYING类型的精确度时报错
        - 原本类型的转换字符串长度小于或等于CHARACTER VARYING类型的精确度时不更新字符串
- 转换为CHARACTER LONG VARYING类型
    - 原本类型为CHARACTER STRING类型的情况
        - 原本类型的字符串无更新
    - 原本类型为数字型类型日期/时间类型时间段类型BOOLEAN类型ROWID类型的情况
        - 原本类型的转换字符串无更新
- 转换为BINARY类型
    - 原本类型为BINARY类型的情况
        - 原本类型的精确度大于BINARY类型精确度时报错
        - 原本类型的精确度等于BINARY类型精确度时二进制字符串无更新
        - 原本类型的精确度小于BINARY类型精确度时二进制字符串后面追加与精确度差异相同数量的X'00'字符
    - 原本类型为 BINARY VARYING类型或BINARY LONG VARYING类型的情况
        - 原本类型的二进制字符串的长度大于BINARY类型的精确度时报错
        - 原本类型的二进制字符串的长度等于BINARY类型的精确度时二进制字符串无更新
        - 原本类型的二进制字符串的长度小于BINARY类型的精确度时二进制字符串后面追加与精确度差异相同数量的X'00'字符
- 转换为BINARY VARYING类型
    - 原本类型为BINARY类型的情况

        - 原本类型的精确度大于BINARY VARYING类型的精确度时报错
        - 原本类型的精确度小于或等于BINARY VARYING类型的精确度时二进制字符串无更新
    - 原本类型为 BINARY VARYING类型或BINARY LONG VARYING类型的情况
        - 原本类型的二进制字符串的长度大于BINARY VARYING类型的精确度时报错
        - 原本类型的二进制字符串的长度小于或等于BINARY VARYING类型的精确度时二进制字符串无更新
- 转换为BINARY LONG VARYING类型
    - 原本类型为BINARY STRING类型时原本类型的二进制字符串无更新
- 转换为数字型类型
    - 原本类型为CHARACTER STRING类型的情况
        - 字符串不符合数字格式时报错
        - 转换类型中定义的精确度与范围（scale）可能会引起溢出（overflow）或 舍入（ rounding）
    - 原本类型为数字型类型的情况
        - 转换类型中定义的精确度与范围（scale）可能会引起溢出（overflow）或 舍入（ rounding）
    - 原本类型为时间段类型的情况
        - 原本类型为单个字段(YEAR, MONTH, DAY, HOUR, MINUTE, SECOND)时才能转换为数字型类型
        - 转换类型中定义的精确度与范围（scale）可能会引起溢出（overflow）或 舍入（ rounding）
- 转换为DATE类型
    - 原本类型为CHARACTER STRING类型的情况
        - 字符串不符合DATE格式时报错

        - 转换类型中定义的值的范围可能会引起溢出（overflow）
    - 原本类型为DATE类型或TIMESTAMP类型的情况
        - 不报错
    - 原本类型为TIMESTAMP WITH TIME ZONE类型的情况
        - 考虑到时区偏移（time zone offset）将其转换为DATE类型的值
- 转换为TIME类型
    - 原本类型为CHARACTER STRING类型的情况
        - 字符串不符合TIME格式时报错
        - 转换类型中定义的值的范围可能会引起舍入（rounding）
    - 原本类型为TIME类型或TIMESTAMP类型的情况
        - 不报错
    - 原本类型为TIME WITH TIME ZONE类型或者TIMESTAMP WITH TIME ZONE类型的情况
        - 考虑到时区偏移（time zone offset）将其转换为TIME类型的值
- 转换为TIME WITH TIME ZONE类型
    - 原本类型为CHARACTER STRING类型的情况
        - 字符串不符合TIME WITH TIME ZONE格式时报错
        - 转换类型中定义的值的范围可能会引起舍入（rounding）
    - 原本类型为TIME类型TIME WITH TIME ZONE类型或者TIMESTAMP WITH TIME ZONE类型的情况
        - 考虑到时区偏移（time zone offset）将其转换为TIME WITH TIME ZONE类型的值
- 转换为TIMESTAMP类型
    - 原本类型为CHARACTER STRING类型的情况
        - 字符串不符合TIMESTAMP格式时报错
        - 转换类型中定义的值的范围可能会引起溢出（overflow）或舍入（rounding）

    - 原本类型为DATE类型或者TIMESTAMP类型的情况
        - 不报错
    - 原本类型为TIMESTAMP WITH TIME ZONE类型的情况
        - 考虑到时区偏移（time zone offset）将其转换为TIMESTAMP类型的值
- 转换为TIMESTAMP WITH TIME ZONE类型
    - 原本类型为CHARACTER STRING类型的情况
        - 字符串不符合TIMESTAMP WITH TIME ZONE格式时报错
        - 转换类型中定义的值的范围可能会引起溢出（overflow）或舍入（rounding）
    - 原本类型为DATE类型TIMESTAMP类型或者TIMESTAMP WITH TIME ZONE类型的情况
        - 考虑到时区偏移（time zone offset）将其转换为TIMESTAMP WITH TIME ZONE类型的值
- 转换为INTERVAL YEAR TO MONTH系列类型
    - 原本类型为CHARACTER STRING类型的情况
        - 字符串不符合year-month区间字面量格式时报错
        - 详细内容参考**区间字面量（Interval Literals）**
        - 转换类型中定义的精确度可能会引起溢出（overflow）
    - 原本类型为NATIVE_SMALLINT, NATIVE_INTEGER, NATIVE_BIGINT, NUMBER, NUMERIC或FLOAT类型的情况
        - 转换类型应为单个字段(YEAR, MONTH)
        - 转换类型中定义的精确度可能会引起溢出（overflow）
    - 原本类型为INTERVAL YEAR TO MONTH系列类型的情况
        - 转换类型中定义的精确度可能会引起溢出（overflow）
- 转换为INTERVAL DAY TO SECOND系列类型
    - 原本类型为CHARACTER STRING类型的情况
        - 字符串不符合day-time区间字面量格式时报错

        - 详细内容参考区间字面量（Interval Literals）
        - 转换类型中定义的精确度可能会引起溢出（overflow）或舍入（rounding）
    - 原本类型为NATIVE_SMALLINT, NATIVE_INTEGER, NATIVE_BIGINT, NUMBER, NUMERIC或FLOAT类型的情况
        - 转换类型应为单个字段(DAY, HOUR, MINUTE, SECOND)
        - 转换类型中定义的精确度可能会引起溢出（overflow）或舍入（rounding）
    - 原本类型为INTERVAL DAY TO SECOND系列类型的情况
        - 转换类型中定义的精确度可能会引起溢出（overflow）或舍入（rounding）
- 转换为BOOLEAN类型
    - 原本类型为CHARACTER STRING类型的情况
        - 不区分大小写当字符串为“TRUE”或“FALSE”时可转换（即使前后包含空格也可以转换）
    - 原本类型为BOOLEAN类型的情况
        - 不报错
- 转换为ROWID类型
    - 原本类型为CHARACTER STRING类型的情况
        - 字符串不符合ROWID类型的格式时报错
    - 原本类型为ROWID类型的情况
        - 不报错

类型间组合（type combination）

需要类型间组合的情况

CASE运算符与集合运算符(**set operator**)将多个表达式作为运算结果

如下所示各表达式为不同类型时需要决定结果类型

- CASE运算符

```
SELECT CASE expr WHEN expr THEN char(3)
                 WHEN expr THEN char(5)
                 ELSE char(1)
       END
  FROM t1;
```

- 集合运算符

```
SELECT float_column
  FROM t1
UNION ALL
SELECT number_precision_column
  FROM t2
UNION ALL
SELECT native_integer_column
  FROM t3;
```

通过相应规则决定类型间组合的结果类型其规则应用示例如下

- 结果类型组合（Result Type Combination）规则
  - 集合运算符(**set operator**)
  - CASE运算符：
    - **CASE表达式（CASE Expression）**
    - **COALESCE**
    - **NULLIF**

结果类型组合规则

各表达式的数据类型应为可组合的相同系列的类型

- 应用结果类型组合规则的示例
  - 集合运算符（**set operator**)
  - CASE运算符
    - **CASE表达式（CASE Expression）**
    - **COALESCE**
    - **NULLIF**

通过下表说明根据结果类型组合规则决定的结果类型

> Tip:
>
> 以下缩写用于说明结果类型组合规则
>
> * "VC" : CHARACTER VARYING
>
> * "LC" : CHARACTER LONG VARYING

* "VB" : BINARY VARYING

* "LB" : BINARY LONG VARYING

* "NS" : NATIVE_SMALLINT

* "NI" : NATIVE_INTEGER

* "NB" : NATIVE_BIGINT

* "NR" : NATIVE_REAL

* "ND" : NATIVE_DOUBLE

* "FL" : FLOAT

* "NU" : NUMBER

* "DA" : DATE

* "TI" : TIME

* "TZ" : TIME WITH TIMEZONE

* "TS" : TIMESTAMP

* "SZ" : TIMESTAMP WITH TIMEZONE

* "YM" : INTERVAL YEAR TO MONTH

* "DS" : INTERVAL DAY TO SECOND

* "BO" : BOOLEAN

* "RI" : ROWID

Note:

在根据结果类型组合规则的结果类型表中built-in数据类型的缩写字符串使用引号（“”）表示

* "CHAR": CHARACTER

* "VARCHAR": CHARACTER VARYING

* "LONG VARCHAR": CHARACTER LONG VARYING

* "BINARY": BINARY

* "VARBINARY": BINARY VARYING

* "LONG VARBINARY": BINARY LONG VARYING

* "TIME_TZ": TIME WITH TIMEZONE

* "TIMESTAMP_TZ": TIMESTAMP WITH TIMEZONE

* "INTERVAL_YM": INTERVAL YEAR TO MONTH

* "INTERVAL_DS": INTERVAL DAY TO SECOND

Data type	CHAR	VARCHAR	LONG VARCHAR	BINARY	VARBINARY	LONG VARBINARY	NATIVE_SMALLINT	NATIVE_INTEGER	NATIVE_BIGINT	NATIVE_REAL	NATIVE_DOUBLE	NUMBER	NUMERIC	FLOAT	DATE	TIME
CHAR	VC	VC														
VARCHAR	VC	VC														
LONG VARCHAR			LC													
BINARY				VB	VB											
VARBINARY				VB	VB											
LONG VARBINARY						LB										
NATIVE_SMALLINT							NS	NI	NB	ND	ND	NU	NU	NU		

Data type	CHAR	VARCHAR	LONG VARCHAR	BINARY	VARBINARY	LONG VARBINARY	NATIVE SMALLINT	NATIVE INTEGER	NATIVE BIGINT	NATIVE REAL	NATIVE DOUBLE	NUMBER	NUMERIC	FLOAT	DATE	TIME
NATIVE_INTEGER							NI	NI	NB	ND	ND	NU	NU	NU		
NATIVE_BIGINT							NB	NB	NB	ND	ND	NU	NU	NU		
NATIVE_REAL							ND	ND	ND	NR	ND	ND	ND	ND		
NATIVE_DOUBLE							ND	ND	ND	ND	ND	ND	ND	ND		
NUMBER							NU	NU	NU	ND	ND	NU	NU	NU		
NUMERIC							NU	NU	NU	ND	ND	NU	NU	NU		

Data type	CHAR	VARCHAR	LONG VARCHAR	BINARY	VARBINARY	LONG VARBINARY	NATIVE SMALLINT	NATIVE INTEGER	NATIVE BIGINT	NATIVE REAL	NATIVE DOUBLE	NUMBER	NUMERIC	FLOAT	DATE	TIME
FLOAT							NU	NU	NU	ND	ND	NU	NU	FL		
DATE															DA	
TIME																TI
TIME_TZ																TZ
TIMESTAMP															TS	
TIMESTAMP_TZ															SZ	

Data type	CHAR	VARCHAR	LONG	BINARY	VARBINARY	LONG RAW	NATIVE_SMALLINT	NATIVE_INTEGER	NATIVE_BIGINT	NATIVE_REAL	NATIVE_DOUBLE	NUMBER	NUMERIC	FLOAT	DATE	TIME
INTERVAL_YM																
INTERVAL_DS																
BOOLEAN																
ROWID																

Table 1-24 根据结果类型组合规则决定的结果类型

- 全部为CHAR类型时的结果类型
    - 长度不同时为VARCHAR类型

- 长度相同时为CHAR类型

- 全部为BINARY类型时的结果类型
  - 长度不同时为VARBINARY类型
  - 长度相同时为BINARY类型

- INTERVAL YEAR TO MONTH类型的结果类型
  - YEAR和MONTH共存时为INTERVAL YEAR TO MONTH类型
  - 只有YEAR时为INTERVAL YEAR类型
  - 只有MONTH时为INTERVAL MONTH类型

- INTERVAL DAY TO SECOND类型的结果类型
  - 起始字段(Start field)为各对象表达式中范围最大的起始字段
  - 结束字段(End field)为各对象表达式中范围最小的结束字段
  - eg) {INTERVAL DAY, INTERVAL HOUR} → INTERVAL DAY TO HOUR

- 决定结果类型的精度（precision）和小数位数（ scale）
  - CHARACTER STRING类型
    - 对象表达式中的最长字符长度
  - BINARY STRING类型
    - 对象表达式中的最长字符长度
  - 数字型
    - 指定为可容纳对应类型最大值的范围
  - TIME/TIMESTAMP类型
    - 对象表达式中最大小数秒（fractional seconds）精确度
  - INTERVAL YEAR TO MONTH
    - 对象表达式中最大前导精确度（leading precision）
  - INTERVAL DAY TO SECOND
    - 前导精确度（leading precision）：是起始字段的最大前导精确度（ leading

precision）

  - fractional seconds精确度是最大fractional seconds精确度

- 详细内容请参考**类型间比较（type comparison）**

兼容性

数据类型的SQL标准兼容性如下

Feature ID	说明	是否支持
B033	Untyped SQL-invoked function arguments	X
E011-01	INTEGER and SMALLINT data types	O
E011-02	REAL, DOUBLE PRECISION, and FLOAT data types	O
E011-03	DECIMAL and NUMERIC data types	X
E011-04	Arithmetic operators	O
E011-05	Numeric comparison	O
E011-06	Implicit casting among the numeric data types	O
E021-01	CHARACTER data type	O
E021-02	CHARACTER VARYING data type	O
E021-03	Character literals	O
E021-04	CHARACTER_LENGTH function	O

Feature ID	说明	是否支持
E021-05	OCTET_LENGTH function	O
E021-06	SUBSTRING function	O
E021-07	Character concatenation	O
E021-08	UPPER and LOWER functions	O
E021-09	TRIM function	O
E021-10	Implicit casting among the fixed-length and variable-length character string types	O
E021-11	POSITION function	O
E021-12	Character comparison	O
E071-05	Columns combined via table operators need not have exactly the same data type	O
F051-01	DATE data type (including support of DATE literal)	O
F051-02	TIME data type (including support of TIME literal) with fractional seconds precision of at least 0	O
F051-03	TIMESTAMP data type (including support of TIMESTAMP literal) with fractional seconds precision of at least 0 and 6	O
F051-04	Comparison predicate on DATE, TIME, and TIMESTAMP data types	X
F051-05	Explicit CAST between datetime types and character string types	O
F054	TIMESTAMP in DATE type precedence list	X

Feature ID	说明	是否支持
F382	Alter column data type	O
F611	Indicator data types	X
F741	Referential MATCH types	X
J521	JDBC data types	X
J622	external Java types	X
S011-01	USER_DEFINED_TYPES view	X
S023	Basic structured types	X
S024	Enhanced structured types	X
S025	Final structured types	X
S026	Self-referencing structured types	X
S041	Basic reference types	X
S043	Enhanced reference types	X
S051	Create table of type	X
S071	SQL paths in function and type name resolution	X
S091-01	Arrays of built-in data types	X
S091-02	Arrays of distinct types	X
S092	Arrays of user-defined types	X
S094	Arrays of reference types	X

Feature ID	说明	是否支持
S161	Subtype treatment	X
S162	Subtype treatment for references	X
S201-02	Array as result type of functions	X
S231	Structured type locators	X
S261	Specific type method	X
S272	Multisets of user-defined types	X
S274	Multisets of reference types	X
S281	Nested collection types	X
S401	Distinct types based on array types	X
S402	Distinct types based on distinct types	X
T021	BINARY and VARBINARY data types	O
T022	Advanced support for BINARY and VARBINARY data types	O
T031	BOOLEAN data type	O
T041	Basic LOB data type support	X
T042	Extended LOB data type support	X
T051	Row types	X
T071	BIGINT data type	O
T201	Comparable data types for referential constraints	X

Feature ID	说明	是否支持
T322	Declared data type attributes	X
X010	XML type	X
X011	Arrays of XML type	X
X012	XMultisets of XML type	X
X013	Distinct types of XML type	X
X014	Attributes of XML type	X
X015	Fields of XML type	X
X181	XML(DOCUMENT(UNTYPED)) type	X
X182	XML(DOCUMENT(ANY)) type	X
X190	XML(SEQUENCE) type	X
X191	XML(DOCUMENT(XMLSCHEMA)) type	X
X192	XML(CONTENT(XMLSCHEMA)) type	X
X231	XML(CONTENT(UNTYPED)) type	X
X232	XML(CONTENT(ANY)) type	X
X251	Persistent XML values of XML(DOCUMENT(UNTYPED)) type	X
X252	Persistent XML values of XML(DOCUMENT(ANY)) type	X
X253	Persistent XML values of XML(CONTENT(UNTYPED)) type	X
X254	Persistent XML values of XML(CONTENT(ANY)) type	X

Feature ID	说明	是否支持
X255	Persistent XML values of XML(SEQUENCE) type	X
X256	Persistent XML values of XML(DOCUMENT(XMLSCHEMA)) type	X
X257	Persistent XML values of XML(CONTENT(XMLSCHEMA)) type	X
X260	XML type: ELEMENT clause	X
X261	XML type: NAMESPACE without ELEMENT clause	X
X263	XML type: NO NAMESPACE with ELEMENT clause	X
X264	XML type: schema location	X
X410	Alter column data type: XML type	X

Table 1-25 SQL标准兼容性

1.3 格式（Format）字符串

格式字符串是用于将数字类型或日期/时间类型转换为字符串或将字符串转换为数字类型或日期/时间类型时定义其形式的字符串

- 将数字类型或日期/时间类型转换为字符串时以如下形式表示字符串
  - **TO_CHAR(number)**, **TO_CHAR(datetime)**
  - 数字类型 : TO_CHAR(1234.56, 'S9,999.99') → '+1,234.56'
  - 日期/时间类型 : TO_CHAR(SYSDATE, 'YYYY-MM-DD') → '2012-07-15'
- 将字符串转换为数字类型或日期/时间类型时以如下形式表示字符串
  - **TO_NUMBER TO_NATIVE_REAL**, **TO_NATIVE_DOUBLE**
  - **TO_DATE**
  - **TO_TIMESTAMP**, **TO_TIMESTAMP_WITH_TIME_ZONE**
  - **TO_TIME**, **TO_TIME_WITH_TIME_ZONE**
  - 数字类型 : TO_NUMBER('+1,234.56', 'S9,999.99') → NUMBER TYPE
  - 日期/时间类型 : TO_DATE('2012-07-15', 'YYYY-MM-DD') → DATE TYPE

格式字符串根据如下类型进行区分

- 数字类型 ： **数字格式字符串**
- 日期/时间类型 ： **日期时间格式字符串**

数字格式字符串

数字格式（number format）字符串是定义将数字类型转换为字符串或将字符串转换为数字类型

时定义格式的字符串

数字格式字符串用作**17.196 TO_CHAR(number)**, **TO_NATIVE_SMALLINT**,

TO_NATIVE_INTEGER, **TO_NATIVE_BIGINT**, **TO_NUMBER**, **TO_NATIVE_REAL**,

TO_NATIVE_DOUBLE函数的参数

数字格式字符串可根据所需格式指定多个格式元素

所有数字格式元素都按照其格式进行四舍五入

要转换的值的小数点之前的位数大于格式字符串中指定的位数时代替为'#'字符

如果未指定表示MI, S, PR符号的格式元素时负数在数字前加'-'符号正数的数字前为空白

格式元素	示例	说明
, (comma)	9,999	在指定位置上返回逗号 可以指定多个逗号 格式字符串不允许以逗号开头而且不能位于小数点(.)后面
. (period)	99.99	在指定位置上返回小数点 格式字符串中只能指定一次小数点
$	$9999	在数字前面返回$符号
0	0999 9990	在数字前面或最后返回0 要转换的值的位数小于直到格式字符串0位置的位数时用0填满差异后返回

格式元素	示例	说明
9	9999	按照符号与已指定的9的数量返回空白与数字 如果要转换的值的位数小于已指定的9的数量时以空白填满差异后返回 正数数字前面为空白负数数字前面加'-'符号后返回 如果格式字符串的小数点前面的正数值为0时该0返回为空白 例: TO_CHAR(0.123, '9.999') -> .123 例: TO_CHAR(0, '9') -> 0
B	B9999	值为0时返回空白
EEEE	9.9EEEE	以指数表示法返回 可以在格式字符串的最后也可以在SMIPR之前 不能与逗号（comma）同时指定
MI	9999MI	在数字的末尾以'-'表示负数以空白表示正数 只能在格式字符串的末尾指定并且不允许与SPR同时指定
PR	9999PR	负数时将数字返回至<>中<数字> 正数时在数字开头和末尾返回空白 只能在格式字符串的末尾指定并且不允许与SMI同时指定
RN rn	RN rn	用大写返回罗马数字 用小写返回罗马数字 只能返回1～3999范围内的数字 除FM格式元素外无法与其他格式元素同时指定 不能用于TO_NUMBER函数

格式元素	示例	说明
S	S9999 9999S	正数时在数字前面返回'+'负数时在数字前面返回'-'(S9999) 正数时在数字末尾返回'+'负数时在数字末尾返回'-'(9999S) 只能在格式字符串的开头或末尾指定 不能与MIPR同时指定
V	999V99	V格式元素后面的9的位数为n时值返回10^n 不能与小数点(.)同时指定 不能用于TO_NUMBER函数
X	XXXX xxxx	根据指定的X位数返回空白与16进制数 以16进制返回整数值（非整数时四舍五入为整数） XXXX返回16进制大写xxxx返回16进制小写 转换的16进制的位数小于指定的X的数量时以空白填满差异后返回 只处理0和正整数负数用'#'字符代替 只能与格式元素 0和FM同时指定不能与其他格式元素同时指定
FM	FM	返回删除前后空白后靠左对齐的效果 删除数字前后的空白 删除小数点后面的由9 格式元素增加的0

Table 1-26 数字格式元素

以下为数字格式字符串的使用示例

```
TO_CHAR( 12345, '99,999' )          : ' 12,345'
```

```
TO_CHAR( 123456789, '999,999,999' )  : ' 123,456,789'
TO_CHAR( 12.345, '99.999' )          : ' 12.345'
TO_CHAR( 1234.56, '$9,999.99' )      : ' $1,234.56'
TO_CHAR( 123, '099999' )             : ' 000123'
TO_CHAR( 0.2, '0.9' )                : ' 0.2'
TO_CHAR( 123.45, '999999.99' )       : '    123.45'
TO_CHAR( -123.45, '999999.99' )      : '   -123.45'
TO_CHAR( 123.45, 'FM999999.99' )     : '123.45'
TO_CHAR( -123.45, 'FM999999.99' )    : '-123.45'
TO_CHAR( 12345.67, '999.99' )        : '#######'
TO_CHAR( 123.100567, '999.999' )     : ' 123.101'
TO_CHAR( 0.2, '90.99' )              : '  0.20'
TO_CHAR( 0.2, '99.99' )              : '   .20'
TO_CHAR( 0, '90.99' )                : '  0.00'
TO_CHAR( 0, 'B90.99' )               : '      '
TO_CHAR( 123.45, '9.9EEEE' )         : '  1.2E+02'
TO_CHAR( 123.45, '999.99MI' )        : '123.45 '
TO_CHAR( -123.45, '999.99MI' )       : '123.45-'
TO_CHAR( 123.45, '999.99PR' )        : ' 123.45 '
TO_CHAR( -123.45, '999.99PR' )       : '<123.45>'
TO_CHAR( 123, 'RN' )                 : '         CXXIII'
TO_CHAR( 123, 'rn' )                 : '         cxxiii'
TO_CHAR( 123, 'FMRN' )               : 'CXXIII'
TO_CHAR( 4000, 'RN' )                : '###############'
TO_CHAR( 123.45, 'S999.99' )         : '+123.45'
```

```
TO_CHAR( -123.45, 'S999.99' )        : '-123.45'
TO_CHAR( 123.45, '999.99S' )         : '123.45+'
TO_CHAR( -123.45, '999.99S' )        : '123.45-'
TO_CHAR( 123.45, '999V999' )         : ' 123450'
TO_CHAR( 123, 'XX' )                 : ' 7B'
TO_CHAR( 123, 'xx' )                 : ' 7b'
TO_CHAR( 45678, 'XXXXXXX' )          : '    B26E'
TO_CHAR( 45678, 'FMXXXXXXX' )        : 'B26E'
TO_CHAR( 123.45, '99,999.999999' )   : '    123.450000'
TO_CHAR( 123.45, 'FM99,999.999999' ) : '123.45'
```

日期时间格式字符串

日期时间格式字符串是定义将日期/时间类型转换为字符串或将字符串转换为日期/时间类型时的格式的字符串

日期时间格式字符串用作**TO_CHAR(datetime)**, **TO_DATE**, **TO_TIMESTAMP**, **TO_TIMESTAMP_WITH_TIME_ZONE**, **TO_TIME**, **TO_TIME_WITH_TIME_ZONE**函数的参数

如果日期/时间类型未指定格式字符串则处理为默认值各类型的默认值为在会话属性NLS_*_FORMAT中指定的值

- DATE : **NLS_DATE_FORMAT**
- TIMESTAMP : **NLS_TIMESTAMP_FORMAT**
- TIMESTAMP WITH TIME ZONE : **NLS_TIMESTAMP_WITH_TIME_ZONE_FORMAT**

- TIME : **NLS_TIME_FORMAT**
- TIME WITH TIME ZONE : **NLS_TIME_WITH_TIME_ZONE_FORMAT**

NLS_*_FORMAT值可以使用**ALTER SESSION SET property_name**语句进行更改

日期时间格式字符串可根据所需表示形式指定多个格式元素

格式元素	是否使用TO_* datetime	说明
- / , . ; : "text" 特殊字符	Y	在指定位置返回格式元素的字符
AD A.D.	Y	公元
AM A.M.	Y	上午
BC B.C.	Y	公元前

格式元素	是否使用TO_* datetime	说明
CC	N	世纪 如果四位数年份的最后两位数字是01~99则返回在前两位数字上加1的值（例如: 年份为2005年时返回21） 如果四位数年份的最后两位为00时则返回前两位数字（例如: 年份为2000年时返回20）
D	Y	返回一周中的第几天（1～7） 星期日(1) ～ 星期六(7)
DAY Day day	Y	返回星期（例如: SUNDAY） • DAY：全部返回大写 • Day：第一个字母返回大写其余返回小写 • day：全部返回小写
DD	Y	返回一个月中的第几天（1～31）
DDD	Y	返回一年中的第几天（1～366）
DY Dy dy	Y	返回星期的简称（例如: SUN） • DY：全部返回大写 • Dy：第一个字母返回大写其余返回小写 • dy：全部返回小写

格式元素	是否使用TO_* datetime	说明
FF[1..6]	Y	按照FF之后指定的数字（1～6）的数量返回小数秒（fractional seconds） 如果未指定数字则默认值为6（FF等于FF6） 如果小数秒（fractional seconds）的位数多于FF之后指定的数字则舍掉 如果小数秒（fractional seconds）的位数少于FF之后指定的数字则用0填满指定的数字 DATE类型中无法使用
HH HH12	Y	小时（1～12）
HH24	Y	小时（0～23）
IW	-	ISO 8601标准定义的Calendar week（1~52周或1~53周） 指定年度的第一个周四所属的周为第一周 • 日历周从星期一开始 • 第一个日历周包含1月4日 • 第一个日历周可能包含12月29日30日和31日 • 上一个日历周可能包含1月1日2日和3日
IYYY	-	包含ISO 8601标准定义的日历周的4位数年份

格式元素	是否使用TO_* datetime	说明
IYY IY I	-	包含ISO 8601标准定义的日历周的3位数年份 包含ISO 8601标准定义的日历周的2位数年份 包含ISO 8601标准定义的日历周的1位数年份
J	Y	返回从BC 4714-11-24日开始经过的日期
MI	Y	分（0～59）
MM	Y	月（01～12）1月（01） ～12月（12）
MON Mon mon	Y	月份的简称（例如: JAN） • MON：全部返回大写 • Mon：第一个字母返回大写其余返回小写 • mon：全部返回小写
MONTH Month month	Y	月份的名称（例如: JANUARY） • MONTH：全部返回大写 • Month：第一个字母返回大写其余返回小写 • month：全部返回小写
PM P.M.	Y	下午

格式元素	是否使用TO_* datetime	说明
Q	N	年度的季度（1～4） 1月到3月(1)～10月到12月(4)
RM Rm rm	Y	用罗马数字返回月份（例如: I） • RM：全部返回大写 • Rm：第一个字母返回大写其余返回小写 • rm：全部返回小写

格式元素	是否使用TO_*datetime	说明

RR	Y	已调整的2位数年份
		将表示为RR的2位数年份转换为4位数年份的方法为如下
		• 以RR表示的2位数年份为00 ~ 49时
		  ◦ 若当前年份的最后2位为00 ~ 50
		  ▪ 则使用当前年度的前两位与表示为RR的两位年份
		  ◦ 若当前年份的最后2位为51 ~ 99时
		  ▪ 则使用（ 当前年份的前两位数字+ 1）与表示为RR的两位年份
		• 以RR表示的2位数年份为50 ~ 99时
		  ◦ 若当前年份的最后2位为00 ~ 50
		  ▪ 则使用(当前年份的前两位数字 - 1）与表示为RR的两位年份
		  ◦ 若当前年份的最后2位为51 ~ 99
		  ▪ 则使用当前年份的前两位与表示为RR的两位年份

RRRR	Y	已调整的4位数年份 可输入四位或两位 输入2位时处理方式与RR相同
SS	Y	秒（0 ~ 59）

格式元素	是否使用TO_* datetime	说明
SSSSS	Y	上一个零点为基准经过的秒（0 ~ 86399）
TZH	Y	时区小时（Time zone hour） 无法在DATE, TIMESTAMP, TIME类型中使用只能用于 TIMESTAMP WITH TIME ZONE, TIME WITH TIME ZONE类型
TZM	Y	时区分钟（Time zone minute） 无法在DATE, TIMESTAMP, TIME类型中使用只能用于 TIMESTAMP WITH TIME ZONE, TIME WITH TIME ZONE类型
WW	N	返回年度的第几周（1 ~ 53） 第一周1为从1月1日开始到7日
W	N	返回月份的第几周（1 ~ 5） 第一周1为月份的第1日开始到7日
Y,YYY	Y	返回包含逗号的Y,YYY形式的年份
YYYY SYYYY	Y	4位数年份 SYYYY标记年度的符号 • BC时标记为‘-’AD时标记为‘ ’

格式元素	是否使用TO_*datetime	说明
YYY YY Y	Y	• YYY：当前年份的最后3位数年份 • YY：当前年份的最后2位数年份 • Y：当前年份的最后1位数年份

Table 1-27 日期时间格式元素

以下为日期时间格式字符串的使用示例

```
* - / , . ; : "text" Special character
  • TO_CHAR( TO_DATE( '2012-07-15 03:30:30', 'YYYY-MM-DD HH12:MI:SS' ),
             'YYYY/MM/DD HH12:MI:SS' )
    ==> '2012/07/15 03:30:30'
  • TO_CHAR( TO_DATE( '2012-07-15', 'YYYY-MM-DD' ),
             'YYYY"year" DDD"th day"' )
    ==> '2012year 197th day'
```

```
* AD
  • TO_CHAR( TO_DATE( '2012-07-15 AD', 'YYYY-MM-DD AD' ), 'YYYY AD' )
    ==> '2012 AD'
  • TO_CHAR( TO_DATE( '0001-01-01 BC', 'YYYY-MM-DD AD'), 'YYYY AD' )
```

```
    ==> '0001 BC'


* BC
  • TO_CHAR( TO_DATE( '0001-01-01 BC', 'YYYY-MM-DD BC'), 'YYYY BC' )
    ==> '0001 BC'
  • TO_CHAR( TO_DATE( '2012-07-15', 'YYYY-MM-DD' ), 'YYYY BC' )
    ==> '2012 AD'



* AM
  • TO_CHAR( TO_DATE( '2012-07-15 03:30:30 AM',
                      'YYYY-MM-DD HH12:MI:SS AM' ),
             'HH12:MI:SS AM' )
    ==> '03:30:30 AM'
  • TO_CHAR( TO_DATE( '2012-07-15 21:30:30', 'YYYY-MM-DD HH24:MI:SS' ),
             'HH12:MI:SS AM' )
    ==> '09:30:30 PM'


* PM
  • TO_CHAR( TO_DATE( '2012-07-15 03:30:30',
                      'YYYY-MM-DD HH24:MI:SS' ),
             'HH12:MI:SS PM' )
    ==> '03:30:30 AM'
  • TO_CHAR( TO_DATE( '2012-07-15 09:30:30 PM',
                      'YYYY-MM-DD HH12:MI:SS PM' ),
```

```
            'HH12:MI:SS PM' )
    ==> '09:30:30 PM'
```

```
* CC
  • TO_CHAR( TO_DATE( '2012-07-15', 'YYYY-MM-DD' ), 'CC' )
    ==> '21'
```

```
* D
  • TO_CHAR( TO_DATE( '2012-07-15', 'YYYY-MM-DD' ), 'D' )
    ==> '1'

* DD
  • TO_CHAR( TO_DATE( '2012-07-15', 'YYYY-MM-DD' ), 'DD' )
    ==>  '15'

* DDD
  • TO_CHAR( TO_DATE( '2012-07-15', 'YYYY-MM-DD' ), 'DDD' )
    ==> '197'
```

```
* DAY
  • TO_CHAR( TO_DATE( '2012-07-15', 'YYYY-MM-DD' ), 'DAY' )
```

==> 'SUNDAY   '

- TO_CHAR(TO_DATE('2012-07-15', 'YYYY-MM-DD'), 'Day')

  ==> 'Sunday   '

- TO_CHAR(TO_DATE('2012-07-15', 'YYYY-MM-DD'), 'day')

  ==> 'sunday   '

* DY

- TO_CHAR(TO_DATE('2012-07-15', 'YYYY-MM-DD'), 'DY')

  ==> 'SUN'

- TO_CHAR(TO_DATE('2012-07-15', 'YYYY-MM-DD'), 'Dy')

  ==> 'Sun'

- TO_CHAR(TO_DATE('2012-07-15', 'YYYY-MM-DD'), 'dy')

  ==> 'sun'

* FF[1 ... 6]

- TO_CHAR(TO_TIMESTAMP('2012-07-15 03:30:45.123456',
                         'YYYY-MM-DD HH24:MI:SS.FF6'),
           'FF')

  ==> '123456'

- TO_CHAR(TO_TIMESTAMP('2012-07-15 03:30:45.123456',
                         'YYYY-MM-DD HH24:MI:SS.FF6'),
           'FF5')

  ==> '12345'

- TO_CHAR(TO_TIMESTAMP('2012-07-15 03:30:45.9',

```
                          'YYYY-MM-DD HH24:MI.SS.FF1' ) ,
            'FF6' )
    ==> '900000'
```

```
* HH HH12 HH24
  • TO_CHAR( TO_TIMESTAMP( '2012-07-15 03:30:45.123456',
                          'YYYY-MM-DD HH24:MI:SS.FF6' ),
            'HH12' )
    ==> '03'
  • TO_CHAR( TO_TIMESTAMP( '2012-07-15 23:30:45.123456',
                          'YYYY-MM-DD HH24:MI:SS.FF6' ),
            'HH12' )
    ==> '11'
  • TO_CHAR( TO_TIMESTAMP( '2012-07-15 23:30:45.123456',
                          'YYYY-MM-DD HH24:MI:SS.FF6' ),
            'HH24' )
    ==> '23'
```

```
* IW
  • TO_CHAR( DATE'2016-01-01', 'IW' )
    ==> 53
  • TO_CHAR( DATE'2014-12-30', 'IW' )
```

==> 01

* IYYY
  - TO_CHAR(DATE'2016-01-01', 'IYYY')
    ==> 2015
  - TO_CHAR(DATE'2014-12-30', 'IYYY')
    ==> 2015

* IYY
  - TO_CHAR(DATE'2016-01-01', 'IYY')
    ==> 015

* IY
  - TO_CHAR(DATE'2016-01-01', 'IY')
    ==> 15

* I
  - TO_CHAR(DATE'2016-01-01', 'I')
    ==> 5

* J
  - TO_CHAR(TO_DATE('2012-07-15', 'YYYY-MM-DD'), 'J')

==> '2456124'

- TO_CHAR(TO_DATE('2456124', 'J'), 'YYYY-MM-DD')

  ==> '2012-07-15'

* MI

- TO_CHAR(TO_TIMESTAMP('2012-07-15 23:30:45.123456',
  'YYYY-MM-DD HH24:MI:SS.FF6'),
  'MI')

  ==> '30'

* MM

- TO_CHAR(TO_TIMESTAMP('2012-07-15 23:30:45.123456',
  'YYYY-MM-DD HH24:MI:SS.FF6'),
  'MM')

  ==> '07'

* MON

- TO_CHAR(TO_DATE('2012-07-15', 'YYYY-MM-DD'), 'MON')

  ==> 'JUL'

- TO_CHAR(TO_DATE('2012-07-15', 'YYYY-MM-DD'), 'Mon')

  ==> 'Jul'

- TO_CHAR(TO_DATE('2012-07-15', 'YYYY-MM-DD'), 'mon')
  ==> 'jul'

* MONTH

- TO_CHAR(TO_DATE('2012-07-15', 'YYYY-MM-DD'), 'MONTH')
  ==> 'JULY   '
- TO_CHAR(TO_DATE('2012-07-15', 'YYYY-MM-DD'), 'Month')
  ==> 'July   '
- TO_CHAR(TO_DATE('2012-07-15', 'YYYY-MM-DD'), 'month')
  ==> 'july   '

* Q

- TO_CHAR(TO_DATE('2012-07-15', 'YYYY-MM-DD'), 'Q')
  ==> '3'

* RM

- TO_CHAR(TO_DATE('2012-07-15', 'YYYY-MM-DD'), 'RM')
  ==> 'VII '
- TO_CHAR(TO_DATE('2012-07-15', 'YYYY-MM-DD'), 'Rm')
  ==> 'Vii '
- TO_CHAR(TO_DATE('2012-07-15', 'YYYY-MM-DD'), 'rm')
  ==> 'vii '

* RR, RRRR (当前年份为2014时)

  - TO_CHAR(TO_DATE('49-07-15', 'RR-MM-DD'), 'RRRR')

    ==> '2049'

  - TO_CHAR(TO_DATE('49-07-15', 'RR-MM-DD'), 'YYYY')

    ==> '2049'

  - TO_CHAR(TO_DATE('50-07-15', 'RR-MM-DD'), 'RRRR')

    ==> '1950'

  - TO_CHAR(TO_DATE('50-07-15', 'RR-MM-DD'), 'YYYY')

    ==> '1950'

  - TO_CHAR(TO_DATE('50-07-15', 'YY-MM-DD'), 'RRRR')

    ==> '2050'

  - TO_CHAR(TO_DATE('49-07-15', 'RRRR-MM-DD'), 'YYYY')

    ==> '2049'

  - TO_CHAR(TO_DATE('50-07-15', 'RRRR-MM-DD'), 'YYYY')

    ==> '1950'

* RR, RRRR (当前年份为2051时)

  - TO_CHAR(TO_DATE('49-07-15', 'RR-MM-DD'), 'RRRR')

    ==> '2149'

  - TO_CHAR(TO_DATE('49-07-15', 'RR-MM-DD'), 'YYYY')

    ==> '2149'

  - TO_CHAR(TO_DATE('50-07-15', 'RR-MM-DD'), 'RRRR')

    ==> '2050'

- TO_CHAR(TO_DATE('50-07-15', 'RR-MM-DD'), 'YYYY')
  ==> '2050'
- TO_CHAR(TO_DATE('50-07-15', 'YY-MM-DD'), 'RRRR')
  ==> '2050'
- TO_CHAR(TO_DATE('49-07-15', 'RRRR-MM-DD'), 'YYYY')
  ==> '2149'
- TO_CHAR(TO_DATE('50-07-15', 'RRRR-MM-DD'), 'YYYY')
  ==> '2050'

* SS
  - TO_CHAR(TO_TIMESTAMP('2012-07-15 23:30:45.123456',
                           'YYYY-MM-DD HH24:MI:SS.FF6'),
             'SS')
    ==> '45'

* SSSSS
  - TO_CHAR(TO_TIMESTAMP('2012-07-15 23:30:45.123456',
                           'YYYY-MM-DD HH24:MI:SS.FF6'),
             'SSSSS')
    ==> '84645'

* TZH

```
  • TO_CHAR( TO_TIMESTAMP_TZ( '2012-07-15 23:30:45.123456 +09:00',
                              'YYYY-MM-DD HH24:MI:SS.FF6 TZH:TZM' ),
            'TZH' )
    ==> '+09'


* TZM
  • TO_CHAR( TO_TIMESTAMP_TZ( '2012-07-15 23:30:45.123456 +09:00',
                              'YYYY-MM-DD HH24:MI:SS.FF6 TZH:TZM' ),
            'TZM' )
    ==> '00'


  • TO_CHAR( TO_TIMESTAMP_TZ( '2012-07-15 23:30:45.123456 +09:00',
                              'YYYY-MM-DD HH24:MI:SS.FF6 TZH:TZM' ),
            'TZH:TZM' )
    ==> '+09:00'
```

```
* WW
  • TO_CHAR( TO_DATE( '2012-07-15', 'YYYY-MM-DD' ), 'WW' )
    ==> '29'


* W
  • TO_CHAR( TO_DATE( '2012-07-15', 'YYYY-MM-DD' ), 'W' )
    ==> '3'
```

* Y,YYY

  • TO_CHAR(TO_DATE('2,012-07-15', 'Y,YYY-MM-DD'), 'Y,YYY')

    ==> '2,012'

* YYYY

  • TO_CHAR(TO_DATE('2012-07-15', 'YYYY-MM-DD'), 'YYYY')

    ==> '2012'

* SYYYY

  • TO_CHAR(TO_DATE('-0001-01-01', 'SYYYY-MM-DD'), 'SYYYY')

    ==> '-0001'

  • TO_CHAR(TO_DATE('2000-01-01', 'SYYYY-MM-DD'), 'SYYYY')

    ==> ' 2000'

* YYY • TO_CHAR(TO_DATE('012-07-15', 'YYY-MM-DD'), 'YYY')

    ==> '012'

  • TO_CHAR(TO_DATE('012-07-15', 'YYY-MM-DD'), 'YYYY')

    ==> '2012' ((当前年度 / 1000年)为2的情况)

* YY

  • TO_CHAR(TO_DATE('12-07-15', 'YY-MM-DD'), 'YY')

    ==> '12'

  • TO_CHAR(TO_DATE('12-07-15', 'YY-MM-DD'), 'YYYY')

==> '2012' ((当前年度 / 100年)为20的情况)

==> '2112' ((当前年度 / 100年)为21的情况)

* Y

- TO_CHAR(TO_DATE('2-07-15', 'Y-MM-DD'), 'Y')

  ==> '2'

- TO_CHAR(TO_DATE('12-07-15', 'YY-MM-DD'), 'YYYY')

  ==> '2012' ((当前年度 / 10年)为201的情况)

  ==> '2052' ((当前年度 / 10年)为205的情况)

1.4 表达式（Expressions）

表达式是用于获取数据值的值运算符函数的组合

可使用表达式的SQL语句的位置如下

- SELECT target语句
- SELECT的 GROUP BY语句
- SELECT的 ORDER BY语句
- SELECT的 WHERE语句 HAVING语句
- INSERT VALUES语句
- UPDATE SET语句
- INSERT, DELETE, UPDATE的RETURN语句

表达式类型如下

- Simple expression
- Compound expresssion
- Boolean value expression
- Case expression
- Datetime expression
- Scalar subquery expression
- Sequence manipulation expression

简单表达式（Simple expression ）: Column, pseudo columns, literals, null value

复合表达式（Compound expression）： 多个表达式的组合

详细内容参考如下

- **空值（Null Value）**
- **字面量（Literals）**

- 伪列（Pseudo Columns）
- Operators
- Functions

Boolean值表达式（Boolean Value Expression）

语法

```
<boolean value expression> ::=
        <boolean term>
      | <boolean value expression> OR <boolean term>

<boolean term> ::=
        <boolean factor>
      | <boolean term> AND <boolean factor>

<boolean factor> ::=
        [ NOT ] <boolean test>

<boolean test> ::=
        <boolean primary> [ IS [ NOT ] <truth value> ]

<truth value> ::=
        TRUE
```

```
      | FALSE
      | UNKNOWN

<boolean primary> ::=
        <column>
        <condition>
      | <boolean predicand>

<boolean predicand> ::=
        <parenthesized boolean value expression>
      | <nonparenthesized value expression primary>

<parenthesized boolean value expression> ::=
        <left paren> <boolean value expression> <right paren>
```

说明

<boolean value expression>描述Boolean值具有boolean值的<boolean primary>有<column>与<condition><boolean predicand><column>应声明为BOOLEAN type的column也可以使用CAST返回boolean值

<boolean值表达式（boolean value expression）>可以使用AND或OR, NOT等逻辑运算符也支持Boolean值的专用运算符 IS, IS NOT

在<boolean测试>中描述的ISIS NOT运算符判断<boolean primary>中描述的Boolean值是否与为<truth value>的TRUEFALSEUNKNOWN中的一个相匹配

详细内容参考**Conditions**

使用示例

```
gSQL> SELECT * FROM T1 WHERE CAST('TRUE' AS BOOLEAN);


I1

-----

TRUE

FALSE

null


3 rows selected.


gSQL> SELECT * FROM T1 WHERE I1;


I1

----

TRUE


1 row selected.


gSQL> SELECT * FROM T1 WHERE I1 IS TRUE;


I1
```

```
----

TRUE


1 row selected.


gSQL> SELECT * FROM T1 WHERE I1 IS NOT FALSE;


I1

----

TRUE

null


2 rows selected.


gSQL> SELECT * FROM T1 WHERE I1 IS UNKNOWN;


I1

----

null


1 row selected.
```

CASE表达式（CASE Expression）

语法

```
<case expression> ::=
        <simple case>
      | <searched case>

<simple case> ::=
        CASE expr WHEN comparison_expr THEN result
                 [ WHEN comparison_expr THEN result ... ]
                 [ ELSE result ]
        END

<searched case> ::=
        CASE WHEN condition THEN result
             [ WHEN condition THEN result ... ]
             [ ELSE result ]
        END
```

说明

按照CASE语句中的顺序判断WHEN ... THEN语句

如果比较结果为FALSE则继续判断后续的WHEN...THEN语句直到显示TRUE为止

如果比较结果为TRUE则返回结果不再继续判断

• Simple case

用equal运算(expr = comparison_expr)判断CASE expr与WHEN ... THEN语句的comparison_expr

• Searched case

判断WHEN ... THEN语句的条件（condition）

若WHEN语句的所有判断结果为FALSE则返回ELSE语句的结果

若省略ELSE语句则结果返回NULL

存在多种类型的THEN或ELSE语句的结果时则根据**结果类型组合规则**决定结果类型

详细内容参考以下链接

• **COALESCE**

• **NULLIF**

使用示例

- 简单示例（Simple case）

```
gSQL> SELECT I1,
            CASE I1 WHEN 1 THEN 'ONE'
                    WHEN 2 THEN 'TWO'
                    ELSE 'NUMBER'
            END AS CASE_RESULT1,
            CASE I1 WHEN 1 THEN 'ONE'
                    WHEN 2 THEN 'TWO'
            END AS CASE_RESULT2
        FROM T1;
```

```
I1 CASE_RESULT1 CASE_RESULT2

-- ------------ -----------

 1 ONE          ONE

 2 TWO          TWO

 3 NUMBER       null

3 rows selected.
```

- 搜索示例（Searched case）

```
gSQL> SELECT I1,

             CASE WHEN I1 = 1 THEN 'ONE'

                  WHEN I1 = 2 THEN 'TWO'

                  ELSE 'NUMBER'

             END AS CASE_RESULT1,

             CASE WHEN I1 = 1 THEN 'ONE'

                  WHEN I1 = 2 THEN 'TWO'

             END AS CASE_RESULT2

        FROM T1;

I1 CASE_RESULT1 CASE_RESULT2

-- ------------ ------------

 1 ONE          ONE

 2 TWO          TWO

 3 NUMBER       null

3 rows selected.
```

CAST规范（CAST specification）

语法

```
CAST( expression AS data_type )
```

说明

CAST将表达式的数据类型转换为指定的data_type的数据类型

使用示例

```
gSQL> SELECT CAST( '1-2' AS INTERVAL YEAR TO MONTH ) AS RESULT FROM DUAL;

RESULT

------

+01-02

1 row selected.
```

标量子查询表达式（Scalar Subquery Expression）

标量子查询表达式是指返回只有一个column的一个row的子查询（Subquery）标量子查询表达式的结果值为子查询的<select list>中描述的值

如果子查询返回0条row则结果值为NULL返回一个以上的row时报错

标量子查询表达式可用于使用表达式（expression）的大部分位置描述子查询时必须使用括号即使用作函数等将标量子查询表达式描述在括号中也需要用单独的括号括住子查询否则会报错

标量子查询表达式的使用示例如下

```
gSQL> select * from dual where dummy = (select * from dual);


DUMMY

-----

X


1 row selected.


gSQL> select sum(select 1 from dual) from dual;


ERR-42000(40000): syntax error

select sum(select 1 from dual) from dual

...........^    ^

Error at line 1


gSQL> select sum((select 1 from dual)) from dual;


SUM((SELECT 1 FROM DUAL))

-------------------------

                        1
```

```
1 row selected.
```

兼容性

表达式的SQL标准兼容性如下

Feature ID	说明	是否支持
E121-03	Value expressions in ORDER BY clause	O
F051-05	Basic date and time Explicit CAST between datetime types and character string types	O
F201	CAST function	O
F261-01	Simple CASE	O
F261-02	Searched CASE	O
F261-03	NULLIF	O
F261-04	COALESCE	O
F263	Comma-separated predicates in simple CASE expression	X
F301	CORRESPONDING in query expressions	X
F385	Drop column generation expression clause	X
F561	Full value expressions	X
F846	Octet support in regular expression operators	X

F847	Nonconstant regular expressions	X
F850	Top-level <order by clause> in <query expression>	O
F855	Nested <order by clause> in <query expression>	O
F856	Nested <fetch first clause> in <query expression>	O
F857	Top-level <fetch first clause> in <query expression>	O
F861	Top-level <result offset clause> in <query expression>	O
F863	Nested <result offset clause> in <query expression>	O
S091-03	Arrays expressions	X
S111	ONLY in query expressions	X
T121	WITH (excluding RECURSIVE) in query expression	O
T581	Regular expression substring function	X

Table 1-28 SQL标准兼容性

1.5 伪列（Pseudo Columns）

Pseudo Column（伪列）与函数类似但执行Pseudo Column时以行为单位每次返回不同的值因此也类似于表的column

名称	说明	参考
CURRVAL	与Sequence有关的伪例	**CURRVAL**
NEXTVAL	与Sequence相关的伪例	**NEXTVAL**
ROWNUM	满足条件的行的编号	**ROWNUM**
ROWID	返回数据库中的记录标识符	**ROWID Pseudo Column**
CLUSTER_GROUP_ID	返回存储记录的Group标识符	**CLUSTER_GROUP_ID Pseudo Column**
CLUSTER_MEMBER_ID	返回存储记录的成员标识符	**CLUSTER_MEMBER_ID Pseudo Column**
CLUSTER_GROUP_NAME	返回存储记录的Group名称	**CLUSTER_GROUP_NAME Pseudo Column**
CLUSTER_MEMBER_NAME	返回存储记录的成员名称	**CLUSTER_MEMBER_NAME Pseudo Column**

CLUSTER_SHARD_ID	返回存储记录的shard的标识符	CLUSTER_SHARD_ID Pseudo Column

Table 1-29 支持的Pseudo Column

ROWID Pseudo Column

ROWID pseudo column为记录标识符返回数据库中每条记录的识别信息

ROWID为了识别数据库中的位置信息根据系统拥有如下信息

单机版系统

- OBJECT_ID
- TABLESPACE_ID
- PAGE_ID
- PAGE中的OFFSET

集群系统

- GRID_BLOCK_SEQUENCE
- GRID_BLOCK_ID
- MEMBER_ID
- SHARD_ID

查询ROWID时以base 64 encoding将内部的存储信息转换为A-Z, a-z, 0-9, +, / 的值后输出

可通过ROWID-related functions获取ROWID内存储的用于识别数据库中的地址的各个信息

记录被删除时该记录的地址可分配给新插入的记录

ROWID pseudo column只能进行SELECT无法进行INSERT, UPDATE, DELETE

详细内容参考**ROWID**, **ROWID-related Functions**

以下为查询ROWID pseudo column的示例

```
gSQL> SELECT ROWID FROM T1;
                  ROWID
----------------------
AAAAAAAAFpEAACAAAEAkAAA
AAAAAAAAFpEAACAAAEAkAAB
AAAAAAAAFpEAACAAAEAkAAC
AAAAAAAAFpEAACAAAEAkAAD
AAAAAAAAFpEAACAAAEAkAAE
5 rows selected.
```

CLUSTER_GROUP_ID Pseudo Column

CLUSTER_GROUP_ID pseudo column返回存储记录的服务器的Group标识符

CLUSTER_GROUP_ID pseudo column只能进行SELECT无法进行INSERT, UPDATE, DELETE

> Note:
>
> 此信息在集群系统中有效

以下为CLUSTER_GROUP_ID pseudo column的查询示例

```
gSQL> SELECT T1.C1, T1.CLUSTER_GROUP_ID FROM T1;

C1 T1.CLUSTER_GROUP_ID

-- -------------------

A                    1

B                    2

C                    3


3 rows selected.
```

CLUSTER_MEMBER_ID Pseudo Column

CLUSTER_MEMBER_ID pseudo column返回存储记录的服务器的成员标识符

CLUSTER_MEMBER_ID pseudo column只能进行SELECT，无法进行INSERT, UPDATE, DELETE

> Note:
>
> 此信息在集群系统中有效

以下为CLUSTER_MEMBER_ID Pseudo column的查询示例

```
gSQL> SELECT T1.C1, T1.CLUSTER_MEMBER_ID FROM T1;

C1 T1.CLUSTER_MEMBER_ID

-- --------------------
```

```
A                    1

B                    3

C                    5


3 rows selected.
```

CLUSTER_GROUP_NAME Pseudo Column

CLUSTER_GROUP_NAME pseudo column返回存储记录的服务器的group名称

CLUSTER_GROUP_NAME pseudo column只能进行SELECT，无法进行INSERT, UPDATE, DELETE

> Note:
>
> 此信息在集群系统中有效

以下为CLUSTER_GROUP_NAME pseudo column的查询示例

```
gSQL> SELECT T1.C1, T1.CLUSTER_GROUP_NAME FROM T1;

C1 T1.CLUSTER_GROUP_NAME

-- --------------------

A  G1

B  G2

C  G3


3 rows selected.
```

CLUSTER_MEMBER_NAME Pseudo Column

CLUSTER_MEMBER_NAME pseudo column返回存储记录的服务器的成员名称

CLUSTER_MEMBER_NAME pseudo column只能进行SELECT，无法进行INSERT, UPDATE, DELETE

> Note:
>
> 此信息在集群系统中有效

以下为CLUSTER_MEMBER_NAME Pseudo column的查询示例

```
gSQL> SELECT T1.C1, T1.CLUSTER_MEMBER_NAME FROM T1;
C1 T1.CLUSTER_MEMBER_NAME
-- ----------------------
A  G1N1
B  G2N1
C  G3N1


3 rows selected.
```

CLUSTER_SHARD_ID Pseudo Column

CLUSTER_SHARD_ID pseudo column返回存储记录的shard标识符

CLUSTER_SHARD_ID pseudo column只能进行SELECT无法进行INSERTUPDATEDELETE

> Note:
>
> 此信息在集群系统中有效

以下为检索CLUSTER_SHARD_ID pseudo column的示例

```
gSQL> SELECT T1.C1, T1.CLUSTER_SHARD_ID FROM T1;


C1 CLUSTER_SHARD_ID

-- ----------------

A                14

B                17

C                 4


3 rows selected.
```

兼容性

Pseudo column的SQL标准兼容性如下

Feature ID	说明	是否支持
T176	Sequence generator support	O

T177	Sequence generator support: simple restart option	O

Table 1-30 SQL标准兼容性

1.6 Operators

Operator在语法上表示为一个以上的特定符号或keyword用1个以上的argument执行功能

Operator分为以下几种类型

• Arithmetic operator

• Concatenation operator

• Set operator

Arithmetic Operator

语法

```
<arithmetic operator> ::=
        <value term>
      | <expression> + <value term>
      | <expression> - <value term>

<value term> ::=
        <value factor>
      | <value term> * <value factor>
      | <value term> / <value factor>

<value factor> ::=
```

```
        <expression>
      | + <expression>
      | - <expression>
```

说明

Arithmetic operator执行数字日期/时间INTERVAL类型的算数运算

Arithmetic operator的优先顺序如下

1. **+ (POSITIVE)- (NEGATIVE)**
2. *** (MULTIPLICATION)/ (DIVISION)**
3. **+ (ADDITION)- (SUBTRACTION)**

Concatenation Operator

语法

```
<concatenation operator> ::=
        <expression> || <expression>
```

说明

Concatenation Operator返回连接CHARACTER STRING类型或BINARY STRING类型的值的字符串

详细内容参考**|| (CONCATENATE)**, **CONCATENATE**

Set Operator

语法

```
<set operator> ::=
        <set operator term>
      | <subquery> UNION [ ALL | DISTINCT ] <set operator term>
      | <subquery> EXCEPT [ ALL | DISTINCT ] <set operator term>
      | <subquery> MINUS [ ALL | DISTINCT ] <set operator term>

<set operator term> ::=
        <subquery>
      | <subquery> INTERSECT [ ALL | DISTINCT ] <set operator term>
```

说明

Set Operator执行子查询(subquery)结果的集合(set)运算

详细内容参考**set operator**

INTERSECT ALL / DISTINCT优先于其他set operator

Operator	说明
UNION ALL	未删除重复的子查询结果的合集
UNION DISTINCT	删除重复的子查询结果的合集

Operator	说明
EXCEPT ALL	未删除重复的子查询结果的差集
EXCEPT DISTINCT	删除重复的子查询结果的差集
MINUS ALL	与EXCEPT ALL相同
MINUS DISTINCT	与EXCEPT DISTINCT相同
INTERSECT ALL	未删除重复的子查询结果的交集
INTERSECT DISTINCT	删除重复的子查询结果的交集

Table 1-31 Set Operators

兼容性

Operator的SQL标准兼容性如下

Feature ID	说明	是否支持
E011-04	Arithmetic operators	O
E021-07	Character concatenation	O
E071-01	UNION DISTINCT table operator	O
E071-02	UNION ALL table operator	O
E071-03	EXCEPT DISTINCT table operator	O

E071-05	Columns combined via table operators need not have exactly the same data type	O
E071-06	Table operators in subqueries	O
F041-08	All comparison operators are supported (rather than just =)	O
F302-01	INTERSECT DISTINCT table operator	O
F302-02	INTERSECT ALL table operator	O
F304	EXCEPT ALL table operator	O
F846	Octet support in regular expression operators	X
J571	NEW operator	X

Table 1-32 SQL标准兼容性

1.7 Functions

功能上与operator类似但在function名后面使用括号指定argument可以包含0个以上的argument

Function主要分以下两种

• Single row function

• Aggregate function

Single Row Function

Single row function为表或视图的每条row生成一个结果row的函数

Single row function分以下几种

• Numeric function

• Character string function returning character values

• Character string function returning number values

• Datetime function

• General comparison function

• Conversion function

• Conditional function

• NULL-related function

• ROWID-related function

• Encryption function

• System information function

Numeric Functions

Numeric function为输入数字型值后返回数字型结果的函数

Numeric function的种类如下

- **ABS**
- **ACOS**
- **ASIN**
- **ATAN**
- **ATAN2**
- **BITAND**
- **BITNOT**
- **BITOR**
- **BITXOR**
- **CBRT**
- **CEIL**
- **COS**
- **COT**
- **DEGREES**
- **EXP**
- **FACTORIAL**
- **FLOOR**
- **LN**
- **LOG**

- **MOD**
- **PI**
- **POWER**
- **RADIANS**
- **RANDOM**
- **ROUND(number)**
- **SHARD_ID**
- **SHIFT_LEFT**
- **SHIFT_RIGHT**
- **SIGN**
- **SIN**
- **SQRT**
- **TAN**
- **TRUNC(number)**
- **WIDTH_BUCKET**

Character String Functions Returning Character Values

Character string functions returning character values是输入CHARACTER STRING型的值后返回CHARACTER STRING型结果的函数

Character string functions returning character value的种类如下

- **CHR**
- **CONCAT**

- CONCATENATE
- INITCAP
- LOWER
- LPAD
- LTRIM
- OVERLAY
- REPEAT
- REPLACE
- REVERSE
- RPAD
- RTRIM
- SPLIT_PART
- SUBSTR
- SUBSTRB
- TRANSLATE
- TRIM
- UPPER

Character String Functions Returning Number Values

Character string functions returning number values是输入CHARACTER STRING型值后返回数字型结果的函数

Character string functions returning number value的类型如下

- **ASCII**
- **BIT_LENGTH**
- **BYTE_LENGTH**
- **CHAR_LENGTH**
- **INSTR**
- **LENGTH**
- **LENGTHB**
- **OCTET_LENGTH**
- **POSITION**

Datetime Functions

Datetime function是输入DATE / TIME / TIMESTAMP / INTERVAL型值后返回DATE / TIME / TIMESTAMP / INTERVAL型结果的函数

Datetime function的种类如下

- **ADDDATE**
- **ADDTIME**
- **ADD_MONTHS**
- **DATEADD**
- **DATEDIFF**
- **DATE_ADD**
- **DATE_PART**
- **EXTRACT**

- **FROM_TZ**
- **LAST_DAY**
- **MONTHS_BETWEEN**

General Comparison Functions

General comparison function是求value集合的最小值或最大值的函数

General comparison function的类型如下

- **GREATEST**
- **LEAST**

Conversion Functions

Conversion function是设置特定数据类型的值的函数

Conversion function的种类如下

- **NUMTODSINTERVAL**
- **NUMTOYMINTERVAL**
- **TO_CHAR(datetime)**
- **TO_CHAR(number)**
- **TO_DATE**
- **TO_NATIVE_BIGINT**
- **TO_NATIVE_DOUBLE**

- **TO_NATIVE_INTEGER**
- **TO_NATIVE_REAL**
- **TO_NATIVE_SMALLINT**
- **TO_NUMBER**
- **TO_TIME**
- **TO_TIME_TZ**
- **TO_TIME_WITH_TIME_ZONE**
- **TO_TIMESTAMP**
- **TO_TIMESTAMP_TZ**
- **TO_TIMESTAMP_WITH_TIME_ZONE**

Conditional Functions

Conditional function为根据条件返回特定值的函数

Conditional function的种类如下

- **CASE2**
- **DECODE**

NULL-related Functions

NULL-related function是根据输入值是否为NULL返回特定值的函数

NULL-related function的种类如下

- **COALESCE**
- **NULLIF**
- **NVL**
- **NVL2**

ROWID-related Functions

ROWID-related function是用于获取ROWID信息的函数

ROWID-related function的种类如下

- 在单机版有效的函数
  - **ROWID_OBJECT_ID**
  - **ROWID_TABLESPACE_ID**
  - **ROWID_PAGE_ID**
  - **ROWID_ROW_NUMBER**
- 在集群中有效的函数
  - **ROWID_GRID_BLOCK_ID**
  - **ROWID_GRID_BLOCK_SEQ**
  - **ROWID_MEMBER_ID**
  - **ROWID_SHARD_ID**

Encryption Functions

Encryption function是用特定算法将已有的plain text进行encrypt/decrypt或返回hash结果值的

函数

Encryption function的详细内容参考**DIGEST**

System Information Functions

System information function是用于获取会话与系统信息的函数

System information function的种类如下

- **CLOCK_DATE**
- **CLOCK_LOCALTIME**
- **CLOCK_LOCALTIMESTAMP**
- **CURRENT_CATALOG**
- **CURRENT_DATE**
- **CURRENT_SCHEMA**
- **CURRENT_TIME**
- **CURRENT_TIMESTAMP**
- **CURRENT_USER**
- **LAST_IDENTITY_VALUE**
- **LOCALTIME**
- **LOCALTIMESTAMP**
- **LOGON_USER**
- **SESSION_ID**
- **SESSION_SERIAL**
- **SESSION_USER**

- SESSIONTIMEZONE
- STATEMENT_DATE
- STATEMENT_LOCALTIME
- STATEMENT_LOCALTIMESTAMP
- STATEMENT_TIME
- STATEMENT_TIMESTAMP
- STATEMENT_VIEW_SCN
- SYSDATE
- SYSTIME
- SYSTIMESTAMP
- TRANSACTION_DATE
- TRANSACTION_LOCALTIME
- TRANSACTION_LOCALTIMESTAMP
- TRANSACTION_TIME
- TRANSACTION_TIMESTAMP
- USER_ID
- VERSION

Aggregate Function

Aggregate function是对多个row生成一个结果row的函数

Aggregation function的种类如下

- COUNT

- **COUNT(*)**
- **SUM**
- **AVG**
- **MIN**
- **MAX**
- **STDDEV**
- **STDDEV_POP**
- **STDDEV_SAMP**
- **VAR_POP**
- **VAR_SAMP**
- **VARIANCE**

Window Function

返回定义的记录范围的function的结果的函数

定义记录范围称为window在OVER <window name or specification>中定义执行范围

Window的详细内容请参考**window clause**

组内各记录拥有对window(定义记录范围)执行window function的结果所以和aggregate function

不同window function给各组返回多个记录

Window function可在select list和order by clause中描述

Window function的种类如下

- AVG() OVER
- CORR() OVER
- COUNT() OVER
- COUNT(*) OVER
- COVAR_POP() OVER
- COVAR_SAMP() OVER
- CUME_DIST() OVER
- DENSE_RANK() OVER
- FIRST() OVER
- FIRST_VALUE() OVER
- LAG() OVER
- LAST() OVER
- LAST_VALUE() OVER
- LEAD() OVER
- LISTAGG() OVER
- MAX() OVER
- MEDIAN() OVER
- MIN() OVER
- NTH_VALUE() OVER
- NTILE() OVER
- PERCENT_RANK() OVER
- PERCENTILE_CONT() OVER
- PERCENTILE_DISC() OVER
- RANK() OVER
- RATIO_TO_REPORT() OVER

- REGR_AVGX() OVER
- REGR_AVGY() OVER
- REGR_COUNT() OVER
- REGR_INTERCEPT() OVER
- REGR_R2() OVER
- REGR_SLOPE() OVER
- REGR_SXX() OVER
- REGR_SXY() OVER
- REGR_SYY() OVER
- ROW_NUMBER() OVER
- STDDEV() OVER
- STDDEV_POP() OVER
- STDDEV_SAMP() OVER
- STRING_AGG() OVER
- SUM() OVER
- VAR_POP() OVER
- VAR_SAMP() OVER
- VARIANCE() OVER

兼容性

function的SQL标准兼容性如下

Feature ID	说明	是否支持
B033	Untyped SQL-invoked function arguments	X
E021-04	CHARACTER_LENGTH function	O
E021-05	OCTET_LENGTH function	O
E021-06	SUBSTRING function	O
E021-08	UPPER and LOWER functions	O
E021-09	TRIM function	O
E021-11	POSITION function	O
E091-01	AVG	O
E091-02	COUNT	O
E091-03	MAX	O
E091-04	MIN	O
E091-05	SUM	O
E091-06	ALL quantifier	O
E091-07	DISTINCT quantifier	O
F131-03	Set functions supported in queries with grouped views	O
F201	CAST function	O
F441	Extended set function support	X
F442	Mixed column references in set functions	X

Feature ID	说明	是否支持
F801	Full set function	X
F842	OCCURRENCES_REGEX function	X
F843	POSITION_REGEX function	X
S071	SQL paths in function and type name resolution	X
S201-02	Array as result type of functions	X
S211	User-defined cast functions	X
S241	Transform functions	X
T041-03	POSITION, LENGTH, LOWER, TRIM, UPPER, and SUBSTRING functions for LOB data types	X
T312	OVERLAY function	O
T321-01	User-defined functions with no overloading	O
T326	Table functions	X
T341	Overloading of SQL-invoked functions and SQL-invoked procedures	X
T433	Multiargument GROUPING function	X
T441	ABS and MOD functions	O
T571	Array-returning external SQL-invoked functions	X
T572	Multiset-returning external SQL-invoked functions	X

Feature ID	说明	是否支持
T581	Regular expression substring function	X
T614	NTILE function	O
T615	LEAD and LAG functions	O
T616	Null treatment option for LEAD and LAG functions	O
T617	FIRST_VALUE and LAST_VALUE functions	O
T618	NTH_VALUE function	O
T619	Nested window functions	X
T621	Enhanced numeric functions	O

Table 1-33 SQL标准兼容性

1.8 Conditions

Condition

判断为TRUE, FALSE, UNKNOWN的表达式

可使用条件的SQL语句的位置如下

- DELETE, UPDATE的WHERE语句
- SELECT的WHERE, HAVING语句
- 其他可用BOOLEAN TYPE的位置

条件的种类如下

- Comparison condition
- Logical condition
- Null condition
- Compound condition
- Pattern-matching condition
- Between condition
- In condition
- Exists condition
- Distinct condition

优先顺序	条件的种类
1	用于条件语句的运算符
2	=, !=, <, >, <=, >=
3	IS [NOT] NULL, [NOT] BETWEEN, [NOT] IN, LIKE, EXISTS, IS [NOT] DISTINCT FROM
4	NOT
5	AND
6	OR

Table 1-34 条件的优先顺序

Comparison Conditions

对比条件的两边并返回TRUE, FALSE, UNKNOWN值的Boolean类型

Condition	说明
=	检查两个表达式是否相同
!=, <>	检查两个表达式是否相异

Condition	说明
>	对比两个表达式以检查是否大于
<	对比两个表达式以检查是否小于
>=	对比两个表达式以检查是否大于或等于
<=	对比两个表达式以检查是否小于或等于
ANY, SOME	左侧的表达式满足右侧expr_list(或subqury结果)中的一个以上的条件则返回TRUE 右侧的subquery无结果时返回FALSE
ALL	左侧的表达式满足所有右侧expr_list(或subqury结果)的条件则返回TRUE 右侧的subquery无结果时返回TRUE

Table 1-35 Comparison Conditions

详细内容参考**类型间比较（type comparison）**

< Simple Comparison Conditions >

语法

```
<simple_comparison_condition> ::=

        <expr>          <comparison_operator> <expr>

      | <expr>          <comparison_operator> ( <subquery> )

      | ( <subquery> )  <comparison_operator> <expr>

      | ( <subquery> )  <comparison_operator> ( <subquery> )
```

```
      | ( <expr_list> ) <comparison_operator> ( <expr_list> )
      | ( <expr_list> ) <comparison_operator> ( <subquery> )
      | ( <subquery> )  <comparison_operator> ( <expr_list> )
      | ( <subquery> )  <comparison_operator> ( <subquery> )


<comparison_operator> ::=
        <  =   >
      | <  !=  >
      | <  <   >
      | <  >   >
      | <  <=  >
      | <  >=  >


<expr_list> ::=
        <expr>
      | <expr>, ... , <expr>
      | ( <expr> )
      | ( <expr> , ... , <expr> )
```

详细内容参考**标量子查询表达式（Scalar Subquery Expression）**

说明

comparison_operator两边为expr_list或subquery时被比较的expr数量或subquery target的数量应相同

为subquery时结果记录应为一条

使用示例

条件	结果
'abc' = 'abc'	TRUE
'abc' != 'abc'	FALSE
'abc' < 'abc'	FALSE
'abc' <= 'abc'	TRUE
'abc' > 'abc'	FALSE
'abc' >= 'abc'	TRUE
(1, 2, 3) = (1, 2, 3)	TRUE
(1, 2, 3) = (1, 2, 4)	FALSE
(1, 2, 3) != (4, 5, 6)	TRUE
(1, 2, 3) != (1, 2, 3)	FALSE
(1, 2, 3) < (1, 2, 4)	TRUE
(1, 2, 3) < (1, 2, 3)	FALSE
(1, 2, 3) <= (1, 2, 4)	TRUE
(1, 2, 3) <= (1, 2, 2)	FALSE
(1, 2, 3) > (1, 2, 2)	TRUE
(1, 2, 3) > (1, 2, 4)	FALSE

条件	结果
(1, 2, 3) >= (1, 2, 2)	TRUE
(1, 2, 3) >= (1, 2, 4)	FALSE

Table 1-36　Simple comparison condition的示例

< Group Comparison Conditions >

语法

```
<group_comparison_condition> ::=

   <expr>          <comparison_operator> <quantifier> ( <expr_list> )

 | <expr>          <comparison_operator> <quantifier> ( <subquery> )

 | ( <expr_list> ) <comparison_operator> <quantifier> ( <expr_list_list> )

 | ( <expr_list> ) <comparison_operator> <quantifier> ( <subquery> )

 | ( <subquery> )  <comparison_operator> <quantifier> ( <expr_list> )

 | ( <subquery> )  <comparison_operator> <quantifier> ( <expr_list_list> )

 | ( <subquery> )  <comparison_operator> <quantifier> ( <subquery> )


<comparison_operator> ::=

        <  =   >

      | <  !=  >

      | <  <   >

      | <  >   >
```

```
      | <  <=  >

      | <  >=  >


<quantifier> ::=

        ALL

      | ANY

      | SOME


<expr_list> ::=

        <expr>

      | <expr>, ... , <expr>

      | ( <expr> )

      | ( <expr> , ... , <expr> )


<expr_list_list> ::=

        <expr_list>

      | <expr_list>, ... , <expr_list>
```

详细内容参考**标量子查询表达式（Scalar Subquery Expression）**

说明

comparison_operator两边为expr_list或subquery时被比较的expr数量或subquery target的数量应相同

comparison_operator左侧为subquery时结果记录应为一条

comparison_operator右侧为subquery时结果记录可以为多条

使用示例

条件	结果
1 =any (1, 2, 3, 4, 5)	TRUE
1 =any (1, 2, null, 4, 5)	TRUE
1 =any (2, null, 4, 5)	NULL
1 =any (100, 2, 3, 4, 5)	FALSE
1 =all (1, +1, 1E+0)	TRUE
1 =all (1, +1, 1E+0, null)	NULL
1 =all (1, 2, 3, 4, 5)	FALSE
(1, 2) =any ((0, 1), (1, 2), (3, 4))	TRUE
(1, 2) =any ((0, 1), (1, 2), (null, null))	TRUE
(1, 2) =any ((0, 1), (2, 3), (3, 4))	FALSE
(1, 2) =all ((1, 2), (+1, +2), (1E+0, 2E+0))	TRUE
(1, 2) =all ((1, 2), (+1, +2), (null, null))	NULL
(1, 2) =all ((0, 1), (2, 3), (3, 4))	FALSE
comparison_operator右侧的subquery的结果记录为0时	
('X') =any (select dummy from dual where dummy = 'Y')	FALSE
('X') =all (select dummy from dual where dummy = 'Y')	TRUE

Table 1-37 Group comparison condition的示例

Logical Conditions

Logical条件有AND, OR, NOT

AND

语法

```
<boolean value expression> AND <boolean value expression>
```

说明

AND	True	False	Unknown
True	True	False	Unknown
False	False	False	False
Unknown	Unknown	False	Unknown

Table 1-38 AND boolean operator的Truth table

OR

语法

```
<boolean value expression> OR <boolean value expression>
```

说明

OR	True	False	Unknown
True	True	True	True
False	True	False	Unknown
Unknown	True	Unknown	Unknown

Table 1-39 OR boolean operator的Truth table

NOT

语法

```
NOT <boolean value expression>
```

说明

expr	NOT
True	False
False	True
Unknown	Unknown

Table 1-40 NOT boolean operator的Truth table

Null Condition

语法

```
<expr> IS [NOT] NULL
```

说明

检查expr的结果是否为NULL值

expr	IS NULL	IS NOT NULL
NULL	True	False
NOT NULL	False	True

Table 1-41 Is Null条件的结果表

Compound Condition

组合多个条件的条件式

```
compound_condition ::=
        ( condition )
      | NOT condition
      | condition < AND | OR > condition
```

Pattern-matching Conditions

LIKE Condition

语法

```
like_condition ::=
        string [NOT] LIKE pattern [ ESCAPE escape_character ]
```

说明

检查string是否与指定的pattern一致

string, pattern, escape_charater可以接CHARACTER, CHARACTER VARYING, CHARACTER LONG

VARYING等字符类型或可转换为字符类型的类型

string, pattern, escape_character为NULL时结果返回NULL

省略escape_character时无默认值

指定escape_character时escape_character应为一个字符

pattern中不包含'_'或'%'时处理方式与equal运算(string = pattern)相同

pattern中包含'_'或'%'时如下由string判断是否一致

- '_'：对应于1个任意字符
- '%'：对应于拥有0个以上的字符的任意字符串

用字符对比pattern中包含的'_'或'%'时使用ESCAPE语句

指定escape_character后把指定的escape_character写于pattern的'_'或'%'之前

使用示例

```
gSQL> SELECT 'hello%' LIKE 'h%o!%' ESCAPE '!' AS RESULT FROM DUAL;
RESULT
------
TRUE
```

- 'represent' LIKE 'represent'   => TRUE
- 'represent' LIKE ' represent ' => FALSE
- 'represent' LIKE 'REPRESENT'   => FALSE
- 'represent' LIKE 'r_pr_s_nt'   => TRUE
- 'represent' LIKE 're%t'   => TRUE
- 'represent' LIKE 'rep'   => FALSE

- 'summer_vacation' LIKE 'summer_vacation' ESCAPE '\' => TRUE
- NULL LIKE 'summer_vacation' ESCAPE '\'   => NULL
- 'summer_vacation' LIKE NULL ESCAPE '\'   => NULL
- 'summer_vacation' LIKE 'summer_vacation' ESCAPE NULL => NULL

BETWEEN Condition

语法

```
<between condition> ::=

   <expr1> [ NOT ] BETWEEN [ ASYMMETRIC | SYMMETRIC ] <expr2> AND <expr3>
```

说明

检查expr1是否为expr2和expr3范围内的条件

省略ASYMMETRIC或SYMMETRIC时默认为ASYMMETRIC

expr1, expr2, expr3的数据类型不一致时执行conversion

详细内容参考**类型间比较（type comparison）类型间转换（type conversion）**

A	B
X BETWEEN ASYMMETRIC Y AND Z	X BETWEEN Y AND Z
X BETWEEN Y AND Z	X >= Y AND X <= Z

A	B
X NOT BETWEEN Y AND Z	NOT(X BETWEEN Y AND Z)
X BETWEEN SYMMETRIC Y AND Z	((X BETWEEN Y AND Z) OR (X BETWEEN Z AND Y)
X NOT BETWEEN SYMMETRIC Y AND Z	NOT(X BETWEEN SYMMETRIC Y AND Z)

Table 1-42 Between语句等值

使用示例

Condition		结果
BETWEEN[ASYMMETRIC]	3 BETWEEN 1 AND 5	TRUE
	NULL BETWEEN 1 AND 5 3 BETWEEN NULL AND 5 3 BETWEEN 1 AND NULL	NULL
	3 BETWEEN 5 AND 1	FALSE
BETWEEN SYMMETRIC	3 BETWEEN SYMMETRIC 1 AND 5	TRUE
	NULL BETWEEN SYMMETRIC 1 AND 5 3 BETWEEN SYMMETRIC NULL AND 5 3 BETWEEN SYMMETRIC 1 AND NULL	NULL
	3 BETWEEN SYMMETRIC 5 AND 1	TRUE

Table 1-43 Between语句的示例

IN Condition

语法

```
<in_condition> ::=

        <expr>          [NOT] IN ( <expr_list> )

      | <expr>          [NOT] IN ( <subquery> )

      | ( <expr_list> ) [NOT] IN ( <expr_list_list> )

      | ( <expr_list> ) [NOT] IN ( <subquery> )

      | ( <subquery> )  [NOT] IN ( <expr_list> )

      | ( <subquery> )  [NOT] IN ( <expr_list_list> )

      | ( <subquery> )  [NOT] IN ( <subquery> )


<expr_list> ::=

        <expr>

      | <expr>, ... , <expr>

      | ( <expr> )

      | ( <expr> , ... , <expr> )


<expr_list_list> ::=

        <expr_list>

      | <expr_list>, ... , <expr_list>
```

说明

IN Condition返回与=ANY相同的结果

NOT IN Condition返回与!=ALL相同的结果

详细内容参考**Comparison Conditions**

使用示例

条件式	结果
1 IN (1, 2, 3, 4, 5)	TRUE
1 IN (1, 2, null, 4, 5)	TRUE
1 IN (2, null, 4, 5)	NULL
1 IN (100, 2, 3, 4, 5)	FALSE
NULL IN (1, 2, 3)	NULL
1 NOT IN (2, 3, 4, 5)	TRUE
1 NOT IN (2, null, 4, 5)	NULL
1 NOT IN (1, 2, null, 4, 5)	FALSE
1 NOT IN (100, 2, 3, 4, 5)	TRUE
NULL NOT IN (1, 2, 3)	NULL

Table 1-44 IN condition的示例

EXISTS Condition

语法

```
exists_conditions ::=

        EXISTS ( subquery )
```

说明

检查是否存在subquery的结果记录

存在时返回TRUE不存在则返回FALSE

使用示例

```
gSQL> SELECT * FROM DUAL WHERE EXISTS ( SELECT * FROM DUAL );

      DUMMY

      -----

      X

      1 row selected.


gSQL> SELECT * FROM DUAL

       WHERE EXISTS ( SELECT * FROM DUAL WHERE DUMMY = 'Y' );

      no rows selected.
```

DISTINCT Condition

语句

```
distinct_conditions ::=

        <expr> IS [NOT] DISTINCT FROM <expr>

      | ( <expr_list> ) IS [NOT] DISTINCT FROM ( <expr_list> )
```

说明

Distinct condition的操作数必须是可互相进行比较的类型

当操作数为<expr_list>时相同position的数据将会成为比较对象

Distinct condition的所有操作数为not null value时

is distinct from和not equal(!=)相同

is not distinct from返回的值与eqaul(=) 相同

Distinct condition与其他比较运算符的差别在于将NULL value作为一般数据处理而非unknown

- IS DISTINCT FROM
  - 操作数全部为NULL的情况
    - NULL is distinct from NULL => FALSE
  - 操作数中有一个是NULL的情况
    - NULL is distinct from 1 => TRUE
    - 1 is distinct from NULL => TRUE
  - 操作数全部都是not NULL的情况
    - 1 is distinct from 1 => FALSE

        - 1 is distinct from 2 => TRUE
    - 操作数为 <expr_list>的情况
        - (1, 2, 3) is distinct from (1, 2, 3) => FALSE
        - (1, 2, 3) is distinct from (1, 3, 3) => TRUE
        - (1, 2, 3) is distinct from (4, 5, 6) => TRUE
- IS NOT DISTINCT FROM
    - 操作数全部为NULL的情况
        - NULL is not distinct from NULL => TRUE
    - 操作数中有一个是NULL的情况
        - NULL is not distinct from 1 => FALSE
        - 1 is not distinct from NULL => FALSE
    - 操作数全部都是not NULL的情况
        - 1 is not distinct from 1 => TRUE
        - 1 is not distinct from 2 => FALSE
    - 操作数为 <expr_list>的情况
        - (1, 2, 3) is not distinct from (1, 2, 3) => TRUE
        - (1, 2, 3) is not distinct from (1, 3, 3) => FALSE
        - (1, 2, 3) is not distinct from (4, 5, 6) => FALSE

使用示例

```
* IS DISTINCT FROM


gSQL>

SELECT i1,
```

```
       i2,
       i1 IS DISTINCT FROM i2 AS IsDistinct
  FROM t1;


  I1   I2 ISDISTINCT
---- ---- ----------
   1 null TRUE
   1    1 FALSE
   1    2 TRUE
null null FALSE
null    1 TRUE
null    2 TRUE

6 rows selected.

gSQL>
SELECT i1,
       i2,
       i3,
       ( I1, I2, I3 ) IS DISTINCT FROM ( 1, 1, 1 ) AS RESULT
  FROM t1;


  I1   I2   I3 RESULT
---- ---- ---- ------
   1    1    1 FALSE
```

```
   2 null    3 TRUE
null null null TRUE


3 rows selected.



* IS NOT DISTINCT FROM


gSQL>
SELECT i1,
       i2,
       i1 IS NOT DISTINCT FROM i2 AS IsNotDistinct
  FROM t1;


  I1   I2 ISNOTDISTINCT
---- ---- -------------
   1 null FALSE
   1    1 TRUE
   1    2 FALSE
null null TRUE
null    1 FALSE
null    2 FALSE


6 rows selected.
```

```
gSQL>
SELECT i1,
       i2,
       i3,
       ( I1, I2, I3 ) IS NOT DISTINCT FROM ( 1, 1, 1 ) AS RESULT
 FROM t1;
  I1   I2   I3 RESULT
---- ---- ---- ------
   1    1    1 TRUE
   2 null    3 FALSE
null null null FALSE

3 rows selected.
```

兼容性

条件的SQL标准兼容性如下

Feature ID	说明	是否支持
E061-01	Comparison predicate	O
E061-02	BETWEEN predicate	O
E061-03	IN predicate with list of values	O
E061-04	LIKE predicate	O

Feature ID	说明	是否支持
E061-05	LIKE predicate: ESCAPE clause	O
E061-06	NULL predicate	O
E061-07	Quantified comparison predicate	O
E061-08	EXISTS predicate	O
E061-09	Subqueries in comparison predicate	O
E061-11	Subqueries in IN predicate	O
E061-12	Subqueries in quantified comparison predicate	O
E061-13	Correlated subqueries	O
E061-14	Search condition	O
F051-04	Comparison predicate on DATE, TIME, and TIMESTAMP data types	X
F053	OVERLAPS predicate	X
F263	Comma-separated predicates in simple CASE expression	X
F291	UNIQUE predicate	X
F481	Expanded NULL predicate	O
F841	LIKE_REGEX predicate	X
P008	Comma-separated predicates in a CASE statement Extended CASE	X
S151	Type predicate	X
T141	SIMILAR predicate	X

Feature ID	说明	是否支持
T151	DISTINCT predicate	X
T152	DISTINCT predicate with negation	X
T461	Symmetric BETWEEN predicate	O
T501	Enhanced EXISTS predicate	X
T631	IN predicate with one list element	X
X090	XML document predicate	X
X091	XML content predicate	X
X141	IS VALID predicate: data-driven case	X
X142	IS VALID predicate: ACCORDING TO clause	X
X143	IS VALID predicate: ELEMENT clause	X
X144	IS VALID predicate: schema location	X
X145	IS VALID predicate outside check constraints	X
X151	IS VALID predicate with DOCUMENT option	X
X152	IS VALID predicate with CONTENT option	X
X153	IS VALID predicate with SEQUENCE option	X
X155	IS VALID predicate: NAMESPACE without ELEMENT clause	X
X157	IS VALID predicate: NO NAMESPACE with ELEMENT clause	X

Table 1-45 SQL标准兼容性

2.SQL Languages

SQL(J结构化查询语言)大致分为以下几种

- Data Definition Language：数据定义语言
- Data Manipulation Language：数据操作语言
- Data Query Language：数据查询语言
- Control Language：控制语言

2.1 数据定义语言（Data Definition Language）

DDL相关语句

相关内容参考如下

- Non-schema object DDL
  - **数据库相关语句**
  - **Profile相关语句**
  - **Audit Policy相关语句**
  - **Authorization相关语句**
  - **Schema相关语句**
  - **Tablespace相关语句**

- SQL schema object DDL
  - **表相关语句**
  - **Index相关语句**
  - **View相关语句**
  - **Sequence相关语句**
  - **Synonym相关语句**
- 集群对象（cluster object）DDL
  - **集群系统相关语句**
  - **Cluster Group 相关语句**
  - **Cluster Member 相关语法**
  - **Cluster Location 相关语句**
  - **Global Secondary Index 相关语句**

DDL概念

数据定义语言（Data Definition LanguageDDL）是创建（CREATE）删除（DROP）变更（ALTER）SQL对象的SQL语言

构成数据库的SQL对象如下表各对象的详细说明参考相关链接

区分	对象	对象说明	参考
Non-schema 对象	Profile	定义密码管理策略的对象	**Profile**
	Audit policy	定义SQL审计策略的对象	**Audit Policy**
	User	由权限的集合组成的用户对象	**Authorization**

区分	对象	对象说明	参考
	Schema	包含表等SQL schema对象的逻辑位置	**Schema**
	Tablespace	表索引等对象的物理存储位置	**Tablespace**
	Public synonym	公用同义词	**Public Synonym**
SQL schema 对象	Table	存储数据的物理关系	**Table**
	View	由查询组成的逻辑关系	**View**
	Index	提高语句性能的索引对象	**Index**
	Sequence	生成序列号的对象	**Sequence**
	Synonym	声明对象的替换名的对象	**Synonym**
	Stored procedure	用户定义的过程对象	**Stored Procedure**
	Stored function	用户定义的函数对象	**Stored Function**
Cluster 对象	Cluster group	集群成员集	**Cluster Group**
	Cluster member	构成集群系统的数据服务器	**Cluster Member**
	Cluster location	集群成员的位置信息对象	**Cluster Location**
	Shard	水平划分集群表的row集合	**集群表与分片**
	Global secondary index	集群的row标识符的索引	**Global Secondary Index（全局二级索引）**

Table 2-1 SQL对象种类

DDL与事务

SUNDB的事务不仅是插入（INSERT）删除（DELETE）更新（UPDATE）数据的DML还包含创建（CREATE）删除（DROP）变更（ALTER）对象的DDL大部分数据库执行DDL时默认执行COMMIT相反SUNDB的事务中包含DDL以此保证事务的原子性（atomicity）与一致性（consistency）

此功能主要用于进行数据库迁移或安装工具等原子性地执行批量DDL时或由于用户失误执行DROP TABLETRUNCATE TABLE等语句后需要通过回滚进行恢复等的情况

DDL语句的属性为自动提交时执行语句时自动执行COMMIT不是自动提交时即使在执行语句之后也可以回滚事务如下可使用**V$SQL_COMMAND**视图查看DDL是否为自动提交

```
gSQL>
SELECT command, auto_commit
  FROM V$SQL_COMMAND
 WHERE is_ddl = 'YES';



COMMAND                                                  AUTO_COMMIT
-------------------------------------------------------- -----------
ALTER AUDIT POLICY                                       YES
ALTER CLUSTER GROUP .. ADD CLUSTER MEMBER                YES
ALTER CLUSTER GROUP .. OFFLINE CLUSTER MEMBER            YES
```

```
ALTER DATABASE DROP INACTIVE CLUSTER MEMBERS              YES

ALTER DATABASE OFFLINE INACTIVE CLUSTER MEMBERS           YES

ALTER DATABASE RESET LOCAL CLUSTER MEMBER                 YES

ALTER DATABASE ADD LOGFILE GROUP                          YES

ALTER DATABASE ADD LOGFILE MEMBER                         YES

ALTER DATABASE DROP LOGFILE GROUP                         YES

ALTER DATABASE DROP LOGFILE MEMBER                        YES

ALTER DATABASE RENAME GLOBAL TRANSACTION LOGFILE          YES

ALTER DATABASE RENAME LOGFILE                             YES

ALTER DATABASE ARCHIVELOG                                 YES

ALTER DATABASE NOARCHIVELOG                               YES

ALTER DATABASE DATAFILE AUTOEXTEND ..                     YES

ALTER DATABASE CLEAR PASSWORD HISTORY                     NO

ALTER FUNCTION                                            NO

ALTER INDEX AGING                                         NO

ALTER INDEX .. STORAGE                                    NO

ALTER INDEX .. RENAME                                     NO

ALTER INDEX .. REBUILD                                    YES

ALTER INDEX .. COALESCE                                   NO

ALTER PACKAGE                                             YES

ALTER PROCEDURE                                           NO

ALTER PROFILE                                             YES

ALTER SEQUENCE                                            YES

ALTER SEQUENCE .. SYNCHRONIZE                             YES

ALTER SYSTEM SWITCH LOGFILE                               YES
```

```
ALTER TABLE .. ADD COLUMN                                    NO

ALTER TABLE .. SET UNUSED COLUMN                             NO

ALTER TABLE .. ALTER COLUMN .. SET DEFAULT                   NO

ALTER TABLE .. ALTER COLUMN .. DROP DEFAULT                  NO

ALTER TABLE .. ALTER COLUMN .. SET NOT NULL                  NO

ALTER TABLE .. ALTER COLUMN .. DROP NOT NULL                 NO

ALTER TABLE .. ALTER COLUMN .. SET DATA TYPE                 YES

ALTER TABLE .. ALTER COLUMN .. AS IDENTITY                   YES

ALTER TABLE .. ALTER COLUMN .. DROP IDENTITY                 YES

ALTER TABLE .. RENAME COLUMN                                 NO

ALTER TABLE .. STORAGE                                       NO

ALTER TABLE .. ADD CONSTRAINT                                NO

ALTER TABLE .. ALTER CONSTRAINT                              NO

ALTER TABLE .. DROP CONSTRAINT                               NO

ALTER TABLE .. RENAME CONSTRAINT                             NO

ALTER TABLE .. RENAME TO ..                                  NO

ALTER TABLE .. REBALANCE ..                                  YES

ALTER TABLE .. MOVE SHARD .. TO CLUSTER GROUP ..             YES

ALTER TABLE .. SPLIT SHARD .. INTO .. AT CLUSTER GROUP .. YES

ALTER TABLE .. MERGE SHARDS .. INTO ..                       YES

ALTER TABLE .. RENAME SHARD .. TO ..                         NO

ALTER TABLE .. ADD SUPPLEMENTAL LOG                          NO

ALTER TABLE .. ADD GLOBAL SECONDARY INDEX                    NO

ALTER TABLE .. ALTER GLOBAL SECONDARY INDEX                  NO

ALTER TABLE .. ALTER GLOBAL SECONDARY INDEX AGING            NO
```

```
ALTER TABLE .. DROP GLOBAL SECONDARY INDEX                NO

ALTER TABLE .. ALTER GLOBAL SECONDARY INDEX REBUILD       YES

ALTER TABLE .. ALTER GLOBAL SECONDARY INDEX COALESCE      NO

ALTER TABLE .. DROP SUPPLEMENTAL LOG                      NO

ALTER TABLE .. READ ONLY                                  YES

ALTER TABLE .. READ WRITE                                 YES

ALTER TABLE .. SYNCHRONIZE IDENTITY COLUMN                YES

ALTER TABLESPACE .. ADD                                   YES

ALTER TABLESPACE .. DROP                                  YES

ALTER TABLESPACE .. ONLINE                                YES

ALTER TABLESPACE .. OFFLINE                               YES

ALTER TABLESPACE .. RENAME TO                             YES

ALTER TABLESPACE .. RENAME { DATAFILE | MEMORY }          YES

ALTER USER                                                YES

ALTER USER .. IDENTIFIED BY                               YES

ALTER VIEW                                                NO

ANALYZE SYSTEM COMPUTE STATISTICS                         NO

ANALYZE SYSTEM DELETE STATISTICS                          NO

ANALYZE TABLE .. [COMPUTE|ESTIMATE] STATISTICS            YES

ANALYZE TABLE .. DELETE STATISTICS                        NO

AUDIT POLICY                                              YES

COMMENT ON .. IS                                          NO

CREATE AUDIT POLICY                                       YES

CREATE CLUSTER GROUP                                      YES

CREATE FUNCTION                                           NO
```

CREATE INDEX	NO
CREATE PACKAGE	YES
CREATE PACKAGE BODY	YES
CREATE PROCEDURE	NO
CREATE PROFILE	YES
CREATE SCHEMA	YES
CREATE SEQUENCE	YES
CREATE SYNONYM	NO
CREATE TABLE	NO
CREATE TABLE ... AS SELECT	NO
CREATE TABLESPACE	YES
CREATE USER	YES
CREATE VIEW	NO
DROP AUDIT POLICY	YES
DROP CLUSTER GROUP	YES
DROP FUNCTION	NO
DROP INDEX	NO
DROP PACKAGE	YES
DROP PROCEDURE	NO
DROP PROFILE	YES
DROP SCHEMA	YES
DROP SEQUENCE	YES
DROP SYNONYM	NO
DROP TABLE	NO
DROP TABLESPACE	YES

DROP USER	YES
DROP VIEW	NO
GRANT .. ON DATABASE	NO
GRANT .. ON TABLESPACE	NO
GRANT .. ON SCHEMA	NO
GRANT .. ON TABLE	NO
GRANT USAGE ON ..	NO
GRANT .. ON PROCEDURE	NO
GRANT .. ON PACKAGE	NO
NOAUDIT POLICY	YES
REVOKE .. ON DATABASE	NO
REVOKE .. ON TABLESPACE	NO
REVOKE .. ON SCHEMA	NO
REVOKE .. ON TABLE	NO
REVOKE USAGE ON ..	NO
REVOKE .. ON PROCEDURE	NO
REVOKE .. ON PACKAGE	NO
TRUNCATE TABLE	NO
PURGE CONSTRAINT	NO
PURGE INDEX	NO
PURGE TABLE	NO
PURGE TABLESPACE	YES
PURGE RECYCLEBIN	YES
PURGE DBA_RECYCLEBIN	YES
FLASHBACK TABLE	YES

```
128 rows selected.
```

以下举例说明表相关DDL语句包含在事务内并执行COMMIT或ROLLBACK的情况与对其他事务产生的影响通过以下例子可以说明包含DDL的事务也可保证事务的原子性（atomicity）并在提交之前或回滚时也不影响其他事务的读取一致性（consistency）

对象的创建与事务

- 创建表的事务被提交/回滚前

如下创建表后未提交事务时可以在执行DDL的事务内操作该表的数据但其他事务中无法查询该表直到提交创建表的事务即与INSERT语句相同在其他事务中提交事务之前无法查询CREATE TABLE语句

  - 事务A：创建t1表后不提交事务

```
gSQL> CREATE TABLE t1 ( id INTEGER, name VARCHAR(128) );
Table created.

gSQL> INSERT INTO t1 VALUES ( 1, 'leekmo' );
1 row created.

gSQL> SELECT * FROM t1;

ID NAME
```

```
-- ------
 1 leekmo


1 row selected.
```

如上在事务A中创建t1表后未提交时如下在其他会话中执行的事务B无法查询t1表而且无法创建与未COMMIT的t1表相同名称的表

- 事务B：提交事务A之前无法查看t1表

```
gSQL> SELECT * FROM t1;


ERR-42000(16040): table or view does not exist :
SELECT * FROM t1
              *
ERROR at line 1:
```

- 回滚事务A之前无法创建t1表

```
gSQL> CREATE TABLE t1 ( emp_no INTEGER );


ERR-HYT00(14026): resource busy or timeout expired
```

- 提交创建表的事务后

提交事务A后如下事务B可查询t1表创建表的语句返回表示t1表已存在的有效报错

- 事务B：提交事务A后如下可查询表

```
gSQL> SELECT * FROM t1;


ID NAME
-- ------
 1 leekmo


1 row selected.
```

- 提交事务A后如下返回有效报错

```
gSQL> CREATE TABLE t1 ( emp_no INTEGER );


ERR-42000(16005): name 'PUBLIC.T1' is already used by an existing object :
CREATE TABLE t1 ( emp_no INTEGER )
              *
ERROR at line 1:
```

- 回滚创建表的事务后

回滚事务A时创建t1表也会被回滚如下事务B可以创建t1表

- 事务B：事务A被回滚后成为与创建t1表之前相同的状态

```
gSQL> SELECT * FROM t1;
```

```
ERR-42000(16040): table or view does not exist :
SELECT * FROM t1
              *
ERROR at line 1:
```

- 回滚事务A后可创建t1表

```
gSQL> CREATE TABLE t1 ( emp_no INTEGER );


Table created.
```

对象的删除与事务

- 提交或回滚删除表的事务前

删除表后未提交事务时在执行DROP TABLE语句的事务提交之前其他事务可查询被删除的表即与DELETE语句相同DROP TABLE语句也在事务被提交之前在其他事务可以查询删除表之前的状态

以下为在事务A中删除t1表并创建新的表t1后尚未提交事务的状态

- 事务A：DROP前t1表中有一个拥有两个column的row

```
gSQL> SELECT * FROM t1;


ID NAME
-- ------
```

```
 1 leekmo


1 row selected.
```

- 删除原有的表t1

```
gSQL> DROP TABLE t1;


Table dropped.
```

- 创建新的表t1

```
gSQL> CREATE TABLE t1 ( addr VARCHAR(128) );


Table created.
```

- 在新的表t1中创建新的row

```
gSQL> INSERT INTO t1 VALUES ( 'Seoul, Korea' );


1 row created.


gSQL> SELECT * FROM t1;


ADDR
------------
Seoul, Korea
```

```
1 row selected.
```

未提交事务A时如下事务B执行查询则查询到执行事务A之前的状态信息即与DELETE语句相同

DROP TABLE语句在事务被提交之前不影响其他事务

- 事务B：事务B查询执行事务A之前的表

```
gSQL> SELECT * FROM t1;


ID NAME
-- ------
 1 leekmo


1 row selected.
```

- 提交删除表的事务后

提交事务A后如下事务B查询新创建的t1表

- 事务B：提交事务A后查询新创建的表t1

```
gSQL> SELECT * FROM t1;


ADDR
------------
Seoul, Korea
```

```
1 row selected.
```

- 回滚删除表的事务后

回滚事务A后如下事务B查询执行事务A之前的t1表即回滚事务A后事务B查询的数据不受回滚事务的影响

  - 事务B：回滚事务A时事务B查询执行事务A之前的状态信息

```
gSQL> SELECT * FROM t1;


ID NAME
-- ------
 1 leekmo


1 row selected.
```

对象的变更与事务

与创建删除表相同更改表结构的ALTER TABLE语句也保证事务的原子性（atomicity）与一致性（consistency）如下在表中添加column的事务被提交之前其他事务查询到的是原有的表即与UPDATE语句相同ALTER TABLE语句也在事务被提交之前其他事务查询到执行DDL之前的信息

- 事务A：添加新的UPDATE_TIME column

```
gSQL> ALTER TABLE t1 ADD COLUMN ( update_time TIMESTAMP DEFAULT
```

```
CURRENT_TIMESTAMP );


Table altered.
```

- 查询到包含已添加的column的t1表

```
gSQL> select * from t1;


ID NAME   UPDATE_TIME
-- ------ --------------------------
 1 leekmo 2014-07-10 12:50:33.540495


1 row selected.
```

如下提交事务A之前执行事务B查询到的是添加column之前的表

- 事务B：未提交事务A因此无法查询到已添加的column UPDATE_TIME

```
gSQL> SELECT * FROM t1;


ID NAME
-- ------
 1 leekmo


1 row selected.
```

2.2 Data Manipulation Language

DML相关语句

详细内容参考如下

- INSERT相关语句
  - **INSERT INTO**
  - **INSERT INTO name RETURNING**
  - **INSERT INTO name RETURNING .. INTO**
- UPDATE相关语句
  - **UPDATE**
  - **UPDATE name RETURNING**
  - **UPDATE name RETURNING .. INTO**
  - **UPDATE name WHERE CURRENT OF cursor_name**
- DELETE相关语句
  - **DELETE FROM**
  - **DELETE FROM name RETURNING**
  - **DELETE FROM name RETURNING .. INTO**
  - **DELETE FROM name WHERE CURRENT OF cursor_name**
- SELECT相关语句: **SELECT .. INTO**
- Dynamic SQL相关语句
  - **EXECUTE IMMEDIATE 'sql_string'**
  - **PREPARE statement_name**

- **EXECUTE statement_name**

DML概念

数据库操作语言（Data Manipulation LanguageDML)是插入（INSERT）删除（DELETE）更新（UPDATE）等操作表数据或查询表数据的SQL语言

关于查询参考**Data Query Language**本节仅介绍更新数据的DML语句

DDL语句是更改SQL对象结构的语言相反DML是操作对象内容的语言例如ALTER TABLE语句更改表的结构而INSERT语句是在表中添加一条以上的行的语句

插入删除更新表数据等操作数据的语句主要分为如下

区分	语句类型	说明
插入	INSERT .. VALUES	在表中添加一行
	INSERT .. SELECT	在表中添加查询结果
	INSERT .. RETURN .. INTO	将添加的行的值设置为变量
	INSERT .. RETURN	以查询结果搜索添加的行
删除	DELETE .. WHERE	删除满足条件的行
	DELETE .. WHERE CURRENT OF	删除游标位置的行

区分	语句类型	说明
	DELETE .. RETURN .. INTO	将删除的行的值设置为变量
	DELETE .. RETURN	以查询结果搜索删除的行
更新	UPDATE .. WHERE	更新满足条件的行
	UPDATE .. WHERE CURRENT OF	更新游标位置的行
	UPDATE .. RETURN .. INTO	将更新的行的值设置为变量
	UPDATE .. RETURN	以查询结果搜索更新的行

Table 2-2 数据操作语句

插入数据

在表中插入数据时以行为单位插入数据通过INSERT语句可插入一条以上的行即使省略部分列的数据也可以在r行的所有列上以完整的形式将行插入到表中

例如假设有如下表

```
CREATE TABLE t1
(
```

```
    id   NUMBER(10,0),
    name VARCHAR(128),
    addr VARCHAR(1024) DEFAULT 'n/a'
);
```

最基本的插入行的方式如下

```
INSERT INTO t1 VALUES ( 1, 'leekmo', 'Seoul, Korea' );
```

按照创建表时列出的列顺序输入VALUES语句中列出的值但在如上述语句插入或删除列时会出现异常报错因此如下语句最好指定列名称

```
INSERT INTO t1 (id, name, addr) VALUES ( 1, 'leekmo', 'Seoul, Korea' );
INSERT INTO t1 (name, addr, id) VALUES ( 'leekmo', 'Seoul, Korea', 1 );
```

以上两个INSERT语句只是column的顺序不同但插入了拥有相同数据的行

如果未列出表的所有列则在未指定的列设置默认值后完成行如下在语句内未使用的addr column中存储创建表时定义的默认值'n/a'值

```
INSERT INTO t1 ( id, name ) VALUES ( 1, 'leekmo' );
INSERT INTO t1 ( id, name ) SELECT id, name FROM emp;
```

如果要明确使用列的默认值时可如下指定默认值

```
INSERT INTO t1 ( id, name, addr ) VALUES ( 1, 'leekmo', DEFAULT );
```

要对所有列使用默认值时可使用以下两种形式的语句

```
INSERT INTO t1 ( id, name, addr ) VALUES ( DEFAULT, DEFAULT, DEFAULT );

INSERT INTO t1 DEFAULT VALUES;
```

如下可使用一个INSERT语句插入多行以下为使用一个INSERT语句插入3个新的行的例子

```
INSERT INTO t1 (id, name, addr) VALUES
  ( 1, 'leekmo', 'Seoul, Korea' ),
  ( 2, 'mkkim', 'Seoul, Korea' ),
  ( 3, 'xcom', 'Inchon, Korea' );
```

如下也可以使用SELECT查询结果插入多行以下为查询入职3年以上的员工数据并插入到t1表的例子

```
INSERT INTO t1 ( id, name, addr )
SELECT id, name, addr
  FROM emp
 WHERE DATEDIFF( YEAR, SYSDATE, join_date ) >= 3;
```

删除数据

与插入数据相同删除数据也以行为单位删除删除行时可以使用WHERE条件或行的id（ROWID）删除

以下为删除满足WHERE条件的行的示例

```
DELETE FROM t1 WHERE id = 1;
```

以下为使用ROWID值删除该行的示例

```
gSQL> SELECT rowid FROM t1 WHERE id = 1;


                  ROWID
-----------------------
AAAAAAAAFNHAACAAAAAiAAA


1 row selected.


gSQL> DELETE FROM t1 WHERE ROWID = 'AAAAAAAAFNHAACAAAAAiAAA';


1 row deleted.
```

如下无WHERE子句的DELETE语句删除表的所有行无WHERE子句的DELETE语句与TRUNCATE TABLE语句相同均删除所有行但建议使用TRUNCATE TABLE语句

```
DELETE FROM t1;
TRUNCATE TABLE t1;
```

更新数据

使用UPDATE语句更新数据可以更新一个以上的行也可更新一个列或多个列不影响未在UPDATE语句记述的其他列

以下为更新满足条件的行的一个列的示例

```
UPDATE t1 SET page_view = page_view + 1 WHERE id = 1;
```

以下为更新多个列的UPDATE语句两个语句的功能相同

```
UPDATE t1 SET page_view = page_view + 1, status = 'F' WHERE id = 1;

UPDATE t1 SET (page_view, status) = (page_view + 1, 'F') WHERE id = 1;
```

将列值设置为默认值时如下使用DEFAULT

```
UPDATE t1 SET addr = DEFAULT WHERE id = 1;
```

使用游标操作数据

游标是执行查询并操作查询结果的会话对象可以使用游标更新或删除查询结果的集合

以下为声明更新游标（updatable cursor）并使用此游标更新或删除当前游标位置的行的示例

```
gSQL> DECLARE cur1 CURSOR FOR SELECT id, data FROM t1 FOR UPDATE;


Cursor declared.


gSQL> OPEN cur1;


Cursor is open.


gSQL> \var v_id INTEGER
```

```
gSQL> \var v_data VARCHAR(128)

gSQL> FETCH cur1 INTO :v_id, :v_data;


V_ID V_DATA

---- ------

   1 data_1


1 row fetched.


gSQL> FETCH cur1 INTO :v_id, :v_data;


V_ID V_DATA

---- ------

   2 data_2


1 row fetched.


gSQL> DELETE FROM t1 WHERE CURRENT OF cur1;


1 row deleted.


gSQL> FETCH cur1 INTO :v_id, :v_data;


V_ID V_DATA

---- ------
```

```
   3 data_3


1 row fetched.


gSQL> UPDATE t1 SET id = id + :v_id WHERE CURRENT OF cur1;


1 row updated.


gSQL> COMMIT;


Commit complete.


gSQL> SELECT * FROM t1 ORDER BY 1;


ID DATA
-- ------
 1 data_1
 6 data_3


2 rows selected.
```

如上通过**DECLARE cursor_name**语句声明FOR UPDATE游标再通过**OPEN cursor_name**语句打开该游标通过**FETCH cursor_name**语句移动游标位置并通过**DELETE FROM name WHERE CURRENT OF cursor_name**语句删除该位置的行或通过**UPDATE name WHERE CURRENT OF cursor_name**语句更新行

FOR UPDATE游标可使用**CLOSE cursor_name**语句关闭或执行提交时与事务同时关闭

DML Query

执行更新数据的DML语句时可以使用RETURNING子句查询更新的数据DML的RETURNING子句与SELECT相同可以查询数据因此可以用一个DML Query代替DML与SELECT两个语句的执行

以下为通过**INSERT INTO name RETURNING**语句插入数据并查询对应结果的示例如下可用一个语句查询通过SYSDATE函数插入的join_date值

```
gSQL> INSERT INTO t1 ( id, join_date )  VALUES ( 1, SYSDATE ) RETURNING
id, join_date;


ID JOIN_DATE
-- ----------
 1 2014-07-18


1 row created.
```

以下为通过**DELETE FROM name RETURNING**语句删除数据并查询已删除的行的示例可以在RETURNING子句中使用运算操作数据

```
gSQL> DELETE FROM t1 RETURNING ( id || ': ' || join_date ) AS
id_and_join_date;


ID_AND_JOIN_DATE
```

```
----------------

1: 2014-07-18


1 row deleted.
```

以下为使用**UPDATE name RETURNING**语句以更新的row值为准查询的示例使用OLD子句时可以查询变更之前的值

- 更新行并查询已更新的值

```
gSQL> UPDATE t1 SET page_view = page_view + 1 WHERE id = 1 RETURNING
page_view;


PAGE_VIEW
---------
      102


1 row updated.
```

- 更新行并查询更新之前的值

```
gSQL> UPDATE t1 SET page_view = page_view + 1 WHERE id = 1 RETURNING OLD
page_view;


PAGE_VIEW
---------
```

```
      102


1 row updated.
```

DML中使用的RETURNING子句与SELECT query相同可查询多条查询结果相反如果要对一个行执行DML时可使用RETURNING INTO子句获取主机变量值此时与**SELECT .. INTO**语句相同更新的行的数量应为一条以下

以下为通过各DML的RETURNING .. INTO子句设置主机变量值的示例

- 声明主机变量

```
gSQL> \var v_id        INTEGER
gSQL> \var v_page_view BIGINT
gSQL> \var v_date      DATE
```

- 插入行后获取主机变量值

```
gSQL> INSERT INTO t1 ( id, join_date ) VALUES ( 1, SYSDATE ) RETURNING
join_date INTO :v_date;


V_DATE
--------------------------
2014-07-18 16:57:11.000000


1 row created.
```

- 更新行后获取主机变量值

```
gSQL> UPDATE t1 SET page_view = page_view + 1 WHERE id = 1 RETURN
page_view INTO :v_page_view;


V_PAGE_VIEW
-----------
        101


1 row updated.
```

- 删除行后获取主机变量值

```
gSQL> DELETE FROM t1 WHERE id = 1 RETURN id, page_view
INTO :v_id, :v_page_view;


V_ID V_PAGE_VIEW
---- -----------
   1         101


1 row deleted.
```

DML query相关详细内容参考以下语句

- **INSERT INTO name RETURNING**
- **INSERT INTO name RETURNING .. INTO**

- **DELETE FROM name RETURNING**
- **DELETE FROM name RETURNING .. INTO**
- **UPDATE name RETURNING**
- **UPDATE name RETURNING .. INTO**

2.3 Data Query Language

Query相关语句

相关内容参考如下

- SELECT query相关语句
  - **SELECT**
  - **SELECT .. FOR UPDATE**
- DML query相关语句
  - **INSERT INTO name RETURNING**
  - **UPDATE name RETURNING**
  - **DELETE FROM name RETURNING**
- Cursor相关语句
  - **DECLARE cursor_name**
  - **OPEN cursor_name**
  - **FETCH cursor_name**
  - **CLOSE cursor_name**
  - **DELETE FROM name WHERE CURRENT OF cursor_name**
  - **UPDATE name WHERE CURRENT OF cursor_name**

Query概念

查询（query）是搜索一个以上的表或视图数据的一系列运算操作可使用query从储存的数据中以所需形式获取满足条件的数据

Query是在以';'区分的所有SQL语句中位于最上层的SELECT语句最上层的SELECT语句可在其内包含另一个SELECT语句这时被包含在最上层SELECT语句中的下层SELECT语句叫做子查询（subquery）

SUNDB的查询（query）大体上分为SELECT 查询（query）与DML查询（query）游标（cursor）SELECT 查询是使用SELECT语句返回结果的查询DML 查询是在INSERTDELETEUPDATE语句中使用RETURNING语句返回结果的查询Cursor是临时存储查询一次的结果集并在其结果集中访问任意行读取所需结果的查询SELECT查询与DML查询可通过一次查询直接获取结果而cursor在OPEN cursor执行明确指定为DECLARE cursor的SELECT语句并维持其结果集直到CLOSE cursor调用其结果集并通过FETCH cursor反复访问结果集的任意行并获取所需的结果

本节介绍SELECT queryDML query以及cursor

基本查询

最基本的查询形式为SELECT <select list> FROM <table expression>SELECT关键字与FROM关键字中间的<select list>中指定包含于对 <table expression>的表或视图返回的结果row的一个以上的column或表达式（expression）

```
SELECT n_name
     , INITCAP( n_name )
  FROM nation
 WHERE n_regionkey = 1;



N_NAME                    INITCAP( N_NAME )
------------------------- -------------------------
ARGENTINA                 Argentina
BRAZIL                    Brazil
CANADA                    Canada
PERU                      Peru
UNITED STATES             United States

5 rows selected.
```

<table expression>中可以使用一个以上的表或视图这时两个表或视图中可存在相同的列这时为了在<select list>中描述该列需同时描述该表或视图的名称描述表或列时建议指定模式名称与表名称等

- 错误的示例

```
SELECT n_name
  FROM nation   AS n
     , v_nation AS v
 WHERE n.n_nationkey = v.n_nationkey
```

```
   AND v.n_regionkey = 1;



ERR-42000(16142): column ambiguously defined :
SELECT n_name
       *
ERROR at line 1:
```

- 正确的示例

```
SELECT n.n_name
  FROM nation   AS n
     , v_nation AS v
 WHERE n.n_nationkey = v.n_nationkey
   AND v.n_regionkey = 1;


N_NAME
-------------------------
ARGENTINA
BRAZIL
CANADA
PERU
UNITED STATES


5 rows selected.
```

<select list>支持别名（alias name）可更新用逗号(',')区分的各列中要输出的列的名称 Alias name仅可在<order by clause>中使用无法在其他语句中使用

```
SELECT p_type
     , p_retailprice * 0.9 AS discount_price
  FROM part
 ORDER BY discount_price
 FETCH 5;


    2     3     4     5
P_TYPE                 DISCOUNT_PRICE
---------------------- --------------
PROMO BURNISHED COPPER          810.9
ECONOMY BRUSHED NICKEL          810.9
LARGE BRUSHED BRASS             811.8
LARGE BRUSHED NICKEL            811.8
PROMO ANODIZED STEEL            811.8


5 rows selected.
```

SELECT关键字与FROM关键字中间除<select list>外还可以使用<hint clause>与<set quantifier><hint clause>为用户直接控制Query执行计划的语句详细内容参考**SQL Hint**通过<set quantifier>语句可以删除结果row的重复数据详细内容参考**query specification**

- hint使用示例

```
SELECT
       /*+ INDEX( part ) */
       p_type
     , p_retailprice
  FROM part
 WHERE p_partkey = 100;


P_TYPE               P_RETAILPRICE
-------------------- -------------
ECONOMY ANODIZED TIN        1000.1


1 row selected.
```

- <set quantifier>使用示例

```
SELECT DISTINCT
       o_orderpriority
  FROM orders;


O_ORDERPRIORITY
---------------
5-LOW
2-HIGH
3-MEDIUM
1-URGENT
4-NOT SPECIFIED
```

```
5 rows selected.
```

SET运算符

SET运算符将两个以上的查询（query）的结果集结合为一个结果集SET运算符有UNIONEXCEPTINTERSECT等也提供与EXCEPT拥有相同功能的MINUS运算符每个SET运算符可以有ALL与DISTINCT等追加选项省略时视为DISTINCT

使用SET运算符记述两个以上的query时通常是从左侧query开始向右按照顺序处理使用括号明确指定处理顺序时从该query开始处理

```
SELECT n_name
  FROM nation
 WHERE n_nationkey < 10
INTERSECT
( SELECT n_name
    FROM nation
   WHERE n_regionkey = 1
  UNION ALL
  SELECT n_name
    FROM nation
   WHERE n_regionkey = 2 );


N_NAME
```

```
-------------------------
BRAZIL
ARGENTINA
INDONESIA
INDIA
CANADA


5 rows selected.
```

SET运算符中的每个查询要有相同数量的目标（target）而且每个查询的同一个位置的target应拥有相同组的数据类型

SET运算符拥有排列最终结果集的<order by clause>而且SET运算符的每个查询也可以拥有排列自身查询的<order by clause>

SET运算符的详细内容参考**set operator**

Common Table Expression (CTE)

通过<with clause>构成的临时结果集叫做Common Table Expression (CTE)在语句定义的CTE可在执行范围内参照在<with clause>构成一个以上的CTE各个CTE可参照并构成包含自身的CTE为了构成临时结果集而反复执行查询叫做recursive subquery factoring

CTE分为recursive CTE和non-recursive CTE

在CTE内参照当前定义的CTE（(self-reference CTE）时叫做recursive CTE非recursive CTE时叫

做non-recursive CTE

```
* recursive CTE
WITH RECURSIVE_CTE ( c1 ) AS
     (
          SELECT 1
            FROM dual
          UNION ALL
          SELECT c1 + 1
            FROM RECURSIVE_CTE     ❶ Self reference
           WHERE c1 < 10
     )
SELECT c1 FROM RECURSIVE_CTE;
```

```
* non-recursive CTE
WITH NON_RECURSIVE_CTE ( c1 ) AS
     (
          SELECT i1
            FROM t1
          UNION ALL
          SELECT i1
            FROM t2
     )
SELECT c1 FROM NON_RECURSIVE_CTE;
```

Recursive CTE

Recursive CTE始终由使用UNION ALL的两个query block的集合组成包含self-reference CTE的query block叫做recursive member query其余query block叫做anchor member query

```
WITH CTE_RECURSIVE( c1, c2 ) AS
    (
        SELECT i1, i2                                  ❶ Anchor member query
          FROM t1
         WHERE i2 IS NULL
        UNION ALL
        SELECT i1, i2                                  ❷ Recursive member query
          FROM CTE_RECURSIVE, t1
         WHERE CTE_RECURSIVE.c1 = t1.i2
    )
SELECT c1, c2 FROM CTE_RECURSIVE;
```

Recursive CTE由从anchor member query获取的record组成临时结果集在recursive member query参照CTE获取该record另外recursive CTE也将由从recursive member query获取的结果组成临时结果集并再次尝试执行recursive member query反复此过程直到无法再组成临时结果集

使用<search clause>可指定在当前阶段组成的临时结果集的record组成顺序<search clause>支持DEPTH FIRST BY方式和BREADTH FIRST BY方式<search clause>只能在recursive CTE中描述

详细内容参考**<search clause>**

```
gSQL> SELECT * FROM t1;
```

```
I1  I2

--- ---

A   ---

AA    A

AB    A

AC    A

AAX   AA

ABX   AB

ACX   AC

7 rows selected.


* SEARCH BREADTH FIRST BY

gSQL> WITH w1( w_i1, w_i2 ) AS

    (

         SELECT i1, i2

           FROM t1

          WHERE i1 = 'A'

         UNION ALL

         SELECT i1, i2

           FROM w1, t1

          WHERE w_i1 = i2

    ) SEARCH BREADTH FIRST BY w_i1, w_i2 SET w_seq

SELECT w_i1, w_i2, w_seq

 FROM w1;

W_I1 W_I2 W_SEQ
```

```
---- ---- -----
A    ---      1
AA   A        2
AB   A        3
AC   A        4
AAX  AA       5
ABX  AB       6
ACX  AC       7
7 rows selected.


* SEARCH DEPTH FIRST BY
gSQL> WITH w1( w_i1, w_i2 ) AS
    (
         SELECT i1, i2
           FROM t1
          WHERE i1 = 'A'
         UNION ALL
         SELECT i1, i2
           FROM w1, t1
          WHERE w_i1 = i2
    ) SEARCH DEPTH FIRST BY w_i1, w_i2 SET w_seq
SELECT w_i1, w_i2, w_seq
 FROM w1;
W_I1 W_I2 W_SEQ
---- ---- -----
```

```
A    ---      1
AA   A        2
AAX  AA       3
AB   A        4
ABX  AB       5
AC   A        6
ACX  AC       7
7 rows selected.
```

将在上一个阶段已组成的结果再次组成为当前的临时结果集时recursive CTE无限反复执行此时系统会判断为出现cycle而产生cycle detected error

通过<cycle clause>可选定用于判断是否产生cycle的比较对象也可以查看是否出现cycle使用<cycle clause>时在出现cycle时cycle detected不报错

未明示<cycle clause>时将定义CTE时使用的所有列选定为判断是否出现cycle的对象

```
gSQL> SELECT * FROM t1;
I1  I2
--- ---
A   ---
AA  A
AB  A
AC  A
AA  AA
AAX AA
```

```
ABX AB

ACX AC

8 rows selected.
```

- 出现cycle并未描述cycle clause的情况

```
gSQL> WITH w1( w_i1, w_i2 ) AS
     (           SELECT i1, i2
            FROM t1
           WHERE i1 = 'A'
          UNION ALL
          SELECT i1, i2
            FROM w1, t1
           WHERE w_i1 = i2
     )
SELECT w_i1, w_i2
  FROM w1;
ERR-42000(16511): cycle detected while executing recursive WITH query
```

- 出现cycle并描述cycle clause的情况

```
gSQL> WITH w1( w_i1, w_i2 ) AS
     (
          SELECT i1, i2
            FROM t1
           WHERE i1 = 'A'
```

```
          UNION ALL
          SELECT i1, i2
            FROM w1, t1
           WHERE w_i1 = i2
     ) CYCLE w_i1, w_i2 SET c_cycle TO 'T' DEFAULT 'F'
SELECT w_i1, w_i2, c_cycle
  FROM w1;
W_I1 W_I2 C_CYCLE
---- ---- -------
A    ---  F
AC   A    F
AB   A    F
AA   A    F
ACX  AC   F
ABX  AB   F
AAX  AA   F
AA   AA   F
AAX  AA   F
AA   AA   T
10 rows selected.
```

Join

Join是结合<from clause>中的两个以上的表或视图的各个row的查询无条件的join运算返回两个

表或视图的左侧结果的所有row与右侧结果的所有row结合各自结合为一个row的row

当<from clause>的两个以上的表或视图执行Join时两个表或视图中存在相同名称的列时需要在<select list>与<where clause>等语句中使用表名称等明确加以区分否则会报验证错误（validation error）

执行Join可分为有Join条件的情况与无Join条件的情况所谓Join条件是指对比参与join的互不相同的两个表或视图的column的条件无Join条件时返回将两个表或视图的每个row结合为一个row的结果有Join条件时在两个表或视图的各个row中将满足Join条件的row结合为一个row后返回

Join条件中有使用同等比较(=)的条件时称为equi-joinJoin条件中属于equi-join的条件是优化器优化join的重要要素

Join运算中<from clause>里仅存在相同的表时称为self-join为了在<select list>等中使用列而在各表中记述别名并在列中使用表的别名

CROSS JOIN

CROSS JOIN是无join条件的join运算又称笛卡尔积（Cartesian Product）CROSS JOIN对表或视图的row返回与各个其他表或视图的row结合的结果

```
SELECT a.r_name
     , b.r_name
  FROM region AS a
     , region AS b
 FETCH 5;
```

```
R_NAME                    R_NAME
------------------------- -------------------------
AFRICA                    AFRICA
AFRICA                    AMERICA
AFRICA                    ASIA
AFRICA                    EUROPE
AFRICA                    MIDDLE EAST

5 rows selected.
```

INNER JOIN

INNER JOIN是对两个以上的表或视图返回满足join条件的row的join运算INNER JOIN指在<from clause>中指定inner join的情况与在<from clause>中用逗号(,)list列出表或视图并在<where clause>中描述其表或视图的Join条件的情况在<from clause>中指定inner join时如果<where clause>中也有join条件则不加以区分并组合成一个join条件进行处理

- INNER JOIN语句的使用示例

```
SELECT n_name
  FROM region INNER JOIN nation ON r_regionkey = n_regionkey
 WHERE r_name = 'AFRICA';

N_NAME
-------------------------
```

```
ALGERIA

ETHIOPIA

KENYA

MOROCCO

MOZAMBIQUE


5 rows selected.
```

- 使用逗号(,)list列出的例子

```
SELECT n_name

  FROM region

     , nation

 WHERE r_regionkey = n_regionkey

   AND r_name = 'AFRICA';


N_NAME

-------------------------

ALGERIA

ETHIOPIA

KENYA

MOROCCO

MOZAMBIQUE


5 rows selected.
```

OUTER JOIN

OUTER JOIN返回满足两个以上的表或视图的Join条件的row并根据OUTER JOIN的方向返回不满足单方向或双方向的表或视图的Join条件的row

OUTER JOIN分为LEFT OUTER JOINRIGHT OUTER JOIN与FULL OUTER JOIN三种OUTER JOIN均返回满足join条件的row此外LEFT OUTER JOIN返回以左row为准对右row不满足条件的部分均填充NULL的结果RIGHT OUTER JOIN以右row为准对左row不满足条件的部分均填充NULL的结果FULL OUTER JOIN返回LEFT OUTER JOIN与RIGHT OUTER JOIN的所有附加结果

```
SELECT r_name
     , n_name
  FROM region LEFT OUTER JOIN nation
       ON r_regionkey = n_regionkey AND r_name = 'AFRICA';


R_NAME                    N_NAME
------------------------- -------------------------
AFRICA                    ALGERIA
AFRICA                    ETHIOPIA
AFRICA                    KENYA
AFRICA                    MOROCCO
AFRICA                    MOZAMBIQUE
AMERICA                   null
ASIA                      null
EUROPE                    null
MIDDLE EAST               null
```

```
9 rows selected.
```

SUNDB为了与Oracle的兼容性支持在SQL标准不支持但在Oracle支持的外连接operator(+)外连接operator(+)在<from clause>中以逗号(,)区分列出表在<where clause>中outer node的column添加'(+)'符号

使用outer join operator(+)时如下应在列的右侧添加符号（+）

```
select * from t1, t2 where t1.i1 = t2.i1(+);
```

使用outer join operator(+)的语句规则如下

- 除<where clause>外不可使用<join outer operator>
- <join outer operator>仅可对<joined table>对象的<table reference>的<column reference>使用
- 无法与包含<join outer operator>的<value expression>和使用OR logical operator的其他条件相结合
- 无法将包含<join outer operator>的<column reference>作为IN function的参数使用
- 一个<table reference>不能用作多个outer join的null-generated table（Outer join的约束条件）
- 执行两个表以上的outer join时以left outer join顺序罗列从最左侧开始执行outer join
- 执行两个表以上的outer join一个表与多个表outer join时按照优化器的计算顺序执行outer join

如下情况即使指定外连接运算符（outer join operator）(+)也会被忽略

- <join outer operator>可用于可作为两个表之间的连接条件（join condition）的<value expression>如果不能用作连接条件时被忽略不出现error或warning
- 用于outer query的<column reference>中的<join outer operator>被忽略不出现error或warning
- 使用<join outer operator>对两个<table reference>执行outer join时属于null-generated table的所有<column reference>中需使用<join outer operator>否则两个<table reference>的join会识别为inner join不出现error或warning

SUNDB与Oracle的outer join operator(+)有以下区别

- 量化对比（quantified comparison ）(in, = any, = all = row等)
  - SUNDB：将该运算处理为validation error
  - Oracle：用and/or解析该运算后执行validation检查
- And子句的下层有or子句的情况
  - SUNDB：对下层的or子句的column也适用validation
  - Oracle：在validation中忽略下层的or子句的column
- 条件子句里有子查询(subquery)时
  - SUNDB：用outer join解析并将该条件处理为join condition
  - Oracle：用outer join解析但将该条件处理为where filter

> Note:
>
> SUNDB支持outer join operator(+)功能是为了与Oracle的兼容性建议使用在<from clause>中指定OUTER JOIN的方法（Oracle也同样推荐在<from clause>中指定OUTER JOIN）

NATURAL JOIN

NATURAL JOIN是对于两个以上的表或视图中相同名称的column使用同等对比(=)条件作为join条件的join运算除在内部生成并使用对相同名称的column的join条件外其余与INNER JOIN相同

```
SELECT r_name
  FROM region a NATURAL JOIN region b;


R_NAME
-------------------------
AFRICA
AMERICA
ASIA
EUROPE
MIDDLE EAST


5 rows selected.
```

SEMI JOIN

SEMI JOIN运算作为结果返回右侧row中满足join条件的row的左侧row与结合左侧的row与右侧的row返回结果的其他join运算不同SEMI JOIN仅返回左侧的row

- semi join的使用示例

```
SELECT r_name
```

```
  FROM region
 WHERE r_regionkey IN ( SELECT n_regionkey
                          FROM nation
                         WHERE n_nationkey < 5 );


R_NAME
-------------------------
AFRICA
AMERICA
MIDDLE EAST


3 rows selected.
```

ANTI-SEMI JOIN

ANTI-SEMI JOIN运算作为结果返回右侧row中不满足join条件的row的左侧rowANTI-SEMI JOIN与SEMI JOIN相同作为结果仅返回左侧的row

- Anti-semi join的使用示例

```
SELECT r_name
  FROM region
 WHERE r_regionkey NOT IN ( SELECT n_regionkey
                              FROM nation
                             WHERE n_nationkey < 5 );
```

```
R_NAME

-------------------------

ASIA

EUROPE


2 rows selected.
```

JOIN相关详细内容参考**joined table**

层次查询(Hierarchical Query)

层次查询(hierarchical query)是可处理层次模式数据的查询层次模式数据拥有连接条件并组成层次关系（hierarchical relationship）

可使用recursive CTE表示层次查询详细内容参考**Recursive CTE**

组成层次查询的其他方法是使用<hierarchical query clause>

<hierarchical query clause>使用给予的开始条件(<start with clause>)和下级连接条件(<connect by clause>)创建层次结构并以depth-first方式组成结果record<hierarchical query clause>必须包含<connect by clause>

```
SELECT *
  FROM r_region
 WHERE r_population > 10000000
 START WITH r_name = 'EARTH'              ❶ 开始条件
```

```
CONNECT BY r_domain = PRIOR r_name          ❷ 连接条件
```

在<connect by>描述的expression中用作PRIOR运算符因子的所有expression组成为当前层次的结果只有组成为结果的expression用作查看是否出现cycle的对象由此组成的结果可参考 **<hierarchy expression>**

在层级查询中查看是否出现cycle以当前组成的结果为准反复搜索上级层次并查看是否存在相同的结果以当前结果为准判断为出现cycle时将信息设置为当前结果record的上级record中包含出现cycle的record

搜索到拥有cycle产生信息的record时系统判断为出现cycle并报cycle detected error在<connect by>语句描述NOCYCLE时不报cycle detected error可通过<hierarchical expression>中一个的CONNECT_BY_ISCYCLE查看是否出现cycle

```
gSQL>
SELECT * FROM t1;


I1 I2
-- ----
A  null
AA A
AB A
AC A
AA AA
AB AA
```

```
6 rows selected.


gSQL>
SELECT i1, i2, CONNECT_BY_ISCYCLE
  FROM t1
START WITH i1 = 'A'
CONNECT BY NOCYCLE i2 = prior i1;


I1 I2   CONNECT_BY_ISCYCLE
-- ---- ------------------
A  null                  0
AA A                     1
AB AA                    0
AB A                     0
AC A                     0


5 rows selected.
```

拥有相同上级（parent）record的sibling record通过<order sibling by clause>排列组成按照各个层次的结果record时应用<order sibling by clause>

而形式与此相似的<order by clause>对从query block获取的所有record进行排序

因此<order sibling by clause>和<order by clause>互不影响其之间也没有使用约束

```
gSQL>
SELECT * FROM t1;
```

```
I1  I2

--- ----

A   null

AA  A

AB  A

fAA AA

eAA AA

bAA AA

dAB AB

cAB AB

aAB AB


9 rows selected.
```

- 以下为指定拥有相同parent record的sibling record之间的fetch顺序的示例

```
gSQL>

SELECT LEVEL, i1, i2

  FROM t1

START WITH i1 = 'A'

CONNECT BY i2 = PRIOR i1

ORDER SIBLINGS BY i1;


LEVEL I1  I2

----- --- ----

    1 A   null
```

```
    2 AA  A

    3 bAA AA

    3 eAA AA

    3 fAA AA

    2 AB  A

    3 aAB AB

    3 cAB AB

    3 dAB AB


9 rows selected.
```

- 以下为使用ORDER BY子句按照LEVEL顺序对通过层次结构检索的所有结果进行排列的示例

```
gSQL>
SELECT LEVEL, i1, i2
  FROM t1
START WITH i1 = 'A'
CONNECT BY i2 = PRIOR i1
ORDER SIBLINGS BY i1
ORDER BY LEVEL;


LEVEL I1  I2
----- --- ----
    1 A   null
    2 AA  A
    2 AB  A
```

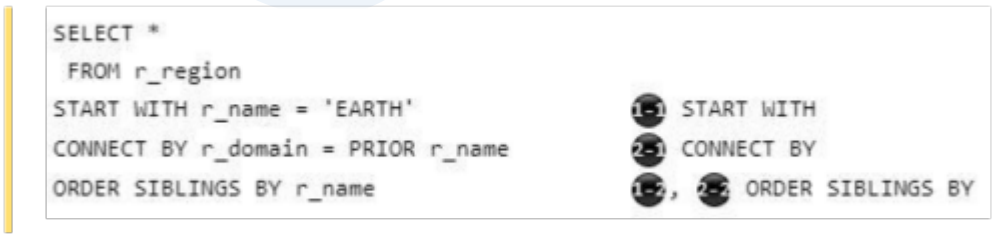
```
    3 bAA AA

    3 eAA AA

    3 fAA AA

    3 aAB AB

    3 cAB AB

    3 dAB AB


9 rows selected.
```

层级查询的语句评估顺序

<hierarchical query clause>由<start with connect by clause>和<order siblings by clause>组成

<hierarchical query clause>内的语句按照以下顺序执行

通过<order siblings by clause>排列通过<start with connect by clause>从各个层次获取的结果对最上级层次评估<start with clause>并应用<order siblings by clause>对后续组成的下级层次使用<connect by clause>和<order siblings by clause>组成结果



Query block内的<hierarchical query clause>按照如下顺序执行

<hierarchical query clause>在<from clause>后描述在<where clause>和<group by clause>之间与<hierarchical query clause>的描述顺序不同在<from clause>之后<where clause>之前评估<hierarchical query clause>

描述在<where clause>的条件不影响评估<hierarchical query clause>

以下为根据<from clause>和<where clause>构成的<hierarchical query clause>的评估顺序

- From子句中只有单张表时

```
SELECT *
 FROM r_region
WHERE r_population > 10000000              ❸ WHERE子句的条件
START WITH r_name = 'EARTH'                ❶ START WITH
CONNECT BY r_domain = PRIOR r_name         ❷ CONNECT BY
```

- From子句中有由多张表组成的join条件时
  - 在FROM子句的ON子句描述join条件时

```
SELECT *
  FROM r_region INNER JOIN s_region
       ON r_id = s_id                       ❶ ON 子句的join条件
 WHERE r_population > 10000000              ❹ WHERE 子句的条件
START WITH r_name = 'EARTH'                 ❷ START WITH
CONNECT BY r_domain = PRIOR r_name          ❸ CONNECT BY
```

  - 在WHERE子句描述join条件时

```
SELECT *
  FROM r_region, s_region
 WHERE r_population > 10000000          ❸ WHERE 子句的条件
   AND r_id = s_id                      ❸ WHERE 子句的 join条件
START WITH r_name = 'EARTH'             ❶ START WITH
CONNECT BY r_domain = PRIOR r_name      ❷ CONNECT BY
```

- 在FROM子句的ON子句和WHERE子句均描述join条件时

```
SELECT *
  FROM r_region INNER JOIN s_region
       ON r_name = s_name               ❶ ON子句的join条件
 WHERE r_population > 10000000          ❹ WHERE 子句的条件
   AND r_id = s_id                      ❹ WHERE 子句的join条件
START WITH r_name = 'EARTH'             ❷ START WITH
CONNECT BY r_domain = PRIOR r_name      ❸ CONNECT BY
```

在包含<hierarchical query clause>的query block中使用<group by clause>时可使用<hierarchical expression>组成结果集集合

```
SELECT COUNT(*)
  FROM r_region
 START WITH r_name = 'EARTH'
CONNECT BY r_domain = PRIOR r_name
 GROUP BY PROIR r_domain
```

结果集集合（group by）

<group by clause>用于以一个以上的cloumn为基准将拥有相同column的row捆绑为一个group进行处理<group by clause>为了区分group以逗号(,)罗列columnSUNDB以此为基准执行对group的运算

```
SELECT
       o_orderpriority
     , MIN( o_totalprice ) AS min_price
     , MAX( o_totalprice ) AS max_price
  FROM orders
 GROUP BY
       o_orderpriority;


O_ORDERPRIORITY MIN_PRICE MAX_PRICE
--------------- --------- ---------
5-LOW              857.71 530604.44
2-HIGH              896.8 522720.61
3-MEDIUM           875.52 508668.52
1-URGENT            866.9 544089.09
4-NOT SPECIFIED    884.82 555285.16


5 rows selected.
```

使用<group by clause>时在<select list>中只能出现<group by clause>中的column与集合函数

<group by clause>中描述非column的常数或仅描述括号时被视为在每行row均有相同值的虚拟column的empty grouping set并以此column为准执行集合运算也适用于没有<group by clause>而仅使用<having clause>的情况

通过<having clause>可以对以<group by clause>区分group的结果中记述仅返回特定row的条件<having clause>的作用是描述各个group的条件可描述使用集合运算的条件等

```
SELECT
       o_orderpriority
     , MIN( o_totalprice ) AS min_price
     , MAX( o_totalprice ) AS max_price
  FROM orders
 GROUP BY
       o_orderpriority
 HAVING
       MIN( o_totalprice ) < 870;


O_ORDERPRIORITY MIN_PRICE MAX_PRICE
--------------- --------- ---------
5-LOW              857.71 530604.44
1-URGENT            866.9 544089.09


2 rows selected.
```

关于group的相关详细内容参考**group by clause**

Window Query

窗口函数（window function）返回定义的记录范围内的函数执行结果定义的记录范围称为窗口（window）在OVER <window name or specification>中定义记录范围与一般函数或聚合函数（aggregate function）不同窗口函数描述OVER子句

```
SUM( sales ) OVER ()

SUM( sales ) OVER window_name

SUM( sales ) OVER ( PARTITION BY item_no
                    ORDER BY sales
                    ROWS BETWEEN UNBOUNDED PRECEDING
                             AND CURRENT ROW )
```

窗口函数与聚合函数相似返回多个记录的函数执行结果但是聚合函数各组返回一条记录而窗口函数各组返回多条记录

窗口函数组内的各记录拥有对窗口(定义的记录范围)执行函数的结果所以与聚合函数不同窗口函数返回各组的所有记录

以下是展示聚合函数和窗口函数的执行结果的示例

- 以下是示例表

```
gSQL>

SELECT * FROM store;


ITEM_NO SALES_DATE SALES
```

```
------- ---------- -----
    100 2001-01-01   150
    100 2001-01-02   100
    100 2001-01-03   170
    100 2001-01-04    90
    100 2001-01-05   200
    235 2001-01-01    70
    235 2001-01-02   130
    235 2001-01-03   190
    235 2001-01-04   150
    235 2001-01-05    50

10 rows selected.
```

- 聚合函数执行结果

```
gSQL>
SELECT SUM( sales ) AS aggrfunc_sum
  FROM store;

AGGRFUNC_SUM
------------
        1300

1 row selected.
```

- 窗口函数执行结果

```
gSQL>
SELECT item_no,
       sales,
       SUM( sales ) OVER () as windowfunc_sum
  FROM store;


ITEM_NO SALES WINDOWFUNC_SUM
------- ----- --------------
    100   150           1300
    100   100           1300
    100   170           1300
    100    90           1300
    100   200           1300
    235    70           1300
    235   130           1300
    235   190           1300
    235   150           1300
    235    50           1300


10 rows selected.
```

窗口函数在OVER <window name or specification>子句中定义窗口(函数的执行范围)

根据<window partition clause> PARTITION BY中的描述进行分组

根据<window order clause> ORDER BY中的描述对组内记录进行排序

根据<window frame clause>中的描述定义组内排序的记录的窗口函数对象记录范围

关于窗口定义的详细内容请参考**window clause**

窗口函数以<window partition clause> PARTITION BY中定义的分区为单位执行省略PARTITION BY时整个结果记录为一个分区

若描述<window order clause> ORDER BY则窗口函数的执行范围window frame应用于分区内各记录(current row)

Window frame通过描述<window frame clause>而定义并同时定义应用单位(ROWS/ RANGE/ GROUPS)起始点结束点和要排除的记录

省略<window frame clause>时默认应用RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW此时应用范围为分区起始记录到当前记录的所有peer记录

<window frame clause>的详细内容请参考**<window frame clause>**

以下为显示省略<window order clause> ORDER BY时和描述<window order clause> ORDER BY后应用window frame时的结果差异的示例

- 省略<window order clause> ORDER BY时
  - 描述PARTITION BY时以指定分区单位计算sum(sales)

```
gSQL>
SELECT item_no,
       sales,
```

```
       SUM( sales ) OVER ( PARTITION BY item_no ) as windowfunc_sum
  FROM store;


ITEM_NO SALES WINDOWFUNC_SUM
------- ----- --------------
    100   150            710
    100   100            710
    100   170            710
    100    90            710
    100   200            710
    235    70            590
    235   130            590
    235   190            590
    235   150            590
    235    50            590


10 rows selected.
```

- 省略PARTITION BY时因所有记录被看作一个分区所以计算所有记录的sum(sales)

```
gSQL>
SELECT item_no,
       sales,
       SUM( sales ) OVER ( ) as windowfunc_sum
  FROM store;
```

```
ITEM_NO SALES WINDOWFUNC_SUM

------- ----- --------------

    100   150           1300

    100   100           1300

    100   170           1300

    100    90           1300

    100   200           1300

    235    70           1300

    235   130           1300

    235   190           1300

    235   150           1300

    235    50           1300


10 rows selected.
```

- 描述<window order clause> ORDER BY时
  - 描述PARTITION BY时：因<window frame clause>被省略所以默认应用RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW计算从分区起始记录到当前记录的所有peer记录的sum(sales)

```
gSQL>

SELECT item_no,

       sales,

       SUM( sales ) OVER ( PARTITION BY item_no

                           ORDER BY sales ) as windowfunc_sum

  FROM store;
```

```
ITEM_NO SALES WINDOWFUNC_SUM
------- ----- --------------
    100    90             90
    100   100            190
    100   150            340
    100   170            510
    100   200            710
    235    50             50
    235    70            120
    235   130            250
    235   150            400
    235   190            590

10 rows selected.
```

- 省略PARTITION BY时：所有记录被看作一个分区因<window frame clause>被省略默认应用RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW计算从分区起始记录到当前记录的所有peer记录的sum(sales)

```
gSQL>
SELECT item_no,
       sales,
       SUM( sales ) OVER ( ORDER BY sales ) as windowfunc_sum
  FROM store;
```

```
ITEM_NO SALES WINDOWFUNC_SUM

------- ----- --------------

    235    50             50

    235    70            120

    100    90            210

    100   100            310

    235   130            440

    100   150            740

    235   150            740

    100   170            910

    235   190           1100

    100   200           1300


10 rows selected.
```

对同一窗口(定义的记录范围)执行多个窗口函数时在WINDOW子句中定义window name并引用

```
gSQL>
SELECT item_no,
       sales_date,
       sales,
       SUM( sales ) OVER W1 cumulative_sales,
       AVG( sales ) OVER w1 avg_sales
  FROM store
WINDOW w1 AS ( PARTITION BY item_no
               ORDER BY sales_date
```

```
                ROWS BETWEEN UNBOUNDED PRECEDING

                         AND CURRENT ROW );


ITEM_NO SALES_DATE SALES CUMULATIVE_SALES AVG_SALES

------- ---------- ----- ---------------- ---------

    100 2001-01-01   150              150       150

    100 2001-01-02   100              250       125

    100 2001-01-03   170              420       140

    100 2001-01-04    90              510     127.5

    100 2001-01-05   200              710       142

    235 2001-01-01    70               70        70

    235 2001-01-02   130              200       100

    235 2001-01-03   190              390       130

    235 2001-01-04   150              540       135

    235 2001-01-05    50              590       118


10 rows selected.
```

可在select list和order by clause中描述窗口函数

窗口函数执行FROM, WHERE, GROUP BY, HAVING子句执行后的结果集在查询中使用aggregateGROUP BYHAVING子句时则需要在窗口函数中描述组的列而不是原表的列

相关内容请参考**Window Function**和**window clause**

结果集的排序（order by）

要以所需column为基准排列已查询的结果集时使用<order by clause><order by clause>可以以除LONG类型外的所有列为准进行排列

<order by clause>中使用正整数时表示<select list>中target的整数值对应位置的column<order by clause>中使用的正整数的范围为1到<select list>中的target的数量

```
SELECT
       o_orderpriority
     , MIN( o_totalprice ) AS min_price
     , MAX( o_totalprice ) AS max_price
  FROM orders
 GROUP BY
       o_orderpriority
 ORDER BY 1;


O_ORDERPRIORITY MIN_PRICE MAX_PRICE
--------------- --------- ---------
1-URGENT            866.9 544089.09
2-HIGH              896.8 522720.61
3-MEDIUM           875.52 508668.52
4-NOT SPECIFIED    884.82 555285.16
5-LOW              857.71 530604.44
```

```
5 rows selected.
```

<order by clause>中column的数据类型为numeric data时对比数字进行排序；为character data时对比文字进行排序

<order by clause>中的各个column可记述为升序（ASC）或降序（DESC）进行排序省略时默认为升序

排序相关详细内容参考**order by clause**

子查询（Subquery）

子查询（subquery）执行由多阶段构成的搜索请求由多阶段构成的查询请求是指由下层查询结果决定当前查询结果例如在特定群组中搜索比平均年龄大的人员时首先执行求特定群组的平均年龄的查询后再用其执行搜索比平均年龄大的人员的查询

```
SELECT e_name
  FROM emp
 WHERE e_age > ( SELECT AVG(e_age)
                   FROM emp
                  WHERE e_dept = 'RND' );
```

子查询可以用于<from clause>与<where clause>用于<from clause>的子查询叫"内联视图（inline view）"用于<where clause>的子查询叫"嵌套子查询（nested subquery）"

使用嵌套子查询时嵌套子查询中的表或视图的column名可能与包含嵌套子查询的query的表或

视图的column名相同这时嵌套子查询的<select list>中仅使用column名时参考嵌套子查询中的表或视图的column如果嵌套子查询的<select list>中使用了未存在于嵌套子查询的表或视图中的column名时则参考包含嵌套子查询的query的表或视图的column名

```
SELECT r_name
  FROM region
 WHERE EXISTS ( SELECT *
                  FROM nation
                 WHERE n_nationkey < 5             /* nation.n_nationkey
*/
                   AND n_regionkey = r_regionkey ) /* nation.n_regionkey =
region.r_regionkey */
;
```

存在于<where clause>的嵌套子查询可由优化器unnest到包含嵌套子查询的query并处理为SEMI JOIN或ANTI-SEMI JOIN等其为优化嵌套子查询当在INNOT INEXISTSNOT EXISTSquantify operator中有子查询时优化器将计算unnest的成本后进行判断如果用户想强制unnest该嵌套子查询则可以使用嵌套子查询的<hint clause>进行调整

嵌套子查询的unnest的<hint clause>的相关内容参考**SQL Hint**

```
SELECT r_name
  FROM region
 WHERE r_regionkey IN ( SELECT /*+ UNNEST */
                                n_regionkey
                          FROM nation
```

```
                    WHERE n_nationkey < 5 );
```

子查询相关详细内容参考**subquery**

2.4 Control Language

Control Language相关语句

详细内容参考如下

- Transaction control相关语句
  - **COMMIT**
  - **ROLLBACK**
  - **LOCK TABLE**
  - **SAVEPOINT savepoint_specifier**
  - **RELEASE SAVEPOINT savepoint_specifier**
- Session control相关语句
  - **ALTER SESSION SET property_name**
  - **SET SESSION AUTHORIZATION user_identifier**
  - **SET SESSION CHARACTERISTICS AS transaction_mode**
  - **SET TIME ZONE**
  - **SET TRANSACTION transaction_mode**
- System control相关语句
  - **ALTER SYSTEM CHECKPOINT**
  - **ALTER SYSTEM {MOUNT | OPEN} DATABASE**
  - **ALTER SYSTEM [KILL | DISCONNECT] SESSION**
  - **ALTER SYSTEM SET property_name**
  - **ALTER SYSTEM RESET property_name**

- ALTER SYSTEM SWITCH LOGFILE

Transaction Control

事务控制语句用于在事务内管理DML语句与DDL语句产生的变更事项事务控制语句中的COMMIT永久保存变更事项回滚（ROLLBACK）撤销变更事项

事务控制语句可如下进行分类

语句	说明	参考链接
COMMIT	事务的正常结束	**COMMIT**
ROLLBACK	撤销事务	**ROLLBACK**
SAVEPOINT	生成保存点	**SAVEPOINT savepoint_specifier**
RELEASE SAVEPOINT	解除保存点	**RELEASE SAVEPOINT savepoint_specifier**
LOCK TABLE	设置Table-level锁	**LOCK TABLE**
SET TRANSACTION	控制事务的属性（读/写 Isolation Level）	**SET TRANSACTION transaction_mode**
SET CONSTRAINTS	控制可延迟约束条件的检查点	**SET CONSTRAINTS**

Table 2-3 事务控制语句

第一次执行更新数据的DML或变更SQL对象的DDL时自动生成事务但执行SELECT或控制语句时不生成事务

事务的回滚分为全部回滚（Total Rollback）与部分回滚（Partial Rollback）部分回滚又分为显式回滚与隐式回滚全部回滚通过ROLLBACK语句执行将在事务内执行的DMLDDL的所有变更内容恢复至之前的状态

用户如下通过使用savepoint的ROLLBACK语句执行显式部分回滚以下例子中声明存储点sp1与sp2并用其显式执行部分回滚

```
gSQL> CREATE TABLE t1 ( id INTEGER, name VARCHAR(128) );

Table created.

gSQL> COMMIT;

Commit complete.

gSQL> INSERT INTO t1 VALUES ( 1, 'leekmo' );

1 row created.

gSQL> SAVEPOINT sp1;

Savepoint created.
```

```
gSQL> INSERT INTO t1 VALUES ( 2, 'mkkim' );

1 row created.

gSQL> SAVEPOINT sp2;

Savepoint created.

gSQL> INSERT INTO t1 VALUES ( 3, 'xcom' );

1 row created.

gSQL> ROLLBACK TO SAVEPOINT sp2;

Rollback complete.

gSQL> SELECT * FROM t1;

ID NAME
-- ------
 1 leekmo
 2 mkkim

2 rows selected.
```

```
gSQL> ROLLBACK TO SAVEPOINT sp1;


Rollback complete.


gSQL> SELECT * FROM t1;


ID NAME
-- ------
 1 leekmo


1 row selected.


gSQL> ROLLBACK WORK;


Rollback complete.


gSQL> SELECT * FROM t1;


no rows selected.
```

执行语句时报错则仅回滚该语句的变更事项其称为隐式部分回滚以下为隐式部分回滚的例子违反唯一约束条件时仅回滚该INSERT语句并维持事务的之前变更事项

```
gSQL> ALTER TABLE t1 ADD CONSTRAINT t1_uk UNIQUE(id);


Table altered.
```

```
gSQL> COMMIT;

Commit complete.

gSQL> INSERT INTO t1 VALUES ( 4, 'egonspace' );

1 row created.

gSQL> INSERT INTO t1 VALUES ( 1, 'jhkim' );

ERR-40002(16057): unique constraint (PUBLIC.T1_UK) violated

gSQL> SELECT * FROM t1;

ID NAME
-- ---------
 1 leekmo
 2 mkkim
 3 xcom
 4 egonspace

4 rows selected.
```

Session Control

会话为管理访问数据库的用户状态信息的逻辑对象可通过会话控制语句变更会话的属性

会话控制语句分为如下

语句	说明	参考链接
SET SESSION CHARACTERISTICS	控制会话内的事务属性	**SET SESSION CHARACTERISTICS AS transaction_mode**
SET TIME ZONE	变更会话的time zone	**SET TIME ZONE**
SET SESSION AUTHORIZATION	变更会话的用户	**SET SESSION AUTHORIZATION user_identifier**
SET SCHEMA	变更会话的schema	**SET SCHEMA schema_name**
ALTER SESSION SET	变更会话的参数值	**ALTER SESSION SET property_name**

Table 2-4 会话控制语句

事务控制语句SET TRANSACTION语句与会话控制语句SET SESSION CHARACTERISTICS均为控制事务属性的语句但有如下区别SET TRANSACTION语句仅适用于下一个执行的一个事务而SET SESSION CHARACTERISTICS语句适用于该会话中后续产生的所有事务

以下为通过SET TIME ZONE语句变更time zone后使用CURRENT_TIMESTAMP函数获取当前日期

/时间的示例

```
gSQL> SELECT CURRENT_TIMESTAMP FROM DUAL;


CURRENT_TIMESTAMP

---------------------------------

2014-07-21 11:42:49.828276 +09:00


1 row selected.


gSQL> SET TIME ZONE '+00:00';


Session set.


gSQL> SELECT CURRENT_TIMESTAMP FROM DUAL;


CURRENT_TIMESTAMP

---------------------------------

2014-07-21 02:43:03.437940 +00:00


1 row selected.
```

System Control

系统控制语句为管理数据库系统的语句分为如下

语句	说明	参考
ALTER SYSTEM {OPEN|MOUNT} DATABASE	启动数据库	**ALTER SYSTEM {MOUNT | OPEN} DATABASE**
ALTER SYSTEM CHECKPOINT	执行 Checkpoint	**ALTER SYSTEM CHECKPOINT**
ALTER SYSTEM KILL SESSION	强制结束特定会话	**ALTER SYSTEM [KILL | DISCONNECT] SESSION**
ALTER SYSTEM SWITCH LOGFILE	切换日志文件	**ALTER SYSTEM SWITCH LOGFILE**
ALTER SYSTEM SET	设置系统属性	**ALTER SYSTEM SET property_name**
ALTER SYSTEM RESET	删除系统属性	**ALTER SYSTEM RESET property_name**

Table 2-5 系统控制语句

以下为查询已连接数据库的会话后强制结束其中的特定会话的示例

```
gSQL> SELECT USER_NAME, SESSION_ID, SERIAL_NO, SESSION_STATUS,
PROGRAM_NAME FROM V$SESSION WHERE USER_NAME = 'TEST';


USER_NAME SESSION_ID SERIAL_NO SESSION_STATUS PROGRAM_NAME
--------- ---------- --------- -------------- ------------
TEST              62        49 CONNECTED      gsql
TEST              65       109 CONNECTED      gsqlnet
TEST              66       130 CONNECTED      gsql
```

```
3 rows selected.


gSQL> ALTER SYSTEM DISCONNECT SESSION 65, 109;


System altered.
```

2.5 集群的SQL处理

本章介绍集群环境中的多种SQL语句的处理过程

集群的DDL处理

集群的DDL处理过程

SUNDB集群没有单独的meta服务器用户可在构成集群系统的所有集群成员中执行DDL

在集群系统中按照如下步骤执行DDL

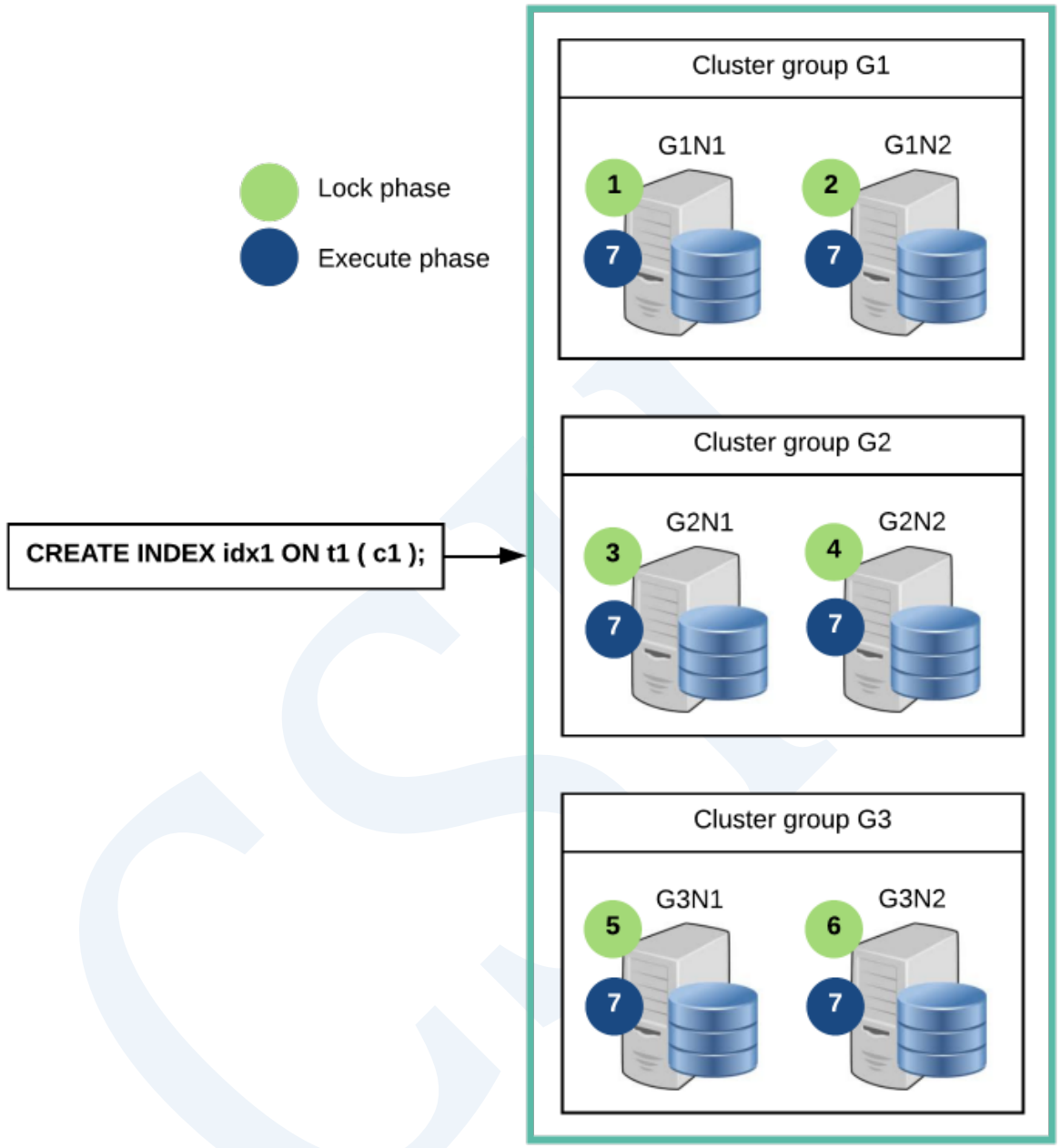


Figure 2-1 集群的DDL处理过程

DDL的执行分为lock phase与execution phase阶段Lock phase为获取执行DDL所需的lock的阶段

对所有集群成员按照顺序执行DDL在Execute phase中对所有集群成员同时处理DDL

在所有集群成员中成功执行DDL时完成DDL在特定集群成员中失败时取消所有集群成员的DDL

操作存在发生故障的集群成员时无法执行DDL所有集群成员通过这种过程同步对象的meta信息

DDL同时执行

无法同时执行变更集群系统构成的集群对象的DDL与SQL对象的DDL是否可同时执行集群对象的DDL与SQL对象的DDL情况如下

区分	集群对象DDL	SQL对象DDL
集群对象DDL	X	X
SQL对象DDL	X	O

Table 2-6 是否可同时执行DDL

如上图所示无法同时执行以下DDL

- 集群对象DDL与集群对象DDL
  - (X) CREATE CLUSTER GROUP g2 CLUSTER MEMBER g2n1 HOST '192.168.0.21' PORT 10210;
  - (X) ALTER CLUSTER GROUP g1 ADD CLUSTER MEMBER g1n2 HOST '192.168.0.12' PORT 10120;
- 集群对象DDL与SQL对象DDL
  - (X) CREATE CLUSTER GROUP g2 CLUSTER MEMBER g2n1 HOST '192.168.0.21' PORT 10210;
  - (X) CREATE TABLE t1 (c1 INTEGER);
- SQL对象DDL与SQL对象DDL
  - (O) CREATE TABLE t1 (c1 INTEGER);

- (O) CREATE TABLE t2 (a1 INTEGER);

集群的SELECT处理

集群中的查询处理基本上与单机版相同但数据不仅存在于本地服务器也存在于remote服务器时向remote服务器请求查询处理后汇总其结果方面存在区别

集群环境中有sharded table与cloned table(参考**14.5 集群表与分片**)各个表数据存储于本地服务器和Remote服务器Sharded table的数据分散存储于group数据复制并存储在相同group的成员中Cloned table的数据复制并存储于所有group与成员中

以下为3 by 2结构的SUNDB的集群与存储在该集群的表

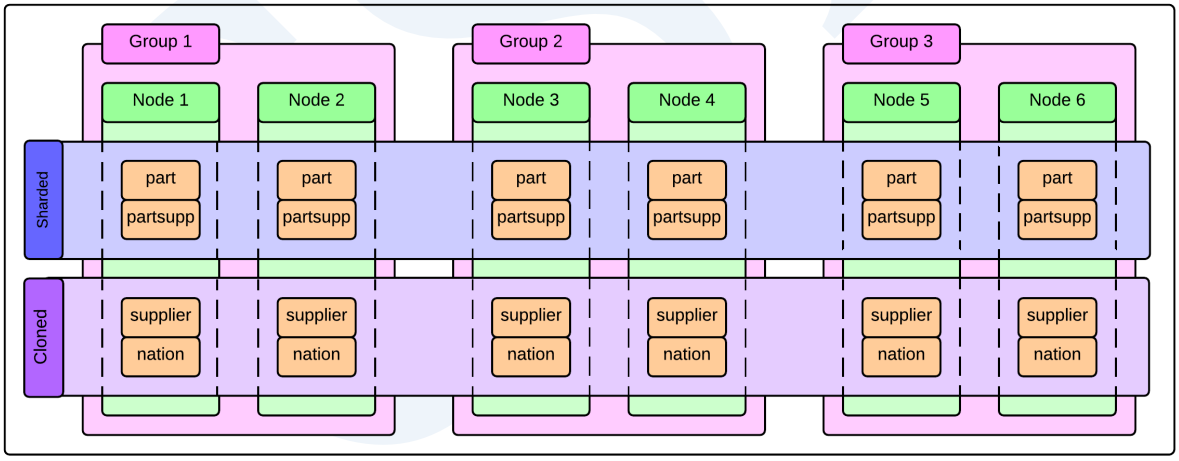


Figure 2-2 3 by 2集群结构与表

以下为创建上图中表的DDL语句

```
CREATE TABLE part
```

```
(
    p_partkey     INTEGER
  , p_name        VARCHAR(55)
  , p_brand       CHAR(10)
  , p_type        VARCHAR(25)
  , p_size        INTEGER
  , p_retailprice NUMERIC(12,2)
  , CONSTRAINT part_pk PRIMARY KEY( p_partkey ) INDEX part_pk_index
)
    SHARDING BY HASH(p_partkey)
    SHARD COUNT 3;

CREATE TABLE partsupp
(
    ps_partkey    INTEGER
  , ps_suppkey    INTEGER
  , ps_availqty   INTEGER
  , ps_supplycost NUMERIC(12,2)
  , CONSTRAINT partsupp_pk PRIMARY KEY( ps_partkey, ps_suppkey ) INDEX
partsupp_pk_index
)
   SHARDING BY HASH(ps_partkey)
   SHARD COUNT 3;
```

```
CREATE TABLE supplier
(
    s_suppkey     INTEGER
  , s_name        CHAR(25)
  , s_nationkey   INTEGER
  , s_phone       CHAR(15)
  , CONSTRAINT supplier_pk PRIMARY KEY( s_suppkey ) INDEX
supplier_pk_index
)  CLONED;



CREATE TABLE nation
(
    n_nationkey   INTEGER
  , n_name        CHAR(25)
)  CLONED;
```

上图中part table与partsupp table为sharded table数据分散存储于各个groupSupplier table和nation table为cloned table数据复制并存储于所有node

SUNDB的集群环境中根据table形式与搜索数据位置的不同查询处理的执行方式会有所不同集群根据汇集数据的方法和操作已获取的数据的方法处理查询

集群的查询处理方法

为了处理集群查询需要从本地服务器和远程服务器都收集数据为了收集数据生成SQL形式的语

句并传输至本地服务器和远程服务器获取查询执行结果并汇集

根据汇集的数据的语句类型执行数据操作例如group by语句的集群查询时收集数据后以执行groping的方式进行处理

数据收集与操作由被称为cluster puller的plan node负责

Cluster Puller

Cluster puller node执行数据收集（collection）和数据操作（manipulation）

- 数据收集是从多个服务器汇集数据的过程
- 数据操作是以汇集的数据为基础导出新数据的过程

下图为执行cluster puller的进程

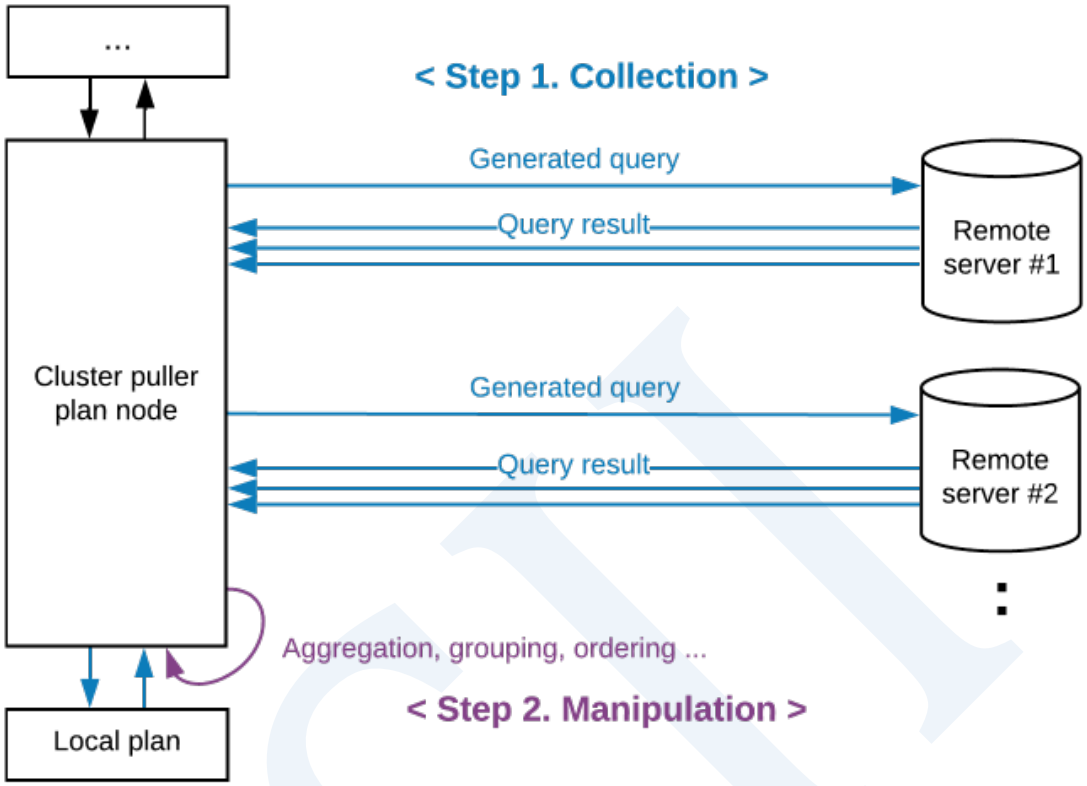


Figure 2-3 执行cluster puller

Cluster puller根据从本地服务器获取数据的方式与是否区分从远程服务器获取数据时传输的远程服务器进行区分

Cluster puller如下分为三种plan node

Cluster puller plan node	Local数据收集方法	Remote数据收集方法
Plan based cluster	执行在local构成的plan	执行generated query（收集结果不区分远程服务器）

Cluster puller plan node	Local数据收集方法	Remote数据收集方法
Single cluster	执行generated query	执行generated query（收集结果不区分远程服务器）
Multiple cluster	执行generated query	执行generated query（按照各个远程服务器进行区分并收集结果）

Table 2-7 Cluster puller plan node

Generated query是在执行用户提供的query的过程中用于收集或更新数据而构成的查询详细内容参考**Generated Query**

在下节说明各个cluster puller node支持的各功能的数据收集方法和数据操作方法

数据收集方法

- By pass：按照接收的顺序汇集执行结果
- Merge sort：按照所指定的排列顺序依次汇集执行结果

已汇集的数据的操作方法

- No manipulation：不执行数据操作
- Aggregation：对汇集的数据执行aggregation
- Grouping：对汇集的数据执行grouping

- Ordering：对汇集的数据执行ordering
- Intersect key group：以接收的服务器为准区分汇集的数据用指定key对各个服务器数据执行grouping后以group为单位执行intersect
- Distinct key group：以接收的服务器为准区分汇集的数据用指定key对各个服务器数据执行grouping后以group为单位执行distinct

Cluster Puller功能

Cluster puller node从本地服务器和远程服务器收集数据决定作为收集对象的服务器后传输generated query并获取数据

以下为没有where条件查询single table的示例

```
gSQL> \EXPLAIN PLAN SELECT c1 FROM t_shard_1;


C1
--
 1
 2
 3


3 rows selected.


>>>  start print plan
```

```
< Execution Plan >
================================================================================
=
|  IDX  |  NODE DESCRIPTION                          |               ROWS
|
--------------------------------------------------------------------------------
-
|    0  |  SELECT STATEMENT                          |                 3
|
|    1  |    QUERY BLOCK ("$QB_IDX_2")               |                 3
|
|    2  |      PLAN BASED CLUSTER                    | LOCAL/REMOTE    3
|
|    3  |        INDEX ACCESS ("T_SHARD_1", "IDX")   | (         1)    1
|
================================================================================
=


     1  -  TARGET : T_SHARD_1.C1
     2  -  SQL : SELECT /*+ INDEX( _A1, "PUBLIC"."IDX" ) */ "_A1"."C1"
FROM "PUBLIC"."T_SHARD_1"@LOCAL AS "_A1"
           TARGET DOMAIN : G1(G1N1,G1N2) 1 rows, G2(G2N1,G2N2) 1 rows,
G3(G3N1,G3N2) 1 rows
     3  -  RANGE SHARD ( # 3 )
           READ INDEX COLUMN : T_SHARD_1.C1
```

```
<<<  end print plan
```

作为执行结果的cluster puller使用了plan based cluster上述输出的执行信息中plan based cluster属于<Execution Plan>的idx为2的plan

输出到cluster puller的ROWS字段的信息为如下

- LOCAL ONLY n: 从local server获取的数据数量
- REMOTE ONLY n: 从remote server获取的数据数量
- LOCAL/REMOTE n: 从local server和remote server获取的数据数量的合计

PLAN BASED CLUSTER的详细信息为如下

- SQL: Generated query
- TARGET DOMAIN：发送Generated query的对象group和member及从对应group收到的数据数量

在Cluster Puller选择执行对象服务器

分析以下信息并决定执行generated query的对象

- 表副本（replica）部署策略
- **Cluster Domain**
- **减少Cluster Puller的Target Domain**

以包含在generated query中的表为对象分析上述列出的信息查找其共同包含的服务器并确定为generated query执行对象服务器Cluster puller node中将其分类为target domain

以下为为了说明**Generated Query**概括了定义的表的副本（replica）部署策略

```
t_shard_1 (shard table)  : at cluster group G1, G2, G3
t_shard_2 (shard table)  : at cluster group G1, G3
t_clone_1 (cloned table) : at cluster group G1, G2, G3 (cluster wide)
t_clone_2 (cloned table) : at cluster group G2, G3
```

以下为根据表副本（replica）部署策略选择执行对象服务器的示例

- 执行对象服务器：G1G3中的所有服务器

```
gSQL> SELECT c1 FROM t_shard_2;
```

- 执行对象服务器：G1G2G3中的所有服务器

```
gSQL> SELECT c1 FROM t_clone_1;
```

- 执行对象服务器：G1G3中的所有服务器（共同包含的集群组）

```
gSQL> SELECT c1 FROM t_shard_2, t_clone_1;
```

以下为根据描述的cluster domain选择执行对象服务器的示例

- 执行对象服务器：G1G2G3中的所有服务器

```
gSQL> SELECT c1 FROM t_shard_1@GLOBAL;
```

- 执行对象服务器：收到用户query请求的服务器

```
gSQL> SELECT c1 FROM t_shard_1@LOCAL;
```

- 执行对象服务器：G2中的所有服务器

```
gSQL> SELECT c1 FROM t_shard_1@G2;
```

- 执行对象服务器：G3N1 server

```
gSQL> SELECT c1 FROM t_shard_1@G3N1;
```

- 执行对象服务器：无对象服务器

```
gSQL> SELECT c1 FROM t_shard_1@G4;
```

以下为根据sharding key检索条件选择执行对象服务器的示例

- 执行对象服务器：G1, G2, G3中的所有服务器

```
gSQL> SELECT c1 FROM t_shard_1;
```

- 执行对象服务器：G1, G2, G3中的所有服务器

```
gSQL> SELECT c1 FROM t_shard_1 WHERE c1 = 1;
```

- 执行对象服务器：G3中的所有服务器

```
gSQL> SELECT c1 FROM t_shard_1 WHERE shard_key = 500;
```

- 执行对象服务器：无对象服务器

```
gSQL> SELECT c1 FROM t_shard_1 WHERE shard_key = 100 AND shard_key = 500;
```

为了查找数据执行generated query时通过表副本（replica）部署策略和cluster domainsharding key检索条件查找共同的服务器同一个集群组中的多个服务器作为执行对象时考虑到执行时的网络状态等对每个集群组中可访问的一个服务器执行generated query

以下为选择执行对象服务器的示例

- 表副本（replica）部署策略：G1, G2, G3中的所有服务器
- Cluster domain
- t_shard_1 : G2 中的所有 server
- t_clone_1 : G1, G2, G3 中的所有 server
- join : G1, G2, G3 中的所有 server
- Sharding key检索条件：G2中的所有服务器
- Generated query执行对象服务器：G2中的一个服务器

```
gSQL> \EXPLAIN PLAN ONLY SELECT * FROM t_shard_1@G2, t_clone_1 WHERE
t_shard_1.shard_key = 300;


>>>  start print plan


< Execution Plan >
======================================================================
|  IDX  |  NODE DESCRIPTION                      |           ROWS |
----------------------------------------------------------------------
|    0  |  SELECT STATEMENT                      |              0 |
```

```
|    1  |    QUERY BLOCK ("$QB_IDX_2")          |             0 |
|    2  |      PLAN BASED CLUSTER               |             0 |
|    3  |        NESTED JOIN (INNER JOIN)       |             0 |
|    4  |          TABLE ACCESS ("T_SHARD_1")   |             0 |
|    5  |          TABLE ACCESS ("T_CLONE_1")   |             0 |
=========================================================================


     1  -  TARGET : T_SHARD_1.SHARD_KEY, T_SHARD_1.C1, T_CLONE_1.C1
     2  -  SQL : SELECT /*+ KEEP_JOINED_TABLE USE_NL_IN( _A1 ) FULL( _A2 )
FULL( _A1 ) */ "_A2"."SHARD_KEY", "_A2"."C1", "_A1"."C1" FROM
( "PUBLIC"."T_SHARD_1"@LOCAL AS "_A2" INNER JOIN
"PUBLIC"."T_CLONE_1"@LOCAL AS "_A1" ON true ) ALIAS "_A3" WHERE
"_A2"."SHARD_KEY" = :_V0
            TARGET DOMAIN : G2(G2N1,G2N2) 0 rows
     3  -  JOINED COLUMN : T_SHARD_1.SHARD_KEY, T_SHARD_1.C1, T_CLONE_1.C1
     4  -  RANGE SHARD ( # 3 )
            READ COLUMN : T_SHARD_1.SHARD_KEY, T_SHARD_1.C1
              PHYSICAL FILTER : T_SHARD_1.SHARD_KEY = 300
     5  -  CLONED
            READ COLUMN : T_CLONE_1.C1


<<<  end print plan
```

使用Cluster Puller

根据数据操作方法选择cluster puller node不需要操作数据时可使用plan based cluster或single

cluster需要结果的ordering时使用multiple cluster数据收集方法取决于数据操作方法

区分	数据收集方法	数据操作方法
Plan based cluster	By pass	No manipulation
Single cluster	By pass	No manipulation Aggregation Grouping
Multiple cluster	Merge sort	Ordering Grouping Intersect key group Distinct key group

Table 2-8 Cluster puller node支持功能

Cluster Puller Plan Node

Cluster puller plan node根据从本地服务器和远程服务器获取数据方法进行分类根据收集数据的cluster puller plan node的分类参考**表 12-7 Cluster puller plan node**

Plan Based Cluster

在数据收集阶段按照接收的顺序汇集执行结果

- 本地数据收集：执行在本地构成的plan
- 远程数据收集：执行generated query

汇集的数据除使用filter外不进行其他操作（No manipulation）

以下为执行plan based cluster的示例

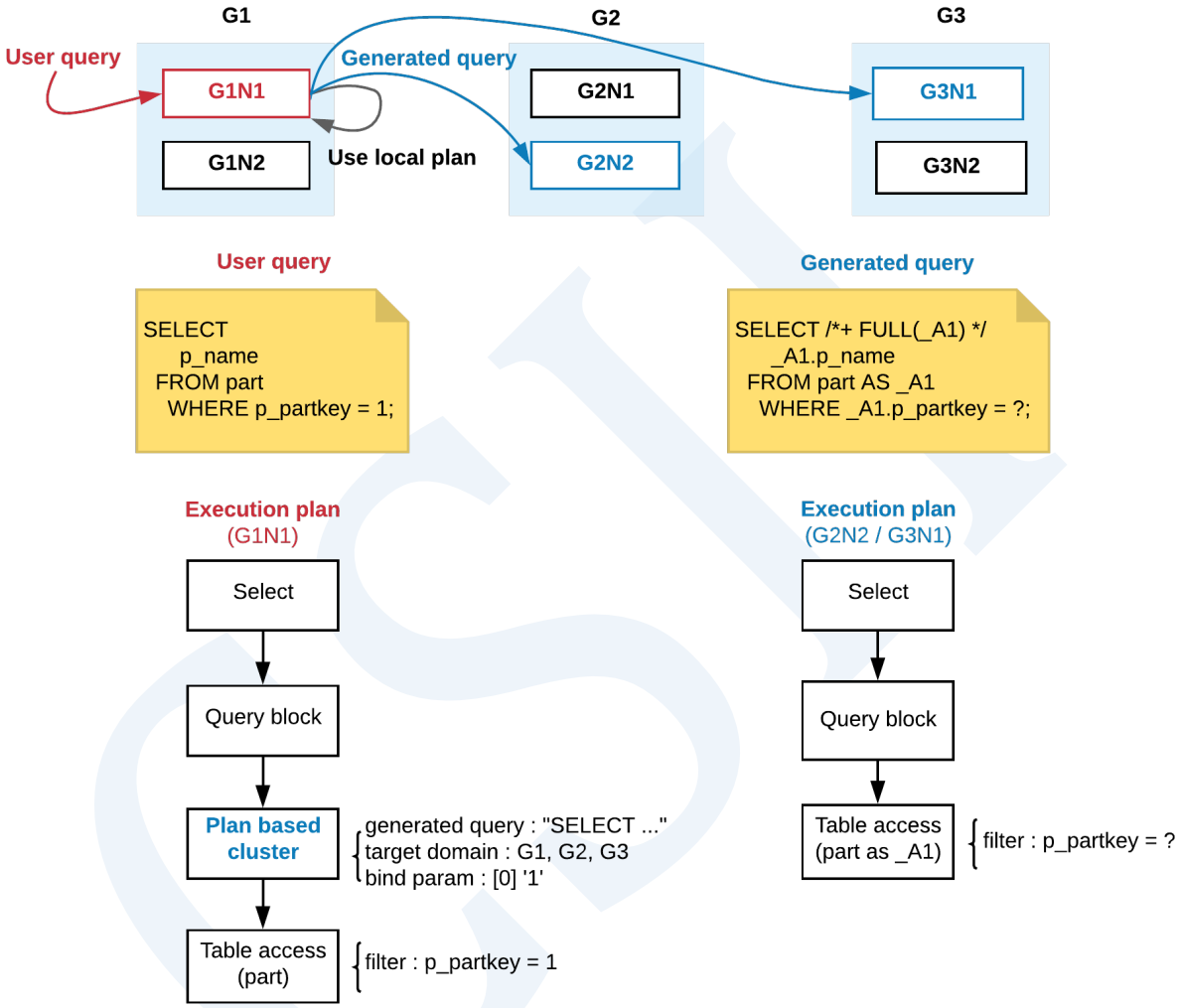


Figure 2-4 Plan based cluster

Single Cluster

在数据收集阶段按照接收的顺序汇集执行结果

- 本地数据收集：执行generated query

- 远程数据收集：执行generated query

Single cluster支持以下数据操作方法

- No manipulation
- Aggregation
- Grouping

以下为执行single cluster的示例

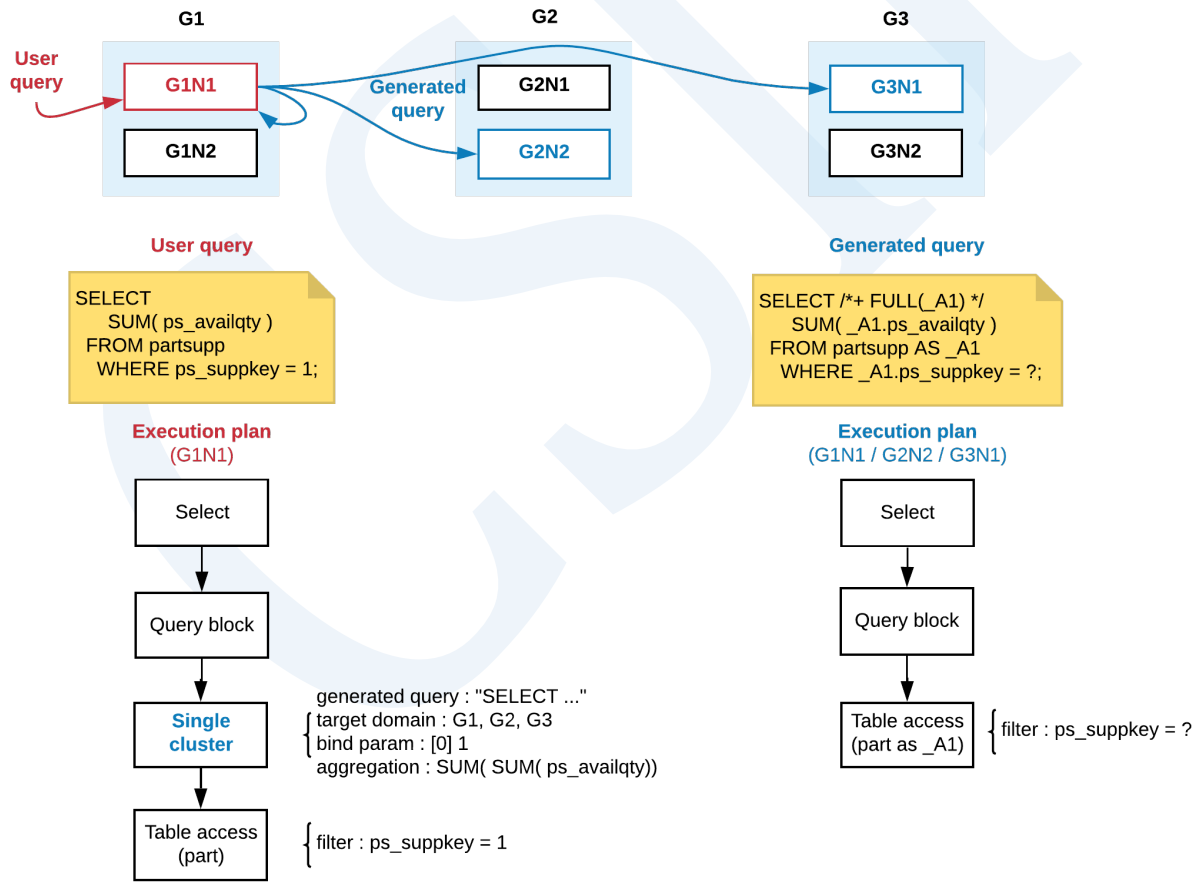


Figure 2-5 Single cluster

Multiple Cluster

Multiple cluster在每个group使用不同的cluster executor执行generated query在数据收集阶段merge sort通过所有cluster executor接收的数据并汇集

- 本地数据收集：执行generated query
- 远程数据收集：执行generated query

Multiple cluster支持以下数据操作方法

- Ordering
- Grouping
- Intersect key group
- Distinct key group

以下为执行multiple cluster的示例

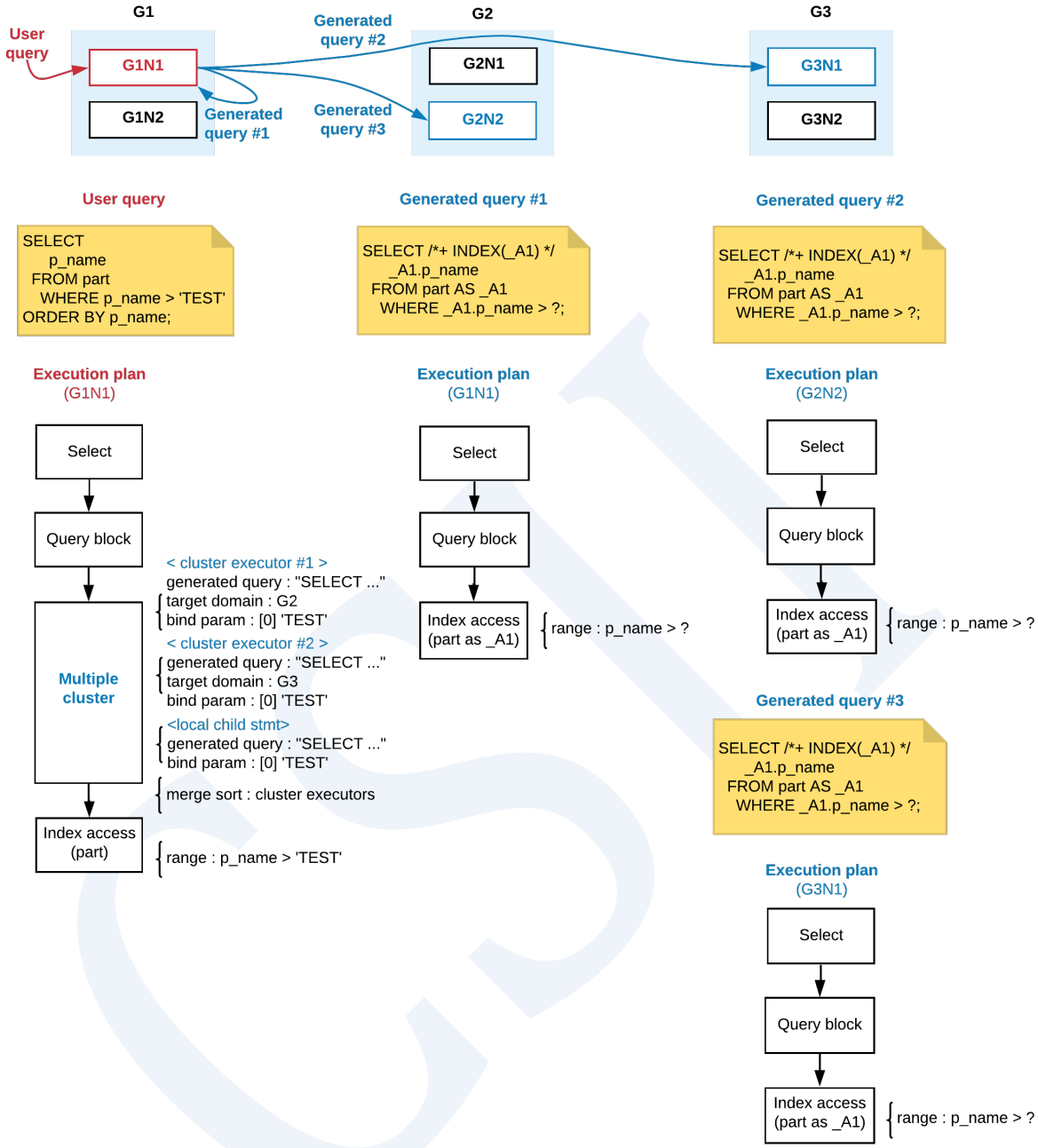


Figure 2-6 Multiple cluster

Cluster Domain

Cluster domain在集群环境中仅从限制的服务器汇集数据例如在按照各区间划分salary并构成的sharded table中检索特定区间的员工时可将属于其的cluster group设置为cluster domain并如下

执行query

- 按照salary区间构成sharded table

```
gSQL> CREATE TABLE t1( name VARCHAR(128), salary INTEGER )
    SHARDING BY RANGE( salary )
        SHARD s1 VALUES LESS THAN ( 200 )       AT CLUSTER GROUP G1,
        SHARD s2 VALUES LESS THAN ( 400 )       AT CLUSTER GROUP G2,
        SHARD s3 VALUES LESS THAN ( MAXVALUE )  AT CLUSTER GROUP G3;

Table created.

gSQL> INSERT INTO t1 VALUES ( 'A', 500 );

1 row created.

gSQL> INSERT INTO t1 VALUES ( 'B', 100 );

1 row created.

gSQL> INSERT INTO t1 VALUES ( 'C', 300 );

1 row created.
```

- 检索特定salary区间的员工

```
gSQL> SELECT name, salary FROM t1@G2;


NAME SALARY

---- ------

C       300


1 row selected.
```

Cluster domain以描述在**from clause**的表或视图为对象参考**<cluster domain>**定义此时可选择以下中的一个

- 执行用户查询的集群成员
- 以offline table为对象执行用户查询的集群成员
- 所有集群组
- 一个集群组
- 一个集群成员

集群成员被选为cluster domain时访问该服务器并汇集数据如果该服务器不拥有对象表的数据分配时检索结果不存在

为了在用户访问的服务器查询offline状态的表数据支持"@LOCAL_OFFLINE" cluster domain如果该table为online状态则报错

集群组被选为cluster domain时访问在该组内拥有对象表的数据分配并可进行通信的一个服务器并汇集数据如果没有可访问的服务器时检索结果不存在

所有集群组被选为cluster domain时向用户传输收集并汇集各个集群组数据的结果

Cluster domain不能用于作为指定数据更新对象即<cluster domain>不适用于DML对象表

DML对象表的cluster domain产生如下syntax error

```
gSQL> INSERT INTO t1@LOCAL VALUES ( 1 );


ERR-42000(16062): syntax error :
INSERT INTO t1@LOCAL VALUES ( 1 )
              *
ERROR at line 1:




gSQL> UPDATE t1@GLOBAL SET c1 = 1;


ERR-42000(16062): syntax error :
UPDATE t1@GLOBAL SET c1 = 1
         *
ERROR at line 1:




gSQL> DELETE FROM t1@G1;


ERR-42000(16062): syntax error :
DELETE FROM t1@G1
              *
ERROR at line 1:
```

SELECT FOR UPDATE的变更对象表的cluster domain如下产生syntax error

- 数据汇集对象的cluster domain

```
gSQL> SELECT c1 FROM t1@LOCAL;


I1

--

 1


1 row selected.
```

- 数据更新对象的cluster domain

```
gSQL> SELECT c1 FROM t1@LOCAL FOR UPDATE;


ERR-42000(16062): syntax error :

SELECT c1 FROM t1@LOCAL FOR UPDATE

                  *

ERROR at line 1:
```

减少Cluster Puller的Target Domain

减少发送generated query处理的target domian的方法为如下

- **Cluster Domain**：指定domain并减少target domain
- Shard key filter：赋予Sharding key条件并减少target domain

- Rowinfo domain filter：赋予table相关pseudo column条件并减少target domain

Shard key filter和rowinfo domain filter合起来被称为domian filter

以下为在表指定查询对象domain并查询single table的示例

```
gSQL> \EXPLAIN PLAN SELECT c1 FROM t_shard_1@G2;


C1
--
 2


1 row selected.


>>>  start print plan


< Execution Plan >
=============================================================================
|  IDX  |  NODE DESCRIPTION                          |              ROWS |
-----------------------------------------------------------------------------
|    0  |  SELECT STATEMENT                          |                 1 |
|    1  |    QUERY BLOCK ("$QB_IDX_2")               |                 1 |
|    2  |      PLAN BASED CLUSTER                    | REMOTE ONLY     1
|
|    3  |        INDEX ACCESS ("T_SHARD_1", "IDX")   | (         0)    0 |
=============================================================================
```

```
     1  -  TARGET : T_SHARD_1.C1
     2  -  SQL : SELECT /*+ INDEX( _A1, "PUBLIC"."IDX" ) */ "_A1"."C1"
FROM "PUBLIC"."T_SHARD_1"@LOCAL AS "_A1"
           TARGET DOMAIN : G2(G2N1,G2N2) 1 rows
     3  -  RANGE SHARD ( # 3 )
           READ INDEX COLUMN : T_SHARD_1.C1

<<<  end print plan
```

指定查询对象domian时可看到TARGET DOMAIN减少到G2Domain相关详细说明参考**Cluster Domain**

以下为在where子句添加shard key条件并限制TARGET DOMAIN的示例

```
gSQL> \EXPLAIN PLAN SELECT c1 FROM t_shard_1 WHERE shard_key = 555;

C1
--
 3

1 row selected.

>>>  start print plan

< Execution Plan >
```

```
==========================================================================
|  IDX  |  NODE DESCRIPTION                       |                ROWS |
--------------------------------------------------------------------------
|    0  |  SELECT STATEMENT                       |                   1 |
|    1  |    QUERY BLOCK ("$QB_IDX_2")            |                   1 |
|    2  |      PLAN BASED CLUSTER                 | REMOTE ONLY       1 |
|    3  |        TABLE ACCESS ("T_SHARD_1")       |                   0 |
==========================================================================


     1  -  TARGET : T_SHARD_1.C1
     2  -  SQL : SELECT /*+ FULL( _A1 ) */ "_A1"."C1" FROM
"PUBLIC"."T_SHARD_1"@LOCAL AS "_A1" WHERE "_A1"."SHARD_KEY" = :_V0
           TARGET DOMAIN : G3(G3N1,G3N2) 1 rows
     3  -  RANGE SHARD ( # 3 )
           READ COLUMN : T_SHARD_1.SHARD_KEY, T_SHARD_1.C1
             PHYSICAL FILTER : T_SHARD_1.SHARD_KEY = 555


<<<  end print plan
```

如上添加拥有常数值的shard key条件时TARGET DOMAIN有所减少

可通过shard key条件限制查找对象group拥有常数值的shard key条件可在部署plan的过程中决定查找对象group非常数值的shard key条件在执行该查询的时间点决定查找对象group

即添加非常数值的shard key条件时不减少TARGET DOMAIN而在执行时间点构成决定domain的信息SHARD KEY FILTER条件

以下为在where子句添加非常数值的shard key条件限制TARGET DOMAIN的示例

```
gSQL> VAR V1 INTEGER

gSQL> EXEC :V1 := 555

gSQL> \EXPLAIN PLAN SELECT c1 FROM t_shard_1 WHERE shard_key = :V1;


C1

--

 3


1 row selected.


>>>  start print plan


< Execution Plan >

=========================================================================

|  IDX  |  NODE DESCRIPTION                    |                ROWS |

-------------------------------------------------------------------------

|    0  |  SELECT STATEMENT                    |                   1 |

|    1  |    QUERY BLOCK ("$QB_IDX_2")         |                   1 |

|    2  |      PLAN BASED CLUSTER              | REMOTE ONLY       1 |

|    3  |        TABLE ACCESS ("T_SHARD_1")    |                   0 |

=========================================================================


     1  -  TARGET : T_SHARD_1.C1
```

```
    2  -  SQL : SELECT /*+ FULL( _A1 ) */ "_A1"."C1" FROM
"PUBLIC"."T_SHARD_1"@LOCAL AS "_A1" WHERE "_A1"."SHARD_KEY" = :_V0
          TARGET DOMAIN : G1(G1N1,G1N2) 0 rows, G2(G2N1,G2N2) 0 rows,
G3(G3N1,G3N2) 1 rows
            SHARD KEY FILTER : ( T_SHARD_1.SHARD_KEY = :V1 )
    3  -  RANGE SHARD ( # 3 )
          READ COLUMN : T_SHARD_1.SHARD_KEY, T_SHARD_1.C1
            PHYSICAL FILTER : T_SHARD_1.SHARD_KEY = :V1

<<<  end print plan
```

减少TARGET DOMAIN的方法还有赋予table相关伪列（Pseudo Columns）条件的方法只有在使用pseudo column的equal (=)条件的情况下可用作domain filter

Pseudo column	是否可使用domain filter
CURRVAL	不可以
NEXTVAL	不可以
ROWNUM	不可以
ROWID	可以
CLUSTER_GROUP_ID	可以
CLUSTER_MEMBER_ID	可以
CLUSTER_GROUP_NAME	可以
CLUSTER_NAME_ID	可以

Pseudo column	是否可使用domain filter
CLUSTER_SHARD_ID	可以

Table 2-9 可用作domain filter的pseudo column

以下为添加pseudo column条件限制TARGET DOMAIN的示例

```
gSQL> \EXPLAIN PLAN SELECT c1 FROM t_shard_1 WHERE cluster_group_name =
'G3';


C1
--
 3


1 row selected.


>>>  start print plan


< Execution Plan >
================================================================================
==
|  IDX  |  NODE DESCRIPTION                          |               ROWS
|
--------------------------------------------------------------------------------
--
```

```
|    0  |  SELECT STATEMENT                          |                  1
|
|    1  |    QUERY BLOCK ("$QB_IDX_2")               |                  1
|
|    2  |      PLAN BASED CLUSTER                    | REMOTE ONLY      1
|
|    3  |        INDEX ACCESS ("T_SHARD_1", "IDX")   | (         0)     0
|
================================================================================
==


     1  -  TARGET : T_SHARD_1.C1
     2  -  SQL : SELECT /*+ INDEX( _A1, "PUBLIC"."IDX" ) */ "_A1"."C1"
FROM "PUBLIC"."T_SHARD_1"@LOCAL AS "_A1" WHERE "_A1"."CLUSTER_GROUP_NAME"
= :_V0
            TARGET DOMAIN : G3(G3N1,G3N2) 1 rows
     3  -  RANGE SHARD ( # 3 )
            READ INDEX COLUMN : T_SHARD_1.C1
              LOGICAL KEY FILTER : T_SHARD_1.CLUSTER_GROUP_NAME = 'G3'


<<<  end print plan
```

使用常数值的pseudo column条件减少TARGET DOMAIN

以下为使用非常数值的pseudo column条件的示例

```
gSQL> VAR V1 VARCHAR( 10 )

gSQL> EXEC :V1 := 'G3N1'

gSQL> \EXPLAIN PLAN SELECT c1 FROM t_shard_1 WHERE cluster_member_name
= :V1;


C1
--
 3


1 row selected.


>>>  start print plan


< Execution Plan >
================================================================================
=
|  IDX  |  NODE DESCRIPTION                          |               ROWS
|
--------------------------------------------------------------------------------
-
|    0  |  SELECT STATEMENT                          |                  1
|
|    1  |    QUERY BLOCK ("$QB_IDX_2")               |                  1
|
|    2  |      PLAN BASED CLUSTER                    | REMOTE ONLY      1
```

```
|
|    3  |        INDEX ACCESS ("T_SHARD_1", "IDX")   | (       0)       0
|
=============================================================================
=


     1  -  TARGET : T_SHARD_1.C1
     2  -  SQL : SELECT /*+ INDEX( _A1, "PUBLIC"."IDX" ) */ "_A1"."C1"
FROM "PUBLIC"."T_SHARD_1"@LOCAL AS "_A1" WHERE "_A1"."CLUSTER_MEMBER_NAME"
= :_V0
           TARGET DOMAIN : G1(G1N1,G1N2) 0 rows, G2(G2N1,G2N2) 0 rows,
G3(G3N1,G3N2) 1 rows
             ROWINFO DOMAIN FILTER : T_SHARD_1.CLUSTER_MEMBER_NAME = :V1
     3  -  RANGE SHARD ( # 3 )
           READ INDEX COLUMN : T_SHARD_1.C1
             LOGICAL KEY FILTER : T_SHARD_1.CLUSTER_MEMBER_NAME = :V1


<<<  end print plan
```

添加非常数值的pseudo column条件时不减少TARGET DOMAIN而在执行时间点构成决定domain的信息ROWINFO DOMAIN FILTER条件

Generated Query

处理用户提供的查询时为了参照或更新其他数据库的数据创建generated query

```
SELECT c1 FROM t1;
```

指定如上用户查询时收到用户查询的服务器为了从所有相关集群组汇集t1的数据而创建类似如下的generated query

```
SELECT c1 FROM t1@LOCAL;
```

以下为用于说明generated query而构成的例表

```
CREATE TABLE t_shard_1( shard_key INTEGER, c1 INTEGER )
    SHARDING BY RANGE( shard_key )
        SHARD s1 VALUES LESS THAN ( 200 )       AT CLUSTER GROUP G1,
        SHARD s2 VALUES LESS THAN ( 400 )       AT CLUSTER GROUP G2,
        SHARD s3 VALUES LESS THAN ( MAXVALUE )  AT CLUSTER GROUP G3;

CREATE TABLE t_shard_2( shard_key INTEGER, c1 INTEGER )
    SHARDING BY RANGE( shard_key )
        SHARD s1 VALUES LESS THAN ( 300 )       AT CLUSTER GROUP G1,
        SHARD s2 VALUES LESS THAN ( MAXVALUE )  AT CLUSTER GROUP G3;


CREATE TABLE t_clone_1( c1 INTEGER ) CLONED AT CLUSTER WIDE;


CREATE TABLE t_clone_2( c1 INTEGER ) CLONED AT CLUSTER GROUP G2, G3;
```

Generated query是基于cluster puller plan node重构的查询

根据**使用Cluster Puller**各个 cluster puller plan node构成**数据收集方法**的generated query

用于No Manipulation的Generated Query

对汇集的数据不执行操作时通过cluster puller plan node使用plan based cluster或single cluster

不执行数据操作的cluster puller plan node的explain plan结果中构成并输出generated query但不输出manipulation的信息

以下为使用plan based cluster的no manipulation的示例

```
gSQL> \EXPLAIN PLAN SELECT c1 FROM t_shard_1;


C1
--
 1
 3
 2


3 rows selected.


>>>  start print plan


< Execution Plan >
================================================================================
```

```
==
|  IDX   |  NODE DESCRIPTION                                |              ROWS
|
-------------------------------------------------------------------------------
--
|    0   |  SELECT STATEMENT                                |                 3
|
|    1   |    QUERY BLOCK ("$QB_IDX_2")                     |                 3
|
|    2   |      PLAN BASED CLUSTER                          | LOCAL/REMOTE    3
|
|    3   |        TABLE ACCESS ("T_SHARD_1")                |                 1
|
===============================================================================
==


     1  -  TARGET : T_SHARD_1.C1
     2  -  SQL : SELECT /*+ FULL( _A1 ) */ "_A1"."C1" FROM
"PUBLIC"."T_SHARD_1"@LOCAL AS "_A1"
           TARGET DOMAIN : G1(G1N1,G1N2) 1 rows, G2(G2N1,G2N2) 1 rows,
G3(G3N1,G3N2) 1 rows
     3  -  RANGE SHARD ( # 3 )
           READ COLUMN : T_SHARD_1.C1


<<<  end print plan
```

以下为使用single cluster的no manipulation的示例

```
gSQL> \EXPLAIN PLAN SELECT t_shard_1.c1 FROM t_shard_1, t_shard_2 WHERE
t_shard_1.shard_key = t_shard_2.c1;


C1
--
 1
 2


2 rows selected.


>>>  start print plan


< Execution Plan >
================================================================================
==
| IDX | NODE DESCRIPTION                                |           ROWS
|
--------------------------------------------------------------------------------
--
|   0 | SELECT STATEMENT                                |              2
|
|   1 |   QUERY BLOCK ("$QB_IDX_2")                     |              2
|
```

```
|   2 |     SINGLE CLUSTER                                | LOCAL/REMOTE 2
|
|   3 |       CLUSTER PUSHER ("_$NI_6")                   |              3
|
|   4 |         PLAN BASED CLUSTER                        | LOCAL/REMOTE 3
|
|   5 |           TABLE ACCESS ("T_SHARD_2")              |              2
|
|   6 |       SELECT STATEMENT                            |              0
|
|   7 |         QUERY BLOCK ("$QB_IDX_2")                 |              0
|
|   8 |           NESTED JOIN (INNER JOIN)                |              0
|
|   9 |             TABLE ACCESS ("T_SHARD_1" AS _A2)     |              1
|
|  10 |             PUSHER TABLE ACCESS ("_$NI_6" AS _A1) |              0
|
================================================================================
==

     1  -  TARGET : T_SHARD_1.C1
     2  -  SQL : SELECT /*+ KEEP_JOINED_TABLE USE_NL_IN( _A1 ) FULL( _A2 )
FULL( _A1 ) */ "_A2"."C1" FROM ( "PUBLIC"."T_SHARD_1"@LOCAL AS "_A2" INNER JOIN
"SESSION_SCHEMA"."_$NI_6"@LOCAL AS "_A1" ON "_A2"."SHARD_KEY" = "_A1"."C1")
```

```
ALIAS "_A3"
           TARGET DOMAIN : G1(G1N1,G1N2) 1 rows, G2(G2N1,G2N2) 1 rows,
G3(G3N1,G3N2) 0 rows
     3  -  SQL : DECLARE INSTANT TABLE "SESSION_SCHEMA"."_$NI_6" ( "C1"
NUMBER(10, 0) )
           COLUMN : T_SHARD_2.C1 AS C1
           SHARDED : T_SHARD_2.C1
           TARGET DOMAIN : G1(G1N1,G1N2) 1 rows, G2(G2N1,G2N2) 2 rows,
G3(G3N1,G3N2) 0 rows
     4  -  SQL : SELECT /*+ FULL( _A1 ) */ "_A1"."C1" FROM
"PUBLIC"."T_SHARD_2"@LOCAL AS "_A1"
           TARGET DOMAIN : G1(G1N1,G1N2) 1 rows, G3(G3N1,G3N2) 2 rows
     5  -  RANGE SHARD ( # 4 )
           READ COLUMN : T_SHARD_2.C1
     7  -  TARGET : _A2.C1
     8  -  JOINED COLUMN : _A2.SHARD_KEY, _A1.C1, _A2.C1
             ON FILTER : _A2.SHARD_KEY = _A1.C1
     9  -  RANGE SHARD ( # 3 )
           READ COLUMN : _A2.SHARD_KEY, _A2.C1
    10  -  READ COLUMN : _A1.C1


<<<  end print plan
```

用于Aggregation的Generated Query

用于aggregation的generated query支持按照各个group执行aggregation再次汇集通过generated

query收集的数据后构成最终结果

用户查询中的aggregation	Generated query中包含的运算形式	汇集结果的运算
COUNT()	COUNT()	SUM(count())
SUM()	SUM()	SUM(SUM())
AVG()	COUNT(), SUM()	SUM(SUM()) / SUM(COUNT())
MIN()	MIN()	MIN(MIN())
MAX()	MAX()	MAX(MAX())

Table 2-10 Aggregation数据操作方法

为了处理用户query的COUNT运算generated query构成包含COUNT运算的查询汇集通过Generated query收集的数据累计各个group的COUNT结果所有group的COUNT累计结果值为用户COUNT运算的结果

以下为使用single cluster的aggregation的示例

```
gSQL> \EXPLAIN PLAN SELECT COUNT( c1 ) FROM t_shard_1;


COUNT( C1 )
-----------
          3


1 row selected.
```

```
>>>  start print plan


< Execution Plan >

================================================================================
|  IDX  |  NODE DESCRIPTION                          |            ROWS |
--------------------------------------------------------------------------------
|    0  |  SELECT STATEMENT                          |               1 |
|    1  |    QUERY BLOCK ("$QB_IDX_2")               |               1 |
|    2  |      SINGLE CLUSTER                        | LOCAL/REMOTE  1 |
|    3  |        SELECT STATEMENT                    |               1 |
|    4  |          QUERY BLOCK ("$QB_IDX_2")         |               1 |
|    5  |            TABLE ACCESS ("T_SHARD_1" AS _A1) |               1 |
================================================================================


     1  -  TARGET : COUNT( T_SHARD_1.C1 )
     2  -  SQL : SELECT /*+ FULL( _A1 ) */ COUNT( "_A1"."C1" ) FROM
"PUBLIC"."T_SHARD_1"@LOCAL AS "_A1"
           TARGET DOMAIN : G1(G1N1,G1N2) 1 rows, G2(G2N1,G2N2) 1 rows,
G3(G3N1,G3N2) 1 rows
           RE-AGGREGATION
             AGGREGATION : SUM( COUNT( T_SHARD_1.C1 ) )
     4  -  TARGET : COUNT( _A1.C1 )
     5  -  RANGE SHARD ( # 3 )
           READ COLUMN : _A1.C1
```

```
            AGGREGATION : COUNT( _A1.C1 )


<<<  end print plan
```

如COUNT(DISTINCT c1)在aggregation运算指定DISTINCT时无法构成包含aggregation的generated querycluster puller plan node中不支持aggregation相关数据操作

以下为COUNT(DISTINCT)运算的示例

```
gSQL> \EXPLAIN PLAN SELECT COUNT( DISTINCT c1 ) FROM t_shard_1;


COUNT( DISTINCT C1 )
--------------------
                   3


1 row selected.


>>>  start print plan


< Execution Plan >
================================================================================
==
|  IDX  |  NODE DESCRIPTION                                |            ROWS
|
--------------------------------------------------------------------------------
--
```

```
|    0   |  SELECT STATEMENT                                |               1
|
|    1   |    QUERY BLOCK ("$QB_IDX_2")                     |               1
|
|    2   |      AGGREGATION BY HASH                         |               1
|
|    3   |        PLAN BASED CLUSTER                        | LOCAL/REMOTE  3
|
|    4   |          TABLE ACCESS ("T_SHARD_1")              |               1
|
================================================================================
==


     1  -  TARGET : COUNT( DISTINCT T_SHARD_1.C1 )
     2  -  DISTINCT AGGREGATION : COUNT( DISTINCT T_SHARD_1.C1 )
     3  -  SQL : SELECT /*+ FULL( _A1 ) */ "_A1"."C1" FROM
"PUBLIC"."T_SHARD_1"@LOCAL AS "_A1"
           TARGET DOMAIN : G1(G1N1,G1N2) 1 rows, G2(G2N1,G2N2) 1 rows,
G3(G3N1,G3N2) 1 rows
     4  -  RANGE SHARD ( # 3 )
           READ COLUMN : T_SHARD_1.C1


<<<  end print plan
```

使用包含DISTINCT的aggregation时cluster puller的generated query不包含aggregation由cluster

puller收集的数据通过单独的plan node执行汇集运算

用于Grouping的Generated Query

用于Grouping的Generated Query按组执行grouping对通过generated query收集的数据再次进行grouping

在cluster puller plan node完成grouping后应用HAVING条件

以下为在cluster puller plan node处理grouping的示例

```
gSQL> \EXPLAIN PLAN SELECT COUNT( c1 ) FROM t_shard_1 AS A GROUP BY c1
HAVING MIN( shard_key ) > 1;


COUNT( C1 )
-----------
          1
          1
          1


3 rows selected.


>>>  start print plan


< Execution Plan >
================================================================================
==
```

```
|  IDX  |  NODE DESCRIPTION                                |             ROWS
|
-------------------------------------------------------------------------------
--
|    0  |  SELECT STATEMENT                                |                3
|
|    1  |    QUERY BLOCK ("$QB_IDX_2")                     |                3
|
|    2  |      SINGLE CLUSTER                              | LOCAL/REMOTE   3
|
|    3  |        SELECT STATEMENT                          |                1
|
|    4  |          QUERY BLOCK ("$QB_IDX_2")               |                1
|
|    5  |            GROUP HASH INSTANT                    |                1
|
|    6  |              TABLE ACCESS ("T_SHARD_1" AS _A1)   |                1
|
===============================================================================
==

     1  -  TARGET : COUNT( A.C1 )
     2  -  SQL : SELECT /*+ USE_GROUP_HASH(100) FULL( _A1 ) */ "_A1"."C1",
MIN( "_A1"."SHARD_KEY" ), COUNT( "_A1"."C1" ) FROM "PUBLIC"."T_SHARD_1"@LOCAL
AS "_A1" GROUP BY "_A1"."C1"
```

```
            TARGET DOMAIN : G1(G1N1,G1N2) 1 rows, G2(G2N1,G2N2) 1 rows,
G3(G3N1,G3N2) 1 rows
            RE-GROUPING
              GROUP KEY : A.C1
              AGGREGATION : MIN( MIN( A.SHARD_KEY ) ), SUM( COUNT( A.C1 ) )
              PHYSICAL FILTER : MIN( A.SHARD_KEY ) > 1
     4   -  TARGET : _A1.C1, MIN( _A1.SHARD_KEY ), COUNT( _A1.C1 )
     5   -  GROUP KEY : _A1.C1
            RECORD COLUMN : MIN( _A1.SHARD_KEY ), COUNT( _A1.C1 )
            READ KEY COLUMN : _A1.C1
            READ RECORD COLUMN : MIN( _A1.SHARD_KEY ), COUNT( _A1.C1 )
     6   -  RANGE SHARD ( # 3 )
            READ COLUMN : _A1.SHARD_KEY, _A1.C1


<<<  end print plan
```

根据按照各group进行grouping的**数据收集方法**决定cluster puller plan node以by pass的方式收集数据时使用single cluster以merge sort方式收集数据时使用multiple cluster

使用用于grouping的cluster puller plan node下级节点的preserved order时通过merge sort方式收集数据

以下为使用single cluster的grouping示例

```
gSQL> \EXPLAIN PLAN SELECT c1 FROM t_shard_1 AS A GROUP BY c1;
```

```
C1
--
 1
 2
 3

3 rows selected.

>>>  start print plan

< Execution Plan >
================================================================================
| IDX | NODE DESCRIPTION                                |            ROWS |
--------------------------------------------------------------------------------
|   0 | SELECT STATEMENT                                |               3 |
|   1 |  QUERY BLOCK ("$QB_IDX_2")                      |               3 |
|   2 |   SINGLE CLUSTER                                | LOCAL/REMOTE  3 |
|   3 |    SELECT STATEMENT                             |               1 |
|   4 |     QUERY BLOCK ("$QB_IDX_2")                   |               1 |
|   5 |      GROUP                                      |               1 |
|   6 |       INDEX ACCESS ("T_SHARD_1" AS _A1, "IDX") | (      1)     1 |
================================================================================

     1  -  TARGET : A.C1
     2  -  SQL : SELECT /*+ INDEX( _A1, "PUBLIC"."IDX" ) */ "_A1"."C1" FROM
```

```
"PUBLIC"."T_SHARD_1"@LOCAL AS "_A1" GROUP BY "_A1"."C1"
            TARGET DOMAIN : G1(G1N1,G1N2) 1 rows, G2(G2N1,G2N2) 1 rows,
G3(G3N1,G3N2) 1 rows
            RE-GROUPING
              GROUP KEY : A.C1
     4   -  TARGET : _A1.C1
     5   -  GROUP KEY : _A1.C1
     6   -  RANGE SHARD ( # 3 )
            READ INDEX COLUMN : _A1.C1


<<<  end print plan
```

使用single cluster的grouping在收集所有数据后可获取grouping结果

以下为使用multiple cluster的grouping示例

```
gSQL> \EXPLAIN PLAN SELECT /*+ MERGE_GROUP */ c1 FROM t_shard_1 AS A GROUP
BY c1;


C1
--
 1
 2
 3


3 rows selected.
```

```
>>>  start print plan


< Execution Plan >

================================================================================
| IDX | NODE DESCRIPTION                                |            ROWS |
--------------------------------------------------------------------------------
|   0 | SELECT STATEMENT                                |               3 |
|   1 |  QUERY BLOCK ("$QB_IDX_2")                      |               3 |
|   2 |   MULTIPLE CLUSTER                              | LOCAL/REMOTE  3 |
|   3 |    SELECT STATEMENT                             |               1 |
|   4 |     QUERY BLOCK ("$QB_IDX_2")                   |               1 |
|   5 |      GROUP                                      |               1 |
|   6 |       INDEX ACCESS ("T_SHARD_1" AS _A1, "IDX") | (      1)     1 |
================================================================================


     1  -  TARGET : A.C1

     2  -  SQL : SELECT /*+ INDEX( _A1, "PUBLIC"."IDX" ) */ "_A1"."C1" FROM
"PUBLIC"."T_SHARD_1"@LOCAL AS "_A1" GROUP BY "_A1"."C1" ORDER BY "_A1"."C1" ASC
NULLS LAST

           TARGET DOMAIN : G1(G1N1,G1N2) 1 rows, G2(G2N1,G2N2) 1 rows,
G3(G3N1,G3N2) 1 rows

           MERGE GROUPING

             SORT KEY : A.C1

             GROUP KEY : A.C1
```

```
     4  -  TARGET : _A1.C1
     5  -  GROUP KEY : _A1.C1
     6  -  RANGE SHARD ( # 3 )
           READ INDEX COLUMN : _A1.C1


<<<  end print plan
```

使用multiple cluster的grouping在以grouping key为单位完成收集数据后可获取grouping结果

但所有sharding key用作grouping条件时构成使用no manipulation的cluster puller该cluster puller构成包含grouping的generated query不操作所收集的数据直接向上层plan传达结果

以下为在用户查询中包含了GROUP BY语句但通过no manipulation处理grouping的示例

```
gSQL> \EXPLAIN PLAN SELECT shard_key FROM t_shard_1 AS A GROUP BY
shard_key;


SHARD_KEY
---------
      111
      555
      333


3 rows selected.


>>>  start print plan
```

```
< Execution Plan >
================================================================================
|  IDX  |  NODE DESCRIPTION                          |              ROWS |
--------------------------------------------------------------------------------
|    0  |  SELECT STATEMENT                          |                 3 |
|    1  |    QUERY BLOCK ("$QB_IDX_2")               |                 3 |
|    2  |      PLAN BASED CLUSTER                    | LOCAL/REMOTE    3 |
|    3  |        GROUP HASH INSTANT                  |                 1 |
|    4  |          TABLE ACCESS ("T_SHARD_1" AS A)   |                 1 |
================================================================================

     1  -  TARGET : A.SHARD_KEY
     2  -  SQL : SELECT /*+ USE_GROUP_HASH(10) FULL( _A1 ) */ "_A1"."SHARD_KEY"
FROM "PUBLIC"."T_SHARD_1"@LOCAL AS "_A1" GROUP BY "_A1"."SHARD_KEY"
           TARGET DOMAIN : G1(G1N1,G1N2) 1 rows, G2(G2N1,G2N2) 1 rows,
G3(G3N1,G3N2) 1 rows
     3  -  GROUP KEY : A.SHARD_KEY
           READ KEY COLUMN : A.SHARD_KEY
     4  -  RANGE SHARD ( # 3 )
           READ COLUMN : A.SHARD_KEY

<<<  end print plan
```

用于Ordering的Generated Query

用于ordering的generated query按照各个group执行ordering对通过generated query收集的数据重新进行ordering构成结果

用于ordering的数据收集以merge sort方式支持Merge sort方式使用multiple cluster

以下为在multiple cluster使用preserved order处理ordering的示例

```
gSQL> \EXPLAIN PLAN SELECT c1 FROM t_shard_1 ORDER BY c1;


C1
--
 1
 2
 3


3 rows selected.


>>>  start print plan


< Execution Plan >
================================================================================
==
| IDX | NODE DESCRIPTION                                |           ROWS
|
--------------------------------------------------------------------------------
```

```
--
|   0 |SELECT STATEMENT                                  |              3
|
|   1 |  QUERY BLOCK ("$QB_IDX_2")                       |              3
|
|   2 |    MULTIPLE CLUSTER                              | LOCAL/REMOTE 3
|
|   3 |      SELECT STATEMENT                            |              1
|
|   4 |        QUERY BLOCK ("$QB_IDX_2")                 |              1
|
|   5 |          INDEX ACCESS ("T_SHARD_1" AS _A1, "IDX") | (      1)    1
|
================================================================================
==


     1  -  TARGET : T_SHARD_1.C1
     2  -  SQL : SELECT /*+ INDEX( _A1, "PUBLIC"."IDX" ) */ "_A1"."C1" FROM
"PUBLIC"."T_SHARD_1"@LOCAL AS "_A1" ORDER BY "_A1"."C1" ASC NULLS LAST
           TARGET DOMAIN : G1(G1N1,G1N2) 1 rows, G2(G2N1,G2N2) 1 rows,
G3(G3N1,G3N2) 1 rows
           MERGE SORTING
             SORT KEY : T_SHARD_1.C1
     4  -  TARGET : _A1.C1
     5  -  RANGE SHARD ( # 3 )
```

```
            READ INDEX COLUMN : _A1.C1


<<<  end print plan
```

以下为在multiple cluster中没有preserved order情况下处理ordering的示例

```
gSQL> \EXPLAIN PLAN SELECT /*+ FULL( t_shard_1 ) */ c1 FROM t_shard_1
ORDER BY c1;


C1
--
 1
 2
 3


3 rows selected.


>>>  start print plan


< Execution Plan >
================================================================================
==
|  IDX  |  NODE DESCRIPTION                                |            ROWS
|
--------------------------------------------------------------------------------
--
```

```
|    0   |  SELECT STATEMENT                                |               3
|
|    1   |    QUERY BLOCK ("$QB_IDX_2")                     |               3
|
|    2   |      MULTIPLE CLUSTER                            | LOCAL/REMOTE  3
|
|    3   |        SELECT STATEMENT                          |               1
|
|    4   |          QUERY BLOCK ("$QB_IDX_2")               |               1
|
|    5   |            SORT INSTANT                          |               1
|
|    6   |              TABLE ACCESS ("T_SHARD_1" AS _A1)   |               1
|
================================================================================
==


     1  -  TARGET : T_SHARD_1.C1
     2  -  SQL : SELECT /*+ USE_ORDER_SORT FULL( _A1 ) */ "_A1"."C1" FROM
"PUBLIC"."T_SHARD_1"@LOCAL AS "_A1" ORDER BY "_A1"."C1" ASC NULLS LAST
            TARGET DOMAIN : G1(G1N1,G1N2) 1 rows, G2(G2N1,G2N2) 1 rows,
G3(G3N1,G3N2) 1 rows
            MERGE SORTING
              SORT KEY : T_SHARD_1.C1
     4  -  TARGET : _A1.C1
```

```
     5  -  SORT KEY : "_A1.C1 ASC NULLS LAST"
           READ KEY COLUMN : _A1.C1
     6  -  RANGE SHARD ( # 3 )
           READ COLUMN : _A1.C1


<<<  end print plan
```

用于Intersect Key Group的Generated Query

Intersect key group评估用于判断是否从所有group接收到了相同的数据

并非应用于收集到的所有数据仅限于指定为key group的值相同指定为nil expression的值均为null的情况应用intersect

nil expression值不是null时不评估intersect key group直接构成结果

nil expression值均为null时仅限于从所有group接收相同记录的情况构成intersect key group结果

用于intersect key group的generated query按照key group顺序进行ordering通过generated query收集的数据再次按照key group顺序进行排列排列的数据按照nil expression值应用intersect key group

以下为在multiple cluster处理intersect key group的示例

```
gSQL> \EXPLAIN PLAN
      SELECT /*+ REMOTE_JOIN( t_clone_1 ) */ t_clone_1.c1, t_shard_1.c1
        FROM t_clone_1
             LEFT OUTER JOIN
```

```
            t_shard_1

            ON t_clone_1.c1 = t_shard_1.c1;


C1   C1

-- ----

 1    1

 3    3

 5 null


3 rows selected.


>>>  start print plan


< Execution Plan >

================================================================================

=

| IDX | NODE DESCRIPTION                              |            ROWS

|

--------------------------------------------------------------------------------

-

|   0 | SELECT STATEMENT                              |               3

|

|   1 |  QUERY BLOCK ("$QB_IDX_2")                    |               3

|

|   2 |   MULTIPLE CLUSTER                            | LOCAL/REMOTE  3
```

```
|
|   3 |    SELECT STATEMENT                              |              3
|
|   4 |     QUERY BLOCK ("$QB_IDX_2")                    |              3
|
|   5 |      SORT INSTANT                                |              3
|
|   6 |       NESTED JOIN (LEFT OUTER JOIN)              |              3
|
|   7 |        TABLE ACCESS ("T_CLONE_1" AS _A2)         |              3
|
|   8 |        INDEX ACCESS ("T_SHARD_1" AS _A1, "IDX") | (    1)       1
|
================================================================================
=

     1  -  TARGET : T_CLONE_1.C1, T_SHARD_1.C1
     2  -  SQL : SELECT /*+ KEEP_JOINED_TABLE USE_NL_IN( _A1 ) FULL( _A2 )
INDEX( _A1, "PUBLIC"."IDX" ) */ "_A2"."C1", "_A1"."C1" FROM
( "PUBLIC"."T_CLONE_1"@"G1N1"|"G1N2"|"G2N1"|"G2N2"|"G3N1"|"G3N2" AS "_A2" LEFT
OUTER JOIN "PUBLIC"."T_SHARD_1"@"G1N1"|"G1N2"|"G2N1"|"G2N2"|"G3N1"|"G3N2" AS
"_A1" ON "_A1"."C1" = "_A2"."C1") ALIAS "_A3" ORDER BY "_A2"."C1" ASC NULLS LAST
            TARGET DOMAIN : G1(G1N1,G1N2) 3 rows, G2(G2N1,G2N2) 3 rows,
G3(G3N1,G3N2) 3 rows
            INTERSECT KEY GROUP
```

```
            KEY GROUP : T_CLONE_1.C1
            Nil Expression : T_SHARD_1.C1
    4   -   TARGET : _A2.C1, _A1.C1
    5   -   SORT KEY : "_A2.C1 ASC NULLS LAST"
        RECORD COLUMN : _A1.C1
        READ KEY COLUMN : _A2.C1
        READ RECORD COLUMN : _A1.C1
    6   -   JOINED COLUMN : _A2.C1, _A1.C1
    7   -   CLONED
        READ COLUMN : _A2.C1
    8   -   RANGE SHARD ( # 3 )
        READ INDEX COLUMN : _A1.C1
            MIN RANGE : _A1.C1 = {_A2.C1}
            MAX RANGE : _A1.C1 = {_A2.C1}

<<<  end print plan
```

用于Distinct Key Group的Generated Query

Distinct key group评估用于判断是否从两个以上的group收到了相同的数据

指定为key group的值相同并从互不相同的group接收数据的情况应用distinct相同group的数据不应用distinct

用于distinct key group的generated query按照key group顺序进行ordering通过generated query收集的数据再次按照key group顺序进行排列排列的数据按照接收的group应用distinct key group

以下为在multiple cluster处理distinct key group的示例

```
gSQL> \EXPLAIN PLAN
      SELECT t_clone_1.c1
        FROM t_clone_1
       WHERE t_clone_1.c1 IN ( SELECT /*+ REMOTE_UNNEST */ t_shard_1.c1
FROM t_shard_1 );

C1
--
 1
 3

2 rows selected.

>>>  start print plan

< Execution Plan >
================================================================================
==
| IDX |  NODE DESCRIPTION                                |            ROWS
|
--------------------------------------------------------------------------------
--
|   0 | SELECT STATEMENT                                 |               2
```

```
|
|   1 |  QUERY BLOCK ("$QB_IDX_2")                          |               2
|
|   2 |   MULTIPLE CLUSTER                                  | LOCAL/REMOTE  2
|
|   3 |    SELECT STATEMENT                                 |               1
|
|   4 |     QUERY BLOCK ("$QB_IDX_2")                       |               1
|
|   5 |      SORT INSTANT                                   |               1
|
|   6 |       HASH JOIN (SEMI)                              |               1
|
|   7 |        TABLE ACCESS ("T_CLONE_1" AS _A2)            |               3
|
|   8 |        HASH JOIN INSTANT (UNIQUE)                   |               1
|
|   9 |         INDEX ACCESS ("T_SHARD_1" AS _A1, "IDX") | (     1)      1
|
==========================================================================
==

     1  -  TARGET : T_CLONE_1.C1
     2  -  SQL : SELECT /*+ KEEP_JOINED_TABLE USE_HASH_IN( _A1, 100 ) FULL( _A2 )
INDEX( _A1, "PUBLIC"."IDX" ) */ "_A2"."C1" FROM
```

```
( "PUBLIC"."T_CLONE_1"@"G1N1"|"G1N2"|"G2N1"|"G2N2"|"G3N1"|"G3N2" AS "_A2" SEMI
JOIN "PUBLIC"."T_SHARD_1"@"G1N1"|"G1N2"|"G2N1"|"G2N2"|"G3N1"|"G3N2" AS "_A1"
ON "_A1"."C1" = "_A2"."C1") ALIAS "_A3" ORDER BY "_A2"."C1" ASC NULLS LAST
            TARGET DOMAIN : G1(G1N1,G1N2) 1 rows, G2(G2N1,G2N2) 0 rows,
G3(G3N1,G3N2) 1 rows
            DISTINCT KEY GROUP
              KEY GROUP : T_CLONE_1.C1
     4   -  TARGET : _A2.C1
     5   -  SORT KEY : "_A2.C1 ASC NULLS LAST"
            READ KEY COLUMN : _A2.C1
     6   -  JOINED COLUMN : _A2.C1
     7   -  CLONED
            READ COLUMN : _A2.C1
     8   -  HASH KEY : _A1.C1
            READ KEY COLUMN : _A1.C1
              HASH FILTER : _A1.C1 = _A2.C1
            FETCH ONE ROW
     9   -  RANGE SHARD ( # 3 )
            READ INDEX COLUMN : _A1.C1


<<<  end print plan
```

Generated Query构成的约束事项

如下情况中generated query构成受到约束

- Offset & Limit
- 使用non-deterministic expression
- 无法对子查询（subquery）执行unnset时

Offset & Limit

> Note:
>
> generated query不包含offset limit clause信息

- Generated query不包含offset信息

```
gSQL> \EXPLAIN PLAN ONLY SELECT c1 FROM t_shard_1 OFFSET 1;


>>>  start print plan


< Execution Plan >

=========================================================================
|  IDX  |  NODE DESCRIPTION                                |        ROWS |
-------------------------------------------------------------------------
|    0  |  SELECT STATEMENT                                |           0 |
|    1  |    QUERY BLOCK ("$QB_IDX_2")                     |           0 |
|    2  |      PLAN BASED CLUSTER                          |           0 |
|    3  |        INDEX ACCESS ("T_SHARD_1", "IDX")         |           0 |
=========================================================================
```

```
     1  -  TARGET : T_SHARD_1.C1
     2  -  SQL : SELECT /*+ INDEX( _A1, "PUBLIC"."IDX" ) */ "_A1"."C1" FROM
"PUBLIC"."T_SHARD_1"@LOCAL AS "_A1"
           TARGET DOMAIN : G1(G1N1,G1N2) 0 rows, G2(G2N1,G2N2) 0 rows,
G3(G3N1,G3N2) 0 rows
     3  -  RANGE SHARD ( # 3 )
           READ INDEX COLUMN : T_SHARD_1.C1


<<<  end print plan
```

- Generated query不包含limit信息

```
gSQL> \EXPLAIN PLAN ONLY SELECT c1 FROM t_shard_1 LIMIT 1;


>>>  start print plan


< Execution Plan >
=========================================================================
|  IDX  |  NODE DESCRIPTION                              |        ROWS |
-------------------------------------------------------------------------
|    0  |  SELECT STATEMENT                              |           0 |
|    1  |    QUERY BLOCK ("$QB_IDX_2")                   |           0 |
|    2  |      PLAN BASED CLUSTER                        |           0 |
|    3  |        INDEX ACCESS ("T_SHARD_1", "IDX")       |           0 |
=========================================================================
```

```
     1  -  TARGET : T_SHARD_1.C1
     2  -  SQL : SELECT /*+ INDEX( _A1, "PUBLIC"."IDX" ) */ "_A1"."C1" FROM
"PUBLIC"."T_SHARD_1"@LOCAL AS "_A1"
           TARGET DOMAIN : G1(G1N1,G1N2) 0 rows, G2(G2N1,G2N2) 0 rows,
G3(G3N1,G3N2) 0 rows
     3  -  RANGE SHARD ( # 3 )
           READ INDEX COLUMN : T_SHARD_1.C1


<<<  end print plan
```

使用non-deterministic expression

> Note:
>
> 使用non-deterministic expression时无法常数化non-deterministic expression时不包含在generated query中

- Generated query不包含sequence相关expression

```
gSQL> \EXPLAIN PLAN ONLY SELECT seq.nextval FROM t_shard_1;


>>>  start print plan


< Execution Plan >

================================================================================
|  IDX  |  NODE DESCRIPTION                                |         ROWS |
```

```
-------------------------------------------------------------------------
|    0  |  SELECT STATEMENT                              |            0 |
|    1  |    QUERY BLOCK ("$QB_IDX_2")                   |            0 |
|    2  |      PLAN BASED CLUSTER                        |            0 |
|    3  |        INDEX ACCESS ("T_SHARD_1", "IDX")       |            0 |
=========================================================================


     1  -  TARGET : NEXTVAL(SEQ)
     2  -  SQL : SELECT /*+ INDEX( _A1, "PUBLIC"."IDX" ) */ NULL FROM
"PUBLIC"."T_SHARD_1"@LOCAL AS "_A1"
           TARGET DOMAIN : G1(G1N1,G1N2) 0 rows, G2(G2N1,G2N2) 0 rows,
G3(G3N1,G3N2) 0 rows
     3  -  RANGE SHARD ( # 3 )
           READ INDEX COLUMN : NOTHING


<<<  end print plan
```

- 可常数化的non-deterministic expression被下级cluster puller node常数化
- Generated query以bind parameter形式包含non-deterministic expression

```
gSQL> \EXPLAIN PLAN ONLY SELECT c1 FROM t_shard_1 WHERE shard_key =
random( 1, 100 );


>>>  start print plan


< Execution Plan >
```

```
==========================================================================
|  IDX  |  NODE DESCRIPTION                        |               ROWS |
--------------------------------------------------------------------------
|    0  |  SELECT STATEMENT                        |                  0 |
|    1  |    QUERY BLOCK ("$QB_IDX_2")             |                  0 |
|    2  |      PLAN BASED CLUSTER                  |                  0 |
|    3  |        TABLE ACCESS ("T_SHARD_1")        |                  0 |
==========================================================================


     1  -  TARGET : T_SHARD_1.C1
     2  -  SQL : SELECT /*+ FULL( _A1 ) */ "_A1"."C1" FROM
"PUBLIC"."T_SHARD_1"@LOCAL AS "_A1" WHERE "_A1"."SHARD_KEY" = :_V0
           TARGET DOMAIN : G1(G1N1,G1N2) 0 rows, G2(G2N1,G2N2) 0 rows,
G3(G3N1,G3N2) 0 rows
     3  -  RANGE SHARD ( # 3 )
           READ COLUMN : T_SHARD_1.SHARD_KEY, T_SHARD_1.C1
             PHYSICAL FILTER : T_SHARD_1.SHARD_KEY = RANDOM(1,100)


<<<  end print plan
```

- generated query中不包含无法常数化的non-deterministic expression
- 在cluster puller node中收集数据后处理non-deterministic expression

```
gSQL> \EXPLAIN PLAN ONLY SELECT c1 FROM t_shard_1 WHERE shard_key =
random( c1, 100 );
```

```
>>>  start print plan


< Execution Plan >

=============================================================================
|  IDX  |  NODE DESCRIPTION                        |               ROWS |
-----------------------------------------------------------------------------
|    0  |  SELECT STATEMENT                        |                  0 |
|    1  |    QUERY BLOCK ("$QB_IDX_2")             |                  0 |
|    2  |      PLAN BASED CLUSTER                  |                  0 |
|    3  |        TABLE ACCESS ("T_SHARD_1")        |                  0 |
=============================================================================


     1  -  TARGET : T_SHARD_1.C1

     2  -  SQL : SELECT /*+ FULL( _A1 ) */ "_A1"."SHARD_KEY", "_A1"."C1"
FROM "PUBLIC"."T_SHARD_1"@LOCAL AS "_A1"

           TARGET DOMAIN : G1(G1N1,G1N2) 0 rows, G2(G2N1,G2N2) 0 rows,
G3(G3N1,G3N2) 0 rows

             POST FILTER : T_SHARD_1.SHARD_KEY = RANDOM(T_SHARD_1.C1,100)

     3  -  RANGE SHARD ( # 3 )

           READ COLUMN : T_SHARD_1.SHARD_KEY, T_SHARD_1.C1


<<<  end print plan
```

无法对子查询（subquery）执行unnset时

> Note:
>
> 可常数化的子查询以bind parameter形式包含在generated query中
>
> 无法常数化的子查询不包含在generated query中

子查询unnest相关详细内容参考**子查询（Subquery）**

- Generated query不包含未unnest的子查询
- 子查询在cluster puller node 或上级节点被处理

```
gSQL> \EXPLAIN PLAN
      SELECT shard_key
        FROM t_shard_1
       WHERE t_shard_1.c1 IN ( SELECT /*+ NO_QUERY_TRANSFORMATION */
t_clone_1.c1 FROM t_clone_1 );


SHARD_KEY
---------
      111
      555


2 rows selected.


>>>  start print plan
```

```
< Execution Plan >
================================================================================
|  IDX  |  NODE DESCRIPTION                        |                  ROWS |
--------------------------------------------------------------------------------
|    0  |  SELECT STATEMENT                        |                     2 |
|    1  |    QUERY BLOCK ("$QB_IDX_2")             |                     2 |
|    2  |      PLAN BASED CLUSTER                  | LOCAL/REMOTE        2 |
|    3  |        TABLE ACCESS ("T_SHARD_1")        |                     1 |
|    4  |  SUB QUERY LIST                          |                       |
|    5  |    INLINE_VIEW ("$V5") (MATERIALIZED)    |                     2 |
|    6  |      QUERY BLOCK ("$QB_IDX_6")           |                     3 |
|    7  |        TABLE ACCESS ("T_CLONE_1")        |                     3 |
================================================================================


     1  -  TARGET : T_SHARD_1.SHARD_KEY
     2  -  SQL : SELECT /*+ FULL( _A1 ) */ "_A1"."SHARD_KEY", "_A1"."C1"
FROM "PUBLIC"."T_SHARD_1"@LOCAL AS "_A1"
           TARGET DOMAIN : G1(G1N1,G1N2) 1 rows, G2(G2N1,G2N2) 1 rows,
G3(G3N1,G3N2) 1 rows
              POST FILTER : ( T_SHARD_1.C1 ) IN ( $V5.C1 )
     3  -  RANGE SHARD ( # 3 )
           READ COLUMN : T_SHARD_1.SHARD_KEY, T_SHARD_1.C1
     5  -  COLUMN : T_CLONE_1.C1 AS C1
     6  -  TARGET : T_CLONE_1.C1
```

```
     7  -  CLONED
           READ COLUMN : T_CLONE_1.C1


<<<  end print plan
```

- Generated query中不包含不可常数化的子查询

```
gSQL> \EXPLAIN PLAN
      SELECT shard_key
        FROM t_shard_1
       WHERE shard_key = ( SELECT c1 FROM dual );

no rows selected.

>>>  start print plan

< Execution Plan >
================================================================================
=
|  IDX  |  NODE DESCRIPTION                          |               ROWS
|
--------------------------------------------------------------------------------
-
|    0  |  SELECT STATEMENT                          |                  0
|
|    1  |    QUERY BLOCK ("$QB_IDX_2")               |                  0
```

```
|
|    2  |      PLAN BASED CLUSTER                    | LOCAL/REMOTE     0
|
|    3  |        TABLE ACCESS ("T_SHARD_1")          |                  1
|
|    4  |      SUB QUERY LIST                        |
|
|    5  |        INLINE_VIEW ("$V5")                 |                  3 |
|    6  |          QUERY BLOCK ("$QB_IDX_6")         |                  3
|
|    7  |            FAST DUAL ACCESS ("DUAL")       |                  3
|
================================================================================
=


     1  -  TARGET : T_SHARD_1.SHARD_KEY
     2  -  SQL : SELECT /*+ FULL( _A1 ) */ "_A1"."SHARD_KEY", "_A1"."C1"
FROM "PUBLIC"."T_SHARD_1"@LOCAL AS "_A1"
           TARGET DOMAIN : G1(G1N1,G1N2) 1 rows, G2(G2N1,G2N2) 1 rows,
G3(G3N1,G3N2) 1 rows
             POST FILTER : T_SHARD_1.SHARD_KEY = $V5.C1
     3  -  RANGE SHARD ( # 3 )
           READ COLUMN : T_SHARD_1.SHARD_KEY, T_SHARD_1.C1
     5  -  COLUMN : {T_SHARD_1.C1} AS C1
     6  -  TARGET : {T_SHARD_1.C1}
```

```
     7  -  READ COLUMN : NOTHING


<<<  end print plan
```

- Generated query以bind parameter的形式包含常数化的子查询

```
gSQL> \EXPLAIN PLAN
SELECT shard_key
  FROM t_shard_1
 WHERE shard_key = ( SELECT 111 FROM dual );


SHARD_KEY
---------
      111


1 row selected.


>>>  start print plan


< Execution Plan >
=========================================================================
|  IDX  |  NODE DESCRIPTION                      |              ROWS |
-------------------------------------------------------------------------
|    0  |  SELECT STATEMENT                      |                 1 |
|    1  |    QUERY BLOCK ("$QB_IDX_2")           |                 1 |
|    2  |      PLAN BASED CLUSTER                | LOCAL ONLY      1 |
```

```
|    3  |        TABLE ACCESS ("T_SHARD_1")      |                 1 |
|    4  |  SUB QUERY LIST                        |                   |
|    5  |    INLINE_VIEW ("$V5")                 |                 1 |
|    6  |      QUERY BLOCK ("$QB_IDX_6")         |                 1 |
|    7  |        FAST DUAL ACCESS ("DUAL")       |                 1 |
=========================================================================


     1  -  TARGET : T_SHARD_1.SHARD_KEY
     2  -  SQL : SELECT /*+ FULL( _A1 ) */ "_A1"."SHARD_KEY" FROM
"PUBLIC"."T_SHARD_1"@LOCAL AS "_A1" WHERE "_A1"."SHARD_KEY" = :_V0
           TARGET DOMAIN : G1(G1N1,G1N2) 1 rows, G2(G2N1,G2N2) 0 rows,
G3(G3N1,G3N2) 0 rows
             SHARD KEY FILTER : ( T_SHARD_1.SHARD_KEY = $V5.$C0 )
     3  -  RANGE SHARD ( # 3 )
           READ COLUMN : T_SHARD_1.SHARD_KEY
             PHYSICAL FILTER : T_SHARD_1.SHARD_KEY = $V5.$C0
     5  -  COLUMN : 111 AS $C0
     6  -  TARGET : 111
     7  -  READ COLUMN : NOTHING


<<<  end print plan
```

Cluster Pusher

Cluster pusher node创建并管理虚拟表以执行cluster puller node的有效查询

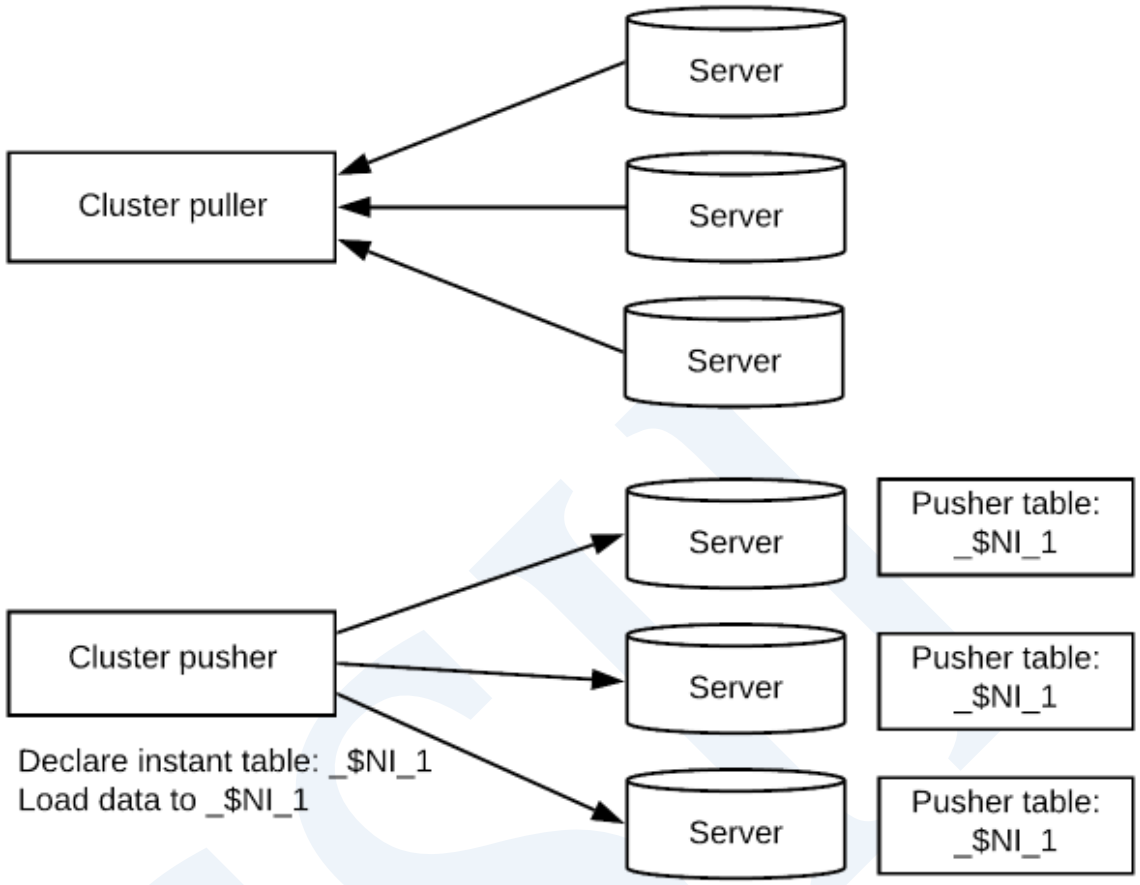


Figure 2-7 Cluster puller vs cluster pusher

Cluster puller node收集数据Cluster pusher node将数据以新的表形式进行分配

Cluster pusher node声明（declaration）pusher table并加载数据（loading data）

- 声明pusher table：在local server和remote server创建instant table
- 加载数据：将从cluster pusher node下级获取的数据加载到pusher table

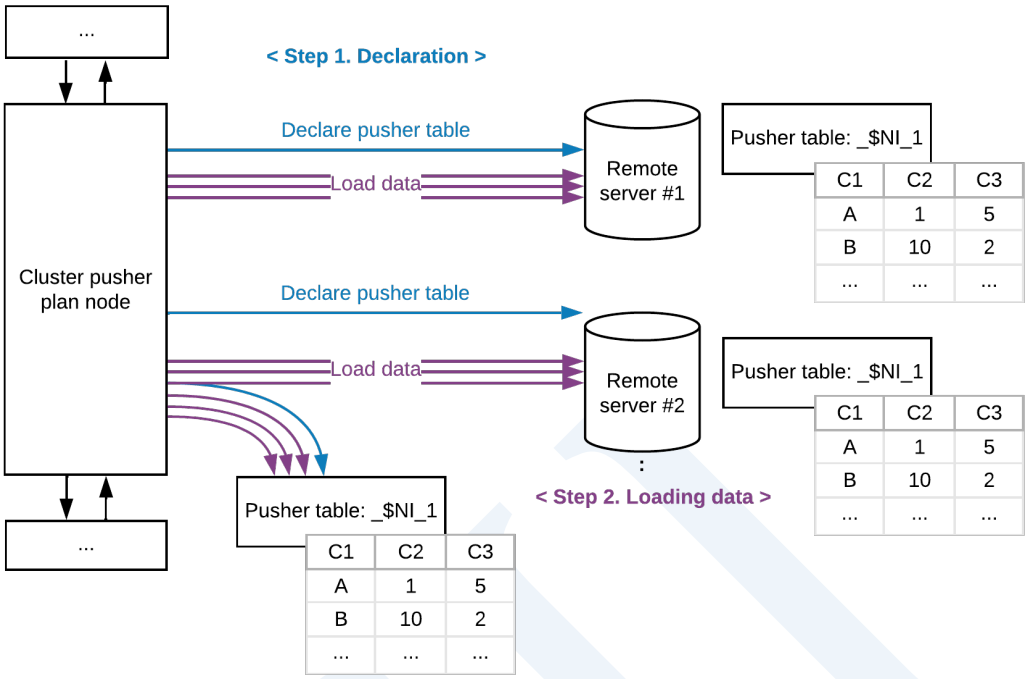


Figure 2-8 执行cluster pusher

以下为使用cluster pusher的示例

```
gSQL> \EXPLAIN PLAN ONLY

      SELECT A.c1

        FROM t_shard_1 AS A, t_shard_1 AS B

       WHERE A.shard_key = B.c1;


>>>  start print plan


< Execution Plan >

================================================================================

|  IDX  |  NODE DESCRIPTION                                      |      ROWS |

--------------------------------------------------------------------------------
```

```
|    0  |  SELECT STATEMENT                                  |         0 |
|    1  |    QUERY BLOCK ("$QB_IDX_2")                       |         0 |
|    2  |      SINGLE CLUSTER                                |         0 |
|    3  |        CLUSTER PUSHER ("_$NI_7")                   |         0
|
|    4  |          PLAN BASED CLUSTER                        |         0 |
|    5  |            INDEX ACCESS ("T_SHARD_1" AS B, "IDX")  |         0 |
|    6  |        HASH JOIN (INNER JOIN)                      |         0 |
|    7  |          TABLE ACCESS ("T_SHARD_1" AS A)           |         0 |
|    8  |          HASH JOIN INSTANT                         |         0 |
|    9  |            PUSHER TABLE ACCESS ("_$NI_7")          |         0 |
=============================================================================


     1  -  TARGET : A.C1
     2  -  SQL : SELECT /*+ KEEP_JOINED_TABLE USE_HASH_IN( _A1, 100 )
FULL( _A2 ) FULL( _A1 ) */ "_A2"."C1" FROM ( "PUBLIC"."T_SHARD_1"@LOCAL AS
"_A2" INNER JOIN "SESSION_SCHEMA"."_$NI_7"@LOCAL AS "_A1" ON "_A1"."C1" =
"_A2"."SHARD_KEY") ALIAS "_A3"
           TARGET DOMAIN : G1(G1N1,G1N2) 0 rows, G2(G2N1,G2N2) 0 rows,
G3(G3N1,G3N2) 0 rows
     3  -  SQL : DECLARE INSTANT TABLE "SESSION_SCHEMA"."_$NI_7" ( "C1"
NUMBER(10, 0) )
           COLUMN : B.C1 AS C1
           SHARDED : B.C1
           TARGET DOMAIN : G1(G1N1,G1N2) 0 rows, G2(G2N1,G2N2) 0 rows,
```

```
G3(G3N1,G3N2) 0 rows
     4  -  SQL : SELECT /*+ INDEX( _A1, "PUBLIC"."IDX" ) */ "_A1"."C1"
FROM "PUBLIC"."T_SHARD_1"@LOCAL AS "_A1"
           TARGET DOMAIN : G1(G1N1,G1N2) 0 rows, G2(G2N1,G2N2) 0 rows,
G3(G3N1,G3N2) 0 rows
     5  -  RANGE SHARD ( # 3 )
           READ INDEX COLUMN : B.C1
     6  -  JOINED COLUMN : A.C1
     7  -  RANGE SHARD ( # 3 )
           READ COLUMN : A.SHARD_KEY, A.C1
     8  -  HASH KEY : _$NI_7.C1
           READ KEY COLUMN : _$NI_7.C1
             HASH FILTER : _$NI_7.C1 = A.SHARD_KEY

<<<  end print plan
```

上述输出的执行信息中<Execution Plan>的idx为3的plan属于cluster pusher

Cluster pusher的NODE DESCRIPTION中输出pusher table名称

CLUSTER PUSHER的详细信息如下

- SQL：Pusher table声明查询
- COLUMN：Pusher table的各个column的源expression
- SHARDED：用作Pusher table数据的分配标准的pusher table的column
- TARGET DOMAIN：构成pusher table的对象group和member以及要向该group传输的数据

数量

Pusher Table

Pusher table是为了有效使用cluster puller由查询处理器构成的用户查询单位instant table用于以relation形式部署收集在driver server的数据

构成的pusher table与一般table相同可在generated query中引用

以下为执行不使用pusher table的join查询的示例

```
gSQL> \EXPLAIN PLAN
      SELECT /*+ LOCAL_JOIN( t_shard_1 ) */ t_shard_1.c1
        FROM t_shard_1, t_shard_2
       WHERE t_shard_1.shard_key = t_shard_2.shard_key;


C1
--
 1
 3
 2


3 rows selected.


>>>  start print plan


< Execution Plan >
```

```
=============================================================================
|  IDX   |  NODE DESCRIPTION                          |                ROWS |
-----------------------------------------------------------------------------
|    0   |  SELECT STATEMENT                          |                   3 |
|    1   |    QUERY BLOCK ("$QB_IDX_2")               |                   3 |
|    2   |      HASH JOIN (INNER JOIN)                |                   3 |
|    3   |        PLAN BASED CLUSTER                  | LOCAL/REMOTE      3 |
|    4   |          TABLE ACCESS ("T_SHARD_1")        |                   1 |
|    5   |        HASH JOIN INSTANT                   |                   3 |
|    6   |          PLAN BASED CLUSTER                | LOCAL/REMOTE      3 |
|    7   |            TABLE ACCESS ("T_SHARD_2")      |                   1 |
=============================================================================


     1  -  TARGET : T_SHARD_1.C1

     2  -  JOINED COLUMN : T_SHARD_1.C1

     3  -  SQL : SELECT /*+ FULL( _A1 ) */ "_A1"."SHARD_KEY", "_A1"."C1"
FROM "PUBLIC"."T_SHARD_1"@LOCAL AS "_A1"

           TARGET DOMAIN : G1(G1N1,G1N2) 1 rows, G2(G2N1,G2N2) 1 rows,
G3(G3N1,G3N2) 1 rows

     4  -  RANGE SHARD ( # 3 )

           READ COLUMN : T_SHARD_1.SHARD_KEY, T_SHARD_1.C1

     5  -  HASH KEY : T_SHARD_2.SHARD_KEY

           READ KEY COLUMN : T_SHARD_2.SHARD_KEY

             HASH FILTER : T_SHARD_2.SHARD_KEY = T_SHARD_1.SHARD_KEY

     6  -  SQL : SELECT /*+ FULL( _A1 ) */ "_A1"."SHARD_KEY" FROM
```

```
"PUBLIC"."T_SHARD_2"@LOCAL AS "_A1"
           TARGET DOMAIN : G1(G1N1,G1N2) 1 rows, G3(G3N1,G3N2) 2 rows
     7  -  RANGE SHARD ( # 4 )
           READ COLUMN : T_SHARD_2.SHARD_KEY


<<<  end print plan
```

如上执行join时可看到使用了两个cluster puller plan这是因为无法通过一个generated query处理拥有互不相同的sharding策略的两个表的join

以下为使用pusher join执行join查询的示例

```
gSQL> \EXPLAIN PLAN
SELECT /*+ REMOTE_JOIN( t_shard_1 ) */ t_shard_1.c1
  FROM t_shard_1, t_shard_2
 WHERE t_shard_1.shard_key = t_shard_2.shard_key;


C1
--
 1
 2
 3


3 rows selected.


>>>  start print plan
```

```
< Execution Plan >
================================================================================
| IDX | NODE DESCRIPTION                                |            ROWS |
--------------------------------------------------------------------------------
|   0 | SELECT STATEMENT                                |               3 |
|   1 |  QUERY BLOCK ("$QB_IDX_2")                      |               3 |
|   2 |   SINGLE CLUSTER                                | LOCAL/REMOTE  3 |
|   3 |    CLUSTER PUSHER ("_$NI_7")                    |               3 |
|   4 |     PLAN BASED CLUSTER                          | LOCAL/REMOTE  3 |
|   5 |      TABLE ACCESS ("T_SHARD_2")                 |               1 |
|   6 |    SELECT STATEMENT                             |               1 |
|   7 |     QUERY BLOCK ("$QB_IDX_2")                   |               1 |
|   8 |      HASH JOIN (INNER JOIN)                     |               1 |
|   9 |       TABLE ACCESS ("T_SHARD_1" AS _A2)         |               1 |
|  10 |       HASH JOIN INSTANT                         |               1 |
|  11 |        PUSHER TABLE ACCESS ("_$NI_7" AS _A1)    |               1 |
================================================================================

     1  -  TARGET : T_SHARD_1.C1
     2  -  SQL : SELECT /*+ KEEP_JOINED_TABLE USE_HASH_IN( _A1, 10 ) FULL( _A2 )
FULL( _A1 ) */ "_A2"."C1" FROM ( "PUBLIC"."T_SHARD_1"@LOCAL AS "_A2" INNER JOIN
"SESSION_SCHEMA"."_$NI_7"@LOCAL AS "_A1" ON "_A1"."SHARD_KEY" =
"_A2"."SHARD_KEY") ALIAS "_A3"
           TARGET DOMAIN : G1(G1N1,G1N2) 1 rows, G2(G2N1,G2N2) 1 rows,
```

```
G3(G3N1,G3N2) 1 rows
     3  -  SQL : DECLARE INSTANT TABLE "SESSION_SCHEMA"."_$NI_7"
( "SHARD_KEY" NUMBER(10, 0) )
           COLUMN : T_SHARD_2.SHARD_KEY AS SHARD_KEY
           SHARDED : T_SHARD_2.SHARD_KEY
           TARGET DOMAIN : G1(G1N1,G1N2) 1 rows, G2(G2N1,G2N2) 1 rows,
G3(G3N1,G3N2) 1 rows
     4  -  SQL : SELECT /*+ FULL( _A1 ) */ "_A1"."SHARD_KEY" FROM
"PUBLIC"."T_SHARD_2"@LOCAL AS "_A1"
           TARGET DOMAIN : G1(G1N1,G1N2) 1 rows, G3(G3N1,G3N2) 2 rows
     5  -  RANGE SHARD ( # 4 )
           READ COLUMN : T_SHARD_2.SHARD_KEY
     7  -  TARGET : _A2.C1
     8  -  JOINED COLUMN : _A2.C1
     9  -  RANGE SHARD ( # 3 )
           READ COLUMN : _A2.SHARD_KEY, _A2.C1
    10  -  HASH KEY : _A1.SHARD_KEY
           READ KEY COLUMN : _A1.SHARD_KEY
             HASH FILTER : _A1.SHARD_KEY = _A2.SHARD_KEY
    11  -  READ COLUMN : _A1.SHARD_KEY


<<<  end print plan
```

上述执行结果构成的cluster puller的generated query包含join

在SINGLE CLUSTER下级构成的CLUSTER PUSHER通过DECLARE语句定义pusher table "_$NI_7"

在CLUSTER PUSHER使用t_shard_1的sharding策略将从t_shard_2表收集的数据分散到本地服务器和远程服务器的pusher tablet_shard_1和pusher table使用相同的sharding策略因此可以用一个generated query表示并处理这两个表的join

Pusher table拥有如下特征

- 在cluster pusher plan node声明pusher table并加载数据
- 由包含在SESSION_SCHEMA的table构成
- 无法构成索引
- 一个用户查询中可构成多个pusher table
- 不共享用户查询之间的pusher table
- 用户查询执行周期和pusher table管理周期相同
- 构成在各个服务器的pusher table拥有复制的数据或分配的数据

由复制的数据构成的Pusher Table

根据cluster puller的执行pusher table的数据构成会发生变化

如下为了处理outer join构成的pusher table中所有服务器同样拥有t_shard_2的所有数据并处理包含outerjoin的generated query

```
gSQL> \EXPLAIN PLAN ONLY
      SELECT A.c1
        FROM t_shard_1 AS A LEFT OUTER JOIN t_shard_2 AS B ON A.c1 = B.c1;


>>>  start print plan
```

```
< Execution Plan >

================================================================================
|  IDX  |  NODE DESCRIPTION                                |      ROWS |
--------------------------------------------------------------------------------
|    0  |  SELECT STATEMENT                                |         0 |
|    1  |    QUERY BLOCK ("$QB_IDX_2")                     |         0 |
|    2  |      SINGLE CLUSTER                              |         0 |
|    3  |        CLUSTER PUSHER ("_$NI_5")                 |         0 |
|    4  |          PLAN BASED CLUSTER                      |         0 |
|    5  |            TABLE ACCESS ("T_SHARD_2" AS B)       |         0 |
|    6  |        HASH JOIN (INVERTED LEFT OUTER JOIN)      |         0 |
|    7  |          PUSHER TABLE ACCESS ("_$NI_5")          |         0 |
|    8  |          HASH JOIN INSTANT                       |         0 |
|    9  |            INDEX ACCESS ("T_SHARD_1" AS A, "IDX")|         0 |
================================================================================


     1  -  TARGET : A.C1
     2  -  SQL : SELECT /*+ KEEP_JOINED_TABLE USE_HASH_IN( _A1, 100 )
FULL( _A2 ) INDEX( _A1, "PUBLIC"."IDX" ) */ "_A1"."C1" FROM
( "PUBLIC"."T_SHARD_1"@"G1N1"|"G1N2"|"G2N1"|"G2N2"|"G3N1"|"G3N2" AS "_A1"
LEFT OUTER JOIN "SESSION_SCHEMA"."_$NI_5"@LOCAL AS "_A2" ON "_A1"."C1" =
"_A2"."C1") ALIAS "_A3"
           TARGET DOMAIN : G1(G1N1,G1N2) 0 rows, G2(G2N1,G2N2) 0 rows,
G3(G3N1,G3N2) 0 rows
     3  -  SQL : DECLARE INSTANT TABLE "SESSION_SCHEMA"."_$NI_5" ( "C1"
```

```
NUMBER(10, 0) )
            COLUMN : B.C1 AS C1
            CLONED
            TARGET DOMAIN : G1(G1N1,G1N2) 0 rows, G2(G2N1,G2N2) 0 rows,
G3(G3N1,G3N2) 0 rows
     4   -  SQL : SELECT /*+ FULL( _A1 ) */ "_A1"."C1" FROM
"PUBLIC"."T_SHARD_2"@LOCAL AS "_A1"
            TARGET DOMAIN : G1(G1N1,G1N2) 0 rows, G3(G3N1,G3N2) 0 rows
     5   -  RANGE SHARD ( # 4 )
            READ COLUMN : B.C1
     6   -  JOINED COLUMN : A.C1
     8   -  HASH KEY : A.C1
            READ KEY COLUMN : A.C1
              HASH FILTER : A.C1 = _$NI_5.C1
     9   -  RANGE SHARD ( # 3 )
            READ INDEX COLUMN : A.C1


<<<  end print plan
```

由复制数据构成的pusher table如上述结果输出CLONED

由分配的数据构成的Pusher Table

以下为以t_shard_1的sharding key作为join条件处理outer join的示例

```
gSQL> \EXPLAIN PLAN ONLY
```

```
      SELECT A.c1
        FROM t_shard_1 AS A LEFT OUTER JOIN t_shard_2 AS B ON A.shard_key
= B.c1;


>>>  start print plan


< Execution Plan >
=============================================================================
|  IDX  |  NODE DESCRIPTION                          |          ROWS |
-----------------------------------------------------------------------------
|    0  |  SELECT STATEMENT                          |             0 |
|    1  |    QUERY BLOCK ("$QB_IDX_2")               |             0 |
|    2  |      SINGLE CLUSTER                        |             0 |
|    3  |        CLUSTER PUSHER ("_$NI_7")           |             0 |
|    4  |          PLAN BASED CLUSTER                |             0 |
|    5  |            TABLE ACCESS ("T_SHARD_2" AS B) |             0 |
|    6  |        HASH JOIN (LEFT OUTER JOIN)         |             0 |
|    7  |          TABLE ACCESS ("T_SHARD_1" AS A)   |             0 |
|    8  |          HASH JOIN INSTANT                 |             0 |
|    9  |            PUSHER TABLE ACCESS ("_$NI_7")  |             0 |
=============================================================================


     1  -  TARGET : A.C1
     2  -  SQL : SELECT /*+ KEEP_JOINED_TABLE USE_HASH_IN( _A1, 10 )
FULL( _A2 ) FULL( _A1 ) */ "_A2"."C1" FROM
```

```
( "PUBLIC"."T_SHARD_1"@"G1N1"|"G1N2"|"G2N1"|"G2N2"|"G3N1"|"G3N2" AS "_A2"
LEFT OUTER JOIN "SESSION_SCHEMA"."_$NI_7"@LOCAL AS "_A1" ON "_A1"."C1" =
"_A2"."SHARD_KEY") ALIAS "_A3"
           TARGET DOMAIN : G1(G1N1,G1N2) 0 rows, G2(G2N1,G2N2) 0 rows,
G3(G3N1,G3N2) 0 rows
     3  -  SQL : DECLARE INSTANT TABLE "SESSION_SCHEMA"."_$NI_7" ( "C1"
NUMBER(10, 0) )
           COLUMN : B.C1 AS C1
           SHARDED : B.C1
           TARGET DOMAIN : G1(G1N1,G1N2) 0 rows, G2(G2N1,G2N2) 0 rows,
G3(G3N1,G3N2) 0 rows
     4  -  SQL : SELECT /*+ FULL( _A1 ) */ "_A1"."C1" FROM
"PUBLIC"."T_SHARD_2"@LOCAL AS "_A1"
           TARGET DOMAIN : G1(G1N1,G1N2) 0 rows, G3(G3N1,G3N2) 0 rows
     5  -  RANGE SHARD ( # 4 )
           READ COLUMN : B.C1
     6  -  JOINED COLUMN : A.C1
     7  -  RANGE SHARD ( # 3 )
           READ COLUMN : A.SHARD_KEY, A.C1
     8  -  HASH KEY : _$NI_7.C1
           READ KEY COLUMN : _$NI_7.C1
             HASH FILTER : _$NI_7.C1 = A.SHARD_KEY


<<<  end print plan
```

Pusher table按照各个group分别拥有t_shard_2的数据根据在t_shard_1的sharding key构成的sharding策略以t_shard_2的C1 column为准分配数据

由分配的数据构成的pusher table如上述结果输出SHARDED

各SELECT语句的Cluster查询处理

如上所述为了处理集群的SELECT对**Cluster Pusher**和**Cluster Puller**plan node进行了说明现在开始说明使用此的SELECT的各语句的集群查询处理

FROM语句（Single Table）

Single table的查询处理分为sharded table中的处理和cloned table中的处理在sharded table中处理时分开存储在n个group因此默认向本地服务器和远程服务器请求查询并将其整合为一个后创建结果集合向远程服务器请求查询并获取结果时SUNDB使用plan based cluster或single cluster plan node其cluster puller向本地服务器和远程服务器同时发送查询并收集其结果后创建结果集合

以下为在为sharded table的part table中处理查询的示例

```
gSQL> \EXPLAIN PLAN
      SELECT p_name, p_brand, p_type, cluster_group_id
        FROM part;


P_NAME P_BRAND    P_TYPE CLUSTER_GROUP_ID
------ ---------- ------ ----------------
Part#3 Brand#2    STEEL                 1
```

```
Part#2 Brand#1    NICKEL                2

Part#5 Brand#3    STEEL                 2

Part#1 Brand#1    COPPER                3

Part#4 Brand#3    NICKEL                3


5 rows selected.


>>>  start print plan


< Execution Plan >

================================================================

|  IDX  |  NODE DESCRIPTION              |               ROWS |

----------------------------------------------------------------

|    0  |  SELECT STATEMENT              |                  5 |

|    1  |    QUERY BLOCK ("$QB_IDX_2")   |                  5 |

|    2  |      PLAN BASED CLUSTER        | LOCAL/REMOTE     5 |

|    3  |        TABLE ACCESS ("PART")   |                  1 |

================================================================


     1  -  TARGET : PART.P_NAME, PART.P_BRAND, PART.P_TYPE,

PART.CLUSTER_GROUP_ID

     2  -  SQL : SELECT /*+ FULL( _A1 ) */ "_A1"."CLUSTER_GROUP_ID",

"_A1"."P_NAME", "_A1"."P_BRAND", "_A1"."P_TYPE" FROM "PUBLIC"."PART"@LOCAL

AS "_A1"

           TARGET DOMAIN : G1(G1N1,G1N2) 1 rows, G2(G2N1,G2N2) 2 rows,
```

```
G3(G3N1,G3N2) 2 rows

     3  -  HASH SHARD ( # 3 )

           READ COLUMN : PART.P_NAME, PART.P_BRAND, PART.P_TYPE


<<<  end print plan
```

以cloned strategy作为AT CLUSTER WIDE策略创建cloned table时所有节点都拥有副本因此可以在大部分本地服务器处理cloned table的查询但添加新的group或member时该group或member中没有cloned table的数据因此需要从远程服务器获取数据此时构成cluster puller Cloned strategy的详细说明参考**Cloned Strategy**

以下为在为cloned table的supplier table中处理查询的示例

```
gSQL> \EXPLAIN PLAN

      SELECT s_name, s_nationkey

        FROM supplier;


S_NAME                    S_NATIONKEY

------------------------- ---------------

Supplier#1                FRANCE

Supplier#2                KOREA

Supplier#3                GERMANY

Supplier#4                UNITED STATES

Supplier#5                CANADA


5 rows selected.
```

```
>>>  start print plan


< Execution Plan >

=============================================================================
|  IDX  |  NODE DESCRIPTION                          |            ROWS |
-----------------------------------------------------------------------------
|    0  |  SELECT STATEMENT                          |               5 |
|    1  |    QUERY BLOCK ("$QB_IDX_2")               |               5 |
|    2  |      TABLE ACCESS ("SUPPLIER")             |               5 |
=============================================================================


     1  -  TARGET : SUPPLIER.S_NAME, SUPPLIER.S_NATIONKEY
     2  -  CLONED
           READ COLUMN : SUPPLIER.S_NAME, SUPPLIER.S_NATIONKEY


<<<  end print plan
```

如上可以从本地服务器获取所有Supplier table的数据因此不构成cluster puller

以下为使用domain在为cloned table的supplier table中处理查询的示例

```
gSQL> \EXPLAIN PLAN
      SELECT s_name, s_nationkey
        FROM supplier@G2;
```

```
S_NAME                    S_NATIONKEY
------------------------- ---------------
Supplier#1                FRANCE
Supplier#2                KOREA
Supplier#3                GERMANY
Supplier#4                UNITED STATES
Supplier#5                CANADA

5 rows selected.

>>>  start print plan

< Execution Plan >
=============================================================================
|  IDX  |  NODE DESCRIPTION                      |               ROWS |
-----------------------------------------------------------------------------
|    0  |  SELECT STATEMENT                      |                  5 |
|    1  |    QUERY BLOCK ("$QB_IDX_2")           |                  5 |
|    2  |      PLAN BASED CLUSTER                | REMOTE ONLY      5 |
|    3  |        TABLE ACCESS ("SUPPLIER")       |                  0 |
=============================================================================

     1  -  TARGET : SUPPLIER.S_NAME, SUPPLIER.S_NATIONKEY
     2  -  SQL : SELECT /*+ FULL( _A1 ) */ "_A1"."S_NAME",
"_A1"."S_NATIONKEY" FROM "PUBLIC"."SUPPLIER"@LOCAL AS "_A1"
```

```
            TARGET DOMAIN : G2(G2N1,G2N2) 5 rows
      3   -  CLONED
            READ COLUMN : SUPPLIER.S_NAME, SUPPLIER.S_NATIONKEY


<<<  end print plan
```

如上为了从远程服务器获取supplier table的数据构成cluster puller

FROM语句（Join）

用于Cluster puller plan node的数据收集以如下两种形式中的一个执行

- 在一个group收集数据：仅由cloned table构成的查询或cluster domain限于一个group的情况
- 在多个group收集数据：包含sharded table的查询

使用Cluster puller plan node的join也以上述两种形式执行

在一个group通过收集数据执行join时两个cloned table之间的join通过在一个group收集数据构成cluster puller

如果两个cloned table之间没有共同的cluster domian时可使用如下两种方法中的一个执行

第一个方法是对各join对象table构成cluster puller后执行join

```
gSQL> \EXPLAIN PLAN
      SELECT s_suppkey, n_name
        FROM supplier@g2, nation@g3
       WHERE s_nationkey = n_nationkey;
```

```
S_SUPPKEY N_NAME
--------- -------------------------
        1 FRANCE
        2 INDIA
        3 GERMANY
        4 CANADA
        5 UNITED STATES

5 rows selected.

>>>  start print plan

< Execution Plan >
================================================================================
|  IDX  |  NODE DESCRIPTION                          |              ROWS |
--------------------------------------------------------------------------------
|    0  |  SELECT STATEMENT                          |                 5 |
|    1  |    QUERY BLOCK ("$QB_IDX_2")               |                 5 |
|    2  |      HASH JOIN (INNER JOIN)                |                 5 |
|    3  |        PLAN BASED CLUSTER                  | REMOTE ONLY     5 |
|    4  |          TABLE ACCESS ("SUPPLIER")         |                 0 |
|    5  |        HASH JOIN INSTANT                   |                 5 |
|    6  |          PLAN BASED CLUSTER                | REMOTE ONLY    30 |
|    7  |            TABLE ACCESS ("NATION")         |                 0 |
```

```
================================================================================


     1  -  TARGET : SUPPLIER.S_SUPPKEY, NATION.N_NAME

     2  -  JOINED COLUMN : SUPPLIER.S_SUPPKEY, NATION.N_NAME

     3  -  SQL : SELECT /*+ FULL( _A1 ) */ "_A1"."S_SUPPKEY",
"_A1"."S_NATIONKEY" FROM "PUBLIC"."SUPPLIER"@LOCAL AS "_A1"

           TARGET DOMAIN : G2(G2N1,G2N2) 5 rows

     4  -  CLONED

           READ COLUMN : SUPPLIER.S_SUPPKEY, SUPPLIER.S_NATIONKEY

     5  -  HASH KEY : NATION.N_NATIONKEY

           RECORD COLUMN : NATION.N_NAME

           READ KEY COLUMN : NATION.N_NATIONKEY, NATION.N_NAME

             HASH FILTER : NATION.N_NATIONKEY = SUPPLIER.S_NATIONKEY

     6  -  SQL : SELECT /*+ FULL( _A1 ) */ "_A1"."N_NATIONKEY",
"_A1"."N_NAME" FROM "PUBLIC"."NATION"@LOCAL AS "_A1"

           TARGET DOMAIN : G3(G3N1,G3N2) 30 rows

     7  -  CLONED

           READ COLUMN : NATION.N_NATIONKEY, NATION.N_NAME


<<<  end print plan
```

但是如上执行时需要获取两个表的数据

第二种方法是构成一个表的pusher tabl并执行包含pusher table的join查询此时如下在一个group中执行数据收集

```
gSQL> \EXPLAIN PLAN
      SELECT /*+ REMOTE_JOIN( supplier ) */ s_suppkey, n_name
        FROM supplier@g2, nation@g3
       WHERE s_nationkey = n_nationkey;

S_SUPPKEY N_NAME
--------- -------------------------
        4 CANADA
        1 FRANCE
        3 GERMANY
        2 INDIA
        5 UNITED STATES

5 rows selected.

>>>  start print plan

< Execution Plan >
================================================================================
=
|  IDX  |  NODE DESCRIPTION                          |              ROWS
|
--------------------------------------------------------------------------------
-
|    0  |  SELECT STATEMENT                          |                 5
```

```
|
|    1  |    QUERY BLOCK ("$QB_IDX_2")                |                5
|
|    2  |      SINGLE CLUSTER                          | REMOTE ONLY    5
|
|    3  |        CLUSTER PUSHER ("_$NI_7")             |                5 |
|    4  |          PLAN BASED CLUSTER                  | REMOTE ONLY    5
|
|    5  |            TABLE ACCESS ("SUPPLIER")         |                0
|
|    6  |        HASH JOIN (INNER JOIN)                |                0
|
|    7  |          TABLE ACCESS ("NATION")             |                0
|
|    8  |          HASH JOIN INSTANT                   |                0
|
|    9  |            PUSHER TABLE ACCESS ("_$NI_7")    |                0
|
================================================================================
=


     1  -  TARGET : _$NI_7.S_SUPPKEY, NATION.N_NAME
     2  -  SQL : SELECT /*+ KEEP_JOINED_TABLE USE_HASH_IN( _A1, 5 )
FULL( _A2 ) FULL( _A1 ) */ "_A1"."S_SUPPKEY", "_A2"."N_NAME" FROM
( "PUBLIC"."NATION"@LOCAL AS "_A2" INNER JOIN "SESSION_SCHEMA"."_$NI_7"@LOCAL
```

```
AS "_A1" ON "_A1"."S_NATIONKEY" = "_A2"."N_NATIONKEY") ALIAS "_A3"
            TARGET DOMAIN : G3(G3N1,G3N2) 5 rows
     3   -  SQL : DECLARE INSTANT TABLE "SESSION_SCHEMA"."_$NI_7"
( "S_NATIONKEY" NUMBER(10, 0), "S_SUPPKEY" NUMBER(10, 0) )
            COLUMN : SUPPLIER.S_NATIONKEY AS S_NATIONKEY,
SUPPLIER.S_SUPPKEY AS S_SUPPKEY
            CLONED
            TARGET DOMAIN : G3(G3N1,G3N2) 5 rows
     4   -  SQL : SELECT /*+ FULL( _A1 ) */ "_A1"."S_SUPPKEY",
"_A1"."S_NATIONKEY" FROM "PUBLIC"."SUPPLIER"@LOCAL AS "_A1"
            TARGET DOMAIN : G2(G2N1,G2N2) 5 rows
     5   -  CLONED
            READ COLUMN : SUPPLIER.S_SUPPKEY, SUPPLIER.S_NATIONKEY
     6   -  JOINED COLUMN : _$NI_7.S_SUPPKEY, NATION.N_NAME
     7   -  CLONED
            READ COLUMN : NATION.N_NATIONKEY, NATION.N_NAME
     8   -  HASH KEY : _$NI_7.S_NATIONKEY
            RECORD COLUMN : _$NI_7.S_SUPPKEY
            READ KEY COLUMN : _$NI_7.S_NATIONKEY, _$NI_7.S_SUPPKEY
              HASH FILTER : _$NI_7.S_NATIONKEY = NATION.N_NATIONKEY


<<<  end print plan
```

通过在多个group中通过收集数据执行join时sharded table的数据分散在多个group因此处理相关查询时需要从多个group收集数据

包含sharded table的join分类如下

- Case 1: Joining sharded table and cloned table
  - 分配了sharded table的数据的所有group中分配了cloned table的数据的情况
- Case 2: Joining sharded table and cloned table
  - 分配了sharded table的数据的所有group中至少有一个没有分配cloned table的数据的情况
- Case 3: Joining sharded table and sharded table
  - 两个sharded table的sharding key之间有equi-join条件的情况
- Case 4: Joining sharded table and sharded table
  - 存在使用一个sharded table的sharding key的equi-join条件的情况
- Case 5: Joining sharded table and sharded table
  - 没有使用sharding key的equi-join条件的情况

Case 1的情况分配sharded table的数据的所有group中有cloned table的数据时如下汇集在各个group中的join执行结果为整体join结果

```
gSQL> \EXPLAIN PLAN
      SELECT ps_partkey, s_name
        FROM partsupp, supplier
       WHERE ps_suppkey = s_suppkey;


PS_PARTKEY S_NAME
---------- -------------------------
         3 Supplier#1
         3 Supplier#4
```

```
         2 Supplier#2

         2 Supplier#5

         5 Supplier#1

         5 Supplier#4

         1 Supplier#2

         1 Supplier#3

         4 Supplier#3

         4 Supplier#5


10 rows selected.


>>>  start print plan


< Execution Plan >

================================================================================
=
|IDX| NODE DESCRIPTION                                |            ROWS
|
--------------------------------------------------------------------------------
-
| 0 | SELECT STATEMENT                                |              10
|
| 1 |  QUERY BLOCK ("$QB_IDX_2")                      |              10
|
| 2 |    PLAN BASED CLUSTER                            | LOCAL/REMOTE 10
```

```
|
| 3 |    HASH JOIN (INNER JOIN)                          |              2
|
| 4 |     INDEX ACCESS ("PARTSUPP", "PARTSUPP_PK_INDEX")| (    2)       2
|
| 5 |     HASH JOIN INSTANT                              |              2
|
| 6 |      TABLE ACCESS ("SUPPLIER")                     |              5
|
=============================================================================
=

     1  -  TARGET : PARTSUPP.PS_PARTKEY, SUPPLIER.S_NAME
     2  -  SQL : SELECT /*+ KEEP_JOINED_TABLE USE_HASH_IN( _A1, 5 )
INDEX( _A2, "PUBLIC"."PARTSUPP_PK_INDEX" ) FULL( _A1 ) */
"_A2"."PS_PARTKEY", "_A1"."S_NAME"  FROM ( "PUBLIC"."PARTSUPP"@LOCAL AS
"_A2" INNER JOIN "PUBLIC"."SUPPLIER"@LOCAL AS "_A1" ON "_A1"."S_SUPPKEY" =
"_A2"."PS_SUPPKEY") ALIAS "_A3"
           TARGET DOMAIN : G1(G1N1,G1N2) 2 rows, G2(G2N1,G2N2) 4 rows,
G3(G3N1,G3N2) 4 rows
     3  -  JOINED COLUMN : PARTSUPP.PS_PARTKEY, SUPPLIER.S_NAME
     4  -  HASH SHARD ( # 3 )
           READ INDEX COLUMN : PARTSUPP.PS_PARTKEY, PARTSUPP.PS_SUPPKEY
     5  -  HASH KEY : SUPPLIER.S_SUPPKEY
           RECORD COLUMN : SUPPLIER.S_NAME
```

```
            READ KEY COLUMN : SUPPLIER.S_SUPPKEY, SUPPLIER.S_NAME
              HASH FILTER : SUPPLIER.S_SUPPKEY = PARTSUPP.PS_SUPPKEY
            FETCH ONE ROW
      6   -  CLONED
            READ COLUMN : SUPPLIER.S_SUPPKEY, SUPPLIER.S_NAME


<<<  end print plan
```

Case 2的情况无法构成包含join的generated query此时如下可对各个join对象table构成cluster puller后执行join

```
gSQL> \EXPLAIN PLAN
      SELECT ps_partkey, s_name
        FROM partsupp, supplier@G2|G3
       WHERE ps_suppkey = s_suppkey;


PS_PARTKEY S_NAME
---------- -------------------------
         3 Supplier#1
         3 Supplier#4
         1 Supplier#2
         1 Supplier#3
         4 Supplier#3
         4 Supplier#5
         2 Supplier#2
         2 Supplier#5
```

```
         5 Supplier#1

         5 Supplier#4


10 rows selected.


>>>  start print plan


< Execution Plan >

================================================================================
=
|IDX| NODE DESCRIPTION                                  |             ROWS
|
--------------------------------------------------------------------------------
-
| 0 |SELECT STATEMENT                                  |               10
|
| 1 | QUERY BLOCK ("$QB_IDX_2")                        |               10
|
| 2 |  HASH JOIN (INNER JOIN)                          |               10 |
| 3 |   PLAN BASED CLUSTER                             | LOCAL/REMOTE 10
|
| 4 |    INDEX ACCESS ("PARTSUPP", "PARTSUPP_PK_INDEX") |(    2)         2
|
| 5 |   HASH JOIN INSTANT                              |               10
|
```

```
| 6 |     PLAN BASED CLUSTER                                  | REMOTE ONLY    5
|
| 7 |      TABLE ACCESS ("SUPPLIER")                          |              0
|
=============================================================================
=


     1  -  TARGET : PARTSUPP.PS_PARTKEY, SUPPLIER.S_NAME
     2  -  JOINED COLUMN : PARTSUPP.PS_PARTKEY, SUPPLIER.S_NAME
     3  -  SQL : SELECT /*+ INDEX( _A1, "PUBLIC"."PARTSUPP_PK_INDEX" ) */
"_A1"."PS_PARTKEY", "_A1"."PS_SUPPKEY" FROM "PUBLIC"."PARTSUPP"@LOCAL AS
"_A1"
           TARGET DOMAIN : G1(G1N1,G1N2) 2 rows, G2(G2N1,G2N2) 4 rows,
G3(G3N1,G3N2) 4 rows
     4  -  HASH SHARD ( # 3 )
           READ INDEX COLUMN : PARTSUPP.PS_PARTKEY, PARTSUPP.PS_SUPPKEY
     5  -  HASH KEY : SUPPLIER.S_SUPPKEY
           RECORD COLUMN : SUPPLIER.S_NAME
           READ KEY COLUMN : SUPPLIER.S_SUPPKEY, SUPPLIER.S_NAME
             HASH FILTER : SUPPLIER.S_SUPPKEY = PARTSUPP.PS_SUPPKEY
     6  -  SQL : SELECT /*+ FULL( _A1 ) */ "_A1"."S_SUPPKEY",
"_A1"."S_NAME" FROM "PUBLIC"."SUPPLIER"@LOCAL AS "_A1"
           TARGET DOMAIN : G2(G2N1,G2N2) 5 rows, G3(G3N1,G3N2) 0 rows
     7  -  CLONED
           READ COLUMN : SUPPLIER.S_SUPPKEY, SUPPLIER.S_NAME
```

```
<<<  end print plan
```

Case 3的情况两个表的sharding策略相同时汇集各个group的join执行结果为整体join结果如果两个表的sharding策略不相同时则为与case4相同的情况

两个表的sharding策略相同时如下执行case 3

```
gSQL> \EXPLAIN PLAN
      SELECT p_name, ps_suppkey
        FROM part, partsupp
       WHERE p_partkey = ps_partkey;

P_NAME PS_SUPPKEY
------ ----------
Part#3          4
Part#3          1
Part#2          5
Part#2          2
Part#5          4
Part#5          1
Part#1          3
Part#1          2
Part#4          5
Part#4          3
```

```
10 rows selected.


>>>  start print plan


< Execution Plan >

================================================================================
=
|IDX| NODE DESCRIPTION                                |            ROWS
|
--------------------------------------------------------------------------------
-
| 0 |SELECT STATEMENT                                |              10
|
| 1 | QUERY BLOCK ("$QB_IDX_2")                      |              10
|
| 2 |  PLAN BASED CLUSTER                             | LOCAL/REMOTE 10
|
| 3 |   HASH JOIN (INNER JOIN)                       |               2
|
| 4 |    TABLE ACCESS ("PART")                       |               1
|
| 5 |    HASH JOIN INSTANT                           |               2
|
| 6 |     INDEX ACCESS ("PARTSUPP", "PARTSUPP_PK_INDEX")| (      2)     2
|
```

```
================================================================================
=


     1  -  TARGET : PART.P_NAME, PARTSUPP.PS_SUPPKEY
     2  -  SQL : SELECT /*+ KEEP_JOINED_TABLE USE_HASH_IN( _A1, 10 )
FULL( _A2 ) INDEX( _A1, "PUBLIC"."PARTSUPP_PK_INDEX" ) */ "_A2"."P_NAME",
"_A1"."PS_SUPPKEY"  FROM ( "PUBLIC"."PART"@LOCAL AS "_A2" INNER JOIN
"PUBLIC"."PARTSUPP"@LOCAL AS "_A1" ON "_A1"."PS_PARTKEY" = "_A2"."P_PARTKEY")
ALIAS "_A3"
           TARGET DOMAIN : G1(G1N1,G1N2) 2 rows, G2(G2N1,G2N2) 4 rows,
G3(G3N1,G3N2) 4 rows
     3  -  JOINED COLUMN : PART.P_NAME, PARTSUPP.PS_SUPPKEY
     4  -  HASH SHARD ( # 3 )
           READ COLUMN : PART.P_PARTKEY, PART.P_NAME
     5  -  HASH KEY : PARTSUPP.PS_PARTKEY
           RECORD COLUMN : PARTSUPP.PS_SUPPKEY
           READ KEY COLUMN : PARTSUPP.PS_PARTKEY, PARTSUPP.PS_SUPPKEY
             HASH FILTER : PARTSUPP.PS_PARTKEY = PART.P_PARTKEY
     6  -  HASH SHARD ( # 3 )
           READ INDEX COLUMN : PARTSUPP.PS_PARTKEY, PARTSUPP.PS_SUPPKEY


<<<  end print plan
```

Case 4的情况对两个sharded table不能构成包含join的generated query创建在equi-join条件使用sharding key的table的对象表的pusher table与case 1同样处理

对两个sharded tablecase 4如下执行

```
gSQL> \EXPLAIN PLAN
      SELECT /*+ REMOTE_JOIN( part ) */ p_name, ps_suppkey
        FROM part, partsupp
       WHERE p_partkey = ps_suppkey;

P_NAME PS_SUPPKEY
------ ----------
Part#3          3
Part#3          3
Part#1          1
Part#1          1
Part#4          4
Part#4          4
Part#2          2
Part#2          2
Part#5          5
Part#5          5

10 rows selected.

>>>  start print plan
```

```
< Execution Plan >
================================================================================
=
|IDX| NODE DESCRIPTION                                |             ROWS
|
--------------------------------------------------------------------------------
-
|  0|SELECT STATEMENT                                 |              10
|
|  1| QUERY BLOCK ("$QB_IDX_2")                       |              10
|
|  2|  SINGLE CLUSTER                                 | LOCAL/REMOTE 10
|
|  3|   CLUSTER PUSHER ("_$NI_7")                     |              10 |
|  4|    PLAN BASED CLUSTER                           | LOCAL/REMOTE 10
|
|  5|     INDEX ACCESS ("PARTSUPP", "PARTSUPP_PK_INDEX")| (     2)      2
|
|  6|   SELECT STATEMENT                              |               2
|
|  7|    QUERY BLOCK ("$QB_IDX_2")                    |               2
|
|  8|     HASH JOIN (INNER JOIN)                      |               2
|
|  9|      TABLE ACCESS ("PART" AS _A2)               |               1
```

```
|
| 10|      HASH JOIN INSTANT                              |              2
|
| 11|       PUSHER TABLE ACCESS ("_$NI_7" AS _A1)         |              2
|
=============================================================================
=


     1  -  TARGET : PART.P_NAME, _$NI_7.PS_SUPPKEY
     2  -  SQL : SELECT /*+ KEEP_JOINED_TABLE USE_HASH_IN( _A1, 10 )
FULL( _A2 ) FULL( _A1 ) */ "_A2"."P_NAME", "_A1"."PS_SUPPKEY" FROM
( "PUBLIC"."PART"@LOCAL AS "_A2" INNER JOIN "SESSION_SCHEMA"."_$NI_7"@LOCAL AS
"_A1" ON "_A1"."PS_SUPPKEY" = "_A2"."P_PARTKEY") ALIAS "_A3"
           TARGET DOMAIN : G1(G1N1,G1N2) 2 rows, G2(G2N1,G2N2) 4 rows,
G3(G3N1,G3N2) 4 rows
     3  -  SQL : DECLARE INSTANT TABLE "SESSION_SCHEMA"."_$NI_7"
( "PS_SUPPKEY" NUMBER(10, 0) )
           COLUMN : PARTSUPP.PS_SUPPKEY AS PS_SUPPKEY
           SHARDED : PARTSUPP.PS_SUPPKEY
           TARGET DOMAIN : G1(G1N1,G1N2) 2 rows, G2(G2N1,G2N2) 4 rows,
G3(G3N1,G3N2) 4 rows
     4  -  SQL : SELECT /*+ INDEX( _A1, "PUBLIC"."PARTSUPP_PK_INDEX" ) */
"_A1"."PS_SUPPKEY" FROM "PUBLIC"."PARTSUPP"@LOCAL AS "_A1"
           TARGET DOMAIN : G1(G1N1,G1N2) 2 rows, G2(G2N1,G2N2) 4 rows,
G3(G3N1,G3N2) 4 rows
```

```
     5  -  HASH SHARD ( # 3 )
           READ INDEX COLUMN : PARTSUPP.PS_SUPPKEY
     7  -  TARGET : _A2.P_NAME, _A1.PS_SUPPKEY
     8  -  JOINED COLUMN : _A2.P_NAME, _A1.PS_SUPPKEY
     9  -  HASH SHARD ( # 3 )
           READ COLUMN : _A2.P_PARTKEY, _A2.P_NAME
    10  -  HASH KEY : _A1.PS_SUPPKEY
           READ KEY COLUMN : _A1.PS_SUPPKEY
             HASH FILTER : _A1.PS_SUPPKEY = _A2.P_PARTKEY
    11  -  READ COLUMN : _A1.PS_SUPPKEY


<<<  end print plan
```

Case 5的情况无法使用对两个sharded table的sharding策略此时如下将一个join对象table构成为cloned table形式的pusher table后执行join

```
gSQL> \EXPLAIN PLAN
      SELECT SUM( A.p_size )
        FROM part A, part B
       WHERE A.p_type = B.p_type;


SUM( A.P_SIZE )
---------------
            109


1 row selected.
```

```
>>>  start print plan

< Execution Plan >
================================================================================
=
|IDX|  NODE DESCRIPTION                                |           ROWS
|
--------------------------------------------------------------------------------
-
|  0|SELECT STATEMENT                                  |              1
|
|  1|  QUERY BLOCK ("$QB_IDX_2")                       |              1
|
|  2|    SINGLE CLUSTER                                | LOCAL/REMOTE 1
|
|  3|      CLUSTER PUSHER ("_$NI_6")                   |              5 |
|  4|        PLAN BASED CLUSTER                        | LOCAL/REMOTE 5
|
|  5|          TABLE ACCESS ("PART" AS B)              |              1
|
|  6|      SELECT STATEMENT                            |              1
|
|  7|        QUERY BLOCK ("$QB_IDX_2")                 |              1
|
```

```
|  8|          AGGREGATION BY HASH                          |              1
|
|  9|            HASH JOIN (INNER JOIN)                     |              2
|
| 10|              PUSHER TABLE ACCESS ("_$NI_6" AS _A2) |              5
|
| 11|              HASH JOIN INSTANT                        |              2
|
| 12|                TABLE ACCESS ("PART" AS _A1)        |              1
|
================================================================================
=


     1  -  TARGET : SUM( A.P_SIZE )
     2  -  SQL : SELECT /*+ KEEP_JOINED_TABLE USE_HASH_IN( _A1, 10 )
FULL( _A2 ) FULL( _A1 ) */ SUM( "_A1"."P_SIZE" ) FROM
( "SESSION_SCHEMA"."_$NI_6"@LOCAL AS "_A2" INNER JOIN "PUBLIC"."PART"@LOCAL AS
"_A1" ON "_A1"."P_TYPE" = "_A2"."P_TYPE") ALIAS "_A3"
           TARGET DOMAIN : G1(G1N1,G1N2) 1 rows, G2(G2N1,G2N2) 1 rows,
G3(G3N1,G3N2) 1 rows
           RE-AGGREGATION
             AGGREGATION : SUM( SUM( A.P_SIZE ) )
     3  -  SQL : DECLARE INSTANT TABLE "SESSION_SCHEMA"."_$NI_6" ( "P_TYPE"
VARCHAR(25 OCTETS) )
           COLUMN : B.P_TYPE AS P_TYPE
```

```
            CLONED
            TARGET DOMAIN : G1(G1N1,G1N2) 5 rows, G2(G2N1,G2N2) 5 rows,
G3(G3N1,G3N2) 5 rows
     4   -  SQL : SELECT /*+ FULL( _A1 ) */ "_A1"."P_TYPE" FROM
"PUBLIC"."PART"@LOCAL AS "_A1"
            TARGET DOMAIN : G1(G1N1,G1N2) 1 rows, G2(G2N1,G2N2) 2 rows,
G3(G3N1,G3N2) 2 rows
     5   -  HASH SHARD ( # 3 )
            READ COLUMN : B.P_TYPE
     7   -  TARGET : SUM( _A1.P_SIZE )
     8   -  AGGREGATION : SUM( _A1.P_SIZE )
     9   -  JOINED COLUMN : _A1.P_SIZE
    10   -  READ COLUMN : _A2.P_TYPE
    11   -  HASH KEY : _A1.P_TYPE
            RECORD COLUMN : _A1.P_SIZE
            READ KEY COLUMN : _A1.P_TYPE, _A1.P_SIZE
              HASH FILTER : _A1.P_TYPE = _A2.P_TYPE
    12   -  HASH SHARD ( # 3 )
            READ COLUMN : _A1.P_TYPE, _A1.P_SIZE


<<<  end print plan
```

FROM语句（Outer Join）

Outer join中的查询处理与**FROM语句（Join）**相同分为在一个group收集数据的方法和在多个group收集数据的方法

通过在一个group的数据收集的outer join仅可在仅由cloned table构成的查询或cluster domain仅限于一个group的情况适用

如下在一个group执行包含outer join的generated query后构成join结果

```
gSQL> \EXPLAIN PLAN
      SELECT s_suppkey, n_name
        FROM supplier@g2
             LEFT OUTER JOIN
             nation@g2
             ON s_nationkey = n_nationkey;

S_SUPPKEY N_NAME
--------- -------------------------
        4 CANADA
        1 FRANCE
        3 GERMANY
        2 INDIA
        5 UNITED STATES

5 rows selected.

>>>  start print plan

< Execution Plan >
=========================================================================
```

```
|  IDX  |  NODE DESCRIPTION                          |           ROWS |
-------------------------------------------------------------------------
|    0  |  SELECT STATEMENT                          |              5 |
|    1  |    QUERY BLOCK ("$QB_IDX_2")               |              5 |
|    2  |      SINGLE CLUSTER                        | REMOTE ONLY  5 |
|    3  |        HASH JOIN (INVERTED LEFT OUTER JOIN) |              0 |
|    4  |          TABLE ACCESS ("NATION")           |              0 |
|    5  |          HASH JOIN INSTANT                 |              0 |
|    6  |            TABLE ACCESS ("SUPPLIER")       |              0 |
=========================================================================


     1  -  TARGET : SUPPLIER.S_SUPPKEY, NATION.N_NAME
     2  -  SQL : SELECT /*+ KEEP_JOINED_TABLE USE_HASH_IN( _A1, 5 )
FULL( _A2 ) FULL( _A1 ) */ "_A1"."S_SUPPKEY", "_A2"."N_NAME" FROM
( "PUBLIC"."SUPPLIER"@"G2N1"|"G2N2" AS "_A1" LEFT OUTER JOIN
"PUBLIC"."NATION"@"G2N1"|"G2N2" AS "_A2" ON "_A1"."S_NATIONKEY" =
"_A2"."N_NATIONKEY") ALIAS "_A3"
            TARGET DOMAIN : G2(G2N1,G2N2) 5 rows
     3  -  JOINED COLUMN : SUPPLIER.S_SUPPKEY, NATION.N_NAME
     4  -  CLONED
            READ COLUMN : NATION.N_NATIONKEY, NATION.N_NAME
     5  -  HASH KEY : SUPPLIER.S_NATIONKEY
            RECORD COLUMN : SUPPLIER.S_SUPPKEY
            READ KEY COLUMN : SUPPLIER.S_NATIONKEY, SUPPLIER.S_SUPPKEY
              HASH FILTER : SUPPLIER.S_NATIONKEY = NATION.N_NATIONKEY
```

```
    6  -  CLONED
          READ COLUMN : SUPPLIER.S_SUPPKEY, SUPPLIER.S_NATIONKEY


<<<  end print plan
```

通过在多个group收集数据的outer join可在以下三种情况执行

- Case 1: Joining sharded table and cloned table
  - 在分配sharded table的数据的所有group分配cloned table的数据的情况
- Case 2: Joining sharded table and sharded table
  - 两个sharded table的sharding key之间有equi-join条件的情况
- Case 3: 包含一个以上的sharded table并sharding key之间没有equi-join条件的情况

Case 1的情况构成包含outer join的generated query

如下汇集在各个group处理outer join的结果为整体join结果

```
gSQL> \EXPLAIN PLAN
      SELECT p_name, ps_suppkey
        FROM part
             LEFT OUTER JOIN
             partsupp
             ON p_partkey = ps_partkey;


P_NAME PS_SUPPKEY
------ ----------
Part#3          4
```

```
Part#3          1

Part#1          3

Part#1          2

Part#4          5

Part#4          3

Part#2          5

Part#2          2

Part#5          4

Part#5          1


10 rows selected.


>>>  start print plan


< Execution Plan >

================================================================================

|IDX| NODE DESCRIPTION                                |             ROWS |

--------------------------------------------------------------------------------

| 0 | SELECT STATEMENT                                |               10 |

| 1 |  QUERY BLOCK ("$QB_IDX_2")                      |               10 |

| 2 |   SINGLE CLUSTER                                | LOCAL/REMOTE 10 |

| 3 |    SELECT STATEMENT                             |                2 |

| 4 |     QUERY BLOCK ("$QB_IDX_2")                   |                2 |

| 5 |      HASH JOIN (LEFT OUTER JOIN)                |                2 |

| 6 |       TABLE ACCESS ("PART" AS _A2)              |                1 |
```

```
| 7 |       HASH JOIN INSTANT                          |              2 |
| 8 |        INDEX ACCESS ("PARTSUPP" AS _A1, ... )    | (     2)     2 |
=========================================================================


     1  -  TARGET : PART.P_NAME, PARTSUPP.PS_SUPPKEY
     2  -  SQL : SELECT /*+ KEEP_JOINED_TABLE USE_HASH_IN( _A1, 10 )
FULL( _A2 ) INDEX( _A1, "PUBLIC"."PARTSUPP_PK_INDEX" ) */ "_A2"."P_NAME",
"_A1"."PS_SUPPKEY" FROM
( "PUBLIC"."PART"@"G1N1"|"G1N2"|"G2N1"|"G2N2"|"G3N1"|"G3N2" AS "_A2" LEFT
OUTER JOIN "PUBLIC"."PARTSUPP"@"G1N1"|"G1N2"|"G2N1"|"G2N2"|"G3N1"|"G3N2" AS
"_A1" ON "_A1"."PS_PARTKEY" = "_A2"."P_PARTKEY") ALIAS "_A3"
           TARGET DOMAIN : G1(G1N1,G1N2) 2 rows, G2(G2N1,G2N2) 4 rows,
G3(G3N1,G3N2) 4 rows
     4  -  TARGET : _A2.P_NAME, _A1.PS_SUPPKEY
     5  -  JOINED COLUMN : _A2.P_NAME, _A1.PS_SUPPKEY
     6  -  HASH SHARD ( # 3 )
           READ COLUMN : _A2.P_PARTKEY, _A2.P_NAME
     7  -  HASH KEY : _A1.PS_PARTKEY
           RECORD COLUMN : _A1.PS_SUPPKEY
           READ KEY COLUMN : _A1.PS_PARTKEY, _A1.PS_SUPPKEY
             HASH FILTER : _A1.PS_PARTKEY = _A2.P_PARTKEY
     8  -  HASH SHARD ( # 3 )
           READ INDEX COLUMN : _A1.PS_PARTKEY, _A1.PS_SUPPKEY


<<<  end print plan
```

Case 2的情况与case 1相同构成包含outer join的generated query

如下汇集在各个group处理outer join的结果为整体join结果

```
gSQL> \EXPLAIN PLAN
      SELECT p_name, ps_suppkey
        FROM part
             LEFT OUTER JOIN
             partsupp
             ON p_partkey = ps_partkey;


P_NAME PS_SUPPKEY
------ ----------
Part#3          4
Part#3          1
Part#1          3
Part#1          2
Part#4          5
Part#4          3
Part#2          5
Part#2          2
Part#5          4
Part#5          1


10 rows selected.
```

```
>>>  start print plan


< Execution Plan >

================================================================================

|IDX| NODE DESCRIPTION                                |            ROWS |

--------------------------------------------------------------------------------

| 0 | SELECT STATEMENT                                |              10 |

| 1 |  QUERY BLOCK ("$QB_IDX_2")                      |              10 |

| 2 |   SINGLE CLUSTER                                | LOCAL/REMOTE 10 |

| 3 |    SELECT STATEMENT                             |               2 |

| 4 |     QUERY BLOCK ("$QB_IDX_2")                   |               2 |

| 5 |      HASH JOIN (LEFT OUTER JOIN)                |               2 |

| 6 |       TABLE ACCESS ("PART" AS _A2)              |               1 |

| 7 |       HASH JOIN INSTANT                         |               2 |

| 8 |        INDEX ACCESS ("PARTSUPP" AS _A1, ... )   | (     2)      2 |

================================================================================


     1  -  TARGET : PART.P_NAME, PARTSUPP.PS_SUPPKEY

     2  -  SQL : SELECT /*+ KEEP_JOINED_TABLE USE_HASH_IN( _A1, 10 )
FULL( _A2 ) INDEX( _A1, "PUBLIC"."PARTSUPP_PK_INDEX" ) */ "_A2"."P_NAME",
"_A1"."PS_SUPPKEY" FROM
( "PUBLIC"."PART"@"G1N1"|"G1N2"|"G2N1"|"G2N2"|"G3N1"|"G3N2" AS "_A2" LEFT
OUTER JOIN "PUBLIC"."PARTSUPP"@"G1N1"|"G1N2"|"G2N1"|"G2N2"|"G3N1"|"G3N2" AS
"_A1" ON "_A1"."PS_PARTKEY" = "_A2"."P_PARTKEY") ALIAS "_A3"
           TARGET DOMAIN : G1(G1N1,G1N2) 2 rows, G2(G2N1,G2N2) 4 rows,
```

```
G3(G3N1,G3N2) 4 rows

     4  -  TARGET : _A2.P_NAME, _A1.PS_SUPPKEY

     5  -  JOINED COLUMN : _A2.P_NAME, _A1.PS_SUPPKEY

     6  -  HASH SHARD ( # 3 )

           READ COLUMN : _A2.P_PARTKEY, _A2.P_NAME

     7  -  HASH KEY : _A1.PS_PARTKEY

           RECORD COLUMN : _A1.PS_SUPPKEY

           READ KEY COLUMN : _A1.PS_PARTKEY, _A1.PS_SUPPKEY

             HASH FILTER : _A1.PS_PARTKEY = _A2.P_PARTKEY

     8  -  HASH SHARD ( # 3 )

           READ INDEX COLUMN : _A1.PS_PARTKEY, _A1.PS_SUPPKEY


<<<  end print plan
```

Case 3的情况 与case 1相同在各个group汇集包含outer join的generated query执行结果时会获取重复的anti join结果为了删除重复的anti join结果使用intersect key group方式的数据操作

Intersect key group的详细内容参考**为了Intersect Key Group的Generated Query**

以下为处理用于outer join的intersect key group的示例

```
gSQL> \EXPLAIN PLAN

      SELECT /*+ REMOTE_JOIN( supplier ) */ ps_partkey, s_name

        FROM supplier

             LEFT OUTER JOIN

             partsupp

             ON ps_suppkey = s_suppkey;
```

```
PS_PARTKEY S_NAME
---------- -------------------------
         3 Supplier#1
         5 Supplier#1
         2 Supplier#2
         1 Supplier#2
         1 Supplier#3
         4 Supplier#3
         3 Supplier#4
         5 Supplier#4
         2 Supplier#5
         4 Supplier#5

10 rows selected.

>>>  start print plan

< Execution Plan >
================================================================================
=
|IDX| NODE DESCRIPTION                                |            ROWS
|
--------------------------------------------------------------------------------
-
```

```
| 0 |SELECT STATEMENT                                |              10
|
| 1 |  QUERY BLOCK ("$QB_IDX_2")                     |              10
|
| 2 |    MULTIPLE CLUSTER                             | LOCAL/REMOTE 10
|
| 3 |      SELECT STATEMENT                          |               5
|
| 4 |        QUERY BLOCK ("$QB_IDX_2")               |               5
|
| 5 |          SORT INSTANT                          |               5
|
| 6 |            HASH JOIN (INVERTED LEFT OUTER JOIN)|               5
|
| 7 |              INDEX ACCESS ("PARTSUPP" AS _A2, ... | (     2)      2
|
| 8 |              HASH JOIN INSTANT                 |               5
|
| 9 |                TABLE ACCESS ("SUPPLIER" AS _A1)|               5
|
================================================================================
=

     1  -  TARGET : PARTSUPP.PS_PARTKEY, SUPPLIER.S_NAME
     2  -  SQL : SELECT /*+ KEEP_JOINED_TABLE USE_HASH_IN( _A1, 5 )
```

```
INDEX( _A2, "PUBLIC"."PARTSUPP_PK_INDEX" ) FULL( _A1 ) */
"_A1"."S_SUPPKEY", "_A2"."PS_SUPPKEY", "_A2"."PS_PARTKEY", "_A1"."S_NAME"
FROM ( "PUBLIC"."SUPPLIER"@"G1N1"|"G1N2"|"G2N1"|"G2N2"|"G3N1"|"G3N2" AS "_A1"
LEFT OUTER JOIN
"PUBLIC"."PARTSUPP"@"G1N1"|"G1N2"|"G2N1"|"G2N2"|"G3N1"|"G3N2" AS "_A2" ON
"_A1"."S_SUPPKEY" = "_A2"."PS_SUPPKEY") ALIAS "_A3" ORDER BY "_A1"."S_SUPPKEY"
ASC NULLS LAST
            TARGET DOMAIN : G1(G1N1,G1N2) 5 rows, G2(G2N1,G2N2) 5 rows,
G3(G3N1,G3N2) 6 rows
            INTERSECT KEY GROUP
              KEY GROUP : SUPPLIER.S_SUPPKEY
              Nil Expression : PARTSUPP.PS_SUPPKEY
     4  -  TARGET : _A1.S_SUPPKEY, _A2.PS_SUPPKEY, _A2.PS_PARTKEY,
_A1.S_NAME
     5  -  SORT KEY : "_A1.S_SUPPKEY ASC NULLS LAST"
            RECORD COLUMN : _A2.PS_SUPPKEY, _A2.PS_PARTKEY, _A1.S_NAME
            READ KEY COLUMN : _A1.S_SUPPKEY
            READ RECORD COLUMN : _A2.PS_SUPPKEY, _A2.PS_PARTKEY, _A1.S_NAME
     6  -  JOINED COLUMN : _A1.S_SUPPKEY, _A2.PS_SUPPKEY, _A2.PS_PARTKEY,
_A1.S_NAME
     7  -  HASH SHARD ( # 3 )
            READ INDEX COLUMN : _A2.PS_PARTKEY, _A2.PS_SUPPKEY
     8  -  HASH KEY : _A1.S_SUPPKEY
            RECORD COLUMN : _A1.S_NAME
            READ KEY COLUMN : _A1.S_SUPPKEY, _A1.S_NAME
```

```
            HASH FILTER : _A1.S_SUPPKEY = _A2.PS_SUPPKEY
     9   -  CLONED
            READ COLUMN : _A1.S_SUPPKEY, _A1.S_NAME


<<<  end print plan
```

如上generated query包含包含在equi-join的column的ordering按照ordering顺序从各个group汇集收集的数据仅限于nil expression值为null的情况适用intersect key group

FROM语句（包含子查询的Join）

包含子查询（subquery）的join根据是否将子查询作为join条件进行区分处理未将子查询作为join条件的join的集群时通过cluster puller收集数据并操作后应用与子查询相关的filter将子查询作为join条件的join按照join运算区分处理

以下为处理未用作join条件的子查询的示例

```
gSQL> \EXPLAIN PLAN
      SELECT p_name, ps_suppkey
        FROM part, partsupp
       WHERE p_partkey = ps_partkey
             AND ps_suppkey IN ( SELECT /*+ NO_UNNEST */ s_suppkey FROM
supplier );


P_NAME PS_SUPPKEY
------ ----------
Part#3          4
```

```
Part#3          1

Part#2          5

Part#2          2

Part#5          4

Part#5          1

Part#1          3

Part#1          2

Part#4          5

Part#4          3


10 rows selected.


>>>  start print plan


< Execution Plan >

================================================================================

|  IDX  |  NODE DESCRIPTION                          |             ROWS |

--------------------------------------------------------------------------------

|    0  |  SELECT STATEMENT                          |               10 |

|    1  |    QUERY BLOCK ("$QB_IDX_2")               |               10 |

|    2  |      PLAN BASED CLUSTER                    | LOCAL/REMOTE  10 |

|    3  |        HASH JOIN (INNER JOIN)              |                2 |

|    4  |          TABLE ACCESS ("PART")             |                1 |

|    5  |          HASH JOIN INSTANT                 |                2 |

|    6  |            INDEX ACCESS ("PARTSUPP", ...)  | (      2)      2 |
```

```
|    7  |  SUB QUERY LIST                              |                 |
|    8  |    INLINE_VIEW ("$V8") (MATERIALIZED)        |              10 |
|    9  |      QUERY BLOCK ("$QB_IDX_8")               |               5 |
|   10  |        INDEX ACCESS ("SUPPLIER", ...)        | (      5)       5 |
=============================================================================


     1  -  TARGET : PART.P_NAME, PARTSUPP.PS_SUPPKEY
     2  -  SQL : SELECT /*+ KEEP_JOINED_TABLE USE_HASH_IN( _A1, 10 )
FULL( _A2 ) INDEX( _A1, "PUBLIC"."PARTSUPP_PK_INDEX" ) */
"_A1"."PS_SUPPKEY", "_A2"."P_NAME" FROM ( "PUBLIC"."PART"@LOCAL AS "_A2"
INNER JOIN "PUBLIC"."PARTSUPP"@LOCAL AS "_A1" ON "_A1"."PS_PARTKEY" =
"_A2"."P_PARTKEY") ALIAS "_A3"
           TARGET DOMAIN : G1(G1N1,G1N2) 2 rows, G2(G2N1,G2N2) 4 rows,
G3(G3N1,G3N2) 4 rows
             POST FILTER : ( PARTSUPP.PS_SUPPKEY ) IN ( $V8.S_SUPPKEY )
     3  -  JOINED COLUMN : PARTSUPP.PS_SUPPKEY, PART.P_NAME
     4  -  HASH SHARD ( # 3 )
           READ COLUMN : PART.P_PARTKEY, PART.P_NAME
     5  -  HASH KEY : PARTSUPP.PS_PARTKEY
           RECORD COLUMN : PARTSUPP.PS_SUPPKEY
           READ KEY COLUMN : PARTSUPP.PS_PARTKEY, PARTSUPP.PS_SUPPKEY
             HASH FILTER : PARTSUPP.PS_PARTKEY = PART.P_PARTKEY
     6  -  HASH SHARD ( # 3 )
           READ INDEX COLUMN : PARTSUPP.PS_PARTKEY, PARTSUPP.PS_SUPPKEY
     8  -  COLUMN : SUPPLIER.S_SUPPKEY AS S_SUPPKEY
```

```
     9  -  TARGET : SUPPLIER.S_SUPPKEY
    10  -  CLONED
           READ INDEX COLUMN : SUPPLIER.S_SUPPKEY


<<<  end print plan
```

将子查询用作join条件的join按照包含子查询的运算符分类如下

Join运算	包含子查询的运算符
INNER JOIN	除**< Group Comparison Conditions >**外的所有运算符
OUTER JOIN	除**< Group Comparison Conditions >**外的所有运算符
SEMI JOIN	拥有EXISTS, IN, ANY quantifier的**< Group Comparison Conditions >**
ANTI-SEMI JOIN	拥有NOT EXISTS, NOT IN, ALL quantifier的**< Group Comparison Conditions >**

Table 2-11 包含子查询的join

Join运算相关详细内容参考**Join**

在inner join使用包含子查询的join条件或filter时如下收集数据后应用与子查询相关的filter

```
gSQL> \EXPLAIN PLAN
      SELECT ps_partkey, s_name
        FROM supplier
             INNER JOIN
             partsupp
```

```
            ON ps_suppkey = s_suppkey
               AND ps_suppkey = ( SELECT s_suppkey FROM DUAL );


PS_PARTKEY S_NAME
---------- -------------------------
         3 Supplier#1
         3 Supplier#4
         1 Supplier#2
         1 Supplier#3
         4 Supplier#3
         4 Supplier#5
         2 Supplier#2
         2 Supplier#5
         5 Supplier#1
         5 Supplier#4

10 rows selected.

>>>  start print plan

< Execution Plan >
=========================================================================
|  IDX  |  NODE DESCRIPTION                        |             ROWS |
-------------------------------------------------------------------------
|    0  |  SELECT STATEMENT                        |               10 |
```

```
|    1  |    QUERY BLOCK ("$QB_IDX_2")             |              10 |
|    2  |      PLAN BASED CLUSTER                   | LOCAL/REMOTE  10 |
|    3  |        HASH JOIN (INNER JOIN)             |               2 |
|    4  |          INDEX ACCESS ("PARTSUPP", ...)   | (      2)     2 |
|    5  |          HASH JOIN INSTANT                |               2 |
|    6  |            TABLE ACCESS ("SUPPLIER")      |               5 |
|    7  |      SUB QUERY LIST                       |                 |
|    8  |        INLINE_VIEW ("$V8")                |              10 |
|    9  |          QUERY BLOCK ("$QB_IDX_8")        |              10 |
|   10  |            FAST DUAL ACCESS ("DUAL")      |              10 |
=============================================================================


     1  -  TARGET : PARTSUPP.PS_PARTKEY, SUPPLIER.S_NAME
     2  -  SQL : SELECT /*+ KEEP_JOINED_TABLE USE_HASH_IN( _A1, 5 )
INDEX( _A2, "PUBLIC"."PARTSUPP_PK_INDEX" ) FULL( _A1 ) */
"_A2"."PS_SUPPKEY", "_A1"."S_SUPPKEY", "_A2"."PS_PARTKEY", "_A1"."S_NAME"
FROM ( "PUBLIC"."PARTSUPP"@LOCAL AS "_A2" INNER JOIN
"PUBLIC"."SUPPLIER"@LOCAL AS "_A1" ON "_A1"."S_SUPPKEY" = "_A2"."PS_SUPPKEY")
ALIAS "_A3"
           TARGET DOMAIN : G1(G1N1,G1N2) 2 rows, G2(G2N1,G2N2) 4 rows,
G3(G3N1,G3N2) 4 rows
              POST FILTER : PARTSUPP.PS_SUPPKEY = $V8.S_SUPPKEY
     3  -  JOINED COLUMN : PARTSUPP.PS_SUPPKEY, SUPPLIER.S_SUPPKEY,
PARTSUPP.PS_PARTKEY, SUPPLIER.S_NAME
     4  -  HASH SHARD ( # 3 )
```

```
           READ INDEX COLUMN : PARTSUPP.PS_PARTKEY, PARTSUPP.PS_SUPPKEY
     5   -  HASH KEY : SUPPLIER.S_SUPPKEY
           RECORD COLUMN : SUPPLIER.S_NAME
           READ KEY COLUMN : SUPPLIER.S_SUPPKEY, SUPPLIER.S_NAME
             HASH FILTER : SUPPLIER.S_SUPPKEY = PARTSUPP.PS_SUPPKEY
           FETCH ONE ROW
     6   -  CLONED
           READ COLUMN : SUPPLIER.S_SUPPKEY, SUPPLIER.S_NAME
     8   -  COLUMN : {SUPPLIER.S_SUPPKEY} AS S_SUPPKEY
     9   -  TARGET : {SUPPLIER.S_SUPPKEY}
    10   -  READ COLUMN : NOTHING


<<<  end print plan
```

Outer join中有包含子查询的join条件时无法构成包含outer join条件的generated query

此时如下对各个join对象表构成cluster puller后执行join

```
gSQL> \EXPLAIN PLAN
      SELECT ps_partkey, s_name
        FROM supplier
             LEFT OUTER JOIN
             partsupp
             ON ps_suppkey = ( SELECT s_suppkey FROM DUAL );


PS_PARTKEY S_NAME
---------- -------------------------
```

```
         3 Supplier#1

         5 Supplier#1

         2 Supplier#2

         1 Supplier#2

         1 Supplier#3

         4 Supplier#3

         3 Supplier#4

         5 Supplier#4

         4 Supplier#5

         2 Supplier#5


10 rows selected.


>>>  start print plan


< Execution Plan >

=============================================================================
|  IDX  |  NODE DESCRIPTION                          |              ROWS |
-----------------------------------------------------------------------------
|    0  |  SELECT STATEMENT                          |                10 |
|    1  |    QUERY BLOCK ("$QB_IDX_2")               |                10 |
|    2  |      NESTED JOIN (LEFT OUTER JOIN)         |                10 |
|    3  |        TABLE ACCESS ("SUPPLIER")           |                 5 |
|    4  |        PLAN BASED CLUSTER                  | LOCAL/REMOTE   50 |
|    5  |          INDEX ACCESS ("PARTSUPP", ...)    | (        10)   10 |
```

```
|    6  |      SUB QUERY LIST                        |                  |
|    7  |        INLINE_VIEW ("$V7")                 |               50 |
|    8  |          QUERY BLOCK ("$QB_IDX_8")         |               50 |
|    9  |            FAST DUAL ACCESS ("DUAL")       |               50 |
=============================================================================

     1  -  TARGET : PARTSUPP.PS_PARTKEY, SUPPLIER.S_NAME
     2  -  JOINED COLUMN : PARTSUPP.PS_SUPPKEY, SUPPLIER.S_SUPPKEY,
PARTSUPP.PS_PARTKEY, SUPPLIER.S_NAME
                POST ON FILTER : PARTSUPP.PS_SUPPKEY = $V7.S_SUPPKEY
     3  -  CLONED
           READ COLUMN : SUPPLIER.S_SUPPKEY, SUPPLIER.S_NAME
     4  -  SQL : SELECT /*+ INDEX( _A1, "PUBLIC"."PARTSUPP_PK_INDEX" ) */
"_A1"."PS_PARTKEY", "_A1"."PS_SUPPKEY" FROM "PUBLIC"."PARTSUPP"@LOCAL AS
"_A1"
           TARGET DOMAIN : G1(G1N1,G1N2) 10 rows, G2(G2N1,G2N2) 20 rows,
G3(G3N1,G3N2) 20 rows
     5  -  HASH SHARD ( # 3 )
           READ INDEX COLUMN : PARTSUPP.PS_PARTKEY, PARTSUPP.PS_SUPPKEY
     7  -  COLUMN : {SUPPLIER.S_SUPPKEY} AS S_SUPPKEY
     8  -  TARGET : {SUPPLIER.S_SUPPKEY}
     9  -  READ COLUMN : NOTHING

<<<  end print plan
```

在outer join中包含子查询的filter不包含在generated queryouter join的cluster puller收集并操作数据后执行不包含在generated query中的filter

以下为在outer join处理包含子查询的filter的示例

```
gSQL> \EXPLAIN PLAN
      SELECT p_name, ps_suppkey
        FROM partsupp
             LEFT OUTER JOIN
             part
             ON p_partkey = ps_partkey
       WHERE ps_supplycost > ( SELECT p_retailprice FROM supplier WHERE
s_suppkey = ps_suppkey );

P_NAME PS_SUPPKEY
------ ----------
Part#3          4
Part#1          2
Part#5          1

3 rows selected.

>>>  start print plan

< Execution Plan >
=============================================================================
```

```
=
|  IDX  |  NODE DESCRIPTION                              |             ROWS
|
-------------------------------------------------------------------------------
-
|    0  |  SELECT STATEMENT                              |                3
|
|    1  |    QUERY BLOCK ("$QB_IDX_2")                   |                3
|
|    2  |      SINGLE CLUSTER                            | LOCAL/REMOTE   3
|
|    3  |        SELECT STATEMENT                        |                2
|
|    4  |          QUERY BLOCK ("$QB_IDX_2")             |                2
|
|    5  |            HASH JOIN (LEFT OUTER JOIN)         |                2
|
|    6  |              TABLE ACCESS ("PARTSUPP" AS _A2)  |                2
|
|    7  |              HASH JOIN INSTANT                 |                2
|
|    8  |                TABLE ACCESS ("PART" AS _A1)    |                1
|
|    9  |      SUB QUERY LIST                            |
|
```

```
|   10  |        INLINE_VIEW ("$V8")                    |              10
|
|   11  |          QUERY BLOCK ("$QB_IDX_8")            |              10
|
|   12  |            INDEX ACCESS ("SUPPLIER", ...)     | (     10)    10
|
================================================================================
=


     1  -  TARGET : PART.P_NAME, PARTSUPP.PS_SUPPKEY
     2  -  SQL : SELECT /*+ KEEP_JOINED_TABLE USE_HASH_IN( _A1, 500 )
FULL( _A2 ) FULL( _A1 ) */ "_A2"."PS_SUPPLYCOST", "_A2"."PS_SUPPKEY",
"_A1"."P_RETAILPRICE", "_A1"."P_NAME" FROM
( "PUBLIC"."PARTSUPP"@"G1N1"|"G1N2"|"G2N1"|"G2N2"|"G3N1"|"G3N2" AS "_A2" LEFT
OUTER JOIN "PUBLIC"."PART"@"G1N1"|"G1N2"|"G2N1"|"G2N2"|"G3N1"|"G3N2" AS "_A1"
ON "_A1"."P_PARTKEY" = "_A2"."PS_PARTKEY") ALIAS "_A3"
           TARGET DOMAIN : G1(G1N1,G1N2) 2 rows, G2(G2N1,G2N2) 4 rows,
G3(G3N1,G3N2) 4 rows
              POST FILTER : PARTSUPP.PS_SUPPLYCOST > $V8.P_RETAILPRICE
     4  -  TARGET : _A2.PS_SUPPLYCOST, _A2.PS_SUPPKEY, _A1.P_RETAILPRICE,
_A1.P_NAME
     5  -  JOINED COLUMN : _A2.PS_SUPPLYCOST, _A2.PS_SUPPKEY,
_A1.P_RETAILPRICE, _A1.P_NAME
     6  -  HASH SHARD ( # 3 )
           READ COLUMN : _A2.PS_PARTKEY, _A2.PS_SUPPKEY, _A2.PS_SUPPLYCOST
```

```
     7  -  HASH KEY : _A1.P_PARTKEY
           RECORD COLUMN : _A1.P_RETAILPRICE, _A1.P_NAME
           READ KEY COLUMN : _A1.P_PARTKEY, _A1.P_RETAILPRICE, _A1.P_NAME
             HASH FILTER : _A1.P_PARTKEY = _A2.PS_PARTKEY
           FETCH ONE ROW
     8  -  HASH SHARD ( # 3 )
           READ COLUMN : _A1.P_PARTKEY, _A1.P_NAME, _A1.P_RETAILPRICE
    10  -  COLUMN : {PART.P_RETAILPRICE} AS P_RETAILPRICE
    11  -  TARGET : {PART.P_RETAILPRICE}
    12  -  CLONED
           READ INDEX COLUMN : SUPPLIER.S_SUPPKEY
             MIN RANGE : SUPPLIER.S_SUPPKEY = {PARTSUPP.PS_SUPPKEY}
             MAX RANGE : SUPPLIER.S_SUPPKEY = {PARTSUPP.PS_SUPPKEY}
           FETCH ONE ROW


<<<  end print plan
```

对在where语句中描述的子查询进行<subquery unnest>后变更为semi join时用于处理semi join的generated query包含semi join语句汇集从多个group执行generated query的结果会获取重复的semi join结果通过distinct key group方式操作数据并删除重复的semi join结果Distinct key group相关详细说明参考用于**用于Distinct Key Group的Generated Query**

以下为处理用于semi join的distinct key group的示例

```
gSQL> \EXPLAIN PLAN
      SELECT s_name
```

```
        FROM supplier
       WHERE s_suppkey IN ( SELECT /*+ REMOTE_UNNEST */ ps_suppkey FROM
partsupp );


S_NAME
-------------------------
Supplier#1
Supplier#2
Supplier#3
Supplier#4
Supplier#5


5 rows selected.


>>>  start print plan


< Execution Plan >
================================================================================
=
|IDX|  NODE DESCRIPTION                                |           ROWS
|
--------------------------------------------------------------------------------
-
| 0 |SELECT STATEMENT                                  |              5
|
```

```
| 1 |  QUERY BLOCK ("$QB_IDX_2")                          |              5
|
| 2 |    MULTIPLE CLUSTER                                 | LOCAL/REMOTE 5
|
| 3 |      SELECT STATEMENT                               |              2
|
| 4 |        QUERY BLOCK ("$QB_IDX_2")                    |              2
|
| 5 |          SORT INSTANT                               |              2
|
| 6 |            HASH JOIN (INVERTED SEMI)                |              2
|
| 7 |              INDEX ACCESS ("PARTSUPP" AS _A2, ...) | (    2)      2
|
| 8 |              HASH JOIN INSTANT                      |              2
|
| 9 |                TABLE ACCESS ("SUPPLIER" AS _A1)     |              5
|
=========================================================================
=


     1  -  TARGET : SUPPLIER.S_NAME
     2  -  SQL : SELECT /*+ KEEP_JOINED_TABLE USE_HASH_IN( _A1, 5 )
INDEX( _A2, "PUBLIC"."PARTSUPP_PK_INDEX" ) FULL( _A1 ) */
"_A1"."S_SUPPKEY", "_A1"."S_NAME" FROM
```

```
( "PUBLIC"."SUPPLIER"@"G1N1"|"G1N2"|"G2N1"|"G2N2"|"G3N1"|"G3N2" AS "_A1" SEMI
JOIN "PUBLIC"."PARTSUPP"@"G1N1"|"G1N2"|"G2N1"|"G2N2"|"G3N1"|"G3N2" AS "_A2"
ON "_A1"."S_SUPPKEY" = "_A2"."PS_SUPPKEY") ALIAS "_A3" ORDER BY "_A1"."S_SUPPKEY"
ASC NULLS LAST
           TARGET DOMAIN : G1(G1N1,G1N2) 2 rows, G2(G2N1,G2N2) 4 rows,
G3(G3N1,G3N2) 3 rows
           DISTINCT KEY GROUP
             KEY GROUP : SUPPLIER.S_SUPPKEY
     4   -  TARGET : _A1.S_SUPPKEY, _A1.S_NAME
     5   -  SORT KEY : "_A1.S_SUPPKEY ASC NULLS LAST"
           RECORD COLUMN : _A1.S_NAME
           READ KEY COLUMN : _A1.S_SUPPKEY
           READ RECORD COLUMN : _A1.S_NAME
     6   -  JOINED COLUMN : _A1.S_SUPPKEY, _A1.S_NAME
     7   -  HASH SHARD ( # 3 )
           READ INDEX COLUMN : _A2.PS_SUPPKEY
     8   -  HASH KEY : _A1.S_SUPPKEY
           RECORD COLUMN : _A1.S_NAME
           READ KEY COLUMN : _A1.S_SUPPKEY, _A1.S_NAME
             HASH FILTER : _A1.S_SUPPKEY = _A2.PS_SUPPKEY
     9   -  CLONED
           READ COLUMN : _A1.S_SUPPKEY, _A1.S_NAME


<<<  end print plan
```

对在where语句中描述的子查询进行<subquery unnest>变更为anti-semi join时用于处理anti-semi join的generated query包含anti-semi join语句汇集从多个group执行generated query的结果会获取重复的anti-semi join结果通过intersect key group方式操作数据并删除重复的anti-semi join结果Intersect key group相关详细说明参考**为了Intersect Key Group的Generated Query**

以下为处理用于anti-semi join的intersect key group的示例

```
gSQL> \EXPLAIN PLAN
      SELECT s_name
        FROM supplier
       WHERE s_suppkey NOT IN ( SELECT /*+ REMOTE_UNNEST */ ps_suppkey
FROM partsupp WHERE ps_supplycost > 900 );

S_NAME
-------------------------
Supplier#3
Supplier#5

2 rows selected.

>>>  start print plan

< Execution Plan >
================================================================================
|IDX|  NODE DESCRIPTION                                  |           ROWS |
--------------------------------------------------------------------------------
```

```
| 0 |  SELECT STATEMENT                                  |              2 |
| 1 |    QUERY BLOCK ("$QB_IDX_2")                       |              2 |
| 2 |      MULTIPLE CLUSTER                              | LOCAL/REMOTE 2 |
| 3 |        SELECT STATEMENT                            |              4 |
| 4 |          QUERY BLOCK ("$QB_IDX_2")                 |              4 |
| 5 |            SORT INSTANT                            |              4 |
| 6 |              HASH JOIN (ANTI SEMI)                 |              4 |
| 7 |                TABLE ACCESS ("SUPPLIER" AS _A2)    |              5 |
| 8 |                HASH JOIN INSTANT (UNIQUE)          |              4 |
| 9 |                  TABLE ACCESS ("PARTSUPP" AS _A1)  |              1 |
================================================================================


     1  -  TARGET : SUPPLIER.S_NAME
     2  -  SQL : SELECT /*+ KEEP_JOINED_TABLE USE_HASH_IN( _A1, 10 )
FULL( _A2 ) FULL( _A1 ) */ "_A2"."S_SUPPKEY", "_A2"."S_NAME" FROM
( "PUBLIC"."SUPPLIER"@"G1N1"|"G1N2"|"G2N1"|"G2N2"|"G3N1"|"G3N2" AS "_A2" ANTI
SEMI JOIN "PUBLIC"."PARTSUPP"@"G1N1"|"G1N2"|"G2N1"|"G2N2"|"G3N1"|"G3N2" AS
"_A1" ON "_A1"."PS_SUPPKEY" = "_A2"."S_SUPPKEY" AND "_A1"."PS_SUPPLYCOST" > :_V0)
ALIAS "_A3" ORDER BY "_A2"."S_SUPPKEY" ASC NULLS LAST
            TARGET DOMAIN : G1(G1N1,G1N2) 4 rows, G2(G2N1,G2N2) 4 rows,
G3(G3N1,G3N2) 4 rows
            INTERSECT KEY GROUP
              KEY GROUP : SUPPLIER.S_SUPPKEY
     4  -  TARGET : _A2.S_SUPPKEY, _A2.S_NAME
     5  -  SORT KEY : "_A2.S_SUPPKEY ASC NULLS LAST"
```

```
            RECORD COLUMN : _A2.S_NAME
            READ KEY COLUMN : _A2.S_SUPPKEY
            READ RECORD COLUMN : _A2.S_NAME
      6   -  JOINED COLUMN : _A2.S_SUPPKEY, _A2.S_NAME
      7   -  CLONED
            READ COLUMN : _A2.S_SUPPKEY, _A2.S_NAME
      8   -  HASH KEY : _A1.PS_SUPPKEY
            READ KEY COLUMN : _A1.PS_SUPPKEY
              HASH FILTER : _A1.PS_SUPPKEY = _A2.S_SUPPKEY
            FETCH ONE ROW
      9   -  HASH SHARD ( # 3 )
            READ COLUMN : _A1.PS_SUPPKEY, _A1.PS_SUPPLYCOST
              PHYSICAL FILTER : _A1.PS_SUPPLYCOST > :_V0

<<<  end print plan
```

WHERE语句

在plan node构成的filter分为如下三种

- Constant filter：以plan node为单位常数化后处理的filter
- Post filter：包含子查询或由non-deterministic的expression构成的filter
- Filter：未分类为Constant filter或post filter的其他filter

Generated query由cluster puller及下级节点拥有的filter构成查询Constant filter常数化后以bind parameter形式构成查询filter构成查询不需要变更但对post filter不构成generated query

以下为包含constant filter的generated query的示例

```
gSQL> \VAR v1 INTEGER

gSQL> \EXEC :v1 := 1

gSQL> \EXPLAIN PLAN

      SELECT p_name, p_brand, p_type, cluster_group_id

        FROM part

       WHERE :v1 = 1;


P_NAME P_BRAND    P_TYPE CLUSTER_GROUP_ID

------ ---------- ------ ----------------

Part#3 Brand#2    STEEL                 1

Part#2 Brand#1    NICKEL                2

Part#5 Brand#3    STEEL                 2

Part#1 Brand#1    COPPER                3

Part#4 Brand#3    NICKEL                3


5 rows selected.


>>>  start print plan


< Execution Plan >

=========================================================================

|  IDX  |  NODE DESCRIPTION                     |               ROWS |

-------------------------------------------------------------------------
```

```
|    0  |  SELECT STATEMENT                    |                  5 |
|    1  |    QUERY BLOCK ("$QB_IDX_2")         |                  5 |
|    2  |      PLAN BASED CLUSTER              | LOCAL/REMOTE     5 |
|    3  |        TABLE ACCESS ("PART")         |                  1 |
=============================================================================


     1  -  TARGET : PART.P_NAME, PART.P_BRAND, PART.P_TYPE,
PART.CLUSTER_GROUP_ID
     2  -  SQL : SELECT /*+ FULL( _A1 ) */ "_A1"."CLUSTER_GROUP_ID",
"_A1"."P_NAME", "_A1"."P_BRAND", "_A1"."P_TYPE" FROM "PUBLIC"."PART"@LOCAL
AS "_A1" WHERE :_V0
           TARGET DOMAIN : G1(G1N1,G1N2) 1 rows, G2(G2N1,G2N2) 2 rows,
G3(G3N1,G3N2) 2 rows
             CONSTANT FILTER  : :V1 = 1
     3  -  HASH SHARD ( # 3 )
           READ COLUMN : PART.P_NAME, PART.P_BRAND, PART.P_TYPE
             CONSTANT FILTER : :V1 = 1


<<<  end print plan
```

以下为包含cluster puller下级的filter的generated query的示例

```
gSQL> \EXPLAIN PLAN
      SELECT p_name, p_brand, p_type, cluster_group_id
        FROM part
       WHERE p_partkey = 1;
```

```
P_NAME P_BRAND    P_TYPE CLUSTER_GROUP_ID
------ ---------- ------ ----------------
Part#1 Brand#1    COPPER                3


1 row selected.


>>>  start print plan


< Execution Plan >
================================================================================
=
|  IDX  |  NODE DESCRIPTION                              |           ROWS
|
--------------------------------------------------------------------------------
-
|    0  |  SELECT STATEMENT                              |              1
|
|    1  |    QUERY BLOCK ("$QB_IDX_2")                   |              1
|
|    2  |      PLAN BASED CLUSTER                        | REMOTE ONLY  1
|
|    3  |        INDEX ACCESS ("PART", "PART_PK_INDEX")  | (     0)     0
|
================================================================================
```

```
=


     1  -  TARGET : PART.P_NAME, PART.P_BRAND, PART.P_TYPE,
PART.CLUSTER_GROUP_ID
     2  -  SQL : SELECT /*+ INDEX( _A1, "PUBLIC"."PART_PK_INDEX" ) */
"_A1"."CLUSTER_GROUP_ID", "_A1"."P_NAME", "_A1"."P_BRAND", "_A1"."P_TYPE"
FROM "PUBLIC"."PART"@LOCAL AS "_A1" WHERE "_A1"."P_PARTKEY" = :_V0
            TARGET DOMAIN : G3(G3N1,G3N2) 1 rows
     3  -  HASH SHARD ( # 3 )
            READ INDEX COLUMN : PART.P_PARTKEY
            READ TABLE COLUMN : PART.P_NAME, PART.P_BRAND, PART.P_TYPE
              MIN RANGE : PART.P_PARTKEY = 1
              MAX RANGE : PART.P_PARTKEY = 1
            FETCH ONE ROW


<<<  end print plan
```

以下为cluster puller拥有post filter时的generated query的示例

```
gSQL> \EXPLAIN PLAN
      SELECT p_name, p_brand, p_type, cluster_group_id
        FROM part
       WHERE p_name = 'Part#5' AND p_partkey IN ( SELECT /*+ NO_UNNEST */
p_partkey FROM DUAL );


P_NAME P_BRAND    P_TYPE CLUSTER_GROUP_ID
```

```
------ ---------- ------ ----------------

Part#5 Brand#3    STEEL                 2


1 row selected.


>>>  start print plan


< Execution Plan >

=========================================================================

|  IDX  |  NODE DESCRIPTION                     |             ROWS |

-------------------------------------------------------------------------

|    0  |  SELECT STATEMENT                     |                1 |

|    1  |    QUERY BLOCK ("$QB_IDX_2")          |                1 |

|    2  |      PLAN BASED CLUSTER               | LOCAL/REMOTE   1 |

|    3  |        TABLE ACCESS ("PART")          |                0 |

|    4  |      SUB QUERY LIST                   |                  |

|    5  |        INLINE_VIEW ("$V5")            |                1 |

|    6  |          QUERY BLOCK ("$QB_IDX_6")    |                1 |

|    7  |            FAST DUAL ACCESS ("DUAL")  |                1 |

=========================================================================


     1  -  TARGET : PART.P_NAME, PART.P_BRAND, PART.P_TYPE,

PART.CLUSTER_GROUP_ID

     2  -  SQL : SELECT /*+ FULL( _A1 ) */ "_A1"."CLUSTER_GROUP_ID",

"_A1"."P_PARTKEY", "_A1"."P_NAME", "_A1"."P_BRAND", "_A1"."P_TYPE" FROM
```

```
"PUBLIC"."PART"@LOCAL AS "_A1" WHERE "_A1"."P_NAME" = :_V0
           TARGET DOMAIN : G1(G1N1,G1N2) 0 rows, G2(G2N1,G2N2) 1 rows,
G3(G3N1,G3N2) 0 rows
              POST FILTER : ( PART.P_PARTKEY ) IN ( $V5.P_PARTKEY )
     3  -  HASH SHARD ( # 3 )
           READ COLUMN : PART.P_PARTKEY, PART.P_NAME, PART.P_BRAND,
PART.P_TYPE
             PHYSICAL FILTER : PART.P_NAME = 'Part#5'
     5  -  COLUMN : {PART.P_PARTKEY} AS P_PARTKEY
     6  -  TARGET : {PART.P_PARTKEY}
     7  -  READ COLUMN : NOTHING

<<<  end print plan
```

使用ROWNUM

Generated query不能包含non-deterministic的语句因此不能包含rownum使用rownum时构成COUNT plan node最终cluster puller无法在COUNT plan上级中构成

以下为使用rownum时的generated query的示例

```
gSQL> \EXPLAIN PLAN
      SELECT rownum, p_name, p_brand, p_type, cluster_group_id
        FROM part;

ROWNUM P_NAME P_BRAND    P_TYPE CLUSTER_GROUP_ID
```

```
------ ------ ---------- ------ ----------------

     1 Part#3 Brand#2    STEEL                1

     2 Part#1 Brand#1    COPPER               3

     3 Part#4 Brand#3    NICKEL               3

     4 Part#2 Brand#1    NICKEL               2

     5 Part#5 Brand#3    STEEL                2


5 rows selected.


>>>  start print plan


< Execution Plan >

=================================================================

|  IDX  |  NODE DESCRIPTION                |               ROWS |

-----------------------------------------------------------------

|    0  |  SELECT STATEMENT                |                  5 |

|    1  |    QUERY BLOCK ("$QB_IDX_2")     |                  5 |

|    2  |      COUNT                       |                  5 |

|    3  |        PLAN BASED CLUSTER        | LOCAL/REMOTE     5 |

|    4  |          TABLE ACCESS ("PART")   |                  1 |

=================================================================


     1  -  TARGET : ROWNUM, PART.P_NAME, PART.P_BRAND, PART.P_TYPE,

PART.CLUSTER_GROUP_ID

     3  -  SQL : SELECT /*+ FULL( _A1 ) */ "_A1"."CLUSTER_GROUP_ID",
```

```
"_A1"."P_NAME", "_A1"."P_BRAND", "_A1"."P_TYPE" FROM "PUBLIC"."PART"@LOCAL
AS "_A1"
           TARGET DOMAIN : G1(G1N1,G1N2) 1 rows, G2(G2N1,G2N2) 2 rows,
G3(G3N1,G3N2) 2 rows
     4  -  HASH SHARD ( # 3 )
           READ COLUMN : PART.P_NAME, PART.P_BRAND, PART.P_TYPE


<<<  end print plan
```

以下为使用rownum filter时的generated query的示例

```
gSQL> \EXPLAIN PLAN
      SELECT p_name, p_brand, p_type, cluster_group_id
        FROM part
       WHERE rownum < 3;


P_NAME P_BRAND    P_TYPE CLUSTER_GROUP_ID
------ ---------- ------ ----------------
Part#3 Brand#2    STEEL                 1
Part#1 Brand#1    COPPER                3


2 rows selected.


>>>  start print plan


< Execution Plan >
```

```
=================================================================
|  IDX  |  NODE DESCRIPTION                |                ROWS |
-----------------------------------------------------------------
|    0  |  SELECT STATEMENT                |                   2 |
|    1  |    QUERY BLOCK ("$QB_IDX_2")     |                   2 |
|    2  |      COUNT                       |                   2 |
|    3  |        PLAN BASED CLUSTER        | LOCAL/REMOTE      3 |
|    4  |          TABLE ACCESS ("PART")   |                   1 |
=================================================================


     1  -  TARGET : PART.P_NAME, PART.P_BRAND, PART.P_TYPE,
PART.CLUSTER_GROUP_ID
     2  -  STOP KEY FILTER : ROWNUM < 3
     3  -  SQL : SELECT /*+ FULL( _A1 ) */ "_A1"."CLUSTER_GROUP_ID",
"_A1"."P_NAME", "_A1"."P_BRAND", "_A1"."P_TYPE" FROM "PUBLIC"."PART"@LOCAL
AS "_A1"
           TARGET DOMAIN : G1(G1N1,G1N2) 1 rows, G2(G2N1,G2N2) 0 rows,
G3(G3N1,G3N2) 2 rows
     4  -  HASH SHARD ( # 3 )
           READ COLUMN : PART.P_NAME, PART.P_BRAND, PART.P_TYPE


<<<  end print plan
```

GROUP BY语句

group by语句的查询处理根据可通过下级plan node使用的sharding策略和grouping构成不同

gruoping执行方法会有所不同

下级节点的sharding策略和grouping构成	数据收集方法	数据操作方法	having子句处理
下级节点的所有sharding key均包含在grouping key时	在所有group执行包含group by语句的generated query	no manipulation	generated query中包含having子句内容
下级节点为cloned时	只在一个group执行包含group by语句的generated query	no manipulation	generated query中包含having子句内容
无法使用下级节点的sharding策略时	在所有group执行不包含group by语句的generated query	grouping	操作数据后应用having子句

Table 2-12 根据下级节点的sharding策略的grouping执行方法

> Note:
> 除下级节点为colned的情况外having子句中包含non-deterministic信息时无法构成包含grouping的generated query最终cluster puller plan node部署在group by的下级

以下为下级节点的sharding key均包含在grouping key时进行grouping的示例

```
gSQL> \EXPLAIN PLAN
      SELECT p_partkey
```

```
        FROM part
       GROUP BY p_partkey
      HAVING SUM( p_size ) > 0;


P_PARTKEY
---------
        3
        1
        4
        2
        5

5 rows selected.

>>>  start print plan

< Execution Plan >
=========================================================================
|  IDX  |  NODE DESCRIPTION                    |                ROWS |
-------------------------------------------------------------------------
|    0  |  SELECT STATEMENT                    |                   5 |
|    1  |    QUERY BLOCK ("$QB_IDX_2")         |                   5 |
|    2  |      PLAN BASED CLUSTER              | LOCAL/REMOTE      5 |
|    3  |        GROUP HASH INSTANT            |                   1 |
|    4  |          TABLE ACCESS ("PART")       |                   1 |
```

```
=====================================================================


     1  -  TARGET : PART.P_PARTKEY

     2  -  SQL : SELECT /*+ USE_GROUP_HASH(500) FULL( _A1 ) */
"_A1"."P_PARTKEY" FROM "PUBLIC"."PART"@LOCAL AS "_A1" GROUP BY
"_A1"."P_PARTKEY" HAVING SUM( "_A1"."P_SIZE" ) > :_V0

           TARGET DOMAIN : G1(G1N1,G1N2) 1 rows, G2(G2N1,G2N2) 2 rows,
G3(G3N1,G3N2) 2 rows

     3  -  GROUP KEY : PART.P_PARTKEY

           RECORD COLUMN : SUM( PART.P_SIZE )

           READ KEY COLUMN : PART.P_PARTKEY

           READ RECORD COLUMN : SUM( PART.P_SIZE )

             PHYSICAL FILTER : SUM( PART.P_SIZE ) > 0

     4  -  HASH SHARD ( # 3 )

           READ COLUMN : PART.P_PARTKEY, PART.P_SIZE


<<<  end print plan
```

以下为下级节点为cloned时进行grouping的示例

```
gSQL> \EXPLAIN PLAN

      SELECT s_nationkey

        FROM supplier

       GROUP BY s_nationkey

      HAVING COUNT( DISTINCT s_name ) > 0;
```

```
S_NATIONKEY

---------------

CANADA

UNITED STATES

GERMANY

KOREA

FRANCE


5 rows selected.


>>>  start print plan


< Execution Plan >

================================================================================

|  IDX  |  NODE DESCRIPTION                          |             ROWS |

--------------------------------------------------------------------------------

|    0  |  SELECT STATEMENT                          |                5 |

|    1  |    QUERY BLOCK ("$QB_IDX_2")               |                5 |

|    2  |      GROUP HASH INSTANT                    |                5 |

|    3  |        TABLE ACCESS ("SUPPLIER")           |                5 |

================================================================================


     1  -  TARGET : SUPPLIER.S_NATIONKEY

     2  -  GROUP KEY : SUPPLIER.S_NATIONKEY

           RECORD COLUMN : COUNT( DISTINCT SUPPLIER.S_NAME )
```

```
            READ KEY COLUMN : SUPPLIER.S_NATIONKEY

            READ RECORD COLUMN : COUNT( DISTINCT SUPPLIER.S_NAME )

              PHYSICAL FILTER : COUNT( DISTINCT SUPPLIER.S_NAME ) > 0

      3   -  CLONED

            READ COLUMN : SUPPLIER.S_NAME, SUPPLIER.S_NATIONKEY


<<<  end print plan
```

如上述结果所示对cloned table进行grouping时仅访问了local server（一个gorop）以下为对remot server进行数据访问时对cloned table进行grouping的示例

```
gSQL> \EXPLAIN PLAN

       SELECT s_nationkey

         FROM supplier@g2

        GROUP BY s_nationkey

       HAVING COUNT( DISTINCT s_name ) > 0;


S_NATIONKEY

---------------

CANADA

UNITED STATES

GERMANY

KOREA

FRANCE


5 rows selected.
```

```
>>>  start print plan


< Execution Plan >

================================================================================
|  IDX  |  NODE DESCRIPTION                        |                  ROWS |
--------------------------------------------------------------------------------
|    0  |  SELECT STATEMENT                        |                     5 |
|    1  |    QUERY BLOCK ("$QB_IDX_2")             |                     5 |
|    2  |      PLAN BASED CLUSTER                  | REMOTE ONLY         5
|
|    3  |        GROUP HASH INSTANT                |                     0 |
|    4  |          TABLE ACCESS ("SUPPLIER")       |                     0 |
================================================================================


     1  -  TARGET : SUPPLIER.S_NATIONKEY
     2  -  SQL : SELECT /*+ USE_GROUP_HASH(10) FULL( _A1 ) */
"_A1"."S_NATIONKEY" FROM "PUBLIC"."SUPPLIER"@LOCAL AS "_A1" GROUP BY
"_A1"."S_NATIONKEY" HAVING COUNT( DISTINCT "_A1"."S_NAME" ) > :_V0
           TARGET DOMAIN : G2(G2N1,G2N2) 5 rows, G3(G3N1,G3N2) 0 rows
     3  -  GROUP KEY : SUPPLIER.S_NATIONKEY
           RECORD COLUMN : COUNT( DISTINCT SUPPLIER.S_NAME )
           READ KEY COLUMN : SUPPLIER.S_NATIONKEY
           READ RECORD COLUMN : COUNT( DISTINCT SUPPLIER.S_NAME )
             PHYSICAL FILTER : COUNT( DISTINCT SUPPLIER.S_NAME ) > 0
```

```
    4  -  CLONED
          READ COLUMN : SUPPLIER.S_NAME, SUPPLIER.S_NATIONKEY


<<<  end print plan
```

处理grouping时无法使用下级节点的sharding策略的情况通过generated query按照各个group进行grouping收集数据后再次进行grouping并生成grouping结果此时构成的generated query不包含having子句having子句在生成grouping结果后评估

以下为无法使用下级节点的sharding策略的grouping示例

```
gSQL> \EXPLAIN PLAN
      SELECT p_type
        FROM part
       GROUP BY p_type
      HAVING SUM( p_size ) > 0;


P_TYPE
------
STEEL
NICKEL
COPPER


3 rows selected.


>>>  start print plan
```

```
< Execution Plan >
================================================================================
=
|  IDX  |  NODE DESCRIPTION                          |              ROWS
|
--------------------------------------------------------------------------------
-
|    0  |  SELECT STATEMENT                          |                 3
|
|    1  |    QUERY BLOCK ("$QB_IDX_2")               |                 3
|
|    2  |      SINGLE CLUSTER                        | LOCAL/REMOTE    3
|
|    3  |        SELECT STATEMENT                    |                 1
|
|    4  |          QUERY BLOCK ("$QB_IDX_2")         |                 1
|
|    5  |            GROUP HASH INSTANT              |                 1
|
|    6  |              TABLE ACCESS ("PART" AS _A1)  |                 1
|
================================================================================
=
```

```
     1  -  TARGET : PART.P_TYPE
     2  -  SQL : SELECT /*+ USE_GROUP_HASH(10) FULL( _A1 ) */
"_A1"."P_TYPE", SUM( "_A1"."P_SIZE" ) FROM "PUBLIC"."PART"@LOCAL AS "_A1"
GROUP BY "_A1"."P_TYPE"
           TARGET DOMAIN : G1(G1N1,G1N2) 1 rows, G2(G2N1,G2N2) 2 rows,
G3(G3N1,G3N2) 2 rows
           RE-GROUPING
             GROUP KEY : PART.P_TYPE
             AGGREGATION : SUM( SUM( PART.P_SIZE ) )
             PHYSICAL FILTER : SUM( PART.P_SIZE ) > 0
     4  -  TARGET : _A1.P_TYPE, SUM( _A1.P_SIZE )
     5  -  GROUP KEY : _A1.P_TYPE
           RECORD COLUMN : SUM( _A1.P_SIZE )
           READ KEY COLUMN : _A1.P_TYPE
           READ RECORD COLUMN : SUM( _A1.P_SIZE )
     6  -  HASH SHARD ( # 3 )
           READ COLUMN : _A1.P_TYPE, _A1.P_SIZE


<<<  end print plan
```

如上结果所示使用single cluster收集数据后进行了grouping

无法使用下级节点的sharding策略时可在generated query使用grouping key的ordering进行处理通过merge sorting按照grouping key对从各个group处理generated query收集的数据进行排列基于排列的数据再次进行grouping的操作被称为merge-grouping

以下为使用merge-grouping进行grouping的示例

```
gSQL> \EXPLAIN PLAN
      SELECT /*+ MERGE_GROUP */ p_brand
        FROM part
       GROUP BY p_brand
      HAVING SUM( p_size ) > 0;

P_BRAND
----------
Brand#1
Brand#2
Brand#3

3 rows selected.

>>>  start print plan

< Execution Plan >
================================================================================
==
| IDX | NODE DESCRIPTION                                |           ROWS
|
--------------------------------------------------------------------------------
--
```

```
|   0 | SELECT STATEMENT                                |              3
|
|   1 |  QUERY BLOCK ("$QB_IDX_2")                      |              3
|
|   2 |   MULTIPLE CLUSTER                              | LOCAL/REMOTE 3
|
|   3 |    SELECT STATEMENT                             |              1
|
|   4 |     QUERY BLOCK ("$QB_IDX_2")                   |              1
|
|   5 |      GROUP                                      |              1
|
|   6 |       INDEX ACCESS ("PART" AS _A1, "IDX_P_BRAND") | (    1)      1
|
================================================================================
==


     1  -  TARGET : PART.P_BRAND
     2  -  SQL : SELECT /*+ INDEX( _A1, "PUBLIC"."IDX_P_BRAND" ) */
"_A1"."P_BRAND", SUM( "_A1"."P_SIZE" ) FROM "PUBLIC"."PART"@LOCAL AS "_A1"
GROUP BY "_A1"."P_BRAND" ORDER BY "_A1"."P_BRAND" ASC NULLS LAST
           TARGET DOMAIN : G1(G1N1,G1N2) 1 rows, G2(G2N1,G2N2) 2 rows,
G3(G3N1,G3N2) 2 rows
           MERGE GROUPING
             SORT KEY : PART.P_BRAND
```

```
            GROUP KEY : PART.P_BRAND
            AGGREGATION : SUM( SUM( PART.P_SIZE ) )
            LOGICAL FILTER : SUM( PART.P_SIZE ) > 0
     4   -  TARGET : _A1.P_BRAND, SUM( _A1.P_SIZE )
     5   -  GROUP KEY : _A1.P_BRAND
            RECORD COLUMN : SUM( _A1.P_SIZE )
     6   -  HASH SHARD ( # 3 )
            READ INDEX COLUMN : _A1.P_BRAND
            READ TABLE COLUMN : _A1.P_SIZE

<<<  end print plan
```

ORDER BY语句

Cluster puller为了操作order by语句的数据使用merge sorting构成包含ordering的generated query收集各个group的数据收集到的数据按照ordering key的顺序排列并进行mergeOrder by节点无法拥有filter因此不变更merge sorting结果直接传输至上级节点

为了执行merge sorting使用multiple cluster

以下为使用merge sorting处理sharded table的order by语句的示例

```
gSQL> \EXPLAIN PLAN
      SELECT p_type
        FROM part
       ORDER BY p_size;
```

```
P_TYPE

------

NICKEL

COPPER

NICKEL

STEEL

STEEL


5 rows selected.


>>>  start print plan


< Execution Plan >

================================================================================

|  IDX  |  NODE DESCRIPTION                          |             ROWS |

--------------------------------------------------------------------------------

|    0  |  SELECT STATEMENT                          |                5 |

|    1  |    QUERY BLOCK ("$QB_IDX_2")               |                5 |

|    2  |      MULTIPLE CLUSTER                      | LOCAL/REMOTE   5 |

|    3  |        SELECT STATEMENT                    |                1 |

|    4  |          QUERY BLOCK ("$QB_IDX_2")         |                1 |

|    5  |            SORT INSTANT                    |                1 |

|    6  |              TABLE ACCESS ("PART" AS _A1)   |                1 |

================================================================================
```

```
     1  -  TARGET : PART.P_TYPE
     2  -  SQL : SELECT /*+ USE_ORDER_SORT FULL( _A1 ) */ "_A1"."P_SIZE",
"_A1"."P_TYPE" FROM "PUBLIC"."PART"@LOCAL AS "_A1" ORDER BY "_A1"."P_SIZE"
ASC NULLS LAST
           TARGET DOMAIN : G1(G1N1,G1N2) 1 rows, G2(G2N1,G2N2) 2 rows,
G3(G3N1,G3N2) 2 rows
           MERGE SORTING
             SORT KEY : PART.P_SIZE
     4  -  TARGET : _A1.P_SIZE, _A1.P_TYPE
     5  -  SORT KEY : "_A1.P_SIZE ASC NULLS LAST"
           RECORD COLUMN : _A1.P_TYPE
           READ KEY COLUMN : _A1.P_SIZE
           READ RECORD COLUMN : _A1.P_TYPE
     6  -  HASH SHARD ( # 3 )
           READ COLUMN : _A1.P_TYPE, _A1.P_SIZE

<<<  end print plan
```

Ordering key中包含non-deterministic信息时generated query无法包含ordering信息此时cluster puller plan node部署在order by node的下级

以下为包含non-deterministic信息的ordering示例

```
gSQL> \EXPLAIN PLAN
      SELECT p_type
        FROM part
```

```
       ORDER BY p_size, RANDOM( 1, 1 );


P_TYPE

------

NICKEL

COPPER

NICKEL

STEEL

STEEL


5 rows selected.


>>>  start print plan


< Execution Plan >

=============================================================================

|  IDX  |  NODE DESCRIPTION                    |                ROWS |

-----------------------------------------------------------------------------

|    0  |  SELECT STATEMENT                    |                   5 |

|    1  |    QUERY BLOCK ("$QB_IDX_2")         |                   5 |

|    2  |      SORT INSTANT                    |                   5 |

|    3  |        PLAN BASED CLUSTER            | LOCAL/REMOTE      5 |

|    4  |          TABLE ACCESS ("PART")       |                   1 |

=============================================================================
```

```
     1  -  TARGET : PART.P_TYPE
     2  -  SORT KEY : "PART.P_SIZE ASC NULLS LAST", "RANDOM(1,1) ASC NULLS
LAST"
           RECORD COLUMN : PART.P_TYPE
           READ RECORD COLUMN : PART.P_TYPE
     3  -  SQL : SELECT /*+ FULL( _A1 ) */ "_A1"."P_TYPE", "_A1"."P_SIZE"
FROM "PUBLIC"."PART"@LOCAL AS "_A1"
           TARGET DOMAIN : G1(G1N1,G1N2) 1 rows, G2(G2N1,G2N2) 2 rows,
G3(G3N1,G3N2) 2 rows
     4  -  HASH SHARD ( # 3 )
           READ COLUMN : PART.P_TYPE, PART.P_SIZE


<<<  end print plan
```

以下情况可以通过一个cluster puller处理order by语句和group by语句

- 所有ordering key包含在grouping key时
- grouping下级节点为cloned时
- grouping下级节点的所有sharding key用作grouping key时

以下为所有ordering key包含在grouping key时的示例

```
gSQL> \EXPLAIN PLAN
      SELECT p_partkey, COUNT( p_type )
        FROM part
       GROUP BY p_partkey
```

```
       ORDER BY p_partkey;



P_PARTKEY COUNT( P_TYPE )
--------- ---------------
        1               1
        2               1
        3               1
        4               1
        5               1

5 rows selected.

>>>  start print plan

< Execution Plan >
================================================================================
|  IDX  |  NODE DESCRIPTION                          |             ROWS |
--------------------------------------------------------------------------------
|    0  |  SELECT STATEMENT                          |                5 |
|    1  |    QUERY BLOCK ("$QB_IDX_2")               |                5 |
|    2  |      MULTIPLE CLUSTER                       | LOCAL/REMOTE   5
|
|    3  |        SELECT STATEMENT                    |                1 |
|    4  |          QUERY BLOCK ("$QB_IDX_2")         |                1 |
```

```
|    5  |            SORT INSTANT                      |               1 |
|    6  |              GROUP HASH INSTANT              |               1 |
|    7  |                TABLE ACCESS ("PART" AS _A1)  |               1 |
=============================================================================


     1  -  TARGET : PART.P_PARTKEY, COUNT( PART.P_TYPE )
     2  -  SQL : SELECT /*+ USE_ORDER_SORT USE_GROUP_HASH(500) FULL( _A1 )
*/ "_A1"."P_PARTKEY", COUNT( "_A1"."P_TYPE" ) FROM "PUBLIC"."PART"@LOCAL
AS "_A1"  GROUP BY "_A1"."P_PARTKEY" ORDER BY "_A1"."P_PARTKEY" ASC NULLS
LAST
           TARGET DOMAIN : G1(G1N1,G1N2) 1 rows, G2(G2N1,G2N2) 2 rows,
G3(G3N1,G3N2) 2 rows
           MERGE SORTING
             SORT KEY : PART.P_PARTKEY
     4  -  TARGET : _A1.P_PARTKEY, COUNT( _A1.P_TYPE )
     5  -  SORT KEY : "_A1.P_PARTKEY ASC NULLS LAST"
           RECORD COLUMN : COUNT( _A1.P_TYPE )
           READ KEY COLUMN : _A1.P_PARTKEY
           READ RECORD COLUMN : COUNT( _A1.P_TYPE )
     6  -  GROUP KEY : _A1.P_PARTKEY
           RECORD COLUMN : COUNT( _A1.P_TYPE )
           READ KEY COLUMN : _A1.P_PARTKEY
           READ RECORD COLUMN : COUNT( _A1.P_TYPE )
     7  -  HASH SHARD ( # 3 )
           READ COLUMN : _A1.P_PARTKEY, _A1.P_TYPE
```

```
<<<  end print plan
```

以下为grouping下级节点为cloned时通过一个cluster puller执行ordering和grouping的示例

```
gSQL> \EXPLAIN PLAN
      SELECT s_nationkey, COUNT( s_suppkey )
        FROM supplier@G2|G3
       GROUP BY s_nationkey
       ORDER BY COUNT( s_suppkey );

S_NATIONKEY     COUNT( S_SUPPKEY )
--------------- ------------------
CANADA                          1
UNITED STATES                   1
GERMANY                         1
KOREA                           1
FRANCE                          1

5 rows selected.

>>>  start print plan

< Execution Plan >
======================================================================
|  IDX  |  NODE DESCRIPTION                      |              ROWS |
```

```
-------------------------------------------------------------------------
|    0  |  SELECT STATEMENT                       |                 5 |
|    1  |    QUERY BLOCK ("$QB_IDX_2")            |                 5 |
|    2  |      PLAN BASED CLUSTER                 | REMOTE ONLY     5 |
|    3  |        SORT INSTANT                     |                 0 |
|    4  |          GROUP HASH INSTANT             |                 0 |
|    5  |            TABLE ACCESS ("SUPPLIER")    |                 0 |
=========================================================================


     1  -  TARGET : SUPPLIER.S_NATIONKEY, COUNT( SUPPLIER.S_SUPPKEY )
     2  -  SQL : SELECT /*+ USE_ORDER_SORT USE_GROUP_HASH(10) FULL( _A1 )
*/ COUNT( "_A1"."S_SUPPKEY" ), "_A1"."S_NATIONKEY" FROM
"PUBLIC"."SUPPLIER"@LOCAL AS "_A1"  GROUP BY "_A1"."S_NATIONKEY" ORDER BY
COUNT( "_A1"."S_SUPPKEY" ) ASC NULLS LAST
            TARGET DOMAIN : G2(G2N1,G2N2) 5 rows, G3(G3N1,G3N2) 0 rows
     3  -  SORT KEY : "COUNT( SUPPLIER.S_SUPPKEY ) ASC NULLS LAST"
            RECORD COLUMN : SUPPLIER.S_NATIONKEY
            READ KEY COLUMN : COUNT( SUPPLIER.S_SUPPKEY )
            READ RECORD COLUMN : SUPPLIER.S_NATIONKEY
     4  -  GROUP KEY : SUPPLIER.S_NATIONKEY
            RECORD COLUMN : COUNT( SUPPLIER.S_SUPPKEY )
            READ KEY COLUMN : SUPPLIER.S_NATIONKEY
            READ RECORD COLUMN : COUNT( SUPPLIER.S_SUPPKEY )
     5  -  CLONED
            READ COLUMN : SUPPLIER.S_SUPPKEY, SUPPLIER.S_NATIONKEY
```

```
<<<  end print plan
```

以下为grouping下级节点的所有sharding key用作grouping key时通过一个cluster puller执行ordering和grouping的示例

```
gSQL> \EXPLAIN PLAN
      SELECT p_partkey, COUNT( p_type )
        FROM part
       GROUP BY p_partkey
       ORDER BY COUNT( p_type );

P_PARTKEY COUNT( P_TYPE )
--------- ---------------
        3               1
        2               1
        5               1
        1               1
        4               1

5 rows selected.

>>>  start print plan

< Execution Plan >
================================================================================
```

```
=
|  IDX   |  NODE DESCRIPTION                                |             ROWS
|
-------------------------------------------------------------------------------
-
|    0   |  SELECT STATEMENT                                |               5
|
|    1   |    QUERY BLOCK ("$QB_IDX_2")                     |               5
|
|    2   |      MULTIPLE CLUSTER                            | LOCAL/REMOTE  5
|
|    3   |        SELECT STATEMENT                          |               1
|
|    4   |          QUERY BLOCK ("$QB_IDX_2")               |               1
|
|    5   |            SORT INSTANT                          |               1
|
|    6   |              GROUP HASH INSTANT                  |               1
|
|    7   |                TABLE ACCESS ("PART" AS _A1)      |               1
|
===============================================================================
=


     1  -  TARGET : PART.P_PARTKEY, COUNT( PART.P_TYPE )
```

```
     2  -  SQL : SELECT /*+ USE_ORDER_SORT USE_GROUP_HASH(500) FULL( _A1 )
*/ COUNT( "_A1"."P_TYPE" ), "_A1"."P_PARTKEY" FROM "PUBLIC"."PART"@LOCAL
AS "_A1"  GROUP BY "_A1"."P_PARTKEY" ORDER BY COUNT( "_A1"."P_TYPE" ) ASC
NULLS LAST
           TARGET DOMAIN : G1(G1N1,G1N2) 1 rows, G2(G2N1,G2N2) 2 rows,
G3(G3N1,G3N2) 2 rows
           MERGE SORTING
             SORT KEY : COUNT( PART.P_TYPE )
     4  -  TARGET : COUNT( _A1.P_TYPE ), _A1.P_PARTKEY
     5  -  SORT KEY : "COUNT( _A1.P_TYPE ) ASC NULLS LAST"
           RECORD COLUMN : _A1.P_PARTKEY
           READ KEY COLUMN : COUNT( _A1.P_TYPE )
           READ RECORD COLUMN : _A1.P_PARTKEY
     6  -  GROUP KEY : _A1.P_PARTKEY
           RECORD COLUMN : COUNT( _A1.P_TYPE )
           READ KEY COLUMN : _A1.P_PARTKEY
           READ RECORD COLUMN : COUNT( _A1.P_TYPE )
     7  -  HASH SHARD ( # 3 )
           READ COLUMN : _A1.P_PARTKEY, _A1.P_TYPE


<<<  end print plan
```

DISTINCT语句

执行distinct语句的cluster puller的generated query包括distinct语句对sharded table执行distinct时收集数据后通过grouping生成结果Cloned table的distinct执行不变更收集的数据直接生成为结

果

以下为处理sharded table的distinct语句的示例

```
gSQL> \EXPLAIN PLAN
      SELECT DISTINCT p_name
        FROM part;

P_NAME
------
Part#2
Part#4
Part#3
Part#1
Part#5

5 rows selected.

>>>  start print plan

< Execution Plan >
================================================================================
=
|  IDX  |  NODE DESCRIPTION                              |              ROWS
|
--------------------------------------------------------------------------------
```

```
-
|    0  |  SELECT STATEMENT                          |                 5
|
|    1  |    QUERY BLOCK ("$QB_IDX_2")               |                 5
|
|    2  |      SINGLE CLUSTER                        | LOCAL/REMOTE    5
|
|    3  |        SELECT STATEMENT                    |                 1
|
|    4  |          QUERY BLOCK ("$QB_IDX_2")         |                 1
|
|    5  |            GROUP HASH INSTANT              |                 1
|
|    6  |              TABLE ACCESS ("PART" AS _A1)  |                 1
|
================================================================================
=


     1  -  TARGET : PART.P_NAME
     2  -  SQL : SELECT /*+ USE_DISTINCT_HASH(10) FULL( _A1 ) */ DISTINCT
"_A1"."P_NAME" FROM "PUBLIC"."PART"@LOCAL AS "_A1"
           TARGET DOMAIN : G1(G1N1,G1N2) 1 rows, G2(G2N1,G2N2) 2 rows,
G3(G3N1,G3N2) 2 rows
           RE-GROUPING
             GROUP KEY : PART.P_NAME
```

```
     4  -  TARGET : _A1.P_NAME
     5  -  GROUP KEY : _A1.P_NAME
           READ KEY COLUMN : _A1.P_NAME
     6  -  HASH SHARD ( # 3 )
           READ COLUMN : _A1.P_NAME

<<<  end print plan
```

以下为处理remote server的cloned table的distinct语句的示例

```
gSQL> \EXPLAIN PLAN
      SELECT DISTINCT s_name, s_nationkey
        FROM supplier@G2;

S_NAME                    S_NATIONKEY
------------------------- ---------------
Supplier#1                FRANCE
Supplier#5                CANADA
Supplier#4                UNITED STATES
Supplier#3                GERMANY
Supplier#2                KOREA

5 rows selected.

>>>  start print plan
```

```
< Execution Plan >

=========================================================================

|  IDX  |  NODE DESCRIPTION                      |              ROWS |

-------------------------------------------------------------------------

|    0  |  SELECT STATEMENT                      |                 5 |

|    1  |    QUERY BLOCK ("$QB_IDX_2")           |                 5 |

|    2  |      PLAN BASED CLUSTER                | REMOTE ONLY     5 |

|    3  |        GROUP HASH INSTANT              |                 0 |

|    4  |          TABLE ACCESS ("SUPPLIER")     |                 0 |

=========================================================================


     1  -  TARGET : SUPPLIER.S_NAME, SUPPLIER.S_NATIONKEY

     2  -  SQL : SELECT /*+ USE_DISTINCT_HASH(100) FULL( _A1 ) */
DISTINCT "_A1"."S_NAME", "_A1"."S_NATIONKEY" FROM "PUBLIC"."SUPPLIER"@LOCAL AS
"_A1"

           TARGET DOMAIN : G2(G2N1,G2N2) 5 rows

     3  -  GROUP KEY : SUPPLIER.S_NAME, SUPPLIER.S_NATIONKEY

           READ KEY COLUMN : SUPPLIER.S_NAME, SUPPLIER.S_NATIONKEY

     4  -  CLONED

           READ COLUMN : SUPPLIER.S_NAME, SUPPLIER.S_NATIONKEY


<<<  end print plan
```

Single Row语句

集群中single row查询处理按照cloned table或sharded table进行分类处理cloned table的single

row查询的generated query包含所有aggregation function收集仅在一个group执行generated query的数据并生成结果

以下为remote server的cloned table的single row语句示例

```
gSQL> \EXPLAIN PLAN
      SELECT COUNT( DISTINCT s_name ), SUM( s_suppkey )
        FROM supplier@G2;

COUNT( DISTINCT S_NAME ) SUM( S_SUPPKEY )
------------------------ ----------------
                       5               15

1 row selected.

>>>  start print plan

< Execution Plan >
=====================================================================
|  IDX  |  NODE DESCRIPTION                      |              ROWS |
---------------------------------------------------------------------
|    0  |  SELECT STATEMENT                      |                 1 |
|    1  |    QUERY BLOCK ("$QB_IDX_2")           |                 1 |
|    2  |      PLAN BASED CLUSTER                | REMOTE ONLY      1 |
|    3  |        AGGREGATION BY HASH             |                 0 |
|    4  |          TABLE ACCESS ("SUPPLIER")     |                 0 |
```

```
=========================================================================


     1  -  TARGET : COUNT( DISTINCT SUPPLIER.S_NAME ),
SUM( SUPPLIER.S_SUPPKEY )
     2  -  SQL : SELECT /*+ FULL( _A1 ) */ COUNT( DISTINCT "_A1"."S_NAME" ),
SUM( "_A1"."S_SUPPKEY" ) FROM "PUBLIC"."SUPPLIER"@LOCAL AS "_A1"
           TARGET DOMAIN : G2(G2N1,G2N2) 1 rows
     3  -  AGGREGATION : SUM( SUPPLIER.S_SUPPKEY )
           DISTINCT AGGREGATION : COUNT( DISTINCT SUPPLIER.S_NAME )
     4  -  CLONED
           READ COLUMN : SUPPLIER.S_SUPPKEY, SUPPLIER.S_NAME


<<<  end print plan
```

Sharded table的single row查询处理根据aggregation function是否包含distinct进行区分

有一个以上的包含distinct的aggregation function时

- Cluster puller plan node部署在处理single row的plan的下级
- Generated query不包含aggregation function

以下为处理有包含distinct的aggregation的sharded table的single row的示例

```
gSQL> \EXPLAIN PLAN
      SELECT COUNT( DISTINCT p_name ), SUM( p_size )
        FROM part;
```

```
COUNT( DISTINCT P_NAME ) SUM( P_SIZE )
------------------------ -------------
                       5            58


1 row selected.


>>>  start print plan


< Execution Plan >
=====================================================================
|  IDX  |  NODE DESCRIPTION                  |               ROWS |
---------------------------------------------------------------------
|    0  |  SELECT STATEMENT                  |                  1 |
|    1  |    QUERY BLOCK ("$QB_IDX_2")       |                  1 |
|    2  |      AGGREGATION BY HASH           |                  1 |
|    3  |        PLAN BASED CLUSTER          | LOCAL/REMOTE     5 |
|    4  |          TABLE ACCESS ("PART")     |                  1 |
=====================================================================


     1  -  TARGET : COUNT( DISTINCT PART.P_NAME ), SUM( PART.P_SIZE )
     2  -  AGGREGATION : SUM( PART.P_SIZE )
           DISTINCT AGGREGATION : COUNT( DISTINCT PART.P_NAME )
     3  -  SQL : SELECT /*+ FULL( _A1 ) */ "_A1"."P_NAME", "_A1"."P_SIZE"
FROM "PUBLIC"."PART"@LOCAL AS "_A1"
           TARGET DOMAIN : G1(G1N1,G1N2) 1 rows, G2(G2N1,G2N2) 2 rows,
```

```
G3(G3N1,G3N2) 2 rows

     4  -  HASH SHARD ( # 3 )

           READ COLUMN : PART.P_NAME, PART.P_SIZE


<<<  end print plan
```

所有aggregation function不包含distinct时

- 不构成处理single row的plan
- Generated query包含aggregation function
- 收集数据后执行aggregation并生成结果

以下为处理没有包含distinct的aggregation的sharded table的single row的示例

```
gSQL> \EXPLAIN PLAN

      SELECT COUNT( p_name ), SUM( p_size )

        FROM part;


COUNT( P_NAME ) SUM( P_SIZE )

--------------- -------------

              5            58


1 row selected.


>>>  start print plan
```

```
< Execution Plan >

================================================================================
|  IDX  |  NODE DESCRIPTION                          |               ROWS |
--------------------------------------------------------------------------------
|    0  |  SELECT STATEMENT                          |                  1 |
|    1  |    QUERY BLOCK ("$QB_IDX_2")               |                  1 |
|    2  |      SINGLE CLUSTER                        | LOCAL/REMOTE     1 |
|    3  |        SELECT STATEMENT                    |                  1 |
|    4  |          QUERY BLOCK ("$QB_IDX_2")         |                  1 |
|    5  |            TABLE ACCESS ("PART" AS _A1)    |                  1 |
================================================================================


     1  -  TARGET : COUNT( PART.P_NAME ), SUM( PART.P_SIZE )
     2  -  SQL : SELECT /*+ FULL( _A1 ) */ COUNT( "_A1"."P_NAME" ),
SUM( "_A1"."P_SIZE" ) FROM "PUBLIC"."PART"@LOCAL AS "_A1"
           TARGET DOMAIN : G1(G1N1,G1N2) 1 rows, G2(G2N1,G2N2) 1 rows,
G3(G3N1,G3N2) 1 rows
           RE-AGGREGATION
             AGGREGATION : SUM( COUNT( PART.P_NAME ) ),
SUM( SUM( PART.P_SIZE ) )
     4  -  TARGET : COUNT( _A1.P_NAME ), SUM( _A1.P_SIZE )
     5  -  HASH SHARD ( # 3 )
           READ COLUMN : _A1.P_NAME, _A1.P_SIZE
           AGGREGATION : COUNT( _A1.P_NAME ), SUM( _A1.P_SIZE )
```

```
<<<  end print plan
```

集群的DML处理

在SUNDB用户可以在构成集群系统的所有集群成员中执行DML

在集群环境中操作数据时对拥有相同数据副本（replica）的集群成员同样执行数据变更

以下为说明数据操作的示例表

```
CREATE TABLE t1( shard_key INTEGER, c1 INTEGER )
    SHARDING BY RANGE( shard_key )
        SHARD s1 VALUES LESS THAN ( 200 )       AT CLUSTER GROUP G1,
        SHARD s2 VALUES LESS THAN ( 400 )       AT CLUSTER GROUP G2,
        SHARD s3 VALUES LESS THAN ( MAXVALUE )  AT CLUSTER GROUP G3;
```

集群环境中的DML执行流程如下

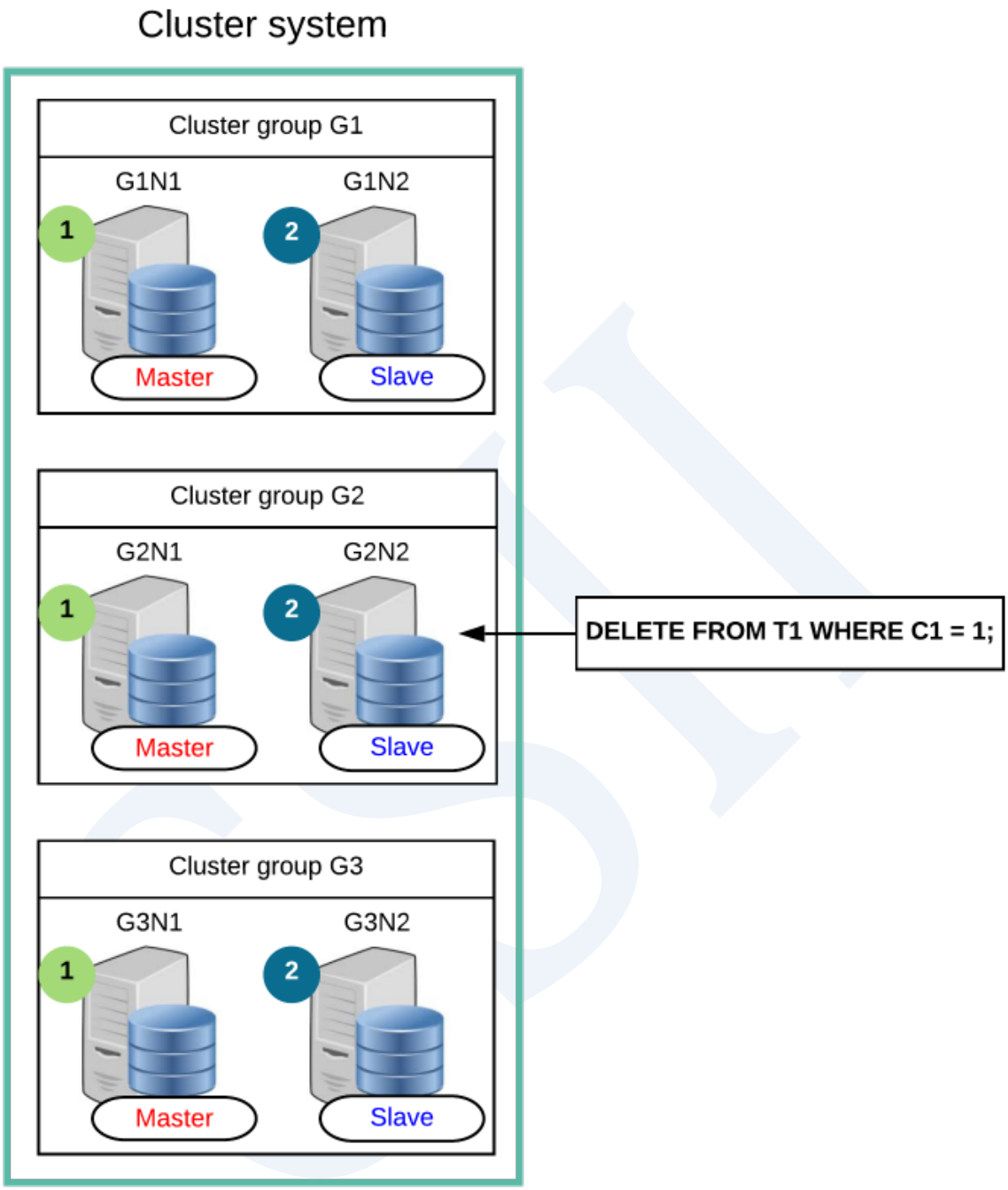


Figure 2-9 集群的DML处理

为了集群的DML处理定义各群组的master服务器与slave服务器

选定各集群群组的Master服务器

在集群环境中执行操作数据时在各集群群组中的可访问的集群成员中选择最先包含的集群成员

选定各集群群组的Slave服务器

在集群环境中执行操作数据时在各集群群组中可访问的集群成员中选择除master服务器外的其余集群成员

通过**DBA_CLUSTER**可查看集群群组及集群成员的构成信息

```
gSQL> SELECT * FROM DBA_CLUSTER;


GROUP_ID GROUP_NAME MEMBER_ID MEMBER_NAME MEMBER_HOST MEMBER_PORT
MEMBER_POSITION
-------- ---------- --------- ----------- ----------- ----------- --------
-------
       1 G1                 1 G1N1        127.0.0.1         11150
0
       1 G1                 2 G1N2        127.0.0.1         11250
1
       2 G2                 3 G2N1        127.0.0.1         12150
2
       2 G2                 4 G2N2        127.0.0.1         12250
3
       3 G3                 5 G3N1        127.0.0.1         13150
```

```
4
       3 G3                 6 G3N2        127.0.0.1         13250
5


6 rows selected.
```

执行DML

DML的执行分为master服务器反映阶段与slave服务器反映阶段

- master服务器反映阶段
  - 操作各个集群群组的master服务器的数据
- slave服务器反映阶段
  - 对slave服务器执行与各集群群组的master服务器相同的操作

SUNDB维持各个集群群组的master服务器与slave服务器之间的同步并支持以下两种操作数据的方法

- **基于Query的DML**
- **基于Global Rowid的DML**

基于Query的DML

基于query的DML是从用户接收query的服务器使用内部生成的generated query操作各个服务器的记录的方法仅限于各个服务器使用generated query可保障执行DML的结果的统一性时支持

详细内容参考**Generated Query**

根据是否返回操作的记录的结果master服务器与slave服务器的generated query会有所不同

使用generated query的数据操作按照如下步骤执行

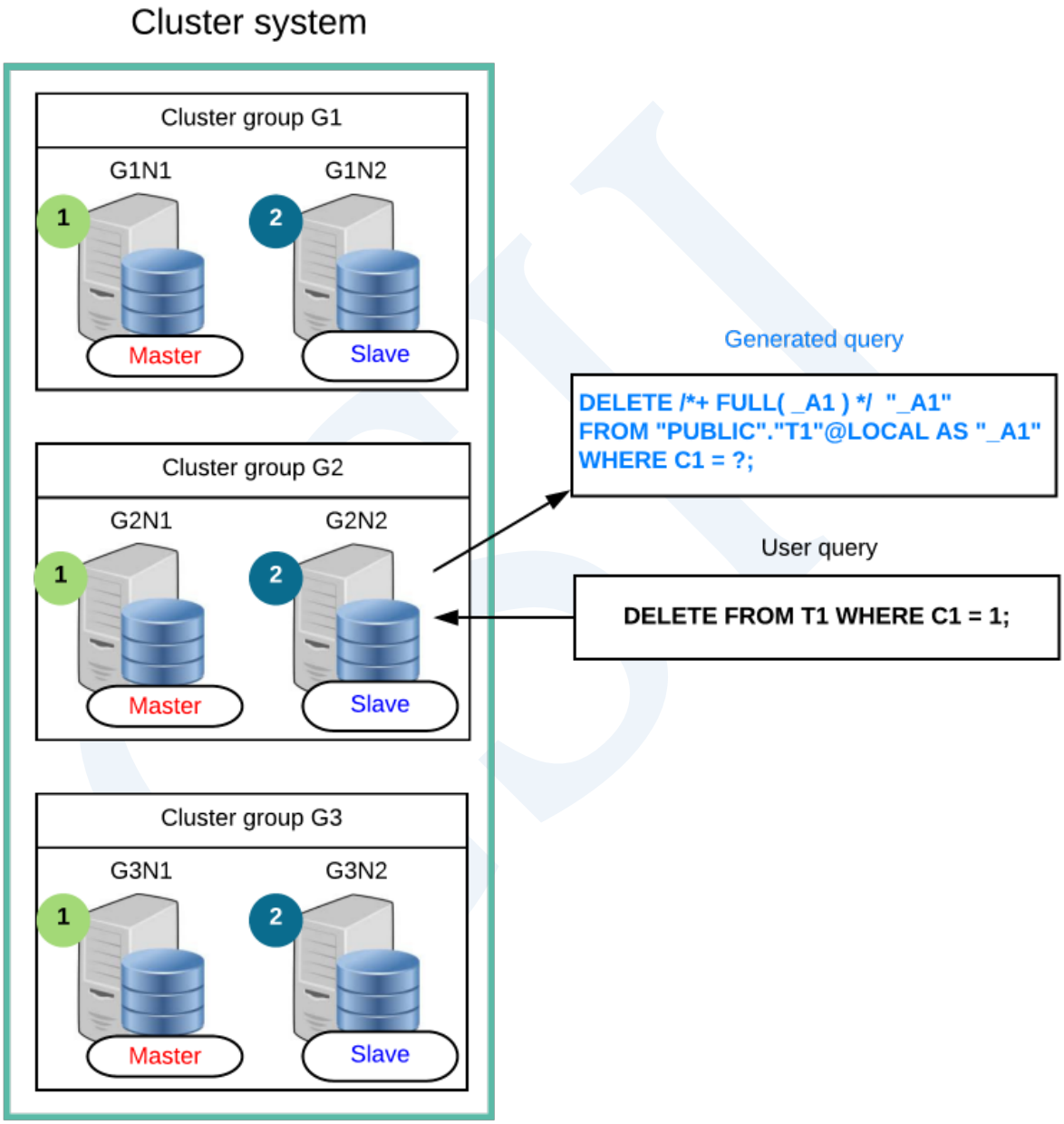


Figure 2-10 使用generated query的数据变更（以整体集群群组为对象进行变更)

如上图所示条件语句上无选择变更对象集群群组的条件时所有集群群组成为使用generated

query的数据操作的对象

使用generated query的数据操作按照如下步骤执行

1. 在各个master服务器执行generated query
2. 在各个slave服务器执行generated query

通过条件语句将特定集群群组指定为数据操作对象时按照如下步骤执行

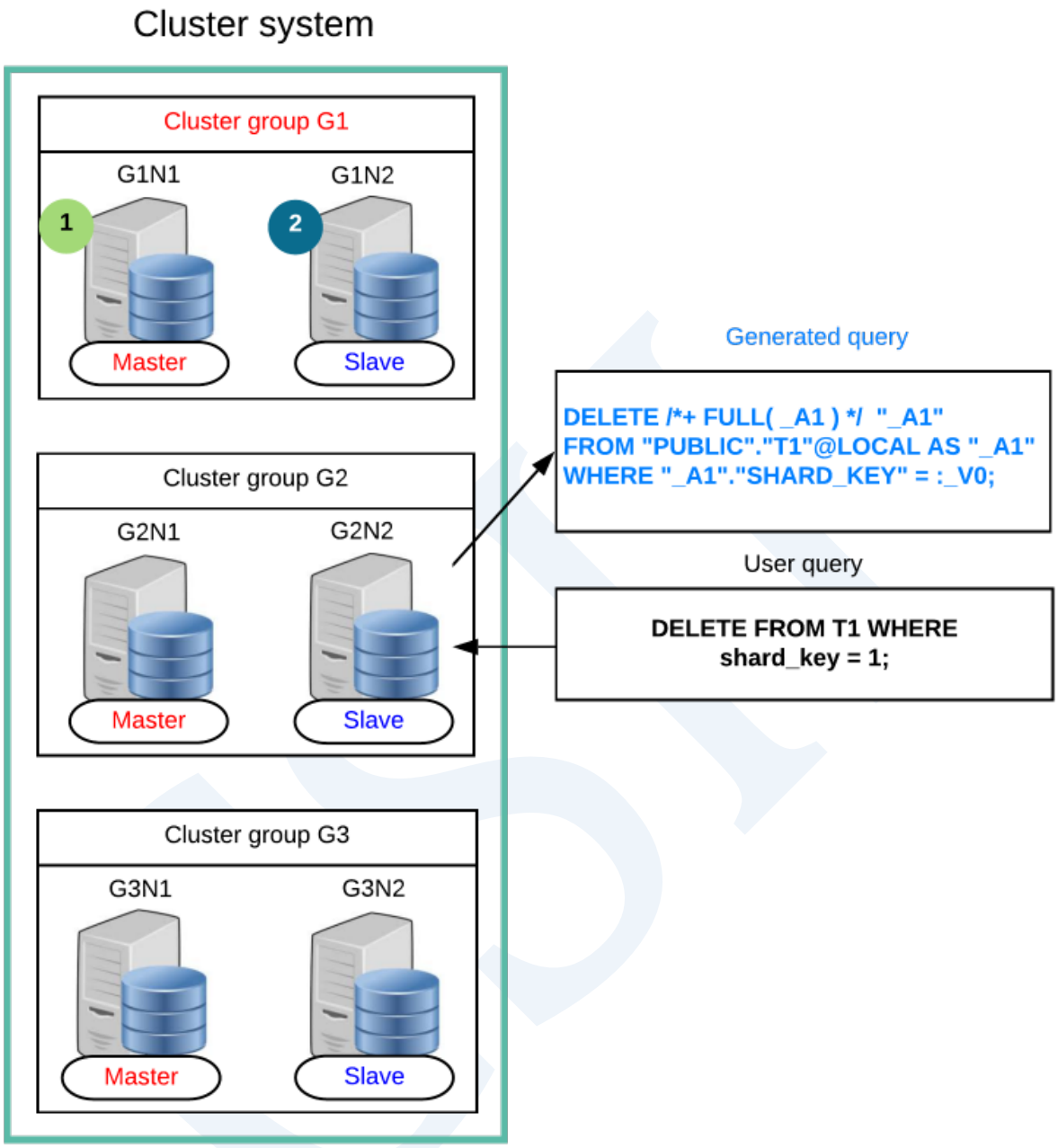


Figure 2-11 使用generated query的数据变更（以特定集群群组为对象进行变更)

如上图所示使用shard_key = 1的搜索条件指定操作对象时根据sharding策略shard_key为1的记录在集群群组的G1中因此仅删除G1集群群组的记录

以特定集群群组为对象进行数据操作时与以整体为对象进行操作的步骤相同先反映到Master后

执行Slave

基于query执行DML叫做DML cluster由plan node负责详细内容参考**DML Cluster**

限于如下情况支持使用generated query的数据操作

- 可生成可保障在一个集群群组的各个集群成员中执行generated query的一致性的generated query时
  - 参考**Generated Query构成约束事项**
- 构成generated query时数据参照对象服务器与数据操作对象服务器限于相同的一个服务器时
  - 指执行generated query的服务器在处理查询的过程中不需要访问其他服务器

支持使用generated query的数据操作的用户查询类型如下

- **SELECT .. FOR UPDATE**
- **SELECT .. INTO .. FOR UPDATE**
- **DELETE FROM**
- **DELETE FROM name RETURNING**
- **DELETE FROM name RETURNING .. INTO**
- **UPDATE**
- **UPDATE name RETURNING**
- **UPDATE name RETURNING .. INTO**

未构成**Global Secondary Index（全局二级索引）**时也可执行使用generated query的数据操作

DML Cluster

DML Cluster使用generated query操作各个服务器的数据必要时收集数据

以下为通过基于query的DML处理sharded table的DELETE RETURN语句的示例

```
gSQL> \EXPLAIN PLAN DELETE FROM part WHERE p_partkey = 5 RETURN p_name;


P_NAME

------

Part#3


1 row deleted.


>>>  start print plan


< Execution Plan >

================================================================================

|  IDX  |  NODE DESCRIPTION                                |           ROWS |

--------------------------------------------------------------------------------

|    0  |  DELETE STATEMENT ("PART")                       |              1 |

|    1  |    QUERY BLOCK ("$QB_IDX_2")                     |              0 |

|    2  |      DML CLUSTER                                  | REMOTE ONLY  1

|

|    3  |        INDEX ACCESS ("PART", "PART_PK_INDEX") | (    0)      0 |

================================================================================
```

```
     1   -  TARGET : PART.P_NAME
     2   -  FETCH
            Fetch SQL : DELETE /*+ INDEX( _A1, "PUBLIC"."PART_PK_INDEX" )
*/  "_A1" FROM "PUBLIC"."PART"@LOCAL AS "_A1" WHERE "_A1"."P_PARTKEY"
= :_V0 RETURN "_A1"."$PHYSICAL_ROWID", "_A1"."P_PARTKEY", "_A1"."P_NAME",
"_A1"."P_BRAND"
            Non-Fetch SQL : DELETE /*+ INDEX( _A1,
"PUBLIC"."PART_PK_INDEX" ) */  "_A1" FROM "PUBLIC"."PART"@LOCAL AS "_A1"
WHERE "_A1"."P_PARTKEY" = :_V0
            TARGET DOMAIN : G2(G2N1,G2N2) 1 rows
     3   -  HASH SHARD ( # 3 )
            READ INDEX COLUMN : PART.P_PARTKEY
            READ TABLE COLUMN : PART.P_NAME, PART.P_BRAND
              MIN RANGE : PART.P_PARTKEY = 5
              MAX RANGE : PART.P_PARTKEY = 5
            FETCH ONE ROW

<<<  end print plan
```

如上述查询执行结果为了执行DELETE语句使用了DML cluster上述输出的执行信息中DML cluster属于<Execution Plan>的idx为2的plan

DML cluster的详细信息为如下

- DML cluster使用类型：FETCH, WITHOUT FETCH, SHARD KEY UPDATE

- Fetch SQL：执行DML和收集数据的generated query
- Non-fetch SQL：仅执行DML的generated query
- TARGET DOMAIN：发送Generated query的对象group和从成员及对应group接收的数据数量
- Shard key update信息：将一个UPDATE语句分为UPDATESELECTDELETE的generated query

DML cluster使用类型按照用户查询进行区分

- WITHOUT FETCH：不获取执行结果数据的DML(DELETE, UPDATE）
- FETCH：用于获取执行结果数据的DML(SELECT FOR UPDATE, DELETE RETURN, UPDATE RETURN)
- SHARD KEY UPDATE：更新sharding key的UPDATE (UPDATE, UPDATE RETURN)

DML Cluster (WITHOUT FETCH)

DML cluster使用类型为WITHOUT FETCH时DML cluster的详细信息如下由non-fetch SQL和TARGET DOMAIN构成

```
gSQL> \EXPLAIN PLAN DELETE FROM supplier;


5 rows deleted.


>>>  start print plan


< Execution Plan >

========================================================================
```

```
|  IDX  |  NODE DESCRIPTION                          |           ROWS |
-------------------------------------------------------------------------
|    0  |  DELETE STATEMENT ("SUPPLIER")             |              5 |
|    1  |    QUERY BLOCK ("$QB_IDX_2")               |              0 |
|    2  |      DML CLUSTER                           |              5 |
|    3  |        INDEX ACCESS ("SUPPLIER", ...)      | (     5)     5 |
=========================================================================


     1  -  TARGET : NOTHING
     2  -  WITHOUT FETCH
           Non-Fetch SQL : DELETE /*+ INDEX( _A1,
"PUBLIC"."SUPPLIER_PK_INDEX" ) */  "_A1" FROM "PUBLIC"."SUPPLIER"@LOCAL AS
"_A1"
           TARGET DOMAIN : G1(G1N1,G1N2) 5 rows, G2(G2N1,G2N2) 5 rows,
G3(G3N1,G3N2) 5 rows
     3  -  CLONED
           READ INDEX COLUMN : SUPPLIER.S_SUPPKEY


<<<  end print plan
```

如下sharded表数据操作仅影响特定group时执行non-fetch SQL的group有限

```
gSQL> \EXPLAIN PLAN DELETE FROM part WHERE p_partkey = 5;


1 row deleted.
```

```
>>>  start print plan


< Execution Plan >

=============================================================================
|  IDX  |  NODE DESCRIPTION                                |        ROWS |
-----------------------------------------------------------------------------
|    0  |  DELETE STATEMENT ("PART")                       |           1 |
|    1  |    QUERY BLOCK ("$QB_IDX_2")                     |           0 |
|    2  |      DML CLUSTER                                 |           1 |
|    3  |        INDEX ACCESS ("PART", "PART_PK_INDEX") | (    0)   0 |
=============================================================================


     1  -  TARGET : NOTHING
     2  -  WITHOUT FETCH
           Non-Fetch SQL : DELETE /*+ INDEX( _A1,
"PUBLIC"."PART_PK_INDEX" ) */  "_A1" FROM "PUBLIC"."PART"@LOCAL AS "_A1"
WHERE "_A1"."P_PARTKEY" = :_V0
           TARGET DOMAIN : G2(G2N1,G2N2) 1 rows
     3  -  HASH SHARD ( # 3 )
           READ INDEX COLUMN : PART.P_PARTKEY
           READ TABLE COLUMN : PART.P_BRAND
             MIN RANGE : PART.P_PARTKEY = 5
             MAX RANGE : PART.P_PARTKEY = 5
           FETCH ONE ROW
```

```
<<<  end print plan
```

由WITHOUT FETCH构成的cluster的non-fetch SQL不区分cloned表和sharded表相同的在所有master服务器和slave服务器执行

DML Cluster (FETCH)

DML cluster的使用类型为FETCH时DML cluster的详细信息如下由fetch SQLnon-fetch SQLTARGET DOMAIN构成

```
gSQL> \EXPLAIN PLAN SELECT p_name FROM part FOR UPDATE;


P_NAME

------

Part#1

Part#4

Part#3

Part#2

Part#5


5 rows selected.


>>>  start print plan


< Execution Plan >

================================================================
```

```
|  IDX   |  NODE DESCRIPTION                |               ROWS |
-----------------------------------------------------------------
|    0   |  SELECT FOR UPDATE STATEMENT     |                  5 |
|    1   |    QUERY BLOCK ("$QB_IDX_2")     |                  0 |
|    2   |      DML CLUSTER                 | REMOTE ONLY      5 |
|    3   |        TABLE ACCESS ("PART")     |                  2 |
=================================================================


     1  -  TARGET : PART.P_NAME
     2  -  FETCH
           Fetch SQL : SELECT /*+ FULL( _A1 ) */ "_A1"."$PHYSICAL_ROWID",
"_A1"."P_NAME" FROM "PUBLIC"."PART"@LOCAL AS "_A1" FOR UPDATE OF
"_A1"."P_PARTKEY"
           Non-Fetch SQL : SELECT /*+ FULL( _A1 ) */  NULL FROM
"PUBLIC"."PART"@LOCAL AS "_A1" FOR UPDATE OF "_A1"."P_PARTKEY" WITHOUT
FETCH
           TARGET DOMAIN : G1(G1N1,G1N2) 2 rows, G2(G2N1,G2N2) 1 rows,
G3(G3N1,G3N2) 2 rows
     3  -  HASH SHARD ( # 3 )
           READ COLUMN : PART.P_NAME


<<<  end print plan
```

上述结果中non-fetch SQL由SELECT FOR UPDATE语句构成这样构成的non-tetch SQL在local服务器和remote服务器中执行但不通过SELECT FOR UPDATE语句收集数据

Cloned表的基于query的DML的fetch SQL在所有group中的一个member中执行如果local服务器拥有cloned表的复制时在local服务器执行Fetch SQLLocal服务器中没有cloned表的复制时在所有master服务器中的任意一个服务器执行fetch SQL除执行Fetch SQL的服务器外的拥有cloned表的复制的服务器均执行non-fetch SQL

Sharded表的基于query的DML的fetch SQL在各个group的master服务器中执行所有Slave服务器均执行non-fetch SQL

如下sharded表数据操作仅影响特定group时执行Fetch SQL和non-fetch SQL的group是有限的

```
gSQL> \EXPLAIN PLAN SELECT p_name FROM part WHERE p_partkey = 5 FOR
UPDATE;


P_NAME
------
Part#3


1 row selected.


>>>  start print plan


< Execution Plan >
================================================================================
|  IDX  |  NODE DESCRIPTION                              |           ROWS |
--------------------------------------------------------------------------------
|    0  |  SELECT FOR UPDATE STATEMENT                   |              1 |
```

```
|    1  |    QUERY BLOCK ("$QB_IDX_2")                    |              0 |
|    2  |      DML CLUSTER                                 | REMOTE ONLY  1
|
|    3  |        INDEX ACCESS ("PART", "PART_PK_INDEX") | (    0)      0 |
==========================================================================


     1  -  TARGET : PART.P_NAME
     2  -  FETCH
           Fetch SQL : SELECT /*+ INDEX( _A1, "PUBLIC"."PART_PK_INDEX" )
*/ "_A1"."$PHYSICAL_ROWID", "_A1"."P_NAME" FROM "PUBLIC"."PART"@LOCAL AS
"_A1" WHERE "_A1"."P_PARTKEY" = :_V0 FOR UPDATE OF "_A1"."P_PARTKEY"
           Non-Fetch SQL : SELECT /*+ INDEX( _A1,
"PUBLIC"."PART_PK_INDEX" ) */  NULL FROM "PUBLIC"."PART"@LOCAL AS "_A1"
WHERE "_A1"."P_PARTKEY" = :_V0 FOR UPDATE OF "_A1"."P_PARTKEY" WITHOUT
FETCH
           TARGET DOMAIN : G2(G2N1,G2N2) 1 rows
     3  -  HASH SHARD ( # 3 )
           READ INDEX COLUMN : PART.P_PARTKEY
           READ TABLE COLUMN : PART.P_NAME
             MIN RANGE : PART.P_PARTKEY = 5
             MAX RANGE : PART.P_PARTKEY = 5
           FETCH ONE ROW


<<<  end print plan
```

DML Cluster (SHARD KEY UPDATE)

Sharding key column的UPDATE根据操作数据前后记录所属的shard是否发生变更分为如下两种

- In-place update：操作数据前后记录所属的shard相同
- Out-place update：操作数据前后记录所属的shard不同

In-place update不移动记录而变更值不变更通过In-place update变更的记录的rowid信息

Out-place update删除原有记录后插入新的记录在通过out-place update变更的记录中设置新的rowid信息

SHARD KEY UPDATE分为in-place update和out-place update并构成generated query

In-place update的generated query由UPDATE语句构成Generated query拥有保障变更前的值所属的shard和变更后的值所属的shard一致的filter

Out-place update的generated query由SELECT语句和DELETE语句构成各个generated query拥有保障变更前的值所属的shard和变更后的值所属的shard不同的filter通过由SELECT语句构成的generated query收集变更前的数据并构成新的记录新的记录使用<基于global rowid的DML>进行insert通过DELETE语句删除之前的所有记录

DML cluster使用类型为SHARD KEY UPDATE时DML cluster的详细信息由如下in-place update的UPDATE SQL和out-place update的SELECT SQL & DELETE SQL构成

```
gSQL> \EXPLAIN PLAN UPDATE part SET p_partkey = p_partkey + 10;


5 rows updated.
```

```
>>>  start print plan


< Execution Plan >

================================================================================
|  IDX  |  NODE DESCRIPTION                          |            ROWS |
--------------------------------------------------------------------------------
|    0  |  UPDATE STATEMENT ("PART")                 |               5 |
|    1  |    QUERY BLOCK ("$QB_IDX_2")               |               0 |
|    2  |      DML CLUSTER                           |               5
|
|    3  |        UPDATE STATEMENT ("PART")           |               0
|
|    4  |          QUERY BLOCK ("$QB_IDX_2")         |               0 |
|    5  |            DML CLUSTER                     |               0 |
|    6  |              TABLE ACCESS ("PART" AS _A1)  |               0 |
|    7  |          QUERY BLOCK ("$QB_IDX_6")         |               0 |
|    8  |            INDEX ACCESS ("PART" AS _A1, ...) |               0 |
|    9  |        SELECT STATEMENT                    |               5
|
|   10  |          QUERY BLOCK ("$QB_IDX_2")         |               5 |
|   11  |            PLAN BASED CLUSTER              | LOCAL/REMOTE  5 |
|   12  |              TABLE ACCESS ("PART" AS _A1)  |               2 |
|   13  |        DELETE STATEMENT ("PART")           |               5
|
```

```
|   14  |          QUERY BLOCK ("$QB_IDX_2")          |              0 |
|   15  |            DML CLUSTER                      |              5 |
|   16  |              TABLE ACCESS ("PART" AS _A1)   |              2 |
|   17  |    QUERY BLOCK ("$QB_IDX_6")                |              0 |
|   18  |      TABLE ACCESS ("PART")                  |              0 |
=============================================================================


     1  -  TARGET : NOTHING
     2  -  SHARD KEY UPDATE
           UPDATE SQL : UPDATE /*+ FULL( _A1 ) */
"PUBLIC"."PART"@"G1N1"|"G1N2"|"G2N1"|"G2N2"|"G3N1"|"G3N2" AS "_A1" SET
( "_A1"."P_PARTKEY" ) = ( CAST( "_A1"."P_PARTKEY" + :_V0 AS NUMBER(10,
0) ) ) FROM "PUBLIC"."PART"@"G1N1"|"G1N2"|"G2N1"|"G2N2"|"G3N1"|"G3N2" AS
"_A1" WHERE SHARD_ID("PUBLIC"."PART", "_A1"."P_PARTKEY") =
SHARD_ID("PUBLIC"."PART", CAST( "_A1"."P_PARTKEY" + :_V0 AS NUMBER(10,
0) ))
           SELECT SQL : SELECT /*+ FULL( _A1 ) */ "_A1"."$PHYSICAL_ROWID",
"_A1"."P_PARTKEY", "_A1"."P_NAME", "_A1"."P_BRAND", "_A1"."P_TYPE",
"_A1"."P_SIZE", "_A1"."P_RETAILPRICE" FROM
"PUBLIC"."PART"@"G1N1"|"G1N2"|"G2N1"|"G2N2"|"G3N1"|"G3N2" AS "_A1" WHERE
SHARD_ID("PUBLIC"."PART", "_A1"."P_PARTKEY") <> SHARD_ID("PUBLIC"."PART",
CAST( "_A1"."P_PARTKEY" + :_V0 AS NUMBER(10, 0) ))
           DELETE SQL : DELETE /*+ FULL( _A1 ) */  "_A1" FROM
"PUBLIC"."PART"@"G1N1"|"G1N2"|"G2N1"|"G2N2"|"G3N1"|"G3N2" AS "_A1" WHERE
SHARD_ID("PUBLIC"."PART", "_A1"."P_PARTKEY") <> SHARD_ID("PUBLIC"."PART",
```

```
CAST( "_A1"."P_PARTKEY" + :_V0 AS NUMBER(10, 0) ))
     4  -  TARGET : NOTHING
     5  -  WITHOUT FETCH
           Non-Fetch SQL : UPDATE /*+ FULL( _A1 ) */ "PUBLIC"."PART"@LOCAL
AS "_A1" SET ( "_A1"."P_PARTKEY" ) = ( CAST( "_A1"."P_PARTKEY" + :_V0 AS
NUMBER(10, 0) ) ) FROM "PUBLIC"."PART"@LOCAL AS "_A1" WHERE
SHARD_ID("PUBLIC"."PART","_A1"."P_PARTKEY") =
SHARD_ID("PUBLIC"."PART",CAST( "_A1"."P_PARTKEY" + :_V1 AS NUMBER(10,
0) ))
           TARGET DOMAIN : G1(G1N1,G1N2) 0 rows, G2(G2N1,G2N2) 0 rows,
G3(G3N1,G3N2) 0 rows
     6  -  HASH SHARD ( # 3 )
           READ COLUMN : _A1.P_PARTKEY
             LOGICAL FILTER : SHARD_ID( "PUBLIC"."PART",_A1.P_PARTKEY) =
SHARD_ID( "PUBLIC"."PART",CAST( _A1.P_PARTKEY + :_V0 AS NUMBER(10, 0) ))
     7  -  TARGET : NOTHING
     8  -  HASH SHARD ( # 3 )
           READ INDEX COLUMN : _A1.P_PARTKEY
    10  -  TARGET : _A1.$PHYSICAL_ROWID, _A1.P_PARTKEY, _A1.P_NAME,
_A1.P_BRAND, _A1.P_TYPE, _A1.P_SIZE, _A1.P_RETAILPRICE
    11  -  SQL : SELECT /*+ FULL( _A1 ) */ "_A1"."$PHYSICAL_ROWID",
"_A1"."P_PARTKEY", "_A1"."P_NAME", "_A1"."P_BRAND", "_A1"."P_TYPE",
"_A1"."P_SIZE", "_A1"."P_RETAILPRICE" FROM "PUBLIC"."PART"@LOCAL AS "_A1"
WHERE SHARD_ID("PUBLIC"."PART","_A1"."P_PARTKEY") <>
SHARD_ID("PUBLIC"."PART",CAST( "_A1"."P_PARTKEY" + :_V0 AS NUMBER(10,
```

```
0) ))
           TARGET DOMAIN : G1(G1N1,G1N2) 2 rows, G2(G2N1,G2N2) 1 rows,
G3(G3N1,G3N2) 2 rows
    12  -  HASH SHARD ( # 3 )
           READ COLUMN : _A1.P_PARTKEY, _A1.P_NAME, _A1.P_BRAND,
_A1.P_TYPE, _A1.P_SIZE, _A1.P_RETAILPRICE
             LOGICAL FILTER : SHARD_ID( "PUBLIC"."PART",_A1.P_PARTKEY) <>
SHARD_ID( "PUBLIC"."PART",CAST( _A1.P_PARTKEY + :_V0 AS NUMBER(10, 0) ))
    14  -  TARGET : NOTHING
    15  -  WITHOUT FETCH
           Non-Fetch SQL : DELETE /*+ FULL( _A1 ) */  "_A1" FROM
"PUBLIC"."PART"@LOCAL AS "_A1" WHERE
SHARD_ID("PUBLIC"."PART","_A1"."P_PARTKEY") <>
SHARD_ID("PUBLIC"."PART",CAST( "_A1"."P_PARTKEY" + :_V0 AS NUMBER(10,
0) ))
           TARGET DOMAIN : G1(G1N1,G1N2) 2 rows, G2(G2N1,G2N2) 1 rows,
G3(G3N1,G3N2) 2 rows
    16  -  HASH SHARD ( # 3 )
           READ COLUMN : _A1.P_PARTKEY, _A1.P_BRAND
             LOGICAL FILTER : SHARD_ID( "PUBLIC"."PART",_A1.P_PARTKEY) <>
SHARD_ID( "PUBLIC"."PART",CAST( _A1.P_PARTKEY + :_V0 AS NUMBER(10, 0) ))
    17  -  TARGET : NOTHING
    18  -  HASH SHARD ( # 3 )
           READ COLUMN : PART.P_PARTKEY, PART.P_NAME, PART.P_BRAND,
PART.P_TYPE, PART.P_SIZE, PART.P_RETAILPRICE
```

```
<<<  end print plan
```

DML Cluster的join运算

包含子查询的语句与**包含子查询的join**相同可被查询处理器变更为join运算DML语句的子查询也可变更为join运算DML cluster plan node支持包含join的generated query构成

以下为执行包含子查询的执行DELETE的示例

```
gSQL> \EXPLAIN PLAN
      DELETE FROM part WHERE p_partkey IN ( SELECT ps_partkey FROM
partsupp );

3 rows deleted.

>>>  start print plan

< Execution Plan >
=========================================================================
|  IDX  |  NODE DESCRIPTION                          |            ROWS |
-------------------------------------------------------------------------
|    0  |  DELETE STATEMENT ("PART")                 |               3 |
|    1  |    QUERY BLOCK ("$QB_IDX_2")               |               0 |
|    2  |      DML CLUSTER                           |               3 |
|    3  |        HASH JOIN (SEMI)                    |               1 |
```

```
|    4  |          TABLE ACCESS ("PART")            |               2 |
|    5  |          HASH JOIN INSTANT (UNIQUE)       |               1 |
|    6  |            INDEX ACCESS ("PARTSUPP", ...) | (     2)      2 |
=========================================================================


     1  -  TARGET : NOTHING
     2  -  WITHOUT FETCH
           Non-Fetch SQL : DELETE /*+ KEEP_JOINED_TABLE USE_HASH_IN( _A2,
10 ) FULL( _A1 ) INDEX( _A2, "PUBLIC"."PARTSUPP_PK_INDEX" ) */  "_A1"
FROM ( "PUBLIC"."PART"@"G1N1"|"G1N2"|"G2N1"|"G2N2"|"G3N1"|"G3N2" AS "_A1"
SEMI JOIN "PUBLIC"."PARTSUPP"@"G1N1"|"G1N2"|"G2N1"|"G2N2"|"G3N1"|"G3N2" AS
"_A2" ON "_A2"."PS_PARTKEY" = "_A1"."P_PARTKEY") ALIAS "_A3"
           TARGET DOMAIN : G1(G1N1,G1N2) 1 rows, G2(G2N1,G2N2) 1 rows,
G3(G3N1,G3N2) 1 rows
     3  -  JOINED COLUMN : PART.$PHYSICAL_ROWID, PART.P_PARTKEY,
PART.P_BRAND
     4  -  HASH SHARD ( # 3 )
           READ COLUMN : PART.P_PARTKEY, PART.P_BRAND
     5  -  HASH KEY : PARTSUPP.PS_PARTKEY
           READ KEY COLUMN : PARTSUPP.PS_PARTKEY
             HASH FILTER : PARTSUPP.PS_PARTKEY = PART.P_PARTKEY
           FETCH ONE ROW
     6  -  HASH SHARD ( # 3 )
           READ INDEX COLUMN : PARTSUPP.PS_PARTKEY
```

```
<<<  end print plan
```

但DML cluster plan node不支持对汇集的数据的操作通过Generated query收集数据后需要操作数据的join运算不能构成DML cluster

以下为将包含子查询的DML变更为join运算但无法构成DML cluster的示例

```
gSQL> \EXPLAIN PLAN
      DELETE FROM supplier WHERE s_suppkey IN ( SELECT ps_partkey FROM
partsupp );

5 rows deleted.

>>>  start print plan

< Execution Plan >
================================================================================
=
|IDX|  NODE DESCRIPTION                                  |           ROWS
|
--------------------------------------------------------------------------------
-
| 0 |  DELETE STATEMENT ("SUPPLIER")                     |              5
|
| 1 |    QUERY BLOCK ("$QB_IDX_2")                       |              5
|
```

```
| 2 |      SINGLE CLUSTER                                | LOCAL/REMOTE 5
|
| 3 |        SELECT STATEMENT                            |              1
|
| 4 |          QUERY BLOCK ("$QB_IDX_2")                 |              1
|
| 5 |            NESTED JOIN (SEMI)                      |              1
|
| 6 |              INDEX ACCESS ("SUPPLIER" AS _A2, ...) | (     5)     5
|
| 7 |              INDEX ACCESS ("PARTSUPP" AS _A1, ...) | (     1)     1
|
================================================================================
=


     1  -  TARGET : NOTHING
     2  -  SQL : SELECT /*+ KEEP_JOINED_TABLE USE_NL_IN( _A1 ) INDEX( _A2,
"PUBLIC"."SUPPLIER_PK_INDEX" ) INDEX( _A1, "PUBLIC"."PARTSUPP_PK_INDEX" )
*/ "_A2"."$PHYSICAL_ROWID", "_A2"."S_SUPPKEY" FROM
( "PUBLIC"."SUPPLIER"@"G1N1"|"G1N2"|"G2N1"|"G2N2"|"G3N1"|"G3N2" AS "_A2"
SEMI JOIN "PUBLIC"."PARTSUPP"@"G1N1"|"G1N2"|"G2N1"|"G2N2"|"G3N1"|"G3N2" AS
"_A1" ON "_A1"."PS_PARTKEY" = "_A2"."S_SUPPKEY") ALIAS "_A3"
           TARGET DOMAIN : G1(G1N1,G1N2) 1 rows, G2(G2N1,G2N2) 2 rows,
G3(G3N1,G3N2) 2 rows
     4  -  TARGET : _A2.$PHYSICAL_ROWID, _A2.S_SUPPKEY
```

```
    5  -  JOINED COLUMN : _A2.$PHYSICAL_ROWID, _A2.S_SUPPKEY
    6  -  CLONED
          READ INDEX COLUMN : _A2.S_SUPPKEY
    7  -  HASH SHARD ( # 3 )
          READ INDEX COLUMN : _A1.PS_PARTKEY
            MIN RANGE : _A1.PS_PARTKEY = {_A2.S_SUPPKEY}
            MAX RANGE : _A1.PS_PARTKEY = {_A2.S_SUPPKEY}

<<<  end print plan
```

基于Global Rowid的DML

基于Global rowid的DML是使用rowid操作存储于不同cluster member的相同记录的方法添加新的记录或无法进行**基于Query的DML**时使用此方法

在集群环境提供的rowid作为赋予记录的逻辑识别信息是判断记录之间是否相同的标准一个记录存储于互不相同的多个集群成员时均拥有相同的rowid值生成记录时赋予rowid信息更新sharding key column的值时也会赋予新的值

rowid相关详细内容参考**ROWID Pseudo Column**

使用rowid信息的数据操作分为rowid信息收集阶段和数据操作阶段进行处理操作两个以上的记录时反复执行各个记录的rowid信息收集阶段和数据操作阶段

- Rowid信息搜集阶段
  - 收集为添加或操作对象的记录的rowid信息
- 数据操作阶段

- 操作属于收集的rowid的cluster group的记录
- 操作master服务器的记录后依次操作slave服务器记录

使用Rowid信息的数据操作根据是否使用**Global Secondary Index（全局二级索引）**支持如下两种方法

- **不使用Global Secondary Index的基于global rowid的DML**
- **使用全局二级索引的基于global rowid的DML**

不使用Global Secondary Index的基于global rowid的DML

不使用global secondary index的基于global rowid的DML在不需要判断master server和slave server的变更对象记录是否为相同记录时使用INSERT等不需要参照原有记录而构成新的记录的情况属于此情况

利用不使用global secondary index的rowid的数据操作步骤为如下

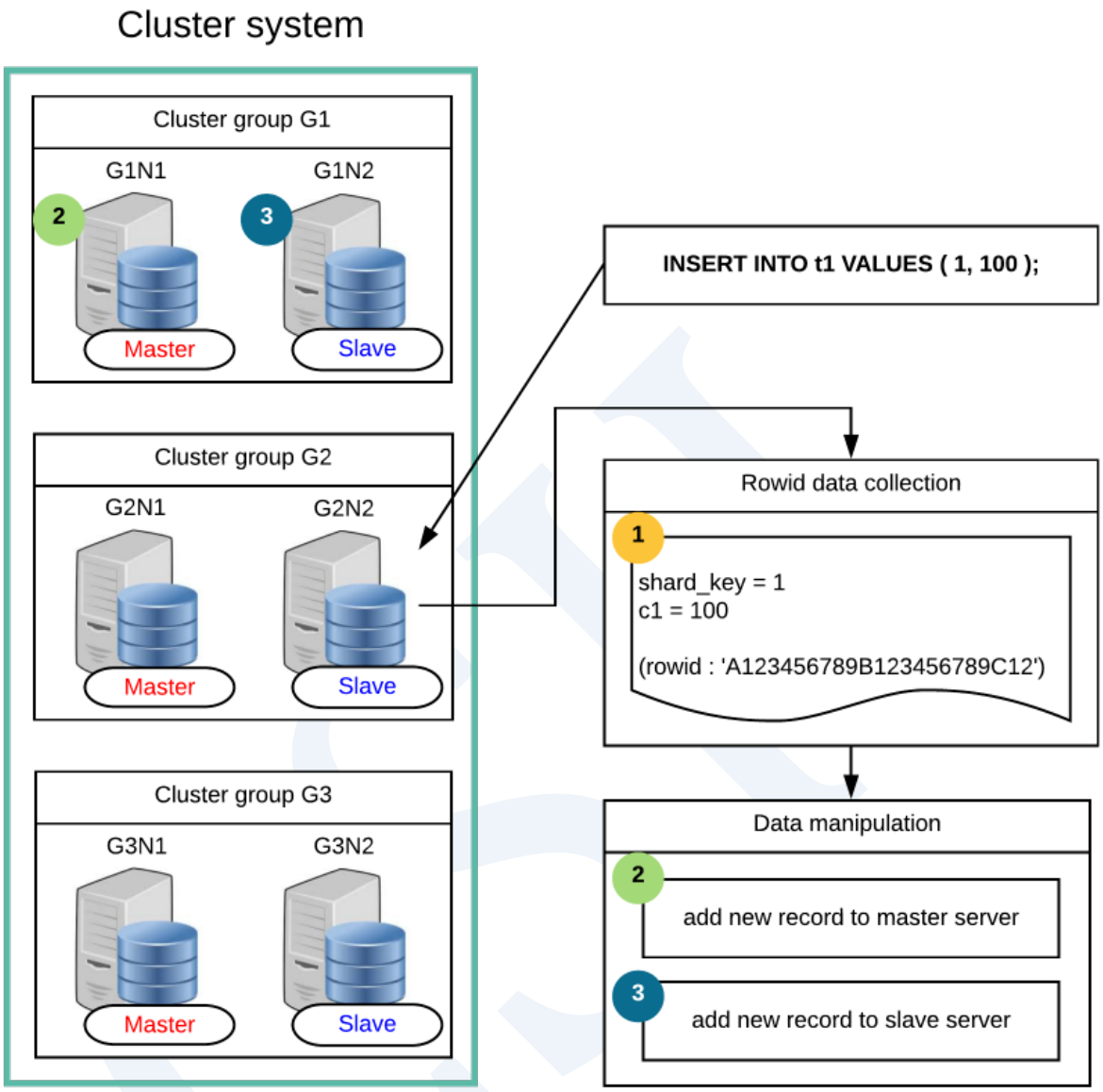


Figure 2-12 不使用global secondary index的基于global rowid的DML

增加记录时对新增记录赋予rowid并将各个记录存储于master服务器的适当位置将所有记录反映到master服务器后反映到slave服务器

如上图所示使用shard_key = 1的值构成sharding key column时可看到根据sharding策略shard_key为1的记录位于集群群组G1中因此仅在G1集群群组增加记录

以下为执行**不使用Global Secondary Index的基于global rowid的DML**的用户查询的结果执行

结果为无为了基于global rowid的DML而输出的信息

```
gSQL> \EXPLAIN PLAN INSERT INTO t1 VALUES ( 1, 100 );


1 row created.


>>>  start print plan


< Execution Plan >

=========================================================================

|  IDX  |  NODE DESCRIPTION                      |             ROWS |

-------------------------------------------------------------------------

|    0  |  INSERT VALUES STATEMENT ("T1")        |                1 |

|    1  |    QUERY BLOCK ("$QB_IDX_2")           |                0 |

|    2  |      TABLE ACCESS ("T1")               |                0 |

=========================================================================


     1  -  TARGET : NOTHING

     2  -  RANGE SHARD ( # 3 )

           READ COLUMN : T1.SHARD_KEY, T1.C1


<<<  end print plan
```

不使用Global secondary index的基于global rowid的DML支持如下查询类型

- **INSERT INTO**

- **INSERT INTO name RETURNING**
- **INSERT INTO name RETURNING .. INTO**

使用全局二级索引的基于global rowid的DML

包含在master服务器和slave服务器的复制record拥有相同的global rowid值

操作原有记录时需要使用global rowid信息保障master服务器和slave服务器的相同的数据进行相同的操作全局二级索引用于通过global rowid获取复制的记录的信息

利用使用全局二级索引的rowid的数据操作过程如下图所示

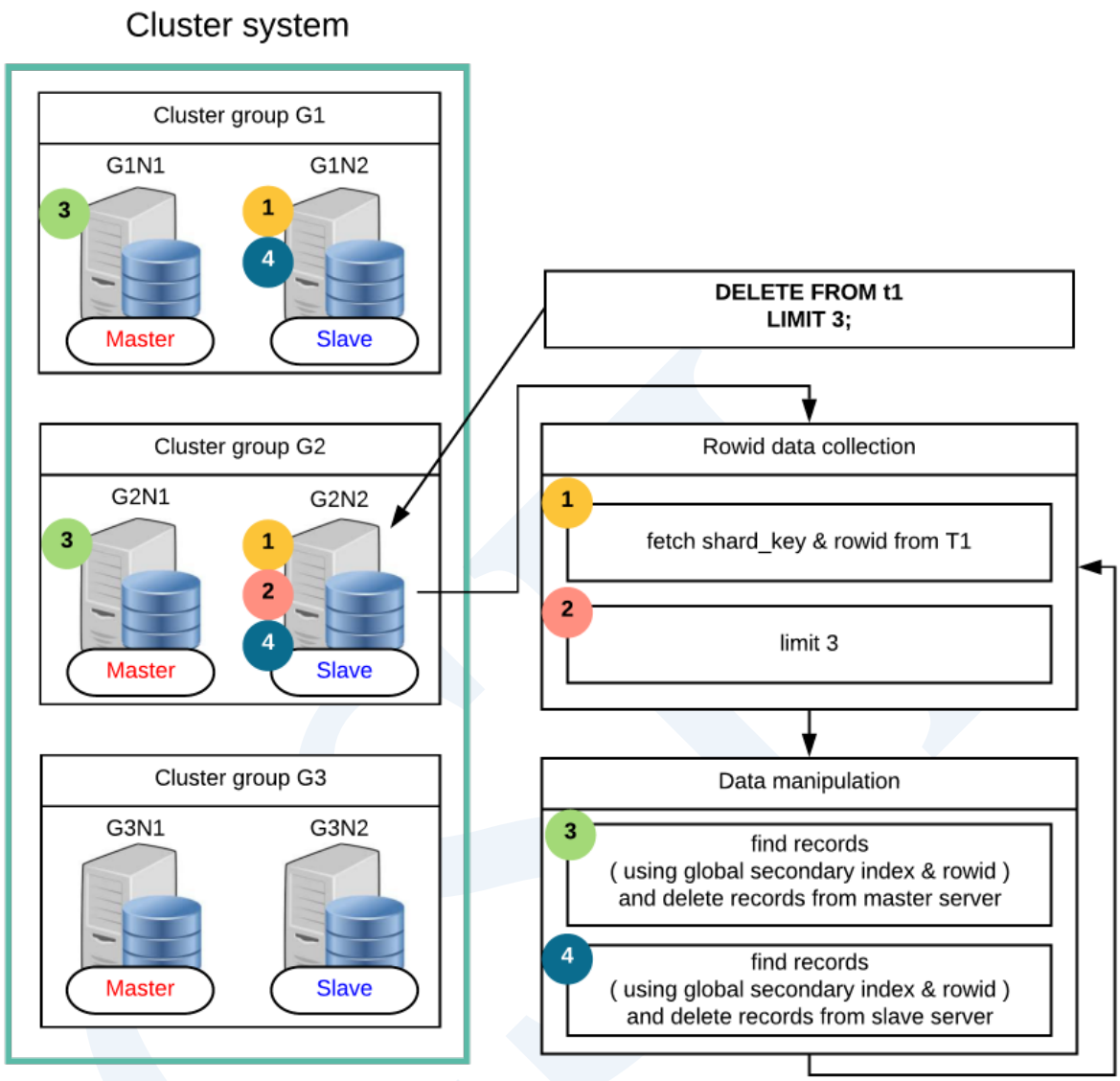


Figure 2-13 利用使用global secondary index的rowid的数据操作

在rowid信息收集阶段由接收到用户query的服务器收集数据操作对象记录的rowid和column值

在数据操作阶段以在rowid信息收集阶段选定的记录所属的集群群组为对象按照顺序操作master服务器与slave服务器的数据

在数据操作阶段负责收集rowid信息的服务器向master服务器与slave服务器传送rowid信息并请求更新数据接收到请求的服务器使用收到的rowid信息与全局二级索引查找存储于各个服务器

中的记录并操作数据

Rowid信息收集阶段与数据操作阶段持续反复执行直到无法再找到操作对象记录

以下为执行**利用使用global secondary index的rowid的数据操作**的用户查询的结果执行结果为无为了基于global rowid的DML而输出的信息

```
gSQL> \EXPLAIN PLAN DELETE FROM t1 LIMIT 3;


3 rows deleted.


>>>  start print plan


< Execution Plan >

=================================================================
|  IDX  |  NODE DESCRIPTION                |             ROWS |
-----------------------------------------------------------------
|    0  |  DELETE STATEMENT ("T1")         |                3 |
|    1  |    QUERY BLOCK ("$QB_IDX_2")     |                3 |
|    2  |      PLAN BASED CLUSTER          | LOCAL/REMOTE   3 |
|    3  |        TABLE ACCESS ("T1")       |                2 |
=================================================================


     1  -  TARGET : NOTHING

     2  -  SQL : SELECT /*+ FULL( _A1 ) */ "_A1"."$PHYSICAL_ROWID" FROM
"PUBLIC"."T1"@LOCAL AS "_A1"
```

```
           TARGET DOMAIN : G1(G1N1,G1N2) 2 rows, G2(G2N1,G2N2) 1 rows,
G3(G3N1,G3N2) 0 rows
     3  -  RANGE SHARD ( # 3 )
           READ COLUMN : NOTHING


<<<  end print plan
```

以下为支持使用global secondary index的基于global rowid的DML的query类型

- **SELECT .. FOR UPDATE**
- **SELECT .. INTO .. FOR UPDATE**
- **DELETE FROM**
- **DELETE FROM name RETURNING**
- **DELETE FROM name RETURNING .. INTO**
- **DELETE FROM name WHERE CURRENT OF cursor_name**
- **UPDATE**
- **UPDATE name RETURNING**
- **UPDATE name RETURNING .. INTO**
- **UPDATE name WHERE CURRENT OF cursor_name**

3.SQL Objects

本章节说明构成数据库的下列对象的概念与特征

- Authorization : User与privilege
- Schema
- Tablespace
- Table
- Index
- Sequence
- View
- Synonym
- Stored procedure
- Stored function

3.1 Database

数据库相关语句

详细内容参考如下

- 启动数据库: **ALTER SYSTEM {MOUNT | OPEN} DATABASE**
- 备份与恢复

  - **ALTER DATABASE BACKUP**
  - **ALTER DATABASE DELETE BACKUP**
  - **ALTER DATABASE RECOVER**
  - **ALTER DATABASE REGISTER**
  - **ALTER DATABASE RESTORE**
- 创建删除变更日志文件
  - **ALTER SYSTEM CHECKPOINT**
  - **ALTER SYSTEM SWITCH LOGFILE**
  - **ALTER DATABASE ARCHIVELOG**
  - **ALTER DATABASE ADD LOGFILE**
  - **ALTER DATABASE DROP LOGFILE**
  - **ALTER DATABASE RENAME LOGFILE**
- 对象的注释: **COMMENT ON name IS**
- 系统统计信息: **ANALYZE SYSTEM**

可通过以下视图查询数据库对象及其相关信息

对象集合	视图名称	说明
DICTIONARY_SCHEMA	**ALL_NONSCHEMA_COMMENTS**	用户可访问的non-schema对象的注释信息
INFORMATION_SCHEMA	**INFORMATION_SCHEMA_CATALOG_NAME**	数据库名称信息
	SQL_FEATURES	SUNDB的SQL标准兼容性信息

对象集合	视图名称	说明
	SQL_IMPLEMENTATION_INFO	SUNDB的SQL标准兼容性信息
	SQL_PACKAGES	SUNDB的SQL标准兼容性信息
	SQL_PARTS	SUNDB的SQL标准兼容性信息
	SQL_SIZING	SUNDB的SQL标准兼容性信息

Table 3-1 数据库对象相关信息

数据库构成对象

构成数据库的SQL对象

数据库由多个SQL object构成

数据库的SQL object根据是否属于SCHEMA分为SQL schema object与non-schema object

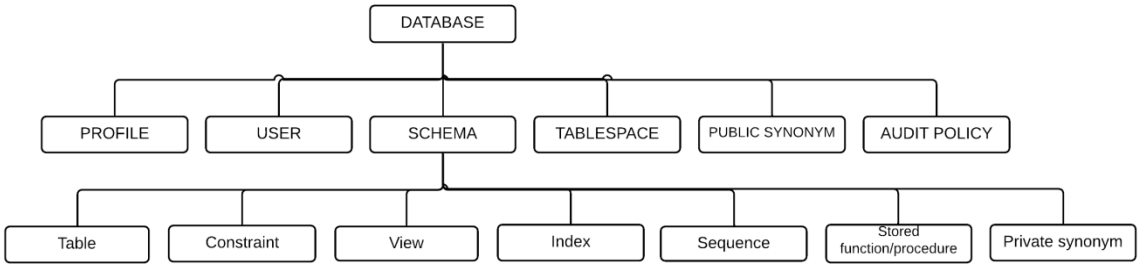


Figure 3-1 SQL objects

SQL schema object为属于SCHEMA的对象分为以下几种

- TABLE：由列与行组成的存储物理数据的对象
- VIEW：赋予查询的relation名访问方式与TABLE类似的逻辑对象
- INDEX：用于提高语句处理性能的对象
- SEQUENCE：生成序列号的对象
- CONSTRAINT：用于维持TABLE的完整性的对象
- SYNONYM：TABLEVIEWSEQUENCE其他SYNONYM的替代名
- STORED PROCEDURE：过程形式的持久存储模块（Persistent Stored Module）对象
- STORED FUNCTION：函数形式的持久存储模块对象

SQL schema object可以指定或省略schema名称省略时schema名解析为user的schema path

以下为指定schema名称创建object的示例

```
gSQL> CREATE TABLE my_schema.lineitem ( id INTEGER );

gSQL> CREATE INDEX my_schema.my_index ON my_schema.lineitem ( id );
```

Non-schema object是不属于SCHEMA的对象分为以下几种

- PROFILE：密码管理策略
- AUDIT POLICY：审计策略
- USER：用户
- SCHEMA：SQL schema object的逻辑位置
- TABLESPACE：SQL schema object的物理空间
- PUBLIC SYNONYM：无SCHEMA名的TABLEVIEWSEQUENCE其他SYNONYM的替代名

标准SQL明确规定SCHEMA对象的概念及语句但只有USER与DATABASE的概念而未定义语句标准SQL不定义TABLESPACE对象即标准SQL不明确规定Non-Schema对象的关系

SUNDB中USERSCHEMATABLESPACE定义为DATABASE的下层对象而其他数据库如下定义Non-Schema对象

- SUNDB
  - USER与SCHEMA为不同的对象
  - USER不拥有属于自己的SCHEMA或可以有多个SCHEMA
  - USER : SCHEMA = 1 : N的关系
- Oracle
  - USER与SCHEMA定义为类似的概念
  - USER : SCHEMA = 1 : 1的关系
- DB2
  - USER不是DATABASE的下层对象
  - USER : SCHEMA = 1 : N的关系
- Postgres
  - USER不是DATABASE的下层对象
  - USER : SCHEMA = 1 : N的关系

- MySQL
  - USER与SCHEMA定义为类似的概念
  - USER为DATABASE(SCHEMA)的下层对象

对象的Name Space

每个数据库的对象拥有可识别的名称

在schema内SQL schema object名称不重复

如下在不同schema中可生成相同名称的lineitem表对象

```
gSQL> CREATE TABLE my_schema.lineitem ( id INTEGER );
gSQL> CREATE TABLE your_schema.lineitem ( name VARCHAR(128) );
```

SQL schema object在单个schema内有如下name space

- TABLE, VIEW, SEQUENCE, PRIVATE SYNONYM, STORED PROCEDURE, STORED FUNCTION
- INDEX
- CONSTRAINT

即无法生成相同名称的表（table）与视图（view）但可以生成相同名称的表（table）与索引（index）

- 无法生成相同名称的table与view

```
gSQL> CREATE TABLE my_relation ( id INTEGER );
gSQL> CREATE VIEW my_relation ( name ) AS SELECT name FROM tmp_relation;
```

- 可以生成相同名称的table与index

```
gSQL> CREATE TABLE my_object ( id INTEGER );

gSQL> CREATE INDEX my_object ON my_table ( name );
```

Non-Schema object在数据库内拥有可识别的名称相关name space如下

- PROFILE
- AUDIT POLICY
- USER
- SCHEMA
- TABLESPACE

即无法生成相同名称的USER但可以生成相同名称的USER与SCHEMA

- 生成my_name USER对象与my_name SCHEMA对象

```
gSQL> CREATE USER my_name IDENTIFIED BY my_name WITH SCHEMA my_name;
```

Built-in对象

SUNDB在创建数据库时自动创建系统运行所需的userschematablespace对象

Built-in User

创建数据库时自动创建以下账号除TEST用户外无法删除内置（built-in）账号

- _SYSTEM账号
  - 在系统内部使用的账号是创建数据库时的built-in对象的所有者是用户创建对象时给对象所有者赋予权限的授予者（grantor）
- SYS账号
  - 管理数据库的用户
- TEST账号
  - 为了测试而创建的用户可删除的对象
- ADMIN账号
  - 管理数据库的角色
- SYSDBA账号
  - 启动/关闭数据库的角色
- PUBLIC账号
  - 代表所有账号用于管理所有用户的权限

Built-in Schema

创建数据库时自动创建以下SCHEMA所有built-in SCHEMA均不可删除

- DEFINITION_SCHEMA
  - 由存储数据库所有对象信息的物理表构成
- FIXED_TABLE_SCHEMA
  - 由以表的形式显示系统资料结构信息的固定表（fixed table）构成
- DICTIONARY_SCHEMA
  - 由用于查询数据库对象信息的用户角度的视图构成
- INFORMATION_SCHEMA

- 由用于查询数据库对象信息的标准SQL定义的视图构成

- PERFORMANCE_VIEW_SCHEMA
  - 由用于查看系统状态的用户角度的视图构成
- SESSION_SCHEMA
  - 作为用于管理以会话为单位的对象的系统模式user无法使用
- PUBLIC
  - 作为所有用户都可创建对象的SCHEMA与表示所有用户的PUBLIC账号不同

Built-in Tablespace

创建数据库时自动创建如下表空间所有built-in表空间均不能删除

- DICTIONARY_TBS
  - 存储用于管理SQL对象信息的dictionary table
- MEM_UNDO_TBS
  - 存储还原段（Undo segment）与事务信息
- MEM_DATA_TBS
  - 最初创建的memory data tablespace
  - data tablespace存储用户创建的table等对象
- MEM_TEMP_TBS
  - 最初创建的temporary tablespace
  - temporary tablespace存储语句处理中创建的sort或hash等临时对象
- DISK_DATA_TBS
  - 最初创建的磁盘数据表空间
  - 数据表空间存储用户创建的表等对象
- MEM_AUX_TBS

- 是System auxiliary tablespace存储audit record等自动创建的记录

- MEM_TRANS_TBS
  - 在集群系统中有效管理全局事务信息
  - 不在单机版创建

Built-in Profile

创建数据库时自动创建以下"DEFAULT" profile自动创建的"DEFAULT" profile的密码参数（password parameter）信息如下

parameter	value
FAILED_LOGIN_ATTEMPTS	10
PASSWORD_LOCK_TIME	1
PASSWORD_LIFE_TIME	180
PASSWORD_GRACE_TIME	7
PASSWORD_REUSE_MAX	UNLIMITED
PASSWORD_REUSE_TIME	UNLIMITED
PASSWORD_VERIFY_FUNCTION	NULL

Table 3-2 DEFAULT profile的构成

"DEFAULT" profile的默认值有以下特点

- 锁定账号

    - 连续10次登录失败时(FAILED_LOGIN_ATTEMPTS)锁定账号1天(PASSWORD_LOCK_TIME)
- 密码过期
    - 经过180天(PASSWORD_LIFE_TIME)后再过7天(PASSWORD_GRACE_TIME)的宽限期后密码过期
- 是否可重新使用密码
    - 可以重新使用之前的密码
- 检查密码复杂度
    - 不检查

3.2 Profile

Profile相关语句

详细内容参考如下链接

- 生成profile：**CREATE PROFILE**
- 删除profile：**DROP PROFILE**
- 变更profile：**ALTER PROFILE**
- 给用户设置profile：**CREATE USERALTER USER**
- 清除密码历史记录：**ALTER DATABASE CLEAR PASSWORD HISTORY**

对象集合	视图	说明
DICTIONARY_SCHEMA	**DBA_PROFILES**	所有profile信息
	DBA_USERS	用户的profile信息

Table 3-3 Profile对象相关信息

Profile概念

SUNDB为了数据库的安全执行用户认证此时使用密码为了防止密码被盗伪造错误使用需要密码管理策略

Profile包含此类密码管理策略信息DBA或安全管理员把profile分配给用户并应用符合用户角色的密码管理策略

Profile的创建变更及分配

通过CREATE PROFILE语句创建profile

```
CREATE PROFILE profile1 LIMIT
       FAILED_LOGIN_ATTEMPTS     10
       PASSWORD_LOCK_TIME        1
       PASSWORD_LIFE_TIME        180
       PASSWORD_GRACE_TIME       7
       PASSWORD_REUSE_MAX        UNLIMITED
       PASSWORD_REUSE_TIME       UNLIMITED
       PASSWORD_VERIFY_FUNCTION  NULL
```

通过CREATE USER或ALTER USER语句分配profile

```
CREATE USER u1 IDENTIFIED BY u1 PROFILE profile1;
ALTER USER u2 PROFILE profile1;
```

通过ALTER PROFILE语句更改profile的参数

```
ALTER PROFILE profile1 LIMIT
      PASSWORD_REUSE_MAX        3
      PASSWORD_REUSE_TIME       30;
```

如果不给用户分配profile则用户创建及使用密码不受任何约束

给用户分配创建的profile或DEFAULT profile后用户创建及使用密码时将遵循对应profile的密码管理策略

DEFAULT profile的密码设置

给用户分配DEFAULT profile时以如下方式管理密码

Parameter	Default setting	Description
FAILED_LOGIN_ATTEMPS	10	允许的登录失败次数 连续登录失败超过10次时账号将被锁定
PASSWORD_LOCK_TIME	1	账号锁定期限 超过允许的连续登陆失败次数后账号将被锁定1天
PASSWORD_LIFE_TIME	180	密码的生命周期 180天后密码过期
PASSWORD_GRACE_TIME	7	密码过期后需要变更密码的期限 从密码过期后第一次登录起7天之内需要变更密码如果在此期间不更改密码则不能用该密码登陆
PASSWORD_REUSE_MAX	UNLIMITED	密码不可重新使用的次数 PASSWORD_REUSE_MAX与PASSWORD_REUSE_TIME需同时设置两个参数均为UNLIMITED时之前的密码一直可以重新使用

Parameter	Default setting	Description
PASSWORD_REUSE_TIME	UNLIMITED	密码不能重新使用的期限

Table 3-4 密码的DEFAULT Profile

锁定账号

连续登陆失败超过FAILED_LOGIN_ATTEMPTS指定的次数时账号将锁定与PASSWORD_LOCK_TIME指定的时间段相同的期限

```
CREATE PROFILE profile1 LIMIT
       FAILED_LOGIN_ATTEMPTS     10
       PASSWORD_LOCK_TIME        1;


ALTER USER u1 PROFILE profile1;
```

u1用户连续登陆失败10次以上时账号将被锁定1天1天后账号自动被解锁

不指定PASSWORD_LOCK_TIME时默认为Default profile的PASSWORD_LOCK_TIME指定的值

PASSWORD_LOCK_TIME为UNLIMITED时不会自动解锁因此需要通过以下语句手动解锁

```
ALTER USER u1 ACCOUNT UNLOCK;
```

登陆成功后登陆失败次数将被初始化为0

安全管理员可以锁定用户此时不能自动解锁需要安全管理员手动解锁

```
ALTER USER u1 ACCOUNT LOCK;

ALTER USER u1 ACCOUNT UNLOCK;
```

密码的生命周期

PASSWORD_LIFE_TIME为密码的生命周期过了这段时间后密码将过期

密码过期后用户DBA或安全管理员需要更改密码

```
CREATE PROFILE profile1 LIMIT
       PASSWORD_LIFE_TIME        180
       PASSWORD_GRACE_TIME       7;


ALTER USER u1 PROFILE profile1;
```

180天后从u1用户第一次登陆时开始计算宽限期（grace time）

7天的宽限期内用户每次登陆时提醒更改密码直到设置新的密码

如果7天的宽限期后还未更改密码设置新的密码之前将被拒绝访问

通过CREATE USER或ALTER USER语句可以做密码过期处理

```
ALTER USER u1 PASSWORD EXPIRE;
```

密码过期后登录时如下报 (ERR-28000(16312) the password has expired)密码过期错误需要输入新的密码

```
% gsql u1 u1


ERR-28000(16312): the password has expired


Changing password for u1
New password:
Retype new password:
Connected to SUNDB Database.


gSQL>
```

密码的重新使用

如果要重新使用相同的密码需按照PASSWORD_REUSE_MAX中指定的次数使用其他密码并经过PASSWORD_REUSE_TIME指定的时间后才能重新使用相同的密码

```
CREATE PROFILE profile1 LIMIT
       PASSWORD_REUSE_MAX        2
       PASSWORD_REUSE_TIME      10;


ALTER USER u1 PROFILE profile1;
```

u1用户为了重新使用当前密码需变更2次密码并经过10天的时间

```
ALTER USER u1 IDENTIFIED BY u1 REPLACE u1;
ALTER USER u1 IDENTIFIED BY u1 REPLACE u1
```

```
*
ERROR at line 1:
ORA-28007: the password cannot be reused

ALTER USER u1 IDENTIFIED BY u2 REPLACE u1;

User altered.

ALTER USER u1 IDENTIFIED BY u3 REPLACE u2;

User altered.
```

- 10天后

```
ALTER USER u1 IDENTIFIED BY u1 REPLACE u3;

User altered.
```

同时满足两个条件时才可重新使用PASSWORD_REUSE_MAX与PASSWORD_REUSE_TIME两个值中的一个为UNLIMITED时无法重新使用密码

两个参数均为UNLIMITED时可以重新使用

PASSWORD_REUSE_MAX	PASSWORD_REUSE_TIME	可重复使用的条件
Integer value	Integer value	同时满足两个条件时可重新使用

PASSWORD_REUSE_MAX	PASSWORD_REUSE_TIME	可重复使用的条件
Integer value	UNLIMITED	不可重新使用
UNLIMITED	Integer value	不可重新使用
UNLIMITED	UNLIMITED	总是可以重复使用

Table 3-5 重新使用密码

检查密码复杂度

检查密码复杂度验证是否为系统入侵者容易猜测的密码

SUNDB提供以下检查密码复杂度的方法

复杂度检查方法	复杂度检查内容
KISA_VERIFY_FUNCTION	• 8个字符以上 • 1个以上的字母 • 1个以上的数字 • 1个以上的特殊符号

复杂度检查方法	复杂度检查内容
ORA12C_VERIFY_FUNCTION	• 8个字符以上 • 1个以上的字母 • 1个以上的数字 • 不能包含数据库名称 • 不能包含用户名或倒序的用户名 • 不能包含'sundb' • 不能包含'oracle' • 不能使用如下简单的密码 ○ welcome1, database1, account1, user1234, password1, oracle123, computer1, abcdefg1, change_on_intall • 与之前的密码至少有3个不同的字母
ORA12C_STRONG_VERIFY_FUNCTION	• 9个字符以上 • 2个以上的大写字母 • 2个以上的小写字母 • 2个以上的数字 • 2个以上的特殊符号 • 与之前的密码至少有4个不同的字符

复杂度检查方法	复杂度检查内容
VERIFY_FUNCTION_11G	• 8个字符以上 • 1个以上的字母 • 1个以上的数字 • 不能包含用户名 • 与之前的密码至少有3个不同的字符
VERIFY_FUNCTION	• 不能与用户名相同 • 4个字符以上 • 1个以上的字母 • 1个以上的数字 • 1个以上的特殊符号 • 不能使用如下简单的密码 ○ welcome, database, account, user, password, oracle, computer, abcd • 与之前的密码至少有3个不同的字符

Table 3-6　检查密码复杂度

Profile和用户设置相关的详细内容参考**CREATE PROFILE**和**CREATE USER**

3.3 Audit Policy

Audit Policy相关语句

详细内容参考如下

- 创建audit policy: **CREATE AUDIT POLICY**
- 删除audit policy: **DROP AUDIT POLICY**
- 变更audit policy: **ALTER AUDIT POLICY**
- 激活audit policy: **AUDIT POLICY**
- 禁用audit policy: **NOAUDIT POLICY**
- 删除audit trail: **ALTER DATABASE CLEAR AUDIT TRAIL**

对象集合	视图	说明
DICTIONARY_SCHEMA	**AUDIT_POLICIES**	所有audit policy信息
	AUDIT_POLICY_OPTIONS	Audit policy选项信息
	AUDIT_POLICY_ENABLED	Audit policy激活信息

Table 3-7 Audit policy对象相关信息

使用示例

执行如下示例需要有AUDIT SYSTEM ON DATABASE权限

创建Audit Policy

通过执行**CREATE AUDIT POLICY**语句创建audit policy对象

以下语句是为了审计u1.t1表的DML而创建audit_t1_dml对象的示例

```
CREATE AUDIT POLICY audit_t1_dml
       ACTIONS INSERT ON u1.t1
             , DELETE ON u1.t1
             , UPDATE ON u1.t1
;

Audit policy created.
```

查询**AUDIT_POLICY_OPTIONS**视图（view）查看audit policy的选项信息

```
SELECT policy_name
     , audit_option
     , object_schema
     , object_name
  FROM audit_policy_options
 WHERE policy_name = 'AUDIT_T1_DML'
 ORDER BY audit_option
;

POLICY_NAME  AUDIT_OPTION OBJECT_SCHEMA OBJECT_NAME
------------ ------------ ------------- -----------
```

```
AUDIT_T1_DML DELETE       U1            T1

AUDIT_T1_DML INSERT       U1            T1

AUDIT_T1_DML UPDATE       U1            T1


3 rows selected.
```

激活Audit Policy

使用**AUDIT POLICY**语句激活audit policy

以下语句为除了u1sys以外的用户成功执行u1.t1表的DML时记录audit record而激活audit policy的示例

```
AUDIT POLICY audit_t1_dml
      EXCEPT u1, sys
      WHENEVER SUCCESSFUL
;
```

被激活的audit policy适用于新创建的会话不会对已有会话产生影响

通过查询**AUDIT_POLICY_ENABLED**视图可查看audit policy的激活信息

```
SELECT policy_name
     , enabled_opt
     , user_name
     , when_success
     , when_failure
```

```
  FROM audit_policy_enabled
 WHERE policy_name = 'AUDIT_T1_DML'
 ORDER BY user_name
;

POLICY_NAME             ENABLED_OPT           USER_NAME  WHEN_SUCCESS
WHEN_FAILURE
------------------- -------------------- ---------- ------------ -------
--
AUDIT_T1_DML         EXCEPT                SYS        YES          NO
AUDIT_T1_DML         EXCEPT                U1         YES          NO

2 rows selected.
```

激活audit policy后符合条件的action创建audit record

以下为u2用户全部成功执行SQL语句的示例

```
SELECT COUNT(*) FROM u1.t1;
INSERT INTO u1.t1 VALUES ( 1 );
UPDATE u1.t1 SET id = id + 1 WHERE id = 1;
DELETE u1.t1 WHERE id = 2;
COMMIT;
```

由于上述示例中的INSERTUPDATEDELETE是审计对象action因此创建audit record而

SELECTCOMMIT不是审计对象action因此不创建audit record

查询Audit Trail

为了查询audit recordAUDIT_TRAIL view需要有SELECT ON DICTIONARY_SCHEMA.AUDIT_TRAIL权限

以下为查询audit_t1_dml审计策略创建的audit trail的示例

```
SELECT policy_name
     , logon_username
     , action_name
     , object_schema
     , object_name
     , sql_text
  FROM audit_trail
 WHERE policy_name = 'AUDIT_T1_DML'
;


POLICY_NAME  LOGON_USERNAME ACTION_NAME OBJECT_SCHEMA OBJECT_NAME SQL_TEXT
------------ -------------- ----------- ------------- ----------- --------
---------------------------------
AUDIT_T1_DML U2             INSERT      U1            T1          INSERT
INTO u1.t1 VALUES ( 1 )
AUDIT_T1_DML U2             UPDATE      U1            T1          UPDATE
u1.t1 SET c1 = c1 + 1 WHERE c1 = 1
AUDIT_T1_DML U2             DELETE      U1            T1          DELETE
u1.t1 WHERE c1 = 2
```

```
3 rows selected.
```

删除Audit Trail

若激活audit policyaudit trail的大小将持续增加

通过执行以下语句删除audit trail

```
ALTER DATABASE CLEAR AUDIT TRAIL;
```

存储audit trail需要如下存储到用户表后删除

- 创建用户表

```
CREATE TABLE my_audit_trail
AS SELECT *
     FROM audit_trail
     WITH NO DATA;
```

- 保存在用户表后删除

```
INSERT INTO my_audit_trail SELECT * FROM audit_trail;
ALTER DATABASE CLEAR AUDIT TRAIL;
```

同时查询保存的audit trail与当前audit trail时如下创建视图并进行管理

```
CREATE VIEW unified_audit_trail
```

```
AS
SELECT * FROM my_audit_trail
 UNION ALL
SELECT * FROM dictionary_schema.audit_trail
;
```

禁用Audit Policy

使用下列语句禁用audit policy

```
NOAUDIT POLICY audit_t1_dml;
```

禁用audit policy只对新会话产生影响不影响原有会话的激活信息

删除Audit Policy

执行下列语句删除audit policy

```
DROP AUDIT POLICY audit_t1_dml;
```

Audit policy处于禁用状态时才可删除删除后不影响原有会话

Audit Policy概念

Audit Trail

查询Audit Trail

Audit record可使用DICTIONARY_SCHEMA.AUDIT_TRAIL view进行查询

查询AUDIT_TRAIL view需要有SELECT权限

```
GRANT SELECT ON DICTIONARY_SCHEMA.AUDIT_TRAIL TO user_name;
```

AUDIT_TRAIL view拥有以下信息

信息	Column名称	说明
Session information	MEMBER_NAME	Cluster member name
	SESSION_ID	Session identifier
	SESSION_SERIAL	Session serial number
	LOGON_USERNAME	Logon user name of the user whose axtions were audited
	CURRENT_USERNAME	Effective user for the statement execution
	SERVER_PROCESS	Server process identifier for the session
Peer client	CLIENT_PROGRAM_NAME	Client program used for session

信息	Column名称	说明
information	CLIENT_USERNAME	Client operating system user name for the session
	CLIENT_PROCESS	Client process identifier for the session
	CLIENT_HOST	Client host ip address for the session
	CLIENT_PORT	Client port number for the session
	CLIENT_TERMINAL	Client terminal name for the session
SQL information	TRANSACTION_ID	Transaction identifier
	SCN	System change number (SCN) string of the query at the time of the event
	GCN	Global change number (GCN) of the query at the time of the event
	DCN	Domain change number (DCN) of the query at the time of the event
	LCN	Local change number (LCN) of the query at the time of the event
	STMT_NO	Numeric number for each statement run in a session
	SQL_TEXT	SQL associated with the event
	SQL_BINDS	List of bind variables, if any, associated with SQL_TEXT

信息	Column名称	说明
	RETURN_CODE	Error code generated by the action, zero if the action succeeded
	ERROR_MESSAGE	Error message generated by the action, null if the action succeeded
Event information	ENTRY_ID	Audit trail entry identifier in the session
	EVENT_TIMESTAMP	Timestamp of the creation of the audit trail entry in local time zone
	POLICY_NAME	Audit policy name that caused the current audit record
	PRIVILEGE_USED	Database privilege used to execute the action
	ACTION_NAME	Action name executed by the user
	OBJECT_TYPE	Object type of object affected by the action
	OBJECT_SCHEMA	Schema name of object affected by the action
	OBJECT_NAME	Object name of object affected by the action

Table 3-8 Column信息

保存Audit Trail

构成AUDIT_TRAIL view的table如下

- AUDIT_TRAIL_SESSION

    - 每个session记录一个record
    - Session information
    - Peer client information
- AUDIT_TRAIL_SQL
    - 每个SQL记录一个record
    - SQL information
- AUDIT_TRAIL_EVENT
    - 每个audit option记录一个record
    - Audit event information

构成audit trail的audit record分开保存到多个表中

表的schema为DEFINITION_SCHEMA，保存到MEM_AUX_TBS表空间中

- DEFINITION_SCHEMA
    - 保存dictionary table的schema
- MEM_AUX_TBS
    - System auxiliary tablespace
    - 用于存储数据库自动创建的record的表空间
    - 为了不让自动创建的record影响服务运行使用另外的表空间进行管理

创建Audit Record

激活audit policy产生符合条件的action时创建audit record

产生多个符合审计条件的action时创建一个以上的audit record

- 如下罗列类似的audit option时创建一个audit record

- 定义audit policy

```
CREATE AUDIT POLICY p1
       PRIVILEGES INSERT ANY TABLE
       ACTIONS INSERT;

AUDIT POLICY p1;
```

- 执行audit action

```
INSERT INTO other_user.t1 VALUES ( 1 );
```

- 如下罗列互不相同的audit option时创建两个audit record
  - 定义audit policy

```
CREATE AUDIT POLICY p1
       ACTIONS SELECT ON u1.t1
             , SELECT ON u1.t2;

AUDIT POLICY p1;
```

- 执行audit action

```
SELECT COUNT(*) FROM u1.t1 A, u1.t2 B WHERE A.id = B.id;
```

- 如下在相同的action激活多个audit policy时创建两个audit record
  - 定义audit policy

```
CREATE AUDIT POLICY p1
       PRIVILEGES INSERT ANY TABLE;
AUDIT POLICY p1;

CREATE AUDIT POLICY p2
       ACTIONS INSERT;
AUDIT POLICY p2;
```

- 执行audit action

```
INSERT INTO other.t1 VALUES (1);
```

删除（purge） Audit Trail

执行下列语句删除audit trail

```
ALTER DATABASE CLEAR AUDIT TRAIL;
```

如下创建用户表存储之前的audit record

```
CREATE TABLE my_audit_trail AS SELECT * FROM audit_trail WITH NO DATA;
```

在周期性地删除audit trail之前使用INSERT .. SELECT语句进行保存

```
INSERT INTO my_audit_trail SELECT * FROM audit_trail;

ALTER DATABASE CLEAR AUDIT TRAIL;
```

使用EVENT_TIMESTAMP column执行DELETE语句保存特定期间的audit record

```
INSERT INTO my_audit_trail SELECT * FROM audit_trail;


DELETE FROM my_audit_trail WHERE event_timestamp < ADD_MONTHS( sysdate, -
3 );


ALTER DATABASE CLEAR AUDIT TRAIL;
```

创建如下视图同时查询之前的audit record与当前audit record

```
CREATE VIEW audit_trail_view
AS SELECT * FROM dictionary_schema.audit_trail
   UNION ALL
   SELECT * FROM my_audit_trail;


SELECT * FROM audit_trail_view;
```

设置Audit Policy

可包含在audit policy中的选项如下

- Privilege auditing
  - 使用database privilege审计SQL执行
- Object action auditing
  - 审计特定对象的SQL执行

- System action auditing
    - 审计所有对象的SQL执行

虽然可创建并管理多个audit policy但建议用少数audit policy管理多个audit option

激活的audit policy信息在logon时间点构成会话信息因此audit policy的数量越少负荷越小

另外激活多个audit policy时由于对单个SQL语句判断是否创建audit record而产生创建多个audit record的负荷

在Logon时间点在会话中构成的audit policy信息不受audit policy的删除变更激活禁用等的影响

变更Audit policy仅适用于新logon的会话

Privilege Auditing

当使用database privilege成功执行SQL语句时privilege auditing设置为审计

作为数据库所有者的sys用户不创建根据privilege auditing的audit record

为了Privilege auditing可罗列的database privilege可通过查询 **V$AUDITABLE_DB_PRIVILEGES** 视图查看

```
SELECT privilege_name FROM v$auditable_db_privileges;


PRIVILEGE_NAME

----------------------

ADMINISTRATION

ALTER DATABASE

ALTER SYSTEM

ACCESS CONTROL
```

```
CREATE USER

ALTER USER

DROP USER

CREATE TABLESPACE

ALTER TABLESPACE

DROP TABLESPACE

USAGE TABLESPACE

CREATE SCHEMA

DROP SCHEMA

ANALYZE ANY

CREATE ANY TABLE

ALTER ANY TABLE

DROP ANY TABLE

SELECT ANY TABLE

INSERT ANY TABLE

DELETE ANY TABLE

UPDATE ANY TABLE

LOCK ANY TABLE

CREATE ANY VIEW

DROP ANY VIEW

CREATE ANY SEQUENCE

ALTER ANY SEQUENCE

DROP ANY SEQUENCE

USAGE ANY SEQUENCE

CREATE ANY INDEX
```

```
ALTER ANY INDEX

DROP ANY INDEX

CREATE ANY SYNONYM

DROP ANY SYNONYM

CREATE PUBLIC SYNONYM

DROP PUBLIC SYNONYM

CREATE PROFILE

ALTER PROFILE

DROP PROFILE

CREATE ANY PROCEDURE

ALTER ANY PROCEDURE

DROP ANY PROCEDURE

EXECUTE ANY PROCEDURE

AUDIT SYSTEM

PURGE DBA_RECYCLEBIN


44 rows selected.
```

以下为拥有SELECT ANY TABLE权限的用户u1为了privilege auditing而激活audit policy的示例

```
CREATE AUDIT POLICY p1
       PRIVILEGES SELECT ANY TABLE;


AUDIT POLICY p1;
```

以下为用户u1执行如下两个语句时是否创建audit record

- SELECT * FROM u1.t1
  - 由于是对象的所有者因此不使用SELECT ANY TABLE权限
  - 无Audit record
- SELECT * FROM other.t1
  - 无SELECT ON other.t1权限但可通过SELECT ANY TABLE权限成功执行
  - 创建Audit record

查看Privilege auditing信息需如下查询AUDIT_POLICY_OPTIONS view

```
SELECT audit_option
     , audit_option_type
     , object_schema
     , object_name
  FROM audit_policy_options
 WHERE policy_name = 'P1'
;

AUDIT_OPTION         AUDIT_OPTION_TYPE    OBJECT_SCHEMA     OBJECT_NAME
-------------------- -------------------- ----------------- ------------
SELECT ANY TABLE     DATABASE PRIVILEGE   null              null
```

Object Action Auditing

审计对特定对象执行的SQL

各对象类型可审计的action如下

Object type	actions
Table	ALTER, COMMENT, DELETE, GRANT, INDEX, INSERT, LOCK, RENAME, SELECT, UPDATE
View	ALTER, COMMENT, GRANT, SELECT
Sequence	ALTER, COMMENT, GRANT, SELECT
Stored function/ procedure	ALTER, COMMENT, EXECUTE, GRANT

Table 3-9 各对象类型的audit action

例如审计表u1.t1的DML时创建生成audit policy

```
CREATE AUDIT POLICY audit_t1_dml
       ACTIONS INSERT ON u1.t1
             , DELETE ON u1.t1
             , UPDATE ON u1.t1
;
```

如下通过查询AUDIT_POLICY_OPTIONS视图查看Object action auditing信息

```
SELECT audit_option
     , audit_option_type
     , object_schema
     , object_name
  FROM audit_policy_options
```

```
 WHERE policy_name = 'AUDIT_T1_DML'
 ORDER BY audit_option
;

AUDIT_OPTION AUDIT_OPTION_TYPE OBJECT_SCHEMA OBJECT_NAME
------------ ----------------- ------------- -----------
DELETE       OBJECT ACTION     U1            T1
INSERT       OBJECT ACTION     U1            T1
UPDATE       OBJECT ACTION     U1            T1

3 rows selected.
```

ALL选项（例如ALL ON schema.object）表示可以为该对象定义的所有的audit action

以下为与ALL选项同时使用其他选项的示例

```
CREATE AUDIT POLICY p1
       ACTIONS ALL ON u1.seq1
             , ALTER ON u1.seq1
;

Audit policy created.

SELECT audit_option
     , audit_option_type
     , object_schema
     , object_name
```

```
  FROM audit_policy_options
 WHERE policy_name = 'P1'
 ORDER BY audit_option
;


AUDIT_OPTION AUDIT_OPTION_TYPE OBJECT_SCHEMA OBJECT_NAME
------------ ----------------- ------------- -----------
ALL          OBJECT ACTION     U1            SEQ1
ALTER        OBJECT ACTION     U1            SEQ1


2 rows selected.
```

如下删除ALL选项并非删除所有audit option而是仅删除ALL选项

```
ALTER AUDIT POLICY p1
      DROP ACTIONS ALL ON u1.seq1;


Audit Policy altered.




SELECT audit_option
     , audit_option_type
     , object_schema
     , object_name
  FROM audit_policy_options
```

```
 WHERE policy_name = 'P1'
 ORDER BY audit_option;



AUDIT_OPTION AUDIT_OPTION_TYPE OBJECT_SCHEMA OBJECT_NAME
------------ ----------------- ------------- -----------
ALTER        OBJECT ACTION     U1            SEQ1

1 row selected.
```

Stored function或stored procedure的EXECUTE成功失败的审计仅通过是否可在实际执行时间点执行来决定

- WHENEVER NOT SUCCESSFUL在无法执行stored function/procedure时创建审计记录
- WHENEVER SUCCESSFUL即使在stored function/procedure内部执行SQL语句的过程中报错也会创建审计记录
- 如果需要对stored function/procedure内部的SQL语句失败进行审计该SQL语句需包含在审计对象内

以下为执行包含stored function的SELECT语句的示例

```
SELECT others.func1( t1.c1 )
  FROM t1;
```

由于权限不足等错误导致的others.func1()调用失败属于WHENEVER NOT SUCCESSFUL

成功调用others.func1()时即使执行stored function中的SQL语句的过程中报错也属于

WHENEVER SUCCESSFUL

System Action Auditing

与特定对象无关审计SQL语句

有效的system action查询V$AUDITABLE_SYSTEM_ACTIONS

```
SELECT action_name FROM v$auditable_system_actions;


ACTION_NAME

----------------------

ALL

DDL

SELECT

INSERT

UPDATE

DELETE

EXECUTE

CREATE TABLE

DROP TABLE

ALTER TABLE

LOCK TABLE

TRUNCATE TABLE

ANALYZE TABLE

RENAME

CREATE INDEX
```

DROP INDEX

ALTER INDEX

CREATE SEQUENCE

DROP SEQUENCE

ALTER SEQUENCE

GRANT

REVOKE

CREATE SYNONYM

DROP SYNONYM

CREATE VIEW

DROP VIEW

ALTER VIEW

CREATE PROCEDURE

DROP PROCEDURE

ALTER PROCEDURE

CREATE FUNCTION

DROP FUNCTION

ALTER FUNCTION

COMMENT

ALTER DATABASE

CREATE PROFILE

DROP PROFILE

ALTER PROFILE

CREATE TABLESPACE

DROP TABLESPACE

```
ALTER TABLESPACE

CREATE USER

DROP USER

ALTER USER

CHANGE PASSWORD

CREATE SCHEMA

DROP SCHEMA

CREATE AUDIT POLICY

DROP AUDIT POLICY

ALTER AUDIT POLICY

AUDIT

NOAUDIT

ALTER SYSTEM

ALTER SESSION

ANALYZE SYSTEM

COMMIT

ROLLBACK

SAVEPOINT

LOGON

LOGOFF

SET SESSION

SET TRANSACTION

SET CONSTRAINTS

CREATE CLUSTER GROUP

DROP CLUSTER GROUP
```

```
ALTER CLUSTER GROUP

CREATE CLUSTER LOCATION

DROP CLUSTER LOCATION

ALTER CLUSTER LOCATION

PURGE CONSTRAINT

PURGE INDEX

PURGE TABLE

PURGE TABLESPACE

PURGE RECYCLEBIN

PURGE DBA_RECYCLEBIN

FLASHBACK TABLE


76 rows selected.
```

各SQL语句的system action名称通过查询V$SQL_COMMAND view进行查看

```
SELECT command, audit_action FROM v$sql_command;


COMMAND                                                  AUDIT_ACTION

-------------------------------------------------------- ----------------

-------

ALTER AUDIT POLICY                                       ALTER AUDIT

POLICY

ALTER CLUSTER GROUP .. ADD CLUSTER MEMBER                ALTER CLUSTER

GROUP

ALTER CLUSTER GROUP .. OFFLINE CLUSTER MEMBER            ALTER CLUSTER
```

```
GROUP
ALTER DATABASE DROP INACTIVE CLUSTER MEMBERS              ALTER DATABASE
ALTER DATABASE OFFLINE INACTIVE CLUSTER MEMBERS           ALTER DATABASE


... 省略 ...


DROP CLUSTER LOCATION                                     DROP CLUSTER
LOCATION
PROCEDUAL LANGUAGE BLOCK                                  null
PURGE CONSTRAINT                                          PURGE CONSTRAINT
PURGE INDEX                                               PURGE INDEX
PURGE TABLE                                               PURGE TABLE
PURGE TABLESPACE                                          PURGE TABLESPACE
PURGE RECYCLEBIN                                          PURGE RECYCLEBIN
PURGE DBA_RECYCLEBIN                                      PURGE
DBA_RECYCLEBIN
FLASHBACK TABLE                                           FLASHBACK TABLE


185 rows selected.
```

以下为创建包含system action的audit policy并查询audit option的示例

```
CREATE AUDIT POLICY p1
       ACTIONS SELECT
             , DROP TABLE
             , DROP USER
```

```
;


Audit Policy created.



SELECT audit_option
     , audit_option_type
     , object_schema
     , object_name
  FROM audit_policy_options
 WHERE policy_name = 'P1'
 ORDER BY audit_option
;


AUDIT_OPTION AUDIT_OPTION_TYPE OBJECT_SCHEMA OBJECT_NAME
------------ ----------------- ------------- -----------
DROP TABLE   SYSTEM ACTION     null          null
DROP USER    SYSTEM ACTION     null          null
SELECT       SYSTEM ACTION     null          null


3 rows selected.
```

Useful Audit Policy

以下为定义有效audit policy的示例

- 审计logon failure

```
CREATE AUDIT POLICY AUDIT_LOGON_FAILURES
       ACTIONS LOGON
;


AUDIT POLICY AUDIT_LOGON_FAILURES
      WHENEVER NOT SUCCESSFUL
;
```

- 审计Data Definition Language (DDL)执行

```
CREATE AUDIT POLICY AUDIT_DDL
       ACTIONS DDL
;


AUDIT POLICY AUDIT_DDL
      WHENEVER SUCCESSFUL
;
```

- 审计参数变更

```
CREATE AUDIT POLICY AUDIT_DATABASE_PARAMETER
       ACTIONS ALTER DATABASE
             , ALTER SYSTEM
;
```

```
AUDIT POLICY AUDIT_DATABASE_PARAMETER
      WHENEVER SUCCESSFUL
;
```

- 审计账号变更

```
CREATE AUDIT POLICY AUDIT_ACCOUNT_MGMT
       ACTIONS CREATE USER
             , DROP USER
             , ALTER USER
             , CHANGE PASSWORD
             , GRANT
             , REVOKE
;

AUDIT POLICY AUDIT_ACCOUNT_MGMT
;
```

- 审计Center for Internet Security(CIS)建议事项

```
CREATE AUDIT POLICY AUDIT_CIS_RECOMMENDATIONS
       PRIVILEGES ALTER SYSTEM
                , ALTER DATABASE
          ACTIONS CREATE USER
                , DROP USER
```

```
                , ALTER USER
                , CHANGE PASSWORD
                , GRANT
                , REVOKE
                , CREATE PROFILE
                , ALTER PROFILE
                , DROP PROFILE
                , CREATE SYNONYM
                , DROP SYNONYM
                , CREATE PROCEDURE
                , DROP PROCEDURE
                , ALTER PROCEDURE
;


AUDIT POLICY AUDIT_CIS_RECOMMENDATIONS
      WHENEVER SUCCESSFUL
;
```

Audit Policy运行

激活Audit Policy

Audit policy对象在被激活之前不执行审计

要执行审计需如下使用AUDIT POLICY语句激活audit policy对象

```
CREATE AUDIT POLICY audit_t1_dml
       ACTIONS INSERT ON u1.t1
             , DELETE ON u1.t1
             , UPDATE ON u1.t1
;

Audit policy created.

AUDIT POLICY audit_t1_dml;

Audit succeeded.
```

激活audit policy后仅对新的会话执行审计不影响原有会话

使用AUDIT POLICY语句激活audit policy时使用BY子句或EXCEPT子句指定执行审计的用户或使用WHENEVER子句审计audit action的成功/失败

- BY | EXCEPT
  - BY user_list：指定执行审计的用户
  - EXCEPT user_list：审计除指定用户外的所有用户
  - 省略时审计所有用户
- WHENEVER
  - WHENEVER SUCCESSFUL：Audit action成功时创建audit record
  - WHENEVER NOT SUCCESSFUL：Audit action失败时创建audit record
  - 省略时不管audit action成功与否均创建audit record

Audit policy的激活信息可通过查询AUDIT_POLICY_ENABLED view进行查看

```
SELECT policy_name
     , enabled_opt
     , user_name
     , when_success
     , when_failure
  FROM audit_policy_enabled
 WHERE policy_name = 'AUDIT_T1_DML'
;


POLICY_NAME  ENABLED_OPT USER_NAME  WHEN_SUCCESS WHEN_FAILURE
------------ ----------- ---------- ------------ ------------
AUDIT_T1_DML BY          ALL USERS  YES          YES
```

如上述示例省略审计对象用户时则输出代表所有用户的ALL USERS

使用BY与EXCEPT子句时需注意如下事项

- 不能对相同audit policy同时使用BY子句与EXCEPT子句

```
AUDIT POLICY audit_t1_dml BY u1;


Audit succeeded.


AUDIT POLICY audit_t1_dml EXCEPT u2;
```

```
ERR-42000(16475): audit policy already applied with the BY clause
```

- 在相同的audit policy使用多个AUDIT POLICY BY子句时激活user的集合即以下两个示例具有相同意义
  - 例1

```
AUDIT POLICY audit_t1_dml BY u1;


Audit succeeded.


AUDIT POLICY audit_t1_dml BY u2;


Audit succeeded.


SELECT policy_name
     , enabled_opt
     , user_name
     , when_success
     , when_failure
  FROM audit_policy_enabled
 WHERE policy_name = 'AUDIT_T1_DML'
;


POLICY_NAME  ENABLED_OPT USER_NAME WHEN_SUCCESS WHEN_FAILURE
------------ ----------- --------- ------------ ------------
AUDIT_T1_DML BY          U1        YES          YES
```

```
AUDIT_T1_DML BY          U2        YES          YES


2 rows selected.
```

- 例2

```
AUDIT POLICY audit_t1_dml BY u1, u2;


Audit succeeded.


SELECT policy_name
     , enabled_opt
     , user_name
     , when_success
     , when_failure
  FROM audit_policy_enabled
 WHERE policy_name = 'AUDIT_T1_DML'
;


POLICY_NAME  ENABLED_OPT USER_NAME WHEN_SUCCESS WHEN_FAILURE
------------ ----------- --------- ------------ ------------
AUDIT_T1_DML BY          U1        YES          YES
AUDIT_T1_DML BY          U2        YES          YES


2 rows selected.
```

- 在相同的audit policy使用多个AUDIT POLICY EXCEPT子句时只有最后一个AUDIT POLICY有效即以下两个示例具有不同意义
    - 例1

```
AUDIT POLICY audit_t1_dml EXCEPT u1;


Audit succeeded.


AUDIT POLICY audit_t1_dml EXCEPT u2;


Audit succeeded.


SELECT policy_name
     , enabled_opt
     , user_name
     , when_success
     , when_failure
  FROM audit_policy_enabled
 WHERE policy_name = 'AUDIT_T1_DML'
;


POLICY_NAME  ENABLED_OPT USER_NAME WHEN_SUCCESS WHEN_FAILURE
------------ ----------- --------- ------------ ------------
AUDIT_T1_DML EXCEPT      U2        YES          YES
```

```
1 row selected.
```

- 例2

```
AUDIT POLICY audit_t1_dml EXCEPT u1, u2;


Audit succeeded.


SELECT policy_name
     , enabled_opt
     , user_name
     , when_success
     , when_failure
  FROM audit_policy_enabled
 WHERE policy_name = 'AUDIT_T1_DML'
;


POLICY_NAME  ENABLED_OPT USER_NAME WHEN_SUCCESS WHEN_FAILURE
------------ ----------- --------- ------------ ------------
AUDIT_T1_DML EXCEPT      U1        YES          YES
AUDIT_T1_DML EXCEPT      U2        YES          YES


2 rows selected.
```

- 与BY子句同时使用的WHENEVER子句会被累计即以下两个示例具有相同意义
  - 例1

```
AUDIT POLICY audit_t1_dml BY u1 WHENEVER SUCCESSFUL;


Audit succeeded.


SELECT policy_name
     , enabled_opt
     , user_name
     , when_success
     , when_failure
  FROM audit_policy_enabled
 WHERE policy_name = 'AUDIT_T1_DML'
;


POLICY_NAME  ENABLED_OPT USER_NAME WHEN_SUCCESS WHEN_FAILURE
------------ ----------- --------- ------------ ------------
AUDIT_T1_DML BY          U1        YES          NO


1 row selected.


AUDIT POLICY audit_t1_dml BY u1 WHENEVER NOT SUCCESSFUL;


Audit succeeded.


SELECT policy_name
     , enabled_opt
```

```
     , user_name
     , when_success
     , when_failure
  FROM audit_policy_enabled
 WHERE policy_name = 'AUDIT_T1_DML'
;

POLICY_NAME  ENABLED_OPT USER_NAME WHEN_SUCCESS WHEN_FAILURE
------------ ----------- --------- ------------ ------------
AUDIT_T1_DML BY          U1        YES          YES

1 row selected.
```

- 例2

```
AUDIT POLICY audit_t1_dml BY u1;

Audit succeeded.

SELECT policy_name
     , enabled_opt
     , user_name
     , when_success
     , when_failure
  FROM audit_policy_enabled
 WHERE policy_name = 'AUDIT_T1_DML'
```

```
;


POLICY_NAME  ENABLED_OPT USER_NAME WHEN_SUCCESS WHEN_FAILURE

------------ ----------- --------- ------------ ------------

AUDIT_T1_DML BY          U1        YES          YES


1 row selected.
```

- 与EXCEPT子句同时使用的WHENEVER子句只有最后一个有效即以下两个示例具有不同意义
  - 例1

```
AUDIT POLICY audit_t1_dml EXCEPT u1 WHENEVER SUCCESSFUL;


Audit succeeded.


AUDIT POLICY audit_t1_dml EXCEPT u1 WHENEVER NOT SUCCESSFUL;


Audit succeeded.


SELECT policy_name
     , enabled_opt
     , user_name
     , when_success
     , when_failure
  FROM audit_policy_enabled
```

```
 WHERE policy_name = 'AUDIT_T1_DML'
;

POLICY_NAME  ENABLED_OPT USER_NAME WHEN_SUCCESS WHEN_FAILURE
------------ ----------- --------- ------------ ------------
AUDIT_T1_DML EXCEPT      U1        NO           YES

1 row selected.
```

- 例2

```
AUDIT POLICY audit_t1_dml EXCEPT u1;

Audit succeeded.

SELECT policy_name
     , enabled_opt
     , user_name
     , when_success
     , when_failure
  FROM audit_policy_enabled
 WHERE policy_name = 'AUDIT_T1_DML'
;

POLICY_NAME  ENABLED_OPT USER_NAME WHEN_SUCCESS WHEN_FAILURE
------------ ----------- --------- ------------ ------------
```

```
AUDIT_T1_DML EXCEPT      U1        YES          YES


1 row selected.
```

禁用Audit Policy

为了禁用Audit policy需执行NOAUDIT POLICY语句

NOAUDIT POLICY语句仅适用于新的会话不影响原有会话

使用如下命令设置为无激活信息时完全禁用audit policy

```
SELECT policy_name
     , enabled_opt
     , user_name
     , when_success
     , when_failure
  FROM audit_policy_enabled
 WHERE policy_name = 'AUDIT_T1_DML'
;


no rows selected.
```

NOAUDIT POLICY语句删除根据AUDIT POLICY指定方式生成的个别激活信息

以下为仅禁用对audit_t1_dml的u1用户的审计的示例

```
SELECT policy_name
```

```
     , enabled_opt
     , user_name
     , when_success
     , when_failure
  FROM audit_policy_enabled
 WHERE policy_name = 'AUDIT_T1_DML'
;


POLICY_NAME  ENABLED_OPT USER_NAME WHEN_SUCCESS WHEN_FAILURE
------------ ----------- --------- ------------ ------------
AUDIT_T1_DML BY          U1        YES          YES
AUDIT_T1_DML BY          SYS       YES          YES


2 rows selected.


NOAUDIT POLICY audit_t1_dml BY u1;


Noaudit succeeded.


SELECT policy_name
     , enabled_opt
     , user_name
     , when_success
     , when_failure
  FROM audit_policy_enabled
```

```
 WHERE policy_name = 'AUDIT_T1_DML'
;

POLICY_NAME  ENABLED_OPT USER_NAME WHEN_SUCCESS WHEN_FAILURE
------------ ----------- --------- ------------ ------------
AUDIT_T1_DML BY          SYS       YES          YES

1 row selected.
```

使用AUDIT POLICY name BY子句时需通过NOAUDIT POLICY name BY语句禁用

使用AUDIT POLICY name EXCEPT子句时需通过NOAUDIT POLICY name语句禁用无需BY子句

如要禁用AUDIT POLICY的各个选项根据其使用方法如下使用NOAUDIT POLICY语句

类型	AUDIT POLICY语句	NOAUDIT POLICY语句
所有用户	AUDIT POLICY p1	NOAUDIT POLICY p1
使用BY	AUDIT POLICY p1 BY u1	NOAUDIT POLICY p1 BY u1
使用EXCEPT	AUDIT POLICY p1 EXCEPT u1	NOAUDIT POLICY p1

Table 3-10 激活/禁用Audit policy

如下激活所有用户时NOAUDIT POLICY BY子句不产生任何影响

```
AUDIT POLICY audit_t1_dml;

Audit succeeded.
```

```
SELECT policy_name
     , enabled_opt
     , user_name
     , when_success
     , when_failure
  FROM audit_policy_enabled
 WHERE policy_name = 'AUDIT_T1_DML'
;

POLICY_NAME  ENABLED_OPT USER_NAME WHEN_SUCCESS WHEN_FAILURE
------------ ----------- --------- ------------ ------------
AUDIT_T1_DML BY          ALL USERS YES          YES

1 row selected.

NOAUDIT POLICY audit_t1_dml BY u1;

Noaudit succeeded.

SELECT policy_name
     , enabled_opt
     , user_name
     , when_success
     , when_failure
```

```
  FROM audit_policy_enabled
 WHERE policy_name = 'AUDIT_T1_DML'
;

POLICY_NAME  ENABLED_OPT USER_NAME WHEN_SUCCESS WHEN_FAILURE
------------ ----------- --------- ------------ ------------
AUDIT_T1_DML BY          ALL USERS YES          YES

1 row selected.
```

如果单独激活一个或多个用户需根据AUDIT POLICY设置方法使用NOAUDIT POLICY语句

- 使用BY子句激活时
  - 以下为使用BY子句激活的示例

```
AUDIT POLICY p1 WHENEVER NOT SUCCESSFUL;
AUDIT POLICY p1 BY u1;
AUDIT POLICY p1 BY u2;
```

  - 以下为查询激活信息的结果

```
SELECT policy_name
     , enabled_opt
     , user_name
     , when_success
     , when_failure
```

```
  FROM audit_policy_enabled
 WHERE policy_name = 'P1';


POLICY_NAME ENABLED_OPT USER_NAME WHEN_SUCCESS WHEN_FAILURE
----------- ----------- --------- ------------ ------------
P1          BY          ALL USERS NO           YES
P1          BY          U1        YES          YES
P1          BY          U2        YES          YES


3 rows selected.
```

- 以下为执行NOAUDIT语句时的激活信息

```
NOAUDIT POLICY p1;


Noaudit succeeded.


SELECT policy_name
     , enabled_opt
     , user_name
     , when_success
     , when_failure
  FROM audit_policy_enabled
 WHERE policy_name = 'P1';


POLICY_NAME ENABLED_OPT USER_NAME WHEN_SUCCESS WHEN_FAILURE
```

```
----------- ----------- --------- ------------ ------------

P1          BY          U1        YES          YES

P1          BY          U2        YES          YES


2 rows selected.
```

上述示例禁用了对ALL USERS的审计但对用户u1u2的审计仍为激活状态

如下使用BY选项再次使用NOAUDIT POLICY语句时完全禁用audit policy p1

```
NOAUDIT POLICY p1 BY u1, u2;


Noaudit succeeded.


SELECT policy_name
     , enabled_opt
     , user_name
     , when_success
     , when_failure
  FROM audit_policy_enabled
 WHERE policy_name = 'P1';


no rows selected.
```

- 使用EXCEPT激活时
    - 以下为使用audit policy激活的示例

```
AUDIT POLICY p1 EXCEPT u1, sys;
```

- 以下为查询激活信息的结果

```
SELECT policy_name
     , enabled_opt
     , user_name
     , when_success
     , when_failure
  FROM audit_policy_enabled
 WHERE policy_name = 'P1';


POLICY_NAME ENABLED_OPT USER_NAME WHEN_SUCCESS WHEN_FAILURE
----------- ----------- --------- ------------ ------------
P1          EXCEPT      U1        YES          YES
P1          EXCEPT      SYS       YES          YES


2 rows selected.
```

- NOAUDIT POLICY语句与AUDIT POLICY语句不同无EXCEPT option并如下无选项地执行语句

```
NOAUDIT POLICY p1;


Noaudit succeeded.
```

```
SELECT policy_name
     , enabled_opt
     , user_name
     , when_success
     , when_failure
  FROM audit_policy_enabled
 WHERE policy_name = 'P1';


no rows selected.
```

即使用EXCEPT选项激活audit policy时无法使用NOAUDIT POLICY语句重新激活个别用户

3.4 Authorization

Authorization相关语句

详细内容参考如下

- 创建User：**CREATE USER**
- 删除User：**DROP USER**
- 更改User：**ALTER USER**
- 赋予权限：**GRANT privileges TO**
- 收回权限：**REVOKE privileges FROM**

通过以下视图可查询用户对象与权限相关信息

对象集合	视图	说明
DICTIONARY_SCHEMA	**ALL_COL_PRIVS**	用户可访问的column的权限
	ALL_COL_PRIVS_MADE	用户为grantor的column权限
	ALL_COL_PRIVS_RECD	用户为grantee的column权限
	ALL_DB_PRIVS	与用户相关的DB权限
	ALL_DB_PRIVS_MADE	用户为grantor的DB权限
	ALL_DB_PRIVS_RECD	用户为grantee的DB权限

对象集合	视图	说明
	ALL_PROC_PRIVS	与用户相关的Stored Procedure/Function权限
	ALL_PROC_PRIVS_MADE	用户为grantor的Stored Procedure/Function权限
	ALL_PROC_PRIVS_RECD	用户为grantee的Stored Procedure/Function权限
	ALL_SCHEMA_PRIVS	用户可访问的SCHEMA的权限
	ALL_SCHEMA_PRIVS_MADE	用户为grantor的SCHEMA权限
	ALL_SCHEMA_PRIVS_RECD	用户为grantee的SCHEMA权限
	ALL_SEQ_PRIVS	用户可访问的序列权限
	ALL_SEQ_PRIVS_MADE	用户为grantor的序列的权限
	ALL_SEQ_PRIVS_RECD	用户为grantee的序列权限
	ALL_TAB_PRIVS	用户可访问的表的权限
	ALL_TAB_PRIVS_MADE	用户为grantor的表权限
	ALL_TAB_PRIVS_RECD	用户为grantee的表权限
	ALL_TBS_PRIVS	用户可访问的表空间的权限
	ALL_TBS_PRIVS_MADE	用户为grantor的表空间权限
	ALL_TBS_PRIVS_RECD	用户为grantee的表空间权限

对象集合	视图	说明
	ALL_USERS	用户可访问的用户信息
	USER_COL_PRIVS	用户所拥有的column的权限信息
	USER_COL_PRIVS_MADE	用户所拥有的column的权限赋予信息
	USER_COL_PRIVS_RECD	用户所拥有的column的权限获取信息
	USER_PROC_PRIVS	用户所拥有的stored procedure/ function的权限信息
	USER_PROC_PRIVS_MADE	用户所拥有的stored procedure/ function的权限赋予信息
	USER_PROC_PRIVS_RECD	用户所拥有的stored procedure/ function的权限获取信息
	USER_SCHEMA_PRIVS	用户所拥有的SCHEMA的权限信息
	USER_SCHEMA_PRIVS_MADE	用户所拥有的SCHEMA的权限赋予信息
	USER_SCHEMA_PRIVS_RECD	用户所拥有的SCHEMA的权限获取信息
	USER_SEQ_PRIVS	用户所拥有的序列的权限信息
	USER_SEQ_PRIVS_MADE	用户所拥有的序列的权限赋予信息
	USER_SEQ_PRIVS_RECD	用户所拥有的序列的权限获取信息
	USER_TAB_PRIVS	用户所拥有的表的权限信息
	USER_TAB_PRIVS_MADE	用户所拥有的表的权限赋予信息

对象集合	视图	说明
	USER_TAB_PRIVS_RECD	用户所拥有的表的权限获取信息
	USER_USERS	当前用户信息
INFORMATION_SCHEMA	COLUMN_PRIVILEGES	用户可访问的dolumn权限信息
	ROUTINE_PRIVILEGES	用户可访问的stored procedure/ function权限信息
	TABLE_PRIVILEGES	用户可访问的表权限信息
	USAGE_PRIVILEGES	用户可访问的序列权限信息

Table 3-11 Authorization对象相关信息

User概念

User对象由用户可执行权限的集合组成特定用户为了执行SQL语句需要有可执行相关对象的SQL语句的权限

例如使用**CREATE USER**语句创建的用户没有任何权限无法连接数据库也无法执行任何SQL语句为了能连接数据库该用户需要有可在数据库对象创建会话的CREATE SESSION ON DATABASE权限即如下执行CREATE USER语句后需要赋予恰当的权限

```
CREATE USER u1 IDENTIFIED BY u1_password;
GRANT CREATE SESSION ON DATABASE TO u1;
COMMIT;
```

创建User对象后需赋予的权限参考**CREATE USER**语句的**使用示例**和**GRANT privileges TO**语句

对象的创建与权限

SQL Schema Object的创建

创建表等SQL schema object时特定用户为了执行CREATE TABLE语句需要先给表所属的上层non-schema object赋予相关权限

以下为CREATE TABLE语句的示例

```
CREATE TABLE t1 ( id BIGINT, name VARCHAR(128) );
```

解析为如下时

```
CREATE TABLE u1.t1 ( id BIGINT, name VARCHAR(128) ) TABLESPACE
mem_data_tbs;
```

如下图t1表的所有者为u1 schema的所有者u1账号表所属的逻辑位置为u1 SCHEMA存储表的物理空间为mem_data_tbs表空间

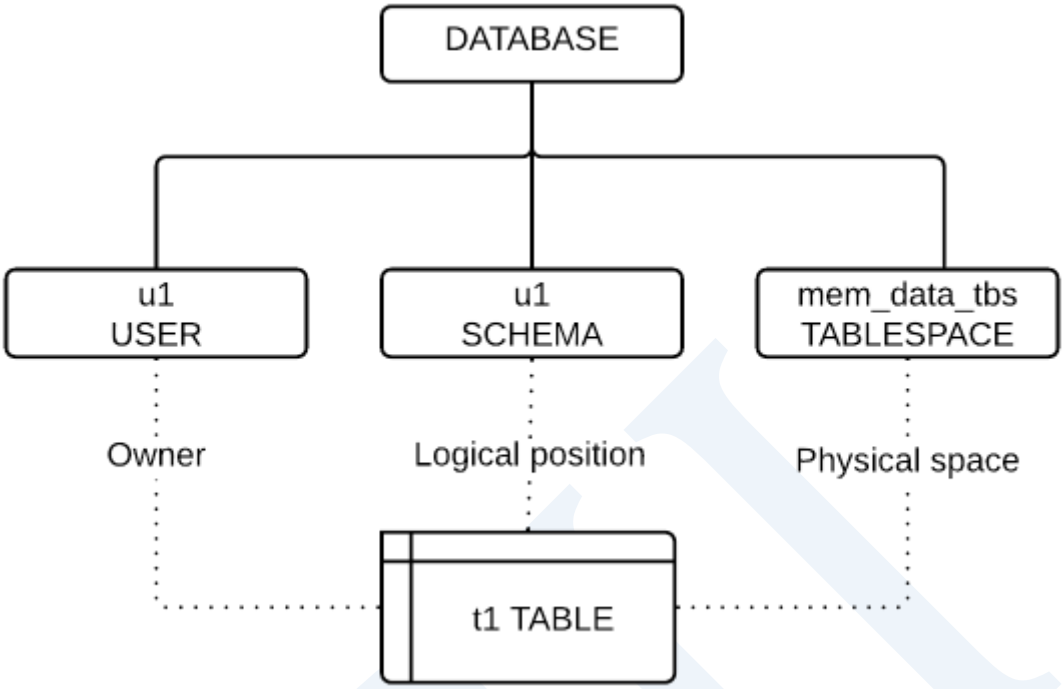


Figure 3-2 CREATE TABLE与non-schema objects

这时执行语句的u1用户对t1表所属的模式与表空间需要有如下权限

为了在表的逻辑位置u1 SCHEMA里创建表需要有下列中的一个权限

- 可在u1 SCHEMA中创建表的权限
  - CREATE TABLE ON SCHEMA ON u1
- 在database中可以创建任意表的权限
  - CREATE ANY TABLE ON DATABASE
  - 即使没有SCHEMA权限如果拥有为u1 SCHEMA上层对象的database的相关权限也可以创建表

为了在表的物理存储空间mem_data_tbs表空间中生成表需要有下列中的一个权限

- 在mem_data_tbs表空间里可以创建对象的权限
  - CREATE OBJECT ON TABLESPACE mem_data_tbs
- 在database中可以使用任意表空间的权限
  - USAGE TABLESPACE ON DATABASE
  - 即使没有表空间的权限如果拥有为mem_data_tbs上层对象的database的相关权限也可以创建表

如下决定表对象的所有者

- 表所属的schema的所有者
- 表所属的schema为PUBLIC时执行语句的使用者

表对象的所有者将拥有如下可变更表结构及操作表数据的权限

- SELECT ON TABLE
  - 可对相应表执行SELECT语句的权限
- INSERT ON TABLE
  - 可对相应表执行INSERT语句的权限
- UPDATE ON TABLE
  - 可对相应表执行UPDATE语句的权限
- DELETE ON TABLE
  - 可对相应表执行DELETE语句的权限
- LOCK ON TABLE
  - 可对相应应表执行LOCK语句的权限
- INDEX ON TABLE
  - 可对相应表执行CREATE/DROP/ALTER INDEX语句的权限

- ALTER ON TABLE
    - 可对相应表执行ALTER TABLE语句的权限

这时被授予的权限的grantor（授权者）是内部使用的_SYSTEM账号表的所有者为grantee（被授权者）因此包括SYS账号在内的任何用户都无法通过**REVOKE privileges FROM**语句删除或变更所有者的权限表的所有者被赋予的权限随着表的删除而消失

创建Non-schema Object

与创建表等的SQL schema object相同创建userschematablespace等non-schema object时也需要拥有为上层对象的database的相关权限

执行以下语句的用户需要拥有如下各个语句的为上层对象的database对象的相关权限

- CREATE USER语句
    - CREATE USER ON DATABASE
    - 可在database中执行CREATE USER语句的权限
- CREATE TABLESPACE语句
    - CREATE TABLESPACE ON DATABASE
    - 可在database中执行CREATE TABLESPACE语句的权限
- CREATE SCHEMA语句
    - CREATE SCHEMA ON DATABASE
    - 可在database中执行CREATE SCHEMA语句的权限
- CREATE PUBLIC SYNONYM语句
    - 可在database中执行CREATE PUBLIC SYNONYM语句的权限

与创建SQL schema object不同non-schema object的所有者不是执行CREATE语句的用户每个对象有以下特点

- User对象
  - 无所有者
  - 为了删除user对象需要有DROP USER ON DATABASE权限
- Tablespace对象
  - 无所有者
  - 为了删除tablespace对象需要有DROP TABLESPACE ON DATABASE权限
- Schema对象
  - CREATE SCHEMA语句中指定的所有者
  - Schema对象的所有者可删除schema对象
- Public synonym对象
  - 无所有者
  - 为了删除public synonym对象需要有DROP PUBLIC SYNONYM ON DATABASE权限

权限（Privilege）

赋予权限

非表等SQL schema object的所有者的用户插入（INSERT）删除（DELETE）更新（UPDATE）或查询（SELECT）数据时需通过**GRANT privileges TO**语句赋予对应表的相关权限

例如非表所有者的用户如下执行SELECT语句时需要有下列中的一个权限如下图即使对t1表没有SELECT权限但如果对上层对象u1 SCHEMA或databasae有SELECT权限也可以查询u1.t1表

- 由test用户执行

```
SELECT id, name FROM u1.t1 WHERE id < 100;
```

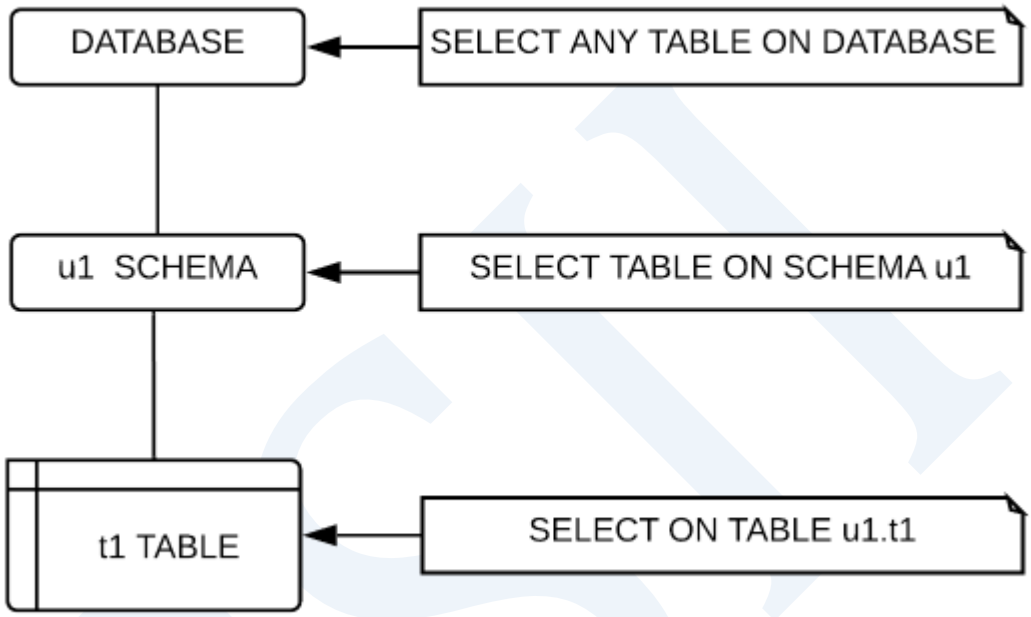


Figure 3-3 执行SELECT语句的权限

- SELECT ON TABLE u1.t1
  - 对u1.t1表的SELECT权限
- SELECT ON SCHEMA u1
  - 对为上层对象的u1 SCHEMA的SELECT权限
  - 可对u1 SCHEMA的所有表进行SELECT的权限
- SELECT ANY TABLE ON DATABASE
  - 对为次上层对象的database的SELECT权限
  - 可对database的所有表进行SELECT权限

为了给其他用户赋予SELECT ON TABLE u1.t1权限执行GRANT语句的用户应是下列中的一个

- u1.t1对象的所有者
  - 对象的所有者拥有_SYSTEM账号赋予的SELECT ON TABLE u1.t1 WITH GRANT OPTION权限
- 拥有SELECT ON TABLE u1.t1 WITH GRANT OPTION的用户
  - WITH GRANT OPTION可将被赋予的权限再赋予给其他用户
- 拥有ACCESS CONTROL ON DATABASE权限的用户
  - 此用户可控制所有对象的所有权限

以下为对象的所有者给其他用户赋予权限的示例

- u1.t1表的所有者执行GRANT语句

```
GRANT SELECT ON TABLE u1.t1 TO test;
```

权限种类参考**GRANT privileges TO**语句执行各SQL语句的权限参考**SQL References**部分的各语句的使用范围与访问权限内容

收回权限

通过**REVOKE privileges FROM**语句收回已赋予给特定用户的权限但删除对象之前无法收回对象所有者的权限

例如以下为可执行REVOKE语句从test用户收回SELECT权限的用户即使用户为表的所有者也不能收回不是自己赋予的权限

```
REVOKE SELECT ON TABLE u1.t1 FROM test;
```

- 赋予权限的用户
  - 给test用户赋予SELECT ON TABLE u1.t1权限的用户
- 拥有ACCESS CONTROL ON DATABASE权限的用户
  - 此用户可控制所有对象的所有权限

如下多个用户给test用户赋予相同的权限时在收回所赋予的所有权限之前test用户可以执行SELECT语句

- 对象的所有者u1给test用户赋予权限

```
GRANT SELECT ON TABLE u1.t1 TO test;
```

- 对象的所有者u1给用户u2赋予WITH GRANT OPTION权限

```
GRANT SELECT ON TABLE u1.t1 TO u2 WITH GRANT OPTION;
```

- 由用户u2执行
- 拥有WITH GRANT OPTION权限的u2给test用户赋予权限

```
GRANT SELECT ON TABLE u1.t1 TO test;
```

上述例子中test用户拥有grantor为u1与u2用户的两个SELECT ON TABLE u1.t1权限即权限信息由{grantor, grantee, 对象权限}构成

PUBLIC账号

PUBLIC账号是指所有用户的特殊账号

例如如下对PUBLIC账号赋予SELECT权限时所有用户均可对u1.t1表执行SELECT语句

```
GRANT SELECT ON TABLE u1.t1 TO PUBLIC;
```

即即使没有被单独赋予SELECT权限的用户也可以使用PUBLIC账号的SELECT权限对ut1.t1表执行SELECT语句

给PUBLIC账号赋予权限时不是给已存在的所有用户赋予权限而是给PUBLIC账号本身赋予权限因此即使创建新的用户新的用户也能执行SELECT语句

同样如下收回PUBLIC账号权限时不是收回所有用户的对应权限如果用户对u1.t1表有SELECT权限则可以执行SELECT语句

- 给test用户赋予权限

```
GRANT SELECT ON TABLE u1.t1 TO test;
```

- 给PUBLIC账号赋予权限

```
GRANT SELECT ON TALBE u1.t1 TO PUBLIC;
```

- 虽然收回了PUBLIC账号的权限但test用户依然有SELECT ON TABLE u1.t1的权限

```
REVOKE SELECT ON TABLE u1.t1 FROM PUBLIC;
```

字段（column）权限

通过对表的特定字段赋予权限控制其他用户的DML或SELECT语句执行

参考以下示例表

```
CREATE TABLE u1.t1
(
   id     BIGINT,
   name   VARCHAR(128),
   addr   VARCHAR(1024),
   salary NUMBER(20,0)
);
```

给其他用户赋予除u1.t1表的salary column信息外的SELECT权限时如下罗列需赋予权限的字段后执行**GRANT privileges TO**语句获得权限的test用户无法查询salary column

- 给test用户赋予SELECT字段权限

```
GRANT SELECT( id, name, addr ) ON TABLE u1.t1 TO test;
```

如下赋予表权限时自动对表的所有字段赋予权限同时给字段与表赋予权限时字段的权限信息为相同信息不进行重复管理

- 给test用户赋予SELECT权限

```
GRANT SELECT ON TABLE u1.t1 TO test;
```

- 如下自动对表的所有column赋予权限

```
GRANT SELECT(id) ON TABLE u1.t1 TO test;

GRANT SELECT(name) ON TABLE u1.t1 TO test;

GRANT SELECT(addr) ON TABLE u1.t1 TO test;

GRANT SELECT(salary) ON TABLE u1.t1 TO test;
```

如下通过执行**REVOKE privileges FROM**语句收回对字段的权限

- 从test用户收回SELECT字段权限

```
REVOKE SELECT( id, name, addr ) ON TABLE u1.t1 FROM test;
```

如下收回表的权限时自动收回对表的所有字段的权限即使分别赋予字段权限和表权限但收回表的权限时自动收回字段的权限

- 收回test用户的SELECT权限
- 自动收回表的所有字段的权限

```
REVOKE SELECT ON TABLE u1.t1 FROM test;
```

如下仅收回column权限时如拥有表权限则仍可使用该表权限执行对应语句因此要给特定字段赋予权限时需先收回表的权限后再给各字段赋权限

- 给test用户赋予SELECT表权限

```
GRANT SELECT ON TABLE u1.t1 TO test;
```

- 从test用户收回SELECT(salary) column权限
- test用户拥有SELECT表权限因此可查询u1.t1表的salary字段

```
REVOKE SELECT(salary) ON TABLE u1.t1 FROM test;
```

表和字段的权限相关内容参考**GRANT privileges TO**语句

3.5 Schema

Schema相关语句

详细内容参考下列链接

- 创建Schema： **CREATE SCHEMA**
- 删除Schema： **DROP SCHEMA**

通过以下视图可查询SCHEMA对象相关信息

对象集合	视图	说明
DICTIONARY_SCHEMA	**ALL_SCHEMAS**	用户可访问的schema信息
	ALL_SCHEMA_PATH	用户可访问的schema路径
	USER_SCHEMAS	用户拥有的schema信息
	USER_SCHEMA_PATH	用户的schema路径信息
INFORMATION_SCHEMA	**SCHEMATA**	用户可访问的schema信息

Table 3-12 SCHEMA对象相关信息

Schema概念

数据库由一个以上的SCHEMA构成SCHEMA由表索引视图序列等操纵数据的对象构成这种属于SCHEMA的对象称为SQL schema object

SCHEMA类似于操作系统的目录SCHEMA与表的关系类似于操作系统的目录与文件的关系即SCHEMA为SQL schema object的逻辑位置是区分名称的标准各个SQL schema object在schema内的名称不可重复SCHEMA内的naming space如下所示

- Table, view, sequence, private synonym, stored procedure, stored function
- Index
- Constraint

不同SCHEMA内可以有相同名称的表当访问相同名称的不同表时需要连同SCHEMA名称一起指定并执行语句

```
gSQL> SELECT u1.t1.id, u1.t1.name, u2.t1.addr
        FROM u1.t1, u2.t1
       WHERE u1.t1.id = u2.t1.id;
```

SQL schema object的名称可如下指定或省略SCHEMA名省略时schema的名称取决于执行语句的用户的schema pathSchema path的详细内容参考**Schema Path**

- 指定schema名时

```
gSQL> SELECT u1.t1.id, u1.t1.name FROM u1.t1;
```

- 未指定schema名时

```
gSQL> SELECT t1.id, t1.name FROM t1;
```

User与Schema

SUNDB的用户与SCHEMA之间的关系是1:N的关系即用户可以没有属于自己的SCHEMA或可以有多个SCHEMA

SQL标准未明确规定userschemadatabase等non-schema对象之间的关系不同的数据库对non-schema对象之间的关系有以下不同的定义

- Oracle
  - user : schema = 1 : 1关系
- DB2
  - user : schema = 1 : N关系
- Postgres
  - user : schema = 1 : N关系
- MySQL
  - database : schema = 1 : 1关系
  - user为schema(database) 的下层对象

创建SUNDB数据库时可如下构成适合客户系统特征的多种用户与SCHEMA关系

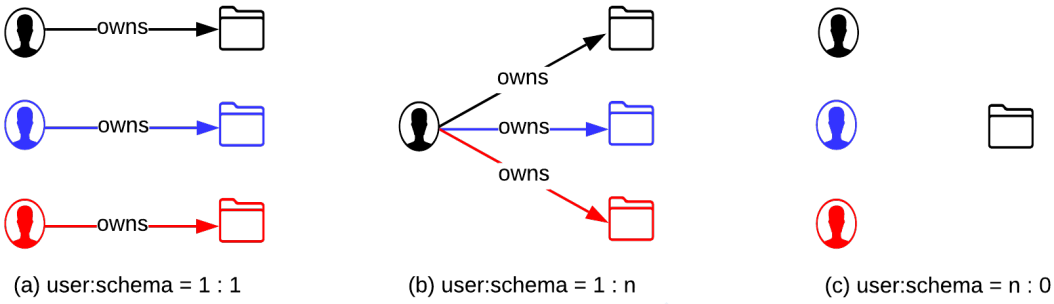


Figure 3-4 User与schema的关系

如图(a)构建数据库使每个用户有对应的SCHEMA时如下创建用户与schema

- 创建u1用户时同时创建并拥有u1 SCHEMA

```
gSQL> CREATE USER u1 IDENTIFIED BY u1_password WITH SCHEMA;
```

- 可以省略WITH SCHEMA语句并创建与用户相同名称的SCHEMA

```
gSQL> CREATE USER u2 IDENTIFIED BY u2_password;
gSQL> CREATE USER u3 IDENTIFIED BY u3_password;
```

如图(b)构建数据库使一个用户有多个SCHEMA时如下创建用户与schema

- 不创建SCHEMA仅创建u1用户

```
gSQL> CREATE USER u1 IDENTIFIED BY u1_password WITHOUT SCHEMA;
```

- 创建多个SCHEMA并将各schema的所有者指定为u1用户

```
gSQL> CREATE SCHEMA s1 AUTHORIZATION u1;
gSQL> CREATE SCHEMA s2 AUTHORIZATION u1;
gSQL> CREATE SCHEMA s3 AUTHORIZATION u1;
```

如图(c)不单独创建SCHEMA构建数据库使多个用户共享一个SCHEMA时如下创建用户以下为所有用户共享**PUBLIC Schema**的示例

- 不创建SCHEMA仅创建用户

```
gSQL> CREATE USER u1 IDENTIFIED BY u1_password WITHOUT SCHEMA;
gSQL> CREATE USER u2 IDENTIFIED BY u2_password WITHOUT SCHEMA;
gSQL> CREATE USER u3 IDENTIFIED BY u3_password WITHOUT SCHEMA;
```

用户与SCHEMA的创建相关内容参考**CREATE USER**语句与**CREATE SCHEMA**语句

Schema Path

使用不带schema名称的表等SQL schema对象名称时通过schema path寻找schema名称Schema path的概念类似于UNIX系统的PATH环境变量即类似于在UNIX 系统中执行命令时按照PATH路径指定的顺序搜索命令

用户拥有多个SCHEMA并如下创建或查询无schema名称的表时根据schema path决定在哪个SCHEMA下创建表

- 在s1 schema创建t1表

```
gSQL> CREATE TABLE s1.t1 ( id INTEGER );
```

- 在s2 schema创建t1表

```
gSQL> CREATE TABLE s2.t1 ( name VARCHAR(128) );
```

- 在哪一个schema创建t1表?

```
gSQL> CREATE TABLE t1 ( address VARCHAR(1024) );
```

- 查询哪一个schema的t1表?

```
gSQL> SELECT * FROM t1;
```

下图为u1用户的schema path的图形化示例u1用户的schema path顺序为{s1, s2, s3}s1 SCHEMA有t1表 s2 SCHEMA有t2表s3 SCHEMA有t1和t3表

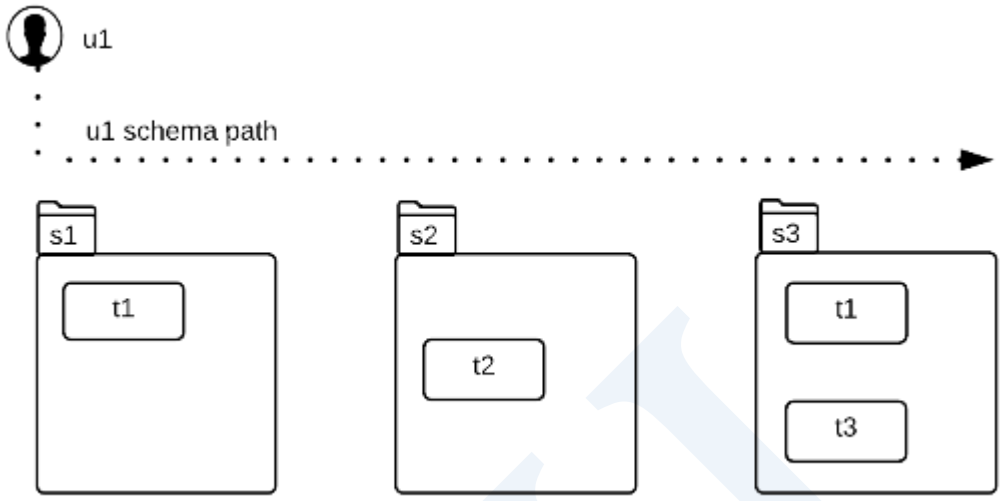


Figure 3-5 Schema path示例

如下SELECT语句中省略SCHEMA名称时根据schema path解析SCHEMA名称

- 根据schema path解析为s1.t1

```
gSQL> SELECT * FROM t1;

gSQL> SELECT * FROM s1.t1;
```

- 根据schema path解析为s2.t2

```
gSQL> SELECT * FROM t2;

gSQL> SELECT * FROM s2.t2;
```

- 根据schema path解析为s3.t3

```
gSQL> SELECT * FROM t3;
```

```
gSQL> SELECT * FROM s3.t3;
```

以上示例中省略schema名称时schema path确定s1.t1表因此为了查询s3.t1表需如下指定SCHEMA名称

- 根据schema path解析为s1.t1表

```
gSQL> SELECT * FROM t1;
```

- 为了查询s3.t1表需要指定s3 schema

```
gSQL> SELECT * FROM s3.t1;
```

如下省略SCHEMA名称的CREATE TABLE语句将生成于schema path的第一个schema （s1）中

- 已存在相同的s1.t1表因此报错

```
gSQL> CREATE TABLE t1 ( id INTEGER );
gSQL> CREATE TABLE s1.t1 ( id INTEGER );
```

- 虽然已存在s2.t2表但创建属于不同schema的s1.t2表

```
gSQL> CREATE TABLE t2 ( name VARCHAR(128) );
gSQL> CREATE TABLE s1.t2 ( name VARCHAR(128) );
```

- 虽然已存在s3.t3表但创建属于不同schema的s1.t3表

```
gSQL> CREATE TABLE t3 ( name VARCHAR(128) );
```

```
gSQL> CREATE TABLE s1.t3 ( name VARCHAR(128) );
```

如果当前用户创建对象时省略SCHEMA名称则使用的SCHEMA名称可以通过SQL标准函数

CURRENT_SCHEMA查询

```
gSQL> SELECT current_schema FROM dual;


CURRENT_SCHEMA
--------------
S1


1 row selected.
```

如下当前用户的schema path信息可通过DICTIONARY_SCHEMA的ALL_SCHEMA_PATH视图查询

```
gSQL> SELECT * FROM all_schema_path;


AUTH_NAME SCHEMA_NAME             SEARCH_ORDER
--------- ----------------------- ------------
U1        S1                                1
U1        S2                                2
U1        S3                                3
U1        PUBLIC                            4
PUBLIC    DICTIONARY_SCHEMA                 5
PUBLIC    INFORMATION_SCHEMA                6
PUBLIC    DEFINITION_SCHEMA                 7
```

```
PUBLIC    PERFORMANCE_VIEW_SCHEMA            8

PUBLIC    FIXED_TABLE_SCHEMA                 9


9 rows selected.
```

如上述例子u1用户的schema path顺序为{s1, s2, s3, public}PUBLIC账号的schema path顺序为{DICTIONARY_SCHEMA, INFORMATION_SCHEMA, DEFINITION_SCHEMA, PERFORMANCE_VIEW_SCHEMA, FIXED_TABLE_SCHEMA}即省略SCHEMA名称并查询任意t1表时先搜索当前用户u1的schema path不存在时再搜索PUBLIC账号的schema path

如下特定用户的schema path可通过**ALTER USER**语句变更PUBLIC账号的schema path使用ALTER USER PUBLIC SCHEMA PATH语句变更与用户的当前schema path一起变更schema时使用CURRENT PATH语句进行变更

- 变更u1用户的schema path

```
gSQL> ALTER USER u1 SCHEMA PATH ( s3, s2, s1 );


User altered.
```

- 变更PUBLIC账号的schema path

```
gSQL> ALTER USER PUBLIC SCHEMA PATH ( s1, s2, s3 );


User altered.
```

- 使用CURRENT PATH变更包含当前schema path的schema path

```
gSQL> ALTER USER u1 SCHEMA PATH ( s4, CURRENT PATH );


User altered.
```

使用**CREATE USER**语句创建用户时自动确定schema path之后使用**CREATE SCHEMA**语句创建的用户SCHEMA不会自动包含于用户的schema path中因此必要时需使用**ALTER USER**语句手动添加到schema path中详细说明参考各个语句

PUBLIC Schema

PUBLIC SCHEMA是所有用户均可创建对象的通用SCHEMA如下没有SCHEMA的用户未指定SCHEMA名称而创建表时此表的SCHEMA为PUBLIC SCHEMA

- 创建没有SCHEMA的用户u1

```
gSQL> CREATE USER u1 IDENTIFIED BY u1_password WITHOUT SCHEMA;
gSQL> GRANT CREATE SESSION TO u1;
gSQL> GRANT CREATE OBJECT ON TABLESPACE mem_data_tbs TO u1;
```

- 在u1用户下创建表

```
% gsql u1 u1_password
gSQL> CREATE TABLE t1 ( id INTEGER );
```

- 以上语句与以下拥有相同意义

```
gSQL> CREATE TABLE public.t1 ( id INTEGER );
```

以上例子中u1用户创建的t1表的SCHEMA为PUBLICPUBLIC SCHEMA是创建数据库时自动创建的built-in SCHEMA赋予与如下语句意义相同的权限使所有用户均可在schema内创建对象

```
gSQL> GRANT CREATE TABLE, CREATE VIEW, CREATE INDEX,  CREATE SEQUENCE, ADD
CONSTRAINT
         ON SCHEMA PUBLIC
         TO PUBLIC;
```

即作为通用schema的PUBLIC SCHEMA不同于表示所有用户的PUBLIC账号以上语句中给表示所有用户的PUBLIC账号（语句中的TO PUBLIC）赋予了可在PUBLIC SCHEMA（语句中的ON SCHEMA PUBLIC）中创建对象的权限所有用户可以创建并操作表但其他用户进行查询PUBLIC SCHEMA中创建的表等操作时需要有对应表的相关权限

以下为GRANT语句使用PUBLIC账号与PUBLIC SCHEMA的示例

- 给PUBLIC账号赋予可SELECT u1.t1表的权限
- 所有用户均可查询u1.t1表

```
gSQL> GRANT SELECT ON TABLE u1.t1 TO PUBLIC;
```

- 给u1用户赋予可SELECT PUBLIC SCHEMA内的所有表的权限
- u1用户可以查询PUBLIC SCHEMA内的所有表

```
gSQL> GRANT SELECT TABLE ON SCHEMA PUBLIC TO u1;
```

User与Schema的使用示例

例如一名数据库管理员与多名开发人员共同使用一个SCHEMA时可通过以下SQL语句进行控制

以下为创建管理SCHEMA our_schema的mgr_user用户与使用our_schema开发应用程序的多个app_user1, app_user2, app_user3用户的示例

如下创建mgr_user用户与our_schema SCHEMA

- 创建mgr_user与our_schema
- mgr_user为our_schema的所有者

```
gSQL> CREATE USER mgr_user IDENTIFIED BY mgr_user WITH SCHEMA our_schema;

User created.
```

- 给mgr_user赋予相关权限

```
gSQL> GRANT ALL PRIVILEGES ON DATABASE TO mgr_user;

Grant succeeded.

gSQL> COMMIT;

Commit complete.
```

如下创建多个app_user 用户

app_user用户对our_schema仅有SELECT与DML权限

- 创建没有schema的多个app_user

```
gSQL> CREATE USER app_user1 IDENTIFIED BY app_user1 WITHOUT SCHEMA;

User created.

gSQL> CREATE USER app_user2 IDENTIFIED BY app_user2 WITHOUT SCHEMA;

User created.

gSQL> CREATE USER app_user3 IDENTIFIED BY app_user3 WITHOUT SCHEMA;

User created.

gSQL> COMMIT;

Commit complete.
```

- 给多个app_user赋予访问权限

```
gSQL> GRANT CREATE SESSION ON DATABASE TO app_user1, app_user2, app_user3;

Grant succeeded.
```

```
gSQL> COMMIT;


Commit complete.
```

- 给多个app_user仅赋予对our_schema的读/写权限

```
gSQL> GRANT SELECT TABLE, INSERT TABLE, UPDATE TABLE, DELETE TABLE ON
SCHEMA our_schema TO app_user1, app_user2, app_user3;


Grant succeeded.


gSQL> COMMIT;


Commit complete.
```

- 将多个app_user的SCHEMA_PATH指定为our_schema

```
gSQL> ALTER USER app_user1 SCHEMA PATH ( our_schema );


User altered.


gSQL>  ALTER USER app_user2 SCHEMA PATH ( our_schema );


User altered.


gSQL> ALTER USER app_user3 SCHEMA PATH ( our_schema );
```

```
User altered.

gSQL> COMMIT;

Commit complete.
```

通过如上操作mgr_user对our_schema拥有创建删除变更对象的DDL权限但多个app_user用户对包含在our_schema中的表仅可执行读/写操作

如下mgr_user用户可执行创建表等管理操作

```
gSQL> \connect mgr_user mgr_user
gSQL> CREATE TABLE t1 ( c1 INTEGER );

Table created.

gSQL> INSERT INTO t1 VALUES ( 1 );

1 row created.

gSQL> INSERT INTO t1 VALUES (2), (3);

2 rows created.

gSQL> COMMIT;
```

```
Commit complete.
```

如下多个app_user未指定SCHEMA名并对our_schema内的表仅有读/写权限但不能创建或删除对象

```
gSQL> \connect app_user1 app_user1
gSQL> SELECT * FROM t1;

C1
--
 1
 2
 3

3 rows selected.

gSQL> INSERT INTO t1 VALUES (4);

1 row created.
gSQL> COMMIT;
Commit complete.

gSQL> DROP TABLE t1;

ERR-42000(16208): insufficient privileges
```

3.6 Tablespace

Tablespace相关语句

详细内容参考如下链接

- 创建Tablespace
  - **CREATE TABLESPACE**
  - **CREATE MEMORY DATA TABLESPACE**
  - **CREATE MEMORY TEMPORARY TABLESPACE**
  - **CREATE DISK DATA TABLESPACE**
- 删除Tablespace: **DROP TABLESPACE**
- 变更Tablespace
  - **ALTER TABLESPACE**
  - **ALTER TABLESPACE name RENAME TO**
  - **ALTER TABLESPACE name BACKUP**
  - **ALTER TABLESPACE name [ONLINE|OFFLINE]**
- 增加删除变更构成Tablespace的文件
  - **ALTER TABLESPACE name ADD [DATAFILE|MEMORY]**
  - **ALTER TABLESPACE name DROP [DATAFILE|MEMORY]**
  - **ALTER TABLESPACE name RENAME DATAFILE**
  - **ALTER DATABASE DATAFILE AUTOEXTEND**

通过以下视图可查询表空间对象相关信息

对象集合	视图	说明
DICTIONARY_SCHEMA	USER_TABLESPACES	用户可访问的表空间信息

Table 3-13 表空间对象相关信息

Tablespace概念

表空间为逻辑概念由一个以上的物理共享内存构成是存储表索引等数据的空间如下图所示存储在表空间的表索引等物理对象可以分散在多个共享内存存储可通过增加共享内存扩展表空间

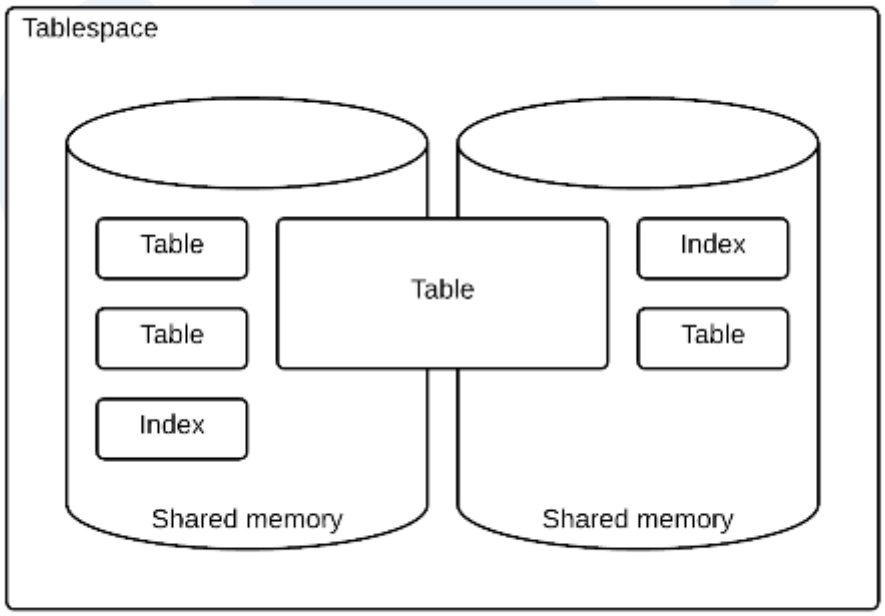


Figure 3-6 Concept of tablespace

表空间根据存储数据的类型可分为以下三种

- DATA TABLESPACE
    - 存储表logging索引等需要持久保存管理的数据表空间
- TEMPORARY TABLESPACE
    - 存储没有日志的索引和语句处理过程中创建的hashsort等易失性数据的临时表空间
- UNDO TABLESPACE
    - 存储管理事务回滚所需的数据变更信息的表空间

为了持久性的管理数据存储在数据表空间的表索引(LOGGING)等会生成Redo Log等相反存储在临时表空间的不记录日志的索引不生成Redo Log因此不记录日志的索引不会记录变更内容而在重启系统时以表数据为基准重构并维持原有的索引功能

表空间为存储SQL schema对象的物理存储空间生成表索引等时可以指定特定表空间即一个表与对应此表的索引为了表的约束条件而生成的索引等可以分别存储在不同的表空间生成对象时指定表空间的语句参考下列语句

- **CREATE TABLE**
- **CREATE INDEX**
- **ALTER TABLE name ADD CONSTRAINT**

如果创建表索引等对象时不指定表空间则使用用户的默认表空间用户的默认表空间设置方法参考如下语句

- **CREATE USER**
- **ALTER USER**

表空间相关详细内容参考**表空间管理**

3.7 Table

表相关语句

创建删除变更表的语句如下

- 创建表
  - **CREATE TABLE**
  - **CREATE TABLE AS SELECT**
  - **CREATE GLOBAL TEMPORARY TABLE**
  - **CREATE IMMUTABLE TABLE**
- 删除表
  - **DROP TABLE**
  - **TRUNCATE TABLE**
- 变更表
  - **ALTER TABLE**
  - **ALTER TABLE name RENAME TO**
  - **ALTER TABLE name STORAGE**
- 在表中增加删除变更column
  - **ALTER TABLE name ADD COLUMN**
  - **ALTER TABLE name SET UNUSED COLUMN**
  - **ALTER TABLE name ALTER COLUMN**
  - **ALTER TABLE name RENAME COLUMN**
- 增加删除变更表的约束条件

    - **ALTER TABLE name ADD CONSTRAINT**
    - **ALTER TABLE name DROP CONSTRAINT**
    - **ALTER TABLE name ALTER CONSTRAINT**
- 增加删除表的附加日志
    - **ALTER TABLE name ADD SUPPLEMENTAL LOG**
    - **ALTER TABLE name DROP SUPPLEMENTAL LOG**
- 表统计信息
    - **18.84 ANALYZE TABLE**
- 表回收站管理
    - **FLASHBACK TABLE**
    - **PURGE**

通过以下视图可查询表对象相关信息

对象集合	视图	说明
DICTIONARY_SCHEMA	**ALL_ALL_TABLES**	用户可访问的表信息
	ALL_COL_COMMENTS	用户可访问的column的注释信息
	ALL_CONSTRAINTS	用户可访问的约束条件信息
	ALL_CONS_COLUMNS	用户可访问的约束条件的column信息
	ALL_TABLES	用户可访问的表信息
	ALL_TAB_COLS	用户可访问的column信息

对象集合	视图	说明
	ALL_TAB_COLUMNS	用户可访问的column信息
	ALL_TAB_COMMENTS	用户可访问的表的注释信息
	ALL_TAB_IDENTITY_COLS	用户可访问的表的identity column信息
	USER_ALL_TABLES	用户拥有的表信息
	USER_COL_COMMENTS	用户拥有的column的注释信息
	USER_CONSTRAINTS	用户拥有的约束条件信息
	USER_CONS_COLUMNS	用户拥有的约束条件的column信息
	USER_RECYCLEBIN	用户拥有的回收站对象信息
	USER_TABLES	用户拥有的表信息
	USER_TAB_COLS	用户拥有的column信息
	USER_TAB_COLUMNS	用户拥有的column信息
	USER_TAB_COMMENTS	用户拥有的表的注释信息
	USER_TAB_IDENTITY_COLS	用户拥有的表的identity column信息
INFORMATION_SCHEMA	COLUMNS	用户可访问的column信息
	CONSTRAINT_COLUMN_USAGE	用户可访问的约束条件的column信息

对象集合	视图	说明
	CONSTRAINT_TABLE_USAGE	用户可访问的约束条件的表信息
	KEY_COLUMN_USAGE	用户可访问的key约束条件的column信息
	TABLES	用户可访问的表信息
	TABLE_CONSTRAINTS	用户可访问的约束条件信息

Table 3-14 表对象相关信息

表概念

表是构成数据库的最基本的对象SQL标准中将表定义为base table视图定义为viewed table

表由字段（column）与记录（row）组成一个表由多条记录组成每条记录的字段数和顺序一致一个表由一个以上的字段组成字段有自己的名称但记录没有记录的顺序未必与插入数据的顺序一致特定记录与特定字段的交叉点的数据叫value字段是同一种数据类型的value的集合

表的每个字段在表内有区分于其他字段的唯一名称拥有符合value特性的数据类型数据类型相关内容参考**数据类型**为了保证数据的完整性可以增加约束条件约束条件相关内容参考**CREATE TABLE**与**ALTER TABLE name ADD CONSTRAINT**语句在表中可通过创建索引提高查询语句的性能索引相关内容参考**13.8 Index**

以下为通过**CREATE TABLE**语句创建lineitem表的例子

```
CREATE TABLE lineitem
(
    l_orderkey      INTEGER    NOT NULL
  , l_partkey       INTEGER    NOT NULL
  , l_suppkey       INTEGER    NOT NULL
  , l_linenumber    INTEGER    NOT NULL
  , l_quantity      NUMERIC(12,2)
  , l_extendedprice NUMERIC(12,2)
  , l_discount      NUMERIC(12,2)
  , l_tax           NUMERIC(12,2)
  , l_returnflag    CHAR(1)    NOT NULL  DEFAULT 'F'
  , l_linestatus    CHAR(1)
  , l_shipdate      DATE
  , l_commitdate    DATE
  , l_receiptdate   DATE
  , PRIMARY KEY (l_orderkey, l_linenumber) INDEX lineitem_pk_idx
TABLESPACE mem_temp_tbs
) TABLESPACE mem_data_tbs;
```

上述示例中lineitem表中定义了多个column与约束条件以约束条件定义column时对l_orderkeyl_partkeyl_suppkeyl_linenumber等字段定义了NOT NULL约束条件组合l_orderkeyl_linenumber两个字段定义了PRIMARY KEY约束条件与字段一起定义的约束条件为in-line约束条件与字段分开定义的约束条件为out-line约束条件

为指定Column的默认值在l_returnflag字段使用DEFAULT子句声明'F'值与PRIMARY KEY约束条件同时生成的索引单独命名为lineitem_pk_idx并将存储此索引的表空间指定为mem_temp_tbs另

外存储表的物理存储空间使用mem_data_tbs表空间

以下为使用**ALTER TABLE name ADD CONSTRAINT**语句新增表的约束条件的示例

```
ALTER TABLE lineitem
      ADD CONSTRAINT lineitem_unique_all_key
      UNIQUE( l_orderkey ASC, l_partkey DESC, l_suppkey DESC, l_linenumber
ASC);
```

如上在lineitem表上新增了UNIQUE约束条件并对自动生成的索引字段指定了ASC/DESC排序

以下为使用**ALTER TABLE name ADD COLUMN**语句新增字段的示例

```
ALTER TABLE lineitem ADD COLUMN
(
    l_shipinstruct  CHAR(25)
  , l_shipmode      CHAR(10)
  , l_comment       VARCHAR(44)
);
```

上述示例中增加了多个字段增加字段时可同时指定in-line约束条件或默认值

以下为使用**CREATE INDEX**语句在表中创建索引的示例

```
CREATE INDEX lineitem_idx_shipdate ON lineitem( l_shipdate ASC NULLS
LAST );
```

如上在常用作查询条件的l_shipdate字段创建索引字段的排序指定为升序（ASC）并存在NULL时

指定放到最后的位置

增加/删除/更新数据的相关DML语句用法参考**Data Manipulation Language**查询表数据的相关SELECT语句用法参考**Data Query Language**与**SELECT**

Global Temporary Table

所有用户均共享表的定义但以各个会话区分使用数据的临时表的一种

执行CREATE GLOBAL TEMPORARY TABLE命令时生成表的定义在会话第一次执行对应表的INSERT命令时生成物理存储空间（segment）并从属于会话当会话终止时释放在该会话生成并分配给global temporary table的所有存储空间生成时根据是否指定选项决定是否TURNCATE COMMIT或ROLLBACK时剩余的数据

除集群相关语句外支持一般表提供的所有DDL与DMLDDL命令向当前会话使用的global temporary table返回错误但global temporary table 的TRUNCATE TABLE命令仅适用于当前会话因此即使其他会话正在使用也不会返回错误

Global temporary table仅可在temporary tablespace定义因此不记录用于重启恢复的日志（redo log）但为了MVCC与rollback而记录之前的值（undo log）记录undo log的空间使用**TEMP_UNDO_ENABLED**参数从system undo tablespace或system temp tablespace中进行选择

TEMP_UNDO_ENABLED值为1时与事务的undo relation分开在session的temp undo relation记录undo log事务仅执行global temporary table的DML时不记录事务记录与提交日志因此提升DML性能

释放会话使用的空间后默认返回至对应表空间重新分配空间时从表空间进行分配表空间分配空

间后返回的过程与其他会话维持同时性以及空间分配释放的成本较高因此通过**TEMP_SEGMENT_CACHE_SIZE**参数不需要向表空间返回会话中使用后释放的空间可直接在会话中重新使用

即TEMP_SEGMENT_CACHE_SIZE值为0（默认值）时使用后释放的空间立即返回至表空间如果该值大于1（最大4294967295）返回空间时会话可重新使用指定数量的空间

会话不再使用global temporary table时使用**ALTER SESSION CLEANUP GLOBAL TEMPORARY SEGMENT POOL;** 语句一次性处理segment cache的segment

以下为使用**CREATE GLOBAL TEMPORARY TABLE**语句创建global temporary table的示例

```
CREATE  GLOBAL TEMPORARY TABLE SESSION_TABLE1(
        COL1    CHAR(10)
       ,COL2    VARCHAR2(20)
       ,COL3    NUMBER(10)
)   ON  COMMIT  DELETE ROWS;
```

与普通表相同生成的global temporary table相关信息可通过DICTIONARY表或视图进行查询

Table Function Derived Table

Table function derived table是由执行table function返回的结果集构成的逻辑表Table function是定义为return table type的functionTable function derived table的定义遵循table function的return statement中定义的table column list的定义与一般表不同Table function derived table无法拥有index或约束条件Table function derived table的row是table function返回的结果集当配置特定表

的子集从所需的表中检索所需数据时使用Table function derived table

关于Table function概念的详细内容请参考**Stored Function**

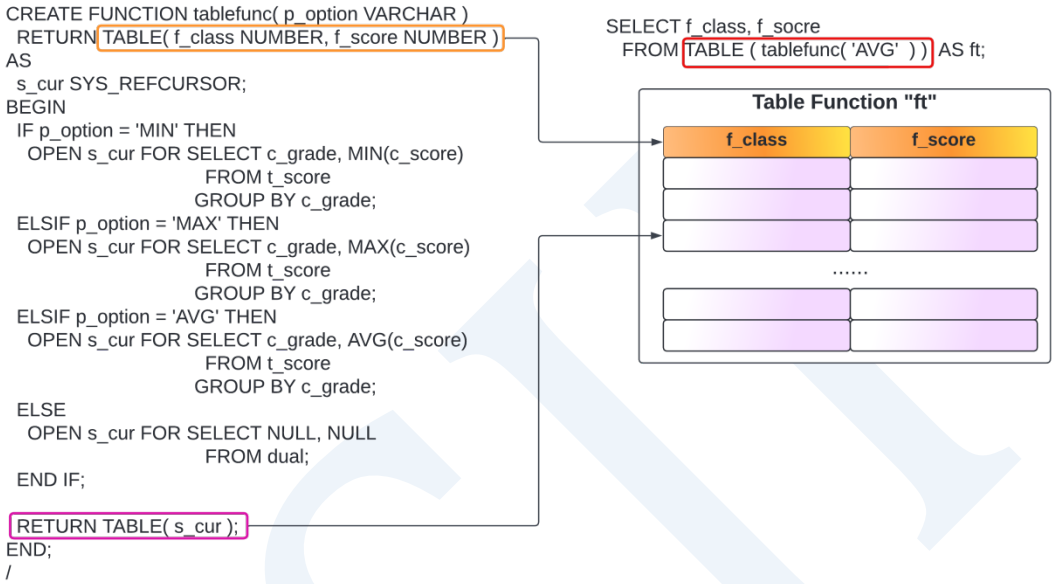


Figure 3-7 Table function derived table定义

集群表

集群环境的表的概念参考**集群表与分片**

表回收站管理

语句

- 恢复保存在回收站的对象
  - **FLASHBACK TABLE**
- 删除保存在回收站的对象
  - **PURGE**

可通过如下视图查询回收站相关信息

对象集合	视图	说明
DICTIONARY_SCHEMA	**DBA_RECYCLEBIN**	数据库的所有回收站信息
	USER_RECYCLEBIN	自己拥有的回收站信息
	RECYCLEBIN	USER_RECYCLEBIN的alias

Table 3-15 回收站相关信息

说明

不立即删除表而将删除的对象保管在回收站的功能与表相联系的约束条件和索引也一起保管在回收站

回收站的概念也称为flashback drop功能可使用PURGE和FLASHBACK TABLE语句删除或恢复保管在回收站的对象

删除表后保管在回收站时变更所有与表相关的对象并保存变更的名称是BIN$unique_name 形式由数据库生成并赋予唯一值 unique_name生成为32字的字符值

要恢复保管在回收站的表时与表相关的约束条件和索引恢复为删除之前的名称如果有与删除之前的对象名相同的名称时恢复为保存在回收站的名称

要使用回收站功能需启用**RECYCLEBIN**参数该参数可通过ALTER SESSION和ALTER SYSTEM更改ALTER SYSTEM拥有DEFERRED属性默认值为FALSE

```
gSQL> ALTER SESSION SET RECYCLEBIN = ON;

Session altered.

gSQL> ALTER SYSTEM SET RECYCLEBIN = ON DEFERRED;

System altered.
```

特征

- 保管在回收站的对象并非物理删除的状态因此不会释放该对象使用的表空间的空间
- 每个用户拥有各自的回收站并可查询自己的回收站
- 删除保管在回收站的对象时如果有名称重复的表则删除最久的对象
- 恢复保管在回收站的对象时如果有名称重复的表则恢复最新的对象
- 可在DICTIONARY VIEW查询到保管在回收站的对象但在INFORMATION VIEW中无法查询
- 删除数据库的所有回收站需要有PURGE DBA_RECYCLEBIN ON DATABASE权限
- PURGE和FLASHBACK TABLE 语句可用 AUDIT SYSTEM ACTION进行audit

- 删除表空间则删除该表空间包含的回收站对象
- 删除schema则删除该schema包含的回收站对象
- 删除用户则删除该用户的回收站对象

Note:

保管在回收站的对象仅允许部分DML和DDL除以下目录外的语句均报错

- SELECT
- SELECT .. FOR UPDATE
- LOCK TABLE
- COMMENT ON TABLE name IS
- COMMENT ON COLUMN name IS
- COMMENT ON INDEX name IS
- COMMENT ON CONSTRAINT name IS
- GRANT privileges TO
- REVOKE privileges FROM
- CREATE TABLE AS SELECT
- CREATE GLOBAL TEMPORARY TABLE AS SELECT
- CREATE AUDIT POLICY
- ALTER AUDIT POLICY
- CREATE VIEW
- CREATE SYNONYM
- CREATE FUNCTION
- CREATE PROCEDURE
- ALTER FUNCTION

- ALTER PROCEDURE
- ALTER DATABASE MOVE SHARD
- ALTER DATABASE REBALANCE
- ALTER DATABASE REBALANCE EXCLUDE CLUSTER GROUP
- ALTER TABLE name REBALANCE
- ALTER TABLE name REBALANCE EXCLUDE CLUSTER GROUP
- ALTER TABLE name MOVE SHARD
- ALTER TABLE name SPLIT SHARD
- ALTER TABLE name MERGE SHARD
- ALTER TABLE name SYNCHRONIZE IDENTITY COLUMN

使用示例

使用回收站功能需要激活RECYCLEBIN参数

```
gSQL> ALTER SESSION SET RECYCLEBIN = ON;

Session altered.

gSQL> CREATE TABLE t1 ( id INTEGER PRIMARY KEY, name VARCHAR(32) );

Table created.

gSQL> DROP TABLE t1;
```

```
Table dropped.

gSQL> CREATE TABLE t1 ( id INTEGER PRIMARY KEY, name VARCHAR(32) );

Table created.

gSQL> DROP TABLE t1;

Table dropped.

gSQL> COMMIT;

Commit complete.

gSQL> SELECT OBJECT_NAME, ORIGINAL_NAME, OBJECT_TYPE, DROPPED_TIME FROM
USER_RECYCLEBIN;

OBJECT_NAME                          ORIGINAL_NAME        OBJECT_TYPE
DROPPED_TIME
---------------------------------- ------------------- ----------- ----
----------------------
BIN$8981D28E172C11EAA7C5D51B86D72AB6 T1                   TABLE
2019-12-05 15:57:44.120000
BIN$8981D2C0172C11EAA7C5D51B86D72AB6 T1_PRIMARY_KEY       CONSTRAINT
2019-12-05 15:57:44.120000
```

```
BIN$8981D2AC172C11EAA7C5D51B86D72AB6 T1_PRIMARY_KEY_INDEX INDEX

2019-12-05 15:57:44.120000

BIN$8F1E9614172C11EAA7C5D51B86D72AB6 T1                   TABLE

2019-12-05 15:57:53.540000

BIN$8F1E9650172C11EAA7C5D51B86D72AB6 T1_PRIMARY_KEY       CONSTRAINT

2019-12-05 15:57:53.540000

BIN$8F1E963C172C11EAA7C5D51B86D72AB6 T1_PRIMARY_KEY_INDEX INDEX

2019-12-05 15:57:53.540000


6 rows selected.


gSQL> PURGE TABLE t1;


Table purged.


gSQL> SELECT OBJECT_NAME, ORIGINAL_NAME, OBJECT_TYPE, DROPPED_TIME FROM

USER_RECYCLEBIN;


OBJECT_NAME                          ORIGINAL_NAME        OBJECT_TYPE

DROPPED_TIME

------------------------------------ -------------------- ----------- ----

----------------------

BIN$8F1E9614172C11EAA7C5D51B86D72AB6 T1                   TABLE

2019-12-05 15:57:53.540000

BIN$8F1E9650172C11EAA7C5D51B86D72AB6 T1_PRIMARY_KEY       CONSTRAINT
```

```
2019-12-05 15:57:53.540000

BIN$8F1E963C172C11EAA7C5D51B86D72AB6 T1_PRIMARY_KEY_INDEX INDEX

2019-12-05 15:57:53.540000


3 rows selected.


gSQL> FLASHBACK TABLE t1 TO BEFORE DROP;


Flashback complete.


gSQL> DESC T1


COLUMN_NAME TYPE         IS_NULLABLE

----------- ------------ -----------

ID          NUMBER(10,0) FALSE

NAME        VARCHAR(32)  TRUE


INDEX_NAME           TABLESPACE_NAME INDEX_TYPE IS_UNIQUE COLUMNS

-------------------- --------------- ---------- --------- -------

T1_PRIMARY_KEY_INDEX MEM_TEMP_TBS    BTREE      TRUE      ID


CONSTRAINT_NAME CONSTRAINT_TYPE ASSOCIATED_INDEX     COLUMNS

--------------- --------------- -------------------- -------

T1_PRIMARY_KEY  PRIMARY KEY     T1_PRIMARY_KEY_INDEX ID
```

```
gSQL> SELECT OBJECT_NAME, ORIGINAL_NAME, OBJECT_TYPE, DROPPED_TIME FROM
USER_RECYCLEBIN;

no rows selected.
```

3.8 Index

Index相关语句

创建删除变更索引的语句如下

- Index创建：**CREATE INDEX**
- Index删除： **DROP INDEX**
- Index变更： **ALTER INDEX**

通过以下视图可查询索引对象及其相关信息

对象集合	视图	说明
DICTIONARY_SCHEMA	**ALL_INDEXES**	用户可访问的索引信息
	ALL_IND_COLUMNS	用户可访问的索引的column信息
	USER_INDEXES	用户拥有的索引信息
	USER_IND_COLUMNS	用户拥有的索引的column信息

Table 3-16 索引对象相关信息

索引的概念

索引是与表相关的对象用于查询表时提升访问数据的性能索引以使用包含在表中一个或多个column中的数据的key值组成是独立于表的对象生成索引时数据库自动构建索引的key数据插入删除更新表的数据时自动管理索引的key数据

例如有以下语句

```
SELECT data FROM t1 WHERE id = 12345;
```

没有索引时检索表的所有row并寻找符合条件的结果如果表的row数多但符合条件的结果少时上述查询的响应效率非常低如下通过**CREATE INDEX**语句在id字段生成索引时语句优化器（optimizer）将评估full scan与index scan的成本之后基于索引查询数据以提高性能

```
CREATE INDEX t1_idx_id ON t1(id);
```

创建索引时可以组合两个以上的字段当作索引的key由两个以上的key组成的索引叫复合索引（composite index）复合索引以第一个key为准排序当第一个key值相同时以第二个key值为准排序按照这种方式根据key的数量进行排序

创建索引时可以指定升序（ASC）或降序（DESC）排序NULL值的排列顺序可以指定为NULLS FIRST或NULLS LAST参考下例

```
gSQL> CREATE TABLE t1 ( value INTEGER );


Table created.
```

```
gSQL> INSERT INTO t1 VALUES (1), (NULL), (3), (2), (NULL);

5 rows created.

gSQL> CREATE INDEX idx1 ON t1 ( value ASC NULLS LAST );

Index created.

gSQL> CREATE INDEX idx2 ON t1 ( value DESC NULLS FIRST );

Index created.

gSQL> SELECT /*+ INDEX(t1, idx1) */ * FROM t1;

VALUE
-----
    1
    2
    3
 null
 null

5 rows selected.

gSQL> SELECT /*+ INDEX(t1, idx2) */ * FROM t1;
```

```
VALUE
-----
 null
 null
    3
    2
    1

5 rows selected.
```

上述示例中idx1索引指定为升序（ASC）NULLS LASTidx2索引指定为降序（DESC）NULLS FIRST执行相同语句查询表时可以设置不同的索引hint并使用该索引查询所有row使用idx1索引时以升序排序并且NULL值在最后面而使用idx2索引时降序排序并且NULL值在最前面

UNIQUE概念

创建索引时可以生成UNIQUE或non-unique索引生成UNIQUE索引时如果key值不唯一则报错

UNIQUE索引与UNIQUE约束条件允许使用NULL值为key值

包含NULL值时是否为UNIQUE的truth 表如下即key为一个时可以有多个null值

value1	value2	是否UNIQUE
1	1	false
1	2	true

value1	value2	是否UNIQUE
1	null	true
null	null	true

Table 3-17 两个值的UNIQUE与否

由两个以上的key构成的UNIQUE索引或UNIQUE约束条件的所有或部分值可以是null复合key中包含null时是否为UNIQUE的truth table如下

row1	row2	UNIQUE与否
(1, 1)	(1, 1)	false
(1, 1)	(1, 2)	true
(1, null)	(1, null)	false
(1, null)	(2, null)	true
(null, null)	(null, null)	true

Table 3-18 复合key的UNIQUE与否

SQL标准中如下定义UNIQUE

> Note:
>
> - 截止SQL1999的UNIQUE定义
>
> If there are no two rows in T such
>
> that the value of each column in one row is non-null and

is equal to the value of the corresponding column in the other row according to Subclause 8.2, ‘‘<comparison predicate>’’,

then the result of the <unique predicate> is true;

otherwise, the result of the <unique predicate> is false.

• SQL2003之后的UNIQUE定义

If there are no two rows in T such

that the value of each column in one row is non-null and

is not distinct from the value of the corresponding column in the other row,

then the result of the <unique predicate> is True;

otherwise, the result of the <unique predicate> is False.

SUNDB遵守SQL2003以后的SQL2011标准SQL标准的UNIQUE定义如下表根据复合key的UNIQUE与否定义

row1	row2	~ SQL1999	SQL2003 ~
(1, 1)	(1, 1)	false	false
(1, 1)	(1, 2)	true	true
(1, null)	(1, null)	ture	false
(1, null)	(2, null)	true	true
(null, null)	(null, null)	true	true

Table 3-19 SQL标准的复合key的UNIQUE与否

不同数据库对定义UNIQUE时遵守不同的SQL标准

- 遵守SQL2003之后标准的数据库：Oracle, SQL Server
- 遵守SQL1999标准的数据库：Postgres, MySQL

3.9 View

View相关语句

创建删除变更的视图的语句如下

- View创建：**CREATE VIEW**
- View删除：**DROP VIEW**
- View变更：**ALTER VIEW**

通过以下视图可查询视图对象及其相关信息

对象集合	视图	说明
DICTIONARY_SCHEMA	**ALL_VIEWS**	用户可访问的视图信息
	ALL_DEPENDENCIES	与用户可访问的视图相关的对象信息
	USER_VIEWS	用户拥有的视图信息
	USER_DEPENDENCIES	与用户拥有的视图相关的对象信息
INFORMATION_SCHEMA	**VIEWS**	用户可访问的视图信息
	VIEW_TABLE_USAGE	生成视图时使用的表信息
	VIEW_ROUTINE_USAGE	生成视图时使用的stored function信息

Table 3-20 视图对象相关信息

View概念

表是存储数据的物理结构而视图是由查询语句组成的逻辑结构SQL标准定义为viewed table视图的查询语句的使用方法与表相同

视图有以下优点

- 可使仅查询表的部分信息并限制数据访问
- 将经常使用的复杂语句做成一个视图简化查询的复杂度
- 通过更改视图的column名称更新数据等可提供与表不同形式的数据
- 可使基于视图的应用程序不受表结构变更的影响

如下通过**CREATE VIEW**语句创建的视图在实际执行过程中被代替为in-line视图

- 创建视图

```
CREATE VIEW v1 ( v_id, v_sum )
AS
SELECT l_partkey, SUM( l_quantity )
  FROM lineitem
 GROUP BY l_partkey;
```

- 查询视图

```
SELECT v_id, v_sum
  FROM v1
 WHERE v_sum > 1000;
```

- 解析视图

```
SELECT v_id, v_sum
  FROM ( SELECT l_partkey, SUM( l_quantity )
           FROM lineitem
          GROUP BY l_partkey
       ) v1 ( v_id, v_sum )
 WHERE v_sum > 1000;
```

如下在SELECT语句中使用表示所有column的asterisk（*）创建v1视图时则在视图访问的t1表中增加新的column addr后查询v1仍然可以查询到包括增加的column在内的所有column

```
gSQL> CREATE TABLE t1 ( id INTEGER, name VARCHAR(128) );

Table created.

gSQL> INSERT INTO t1 VALUES ( 1, 'leekmo' );

1 row created.
```

- 使用Asterisk (*)创建视图

```
gSQL> CREATE VIEW v1 AS SELECT * FROM t1;
```

```
View created.
```

- 查询视图

```
gSQL> SELECT * FROM v1;


ID NAME
-- ------
 1 leekmo


1 row selected.
```

- 在视图引用的表增加column

```
gSQL> ALTER TABLE t1 ADD COLUMN addr VARCHAR(1024) DEFAULT 'N/A';


Table altered.
```

- 增加column后查询视图

```
gSQL> SELECT * FROM v1;


ID NAME   ADDR
-- ------ ----
 1 leekmo N/A
```

```
1 row selected.
```

但如上述使用asterisk（*）创建的视图后在变更表结构时可能引起应用程序的变更因此不推荐使用

3.10 Sequence

Sequence相关语句

创建删除变更使用序列的语句如下

- Sequence创建：**CREATE SEQUENCE**
- Sequence删除：**DROP SEQUENCE**
- Sequence变更：**ALTER SEQUENCE**
- Sequence使用：**NEXTVALCURRVAL**

通过以下视图可查询序列对象及其相关信息

对象集合	视图	说明
DICTIONARY_SCHEMA	**ALL_SEQUENCES**	用户可访问的序列信息
	USER_SEQUENCES	用户拥有的序列信息
INFORMATION_SCHEMA	**SEQUENCES**	用户可访问的序列信息

Table 3-21 序列对象相关信息

Sequence概念

序列为自动生成序列号的对象标准SQL中定义为sequence generator序列自动管理unique key或primary key一个序列可用于多个表

以下为使用一个序列对象自动创建对应id column的值并用于多个表的使用示例

```
gSQL> CREATE SEQUENCE seq;

Sequence created.

gSQL> INSERT INTO t1 (id, name) VALUES ( seq.NEXTVAL, 'leekmo' );

1 row created.

gSQL> INSERT INTO t2 (id, addr) VALUES ( seq.CURRVAL, 'Seoul, Korea' );

1 row created.

gSQL> SELECT * FROM t1;

ID NAME
-- ------
 1 leekmo
```

```
1 row selected.


gSQL> SELECT * FROM t2;


ID ADDR
-- ------------
 1 Seoul, Korea


1 row selected.
```

上述示例中使用seq.NEXTVAL函数自动创建t1表的id column的下一个编号再使用seq.CURRVAL函数把相同的值用作t2表的id column的值创建序列时可指定自动创建的编号的开始值递增值最小值最大值是否循环cache值等详细内容参考**CREATE SEQUENCE**语句

与序列有类似功能的identity column是在一个表中自动生成编号的column使用方法如下

```
gSQL> CREATE TABLE t1 ( id INTEGER GENERATED ALWAYS AS IDENTITY, name
VARCHAR(128) );


Table created.


gSQL> INSERT INTO t1 (name) VALUES ( 'leekmo' );


1 row created.


gSQL> INSERT INTO t1 (name) VALUES ( 'mkkim' );
```

```
1 row created.

gSQL> INSERT INTO t1 (name) VALUES ( 'xcom73' );

1 row created.

gSQL> SELECT * FROM t1;

ID NAME
-- ------
 1 leekmo
 2 mkkim
 3 xcom73

3 rows selected.
```

如上创建t1表时id column创建为identity column执行INSERT语句时id column值输入为identity column自动生成的值Identity column的详细内容参考**CREATE TABLE**语句的**<identity column specification>**部分

序列与identity column均能生成序号但有以下差异

- 序列为SQL schema对象但identity column为表的一个字段
- 序列可以用于多个表但identity column只能用于一个表

创建序列后可通过**NEXTVAL**或**CURRVAL**函数使用序列号序列号与事务分开单独创建不受事务的提交或回滚的影响

```
gSQL> INSERT INTO t1(id) VALUES( seq.NEXTVAL );

1 row created.

gSQL> SELECT id FROM t1;

ID
--
 1

1 row selected.

gSQL> ROLLBACK;

Rollback complete.

gSQL> INSERT INTO t1(id) VALUES( seq.NEXTVAL );

1 row created.

gSQL> SELECT id FROM t1;
```

```
ID

--

 2


1 row selected.
```

如上第一个INSERT语句中通过seq.NEXTVAL函数生成1之后回滚该事务后使用的seq.NEXTVALDE的值与事务无关之后生成的值为重新增加的2

序列号只能在以下位置中使用

- 最上层SELECT语句的select list值
  - SELECT seq.NEXTVAL FROM dual;
- INSERT .. SELECT语句的select list值
  - INSERT INTO t1(id) SELECT seq.NEXTVAL FROM daul;
- INSERT .. VALUES语句的输入值
  - INSERT INTO t1(id) VALUES (seq.NEXTVAL);
- UPDATE语句的SET值
  - UPDATE t1 SET id = seq.NEXTVAL;

除以上位置外不能使用序列号不能位于subqueryaggregation函数的参数WHEREDISTINCTGROUP BYHAVINGORDER BY等语句

集群的序列

以集群系统的结构使用SUNDB时在内部使用global序列对象Global序列对象生成在整个集群系统

共同使用的序列值pool后在各个成员节点调用NEXTVAL时分配与cache大小相同大小的方式即若特定节点分配到了20个序列值则从该值下一个值开始分配给其他节点成员节点将global序列对象分配的序列值加载在自身的local cache后返回其NEXTVAL调用结果直到耗尽其序列值

与原有的单机版数据库用序列相比Global序列对象有如下特征及约束事项

- 无法在ALTER SEQUENCE语句使用INCREMENT BY选项变更符号（可变更大小）
- 使用CYCLE选项时根据整个序列池的大小成员节点之间可返回重复值因此需要CYCLE选项时考虑到INCREMENT BY及CACHE SIZE集群成员节点数量生成大小充足的序列池
- 非故障情况下以特定成员节点为准返回的序列值也有可能不连续当然仅有一个成员节点调用NEXTVAL时可获取连续的序列值
- NOCACHE时CACHE SIZE为1因此不在Local Cache加载额外的序列值此时每次调用NEXTVAL时均从Global序列对象分配到1个序列因此增加网络成本性能会大幅下降
- 创建及变更删除序列默认以AUTO COMMIT操作
- 使用ALTER语句变更CACHE及INCREMENT大小时将重置加载在所有节点的Local Cache的序列值即后续调用NEXTVAL时需要重新从Global序列对象分配到新的序列集

3.11 Synonym

Synonym相关语句

创建删除Synonym的语句如下

- 创建Synonym：**CREATE SYNONYM**
- 删除Synonym：**DROP SYNONYM**

通过以下视图可查询Synonym对象及其相关信息

对象集合	视图	说明
DICTIONARY_SCHEMA	**ALL_SYNONYMS**	所有的同义词信息
	USER_SYNONYMS	用户拥有的同义词信息

Table 3-22 Synonym对象相关信息

Synonym概念

同义词是以下对象的别名

- Table
- View
- Sequence

- Stored procedure
- Stored function
- 其他synonym

可在SELECTINSERTUPDATEDELETELOCK TABLEGRANTREVOKECOMMENT语句中使用别名

同义词的优势体现在即使变更其底层对象的schema也不需要修改应用程序重新定义同义词即可又可以隐藏对象的实际名称与所有者可强化数据库的安全性可变更缩短对象的名称提高可用性

Synonym分为private synonym与public synonymprivate synonym为schema对象public synonym为non-schema对象

以下通过表的private synonym与public synonym的创建和使用示例说明其概念

```
gSQL> \CONNECT u1 u1
gSQL> CREATE TABLE u1.t1 (col1 INTEGER );
gSQL> INSERT INTO u1.t1 VALUES(1);
gSQL> COMMIT;
```

Private Synonym

Private Synonym为Schema对象创建时若省略Schema名称则使用执行语句的用户的默认schema名称

```
gSQL> \CONNECT u2 u2
gSQL> CREATE SYNONYM u2.syn1 FOR u1.t1;
```

```
Synonym created.


gSQL> SELECT * FROM u2.syn1;


ERR-42000(16254): lacks privilege (SELECT ON TABLE "U1"."T1")
```

Synonym只是别名因此即使是创建synonym的用户如果没有相关权限则无法访问默认对象u1.t1

```
gSQL> \CONNECT u1 u1
gSQL> GRANT SELECT ON TABLE u2.syn1 TO u2;
gSQL> \CONNECT u2 u2
gSQL> SELECT * FROM u2.syn1;
COL1
----
   1


1 row selected.


gSQL> SELECT * FROM u1.t1;
COL1
----
   1


1 row selected.


gSQL> DROP SYNONYM u2.syn1;
```

```
Synonym dropped.
```

如上给u2用户赋予'u2.syn1'的SELECT权限相当于给u2赋予'u1.t1'的SELECT权限因此给Synonym赋予权限时需注意

Public Synonym

Public synonym为non-schema对象创建及删除时无法指定schema名称

```
gSQL> \CONNECT u2 u2
gSQL> CREATE PUBLIC SYNONYM pubSyn1 FOR u1.t1;

Synonym created.

gSQL> SELECT * FROM pubSyn1;

ERR-42000(16254): lacks privilege (SELECT ON TABLE "U1"."T1")
```

Public synonym无所有者所有用户均可以使用但若没有默认对象的访问权限则无法访问默认对象

```
gSQL> \CONNECT u1 u1
gSQL> GRANT SELECT ON TABLE pubSyn1 TO u2;
gSQL> \CONNECT u2 u2
gSQL> SELECT * FROM pubSyn1;
```

```
COL1

----

   1


1 row selected.


gSQL> SELECT * FROM u1.t1;

COL1

----

   1


1 row selected.


gSQL> DROP SYNONYM pubSyn1;


Synonym dropped.
```

3.12 Stored Procedure

Stored Procedure相关语句

创建删除变更stored procedure的语句如下

- 创建Stored procedure：**CREATE PROCEDURE**
- 删除Stored procedure：**DROP PROCEDURE**
- 变更Stored procedure：**ALTER PROCEDURE**

可通过如下视图查询stored procedure对象及其相关信息

对象集合	视图	说明
DICTIONARY_SCHEMA	**ALL_ARGUMENTS**	用户可访问的procedurefunction的argument信息
	ALL_DEPENDENCIES	与用户可访问的procedurefunction相关的对象信息
	ALL_PROCEDURES	用户可访问的procedurefunction对象信息
	ALL_SOURCE	用户可访问的procedurefunction的source text信息
	USER_ARGUMENTS	用户拥有的procedurefunction的argument信息

对象集合	视图	说明
	USER_DEPENDENCIES	与用户拥有的procedurefunction相关的对象信息
	USER_PROCEDURES	用户拥有的procedurefunction对象信息
	USER_SOURCE	用户拥有的procedurefunction的source text信息
INFORMATION_SCHEMA	PARAMETERS	用户可访问的procedurefunction的argument信息
	ROUTINES	用户可访问的procedurefunction的对象信息
	ROUTINE_ROUTINE_USAGE	用户可访问的procedurefunction参照的procedurefunction信息
	ROUTINE_SEQUENCE_USAGE	用户可访问的procedurefunction参照的sequence信息
	ROUTINE_TABLE_USAGE	用户可访问的procedurefunction参照的tableview信息

Table 3-23 Stored procedure对象相关信息

Stored Procedure概念

Stored procedure是procedure形式的persistent stored module的一种与其他schema-level

database对象相同以schema为单位定义并管理由于是procedure形式因此不定义返回值通过CALL语句或其他stored procedure或stored function内的直接调用进行使用

Stored procedure相关详细说明参考**Schema-level Procedure**

Stored procedure的使用方法如下

```
CREATE OR REPLACE PROCEDURE PROC1( A1 INTEGER, A2 INTEGER )
IS
  V1 INTEGER;
BEGIN
  SELECT COUNT(*)
    INTO V1
    FROM T1
    WHERE T1.I1 >= A1 AND T1.I1 <= A2;
  DBMS_OUTPUT.PUT_LINE( 'V1 = ' || V1 );
END;
/

BEGIN
  PROC1( 2, 4 ); -- call schema-level procedure
END;
/

V1 = 3

Anonymous PL block executed.
```

3.13 Stored Function

Stored Function相关语句

创建删除变更stored function的语句如下

- 创建Stored function：**CREATE FUNCTION**
- 删除Stored function ：**DROP FUNCTION**
- 变更Stored function：**ALTER FUNCTION**

可通过如下视图查询stored function对象及其相关信息

对象集合	视图	说明
DICTIONARY_SCHEMA	**ALL_ARGUMENTS**	用户可访问的procedurefunction的argument信息
	ALL_DEPENDENCIES	与用户可访问的procedurefunction相关的对象信息
	ALL_PROCEDURES	用户可访问的procedurefunction对象信息
	ALL_SOURCE	用户可访问的procedurefunction的source text信息
	USER_ARGUMENTS	用户拥有的procedurefunction的argument信息

对象集合	视图	说明
	USER_DEPENDENCIES	与用户拥有的procedurefunction相关的对象信息
	USER_PROCEDURES	用户拥有的procedurefunction对象信息
	USER_SOURCE	用户拥有的procedurefunction的source text信息
INFORMATION_SCHEMA	PARAMETERS	用户可访问的procedurefunction的argument信息
	ROUTINES	用户可访问的procedurefunction的对象信息
	ROUTINE_ROUTINE_USAGE	用户可访问的procedurefunction参照的procedurefunction信息
	ROUTINE_SEQUENCE_USAGE	用户可访问的procedurefunction参照的sequence信息
	ROUTINE_TABLE_USAGE	用户可访问的procedurefunction参照的tableview信息

Table 3-24 Stored procedure对象相关信息

Stored Function概念

Stored function是function形式的persistent stored module的一种与其他schema-level database对

象相同以schema为单位定义并管理Stored function根据RETURN子句的定义有如下分类

- 如RETURN <datatype>一样定义在function中返回的返回值的datatype的function
- 如RETURN TABLE (<column_list>)一样定义返回的结果集的table type的function
    - 为table function

stored function相关详细说明参考**Schema-level Function**

RETURN <datatype> Function

在RETURN子句中定义执行function返回的expression的datatypeFunction定义要返回的结果值可以通过CALL语句直接执行与此相同的function或在stored procedure或stored function中用作expression或在SQL语句中用作expression

```
gSQL> CREATE OR REPLACE FUNCTION FUNC1( A1 INTEGER, A2 INTEGER )
RETURN INTEGER
IS
  V1 INTEGER;
BEGIN
  SELECT COUNT(*)
    INTO V1
    FROM T1
    WHERE T1.I1 >= A1 AND T1.I1 <= A2;
  RETURN V1;
END;
/
```

```
Function created.

gSQL> SELECT FUNC1( 2, 4 ) FROM DUAL;

FUNC1( 2, 4 )
-------------
            3

1 row selected.
```

Table Function

在RETURN子句中定义执行function返回的结果集的table column list这被称为table functionTable function使用select statement或cursor variable定义需要返回的结果集Table function可以在SELECT语句的FROM子句中用作table function derived table

Table function将定义的select statement或执行cursor variable的cursor query的结果集返回给上级的SELECT语句返回的结果集在SELECT语句中配置table function derived table另外作为table function的参数可以引用FROM子句中table function derived table前面列出的table的column

```
gSQL> CREATE TABLE t_score( c_grade INTEGER, c_score INTEGER );

Table created.

gSQL> INSERT INTO t_score VALUES ( 1 , 98 ) , ( 1 , 97 ) , ( 1 , 99 ),
```

```
                                ( 2 , 95 ) , ( 2 , 98 ) , ( 2 , 92 ),
                                ( 3 , 98 ) , ( 3 , 96 ) , ( 3 , 94 );


9 rows created.


gSQL> COMMIT;


Commit complete.
```

- 返回Cursor variable的执行结果

```
gSQL>
CREATE OR REPLACE FUNCTION tf_cv( p_grade INTEGER )
  RETURN TABLE( rf_grade INTEGER, rf_score INTEGER ) AS
  cv SYS_REFCURSOR;
BEGIN
  OPEN cv FOR SELECT * FROM t_score WHERE c_grade = p_grade;


  RETURN TABLE( cv );
END;
/


Function created.


gSQL> SELECT rf_grade, rf_score FROM TABLE( tf_cv( 2 ) );
```

```
RF_GRADE RF_SCORE

-------- --------

       2       95

       2       98

       2       92


3 rows selected.
```

- 返回SELECT语句的执行结果

```
gSQL>

CREATE OR REPLACE FUNCTION tf_select( p_grade INTEGER )

  RETURN TABLE( rf_grade INTEGER, rf_score INTEGER ) AS

BEGIN

  RETURN TABLE ( SELECT * FROM t_score WHERE c_grade = p_grade );

END;

/


Function created.


gSQL> SELECT rf_grade, rf_score FROM TABLE( tf_select( 2 ) );


RF_GRADE RF_SCORE

-------- --------

       2       95

       2       98
```

```
       2       92


3 rows selected.
```

3.14 Package

Package相关语句

创建变更删除package的语句如下

- 创建package：**CREATE PACKAGE**
- 创建package body：**CREATE PACKAGE BODY**
- 更改package：**ALTER PACKAGE**
- 删除package：**DROP PACKAGE**

可通过如下视图查询package对象相关信息

对象集合	视图	说明
DICTIONARY_SCHEMA	**ALL_OBJECTS**	用户可访问的object 信息
	ALL_PACKAGE_PRIVS	用户的package相关权限信息
	ALL_PACKAGE_PRIVS_MADE	为了访问package而用户赋予的权限信息
	ALL_PACKAGE_PRIVS_RECD	为了访问package而赋予用户的权限信息
	ALL_SOURCE	用户可访问的procedurefunctionpackage的source text信息

对象集合	视图	说明
	USER_OBJECTS	用户拥有的object信息
	USER_PACKAGE_PRIVS	用户拥有的package相关权限信息
	USER_PACKAGE_PRIVS_MADE	为了访问package而用户赋予的权限信息
	USER_PACKAGE_PRIVS_RECD	为了访问package而赋予用户的权限信息
	USER_SOURCE	用户拥有的procedurefunctionpackage的source text信息
INFORMATION_SCHEMA	MODULES	用户可访问的SQL-server module（package）信息
	MODULE_BODY	用户可访问的package body信息
	MODULE_BODY_MODULE_USAGE	用户可访问的package body正在使用中的其他Package信息
	MODULE_BODY_ROUTINE_USAGE	用户可访问的package body正在使用中的procedure或function信息
	MODULE_BODY_SEQUENCE_USAGE	用户可访问的package body正在使用中的序列信息

对象集合	视图	说明
	MODULE_BODY_TABLE_USAGE	用户可访问的package body正在使用中的表信息
	MODULE_MODULE_USAGE	用户可访问的package正在使用中的其他package信息
	MODULE_PRIVILEGES	用户可访问的与package相关的权限信息
	MODULE_ROUTINE_USAGE	用户可访问的package正在使用中的procedure或function
	MODULE_SEQUENCE_USAGE	用户可访问的package正在使用中的序列信息
	MODULE_TABLE_USAGE	用户可访问的package正在使用中的表信息
	ROUTINE_MODULE_USAGE	用户可访问的procedure或function正在使用中的package信息
	VIEW_MODULE_USAGE	用户可访问的view正在使用中的package信息

Table 3-25 Stored procedure对象相关信息

Package概念

Package是将有逻辑关系的PSM类型变量子程序游标异常等项目捆绑在一起的schema对象

Package经过compile过程存储到数据库作用是在其他程序（其他packageprocedure外部程序等）参考共享执行package项目

Package相关详细说明参考**PSM Packages**

以下为创建package的示例

```
CREATE TABLE emp( empno NUMBER, sal NUMBER, comm NUMBER );
Table created.


INSERT INTO emp VALUES( 3548, 6000, 1000 );
1 row created.


INSERT INTO emp VALUES( 9369, 5000, NULL );
1 row created.


INSERT INTO emp VALUES( 7294, 4000, 500 );
1 row created.


COMMIT;
Commit complete.
```

```
CREATE OR REPLACE PACKAGE emp_mgmt
IS
  PROCEDURE adjust_sal(v_flag VARCHAR, v_empno NUMBER, v_pct NUMBER);
  FUNCTION get_annual_sal(v_empno NUMBER) RETURN NUMBER;
END;
/

Package created.



CREATE OR REPLACE PACKAGE BODY emp_mgmt
IS
  PROCEDURE adjust_sal(v_flag VARCHAR, v_empno NUMBER, v_pct NUMBER) IS
  BEGIN
    IF v_flag = 'INCREASE' THEN
      UPDATE emp SET sal = sal + (sal * (v_pct / 100)) WHERE empno =
v_empno;
    ELSE
      UPDATE emp SET sal = sal - (sal * (v_pct / 100)) WHERE empno =
v_empno;
    END IF;
  END;
  FUNCTION get_annual_sal (v_empno NUMBER) RETURN NUMBER
  IS
    v_sal NUMBER;
```

```
  BEGIN
    SELECT (sal + NVL(comm,0)) * 12 INTO v_sal FROM emp WHERE empno =
v_empno;
    RETURN v_sal;
  END;
END;
/

Package created.
```

以下为使用package的示例

```
call emp_mgmt.adjust_sal('INCREASE',7369, 10);

Procedure Call complete.



SELECT emp_mgmt.get_annual_sal(7294) FROM DUAL;
EMP_MGMT.GET_ANNUAL_SAL(7294)
-----------------------------
                        54000
1 row selected.
```

4.Cluster Objects

4.1 集群系统

集群系统相关语句

详细内容参考如下链接

- 扩展集群系统
  - **CREATE CLUSTER GROUP**
  - **ALTER CLUSTER GROUP name ADD MEMBER**
- 控制inactive集群系统成员
  - **ALTER DATABASE DROP INACTIVE CLUSTER MEMBERS**
  - **ALTER SYSTEM JOIN DATABASE**
- 重新分配数据
  - **18.17 ALTER DATABASE REBALANCE**
  - **ALTER TABLE name REBALANCE**

可以通过以下视图查看集群系统相关信息.

对象集合	视图名称	说明
DICTIONARY_SCHEMA	**DBA_CLUSTER**	组成集群的集群组集群成员的对象信息

对象集合	视图名称	说明
	DBA_CLUSTER_COMMENTS	集群组与集群成员的注释信息
PERFORMANCE_VIEW_SCHEMA	V$CLUSTER_MEMBER	集群成员的状态信息

Table 4-1 集群系统相关信息

集群系统概念

SUNDB集群系统将一个数据库的数据分散或复制到多台服务器进行管理可以在组成集群系统的所有服务器上运行应用程序不管系统组成方式或连接的服务器运行方式与使用单个数据库相同

SUNDB集群系统由一个以上的集群组组成一个集群组由一个以上的集群成员组成不需要单独的应用程序服务器或元服务器应用程序通过连接到属于数据服务器的集群成员来运行

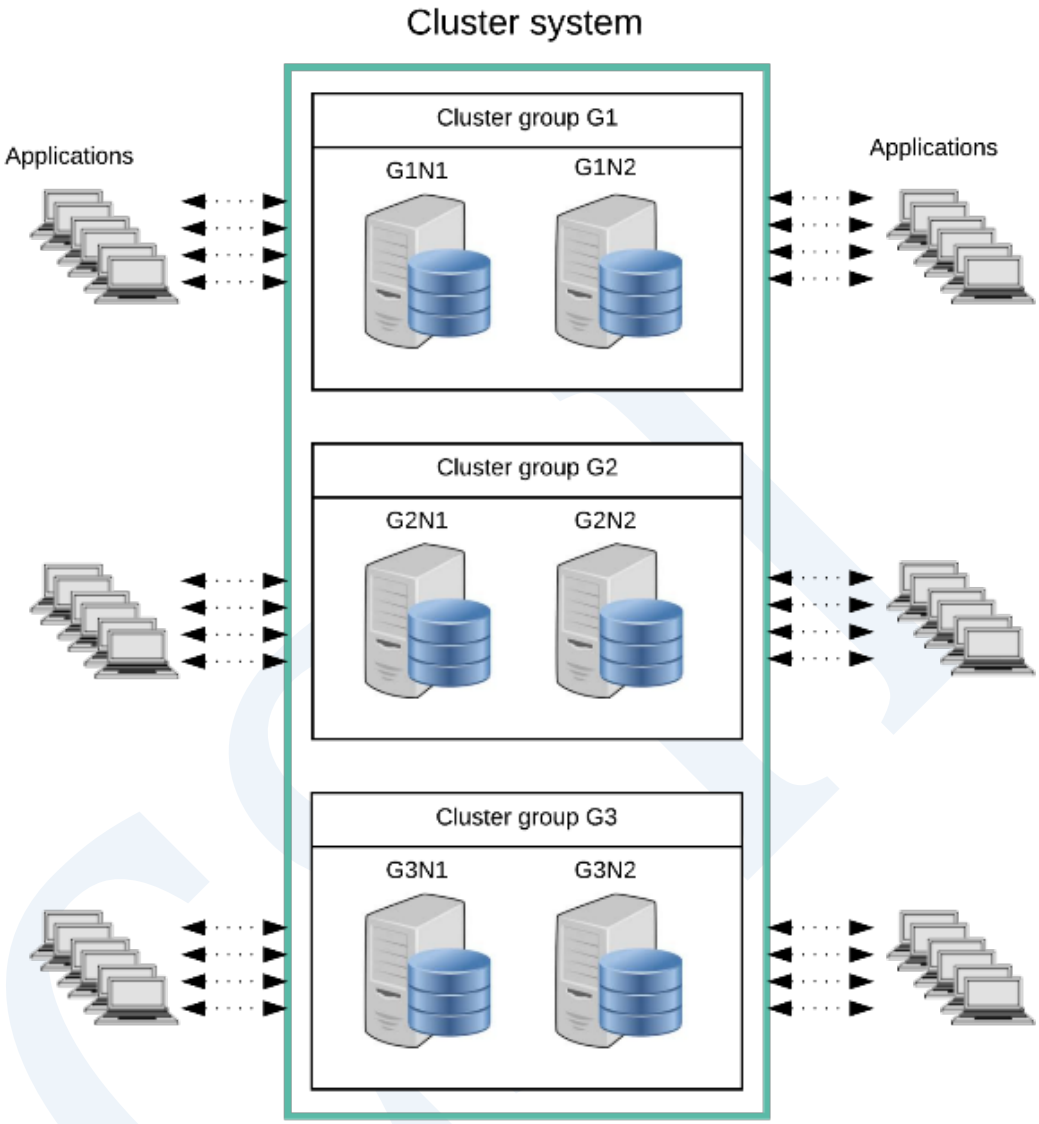


Figure 4-1 3 x 2 cluster system

上图是一个3x2集群系统由3个集群组组成每个集群组有2个集群成员在上图中集群系统由G1G2和G3集群组组成G1集群组由G1N1G1N2集群成员组成G2集群组由G2N1G2N2组成G3集群组由G3N1G3N2集群成员组成 应用程序可以访问六个集群成员中的任何一个运行方式与使用单个服务器相同

表中的数据分片（sharding）分配到各个集群组中集群组中的集群成员保持相同的副本

（replica）下图表示3x2集群中表数据的分配示意图

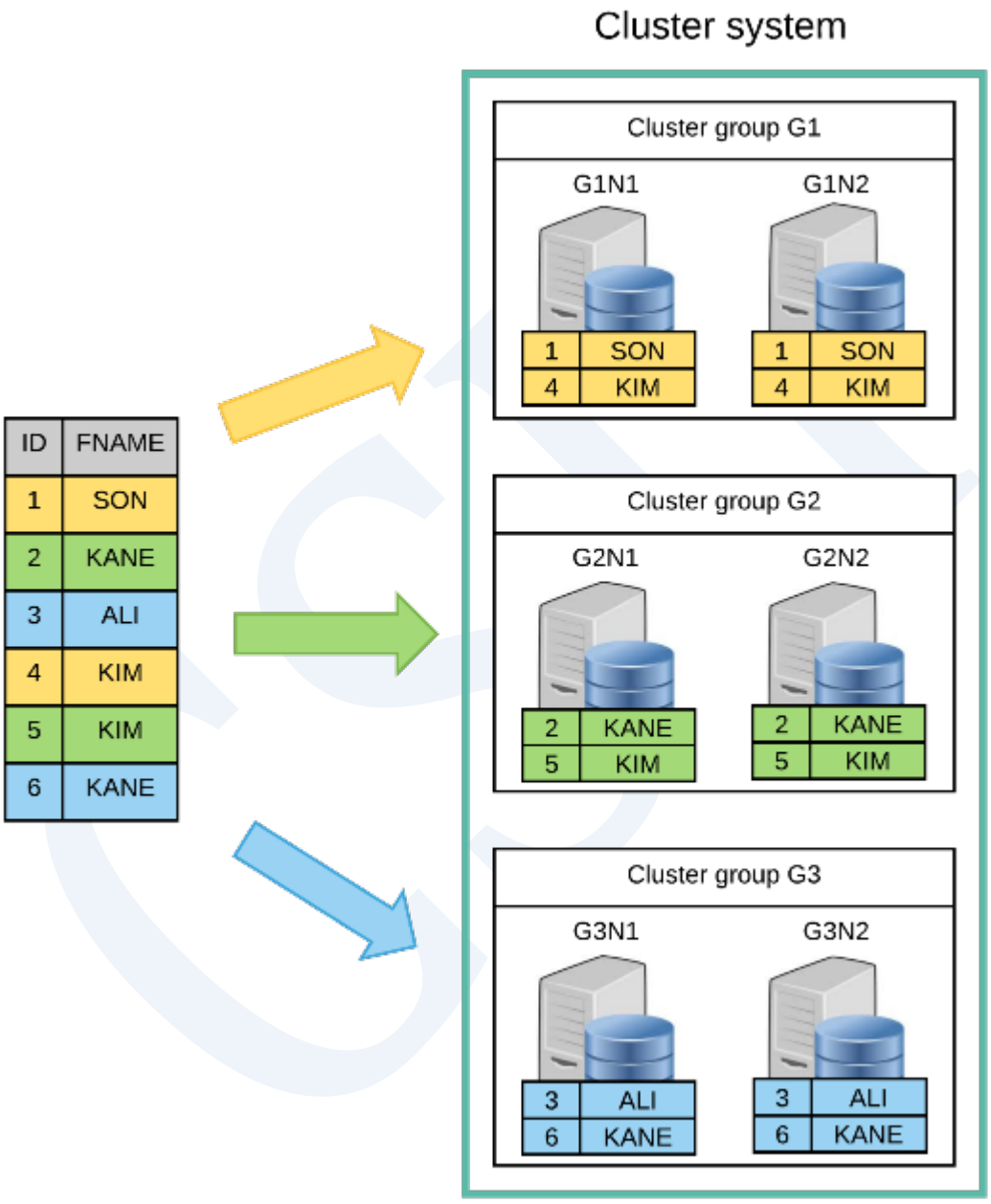


Figure 4-2 Cluster中的分片及复制概念

表中的数据由用户定义的分片策略（基于上图中的ID column）分配到各个集群组分配在集群组

中的数据会在集群组中的集群成员上维持副本

Cluster System的可用性

特定服务器发生故障或网络故障时cluster也能持续提供服务组成各个集群组的cluster成员维持相同的数据副本因此单个集群成员的故障不会导致服务停止即除非Cluster Group中的所有集群成员都发生故障导致data loss其他情况均可持续提供服务

如下图所示在3x2集群中即使三个设备出现故障也可以持续提供服务

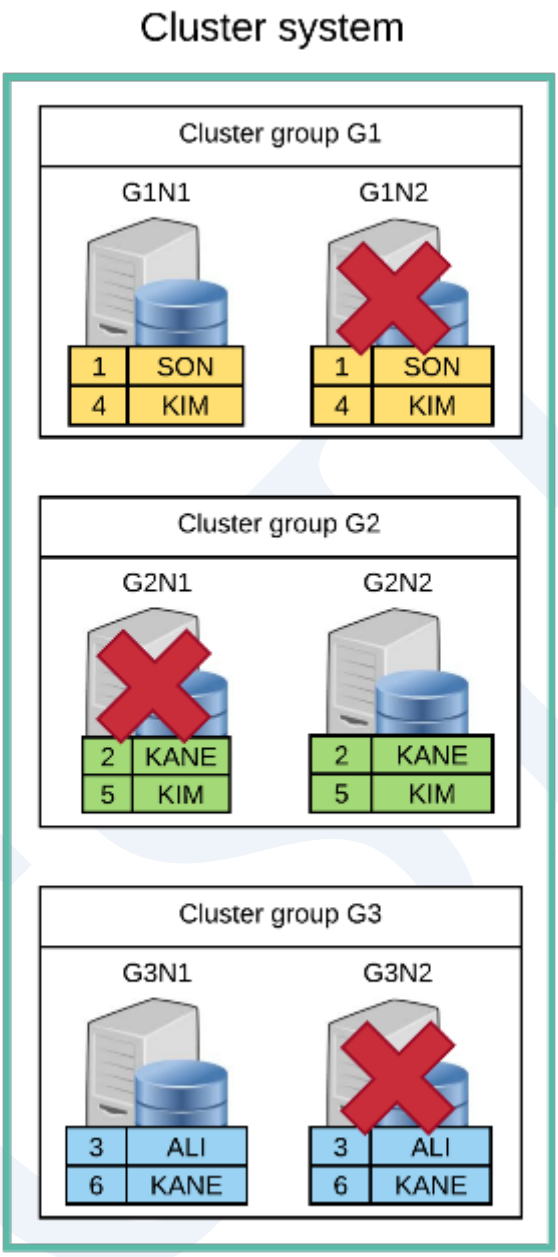


Figure 4-3 Cluster availability

如果在上述情况下又在G1N1G2N2G3N1发生其他故障引起数据丢失并无法持续提供服务因此在发生其他故障之前应将故障设备加入集群系统或添加新的集群成员

- **ALTER SYSTEM JOIN DATABASE**是将出现故障的集群成员重新加入到集群系统的语句
- **ALTER DATABASE DROP INACTIVE CLUSTER MEMBERS**是将出现故障的集群成员从Cluster System中删除的语句
- **ALTER CLUSTER GROUP name ADD MEMBER**是为了高可用性将集群成员添加到集群组的语句
- **ALTER DATABASE REBALANCEALTER TABLE name REBALANCE**是在添加的集群成员中重新分配数据的语句

扩展集群系统

可以在不中断服务的情况下添加新服务器以扩展集群

通过添加集群成员或集群组与将数据重新分配到添加的服务器的过程来实现集群的扩展

以下示例为将2x1集群扩展至3x2集群的示例

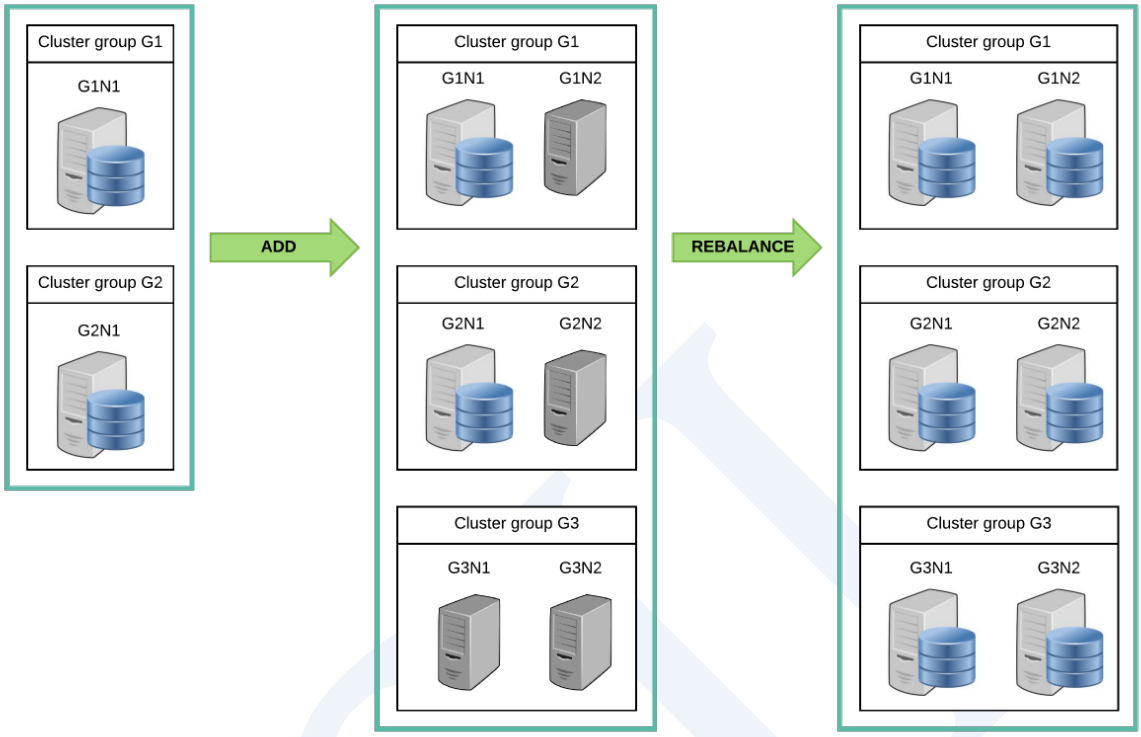


Figure 4-4 集群系统的扩展

为了扩展集群通过以下语句添加集群组与集群成员

- **ALTER CLUSTER GROUP name ADD MEMBER**
- **CREATE CLUSTER GROUP**

为了将新的集群成员添加到集群系统集群成员与集群系统的表空间应相同即应在集群成员上创建所有与集群系统相同的表空间

以下是将3 x 2 集群的集群组与集群成员添加到 2 x 1集群环境的示例以下示例将G1N2成员添加到G1组将G2N2成员添加到G2组另外生成包括G3N1G3N2成员的G3组

- 在G1 group添加G1N2 member

```
gSQL>
ALTER CLUSTER GROUP G1
      ADD CLUSTER MEMBER G1N2 HOST '192.168.0.12' PORT 10120;

Cluster Group altered.
```

- 在G2 group添加G2N2 member

```
gSQL>
ALTER CLUSTER GROUP G2
      ADD CLUSTER MEMBER G2N2 HOST '192.168.0.22' PORT 10220;

Cluster Group altered.
```

- 生成G3 group

```
gSQL>
CREATE CLUSTER GROUP G3
       CLUSTER MEMBER G3N1 HOST '192.168.0.31' PORT 10310,
       CLUSTER MEMBER G3N2 HOST '192.168.0.32' PORT 10320;

Cluster Group created.
```

新添加到集群系统的集群组与集群成员可以同步SQL对象的所有字典信息后提供服务但是由于添加的集群成员中没有分配数据因此无法带来增加可用性与负载均衡的效果为此需要将数据重新分配到添加的集群成员

按照顺序使用如下语句重新分配数据

- **ALTER DATABASE REBALANCE**
- **ALTER TABLE name REBALANCE**

以下为重新分配数据库的所有表数据的示例

```
gSQL> ALTER DATABASE REBALANCE;


Database altered.
```

4.2 Cluster Group

Cluster Group 相关语句

创建删除更改集群组的语句如下

- 创建集群组：**CREATE CLUSTER GROUP**
- 删除集群组：**DROP CLUSTER GROUP**
- 更改集群组：**ALTER CLUSTER GROUP name ADD MEMBER**

可以通过以下视图查看与集群组相关的信息

对象集合	视图名称	说明
DICTIONARY_SCHEMA	**DBA_CLUSTER**	构成集群的集群组集群成员的对象信息
	DBA_CLUSTER_COMMENTS	集群组与集群成员的注释信息

Table 4-2 集群组相关信息

集群组概念

为了运行集群系统至少创建一个以上的集群组

创建集群组

第一个创建的集群组应将自身包含在集群成员中 创建集群组的语句参考**CREATE CLUSTER GROUP**

集群成员是相当于数据服务器的物理概念而集群组是由一个以上的集群成员组成的逻辑概念

根据集群组的组成方式集群系统的高可用性与负载均衡效果会有所不同集群组中的集群成员数量越多可用性越高集群组数量越多数据分散从而增加集群系统的整体吞吐量（throughput）

集群组中的所有集群成员均复制并维持相同的数据除非构成集群组的所有集群成员都发生故障其他情况均可持续提供服务为了保持集群系统的可用性建议每个集群组由两个以上的集群成员组成

表数据根据分片策略(ShardingStrategy)分片并根据分配（Shard Placement）策略存储在不同的cluster group中管理根据服务特性合理的表分片策略与分配策略决定系统整体性能由于以存储Data的集群组为中心处理各个事务与查询因此引用的数据在相同的集群组时有利于提高性能

删除集群组

可能由于各种原因删除加入到集群系统并正在提供服务的集群组 集群组的删除参考**DROP CLUSTER GROUP**

删除集群组之前应将在该集群组中创建的分片表的所有分片移动到其他集群组应按照cluster-widegroup-specific table单独移动

4.3 Cluster Member

Cluster Member 相关语法

详细内容参考如下链接

- 添加cluster member：**ALTER CLUSTER GROUP name ADD MEMBER**
- 删除cluster member：**ALTER DATABASE DROP INACTIVE CLUSTER MEMBERS**
- 控制cluster member
  - **ALTER SYSTEM JOIN DATABASE**
  - **ALTER CLUSTER GROUP name OFFLINE MEMBER**
  - **ALTER DATABASE RESET LOCAL CLUSTER MEMBER**

可以通过以下视图查询与集群成员相关的信息

对象信息	视图名称	说明
DICTIONARY_SCHEMA	**DBA_CLUSTER**	组成cluster的集群组集群成员的对象信息
	DBA_CLUSTER_COMMENTS	集群组与集群成员的注释信息
PERFORMANCE_VIEW_SCHEMA	**V$CLUSTER_MEMBER**	集群成员的状态信息

Table 4-3 Cluster member 相关信息

Cluster Member概念

集群成员是构成集群系统的一个服务器与集群组中的集群成员保持相同的副本

集群成员是存储集群数据库的部分数据的数据服务器是处理应用程序的接入与请求的应用程序服务器是复制并管理对象的meta信息的META服务器即SUNDB集群不需要单独的数据服务器应用程序服务器META服务器

属于同一个集群组的集群成员拥有相同的数据副本因此特定集群成员的故障不会导致整个系统的故障为了系统的高可用性建议每个集群组至少包含两个以上的集群成员一个集群组最多可以由32个集群成员组成

通过**ALTER CLUSTER GROUP name ADD MEMBER**语句在集群组添加新的集群成员

添加的集群成员与集群系统维持相同的SQL对象的meta信息因此可以处理应用程序的接入与请求但是由于未分配数据因此添加集群成员并不保证集群组的高可用性

添加集群成员后通过执行以下语句分配数据

- 重新分配所有数据的情况：**ALTER DATABASE REBALANCE**
- 仅重新分配部分表的情况：**ALTER TABLE name REBALANCE**

集群成员出现故障不会影响服务的运行但无法执行DDL为了正常提供服务应对出现故障的集群成员进行措施

为了将出现故障的集群成员重新加入到集群系统应连接到该集群成员并启动至LOCAL OPEN阶段后执行**ALTER SYSTEM JOIN DATABASE**语句

如果未启动部分集群成员或者在网络断开的状态下启动集群系统时将集群成员启动至LOCAL OPEN阶段后执行**ALTER SYSTEM JOIN DATABASE**语句

如果无法恢复出现故障的集群成员的设备可在集群系统中使用**ALTER DATABASE DROP INACTIVE CLUSTER MEMBERS**语句删除出现故障的集群成员

从集群系统中删除的集群成员仍然拥有从集群系统中删除之前的信息并且无法再次加入集群系统将集群成员初始化至加入集群系统之前的状态时执行**ALTER DATABASE RESET LOCAL CLUSTER MEMBER**语句

与创建新的集群成员的数据库的不同之处在于由于其保留表空间信息因此在集群系统添加新的集群成员时可缩短表空间的创建时间

4.4 Cluster Location

Cluster Location相关语句

创建删除变更cluster location的语句如下

- 创建 cluster location：**CREATE CLUSTER LOCATION**
- 删除 cluster location：**DROP CLUSTER LOCATION**
- 变更 cluster location：**ALTER CLUSTER LOCATION**

可以通过以下视图查询与集群位置相关的信息

对象集合	视图名称	说明
PERFORMANCE_VIEW_SCHEMA	V$CLUSTER_LOCATION	集群位置的信息

Table 4-4 Cluster location相关信息

集群位置的概念

Cluster location是用于集群系统中的每个集群成员之间连接内部集群网络的连接信息所有集群成员均使用集群专用TCP网络以实现事务处理及交换管理信息等各种协议的相互传输在这种情况下将用于连接的成员名称主机IP地址端口统称为cluster location

每个成员必须指定唯一的cluster location信息如果信息重复则集群网络连接失败无法实现正常的集群系统运行

删除或添加集群成员时会自动删除或添加cluster location信息因此很少出现需要用户亲自添加或删除Location信息的情况但是在以下情况下可以使用Cluster Location相关的DDL语句

- 由于删除location control file而丢失cluster location信息时：**CREATE CLUSTER LOCATION**
- 更改已注册的cluster成员的连接信息时即更改了硬件或更改了连接IP端口时：**ALTER CLUSTER LOCATION**

4.5 集群表与分片

分片相关语句

Shard定义与Shard重新分配的语句如下

- Shard定义：**CREATE TABLE** 语句的<table sharding strategy>
- Shard重新分配：**ALTER TABLE name REBALANCE**

可以通过以下视图查询与集群表的分片相关的信息

对象集合	视图名称	说明
DICTIONARY_SCHEMA	**ALL_CLUSTER_TABLES**	用户可访问的cluster table信息
	ALL_SHARD_KEY_COLUMNS	用户可访问的cluster table的shard key column信息
	ALL_TAB_PLACE	用户可访问的cluster table的部署信息
	ALL_TAB_SHARDS	用户可访问的cluster table的shard信息
	USER_CLUSTER_TABLES	用户拥有的cluster table信息
	USER_SHARD_KEY_COLUMNS	用户拥有的cluster table的shard key column信息
	USER_TAB_PLACE	用户拥有的cluster table的部署信息
	USER_TAB_SHARDS	用户拥有的cluster table的shard信息

Table 4-5 集群表与分片相关信息

集群表类型

在集群环境中用户创建的表为以下两种类型之一

- Cloned table：统一复制并管理表的数据
- Sharded table：水平拆分并管理表的数据

Cloned table复制并管理表的所有数据因此适合用于产品目录供应者目录等数据更改较少且数据量相对较少的表在cloned table中添加删除更新数据时均统一反映到部署cloned table的所有集群成员中

Sharded table适合用于交易记录通话记录等表的数据量大而需要进行分片的情况如下分为三种分片策略

- Hash sharded table
  - 以sharding key的HASH值为准将表的数据拆分为多个分片并分配到集群系统中
- Range sharded table
  - 以sharding key的范围(range)值为准将表的数据拆分为多个分片并分配到集群系统中
- List sharded table
  - 以sharding key的列表(list)值为准将表的数据拆分为多个分片并分配到在集群系统中

分片表通过水平分割row并以分片为单位进行管理分片策略使用 **CREATE TABLE**语句的SHARDING BY子句定义 根据分片策略分类的行(row)集合叫shard

每个分片根据用户定义的分配策略分配到集群组中可以通过 **CREATE TABLE**语句中的AT CLUSTER WIDE自动分配或使用AT CLUSTER GROUP子句指定分配的集群组当确定使用AT CLUSTER WIDE自动分配时在使用**CREATE CLUSTER GROUP**语句添加集群组后执行**ALTER TABLE name REBALANCE**语句相反如果使用AT CLUSTER GROUP指定分片分配的集群组时则shard不会分配到新添加的集群组中

通过以下示例说明在3x2环境中根据集群表的类型创建表并分配数据的概念

Cloned Table

Cloned table统一复制并管理表的所有数据

以下为创建cluster-wide cloned table的示例所有表数据均统一复制并分配在3x2结构的集群成员中

```
CREATE TABLE t1 ( id INTEGER )
   CLONED
   AT CLUSTER WIDE
;
```

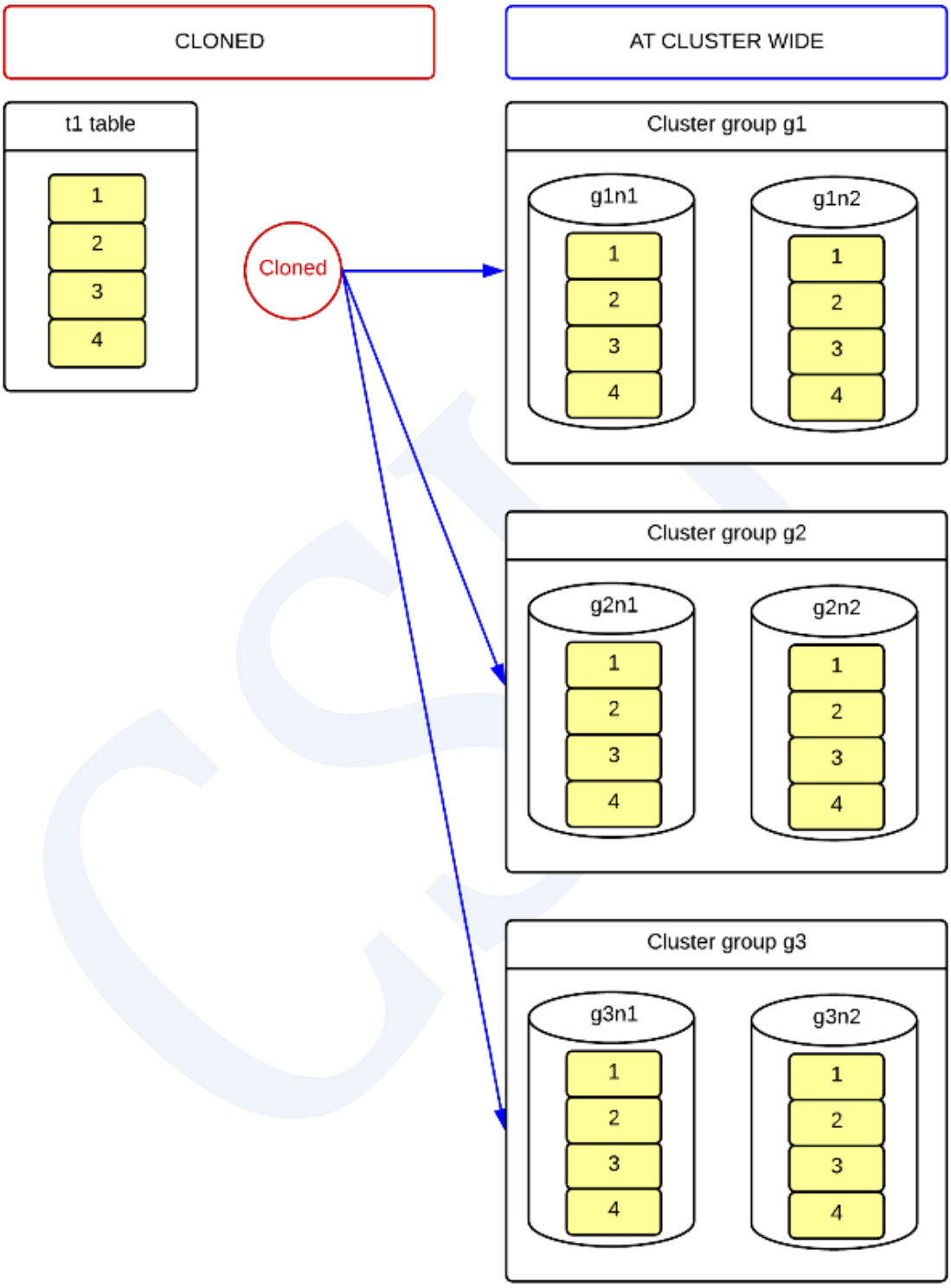


Figure 4-5 Cluster-wide cloned table

以下为创建group-specific cloneed table的示例复制并管理表中的所有数据但复制的表数据仅存在于用户指定的g1与g2组的cluster成员中g3集群组中没有数据

```
CREATE TABLE t1 ( id INTEGER )
   CLONED
   AT CLUSTER GROUP g1, g2
;
```

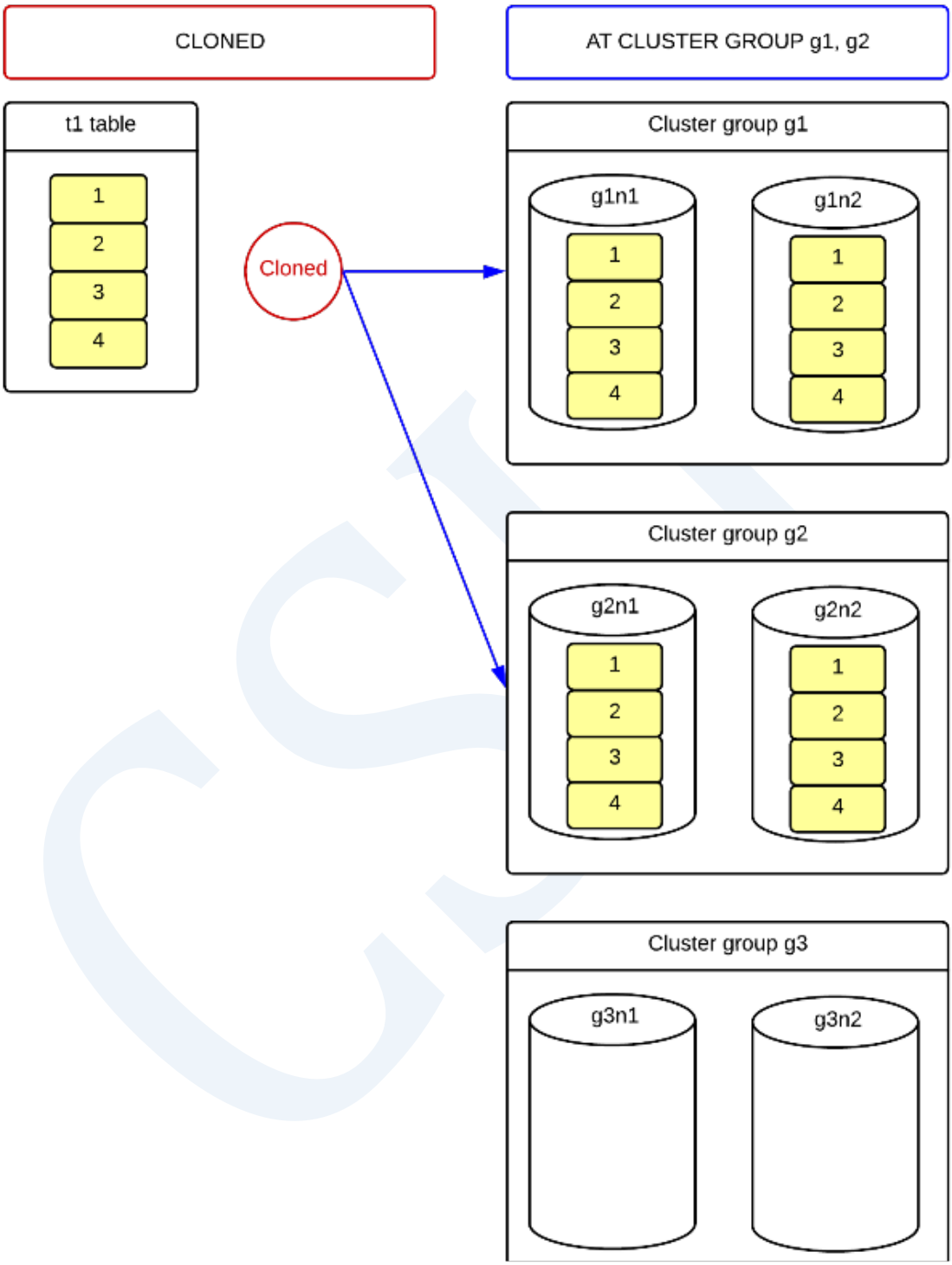


Figure 4-6 Group-specific cloned table

Hash-sharded Table

HASH-sharded table以sharding key的hash值为准将数据拆分为多个shard并将其分配到cluster system

以下为创建cluster-wide hash-sharded table的示例在表添加数据时以ID column的值为准生成hash值并用其决定在5个Shard中分配row的Shard每个Shard均自动分配具有相同ID column值的所有行均包含在同一个shard中并分配到相同的集群组中

```
CREATE TABLE t1 ( id INTEGER )
   SHARDING BY HASH(id)
   SHARD COUNT 5
   AT CLUSTER WIDE
;
```

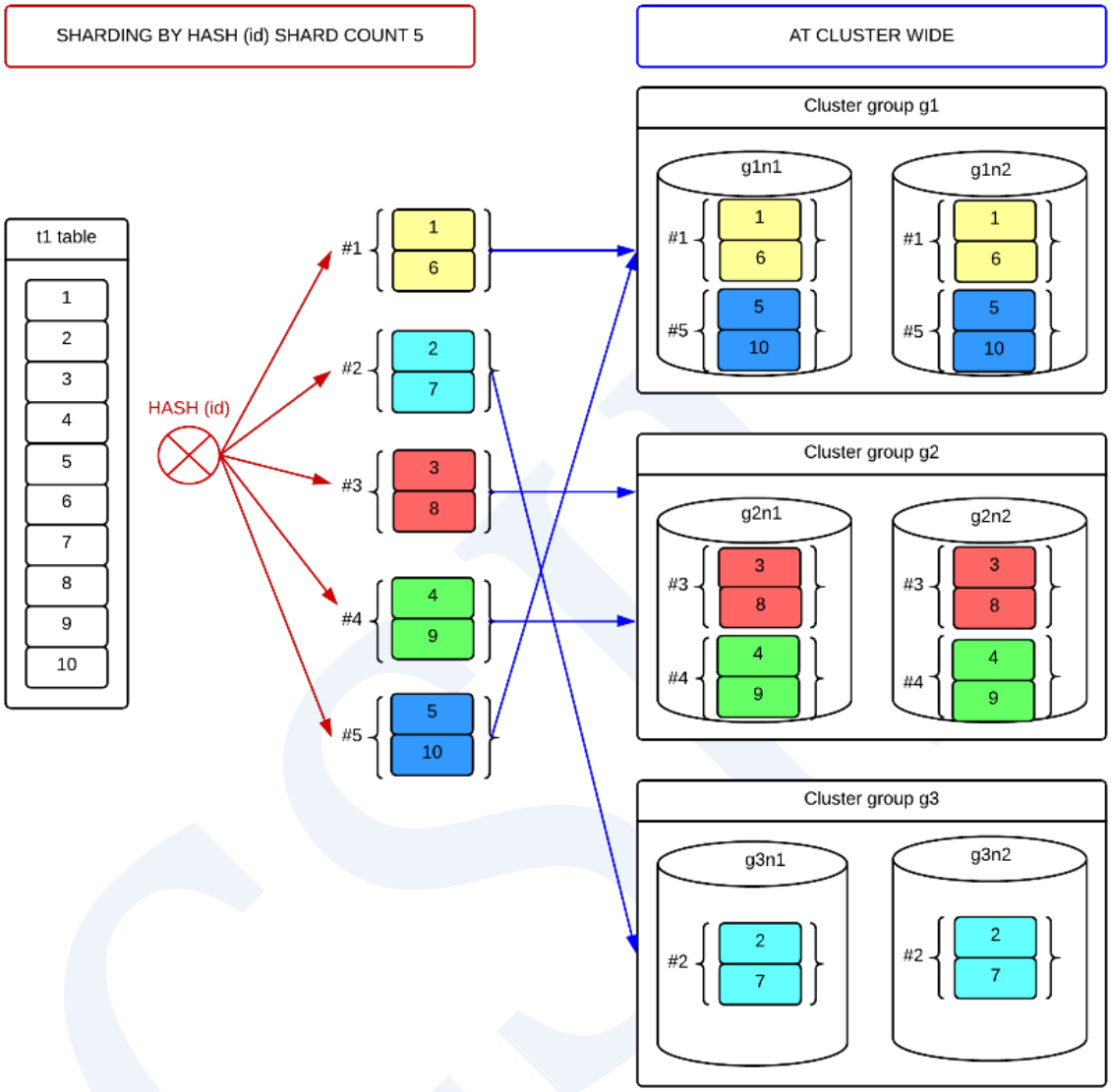


Figure 4-7 Cluster-wide hash-sharded table

以下为创建group-specific hash-sharded table的示例由ID column的hash值来确定shard但每个shard仅分配在用户指定的g1g2 cluster group中

```
CREATE TABLE t1 ( id INTEGER )
   SHARDING BY HASH(id)
   SHARD COUNT 5
```

```
    AT CLUSTER GROUP g1, g2
;
```

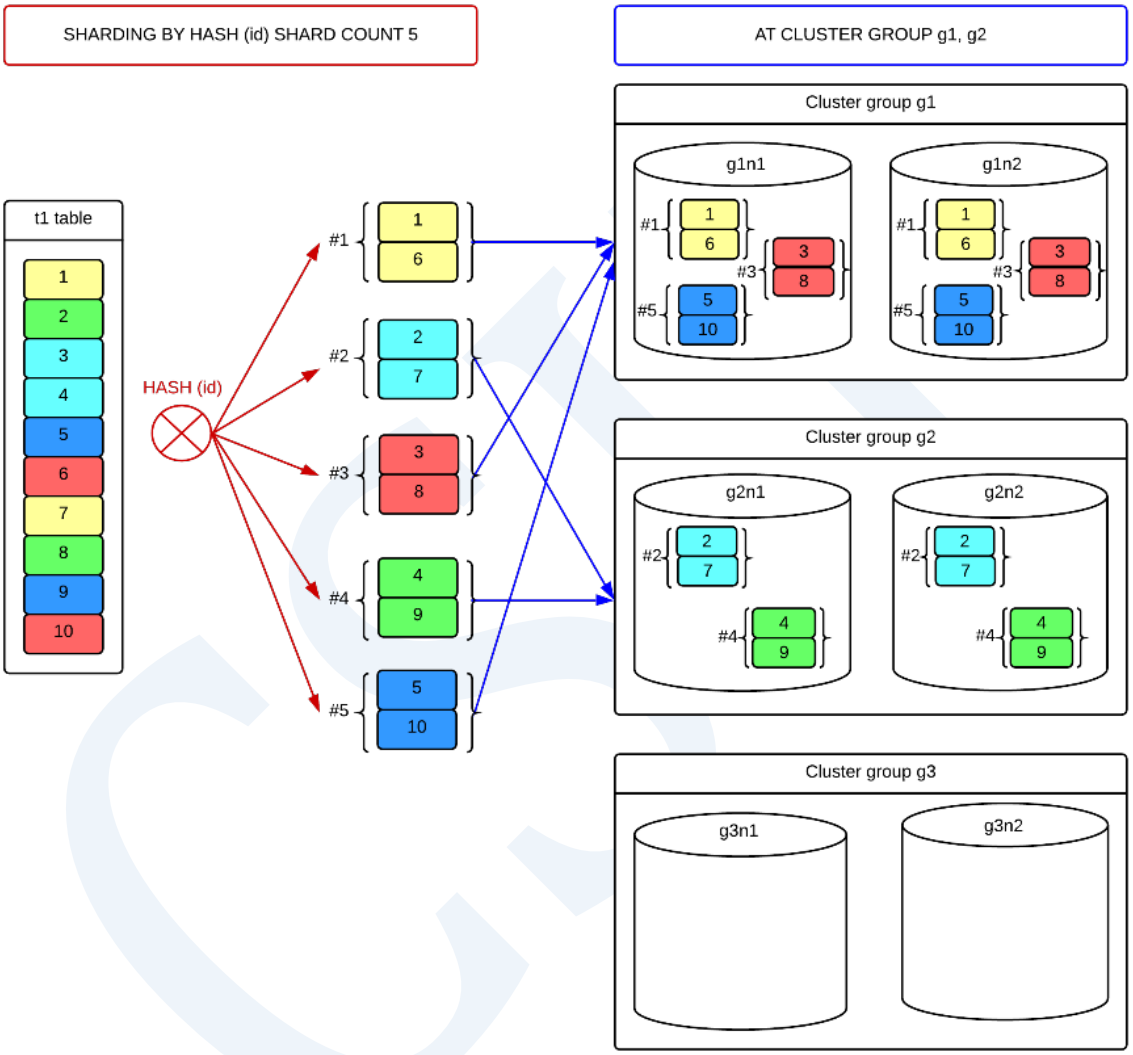


Figure 4-8 Group-specific hash-sharded table

Range-sharded Table

Range-sharded table以sharding key的范围(range)值为准将数据拆分为多个shard并将其分配到cluster System中

以下为创建cluster-wide range-sharded table的示例在表添加数据时以ID column值的范围值为准决定在5个shard中分配row的shard每个shard均自动部署ID column在同一范围内的所有行均包含在同一个shard中并分配到相同的集群组中

```
CREATE TABLE t1 ( id INTEGER )
   SHARDING BY RANGE(id)
   AT CLUSTER WIDE
       SHARD s1 VALUES LESS THAN ( 20 ),
       SHARD s2 VALUES LESS THAN ( 40 ),
       SHARD s3 VALUES LESS THAN ( 60 ),
       SHARD s4 VALUES LESS THAN ( 80 ),
       SHARD s5 VALUES LESS THAN ( MAXVALUE )
;
```

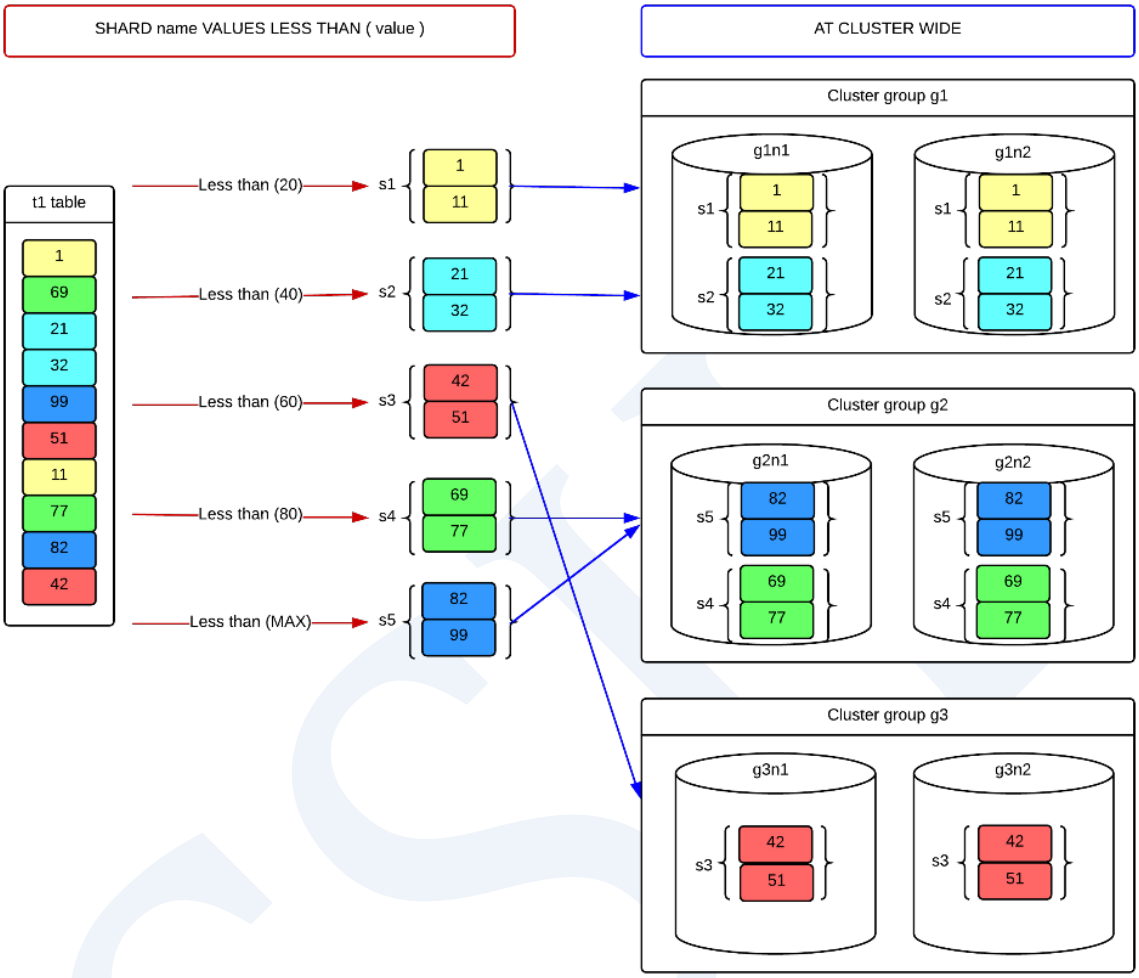


Figure 4-9 Cluster-wide range-sharded table

以下为创建group-specific range-sharded table的示例由ID column的范围值来确定shard但每个shard均分配到用户指定的集群组中

```
CREATE TABLE t1 ( id INTEGER )
   SHARDING BY RANGE(id)
       SHARD s1 VALUES LESS THAN ( 20 )       AT CLUSTER GROUP g1,
       SHARD s2 VALUES LESS THAN ( 40 )       AT CLUSTER GROUP g2,
       SHARD s3 VALUES LESS THAN ( 60 )       AT CLUSTER GROUP g1,
```

```
        SHARD s4 VALUES LESS THAN ( 80 )       AT CLUSTER GROUP g2,
        SHARD s5 VALUES LESS THAN ( MAXVALUE ) AT CLUSTER GROUP g3
;
```

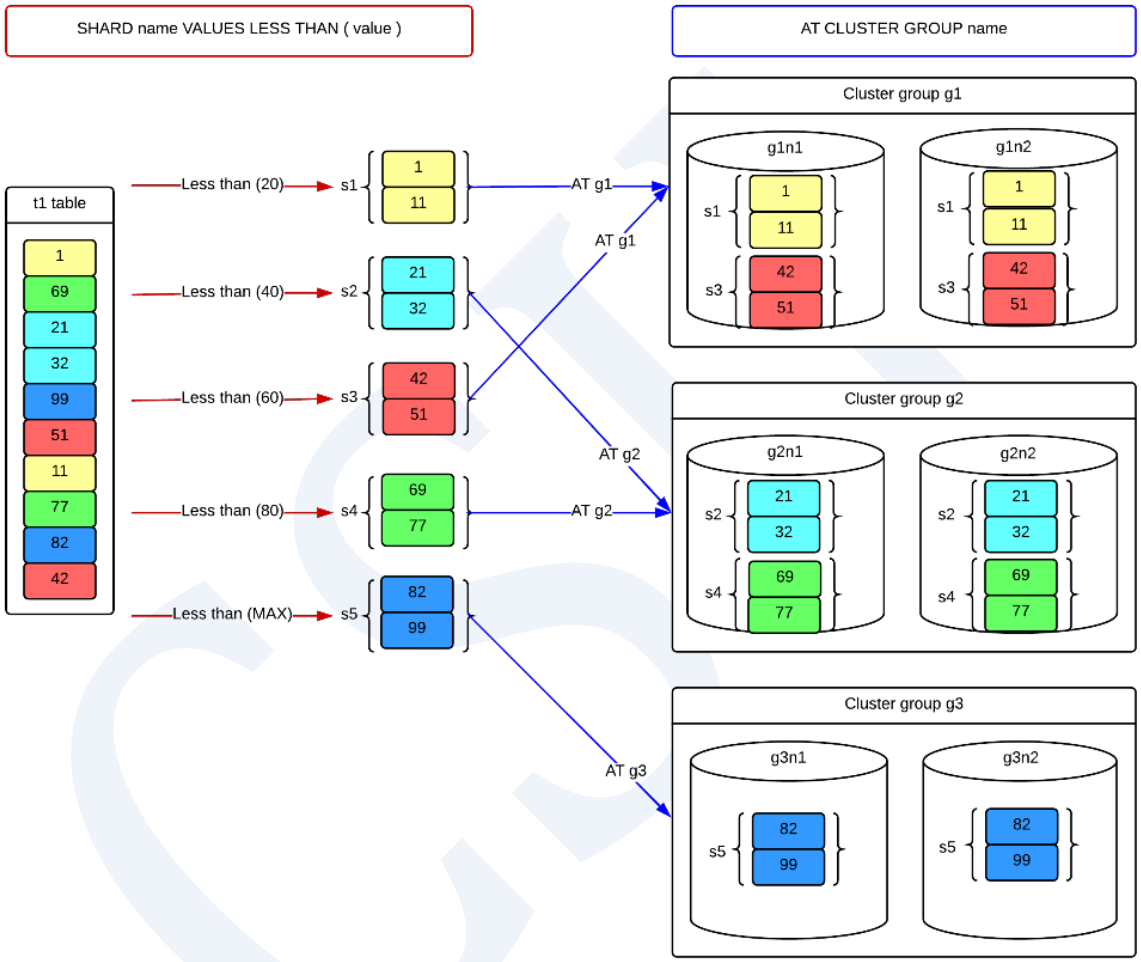


Figure 4-10 Group-specific range-sharded table

List-sharded Table

List-sharded table以分片键的列表（list）值为准将数据拆分为多个shard并分配到集群系统中

以下为创建cluster-wide list-sharded table的示例在表添加数据时将row分配到具有与CITY column值拥有相同列表值的shard中各Shard被自动分配

```
CREATE TABLE t1 ( city VARCHAR(128) )
   SHARDING BY LIST (city)
      AT CLUSTER WIDE
      SHARD s1 VALUES IN ( 'seoul' ),
      SHARD s2 VALUES IN ( 'busan', 'ulsan' ),
      SHARD s3 VALUES IN ( 'suwon', 'ansan', 'osan' ),
      SHARD s4 VALUES IN ( 'goyang', 'paju', 'guri' ),
      SHARD s5 VALUES IN ( DEFAULT )
;
```

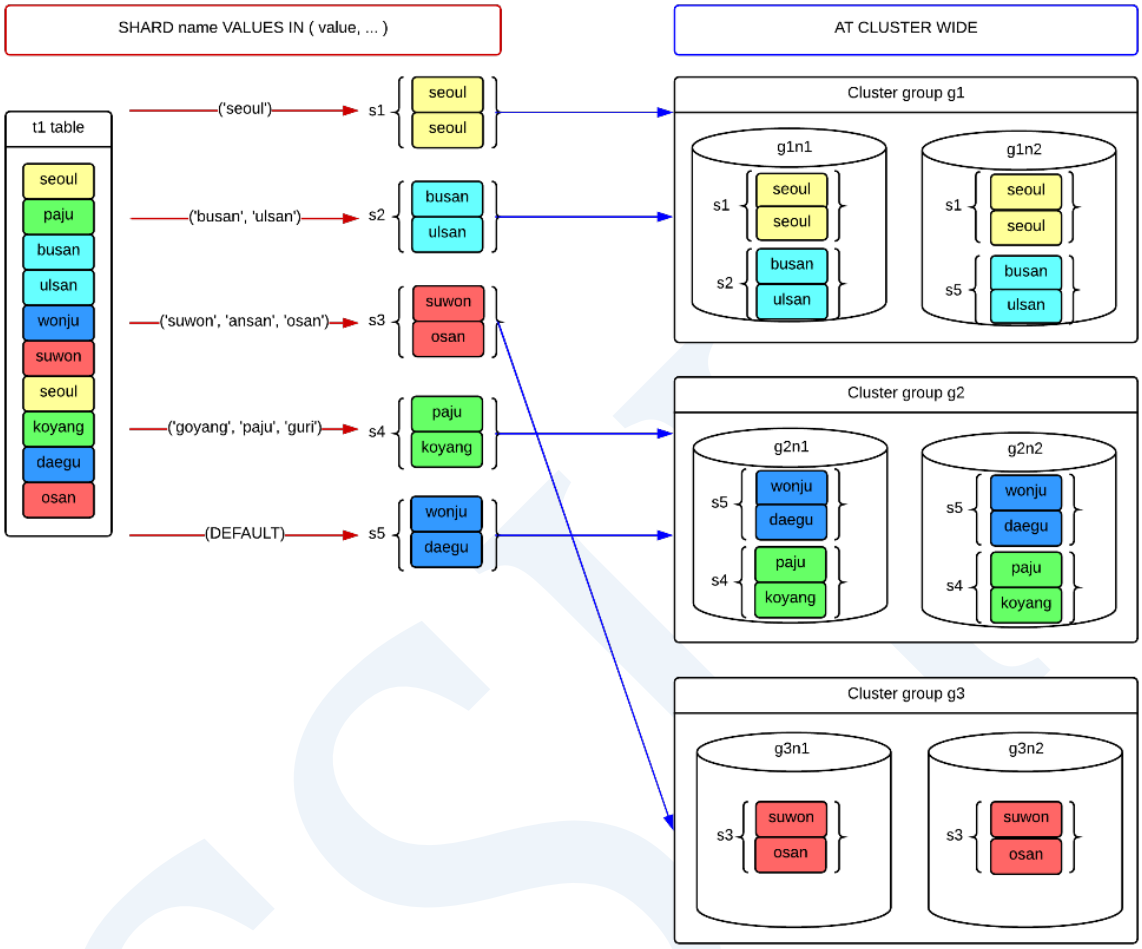


Figure 4-11 Cluster-wide list-sharded table

以下为创建group-specific list-sharded table的示例由CITY column的列表（list）值来确定shard但每个Shard均分配到用户指定的集群组中

```
CREATE TABLE t1 ( city VARCHAR(128) )
   SHARDING BY LIST (city)
      SHARD s1 VALUES IN ( 'seoul' )                  AT CLUSTER GROUP g1,
      SHARD s2 VALUES IN ( 'busan', 'ulsan' )         AT CLUSTER GROUP g2,
      SHARD s3 VALUES IN ( 'suwon', 'ansan', 'osan' ) AT CLUSTER GROUP g1,
```

```
    SHARD s4 VALUES IN ( 'goyang', 'paju', 'guri' ) AT CLUSTER GROUP g2,
    SHARD s5 VALUES IN ( DEFAULT )                  AT CLUSTER GROUP g3
;
```

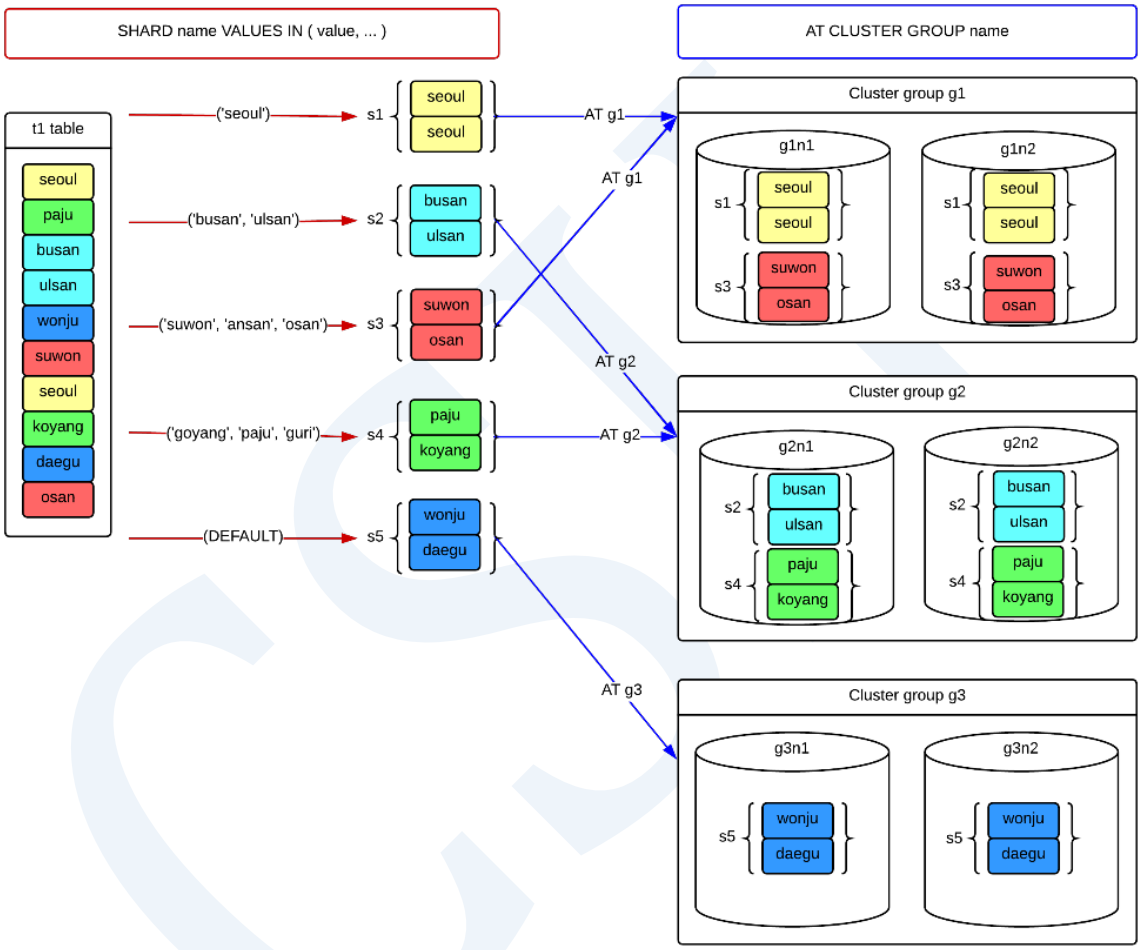


Figure 4-12 Group-specific list-sharded table

集群表的重新分配

使用**ALTER TABLE name REBALANCE**重新分配cluster table的数据

集群表的数据重新分配按照以下单位实现

- Cloned table：整个表
- Sharded table：以分片（shard）为单位

通过AT CLUSTER WIDE指定的表在生成的集群组中自动重新分配分片（shard）而通过AT CLUSTER GROUP指定分配分片的集群组时分片不会重新分配到生成的集群组

- 使用AT CLUSTER WIDE自动重新分配时可以将数据重新分配到新的集群组集群成员

```
CREATE TABLE region
(
    r_regionkey    INTEGER
  , r_name         CHAR(25)
  , r_comment      VARCHAR(152)
)
CLONED
AT CLUSTER WIDE;
```

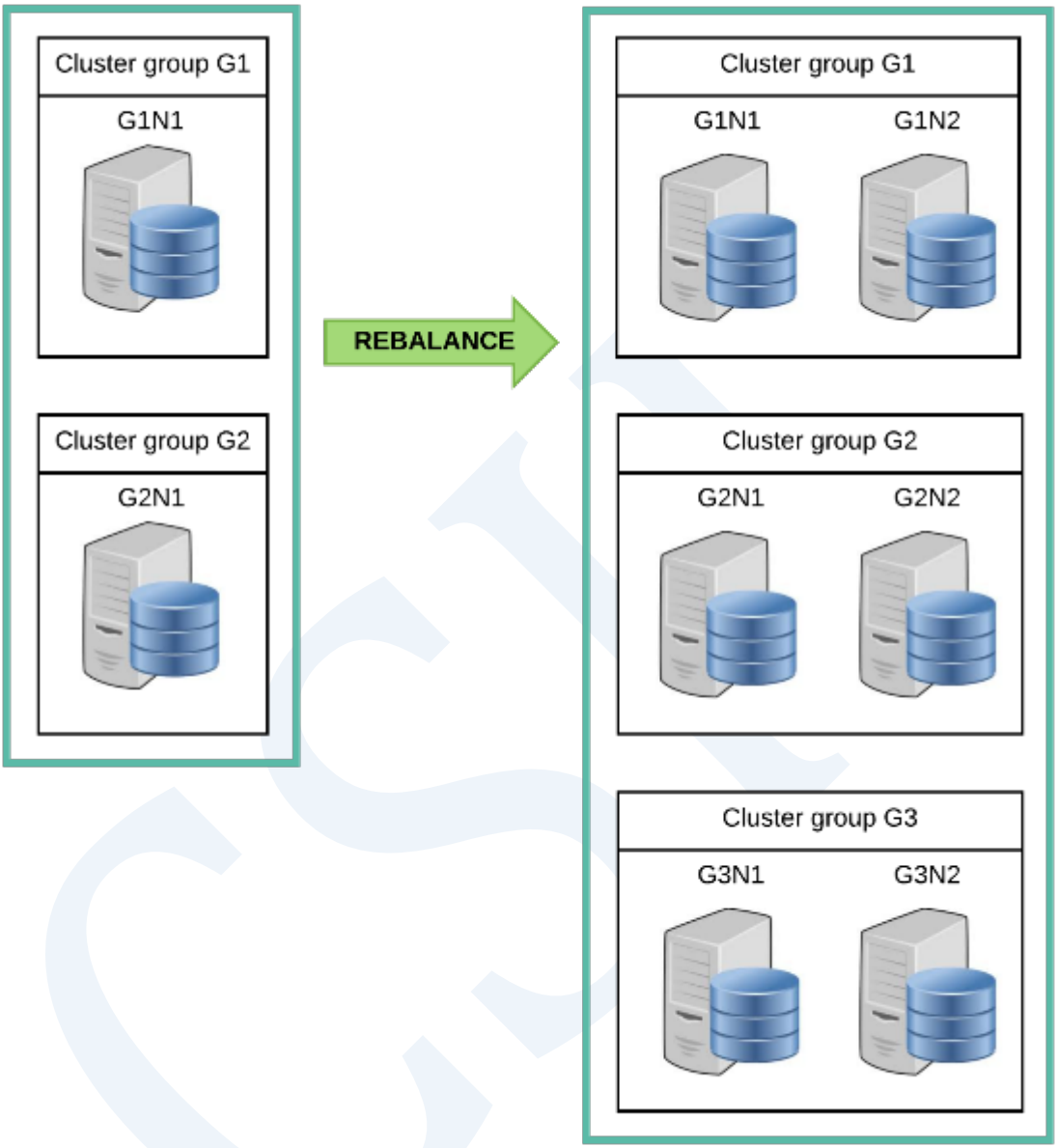


Figure 4-13 AT CLUSTER WIDE定义的区域表的重新分配

- 使用AT CLUSTER GROUP指定分片的分配位置时
  - 数据不会重新分配到新的集群组
  - 可以将数据重新分配到添加到指定集群组的新集群成员

```
CREATE TABLE nation

(

    n_nationkey   INTEGER

  , n_name        CHAR(25)

  , n_regionkey   INTEGER

  , n_comment     VARCHAR(152)

)

CLONED

AT CLUSTER GROUP g1, g2;
```

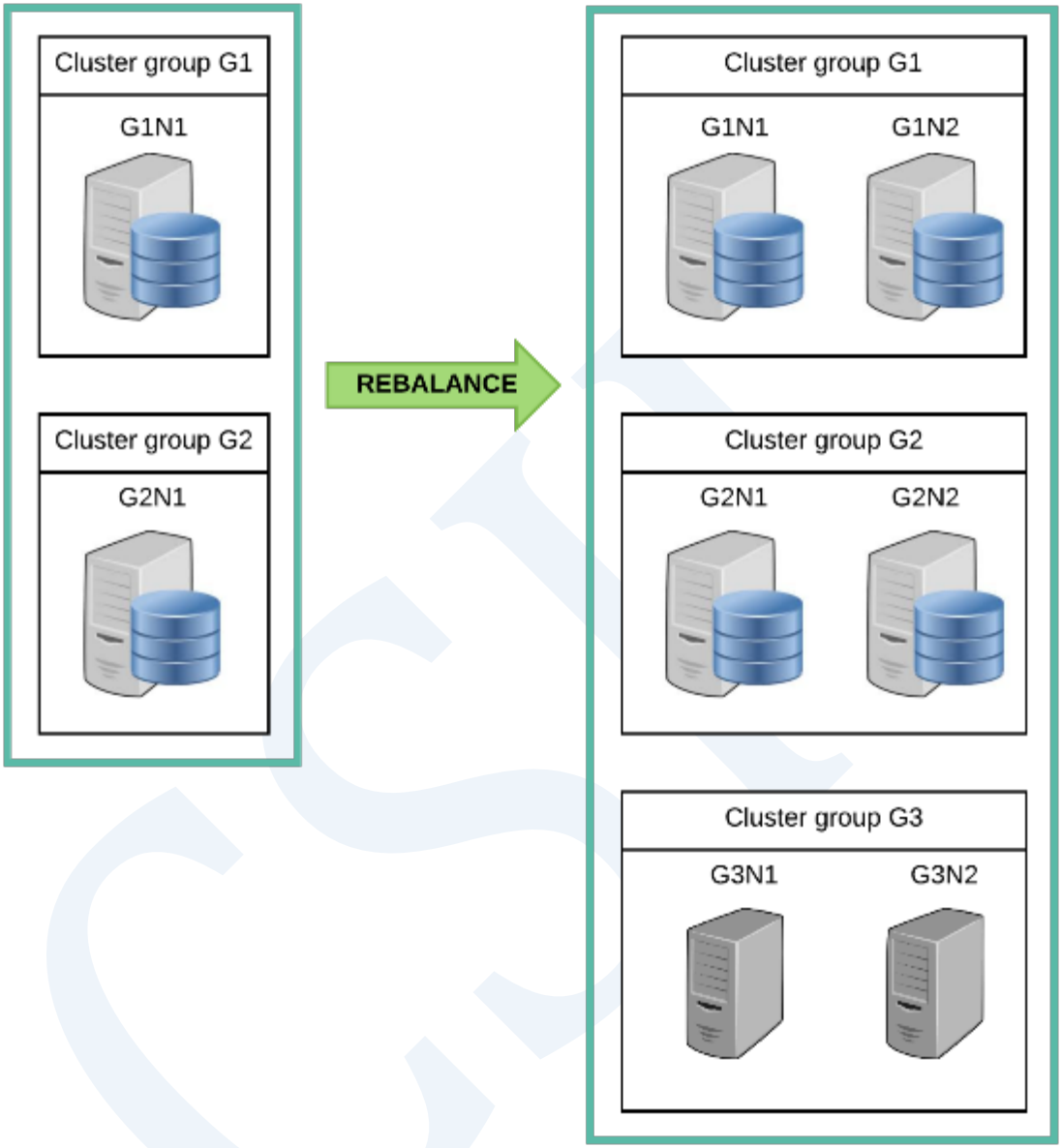


Figure 4-14 AT CLUSTER GROUP定义的nation表的重新分配

如下可以通过以下视图查询表的分配信息

- **USER_TAB_PLACE**
- **ALL_TAB_PLACE**

```
gSQL>
SELECT group_name, member_name
  FROM user_tab_place
 WHERE table_name = 'REGION';


GROUP_NAME MEMBER_NAME
---------- -----------
G1         G1N1
G1         G1N2
G2         G2N1
G2         G2N2
G3         G3N1
G3         G3N2


6 rows selected.
```

分片表以分片为单位执行重新分配随着组的增加重新分配分片的概念如下

```
CREATE TABLE orders
(
    o_orderkey     INTEGER
  , o_custkey      INTEGER
  , o_orderstatus  CHAR(1)
  , o_totalprice   NUMERIC(12,2)
  , o_orderdate     DATE
  , o_orderpriority CHAR(15)
```

```
  , o_clerk        CHAR(15)
  , o_shippriority INTEGER
  , o_comment      VARCHAR(79)
)
SHARDING BY HASH( o_orderkey )
SHARD COUNT 24
AT CLUSTER WIDE
;
```

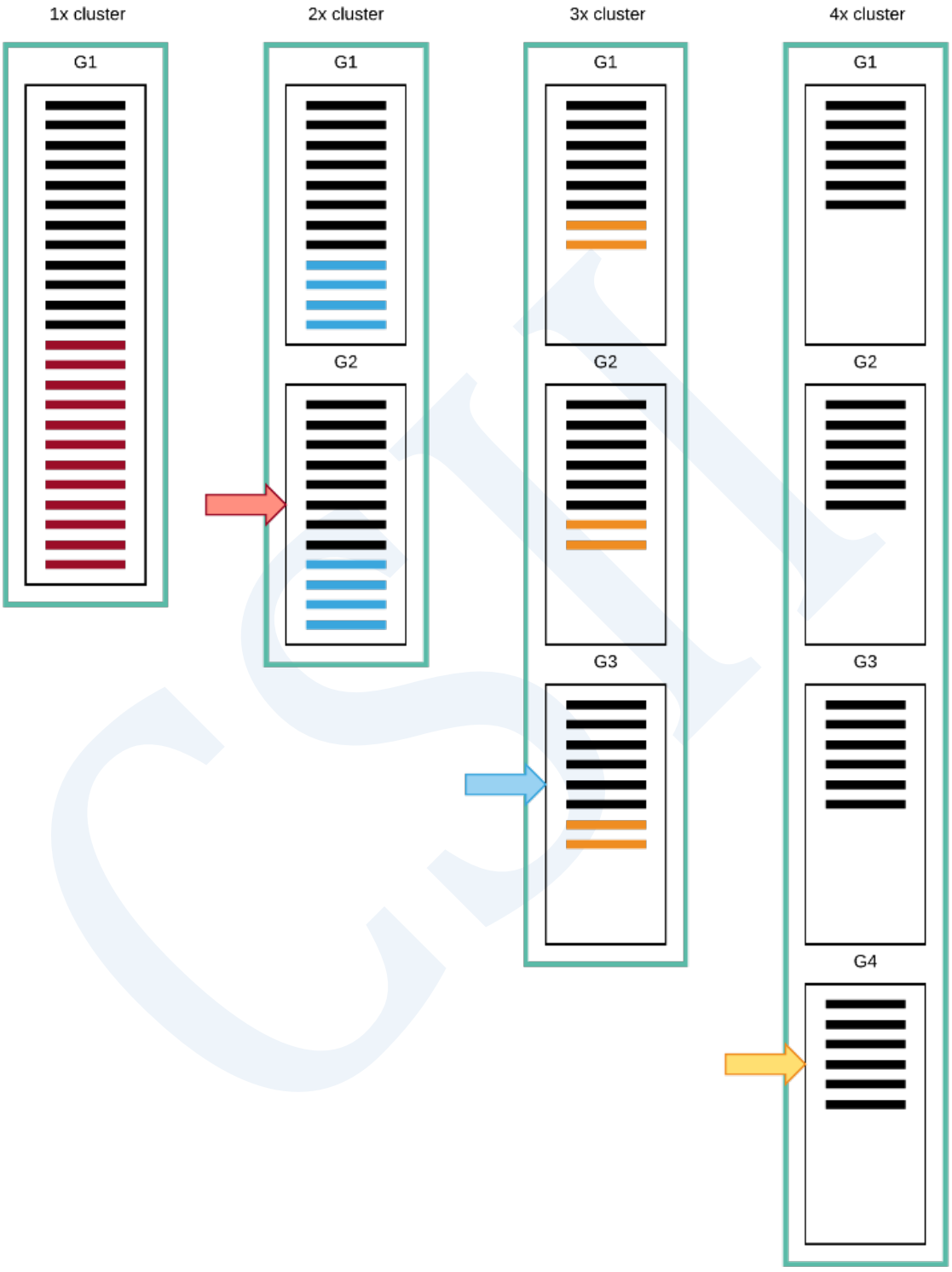


Figure 4-15 随着集群组的增加分片的重新分配

以上示例中orders表的data被拆分为24个Shard并分配在有一个组的1 x cluster中所有shard均部署在一个group中有两个组的2 x Cluster中每个group分配12个shard

2 x cluster扩展至3 x cluster的情况下分片从原有群组移动至新群组但每个群组中的分片数量均为8个 当从3 x cluster扩展至4 x cluster时G1G2和G3分别将两个分片重新分配到新添加的G4组

即分片的重新配置使从原有的组移动到新的组的移动达到最小化并且每个组中的分片数量几乎相同可均匀分布数据

如下可以通过以下视图查询分片表的分片的分配信息

- **USER_TAB_SHARDS**
- **ALL_TAB_SHARDS**

```
gSQL>
SELECT shard_name, group_name
  FROM user_tab_shards
 WHERE table_name = 'ORDERS';


SHARD_NAME   GROUP_NAME
------------ ----------
SHARD_000000 G1
SHARD_000001 G1
SHARD_000002 G1
SHARD_000003 G1
SHARD_000004 G1
SHARD_000005 G1
```

```
SHARD_000006 G1

SHARD_000007 G1

SHARD_000008 G3

SHARD_000009 G3

SHARD_000010 G3

SHARD_000011 G3

SHARD_000012 G2

SHARD_000013 G2

SHARD_000014 G2

SHARD_000015 G2

SHARD_000016 G2

SHARD_000017 G2

SHARD_000018 G2

SHARD_000019 G2

SHARD_000020 G3

SHARD_000021 G3

SHARD_000022 G3

SHARD_000023 G3


24 rows selected.
```

如果如下具有相同的<sharding strategy>的表完成完成重新定位则分片的分配结果相同即使在不同的表也保证具有相同分片键的row与相同shard分配在相同的group

- Table t1
    - 在2X cluster上创建

    - CREATE TABLE t1 (c1 INTEGER) SHARDING BY (c1);
    - 重新分配到4x cluster
    - ALTER TABLE t1 REBALANCE:
- Table t2
    - 在3X cluster上创建
    - CREATE TABLE t2 (a1 INTEGER) SHARDING BY (a1);
    - 重新分配到到4x cluster
    - ALTER TABLE t2 REBALANCE:
- Table t3
    - 在4X cluster上创建
    - CREATE TABLE t3 (i1 INTEGER) SHARDING BY (i1);

即以下查询仅可访问一个集群成员进行处理

```
SELECT COUNT(*)
  FROM t1, t2, t3
 WHERE t1.c1 = t2.a1
   AND t2.a1 = t3.i1
   AND t1.c1 = 1;
```

4.6 Global Secondary Index（全局二级索引）

Global Secondary Index相关语句

创建删除变更global secondary index的语句如下

- 创建 global secondary index：**ALTER TABLE name ADD GLOBAL SECONDARY INDEX**
- 删除 global secondary index：**ALTER TABLE name DROP GLOBAL SECONDARY INDEX**
- 变更 global secondary index：**ALTER TABLE name ALTER GLOBAL SECONDARY INDEX**

可以通过以下视图查询全局二级索引的相关信息

对象集合	视图名称	说明
DICTIONARY_SCHEMA	**ALL_CLUSTER_TABLES**	是否存在用户可访问的表的全局二级索引
	ALL_GLOBAL_SECONDARY_INDEXES	用户可访问的全局二级索引对象信息
	ALL_GSI_PLACE	用户可访问的全局二级索引的部署信息
	USER_CLUSTER_TABLES	是否存在用户拥有的表的全局二级索引

对象集合	视图名称	说明
	USER_GLOBAL_SECONDARY_INDEXES	用户拥有的全局二级索引对象信息
	USER_GSI_PLACE	用户拥有的全局二级索引的部署信息

Table 4-6 Global secondary index相关信息

Global Secondary Index概念

全局二级索引是在集群环境的每个集群成员中将表记录的全局行标识符（GRIDglobal row identifier）值组成为key的B-Tree索引

在集群环境中表复制到群组中的所有成员并且在DML或select时反映并查询相同的记录为此GRID是可区分相同记录的集群成员的唯一值在首次插入记录时分配且传播到群组中的所有成员并与记录一起存储

即使更新记录也不会更改记录的GRID值即使更改了分片键并移动到其他分片也不会更改GRID值

在集群环境中创建表时可以生成或不生成全局二级索引创建表后也可单独生成或删除全局二级索引表可以没有全局二级索引或最多只能创建一个

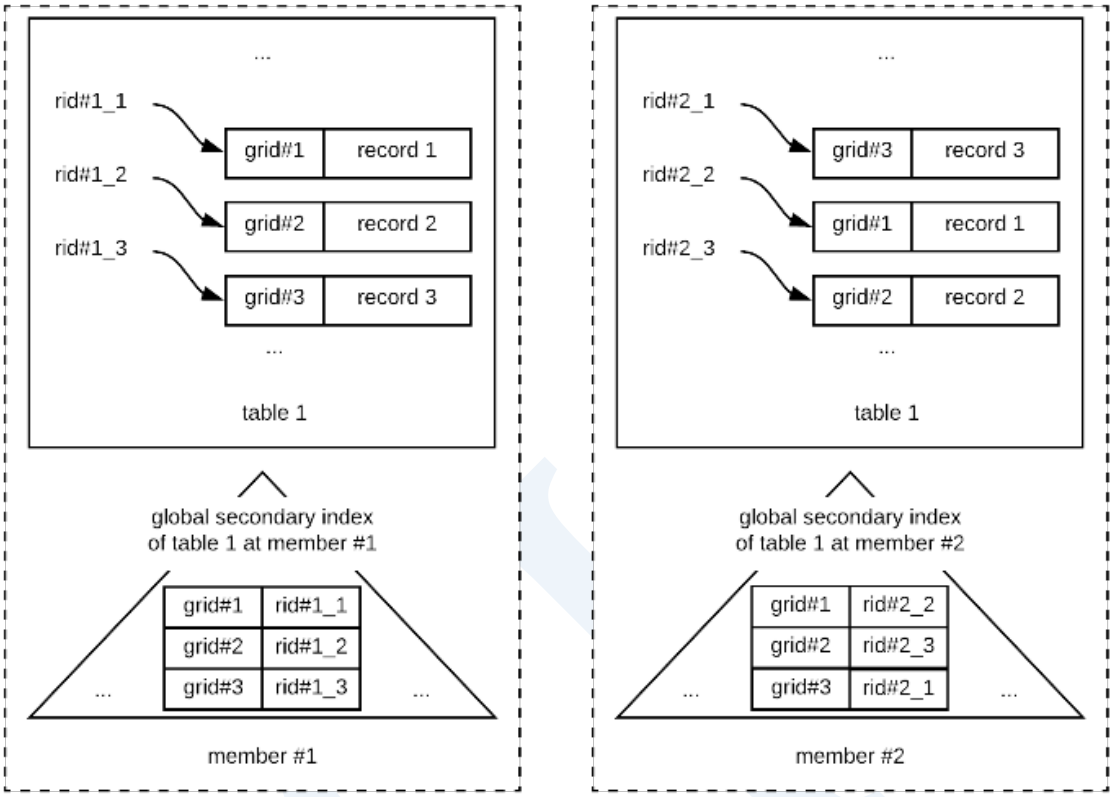


Figure 4-16 Global secondary index 结构

为了执行表的non-deterministic查询必须要有Global Secondary Index如果表中没有Global Secondary Index如下non-deterministic查询将失败

```
gSQL> DELETE FROM T1 LIMIT 1;


ERR-42000(16423): does not support non-deterministic DML in the cluster
system : global secondary index expected
```

可参照ALL_GSI_PLACEDBA_GSI_PLACE或USER_GSI_PLACE等字典来查看表中是否存在全局二级索引

5.SQL Tuning

5.1 SQL Tuning

概要

SQL tuning是分析查询语句并进行修改由此提升性能的过程

通过此过程可缩短查询语句的响应时间或提高作业处理量使查询语句达到所需的性能水平

为了执行SQL tuning需要具备SQL处理过程和optimizer相关知识本章介绍相关内容

SQL处理过程

SQL处理过程如下图

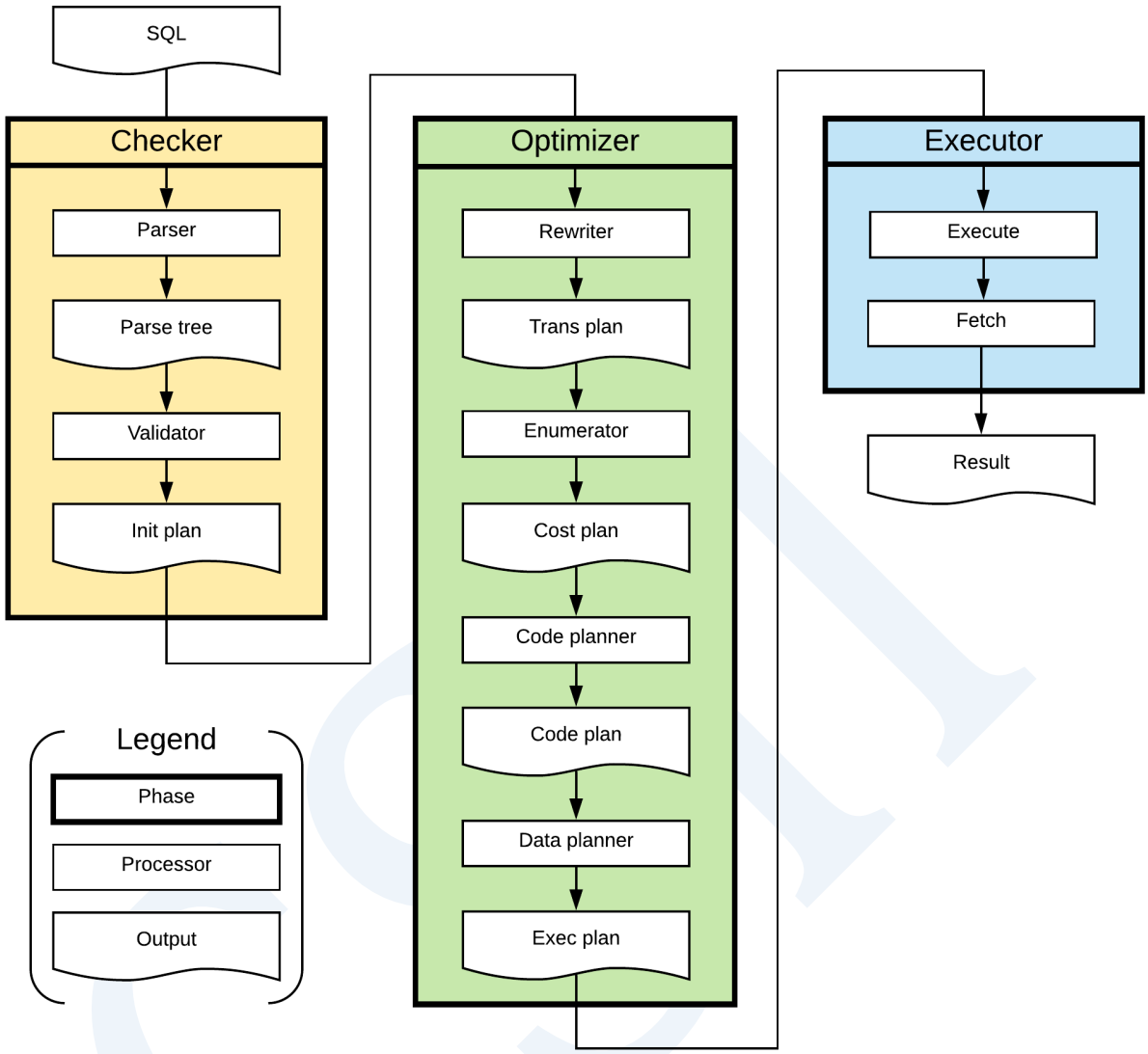


Figure 5-1 SQL processing

用户查询语句经过Parser, Validator, Rewriter, Enumerator, Code Planner, Data Planner, Executor 过程后返回其结果各阶段相关说明为如下

Parser

Parser检查用户输入的SQL语句是否存在语法错误

```
gSQL> SELECT * FORM customer;


ERR-42000(40000): syntax error

SELECT * FORM customer

.........^  ^

Error at line 1
```

SQL语句中没有语法错误则生成parse tree其作为下一阶段的validator的输入参数

Validator

Validator检查输入的parse tree是否存在意义上的错误

检查例如是否存在查询中描述的表或column等的对象用户是否拥有可引用这些对象的权限等

```
gSQL> SELECT * FROM customer;


ERR-42000(16040): table or view does not exist :

SELECT * FROM customer

              *

ERROR at line 1:
```

分析Parse tree后没有意义上的错误时以此为基础生成init plan

Rewriter

Rewriter将SQL语句转换为拥有相同意义的高性能SQL语句

分析init plan后生成trans plan并将其转换为性能更佳的类型的trans plan完成转换的trans plan为enumerator阶段的输入参数

SQL语句转换详细内容参考**Rewriter**

Enumerator

Enumerator基于统计信息计算多个plan的成本并生成最佳的cost plan

关于表的access path, join ordering, join method决定等多种优化方法相关说明参考**Enumerator**

Code Planner

Code planner生成code plan

Code plan是将enumerator最终选择的plan生成为执行计划形式的阶段执行计划由tree结构的节点构成包含如下信息

- Access各表的方法
- 引用表的顺序
- 执行表的join运算的方法
- 数据filter的信息
- 数据grouping及aggregation信息
- 数据排列相关信息

Plan Cache

Code Planner生成的code plan注册在plan cache注册的plan按照plan cache parameters值的一致与否为准区分plan

Parameter	说明
Query text	区分大小写的query text
User information	User ID
Cursor property	需要fetch的query的cursor属性
Bind parameter	Bind parameter数量和各个bind parameter的IN/ OUT属性
Enable atomic	Atomic insertion的使用与否
Enable hint error	Hint中validation error的产生与否

Table 5-1 Plan cache parameters

输入用户查询后查看plan cache中plan cache parameters值是否存在相同的plan存在相同的plan时省略parser-validator-rewriter-enumerator-code planner的执行过程使用注册在plan cache的plan

像这样使用注册在plan cache的plan时可减少用于parser-validator-rewriter-enumerator-code planner执行过程的成本因此可提高相同程度的性能

以下为拥有互不相同的query text值的query示例虽然是相同的query但由于大小写不同无法识别以下query为相同的plan

```
"SELECT * FROM customer"
```

```
"select * from customer"

"Select * From customer"

"SELECT * FROM  customer"
```

> Note:
>
> Plan引用的schema对象（表索引view序列）未commit时无法登记plan

Data Planner

Data planner生成data planData plan在code plan执行时拥有存储中间结果的空间和存储expression结果的临时空间等

Executor

Executor执行code plan和data plan后返回实际执行的结果

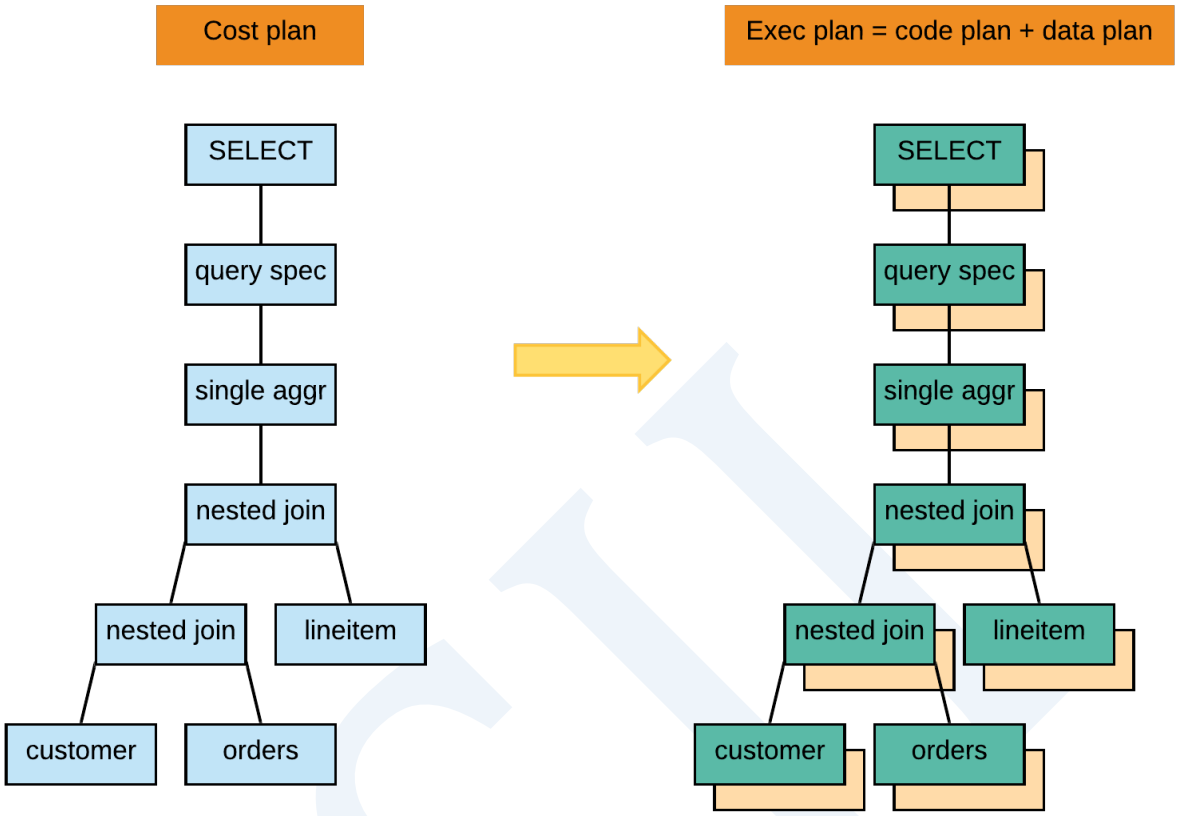


Figure 5-2 Executor

执行计划

执行计划由code plan和data plan构成最上级节点为INSERT, DELETE, UPDATE, SELECT statement的tree形式

执行计划是SQL tuning的最基本的分析工具因为通过执行计划可查看rewriter如何转换了查询语句enumerator选择了哪一个access pathjoin orderjoin method

以下为输出执行计划的语句和执行示例

语句

以下为输出执行计划的语句

```
<explain plan> ::=

      \explain plan [ on | only ] <sql statement>


<sql statement> ::=

        <query expression>

      | <select for update statement>

      | <select statement: single row>

      | <insert statement>

      | <update statement: searched>

      | <delete statement: searched>
```

使用范围及访问权限

执行<explain plan>语句需要有<sql statement>语句的访问权限

语句规则及参数

```
\EXPLAIN PLAN ON

\EXPLAIN PLAN
```

如上指定时执行查询后输出执行计划

```
\EXPLAIN PLAN ONLY
```

如上指定时不执行查询仅输出执行计划

使用示例

如下执行时执行SQL语句并同时输出查询结果和执行计划

执行如下示例的cluster system由G1(G1N1, G1N2), G2(G2N1, G2N2), G3(G3N1, G3N2)构成

Customer是cloned tableorders是data拆分为o_orderkey的sharded table

```
\EXPLAIN PLAN
  SELECT  c_name,
          o_orderdate,
          o_orderstatus
    FROM  customer,
          orders
   WHERE  c_custkey = o_custkey
     AND  o_orderdate >= date '1995-03-15'
     AND  o_orderdate < date '1995-03-15' + interval '1' month;
...

19343 rows selected.

>>>  start print plan

< Execution Plan >
```

Code plan

Data plan

```
|  IDX  |  NODE DESCRIPTION                                                    |                    ROWS |
|-------|----------------------------------------------------------------------|-------------------------|
|    0  |  SELECT STATEMENT                                                    |                   19343 |
|    1  |    QUERY BLOCK ("$QB_IDX_2")                                         |                   19343 |
|    2  |      PLAN BASED CLUSTER                                              | LOCAL/REMOTE      19343 |
|    3  |        NESTED JOIN (INNER JOIN)                                      |                    6389 |
|    4  |          TABLE ACCESS ("ORDERS")                                     |                    6389 |
|    5  |          INDEX ACCESS ("CUSTOMER", "CUSTOMER_PK_INDEX")              | (      6389)       6389 |

     1  -  TARGET : CUSTOMER.C_NAME, ORDERS.O_ORDERDATE, ORDERS.O_ORDERSTATUS
     2  -  SQL : SELECT /*+ KEEP_JOINED_TABLE
                            USE_NL_IN( _A1 )
                            FULL( _A2 )
                            INDEX( _A1, "PUBLIC"."CUSTOMER_PK_INDEX" )
                       */
                        "_A1"."C_NAME", "_A2"."O_ORDERDATE", "_A2"."O_ORDERSTATUS"
                    FROM ( "PUBLIC"."ORDERS"@LOCAL AS "_A2"
                           INNER JOIN
                           "PUBLIC"."CUSTOMER"@LOCAL AS "_A1" ON true
                         ) ALIAS "_A3"
                   WHERE "_A2"."O_ORDERDATE" < :_V0
                     AND "_A2"."O_ORDERDATE" >= :_V1
                     AND "_A1"."C_CUSTKEY" = "_A2"."O_CUSTKEY"
            TARGET DOMAIN : G1(G1N1,G1N2) 6389 rows, G2(G2N1,G2N2) 6462 rows, G3(G3N1,G3N2) 6492 rows
     3  -  JOINED COLUMN : CUSTOMER.C_NAME, ORDERS.O_ORDERDATE, ORDERS.O_ORDERSTATUS
     4  -  HASH SHARD ( # 3 )
            READ COLUMN : ORDERS.O_CUSTKEY, ORDERS.O_ORDERSTATUS, ORDERS.O_ORDERDATE
              PHYSICAL FILTER : ORDERS.O_ORDERDATE < DATE'1995-03-15' + CAST( '1' AS INTERVAL(MONTH) )
                                AND ORDERS.O_ORDERDATE >= DATE'1995-03-15'
     5  -  CLONED
            READ INDEX COLUMN : CUSTOMER.C_CUSTKEY
            READ TABLE COLUMN : CUSTOMER.C_NAME
              MIN RANGE : CUSTOMER.C_CUSTKEY = {ORDERS.O_CUSTKEY}
              MAX RANGE : CUSTOMER.C_CUSTKEY = {ORDERS.O_CUSTKEY}
            FETCH ONE ROW

<<<  end print plan
```

Figure 5-3 Read plan

使用tree形式表示上述执行计划则如下从最下级节点开始执行

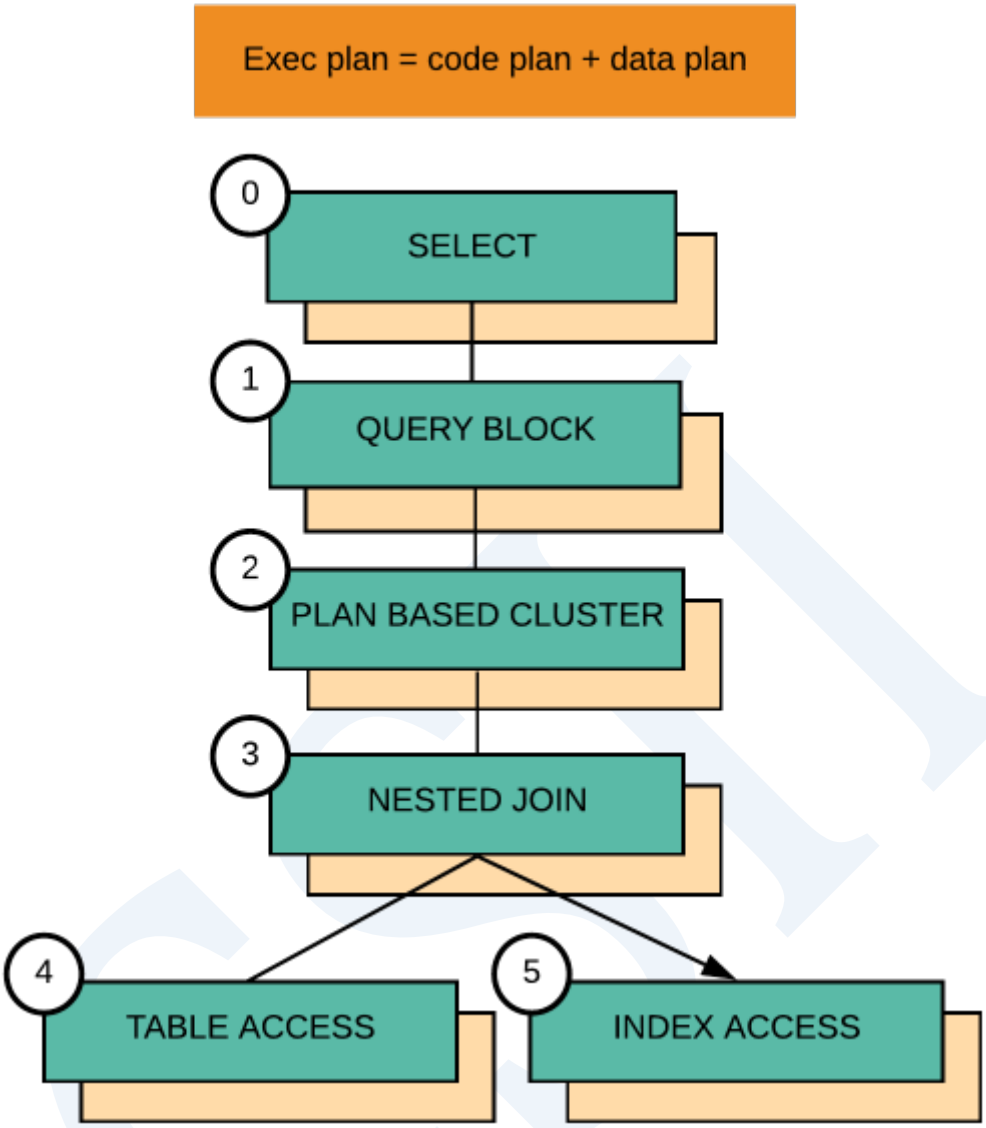


Figure 5-4 Read plan tree

如下执行上述执行tree

首先PLAN BASED CLUSTER(IDX 2)向G1, G2, G3传输如下SQL

```
SELECT /*+ KEEP_JOINED_TABLE
           USE_NL_IN( _A1 )
           FULL( _A2 )
```

```
          INDEX( _A1, "PUBLIC"."CUSTOMER_PK_INDEX" )
      */
      "_A1"."C_NAME", "_A2"."O_ORDERDATE", "_A2"."O_ORDERSTATUS"
 FROM ( "PUBLIC"."ORDERS"@LOCAL AS "_A2"
        INNER JOIN
        "PUBLIC"."CUSTOMER"@LOCAL AS "_A1" ON true
      ) ALIAS "_A3"
WHERE "_A2"."O_ORDERDATE" < :_V0
  AND "_A2"."O_ORDERDATE" >= :_V1
  AND "_A1"."C_CUSTKEY" = "_A2"."O_CUSTKEY"
```

各个G1G2G3中TABLE ACCESS(IDX:4)TABLE ACESS(IDX:5)从customerorders表读取data并执行NESTED JOIN(IDX 3)

PLAN BASED CLUSTER将G1G2G3的NESTED JOIN(IDX 3)结果均获取至local

之后返回其结果

执行计划结构信息

执行计划表的各个column相关信息为如下

- IDX
  - 赋予各个plan node的标识符
- NODE DESCRIPTION
  - Plan node名称
  - 括号中的内容是区分plan node的附加信息

    - 用缩进表示的plan node指下级plan node
        - 从下级plan node开始执行并向上级节点返回其结果
- ROWS
    - Plan node执行的结果记录的数量

各个plan node包含如下优化信息

- 各个表的 **Access Paths**
- 处理**Join**的顺序及Join Method
- **Group By** 处理信息
- **Distinct** 处理信息
- **Single Row Aggregation** 处理信息
- **Order By** 处理信息
- **Cluster Puller** 处理信息
- **Cluster Pusher** 处理信息

5.2 Rewriter

本章介绍rewriter处理的多种query transformation方法

Filter Push Down

将filter下推到最大程度上可下推的位置由此减少需要处理的中间结果

以下为filter push down的示例

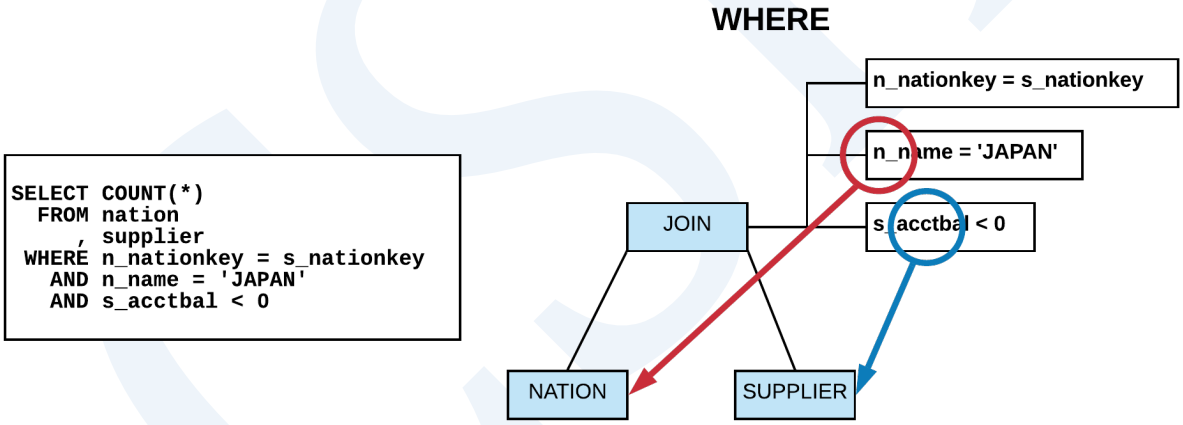


Figure 5-5 Filter push down

执行join前在NATION对满足n_name = JAPAN条件的row进行filtering在SUPPLIER对满足s_acctbal < 0条件的row进行filtering这样处理可减少join处理对象row的数量由此提高性能

以下为向view中进行filter push down的示例

```
CREATE VIEW revenue (supplier_no, total_revenue) AS
  SELECT l_suppkey,
         ROUND( sum(l_extendedprice * (1 - l_discount)), 2)
    FROM lineitem
   WHERE l_shipdate >= date '1996-01-01'
     AND l_shipdate < date '1996-01-01' + interval '3' month
GROUP BY l_suppkey;
```

```
SELECT total_revenue
  FROM revenue
     , supplier
 WHERE supplier_no = 100
```

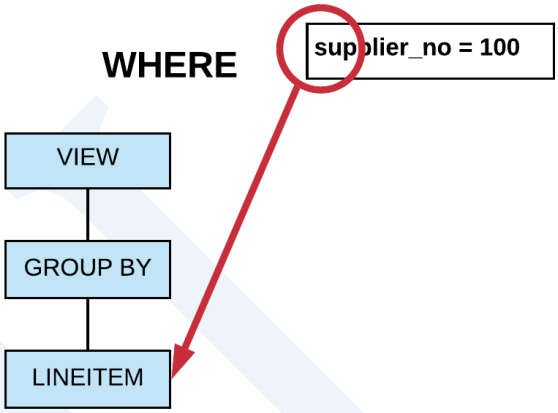


Figure 5-6 Filter push down into view

将supplier_no = 100转化为l_suppkey = 100后下推到lineitem TABLE ACCESS则可减少GROUP BY处理对象row并提高性能

DISTINCT Elimination

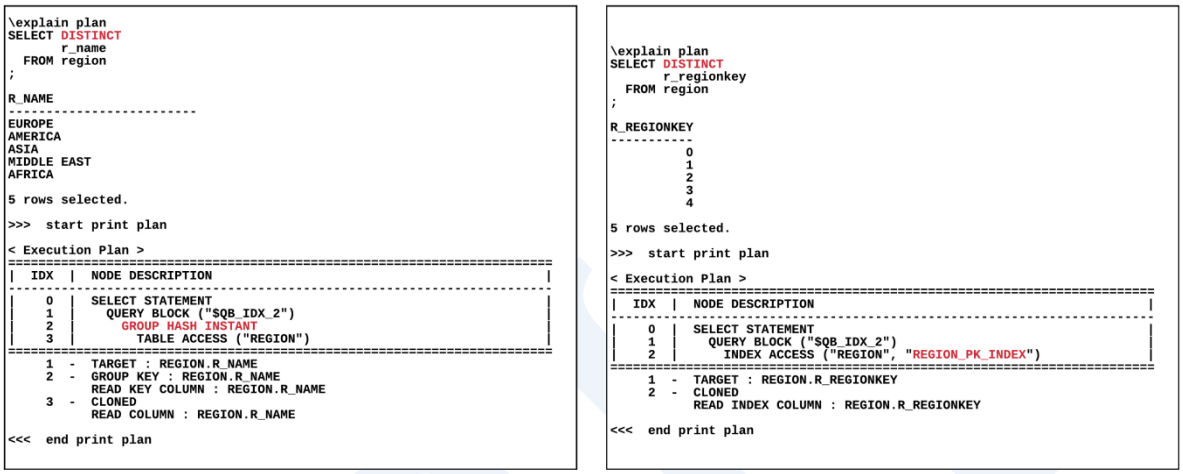
删除不需要的DISTINCT

如下情况可删除DISTINCT

- Single row aggregation的查询结果为一条因此没有DISTINCT也不会出现重复的结果
- 有Group by并group by的所有key column存在于select list时其结果唯一因此没有DISTINCT也不会出现重复的结果
- Primary key的所有key column存在于select list时其结果唯一因此没有DISTINCT也不会出现

重复的结果

以下为DISTINCT elimination的示例



```
\explain plan
SELECT DISTINCT
       r_name
  FROM region
;

R_NAME
-------------------------
EUROPE
AMERICA
ASIA
MIDDLE EAST
AFRICA

5 rows selected.

>>>  start print plan

< Execution Plan >
=========================================================================
|  IDX  |  NODE DESCRIPTION                                             |
-------------------------------------------------------------------------
|    0  |  SELECT STATEMENT                                             |
|    1  |    QUERY BLOCK ("$QB_IDX_2")                                  |
|    2  |      GROUP HASH INSTANT                                       |
|    3  |        TABLE ACCESS ("REGION")                                |
=========================================================================
     1  -  TARGET : REGION.R_NAME
     2  -  GROUP KEY : REGION.R_NAME
           READ KEY COLUMN : REGION.R_NAME
     3  -  CLONED
           READ COLUMN : REGION.R_NAME

<<<  end print plan
```

```
\explain plan
SELECT DISTINCT
       r_regionkey
  FROM region
;

R_REGIONKEY
-----------
          0
          1
          2
          3
          4

5 rows selected.

>>>  start print plan

< Execution Plan >
=========================================================================
|  IDX  |  NODE DESCRIPTION                                             |
-------------------------------------------------------------------------
|    0  |  SELECT STATEMENT                                             |
|    1  |    QUERY BLOCK ("$QB_IDX_2")                                  |
|    2  |      INDEX ACCESS ("REGION", "REGION_PK_INDEX")               |
=========================================================================
     1  -  TARGET : REGION.R_REGIONKEY
     2  -  CLONED
           READ INDEX COLUMN : REGION.R_REGIONKEY

<<<  end print plan
```

Figure 5-7 DISTINCT elimination

左侧execution plan中有处理DISTINCT的GROUP HASH INSTANT节点但右侧execution plan中没有处理DISTINCT的GROUP HASH INSTANT节点

r_regionkey是primary key因此删除DISTINCT其结果也会保障distinct因此可查看到删除了不需要的DISTINCT

ORDER BY Elimination

删除不需要的ORDER BY

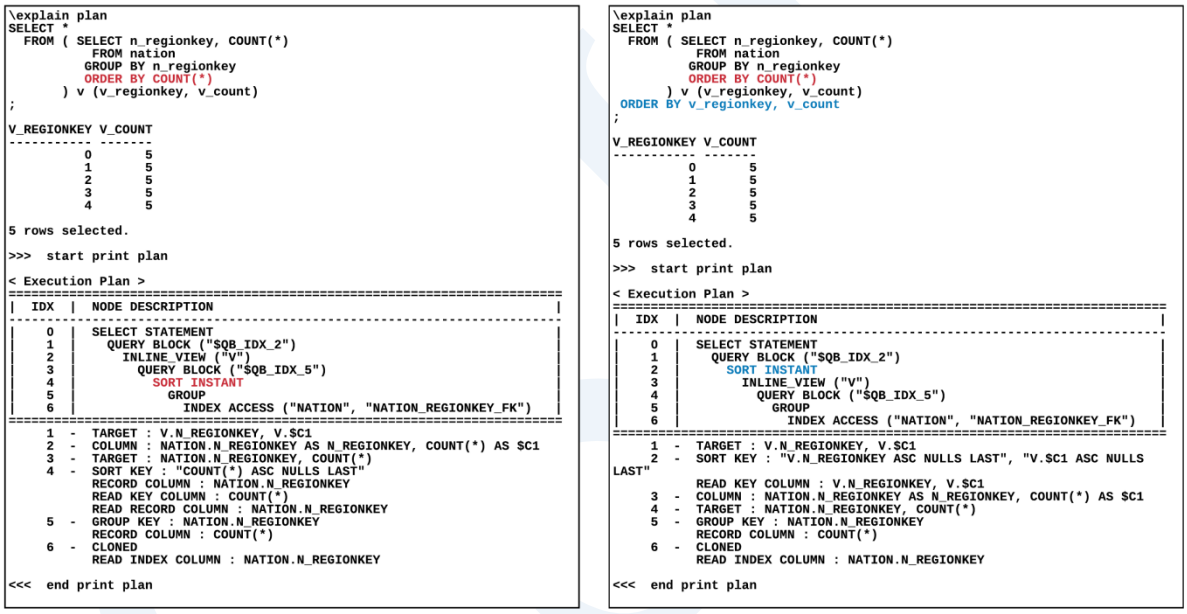
如下情况不需要ORDER BY

- From子句中只有一个view并有group by子句或distinct子句或order by子句时不需要view内

部query block中的order by即使排列也会由于外部的group bydistinctorder by等排列顺序会消失

- 不需要子查询内部query block中的order by子查询的结果仅用于决定是否返回outer query的row

以下为删除ORDER BY的示例



```
\explain plan
SELECT *
  FROM ( SELECT n_regionkey, COUNT(*)
           FROM nation
         GROUP BY n_regionkey
         ORDER BY COUNT(*)
       ) v (v_regionkey, v_count)
;

V_REGIONKEY V_COUNT
----------- -------
          0       5
          1       5
          2       5
          3       5
          4       5

5 rows selected.

>>>  start print plan

< Execution Plan >
=====================================================================
|  IDX  |  NODE DESCRIPTION                                         |
---------------------------------------------------------------------
|     0 |  SELECT STATEMENT                                         |
|     1 |    QUERY BLOCK ("$QB_IDX_2")                              |
|     2 |      INLINE_VIEW ("V")                                    |
|     3 |        QUERY BLOCK ("$QB_IDX_5")                          |
|     4 |          SORT INSTANT                                     |
|     5 |            GROUP                                          |
|     6 |              INDEX ACCESS ("NATION", "NATION_REGIONKEY_FK")   |
=====================================================================
     1  -  TARGET : V.N_REGIONKEY, V.$C1
     2  -  COLUMN : NATION.N_REGIONKEY AS N_REGIONKEY, COUNT(*) AS $C1
     3  -  TARGET : NATION.N_REGIONKEY, COUNT(*)
     4  -  SORT KEY : "COUNT(*) ASC NULLS LAST"
           RECORD COLUMN : NATION.N_REGIONKEY
           READ KEY COLUMN : COUNT(*)
           READ RECORD COLUMN : NATION.N_REGIONKEY
     5  -  GROUP KEY : NATION.N_REGIONKEY
           RECORD COLUMN : COUNT(*)
     6  -  CLONED
           READ INDEX COLUMN : NATION.N_REGIONKEY

<<<  end print plan
```

```
\explain plan
SELECT *
  FROM ( SELECT n_regionkey, COUNT(*)
           FROM nation
         GROUP BY n_regionkey
         ORDER BY COUNT(*)
       ) v (v_regionkey, v_count)
 ORDER BY v_regionkey, v_count
;

V_REGIONKEY V_COUNT
----------- -------
          0       5
          1       5
          2       5
          3       5
          4       5

5 rows selected.

>>>  start print plan

< Execution Plan >
=====================================================================
|  IDX  |  NODE DESCRIPTION                                         |
---------------------------------------------------------------------
|     0 |  SELECT STATEMENT                                         |
|     1 |    QUERY BLOCK ("$QB_IDX_2")                              |
|     2 |      SORT INSTANT                                         |
|     3 |        INLINE_VIEW ("V")                                  |
|     4 |          QUERY BLOCK ("$QB_IDX_5")                        |
|     5 |            GROUP                                          |
|     6 |              INDEX ACCESS ("NATION", "NATION_REGIONKEY_FK")   |
=====================================================================
     1  -  TARGET : V.N_REGIONKEY, V.$C1
     2  -  SORT KEY : "V.N_REGIONKEY ASC NULLS LAST", "V.$C1 ASC NULLS
LAST"
           READ KEY COLUMN : V.N_REGIONKEY, V.$C1
     3  -  COLUMN : NATION.N_REGIONKEY AS N_REGIONKEY, COUNT(*) AS $C1
     4  -  TARGET : NATION.N_REGIONKEY, COUNT(*)
     5  -  GROUP KEY : NATION.N_REGIONKEY
           RECORD COLUMN : COUNT(*)
     6  -  CLONED
           READ INDEX COLUMN : NATION.N_REGIONKEY

<<<  end print plan
```

Figure 5-8 ORDER BY elimination

上图的左右view相同但右侧SQL中order by在view的上级query因此可看到view内部的order by被删除

Simple View Merging

将不包含group bydistinctaggregation等的simple view合并到上级query block

应用simple view merging时optimizer可选择多种access pathjoin orderingjoin method因此可获取更好的执行计划

如下情况无法应用simple view merging

- View内部的query block包含如下项目时
  - Set operator
  - LIMIT, OFFSET
  - DISTINCT
  - GROUP BY
  - Single row aggregation
  - Full outer join
  - Natural join
  - ROWNUM
  - SELECT list中有subquery expression时
- View参与了如下查询时
  - view参与了full outer join
  - view参与了left outer join并在view内部有两个以上的表

以下为合并simple view的示例

```
CREATE VIEW v_nation
(
    v_nationkey,
    v_nation_name,
    v_region_name
```

```
)
AS
SELECT n_nationkey,
       n_name,
       r_name
  FROM nation, region
 WHERE n_regionkey = r_regionkey;
```

```
\EXPLAIN PLAN
SELECT v_nation_name, COUNT(*)
  FROM customer, v_nation
 WHERE c_nationkey = v_nationkey
   AND v_region_name = 'ASIA'
GROUP BY v_nation_name;

V_NATION_NAME             COUNT(*)
------------------------- --------
CHINA                         6024
INDIA                         6042
INDONESIA                     6161
JAPAN                         5948
VIETNAM                       6008

5 rows selected.
```

```
>>>  start print plan


< Execution Plan >

================================================================================
|  IDX  |  NODE DESCRIPTION                                                     |
--------------------------------------------------------------------------------
|    0  |  SELECT STATEMENT                                                     |
|    1  |    QUERY BLOCK ("$QB_IDX_2")                                          |
|    2  |      GROUP HASH INSTANT                                               |
|    3  |        NESTED JOIN (INNER JOIN)                                       |
|    4  |          NESTED JOIN (INNER JOIN)                                     |
|    5  |            TABLE ACCESS ("REGION")                                    |
|    6  |            INDEX ACCESS ("NATION", "NATION_REGIONKEY_FK")             |
|    7  |          INDEX ACCESS ("CUSTOMER", "CUSTOMER_NATIONKEY_FK")           |
================================================================================


     1  -  TARGET : NATION.N_NAME, COUNT(*)
     2  -  GROUP KEY : NATION.N_NAME
           RECORD COLUMN : COUNT(*)
           READ KEY COLUMN : NATION.N_NAME
           READ RECORD COLUMN : COUNT(*)
     3  -  JOINED COLUMN : NATION.N_NAME
     4  -  JOINED COLUMN : NATION.N_NATIONKEY, NATION.N_NAME
     5  -  CLONED
```

```
            READ COLUMN : REGION.R_REGIONKEY, REGION.R_NAME
              PHYSICAL FILTER : REGION.R_NAME = 'ASIA'
      6   -   CLONED
            READ INDEX COLUMN : NATION.N_REGIONKEY
            READ TABLE COLUMN : NATION.N_NATIONKEY, NATION.N_NAME
              MIN RANGE : NATION.N_REGIONKEY = {REGION.R_REGIONKEY}
              MAX RANGE : NATION.N_REGIONKEY = {REGION.R_REGIONKEY}
      7   -   CLONED
            READ INDEX COLUMN : CUSTOMER.C_NATIONKEY
              MIN RANGE : CUSTOMER.C_NATIONKEY = {NATION.N_NATIONKEY}
              MAX RANGE : CUSTOMER.C_NATIONKEY = {NATION.N_NATIONKEY}


<<<  end print plan
```

上述execution plan中没有view可看到view合并到outer query并以如下转换的query形式执行

```
\explain plan
SELECT v_nation_name
     , COUNT(*)
  FROM customer
     , v_nation
 WHERE c_nationkey = v_nationkey
   AND v_region_name = 'ASIA'
 GROUP BY
       v_nation_name
;
```

```
\explain plan
SELECT n_name
     , COUNT(*)
  FROM customer
     , nation
     , region
 WHERE c_nationkey = n_nationkey
   AND r_name = 'ASIA'
   AND n_regionkey = r_regionkey
 GROUP BY
       n_name
;
```

Figure 5-9 Simple view merging

Outer Join Table Elimination

删除不需要的outer join table

如下情况时不需要outer join也不需要访问right table

- 为left outer join，并且
  - ON子句中有'left_table.col = right_table.col' 形式的 predicate
  - 有ON子句的right_table.col的unique index
  - 除ON子句条件外的其他所有子句中不使用right table的column

以下为outer join table elimination的示例

如下示例中对nation表的访问仅存在于ON子句n_nationkey是primary key column因此唯一并且没有null data所以即使删除nation表也不会影响结果

```
\EXPLAIN PLAN
SELECT COUNT(*)
  FROM supplier
       LEFT OUTER JOIN
       nation
       ON s_nationkey = n_nationkey
 WHERE s_acctbal < 0
;

COUNT(*)
--------
```

```
     886


1 row selected.


>>>  start print plan


< Execution Plan >
================================================================================
==
|  IDX  |  NODE DESCRIPTION                                          | ROWS
|
--------------------------------------------------------------------------------
--
|    0  |  SELECT STATEMENT                                          |    1
|
|    1  |    QUERY BLOCK ("$QB_IDX_2")                               |    1
|
|    2  |      TABLE ACCESS ("SUPPLIER")                             |    1
|
================================================================================
==


     1  -  TARGET : COUNT(*)
     2  -  CLONED
           READ COLUMN : SUPPLIER.S_ACCTBAL
```

```
            AGGREGATION : COUNT(*)

              PHYSICAL FILTER : SUPPLIER.S_ACCTBAL < 0


<<<  end print plan
```

如上述execution plan所示对OUTER JOIN和right table的访问均被清除

Outer Join Operation Elimination

删除不需要的outer join operation

- Left outer join的情况
  - Where子句中有属于右侧表的条件子句并其条件子句不是IS NULL时
    - 可变更为Inner join
- Full outer join的情况
  - Where子句中有属于右侧表的条件子句并其条件子句不是IS NULL时
    - 可变更为right outer join
  - Where子句中有属于左侧表的条件子句并其条件子句不是IS NULL时
    - 可变更为left outer join
  - Where子句中有均属于两侧表的条件子句并其条件子句不是IS NULL时
    - 可变更为inner join

以下为删除outer join operation的示例

```
\EXPLAIN PLAN
```

```
  SELECT MAX(COUNT(*))
    FROM customer
         LEFT OUTER JOIN
         orders
         ON  c_custkey = o_custkey
         AND o_comment LIKE '%special%requests%'
   WHERE o_orderpriority = '1-URGENT'
GROUP BY c_custkey;



MAX(COUNT(*))
-------------
            3

1 row selected.

>>>  start print plan

< Execution Plan >
================================================================================
|  IDX  |  NODE DESCRIPTION                                                    |
--------------------------------------------------------------------------------
|    0  |  SELECT STATEMENT                                                    |
|    1  |    QUERY BLOCK ("$QB_IDX_2")                                         |
|    2  |      AGGREGATION BY HASH                                             |
```

```
|    3  |        GROUP                                                    |
|    4  |          HASH JOIN (INNER JOIN)                                 |
|    5  |            INDEX ACCESS ("CUSTOMER", "CUSTOMER_PK_INDEX")      |
|    6  |            HASH JOIN INSTANT                                    |
|    7  |              TABLE ACCESS ("ORDERS")                            |
=====================================================================


     1  -  TARGET : MAX( COUNT(*) )
     2  -  AGGREGATION : MAX( COUNT(*) )
     3  -  GROUP KEY : CUSTOMER.C_CUSTKEY
           RECORD COLUMN : COUNT(*)
     4  -  JOINED COLUMN : CUSTOMER.C_CUSTKEY
     5  -  READ INDEX COLUMN : CUSTOMER.C_CUSTKEY
     6  -  HASH KEY : ORDERS.O_CUSTKEY
           READ KEY COLUMN : ORDERS.O_CUSTKEY
             HASH FILTER : ORDERS.O_CUSTKEY = CUSTOMER.C_CUSTKEY
     7  -  READ COLUMN : ORDERS.O_CUSTKEY, ORDERS.O_ORDERPRIORITY,
                         ORDERS.O_COMMENT
             PHYSICAL FILTER : ORDERS.O_ORDERPRIORITY = '1-URGENT'
             LOGICAL FILTER : ORDERS.O_COMMENT LIKE '%special%requests%'


<<<  end print plan
```

如上所示WHERE子句中有o_orderpriority = '1-URGENT由于此条件right table的结果不可能是所有数据为NULL的row

因此将left outer join更改为inner join其结果也相同

优化EXISTS/NOT EXIST运算Target

在EXISTS或NOT EXISTS中的subquery的SELECT list中减少不需要的expression处理从而提高查询处理性能

EXISTS或NOT EXISTS运算是判断是否存在子查询的结果row的运算符因此子查询的SELECT list中的expression的数量或处理结果不影响运算结果所以将子查询的SELECT list变更为BOOLEAN常数TRUE

以下为优化EXISTS运算target的示例

```
\EXPLAIN PLAN
SELECT o_orderpriority,
       count(*) as order_count
  FROM orders
 WHERE o_orderdate = date '1993-07-01'
   AND EXISTS (
              SELECT /*+ NO_UNNEST */
                     *
                FROM lineitem
               WHERE l_orderkey = o_orderkey
                 AND l_commitdate < l_receiptdate
            )
GROUP BY o_orderpriority
```

```
ORDER BY o_orderpriority;


O_ORDERPRIORITY ORDER_COUNT

--------------- -----------

1-URGENT                113

2-HIGH                  136

3-MEDIUM                112

4-NOT SPECIFIED         103

5-LOW                    97


5 rows selected.


>>>  start print plan


< Execution Plan >

================================================================================

|IDX|  NODE DESCRIPTION                                                        |

--------------------------------------------------------------------------------

| 0 |  SELECT STATEMENT                                                        |

| 1 |    QUERY BLOCK ("$QB_IDX_2")                                             |

| 2 |      SORT INSTANT                                                        |

| 3 |        GROUP HASH INSTANT                                                |

| 4 |          TABLE ACCESS ("ORDERS")                                         |

| 5 |          SUB QUERY LIST                                                  |

| 6 |            INLINE_VIEW ("$V6")                                           |
```

```
| 7 |                QUERY BLOCK ("$QB_IDX_6")                              |
| 8 |                  INDEX ACCESS ("LINEITEM", "LINEITEM_ORDERKEY_FK")    |
==========================================================================


     1  -  TARGET : ORDERS.O_ORDERPRIORITY, COUNT(*) AS ORDER_COUNT
     2  -  SORT KEY : "ORDERS.O_ORDERPRIORITY ASC NULLS LAST"
           RECORD COLUMN : COUNT(*)
           READ KEY COLUMN : ORDERS.O_ORDERPRIORITY
           READ RECORD COLUMN : COUNT(*)
     3  -  GROUP KEY : ORDERS.O_ORDERPRIORITY
           RECORD COLUMN : COUNT(*)
           READ KEY COLUMN : ORDERS.O_ORDERPRIORITY
           READ RECORD COLUMN : COUNT(*)
     4  -  READ COLUMN : ORDERS.O_ORDERKEY, ORDERS.O_ORDERDATE,
                         ORDERS.O_ORDERPRIORITY
             PHYSICAL FILTER : ORDERS.O_ORDERDATE = DATE'1993-07-01'
             POST FILTER : EXISTS( ( $V6.DUMMY_COL ) )
     6  -  COLUMN : $V6.DUMMY_COL AS DUMMY_COL
     7  -  TARGET : NOTHING
     8  -  READ INDEX COLUMN : LINEITEM.L_ORDERKEY
           READ TABLE COLUMN : LINEITEM.L_COMMITDATE,
LINEITEM.L_RECEIPTDATE
             MIN RANGE : LINEITEM.L_ORDERKEY = {ORDERS.O_ORDERKEY}
             MAX RANGE : LINEITEM.L_ORDERKEY = {ORDERS.O_ORDERKEY}
             PHYSICAL TABLE FILTER : LINEITEM.L_RECEIPTDATE >
```

```
LINEITEM.L_COMMITDATE


<<<  end print plan
```

上述示例中lineitem是拥有16个column的表用户查询语句中写着在EXISTS subquery中的SELECT list使用*读取lineitem的所有column但执行时仅获取了是否存在满足条件的row的信息并未获取任何target column value

Quantifier Elimination

如下变更SQL后删除ANY quantifier

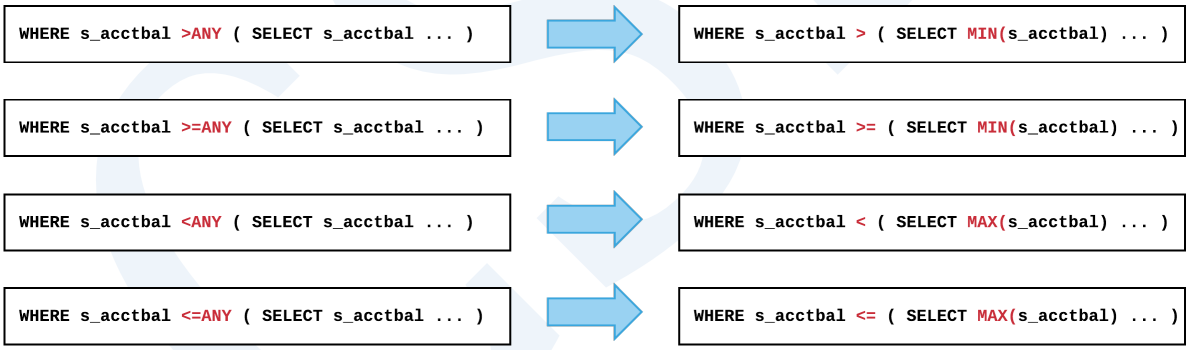


Figure 5-10 Quantifier elimination

以下为删除quantifier的示例

```
\EXPLAIN PLAN
SELECT COUNT(*)
  FROM supplier
 WHERE s_acctbal >ANY ( SELECT s_acctbal
```

```
                          FROM supplier, nation
                         WHERE s_nationkey = n_nationkey
                           AND n_name = 'CHINA' )
;

COUNT(*)
--------
    9950

1 row selected.

>>>  start print plan

< Execution Plan >
================================================================================
=
|  IDX  |  NODE DESCRIPTION
|
--------------------------------------------------------------------------------
-
|    0  |  SELECT STATEMENT
|
|    1  |    QUERY BLOCK ("$QB_IDX_2")
|
|    2  |      TABLE ACCESS ("SUPPLIER")                                      |
```

```
|    3  |  SUB QUERY LIST
|
|    4  |    INLINE_VIEW ("$V4")
|
|    5  |      QUERY BLOCK ("$QB_IDX_6")
|
|    6  |        AGGREGATION BY HASH
|
|    7  |          NESTED JOIN (INNER JOIN)
|
|    8  |            TABLE ACCESS ("NATION")
|
|    9  |            INDEX ACCESS ("SUPPLIER", "SUPPLIER_NATIONKEY_FK")
|
================================================================================
=


     1  -  TARGET : COUNT(*)
     2  -  READ COLUMN : SUPPLIER.S_ACCTBAL
           AGGREGATION : COUNT(*)
             PHYSICAL FILTER : SUPPLIER.S_ACCTBAL > $V4.$C0
     4  -  COLUMN : MIN( SUPPLIER.S_ACCTBAL ) AS $C0
     5  -  TARGET : MIN( SUPPLIER.S_ACCTBAL )
     6  -  AGGREGATION : MIN( SUPPLIER.S_ACCTBAL )
     7  -  JOINED COLUMN : SUPPLIER.S_ACCTBAL
```

```
    8  -  READ COLUMN : NATION.N_NATIONKEY, NATION.N_NAME
            PHYSICAL FILTER : NATION.N_NAME = 'CHINA'
    9  -  READ INDEX COLUMN : SUPPLIER.S_NATIONKEY
          READ TABLE COLUMN : SUPPLIER.S_ACCTBAL
            MIN RANGE : SUPPLIER.S_NATIONKEY = {NATION.N_NATIONKEY}
            MAX RANGE : SUPPLIER.S_NATIONKEY = {NATION.N_NATIONKEY}

<<<  end print plan
```

Transitive Closure

使用join条件在其他表生成常数条件这样可减少join处理量并提高性能

以下为transitive closure示例

```
\explain plan
SELECT COUNT(*)
  FROM customer, supplier
 WHERE c_nationkey = s_nationkey
   AND c_nationkey < 5
;

COUNT(*)
--------
12305943

1 row selected.

>>>  start print plan

< Execution Plan >
=================================================================================================
|  IDX  |  NODE DESCRIPTION                                                          |        ROWS |
-------------------------------------------------------------------------------------------------
|    0  |  SELECT STATEMENT                                                          |           1 |
|    1  |    QUERY BLOCK ("$QB_IDX_2")                                               |           1 |
|    2  |      AGGREGATION BY HASH                                                   |           1 |
|    3  |        MERGE JOIN (INNER JOIN)                                             |    12305943 |
|    4  |          INDEX ACCESS ("CUSTOMER", "CUSTOMER_NATIONKEY_FK")    | (     29914) |       29914 |
|    5  |          INDEX ACCESS ("SUPPLIER", "SUPPLIER_NATIONKEY_FK")    | (      2057) |        2057 |
=================================================================================================
     1  -  TARGET : COUNT(*)
     2  -  AGGREGATION : COUNT(*)
     3  -  JOINED COLUMN : NOTHING
             ON FILTER (Equi) : CUSTOMER.C_NATIONKEY = SUPPLIER.S_NATIONKEY
     4  -  CLONED
             READ INDEX COLUMN : CUSTOMER.C_NATIONKEY
               MAX RANGE : CUSTOMER.C_NATIONKEY < 5
     5  -  CLONED
             READ INDEX COLUMN : SUPPLIER.S_NATIONKEY
               MAX RANGE : SUPPLIER.S_NATIONKEY < 5

<<<  end print plan
```

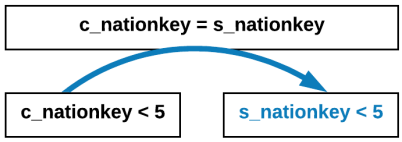


Figure 5-11 Transitive closure

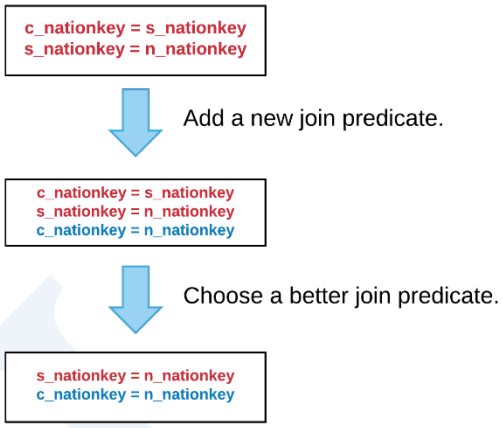
Join Transitive Closure

使用join条件在其他表生成join条件可以选择多种join ordering和join method获取更佳的执行计划

在A = B AND B = C join条件添加A = C join添加的方式

以下为join transitive closure的示例

```
\explain plan
select
    n_name,
    ROUND( sum(l_extendedprice * (1 - l_discount)), 2) as revenue
from
    customer,
    orders,
    lineitem,
    supplier,
    nation,
    region
where
      c_custkey = o_custkey
  and l_orderkey = o_orderkey
  and l_suppkey = s_suppkey
  and c_nationkey = s_nationkey
  and s_nationkey = n_nationkey
  and n_regionkey = r_regionkey
  and r_name = 'ASIA'
  and o_orderdate >= date '1994-01-01'
  and o_orderdate < date '1994-01-01' + interval '1' year
group by
    n_name
order by
    revenue desc;

N_NAME                         REVENUE
------------------------- -----------
INDONESIA                  55502041.17
VIETNAM                    55295087.00
CHINA                      53724494.26
INDIA                      52035512.00
JAPAN                      45410175.70

5 rows selected.
```




```
>>>  start print plan
< Execution Plan >
=============================================================================================
|  IDX  |  NODE DESCRIPTION                                                                 |
---------------------------------------------------------------------------------------------
|    0  |  SELECT STATEMENT                                                                 |
|    1  |    QUERY BLOCK ("$QB_IDX_2")                                                      |
|    2  |      SORT INSTANT                                                                 |
|    3  |        SINGLE CLUSTER                                                             |
|    4  |          SELECT STATEMENT                                                         |
|    5  |            QUERY BLOCK ("$QB_IDX_2")                                              |
|    6  |              GROUP HASH INSTANT                                                   |
|    7  |                HASH JOIN (INNER JOIN)                                             |
|    8  |                  NESTED JOIN (INNER JOIN)                                         |
|    9  |                    NESTED JOIN (INNER JOIN)                                       |
|   10  |                      NESTED JOIN (INNER JOIN)                                     |
|   11  |                        NESTED JOIN (INNER JOIN)                                   |
|   12  |                          TABLE ACCESS ("REGION" AS _A6)                           |
|   13  |                          INDEX ACCESS ("NATION" AS _A5, "NATION_REGIONKEY_FK")    |
|   14  |                        INDEX ACCESS ("CUSTOMER" AS _A4, "CUSTOMER_NATIONKEY_FK")  |
|   15  |                      INDEX ACCESS ("ORDERS" AS _A3, "ORDERS_CUSTKEY_FK")          |
|   16  |                    INDEX ACCESS ("LINEITEM" AS _A2, "LINEITEM_ORDERKEY_FK")       |
|   17  |                  HASH JOIN INSTANT                                                |
|   18  |                    TABLE ACCESS ("SUPPLIER" AS _A1)                               |
=============================================================================================
     1  -  TARGET : NATION.N_NAME, ROUND(SUM(LINEITEM.L_EXTENDEDPRICE * (1 - LINEITEM.L_DISCOUNT)),2) AS REVENUE
     ...
     ...
     ...

    14  -  CLONED
           READ INDEX COLUMN : _A4.C_NATIONKEY
           READ TABLE COLUMN : _A4.C_CUSTKEY
             MIN RANGE : _A4.C_NATIONKEY = {_A5.N_NATIONKEY}
             MAX RANGE : _A4.C_NATIONKEY = {_A5.N_NATIONKEY}
    ...
    ...
    ...

    17  -  HASH KEY : _A1.S_SUPPKEY, _A1.S_NATIONKEY
           READ KEY COLUMN : _A1.S_SUPPKEY, _A1.S_NATIONKEY
             HASH FILTER : _A1.S_SUPPKEY = _A2.L_SUPPKEY AND _A1.S_NATIONKEY = _A5.N_NATIONKEY
           FETCH ONE ROW
    18  -  CLONED
           READ COLUMN : _A1.S_SUPPKEY, _A1.S_NATIONKEY
<<<  end print plan
```

Figure 5-12 Join transitive closure

Subquery Unnesting

Subquery unnesting是将条件子句中的子查询转换为保证相同结果的join语句的功能这样处理后可选择多种access pathjoin methodjoin order因此可获取更佳的执行计划

Subquery可分为如下两种

- Nested subquery (Regular non-scalar subquery)
  - EXISTS/NOT EXIST subquery
  - 比较运算符 (=, >,>=, <, <=, <>) ANY subquery
  - 比较运算符 (=, >,>=, <, <=, <>) ALL subquery
  - IN/NOT IN subquery
- Scalar subquery : 写在WHERE子句或SELECT list仅返回一个值

不是所有子查询会unnest要满足如下约束条件才可以进行subquery unnesting

- 不应包含Set运算符
- Scalar subquery仅可在写在WHERE子句时使用
- 应包含Correlated predicate

Correlated predicate是包含subquery中未定义的outer query block的column的predicate

下列中c.cust_id为correlated columns.cust_id = c.cust_id为correlated predicate

```
SELECT C.cust_last_name, C.country_id

FROM   customers C

WHERE  EXISTS (SELECT 1
```

```
                FROM sales S
               WHERE S.quantity_sold > 1000
                 AND S.cust_id = C.cust_id);
```

Nested Subquery Unnesting

将Subquery转换为semi joinanti-joininner join

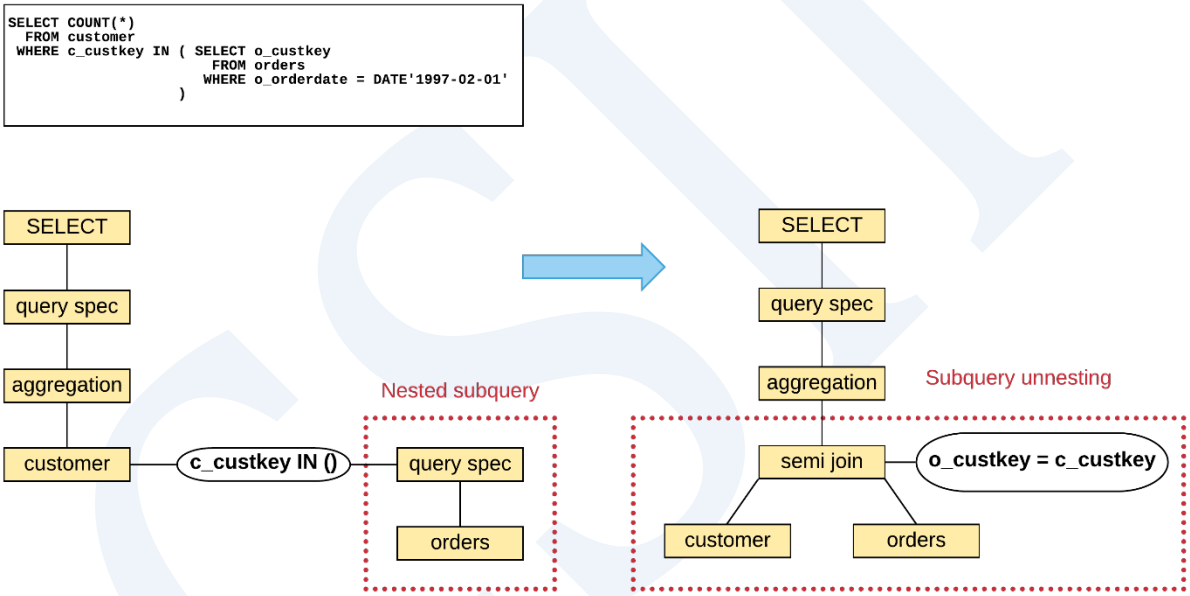


Figure 5-13 Nested subquery unnesting

```
>>>  start print plan
< Execution Plan >
================================================================================
|  IDX  |  NODE DESCRIPTION                                                      |
--------------------------------------------------------------------------------
|    0  |  SELECT STATEMENT                                                      |
|    1  |    QUERY BLOCK ("$QB_IDX_2")                                           |
|    2  |      AGGREGATION BY HASH                                               |
|    3  |        MULTIPLE CLUSTER                                                |
|    4  |          SELECT STATEMENT                                              |
|    5  |            QUERY BLOCK ("$QB_IDX_2")                                   |
|    6  |              SORT INSTANT                                              |
|    7  |                NESTED JOIN (INVERTED SEMI)                             |
|    8  |                  SORT JOIN INSTANT (UNIQUE)                            |
|    9  |                    TABLE ACCESS ("ORDERS" AS _A2)                      |
|   10  |                  INDEX ACCESS ("CUSTOMER" AS _A1, "CUSTOMER_PK_INDEX")  |
================================================================================
     1  -  TARGET : COUNT(*)
     2  -  AGGREGATION : COUNT(*)
     ...
     ...
     ...

    10  -  CLONED
           READ INDEX COLUMN : _A1.C_CUSTKEY
             MIN RANGE : _A1.C_CUSTKEY = {_A2.O_CUSTKEY}
             MAX RANGE : _A1.C_CUSTKEY = {_A2.O_CUSTKEY}
           FETCH ONE ROW
<<<  end print plan
```

Figure 5-14 Nested subquery unnesting plan

Scalar Subquery Unnesting

仅可unnesting WHERE子句中的scalar subquery此时子查询应满足如下条件

- 应为single row aggregation
- 应包含correlated predicate

以下为unnesting scalar subquery的示例

```
\explain plan
SELECT ps_suppkey
  FROM partsupp,
       part
 WHERE ps_partkey = p_partkey
   AND p_name like 'forest%'
   AND ps_availqty > ( SELECT 0.5 * sum(l_quantity)
                         FROM lineitem
                        WHERE l_partkey = ps_partkey
                          AND l_suppkey = ps_suppkey
                          AND l_shipdate >= date'1994-01-01'
                          AND l_shipdate < date'1994-01-01' + interval '1' year
                       )
;
```

```
\explain plan
SELECT ps_suppkey
  FROM partsupp,
       part,
       ( SELECT l_partkey,
                l_suppkey,
                0.5 * sum(l_quantity) as qty
           FROM lineitem
          WHERE l_shipdate >= date'1994-01-01'
            AND l_shipdate < date'1994-01-01' + interval '1' year
          GROUP BY l_partkey, l_suppkey
       )
 WHERE ps_partkey = p_partkey
   AND p_name like 'forest%'
   AND ps_availqty > v1.qty
   AND l_partkey = ps_partkey
   AND l_suppkey = ps_suppkey
;
```

Figure 5-15 Scalar subquery unnesting

```
>>>  start print plan
< Execution Plan >
================================================================================
|  IDX  |  NODE DESCRIPTION                                                      |
--------------------------------------------------------------------------------
|    0  |  SELECT STATEMENT                                                      |
|    1  |    QUERY BLOCK ("$QB_IDX_2")                                           |
|    2  |      NESTED JOIN (INNER JOIN)                                          |
|    3  |        NESTED JOIN (INNER JOIN)                                        |
|    4  |          TABLE ACCESS ("PART")                                         |
|    5  |          INDEX ACCESS ("PARTSUPP", "PARTSUPP_PARTKEY_FK")              |
|    6  |        INLINE_VIEW                                                     |
|    7  |          QUERY BLOCK ("$QB_IDX_8")                                     |
|    8  |            GROUP                                                       |
|    9  |              INDEX ACCESS ("LINEITEM", "LINEITEM_PARTKEY_SUPPKEY_FK")  |
================================================================================
     1  -  TARGET : PARTSUPP.PS_SUPPKEY
     2  -  JOINED COLUMN : PARTSUPP.PS_SUPPKEY
     3  -  JOINED COLUMN : PARTSUPP.PS_PARTKEY, PARTSUPP.PS_SUPPKEY, PARTSUPP.PS_AVAILQTY
     4  -  READ COLUMN : PART.P_PARTKEY, PART.P_NAME
          LOGICAL FILTER : PART.P_NAME LIKE 'forest%'
     5  -  READ INDEX COLUMN : PARTSUPP.PS_PARTKEY
          READ TABLE COLUMN : PARTSUPP.PS_SUPPKEY, PARTSUPP.PS_AVAILQTY
            MIN RANGE : PARTSUPP.PS_PARTKEY = {PART.P_PARTKEY}
            MAX RANGE : PARTSUPP.PS_PARTKEY = {PART.P_PARTKEY}
     6  -  COLUMN : LINEITEM.L_PARTKEY AS L_PARTKEY,
                    LINEITEM.L_SUPPKEY AS L_SUPPKEY,
                    0.5 * SUM( LINEITEM.L_QUANTITY ) AS $C2
     7  -  TARGET : LINEITEM.L_PARTKEY,
                    LINEITEM.L_SUPPKEY,
                    0.5 * SUM( LINEITEM.L_QUANTITY )
     8  -  GROUP KEY : LINEITEM.L_PARTKEY, LINEITEM.L_SUPPKEY
          RECORD COLUMN : SUM( LINEITEM.L_QUANTITY )
            LOGICAL FILTER : {PARTSUPP.PS_AVAILQTY} > ( 0.5 * SUM(LINEITEM.L_QUANTITY) )
     9  -  READ INDEX COLUMN : LINEITEM.L_PARTKEY, LINEITEM.L_SUPPKEY
          READ TABLE COLUMN : LINEITEM.L_QUANTITY, LINEITEM.L_SHIPDATE
            MIN RANGE : LINEITEM.L_PARTKEY = {PARTSUPP.PS_PARTKEY}
                        AND LINEITEM.L_SUPPKEY = {PARTSUPP.PS_SUPPKEY}
            MAX RANGE : LINEITEM.L_PARTKEY = {PARTSUPP.PS_PARTKEY}
                        AND LINEITEM.L_SUPPKEY = {PARTSUPP.PS_SUPPKEY}
            PHYSICAL TABLE FILTER : LINEITEM.L_SHIPDATE < DATE'1994-01-01' + CAST( '1'
AS INTERVAL(YEAR) ) AND LINEITEM.L_SHIPDATE >= DATE'1994-01-01'
<<<  end print plan

;
```

Figure 5-16 Scalar subquery unnesting plan

Complex View Merging

将包含group by的view与上级query block进行合并

通过group by无法大幅减少中间结果与上级query block的join filtering效果大时使用会更有效因

此在通过group by可大幅减少中间结果时应用反而会降低性能

如下情况无法应用complex view merging

- View内部query block包含如下项目
  - SET operator
  - ROWNUM
  - LIMIT/OFFSET
  - Single row aggregation
  - ORDER BY
  - 包含Subquery的SELECT list
  - FULL OUTER JOIN
  - NATURAL JOIN
- View参与如下查询时
  - Inner join以外的join
  - 没有equi join predicate时

以下为合并complex view的示例

```
\explain plan
SELECT ps_suppkey
  FROM partsupp,
       part,
       ( SELECT l_partkey,
                l_suppkey,
                0.5 * sum(l_quantity) qty
           FROM lineitem
          WHERE l_shipdate >= date'1994-01-01'
            AND l_shipdate < date'1994-01-01' + interval '1' year
          GROUP BY l_partkey, l_suppkey
       ) v1
 WHERE ps_partkey = p_partkey
   AND p_name like 'forest%'
   AND l_partkey = ps_partkey
   AND l_suppkey = ps_suppkey
   AND ps_availqty > v1.qty
;
```

```
\explain plan
SELECT ps_suppkey
  FROM ( SELECT MAX(ps_suppkey) as ps_suppkey
           FROM lineitem,
                partsupp,
                part
          WHERE ps_partkey = p_partkey
            AND p_name like 'forest%'
            AND l_partkey = ps_partkey
            AND l_suppkey = ps_suppkey
            AND l_shipdate >= date'1994-01-01'
            AND l_shipdate < date'1994-01-01' + interval '1' year
          GROUP BY l_partkey, l_suppkey, partsupp.rowid, part.rowid
          HAVING MAX(ps_availqty) > 0.5 * sum(l_quantity)
       ) v1;
;
```

Figure 5-17 Complex view merging

```
< Execution Plan >
================================================================================
|  IDX  |  NODE DESCRIPTION                                                    |
--------------------------------------------------------------------------------
|    0  |  SELECT STATEMENT                                                    |
|    1  |    QUERY BLOCK ("$QB_IDX_2")                                         |
|    2  |      INLINE_VIEW ("V1")                                              |
|    3  |        QUERY BLOCK ("$QB_IDX_9")                                     |
|    4  |          GROUP HASH INSTANT                                          |
|    5  |            NESTED JOIN (INNER JOIN)                                  |
|    6  |              NESTED JOIN (INNER JOIN)                                |
|    7  |                TABLE ACCESS ("PART")                                 |
|    8  |                INDEX ACCESS ("LINEITEM", "LINEITEM_PARTKEY_SUPPKEY_FK")   |
|    9  |              INDEX ACCESS ("PARTSUPP", "PARTSUPP_PK_INDEX")          |
================================================================================
     1  -  TARGET : V1.PS_SUPPKEY
     2  -  COLUMN : PS_SUPPKEY AS PS_SUPPKEY
     3  -  TARGET : MAX( PARTSUPP.PS_SUPPKEY ) AS PS_SUPPKEY
     4  -  GROUP KEY : LINEITEM.L_PARTKEY, LINEITEM.L_SUPPKEY, PARTSUPP.$PHYSICAL_ROWID, PART.$PHYSICAL_ROWID
           RECORD COLUMN : SUM( LINEITEM.L_QUANTITY ), MAX( PARTSUPP.PS_SUPPKEY )
           READ RECORD COLUMN : SUM( LINEITEM.L_QUANTITY ), MAX( PARTSUPP.PS_SUPPKEY )
             LOGICAL FILTER : PARTSUPP.PS_AVAILQTY > ( 0.5 * SUM( LINEITEM.L_QUANTITY ) )
     5  -  JOINED COLUMN : LINEITEM.L_PARTKEY, LINEITEM.L_SUPPKEY, PARTSUPP.$PHYSICAL_ROWID, PART.$PHYSICAL_ROWID,
PARTSUPP.PS_AVAILQTY, LINEITEM.L_QUANTITY, PARTSUPP.PS_SUPPKEY
     6  -  JOINED COLUMN : PART.P_PARTKEY, LINEITEM.L_PARTKEY, LINEITEM.L_SUPPKEY, PART.$PHYSICAL_ROWID,
LINEITEM.L_QUANTITY
     7  -  READ COLUMN : PART.P_PARTKEY, PART.P_NAME
             LOGICAL FILTER : PART.P_NAME LIKE 'forest%'
     8  -  READ INDEX COLUMN : LINEITEM.L_PARTKEY, LINEITEM.L_SUPPKEY
           READ TABLE COLUMN : LINEITEM.L_QUANTITY, LINEITEM.L_SHIPDATE
             MIN RANGE : LINEITEM.L_PARTKEY = {PART.P_PARTKEY}
             MAX RANGE : LINEITEM.L_PARTKEY = {PART.P_PARTKEY}
             PHYSICAL TABLE FILTER : LINEITEM.L_SHIPDATE < DATE'1994-01-01' + CAST( '1' AS INTERVAL(YEAR) ) AND
LINEITEM.L_SHIPDATE >= DATE'1994-01-01'
     9  -  READ INDEX COLUMN : PARTSUPP.PS_PARTKEY, PARTSUPP.PS_SUPPKEY
           READ TABLE COLUMN : PARTSUPP.PS_AVAILQTY
             MIN RANGE : PARTSUPP.PS_PARTKEY = {PART.P_PARTKEY} AND
                         PARTSUPP.PS_PARTKEY = {LINEITEM.L_PARTKEY} AND
                         PARTSUPP.PS_SUPPKEY = {LINEITEM.L_SUPPKEY}
             MAX RANGE : PARTSUPP.PS_PARTKEY = {PART.P_PARTKEY} AND
                         PARTSUPP.PS_PARTKEY = {LINEITEM.L_PARTKEY} AND
                         PARTSUPP.PS_SUPPKEY = {LINEITEM.L_SUPPKEY}
           FETCH ONE ROW
<<<  end print plan
```

Figure 5-18 Complex view merging plan

5.3 Enumerator

Enumerator以统计信息为基础计算成本并查找最有效的plan

输入trans plan后生成对此的多种形式的cost plan并基于统计信息计算各个cost plan的cost之后选择cost最小的plan

Access Paths

Access path是访问单张表的方法有如下几种

- Table access
- Index access
- Rowid access
- Index concat

计算根据上述方法的各个cost并将cost最小的access选择为执行计划

Table Access

Table access是按照表的存储方式直接读取所有row的方式

如下情况选择table access

- 没有Index的情况

- 即使有Index但没有可使用index的predicate的情况 (例: WHERE col1 + 1 = 10)
- 没有Index的第一个key column的条件因此表访问成本大的情况

  (例: 仅有composite index (col1, col2)的第二个column的条件的情况, WHERE col2 > 3)
- 由于表小而table access的成本低于index access的情况
- 用户基于表访问hint的情况(例: FULL(t1))
- Index selectivity不好或数据分布严重不均衡的情况

以下为使用table access的示例

```
\EXPLAIN PLAN

SELECT r_regionkey, r_name

  FROM region;


R_REGIONKEY R_NAME

----------- -------------------------

          0 AFRICA

          1 AMERICA

          2 ASIA

          3 EUROPE

          4 MIDDLE EAST


5 rows selected.


>>>  start print plan


< Execution Plan >
```

```
==========================================================================
|  IDX   |  NODE DESCRIPTION                                             |
--------------------------------------------------------------------------
|    0   |  SELECT STATEMENT                                             |
|    1   |    QUERY BLOCK ("$QB_IDX_2")                                  |
|    2   |      TABLE ACCESS ("REGION")                                  |
==========================================================================


     1  -  TARGET : REGION.R_REGIONKEY, REGION.R_NAME
     2  -  READ COLUMN : REGION.R_REGIONKEY, REGION.R_NAME


<<<  end print plan
```

Index Access

使用索引读取表的方式

Index access中有index full scan, index unique scan, index range scan, in key range scan方式由cost estimation选择最佳方式

Index Full Scan

扫描所有索引

以下为 index full scan的示例

```
\EXPLAIN PLAN
```

```
SELECT p_partkey
  FROM part;
...


200000 rows selected.


>>>  start print plan


< Execution Plan >
=============================================================================
|  IDX  |  NODE DESCRIPTION                                                 |
-----------------------------------------------------------------------------
|    0  |  SELECT STATEMENT                                                 |
|    1  |    QUERY BLOCK ("$QB_IDX_2")                                      |
|    2  |      INDEX ACCESS ("PART", "PART_PK_INDEX")                       |
=============================================================================


     1  -  TARGET : PART.P_PARTKEY
     2  -  READ INDEX COLUMN : PART.P_PARTKEY


<<<  end print plan
```

仅查询PART_PK_INDEX的key column的p_partkey因此相比读取整个表读取整个索引后导出结果的成本更低

Index Unique Scan

通过索引仅fetch一个row

以下为index unique scan的示例

```
\EXPLAIN PLAN
SELECT *
  FROM part
 WHERE p_partkey = 1;
...
1 row selected.


>>>  start print plan


< Execution Plan >
============================================================================
|  IDX  |  NODE DESCRIPTION                                                |
----------------------------------------------------------------------------
|    0  |  SELECT STATEMENT                                                |
|    1  |    QUERY BLOCK ("$QB_IDX_2")                                     |
|    2  |      INDEX ACCESS ("PART", "PART_PK_INDEX")                      |
============================================================================


     1  -  TARGET : PART.P_PARTKEY, PART.P_NAME, PART.P_MFGR,
PART.P_BRAND,
```

```
                    PART.P_TYPE, PART.P_SIZE, PART.P_CONTAINER,
                    PART.P_RETAILPRICE, PART.P_COMMENT
     2  -  READ INDEX COLUMN : PART.P_PARTKEY
           READ TABLE COLUMN : PART.P_NAME, PART.P_MFGR, PART.P_BRAND,
                               PART.P_TYPE, PART.P_SIZE, PART.P_CONTAINER,
                               PART.P_RETAILPRICE, PART.P_COMMENT
             MIN RANGE : PART.P_PARTKEY = 1
             MAX RANGE : PART.P_PARTKEY = 1
           FETCH ONE ROW


<<<  end print plan
```

Index Range Scan

通过索引读取满足predicate条件的区间的row该结果row通过index读取因此对于index key column排序

以下为index range scan的示例

```
\EXPLAIN PLAN
SELECT p_partkey, p_brand, p_type
  FROM part
 WHERE p_partkey >= 10
   AND p_partkey < 20;


P_PARTKEY P_BRAND    P_TYPE
```

```
--------- ---------- -------------------------

       10 Brand#54   LARGE BURNISHED STEEL

       11 Brand#25   STANDARD BURNISHED NICKEL

       12 Brand#33   MEDIUM ANODIZED STEEL

       13 Brand#55   MEDIUM BURNISHED NICKEL

       14 Brand#13   SMALL POLISHED STEEL

       15 Brand#15   LARGE ANODIZED BRASS

       16 Brand#32   PROMO PLATED TIN

       17 Brand#43   ECONOMY BRUSHED STEEL

       18 Brand#11   SMALL BURNISHED STEEL

       19 Brand#23   SMALL ANODIZED NICKEL


10 rows selected.


>>>  start print plan


< Execution Plan >

================================================================================

|  IDX  |  NODE DESCRIPTION                                                    |

--------------------------------------------------------------------------------

|    0  |  SELECT STATEMENT                                                    |

|    1  |    QUERY BLOCK ("$QB_IDX_2")                                         |

|    2  |      INDEX ACCESS ("PART", "PART_PK_INDEX")                          |

================================================================================
```

```
    1  -  TARGET : PART.P_PARTKEY, PART.P_BRAND, PART.P_TYPE

    2  -  READ INDEX COLUMN : PART.P_PARTKEY

          READ TABLE COLUMN : PART.P_BRAND, PART.P_TYPE

            MIN RANGE : PART.P_PARTKEY >= 10

            MAX RANGE : PART.P_PARTKEY IS NOT NULL AND PART.P_PARTKEY < 20


<<<  end print plan
```

In Key Range Scan

如下有predicate时可执行in key range scan

```
( col1, col2 ) IN ( (val1, val2), (val3, val4) )
```

- WHERE子句中有IN 或 =ANY list function filter
- col1, col2应为base column(应为没有运算或function的column)
- 可将对应col1的(val1, val3)转换为一个数据类型
- 可将对应col2的(val2, val4)转换为一个数据类型

以下为 in key range scan的示例

```
\EXPLAIN PLAN
  SELECT o_orderstatus
    FROM orders
   WHERE o_custkey IN ( 1, 10, 100, 1000, 10000 );
...
91 rows selected.
```

```
>>>  start print plan


< Execution Plan >

=============================================================================
|  IDX  |  NODE DESCRIPTION                                                 |
-----------------------------------------------------------------------------
|    0  |  SELECT STATEMENT                                                 |
|    1  |    QUERY BLOCK ("$QB_IDX_2")                                      |
|    2  |      INDEX ACCESS ("ORDERS", "ORDERS_CUSTKEY_FK")                 |
=============================================================================


     1  -  TARGET : ORDERS.O_ORDERSTATUS
     2  -  READ INDEX COLUMN : ORDERS.O_CUSTKEY
           READ TABLE COLUMN : ORDERS.O_ORDERSTATUS
           IN KEY RANGE
             MIN RANGE : ORDERS.O_CUSTKEY = ?
             MAX RANGE : ORDERS.O_CUSTKEY = ?


<<<  end print plan
```

Rowid Access

Rowid access是使用rowid直接访问对应page的方式

使用rowid access必须要有对rowid的predicateRowid access通常比其他access path快因此只要

有rowid的predicate则estimator将rowid access选择为最佳access path的可能性会更高

```
gSQL> \EXPLAIN PLAN
SELECT p_brand, p_type
  FROM part
 WHERE rowid = 'AAAAAAAAYe8AACAAAEMJAAA';


P_BRAND    P_TYPE
---------- -------------------------
Brand#33   STANDARD POLISHED COPPER


1 row selected.


>>>  start print plan


< Execution Plan >
================================================================================
|  IDX  |  NODE DESCRIPTION                                                    |
--------------------------------------------------------------------------------
|    0  |  SELECT STATEMENT                                                    |
|    1  |    QUERY BLOCK ("$QB_IDX_2")                                         |
|    2  |      ROWID ACCESS ("PART")                                           |
================================================================================


     1  -  TARGET : PART.P_BRAND, PART.P_TYPE
```

```
     2  -  READ COLUMN : PART.P_BRAND, PART.P_TYPE
              ROWID FILTER : PART.ROWID = 'AAAAAAAAYe8AACAAAEMJAAA'


<<<  end print plan
```

Index Concat

Index concat是整合多个index access并成为一个结果的方式因此仅限于存在OR predicate并各个predicate可index access时使用

存在OR predicate时estimator计算index concat的cost后其cost小于其他access path时选择此方式

以下为使用index concat的示例

```
gSQL> \EXPLAIN PLAN
SELECT p_brand, p_type
  FROM part
 WHERE p_partkey = 1 OR p_partkey = 20;


P_BRAND    P_TYPE
---------- ----------------------
Brand#13   PROMO BURNISHED COPPER
Brand#12   LARGE POLISHED NICKEL


2 rows selected.
```

```
>>>  start print plan


< Execution Plan >

=============================================================================
|  IDX  |  NODE DESCRIPTION                                                 |
-----------------------------------------------------------------------------
|    0  |  SELECT STATEMENT                                                 |
|    1  |    QUERY BLOCK ("$QB_IDX_2")                                      |
|    2  |      CONCAT (Compare Nothing)                                     |
|    3  |        INDEX ACCESS ("PART", "PART_PK_INDEX")                     |
|    4  |        INDEX ACCESS ("PART", "PART_PK_INDEX")                     |
=============================================================================


     1  -  TARGET : PART.P_BRAND, PART.P_TYPE
     2  -  CONCAT COLUMN : PART.P_BRAND, PART.P_TYPE
     3  -  READ INDEX COLUMN : PART.P_PARTKEY
           READ TABLE COLUMN : PART.P_BRAND, PART.P_TYPE
             MIN RANGE : PART.P_PARTKEY = 1
             MAX RANGE : PART.P_PARTKEY = 1
           FETCH ONE ROW
     4  -  READ INDEX COLUMN : PART.P_PARTKEY
           READ TABLE COLUMN : PART.P_BRAND, PART.P_TYPE
             MIN RANGE : PART.P_PARTKEY = 20
             MAX RANGE : PART.P_PARTKEY = 20
           FETCH ONE ROW
```

```
<<<  end print plan
```

Join

Join是组合两个以上的表并成为一个结果集合的过程

此时定义表之间的关系的就是join condition没有join condition时所有表row的乘积成为新的结果集合

Estimator根据join type生成考虑join orderjoin methodaccess path的多种cost plan计算其cost后选择最佳cost plan**Access Paths**在上述章节中有介绍本章节介绍join typejoin methodjoin order

Join Type

Cross Join

没有join condition因此两个表的笛卡尔积成为新的join结果

以下为cross join的示例

```
gSQL> \EXPLAIN PLAN SELECT r_regionkey, n_nationkey FROM region, nation;


R_REGIONKEY N_NATIONKEY
----------- -----------
          0           0
```

```
          0           1

          0           2

          0           3

          0           4

          0           5

          0           6

          0           7

          0           8

          0           9

          0          10

          0          11

          0          12

          0          13

          0          14

          0          15

          0          16

          0          17

          0          18

          0          19


R_REGIONKEY N_NATIONKEY

----------- -----------

          0          20

          0          21

          0          22
```

```
          0          23

          0          24

          1           0

          1           1

         ...         ...

          4          24




125 rows selected.


>>>  start print plan


< Execution Plan >

=============================================================================

|  IDX  |  NODE DESCRIPTION                                                 |

-----------------------------------------------------------------------------

|    0  |  SELECT STATEMENT                                                 |

|    1  |    QUERY BLOCK ("$QB_IDX_2")                                      |

|    2  |      NESTED JOIN (INNER JOIN)                                     |

|    3  |        INDEX ACCESS ("REGION", "REGION_PK_INDEX")                 |

|    4  |        INDEX ACCESS ("NATION", "NATION_PK_INDEX")                 |

=============================================================================


     1  -  TARGET : REGION.R_REGIONKEY, NATION.N_NATIONKEY
```

```
     2  -  JOINED COLUMN : REGION.R_REGIONKEY, NATION.N_NATIONKEY

     3  -  READ INDEX COLUMN : REGION.R_REGIONKEY

     4  -  READ INDEX COLUMN : NATION.N_NATIONKEY


<<<  end print plan
```

Inner Join

两个表的笛卡尔积中只有满足join conditionrow成为结果集合

以下为inner join的示例

```
gSQL> \EXPLAIN PLAN
SELECT r_regionkey, n_nationkey
  FROM region, nation
 WHERE r_regionkey = n_nationkey;


R_REGIONKEY N_NATIONKEY
----------- -----------
          0           0
          1           1
          2           2
          3           3
          4           4


5 rows selected.
```

```
>>>  start print plan


< Execution Plan >

=============================================================================
|  IDX  |  NODE DESCRIPTION                                                 |
-----------------------------------------------------------------------------
|    0  |  SELECT STATEMENT                                                 |
|    1  |    QUERY BLOCK ("$QB_IDX_2")                                      |
|    2  |      NESTED JOIN (INNER JOIN)                                     |
|    3  |        INDEX ACCESS ("REGION", "REGION_PK_INDEX")                 |
|    4  |        INDEX ACCESS ("NATION", "NATION_PK_INDEX")                 |
=============================================================================


     1  -  TARGET : REGION.R_REGIONKEY, NATION.N_NATIONKEY
     2  -  JOINED COLUMN : REGION.R_REGIONKEY, NATION.N_NATIONKEY
     3  -  READ INDEX COLUMN : REGION.R_REGIONKEY
     4  -  READ INDEX COLUMN : NATION.N_NATIONKEY
              MIN RANGE : NATION.N_NATIONKEY = {REGION.R_REGIONKEY}
              MAX RANGE : NATION.N_NATIONKEY = {REGION.R_REGIONKEY}
            FETCH ONE ROW


<<<  end print plan
```

Outer Join

在两个表的笛卡尔积中作为结果返回满足join condition的rowouter table的row即使不满足join condition也会返回为结果即要输出不满足join condition的row时使用outer join

此时属于inner table的值为NULL padding

Left outer join中left table为outer table

因此如下示例中left table的part为outer table并输出不满足join condition的row此时inner table的partsupp值为NULL padding

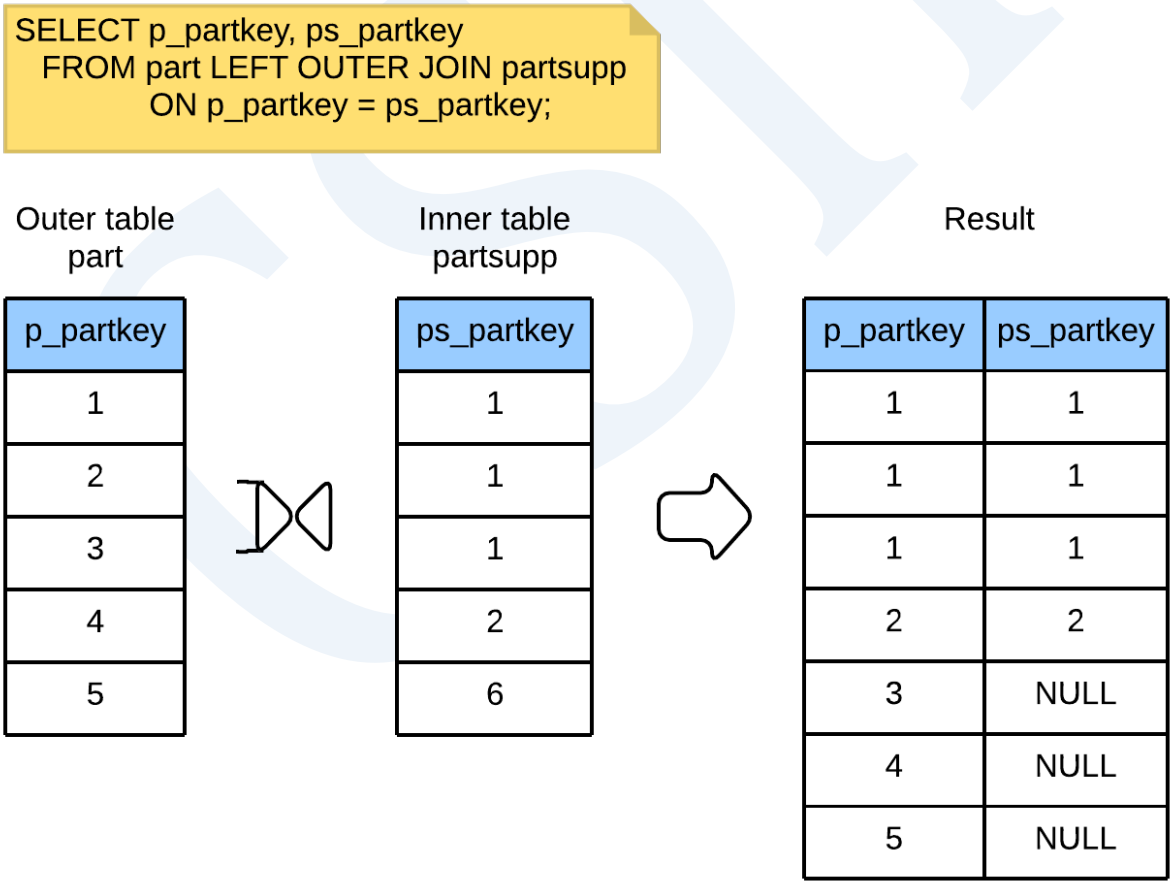


Figure 5-19 Left outer join

Right outer join中right table为outer table

因此如下示例中right table的partsupp为outer table并输出不满足join condition的row此时inner table的parts值为NULL padding

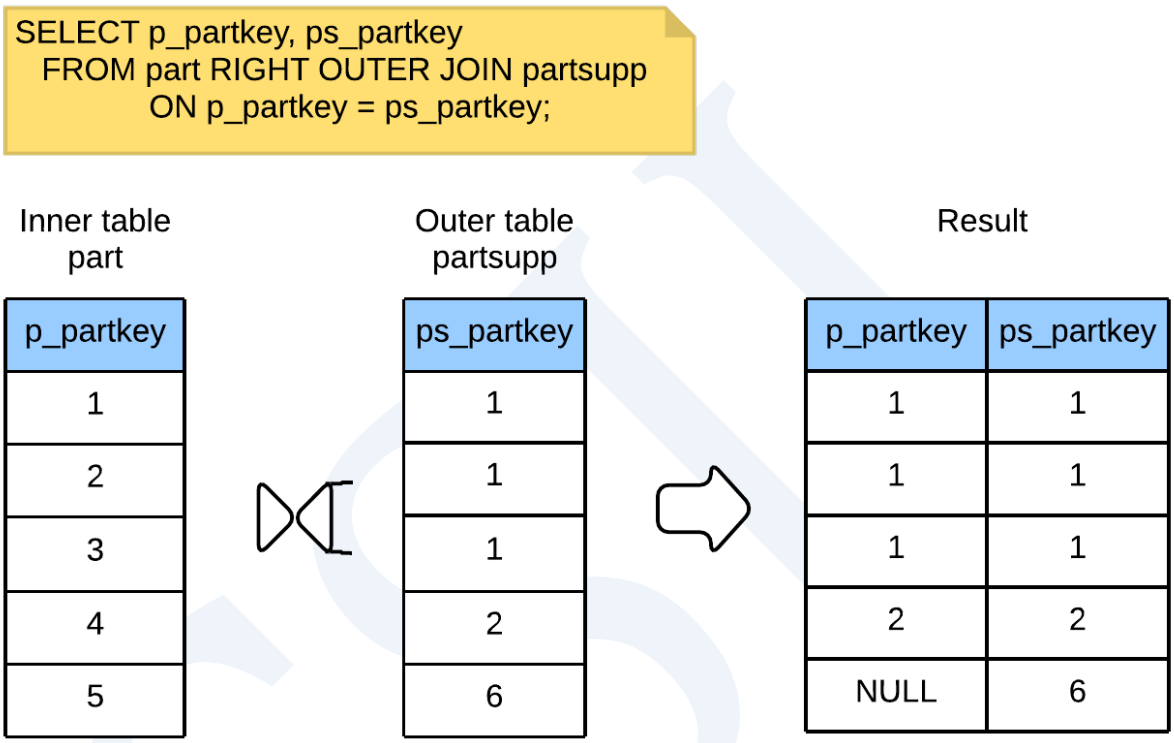


Figure 5-20 Right outer join

Full outer join在输出满足join condition的row后执行一次left outer和一次right outer后输出所有row

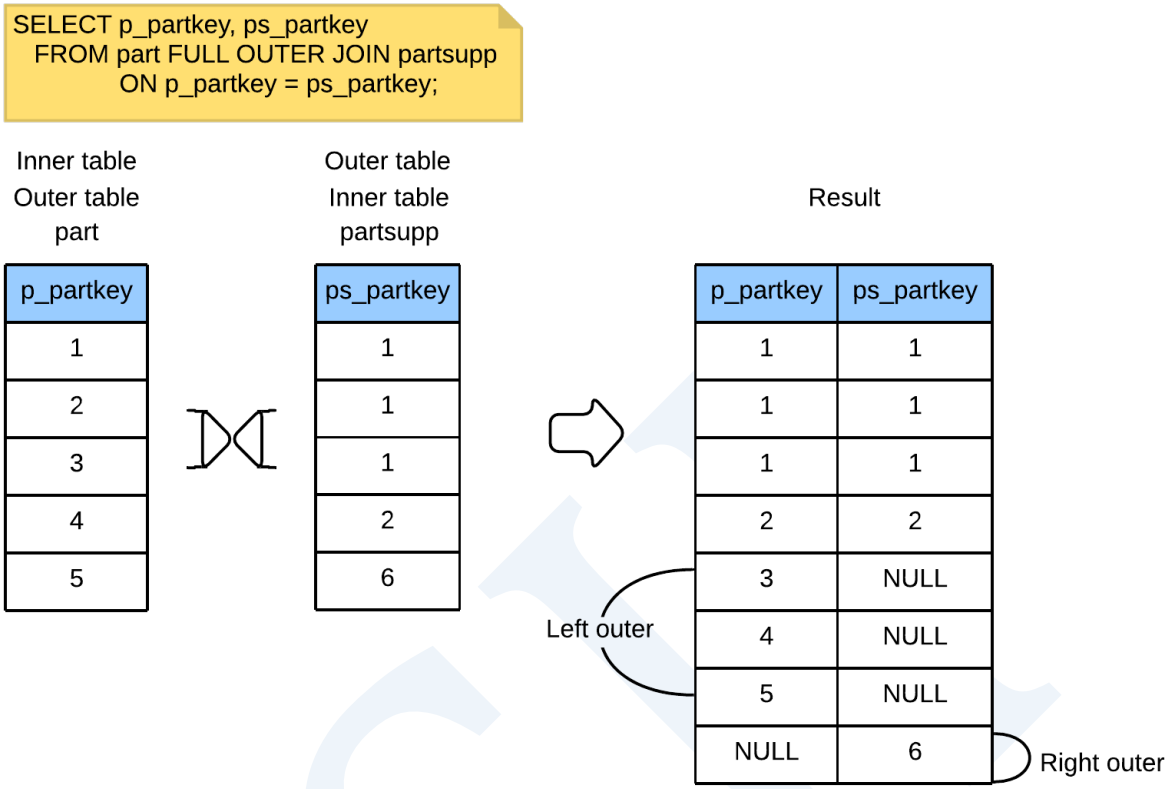


Figure 5-21 Full outer join

Left Outer Join

作为结果返回满足join condition的所有row即使不满足join conditionleft table的row也返回为结果

以下为left outer join的示例

```
gSQL> \EXPLAIN PLAN
SELECT r_name, n_name
  FROM region
       LEFT OUTER JOIN
       nation
```

```
       ON  r_regionkey = n_regionkey
       AND n_nationkey > 20;


R_NAME                    N_NAME
------------------------- -------------------------
AFRICA                    null
AMERICA                   UNITED STATES
ASIA                      VIETNAM
EUROPE                    UNITED KINGDOM
EUROPE                    RUSSIA
MIDDLE EAST               null


6 rows selected.


>>>  start print plan


< Execution Plan >
=============================================================================
|  IDX  |  NODE DESCRIPTION                                                 |
-----------------------------------------------------------------------------
|    0  |  SELECT STATEMENT                                                 |
|    1  |    QUERY BLOCK ("$QB_IDX_2")                                      |
|    2  |      HASH JOIN (LEFT OUTER JOIN)                                  |
|    3  |        TABLE ACCESS ("REGION")                                    |
|    4  |        HASH JOIN INSTANT                                          |
```

```
|    5  |          INDEX ACCESS ("NATION", "NATION_PK_INDEX")          |
=========================================================================


     1  -  TARGET : REGION.R_NAME, NATION.N_NAME
     2  -  JOINED COLUMN : REGION.R_NAME, NATION.N_NAME
     3  -  READ COLUMN : REGION.R_REGIONKEY, REGION.R_NAME
     4  -  HASH KEY : NATION.N_REGIONKEY
           RECORD COLUMN : NATION.N_NAME
           READ KEY COLUMN : NATION.N_REGIONKEY, NATION.N_NAME
             HASH FILTER : NATION.N_REGIONKEY = REGION.R_REGIONKEY
     5  -  READ INDEX COLUMN : NATION.N_NATIONKEY
           READ TABLE COLUMN : NATION.N_NAME, NATION.N_REGIONKEY
             MIN RANGE : NATION.N_NATIONKEY > 20
             MAX RANGE : NATION.N_NATIONKEY IS NOT NULL


<<<  end print plan
```

Right Outer Join

作为结果返回满足join condition的所有row即使不满足join conditionright table的row也返回为结果

以下为right outer join的示例

```
gSQL> \EXPLAIN PLAN
SELECT r_name, n_name
```

```
  FROM region RIGHT OUTER JOIN nation ON  r_regionkey = n_regionkey
                                      AND n_nationkey > 20;
R_NAME N_NAME
------ -------------------------
null                      ALGERIA
null                      ARGENTINA
null                      BRAZIL
null                      CANADA
null                      EGYPT
null                      ETHIOPIA
null                      FRANCE
null                      GERMANY
null                      INDIA
null                      INDONESIA
null                      IRAN
null                      IRAQ
null                      JAPAN
null                      JORDAN
null                      KENYA
null                      MOROCCO
null                      MOZAMBIQUE
null                      PERU
null                      CHINA
null                      ROMANIA
null                      SAUDI ARABIA
```

```
ASIA                      VIETNAM

EUROPE                    RUSSIA

EUROPE                    UNITED KINGDOM

AMERICA                   UNITED STATES


25 rows selected.


>>>  start print plan


< Execution Plan >

=============================================================================

|  IDX  |  NODE DESCRIPTION                                                  |

-----------------------------------------------------------------------------

|    0  |  SELECT STATEMENT                                                  |

|    1  |    QUERY BLOCK ("$QB_IDX_2")                                       |

|    2  |      HASH JOIN (LEFT OUTER JOIN)                                   |

|    3  |        TABLE ACCESS ("NATION")                                     |

|    4  |        HASH JOIN INSTANT                                           |

|    5  |          TABLE ACCESS ("REGION")                                   |

=============================================================================


     1  -  TARGET : REGION.R_NAME, NATION.N_NAME

     2  -  JOINED COLUMN : REGION.R_NAME, NATION.N_NAME

     3  -  READ COLUMN : NATION.N_NATIONKEY, NATION.N_NAME,

NATION.N_REGIONKEY
```

```
    4  -  HASH KEY : REGION.R_REGIONKEY
          RECORD COLUMN : REGION.R_NAME
          READ KEY COLUMN : REGION.R_REGIONKEY, REGION.R_NAME
            HASH FILTER : REGION.R_REGIONKEY = NATION.N_REGIONKEY
            LOGICAL FILTER : {NATION.N_NATIONKEY} > 20
          FETCH ONE ROW
    5  -  READ COLUMN : REGION.R_REGIONKEY, REGION.R_NAME


<<<  end print plan
```

Full Outer Join

作为结果返回满足join condition的所有row即使不满足join condition的left table的row和right table的row也均返回为结果

以下为full outer join的示例

```
gSQL> \EXPLAIN PLAN
SELECT r_name, n_name
  FROM region
       FULL OUTER JOIN
       nation
       ON  r_regionkey = n_regionkey
       AND n_nationkey > 20;


R_NAME N_NAME
```

```
------ -------------------------
null                        ALGERIA
null                        ARGENTINA
null                        BRAZIL
null                        CANADA
null                        EGYPT
null                        ETHIOPIA
null                        FRANCE
null                        GERMANY
null                        INDIA
null                        INDONESIA
null                        IRAN
null                        IRAQ
null                        JAPAN
null                        JORDAN
null                        KENYA
null                        MOROCCO
null                        MOZAMBIQUE
null                        PERU
null                        CHINA
null                        ROMANIA
null                        SAUDI ARABIA
ASIA                        VIETNAM
EUROPE                      RUSSIA
EUROPE                      UNITED KINGDOM
```

```
AMERICA                   UNITED STATES

AFRICA                    null

MIDDLE EAST               null


27 rows selected.


>>>  start print plan


< Execution Plan >

=============================================================================

|  IDX  |  NODE DESCRIPTION                                                  |

-----------------------------------------------------------------------------

|    0  |  SELECT STATEMENT                                                  |

|    1  |    QUERY BLOCK ("$QB_IDX_2")                                       |

|    2  |      HASH JOIN (FULL OUTER JOIN)                                   |

|    3  |        TABLE ACCESS ("NATION")                                     |

|    4  |        HASH JOIN INSTANT                                           |

|    5  |          TABLE ACCESS ("REGION")                                   |

=============================================================================


     1  -  TARGET : REGION.R_NAME, NATION.N_NAME

     2  -  JOINED COLUMN : REGION.R_NAME, NATION.N_NAME

     3  -  READ COLUMN : NATION.N_NATIONKEY, NATION.N_NAME,

NATION.N_REGIONKEY

     4  -  HASH KEY : REGION.R_REGIONKEY
```

```
            RECORD COLUMN : REGION.R_NAME
            READ KEY COLUMN : REGION.R_REGIONKEY, REGION.R_NAME
              HASH FILTER : REGION.R_REGIONKEY = NATION.N_REGIONKEY
              LOGICAL FILTER : {NATION.N_NATIONKEY} > 20
      5   -   READ COLUMN : REGION.R_REGIONKEY, REGION.R_NAME


<<<  end print plan
```

Semi Join

Semi join无法使用SQL语句直接描述用户与INEXISTS=ANY等quantifier同时编写子查询时rewriter在subquery unnesting的过程中转换为semi join

存在满足join condition的row时作为结果返回main query的row

以下为semi join的示例

```
\EXPLAIN PLAN
SELECT o_orderpriority,
       count(*) as order_count
  FROM orders
 WHERE o_orderdate = date '1993-07-01'
   AND EXISTS (
                SELECT *
                  FROM lineitem
                 WHERE l_orderkey = o_orderkey
                   AND l_commitdate < l_receiptdate
```

```
             )
GROUP BY o_orderpriority
ORDER BY o_orderpriority;


O_ORDERPRIORITY ORDER_COUNT
--------------- -----------
1-URGENT                113
2-HIGH                  136
3-MEDIUM                112
4-NOT SPECIFIED         103
5-LOW                    97

5 rows selected.

>>>  start print plan

< Execution Plan >
================================================================================
|  IDX  |  NODE DESCRIPTION                                                    |
--------------------------------------------------------------------------------
|    0  |  SELECT STATEMENT                                                    |
|    1  |    QUERY BLOCK ("$QB_IDX_2")                                         |
|    2  |      SORT INSTANT                                                    |
|    3  |        GROUP HASH INSTANT                                            |
|    4  |          NESTED JOIN (SEMI)                                          |
```

```
|    5  |            TABLE ACCESS ("ORDERS")                              |
|    6  |            INDEX ACCESS ("LINEITEM", "LINEITEM_ORDERKEY_FK")    |
=========================================================================


     1  -  TARGET : ORDERS.O_ORDERPRIORITY, COUNT(*) AS ORDER_COUNT
     2  -  SORT KEY : "ORDERS.O_ORDERPRIORITY ASC NULLS LAST"
           RECORD COLUMN : COUNT(*)
           READ KEY COLUMN : ORDERS.O_ORDERPRIORITY
           READ RECORD COLUMN : COUNT(*)
     3  -  GROUP KEY : ORDERS.O_ORDERPRIORITY
           RECORD COLUMN : COUNT(*)
           READ KEY COLUMN : ORDERS.O_ORDERPRIORITY
           READ RECORD COLUMN : COUNT(*)
     4  -  JOINED COLUMN : ORDERS.O_ORDERPRIORITY
     5  -  READ COLUMN : ORDERS.O_ORDERKEY, ORDERS.O_ORDERDATE,
ORDERS.O_ORDERPRIORITY
             PHYSICAL FILTER : ORDERS.O_ORDERDATE = DATE'1993-07-01'
     6  -  READ INDEX COLUMN : LINEITEM.L_ORDERKEY
           READ TABLE COLUMN : LINEITEM.L_COMMITDATE,
LINEITEM.L_RECEIPTDATE
             MIN RANGE : LINEITEM.L_ORDERKEY = {ORDERS.O_ORDERKEY}
             MAX RANGE : LINEITEM.L_ORDERKEY = {ORDERS.O_ORDERKEY}
             PHYSICAL TABLE FILTER :
                         LINEITEM.L_RECEIPTDATE > LINEITEM.L_COMMITDATE
```

```
<<<  end print plan
```

Anti Semi Join

Anti semi join无法使用SQL语句直接描述用户与NOT INNOT EXISTS!=ALL=ALL等quantifier同时编写子查询时rewriter在subquery unnesting的过程中转换为anti semi join

没有任何满足join condition的row时作为结果返回main query的row

Join condition中有nullable column时执行为null-aware anti-semi join否则执行为anti-semi join

以下为anti semi join的示例

p_partkey和ps_partkey均为primary key column因此均为not null column

```
\EXPLAIN PLAN
SELECT p_name, p_brand
  FROM part
 WHERE p_partkey NOT IN ( SELECT ps_partkey
                            FROM partsupp
                           WHERE ps_availqty > 5000 );
...


12511 rows selected.


>>>  start print plan


< Execution Plan >
```

```
=============================================================================
|  IDX   |  NODE DESCRIPTION                                                |
-----------------------------------------------------------------------------
|    0   |  SELECT STATEMENT                                                |
|    1   |    QUERY BLOCK ("$QB_IDX_2")                                     |
|    2   |      HASH JOIN (ANTI SEMI)                                       |
|    3   |        TABLE ACCESS ("PART")                                     |
|    4   |        HASH JOIN INSTANT (UNIQUE)                                |
|    5   |          TABLE ACCESS ("PARTSUPP")                               |
=============================================================================


     1  -  TARGET : PART.P_NAME, PART.P_BRAND
     2  -  JOINED COLUMN : PART.P_NAME, PART.P_BRAND
     3  -  READ COLUMN : PART.P_PARTKEY, PART.P_NAME, PART.P_BRAND
     4  -  HASH KEY : PARTSUPP.PS_PARTKEY
           READ KEY COLUMN : PARTSUPP.PS_PARTKEY
             HASH FILTER : PARTSUPP.PS_PARTKEY = PART.P_PARTKEY
           FETCH ONE ROW
     5  -  READ COLUMN : PARTSUPP.PS_PARTKEY, PARTSUPP.PS_AVAILQTY
             PHYSICAL FILTER : PARTSUPP.PS_AVAILQTY > 5000


<<<  end print plan
```

以下为null-aware anti-semi join的示例

p_partkey是primary key column因此是not null但l_partkey是nullable

```
\EXPLAIN PLAN

SELECT p_name, p_brand

  FROM part

 WHERE p_partkey NOT IN ( SELECT l_partkey

                            FROM lineitem

                           WHERE l_quantity > 30 );


no rows selected.


>>>  start print plan


< Execution Plan >

=============================================================================

|  IDX  |  NODE DESCRIPTION                                                  |

-----------------------------------------------------------------------------

|    0  |  SELECT STATEMENT                                                  |

|    1  |    QUERY BLOCK ("$QB_IDX_2")                                       |

|    2  |      HASH JOIN (ANTI SEMI NA)                                      |

|    3  |        TABLE ACCESS ("PART")                                       |

|    4  |        HASH JOIN INSTANT (UNIQUE)                                  |

|    5  |          TABLE ACCESS ("LINEITEM")                                 |

=============================================================================


     1  -  TARGET : PART.P_NAME, PART.P_BRAND

     2  -  JOINED COLUMN : PART.P_NAME, PART.P_BRAND
```

```
    3  -  READ COLUMN : PART.P_PARTKEY, PART.P_NAME, PART.P_BRAND
    4  -  HASH KEY : LINEITEM.L_PARTKEY
          READ KEY COLUMN : LINEITEM.L_PARTKEY
            HASH FILTER : LINEITEM.L_PARTKEY = PART.P_PARTKEY
          FETCH ONE ROW
    5  -  READ COLUMN : LINEITEM.L_PARTKEY, LINEITEM.L_QUANTITY
            PHYSICAL FILTER : LINEITEM.L_QUANTITY > 30

<<<  end print plan
```

Join Method

两张表的join运算方法有nested loops joinsort merge joinhash joinEnumerator计算此join method的cost选择成本最低的join运算

Nested Loops Join

对outer table的各个row检索inner table的所有row后查找满足join condition的结果

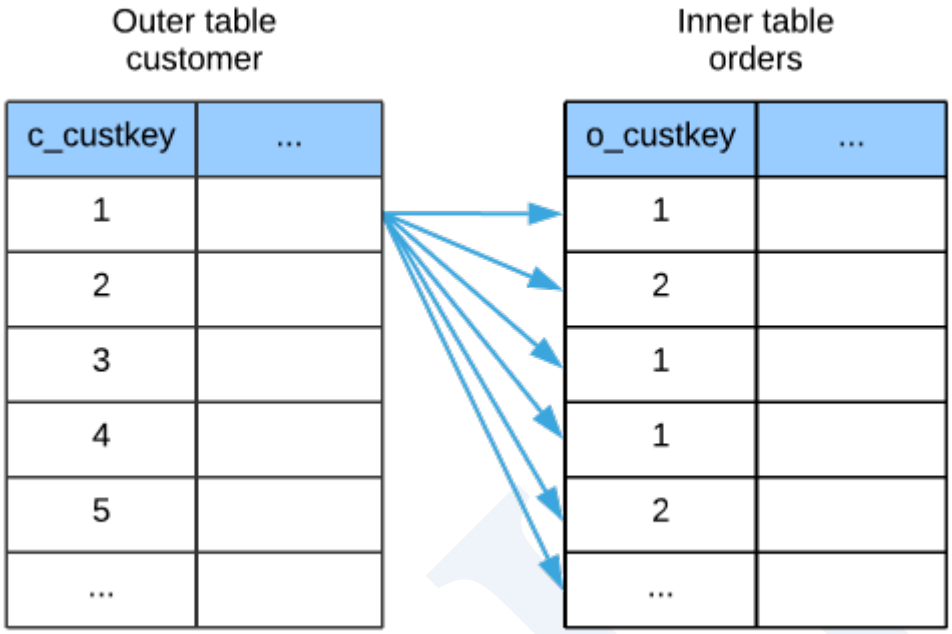


Figure 5-22 Nested loop join

按照outer table的row数量对inner table进行full scan因此outer table的row数量越少越好

没有join condition的join也可通过nested loop join以cartesian product返回执行结果因此无法进行hash joinsort merge join时也可执行nested loop join

Index Nested Loops Join

Inner table中有index因此可通过其index查找符合join条件的row时执行index nested loop joinIndex access仅访问所需的row因此可提高性能

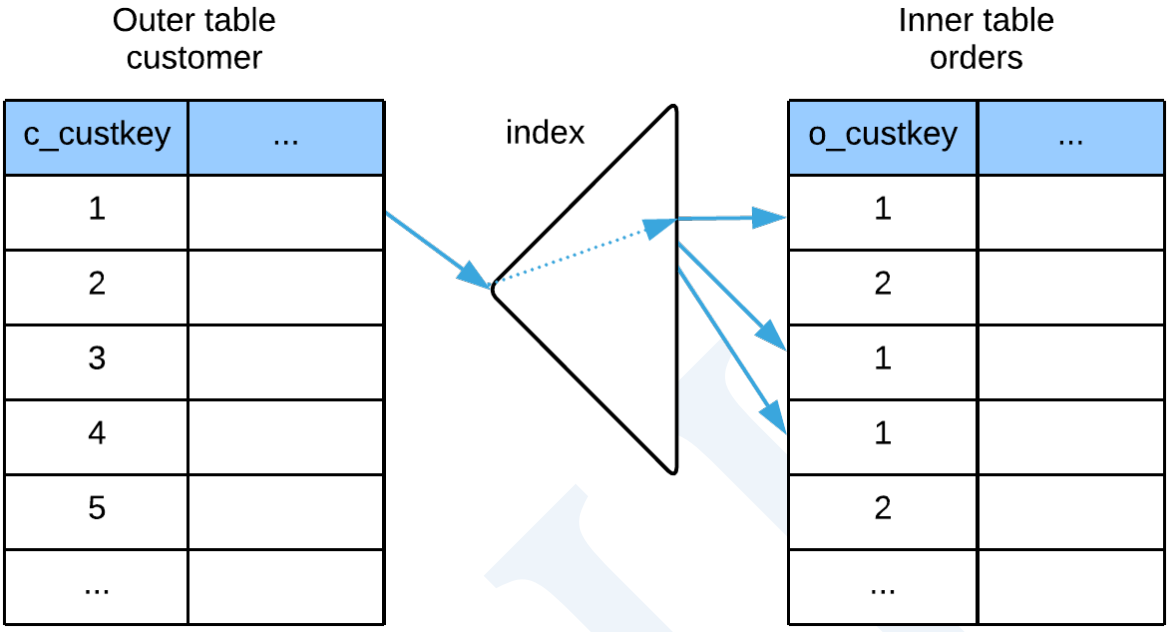


Figure 5-23 Index nested loop join

以下为index nested loop join的示例

```
\EXPLAIN PLAN
  SELECT c_custkey, count(o_orderkey)
    FROM customer, orders
   WHERE c_custkey = o_custkey
     AND c_comment like '%special%requests%'
GROUP BY c_custkey;
...
2265 rows selected.


>>>  start print plan
```

```
< Execution Plan >

=============================================================================
|  IDX  |  NODE DESCRIPTION                                                  |
-----------------------------------------------------------------------------
|    0  |  SELECT STATEMENT                                                  |
|    1  |    QUERY BLOCK ("$QB_IDX_2")                                       |
|    2  |      GROUP HASH INSTANT                                            |
|    3  |        NESTED JOIN (INNER JOIN)                                    |
|    4  |          TABLE ACCESS ("CUSTOMER")                                 |
|    5  |          INDEX ACCESS ("ORDERS", "ORDERS_CUSTKEY_FK")              |
=============================================================================


    1  -  TARGET : CUSTOMER.C_CUSTKEY, COUNT( ORDERS.O_ORDERKEY )
    2  -  GROUP KEY : CUSTOMER.C_CUSTKEY
          RECORD COLUMN : COUNT( ORDERS.O_ORDERKEY )
          READ KEY COLUMN : CUSTOMER.C_CUSTKEY
          READ RECORD COLUMN : COUNT( ORDERS.O_ORDERKEY )
    3  -  JOINED COLUMN : CUSTOMER.C_CUSTKEY, ORDERS.O_ORDERKEY
    4  -  READ COLUMN : CUSTOMER.C_CUSTKEY, CUSTOMER.C_COMMENT
            LOGICAL FILTER : CUSTOMER.C_COMMENT LIKE '%special%requests%'
    5  -  READ INDEX COLUMN : ORDERS.O_CUSTKEY
          READ TABLE COLUMN : ORDERS.O_ORDERKEY
            MIN RANGE : ORDERS.O_CUSTKEY = {CUSTOMER.C_CUSTKEY}
            MAX RANGE : ORDERS.O_CUSTKEY = {CUSTOMER.C_CUSTKEY}
```

```
<<<  end print plan
```

Instant Nested Loops Join

在instant table加载inner table的中间结果后执行nested loop join

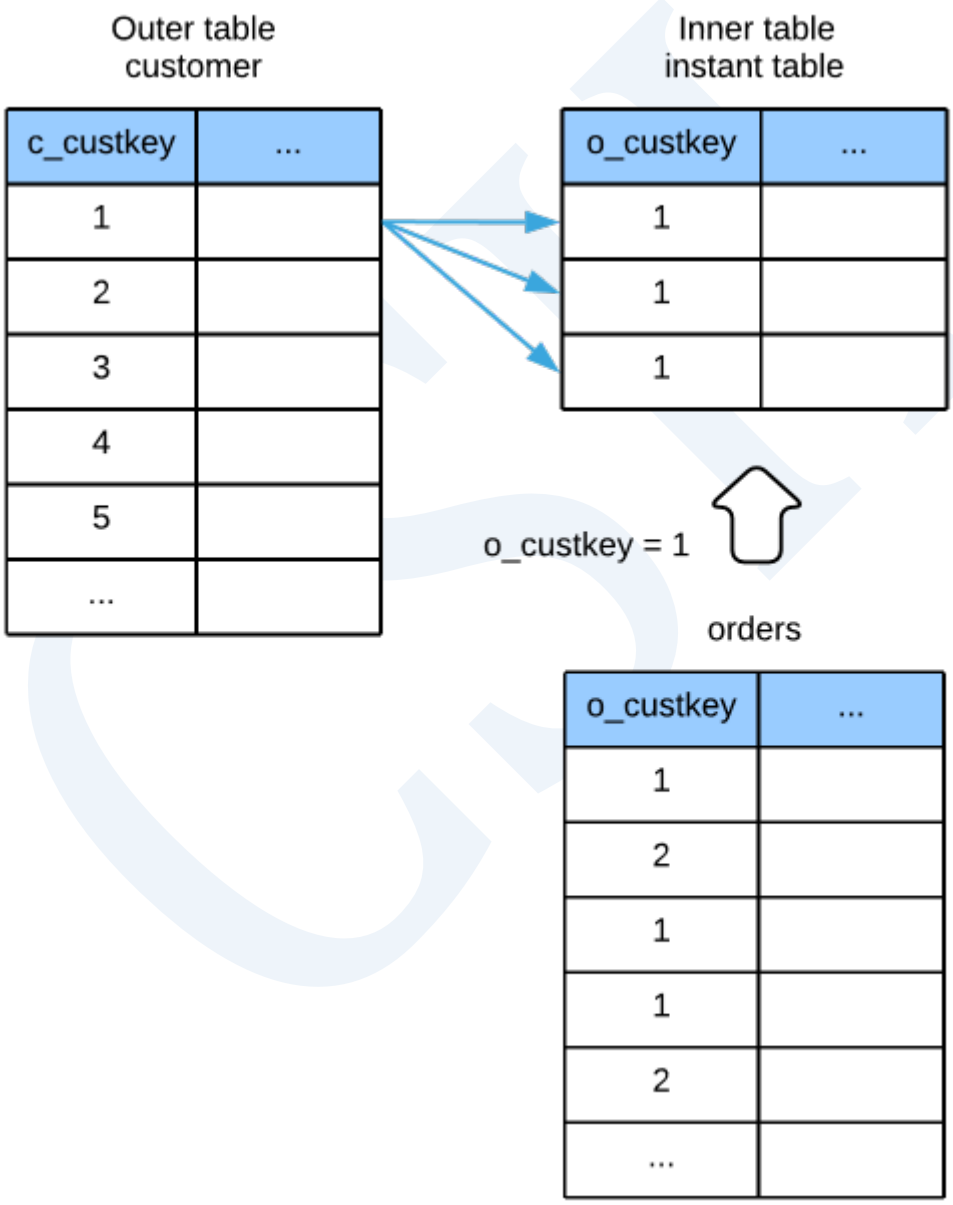


Figure 5-24 Instant nested loop join

如下示例有o_custkey = 1等条件时在instant table加载orders的中间结果并执行nested loop join

以下为instant nested loop join的示例

```
\EXPLAIN PLAN
  SELECT c_custkey, count(o_orderkey)
    FROM customer, orders
   WHERE c_comment like '%special%requests%'
     AND o_orderdate = date '1995-03-15'
GROUP BY c_custkey;


...
3380 rows selected.


>>>  start print plan


< Execution Plan >
================================================================================
|  IDX  |  NODE DESCRIPTION                                                     |
--------------------------------------------------------------------------------
|    0  |  SELECT STATEMENT                                                     |
|    1  |    QUERY BLOCK ("$QB_IDX_2")                                          |
|    2  |      GROUP HASH INSTANT                                               |
|    3  |        NESTED JOIN (INNER JOIN)                                      |
|    4  |          TABLE ACCESS ("ORDERS")                                      |
|    5  |          FLAT JOIN INSTANT                                            |
```

```
|    6  |           TABLE ACCESS ("CUSTOMER")                          |
=======================================================================


     1  -  TARGET : CUSTOMER.C_CUSTKEY, COUNT( ORDERS.O_ORDERKEY )
     2  -  GROUP KEY : CUSTOMER.C_CUSTKEY
           RECORD COLUMN : COUNT( ORDERS.O_ORDERKEY )
           READ KEY COLUMN : CUSTOMER.C_CUSTKEY
           READ RECORD COLUMN : COUNT( ORDERS.O_ORDERKEY )
     3  -  JOINED COLUMN : CUSTOMER.C_CUSTKEY, ORDERS.O_ORDERKEY
     4  -  READ COLUMN : ORDERS.O_ORDERKEY, ORDERS.O_ORDERDATE
             PHYSICAL FILTER : ORDERS.O_ORDERDATE = DATE'1995-03-15'
     5  -  RECORD COLUMN : CUSTOMER.C_CUSTKEY
           READ COLUMN : CUSTOMER.C_CUSTKEY
     6  -  READ COLUMN : CUSTOMER.C_CUSTKEY, CUSTOMER.C_COMMENT
             LOGICAL FILTER : CUSTOMER.C_COMMENT LIKE '%special%requests%'


<<<  end print plan
```

Sort Merge Join

排列outer table和inner table的中间结果后依次对比并检查是否满足join condition后返回join结果

Outer table或inner table中存在index并可使用时该table不使用sort instant通过index获取排列的中间结果

进行sort merge join需要有一个以上的equi join condition

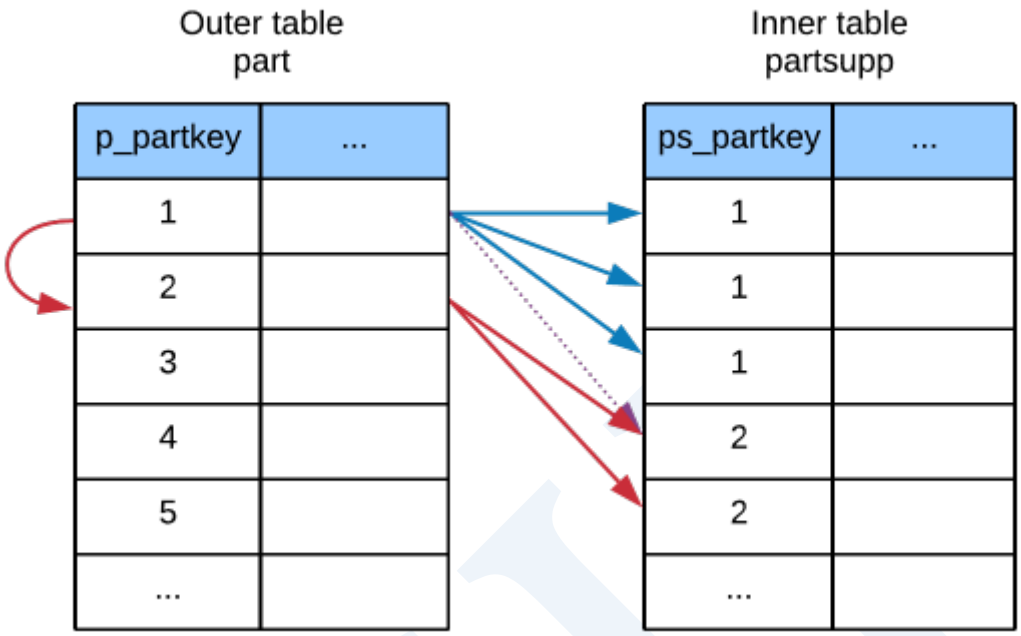


Figure 5-25 Sort merge join

以下为sort merge join的示例

```
\EXPLAIN PLAN

SELECT p_name, p_brand, p_type

  FROM part, partsupp

 WHERE p_partkey = ps_partkey

   AND p_partkey < 10;

...

36 rows selected.


>>>  start print plan


< Execution Plan >
```

```
=============================================================================
|  IDX   |  NODE DESCRIPTION                                                |
-----------------------------------------------------------------------------
|    0   |  SELECT STATEMENT                                                |
|    1   |    QUERY BLOCK ("$QB_IDX_2")                                     |
|    2   |      MERGE JOIN (INNER JOIN)                                     |
|    3   |        INDEX ACCESS ("PART", "PART_PK_INDEX")                    |
|    4   |        INDEX ACCESS ("PARTSUPP", "PARTSUPP_PARTKEY_FK")          |
=============================================================================


     1  -  TARGET : PART.P_NAME, PART.P_BRAND, PART.P_TYPE
     2  -  JOINED COLUMN : PART.P_NAME, PART.P_BRAND, PART.P_TYPE
             ON FILTER (Equi) : PART.P_PARTKEY = PARTSUPP.PS_PARTKEY
     3  -  READ INDEX COLUMN : PART.P_PARTKEY
           READ TABLE COLUMN : PART.P_NAME, PART.P_BRAND, PART.P_TYPE
             MAX RANGE : PART.P_PARTKEY < 10
     4  -  READ INDEX COLUMN : PARTSUPP.PS_PARTKEY
             MAX RANGE : PARTSUPP.PS_PARTKEY < 10


<<<  end print plan
```

Hash Join

在inner table生成hash instant后使用hash返回符合join condition的join结果

进行hash join需要有一个以上的equi-join condition

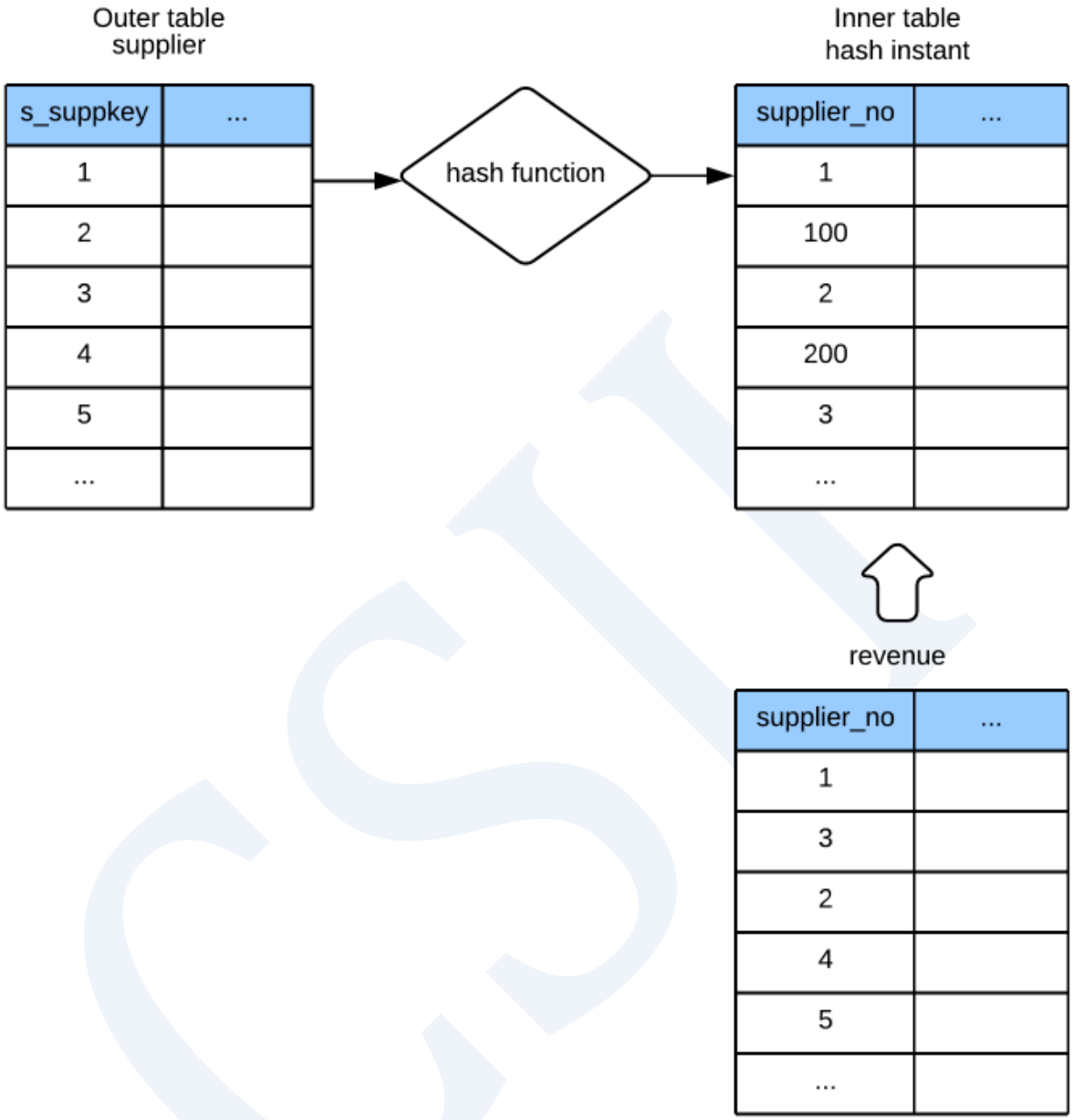


Figure 5-26 Hash join

以下为hash join的示例

```
\EXPLAIN PLAN

SELECT s_suppkey,

       s_name,
```

```
       total_revenue
  FROM supplier,
         (
            SELECT l_suppkey,
                   ROUND( sum(l_extendedprice * (1 - l_discount)), 2)
              FROM lineitem
             WHERE l_shipdate >= date '1996-01-01'
               AND l_shipdate < date '1996-01-01' + interval '3' month
             GROUP BY l_suppkey
       ) revenue(supplier_no, total_revenue)
 WHERE
      s_suppkey = supplier_no;



...
10000 rows selected.


>>>  start print plan


< Execution Plan >
=========================================================================
|  IDX  |  NODE DESCRIPTION                                             |
-------------------------------------------------------------------------
|    0  |  SELECT STATEMENT                                             |
|    1  |    QUERY BLOCK ("$QB_IDX_2")                                  |
```

```
|    2  |      HASH JOIN (INNER JOIN)                                    |
|    3  |        TABLE ACCESS ("SUPPLIER")                               |
|    4  |        HASH JOIN INSTANT                                       |
|    5  |          INLINE_VIEW ("REVENUE")                               |
|    6  |            QUERY BLOCK ("$QB_IDX_7")                           |
|    7  |              GROUP HASH INSTANT                                |
|    8  |                TABLE ACCESS ("LINEITEM")                       |
=====================================================================


     1  -  TARGET : SUPPLIER.S_SUPPKEY, SUPPLIER.S_NAME, REVENUE.$C1
     2  -  JOINED COLUMN : SUPPLIER.S_SUPPKEY, SUPPLIER.S_NAME,
REVENUE.$C1
     3  -  READ COLUMN : SUPPLIER.S_SUPPKEY, SUPPLIER.S_NAME
     4  -  HASH KEY : REVENUE.L_SUPPKEY
           RECORD COLUMN : REVENUE.$C1
           READ KEY COLUMN : REVENUE.L_SUPPKEY, REVENUE.$C1
             HASH FILTER : REVENUE.L_SUPPKEY = SUPPLIER.S_SUPPKEY
     5  -  COLUMN : LINEITEM.L_SUPPKEY AS L_SUPPKEY,
ROUND(SUM( LINEITEM.L_EXTENDEDPRICE * ( 1 - LINEITEM.L_DISCOUNT ) ),2) AS
$C1
     6  -  TARGET : LINEITEM.L_SUPPKEY,
ROUND(SUM( LINEITEM.L_EXTENDEDPRICE * ( 1 - LINEITEM.L_DISCOUNT ) ),2)
     7  -  GROUP KEY : LINEITEM.L_SUPPKEY
           RECORD COLUMN : SUM( LINEITEM.L_EXTENDEDPRICE * ( 1 -
LINEITEM.L_DISCOUNT ) )
```

```
            READ KEY COLUMN : LINEITEM.L_SUPPKEY

            READ RECORD COLUMN : SUM( LINEITEM.L_EXTENDEDPRICE * ( 1 -
LINEITEM.L_DISCOUNT ) )

     8   -   READ COLUMN : LINEITEM.L_SUPPKEY, LINEITEM.L_EXTENDEDPRICE,
LINEITEM.L_DISCOUNT, LINEITEM.L_SHIPDATE

              PHYSICAL FILTER : LINEITEM.L_SHIPDATE < DATE'1996-01-01' +
CAST( '3' AS INTERVAL(MONTH) ) AND LINEITEM.L_SHIPDATE >= DATE'1996-01-01'


<<<  end print plan
```

Join Order

Join三个以上的表时需要决定join顺序通过先join两张表后再通过join其中间结果和下一个表的方式决定join order

如果有三张表时如下有多种join order

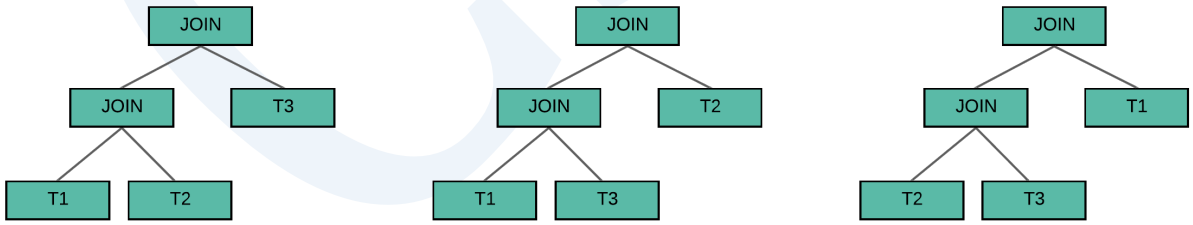


Figure 5-27 Join order

Enumerator根据可行的join orderjoin method以及可行的access path生成多种执行计划的集合首先选择减少中间结果cost少的plan并决定join ordering

Group By

处理group by的过程

通常生成GROUP HASH INSTANT后处理group by

下级中间结果对group by key column排列而上升时可以不积累单独的hash instant而进行处理

以下为生成GROUP HASH INSTANT后处理group by的示例

```
\EXPLAIN PLAN
  SELECT c_custkey, count(o_orderkey)
    FROM customer, orders
 WHERE c_custkey = o_custkey
     AND c_comment like '%special%requests%'
     AND o_orderdate = date '1995-03-15'
GROUP BY c_custkey;


...
12 rows selected.


>>>  start print plan


< Execution Plan >
=============================================================================
|  IDX  |  NODE DESCRIPTION                                                  |
```

```
-------------------------------------------------------------------------
|    0  |  SELECT STATEMENT                                             |
|    1  |    QUERY BLOCK ("$QB_IDX_2")                                  |
|    2  |      GROUP HASH INSTANT                                        |
|    3  |        NESTED JOIN (INNER JOIN)                               |
|    4  |          TABLE ACCESS ("CUSTOMER")                            |
|    5  |          INDEX ACCESS ("ORDERS", "ORDERS_CUSTKEY_FK")         |
=========================================================================


     1  -  TARGET : CUSTOMER.C_CUSTKEY, COUNT( ORDERS.O_ORDERKEY )
     2  -  GROUP KEY : CUSTOMER.C_CUSTKEY
           RECORD COLUMN : COUNT( ORDERS.O_ORDERKEY )
           READ KEY COLUMN : CUSTOMER.C_CUSTKEY
           READ RECORD COLUMN : COUNT( ORDERS.O_ORDERKEY )
     3  -  JOINED COLUMN : CUSTOMER.C_CUSTKEY, ORDERS.O_ORDERKEY
     4  -  READ COLUMN : CUSTOMER.C_CUSTKEY, CUSTOMER.C_COMMENT
             LOGICAL FILTER : CUSTOMER.C_COMMENT LIKE '%special%requests%'
     5  -  READ INDEX COLUMN : ORDERS.O_CUSTKEY
           READ TABLE COLUMN : ORDERS.O_ORDERKEY, ORDERS.O_ORDERDATE
             MIN RANGE : ORDERS.O_CUSTKEY = {CUSTOMER.C_CUSTKEY}
             MAX RANGE : ORDERS.O_CUSTKEY = {CUSTOMER.C_CUSTKEY}
             PHYSICAL TABLE FILTER : ORDERS.O_ORDERDATE = DATE'1995-03-15'


<<<  end print plan
```

以下为使用下级的排列的中间结果后处理group by的示例

```
\EXPLAIN PLAN
  SELECT c_custkey, count(o_orderkey)
    FROM customer, orders
   WHERE c_custkey = o_custkey
     AND c_custkey >= 10
     AND c_custkey < 20
     AND c_comment like '%special%requests%'
     AND o_orderdate = date '1995-03-15'
GROUP BY c_custkey;

...

>>>  start print plan

< Execution Plan >
=========================================================================
|  IDX  |  NODE DESCRIPTION                                             |
-------------------------------------------------------------------------
|    0  |  SELECT STATEMENT                                             |
|    1  |    QUERY BLOCK ("$QB_IDX_2")                                  |
|    2  |      GROUP                                                     |
|    3  |        NESTED JOIN (INNER JOIN)                               |
|    4  |          INDEX ACCESS ("CUSTOMER", "CUSTOMER_PK_INDEX")       |
```

```
|    5  |          INDEX ACCESS ("ORDERS", "ORDERS_CUSTKEY_FK")        |
=======================================================================


     1  -  TARGET : CUSTOMER.C_CUSTKEY, COUNT( ORDERS.O_ORDERKEY )
     2  -  GROUP KEY : CUSTOMER.C_CUSTKEY
           RECORD COLUMN : COUNT( ORDERS.O_ORDERKEY )
     3  -  JOINED COLUMN : CUSTOMER.C_CUSTKEY, ORDERS.O_ORDERKEY
     4  -  READ INDEX COLUMN : CUSTOMER.C_CUSTKEY
           READ TABLE COLUMN : CUSTOMER.C_COMMENT
             MIN RANGE : CUSTOMER.C_CUSTKEY >= 10
             MAX RANGE : CUSTOMER.C_CUSTKEY IS NOT NULL AND
CUSTOMER.C_CUSTKEY < 20
             LOGICAL TABLE FILTER : CUSTOMER.C_COMMENT LIKE
'%special%requests%'
     5  -  READ INDEX COLUMN : ORDERS.O_CUSTKEY
           READ TABLE COLUMN : ORDERS.O_ORDERKEY, ORDERS.O_ORDERDATE
             MIN RANGE : ORDERS.O_CUSTKEY = {CUSTOMER.C_CUSTKEY} AND
ORDERS.O_CUSTKEY >= 10
             MAX RANGE : ORDERS.O_CUSTKEY = {CUSTOMER.C_CUSTKEY} AND
ORDERS.O_CUSTKEY < 20
             LOGICAL KEY FILTER : ORDERS.O_CUSTKEY LIKE
'%special%requests%'
             PHYSICAL TABLE FILTER : ORDERS.O_ORDERDATE = DATE'1995-03-15'


<<<  end print plan
```

Distinct

处理distinct的过程

通常生成GROUP HASH INSTANT后处理distinct

下级中间结果对distinct key column排列而上升时可以不积累单独的hash instant而进行处理

以下为生成GROUP HASH INSTANT后处理distinct的示例

```
\EXPLAIN PLAN
SELECT DISTINCT c_nationkey
  FROM customer
 WHERE c_comment like '%special%requests%'
;


...


>>>  start print plan


< Execution Plan >
=========================================================================
|  IDX  |  NODE DESCRIPTION                                             |
-------------------------------------------------------------------------
|    0  |  SELECT STATEMENT                                             |
|    1  |    QUERY BLOCK ("$QB_IDX_2")                                  |
```

```
|    2  |       GROUP HASH INSTANT                                          |
|    3  |         TABLE ACCESS ("CUSTOMER")                                 |
=====================================================================


     1  -  TARGET : CUSTOMER.C_NATIONKEY
     2  -  GROUP KEY : CUSTOMER.C_NATIONKEY
           READ KEY COLUMN : CUSTOMER.C_NATIONKEY
     3  -  READ COLUMN : CUSTOMER.C_NATIONKEY, CUSTOMER.C_COMMENT
             LOGICAL FILTER : CUSTOMER.C_COMMENT LIKE '%special%requests%'


<<<  end print plan
```

以下为使用下级的排列的中间结果处理distinct的示例

```
\EXPLAIN PLAN
  SELECT DISTINCT c_nationkey
    FROM customer
   WHERE c_nationkey >= 15
     AND c_nationkey < 20
     AND c_comment like '%special%requests%'
;
...
>>>  start print plan


< Execution Plan >
=====================================================================
```

```
|  IDX  |  NODE DESCRIPTION                                                    |
-------------------------------------------------------------------------------
|    0  |  SELECT STATEMENT                                                    |
|    1  |    QUERY BLOCK ("$QB_IDX_2")                                         |
|    2  |      GROUP                                                           |
|    3  |        INDEX ACCESS ("CUSTOMER", "CUSTOMER_NATIONKEY_FK")           |
===============================================================================


     1  -  TARGET : CUSTOMER.C_NATIONKEY
     2  -  GROUP KEY : CUSTOMER.C_NATIONKEY
     3  -  READ INDEX COLUMN : CUSTOMER.C_NATIONKEY
           READ TABLE COLUMN : CUSTOMER.C_COMMENT
             MIN RANGE : CUSTOMER.C_NATIONKEY >= 15
             MAX RANGE : CUSTOMER.C_NATIONKEY IS NOT NULL AND
CUSTOMER.C_NATIONKEY < 20
             LOGICAL TABLE FILTER : CUSTOMER.C_COMMENT LIKE
'%special%requests%'


<<<  end print plan
```

Single Row Aggregation

处理single row aggregation的过程

通常使用hash执行aggregation

获取MIN()MAX()的简单的查询也会使用index

以下为使用hash处理single row aggregation的示例

```
\EXPLAIN PLAN
SELECT count(o_orderkey)
  FROM customer, orders
 WHERE c_custkey = o_custkey
   AND c_comment like '%special%requests%';


COUNT(O_ORDERKEY)
-----------------
            33526


1 row selected.


>>>  start print plan


< Execution Plan >
=========================================================================
|  IDX  |  NODE DESCRIPTION                                             |
-------------------------------------------------------------------------
|    0  |  SELECT STATEMENT                                             |
|    1  |    QUERY BLOCK ("$QB_IDX_2")                                  |
|    2  |      AGGREGATION BY HASH                                       |
|    3  |        NESTED JOIN (INNER JOIN)                               |
```

```
|    4  |          TABLE ACCESS ("CUSTOMER")                              |
|    5  |          INDEX ACCESS ("ORDERS", "ORDERS_CUSTKEY_FK")           |
=====================================================================


     1  -  TARGET : COUNT( ORDERS.O_ORDERKEY )
     2  -  AGGREGATION : COUNT( ORDERS.O_ORDERKEY )
     3  -  JOINED COLUMN : ORDERS.O_ORDERKEY
     4  -  READ COLUMN : CUSTOMER.C_CUSTKEY, CUSTOMER.C_COMMENT
             LOGICAL FILTER : CUSTOMER.C_COMMENT LIKE '%special%requests%'
     5  -  READ INDEX COLUMN : ORDERS.O_CUSTKEY
           READ TABLE COLUMN : ORDERS.O_ORDERKEY
             MIN RANGE : ORDERS.O_CUSTKEY = {CUSTOMER.C_CUSTKEY}
             MAX RANGE : ORDERS.O_CUSTKEY = {CUSTOMER.C_CUSTKEY}


<<<  end print plan
```

以下为使用index处理single row aggregation的示例

```
\EXPLAIN PLAN SELECT MIN(c_custkey), MAX(c_custkey) FROM customer;


MIN(C_CUSTKEY) MAX(C_CUSTKEY)
-------------- --------------
             1         150000


1 row selected.
```

```
>>>  start print plan


< Execution Plan >

=============================================================================

|  IDX  |  NODE DESCRIPTION                                                  |

-----------------------------------------------------------------------------

|    0  |  SELECT STATEMENT                                                  |

|    1  |    QUERY BLOCK ("$QB_IDX_2")                                       |

|    2  |      INDEX ACCESS ("CUSTOMER", "CUSTOMER_PK_INDEX")                |

=============================================================================


     1  -  TARGET : MIN( CUSTOMER.C_CUSTKEY ), MAX( CUSTOMER.C_CUSTKEY )

     2  -  READ INDEX COLUMN : CUSTOMER.C_CUSTKEY

           AGGREGATION : MIN( CUSTOMER.C_CUSTKEY ),

MAX( CUSTOMER.C_CUSTKEY )


<<<  end print plan
```

Order By

处理order by的过程

通常生成SORT INSTANT后处理order by

SORT INSTANT node的order by可按照如下两种方法进行处理

- 使用sort instant table排列
- 使用limit sort排列（order by与limit同时使用时适用limit sort方法）

下级中间结果对order by key column排列而上升时可以不积累单独的hash instant而进行处理

以下为生成SORT INSTANT后处理order by的示例

```
\EXPLAIN PLAN
  SELECT c_nationkey
    FROM customer
   WHERE c_comment like '%special%requests%'
ORDER BY c_nationkey;
...
>>>  start print plan

< Execution Plan >
================================================================================
|  IDX  |  NODE DESCRIPTION                                                    |
--------------------------------------------------------------------------------
|    0  |  SELECT STATEMENT                                                    |
|    1  |    QUERY BLOCK ("$QB_IDX_2")                                         |
|    2  |      SORT INSTANT                                                    |
|    3  |        TABLE ACCESS ("CUSTOMER")                                     |
================================================================================

    1  -  TARGET : CUSTOMER.C_NATIONKEY
```

```
     2  -  SORT KEY : "CUSTOMER.C_NATIONKEY ASC NULLS LAST"
           READ KEY COLUMN : CUSTOMER.C_NATIONKEY
     3  -  READ COLUMN : CUSTOMER.C_NATIONKEY, CUSTOMER.C_COMMENT
             LOGICAL FILTER : CUSTOMER.C_COMMENT LIKE '%special%requests%'


<<<  end print plan
```

以下为使用LIMIT SORT方法处理order by的示例在SORT INSTANT node内执行

```
\EXPLAIN PLAN
  SELECT c_nationkey
    FROM customer
   WHERE c_comment like '%special%requests%'
ORDER BY c_nationkey
   LIMIT 3;
...
>>>  start print plan


< Execution Plan >
================================================================================
========================
|  IDX  |  NODE DESCRIPTION                                                    |
ROWS |
--------------------------------------------------------------------------------
------------------------
|    0  |  SELECT STATEMENT                                                    |
```

```
0 |
|    1  |    QUERY BLOCK ("$QB_IDX_2")                                        |
0 |
|    2  |      SORT INSTANT                                                   |
0 |
|    3  |        TABLE ACCESS ("CUSTOMER")                                    |
0 |
=============================================================================
=========================


     1  -  TARGET : CUSTOMER.C_NATIONKEY
     2  -  LIMIT SORT
           SORT KEY : "CUSTOMER.C_NATIONKEY ASC NULLS LAST"
           READ KEY COLUMN : CUSTOMER.C_NATIONKEY
     3  -  READ COLUMN : CUSTOMER.C_NATIONKEY, CUSTOMER.C_COMMENT
             LOGICAL FILTER : CUSTOMER.C_COMMENT LIKE '%special%requests%'


<<<  end print plan
```

以下为使用下级的排列的中间结果处理order by的示例省略了order by处理

```
>>>  start print plan


\EXPLAIN PLAN
  SELECT c_nationkey
    FROM customer
```

```
   WHERE c_comment like '%special%requests%'
     AND c_nationkey >= 15
     AND c_nationkey < 20
ORDER BY c_nationkey;
...



>>>  start print plan


< Execution Plan >
============================================================================
|  IDX  |  NODE DESCRIPTION                                                |
----------------------------------------------------------------------------
|    0  |  SELECT STATEMENT                                                |
|    1  |    QUERY BLOCK ("$QB_IDX_2")                                     |
|    2  |      INDEX ACCESS ("CUSTOMER", "CUSTOMER_NATIONKEY_FK")          |
============================================================================


     1  -  TARGET : CUSTOMER.C_NATIONKEY
     2  -  READ INDEX COLUMN : CUSTOMER.C_NATIONKEY
           READ TABLE COLUMN : CUSTOMER.C_COMMENT
             MIN RANGE : CUSTOMER.C_NATIONKEY >= 15
             MAX RANGE : CUSTOMER.C_NATIONKEY IS NOT NULL AND
CUSTOMER.C_NATIONKEY < 20
             LOGICAL TABLE FILTER : CUSTOMER.C_COMMENT LIKE
```

```
'%special%requests%'


<<<  end print plan
```

5.4 Cluster

本章介绍集群系统中的cluster query优化

Cluster system相关详细内容参考**SUNDB 集群系统架构**

Table Sharding策略

Table sharding策略有如下两种

- Cloned table
- Sharded table

Table sharding策略的说明与示例如下

Cloned Table

Cloned table中表的所有数据统一复制并存储在所有group的所有node中因此适合更新较少或数据量相对少的表

以下为以cloned生成customer table的示例

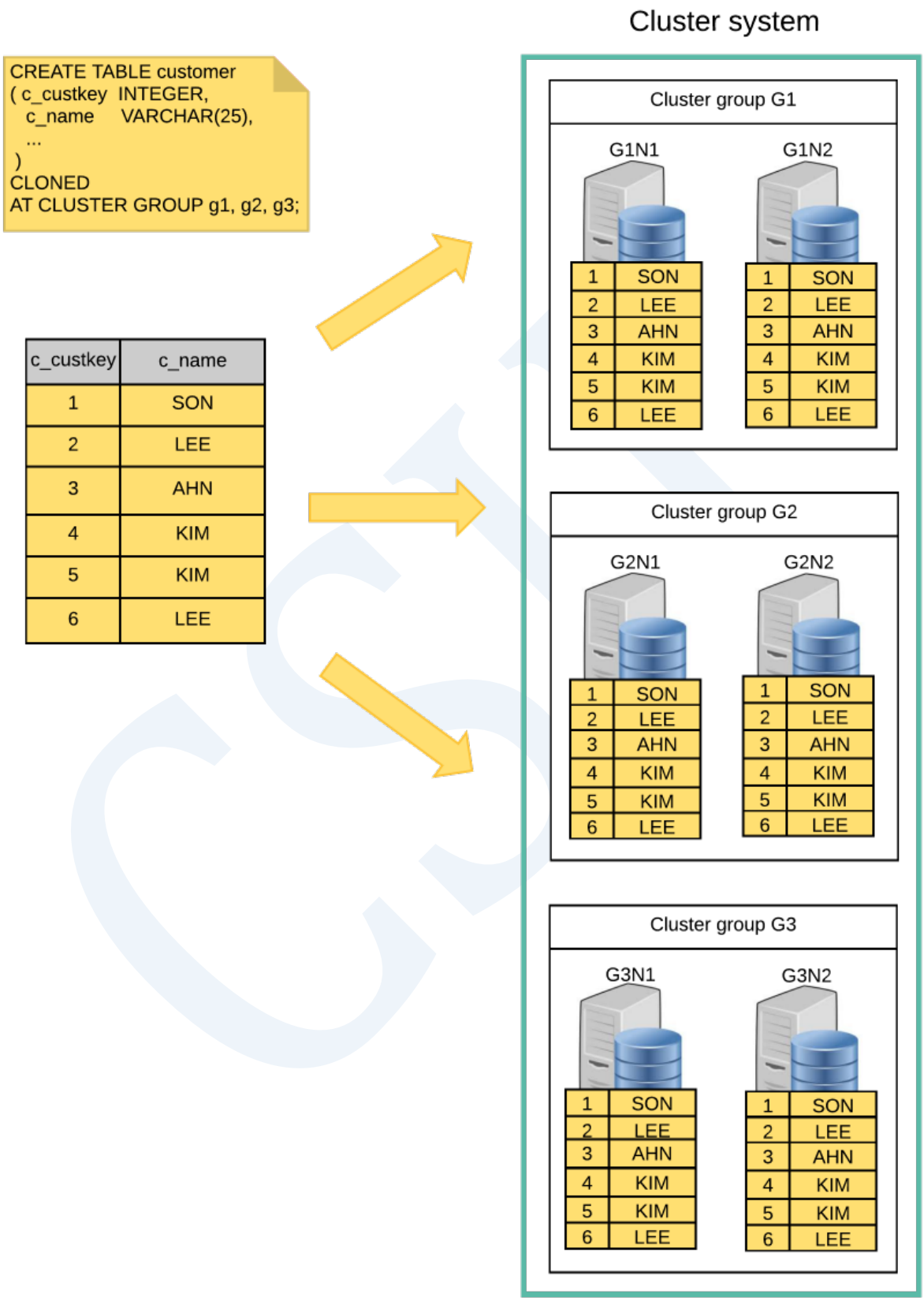


Figure 5-28 Cloned table

Sharded Table

数据由shard key以group为单位分片存储一个group中的node拥有相同的data因此适合数据量多而需要分片的情况根据分片策略分为如下三种

- Hash shard
- Range shard
- List shard

以下为以hash shard创建order table的示例

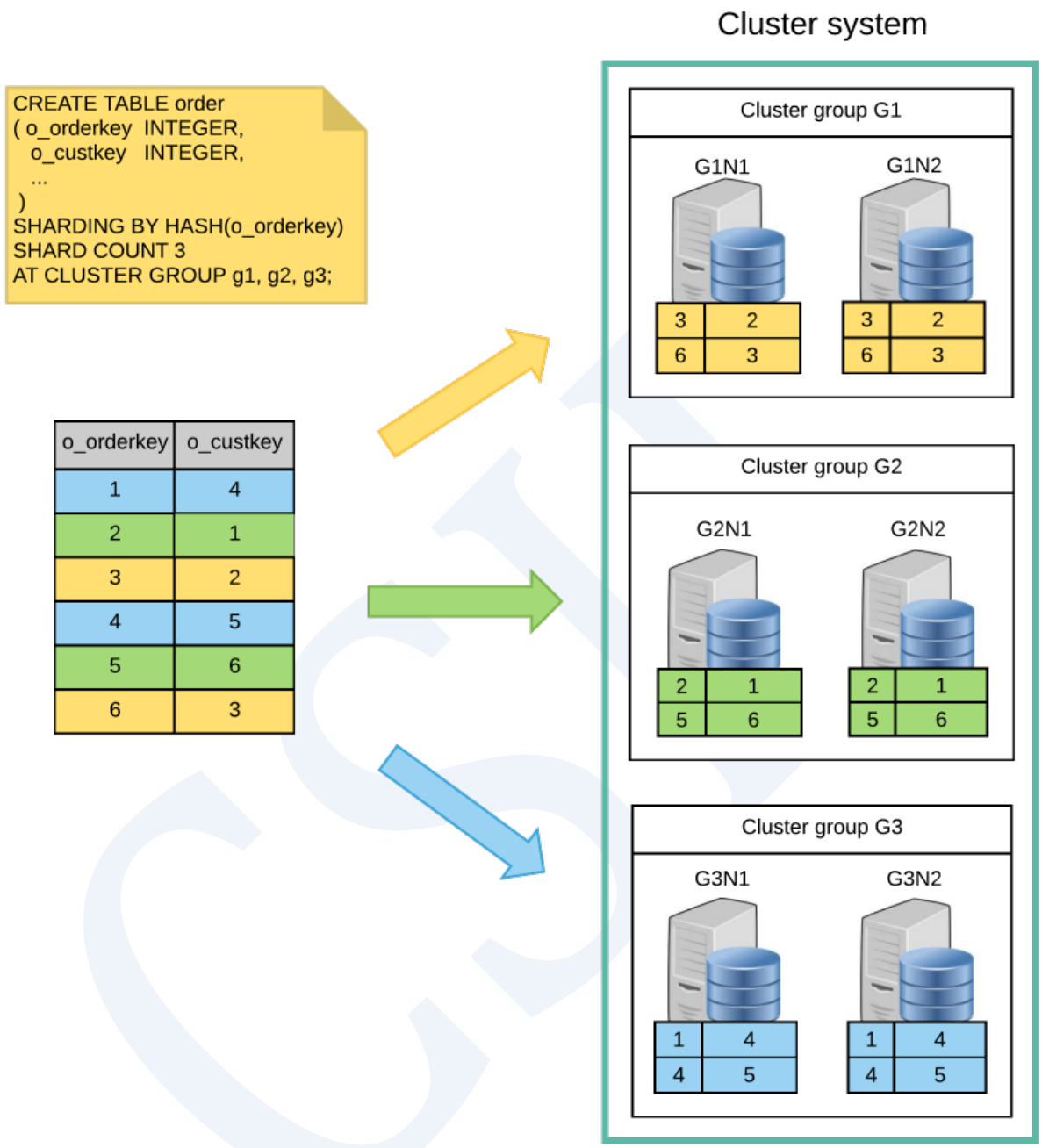


Figure 5-29 Sharded table

Access

本章节中介绍cluster query访问单张表的情况

以下为当前服务器为G1N1时local access和remote access的视图化

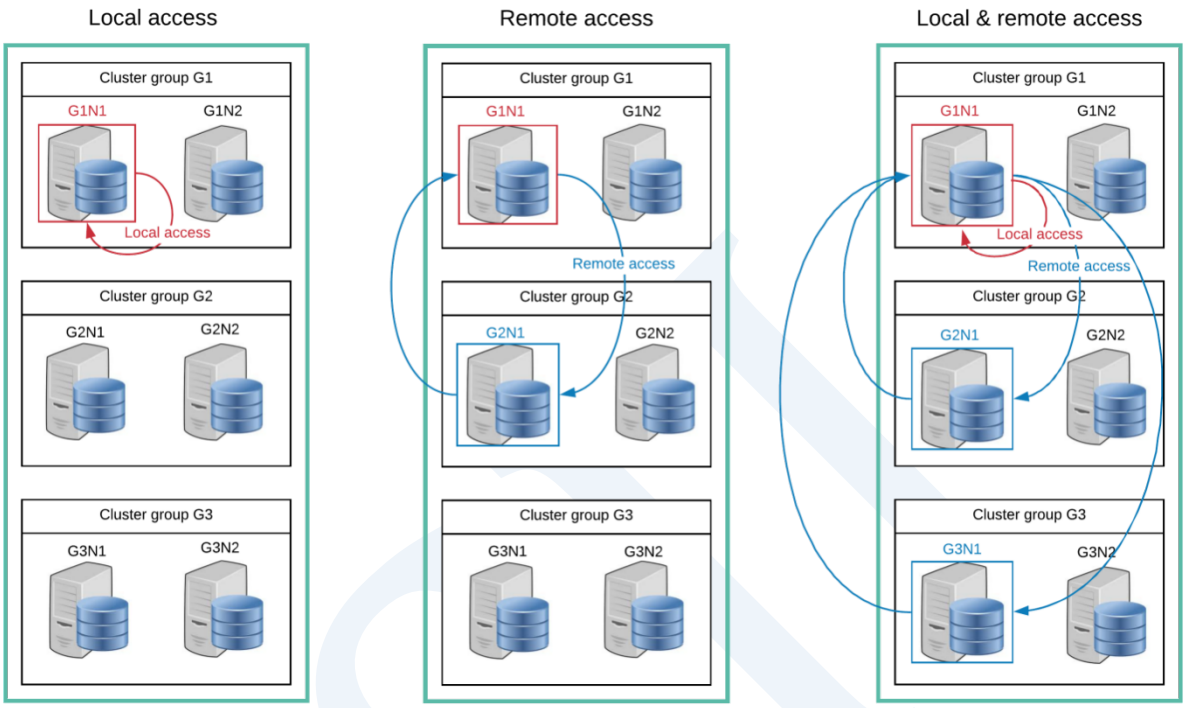


Figure 5-30 Cluster access

Local Access

在driver的角度上仅可在当前服务器执行操作

如下查询cloned table时所有group中的所有node的数据相同因此仅在当前服务器执行即可

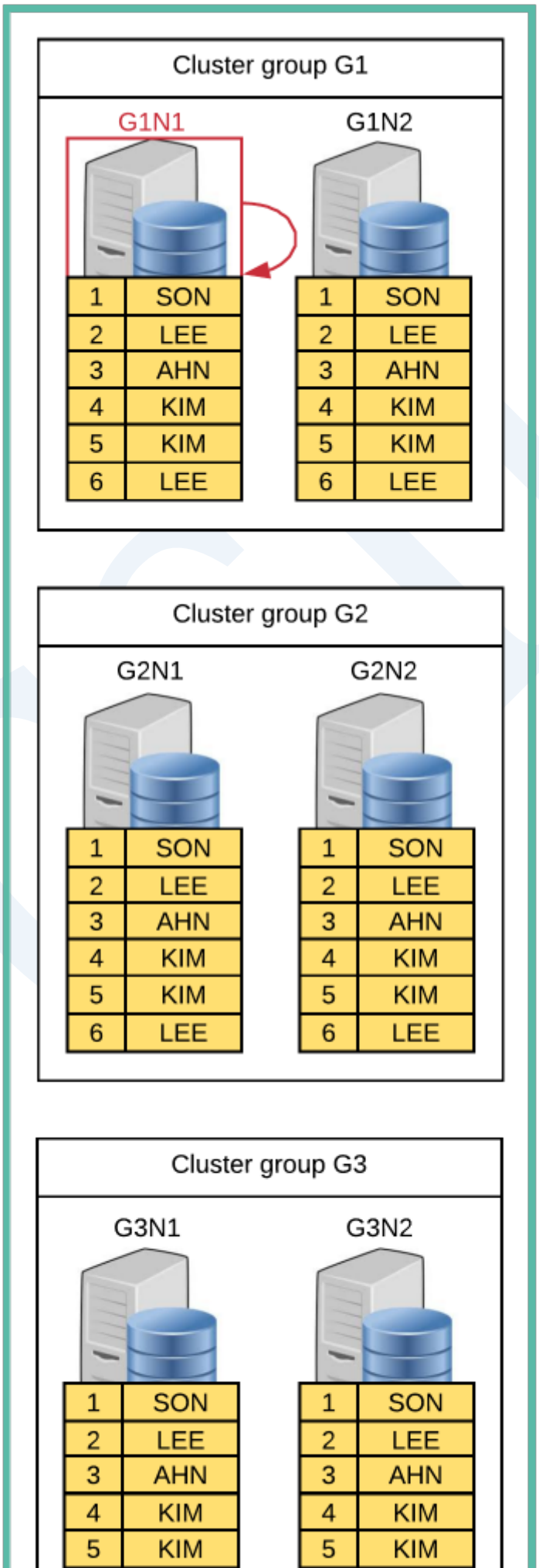

Cluster group G1
G1N1
G1N2
1 SON
2 LEE
3 AHN
4 KIM
5 KIM
6 LEE
Cluster group G2
G2N1
G2N2
1 SON
2 LEE
3 AHN
4 KIM
5 KIM
6 LEE
Cluster group G3
G3N1
G3N2
1 SON
2 LEE
3 AHN
4 KIM
5 KIM

Figure 5-31 Local access (cloned table)

```
\EXPLAIN PLAN SELECT * FROM customer;


C_CUSTKEY C_NAME

--------- ------

        1 SON

        2 LEE

        3 AHN

        4 KIM

        5 KIM

        6 LEE


6 rows selected.


>>>  start print plan


< Execution Plan >

=============================================================================

| IDX |  NODE DESCRIPTION                                          | ROWS |

-----------------------------------------------------------------------------

|  0  |  SELECT STATEMENT                                          |    6 |

|  1  |    QUERY BLOCK ("$QB_IDX_2")                               |    6 |

|  2  |      TABLE ACCESS ("CUSTOMER")                             |    6 |

=============================================================================
```

```
     1  -  TARGET : CUSTOMER.C_CUSTKEY, CUSTOMER.C_NAME
     2  -  CLONED
           READ COLUMN : CUSTOMER.C_CUSTKEY, CUSTOMER.C_NAME


<<<  end print plan
```

Sharded table中数据由shard key以group为单位分布存储因此可确认有shard key的filter并其值仅可在当前服务器执行时进行local access

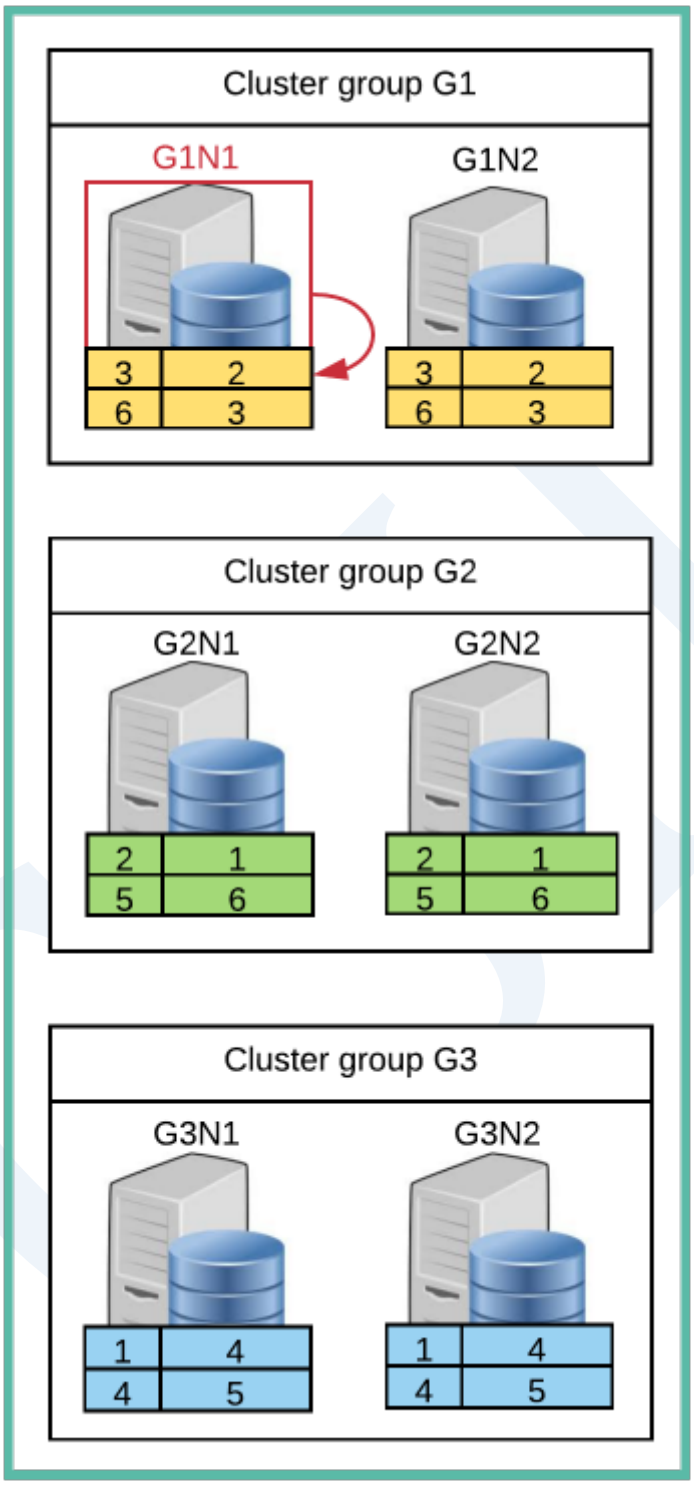


Figure 5-32 Local access (sharded table)

```
\EXPLAIN PLAN SELECT * FROM orders WHERE o_orderkey = 3;
```

```
O_ORDERKEY O_CUSTKEY
---------- ---------
         3         2

1 row selected.

>>>  start print plan

< Execution Plan >
================================================================================
=
| IDX |  NODE DESCRIPTION                                                 | ROWS
|
--------------------------------------------------------------------------------
-
|  0  |  SELECT STATEMENT                                                 |    1
|
|  1  |    QUERY BLOCK ("$QB_IDX_2")                                      |    1
|
|  2  |      TABLE ACCESS ("ORDERS")                                      |    1
|
================================================================================
=
```

```
    1  -  TARGET : ORDERS.O_ORDERKEY, ORDERS.O_CUSTKEY

    2  -  HASH SHARD ( # 3 )

          READ COLUMN : ORDERS.O_ORDERKEY, ORDERS.O_CUSTKEY

            PHYSICAL FILTER : ORDERS.O_ORDERKEY = 3


<<<  end print plan
```

Remote Access

Sharded table中数据由shard key以group为单位分布存储有shard key的filter时可以仅remote access特定服务器并获取结果

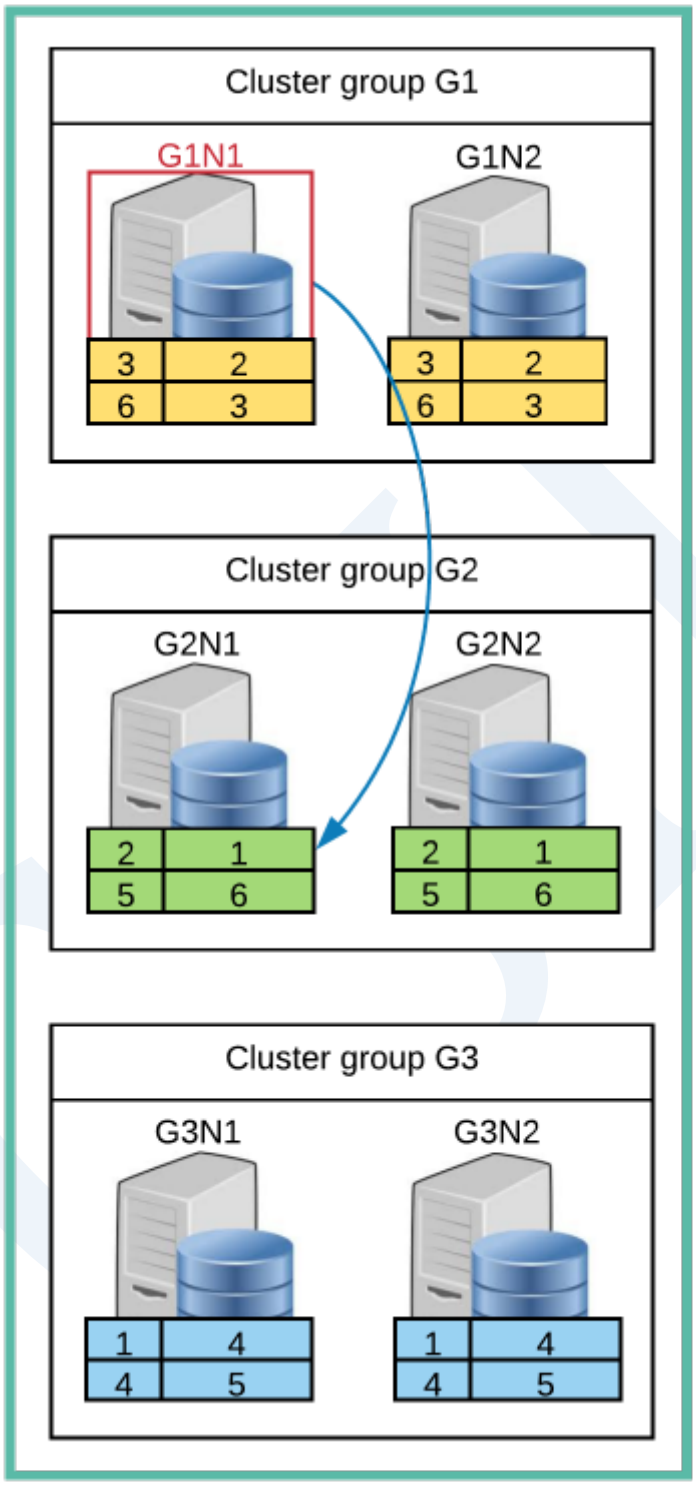


Figure 5-33 Remote access

```
\EXPLAIN PLAN SELECT * FROM orders WHERE o_orderkey = 2;
```

```
O_ORDERKEY O_CUSTKEY
---------- ---------
         2         1

1 row selected.

>>>  start print plan

< Execution Plan >
================================================================================
=
|  IDX  |  NODE DESCRIPTION                                |          ROWS
|
--------------------------------------------------------------------------------
-
|  0  |  SELECT STATEMENT                                  |             1
|
|  1  |    QUERY BLOCK ("$QB_IDX_2")                       |             1
|
|  2  |      PLAN BASED CLUSTER                            | REMOTE ONLY 1
|
|  3  |        TABLE ACCESS ("ORDERS")                     |             0
|
================================================================================
```

```
=


     1  -  TARGET : ORDERS.O_ORDERKEY, ORDERS.O_CUSTKEY
     2  -  SQL : SELECT /*+ FULL( _A1 ) */
                          "_A1"."O_ORDERKEY", "_A1"."O_CUSTKEY"
                    FROM "PUBLIC"."ORDERS"@LOCAL AS "_A1"
                   WHERE "_A1"."O_ORDERKEY" = :_V0
           TARGET DOMAIN : G2(G2N1,G2N2) 1 rows
     3  -  HASH SHARD ( # 3 )
           READ COLUMN : ORDERS.O_ORDERKEY, ORDERS.O_CUSTKEY
             PHYSICAL FILTER : ORDERS.O_ORDERKEY = 2


<<<  end print plan
```

如上述execution plan可查看到通过remote向G2传输SQL并获取了结果

为Sharded table而没有shard key的filter时要向各个服务器传输查询并获取结果

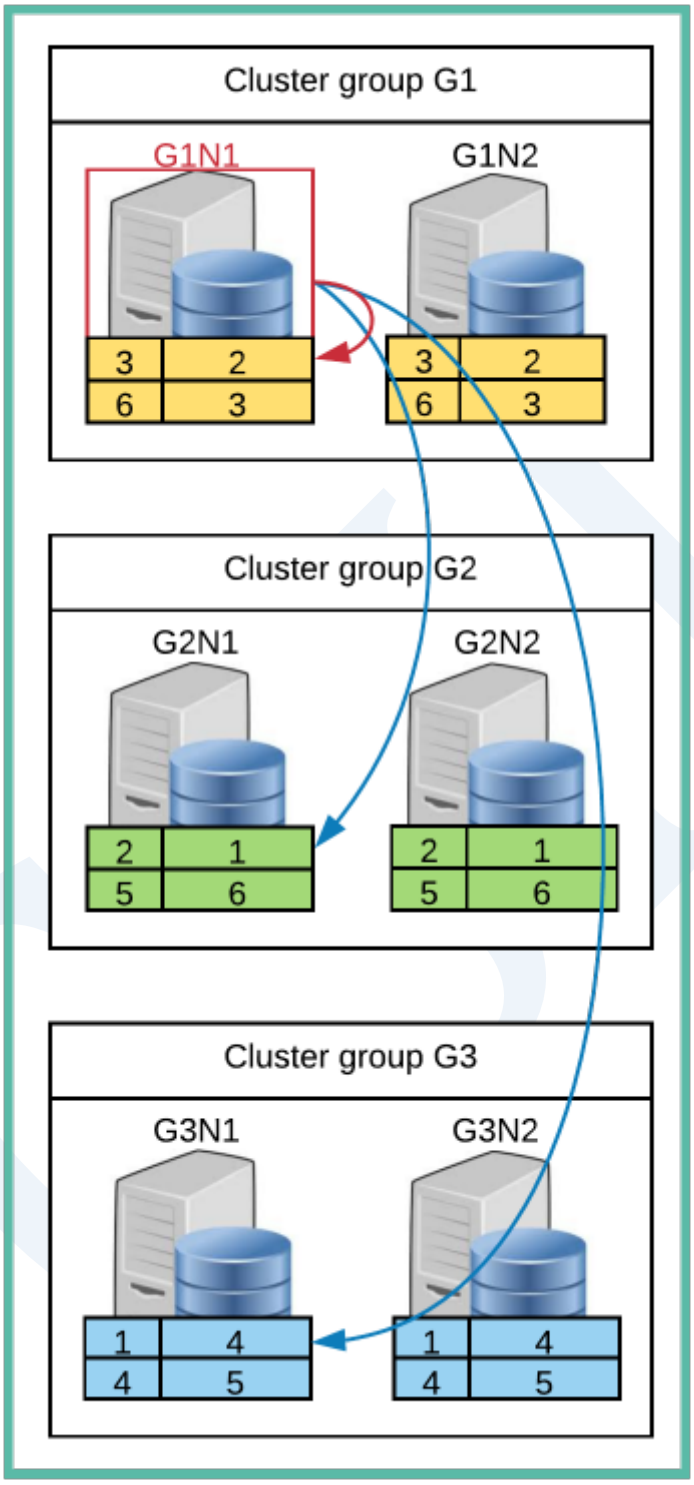


Figure 5-34 Local & remote access

```
\EXPLAIN PLAN SELECT * FROM orders WHERE o_custkey = 1;
```

```
O_ORDERKEY O_CUSTKEY
---------- ---------
         2         1

1 row selected.

>>>  start print plan

< Execution Plan >
================================================================================
=
|  IDX  |  NODE DESCRIPTION                              |          ROWS
|
--------------------------------------------------------------------------------
-
|    0  |  SELECT STATEMENT                              |             1
|
|  1  |    QUERY BLOCK ("$QB_IDX_2")                     |             1
|
|  2  |      PLAN BASED CLUSTER                          |LOCAL/REMOTE 1
|
|  3  |        TABLE ACCESS ("ORDERS")                   |             0
|
================================================================================
```

```
=


     1  -  TARGET : ORDERS.O_ORDERKEY, ORDERS.O_CUSTKEY
     2  -  SQL : SELECT /*+ FULL( _A1 ) */
                          "_A1"."O_ORDERKEY", "_A1"."O_CUSTKEY"
                    FROM "PUBLIC"."ORDERS"@LOCAL AS "_A1"
                   WHERE "_A1"."O_CUSTKEY" = :_V0
           TARGET DOMAIN : G1(G1N1,G1N2) 0 rows,
                           G2(G2N1,G2N2) 1 rows,
                           G3(G3N1,G3N2) 0 rows
     3  -  HASH SHARD ( # 3 )
           READ COLUMN : ORDERS.O_ORDERKEY, ORDERS.O_CUSTKEY
             PHYSICAL FILTER : ORDERS.O_CUSTKEY = 1


<<<  end print plan
```

如上execution plan可查看到通过remote向G1G2G3传输SQL并获取了结果

Join

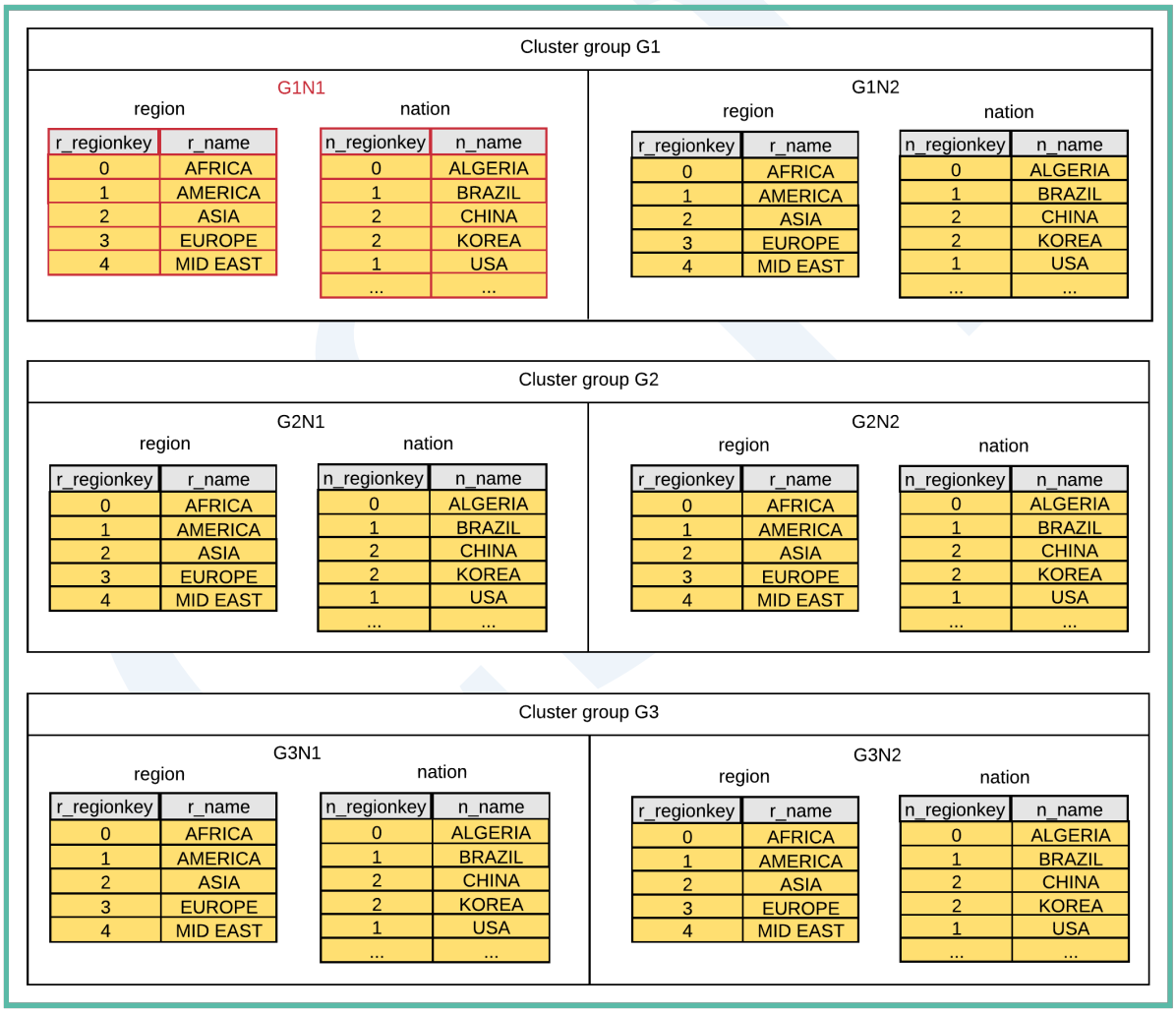
Local Join

在当前服务器执行joinLocal join有如下多种形式

以下为join region和nation的示例两张表均为cloned tableCloned table的所有group的node拥有

相同的数据因此在如下driver为G1N1的示例中仅可通过G1N1的数据进行join

```
\EXPLAIN PLAN
SELECT r_name, n_name
  FROM region, nation
 WHERE r_regionkey = n_regionkey;
```

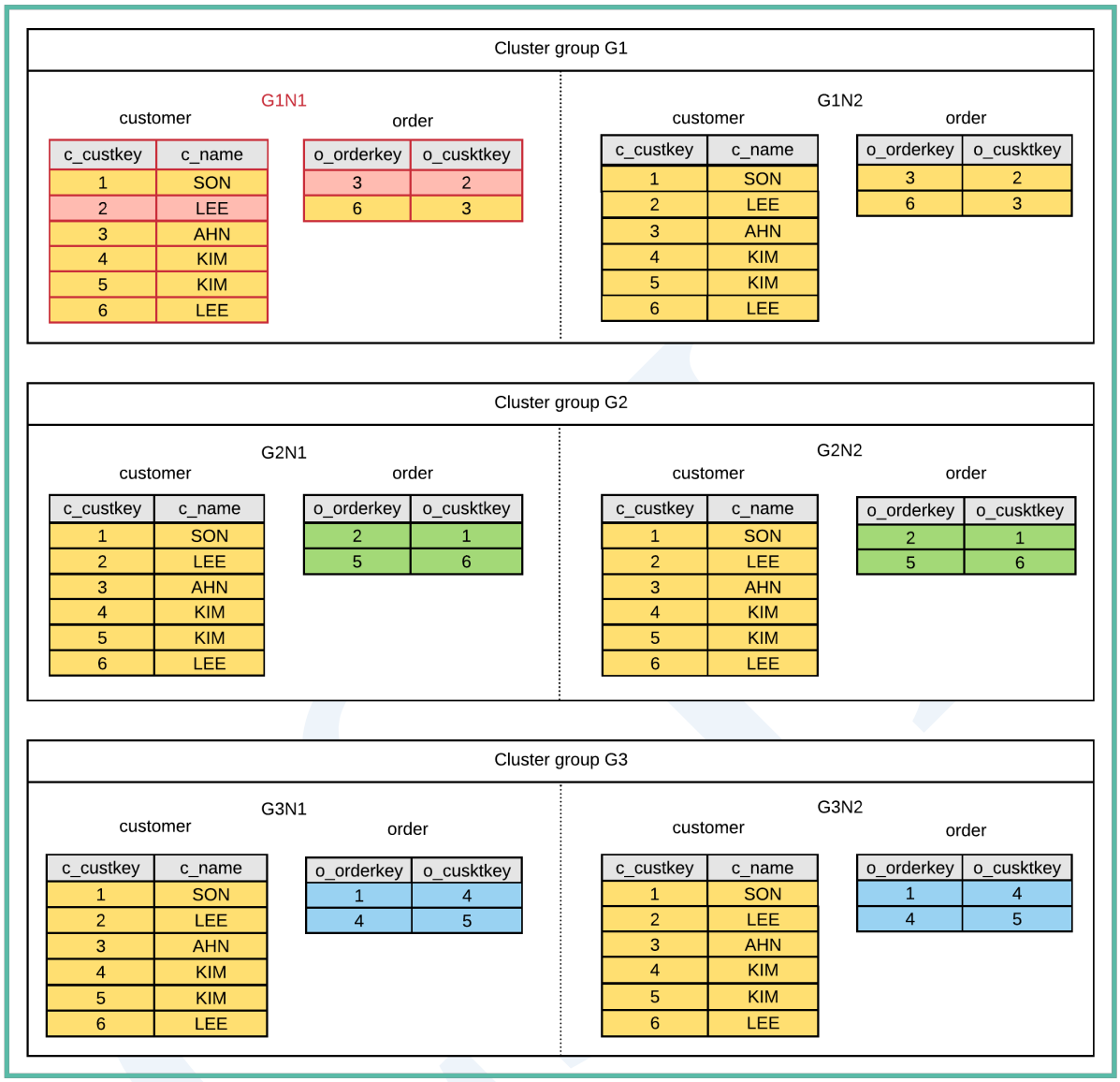



```
>>>  start print plan


< Execution Plan >
```

```
=============================================================================
|  IDX  |  NODE DESCRIPTION                                                 |
-----------------------------------------------------------------------------
|    0  |  SELECT STATEMENT                                                 |
|    1  |    QUERY BLOCK ("$QB_IDX_2")                                      |
|    2  |      HASH JOIN (INNER JOIN)                                       |
|    3  |        TABLE ACCESS ("NATION")                                    |
|    4  |        HASH JOIN INSTANT                                          |
|    5  |          TABLE ACCESS ("REGION")                                  |
=============================================================================


     1  -  TARGET : REGION.R_NAME, NATION.N_NAME
     2  -  JOINED COLUMN : REGION.R_NAME, NATION.N_NAME
     3  -  CLONED
           READ COLUMN : NATION.N_NAME, NATION.N_REGIONKEY
     4  -  HASH KEY : REGION.R_REGIONKEY
           RECORD COLUMN : REGION.R_NAME
           READ KEY COLUMN : REGION.R_REGIONKEY, REGION.R_NAME
             HASH FILTER : REGION.R_REGIONKEY = NATION.N_REGIONKEY
           FETCH ONE ROW
     5  -  CLONED
           READ COLUMN : REGION.R_REGIONKEY, REGION.R_NAME


<<<  end print plan
```

以下为join customer和orders的示例Customer为cloned tableorder为sharded tableSharded table的数据由shard key按照group单位划分如以下示例有shard key column的filter并可知其数据在当前服务器时可如下进行join

```
\EXPLAIN PLAN
SELECT c_custkey, COUNT(o_orderkey)
  FROM customer, orders
 WHERE c_custkey = o_custkey
   AND o_orderkey = 3
GROUP BY c_custkey;
```

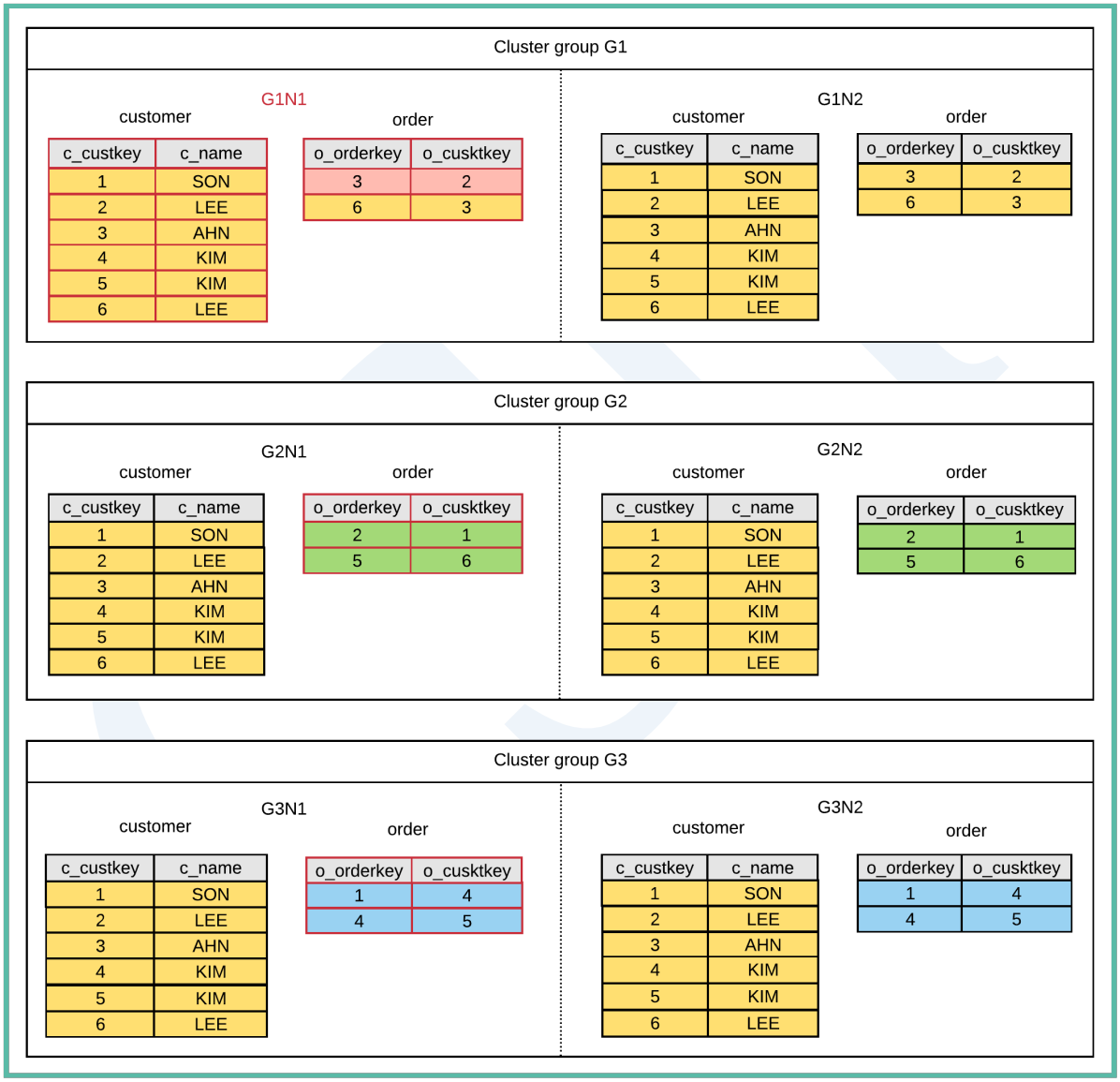



```
>>>  start print plan


< Execution Plan >

================================================================================

|  IDX  |  NODE DESCRIPTION                                                    |

--------------------------------------------------------------------------------

|    0  |  SELECT STATEMENT                                                    |
```

```
|    1  |    QUERY BLOCK ("$QB_IDX_2")                                    |
|    2  |      GROUP HASH INSTANT                                          |
|    3  |        NESTED JOIN (INNER JOIN)                                  |
|    4  |          INDEX ACCESS ("ORDERS", "ORDERS_PK_INDEX")              |
|    5  |          INDEX ACCESS ("CUSTOMER", "CUSTOMER_PK_INDEX")          |
=======================================================================


    1  -  TARGET : CUSTOMER.C_CUSTKEY, COUNT( ORDERS.O_ORDERKEY )
    2  -  GROUP KEY : CUSTOMER.C_CUSTKEY
          RECORD COLUMN : COUNT( ORDERS.O_ORDERKEY )
          READ KEY COLUMN : CUSTOMER.C_CUSTKEY
          READ RECORD COLUMN : COUNT( ORDERS.O_ORDERKEY )
    3  -  JOINED COLUMN : CUSTOMER.C_CUSTKEY, ORDERS.O_ORDERKEY
    4  -  HASH SHARD ( # 3 )
          READ INDEX COLUMN : ORDERS.O_ORDERKEY
          READ TABLE COLUMN : ORDERS.O_CUSTKEY
            MIN RANGE : ORDERS.O_ORDERKEY = 3
            MAX RANGE : ORDERS.O_ORDERKEY = 3
          FETCH ONE ROW
    5  -  CLONED
          READ INDEX COLUMN : CUSTOMER.C_CUSTKEY
            MIN RANGE : CUSTOMER.C_CUSTKEY = {ORDERS.O_CUSTKEY}
            MAX RANGE : CUSTOMER.C_CUSTKEY = {ORDERS.O_CUSTKEY}
          FETCH ONE ROW
```

```
<<<  end print plan
```

以下为customer和orders的join但没有shard key column的filter的情况此时需要从服务器获取sharded table的所有数据才可进行local join




```
\EXPLAIN PLAN
SELECT /*+ LOCAL_JOIN(orders) */
       c_custkey, COUNT(o_orderkey)
```

```
  FROM customer, orders
 WHERE c_custkey = o_custkey
GROUP BY c_custkey;
```

```
< Execution Plan >
================================================================================
==
|IDX|  NODE DESCRIPTION                                |           ROWS
|
--------------------------------------------------------------------------------
--
| 0 |  SELECT STATEMENT                                |              6
|
| 1 |    QUERY BLOCK ("$QB_IDX_2")                     |              6
|
| 2 |      HASH JOIN (INNER JOIN)                      |              6
|
| 3 |        PLAN BASED CLUSTER                        | LOCAL/REMOTE 6
|
| 4 |          INDEX ACCESS ("ORDERS", "ORDERS_PK_INDEX") | (6)          6
|
| 5 |        HASH JOIN INSTANT                         |              6
|
| 6 |          TABLE ACCESS ("CUSTOMER")               |              6
```

```
|
================================================================================
==

     1  -  TARGET : CUSTOMER.C_CUSTKEY, CUSTOMER.C_NAME, ORDERS.O_ORDERKEY
     2  -  JOINED COLUMN : CUSTOMER.C_CUSTKEY, CUSTOMER.C_NAME,
ORDERS.O_ORDERKEY
     3  -  SQL : SELECT /*+ INDEX( _A1, "PUBLIC"."ORDERS_PK_INDEX" ) */
                        "_A1"."O_ORDERKEY"
                   FROM "PUBLIC"."ORDERS"@LOCAL AS "_A1"
            TARGET DOMAIN : G1(G1N1,G1N2) 2 rows,
                            G2(G2N1,G2N2) 2 rows,
                            G3(G3N1,G3N2) 2 rows
     4  -  HASH SHARD ( # 3 )
            READ INDEX COLUMN : ORDERS.O_ORDERKEY
     5  -  HASH KEY : CUSTOMER.C_CUSTKEY
            RECORD COLUMN : CUSTOMER.C_NAME
            READ KEY COLUMN : CUSTOMER.C_CUSTKEY, CUSTOMER.C_NAME
              HASH FILTER : CUSTOMER.C_CUSTKEY = ORDERS.O_ORDERKEY
            FETCH ONE ROW
     6  -  CLONED
            READ COLUMN : CUSTOMER.C_CUSTKEY, CUSTOMER.C_NAME

<<<  end print plan
```

如上述execution plan所示可查看到向G1G2G3传输SQL并获取了order table的所有数据

Remote Join

Remote join在各个服务器处理join

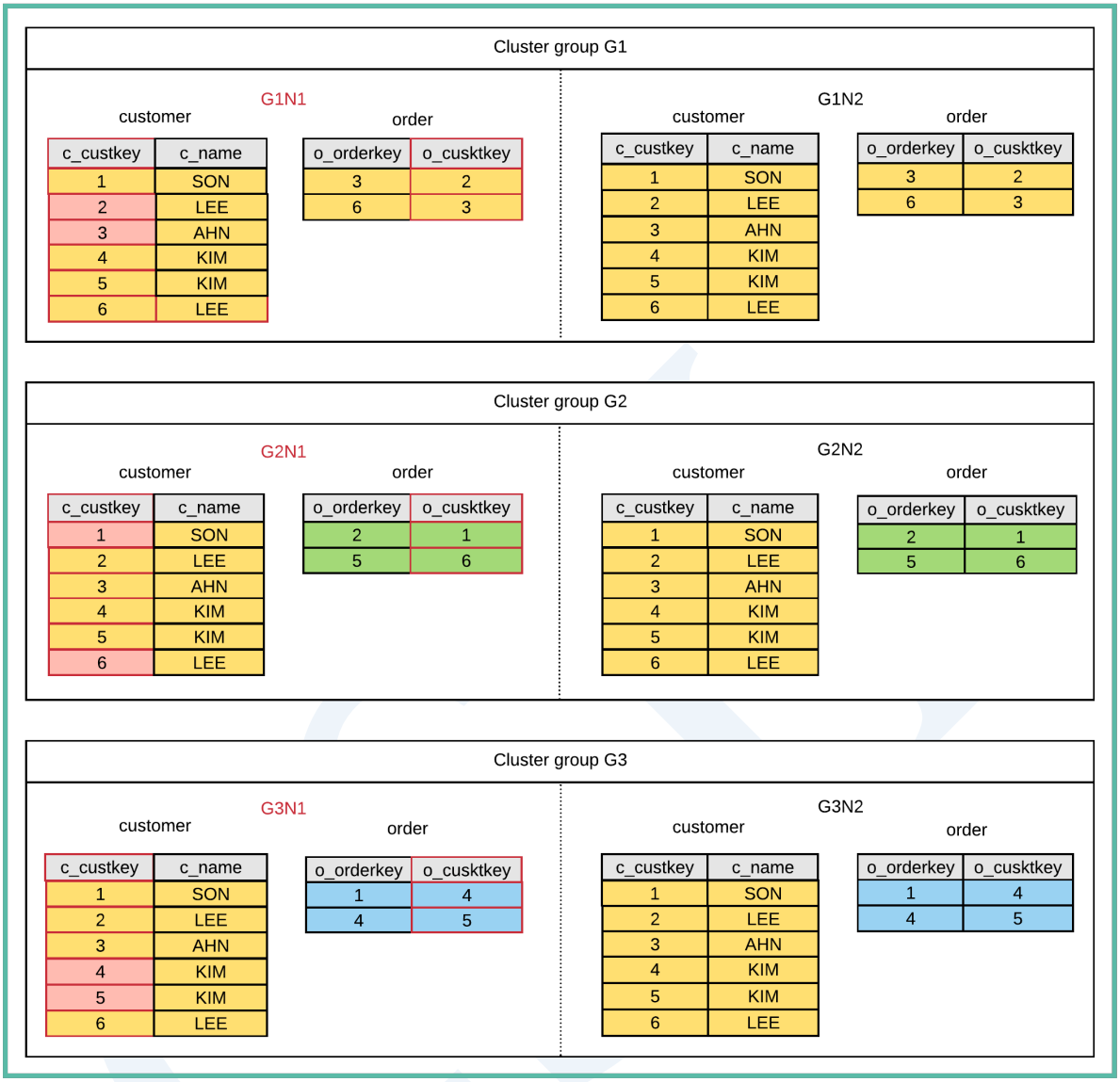
在各个服务器处理join可并行处理而且由于join结果大幅减少时在对应服务器执行join后获取其结果可以减少网络成本

在本节介绍remote的多种形式

Joining Cloned Table and Sharded Table

本节说明cloned table和sharded table的join

以下为join customer和orders的示例Customer为cloned tableorder为sharded table数据分布如下




```
\EXPLAIN PLAN

SELECT /*+ REMOTE_JOIN(orders) */

       c_custkey, COUNT(o_orderkey)

  FROM customer, orders

 WHERE c_custkey = o_custkey

GROUP BY c_custkey;
```

```
>>>  start print plan


< Execution Plan >

================================================================================
==
|IDX|  NODE DESCRIPTION                                |           ROWS
|
--------------------------------------------------------------------------------
--
| 0 |  SELECT STATEMENT                                |              6
|
| 1 |    QUERY BLOCK ("$QB_IDX_2")                     |              6
|
| 2 |      PLAN BASED CLUSTER                          | LOCAL/REMOTE 6
|
| 3 |        HASH JOIN (INNER JOIN)                    |              6
|
| 4 |          TABLE ACCESS ("ORDERS")                 |              6
|
| 5 |          HASH JOIN INSTANT                       |              6
|
| 6 |            TABLE ACCESS ("CUSTOMER")             |              6
|
================================================================================
==
```

```
     1  -  TARGET : CUSTOMER.C_CUSTKEY, CUSTOMER.C_NAME, ORDERS.O_ORDERKEY
     2  -  SQL : SELECT /*+ KEEP_JOINED_TABLE USE_HASH_IN( _A1, 10 )
                                FULL( _A2 )
                                FULL( _A1 ) */
                          "_A1"."C_CUSTKEY", "_A1"."C_NAME", "_A2"."O_ORDERKEY"
                     FROM ("PUBLIC"."ORDERS"@LOCAL AS "_A2"
                           INNER JOIN
                           "PUBLIC"."CUSTOMER"@LOCAL AS "_A1"
                        ON "_A1"."C_CUSTKEY" = "_A2"."O_ORDERKEY") ALIAS "_A3"
           TARGET DOMAIN : G1(G1N1,G1N2) 2 rows,
                           G2(G2N1,G2N2) 2 rows,
                           G3(G3N1,G3N2) 2 rows
     3  -  JOINED COLUMN : CUSTOMER.C_CUSTKEY, CUSTOMER.C_NAME,
ORDERS.O_ORDERKEY
     4  -  HASH SHARD ( # 3 )
           READ COLUMN : ORDERS.O_ORDERKEY
     5  -  HASH KEY : CUSTOMER.C_CUSTKEY
           RECORD COLUMN : CUSTOMER.C_NAME
           READ KEY COLUMN : CUSTOMER.C_CUSTKEY, CUSTOMER.C_NAME
             HASH FILTER : CUSTOMER.C_CUSTKEY = ORDERS.O_ORDERKEY
     6  -  CLONED
           READ COLUMN : CUSTOMER.C_CUSTKEY, CUSTOMER.C_NAME

<<<  end print plan
```

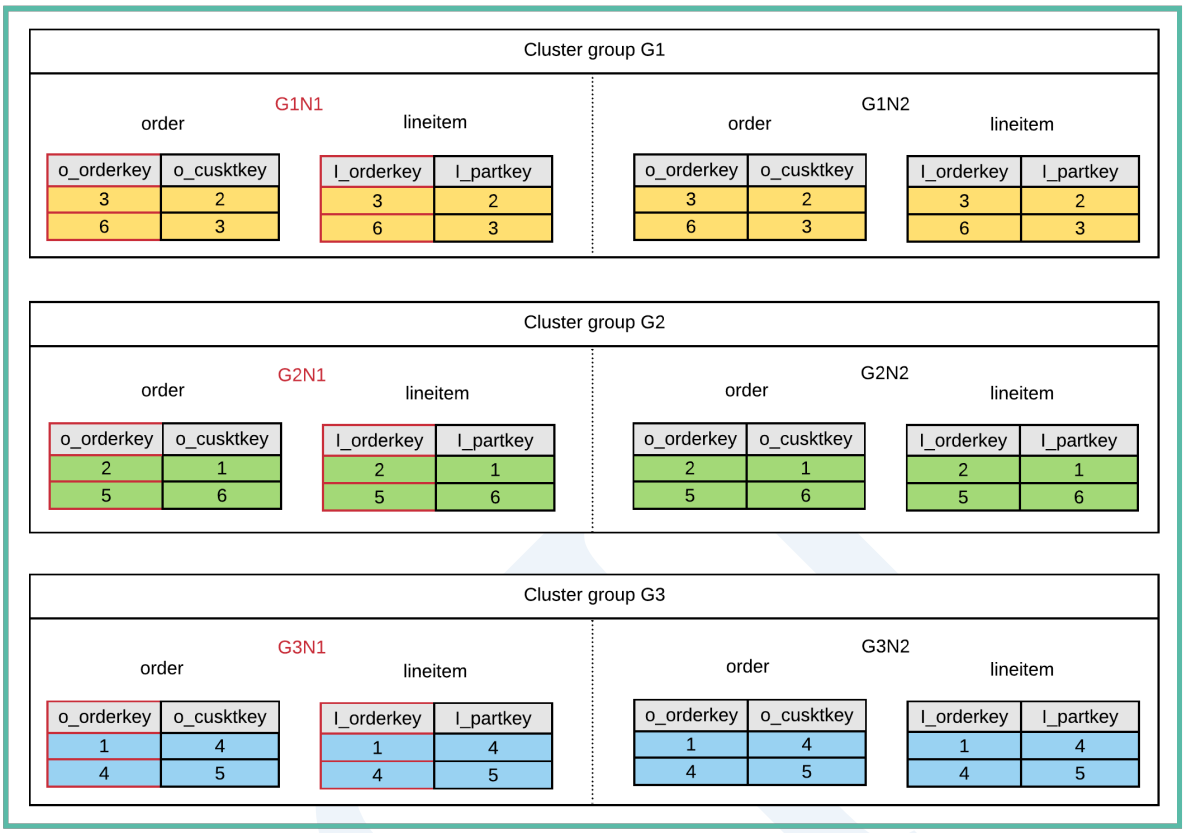
如上述execution plan可查看到向G1G2G3传输SQL并在对应服务器执行join后获取了其结果

Joining Sharded Table and Sharded Table

满足如下条件时可执行remote join

- 有shard key join condition(例: t1.shardKeyCol = t2.shardKeyCol)
- Shard strategy相同
    - Joining hash sharded table and hash sharded table
    - Joining range shard and range shard
    - Joining list shard and list shard
- Shard count相同
- Shard key column的类型相同
- Shard key column数量相同

以下为join orders和lineitem的示例两张表均为hash sharded table有shard key join condition对相同标准的orderkey进行了sharding因此在各个服务器执行join即可




```
\EXPLAIN PLAN

SELECT o_orderkey, l_partkey

  FROM orders, lineitem

 WHERE o_orderkey = l_orderkey;
```

```
>>>  start print plan


< Execution Plan >

=============================================================================

==

|IDX|  NODE DESCRIPTION                                |            ROWS
```

```
|
-------------------------------------------------------------------------
--
| 0 |  SELECT STATEMENT                                 |              6
|
| 1 |    QUERY BLOCK ("$QB_IDX_2")                      |              6
|
| 2 |      PLAN BASED CLUSTER                           | LOCAL/REMOTE 6
|
| 3 |        HASH JOIN (INNER JOIN)                     |              2
|
| 4 |          INDEX ACCESS ("ORDERS", "ORDERS_PK_INDEX") | (2)          2
|
| 5 |          HASH JOIN INSTANT                        |              2
|
| 6 |            TABLE ACCESS ("LINEITEM")              |              2
|
=========================================================================
==


     1  -  TARGET : ORDERS.O_ORDERKEY, LINEITEM.L_PARTKEY
     2  -  SQL : SELECT /*+ KEEP_JOINED_TABLE USE_HASH_IN( _A1, 10 )
                            INDEX( _A2, "PUBLIC"."ORDERS_PK_INDEX" )
                            FULL( _A1 ) */
                       "_A2"."O_ORDERKEY", "_A1"."L_PARTKEY"
```

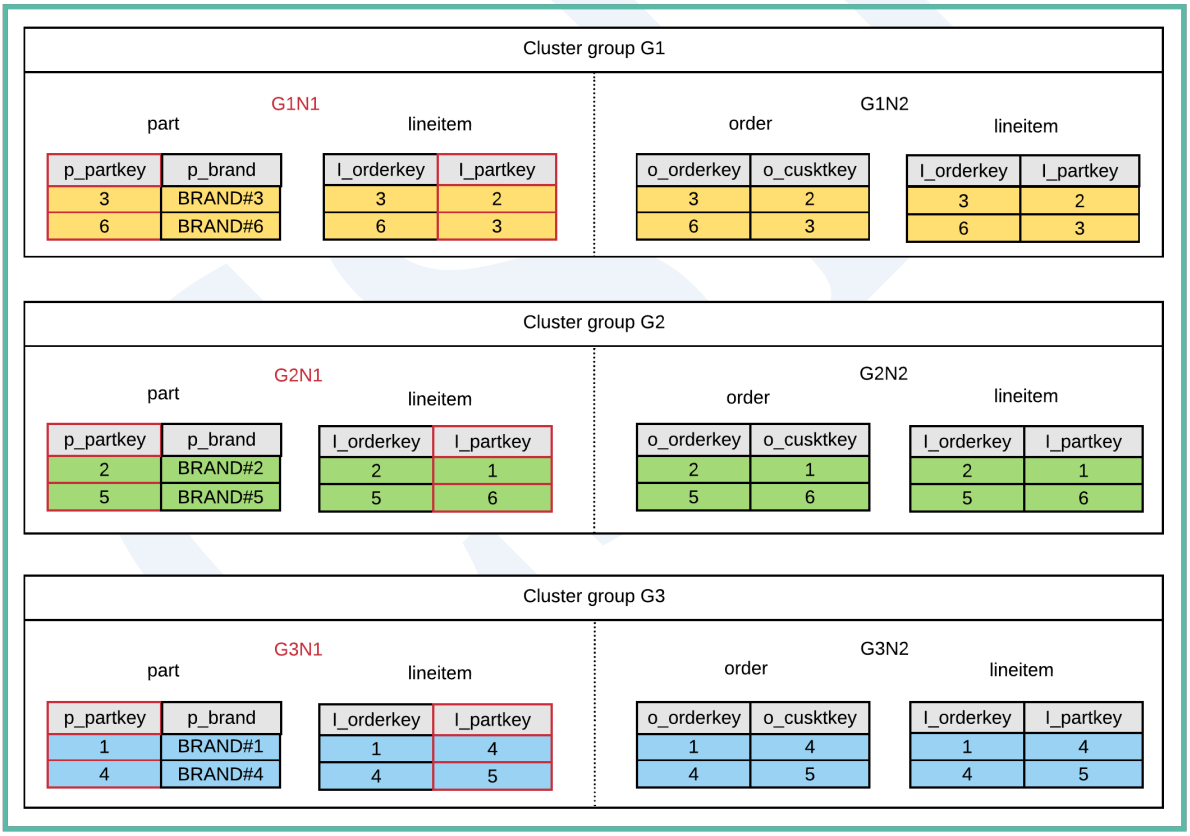
```
                FROM ( "PUBLIC"."ORDERS"@LOCAL AS "_A2"
                       INNER JOIN
                       "PUBLIC"."LINEITEM"@LOCAL AS "_A1"
                  ON "_A1"."L_ORDERKEY" = "_A2"."O_ORDERKEY") ALIAS
"_A3"
          TARGET DOMAIN : G1(G1N1,G1N2) 2 rows,
                          G2(G2N1,G2N2) 2 rows,
                          G3(G3N1,G3N2) 2 rows
    3  -  JOINED COLUMN : ORDERS.O_ORDERKEY, LINEITEM.L_PARTKEY
    4  -  HASH SHARD ( # 3 )
          READ INDEX COLUMN : ORDERS.O_ORDERKEY
    5  -  HASH KEY : LINEITEM.L_ORDERKEY
          RECORD COLUMN : LINEITEM.L_PARTKEY
          READ KEY COLUMN : LINEITEM.L_ORDERKEY, LINEITEM.L_PARTKEY
            HASH FILTER : LINEITEM.L_ORDERKEY = ORDERS.O_ORDERKEY
    6  -  HASH SHARD ( # 3 )
          READ COLUMN : LINEITEM.L_ORDERKEY, LINEITEM.L_PARTKEY


<<<  end print plan
```

如上execution plan所示可查看到向各个服务器传输join SQL并在对应服务器执行join后获取了其结果

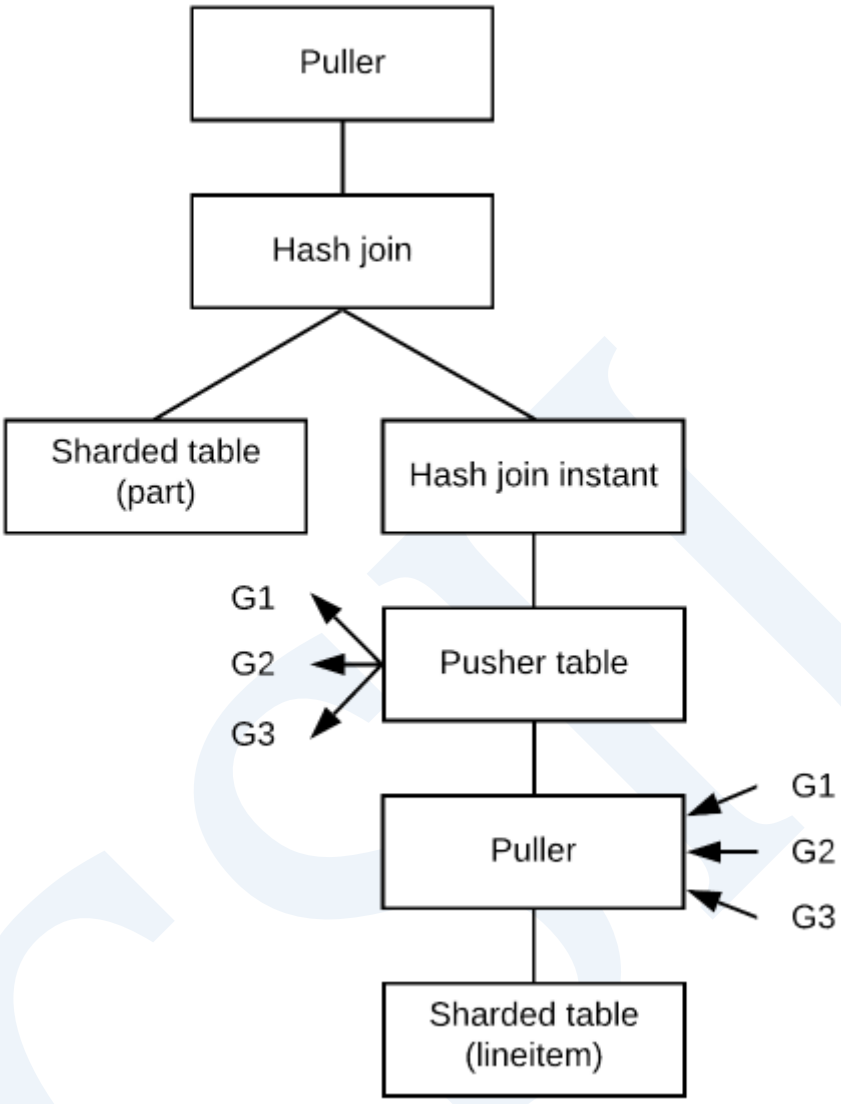
以下为join part和lineitem的示例两张表均为hash sharded table而没有shard key join condition的情况

Part和lineitem均为hash sharded table而part的shard key为p_partkey因此数据以p_partkey为准划分；lineitem的shard key为l_orderkey因此数据以l_orderkey为准划分

```
\EXPLAIN PLAN
SELECT /*+ REMOTE_JOIN(lineitem) */
       l_orderkey, p_partkey
  FROM part, lineitem
 WHERE p_partkey = l_partkey;
```



上述查询要执行remote join需要获取lineitem的所有数据后以l_partkey划分并传输至G1G2G3此时puller和pusher起作用



- **Cluster Puller**: 向各个服务器传输SQL并获取数据
- **Cluster Pusher**: 向各个服务器传输数据

上图中puller从lineitem获取所有数据之后l_partkey创建shard key的pusher table后向G1G2G3传输数据并在G1G2G3执行part和pusher表的remote join

Execution plan如下

```
>>>  start print plan
```

```
< Execution Plan >
================================================================================
==
|IDX|  NODE DESCRIPTION                                          |   ROWS
|
--------------------------------------------------------------------------------
--
| 0 |  SELECT STATEMENT                                          |      6
|
| 1 |    QUERY BLOCK ("$QB_IDX_2")                               |      6
|
| 2 |      SINGLE CLUSTER                                       |LOCAL/REMOTE 6
|
| 3 |        CLUSTER PUSHER ("_$NI_7")                           |      6
|
| 4 |          PLAN BASED CLUSTER                               |LOCAL/REMOTE 6
|
| 5 |            TABLE ACCESS ("LINEITEM")                       |      2
|
| 6 |        SELECT STATEMENT                                    |      2
|
| 7 |          QUERY BLOCK ("$QB_IDX_2")                         |      2
|
| 8 |            HASH JOIN (INNER JOIN)                          |      2
```

```
|
| 9 |              INDEX ACCESS ("PART" AS _A2, "PART_PK_INDEX")  | (2)  2
|
| 10 |              HASH JOIN INSTANT                              |      2
|
| 11 |                PUSHER TABLE ACCESS ("_$NI_7" AS _A1)        |     2
|
=============================================================================
==


     1  -  TARGET : _$NI_7.L_ORDERKEY, PART.P_PARTKEY
     2  -  SQL : SELECT /*+ KEEP_JOINED_TABLE USE_HASH_IN( _A1, 10 )
                            INDEX( _A2, "PUBLIC"."PART_PK_INDEX" )
                            FULL( _A1 ) */
                         "_A1"."L_ORDERKEY", "_A2"."P_PARTKEY"
                    FROM ( "PUBLIC"."PART"@LOCAL AS "_A2"
                             INNER JOIN
                           "SESSION_SCHEMA"."_$NI_7"@LOCAL AS "_A1"
                      ON "_A1"."L_PARTKEY" = "_A2"."P_PARTKEY") ALIAS "_A3"
            TARGET DOMAIN : G1(G1N1,G1N2) 2 rows,
                            G2(G2N1,G2N2) 2 rows,
                            G3(G3N1,G3N2) 2 rows
     3  -  SQL : DECLARE INSTANT TABLE "SESSION_SCHEMA"."_$NI_7"
                  ( "L_PARTKEY" NUMBER(10, 0), "L_ORDERKEY" NUMBER(10, 0) )
            COLUMN : LINEITEM.L_PARTKEY AS L_PARTKEY,
```

```
                    LINEITEM.L_ORDERKEY AS L_ORDERKEY
          SHARDED : LINEITEM.L_PARTKEY
          TARGET DOMAIN : G1(G1N1,G1N2) 2 rows,
                          G2(G2N1,G2N2) 2 rows,
                          G3(G3N1,G3N2) 2 rows
     4  -  SQL : SELECT /*+ FULL( _A1 ) */
                       "_A1"."L_ORDERKEY", "_A1"."L_PARTKEY"
                  FROM "PUBLIC"."LINEITEM"@LOCAL AS "_A1"
          TARGET DOMAIN : G1(G1N1,G1N2) 2 rows,
                          G2(G2N1,G2N2) 2 rows,
                          G3(G3N1,G3N2) 2 rows
     5  -  HASH SHARD ( # 3 )
          READ COLUMN : LINEITEM.L_ORDERKEY, LINEITEM.L_PARTKEY
     7  -  TARGET : _A1.L_ORDERKEY, _A2.P_PARTKEY
     8  -  JOINED COLUMN : _A1.L_ORDERKEY, _A2.P_PARTKEY
     9  -  HASH SHARD ( # 3 )
          READ INDEX COLUMN : _A2.P_PARTKEY
    10  -  HASH KEY : _A1.L_PARTKEY
          RECORD COLUMN : _A1.L_ORDERKEY
          READ KEY COLUMN : _A1.L_PARTKEY, _A1.L_ORDERKEY
            HASH FILTER : _A1.L_PARTKEY = _A2.P_PARTKEY
    11  -  READ COLUMN : _A1.L_PARTKEY, _A1.L_ORDERKEY

<<<  end print plan
```

从上述execution plan可知由4向G1G2G3发送SQL并获取了(l_orderkey, l_partkey)而且将在4获取的(l_partkey, l_orderkey)值存储在3中(l_partkey, l_orderkey)为shard key的pusher table此pusher table以l_partkey分割并发送到G1G2G3后临时存储

Remote join通过part和pusher table的join执行

Group By

Local Group By

在当前服务器执行group by

Cloned table的group by时所有group的node拥有相同的数据因此在当前服务器执行group by即可

```
\EXPLAIN PLAN
  SELECT c_nationkey, COUNT(c_custkey)
    FROM customer
GROUP BY c_nationkey;


>>>  start print plan


< Execution Plan >
================================================================================
|  IDX  |  NODE DESCRIPTION                                                     |
--------------------------------------------------------------------------------
```

```
|    0  |  SELECT STATEMENT                                                |
|    1  |    QUERY BLOCK ("$QB_IDX_2")                                     |
|    2  |      GROUP HASH INSTANT                                          |
|    3  |        TABLE ACCESS ("CUSTOMER")                                 |
=====================================================================


     1  -  TARGET : CUSTOMER.C_NATIONKEY, COUNT( CUSTOMER.C_CUSTKEY )
     2  -  GROUP KEY : CUSTOMER.C_NATIONKEY
           RECORD COLUMN : COUNT( CUSTOMER.C_CUSTKEY )
           READ KEY COLUMN : CUSTOMER.C_NATIONKEY
           READ RECORD COLUMN : COUNT( CUSTOMER.C_CUSTKEY )
     3  -  CLONED
           READ COLUMN : CUSTOMER.C_CUSTKEY, CUSTOMER.C_NATIONKEY


<<<  end print plan
```

Remote Group By

在各个服务器处理group by

在各个服务器处理group by可获得并行处理的效果而且大体上由于group by减少很多结果因此可减少获取数据的网络成本

```
\EXPLAIN PLAN
SELECT c_custkey, COUNT(o_orderkey)
  FROM customer, orders
```

```
 WHERE c_custkey = o_custkey
GROUP BY c_custkey;


>>>  start print plan


< Execution Plan >
================================================================================
=
|IDX|  NODE DESCRIPTION                                |           ROWS
|
--------------------------------------------------------------------------------
-
| 0 |  SELECT STATEMENT                                |              6
|
| 1 |    QUERY BLOCK ("$QB_IDX_2")                     |              6
|
| 2 |     SINGLE CLUSTER                               | LOCAL/REMOTE 6
|
| 3 |        SELECT STATEMENT                          |              2
|
| 4 |          QUERY BLOCK ("$QB_IDX_2")               |              2
|
| 5 |            GROUP HASH INSTANT                    |              2
|
| 6 |              HASH JOIN (INNER JOIN)              |              2
```

```
|
| 7 |                TABLE ACCESS ("CUSTOMER" AS _A2)    |             6
|
| 8 |                HASH JOIN INSTANT                   |             2
|
| 9 |                  TABLE ACCESS ("ORDERS" AS _A1)    |             2
|
================================================================================
=

     1  -  TARGET : CUSTOMER.C_CUSTKEY, COUNT( ORDERS.O_ORDERKEY )
     2  -  SQL : SELECT /*+ USE_GROUP_HASH(10)
                            KEEP_JOINED_TABLE USE_HASH_IN( _A1, 100 )
                            FULL( _A2 ) FULL( _A1 )
                        */
                        "_A2"."C_CUSTKEY", COUNT( "_A1"."O_ORDERKEY" )
                   FROM ( "PUBLIC"."CUSTOMER"@LOCAL AS "_A2"
                          INNER JOIN
                          "PUBLIC"."ORDERS"@LOCAL AS "_A1"
                     ON "_A1"."O_CUSTKEY" = "_A2"."C_CUSTKEY") ALIAS "_A3"
                GROUP BY "_A2"."C_CUSTKEY"
           TARGET DOMAIN : G1(G1N1,G1N2) 2 rows,
                           G2(G2N1,G2N2) 2 rows,
                           G3(G3N1,G3N2) 2 rows
           RE-GROUPING
```

```
            GROUP KEY : CUSTOMER.C_CUSTKEY
            AGGREGATION : SUM( COUNT( ORDERS.O_ORDERKEY ) )
    4   -   TARGET : _A2.C_CUSTKEY, COUNT( _A1.O_ORDERKEY )
    5   -   GROUP KEY : _A2.C_CUSTKEY
            RECORD COLUMN : COUNT( _A1.O_ORDERKEY )
            READ KEY COLUMN : _A2.C_CUSTKEY
            READ RECORD COLUMN : COUNT( _A1.O_ORDERKEY )
    6   -   JOINED COLUMN : _A2.C_CUSTKEY, _A1.O_ORDERKEY
    7   -   CLONED
            READ COLUMN : _A2.C_CUSTKEY
    8   -   HASH KEY : _A1.O_CUSTKEY
            RECORD COLUMN : _A1.O_ORDERKEY
            READ KEY COLUMN : _A1.O_CUSTKEY, _A1.O_ORDERKEY
              HASH FILTER : _A1.O_CUSTKEY = _A2.C_CUSTKEY
    9   -   HASH SHARD ( # 3 )
            READ COLUMN : _A1.O_ORDERKEY, _A1.O_CUSTKEY


<<<  end print plan
```

Distinct

Local Distinct

在当前服务器执行distinct

以下为在cloned table的customer使用distinct的示例Cloned table的所有group的所有node拥有相同的数据因此在当前服务器执行distinct即可

```
\EXPLAIN PLAN
SELECT DISTINCT c_nationkey
  FROM customer;


>>>  start print plan


< Execution Plan >
================================================================================
|  IDX  |  NODE DESCRIPTION                                                    |
--------------------------------------------------------------------------------
|    0  |  SELECT STATEMENT                                                    |
|    1  |    QUERY BLOCK ("$QB_IDX_2")                                         |
|    2  |      GROUP                                                           |
|    3  |        INDEX ACCESS ("CUSTOMER", "CUSTOMER_NATIONKEY_FK")            |
================================================================================


     1  -  TARGET : CUSTOMER.C_NATIONKEY
     2  -  GROUP KEY : CUSTOMER.C_NATIONKEY
     3  -  CLONED
           READ INDEX COLUMN : CUSTOMER.C_NATIONKEY


<<<  end print plan
```

以下为在sharded table的orders使用distinct的示例Sharded table的数据以shard key为准分割因此要以local执行distinct时需要获取所有group的数据

```
\EXPLAIN PLAN
SELECT /*+ LOCAL_DISTINCT */
       DISTINCT o_orderstatus
  FROM orders
 WHERE o_orderdate = date '1995-03-15';


>>>  start print plan


< Execution Plan >
================================================================================
==
|IDX|  NODE DESCRIPTION                                |             ROWS
|
--------------------------------------------------------------------------------
--
| 0 |  SELECT STATEMENT                                |                3
|
| 1 |    QUERY BLOCK ("$QB_IDX_2")                     |                3
|
| 2 |      GROUP HASH INSTANT                          |                3
|
| 3 |        PLAN BASED CLUSTER                        | LOCAL/REMOTE 603
```

```
|
| 4 |          TABLE ACCESS ("ORDERS")                  |              221
|
=========================================================================
==


     1  -  TARGET : ORDERS.O_ORDERSTATUS
     2  -  GROUP KEY : ORDERS.O_ORDERSTATUS
           READ KEY COLUMN : ORDERS.O_ORDERSTATUS
     3  -  SQL : SELECT /*+ FULL( _A1 ) */
                        "_A1"."O_ORDERSTATUS"
                   FROM "PUBLIC"."ORDERS"@LOCAL AS "_A1"
                  WHERE "_A1"."O_ORDERDATE" = :_V0
           TARGET DOMAIN : G1(G1N1,G1N2) 221 rows,
                           G2(G2N1,G2N2) 184 rows,
                           G3(G3N1,G3N2) 198 rows
     4  -  HASH SHARD ( # 3 )
           READ COLUMN : ORDERS.O_ORDERSTATUS, ORDERS.O_ORDERDATE
             PHYSICAL FILTER : ORDERS.O_ORDERDATE = DATE'1995-03-15'


<<<  end print plan
```

从上述execution plan可以看到从G1G2G3获取满足条件的数据后执行了distinct的GROUP HASH INSTANT

Remote Distinct

在各个服务器处理distinct

在各个服务器处理distinct可获得并行处理的效果而且大体上由于distinct减少很多结果因此可减少获取数据的网络成本

```
\EXPLAIN PLAN
SELECT DISTINCT o_orderstatus
  FROM orders
 WHERE o_orderdate = date '1995-03-15';


>>>  start print plan


< Execution Plan >
================================================================================
==
|IDX|  NODE DESCRIPTION                                |           ROWS
|
--------------------------------------------------------------------------------
--
| 0 |  SELECT STATEMENT                                |              3
|
| 1 |    QUERY BLOCK ("$QB_IDX_2")                     |              3
|
| 2 |      SINGLE CLUSTER                              | LOCAL/REMOTE 3
```

```
|
| 3 |        SELECT STATEMENT                              |               3
|
| 4 |          QUERY BLOCK ("$QB_IDX_2")                   |               3
|
| 5 |            GROUP HASH INSTANT                        |               3
|
| 6 |              TABLE ACCESS ("ORDERS" AS _A1)          |             221
|
=============================================================================
==

     1  -  TARGET : ORDERS.O_ORDERSTATUS
     2  -  SQL : SELECT /*+ USE_DISTINCT_HASH(3) FULL( _A1 ) */
                        DISTINCT "_A1"."O_ORDERSTATUS"
                    FROM "PUBLIC"."ORDERS"@LOCAL AS "_A1"
                   WHERE "_A1"."O_ORDERDATE" = :_V0
           TARGET DOMAIN : G1(G1N1,G1N2) 3 rows,
                           G2(G2N1,G2N2) 3 rows,
                           G3(G3N1,G3N2) 3 rows
           RE-GROUPING
             GROUP KEY : ORDERS.O_ORDERSTATUS
     4  -  TARGET : _A1.O_ORDERSTATUS
     5  -  GROUP KEY : _A1.O_ORDERSTATUS
           READ KEY COLUMN : _A1.O_ORDERSTATUS
```

```
    6  -  HASH SHARD ( # 3 )
          READ COLUMN : _A1.O_ORDERSTATUS, _A1.O_ORDERDATE
            PHYSICAL FILTER : _A1.O_ORDERDATE = :_V0

<<<  end print plan
```

从上述execution plan可看到从G1G2G3获取执行distinct的结果后再次以RE-GROUPING进行了distinct即使执行RE-GROUPING也由于减少了很多结果因此可减少网络成本提高效率

Order By

Local Order By

在当前服务器执行order by

以下为在cloned table的customer使用order by的示例Cloned table的所有group的所有node拥有相同的数据因此在当前服务器执行order by即可

```
\EXPLAIN PLAN
  SELECT c_nationkey, COUNT(c_custkey)
    FROM customer
GROUP BY c_nationkey
ORDER BY c_nationkey;

>>>  start print plan
```

```
< Execution Plan >
================================================================================
=
| IDX |  NODE DESCRIPTION                                          |   ROWS
|
--------------------------------------------------------------------------------
-
|  0  |  SELECT STATEMENT                                          |     25
|
|  1  |    QUERY BLOCK ("$QB_IDX_2")                               |     25
|
|  2  |      SORT INSTANT                                          |     25
|
|  3  |        GROUP HASH INSTANT                                  |     25
|
|  4  |          TABLE ACCESS ("CUSTOMER")                         | 150000
|
================================================================================
=


     1  -  TARGET : CUSTOMER.C_NATIONKEY, COUNT( CUSTOMER.C_CUSTKEY )
     2  -  SORT KEY : "CUSTOMER.C_NATIONKEY ASC NULLS LAST"
           RECORD COLUMN : COUNT( CUSTOMER.C_CUSTKEY )
           READ KEY COLUMN : CUSTOMER.C_NATIONKEY
           READ RECORD COLUMN : COUNT( CUSTOMER.C_CUSTKEY )
```

```
     3  -  GROUP KEY : CUSTOMER.C_NATIONKEY
           RECORD COLUMN : COUNT( CUSTOMER.C_CUSTKEY )
           READ KEY COLUMN : CUSTOMER.C_NATIONKEY
           READ RECORD COLUMN : COUNT( CUSTOMER.C_CUSTKEY )
     4  -  CLONED
           READ COLUMN : CUSTOMER.C_CUSTKEY, CUSTOMER.C_NATIONKEY

<<<  end print plan
```

以下为在sharded table的orders使用order by的示例Sharded table的数据以shard key为准分割因此在本地执行order by需要获取所有group的数据

```
\EXPLAIN PLAN
  SELECT /*+ LOCAL_ORDER */ o_orderdate, o_shippriority
    FROM orders
   WHERE o_orderdate >= date '1993-07-01'
     AND o_orderdate < date '1993-07-01' + interval '1' month
ORDER BY o_orderdate;

>>>  start print plan

< Execution Plan >
================================================================================
==
|IDX|  NODE DESCRIPTION                                |               ROWS
|
```

```
-------------------------------------------------------------------------
--
| 0 |  SELECT STATEMENT                          |              19319
|
| 1 |    QUERY BLOCK ("$QB_IDX_2")               |              19319
|
| 2 |      SORT INSTANT                          |              19319
|
| 3 |        PLAN BASED CLUSTER                  | LOCAL/REMOTE 19319
|
| 4 |          TABLE ACCESS ("ORDERS")           |               6453
|
=========================================================================
==

     1  -  TARGET : ORDERS.O_ORDERDATE, ORDERS.O_SHIPPRIORITY
     2  -  SORT KEY : "ORDERS.O_ORDERDATE ASC NULLS LAST"
           RECORD COLUMN : ORDERS.O_SHIPPRIORITY
           READ KEY COLUMN : ORDERS.O_ORDERDATE
           READ RECORD COLUMN : ORDERS.O_SHIPPRIORITY
     3  -  SQL : SELECT /*+ FULL( _A1 ) */
                       "_A1"."O_ORDERDATE", "_A1"."O_SHIPPRIORITY"
                  FROM "PUBLIC"."ORDERS"@LOCAL AS "_A1"
                 WHERE "_A1"."O_ORDERDATE" < :_V0
                   AND "_A1"."O_ORDERDATE" >= :_V1
```

```
           TARGET DOMAIN : G1(G1N1,G1N2) 6453 rows,
                           G2(G2N1,G2N2) 6450 rows,
                           G3(G3N1,G3N2) 6416 rows
     4   -  HASH SHARD ( # 3 )
           READ COLUMN : ORDERS.O_ORDERDATE, ORDERS.O_SHIPPRIORITY
             PHYSICAL FILTER : ORDERS.O_ORDERDATE < DATE'1993-07-01' +
CAST( '1' AS INTERVAL(MONTH) ) AND ORDERS.O_ORDERDATE >= DATE'1993-07-01'


<<<  end print plan
```

从上述execution plan可看到从G1G2G3获取满足条件的数据后执行了order by的SORT INSTANT

Remote Order By

在各个服务器处理order by

在各个服务器处理order by可获得并行处理的效果而且从下级节点排列结果后到上级节点时各个服务器不需要为了order by进行额外的处理driver可合并从各个服务器获取的结果并sort即可

以下为remote order by的示例orders为sharded table

```
\EXPLAIN PLAN
  SELECT o_custkey, o_orderstatus
    FROM orders
   WHERE o_custkey > 0 AND o_custkey < 10
ORDER BY o_custkey;
```

```
>>>  start print plan

< Execution Plan >
================================================================================
==
|IDX| NODE DESCRIPTION                                                  | ROWS
|
--------------------------------------------------------------------------------
--
| 0 | SELECT STATEMENT                                                  |   66
|
| 1 |   QUERY BLOCK ("$QB_IDX_2")                                       |   66
|
| 2 |     MULTIPLE CLUSTER                                  | LOCAL/REMOTE 66
|
| 3 |       SELECT STATEMENT                                            |   24
|
| 4 |         QUERY BLOCK ("$QB_IDX_2")                                 |   24
|
| 5 |           INDEX ACCESS ("ORDERS" AS _A1, "ORDERS_CUSTKEY_FK") |   24
|
================================================================================
==
```

```
     1  -  TARGET : ORDERS.O_CUSTKEY, ORDERS.O_ORDERSTATUS
     2  -  SQL : SELECT /*+ INDEX( _A1, "PUBLIC"."ORDERS_CUSTKEY_FK" ) */
                        "_A1"."O_CUSTKEY", "_A1"."O_ORDERSTATUS"
                   FROM "PUBLIC"."ORDERS"@LOCAL AS "_A1"
                  WHERE "_A1"."O_CUSTKEY" > :_V0
                    AND "_A1"."O_CUSTKEY" < :_V1
                ORDER BY "_A1"."O_CUSTKEY" ASC NULLS LAST
           TARGET DOMAIN : G1(G1N1,G1N2) 24 rows,
                           G2(G2N1,G2N2) 23 rows,
                           G3(G3N1,G3N2) 19 rows
           MERGE SORTING
             SORT KEY : ORDERS.O_CUSTKEY
     4  -  TARGET : _A1.O_CUSTKEY, _A1.O_ORDERSTATUS
     5  -  HASH SHARD ( # 3 )
           READ INDEX COLUMN : _A1.O_CUSTKEY
           READ TABLE COLUMN : _A1.O_ORDERSTATUS
             MIN RANGE : _A1.O_CUSTKEY > :_V0
             MAX RANGE : _A1.O_CUSTKEY IS NOT NULL AND _A1.O_CUSTKEY
< :_V1


<<<  end print plan
```

从以上execution plan可以看到从G1G2G3获取ordering的数据后合并数据的同时对order by key col进行了sorting

Aggregation

Local Aggregation

在当前服务器执行aggregation

以下为local aggregation的示例customer为cloned table

```
\EXPLAIN PLAN
SELECT COUNT(DISTINCT c_nationkey)
  FROM customer;


>>>  start print plan


< Execution Plan >
================================================================================
==
|IDX|  NODE DESCRIPTION                                                |   ROWS
|
--------------------------------------------------------------------------------
--
| 0 |  SELECT STATEMENT                                                |      1
|
| 1 |    QUERY BLOCK ("$QB_IDX_2")                                     |      1
|
| 2 |      AGGREGATION BY HASH                                         |      1
```

```
|
| 3 |        INDEX ACCESS ("CUSTOMER", "CUSTOMER_NATIONKEY_FK")    | 150000
|
=========================================================================
==


     1  -  TARGET : COUNT( DISTINCT CUSTOMER.C_NATIONKEY )
     2  -  DISTINCT AGGREGATION : COUNT( DISTINCT CUSTOMER.C_NATIONKEY )
     3  -  CLONED
           READ INDEX COLUMN : CUSTOMER.C_NATIONKEY


<<<  end print plan
```

Remote Aggregation

在各个服务器处理aggregation

在各个服务器处理aggregation可获得并行处理的效果而且大体上由aggregation减少结果因此可减少获取数据的成本

```
\EXPLAIN PLAN
SELECT sum(ps_supplycost * ps_availqty) * 0.0001
  FROM partsupp,
       supplier,
       nation
 WHERE ps_suppkey = s_suppkey
```

```
   AND s_nationkey = n_nationkey
   AND n_name = 'GERMANY';


>>>  start print plan


< Execution Plan >
================================================================================
=
|IDX|  NODE DESCRIPTION                                          |  ROWS
|
--------------------------------------------------------------------------------
-
| 0 |  SELECT STATEMENT                                          |     1
|
| 1 |    QUERY BLOCK ("$QB_IDX_2")                               |     1
|
| 2 |      SINGLE CLUSTER                                | LOCAL/REMOTE  1
|
| 3 |        SELECT STATEMENT                                    |     1
|
| 4 |          QUERY BLOCK ("$QB_IDX_2")                         |     1
|
| 5 |            AGGREGATION BY HASH                             |     1
|
| 6 |              NESTED JOIN (INNER JOIN)                      | 10508
```

```
|
| 7 |                NESTED JOIN (INNER JOIN)                    |   396
|
| 8 |                  TABLE ACCESS ("NATION" AS _A3)            |     1
|
| 9 |               INDEX ACCESS ("SUPPLIER" AS _A2)             |   396
|
| 10 |                INDEX ACCESS ("PARTSUPP" AS _A1)           | 10508
|
================================================================================
=

     1  -  TARGET : SUM( PARTSUPP.PS_SUPPLYCOST * PARTSUPP.PS_AVAILQTY ) *
0.0001
     2  -  SQL : SELECT /*+ KEEP_JOINED_TABLE USE_NL_IN( _A1 )
                            USE_NL_IN( _A2 )
                            FULL( _A3 )
                            INDEX( _A2, "PUBLIC"."SUPPLIER_NATIONKEY_FK" )
                            INDEX( _A1, "PUBLIC"."PARTSUPP_SUPPKEY_FK" )
                        */
                        SUM( "_A1"."PS_SUPPLYCOST" *
"_A1"."PS_AVAILQTY" )
                    FROM ( ( "PUBLIC"."NATION"@LOCAL AS "_A3"
                             INNER JOIN
                            "PUBLIC"."SUPPLIER"@LOCAL AS "_A2"
```

```
                              ON true
                            ) ALIAS "_A4"
                           INNER JOIN
                           "PUBLIC"."PARTSUPP"@LOCAL AS "_A1" ON true
                         ) ALIAS "_A5"
                  WHERE "_A3"."N_NAME" = :_V0
                    AND "_A2"."S_NATIONKEY" = "_A3"."N_NATIONKEY"
                    AND "_A1"."PS_SUPPKEY" = "_A2"."S_SUPPKEY"
            TARGET DOMAIN : G1(G1N1,G1N2) 1 rows,
                            G2(G2N1,G2N2) 1 rows,
                            G3(G3N1,G3N2) 1 rows
            RE-AGGREGATION
              AGGREGATION : SUM( SUM( PARTSUPP.PS_SUPPLYCOST *
PARTSUPP.PS_AVAILQTY ) )
     4  -  TARGET : SUM( _A1.PS_SUPPLYCOST * _A1.PS_AVAILQTY )
     5  -  AGGREGATION : SUM( _A1.PS_SUPPLYCOST * _A1.PS_AVAILQTY )
     6  -  JOINED COLUMN : _A1.PS_SUPPLYCOST, _A1.PS_AVAILQTY
              CONSTANT FILTER : TRUE
     7  -  JOINED COLUMN : _A2.S_SUPPKEY
              CONSTANT FILTER : TRUE
     8  -  CLONED
            READ COLUMN : _A3.N_NATIONKEY, _A3.N_NAME
              PHYSICAL FILTER : _A3.N_NAME = :_V0
     9  -  CLONED
            READ INDEX COLUMN : _A2.S_NATIONKEY
```

```
            READ TABLE COLUMN : _A2.S_SUPPKEY
              MIN RANGE : _A2.S_NATIONKEY = {_A3.N_NATIONKEY}
              MAX RANGE : _A2.S_NATIONKEY = {_A3.N_NATIONKEY}
    10  -   HASH SHARD ( # 3 )
            READ INDEX COLUMN : _A1.PS_SUPPKEY
            READ TABLE COLUMN : _A1.PS_AVAILQTY, _A1.PS_SUPPLYCOST
              MIN RANGE : _A1.PS_SUPPKEY = {_A2.S_SUPPKEY}
              MAX RANGE : _A1.PS_SUPPKEY = {_A2.S_SUPPKEY}


<<<  end print plan
```

从上述execution plan可以看到从G1G2G3获取数据后进行re-aggregation后返回了结果

5.5 统计信息

Query optimizer使用统计信息计算成本Query optimizer使用的统计信息有表统计信息和column统计信息索引统计信息等

- 表统计信息
  - Row数量
  - 页面数量
- Column统计信息
  - 互不相同的值的数量
  - NULL值的数量
  - 值的平均长度
  - 最小值
  - 最大值
- 索引统计信息
  - 互不相同的key的数量
  - 页面数量
  - Leaf页面数量
  - Tree level
  - 索引集群因子(clustering factor)

要构建统计信息时执行 **ANALYZE TABLE**构建的统计信息存储到数据库并在重新构建统计信息之前使用相同的统计信息

未构建统计信息的表使用catalog信息和查询执行时间点的page信息构建简单的统计信息后使用

Optimizer调整

通常query optimizer使用统计信息选择最高效的plan但是也会有比query optimizer选择的plan更佳的planquery optimizer未选择该plan时用户可指定使用该plan

目前SUNDB的query optimize提供hint描述可应用的hint时与计算的cost无关优先应用用户描述的hint因此有更好的plan时用户可使用hint强行变更plan

Hint相关详细内容参考 **SQL Hint**

5.6 SQL Hint

说明

Hint是用于用户直接向SUNDB optimizer指示SQL语句执行方法的comment由于统计信息不准确而SUNDB optimizer无法准确判断执行计划时用户可使用hint直接选择执行计划

SUNDB optimizer决定执行计划时最优先选择用户描述的hint如果无法使用用户描述的hint时由SUNDB optimizer判断决定

用户描述hint时SUNDB optimizer尽量按照hint运行因此建议仅在认为SUNDB optimizer的SQL语句执行计划有误时使用

尤其是反复使用相同的SQL语句时使用hintSUNDB optimizer则按照用户描述的hint执行SQL语句此时由于包含在该SQL语句的表或view的数据发生变更会降低其他执行计划的性能因此需注意

SUNDB中hint可在SELECT, INSERT SELECT, UPDATE, DELETE等语句使用Hint在各个语句的关键字后面的位置在两边使用/*+关键字和*/关键字并在其中间描述

语句

Hint语句为如下

```
<hint clause> ::=

    /*+ <hint element> [ comment ] [ [ , ] <hint element> [ comment ] ] */
```

```
<hint element> ::=
      <statement hints>
    | <query block hints>
    | <operation hints>

<statement hints> ::=
      <dml hints >
    | <fetch fail over hints>

<dml hints> ::=
      DML_GLOBAL_ROWID

<fetch fail over hints> ::=
      FETCH_FAIL_OVER

<query block hints> ::=
      <push subquery hints>
    | <cte query hints>
    | <query transformation hints>

<push subquery hints> ::=
      PUSH_SUBQ
    | NO_PUSH_SUBQ
```

```
<cte query hints> ::=
      INLINE
    | MATERIALIZE

<query transformation hints> ::=
      NO_QUERY_TRANSFORMATION
    | <unnest subquery hints>
    | <transitive closure hints>
    | <view hints>

<unnest subquery hints> ::=
      <unnest hints>
    | <unnest join operation hints>
    | <unnest join driver hints>
    | <unnest join pusher hints>
    | <unnest merge hints>

<unnest hints> ::=
      UNNEST
    | NO_UNNEST

<unnest join operation hints> ::=
      UNNEST_NL
    | UNNEST_NL_IN
    | UNNEST_NL_OUT
```

```
    | UNNEST_INL
    | UNNEST_INL_IN
    | UNNEST_INL_OUT
    | UNNEST_HASH
    | UNNEST_HASH( hash_bucket_count )
    | UNNEST_HASH_IN
    | UNNEST_HASH_IN( hash_bucket_count )
    | UNNEST_HASH_OUT
    | UNNEST_HASH_OUT( hash_bucket_count )
    | UNNEST_MERGE
    | UNNEST_MERGE_IN
    | UNNEST_MERGE_OUT
    | NL_SJ
    | NL_ISJ
    | NL_AJ
    | INL_SJ
    | INL_AJ
    | MERGE_SJ
    | MERGE_AJ
    | HASH_SJ
    | HASH_ISJ
    | HASH_AJ


<unnest join driver hints> ::=
    | LOCAL_UNNEST
```

```
    | REMOTE_UNNEST


<unnest join pusher hints> ::=

      PUSHER_SUBQ

    | NO_PUSHER_SUBQ

    | PUSHER_OUTQ

    | NO_PUSHER_OUTQ


<unnest merge hints> ::=

      MERGE_SUBQ

    | NO_MERGE_SUBQ


<transitive closure hints> ::=

    | TRANSITIVE_CLOSURE

    | NO_TRANSITIVE_CLOSURE


< view hints > ::=

      <view merge hints>

    | <push view predicate hints>


<view merge hints> ::=

    | MERGE( view_name )

    | NO_MERGE( view_name )


<push view predicate hints> ::=
```

```
      PUSH_PRED
    | NO_PUSH_PRED
    | PUSH_PRED( view_name[ [ , ] table_name ] )
    | NO_PUSH_PRED( view_name[ [ , ] table_name ] )



<operation hints> ::=
      <access path hints>
    | <join hints>
    | <group hints>
    | <distinct hints>
    | <order hints>
    | <aggr hints>
    | <rownum hints>


<access path hints> ::=
      FULL( table_name )
    | INDEX( table_name [ , ] [ index_name [ [ , ] index_name ] ] )
    | NO_INDEX( table_name [ , ] [ index_name [ [ , ] index_name ] ] )
    | INDEX_FORWARD( table_name [ , ] [ index_name [ [ , ]
index_name ] ] )
    | INDEX_BACKWARD( table_name [ , ] [ index_name [ [ , ]
index_name ] ] )
    | INDEX_ASC( table_name [ , ] [ index_name [ [ , ] index_name ] ] )
    | INDEX_DESC( table_name [ , ] [ index_name [ [ , ] index_name ] ] )
```

```
    | INDEX_COMBINE( table_name [ , ] [ index_name [ [ , ]
index_name ] ] )
    | IN_KEY_RANGE( table_name [ , ] [ index_name [ [ , ] index_name ] ] )
    | ROWID( table_name )



<join hints> ::=
    < join order hints >
    | < join operation hints >
    | < join driver hints >
    | < join pusher hints >

<join order hints> ::=
      ORDERED
    | ORDERING( table_name [LEFT | RIGHT] [ , table_name [LEFT | RIGHT] ]
    | LEADING( table_name [ [ , ] table_name ] )
    | KEEP_JOINED_TABLE

<join operation hints> ::=
      USE_HASH( table_name [ [ , ] table_name ] )
    | USE_HASH( table_name [ [ , ] table_name ], hash_bucket_count )
    | NO_USE_HASH( table_name [ [ , ] table_name ] )
    | USE_MERGE( table_name [ [ , ] table_name ] )
    | NO_USE_MERGE( table_name [ [ , ] table_name ] )
    | USE_NL( table_name [ [ , ] table_name ] )
```

```
    | NO_USE_NL( table_name [ [ , ] table_name ] )
    | USE_INL( table_name [ [ , ] table_name ] )
    | NO_USE_INL( table_name [ [ , ] table_name ] )
    | USE_JOIN_COMBINE( alias )
    | NO_USE_JOIN_COMBINE( alias )
    | USE_NL_IN( alias )
    | USE_NL_OUT( alias )
    | USE_INL_IN( alias )
    | USE_INL_OUT( alias )
    | USE_HASH_IN( alias )
    | USE_HASH_IN( alias, hash_bucket_count )
    | USE_HASH_OUT( alias )
    | USE_HASH_OUT( alias, hash_bucket_count )
    | USE_MERGE_IN( alias )
    | USE_MERGE_OUT( alias )


<join driver hints> ::=
    | LOCAL_JOIN( alias )
    | REMOTE_JOIN( alias )


<join pusher hints> ::=
      PUSHER( alias )
    | NO_PUSHER( alias )
```

```
<group hints> ::=
      <group operation hints>
    | <group driver hints>

<group operation hints> ::=
      USE_GROUP_HASH
    | USE_GROUP_HASH( hash_bucket_count )

<group driver hints> ::=
      LOCAL_GROUP
    | REMOTE_GROUP




<distinct hints> ::=
      <distinct operation hints>
    | <distinct driver hints>

<distinct operation hints> ::=
      DISTINCT_HASH
    | USE_DISTINCT_HASH( hash_bucket_count )

<distinct driver hints> ::=
      LOCAL_DISTINCT
    | REMOTE_DISTINCT
```

```
<order hints> ::=
      <order operation hints>
    | <order driver hints>

<order operation hints> ::=
      USE_ORDER_SORT
    | NO_USE_ORDER_SORT
    | USE_ORDER_LIMIT_SORT
    | NO_USE_ORDER_LIMIT_SORT

<order driver hints> ::=
      LOCAL_ORDER
    | REMOTE_ORDER



<aggr hints> ::=
      <aggr driver hint>

<aggr driver hints> ::=
      LOCAL_AGGR
    | REMOTE_AGGR
```

使用范围及访问权限

执行<hint clause>语句时用户需要有执行query的权限

语句规则及参数

使用<hint clause>的默认语句规则为如下

- <hint clause>中可使用空白或comma (,)描述多个<hint element>
- 有两个以上的<hint element>并无法同时应用时仅适用最先描述的<hint element>
- <hint element>的语句发生错误时默认忽略该<hint element>将hint_error property设置为on时处理为<hint clause>的validation error
- <hint clause>中描述的table_nameview_name应与<from clause>中描述的table_nameview_name或指向其的alias中的一个一致
- 描述table_name时无法同时描述schema name
- 即使<hint element>正确但无法适用时忽略该<hint element>

使用示例

以下为在SELECT,INSERT SELECT, UPDATE, DELETE语句使用<hint clause>的示例

- 在SELECT语句使用的情况

```
SELECT /*+ INDEX(orders, o_orderdate_idx) */ *
  FROM orders
 WHERE o_orderdate < date '2019-04-12';
```

- 在INSERT SELECT语句使用的情况

```
INSERT INTO orders_bk SELECT /*+ INDEX(orders, o_orderdate_idx) */ *
                        FROM orders
                       WHERE o_orderdate < date '2019-04-12';
```

- 在UPDATE语句使用的情况

```
UPDATE /*+ INDEX(lineitem, l_shipdate_idx) */ * lineitem
   SET l_receiptdate = CURRENT_DATE
 WHERE l_shipdate = date '2020-04-12';
```

- 在DELETE语句使用的情况

```
DELETE /*+ INDEX(lineitem, l_shipdate_idx) */ * lineitem
 WHERE l_receiptdate < date '2020-04-12';
```

以下为使用ALTER语句将HINT_ERROR property设置为ON后错误使用hint的示例

```
ALTER SESSION SET HINT_ERROR = ON;


SELECT /*+ INDEX(orders, o_orderdate_idxx) */ *
  FROM orders
 WHERE o_orderdate < date '2019-04-12';


ERR-42000(16058): not applicable hint ( cannot find index ) :
SELECT /*+ INDEX(orders, o_orderdate_idxx) */ *
```

```
                         *

ERROR at line 1:
```

所有hint相关说明与示例参考 **Query Block Hint**和 **Operation Hint**

Statement Hint

是以statement为单位应用的hint

<dml hints>

DML_GLOBAL_ROWID

在集群环境处理DML语句时接收user查询的node向其他node发送记录变更信息并反映

在集群环境中DML语句按照以下两种方式处理

- 基于query的DML: 构成新的查询并发送变更语句
- 基于global rowid的DML: 使用记录标识符信息发送特定记录的变更信息

描述DML_GLOBAL_ROWID hint时优先应用基于global rowid的DML方式

此hint适用于SELECT FOR UPDATE, INSERT, UPDATE, DELETE等语句

以下为应用基于query的DML的示例

```
\EXPLAIN PLAN
```

```
DELETE FROM orders;



>>>  start print plan


< Execution Plan >

=============================================================================

|  IDX  |  NODE DESCRIPTION                                                  |

-----------------------------------------------------------------------------

|    0  |  DELETE STATEMENT ("ORDERS")                                       |

|    1  |    QUERY BLOCK ("$QB_IDX_2")                                       |

|    2  |      DML CLUSTER                                                    |

|    3  |        TABLE ACCESS ("ORDERS")                                     |

=============================================================================


     1  -  TARGET : NOTHING

     2  -  WITHOUT FETCH

           Non-Fetch SQL : DELETE /*+ FULL( _A1 ) */  "_A1" FROM
"PUBLIC"."ORDERS"@LOCAL AS "_A1"

           TARGET DOMAIN : G1(G1N1,G1N2) 0 rows, G2(G2N1,G2N2) 0 rows,
G3(G3N1,G3N2) 0 rows

     3  -  HASH SHARD ( # 3 )

           READ COLUMN : ORDERS.O_ORDERKEY, ORDERS.O_CUSTKEY


<<<  end print plan
```

以下为使用DML_GLOBAL_ROWID hint的示例

```
\EXPLAIN PLAN
DELETE /*+ DML_GLOBAL_ROWID */ FROM orders;



>>>  start print plan


< Execution Plan >
=============================================================================
|  IDX  |  NODE DESCRIPTION                                                 |
-----------------------------------------------------------------------------
|    0  |  DELETE STATEMENT ("ORDERS")                                      |
|    1  |    QUERY BLOCK ("$QB_IDX_2")                                      |
|    2  |      PLAN BASED CLUSTER                                           |
|    3  |        TABLE ACCESS ("ORDERS")                                    |
=============================================================================


     1  -  TARGET : NOTHING
     2  -  SQL : SELECT /*+ FULL( _A1 ) */ "_A1"."$PHYSICAL_ROWID",
"_A1"."O_ORDERKEY", "_A1"."O_CUSTKEY" FROM "PUBLIC"."ORDERS"@LOCAL AS
"_A1"
           TARGET DOMAIN : G1(G1N1,G1N2) 0 rows, G2(G2N1,G2N2) 0 rows,
G3(G3N1,G3N2) 0 rows
     3  -  HASH SHARD ( # 3 )
```

```
            READ COLUMN : ORDERS.O_ORDERKEY, ORDERS.O_CUSTKEY


<<<  end print plan
```

<fetch fail over hints>

FETCH_FAIL_OVER

Fetch fail over功能考虑到在集群环境中对SELECT语句执行fetch的过程中发生通信障碍的情况而提供完整的fetch结果

使用FETCH_FAIL_OVER hint可激活fetch fail over功能

此hint适用于SELECT语句

以下为使用FETCH_FAIL_OVER hint的示例

```
\EXPLAIN PLAN
SELECT /*+ FETCH_FAIL_OVER */ o_orderkey FROM orders;



>>>  start print plan


< Execution Plan >
=============================================================================
|  IDX  |  NODE DESCRIPTION                                                 |
-----------------------------------------------------------------------------
|    0  |  SELECT STATEMENT                                                 |
```

```
|    1  |    QUERY BLOCK ("$QB_IDX_2")                                        |
|    2  |      PLAN BASED CLUSTER                                              |
|    3  |        INDEX ACCESS ("ORDERS", "ORDERS_PK_INDEX")                    |
========================================================================


     0  -  FETCH FAIL OVER
     1  -  TARGET : ORDERS.O_ORDERKEY
     2  -  SQL : SELECT /*+ INDEX( _A1, "PUBLIC"."ORDERS_PK_INDEX" ) */
"_A1"."O_ORDERKEY" FROM "PUBLIC"."ORDERS"@LOCAL AS "_A1"
           TARGET DOMAIN : G1(G1N1,G1N2) 0 rows, G2(G2N1,G2N2) 0 rows,
G3(G3N1,G3N2) 0 rows
     3  -  HASH SHARD ( # 3 )
           READ INDEX COLUMN : ORDERS.O_ORDERKEY


<<<  end print plan
```

Query Block Hint

是以query block为单位适用的hint

<push subquery hints>

PUSH_SUBQ

描述PUSH_SUBQ hint时optimizer将该子查询push到可处理的执行计划节点中最下级的节点是未

unnest的子查询的hint因此有优先描述的<unnest subquery hints>时忽略PUSH_SUBQ hint

未描述PUSH_SUBQ hint时optimizer计算cost后将子查询push到cost最佳节点

使用PUSH_SUBQ hint时子查询在未unnest的情况下迅速适用因此子查询的filtering效果大并且仅通过一次子查询执行也可存储其中间结果时可提高性能

以下为在这种情况使用PUSH_SUBQ hint的示例

```
\EXPLAIN PLAN
SELECT
       c_name,
       o_totalprice
  FROM
       customer,
       orders,
       lineitem
 WHERE c_custkey = o_custkey
   AND o_orderkey = l_orderkey
   AND o_orderkey IN (
                       SELECT   /*+ PUSH_SUBQ */
                                l_orderkey
                         FROM   lineitem
                       GROUP BY l_orderkey
                       HAVING   sum(l_quantity) > 300
                     );
```

```
>>>  start print plan


< Execution Plan >

================================================================================
|  IDX  |  NODE DESCRIPTION                                                    |
--------------------------------------------------------------------------------
|    0  |  SELECT STATEMENT                                                    |
|    1  |    QUERY BLOCK ("$QB_IDX_2")                                         |
|    2  |      NESTED JOIN (INNER JOIN)                                        |
|    3  |        MERGE JOIN (INNER JOIN)                                       |
|    4  |          INDEX ACCESS ("ORDERS", "ORDERS_PK_INDEX")                  |
|    5  |          INDEX ACCESS ("LINEITEM", "LINEITEM_PK_INDEX")              |
|    6  |        INDEX ACCESS ("CUSTOMER", "CUSTOMER_PK_INDEX")                |
|    7  |  SUB QUERY LIST                                                      |
|    8  |    INLINE_VIEW ("$V8") (MATERIALIZED)                                |
|    9  |      QUERY BLOCK ("$QB_IDX_10")                                      |
|   10  |        GROUP HASH INSTANT                                            |
|   11  |          TABLE ACCESS ("LINEITEM")                                   |
================================================================================


     1  -  TARGET : CUSTOMER.C_NAME, ORDERS.O_TOTALPRICE

     2  -  JOINED COLUMN : CUSTOMER.C_NAME, ORDERS.O_TOTALPRICE

     3  -  JOINED COLUMN : ORDERS.O_CUSTKEY, ORDERS.O_TOTALPRICE

             ON FILTER (Equi) : ORDERS.O_ORDERKEY = LINEITEM.L_ORDERKEY
```

```
     4  -  READ INDEX COLUMN : ORDERS.O_ORDERKEY
           READ TABLE COLUMN : ORDERS.O_CUSTKEY, ORDERS.O_TOTALPRICE
             POST FILTER : ( ORDERS.O_ORDERKEY ) IN ( $V8.L_ORDERKEY )
     5  -  READ INDEX COLUMN : LINEITEM.L_ORDERKEY
             MIN RANGE : LINEITEM.L_ORDERKEY >= {ORDERS.O_ORDERKEY}
             MAX RANGE : LINEITEM.L_ORDERKEY IS NOT NULL
     6  -  READ INDEX COLUMN : CUSTOMER.C_CUSTKEY
           READ TABLE COLUMN : CUSTOMER.C_NAME
             MIN RANGE : CUSTOMER.C_CUSTKEY = {ORDERS.O_CUSTKEY}
             MAX RANGE : CUSTOMER.C_CUSTKEY = {ORDERS.O_CUSTKEY}
           FETCH ONE ROW
     8  -  COLUMN : LINEITEM.L_ORDERKEY AS L_ORDERKEY
     9  -  TARGET : LINEITEM.L_ORDERKEY
    10  -  GROUP KEY : LINEITEM.L_ORDERKEY
           RECORD COLUMN : SUM( LINEITEM.L_QUANTITY )
           READ KEY COLUMN : LINEITEM.L_ORDERKEY
           READ RECORD COLUMN : SUM( LINEITEM.L_QUANTITY )
             PHYSICAL FILTER : SUM( LINEITEM.L_QUANTITY ) > 300
    11  -  READ COLUMN : LINEITEM.L_ORDERKEY, LINEITEM.L_QUANTITY


<<<  end print plan
```

在上述示例中可看到子查询可在join或order中执行在可执行节点中的最下级节点*INDEX ACCESS ("ORDERS")*中执行并未unnest

NO_PUSH_SUBQ

描述NO_PUSH_SUBQ时optimizer不push子查询因此在可处理的执行计划节点中最上级节点执行子查询由于是未unnest的子查询的hint因此有优先描述的<unnest subquery hints>时忽略NO_PUSH_SUBQ hint

未描述NO_PUSH_SUBQ时optimizer计算cost后向cost最好的节点push子查询

使用NO_PUSH_SUBQ hint时子查询在未unnest的情况下尽可能晚地应用子查询的filtering效果小并反复执行子查询时在由join减少中间结果之前应用子查询则由于反复执行子查询而降低性能这种情况最好使用NO_PUSH_SUBQ hint最大程度上推迟应用子查询的时间

以下为在这种情况使用NO_PUSH_SUBQ hint的示例

```
\EXPLAIN PLAN
SELECT
       c_name,
       o_totalprice
  FROM
       customer,
       orders,
       lineitem
 WHERE c_custkey = o_custkey
   AND o_orderkey = l_orderkey
   AND o_orderkey IN (
                      SELECT   /*+ NO_PUSH_SUBQ */
                               l_orderkey
```

```
                      FROM  lineitem
                     WHERE  o_orderdate = l_shipdate
                   );


>>>  start print plan


< Execution Plan >
=============================================================================
|  IDX  |  NODE DESCRIPTION                                                  |
-----------------------------------------------------------------------------
|    0  |  SELECT STATEMENT                                                  |
|    1  |    QUERY BLOCK ("$QB_IDX_2")                                       |
|    2  |      NESTED JOIN (INNER JOIN)                                      |
|    3  |        MERGE JOIN (INNER JOIN)                                     |
|    4  |          INDEX ACCESS ("ORDERS", "ORDERS_PK_INDEX")                |
|    5  |          INDEX ACCESS ("LINEITEM", "LINEITEM_PK_INDEX")            |
|    6  |        INDEX ACCESS ("CUSTOMER", "CUSTOMER_PK_INDEX")              |
|    7  |      SUB QUERY LIST                                                |
|    8  |        INLINE_VIEW ("$V8")                                         |
|    9  |          QUERY BLOCK ("$QB_IDX_10")                                |
|   10  |            TABLE ACCESS ("LINEITEM")                               |
=============================================================================


     1  -  TARGET : CUSTOMER.C_NAME, ORDERS.O_TOTALPRICE
     2  -  JOINED COLUMN : ORDERS.O_ORDERDATE, ORDERS.O_ORDERKEY,
```

```
CUSTOMER.C_NAME, ORDERS.O_TOTALPRICE
              POST WHERE FILTER : ( ORDERS.O_ORDERKEY ) IN ( $V8.L_ORDERKEY )
     3  -  JOINED COLUMN : ORDERS.O_CUSTKEY, ORDERS.O_ORDERDATE,
ORDERS.O_ORDERKEY, ORDERS.O_TOTALPRICE
              ON FILTER (Equi) : ORDERS.O_ORDERKEY = LINEITEM.L_ORDERKEY
     4  -  READ INDEX COLUMN : ORDERS.O_ORDERKEY
           READ TABLE COLUMN : ORDERS.O_CUSTKEY, ORDERS.O_TOTALPRICE,
ORDERS.O_ORDERDATE
     5  -  READ INDEX COLUMN : LINEITEM.L_ORDERKEY
              MIN RANGE : LINEITEM.L_ORDERKEY >= {ORDERS.O_ORDERKEY}
              MAX RANGE : LINEITEM.L_ORDERKEY IS NOT NULL
     6  -  READ INDEX COLUMN : CUSTOMER.C_CUSTKEY
           READ TABLE COLUMN : CUSTOMER.C_NAME
              MIN RANGE : CUSTOMER.C_CUSTKEY = {ORDERS.O_CUSTKEY}
              MAX RANGE : CUSTOMER.C_CUSTKEY = {ORDERS.O_CUSTKEY}
           FETCH ONE ROW
     8  -  COLUMN : LINEITEM.L_ORDERKEY AS L_ORDERKEY
     9  -  TARGET : LINEITEM.L_ORDERKEY
    10  -  READ COLUMN : LINEITEM.L_ORDERKEY, LINEITEM.L_SHIPDATE
              PHYSICAL FILTER : LINEITEM.L_SHIPDATE = {ORDERS.O_ORDERDATE}
```

上述示例所示可在nested joinmerge joinorders index access执行子查询在可执行节点中最上级节点NESTED JOIN (INNER JOIN)中执行

< cte query hints >

为Common table expression的query block的hint

Common table expression以inline或materialize方式执行

- Inline方式：以inline view形式执行
- Materialize方式：将common table expression结果存储到instant后在该instant table读取row并执行查询

一般情况下仅使用一次的common table expression以inline方式执行使用两次以上的common table expression以materialize方式执行

INLINE

INLINE hint仅在common table expression的query block有效指定此hint时common table expression像inline view一样执行

以下为使用INLINE hint的示例

```
gSQL> \explain plan
WITH revenue ( supplier_no, total_revenue ) AS
     (
        SELECT /*+ INLINE */
               l_suppkey,
               SUM(l_extendedprice * (1 - l_discount))
          FROM lineitem
         WHERE l_shipdate >= DATE '1996-01-01'
```

```
           AND l_shipdate < DATE '1996-01-01' + INTERVAL '3' MONTH
         GROUP BY
               l_suppkey
     )
SELECT
    s_suppkey,
    s_name,
    ROUND( total_revenue, 2 ) as total_revenue
FROM
    supplier,
    revenue
WHERE
      s_suppkey = supplier_no
  AND total_revenue = (
                        select
                            max(total_revenue)
                        from
                            revenue
                      )
ORDER BY
   s_suppkey;



S_SUPPKEY S_NAME                    TOTAL_REVENUE
--------- ------------------------- -------------
```

```
     8449 Supplier#000008449           1772627.21


1 row selected.


>>>  start print plan


< Execution Plan >
================================================================================
|  IDX  |  NODE DESCRIPTION                                                     |
--------------------------------------------------------------------------------
|    0  |  SELECT STATEMENT                                                     |
|    1  |    QUERY BLOCK ("$QB_IDX_2")                                          |
|    2  |      SORT INSTANT                                                     |
|    3  |        NESTED JOIN (INNER JOIN)                                       |
|    4  |          VIEW ("REVENUE")                                             |
|    5  |            QUERY BLOCK ("$QB_IDX_7")                                  |
|    6  |              GROUP HASH INSTANT                                       |
|    7  |                TABLE ACCESS ("LINEITEM")                              |
|    8  |          INDEX ACCESS ("SUPPLIER", "SUPPLIER_PK_INDEX")               |
|    9  |  SUB QUERY LIST                                                       |
|   10  |    INLINE_VIEW ("$V10")                                               |
|   11  |      QUERY BLOCK ("$QB_IDX_14")                                       |
|   12  |        AGGREGATION BY HASH                                            |
|   13  |          VIEW ("REVENUE")                                             |
|   14  |            QUERY BLOCK ("$QB_IDX_17")                                 |
```

```
|   15  |              GROUP HASH INSTANT                                |
|   16  |                TABLE ACCESS ("LINEITEM")                       |
=======================================================================


     1  -  TARGET : SUPPLIER.S_SUPPKEY, SUPPLIER.S_NAME,
ROUND(REVENUE.TOTAL_REVENUE,2) AS TOTAL_REVENUE
     2  -  SORT KEY : "SUPPLIER.S_SUPPKEY ASC NULLS LAST"
           RECORD COLUMN : SUPPLIER.S_NAME, ROUND(REVENUE.TOTAL_REVENUE,2)
           READ KEY COLUMN : SUPPLIER.S_SUPPKEY
           READ RECORD COLUMN : SUPPLIER.S_NAME,
ROUND(REVENUE.TOTAL_REVENUE,2)
     3  -  JOINED COLUMN : SUPPLIER.S_SUPPKEY, SUPPLIER.S_NAME,
REVENUE.TOTAL_REVENUE
     4  -  COLUMN : LINEITEM.L_SUPPKEY AS SUPPLIER_NO,
SUM( LINEITEM.L_EXTENDEDPRICE * ( 1 - LINEITEM.L_DISCOUNT ) ) AS
TOTAL_REVENUE
     5  -  TARGET : LINEITEM.L_SUPPKEY, SUM( LINEITEM.L_EXTENDEDPRICE *
( 1 - LINEITEM.L_DISCOUNT ) )
     6  -  GROUP KEY : LINEITEM.L_SUPPKEY
           RECORD COLUMN : SUM( LINEITEM.L_EXTENDEDPRICE * ( 1 -
LINEITEM.L_DISCOUNT ) )
           READ KEY COLUMN : LINEITEM.L_SUPPKEY
           READ RECORD COLUMN : SUM( LINEITEM.L_EXTENDEDPRICE * ( 1 -
LINEITEM.L_DISCOUNT ) )
             PHYSICAL FILTER : SUM( LINEITEM.L_EXTENDEDPRICE * ( 1 -
```

```
LINEITEM.L_DISCOUNT ) ) = $V10.$C0
     7  -  READ COLUMN : LINEITEM.L_SUPPKEY, LINEITEM.L_EXTENDEDPRICE,
LINEITEM.L_DISCOUNT, LINEITEM.L_SHIPDATE
             PHYSICAL FILTER : LINEITEM.L_SHIPDATE < DATE'1996-01-01' +
CAST( '3' AS INTERVAL(MONTH) ) AND LINEITEM.L_SHIPDATE >= DATE'1996-01-01'
     8  -  READ INDEX COLUMN : SUPPLIER.S_SUPPKEY
           READ TABLE COLUMN : SUPPLIER.S_NAME
             MIN RANGE : SUPPLIER.S_SUPPKEY = {REVENUE.SUPPLIER_NO}
             MAX RANGE : SUPPLIER.S_SUPPKEY = {REVENUE.SUPPLIER_NO}
           FETCH ONE ROW
    10  -  COLUMN : MAX( REVENUE.TOTAL_REVENUE ) AS $C0
    11  -  TARGET : MAX( REVENUE.TOTAL_REVENUE )
    12  -  AGGREGATION : MAX( REVENUE.TOTAL_REVENUE )
    13  -  COLUMN : SUM( LINEITEM.L_EXTENDEDPRICE * ( 1 -
LINEITEM.L_DISCOUNT ) ) AS TOTAL_REVENUE
    14  -  TARGET : SUM( LINEITEM.L_EXTENDEDPRICE * ( 1 -
LINEITEM.L_DISCOUNT ) )
    15  -  GROUP KEY : LINEITEM.L_SUPPKEY
           RECORD COLUMN : SUM( LINEITEM.L_EXTENDEDPRICE * ( 1 -
LINEITEM.L_DISCOUNT ) )
           READ RECORD COLUMN : SUM( LINEITEM.L_EXTENDEDPRICE * ( 1 -
LINEITEM.L_DISCOUNT ) )
    16  -  READ COLUMN : LINEITEM.L_SUPPKEY, LINEITEM.L_EXTENDEDPRICE,
LINEITEM.L_DISCOUNT, LINEITEM.L_SHIPDATE
             PHYSICAL FILTER : LINEITEM.L_SHIPDATE < DATE'1996-01-01' +
```

```
CAST( '3' AS INTERVAL(MONTH) ) AND LINEITEM.L_SHIPDATE >= DATE'1996-01-01'


<<<  end print plan
```

如上所示在WITH子句定义的revenue在SELECT查询被使用了两次

一般在这种情况下最好以materialize方式处理并避免执行多次revenue但materialize成本大于执行两次的成本时可使用INLINE hint以避免通过materialize方式执行

MATERIALIZE

MATERIALIZE hint仅在common table expression的query block有效指定此hint时执行一次common table expression后将其结果存储在instant table

以下为使用MATERIALIZE hint的示例

```
gSQL> \explain plan
WITH revenue ( supplier_no, total_revenue ) AS
     (
        SELECT /*+ MATERIALIZE */
               l_suppkey,
               SUM(l_extendedprice * (1 - l_discount))
          FROM lineitem
         WHERE l_shipdate >= DATE '1996-01-01'
           AND l_shipdate < DATE '1996-01-01' + INTERVAL '3' MONTH
         GROUP BY
               l_suppkey
     )
```

```
SELECT
    s_suppkey,
    s_name,
    ROUND( total_revenue, 2 ) as total_revenue
FROM
    supplier,
    revenue
WHERE
      s_suppkey = supplier_no
  AND total_revenue = (
                        select
                            max(total_revenue)
                        from
                            revenue
                       )
ORDER BY
   s_suppkey;



S_SUPPKEY S_NAME                    TOTAL_REVENUE
--------- ------------------------- -------------
     8449 Supplier#000008449           1772627.21


1 row selected.
```

```
>>>  start print plan


< Execution Plan >

============================================================================

|  IDX  |  NODE DESCRIPTION                                                  |

----------------------------------------------------------------------------

|    0  |  SELECT STATEMENT                                                  |

|    1  |    QUERY BLOCK ("$QB_IDX_2")                                       |

|    2  |      SORT INSTANT                                                  |

|    3  |        NESTED JOIN (INNER JOIN)                                    |

|    4  |          MTR ACCESS ("REVENUE")                                    |

|    5  |          INDEX ACCESS ("SUPPLIER", "SUPPLIER_PK_INDEX")            |

|    6  |    MTR LOADER LIST                                                 |

|    7  |      MTR LOADER ("REVENUE")                                        |

|    8  |        VIEW ("REVENUE")                                            |

|    9  |          QUERY BLOCK ("$QB_IDX_7")                                 |

|   10  |            GROUP HASH INSTANT                                      |

|   11  |              TABLE ACCESS ("LINEITEM")                             |

|   12  |  SUB QUERY LIST                                                    |

|   13  |    INLINE_VIEW ("$V12")                                            |

|   14  |      QUERY BLOCK ("$QB_IDX_14")                                    |

|   15  |        MTR ACCESS ("REVENUE")                                      |

============================================================================


     1  -  TARGET : SUPPLIER.S_SUPPKEY,
```

```
                    SUPPLIER.S_NAME,
                    ROUND(REVENUE.TOTAL_REVENUE,2) AS TOTAL_REVENUE
     2  -   SORT KEY : "SUPPLIER.S_SUPPKEY ASC NULLS LAST"
            RECORD COLUMN : SUPPLIER.S_NAME, ROUND(REVENUE.TOTAL_REVENUE,2)
            READ KEY COLUMN : SUPPLIER.S_SUPPKEY
            READ RECORD COLUMN : SUPPLIER.S_NAME,
                                 ROUND(REVENUE.TOTAL_REVENUE,2)
     3  -   JOINED COLUMN : SUPPLIER.S_SUPPKEY,
                            SUPPLIER.S_NAME,
                            REVENUE.TOTAL_REVENUE
     4  -   READ COLUMN : REVENUE.SUPPLIER_NO, REVENUE.TOTAL_REVENUE
              PHYSICAL FILTER : REVENUE.TOTAL_REVENUE = $V12.$C0
     5  -   READ INDEX COLUMN : SUPPLIER.S_SUPPKEY
            READ TABLE COLUMN : SUPPLIER.S_NAME
              MIN RANGE : SUPPLIER.S_SUPPKEY = {REVENUE.SUPPLIER_NO}
              MAX RANGE : SUPPLIER.S_SUPPKEY = {REVENUE.SUPPLIER_NO}
            FETCH ONE ROW
     7  -   RECORD COLUMN : REVENUE.SUPPLIER_NO, REVENUE.TOTAL_REVENUE
     8  -   COLUMN : LINEITEM.L_SUPPKEY AS SUPPLIER_NO,
                     SUM( LINEITEM.L_EXTENDEDPRICE * ( 1 -
LINEITEM.L_DISCOUNT ) ) AS TOTAL_REVENUE
     9  -   TARGET : LINEITEM.L_SUPPKEY,
                    SUM( LINEITEM.L_EXTENDEDPRICE * ( 1 -
LINEITEM.L_DISCOUNT ) )
    10  -   GROUP KEY : LINEITEM.L_SUPPKEY
```

```
            RECORD COLUMN : SUM( LINEITEM.L_EXTENDEDPRICE * ( 1 -
LINEITEM.L_DISCOUNT ) )
            READ KEY COLUMN : LINEITEM.L_SUPPKEY
            READ RECORD COLUMN : SUM( LINEITEM.L_EXTENDEDPRICE * ( 1 -
LINEITEM.L_DISCOUNT ) )
    11  -   READ COLUMN : LINEITEM.L_SUPPKEY,
                          LINEITEM.L_EXTENDEDPRICE,
                          LINEITEM.L_DISCOUNT, LINEITEM.L_SHIPDATE
          PHYSICAL FILTER : LINEITEM.L_SHIPDATE < DATE'1996-01-01' +
CAST( '3' AS INTERVAL(MONTH) ) AND LINEITEM.L_SHIPDATE >= DATE'1996-01-01'
    13  -   COLUMN : MAX( REVENUE.TOTAL_REVENUE ) AS $C0
    14  -   TARGET : MAX( REVENUE.TOTAL_REVENUE )
    15  -   READ COLUMN : REVENUE.TOTAL_REVENUE
            AGGREGATION : MAX( REVENUE.TOTAL_REVENUE )


<<<  end print plan
```

如上所示在WITH子句定义的revenue在SELECT 查询中被使用了两次

以materialize方式执行时仅执行一次common table expression后将其结果存储到instant table后使用因此会缩短执行时间

NO_QUERY_TRANSFORMATION

描述NO_QUERY_TRANSFORMATION hint时指定hint的query block和其下级的所有query block不执行query transform

以下为使用NO_QUERY_TRANSFORMATION hint的示例

```
\EXPLAIN PLAN
SELECT /*+ NO_QUERY_TRANSFORMATION */
       s_name,
       l_quantity
  FROM supplier,
       (
        SELECT
               l_suppkey,
               l_quantity
          FROM lineitem
         WHERE l_shipdate >= date '1996-01-01'
           AND l_shipdate < date '1996-01-01' + interval '3' month
       ) lineitem_view
 WHERE s_suppkey = l_suppkey
   AND s_nationkey IN (
                        SELECT n_nationkey
                          FROM nation
                         WHERE n_name = 'FRANCE' OR n_name = 'GERMANY'
                      )
;

>>>  start print plan
```

```
< Execution Plan >

=============================================================================

|  IDX  |  NODE DESCRIPTION                                                  |

-----------------------------------------------------------------------------

|    0  |  SELECT STATEMENT                                                  |

|    1  |    QUERY BLOCK ("$QB_IDX_2")                                       |

|    2  |      NESTED JOIN (INNER JOIN)                                      |

|    3  |        INLINE_VIEW ("LINEITEM_VIEW")                               |

|    4  |          QUERY BLOCK ("$QB_IDX_7")                                 |

|    5  |            TABLE ACCESS ("LINEITEM")                               |

|    6  |        INDEX ACCESS ("SUPPLIER", "SUPPLIER_PK_INDEX")              |

|    7  |  SUB QUERY LIST                                                    |

|    8  |    INLINE_VIEW ("$V8") (MATERIALIZED)                              |

|    9  |      QUERY BLOCK ("$QB_IDX_12")                                    |

|   10  |        TABLE ACCESS ("NATION")                                     |

=============================================================================


     1  -  TARGET : SUPPLIER.S_NAME, LINEITEM_VIEW.L_QUANTITY

     2  -  JOINED COLUMN : SUPPLIER.S_NATIONKEY, SUPPLIER.S_NAME,
LINEITEM_VIEW.L_QUANTITY

             POST WHERE FILTER : ( SUPPLIER.S_NATIONKEY ) IN
( $V8.N_NATIONKEY )

     3  -  COLUMN : LINEITEM.L_SUPPKEY AS L_SUPPKEY, LINEITEM.L_QUANTITY
AS L_QUANTITY

     4  -  TARGET : LINEITEM.L_SUPPKEY, LINEITEM.L_QUANTITY
```

```
     5  -  READ COLUMN : LINEITEM.L_SUPPKEY, LINEITEM.L_QUANTITY,
LINEITEM.L_SHIPDATE
             PHYSICAL FILTER : LINEITEM.L_SHIPDATE >= DATE'1996-01-01' AND
LINEITEM.L_SHIPDATE < DATE'1996-01-01' + CAST( '3' AS INTERVAL(MONTH) )
     6  -  READ INDEX COLUMN : SUPPLIER.S_SUPPKEY
            READ TABLE COLUMN : SUPPLIER.S_NAME, SUPPLIER.S_NATIONKEY
              MIN RANGE : SUPPLIER.S_SUPPKEY = {LINEITEM_VIEW.L_SUPPKEY}
              MAX RANGE : SUPPLIER.S_SUPPKEY = {LINEITEM_VIEW.L_SUPPKEY}
            FETCH ONE ROW
     8  -  COLUMN : NATION.N_NATIONKEY AS N_NATIONKEY
     9  -  TARGET : NATION.N_NATIONKEY
    10  -  READ COLUMN : NATION.N_NATIONKEY, NATION.N_NAME
              LOGICAL FILTER : NATION.N_NAME = 'FRANCE' OR NATION.N_NAME =
'GERMANY'


<<<  end print plan
```

如上所示可应用simple view merging和subquery unnsting但由于

NO_QUERY_TRANSFORMATION hint两种query transformation方法均未应用

<unnest subquery hints>

<unnest subquery hints>中有<unnest hints>, <unnest join operation hints>, <unnest join driver hints>, <unnest join pusher hints>, <unnest merge hints>

Subquery unnesting是query transformation过程中的一部分因此使用

NO_QUERY_TRANSFORMATION hint也不执行subquery unnesting

相同的子查询中同时使用NO_QUERY_TRANSFORMATION hint和<unnest subquery hints>时仅应用优先描述的hint忽略余下hint

相同的子查询中同时使用<push subquery hinst>和<unnest subquery hints>时也仅应用优先描述的hint忽略余下hint<push subquery hints>是未unnest的子查询的hint因此无法与<unnest subquery hints>同时应用

各个hint的说明和示例如下

<unnest hints>

UNNEST

描述UNNEST hint时optimizer将子查询转换为保证相同结果的join语句但满足subquery unnesting约束条件时才可应用hint为无法应用hint的子查询则忽略hint
约束条件相关详细内容参考**Subquery Unnesting**

以下为使用UNNEST hint的示例

```
\EXPLAIN PLAN
SELECT
       o_orderdate,
       o_totalprice
  FROM orders
 WHERE o_orderkey IN (
                      SELECT   /*+ UNNEST */
```

```
                              l_orderkey
                      FROM    lineitem
                    GROUP BY l_orderkey
                    HAVING    sum(l_quantity) > 300
                  )
;


>>>  start print plan


< Execution Plan >
=============================================================================
|  IDX  |  NODE DESCRIPTION                                                 |
-----------------------------------------------------------------------------
|    0  |  SELECT STATEMENT                                                 |
|    1  |    QUERY BLOCK ("$QB_IDX_2")                                      |
|    2  |      NESTED JOIN (INNER JOIN)                                     |
|    3  |        INLINE_VIEW ("$V4")                                        |
|    4  |          QUERY BLOCK ("$QB_IDX_6")                                |
|    5  |            GROUP HASH INSTANT                                     |
|    6  |              TABLE ACCESS ("LINEITEM")                            |
|    7  |        INDEX ACCESS ("ORDERS", "ORDERS_PK_INDEX")                 |
=============================================================================


     1  -  TARGET : ORDERS.O_ORDERDATE, ORDERS.O_TOTALPRICE
     2  -  JOINED COLUMN : ORDERS.O_ORDERDATE, ORDERS.O_TOTALPRICE
```

```
    3  -  COLUMN : LINEITEM.L_ORDERKEY AS L_ORDERKEY
    4  -  TARGET : LINEITEM.L_ORDERKEY
    5  -  GROUP KEY : LINEITEM.L_ORDERKEY
          RECORD COLUMN : SUM( LINEITEM.L_QUANTITY )
          READ KEY COLUMN : LINEITEM.L_ORDERKEY
          READ RECORD COLUMN : SUM( LINEITEM.L_QUANTITY )
            PHYSICAL FILTER : SUM( LINEITEM.L_QUANTITY ) > 300
    6  -  READ COLUMN : LINEITEM.L_ORDERKEY, LINEITEM.L_QUANTITY
    7  -  READ INDEX COLUMN : ORDERS.O_ORDERKEY
          READ TABLE COLUMN : ORDERS.O_TOTALPRICE, ORDERS.O_ORDERDATE
            MIN RANGE : ORDERS.O_ORDERKEY = {$V4.L_ORDERKEY}
            MAX RANGE : ORDERS.O_ORDERKEY = {$V4.L_ORDERKEY}
          FETCH ONE ROW

<<<  end print plan
```

NO_UNNEST

描述NO_UNNEST hint时optimizer不unnest子查询及按照子查询形式直接进行filter处理

以下为使用NO_UNNEST hint的示例

```
\EXPLAIN PLAN
SELECT
       o_orderdate,
       o_totalprice
```

```
  FROM orders
 WHERE o_orderkey IN (
                       SELECT   /*+ NO_UNNEST */
                                l_orderkey
                         FROM   lineitem
                       GROUP BY l_orderkey
                       HAVING   sum(l_quantity) > 300
                     )
;


no rows selected.


>>>  start print plan


< Execution Plan >
=============================================================================
|  IDX  |  NODE DESCRIPTION                                                 |
-----------------------------------------------------------------------------
|    0  |  SELECT STATEMENT                                                 |
|    1  |    QUERY BLOCK ("$QB_IDX_2")                                      |
|    2  |      TABLE ACCESS ("ORDERS")                                      |
|    3  |  SUB QUERY LIST                                                   |
|    4  |    INLINE_VIEW ("$V4") (MATERIALIZED)                             |
|    5  |      QUERY BLOCK ("$QB_IDX_6")                                    |
|    6  |        GROUP HASH INSTANT                                         |
```

```
|    7  |          TABLE ACCESS ("LINEITEM")                              |
=======================================================================


     1  -  TARGET : ORDERS.O_ORDERDATE, ORDERS.O_TOTALPRICE
     2  -  READ COLUMN : ORDERS.O_ORDERKEY, ORDERS.O_TOTALPRICE,
ORDERS.O_ORDERDATE
              POST FILTER : ( ORDERS.O_ORDERKEY ) IN ( $V4.L_ORDERKEY )
     4  -  COLUMN : LINEITEM.L_ORDERKEY AS L_ORDERKEY
     5  -  TARGET : LINEITEM.L_ORDERKEY
     6  -  GROUP KEY : LINEITEM.L_ORDERKEY
           RECORD COLUMN : SUM( LINEITEM.L_QUANTITY )
           READ KEY COLUMN : LINEITEM.L_ORDERKEY
           READ RECORD COLUMN : SUM( LINEITEM.L_QUANTITY )
             PHYSICAL FILTER : SUM( LINEITEM.L_QUANTITY ) > 300
     7  -  READ COLUMN : LINEITEM.L_ORDERKEY, LINEITEM.L_QUANTITY


<<<  end print plan
```

<unnest join operation hints>

描述<unnest join operation hints>时optimizer将子查询转换为保证相同结果的join语句并按照hint中指定的方式执行join

UNNEST_NL

描述UNNEST_NL hint时optimizer将子查询转换为保证相同结果的join语句并以nested loop join

方式执行其join

以下为使用UNNEST_NL hint的示例

```
\EXPLAIN PLAN
SELECT
       o_orderdate,
       o_totalprice
  FROM orders
 WHERE o_orderkey IN (
                       SELECT   /*+ UNNEST_NL */
                                l_orderkey
                         FROM   lineitem
                       GROUP BY l_orderkey
                       HAVING   sum(l_quantity) > 300
                     )
;



>>>  start print plan


< Execution Plan >
=============================================================================
|  IDX  |  NODE DESCRIPTION                                                 |
-----------------------------------------------------------------------------
|    0  |  SELECT STATEMENT                                                 |
```

```
|    1  |    QUERY BLOCK ("$QB_IDX_2")                                    |
|    2  |      NESTED JOIN (INNER JOIN)                                    |
|    3  |        INLINE_VIEW ("$V4")                                       |
|    4  |          QUERY BLOCK ("$QB_IDX_6")                               |
|    5  |            GROUP HASH INSTANT                                    |
|    6  |              TABLE ACCESS ("LINEITEM")                           |
|    7  |        INDEX ACCESS ("ORDERS", "ORDERS_PK_INDEX")                |
=======================================================================


     1  -  TARGET : ORDERS.O_ORDERDATE, ORDERS.O_TOTALPRICE
     2  -  JOINED COLUMN : ORDERS.O_ORDERDATE, ORDERS.O_TOTALPRICE
     3  -  COLUMN : LINEITEM.L_ORDERKEY AS L_ORDERKEY
     4  -  TARGET : LINEITEM.L_ORDERKEY
     5  -  GROUP KEY : LINEITEM.L_ORDERKEY
           RECORD COLUMN : SUM( LINEITEM.L_QUANTITY )
           READ KEY COLUMN : LINEITEM.L_ORDERKEY
           READ RECORD COLUMN : SUM( LINEITEM.L_QUANTITY )
             PHYSICAL FILTER : SUM( LINEITEM.L_QUANTITY ) > 300
     6  -  READ COLUMN : LINEITEM.L_ORDERKEY, LINEITEM.L_QUANTITY
     7  -  READ INDEX COLUMN : ORDERS.O_ORDERKEY
           READ TABLE COLUMN : ORDERS.O_TOTALPRICE, ORDERS.O_ORDERDATE
             MIN RANGE : ORDERS.O_ORDERKEY = {$V4.L_ORDERKEY}
             MAX RANGE : ORDERS.O_ORDERKEY = {$V4.L_ORDERKEY}
           FETCH ONE ROW
```

```
<<<  end print plan
```

可看到子查询被unnest到INLINE_VIEW ("$V4")后参与NESTED JOIN

UNNEST_NL_IN

描述UNNEST_NL_IN hint时optimizer将子查询转换为保证相同结果的join语句并以nested loop join方式执行其join并且将unnest到表或view的子查询放在join的right

以下为使用UNNEST_NL_IN hint的示例

```
\EXPLAIN PLAN
SELECT
       o_orderdate,
       o_totalprice
  FROM orders
 WHERE o_orderkey IN (
                       SELECT   /*+ UNNEST_NL_IN */
                                l_orderkey
                         FROM   lineitem
                       GROUP BY l_orderkey
                       HAVING   sum(l_quantity) > 300
                     )
;

>>>  start print plan
```

```
< Execution Plan >

=============================================================================

|  IDX  |  NODE DESCRIPTION                                                   |

-----------------------------------------------------------------------------

|    0  |  SELECT STATEMENT                                                   |

|    1  |    QUERY BLOCK ("$QB_IDX_2")                                        |

|    2  |      NESTED JOIN (INNER JOIN)                                       |

|    3  |        TABLE ACCESS ("ORDERS")                                      |

|    4  |        INLINE_VIEW ("$V5")                                          |

|    5  |          QUERY BLOCK ("$QB_IDX_6")                                  |

|    6  |            GROUP                                                    |

|    7  |              INDEX ACCESS ("LINEITEM", "LINEITEM_PK_INDEX")         |

=============================================================================


     1  -  TARGET : ORDERS.O_ORDERDATE, ORDERS.O_TOTALPRICE

     2  -  JOINED COLUMN : ORDERS.O_ORDERDATE, ORDERS.O_TOTALPRICE

     3  -  READ COLUMN : ORDERS.O_ORDERKEY, ORDERS.O_TOTALPRICE,

ORDERS.O_ORDERDATE

     4  -  COLUMN : LINEITEM.L_ORDERKEY AS L_ORDERKEY

     5  -  TARGET : LINEITEM.L_ORDERKEY

     6  -  GROUP KEY : LINEITEM.L_ORDERKEY

           RECORD COLUMN : SUM( LINEITEM.L_QUANTITY )

             LOGICAL FILTER : SUM( LINEITEM.L_QUANTITY ) > 300

     7  -  READ INDEX COLUMN : LINEITEM.L_ORDERKEY

           READ TABLE COLUMN : LINEITEM.L_QUANTITY
```

```
            MIN RANGE : LINEITEM.L_ORDERKEY = {ORDERS.O_ORDERKEY}

            MAX RANGE : LINEITEM.L_ORDERKEY = {ORDERS.O_ORDERKEY}


<<<  end print plan
```

可看到子查询unnest到INLINE_VIEW ("$V5")并成为NESTED JOIN的right child参与join

UNNEST_NL_OUT

描述UNNEST_NL_OUT hint时optimizer将子查询转换为保证相同结果的join语句并以nested loop join方式执行该join并且将unnest到表或view的子查询放在join的left

以下为使用UNNEST_NL_OUT hint的示例

```
\EXPLAIN PLAN
SELECT
       o_orderdate,
       o_totalprice
  FROM orders
 WHERE o_orderkey IN (
                      SELECT   /*+ UNNEST_NL_OUT */
                               l_orderkey
                        FROM   lineitem
                      GROUP BY l_orderkey
                      HAVING   sum(l_quantity) > 300
                    )
;
```

```
>>>  start print plan


< Execution Plan >

=============================================================================
|  IDX  |  NODE DESCRIPTION                                                  |
-----------------------------------------------------------------------------
|    0  |  SELECT STATEMENT                                                  |
|    1  |    QUERY BLOCK ("$QB_IDX_2")                                       |
|    2  |      NESTED JOIN (INNER JOIN)                                      |
|    3  |        INLINE_VIEW ("$V4")                                         |
|    4  |          QUERY BLOCK ("$QB_IDX_6")                                 |
|    5  |            GROUP HASH INSTANT                                      |
|    6  |              TABLE ACCESS ("LINEITEM")                             |
|    7  |        INDEX ACCESS ("ORDERS", "ORDERS_PK_INDEX")                  |
=============================================================================


     1  -  TARGET : ORDERS.O_ORDERDATE, ORDERS.O_TOTALPRICE
     2  -  JOINED COLUMN : ORDERS.O_ORDERDATE, ORDERS.O_TOTALPRICE
     3  -  COLUMN : LINEITEM.L_ORDERKEY AS L_ORDERKEY
     4  -  TARGET : LINEITEM.L_ORDERKEY
     5  -  GROUP KEY : LINEITEM.L_ORDERKEY
           RECORD COLUMN : SUM( LINEITEM.L_QUANTITY )
           READ KEY COLUMN : LINEITEM.L_ORDERKEY
```

```
            READ RECORD COLUMN : SUM( LINEITEM.L_QUANTITY )
              PHYSICAL FILTER : SUM( LINEITEM.L_QUANTITY ) > 300
      6   -   READ COLUMN : LINEITEM.L_ORDERKEY, LINEITEM.L_QUANTITY
      7   -   READ INDEX COLUMN : ORDERS.O_ORDERKEY
            READ TABLE COLUMN : ORDERS.O_TOTALPRICE, ORDERS.O_ORDERDATE
              MIN RANGE : ORDERS.O_ORDERKEY = {$V4.L_ORDERKEY}
              MAX RANGE : ORDERS.O_ORDERKEY = {$V4.L_ORDERKEY}
            FETCH ONE ROW

<<<  end print plan
```

可看到子查询unnest到INLINE_VIEW ("$V4")并成为NESTED JOIN的left child参与join

UNNEST_INL

描述UNNEST_INL hint时optimizer将子查询转换为保证相同结果的join语句并以instant nested loop join方式执行该join

以下为使用UNNEST_INL hint的示例

```
\EXPLAIN PLAN
SELECT
       o_orderdate,
       o_totalprice
  FROM orders
 WHERE o_orderkey IN (
                      SELECT   /*+ UNNEST_INL */
```

```
                              l_orderkey
                        FROM   lineitem
                      GROUP BY l_orderkey
                      HAVING   sum(l_quantity) > 300
                    )
;


>>>  start print plan


< Execution Plan >
=============================================================================
|  IDX  |  NODE DESCRIPTION                                                 |
-----------------------------------------------------------------------------
|    0  |  SELECT STATEMENT                                                 |
|    1  |    QUERY BLOCK ("$QB_IDX_2")                                      |
|    2  |      NESTED JOIN (INNER JOIN)                                     |
|    3  |        TABLE ACCESS ("ORDERS")                                    |
|    4  |        SORT JOIN INSTANT                                          |
|    5  |          INLINE_VIEW ("$V6")                                      |
|    6  |            QUERY BLOCK ("$QB_IDX_6")                              |
|    7  |              GROUP HASH INSTANT                                   |
|    8  |                TABLE ACCESS ("LINEITEM")                          |
=============================================================================


     1  -  TARGET : ORDERS.O_ORDERDATE, ORDERS.O_TOTALPRICE
```

```
     2  -  JOINED COLUMN : ORDERS.O_ORDERDATE, ORDERS.O_TOTALPRICE
     3  -  READ COLUMN : ORDERS.O_ORDERKEY, ORDERS.O_TOTALPRICE,
ORDERS.O_ORDERDATE
     4  -  SORT KEY : "$V6.L_ORDERKEY ASC NULLS LAST"
           READ KEY COLUMN : $V6.L_ORDERKEY
             MIN RANGE : $V6.L_ORDERKEY = {ORDERS.O_ORDERKEY}
             MAX RANGE : $V6.L_ORDERKEY = {ORDERS.O_ORDERKEY}
     5  -  COLUMN : LINEITEM.L_ORDERKEY AS L_ORDERKEY
     6  -  TARGET : LINEITEM.L_ORDERKEY
     7  -  GROUP KEY : LINEITEM.L_ORDERKEY
           RECORD COLUMN : SUM( LINEITEM.L_QUANTITY )
           READ KEY COLUMN : LINEITEM.L_ORDERKEY
           READ RECORD COLUMN : SUM( LINEITEM.L_QUANTITY )
             PHYSICAL FILTER : SUM( LINEITEM.L_QUANTITY ) > 300
     8  -  READ COLUMN : LINEITEM.L_ORDERKEY, LINEITEM.L_QUANTITY


<<<  end print plan
```

可看到子查询unnest到INLINE_VIEW ("$V6")后参与NESTED JOIN

UNNEST_INL_IN

描述UNNEST_INL_IN hint时optimizer将子查询转换为保证相同结果的join语句并以instant nested loop join 方式执行该join并且将unnest到表或view的子查询放在join的right

以下为使用UNNEST_INL_IN hint的示例

```
\EXPLAIN PLAN
SELECT
       o_orderdate,
       o_totalprice
  FROM orders
 WHERE o_orderkey IN (
                       SELECT   /*+ UNNEST_INL_IN */
                                l_orderkey
                         FROM   lineitem
                       GROUP BY l_orderkey
                       HAVING   sum(l_quantity) > 300
                     )
;

no rows selected.

>>>  start print plan

< Execution Plan >
=============================================================================
|  IDX  |  NODE DESCRIPTION                                                 |
-----------------------------------------------------------------------------
|    0  |  SELECT STATEMENT                                                 |
|    1  |    QUERY BLOCK ("$QB_IDX_2")                                      |
|    2  |      NESTED JOIN (INNER JOIN)                                     |
```

```
|    3  |        TABLE ACCESS ("ORDERS")                                  |
|    4  |        SORT JOIN INSTANT                                        |
|    5  |          INLINE_VIEW ("$V6")                                    |
|    6  |            QUERY BLOCK ("$QB_IDX_6")                            |
|    7  |              GROUP HASH INSTANT                                  |
|    8  |                TABLE ACCESS ("LINEITEM")                         |
=====================================================================


     1  -  TARGET : ORDERS.O_ORDERDATE, ORDERS.O_TOTALPRICE
     2  -  JOINED COLUMN : ORDERS.O_ORDERDATE, ORDERS.O_TOTALPRICE
     3  -  READ COLUMN : ORDERS.O_ORDERKEY, ORDERS.O_TOTALPRICE,
ORDERS.O_ORDERDATE
     4  -  SORT KEY : "$V6.L_ORDERKEY ASC NULLS LAST"
           READ KEY COLUMN : $V6.L_ORDERKEY
             MIN RANGE : $V6.L_ORDERKEY = {ORDERS.O_ORDERKEY}
             MAX RANGE : $V6.L_ORDERKEY = {ORDERS.O_ORDERKEY}
     5  -  COLUMN : LINEITEM.L_ORDERKEY AS L_ORDERKEY
     6  -  TARGET : LINEITEM.L_ORDERKEY
     7  -  GROUP KEY : LINEITEM.L_ORDERKEY
           RECORD COLUMN : SUM( LINEITEM.L_QUANTITY )
           READ KEY COLUMN : LINEITEM.L_ORDERKEY
           READ RECORD COLUMN : SUM( LINEITEM.L_QUANTITY )
             PHYSICAL FILTER : SUM( LINEITEM.L_QUANTITY ) > 300
     8  -  READ COLUMN : LINEITEM.L_ORDERKEY, LINEITEM.L_QUANTITY
```

```
<<<  end print plan
```

子查询unnest到INLINE_VIEW ("$V6")并成为NESTED JOIN的right child参与join

UNNEST_INL_OUT

描述UNNEST_INL_OUT hint时optimizer将子查询转换为保证相同结果的join语句并以instant nested loop join方式执行该join并且将unnest到表或view的子查询放在join的left

以下为使用 UNNEST_INL_OUT hint的示例

```
\EXPLAIN PLAN
SELECT
       o_orderdate,
       o_totalprice
  FROM orders
 WHERE o_orderkey IN (
                      SELECT   /*+ UNNEST_INL_OUT */
                               l_orderkey
                        FROM   lineitem
                      GROUP BY l_orderkey
                      HAVING   sum(l_quantity) > 300
                    )
;


no rows selected.
```

```
>>>  start print plan


< Execution Plan >

=============================================================================
|  IDX  |  NODE DESCRIPTION                                                 |
-----------------------------------------------------------------------------
|    0  |  SELECT STATEMENT                                                 |
|    1  |    QUERY BLOCK ("$QB_IDX_2")                                      |
|    2  |      NESTED JOIN (INNER JOIN)                                     |
|    3  |        INLINE_VIEW ("$V4")                                        |
|    4  |          QUERY BLOCK ("$QB_IDX_6")                                |
|    5  |            GROUP HASH INSTANT                                     |
|    6  |              TABLE ACCESS ("LINEITEM")                            |
|    7  |        SORT JOIN INSTANT                                          |
|    8  |          TABLE ACCESS ("ORDERS")                                  |
=============================================================================


     1  -  TARGET : ORDERS.O_ORDERDATE, ORDERS.O_TOTALPRICE
     2  -  JOINED COLUMN : ORDERS.O_ORDERDATE, ORDERS.O_TOTALPRICE
     3  -  COLUMN : LINEITEM.L_ORDERKEY AS L_ORDERKEY
     4  -  TARGET : LINEITEM.L_ORDERKEY
     5  -  GROUP KEY : LINEITEM.L_ORDERKEY
           RECORD COLUMN : SUM( LINEITEM.L_QUANTITY )
           READ KEY COLUMN : LINEITEM.L_ORDERKEY
           READ RECORD COLUMN : SUM( LINEITEM.L_QUANTITY )
```

```
            PHYSICAL FILTER : SUM( LINEITEM.L_QUANTITY ) > 300
     6  -  READ COLUMN : LINEITEM.L_ORDERKEY, LINEITEM.L_QUANTITY
     7  -  SORT KEY : "ORDERS.O_ORDERKEY ASC NULLS LAST"
           RECORD COLUMN : ORDERS.O_ORDERDATE, ORDERS.O_TOTALPRICE
           READ KEY COLUMN : ORDERS.O_ORDERKEY
           READ RECORD COLUMN : ORDERS.O_ORDERDATE, ORDERS.O_TOTALPRICE
             MIN RANGE : ORDERS.O_ORDERKEY = {$V4.L_ORDERKEY}
             MAX RANGE : ORDERS.O_ORDERKEY = {$V4.L_ORDERKEY}
     8  -  READ COLUMN : ORDERS.O_ORDERKEY, ORDERS.O_TOTALPRICE,
ORDERS.O_ORDERDATE


<<<  end print plan
```

可看到子查询unnest到INLINE_VIEW ("$V4")并成为NESTED JOIN的left child参与join

UNNEST_HASH

描述UNNEST_HASH hint时optimizer将子查询转换为保证相同结果的join语句并以hash join方式执行其join

以下为使用UNNEST_HASH hint的示例

```
\EXPLAIN PLAN
SELECT
       o_orderdate,
       o_totalprice
  FROM orders
```

```
 WHERE o_orderkey IN (
                      SELECT   /*+ UNNEST_HASH */
                               l_orderkey
                        FROM   lineitem
                      GROUP BY l_orderkey
                      HAVING   sum(l_quantity) > 300
                    )
;


>>>  start print plan


< Execution Plan >
=========================================================================
|  IDX  |  NODE DESCRIPTION                                             |
-------------------------------------------------------------------------
|    0  |  SELECT STATEMENT                                             |
|    1  |    QUERY BLOCK ("$QB_IDX_2")                                  |
|    2  |      HASH JOIN (INNER JOIN)                                   |
|    3  |        TABLE ACCESS ("ORDERS")                                |
|    4  |        HASH JOIN INSTANT                                      |
|    5  |          INLINE_VIEW ("$V6")                                  |
|    6  |            QUERY BLOCK ("$QB_IDX_6")                          |
|    7  |              GROUP HASH INSTANT                               |
|    8  |                TABLE ACCESS ("LINEITEM")                      |
=========================================================================
```

```
     1  -  TARGET : ORDERS.O_ORDERDATE, ORDERS.O_TOTALPRICE
     2  -  JOINED COLUMN : ORDERS.O_ORDERDATE, ORDERS.O_TOTALPRICE
     3  -  READ COLUMN : ORDERS.O_ORDERKEY, ORDERS.O_TOTALPRICE,
ORDERS.O_ORDERDATE
     4  -  HASH KEY : $V6.L_ORDERKEY
           READ KEY COLUMN : $V6.L_ORDERKEY
             HASH FILTER : $V6.L_ORDERKEY = ORDERS.O_ORDERKEY
           FETCH ONE ROW
     5  -  COLUMN : LINEITEM.L_ORDERKEY AS L_ORDERKEY
     6  -  TARGET : LINEITEM.L_ORDERKEY
     7  -  GROUP KEY : LINEITEM.L_ORDERKEY
           RECORD COLUMN : SUM( LINEITEM.L_QUANTITY )
           READ KEY COLUMN : LINEITEM.L_ORDERKEY
           READ RECORD COLUMN : SUM( LINEITEM.L_QUANTITY )
             PHYSICAL FILTER : SUM( LINEITEM.L_QUANTITY ) > 300
     8  -  READ COLUMN : LINEITEM.L_ORDERKEY, LINEITEM.L_QUANTITY


<<<  end print plan
```

可看到子查询unnest到INLINE_VIEW ("$V6")后参与NESTED JOIN

UNNEST_HASH(hash_bucket_count)

UNNEST_HASH(hash_bucket_count) hint与UNNEST_HASH hint相同区别在于可指定hash bucket count

因此使用此hint可创建hash_bucket_count大小的hash的bucket

以下为使用UNNEST_HASH(hash_bucket_count) hint的示例

```
\EXPLAIN PLAN
SELECT
       o_orderdate,
       o_totalprice
  FROM orders
 WHERE o_orderkey IN (
                      SELECT   /*+ UNNEST_HASH(3) */
                               l_orderkey
                        FROM   lineitem
                      GROUP BY l_orderkey
                      HAVING   sum(l_quantity) > 300
                    )
;



>>>  start print plan


< Execution Plan >
=============================================================================
|  IDX  |  NODE DESCRIPTION                                                 |
-----------------------------------------------------------------------------
|    0  |  SELECT STATEMENT                                                 |
```

```
|    1  |    QUERY BLOCK ("$QB_IDX_2")                                    |
|    2  |      HASH JOIN (INNER JOIN)                                      |
|    3  |        TABLE ACCESS ("ORDERS")                                   |
|    4  |        HASH JOIN INSTANT                                         |
|    5  |          INLINE_VIEW ("$V6")                                     |
|    6  |            QUERY BLOCK ("$QB_IDX_6")                             |
|    7  |              GROUP HASH INSTANT                                  |
|    8  |                TABLE ACCESS ("LINEITEM")                         |
=======================================================================


     1  -  TARGET : ORDERS.O_ORDERDATE, ORDERS.O_TOTALPRICE
     2  -  JOINED COLUMN : ORDERS.O_ORDERDATE, ORDERS.O_TOTALPRICE
     3  -  READ COLUMN : ORDERS.O_ORDERKEY, ORDERS.O_TOTALPRICE,
ORDERS.O_ORDERDATE
     4  -  HASH KEY : $V6.L_ORDERKEY
           READ KEY COLUMN : $V6.L_ORDERKEY
             HASH FILTER : $V6.L_ORDERKEY = ORDERS.O_ORDERKEY
           FETCH ONE ROW
     5  -  COLUMN : LINEITEM.L_ORDERKEY AS L_ORDERKEY
     6  -  TARGET : LINEITEM.L_ORDERKEY
     7  -  GROUP KEY : LINEITEM.L_ORDERKEY
           RECORD COLUMN : SUM( LINEITEM.L_QUANTITY )
           READ KEY COLUMN : LINEITEM.L_ORDERKEY
           READ RECORD COLUMN : SUM( LINEITEM.L_QUANTITY )
             PHYSICAL FILTER : SUM( LINEITEM.L_QUANTITY ) > 300
```

```
    8  -  READ COLUMN : LINEITEM.L_ORDERKEY, LINEITEM.L_QUANTITY


<<<  end print plan
```

Execution plan与使用UNNEST_HASH hint的示例相同但差异在于使用上述hint在HASH JOIN INSTANT创建hash bucket count大小的hash bucket因此执行cost estimation时由于hash bucket count过大或过小而性能缓慢时可使用此hint进行调整

UNNEST_HASH_IN

描述UNNEST_HASH_IN hint时optimizer将子查询转换为保证相同结果的join语句并以hash join方式执行该join并且将unnest到表或view的子查询放在join的right

以下为使用UNNEST_HASH_IN hint的示例

```
\EXPLAIN PLAN
SELECT
       o_orderdate,
       o_totalprice
  FROM orders
 WHERE o_orderkey IN (
                      SELECT   /*+ UNNEST_HASH_IN */
                               l_orderkey
                        FROM   lineitem
                      GROUP BY l_orderkey
                      HAVING   sum(l_quantity) > 300
                    )
```

```
;


>>>  start print plan


< Execution Plan >

================================================================================
|  IDX  |  NODE DESCRIPTION                                                    |
--------------------------------------------------------------------------------
|    0  |  SELECT STATEMENT                                                    |
|    1  |    QUERY BLOCK ("$QB_IDX_2")                                         |
|    2  |      HASH JOIN (INNER JOIN)                                          |
|    3  |        TABLE ACCESS ("ORDERS")                                       |
|    4  |        HASH JOIN INSTANT                                             |
|    5  |          INLINE_VIEW ("$V6")                                         |
|    6  |            QUERY BLOCK ("$QB_IDX_6")                                 |
|    7  |              GROUP HASH INSTANT                                      |
|    8  |                TABLE ACCESS ("LINEITEM")                             |
================================================================================


     1  -  TARGET : ORDERS.O_ORDERDATE, ORDERS.O_TOTALPRICE
     2  -  JOINED COLUMN : ORDERS.O_ORDERDATE, ORDERS.O_TOTALPRICE
     3  -  READ COLUMN : ORDERS.O_ORDERKEY, ORDERS.O_TOTALPRICE,
ORDERS.O_ORDERDATE
     4  -  HASH KEY : $V6.L_ORDERKEY
           READ KEY COLUMN : $V6.L_ORDERKEY
```

```
            HASH FILTER : $V6.L_ORDERKEY = ORDERS.O_ORDERKEY
          FETCH ONE ROW
     5  -  COLUMN : LINEITEM.L_ORDERKEY AS L_ORDERKEY
     6  -  TARGET : LINEITEM.L_ORDERKEY
     7  -  GROUP KEY : LINEITEM.L_ORDERKEY
          RECORD COLUMN : SUM( LINEITEM.L_QUANTITY )
          READ KEY COLUMN : LINEITEM.L_ORDERKEY
          READ RECORD COLUMN : SUM( LINEITEM.L_QUANTITY )
            PHYSICAL FILTER : SUM( LINEITEM.L_QUANTITY ) > 300
     8  -  READ COLUMN : LINEITEM.L_ORDERKEY, LINEITEM.L_QUANTITY

<<<  end print plan
```

可看到子查询unnest到INLINE_VIEW ("$V4")并成为HASH JOIN的right child参与join

UNNEST_HASH_IN(hash_bucket_count)

UNNEST_HASH_IN(hash_bucket_count) hint与UNNEST_HASH_IN hint相同区别在于它可指定hash bucket count

因此使用此hint根据hash_bucket_count生成hash的bucket

以下为使用UNNEST_HASH_IN(hash_bucket_count) hint的示例

```
\EXPLAIN PLAN
SELECT
       o_orderdate,
       o_totalprice
```

```
  FROM orders
 WHERE o_orderkey IN (
                      SELECT   /*+ UNNEST_HASH_IN(3) */
                               l_orderkey
                        FROM   lineitem
                      GROUP BY l_orderkey
                      HAVING   sum(l_quantity) > 300
                    )
;



>>>  start print plan


< Execution Plan >
=========================================================================
|  IDX  |  NODE DESCRIPTION                                             |
-------------------------------------------------------------------------
|    0  |  SELECT STATEMENT                                             |
|    1  |    QUERY BLOCK ("$QB_IDX_2")                                  |
|    2  |      HASH JOIN (INNER JOIN)                                   |
|    3  |        TABLE ACCESS ("ORDERS")                                |
|    4  |        HASH JOIN INSTANT                                      |
|    5  |          INLINE_VIEW ("$V6")                                  |
|    6  |            QUERY BLOCK ("$QB_IDX_6")                          |
|    7  |              GROUP HASH INSTANT                               |
```

```
|    8  |               TABLE ACCESS ("LINEITEM")                      |
=======================================================================


     1  -  TARGET : ORDERS.O_ORDERDATE, ORDERS.O_TOTALPRICE
     2  -  JOINED COLUMN : ORDERS.O_ORDERDATE, ORDERS.O_TOTALPRICE
     3  -  READ COLUMN : ORDERS.O_ORDERKEY, ORDERS.O_TOTALPRICE,
ORDERS.O_ORDERDATE
     4  -  HASH KEY : $V6.L_ORDERKEY
           READ KEY COLUMN : $V6.L_ORDERKEY
             HASH FILTER : $V6.L_ORDERKEY = ORDERS.O_ORDERKEY
           FETCH ONE ROW
     5  -  COLUMN : LINEITEM.L_ORDERKEY AS L_ORDERKEY
     6  -  TARGET : LINEITEM.L_ORDERKEY
     7  -  GROUP KEY : LINEITEM.L_ORDERKEY
           RECORD COLUMN : SUM( LINEITEM.L_QUANTITY )
           READ KEY COLUMN : LINEITEM.L_ORDERKEY
           READ RECORD COLUMN : SUM( LINEITEM.L_QUANTITY )
             PHYSICAL FILTER : SUM( LINEITEM.L_QUANTITY ) > 300
     8  -  READ COLUMN : LINEITEM.L_ORDERKEY, LINEITEM.L_QUANTITY


<<<  end print plan
```

Execution plan与使用UNNEST_HASH_IN hint的示例相同但使用上述hint在HASH JOIN INSTANT创建hash bucket count大小的hash bucket因此执行cost estimation时由于hash bucket count过大或过小而性能缓慢时可使用此hint进行调整

UNNEST_HASH_OUT

描述UNNEST_HASH_OUT hint时optimizer将子查询转换为保证相同结果的join语句并以hash join方式执行该join并且将unnest到表或view的子查询放在join的left

以下为使用UNNEST_HASH_OUT hint的示例

```
\EXPLAIN PLAN
SELECT
       o_orderdate,
       o_totalprice
  FROM orders
 WHERE o_orderkey IN (
                      SELECT   /*+ UNNEST_HASH_OUT */
                               l_orderkey
                        FROM   lineitem
                      GROUP BY l_orderkey
                      HAVING   sum(l_quantity) > 300
                    )
;

>>>  start print plan

< Execution Plan >
=========================================================================
|  IDX  |  NODE DESCRIPTION                                             |
```

```
-------------------------------------------------------------------------
|    0  |  SELECT STATEMENT                                             |
|    1  |    QUERY BLOCK ("$QB_IDX_2")                                  |
|    2  |      HASH JOIN (INNER JOIN)                                   |
|    3  |        INLINE_VIEW ("$V4")                                    |
|    4  |          QUERY BLOCK ("$QB_IDX_6")                            |
|    5  |            GROUP HASH INSTANT                                 |
|    6  |              TABLE ACCESS ("LINEITEM")                        |
|    7  |        HASH JOIN INSTANT                                      |
|    8  |          TABLE ACCESS ("ORDERS")                              |
=========================================================================


    1  -  TARGET : ORDERS.O_ORDERDATE, ORDERS.O_TOTALPRICE
    2  -  JOINED COLUMN : ORDERS.O_ORDERDATE, ORDERS.O_TOTALPRICE
    3  -  COLUMN : LINEITEM.L_ORDERKEY AS L_ORDERKEY
    4  -  TARGET : LINEITEM.L_ORDERKEY
    5  -  GROUP KEY : LINEITEM.L_ORDERKEY
          RECORD COLUMN : SUM( LINEITEM.L_QUANTITY )
          READ KEY COLUMN : LINEITEM.L_ORDERKEY
          READ RECORD COLUMN : SUM( LINEITEM.L_QUANTITY )
            PHYSICAL FILTER : SUM( LINEITEM.L_QUANTITY ) > 300
    6  -  READ COLUMN : LINEITEM.L_ORDERKEY, LINEITEM.L_QUANTITY
    7  -  HASH KEY : ORDERS.O_ORDERKEY
          RECORD COLUMN : ORDERS.O_ORDERDATE, ORDERS.O_TOTALPRICE
          READ KEY COLUMN : ORDERS.O_ORDERKEY, ORDERS.O_ORDERDATE,
```

```
ORDERS.O_TOTALPRICE
             HASH FILTER : ORDERS.O_ORDERKEY = $V4.L_ORDERKEY
           FETCH ONE ROW
     8   -   READ COLUMN : ORDERS.O_ORDERKEY, ORDERS.O_TOTALPRICE,
ORDERS.O_ORDERDATE


<<<  end print plan
```

可看到子查询unnest到INLINE_VIEW ("$V4")并成为HASH JOIN的left child参与join

UNNEST_HASH_OUT(hash_bucket_count)

UNNEST_HASH_OUT(hash_bucket_count) hint与UNNEST_HASH_OUT hint相同区别在于它可指定hash bucket count

因此使用此hint创建hash_bucket_count大小的hash的bucket

以下为使用UNNEST_HASH_OUT(hash_bucket_count) hint的示例

```
\EXPLAIN PLAN
SELECT
       o_orderdate,
       o_totalprice
  FROM orders
 WHERE o_orderkey IN (
                      SELECT   /*+ UNNEST_HASH_OUT(3) */
                               l_orderkey
                        FROM   lineitem
```

```
                    GROUP BY l_orderkey
                    HAVING   sum(l_quantity) > 300
                  )
;



>>>  start print plan


< Execution Plan >
============================================================================
|  IDX  |  NODE DESCRIPTION                                                |
----------------------------------------------------------------------------
|    0  |  SELECT STATEMENT                                                |
|    1  |    QUERY BLOCK ("$QB_IDX_2")                                     |
|    2  |      HASH JOIN (INNER JOIN)                                      |
|    3  |        INLINE_VIEW ("$V4")                                       |
|    4  |          QUERY BLOCK ("$QB_IDX_6")                               |
|    5  |            GROUP HASH INSTANT                                    |
|    6  |              TABLE ACCESS ("LINEITEM")                           |
|    7  |        HASH JOIN INSTANT                                         |
|    8  |          TABLE ACCESS ("ORDERS")                                 |
============================================================================


     1  -  TARGET : ORDERS.O_ORDERDATE, ORDERS.O_TOTALPRICE
     2  -  JOINED COLUMN : ORDERS.O_ORDERDATE, ORDERS.O_TOTALPRICE
```

```
     3  -  COLUMN : LINEITEM.L_ORDERKEY AS L_ORDERKEY
     4  -  TARGET : LINEITEM.L_ORDERKEY
     5  -  GROUP KEY : LINEITEM.L_ORDERKEY
           RECORD COLUMN : SUM( LINEITEM.L_QUANTITY )
           READ KEY COLUMN : LINEITEM.L_ORDERKEY
           READ RECORD COLUMN : SUM( LINEITEM.L_QUANTITY )
             PHYSICAL FILTER : SUM( LINEITEM.L_QUANTITY ) > 300
     6  -  READ COLUMN : LINEITEM.L_ORDERKEY, LINEITEM.L_QUANTITY
     7  -  HASH KEY : ORDERS.O_ORDERKEY
           RECORD COLUMN : ORDERS.O_ORDERDATE, ORDERS.O_TOTALPRICE
           READ KEY COLUMN : ORDERS.O_ORDERKEY, ORDERS.O_ORDERDATE,
ORDERS.O_TOTALPRICE
             HASH FILTER : ORDERS.O_ORDERKEY = $V4.L_ORDERKEY
           FETCH ONE ROW
     8  -  READ COLUMN : ORDERS.O_ORDERKEY, ORDERS.O_TOTALPRICE,
ORDERS.O_ORDERDATE


<<<  end print plan
```

Execution plan与UNNEST_HASH_OUT hint相同但是使用上述hint在HASH JOIN INSTANT创建hash bucket count大小的hash bucket因此执行cost estimation时由于hash bucket count过大或过小而性能缓慢时可使用此hint调整

UNNEST_MERGE

描述UNNEST_MERGE hint时optimizer将子查询转换为保证相同结果的join语句并以merge join

方式执行该join

以下为使用UNNEST_MERGE hint的示例

```
\EXPLAIN PLAN
SELECT
       o_orderdate,
       o_totalprice
  FROM orders
 WHERE o_orderkey IN (
                       SELECT   /*+ UNNEST_MERGE */
                                l_orderkey
                         FROM   lineitem
                       GROUP BY l_orderkey
                       HAVING   sum(l_quantity) > 300
                     )
;

>>>  start print plan

< Execution Plan >
=============================================================================
|  IDX  |  NODE DESCRIPTION                                                 |
-----------------------------------------------------------------------------
|    0  |  SELECT STATEMENT                                                 |
|    1  |    QUERY BLOCK ("$QB_IDX_2")                                      |
```

```
|    2  |      MERGE JOIN (INNER JOIN)                                    |
|    3  |        INDEX ACCESS ("ORDERS", "ORDERS_PK_INDEX")               |
|    4  |        SORT JOIN INSTANT                                        |
|    5  |          INLINE_VIEW ("$V6")                                    |
|    6  |            QUERY BLOCK ("$QB_IDX_6")                            |
|    7  |              GROUP HASH INSTANT                                 |
|    8  |                TABLE ACCESS ("LINEITEM")                        |
=====================================================================


    1  -  TARGET : ORDERS.O_ORDERDATE, ORDERS.O_TOTALPRICE
    2  -  JOINED COLUMN : ORDERS.O_ORDERDATE, ORDERS.O_TOTALPRICE
            ON FILTER (Equi) : ORDERS.O_ORDERKEY = $V6.L_ORDERKEY
    3  -  READ INDEX COLUMN : ORDERS.O_ORDERKEY
          READ TABLE COLUMN : ORDERS.O_TOTALPRICE, ORDERS.O_ORDERDATE
    4  -  SORT KEY : "$V6.L_ORDERKEY ASC NULLS LAST"
          READ KEY COLUMN : $V6.L_ORDERKEY
            MIN RANGE : $V6.L_ORDERKEY >= {ORDERS.O_ORDERKEY}
            MAX RANGE : $V6.L_ORDERKEY IS NOT NULL
    5  -  COLUMN : LINEITEM.L_ORDERKEY AS L_ORDERKEY
    6  -  TARGET : LINEITEM.L_ORDERKEY
    7  -  GROUP KEY : LINEITEM.L_ORDERKEY
          RECORD COLUMN : SUM( LINEITEM.L_QUANTITY )
          READ KEY COLUMN : LINEITEM.L_ORDERKEY
          READ RECORD COLUMN : SUM( LINEITEM.L_QUANTITY )
            PHYSICAL FILTER : SUM( LINEITEM.L_QUANTITY ) > 300
```

```
    8  -  READ COLUMN : LINEITEM.L_ORDERKEY, LINEITEM.L_QUANTITY


<<<  end print plan
```

可看到子查询unnest到INLINE_VIEW ("$V6")后参与MERGE JOIN

UNNEST_MERGE_IN

描述UNNEST_MERGE_IN hint时optimizer将子查询转换为保证相同结果的join语句并以merge join方式执行该join并且将unnest到表或view的子查询放在join的right

以下为使用UNNEST_MERGE_IN hint的示例

```
\EXPLAIN PLAN
SELECT
       o_orderdate,
       o_totalprice
  FROM orders
 WHERE o_orderkey IN (
                      SELECT   /*+ UNNEST_MERGE_IN */
                               l_orderkey
                        FROM   lineitem
                      GROUP BY l_orderkey
                      HAVING   sum(l_quantity) > 300
                    )
;
```

```
>>>  start print plan


< Execution Plan >

=============================================================================
|  IDX  |  NODE DESCRIPTION                                                 |
-----------------------------------------------------------------------------
|    0  |  SELECT STATEMENT                                                 |
|    1  |    QUERY BLOCK ("$QB_IDX_2")                                      |
|    2  |      MERGE JOIN (INNER JOIN)                                      |
|    3  |        INDEX ACCESS ("ORDERS", "ORDERS_PK_INDEX")                 |
|    4  |        SORT JOIN INSTANT                                          |
|    5  |          INLINE_VIEW ("$V6")                                      |
|    6  |            QUERY BLOCK ("$QB_IDX_6")                              |
|    7  |              GROUP HASH INSTANT                                   |
|    8  |                TABLE ACCESS ("LINEITEM")                          |
=============================================================================


     1  -  TARGET : ORDERS.O_ORDERDATE, ORDERS.O_TOTALPRICE
     2  -  JOINED COLUMN : ORDERS.O_ORDERDATE, ORDERS.O_TOTALPRICE
             ON FILTER (Equi) : ORDERS.O_ORDERKEY = $V6.L_ORDERKEY
     3  -  READ INDEX COLUMN : ORDERS.O_ORDERKEY
           READ TABLE COLUMN : ORDERS.O_TOTALPRICE, ORDERS.O_ORDERDATE
     4  -  SORT KEY : "$V6.L_ORDERKEY ASC NULLS LAST"
           READ KEY COLUMN : $V6.L_ORDERKEY
```

```
              MIN RANGE : $V6.L_ORDERKEY >= {ORDERS.O_ORDERKEY}
              MAX RANGE : $V6.L_ORDERKEY IS NOT NULL
      5   -   COLUMN : LINEITEM.L_ORDERKEY AS L_ORDERKEY
      6   -   TARGET : LINEITEM.L_ORDERKEY
      7   -   GROUP KEY : LINEITEM.L_ORDERKEY
              RECORD COLUMN : SUM( LINEITEM.L_QUANTITY )
              READ KEY COLUMN : LINEITEM.L_ORDERKEY
              READ RECORD COLUMN : SUM( LINEITEM.L_QUANTITY )
                PHYSICAL FILTER : SUM( LINEITEM.L_QUANTITY ) > 300
      8   -   READ COLUMN : LINEITEM.L_ORDERKEY, LINEITEM.L_QUANTITY


<<<  end print plan
```

可看到子查询unnest到INLINE_VIEW ("$V6")并成为MERGE JOIN的right child参与join

UNNEST_MERGE_OUT

描述UNNEST_MERGE_OUT hint时optimizer将子查询转换为保证相同结果的join语句并以merge join方式执行该join并且将unnest到表或view的子查询放在join的left

以下为使用UNNEST_MERGE_OUT hint的示例

```
\EXPLAIN PLAN
SELECT
       o_orderdate,
       o_totalprice
  FROM orders
```

```
 WHERE o_orderkey IN (
                      SELECT   /*+ UNNEST_MERGE_OUT */
                               l_orderkey
                        FROM   lineitem
                      GROUP BY l_orderkey
                      HAVING   sum(l_quantity) > 300
                    )
;



>>>  start print plan


< Execution Plan >
=========================================================================
|  IDX  |  NODE DESCRIPTION                                             |
-------------------------------------------------------------------------
|    0  |  SELECT STATEMENT                                             |
|    1  |    QUERY BLOCK ("$QB_IDX_2")                                  |
|    2  |      MERGE JOIN (INNER JOIN)                                  |
|    3  |        SORT JOIN INSTANT                                      |
|    4  |          INLINE_VIEW ("$V5")                                  |
|    5  |            QUERY BLOCK ("$QB_IDX_6")                          |
|    6  |              GROUP HASH INSTANT                               |
|    7  |                TABLE ACCESS ("LINEITEM")                      |
|    8  |        INDEX ACCESS ("ORDERS", "ORDERS_PK_INDEX")             |
```

```
================================================================================


     1  -  TARGET : ORDERS.O_ORDERDATE, ORDERS.O_TOTALPRICE

     2  -  JOINED COLUMN : ORDERS.O_ORDERDATE, ORDERS.O_TOTALPRICE

           ON FILTER (Equi) : $V5.L_ORDERKEY = ORDERS.O_ORDERKEY

     3  -  SORT KEY : "$V5.L_ORDERKEY ASC NULLS LAST"

           READ KEY COLUMN : $V5.L_ORDERKEY

     4  -  COLUMN : LINEITEM.L_ORDERKEY AS L_ORDERKEY

     5  -  TARGET : LINEITEM.L_ORDERKEY

     6  -  GROUP KEY : LINEITEM.L_ORDERKEY

           RECORD COLUMN : SUM( LINEITEM.L_QUANTITY )

           READ KEY COLUMN : LINEITEM.L_ORDERKEY

           READ RECORD COLUMN : SUM( LINEITEM.L_QUANTITY )

             PHYSICAL FILTER : SUM( LINEITEM.L_QUANTITY ) > 300

     7  -  READ COLUMN : LINEITEM.L_ORDERKEY, LINEITEM.L_QUANTITY

     8  -  READ INDEX COLUMN : ORDERS.O_ORDERKEY

           READ TABLE COLUMN : ORDERS.O_TOTALPRICE, ORDERS.O_ORDERDATE

             MIN RANGE : ORDERS.O_ORDERKEY >= {$V5.L_ORDERKEY}

             MAX RANGE : ORDERS.O_ORDERKEY IS NOT NULL


<<<  end print plan
```

可看到子查询unnest到INLINE_VIEW ("$V5")并成为MERGE JOIN的left child参与join

NL_SJ

描述NL_SJ hint时optimizer以semi join形式unnest子查询以nested loop方式执行该semi join

与UNNEST_NL hint的运行方式相同但UNNEST_NL hint均适用于semi join和anti-semi join而NL_SJ hint仅适用于semi join

因此在以anti-semi join被unnest的子查询描述此hint时则optimizer忽略其

以下为使用 NL_SJ hint的示例

```
\EXPLAIN PLAN
SELECT
       o_orderdate,
       o_totalprice
  FROM orders
 WHERE o_orderkey IN (
                      SELECT   /*+ NL_SJ */
                               l_orderkey
                        FROM   lineitem
                      GROUP BY l_orderkey
                      HAVING   sum(l_quantity) > 300
                    )
;



>>>  start print plan
```

```
< Execution Plan >

================================================================================
|  IDX  |  NODE DESCRIPTION                                                    |
--------------------------------------------------------------------------------
|    0  |  SELECT STATEMENT                                                    |
|    1  |    QUERY BLOCK ("$QB_IDX_2")                                         |
|    2  |      NESTED JOIN (INNER JOIN)                                        |
|    3  |        TABLE ACCESS ("ORDERS")                                       |
|    4  |        INLINE_VIEW ("$V5")                                           |
|    5  |          QUERY BLOCK ("$QB_IDX_6")                                   |
|    6  |            GROUP                                                     |
|    7  |              INDEX ACCESS ("LINEITEM", "LINEITEM_PK_INDEX")          |
================================================================================


     1  -  TARGET : ORDERS.O_ORDERDATE, ORDERS.O_TOTALPRICE
     2  -  JOINED COLUMN : ORDERS.O_ORDERDATE, ORDERS.O_TOTALPRICE
     3  -  READ COLUMN : ORDERS.O_ORDERKEY, ORDERS.O_TOTALPRICE,
ORDERS.O_ORDERDATE
     4  -  COLUMN : LINEITEM.L_ORDERKEY AS L_ORDERKEY
     5  -  TARGET : LINEITEM.L_ORDERKEY
     6  -  GROUP KEY : LINEITEM.L_ORDERKEY
           RECORD COLUMN : SUM( LINEITEM.L_QUANTITY )
             LOGICAL FILTER : SUM( LINEITEM.L_QUANTITY ) > 300
     7  -  READ INDEX COLUMN : LINEITEM.L_ORDERKEY
```

```
            READ TABLE COLUMN : LINEITEM.L_QUANTITY
              MIN RANGE : LINEITEM.L_ORDERKEY = {ORDERS.O_ORDERKEY}
              MAX RANGE : LINEITEM.L_ORDERKEY = {ORDERS.O_ORDERKEY}


<<<  end print plan
```

上述示例中应用NL_SJ hint后子查询unnest到semi join并以nested loop join执行并且由于o_orderkey和l_orderkey均为primary key因此semi join变更为inner join

以下为在只能以anti-semi join形式unnest的子查询使用NL_SJ hint的示例

```
\EXPLAIN PLAN
SELECT
       o_orderdate,
       o_totalprice
  FROM orders
 WHERE o_orderkey NOT IN (
                           SELECT   /*+ NL_SJ */
                                    l_orderkey
                             FROM   lineitem
                            GROUP BY l_orderkey
                            HAVING   sum(l_quantity) > 300
                          )
;


>>>  start print plan
```

```
< Execution Plan >
=============================================================================
|  IDX  |  NODE DESCRIPTION                                                  |
-----------------------------------------------------------------------------
|    0  |  SELECT STATEMENT                                                  |
|    1  |    QUERY BLOCK ("$QB_IDX_2")                                       |
|    2  |      HASH JOIN (ANTI SEMI)                                         |
|    3  |        TABLE ACCESS ("ORDERS")                                     |
|    4  |        HASH JOIN INSTANT (UNIQUE)                                  |
|    5  |          INLINE_VIEW ("$V6")                                       |
|    6  |            QUERY BLOCK ("$QB_IDX_6")                               |
|    7  |              GROUP HASH INSTANT                                    |
|    8  |                TABLE ACCESS ("LINEITEM")                           |
=============================================================================


     1  -  TARGET : ORDERS.O_ORDERDATE, ORDERS.O_TOTALPRICE
     2  -  JOINED COLUMN : ORDERS.O_ORDERDATE, ORDERS.O_TOTALPRICE
     3  -  READ COLUMN : ORDERS.O_ORDERKEY, ORDERS.O_TOTALPRICE,
ORDERS.O_ORDERDATE
     4  -  HASH KEY : $V6.L_ORDERKEY
           READ KEY COLUMN : $V6.L_ORDERKEY
             HASH FILTER : $V6.L_ORDERKEY = ORDERS.O_ORDERKEY
           FETCH ONE ROW
     5  -  COLUMN : LINEITEM.L_ORDERKEY AS L_ORDERKEY
```

```
    6  -  TARGET : LINEITEM.L_ORDERKEY
    7  -  GROUP KEY : LINEITEM.L_ORDERKEY
          RECORD COLUMN : SUM( LINEITEM.L_QUANTITY )
          READ KEY COLUMN : LINEITEM.L_ORDERKEY
          READ RECORD COLUMN : SUM( LINEITEM.L_QUANTITY )
            PHYSICAL FILTER : SUM( LINEITEM.L_QUANTITY ) > 300
    8  -  READ COLUMN : LINEITEM.L_ORDERKEY, LINEITEM.L_QUANTITY


<<<  end print plan
```

以anti-semi join被unnest后以hash join方式执行因此可看出忽略了hint

NL_ISJ

描述NL_ISJ hint时optimizer以inverted semi join形式unnest子查询并以nested loop方式执行其semi join由于是Inverted semi join因此将unnest到表或view的子查询放在join的left

与UNNEST_NL_OUT hint的运行方式相同但UNNEST_NL_OUT hint均适用于semi joinanti-semi join而NL_ISJ hint仅适用于semi join

因此在以anti-semi join unnest的子查询描述上述hintoptimizer忽略其

以下为使用NL_ISJ hint的示例

```
\EXPLAIN PLAN
SELECT
       o_orderdate,
       o_totalprice
```

```
  FROM orders
 WHERE o_orderkey IN (
                      SELECT   /*+ NL_ISJ */
                               l_orderkey
                        FROM   lineitem
                      GROUP BY l_orderkey
                      HAVING   sum(l_quantity) > 300
                    )
;



>>>  start print plan


< Execution Plan >
=============================================================================
|  IDX  |  NODE DESCRIPTION                                                 |
-----------------------------------------------------------------------------
|    0  |  SELECT STATEMENT                                                 |
|    1  |    QUERY BLOCK ("$QB_IDX_2")                                      |
|    2  |      NESTED JOIN (INNER JOIN)                                     |
|    3  |        INLINE_VIEW ("$V4")                                        |
|    4  |          QUERY BLOCK ("$QB_IDX_6")                                |
|    5  |            GROUP HASH INSTANT                                     |
|    6  |              TABLE ACCESS ("LINEITEM")                            |
|    7  |        INDEX ACCESS ("ORDERS", "ORDERS_PK_INDEX")                 |
```

```
========================================================================


     1  -  TARGET : ORDERS.O_ORDERDATE, ORDERS.O_TOTALPRICE

     2  -  JOINED COLUMN : ORDERS.O_ORDERDATE, ORDERS.O_TOTALPRICE

     3  -  COLUMN : LINEITEM.L_ORDERKEY AS L_ORDERKEY

     4  -  TARGET : LINEITEM.L_ORDERKEY

     5  -  GROUP KEY : LINEITEM.L_ORDERKEY

           RECORD COLUMN : SUM( LINEITEM.L_QUANTITY )

           READ KEY COLUMN : LINEITEM.L_ORDERKEY

           READ RECORD COLUMN : SUM( LINEITEM.L_QUANTITY )

             PHYSICAL FILTER : SUM( LINEITEM.L_QUANTITY ) > 300

     6  -  READ COLUMN : LINEITEM.L_ORDERKEY, LINEITEM.L_QUANTITY

     7  -  READ INDEX COLUMN : ORDERS.O_ORDERKEY

           READ TABLE COLUMN : ORDERS.O_TOTALPRICE, ORDERS.O_ORDERDATE

             MIN RANGE : ORDERS.O_ORDERKEY = {$V4.L_ORDERKEY}

             MAX RANGE : ORDERS.O_ORDERKEY = {$V4.L_ORDERKEY}

           FETCH ONE ROW


<<<  end print plan
```

上述示例中应用NL_ISJ hint后子查询unnest到inverted semi join并以nested loop join执行并且o_orderkey和l_orderkey均为primary key因此semi join变更为inner join

以下为在仅可以以anti-semi join形式unnest的子查询使用NL_ISJ hint的示例

```
\EXPLAIN PLAN
```

```
SELECT
       o_orderdate,
       o_totalprice
  FROM orders
 WHERE o_orderkey NOT IN (
                           SELECT   /*+ NL_ISJ */
                                    l_orderkey
                             FROM   lineitem
                            GROUP BY l_orderkey
                            HAVING   sum(l_quantity) > 300
                          )
;


>>>  start print plan


< Execution Plan >
=========================================================================
|  IDX  |  NODE DESCRIPTION                                             |
-------------------------------------------------------------------------
|    0  |  SELECT STATEMENT                                             |
|    1  |    QUERY BLOCK ("$QB_IDX_2")                                  |
|    2  |      HASH JOIN (ANTI SEMI)                                    |
|    3  |        TABLE ACCESS ("ORDERS")                                |
|    4  |        HASH JOIN INSTANT (UNIQUE)                             |
|    5  |          INLINE_VIEW ("$V6")                                  |
```

```
|    6  |            QUERY BLOCK ("$QB_IDX_6")                          |
|    7  |              GROUP HASH INSTANT                                |
|    8  |                TABLE ACCESS ("LINEITEM")                       |
=========================================================================


     1  -  TARGET : ORDERS.O_ORDERDATE, ORDERS.O_TOTALPRICE
     2  -  JOINED COLUMN : ORDERS.O_ORDERDATE, ORDERS.O_TOTALPRICE
     3  -  READ COLUMN : ORDERS.O_ORDERKEY, ORDERS.O_TOTALPRICE,
ORDERS.O_ORDERDATE
     4  -  HASH KEY : $V6.L_ORDERKEY
           READ KEY COLUMN : $V6.L_ORDERKEY
             HASH FILTER : $V6.L_ORDERKEY = ORDERS.O_ORDERKEY
           FETCH ONE ROW
     5  -  COLUMN : LINEITEM.L_ORDERKEY AS L_ORDERKEY
     6  -  TARGET : LINEITEM.L_ORDERKEY
     7  -  GROUP KEY : LINEITEM.L_ORDERKEY
           RECORD COLUMN : SUM( LINEITEM.L_QUANTITY )
           READ KEY COLUMN : LINEITEM.L_ORDERKEY
           READ RECORD COLUMN : SUM( LINEITEM.L_QUANTITY )
             PHYSICAL FILTER : SUM( LINEITEM.L_QUANTITY ) > 300
     8  -  READ COLUMN : LINEITEM.L_ORDERKEY, LINEITEM.L_QUANTITY


<<<  end print plan
```

以anti-semi join进行unnest后以hash join方式执行unnest的子查询在right因此hint已被忽略

INL_SJ

描述INL_SJ hint时optimizer以semi join形式unnest子查询并以instant nested loop方式执行其semi join

与UNNEST_INL hint的运行方式相同但UNNEST_INL hint均适用于semi joinanti-semi join而INL_SJ hint仅适用于semi join

因此在以anti-semi join unnest的子查询描述上述hintoptimizer忽略其

以下为使用INL_SJ hint的示例

```
\EXPLAIN PLAN
SELECT
       o_orderdate,
       o_totalprice
  FROM orders
 WHERE o_orderkey IN (
                      SELECT   /*+ INL_SJ */
                               l_orderkey
                        FROM   lineitem
                      GROUP BY l_orderkey
                      HAVING   sum(l_quantity) > 300
                    )
;
```

```
>>>  start print plan


< Execution Plan >

=============================================================================

|  IDX  |  NODE DESCRIPTION                                                  |

-----------------------------------------------------------------------------

|    0  |  SELECT STATEMENT                                                  |

|    1  |    QUERY BLOCK ("$QB_IDX_2")                                       |

|    2  |      NESTED JOIN (INNER JOIN)                                      |

|    3  |        TABLE ACCESS ("ORDERS")                                     |

|    4  |        SORT JOIN INSTANT                                           |

|    5  |          INLINE_VIEW ("$V6")                                       |

|    6  |            QUERY BLOCK ("$QB_IDX_6")                               |

|    7  |              GROUP HASH INSTANT                                    |

|    8  |                TABLE ACCESS ("LINEITEM")                           |

=============================================================================


     1  -  TARGET : ORDERS.O_ORDERDATE, ORDERS.O_TOTALPRICE

     2  -  JOINED COLUMN : ORDERS.O_ORDERDATE, ORDERS.O_TOTALPRICE

     3  -  READ COLUMN : ORDERS.O_ORDERKEY, ORDERS.O_TOTALPRICE,

ORDERS.O_ORDERDATE

     4  -  SORT KEY : "$V6.L_ORDERKEY ASC NULLS LAST"

           READ KEY COLUMN : $V6.L_ORDERKEY

             MIN RANGE : $V6.L_ORDERKEY = {ORDERS.O_ORDERKEY}

             MAX RANGE : $V6.L_ORDERKEY = {ORDERS.O_ORDERKEY}
```

```
    5  -  COLUMN : LINEITEM.L_ORDERKEY AS L_ORDERKEY
    6  -  TARGET : LINEITEM.L_ORDERKEY
    7  -  GROUP KEY : LINEITEM.L_ORDERKEY
          RECORD COLUMN : SUM( LINEITEM.L_QUANTITY )
          READ KEY COLUMN : LINEITEM.L_ORDERKEY
          READ RECORD COLUMN : SUM( LINEITEM.L_QUANTITY )
            PHYSICAL FILTER : SUM( LINEITEM.L_QUANTITY ) > 300
    8  -  READ COLUMN : LINEITEM.L_ORDERKEY, LINEITEM.L_QUANTITY


<<<  end print plan
```

上述示例中应用INL_SJ hint后子查询unnest到semi join并以instant nested loop join执行并且o_orderkey和l_orderkey均为primary key因此semi join变更为inner join

以下为在仅可以以anti-semi join形式unnest的子查询使用INL_SJ hint的示例

```
\EXPLAIN PLAN
SELECT
       o_orderdate,
       o_totalprice
  FROM orders
 WHERE o_orderkey NOT IN (
                          SELECT   /*+ INL_SJ */
                                   l_orderkey
                            FROM   lineitem
                           GROUP BY l_orderkey
```

```
                          HAVING   sum(l_quantity) > 300
                        )
;


>>>  start print plan


< Execution Plan >
============================================================================
|  IDX  |  NODE DESCRIPTION                                                  |
----------------------------------------------------------------------------
|    0  |  SELECT STATEMENT                                                  |
|    1  |    QUERY BLOCK ("$QB_IDX_2")                                       |
|    2  |      HASH JOIN (ANTI SEMI)                                         |
|    3  |        TABLE ACCESS ("ORDERS")                                     |
|    4  |        HASH JOIN INSTANT (UNIQUE)                                  |
|    5  |          INLINE_VIEW ("$V6")                                       |
|    6  |            QUERY BLOCK ("$QB_IDX_6")                               |
|    7  |              GROUP HASH INSTANT                                    |
|    8  |                TABLE ACCESS ("LINEITEM")                           |
============================================================================


     1  -  TARGET : ORDERS.O_ORDERDATE, ORDERS.O_TOTALPRICE
     2  -  JOINED COLUMN : ORDERS.O_ORDERDATE, ORDERS.O_TOTALPRICE
     3  -  READ COLUMN : ORDERS.O_ORDERKEY, ORDERS.O_TOTALPRICE,
ORDERS.O_ORDERDATE
```

```
    4  -  HASH KEY : $V6.L_ORDERKEY
          READ KEY COLUMN : $V6.L_ORDERKEY
            HASH FILTER : $V6.L_ORDERKEY = ORDERS.O_ORDERKEY
          FETCH ONE ROW
    5  -  COLUMN : LINEITEM.L_ORDERKEY AS L_ORDERKEY
    6  -  TARGET : LINEITEM.L_ORDERKEY
    7  -  GROUP KEY : LINEITEM.L_ORDERKEY
          RECORD COLUMN : SUM( LINEITEM.L_QUANTITY )
          READ KEY COLUMN : LINEITEM.L_ORDERKEY
          READ RECORD COLUMN : SUM( LINEITEM.L_QUANTITY )
            PHYSICAL FILTER : SUM( LINEITEM.L_QUANTITY ) > 300
    8  -  READ COLUMN : LINEITEM.L_ORDERKEY, LINEITEM.L_QUANTITY

<<<  end print plan
```

以Anti-semi join进行unnest后以hash join方式执行因此hint已被忽略

MERGE_SJ

描述MERGE_SJ hint时optimizer以semi join形式unnest子查询并以merge join方式执行其semi join

与UNNEST_MERGE hint的运行方式相同但UNNEST_MERGE hint均适用于semi joinanti-semi join而MERGE_SJ hint仅适用于semi join

因此在以anti-semi join unnest的子查询描述上述hintoptimizer忽略其

以下为使用MERGE_SJ hint的示例

```
\EXPLAIN PLAN
SELECT
       o_orderdate,
       o_totalprice
  FROM orders
 WHERE o_orderkey IN (
                      SELECT   /*+ MERGE_SJ */
                               l_orderkey
                        FROM   lineitem
                      GROUP BY l_orderkey
                      HAVING   sum(l_quantity) > 300
                    )
;



>>>  start print plan


< Execution Plan >
=========================================================================
|  IDX  |  NODE DESCRIPTION                                             |
-------------------------------------------------------------------------
|    0  |  SELECT STATEMENT                                             |
|    1  |    QUERY BLOCK ("$QB_IDX_2")                                  |
|    2  |      MERGE JOIN (INNER JOIN)                                  |
|    3  |        INDEX ACCESS ("ORDERS", "ORDERS_PK_INDEX")             |
```

```
|    4  |        SORT JOIN INSTANT                                          |
|    5  |          INLINE_VIEW ("$V6")                                      |
|    6  |            QUERY BLOCK ("$QB_IDX_6")                              |
|    7  |              GROUP HASH INSTANT                                   |
|    8  |                TABLE ACCESS ("LINEITEM")                          |
=======================================================================


    1  -  TARGET : ORDERS.O_ORDERDATE, ORDERS.O_TOTALPRICE
    2  -  JOINED COLUMN : ORDERS.O_ORDERDATE, ORDERS.O_TOTALPRICE
            ON FILTER (Equi) : ORDERS.O_ORDERKEY = $V6.L_ORDERKEY
    3  -  READ INDEX COLUMN : ORDERS.O_ORDERKEY
          READ TABLE COLUMN : ORDERS.O_TOTALPRICE, ORDERS.O_ORDERDATE
    4  -  SORT KEY : "$V6.L_ORDERKEY ASC NULLS LAST"
          READ KEY COLUMN : $V6.L_ORDERKEY
            MIN RANGE : $V6.L_ORDERKEY >= {ORDERS.O_ORDERKEY}
            MAX RANGE : $V6.L_ORDERKEY IS NOT NULL
    5  -  COLUMN : LINEITEM.L_ORDERKEY AS L_ORDERKEY
    6  -  TARGET : LINEITEM.L_ORDERKEY
    7  -  GROUP KEY : LINEITEM.L_ORDERKEY
          RECORD COLUMN : SUM( LINEITEM.L_QUANTITY )
          READ KEY COLUMN : LINEITEM.L_ORDERKEY
          READ RECORD COLUMN : SUM( LINEITEM.L_QUANTITY )
            PHYSICAL FILTER : SUM( LINEITEM.L_QUANTITY ) > 300
    8  -  READ COLUMN : LINEITEM.L_ORDERKEY, LINEITEM.L_QUANTITY
```

```
<<<  end print plan
```

上述示例中应用MERGE_SJ hint后子查询unnest到semi join并以merge join执行并且o_orderkey和l_orderkey均为primary key因此semi join变更为inner join

以下为在仅可以以anti-semi join形式unnest的子查询使用MERGE_SJ hint的示例

```
\EXPLAIN PLAN
SELECT
       o_orderdate,
       o_totalprice
  FROM orders
 WHERE o_orderkey NOT IN (
                           SELECT   /*+ MERGE_SJ */
                                    l_orderkey
                             FROM   lineitem
                            GROUP BY l_orderkey
                            HAVING   sum(l_quantity) > 300
                          )
;

>>>  start print plan

< Execution Plan >
=========================================================================
|  IDX  |  NODE DESCRIPTION                                              |
```

```
-------------------------------------------------------------------------
|    0  |  SELECT STATEMENT                                             |
|    1  |    QUERY BLOCK ("$QB_IDX_2")                                  |
|    2  |      HASH JOIN (ANTI SEMI)                                    |
|    3  |        TABLE ACCESS ("ORDERS")                                |
|    4  |        HASH JOIN INSTANT (UNIQUE)                             |
|    5  |          INLINE_VIEW ("$V6")                                  |
|    6  |            QUERY BLOCK ("$QB_IDX_6")                          |
|    7  |              GROUP HASH INSTANT                               |
|    8  |                TABLE ACCESS ("LINEITEM")                      |
=========================================================================


     1  -  TARGET : ORDERS.O_ORDERDATE, ORDERS.O_TOTALPRICE
     2  -  JOINED COLUMN : ORDERS.O_ORDERDATE, ORDERS.O_TOTALPRICE
     3  -  READ COLUMN : ORDERS.O_ORDERKEY, ORDERS.O_TOTALPRICE,
ORDERS.O_ORDERDATE
     4  -  HASH KEY : $V6.L_ORDERKEY
           READ KEY COLUMN : $V6.L_ORDERKEY
             HASH FILTER : $V6.L_ORDERKEY = ORDERS.O_ORDERKEY
           FETCH ONE ROW
     5  -  COLUMN : LINEITEM.L_ORDERKEY AS L_ORDERKEY
     6  -  TARGET : LINEITEM.L_ORDERKEY
     7  -  GROUP KEY : LINEITEM.L_ORDERKEY
           RECORD COLUMN : SUM( LINEITEM.L_QUANTITY )
           READ KEY COLUMN : LINEITEM.L_ORDERKEY
```

```
            READ RECORD COLUMN : SUM( LINEITEM.L_QUANTITY )
              PHYSICAL FILTER : SUM( LINEITEM.L_QUANTITY ) > 300
      8   -   READ COLUMN : LINEITEM.L_ORDERKEY, LINEITEM.L_QUANTITY


<<<  end print plan
```

以Anti-semi join进行unnest后以hash join方式执行因此hint已被忽略

HASH_SJ

描述HASH_SJ hint时optimizer以semi join形式unnest子查询并以hash join方式执行其semi join

与UNNEST_HASH hint的运行方式相同但UNNEST_HASH hint均适用于semi joinanti-semi join而HASH_SJ hint仅适用于semi join

因此在以anti-semi join unnest的子查询描述上述hintoptimizer忽略其

以下为使用HASH_SJ hint的示例

```
\EXPLAIN PLAN
SELECT
       o_orderdate,
       o_totalprice
  FROM orders
 WHERE o_orderkey IN (
                      SELECT   /*+ HASH_SJ */
                               l_orderkey
                        FROM   lineitem
```

```
                    GROUP BY l_orderkey
                    HAVING   sum(l_quantity) > 300
                  )
;

no rows selected.

>>>  start print plan

< Execution Plan >
=============================================================================
|  IDX  |  NODE DESCRIPTION                                                 |
-----------------------------------------------------------------------------
|    0  |  SELECT STATEMENT                                                 |
|    1  |    QUERY BLOCK ("$QB_IDX_2")                                      |
|    2  |      HASH JOIN (INNER JOIN)                                       |
|    3  |        TABLE ACCESS ("ORDERS")                                    |
|    4  |        HASH JOIN INSTANT                                          |
|    5  |          INLINE_VIEW ("$V6")                                      |
|    6  |            QUERY BLOCK ("$QB_IDX_6")                              |
|    7  |              GROUP HASH INSTANT                                   |
|    8  |                TABLE ACCESS ("LINEITEM")                          |
=============================================================================

    1  -  TARGET : ORDERS.O_ORDERDATE, ORDERS.O_TOTALPRICE
```

```
     2  -  JOINED COLUMN : ORDERS.O_ORDERDATE, ORDERS.O_TOTALPRICE
     3  -  READ COLUMN : ORDERS.O_ORDERKEY, ORDERS.O_TOTALPRICE,
ORDERS.O_ORDERDATE
     4  -  HASH KEY : $V6.L_ORDERKEY
           READ KEY COLUMN : $V6.L_ORDERKEY
             HASH FILTER : $V6.L_ORDERKEY = ORDERS.O_ORDERKEY
           FETCH ONE ROW
     5  -  COLUMN : LINEITEM.L_ORDERKEY AS L_ORDERKEY
     6  -  TARGET : LINEITEM.L_ORDERKEY
     7  -  GROUP KEY : LINEITEM.L_ORDERKEY
           RECORD COLUMN : SUM( LINEITEM.L_QUANTITY )
           READ KEY COLUMN : LINEITEM.L_ORDERKEY
           READ RECORD COLUMN : SUM( LINEITEM.L_QUANTITY )
             PHYSICAL FILTER : SUM( LINEITEM.L_QUANTITY ) > 300
     8  -  READ COLUMN : LINEITEM.L_ORDERKEY, LINEITEM.L_QUANTITY


<<<  end print plan
```

上述示例中应用HASH_SJ hint后子查询unnest到semi join并以hash join执行并且o_orderkey和l_orderkey均为primary key因此semi join变更为inner join

以下为在仅可以以anti-semi join形式unnest的子查询使用HASH_SJ hint的示例

```
\EXPLAIN PLAN
SELECT
       o_orderdate,
```

```
       o_totalprice
  FROM orders
 WHERE o_orderkey NOT IN (
                           SELECT   /*+ HASH_SJ */
                                    l_orderkey
                             FROM   lineitem
                            WHERE   l_shipdate >= date '1996-01-01'
                              AND   l_shipdate < date '1996-02-01'
                              AND   l_commitdate < l_receiptdate
                            GROUP BY l_orderkey
                            HAVING   sum(l_quantity) > 300
                          )
   AND  o_orderdate >= date '1996-01-01'
   AND  o_orderdate < date '1996-02-01'
   AND  o_comment like '%Customer%Complaints%'
   AND  o_orderstatus = 'F'
;


>>>  start print plan


< Execution Plan >
=============================================================================
|  IDX  |  NODE DESCRIPTION                                                 |
-----------------------------------------------------------------------------
|    0  |  SELECT STATEMENT                                                 |
```

```
|    1  |    QUERY BLOCK ("$QB_IDX_2")                                    |
|    2  |      NESTED JOIN (ANTI SEMI)                                    |
|    3  |        TABLE ACCESS ("ORDERS")                                  |
|    4  |        INLINE_VIEW ("$V5")                                      |
|    5  |          QUERY BLOCK ("$QB_IDX_6")                              |
|    6  |            GROUP                                                |
|    7  |              INDEX ACCESS ("LINEITEM", "LINEITEM_PK_INDEX")     |
=======================================================================


     1  -  TARGET : ORDERS.O_ORDERDATE, ORDERS.O_TOTALPRICE
     2  -  JOINED COLUMN : ORDERS.O_ORDERDATE, ORDERS.O_TOTALPRICE
     3  -  READ COLUMN : ORDERS.O_ORDERKEY, ORDERS.O_ORDERSTATUS,
ORDERS.O_TOTALPRICE, ORDERS.O_ORDERDATE, ORDERS.O_COMMENT
             PHYSICAL FILTER : ORDERS.O_ORDERSTATUS = 'F' AND
ORDERS.O_ORDERDATE >= DATE'1996-01-01' AND ORDERS.O_ORDERDATE < DATE'1996-
02-01'
             LOGICAL FILTER : ORDERS.O_COMMENT LIKE
'%Customer%Complaints%'
     4  -  COLUMN : LINEITEM.L_ORDERKEY AS L_ORDERKEY
     5  -  TARGET : LINEITEM.L_ORDERKEY
     6  -  GROUP KEY : LINEITEM.L_ORDERKEY
           RECORD COLUMN : SUM( LINEITEM.L_QUANTITY )
             LOGICAL FILTER : SUM( LINEITEM.L_QUANTITY ) > 300
     7  -  READ INDEX COLUMN : LINEITEM.L_ORDERKEY
           READ TABLE COLUMN : LINEITEM.L_QUANTITY, LINEITEM.L_SHIPDATE,
```

```
LINEITEM.L_COMMITDATE, LINEITEM.L_RECEIPTDATE
             MIN RANGE : LINEITEM.L_ORDERKEY = {ORDERS.O_ORDERKEY}
             MAX RANGE : LINEITEM.L_ORDERKEY = {ORDERS.O_ORDERKEY}
             PHYSICAL TABLE FILTER : LINEITEM.L_SHIPDATE >= DATE'1996-01-
01' AND LINEITEM.L_SHIPDATE < DATE'1996-02-01' AND
LINEITEM.L_RECEIPTDATE > LINEITEM.L_COMMITDATE


<<<  end print plan
```

以anti-semi join进行unnest后以nested loop join方式执行因此hint已被忽略

HASH_ISJ

描述HASH_ISJ hint时optimizer以inverted semi join形式unnest子查询并以hash join方式执行其semi join

与UNNEST_HASH_OUT hint的运行方式相同但UNNEST_HASH_OUT hint均适用于semi joinanti-semi join而HASH_ISJ hint仅适用于semi join

因此在以anti-semi join unnest的子查询描述上述hintoptimizer忽略其

以下为使用HASH_ISJ hint的示例

```
\EXPLAIN PLAN
SELECT
       o_orderdate,
       o_totalprice
  FROM orders
```

```
 WHERE o_orderkey IN (
                      SELECT   /*+ HASH_ISJ */
                               l_orderkey
                        FROM   lineitem
                      GROUP BY l_orderkey
                      HAVING   sum(l_quantity) > 300
                    )
;


>>>  start print plan


< Execution Plan >
=============================================================================
|  IDX  |  NODE DESCRIPTION                                                 |
-----------------------------------------------------------------------------
|    0  |  SELECT STATEMENT                                                 |
|    1  |    QUERY BLOCK ("$QB_IDX_2")                                      |
|    2  |      HASH JOIN (INNER JOIN)                                       |
|    3  |        INLINE_VIEW ("$V4")                                        |
|    4  |          QUERY BLOCK ("$QB_IDX_6")                                |
|    5  |            GROUP HASH INSTANT                                     |
|    6  |              TABLE ACCESS ("LINEITEM")                            |
|    7  |        HASH JOIN INSTANT                                          |
|    8  |          TABLE ACCESS ("ORDERS")                                  |
=============================================================================
```

```
1  -  TARGET : ORDERS.O_ORDERDATE, ORDERS.O_TOTALPRICE
2  -  JOINED COLUMN : ORDERS.O_ORDERDATE, ORDERS.O_TOTALPRICE
3  -  COLUMN : LINEITEM.L_ORDERKEY AS L_ORDERKEY
4  -  TARGET : LINEITEM.L_ORDERKEY
5  -  GROUP KEY : LINEITEM.L_ORDERKEY
      RECORD COLUMN : SUM( LINEITEM.L_QUANTITY )
      READ KEY COLUMN : LINEITEM.L_ORDERKEY
      READ RECORD COLUMN : SUM( LINEITEM.L_QUANTITY )
        PHYSICAL FILTER : SUM( LINEITEM.L_QUANTITY ) > 300
6  -  READ COLUMN : LINEITEM.L_ORDERKEY, LINEITEM.L_QUANTITY
7  -  HASH KEY : ORDERS.O_ORDERKEY
      RECORD COLUMN : ORDERS.O_ORDERDATE, ORDERS.O_TOTALPRICE
      READ KEY COLUMN : ORDERS.O_ORDERKEY, ORDERS.O_ORDERDATE,
ORDERS.O_TOTALPRICE
        HASH FILTER : ORDERS.O_ORDERKEY = $V4.L_ORDERKEY
      FETCH ONE ROW
8  -  READ COLUMN : ORDERS.O_ORDERKEY, ORDERS.O_TOTALPRICE,
ORDERS.O_ORDERDATE

<<<  end print plan
```

上述示例中应用HASH_ISJ hint后子查询unnest到inverted semi join并以hash join执行并且o_orderkey和l_orderkey均为primary key因此semi join变更为inner join

以下为在仅可以以anti-semi join形式unnest的子查询使用HASH_ISJ hint的示例

```
\EXPLAIN PLAN
SELECT
       o_orderdate,
       o_totalprice
  FROM orders
 WHERE o_orderkey NOT IN (
                           SELECT   /*+ HASH_ISJ */
                                    l_orderkey
                             FROM   lineitem
                            WHERE   l_shipdate >= date '1996-01-01'
                              AND   l_shipdate < date '1996-02-01'
                              AND   l_commitdate < l_receiptdate
                            GROUP BY l_orderkey
                            HAVING   sum(l_quantity) > 300
                          )
   AND  o_orderdate >= date '1996-01-01'
   AND  o_orderdate < date '1996-02-01'
   AND  o_comment like '%Customer%Complaints%'
   AND  o_orderstatus = 'F'
;



>>>  start print plan


< Execution Plan >
```

```
=============================================================================
|  IDX  |  NODE DESCRIPTION                                                 |
-----------------------------------------------------------------------------
|    0  |  SELECT STATEMENT                                                 |
|    1  |    QUERY BLOCK ("$QB_IDX_2")                                      |
|    2  |      NESTED JOIN (ANTI SEMI)                                      |
|    3  |        TABLE ACCESS ("ORDERS")                                    |
|    4  |        INLINE_VIEW ("$V5")                                        |
|    5  |          QUERY BLOCK ("$QB_IDX_6")                                |
|    6  |            GROUP                                                  |
|    7  |              INDEX ACCESS ("LINEITEM", "LINEITEM_PK_INDEX")       |
=============================================================================


     1  -  TARGET : ORDERS.O_ORDERDATE, ORDERS.O_TOTALPRICE
     2  -  JOINED COLUMN : ORDERS.O_ORDERDATE, ORDERS.O_TOTALPRICE
     3  -  READ COLUMN : ORDERS.O_ORDERKEY, ORDERS.O_ORDERSTATUS,
ORDERS.O_TOTALPRICE, ORDERS.O_ORDERDATE, ORDERS.O_COMMENT
             PHYSICAL FILTER : ORDERS.O_ORDERSTATUS = 'F' AND
ORDERS.O_ORDERDATE >= DATE'1996-01-01' AND ORDERS.O_ORDERDATE < DATE'1996-
02-01'
             LOGICAL FILTER : ORDERS.O_COMMENT LIKE
'%Customer%Complaints%'
     4  -  COLUMN : LINEITEM.L_ORDERKEY AS L_ORDERKEY
     5  -  TARGET : LINEITEM.L_ORDERKEY
     6  -  GROUP KEY : LINEITEM.L_ORDERKEY
```

```
            RECORD COLUMN : SUM( LINEITEM.L_QUANTITY )
              LOGICAL FILTER : SUM( LINEITEM.L_QUANTITY ) > 300
     7  -   READ INDEX COLUMN : LINEITEM.L_ORDERKEY
            READ TABLE COLUMN : LINEITEM.L_QUANTITY, LINEITEM.L_SHIPDATE,
LINEITEM.L_COMMITDATE, LINEITEM.L_RECEIPTDATE
              MIN RANGE : LINEITEM.L_ORDERKEY = {ORDERS.O_ORDERKEY}
              MAX RANGE : LINEITEM.L_ORDERKEY = {ORDERS.O_ORDERKEY}
              PHYSICAL TABLE FILTER : LINEITEM.L_SHIPDATE >= DATE'1996-01-
01' AND LINEITEM.L_SHIPDATE < DATE'1996-02-01' AND
LINEITEM.L_RECEIPTDATE > LINEITEM.L_COMMITDATE


<<<  end print plan
```

以anti-semi join进行unnest后以nested loop join方式执行因此hint已被忽略

NL_AJ

描述NL_AJ hint时optimizer以anti-semi join形式unnest子查询并以nested loop方式执行其semi join

与UNNEST_NL hint的运行方式相同但UNNEST_NL hint均适用于semi join和anti-semi join而NL_AJ hint仅适用于anti-semi join

因此在以semi join unnest的子查询描述上述hintoptimizer忽略其

以下为使用NL_AJ hint的示例

```
\EXPLAIN PLAN
```

```
SELECT
       o_orderdate,
       o_totalprice
  FROM orders
 WHERE o_orderkey NOT IN (
                           SELECT   /*+ NL_AJ */
                                    l_orderkey
                             FROM   lineitem
                            GROUP BY l_orderkey
                            HAVING   sum(l_quantity) > 300
                          )
;


>>>  start print plan


< Execution Plan >
============================================================================
|  IDX  |  NODE DESCRIPTION                                                |
----------------------------------------------------------------------------
|    0  |  SELECT STATEMENT                                                |
|    1  |    QUERY BLOCK ("$QB_IDX_2")                                     |
|    2  |      NESTED JOIN (ANTI SEMI)                                     |
|    3  |        TABLE ACCESS ("ORDERS")                                   |
|    4  |        INLINE_VIEW ("$V5")                                       |
|    5  |          QUERY BLOCK ("$QB_IDX_6")                               |
```

```
|    6  |            GROUP                                                    |
|    7  |              INDEX ACCESS ("LINEITEM", "LINEITEM_PK_INDEX")   |
=========================================================================


     1  -  TARGET : ORDERS.O_ORDERDATE, ORDERS.O_TOTALPRICE
     2  -  JOINED COLUMN : ORDERS.O_ORDERDATE, ORDERS.O_TOTALPRICE
     3  -  READ COLUMN : ORDERS.O_ORDERKEY, ORDERS.O_TOTALPRICE,
ORDERS.O_ORDERDATE
     4  -  COLUMN : LINEITEM.L_ORDERKEY AS L_ORDERKEY
     5  -  TARGET : LINEITEM.L_ORDERKEY
     6  -  GROUP KEY : LINEITEM.L_ORDERKEY
           RECORD COLUMN : SUM( LINEITEM.L_QUANTITY )
             LOGICAL FILTER : SUM( LINEITEM.L_QUANTITY ) > 300
     7  -  READ INDEX COLUMN : LINEITEM.L_ORDERKEY
           READ TABLE COLUMN : LINEITEM.L_QUANTITY
             MIN RANGE : LINEITEM.L_ORDERKEY = {ORDERS.O_ORDERKEY}
             MAX RANGE : LINEITEM.L_ORDERKEY = {ORDERS.O_ORDERKEY}


<<<  end print plan
```

以下为在仅可以以semi join形式unnest的子查询使用NL_AJ hint的示例

```
\EXPLAIN PLAN
SELECT
       o_orderdate,
       o_totalprice
```

```
  FROM orders
 WHERE o_comment IN (
                      SELECT   /*+ NL_AJ */
                               l_comment
                        FROM   lineitem
                       WHERE   l_shipdate >= date '1996-01-01'
                         AND   l_shipdate < date '1996-02-01'

                    )
;

>>>  start print plan

< Execution Plan >
================================================================================
|  IDX  |  NODE DESCRIPTION                                                    |
--------------------------------------------------------------------------------
|    0  |  SELECT STATEMENT                                                    |
|    1  |    QUERY BLOCK ("$QB_IDX_2")                                         |
|    2  |      HASH JOIN (SEMI)                                                |
|    3  |        TABLE ACCESS ("ORDERS")                                       |
|    4  |        HASH JOIN INSTANT (UNIQUE)                                    |
|    5  |          TABLE ACCESS ("LINEITEM")                                   |
================================================================================
```

```
     1  -  TARGET : ORDERS.O_ORDERDATE, ORDERS.O_TOTALPRICE
     2  -  JOINED COLUMN : ORDERS.O_ORDERDATE, ORDERS.O_TOTALPRICE
     3  -  READ COLUMN : ORDERS.O_TOTALPRICE, ORDERS.O_ORDERDATE,
ORDERS.O_COMMENT
     4  -  HASH KEY : LINEITEM.L_COMMENT
           READ KEY COLUMN : LINEITEM.L_COMMENT
             HASH FILTER : LINEITEM.L_COMMENT = ORDERS.O_COMMENT
           FETCH ONE ROW
     5  -  READ COLUMN : LINEITEM.L_SHIPDATE, LINEITEM.L_COMMENT
             PHYSICAL FILTER : LINEITEM.L_SHIPDATE >= DATE'1996-01-01' AND
LINEITEM.L_SHIPDATE < DATE'1996-02-01'

<<<  end print plan
```

以semi join进行unnest后以hash join方式执行因此hint已被忽略并且o_orderkey和l_orderkey均为primary key因此semi join变更为inner join

INL_AJ

描述INL_AJ hint时optimizer以anti-semi join形式unnest子查询并以instant nested loop方式执行其join

与UNNEST_INL hint的运行方式相同但UNNEST_INL hint均适用于semi join和anti-semi join而INL_AJ hint仅适用于anti-semi join

因此在以semi join unnest的子查询描述上述hintoptimizer忽略其

以下为使用INL_AJ hint的示例

```
\EXPLAIN PLAN
SELECT
       o_orderdate,
       o_totalprice
  FROM orders
 WHERE o_orderkey NOT IN (
                          SELECT   /*+ INL_AJ */
                                   l_orderkey
                            FROM   lineitem
                           GROUP BY l_orderkey
                           HAVING   sum(l_quantity) > 300
                         )
;

>>>  start print plan

< Execution Plan >
============================================================================
|  IDX  |  NODE DESCRIPTION                                                |
----------------------------------------------------------------------------
|    0  |  SELECT STATEMENT                                                |
|    1  |    QUERY BLOCK ("$QB_IDX_2")                                     |
|    2  |      NESTED JOIN (ANTI SEMI)                                     |
|    3  |        TABLE ACCESS ("ORDERS")                                   |
|    4  |        SORT JOIN INSTANT                                         |
```

```
|    5  |          INLINE_VIEW ("$V6")                                        |
|    6  |            QUERY BLOCK ("$QB_IDX_6")                                |
|    7  |              GROUP HASH INSTANT                                      |
|    8  |                TABLE ACCESS ("LINEITEM")                             |
========================================================================


     1  -  TARGET : ORDERS.O_ORDERDATE, ORDERS.O_TOTALPRICE
     2  -  JOINED COLUMN : ORDERS.O_ORDERDATE, ORDERS.O_TOTALPRICE
     3  -  READ COLUMN : ORDERS.O_ORDERKEY, ORDERS.O_TOTALPRICE,
ORDERS.O_ORDERDATE
     4  -  SORT KEY : "$V6.L_ORDERKEY ASC NULLS LAST"
           READ KEY COLUMN : $V6.L_ORDERKEY
             MIN RANGE : $V6.L_ORDERKEY = {ORDERS.O_ORDERKEY}
             MAX RANGE : $V6.L_ORDERKEY = {ORDERS.O_ORDERKEY}
     5  -  COLUMN : LINEITEM.L_ORDERKEY AS L_ORDERKEY
     6  -  TARGET : LINEITEM.L_ORDERKEY
     7  -  GROUP KEY : LINEITEM.L_ORDERKEY
           RECORD COLUMN : SUM( LINEITEM.L_QUANTITY )
           READ KEY COLUMN : LINEITEM.L_ORDERKEY
           READ RECORD COLUMN : SUM( LINEITEM.L_QUANTITY )
             PHYSICAL FILTER : SUM( LINEITEM.L_QUANTITY ) > 300
     8  -  READ COLUMN : LINEITEM.L_ORDERKEY, LINEITEM.L_QUANTITY


<<<  end print plan
```

以下为在仅可以以semi join形式unnest的子查询使用INL_AJ hint的示例

```
\EXPLAIN PLAN
SELECT
       o_orderdate,
       o_totalprice
  FROM orders
 WHERE o_orderkey IN (
                       SELECT   /*+ INL_AJ */
                                l_orderkey
                         FROM   lineitem
                       GROUP BY l_orderkey
                       HAVING   sum(l_quantity) > 300
                     )
;

>>>  start print plan

< Execution Plan >
=========================================================================
|  IDX  |  NODE DESCRIPTION                                             |
-------------------------------------------------------------------------
|    0  |  SELECT STATEMENT                                             |
|    1  |    QUERY BLOCK ("$QB_IDX_2")                                  |
|    2  |      NESTED JOIN (INNER JOIN)                                 |
```

```
|    3  |        INLINE_VIEW ("$V4")                                    |
|    4  |          QUERY BLOCK ("$QB_IDX_6")                            |
|    5  |            GROUP HASH INSTANT                                 |
|    6  |              TABLE ACCESS ("LINEITEM")                        |
|    7  |        INDEX ACCESS ("ORDERS", "ORDERS_PK_INDEX")             |
=========================================================================


     1  -  TARGET : ORDERS.O_ORDERDATE, ORDERS.O_TOTALPRICE
     2  -  JOINED COLUMN : ORDERS.O_ORDERDATE, ORDERS.O_TOTALPRICE
     3  -  COLUMN : LINEITEM.L_ORDERKEY AS L_ORDERKEY
     4  -  TARGET : LINEITEM.L_ORDERKEY
     5  -  GROUP KEY : LINEITEM.L_ORDERKEY
           RECORD COLUMN : SUM( LINEITEM.L_QUANTITY )
           READ KEY COLUMN : LINEITEM.L_ORDERKEY
           READ RECORD COLUMN : SUM( LINEITEM.L_QUANTITY )
             PHYSICAL FILTER : SUM( LINEITEM.L_QUANTITY ) > 300
     6  -  READ COLUMN : LINEITEM.L_ORDERKEY, LINEITEM.L_QUANTITY
     7  -  READ INDEX COLUMN : ORDERS.O_ORDERKEY
           READ TABLE COLUMN : ORDERS.O_TOTALPRICE, ORDERS.O_ORDERDATE
             MIN RANGE : ORDERS.O_ORDERKEY = {$V4.L_ORDERKEY}
             MAX RANGE : ORDERS.O_ORDERKEY = {$V4.L_ORDERKEY}
           FETCH ONE ROW


<<<  end print plan
```

以semi join进行unnest后以nested loop join方式执行因此hint已被忽略并且o_orderkey和l_orderkey均为primary key因此semi join变更为inner join

MERGE_AJ

描述MERGE_AJ hint时optimizer以anti-semi join形式unnest子查询并以merge join方式执行其join

与UNNEST_MERGE hint的运行方式相同但UNNEST_MERGE hint均适用于semi joinanti-semi join而MERGE_AJ hint仅适用于anti-semi join

因此在以semi join unnest的子查询描述上述hintoptimizer忽略其

以下为使用MERGE_AJ hint的示例

```
\EXPLAIN PLAN
SELECT
       o_orderdate,
       o_totalprice
  FROM orders
 WHERE o_orderkey NOT IN (
                           SELECT   /*+ MERGE_AJ */
                                    l_orderkey
                             FROM   lineitem
                            GROUP BY l_orderkey
                            HAVING   sum(l_quantity) > 300
                          )
;
```

```
>>>  start print plan


< Execution Plan >

=============================================================================
|  IDX  |  NODE DESCRIPTION                                                 |
-----------------------------------------------------------------------------
|    0  |  SELECT STATEMENT                                                 |
|    1  |    QUERY BLOCK ("$QB_IDX_2")                                      |
|    2  |      MERGE JOIN (ANTI SEMI)                                       |
|    3  |        INDEX ACCESS ("ORDERS", "ORDERS_PK_INDEX")                 |
|    4  |        SORT JOIN INSTANT (UNIQUE)                                 |
|    5  |          INLINE_VIEW ("$V6")                                      |
|    6  |            QUERY BLOCK ("$QB_IDX_6")                              |
|    7  |              GROUP HASH INSTANT                                   |
|    8  |                TABLE ACCESS ("LINEITEM")                          |
=============================================================================


     1  -  TARGET : ORDERS.O_ORDERDATE, ORDERS.O_TOTALPRICE
     2  -  JOINED COLUMN : ORDERS.O_ORDERDATE, ORDERS.O_TOTALPRICE
             ON FILTER (Equi) : ORDERS.O_ORDERKEY = $V6.L_ORDERKEY
     3  -  READ INDEX COLUMN : ORDERS.O_ORDERKEY
           READ TABLE COLUMN : ORDERS.O_TOTALPRICE, ORDERS.O_ORDERDATE
     4  -  SORT KEY : "$V6.L_ORDERKEY ASC NULLS LAST"
           READ KEY COLUMN : $V6.L_ORDERKEY
```

```
            MIN RANGE : $V6.L_ORDERKEY >= {ORDERS.O_ORDERKEY}
            MAX RANGE : $V6.L_ORDERKEY IS NOT NULL
     5  -  COLUMN : LINEITEM.L_ORDERKEY AS L_ORDERKEY
     6  -  TARGET : LINEITEM.L_ORDERKEY
     7  -  GROUP KEY : LINEITEM.L_ORDERKEY
           RECORD COLUMN : SUM( LINEITEM.L_QUANTITY )
           READ KEY COLUMN : LINEITEM.L_ORDERKEY
           READ RECORD COLUMN : SUM( LINEITEM.L_QUANTITY )
            PHYSICAL FILTER : SUM( LINEITEM.L_QUANTITY ) > 300
     8  -  READ COLUMN : LINEITEM.L_ORDERKEY, LINEITEM.L_QUANTITY


<<<  end print plan
```

以下为在仅可以以semi join形式unnest的子查询使用MERGE_AJ hint的示例

```
\EXPLAIN PLAN
SELECT
       o_orderdate,
       o_totalprice
  FROM orders
 WHERE o_orderkey IN (
                      SELECT   /*+ MERGE_AJ */
                               l_orderkey
                        FROM   lineitem
                      GROUP BY l_orderkey
                      HAVING   sum(l_quantity) > 300
```

```
                    )
;


>>>  start print plan


< Execution Plan >
=============================================================================
|  IDX  |  NODE DESCRIPTION                                                  |
-----------------------------------------------------------------------------
|    0  |  SELECT STATEMENT                                                  |
|    1  |    QUERY BLOCK ("$QB_IDX_2")                                       |
|    2  |      NESTED JOIN (INNER JOIN)                                      |
|    3  |        INLINE_VIEW ("$V4")                                         |
|    4  |          QUERY BLOCK ("$QB_IDX_6")                                 |
|    5  |            GROUP HASH INSTANT                                      |
|    6  |              TABLE ACCESS ("LINEITEM")                             |
|    7  |        INDEX ACCESS ("ORDERS", "ORDERS_PK_INDEX")                  |
=============================================================================


     1  -  TARGET : ORDERS.O_ORDERDATE, ORDERS.O_TOTALPRICE
     2  -  JOINED COLUMN : ORDERS.O_ORDERDATE, ORDERS.O_TOTALPRICE
     3  -  COLUMN : LINEITEM.L_ORDERKEY AS L_ORDERKEY
     4  -  TARGET : LINEITEM.L_ORDERKEY
     5  -  GROUP KEY : LINEITEM.L_ORDERKEY
           RECORD COLUMN : SUM( LINEITEM.L_QUANTITY )
```

```
            READ KEY COLUMN : LINEITEM.L_ORDERKEY
            READ RECORD COLUMN : SUM( LINEITEM.L_QUANTITY )
              PHYSICAL FILTER : SUM( LINEITEM.L_QUANTITY ) > 300
      6   -  READ COLUMN : LINEITEM.L_ORDERKEY, LINEITEM.L_QUANTITY
      7   -  READ INDEX COLUMN : ORDERS.O_ORDERKEY
            READ TABLE COLUMN : ORDERS.O_TOTALPRICE, ORDERS.O_ORDERDATE
              MIN RANGE : ORDERS.O_ORDERKEY = {$V4.L_ORDERKEY}
              MAX RANGE : ORDERS.O_ORDERKEY = {$V4.L_ORDERKEY}
            FETCH ONE ROW


<<<  end print plan
```

以semi join进行unnest后以nested loop join方式执行因此hint已被忽略并且o_orderkey和l_orderkey均为primary key因此semi join变更为inner join

HASH_AJ

描述HASH_AJ hint时optimizer以anti-semi join形式unnest子查询并以hash join方式执行其join

与UNNEST_HASH hint的运行方式相同但UNNEST_HASH hint均适用于semi join和anti-semi join而HASH_AJ hint仅适用于anti-semi join

因此在以semi join unnest的子查询描述上述hintoptimizer忽略其

以下为使用HASH_AJ hint的示例

```
\EXPLAIN PLAN
SELECT
```

```
       o_orderdate,
       o_totalprice
  FROM orders
 WHERE o_orderkey NOT IN (
                          SELECT   /*+ HASH_AJ */
                                   l_orderkey
                            FROM   lineitem
                           GROUP BY l_orderkey
                           HAVING   sum(l_quantity) > 300
                         )
;


>>>  start print plan


< Execution Plan >
=============================================================================
|  IDX  |  NODE DESCRIPTION                                                 |
-----------------------------------------------------------------------------
|    0  |  SELECT STATEMENT                                                 |
|    1  |    QUERY BLOCK ("$QB_IDX_2")                                      |
|    2  |      HASH JOIN (ANTI SEMI)                                        |
|    3  |        TABLE ACCESS ("ORDERS")                                    |
|    4  |        HASH JOIN INSTANT (UNIQUE)                                 |
|    5  |          INLINE_VIEW ("$V6")                                      |
|    6  |            QUERY BLOCK ("$QB_IDX_6")                              |
```

```
|    7  |              GROUP HASH INSTANT                                    |
|    8  |                TABLE ACCESS ("LINEITEM")                           |
=========================================================================


     1  -  TARGET : ORDERS.O_ORDERDATE, ORDERS.O_TOTALPRICE
     2  -  JOINED COLUMN : ORDERS.O_ORDERDATE, ORDERS.O_TOTALPRICE
     3  -  READ COLUMN : ORDERS.O_ORDERKEY, ORDERS.O_TOTALPRICE,
ORDERS.O_ORDERDATE
     4  -  HASH KEY : $V6.L_ORDERKEY
           READ KEY COLUMN : $V6.L_ORDERKEY
             HASH FILTER : $V6.L_ORDERKEY = ORDERS.O_ORDERKEY
           FETCH ONE ROW
     5  -  COLUMN : LINEITEM.L_ORDERKEY AS L_ORDERKEY
     6  -  TARGET : LINEITEM.L_ORDERKEY
     7  -  GROUP KEY : LINEITEM.L_ORDERKEY
           RECORD COLUMN : SUM( LINEITEM.L_QUANTITY )
           READ KEY COLUMN : LINEITEM.L_ORDERKEY
           READ RECORD COLUMN : SUM( LINEITEM.L_QUANTITY )
             PHYSICAL FILTER : SUM( LINEITEM.L_QUANTITY ) > 300
     8  -  READ COLUMN : LINEITEM.L_ORDERKEY, LINEITEM.L_QUANTITY


<<<  end print plan
```

以下为在仅可以以semi join形式unnest的子查询使用HASH_AJ hint的示例

```
\EXPLAIN PLAN
```

```
SELECT
       o_orderdate,
       o_totalprice
  FROM orders
 WHERE o_orderkey IN (
                       SELECT   /*+ HASH_AJ */
                                l_orderkey
                         FROM   lineitem
                       GROUP BY l_orderkey
                       HAVING   sum(l_quantity) > 300
                     )
;


>>>  start print plan


< Execution Plan >
=============================================================================
|  IDX  |  NODE DESCRIPTION                                                 |
-----------------------------------------------------------------------------
|    0  |  SELECT STATEMENT                                                 |
|    1  |    QUERY BLOCK ("$QB_IDX_2")                                      |
|    2  |      NESTED JOIN (INNER JOIN)                                     |
|    3  |        INLINE_VIEW ("$V4")                                        |
|    4  |          QUERY BLOCK ("$QB_IDX_6")                                |
|    5  |            GROUP HASH INSTANT                                     |
```

```
|    6  |              TABLE ACCESS ("LINEITEM")                          |
|    7  |         INDEX ACCESS ("ORDERS", "ORDERS_PK_INDEX")              |
=====================================================================


     1  -  TARGET : ORDERS.O_ORDERDATE, ORDERS.O_TOTALPRICE
     2  -  JOINED COLUMN : ORDERS.O_ORDERDATE, ORDERS.O_TOTALPRICE
     3  -  COLUMN : LINEITEM.L_ORDERKEY AS L_ORDERKEY
     4  -  TARGET : LINEITEM.L_ORDERKEY
     5  -  GROUP KEY : LINEITEM.L_ORDERKEY
           RECORD COLUMN : SUM( LINEITEM.L_QUANTITY )
           READ KEY COLUMN : LINEITEM.L_ORDERKEY
           READ RECORD COLUMN : SUM( LINEITEM.L_QUANTITY )
             PHYSICAL FILTER : SUM( LINEITEM.L_QUANTITY ) > 300
     6  -  READ COLUMN : LINEITEM.L_ORDERKEY, LINEITEM.L_QUANTITY
     7  -  READ INDEX COLUMN : ORDERS.O_ORDERKEY
           READ TABLE COLUMN : ORDERS.O_TOTALPRICE, ORDERS.O_ORDERDATE
             MIN RANGE : ORDERS.O_ORDERKEY = {$V4.L_ORDERKEY}
             MAX RANGE : ORDERS.O_ORDERKEY = {$V4.L_ORDERKEY}
           FETCH ONE ROW


<<<  end print plan
```

以semi join进行unnest后以nested loop join方式执行因此hint已被忽略并且o_orderkey和l_orderkey均为primary key因此semi join变更为inner join

<unnest join driver hints>

在集群系统执行subquery unnesting时可使用的hint在单机版被忽略

LOCAL_UNNEST

描述LOCAL_UNNEST hint时optimizer将子查询变更为保证相同结果的join语句后在local执行其join即将left child和right child的结果均获取到local后执行join

以下为使用LOCAL_UNNEST hint的示例

```
\EXPLAIN PLAN
SELECT
       o_orderdate,
       o_totalprice
  FROM orders
 WHERE o_orderkey IN (
                      SELECT   /*+ LOCAL_UNNEST */
                               l_orderkey
                        FROM   lineitem
                      GROUP BY l_orderkey
                      HAVING   sum(l_quantity) > 300
                    )
;

>>>  start print plan
```

```
< Execution Plan >

================================================================================

|  IDX  |  NODE DESCRIPTION                                                      |

--------------------------------------------------------------------------------

|    0  |  SELECT STATEMENT                                                      |

|    1  |    QUERY BLOCK ("$QB_IDX_2")                                           |

|    2  |      HASH JOIN (INNER JOIN)                                            |

|    3  |        PLAN BASED CLUSTER                                              |

|    4  |          TABLE ACCESS ("ORDERS")                                       |

|    5  |        HASH JOIN INSTANT                                               |

|    6  |          PLAN BASED CLUSTER                                            |

|    7  |            INLINE_VIEW ("$V8")                                         |

|    8  |              QUERY BLOCK ("$QB_IDX_6")                                 |

|    9  |                GROUP HASH INSTANT                                      |

|   10  |                  TABLE ACCESS ("LINEITEM")                             |

================================================================================


     1  -  TARGET : ORDERS.O_ORDERDATE, ORDERS.O_TOTALPRICE

     2  -  JOINED COLUMN : ORDERS.O_ORDERDATE, ORDERS.O_TOTALPRICE

     3  -  SQL : SELECT /*+ FULL( _A1 ) */ "_A1"."O_ORDERKEY",
"_A1"."O_TOTALPRICE", "_A1"."O_ORDERDATE" FROM "PUBLIC"."ORDERS"@LOCAL AS
"_A1"

           TARGET DOMAIN : G1(G1N1,G1N2) 0 rows, G2(G2N1,G2N2) 0 rows,
G3(G3N1,G3N2) 0 rows

     4  -  HASH SHARD ( # 3 )
```

```
           READ COLUMN : ORDERS.O_ORDERKEY, ORDERS.O_TOTALPRICE,
ORDERS.O_ORDERDATE
     5  -  HASH KEY : $V8.L_ORDERKEY
           READ KEY COLUMN : $V8.L_ORDERKEY
             HASH FILTER : $V8.L_ORDERKEY = ORDERS.O_ORDERKEY
     6  -  SQL : SELECT /*+ NO_MERGE( _A1 ) */ * FROM ( SELECT /*+
USE_GROUP_HASH(10) FULL( _A2 ) */ "_A2"."L_ORDERKEY" FROM
"PUBLIC"."LINEITEM"@LOCAL AS "_A2" GROUP BY "_A2"."L_ORDERKEY" HAVING
SUM( "_A2"."L_QUANTITY" ) > :_V0 ) AS "_A1"("L_ORDERKEY")
           TARGET DOMAIN : G1(G1N1,G1N2) 0 rows, G2(G2N1,G2N2) 0 rows,
G3(G3N1,G3N2) 0 rows
     7  -  COLUMN : LINEITEM.L_ORDERKEY AS L_ORDERKEY
     8  -  TARGET : LINEITEM.L_ORDERKEY
     9  -  GROUP KEY : LINEITEM.L_ORDERKEY
           RECORD COLUMN : SUM( LINEITEM.L_QUANTITY )
           READ KEY COLUMN : LINEITEM.L_ORDERKEY
           READ RECORD COLUMN : SUM( LINEITEM.L_QUANTITY )
             PHYSICAL FILTER : SUM( LINEITEM.L_QUANTITY ) > 300
    10  -  HASH SHARD ( # 3 )
           READ COLUMN : LINEITEM.L_ORDERKEY, LINEITEM.L_QUANTITY


<<<  end print plan
```

如上述execution plan所示IDX 3的PLAN BASED CLUSTER获取了TABLE ACCESS ("ORDERS")结果IDX 6的PLAN BASED CLUSTER获取了 INLINE_VIEW ("$V8")结果并在local执行了HASH JOIN

(INNER JOIN)

REMOTE_UNNEST

描述REMOTE_UNNEST hint时optimizer将子查询转换为保证相同结果的join语句并在remote执行其join即在各个group执行join

以下为使用REMOTE_UNNEST hint的示例

```
\EXPLAIN PLAN
SELECT
       o_orderdate,
       o_totalprice
  FROM orders
 WHERE o_orderkey IN (
                      SELECT   /*+ REMOTE_UNNEST */
                               l_orderkey
                        FROM   lineitem
                      GROUP BY l_orderkey
                      HAVING   sum(l_quantity) > 300
                    )
;



>>>  start print plan


< Execution Plan >
```

```
=========================================================================
|  IDX  |  NODE DESCRIPTION                                             |
-------------------------------------------------------------------------
|    0  |  SELECT STATEMENT                                             |
|    1  |    QUERY BLOCK ("$QB_IDX_2")                                  |
|    2  |      PLAN BASED CLUSTER                                       |
|    3  |        NESTED JOIN (INNER JOIN)                               |
|    4  |          INLINE_VIEW ("$V5")                                  |
|    5  |            QUERY BLOCK ("$QB_IDX_6")                          |
|    6  |              GROUP HASH INSTANT                               |
|    7  |                TABLE ACCESS ("LINEITEM")                      |
|    8  |          INDEX ACCESS ("ORDERS", "ORDERS_PK_INDEX")           |
=========================================================================


     1  -  TARGET : ORDERS.O_ORDERDATE, ORDERS.O_TOTALPRICE
     2  -  SQL : SELECT /*+ KEEP_JOINED_TABLE USE_NL_IN( _A1 )
NO_MERGE( _A2 ) INDEX( _A1, "PUBLIC"."ORDERS_PK_INDEX" ) */
"_A1"."O_ORDERDATE", "_A1"."O_TOTALPRICE" FROM ( ( SELECT /*+
USE_GROUP_HASH(10) FULL( _A3 ) */ "_A3"."L_ORDERKEY" FROM
"PUBLIC"."LINEITEM"@LOCAL AS "_A3" GROUP BY "_A3"."L_ORDERKEY" HAVING
SUM( "_A3"."L_QUANTITY" ) > :_V0 ) AS "_A2"("L_ORDERKEY") INNER JOIN
"PUBLIC"."ORDERS"@LOCAL AS "_A1" ON true ) ALIAS "_A4" WHERE
"_A1"."O_ORDERKEY" = "_A2"."L_ORDERKEY"
           TARGET DOMAIN : G1(G1N1,G1N2) 0 rows, G2(G2N1,G2N2) 0 rows,
G3(G3N1,G3N2) 0 rows
```

```
    3  -  JOINED COLUMN : ORDERS.O_ORDERDATE, ORDERS.O_TOTALPRICE
    4  -  COLUMN : LINEITEM.L_ORDERKEY AS L_ORDERKEY
    5  -  TARGET : LINEITEM.L_ORDERKEY
    6  -  GROUP KEY : LINEITEM.L_ORDERKEY
          RECORD COLUMN : SUM( LINEITEM.L_QUANTITY )
          READ KEY COLUMN : LINEITEM.L_ORDERKEY
          READ RECORD COLUMN : SUM( LINEITEM.L_QUANTITY )
            PHYSICAL FILTER : SUM( LINEITEM.L_QUANTITY ) > 300
    7  -  HASH SHARD ( # 3 )
          READ COLUMN : LINEITEM.L_ORDERKEY, LINEITEM.L_QUANTITY
    8  -  HASH SHARD ( # 3 )
          READ INDEX COLUMN : ORDERS.O_ORDERKEY
          READ TABLE COLUMN : ORDERS.O_TOTALPRICE, ORDERS.O_ORDERDATE
            MIN RANGE : ORDERS.O_ORDERKEY = {$V5.L_ORDERKEY}
            MAX RANGE : ORDERS.O_ORDERKEY = {$V5.L_ORDERKEY}
          FETCH ONE ROW

<<<  end print plan
```

如上述executi plan所示在各个group执行join后在IDX 2的PLAN BASED CLUSTER归集join结果并返回

<unnest join pusher hints>

在集群系统执行subquery unnesting时可使用的hint在单机版被忽略

PUSHER_SUBQ

描述PUSHER_SUBQ时Optimizer将子查询变更为保证相同结果的join语句后在remote执行其join时将unnest子查询后生成的view或表部署到pusher table

以下为使用PUSHER_SUBQ hint的示例

```
\EXPLAIN PLAN
SELECT p_name
  FROM part
 WHERE p_partkey IN ( SELECT /*+ PUSHER_SUBQ */
                             l_partkey
                        FROM lineitem
                       WHERE l_shipdate = date'1998-12-01'
                     );


< Execution Plan >
================================================================================
=
|IDX |  NODE DESCRIPTION                                                 | ROWS
|
--------------------------------------------------------------------------------
|  0 |  SELECT STATEMENT                                                 |  18 |
|  1 |    QUERY BLOCK ("$QB_IDX_2")                                      |  18 |
|  2 |      SINGLE CLUSTER                                 | LOCAL/REMOTE 18 |
|  3 |        CLUSTER PUSHER ("_$NI_6")                                  |  18 |
```

```
|  4 |          PLAN BASED CLUSTER                      | LOCAL/REMOTE 18 |
|  5 |            TABLE ACCESS ("LINEITEM")                          |   7 |
|  6 |        SELECT STATEMENT                                       |   5 |
|  7 |          QUERY BLOCK ("$QB_IDX_2")                            |   5 |
|  8 |            NESTED JOIN (INVERTED SEMI)                        |   5 |
|  9 |              SORT JOIN INSTANT (UNIQUE)                       |   5 |
| 10 |                PUSHER TABLE ACCESS ("_$NI_6" AS _A2)          |   5 |
| 11 |              INDEX ACCESS ("PART" AS _A1, "PART_PK_INDEX")    |   5 |
=============================================================================


    1  -  TARGET : PART.P_NAME
    2  -  SQL :
          SELECT /*+ KEEP_JOINED_TABLE USE_NL_IN( _A1 )
                     FULL( _A2 )
                     INDEX( _A1, "PUBLIC"."PART_PK_INDEX" )
                */
                "_A1"."P_NAME"
          FROM
( "PUBLIC"."PART"@"G1N1"|"G1N2"|"G2N1"|"G2N2"|"G3N1"|"G3N2"
                 AS "_A1"
                SEMI JOIN
                "SESSION_SCHEMA"."_$NI_6"@LOCAL AS "_A2"
                ON "_A1"."P_PARTKEY" = "_A2"."L_PARTKEY"
              ) ALIAS "_A3"
          TARGET DOMAIN : G1(G1N1,G1N2) 5 rows,
```

```
                    G2(G2N1,G2N2) 9 rows,
                    G3(G3N1,G3N2) 4 rows
 3  -  SQL : DECLARE INSTANT TABLE "SESSION_SCHEMA"."_$NI_6"
          ( "L_PARTKEY" NUMBER(10, 0) )
    COLUMN : LINEITEM.L_PARTKEY AS L_PARTKEY
    SHARDED : LINEITEM.L_PARTKEY
    TARGET DOMAIN : G1(G1N1,G1N2) 5 rows,
                    G2(G2N1,G2N2) 9 rows,
                    G3(G3N1,G3N2) 4 rows
 4  -  SQL :
    SELECT /*+ FULL( _A1 ) */
           "_A1"."L_PARTKEY"
      FROM "PUBLIC"."LINEITEM"@LOCAL AS "_A1"
     WHERE "_A1"."L_SHIPDATE" = :_V0
    TARGET DOMAIN : G1(G1N1,G1N2) 7 rows,
                    G2(G2N1,G2N2) 4 rows,
                    G3(G3N1,G3N2) 7 rows
 5  -  HASH SHARD ( # 3 )
    READ COLUMN : LINEITEM.L_PARTKEY, LINEITEM.L_SHIPDATE
      PHYSICAL FILTER : LINEITEM.L_SHIPDATE = DATE'1998-12-01'
 7  -  TARGET : _A1.P_NAME
 8  -  JOINED COLUMN : _A1.P_NAME
 9  -  SORT KEY : "_A2.L_PARTKEY ASC NULLS LAST"
    READ KEY COLUMN : _A2.L_PARTKEY
10  -  READ COLUMN : _A2.L_PARTKEY
```

```
    11  -  HASH SHARD ( # 3 )
           READ INDEX COLUMN : _A1.P_PARTKEY
           READ TABLE COLUMN : _A1.P_NAME
             MIN RANGE : _A1.P_PARTKEY = {_A2.L_PARTKEY}
             MAX RANGE : _A1.P_PARTKEY = {_A2.L_PARTKEY}
           FETCH ONE ROW


<<<  end print plan
```

如上述execution plan中的[IDX 3]创建 l_partkey为shard key的pusher table并存储从[IDX 4]读取的数据之后part和pusher table执行remote semi join

NO_PUSHER_SUBQ

描述NO_PUSHER_SUBQ hint时Optimizer将子查询转换为保证相同结果的join语句后在remote执行其join时使unnest子查询后生成的view或table无法部署到pusher table

因此可能无法remote unnest也可能将sibling table部署到pusher table

以下为使用NO_PUSHER_SUBQ hint的示例

```
\EXPLAIN PLAN
SELECT p_name
  FROM part
 WHERE p_partkey IN ( SELECT /*+ NO_PUSHER_SUBQ */
                             l_partkey
                        FROM lineitem
```

```
                      WHERE l_shipdate = date'1998-12-01'
                    );



>>>  start print plan


< Execution Plan >
================================================================================
==
|IDX|  NODE DESCRIPTION                              |                ROWS
|
--------------------------------------------------------------------------------
--
| 0 |  SELECT STATEMENT                              |                  18
|
| 1 |    QUERY BLOCK ("$QB_IDX_2")                   |                  18
|
| 2 |      HASH JOIN (SEMI)                          |                  18
|
| 3 |        PLAN BASED CLUSTER                      | LOCAL/REMOTE 200000
|
| 4 |          TABLE ACCESS ("PART")                 |               66675
|
| 5 |        HASH JOIN INSTANT (UNIQUE)              |                  18
|
```

```
| 6 |          PLAN BASED CLUSTER                    |     LOCAL/REMOTE 18
|
| 7 |            TABLE ACCESS ("LINEITEM")           |                   7
|
================================================================================
==


     1  -  TARGET : PART.P_NAME
     2  -  JOINED COLUMN : PART.P_NAME
     3  -  SQL : SELECT /*+ FULL( _A1 ) */
                         "_A1"."P_PARTKEY", "_A1"."P_NAME"
                    FROM "PUBLIC"."PART"@LOCAL AS "_A1"
           TARGET DOMAIN : G1(G1N1,G1N2) 66675 rows,
                           G2(G2N1,G2N2) 66664 rows,
                           G3(G3N1,G3N2) 66661 rows
     4  -  HASH SHARD ( # 3 )
           READ COLUMN : PART.P_PARTKEY, PART.P_NAME
     5  -  HASH KEY : LINEITEM.L_PARTKEY
           READ KEY COLUMN : LINEITEM.L_PARTKEY
             HASH FILTER : LINEITEM.L_PARTKEY = PART.P_PARTKEY
           FETCH ONE ROW
     6  -  SQL : SELECT /*+ FULL( _A1 ) */
                         "_A1"."L_PARTKEY"
                     FROM "PUBLIC"."LINEITEM"@LOCAL AS "_A1"
                    WHERE "_A1"."L_SHIPDATE" = :_V0
```

```
           TARGET DOMAIN : G1(G1N1,G1N2) 7 rows,
                           G2(G2N1,G2N2) 4 rows,
                           G3(G3N1,G3N2) 7 rows
     7   -  HASH SHARD ( # 3 )
           READ COLUMN : LINEITEM.L_PARTKEY, LINEITEM.L_SHIPDATE
             PHYSICAL FILTER : LINEITEM.L_SHIPDATE = DATE'1998-12-01'


<<<  end print plan
```

如上述execution plan所示无法remote unnest并以local unnest执行即在part和lineitem从G1G2G3读取满足条件的数据并在当前driver server中以semi join执行

以下为使用REMOTE_UNNEST hint和NO_PUSHER_SUBQ hint的示例

```
\EXPLAIN PLAN
SELECT p_name
  FROM part
 WHERE p_partkey IN ( SELECT /*+ REMOTE_UNNEST NO_PUSHER_SUBQ */
                             l_partkey
                        FROM lineitem
                       WHERE l_shipdate = date'1998-12-01'
                     );


>>>  start print plan


< Execution Plan >
```

```
================================================================================
===
| IDX|  NODE DESCRIPTION                                          |   ROWS
|
--------------------------------------------------------------------------------
--
|  0 |  SELECT STATEMENT                                          |     18
|
|  1 |    QUERY BLOCK ("$QB_IDX_2")                               |     18
|
|  2 |      MULTIPLE CLUSTER                          |     LOCAL/REMOTE 18
|
|  3 |        CLUSTER PUSHER ("_$NI_5")                           | 200000
|
|  4 |          PLAN BASED CLUSTER                    | LOCAL/REMOTE 200000
|
|  5 |            TABLE ACCESS ("PART")                           |  66675
|
|  6 |        SELECT STATEMENT                                    |      7
|
|  7 |          QUERY BLOCK ("$QB_IDX_2")                         |      7
|
|  8 |            SORT INSTANT                                    |      7
|
|  9 |              HASH JOIN (SEMI)                              |      7
```

```
|
| 10 |                PUSHER TABLE ACCESS ("_$NI_5" AS _A2)       | 200000
|
| 11 |                HASH JOIN INSTANT (UNIQUE)                 |      7
|
| 12 |                  TABLE ACCESS ("LINEITEM" AS _A1)         |      7
|
================================================================================
==


     1  -  TARGET : _$NI_5.P_NAME
     2  -  SQL :
           SELECT /*+ KEEP_JOINED_TABLE
                      USE_HASH_IN( _A1, 396 )
                      FULL( _A2 )
                      FULL( _A1 )
                 */
                 "_A2"."P_PARTKEY", "_A2"."P_NAME"
            FROM ( "SESSION_SCHEMA"."_$NI_5"@LOCAL AS "_A2"
                   SEMI JOIN
                   "PUBLIC"."LINEITEM"@"G1N1"|"G1N2"|
                                       "G2N1"|"G2N2"|
                                       "G3N1"|"G3N2"
                   AS "_A1"
                   ON  "_A1"."L_PARTKEY" = "_A2"."P_PARTKEY"
```

```
              AND "_A1"."L_SHIPDATE" = :_V0
            ) ALIAS "_A3"
      ORDER BY "_A2"."P_PARTKEY" ASC NULLS LAST
      TARGET DOMAIN : G1(G1N1,G1N2) 7 rows,
                      G2(G2N1,G2N2) 4 rows,
                      G3(G3N1,G3N2) 7 rows
      DISTINCT KEY GROUP
        KEY GROUP : _$NI_5.P_PARTKEY
3  -  SQL : DECLARE INSTANT TABLE "SESSION_SCHEMA"."_$NI_5"
           ( "P_PARTKEY" NUMBER(10, 0), "P_NAME" VARCHAR(55 OCTETS) )
      COLUMN : PART.P_PARTKEY AS P_PARTKEY, PART.P_NAME AS P_NAME
      CLONED
      TARGET DOMAIN : G1(G1N1,G1N2) 200000 rows,
                      G2(G2N1,G2N2) 200000 rows,
                      G3(G3N1,G3N2) 200000 rows
4  -  SQL :
      SELECT /*+ FULL( _A1 ) */
             "_A1"."P_PARTKEY", "_A1"."P_NAME"
        FROM "PUBLIC"."PART"@LOCAL AS "_A1"
      TARGET DOMAIN : G1(G1N1,G1N2) 66675 rows,
                      G2(G2N1,G2N2) 66664 rows,
                      G3(G3N1,G3N2) 66661 rows
5  -  HASH SHARD ( # 3 )
      READ COLUMN : PART.P_PARTKEY, PART.P_NAME
7  -  TARGET : _A2.P_PARTKEY, _A2.P_NAME
```

```
     8  -  SORT KEY : "_A2.P_PARTKEY ASC NULLS LAST"
           RECORD COLUMN : _A2.P_NAME
           READ KEY COLUMN : _A2.P_PARTKEY
           READ RECORD COLUMN : _A2.P_NAME
     9  -  JOINED COLUMN : _A2.P_PARTKEY, _A2.P_NAME
    10  -  READ COLUMN : _A2.P_PARTKEY, _A2.P_NAME
    11  -  HASH KEY : _A1.L_PARTKEY
           READ KEY COLUMN : _A1.L_PARTKEY
             HASH FILTER : _A1.L_PARTKEY = _A2.P_PARTKEY
           FETCH ONE ROW
    12  -  HASH SHARD ( # 3 )
           READ COLUMN : _A1.L_PARTKEY, _A1.L_SHIPDATE
             PHYSICAL FILTER : _A1.L_SHIPDATE = :_V0

<<<  end print plan
```

如上述execution plan所示无法将子查询累积到pusher table因此以cloned pusher table生成part后执行了remote join

PUSHER_OUTQ

描述PUSHER_OUTQ hint时Optimizer将子查询转换为保证相同结果的join语句并在remote执行其join时在pusher table加载子查询的outer table

以下为使用PUSHER_OUTQ hint的示例

```
\EXPLAIN PLAN
```

```
SELECT p_name
  FROM part
 WHERE p_partkey IN ( SELECT /*+ PUSHER_OUTQ */
                              l_partkey
                        FROM lineitem
                       WHERE l_shipdate = date'1998-12-01'
                     );


< Execution Plan >
================================================================================
==
|IDX|  NODE DESCRIPTION                              |                ROWS
|
--------------------------------------------------------------------------------
--
| 0 |  SELECT STATEMENT                              |                  18
|
| 1 |    QUERY BLOCK ("$QB_IDX_2")                   |                  18
|
| 2 |      HASH JOIN (SEMI)                          |                  18
|
| 3 |        PLAN BASED CLUSTER                      | LOCAL/REMOTE 200000
|
| 4 |          TABLE ACCESS ("PART")                 |               66675
|
```

```
| 5 |        HASH JOIN INSTANT (UNIQUE)              |                  18
|
| 6 |          PLAN BASED CLUSTER                    |     LOCAL/REMOTE 18
|
| 7 |            TABLE ACCESS ("LINEITEM")           |                   7
|
================================================================================
==


     1  -  TARGET : PART.P_NAME
     2  -  JOINED COLUMN : PART.P_NAME
     3  -  SQL : SELECT /*+ FULL( _A1 ) */
                         "_A1"."P_PARTKEY", "_A1"."P_NAME"
                    FROM "PUBLIC"."PART"@LOCAL AS "_A1"
           TARGET DOMAIN : G1(G1N1,G1N2) 66675 rows,
                           G2(G2N1,G2N2) 66664 rows,
                           G3(G3N1,G3N2) 66661 rows
     4  -  HASH SHARD ( # 3 )
           READ COLUMN : PART.P_PARTKEY, PART.P_NAME
     5  -  HASH KEY : LINEITEM.L_PARTKEY
           READ KEY COLUMN : LINEITEM.L_PARTKEY
             HASH FILTER : LINEITEM.L_PARTKEY = PART.P_PARTKEY
           FETCH ONE ROW
     6  -  SQL : SELECT /*+ FULL( _A1 ) */
                         "_A1"."L_PARTKEY"
```

```
                    FROM "PUBLIC"."LINEITEM"@LOCAL AS "_A1"
                   WHERE "_A1"."L_SHIPDATE" = :_V0
            TARGET DOMAIN : G1(G1N1,G1N2) 7 rows,
                            G2(G2N1,G2N2) 4 rows,
                            G3(G3N1,G3N2) 7 rows
      7   -   HASH SHARD ( # 3 )
            READ COLUMN : LINEITEM.L_PARTKEY, LINEITEM.L_SHIPDATE
              PHYSICAL FILTER : LINEITEM.L_SHIPDATE = DATE'1998-12-01'


<<<  end print plan
```

PUSHER_OUTQ为remote semi join时适用的hint

Local semi join的cost比应用PUSHER_OUTQ的remote semi join的cost更佳时选择local semi join

因此即使应用了hint也无法通过execution plan查看

因此如下同时使用REMOTE_UNNEST hint和PUSHER_OUTQ hint才可应用hint

```
\EXPLAIN PLAN
SELECT p_name
  FROM part
 WHERE p_partkey IN ( SELECT /*+ REMOTE_UNNEST PUSHER_OUTQ */
                              l_partkey
                        FROM lineitem
                       WHERE l_shipdate = date'1998-12-01'
                     );
```

```
>>>  start print plan


< Execution Plan >

================================================================================
==
| IDX|  NODE DESCRIPTION                              |                ROWS
|
--------------------------------------------------------------------------------
--
|  0 |  SELECT STATEMENT                              |                  18
|
|  1 |    QUERY BLOCK ("$QB_IDX_2")                   |                  18
|
|  2 |      MULTIPLE CLUSTER                          |     LOCAL/REMOTE 18
|
|  3 |        CLUSTER PUSHER ("_$NI_5")               |              200000
|
|  4 |          PLAN BASED CLUSTER                    | LOCAL/REMOTE 200000
|
|  5 |            TABLE ACCESS ("PART")                           |  66675
|
|  6 |        SELECT STATEMENT                                    |      7
|
|  7 |          QUERY BLOCK ("$QB_IDX_2")                         |      7
|
```

```
|  8 |            SORT INSTANT                                    |      7
|
|  9 |              HASH JOIN (SEMI)                              |      7
|
| 10 |                PUSHER TABLE ACCESS ("_$NI_5" AS _A2)       | 200000
|
| 11 |                HASH JOIN INSTANT (UNIQUE)                  |      7
|
| 12 |                  TABLE ACCESS ("LINEITEM" AS _A1)          |      7
|
=============================================================================
==


     1  -  TARGET : _$NI_5.P_NAME
     2  -  SQL : SELECT /*+ KEEP_JOINED_TABLE
                            USE_HASH_IN( _A1, 396 )
                            FULL( _A2 )
                            FULL( _A1 )
                          */
                           "_A2"."P_PARTKEY", "_A2"."P_NAME"
                    FROM ( "SESSION_SCHEMA"."_$NI_5"@LOCAL AS "_A2"
                             SEMI JOIN
                             "PUBLIC"."LINEITEM"@"G1N1"|"G1N2"|
                                                 "G2N1"|"G2N2"|
                                                 "G3N1"|"G3N2" AS "_A1"
```

```
                        ON  "_A1"."L_PARTKEY" = "_A2"."P_PARTKEY"
                        AND "_A1"."L_SHIPDATE" = :_V0
                      ) ALIAS "_A3"
                ORDER BY "_A2"."P_PARTKEY" ASC NULLS LAST
           TARGET DOMAIN : G1(G1N1,G1N2) 7 rows,
                           G2(G2N1,G2N2) 4 rows,
                           G3(G3N1,G3N2) 7 rows
           DISTINCT KEY GROUP
             KEY GROUP : _$NI_5.P_PARTKEY
     3  -  SQL : DECLARE INSTANT TABLE "SESSION_SCHEMA"."_$NI_5"
                 ( "P_PARTKEY" NUMBER(10, 0), "P_NAME" VARCHAR(55
OCTETS) )
           COLUMN : PART.P_PARTKEY AS P_PARTKEY, PART.P_NAME AS P_NAME
           CLONED
           TARGET DOMAIN : G1(G1N1,G1N2) 200000 rows,
                           G2(G2N1,G2N2) 200000 rows,
                           G3(G3N1,G3N2) 200000 rows
     4  -  SQL : SELECT /*+ FULL( _A1 ) */
                        "_A1"."P_PARTKEY", "_A1"."P_NAME"
                  FROM "PUBLIC"."PART"@LOCAL AS "_A1"
           TARGET DOMAIN : G1(G1N1,G1N2) 66675 rows,
                           G2(G2N1,G2N2) 66664 rows,
                           G3(G3N1,G3N2) 66661 rows
     5  -  HASH SHARD ( # 3 )
           READ COLUMN : PART.P_PARTKEY, PART.P_NAME
```

```
 7  -  TARGET : _A2.P_PARTKEY, _A2.P_NAME
 8  -  SORT KEY : "_A2.P_PARTKEY ASC NULLS LAST"
       RECORD COLUMN : _A2.P_NAME
       READ KEY COLUMN : _A2.P_PARTKEY
       READ RECORD COLUMN : _A2.P_NAME
 9  -  JOINED COLUMN : _A2.P_PARTKEY, _A2.P_NAME
10  -  READ COLUMN : _A2.P_PARTKEY, _A2.P_NAME
11  -  HASH KEY : _A1.L_PARTKEY
       READ KEY COLUMN : _A1.L_PARTKEY
         HASH FILTER : _A1.L_PARTKEY = _A2.P_PARTKEY
       FETCH ONE ROW
12  -  HASH SHARD ( # 3 )
       READ COLUMN : _A1.L_PARTKEY, _A1.L_SHIPDATE
         PHYSICAL FILTER : _A1.L_SHIPDATE = :_V0


<<<  end print plan
```

如上述 execution plan所示以remote semi join执行并将为outer query的table的part部署到了pusher table

NO_PUSHER_OUTQ

描述NO_PUSHER_OUTQ hint时Optimizer将子查询转换为可保证相同结果的join语句后在remote执行其join时不可以在pusher table部署子查询的outer table

以下为使用NO_PUSHER_OUTQ hint的示例

```
\EXPLAIN PLAN
SELECT p_name
  FROM part
 WHERE p_partkey IN ( SELECT /*+ NO_PUSHER_OUTQ */
                             l_partkey
                        FROM lineitem
                       WHERE l_shipdate = date'1998-12-01'
                     );


>>>  start print plan


< Execution Plan >
================================================================================
==
| IDX |  NODE DESCRIPTION                                                 | ROWS
|
--------------------------------------------------------------------------------
--
|  0 |  SELECT STATEMENT                                                 |    1
|
|  1 |    QUERY BLOCK ("$QB_IDX_2")                                      |    1
|
|  2 |      SINGLE CLUSTER                                  |  LOCAL/REMOTE 1
|
|  3 |        CLUSTER PUSHER ("_$NI_7")                                  |   18
```

```
|
|  4 |          PLAN BASED CLUSTER                          | LOCAL/REMOTE 18
|
|  5 |            TABLE ACCESS ("LINEITEM")                          |    7
|
|  6 |        SELECT STATEMENT                                       |    1
|
|  7 |          QUERY BLOCK ("$QB_IDX_2")                            |    1
|
|  8 |            AGGREGATION BY HASH                                |    1
|
|  9 |              NESTED JOIN (INVERTED SEMI)                      |    5
|
| 10 |                SORT JOIN INSTANT (UNIQUE)                     |    5
|
| 11 |                  PUSHER TABLE ACCESS ("_$NI_7" AS _A2)        |    5
|
| 12 |                INDEX ACCESS ("PART" AS _A1, "PART_PK_INDEX") |(5) 5
|
================================================================================
==

     1  -  TARGET : COUNT(*)
     2  -  SQL : SELECT /*+ KEEP_JOINED_TABLE
                            USE_NL_IN( _A1 )
```

```
                    FULL( _A2 )
                    INDEX( _A1, "PUBLIC"."PART_PK_INDEX" )
                */
                COUNT(*)
            FROM ( "PUBLIC"."PART"@"G1N1"|"G1N2"|
                                "G2N1"|"G2N2"|
                                "G3N1"|"G3N2" AS "_A1"
                 SEMI JOIN
                "SESSION_SCHEMA"."_$NI_7"@LOCAL AS "_A2"
                 ON "_A1"."P_PARTKEY" = "_A2"."L_PARTKEY"
              ) ALIAS "_A3"
    TARGET DOMAIN : G1(G1N1,G1N2) 1 rows,
                    G2(G2N1,G2N2) 1 rows,
                    G3(G3N1,G3N2) 1 rows
    RE-AGGREGATION
      AGGREGATION : SUM( COUNT(*) )
3 - SQL : DECLARE INSTANT TABLE "SESSION_SCHEMA"."_$NI_7"
          ( "L_PARTKEY" NUMBER(10, 0) )
    COLUMN : LINEITEM.L_PARTKEY AS L_PARTKEY
    SHARDED : LINEITEM.L_PARTKEY
    TARGET DOMAIN : G1(G1N1,G1N2) 5 rows,
                    G2(G2N1,G2N2) 9 rows,
                    G3(G3N1,G3N2) 4 rows
4 - SQL : SELECT /*+ FULL( _A1 ) */
                "_A1"."L_PARTKEY"
```

```
                  FROM "PUBLIC"."LINEITEM"@LOCAL AS "_A1"
                 WHERE "_A1"."L_SHIPDATE" = :_V0
           TARGET DOMAIN : G1(G1N1,G1N2) 7 rows,
                           G2(G2N1,G2N2) 4 rows,
                           G3(G3N1,G3N2) 7 rows
     5   -  HASH SHARD ( # 3 )
           READ COLUMN : LINEITEM.L_PARTKEY, LINEITEM.L_SHIPDATE
             PHYSICAL FILTER : LINEITEM.L_SHIPDATE = DATE'1998-12-01'
     7   -  TARGET : COUNT(*)
     8   -  AGGREGATION : COUNT(*)
     9   -  JOINED COLUMN : NOTHING
    10   -  SORT KEY : "_A2.L_PARTKEY ASC NULLS LAST"
           READ KEY COLUMN : _A2.L_PARTKEY
    11   -  READ COLUMN : _A2.L_PARTKEY
    12   -  HASH SHARD ( # 3 )
           READ INDEX COLUMN : _A1.P_PARTKEY
             MIN RANGE : _A1.P_PARTKEY = {_A2.L_PARTKEY}
             MAX RANGE : _A1.P_PARTKEY = {_A2.L_PARTKEY}
           FETCH ONE ROW


<<<  end print plan
```

如上述execution plan所示未在pusher table部署outer query的table的partunnest子查询的table的lineitem部署到了pusher table

<unnest merge hints>

关于子查询unnest到view时是否merge该view的hint

MERGE_SUBQ

子查询unnest到view时merge该view

以下为使用MERGE_SUBQ hint的示例

```
\EXPLAIN PLAN
SELECT
       o_orderdate,
       o_totalprice
  FROM orders
 WHERE o_orderkey IN (
                      SELECT   /*+ MERGE_SUBQ */
                               l_orderkey
                        FROM   lineitem
                      GROUP BY l_orderkey
                    )
;

no rows selected.

>>>  start print plan
```

```
< Execution Plan >

================================================================================
=
|  IDX  |  NODE DESCRIPTION
|
--------------------------------------------------------------------------------
-
|    0  |  SELECT STATEMENT
|
|    1  |    QUERY BLOCK ("$QB_IDX_2")
|
|    2  |      INLINE_VIEW ("$V3")
|
|    3  |        QUERY BLOCK ("$QB_IDX_6")
|
|    4  |          GROUP HASH INSTANT
|
|    5  |            HASH JOIN (INNER JOIN)
|
|    6  |              TABLE ACCESS ("ORDERS")
|
|    7  |              HASH JOIN INSTANT
|
|    8  |                INDEX ACCESS ("LINEITEM", "LINEITEM_PK_INDEX")
|
```

```
================================================================================
=


     1  -  TARGET : $V3.O_ORDERDATE, $V3.O_TOTALPRICE
     2  -  COLUMN : O_TOTALPRICE AS O_TOTALPRICE, O_ORDERDATE AS
O_ORDERDATE
     3  -  TARGET : MAX( ORDERS.O_TOTALPRICE ) AS O_TOTALPRICE,
MAX( ORDERS.O_ORDERDATE ) AS O_ORDERDATE
     4  -  GROUP KEY : LINEITEM.L_ORDERKEY, ORDERS.$PHYSICAL_ROWID
           RECORD COLUMN : MAX( ORDERS.O_TOTALPRICE ),
MAX( ORDERS.O_ORDERDATE )
           READ RECORD COLUMN : MAX( ORDERS.O_TOTALPRICE ),
MAX( ORDERS.O_ORDERDATE )
     5  -  JOINED COLUMN : LINEITEM.L_ORDERKEY, ORDERS.$PHYSICAL_ROWID,
ORDERS.O_TOTALPRICE, ORDERS.O_ORDERDATE
     6  -  READ COLUMN : ORDERS.O_ORDERKEY, ORDERS.O_TOTALPRICE,
ORDERS.O_ORDERDATE
     7  -  HASH KEY : LINEITEM.L_ORDERKEY
           READ KEY COLUMN : LINEITEM.L_ORDERKEY
             HASH FILTER : LINEITEM.L_ORDERKEY = ORDERS.O_ORDERKEY
     8  -  READ INDEX COLUMN : LINEITEM.L_ORDERKEY


<<<  end print plan
```

如上述execution plan所示子查询unnest到view时该view进行了complex view merging

NO_MERGE_SUBQ

子查询unnest到view时不merge该view

以下为使用NO_MERGE_SUBQ hint的示例

```
\EXPLAIN PLAN
SELECT
       o_orderdate,
       o_totalprice
  FROM orders
 WHERE o_orderkey IN (
                      SELECT   /*+ NO_MERGE_SUBQ */
                               l_orderkey
                        FROM   lineitem
                      GROUP BY l_orderkey
                    )
;

>>>  start print plan

< Execution Plan >
================================================================================
|  IDX  |  NODE DESCRIPTION                                                    |
--------------------------------------------------------------------------------
|    0  |  SELECT STATEMENT                                                    |
```

```
|    1  |    QUERY BLOCK ("$QB_IDX_2")                                        |
|    2  |      NESTED JOIN (INNER JOIN)                                       |
|    3  |        INLINE_VIEW ("$V4")                                          |
|    4  |          QUERY BLOCK ("$QB_IDX_6")                                  |
|    5  |            GROUP                                                    |
|    6  |              INDEX ACCESS ("LINEITEM", "LINEITEM_PK_INDEX")         |
|    7  |        INDEX ACCESS ("ORDERS", "ORDERS_PK_INDEX")                   |
=========================================================================


     1  -  TARGET : ORDERS.O_ORDERDATE, ORDERS.O_TOTALPRICE
     2  -  JOINED COLUMN : ORDERS.O_ORDERDATE, ORDERS.O_TOTALPRICE
     3  -  COLUMN : LINEITEM.L_ORDERKEY AS L_ORDERKEY
     4  -  TARGET : LINEITEM.L_ORDERKEY
     5  -  GROUP KEY : LINEITEM.L_ORDERKEY
     6  -  READ INDEX COLUMN : LINEITEM.L_ORDERKEY
     7  -  READ INDEX COLUMN : ORDERS.O_ORDERKEY
           READ TABLE COLUMN : ORDERS.O_TOTALPRICE, ORDERS.O_ORDERDATE
             MIN RANGE : ORDERS.O_ORDERKEY = {$V4.L_ORDERKEY}
             MAX RANGE : ORDERS.O_ORDERKEY = {$V4.L_ORDERKEY}
           FETCH ONE ROW


<<<  end print plan
```

如上述execution plan所示子查询unnest到view时该view直接执行了nested join

<transitive closure hints>

关于是否适用Join transitive closure方法的hint

Join transitive closure相关详细内容参考**Join Transitive Closure**

TRANSITIVE_CLOSURE

描述TRANSITIVE_CLOSURE hint时在rewriter过程中应用join transitive closure方法

以下为使用TRANSITIVE_CLOSURE hint的示例

```
\EXPLAIN PLAN
SELECT /*+ TRANSITIVE_CLOSURE */
       l_extendedprice * (1 - l_discount) - ps_supplycost * l_quantity as
amount
  FROM part,
       lineitem,
       partsupp
 WHERE ps_suppkey = l_suppkey
   AND ps_partkey = l_partkey
   AND p_partkey = l_partkey
   AND p_name like '%green%';


>>>  start print plan


< Execution Plan >
```

```
================================================================================
==
|  IDX  |  NODE DESCRIPTION                                        |   ROWS
|
--------------------------------------------------------------------------------
--
| 0 |SELECT STATEMENT                                            | 319404
|
| 1 |  QUERY BLOCK ("$QB_IDX_2")                                 | 319404
|
| 2 |    NESTED JOIN (INNER JOIN)                                | 319404
|
| 3 |      NESTED JOIN (INNER JOIN)                              |  42656
|
| 4 |        TABLE ACCESS ("PART")                               |  10664
|
| 5 |        INDEX ACCESS ("PARTSUPP", "PARTSUPP_PARTKEY_FK")    |  42656
|
| 6 |      INDEX ACCESS ("LINEITEM","LINEITEM_PARTKEY_SUPPKEY_FK")| 319404
|
================================================================================
==


     1  -  TARGET : ( LINEITEM.L_EXTENDEDPRICE * ( 1 -
LINEITEM.L_DISCOUNT ) ) - ( PARTSUPP.PS_SUPPLYCOST * LINEITEM.L_QUANTITY )
```

```
AS AMOUNT
     2  -  JOINED COLUMN : LINEITEM.L_EXTENDEDPRICE, LINEITEM.L_DISCOUNT,
PARTSUPP.PS_SUPPLYCOST, LINEITEM.L_QUANTITY
     3  -  JOINED COLUMN : PART.P_PARTKEY, PARTSUPP.PS_SUPPKEY,
PARTSUPP.PS_SUPPLYCOST
     4  -  READ COLUMN : PART.P_PARTKEY, PART.P_NAME
              LOGICAL FILTER : PART.P_NAME LIKE '%green%'
     5  -  READ INDEX COLUMN : PARTSUPP.PS_PARTKEY
            READ TABLE COLUMN : PARTSUPP.PS_SUPPKEY, PARTSUPP.PS_SUPPLYCOST
              MIN RANGE : PARTSUPP.PS_PARTKEY = {PART.P_PARTKEY}
              MAX RANGE : PARTSUPP.PS_PARTKEY = {PART.P_PARTKEY}
     6  -  READ INDEX COLUMN : LINEITEM.L_PARTKEY, LINEITEM.L_SUPPKEY
            READ TABLE COLUMN : LINEITEM.L_QUANTITY,
LINEITEM.L_EXTENDEDPRICE, LINEITEM.L_DISCOUNT
              MIN RANGE : LINEITEM.L_PARTKEY = {PART.P_PARTKEY} AND
LINEITEM.L_SUPPKEY = {PARTSUPP.PS_SUPPKEY}
              MAX RANGE : LINEITEM.L_PARTKEY = {PART.P_PARTKEY} AND
LINEITEM.L_SUPPKEY = {PARTSUPP.PS_SUPPKEY}


<<<  end print plan
```

在上述示例的查询partsupp和part中没有join condition但execution plan中partsupp和part中有join condition (ps_partkey = p_partkey)

用户未描述ps_partkey = p_partkey但通过ps_partkey = l_partkey AND p_partkey = l_partkey生成这样使用join条件生成其他join条件的方法叫做join transitive closure由此可优先join part和

partsupp由于优先执行了中间结果相对少的join因此提高了性能

NO_TRANSITIVE_CLOSURE

描述NO_TRANSITIVE_CLOSURE hint时在rewriter过程中不应用join transitive closure方法

以下为使用NO_TRANSITIVE_CLOSURE hint的示例

```
\EXPLAIN PLAN
SELECT /*+ NO_TRANSITIVE_CLOSURE */
       l_extendedprice * (1 - l_discount) - ps_supplycost * l_quantity as
amount
  FROM part,
       lineitem,
       partsupp
 WHERE ps_suppkey = l_suppkey
   AND ps_partkey = l_partkey
   AND p_partkey = l_partkey
   AND p_name like '%green%';

>>>  start print plan

< Execution Plan >
================================================================================
=
|  IDX  |  NODE DESCRIPTION                                          |  ROWS
```

```
|
-------------------------------------------------------------------------
| 0 |SELECT STATEMENT                                          | 319404
|
| 1 |  QUERY BLOCK ("$QB_IDX_2")                               | 319404
|
| 2 |    HASH JOIN (INNER JOIN)                                | 319404
|
| 3 |      TABLE ACCESS ("PARTSUPP")                           | 800000
|
| 4 |      HASH JOIN INSTANT                                   | 319404
|
| 5 |        NESTED JOIN (INNER JOIN)                          | 319404
|
| 6 |          TABLE ACCESS ("PART")                           |  10664
|
| 7 |          INDEX ACCESS ("LINEITEM","LINEITEM_PARTKEY_SUPPKEY_FK")|
319404 |
=========================================================================
==


     1  -  TARGET : ( LINEITEM.L_EXTENDEDPRICE * ( 1 -
LINEITEM.L_DISCOUNT ) ) - ( PARTSUPP.PS_SUPPLYCOST * LINEITEM.L_QUANTITY )
AS AMOUNT
     2  -  JOINED COLUMN : LINEITEM.L_EXTENDEDPRICE, LINEITEM.L_DISCOUNT,
```

```
PARTSUPP.PS_SUPPLYCOST, LINEITEM.L_QUANTITY
     3  -  READ COLUMN : PARTSUPP.PS_PARTKEY, PARTSUPP.PS_SUPPKEY,
PARTSUPP.PS_SUPPLYCOST
     4  -  HASH KEY : LINEITEM.L_PARTKEY, LINEITEM.L_SUPPKEY
           RECORD COLUMN : LINEITEM.L_EXTENDEDPRICE, LINEITEM.L_DISCOUNT,
LINEITEM.L_QUANTITY
           READ KEY COLUMN : LINEITEM.L_PARTKEY, LINEITEM.L_SUPPKEY,
LINEITEM.L_EXTENDEDPRICE, LINEITEM.L_DISCOUNT, LINEITEM.L_QUANTITY
             HASH FILTER : LINEITEM.L_PARTKEY = PARTSUPP.PS_PARTKEY AND
LINEITEM.L_SUPPKEY = PARTSUPP.PS_SUPPKEY
     5  -  JOINED COLUMN : LINEITEM.L_PARTKEY, LINEITEM.L_SUPPKEY,
LINEITEM.L_EXTENDEDPRICE, LINEITEM.L_DISCOUNT, LINEITEM.L_QUANTITY
     6  -  READ COLUMN : PART.P_PARTKEY, PART.P_NAME
             LOGICAL FILTER : PART.P_NAME LIKE '%green%'
     7  -  READ INDEX COLUMN : LINEITEM.L_PARTKEY, LINEITEM.L_SUPPKEY
           READ TABLE COLUMN : LINEITEM.L_QUANTITY,
LINEITEM.L_EXTENDEDPRICE, LINEITEM.L_DISCOUNT
             MIN RANGE : LINEITEM.L_PARTKEY = {PART.P_PARTKEY}
             MAX RANGE : LINEITEM.L_PARTKEY = {PART.P_PARTKEY}

<<<  end print plan
```

上述示例中TRANSITIVE_CLOSURE hint的示例和查询语句相同但execution plan不同这是因为仅通过用户描述的join condition生成了执行计划

统计信息准确的情况上述execution plan的查询执行时间可能长于TRANSITIVE_CLOSURE hint的

示例这是因为join的中间结果数量多

但是统计信息不准确导致lineitem的row数量更少时仅通过用户描述的join condition生成执行计划会更有助于性能所以此时使用NO_TRANSITIVE_CLOSURE hint

<view hints>

<view merge hints>

关于在rewriter过程中是否适用view merging方法的hint

MERGE(view_name)

描述MERGE(view_name) hint时merge拥有view_name的view和outer query

以下为使用MERGE(view_name) hint的示例

```
CREATE OR REPLACE VIEW v_nation
  (
     v_nationkey,
     v_nation_name,
     v_region_name
  )
AS SELECT n_nationkey,
          n_name,
          r_name
     FROM nation,
          region
```

```
    WHERE n_regionkey = r_regionkey;



\EXPLAIN PLAN
  SELECT /*+ MERGE(v_nation) */
         v_nation_name,
         count(*)
    FROM customer,
         v_nation
   WHERE c_nationkey = v_nationkey
     AND v_region_name = 'ASIA'
GROUP BY v_nation_name ;


>>>  start print plan


< Execution Plan >
=============================================================================
|  IDX  |  NODE DESCRIPTION                                                 |
-----------------------------------------------------------------------------
|    0  |  SELECT STATEMENT                                                 |
|    1  |    QUERY BLOCK ("$QB_IDX_2")                                      |
|    2  |      GROUP HASH INSTANT                                           |
|    3  |        NESTED JOIN (INNER JOIN)                                   |
|    4  |          NESTED JOIN (INNER JOIN)                                 |
|    5  |            TABLE ACCESS ("REGION")                                |
```

```
|    6  |            INDEX ACCESS ("NATION", "NATION_REGIONKEY_FK")     |
|    7  |          INDEX ACCESS ("CUSTOMER", "CUSTOMER_NATIONKEY_FK")   |
=========================================================================


     1  -  TARGET : NATION.N_NAME, COUNT(*)
     2  -  GROUP KEY : NATION.N_NAME
           RECORD COLUMN : COUNT(*)
           READ KEY COLUMN : NATION.N_NAME
           READ RECORD COLUMN : COUNT(*)
     3  -  JOINED COLUMN : NATION.N_NAME
     4  -  JOINED COLUMN : NATION.N_NATIONKEY, NATION.N_NAME
     5  -  READ COLUMN : REGION.R_REGIONKEY, REGION.R_NAME
             PHYSICAL FILTER : REGION.R_NAME = 'ASIA'
     6  -  READ INDEX COLUMN : NATION.N_REGIONKEY
           READ TABLE COLUMN : NATION.N_NATIONKEY, NATION.N_NAME
             MIN RANGE : NATION.N_REGIONKEY = {REGION.R_REGIONKEY}
             MAX RANGE : NATION.N_REGIONKEY = {REGION.R_REGIONKEY}
     7  -  READ INDEX COLUMN : CUSTOMER.C_NATIONKEY
             MIN RANGE : CUSTOMER.C_NATIONKEY = {NATION.N_NATIONKEY}
             MAX RANGE : CUSTOMER.C_NATIONKEY = {NATION.N_NATIONKEY}


<<<  end print plan
```

如上述execution plan所示v_nation merging到了outer query

NO_MERGE(view_name)

描述NO_MERGE(view_name) hint时不merge拥有view_name的view和outer query

以下为使用NO_MERGE(view_name) hint的示例

```
CREATE OR REPLACE VIEW v_nation
  (
     v_nationkey,
     v_nation_name,
     v_region_name
  )
AS SELECT n_nationkey,
          n_name,
          r_name
     FROM nation,
          region
    WHERE n_regionkey = r_regionkey;
```

- 不进行view merge

```
\EXPLAIN PLAN
  SELECT /*+ NO_MERGE(v_nation) */
         v_nation_name,
         count(*)
    FROM customer,
         v_nation
```

```
   WHERE c_nationkey = v_nationkey
     AND v_region_name = 'ASIA'
GROUP BY v_nation_name ;


>>>  start print plan


< Execution Plan >
================================================================================
=
|  IDX  |  NODE DESCRIPTION
|
--------------------------------------------------------------------------------
-
|    0  |  SELECT STATEMENT
|
|    1  |    QUERY BLOCK ("$QB_IDX_2")
|
|    2  |      GROUP HASH INSTANT
|
|    3  |        NESTED JOIN (INNER JOIN)
|
|    4  |          VIEW ("V_NATION")                                    |
|    5  |            QUERY BLOCK ("$QB_IDX_7")
|
|    6  |              NESTED JOIN (INNER JOIN)
```

```
|
|    7  |                TABLE ACCESS ("REGION")
|
|    8  |                INDEX ACCESS ("NATION", "NATION_REGIONKEY_FK")
|
|    9  |          INDEX ACCESS ("CUSTOMER", "CUSTOMER_NATIONKEY_FK")
|
================================================================================
=


     1  -  TARGET : V_NATION.V_NATION_NAME, COUNT(*)
     2  -  GROUP KEY : V_NATION.V_NATION_NAME
           RECORD COLUMN : COUNT(*)
           READ KEY COLUMN : V_NATION.V_NATION_NAME
           READ RECORD COLUMN : COUNT(*)
     3  -  JOINED COLUMN : V_NATION.V_NATION_NAME
     4  -  COLUMN : V_NATIONKEY AS V_NATIONKEY, V_NATION_NAME AS
V_NATION_NAME, V_REGION_NAME AS V_REGION_NAME
     5  -  TARGET : NATION.N_NATIONKEY AS V_NATIONKEY, NATION.N_NAME AS
V_NATION_NAME, REGION.R_NAME AS V_REGION_NAME
     6  -  JOINED COLUMN : NATION.N_NATIONKEY, NATION.N_NAME,
REGION.R_NAME
     7  -  READ COLUMN : REGION.R_REGIONKEY, REGION.R_NAME
             PHYSICAL FILTER : REGION.R_NAME = 'ASIA'
     8  -  READ INDEX COLUMN : NATION.N_REGIONKEY
```

```
            READ TABLE COLUMN : NATION.N_NATIONKEY, NATION.N_NAME
              MIN RANGE : NATION.N_REGIONKEY = {REGION.R_REGIONKEY}
              MAX RANGE : NATION.N_REGIONKEY = {REGION.R_REGIONKEY}
      9   -   READ INDEX COLUMN : CUSTOMER.C_NATIONKEY
              MIN RANGE : CUSTOMER.C_NATIONKEY = {V_NATION.V_NATIONKEY}
              MAX RANGE : CUSTOMER.C_NATIONKEY = {V_NATION.V_NATIONKEY}


<<<  end print plan
```

如上述execution plan所示可看到v_nation未merging到outer query并以view形式存在

<push view predicate hints>

用于使与view相关的join predicate push到view中或不允许push的hint

PUSH_PRED

描述PUSH_PRED hint时将from子句中列出的与view相关的所有join predicate push到view中

以下为使用PUSH_PRED hint的示例

```
CREATE OR REPLACE VIEW v_nation
  (
     v_nationkey,
     v_nation_name,
     v_region_name
  )
AS SELECT n_nationkey,
```

```
          n_name,
          r_name
     FROM nation,
          region
    WHERE n_regionkey = r_regionkey;


\EXPLAIN PLAN
  SELECT /*+ PUSH_PRED */
         v_nation_name,
         count(*)
    FROM customer,
         v_nation
   WHERE c_nationkey = v_nationkey
     AND v_region_name = 'ASIA'
GROUP BY v_nation_name;


>>>  start print plan


< Execution Plan >
================================================================================
|  IDX  |  NODE DESCRIPTION                                                    |
--------------------------------------------------------------------------------
|    0  |  SELECT STATEMENT                                                    |
|    1  |    QUERY BLOCK ("$QB_IDX_2")                                         |
|    2  |      GROUP HASH INSTANT                                              |
```

```
|    3  |        NESTED JOIN (INNER JOIN)                                  |
|    4  |          INDEX ACCESS ("CUSTOMER", "CUSTOMER_NATIONKEY_FK")      |
|    5  |          VIEW ("V_NATION")                                        |
|    6  |            QUERY BLOCK ("$QB_IDX_7")                              |
|    7  |              NESTED JOIN (INNER JOIN)                             |
|    8  |                INDEX ACCESS ("NATION", "NATION_PK_INDEX")         |
|    9  |                INDEX ACCESS ("REGION", "REGION_PK_INDEX")         |
=============================================================================


     1  -  TARGET : V_NATION.V_NATION_NAME, COUNT(*)
     2  -  GROUP KEY : V_NATION.V_NATION_NAME
           RECORD COLUMN : COUNT(*)
           READ KEY COLUMN : V_NATION.V_NATION_NAME
           READ RECORD COLUMN : COUNT(*)
     3  -  JOINED COLUMN : V_NATION.V_NATION_NAME
     4  -  READ INDEX COLUMN : CUSTOMER.C_NATIONKEY
     5  -  COLUMN : V_NATIONKEY AS V_NATIONKEY, V_NATION_NAME AS
V_NATION_NAME, V_REGION_NAME AS V_REGION_NAME
     6  -  TARGET : NATION.N_NATIONKEY AS V_NATIONKEY, NATION.N_NAME AS
V_NATION_NAME, REGION.R_NAME AS V_REGION_NAME
     7  -  JOINED COLUMN : NATION.N_NATIONKEY, NATION.N_NAME,
REGION.R_NAME
     8  -  READ INDEX COLUMN : NATION.N_NATIONKEY
           READ TABLE COLUMN : NATION.N_NAME, NATION.N_REGIONKEY
             MIN RANGE : NATION.N_NATIONKEY = {CUSTOMER.C_NATIONKEY}
```

```
              MAX RANGE : NATION.N_NATIONKEY = {CUSTOMER.C_NATIONKEY}
            FETCH ONE ROW
     9   -  READ INDEX COLUMN : REGION.R_REGIONKEY
            READ TABLE COLUMN : REGION.R_NAME
              MIN RANGE : REGION.R_REGIONKEY = {NATION.N_REGIONKEY}
              MAX RANGE : REGION.R_REGIONKEY = {NATION.N_REGIONKEY}
              PHYSICAL TABLE FILTER : REGION.R_NAME = 'ASIA'
            FETCH ONE ROW

<<<  end print plan
```

如上述 execution plan所示c_nationkey = v_nationkey push到niew内部使用

NO_PUSH_PRED

描述NO_PUSH_PRED hint时未将from子句中列出的与view相关的join predicate push到view中

以下为使用NO_PUSH_PRED hint的示例

```
CREATE OR REPLACE VIEW v_nation
  (
     v_nationkey,
     v_nation_name,
     v_region_name
  )
AS SELECT n_nationkey,
          n_name,
```

```
          r_name
     FROM nation,
          region
    WHERE n_regionkey = r_regionkey;


\EXPLAIN PLAN
  SELECT /*+ NO_PUSH_PRED */
         v_nation_name,
         count(*)
    FROM customer,
         v_nation
   WHERE c_nationkey = v_nationkey
     AND v_region_name = 'ASIA'
GROUP BY v_nation_name;


>>>  start print plan


< Execution Plan >
================================================================================
=
|  IDX  |  NODE DESCRIPTION
|
--------------------------------------------------------------------------------
-
|    0  |  SELECT STATEMENT
```

```
|
|    1  |    QUERY BLOCK ("$QB_IDX_2")
|
|    2  |      GROUP HASH INSTANT
|
|    3  |        NESTED JOIN (INNER JOIN)
|
|    4  |          VIEW ("V_NATION")
|
|    5  |            QUERY BLOCK ("$QB_IDX_7")
|
|    6  |              NESTED JOIN (INNER JOIN)
|
|    7  |                TABLE ACCESS ("REGION")
|
|    8  |                INDEX ACCESS ("NATION", "NATION_REGIONKEY_FK")
|
|    9  |          INDEX ACCESS ("CUSTOMER", "CUSTOMER_NATIONKEY_FK")
|
================================================================================
=


     1  -  TARGET : V_NATION.V_NATION_NAME, COUNT(*)
     2  -  GROUP KEY : V_NATION.V_NATION_NAME
           RECORD COLUMN : COUNT(*)
```

```
            READ KEY COLUMN : V_NATION.V_NATION_NAME
            READ RECORD COLUMN : COUNT(*)
     3   -  JOINED COLUMN : V_NATION.V_NATION_NAME
     4   -  COLUMN : V_NATIONKEY AS V_NATIONKEY, V_NATION_NAME AS
V_NATION_NAME, V_REGION_NAME AS V_REGION_NAME
     5   -  TARGET : NATION.N_NATIONKEY AS V_NATIONKEY, NATION.N_NAME AS
V_NATION_NAME, REGION.R_NAME AS V_REGION_NAME
     6   -  JOINED COLUMN : NATION.N_NATIONKEY, NATION.N_NAME,
REGION.R_NAME
     7   -  READ COLUMN : REGION.R_REGIONKEY, REGION.R_NAME
              PHYSICAL FILTER : REGION.R_NAME = 'ASIA'
     8   -  READ INDEX COLUMN : NATION.N_REGIONKEY
            READ TABLE COLUMN : NATION.N_NATIONKEY, NATION.N_NAME
              MIN RANGE : NATION.N_REGIONKEY = {REGION.R_REGIONKEY}
              MAX RANGE : NATION.N_REGIONKEY = {REGION.R_REGIONKEY}
     9   -  READ INDEX COLUMN : CUSTOMER.C_NATIONKEY
              MIN RANGE : CUSTOMER.C_NATIONKEY = {V_NATION.V_NATIONKEY}
              MAX RANGE : CUSTOMER.C_NATIONKEY = {V_NATION.V_NATIONKEY}


<<<  end print plan
```

如上述execution plan所示c_nationkey = v_nationkey在view外部使用

PUSH_PRED(view_name[[,] view_name])

描述PUSH_PRED(view_name[[,] view_name]) hint时将from子句中列出的与view相关的所有

join predicate push到view中

以下为使用PUSH_PRED(view_name[[,] view_name]) hint的示例

```
CREATE OR REPLACE VIEW v_nation
  (
     v_nationkey,
     v_nation_name,
     v_region_name
  )
AS SELECT n_nationkey,
          n_name,
          r_name
     FROM nation,
          region
    WHERE n_regionkey = r_regionkey;


\EXPLAIN PLAN
  SELECT /*+ PUSH_PRED(v_nation) */
         v_nation_name,
         count(*)
    FROM customer,
         v_nation
   WHERE c_nationkey = v_nationkey
     AND v_region_name = 'ASIA'
GROUP BY v_nation_name;
```

```
>>>  start print plan


< Execution Plan >

================================================================================
|  IDX  |  NODE DESCRIPTION                                                    |
--------------------------------------------------------------------------------
|    0  |  SELECT STATEMENT                                                    |
|    1  |    QUERY BLOCK ("$QB_IDX_2")                                         |
|    2  |      GROUP HASH INSTANT                                              |
|    3  |        NESTED JOIN (INNER JOIN)                                      |
|    4  |          INDEX ACCESS ("CUSTOMER", "CUSTOMER_NATIONKEY_FK")          |
|    5  |          VIEW ("V_NATION")                                           |
|    6  |            QUERY BLOCK ("$QB_IDX_7")                                 |
|    7  |              NESTED JOIN (INNER JOIN)                                |
|    8  |                INDEX ACCESS ("NATION", "NATION_PK_INDEX")             |
|    9  |                INDEX ACCESS ("REGION", "REGION_PK_INDEX")             |
================================================================================


     1  -  TARGET : V_NATION.V_NATION_NAME, COUNT(*)
     2  -  GROUP KEY : V_NATION.V_NATION_NAME
           RECORD COLUMN : COUNT(*)
           READ KEY COLUMN : V_NATION.V_NATION_NAME
           READ RECORD COLUMN : COUNT(*)
     3  -  JOINED COLUMN : V_NATION.V_NATION_NAME
```

```
     4  -  READ INDEX COLUMN : CUSTOMER.C_NATIONKEY
     5  -  COLUMN : V_NATIONKEY AS V_NATIONKEY, V_NATION_NAME AS
V_NATION_NAME, V_REGION_NAME AS V_REGION_NAME
     6  -  TARGET : NATION.N_NATIONKEY AS V_NATIONKEY, NATION.N_NAME AS
V_NATION_NAME, REGION.R_NAME AS V_REGION_NAME
     7  -  JOINED COLUMN : NATION.N_NATIONKEY, NATION.N_NAME,
REGION.R_NAME
     8  -  READ INDEX COLUMN : NATION.N_NATIONKEY
           READ TABLE COLUMN : NATION.N_NAME, NATION.N_REGIONKEY
             MIN RANGE : NATION.N_NATIONKEY = {CUSTOMER.C_NATIONKEY}
             MAX RANGE : NATION.N_NATIONKEY = {CUSTOMER.C_NATIONKEY}
           FETCH ONE ROW
     9  -  READ INDEX COLUMN : REGION.R_REGIONKEY
           READ TABLE COLUMN : REGION.R_NAME
             MIN RANGE : REGION.R_REGIONKEY = {NATION.N_REGIONKEY}
             MAX RANGE : REGION.R_REGIONKEY = {NATION.N_REGIONKEY}
             PHYSICAL TABLE FILTER : REGION.R_NAME = 'ASIA'
           FETCH ONE ROW


<<<  end print plan
```

NO_PUSH_PRED(view_name[[,] view_name])

描述NO_PUSH_PRED(view_name[[,] view_name]) hint时不将from子句中列出的与view相关的所有join predicate push到view中

以下为使用NO_PUSH_PRED(view_name[[,] view_name]) hint的示例

```
CREATE OR REPLACE VIEW v_nation
  (
     v_nationkey,
     v_nation_name,
     v_region_name
  )
AS SELECT n_nationkey,
          n_name,
          r_name
     FROM nation,
          region
    WHERE n_regionkey = r_regionkey;


\EXPLAIN PLAN
  SELECT /*+ NO_PUSH_PRED(v_nation) */
         v_nation_name,
         count(*)
    FROM customer,
         v_nation
   WHERE c_nationkey = v_nationkey
     AND v_region_name = 'ASIA'
GROUP BY v_nation_name;
```

```
>>>  start print plan


< Execution Plan >

================================================================================
=
|  IDX  |  NODE DESCRIPTION
|
--------------------------------------------------------------------------------
-
|    0  |  SELECT STATEMENT
|
|    1  |    QUERY BLOCK ("$QB_IDX_2")
|
|    2  |      GROUP HASH INSTANT
|
|    3  |        NESTED JOIN (INNER JOIN)
|
|    4  |          VIEW ("V_NATION")
|
|    5  |            QUERY BLOCK ("$QB_IDX_7")
|
|    6  |              NESTED JOIN (INNER JOIN)
|
|    7  |                TABLE ACCESS ("REGION")
|
```

```
|    8  |                INDEX ACCESS ("NATION", "NATION_REGIONKEY_FK")
|
|    9  |          INDEX ACCESS ("CUSTOMER", "CUSTOMER_NATIONKEY_FK")
|
================================================================================
=


     1  -  TARGET : V_NATION.V_NATION_NAME, COUNT(*)
     2  -  GROUP KEY : V_NATION.V_NATION_NAME
           RECORD COLUMN : COUNT(*)
           READ KEY COLUMN : V_NATION.V_NATION_NAME
           READ RECORD COLUMN : COUNT(*)
     3  -  JOINED COLUMN : V_NATION.V_NATION_NAME
     4  -  COLUMN : V_NATIONKEY AS V_NATIONKEY, V_NATION_NAME AS
V_NATION_NAME, V_REGION_NAME AS V_REGION_NAME
     5  -  TARGET : NATION.N_NATIONKEY AS V_NATIONKEY, NATION.N_NAME AS
V_NATION_NAME, REGION.R_NAME AS V_REGION_NAME
     6  -  JOINED COLUMN : NATION.N_NATIONKEY, NATION.N_NAME,
REGION.R_NAME
     7  -  READ COLUMN : REGION.R_REGIONKEY, REGION.R_NAME
             PHYSICAL FILTER : REGION.R_NAME = 'ASIA'
     8  -  READ INDEX COLUMN : NATION.N_REGIONKEY
           READ TABLE COLUMN : NATION.N_NATIONKEY, NATION.N_NAME
             MIN RANGE : NATION.N_REGIONKEY = {REGION.R_REGIONKEY}
             MAX RANGE : NATION.N_REGIONKEY = {REGION.R_REGIONKEY}
```

```
    9  -  READ INDEX COLUMN : CUSTOMER.C_NATIONKEY
            MIN RANGE : CUSTOMER.C_NATIONKEY = {V_NATION.V_NATIONKEY}
            MAX RANGE : CUSTOMER.C_NATIONKEY = {V_NATION.V_NATIONKEY}


<<<  end print plan
```

Operation Hint

以relation为单位应用的hint

<access path hints>

指定access单张表的方法的hint

FULL(table_name)

描述FULL(table_name) hint时optimizer对描述的table执行table full scan

table_name只能描述一个并存在于<from clause>

以下为使用FULL(table_name) hint的示例

```
SELECT count(*) FROM nation;


COUNT(*)
--------
      25
```

```
1 row selected.



\EXPLAIN PLAN
SELECT /*+ FULL( nation ) */
       n_name
  FROM nation
 WHERE n_nationkey < 10;


>>>  start print plan


< Execution Plan >
=====================================================================
|  IDX  |  NODE DESCRIPTION                                | ROWS |
---------------------------------------------------------------------
|    0  |  SELECT STATEMENT                                |   10 |
|    1  |    QUERY BLOCK ("$QB_IDX_2")                     |   10 |
|    2  |      TABLE ACCESS ("NATION")                     |   10 |
=====================================================================


     1  -  TARGET : NATION.N_NAME
     2  -  READ COLUMN : NATION.N_NATIONKEY, NATION.N_NAME
              PHYSICAL FILTER : NATION.N_NATIONKEY < 10
```

```
<<<  end print plan
```

上述示例中n_nationkey是primary key column因此可以index access但在nation表的所有row数量只有25个的小表中select10条row时table access性能高于index access

INDEX(table_name [,] [index_name [[,] index_name]])

描述INDEX(table_name [,] [index_name [[,] index_name]]) hint时optimizer对描述的table执行index scan此时在列出的index中选择cost最佳的index但未列出index_name时在该表的所有index中选择cost最佳的index

table_name只能描述一个并要存在于<from clause>

index name可描述一个以上或省略应为存在于属于table_name的table的 index_name

以下为使用INDEX(table_name) hint的使用示例

```
\EXPLAIN PLAN
  SELECT /*+ INDEX( nation ) */
         n_nationkey,
         n_name
    FROM nation
   WHERE n_nationkey < 10
ORDER BY n_nationkey;

>>>  start print plan
```

```
< Execution Plan >

=====================================================================

|  IDX  |  NODE DESCRIPTION                                  | ROWS |

---------------------------------------------------------------------

| 0 |  SELECT STATEMENT                                      |  10  |

| 1 |    QUERY BLOCK ("$QB_IDX_2")                           |  10  |

| 2 |      INDEX ACCESS ("NATION", "NATION_PK_INDEX")        |  10  |

=====================================================================


     1  -  TARGET : NATION.N_NATIONKEY, NATION.N_NAME

     2  -  READ INDEX COLUMN : NATION.N_NATIONKEY

           READ TABLE COLUMN : NATION.N_NAME

             MAX RANGE : NATION.N_NATIONKEY < 10


<<<  end print plan
```

上述示例中nation表是整体row的数量只有25个的小表并且是select 10条的情况因此table access的性能比index access好而执行index access则不需要额外处理order by也会导出与执行order by相同的结果因此可省略order by处理所以此时进行index access会更好

NO_INDEX(table_name [,] [index_name [[,] index_name]])

描述NO_INDEX(table_name [,] [index_name [[,] index_name]]) hint时optimizer指定不使用描述的index未列出index_name时不使用所有index即不进行index scan

只能描述一个table_name并要存在于<from clause>

index name可描述一个以上或省略应为存在于属于table_name的table的 index_name

以下为使用NO_INDEX(table_name) hint的示例

```
\EXPLAIN PLAN
  SELECT /*+ NO_INDEX( nation ) */
         n_nationkey,
         n_name
    FROM nation
   WHERE n_nationkey < 10
ORDER BY n_nationkey;


>>>  start print plan


< Execution Plan >
=========================================================================
|  IDX  |  NODE DESCRIPTION                                  | ROWS |
-------------------------------------------------------------------------
|    0  |  SELECT STATEMENT                                  |  10  |
|    1  |    QUERY BLOCK ("$QB_IDX_2")                       |  10  |
|    2  |      SORT INSTANT                                  |  10  |
|    3  |        TABLE ACCESS ("NATION")                     |  10  |
=========================================================================


     1  -  TARGET : NATION.N_NATIONKEY, NATION.N_NAME
     2  -  SORT KEY : "NATION.N_NATIONKEY ASC NULLS LAST"
```

```
            RECORD COLUMN : NATION.N_NAME
            READ KEY COLUMN : NATION.N_NATIONKEY
            READ RECORD COLUMN : NATION.N_NAME
      3   -  READ COLUMN : NATION.N_NATIONKEY, NATION.N_NAME
               PHYSICAL FILTER : NATION.N_NATIONKEY < 10


<<<  end print plan
```

上述实例中使其不使用所有index因此执行了table access

INDEX_FORWARD(table_name [,] [index_name [[,] index_name]])

与INDEX(table_name [,] [index_name[[,] index_name]]) hint相同

指forward scan索引因此以ascending order生成索引时以ascending order输出以descending order生成时以descending order输出

以下为使用INDEX_FORWARD(table_name) hint的示例

```
\EXPLAIN PLAN
  SELECT /*+ INDEX_FORWARD( nation ) */
         n_nationkey,
         n_name
    FROM nation
   WHERE n_nationkey < 10
ORDER BY n_nationkey ASC;
>>>  start print plan
```

```
< Execution Plan >
=====================================================================
|  IDX   |  NODE DESCRIPTION                                  | ROWS |
---------------------------------------------------------------------
|    0   |  SELECT STATEMENT                                  |  10  |
|    1   |    QUERY BLOCK ("$QB_IDX_2")                       |  10  |
|    2   |      INDEX ACCESS ("NATION", "NATION_PK_INDEX")    |  10  |
=====================================================================

     1  -  TARGET : NATION.N_NATIONKEY, NATION.N_NAME
     2  -  READ INDEX COLUMN : NATION.N_NATIONKEY
           READ TABLE COLUMN : NATION.N_NAME
             MAX RANGE : NATION.N_NATIONKEY < 10

<<<  end print plan
```

上述示例中NATION_PK_INDEX以ascending生成时forward scan索引则不需要为了ORDER BY的额外的sorting

INDEX_BACKWARD(table_name [,] [index_name [[,] index_name]])

指backward scan索引因此以ascending order生成索引时以descending order输出以descending order生成时以ascending order输出

语句规则与INDEX(table_name [,] [index_name[[,] index_name]]) hint相同

以下为使用INDEX_BACKWARD(table_name) hint的示例

```
\EXPLAIN PLAN
  SELECT /*+ INDEX_BACKWARD( nation ) */
         n_nationkey,
         n_name
    FROM nation
   WHERE n_nationkey < 10
ORDER BY n_nationkey DESC;


>>>  start print plan


< Execution Plan >
=========================================================================
|  IDX  |  NODE DESCRIPTION                                  | ROWS |
-------------------------------------------------------------------------
|    0  |  SELECT STATEMENT                                  |  10  |
|    1  |    QUERY BLOCK ("$QB_IDX_2")                       |  10  |
|    2  |      INDEX ACCESS ("NATION", "NATION_PK_INDEX")    |  10  |
=========================================================================


     1  -  TARGET : NATION.N_NATIONKEY, NATION.N_NAME
     2  -  READ INDEX COLUMN : NATION.N_NATIONKEY
           READ TABLE COLUMN : NATION.N_NAME
             MAX RANGE : NATION.N_NATIONKEY < 10
```

```
<<<  end print plan
```

上述示例中以ascending生成NATION_PK_INDEX时backward scan索引时不需要为了ORDER BY的额外的sorting

INDEX_ASC(table_name [,] [index_name [[,] index_name]])

与INDEX(table_name [,] [index_name[[,] index_name]]) hint相同

指forward scan索引因此以ascending order生成索引时以ascending order输出以descending order生成时以descending order输出

以下为使用INDEX_ASC(table_name) hint的示例

```
\EXPLAIN PLAN
  SELECT /*+ INDEX_ASC( nation ) */
         n_nationkey,
         n_name
    FROM nation
   WHERE n_nationkey < 10
ORDER BY n_nationkey ASC;
>>>  start print plan

< Execution Plan >
=====================================================================
|  IDX  |  NODE DESCRIPTION                                  | ROWS |
```

```
-----------------------------------------------------------------
|    0  |  SELECT STATEMENT                              |  10  |
|    1  |    QUERY BLOCK ("$QB_IDX_2")                   |  10  |
|    2  |      INDEX ACCESS ("NATION", "NATION_PK_INDEX")|  10  |
=================================================================


     1  -  TARGET : NATION.N_NATIONKEY, NATION.N_NAME
     2  -  READ INDEX COLUMN : NATION.N_NATIONKEY
           READ TABLE COLUMN : NATION.N_NAME
             MAX RANGE : NATION.N_NATIONKEY < 10


<<<  end print plan
```

上述示例中NATION_PK_INDEX以ascending生成时forward scan索引则不需要用于ORDER BY的额外的sorting

INDEX_DESC(table_name [,] [index_name [[,] index_name]])

指backward scan索引因此以ascending order生成索引时以descending order输出以descending order生成时以ascending order输出

语句规则与INDEX(table_name [,] [index_name[[,] index_name]]) hint相同

以下为使用INDEX_DESC(table_name) hint的示例

```
\EXPLAIN PLAN
  SELECT /*+ INDEX_DESC( nation ) */
```

```
         n_nationkey,
         n_name
    FROM nation
   WHERE n_nationkey < 10
ORDER BY n_nationkey DESC;


>>>  start print plan


< Execution Plan >
=================================================================
|  IDX  |  NODE DESCRIPTION                                | ROWS |
-----------------------------------------------------------------
|    0  |  SELECT STATEMENT                                |  10  |
|    1  |    QUERY BLOCK ("$QB_IDX_2")                     |  10  |
|    2  |      INDEX ACCESS ("NATION", "NATION_PK_INDEX")  |  10  |
=================================================================


     1  -  TARGET : NATION.N_NATIONKEY, NATION.N_NAME
     2  -  READ INDEX COLUMN : NATION.N_NATIONKEY
           READ TABLE COLUMN : NATION.N_NAME
             MAX RANGE : NATION.N_NATIONKEY < 10


<<<  end print plan
```

上述示例中以ascending生成NATION_PK_INDEX时backward scan索引则不需要用于ORDER BY的

额外的sorting

INDEX_COMBINE(table_name [,] [index_name [[,] index_name]])

描述INDEX_COMBINE(table_name [,] [index_name [[,] index_name]]) hint时optimizer指定对描述的table分离OR语句并各自执行index scan后整合结果此时决定各个index scan时在列出的index中选择cost最佳的index未列出index_name时在该table的所有index中选择cost最佳的index因此根据OR语句可选择不同的index

描述INDEX_COMBINE hint时table name应存在于<from clause>index name也应为该table中存在的index名称

执行INDEX_COMBINE hint时用于scan该表的条件中必须要有or语句如果没有or语句则optimizer忽略该hint

以下为使用INDEX_COMBINE(table_name) hint的示例

```
\EXPLAIN PLAN
SELECT /*+ INDEX_COMBINE( orders ) */
       o_orderstatus
  FROM orders
 WHERE o_orderkey = 1 OR o_custkey = 1;


>>>  start print plan


< Execution Plan >
================================================================================
```

```
|  IDX  |  NODE DESCRIPTION                                          | ROWS |
-------------------------------------------------------------------------
|    0  |  SELECT STATEMENT                                          |   7  |
|    1  |    QUERY BLOCK ("$QB_IDX_2")                               |   7  |
|    2  |      CONCAT (Compare RID)                                  |   7  |
|    3  |        INDEX ACCESS ("ORDERS", "ORDERS_PK_INDEX")          |   1  |
|    4  |        INDEX ACCESS ("ORDERS", "ORDERS_CUSTKEY_FK")        |   6  |
=========================================================================


     1  -  TARGET : ORDERS.O_ORDERSTATUS
     2  -  CONCAT COLUMN : ORDERS.$PHYSICAL_ROWID, ORDERS.O_ORDERSTATUS
     3  -  READ INDEX COLUMN : ORDERS.O_ORDERKEY
           READ TABLE COLUMN : ORDERS.O_ORDERSTATUS
             MIN RANGE : ORDERS.O_ORDERKEY = 1
             MAX RANGE : ORDERS.O_ORDERKEY = 1
           FETCH ONE ROW
     4  -  READ INDEX COLUMN : ORDERS.O_CUSTKEY
           READ TABLE COLUMN : ORDERS.O_ORDERSTATUS
             MIN RANGE : ORDERS.O_CUSTKEY = 1
             MAX RANGE : ORDERS.O_CUSTKEY = 1


<<<  end print plan
```

上述示例中无法用整个o_orderkey = 1 OR o_custkey = 1条件进行index scan因此性能会下降因此分离OR语句各自对 o_orderkey = 1和o_custkey = 1进行index scan后整合其结果会提升性能

IN_KEY_RANGE(table_name [,] [index_name [[,] index_name]])

描述IN_KEY_RANGE(table_name [,] [index_name [[,] index_name]]) hint时optimizer对描述的表执行IN key range scan此时在列出的index中选择cost最佳的index未列出index_name时在该表的所有index中选择cost最佳的index

但应用IN_KEY_RANGE(table_name [,] [index_name [[,] index_name]]) hint时需要要可以IN key range的filter

可以IN key range的filter的条件为如下

例: (col1, col2) IN ((val1, val2), (val3, val4))

- WHERE子句中应有IN 或 =ANY List Function Filter
- col1和col2应为base column即不可以是运算或function
- 对应col1的(val1, val3)应可以转换为一个数据类型

  对应col2的(val2, val4)应可以转换为一个数据类型

以下为使用IN_KEY_RANGE hint的示例

```
\EXPLAIN PLAN
  SELECT /*+ IN_KEY_RANGE( orders ) */
         o_orderstatus
    FROM orders
   WHERE o_orderkey > 500 AND o_custkey IN ( 1, 10, 100, 1000, 10000 );


< Execution Plan >
=============================================================================
|  IDX  |  NODE DESCRIPTION                                          | ROWS |
-----------------------------------------------------------------------------
```

```
|    0  |  SELECT STATEMENT                                    |   91 |
|    1  |    QUERY BLOCK ("$QB_IDX_2")                         |   91 |
|    2  |      INDEX ACCESS ("ORDERS", "ORDERS_CUSTKEY_FK")    |   91 |
=========================================================================


     1  -  TARGET : ORDERS.O_ORDERSTATUS
     2  -  READ INDEX COLUMN : ORDERS.O_CUSTKEY
           READ TABLE COLUMN : ORDERS.O_ORDERKEY, ORDERS.O_ORDERSTATUS
           IN KEY RANGE
             MIN RANGE : ORDERS.O_CUSTKEY = ?
             MAX RANGE : ORDERS.O_CUSTKEY = ?
             PHYSICAL TABLE FILTER : ORDERS.O_ORDERKEY > 500


<<<  end print plan
```

如上述execution plan所示可看到执行了IN key range scan

ROWID(table_name)

描述ROWID(table_name) hint时optimizer对描述的table执行rowid scan

描述ROWID hint时table name应存在于<from clause>

应用ROWID hint时该表中应有使用ROWID的equal条件没有这种条件时optimizer忽略此hint

以下为使用ROWID hint的示例

```
\EXPLAIN PLAN
```

```
  SELECT /*+ ROWID( orders ) */
         o_orderstatus
    FROM orders
   WHERE rowid = 'AAAAAAAAYfAAACAAACGjAAA';


>>>  start print plan


< Execution Plan >
============================================================================
|  IDX  |  NODE DESCRIPTION                                        | ROWS |
----------------------------------------------------------------------------
|    0  |  SELECT STATEMENT                                        |    1 |
|    1  |    QUERY BLOCK ("$QB_IDX_2")                             |    1 |
|    2  |      ROWID ACCESS ("ORDERS")                             |    1 |
============================================================================


     1  -  TARGET : ORDERS.O_ORDERSTATUS
     2  -  READ COLUMN : ORDERS.O_ORDERSTATUS
             ROWID FILTER : ORDERS.ROWID = 'AAAAAAAAYfAAACAAACGjAAA'


<<<  end print plan
```

<join hints>

<join order hints>

指定join ordering方法的hint

ORDERED

描述ORDERED hint时optimizer指定在join ordering时按照 <from clause>中描述的顺序进行join

用户准确了解满足join条件的row数并知道最佳join order时可使用ORDERED hint减少join ordering成本

以下为使用ORDERED hint的示例

```
\EXPLAIN PLAN
  SELECT  /*+ ORDERED */
          l_orderkey,
          ROUND( sum(l_extendedprice*(1-l_discount)), 2) as revenue,
          o_orderdate,
          o_shippriority
   FROM   customer,
          orders,
          lineitem
  WHERE   c_mktsegment = 'BUILDING'
    AND   c_custkey = o_custkey
    AND   l_orderkey = o_orderkey
    AND   o_orderdate < date '1995-03-15'
```

```
    AND   l_shipdate > date '1995-03-15'
GROUP BY  l_orderkey,
          o_orderdate,
          o_shippriority;

>>>  start print plan

< Execution Plan >
================================================================================
==
|  IDX  |  NODE DESCRIPTION                                          |   ROWS
|
--------------------------------------------------------------------------------
--
| 0 |  SELECT STATEMENT                                              |  11620
|
| 1 |    QUERY BLOCK ("$QB_IDX_2")                                   |  11620
|
| 2 |      GROUP HASH INSTANT                                        |  11620
|
| 3 |        NESTED JOIN (INNER JOIN)                                |  30519
|
| 4 |          NESTED JOIN (INNER JOIN)                              | 147126
|
| 5 |            TABLE ACCESS ("CUSTOMER")                           |  30142
```

```
|
| 6 |            INDEX ACCESS ("ORDERS", "ORDERS_CUSTKEY_FK")     | 147126
|
| 7 |          INDEX ACCESS ("LINEITEM", "LINEITEM_ORDERKEY_FK")  |  30519
|
================================================================================
==


     1  -  TARGET : LINEITEM.L_ORDERKEY,
ROUND(SUM( LINEITEM.L_EXTENDEDPRICE * ( 1 - LINEITEM.L_DISCOUNT ) ),2) AS
REVENUE, ORDERS.O_ORDERDATE, ORDERS.O_SHIPPRIORITY
     2  -  GROUP KEY : LINEITEM.L_ORDERKEY, ORDERS.O_ORDERDATE,
ORDERS.O_SHIPPRIORITY
           RECORD COLUMN : SUM( LINEITEM.L_EXTENDEDPRICE * ( 1 -
LINEITEM.L_DISCOUNT ) )
           READ KEY COLUMN : LINEITEM.L_ORDERKEY, ORDERS.O_ORDERDATE,
ORDERS.O_SHIPPRIORITY
           READ RECORD COLUMN : SUM( LINEITEM.L_EXTENDEDPRICE * ( 1 -
LINEITEM.L_DISCOUNT ) )
     3  -  JOINED COLUMN : LINEITEM.L_ORDERKEY, ORDERS.O_ORDERDATE,
ORDERS.O_SHIPPRIORITY, LINEITEM.L_EXTENDEDPRICE, LINEITEM.L_DISCOUNT
     4  -  JOINED COLUMN : ORDERS.O_ORDERKEY, ORDERS.O_ORDERDATE,
ORDERS.O_SHIPPRIORITY
     5  -  READ COLUMN : CUSTOMER.C_CUSTKEY, CUSTOMER.C_MKTSEGMENT
             PHYSICAL FILTER : CUSTOMER.C_MKTSEGMENT = 'BUILDING'
```

```
     6  -  READ INDEX COLUMN : ORDERS.O_CUSTKEY
           READ TABLE COLUMN : ORDERS.O_ORDERKEY, ORDERS.O_ORDERDATE,
ORDERS.O_SHIPPRIORITY
             MIN RANGE : ORDERS.O_CUSTKEY = {CUSTOMER.C_CUSTKEY}
             MAX RANGE : ORDERS.O_CUSTKEY = {CUSTOMER.C_CUSTKEY}
             PHYSICAL TABLE FILTER : ORDERS.O_ORDERDATE < DATE'1995-03-15'
     7  -  READ INDEX COLUMN : LINEITEM.L_ORDERKEY
           READ TABLE COLUMN : LINEITEM.L_EXTENDEDPRICE,
LINEITEM.L_DISCOUNT, LINEITEM.L_SHIPDATE
             MIN RANGE : LINEITEM.L_ORDERKEY = {ORDERS.O_ORDERKEY}
             MAX RANGE : LINEITEM.L_ORDERKEY = {ORDERS.O_ORDERKEY}
             PHYSICAL TABLE FILTER : LINEITEM.L_SHIPDATE > DATE'1995-03-
15'

<<<  end print plan
```

ORDERING(table_name [, table_name [, table_name [LEFT | RIGHT]]])

描述ORDERING(table_name [, table_name [, table_name [LEFT | RIGHT]]]) hint时optimizer指定join ordering时按照hint中描述的table顺序进行join

第一个和第二个table中无法描述位置指定选项从第三个table开始可描述位置指定选项位置指定选项有LEFT和RIGHTLEFT使该table位于join的left node（outer node）RIGHT使该table位于join的right node（inner node）不使用位置指定选项时由cost estimation决定位置

以下为使用ORDERING hint的示例

```
\EXPLAIN PLAN
  SELECT  /*+ ORDERING( customer, orders, lineitem RIGHT ) */
          l_orderkey,
          ROUND( sum(l_extendedprice*(1-l_discount)), 2) as revenue,
          o_orderdate,
          o_shippriority
   FROM   lineitem,
          orders,
          customer
  WHERE   c_mktsegment = 'BUILDING'
    AND   c_custkey = o_custkey
    AND   l_orderkey = o_orderkey
    AND   o_orderdate < date '1995-03-15'
    AND   l_shipdate > date '1995-03-15'
GROUP BY  l_orderkey,
          o_orderdate,
          o_shippriority;


>>>  start print plan


< Execution Plan >
================================================================================
==
|  IDX  |  NODE DESCRIPTION                                          |   ROWS
|
```

```
-------------------------------------------------------------------------
--
| 0 |  SELECT STATEMENT                                      |  11620
|
| 1 |    QUERY BLOCK ("$QB_IDX_2")                           |  11620
|
| 2 |      GROUP HASH INSTANT                                |  11620
|
| 3 |        NESTED JOIN (INNER JOIN)                        |  30519
|
| 4 |          NESTED JOIN (INNER JOIN)                      | 147126
|
| 5 |            TABLE ACCESS ("CUSTOMER")                   |  30142
|
| 6 |            INDEX ACCESS ("ORDERS", "ORDERS_CUSTKEY_FK")     | 147126
|
| 7 |          INDEX ACCESS ("LINEITEM", "LINEITEM_ORDERKEY_FK")  |  30519
|
=========================================================================
==


     1  -  TARGET : LINEITEM.L_ORDERKEY,
ROUND(SUM( LINEITEM.L_EXTENDEDPRICE * ( 1 - LINEITEM.L_DISCOUNT ) ),2) AS
REVENUE, ORDERS.O_ORDERDATE, ORDERS.O_SHIPPRIORITY
     2  -  GROUP KEY : LINEITEM.L_ORDERKEY, ORDERS.O_ORDERDATE,
```

```
ORDERS.O_SHIPPRIORITY
           RECORD COLUMN : SUM( LINEITEM.L_EXTENDEDPRICE * ( 1 -
LINEITEM.L_DISCOUNT ) )
           READ KEY COLUMN : LINEITEM.L_ORDERKEY, ORDERS.O_ORDERDATE,
ORDERS.O_SHIPPRIORITY
           READ RECORD COLUMN : SUM( LINEITEM.L_EXTENDEDPRICE * ( 1 -
LINEITEM.L_DISCOUNT ) )
     3  -  JOINED COLUMN : LINEITEM.L_ORDERKEY, ORDERS.O_ORDERDATE,
ORDERS.O_SHIPPRIORITY, LINEITEM.L_EXTENDEDPRICE, LINEITEM.L_DISCOUNT
     4  -  JOINED COLUMN : ORDERS.O_ORDERKEY, ORDERS.O_ORDERDATE,
ORDERS.O_SHIPPRIORITY
     5  -  READ COLUMN : CUSTOMER.C_CUSTKEY, CUSTOMER.C_MKTSEGMENT
             PHYSICAL FILTER : CUSTOMER.C_MKTSEGMENT = 'BUILDING'
     6  -  READ INDEX COLUMN : ORDERS.O_CUSTKEY
           READ TABLE COLUMN : ORDERS.O_ORDERKEY, ORDERS.O_ORDERDATE,
ORDERS.O_SHIPPRIORITY
             MIN RANGE : ORDERS.O_CUSTKEY = {CUSTOMER.C_CUSTKEY}
             MAX RANGE : ORDERS.O_CUSTKEY = {CUSTOMER.C_CUSTKEY}
             PHYSICAL TABLE FILTER : ORDERS.O_ORDERDATE < DATE'1995-03-15'
     7  -  READ INDEX COLUMN : LINEITEM.L_ORDERKEY
           READ TABLE COLUMN : LINEITEM.L_EXTENDEDPRICE,
LINEITEM.L_DISCOUNT, LINEITEM.L_SHIPDATE
             MIN RANGE : LINEITEM.L_ORDERKEY = {ORDERS.O_ORDERKEY}
             MAX RANGE : LINEITEM.L_ORDERKEY = {ORDERS.O_ORDERKEY}
             PHYSICAL TABLE FILTER : LINEITEM.L_SHIPDATE > DATE'1995-03-
```

```
15'


<<<  end print plan
```

LEADING(table_name [[,] table_name])

描述LEADING(table_name [[,] table_name]) hint时optimizer指定在join ordering时按照hint中描述的table顺序进行join与ORDERING hint不同无法指定位置仅可指定参与join ordering的表的顺序

以下为使用LEADING hint的示例

```
\EXPLAIN PLAN
  SELECT  /*+ LEADING( customer, orders, lineitem ) */
          l_orderkey,
          ROUND( sum(l_extendedprice*(1-l_discount)), 2) as revenue,
          o_orderdate,
          o_shippriority
   FROM   lineitem,
          orders,
          customer
  WHERE   c_mktsegment = 'BUILDING'
    AND   c_custkey = o_custkey
    AND   l_orderkey = o_orderkey
    AND   o_orderdate < date '1995-03-15'
    AND   l_shipdate > date '1995-03-15'
```

```
GROUP BY  l_orderkey,
          o_orderdate,
          o_shippriority;


>>>  start print plan


< Execution Plan >
================================================================================
==
|  IDX  |  NODE DESCRIPTION                                          |   ROWS
|
--------------------------------------------------------------------------------
--
| 0 |  SELECT STATEMENT                                              |  11620
|
| 1 |    QUERY BLOCK ("$QB_IDX_2")                                   |  11620
|
| 2 |      GROUP HASH INSTANT                                        |  11620
|
| 3 |        NESTED JOIN (INNER JOIN)                                |  30519
|
| 4 |          NESTED JOIN (INNER JOIN)                              | 147126
|
| 5 |            TABLE ACCESS ("CUSTOMER")                           |  30142
|
```

```
| 6 |            INDEX ACCESS ("ORDERS", "ORDERS_CUSTKEY_FK")     | 147126
|
| 7 |          INDEX ACCESS ("LINEITEM", "LINEITEM_ORDERKEY_FK")  |  30519
|
================================================================================
==


 1  -  TARGET : LINEITEM.L_ORDERKEY, ROUND(SUM( LINEITEM.L_EXTENDEDPRICE *
( 1 - LINEITEM.L_DISCOUNT ) ),2) AS REVENUE, ORDERS.O_ORDERDATE,
ORDERS.O_SHIPPRIORITY
     2  -  GROUP KEY : LINEITEM.L_ORDERKEY, ORDERS.O_ORDERDATE,
ORDERS.O_SHIPPRIORITY
           RECORD COLUMN : SUM( LINEITEM.L_EXTENDEDPRICE * ( 1 -
LINEITEM.L_DISCOUNT ) )
           READ KEY COLUMN : LINEITEM.L_ORDERKEY, ORDERS.O_ORDERDATE,
ORDERS.O_SHIPPRIORITY
           READ RECORD COLUMN : SUM( LINEITEM.L_EXTENDEDPRICE * ( 1 -
LINEITEM.L_DISCOUNT ) )
     3  -  JOINED COLUMN : LINEITEM.L_ORDERKEY, ORDERS.O_ORDERDATE,
ORDERS.O_SHIPPRIORITY, LINEITEM.L_EXTENDEDPRICE, LINEITEM.L_DISCOUNT
     4  -  JOINED COLUMN : ORDERS.O_ORDERKEY, ORDERS.O_ORDERDATE,
ORDERS.O_SHIPPRIORITY
     5  -  READ COLUMN : CUSTOMER.C_CUSTKEY, CUSTOMER.C_MKTSEGMENT
             PHYSICAL FILTER : CUSTOMER.C_MKTSEGMENT = 'BUILDING'
     6  -  READ INDEX COLUMN : ORDERS.O_CUSTKEY
```

```
           READ TABLE COLUMN : ORDERS.O_ORDERKEY, ORDERS.O_ORDERDATE,
ORDERS.O_SHIPPRIORITY
             MIN RANGE : ORDERS.O_CUSTKEY = {CUSTOMER.C_CUSTKEY}
             MAX RANGE : ORDERS.O_CUSTKEY = {CUSTOMER.C_CUSTKEY}
             PHYSICAL TABLE FILTER : ORDERS.O_ORDERDATE < DATE'1995-03-15'
     7   -   READ INDEX COLUMN : LINEITEM.L_ORDERKEY
           READ TABLE COLUMN : LINEITEM.L_EXTENDEDPRICE,
LINEITEM.L_DISCOUNT, LINEITEM.L_SHIPDATE
             MIN RANGE : LINEITEM.L_ORDERKEY = {ORDERS.O_ORDERKEY}
             MAX RANGE : LINEITEM.L_ORDERKEY = {ORDERS.O_ORDERKEY}
             PHYSICAL TABLE FILTER : LINEITEM.L_SHIPDATE > DATE'1995-03-
15'


<<<  end print plan
```

<join operation hints>

USE_HASH(table_name [[,] table_name])

描述USE_HASH(table_name [[,] table_name]) hint时optimizer决定table_name参与的join的join operation时选择hash join

需要描述一个以上的table并且无法描述两个以上的相同表描述的table中已有<join operation hints>时忽略此hint而且参与join的两个表中描述了各不相同的join operation hint时优先应用在right node(inner node)描述的hint

即使描述了USE_HASH hint也要有equi-join condition才能应用hint没有可以hash join的join

condition时optimizer通过cost estimation选择最佳join operation

以下为使用USE_HASH(table_name [[,] table_name]) hint的示例

```
\EXPLAIN PLAN
  SELECT  /*+ USE_HASH( orders ) */
          c_name,
          o_orderdate,
          o_orderstatus
    FROM  customer,
          orders
   WHERE  c_custkey = o_custkey
     AND  o_orderdate >= date '1995-03-15'
     AND  o_orderdate < date '1995-03-15' + interval '1' month;


>>>  start print plan


< Execution Plan >
================================================================================
==
|  IDX  |  NODE DESCRIPTION                                          |   ROWS
|
--------------------------------------------------------------------------------
--
|    0  |  SELECT STATEMENT                                          |  19343
|
```

```
|    1  |    QUERY BLOCK ("$QB_IDX_2")                              |  19343
|
|    2  |      HASH JOIN (INNER JOIN)                               |  19343
|
|    3  |        TABLE ACCESS ("CUSTOMER")                          | 150000
|
|    4  |        HASH JOIN INSTANT                                  |  19343
|
|    5  |          TABLE ACCESS ("ORDERS")                          |  19343
|
=============================================================================
==


     1  -  TARGET : CUSTOMER.C_NAME, ORDERS.O_ORDERDATE,
ORDERS.O_ORDERSTATUS
     2  -  JOINED COLUMN : CUSTOMER.C_NAME, ORDERS.O_ORDERDATE,
ORDERS.O_ORDERSTATUS
     3  -  READ COLUMN : CUSTOMER.C_CUSTKEY, CUSTOMER.C_NAME
     4  -  HASH KEY : ORDERS.O_CUSTKEY
           RECORD COLUMN : ORDERS.O_ORDERDATE, ORDERS.O_ORDERSTATUS
           READ KEY COLUMN : ORDERS.O_CUSTKEY, ORDERS.O_ORDERDATE,
ORDERS.O_ORDERSTATUS
             HASH FILTER : ORDERS.O_CUSTKEY = CUSTOMER.C_CUSTKEY
     5  -  READ COLUMN : ORDERS.O_CUSTKEY, ORDERS.O_ORDERSTATUS,
ORDERS.O_ORDERDATE
```

```
            PHYSICAL FILTER : ORDERS.O_ORDERDATE < DATE'1995-03-15' +
CAST( '1' AS INTERVAL(MONTH) ) AND ORDERS.O_ORDERDATE >= DATE'1995-03-15'


<<<  end print plan
```

USE_HASH(table_name [[,] table_name] , hash_bucket_count)

USE_HASH(table_name [[,] table_name] , hash_bucket_count) hint与USE_HASH hint相同区别在于可指定hash bucket count

统计信息不准确时由于为了hash join生成的hash instant的hash bucket count过多或过少而会降低性能此时可通过hint指定 hash bucket count防止性能下降

以下为使用USE_HASH(table_name [[,] table_name], hash_bucket_count) hint的示例

```
SELECT COUNT(DISTINCT o_custkey)
  FROM orders
 WHERE o_orderdate >= date '1995-03-15'
   AND o_orderdate < date '1995-03-15' + interval '1' month;


COUNT(DISTINCT O_CUSTKEY)
-------------------------
                    17430


1 row selected.


\EXPLAIN PLAN VERBOSE
```

```
  SELECT  /*+ USE_HASH( orders, 17430 ) */
          c_name,
          o_orderdate,
          o_orderstatus
    FROM  customer,
          orders
   WHERE  c_custkey = o_custkey
     AND  o_orderdate >= date '1995-03-15'
     AND  o_orderdate < date '1995-03-15' + interval '1' month;



>>>  start print plan


< Execution Plan >
================================================================================
=
|  IDX  |  NODE DESCRIPTION                                |  ROWS | 省略
|
--------------------------------------------------------------------------------
-
|  0  |  SELECT STATEMENT                                  |  19343 | ...
|
|  1  |    QUERY BLOCK ("$QB_IDX_2")                       |  19343 | ...
|
|  2  |      HASH JOIN (INNER JOIN)                        |  19343 | ...
```

```
|
|  3  |        TABLE ACCESS ("CUSTOMER")                  | 150000 | ...
|
|  4  |        HASH JOIN INSTANT                          |  19343 | ...
|
|  5  |          TABLE ACCESS ("ORDERS")                  |  19343 | ...
|
================================================================================
=


     1  -  TARGET : CUSTOMER.C_NAME, ORDERS.O_ORDERDATE,
ORDERS.O_ORDERSTATUS
     2  -  JOINED COLUMN : CUSTOMER.C_NAME, ORDERS.O_ORDERDATE,
ORDERS.O_ORDERSTATUS
     3  -  READ COLUMN : CUSTOMER.C_CUSTKEY, CUSTOMER.C_NAME
     4  -  HASH KEY : ORDERS.O_CUSTKEY
           RECORD COLUMN : ORDERS.O_ORDERDATE, ORDERS.O_ORDERSTATUS
           READ KEY COLUMN : ORDERS.O_CUSTKEY, ORDERS.O_ORDERDATE,
ORDERS.O_ORDERSTATUS
             HASH FILTER : ORDERS.O_CUSTKEY = CUSTOMER.C_CUSTKEY
           HASH BUCKET COUNT : 17430
     5  -  READ COLUMN : ORDERS.O_CUSTKEY, ORDERS.O_ORDERSTATUS,
ORDERS.O_ORDERDATE
             PHYSICAL FILTER : ORDERS.O_ORDERDATE < DATE'1995-03-15' +
CAST( '1' AS INTERVAL(MONTH) ) AND ORDERS.O_ORDERDATE >= DATE'1995-03-15'
```

```
<<<  end print plan
```

上述示例中参与join的orders的output中指定了与distinct value数量相同的hash bucket count

USE_HASH_IN(alias)

描述USE_HASH_IN(alias) hint时optimizer在决定alias参与的join的join operation时选择hash join并将alias放在join的right(inner)

描述的alias中已有<join operation hints>时忽略此hint而且参与join的两张table中描述了各不相同的join operation时优先应用描述在right node(inner node)的hint

Alias可以是table name, table alias name, view name, view alias name, join alias name

以下为使用USE_HASH_IN(alias) hint的示例

```
--# hash join
\EXPLAIN PLAN
  SELECT  /*+ USE_HASH_IN( orders ) */
          c_name,
          o_orderdate,
          o_orderstatus
    FROM  customer,
          orders
   WHERE  c_custkey = o_custkey
     AND  o_orderdate >= date '1995-03-15'
     AND  o_orderdate < date '1995-03-15' + interval '1' month;
```

```
>>>  start print plan


< Execution Plan >

=============================================================================
==
|  IDX  |  NODE DESCRIPTION                                      |   ROWS
|
-----------------------------------------------------------------------------
--
|    0  |  SELECT STATEMENT                                      |  19343
|
|    1  |    QUERY BLOCK ("$QB_IDX_2")                           |  19343
|
|    2  |      HASH JOIN (INNER JOIN)                            |  19343
|
|    3  |        TABLE ACCESS ("CUSTOMER")                       | 150000
|
|    4  |        HASH JOIN INSTANT                               |  19343
|
|    5  |          TABLE ACCESS ("ORDERS")                       |  19343
|
=============================================================================
==
```

```
     1  -  TARGET : CUSTOMER.C_NAME, ORDERS.O_ORDERDATE,
ORDERS.O_ORDERSTATUS
     2  -  JOINED COLUMN : CUSTOMER.C_NAME, ORDERS.O_ORDERDATE,
ORDERS.O_ORDERSTATUS
     3  -  READ COLUMN : CUSTOMER.C_CUSTKEY, CUSTOMER.C_NAME
     4  -  HASH KEY : ORDERS.O_CUSTKEY
           RECORD COLUMN : ORDERS.O_ORDERDATE, ORDERS.O_ORDERSTATUS
           READ KEY COLUMN : ORDERS.O_CUSTKEY, ORDERS.O_ORDERDATE,
ORDERS.O_ORDERSTATUS
             HASH FILTER : ORDERS.O_CUSTKEY = CUSTOMER.C_CUSTKEY
     5  -  READ COLUMN : ORDERS.O_CUSTKEY, ORDERS.O_ORDERSTATUS,
ORDERS.O_ORDERDATE
             PHYSICAL FILTER : ORDERS.O_ORDERDATE < DATE'1995-03-15' +
CAST( '1' AS INTERVAL(MONTH) ) AND ORDERS.O_ORDERDATE >= DATE'1995-03-15'


<<<  end print plan
```

以下是使用join alias使用USE_HASH_IN(alias) hint的示例

```
\EXPLAIN PLAN
  SELECT  /*+ USE_HASH_IN( j1 )  */
          l_orderkey,
          ROUND( sum(l_extendedprice*(1-l_discount)), 2) as revenue,
          o_orderdate,
          o_shippriority
   FROM   ( customer INNER JOIN orders ON  c_mktsegment = 'BUILDING'
```

```
                                    AND c_custkey = o_custkey
                                    AND o_orderdate < date '1995-03-15'
          ) ALIAS j1
          INNER JOIN lineitem ON  l_orderkey = o_orderkey
                              AND l_shipdate > date '1995-03-15'
GROUP BY  l_orderkey,
          o_orderdate,
          o_shippriority;


>>>  start print plan


< Execution Plan >
=============================================================================
=
| IDX |  NODE DESCRIPTION                                       |    ROWS
|
-----------------------------------------------------------------------------
-
| 0 |  SELECT STATEMENT                                         |   11620
|
| 1 |    QUERY BLOCK ("$QB_IDX_2")                              |   11620
|
| 2 |      GROUP HASH INSTANT                                   |   11620
|
| 3 |        HASH JOIN (INNER JOIN)                             |   30519
```

```
|
| 4 |          TABLE ACCESS ("LINEITEM")                          | 3241776
|
| 5 |          HASH JOIN INSTANT                                  |   30519
|
| 6 |            NESTED JOIN (INNER JOIN)                         |  147126
|
| 7 |              TABLE ACCESS ("CUSTOMER")                      |   30142
|
| 8 |              INDEX ACCESS ("ORDERS", "ORDERS_CUSTKEY_FK") |  147126
|
=============================================================================
=


     1  -  TARGET : LINEITEM.L_ORDERKEY,
ROUND(SUM( LINEITEM.L_EXTENDEDPRICE * ( 1 - LINEITEM.L_DISCOUNT ) ),2) AS
REVENUE, ORDERS.O_ORDERDATE, ORDERS.O_SHIPPRIORITY
     2  -  GROUP KEY : LINEITEM.L_ORDERKEY, ORDERS.O_ORDERDATE,
ORDERS.O_SHIPPRIORITY
           RECORD COLUMN : SUM( LINEITEM.L_EXTENDEDPRICE * ( 1 -
LINEITEM.L_DISCOUNT ) )
           READ KEY COLUMN : LINEITEM.L_ORDERKEY, ORDERS.O_ORDERDATE,
ORDERS.O_SHIPPRIORITY
           READ RECORD COLUMN : SUM( LINEITEM.L_EXTENDEDPRICE * ( 1 -
LINEITEM.L_DISCOUNT ) )
```

```
     3  -  JOINED COLUMN : LINEITEM.L_ORDERKEY, ORDERS.O_ORDERDATE,
ORDERS.O_SHIPPRIORITY, LINEITEM.L_EXTENDEDPRICE, LINEITEM.L_DISCOUNT
     4  -  READ COLUMN : LINEITEM.L_ORDERKEY, LINEITEM.L_EXTENDEDPRICE,
LINEITEM.L_DISCOUNT, LINEITEM.L_SHIPDATE
              PHYSICAL FILTER : LINEITEM.L_SHIPDATE > DATE'1995-03-15'
     5  -  HASH KEY : ORDERS.O_ORDERKEY
            RECORD COLUMN : ORDERS.O_ORDERDATE, ORDERS.O_SHIPPRIORITY
            READ KEY COLUMN : ORDERS.O_ORDERKEY, ORDERS.O_ORDERDATE,
ORDERS.O_SHIPPRIORITY
              HASH FILTER : ORDERS.O_ORDERKEY = LINEITEM.L_ORDERKEY
     6  -  JOINED COLUMN : ORDERS.O_ORDERKEY, ORDERS.O_ORDERDATE,
ORDERS.O_SHIPPRIORITY
     7  -  READ COLUMN : CUSTOMER.C_CUSTKEY, CUSTOMER.C_MKTSEGMENT
              PHYSICAL FILTER : CUSTOMER.C_MKTSEGMENT = 'BUILDING'
     8  -  READ INDEX COLUMN : ORDERS.O_CUSTKEY
            READ TABLE COLUMN : ORDERS.O_ORDERKEY, ORDERS.O_ORDERDATE,
ORDERS.O_SHIPPRIORITY
              MIN RANGE : ORDERS.O_CUSTKEY = {CUSTOMER.C_CUSTKEY}
              MAX RANGE : ORDERS.O_CUSTKEY = {CUSTOMER.C_CUSTKEY}
              PHYSICAL TABLE FILTER : ORDERS.O_ORDERDATE < DATE'1995-03-15'


<<<  end print plan
```

如上述execution plan所示join alias j1位于hash join的right

USE_HASH_OUT(alias)

描述USE_HASH_OUT(alias) hint时optimizer在决定alias参与的join的join operation时选择hash join并将alias放在join的left(outer)

描述的alias中已有<join operation hints>时忽略此hint而且参与join的两张table中描述了各不相同的join operation时优先应用描述在right node(inner node)的hint

Alias可以是table name, table alias name, view name, view alias name, join alias name

以下为使用USE_HASH_OUT(alias) hint的示例

```
\EXPLAIN PLAN
  SELECT  /*+ USE_HASH_OUT( orders ) */
          c_name,
          o_orderdate,
          o_orderstatus
    FROM  customer,
          orders
   WHERE  c_custkey = o_custkey
     AND  o_orderdate >= date '1995-03-15'
     AND  o_orderdate < date '1995-03-15' + interval '1' month;

>>>  start print plan

< Execution Plan >
================================================================================
```

```
==
|  IDX  |  NODE DESCRIPTION                                      |   ROWS
|
-------------------------------------------------------------------------
--
|    0  |  SELECT STATEMENT                                      |  19343
|
|    1  |    QUERY BLOCK ("$QB_IDX_2")                           |  19343
|
|    2  |      HASH JOIN (INNER JOIN)                            |  19343
|
|    3  |        TABLE ACCESS ("ORDERS")                         |  19343
|
|    4  |        HASH JOIN INSTANT                               |  19343
|
|    5  |          TABLE ACCESS ("CUSTOMER")                     | 150000
|
=========================================================================
==


     1  -  TARGET : CUSTOMER.C_NAME, ORDERS.O_ORDERDATE,
ORDERS.O_ORDERSTATUS
     2  -  JOINED COLUMN : CUSTOMER.C_NAME, ORDERS.O_ORDERDATE,
ORDERS.O_ORDERSTATUS
     3  -  READ COLUMN : ORDERS.O_CUSTKEY, ORDERS.O_ORDERSTATUS,
```

```
ORDERS.O_ORDERDATE
             PHYSICAL FILTER : ORDERS.O_ORDERDATE < DATE'1995-03-15' +
CAST( '1' AS INTERVAL(MONTH) ) AND ORDERS.O_ORDERDATE >= DATE'1995-03-15'
     4  -  HASH KEY : CUSTOMER.C_CUSTKEY
           RECORD COLUMN : CUSTOMER.C_NAME
           READ KEY COLUMN : CUSTOMER.C_CUSTKEY, CUSTOMER.C_NAME
             HASH FILTER : CUSTOMER.C_CUSTKEY = ORDERS.O_CUSTKEY
           FETCH ONE ROW
     5  -  READ COLUMN : CUSTOMER.C_CUSTKEY, CUSTOMER.C_NAME


<<<  end print plan
```

NO_USE_HASH(table_name [[,] table_name])

描述NO_USE_HASH(table_name [[,] table_name]) hint时optimizer在决定table_name参与的join的join operation时排除hash join因此在除hash join外的其他join operation中选择最佳cost estimation

应描述一个以上的table并且无法描述两个以上的相同的table描述的表中已有<join operation hints>时忽略此hint而且参与join的两张表中描述了各不相同的join operation hint时优先应用在right node(inner node)描述的hint

以下为使用NO_USE_HASH(table_name [[,] table_name]) hint的示例

```
\EXPLAIN PLAN
  SELECT  /*+ NO_USE_HASH( orders ) */
          c_name,
```

```
          o_orderdate,
          o_orderstatus
    FROM  customer,
          orders
   WHERE  c_custkey = o_custkey
     AND  o_orderdate >= date '1995-03-15'
     AND  o_orderdate < date '1995-03-15' + interval '1' month;


>>>  start print plan


< Execution Plan >
================================================================================
=
| IDX |  NODE DESCRIPTION                                          |  ROWS
|
--------------------------------------------------------------------------------
-
|  0  |  SELECT STATEMENT                                          | 19343
|
|  1  |    QUERY BLOCK ("$QB_IDX_2")                               | 19343
|
|  2  |      NESTED JOIN (INNER JOIN)                              | 19343
|
|  3  |        TABLE ACCESS ("ORDERS")                             | 19343
|
```

```
|  4  |        INDEX ACCESS ("CUSTOMER", "CUSTOMER_PK_INDEX")     | 19343
|
=============================================================================
=


     1  -  TARGET : CUSTOMER.C_NAME, ORDERS.O_ORDERDATE,
ORDERS.O_ORDERSTATUS
     2  -  JOINED COLUMN : CUSTOMER.C_NAME, ORDERS.O_ORDERDATE,
ORDERS.O_ORDERSTATUS
     3  -  READ COLUMN : ORDERS.O_CUSTKEY, ORDERS.O_ORDERSTATUS,
ORDERS.O_ORDERDATE
              PHYSICAL FILTER : ORDERS.O_ORDERDATE < DATE'1995-03-15' +
CAST( '1' AS INTERVAL(MONTH) ) AND ORDERS.O_ORDERDATE >= DATE'1995-03-15'
     4  -  READ INDEX COLUMN : CUSTOMER.C_CUSTKEY
            READ TABLE COLUMN : CUSTOMER.C_NAME
              MIN RANGE : CUSTOMER.C_CUSTKEY = {ORDERS.O_CUSTKEY}
              MAX RANGE : CUSTOMER.C_CUSTKEY = {ORDERS.O_CUSTKEY}
            FETCH ONE ROW


<<<  end print plan
```

USE_MERGE(table_name [[,] table_name])

描述USE_MERGE(table_name [[,] table_name]) hint时optimizer决定table_name参与的join的join operation时选择merge join

应描述一个以上的table并且无法描述两个以上的相同的table描述的表中已有<join operation hints>时忽略此hint而且参与join的两张表中描述了各不相同的join operation hint时优先应用在right node(inner node)描述的hint

以下为使用USE_MERGE(table_name [[,] table_name]) hint的示例

```
\EXPLAIN PLAN
  SELECT  /*+ USE_MERGE( orders ) */
          c_name,
          o_orderdate,
          o_orderstatus
    FROM  customer,
          orders
   WHERE  c_custkey = o_custkey
     AND  o_orderdate >= date '1995-03-15'
     AND  o_orderdate < date '1995-03-15' + interval '1' month;


>>>  start print plan


< Execution Plan >
================================================================================
==
|  IDX  |  NODE DESCRIPTION                                          |   ROWS
|
--------------------------------------------------------------------------------
--
```

```
|  0  |  SELECT STATEMENT                                          |  19343
|
|  1  |    QUERY BLOCK ("$QB_IDX_2")                               |  19343
|
|  2  |      MERGE JOIN (INNER JOIN)                               |  19343
|
|  3  |        INDEX ACCESS ("CUSTOMER", "CUSTOMER_PK_INDEX")      | 150000
|
|  4  |        SORT JOIN INSTANT                                   |  19343
|
|  5  |          TABLE ACCESS ("ORDERS")                           |  19343
|
================================================================================
==


     1  -  TARGET : CUSTOMER.C_NAME, ORDERS.O_ORDERDATE,
ORDERS.O_ORDERSTATUS
     2  -  JOINED COLUMN : CUSTOMER.C_NAME, ORDERS.O_ORDERDATE,
ORDERS.O_ORDERSTATUS
             ON FILTER (Equi) : CUSTOMER.C_CUSTKEY = ORDERS.O_CUSTKEY
     3  -  READ INDEX COLUMN : CUSTOMER.C_CUSTKEY
           READ TABLE COLUMN : CUSTOMER.C_NAME
     4  -  SORT KEY : "ORDERS.O_CUSTKEY ASC NULLS LAST"
           RECORD COLUMN : ORDERS.O_ORDERDATE, ORDERS.O_ORDERSTATUS
           READ KEY COLUMN : ORDERS.O_CUSTKEY
```

```
            READ RECORD COLUMN : ORDERS.O_ORDERDATE, ORDERS.O_ORDERSTATUS
              MIN RANGE : ORDERS.O_CUSTKEY >= {CUSTOMER.C_CUSTKEY}
              MAX RANGE : ORDERS.O_CUSTKEY IS NOT NULL
     5   -  READ COLUMN : ORDERS.O_CUSTKEY, ORDERS.O_ORDERSTATUS,
ORDERS.O_ORDERDATE
              PHYSICAL FILTER : ORDERS.O_ORDERDATE < DATE'1995-03-15' +
CAST( '1' AS INTERVAL(MONTH) ) AND ORDERS.O_ORDERDATE >= DATE'1995-03-15'


<<<  end print plan
```

USE_MERGE_IN(alias)

描述USE_MERGE_IN(alias) hint时optimizer决定alias参与的join的join operation时选择merge join并且将alias放在join的right(inner)

描述的alias中已有<join operation hints>时忽略此hint而且参与join的两张表中描述了各不相同的join operation hint时优先适用在right node(inner node)描述的hint

Alias可以为table name, table alias name, view name, view alias name, join alias name

以下为使用USE_MERGE_IN(alias) hint的示例

```
\EXPLAIN PLAN
  SELECT  /*+ USE_MERGE_IN( orders ) */
          c_name,
          o_orderdate,
          o_orderstatus
```

```
    FROM  customer,
          orders
   WHERE  c_custkey = o_custkey
     AND  o_orderdate >= date '1995-03-15'
     AND  o_orderdate < date '1995-03-15' + interval '1' month;


< Execution Plan >
================================================================================
=
|  IDX  |  NODE DESCRIPTION                                          |   ROWS
|
--------------------------------------------------------------------------------
-
|  0  |  SELECT STATEMENT                                            |  19343
|
|  1  |    QUERY BLOCK ("$QB_IDX_2")                                 |  19343
|
|  2  |      MERGE JOIN (INNER JOIN)                                 |  19343
|
|  3  |        INDEX ACCESS ("CUSTOMER", "CUSTOMER_PK_INDEX")        | 150000
|
|  4  |        SORT JOIN INSTANT                                     |  19343
|
|  5  |          TABLE ACCESS ("ORDERS")                             |  19343
|
```

```
================================================================================
=


     1  -  TARGET : CUSTOMER.C_NAME, ORDERS.O_ORDERDATE,
ORDERS.O_ORDERSTATUS
     2  -  JOINED COLUMN : CUSTOMER.C_NAME, ORDERS.O_ORDERDATE,
ORDERS.O_ORDERSTATUS
              ON FILTER (Equi) : CUSTOMER.C_CUSTKEY = ORDERS.O_CUSTKEY
     3  -  READ INDEX COLUMN : CUSTOMER.C_CUSTKEY
            READ TABLE COLUMN : CUSTOMER.C_NAME
     4  -  SORT KEY : "ORDERS.O_CUSTKEY ASC NULLS LAST"
            RECORD COLUMN : ORDERS.O_ORDERDATE, ORDERS.O_ORDERSTATUS
            READ KEY COLUMN : ORDERS.O_CUSTKEY
            READ RECORD COLUMN : ORDERS.O_ORDERDATE, ORDERS.O_ORDERSTATUS
              MIN RANGE : ORDERS.O_CUSTKEY >= {CUSTOMER.C_CUSTKEY}
              MAX RANGE : ORDERS.O_CUSTKEY IS NOT NULL
     5  -  READ COLUMN : ORDERS.O_CUSTKEY, ORDERS.O_ORDERSTATUS,
ORDERS.O_ORDERDATE
              PHYSICAL FILTER : ORDERS.O_ORDERDATE < DATE'1995-03-15' +
CAST( '1' AS INTERVAL(MONTH) ) AND ORDERS.O_ORDERDATE >= DATE'1995-03-15'


<<<  end print plan
```

USE_MERGE_OUT(alias)

描述USE_MERGE_OUT(alias) hint时optimizer决定alias参与的join的join operation时选择merge

join并且将alias放在join的left(outer)

描述的alias中已有<join operation hints>时忽略此hint而且参与join的两张表中描述了各不相同的join operation hint时优先应用在right node(inner node)描述的hint

以下为使用USE_MERGE_OUT(alias) hint的示例

```
\EXPLAIN PLAN
  SELECT  /*+ USE_MERGE_OUT( orders ) */
          c_name,
          o_orderdate,
          o_orderstatus
    FROM  customer,
          orders
   WHERE  c_custkey = o_custkey
     AND  o_orderdate >= date '1995-03-15'
     AND  o_orderdate < date '1995-03-15' + interval '1' month;



< Execution Plan >
================================================================================
==
|  IDX  |  NODE DESCRIPTION                                          |   ROWS
|
--------------------------------------------------------------------------------
--
```

```
|  0  |  SELECT STATEMENT                                          |  19343
|
|  1  |    QUERY BLOCK ("$QB_IDX_2")                               |  19343
|
|  2  |      MERGE JOIN (INNER JOIN)                               |  19343
|
|  3  |        SORT JOIN INSTANT                                   |  19343
|
|  4  |          TABLE ACCESS ("ORDERS")                           |  19343
|
|  5  |        INDEX ACCESS ("CUSTOMER", "CUSTOMER_PK_INDEX")      | 149991
|
================================================================================
==


     1  -  TARGET : CUSTOMER.C_NAME, ORDERS.O_ORDERDATE,
ORDERS.O_ORDERSTATUS
     2  -  JOINED COLUMN : CUSTOMER.C_NAME, ORDERS.O_ORDERDATE,
ORDERS.O_ORDERSTATUS
             ON FILTER (Equi) : ORDERS.O_CUSTKEY = CUSTOMER.C_CUSTKEY
     3  -  SORT KEY : "ORDERS.O_CUSTKEY ASC NULLS LAST"
           RECORD COLUMN : ORDERS.O_ORDERDATE, ORDERS.O_ORDERSTATUS
           READ KEY COLUMN : ORDERS.O_CUSTKEY
           READ RECORD COLUMN : ORDERS.O_ORDERDATE, ORDERS.O_ORDERSTATUS
     4  -  READ COLUMN : ORDERS.O_CUSTKEY, ORDERS.O_ORDERSTATUS,
```

```
ORDERS.O_ORDERDATE
             PHYSICAL FILTER : ORDERS.O_ORDERDATE < DATE'1995-03-15' +
CAST( '1' AS INTERVAL(MONTH) ) AND ORDERS.O_ORDERDATE >= DATE'1995-03-15'
     5  -  READ INDEX COLUMN : CUSTOMER.C_CUSTKEY
           READ TABLE COLUMN : CUSTOMER.C_NAME
             MIN RANGE : CUSTOMER.C_CUSTKEY >= {ORDERS.O_CUSTKEY}
             MAX RANGE : CUSTOMER.C_CUSTKEY IS NOT NULL

<<<  end print plan
```

NO_USE_MERGE(table_name [[,] table_name])

描述NO_USE_MERGE(table_name [[,] table_name]) hint时optimizer在决定table_name参与的join的join operation时排除merge join因此在除merge join外的其他join operation中选择最佳cost estimation

应描述一个以上的table并且无法描述两个以上的相同的table描述的表中已有<join operation hints>时忽略此hint而且参与join的两张表中描述了各不相同的join operation hint时优先应用在right node(inner node)描述的hint

以下为使用NO_USE_MERGE(table_name [[,] table_name]) hint的示例

```
\EXPLAIN PLAN
  SELECT  /*+ NO_USE_MERGE( orders ) */
          c_name,
          o_orderdate,
          o_orderstatus
```

```
    FROM  customer,
          orders
   WHERE  c_custkey = o_custkey
     AND  o_orderdate >= date '1995-03-15'
     AND  o_orderdate < date '1995-03-15' + interval '1' month;


< Execution Plan >
================================================================================
=
|  IDX  |  NODE DESCRIPTION                                          | ROWS
|
--------------------------------------------------------------------------------
-
|  0  |  SELECT STATEMENT                                            | 19343
|
|  1  |    QUERY BLOCK ("$QB_IDX_2")                                 | 19343
|
|  2  |      NESTED JOIN (INNER JOIN)                                | 19343
|
|  3  |        TABLE ACCESS ("ORDERS")                               | 19343
|
|  4  |        INDEX ACCESS ("CUSTOMER", "CUSTOMER_PK_INDEX")        | 19343
|
================================================================================
=
```

```
     1  -  TARGET : CUSTOMER.C_NAME, ORDERS.O_ORDERDATE,
ORDERS.O_ORDERSTATUS
     2  -  JOINED COLUMN : CUSTOMER.C_NAME, ORDERS.O_ORDERDATE,
ORDERS.O_ORDERSTATUS
     3  -  READ COLUMN : ORDERS.O_CUSTKEY, ORDERS.O_ORDERSTATUS,
ORDERS.O_ORDERDATE
              PHYSICAL FILTER : ORDERS.O_ORDERDATE < DATE'1995-03-15' +
CAST( '1' AS INTERVAL(MONTH) ) AND ORDERS.O_ORDERDATE >= DATE'1995-03-15'
     4  -  READ INDEX COLUMN : CUSTOMER.C_CUSTKEY
            READ TABLE COLUMN : CUSTOMER.C_NAME
              MIN RANGE : CUSTOMER.C_CUSTKEY = {ORDERS.O_CUSTKEY}
              MAX RANGE : CUSTOMER.C_CUSTKEY = {ORDERS.O_CUSTKEY}
            FETCH ONE ROW

<<<  end print plan
```

USE_NL(table_name [[,] table_name])

描述USE_NL(table_name [[,] table_name]) hint时optimizer决定table_name参与的join的join operation时选择nested loop join

应描述一个以上的table并且无法描述两个以上的相同的table描述的表中已有<join operation hints>时忽略此hint而且参与join的两张表中描述了各不相同的join operation hint时优先应用在right node(inner node)描述的hint

以下为使用USE_NL(table_name [[,] table_name]) hint的示例

```
\EXPLAIN PLAN
  SELECT  /*+ USE_NL( orders ) */
          c_name,
          o_orderdate,
          o_orderstatus
    FROM  customer,
          orders
   WHERE  c_custkey = o_custkey
     AND  o_orderdate >= date '1995-03-15'
     AND  o_orderdate < date '1995-03-15' + interval '1' month;


< Execution Plan >
================================================================================
=
|  IDX  |  NODE DESCRIPTION                                          | ROWS
|
--------------------------------------------------------------------------------
-
|  0  |  SELECT STATEMENT                                            | 19343
|
|  1  |    QUERY BLOCK ("$QB_IDX_2")                                 | 19343
|
|  2  |      NESTED JOIN (INNER JOIN)                                | 19343
|
|  3  |        TABLE ACCESS ("ORDERS")                               | 19343
```

```
|
|  4  |        INDEX ACCESS ("CUSTOMER", "CUSTOMER_PK_INDEX")     | 19343
|
=============================================================================
=


     1  -  TARGET : CUSTOMER.C_NAME, ORDERS.O_ORDERDATE,
ORDERS.O_ORDERSTATUS
     2  -  JOINED COLUMN : CUSTOMER.C_NAME, ORDERS.O_ORDERDATE,
ORDERS.O_ORDERSTATUS
     3  -  READ COLUMN : ORDERS.O_CUSTKEY, ORDERS.O_ORDERSTATUS,
ORDERS.O_ORDERDATE
             PHYSICAL FILTER : ORDERS.O_ORDERDATE < DATE'1995-03-15' +
CAST( '1' AS INTERVAL(MONTH) ) AND ORDERS.O_ORDERDATE >= DATE'1995-03-15'
     4  -  READ INDEX COLUMN : CUSTOMER.C_CUSTKEY
            READ TABLE COLUMN : CUSTOMER.C_NAME
              MIN RANGE : CUSTOMER.C_CUSTKEY = {ORDERS.O_CUSTKEY}
              MAX RANGE : CUSTOMER.C_CUSTKEY = {ORDERS.O_CUSTKEY}
            FETCH ONE ROW


<<<  end print plan
```

USE_NL_IN(alias)

描述USE_NL_IN(alias) hint时optimizer在决定alias参与的join的join operation时选择nested loop join并且将alias放在join的 right(inner)

Alias可以是table name, table alias name, view name, view alias name, join alias name

以下为使用USE_NL_IN(alias) hint的示例

```
\EXPLAIN PLAN
  SELECT  /*+ USE_NL_IN( orders ) */
          c_name,
          o_orderdate,
          o_orderstatus
    FROM  customer,
          orders
   WHERE  c_custkey = o_custkey
     AND  o_orderdate >= date '1995-03-15'
     AND  o_orderdate < date '1995-03-15' + interval '1' month;


>>>  start print plan


< Execution Plan >
================================================================================
==
| IDX |  NODE DESCRIPTION                                          |   ROWS
|
--------------------------------------------------------------------------------
--
|  0  |  SELECT STATEMENT                                          |  19343
|
```

```
|  1  |    QUERY BLOCK ("$QB_IDX_2")                                  |  19343
|
|  2  |      NESTED JOIN (INNER JOIN)                                 |  19343
|
|  3  |        TABLE ACCESS ("CUSTOMER")                              | 150000
|
|  4  |        INDEX ACCESS ("ORDERS", "ORDERS_CUSTKEY_FK")           |  19343
|
================================================================================
==


     1  -  TARGET : CUSTOMER.C_NAME, ORDERS.O_ORDERDATE,
ORDERS.O_ORDERSTATUS
     2  -  JOINED COLUMN : CUSTOMER.C_NAME, ORDERS.O_ORDERDATE,
ORDERS.O_ORDERSTATUS
     3  -  READ COLUMN : CUSTOMER.C_CUSTKEY, CUSTOMER.C_NAME
     4  -  READ INDEX COLUMN : ORDERS.O_CUSTKEY
           READ TABLE COLUMN : ORDERS.O_ORDERSTATUS, ORDERS.O_ORDERDATE
             MIN RANGE : ORDERS.O_CUSTKEY = {CUSTOMER.C_CUSTKEY}
             MAX RANGE : ORDERS.O_CUSTKEY = {CUSTOMER.C_CUSTKEY}
             PHYSICAL TABLE FILTER : ORDERS.O_ORDERDATE < DATE'1995-03-15'
+ CAST( '1' AS INTERVAL(MONTH) ) AND ORDERS.O_ORDERDATE >= DATE'1995-03-
15'


<<<  end print plan
```

USE_NL_OUT(alias)

描述USE_NL_OUT(alias) hint时optimizer在决定alias参与的join的join operation时选择nested loop join并且将alias放在join的left(outer)

Alias可以是table name, table alias name, view name, view alias name, join alias name

以下为使用USE_NL_OUT(alias) hint的示例

```
\EXPLAIN PLAN
  SELECT  /*+ USE_NL_OUT( orders ) */
          c_name,
          o_orderdate,
          o_orderstatus
    FROM  customer,
          orders
   WHERE  c_custkey = o_custkey
     AND  o_orderdate >= date '1995-03-15'
     AND  o_orderdate < date '1995-03-15' + interval '1' month;


< Execution Plan >
================================================================================
=
|  IDX  |  NODE DESCRIPTION                                          | ROWS
|
--------------------------------------------------------------------------------
-
```

```
|  0  |  SELECT STATEMENT                                          | 19343
|
|  1  |    QUERY BLOCK ("$QB_IDX_2")                               | 19343
|
|  2  |      NESTED JOIN (INNER JOIN)                              | 19343
|
|  3  |        TABLE ACCESS ("ORDERS")                             | 19343
|
|  4  |        INDEX ACCESS ("CUSTOMER", "CUSTOMER_PK_INDEX")      | 19343
|
================================================================================
=


     1  -  TARGET : CUSTOMER.C_NAME, ORDERS.O_ORDERDATE,
ORDERS.O_ORDERSTATUS
     2  -  JOINED COLUMN : CUSTOMER.C_NAME, ORDERS.O_ORDERDATE,
ORDERS.O_ORDERSTATUS
     3  -  READ COLUMN : ORDERS.O_CUSTKEY, ORDERS.O_ORDERSTATUS,
ORDERS.O_ORDERDATE
              PHYSICAL FILTER : ORDERS.O_ORDERDATE < DATE'1995-03-15' +
CAST( '1' AS INTERVAL(MONTH) ) AND ORDERS.O_ORDERDATE >= DATE'1995-03-15'
     4  -  READ INDEX COLUMN : CUSTOMER.C_CUSTKEY
            READ TABLE COLUMN : CUSTOMER.C_NAME
              MIN RANGE : CUSTOMER.C_CUSTKEY = {ORDERS.O_CUSTKEY}
              MAX RANGE : CUSTOMER.C_CUSTKEY = {ORDERS.O_CUSTKEY}
```

```
            FETCH ONE ROW


<<<  end print plan
```

NO_USE_NL(table_name [[,] table_name])

描述NO_USE_NL(table_name [[,] table_name]) hint时optimizer在决定table_name参与的join的join operation时排除nested loop join因此在除nested loop join外的其他join operation中选择最佳cost estimation

以下为使用NO_USE_NL(table_name [[,] table_name]) hint的示例

```
\EXPLAIN PLAN
  SELECT  /*+ NO_USE_NL( orders ) */
          c_name,
          o_orderdate,
          o_orderstatus
    FROM  customer,
          orders
   WHERE  c_custkey = o_custkey
     AND  o_orderdate >= date '1995-03-15'
     AND  o_orderdate < date '1995-03-15' + interval '1' month;


>>>  start print plan


< Execution Plan >
```

```
================================================================================
==
| IDX |  NODE DESCRIPTION                                          |   ROWS
|
--------------------------------------------------------------------------------
--
|  0  |  SELECT STATEMENT                                          |  19343
|
|  1  |    QUERY BLOCK ("$QB_IDX_2")                               |  19343
|
|  2  |      HASH JOIN (INNER JOIN)                                |  19343
|
|  3  |        TABLE ACCESS ("CUSTOMER")                           | 150000
|
|  4  |        HASH JOIN INSTANT                                   |  19343
|
|  5  |          TABLE ACCESS ("ORDERS")                           |  19343
|
================================================================================
==


     1  -  TARGET : CUSTOMER.C_NAME, ORDERS.O_ORDERDATE,
ORDERS.O_ORDERSTATUS
     2  -  JOINED COLUMN : CUSTOMER.C_NAME, ORDERS.O_ORDERDATE,
ORDERS.O_ORDERSTATUS
```

```
     3  -  READ COLUMN : CUSTOMER.C_CUSTKEY, CUSTOMER.C_NAME
     4  -  HASH KEY : ORDERS.O_CUSTKEY
           RECORD COLUMN : ORDERS.O_ORDERDATE, ORDERS.O_ORDERSTATUS
           READ KEY COLUMN : ORDERS.O_CUSTKEY, ORDERS.O_ORDERDATE,
ORDERS.O_ORDERSTATUS
             HASH FILTER : ORDERS.O_CUSTKEY = CUSTOMER.C_CUSTKEY
     5  -  READ COLUMN : ORDERS.O_CUSTKEY, ORDERS.O_ORDERSTATUS,
ORDERS.O_ORDERDATE
             PHYSICAL FILTER : ORDERS.O_ORDERDATE < DATE'1995-03-15' +
CAST( '1' AS INTERVAL(MONTH) ) AND ORDERS.O_ORDERDATE >= DATE'1995-03-15'


<<<  end print plan
```

USE_INL(table_name [[,] table_name])

描述USE_INL(table_name [[,] table_name]) hint时optimizer在决定table_name参与的join的join operation时选择instant nested loop join

应描述一个以上的table并且无法描述两个以上的相同的table描述的表中已有<join operation hints>时忽略此hint而且参与join的两张表中描述了各不相同的join operation hint时优先应用在right node(inner node)描述的hint

以下为使用USE_INL(table_name [[,] table_name]) hint的示例

```
\EXPLAIN PLAN
  SELECT  /*+ USE_INL( orders ) */
          c_name,
```

```
          o_orderdate,
          o_orderstatus
    FROM  customer,
          orders
   WHERE  c_custkey = o_custkey
     AND  o_orderdate >= date '1995-03-15'
     AND  o_orderdate < date '1995-03-15' + interval '1' month;


>>>  start print plan


< Execution Plan >
================================================================================
=
| IDX |  NODE DESCRIPTION                                          |   ROWS
|
--------------------------------------------------------------------------------
-
|  0  |  SELECT STATEMENT                                          |  19343
|
|  1  |    QUERY BLOCK ("$QB_IDX_2")                               |  19343
|
|  2  |      NESTED JOIN (INNER JOIN)                              |  19343
|
|  3  |        TABLE ACCESS ("ORDERS")                             |  19343
|
```

```
|  4  |        SORT JOIN INSTANT                                  |  19343
|
|  5  |          TABLE ACCESS ("CUSTOMER")                        | 150000
|
=============================================================================
=


     1  -  TARGET : CUSTOMER.C_NAME, ORDERS.O_ORDERDATE,
ORDERS.O_ORDERSTATUS
     2  -  JOINED COLUMN : CUSTOMER.C_NAME, ORDERS.O_ORDERDATE,
ORDERS.O_ORDERSTATUS
     3  -  READ COLUMN : ORDERS.O_CUSTKEY, ORDERS.O_ORDERSTATUS,
ORDERS.O_ORDERDATE
             PHYSICAL FILTER : ORDERS.O_ORDERDATE < DATE'1995-03-15' +
CAST( '1' AS INTERVAL(MONTH) ) AND ORDERS.O_ORDERDATE >= DATE'1995-03-15'
     4  -  SORT KEY : "CUSTOMER.C_CUSTKEY ASC NULLS LAST"
           RECORD COLUMN : CUSTOMER.C_NAME
           READ KEY COLUMN : CUSTOMER.C_CUSTKEY
           READ RECORD COLUMN : CUSTOMER.C_NAME
             MIN RANGE : CUSTOMER.C_CUSTKEY = {ORDERS.O_CUSTKEY}
             MAX RANGE : CUSTOMER.C_CUSTKEY = {ORDERS.O_CUSTKEY}
     5  -  READ COLUMN : CUSTOMER.C_CUSTKEY, CUSTOMER.C_NAME


<<<  end print plan
```

USE_INL_IN(alias)

描述USE_INL_IN(alias) hint时optimizer在决定alias参与的join的join operation时选择instant nested loop join并且将alias放在join的right(inner)

Alias可以是table name, table alias name, view name, view alias name, join alias name

以下为使用USE_INL_IN(alias) hint的示例

```
\EXPLAIN PLAN
  SELECT  /*+ USE_INL_IN( orders ) */
          c_name,
          o_orderdate,
          o_orderstatus
    FROM  customer,
          orders
   WHERE  c_custkey = o_custkey
     AND  o_orderdate >= date '1995-03-15'
     AND  o_orderdate < date '1995-03-15' + interval '1' month;


>>>  start print plan


< Execution Plan >
================================================================================
==
|  IDX  |  NODE DESCRIPTION                                          |   ROWS
|
```

```
-------------------------------------------------------------------------
--
|  0  |  SELECT STATEMENT                                    |  19343
|
|  1  |    QUERY BLOCK ("$QB_IDX_2")                         |  19343
|
|  2  |      NESTED JOIN (INNER JOIN)                        |  19343
|
|  3  |        TABLE ACCESS ("CUSTOMER")                     | 150000
|
|  4  |        SORT JOIN INSTANT                             |  19343
|
|  5  |          TABLE ACCESS ("ORDERS")                     |  19343
|
=========================================================================
==


     1  -  TARGET : CUSTOMER.C_NAME, ORDERS.O_ORDERDATE,
ORDERS.O_ORDERSTATUS
     2  -  JOINED COLUMN : CUSTOMER.C_NAME, ORDERS.O_ORDERDATE,
ORDERS.O_ORDERSTATUS
     3  -  READ COLUMN : CUSTOMER.C_CUSTKEY, CUSTOMER.C_NAME
     4  -  SORT KEY : "ORDERS.O_CUSTKEY ASC NULLS LAST"
           RECORD COLUMN : ORDERS.O_ORDERDATE, ORDERS.O_ORDERSTATUS
           READ KEY COLUMN : ORDERS.O_CUSTKEY
```

```
            READ RECORD COLUMN : ORDERS.O_ORDERDATE, ORDERS.O_ORDERSTATUS
              MIN RANGE : ORDERS.O_CUSTKEY = {CUSTOMER.C_CUSTKEY}
              MAX RANGE : ORDERS.O_CUSTKEY = {CUSTOMER.C_CUSTKEY}
     5   -  READ COLUMN : ORDERS.O_CUSTKEY, ORDERS.O_ORDERSTATUS,
ORDERS.O_ORDERDATE
              PHYSICAL FILTER : ORDERS.O_ORDERDATE < DATE'1995-03-15' +
CAST( '1' AS INTERVAL(MONTH) ) AND ORDERS.O_ORDERDATE >= DATE'1995-03-15'


<<<  end print plan
```

USE_INL_OUT(alias)

描述USE_INL_OUT(alias) hint时optimizer在决定alias参与的join的join operation时选择instant nested loop join并且将alias放在join的left(outer)

Alias可以是table name, table alias name, view name, view alias name, join alias name

以下为使用USE_INL_OUT(alias) hint的示例

```
\EXPLAIN PLAN
  SELECT  /*+ USE_INL_OUT( orders ) */
          c_name,
          o_orderdate,
          o_orderstatus
    FROM  customer,
          orders
   WHERE  c_custkey = o_custkey
```

```
     AND  o_orderdate >= date '1995-03-15'
     AND  o_orderdate < date '1995-03-15' + interval '1' month;

>>>  start print plan

< Execution Plan >
================================================================================
=
| IDX |  NODE DESCRIPTION                                          |   ROWS
|
--------------------------------------------------------------------------------
-
|  0  |  SELECT STATEMENT                                          |  19343
|
|  1  |    QUERY BLOCK ("$QB_IDX_2")                               |  19343
|
|  2  |      NESTED JOIN (INNER JOIN)                              |  19343
|
|  3  |        TABLE ACCESS ("ORDERS")                             |  19343
|
|  4  |        SORT JOIN INSTANT                                   |  19343
|
|  5  |          TABLE ACCESS ("CUSTOMER")                         | 150000
|
================================================================================
```

```
=


     1  -  TARGET : CUSTOMER.C_NAME, ORDERS.O_ORDERDATE,
ORDERS.O_ORDERSTATUS
     2  -  JOINED COLUMN : CUSTOMER.C_NAME, ORDERS.O_ORDERDATE,
ORDERS.O_ORDERSTATUS
     3  -  READ COLUMN : ORDERS.O_CUSTKEY, ORDERS.O_ORDERSTATUS,
ORDERS.O_ORDERDATE
              PHYSICAL FILTER : ORDERS.O_ORDERDATE < DATE'1995-03-15' +
CAST( '1' AS INTERVAL(MONTH) ) AND ORDERS.O_ORDERDATE >= DATE'1995-03-15'
     4  -  SORT KEY : "CUSTOMER.C_CUSTKEY ASC NULLS LAST"
            RECORD COLUMN : CUSTOMER.C_NAME
            READ KEY COLUMN : CUSTOMER.C_CUSTKEY
            READ RECORD COLUMN : CUSTOMER.C_NAME
              MIN RANGE : CUSTOMER.C_CUSTKEY = {ORDERS.O_CUSTKEY}
              MAX RANGE : CUSTOMER.C_CUSTKEY = {ORDERS.O_CUSTKEY}
     5  -  READ COLUMN : CUSTOMER.C_CUSTKEY, CUSTOMER.C_NAME


<<<  end print plan
```

NO_USE_INL(table_name [[,] table_name])

描述NO_USE_INL(table_name [[,] table_name]) hint时optimizer在决定table_name参与的join的join operation时排除instant nested loop join因此在除instant nested loop join外的其他join operation中选择最佳cost estimation

应描述一个以上的table并且无法描述两个以上的相同的table描述的表中已有<join operation hints>时忽略此hint而且参与join的两张表中描述了各不相同的join operation hint时优先应用在right node(inner node)描述的hint

以下为使用NO_USE_INL(table_name [[,] table_name]) hint的示例

```
\EXPLAIN PLAN
  SELECT  /*+ NO_USE_INL( orders ) */
          c_name,
          o_orderdate,
          o_orderstatus
    FROM  customer,
          orders
   WHERE  c_custkey = o_custkey
     AND  o_orderdate >= date '1995-03-15'
     AND  o_orderdate < date '1995-03-15' + interval '1' month;


< Execution Plan >
================================================================================
=
|  IDX  |  NODE DESCRIPTION                                              | ROWS
|
--------------------------------------------------------------------------------
-
|  0  |  SELECT STATEMENT                                                | 19343
|
```

```
|  1  |    QUERY BLOCK ("$QB_IDX_2")                                  | 19343
|
|  2  |      NESTED JOIN (INNER JOIN)                                 | 19343
|
|  3  |        TABLE ACCESS ("ORDERS")                                | 19343
|
|  4  |        INDEX ACCESS ("CUSTOMER", "CUSTOMER_PK_INDEX")         | 19343
|
=============================================================================
=


     1  -  TARGET : CUSTOMER.C_NAME, ORDERS.O_ORDERDATE,
ORDERS.O_ORDERSTATUS
     2  -  JOINED COLUMN : CUSTOMER.C_NAME, ORDERS.O_ORDERDATE,
ORDERS.O_ORDERSTATUS
     3  -  READ COLUMN : ORDERS.O_CUSTKEY, ORDERS.O_ORDERSTATUS,
ORDERS.O_ORDERDATE
              PHYSICAL FILTER : ORDERS.O_ORDERDATE < DATE'1995-03-15' +
CAST( '1' AS INTERVAL(MONTH) ) AND ORDERS.O_ORDERDATE >= DATE'1995-03-15'
     4  -  READ INDEX COLUMN : CUSTOMER.C_CUSTKEY
            READ TABLE COLUMN : CUSTOMER.C_NAME
              MIN RANGE : CUSTOMER.C_CUSTKEY = {ORDERS.O_CUSTKEY}
              MAX RANGE : CUSTOMER.C_CUSTKEY = {ORDERS.O_CUSTKEY}
            FETCH ONE ROW
```

```
<<<  end print plan
```

USE_JOIN_COMBINE(alias)

描述USE_JOIN_COMBINE(alias) hint时optimizer在决定alias参与的join的join operation时选择join combine

Alias可以是table name, table alias name, view name, view alias name, join alias name

以下为使用USE_JOIN_COMBINE(alias) hint的示例

```
\EXPLAIN PLAN
SELECT /*+ USE_JOIN_COMBINE(part) */
        sum(l_extendedprice * (1 - l_discount) ) as revenue
  FROM lineitem,
       part
 WHERE
       (
            p_partkey = l_partkey
        and p_brand = 'Brand#12'
        and p_container in ( 'SM CASE', 'SM BOX', 'SM PACK', 'SM PKG')
        and l_quantity >= 1 and l_quantity <= 1 + 10
        and p_size between 1 and 5
        and l_shipmode in ('AIR', 'AIR REG')
        and l_shipinstruct = 'DELIVER IN PERSON'
       )
       or
```

```
       (
            p_partkey = l_partkey
        and p_brand = 'Brand#23'
        and p_container in ('MED BAG', 'MED BOX', 'MED PKG', 'MED PACK')
        and l_quantity >= 10 and l_quantity <= 10 + 10
        and p_size between 1 and 10
        and l_shipmode in ('AIR', 'AIR REG')
        and l_shipinstruct = 'DELIVER IN PERSON'
       )
      or
      (
            p_partkey = l_partkey
        and p_brand = 'Brand#34'
        and p_container in ( 'LG CASE', 'LG BOX', 'LG PACK', 'LG PKG')
        and l_quantity >= 20 and l_quantity <= 20 + 10
        and p_size between 1 and 15
        and l_shipmode in ('AIR', 'AIR REG')
        and l_shipinstruct = 'DELIVER IN PERSON'
      );


>>>  start print plan


< Execution Plan >

================================================================================
=
```

```
|IDX|  NODE DESCRIPTION

|

-------------------------------------------------------------------------
-

| 0 |  SELECT STATEMENT

|

| 1 |    QUERY BLOCK ("$QB_IDX_2")

|

| 2 |      AGGREGATION BY HASH

|

| 3 |        CONCAT (Compare Nothing)

|

| 4 |          NESTED JOIN (INNER JOIN)

|

| 5 |            TABLE ACCESS ("PART")

|

| 6 |            INDEX ACCESS ("LINEITEM", "LINEITEM_PARTKEY_SUPPKEY_FK")

|

| 7 |          NESTED JOIN (INNER JOIN)

|

| 8 |            TABLE ACCESS ("PART")

|

| 9 |            INDEX ACCESS ("LINEITEM", "LINEITEM_PARTKEY_SUPPKEY_FK")

|

|10 |          NESTED JOIN (INNER JOIN)
```

```
|
|11 |            TABLE ACCESS ("PART")
|
|12 |            INDEX ACCESS ("LINEITEM", "LINEITEM_PARTKEY_SUPPKEY_FK")
|
================================================================================
=



     1  -  TARGET : SUM( LINEITEM.L_EXTENDEDPRICE * ( 1 -
LINEITEM.L_DISCOUNT ) ) AS REVENUE
     2  -  AGGREGATION : SUM( LINEITEM.L_EXTENDEDPRICE * ( 1 -
LINEITEM.L_DISCOUNT ) )
     3  -  CONCAT COLUMN : LINEITEM.L_EXTENDEDPRICE, LINEITEM.L_DISCOUNT
     4  -  JOINED COLUMN : LINEITEM.L_EXTENDEDPRICE, LINEITEM.L_DISCOUNT
     5  -  READ COLUMN : PART.P_PARTKEY, PART.P_BRAND, PART.P_SIZE,
PART.P_CONTAINER
             PHYSICAL FILTER : PART.P_BRAND = 'Brand#12' AND PART.P_SIZE
<= 5 AND PART.P_SIZE >= 1 AND ( PART.P_CONTAINER ) IN ( 'SM CASE', 'SM
BOX', 'SM PACK', 'SM PKG' )
     6  -  READ INDEX COLUMN : LINEITEM.L_PARTKEY
           READ TABLE COLUMN : LINEITEM.L_QUANTITY,
LINEITEM.L_EXTENDEDPRICE, LINEITEM.L_DISCOUNT, LINEITEM.L_SHIPINSTRUCT,
LINEITEM.L_SHIPMODE
             MIN RANGE : LINEITEM.L_PARTKEY = {PART.P_PARTKEY}
```

```
            MAX RANGE : LINEITEM.L_PARTKEY = {PART.P_PARTKEY}
            PHYSICAL TABLE FILTER : LINEITEM.L_QUANTITY <= 1 + 10 AND
LINEITEM.L_QUANTITY >= 1 AND LINEITEM.L_SHIPINSTRUCT = 'DELIVER IN PERSON'
AND ( LINEITEM.L_SHIPMODE ) IN ( 'AIR', 'AIR REG' )
     7  -  JOINED COLUMN : LINEITEM.L_EXTENDEDPRICE, LINEITEM.L_DISCOUNT
     8  -  READ COLUMN : PART.P_PARTKEY, PART.P_BRAND, PART.P_SIZE,
PART.P_CONTAINER
            PHYSICAL FILTER : PART.P_BRAND = 'Brand#23' AND PART.P_SIZE
<= 10 AND PART.P_SIZE >= 1 AND ( PART.P_CONTAINER ) IN ( 'MED BAG', 'MED
BOX', 'MED PKG', 'MED PACK' )
     9  -  READ INDEX COLUMN : LINEITEM.L_PARTKEY
           READ TABLE COLUMN : LINEITEM.L_QUANTITY,
LINEITEM.L_EXTENDEDPRICE, LINEITEM.L_DISCOUNT, LINEITEM.L_SHIPINSTRUCT,
LINEITEM.L_SHIPMODE
            MIN RANGE : LINEITEM.L_PARTKEY = {PART.P_PARTKEY}
            MAX RANGE : LINEITEM.L_PARTKEY = {PART.P_PARTKEY}
            PHYSICAL TABLE FILTER : LINEITEM.L_QUANTITY <= 10 + 10 AND
LINEITEM.L_QUANTITY >= 10 AND LINEITEM.L_SHIPINSTRUCT = 'DELIVER IN
PERSON' AND ( LINEITEM.L_SHIPMODE ) IN ( 'AIR', 'AIR REG' )
    10  -  JOINED COLUMN : LINEITEM.L_EXTENDEDPRICE, LINEITEM.L_DISCOUNT
    11  -  READ COLUMN : PART.P_PARTKEY, PART.P_BRAND, PART.P_SIZE,
PART.P_CONTAINER
            PHYSICAL FILTER : PART.P_BRAND = 'Brand#34' AND PART.P_SIZE
<= 15 AND PART.P_SIZE >= 1 AND ( PART.P_CONTAINER ) IN ( 'LG CASE', 'LG
BOX', 'LG PACK', 'LG PKG' )
```

```
    12  -  READ INDEX COLUMN : LINEITEM.L_PARTKEY
           READ TABLE COLUMN : LINEITEM.L_QUANTITY,
LINEITEM.L_EXTENDEDPRICE, LINEITEM.L_DISCOUNT, LINEITEM.L_SHIPINSTRUCT,
LINEITEM.L_SHIPMODE
             MIN RANGE : LINEITEM.L_PARTKEY = {PART.P_PARTKEY}
             MAX RANGE : LINEITEM.L_PARTKEY = {PART.P_PARTKEY}
             PHYSICAL TABLE FILTER : LINEITEM.L_QUANTITY <= 20 + 10 AND
LINEITEM.L_QUANTITY >= 20 AND LINEITEM.L_SHIPINSTRUCT = 'DELIVER IN
PERSON' AND ( LINEITEM.L_SHIPMODE ) IN ( 'AIR', 'AIR REG' )


<<<  end print plan
```

NO_USE_JOIN_COMBINE(alias)

描述NO_USE_JOIN_COMBINE(alias) hint时optimizer在决定alias参与的join的join operation时排除join combine

Alias可以是table name, table alias name, view name, view alias name, join alias name

以下为使用 NO_USE_JOIN_COMBINE(alias) hint的示例

```
\EXPLAIN PLAN
SELECT /*+ NO_USE_JOIN_COMBINE(part) */
        sum(l_extendedprice * (1 - l_discount) ) as revenue
  FROM lineitem,
       part
 WHERE
```

```
 (
      p_partkey = l_partkey
  and p_brand = 'Brand#12'
  and p_container in ( 'SM CASE', 'SM BOX', 'SM PACK', 'SM PKG')
  and l_quantity >= 1 and l_quantity <= 1 + 10
  and p_size between 1 and 5
  and l_shipmode in ('AIR', 'AIR REG')
  and l_shipinstruct = 'DELIVER IN PERSON'
 )
 or
 (
      p_partkey = l_partkey
  and p_brand = 'Brand#23'
  and p_container in ('MED BAG', 'MED BOX', 'MED PKG', 'MED PACK')
  and l_quantity >= 10 and l_quantity <= 10 + 10
  and p_size between 1 and 10
  and l_shipmode in ('AIR', 'AIR REG')
  and l_shipinstruct = 'DELIVER IN PERSON'
 )
or
(
      p_partkey = l_partkey
  and p_brand = 'Brand#34'
  and p_container in ( 'LG CASE', 'LG BOX', 'LG PACK', 'LG PKG')
  and l_quantity >= 20 and l_quantity <= 20 + 10
```

```
        and p_size between 1 and 15
        and l_shipmode in ('AIR', 'AIR REG')
        and l_shipinstruct = 'DELIVER IN PERSON'
      );


>>>  start print plan


< Execution Plan >
================================================================================
=
|IDX|  NODE DESCRIPTION
|
--------------------------------------------------------------------------------
-
| 0 |  SELECT STATEMENT
|
| 1 |    QUERY BLOCK ("$QB_IDX_2")
|
| 2 |      AGGREGATION BY HASH
|
| 3 |        NESTED JOIN (INNER JOIN)
|
| 4 |          TABLE ACCESS ("PART")
|
| 5 |          CONCAT (Compare Nothing)
```

```
|
| 6 |            INDEX ACCESS ("LINEITEM", "LINEITEM_PARTKEY_SUPPKEY_FK")
|
| 7 |            INDEX ACCESS ("LINEITEM", "LINEITEM_PARTKEY_SUPPKEY_FK")
|
| 8 |            INDEX ACCESS ("LINEITEM", "LINEITEM_PARTKEY_SUPPKEY_FK")
|
=========================================================================
=


     1  -  TARGET : SUM( LINEITEM.L_EXTENDEDPRICE * ( 1 -
LINEITEM.L_DISCOUNT ) ) AS REVENUE
     2  -  AGGREGATION : SUM( LINEITEM.L_EXTENDEDPRICE * ( 1 -
LINEITEM.L_DISCOUNT ) )
     3  -  JOINED COLUMN : LINEITEM.L_EXTENDEDPRICE, LINEITEM.L_DISCOUNT
     4  -  READ COLUMN : PART.P_PARTKEY, PART.P_BRAND, PART.P_SIZE,
PART.P_CONTAINER
     5  -  CONCAT COLUMN : LINEITEM.L_EXTENDEDPRICE, LINEITEM.L_DISCOUNT
     6  -  READ INDEX COLUMN : LINEITEM.L_PARTKEY
           READ TABLE COLUMN : LINEITEM.L_QUANTITY,
LINEITEM.L_EXTENDEDPRICE, LINEITEM.L_DISCOUNT, LINEITEM.L_SHIPINSTRUCT,
LINEITEM.L_SHIPMODE
             MIN RANGE : LINEITEM.L_PARTKEY = {PART.P_PARTKEY}
             MAX RANGE : LINEITEM.L_PARTKEY = {PART.P_PARTKEY}
             PHYSICAL TABLE FILTER : LINEITEM.L_QUANTITY <= 1 + 10 AND
```

```
LINEITEM.L_QUANTITY >= 1 AND LINEITEM.L_SHIPINSTRUCT = 'DELIVER IN PERSON'
AND ( LINEITEM.L_SHIPMODE ) IN ( 'AIR', 'AIR REG' )
             CONSTANT FILTER : {PART.P_BRAND} = 'Brand#12' AND
( {PART.P_CONTAINER} ) IN ( 'SM CASE', 'SM BOX', 'SM PACK', 'SM PKG' ) AND
{PART.P_SIZE} <= 5 AND {PART.P_SIZE} >= 1
     7  -  READ INDEX COLUMN : LINEITEM.L_PARTKEY
           READ TABLE COLUMN : LINEITEM.L_QUANTITY,
LINEITEM.L_EXTENDEDPRICE, LINEITEM.L_DISCOUNT, LINEITEM.L_SHIPINSTRUCT,
LINEITEM.L_SHIPMODE
             MIN RANGE : LINEITEM.L_PARTKEY = {PART.P_PARTKEY}
             MAX RANGE : LINEITEM.L_PARTKEY = {PART.P_PARTKEY}
             PHYSICAL TABLE FILTER : LINEITEM.L_QUANTITY <= 10 + 10 AND
LINEITEM.L_QUANTITY >= 10 AND LINEITEM.L_SHIPINSTRUCT = 'DELIVER IN
PERSON' AND ( LINEITEM.L_SHIPMODE ) IN ( 'AIR', 'AIR REG' )
             CONSTANT FILTER : {PART.P_BRAND} = 'Brand#23' AND
( {PART.P_CONTAINER} ) IN ( 'MED BAG', 'MED BOX', 'MED PKG', 'MED PACK' )
AND {PART.P_SIZE} <= 10 AND {PART.P_SIZE} >= 1
     8  -  READ INDEX COLUMN : LINEITEM.L_PARTKEY
           READ TABLE COLUMN : LINEITEM.L_QUANTITY,
LINEITEM.L_EXTENDEDPRICE, LINEITEM.L_DISCOUNT, LINEITEM.L_SHIPINSTRUCT,
LINEITEM.L_SHIPMODE
             MIN RANGE : LINEITEM.L_PARTKEY = {PART.P_PARTKEY}
             MAX RANGE : LINEITEM.L_PARTKEY = {PART.P_PARTKEY}
             PHYSICAL TABLE FILTER : LINEITEM.L_QUANTITY <= 20 + 10 AND
LINEITEM.L_QUANTITY >= 20 AND LINEITEM.L_SHIPINSTRUCT = 'DELIVER IN
```

```
PERSON' AND ( LINEITEM.L_SHIPMODE ) IN ( 'AIR', 'AIR REG' )

             CONSTANT FILTER : {PART.P_BRAND} = 'Brand#34' AND

( {PART.P_CONTAINER} ) IN ( 'LG CASE', 'LG BOX', 'LG PACK', 'LG PKG' ) AND

{PART.P_SIZE} <= 15 AND {PART.P_SIZE} >= 1


<<<  end print plan
```

<join driver hints>

在集群系统执行join时可使用的hint在单机版被忽略

LOCAL_JOIN(alias)

Alias参与join时在当前服务器执行joinLeft child和right child均为sharded table时将下级的所有row获取到local并执行join

以下为使用LOCAL_JOIN hint的示例

```
\EXPLAIN PLAN

SELECT /*+ LOCAL_JOIN(lineitem) */

       l_orderkey, p_partkey

  FROM part, lineitem

 WHERE p_partkey = l_partkey;


>>>  start print plan


< Execution Plan >
```

```
================================================================================
==
|  IDX  |  NODE DESCRIPTION                          |                ROWS
|
--------------------------------------------------------------------------------
--
| 0 |  SELECT STATEMENT                              |                2528
|
| 1 |    QUERY BLOCK ("$QB_IDX_2")                   |                2528
|
| 2 |      HASH JOIN (INNER JOIN)                    |                2528
|
| 3 |        PLAN BASED CLUSTER                      | LOCAL/REMOTE 200000
|
| 4 |          INDEX ACCESS ("PART", "PART_PK_INDEX")|       (66675) 66675
|
| 5 |        HASH JOIN INSTANT                       |                2528
|
| 6 |          PLAN BASED CLUSTER                    |   LOCAL/REMOTE 2528
|
| 7 |            TABLE ACCESS ("LINEITEM")           |                 840
|
================================================================================
==
```

```
1  -  TARGET : LINEITEM.L_ORDERKEY, PART.P_PARTKEY
2  -  JOINED COLUMN : LINEITEM.L_ORDERKEY, PART.P_PARTKEY
3  -  SQL : SELECT /*+ INDEX( _A1, "PUBLIC"."PART_PK_INDEX" ) */
                      "_A1"."P_PARTKEY"
                FROM "PUBLIC"."PART"@LOCAL AS "_A1"
      TARGET DOMAIN : G1(G1N1,G1N2) 66675 rows,
                      G2(G2N1,G2N2) 66664 rows,
                      G3(G3N1,G3N2) 66661 rows
4  -  HASH SHARD ( # 3 )
      READ INDEX COLUMN : PART.P_PARTKEY
5  -  HASH KEY : LINEITEM.L_PARTKEY
      RECORD COLUMN : LINEITEM.L_ORDERKEY
      READ KEY COLUMN : LINEITEM.L_PARTKEY, LINEITEM.L_ORDERKEY
        HASH FILTER : LINEITEM.L_PARTKEY = PART.P_PARTKEY
6  -  SQL : SELECT /*+ FULL( _A1 ) */
                      "_A1"."L_ORDERKEY", "_A1"."L_PARTKEY"
                FROM "PUBLIC"."LINEITEM"@LOCAL AS "_A1"
               WHERE "_A1"."L_SHIPDATE" = :_V0
      TARGET DOMAIN : G1(G1N1,G1N2) 840 rows,
                      G2(G2N1,G2N2) 824 rows,
                      G3(G3N1,G3N2) 864 rows
7  -  HASH SHARD ( # 3 )
      READ COLUMN : LINEITEM.L_ORDERKEY, LINEITEM.L_PARTKEY,
LINEITEM.L_SHIPDATE
        PHYSICAL FILTER : LINEITEM.L_SHIPDATE = DATE'1995-03-15'
```

```
<<<  end print plan
```

Part和lineitem均为hash sharded table上述execution plan中两张表均从G1G2G3获取并在local执行了join

REMOTE_JOIN(alias)

Alias参与join时在各个服务器执行join

以下为执行REMOTE_JOIN hint的示例

```
\EXPLAIN PLAN
SELECT /*+ REMOTE_JOIN(lineitem) */
       l_orderkey, p_partkey
  FROM part, lineitem
 WHERE p_partkey = l_partkey
   AND l_shipdate = date '1995-03-15';



>>>  start print plan

< Execution Plan >
================================================================================
==
| IDX |NODE DESCRIPTION                                        |      ROWS
|
```

```
-------------------------------------------------------------------------
--
|  0 |SELECT STATEMENT                                    |      2528
|
|  1 |  QUERY BLOCK ("$QB_IDX_2")                         |      2528
|
|  2 |    SINGLE CLUSTER                                  | LOCAL/REMOTE 2528
|
|  3 |      CLUSTER PUSHER ("_$NI_5")                     |      2528
|
|  4 |        PLAN BASED CLUSTER                          | LOCAL/REMOTE 2528
|
|  5 |          TABLE ACCESS ("LINEITEM")                 |       840
|
|  6 |      SELECT STATEMENT                              |       857
|
|  7 |        QUERY BLOCK ("$QB_IDX_2")                   |       857
|
|  8 |          NESTED JOIN (INNER JOIN)                  |       857
|
|  9 |            PUSHER TABLE ACCESS ("_$NI_5" AS _A2)   |       857
|
| 10 |            INDEX ACCESS ("PART" AS _A1, "PART_PK_INDEX")| (857) 857
|
=========================================================================
```

```
==

     1  -  TARGET : _$NI_5.L_ORDERKEY, PART.P_PARTKEY
     2  -  SQL : SELECT /*+ KEEP_JOINED_TABLE
                            USE_NL_IN( _A1 )
                            FULL( _A2 )
                            INDEX( _A1, "PUBLIC"."PART_PK_INDEX" )
                          */
                          "_A2"."L_ORDERKEY", "_A1"."P_PARTKEY"
                     FROM ( "SESSION_SCHEMA"."_$NI_5"@LOCAL AS "_A2"
                              INNER JOIN
                              "PUBLIC"."PART"@LOCAL AS "_A1"
                              ON true
                            ) ALIAS "_A3"
                    WHERE "_A1"."P_PARTKEY" = "_A2"."L_PARTKEY"
            TARGET DOMAIN : G1(G1N1,G1N2) 857 rows,
                            G2(G2N1,G2N2) 845 rows,
                            G3(G3N1,G3N2) 826 rows
     3  -  SQL : DECLARE INSTANT TABLE "SESSION_SCHEMA"."_$NI_5"
                   ( "L_PARTKEY" NUMBER(10, 0), "L_ORDERKEY" NUMBER(10, 0) )
            COLUMN : LINEITEM.L_PARTKEY AS L_PARTKEY, LINEITEM.L_ORDERKEY
AS L_ORDERKEY
            SHARDED : LINEITEM.L_PARTKEY
            TARGET DOMAIN : G1(G1N1,G1N2) 857 rows,
                            G2(G2N1,G2N2) 845 rows,
```

```
                          G3(G3N1,G3N2) 826 rows
     4  -  SQL : SELECT /*+ FULL( _A1 ) */
                        "_A1"."L_ORDERKEY", "_A1"."L_PARTKEY"
                    FROM "PUBLIC"."LINEITEM"@LOCAL AS "_A1"
                   WHERE "_A1"."L_SHIPDATE" = :_V0
           TARGET DOMAIN : G1(G1N1,G1N2) 840 rows,
                          G2(G2N1,G2N2) 824 rows,
                          G3(G3N1,G3N2) 864 rows
     5  -  HASH SHARD ( # 3 )
           READ COLUMN : LINEITEM.L_ORDERKEY, LINEITEM.L_PARTKEY,
LINEITEM.L_SHIPDATE
             PHYSICAL FILTER : LINEITEM.L_SHIPDATE = DATE'1995-03-15'
     7  -  TARGET : _A2.L_ORDERKEY, _A1.P_PARTKEY
     8  -  JOINED COLUMN : _A2.L_ORDERKEY, _A1.P_PARTKEY
             CONSTANT FILTER : TRUE
     9  -  READ COLUMN : _A2.L_PARTKEY, _A2.L_ORDERKEY
    10  -  HASH SHARD ( # 3 )
           READ INDEX COLUMN : _A1.P_PARTKEY
             MIN RANGE : _A1.P_PARTKEY = {_A2.L_PARTKEY}
             MAX RANGE : _A1.P_PARTKEY = {_A2.L_PARTKEY}
           FETCH ONE ROW


<<<  end print plan
```

如上述execution plan所示将lineitem生成至l_partkey为shard key的pusher table后向各个服务器

传输SQL并执行了remote join

<join pusher hints>

为在集群系统执行remote join时可使用的hint在单机版被忽略

PUSHER(alias)

Alias参与remote join时在pusher table部署属于alias的表或view

即使没有pusher table但可以remote join时不应用hint

以下为使用PUSHER hint的示例

```
\EXPLAIN PLAN
SELECT /*+ PUSHER(lineitem) */
       l_orderkey, p_partkey
  FROM part, lineitem
 WHERE p_partkey = l_partkey
   AND l_shipdate = date '1995-03-15';


>>>  start print plan


< Execution Plan >
=============================================================================
==
|  IDX  |  NODE DESCRIPTION                                          | ROWS
|
```

```
-------------------------------------------------------------------------
--
|  0 |  SELECT STATEMENT                                        | 2528
|
|  1 |    QUERY BLOCK ("$QB_IDX_2")                             | 2528
|
|  2 |      SINGLE CLUSTER                             | LOCAL/REMOTE 2528
|
|  3 |        CLUSTER PUSHER ("_$NI_5")                         | 2528
|
|  4 |          PLAN BASED CLUSTER                     | LOCAL/REMOTE 2528
|
|  5 |            TABLE ACCESS ("LINEITEM")                     | 840
|
|  6 |        SELECT STATEMENT                                  | 857
|
|  7 |          QUERY BLOCK ("$QB_IDX_2")                       | 857
|
|  8 |            NESTED JOIN (INNER JOIN)                      | 857
|
|  9 |              PUSHER TABLE ACCESS ("_$NI_5" AS _A2)       | 857
|
| 10 |              INDEX ACCESS ("PART" AS _A1, "PART_PK_INDEX")    | 857
|
=========================================================================
```

```
==


     1  -  TARGET : _$NI_5.L_ORDERKEY, PART.P_PARTKEY
     2  -  SQL : SELECT /*+ KEEP_JOINED_TABLE
                            USE_NL_IN( _A1 )
                            FULL( _A2 )
                            INDEX( _A1, "PUBLIC"."PART_PK_INDEX" )
                          */
                          "_A2"."L_ORDERKEY", "_A1"."P_PARTKEY"
                    FROM ( "SESSION_SCHEMA"."_$NI_5"@LOCAL AS "_A2"
                           INNER JOIN
                           "PUBLIC"."PART"@LOCAL AS "_A1"
                           ON true
                         ) ALIAS "_A3"
                   WHERE "_A1"."P_PARTKEY" = "_A2"."L_PARTKEY"
           TARGET DOMAIN : G1(G1N1,G1N2) 857 rows,
                           G2(G2N1,G2N2) 845 rows,
                           G3(G3N1,G3N2) 826 rows
     3  -  SQL : DECLARE INSTANT TABLE "SESSION_SCHEMA"."_$NI_5"
                  ( "L_PARTKEY" NUMBER(10, 0), "L_ORDERKEY" NUMBER(10, 0) )
           COLUMN : LINEITEM.L_PARTKEY AS L_PARTKEY, LINEITEM.L_ORDERKEY
AS L_ORDERKEY
           SHARDED : LINEITEM.L_PARTKEY
           TARGET DOMAIN : G1(G1N1,G1N2) 857 rows,
                           G2(G2N1,G2N2) 845 rows,
```

```
                          G3(G3N1,G3N2) 826 rows
     4  -  SQL : SELECT /*+ FULL( _A1 ) */
                        "_A1"."L_ORDERKEY", "_A1"."L_PARTKEY"
                    FROM "PUBLIC"."LINEITEM"@LOCAL AS "_A1"
                   WHERE "_A1"."L_SHIPDATE" = :_V0
           TARGET DOMAIN : G1(G1N1,G1N2) 840 rows,
                          G2(G2N1,G2N2) 824 rows,
                          G3(G3N1,G3N2) 864 rows
     5  -  HASH SHARD ( # 3 )
           READ COLUMN : LINEITEM.L_ORDERKEY, LINEITEM.L_PARTKEY,
LINEITEM.L_SHIPDATE
             PHYSICAL FILTER : LINEITEM.L_SHIPDATE = DATE'1995-03-15'
     7  -  TARGET : _A2.L_ORDERKEY, _A1.P_PARTKEY
     8  -  JOINED COLUMN : _A2.L_ORDERKEY, _A1.P_PARTKEY
             CONSTANT FILTER : TRUE
     9  -  READ COLUMN : _A2.L_PARTKEY, _A2.L_ORDERKEY
    10  -  HASH SHARD ( # 3 )
           READ INDEX COLUMN : _A1.P_PARTKEY
             MIN RANGE : _A1.P_PARTKEY = {_A2.L_PARTKEY}
             MAX RANGE : _A1.P_PARTKEY = {_A2.L_PARTKEY}
           FETCH ONE ROW


<<<  end print plan
```

如上述execution plan所示将lineitem生成至l_partkey为shard key的pusher table后向各个服务器

传输SQL并执行了remote join

NO_PUSHER(alias)

Alias参与remote join时不在pusher table部署属于alias的表或view

在pusher table部署alias的sibling时如果remote join成本高则会选择local join

此时如果同时使用REMOTE_JOIN hint则在pusher table部署alias的sibling并执行remote join

以下为使用NO_PUSHER hint的示例

```
\EXPLAIN PLAN
SELECT /*+ NO_PUSHER(lineitem) */
       l_orderkey, p_partkey
  FROM part, lineitem
 WHERE p_partkey = l_partkey
   AND l_shipdate = date '1995-03-15';


>>>  start print plan


< Execution Plan >
================================================================================
==
|IDX|  NODE DESCRIPTION                                |                ROWS
|
--------------------------------------------------------------------------------
--
```

```
| 0 | SELECT STATEMENT                                |                2528
|
| 1 |   QUERY BLOCK ("$QB_IDX_2")                     |                2528
|
| 2 |     HASH JOIN (INNER JOIN)                      |                2528
|
| 3 |       PLAN BASED CLUSTER                        | LOCAL/REMOTE 200000
|
| 4 |         INDEX ACCESS ("PART", "PART_PK_INDEX") |     (66675)   66675
|
| 5 |       HASH JOIN INSTANT                         |                2528
|
| 6 |         PLAN BASED CLUSTER                      | LOCAL/REMOTE   2528
|
| 7 |           TABLE ACCESS ("LINEITEM")             |                 840
|
================================================================================
==


     1  -  TARGET : LINEITEM.L_ORDERKEY, PART.P_PARTKEY
     2  -  JOINED COLUMN : LINEITEM.L_ORDERKEY, PART.P_PARTKEY
     3  -  SQL : SELECT /*+ INDEX( _A1, "PUBLIC"."PART_PK_INDEX" ) */
                        "_A1"."P_PARTKEY"
                   FROM "PUBLIC"."PART"@LOCAL AS "_A1"
            TARGET DOMAIN : G1(G1N1,G1N2) 66675 rows,
```

```
                          G2(G2N1,G2N2) 66664 rows,
                          G3(G3N1,G3N2) 66661 rows
     4  -  HASH SHARD ( # 3 )
           READ INDEX COLUMN : PART.P_PARTKEY
     5  -  HASH KEY : LINEITEM.L_PARTKEY
           RECORD COLUMN : LINEITEM.L_ORDERKEY
           READ KEY COLUMN : LINEITEM.L_PARTKEY, LINEITEM.L_ORDERKEY
             HASH FILTER : LINEITEM.L_PARTKEY = PART.P_PARTKEY
     6  -  SQL : SELECT /*+ FULL( _A1 ) */
                        "_A1"."L_ORDERKEY", "_A1"."L_PARTKEY"
                    FROM "PUBLIC"."LINEITEM"@LOCAL AS "_A1"
                   WHERE "_A1"."L_SHIPDATE" = :_V0
           TARGET DOMAIN : G1(G1N1,G1N2) 840 rows,
                          G2(G2N1,G2N2) 824 rows,
                          G3(G3N1,G3N2) 864 rows
     7  -  HASH SHARD ( # 3 )
           READ COLUMN : LINEITEM.L_ORDERKEY, LINEITEM.L_PARTKEY,
LINEITEM.L_SHIPDATE
             PHYSICAL FILTER : LINEITEM.L_SHIPDATE = DATE'1995-03-15'


<<<  end print plan
```

如上述execution plan所示以local join执行在pusher table部署alias的sibling时如果remote join成本高如上可选择local join

```
\EXPLAIN PLAN
```

```
SELECT /*+ REMOTE_JOIN(lineitem) NO_PUSHER(lineitem) */
       l_orderkey, p_partkey
  FROM part, lineitem
 WHERE p_partkey = l_partkey
   AND l_shipdate = date '1995-03-15';


>>>  start print plan


< Execution Plan >
================================================================================
==
| IDX|  NODE DESCRIPTION                                |               ROWS
|
--------------------------------------------------------------------------------
--
|  0 |  SELECT STATEMENT                                |               2528
|
|  1 |    QUERY BLOCK ("$QB_IDX_2")                     |               2528
|
|  2 |      SINGLE CLUSTER                              |LOCAL/REMOTE   2528
|
|  3 |        CLUSTER PUSHER ("_$NI_5")                 |             200000
|
|  4 |          PLAN BASED CLUSTER                      |LOCAL/REMOTE 200000
|
```

```
|  5 |            INDEX ACCESS ("PART", "PART_PK_INDEX")    |(66675) 66675
|
|  6 |        SELECT STATEMENT                                |    840
|
|  7 |          QUERY BLOCK ("$QB_IDX_2")                     |    840
|
|  8 |            HASH JOIN (INNER JOIN)                      |    840
|
|  9 |              PUSHER TABLE ACCESS ("_$NI_5" AS _A2)     | 200000
|
| 10 |              HASH JOIN INSTANT                         |    840
|
| 11 |                TABLE ACCESS ("LINEITEM" AS _A1)        |    840
|
================================================================================
==


     1  -  TARGET : LINEITEM.L_ORDERKEY, _$NI_5.P_PARTKEY
     2  -  SQL : SELECT /*+ KEEP_JOINED_TABLE
                            USE_HASH_IN( _A1, 396 )
                            FULL( _A2 )
                            FULL( _A1 )
                       */
                            "_A1"."L_ORDERKEY", "_A2"."P_PARTKEY"
                    FROM ( "SESSION_SCHEMA"."_$NI_5"@LOCAL AS "_A2"
```

```
                    INNER JOIN
                    "PUBLIC"."LINEITEM"@LOCAL AS "_A1"
                    ON "_A1"."L_PARTKEY" = "_A2"."P_PARTKEY"
                  ) ALIAS "_A3"
            WHERE "_A1"."L_SHIPDATE" = :_V0
      TARGET DOMAIN : G1(G1N1,G1N2) 840 rows,
                      G2(G2N1,G2N2) 824 rows,
                      G3(G3N1,G3N2) 864 rows
3  -  SQL : DECLARE INSTANT TABLE "SESSION_SCHEMA"."_$NI_5"
           ( "P_PARTKEY" NUMBER(10, 0) )
      COLUMN : PART.P_PARTKEY AS P_PARTKEY
      CLONED
      TARGET DOMAIN : G1(G1N1,G1N2) 200000 rows,
                      G2(G2N1,G2N2) 200000 rows,
                      G3(G3N1,G3N2) 200000 rows
4  -  SQL : SELECT /*+ INDEX( _A1, "PUBLIC"."PART_PK_INDEX" ) */
                  "_A1"."P_PARTKEY"
              FROM "PUBLIC"."PART"@LOCAL AS "_A1"
      TARGET DOMAIN : G1(G1N1,G1N2) 66675 rows,
                      G2(G2N1,G2N2) 66664 rows,
                      G3(G3N1,G3N2) 66661 rows
5  -  HASH SHARD ( # 3 )
      READ INDEX COLUMN : PART.P_PARTKEY
7  -  TARGET : _A1.L_ORDERKEY, _A2.P_PARTKEY
8  -  JOINED COLUMN : _A1.L_ORDERKEY, _A2.P_PARTKEY
```

```
 9  -  READ COLUMN : _A2.P_PARTKEY
10  -  HASH KEY : _A1.L_PARTKEY
       RECORD COLUMN : _A1.L_ORDERKEY
       READ KEY COLUMN : _A1.L_PARTKEY, _A1.L_ORDERKEY
         HASH FILTER : _A1.L_PARTKEY = _A2.P_PARTKEY
11  -  HASH SHARD ( # 3 )
       READ COLUMN : _A1.L_ORDERKEY, _A1.L_PARTKEY, _A1.L_SHIPDATE
         PHYSICAL FILTER : _A1.L_SHIPDATE = :_V0


<<<  end print plan
```

如上述execution plan所示在pusher table部署alias的sibling并执行了remote join

<group hints>

为Group by处理相关hint因此仅在有group by子句时有效

<group operation hints>

USE_GROUP_HASH

描述USE_GROUP_HASH hint时optimizer使用hash instant处理group by

通常处理group by时使用hash instant而从下级节点row相对于group by key column排列上来时不积累hash instant以对比row的group by key column value的方式处理group by

Optimizer执行cost estimation时不积累hash instant诱导下级节点使用index但统计信息不准确时

此方法的成本会更高因此这种情况可使用USE_GROUP_HASH hint

以下为使用USE_GROUP_HASH hint的示例对比使用hint前的execution plan和使用hint后的execution plan可理解USE_GROUP_HASH hint处理方式

```
\EXPLAIN PLAN
   SELECT
           c_custkey,
           count(o_orderkey) as c_count
     FROM  customer LEFT OUTER JOIN orders
           ON  c_custkey = o_custkey
           AND o_comment not like '%special%requests%'
           AND c_mktsegment = 'BUILDING'
 GROUP BY   c_custkey;



>>>  start print plan


< Execution Plan >
=========================================================================
|  IDX  |  NODE DESCRIPTION                                             |
-------------------------------------------------------------------------
|    0  |  SELECT STATEMENT                                             |
|    1  |    QUERY BLOCK ("$QB_IDX_2")                                  |
|    2  |      GROUP                                                |
|    3  |        HASH JOIN (LEFT OUTER JOIN)                            |
```

```
|    4  |           INDEX ACCESS ("CUSTOMER", "CUSTOMER_PK_INDEX")   |
|    5  |           HASH JOIN INSTANT                                 |
|    6  |             TABLE ACCESS ("ORDERS")                         |
=======================================================================


     1  -  TARGET : CUSTOMER.C_CUSTKEY, COUNT( ORDERS.O_ORDERKEY ) AS
C_COUNT
     2  -  GROUP KEY : CUSTOMER.C_CUSTKEY
           RECORD COLUMN : COUNT( ORDERS.O_ORDERKEY )
           READ KEY COLUMN : CUSTOMER.C_CUSTKEY
           READ RECORD COLUMN : COUNT( ORDERS.O_ORDERKEY )
     3  -  JOINED COLUMN : CUSTOMER.C_CUSTKEY, ORDERS.O_ORDERKEY
     4  -  READ COLUMN : ORDERS.O_ORDERKEY, ORDERS.O_CUSTKEY,
ORDERS.O_COMMENT
             LOGICAL FILTER : ORDERS.O_COMMENT NOT LIKE
'%special%requests%'
     5  -  HASH KEY : CUSTOMER.C_CUSTKEY
           READ KEY COLUMN : CUSTOMER.C_CUSTKEY
             HASH FILTER : CUSTOMER.C_CUSTKEY = ORDERS.O_CUSTKEY
     6  -  READ INDEX COLUMN : CUSTOMER.C_CUSTKEY


<<<  end print plan


\EXPLAIN PLAN
   SELECT  /*+ USE_GROUP_HASH */
```

```
           c_custkey,
           count(o_orderkey) as c_count
     FROM  customer LEFT OUTER JOIN orders
           ON  c_custkey = o_custkey
           AND o_comment not like '%special%requests%'
           AND c_mktsegment = 'BUILDING'
 GROUP BY   c_custkey;


>>>  start print plan


< Execution Plan >
=============================================================================
|  IDX  |  NODE DESCRIPTION                                                  |
-----------------------------------------------------------------------------
|    0  |  SELECT STATEMENT                                                  |
|    1  |    QUERY BLOCK ("$QB_IDX_2")                                       |
|    2  |      GROUP HASH INSTANT                                            |
|    3  |        HASH JOIN (LEFT OUTER JOIN)                                 |
|    4  |          TABLE ACCESS ("CUSTOMER")                                 |
|    5  |          HASH JOIN INSTANT                                         |
|    6  |            TABLE ACCESS ("ORDERS")                                 |
=============================================================================


     1  -  TARGET : CUSTOMER.C_CUSTKEY, COUNT( ORDERS.O_ORDERKEY ) AS
C_COUNT
```

```
     2  -  GROUP KEY : CUSTOMER.C_CUSTKEY
           RECORD COLUMN : COUNT( ORDERS.O_ORDERKEY )
           READ KEY COLUMN : CUSTOMER.C_CUSTKEY
           READ RECORD COLUMN : COUNT( ORDERS.O_ORDERKEY )
     3  -  JOINED COLUMN : CUSTOMER.C_CUSTKEY, ORDERS.O_ORDERKEY
     4  -  READ COLUMN : CUSTOMER.C_CUSTKEY, CUSTOMER.C_MKTSEGMENT
     5  -  HASH KEY : ORDERS.O_CUSTKEY
           RECORD COLUMN : ORDERS.O_ORDERKEY
           READ KEY COLUMN : ORDERS.O_CUSTKEY, ORDERS.O_ORDERKEY
             HASH FILTER : ORDERS.O_CUSTKEY = CUSTOMER.C_CUSTKEY
             LOGICAL FILTER : {CUSTOMER.C_MKTSEGMENT} = 'BUILDING'
     6  -  READ COLUMN : ORDERS.O_ORDERKEY, ORDERS.O_CUSTKEY,
ORDERS.O_COMMENT
             LOGICAL FILTER : ORDERS.O_COMMENT NOT LIKE
'%special%requests%'


<<<  end print plan
```

USE_GROUP_HASH(hash_bucket_count)

USE_GROUP_HASH(hash_bucket_count) hint与USE_GROUP_HASH hint相同区别在于它可指定hash bucket count

统计信息不准确时由于为了GROUP BY而生成的hash instant的hash bucket count过多或过少而降低性能此时可使用hint指定hash bucket count防止性能下降

以下为使用USE_GROUP_HASH(hash_bucket_count) hint的示例

```
\EXPLAIN PLAN VERBOSE
   SELECT  /*+ USE_GROUP_HASH(15000) */
           c_custkey,
           count(o_orderkey) as c_count
     FROM  customer LEFT OUTER JOIN orders
           ON  c_custkey = o_custkey
           AND o_comment not like '%special%requests%'
           AND c_mktsegment = 'BUILDING'
 GROUP BY   c_custkey;


>>>  start print plan


< Execution Plan >
================================================================================
==
|IDX|  NODE DESCRIPTION                                    |   ROWS  | 省
略 |
--------------------------------------------------------------------------------
--
| 0 |  SELECT STATEMENT                                    |  150000 | ...
|
| 1 |    QUERY BLOCK ("$QB_IDX_2")                         |  150000 | ...
|
| 2 |      GROUP HASH INSTANT                              |  150000 | ...
|
```

```
| 3 |        HASH JOIN (INVERTED LEFT OUTER JOIN)          |  430506 | ...
|
| 4 |          TABLE ACCESS ("ORDERS")                     | 1483918 | ...
|
| 5 |          HASH JOIN INSTANT                           |  430506 | ...
|
| 6 |            TABLE ACCESS ("CUSTOMER")                 |  150000 | ...
|
================================================================================
==

     1  -  TARGET : CUSTOMER.C_CUSTKEY, COUNT( ORDERS.O_ORDERKEY ) AS
C_COUNT
     2  -  GROUP KEY : CUSTOMER.C_CUSTKEY
           RECORD COLUMN : COUNT( ORDERS.O_ORDERKEY )
           READ KEY COLUMN : CUSTOMER.C_CUSTKEY
           READ RECORD COLUMN : COUNT( ORDERS.O_ORDERKEY )
           HASH BUCKET COUNT : 15000
     3  -  JOINED COLUMN : CUSTOMER.C_CUSTKEY, ORDERS.O_ORDERKEY
     4  -  READ COLUMN : ORDERS.O_ORDERKEY, ORDERS.O_CUSTKEY,
ORDERS.O_COMMENT
             LOGICAL FILTER : ORDERS.O_COMMENT NOT LIKE
'%special%requests%'
     5  -  HASH KEY : CUSTOMER.C_CUSTKEY
           RECORD COLUMN : CUSTOMER.C_MKTSEGMENT
```

```
            READ KEY COLUMN : CUSTOMER.C_CUSTKEY
            READ RECORD COLUMN : CUSTOMER.C_MKTSEGMENT
              HASH FILTER : CUSTOMER.C_CUSTKEY = ORDERS.O_CUSTKEY
              PHYSICAL FILTER : CUSTOMER.C_MKTSEGMENT = 'BUILDING'
            HASH BUCKET COUNT : 150000
      6   -  READ COLUMN : CUSTOMER.C_CUSTKEY, CUSTOMER.C_MKTSEGMENT


<<<  end print plan
```

上述示例中hash bucket count按照GROUP HASH INSTANT的预计output row数量指定

NO_USE_GROUP_HASH

描述NO_USE_GROUP_HASH hint时optimizer不使用hash instant进行group by处理

即在下级plan中当中间结果相对group key进行排序并上升时使用它否则使用sort instant进行group by处理

```
\EXPLAIN PLAN
   SELECT  /*+ NO_USE_GROUP_HASH */
           c_custkey,
           count(o_orderkey) as c_count
     FROM  customer LEFT OUTER JOIN orders
           ON  c_custkey = o_custkey
           AND o_comment not like '%special%requests%'
           AND c_mktsegment = 'BUILDING'
 GROUP BY   c_custkey;
```

```
>>>  start print plan


< Execution Plan >

=============================================================================
|  IDX  |  NODE DESCRIPTION                                                  |
-----------------------------------------------------------------------------
|    0  |  SELECT STATEMENT                                                  |
|    1  |    QUERY BLOCK ("$QB_IDX_2")                                       |
|  2    |      GROUP SORT INSTANT                                            |
|    3  |        HASH JOIN (INVERTED LEFT OUTER JOIN)                        |
|    4  |          TABLE ACCESS ("ORDERS")                                   |
|    5  |          HASH JOIN INSTANT                                         |
|    6  |            TABLE ACCESS ("CUSTOMER")                               |
=============================================================================


     1  -  TARGET : CUSTOMER.C_CUSTKEY, COUNT( ORDERS.O_ORDERKEY ) AS
C_COUNT
     2  -  GROUP KEY : CUSTOMER.C_CUSTKEY
           RECORD COLUMN : COUNT( ORDERS.O_ORDERKEY )
           READ KEY COLUMN : CUSTOMER.C_CUSTKEY
           READ RECORD COLUMN : COUNT( ORDERS.O_ORDERKEY )
     3  -  JOINED COLUMN : CUSTOMER.C_CUSTKEY, ORDERS.O_ORDERKEY
     4  -  READ COLUMN : ORDERS.O_ORDERKEY, ORDERS.O_CUSTKEY,
ORDERS.O_COMMENT
```

```
            LOGICAL FILTER : ORDERS.O_COMMENT NOT LIKE
'%special%requests%'
     5  -  HASH KEY : CUSTOMER.C_CUSTKEY
           RECORD COLUMN : CUSTOMER.C_MKTSEGMENT
           READ KEY COLUMN : CUSTOMER.C_CUSTKEY
           READ RECORD COLUMN : CUSTOMER.C_MKTSEGMENT
             HASH FILTER : CUSTOMER.C_CUSTKEY = ORDERS.O_CUSTKEY
             PHYSICAL FILTER : CUSTOMER.C_MKTSEGMENT = 'BUILDING'
     6  -  READ COLUMN : CUSTOMER.C_CUSTKEY, CUSTOMER.C_MKTSEGMENT


<<<  end print plan
```

USE_GROUP_SORT

描述USE_GROUP_SORT hint时optimizer使用sort instant进行group by处理

```
\EXPLAIN PLAN
   SELECT  /*+ USE_GROUP_SORT */
           c_custkey,
           count(o_orderkey) as c_count
     FROM  customer LEFT OUTER JOIN orders
           ON  c_custkey = o_custkey
           AND o_comment not like '%special%requests%'
           AND c_mktsegment = 'BUILDING'
 GROUP BY   c_custkey;
```

```
>>>  start print plan


< Execution Plan >

=============================================================================

|  IDX  |  NODE DESCRIPTION                                                  |

-----------------------------------------------------------------------------

|    0  |  SELECT STATEMENT                                                  |

|    1  |    QUERY BLOCK ("$QB_IDX_2")                                       |

|  2    |      GROUP SORT INSTANT                                          |

|    3  |        HASH JOIN (INVERTED LEFT OUTER JOIN)                        |

|    4  |          TABLE ACCESS ("ORDERS")                                   |

|    5  |          HASH JOIN INSTANT                                         |

|    6  |            TABLE ACCESS ("CUSTOMER")                               |

=============================================================================


     1  -  TARGET : CUSTOMER.C_CUSTKEY, COUNT( ORDERS.O_ORDERKEY ) AS
C_COUNT

     2  -  GROUP KEY : CUSTOMER.C_CUSTKEY

           RECORD COLUMN : COUNT( ORDERS.O_ORDERKEY )

           READ KEY COLUMN : CUSTOMER.C_CUSTKEY

           READ RECORD COLUMN : COUNT( ORDERS.O_ORDERKEY )

     3  -  JOINED COLUMN : CUSTOMER.C_CUSTKEY, ORDERS.O_ORDERKEY

     4  -  READ COLUMN : ORDERS.O_ORDERKEY, ORDERS.O_CUSTKEY,
ORDERS.O_COMMENT

            LOGICAL FILTER : ORDERS.O_COMMENT NOT LIKE
```

```
'%special%requests%'
     5  -  HASH KEY : CUSTOMER.C_CUSTKEY
           RECORD COLUMN : CUSTOMER.C_MKTSEGMENT
           READ KEY COLUMN : CUSTOMER.C_CUSTKEY
           READ RECORD COLUMN : CUSTOMER.C_MKTSEGMENT
             HASH FILTER : CUSTOMER.C_CUSTKEY = ORDERS.O_CUSTKEY
             PHYSICAL FILTER : CUSTOMER.C_MKTSEGMENT = 'BUILDING'
     6  -  READ COLUMN : CUSTOMER.C_CUSTKEY, CUSTOMER.C_MKTSEGMENT


<<<  end print plan
```

NO_USE_GROUP_SORT

描述NO_USE_GROUP_SORT hint时optimizer不使用sort instant进行group by处理

```
\EXPLAIN PLAN
   SELECT  /*+ NO_USE_GROUP_SORT */
           c_custkey,
           count(o_orderkey) as c_count
     FROM  customer LEFT OUTER JOIN orders
           ON  c_custkey = o_custkey
           AND o_comment not like '%special%requests%'
           AND c_mktsegment = 'BUILDING'
 GROUP BY   c_custkey;


>>>  start print plan
```

```
< Execution Plan >
=============================================================================
|  IDX  |  NODE DESCRIPTION                                                  |
-----------------------------------------------------------------------------
|    0  |  SELECT STATEMENT                                                  |
|    1  |    QUERY BLOCK ("$QB_IDX_2")                                       |
|   2   |    GROUP HASH INSTANT                                              |
|    3  |        HASH JOIN (INVERTED LEFT OUTER JOIN)                        |
|    4  |          TABLE ACCESS ("ORDERS")                                   |
|    5  |          HASH JOIN INSTANT                                         |
|    6  |            TABLE ACCESS ("CUSTOMER")                               |
=============================================================================

     1  -  TARGET : CUSTOMER.C_CUSTKEY, COUNT( ORDERS.O_ORDERKEY ) AS
C_COUNT
     2  -  GROUP KEY : CUSTOMER.C_CUSTKEY
           RECORD COLUMN : COUNT( ORDERS.O_ORDERKEY )
           READ KEY COLUMN : CUSTOMER.C_CUSTKEY
           READ RECORD COLUMN : COUNT( ORDERS.O_ORDERKEY )
     3  -  JOINED COLUMN : CUSTOMER.C_CUSTKEY, ORDERS.O_ORDERKEY
     4  -  READ COLUMN : ORDERS.O_ORDERKEY, ORDERS.O_CUSTKEY,
ORDERS.O_COMMENT
             LOGICAL FILTER : ORDERS.O_COMMENT NOT LIKE
'%special%requests%'
```

```
    5  -  HASH KEY : CUSTOMER.C_CUSTKEY
          RECORD COLUMN : CUSTOMER.C_MKTSEGMENT
          READ KEY COLUMN : CUSTOMER.C_CUSTKEY
          READ RECORD COLUMN : CUSTOMER.C_MKTSEGMENT
            HASH FILTER : CUSTOMER.C_CUSTKEY = ORDERS.O_CUSTKEY
            PHYSICAL FILTER : CUSTOMER.C_MKTSEGMENT = 'BUILDING'
    6  -  READ COLUMN : CUSTOMER.C_CUSTKEY, CUSTOMER.C_MKTSEGMENT


<<<  end print plan
```

<group driver hints>

是在集群系统中执行group by子句时使用的hint在单机版被忽略

LOCAL_GROUP

在当前服务器执行group by

以下为使用LOCAL_GROUP hint的示例

```
\EXPLAIN PLAN
SELECT /*+ LOCAL_GROUP */
       c_custkey, COUNT(o_orderkey)
  FROM customer, orders
 WHERE c_custkey = o_custkey
   AND o_orderdate >= date '1993-07-01'
   AND c_nationkey = 1
```

```
   AND o_orderstatus = 'F'
   AND o_orderpriority = '2-HIGH'
GROUP BY c_custkey;


>>>  start print plan


< Execution Plan >
================================================================================
==
|IDX|  NODE DESCRIPTION                                                  | ROWS
|
--------------------------------------------------------------------------------
--
| 0 |  SELECT STATEMENT                                                  | 2094
|
| 1 |    QUERY BLOCK ("$QB_IDX_2")                                       | 2094
|
| 2 |      GROUP HASH INSTANT                                            | 2094
|
| 3 |        PLAN BASED CLUSTER                          |LOCAL/REMOTE 3049
|
| 4 |          NESTED JOIN (INNER JOIN)                                  | 1004
|
| 5 |            INDEX ACCESS ("CUSTOMER", "CUSTOMER_NATIONKEY_FK")      | 5975
|
```

```
| 6 |            INDEX ACCESS ("ORDERS", "ORDERS_CUSTKEY_FK")       | 1004
|
=============================================================================
==


     1  -  TARGET : CUSTOMER.C_CUSTKEY, COUNT( ORDERS.O_ORDERKEY )
     2  -  GROUP KEY : CUSTOMER.C_CUSTKEY
           RECORD COLUMN : COUNT( ORDERS.O_ORDERKEY )
           READ KEY COLUMN : CUSTOMER.C_CUSTKEY
           READ RECORD COLUMN : COUNT( ORDERS.O_ORDERKEY )
     3  -  SQL : SELECT /*+ KEEP_JOINED_TABLE
                            USE_NL_IN( _A1 )
                            INDEX( _A2, "PUBLIC"."CUSTOMER_NATIONKEY_FK" )
                            INDEX( _A1, "PUBLIC"."ORDERS_CUSTKEY_FK" )
                         */
                         "_A2"."C_CUSTKEY", "_A1"."O_ORDERKEY"
                    FROM ( "PUBLIC"."CUSTOMER"@LOCAL AS "_A2"
                             INNER JOIN
                             "PUBLIC"."ORDERS"@LOCAL AS "_A1" ON true
                         ) ALIAS "_A3"
                   WHERE "_A2"."C_NATIONKEY" = :_V0
                     AND "_A1"."O_CUSTKEY" = "_A2"."C_CUSTKEY"
                     AND "_A1"."O_ORDERSTATUS" = :_V1
                     AND "_A1"."O_ORDERDATE" >= :_V2
                     AND "_A1"."O_ORDERPRIORITY" = :_V3
```

```
           TARGET DOMAIN : G1(G1N1,G1N2) 1004 rows,
                           G2(G2N1,G2N2) 995 rows,
                           G3(G3N1,G3N2) 1050 rows
     4  -  JOINED COLUMN : CUSTOMER.C_CUSTKEY, ORDERS.O_ORDERKEY
     5  -  CLONED
           READ INDEX COLUMN : CUSTOMER.C_NATIONKEY
           READ TABLE COLUMN : CUSTOMER.C_CUSTKEY
             MIN RANGE : CUSTOMER.C_NATIONKEY = 1
             MAX RANGE : CUSTOMER.C_NATIONKEY = 1
     6  -  HASH SHARD ( # 3 )
           READ INDEX COLUMN : ORDERS.O_CUSTKEY
           READ TABLE COLUMN : ORDERS.O_ORDERKEY, ORDERS.O_ORDERSTATUS,
ORDERS.O_ORDERDATE, ORDERS.O_ORDERPRIORITY
             MIN RANGE : ORDERS.O_CUSTKEY = {CUSTOMER.C_CUSTKEY}
             MAX RANGE : ORDERS.O_CUSTKEY = {CUSTOMER.C_CUSTKEY}
             PHYSICAL TABLE FILTER : ORDERS.O_ORDERSTATUS = 'F' AND
ORDERS.O_ORDERDATE >= DATE'1993-07-01' AND ORDERS.O_ORDERPRIORITY = '2-
HIGH'

<<<  end print plan
```

如上述execution plan从G1G2G3获取join的所有结果后在local处理了group by

通常在sharded table情况可进行remote group时最好以remote group处理由于可并行处理grouping并减少接收的中间结果

但由于group by要处理的对象row数量少而无法获得并行处理效果时以local处理group by会更好

REMOTE_GROUP

在各个group的服务器执行group by

以下为使用REMOTE_GROUP hint的示例

```
\EXPLAIN PLAN
SELECT /*+ REMOTE_GROUP */
       c_custkey, COUNT(o_orderkey)
  FROM customer, orders
 WHERE c_custkey = o_custkey
GROUP BY c_custkey;


>>>  start print plan


< Execution Plan >
================================================================================
==
| IDX |  NODE DESCRIPTION                                              |   ROWS
|
--------------------------------------------------------------------------------
--
| 0 |  SELECT STATEMENT                                                | 99996
|
| 1 |    QUERY BLOCK ("$QB_IDX_2")                                     | 99996
```

```
|
| 2 |      SINGLE CLUSTER                                | LOCAL/REMOTE 99996
|
| 3 |        SELECT STATEMENT                                          |  98218
|
| 4 |          QUERY BLOCK ("$QB_IDX_2")                               |  98218
|
| 5 |            GROUP HASH INSTANT                                    |  98218
|
| 6 |              HASH JOIN (INNER JOIN)                              | 500004
|
| 7 |                TABLE ACCESS ("ORDERS" AS _A2)                    | 500004
|
| 8 |                HASH JOIN INSTANT                                 | 500004
|
| 9 |                  INDEX ACCESS ("CUSTOMER" AS _A1")               | 150000
|
==========================================================================
==

     1  -  TARGET : CUSTOMER.C_CUSTKEY, COUNT( ORDERS.O_ORDERKEY )
     2  -  SQL : SELECT /*+ USE_GROUP_HASH(150000)
                            KEEP_JOINED_TABLE
                            USE_HASH_IN( _A1, 150000 )
                            FULL( _A2 )
```

```
                        INDEX( _A1, "PUBLIC"."CUSTOMER_PK_INDEX" )
                    */
                    "_A1"."C_CUSTKEY", COUNT( "_A2"."O_ORDERKEY" )
                FROM ( "PUBLIC"."ORDERS"@LOCAL AS "_A2"
                        INNER JOIN
                       "PUBLIC"."CUSTOMER"@LOCAL AS "_A1"
                       ON "_A1"."C_CUSTKEY" = "_A2"."O_CUSTKEY"
                    ) ALIAS "_A3"
              GROUP BY "_A1"."C_CUSTKEY"
        TARGET DOMAIN : G1(G1N1,G1N2) 98218 rows,
                        G2(G2N1,G2N2) 98174 rows,
                        G3(G3N1,G3N2) 98138 rows
        RE-GROUPING
          GROUP KEY : CUSTOMER.C_CUSTKEY
          AGGREGATION : SUM( COUNT( ORDERS.O_ORDERKEY ) )
4   -   TARGET : _A1.C_CUSTKEY, COUNT( _A2.O_ORDERKEY )
5   -   GROUP KEY : _A1.C_CUSTKEY
        RECORD COLUMN : COUNT( _A2.O_ORDERKEY )
        READ KEY COLUMN : _A1.C_CUSTKEY
        READ RECORD COLUMN : COUNT( _A2.O_ORDERKEY )
6   -   JOINED COLUMN : _A1.C_CUSTKEY, _A2.O_ORDERKEY
7   -   HASH SHARD ( # 3 )
        READ COLUMN : _A2.O_ORDERKEY, _A2.O_CUSTKEY
8   -   HASH KEY : _A1.C_CUSTKEY
        READ KEY COLUMN : _A1.C_CUSTKEY
```

```
            HASH FILTER : _A1.C_CUSTKEY = _A2.O_CUSTKEY
          FETCH ONE ROW
    9   -  CLONED
          READ INDEX COLUMN : _A1.C_CUSTKEY


<<<  end print plan
```

如上述execution plan所示在各个服务器处理了group by在各个服务器处理join的结果grouping约50万条后输出了10万条的中间结果这样可以并行处理grouping也减少网络接收的中间结果

如果以local处理上述示例则需要获取150万条join结果后对150万条处理group by这样需要获取150万条的网络成本和对150万条一次性处理grouping的成本

<group cluster regrouping hints>

是在集群系统中执行group by子句时使用的hint在单机版被忽略

MERGE_GROUP

满足如下条件时可使用MERGE_GROUP hint

- 有group by子句
- 可以以remote group执行
- 在group节点的下级节点中中间结果以保证group by key column的order的状态上升

使用MERGE_GROUP hint时driver获取中间结果后也可保证order

以下为使用MERGE_GROUP hint的示例

```
\EXPLAIN PLAN
SELECT /*+ MERGE_GROUP */
       o_custkey
  FROM orders
 WHERE o_custkey > 0
GROUP BY o_custkey;


O_CUSTKEY
---------
        1
        2
        4
        5
        7
        8
       10
       11
      ...
   149992
   149993
   149995
   149996
   149998
   149999
```

```
99996 rows selected.



>>>  start print plan


< Execution Plan >

================================================================================
==
|IDX|  NODE DESCRIPTION                                |                ROWS
|
--------------------------------------------------------------------------------
--
| 0 |  SELECT STATEMENT                                |               99996
|
| 1 |    QUERY BLOCK ("$QB_IDX_2")                     |               99996
|
| 2 |      MULTIPLE CLUSTER                            | LOCAL/REMOTE 99996
|
| 3 |        SELECT STATEMENT                          |               98218
|
| 4 |          QUERY BLOCK ("$QB_IDX_2")               |               98218
|
| 5 |            GROUP                                 |               98218
|
| 6 |              INDEX ACCESS ("ORDERS" AS _A1)      |              500004
```

```
|
================================================================================
==


     1  -  TARGET : ORDERS.O_CUSTKEY
     2  -  SQL : SELECT /*+ INDEX( _A1, "PUBLIC"."ORDERS_CUSTKEY_FK" ) */
                        "_A1"."O_CUSTKEY"
                   FROM "PUBLIC"."ORDERS"@LOCAL AS "_A1"
                  WHERE "_A1"."O_CUSTKEY" > :_V0
               GROUP BY "_A1"."O_CUSTKEY"
               ORDER BY "_A1"."O_CUSTKEY" ASC NULLS LAST
           TARGET DOMAIN : G1(G1N1,G1N2) 98218 rows,
                           G2(G2N1,G2N2) 98174 rows,
                           G3(G3N1,G3N2) 98138 rows
           MERGE GROUPING
             SORT KEY : ORDERS.O_CUSTKEY
             GROUP KEY : ORDERS.O_CUSTKEY
     4  -  TARGET : _A1.O_CUSTKEY
     5  -  GROUP KEY : _A1.O_CUSTKEY
     6  -  HASH SHARD ( # 3 )
           READ INDEX COLUMN : _A1.O_CUSTKEY
             MIN RANGE : _A1.O_CUSTKEY > :_V0
             MAX RANGE : _A1.O_CUSTKEY IS NOT NULL


<<<  end print plan
```

如上述执行结果按照group key col顺序ordering并输出

<distinct hints>

是distinct处理相关hint因此仅在有distinct的情况有效

<distinct operation hints>

USE_DISTINCT_HASH

描述USE_DISTINCT_HASH hint时optimizer使用hash instant处理distinct

通常处理distinct时使用hash instant而从下级节点row对distinct key column排列上来时不积累hash instant以对比row的distinct key column value的方式处理distinct

Optimizer执行cost estimation时不积累hash instant诱导下级节点使用index但统计信息不准确时此方法的成本会更高因此这种情况可使用USE_DISTINCT_HASH hint

以下为使用USE_DISTINCT_HASH hint的示例对比使用hint前的execution plan和使用hint后的execution plan可理解USE_DISTINCT_HASH hint处理方式

```
EXPLAIN PLAN
   SELECT
          DISTINCT
          c_nationkey
     FROM  customer
    WHERE  c_comment not like '%special%requests%'
      AND  c_nationkey > 5;
```

```
>>>  start print plan


< Execution Plan >

=============================================================================

|  IDX  |  NODE DESCRIPTION                                                  |

-----------------------------------------------------------------------------

|    0  |  SELECT STATEMENT                                                  |

|    1  |    QUERY BLOCK ("$QB_IDX_2")                                       |

|    2  |      GROUP                                                         |

|    3  |        INDEX ACCESS ("CUSTOMER", "CUSTOMER_NATIONKEY_FK")          |

=============================================================================


     1  -  TARGET : CUSTOMER.C_NATIONKEY

     2  -  GROUP KEY : CUSTOMER.C_NATIONKEY

     3  -  READ INDEX COLUMN : CUSTOMER.C_NATIONKEY

           READ TABLE COLUMN : CUSTOMER.C_COMMENT

             MIN RANGE : CUSTOMER.C_NATIONKEY > 5

             MAX RANGE : CUSTOMER.C_NATIONKEY IS NOT NULL

             LOGICAL TABLE FILTER : CUSTOMER.C_COMMENT NOT LIKE
'%special%requests%'


<<<  end print plan


\EXPLAIN PLAN

   SELECT  /*+ USE_DISTINCT_HASH */
```

```
           DISTINCT
           c_nationkey
     FROM  customer
    WHERE  c_comment not like '%special%requests%'
      AND  c_nationkey > 5;


>>>  start print plan


< Execution Plan >
=========================================================================
|  IDX  |  NODE DESCRIPTION                                             |
-------------------------------------------------------------------------
|    0  |  SELECT STATEMENT                                             |
|    1  |    QUERY BLOCK ("$QB_IDX_2")                                  |
|    2  |      GROUP HASH INSTANT                                       |
|    3  |        INDEX ACCESS ("CUSTOMER", "CUSTOMER_NATIONKEY_FK")     |
=========================================================================


     1  -  TARGET : CUSTOMER.C_NATIONKEY
     2  -  GROUP KEY : CUSTOMER.C_NATIONKEY
           READ KEY COLUMN : CUSTOMER.C_NATIONKEY
     3  -  READ INDEX COLUMN : CUSTOMER.C_NATIONKEY
           READ TABLE COLUMN : CUSTOMER.C_COMMENT
             MIN RANGE : CUSTOMER.C_NATIONKEY > 5
             MAX RANGE : CUSTOMER.C_NATIONKEY IS NOT NULL
```

```
            LOGICAL TABLE FILTER : CUSTOMER.C_COMMENT NOT LIKE
'%special%requests%'


<<<  end print plan
```

USE_DISTINCT_HASH(hash__bucket_count)

USE_DISTINCT_HASH(hash__bucket_count) hint与USE_DISTINCT_HASH hint相同区别在于它可指定hash bucket count

统计信息不准确时由于为了DISTINCT而生成的hash instant的hash bucket count过多或过少而降低性能此时可使用hint指定hash bucket count防止性能下降

以下为使用USE_DISTINCT_HASH hint(hash_bucket_count)的示例

```
\EXPLAIN PLAN VERBOSE
   SELECT  /*+ USE_DISTINCT_HASH(19) */
           DISTINCT
           c_nationkey
     FROM  customer
    WHERE  c_comment not like '%special%requests%'
      AND  c_nationkey > 5;



>>>  start print plan


< Execution Plan >
```

```
================================================================================
==
|  IDX  |  NODE DESCRIPTION                                    |  ROWS | 省
略 |
--------------------------------------------------------------------------------
--
| 0 |SELECT STATEMENT                                        |    19 | ...
|
| 1 |  QUERY BLOCK ("$QB_IDX_2")                             |    19 | ...
|
| 2 |    GROUP HASH INSTANT                                  |    19 | ...
|
| 3 |      INDEX ACCESS ("CUSTOMER", "CUSTOMER_NATIONKEY_FK")|111562 | ...
|
================================================================================
==


     1  -  TARGET : CUSTOMER.C_NATIONKEY
     2  -  GROUP KEY : CUSTOMER.C_NATIONKEY
           READ KEY COLUMN : CUSTOMER.C_NATIONKEY
           HASH BUCKET COUNT : 19
     3  -  READ INDEX COLUMN : CUSTOMER.C_NATIONKEY
           READ TABLE COLUMN : CUSTOMER.C_COMMENT
             MIN RANGE : CUSTOMER.C_NATIONKEY > 5
             MAX RANGE : CUSTOMER.C_NATIONKEY IS NOT NULL
```

```
            LOGICAL TABLE FILTER : CUSTOMER.C_COMMENT NOT LIKE
'%special%requests%'


<<<  end print plan
```

上述示例中hash bucket count按照GROUP HASH INSTANT的预计output row数量指定

NO_USE_DISTINCT_HASH

描述NO_USE_DISTINCT_HASH hint时optimizer不使用hash instant并处理distinct

即在下级plan中中间结果相对于distinct key进行排序上升时使用它否则使用sort instant处理distinct

```
\EXPLAIN PLAN
   SELECT  /*+ NO_USE_DISTINCT_HASH */
           DISTINCT
           c_nationkey
     FROM  customer
    WHERE  c_comment not like '%special%requests%'
      AND  c_nationkey > 5;


>>>  start print plan


< Execution Plan >
=========================================================================
|  IDX  |  NODE DESCRIPTION                                             |
```

```
-------------------------------------------------------------------------
|    0  |  SELECT STATEMENT                                             |
|    1  |    QUERY BLOCK ("$QB_IDX_2")                                  |
|    2  |      GROUP                                                    |
|    3  |        INDEX ACCESS ("CUSTOMER", "CUSTOMER_NATIONKEY_FK")    |
=========================================================================


     1  -  TARGET : CUSTOMER.C_NATIONKEY
     2  -  GROUP KEY : CUSTOMER.C_NATIONKEY
     3  -  READ INDEX COLUMN : CUSTOMER.C_NATIONKEY
           READ TABLE COLUMN : CUSTOMER.C_COMMENT
             MIN RANGE : CUSTOMER.C_NATIONKEY > 5
             MAX RANGE : CUSTOMER.C_NATIONKEY IS NOT NULL
             LOGICAL TABLE FILTER : CUSTOMER.C_COMMENT NOT LIKE
'%special%requests%'


<<<  end print plan
```

USE_DISTINCT_SORT

描述USE_DISTINCT_SORT hint时optimizer使用sort instant处理distinct

```
\EXPLAIN PLAN
   SELECT  /*+ USE_DISTINCT_SORT */
           DISTINCT
           c_nationkey
```

```
     FROM  customer
    WHERE  c_comment not like '%special%requests%'
      AND  c_nationkey > 5;


>>>  start print plan


< Execution Plan >
=============================================================================
|  IDX  |  NODE DESCRIPTION                                                  |
-----------------------------------------------------------------------------
|    0  |  SELECT STATEMENT                                                  |
|    1  |    QUERY BLOCK ("$QB_IDX_2")                                       |
|    2  |      GROUP SORT INSTANT                                            |
|    3  |        TABLE ACCESS ("CUSTOMER")                                   |
=============================================================================


     1  -  TARGET : CUSTOMER.C_NATIONKEY
     2  -  GROUP KEY : CUSTOMER.C_NATIONKEY
           READ KEY COLUMN : CUSTOMER.C_NATIONKEY
     3  -  READ COLUMN : CUSTOMER.C_NATIONKEY, CUSTOMER.C_COMMENT
             PHYSICAL FILTER : CUSTOMER.C_NATIONKEY > 5
             LOGICAL FILTER : CUSTOMER.C_COMMENT NOT LIKE
'%special%requests%'


<<<  end print plan
```

NO_USE_DISTINCT_SORT

描述NO_USE_DISTINCT_SORT hint时optimizer不使用sort instant并处理distinct

```
>>>  start print plan


< Execution Plan >

=============================================================================

|  IDX  |  NODE DESCRIPTION                                                  |

-----------------------------------------------------------------------------

|    0  |  SELECT STATEMENT                                                  |

|    1  |    QUERY BLOCK ("$QB_IDX_2")                                       |

|    2  |      GROUP HASH INSTANT                                            |

|    3  |        TABLE ACCESS ("CUSTOMER")                                   |

=============================================================================


     1  -  TARGET : CUSTOMER.C_NATIONKEY

     2  -  GROUP KEY : CUSTOMER.C_NATIONKEY

           READ KEY COLUMN : CUSTOMER.C_NATIONKEY

     3  -  READ COLUMN : CUSTOMER.C_NATIONKEY, CUSTOMER.C_COMMENT

             PHYSICAL FILTER : CUSTOMER.C_NATIONKEY > 5

             LOGICAL FILTER : CUSTOMER.C_COMMENT NOT LIKE

'%special%requests%'


<<<  end print plan
```

<distinct driver hints>

是在集群系统中执行distinct子句时使用的hint在单机版被忽略

LOCAL_DISTINCT

在当前服务器执行distinct

以下为使用LOCAL_DISTINCT hint的示例

```
\EXPLAIN PLAN
SELECT /*+ LOCAL_DISTINCT */
       DISTINCT o_custkey
  FROM orders
 WHERE o_orderdate = date '1995-03-15'
   AND o_orderstatus = 'F'
   AND o_orderpriority = '2-HIGH';



>>>  start print plan


< Execution Plan >
================================================================================
==
|  IDX  |  NODE DESCRIPTION                                |            ROWS
|
--------------------------------------------------------------------------------
```

```
--
|    0  |  SELECT STATEMENT                                |               34
|
|    1  |    QUERY BLOCK ("$QB_IDX_2")                     |               34
|
|    2  |      GROUP HASH INSTANT                          |               34
|
|    3  |        PLAN BASED CLUSTER                        | LOCAL/REMOTE 34
|
|    4  |          TABLE ACCESS ("ORDERS")                 |               13
|
================================================================================
==


     1  -  TARGET : ORDERS.O_CUSTKEY
     2  -  GROUP KEY : ORDERS.O_CUSTKEY
           READ KEY COLUMN : ORDERS.O_CUSTKEY
     3  -  SQL : SELECT /*+ FULL( _A1 ) */
                         "_A1"."O_CUSTKEY"
                   FROM "PUBLIC"."ORDERS"@LOCAL AS "_A1"
                  WHERE "_A1"."O_ORDERSTATUS" = :_V0
                    AND "_A1"."O_ORDERDATE" = :_V1
                    AND "_A1"."O_ORDERPRIORITY" = :_V2
           TARGET DOMAIN : G1(G1N1,G1N2) 13 rows,
                           G2(G2N1,G2N2) 11 rows,
```

```
                          G3(G3N1,G3N2) 10 rows
     4  -  HASH SHARD ( # 3 )
           READ COLUMN : ORDERS.O_CUSTKEY, ORDERS.O_ORDERSTATUS,
ORDERS.O_ORDERDATE, ORDERS.O_ORDERPRIORITY
             PHYSICAL FILTER : ORDERS.O_ORDERSTATUS = 'F' AND
ORDERS.O_ORDERDATE = DATE'1995-03-15' AND ORDERS.O_ORDERPRIORITY = '2-
HIGH'

<<<  end print plan
```

如上述execution plan所示将orders中满足条件的中间结果传输至local并生成了用于distinct的GROUP HASH INSTANT

REMOTE_DISTINCT

在各个group的服务器执行distinct

以下为使用REMOTE_DISTINCT hint的示例

```
\EXPLAIN PLAN
SELECT /*+ REMOTE_DISTINCT */
       DISTINCT o_orderstatus
  FROM orders
 WHERE o_orderdate = date '1995-03-15';

>>>  start print plan
```

```
< Execution Plan >
================================================================================
==
|  IDX  |  NODE DESCRIPTION                                |          ROWS
|
--------------------------------------------------------------------------------
--
|    0  |  SELECT STATEMENT                                |             3
|
|    1  |    QUERY BLOCK ("$QB_IDX_2")                     |             3
|
|    2  |      SINGLE CLUSTER                              |LOCAL/REMOTE 3
|
|    3  |        SELECT STATEMENT                          |             3
|
|    4  |          QUERY BLOCK ("$QB_IDX_2")               |             3
|
|    5  |            GROUP HASH INSTANT                    |             3
|
|    6  |              TABLE ACCESS ("ORDERS" AS _A1)      |           221
|
================================================================================
==

     1  -  TARGET : ORDERS.O_ORDERSTATUS
```

```
    2   -   SQL : SELECT /*+ USE_DISTINCT_HASH(3)
                              FULL( _A1 )
                            */
                            DISTINCT "_A1"."O_ORDERSTATUS"
                          FROM "PUBLIC"."ORDERS"@LOCAL AS "_A1"
                         WHERE "_A1"."O_ORDERDATE" = :_V0
            TARGET DOMAIN : G1(G1N1,G1N2) 3 rows,
                            G2(G2N1,G2N2) 3 rows,
                            G3(G3N1,G3N2) 3 rows
            RE-GROUPING
              GROUP KEY : ORDERS.O_ORDERSTATUS
    4   -   TARGET : _A1.O_ORDERSTATUS
    5   -   GROUP KEY : _A1.O_ORDERSTATUS
            READ KEY COLUMN : _A1.O_ORDERSTATUS
    6   -   HASH SHARD ( # 3 )
            READ COLUMN : _A1.O_ORDERSTATUS, _A1.O_ORDERDATE
              PHYSICAL FILTER : _A1.O_ORDERDATE = :_V0


<<<  end print plan
```

上述execution paln所示传输包含DISTINCT的SQL并以remote distinct执行

<distinct cluster regrouping hints>

是在集群系统中执行distinct子句时使用的hint在单机版被忽略

MERGE_DISTINCT

满足以下条件时可使用MERGE_DISTINCT hint

- 有distinct子句
- 可以以remote distinct执行
- 在distinct节点的下级节点中中间结果以保证distinct key column的order的状态上升

使用MERGE_DISTINCT hint时driver获取中间结果后也能保证order

以下为使用MERGE_DISTINCT hint的示例

```
\EXPLAIN PLAN
SELECT /*+ MERGE_DISTINCT */
       DISTINCT o_custkey
  FROM orders
 WHERE o_custkey > 0
ORDER BY o_custkey;


O_CUSTKEY
---------
        1
        2
        4
      ...


99996 rows selected.
```

```
>>>  start print plan


< Execution Plan >

================================================================================
|IDX|  NODE DESCRIPTION                                                        |
--------------------------------------------------------------------------------
| 0 |  SELECT STATEMENT                                                        |
| 1 |    QUERY BLOCK ("$QB_IDX_2")                                             |
| 2 |      MULTIPLE CLUSTER                                                    |
| 3 |        SELECT STATEMENT                                                  |
| 4 |          QUERY BLOCK ("$QB_IDX_2")                                       |
| 5 |            GROUP                                                         |
| 6 |              INDEX ACCESS ("ORDERS" AS _A1, "ORDERS_CUSTKEY_FK")         |
================================================================================


     1  -  TARGET : ORDERS.O_CUSTKEY
     2  -  SQL : SELECT /*+ INDEX( _A1, "PUBLIC"."ORDERS_CUSTKEY_FK" ) */
                        DISTINCT "_A1"."O_CUSTKEY"
                   FROM "PUBLIC"."ORDERS"@LOCAL AS "_A1"
                  WHERE "_A1"."O_CUSTKEY" > :_V0
               ORDER BY "_A1"."O_CUSTKEY" ASC NULLS LAST
            TARGET DOMAIN : G1(G1N1,G1N2) 98218 rows,
                            G2(G2N1,G2N2) 98174 rows,
                            G3(G3N1,G3N2) 98138 rows
```

```
            MERGE GROUPING
              SORT KEY : ORDERS.O_CUSTKEY
              GROUP KEY : ORDERS.O_CUSTKEY
      4   -   TARGET : _A1.O_CUSTKEY
      5   -   GROUP KEY : _A1.O_CUSTKEY
      6   -   HASH SHARD ( # 3 )
            READ INDEX COLUMN : _A1.O_CUSTKEY
              MIN RANGE : _A1.O_CUSTKEY > :_V0
              MAX RANGE : _A1.O_CUSTKEY IS NOT NULL

<<<  end print plan
```

如上述execution plan所示*MULTIPLE CLUSTER(IDX:2)*在维持着o_custkey的order因此也不需要用于ORDER BY的SORT而被删除

<order hints>

为与order by处理相关的hint因此仅在有order by子句的情况下有效

<order operation hints>

USE_ORDER_SORT

描述USE_ORDER_SORT hint时optimizer使用sort instant处理order by

通常处理order by时使用sort instant而从下级节点row对sort key column排列上来时不积累sort instant并处理order by

Optimizer执行cost estimation时不积累sort instant诱导下级节点使用index但统计信息不准确时此方法的成本会更高因此这种情况可使用USE_ORDER_SORT hint

以下为使用USE_ORDER_SORT hint的示例对比使用hint前的execution plan和使用hint后的execution plan可理解USE_ORDER_SORT hint处理方式

```
\EXPLAIN PLAN
  SELECT n_nationkey,
         n_name
    FROM nation
ORDER BY n_nationkey;


>>>  start print plan


< Execution Plan >
================================================================================
|  IDX  |  NODE DESCRIPTION                                                     |
--------------------------------------------------------------------------------
|    0  |  SELECT STATEMENT                                                     |
|    1  |    QUERY BLOCK ("$QB_IDX_2")                                          |
|    2  |      INDEX ACCESS ("NATION", "NATION_PK_INDEX")                       |
================================================================================


     1  -  TARGET : NATION.N_NATIONKEY, NATION.N_NAME
     2  -  READ INDEX COLUMN : NATION.N_NATIONKEY
           READ TABLE COLUMN : NATION.N_NAME
```

```
<<<  end print plan


\EXPLAIN PLAN
  SELECT /*+ USE_ORDER_SORT */
         n_nationkey,
         n_name
    FROM nation
ORDER BY n_nationkey;


>>>  start print plan


< Execution Plan >
=============================================================================
|  IDX  |  NODE DESCRIPTION                                                  |
-----------------------------------------------------------------------------
|    0  |  SELECT STATEMENT                                                  |
|    1  |    QUERY BLOCK ("$QB_IDX_2")                                       |
|    2  |      SORT INSTANT                                                  |
|    3  |        TABLE ACCESS ("NATION")                                     |
=============================================================================


     1  -  TARGET : NATION.N_NATIONKEY, NATION.N_NAME
     2  -  SORT KEY : "NATION.N_NATIONKEY ASC NULLS LAST"
           RECORD COLUMN : NATION.N_NAME
```

```
            READ KEY COLUMN : NATION.N_NATIONKEY

            READ RECORD COLUMN : NATION.N_NAME

      3   -   READ COLUMN : NATION.N_NATIONKEY, NATION.N_NAME


<<<  end print plan
```

NO_USE_ORDER_SORT

描述NO_USE_ORDER_SORT hint时optimizer在下级节点使row对sort key column排列上升不积累sort instant并处理

```
\EXPLAIN PLAN
  SELECT /*+ NO_USE_ORDER_SORT */
         n_nationkey,
         n_name
    FROM nation
ORDER BY n_nationkey;


>>>  start print plan


< Execution Plan >
================================================================================
|  IDX  |  NODE DESCRIPTION                                                    |
--------------------------------------------------------------------------------
|    0  |  SELECT STATEMENT                                                    |
|    1  |    QUERY BLOCK ("$QB_IDX_2")                                         |
```

```
|    2  |      INDEX ACCESS ("NATION", "NATION_PK_INDEX")              |
=========================================================================


     1  -  TARGET : NATION.N_NATIONKEY, NATION.N_NAME
     2  -  READ INDEX COLUMN : NATION.N_NATIONKEY
           READ TABLE COLUMN : NATION.N_NAME


<<<  end print plan
```

如上述execution plan没有SORT INSTANTINDEX ACCESS代替执行输出对n_nationkey排列的结果

USE_ORDER_LIMIT_SORT

描述USE_ORDER_LIMIT_SORT hint时optimizer使用limit sort方法排列结果当然要同时使用LIMIT子句和ORDER BY子句才能应用

```
\EXPLAIN PLAN
  SELECT /*+ USE_ORDER_LIMIT_SORT */
         n_nationkey,
         n_name
    FROM nation
ORDER BY n_nationkey
   LIMIT 10,10;



>>>  start print plan
```

```
< Execution Plan >
=============================================================================
|  IDX  |  NODE DESCRIPTION                                                  |
-----------------------------------------------------------------------------
|    0  |  SELECT STATEMENT                                                  |
|    1  |    QUERY BLOCK ("$QB_IDX_2")                                       |
|    2  |      SORT INSTANT                                                  |
|    3  |        TABLE ACCESS ("NATION")                                     |
=============================================================================


     1  -  TARGET : NATION.N_NATIONKEY, NATION.N_NAME
     2  -  LIMIT SORT
           SORT KEY : "NATION.N_NATIONKEY ASC NULLS LAST"
           RECORD COLUMN : NATION.N_NAME
           READ KEY COLUMN : NATION.N_NATIONKEY
           READ RECORD COLUMN : NATION.N_NAME
     3  -  READ COLUMN : NATION.N_NATIONKEY, NATION.N_NAME


<<<  end print plan
```

如上述execution plan可看到SORT INSTANT以LIMIT SORT执行

NO_USE_ORDER_LIMIT_SORT

描述NO_USE_ORDER_LIMIT_SORT hint时optimizer不应用limit sort方法因此生成SORT INSTANT

处理或在下级节点使row相对sort key column排列上升

```
\EXPLAIN PLAN
  SELECT /*+ NO_USE_ORDER_LIMIT_SORT */
         n_nationkey,
         n_name
    FROM nation
ORDER BY n_nationkey
   LIMIT 10,10;



>>>  start print plan


< Execution Plan >
=============================================================================
|  IDX  |  NODE DESCRIPTION                                                 |
-----------------------------------------------------------------------------
|    0  |  SELECT STATEMENT                                                 |
|    1  |    QUERY BLOCK ("$QB_IDX_2")                                      |
|    2  |      INDEX ACCESS ("NATION", "NATION_PK_INDEX")                   |
=============================================================================


     1  -  TARGET : NATION.N_NATIONKEY, NATION.N_NAME
     2  -  READ INDEX COLUMN : NATION.N_NATIONKEY
           READ TABLE COLUMN : NATION.N_NAME
```

```
<<<  end print plan
```

如上述execution plan可看到未应用LIMIT SORT方法

<order driver hints>

是在集群系统中执行order by子句时使用的hint在单机版被忽略

LOCAL_ORDER

在当前服务器执行order by

以下为使用LOCAL_ORDER hint的示例

```
\EXPLAIN PLAN
  SELECT /*+ LOCAL_ORDER */
         o_orderdate, o_orderstatus
    FROM orders
   WHERE o_orderdate >= date '1993-07-01'
     AND o_orderdate < date '1993-07-01' + interval '1' month
ORDER BY o_orderdate;


>>>  start print plan


< Execution Plan >
================================================================================
==
```

```
|  IDX  |  NODE DESCRIPTION                          |                ROWS
|
-------------------------------------------------------------------------
--
|    0  |  SELECT STATEMENT                          |               19319
|
|    1  |    QUERY BLOCK ("$QB_IDX_2")               |               19319
|
|    2  |      SORT INSTANT                          |               19319
|
|    3  |        PLAN BASED CLUSTER                  | LOCAL/REMOTE 19319
|
|    4  |          TABLE ACCESS ("ORDERS")           |                6453
|
=========================================================================
==

     1  -  TARGET : ORDERS.O_ORDERDATE, ORDERS.O_ORDERSTATUS
     2  -  SORT KEY : "ORDERS.O_ORDERDATE ASC NULLS LAST"
           RECORD COLUMN : ORDERS.O_ORDERSTATUS
           READ KEY COLUMN : ORDERS.O_ORDERDATE
           READ RECORD COLUMN : ORDERS.O_ORDERSTATUS
     3  -  SQL : SELECT /*+ FULL( _A1 ) */
                        "_A1"."O_ORDERSTATUS", "_A1"."O_ORDERDATE"
                   FROM "PUBLIC"."ORDERS"@LOCAL AS "_A1"
```

```
                WHERE "_A1"."O_ORDERDATE" < :_V0
                  AND "_A1"."O_ORDERDATE" >= :_V1
          TARGET DOMAIN : G1(G1N1,G1N2) 6453 rows,
                          G2(G2N1,G2N2) 6450 rows,
                          G3(G3N1,G3N2) 6416 rows
     4  -  HASH SHARD ( # 3 )
          READ COLUMN : ORDERS.O_ORDERSTATUS, ORDERS.O_ORDERDATE
            PHYSICAL FILTER : ORDERS.O_ORDERDATE < DATE'1993-07-01' +
CAST( '1' AS INTERVAL(MONTH) ) AND ORDERS.O_ORDERDATE >= DATE'1993-07-01'
```

REMOTE_ORDER

在各个group的服务器执行order by

以下为使用REMOTE_ORDER hint的示例

```
\EXPLAIN PLAN
  SELECT /*+ REMOTE_ORDER */
         o_orderdate, o_orderstatus
    FROM orders
   WHERE o_orderdate >= date '1993-07-01'
     AND o_orderdate < date '1993-07-01' + interval '1' month
ORDER BY o_orderdate;

>>>  start print plan
```

```
< Execution Plan >
================================================================================
==
|  IDX  |  NODE DESCRIPTION                          |               ROWS
|
--------------------------------------------------------------------------------
--
|    0  |  SELECT STATEMENT                          |              19319
|
|    1  |    QUERY BLOCK ("$QB_IDX_2")               |              19319
|
|    2  |      MULTIPLE CLUSTER                      | LOCAL/REMOTE 19319
|
|    3  |        SELECT STATEMENT                    |               6453
|
|    4  |          QUERY BLOCK ("$QB_IDX_2")         |               6453
|
|    5  |            SORT INSTANT                    |               6453
|
|    6  |              TABLE ACCESS ("ORDERS" AS _A1) |               6453
|
================================================================================
==

     1  -  TARGET : ORDERS.O_ORDERDATE, ORDERS.O_ORDERSTATUS
```

```
     2  -  SQL : SELECT /*+ USE_ORDER_SORT
                              FULL( _A1 )
                            */
                            "_A1"."O_ORDERDATE", "_A1"."O_ORDERSTATUS"
                       FROM "PUBLIC"."ORDERS"@LOCAL AS "_A1"
                      WHERE "_A1"."O_ORDERDATE" < :_V0
                        AND "_A1"."O_ORDERDATE" >= :_V1
                    ORDER BY "_A1"."O_ORDERDATE" ASC NULLS LAST
           TARGET DOMAIN : G1(G1N1,G1N2) 6453 rows,
                           G2(G2N1,G2N2) 6450 rows,
                           G3(G3N1,G3N2) 6416 rows
           MERGE SORTING
             SORT KEY : ORDERS.O_ORDERDATE
     4  -  TARGET : _A1.O_ORDERDATE, _A1.O_ORDERSTATUS
     5  -  SORT KEY : "_A1.O_ORDERDATE ASC NULLS LAST"
           RECORD COLUMN : _A1.O_ORDERSTATUS
           READ KEY COLUMN : _A1.O_ORDERDATE
           READ RECORD COLUMN : _A1.O_ORDERSTATUS
     6  -  HASH SHARD ( # 3 )
           READ COLUMN : _A1.O_ORDERSTATUS, _A1.O_ORDERDATE
             PHYSICAL FILTER : _A1.O_ORDERDATE < :_V0 AND
_A1.O_ORDERDATE >= :_V1


<<<  end print plan
```

<aggregation hints>

是执行single row aggregation时可使用的hint

<aggregation driver hints>

是在集群系统中执行single row aggregation时使用的hint在单机版被忽略

LOCAL_AGGR

在当前服务器执行aggregation

以下为使用LOCAL_AGGR hint的示例

```
\EXPLAIN PLAN
SELECT /*+ LOCAL_AGGR */
       sum(ps_supplycost * ps_availqty) * 0.0001
  FROM partsupp,
       supplier,
       nation
 WHERE ps_suppkey = s_suppkey
   AND s_nationkey = n_nationkey
   AND n_name = 'GERMANY';


>>>  start print plan


< Execution Plan >
================================================================================
```

```
==
|IDX|  NODE DESCRIPTION                                |                ROWS
|
-------------------------------------------------------------------------------
--
| 0 |  SELECT STATEMENT                                |                   1
|
| 1 |    QUERY BLOCK ("$QB_IDX_2")                     |                   1
|
| 2 |      AGGREGATION BY HASH                         |                   1
|
| 3 |        PLAN BASED CLUSTER                        | LOCAL/REMOTE 31680
|
| 4 |          NESTED JOIN (INNER JOIN)                |               10508
|
| 5 |            NESTED JOIN (INNER JOIN)              |                 396
|
| 6 |              TABLE ACCESS ("NATION")             |                   1
|
| 7 |              INDEX ACCESS ("SUPPLIER")           |                 396
|
| 8 |            INDEX ACCESS ("PARTSUPP")             |               10508
|
===============================================================================
==
```

```
     1  -  TARGET : SUM( PARTSUPP.PS_SUPPLYCOST * PARTSUPP.PS_AVAILQTY ) *
0.0001
     2  -  AGGREGATION : SUM( PARTSUPP.PS_SUPPLYCOST *
PARTSUPP.PS_AVAILQTY )
     3  -  SQL : SELECT /*+ KEEP_JOINED_TABLE
                            USE_NL_IN( _A1 )
                            USE_NL_IN( _A2 )
                            FULL( _A3 )
                            INDEX( _A2, "PUBLIC"."SUPPLIER_NATIONKEY_FK" )
                            INDEX( _A1, "PUBLIC"."PARTSUPP_SUPPKEY_FK" )
                          */
                          "_A1"."PS_SUPPLYCOST", "_A1"."PS_AVAILQTY"
                     FROM ( ( "PUBLIC"."NATION"@LOCAL AS "_A3"
                              INNER JOIN
                              "PUBLIC"."SUPPLIER"@LOCAL AS "_A2" ON true
                            ) ALIAS "_A4"
                            INNER JOIN
                            "PUBLIC"."PARTSUPP"@LOCAL AS "_A1" ON true
                          ) ALIAS "_A5"
                    WHERE "_A3"."N_NAME" = :_V0
                      AND "_A2"."S_NATIONKEY" = "_A3"."N_NATIONKEY"
                      AND "_A1"."PS_SUPPKEY" = "_A2"."S_SUPPKEY"
               TARGET DOMAIN : G1(G1N1,G1N2) 10508 rows,
                               G2(G2N1,G2N2) 10567 rows,
```

```
                          G3(G3N1,G3N2) 10605 rows
     4  -  JOINED COLUMN : PARTSUPP.PS_SUPPLYCOST, PARTSUPP.PS_AVAILQTY
     5  -  JOINED COLUMN : SUPPLIER.S_SUPPKEY
     6  -  CLONED
           READ COLUMN : NATION.N_NATIONKEY, NATION.N_NAME
             PHYSICAL FILTER : NATION.N_NAME = 'GERMANY'
     7  -  CLONED
           READ INDEX COLUMN : SUPPLIER.S_NATIONKEY
           READ TABLE COLUMN : SUPPLIER.S_SUPPKEY
             MIN RANGE : SUPPLIER.S_NATIONKEY = {NATION.N_NATIONKEY}
             MAX RANGE : SUPPLIER.S_NATIONKEY = {NATION.N_NATIONKEY}
     8  -  HASH SHARD ( # 3 )
           READ INDEX COLUMN : PARTSUPP.PS_SUPPKEY
           READ TABLE COLUMN : PARTSUPP.PS_AVAILQTY,
PARTSUPP.PS_SUPPLYCOST
             MIN RANGE : PARTSUPP.PS_SUPPKEY = {SUPPLIER.S_SUPPKEY}
             MAX RANGE : PARTSUPP.PS_SUPPKEY = {SUPPLIER.S_SUPPKEY}


<<<  end print plan
```

上述execution所示将所有join结果获取至local后执行了aggregation

REMOTE_AGGR

在各个group的服务器执行aggregation

以下为使用REMOTE_AGGR hint的示例

```
\EXPLAIN PLAN
SELECT /*+ REMOTE_AGGR */
       sum(ps_supplycost * ps_availqty) * 0.0001
  FROM partsupp,
       supplier,
       nation
 WHERE ps_suppkey = s_suppkey
   AND s_nationkey = n_nationkey
   AND n_name = 'GERMANY';


>>>  start print plan


< Execution Plan >
================================================================================
==
| IDX |  NODE DESCRIPTION                                |           ROWS
|
--------------------------------------------------------------------------------
--
|  0 |  SELECT STATEMENT                                |              1
|
|  1 |    QUERY BLOCK ("$QB_IDX_2")                     |              1
|
|  2 |      SINGLE CLUSTER                              | LOCAL/REMOTE 1
|
```

```
|  3 |        SELECT STATEMENT                          |              1
|
|  4 |          QUERY BLOCK ("$QB_IDX_2")               |              1
|
|  5 |            AGGREGATION BY HASH                   |              1
|
|  6 |              NESTED JOIN (INNER JOIN)            |          10508
|
|  7 |                NESTED JOIN (INNER JOIN)          |            396
|
|  8 |                  TABLE ACCESS ("NATION" AS _A3)  |              1
|
|  9 |                  INDEX ACCESS ("SUPPLIER" AS _A2)|            396
|
| 10 |                INDEX ACCESS ("PARTSUPP" AS _A1)  |          10508
|
================================================================================
==

     1  -  TARGET : SUM( PARTSUPP.PS_SUPPLYCOST * PARTSUPP.PS_AVAILQTY ) *
0.0001
     2  -  SQL : SELECT /*+ KEEP_JOINED_TABLE
                            USE_NL_IN( _A1 )
                            USE_NL_IN( _A2 )
                            FULL( _A3 )
```

```
                          INDEX( _A2, "PUBLIC"."SUPPLIER_NATIONKEY_FK" )
                          INDEX( _A1, "PUBLIC"."PARTSUPP_SUPPKEY_FK" )
                        */
                        SUM( "_A1"."PS_SUPPLYCOST" *
"_A1"."PS_AVAILQTY" )
                   FROM ( ( "PUBLIC"."NATION"@LOCAL AS "_A3"
                          INNER JOIN
                          "PUBLIC"."SUPPLIER"@LOCAL AS "_A2" ON true
                        ) ALIAS "_A4"
                        INNER JOIN
                        "PUBLIC"."PARTSUPP"@LOCAL AS "_A1" ON true
                      ) ALIAS "_A5"
                  WHERE "_A3"."N_NAME" = :_V0
                    AND "_A2"."S_NATIONKEY" = "_A3"."N_NATIONKEY"
                    AND "_A1"."PS_SUPPKEY" = "_A2"."S_SUPPKEY"
            TARGET DOMAIN : G1(G1N1,G1N2) 1 rows,
                            G2(G2N1,G2N2) 1 rows,
                            G3(G3N1,G3N2) 1 rows
            RE-AGGREGATION
              AGGREGATION : SUM( SUM( PARTSUPP.PS_SUPPLYCOST *
PARTSUPP.PS_AVAILQTY ) )
     4  -  TARGET : SUM( _A1.PS_SUPPLYCOST * _A1.PS_AVAILQTY )
     5  -  AGGREGATION : SUM( _A1.PS_SUPPLYCOST * _A1.PS_AVAILQTY )
     6  -  JOINED COLUMN : _A1.PS_SUPPLYCOST, _A1.PS_AVAILQTY
              CONSTANT FILTER : TRUE
```

```
     7  -  JOINED COLUMN : _A2.S_SUPPKEY
             CONSTANT FILTER : TRUE
     8  -  CLONED
           READ COLUMN : _A3.N_NATIONKEY, _A3.N_NAME
             PHYSICAL FILTER : _A3.N_NAME = :_V0
     9  -  CLONED
           READ INDEX COLUMN : _A2.S_NATIONKEY
           READ TABLE COLUMN : _A2.S_SUPPKEY
             MIN RANGE : _A2.S_NATIONKEY = {_A3.N_NATIONKEY}
             MAX RANGE : _A2.S_NATIONKEY = {_A3.N_NATIONKEY}
    10  -  HASH SHARD ( # 3 )
           READ INDEX COLUMN : _A1.PS_SUPPKEY
           READ TABLE COLUMN : _A1.PS_AVAILQTY, _A1.PS_SUPPLYCOST
             MIN RANGE : _A1.PS_SUPPKEY = {_A2.S_SUPPKEY}
             MAX RANGE : _A1.PS_SUPPKEY = {_A2.S_SUPPKEY}


<<<  end print plan
```

如上述execution plan所示在各服务器执行aggregation后将其结果传输至local并执行了re-aggregation

5.7 SQL Trace Log

概要

SQL trace log是记录可分析的用户查询执行信息的logSQL trace log分为进程ID和会话ID并各自以独立的文件生成在$SUNDB_DATA/trc目录SQL trace log由用户查询SQL执行计划SQL处理的各过程执行时间等多种信息组成并输出到文件

输出

为了输出SQL trace log需要使用ALTER SESSION或ALTER SYSTEM语句设置TRACE_LOG_ID的值设定值参考server property的**TRACE_LOG_ID**

Trace log均可记录成功的SQL查询和失败的SQL查询其可组合TRACE_LOG_ID的flag值设置

> Note:
>
> **SQL处理过程** 中在executor过程失败的SQL查询语句叫执行失败的SQL查询语句因此在parsevalidatorrewriterenumeratorcode plannerdata planner过程失败的SQL语句不记录trace log

以下为使用TRACE_LOG_ID参数输出SQL trace log的示例

- 输出成功的SQL语句和失败的SQL语句

```
gSQL> ALTER SESSION SET TRACE_LOG_ID = 110000;


Session altered.
```

- 输出成功的SQL语句和bind值

```
gSQL> ALTER SYSTEM SET TRACE_LOG_ID = 100010;


System altered.
```

将输出SQL trace log的参数TRACE_LOG_ID设置为ALTER SESSION时仅在该会话进行反映并操作设置为ALTER SYSTEM时反映到访问服务器的所有进程的所有会话因此要查看当前会话的SQL trace log时使用ALTER SESSION要查看当前执行的其他进程及其他会话的 SQL trace log时使用ALTER SYSTEM

> Caution:
>
> 设置为ALTER SYSTEM的状态下访问服务器的进程和会话数量多时也会生成相同数量的SQL trace log因此须注意使用

SQL trace log 文件在trc目录下生成文件名规则如下

```
opt_p[进程ID]_s[会话ID].trc
```

文件名前opt在其后p标识符和进程ID在其后s标识符和会话ID_为分隔符扩展名为trc如果在相同进程的相同会话中生成的内容大于SQL trace log文件的最大大小时原有文件变更为在文件名后增加当前时间点的时间的文件名并以当前文件名继续创建新的文件并记录

以下为SQL trace log文件名的示例

```
opt_p17104_s12.trc
```

输出类型

SQL trace log大致分为<SQL query string>, <Execution plan>, <Execution type>, <Bind param value>, <Time info>

SQL Query String

包含当前时间和成功与否查询处理时间输出用户输入的查询输出形式如下

```
[当前时间] [成功与否][查询处理时间] SQL 语句
```

[当前时间]输出日期和us单位的时间[成功与否]在成功时输出s失败时输出F查询处理时间输出us单位的时间SQL语句是用户输入的SQL语句

未将**TRACE_LOG_TIME_DETAIL**参数设置为*ON*时查询处理时间测定为10ms单位但此参数设置为*ON*时会降低查询处理性能因此需注意

Execution Plan

输出SQL语句的执行计划其基本与执行计划相同在execution plan node table追加输出total time column输出到statement的total time指执行所有查询的时间其他输出到各个节点的total time指在各个节点执行的时间此时以10ms为单位输出total time

使用**TRACE_LOG_TIME_DETAIL** 参数可输出更加详细的时间但此参数设置为*ON*时会降低查询处理性能因此需注意

Execution Type

SQL语句的执行形式直接执行查询时输出*DIRECT EXECUTE*使用prepare执行查询时输出*PREPARE EXECUTE*한다

Bind Param Value

在SQL语句使用bind param value时输出该bind param value的信息如果SQL语句不使用bind param value时输出*No Bind Param*

Time Info

输出SQL处理过程的各阶段执行时间Time info分为moduletimeratecall输出module输出parsevalidate等阶段名time输出实际执行时间rate输出在整体执行时间中各阶段的执行时间占据的比例Call输出各阶段被调用的次数

Module分为parse, validate, code opt, optimizer, data opt, execute, fetch七个阶段和totalParse对查询进行parsingvalidate对parsing的查询执行validationCode opt是执行SQL optimizer的预处理阶段optimizer实际执行SQL optimizerData opt是实际执行SQL执行计划的准备阶段execute执行SQL执行计划Fetch收集SELECT语句等查询结果并返回结果

使用plan cache时可能不调用validatecode optoptimizer等Time仅输出10ms单位10ms以下的执行时间输出为0而且rate是整体执行时间中的各阶段执行时间的比例因此将total作为100%输出

除以各阶段执行时间的比例此时各阶段执行时间为0时输出0%

使用**TRACE_LOG_TIME_DETAIL** 参数可输出更加详细的时间但此参数设置为*ON*时会降低查询处理性能因此需注意

输出示例

以下为没有bind param value的SQL语句

```
SELECT O_TOTALPRICE, O_ORDERDATE, L_QUANTITY
  FROM ORDERS, LINEITEM
 WHERE O_ORDERKEY = L_ORDERKEY
   AND O_ORDERDATE >= DATE '1996-01-01'
   AND L_SHIPMODE = 'AIR';
```

以下为将TRACE_LOG_ID设置为101111后执行上述SQL语句时输出的SQL trace log

```
[2017-05-25 12:27:59.199657] [S][0.000000] SELECT O_TOTALPRICE,
O_ORDERDATE, L_QUANTITY
  FROM ORDERS, LINEITEM
 WHERE O_ORDERKEY = L_ORDERKEY
   AND O_ORDERDATE >= DATE '1996-01-01'
   AND L_SHIPMODE = 'AIR'


< Execution Plan >
================================================================================
```

```
==
| IDX |  NODE DESCRIPTION                                | ROWS | Total Time
|
-------------------------------------------------------------------------
--
|  0  |  SELECT STATEMENT                                |      | 0:00:00.00
|
|  1  |    NESTED LOOP JOIN (INNER JOIN)                 |    1 | 0:00:00.00
|
|  2  |      TABLE ACCESS ("LINEITEM")                   |    2 | 0:00:00.00
|
|  3  |      INDEX ACCESS ("ORDERS, ORDERS_PK_INDEX")    |    1 | 0:00:00.00
|
=========================================================================
==


     1  -  JOINED COLUMNS : ORDERS.O_TOTALPRICE, ORDERS.O_ORDERDATE,
LINEITEM.L_QUANTITY
     2  -  READ COLUMNS : L_ORDERKEY, L_QUANTITY, L_SHIPMODE
              PHYSICAL FILTER : L_SHIPMODE = 'AIR'
     3  -  READ INDEX COLUMNS : O_ORDERKEY
           READ TABLE COLUMNS : O_TOTALPRICE, O_ORDERDATE
              MIN RANGE : O_ORDERKEY = {L_ORDERKEY}
              MAX RANGE : O_ORDERKEY = {L_ORDERKEY}
              PHYSICAL TABLE FILTER : O_ORDERDATE >= CAST( '1996-01-01' AS
```

```
DATE )




< Execution Type >

----------------------

  DIRECT EXECUTE



< Bind Param Value >

--------------------------

  No Bind Param.



< Time Info >

============================================

| Module    | Time       | Rate     | Call |

--------------------------------------------

| Parse     | 0:00:00.00 |   0.00 % |    1 |

| Validate  | 0:00:00.00 |   0.00 % |    1 |

| Code Opt  | 0:00:00.00 |   0.00 % |    1 |

| Optimizer | 0:00:00.00 |   0.00 % |    1 |

| Data Opt  | 0:00:00.00 |   0.00 % |    1 |

| Execute   | 0:00:00.00 |   0.00 % |    1 |

| Fetch     | 0:00:00.00 |   0.00 % |    1 |
```

```
| Total     | 0:00:00.00 | 100.00 % |      |
==========================================
```

以下为有bind param value的SQL语句

```
SELECT L_QUANTITY
  FROM LINEITEM
 WHERE L_SHIPMODE = :V1;
```

以下为将TRACE_LOG_ID设置为101111后执行上述SQL语句时输出的SQL trace log

```
[2017-05-25 12:27:59.200204] [S][0.000000] SELECT L_QUANTITY
  FROM LINEITEM
 WHERE L_SHIPMODE = :V1


< Execution Plan >
================================================================================
==
|  IDX  |  NODE DESCRIPTION                      |       ROWS | Total Time
|
--------------------------------------------------------------------------------
--
|    0  |  SELECT STATEMENT                      |            | 0:00:00.00
|
|    1  |    TABLE ACCESS ("LINEITEM")           |          0 | 0:00:00.00
|
```

```
================================================================================
==

     1  -  READ COLUMNS : L_QUANTITY, L_SHIPMODE
              PHYSICAL FILTER : L_SHIPMODE = :V1




< Execution Type >
----------------------
  DIRECT EXECUTE



< Bind Param Value >
--------------------------
   1 - :V1(IN, "AIR")



< Time Info >
============================================
| Module    | Time       | Rate     | Call |
--------------------------------------------
| Parse     | 0:00:00.00 |   0.00 % |    1 |
| Validate  | 0:00:00.00 |   0.00 % |    1 |
| Code Opt  | 0:00:00.00 |   0.00 % |    1 |
```

```
| Optimizer | 0:00:00.00 |   0.00 % |    1 |

| Data Opt  | 0:00:00.00 |   0.00 % |    1 |

| Execute   | 0:00:00.00 |   0.00 % |    1 |

| Fetch     | 0:00:00.00 |   0.00 % |    1 |

| Total     | 0:00:00.00 | 100.00 % |      |

============================================
```

6.Built-in Data Type References

6.1 Aliases of Built-in Data Types

- BIGINT
  - 与NUMBER(19,0)相同
  - 参考：**NUMBER**
- BINARY
  - 参考：**BINARY**
- BINARY VARYING
  - 参考：**BINARY VARYING**
- BINARY LONG VARYING
  - 参考：**BINARY LONG VARYING**
- BOOLEAN
  - 参考：**BOOLEAN**
- CHAR
  - 与CHARACTER相同
  - 参考：**CHARACTER**
- CHARACTER
  - 参考：**CHARACTER**
- CHARACTER VARYING
  - 参考：**CHARACTER VARYING**
- CHARACTER LONG VARYING

- 参考：**CHARACTER LONG VARYING**

- DATE
  - 参考：**DATE**
- DEC
  - 与NUMERIC相同
  - 参考：**NUMERIC**
- DECIMAL
  - 与NUMERIC相同
  - 参考：**NUMERIC**
- DOUBLE
  - 与FLOAT(53)相同
  - 参考：**FLOAT**
- DOUBLE PRECISION
  - 与FLOAT(53)相同
  - 参考：**FLOAT**
- FLOAT
  - 参考：**FLOAT**
- FLOAT4
  - 与FLOAT(24)相同
  - 参考：**FLOAT**
- FLOAT8
  - 与FLOAT(53)相同
  - 参考：**FLOAT**
- INT
  - 与NUMBER(10,0)相同

- 参考：**NUMBER**

- INT2
  - 与NUMBER(5,0) 相同
  - 参考：**NUMBER**
- INT4
  - 与NUMBER(10,0)相同
  - 参考：**NUMBER**
- INT8
  - 与NUMBER(19,0)相同
  - 参考：**NUMBER**
- INTEGER
  - 与NUMBER(10,0)相同
  - 参考：**NUMBER**
- INTERVAL
  - 参考：**INTERVAL**
- LONG BINARY VARYING
  - 与BINARY LONG VARYING相同
  - 参考：**BINARY LONG VARYING**
- LONG CHAR VARYING
  - 与CHARACTER LONG VARYING相同
  - 参考：**CHARACTER LONG VARYING**
- LONG CHARACTER VARYING
  - 与CHARACTER LONG VARYING相同
  - 参考：**CHARACTER LONG VARYING**
- LONG VARCHAR

- 与CHARACTER LONG VARYING相同
- 参考：**CHARACTER LONG VARYING**

- NATIVE_BIGINT
  - 参考：**NATIVE_BIGINT**
- NATIVE_DOUBLE
  - 参考：**NATIVE_DOUBLE**
- NATIVE_INTEGER
  - 参考：**NATIVE_INTEGER**
- NATIVE_REAL
  - 参考：**NATIVE_REAL**
- NATIVE_SMALLINT
  - 参考：**NATIVE_SMALLINT**
- NUMBER
  - 参考：**NUMBER**
- NUMERIC
  - 参考：**NUMERIC**
- ROWID
  - 参考：**ROWID**
- SMALLINT
  - 与NUMBER(5,0)相同
  - 参考：**NUMBER**
- TIME
  - 参考：**TIME**
- TIMESTAMP
  - 参考：**TIMESTAMP**

- VARBINARY
  - 与BINARY VARYING相同
  - 参考：**BINARY VARYING**
- VARCHAR
  - 与CHARACTER VARYING相同
  - 参考：**CHARACTER VARYING**
- VARCHAR2
  - 与CHARACTER VARYING相同
  - 参考：**CHARACTER VARYING**

6.2 BINARY

语句

```
BINARY [ (length) ]
```

语句规则及参数

- length：Binary string的长度
  - 范围：1~2000
  - 默认值：1

说明

存储固定长度的binary string

binary string的长度小于指定的length时其余部分存储为X'00'

- 存储空间的大小：对应length值的bytes

参考

- **BINARY VARYING**
- **BINARY LONG VARYING**

6.3 BINARY VARYING

语句

```
BINARY VARYING (length)
```

语句规则及参数

- length：Binary string的最大长度
  - 范围：1~4000

说明

存储可变长度的binary string

- 存储空间的大小：要存储的binary string的字节大小
- Alias names: VARBINARY

参考

- **BINARY**
- **BINARY LONG VARYING**

6.4 BINARY LONG VARYING

语句

```
BINARY LONG VARYING
```

说明（Description）

存储长的可变binary string

- 最大存储空间：100 Mega bytes
- 存储空间的大小：要存储的binary string的字节大小
- Alias names: LONG BINARY VARYING, LONG VARBINARY

无法用作key column因此有以下约束

- 无法用作索引的key column
- 无法用作ORDER BY语句的expression
- 无法用作GROUP BY语句的expression
- 无法用作DISTINCT语句的expression
- 无法用作UNION, INTERSECT, EXCEPT语句的expression

参考

- BINARY
- BINARY VARYING

6.5 BOOLEAN

语句

```
BOOLEAN
```

说明

存储TRUE或FALSE值

- 存储空间的大小：1 byte

6.6 CHARACTER

语句

```
CHARACTER [ (length [ CHARACTERS | OCTETS | CHAR | BYTE ] ) ]
```

语句规则及参数

- length：字符串的长度
  - 范围：1 ~ 2000
  - 默认值：1
- [CHARACTERS | OCTETS | CHAR | BYTE]： length的单位
  - CHARACTERS
    - 字符的数量*** CHAR与CHARACTERS相同
  - OCTETS
    - byte的数量
    - BYTE与OCTETS相同
  - 省略时遵循生成数据库时设置的**CHAR_LENGTH_UNITS**参数值

说明

存储固定长度的字符串

要存储的string的长度小于指定length时其余部分以空白存储

- 存储空间的大小：与length值相同的bytes大小
- Alias names: CHAR

参考

- **CHARACTER VARYING**
- **CHARACTER LONG VARYING**

6.7 CHARACTER VARYING

语句

```
CHARACTER VARYING ( length [ CHARACTERS | OCTETS | CHAR | BYTE ] )
```

语句规则及参数

- length：字符串的长度
  - 范围：1 ~ 4000
- [CHARACTERS | OCTETS | CHAR | BYTE]: length的单位
  - CHARACTERS
    - 字符数
    - CHAR与CHARACTERS相同
  - OCTETS
    - 字节数
    - BYTE与OCTETS相同
  - 省略时遵循创建数据库时设置的**CHAR_LENGTH_UNITS**参数值

说明

存储可变长度的字符串

- 存储空间的大小：要存储的字符串的byte大小
- Alias names: VARCHAR, VARCHAR2

参考

- **CHARACTER**
- **CHARACTER LONG VARYING**

6.8 CHARACTER LONG VARYING

语句

```
CHARACTER LONG VARYING
```

说明

存储长的可变字符串值

- 最大存储空间：100 Mega bytes
- 存储空间的大小：要存储的字符串的字节大小
- Alias names: LONG CHARACTER VARYING, LONG CHAR VARYING, LONG VARCHAR

无法用作key column因此有以下约束

- 无法用作索引的key column
- 无法用作ORDER BY的expression
- 无法用作GROUP BY的expression
- 无法用作DISTINCT的expression
- 无法用作UNION, INTERSECT, EXCEPT的expression

参考

- CHARACTER
- CHARACTER VARYING

6.9 DATE

语法

```
DATE
```

说明

包含YEAR, MONTH, DAY, HOUR, MINUTE, SECOND(fractional seconds除外)的日期类型

- 值的范围：'4714-11-24 BC' ~ '9999-12-31 AD'范围的日期值
- 存储空间的大小：8 bytes

参考

- **日期字面量（Date Literals）**
- **TIME**
- **TIMESTAMP**

6.10 FLOAT

语句

```
FLOAT[ ( precision ) ]
```

语句规则及参数

- precision：有效数字的二进制精确度
  - precision的范围：1 ~ 126
  - 默认值：126

说明

存储有二进制精确度的的浮动小数点值

- 指数范围：1E-130 ~ 1E+125
- 存储空间的大小：(正数位数 + 1) / 2 + (小数位数 + 1) / 2 + 1(Exponent与sign信息)

与**NUMBER**, **NUMERIC**类型不同具有二进制precision值

- Alias names: REAL = FLOAT(24), DOUBLE = FLOAT(53), DOUBLE PRECISION = FLOAT(53), FLOAT4 = FLOAT(24), FLOAT8 = FLOAT(53)

参考

- **NUMBER**
- **NUMERIC**

6.11 INTERVAL

语句

```
<interval_type> ::=

      INTERVAL YEAR [ ( leading_precision ) ]

    | INTERVAL MONTH [ ( leading_precision ) ]

    | INTERVAL DAY [ ( leading_precision ) ]

    | INTERVAL HOUR [ ( leading_precision ) ]

    | INTERVAL MINUTE [ ( leading_precision ) ]

    | INTERVAL SECOND [ ( leading_precision  [ ,

fractional_seconds_precision ] ) ]

    | INTERVAL YEAR [ ( leading_precision ) ] TO MONTH

    | INTERVAL DAY [ ( leading_precision ) ] TO HOUR

    | INTERVAL DAY [ ( leading_precision ) ] TO MINUTE

    | INTERVAL DAY [ ( leading_precision ) ] TO SECOND

[ ( fractional_seconds_precision ) ]

    | INTERVAL HOUR [ ( leading_precision ) ] TO MINUTE

    | INTERVAL HOUR [ ( leading_precision ) ] TO SECOND

[ ( fractional_seconds_precision ) ]

    | INTERVAL MINUTE [ ( leading_precision ) ] TO SECOND

[ ( fractional_seconds_precision ) ]
```

语句规则及参数

- INTERVAL YEAR [(leading_precision)]：存储YEAR的时间段长度
  - leading_precision
    - YEAR的digit数
    - 值的范围：2 ~ 6
    - 默认值：2
- INTERVAL MONTH [(leading_precision)]：存储MONTH的期限
  - leading_precision
    - MONTH的digit数
    - 值的范围：2 ~ 6
    - 默认值：2
- INTERVAL YEAR [(leading_precision)] TO MONTH：存储YEAR 和 MONTH的期限
  - leading_precision
    - YEAR的digit数
    - 值的范围：2 ~ 6
    - 默认值：2
- INTERVAL DAY [(leading_precision)]：存储DAY的期限
  - leading_precision
    - DAY的digit数
    - 值的范围：2 ~ 6
    - 默认值：2
- INTERVAL HOUR [(leading_precision)]：存储HOUR的期限
  - leading_precision
    - HOUR的digit数

        - 值的范围：2 ~ 6
        - 默认值：2
- INTERVAL MINUTE [(leading_precision)]: 存储MINUTE的期限
    - leading_precision
        - MINUTE的digit数
        - 值的范围：2 ~ 6
        - 默认值：2
- INTERVAL SECOND [(leading_precision [, fractional_seconds_precision])]: 存储SECOND的期限
    - leading_precision
        - SECOND的digit数
        - 值的范围：2 ~ 6
        - 默认值：2
    - fractional_seconds_precision
        - fractional seconds的digit数
        - 值的范围：0 ~ 6
        - 默认值：6
- INTERVAL DAY [(leading_precision)] TO HOUR: 存储DAY 和 HOUR的期限
    - leading_precision
        - DAY的digit数
        - 值的范围：2 ~ 6
        - 默认值：2
- INTERVAL DAY [(leading_precision)] TO MINUTE: 存储DAY, HOUR, MINUTE的期限
    - leading_precision
        - DAY的digit数

        - 值的范围：2 ~ 6
        - 默认值：2
- INTERVAL DAY [(leading_precision)] TO SECOND [(fractional_seconds_precision)]：存储DAY, HOUR, MINUTE, SECOND的期限
    - leading_precision
        - DAY的digit数
        - 值的范围：2 ~ 6
        - 默认值：2
    - fractional_seconds_precision
        - fractional seconds的digit数
        - 值的范围：0 ~ 6
        - 默认值：6
- INTERVAL HOUR [(leading_precision)] TO MINUTE：存储HOUR和 MINUTE的期限
    - leading_precision
        - HOUR的digit数
        - 值的范围：2 ~ 6
        - 默认值：2
- INTERVAL HOUR [(leading_precision)] TO SECOND [(fractional_seconds_precision)]：存储HOUR, MINUTE, SECOND的期限
    - leading_precision
        - HOUR的digit数
        - 值的范围：2 ~ 6
        - 默认值：2
    - fractional_seconds_precision
        - fractional seconds的digit数

    - 值的范围：0 ~ 6
    - 默认值：6
- INTERVAL MINUTE [(leading_precision)] TO SECOND [(fractional_seconds_precision)]:

  存储MINUTE 和 SECOND的期限
  - leading_precision
    - MINUTE的digit数
    - 值的范围：2 ~ 6
    - 默认值：2
  - fractional_seconds_precision
    - fractional seconds的digit数
    - 值的范围：0 ~ 6
    - 默认值：6

说明

INTERVAL类型根据值的表示范围可分为YEAR TO MONTH系列和DAY TO SECOND系列

- YEAR TO MONTH系列：存储空间大小为8 byte
  - INTERVAL YEAR
  - INTERVAL MONTH
  - INTERVAL YEAR TO MONTH
- DAY TO SECOND系列：存储空间大小为16 byte
  - INTERVAL DAY
  - INTERVAL HOUR
  - INTERVAL MINUTE

- INTERVAL SECOND
- INTERVAL DAY TO HOUR
- INTERVAL DAY TO MINUTE
- INTERVAL DAY TO SECOND
- INTERVAL HOUR TO MINUTE
- INTERVAL HOUR TO SECOND
- INTERVAL MINUTE TO SECOND

数字超过指定leading_precision的field的digit时报错

数字超过指定fractional_seconds_precision的field的digit时四舍五入

field	precision	值的范围
MONTH	2	0 ~ 11
HOUR	2	0 ~ 23
MINUTE	2	0 ~ 59
SECOND（小数点之前的值）	2	0 ~ 59

Table 6-1 INTERVAL * TO * 中第二个之后field的precision与值的范围

参考

区间字面量（Interval Literals）

6.12 NATIVE_BIGINT

语句

```
NATIVE_BIGINT
```

说明

存储signed 8-byte整数

与C语言的long long (8 bytes integer)相同

- 值的范围： -9,223,372,036,854,775,808 ~ 9,223,372,036,854,775,807
- 存储空间的大小：8 bytes

6.13 NATIVE_DOUBLE

语句

```
NATIVE_DOUBLE
```

说明

存储double precision floating-point number (8 bytes)

与C语言的double相同

- 指数值的范围：1E-307 ~ 1E+308
- 存储空间的大小：8 bytes

6.14 NATIVE_INTEGER

语句

```
NATIVE_INTEGER
```

说明

存储signed 4-byte整数

与C语言的integer(4 bytes)相同

- 值的范围：-2,147,483,648 ~ +2,147,483,647
- 存储空间的大小：4 bytes

6.15 NATIVE_REAL

语法

```
NATIVE_REAL
```

说明

存储single precision floating-point number (4 bytes)

与C语言的float相同

- 指数值的范围：1E-37 ~ 1E+37
- 存储空间的大小：4 bytes

6.16 NATIVE_SMALLINT

语句

```
NATIVE_SMALLINT
```

说明

存储signed 2-byte整数

与C语言的short相同

- 值的范围：-32,768 ~ 32,767
- 存储空间的大小：2 bytes

6.17 NUMBER

语句

```
NUMBER  [ ( precision [ , scale ] ) ]
```

语句规则及参数

- NUMBER：存储无precision与scale的浮动小数点数字
  - 有效数字的范围：38
  - 指数值的范围：1E-130 ~ 1E+125
  - 与FLOAT(126)相同
- NUMBER(precision)：存储拥有precision的有效位数的整数
  - precision范围：1 ~ 38
  - scale值：0
  - 与**FLOAT**类型不同有十进制precision值
  - 与NUMBER(precision, 0)NUMERIC(precision, 0)相同
- NUMBER(precision, scale)：存储拥有precision与scale的固定小数点数字
  - precision的范围：1 ~ 38
  - scale的范围：-84 ~ 127
  - 与**FLOAT**类型不同有十进制precision值
  - 与NUMERIC(precision,scale)相同
  - Alias names: SMALLINT = NUMBER(5,0), INTEGER = NUMBER(10,0), BIGINT =

NUMBER(19,0), INT2 = NUMBER(5,0), INT4 = NUMBER(10,0), INT8 = NUMBER(19,0)

说明

NUMBER类型与NUMERIC类型相似但省略precision与scale时NUMBER类型存储未指定precision与scale的浮动小数点数字

- 无precision, scale的NUMBER：浮动小数点数字
- 无precision, scale的NUMERIC：NUMERIC(38,0)的固定小数点数字
- 存储空间的大小：(整数位数 + 1) / 2 + (小数位数 + 1) / 2 + 1(Exponent与sign信息)

参考

- **FLOAT**
- **NUMERIC**

6.18 NUMERIC

语句

```
NUMERIC  [ ( precision [ , scale ] ) ]
```

语句规则及参数

- precision：有效数字的十进制精确度
  - precision的范围：1 ~ 38
  - 默认值：38
- scale：小数点的范围
  - scale的范围：-84 ~ 127
  - 默认值：0

说明

存储有precision与scale的固定小数点数字

如下省略precision与scale时意义如下

- NUMERIC = NUMERIC(38,0)
- NUMERIC(p) = NUMERIC(p,0)

NUMBER类型与NUMERIC类型相似但省略precision与scale时NUMBER类型存储未指定precision与scale的浮动小数点数字

- 无指定precision, scale的NUMBER：浮动小数点数字
- 无指定precision, scale的NUMERIC：NUMERIC(38,0)的固定小数点数字
- 存储空间的大小：(整数位数 + 1) / 2 + (小数位数 + 1) / 2 + 1(Exponent与sign信息)

参考

- **FLOAT**
- **NUMBER**

6.19 ROWID

语句

```
ROWID
```

说明

ROWID类型存储记录标识符（ROWID）

记录标识符（ROWID）为数据库的每条记录的识别信息

查询ROWID pseudo column可查看记录识别符（ROWID）的值该ROWID pseudo column拥有ROWID data type信息

单机版系统的ROWID类型的构成要素如下

- OBJECT_ID
- TABLESPACE_ID
- PAGE_ID
- PAGE中的OFFSET

集群系统的ROWID类型的构成要素如下

- GRID_BLOCK_SEQUENCE
- GRID_BLOCK_ID
- MEMBER_ID

• SHARD_ID

ROWID存储为可包含A~Z, a~z, 0~9, +, / 的base 64 value

通过ROWID-Related Functions可获取ROWID的各个构成要素信息

- 存储空间的大小：16 bytes

参考

- **ROWID Pseudo Column**
- **ROWID-related Functions**

6.20 TIME

语句

```
TIME [ ( fractional_seconds_precision ) ] [ WITH TIME ZONE | WITHOUT TIME
ZONE ]
```

语句规则及参数

fractional_seconds_precision：fractional seconds的有效位数

- fractional_seconds_precision的范围：0 ~ 6
- 默认值：6

[WITH TIME ZONE | WITHOUT TIME ZONE]：是否存储TIME ZONE值

- WITH TIME ZONE：含time zone的时间
- WITHOUT TIME ZONE：不含time zone的时间
- 默认值：WITHOUT TIME ZONE

说明

存储拥有HOUR, MINUTE, SECOND的时间

- 存储空间的大小
  - TIME WITHOUT TIME ZONE : 8 bytes
  - TIME WITH TIME ZONE : 12 bytes

参考

- **时间字面量（Time Literals）**
- **有时区的时间字面量（Time with time zone Literals）**
- **DATE**
- **TIMESTAMP**

6.21 TIMESTAMP

语句

```
TIMESTAMP [ ( fractional_seconds_precision ) ] [ WITH TIME ZONE | WITHOUT
TIME ZONE ]
```

语句规则及参数

- fractional_seconds_precision: fractional seconds的有效位数
  - fractional_seconds_precision的范围：0 ~ 6
  - 默认值：6
- [WITH TIME ZONE | WITHOUT TIME ZONE]: 是否存储TIME ZONE值
  - WITH TIME ZONE：包含time zone的时间
  - WITHOUT TIME ZONE：不包含time zone的时间
  - 默认值：WITHOUT TIME ZONE

说明

存储拥有YEAR, MONTH, DATE, HOUR, MINUTE, SECOND的时间

- 存储空间的大小
  - TIMESTAMP WITHOUT TIME ZONE : 8 bytes

- TIMESTAMP WITH TIME ZONE : 12 bytes

参考

- 时间戳字面量（Timestamp Literals）
- 有时区的时间戳字面量（Timestamp with time zone Literals）
- DATE
- TIME

7.Built-in Function References

7.1 * (MULTIPLICATION)

语句

```
expr1 * expr2
```

说明

返回expr1乘以expr2的运算结果

乘法运算的类型与结果如下图所示

详细内容参考**类型间转换（type conversion）**

expr1 (expr2)	expr2 (expr1)	结果类型
NATIVE INTEGER系列 • NATIVE_SMALLINT • NATIVE_INTEGER • NATIVE_BIGINT	NATIVE INTEGER系列 • NATIVE_SMALLINT • NATIVE_INTEGER • NATIVE_BIGINT	NATIVE_BIGINT

expr1 (expr2)	expr2 (expr1)	结果类型
NUMBER	NUMBER	NUMBER
NATIVE DOUBLE系列 • NATIVE_REAL • NATIVE_DOUBLE	NATIVE DOUBLE系列 • NATIVE_REAL • NATIVE_DOUBLE	NATIVE_DOUBLE

Table 7-1 数字型 * 运算

expr1 (expr2)	expr2 (expr1)	结果类型
INTERVAL YEAR TO MONTH	数字型类型	INTERVAL YEAR TO MONTH （结果类型为interval type）
INTERVAL DAY TO SECOND	数字型类型	INTERVAL DAY TO SECOND （结果类型为interval type）
参考：**表中的INTERVAL类型包含的INTERVAL详细类型**		

Table 7-2 INTERVAL * 运算

INTERVAL YEAR TO MONTH	INTERVAL DAY TO SECOND
• INTERVAL YEAR • INTERVAL MONTH • INTERVAL YEAR TO MONTH	• INTERVAL DAY • INTERVAL HOUR • INTERVAL MINUTE • INTERVAL SECOND • INTERVAL DAY TO HOUR • INTERVAL DAY TO MINUTE • INTERVAL DAY TO SECOND • INTERVAL HOUR TO MINUTE • INTERVAL HOUR TO SECOND • INTERVAL MINUTE TO SECOND

Table 7-3 表中的INTERVAL类型包含的INTERVAL详细类型

使用示例

```
gSQL> SELECT INTERVAL'1-2'YEAR TO MONTH * 2 AS RESULT FROM DUAL;

RESULT

----------

+000002-04

1 row selected.


gSQL> SELECT INTERVAL'1 01:02:03.400000'DAY TO SECOND * 2 AS RESULT
```

```
      FROM DUAL;

RESULT

-----------------------

+000002 02:04:06.800000

1 row selected.
```

7.2 + (ADDITION)

语句

```
expr1 + expr2
```

说明

返回expr1加expr2的运算结果

加法运算的种类与结果类型如下图所示

详细内容参考**类型间转换（type conversion）**

expr1 (expr2)	expr2 (expr1)	结果类型
NATIVE INTEGER系列 • NATIVE_SMALLINT • NATIVE_INTEGER • NATIVE_BIGINT	NATIVE INTEGER系列 • NATIVE_SMALLINT • NATIVE_INTEGER • NATIVE_BIGINT	NATIVE_BIGINT
NUMBER	NUMBER	NUMBER

expr1 (expr2)	expr2 (expr1)	结果类型
NATIVE DOUBLE系列 • NATIVE_REAL • NATIVE_DOUBLE	NATIVE DOUBLE系列 • NATIVE_REAL • NATIVE_DOUBLE	NATIVE_DOUBLE

Table 7-4 数字型 + 运算

expr1 (expr2)	expr2 (expr1)	结果类型
DATE	NUMERIC	DATE
DATE	INTERVAL YEAR TO MONTH	DATE
DATE	INTERVAL DAY	DATE
DATE	INTERVAL DAY TO SECOND	TIMESTAMP
TIME	NUMERIC	TIME
TIME	INTERVAL YEAR TO MONTH	TIME
TIME	INTERVAL DAY TO SECOND	TIME
TIME WITH TIME ZONE	NUMERIC	TIME WITH TIME ZONE
TIME WITH TIME ZONE	INTERVAL YEAR TO MONTH	TIME WITH TIME ZONE
TIME WITH TIME ZONE	INTERVAL DAY TO SECOND	TIME WITH TIME ZONE
TIMESTAMP	NUMERIC	TIMESTAMP
TIMESTAMP	INTERVAL YEAR TO MONTH	TIMESTAMP

expr1 (expr2)	expr2 (expr1)	结果类型
TIMESTAMP	INTERVAL DAY TO SECOND	TIMESTAMP
TIMESTAMP WITH TIME ZONE	NUMERIC	TIMESTAMP WITH TIME ZONE
TIMESTAMP WITH TIME ZONE	INTERVAL YEAR TO MONTH	TIMESTAMP WITH TIME ZONE
TIMESTAMP WITH TIME ZONE	INTERVAL DAY TO SECOND	TIMESTAMP WITH TIME ZONE
INTERVAL YEAR TO MONTH	INTERVAL YEAR TO MONTH	INTERVAL YEAR TO MONTH（决定为可包含所有expr1与expr2的 interval范围的类型）
INTERVAL DAY TO SECOND	INTERVAL DAY TO SECOND	INTERVAL DAY TO SECOND（决定为可包含所有expr1与expr2的interval范围的类型）
参考：**表中的INTERVAL类型包含的INTERVAL详细类型**		

Table 7-5 (DATETIME/INTERVAL) + 运算

使用示例

```
gSQL> SELECT TO_DATE( '2012-05-05', 'YYYY-MM-DD' ) + 5 AS RESULT FROM
DUAL;
RESULT
----------
2012-05-10
```

```
1 row selected.


gSQL> SELECT
      TO_DATE( '2012-05-05', 'YYYY-MM-DD' ) + INTERVAL'01-01'YEAR TO MONTH
      AS RESULT
      FROM DUAL;
RESULT
----------
2013-06-05
1 row selected.


gSQL> SELECT
      INTERVAL'01-01'YEAR TO MONTH + INTERVAL'02-10'YEAR TO MONTH
      AS RESULT
      FROM DUAL;
RESULT
----------
+000003-11
1 row selected.
```

7.3 + (POSITIVE)

语句

```
+ expr
```

说明

在expr标注+符号

使用示例

```
gSQL> SELECT +3 AS RESULT1, +(-3) AS RESULT2 FROM DUAL;

RESULT1 RESULT2

------- -------

      3      -3

1 row selected.
```

7.4 - (NEGATIVE)

语句

```
- expr
```

说明

在expr标注-符号

使用示例

```
gSQL> SELECT -3 AS RESULT1, -(-3) AS RESULT2 FROM DUAL;

RESULT1 RESULT2

------- -------

     -3       3

1 row selected.
```

7.5 - (SUBTRACTION)

语句

```
expr1 - expr2
```

说明

返回expr1减expr2的运算结果

减法运算的种类与结果类型如下图所示

详细内容参考**类型间转换（type conversion）**

expr1	expr2	结果类型
NATIVE INTEGER系列 • NATIVE_SMALLINT • NATIVE_INTEGER • NATIVE_BIGINT	NATIVE INTEGER系列 • NATIVE_SMALLINT • NATIVE_INTEGER • NATIVE_BIGINT	NATIVE_BIGINT
NUMBER	NUMBER	NUMBER

expr1	expr2	结果类型
NATIVE DOUBLE系列 • NATIVE_REAL • NATIVE_DOUBLE	NATIVE DOUBLE系列 • NATIVE_REAL • NATIVE_DOUBLE	NATIVE_DOUBLE

Table 7-6 数字型 - 运算

expr1	expr2	结果类型
DATE	DATE	NUMBER
DATE	数字类型	DATE
DATE	INTERVAL YEAR TO MONTH	DATE
DATE	INTERVAL DAY	DATE
DATE	INTERVAL DAY TO SECOND	TIMESTAMP
TIME	TIME	INTERVAL DAY TO SECOND
TIME	数字类型	TIME
TIME	INTERVAL YEAR TO MONTH	TIME
TIME	INTERVAL DAY TO SECOND	TIME
TIME WITH TIME ZONE	数字类型	TIME WITH TIME ZONE
TIME WITH TIME ZONE	INTERVAL YEAR TO MONTH	TIME WITH TIME ZONE
TIME WITH TIME ZONE	INTERVAL DAY TO SECOND	TIME WITH TIME ZONE

expr1	expr2	结果类型
TIMESTAMP	TIMESTAMP	INTERVAL DAY TO SECOND
TIMESTAMP	数字类型	TIMESTAMP
TIMESTAMP	INTERVAL YEAR TO MONTH	TIMESTAMP
TIMESTAMP	INTERVAL DAY TO SECOND	TIMESTAMP
TIMESTAMP WITH TIME ZONE	TIMESTAMP WITH TIME ZONE	INTERVAL DAY TO SECOND
TIMESTAMP WITH TIME ZONE	数字类型	TIMESTAMP WITH TIME ZONE
TIMESTAMP WITH TIME ZONE	INTERVAL YEAR TO MONTH	TIMESTAMP WITH TIME ZONE
TIMESTAMP WITH TIME ZONE	INTERVAL DAY TO SECOND	TIMESTAMP WITH TIME ZONE
INTERVAL YEAR TO MONTH	INTERVAL YEAR TO MONTH	INTERVAL YEAR TO MONTH（决定为可包含所有expr1与expr2的 interval范围的类型）
INTERVAL DAY TO SECOND	INTERVAL DAY TO SECOND	INTERVAL DAY TO SECOND（决定为可包含所有expr1与expr2的 interval范围的类型）
参考：**表中的INTERVAL类型包含的INTERVAL详细类型**		

Table 7-7 (DATETIME/INTERVAL) - 运算

使用示例

```
gSQL> SELECT
      TO_DATE( '2012-05-05' ) - TO_DATE( '2012-05-01' ) AS RESULT
      FROM DUAL;
RESULT
------
     4
1 row selected.


gSQL> SELECT TO_DATE( '2012-05-05' ) - 3 AS RESULT FROM DUAL;
RESULT
----------
2012-05-02
1 row selected.


gSQL> SELECT
      TO_DATE( '2012-05-05' ) - INTERVAL'01-02'YEAR TO MONTH AS RESULT
      FROM DUAL;
RESULT
----------
2011-03-05
1 row selected.


gSQL> SELECT
```

```
      INTERVAL'05-01'YEAR TO MONTH - INTERVAL'02-01'YEAR TO MONTH
      AS RESULT
      FROM DUAL;
RESULT
----------
+000003-00
1 row selected.


gSQL> SELECT INTERVAL'15 23:59:59.999999'DAY TO SECOND
           - INTERVAL'10 23:59:59.999999'DAY TO SECOND AS RESULT
      FROM DUAL;
RESULT
----------------------
+000005 00:00:00.000000
1 row selected.
```

7.6 / (DIVISION)

语句

```
expr1 / expr2
```

说明

返回expr1除以expr2的运算结果

除法运算的种类与结果类型如下图所示

详细内容参考**类型间转换（type conversion）**

expr1	expr2	结果类型
NATIVE INTEGER系列 • NATIVE_SMALLINT • NATIVE_INTEGER • NATIVE_BIGINT	NATIVE INTEGER系列 • NATIVE_SMALLINT • NATIVE_INTEGER • NATIVE_BIGINT	NATIVE_DOUBLE
NUMBER	NUMBER	NUMBER
NATIVE_DOUBLE	NATIVE_DOUBLE	NATIVE_DOUBLE

Table 7-8 数字型 / 运算

expr1	expr2	结果类型
INTERVAL YEAR TO MONTH	数字类型	INTERVAL YEAR TO MONTH（由interval type决定结果类型）
INTERVAL DAY TO SECOND	数字类型	INTERVAL DAY TO SECOND（由interval type决定结果类型）
参考：**表中的INTERVAL类型包含的INTERVAL详细类型**		

Table 7-9 (DATETIME/INTERVAL) / 运算

使用示例

```
gSQL> SELECT INTERVAL'20-10'YEAR TO MONTH / 2 AS RESULT FROM DUAL;
RESULT
----------
+000010-05
1 row selected.

gSQL> SELECT   INTERVAL'02 02:04:06.800000'DAY TO SECOND / 2 AS RESULT
      FROM DUAL;
RESULT
```

```
-----------------------

+000001 01:02:03.400000

1 row selected.
```

7.7 || (CONCATENATE)

语句

```
str1 || str2
```

说明

CONCATENATE返回连接str1与str2的字符串

str1和str2中一个为null时返回非null的其余strstr1和str2均为null时返回NULL

为可转换为character string类型或binary string类型的类型

详细内容参考**类型间转换（type conversion）**

是**CONCAT**, **CONCATENATE**的alias

结果类型如下图所示

Data type	CHAR	VARCHAR	LONG VARCHAR
CHAR	CHAR	VARCHAR	LONG VARCHAR
VARCHAR	VARCHAR	VARCHAR	LONG VARCHAR
LONG VARCHAR	LONG VARCHAR	LONG VARCHAR	LONG VARCHAR
Data type	BINARY	VARBINARY	LONG VARBINARY

Data type	CHAR	VARCHAR	LONG VARCHAR
BINARY	BINARY	VARBINARY	LONG VARBINARY
VARBINARY	VARBINARY	VARBINARY	LONG VARBINARY
LONG VARBINARY	LONG VARBINARY	LONG VARBINARY	LONG VARBINARY

Table 7-10 || (CONCATENATE)的结果类型

使用示例

```
gSQL> SELECT 'DATA' || 'BASE' AS RESULT1,
             'DATA' || NULL   AS RESULT2,
               NULL || NULL   AS RESULT3
       FROM DUAL;
RESULT1  RESULT2 RESULT3
-------- ------- -------
DATABASE DATA    null
1 row selected.
```

7.8 ABS

语句

```
ABS( num )
```

说明

ABS返回num的绝对值

num可以是数字类型或可转换为数字的类型

num为NULL时返回NULL

使用示例

```
gSQL> SELECT ABS(-1) AS RESULT1, ABS(1) AS RESULT2 FROM DUAL;

RESULT1 RESULT2

------- -------

      1       1

1 row selected.
```

7.9 ACOS

语句

```
ACOS( num )
```

说明

ACOS函数返回num的arc cosine值

参数num应为-1以上1以下的值

num为NULL时返回NULL

返回0 ~ pi之间的弧度值

使用示例

```
gSQL> SELECT ACOS( 1 ) FROM DUAL;

ACOS( 1 )

---------

        0

1 row selected.
```

DDDATE

语句

```
ADDDATE( date, INTERVAL expr unit  )

ADDDATE( expr, days )
```

说明

ADDDATE返回输入的第一个参数加上第二个参数的运算结果

第一个参数可以是DATE, TIMESTAMP, TIMESTAMP WITH TIME ZONE类型第二个参数可以是INTERVAL或数字类型

两个输入参数中只要有一个为NULL则结果值也为NULL

结果类型与**(DATETIME/INTERVAL) + 运算**相同

使用示例

```
gSQL> SELECT ADDDATE( TO_DATE( '2012-12-12', 'YYYY-MM-DD' ), 1 ) AS RESULT
        FROM DUAL;
RESULT
----------
2012-12-13
```

```
1 row selected.


gSQL> SELECT ADDDATE( TO_DATE( '2012-11-11', 'YYYY-MM-DD' ),
                      INTERVAL'01-01'YEAR TO MONTH ) AS RESULT
      FROM DUAL;
RESULT
----------
2013-12-11
1 row selected.
```

7.10 ADDTIME

语句

```
ADDTIME( expr1, expr2 )
```

说明

ADDTIME返回expr1加上输入的expr2的运算结果

expr1可以是TIME,TIME WITH TIME ZONETIMESTAMPTIMESTAMP WITH TIME ZONE TYPEexpr2可以是INTERVAL DAY TO SECOND TYPE

expr1或expr2为NULL则结果值为NULL

结果类型与**(DATETIME/INTERVAL) + 运算**相同

使用示例

```
gSQL> SELECT
      ADDTIME( TO_TIMESTAMP( '2001-05-05 06:00:00',
                             'YYYY-MM-DD HH24:MI:SS' ),
               INTERVAL'0 00:06:06.666666'DAY TO SECOND ) AS RESULT
```

```
      FROM DUAL;

RESULT

--------------------------

2001-05-05 06:06:06.666666

1 row selected.
```

7.11 ADD_MONTHS

语句

```
ADD_MONTHS( date, number )
```

说明

ADD_MONTHS返回date加上与number数字相同的月份的值

如果ADD_MONTHS的运算结果大于当月的最后一天则结果值调整为当月的最后一天

date可以是DATETIMESTAMPTIMESTAMP WITH TIME ZONE类型number可以是数字类型

两个中只要有一个为NULL则结果值为NULL

结果类型与输入date类型无关始终为DATE类型

使用示例

```
gSQL> SELECT
      ADD_MONTHS( TO_DATE( '2001-07-31', 'YYYY-MM-DD' ), 1 ) AS RESULT1,
      ADD_MONTHS( TO_DATE( '2001-07-31', 'YYYY-MM-DD' ), 2 ) AS RESULT2
      FROM DUAL;
RESULT1    RESULT2
---------- ----------
```

```
2001-08-31 2001-09-30

1 row selected.
```

7.12 ASCII

语句

```
ASCII( char )
```

说明

以十进制返回char的第一个字符的database character set code

char可以是与CHARACTERCHARACTER VARYINGCHARACTER LONG VARYING相同的字符类型或可转换为字符类型的类型返回类型是NUMBER

char为NULL时返回NULL

使用示例

```
gSQL> SELECT ASCII( 'G' ) AS RESULT FROM DUAL;
RESULT
------
    71
1 row selected.
```

7.13 ASIN

语句

```
ASIN( num )
```

说明

ASIN函数返回num的arc sin值

参数num应为-1以上1以下的值

num为NULL时返回NULL

返回-pi/2 ~ pi/2 之间的弧度值

使用示例

```
gSQL> SELECT ASIN( 0 ) FROM DUAL;
ASIN( 0 )
---------
        0
1 row selected.ATAN
```

语句

```
ATAN( num )
```

说明

ATAN函数返回num的arc tangent值

num值范围没有限制返回-pi/2 ~ pi/2之间的弧度值

num为NULL时返回NULL

使用示例

```
gSQL> SELECT ATAN(0.5) FROM DUAL;

       ATAN(0.5)

----------------

.463647609000806

1 row selected.
```

7.14 ATAN2

语句

```
ATAN2( num1, num2 )
```

说明

ATAN2函数返回num1与num2的arc tangent值

参数num1值的范围没有限制返回 -pi ~ pi 之间的弧度值

num1或num2中只要有一个为NULL则返回NULL

使用示例

```
gSQL> SELECT ATAN2(1,2) FROM DUAL;

      ATAN2(1,2)

----------------

.463647609000806

1 row selected.
```

7.15 AVG

语句

```
AVG( [ ALL | DISTINCT ] num )
```

说明

aggregation函数用于获取expr的平均值

指定ALL时执行对所有值的aggregation

指定DISTINCT时执行对排除重复值的aggregation

不指定ALL或DISTINCT时处理方式与指定ALL相同

使用示例

```
gSQL> SELECT AVG(c1) FROM t1;


AVG(C1)
-------
      2


1 row selected.
```

7.16 AVG() OVER

语句

```
AVG ( expr ) OVER < window name or specification >
```

关于< window name or specification >的详细内容请参阅**window clause**

说明

Window function AVG是计算expr的平均值的函数

NULL值不在计算范围内

使用示例

```
gSQL> SELECT min_price AS "MIN_PRICE"
             , AVG( min_price ) OVER ( ORDER BY min_price ) AS "AVG"
        FROM product_information
       WHERE supplier_id = 102050;
```

```
MIN_PRICE              AVG
--------- ----------------
       73               73
      247              160
      731 350.333333333333
     null 350.333333333333
     null 350.333333333333

5 rows selected.
```

7.17 BITAND

语句

```
BITAND( num1, num2 )
```

说明

返回num1和num2的bit的AND运算结果

输入参数可以是NATIVE_SMALLINTNATIVE_INTEGERNATIVE_BIGINT或可转换为NATIVE_BIGINT的类型

转换为NATIVE_BIGINT类型时会对小数点进行TRUNCATE处理

即使输入参数值有一个为 NULL结果值也为NULL

结果类型为NATIVE_BIGINT

使用示例

```
gSQL> SELECT BITAND( 5, 3 ) AS RESULT FROM DUAL;

RESULT

------

     1

1 row selected.
```

7.18 BITNOT

语句

```
BITNOT( num )
```

说明

返回num的bit的NOT运算结果

输入参数可以是NATIVE_SMALLINTNATIVE_INTEGERNATIVE_BIGINT或可转换为NATIVE_BIGINT的类型

转换为NATIVE_BIGINT类型时会对小数点进行TRUNCATE处理

若输入参数为NULL则结果值也为NULL

结果类型如下

- 输入参数为NATIVE_SMALLINT时NATIVE_SMALLINT
- 输入参数为NATIVE_INTEGER时NATIVE_INTEGER
- 输入参数为NATIVE_BIGINT时NATIVE_BIGINT

使用示例

```
gSQL> SELECT BITNOT( 5 ) AS RESULT FROM DUAL;
RESULT
```

```
------

    -6

1 row selected.
```

7.19 BITOR

语句

```
BITOR( num1, num2 )
```

说明

返回num1与num2的bit的OR运算结果

输入参数可以是NATIVE_SMALLINTNATIVE_INTEGERNATIVE_BIGINT或可转换为NATIVE_BIGINT的类型

转换为NATIVE_BIGINT类型时会对小数点进行TRUNCATE处理

即使输入参数值有一个为 NULL结果值也为NULL

结果类型为NATIVE_BIGINT

使用示例

```
gSQL> SELECT BITOR( 5, 3 ) FROM DUAL;


BITOR( 5, 3 )
-------------
            7
1 row selected.
```

7.20 BITXOR

语句

```
BITXOR( num1, num2 )
```

说明

返回num1与num2的bit的XOR运算结果

输入参数可以是NATIVE_SMALLINTNATIVE_INTEGERNATIVE_BIGINT或可转换为NATIVE_BIGINT的类型

转换为NATIVE_BIGINT类型时会对小数点进行TRUNCATE处理

即使输入参数值有一个为 NULL结果值也为NULL

结果类型为NATIVE_BIGINT

使用示例

```
gSQL> SELECT BITXOR( 5, 3 ) FROM DUAL;
BITXOR( 5, 3 )
--------------
             6
1 row selected.
```

7.21 BIT_LENGTH

语句

```
BIT_LENGTH( str )
```

说明

BIT_LENGTH返回str的bit数

str若为NULL则返回NULL

示例

```
gSQL> SELECT BIT_LENGTH( 'LIKE' ) AS RESULT FROM DUAL;
     RESULT
     ---------------
             32
    1 row selected.
```

7.22 BYTE_LENGTH

语句

```
BYTE_LENGTH( str )
```

说明

是OCTET_LENGTH的alias

参考：**OCTET_LENGTH**, **LENGTHB**

使用示例

- Multi byte character set (示例: UTF8): 1 byte character

```
gSQL> SELECT BYTE_LENGTH( 'OCTET_LENGTH' ) AS RESULT_1BYTE_CHARACTERS
        FROM DUAL;
RESULT_1BYTE_CHARACTERS
-----------------------
                    12
1 row selected.
```

- Multi byte character set (示例: UTF8): 2 byte character

```
gSQL> SELECT BYTE_LENGTH( 'αβ' ) AS RESULT_2BYTE_CHARACTERS FROM DUAL;

RESULT_2BYTE_CHARACTERS

-----------------------

                      4

1 row selected.
```

7.23 CASE2

语句

```
CASE2( condition1, result1
      [, condition2, result2
       , ...
       , conditionN, resultN ]
      [, default ] )
```

说明

CASE2按照顺序判断condition（条件）

对比结果为FALSE时判断到显示TRUE为止

对比结果为TRUE时返回对应的result并不继续进行判断

对比结果均为FALSE则返回default省略default时返回NULL

如果result是多个类型将根据**结果类型组合规则**决定result类型

CASE2的处理方式与CASE相同

- CASE2(condition1, res1, condition2, res2)

```
CASE WHEN condition1 THEN res1
     WHEN condition2 THEN res2
```

```
    ELSE NULL
  END
```

- CASE2(condition1, res1, condition2, res2, default)

```
CASE WHEN condition1 THEN res1
    WHEN condition2 THEN res2
    ELSE default
  END
```

使用示例

```
gSQL> SELECT I1,
        CASE2( I1 = 1, 'ONE', I1 = 2, 'TWO' ) AS CASE2_RESULT1,
        CASE2( I1 = 1, 'ONE', I1 = 2, 'TWO', 'NUMBER' ) AS CASE2_RESULT2
      FROM T1;
I1 CASE2_RESULT1 CASE2_RESULT2
-- ------------- -------------
 1 ONE           ONE
 2 TWO           TWO
 3 null          NUMBER
3 rows selected.
```

7.24 CBRT

语句

```
CBRT( num )
```

说明

返回num的立方根

num为NULL时结果值也为NULL

使用示例

```
gSQL> SELECT CBRT( 27 ) FROM DUAL;
CBRT( 27 )
----------
         3
1 row selected.
```

7.25 CEIL

语句

```
CEIL( num )

CEILING( num )
```

说明

CEIL函数返回大于或等于num的最小整数

若num为NULL则返回NULL

使用示例

```
gSQL> SELECT CEIL( 3.5 ) AS RESULT1, CEIL( -3.5 ) AS RESULT2 FROM DUAL;

RESULT1 RESULT2

------- -------

      4      -3

1 row selected.
```

7.26 CHAR_LENGTH

语句

```
CHAR_LENGTH( str )

CHARACTER_LENGTH( str )
```

说明

CHAR_LENGTH对str返回根据character set的字符数量

str可以是CHARACTERCHARACTER VARYINGCHARACTER LONG VARYING等字符类型或可转换为字符类型的类型返回类型为NATIVE_BIGINT

str的类型为CHARACTER类型时包含空字符（trailing blank）

str为NULL时返回NULL

是**LENGTH**的alias

使用示例

Multi byte character set: (例:UTF8)

```
gSQL> SELECT CHAR_LENGTH( 'αβ-SUMMER' ) AS RESULT FROM DUAL;
```

```
 RESULT

 ---------------

          9

1 row selected.
```

7.27 CHR

语句

```
CHR（num）
```

说明

返回对应num的database character set code中的character

num为数字类型

若num为NULL则返回NULL

结果类型为VARCHAR

使用示例

```
gSQL> SELECT CHR(71) FROM DUAL;

CHR(71)

-------

G

1 row selected.
```

7.28 CLOCK_DATE

语句

```
CLOCK_DATE()
```

说明

调用函数时每次均返回当前日期（DATE type）值

返回当前日期的函数之间有如下差异

- TRANSACTION_DATE()：事务内的所有日期值均相同
- STATEMENT_DATE()：一个SQL语句内的所有日期值均相同
- CLOCK_DATE()：调用函数时每次均获取当前日期值

使用示例

每条row为不同的值

```
gSQL> SELECT CLOCK_DATE() FROM t1;


CLOCK_DATE()
```

```
------------

2013-12-12

2013-12-12

2013-12-13


3 rows selected.
```

7.29 CLOCK_LOCALTIME

语句

```
CLOCK_LOCALTIME()
```

说明

调用函数时每次均获取无TIME ZONE的当前时间（TIME WITHOUT TIME ZONE type）值

获取当前时间的函数之间有如下差异

- TRANSACTION_LOCALTIME()：事务内的所有时间值均相同
- STATEMENT_LOCALTIME()：一个SQL语句内的所有时间值均相同
- CLOCK_LOCALTIME()：调用函数时每次均获取当前时间值

使用示例

每条row可以是不同的时间值

```
gSQL> SELECT CLOCK_LOCALTIME() FROM t1;


CLOCK_LOCALTIME()
```

```
-----------------

14:42:05.470757

14:42:05.470759

14:42:05.470759


3 rows selected.
```

7.30 CLOCK_LOCALTIMESTAMP

语句

```
CLOCK_LOCALTIMESTAMP()
```

说明

调用函数时每次均获取无TIME ZONE的当前TIMESTAMP（TIMESTAMP WITHOUT TIME ZONE type）值

获取当前TIMESTAMP 的函数之间有如下差异

- TRANSACTION_LOCALTIMESTAMP()：事务内的所有TIMESTAMP值均相同
- STATEMENT_LOCALTIMESTAMP()：一个SQL语句内的所有TIMESTAMP值均相同
- CLOCK_LOCALTIMESTAMP()：调用函数时每次均获取当前TIMESTAMP值

使用示例

每条row可以是不同的值

```
gSQL> SELECT CLOCK_LOCALTIMESTAMP() FROM t1;
```

```
CLOCK_LOCALTIMESTAMP()

--------------------------

2013-12-12 14:46:17.309206

2013-12-12 14:46:17.309209

2013-12-12 14:46:17.309209
```

7.31 CLOCK_TIME

语句

```
CLOCK_TIME()
```

说明

调用函数时每次均获取有TIME ZONE的当前时间（TIME WITH TIME ZONE type）值

获取当前时间的函数之间有如下差异

- TRANSACTION_TIME()：事务内的所有时间值均相同
- STATEMENT_LTIME()：一个SQL语句内的所有时间值均相同
- CLOCK_TIME()：调用函数时每次均获取当前时间值

使用示例

每条row为不同的值

```
gSQL> SELECT CLOCK_TIME() FROM t1;


CLOCK_TIME()
```

```
----------------------

14:48:21.052324 +09:00

14:48:21.052326 +09:00

14:48:21.052327 +09:00


3 rows selected.
```

7.32 CLOCK_TIMESTAMP

语句

```
CLOCK_TIMESTAMP()
```

说明

调用函数时每次均获取有TIME ZONE的当前TIMESTAMP（TIMESTAMP WITH TIME ZONE type）值

获取当前TIMESTAMP的函数之间的差异如下

- TRANSACTION_TIMESTAMP()：事务内的所有TIMESTAMP值均相同
- STATEMENT_TIMESTAMP()：一个SQL语句内的所有TIMESTAMP值均相同
- CLOCK_TIMESTAMP()：调用函数时每次均获取当前TIMESTAMP值

使用示例

每条row为不同的值

```
gSQL> SELECT CLOCK_TIMESTAMP() FROM t1;
```

```
CLOCK_TIMESTAMP()

---------------------------------

2013-12-12 14:49:45.051709 +09:00

2013-12-12 14:49:45.051714 +09:00

2013-12-12 14:49:45.051714 +09:00


3 rows selected.
```

7.33 COALESCE

语句

```
COALESCE( expr1, ..., exprN )
```

说明

返回expr list中非null的第一个expr

expr list均为null时返回null

expr应为2个以上

expr list为多个类型时根据结果类型组合规则决定结果类型

- 可以使用CASE相同地表示COALESCE
  - COALESCE(expr1, expr2)

```
CASE WHEN expr1 IS NOT NULL THEN expr1
       ELSE expr2
  END
```

  - COALESCE(expr1, expr2, ..., exprN)

```
CASE WHEN expr1 IS NOT NULL THEN expr1
       ELSE COALESCE( expr2, ..., exprN )
```

```
END
```

使用示例

```
gSQL> SELECT COALESCE( NULL, 1, 2 ) FROM DUAL;

COALESCE( NULL, 1, 2 )

----------------------

                     1

1 row selected.


gSQL> SELECT COALESCE( NULL, NULL, NULL ) FROM DUAL;

COALESCE( NULL, NULL, NULL )

----------------------------

null

1 row selected.
```

7.34 CONCAT

语句

```
CONCAT( str1, str2, ... )
```

说明

是|| (CONCATENATE) 的alias

作为CONCAT函数的argument可设置2~254个

详细内容参考：**|| (CONCATENATE)**, **CONCATENATE**

使用示例

```
gSQL> SELECT CONCAT( 'DATA', 'BASE' ) AS RESULT FROM DUAL;

RESULT

--------

DATABASE

1 row selected.
```

7.35 CONCATENATE

语句

```
CONCATENATE( str1, str2, ... )
```

说明

是|| (CONCATENATE)的alias

作为CONCATENATE函数的argument可设置2~254个

详细内容参考：**CONCAT, || (CONCATENATE)**

使用示例

```
gSQL> SELECT CONCATENATE( 'DATA', 'BASE' ) AS RESULT FROM DUAL;

RESULT

--------

DATABASE

1 row selected.
```

7.36 CORR() OVER

语句

```
CORR( expr1, expr2 ) OVER < window name or specification >
```

关于< window name or specification >的详细内容请参阅**window clause**

说明

Window function CORR为计算expr对相关系数 (coefficient of correlation)的函数

如果expr1或expr2为NULL则不计算

当expr对的row数量为一个以下时则返回NULL结果

使用示例

```
gSQL> SELECT employee_id, TO_CHAR( hire_date, 'YYYY' ) AS hire_date,
salary,
             CORR( TO_CHAR( hire_date, 'YYYY' ), salary ) OVER ( ORDER BY
employee_id ) AS corr
      FROM employees
      WHERE department_id = 60;
```

```
EMPLOYEE_ID HIRE_DATE SALARY              CORR

----------- --------- ------ -----------------

        103 1990        9000              null

        104 1991        6000                -1

        105 1997        4800 -.805837379342809

        106 1998        4800 -.840210805972693

        107 1999        4200 -.875185734200534


5 rows selected.
```

7.37 COS

语句

```
COS(num)
```

说明

返回num的COSINE值

参数值num为NULL时结果值也为NULL

使用示例

```
gSQL> SELECT COS( 0 ) FROM DUAL;
COS( 0 )
--------
       1
1 row selected.
```

7.38 COT

语句

```
COT(num)
```

说明

返回num的COTANGENT值

参数值num为NULL时结果值也为NULL

使用示例

```
gSQL> SELECT COT( 1 ) FROM DUAL;
        COT( 1 )
----------------
.642092615934331
1 row selected.
```

7.39 COUNT

语句

```
COUNT( [ ALL | DISTINCT ] expr )
```

说明

作为aggregation函数获取expr不是NULL值的row的条数

指定ALL时对所有值执行聚合操作（aggregation）

指定DISTINCT时对排除重复的值执行聚合操作

未指定ALL或DISTINCT时与指定ALL的处理方法相同

使用示例

```
gSQL> SELECT COUNT(c1) FROM t1;


COUNT(C1)

---------

        3


1 row selected.
```

7.40 COUNT() OVER

语句

```
COUNT ( expr ) OVER < window name or specification >
```

关于< window name or specification >的详细内容请参阅**window clause**

说明

Window function COUNT是计算row数量的函数

不计算NULL值

使用示例

```
gSQL> SELECT min_price AS "MIN_PRICE"
             , COUNT( min_price ) OVER ( ORDER BY min_price ) AS "COUNT"
        FROM product_information
       WHERE supplier_id = 102050;


MIN_PRICE COUNT
--------- -----
       73     1
```

```
      247     2

      731     3

     null     3

     null     3


5 rows selected.
```

7.41 COUNT(*)

语句

```
COUNT(*)
```

说明

作为聚合操作函数获取row的数量

不单独指定expression因此与值是否为NULL无关

使用示例

```
gSQL> SELECT COUNT(*) FROM t1;


COUNT(*)
--------
       4


1 row selected.
```

7.42 COUNT(*) OVER

语句

```
COUNT(*) OVER < window name or specification >
```

关于< window name or specification >的详细内容请参阅**window clause**

说明

Window function COUNT(*)是计算row数量的函数

因不单独指定表达式所以和值是否是NULL无关

使用示例

```
gSQL> SELECT min_price AS "MIN_PRICE"
             , COUNT(*) OVER ( ORDER BY min_price ) AS "COUNT(*)"
        FROM product_information
       WHERE supplier_id = 102050;


MIN_PRICE COUNT(*)
--------- --------
       73        1
```

```
      247         2

      731         3

     null         5

     null         5


5 rows selected.
```

7.43 COVAR_POP() OVER

语句

```
COVAR_POP( expr1, expr2 ) OVER < window name or specification >
```

关于< window name or specification >的详细内容请参阅**window clause**

说明

Window function COVAR_POP是计算expr对的总体协方差 (population covariance)的函数

如果expr1或expr2为NULL则不计算

当expr对的row数量为一个以下时则返回结果为0

使用示例

```
gSQL> SELECT employee_id, TO_CHAR( hire_date, 'YYYY' ) AS hire_date,
salary,
             COVAR_POP( TO_CHAR( hire_date, 'YYYY' ), salary ) OVER
( ORDER BY employee_id ) AS covar_pop
      FROM employees
      WHERE department_id = 60;
```

```
EMPLOYEE_ID HIRE_DATE SALARY COVAR_POP
----------- --------- ------ ---------
        103 1990        9000         0
        104 1991        6000      -750
        105 1997        4800     -4400
        106 1998        4800     -5100
        107 1999        4200     -5640

5 rows selected.
```

7.44 COVAR_SAMP() OVER

语句

```
COVAR_SAMP( expr1, expr2 ) OVER < window name or specification >
```

关于< window name or specification >的详细内容参考**window clause**

说明

Window function COVAR_SAMP是计算expr对的样本协方差 (sample covariance)的函数

expr1或expr2为NULL时不计算

expr对的row数量为一个以下时返回结果为NULL

使用示例

```
gSQL> SELECT employee_id, TO_CHAR( hire_date, 'YYYY' ) AS hire_date,
salary,
             COVAR_SAMP( TO_CHAR( hire_date, 'YYYY' ), salary ) OVER
( ORDER BY employee_id ) AS covar_samp
      FROM employees
      WHERE department_id = 60;
```

```
EMPLOYEE_ID HIRE_DATE SALARY COVAR_SAMP
----------- --------- ------ ----------
        103 1990        9000       null
        104 1991        6000      -1500
        105 1997        4800      -6600
        106 1998        4800      -6800
        107 1999        4200      -7050

5 rows selected.
```

7.45 CUME_DIST() OVER

语句

```
CUME_DIST( ) OVER < window name or specification >
```

关于< window name or specification >的详细内容参考 **window clause**

说明

Window function CUME_DIST根据当前row值的相对位置计算累积分布

CUME_DIST函数的结果是从0到1之间的数字

Row值相同时则以其中最大的累积分布值返回相同结果

无法使用window frame

使用示例

```
gSQL> SELECT department_id, salary,
             CUME_DIST() OVER ( ORDER BY salary ) AS cume_dist
      FROM employees
      WHERE department_id = 60;
```

```
DEPARTMENT_ID SALARY CUME_DIST
------------- ------ ---------
           60   4200        .2
           60   4800        .6
           60   4800        .6
           60   6000        .8
           60   9000         1

5 rows selected.
```

7.46 CURRENT_CATALOG

语句

```
CURRENT_CATALOG [()]
```

说明

获取catalog name (database名称)

使用示例

```
gSQL> SELECT CURRENT_CATALOG FROM dual;


CURRENT_CATALOG

---------------

TEST_DB


1 row selected.
```

7.47 CURRENT_DATE

语句

```
CURRENT_DATE [()]

STATEMENT_DATE()
```

说明

获取当前日期（DATE type）值

CURRENT_DATE为SQL标准函数

获取当前日期的函数之间有如下差异

- TRANSACTION_DATE()：事务内的所有日期值均相同
- CURRENT_DATE, STATEMENT_DATE()：一个SQL语句内的所有日期值均相同
- CLOCK_DATE()：调用函数时每次均获取当前日期值

使用示例

```
gSQL> SELECT CURRENT_DATE FROM t1;
```

```
CURRENT_DATE

------------

2013-12-12

2013-12-12

2013-12-12


3 rows selected.
```

7.48 CURRENT_SCHEMA

语句

```
CURRENT_SCHEMA [()]
```

说明

获取用户的当前SCHEMA

使用示例

```
gSQL> SELECT CURRENT_SCHEMA FROM dual;


CURRENT_SCHEMA

--------------

PUBLIC


1 row selected.
```

7.49 CURRENT_TIME

语句

```
CURRENT_TIME [()]

STATEMENT_TIME()
```

说明

以会话时间为准获取当前TIME WITH TIME ZONE type值

CURRENT_TIME为SQL标准函数

获取当前日期的函数之间有如下差异

- TRANSACTION_TIME()：事务内的所有时间值均相同
- CURRENT_TIME, STATEMENT_TIME()：一个SQL语句内的所有时间值均相同
- CLOCK_TIME()：调用函数时每次均获取当前时间值

使用示例

所有row拥有相同的时间值

```
gSQL> SELECT CURRENT_TIME FROM t1;


CURRENT_TIME

----------------------

16:27:10.116396 +09:00

16:27:10.116396 +09:00

16:27:10.116396 +09:00


3 rows selected.
```

7.50 CURRENT_TIMESTAMP

语句

```
CURRENT_TIMESTAMP [()]

STATEMENT_TIMESTAMP()
```

说明

以会话时间为准获取TIMESTAMP WITH TIME ZONE type值

CURRENT_TIMESTAMP为SQL标准函数

获取当前TIMESTAMP的函数之间有如下差异

- TRANSACTION_TIMESTAMP()：事务内的所有TIMESTAMP值均相同
- CURRENT_TIMESTAMP, STATEMENT_TIMESTAMP()：一个SQL语句内的所有TIMESTAMP值均相同
- CLOCK_TIMESTAMP()：调用函数时每次均获取当前TIMESTAMP值

使用示例

所有row拥有相同的值

```
gSQL> SELECT CURRENT_TIMESTAMP FROM t1;


CURRENT_TIMESTAMP

---------------------------------

2013-12-12 16:34:55.649632 +09:00

2013-12-12 16:34:55.649632 +09:00

2013-12-12 16:34:55.649632 +09:00


3 rows selected.
```

7.51 CURRENT_USER

语句

```
CURRENT_USER [()]
```

说明

返回当前用户

以如下三种形式管理用户信息

- Logon user：执行login的用户维持到关闭连接
- Session user：与最初的logon user相同但可以用SET SESSION AUTHORIZATION语句进行变更
- Current user：一般情况下与session user相同但使用PSM, View等时为了控制访问等在系统内部暂时变更
    - Session user与current user的差异和unix系统的real user与effective user的差异类似

使用示例

```
% gsql sys gliese
```

```
gSQL> SET SESSION AUTHORIZATION test;


Session set.


gSQL> SELECT
        LOGON_USER() AS result1,
        SESSION_USER() AS result2,
        CURRENT_USER() AS result3
      FROM DUAL;


RESULT1 RESULT2 RESULT3
------- ------- -------
SYS     TEST    TEST


1 row selected.
```

7.52 CURRVAL

语句

```
seq_name.CURRVAL

CURRVAL(seq_name)
```

说明

获取序列对象的当前值

至少设置一次NEXTVAL(seq_name)等序列值

使用示例

```
gSQL> SELECT seq.CURRVAL FROM dual;


SEQ.CURRVAL

-----------

          1


1 row selected.
```

7.53 DATEADD

语句

```
DATEADD( datepart, number, date )
```

说明

返回date指定的datepart加上number的值

number为小数点时不进行四舍五入

date为DATETIMESTAMPTIMESTAMP WITH TIME ZONETIMETIME WITH TIME ZONE类型

number或date为NULL时结果值也为NULL

结果类型返回与作为返回的date类型相同的类型

datepart	说明
YEAR	年
QUARTER	季度
MONTH	月
DAYOFYEAR	日
DAY	日

WEEK	周
WEEKDAY	周工作日
HOUR	时
MINUTE	分
SECOND	秒
MILLISECOND	毫秒
MICROSECOND	微妙

Table 7-11 datepart中可使用的格式字符串

使用示例

```
gSQL> SELECT
      DATEADD( YEAR, 1, TO_DATE( '2013-05-14', 'YYYY-MM-DD' ) ) AS RESULT
      FROM DUAL;
RESULT
----------
2014-05-14
1 row selected.

gSQL> SELECT
      DATEADD( MONTH, 13, TO_DATE('2013-05-14', 'YYYY-MM-DD') ) AS RESULT
```

```
      FROM DUAL;
RESULT
----------
2014-06-14
1 row selected.


gSQL> SELECT
      DATEADD( DAY, 397, TO_DATE('2013-05-14', 'YYYY-MM-DD') ) AS RESULT
      FROM DUAL;
RESULT
----------
2014-06-15
1 row selected.
```

7.54 DATEDIFF

语句

```
DATEDIFF( datepart, startdate, enddate )
```

说明

向指定datepart返回在enddate减去startdate的值

startdate与enddate为DATETIMESTAMPTIMESTAMP WITH TIME ZONETIME TYPE

startdate或enddate为NULL时结果值也为NULL

结果类型为NUMBER

datepart	说明
YEAR	年
QUARTER	季度
MONTH	月
DAYOFYEAR	日
DAY	周工作日
HOUR	时

MINUTE	分
SECOND	秒
MILLISECOND	毫秒
MICROSECOND	微妙

Table 7-12 datepart中可使用的格式字符串

使用示例

```
gSQL>  SELECT
       DATEDIFF( YEAR,
                 TO_DATE( '2013-05-14', 'YYYY-MM-DD' ),
                 TO_DATE( '2014-06-15', 'YYYY-MM-DD' ) ) AS RESULT
       FROM DUAL;
RESULT
------
     1
1 row selected.


gSQL> SELECT
      DATEDIFF( MONTH,
                TO_DATE( '2013-05-14', 'YYYY-MM-DD' ),
                TO_DATE( '2014-06-15', 'YYYY-MM-DD' ) ) AS RESULT
```

```
      FROM DUAL;
RESULT
------
    13
1 row selected.


gSQL> SELECT
      DATEDIFF( DAY,
                TO_DATE( '2013-05-14', 'YYYY-MM-DD' ),
                TO_DATE( '2014-06-15', 'YYYY-MM-DD' ) ) AS RESULT
      FROM DUAL;
RESULT
------
   397
1 row selected.
```

7.55 DATE_ADD

语句

```
DATE_ADD( date, INTERVAL expr unit  )
```

说明

是与**17.10 ADDDATE**(date, INTERVAL expr unit)相同的函数

使用示例

```
gSQL> SELECT
      DATE_ADD( TO_DATE( '2012-01-02', 'YYYY-MM-DD' ),
                INTERVAL '2-2' YEAR TO MONTH ) AS RESULT
      FROM DUAL;
RESULT
----------
2014-03-02
1 row selected.
```

7.56 DATE_PART

语句

```
DATE_PART( field, datetime )
```

说明

DATE_PART函数的结果值与EXTRACT函数相同它在输入的datetime类型中搜索指定的field并返回

参数field只能为字符字面常量可以将YEAR, MONTH, DAY, HOUR, MINUTE, SECOND, TIMEZONE_HOUR, TIMEZONE_MINUTE指定为字符字面常量

参数datetime可以是DATE, TIMESTAMP, TIMESTAMP WITH TIME ZONE, TIME, TIME WITH TIME ZONE, INTERVAL类型

field不在datetime范围内时报错

另外为DATE类型时field只能为YEAR, MONTH, DAY其余报错

若datetime为NULL则返回NULL

返回类型为NUMBER

详细内容参考**EXTRACT**

使用示例

```
gSQL> SELECT
      DATE_PART( 'DAY', TO_DATE( '2012-01-02', 'YYYY-MM-DD' ) ) AS RESULT
      FROM DUAL;
RESULT
------
     2
1 row selected.


gSQL> SELECT
      DATE_PART( 'YEAR', INTERVAL'9-11'YEAR TO MONTH ) AS RESULT
      FROM DUAL;
RESULT
------
     9
1 row selected.
```

7.57 DECODE

语句

```
DECODE( expr, comparison_expr1, result1
          [, comparison_expr2, result2
           , ...
           , comparison_exprN, resultN ]
          [, default ] )
```

说明

按DECODE语句中的顺序通过equal运算判断expr与comparison_expr

对比结果为FALSE时持续判断直到结果为TRUE

对比结果为TRUE时返回对应的result并停止判断

expr与comparison_expr相同或expr与comparison_expr均为null时（null = null）判断为TRUE因此返回对应的result

如果判断结果均为FALSE则返回default省略default时返回NULL

- 对比expr与comparison_expr

  所有exprcomparison_expr1...comparison_exprN转换为comparison_expr1（第一个comparison_expr）的数据类型后进行对比

  comparison_expr1（第一个comparison_expr）为字符类型与数字类型时均决定为可包含

exprcomparison_expr1...comparison_exprN中描述的类型范围的类型

exprcomparison_expr1...comparison_exprN中描述的类型均为CHAR类型时以VARCHAR类型执行对比

- 结果类型

  结果类型为result1（第一个result）的数据类型

  如果result1（第一个result）的数据类型为数字类型与字符类型则均为可包含result1...resultN中描述的类型范围的类型

  result1（第一个result）为CHAR类型或NULL时结果类型为VARCHAR

可以使用CASE相同地表示DECODE

- DECODE(expr, comp_expr1, res1, comp_expr2, res2)

```
CASE WHEN (expr = comp_expr1) OR (expr IS NULL AND comp_expr1 IS NULL )
THEN res1
     WHEN (expr = comp_expr2) OR (expr IS NULL AND comp_expr2 IS NULL )
THEN res2
     ELSE NULL
  END
```

- DECODE(expr, comp_expr1, res1, comp_expr2, res2, default)

```
CASE WHEN (expr = comp_expr1) OR (expr IS NULL AND comp_expr1 IS NULL )
THEN res1
     WHEN (expr = comp_expr2) OR (expr IS NULL AND comp_expr2 IS NULL )
THEN res2
```

```
    ELSE default
  END
```

使用示例

```
gSQL> SELECT I1,
             DECODE( I1, 1, 'ONE',
                         2, 'TWO',
                         NULL, 'NULL VALUE',
                         'DEFAULT VALUE' ) AS DECODE_RESULT
      FROM T1;
  I1 DECODE_RESULT
---- -------------
   1 ONE
   2 TWO
null NULL VALUE
   3 DEFAULT VALUE
4 rows selected.
```

7.58 DEGREES

语句

```
DEGREES( radians )
```

说明

返回将弧度为单位的角度radians转换为以度为单位的值

若radians为NULL则返回NULL

使用示例

```
gSQL> SELECT DEGREES( PI() ) AS RESULT FROM DUAL;
RESULT
------
   180
1 row selected.
```

7.59 DENSE_RANK() OVER

语句

```
DENSE_RANK( ) OVER < window name or specification >
```

关于< window name or specification >的详细内容参考**window clause**

说明

Window function DENSE_RANK是计算排名的函数

排名为从1开始的连续整数值相同的row的排名也相同

但是与RANK不同的是即使出现值相同的row也不会跳过排名

无法使用window frame

使用示例

```
gSQL> SELECT department_id, salary,
            DENSE_RANK() OVER ( ORDER BY salary ) AS d_rank
      FROM employees
      WHERE department_id = 60;
```

```
DEPARTMENT_ID SALARY D_RANK
------------- ------ ------
           60   4200      1
           60   4800      2
           60   4800      2
           60   6000      3
           60   9000      4

5 rows selected.
```

7.60 DIGEST

语句

```
DIGEST( data, type )
```

说明

将用指定类型hash的数据的结果值转换为VARBINARY类型

根据下列规则输入数据类型时可产生隐式转换（implicit conversion）

- BINARY, VARBINARY类型的数据以VARBINARY输入
- LONG VARBINARY类型的数据以LONG VARBINARY输入
- LONG VARCHAR类型的数据以LONG VARCHAR输入
- 其余所有类型的数据均以VARCHAR类型进行隐式转换（implicit conversion）后输入

DIGEST函数支持的hash type如下

- 'SHA1'的结果为20byte varbinary
- 'SHA224'的结果为28byte varbinary
- 'SHA256'的结果为32byte varbinary
- 'SHA384'的结果为48byte varbinary
- 'SHA512'的结果为64byte varbinary

返回VARBINARY类型的结果因此要以hexadecimal字符查看结果时需要用HEX函数此时其长度为原本的2倍

使用示例

```
gSQL> SELECT HEX( DIGEST( 'my password', 'SHA256' ) ) AS RESULT FROM DUAL;


RESULT

----------------------------------------------------------------

BB14292D91C6D0920A5536BB41F3A50F66351B7B9D94C804DFCE8A96CA1051F2


1 row selected.
```

7.61 DUMP

语句

```
DUMP( expr )
```

说明

DUMP函数返回expr内部的表示信息

内部表示信息显示为数据类型长度 (byte length)以及数据信息

expr可以是所有类型

若expr为NULL则返回NULL

返回类型为CHARACTER VARYING

使用示例

```
gSQL> SELECT DUMP( 'DUMP' ) AS RESULT FROM DUAL;

RESULT

---------------------------------

Type=CHAR Len=4 : Str=68,85,77,80

1 row selected.
```

7.62 EXP

语句

```
EXP( num )
```

说明

EXP函数返回e（自然对数为底数）的num二次方值

若num为NULL则返回NULL

使用示例

```
gSQL> SELECT EXP( 1 ) AS RESULT FROM DUAL;
          RESULT
----------------
2.71828182845905
1 row selected.
```

7.63 EXTRACT

语句

```
EXTRACT( <field> FROM datetime )


<field> ::=
        YEAR
      | MONTH
      | DAY
      | HOUR
      | MINUTE
      | SECOND
      | TIMEZONE_HOUR
      | TIMEZONE_MINUTE
```

说明

EXTRACT从输入的datetime类型中找到指定的field并返回

datetime为DATE, TIMESTAMP, TIMESTAMP WITH TIME ZONE, TIME, TIME WITH TIME ZONE, INTERVAL类型

field不在datetime范围内时报错

另外是DATE类型时field只能是YEAR, MONTH, DAY其余报错

返回类型为NUMBER

EXTRACT函数的结果与**DATE_PART**相同

使用示例

```
gSQL> SELECT
      EXTRACT( SECOND FROM TO_TIMESTAMP( '2012-12-13 01:23:44.5',
                                         'YYYY-MM-DD HH24:MI:SS.FF1' )  )
              AS RESULT
      FROM DUAL;
RESULT
------
  44.5
1 row selected.


gSQL> SELECT
      EXTRACT( YEAR FROM CAST('2-3' AS INTERVAL YEAR TO MONTH) ) AS RESULT
      FROM DUAL;
RESULT
------
     2
1 row selected.
```

7.64 FACTORIAL

语句

```
FACTORIAL( num )
```

说明

FACTORIAL函数返回从1到num的连续自然数的阶乘

若num为NULL则返回NULL

使用示例

```
gSQL> SELECT FACTORIAL( 5 ) AS RESULT FROM DUAL;
RESULT
------
   120
1 row selected.
```

7.65 FIRST() OVER

语句

```
aggregation_function KEEP ( DENSE_RANK FIRST ORDER BY <sort specification
list> ) OVER < window name or specification >
```

关于< window name or specification >的详细内容参考**window clause**

说明

Window function FIRST函数排列KEEP子句内写在order by中的sort specification list后返回DENSE_RANK排名为1的row的aggregation function值

aggregation_function函数为AVG, COUNT, COUNT(*), SUM, MAX, MIN, STDDEV, VARIANCE

无法在window子句内使用order by

无法使用window frame

使用示例

```
gSQL> SELECT department_id, salary,
           DENSE_RANK() OVER ( ORDER BY department_id ) AS "DENSE_RANK",
```

```
          MAX( salary ) KEEP ( DENSE_RANK FIRST ORDER BY department_id )
OVER () AS "MAX_FIRST"
  FROM employees
 WHERE department_id BETWEEN 90 AND 100;


DEPARTMENT_ID SALARY DENSE_RANK MAX_FIRST
------------- ------ ---------- ---------
           90  24000          1     24000
           90  17000          1     24000
           90  17000          1     24000
          100  12000          2     24000
          100   9000          2     24000
          100   8200          2     24000
          100   7700          2     24000
          100   7800          2     24000
          100   6900          2     24000

9 rows selected.
```

7.66 FIRST_VALUE() OVER

语句

```
FIRST_VALUE ( expr ) [ RESPECT NULLS | IGNORE NULLS ] OVER < window name
or specification >


FIRST_VALUE ( expr [ RESPECT NULLS | IGNORE NULLS ] ) OVER < window name
or specification >
```

关于< window name or specification >的详细内容参考 **window clause**

说明

Window function FIRST_VALUE返回expr的首个值

RESPECT NULLS返回包含NULL值在内的row中的首个值

IGNORE NULLS返回非NULL的row中的首个值

未明确指定时默认值为RESPECT NULLS

无法在window句中使用order by

无法使用window frame

使用示例

```
gSQL> SELECT min_price AS "MIN_PRICE"
            , FIRST_VALUE( min_price ) OVER ( ORDER BY min_price NULLS
FIRST ) AS "FIRST_VALUE"
        FROM product_information
       WHERE supplier_id = 102050;


MIN_PRICE FIRST_VALUE
--------- -----------
     null        null
     null        null
       73        null
      247        null
      731        null

5 rows selected.
```

以下为在null_treatment中指定IGNORE NULLS的示例

```
gSQL> SELECT min_price AS "MIN_PRICE"
            , FIRST_VALUE( min_price IGNORE NULLS ) OVER ( ORDER BY
min_price NULLS FIRST )
                                                    AS "FIRST_VALUE"
        FROM product_information
```

```
       WHERE supplier_id = 102050;


MIN_PRICE FIRST_VALUE
--------- -----------
     null        null
     null        null
       73          73
      247          73
      731          73

5 rows selected.
```

7.67 FLOOR

语句

```
FLOOR( num )
```

说明

FLOOR函数返回小于num的最大整数

若num为NULL则返回NULL

使用示例

```
gSQL> SELECT FLOOR(42.8) AS RESULT1, FLOOR(-42.8) AS RESULT2 FROM DUAL;
RESULT1 RESULT2
------- -------
     42     -43
1 row selected.
```

7.68 FROM_BASE64

语句

```
FROM_BASE64( str )
```

说明

FROM_BASE64返回输入通过base64编码转换的字符后decoding的binary string

输入参数可以是CHARACTER VARYING, CHARACTER LONG VARYING的CHARACTER字符类型结果类型为BINARY VARYING或BINARY LONG VARYING的BINARY字符类型

str为NULL时结果值也为NULL

str中包含不属于base64范围的字符时报错

Decoding时忽略str的newlinecarriage returntabspace

详细内容参考参考**TO_BASE64**

使用示例

```
gSQL> SELECT FROM_BASE64( TO_BASE64( 'abc' ) ),
             FROM_BASE64( TO_BASE64( 'abcd' ) )
        FROM DUAL;
```

```
FROM_BASE64( TO_BASE64( 'abc' ) ) FROM_BASE64( TO_BASE64( 'abcd' ) )

--------------------------------- ---------------------------------

616263                            61626364

1 row selected.
```

7.69 FROM_TZ

语句

```
FROM_TZ( timestamp, timezone )
```

说明

FROM_TZ函数将timestamp和指定的format的timezone转换为TIMESTAMP WITH TIME ZONE类型之后返回

参数timestamp应为TIMESTAMP类型或可转换为TIMESTAMP类型

参数timestamp为NULL时结果为NULL

参数timezone应为CHARACTERCHARACTER VARYING等CHARACTER字符类型format为'TZH:TZM'

参数timezone为NULL时结果为NULL

结果类型为TIMESTAMP(6) WITH TIME ZONE

使用示例

```
gSQL> SELECT
      FROM_TZ( TIMESTAMP'2021-01-01 10:10:20.000000', '+06:00' ) AS RESULT
```

```
        FROM DUAL;

RESULT

-----------------------------------

2021-01-01 10:10:20.000000 +06:00

1 row selected.
```

7.70 GREATEST

语句

```
GREATEST( expr1 [, expr2, ... exprn ] )
```

说明

GREATEST函数返回作为输入的expr中的最大值

作为输入的expr中只要有一个为NULL则结果值为NULL

结果类型是expr1（第一个expr）的数据类型

expr1（第一个expr）的数据类型为数字类型和字符类型时均确定为可包含expr1...exprN范围的类型

expr1...exprN中均为CHAR类型时所有expr对比为VARCHAR类型结果类型确定为VARCHAR类型

使用示例

```
gSQL> SELECT GREATEST( 100, 0, 200, 150, 1 ) AS RESULT FROM DUAL;

RESULT

------

   200

1 row selected.
```

7.71 HEX

语句

```
HEX( str )
```

说明

以16进制字符返回str

str可以是CHARACTERCHARACTER VARYING,CHARACTER LONG VARYING的CHARACTER字符类型或可转换为字符类型的类型与BINARYBINARY VARYINGBINARY LONG VARYING的BINARY字符类型

结果类型为CHARACTER VARYING或CHARACTER LONG VARYING的CHARACTER字符类型

str为NULL时结果值也为NULL

HEX函数的参数为数字类型时报错

将10进制数字转换为16进制时可用使用'X' number format的TO_CHAR()函数

例: TO_CHAR（255'XX'）

详细内容参考：**UNHEX**

使用示例

```
gSQL> SELECT HEX( 'abc' ) FROM DUAL;

HEX( 'abc' )

------------

616263

1 row selected.
```

7.72 INITCAP

语句

```
INITCAP( str )
```

说明

INITCAP函数将字符串str中每个单词的第一个字母转换为大写将其余字母均转换为小写后进行返回

str为CHARACTER, CHARACTER VARYING, CHARACTER LONG VARYING等字符类型

字符串中的每个单词都由空格字母或非数字字符进行区分

str为NULL时结果值也为NULL

返回类型与作为输入的str的类型相同

使用示例

```
gSQL> SELECT INITCAP( 'hi GLIESE' ) AS RESULT FROM DUAL;
RESULT
---------
```

```
Hi Gliese

1 row selected.
```

7.73 INSTR

语句

```
INSTR( str, substr [, position [, occurrence ] ] )
```

说明

INSTR函数从str的position开始查找第occurrence个substr后返回其位置

参数strsubstr可以是CHARACTERCHARACTER VARYINGCHARACTER LONG VARYING等CHARACTER字符类型和BINARYBINARY VARYINGBINARY LONG VARYING等BINARY字符类型

参数positionoccurrence为数字类型

省略position与occurrence时默认值为1

position与occurrence从1开始按照character set的字符单位计算（不是byte单位）

position是str中开始搜索substr的第一个位置应为非0的整数值

- 正数时：从str的前面开始向右对比查找position的位置直到找到substr为止
- 负数时：从str的后面开始向左对比查找position的位置直到找到substr为止
- 0时：结果值为0

occurrence指str中substr重复的次数需为正整数

即使输入参数中一个为NULL结果值也为NULL

使用示例

```
gSQL> SELECT INSTR( 'ABCD ABCD ABCDABCD', 'BC' ) AS RESULT1,
             INSTR( 'ABCD ABCD ABCDABCD', 'BC', 4 ) AS RESULT2
      FROM DUAL;
RESULT1 RESULT2
------- -------
      2       7
1 row selected.


gSQL> SELECT INSTR( 'ABCD ABCD ABCDABCD', 'BC', 5 , 3 ) AS RESULT1,
             INSTR( 'ABCD ABCD ABCDABCD', 'BC', -5,  3 ) AS RESULT2
      FROM DUAL;
RESULT1 RESULT2
------- -------
     16       2
1 row selected.
```

7.74 LAG() OVER

语句

```
LAG ( expr [, offset [, default ] ] ) [ RESPECT NULLS | IGNORE NULLS ]
OVER < window name or specification >


LAG ( expr [ RESPECT NULLS | IGNORE NULLS ] [, offset [, default ] ] )
OVER < window name or specification >
```

关于< window name or specification >的详细内容参考**window clause**

说明

Window function LAG返回当前row中offset前的row的值

当Offset超出window范围时返回default值

如果未指定Offset, default值则设置为默认值

Offset默认值为1default默认值为NULL

RESPECT NULLS返回包含NULL值在内的offset前的row的值

IGNORE NULLS返回非NULL的row中offset前的row的值

未指定时默认值为RESPECT NULLS

无法使用window frame

使用示例

```
gSQL> SELECT department_id, employee_id, manager_id,
             LAG( manager_id ) OVER ( ORDER BY employee_id ) AS lag
        FROM employees
       WHERE department_id = 90;


DEPARTMENT_ID EMPLOYEE_ID MANAGER_ID  LAG
------------- ----------- ---------- ----
           90         100       null null
           90         101        100 null
           90         102        100  100


3 rows selected.
```

以下为指定offset的示例

```
gSQL> SELECT department_id, employee_id, manager_id,
             LAG( manager_id, 2 ) OVER ( ORDER BY employee_id ) AS lag
        FROM employees
       WHERE department_id = 90;


DEPARTMENT_ID EMPLOYEE_ID MANAGER_ID  LAG
------------- ----------- ---------- ----
           90         100       null null
```

```
           90         101        100 null

           90         102        100 null


3 rows selected.
```

以下为指定offset和default的示例

```
gSQL> SELECT department_id, employee_id, manager_id,
             LAG( manager_id, 2, 0 ) OVER ( ORDER BY employee_id ) AS lag
        FROM employees
       WHERE department_id = 90;


DEPARTMENT_ID EMPLOYEE_ID MANAGER_ID  LAG
------------- ----------- ---------- ----
           90         100       null    0
           90         101        100    0
           90         102        100 null


3 rows selected.
```

7.75 LAST() OVER

语句

```
aggregation_function KEEP ( DENSE_RANK LAST ORDER BY <sort specification
list> ) OVER < window name or specification >
```

关于< window name or specification >的详细内容参考**window clause**

说明

Window function LAST函数给KEEP子句内写于order by的sort specification list排序后返回DENSE_RANK排名最后的row的aggregation function值

aggregation_function函数有AVG, COUNT, COUNT(*), SUM, MAX, MIN, STDDEV, VARIANCE

window子句内无法使用order by

无法使用window frame

使用示例

```
gSQL> SELECT department_id, salary,
           DENSE_RANK() OVER ( ORDER BY department_id ) AS "DENSE_RANK",
```

```
          MAX( salary ) KEEP ( DENSE_RANK LAST ORDER BY department_id )
OVER () AS "MAX_LAST"
  FROM employees
 WHERE department_id BETWEEN 90 AND 100;


DEPARTMENT_ID SALARY DENSE_RANK MAX_LAST
------------- ------ ---------- --------
           90  24000          1    12000
           90  17000          1    12000
           90  17000          1    12000
          100  12000          2    12000
          100   9000          2    12000
          100   8200          2    12000
          100   7700          2    12000
          100   7800          2    12000
          100   6900          2    12000

9 rows selected.
```

7.76 LAST_DAY

语句

```
LAST_DAY( date )
```

说明

LAST_DAY函数返回date包含的月份的最后一天

参数date为DATE, TIMESTAMP, TIMESTAMP WITH TIME ZONE类型
返回类型与参数date的类型无关始终为DATE

若date为NULL则返回NULL

使用示例

```
gSQL> SELECT
      LAST_DAY( TO_DATE( '2012-07-10', 'YYYY-MM-DD' ) ) AS RESULT FROM
DUAL;
RESULT
----------
2012-07-31
1 row selected.
```

7.77 LAST_IDENTITY_VALUE

语句

```
LAST_IDENTITY_VALUE()
```

说明

在当前会话中为了identity dolumn而自动生成的最近值结果类型为NATIVE_BIGINT

无自动生成的值时返回null

与MS-SQL的@@IDENTITYMySQL的LAST_INSERT_ID()的功能类似对多个表执行DML时如下由最后变更的表决定值因此需注意使用

```
gSQL> INSERT INTO t1(name) VALUES ( 'leekmo' );


1 row created.


gSQL> SELECT LAST_IDENTITY_VALUE() FROM dual;


LAST_IDENTITY_VALUE()
---------------------
                   12
```

```
1 row selected.



gSQL> INSERT INTO t2(name) VALUES ( 'leekmo' );

1 row created.




gSQL> SELECT LAST_IDENTITY_VALUE() FROM dual;

LAST_IDENTITY_VALUE()
---------------------
                    2

1 row selected.
```

需获取INSERT时生成的identity column的值时如下使用**INSERT INTO name RETURNING .. INTO**语句

```
gSQL> CREATE TABLE t1 ( id INTEGER GENERATED BY DEFAULT AS IDENTITY, name
VARCHAR(32) );

Table created.
```

```
gSQL> \var v1 integer

gSQL> INSERT INTO t1(name) VALUES ( 'leekmo' ) RETURN id INTO :v1;


V1

--

 1


1 row created.
```

使用示例

以下为LAST_IDENTITY_VALUE()函数的使用示例

```
gSQL> CREATE TABLE t1 ( id   INTEGER GENERATED BY DEFAULT AS IDENTITY,

                        name VARCHAR(32) );


Table created.


gSQL> COMMIT;


Commit complete.
```

- 没有当前会话中生成的identity value

```
gSQL> SELECT LAST_IDENTITY_VALUE() FROM dual;
```

```
LAST_IDENTITY_VALUE()
---------------------
                  null

1 row selected.
```

- 自动生成identity value (1)

```
gSQL> INSERT INTO t1(name) VALUES ( 'leekmo' );

1 row created.
```

- Result: 1

```
gSQL> SELECT LAST_IDENTITY_VALUE() FROM dual;

LAST_IDENTITY_VALUE()
---------------------
                    1

1 row selected.
```

- 作为默认值自动生成 identity value (2)

```
gSQL> UPDATE t1 SET id = DEFAULT;
```

```
1 row updated.
```

- Result: 2

```
gSQL> SELECT LAST_IDENTITY_VALUE() FROM dual;


LAST_IDENTITY_VALUE()

---------------------

                    2


1 row selected.
```

- 未按照用户输入的值自动生成identity value

```
INSERT INTO t1 VALUES ( 100, 'jhkim' );


1 row updated.
```

- Result: 2

```
SELECT LAST_IDENTITY_VALUE() FROM dual;


LAST_IDENTITY_VALUE()

---------------------

                    2


1 row selected.
```

7.78 LAST_VALUE() OVER

语句

```
LAST_VALUE ( expr ) [ RESPECT NULLS | IGNORE NULLS ] OVER < window name or
specification >


LAST_VALUE ( expr [ RESPECT NULLS | IGNORE NULLS ] ) OVER < window name or
specification >
```

关于< window name or specification >的详细内容参考**window clause**

说明

Window function LAST_VALUE返回expr的最后一个值

RESPECT NULLS返回包含NULL值的row的最后一个值

IGNORE NULLS返回非NULL的row的最后一个值

未指定时默认值为RESPECT NULLS

window句内无法使用order by

无法使用window frame

使用示例

```
gSQL> SELECT min_price AS "MIN_PRICE"
            , LAST_VALUE( min_price ) OVER ( ORDER BY min_price ) AS
"LAST_VALUE"
        FROM product_information
       WHERE supplier_id = 102050;


MIN_PRICE LAST_VALUE
--------- ----------
       73         73
      247        247
      731        731
     null       null
     null       null

5 rows selected.
```

以下为在null_treatment中指定IGNORE NULLS的示例

```
gSQL> SELECT min_price AS "MIN_PRICE"
            , LAST_VALUE( min_price IGNORE NULLS ) OVER ( ORDER BY
min_price ) AS "LAST_VALUE"
        FROM product_information
       WHERE supplier_id = 102050;
```

```
MIN_PRICE LAST_VALUE
--------- ----------
       73         73
      247        247
      731        731
     null        731
     null        731

5 rows selected.
```

7.79 LEAD() OVER

语句

```
LEAD ( expr [, offset [, default ] ] ) [ RESPECT NULLS | IGNORE NULLS ]
OVER < window name or specification >


LEAD ( expr [ RESPECT NULLS | IGNORE NULLS ] [, offset [, default ] ] )
OVER < window name or specification >
```

关于< window name or specification >的详细内容参考**window clause**

说明

Window function LEAD返回当前row中到offset之后的row的值

当offset超出window范围时返回default值

未指定offset, default值时设置为默认值

offset的默认值为1default的默认值为NULL

RESPECT NULLS返回包含NULL值的row中offset后的row的值

IGNORE NULLS返回非NULL的row中offset后的row值

未指定时默认值为RESPECT NULLS

无法使用window frame

使用示例

```
gSQL> SELECT department_id, employee_id, manager_id,
             LEAD( manager_id ) OVER ( ORDER BY employee_id DESC ) AS lead
        FROM employees
       WHERE department_id = 90;


DEPARTMENT_ID EMPLOYEE_ID MANAGER_ID LEAD
------------- ----------- ---------- ----
           90         102        100  100
           90         101        100 null
           90         100       null null


3 rows selected.
```

以下是指定offset时的示例

```
gSQL> SELECT department_id, employee_id, manager_id,
             LEAD( manager_id, 2 ) OVER ( ORDER BY employee_id DESC ) AS
lead
        FROM employees
       WHERE department_id = 90;


DEPARTMENT_ID EMPLOYEE_ID MANAGER_ID LEAD
------------- ----------- ---------- ----
```

```
           90         102        100 null

           90         101        100 null

           90         100       null null


3 rows selected.
```

以下是指定offset和default时的示例

```
gSQL> SELECT department_id, employee_id, manager_id,

             LEAD( manager_id, 2, 0 ) OVER ( ORDER BY employee_id DESC )

AS lead

        FROM employees

       WHERE department_id = 90;


DEPARTMENT_ID EMPLOYEE_ID MANAGER_ID LEAD

------------- ----------- ---------- ----

           90         102        100 null

           90         101        100    0

           90         100       null    0


3 rows selected.
```

7.80 LEAST

语句

```
LEAST( expr1 [, expr2, ... exprn ] )
```

说明

LEAST函数返回作为输入的expr中的最小值

作为输入的expr中只要有一个为NULL则结果值为NULL

结果类型取决于expr1（第一个expr）的数据类型

expr1（第一个expr）的数据类型为数字类型或字符类型时均确定为可包含expr1...exprN范围的类型

expr1...exprN中均为CHAR类型时所有expr对比为VARCHAR类型结果类型为VARCHAR类型

使用示例

```
gSQL> SELECT LEAST( 100, 0, 200, 150, 1 ) AS RESULT FROM DUAL;
RESULT
------
     0
```

```
1 row selected.
```

7.81 LENGTH

语句

```
LENGTH( str )
```

说明

是**CHAR_LENGTH**的alias

使用示例

Multi byte character set: (例:UTF8)

```
gSQL> SELECT LENGTH( 'αβ-SUMMER' ) AS RESULT FROM DUAL;
     RESULT
     ---------------
              9
    1 row selected.
```

7.82 LENGTHB

语句

```
LENGTHB( str )
```

说明

是OCTET_LENGTH的aliase

参考：**OCTET_LENGTH**, **BYTE_LENGTH**

使用示例

- Multi byte character set (例: UTF8): 1 byte character

```
gSQL> SELECT LENGTHB( 'OCTET_LENGTH' ) AS RESULT_1BYTE_CHARACTERS
        FROM DUAL;
RESULT_1BYTE_CHARACTERS
-----------------------
                     12
1 row selected.
```

- Multi byte character set (例: UTF8): 2 byte character

```
gSQL> SELECT LENGTHB( 'αβ' ) AS RESULT_2BYTE_CHARACTERS FROM DUAL;

RESULT_2BYTE_CHARACTERS

-----------------------

                      4

1 row selected.
```

7.83 LISTAGG() OVER

语句

```
LISTAGG( str [, delimiter] ) WITHIN GROUP ( ORDER BY <sort specification
list> ) OVER < window name or specification >
```

关于< window name or specification >的详细内容参考**window clause**

说明

Window function LISTAGG是按照每个组中排序顺序连接str的函数

LISTAGG的OVER()子句中仅可描述PARTITION BY子句

使用OVER()子句将查询结果集划分为组

使用WITHIN GROUP (ORDER BY <sort specification list>)给组内记录排序

按照各组内排序的记录顺序连接str

当str为NULL时排除

delimiter是str连接分隔符省略时默认值为NULL

str可以是character string或者binary string

如果str是character string则结果类型为varchar

如果str是binary string则结果类型为varbinary

使用示例

```
gSQL>
SELECT regionkey,
       name,
       LISTAGG( name ) WITHIN GROUP ( ORDER BY nationkey )
                      OVER ( PARTITION BY regionkey )
       AS "LISTAGG( name ) RESULT",
       LISTAGG( name, ', ' ) WITHIN GROUP ( ORDER BY nationkey )
                             OVER ( PARTITION BY regionkey )
       AS "LISTAGG( name, ', ' ) RESULT"
  FROM nation;


REGIONKEY NAME    LISTAGG( name ) RESULT    LISTAGG( name, ', ' ) RESULT
--------- ------- ------------------------ ----------------------------
        1 BRAZIL  BRAZILCANADAPERU          BRAZIL, CANADA, PERU
        1 CANADA  BRAZILCANADAPERU          BRAZIL, CANADA, PERU
        1 PERU    BRAZILCANADAPERU          BRAZIL, CANADA, PERU
        1 null    BRAZILCANADAPERU          BRAZIL, CANADA, PERU
        2 null    INDIAJAPANCHINAVIETNAM    INDIA, JAPAN, CHINA, VIETNAM
        2 INDIA   INDIAJAPANCHINAVIETNAM    INDIA, JAPAN, CHINA, VIETNAM
        2 null    INDIAJAPANCHINAVIETNAM    INDIA, JAPAN, CHINA, VIETNAM
```

```
        2 null    INDIAJAPANCHINAVIETNAM    INDIA, JAPAN, CHINA, VIETNAM
        2 JAPAN   INDIAJAPANCHINAVIETNAM    INDIA, JAPAN, CHINA, VIETNAM
        2 CHINA   INDIAJAPANCHINAVIETNAM    INDIA, JAPAN, CHINA, VIETNAM
        2 null    INDIAJAPANCHINAVIETNAM    INDIA, JAPAN, CHINA, VIETNAM
        2 VIETNAM INDIAJAPANCHINAVIETNAM    INDIA, JAPAN, CHINA, VIETNAM
        3 EGYPT   EGYPTIRANIRAQ             EGYPT, IRAN, IRAQ
        3 IRAN    EGYPTIRANIRAQ             EGYPT, IRAN, IRAQ
        3 IRAQ    EGYPTIRANIRAQ             EGYPT, IRAN, IRAQ


15 rows selected.
```

7.84 LN

语句

```
LN( num )
```

说明

LN函数返回num的自然对数值

num应为大于0的值

若num为NULL则返回NULL

使用示例

```
gSQL> SELECT LN( 2.71828182845905 ) AS RESULT FROM DUAL;

RESULT

------

     1

1 row selected.
```

7.85 LNNVL

语句

```
LNNVL( expr )
```

说明

Logical Not Null VaLue (LNNVL) 函数与NOT logical operator 类似但如下输入值为null时将返回TRUE

使用示例

```
gSQL> SELECT c1, c2, (c1 = c2), NOT(c1 = c2), LNNVL(c1 = c2) FROM t1;


C1   C2 (C1 = C2) NOT(C1 = C2) LNNVL(C1 = C2)
-- ---- --------- ------------ --------------
 1    1 TRUE      FALSE        FALSE
 1    2 FALSE     TRUE         TRUE
 1 null null      null         TRUE


3 rows selected.
```

7.86 LOCALTIME

语句

```
LOCALTIME [()]

STATEMENT_LOCALTIME()
```

说明

以会话时间为准获取当前TIME WITHOUT TIME ZONE type值

LOCALTIME为SQL标准函数

获取当前时间的函数之间有如下差异

- TRANSACTION_LOCALTIME()：事务内的所有时间值均相同
- LOCALTIME, STATEMENT_LOCALTIME()：一个SQL语句内的所有时间值均相同
- CLOCK_LOCALTIME()：调用函数时每次均返回当前时间值

使用示例

每条row的时间值均相同

```
gSQL>  SELECT LOCALTIME FROM t1;


LOCALTIME

---------------

16:17:08.592459

16:17:08.592459

16:17:08.592459


3 rows selected.
```

7.87 LOCALTIMESTAMP

语句

```
LOCALTIMESTAMP [()]

STATEMENT_LOCALTIMESTAMP()
```

说明

以会话时间为准获取当前TIMESTAMP WITHOUT TIME ZONE type值

LOCALTIMESTAMP为SQL标准函数

获取当前TIMESTAMP的函数之间有如下差异

- TRANSACTION_LOCALTIMESTAMP()：事务内的所有TIMESTAMP值均相同
- OCALTIMESTAMP, STATEMENT_LOCALTIMESTAMP()：一个SQL语句内的所有TIMESTAMP值均相同
- CLOCK_LOCALTIMESTAMP()：调用函数时每次均获取当前TIMESTAMP值

使用示例

每条row的值均相同

```
gSQL> SELECT LOCALTIMESTAMP FROM t1;


LOCALTIMESTAMP

--------------------------

2013-12-12 16:21:51.790614

2013-12-12 16:21:51.790614

2013-12-12 16:21:51.790614


3 rows selected.
```

7.88 LOCAL_GROUP_ID

语句

```
LOCAL_GROUP_ID()
```

说明

返回执行用户查询的服务器的Cluster Group ID的函数

> Note:
>
> 仅为集群系统中有效的信息

使用示例

每条row的值均相同

```
gSQL> SELECT LOCAL_GROUP_ID() FROM DUAL;


LOCAL_GROUP_ID()
----------------
               1
```

```
1 row selected.
```

7.89 LOCAL_GROUP_NAME

语句

```
LOCAL_GROUP_NAME()
```

说明

返回执行用户查询的服务器的Cluster Group Name的函数

> Note:
>
> 仅为集群系统中有效的信息

使用示例

每条row的值均相同

```
gSQL> SELECT LOCAL_GROUP_NAME() FROM DUAL;
```

```
LOCAL_GROUP_NAME()

------------------

G1


1 row selected.
```

7.90 LOCAL_MEMBER_ID

语句

```
LOCAL_MEMBER_ID()
```

说明

返回执行用户查询的服务器的Cluster Member ID的函数

> Note:
>
> 仅为集群系统中有效的信息

使用示例

每条row的值均相同

```
gSQL> SELECT LOCAL_MEMBER_ID() FROM DUAL;

LOCAL_MEMBER_ID()
-----------------
                1

1 row selected.
```

7.91 LOCAL_MEMBER_NAME

语句

```
LOCAL_MEMBER_NAME()
```

说明

返回执行用户查询的服务器的cluster member name的函数

> Note:
>
> 仅为集群系统中有效的信息

使用示例

每条row的值均相同

```
gSQL> SELECT LOCAL_MEMBER_NAME() FROM DUAL;


LOCAL_MEMBER_NAME()

--------------------

G1N1
```

```
1 row selected.
```

7.92 LOG

语句

```
LOG( num2 )

LOG( num1, num2 )
```

说明

LOG函数返回底数为num1的num2的对数值

省略num1时返回以底数为10计算的值

num1应为非1与0的正数num2应为正数

若num1或num2为NULL则返回NULL

使用示例

```
gSQL> SELECT LOG( 100 ) AS RESULT1, LOG( 4, 16 ) AS RESULT2 FROM DUAL;

RESULT1 RESULT2

------- -------

      2       2

1 row selected.
```

7.93 LOGON_USER

语句

```
LOGON_USER()
```

说明

返回login的用户

用如下三种形式管理用户信息

- Logon user：执行login的用户维持到关闭连接
- Session user：与最初的logon user相同但可以用SET SESSION AUTHORIZATION语句进行变更
- Current user：一般情况下与session user相同但使用PSMView时为了控制访问等在系统内部暂时变更
    - Session user与current user的差异和unix系统的real user与effective user的差异类似

使用示例

```
% gsql test test
```

```
gSQL> SELECT LOGON_USER() AS result FROM DUAL;


RESULT

------

TEST


1 row selected.
```

7.94 LOWER

语句

```
LOWER( str )
```

说明

LOWER函数返回str的小写

参数str为CHARACTER, CHARACTER VARYING, CHARACTER LONG VARYING等字符类型

str为NULL时结果值也为NULL

返回类型与参数str类型相同

使用示例

```
gSQL> SELECT LOWER( 'SPRING' ) AS RESULT FROM DUAL;
RESULT
------
spring
1 row selected.
```

7.95 LPAD

语句

```
LPAD( str, length, [, fill] )
```

说明

LPAD函数返回在str左侧添加字符串fill的值直到string的长度达到length

str为CHARACTERCHARACTER VARYINGCHARACTER LONG VARYING的CHARACTER字符类型与BINARYBINARY VARYINGBINARY LONG VARYING 的BINARY字符类型

length为数字类型

length为字符的数量最大范围为结果类型的最大PRECISION

省略fill时添加空白字符

str长度大于length时减掉与length长度相同的数量并返回

strlengthfill中只要一个为NULL则结果值为NULLlength为0或负数时结果值也为NULL

结果类型如下表

str的类型	结果类型
CHAR或VARCHAR	VARCHAR

str的类型	结果类型
LONG VARCHAR	LONG VARCHAR
BINARY或VARBINARY	VARBINARY
LONG VARBINARY	LONG VARBINARY

Table 7-13　LPAD的结果类型

使用示例

```
gSQL> SELECT LPAD('AA', 5) AS RESULT1,
             LPAD('AA', 5, 'X' ) AS RESULT2,
             LPAD('AA', 1 ) AS RESULT3
      FROM DUAL;
RESULT1 RESULT2 RESULT3
------- ------- -------
   AA   XXXAA   A
1 row selected.
```

7.96 LTRIM

语句

```
LTRIM( trim_source [, trim_character ] )
```

说明

LTRIM函数返回在trim_source从左侧开始对比删掉trim_character直到没有一致的字符的结果

trim_character和trim_source为CHARACTERCHARACTER VARYINGCHARACTER LONG VARYING等CHARACTER字符类型与BINARYBINARY VARYINGBINARY LONG VARYING等BINARY字符类型

trim_charactertrim_source中只要一个为NULL则结果值为NULL

省略trim_character时默认指定为single blank space(' ')

结果类型如下表

trim_source, trim_character类型	结果类型
CHAR或VARCHAR	VARCHAR
LONG VARCHAR	LONG VARCHAR
BINARY或VARBINARY	VARBINARY
LONG VARBINARY	LONG VARBINARY

Table 7-14　LTRIM的结果类型

使用示例

```
gSQL> SELECT LTRIM( '_____LTRIM', '_' ) AS RESULT FROM DUAL;

RESULT

------

LTRIM

1 row selected.
```

7.97 MAX

语句

```
MAX( [ ALL | DISTINCT ] expr )
```

说明

作为aggregation函数获取row的expr中的最大值

指定ALL时对所有值执行聚合操作
指定DISTINCT时对排除重复的值执行聚合操作
不指定ALL或DISTINCT时与指定ALL的处理方式相同

MAX不受ALL与DISTINCT的影响返回相同的结果

使用示例

```
gSQL> SELECT MAX(c1) FROM t1;


MAX(C1)

-------

      3


1 row selected.
```

7.98 MAX() OVER

语句

```
MAX ( expr ) OVER < window name or specification >
```

关于< window name or specification >的详细内容参考**window clause**

说明

Window function MAX是获取expr中最大值的函数

不计算NULL值

使用示例

```
gSQL> SELECT product_id AS "PRODUCT_ID", min_price AS "MIN_PRICE"
             , MAX( min_price ) OVER ( ORDER BY product_id ) AS "MAX"
        FROM product_information
       WHERE supplier_id = 102050;


PRODUCT_ID MIN_PRICE  MAX
---------- --------- ----
      1769      null null
```

```
      1770         73   73

      2378        247  247

      2382        731  731

      3355       null  731


5 rows selected.
```

7.99 MEDIAN() OVER

语句

```
MEDIAN ( expr ) OVER < window name or specification >
```

关于< window name or specification >的详细内容参考**window clause**

说明

Window function MEDIAN是返回中间位置row值的函数

不计算NULL

window子句内无法使用order by

无法使用window frame

使用示例

```
gSQL> SELECT supplier_id, min_price
            , MEDIAN( min_price ) OVER ( PARTITION BY supplier_id ) AS
"MEDIAN"
        FROM product_information
       WHERE supplier_id = 102050;
```

```
SUPPLIER_ID MIN_PRICE MEDIAN

----------- --------- ------

     102050        73    247

     102050       247    247

     102050       731    247

     102050      null    247

     102050      null    247


5 rows selected.
```

7.100 MIN

语句

```
MIN( [ ALL | DISTINCT ] expr )
```

说明

作为aggregation函数获取row的expr中的最小值

指定ALL时对所有值执行聚合操作
指定DISTINCT时对排除重复的值执行聚合操作
不指定ALL或DISTINCT时与指定ALL的处理方式相同

MIN不受ALL与DISTINCT的影响返回相同的结果

使用示例

```
gSQL> SELECT MIN(c1) FROM t1;


MIN(C1)

-------

      1


1 row selected.
```

7.101 MIN() OVER

语句

```
MIN ( expr ) OVER < window name or specification >
```

关于< window name or specification >的详细内容参考**window clause**

说明

Window function MIN是获取expr中最小值的函数

不计算NULL值

使用示例

```
gSQL> SELECT product_id AS "PRODUCT_ID", min_price AS "MIN_PRICE"
             , MIN( min_price ) OVER ( ORDER BY product_id DESC ) AS "MIN"
        FROM product_information
       WHERE supplier_id = 102050;


PRODUCT_ID MIN_PRICE  MIN
---------- --------- ----
      3355      null null
```

```
      2382       731  731

      2378       247  247

      1770        73   73

      1769      null   73


5 rows selected.
```

7.102 MOD

语句

```
MOD( num1, num2 )
```

说明

MOD返回num1除以num2的余数

num1, num2为数字类型

num2为0时报错

若参数num1或num2为NULL则返回NULL

使用示例

```
gSQL> SELECT MOD(5, 4) AS RESULT1, MOD(-5, 4) AS RESULT2 FROM DUAL;
RESULT1 RESULT2
------- -------
      1      -1
1 row selected.
```

7.103 MONTHS_BETWEEN

语句

```
MONTHS_BETWEEN( date1, date2 )
```

说明

MONTHS_BETWEEN返回date2与date1之间的天数除以31的月份数

date1或date2为NULL时结果为NULL

date1date2为DATETIMESTAMPTIMESTAMP WITH TIME ZONE类型

结果类型为NUMBER

> Note:
>
> date1与date2均包含相同的日期（例：2014-01-15与2014-02-15）或月份的最后一天（例：2014-08-31与2014-09-30）时不管TIMESTAMP区间（存在时）是否一致均返回整数结果

使用示例

```
gSQL> SELECT
        MONTHS_BETWEEN('2018-01-18', '2018-01-17')
  FROM DUAL;

MONTHS_BETWEEN('2018-01-18', '2018-01-17')
------------------------------------------
                      3.225806451612903E-2
1 row selected.

gSQL> SELECT
        MONTHS_BETWEEN('2018-02-17', '2018-01-17')
  FROM DUAL;

MONTHS_BETWEEN('2018-02-17', '2018-01-17')
------------------------------------------
                                         1
1 row selected.

gSQL> SELECT
        MONTHS_BETWEEN('2018-02-28', '2018-01-31')
  FROM DUAL;

MONTHS_BETWEEN('2018-02-28', '2018-01-31')
```

```
----------------------------------------

                                       1

1 row selected.
```

7.104 NEXT_DAY

语句

```
NEXT_DAY( date, day )
```

说明

经过给定的date(日期)求出第一次到来的day(星期)的日期

第二个day为指代day的字符串或数字

- 字符串：SUNDAY ~ SATURDAY 或 SUN ~ SAT
- 数字：1（sunday） ~ 7（saturday）

如果输入参数中只要有一个为NULL则为NULL

与date的输入类型无关返回类型始终为DATE类型

结果值的时分秒返回与输入date相同的时分秒

使用示例

- 2020-08-11是星期二

```
gSQL> SELECT NEXT_DAY( TO_DATE( '2020-08-11', 'YYYY-MM-DD'),
                       'SUNDAY' ) AS RESULT1
```

```
      FROM DUAL;
RESULT1
----------
2020-08-16
1 row selected.

gSQL> SELECT NEXT_DAY( TO_DATE( '2020-08-11', 'YYYY-MM-DD' ),
                       'SUN' ) AS RESULT1
FROM DUAL;
RESULT1
----------
2020-08-16
1 row selected.

gSQL> SELECT NEXT_DAY( TO_DATE( '2020-08-11', 'YYYY-MM-DD' ),
                       1 ) AS RESULT1
FROM DUAL;
RESULT1
----------
2020-08-16
1 row selected.

gSQL> SELECT TO_CHAR( NEXT_DAY( TO_DATE( '2020-08-11', 'YYYY-MM-DD' ),
                                'SUNDAY' ),
                      'YYYY-MM-DD HH24:MI:SS' ) AS RESULT1
```

```
FROM DUAL;

RESULT1

-------------------

2020-08-16 00:00:00

1 row selected.
```

7.105 NEXTVAL

语句

```
seq_name.NEXTVAL

NEXTVAL( seq_name )

NEXT VALUE FOR seq_name
```

说明

获取序列对象的下一个值

使用示例

```
gSQL> CREATE SEQUENCE seq;


Sequence created.


gSQL> COMMIT;


Commit complete.


gSQL> SELECT seq.NEXTVAL FROM dual;
```

```
SEQ.NEXTVAL

-----------

          1


1 row selected.


gSQL> SELECT NEXTVAL( seq ) FROM dual;


NEXTVAL( SEQ )

--------------

             2


1 row selected.


gSQL> SELECT NEXT VALUE FOR seq FROM dual;


NEXT VALUE FOR SEQ

------------------

                 3


1 row selected.
```

7.106 NTH_VALUE() OVER

语句

```
NTH_VALUE ( expr, n ) [ FROM { FIRST | LAST } ][ { RESPECT | IGNORE }
NULLS ] OVER < window name or specification >
```

关于< window name or specification >的详细内容参考**window clause**

说明

Window function NTH_VALUE返回第n个row的expr值

window的row数量比n少时返回NULL

参数n可以是数字类型或可以转换成数字的类型

FROM FIRST从第一个row指向第n个row

FROM LAST从最后一个row指向第n个row

未指定时默认值为FROM FIRST

RESPECT NULLS返回包含NULL值在内的第n个row的值

IGNORE NULLS返回非NULL的row中第n个值

未指定时默认值为RESPECT NULLS

使用示例

```
gSQL> SELECT product_id, min_price,
             NTH_VALUE( min_price, 2 ) OVER ( ORDER BY product_id ) AS
nth_value
      FROM product_information
      WHERE supplier_id = 102050;


PRODUCT_ID MIN_PRICE NTH_VALUE
---------- --------- ---------
      1769      null      null
      1770        73        73
      2378       247        73
      2382       731        73
      3355      null        73


5 rows selected.
```

以下为指定FROM LAST时的示例

```
gSQL> SELECT product_id, min_price,
             NTH_VALUE( min_price, 2 ) FROM LAST OVER ( ORDER BY
product_id ) AS nth_value
      FROM product_information
      WHERE supplier_id = 102050;
```

```
PRODUCT_ID MIN_PRICE NTH_VALUE
---------- --------- ---------
      1769      null      null
      1770        73      null
      2378       247        73
      2382       731       247
      3355      null       731

5 rows selected.
```

以下为指定IGNORE NULLS时的示例

```
gSQL> SELECT product_id, min_price,
             NTH_VALUE( min_price, 2 ) IGNORE NULLS OVER ( ORDER BY
product_id ) AS nth_value
      FROM product_information
      WHERE supplier_id = 102050;

PRODUCT_ID MIN_PRICE NTH_VALUE
---------- --------- ---------
      1769      null      null
      1770        73      null
      2378       247       247
      2382       731       247
      3355      null       247
```

```
5 rows selected.
```

7.107 NTILE() OVER

语句

```
NTILE( expr ) OVER < window name or specification >
```

关于< window name or specification >的详细内容参考**window clause**

说明

Window function NTILE返回各row的存储桶编号

存储桶编号为从1开始连续的整数存储桶数量与expr数量相同

存储桶数量较row数量多时给各row分配一个存储桶后剩下的存储桶空置

expr须为正常数expr不是固定数字时expr须为window partition by的对象

无法使用window frame

使用示例

```
gSQL> SELECT department_id, salary,
             NTILE(3) OVER ( ORDER BY salary ) AS ntile
      FROM employees
```

```
      WHERE department_id = 60;


DEPARTMENT_ID SALARY NTILE

------------- ------ -----

           60   4200     1

           60   4800     1

           60   4800     2

           60   6000     2

           60   9000     3


5 rows selected.
```

以下为存储桶数量(expr数量)较row数量大的示例

```
gSQL> SELECT department_id, salary,

             NTILE(6) OVER ( ORDER BY salary ) AS ntile

      FROM employees

      WHERE department_id = 60;


DEPARTMENT_ID SALARY NTILE

------------- ------ -----

           60   4200     1

           60   4800     2

           60   4800     3

           60   6000     4

           60   9000     5
```

```
5 rows selected.
```

7.108 NULLIF

语句

```
NULLIF( expr1, expr2 )
```

说明

expr1与expr2相同时返回null不同时返回第一个expr1

expr1与expr2的类型不同时根据结果类型组合规则决定结果类型

NULLIF与CASE的处理方式相同

- NULLIF(expr1, expr2)

```
CASE WHEN expr1 = expr2 THEN NULL
       ELSE expr1
  END
```

使用示例

```
gSQL> SELECT NULLIF( 'SUN', 'SUN' ) AS RESULT1,
```

```
            NULLIF( 'SUN', 'MOON' ) AS RESULT2
       FROM DUAL;
RESULT1 RESULT2
------- -------
null    SUN
1 row selected.
```

7.109 NUMTODSINTERVAL

语句

```
NUMTODSINTERVAL( num, interval_indicator )
```

说明

将interval_indicator单位的number转换为interval day to second类型后返回

参数number为数字类型

参数interval_indicator应为CHARVARCHAR等字符类型不区分大小写的

'DAY''HOUR''MINUTE''SECOND'中的一个

参数中即使只有一个NULL结果也会返回NULL

返回interval day(6) to second(6)类型的结果用户无法任意变更precision转换的结果的leading precision超过默认precision时报错fraction precision超过默认precision时返回四舍五入的结果

使用示例

- 将1 Day转换为interval day to second类型

```
gSQL> SELECT NUMTODSINTERVAL(1, 'DAY') FROM DUAL;


NUMTODSINTERVAL(1, 'DAY')

-------------------------

+000001 00:00:00.000000


1 row selected.
```

- 将36 Hour转换为interval day to second类型

```
gSQL> SELECT NUMTODSINTERVAL(36, 'HOUR') FROM DUAL;


NUMTODSINTERVAL(36, 'HOUR')

---------------------------

+000001 12:00:00.000000


1 row selected.
```

- 将1530 Minute转化为interval day to second类型

```
gSQL> SELECT NUMTODSINTERVAL(1530, 'MINUTE') FROM DUAL;


NUMTODSINTERVAL(1530, 'MINUTE')

-------------------------------

+000001 01:30:00.000000
```

```
1 row selected.
```

- 将90100.1234567 Second转换为interval day to second类型

```
gSQL> SELECT NUMTODSINTERVAL(90100.1234567, 'SECOND') FROM DUAL;


NUMTODSINTERVAL(90100.1234567, 'SECOND')

---------------------------------------

+000001 01:01:40.123457


1 row selected.
```

7.110 NUMTOYMINTERVAL

语句

```
NUMTOYMINTERVAL( num, interval_indicator )
```

说明

将interval_indicator单位的number转换为interval year to month类型后返回

参数number为数字类型

参数interval_indicator应为CHARVARCHAR等字符类型不区分大小写的'YEAR''MONTH'中的一个

参数中即使只有一个NULL结果也会返回NULL

返回interval year(6) to month类型的结果用户无法任意变更precision转换的结果的leading precision超过默认precision时报错

使用示例

- 将1 Year转换为interval year to month

```
gSQL> SELECT NUMTOYMINTERVAL(1, 'YEAR') FROM DUAL;
```

```
NUMTOYMINTERVAL(1, 'YEAR')
--------------------------
+000001-00

1 row selected.
```

- 将13.5 Month转换为interval year to month

```
gSQL> SELECT NUMTOYMINTERVAL(13.5, 'MONTH') FROM DUAL;

NUMTOYMINTERVAL(13.5, 'MONTH')
------------------------------
+000001-02

1 row selected.
```

7.111 NVL

语句

```
NVL( expr1, expr2 )
```

说明

expr1不为null时返回expr1expr1为null时返回expr2

结果类型取决于expr1的数据类型

expr1中记述了NULL时结果类型取决于expr2的类型

expr1的数据类型为数字类型与字符类型时均确定为可包含expr1expr2的范围的类型

expr1expr2的数据类型均为CHAR时结果类型确定为VARCHAR

使用示例

```
gSQL> SELECT I1, NVL( I1, 0 ) FROM T1;
  I1 NVL( I1, 0 )
---- ------------
   1            1
null            0
2 rows selected.
```

7.112 NVL2

语句

```
NVL2( expr1, expr2, expr3 )
```

说明

expr1不为null时返回expr2expr1为null时返回expr3

结果类型取决于expr2的数据类型

expr2中记述了NULL时结果类型取决于expr3的类型

expr2的数据类型为数字类型与字符类型时均确定为可包含expr2expr3的范围的类型

expr2expr3的数据类型均为CHAR时结果类型确定为VARCHAR

使用示例

```
gSQL> SELECT I1, NVL2( I1, I1 * 1000, 0 ) FROM T1;
  I1 NVL2( I1, I1 * 1000, 0 )
---- ------------------------
   1                    1000
null                       0
2 rows selected.
```

7.113 OCTET_LENGTH

语句

```
OCTET_LENGTH( str )

BYTE_LENGTH( str )

LENGTHB( str )
```

说明

OCTECT_LENTEH返回str的字节数

str为CHARACTERCHARACTER VARYINGCHARACTER LONG VARYING等CHARACTER字符类型与BINARYBINARY VARYINGBINARY LONG VARYING等BINARY字符类型

str的类型为CHARACTER时包含空字符进行计算

str为NULL时结果值也为NULL

OCTET_LENGTH的aliase有**BYTE_LENGTH**和**LENGTHB**函数

使用示例

- Multi byte character set (例: UTF8): 1 byte character

```
gSQL> SELECT OCTET_LENGTH( 'OCTET_LENGTH' ) AS RESULT_1BYTE_CHARACTERS
        FROM DUAL;
RESULT_1BYTE_CHARACTERS
-----------------------
                     12
1 row selected.
```

- Multi byte character set (例: UTF8): 2 byte character

```
gSQL> SELECT OCTET_LENGTH( 'αβ' ) AS RESULT_2BYTE_CHARACTERS FROM DUAL;
RESULT_2BYTE_CHARACTERS
-----------------------
                      4
1 row selected.
```

7.114 OVERLAY

语句

```
OVERLAY( str1 PLACING str2 FROM start_position  [ FOR string_length ] )
```

说明

OVERLAY函数返回用str2替换从str1的start_position开始到string_length的字符的结果

str1与str2为CHARACTERCHARACTER VARYINGCHARACTER LONG VARYING等CHARCATER字符类型与BINARYBINARY VARYINGBINARY LONG VARYING等BINARY字符类型

start_position与string_length为数字类型

- OVERLAY的结果如下
  - 指定FOR时

    SUBSTRING(str1 FROM 1 FOR (start_position - 1))

    || str2

    || SUBSTRING(str1 FROM (start_position + string_length)
  - 省略FOR时

    SUBSTRING(str1 FROM 1 FOR (start_position - 1))

    || str2

    || SUBSTRING(str1 FROM (start_position + CHAR_LENGTH(str2))

详细内容参考：**SUBSTRING**

结果类型如下表

str1, str2类型	结果类型
CHAR或VARCHAR	VARCHAR
LONG VARCHAR	LONG VARCHAR
BINARY或VARBINARY	VARBINARY
LONG VARBINARY	LONG VARBINARY

使用示例

```
gSQL> SELECT
         OVERLAY( 'RESULT_OF_XXX_FUNC' PLACING 'OVERLAY' FROM 11 )
         AS RESULT1,
         OVERLAY( 'RESULT_OF_XXX_FUNC' PLACING 'OVERLAY' FROM 11 FOR 3 )
         AS RESULT2
      FROM DUAL;
RESULT1            RESULT2
------------------ ----------------------
RESULT_OF_OVERLAYC RESULT_OF_OVERLAY_FUNC
1 row selected.
```

7.115 PERCENT_RANK() OVER

语句

```
PERCENT_RANK( ) OVER < window name or specification >
```

关于< window name or specification >的详细内容参考**window clause**

说明

Window function PERCENT_RANK计算各个row占所有row数量的排名比例

PERCENT_RANK函数的结果为从0到1之间的数字row值相同则比例值也相同

无法使用window frame

使用示例

```
gSQL> SELECT department_id, salary,
             PERCENT_RANK() OVER ( ORDER BY salary ) AS p_rank
      FROM employees
      WHERE department_id = 60;


DEPARTMENT_ID SALARY P_RANK
```

```
------------- ------ ------

           60   4200      0

           60   4800    .25

           60   4800    .25

           60   6000    .75

           60   9000      1


5 rows selected.
```

7.116 PERCENTILE_CONT() OVER

语句

```
PERCENTILE_CONT( expr ) WITHIN GROUP ( ORDER BY <sort specification> )
OVER < window name or specification >
```

关于< window name or specification >的详细内容请参阅**window clause**

说明

Window function PERCENTILE_CONT是一个假设连续分布模型的逆分布函数(inverse distribution function)

计算组内已排序的not null value的指定百分位数的值

计算值可能与组内已排序的特定值不同

expr须为0到1之间的百分位数值

OVER()子句中仅可描述PARTITION BY子句

使用PARTITION BY子句将查询结果集划分为组

使用WITHIN GROUP (ORDER BY <sort specification>)对组内记录进行排序

仅可在ORDER BY中指定一个<sort specification>

在排序value中将NULL排除

计算公式如下

```
P : 百分位数

N : 组内已排序的not null value记录的数量

RN = ( 1 + ( P * (N-1) ) )

CRN = CEILING( RN )

FRM = FLOOR( RN )


* if ( CRN = FRN = RN )

     RN的sort expression value

* else

     ( CRN - RN ) * FRN的sort expression value + ( RN - FRN ) * CRN的sort

expression value
```

MEDIAN window函数为PERCENTILE_CONT window函数中的特定情况其默认值为百分位数0.5

更多内容请参阅以下链接

- **MEDIAN() OVER**
- **PERCENTILE_DISC() OVER**

使用示例

```
gSQL>
```

```
SELECT item_no,
       sales,
       PERCENTILE_CONT( 0.5 ) WITHIN GROUP ( ORDER BY sales )
                              OVER ( PARTITION BY item_no )
       AS PERCENTILE_CONT
  FROM store;


ITEM_NO SALES PERCENTILE_CONT
------- ----- ---------------
    100    50             125
    100    90             125
    100    90             125
    100   100             125
    100   120             125
    100   130             125
    100   150             125
    100   150             125
    100   170             125
    100   200             125
    235    50              90
    235    70              90
    235    90              90
    235   130              90
    235   190              90


15 rows selected.
```

7.117 PERCENTILE_DISC() OVER

语句

```
PERCENTILE_DISC( expr ) WITHIN GROUP ( ORDER BY <sort specification> )
OVER < window name or specification >
```

关于< window name or specification >的详细内容参考**window clause**

说明

Window function PERCENTILE_DISC是假设离散分布模型的逆分布函数(inverse distribution function)

计算组中排序的not null value的百分位数

决定已计算的百分位数中较参数百分位数大或和其相同的值中最小的值

返回决定的百分位数值

expr须为0到1之间的百分位数值

OVER()子句中仅可描述PARTITION BY子句

使用PARTITION BY子句将查询结果集划分为组

使用WITHIN GROUP (ORDER BY <sort specification>)给组中记录排序

仅可在ORDER BY中指定一个<sort specification>

使用排序的记录计算sort expression value的CUME_DIST

计算CUME_DIST时排除NULL

决定比指定为expr的百分位数大或者和其相同的CUME_DIST中最小的值

返回决定的CUME_DIST对应的值

结果类型和sort expression value类型相同

更多内容请参阅**CUME_DIST() OVER**

使用示例

```
gSQL>
SELECT item_no,
       sales,
       CUME_DIST() OVER ( PARTITION BY item_no ORDER BY sales )
       AS CUME_DIST,
       PERCENTILE_DISC( 0.5 ) WITHIN GROUP ( ORDER BY sales )
                              OVER ( PARTITION BY item_no )
       AS PERCENTILE_DISC
  FROM store;


ITEM_NO SALES CUME_DIST PERCENTILE_DISC
------- ----- --------- ---------------
    100    50        .1             120
    100    90        .3             120
```

```
    100    90        .3            120
    100   100        .4            120
    100   120        .5            120
    100   130        .6            120
    100   150        .8            120
    100   150        .8            120
    100   170        .9            120
    100   200         1            120
    235    50        .2             90
    235    70        .4             90
    235    90        .6             90
    235   130        .8             90
    235   190         1             90

15 rows selected.
```

7.118 PHYSICAL_LENGTH

语句

```
PHYSICAL_LENGTH( expr )
```

说明

PHYSICAL_LENGTH函数返回expr的内部表达信息byte数

参数expr可以是所有数据类型

输入参数为NULL时结果为0

使用示例

- 输入参数为NULL时

```
gSQL> SELECT PHYSICAL_LENGTH( NULL ) AS RESULT FROM DUAL;
RESULT
------
     0
1 row selected.
```

- 以下为显示112312345的NUMBER type byte数的示例

```
gSQL> SELECT PHYSICAL_LENGTH( 1 ) AS RESULT FROM DUAL;

RESULT

------

     2

1 row selected.


gSQL> SELECT PHYSICAL_LENGTH( 123 ) AS RESULT FROM DUAL;

RESULT

------

     3

1 row selected.


gSQL> SELECT PHYSICAL_LENGTH( 12345 ) AS RESULT FROM DUAL;

RESULT

------

     4

1 row selected.
```

7.119 PI

语句

```
PI()
```

说明

PI返回"π"常量（constant）

使用示例

```
gSQL> SELECT PI() AS RESULT FROM DUAL;
               RESULT
--------------------
3.141592653589793E+0
1 row selected.
```

7.120 POSITION

语句

```
POSITION( str1 IN str2 )
```

说明

POSITION函数在str2中查找第一个str1并返回其位置

str1与str2为CHARACTERCHARACTER VARYINGCHARACTER LONG VARYING等CHARCATER字符类型与BINARYBINARY VARYINGBINARY LONG VARYING等BINARY字符类型

无法在str2中找到str1时返回值为0
在str2中找到str1时从1开始返回其位置
返回的位置值以CHARACTER为单位进行计算（不是以byte为单位）
str1或str2为NULL时返回值也为NULL

使用示例

```
gSQL> SELECT POSITION( 'CHAR' IN 'LONG CHAR 2000' ) AS RESULT FROM DUAL;

RESULT

------

     6

1 row selected.
```

7.121 POWER

语句

```
POWER( num1, num2 )
```

说明

POWER函数返回num1的num2次方值

num1与num2为数字类型

num1为负数时num2应为正数

num1或num2的值为NULL时结果值也为NULL

使用示例

```
gSQL> SELECT POWER( 2, 3 ) AS RESULT FROM DUAL;
RESULT
------
     8
1 row selected.
```

7.122 RADIANS

语句

```
RADIANS( degrees )
```

说明

RADIANS函数返回degrees的弧度

参数degrees可以为数字类型

参数degrees为NULL时返回NULL

使用示例

```
gSQL> SELECT RADIANS( 180 ) AS RESULT FROM DUAL;

          RESULT

----------------

3.14159265358979

1 row selected.
```

7.123 RANDOM

语句

```
RANDOM( min, max )
```

说明

RANDOM返回min以上max以下的随机值

参数minmax可以为数字类型

参数min或max为NULL时返回NULL

使用示例

```
gSQL> SELECT RANDOM( 1, 100 ) AS RESULT FROM DUAL;

          RESULT

----------------

34.1870528003201

1 row selected.
```

7.124 RANK() OVER

语句

```
RANK( ) OVER < window name or specification >
```

关于< window name or specification >的详细内容参考**window clause**

说明

Window function RANK为计算排序的函数

排序为从1开始的整数值相同的row排序也相同

无法使用window frame

使用示例

```
gSQL> SELECT department_id, salary,
             RANK() OVER ( ORDER BY salary ) AS rank
      FROM employees
      WHERE department_id = 60;


DEPARTMENT_ID SALARY RANK
```

```
------------- ------ ----

           60   4200    1

           60   4800    2

           60   4800    2

           60   6000    4

           60   9000    5


5 rows selected.
```

7.125 RATIO_TO_REPORT() OVER

语句

```
RATIO_TO_REPORT ( expr ) OVER < window name or specification >
```

关于< window name or specification >的详细内容参考**window clause**

说明

Window function RATIO_TO_REPORT计算各row值在expr值总和中所占的比例

row值为NULL时返回结果值为NULL

无法在window子句内使用order by

无法使用window frame

使用示例

```
gSQL> SELECT supplier_id, min_price
             , RATIO_TO_REPORT( min_price ) OVER ( PARTITION BY
supplier_id ) AS "RATIO"
        FROM product_information
       WHERE supplier_id = 102050;
```

```
SUPPLIER_ID MIN_PRICE             RATIO

----------- --------- -----------------

     102050       731  .695528068506185

     102050      null              null

     102050        73 .0694576593720266

     102050       247  .235014272121789

     102050      null              null


5 rows selected.
```

7.126 REGR_AVGX() OVER

语句

```
REGR_AVGX( expr1, expr2 ) OVER < window name or specification >
```

关于< window name or specification >的详细内容参考**window clause**

说明

Window function REGR_AVGX为线性回归函数 (linear regression function)

计算(X, Y)集合的least-squares-fit线性方程自变量expr2的平均值

expr1和expr2的参数为数字类型

expr1为因变量(y), expr2为自变量(x)

当expr1为NULL或expr2为NULL时从对象中排除

计算公式如下

```
AVG( expr2 )
```

返回类型为数字类型

返回值可以是计算得出的值或NULL

如果因expr1或expr2为NULL导致所有记录从对象中排除时返回NULL

使用示例

```
gSQL>
SELECT item_no,
       price,
       year,
       REGR_AVGX( price, year ) OVER ( PARTITION BY item_no )
       AS "REGR_AVGX(price,year)"
  FROM store;


ITEM_NO PRICE YEAR REGR_AVGX(price,year)
------- ----- ---- ---------------------
   3758 15000 2000                  2003
   3758 14700 2001                  2003
   3758 15300 2002                  2003
   3758 15200 2003                  2003
   3758 15100 2004                  2003
   3758 15300 2005                  2003
   3758 15350 2006                  2003


7 rows selected.
```

7.127 REGR_AVGY() OVER

语句

```
REGR_AVGY( expr1, expr2 ) OVER < window name or specification >
```

关于< window name or specification >的详细内容参考**window clause**

说明

Window function REGR_AVGY为线性回归函数 (linear regression function)

计算(X, Y)集合的least-squares-fit线性方程的因变量expr1的平均值

expr1和expr2的参数为数字类型

expr1是因变量(y), expr2是自变量(x)

当expr1为NULL或expr2为NULL时从对象中排除

计算公式如下

```
AVG( expr1 )
```

返回类型为数字类型

返回值可以是计算所得值或NULL

因expr1或expr2为NULL导致所有记录从对象中排除时返回NULL

使用示例

```
gSQL>
SELECT item_no,
       price,
       year,
       round( REGR_AVGY( price, year ) OVER ( PARTITION BY item_no ), 5 )
       AS "REGR_AVGY(price,year)"
  FROM store;


ITEM_NO PRICE YEAR REGR_AVGY(price,year)
------- ----- ---- ---------------------
   3758 15000 2000           15135.71429
   3758 14700 2001           15135.71429
   3758 15300 2002           15135.71429
   3758 15200 2003           15135.71429
   3758 15100 2004           15135.71429
   3758 15300 2005           15135.71429
   3758 15350 2006           15135.71429


7 rows selected.
```

7.128 REGR_COUNT() OVER

语句

```
REGR_COUNT( expr1, expr2 ) OVER < window name or specification >
```

关于< window name or specification >的详细内容参考**window clause**

说明

Window function REGR_COUNT是线性回归函数 (linear regression function)

计算(X, Y)集合的least-squares-fit线性方程

返回为线性方程执行对象的都是not NULL的(X, Y)对的数量

expr1和expr2的参数为数字类型

expr1为因变量(y), expr2为自变量(x)

当expr1为NULL或expr2为NULL时从对象中排除

返回类型为数字类型

返回expr1和expr2都是not NULL的对的数量

因为expr1或expr2为NULL导致所有记录从对象中排除时返回0

使用示例

```
gSQL>
SELECT item_no,
       price,
       year,
       REGR_COUNT( price, year ) OVER ( PARTITION BY item_no )
       AS "REGR_COUNT(price,year)"
  FROM store;


ITEM_NO PRICE YEAR REGR_COUNT(price,year)
------- ----- ---- ----------------------
   3758 15000 2000                      7
   3758 14700 2001                      7
   3758 15300 2002                      7
   3758 15200 2003                      7
   3758 15100 2004                      7
   3758 15300 2005                      7
   3758 15350 2006                      7


7 rows selected.
```

7.129 REGR_INTERCEPT() OVER

语句

```
REGR_INTERCEPT( expr1, expr2 ) OVER < window name or specification >
```

关于< window name or specification >的详细内容参考**window clause**

说明

Window function REGR_INTERCEPT为线性回归函数 (linear regression function)

计算(X, Y)集合的least-squares-fit线性方程的y截距

expr1和expr2的参数为数字类型

expr1为因变量(y), expr2为自变量(x)

expr1为NULL或expr2为NULL时从对象中排除

以下为计算公式

```
AVG( expr1 ) - REGR_SLOPE( expr1, expr2 ) * AVG( expr2 )
```

返回类型为数字类型

返回值可以是计算所得的值或NULL

如下情况返回NULL

- 因expr1或expr2为NULL导致所有记录从对象中排除时
- REGR_SLOPE的结果为null时

使用示例

```
gSQL>
SELECT item_no,
       price,
       year,
       REGR_INTERCEPT( price, year ) OVER ( PARTITION BY item_no )
       AS "REGR_INTERCEPT(price,year)"
  FROM store;

ITEM_NO PRICE YEAR REGR_INTERCEPT(price,year)
------- ----- ---- --------------------------
   3758 15000 2000                  -131512.5
   3758 14700 2001                  -131512.5
   3758 15300 2002                  -131512.5
   3758 15200 2003                  -131512.5
   3758 15100 2004                  -131512.5
   3758 15300 2005                  -131512.5
   3758 15350 2006                  -131512.5

7 rows selected.
```

7.130 REGR_R2() OVER

语句

```
REGR_R2( expr1, expr2 ) OVER < window name or specification >
```

关于< window name or specification >的详细内容参考**window clause**

说明

Window function REGR_R2是线性回归函数 (linear regression function)

计算(X, Y)集合的least-squares-fit线性方程的决定系数 (R-squared或适用性)

expr1和expr2的参数是数字类型

expr1是因变量(y), expr2是自变量(x)

当expr1为NULL或expr2为NULL时从对象中排除

计算公式如下

```
VAR_POP( expr2 ) = 0时,  NULL

VAR_POP( expr1 ) = 0, VAR_POP( expr2 ) != 0时, 1

VAR_POP( expr1 ) > 0, VAR_POP( expr2 ) != 0时, POWER( CORR( expr1, expr2),

2 )
```

返回类型为数字类型

返回值可以是计算所得值或NULL

以下情况返回NULL

- 因expr1或expr2为NULL导致所有记录从对象中排除时
- VAR_POP (expr2)的结果为0时

使用示例

```
gSQL>
SELECT item_no,
       price,
       year,
       round( REGR_R2( price, year ) OVER ( PARTITION BY item_no ), 10 )
       AS "REGR_R2(price,year)"
  FROM store
 WHERE item_no = 3758;


ITEM_NO PRICE YEAR REGR_R2(price,year)
------- ----- ---- --------------------
   3758 15000 2000         .4786446469
   3758 14700 2001         .4786446469
   3758 15300 2002         .4786446469
   3758 15200 2003         .4786446469
```

```
   3758 15100 2004         .4786446469

   3758 15300 2005         .4786446469

   3758 15350 2006         .4786446469


7 rows selected.
```

7.131 REGR_SLOPE() OVER

语句

```
REGR_SLOPE( expr1, expr2 ) OVER < window name or specification >
```

关于< window name or specification >的详细内容参考**window clause**

说明

Window function REGR_SLOPE为线性回归函数 (linear regression function)

计算(X, Y)集合的least-squares-fit线性方程的斜率

expr1和expr2的参数为数字类型

expr1为因变量(y), expr2为自变量(x)

当expr1为NULL或expr2为NULL时从对象中排除

计算公式如下

```
COVAR_POP(expr1, expr2) / VAR_POP(expr2)
```

返回类型为数字类型

返回值可以是计算所得值或NULL

以下情况返回NULL

- 因expr1或expr2为NULL导致所有记录从对象中排除时
- VAR_POP的结果为0时

使用示例

```
gSQL>
SELECT item_no,
       price,
       year,
       round( REGR_SLOPE( price, year ) OVER ( PARTITION BY item_no ), 10 )
       AS "REGR_SLOPE(price,year)"
  FROM store;

ITEM_NO PRICE YEAR REGR_SLOPE(price,year)
------- ----- ---- ----------------------
   3758 15000 2000          73.2142857143
   3758 14700 2001          73.2142857143
   3758 15300 2002          73.2142857143
   3758 15200 2003          73.2142857143
   3758 15100 2004          73.2142857143
   3758 15300 2005          73.2142857143
   3758 15350 2006          73.2142857143

7 rows selected.
```

7.132 REGR_SXX() OVER

语句

```
REGR_SXX( expr1, expr2 ) OVER < window name or specification >
```

关于< window name or specification >的详细内容参考**window clause**

说明

Window function REGR_SXX为线性回归函数(linear regression function)

是计算(X, Y)集合的least-squares-fit线性方程诊断统计的辅助函数

expr1和expr2的参数为数字类型

expr1是因变量(y), expr2是自变量(x)

当expr1为NULL或expr2为NULL时从对象中排除

计算公式如下

```
REGR_COUNT( expr1, expr2 ) * VAR_POP( expr2 )
```

返回类型为数字类型

返回值可以是计算所得值或NULL

因expr1或expr2为NULL导致所有记录从对象中排除时返回NULL

使用示例

```
gSQL>
SELECT item_no,
       price,
       year,
       REGR_SXX( price, year ) OVER ( PARTITION BY item_no )
       AS "REGR_SXX(price,year)"
  FROM store;


ITEM_NO PRICE YEAR REGR_SXX(price,year)
------- ----- ---- --------------------
   3758 15000 2000                   28
   3758 14700 2001                   28
   3758 15300 2002                   28
   3758 15200 2003                   28
   3758 15100 2004                   28
   3758 15300 2005                   28
   3758 15350 2006                   28


7 rows selected.
```

7.133 REGR_SXY() OVER

语句

```
REGR_SXY( expr1, expr2 ) OVER < window name or specification >
```

关于< window name or specification >的详细内容参考**window clause**

说明

Window function REGR_SXY为线性回归函数(linear regression function)

是计算(X, Y)集合的least-squares-fit线性方程的诊断统计的辅助函数

expr1和expr2的参数为数字类型

expr1为因变量(y), expr2为自变量(x)

当expr1为NULL或expr2为NULL时从对象中排除

计算公式如下

```
REGR_COUNT( expr1, expr2 ) * COVAR_POP( expr1, expr2 )
```

返回类型为数字类型

返回值可以是计算所得值或 NULL

因expr1或expr2为NULL导致所有记录从对象中排除时返回NULL

使用示例

```
gSQL>
SELECT item_no,
       price,
       year,
       REGR_SXY( price, year ) OVER ( PARTITION BY item_no )
       AS "REGR_SXY(price,year)"
  FROM store;


ITEM_NO PRICE YEAR REGR_SXY(price,year)
------- ----- ---- --------------------
   3758 15000 2000                 2050
   3758 14700 2001                 2050
   3758 15300 2002                 2050
   3758 15200 2003                 2050
   3758 15100 2004                 2050
   3758 15300 2005                 2050
   3758 15350 2006                 2050


7 rows selected.
```

7.134 REGR_SYY() OVER

语句

```
REGR_SYY( expr1, expr2 ) OVER < window name or specification >
```

关于< window name or specification >的详细内容参考**window clause**

说明

Window function REGR_SYY为线性回归函数(linear regression function)

是计算(X, Y)集合的least-squares-fit线性方程的诊断统计的辅助函数

expr1和expr2的参数为数字类型

expr1为因变量(y), expr2为自变量(x)

当expr1为NULL或expr2为NULL时从对象中排除

计算公式如下

```
REGR_COUNT( expr1, expr2 ) * VAR_POP( expr1 )
```

返回类型为数字类型

返回值可以是计算所得值或NULL

因expr1或expr2为NULL导致所有记录从对象中排除时返回NULL

使用示例

```
gSQL>
SELECT item_no,
       price,
       year,
       round( REGR_SYY( price, year ) OVER ( PARTITION BY item_no ), 4 )
       AS "REGR_SYY(price,year)"
  FROM store;


ITEM_NO PRICE YEAR REGR_SYY(price,year)
------- ----- ---- --------------------
   3758 15000 2000          313571.4286
   3758 14700 2001          313571.4286
   3758 15300 2002          313571.4286
   3758 15200 2003          313571.4286
   3758 15100 2004          313571.4286
   3758 15300 2005          313571.4286
   3758 15350 2006          313571.4286


7 rows selected.
```

7.135 REPEAT

语句

```
REPEAT( str, num )
```

说明

REPEAT函数返回将str重复与num指定的次数相同的string

str为CHARACTERCHARACTER VARYINGCHARACTER LONG VARYING等CHARCATER字符类型与BINARYBINARY VARYINGBINARY LONG VARYING等BINARY字符类型

num为数字类型

str或num中只要有一个为NULL时结果值也为NULL

num的值为0或负数时结果值为NULL

结果类型如下表

str类型	结果类型
CHAR或VARCHAR	VARCHAR
LONG VARCHAR	LONG VARCHAR

str类型	结果类型
BINARY或VARBINARY	VARBINARY
LONG VARBINARY	LONG VARBINARY

Table 7-15 REPEAT的结果类型

使用示例

```
gSQL> SELECT REPEAT( 'ab', 3 ) AS RESULT FROM DUAL;

RESULT

------

ababab

1 row selected.
```

7.136 REPLACE

语句

```
REPLACE( str, from, to )
```

说明

REPLACE将str string的所有from string替换为to string后返回

str, from, to为CHARACTERCHARACTER VARYINGCHARACTER LONG VARYING等CHARACTER字符类型

str值为NULL时结果值为NULL

from值为NULL时返回无转换的str值

省略to值或to值为NULL时返回从str中删除from的值

结果类型如下表

str类型	结果类型
CHAR或VARCHAR	VARCHAR
LONG VARCHAR	LONG VARCHAR

Table 7-16 REPLACE的结果类型

使用示例

```
gSQL> SELECT REPLACE( 'HI GLIESE', 'HI', 'HELLO' ) AS RESULT FROM DUAL;

RESULT

------------

HELLO GLIESE

1 row selected.
```

7.137 REVERSE

语句

```
REVERSE( str )
```

说明

REVERSE倒序返回str字符

字符类型为可转换为character string类型或binary string的类型character string类型是相应字符单位以byte为单位执行binary string类型

str为NULL时返回NULL

参数与结果类型如下表

str	结果类型
CHAR	CHAR
VARCHAR	VARCHAR
LONG VARCHAR	LONG VARCHAR
BINARY	BINARY
VARBINARY	VARBINARY

str	结果类型
LONG VARBINARY	LONG VARBINARY

Table 7-17　REVERSE参数与结果类型

使用示例

```
gSQL> SELECT REVERSE( 'SUNDB' ) AS RESULT FROM DUAL;


RESULT

----------

BDNUS


1 row selected.


gSQL> SELECT REVERSE( 'CSII 2018' ) AS RESULT FROM DUAL;


RESULT

---------------

8102 IISC


1 row selected.
```

7.138 ROUND(number)

语句

```
ROUND( num [, scale ] )
```

说明

ROUND返回以scale为准四舍五入num的值

numscale可以为数字类型

省略scale时scale为0与ROUND(num, 0)同时执行

scale为正数时以小数点右侧的位数为准进行四舍五入scale为负数时以小数点左侧的位数为准进行四舍五入

参数num或scale为NULL时返回NULL

使用示例

```
gSQL> SELECT ROUND( 152.4282, 2 ) AS RESULT FROM DUAL;
RESULT
------
```

```
152.43

1 row selected.


gSQL> SELECT ROUND( 152.4282, -2 ) AS RESULT FROM DUAL;

RESULT

------

   200

1 row selected.
```

7.139 ROUND(date)

语句

```
ROUND( date [ , fmt ] )
```

说明

ROUND（date）函数返回以指定的fmt为单位四舍五入date的值

date为DATE, TIMESTAMP, TIMESTAMP WITH TIME ZONE类型

fmt为CHARACTER, CHARACTER VARYING等CHARACTER字符类型

参数date或fmt为NULL时返回NULL

结果类型与接收的date类型无关始终为DATE类型

省略fmt时默认值为DAY可使用的格式字符串如下表

字符串	说明
CC, SCC	从51年开始四舍五入以四位数表示年份 (示例: XX01)
YYYY, YEAR, SYYYY, SYEAR, YYY, YY, Y	从7月1日开始四舍五入

IYYY, IYY, IY, I	使用ISO 8601标准定义的Calendar week的年份从7月1日开始四舍五入
Q	从季度的第二个月的16日开始四舍五入
MONTH, MON, MM, RM	从16日开始四舍五入
WW	每年的1月1日作为一周的开始从对应周的周三下午12点开始四舍五入
IW	ISO 8601标准定义的Calendar week（1-52周或1-53周）从周四下午12点开始四舍五入
W	月的1日作为一周的开始从对应周的周三下午12点开始四舍五入
DDD, DD, J	从下午12点开始四舍五入
DAY, DY, D	从每周的周三下午12点开始四舍五入
HH, HH12, HH24	从30分开始四舍五入
MI	从30秒开始四舍五入

Table 7-18 fmt中可使用的格式字符串

使用示例

```
gSQL> SELECT
      ROUND( TO_DATE( '2051-07-16', 'YYYY-MM-DD' ), 'CC' ) AS RESULT
      FROM DUAL;
```

```
RESULT

----------

2101-01-01

1 row selected.


gSQL> SELECT

      ROUND( TO_DATE( '2051-07-16', 'YYYY-MM-DD' ), 'YYYY' ) AS RESULT

      FROM DUAL;

RESULT

----------

2052-01-01

1 row selected.


gSQL> SELECT

      ROUND( TO_DATE( '2051-07-16', 'YYYY-MM-DD' ), 'MONTH' ) AS   RESULT

      FROM DUAL;

RESULT

----------

2051-08-01

1 row selected.


gSQL> SELECT

      ROUND( TO_TIMESTAMP( '2001-05-05 15:22:33.999999',

                           'YYYY-MM-DD HH24:MI:SS.FF6' ) ) AS RESULT

      FROM DUAL;
```

```
RESULT

----------

2001-05-06

1 row selected.
```

7.140 ROW_NUMBER() OVER

语句

```
ROW_NUMBER( ) OVER < window name or specification >
```

关于< window name or specification >的详细内容参考**window clause**

说明

Window function ROW_NUMBER给各row分配唯一编号

唯一编号是从1开始的连续整数

无法使用window frame

使用示例

```
gSQL> SELECT department_id, salary,
             ROW_NUMBER() OVER ( ORDER BY salary ) AS row_num
      FROM employees
      WHERE department_id = 60;


DEPARTMENT_ID SALARY ROW_NUM
```

```
------------ ------ -------

          60   4200       1

          60   4800       2

          60   4800       3

          60   6000       4

          60   9000       5


5 rows selected.
```

7.141 ROWID_GRID_BLOCK_ID

语句

```
ROWID_GRID_BLOCK_ID( rowid )
```

说明

返回GRID block ID的函数

> Note:

> 此信息在集群系统中有效

使用示例

```
gSQL> SELECT C1, ROWID_GRID_BLOCK_ID( ROWID ) FROM T1;

C1 ROWID_GRID_BLOCK_ID( ROWID )

-- ----------------------------

 1                            52

 2                            52

 3                            52


3 rows selected.
```

7.142 ROWID_GRID_BLOCK_SEQ

语句

```
ROWID_GRID_BLOCK_SEQ( rowid )
```

说明

返回GRID block sequence的函数

> Note:
>
> 此信息在集群系统中有效

使用示例

```
gSQL> SELECT C1, ROWID_GRID_BLOCK_SEQ( ROWID ) FROM T1;

C1 ROWID_GRID_BLOCK_SEQ( ROWID )

-- ------------------------------

 1                         747465

 2                         747466

 3                         747467


3 rows selected.
```

7.143 ROWID_MEMBER_ID

语句

```
ROWID_MEMBER_ID( rowid )
```

说明

返回member ID的函数

> Note:
>
> 此信息在集群系统中有效

使用示例

```
gSQL> SELECT C1, ROWID_MEMBER_ID( ROWID ) FROM T1;
C1 ROWID_MEMBER_ID( ROWID )
-- ------------------------
 1                        1
 2                        1
 3                        1
```

```
3 rows selected.
```

7.144 ROWID_OBJECT_ID

语句

```
ROWID_OBJECT_ID( rowid )
```

说明

返回object ID的函数

> Note:
>
> 此信息在集群系统中无效

使用示例

```
gSQL> SELECT ROWID_OBJECT_ID( t1.ROWID ) FROM t1;

ROWID_OBJECT_ID( T1.ROWID )
```

```
--------------------------
                       22012
                       22012
                       22012
                       22012
4 rows selected.
```

7.145 ROWID_PAGE_ID

语句

```
ROWID_PAGE_ID( rowid )
```

说明

返回page ID的函数

> Note:
>
> 此信息在集群系统中无效

使用示例

```
gSQL> SELECT ROWID_PAGE_ID( t1.ROWID ) FROM t1;

ROWID_PAGE_ID( T1.ROWID )

-------------------------

                      8227

                      8227

                      8227

                      8227

4 rows selected.
```

7.146 ROWID_ROW_NUMBER

语句

```
ROWID_ROW_NUMBER( rowid )
```

说明

返回row number的函数

> Note:
>
> 此信息在集群系统中无效

使用示例

```
gSQL> SELECT ROWID_ROW_NUMBER( t1.ROWID ) FROM t1;
ROWID_ROW_NUMBER( T1.ROWID )
----------------------------
                           0
                           1
                           2
                           3
```

```
4 rows selected.
```

7.147 ROWID_SHARD_ID

语句

```
ROWID_SHARD_ID( rowid )
```

说明

返回shard ID的函数

> Note:
>
> 此信息在集群系统中有效

使用示例

```
gSQL> SELECT C1, ROWID_SHARD_ID( ROWID ) FROM T1;

C1 ROWID_SHARD_ID( ROWID )
```

```
-- ----------------------

 1                       0

 2                       1

 3                       2


3 rows selected.
```

7.148 ROWID_TABLESPACE_ID

语句

```
ROWID_TABLESPACE_ID( rowid )
```

说明

返回tablespace ID的函数

> Note:
>
> 此信息在集群系统中无效

使用示例

```
gSQL> SELECT ROWID_TABLESPACE_ID( t1.ROWID ) FROM t1;

ROWID_TABLESPACE_ID( T1.ROWID )

------------------------------

                             2

                             2

                             2

                             2

4 rows selected.
```

7.149 ROWNUM

语句

```
ROWNUM
```

说明

对满足WHERE条件的row从1开始依次赋予编号

为了与Oracle兼容允许在WHERE语句中使用ROWNUM

但是要限制查询结果的数量时推荐使用如下SQL标准的**offset limit clause**

- (非标准）使用ROWNUM设置结果数量时

```
gSQL> SELECT * FROM t1 WHERE ROWNUM <= 3;


C1
--
A
B
C


3 rows selected.
```

- (SQL标准）使用FETCH语句设置结果数量时

```
gSQL> SELECT * FROM t1 FETCH 3;


C1
--
A
B
C


3 rows selected.
```

需要限制查询结果的部分范围时推荐使用如下OFFSET, FETCH语句

- (非标准）使用ROWNUM设置结果数量的范围时

```
gSQL> SELECT c1
        FROM ( SELECT ROWNUM rn, c1
                 FROM t1 )
        WHERE rn BETWEEN 2 AND 3;


C1
--
B
C
```

```
2 rows selected.
```

- (SQL标准）使用OFFSETFETCH语句设置结果数量的范围时

```
gSQL> SELECT c1 FROM t1 OFFSET 1 FETCH 2;


C1
--
B
C


2 rows selected.
```

在WHERE语句的ROWNUM不推荐用于结果数量限制外的其他用途

如下所示使用模糊条件（WHERE c1 < ROWNUM + 3）时相同的查询根据执行方式结果会有所不同

- 生成数据

```
CREATE TABLE t1 ( c1 INTEGER );
CREATE INDEX t1_idx ON t1(c1);
INSERT INTO t1 VALUES (1);
INSERT INTO t1 VALUES (2);
INSERT INTO t1 VALUES (3);
INSERT INTO t1 VALUES (4);
INSERT INTO t1 VALUES (5);
```

```
COMMIT;
```

- Oracle的情况

```
SQL> SELECT ROWNUM, c1 FROM t1 WHERE c1 < ROWNUM + 3;


    ROWNUM       C1
---------- ----------
     1        1
     2        2


SQL> DROP INDEX t1_idx;
```

  - 索引被删除

```
SQL> SELECT ROWNUM, c1 FROM t1 WHERE c1 < ROWNUM + 3;


    ROWNUM       C1
---------- ----------
     1        1
     2        2
     3        3
     4        4
     5        5
```

- SUNDB的情况

```
gSQL> SELECT ROWNUM, c1 FROM t1 WHERE c1 < ROWNUM + 3;


ROWNUM C1

------ --

     1  1

     2  2

     3  3

     4  4

     5  5


5 rows selected.


gSQL> DROP INDEX t1_idx;


Index dropped.


gSQL> SELECT ROWNUM, c1 FROM t1 WHERE c1 < ROWNUM + 3;


ROWNUM C1

------ --

     1  1

     2  2

     3  3

     4  4

     5  5
```

```
5 rows selected.
```

使用示例

```
gSQL> SELECT ROWNUM, c1 FROM t1;


ROWNUM C1

------ --

     1 A

     2 B

     3 C

     4 D

     5 E


5 rows selected.
```

7.150 RPAD

语句

```
RPAD( str, length, [, fill] )
```

说明

RPAD函数返回在str右侧填充字符串fill直到sting的长度为length的值

str为CHARACTERCHARACTER VARYINGCHARACTER LONG VARYING等CHARACTER字符类型与BINARYBINARY VARYINGBINARY LONG VARYING等BINARY字符类型

length为数字类型

length为字符的数量最大范围为结果类型的最大PRECISION

省略fill时填充空字符

str长度大于length时str减掉与length相同的数量后返回

strlengthfill中只要有一个为NULL则结果值为NULLlength为0或负数时结果值也为NULL

结果类型如下表

str的类型	结果类型
CHAR或VARCHAR	VARCHAR

str的类型	结果类型
LONG VARCHAR	LONG VARCHAR
BINARY或VARBINARY	VARBINARY
LONG VARBINARY	LONG VARBINARY

Table 7-19 PRAD的结果类型

使用示例

```
gSQL> SELECT RPAD('AA', 5) AS RESULT1,
             RPAD('AA', 5, 'X') AS RESULT2,
             RPAD('AA', 1) AS RESULT3
      FROM DUAL;
RESULT1 RESULT2 RESULT3
------- ------- -------
AA      AAXXX   A
1 row selected.
```

7.151 RTRIM

语句

```
RTRIM( trim_source [, trim_character ] )
```

说明

RTRIM函数返回从trim_source的右侧开始对比并删除trim_character直到没有一致的字符的结果

trim_character与trim_source为CHARACTERCHARACTER VARYINGCHARACTER LONG VARYING等CHARACTER字符类型与BINARYBINARY VARYINGBINARY LONG VARYING等BINARY字符类型

trim_charactertrim_source中只要有一个为NULL则结果值为NULL

省略trim_character时默认指定为single blank space(' ')

结果类型如下表

trim_source, trim_character类型	结果类型
CHAR或VARCHAR	VARCHAR
LONG VARCHAR	LONG VARCHAR
BINARY或VARBINARY	VARBINARY
LONG VARBINARY	LONG VARBINARY

Table 7-20 RTRIM的结果类型

使用示例

```
gSQL> SELECT RTRIM('      rtrim      ') AS RESULT1,
             RTRIM('______rtrim______','_') AS RESULT2
      FROM DUAL;
RESULT1     RESULT2
----------- -----------
      rtrim ______rtrim
1 row selected.
```

7.152 SESSION_ID

语句

```
SESSION_ID()
```

说明

获取当前会话的ID

使用示例

```
gSQL> SELECT SESSION_ID() FROM dual;


SESSION_ID()
------------
           4


1 row selected.
```

7.153 SESSION_SERIAL

语句

```
SESSION_SERIAL()
```

说明

获取当前会话的serial编号

使用示例

```
gSQL> SELECT SESSION_SERIAL() FROM dual;


SESSION_SERIAL()
----------------
              16


1 row selected.
```

7.154 SESSION_USER

语句

```
SESSION_USER[()]
```

说明

返回会话用户

用如下三种形式管理用户信息

- Logon user：执行login的用户维持到关闭连接
- Session user：与最初的logon user相同可以用SET SESSION AUTHORIZATION语句进行变更
- Current user：一般情况下与session user相同但使用PSM, View时为了控制访问在系统内部暂时变更
  - Session user与current user的差异与unix系统的real user与effective user的差异类似

使用示例

```
% gsql sys gliese

gSQL>  SET SESSION AUTHORIZATION test;
```

```
Session set.


gSQL> SELECT LOGON_USER() AS result1, SESSION_USER() AS result2 FROM DUAL;


RESULT1 RESULT2

------- -------

SYS     TEST


1 row selected.
```

7.155 SESSIONTIMEZONE

语句

```
SESSIONTIMEZONE()
```

说明

SESSIONTIMEZONE返回当前会话的time zone

返回值是类型为'[+|-]TZH:TZM' format的文本

返回类型为varchar

使用示例

查看当前session的time zone

```
gSQL> SELECT SESSIONTIMEZONE() FROM dual;


SESSIONTIMEZONE()

-----------------

+09:00


1 row selected.
```

变更当前session的time zone时返回变更后的time zone值

```
gSQL> SET TIME ZONE '-05:00';


Session set.


gSQL> SELECT SESSIONTIMEZONE() FROM dual;


SESSIONTIMEZONE()

-----------------

-05:00


1 row selected.
```

7.156 SHARD_GROUP_ID

语句

```
SHARD_GROUP_ID( table_name, shard_key_value [, ... ] )
```

说明

SHARD_GROUP_ID函数在table_name定义了shard strategy时返回管理可存储shared_key_value值的shard的group ID

输入参数table_name应为identifier对应table_name的对象并非base table或未定义shard strategy时报错

输入参数shard_key_value应按照table_name中定义的shard strategy的shard key column顺序排列

shard_key_value数量与shard key column数量不一致时报错

结果类型为NATIVE_BIGINT

> Note:
>
> 此信息在集群系统中有效

使用示例

```
gSQL> SELECT T1.C1, SHARD_GROUP_ID( T1, T1.C1 ) FROM T1;

C1 SHARD_GROUP_ID( T1, T1.C1 )

-- ---------------------------

A                             1

B                             2

C                             3


3 rows selected.



gSQL> SELECT SHARD_GROUP_ID( T1, 'B' ) FROM DUAL;

SHARD_GROUP_ID( T1, 'B' )

-------------------------

                        2


1 row selected.
```

7.157 SHARD_GROUP_NAME

语句

```
SHARD_GROUP_NAME( table_name, shard_key_value [, ... ] )
```

说明

SHARD_GROUP_NAME函数在table_name定义了shard strategy时返回管理可存储

shared_key_value值的shard的group NAME

输入参数table_name应为identifier对应table_name的对象并非base table或未定义shard strategy时报错

输入参数shard_key_value应按照table_name中定义的shard strategy的shard key column顺序排列

shard_key_value数量与shard key column数量不一致时报错

结果类型为VARCHAR

> Note:
>
> 此信息在集群系统中有效

使用示例

```
gSQL> SELECT T1.C1, SHARD_GROUP_NAME( T1, T1.C1 ) FROM T1;


C1 SHARD_GROUP_NAME( T1, T1.C1 )

-- ----------------------------

A  G1

B  G2

C  G3


3 rows selected.


gSQL> SELECT SHARD_GROUP_NAME( T1, 'B' ) FROM DUAL;


SHARD_GROUP_NAME( T1, 'B' )

---------------------------

G2


1 row selected.
```

7.158 SHARD_ID

语句

```
SHARD_ID( table_name, shard_key_value [, ... ] )
```

说明

SHARD_ID函数返回在table_name定义了shard strategy时可存储shared_key_value值的shard的ID

输入参数table_name应为identifier对应table_name的对象并非base table或未定义shard strategy时报错

输入参数shard_key_value应按照table_name中定义的shard strategy的shard key column顺序排列

shard_key_value数量与shard key column数量不一致时报错

结果类型为NATIVE_BIGINT

> Note:
>
> 此信息在集群系统中有效

使用示例

```
gSQL> SELECT T1.C1, SHARD_ID( T1, T1.C1 ) FROM T1;

C1 SHARD_ID( T1, T1.C1 )

-- --------------------

A                      0

B                      1

C                      2


3 rows selected.



gSQL> SELECT SHARD_ID( T1, 'B' ) FROM DUAL;

SHARD_ID( T1, 'B' )

-------------------

                   1


1 row selected.
```

7.159 SHARD_NAME

语句

```
SHARD_NAME( table_name, shard_key_value [, ... ] )
```

说明

SHARD_NAME函数返回在table_name定义了shard strategy时可存储shared_key_value值的shard的NAME

输入参数table_name应为identifier对应table_name的对象并非base table或未定义shard strategy时报错

输入参数shard_key_value应按照table_name中定义的shard strategy的shard key column顺序排列shard_key_value数量与shard key column数量不一致时报错

结果类型为VARCHAR

> Note:
>
> 此信息在集群系统中有效

使用示例

```
gSQL> SELECT T1.C1, SHARD_NAME( T1, T1.C1 ) FROM T1;


C1 SHARD_NAME( T1, T1.C1 )
-- ----------------------
A  S1
B  S2
C  S3


3 rows selected.


gSQL> SELECT SHARD_NAME( T1, 'B' ) FROM DUAL;


SHARD_NAME( T1, 'B' )
---------------------
S2


1 row selected.
```

7.160 SHIFT_LEFT

语句

```
SHIFT_LEFT( num, cnt )
```

说明

SHIFT_LEFT函数返回num左移cnt位的值

输入参数numcnt为NATIVE_SMALLINTNATIVE_INTEGERNATIVE_BIGINT或可转换为NATIVE_BIGINT的类型

转换为NATIVE_BIGINT类型时小数点将被TRUNCATE

运算时cnt以6bit计算处理为6bit范围内的值

num或cnt为NULL时返回NULL

结果类型为NATIVE_BIGINT

使用示例

```
gSQL> SELECT SHIFT_LEFT(1, 3) FROM DUAL;

SHIFT_LEFT(1, 3)
```

```
----------------

               8

1 row selected.
```

7.161 SHIFT_RIGHT

语句

```
SHIFT_RIGHT( num, cnt )
```

说明

SHIFT_RIGHT函数num右移cnt位的值

输入参数numcnt为NATIVE_SMALLINTNATIVE_INTEGERNATIVE_BIGINT或可转换为NATIVE_BIGINT的类型

转换为NATIVE_BIGINT类型时小数点将被TRUNCATE

运算时cnt以6bit计算处理为6bit范围内的值

num或cnt为NULL时返回NULL

结果类型为NATIVE_BIGINT

使用示例

```
gSQL> SELECT SHIFT_RIGHT(8, 3) FROM DUAL;

SHIFT_RIGHT(8, 3)

-----------------

                1

1 row selected.
```

7.162 SIGN

语句

```
SIGN( num )
```

说明

SIGN函数返回num的符号

num为数字类型

返回值如下

- num < 0时-1
- num = 0时0
- num > 0时1

num为NULL时返回NULL

使用示例

```
gSQL> SELECT SIGN(-10) AS RESULT1,
             SIGN(0) AS RESULT2,
             SIGN(10) AS RESULT3 FROM DUAL;
RESULT1 RESULT2 RESULT3
------- ------- -------
     -1       0       1
1 row selected.
```

7.163 SIN

语句

```
SIN( num )
```

说明

SIN函数返回num的sine值

null为NULL时返回NULL

使用示例

```
gSQL> SELECT SIN( 0 ) AS RESULT FROM DUAL;

RESULT

------

     0

1 row selected.
```

7.164 SPLIT_PART

语句

```
SPLIT_PART( string, delimiter, field )
```

说明

SPLIT_PART函数在string中以指定为delimiter的字符为分隔符返回field的字符串

stringdelimiter为CHARACTERCHARACTER VARYINGCHARACTER LONG VARYING的CHARACTER字符类型

field为数字类型

string, delimiter, field中只要有一个为NULL结果值也为NULL

field只能为1以上的数字值为0或负数时报错

结果类型如下表

string类型	结果类型
CHAR或VARCHAR	VARCHAR
LONG VARCHAR	LONG VARCHAR

Table 7-21 SPLIT_PART的结果类型

使用示例

```
gSQL> SELECT SPLIT_PART( 'AB;CD;EF;GH', ';', 3  ) AS RESULT FROM DUAL;

RESULT

------

EF

1 row selected.
```

7.165 SQRT

语句

```
SQRT( num )
```

说明

SQRT函数返回num的平方根

num为数字类型应为0以上的值

num为NULL时返回NULL

使用示例

```
gSQL> SELECT SQRT( 9 ) AS RESULT FROM DUAL;
RESULT
------
     3
1 row selected.
```

7.166 STATEMENT_DATE

语句

```
STATEMENT_DATE()

CURRENT_DATE [()]
```

说明

获取当前日期（DATE type）值

获取当前日期的函数之间有如下差异

- TRANSACTION_DATE()：事务内的所有日期值均相同
- STATEMENT_DATE()：一个SQL语句内的所有日期值均相同
- CLOCK_DATE()：调用函数时每次均返回当前日期值

使用示例

```
gSQL> SELECT STATEMENT_DATE() AS result FROM t1;


RESULT

----------
```

```
2013-12-12

2013-12-12

2013-12-12


3 rows selected.
```

7.167 STATEMENT_LOCALTIME

语句

```
STATEMENT_LOCALTIME()

LOCALTIME [()]
```

说明

以会话时间为准获取当前TIME WITHOUT TIME ZONE type值

LOCALTIME为SQL标准函数

获取当前时间的函数之间有如下差异

- TRANSACTION_LOCALTIME()：事务内的所有时间值均相同
- STATEMENT_LOCALTIME()：一个SQL语句内的所有时间值均相同
- CLOCK_LOCALTIME()：调用函数时每次均获得当前时间值

使用示例

所有row都具有相同的时间值

```
gSQL> SELECT STATEMENT_LOCALTIME() AS result FROM t1;


RESULT

---------------

16:18:50.775870

16:18:50.775870

16:18:50.775870


3 rows selected.
```

7.168 STATEMENT_LOCALTIMESTAMP

语句

```
STATEMENT_LOCALTIMESTAMP()

LOCALTIMESTAMP [()]
```

说明

以会话时间为准获取当前TIMESTAMP WITHOUT TIME ZONE type值

LOCALTIMESTAMP为SQL标准函数

获取当前TIMESTAMP的函数之间有如下差异

- TRANSACTION_LOCALTIMESTAMP()：事务内的所有TIMESTAMP值均相同
- STATEMENT_LOCALTIMESTAMP()：一个SQL语句内的所有TIMESTAMP值均相同
- CLOCK_LOCALTIMESTAMP()：调用函数时每次均获得当前TIMESTAMP值

使用示例

所有row都具有相同的值

```
gSQL> SELECT STATEMENT_LOCALTIMESTAMP() FROM t1;


STATEMENT_LOCALTIMESTAMP()

--------------------------

2013-12-12 16:23:39.782187

2013-12-12 16:23:39.782187

2013-12-12 16:23:39.782187


3 rows selected.
```

7.169 STATEMENT_TIME

语句

```
STATEMENT_TIME()

CURRENT_TIME [()]
```

说明

获取有TIME ZONE的当前时间(TIME WITH TIME ZONE type)值

CURRENT_TIME为SQL标准函数

获取当前时间的函数之间有如下差异

- TRANSACTION_TIME()：事务内的所有时间值均相同
- STATEMENT_TIME()：一个SQL语句内的所有时间值均相同
- CLOCK_TIME()：调用函数时每次均获得当前时间值

使用示例

所有row都具有相同的时间值

```
gSQL> SELECT STATEMENT_TIME() AS result FROM t1;


RESULT

----------------------

16:28:19.268513 +09:00

16:28:19.268513 +09:00

16:28:19.268513 +09:00


3 rows selected.
```

7.170 STATEMENT_TIMESTAMP

语句

```
STATEMENT_TIMESTAMP()

CURRENT_TIMESTAMP [()]
```

说明

获取有TIME ZONE的当前TIMESTAMP (TIMESTAMP WITH TIME ZONE type)值

CURRENT_TIMESTAMP为SQL标准函数

获取当前TIMESTAMP的函数之间有如下差异

- TRANSACTION_TIMESTAMP()：事务内的所有TIMESTAMP值均相同
- STATEMENT_TIMESTAMP()：一个SQL语句内的所有TIMESTAMP值均相同
- CLOCK_TIMESTAMP()：调用函数时每次均获得当前TIMESTAMP值

使用示例

所有row都具有相同的值

```
gSQL> SELECT STATEMENT_TIMESTAMP() AS result FROM t1;


RESULT

---------------------------------

2013-12-12 16:36:11.032957 +09:00

2013-12-12 16:36:11.032957 +09:00

2013-12-12 16:36:11.032957 +09:00


3 rows selected.
```

7.171 STATEMENT_VIEW_SCN

语句

```
STATEMENT_VIEW_SCN()
```

说明

获取当前STATEMENT的VIEW SCN

使用示例

```
gSQL> SELECT STATEMENT_VIEW_SCN() FROM dual;


STATEMENT_VIEW_SCN()

--------------------

17697.658.17880


1 row selected.
```

7.172 STATEMENT_VIEW_SCN_DCN

语句

```
STATEMENT_VIEW_SCN_DCN()
```

说明

获取当前STATEMENT的VIEW SCN的Domain Change Number（DCN）值

使用示例

```
gSQL> SELECT STATEMENT_VIEW_SCN_DCN() FROM dual;


STATEMENT_VIEW_SCN_DCN()
------------------------
                     658


1 row selected.
```

7.173 STATEMENT_VIEW_SCN_GCN

语句

```
STATEMENT_VIEW_SCN_GCN()
```

说明

获取当前STATEMENT的VIEW SCN的Global Change Number（GCN）值

使用示例

```
gSQL> SELECT STATEMENT_VIEW_SCN_GCN() FROM dual;


STATEMENT_VIEW_SCN_GCN()
------------------------
                   17697


1 row selected.
```

7.174 STATEMENT_VIEW_SCN_LCN

语句

```
STATEMENT_VIEW_SCN_LCN()
```

说明

获取当前STATEMENT的VIEW SCN的Local Change Number（LCN）值

使用示例

```
gSQL> SELECT STATEMENT_VIEW_SCN_LCN() FROM dual;


STATEMENT_VIEW_SCN_LCN()
------------------------
                   17880


1 row selected.
```

7.175 STDDEV

语句

```
STDDEV( [ ALL | DISTINCT ] expr )
```

说明

是聚合操作函数获取expr set的标准偏差（standard deviation）

指定ALL时对所有值执行该操作指定DISTINCT时仅对删除重复的值执行该操作未指定时处理方式与指定ALL相同

除NULL值外以DISTINCT删除重复后的expr set的数量为1时与 **VARIANCE**相同返回0

参数与结果类型如下图所示

expr	结果类型
NATIVE_INTEGER系列 • NATIVE_SMALLINT • NATIVE_INTEGER • NATIVE_BIGINT	NATIVE_DOUBLE

expr	结果类型
NUMBER	NUMBER
NATIVE_DOUBLE系列 • NATIVE_REAL • NATIVE_DOUBLE	NATIVE_DOUBLE

Table 7-22　STDDEV参数与结果类型

SUNDB的标准偏差的计算方式如下

• expr set的数量为1时返回0

• expr set的数量大于1时返回**STDDEV_SAMP(expr)**值

Note:

标准偏差为方差平方根需要在方差加上方根进行计算

即STDDEV函数与在 **VARIANCE**函数加上方根相同

STDDEV([ALL] expr)

= SQRT(VARIANCE([ALL] expr))

STDDEV(DISTINCT expr)

= SQRT(VARIANCE(DISTINCT expr))

使用示例

```
gSQL> SELECT STDDEV(c1) FROM t1;


      STDDEV(C1)
-----------------
11.4978258814438


1 row selected.


gSQL> SELECT STDDEV(ALL c1) FROM t1;


  STDDEV(ALL C1)
-----------------
11.4978258814438


1 row selected.


gSQL> SELECT STDDEV(DISTINCT c1) FROM t1;


STDDEV(DISTINCT C1)
-------------------
   13.2759180473518


1 row selected.
```

7.176 STDDEV() OVER

语句

```
STDDEV ( expr ) OVER < window name or specification >
```

关于< window name or specification >的详细内容参考**window clause**

说明

Window function STDDEV是计算expr的标准偏差 (standard deviation)的函数

如果去除NULL值的expr数量为一个则返回结果为0

使用示例

```
gSQL> SELECT min_price AS "MIN_PRICE"
             , STDDEV( min_price ) OVER ( ORDER BY min_price ) AS "STDDEV"
        FROM product_information
       WHERE supplier_id = 102050;


MIN_PRICE           STDDEV
--------- ----------------
       73                0
```

```
      247 123.036579926459

      731 340.953564775811

     null 340.953564775811

     null 340.953564775811


5 rows selected.
```

7.177 STDDEV_POP

语句

```
STDDEV_POP( expr )
```

说明

为聚合操作函数获取expr set的总体标准偏差（population standard deviation）

除NULL值外expr set的数量为1时返回0

参数与结果类型如下图所示

expr	结果类型
NATIVE_INTEGER系列 • NATIVE_SMALLINT • NATIVE_INTEGER • NATIVE_BIGINT	NATIVE_DOUBLE
NUMBER	NUMBER

expr	结果类型
NATIVE_DOUBLE系列 • NATIVE_REAL • NATIVE_DOUBLE	NATIVE_DOUBLE

Table 7-23　STDDEV_POP的参数与结果类型

> Note:
>
> 总体标准偏差为总体方差平方根需要在总体方差加上方根进行计算
>
> 即STDDEV_POP函数与在 **VAR_POP**函数加上方根相同
>
> STDDEV_POP(expr)
>
> = SQRT(VAR_POP(expr))

使用示例

```
gSQL> SELECT STDDEV_POP(c1) FROM t1;


 STDDEV_POP(C1)

---------------

10.283968105746


1 row selected.
```

7.178 STDDEV_POP() OVER

语句

```
STDDEV_POP ( expr ) OVER < window name or specification >
```

关于< window name or specification >的详细内容参考**window clause**

说明

Window function STDDEV_POP是计算expr的总体标准偏差 (population standard deviation)的函数

去除NULL值以外的expr的数量为一个时返回结果为0

使用示例

```
gSQL> SELECT min_price AS "MIN_PRICE"
             , STDDEV_POP( min_price ) OVER ( ORDER BY min_price ) AS
"STDDEV_POP"
         FROM product_information
        WHERE supplier_id = 102050;


MIN_PRICE      STDDEV_POP
```

```
--------- ---------------

       73                0

      247               87

      731 278.38741989457

     null 278.38741989457

     null 278.38741989457


5 rows selected.
```

7.179 STDDEV_SAMP

语句

```
STDDEV_SAMP( expr )
```

说明

为聚合操作函数获取expr set样本标准偏差（sample standard deviation）

除NULL值外expr set的数量为1时返回NULL值

参数与结果类型如下图所示

expr	结果类型
NATIVE_INTEGER系列 • NATIVE_SMALLINT • NATIVE_INTEGER • NATIVE_BIGINT	NATIVE_DOUBLE
NUMBER	NUMBER

expr	结果类型
NATIVE_DOUBLE系列 • NATIVE_REAL • NATIVE_DOUBLE	NATIVE_DOUBLE

Table 7-24　STDDEV_SAMP的参数与结果类型

> Note:
>
> 样本标准偏差为样本方差平方根需要在样本方差加上方根进行计算
>
> 即STDDEV_SAMP函数与在 **VAR_SAMP**函数加上方根相同
>
> STDDEV_SAMP(expr)
>
> = SQRT(VAR_SAMP(expr))

使用示例

```
gSQL> SELECT STDDEV_SAMP(c1) FROM t1;

 STDDEV_SAMP(C1)

-----------------

11.4978258814438


1 row selected.
```

7.180 STDDEV_SAMP() OVER

语句

```
STDDEV_SAMP ( expr ) OVER < window name or specification >
```

关于< window name or specification >的详细内容参考**window clause**

说明

Window function STDDEV_SAMP是计算expr样本标准偏差 (sample standard deviation)的函数

去除NULL值的expr数量为一个时则返回结果为NULL

使用示例

```
gSQL> SELECT min_price AS "MIN_PRICE"
             , STDDEV_SAMP( min_price ) OVER ( ORDER BY min_price ) AS
"STDDEV_SAMP"
        FROM product_information
       WHERE supplier_id = 102050;


MIN_PRICE      STDDEV_SAMP
---------- ----------------
```

```
       73              null

      247 123.036579926459

      731 340.953564775811

     null 340.953564775811

     null 340.953564775811


5 rows selected.
```

7.181 STRING_AGG() OVER

语句

```
STRING_AGG( str [, delimiter] ) OVER < window name or specification >
```

关于< window name or specification >的详细内容参考**window clause**

说明

Window function STRING_AGG是根据OVER子句中定义的函数的执行范围连接str的函数

str为NULL时排除

delimiter是str连接分隔符省略时默认值为NULL

str可以为character string或binary string

如果str为character string则结果类型为varchar

如果str为binary string，则结果类型为varbinary

使用示例

```
gSQL>

SELECT regionkey,
```

```
       name,
       STRING_AGG( name ) OVER ( PARTITION BY regionkey
                                 ORDER BY nationkey
                                 ROWS BETWEEN UNBOUNDED PRECEDING
                                          AND CURRENT ROW )
       AS "STRING_AGG( name ) OVER",
       STRING_AGG( name, ', ' ) OVER ( PARTITION BY regionkey
                                       ORDER BY nationkey
                                       ROWS BETWEEN UNBOUNDED PRECEDING
                                                AND CURRENT ROW )
       AS "STRING_AGG( name, ', ' ) OVER"
  FROM nation;


REGIONKEY NAME    STRING_AGG( name ) OVER STRING_AGG( name, ', ' ) OVER
--------- ------- ----------------------- -----------------------------
        1 BRAZIL  BRAZIL                  BRAZIL
        1 CANADA  BRAZILCANADA            BRAZIL, CANADA
        1 PERU    BRAZILCANADAPERU        BRAZIL, CANADA, PERU
        1 null    BRAZILCANADAPERU        BRAZIL, CANADA, PERU
        2 null    null                    null
        2 INDIA   INDIA                   INDIA
        2 null    INDIA                   INDIA
        2 null    INDIA                   INDIA
        2 JAPAN   INDIAJAPAN              INDIA, JAPAN
        2 CHINA   INDIAJAPANCHINA         INDIA, JAPAN, CHINA
```

```
        2 null    INDIAJAPANCHINA          INDIA, JAPAN, CHINA

        2 VIETNAM INDIAJAPANCHINAVIETNAM   INDIA, JAPAN, CHINA, VIETNAM

        3 EGYPT   EGYPT                    EGYPT

        3 IRAN    EGYPTIRAN                EGYPT, IRAN

        3 IRAQ    EGYPTIRANIRAQ            EGYPT, IRAN, IRAQ


15 rows selected.
```

7.182 SUBSTR

语句

```
SUBSTR( str FROM start_position [ FOR string_length ] )

SUBSTR( str, start_position [ , string_length ] )
```

说明

是**SUBSTRING**的alias

使用示例

- Multi byte character set (例: UTF8): 1 byte character

```
gSQL> SELECT

      SUBSTR( 'DATABASE MANAGEMENT SYSTEM', 10, 10 ) AS RESULT

      FROM DUAL;

RESULT

----------

MANAGEMENT

1 row selected.
```

- Multi byte character set (例: UTF8): 2 bytes or 3 bytes character

```
gSQL> SELECT SUBSTR( '“αβ≠AB”', 2, 5 ) AS RESULT FROM DUAL;

RESULT

------

αβ≠AB

1 row selected.
```

7.183 SUBSTRB

语句

```
SUBSTRB( str, start_position [ , string_length ] )
```

说明

SUBSTRB函数返回从str的start_position开始到string_length范围中抽取的字符

SUBSTRB函数除了start_position与string_length以byte为单位计算外与**SUBSTRING**函数相同

使用示例

- Multi byte character set (例: UTF8): 1 byte character

```
gSQL> SELECT
      SUBSTRB( 'DATABASE MANAGEMENT SYSTEM', 10, 10 ) AS RESULT
      FROM DUAL;
RESULT
----------
MANAGEMENT
1 row selected.
```

- Multi byte character set (例: UTF8): 2 bytes or 3 bytes character

```
gSQL> SELECT SUBSTRB( '“αβ≠AB”', 4, 11 ) AS RESULT FROM DUAL;

RESULT

------

αβ≠AB

1 row selected.
```

7.184 SUBSTRING

语句

```
SUBSTRING( str FROM start_position [ FOR string_length ] )

SUBSTRING( str, start_position [ , string_length ] )
```

说明

SUBSTRING函数返回从str的start_position开始到string_length的范围中抽取的字符

str为CHARACTERCHARACTER VARYINGCHARACTER LONG VARYING等CHARACTER字符类型与BINARYBINARY VARYINGBINARY LONG VARYING等BINARY字符类型

start_position与string_length为数字类型

strstart_position和string_length中只要有一个为NULL则结果值也为NULL

start_position与string_length从1开始按character set的字符为单位计算（不以byte为单位）

start_position为0时处理为1

start_position为正数时从str的前面开始查找position的位置

start_position为负数时从str的后面开始查找position的位置

省略string_length时返回从start_position开始到str的最后一个字符

string_length为0或负数时结果值为NULL

start_position > (str的长度)时结果值为NULL

(str的长度+ start_position) < 0时结果值为NULL

SUBSTRING的aliase

参考：**SUBSTR**, **SUBSTRB**

结果类型如下表

str	结果类型
CHAR或VARCHAR	VARCHAR
LONG VARCHAR	LONG VARCHAR
BINARY或VARBINARY	VARBINARY
LONG VARBINARY	LONG VARBINARY

Table 7-25 SUBSTRING的结果类型

使用示例

- Multi byte character set (例: UTF8): 1 byte character

```
gSQL> SELECT
      SUBSTRING( 'DATABASE MANAGEMENT SYSTEM' FROM 10 FOR 10 ) AS RESULT
      FROM DUAL;
RESULT
----------
MANAGEMENT
```

```
1 row selected.
```

- Multi byte character set (例: UTF8): 2 bytes or 3 bytes character

```
gSQL> SELECT SUBSTRING( '“αβ≠AB”' FROM 2 FOR 5 ) AS RESULT FROM DUAL;

RESULT

------

αβ≠AB

1 row selected.
```

7.185 SUM

语句

```
SUM( [ ALL | DISTINCT ] expr )
```

说明

为聚合操作函数获取expr值的和

指定ALL时对所有值执行聚合操作

指定DISTINCT时对删除重复的值执行聚合操作

未明确指定ALL或DISTINCT时与指定ALL的处理方式相同

使用示例

```
gSQL> SELECT SUM(c1) FROM t1;


SUM(C1)

-------

      6


1 row selected.
```

7.186 SUM() OVER

语句

```
SUM ( expr ) OVER < window name or specification >
```

关于< window name or specification >的详细内容参考**window clause**

说明

Window function SUM是计算expr值的总和的函数

不计算NULL值

使用示例

```
gSQL> SELECT min_price AS "MIN_PRICE"
             , SUM( min_price ) OVER ( ORDER BY min_price ) AS "SUM"
        FROM product_information
       WHERE supplier_id = 102050;


MIN_PRICE  SUM
---------- ----
       73   73
```

```
      247  320

      731 1051

     null 1051

     null 1051


5 rows selected.
```

7.187 SYSDATE

语句

```
SYSDATE
```

说明

以数据库服务器的操作系统时间为准获取当前DATE type值

示例

```
gSQL> SELECT SYSDATE FROM t1;


SYSDATE

----------

2013-12-12

2013-12-12

2013-12-12


3 rows selected.
```

7.188 SYS_EXTRACT_UTC

语句

```
SYS_EXTRACT_UTC( datetime_with_timezone )
```

说明

SYS_EXTRACT_UTC返回UTC（Coordinated Universal Time-formerly Greenwich Mean Time）值

未指定timezone时计算为session time zone

输入参数为timetime with time zonetimestamptimestamp with time zone类型

结果类型为time或timestamp

使用示例

```
gSQL> SELECT
      SYS_EXTRACT_UTC(
          TO_TIMESTAMP_TZ( '2017-05-25 00:00:00.000000 +09:00',
                           'YYYY-MM-DD HH24:MI:SS.FF6 TZH:TZM' )
          ) AS RESULT
      FROM DUAL;
RESULT
--------------------------
```

```
2017-05-24 15:00:00.000000

1 row selected.
```

7.189 SYSTIME

语句

```
SYSTIME
```

说明

以数据库服务器的操作系统时间为准获取有TIME ZONE的当前时间（TIME WITH TIME ZONE type）值

使用示例

```
gSQL> SELECT SYSTIME FROM t1;


SYSTIME

----------------------

16:30:46.954941 +09:00

16:30:46.954941 +09:00

16:30:46.954941 +09:00


3 rows selected.
```

7.190 SYSTIMESTAMP

语句

```
SYSTIMESTAMP
```

说明

以数据库服务器的操作系统时间为准获取有TIME ZONE的当前TIMESTAMP WITH TIME ZONE type值

使用示例

```
gSQL> SELECT SYSTIMESTAMP FROM t1;


SYSTIMESTAMP

--------------------------------

2013-12-12 16:37:34.432241 +09:00

2013-12-12 16:37:34.432241 +09:00

2013-12-12 16:37:34.432241 +09:00


3 rows selected.
```

7.191 TAN

语句

```
TAN( num )
```

说明

TAN函数以弧度为单位返回num的tangent值

num为NULL时返回NULL

使用示例

```
gSQL> SELECT TAN( 1 ) AS RESULT FROM DUAL;
         RESULT
---------------
1.5574077246549
1 row selected.
```

7.192 TO_BASE64

语句

```
TO_BASE64( str )
```

说明

TO_BASE64返回将str转换为base64 encoding的字符

str为CHARACTERCHARACTER VARYINGCHARACTER LONG VARYING等CHARACTER字符类型或可转换为字符类型的类型与BINARYBINARY VARYINGBINARY LONG VARYING等BINARY字符类型

结果类型为CHARACTER VARYING或CHARACTER LONG VARYING等CHARACTER字符类型

str为NULL时结果值也为NULL

Base64 encoding将8bit的binary数据表示为以ascii领域构成的64个字符

64个字符以A~Za~z0~9+/构成

以一个字符表示6bit以3个字符（24bit）为一个单位表示为4个字符

encoding的字符不足4个字符时用'='填充

encoding的字符超过76个时添加newline后分为多行

base64 encoding的字符的decoding使用FROM_BASE64()函数

decoding base64时忽略newlinecarriage returntabspace

详细内容参考**FROM_BASE64**

使用示例

```
gSQL> SELECT TO_BASE64( 'abc' ), TO_BASE64( 'abcd' ) FROM DUAL;


TO_BASE64( 'abc' ) TO_BASE64( 'abcd' )
------------------ ------------------
YWJj               YWJjZA==
1 row selected.
```

7.193 TO_CHAR(datetime)

语句

```
TO_CHAR( datetime [, fmt ] )
```

说明

TO_CHAR（datetime）函数将datetime转换为指定的fmt形式字符串后进行返回

datetime为DATE, TIMESTAMP, TIMESTAMP WITH TIME ZONE, TIME, TIME WITH TIME ZONE, INTERVAL类型

fmt为CHARACTERCHARACTER VARYING等CHARACTER字符类型

只要有一个输入参数为NULL则返回NULL

省略fmt时根据default format形式

- DATE : **NLS_DATE_FORMAT**
- TIMESTAMP : **NLS_TIMESTAMP_FORMAT**
- TIMESTAMP WITH TIME ZONE : **NLS_TIMESTAMP_WITH_TIME_ZONE_FORMAT**
- TIME : **NLS_TIME_FORMAT**
- TIME WITH TIME ZONE : **NLS_TIME_WITH_TIME_ZONE_FORMAT**

datetime为INTERVAL类型时与fmt无关转换为string后返回

fmt中可指定的字符串参考**日期时间格式字符串**

结果类型为CHARACTER VARYING

使用示例

以下为省略fmt, NLS_DATE_FORMAT = 'YYYY-MM-DD'时的示例

```
gSQL> SELECT
      TO_CHAR( TO_DATE( '2012-03-15','YYYY-MM-DD' ) ) AS RESULT
      FROM DUAL;
RESULT
----------
2012-03-15
1 row selected.
```

以下为指定fmt时的示例

```
gSQL> SELECT
      TO_CHAR( TO_DATE('2012-03-15','YYYY-MM-DD'), 'DD-MON-YY' ) AS RESULT
      FROM DUAL;
RESULT
---------
15-MAR-12
1 row selected.
```

7.194 TO_CHAR(number)

语句

```
TO_CHAR( number [, fmt ] )
```

说明

TO_CHAR（number）函数将number转换为指定的fmt形式字符串后返回

number为数字类型

fmt为CHARACTER, CHARACTER VARYING等CHARACTER字符类型

省略fmt时以字符串形式返回所有有效数字

fmt中可指定的字符串参考**数字格式字符串**

只要有一个输入参数为NULL则返回NULL

结果类型为CHARACTER VARYING

使用示例

```
gSQL> SELECT TO_CHAR( 12500000 ) AS RESULT FROM DUAL;
RESULT
--------
```

```
12500000

1 row selected.


gSQL> SELECT TO_CHAR( 12500000, 'S999,999,999' ) AS RESULT FROM DUAL;

RESULT

------------

 +12,500,000

1 row selected.
```

7.195 TO_DATE

语句

```
TO_DATE( str [, fmt ] )
```

说明

TO_DATE函数将指定fmt格式的字符串str转换为DATE类型后返回

str和fmt为CHARACTER, CHARACTER VARYING, CHARACTER LONG VARYING等CHARACTER字符类型

省略fmt时NLS_DATE_FORMAT为default format格式此时str应为default format格式的字符串

fmt中可指定的字符串参考**日期时间格式字符串**

详细内容参考 **NLS_DATE_FORMAT**

str或fmt为NULL时返回NULL

结果类型为DATE

使用示例

以下为省略fmtNLS_DATE_FORMAT = 'YYYY-MM-DD'时的示例

```
gSQL> SELECT TO_DATE( '2009-07-29' ) AS RESULT FROM DUAL;

RESULT

----------

2009-07-29

1 row selected.
```

以下为指定fmt时的示例

```
gSQL> SELECT TO_DATE( '29-JUL-09', 'DD-MON-YY' ) AS RESULT FROM DUAL;

RESULT

----------

2009-07-29

1 row selected.
```

7.196 TO_NATIVE_BIGINT

语句

```
TO_NATIVE_BIGINT( str [, fmt ] )
```

说明

TO_NATIVE_BIGINT函数将指定fmt格式的字符串str转换为NATIVE_BIGINT类型后返回

str与fmt为CHARACTERCHARACTER VARYINGCHARACTER LONG VARYING等CHARACTER 字符类型

str与fmt中只要有一个为NULL则结果值也为NULL

fmt中可指定的字符串参考**数字格式字符串**

结果类型为NATIVE_BIGINT

使用示例

```
gSQL> SELECT TO_NATIVE_BIGINT( '123.45' ) AS RESULT1,
             TO_NATIVE_BIGINT( '+123.45', 'S999.99' ) AS RESULT2
        FROM DUAL;
```

```
RESULT1 RESULT2

------- -------

    123     123

1 row selected.
```

7.197 TO_NATIVE_DOUBLE

语句

```
TO_NATIVE_DOUBLE( str [, fmt ] )
```

说明

TO_NATIVE_DOUBLE函数将指定fmt格式的字符串str转换为NATIVE_DOUBLE类型后返回

str与fmt为CHARACTERCHARACTER VARYINGCHARACTER LONG VARYING等CHARACTER字符类型

str与fmt中只要有一个为NULL则结果值也为NULL

fmt中可指定的字符串参考**数字格式字符串**

结果类型为NATIVE_DOUBLE

使用示例

```
gSQL> SELECT TO_NATIVE_DOUBLE( '123.45' ) AS RESULT1,
             TO_NATIVE_DOUBLE( '+123.45', 'S999.99' ) AS RESULT2
        FROM DUAL;
```

```
RESULT1 RESULT2

------- -------

 123.45  123.45

1 row selected.
```

7.198 TO_NATIVE_INTEGER

语句

```
TO_NATIVE_INTEGER( str [, fmt ] )
```

说明

TO_NATIVE_INTEGER函数将指定fmt格式的字符串str转换为NATIVE_INTEGER类型后返回

str与fmt为CHARACTERCHARACTER VARYINGCHARACTER LONG VARYING等CHARACTER字符类型

str与fmt中只要有一个为NULL则结果值也为NULL

fmt中可指定的字符串参考**数字格式字符串**

结果类型为NATIVE_INTEGER

使用示例

```
gSQL> SELECT TO_NATIVE_INTEGER( '123.45' ) AS RESULT1,
             TO_NATIVE_INTEGER( '+123.45', 'S999.99' ) AS RESULT2
        FROM DUAL;
```

```
RESULT1 RESULT2

------- -------

    123     123

1 row selected.
```

7.199 TO_NATIVE_REAL

语句

```
TO_NATIVE_REAL( str [, fmt ] )
```

说明

TO_NATIVE_REAL函数将指定fmt格式的字符串str转换为NATIVE_REAL类型后返回

str与fmt为CHARACTERCHARACTER VARYINGCHARACTER LONG VARYING等CHARACTER字符类型

str与fmt中只要有一个为NULL则结果值也为NULL

fmt中可指定的字符串参考**数字格式字符串**

结果类型为NATIVE_REAL

使用示例

```
gSQL> SELECT TO_NATIVE_REAL( '123.45' ) AS RESULT1,
             TO_NATIVE_REAL( '+123.45', 'S999.99' ) AS RESULT2
        FROM DUAL;
```

```
RESULT1 RESULT2

------- -------

 123.45  123.45
```

7.200 TO_NATIVE_SMALLINT

语句

```
TO_NATIVE_SMALLINT( str [, fmt ] )
```

说明

TO_NATIVE_SMALLINT函数将指定fmt格式的字符串str转换为NATIVE_SMALLINT类型后返回

str与fmt为CHARACTERCHARACTER VARYINGCHARACTER LONG VARYING等CHARACTER字符类型

str与fmt中只要有一个为NULL则结果值也为NULL

fmt中可指定的字符串参考**数字格式字符串**

结果类型为NATIVE_SMALLINT

使用示例

```
gSQL> SELECT TO_NATIVE_SMALLINT( '123.45' ) AS RESULT1,
             TO_NATIVE_SMALLINT( '+123.45', 'S999.99' ) AS RESULT2
        FROM DUAL;
```

```
RESULT1 RESULT2

------- -------

    123     123

1 row selected.
```

7.201 TO_NUMBER

语句

```
TO_NUMBER( str [, fmt] )
```

说明

TO_NUMBER函数将指定fmt格式的字符串str转换为NUMBER类型后返回

str与fmt为CHARACTERCHARACTER VARYINGCHARACTER LONG VARYING等CHARACTER字符类型

str与fmt中只要有一个为NULL则结果值也为NULL

fmt中可指定的字符串参考**数字格式字符串**

结果类型为NUMBER

使用示例

```
gSQL> SELECT TO_NUMBER( '123.45' ) AS RESULT1,
             TO_NUMBER( '+123.45', 'S999.99' ) AS RESULT2
        FROM DUAL;
```

```
RESULT1 RESULT2

------- -------

 123.45  123.45

1 row selected.
```

7.202 TO_TIME

语句

```
TO_TIME( str [, fmt ] )
```

说明

TO_TIME函数将指定fmt格式的字符串str转换为TIME类型后返回

str与fmt为CHARACTERCHARACTER VARYINGCHARACTER LONG VARYING等CHARACTER字符类型

省略fmt时NLS_TIME_FORMAT为默认格式这时str也应为默认格式的字符串

fmt中可指定的字符串参考**日期时间格式字符串**

详细内容参考 **NLS_TIME_FORMAT**

str或fmt为NULL时返回NULL

结果类型为TIME

使用示例

以下为省略fmt, NLS_TIME_FORMAT = 'HH24:MI:SS.FF6'时的示例

```
gSQL> SELECT TO_TIME( '11:22:33.999999' ) AS RESULT FROM DUAL;
RESULT
---------------
11:22:33.999999
1 row selected.
```

以下为指定fmt时的示例

```
gSQL> SELECT
      TO_TIME( '112233.999999/P.M.', 'HH12MISS.FF6/P.M.' ) AS RESULT
      FROM DUAL;
RESULT
---------------
23:22:33.999999
1 row selected.
```

7.203 TO_TIME_TZ

语句

```
TO_TIME_TZ( str [, fmt ] )
```

说明

是TO_TIME_WITH_TIME_ZONE的aliase

详细内容参考**TO_TIME_WITH_TIME_ZONE**, **NLS_TIME_WITH_TIME_ZONE_FORMAT**

使用示例

以下为省略fmtNLS_TIME_WITH_TIME_ZONE_FORMAT = 'HH24:MI:SS.FF6 TZH:TZM'时的示例

```
gSQL> SELECT TO_TIME_TZ( '11:22:33.999999 +09:00' ) AS RESULT FROM DUAL;

RESULT

----------------------

11:22:33.999999 +09:00

1 row selected.
```

以下为指定fmt时的示例

```
gSQL> SELECT TO_TIME_TZ( '11:22:33.999999 +09:00 PM',
                         'HH12:MI:SS.FF6 TZH:TZM PM' ) AS RESULT
      FROM DUAL;
RESULT
----------------------
23:22:33.999999 +09:00
1 row selected.
```

7.204 TO_TIME_WITH_TIME_ZONE

语句

```
TO_TIME_WITH_TIME_ZONE( str [, fmt ] )

TO_TIME_TZ( str [, fmt ] )
```

说明

TO_TIME_WITH_TIME_ZONE函数将指定fmt格式的字符串str转换为TIME WITH TIME ZONE类型后返回

str与fmt为CHARACTER, CHARACTER VARYING, CHARACTER LONG VARYING等CHARACTER字符类型

省略fmt时NLS_TIME_WITH_TIME_ZONE_FORMAT为默认格式这时str也应为默认格式的字符串

fmt中可指定的字符串参考**日期时间格式字符串**

详细内容参考**NLS_TIME_WITH_TIME_ZONE_FORMAT**

str或fmt为NULL时返回NULL

TO_TIME_WITH_TIME_ZONE的aliase为**TO_TIME_TZ**

结果类型为TIME WITH TIME ZONE

使用示例

以下为省略 fmtNLS_TIME_WITH_TIME_ZONE_FORMAT = 'HH24:MI:SS.FF6 TZH:TZM'时的示例

```
gSQL> SELECT
      TO_TIME_WITH_TIME_ZONE( '11:22:33.999999 +09:00' ) AS RESULT
      FROM DUAL;
RESULT
----------------------
11:22:33.999999 +09:00
1 row selected.
```

以下为指定fmt时的示例

```
gSQL> SELECT
      TO_TIME_WITH_TIME_ZONE( '11:22:33.999999 +09:00 PM',
                              'HH12:MI:SS.FF6 TZH:TZM PM' )
      AS RESULT
      FROM DUAL;
RESULT
----------------------
23:22:33.999999 +09:00
1 row selected.
```

7.205 TO_TIMESTAMP

语句

```
TO_TIMESTAMP( str [, fmt ] )
```

说明

TO_TIMESTAMP函数将指定fmt格式的字符串str转换为TIMESTAMP 类型后返回

str与fmt为CHARACTERCHARACTER VARYINGCHARACTER LONG VARYING等CHARACTER字符类型

省略fmt时NLS_TIMESTAMP_FORMAT为默认格式这时str也应为默认格式的字符串

fmt中可指定的字符串参考**日期时间格式字符串**

详细内容参考**NLS_TIMESTAMP_FORMAT**

str或fmt为NULL时返回NULL

结果类型为TIMESTAMP

使用示例

以下为省略fmt, NLS_TIMESTAMP_FORMAT = 'YYYY-MM-DD HH24:MI:SS.FF6'时的示例

```
gSQL> SELECT
      TO_TIMESTAMP( '2009-07-29 11:22:33.999999' ) AS RESULT
      FROM DUAL;
RESULT
--------------------------
2009-07-29 11:22:33.999999
1 row selected.
```

以下为指定fmt时的示例

```
gSQL> SELECT
      TO_TIMESTAMP( '090729 112233999999 PM', 'YYMMDD HH12MISSFF6 PM' )
      AS RESULT
      FROM DUAL;


RESULT
--------------------------
2009-07-29 23:22:33.999999
1 row selected.
```

7.206 TO_TIMESTAMP_TZ

语句

```
TO_TIMESTAMP_TZ( str [, fmt ] )
```

说明

是TO_TIMESTAMP_WITH_TIME_ZONE的aliase

详细内容参考**TO_TIMESTAMP_WITH_TIME_ZONE**,

NLS_TIMESTAMP_WITH_TIME_ZONE_FORMAT

使用示例

以下为省略fmtNLS_TIMESTAMP_WITH_TIME_ZONE_FORMAT = 'YYYY-MM-DD HH24:MI:SS.FF6 TZH:TZM'时的示例

```
gSQL> SELECT
      TO_TIMESTAMP_TZ( '2009-07-29 11:22:33.999999 +09:00' ) AS RESULT
      FROM DUAL;
RESULT
--------------------------------
2009-07-29 11:22:33.999999 +09:00
```

```
1 row selected.
```

以下为指定fmt时的示例

```
gSQL> SELECT
      TO_TIMESTAMP_TZ( '29-JUL-09 11:22:33.999999 +09:00',
                       'DD-MON-RR HH12:MI:SS.FF6 TZH:TZM' ) AS RESULT
      FROM DUAL;
RESULT
---------------------------------
2009-07-29 11:22:33.999999 +09:00
1 row selected.
```

7.207 TO_TIMESTAMP_WITH_TIME_ZONE

语句

```
TO_TIMESTAMP_WITH_TIME_ZONE( str [, fmt ] )

TO_TIMESTAMP_TZ( str [, fmt ] )
```

说明

TO_TIMESTAMP_WITH_TIME_ZONE函数将指定fmt格式的字符串str转换为TIMESTAMP WITH TIME ZONE类型后返回

str与fmt为CHARACTERCHARACTER VARYINGCHARACTER LONG VARYING等CHARACTER字符类型

省略fmt时NLS_TIMESTAMP_WITH_TIME_ZONE_FORMAT为默认格式这时str也应为默认格式的字符串

fmt中可指定的字符串参考**日期时间格式字符串**

详细内容参考**NLS_TIMESTAMP_WITH_TIME_ZONE_FORMAT**

str或fmt为NULL时返回NULL

TO_TIMESTAMP_WITH_TIME_ZONE的alias为**TO_TIMESTAMP_TZ**函数

结果类型为TIMESTAMP WITH TIME ZONE

使用示例

以下为省略fmtNLS_TIMESTAMP_WITH_TIME_ZONE_FORMAT = 'YYYY-MM-DD HH24:MI:SS.FF6 TZH:TZM'时的示例

```
gSQL> SELECT
      TO_TIMESTAMP_WITH_TIME_ZONE( '2009-07-29 11:22:33.999999 +09:00' )
      AS RESULT
      FROM DUAL;
RESULT
---------------------------------
2009-07-29 11:22:33.999999 +09:00
1 row selected.
```

以下为指定fmt时的示例

```
gSQL> SELECT
      TO_TIMESTAMP_WITH_TIME_ZONE( '29-JUL-09 11:22:33.999999 +09:00',
                                   'DD-MON-RR HH12:MI:SS.FF6 TZH:TZM' )
      AS RESULT
      FROM DUAL;
RESULT
---------------------------------
2009-07-29 11:22:33.999999 +09:00
1 row selected.
```

7.208 TRANSACTION_DATE

语句

```
TRANSACTION_DATE()
```

说明

以会话时间为准获取当前日期(DATE type)值

获取当前日期的函数之间有如下差异

- TRANSACTION_DATE()：事务内的所有日期值均相同
- STATEMENT_DATE()：一个SQL语句内的所有日期值均相同
- CLOCK_DATE()：每次调用函数时均获得当前日期值

使用示例

相同事务中的日期值始终相同

```
gSQL> SELECT TRANSACTION_DATE() FROM dual;


TRANSACTION_DATE()
```

```
------------------

2013-12-12


1 row selected.


gSQL> SELECT TRANSACTION_DATE() FROM dual;


TRANSACTION_DATE()

------------------

2013-12-12


1 row selected.


gSQL> COMMIT;


Commit complete.


gSQL> SELECT TRANSACTION_DATE() FROM dual;


TRANSACTION_DATE()

------------------

2013-12-13


1 row selected.
```

7.209 TRANSACTION_LOCALTIME

语句

```
TRANSACTION_LOCALTIME()
```

说明

以会话时间为准获取没有TIME ZONE的当前时间(TIME WITHOUT TIME ZONE type)值

获取当前时间的函数之间有如下差异

- TRANSACTION_LOCALTIME()：事务内的所有时间值均相同
- STATEMENT_LOCALTIME()：一个SQL语句内的所有时间值均相同
- CLOCK_LOCALTIME()：每次调用函数时获得当前时间值

使用示例

相同事务内的时间值均相同

```
gSQL> SELECT TRANSACTION_LOCALTIME() FROM dual;


TRANSACTION_LOCALTIME()
```

```
-----------------------

16:43:24.391834


1 row selected.


gSQL> SELECT TRANSACTION_LOCALTIME() FROM dual;


TRANSACTION_LOCALTIME()

-----------------------

16:43:24.391834


1 row selected.


gSQL> COMMIT;


Commit complete.


gSQL> SELECT TRANSACTION_LOCALTIME() FROM dual;


TRANSACTION_LOCALTIME()

-----------------------

16:43:32.651833


1 row selected.
```

7.210 TRANSACTION_LOCALTIMESTAMP

语句

```
TRANSACTION_LOCALTIMESTAMP()
```

说明

以会话时间为准获取没有TIME ZONE的当前TIMESTAMP(TIMESTAMP WITHOUT TIME ZONE type)值

获取当前TIMESTAMP的函数之间有如下差异

- TRANSACTION_LOCALTIMESTAMP() ：事务内的所有TIMESTAMP值均相同
- STATEMENT_LOCALTIMESTAMP()：一个SQL语句内的所有TIMESTAM值均相同
- CLOCK_LOCALTIMESTAMP()：每次调用函数时获得当前TIMESTAMP值

使用示例

相同事务内的TIMESTAMP值均相同

```
gSQL> SELECT TRANSACTION_LOCALTIMESTAMP() FROM dual;
```

```
TRANSACTION_LOCALTIMESTAMP()

----------------------------

2013-12-12 16:43:32.651833


1 row selected.


gSQL> SELECT TRANSACTION_LOCALTIMESTAMP() FROM dual;


TRANSACTION_LOCALTIMESTAMP()

----------------------------

2013-12-12 16:43:32.651833


1 row selected.


gSQL> COMMIT;


Commit complete.


gSQL> SELECT TRANSACTION_LOCALTIMESTAMP() FROM dual;


TRANSACTION_LOCALTIMESTAMP()

----------------------------

2013-12-12 16:46:07.831834


1 row selected.
```

7.211 TRANSACTION_TIME

语句

```
TRANSACTION_TIME()
```

说明

以会话时间为准获取有TIME ZONE的当前时间(TIME WITH TIME ZONE type)值

获取当前时间的函数之间有如下差异

- TRANSACTION_TIME() ：事务内的所有时间值均相同
- STATEMENT_TIME()：一个SQL语句内的所有时间值均相同
- CLOCK_TIME()：每次调用函数时获得当前时间值

使用示例

相同事务内的时间值始终相同

```
gSQL> SELECT TRANSACTION_TIME() FROM dual;


TRANSACTION_TIME()
```

```
----------------------

16:46:07.831834 +09:00


1 row selected.


gSQL> SELECT TRANSACTION_TIME() FROM dual;


TRANSACTION_TIME()

----------------------

16:46:07.831834 +09:00


1 row selected.


gSQL> COMMIT;


Commit complete.


gSQL> SELECT TRANSACTION_TIME() FROM dual;


TRANSACTION_TIME()

----------------------

16:48:00.691827 +09:00


1 row selected.
```

7.212 TRANSACTION_TIMESTAMP

语句

```
TRANSACTION_TIMESTAMP()
```

说明

以会话时间为准获取有TIME ZONE的当前TIMESTAMP(TIMESTAMP WITH TIME ZONE type)值

获取当前TIMESTAMP的函数之间有如下差异

- TRANSACTION_TIMESTAMP()：事务内的所有TIMESTAMP值均相同
- STATEMENT_TIMESTAMP()：一个SQL语句内的所有TIMESTAMP值均相同
- CLOCK_TIMESTAMP() ：每次调用函数时获得当前TIMESTAMP值

使用示例

相同事务内的TIMESTAMP值始终相同

```
gSQL> SELECT TRANSACTION_TIMESTAMP() FROM dual;


TRANSACTION_TIMESTAMP()
```

```
---------------------------------

2013-12-12 16:48:00.691827 +09:00


1 row selected.


gSQL> SELECT TRANSACTION_TIMESTAMP() FROM dual;


TRANSACTION_TIMESTAMP()

---------------------------------

2013-12-12 16:48:00.691827 +09:00


1 row selected.


gSQL> COMMIT;


Commit complete.


gSQL>  SELECT TRANSACTION_TIMESTAMP() FROM dual;


TRANSACTION_TIMESTAMP()

---------------------------------

2013-12-12 16:49:26.291827 +09:00


1 row selected.
```

7.213 TRANSLATE

语句

```
TRANSLATE( string, from, to )
```

说明

TRANSLATE是替换字符的函数将与from的字符一致的string的字符替换为与from的字符位于相同位置的to的字符后返回

stringfromto可以是CHARACTER, CHARACTER VARYING, CHARACTER LONG VARYING等CHARACTER字符类型

strfromto值中只要有一个为NULL则结果值为NULL

- 有与from的字符一致的string的字符时
  - from的长度与to的长度相同时替换为与from的字符位于相同位置的to的字符
  - from的长度大于to的长度时从string删除to的字符长度外的from的字符
  - from的字符由重复的字符构成时from的重复字符的第一个位置替换为相同位置的to的字符
- 没有与from的字符一致的string字符时不替换string

结果类型如下表

string类型	结果类型
CHAR或VARCHAR	VARCHAR
LONG VARCHAR	LONG VARCHAR

使用示例

- 有与from的字符一致的string的字符时替换为相同位置的to的字符
  - A → Z, C → Y, E → X, G → W

```
gSQL> SELECT TRANSLATE('ABCDEFG', 'ACEG', 'ZYXW') AS RESULT
FROM DUAL;
RESULT
-------
ZBYDXFW
1 row selected.
```

- from的长度大于to的长度时从string删除to的字符长度外的from的字符后替换
  - A → Z, C → Y, 删除E, 删除G

```
gSQL> SELECT TRANSLATE('ABCDEFG', 'ACEG', 'ZY') AS RESULT
      FROM DUAL;
RESULT
------
ZBYDF
```

```
1 row selected.
```

7.214 TRIM

语句

```
TRIM([ [ LEADING | TRAILING | BOTH ]  [trim_character] FROM ] trim_source)
```

说明

TRIM函数在trim_source中从LEADINGTRALINGBOTH方向对比time_character并删除一致的字符后返回

trim_character与trim_source为CHARACTERCHARACTER VARYINGCHARACTER LONG VARYING等CHARACTER 字符类型与BINARYBINARY VARYINGBINARY LONG VARYING等BINARY字符类型

trim_character, trim_source中只要有一个为NULL则结果值为NULL

- [LEADING | TRAILING | BOTH]
  - LEADING : 从trim_source的前面部分开始删除trim_character
  - TRAILING : 从trim_source的后面部分开始删除trim_character
  - BOTH : 从trim_source的前后两个方向开始删除trim_character

- trim_character只能为一个字符
- 省略trim_character时默认指定为single blank space(' ')
- 指定FROM时
  - 应指定[LEADING | TRAILING | BOTH]或trim_character或[LEADING | TRAILING | BOTH] trim_character
    - 示例： TRIM(LEADING FROM ' abc') , TRIM('x' FROM 'xabc') , TRIM(LEADING 'x' FROM 'xabc')
  - 省略[LEADING | TRAILING | BOTH]时默认指定为BOTH
- 省略FROM时
  - 为TRIM(trim_source)与TRIM(BOTH ' ' FROM trim_source)执行方法相同

结果类型如下表

trim_character, trim_source类型	结果类型
CHAR或VARCHAR	VARCHAR
LONG VARCHAR	LONG VARCHAR
BINARY或VARBINARY	VARBINARY
LONG VARBINARY	LONG VARBINARY

使用示例

```
gSQL> SELECT TRIM( LEADING '_' FROM '___TRIM FUNCTION___' ) AS RESULT
      FROM DUAL;
```

```
RESULT

----------------

TRIM FUNCTION___

1 row selected.


gSQL> SELECT TRIM( TRAILING '_' FROM '___TRIM FUNCTION___' ) AS RESULT

      FROM DUAL;

RESULT

----------------

___TRIM FUNCTION

1 row selected.


gSQL> SELECT TRIM( BOTH '_' FROM '___TRIM FUNCTION___' ) AS RESULT

      FROM DUAL;

RESULT

-------------

TRIM FUNCTION

1 row selected.
```

7.215 TRUNC(number)

语句

```
TRUNC( num [ , scale ] )
```

说明

TRUNC（number）函数返回以scale为准去掉num的值

num与scale为数字类型

num或scale为NULL时返回NULL

省略scale时scale默认为0与TRUNC(num, 0)时的执行方法相同

scale为正数时以小数点右侧位数为准去掉

scale为负数时以小数点左侧位数为准去掉

使用示例

```
gSQL> SELECT TRUNC( 142.4282, 2 ) AS RESULT FROM DUAL;
RESULT
------
142.42
1 row selected.
```

```
gSQL> SELECT TRUNC( 142.4282, -2 ) AS RESULT FROM DUAL;

RESULT

------

   100

1 row selected.
```

7.216 TRUNC(date)

语句

```
TRUNC( date [ , fmt ] )
```

说明

TRUNC（date）函数返回以指定fmt为单位去掉date的值

date为DATE, TIMESTAMP, TIMESTAMP WITH TIME ZONE类型

fmt为CHARACTER, CHARACTER VARYING等CHARACTER字符类型

date或fmt为NULL时返回NULL

结果类型与输入参数date类型无关始终为DATE类型

省略fmt时默认值为'DAY'可使用的格式字符串如下表

格式字符串	说明
CC, SCC	世纪
YYYY, YEAR, SYYYY, SYEAR, YYY, YY, Y	年
IYYY, IYY, IY, I	使用ISO 8601标准定义的Calendar week的年份
Q	季度

MONTH, MON, MM, RM	月
WW	年份的1月1日为一周的开始的周
IW	ISO 8601标准定义的Calendar week（1-52周或1-53周）指定年份的第一个周四所属的周为第一周
W	月份的1日为一周的开始的周
DDD, DD, J	日
DAY, DY, D	星期
HH, HH12, HH24	时
MI	分

Table 7-26 fmt中可使用的格式字符串

使用示例

```
gSQL> SELECT
      TRUNC( TO_DATE( '2051-07-16', 'YYYY-MM-DD' ), 'CC' ) AS RESULT
      FROM DUAL;
RESULT
----------
2001-01-01
1 row selected.
```

```
gSQL> SELECT
      TRUNC( TO_DATE( '2051-07-16', 'YYYY-MM-DD' ), 'YYYY' ) AS RESULT
      FROM DUAL;
RESULT
----------
2051-01-01
1 row selected.


gSQL> SELECT
      TRUNC( TO_DATE( '2051-07-16', 'YYYY-MM-DD' ), 'MONTH' ) AS RESULT
      FROM DUAL;
RESULT
----------
2051-07-01
1 row selected.


gSQL> SELECT
      TRUNC( TO_TIMESTAMP( '2001-05-05 11:22:33.999999',
             'YYYY-MM-DD HH24:MI:SS.FF6' ) ) AS RESULT
      FROM DUAL;
RESULT
----------
2001-05-05
1 row selected.
```

7.217 UPPER

语句

```
UPPER( str )
```

说明

UPPER函数返回str的大写

str为CHARACTER, CHARACTER VARYING, CHARACTER LONG VARYING等字符类型

str为NULL时结果值也为NULL

结果类型与str的类型相同

使用示例

```
gSQL> SELECT UPPER( 'spring' ) AS RESULT FROM DUAL;
RESULT
------
SPRING
1 row selected.
```

7.218 UNHEX

语句

```
UNHEX( str )
```

说明

str为16进制字符均用byte表示并返回为binary string

输入参数为CHARACTER VARYINGCHARACTER LONG VARYING等CHARACTER字符类型结果类型为BINARY VARYING或BINARY LONG VARYING等BINARY字符类型

str为NULL时结果值也为NULL

str中包含不属于16进制范围的字符时报错

详细内容参考**HEX**

使用示例

```
gSQL> SELECT UNHEX( HEX( 'abc' ) ) FROM DUAL;

UNHEX( HEX( 'abc' ) )

---------------------

616263
```

```
1 row selected.
```

7.219 UNHEX_TO_CHARSTR

语句

```
UNHEX_TO_CHARSTR( str )
```

说明

str为16进制字符均用byte表示并返回为character string

输入参数为CHARACTER VARYINGCHARACTER LONG VARYING等CHARACTER字符类型结果类型为CHARACTER VARYING或CHARACTER LONG VARYING等CHARACTER字符类型

str为NULL时结果值也为NULL

str中包含不属于16进制范围的字符时报错

详细内容参考**HEX**,**UNHEX**

使用示例

```
gSQL> SELECT UNHEX_TO_CHARSTR( '616263' ) FROM DUAL;

UNHEX_TO_CHARSTR( '616263' )

----------------------------

abc

1 row selected.


gSQL> SELECT UNHEX_TO_CHARSTR( HEX( 'abc' ) ) FROM DUAL;

UNHEX_TO_CHARSTR( HEX( 'abc' ) )

--------------------------------

abc

1 row selected.
```

7.220 USER_ID

语句

```
USER_ID ()
```

说明

获取当前用户的Number ID

> Note:
>
> 根据集群系统访问的服务器会拥有不同的值
>
> 推荐使用返回当前用户名称的**CURRENT_USER**函数

使用示例

```
% gsql test test


gSQL> SELECT USER_ID() FROM dual;


USER_ID()
---------
```

```
        6


1 row selected.
```

7.221 UUID

语句

```
UUID ()
```

说明

UUID函数生成并返回通用唯一标识符（Universal Unique Identifier）

返回类型为VARBINARY类型内部由16byte构成

使用示例

```
gSQL> SELECT HEX( UUID() ) FROM DUAL;

HEX( UUID() )

--------------------------------

E6F0A5C2387511E8B95259E479C2FD50

1 row selected.
```

7.222 VAR_POP

语句

```
VAR_POP( expr )
```

说明

为聚合操作函数获取expr set的总体方差（population variance）

除NULL值外的expr set的数量为1时返回0

参数与结果类型如下表

expr	结果类型
NATIVE_INTEGER系列 • NATIVE_SMALLINT • NATIVE_INTEGER • NATIVE_BIGINT	NATIVE_DOUBLE
NUMBER	NUMBER

expr	结果类型
NATIVE_DOUBLE系列 • NATIVE_REAL • NATIVE_DOUBLE	NATIVE_DOUBLE

Table 7-27　VAR_POP参数与结果类型

> Note:
>
> 总体方差指总体（全部）的方差方差是偏差平方的平均值即从数据中的每个值中减去总均值（总体平均值）加上所有平方然后除以总体中的数据数
>
> 其用于确定每个观察值的平均方差程度

详细内容参考**STDDEV_POP**

使用示例

```
gSQL> SELECT VAR_POP(c1) FROM t1;


VAR_POP(C1)

-----------

     105.76
```

```
1 row selected.
```

7.223 VAR_POP() OVER

语句

```
VAR_POP ( expr ) OVER < window name or specification >
```

关于< window name or specification >的详细内容请参阅**window clause**

说明

Window function VAR_POP是计算expr的总体方差 (population variance)的函数

如果去除NULL值的expr数量为1则返回结果为0

使用示例

```
gSQL> SELECT product_id, min_price
            , VAR_POP( min_price ) OVER ( ORDER BY product_id ) AS
"VAR_POP"
```

```
        FROM product_information
       WHERE supplier_id = 102050;


PRODUCT_ID MIN_PRICE          VAR_POP
---------- --------- ----------------
      1769      null             null
      1770        73                0
      2378       247             7569
      2382       731 77499.5555555556
      3355      null 77499.5555555556


5 rows selected.
```

7.224 VAR_SAMP

语句

```
VAR_SAMP( expr )
```

说明

为聚合操作函数获取expr set的样本方差（sample variance）

除NULL值外的expr set的数量为1时返回NULL值

参数与结果类型如下表

expr	结果类型
NATIVE_INTEGER系列 • NATIVE_SMALLINT • NATIVE_INTEGER • NATIVE_BIGINT	NATIVE_DOUBLE
NUMBER	NUMBER

expr	结果类型
NATIVE_DOUBLE系列 • NATIVE_REAL • NATIVE_DOUBLE	NATIVE_DOUBLE

Table 7-28　VAR_SAMP参数与结果类型

> Note:
>
> 样本方差与总体方差不同用抽取的样本计算平均与偏差即从数据的每个值中减掉样本平均后加各个的平方值并除以总样本的数据数-1
>
> 其用于预测总体的方差程度

详细内容参考**STDDEV_SAMP**

使用示例

```
gSQL> SELECT VAR_SAMP(c1) FROM t1;


VAR_SAMP(C1)

------------

       132.2
```

```
1 row selected.
```

7.225 VAR_SAMP() OVER

语句

```
VAR_SAMP ( expr ) OVER < window name or specification >
```

关于< window name or specification >的详细内容请参阅**window clause**

说明

Window function VAR_SAMP为计算expr的样本方差 (sample variance)的函数

如果去除NULL值的expr的数量为1则返回结果为NULL

使用示例

```
gSQL> SELECT product_id, min_price
            , VAR_SAMP( min_price ) OVER ( ORDER BY product_id ) AS
"VAR_SAMP"
```

```
        FROM product_information
       WHERE supplier_id = 102050;


PRODUCT_ID MIN_PRICE         VAR_SAMP
---------- --------- ----------------
      1769      null             null
      1770        73             null
      2378       247            15138
      2382       731 116249.333333333
      3355      null 116249.333333333


5 rows selected.
```

7.226 VARIANCE

语句

```
VARIANCE( [ ALL | DISTINCT ] expr )
```

说明

为聚合操作函数获取expr set的方差（variance）

指定ALL时对所有值执行指定DISTINCT时对去掉重复的值执行未明确指定ALL或DISTINCT时与指定ALL的处理方式相同

除NULL值外用DISTINCT去掉重复后的expr set的数量为1时返回0

参数与结果类型如下表

expr	结果类型
NATIVE_INTEGER系列 • NATIVE_SMALLINT • NATIVE_INTEGER • NATIVE_BIGINT	NATIVE_DOUBLE

expr	结果类型
NUMBER	NUMBER
NATIVE_DOUBLE系列 • NATIVE_REAL • NATIVE_DOUBLE	NATIVE_DOUBLE

Table 7-29　VARIANCE与结果类型

> Note:
>
> SUNDB方差的计算方式如下
>
> * expr set的数量为1时返回0
>
> * expr set的数量大于1时返回**STDDEV_SAMP(expr)**值

详细参考**STDDEV**

使用示例

```
gSQL> SELECT VARIANCE(c1) FROM t1;


VARIANCE(C1)
------------
       132.2
```

```
1 row selected.



gSQL> SELECT VARIANCE(ALL c1) FROM t1;

VARIANCE(ALL C1)
----------------
           132.2

1 row selected.



gSQL> SELECT VARIANCE(DISTINCT c1) FROM t1;

VARIANCE(DISTINCT C1)
---------------------
               176.25

1 row selected.
```

7.227 VARIANCE() OVER

语句

```
VARIANCE ( expr ) OVER < window name or specification >
```

关于< window name or specification >的详细内容请参阅**window clause**

说明

Window function VARIANCE为计算expr的方差 (variance)的函数

如果去除NULL值的expr的数量为1则返回结果为0

使用示例

```
gSQL> SELECT product_id, min_price
             , VARIANCE( min_price ) OVER ( ORDER BY product_id ) AS
"VARIANCE"
        FROM product_information
       WHERE supplier_id = 102050;


PRODUCT_ID MIN_PRICE         VARIANCE
---------- --------- ----------------
```

```
      1769      null              null

      1770        73                 0

      2378       247             15138

      2382       731 116249.333333333

      3355      null 116249.333333333


5 rows selected.
```

7.228 VERSION

语句

```
VERSION()
```

说明

获取产品的版本字符串（version string）

使用示例

```
gSQL> SELECT VERSION() FROM dual;


VERSION()

-------------------------------------

Release Name.X.X.X revision(XXXXX)


1 row selected.
```

7.229 WIDTH_BUCKET

语句

```
WIDTH_BUCKET( num, min, max, cnt )
```

说明

WIDTH_BUCKET函数在指定的minmax范围内生成与cnt的宽度相同的区间返回num所属区间的位置

num, min, max, cnt为数字类型

minmax为区间范围minmax值相同时报错

cnt为区间数量应为正整数为0或负数时报错

区间位置从1开始赋予编号

numminmaxcnt中只要有一个为NULL则结果值也为NULL

使用示例

```
gSQL> SELECT WIDTH_BUCKET( 5, 1, 20, 5 ) AS RESULT FROM DUAL;
RESULT
```

```
------

     2

1 row selected.
```

8.SQL References (A~B)

8.1 ALTER AUDIT POLICY

功能

在审计策略（audit policy）对象中添加或删除审计对象

语句

```
<alter audit policy statement> ::=
    ALTER AUDIT POLICY policy_name
    { <add_audit_option> | <drop_audit_option> }
    ;

<add_audit_option> ::=
    ADD { <privilege_audit_clause> |  <action_audit_clause> |
<privilege_audit_clause> <action_audit_clause> }

<drop_audit_option> ::=
    DROP { <privilege_audit_clause> |  <action_audit_clause> |
<privilege_audit_clause> <action_audit_clause> }
```

```
<privilege_audit_clause> ::=
    PRIVILEGES <database_privilege> [, ...]

<action_audit_clause> ::=
    ACTIONS { <object_action_audit> | <system_action_audit> } [, ...]

<object_action_audit> ::=
      ALL ON [schema_name.]object_name
    | <object_action> ON [schema_name.]object_name

<system_action_audit> ::=
      ALL
    | <system_action>
```

使用范围及访问权限

为了执行<alter audit policy statement>语句用户需要有AUDIT SYSTEM ON DATABASE权限

语句规则及参数

policy_name

要变更的audit policy对象的名称

<add_audit_option>

在审计策略中添加审计对象

<drop_audit_option>

在审计策略中删除审计对象

<privilege_audit_clause>

详细内容参考**CREATE AUDIT POLICY**

<action_audit_clause>

详细内容参考**CREATE AUDIT POLICY**

说明

可变更已激活的审计策略但不影响现有会话仅影响新生成的会话

> Note:
>
> 如下删除（DROP）ALL选项时并非删除所有的action仅删除该ALL选项

```
CREATE AUDIT POLICY p1
       ACTIONS ALL ON u1.t1,
               SELECT ON u1.t1;


ALTER AUDIT POLICY p1 DROP
      ACTIONS ALL ON u1.t1;
```

使用示例

以下为在审计策略中添加新audit option的示例

```
ALTER AUDIT POLICY policy_dml
      ADD ACTIONS SELECT ON u1.t1;
```

以下为在审计策略中删除audit option的示例

```
ALTER AUDIT POLICY policy_dml
      DROP ACTIONS SELECT ON u1.t1;
```

兼容性

标准SQL中无审计策略

参考

相关内容参考下文

- 管理审计策略对象
  - **CREATE AUDIT POLICY**
  - **DROP AUDIT POLICY**
  - **ALTER AUDIT POLICY**
- 激活/禁用审计策略
  - **AUDIT POLICY**
  - **NOAUDIT POLICY**
- 查询audit trail: **AUDIT_TRAIL**
- 删除audit trail: **ALTER DATABASE CLEAR AUDIT TRAIL**

8.2 ALTER CLUSTER GROUP name ADD MEMBER

功能

在集群组（cluster group）中添加集群成员（cluster member）

语句

```
<alter cluster group add member statement> ::=
    ALTER CLUSTER GROUP group_name ADD
        <cluster member definition> [, ...]
    ;

<cluster member definition> ::=
    CLUSTER MEMBER member_name <connection attribute> [<member position>]

<connection attribute> ::=
    HOST 'address' PORT port_no

<member position> ::=
    POSITION DEFAULT
  | POSITION MAX
```

```
| POSITION number
```

使用范围及访问权限

可在集群系统（cluster system）中执行

用户需要拥有ADMINISTRATION ON DATABASE权限才能执行<alter cluster group add member statement>语句

语句规则及参数

group_name

集群组（cluster group）的名称

<cluster member definition>

定义包含在集群组中的集群成员

集群组最多可包含32个集群成员

member_name

集群成员的名称

集群成员名称应与在生成该成员的数据库时定义的成员名称相同

不可存在相同的集群组集群成员名称

名称的长度应小于128字节

集群成员的start-up阶段应为GLOBAL OPEN阶段

<connection attribute>

定义用于集群成员之间通信的连接信息

<connection attribute> 应与在生成该成员的数据库时定义的HOSTPORT相同

HOST与PORT组合应在集群系统中具有唯一性

- HOST 'address'使用host name或者IPv4地址 使用host name时使用系统的第一个IPv4地址
- PORT port_no应为1024 ~ 49151范围的值

<member position>

指定cluster member的position number

- POSITION DEFAULT
  - 系统自动指定position number
- POSITION MAX
  - 即使有空的position number也指定新的member position number
  - 指定比最大的member position更大的值
- POSITION number
  - 指定属于number的position number

    - 该position number应在集群系统中是唯一的
    - 该position number是空的position number应小于或等于最大的position number
- 省略时默认值为POSITION DEFAULT

集群成员的member_position信息可通过DBA_CLUSTER view查询

```
SELECT member_name, member_id, member_position FROM dba_cluster;
```

例如使用如下position number时

- G1N1: 0
- G1N2: 1
- G2N2: 3
- G3N2: 5

根据如下选项指定如下值

- POSITION DEFAULT
    - 指定空值2
- POSITION MAX
    - 指定新的position number值6
- POSITION 3
    - 重复因此error
- POSITION 4
    - 指定position number 4

说明

<alter cluster group add member statement> 语句不重新分配表的shard

为了在添加的集群成员中重新分配shard需执行以下语句

- **ALTER DATABASE REBALANCE**
- **ALTER TABLE name REBALANCE**

使用示例

以下为在集群组添加两个集群成员的示例

```
gSQL>
ALTER CLUSTER GROUP g1 ADD
    CLUSTER MEMBER g1n3 HOST '192.168.0.13' PORT 10130,
    CLUSTER MEMBER g1n4 HOST '192.168.0.14' PORT 10140
;


Cluster Group altered.
```

以下为将空的member position指定为cluster member position的示例

```
ALTER CLUSTER GROUP g2 ADD
    CLUSTER MEMBER g2n1 HOST '192.168.0.21' PORT 10210 POSITION 4
;
```

兼容性

标准SQL中未定义集群的概念

参考

相关内容参考下文

- **CREATE CLUSTER GROUP**
- **DROP CLUSTER GROUP**
- **ALTER DATABASE REBALANCE**
- **ALTER TABLE name REBALANCE**

8.3 ALTER CLUSTER GROUP name OFFLINE MEMBER

功能

将集群组的集群成员变更为offline状态

语句

```
<alter cluster group offline member statement> ::=
    ALTER CLUSTER GROUP group_name OFFLINE CLUSTER MEMBER member_name
    ;
```

使用范围及访问权限

可在集群系统中执行

用户需要拥有ADMINISTRATION ON DATABASE权限才能执行<alter cluster group offline member statement>语句

语句规则及参数

group_name

集群组的名称

member_name

集群成员的名称

集群成员应包含在group_name的集群组

集群成员应为inactive状态

说明

使Inactive状态的集群成员offline

<alter cluster group offline member statement>语句不重新分配表的shard

使用示例

将非Inactive状态的集群成员变更为offline会发生如下错误

```
gSQL>
```

```
ALTER CLUSTER GROUP g1 OFFLINE CLUSTER MEMBER g1n2;


ERR-42000(16417): active member 'G1N2' cannot be offlined
```

以下将特定集群成员变更为offline状态的示例

```
gSQL>


ALTER CLUSTER GROUP g1 OFFLINE

    CLUSTER MEMBER g1n3

;

Cluster Group altered.
```

兼容性

标准SQL中未定义集群的概念

参考

相关内容参考下文

- **CREATE CLUSTER GROUP**
- **DROP CLUSTER GROUP**
- **ALTER DATABASE REBALANCE**
- **ALTER TABLE name REBALANCE**

8.4 ALTER CLUSTER LOCATION

功能

修改集群位置（cluster location）信息

语句

```
<alter cluster location statement> ::=
    ALTER CLUSTER LOCATION member_name
    <cluster connection attribute>
    ;

<cluster connection attribute> ::
       HOST 'address' PORT port_no
```

使用范围及访问权限

可在集群系统中执行

用户需要有ADMINISTRATION ON DATABASE权限才能执行<alter cluster location statement>语句

语句规则及参数

member_name

集群成员的名称

已有的集群位置信息中应存在相同的集群成员名称

名称的长度应小于128字节

<cluster connection attribute>

定义用于集群成员之间通信的连接信息

HOST与PORT组合应在集群系统中具有唯一性

- HOST 'address'使用host name或者IPv4地址 使用host name时使用系统的第一个IPv4地址
- PORT port_no应为1024 ~ 49151范围的值

说明

变更集群位置的连接信息时无需删除或重建集群成员可使用**ALTER CLUSTER LOCATION** 变更连接信息

使用示例

```
gSQL>
ALTER CLUSTER LOCATION g1n2
    HOST '192.168.0.12' PORT 10120
;

Location altered.
```

兼容性

标准SQL中未定义集群的概念

参考

相关内容参考下文

- **CREATE CLUSTER LOCATION**
- **DROP CLUSTER LOCATION**

8.5 ALTER DATABASE ADD LOGFILE

功能

在数据库中新增日志文件组或日志文件成员

语句

```
<alter database add logfile statement> ::=
      <add logfile member statement>
    | <add logfile group statement>
    ;

<add logfile member statement> ::=
    ALTER DATABASE ADD LOGFILE MEMBER <add logfile clause> [, ...] TO
        <group clause>

<add logfile group statement> ::=
    ALTER DATABASE ADD LOGFILE <group clause> ( 'logfile_name' )
        <size clause> [ REUSE ]

<group clause> ::=
    GROUP integer
```

```
<add logfile clause> ::=
    'logfile_name' [ REUSE ]

<size clause> ::=
    integer [ M | G ]
```

使用范围及访问权限

用户需要拥有ALTER DATABASE ON DATABASE权限才能执行<alter database add logfile statement>语句

语句规则及参数

<alter database add logfile statement>

数据库应为MOUNT状态

<add logfile member statement>

在现有的日志文件组中新增日志文件成员

- <add logfile clause>
    - 'logfile_name'是要添加到日志文件组的日志文件成员的文件名

- 文件不存在时生成新的文件
- 'logfile_name'的长度应小于1024byte

- <group clause>
  - 指定新增到数据库的日志文件组的标识符
  - integer应为已有日志文件组的标识符
  - 不存在integer对应的标识符时报错

<add logfile group statement>

新增日志文件组

- 在CURRENT日志文件组后面新增日志文件组
- <group clause>
  - 指定要新增到数据库的日志文件组的标识符
  - integer应为不存在的日志文件组的标识符
  - 存在integer对应的标识符时报错
- <size clause>
  - 文件大小可指定为20MB ~ 120GB
  - 文件大小应大于日志缓冲区（redo log buffer）和延迟日志缓冲区（pending log buffer）大小之和
- 已存在logfile_name并使用REUSE选项时如果与日志文件组的其他日志文件成员大小一致则再使用现有的日志文件

说明

添加新的日志文件组或日志文件成员时存储在控制文件因此以防控制文件损坏建议备份控制文件

使用示例

以下为在现有的日志文件GROUP 3新增2个日志文件成员的示例

```
ALTER DATABASE ADD LOGFILE MEMBER 'logfile1.log', 'logfile2.log' TO GROUP 3;
```

以下为在数据库新增日志文件大小为100M名称为'logfile1.log'的新日志文件组4的示例

```
ALTER DATABASE ADD LOGFILE GROUP 4 ( 'logfile1.log' ) SIZE 100M;
```

> Note:
>
> 新增日志文件组时至少要使用一个日志文件在现有的日志组中可新增多个日志文件成员

兼容性

标准SQL未定义ALTER DATABASE语句

参考

相关内容参考下文

- **ALTER DATABASE ADD LOGFILE**
- **ALTER DATABASE DROP LOGFILE**
- **ALTER DATABASE RENAME LOGFILE**

8.6 ALTER DATABASE ARCHIVELOG

功能

变更数据库的在线日志文件的归档模式

语句

```
<alter database archivelog statement> ::=
    ALTER DATABASE { ARCHIVELOG | NOARCHIVELOG }
    ;
```

使用范围及访问权限

用户需要拥有ALTER DATABASE ON DATABASE权限才能执行<alter database archivelog statement>语句

语句规则及参数

<alter database archivelog statement>

- 数据库应为MOUNT状态

- ARCHIVELOG
  - 归档在线日志文件
- NOARCHIVELOG
  - 不归档在线日志文件

说明

在归档模式下才能执行数据库备份与使用备份的恢复（media recovery）

使用示例

以下为把数据库设置为归档模式的示例

```
ALTER DATABASE ARCHIVELOG;
```

兼容性

标准SQL未定义ALTER DATABASE语句

参考

相关内容参考下文

- **ALTER DATABASE BACKUP**
- **ALTER TABLESPACE name BACKUP**

8.7 ALTER DATABASE BACKUP

功能

为了执行数据库完全备份（full backup）把备份状态设置为'ACTIVE'或'INACTIVE'之后执行数据库增量备份（incremental backup）控制文件（control file）备份

语句

```
<alter database backup statement> ::=
      <database begin backup statement>
    | <database end backup statement>
    | <database incremental backup statement>
    | <database controlfile backup statement>
    ;

<database begin backup statement> ::=
    ALTER DATABASE BEGIN BACKUP [ AT <domain name> ]
    ;

<database end backup statement> ::=
    ALTER DATABASE END BACKUP [ AT <domain name> ]
    ;
```

```
<database incremental backup statement> ::=
    ALTER DATABASE BACKUP INCREMENTAL
        <incremental backup option> [ AT <domain name> ]    ;

<incremental backup option> ::=
      LEVEL integer [ CUMULATIVE | DIFFERENTIAL ]

<database controlfile backup statement> ::=
    ALTER DATABASE BACKUP CONTROLFILE TO 'target_name'
        [ AT <domain name> ]    ;
```

使用范围及访问权限

用户需要拥有ALTER DATABASE ON DATABASE权限才能执行<alter database backup statement> 语句

语句规则及参数

<database begin backup clause>

把数据库设置为可执行完全备份的状态

- 把在数据库中生成使用的所有'ONLINE'状态的表空间设置为可执行完全备份的状态

- 数据库应为OPEN状态并以归档模式运行
- BEGIN BACKUP开始后无法执行如下需要写入数据文件的操作
  - SHUTDOWN NORMAL
  - OFFLINE / DROP TABLESPACE
  - ADD / DROP DATAFILE
- 完全备份状态为'ACTIVE'时如果实例非正常结束则重启时可以执行介质恢复

<database end backup clause>

把数据库设置为不可执行完全备份的状态

- 把在数据库中生成并使用的所有'ONLINE'状态的表空间设置为不可执行备份的状态
- 数据库应为OPEN状态并以归档日志模式运行

<database incremental backup statement>

- 对数据库执行增量备份
- 数据库应为OPEN状态并以归档日志模式运行

<incremental backup option>

- 'integer'可指定0 ~ 4
- 'LEVEL 0'无法指定CUMULATIVE或DIFFERENTIAL
- CUMULATIVE | DIFFERENTIAL
  - CUMULATIVE

        - 'integer'为n时备份最近的'LEVEL 0' ~ 'LEVEL n-1'之后备份变更的所有页
    - DIFFERENTIAL
        - 'integer'为n时备份最近的'LEVEL 0' ~ 'LEVEL n'之后备份变更的所有页
    - 省略时默认为DIFFERENTIAL

<database controlfile backup statement>

- 备份控制文件（controlfile）
    - 'target_name'的名称应小于1024byte
    - 'target_name'已存在时该操作将失败
- 数据库应为OPEN状态并以归档日志模式运行

> Note:
>
> 在SUNDB中管理的'target_name'的最大长度为1024byte但由于每个操作系统所允许的文件名的最长限度不同因此实际生成的'target_name'的长度可能小于1024byte

<domain name>

- 执行语句的成员名或群组的名
- 未指定时将对所有群组执行此命令

说明

备份数据库的数据文件和控制文件数据库的完全备份在执行BEGIN BACKUP后使用操作系统的

文件拷贝拷贝数据文件后执行END BACKUP结束备份而增量备份通过一条语句在BACKUP_DIR_1 property中指定的路径下生成增量备份文件

使用示例

以下为将完全备份状态设置为'ACTIVE'的示例

```
ALTER DATABASE BEGIN BACkUP;
```

以下为将完全备份状态设置为'INACTIVE'的示例

```
ALTER DATABASE END BACkUP;
```

以下为使用DIFFERENTIAL生成LEVEL 1增量备份的示例

```
ALTER DATABASE BACKUP INCREMENTAL LEVEL 1 DIFFERENTIAL;
```

以下为生成控制文件的'controlfile.bak'备份文件的示例不包含绝对路径时在LOG_DIR property中指定的路径下生成备份文件

```
ALTER DATABASE BACKUP CONTROLFILE TO 'controlfile.bak';
```

兼容性

标准SQL未定义ALTER DATABASE语句

参考

相关内容参考下文

- **ALTER TABLESPACE name BACKUP**
- **ALTER DATABASE RECOVER**

8.8 ALTER DATABASE CLEAR AUDIT TRAIL

功能

删除(purge)由于应用audit policy而累计的audit record

语句

```
<clear audit trail statement> ::=

    ALTER DATABASE CLEAR AUDIT TRAIL

;
```

使用范围及访问权限

用户需要拥有AUDIT SYSTEM ON DATABASE权限才能执行<clear audit trail statement>语句

说明

激活audit policy后随着时间推移audit trail持续扩大

构成audit trail的表存储在 MEM_AUX_TBS表空间应防止audit trail持续扩大

存储Audit Trail

需存储audit trail时按照以下步骤存储后删除audit trail

- 首次执行时

```
CREATE TABLE backup_audit_trail AS SELECT * FROM AUDIT_TRAIL;
COMMIT;
```

- 反复执行时

```
INSERT INTO backup_audit_trail SELECT * FROM AUDIT_TRAIL;
COMMIT;
```

- 删除audit trail时

```
ALTER DATABASE CLEAR AUDIT TRAIL;
```

使用示例

使用以下语句删除（purge）audit trail

```
ALTER DATABASE CLEAR AUDIT TRAIL;
```

兼容性

标准SQL中无audit policy

参考

相关内容参考下文

- 管理audit policy
  - **CREATE AUDIT POLICY**
  - **DROP AUDIT POLICY**
  - **ALTER AUDIT POLIC**
- 激活/禁用审计策略
  - **AUDIT POLICY**
  - **NOAUDIT POLICY**
- 查询audit trail: **AUDIT_TRAIL**
- 删除audit trail: **ALTER DATABASE CLEAR AUDIT TRAIL**

8.9 ALTER DATABASE CLEAR PASSWORD HISTORY

功能

删除因应用profile而累计的用户密码历史记录

语句

```
<clear password history statement> ::=
    ALTER DATABASE CLEAR PASSWORD HISTORY
    ;
```

使用范围及访问权限

用户需要拥有ALTER DATABASE ON DATABASE权限才能执行<clear password history statement>语句

说明

在用户应用profile时根据PASSWORD_REUSE_MAXPASSWORD_REUSE_TIME的策略累计用户的密码变更历史记录

PASSWORD_REUSE_MAX	PASSWORD_REUSE_TIME	变更历史记录管理
value	value	仅管理value范围内的变更历史记录自动删除超出范围的变更历史记录
value	UNLIMITED	需要检测所有变更历史记录因此仅累计而不删除
UNLIMITED	value	需要检测所有变更历史记录因此仅累计而不删除
UNLIMITED	UNLIMITED	不检测变更历史记录因此也不管理变更历史记录

Table 8-1 变更历史记录管理

<clear password history statement>语句删除累计的用户密码变更历史记录

使用示例

以下为执行<clear password history statement>语句的示例

```
gSQL> ALTER DATABASE CLEAR PASSWORD HISTORY;

Database altered.

gSQL> COMMIT;
```

```
Commit complete.
```

兼容性

标准SQL未定义ALTER DATABASE语句

参考

相关内容参考下文

- **CREATE PROFILE**
- **CREATE USER**

8.10 ALTER DATABASE DATAFILE AUTOEXTEND

功能

变更磁盘表空间数据文件的自动扩张属性将自动扩张属性变更为ON时也可变更要扩张的大小和数据文件的最大大小

语句

```
<alter database datafile autoextend statement> ::=
    ALTER DATABASE DATAFILE datafile_name <autoextend clause>
        [ AT <domain name> ]
    ;

<autoextend clause>
    AUTOEXTEND { ON [ <next size clause> ] [ <max size clause> ] | OFF }

<next size clause>
    NEXT <size clause>

<max size clause>
    MAXSIZE { <size clause> | UNLIMITED }
```

使用范围及访问权限

用户需要拥有ALTER DATABASE ON DATABASE权限才能执行<alter database datafile autoextend statement>语句

变更数据文件的自动扩张属性仅限于磁盘表空间

datafile_name

指定要变更的数据文件的名称

<autoextend clause>

将自动扩张属性设置为ON或OFF设置为ON时可指定自动扩张大小和数据文件的最大大小

<next size clause>

当前使用中的数据文件没有可使用的空间时指定要扩张的大小

<max size clause>

指定数据文件可扩张的最大大小

说明

参考各语句的使用规则

使用示例

以下为变更数据文件的自动扩张属性和自动扩张大小数据文件大小的示例

```
gSQL> ALTER DATABASE DATAFILE 'DISK_TBS.dbf' AUTOEXTEND OFF;

Database altered.

gSQL> ALTER DATABASE DATAFILE 'DISK_TBS.dbf' AUTOEXTEND ON;

Database altered.

gSQL> ALTER DATABASE DATAFILE 'DISK_TBS.dbf' AUTOEXTEND ON NEXT 20M;

Database altered.

gSQL> ALTER DATABASE DATAFILE 'DISK_TBS.dbf' AUTOEXTEND ON MAXSIZE 1G;

Database altered.
```

```
gSQL> ALTER DATABASE DATAFILE 'DISK_TBS.dbf' AUTOEXTEND ON 20M MAXSIZE 1G;


Database altered.
```

兼容性

标准SQL未定义数据文件相关概念

参考

相关内容参考**CREATE DISK DATA TABLESPACE**

8.11 ALTER DATABASE DELETE BACKUP

功能

删除增量备份（incremental backup）的备份信息与备份文件可以删除数据库的所有增量备份或选择性的删除不再需要的备份（obsolete backup）

语句

```
<alter database delete backup statement> ::=
    ALTER DATABASE DELETE <delete backup list option>
        BACKUP LIST [ <including backup file option> ]
    ;

<delete backup list option> ::=
      OBSOLETE
    | ALL

<including backup file option> ::=
    INCLUDING BACKUP FILES
```

使用范围及访问权限

用户需要拥有ALTER DATABASE ON DATABASE权限才能执行<alter database delete backup statement> 语句

语句规则及参数

<alter database delete backup statement>

数据库应为MOUNT或OPEN状态

<delete backup list option>

选择现有增量备份中要删除的对象

- OBSOLETE：删除对象是最近的数据库'LEVEL 0'备份之前备份的数据库或表空间备份文件
- ALL：删除对象是所有的增量备份

<including backup file option>

- 省略时在控制文件中仅删除备份信息
- 同时删除备份信息和备份文件

说明

删除OBSOLETE增量备份时删除最近执行的LEVEL 0数据库备份之前的增量备份即如果执行不是LEVEL 0的增量备份时即使包含之前执行的增量备份的增量备份也不会删除因为在使用增量备份执行不完全恢复时可能会使用该增量备份

> Caution:
>
> 删除增量备份时如果同时删除备份文件则即使使用有增量备份信息的已备份的控制文件也无法执行恢复因此需谨慎

使用示例

以下为删除所有现有增量备份的备份信息与备份文件的示例

```
ALTER DATABASE DELETE ALL BACKUP LIST INCLUING BACKUP FILES;
```

兼容性

标准SQL未定义ALTER DATABASE语句

参考

相关内容参考下文

- **ALTER TABLESPACE name BACKUP**
- **ALTER DATABASE RECOVER**

8.12 ALTER DATABASE DROP INACTIVE CLUSTER MEMBERS

功能

删除所有inactive cluster member

语句

```
<alter database drop inactive members statement> ::=
    ALTER DATABASE DROP [ FORCE | NO FORCE ] INACTIVE CLUSTER MEMBERS
    ;
```

使用范围及访问权限

可在集群系统中执行

用户需要拥有ADMINISTRATION ON DATABASE权限才能执行<alter database drop inactive cluster members statement>语句

语句规则及参数

[FORCE | NO FORCE]

- FORCE
  - 即便有可能会丢失数据但也会删除inactive cluster member
- NO FORCE
  - 有可能丢失数据时不能删除inactive cluster member
- 默认值为NO FORCE

说明

删除所有inactive cluster member

集群成员的inactive状态为未与集群系统连接的状态发生在如下情况中

- 在运行中的集群系统中该集群成员发生故障时
- 不启动该集群成员而尝试启动集群系统时

但删除cluster member时丢失table的shard则无法删除inactive cluster member

并且删除cluster member时有可能导致数据丢失的话那么不能删除inactive cluster member 当能保证该cluster group的其他成员的数据比要删除的inactive cluster member的表或者shard的replica的数据新时才能防止数据丢失所以cloned table时在整个clustersharded table时在同一个cluster group中至少有一个online成员存在时才会允许删除inactive cluster member

但cluster group中没有online状态的cluster member又因inactive cluster member不能继续提供服务时可以在接受数据丢失风险的情况下使用FORCE参数删除inactive cluster member

建议集群系统中无法再包含所有inactive cluster member时使用<alter database drop inactive members statement>语句

使用示例

以下为执行<alter database drop inactive members statement>语句的示例

```
gSQL> ALTER DATABASE DROP INACTIVE CLUSTER MEMBERS;


Database altered.
```

兼容性

标准SQL中未定义集群的概念

参考

相关内容参考**ALTER SYSTEM JOIN DATABASE**

8.13 ALTER DATABASE DROP LOGFILE

功能

删除数据库中的日志文件组或成员

语句

```
<alter database drop logfile statement> ::=
      <drop logfile group statement>
    | <drop logfile member statement>
    ;

<drop logfile group statement> ::=
    ALTER DATABASE DROP LOGFILE <group clause>

<group clause> ::=
    GROUP integer

<drop logfile member statement> ::=
    ALTER DATABASE DROP LOGFILE MEMBER <logfile_list>

<logfile_list> ::=
```

```
    'logfile_name'
  | <logfile_list> , 'logfile_name'
```

使用范围及访问权限

用户需要有ALTER DATABASE ON DATABASE权限才能执行<alter database drop logfile statement> 语句

语句规则及参数

<alter database drop logfile statement>

数据库应为MOUNT状态

要删除的日志文件为CURRENT或ACTIVE状态时报错

删除后至少留下4个日志文件组

<drop logfile group statement>

删除现有的日志文件组

- <group clause>
  - 指定要删除的日志文件组
  - integer应为已有的日志文件的标识符
  - 不存在integer时报错

<drop logfile member statement>

删除现有的日志文件成员

- <logfile_list>
  - 要删除的日志文件成员的列表
  - 'logfile_name'应为已有的文件名
  - 'logfile_name'不存在时报错

说明

参考各语句的使用规则

使用示例

以下为删除现有日志文件GROUP 3的示例

```
ALTER DATABASE DROP LOGFILE GROUP 3;
```

以下为在现有日志文件GROUP 3删除'logfile1.log'与'logfile2.log'的示例

```
ALTER DATABASE DROP LOGFILE MEMBER 'logfile1.log', 'logfile2.log';
```

兼容性

标准SQL未定义ALTER DATABASE语句

参考

相关内容参考下文

- **ALTER DATABASE ADD LOGFILE**
- **ALTER DATABASE RENAME LOGFILE**

8.14 ALTER DATABASE DROP OFFLINE SEGMENTS

功能

用于删除所有表的离线shard的段

语句

```
<alter database drop offline segments statement> ::=
    ALTER DATABASE DROP OFFLINE SEGMENTS
    ;
```

使用范围及访问权限

可在集群系统中执行

用户须拥有ALTER DATABASE ON DATABASE权限才能执行<alter database drop offline segments statement>语句

说明

用于删除所有表的离线shard的段即使存在inactive cluster member也可执行

集群成员的inactive状态表示未和集群系统连接会在以下情况中发生

- 当相应集群成员在运行中的集群系统中发生故障时
- 在不运行相应集群成员的情况下试图启动集群系统时

<alter database drop offline segments statement>对各表执行**<alter table drop offline segments statement>**它相当于以下查询的集合

```
ALTER TABLE t1 DROP OFFLINE SEGMENTS;
COMMIT;
ALTER TABLE t2 DROP OFFLINE SEGMENTS;
COMMIT;
ALTER TABLE t3 DROP OFFLINE SEGMENTS;
COMMIT;

...

ALTER TABLE tn DROP OFFLINE SEGMENTS;
COMMIT;
```

即使在特定表中发生错误<alter database drop offline segments statement>也不会终止而是对下一个表继续执行并在显示以下警告的同时成功执行

```
gSQL> ALTER DATABASE DROP OFFLINE SEGMENTS;


ERR-42000(16553): of the total '5' tables, '1' tables failed to drop
offline segments
Database altered.
```

以上报错信息表示五个表中的一个表失败了

如果在对错误进行适当的处理后重新执行<alter database drop offline segments statement>语句的话则会仅对失败的表重新执行该语句

关于错误的详细内容参考执行语句成员的系统跟踪日志(system.trc)

使用示例

以下为执行<alter database drop offline segments statement>语句的示例

```
gSQL> ALTER DATABASE DROP OFFLINE SEGMENTS;


Database altered.
```

兼容性

标准SQL中未定义集群的概念

参考

相关内容参考**ALTER TABLE name DROP OFFLINE SEGMENTS**

8.15 ALTER DATABASE MOVE SHARD

功能

将特定集群组的所有表的shard重新分配到其他集群组中

语句

```
<alter database move shard statement> ::=
    ALTER DATABASE MOVE SHARD FROM CLUSTER GROUP src_cluster_group
        TO CLUSTER GROUP dest_cluster_group
       [ ONLINE | OFFLINE ]
       [ <shard divisor> ]
       [ <parallel clause> ]
    ;

<shard divisor> ::=
    SHARD DIVISOR integer

<parallel clause> ::=
    NOPARALLEL
  | PARALLEL [ integer ]
```

使用范围及访问权限

可在集群系统中执行

用户需要拥有ALTER DATABASE ON DATABASE权限才能执行<alter database move shard statement>语句

语句规则及参数

src_cluster_group

移动表的shard的集群组

dest_cluster_group

移动表的shard的目标集群组

[ONLINE | OFFLINE]

决定在重新分配表的shard时是否允许DML

- ONLINE
  - 允许INSERT, UPDATE, DELETE
- OFFLINE
  - 不允许INSERT, UPDATE, DELETE

- 省略时默认值为ONLINE

<shard divisor>

指定shard分区数量

- 按照分区的数量划分shard并重新分配到远程服务器
- 整数可从0开始使用最大值为1000
- 省略时根据REBALANCE_SHARD_DIVISOR属性决定
- integer小于parallel integer时修改为和parallel integer相同的值

<parallel clause>

指定重新分配表时使用的线程数量

- NOPARALLEL
  - 不并行重新分配表
- PARALLEL [integer]
  - 并行重新分配表
  - 整数可以从0开始使用最大值为64
  - 省略整数时为0
  - 整数为0时 系统确定最佳值

说明

通过以下语句添加cluster member和cluster group时不重新分配表的shard

- **CREATE CLUSTER GROUP**
- **ALTER CLUSTER GROUP name ADD MEMBER**

添加集群组和集群成员时执行<alter database rebalance statement>语句重新分配未重新分配的所有表的shard

如下对未重新分配shard的表执行<alter database move shard statement>语句

```
ALTER TABLE t1 MOVE SHARD FROM CLUSTER GROUP src_group TO CLUSTER GROUP
dest_group;
COMMIT;
ALTER TABLE t2 MOVE SHARD FROM CLUSTER GROUP src_group TO CLUSTER GROUP
dest_group;
COMMIT;
ALTER TABLE t3 MOVE SHARD FROM CLUSTER GROUP src_group TO CLUSTER GROUP
dest_group;
COMMIT;
...
...
ALTER TABLE t_n MOVE SHARD FROM CLUSTER GROUP src_group TO CLUSTER GROUP
dest_group;
COMMIT;
```

除了CLONED表或设置为CLUSTER WIDE的表以外其他表均按上述方式执行即使特定表的shard重新分配执行失败<alter database move shard statement>语句也会持续进行也不会回滚成功执行重新分配shard的表

因此对报错信息采取适当的应对措施后重新执行<alter database move shard statement>语句时已成功执行重新分配shard的表不包含在重新分配对象内只对需要重新分配的表重新分配shard

使用示例

以下为执行<alter database move shard statement>语句的示例

```
gSQL> ALTER DATABASE MOVE SHARD FROM CLUSTER GROUP G1 TO CLUSTER GROUP G2;


Database altered.
```

兼容性

标准SQL中未定义集群的概念

参考

相关内容参考下文

- **ALTER TABLE name MOVE SHARD**

- **CREATE CLUSTER GROUP**
- **ALTER CLUSTER GROUP name ADD MEMBER**

8.16 ALTER DATABASE OFFLINE INACTIVE CLUSTER MEMBERS

功能

将所有禁用的inactive cluster member变更为offline状态即将相应集群成员的shard map变更为offline状态

语句

```
<alter database offline inactive cluster members statement> ::=
    ALTER DATABASE OFFLINE INACTIVE CLUSTER MEMBERS
    ;
```

使用范围及访问权限

可在集群系统中执行

用户需要拥有ADMINISTRATION ON DATABASE权限才能执行<alter database offline inactive cluster members statement>语句

语句规则及参数

将所有Inactive cluster member变更为offline状态

集群成员的inactive状态为未与集群系统连接的状态发生在如下情况

- 在运行中的集群系统中该集群成员发生故障时
- 不启动该集群成员而尝试启动集群系统时

说明

当所有的inactive cluster member不再包含在集群系统中时建议使用<alter database offline inactive members statement>语法

如果Inactive cluster member可参与到集群系统则执行**ALTER SYSTEM JOIN DATABASE**语句使其包含在集群系统中

变更为offline状态的集群成员在join后可通过以下语句重新变更为online状态

- **ALTER DATABASE REBALANCE**
- **ALTER TABLE name REBALANCE**

使用示例

```
gSQL> ALTER DATABASE OFFLINE INACTIVE CLUSTER MEMBERS;
```

兼容性

标准SQL中未定义集群的概念

参考

相关内容参考下文

- **ALTER SYSTEM JOIN DATABASE**
- **ALTER DATABASE REBALANCE**
- **ALTER TABLE name REBALANCE**

8.17 ALTER DATABASE REBALANCE

功能

重新分配所有表的shard

语句

```
<alter database rebalance statement> ::=
    ALTER DATABASE REBALANCE
       [ ONLINE | OFFLINE ]
       [ <shard divisor> ]
       [ <parallel clause> ]
    ;

<shard divisor> ::=
    SHARD DIVISOR integer

<parallel clause> ::=
    NOPARALLEL
  | PARALLEL [ integer ]
```

使用范围及访问权限

可在集群系统中执行

用户需要拥有ALTER DATABASE ON DATABASE权限才能执行<alter database rebalance statement>语句

语句规则及参数

[ONLINE | OFFLINE]

决定重新分配表的shard时是否允许DML

- ONLINE
  - 允许INSERT, UPDATE, DELETE
- OFFLINE
  - 不允许INSERT, UPDATE, DELETE
- 省略时默认值为ONLINE

<shard divisor>

指定shard的分区数量

- 按照分区的数量划分shard并重新分配到远程服务器
- 整数可从0开始使用最大值为1000

- 省略时根据REBALANCE_SHARD_DIVISOR属性决定
- integer小于parallel integer时修改为和parallel integer相同的值

<parallel clause>

指定重新分配表时使用的线程数量

- NOPARALLEL
  - 不并行重新分配表
- PARALLEL [integer]
  - 并行重新分配表
  - 整数可以从0开始使用最大值为64
  - 省略整数时为0
  - 整数为0时 系统确定最佳值

说明

通过以下语句在添加cluster member, cluster group时 不重新分配表的shard

- **CREATE CLUSTER GROUP**
- **ALTER CLUSTER GROUP name ADD MEMBER**

执行<alter database rebalance statement>语句重新分配添加的集群组与集群成员中未重新分配的所有表的shard

如下对未重新分配shard的表执行<alter database rebalance statement>语句

```
ALTER TABLE t1 REBALANCE;

COMMIT;

ALTER TABLE t2 REBALANCE;

COMMIT;

ALTER TABLE t3 REBALANCE;

COMMIT;

...

...

ALTER TABLE t_n REBALANCE;

COMMIT;
```

即使特定表的shard重新分配执行失败<alter database rebalance statement>语句也会持续进行并不会回滚已成功执行重新分配shard的表

因此在采取适当的应对措施后重新执行<alter database rebalance statement>语句时已成功执行重新分配shard的表不包含在重新分配对象内只对需要重新分配的表重新分配shard

使用示例

以下为执行<alter database rebalance statement>语句的示例

```
gSQL> ALTER DATABASE REBALANCE;


Database altered.
```

兼容性

标准SQL中未定义集群的概念

参考

相关内容参考**ALTER TABLE name REBALANCE**

8.18 ALTER DATABASE REBALANCE EXCLUDE CLUSTER GROUP

功能

重新分配所有表的shard使特定集群组不包含shard

语句

```
<alter database rebalance exclude cluster group statement> ::=
    ALTER DATABASE REBALANCE EXCLUDE CLUSTER GROUP cluster_group_name
       [ ONLINE | OFFLINE ]
       [ <shard divisor> ]
       [ <parallel clause> ]
    ;

<shard divisor> ::=
    SHARD DIVISOR integer

<parallel clause> ::=
    NOPARALLEL
  | PARALLEL [ integer ]
```

使用范围及访问权限

可在集群系统中执行

用户需要拥有ALTER DATABASE ON DATABASE权限才能执行<alter database rebalance exclude cluster group statement>语句

语句规则及参数

cluster_group_name

不包含表的shard的集群组的名称

指定的集群组是唯一的集群组时无法执行此语句

[ONLINE | OFFLINE]

决定重新分配表的shard时是否允许DML

- ONLINE
  - 允许INSERT, UPDATE, DELETE
- OFFLINE
  - 不允许INSERT, UPDATE, DELETE
- 省略时默认值为ONLINE

<shard divisor>

指定shard的分区数量

- 按照分区的数量划分shard并重新分配到远程服务器
- 整数可从0开始使用最大值为1000
- 省略时根据REBALANCE_SHARD_DIVISOR属性决定
- integer小于parallel integer时修改为和parallel integer相同的值

<parallel clause>

指定重新分配表时使用的线程数量

- NOPARALLEL
  - 不并行重新分配表
- PARALLEL [integer]
  - 并行重新分配表
  - 整数可以从0开始使用最大值为64
  - 省略整数时为0
  - 整数为0时 系统确定最佳值

说明

通过**DROP CLUSTER GROUP**语句删除集群组时该集群组中不能有shard

执行<alter database rebalance exclude cluster group statement>语句使该集群组不包含shard对

该集群组的包含shard的表执行 <alter database rebalance exclude cluster group statement>语句有如下意义

```
ALTER TABLE t1 REBALANCE EXCLUDE CLUSTER GROUP g3;
COMMIT;
ALTER TABLE t2 REBALANCE EXCLUDE CLUSTER GROUP g3;
COMMIT;
ALTER TABLE t3 REBALANCE EXCLUDE CLUSTER GROUP g3;
COMMIT;
...
...
ALTER TABLE t_n REBALANCE EXCLUDE CLUSTER GROUP g3;
COMMIT;
```

由于存储空间不足等导致<alter database rebalance exclude cluster group statement>语句失败时不回滚已成功执行排除shard的表

因此对报错采取适当的应对措施后重新执行<alter database rebalance exclude cluster group statement>语句时已成功执行排除shard的表不包含在重新分配的对象内只对需要重新分配的表进行排除shard并重新分配

使用示例

以下为执行<alter database rebalance exclude cluster group statement>语句的示例

```
gSQL> ALTER DATABASE REBALANCE EXCLUDE CLUSTER GROUP g3;
```

```
Database altered.
```

兼容性

标准SQL中未定义集群的概念

参考

相关内容参考下文

- **DROP CLUSTER GROUP**
- **ALTER TABLE name REBALANCE EXCLUDE CLUSTER GROUP cluster_group_list**

8.19 ALTER DATABASE RECOVER

功能

使用在线/归档日志文件恢复数据库的所有或部分数据文件

语句

```
<alter database recover statement> ::=
      <complete database recover statement>
    | <datafile recover statement>
    | <complete tablespace recover statement>
    | <incomplete database recover statement>
    ;

<complete database recover statement> ::=
    ALTER DATABASE RECOVER

<datafile recover statement> ::=
    ALTER DATABASE RECOVER DATAFILE <datafile recovery clause>

<datafile recovery clause> ::=
    <datafile recovery object> [, ...]
```

```
<datafile recovery object> ::=
    'datafile_name' [<recovery using backup option>] [recovery corruption
option>]

<recovery using backup option> ::=
    USING BACKUP 'backup_datafile_name'

<recovery corruption option> ::=
    CORRUPTION

<complete tablespace recover statement> ::=
    ALTER DATABASE RECOVER TABLESPACE tablespace_name

<incomplete database recover statement> ::=
      <batch incomplete recovery statement>
    | <interactive incomplete recovery statement>
    ;

<batch incomplete recovery statement> ::=
    ALTER DATABASE RECOVER <until clause> [<using backup controlfile
option>]

<until clause> ::=
      UNTIL CHANGE integer
```

```
<using backup controlfile option> ::=
    USING BACKUP CONTROLFILE

<interactive incomplete recovery statement> ::=
    ALTER DATABASE <incomplete recovery option> [<using backup controlfile
option>]

<incomplete recovery option> ::=
      BEGIN INCOMPLETE RECOVERY
    | END INCOMPLETE RECOVERY
    | RECOVER 'logfile name'
    | RECOVER AUTOMATICALLY
    | RECOVER SUGGESTION
    ;
```

使用范围及访问权限

用户需要拥有ALTER DATABASE ON DATABASE权限才能执行<alter database recover statement> 语句

语句规则及参数

<complete database recover statement>

使用在线及归档日志文件将数据库的数据文件恢复到最新状态

- 恢复'ONLINE'状态的所有表空间
- 数据库应为MOUNT状态并在归档模式下运行
- 所需的归档日志文件不存在时失败

<datafile recover statement>

通过Immediate option将offline状态的表空间数据文件备份的数据文件或由于备份过程中发生故障而需要使用归档日志文件进行恢复的表空间数据文件恢复到最新状态

- 数据文件可在MOUNT或OPEN阶段进行恢复
- OPEN阶段仅可恢复OFFLINE状态的表空间数据文件MOUNT阶段可恢复所有ONLINE/OFFLINE状态的数据文件
- 不存在所需的归档日志文件时恢复失败
- <datafile recovery clause>
    - 记述一个以上要恢复的datafile object list
- <datafile recovery object>
    - 设置要恢复的数据文件名称与恢复选项
- <recovery using backup option>
    - 设置要恢复的数据文件的备份数据文件名称
- <recovery corruption option>

- 设置是否仅恢复要恢复的数据文件中损坏（corrupt）的页（page）

<complete tablespace recover statement>

把表空间的数据文件恢复到最新状态

- 恢复数据库时数据库应为MOUNT或OPEN状态
- OPEN状态下仅可恢复OFFLINE状态的表空间MOUNT状态下可以恢复所有ONLINE / OFFLINE状态的表空间
- 不存在所需的归档日志文件时恢复失败
- 以下为需要表空间恢复操作的情况
  - 通过IMMEDIATE变更为OFFLINE状态的表空间
  - 需要使用备份的数据文件时
  - 完全备份过程中发生故障时

<incomplete database recover statement>

<batch incomplete database recover statement>

使用在线及归档日志文件将数据库的数据文件批量恢复到非最新状态的特定时间点

- 恢复'ONLINE'状态的所有表空间
- 数据库应为MOUNT状态并在归档模式下运行
- 使用不完全恢复的时间点之后的数据文件则失败
- 完成不完全恢复后必须通过RESETLOGS启动数据库
- <until clause>

    - 不完全恢复的特定时间点
    - UNTIL CHANGE：以日志为单位指定不完全恢复的时间点
- <using backup controlfile>
    - Deprecated

<interactive incomplete database recover statement>

使用在线及归档日志文件将数据库的数据文件以与用户对话的模式恢复至非最新状态的特定时间点

- 恢复ONLINE状态的所有表空间
- 数据库应为MOUNT状态并在归档模式下运行
- 使用不完全恢复的特定时间点之后的数据文件则失败
- 完成不完全恢复后必须通过RESETLOGS启动数据库
- <incomplete recovery option>
    - 用于以日志文件为单位执行对话式不完全恢复的选项
    - BEGIN INCOMPLETE RECOVERY：开始不完全恢复
    - END INCOMPLETE RECOVERY：结束不完全恢复
    - RECOVER 'logfile name'：用户手动设置执行恢复的日志文件
    - RECOVER AUTOMATICALLY：恢复可恢复的所有归档日志文件
    - RECOVER SUGGESTION：为了执行系统推荐的恢复恢复所需的归档日志文件
- <using backup controlfile>
    - Deprecated.

说明

数据库不完全恢复很难一次找到完成恢复的时间点所以需要执行多次并找到所需的恢复时间点但是如果执行不完全恢复后通过RESETLOGS选项启动数据库则会变为新的数据库因此为了执行多次不完全恢复需创建在线及归档日志文件的复件后再执行

使用示例

以下为执行数据库完全恢复的示例

```
ALTER DATABASE RECOVER;
```

以下为恢复数据文件的示例

```
ALTER DATABASE RECOVER DATAFILE 'test.dbf';
```

以下为恢复表空间的示例

```
ALTER DATABASE RECOVER TABLESPACE test_tbs;
```

以下为把数据库不完全恢复到LSN为11123的示例

```
ALTER DATABASE RECOVER UNTIL CHANGE 11123;
```

以下为恢复到可恢复的归档日志文件的对话式不完全恢复的示例

```
ALTER DATABASE BEGIN INCOMPLETE RECOVERY;

ALTER DATABASE RECOVER AUTOMATICALLY;

ALTER DATABASE END INCOMPLETE RECOVERY;
```

兼容性

标准SQL未定义ALTER DATABASE语句

参考

相关内容参考下文

- **ALTER DATABASE BACKUP**
- **ALTER TABLESPACE name BACKUP**
- **ALTER SYSTEM {MOUNT | OPEN} DATABASE**

8.20 ALTER DATABASE REGISTER

功能

在数据库中登记不可恢复的段（Segment）

语句

```
<alter database register statement> ::=
    ALTER DATABASE REGISTER IRRECOVERALBE SEGMENT
        <segment physical identifier list>
    ;

<segment physical identifier list> ::=
      integer
    | <segment physical identifier list> , integer
```

使用范围及访问权限

用户需要拥有ALTER DATABASE ON DATABASE权限才能执行<alter database register statement>

语句规则及参数

<alter database register statement>

在数据库中登记不可恢复的段（Segment）此语句只能在没有备份并无法恢复数据库时假设不再使用段的情况才能使用

- 数据库应为MOUNT状态
- 重启时初始化已登记的段标识符列表
- 成功重启服务器时已登记的段将变为'UNUSABLE'状态因此必须删除相应的段

<segment physical identifier list>

不可恢复的段的标识符列表

• Interger：8byte正数型段标识符

说明

服务器非正常结束后重启时执行数据库恢复在此过程中数据库为了恢复在上一个服务阶段未反映到磁盘的页使用重做日志在页上执行重新执行

如果执行重做操作时发生意外故障可使用该语句忽略其故障并执行恢复过程

使用示例

以下为放弃恢复标识符为4028679323648的段的示例

```
ALTER DATABASE REGISTER IRRECOVERABLE SEGMENT 4028679323648;
```

兼容性

标准SQL未定义ALTER DATABASE语句

参考

相关内容参考下文

- **ALTER TABLESPACE name BACKUP**
- **ALTER DATABASE RECOVER**

8.21 ALTER DATABASE RENAME GLOBAL TRANSACTION LOGFILE

功能

变更数据库中的全局事务日志文件名称

语句

```
<alter database rename global transaction logfile statement> ::=

    ALTER DATABASE RENAME GLOBAL TRANSACTION LOGFILE <source_clause>
       TO <target_clause>
    ;

<source_clause> ::= <logfile_list>

<target_clause> ::= <logfile_list>

<logfile_list> ::=
      'logfile_name'
    | <logfile_list>, 'logfile_name'
```

使用范围及访问权限

用户需要拥有ALTER DATABASE ON DATABASE权限才能执行<alter database rename global transaction logfile statement>语句

语句规则及参数

<alter database rename global transaction logfile statement>

- 数据库应为MOUNT状态
- source_clause
  - 数据库中要变更的全局事务日志文件列表
- target_clause
  - 数据库中要变更的全局事务日志文件列表
  - 文件不存在时报错
  - 包括路径的名称长度应小于1024字节

说明

参考各语句的使用规则

使用示例

以下为变更全局事务日志文件的示例

```
ALTER DATABASE RENAME GLOBAL TRANSACTION LOGFILE
 'org_commit_0.log', 'org_commit_1.log' TO 'new_commit_0.log',
'new_commit_1.log';
```

兼容性

标准SQL未定义ALTER DATABASE语句

参考

相关内容参考**ALTER DATABASE RENAME LOGFILE**

8.22 ALTER DATABASE RENAME LOGFILE

功能

变更数据库中的日志文件名称

语句

```
<alter database rename logfile statement> ::=
    ALTER DATABASE RENAME LOGFILE <logfile_list> TO <logfile_list>
    ;

<logfile_list> ::=
      'logfile_name'
    | <logfile_list> , 'logfile_name'
```

使用范围及访问权限

用户需要拥有ALTER DATABASE ON DATABASE权限才能执行<alter database rename logfile statement> 语句

语句规则及参数

<alter database rename logfile statement>

- 数据库应为MOUNT状态
- FROM <logfile_list>
  - 数据库中要变更的日志文件名称列表
- TO <logfile_list>
  - 数据库中要变更的日志文件名称列表
  - <logfile_list>应为已有的文件
  - 文件不存在时报错

说明

参考各语句的使用规则

使用示例

以下为把现有的日志文件'logfile.log'变更为'newlogfile.log'的示例

```
ALTER DATABASE RENAME LOGFILE 'logfile.log' TO 'newlogfile.log';
```

兼容性

标准SQL未定义ALTER DATABASE语句

参考

相关内容参考下文

- **ALTER DATABASE ADD LOGFILE**
- **ALTER DATABASE DROP LOGFILE**

8.23 ALTER DATABASE RESET LOCAL CLUSTER MEMBER

功能

将排除表空间对象的本地集群成员初始化至数据库创建时间点

语句

```
<alter database reset local cluster member statement> ::=
    ALTER DATABASE RESET LOCAL CLUSTER MEMBER
    ;
```

使用范围及访问权限

可在集群系统中执行

可在启动过程中的LOCAL OPEN阶段执行

用户需要拥有ADMINISTRATION ON DATABASE权限才能执行<alter database reset local cluster member statement>语句

说明

将排除表空间对象的本地集群成员初始化至数据库创建时间点

删除除表空间对象外的用户创建的所有对象

<alter database reset local cluster member statement>语句初始化inactive cluster member

并用于将新的集群成员加入到集群系统中

与集群系统断开连接的inactive cluster member可进行以下处理

- 可重新加入到集群系统时使用JOIN语句加入
  - **ALTER SYSTEM JOIN DATABASE**
- 无法重新加入到集群系统时使用DROP语句从集群系统中排除
  - **ALTER DATABASE DROP INACTIVE CLUSTER MEMBERS**

此时从集群系统中排除的集群成员所对应的设备可通过以下两种方法重新使用

- 方法1：重新生成本地集群成员的数据库
- 方法2：使用<alter database reset local cluster member statement>语句初始化本地集群成员

方法2比方法1更加降低重新创建表空间的费用

使用示例

以下为将从集群系统中排除的本地集群成员启动至LOCAL OPEN阶段后使用<alter database reset local cluster member statement>语句进行初始化的示例

```
gSQL> \startup nomount

Startup success


gSQL> ALTER SYSTEM MOUNT DATABASE;

System altered.


gSQL> ALTER SYSTEM OPEN LOCAL DATABASE;

System altered.

gSQL> ALTER DATABASE RESET LOCAL CLUSTER MEMBER;

Database altered.
```

兼容性

标准SQL中未定义集群的概念

参考

相关内容参考下文

- **ALTER SYSTEM JOIN DATABASE**
- **ALTER DATABASE DROP INACTIVE CLUSTER MEMBERS**

8.24 ALTER DATABASE RESTORE

功能

使用增量备份复原数据库或表空间的数据文件

语句

```
<alter database restore statement> ::=
      <database restore statement>
    | <tablespace restore statement>
    | <controlfile restore statement>
    ;

<database restore statement> ::=
    ALTER DATABASE RESTORE [ <until clause> ]

<until clause> ::=
    UNTIL CHANGE integer

<tablespace restore statement> ::=
    ALTER DATABASE RESTORE TABLESPACE tablespace_name
```

```
<controlfile restore statement> ::=

    ALTER DATABASE RESTORE CONTROLFILE FROM 'file_name'
```

使用范围及访问权限

用户需要拥有ALTER DATABASE ON DATABASE权限才能执行<alter database restore statement> 语句

语句规则及参数

<database restore statement>

使用增量备份复原数据库的数据文件

数据库应为MOUNT状态

<tablespace restore statement>

使用增量备份复原表空间的数据文件

- 数据库应为MOUNT或OPEN状态
- OPEN状态下仅可复原OFFLINE状态的表空间MOUNT状态下可复原所有ONLINE / OFFLINE状态的表空间

<controlfile restore statement>

使用'file_name'复原控制文件

- 数据库应为NOMOUNT状态
- 'file_name'建议使用绝对路径如果使用相对路径则使用<SUNDB_HOME>/wal/'file_name'

说明

使用完全备份的数据文件复原是通过OS的拷贝命令将备份的文件直接拷贝到数据文件路径的方法使用增量备份的数据文件复原仅复原被删除的数据文件或之前的数据文件

使用示例

以下为使用增量备份复原数据库的示例

```
ALTER DATABASE RESTORE;
```

以下为使用增量备份复原表空间的示例

```
ALTER DATABASE RESTORE TABLESPACE test_tbs;
```

以下为仅使用LSN小于11123的增量备份复原数据库的示例

```
ALTER DATABASE RESTORE UNTIL CHANGE 11123;
```

以下为使用controlfile.bak复原控制文件的示例

```
ALTER DATABASE RESTORE CONTROLFILE FROM 'controlfile.bak'
```

兼容性

标准SQL未定义ALTER DATABASE语句

参考

相关内容参考下文

- **ALTER DATABASE BACKUP**
- **ALTER TABLESPACE name BACKUP**
- **ALTER DATABASE RECOVER**

8.25 ALTER DATABASE SYNCHRONIZE

功能

远程同步所有表的shard和序列

语句

```
<alter database synchronize statement> ::=
    ALTER DATABASE SYNCHRONIZE
       [ <synchronize target> ]
       [ ONLINE | OFFLINE ]
       [ <shard divisor> ]
       [ <parallel clause> ]
    ;

<synchronize target> ::=
    TABLE
  | SEQUENCE
  | TABLE AND SEQUENCE
  | SEQUENCE AND TABLE

<shard divisor> ::=
```

```
    SHARD DIVISOR integer


<parallel clause> ::=
    NOPARALLEL
  | PARALLEL [ integer ]
```

使用范围及访问权限

可在集群系统中执行

用户需要拥有ALTER DATABASE ON DATABASE权限才能执行<alter database synchronize statement>语句

语句规则及参数

<synchronize target>

指定同步目标对象

- TABLE
  - 同步表对象
- SEQUENCE
  - 同步序列对象
- TABLE AND SEQUENCE或SEQUENCE AND TABLE
  - 同步表和序列对象

- 省略时默认值为TABLE AND SEQUENCE

[ONLINE | OFFLINE]

确定同步时是否允许进行DML

- ONLINE
  - 允许INSERTUPDATE和DELETE
- OFFLINE
  - 不允许INSERTUPDATE和DELETE
- 省略时默认值为ONLINE

<shard divisor>

指定shard的分区数量

- 按照分区数量划分shard然后同步到远程服务器
- 整数可以从0开始使用最大值为1000
- 省略时根据REBALANCE_SHARD_DIVISOR属性决定
- integer小于parallel integer时修改为和parallel integer相同的值

<synchronize target>中仅指定SEQUENCE时将被忽略

<parallel clause>

指定同步表时使用的线程数量

- NOPARALLEL
  - 不并行同步表
- PARALLEL [integer]
  - 并行同步表
  - 整数可以从0开始使用最大值为64
  - 省略整数时为0
  - 当整数为0时系统会决定最佳值

当<synchronize target>仅指定SEQUENCE时将被忽略

说明

同步现存的所有离线shard和序列后更改为在线与**ALTER DATABASE REBALANCE**不同即使存在inactive cluster member也可以执行

集群成员的inactive状态代表未与集群系统连接会在以下情况中发生

- 相应集群成员在运行的集群系统中出现故障时
- 在未运行相应集群成员的情况下尝试启动集群系统时

<alter database synchronize statement>对每个表执行**<alter table synchronize statement>**相当于以下查询的集合

```
ALTER TABLE t1 SYNCHRONIZE;

COMMIT;

ALTER TABLE t2 SYNCHRONIZE;
```

```
COMMIT;

ALTER TABLE t3 SYNCHRONIZE;

COMMIT;


...


ALTER TABLE tn SYNCHRONIZE;

COMMIT;
```

即使在同步特定表时发生错误<alter database synchronize statement>也不会终止而是继续同步下一个表并在显示以下警告的同时成功执行

```
gSQL> ALTER DATABASE SYNCHRONIZE;


ERR-42000(16555): of the total '5' tables, '1' tables failed to

synchronize

Database altered.
```

以上报错信息表示5个表中的1个表失败

在对错误进行适当处理后重新执行<alter database synchronize statement>语句的话则仅对失败表执行

错误详情请参考执行语句成员的系统跟踪日志(system.trc)

使用示例

以下是执行<alter database synchronize statement>语句的示例

```
gSQL> ALTER DATABASE SYNCHRONIZE;


Database altered.
```

兼容性

标准SQL中未定义集群概念

参考

相关内容请参考以下链接

- **ALTER DATABASE REBALANCE**
- **ALTER TABLE name REBALANCE**

8.26 ALTER INDEX

功能

变更索引的定义

语句

```
<alter index statement> ::=

      <alter index physical attribute statement>

    | <rename index statement>

    | <aging index statement>

    | <rebuild index statement>

    | <index coalesce statement>

    ;
```

使用范围及访问权限

用户需要拥有以下权限中的一个才能执行<alter index statement>语句

- 索引的所有者
- 索引所在的表的所有者
- 对索引所在的表有CONTROL TABLE ON TABLE

- 对索引所在的SCHEMA有 （ALTER INDEX或CONTROL SCHEMA）ON SCHEMA
- ALTER ANY INDEX ON DATABASE

语句规则及参数

<alter index physical attribute statement>

变更索引的物理属性

详细内容参考**ALTER INDEX name STORAGE**语句

<rename index statement>

变更索引的名称

详细内容参考**ALTER INDEX name RENAME TO**语句

<aging index statement>

删除索引中的空白页

详细内容参考**ALTER INDEX name AGING**语句

<rebuild index statement>

重构索引

详细内容参考**ALTER INDEX name REBUILD**语句

<index coalesce statement>

删除索引碎片

详细内容参考**ALTER INDEX name COALESCE**语句

说明

参考各详细语句的说明

使用示例

参考各详细语句的使用示例

兼容性

标准SQL未定义索引的概念

8.27 ALTER INDEX name AGING

功能

删除索引中的空白页

语句

```
<aging index statement> ::=

    ALTER INDEX index_name AGING

    ;
```

使用范围及访问权限

用户需要拥有以下权限中的一个才能执行<aging index statement>语句

- 索引的所有者
- 索引所在的表的所有者
- 对索引所在的表有CONTROL TABLE ON TABLE
- 对索引所在的SCHEMA有 （ALTER INDEX或CONTROL SCHEMA）ON SCHEMA
- ALTER ANY INDEX ON DATABASE

语句规则及参数

index_name

对象索引的名称

说明

该语句以段（segment）返回索引页中删除所有Key的页Aging分为逻辑删除与物理删除两个阶段逻辑删除是断开索引中页的连接的操作并且当删除页的最后一个Key时的SCN小于系统的agable SCN的情况下执行之后在执行物理删除在逻辑删除时的SCN小于系统的agable SCN的情况下执行

> Caution:
>
> 如果系统的agable SCN未增加则即使成功执行索引AGING语句也可能不会删除空白页

使用示例

以下为aging索引的示例

```
gSQL> select index_name, empty_blocks from user_indexes where index_name =
'T1X';
```

```
INDEX_NAME EMPTY_BLOCKS

---------- ------------

T1X                   2


1 row selected.


gSQL> alter index t1x aging;


Index altered.


gSQL> select index_name, empty_blocks from user_indexes where index_name =

'T1X';


INDEX_NAME EMPTY_BLOCKS

---------- ------------

T1X                   0


1 row selected.
```

兼容性

标准SQL未定义索引的概念

参考

相关内容参考下文

- **CREATE INDEX**
- **ALTER INDEX**
- **DROP INDEX**

8.28 ALTER INDEX name COALESCE

功能

合并索引的相邻叶页从而减少索引的使用空间

语句

```
<index coalesce statement> ::=

    ALTER INDEX index_name COALESCE

    ;
```

使用范围及访问权限

用户须满足以下条件才能执行<index coalesce statement>语句

- 索引的所有者
- 索引所属表的所有者
- 对索引所属表有CONTROL TABLE ON TABLE
- 对索引所属模式有(ALTER INDEX或CONTROL SCHEMA) ON SCHEMA
- ALTER ANY INDEX ON DATABASE

对于要在其中创建索引的表空间需要有以下权限中的一种权限

- 对相应表空间有CREATE OBJECT ON TABLESPACE
- USAGE TABLESPACE ON DATABASE

语句规则及参数

index_name

目标索引的名称

可以指定schema名称省略时使用用户的默认schema名称

说明

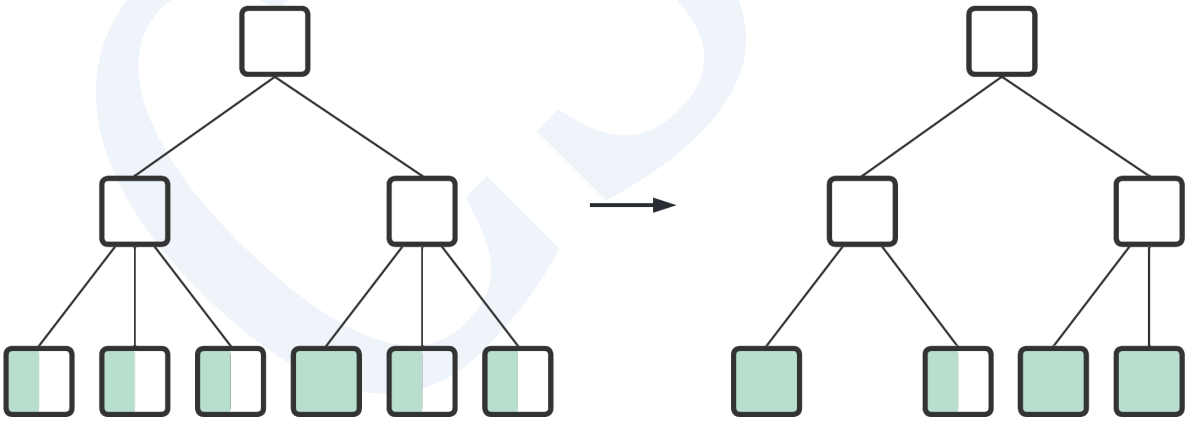


Figure 8-1 Index coalesce

- 依次扫描叶页发现可以合并的页面时进行页面合并并将已删除的页面返回给段
- 可以解决因UPDATE/ DELETE等发生的叶页碎片化问题
- 删除与无效shard相关的key解除shard sequence限制

- 因仅在邻近叶页可合并时运行所以如果碎片化程度较低可能不会有效果
- 如果索引碎片化程度高那么处理时间可能会比INDEX REBUILD要长

	INDEX REBUILD	INDEX COALESCE
更改索引属性	可以	不可以
移动表空间	可以	不可以
锁表	需要	不需要
用于执行的额外空间	需要	不需要
降低树高	可以	不可以

Table 8-2 与INDEX REBUILD比较

使用示例

```
gsql> ALTER INDEX T1X COALESCE;


Index altered.
```

兼容性

标准SQL未定义索引概念

参考

相关内容参考下文

ALTER INDEX

ALTER INDEX name REBUILD

8.29 ALTER INDEX name REBUILD

功能

重构索引

语句

```
<rebuild index statement> ::=
    ALTER INDEX index_name REBUILD
        [ ONLINE | OFFLINE ]
        [ <index attributes> [...] ]
        [ TABLESPACE tablespace_name ]
    ;

<index attributes> ::=
      <physical attribute clause>
    | STORAGE ( <segment attr clause> [...] )
    | <parallel clause>

<physical attribute clause> ::=
      PCTFREE integer
    | INITRANS integer
```

```
    | MAXTRANS integer


<segment attr clause> ::=

      INITIAL <size_clause>

    | NEXT <size_clause>

    | MINSIZE <size_clause>

    | MAXSIZE <size_clause>


<size clause> ::=

      integer [ K | M | G | T ]


<parallel clause> ::=

      NOPARALLEL

    | PARALLEL [ integer ]
```

使用范围及访问权限

执行<rebuild index statement>语句需满足以下条件

- 索引的所有者
- 索引所属的表的所有者
- 对索引所属的表有CONTROL TABLE ON TABLE
- 对索引所属的表有(ALTER INDEX或CONTROL SCHEMA) ON SCHEMA
- ALTER ANY INDEX ON DATABASE

对创建索引的表空间有以下权限中的一个

- 对该表空间有CREATE OBJECT ON TABLESPACE
- USAGE TABLESPACE ON DATABASE

语句规则及参数

index_name

对象索引的名称

可指定schema名称省略时使用用户的默认schema名称

[ONLINE | OFFLINE]

重构索引时确定该表是否允许DML

- ONLINE
  - 允许INSERTUPDATEDELETE
- OFFLINE
  - 不允许INSERTUPDATEDELETE
- 省略时默认值为ONLINE

<physical attribute clause>

定义索引的物理属性信息

- PCTFREE integer
  - 定义
    - 为了调整页内插入key引起的页分隔频率而预留的空间
  - 可使用0~99的值
  - 省略时使用原有的索引的设定值
- INITRANS integer
  - 定义
    - 表示可同时访问页的初始事务的数量
    - 访问索引的用户数量少时将INITRANS设置较低同时访问的用户多时将INITRANS设置较高
    - 必要时自动增加至设置的MAXTRANS
  - 可使用1~32的值
  - 省略时使用原有索引的设定值
- MAXTRANS integer
  - 定义
    - 表示可同时访问页的事务的最大数量
  - 可使用1~32的值
  - 省略时使用原有索引的设定值

<segment attr clause>

说明存储索引的空间的信息

- INITIAL integer
  - 定义

        - 描述创建索引时初期分配的物理空间的大小
        - 其大小align到表所属的TABLESPACE的EXTENT大小（例EXT大小为8192 bytes时'INITIAL 100'实际以8192 bytes运行）
        - 其大小（align到TABLESPACE的EXTENT的大小）应大于或等于MINEXTENTS的大小并小于或等于MAXEXTENTS的大小
    - 最小值为1最大值根据系统环境有所不同
    - 省略时使用原有索引的设定值
- NEXT integer
    - 定义
        - 在索引增加物理空间时描述要分配的物理空间的大小
        - 其大小align到表所属的TABLESPACE的EXTENT大小（例EXT大小为8192 bytes时'NEXT 100'实际以8192 bytes运行）
        - NEXT根据目前索引可使用的剩余空间的大小（在MAXEXTENTS的大小减去当前使用中的空间的大小）如下运行
            - 剩余空间的大小为0时无法再扩展空间
            - 剩余空间的大小的大于0小于NEXT时分配与剩余空间大小相同的大小
            - 剩余空间的大小大于NEXT时分配与NEXT大小相同的大小
    - 最小值为1最大值根据系统环境有所不同
    - 省略时使用原有索引的设定值
- MINSIZE integer
    - 定义
        - 索引要维持的最小空间的大小
        - 此值应小于或等于MAXSIZE的值
    - 其大小align到索引所属的 TABLESPACE的EXTENT大小
    - 最小值为1最大值根据系统环境有所不同

    - 小于两个EXTENT的大小时确定为两个EXTENT的大小
    - 省略时使用原有索引的设定值
- MAXSIZE integer
    - 定义
        - 可在索引中分配的最大空间的大小
        - 此值应大于或等于MINSIZE的值
    - 其大小align到索引所属的TABLESPACE的EXTENT大小
    - 最小值为1最大值根据系统环境有所不同
    - 省略时使用原有索引的设定值

<size clause>

指定文件的byte大小（未指定单位时使用bytes）

- K: Kilobytes
- M: Megabytes
- G: Gigabytes
- T: Terabytes

NOPARALLEL | PARALLEL [integer]

指定重构索引的过程中使用的thread的数量

- NOPARALLEL
    - 不并列重构索引
- PARALLEL [integer]

    - 并列重构索引
    - 省略ingeger或指定为0时遵循INDEX_BUILD_PARALLEL_FACTOR参数
    - integer可从0开始使用最大值为64
    - 如果integer或参数的值为0则由系统决定最优值
- 未指定时默认值为NOPARALLEL

TABLESPACE tablespace_name

指定重构索引的表空间的名称

- 指定tablespace_name时
    - tablespace_name为data tablespace时重建为LOGGING索引
    - tablespace_name为temporary tablespace 或nologging tablespace时重建为NOLOGGING索引
- 省略TABLESPACE 子句时设置为原有索引的表空间

说明

- 删除索引碎片化
    - 在索引频繁执行DML时索引页可能产生碎片化相比有效的数据Tree过于大时索引的容量增加而性能下降此时重构索引可解决索引页的碎片化缩小索引容量并恢复索引的性能
- 变更索引的表空间
    - 可变更原先创建的索引的表空间
    - 但要根据表空间的TEMPORARY与否设置恰当的logging与否

- 变更索引的logging设置
  - 要变更为LOGGING时应在TABLESPACE选项指定数据表空间
  - 要变更为NOLOGGING时应在TABLESPACE选项指定temporary tablespace或nologging tablespace
- 删除无效的shard相关的key
  - 更改shard时与更改前shard关联的key可能会留在索引中如果不将其删除积累起来可能会发生shard sequence exceed错误这个问题在频繁更改shard的情况下发生可以通过重构索引来解决

使用示例

以下为变更索引的logging设置和表空间的示例

```
gsql> SELECT INDEX_NAME, TABLESPACE_NAME FROM INDEXES AS IDX, TABLESPACES
AS TBS WHERE IDX.TABLESPACE_ID = TBS.TABLESPACE_ID AND IDX.INDEX_NAME =
'T1X';


INDEX_NAME TABLESPACE_NAME
---------- ---------------
T1X        MEM_TEMP_TBS


1 row selected.


gsql> ALTER INDEX T1X REBUILD TABLESPACE MEM_DATA_TBS;
```

```
SELECT INDEX_NAME, TABLESPACE_NAME FROM INDEXES AS IDX, TABLESPACES AS TBS
WHERE IDX.TABLESPACE_ID = TBS.TABLESPACE_ID AND IDX.INDEX_NAME = 'T1X';


INDEX_NAME TABLESPACE_NAME
---------- ---------------
T1X        MEM_DATA_TBS


1 row selected.
```

兼容性

标准SQL未定义索引的概念

参考

相关内容参考如下

- **CREATE INDEX**
- **ALTER INDEX**
- **DROP INDEX**

8.30 ALTER INDEX name RENAME TO

功能

变更索引的名称

语句

```
<rename index statement> ::=
    ALTER INDEX index_name
        RENAME TO new_index_name
    ;
```

使用范围及访问权限

用户需要有以下权限中的一个才能执行<rename index statement>语句

- 索引的所有者
- 索引所在的表的所有者
- 对索引所在的表有CONTROL TABLE ON TABLE
- 对索引所在的SCHEMA有 （ALTER INDEX或CONTROL SCHEMA）ON SCHEMA
- ALTER ANY INDEX ON DATABASE

语句规则及参数

index_name

对象索引的名称

无法描述schema的名称并且拥有与现有索引相同的schema名称

new_index_name

新的索引名并且应为schema中唯一的索引名

说明

参考各语句的使用规则

使用示例

以下为变更索引名的示例

```
gSQL> ALTER INDEX t1_idx1 RENAME TO idx_t1_id;


Index altered.
```

兼容性

标准SQL未定义索引的概念

参考

相关内容参考下文

- **CREATE INDEX**
- **ALTER INDEX**
- **DROP INDEX**

8.31 ALTER INDEX name STORAGE

功能

变更索引的物理属性

语句

```
<alter index physical attribute statement> ::=
    ALTER INDEX index_name
    | <physical attribute clause>
    | [ STORAGE ( <segment attr clause> [...] ) ]
    ;

<physical attribute clause> ::=
      PCTFREE integer
    | INITRANS integer
    | MAXTRANS integer

<segment attr clause> ::=
      INITIAL <size_clause>
    | NEXT <size_clause>
    | MINSIZE <size_clause>
```

```
    | MAXSIZE <size_clause>


<size clause> ::=

      integer [ K | M | G | T ]
```

使用范围及访问权限

用户需要有以下权限中的一个才能执行<alter index physical attribute statement>语句

- 索引的所有者
- 索引所在的表的所有者
- 对索引所在的表有CONTROL TABLE ON TABLE
- 对索引所在的SCHEMA有 （ALTER INDEX或CONTROL SCHEMA）ON SCHEMA
- ALTER ANY INDEX ON DATABASE

语句规则及参数

index_name

对象索引的名称

<physical attribute clause>

定义索引的物理属性信息

- PCTFREE integer
  - 定义
    - 为调节在页内插入key而引起的页分割频率预留的空间
    - 仅应用于自下向上（bottom-up）构建索引时
  - 可使用0~99之间的值
  - 省略时默认值使用DEFAULT_INDEX_PCTFREE property中设置的值
- INITRANS integer
  - 定义
    - 页上可同时访问的事务的初始数量
    - 当访问索引的用户数较少时INITRANS设置为较低值同时访问索引的用户数较多时设置为较高值
    - 必要时自动增加至设置的MAXTRANS
  - 可使用1~32之间的值
  - 省略时默认值为4
- MAXTRANS integer
  - 定义
    - 页上可同时访问的事务的最大数量
  - 可使用1~32之间的值
  - 省略时默认值为8

<segment attr clause>

描述有关存储索引的空间的信息

- INITIAL integer

- 定义
  - 描述创建索引时要分配的初始物理空间的大小
  - 此大小align到表所在的表空间的EXTENT大小后使用（例: EXT大小为8192 bytes时'INITIAL 100'的实际大小为8192 bytes）
  - 此大小（align到表空间EXTENT大小的大小）应大于或等于MINEXTENTS的大小并小于或等于MAXEXTENTS的大小
  - 仅应用于自下向上（bottom-up）构建索引时
- 最小值为1最大值根据系统环境有所不同
- 省略时默认值为表所在的表空间的一个EXTENT的大小

- NEXT integer
  - 定义
    - 描述添加索引的物理空间时要分配的物理空间的大小
    - 此大小align到表所在的表空间的EXTENT大小后使用（例: EXT大小为8192 bytes时'NEXT 100'的实际大小为8192 bytes）
    - NEXT根据当前索引可使用的剩余空间的大小（MAXEXTENTS的大小减去当前使用中的空间的大小）如下进行操作
      - 剩余空间大小为0时无法再扩大空间
      - 剩余空间大小大于0小于NEXT时分配与剩余空间相同的大小
      - 剩余空间大小大于NEXT时分配与NEXT大小相同的大小
  - 最小值为1最大值根据系统环境有所不同
  - 省略时默认值为表所在的表空间的一个EXTENT的大小

- MINSIZE integer
  - 定义
    - 索引需保留的最小空间的大小
    - 该值应小于或等于MAXSIZE的值

    - 此大小align到索引所在的表空间的EXTENT大小后使用
    - 最小值为1最大值根据系统环境有所不同
    - 小于两个EXTENT的大小时确定为两个EXTENT的大小
    - 省略时默认值为两个EXTENT的大小
- MAXSIZE integer
    - 定义
        - 可在索引中分配的最大空间的大小
        - 该值应大于或等于MINSIZE的值
    - 此大小align到索引所在的表空间的EXTENT大小后使用
    - 最小值为1最大值根据系统环境发生变化
    - 省略时默认值为32 terabyte（35,184,372,088,832）
    - 即使指定大于32 terabyte的值也会修改成32 terabyte进行设置

<size clause>

指定文件的字节大小（未描述单位时为bytes）

- K: kilobytes
- M: megabytes
- G: gigabytes
- T: terabytes

说明

参考各语句的使用规则

使用示例

以下为变更索引的物理属性的示例

```
gSQL> ALTER INDEX idx_t1_id PCTFREE 10 INITRANS 4 MAXTRANS 8;


Index altered.
```

兼容性

标准SQL未定义索引的概念

参考

相关内容参考下文

- **CREATE INDEX**
- **ALTER INDEX**
- **DROP INDEX**

8.32 ALTER PROFILE

功能

变更密码管理方法

语句

```
<alter profile statement> ::=
    ALTER PROFILE profile_name LIMIT
    { <password_parameters>, ...}
    ;

<password parameters> ::=
      FAILED_LOGIN_ATTEMPTS { integer | UNLIMITED | DEFAULT }
    | PASSWORD_LOCK_TIME  { password_parameter_number_interval | UNLIMITED
| DEFAULT }
    | PASSWORD_LIFE_TIME  { password_parameter_number_interval | UNLIMITED
| DEFAULT }
    | PASSWORD_GRACE_TIME { password_parameter_number_interval | UNLIMITED
| DEFAULT }
    | PASSWORD_REUSE_MAX  { integer | UNLIMITED | DEFAULT }
    | PASSWORD_REUSE_TIME { password_parameter_number_interval | UNLIMITED
```

```
| DEFAULT }

    | PASSWORD_VERIFY_FUNCTION { <verify_policy> | NULL | DEFAULT }


<verify_policy> ::=

      KISA_VERIFY_FUNCTION

    | ORA12C_VERIFY_FUNCTION

    | ORA12C_STRONG_VERIFY_FUNCTION

    | VERIFY_FUNCTION_11G

    | VERIFY_FUNCTION


<password_parameter_number_interval> ::=

   integer

 | integer / integer
```

使用范围及访问权限

为了执行<alter profile statement>语句用户需要有ALTER PROFILE ON DATABASE权限

语句规则及参数

profile_name

要变更的profile名称

FAILED_LOGIN_ATTEMPTS

设置允许连续登陆失败的次数

详细内容参考**CREATE PROFILE**

PASSWORD_LOCK_TIME

设置连续登陆失败后账号被锁定的时间（day）

详细内容参考**CREATE PROFILE**

PASSWORD_LIFE_TIME

设置密码的有效期（day）

详细内容参考**CREATE PROFILE**

PASSWORD_GRACE_TIME

设置PASSWORD_LIFE_TIME之后继续可以使用当前密码的宽限期

详细内容参考**CREATE PROFILE**

PASSWORD_REUSE_MAX

指定尝试重新使用旧密码时无法重新使用的最近密码数量

详细内容参考**CREATE PROFILE**

PASSWORD_REUSE_TIME

指定尝试重新使用旧密码所需的经过时间

详细内容参考**CREATE PROFILE**

PASSWORD_VERIFY_FUNCTION

设置验证密码复杂度的方法

详细内容参考**CREATE PROFILE**

使用示例

以下为为了控制账号锁定而变更profile的示例

```
gSQL> ALTER PROFILE prof1 LIMIT
        FAILED_LOGIN_ATTEMPTS 3
        PASSWORD_LOCK_TIME 3;

Profile altered.

gSQL> COMMIT;

Commit complete.
```

以下为为了控制密码到期而变更profile的示例

```
gSQL> ALTER PROFILE prof1 LIMIT
        PASSWORD_LIFE_TIME 90
        PASSWORD_GRACE_TIME 7;

Profile altered.

gSQL> COMMIT;

Commit complete.
```

以下为为了控制是否重新使用密码而变更profile的示例

```
gSQL> ALTER PROFILE prof1 LIMIT
        PASSWORD_REUSE_MAX  DEFAULT
        PASSWORD_REUSE_TIME DEFAULT;

Profile altered.

gSQL> COMMIT;

Commit complete.
```

以下为为了检查密码复杂度而变更profile的示例

```
gSQL> ALTER PROFILE prof1 LIMIT
        PASSWORD_VERIFY_FUNCTION KISA_VERIFY_FUNCTION;
```

```
Profile altered.

gSQL> COMMIT;

Commit complete.
```

兼容性

标准SQL未定义profile的概念

参考

相关内容参考**DROP PROFILE**

8.33 ALTER SEQUENCE

功能

变更序列（sequence）

语句

```
<alter sequence generator statement> ::=
    ALTER SEQUENCE sequence_name <alter sequence generator options>
    ;

<alter sequence generator options> ::=
    <alter sequence generator option> [, ...]

<alter sequence generator option> ::=
      <alter sequence generator restart option>
    | <basic sequence generator option>

<alter sequence generator restart option> ::=
    RESTART [ WITH integer ]

<basic sequence generator option> ::=
```

```
      <sequence generator increment by option>
    | <sequence generator maxvalue option>
    | <sequence generator minvalue option>
    | <sequence generator cycle option>
    | <sequence generator cache option>



<sequence generator increment by option> ::=
    INCREMENT BY integer


<sequence generator maxvalue option> ::=
      MAXVALUE integer
    | (NO MAXVALUE | NOMAXVALUE)


<sequence generator minvalue option> ::=
      MINVALUE integer
    | (NO MINVALUE | NOMINVALUE)


<sequence generator cycle option> ::=
      CYCLE
    | (NO CYCLE | NOCYCLE)


<sequence generator cache option> ::=
      CACHE integer
    | (NO CACHE | NOCACHE)
```

使用范围及访问权限

用户需要拥有以下权限中的一个才能执行<alter sequence generator statement>语句

- 相关序列的所有者
- 对序列所在的SCHEMA有（ALTER SEQUENCE或CONTROL SCHEMA）ON SCHEMA
- ALTER ANY SEQUENCE ON DATABASE

语句规则及参数

sequence_name

要变更的序列的名称

与schema_name.sequence_name相同可定义序列所在的SCHEMA省略schema_name时使用执行语句的用户的默认SCHEMA名

<alter sequence generator restart option>

设置序列的NEXT VALUE

但不变更**CREATE SEQUENCE** 语句中定义的START WITH值

- RESTART
  - 未指定值时<sequence generator definition>定义的START WITH值设置NEXT VALUE
- RESTART WITH integer

- integer值设置为NEXT VALUE
- integer值应为MINVALUE与MAXVALUE之间的值

未指定<alter sequence generator restart option>子句时以序列的当前值为准变更序列属性

<sequence generator increment by option>

变更序列号的间隔

有以下约束及特点

- 可以使用正数或负数但不能使用0
- 间隔的绝对值应小于MINVALUE与MAXVALUE之差
- 正数时为升序序列负数时为降序系列

<sequence generator maxvalue option>

变更可创建为序列的最大值

但MAXVALUE值不能小于序列的当前值

- MAXVALUE integer
  - 最大值的范围为64bit整数的最小值(−9,223,372,036,854,775,808)到64bit整数的最大值(+9,223,372,036,854,775,807)
  - 应大于或等于START WITH的值并大于MINVALUE的值
- NO MAXVALUE | NOMAXVALUE
  - 如下变更最大值
    - 升序序列时为64 bit整数的最大值(+9,223,372,036,854,775,807)

        - 降序序列时为-1
        - NO MAXVALUE (标准SQL)与NOMAXVALUE是意义相同的保留字因此可使用两个中的任何一个

<sequence generator minvalue option>

变更可创建为序列的最小值

但MINVALUE不能大于序列的当前值

- MINVALUE integer
    - 最小值的范围为64bit整数的最小值(−9,223,372,036,854,775,808)到64bit整数的最大值(+9,223,372,036,854,775,807)
    - 应小于或等于START WITH的值并小于MAXVALUE的值
- NO MINVALUE | NOMINVALUE
    - 如下变更最小值
        - 升序序列时为1
        - 降序序列时为64bit整数的最小值(−9,223,372,036,854,775,808)
    - NO MINVALUE (标准SQL)与NOMINVALUE是意义相同的保留字因此可使用两个中的任何一个

<sequence generator cycle option>

当序列的值达到最大值或最小值时更改是否继续创建值

- CYCLE
    - 升序序列达到最大值时从最小值开始重新创建

- 降序序列达到最小值时从最大值开始重新创建

- NO CYCLE | NOCYCLE
  - 达到最大值或最小值时无法创建序列值
  - NO CYCLE (标准SQL)与NOCYCLE是意义相同的保留字因此可使用两个中的任何一个

<sequence generator cache option>

为了快速访问序列在内存上预先定义要加载的序列值的数量

重启数据库时丢失内存中加载的序列值并从加载后的值开始

- CACHE integer
  - CACHE值应大于或等于2
  - 有CYCLE时CACHE值应小于CYCLE长度
    - CYCLE长度：CEIL(MAXVALUE - MINVALUE) / ABS(INCREMENT)
- NO CACHE | NOCACHE
  - 不在内存上预加载序列值

说明

无法变更**CREATE SEQUENCE** 语句中定义的序列属性中的START WITH的值

需要先执行**DROP SEQUENCE** 后执行**CREATE SEQUENCE**语句重新创建序列才能变更START WITH属性

使用示例

以下为使用RESTART选项重新开始序列值并用其重新赋予ID值的示例

```
gSQL> SELECT id, name FROM t1 ORDER BY 1;


 ID NAME
--- ------
 10 leekmo
 42 mkkim
 51 jhkim
172 ehpark


4 rows selected.



gSQL> ALTER SEQUENCE seq1 RESTART;


Sequence altered.



gSQL> UPDATE t1 SET id = seq1.NEXTVAL;


4 rows updated.
```

```
gSQL> SELECT id, name FROM t1 ORDER BY 1;


ID NAME
-- ------
 1 leekmo
 2 mkkim
 3 jhkim
 4 ehpark


4 rows selected.
```

兼容性

标准SQL中未定义CACHE/ NO CACHE

Feature ID	说明	是否支持
T176	Sequence generator support	O
T177	Sequence generator support: simple restart option	O

Table 8-3 标准SQL兼容性

参考

相关内容参考下文

- **CREATE SEQUENCE**
- **DROP SEQUENCE**

8.34 ALTER SESSION CLEANUP GLOBAL TEMPORARY SEGMENT POOL;

功能

为了在会话中重新使用将catching的所有空间返回到相应表空间

语句

```
<alter session cleanup global temporary segment pool statement> ::=
    ALTER SESSION CLEANUP GLOBAL TEMPORARY SEGMENT POOL
    ;
```

说明

在执行中的会话中仅cleanup segment cache的segment

使用示例

以下为cleanup会话segment cache的示例

```
gSQL> ALTER SESSION CLEANUP GLOBAL TEMPORARY SEGMENT POOL;
```

```
Session altered.
```

兼容性

标准SQL未定义global temporary table, global temporary index的segment cache概念

参考

相关内容参考**Global Temporary Table**

8.35 ALTER SESSION SET property_name

功能

设置会话的参数值

语句

```
<alter session set statement> ::=

    ALTER SESSION SET <property name> { = <property value> | TO DEFAULT }

    ;
```

语句规则及参数

<property name>

要设置的参数名

详细内容参考Database Administration用户手册的服务器属性章节

<property value>

要设置的参数值

TO DEFAULT

将会话参数值设置为系统参数值

说明

各参数的详细说明参考Database Administration用户手册的服务器属性章节

使用示例

以下为设置HINT_ERROR属性并在hint语句中存在错误时报错的示例

```
gSQL> ALTER SESSION SET HINT_ERROR = ON;


Session altered.


gSQL> SELECT /*+ INDEX( t1, invalid_index ) */ name FROM t1 WHERE id = 1;


ERR-42000(16058): not applicable hint :
SELECT /*+ INDEX( t1, invalid_index ) */ name FROM t1 WHERE id = 1
           *
ERROR at line 1:
```

以下为将会话参数值设置为系统参数值的示例

```
gSQL> ALTER SESSION SET HINT_ERROR TO DEFAULT;


Session altered.
```

兼容性

标准SQL未定义会话参数的概念

参考

相关内容参考**ALTER SESSION SET property_name**

8.36 ALTER SYSTEM CHECKPOINT

功能

执行CHECKPOINT

语句

```
<alter system checkpoint statement> ::=

    ALTER SYSTEM CHECKPOINT

    [ AT <domain name> ]

    ;
```

使用范围及访问权限

用户需要拥有ALTER SYSTEM ON DATABASE权限才能执行<alter system checkpoint statement>语句

语句规则及参数

<alter system checkpoint statement>

CHECKPOINT是保障在磁盘中记录所有已提交的事务所变更的数据的操作

- 数据库应为OPEN状态
- 数据库应为TDS模式
- 正在执行完全备份时变更的页不记录到数据文件仅在磁盘记录重做日志与控制文件此时如果服务器非正常结束则需执行介质恢复

<domain name>

- 执行语句的成员或群组名
- 未指定时在所有群组中执行

说明

CHECKPOINT操作将已提交的事务变更的所有内容记录到磁盘因此系统故障时可迅速进行恢复

使用示例

以下为执行CHECKPOINT的示例

```
ALTER SYSTEM CHECKPOINT;
```

兼容性

标准SQL未定义CHECKPOINT的概念

8.37 ALTER SYSTEM CLEANUP BUFFER_CACHE

功能

清空在buffer cache可free的所有buffer page

语句

```
<alter system cleanup buffer_cache statement> ::=
    ALTER SYSTEM CLEANUP BUFFER_CACHE
    [ AT <domain name> ]
    ;
```

使用范围及访问权限

用户需要拥有ALTER SYSTEM ON DATABASE权限才能执行<alter system cleanup buffer_cache statement>语句

语句规则及参数

<alter system cleanup buffer_cache statement>

没有该语句规则或参数

<domain name>

执行语句的成员或群组的名称

未指定时在所有群组执行

说明

Flush缓存到缓冲区的可free的所有buffer page并free

> Caution:
>
> 应该用于性能测试前清空buffer cache
>
> 如果在运行过程中在服务器使用则会对性能造成严重影响

使用示例

以下为执行CLEANUP BUFFER_CACHE的示例

```
ALTER SYSTEM CLEANUP BUFFER_CACHE;
```

兼容性

标准SQL未定义CLEANUP BUFFER_CACHE的概念

8.38 ALTER SYSTEM CLEANUP PLAN

功能

清理所有SQL计划

语句

```
<alter system cleanup plan statement> ::=

    ALTER SYSTEM CLEANUP PLAN

    [ AT <domain name> ]

    ;
```

使用范围及访问权限

用户需要拥有ALTER SYSTEM ON DATABASE权限才能执行<alter system cleanup plan statement>语句

语句规则及参数

<alter system cleanup plan statement>

无对应语句规则或参数

<domain name>

执行语句的成员或群组的名称

未指定时在所有群组中执行

说明

清理所有缓存的SQL计划

但是不清理V $ SQL_CACHE.REF_COUNT大于0的计划（在准备好的语句中引用的计划）

使用示例

以下为执行CLEANUP PLAN的示例

```
ALTER SYSTEM CLEANUP PLAN;
```

兼容性

标准SQL未定义CLEANUP PLAN的概念

8.39 ALTER SYSTEM IRRECOVERABLE CLUSTER MEMBER

功能

指定不可恢复的集群成员

语句

```
<alter system irrecoverable cluster member statement> ::=
    ALTER SYSTEM IRRECOVERABLE CLUSTER MEMBER <domain name>
    ;
```

使用范围及访问权限

用户需要拥有ALTER SYSTEM ON DATABASE权限才能执行<alter system irrecoverable cluster member statement>语句

语句规则及参数

<alter system irrecoverable cluster member statement>

无对应语句规则或参数

<domain name>

不可恢复的成员名

无法将群组内的所有成员指定为不可恢复的成员

说明

用于由于不可恢复的成员而导致集群重启失败时则除该成员外进行重启系统系统成功重启后必须使用**ALTER DATABASE DROP INACTIVE CLUSTER MEMBERS**语句删除该成员

使用示例

以下为执行IRRECOVERABLE CLUSTER MEMBER的示例

```
gSQL> ALTER SYSTEM IRRECOVERABLE CLUSTER MEMBER g1n1;
```

兼容性

标准SQL未定义IRRECOVERABLE CLUSTER MEMBER的概念

8.40 ALTER SYSTEM JOIN DATABASE

功能

将禁用的特定集群成员重新包含到集群系统中

语句

```
<alter system join database statement> ::=

    ALTER SYSTEM JOIN DATABASE

    ;
```

使用范围及访问权限

可在集群系统中执行

用户需要拥有ADMINISTRATION ON DATABASE权限才能执行<alter system join database statement>语句

说明

集群成员的inactive状态为未与集群系统连接的状态发生在以下情况中

- 在运行中的集群系统中该集群成员发生故障时
- 不启动包含在集群系统中的集群成员而尝试启动集群系统时

在特定集群成员为inactive的状态下通过以下步骤可将该成员包含在集群系统中

- 将未启动的集群成员启动至local open阶段

```
$ gsql sys gliese --as sysdba --dsn=G3N2
gSQL> \startup
```

- 通过<alter system join database statement>语句将其包含在集群系统中

```
$ gsql sys gliese --as sysdba --dsn=G3N2
gSQL> ALTER SYSTEM JOIN DATABASE;
```

不shutdown以local open启动的inactive cluster member并将其参与到集群系统时使用<alter system join database statement>语句

为了使Inactive cluster member重新参与到集群系统中集群系统与Inactive cluster member的数据库状态应保持一致

在集群系统中完成变更数据库的事务后Inactive cluster member无法重新再参与到集群系统中

有多个Inactive cluster member时为了能正常运行集群系统需通过以下步骤整理Inactive cluster member

1. JOIN可参与到集群系统的inactive cluster member

```
gSQL> ALTER SYSTEM JOIN DATABASE;
```

2. DROP不可参与到集群系统的inactive cluster member

```
gSQL> ALTER DATABASE DROP INACTIVE CLUSTER MEMBERS;
```

执行<alter database drop inactive cluster member statement>语句将删除集群系统中的所有inactive cluster member因此在DROP之前应将可参与到集群系统中的所有inactive cluster member包含在集群系统中

使用示例

```
gSQL> ALTER SYSTEM JOIN DATABASE;
```

兼容性

标准SQL中未定义集群的概念

参考

相关内容参考**ALTER DATABASE DROP INACTIVE CLUSTER MEMBERS**

8.41 ALTER SYSTEM [KILL | DISCONNECT] SESSION

功能

结束会话

语句

```
<alter system end session statement> ::=

      ALTER SYSTEM DISCONNECT SESSION [<member_position>,] <session_id>,

          <serial#> [<disconnect_option>] [AT <domain_name>]

    |  ALTER SYSTEM KILL SESSION [<member_position>,]

           <session_id>, <serial#>  [AT <domain_name>]    ;


<disconnect_option> ::=

      POST_TRANSACTION

    | IMMEDIATE
```

使用范围及访问权限

用户需要拥有ALTER SYSTEM ON DATABASE权限才能执行<alter system end session statement>

语句

语句规则及参数

<member_position>

在集群环境中要断开连接/终止的会话的成员位置

<session_id>

会话的ID

<serial#>

会话的序列号（SERIAL NUMBER）

<disconnect_option>

- POST_TRANSACTION：事务结束后结束会话
- IMMEDIATE：不等待事务结束立即结束会话

不使用<disconnect_option>时默认为IMMEDIATE

<domain name>

执行语句的成员或群组名

未指定时在所有群组中执行

说明

DISCONNECT SESSION可以指定POST_TRANSACTION与IMMEDIATE选项POST_TRANSACTION等待执行中的事务结束后结束会话IMMEDIATE不等待事务结束立即结束会话

KILL SESSION终止系统上残留而没有对应进程的非正常会话

使用示例

```
gSQL> SELECT USER_NAME, SESSION_ID, SERIAL_NO, SESSION_STATUS,
PROGRAM_NAME FROM V$SESSION WHERE USER_NAME = 'TEST';


USER_NAME SESSION_ID SERIAL_NO SESSION_STATUS PROGRAM_NAME
--------- ---------- --------- -------------- ------------
TEST              62        49 CONNECTED      gsql
TEST              65       109 CONNECTED      gsqlnet
TEST              66       130 CONNECTED      gsql


3 rows selected.
```

```
gSQL> ALTER SYSTEM DISCONNECT SESSION 65, 109;

System altered.
```

兼容性

标准SQL中未定义

8.42 ALTER SYSTEM {MOUNT | OPEN} DATABASE

功能

将数据库变更为MOUNT状态或可提供服务的状态

语句

```
<alter system database statement> ::=
    ALTER SYSTEM <alter system database clause>
    ;

<alter system database clause> ::=
      MOUNT DATABASE
    | OPEN [ <database_scope> ] DATABASE [ <open_database_option> ]

<open_database_option> ::=
      NORESETLOGS
    | RESETLOGS

<database_scope> ::=
      LOCAL
```

| GLOBAL

使用范围及访问权限

用户需要拥有ADMINISTRATION ON DATABASE权限才能执行<alter system database statement>语句

语句规则及参数

<alter system database clause>

- MOUNT DATATABASE
  - 将数据库变更为MOUNT状态
- OPEN DATABASE
  - 将数据库变更为可提供服务的状态

<open database option>

- RESETLOGS / NORESETLOGS
  - 选择恢复数据库后是否保留在线重做日志
  - NORESETLOGS保留现有的重做日志相反RESETLOGS将其初始化
  - 不完全恢复数据库时必须指定RESETLOGS
  - 省略时默认为NORESETLOGS

<database_scope>

- LOCAL
  - 将LOCAL区域服务器启动到OPEN阶段
- GLOBAL
  - GLOBAL区域即将所有服务器启动到OPEN阶段
- Cluster环境下省略时以GLOBAL启动

使用示例

以下为初始化在线重做日志的示例

```
ALTER SYSTEM OPEN DATABASE RESETLOGS;
```

兼容性

标准SQL未定义数据库的MOUNT或OPEN的概念

参考

相关内容参考 **ALTER DATABASE RECOVER**

8.43 ALTER SYSTEM RECONNECT GLOBAL CONNECTION

功能

设置是否重新连接通过GLOBAL CONNECTION形式连接的会话

语句

```
<alter system reconnect global connection statement> ::=
    ALTER SYSTEM RECONNECT GLOBAL CONNECTION
    ;
```

使用范围及访问权限

用户需要拥有ALTER SYSTEM ON DATABASE权限才能执行<alter system reconnect global connection statement>语句

说明

GLOBAL CONNECTION通过对比在初始连接时从服务器获取的系统对象的SCN与当前服务器的系统对象的SCN来决定是否重新连接客户端该语句使系统对象的SCN上升引起客户端重新连接

执行该语句后客户端并非立即重新连接当客户端向服务器执行命令时通过比较SCN重新连接如果从客户端到所有成员的连接有效则不会尝试重新连接

所有示例

以下为执行该语句的示例

```
gSQL> ALTER SYSTEM RECONNECT GLOBAL CONNECTION;


System altered.
```

兼容性

标准SQL未定义GLOBAL CONNECTION的概念

8.44 ALTER SYSTEM RESET property_name

功能

在参数文件里删除参数值

语句

```
<alter system reset statement> ::=

    ALTER SYSTEM { RESET | UNSET } <property name>

        [ SCOPE = { FILE | SPFILE } ]

        [ AT <domain name>]

    ;
```

使用范围及访问权限

用户需要拥有ALTER SYSTEM ON DATABASE权限才能执行<alter system reset statement>语句

语句规则及参数

{ RESET | UNSET }

RESET与UNSET为相同意义的保留字所以两个均可以使用

<property name>

要删除的参数名

详细内容参考Database Administration用户手册的服务器属性章节

[SCOPE = { FILE | SPFILE }]

在参数文件里删除因此只能使用SCOPE=FILE/SPFILE

- SCOPE = FILE
  - FILE与SPFILE为相同意义的保留字所以两个均可以使用
  - 在文件里删除参数不应用于当前状态
  - 数据库重启时应用变更事项

未指定SCOPE子句时默认值为SCOPE = FILE

<domain name>

执行语句的成员或群组名

未指定时在所有群组中执行

说明

使用SCOPE=FILE/SPFILE变更参数时变更的值存储于参数文件并在重启数据库时反映其变更值

执行RESET时从参数文件中删除变更的参数值并在重启数据库时使用默认值

使用示例

以下为使用SCOPE=FILE变更参数的示例

```
gSQL> ALTER SYSTEM SET PROCESS_MAX_COUNT=128 SCOPE=FILE;

System altered.
```

以下为删除上述更改的参数的示例

```
gSQL> ALTER SYSTEM RESET PROCESS_MAX_COUNT SCOPE=FILE;

System altered.

gSQL> ALTER SYSTEM RESET PROCESS_MAX_COUNT SCOPE=SPFILE;

System altered.
```

```
gSQL> ALTER SYSTEM RESET PROCESS_MAX_COUNT;

System altered.

gSQL> ALTER SYSTEM UNSET PROCESS_MAX_COUNT;

System altered.
```

兼容性

标准SQL未定义系统参数概念

参考

相关内容参考**ALTER SYSTEM SET property_name**

8.45 ALTER SYSTEM SET property_name

功能

设置系统参数值

语句

```
<alter system set statement> ::=

    ALTER SYSTEM SET <property name> { = <property value> | TO DEFAULT }

        [ DEFERRED ]

        [ SCOPE = [ MEMORY | { FILE | SPFILE } | BOTH ] ]

    [AT <domain name>]

    ;
```

使用范围及访问权限

用户需要拥有ALTER SYSTEM ON DATABASE权限才能执行<alter system set statement>语句

语句规则及参数

<property name>

要设置的参数名

详细内容参考Database Administration用户手册的服务器属性章节

<property value>

要设置的参数值

TO DEFAULT

将系统参数值设置为启动系统时的初始值

[DEFERRED]

定义应用变更参数的时间点

- DEFERRED
  - 不影响当前会话应用于新生成的会话
  - 参数的ISSYS_MODIFIABLE属性值为IMMEDIATE/DEFERRED时可应用必须明确指定
  - 参数的ISSYS_MODIFIABLE属性值为FALSE时不可用

参数的ISSYS_MODIFIABLE属性值为IMMEDIATE时如果不指定DEFERRED则立即应用于所有会话

[SCOPE = [MEMORY | { FILE | SPFILE } | BOTH]]

定义系统参数变更的影响范围

- SCOPE = MEMORY
  - 变更事项仅应用于当前状态重启数据库时该值将消失
- SCOPE = FILE
  - FILE与SPFILE为相同意义的保留字所以两个均可以使用
  - 变更内容存储于文件不应用于当前状态
  - 重启数据库时应用变更值
- SCOPE = BOTH
  - 变更内容存储于文件并同时应用于当前状态

未指定SCOPE子句时默认值为SCOPE = MEMORY

参数的ISSYS_MODIFIABLE属性值为FALSE时必须指定为SCOPE=FILE/SPFILE

<domain name>

执行语句的成员或群组名

未指定时在所有群组中执行

说明

各参数的详细说明参考Database Administration用户手册的**服务器属性**章节

使用示例

以下为变更ISSYS_MODIFIABLE属性为DEFERRED的参数的示例

```
gSQL> ALTER SYSTEM SET SPIN_COUNT=1000;

ERR-22000(13019): Specified property cannot be modified.(SPIN_COUNT)

gSQL> ALTER SYSTEM SET SPIN_COUNT=1000 DEFERRED;

System altered.
```

以下为变更ISSYS_MODIFIABLE属性为FALSE的参数的示例

```
gSQL> ALTER SYSTEM SET PROCESS_MAX_COUNT=128;

ERR-22000(13018): Specified property cannot be modified with this SCOPE
option.(PROCESS_MAX_COUNT)

gSQL> ALTER SYSTEM SET PROCESS_MAX_COUNT=128 SCOPE=FILE;

System altered.
```

以下为将变更的属性更改为连接会话时的默认值的示例

```
gSQL> ALTER SYSTEM SET TRANSACTION_COMMIT_WRITE_MODE=0;
```

```
System altered.

gSQL> ALTER SYSTEM SET TRANSACTION_COMMIT_WRITE_MODE TO DEFAULT;

System altered.

gSQL> ALTER SYSTEM SET TRANSACTION_COMMIT_WRITE_MODE TO DEFAULT DEFERRED;

System altered.
```

兼容性

标准SQL未定义系统参数的概念

参考

相关内容参考**ALTER SYSTEM RESET property_name**

8.46 ALTER SYSTEM SWITCH LOGFILE

功能

将数据库中CURRENT状态的日志文件变更为ACTIVE状态

语句

```
<alter system switch logfile statement> ::=

    ALTER SYSTEM SWITCH LOGFILE

    [ AT <domain name> ]

    ;
```

使用范围及访问权限

用户需要拥有ALTER SYSTEM ON DATABASE权限才能执行<alter system switch logfile statement>语句

语句规则及参数

<alter system switch logfile statement>

数据库应为MOUNT或OPEN状态

<domain name>

执行语句的成员或群组名

未指定时在所有群组中执行

说明

通常情况下CURRENT状态的日志文件写满时将自动发生日志切换该语句在特殊情况下需强制切换日志时使用

使用示例

```
ALTER SYSTEM SWITCH LOGFILE;
```

兼容性

标准SQL未定义日志文件的概念

参考

相关内容参考**ALTER SYSTEM {MOUNT | OPEN} DATABASE**

8.47 ALTER TABLE

功能

变更表的定义

语句

```
<alter table statement> ::=
      <alter table physical attribute statement>
    | <rename table statement>
    | <add column definition>
    | <drop column definition>
    | <alter column definition>
    | <rename column statement>
    | <add table constraint definition>
    | <drop table constraint definition>
    | <alter table constraint definition>
    | <alter table drop offline segments statement>
    | <rename table constraint statement>
    | <add table supplemental log statement>
    | <drop table supplemental log statement>
    | <rebalance statement>
```

```
    | <move shard statement>
    | <merge shards statement>
    | <split shard statement>
    | <alter table synchronize statement>
    | <rename shard statement>
    | <read { only | write } statement>
    ;
```

使用范围及访问权限

用户需要拥有以下权限中的一个才能执行<alter table statement>语句

- 对表有(ALTER或CONTROL TABLE) ON TABLE
- 对表所在的SCHEMA有(ALTER TABLE或CONTROL SCHEMA) ON SCHEMA
- ALTER ANY TABLE ON DATABASE

语句规则及参数

<alter table physical attribute statement>

变更表的物理属性

详细内容参考**ALTER TABLE name STORAGE**

<rename table statement>

变更表名称

详细内容参考**ALTER TABLE name RENAME TO**

<add column definition>

在表添加column

详细内容参考**ALTER TABLE name ADD COLUMN**

<drop column definition>

删除表的column

详细内容参考**ALTER TABLE name SET UNUSED COLUMN**

<alter column definition>

变更表column的定义

详细内容参考**ALTER TABLE name ALTER COLUMN**

<rename column statement>

变更表column的名称

详细内容参考**ALTER TABLE name RENAME COLUMN**

<add table constraint definition>

添加表的约束条件

详细内容参考**ALTER TABLE name ADD CONSTRAINT**

<drop table constraint definition>

删除表的约束条件

详细内容参考**ALTER TABLE name DROP CONSTRAINT**

<alter table constraint definition>

变更表的约束条件

详细内容参考**ALTER TABLE name ALTER CONSTRAINT**

<alter table drop offline segments statement>

删除表的离线shard

详细内容参考**ALTER TABLE name DROP OFFLINE SEGMENTS**

<rename table constraint statement>

变更表的约束条件名

详细内容参考**ALTER TABLE name RENAME CONSTRAINT**

<add table supplemental log statement>

表的数据发生变更时在重做日志上添加附加信息

详细内容参考**ALTER TABLE name ADD SUPPLEMENTAL LOG**

<drop table supplemental log statement>

表的数据发生变更时不在重做日志上添加附加信息

详细内容参考**ALTER TABLE name DROP SUPPLEMENTAL LOG**

<rebalance statement>

在cluster环境中重新分配表的shard或同步失去一致性的shard以恢复一致性

详细内容参考**ALTER TABLE REBALANCE**

<alter table synchronize statement>

同步cluster环境中已分配的离线shard来恢复一致性

详细内容参考**ALTER TABLE name SYNCHRONIZE**

<move shard statement>

在cluster环境中将表的特定shard重新分配到特定集群组中

详细内容参考**ALTER TABLE MOVE SHARD**

<merge shards statement>

在cluster环境中合并表的特定shard并重新分配

详细内容参考**ALTER TABLE name MERGE SHARDS**

<split shard statement>

在cluster环境中分散表的特定shard并重新分配到特定集群组中

详细内容参考 **ALTER TABLE SPLIT SHARD**

<rename shard statement>

在cluster环境中变更表的特定shard名

详细内容参考**ALTER TABLE name RENAME SHARD**

<read { only | write } statement>

设置表的READ { only | write }

详细内容参考**ALTER TABLE name READ { ONLY | WRITE }**

说明

参考各语句的详细说明

使用示例

参考各语句的详细使用示例

兼容性

标准SQL中未定义以下语句

- <alter table physical attribute statement>
- <rename table statement>
- <rename column statement>
- <rename table constraint statement>
- <add table supplemental log statement>
- <drop table supplemental log statement>
- <rebalance statement>
- <move shard statement>
- <split shard statement>
- <rename shard statement>
- <read { only | write } statement>

8.48 ALTER TABLE name ADD COLUMN

功能

在表添加column

语句

```
<add column definition> ::=

      ALTER TABLE table_name ADD [ COLUMN ] <column definition>

    | ALTER TABLE table_name ADD [ COLUMN ] ( <column definition>

[, ...] )

    ;
```

使用范围及访问权限

用户应满足以下条件才能执行<add column definition>语句

- 需要有以下权限中的一个才能变更表
  - 对相应表有(ALTER或CONTROL TABLE) ON TABLE
  - 对表所在的SCHEMA有(ALTER TABLE或CONTROL SCHEMA) ON SCHEMA
  - ALTER ANY TABLE ON DATABASE
- 新增Column时如指定了约束条件则需要满足如下可创建约束的条件

    - 对创建约束条件的SCHEMA需要有以下权限中的一个
        - 对该SCHEMA有(ADD CONSTRAINT或CONTROL SCHEMA) ON SCHEMA
        - ALTER ANY TABLE ON DATABASE
    - 创建的约束条件为key约束条件时对创建索引的表空间需要有以下权限中的一个
        - 对该表空间有CREATE OBJECT ON TABLESPACE
        - USAGE TABLESPACE ON DATABASE
- 该表的所有者对新增的Column有以下权限
    - 对新增的所有Column的权限
        - SELECT(columns) ON TABLE WITH GRANT OPTION
        - INSERT(columns) ON TABLE WITH GRANT OPTION
        - UPDATE(columns) ON TABLE WITH GRANT OPTION
        - REFERENCES(columns) ON TABLE WITH GRANT OPTION
    - 对同时创建的约束条件的权限
        - 约束条件的所有者
        - 与约束条件同时创建的索引的所有者

语句规则及参数

table_name

要变更的表名

与schema_name.table_name相同可定义表所在的SCHEMA省略schema_name时使用执行语句的用户的默认SCHEMA名

ADD [COLUMN]

可省略COLUMN保留字

<column definition>

定义新增的Column

详细内容参考**CREATE TABLE**的**<column definition>**

同一个表内不能有相同名称的Column

定义column时若指定了DEFAULT子句则将所有row的默认值存储于新增的column

定义column时若指定了<identity column specification>子句则将所有各个row自动生成的值存储于添加的column

定义column时若同时指定了NOT NULL约束条件则清空表或应同时描述DEFAULT子句或<identity column specification>子句

(<column definition> [, ...])

新增多个Column

在括号内罗列多个<column definition>

说明

新增的column位于现有column的后面

指定DEFAULT子句或<identity column specification>时执行时间与表中存在的行数成正比

使用示例

以下为新增一个column的示例

```
gSQL> ALTER TABLE region ADD COLUMN r_new_comment VARCHAR(152);


Table altered.
```

以下为新增多个column的示例

```
gSQL> ALTER TABLE partsupp ADD COLUMN (
   ps_retailprice NUMERIC(12,2),
   ps_acctbal NUMERIC(12,2), ps_mktsegment  CHAR(10) );


Table altered.
```

以下为新增拥有identity column与DEFAULT子句的column的示例

```
gSQL> ALTER TABLE region ADD COLUMN (
    r_regionkey INTEGER GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
    r_comment   VARCHAR(152) DEFAULT 'N/A' );


Table altered.


gSQL> SELECT r_regionkey, r_name, r_comment FROM region;
```

```
R_REGIONKEY R_NAME                    R_COMMENT
----------- ------------------------- ---------
          1 AFRICA                    N/A
          2 AMERICA                   N/A
          3 ASIA                      N/A
          4 EUROPE                    N/A
          5 MIDDLE EAST               N/A

5 rows selected.
```

以下为新增可延时的约束条件column的示例

```
gSQL> ALTER TABLE t1 ADD COLUMN ( id INTEGER CONSTRAINT t1_uk UNIQUE
DEFERRABLE );

Table altered.

gSQL> COMMIT;

Commit complete.
```

兼容性

标准SQL中未定义新增多个column definition

参考

相关内容参考下文

- **ALTER TABLE**
- **ALTER TABLE name SET UNUSED COLUMN**
- **ALTER TABLE name ALTER COLUMN**
- **ALTER TABLE name RENAME COLUMN**

8.49 ALTER TABLE name ADD CONSTRAINT

功能

添加表的约束条件

语句

```
<add table constraint definition> ::=
    ALTER TABLE table_name
        ADD <table constraint definition>
    ;
```

使用范围及访问权限

用户应满足以下条件才能执行<add table constraint definition>语句

- 对生成约束条件的表需要有以下权限中的一个
  - 对相应表有(ALTER或CONTROL TABLE) ON TABLE
  - 对表所在的SCHEMA有(ALTER TABLE或CONTROL SCHEMA) ON SCHEMA
  - ALTER ANY TABLE ON DATABASE
- 对生成约束条件的SCHEMA需要有以下权限中的一个
  - 对相应SCHEMA有(ADD CONSTRAINT或CONTROL SCHEMA) ON SCHEMA

- ALTER ANY TABLE ON DATABASE

- 生成key约束条件时对生成索引的表空间需要有以下权限中的一个
  - 对相应表空间有CREATE OBJECT ON TABLESPACE
  - USAGE TABLESPACE ON DATABASE
- 如下决定生成的约束条件的所有者
  - 约束条件所在的schema的所有者
  - 约束条件所在的schema为PUBLIC时执行语句的用户

> Note:
>
> 在集群系统中PRIMARY KEY与UNIQUE应包含在所有shrding key中

语句规则及参数

table_name

要变更的表名

与schema_name.table_name相同可定义表所在的SCHEMA省略schema_name时使用执行语句的用户的默认SCHEMA名

<table constraint definition>

定义要添加的约束条件

NOT NULL约束条件不能用ALTER TABLE .. ADD CONSTRAINT语句添加可如下使用**ALTER TABLE name ALTER COLUMN**语句定义

```
ALTER TABLE t1 ALTER COLUMN c1 SET NOT NULL;
```

详细内容参考**CREATE TABLE**语句的**<table constraint definition>**

说明

添加primary keyunique key等key约束条件时自动生成对应的索引

使用示例

以下为添加primary key约束条件的示例

```
gSQL> ALTER TABLE t1 ADD PRIMARY KEY ( id );


Table altered.
```

以下为添加表的primary key约束条件时指定约束条件名的示例

```
gSQL> ALTER TABLE t1 ADD CONSTRAINT t1_pk PRIMARY KEY ( id );


Table altered.
```

以下为新增可延时的约束条件的示例

```
gSQL> ALTER TABLE t1 ADD CONSTRAINT t1_uk UNIQUE ( id ) DEFERRABLE
```

```
INITIALLY DEFERRED;


Table altered.


gSQL> COMMIT;


Commit complete.
```

兼容性

Feature ID	说明	是否支持
F381	Extended schema manipulation	O

Table 8-4 标准SQL兼容性

参考

相关内容参考下文

- **CREATE TABLE**
- **CREATE INDEX**
- **ALTER TABLE**
- **ALTER TABLE name DROP CONSTRAINT**

8.50 ALTER TABLE name ADD GLOBAL SECONDARY INDEX

功能

在表创建全局二级索引（global secondary index）

语句

```
<alter table add global secondary index definition> ::=
    ALTER TABLE table_name
        ADD GLOBAL SECONDARY INDEX
        [ <index attributes> [...] ] [ TABLESPACE tablespace_name ]
    ;

<index attributes> ::=
      <physical attribute clause>
    | STORAGE ( <segment attr clause> [...] )
    | <parallel clause>

<physical attribute clause> ::=
      PCTFREE integer
    | INITRANS integer
```

```
    | MAXTRANS integer


<segment attr clause> ::=

      INITIAL <size_clause>

    | NEXT <size_clause>

    | MINSIZE <size_clause>

    | MAXSIZE <size_clause>


<size clause> ::=

      integer [ K | M | G | T ]


<parallel clause> ::=

      NOPARALLEL

    | PARALLEL [ integer ]
```

使用范围及访问权限

可在集群系统中定义<alter table add global secondary index definition>语句并且用户需要满足以下条件

- 对创建索引的表有以下权限中的一个权限
  - 对表有(ALTER或CONTROL TABLE) ON TABLE
  - 对表所在的SCHEMA有(ALTER TABLE或CONTROL SCHEMA) ON SCHEMA
  - ALTER ANY TABLE ON DATABASE
- 对创建索引的表空间有以下权限中的一个权限

- 对表空间有CREATE OBJECT ON TABLESPACE
- USAGE TABLESPACE ON DATABASE

语句规则及参数

table_name

要生成索引的表名称

与schema_name.table_name相同可定义表所在的SCHEMA省略schema_name时使用执行语句的用户的默认SCHEMA名

<physical attribute clause>

定义索引的物理属性信息

- PCTFREE integer
  - 定义
    - 为了调节在页内插入key引起的页分割频率而预留的空间
  - 可使用从0~99之间的值
  - 省略时使用现有索引的设置值
- INITRANS integer
  - 定义
    - 表示可同时访问页的初始事务的数量
    - 访问索引的用户数量少时设置低的INITRANS同时访问的用户多时设置高的INITRANS

        - 必要时自动扩展至已设置的MAXTRANS
    - 可使用1~32之间的值
    - 省略时使用现有索引的设置值
- MAXTRANS integer
    - 定义
        - 表示可同时访问页的事务的最大值
    - 可使用1~32之间的值
    - 省略时使用现有索引的设置值

<segment attr clause>

描述存储索引的空间相关信息

- INITIAL integer
    - 定义
        - 描述索引创建初期分配的物理空间大小
        - 该大小应align到表所属的TABLESPACE的EXTENT大小后使用（ex：EXT大小为8192bytes时‘INITIAL 100’实际以8192bytes运行）
        - 该大小（align到TABLESPACE的EXTENT大小的大小）应大于或等于MINEXTENTS的大小小于或等于MAXEXTENTS的大小
    - 最小值为1最大值根据系统环境有所不同
    - 省略时使用现有索引的设置值
- NEXT integer
    - 定义
        - 描述添加索引的物理空间时分配的物理空间大小

        - 该大小应align到表所属的TABLESPACE的EXTENT大小（ex：EXT大小为8192bytes时‘INITIAL 100’实际以8192bytes运行）
        - NEXT根据当前索引可使用的剩余空间的大小（从MAXEXTENTS的大小减去当前使用中的空间的大小）如下运行
            - 剩余空间的大小为0时无法再扩展空间
            - 剩余空间的大小大于0且小于NEXT时分配与剩余空间大小相同的空间
            - 剩余空间的大小大于NEXT时分配与NEXT大小相同的空间
    - 最小值为1最大值根据系统环境有所不同
    - 省略时使用现有索引的设置值
- MINSIZE integer
    - 定义
        - 索引需维持的最小空间大小
        - 该值应小于或等于MAXSIZE的值
    - 该大小应align到索引所属的TABLESPACE的EXTENT大小后使用
    - 最小值为1最大值根据系统环境有所不同
    - 如小于2个EXTENT的大小时确定为2个EXTENT的大小
    - 省略时使用现有索引的设置值
- MAXSIZE integer
    - 定义
        - 可在索引中分配到的最大空间大小
        - 该值应大于或等于MINSIZE的值
    - 该大小应align到索引所属的TABLESPACE的EXTENT大小后使用
    - 最小值为1最大值根据系统环境有所不同
    - 省略时使用现有索引的设置值

<size clause>

指定文件的byte大小（未描述单位时使用bytes）

- K : kilobytes
- M : megabytes
- G : gigabytes
- T : terabytes

NOPARALLEL | PARALLEL [integer]

指定创建索引的过程中使用的线程数

- NOPARALLEL
  - 不并行创建索引
- PARALLEAL[integer]
  - 并行创建索引
  - 省略integer或指定为0时遵循参数(INDEX_BUILD_PARALLEL_FACTOR)
  - integer可使用从0开始的值最大值为64
  - 如果integer或参数值为0时由系统决定最佳值
- 未指定时默认值为NOPARALLEL

TABLESPACE tablespace_name

指定存储索引的表空间的名称

- 指定tablespace_name时
  - 要变更为LOGGING索引时tablespace_name应为data tablespace
  - 要变更为NOLOGGING索引时tablespace_name应为temporary tablespace或nologging tablespace
- 省略TABLESPACE子句时遵循原有索引的设置

说明

进行non-deterministic查询必须要有全局二级索引LOGGING索引与NOLOGGING索引有如下trade-off

- LOGGING索引
  - 优点：启动系统时使用日志自动恢复索引不需要单独的创建过程
  - 缺点：变更相关row时在日志记录索引变更内容引发磁盘I/O
- NOLOGGING索引
  - 优点：变更相关row时对索引变更内容不产生磁盘I/O
  - 缺点：启动系统时无索引的日志信息自动重新创建索引

使用示例

以下为在表T1添加全局二级索引的示例

```
gSQL> ALTER TABLE T1 ADD GLOBAL SECONDARY INDEX;
```

```
Table altered.

gSQL> COMMIT;

Commit complete.
```

以下为通过logging option在表1的tablespace USER_DATA_TBS创建全局二级索引的示例

```
gSQL> ALTER TABLE T1 ADD GLOBAL SECONDARY INDEX TABLESPACE USER_DATA_TBS;

Table altered.

gSQL> COMMIT;

Commit complete.
```

以下为通过nologging option在表T1的tablespace USER_TEMP_TBS创建全局二级索引的示例

```
gSQL> ALTER TABLE T1 ADD GLOBAL SECONDARY INDEX TABLESPACE USER_TEMP_TBS;

Table altered.

gSQL> COMMIT;

Commit complete.
```

兼容性

标准SQL未定义全局二级索引相关概念

参考

相关内容参考下文

- **ALTER TABLE name DROP GLOBAL SECONDARY INDEX**
- **ALTER TABLE name ALTER GLOBAL SECONDARY INDEX**
- **CREATE TABLE**

8.51 ALTER TABLE name ADD SUPPLEMENTAL LOG

功能

变更表的数据时如果该表有Primary Key则在Redo Log中记录Primary Key值

语句

```
<add table supplemental log statement> ::=
    ALTER TABLE table_name
        ADD SUPPLEMENTAL LOG DATA ( PRIMARY KEY ) COLUMNS
    ;
```

使用范围及访问权限

用户需要拥有以下权限中的一个才能执行<add table supplemental log statement>语句

- 对表有(ALTER或CONTROL TABLE) ON TABLE
- 对表所在的SCHEMA有(ALTER TABLE或CONTROL SCHEMA) ON SCHEMA
- ALTER ANY TABLE ON DATABASE

语句规则及参数

table_name

要变更的表名

与schema_name.table_name相同可定义表所在的SCHEMA省略schema_name时使用执行语句的用户的默认SCHEMA名

表中没有Primary Key也能执行此语句

说明

对表执行UPDATE/DELETE时记录SUPPLEMENTAL LOG记录的SUPPLEMENTAL LOG用于分析CDC等工具或日志

要对所有表记录SUPPLEMENTAL LOG时参数设置为SUPPLEMENTAL_LOG_DATA_PRIMARY_KEY = YES

使用示例

以下为设置变更表的数据时在redo log里记录primary key值的示例

```
gSQL> ALTER TABLE t1 ADD SUPPLEMENTAL LOG DATA ( PRIMARY KEY ) COLUMNS;
```

```
Table altered.
```

兼容性

标准SQL未定义<add table supplemental log statement>

参考

相关内容参考**ALTER TABLE name DROP SUPPLEMENTAL LOG**

8.52 ALTER TABLE name ALTER COLUMN

功能

变更column的定义

语句

```
<alter column definition> ::=

    ALTER TABLE table_name

        ALTER [ COLUMN ] column_name <alter column action>


<alter column action> ::=


      <set column default clause>

    | <drop column default clause>

    | <set column not null clause>

    | <drop column not null clause>

    | <alter column data type clause>

    | <alter identity column specification>

    | <drop identity property clause>

    ;
```

```
<set column default clause> ::=

    SET DEFAULT <default option>


<drop column default clause> ::=

    DROP DEFAULT


<set column not null clause> ::=

    SET [ CONSTRAINT constraint_name ] NOT NULL [ <constraint

characteristics> ]




<constraint characteristics> ::=

      [ NOT ] DEFERRABLE [ <constraint check time> ]

    | <constraint check time> [ [ NOT ] DEFERRABLE ]


<constraint check time> ::=

      INITIALLY DEFERRED

    | INITIALLY IMMEDIATE


<drop column not null clause> ::=

    DROP NOT NULL


<alter column data type clause> ::=

    SET DATA TYPE <data type>
```

```
<alter identity column specification> ::=
     <set identity column generation clause> [ <alter identity column
option> ... ]
   | <alter identity column option> ...

<set identity column generation clause> ::=
   SET GENERATED { ALWAYS | BY DEFAULT }

<alter identity column option> ::=
     <alter sequence generator restart option>
   | [ SET ] <basic sequence generator option>

<alter sequence generator restart option> ::=
    RESTART [ WITH integer ]

<basic sequence generator option> ::=
      <sequence generator increment by option>
    | <sequence generator maxvalue option>
    | <sequence generator minvalue option>
    | <sequence generator cycle option>
    | <sequence generator cache option>

<drop identity property clause> ::=
    DROP IDENTITY
```

使用范围及访问权限

用户需要拥有以下权限中的一个才能执行<alter column definition>语句

- 对相应表有(ALTER或CONTROL TABLE) ON TABLE
- 对表所在的SCHEMA有(ALTER TABLE或CONTROL SCHEMA) ON SCHEMA
- ALTER ANY TABLE ON DATABASE

语句规则及参数

table_name

要变更的表名

与schema_name.table_name相同可定义表所在的SCHEMA省略schema_name时使用执行语句的用户的默认SCHEMA名

ALTER [COLUMN]

可省略COLUMN保留字

column_name

要变更的column名

<set column default clause>

设置column的默认值

不能为标识列（identity column）

在后续执行的插入语句等中使用DEFAULT时使用设置的默认值

DEFAULT expression的数据类型与column的数据类型应兼容

数据类型不兼容或空间不足时在INSERTUPDATE语句中使用DEFAULT时报错

详细说明参考**CREATE TABLE**的**<default clause>**

<drop column default clause>

删除column的默认值

不能为标识列（identity column）

如果删除了默认值则在INSERT语句中使用DEFAULT子句时会将其设置为NULL

<set column not null clause>

- SET [CONSTRAINT constraint_name] NOT NULL [<constraint characteristics>]
  - 在Column设置NOT NULL约束条件
  - Column值不允许NULL值
  - 此column上不能有NULL值
- 省略[CONSTRAINT constraint_name]时自动赋予约束条件名称
- 省略<constraint characteristics>时具有NOT DEFERRABLE INITIALLY IMMEDIATE属性

- Identity column不能拥有DEFERRABLE属性

可延时的约束条件相关详细说明参考**SET CONSTRAINTS**语句

<drop column not null clause>

- DROP NOT NULL
  - 删除Column的NOT NULL约束条件

<alter column data type clause>

- SET DATA TYPE <data type>
  - 变更Column的数据类型

> Note:
>
> SET DATA TYPE为自动提交的DDL语句

同一系列之间可转换类型此时需要满足以下条件

from \ to	CHAR(n)	VARHCAR(n)	LONG VARCHAR
CHAR(m)	X	X	X
VARCHAR(m)	X	n >= m	X
LONG VARCHAR	X	X	O

Table 8-5 character string type转换

变更char length unit时需要满足以下条件

from \ to	OCTETS	CHARACTERS
OCTETS	O	O
CHARACTERS	X	O

Table 8-6 character length unit转换

from \ to	BINARY(n)	VARBINARY(n)	LONG VARBINARY
BINARY(m)	X	X	X
VARBINARY(m)	X	n >= m	X
LONG VARBINARY	X	X	O

Table 8-7 binary string type转换

from \ to	SMALLINT	INTEGER	BIGINT	NUMERIC	NUMERIC(q)	NUMERIC(q,t)	NUMBER(q)
SMALLINT	O	O	O	O	q >= 5	q >= 5 t >= 0 (q-t) >= 5	q >= 5
INTEGER	X	O	O	O	q >= 10	q >= 10 t >= 0 (q-t) >= 10	q >= 10

from \ to	SMALLINT	INTEGER	BIGINT	NUMERIC	NUMERIC(q)	NUMERIC(q,t)	NUMBER(q)
BIGINT	X	X	O	O	q >= 19	q >= 19 t >= 0 (q-t) >= 19	q >= 19
NUMERIC	X	X	X	O	q == 38	q == 38 t == 0	q == 38
NUMERIC(p)	5 >= p	10 >= p	19 >= p	O	q >= p	q >= p t >= 0 (q-t) >= p	q >= p
NUMERIC(p,s)	5 >= p 0 >= s 5 >= (p-s)	10 >= p 0 >= s 10 >= (p-s)	19 >= p 0 >= s 19 >= (p-s)	38 >= p 0 >= s 38 >= (p-s)	q >= p 0 >= s q >= (p-s)	q >= p t >= s (q-t) >= (p-s)	q >= p 0 >= s q >= (p-s)
NUMBER(p)	5 >= p	10 >= p	19 >= p	O	q >= p	q >= p t >= 0 (q-t) >= p	q >= p
NUMBER(p,s)	5 >= p 0 >= s 5 >= (p-s)	10 >= p 0 >= s 10 >= (p-s)	19 >= p 0 >= s 19 >= (p-s)	38 >= p 0 >= s 3 8 >= (p-s)	q >= p 0 >= s q >= (p-s)	q >= p t >= s (q-t) >= (p-s)	q >= p 0 >= s q >= (p-s)

from \ to	SMALLINT	INTEGER	BIGINT	NUMERIC	NUMERIC(q)	NUMERIC(q,t)	NUMBER(q)
REAL	X	X	X	X	X	X	X
DOUBLE PRECISION	X	X	X	X	X	X	X
FLOAT	X	X	X	X	X	X	X
FLOAT(p)	X	X	X	X	X	X	X
NUMBER	X	X	X	X	X	X	X

Table 8-8 numeric type

FLOAT(p)的ddc(decimal digit count)值如下

FLOAT(p)	ddc(p)
1 ~ 3	1
4 ~ 6	2
7 ~ 9	3
10 ~ 13	4

FLOAT(p)	ddc(p)
14 ~ 16	5
17 ~ 19	6
20 ~ 23	7
24 ~ 26	8
27 ~ 29	9
30 ~ 33	10
34 ~ 36	11
37 ~ 39	12
40 ~ 43	13
44 ~ 46	14
47 ~ 49	15
50 ~ 53	16
54 ~ 56	17
57 ~ 59	18
60 ~ 63	19
64 ~ 66	20
67 ~ 69	21
70 ~ 73	22

FLOAT(p)	ddc(p)
74 ~ 76	23
77 ~ 79	24
80 ~ 83	25
84 ~ 86	26
87 ~ 89	27
90 ~ 93	28
94 ~ 96	29
97 ~ 99	30
100 ~ 103	31
104 ~ 106	32
107 ~ 109	33
110 ~ 113	34
114 ~ 116	35
117 ~ 119	36
120 ~ 123	37
124 ~ 126	38

Table 8-9 ddc(decimal digit count)值

所有numeric type以相同结构进行管理各numeric type与以下NUMBER(p,s)表达式相同

Numeric type	NUMBER(p,s) 表达式
SMALLINT	NUMBER(5,0)
INTEGER	NUMBER(10,0)
BIGINT	NUMBER(19,0)
NUMERIC	NUMBER(38,0)
NUMERIC(p)	NUMBER(p,0)
NUMERIC(p,s)	NUMBER(p,s)
NUMBER(p)	NUMBER(p,0)
NUMBER(p,s)	NUMBER(p,s)
REAL	NUMBER(8,N/A) <= FLOAT(24)
DOUBLE PRECISION	NUMBER(16,N/A) <= FLOAT(53)
FLOAT	NUMBER(38,N/A) <= FLOAT(126)
FLOAT(p)	NUMBER(ddc(p), N/A)
NUMBER	NUMBER(38, N/A)

Table 8-10 Numeric type的NUMBER表达式

Native数字型与C语言的数字型类型相同不能转换为互不相同的类型

from \ to	NATIVE_SMALLINT	NATIVE_INTEGER	NATIVE_BIGINT	NATIVE_REAL	NATIVE_DOUBL
NATIVE_SMALLINT	O	X	X	X	X
NATIVE_INTEGER	X	O	X	X	X
NATIVE_BIGINT	X	X	O	X	X
NATIVE_REAL	X	X	X	O	X
NATIVE_DOUBLE	X	X	X	X	O

Table 8-11 Native数字型的转换

from \ to	BOOLEAN
BOOLEAN	O

Table 8-12 Boolean type的转换

from \ to	DATE	TIME	TIME(g)	TIME TZ	TIME(g) TZ	TIMESTAMP	TIMESTAMP(g)	TIMESTAMP TZ
DATE	O	X	X	X	X	X	X	X
TIME	X	O	g >= 6	X	X	X	X	X
TIME(f)	X	6 >= f	g >= f	X	X	X	X	X
TIME TZ	X	X	X	O	g >= 6	X	X	X
TIME(f) TZ	X	X	X	6 >= f	g >= f	X	X	X
TIMESTAMP	X	X	X	X	X	O	g >= 6	X

from \ to	DATE	TIME	TIME(g)	TIME TZ	TIME(g) TZ	TIMESTAMP	TIMESTAMP(g)	TIMESTAMP TZ
TIMESTAMP(f)	X	X	X	X	X	6 >= f	g >= f	X
TIMESTAMP TZ	X	X	X	X	X	X	X	O
TIMESTAMP(f) TZ	X	X	X	X	X	X	X	6 >= f

Table 8-13 Date/ time type的转换(TZ: WITH TIME ZONE)

from \ to	YEAR(q)	MONTH(q)	YEAR(q) TO MONTH
YEAR(p)	q >= p	X	X
MONTH(p)	X	q >= p	X
YEAR(p) TO MONTH	X	X	q >= p

Table 8-14 INTERVAL YEAR TO MONTH系列的type转换(省略p,q时默认为2)

from \ to	DAY(q)	HOUR(q)	MINUTE(q)	SECOND(q,g)	DAY(q) TO HOUR	DAY(q) TO MINUTE	DAY(q) TO SECOND(g)	HOUR(q) TO MINUTE
DAY(p)	q >= p	X	X	X	X	X	X	X
HOUR(p)	X	q >= p	X	X	X	X	X	X
MINUTE(p)	X	X	q >= p	X	X	X	X	X

from \ to	DAY(q)	HOUR(q)	MINUTE(q)	SECOND(q,g)	DAY(q) TO HOUR	DAY(q) TO MINUTE	DAY(q) TO SECOND(g)	HOUR(q) TO MINUTE
SECOND(p,f)	X	X	X	q >= p g >= f	X	X	X	X
DAY(p) TO HOUR	X	X	X	X	q >= p	X	X	X
DAY(p) TO MINUTE	X	X	X	X	X	q >= p	X	X
DAY(p) TO SECOND(f)	X	X	X	X	X	X	q >= p g >= f	X
HOUR(p) TO MINUTE	X	X	X	X	X	X	X	q >= p
HOUR(p) TO SECOND(f)	X	X	X	X	X	X	X	X
MINUTE(p) TO SECOND(f)	X	X	X	X	X	X	X	X

Table 8-15 INTERVAL DAY TO TIME系列的type转换(省略p,q时默认为2) (省略f,g时默认为6)

from \ to	ROWID
ROWID	0

Table 8-16 ROWID type

<alter identity column specification>

变更column的identity属性

Column应为identity column

- SET GENERATED [ALWAYS | BY DEFAULT]
  - 变更identity column的创建方式
  - 详细内容参考**CREATE TABLE**的**<identity column specification>**
- <alter sequence generator restart option>
  - 变更identity column的NEXT VALUE
  - 详细内容参考**ALTER SEQUENCE**的**<alter sequence generator restart option>**
- <basic sequence generator option>
  - 变更identity column的属性
  - 标准SQL中定义应描述为SET <basic sequence generator option>形式但可省略
  - 详细内容参考**ALTER SEQUENCE** 语句

<drop identity property clause>

删除column的identity属性

Column应为identity column

说明

SET NOT NULL 查询null检查执行时间与表的row数成正比

以下column不允许NULL值即即使执行DROP NOT NULL语句如果满足以下条件中的一个则不允许有NULL值

- 有NOT NULL约束条件的Column
- Column包含在primary key约束条件时
- Column为identity column时

使用SET DEFAULT的默认值变更与使用<alter identity column specification>的identity属性变更应用于之后执行的INSERT或UPDATE语句

使用示例

以下为设置column的DEFAULT属性的示例

```
gSQL> ALTER TABLE region ALTER COLUMN r_comment SET DEFAULT 'N/A';


Table altered.
```

以下为删除column的DEFAULT属性的示例

```
gSQL> ALTER TABLE region ALTER COLUMN r_comment DROP DEFAULT;
```

```
Table altered.
```

以下为设置column的NOT NULL约束条件的示例

```
gSQL> ALTER TABLE region ALTER COLUMN r_regionkey SET NOT NULL;


Table altered.
```

以下为删除column的NOT NULL约束条件的示例

```
gSQL> ALTER TABLE region ALTER COLUMN r_regionkey DROP NOT NULL;


Table altered.
```

以下为扩展column的数据类型大小的示例

```
gSQL> ALTER TABLE region ALTER COLUMN r_comment SET DATA TYPE
VARCHAR(512);


Table altered.
```

以下为重新启动identity column的NEXT VALUE的示例

```
gSQL> ALTER TABLE region ALTER COLUMN r_regionkey RESTART;


Table altered.
```

以下为删除column的identity属性的示例

```
gSQL> ALTER TABLE region ALTER COLUMN r_regionkey DROP IDENTITY;


Table altered.
```

兼容性

Feature ID	说明	是否支持
F381	Extended schema manipulation	X
F382	Alter column data type	O
F383	Set column not null clause	O
F384	Drop identity property value	O
F385	Drop column generation expression clause	X
F386	Set identity column generation clause	O
S043	Enhanced reference types	X
T174	Identity columns	O
T178	Identity columns: simple restart option	O

Table 8-17 标准SQL兼容性

参考

相关内容参考下文

- **ALTER TABLE**
- **ALTER TABLE name ADD COLUMN**
- **ALTER TABLE name SET UNUSED COLUMN**
- **ALTER TABLE name RENAME COLUMN**

8.53 ALTER TABLE name ALTER CONSTRAINT

功能

变更表约束条件的特性

语句

```
<alter table constraint definition> ::=

    ALTER TABLE table_name

        ALTER <constraint object> <constraint characteristics>

    ;
```

```
<constraint object> ::=
      CONSTRAINT constraint_name
    | PRIMARY KEY
    | UNIQUE ( column_name [, ...] )

<constraint characteristics> ::=
      [ NOT ] DEFERRABLE [ <constraint check time> ]
    | <constraint check time> [ [ NOT ] DEFERRABLE ]

<constraint check time> ::=
      INITIALLY DEFERRED
    | INITIALLY IMMEDIATE
```

使用范围及访问权限

用户需要有以下权限中的一个才能执行<alter table constraint definition>语句

- 对表有(ALTER或CONTROL TABLE) ON TABLE
- 对表所在的SCHEMA有(ALTER TABLE或CONTROL SCHEMA) ON SCHEMA
- ALTER ANY TABLE ON DATABASE

> Note:
>
> 集群不支持可延时的约束条件

语句规则及参数

table_name

要变更的表名

与schema_name.table_name相同可定义表所在的SCHEMA省略schema_name时使用执行语句的用户的默认SCHEMA名

<constraint object>

如下指定要变更的约束条件

- CONSTRAINT constraint_name
  - 要变更的的约束条件名
- PRIMARY KEY
  - 表的primary key约束条件
- UNIQUE(column [,...])
  - 符合column目录的unique约束条件

DEFERRABLE | NOT DEFERRABLE

变更约束条件的可延时与否

- DEFERRABLE
  - 变更为可延时的约束条件

- NOT DEFERRABLE
  - 变更为不可以延时的约束条件

可延时的约束条件相关内容参考**SET CONSTRAINTS**语句说明

INITIALLY IMMEDIATE | INITIALLY DEFERRED

变更约束条件的检查时间点的初始值

- INITIALLY IMMEDIATE
  - 执行DML时检查约束条件
- INITIALLY DEFERRED
  - 执行COMMIT时检查约束条件

以NOT DEFERRABLE定义的约束条件不能变更为INITIALLY DEFERRED

说明

可延时的约束条件相关详细说明参考**SET CONSTRAINTS**语句

使用示例

以下为将t1_uk约束条件设置为可延时并将检查时间点设置为DEFERRED的示例

```
gSQL> ALTER TABLE t1 ALTER CONSTRAINT t1_uk DEFERRABLE INITIALLY DEFERRED;
```

```
Table altered.

gSQL> COMMIT;

Commit complete.
```

兼容性

标准SQL未定义以下语句

- ALTER PRIMARY KEY
- ALTER UNIQUE(column [,...])

Feature ID	说明	是否支持
F492	Optional table constraint enforcement	X

Table 8-18 标准SQL兼容性

8.54 ALTER TABLE name ALTER GLOBAL SECONDARY INDEX

功能

变更表的全局二级索引的物理属性

语句

```
<alter table alter global secondary index storage statement> ::=
    ALTER TABLE table_name ALTER GLOBAL SECONDARY INDEX
      <physical attribute clause>
    | [ STORAGE ( <segment attr clause> [...] ) ]
    ;

<physical attribute clause> ::=
      PCTFREE integer
    | INITRANS integer
    | MAXTRANS integer

<segment attr clause> ::=
      INITIAL <size_clause>
    | NEXT <size_clause>
```

```
    | MINSIZE <size_clause>
    | MAXSIZE <size_clause>


<size clause> ::=
      integer [ K | M | G | T ]
```

使用范围及访问权限

用户需拥有以下权限中的一个才能执行<alter table alter global secondary index storage statement>语句

- 对表有(ALTER或CONTROL TABLE) ON TABLE
- 对表所在的SCHEMA有(ALTER TABLE或CONTROL SCHEMA) ON SCHEMA
- ALTER ANY TABLE ON DATABASE

语句规则及参数

table_name

要创建索引的表的名称

与schema_name.table_name相同可定义表所在的SCHEMA省略schema_name时使用执行语句的用户的默认SCHEMA名

<physical attribute clause>

定义索引的物理属性信息

- PCTFREE integer
  - 定义
    - 为了调节在页内插入key引起的页分割频率而预留的空间
    - 仅应用于以bottom-up创建索引时
  - 可使用从0~99之间的值
  - 省略时默认值使用DEFAULT_INDEX_PCTFREE property中设置的值
- INITRANS integer
  - 定义
    - 表示可同时访问页的初始事务的数量
    - 访问索引的用户数量少时设置低的INITRANS同时访问的用户多时设置高的INITRANS
    - 必要时自动扩展至已设置的MAXTRANS
  - 可使用1~32之间的值
  - 省略时默认值为4
- MAXTRANS integer
  - 定义
    - 表示可同时访问页的事务的最大值
  - 可设置1~32之间的值
  - 省略时默认值为8

<segment attr clause>

描述存储索引的空间信息

- INITIAL integer
  - 定义
    - 描述索引创建初期分配的物理空间大小
    - 该大小应align到表所属的TABLESPACE的EXTENT大小后使用（ex：EXT大小为8192bytes时‘INITIAL 100’实际以8192bytes运行）
    - 该大小（align到TABLESPACE的EXTENT大小的大小）应大于或等于MINEXTENTS的大小小于或等于MAXEXTENTS的大小
  - 最小值为1最大值根据系统环境有所不同
  - 省略时默认值与表所属的TABLESPACE的一个EXTENT的大小相同
- NEXT integer
  - 定义
    - 描述添加索引的物理空间时分配的物理空间大小
    - 该大小应align到表所属的TABLESPACE的EXTENT大小后使用（ex：EXT大小为8192bytes时‘NEXT 100’实际以8192bytes运行）
    - NEXT根据当前索引可使用的剩余空间的大小（从MAXEXTENTS的大小减去当前使用中的空间的大小）如下运行
      - 剩余空间的大小为0时无法再扩展空间
      - 剩余空间的大小大于0且小于NEXT时分配与剩余空间大小相同的空间
      - 剩余空间的大小大于NEXT时分配与NEXT大小相同的空间
  - 最小值为1最大值根据系统环境有所不同
  - 省略时默认值与表所属的TABLESPACE的一个EXTENT的大小相同
- MINSIZE integer

    - 定义
        - 索引需维持的最小空间大小
        - 该值应小于或等于MAXSIZE的值
    - 该大小应align到索引所属的TABLESPACE的EXTENT大小后使用
    - 最小值为1最大值根据系统环境有所不同
    - 如小于2个EXTENT的大小时确定为2个EXTENT的大小
    - 省略时默认值为2个EXTENT的大小
- MAXSIZE integer
    - 定义
        - 可在索引中分配到的最大空间大小
        - 该值应大于或等于MINSIZE的值
    - 该大小应align到索引所属的TABLESPACE的EXTENT大小后使用
    - 最小值为1最大值根据系统环境有所不同
    - 省略时默认值为32 terabyte（35,184,372,088,832）
    - 即使指定大于32 terabyte的值也会修改成32 terabyte后设置

<size clause>

指定文件的byte大小（未描述单位时为bytes）

- K: Kilobytes
- M: Megabytes
- G: Gigabytes
- T: Terabytes

说明

进行non-deterministic查询必须要有全局二级索引

使用示例

将表T1的全局二级索引使用的最大大小变更为100 Mbyte

```
gSQL> ALTER TABLE T1 ALTER GLOBAL SECONDARY INDEX STORAGE( MAXSIZE 100M );

Table altered.

gSQL> COMMIT;

Commit complete.
```

将表T1的全局二级索引使用的 INITRANSMAXTRANS值变更为2与4.

```
gSQL> ALTER TABLE T1 ALTER GLOBAL SECONDARY INDEX INITRANS 2 MAXTRANS 4;

Table altered.

gSQL> COMMIT;

Commit complete.
```

兼容性

标准SQL未定义全局二级索引相关概念

参考

相关内容参考下文

- **ALTER TABLE name ADD GLOBAL SECONDARY INDEX**
- **ALTER TABLE name DROP GLOBAL SECONDARY INDEX**

8.55 ALTER TABLE name ALTER GLOBAL SECONDARY INDEX COALESCE

功能

去除global secondary index的碎片化

语句

```
<global secondary index coalesce statement> ::=

    ALTER TABLE table_name ALTER GLOBAL SECONDARY INDEX COALESCE

    ;
```

使用范围及访问权限

用户须满足以下条件才能执行<global secondary index coalesce statement>语句

- 要重建索引的表至少需要拥有以下一种权限
  - 对相应表有(ALTER或CONTROL TABLE) ON TABLE
  - 对表所属模式有(ALTER TABLE或CONTROL SCHEMA) ON SCHEMA
  - ALTER ANY TABLE ON DATABASE
- 要重建索引的表空间至少需要拥有以下一种权限

- 对相应表空间有CREATE OBJECT ON TABLESPACE
- USAGE TABLESPACE ON DATABASE

语句规则及参数

table_name

目标表的名称

可以像schema_name.table_name一样定义表所属的模式省略schema_name时将使用执行语句用户的默认模式名称

说明

- 依次扫描叶页发现可以合并的页面时进行页面合并并将已删除的页面返回给段
- 可以解决因UPDATE/ DELETE等发生的叶页碎片化问题
- 因仅在邻近叶页可合并时运行所以如果碎片化程度较低可能不会有效果
- 如果索引碎片化程度高那么处理时间可能会比INDEX REBUILD要长

	INDEX REBUILD	INDEX COALESCE
更改索引属性	可以	不可以
移动表空间	可以	不可以
锁表	需要	不需要

	INDEX REBUILD	INDEX COALESCE
用于执行的额外空间	需要	不需要
降低树高	可以	不可以

Table 8-19 与INDEX REBUILD比较

使用示例

去除表T1的global secondary index碎片化

```
gSQL> ALTER TABLE T1 ALTER GLOBAL SECONDARY INDEX COALESCE;


Table altered.
```

兼容性

标准SQL未定义global secondary index的概念

参考

相关内容请参考**ALTER TABLE name ALTER GLOBAL SECONDARY INDEX REBUILD**

8.56 ALTER TABLE name ALTER GLOBAL SECONDARY INDEX REBUILD

功能

重构global secondary index

语句

```
<global secondary index rebuild statement> ::=
    ALTER TABLE table_name ALTER GLOBAL SECONDARY INDEX REBUILD
        [ ONLINE | OFFLINE ]
        [ <index attributes> [...] ]
        [ TABLESPACE tablespace_name ]
    ;

<index attributes> ::=
      <physical attribute clause>
    | STORAGE ( <segment attr clause> [...] )
    | <parallel clause>

<physical attribute clause> ::=
      PCTFREE integer
```

```
    | INITRANS integer
    | MAXTRANS integer

<segment attr clause> ::=
      INITIAL <size_clause>
    | NEXT <size_clause>
    | MINSIZE <size_clause>
    | MAXSIZE <size_clause>

<size clause> ::=
      integer [ K | M | G | T ]

<parallel clause> ::=
      NOPARALLEL
    | PARALLEL [ integer ]
```

使用范围及访问权限

用户须满足以下条件才能执行<global secondary index rebuild statement>语句

- 对重构索引的表至少需要拥有以下一种权限
  - 对该表有 (ALTER或 CONTROL TABLE) ON TABLE
  - 对表所属的schema有 (ALTER TABLE 或 CONTROL SCHEMA) ON SCHEMA
  - ALTER ANY TABLE ON DATABASE
- 对重构索引的表空间至少需要拥有以下一种权限

- 对该表空间有 CREATE OBJECT ON TABLESPACE
- USAGE TABLESPACE ON DATABASE

语句规则及参数

table_name

要重构索引的表的名称

可以像schema_name.table_name一样定义表所属的模式省略schema_name时将使用执行语句的用户的默认模式名称

[ONLINE | OFFLINE]

决定在重构索引时是否允许对该表进行DML

- ONLINE
  - 允许INSERTUPDATEDELETE
- OFFLINE
  - 不允许INSERTUPDATEDELETE
- 省略时默认值为ONLINE

<physical attribute clause>

定义索引的物理属性信息

- PCTFREE integer
  - 定义
    - 为了调整页内插入key引起的页分割频率而预留的空间
  - 可使用0~99的值
  - 省略时使用原有索引设定值
- INITRANS integer
  - 定义
    - 表示可同时访问页的初始事务的数量
    - 访问索引的用户数量少时INITRANS设置得低同时访问的用户多时INITRANS设置得高
    - 必要时自动增加至设置的MAXTRANS
  - 可使用1~32的值
  - 省略时使用原有索引设定值
- MAXTRANS integer
  - 定义
    - 表示可同时访问页的事务的最大数量
  - 可使用1~32的值
  - 省略时使用原有索引设定值

<segment attr clause>

说明存储索引的空间的信息

- INITIAL integer
  - 定义

        - 描述重构索引时初期分配的物理空间的大小
        - 其大小align到表所属的TABLESPACE的EXTENT大小后使用（例EXT大小为8192 bytes时'INITIAL 100'实际以8192 bytes运行）
        - 其大小（align到TABLESPACE的EXTENT的大小）应大于或等于MINEXTENTS的大小小于或等于MAXEXTENTS的大小
    - 最小值为1最大值根据系统环境有所不同
    - 省略时使用原有索引的设定值
- NEXT integer
    - 定义
        - 在索引增加物理空间时描述要分配的物理空间的大小
        - 其大小align到表所属的TABLESPACE的EXTENT大小后使用（例EXT大小为8192 bytes时'NEXT 100'实际以8192 bytes运行）
        - NEXT根据目前索引可使用的剩余空间的大小（在MAXEXTENTS的大小减去当前使用中的空间的大小）如下运行
            - 剩余空间的大小为0时无法再扩展空间
            - 剩余空间的大小大于0小于NEXT时分配与剩余空间大小相同的大小
            - 剩余空间的大小大于NEXT时分配与NEXT大小相同的大小
    - 最小值为1最大值根据系统环境有所不同
    - 省略时使用原有索引的设定值
- MINSIZE integer
    - 定义
        - 索引要维持的最小空间的大小
        - 此值应小于或等于MAXSIZE的值
    - 其大小align到索引所属的 TABLESPACE的EXTENT大小后使用
    - 最小值为1最大值根据系统环境有所不同

    - 小于两个EXTENT的大小时确定为两个EXTENT的大小
    - 省略时使用原有索引的设定值
- MAXSIZE integer
    - 定义
        - 可在索引中分配到的最大空间的大小
        - 此值应大于或等于MINSIZE的值
    - 其大小align到索引所属的TABLESPACE的EXTENT大小后使用
    - 最小值为1最大值根据系统环境有所不同
    - 省略时使用原有索引的设定值

<size clause>

指定文件的byte大小（未指定单位时使用bytes）

- K: Kilobytes
- M: Megabytes
- G: Gigabytes
- T: Terabytes

NOPARALLEL | PARALLEL [integer]

指定重构索引的过程中使用的thread的数量

- NOPARALLEL
    - 不并行重构索引
- PARALLEL [integer]

    - 并行重构索引
    - 省略ingeger或指定为0时遵循INDEX_BUILD_PARALLEL_FACTOR参数
    - integer可从0开始使用最大值为64
    - 如果integer或参数的值为0则由系统决定最优值
- 未指定时默认值为NOPARALLEL

TABLESPACE tablespace_name

指定重构索引的表空间的名称

- 指定tablespace_name时
    - tablespace_name为data tablespace时重构为LOGGING索引
    - tablespace_name为temporary tablespace或nologging tablespace时重构为NOLOGGING索引
- 省略TABLESPACE子句时设置为原有索引的表空间

说明

- 删除索引碎片化
    - 在索引频繁执行update DML时索引页可能产生碎片化相比有效的数据tree过于大时索引的容量增加而性能下降此时重构索引可解决索引页的碎片化缩小索引容量并恢复索引的性能
- 变更索引的表空间
    - 可变更原先创建的索引的表空间
    - 但要根据表空间的TEMPORARY与否设置恰当的logging与否

- 变更索引的logging设置
  - 可使用TABLESPACE选项变更已创建的索引的logging设置
  - 要变更为LOGGING索引时应在TABLESPACE选项指定数据表空间
  - 要变更为NOLOGGING索引时应在TABLESPACE选项指定temporary tablespace或nologging tablespace

使用示例

重构表T1的global secondary index

```
gSQL> ALTER TABLE T1 ALTER GLOBAL SECONDARY INDEX REBUILD;
```

变更表T1的global secondary index的tablespace和logging设置

```
gSQL> ALTER TABLE T1 ALTER GLOBAL SECONDARY INDEX REBUILD TABLESPACE
MEM_DATA_TBS;


gSQL> ALTER TABLE T1 ALTER GLOBAL SECONDARY INDEX REBUILD TABLESPACE
MEM_TEMP_TBS;
```

兼容性

标准SQL未定义global secondary index的概念

参考

相关内容参考下文

- **ALTER TABLE name ADD GLOBAL SECONDARY INDEX**
- **ALTER TABLE name DROP GLOBAL SECONDARY INDEX**
- **ALTER INDEX name REBUILD**

8.57 ALTER TABLE name DROP CONSTRAINT

功能

删除表的约束条件

语句

```
<drop table constraint definition> ::=
    ALTER TABLE table_name
        DROP <constraint object>
        [ <drop behavior> ]
    ;

<constraint object> ::=
      CONSTRAINT constraint_name
    | PRIMARY KEY
    | UNIQUE ( column_name [, ...] )

<drop behavior> ::=
      RESTRICT
    | CASCADE
    | CASCADE CONSTRAINTS
```

使用范围及访问权限

用户需要有以下权限中的一个才能执行<drop table constraint definition>语句

- 约束条件的所有者
- 对相应表有（ALTER或CONTROL TABLE）ON TABLE
- 对表所在的SCHEMA有（ALTER TABLE或CONTROL SCHEMA）ON SCHEMA
- ALTER ANY TABLE ON DATABASE

语句规则及参数

table_name

要变更的表名

与schema_name.table_name相同可定义表所在的SCHEMA省略schema_name时使用执行语句的用户的默认SCHEMA名

CONSTRAINT constraint_name

要删除的约束条件的名称

PRIMARY KEY

表的primary key约束条件

UNIQUE(column_name [, ...])

Column的unique约束条件

<drop behavior>

省略时默认值为RESTRICT

目前与RESTRICT/CASCADE的运行原理相同

说明

不使用约束条件的名称并删除NOT NULL约束条件时使用**ALTER TABLE name ALTER COLUMN**语句的**<drop column not null clause>**

使用示例

以下为删除primary key约束条件的示例

```
gSQL> ALTER TABLE t1 DROP PRIMARY KEY;


Table altered.
```

以下为指定约束条件的名称后删除表的约束条件的示例

```
gSQL> ALTER TABLE t1 DROP CONSTRAINT t1_pk;
```

```
Table altered.
```

兼容性

标准SQL未定义以下语句

- DROP PRIMARY KEY
- DROP UNIQUE (column_name [, ...])
- CASCADE CONSTRAINTS

Feature ID	说明	是否支持
F381	Extended schema manipulation	O

Table 8-20 标准SQL兼容性

参考

相关内容参考下文

- **ALTER TABLE**
- **ALTER TABLE name ADD CONSTRAINT**
- **DROP INDEX**

8.58 ALTER TABLE name DROP GLOBAL SECONDARY INDEX

功能

删除表的全局二级索引

语句

```
<alter table drop global secondary index definition> ::=
    ALTER TABLE table_name
        DROP GLOBAL SECONDARY INDEX
    ;
```

使用范围及访问权限

用户需要满足以下条件才能在集群系统中定义<alter table drop global secondary index definition>语句

- 对删除索引的表有以下权限中的一个权限
    - 对表有(ALTER或CONTROL TABLE) ON TABLE
    - 对表所在的SCHEMA有(ALTER TABLE或CONTROL SCHEMA) ON SCHEMA
    - ALTER ANY TABLE ON DATABASE

语句规则及参数

table_name

要删除索引的表的名称

说明

进行non-deterministic查询必须要有全局二级索引

使用示例

删除表T1的全局二级索引

```
gSQL> ALTER TABLE T1 DROP GLOBAL SECONDARY INDEX;

Table altered.

gSQL> COMMIT;

Commit complete.
```

兼容性

标准SQL未定义全局二级索引相关概念

参考

相关内容参考下文

- **ALTER TABLE name ADD GLOBAL SECONDARY INDEX**
- **ALTER TABLE name ALTER GLOBAL SECONDARY INDEX**

8.59 ALTER TABLE name DROP OFFLINE SEGMENTS

功能

删除离线shard的段

语句

```
<alter table drop offline segments statement> ::=
    ALTER TABLE table_name
        DROP OFFLINE SEGMENTS
;
```

使用范围及访问权限

可在集群系统中执行

须至少拥有以下权限中的一种才能执行<alter table drop offline segments statement>语句

- 对相应表有(ALTER 或CONTROL TABLE) ON TABLE
- 对表所属的模式有(ALTER TABLE或CONTROL SCHEMA) ON SCHEMA
- ALTER ANY TABLE ON DATABASE

语句规则及参数

table_name

表的名称

如schema_name.table_name一样可以定义表所属的模式省略schema_name时使用执行语句的用户的默认模式名称

说明

删除离线shard的段

即使存在 inactive cluster member也可执行<alter table drop offline segments statement>

若不满足以下条件则失败

- 若想删除cloned表的段集群系统内至少一个成员中应存在cloned表的在线replica
- 若想删除sharded表的段每个组的至少一个成员中应存在该sharded表的在线replica

例如当sharded表t1的cluster group G3中的所有replica处于离线状态时发生如下错误

```
gSQL> ALTER TABLE t1 DROP OFFLINE SEGMENTS;


ERR-42000(16361): sharded table "PUBLIC"."T1" must be accessible to at
least one member of group 'G3'
```

对所有表执行时使用**<alter database drop offline segments statement>**语句

使用示例

以下是对表t1执行<alter table drop offline segments statement>语句的示例

```
gSQL> ALTER TABLE t1 DROP OFFLINE SEGMENTS;


Table altered.
```

兼容性

标准SQL未定义集群的概念

参考

相关内容请参考**ALTER DATABASE DROP OFFLINE SEGMENTS**

8.60 ALTER TABLE name DROP SUPPLEMENTAL LOG

功能

变更表的数据时在Redo Log里不记录Primary Key信息

语句

```
<add table supplemental log statement> ::=
    ALTER TABLE table_name
        DROP SUPPLEMENTAL LOG DATA ( PRIMARY KEY ) COLUMNS
    ;
```

使用范围及访问权限

用户需要有以下权限中的一个才能执行<drop table supplemental log statement>语句

- 对表有(ALTER或CONTROL TABLE) ON TABLE
- 对表所在的SCHEMA有(ALTER TABLE或CONTROL SCHEMA) ON SCHEMA
- ALTER ANY TABLE ON DATABASE

语句规则及参数

table_name

要变更的表的名称

与schema_name.table_name相同可定义表所在的SCHEMA省略schema_name时使用执行语句的

用户的默认SCHEMA名

通过**ALTER TABLE name ADD SUPPLEMENTAL LOG**语句设置

说明

参考各语句的使用规则

使用示例

以下为设置变更表的数据时在redo log里不记录primary key信息的示例

```
gSQL> ALTER TABLE t1 DROP SUPPLEMENTAL LOG DATA ( PRIMARY KEY ) COLUMNS;


Table altered.
```

兼容性

标准SQL未定义<drop table supplemental log statement>

8.61 ALTER TABLE name MERGE SHARDS

功能

在集群环境中merge表的特定shard后重新分配

语句

```
<alter table merge shards statement> ::=

    ALTER TABLE table_name MERGE SHARDS <source shard list>

       INTO dest_shard_name [ <dest shard placement> ]

    ;


<source shard list> ::=

    source_shard_name [, ...]

  | start_shard_name TO end_shard_name


<dest shard placement> ::=

    AT CLUSTER GROUP dest_group_name
```

使用范围及访问权限

可在集群系统执行

需要有以下权限中的一个才能执行<alter table merge shards statement>语句

- 对相应表有(ALTER 或 CONTROL TABLE) ON TABLE
- 对表所属的schema有(ALTER TABLE 或 CONTROL SCHEMA) ON SCHEMA
- ALTER ANY TABLE ON DATABASE

语句规则及参数

table_name

表的名称

如schema_name.table_name可定义表所属的schema省略schema_name时使用执行语句的用户的

默认schema名

只能在对应表为cluster-specific并且是list shard或range shard的情况下执行语句

<source shard list>

要merge的源shard的list

List指定的shard必须存在于对应表

source_shard_name

要merge的源shard的名称

不存在于对应表的shard时无法执行语句

start_shard_name

要merge的范围中起始shard的名称

仅在range shard中使用

end_shard_name

要merge的范围中最后一个shard的名称

仅在range shard中使用

dest_shard_name

对象shard的名称

<dest shard placement>

分配对象shard的集群租的名称

省略该语句时dest_shard_name应包含在<source shard list>

说明

Merge特定表的特定shard分配到任意集群租

- 无法在单机版数据库执行
- 无法在Hash sharded table或cloned table执行
- 无法在以cluster wide生成的表执行
- 进行merge的过程中无法执行对source shard的DML
- Range shard的情况可指定要merge的源shard的开始与结束
- 可在range shard或list shard列出要merge的源shard但此时在range shard列出的shard应均为相邻的shard

以下为在range sharded表尝试merge未相邻的shard时报错

```
CREATE TABLE t1( i1 INTEGER )
    SHARDING BY RANGE (i1)
    SHARD shard1 VALUES LESS THAN ( 200 )      AT CLUSTER GROUP G1,
    SHARD shard2 VALUES LESS THAN ( 400 )      AT CLUSTER GROUP G2,
    SHARD shard3 VALUES LESS THAN ( MAXVALUE ) AT CLUSTER GROUP G3
;

Table created.

ALTER TABLE t1 MERGE SHARDS shard1, shard3 INTO shard4 AT CLUSTER GROUP
G2;
```

```
ERR-42000(16488): shards being merged are not adjacent :
ALTER TABLE t1 MERGE SHARDS shard1, shard3 INTO shard4 AT CLUSTER GROUP G2
                                        *
ERROR at line 1:
```

使用示例

以下为以罗列型merge SHARD的示例

```
gSQL> ALTER TABLE t1 MERGE SHARDS shard1, shard2, shard3 INTO shard4 AT
CLUSTER GROUP G2;

Table altered
```

以下为以范围型合并SHARD的示例

```
gSQL> ALTER TABLE t1 MERGE SHARDS shard1 TO shard3 INTO shard4 AT CLUSTER
GROUP G2;

Table altered
```

兼容性

标准SQL未定义cluster的概念

参考

相关内容参考下文

- **ALTER TABLE name MOVE SHARD**
- **ALTER TABLE name SPLIT SHARD**

8.62 ALTER TABLE name MOVE SHARD

功能

将表的特定shard或特定集群组的所有shard重新部署到特定集群群组

语句

```
<alter table move shard statement> ::=
    ALTER TABLE table_name MOVE SHARD
        { shard_name_list | FROM CLUSTER GROUP src_cluster_group }
        TO CLUSTER GROUP dest_cluster_group
       [ ONLINE | OFFLINE ]
       [ <shard divisor> ]
       [ <parallel clause> ]
    ;


<shard divisor> ::=
    SHARD DIVISOR integer


<parallel clause> ::=
    NOPARALLEL
  | PARALLEL [ integer ]
```

使用范围及访问权限

可在集群系统中执行

需要有以下权限中的一个才能执行<alter table move shard statement>语句

- 对表有(ALTER或CONTROL TABLE) ON TABLE
- 对表所在的SCHEMA有(ALTER TABLE或CONTROL SCHEMA) ON SCHEMA
- ALTER ANY TABLE ON DATABASE

语句规则及参数

table_name

表的名称

与schema_name.table_name相同可定义表所在的SCHEMA省略schema_name时使用执行语句的

用户的默认SCHEMA名

仅在该表为cluster group specific时可执行语句

shard_name_list

要重新分配的shard name list

为不存在于该表的shard时无法执行语句

src_cluster_group

要重新分配的特定集群组的名称

dest_cluster_group

要分配表的shard的目标集群组的名称

该表的shard已存在于指定集群组时无法执行语句

[ONLINE | OFFLINE]

确定在重新分配表的shard时是否允许DML

- ONLINE
  - 允许INSERT, UPDATE, DELETE
- OFFLINE
  - 不允许INSERT, UPDATE, DELETE
- 省略时默认值为ONLINE

<shard divisor>

指定shard分区数量

- 按照分区的数量划分shard并重新分配到远程服务器
- 整数可从0开始使用最大值为1000
- 省略时根据REBALANCE_SHARD_DIVISOR属性决定

- integer小于parallel integer时修改为和parallel integer相同的值

<parallel clause>

指定重新分配表时使用的线程数量

- NOPARALLEL
  - 不并行重新分配表
- PARALLEL [integer]
  - 并行重新分配表
  - 整数可以从0开始使用最大值为64
  - 省略整数时为0
  - 整数为0时 系统确定最佳值

说明

将表的特定shard从特定集群组重新分配到其他集群组

为了删除特定集群组需要在重新分配表的shard后执行**DROP CLUSTER GROUP**语句

将所有表的shard从特定集群组移动至其他集群组时执行ALTER DATABASE MOVE SHARD FROM CLUSTER GROUP TO CLUSTER GROUP语句

对于CLONED表或设置为CLUSTER WIDE的表可能会发生错误而失败

使用示例

以下为执行<alter table move shard statement>语句的示例

```
gSQL> ALTER TABLE t1 MOVE SHARD shard1, shard2 TO CLUSTER GROUP g3;


Table altered.


gSQL> ALTER TABLE t1 MOVE SHARD FROM CLUSTER GROUP g1 TO CLUSTER GROUP g3;


Table altered.
```

以下为CLONED表和设置为CLUSTER WIDE的表进行move shard失败的示例

```
gSQL> CREATE TABLE T1 ( C1 INTEGER ) SHARDING BY RANGE (C1)
         AT CLUSTER WIDE
         SHARD s1 VALUES LESS THAN (10),
         SHARD s2 VALUES LESS THAN (MAXVALUE);


Table created.


gSQL> ALTER TABLE T1 MOVE SHARD s1 TO CLUSTER GROUP g2;


ERR-42000(16440): cannot execute on cluster wide sharded tables
```

```
gSQL> CREATE TABLE T2 ( C1 INTEGER ) CLONED AT CLUSTER GROUP g1, g2;


Table created.


gSQL> ALTER TABLE T2 MOVE SHARD FROM CLUSTER GROUP g1 TO CLUSTER GROUP g3;


ERR-42000(16437): cannot execute on cloned tables
```

兼容性

标准SQL未定义集群的概念

参考

相关内容参考**ALTER DATABASE MOVE SHARD**

8.63 ALTER TABLE name READ { ONLY | WRITE }

功能

设置表的READ { ONLY | WRITE }

语句

```
<alter table read { only | write } statement> :==
    ALTER TABLE table_name
        READ { ONLY | WRITE }
    ;
```

使用范围及访问权限

需要有以下权限中的一个才能执行<alter table read { only | write } statement>语句

- 对表有(ALTER或CONTROL TABLE) ON TABLE
- 对表所在的SCHEMA有(ALTER TABLE或CONTROL SCHEMA) ON SCHEMA
- ALTER ANY TABLE ON DATABASE

语句规则及参数

table_name

表的名称

与schema_name.table_name相同可定义表所在的SCHEMA省略schema_name时使用执行语句的用户的默认SCHEMA

说明

将表的属性指定为READ { ONLY | WRITE }

指定为READ ONLY则无法使用SELECT .. FOR UPDATE语句变更表数据的DML或DDL语句但可使用不变更表数据的DDL语句

> Note:
>
> 指定为READ ONLY时不可用的SQL语句
>
> * INSERT, UPDATE, DELETE
>
> * TRUNCATE
>
> * SELECT .. FOR UPDATE
>
> * ALTER TABLE RENAME/DROP COLUMN
>
> * ALTER TABLE SET COLUMN UNUSED
>
> 指定为READ ONLY时可用的SQL语句

* SELECT

* CREATE/ALTER/DROP INDEX

* ALTER TABLE ADD/ALTER COLUMN

* ALTER TABLE ADD/ALTER/RENAME/DROP CONSTRAINT

* ALTER TABLE for physical property changes

* ALTER TABLE DROP UNUSED COLUMNS

* ALTER TABLE RENAME TO

* DROP TABLE

* ALTER TABLE ADD/DROP SUPPLEMENTAL LOG

* LOCK TABLE

使用示例

以下为执行<alter table read { only | write } statement>语句的示例

```
gSQL> ALTER TABLE t1 READ ONLY;


Table altered.


gSQL> ALTER TABLE t1 READ WRITE;


Table altered.
```

兼容性

标准SQL未定义<alter table read { only | write } statement>语句

参考

相关内容参考**ALTER TABLE**

8.64 ALTER TABLE name REBALANCE

功能

重新分配表的shard

语句

```
<alter table rebalance statement> ::=
    ALTER TABLE table_name REBALANCE
       [ ONLINE | OFFLINE ]
       [ <shard divisor> ]
       [ <parallel clause> ]
    ;

<shard divisor> ::=
    SHARD DIVISOR integer

<parallel clause> ::=
    NOPARALLEL
  | PARALLEL [ integer ]
```

使用范围及访问权限

可在集群系统中执行

需要有以下权限中的一个才能执行<alter table rebalance statement>语句

- 对表有(ALTER或CONTROL TABLE) ON TABLE
- 对表所在的SCHEMA有(ALTER TABLE或CONTROL SCHEMA) ON SCHEMA
- ALTER ANY TABLE ON DATABASE

语句规则及参数

table_name

表的名称

与schema_name.table_name相同可定义表所在的SCHEMA省略schema_name时使用执行语句的用户的默认SCHEMA名

[ONLINE | OFFLINE]

决定重新分配表的shard时是否允许DML

- ONLINE
  - 允许INSERT, UPDATE, DELETE
- OFFLINE

- 不允许INSERT, UPDATE, DELETE

- 省略时默认值为ONLINE

<shard divisor>

指定shard分区数量

- 按照分区的数量划分shard并重新分配到远程服务器
- 整数可从0开始使用最大值为1000
- 省略时根据REBALANCE_SHARD_DIVISOR属性决定
- integer小于parallel integer时修改为和parallel integer相同的值

<parallel clause>

指定重新分配表时使用的线程数量

- NOPARALLEL
  - 不并行重新分配表
- PARALLEL [integer]
  - 并行重新分配表
  - 整数可以从0开始使用最大值为64
  - 省略整数时为0
  - 整数为0时系统确定最佳值

说明

通过以下语句添加集群成员集群组时不重新分配表的shard

- **CREATE CLUSTER GROUP**
- **ALTER CLUSTER GROUP name ADD MEMBER**

重新分配集群组与集群成员的表的shard需要执行<alter table rebalance statement>语句已经重新分配了表的shard时不需要单独进行重新分配的操作

重新分配所有表的shard时执行**ALTER DATABASE REBALANCE**语句

使用示例

以下为执行<alter table rebalance statement>语句的示例

```
gSQL> ALTER TABLE t1 REBALANCE;


Table altered.
```

兼容性

标准SQL未定义集群的概念

8.65 ALTER TABLE name REBALANCE EXCLUDE CLUSTER GROUP cluster_group_list

功能

重新分配表的shard的同时使特定集群组不包含shard

语句

```
<alter table rebalance exclude cluster group statement> ::=
    ALTER TABLE table_name REBALANCE
        EXCLUDE CLUSTER GROUP cluster_group_list
       [ ONLINE | OFFLINE ]
       [ <shard divisor> ]
       [ <parallel clause> ]
    ;

<shard divisor> ::=
    SHARD DIVISOR integer

<parallel clause> ::=
    NOPARALLEL
```

```
| PARALLEL [ integer ]
```

使用范围及访问权限

可在集群系统中执行

需要有以下权限中的一个才能执行<alter table rebalance exclude cluster group statement>语句

- 对表有(ALTER或CONTROL TABLE) ON TABLE
- 对表所在的SCHEMA有(ALTER TABLE或CONTROL SCHEMA) ON SCHEMA
- ALTER ANY TABLE ON DATABASE

语句规则及参数

table_name

表的名称

与schema_name.table_name相同可定义表所在的SCHEMA省略schema_name时使用执行语句的

用户的默认SCHEMA名

仅在该表为cluster-wide时可执行语句

cluster_group_list

不包含表的shard的集群组清单

不参与重新配置的集群组为全部集群组时无法执行语句

[ONLINE | OFFLINE]

决定在重新配置表的shard时是否允许DML

- ONLINE
  - 允许INSERT, UPDATE, DELETE
- OFFLINE
  - 不允许INSERT, UPDATE, DELETE
- 省略时默认值为ONLINE

<shard divisor>

指定shard分区数量

- 按照分区的数量划分shard并重新分配到远程服务器
- 整数可从0开始使用最大值为1000
- 省略时根据REBALANCE_SHARD_DIVISOR属性决定
- integer小于parallel integer时修改为和parallel integer相同的值

<parallel clause>

指定重新分配表时使用的线程数量

- NOPARALLEL

- 不并行重新分配表

- PARALLEL [integer]
  - 并行重新分配表
  - 整数可以从0开始使用最大值为64
  - 省略整数时为0
  - 整数为0时 系统确定最佳值

说明

排除特定集群组后重新分配表的shard

表的shard不存在于该集群组时不需要单独进行重新分配的操作

以表的shard所在的集群组为准重新分配shard

为了删除特定集群组需要在重新分配表的shard后执行**DROP CLUSTER GROUP**语句

在所有表上排除集群组而重新分配shard时执行**ALTER DATABASE REBALANCE EXCLUDE CLUSTER GROUP**语句

使用示例

以下为执行<alter table rebalance exclude cluster group statement>语句的示例

```
gSQL> ALTER TABLE t1 REBALANCE EXCLUDE CLUSTER GROUP g3;


Table altered.
```

兼容性

标准SQL未定义集群的概念

8.66 ALTER TABLE name RENAME COLUMN

功能

变更表的column名

语句

```
<rename column statement> ::=
    ALTER TABLE table_name
        RENAME COLUMN old_column_name TO new_column_name
    ;
```

使用范围及访问权限

用户需要有以下权限中的一个才能执行<rename column statement>语句

- 对相应表有(ALTER或CONTROL TABLE) ON TABLE
- 对表所在的SCHEMA有(ALTER TABLE或CONTROL SCHEMA) ON SCHEMA
- ALTER ANY TABLE ON DATABASE

语句规则及参数

table_name

要变更的表名

与schema_name.table_name相同可定义表所在的SCHEMA省略schema_name时使用执行语句的用户的默认SCHEMA名

old_column_name

要变更的column原名

new_column_name

要变更的Column的新名称

同一个表内不能有相同的Column名称

说明

即使变更了column名也不需要变更之前以该column为准创建的indexconstraint等对象

使用示例

以下为相互变更两个column名为col_1与col_2的Column的示例

```
gSQL> ALTER TABLE t1 RENAME COLUMN col_1 TO col_temp;

Table altered.

gSQL> ALTER TABLE t1 RENAME COLUMN col_2 TO col_1;

Table altered.

gSQL> ALTER TABLE t1 RENAME COLUMN col_temp TO col_2;

Table altered.

gSQL> COMMIT;

Commit complete.
```

兼容性

标准SQL未定义<rename column statement>语句

参考

相关内容参考下文

- **ALTER TABLE**
- **ALTER TABLE name ADD COLUMN**
- **ALTER TABLE name SET UNUSED COLUMN**
- **ALTER TABLE name ALTER COLUMN**

8.67 ALTER TABLE name RENAME CONSTRAINT

功能

变更表约束条件的名称

语句

```
<rename table constraint statement> ::=
    ALTER TABLE table_name
        RENAME <constraint object> TO new_constraint_name
    ;

<constraint object> ::=
      CONSTRAINT constraint_name
    | PRIMARY KEY
    | UNIQUE ( column_name [, ...] )
```

使用范围及访问权限

用户需要有以下权限中的一个才能执行<rename table constraint statement>语句

- 对表有(ALTER或CONTROL TABLE) ON TABLE
- 对表所在的SCHEMA有(ALTER TABLE或CONTROL SCHEMA) ON SCHEMA
- ALTER ANY TABLE ON DATABASE

语句规则及参数

table_name

要变更的表名

与schema_name.table_name相同可定义表所在的SCHEMA省略schema_name时使用执行语句的用户的默认SCHEMA名

<constraint object>

可如下指定要变更的约束条件的原名

- CONSTRAINT constraint_name
  - 要变更的约束条件名
- PRIMARY KEY
  - 表的PRIMARY KEY约束条件
- UNIQUE(column [,...])
  - 符合column目录的UNIQUE约束条件

new_column_name

要变更的约束条件的新名称

说明

以primary keyunique key等key约束条件自动生成的索引名称不会被变更变更索引名称需使用

ALTER INDEX name RENAME TO语句

使用示例

以下为变更表的primary key约束条件名称的示例

```
gSQL> ALTER TABLE t1 RENAME PRIMARY KEY TO pk_t1;


Table altered.
```

以下为指定约束条件名后变更表的约束条件名称的示例

```
gSQL> ALTER TABLE t1 RENAME CONSTRAINT pk_t1 TO t1_pk;


Table altered.
```

兼容性

标准SQL未定义<rename table constraint statement>语句

参考

相关内容参考下文

- **ALTER TABLE**
- **ALTER TABLE name ADD CONSTRAINT**
- **ALTER TABLE name DROP CONSTRAINT**
- **ALTER TABLE name ALTER CONSTRAINT**

8.68 ALTER TABLE name RENAME SHARD

功能

在集群环境中变更表的特定shard的名称

语句

```
<alter table rename shard statement> ::=
    ALTER TABLE table_name
        RENAME SHARD shard_name TO new_shard_name
    ;
```

使用范围及访问权限

可在集群系统中执行

需要有以下权限中的一个才能执行<alter table rename shard statement>语句

- 对表有(ALTER或CONTROL TABLE) ON TABLE
- 对表所在的SCHEMA有(ALTER TABLE或CONTROL SCHEMA) ON SCHEMA
- ALTER ANY TABLE ON DATABASE

语句规则及参数

table_name

要变更的表的名称

与schema_name.table_name相同可定义表所在的SCHEMA省略schema_name时使用执行语句的用户的默认SCHEMA名

shard_name

要变更的shard的原名

shard不存在于该表时无法执行语句

new_shard_name

要变更的shard的新名称

表中不可存在相同的shard名称

说明

变更HashRangeList表的特定shard名称Cloned表无法执行该语句

使用示例

以下为执行<alter table rename shard statement>语句的示例

```
gSQL> ALTER TABLE t_range RENAME SHARD r_01 TO r_new_01;

Table altered.

gSQL> ALTER TABLE t_list RENAME SHARD l_01 TO l_new_01;

Table altered.

gSQL> ALTER TABLE t_hash RENAME SHARD shard_000000 TO h_new_00;

Table altered.

gSQL> COMMIT;

Commit complete.
```

兼容性

标准SQL未定义集群的概念

参考

相关内容参考下文

- **ALTER TABLE**
- **ALTER TABLE name MOVE SHARD**
- **ALTER TABLE name SPLIT SHARD**
- **ALTER TABLE name REBALANCE**

8.69 ALTER TABLE name RENAME TO

功能

变更表名

语句

```
<rename table statement> ::=

    ALTER TABLE table_name

        RENAME TO new_table_name

    ;
```

使用范围及访问权限

用户需要有以下权限中的一个才能执行<rename table statement>语句

- 对表有(ALTER或CONTROL TABLE) ON TABLE
- 对表所在的SCHEMA有(ALTER TABLE或CONTROL SCHEMA) ON SCHEMA
- ALTER ANY TABLE ON DATABASE

语句规则及参数

table_name

变更前的表名

与schema_name.table_name相同可定义表所在的SCHEMA省略schema_name时使用执行语句的用户的默认SCHEMA名

new_table_name

变更后的表名

同一个SCHEMA内不能有相同名称的表

说明

即使表名的变更也不需要变更引用其的索引约束条件等对象

使用示例

以下为相互变更t1与t2表名称的示例

```
gSQL> ALTER TABLE t1 RENAME TO t_temp;
```

```
Table altered.


gSQL> ALTER TABLE t2 RENAME TO t1;


Table altered.


gSQL> ALTER TABLE t_temp RENAME TO t2;


Table altered.


gSQL> COMMIT;


Commit complete.
```

兼容性

标准SQL未定义<rename table statement>

参考

相关内容参考**ALTER TABLE**

8.70 ALTER TABLE name SET UNUSED COLUMN

功能

删除表的column

语句

```
<drop column definition> ::=
    ALTER TABLE table_name <drop column clause>
    ;

<drop column clause> ::=
      SET UNUSED [ COLUMN ] <column_name_list> [ <drop behavior> ]

<column name list> ::=
      column_name
    | ( column_name [, ...] )

<drop behavior> ::=
      RESTRICT
    | CASCADE
```

```
    | CASCADE CONSTRAINTS
```

使用范围及访问权限

用户需要有以下权限中的一个才能执行<drop column definition>语句

- 对相应表有(ALTER或CONTROL TABLE) ON TABLE
- 对表所在的SCHEMA有(ALTER TABLE或CONTROL SCHEMA) ON SCHEMA
- ALTER ANY TABLE ON DATABASE

语句规则及参数

table_name

要变更的表名

与schema_name.table_name相同可定义表所在的SCHEMA省略schema_name时使用执行语句的用户的默认SCHEMA

SET UNUSED [COLUMN]

设置为不使用该column

column_name_list

要删除的一个以上的Column名

- 例: ALTER TABLE t1 SET UNUSED COLUMN c1
- 例: ALTER TABLE t1 SET UNUSED COLUMN (c1, c2)

column_name

要删除的Column的名称

同时删除使用该Column的约束条件与索引

drop behavior

省略时默认值为RESTRICT

目前RESTRICT / CASCADE的运行原理相同

说明

SET UNUSED COLUMN不会物理删除数据所以与行数无关可保证一定的性能

使用示例

以下为设置不使用对应Column的示例

```
gSQL> ALTER TABLE t1 SET UNUSED COLUMN ( addr );


Table altered.
```

兼容性

标准SQL未定义以下语句

- SET UNUSED
- CASCADE CONSTRAINTS
- 罗列多个Column

Feature ID	说明	是否支持
F033	ALTER TABLE statement: DROP COLUMN clause	X

Table 8-21 标准SQL兼容性

参考

相关内容参考下文

- **ALTER TABLE**
- **ALTER TABLE name ADD COLUMN**
- **ALTER TABLE name ALTER COLUMN**

- **ALTER TABLE name RENAME COLUMN**

8.71 ALTER TABLE name SPLIT SHARD

功能

在集群环境中split表的特定shard后重新分配

语句

```
<alter table split shard statement> ::=
    ALTER TABLE table_name SPLIT SHARD source_shard_name
        INTO ( <split shard placement> [, ...] )
    ;

<split shard placement> ::=
    <split shard bound def> AT CLUSTER GROUP dest_group_name

<split shard bound def> ::=
    <split list shard def>
  | <split range shard def>

<split list shard def> :=
    SHARD dest_shard_name VALUES IN ( <split list value clause> )
```

```
<split list value clause> :=

    <split list value> [, ...]


<split list value> :=

    constant

  | NULL


<split range shard def> :=

    SHARD dest_shard_name VALUES LESS THAN ( <split range value clause> )


<split range value clause> :=

    <split range value> [, ...]


<split range value> :=

    constant
```

使用范围及访问权限

可在集群系统中执行

需要有以下权限中的一个才能执行<alter table split shard statement>语句

- 对表有(ALTER或CONTROL TABLE) ON TABLE
- 对表所在的SCHEMA有(ALTER TABLE或CONTROL SCHEMA) ON SCHEMA
- ALTER ANY TABLE ON DATABASE

语句规则及参数

table_name

表的名称

与schema_name.table_name相同可定义表所在的SCHEMA省略schema_name时使用执行语句的用户的默认SCHEMA名

该表为cluster group specific并为list shard或range shard时可执行语句

source_shard_name

要split的源shard名

shard不存在于对应表时无法执行语句

<split shard placement>

将源shard进行split后定义要重新分配的对象shard

<split shard bound def>

定义要split的对象shard的bound

如下可定义为两种bound def中的一个

- <split list shard def>
- <split range shard def>

<split list shard def>

定义list shard的split shard bound

- dest_shard_name
  - 对象shard名称
- <split list value clause>
  - <split list value>应为常数值
  - <split list value>无法使用NULL值
  - <split list value>无法使用DEFAULT
  - S1 : (1, 11, 21, 31, NULL) SPLIT SHARD S1 INTO (<split list value clause> ..)
    - (O) SHARD S11 VALUES IN (1)
    - (O) SHARD S11 VALUES IN (1, NULL)
    - (O) SHARD S11 VALUES IN (1, 11, 21, 31)
    - (X) SHARD S11 VALUES IN (2)
    - (X) SHARD S11 VALUES IN (DEFAULT)
    - (X) SHARD S11 VALUES IN (1, 11, 21, 31, NULL)

<split range shard def>

定义range shard的split shard bound

- <split range value clause>
  - <split list value>应为常数值
  - <split list value>无法使用NULL值
  - <split list value>无法使用MAXVALUE
  - S1 : (100, 100), S2 : (50, 50) SPLIT SHARD S1 INTO (<split range value clause> .. ）

- (O) SHARD S11 VALUES IN (50, 100)
- (O) SHARD S11 VALUES IN (100, 50)
- (O) SHARD S11 VALUES IN (60, 60)
- (X) SHARD S11 VALUES IN (50, NULL)
- (X) SHARD S11 VALUES IN (50, 50)
- (X) SHARD S11 VALUES IN (100, 100)
- (X) SHARD S11 VALUES IN (100, 110)
- (X) SHARD S11 VALUES IN (MAXVALUE, 100)

dest_group_name

要分配split的shard的集群组名

说明

分散特定表的特定shard后分配在任意集群组

其用于特定shard的记录多或负荷偏重于特定集群成员时通过分散shard分散记录及负荷

使用示例

以下为执行<alter table split shard statement>语句的示例

```
gSQL> ALTER TABLE t1 SPLIT SHARD shard1 INTO ( SHARD shard11 VALUES IN
( 11 ) AT CLUSTER GROUP G2 );
```

```
Table altered.


gSQL> ALTER TABLE t1 SPLIT SHARD shard1 INTO ( SHARD shard11 VALUES LESS
THAN ( 11 ) AT CLUSTER GROUP G2 );



Table altered.
```

兼容性

标准SQL未定义集群的概念

参考

相关内容参考下文

- **ALTER TABLE name REBALANCE**
- **ALTER TABLE name REBALANCE EXCLUDE CLUSTER GROUP cluster_group_list**
- **ALTER TABLE name MOVE SHARD**
- **ALTER TABLE name MERGE SHARDS**

8.72 ALTER TABLE name STORAGE

功能

变更表的物理属性

语句

```
<alter table physical attribute statement> ::=
    ALTER TABLE table_name
      [ <physical attribute clause> ]
    | [ STORAGE ( <segment attr clause> [...] ) ]
    ;

<physical attribute clause> ::=
      PCTFREE integer
    | PCTUSED integer
    | INITRANS integer
    | MAXTRANS integer

<segment attr clause> ::=
    NEXT <size_clause>
    | MINSIZE <size_clause>
```

```
    | MAXSIZE <size_clause>


<size clause> ::=

      integer [ K | M | G | T ]
```

使用范围及访问权限

用户需要有以下权限中的一个才能执行<alter table physical attribute statement>语句

- 对表有(ALTER或CONTROL TABLE) ON TABLE
- 对表所在的SCHEMA有(ALTER TABLE或CONTROL SCHEMA) ON SCHEMA
- ALTER ANY TABLE ON DATABASE

语句规则及参数

table_name

要变更的表名

与schema_name.table_name相同可定义表所在的SCHEMA省略schema_name时使用执行语句的用户的默认SCHEMA名

<physical attribute clause>

变更构成表的页的物理属性

不应用于已分配的页只应用于之后新分配的页

详细内容参考**CREATE TABLE**语句的**<table physical attribute clause>**

<segment attr clause>

变更构成段的extent物理属性

不应用于已分配的extent只应用于之后新分配的extent

- MAXSIZE integer
  - 变更可分配的段的空间大小
  - 如果小于已分配的空间大小则MAXSIZE变更为当前分配的大小

说明

参考各语句的使用规则

使用示例

以下为变更表的物理属性的示例

```
gSQL> ALTER TABLE t1 PCTFREE 10 PCTUSED 40 STORAGE ( NEXT 10M  MAXSIZE
100M );


Table altered.
```

兼容性

标准SQL未定义表的物理属性

参考

相关内容参考**ALTER TABLE**

8.73 ALTER TABLE name SYNCHRONIZE

功能

远程同步现有表的shard

语句

```
<alter table synchronize statement> ::=
    ALTER TABLE table_name SYNCHRONIZE
       [ ONLINE | OFFLINE ]
       [ <shard divisor> ]
       [ <parallel clause> ]
    ;

<shard divisor> ::=
    SHARD DIVISOR integer

<parallel clause> ::=
    NOPARALLEL
  | PARALLEL [ integer ]
```

使用范围及访问权限

可以在集群系统中执行

需要有至少有以下一种权限才能执行<alter table synchronize statement>语句

- 对相应表有(ALTER或CONTROL TABLE) ON TABLE
- 对表所属模式有(ALTER TABLE或CONTROL SCHEMA) ON SCHEMA
- ALTER ANY TABLE ON DATABASE

语句规则及参数

table_name

表的名称

如schema_name.table_name一样可以定义表所属的模式省略schema_name时使用执行语句的用户的默认模式名称

[ONLINE | OFFLINE]

确定在同步表的shard时是否允许DML

- ONLINE
  - 允许INSERT, UPDATE, DELETE
- OFFLINE

- 不允许INSERT, UPDATE, DELETE

- 省略时默认值为ONLINE

<shard divisor>

指定shard分区数量

- 按照分区的数量划分shard并与远程服务器同步
- 整数可从0开始使用最大值为1000
- 省略时根据REBALANCE_SHARD_DIVISOR属性决定
- integer小于parallel integer时修改为和parallel integer相同的值

<parallel clause>

指定同步表时使用的线程数量

- NOPARALLEL
  - 不并行同步表
- PARALLEL [integer]
  - 并行同步表
  - 整数可以从0开始使用最大值为64
  - 省略整数时为0
  - 整数为0时系统确定最佳值

说明

同步现有离线shard恢复一致性和**<alter table rebalance statement>**不同即使存在inactive cluster member也可执行

不满足以下条件时失败

- 若想同步cloned表的shard集群系统内至少一个成员中应存在cloned表的在线replica
- 若想同步sharded表的shard每个组的至少一个成员中应存在该sharded表的在线replica

例如当sharded表t1的cluster group G3中的所有replica处于离线状态时发生如下错误

```
gSQL> ALTER TABLE t1 SYNCHRONIZE;


ERR-42000(16546): sharded table "PUBLIC"."T1" must have at least one
online replica of group 'G3'
```

同步所有表的shard时需执行**<alter database synchronize statement>**

使用示例

以下为对表t1执行<alter table synchronize statement>语句的示例

```
gSQL> ALTER TABLE t1 SYNCHRONIZE;


Table altered.
```

兼容性

标准SQL未定义集群的概念

参考

相关内容参考下文

- **ALTER TABLE**
- **ALTER TABLE name REBALANCE**
- **ALTER DATABASE SYNCHRONIZE**

8.74 ALTER TABLESPACE

功能

变更表空间的定义

语句

```
<alter tablespace statement> ::=
      <rename tablespace statement>
    | <backup tablespace statement>
    | <on/offline tablespace statement>
    | <add file statement>
    | <drop file statement>
    | <rename datafile statement>
    ;
```

使用范围及访问权限

用户需要有数据库的ALTER TABLESPACE权限才能执行<alter tablespace statement>语句

语句规则及参数

<rename tablespace statement>

变更表空间的名称

详细内容参考**ALTER TABLESPACE name RENAME TO**

<backup tablespace statement>

备份表空间

详细内容参考**ALTER TABLESPACE name BACKUP**

<on-offline tablespace statement>

把表空间的所有文件变更为online或offline

详细内容参考**ALTER TABLESPACE name [ONLINE|OFFLINE]**语句

<add file statement>

在表空间中添加文件

详细内容参考**ALTER TABLESPACE name ADD [DATAFILE|MEMORY]**

<drop file statement>

删除表空间的文件

详细内容参考**ALTER TABLESPACE name DROP [DATAFILE|MEMORY]**

<rename datafile statement>

变更数据表空间的数据文件名称

详细内容参考**ALTER TABLESPACE name RENAME DATAFIL**

说明

ALTER TABLESPACE语句与其他Data Definition Language (DDL)不同 不能回滚自动提交执行语句的事务

使用示例

参考各语句的详细使用示例

兼容性

标准SQL未定义表空间的概念

参考

相关内容参考下文

- **CREATE TABLESPACE**
- **DROP TABLESPACE**

8.75 ALTER TABLESPACE name ADD [DATAFILE|MEMORY]

功能

扩展表空间的大小

语句

```
<add space statement> ::=
    ALTER TABLESPACE tablespace_name ADD <space specification>
        [AT <domain name>]
    ;

<space specification> ::=
      MEMORY <memory clause> [, ...]
    | DATAFILE <add datafile clause> [, ...]

<size clause> ::=
    integer [ K | M | G | T ]

<memory clause>
     'memory_name' { SIZE <size clause> }
```

```
<add datafile clause> ::=

     'filename'

        { SIZE <size clause> | REUSE | SIZE <size clause> REUSE }

        [ <autoextend clause> ]


<autoextend clause>

    AUTOEXTEND { ON [ <next size clause> ] [ <max size clause> ] | OFF }


<next size clause>

    NEXT <size clause>


<max size clause>

    MAXSIZE { <size clause> | UNLIMITED }
```

使用范围及访问权限

用户需要有ALTER TABLESPACE ON DATABASE权限才能执行<add space statement>语句

语句规则及参数

tablespace_name

要变更的表空间的名称

<file specification>

根据表空间的类型使用以下语句

- 内存数据表空间
  - DATAFILE <add datafile clause>
- 内存临时表空间
  - MEMORY <memory clause>

<add datafile clause>

定义要添加的内存数据文件

- 'filename'
  - 存储管理数据的文件的名称
  - 存储内存数据CHECKPOINT image的空间
  - filename是新的文件或已有的文件
  - filename的长度应小于1024byte
- SIZE <size clause>
  - 新的文件通过SIZE子句指定初始大小
  - 文件已存在时报错
  - 可指定的文件大小范围为1M ~ 30G
- REUSE
  - 已有文件使用REUSE子句
  - 文件不存在时生成新的文件
  - 新生成的文件大小为

    - 数据表空间取决于MEMORY_DATA_TABLESPACE_SIZE参数
    - 临时表空间取决于MEMORY_TEMP_TABLESPACE_SIZE参数
- SIZE <size clause> REUSE
    - 同时指定SIZE子句与REUSE子句时根据filename的存在与否如下执行
        - 新的filename使用SIZE子句指定文件的初始大小
        - 已有的filename使用现有文件将大小调整为SIZE子句的值

<memory clause>

- <size clause>
    - 定义要添加的内存

详细内容参考**CREATE MEMORY TEMPORARY TABLESPACE**语句的 **<memory clause>**

<autoextend clause>

增加磁盘表空间的数据文件时设定自动扩展参数将自动扩展参数设置为ON或者OFF设置为ON时可设置自动扩展大小和数据文件的最大大小

<next size clause>

目前使用中的数据文件中没有更多空间时指定要扩展的大小

<max size clause>

指定数据文件可扩展的最大大小

<domain name>

要执行语句的成员或群组的名称

未指定时在所有群组中执行

说明

参考各语句的使用规则

使用示例

以下为在表空间中添加数据文件的示例

```
gSQL> ALTER TABLESPACE space1 ADD DATAFILE 'test_file_a2.dbf' SIZE 10M
REUSE;


Tablespace altered.
```

兼容性

标准SQL未定义表空间的概念

参考

相关内容参考以下内容

- **CREATE MEMORY DATA TABLESPACE**
- **CREATE MEMORY TEMPORARY TABLESPACE**
- **ALTER TABLESPACE**

8.76 ALTER TABLESPACE name BACKUP

功能

为了备份表空间转换为可备份与不可备份的状态

语句

```
<backup tablespace statement> ::=
      <tablespace begin backup statement>
    | <tablespace end backup statement>
    | <tablespace incremental backup statement>
    ;

<tablespace begin backup statement> ::=
    ALTER TABLESPACE tablespace_name BEGIN BACKUP [AT <domain_name>]
    ;

<tablespace end backup statement> ::=
    ALTER TABLESPACE tablespace_name END BACKUP [AT <domain_name>]
    ;

<tablesapce incremental backup statement> ::=
```

```
    ALTER TABLESPACE tablespace_name

        BACKUP INCREMENTAL <incremental backup option> [AT
<domain_name>]    ;


<incremental backup option> ::=

      LEVEL integer [ CUMULATIVE | DIFFERENTIAL ]
```

使用范围及访问权限

用户需要有ALTER TABLESPACE ON DATABASE权限才能执行<backup space statement>语句

语句规则及参数

<tablespace begin backup statement>

把表空间设置为可备份的状态

- 把使用中的表空间设置为可备份的状态
- 无法变更OFFLINE/Temporary表空间的backup状态

tablespace_name

要变更Backup状态的表空间名称

<tablespace end backup statement>

把表空间设置为不可备份的状态

<tablesapce incremental backup statement>

执行表空间的增量备份

数据库应为OPEN状态并以归档模式运行

<incremental backup option>

- 'integer'可指定0 ~ 4之间的值
- 'LEVEL 0'不能指定CUMULATIVE或DIFFERENTIAL
- CUMULATIVE | DIFFERENTIAL
  - CUMULATIVE
    - 'integer'为n时对备份最近的'LEVEL 0' ~ 'LEVEL n-1'后变更的所有页进行备份
  - DIFFERENTIAL
    - 'integer'为n时对备份最近'LEVEL 0' ~ 'LEVEL n'备份后变更的所有页进行备份
  - 省略时默认为DIFFERENTIAL

<domain_name>

执行语句的成员或组的名称

未指定时在所有组中执行

说明

备份表空间生成的数据文件备份整个表空间需要执行BEGIN BACKUP后使用操作系统的拷贝命令拷贝数据文件后执行END BACKUP而增量备份文件是用一条语句在BACKUP_DIR_1 property设置的路径下生成

使用示例

以下是把DICTIONARY_TBS表空间的完全备份状态设置为'ACTIVE'的示例

```
ALTER TABLESPACE DICTIONARY_TBS BEGIN BACKUP;
```

以下是把DICTIONARY_TBS表空间的完全备份状态设置为'INACTIVE'的示例

```
ALTER TABLESPACE DICTIONARY_TBS END BACKUP;
```

以下为创建DICTIONARY_TBS表空间的LEVEL 0增量备份的示例

```
ALTER TABLESPACE DICTIONARY_TBS BACKUP INCREMENTAL LEVEL 0;
```

兼容性

标准SQL未定义表空间的概念

参考

相关内容参考下文

- **ALTER TABLESPACE**
- **ALTER TABLESPACE name [ONLINE|OFFLINE]**

8.77 ALTER TABLESPACE name DROP [DATAFILE|MEMORY]

功能

缩小表空间的大小

语句

```
<drop space statement> ::=
    ALTER TABLESPACE tablespace_name DROP <file specification>
    [ AT <domain name> ]
    ;

<file specification> ::=
      DATAFILE 'filename'
    | MEMORY 'memory_name'
```

使用范围及访问权限

用户需要有ALTER TABLESPACE ON DATABASE权限才能执行<drop space statement>语句

语句规则及参数

tablespace_name

要变更的表空间的名称

<file specification>

根据表空间的类型使用以下语句

- 内存数据表空间
  - DATAFILE 'filename'
- 内存临时表空间
  - MEMORY 'memory_name'

> Note:
>
> 无法删除离线表空间的文件
>
> 无法删除表空间的第一个文件
>
> 无法删除使用过的数据文件

<domain name>

要执行语句的成员或群组的名称

未指定时在所有群组中执行

说明

参考各语句的使用规则

使用示例

以下为删除表空间文件的示例

```
gSQL> ALTER TABLESPACE space1 DROP DATAFILE 'test_file_f2.dbf';


Tablespace altered.
```

兼容性

标准SQL未定义表空间的概念

参考

相关内容参考下文

- **ALTER TABLESPACE**
- **ALTER TABLESPACE name ADD [DATAFILE|MEMORY]**
- **ALTER TABLESPACE name RENAME DATAFILE**

8.78 ALTER TABLESPACE name [ONLINE|OFFLINE]

功能

变更表空间的状态

语句

```
<on/off tablespace statement> ::=
    ALTER TABLESPACE tablespace_name { ONLINE | OFFLINE [ NORMAL |
IMMEIDATE ] }
    [ AT <domain name> ]
    ;
```

使用范围及访问权限

用户需要有ALTER TABLESPACE ON DATABASE权限才能执行<on/off tablespace statement>语句

语句规则及参数

ONLINE

把OFFLINE状态的表空间变更为ONLINE状态

OFFLINE NORMAL

把ONLINE状态的表空间变更为OFFLINE状态

变更为OFFLINE的表空间是一致的(consistent)状态因此变更为ONLINE时不需要介质恢复

> Note:
>
> 在MOUNT阶段不能使用OFFLINE NORMAL
>
> （但是如果以前的实例被\ SHUTDOWN NORMAL终止则可以使用）

OFFLINE IMMEDIATE

把ONLINE状态的表空间变更为OFFLINE状态

变更为OFFLINE的表空间是非一致性(inconsistent)状态因此变更为ONLINE时需要介质恢复

> Note:
>
> SYSTEM表空间不能变更为OFFLINE
>
> OFFLINE IMMEDIATE需要介质恢复因此只能在归档模式下执行

<domain name>

要执行语句的成员或群组的名

未指定时在所有群组中执行

说明

参考各语句的使用规则

使用示例

以下为把表空间设置为OFFLINE的示例

```
gSQL> ALTER TABLESPACE space1 OFFLINE;


Tablespace altered.
```

以下为在MOUNT阶段对表空间执行'OFFLINE NORMAL'失败的示例

```
gSQL> ALTER TABLESPACE space1 OFFLINE;


ERR-42000(16290): OFFLINE NORMAL is only allowed if the database is in
OPEN phase :
ALTER TABLESPACE space1 OFFLINE
```

```
                  *
ERROR at line 1:


gSQL> ALTER TABLESPACE space1 OFFLINE NORMAL;


ERR-42000(16290): OFFLINE NORMAL is only allowed if the database is in
OPEN phase :
ALTER TABLESPACE space1 OFFLINE NORMAL
                  *
ERROR at line 1:
```

兼容性

标准SQL未定义表空间的概念

参考

相关内容参考下文

- **ALTER TABLESPACE**
- **ALTER TABLESPACE name BACKUP**

8.79 ALTER TABLESPACE name RENAME DATAFILE

功能

变更构成表空间的数据文件名

语句

```
<rename datafile statement> ::=

    ALTER TABLESPACE tablespace_name RENAME DATAFILE <filename_list> TO
<filename_list>

    ;


    <filename_list> ::=

        'filename' [ AT <domain name> ] [, ...]
```

使用范围及访问权限

用户需要有ALTER TABLESPACE ON DATABASE权限才能执行<rename datafile statement>语句

> Note:

> 如果TDS模式下数据库为OPEN状态则不能变更在线表空间的文件（临时内存表空间除外）
>
> 变更后文件也必须存在

语句规则及参数

tablespace_name

要变更的表空间的名称

'filename'

内存临时表空间指'memory_name'其他表空间指'filename'

<domain name>

要执行语句的成员或群组的名称

未指定时在所有群组中执行

说明

根据表空间的状态确定是否能执行运算

- OFFLINE: MOUNT或OPEN阶段可执行

- ONLINE: 仅在MOUNT阶段可执行

使用示例

以下为将'test.dbf'变更为'test1.dbf'的示例

```
gSQL> ALTER TABLESPACE TEST_TBS RENAME DATAFILE 'test.dbf' TO 'test1.dbf';


Tablespace altered.
```

兼容性

标准SQL未定义表空间的概念

参考

相关内容参考下文

- **ALTER TABLESPACE**
- **ALTER TABLESPACE name ADD [DATAFILE|MEMORY]**
- **ALTER TABLESPACE name DROP [DATAFILE|MEMORY]**

8.80 ALTER TABLESPACE name RENAME TO

功能

变更表空间的名称

语句

```
<rename tablespace statement> ::=

    ALTER TABLESPACE tablespace_name RENAME TO <new_tablespace_name>

    ;
```

使用范围及访问权限

用户需要有ALTER TABLESPACE ON DATABASE权限才能执行<rename space statement>语句

语句规则及参数

tablespace_name

原有表空间的名称

- 不能变更built-in表空间的名称
- 不能变更OFFLINE表空间的名称

new_tablespace_name

新的表空间的名称

说明

即使变更表空间名称也不需要变更该表空间中已创建的表索引等

使用示例

以下为变更表空间名称的示例

```
gSQL> ALTER TABLESPACE space1 RENAME TO space2;


Tablespace altered.
```

兼容性

标准SQL未定义表空间的概念

参考

相关内容参考**ALTER TABLESPACE**

8.81 ALTER USER

功能

变更数据库用户的定义

语句

```
<alter user statement> ::=
      ALTER USER user_identifier <alter user action>
    | ALTER USER PUBLIC <alter schema path>
    ;

<alter user action> ::=
      <alter password>
    | <alter profile>
    | <password expire>
    | <account lock>
    | <alter default tablespace>
    | <alter temporary tablespace>
    | <alter index tablespace>
    | <alter schema path>
```

```
<alter password> ::=

    IDENTIFIED BY new_password [ REPLACE old_password ]


<alter profile> ::=

    PROFILE { profile_name | DEFAULT | NULL }


<password expire> ::=

    PASSWORD EXPIRE


<account lock> ::=

    ACCOUNT { LOCK | UNLOCK }


<alter default tablespace> ::=

    DEFAULT TABLESPACE tablespace_name


<alter temporary tablespace> ::=

    TEMPORARY TABLESPACE tablespace_name


<alter index tablespace> ::=

    INDEX TABLESPACE { tablespace_name | NULL }


<alter schema path> ::=

    SCHEMA PATH ( { schema_name | CURRENT PATH } [, ...] )
```

使用范围及访问权限

用户需要有ALTER USER ON DATABASE权限才能执行<alter user statement>语句

但<alter password>与user_identifier为同一个用户时没有权限也可以执行

语句规则及参数

user_identifier

要变更的用户名

<alter password>

变更用户的密码

- IDENTIFIED BY new_password
  - 加密存储新密码
  - 密码长度应小于128byte
  - 密码区分大小写
- REPLACE old_password
  - 有ALTER USER ON DATABASE权限时可省略
  - 没有ALTER USER ON DATABASE权限时不可省略
    - 用户与user_identifier应相同

<alter profile>

变更密码管理策略的profile

- PROFILE profile_name
  - 分配用户生成的profile_name
- PROFILE DEFAULT
  - 分配默认profile "DEFAULT"
- PROFILE NULL
  - 不分配profile

<password expire>

将用户密码设置为过期

<account lock>

- ACCOUNT LOCK
  - 锁定用户账号
- ACCOUNT UNLOCK
  - 解锁用户账号

<alter default tablespace>

变更用户的默认表空间

tablespace_name应为数据表空间

<alter temporary tablespace>

变更用户的临时表空间

tablespace_name应为临时表空间

<alter index tablespace>

变更用户的索引表空间

- 指定INDEX TABLESPACE tablespace_name
  - 指定数据表空间时成为LOGGING索引
  - 指定临时表空间时成为NOLOGGING索引
- INDEX TABLESPACE NULL
  - 不指定索引表空间

<alter schema path>

变更用户的schema访问路径

用户的SQL语句中未指定SCHEMA时schema访问路径取决于对象的naming resoluntion的schema顺序

schema名称与现有schema访问路径中罗列的schema名称相同时不会重复反映

以下为执行*ALTER USER u1 SCHEMA PATH (u1, s2, public);* 语句时存在于SCHEMA中的对象的示例

SCHEMA名称	u1	s2	public
-	t1	-	t1
-	-	t2	-
-	-	-	t3

如下u1用户执行的未指定SCHEMA的对象名称根据schema path解释如下

- CREATE语句
  - CREATE TABLE t1 (c1 INTEGER);
    - 报错 : CREATE TABLE u1.t1 (c1 INTEGER);
  - CREATE TABLE t2 (c1 INTEGER);
    - 执行 : CREATE TABLE u1.t2 (c1 INTEGER);
- SELECT语句
  - SELECT * FROM t1;
    - 执行 : SELECT * FROM u1.t1;
  - SELECT * FROM t2;
    - 执行 : SELECT * FROM s2.t2;
  - SELECT * FROM t3;
    - 执行 : SELECT * FROM public.t3;

CURRENT PATH

当前用户的schema path

如下使用CURRENT PATH保留原有的schema path并可添加新的schema path

- u1的当前schema path
  - (u1, public)
- 执行语句
  - ALTER USER u1 SCHEMA PATH (s1, CURRENT PATH, s2);
- u1的schema path变更如下
  - (s1, u1, public, s2)

ALTER USER PUBLIC <alter schema path>

变更PUBLIC账号的schema path

PUBLIC账号的schema path包含于所有用户的schema path

PUBLIC账号的初始schema path如下

- DICTIONARY_SCHEMA
- INFORMATION_SCHEMA
- DEFINITION_SCHEMA
- PERFORMANCE_VIEW_SCHEMA
- FIXED_TABLE_SCHEMA

说明

参考各语句的使用规则

使用示例

以下为变更用户密码的示例

```
gSQL> ALTER USER u1 IDENTIFIED BY new_password;

User altered.
```

以下为给用户分配profile的示例

```
gSQL> ALTER USER u1 PROFILE prof1;

User altered.

gSQL> COMMIT;

Commit complete.
```

以下为删除用户的profile的示例

```
gSQL> ALTER USER u1 PROFILE NULL;

User altered.

gSQL> COMMIT;
```

```
Commit complete.
```

以下为将用户的密码设置为过期的示例

```
gSQL> ALTER USER u1 PASSWORD EXPIRE;

User altered.

gSQL> COMMIT;

Commit complete.
```

以下为解锁用户账号的示例

```
gSQL> ALTER USER u1 ACCOUNT UNLOCK;

User altered.

gSQL> COMMIT;

Commit complete.
```

以下为变更用户默认表空间的示例

```
gSQL> ALTER USER u1 DEFAULT TABLESPACE mem_data_tbs;

User altered.
```

以下为变更用户临时表空间的示例

```
gSQL> ALTER USER u1 TEMPORARY TABLESPACE mem_temp_tbs;


User altered.
```

以下为变更用户索引表空间的示例

```
gSQL> ALTER USER u1 INDEX TABLESPACE mem_temp_tbs;


User altered.
```

以下为变更用户schema path的示例

```
gSQL> ALTER USER u1 SCHEMA PATH ( s1, CURRENT PATH );


User altered.
```

兼容性

标准SQL定义了user的概念但未定义user的生成变更及删除相关的SQL语句

参考

相关内容参考以下内容

- **CREATE USER**
- **DROP USER**

8.82 ALTER VIEW

功能

变更视图的定义

语句

```
<alter view statement> ::=

    ALTER VIEW view_name <alter view action>

    ;


<alter view action> ::=

    COMPILE
```

使用范围及访问权限

用户需要有以下权限中的一个才能执行<alter view statement>语句

- 对视图有(ALTER或CONTROL TABLE) ON TABLE
- 对视图所在的SCHEMA有(ALTER TABLE或CONTROL SCHEMA) ON SCHEMA
- ALTER ANY TABLE ON DATABASE

语句规则及参数

view_name

要变更的视图的名称

与schema_name.view_name相同可定义视图所在的SCHEMA省略schema_name时使用执行语句的用户的默认SCHEMA名

COMPILE

重新编译视图

初始化视图Column的注释(COMMENT)

说明

视图引用照的表或视图发生变更或被删除时该视图也受到影响

这些信息可以通过INFORMATION_SCHEMA.VIEWS进行查询

- IS_COMPILEDColumn
  - TRUE : 正常生成视图
  - FALSE: 在存在错误的状态下以FORCE选项生成视图
- IS_AFFECTEDColumn
  - TRUE : 视图引用的表或视图发生了变更
  - FALSE: 生成视图并编译后不变更视图引用的表或视图

使用示例

以下为视图引用的表发生变更后编译受影响的视图的示例

```
gSQL> SELECT TABLE_NAME, IS_AFFECTED
        FROM INFORMATION_SCHEMA.VIEWS
       WHERE TABLE_SCHEMA = 'PUBLIC'
         AND TABLE_NAME = 'V1';

TABLE_NAME IS_AFFECTED
---------- -----------
V1         TRUE

1 row selected.


gSQL> ALTER VIEW v1 COMPILE;

View altered.

COMMIT;

Commit complete.

gSQL> SELECT TABLE_NAME, IS_AFFECTED
```

```
        FROM INFORMATION_SCHEMA.VIEWS
       WHERE TABLE_SCHEMA = 'PUBLIC'
         AND TABLE_NAME = 'V1';


TABLE_NAME IS_AFFECTED
---------- -----------
V1         FALSE


1 row selected.
```

兼容性

标准SQL未定义<alter view statement>语句

参考

相关内容参考下文

- **CREATE VIEW**
- **DROP VIEW**

8.83 ANALYZE SYSTEM

功能

控制系统的统计信息

语句

```
<analyze system statement> ::=

    ANALYZE SYSTEM [ <analyze action> ]

    ;


<analyze action> ::=

      COMPUTE STATISTICS

    | DELETE STATISTICS
```

使用范围及访问权限

用户需要有ANALYZE ANY ON DATABASE权限才能执行<analyze system statement>语句

语句规则及参数

<analyze action>

省略时默认值为COMPUTE STATISTICS

COMPUTE STATISTICS

建立系统相关的如下统计信息

- CPU_OPS (Operations Per Second)
  - CPU每秒处理的operation数量
- NETWORK_IOPS (I/O operations Per Second)
  - Cluster的情况下有效
  - 每秒可处理的Network I/O 次数
- BUFFER_MISS_PERCENT
  - 磁盘buffer miss概率

DELETE STATISTICS

删除系统统计信息

说明

建立的系统统计信息用于计算查询处理的优化过程的成本

使用示例

以下为通过<analyze system statement>语句建立系统统计信息的示例

```
gSQL> ANALYZE SYSTEM COMPUTE STATISTICS;


analyzed.
```

以下为查询建立的系统统计信息的示例

```
gSQL>
SELECT * FROM DBA_STAT_SYSTEM;


 CPU_OPS NETWORK_IOPS NETWORK_BUFSIZE LAST_ANALYZED
-------- ------------ --------------- --------------------------
53000412         2914           65536 2017-03-30 16:49:42.200000


1 row selected.
```

兼容性

标准SQL未定义统计信息相关概念

参考

相关内容参考**ANALYZE SYSTEM**

8.84 ANALYZE TABLE

功能

控制表的统计信息

语句

```
<analyze table statement> ::=
    ANALYZE TABLE table_name
    [ <parallel clause> ]
    [ <analyze action> ]
    ;

<parallel clause> ::=
      NOPARALLEL
    | PARALLEL [thread_count]

<analyze action> ::=
      COMPUTE STATISTICS [ <for_clause> ]
    | ESTIMATE STATISTICS <sample_clause> [ <for_clause> ]
    | DELETE STATISTICS
```

```
<sample_clause>

      SAMPLE row_count ROWS

    | SAMPLE percentage PERCENT


<for_clause>

      FOR ALL COLUMNS

    | FOR ALL INDEXED COLUMNS

    | FOR COLUMNS column_name [, ...]

    | FOR ALL INDEXES

    | FOR INDEXES index_name [, ...]
```

使用范围及访问权限

用户需要拥有ANALYZE ANY ON DATABASE权限才能执行<analyze table statement>语句

语句规则及参数

table_name

表名称

与schema_name.view_name相同可定义视图所在的SCHEMA省略schema_name时使用执行语句的用户的默认SCHEMA名

<parallel clause>

指定分析过程使用的线程的数量

未指定时默认值为PARALLEL

- NOPARALLEL
  - 不并列分析
- PARALLEL [thread_count]
  - 并列分析
  - thread_count可使用从0开始的值最大值为64
  - thread_count为0或省略时取决于系统的CPU数量

<analyze action>

省略时默认值为COMPUTE STATISTICS

COMPUTE STATISTICS

通过全数检查建立如下所示的表相关统计信息

- 表统计信息
  - row count
  - 页面数量
- 各column的统计信息
  - 互不相同的值的数量
  - NULL值的数量

    - 值得平均长度
    - 最小值
    - 最大值
- 索引统计信息
    - 互不相同的key的数量
    - 页面数量
    - Leaf页面数量
    - Tree level
    - 索引集群因子(clustering factor)

根据column的数据类型建立如下统计信息

数据类型	NUM_DISTINCT	NUM_NULLS	AVG_LENGTH	MIN/MAX
BOOLEAN	O	O	O	X
NATIVE_SMALLINT	O	O	O	O
NATIVE_INTEGER	O	O	O	O
NATIVE_BIGINT	O	O	O	O
NATIVE_REAL	O	O	O	O
NATIVE_DOUBLE	O	O	O	O
NUMBER	O	O	O	O
NUMERIC	O	O	O	O
FLOAT	O	O	O	O

数据类型	NUM_DISTINCT	NUM_NULLS	AVG_LENGTH	MIN/MAX
CHAR(n)	0	0	0	64bytes以下时建立
VARCHAR(n)	0	0	0	64bytes以下时建立
LONG VARCHAR	X	X	X	X
BINARY	0	0	0	X
VARBINARY	0	0	0	X
LONG VARBINARY	X	X	X	X
DATE	0	0	0	0
TIME	0	0	0	0
TIMESTAMP	0	0	0	0
INTERVAL	0	0	0	0
ROWID	0	0	0	X

Table 8-22 根据数据类型建立的统计信息

ESTIMATE STATISTICS <sample_clause>

使用指定的与<sample_clause>数量相同的样品构建column与索引的统计信息

- SAMPLE row_count ROWS

- 使用与指定row数量相同的样品
- row_count为大于0的正整数

- SAMPLE percentage PERCENT
  - 使用指与定比例相同的样品数量
  - percentage为1~99范围内的正整数

抽样row的数量小于**MIN_SAMPLE_ROW_COUNT**参数值时遵循参数值

<for_clause>

省略时构建可建立统计信息的所有column与所有索引的统计信息

FOR ALL COLUMNS

构建可建立统计信息的所有column的统计信息

不构建索引统计信息

FOR ALL INDEXED COLUMNS

构建包含在索引的所有column的统计信息

不构建除此之外的column统计信息

不构建索引统计信息

FOR COLUMNS column_name [, ...]

构建列出的column的统计信息

不构建未描述的column的统计信息

不构建索引统计信息

FOR ALL INDEXES

构建所有索引的统计信息

不构建column统计信息

FOR INDEXES index_name [, ...]

构建列出的索引的统计信息

不构建未描述的索引的统计信息

不构建column统计信息

DELETE STATISTICS

删除表的统计信息

说明

表统计信息是影响查询优化准确度的重要信息

表的数据量与统计信息的构建时间成正比因此数据量较多时可以通过抽样构建统计信息或只对影响查询的主要信息构建统计信息

- 以下为使用抽样建立统计信息的示例

```
ANALYZE TABLE lineitem ESTIMATE STATISTICS SAMPLE 10 PERCENT;
```

- 以下为仅对主要column与索引建立统计信息的示例

```
ANALYZE TABLE lineitem COMPUTE STATISTICS FOR ALL INDEXED COLUMNS;

ANALYZE TABLE lineitem COMPUTE STATISTICS FOR ALL INDEXES;
```

使用示例

以下为通过全数检查构建统计信息的示例

```
gSQL> ANALYZE TABLE orders;


Table analyzed.
```

以下为查询已构建的表的统计信息的示例

```
gSQL>
SELECT
        TABLE_NAME
      , NUM_ROWS
```

```
  FROM
       DICTIONARY_SCHEMA.USER_TABLES
 WHERE
       TABLE_SCHEMA = 'PUBLIC'
   AND TABLE_NAME   = 'ORDERS'
;


TABLE_NAME NUM_ROWS
---------- --------
ORDERS      1500000


1 row selected.



gSQL>
SELECT
       TABLE_NAME
     , COLUMN_NAME
     , NUM_DISTINCT
     , NUM_NULLS
     , LOW_VALUE
     , HIGH_VALUE
  FROM
       DICTIONARY_SCHEMA.USER_TAB_COLUMNS
 WHERE
```

```
       TABLE_SCHEMA = 'PUBLIC'
   AND TABLE_NAME   = 'ORDERS'
;


TABLE_NAME COLUMN_NAME     NUM_DISTINCT NUM_NULLS LOW_VALUE
HIGH_VALUE
---------- --------------- ------------ --------- ------------------- ----
---------------
ORDERS     O_ORDERKEY           1500000         0 1
6000000
ORDERS     O_CUSTKEY              99996         0 1
149999
ORDERS     O_ORDERSTATUS              3         0 F                   P
ORDERS     O_TOTALPRICE         1464556         0 857.71
555285.16
ORDERS     O_ORDERDATE             2406         0 1992-01-01 00:00:00
1998-08-02 00:00:00
ORDERS     O_ORDERPRIORITY            5         0 1-URGENT            5-
LOW
ORDERS     O_CLERK                 1000         0 Clerk#000000001
Clerk#000001000
ORDERS     O_SHIPPRIORITY             1         0 0                   0
ORDERS     O_COMMENT            1482071         0 null                null


9 rows selected.
```

```
gSQL>
SELECT
       TABLE_NAME
     , INDEX_NAME
     , DISTINCT_KEYS
  FROM
       DICTIONARY_SCHEMA.USER_INDEXES
 WHERE
       TABLE_SCHEMA = 'PUBLIC'
   AND TABLE_NAME   = 'ORDERS'
;

TABLE_NAME INDEX_NAME        DISTINCT_KEYS
---------- ----------------- -------------
ORDERS     ORDERS_PK_INDEX         1500000
ORDERS     ORDERS_CUSTKEY_FK         99996

2 rows selected.
```

兼容性

标准SQL未定义统计信息相关概念

参考

相关内容参考**ANALYZE SYSTEM**

8.85 AUDIT POLICY

功能

激活audit policy

语句

```
<audit policy statement> ::=
    AUDIT POLICY policy_name
    [ <specified_user_option> ]
    [ <specified_success_option> ]
    ;

<specified_user_option> ::=
      BY user_name [, ...]
    | EXCEPT user_name [, ...]

<specified_success_option> ::=
      WHENEVER SUCCESSFUL
    | WHENEVER NOT SUCCESSFUL
```

使用范围及访问权限

用户需要有AUDIT SYSTEM ON DATABASE权限才能执行<audit policy statement>语句

语句规则及参数

policy_name

要激活的audit policy对象名

已激活的audit policy不影响原有的会话仅影响新创建的会话

<specified_user_option>

指定要执行审计的用户

省略时对所有用户执行审计

在同一个audit policy中无法同时使用BY子句与EXCEPT子句

- BY user_list: 特定要执行审计的用户时使用BY子句
- EXCEPT user_list: 要排除特定用户并对其他用户执行审计时使用EXCEPT子句

<specified_success_option>

- WHENEVER SUCCESSFUL

    - action成功时创建audit record
- WHENEVER NOT SUCCESSFUL
    - action失败时创建audit record
- 省略时成功/失败均创建audit record

说明

激活audit policy不影响原有的会话对新创建的会话开始执行审计

查询audit record

符合审计条件时创建audit record如下可通过DICTIONARY_SCHEMA.AUDIT_TRAIL view查询

```
SELECT logon_user
     , event_timestamp
     , action_name
     , object_name
     , sql_text
  FROM audit_trail
 WHERE policy_name = 'P1'
;
```

普通用户查询AUDIT_TRAIL需要有SELECT权限

```
GRANT SELECT ON DICTIONARY_SCHEMA.AUDTI_TRAIL TO user_name;
```

Audit Policy信息查询

可通过DICTIONARY_SCHEMA.AUDIT_POLICY_OPTIONS views查询audit policy对象信息

```
SELECT policy_name
     , audit_option
     , object_schema
     , object_name
  FROM audit_policy_options
;
```

可通过DICTIONARY_SCHEMA.AUDIT_POLICY_ENABLED view查询audit policy对象的激活信息

```
SELECT policy_name
     , enabled_opt
     , user_name
     , when_success
     , when_failure
  FROM audit_policy_enabled
```

使用BY子句与EXCEPT子句时的注意事项

对相同的audit policy使用多个AUDIT POLICY BY子句时激活user的集合

即以下两个示例的意义相同

- 例1：对u1u2激活p1 audit policy

```
AUDIT POLICY p1 BY u1;

AUDIT POLICY p1 BY u2;
```

- 例2：对u1u2激活p1 audit policy

```
AUDIT POLICY p1 BY u1, u2;
```

对相同的audit policy使用多个AUDIT POLICY EXCEPT子句时仅最后一个AUDIT POLICY有效

即以下两个示例的意义不同

- 例1：仅最后一个语句有效排除u2激活p1 audit policy

```
AUDIT POLICY p1 EXCEPT u1;

AUDIT POLICY p1 EXCEPT u2;
```

- 例2：排除u1与u2激活p1 audit policy

```
AUDIT POLICY p1 EXCEPT u1, u2;
```

对相同的audit policy无法同时使用BY与EXCEPT

- 使用BY子句激活audit policy时其后续只能使用BY子句

```
AUDIT POLICY p1 BY u1;
```

  - Error

```
AUDIT POLICY p1 EXCEPT u2;
```

- 使用EXCEPT子句激活audit policy时其后续只能使用EXCEPT子句

```
AUDIT POLICY p1 EXCEPT u1;
```

  - Error：属于by all users

```
AUDIT POLICY p1;
```

将使用BY激活的audit policy转换为EXCEPT或将使用EXCEPT激活的audit policy转换为BY需要禁用已激活的audit policy后进行变更

如下通过NOAUDIT POLICY语句禁用

- AUDIT POLICY p1 BY u1, u2;
  - NOAUDIT POLICY p1 BY u1, u2;
- AUDIT POLICY p1;
  - NOAUDIT POLICY p1;
- AUDIT POLICY p1 EXCEPT u1, u2;
  - NOAUDIT POLICY p1;
  - NOAUDIT POLICY语句无EXCEPT option

与BY子句同时使用的WHENEVER子句会被累计

以下两个示例的意义相同

- 例1：与成功/失败无关生成audit record

```
AUDIT POLICY p1 BY u1 WHENEVER SUCCESSFUL;
```

```
AUDIT POLICY p1 BY u1 WHENEVER NOT SUCCESSFUL;
```

- 例2：与成功/失败无关生成audit record

```
AUDIT POLICY p1 BY u1;
```

WHENEVER子句与EXCEPT子句同时使用时仅最后的WHENEVER子句有效

以下两个示例的意义不同

- 例1：失败时生成audit record

```
AUDIT POLICY p1 EXCEPT u1 WHENEVER SUCCESSFUL;

AUDIT POLICY p1 EXCEPT u1 WHENEVER NOT SUCCESSFUL;
```

- 例2：与成功/失败无关生成audit record

```
AUDIT POLICY p1 EXCEPT u1;
```

使用示例

以下为对所有用户激活audit policy的示例

```
AUDIT POLICY table_pol;
```

可通过以下查询查看激活信息

```
SELECT policy_name
     , enabled_opt
     , user_name
  FROM audit_policy_enabled
 WHERE policy_name = 'TABLE_POL';


POLICY_NAME  ENABLED_OPT  USER_NAME
-----------  -----------  ---------
TABLE_POL    BY           ALL USERS
```

以下为激活特定用户的audit policy的示例

```
AUDIT POLICY dml_pol BY u1, u2;
```

以下为排除特定用户后激活audit policy的示例

```
AUDIT POLICY read_seq_pol EXCEPT sys;
```

以下为审计特定用户的SQL语句失败时的示例

```
AUDIT POLICY delete_pol BY u1 WHENEVER NOT SUCCESSFUL;
```

兼容性

标准SQL没有audit policy

参考

相关内容参考下文

- Audit policy 对象管理
  - **CREATE AUDIT POLICY**
  - **DROP AUDIT POLICY**
  - **ALTER AUDIT POLICY**
- Audit policy 激活/禁用
  - **AUDIT POLICY**
  - **NOAUDIT POLICY**
- Audit trail 查询: **AUDIT_TRAIL**
- Audit trail 清除: **ALTER DATABASE CLEAR AUDIT TRAIL**

9.SQL References (C~G)

9.1 CLOSE cursor_name

功能

关闭游标

语句

```
<close statement> ::=

    CLOSE cursor_name

    ;
```

语句规则及参数

cursor_name

应打开游标

应为在会话中通过**DECLARE cursor_name**语句声明的游标

说明

游标是会话内的对象不影响其他会话的游标

使用示例

以下为使用interactive SQL tool (gsql)将游标DECLAREOPENFETCHCLOSE的示例

```
gSQL> DECLARE cur1 CURSOR FOR SELECT id, data FROM t1;


Cursor declared.


gSQL> OPEN cur1;


Cursor is open.


gSQL> \var v_id   INTEGER
gSQL> \var v_data VARCHAR(128)


gSQL> FETCH cur1 INTO :v_id, :v_data;


V_ID V_DATA
---- ------
   1 data_1
```

```
1 row fetched.



gSQL> FETCH cur1 INTO :v_id, :v_data;

V_ID V_DATA
---- ------
   2 data_2

1 row fetched.

gSQL> FETCH cur1 INTO :v_id, :v_data;

no rows fetched.

gSQL> CLOSE cur1;

Cursor closed.
```

兼容性

Feature ID	说明	是否支持
B031	Basic dynamic SQL	O

Table 9-1 标准SQL兼容性

参考

相关内容参考下文

- **DECLARE cursor_name**
- **OPEN cursor_name**
- **FETCH cursor_name**

9.2 COMMENT ON name IS

功能

将对象的说明存储于字典（dictionary）

语句

```
<comment statement> ::=

    COMMENT ON <comment object> IS 'comment string'

    ;


<comment object> ::=

      CLUSTER GROUP group_name

    | CLUSTER MEMBER member_name

    | DATABASE

    | PROFILE profile_name

    | AUDIT POLICY policy_name

    | AUTHORIZATION user_name

    | TABLESPACE tablespace_name

    | SCHEMA schema_name

    | TABLE [schema_name].table_name

    | COLUMN [schema_name].table_name.column_name
```

```
| INDEX [schema_name].index_name
| SEQUENCE [schema_name].sequence_name
| CONSTRAINT [schema_name].constraint_name
| PROCEDURE [schema_name].procedure_name
```

使用范围及访问权限

需要对各对象如下变更权限才能执行<comment statement>语句

- CLUSTER GROUP
  - ALTER SYSTEM ON DATABASE
- CLUSTER MEMBER
  - ALTER SYSTEM ON DATABASE
- DATABASE
  - ALTER DATABASE ON DATABASE
- PROFILE
  - ALTER PROFILE ON DATABASE
- AUDIT POLICY
  - AUDIT SYSTEM ON DATABASE
- AUTHORIZATION (user)
  - ALTER USER ON DATABASE
- AUTHORIZATION (role)
  - ALTER ROLE ON DATABASE
- TABLESPACE
  - ALTER TABLESPACE ON DATABASE

- SCHEMA : 需要有以下权限中的一个
  - schema的所有者
  - 对schema有CONTROL SCHEMA ON SCHEMA
  - ALTER SCHEMA ON DATABASE
- TABLE : 需要有以下权限中的一个
  - 表的所有者
  - 对表有CONTROL TABLE ON TABLE
  - 对表所属的schema有CONTROL SCHEMA ON SCHEMA
  - ALTER ANY TABLE ON DATABASE
- COLUMN : 需要有以下权限中的一个
  - column所属的表的所有者
  - 对column所属的表有CONTROL TABLE ON TABLE
  - 对column所属的表的schema有CONTROL SCHEMA ON SCHEMA
  - ALTER ANY TABLE ON DATABASE
- INDEX : 需要有以下权限中的一个
  - 索引的所有者
  - 对索引所属的schema有CONTROL SCHEMA ON SCHEMA
  - ALTER ANY INDEX ON DATABASE
- SEQUENCE
  - 序列的所有者
  - 对序列所属的schema有CONTROL SCHEMA ON SCHEMA
  - ALTER ANY SEQUENCE ON DATABASE
- CONSTRAINT
  - 约束条件的所有者
  - 对约束条件所属的schema有CONTROL SCHEMA ON SCHEMA

    - ALTER ANY TABLE ON DATABASE
- PROCEDURE
    - Stored Procedure/Function的所有者
    - 对Stored Procedure / Function所属的schema有CONTROL SCHEMA ON SCHEMA
    - ALTER ANY PROCEDURE ON DATABASE

语句规则及参数

<comment object>

作为存储注释的目标对象可以存储以下数据库对象的注释

- Cluster object
    - CLUSTER GROUP
    - CLUSTER MEMBER
- Non-schema object
    - DATABASE
    - PROFILE
    - AUDIT POLICY
    - AUTHORIZATION(User or Role)
    - TABLESPACE
    - SCHEMA
- Schema object
    - TABLE 或 VIEW
    - COLUMN

- INDEX
- SEQUENCE
- CONSTRAINT
- PROCEDURE 或 FUNCTION

Schema object中如果不指定schema_name则SCHEMA名取决于执行语句的用户的 **Schema Path**

```
COMMENT ON TABLE test_table IS 'test comment';

→ COMMENT ON TABLE user_default_schema.test_table IS 'test comment';
```

'comment string'

描述要存储的注释内容

如下使用empty string ('')删除注释

```
COMMENT ON TABLE test_table IS '';
```

Comment string的长度不能超过1024 bytes

说明

从dictionary view的COMMENTS column可以查看各对象类型的注释内容

- Cluster objects
  - CLUSTER GROUP & CLUSTER MEMBER
    - DICTIONARY_SCHEMA.DBA_CLUSTER_COMMENTS view

- Non-schema objects
  - DATABASE
    - DICTIONARY_SCHEMA.ALL_NONSCHEMA_COMMENTS view
    - DICTIONARY_SCHEMA.DBA_NONSCHEMA_COMMENTS view
  - PROFILE
    - DICTIONARY_SCHEMA.DBA_NONSCHEMA_COMMENTS view
  - AUDIT POLICY
    - DICTIONARY_SCHEMA.AUDIT_POLICIES view
  - USER
    - DICTIONARY_SCHEMA.ALL_NONSCHEMA_COMMENTS view
    - DICTIONARY_SCHEMA.DBA_NONSCHEMA_COMMENTS view
  - TABLESPACE
    - DICTIONARY_SCHEMA.ALL_NONSCHEMA_COMMENTS view
    - DICTIONARY_SCHEMA.DBA_NONSCHEMA_COMMENTS view
  - SCHEMA
    - DICTIONARY_SCHEMA.ALL_NONSCHEMA_COMMENTS view
    - DICTIONARY_SCHEMA.DBA_NONSCHEMA_COMMENTS view
- SQL schema objects
  - TABLE
    - DICTIONARY_SCHEMA.USER_TAB_COMMENTS view
    - DICTIONARY_SCHEMA.ALL_TAB_COMMENTS view
    - DICTIONARY_SCHEMA.DBA_TAB_COMMENTS view
  - VIEW
    - DICTIONARY_SCHEMA.USER_TAB_COMMENTS view
    - DICTIONARY_SCHEMA.ALL_TAB_COMMENTS view

        - DICTIONARY_SCHEMA.DBA_TAB_COMMENTS view
    - COLUMN
        - DICTIONARY_SCHEMA.USER_COL_COMMENTS view
        - DICTIONARY_SCHEMA.ALL_COL_COMMENTS view
        - DICTIONARY_SCHEMA.DBA_COL_COMMENTS view
    - INDEX
        - DICTIONARY_SCHEMA.USER_INDEXES view
        - DICTIONARY_SCHEMA.ALL_INDEXES view
        - DICTIONARY_SCHEMA.DBA_INDEXES view
    - SEQUENCE
        - DICTIONARY_SCHEMA.USER_SEQUENCES view
        - DICTIONARY_SCHEMA.ALL_SEQUENCES view
        - DICTIONARY_SCHEMA.DBA_SEQUENCES view
    - CONSTRAINT
        - DICTIONARY_SCHEMA.USER_CONSTRAINTS view
        - DICTIONARY_SCHEMA.ALL_CONSTRAINTS view
        - DICTIONARY_SCHEMA.DBA_CONSTRAINTS view
    - PROCEDURE & FUNCTION
        - DICTIONARY_SCHEMA.USER_PROCEDURES view
        - DICTIONARY_SCHEMA.ALL_PROCEDURES view
        - DICTIONARY_SCHEMA.DBA_PROCEDURES view

各个视图的详细内容参考**DICTIONARY_SCHEMA**

使用示例

以下为在表设置注释的示例

```
gSQL> COMMENT ON TABLE t1 IS 'test comment on table t1';


Comment created.
```

以下为在column设置注释的示例

```
gSQL> COMMENT ON COLUMN t1.id IS 'test comment on column t1.id';


Comment created.
```

以下在SCHEMA设置注释的示例

```
gSQL> COMMENT ON SCHEMA s1 IS 'test comment on schema s1';


Comment created.
```

兼容性

标准SQL没有<comment statement>

9.3 COMMIT

功能

结束当前事务将所有变更内容持久化

语句

```
<commit statement> ::=
    COMMIT [ WORK ]
       [ [ <commit comment clause> ] [ <commit write clause> ] |
         [ <commit force clause> ] [ <commit comment clause> ] ]
    ;

<commit comment clause> ::=
      COMMENT 'comment_string'

<commit write clause> ::=
      WRITE [ WAIT | NOWAIT ]

<commit force clause> ::=
    FORCE 'xid_string'
```

语句规则及参数

WORK

不影响操作的保留字

<commit comment clause>

- COMMENT 'comment_string'
    - 提交事务时在事务指定注释

<commit write clause>

设置是否等待直到commit运算生成的重做日志完成记录到日志文件

- WAIT
    - 等待commit运算生成的重做日志完成记录到日志文件后结束运算
- NOWAIT
    - commit运算生成的重做日志记录到日志缓冲即结束运算
- 未指定时遵循参数

<commit force clause>

用于手动提交分布式事务

- FORCE 'xid_string'

- 提交属于'xid_string'的分布式事务
- 'xid_string'由'*format_id.transaction_id.branch_id*'组成

说明

COMMIT语句结束事务内执行的如下语句

- Data Manipulation Language (DML)语句
  - 变更数据的INSERTUPDATEDELETE等语句
- Data Definition Language (DDL) 语句
  - 变更对象的结构及定义的CREATEDROPALTERTRUNCATEGRANTREVOKE等语句

另外DDL中使用操作系统资源或变更数据类型的如下语句是自动提交的

- **CREATE TABLESPACE**
- **DROP TABLESPACE**
- **ALTER TABLESPACE**
- ALTER TABLE .. ALTER COLUMN .. SET DATA TYPE: **<alter column data type clause>**

执行COMMIT时将自动关闭以WITHOUT HOLD选项打开的游标游标相关的详细内容参考以下语句

- **DECLARE cursor_name**
- **OPEN cursor_name**

事务违反延时的(DEFERRED)约束条件时COMMIT语句失败并回滚事务延时约束条件相关的详细

内容参考**SET CONSTRAINTS** 语句

使用示例

以下为执行插入语句后commit的示例

```
gSQL> INSERT INTO t1 VALUES ( 1, 'anonymous' );


1 row created.


gSQL> COMMIT WORK COMMENT 'INSERT T1';


Commit complete.
```

兼容性

Feature ID	说明	是否支持
T261	Chained transactions	X

Table 9-2 标准SQL兼容性

参考

相关内容参考下文

- **ROLLBACK**
- **SAVEPOINT savepoint_specifier**

9.4 CREATE AUDIT POLICY

功能

创建audit policy对象

需执行AUDIT POLICY语句才能激活创建的audit policy

语句

```
<audit policy definition> ::=
    CREATE AUDIT POLICY policy_name
    { <privilege_audit_clause> |  <action_audit_clause> |
<privilege_audit_clause> <action_audit_clause> }
    ;

<privilege_audit_clause> ::=
    PRIVILEGES <database_privilege> [, ...]

<action_audit_clause> ::=
    ACTIONS { <object_action_audit> | <system_action_audit> } [, ...]

<object_action_audit> ::=
      ALL ON [schema_name.]object_name
```

```
    | <object_action> ON [schema_name.]object_name


<system_action_audit> ::=
      ALL
    | DDL
    | <system_action>
```

使用范围及访问权限

用户需要有AUDIT SYSTEM ON DATABASE权限才能执行<audit policy definition> 语句

语句规则及参数

policy_name

要创建的audit policy的名称

<privilege_audit_clause>

权限审计对使用database privilege成功执行SQL语句的情况进行审计

可审计特定用户使用database privilege执行的SQL语句对数据库的所有者SYS用户不记录权限审计记录

以下为对u1用户赋予SELECT ANY TABLE权限并激活audit policy的示例

```
CREATE AUDIT POLICY p1
       PRIVILEGES SELECT ANY TABLE;


AUDIT POLICY p1;
```

用户u1如下执行SQL语句时权限审计的运行会有所不同

- SELECT * FROM u1.t1;
  - 以u1.t1表的所有者权限执行SQL语句因此不创建audit record
- SELECT * FROM u2.t1;
  - 以SELECT ANY TABLE权限执行SQL语句因此创建audit record

可用如下语句查询权限审计中可指定的<database_privilege>

```
SELECT PRIVILEGE_NAME FROM V$AUDITABLE_DB_PRIVILEGES;
```

<action_audit_clause>

审计特定对象的action与数据库整体的action

<object_action_audit>

ALL ON object_name

指属于object_name的对象的所有可列出的action

各对象类型中可审计的audit action如下表

Object type	Action
Table	ALTER, COMMENT, DELETE, GRANT, INDEX, INSERT, LOCK, RENAME, SELECT, UPDATE
View	ALTER, COMMENT, GRANT, SELECT
Sequence	ALTER, COMMENT, GRANT, SELECT
Stored function/ procedure	ALTER, COMMENT, EXECUTE, GRANT

Table 9-3 各对象的audit action

<object_action> ON object_name

如下可指定ON子句并一个一个列出特定object的个别action

```
CREATE AUDIT POLICY p1
       ACTIONS INSERT ON u1.t1
             , DELETE ON u1.t1
             , UPDATE ON u1.t1
;
```

EXECUTE action注意事项

对stored function或stored procedure的EXECUTE action成功失败与否的审计仅通过是否可在实际执行时间点执行来进行判断

- WHENEVER NOT SUCCESSFUL时，无法执行stored function/ procedure时创建审计记录
- WHENEVER SUCCESSFUL时，即使在stored function/ procedure内部执行SQL语句时出错也创建审计记录
- 如需要审计stored function/ procedure内部的SQL语句失败需将该SQL语句包含在审计对象中

<system_action_audit>

与特定对象无关审计数据库产生的系统action

- <system_action>

可通过以下语句查询有效的系统action

```
SELECT ACTION_NAME FROM V$AUDITABLE_SYSTEM_ACTIONS;
```

- ALL

指所有系统action

- DDL

指所有Data Definition Language (DDL)语句

说明

Audit policy为定义要审计的对象的对象

需要执行AUDIT POLICY语句以激活audit policy

可定义并激活多个audit policy但推荐维持限制数量的audit policy

推荐捆绑多个小块policy组成少量policy群组

如下可通过AUDIT_POLICY_OPTIONS view查询创建的audit policy对象的选项信息

```
SELECT audit_option
     , audit_option_type
     , object_schema
     , object_name
  FROM audit_policy_options
 WHERE policy_name = 'P1'
;

AUDIT_OPTION AUDIT_OPTION_TYPE OBJECT_SCHEMA  OBJECT_NAME
------------ ----------------- -------------- ------------
DELETE         OBJECT ACTION     U1           T1
INSERT         OBJECT ACTION     U1           T1
UPDATE         OBJECT ACTION     U1           T1
```

生成audit record

产生符合多个审计条件的action时创建一个或多个audit record

如下列出类似的audit potion时创建一个audit record

- 定义audit policy

```
CREATE AUDIT POLICY p1
       PRIVILEGES SELECT ANY TABLE
       ACTIONS SELECT;

AUDIT POLICY p1;
```

- 执行audit action

```
SELECT * FROM other_user.t1;
```

如下列出不同的audit option时创建两个audit record

- 定义audit policy

```
CREATE AUDIT POLICY p1
       ACTIONS SELECT ON u1.t1
             , SELECT ON u2.t2;

AUDIT POLICY p1;
```

- 执行audit action

```
SELECT COUNT(*) FROM u1.t1 A, u2.t2 B WHERE A.id = B.id;
```

如下对相同的action激活多个audit policy时创建两个audit record

- 定义audit policy

```
CREATE AUDIT POLICY p1
       PRIVILEGES SELECT ANY TABLE;
AUDIT POLICY p1;


CREATE AUDIT POLICY p2
       ACTIONS SELECT;
AUDIT POLICY p2;
```

- 执行audit action

```
SELECT * FROM other.t1;
```

使用示例

以下为定义审计权限的audit policy的示例

```
CREATE AUDIT POLICY policy_table
       PRIVILEGES CREATE ANY TABLE
                , DROP ANY TABLE
;
```

以下为定义审计对象的action的audit policy的示例

```
CREATE AUDIT POLICY policy_dml
       ACTIONS INSERT ON u1.t1
             , DELETE ON u1.t1
             , UPDATE ON u1.t1
             , ALL    ON u1.t2
;
```

以下为定义审计系统action的audit policy的示例

```
CREATE AUDIT POLICY policy_drop
       ACTIONS DROP TABLE, TRUNCATE TABLE
;
```

以下为定义集合上述所有示例的audit policy的示例

```
CREATE AUDIT POLICY policy_group
       PRIVILEGES CREATE ANY TABLE
                , DROP ANY TABLE
       ACTIONS INSERT ON u1.t1
             , DELETE ON u1.t1
             , UPDATE ON u1.t1
             , ALL    ON u1.t2
             , DROP TABLE
             , TRUNCATE TABLE
;
```

兼容性

标准SQL无Audit Policy

参考

相关内容参考下文

- Audit policy 对象管理
  - **CREATE AUDIT POLICY**
  - **DROP AUDIT POLICY**
  - **ALTER AUDIT POLICY**
- Audit policy 激活/禁用
  - **AUDIT POLICY**
  - **NOAUDIT POLICY**
- Audit trail 查询: **AUDIT_TRAIL**
- Audit trail 清除: **ALTER DATABASE CLEAR AUDIT TRAIL**

9.5 CREATE CLUSTER GROUP

功能

创建参与集群系统的集群组

语句

```
<cluster group definition> ::=
    CREATE CLUSTER GROUP group_name
        <cluster member definition> [, ...]
    ;

<cluster member definition> ::=
    CLUSTER MEMBER member_name <connection attribute> [<member position>]

<connection attribute> ::=
    HOST 'address' PORT port_no

<member position> ::=
    POSITION DEFAULT
  | POSITION MAX
  | POSITION number
```

使用范围及访问权限

可在集群系统中执行

用户需要有ADMINISTRATION ON DATABASE权限才能执行<cluster group definition>语句

语句规则及参数

group_name

集群组的名称

不可存在相同的集群组集群成员名称

名称长度应小于128字节

<cluster member definition>

定义可包含在集群组的集群成员

集群组最多可包含32个集群成员

集群系统初次创建的集群组仅可定义一个集群成员并自身要包含为集群成员

member_name

集群成员的名称

集群成员的名称应与创建该成员的数据库时定义的成员名称相同

不可有相同的集群组集群成员名

名称长度应小于128字节

集群成员的start-up阶段应为GLOBAL OPEN阶段

<connection attribute>

定义用于集群成员之间通信的连接信息

<connection attribute>应与创建该成员的数据库时定义的HOSTPORT相同

在集群系统中HOST与PORT组合应该是唯一的

- HOST 'address'使用host name或者IPv4地址 使用host name时使用系统的第一个IPv4地址
- PORT port_no应为1024 ~ 49151范围内的值

<member position>

指定cluster member的position number

- POSITION DEFAULT
  - 系统自动指定position number
- POSITION MAX
  - 即使有空的position number也指定新的member position number
  - 指定大于最大member position的值

- POSITION number
  - 指定对应number的position number

    - 该position number应在集群系统中是唯一的
    - 该position number是空的position number应小于或等于最大的position number
- 省略时默认值为POSITION DEFAULT

集群成员的member_position信息可通过DBA_CLUSTER view查询

```
SELECT member_name, member_id, member_position FROM dba_cluster;
```

例如使用如下position number时

- G1N1: 0
- G1N2: 1
- G2N2: 3
- G3N2: 5

根据各选项指定如下值

- POSITION DEFAULT
    - 指定空值2
- POSITION MAX
    - 指定新的position number值6
- POSITION 3
    - 重复因此error
- POSITION 4
    - 指定position number 4

说明

<cluster group definition>不重新分配表的shard

需执行如下语句才能在增加的集群组中重新分配shard

- **ALTER DATABASE REBALANCE**
- **ALTER TABLE name REBALANCE**

使用示例

以下为创建由两个集群成员组成的集群组的示例

```
gSQL>
CREATE CLUSTER GROUP g1
    CLUSTER MEMBER g1n1 HOST '192.168.0.11' PORT 10110
;


Cluster Group created.


gSQL>
ALTER CLUSTER GROUP g1
    ADD CLUSTER MEMBER g1n2 HOST '192.168.0.12' PORT 10120
;
```

```
Cluster Group altered.


gSQL>
CREATE CLUSTER GROUP g2
    CLUSTER MEMBER g2n1 HOST '192.168.0.21' PORT 10210,
    CLUSTER MEMBER g2n2 HOST '192.168.0.22' PORT 10220
;

Cluster Group created.
```

兼容性

标准SQL未定义集群相关概念

参考

相关内容参考下文

- **DROP CLUSTER GROUP**
- **ALTER CLUSTER GROUP name ADD MEMBER**

9.6 CREATE CLUSTER LOCATION

功能

生成集群成员的访问信息

语句

```
<cluster location definition> ::=
    CREATE CLUSTER LOCATION member_name
    <cluster connection attribute>
    ;

<cluster connection attribute> ::
       HOST 'address' PORT port_no
```

使用范围及访问权限

可在集群系统中执行

用户需要有ADMINISTRATION ON DATABASE权限才能执行<cluster location definition>语句

语句规则及参数

member_name

集群成员的名称

注册的集群位置信息中不可有相同的集群成员名称

名称长度应小于128字节

<cluster connection attribute>

定义用于集群成员之间进行通信的连接信息

在集群系统中HOST与PORT组合应该是唯一的

- HOST 'address'使用host name或者IPv4地址 使用host name时使用系统的第一个IPv4地址
- PORT port_no应为1024 ~ 49151范围内的值

说明

一般情况下集群位置信息使用创建集群组或添加集群成员时提供的访问信息自动生成删除集群成员及群组时同时删除生成的信息

集群位置的访问信息发生变更时不需要删除或重新创建集群成员可用**ALTER CLUSTER LOCATION**变更访问信息

使用示例

```
gSQL>
CREATE CLUSTER LOCATION g1n2
    HOST '192.168.0.12' PORT 10120,
;

Created
```

兼容性

标准SQL未定义集群相关概念

参考

相关内容参考**DROP CLUSTER LOCATION**

9.7 CREATE DISK DATA TABLESPACE

功能

定义磁盘数据表空间

语句

```
<disk data tablespace statement> ::=

    CREATE DISK [ DATA ] TABLESPACE tablespace_name

        DATAFILE <disk datafile clause> [, ...]

        [ <data tablespace management clause> [, ...] ]


<disk datafile clause> ::=

     'filename'

        [ SIZE <size clause> | REUSE | SIZE <size clause> REUSE ]

        [ <autoextend clause> ]

        [ AT <domain_name> ]


<autoextend clause>

    AUTOEXTEND { ON [ <next size clause> ] [ <max size clause> ] | OFF }


<next size clause>
```

```
    NEXT <size clause>


<max size clause>

    MAXSIZE { <size clause> | UNLIMITED }


<size clause> ::=

    integer [ K | M | G | T ]


<data tablespace management clause> ::=

      { ONLINE | OFFLINE }

    | EXTSIZE <size clause>
```

使用范围及访问权限

用户需要有CREATE TABLESPACE ON DATABASE权限才能执行<disk data tablespace definition>语句

执行语句的用户对创建的表空间拥有CREATE OBJECT ON TABLESPACE权限

为了在创建的表空间创建对象用户要有以下权限中的一个

- 对该表空间拥有CREATE OBJECT ON TABLESPACE
- USAGE TABLESPACE ON DATABASE

语句规则及参数

tablespace_name

要创建的表空间的名称

表空间名称的长度应小于128字节

<disk datafile clause>

- 'filename'
  - 存储管理数据的文件的名称
  - 存储在磁盘表空间创建的表索引page的空间
  - filename为新的文件或已存在的文件
  - filename的长度应小于1024字节
- SIZE <size clause>
  - 对于新的文件使用SIZE子句指定初始大小
  - 文件已存在时报错
  - 文件的大小可指定为最小1M~最大30G
- REUSE
  - 对于已存在的文件使用REUSE子句
  - 文件不存在时创建新的文件
  - 新创建的文件大小由USER_DATA_TABLESPACE_SIZE参数决定
- SIZE <size clause> REUSE
  - 指定SIZE子句和REUSE子句时根据filename的存在与否如下运行
    - 对于新的filename使用SIZE子句指定初始文件大小

- 对于已存在的filename使用原有的文件通过SIZE子句的值调整大小

<autoextend clause>

将自动扩张属性设置为ON或OFF设置为ON时可指定自动扩张大小和数据文件的最大大小

<next size clause>

指定当前使用中的数据文件中没有可使用的空间时要扩张的大小

<max size clause>

指定数据文件可扩张的最大大小

<size clause>

指定文件的字节大小（未指定时单位为bytes）

- K: Kilobytes
- M: Megabytes
- G: Gigabytes
- T: Terabytes

<domain_name>

执行语句的成员或群组的名称

未指定时在所有群组中执行

ONLINE | OFFLINE

设置表空间ONLINE/ OFFLINE与否

- ONLINE是生成表空间后可立即使用的状态
- OFFLINE是不可使用的状态因此变更为ONLINE之后才可使用

EXTSIZE <size clause>

指定表空间的extent大小

- 以字节为单位描述extent大小选择六个(64 K, 128 K, 256 K, 512 K, 1 M, 2 M)中的一个
- 如果extent大小指定为64k~128k之间的值时设置为128k指定为2M以上时设置为2M

说明

数据表空间是提供存储tableindex (LOGGING)等SQL schema的物理空间的对象

使用示例

以下为创建磁盘表空间的示例

```
gSQL> CREATE DISK TABLESPACE space1 DATAFILE 'test_file_1.dbf' SIZE 10M
REUSE;


Tablespace created.
```

以下为创建由多个数据文件构成的表空间的示例

```
gSQL> CREATE DISK TABLESPACE space1
             DATAFILE 'test_file_3_1.dbf' SIZE 10M REUSE,
                      'test_file_3_2.dbf' SIZE 10M REUSE;


Tablespace created.
```

兼容性

标准SQL未定义表空间相关概念

参考

相关内容参考下文

- **DROP TABLESPACE**
- **ALTER TABLESPACE**
- **ALTER DATABASE DATAFILE AUTOEXTEND**

9.8 CREATE GLOBAL TEMPORARY TABLE

功能

创建新的global temporary table

语句

```
<global temporary table definition> ::=
    CREATE GLOBAL TEMPORARY TABLE table_name
        ( <table element> [, ...] )
        [ <table commit action clause> ]
        [ TABLESPACE tablespace_name ]
    ;

<global temporary table definition: AS query expression> ::=
    CREATE GLOBAL TEMPORARY TABLE table_name
        [ TABLESPACE tablespace_name ]
        AS <query expression> [ WITH [ NO ] DATA ]
    ;

<table commit action clause> ::=
    ON COMMIT { PRESERVE | DELETE } ROWS
```

> Note:
>
> <table element>的定义与<table_definition>的定义相同详细内容参考**CREATE TABLE**

使用范围及访问权限

用户需要满足以下条件才能执行<global temporary table definition>语句

- 创建表的权限
  - 参考**CREATE TABLE**语句的访问权限
- SELECT访问权限
  - 参考**SELECT**语句的访问权限

语句规则及参数

table_name

要创建的表的名称

详细内容参考**table_name**语句

other syntax

其他语句规则参考**CREATE TABLE**及**CREATE TABLE AS SELECT**语句的syntax

说明

GLOBAL TEMPORARY TABLE是用于保管一个事务或会话执行的期间维持的数据的临时表

类似于开发者开发应用程序时在运算的过程中暂时保存数据的变量

GLOBAL TEMPORARY TABLE有以下特点

- 可在所有会话中查看GLOBAL TEMPORARY TABLE的定义
- 定义GLOBAL TEMPORARY TABLE时不分配物理空间在初次插入时分配属于对应会话的实际空间（segment）
- GLOBAL TEMPORARY TABLE数据仅可在插入的会话或事务中查看
- 如下确定保存GLOBAL TEMPORARY TABLE数据的表空间

是否指定表空间	创建表的表空间
指定表空间	在指定的表空间中创建
未指定表空间	在当前会话用户的默认临时表空间中创建

- Global Temporary Table的索引从属于与对应表相同的会话中持续时间也与对应表相同
- 可定义Global Temporary Table的视图
- Global Temporary Table中无法指定描述表的集群相关特性的<table sharding strategy>语句或<table global secondary index clause>语句
- Global Temporary Table中无法指定描述表的物理特性的<table attribute clause>或描述索引的物理特性的<index attribute clause>
- 以Global Temporary Table作为Base Table创建的索引中无法指定描述索引的物理特性的<index attribute clause>

- 由<table commit action clause>结束事务时可确定剩余数据的处理方法

table commit action	说明
ON COMMIT PRESERVE ROWS	即使提交或回滚也维持表中剩余的数据
ON COMMIT DELETE ROWS(default)	提交或回滚的同时删除表中所有剩余的数据（TRUNCATE）

- 支持普通表的大部分DDL（包含ALTERTRUNCATE）
  - 不支持集群相关语句（SHARD以及全局二级索引相关语句等）
  - 使自身的会话或其他会话当前使用中的Global Temporary Table的DDL报错
  - 在使用中的所有会话中用TRUNCATE TABLE或COMMIT等删除所有使用中的Segment后可执行DDL
- 支持普通表的所有DML和Select语句
- Global Temporary Table的所有变更（DML）不记录重做日志
- Global Temporary Table的所有变更（DML）记录Undo日志根据TEMP_UNDO_ENABLED参数确定Undo日志的位置

TEMP_UNDO_ENABLED值	说明
TRUE	在数据库系统的默认临时表空间中记录undo日志
FALSE	在数据库系统的undo表空间中记录undo日志

- Global temporary table的TRUNCATE命令仅truncate当前会话的segment
- 会话终止时TRUNCATE所有的segment后返回

使用示例

以下为执行CREATE GLOBAL TEMPORARY TABLE语句的示例

```
gSQL> CREATE  GLOBAL TEMPORARY TABLE SESSION_TABLE1(
        COL1    CHAR(10)
       ,COL2    VARCHAR2(20)
       ,COL3    NUMBER(10)
)   ON  COMMIT  DELETE ROWS;

Table created.
```

以下为执行CREATE GLOBAL TEMPORARY TABLE ... AS SELECT语句的示例

```
gSQL> CREATE  GLOBAL TEMPORARY TABLE SESSION_TABLE2
    ON  COMMIT  PRESERVE ROWS
    AS  SELECT  *
          FROM  EMPLOYEES;

Table created.
```

兼容性

CREATE GLOBAL TEMPORARY TABLE以及CREATE GLOBAL TEMPORARY TABLE AS SELECT语句遵循标准SQL的<table definition>定义但以下为标准的扩展

- 标准中SELECT语句外面的括号为必选但Sundb中为可选
- 标准中WITH [NO] DATA语句为必选但Sundb中为可选
- Sundb表空间概念是标准中没有的扩展概念

Feature ID	说明	是否支持
T171	LIKE clause in table definition	X
T172	AS subquery clause in table definition	O
F531	Temporary tables	X
S051	Create table of type	X
S043	Enhanced reference types	X
S081	Subtables	X
T173	Extended LIKE clause in table definition	X
T180	System-versioned tables	X
F692	Extended collation support	X
T174	Identity columns	O
T175	Generated columns	X
S071	SQL paths in function and type name resolution	X
F321	User authorization	O
T322	Extended roles	X
F762	CURRENT_CATALOG	O

Feature ID	说明	是否支持
F763	CURRENT_SCHEMA	O

Table 9-4 标准SQL兼容性

参考

相关内容参考下文

- **CREATE TABLE**
- **CREATE TABLE AS SELECT**

9.9 CREATE IMMUTABLE TABLE

功能

创建新的immutable table

语句

```
<immutable table definition> ::=
    CREATE IMMUTABLE TABLE table_name
        ( <table element> [, ...] )
        [ <table sharding strategy> ]
        [ <table attribute clause> [...] ]
        [ TABLESPACE tablespace_name ]
        [ <table global secondary index clause> ]
    ;


<immutable table definition: AS query expression> ::=
    CREATE IMMUTABLE TABLE table_name
        [ ( column_name [, ...] ) ]
        [ <table sharding strategy> ]
        [ <table attribute clause> [, ...] ]
        [ TABLESPACE tablespace_name ]
```

```
    [ <table global secondary index clause> ]
    AS <query expression> [ WITH [ NO ] DATA ]
;
```

Note:

<table element>, <table sharding strategy>, <table attribute clause>, <table global secondary index clause>的定义与<table_definition>的定义相同详细内容参考**CREATE TABLE**

使用范围及访问权限

执行<immutable table definition> 语句需满足以下条件

- 创建表的权限
  - 参考**CREATE TABLE**语句的访问权限
- 访问SELECT的权限
  - 参考**SELECT**语句的访问权限

语句规则及参数

table_name

要创建的表的名称并且在schema内应是唯一的

如schema_name.table_name可定义表所属的schema省略schema_name时使用执行语句的用户的默认schema名

表名称的长度应小于128字节

其他语句规则

其他语句规则参考**CREATE TABLE**和**CREATE TABLE AS SELECT**语句的syntax

说明

Immutable table不仅可以防止变更或删除已存储的记录也可以防止删除整个表

> Caution:
>
> 删除用户schema表空间集群组时也可以删除immutable table

> Note:
>
> 以immutable table创建时不可使用的SQL语句
>
> * UPDATE
> * DELETE
> * ALTER TABLE RENAME/DROP COLUMN
> * ALTER TABLE SET UNUSED COLUMN

* ALTER TABLE DROP UNUSED COLUMNS

* ALTER TABLE RENAME TO

* TRUNCATE TABLE

* DROP TABLE

以immutable table创建时可使用的SQL语句

* INSERT

* SELECT

* SELECT .. FOR UPDATE

* CREATE/ALTER/DROP INDEX

* ALTER TABLE ADD/ALTER COLUMN

* ALTER TABLE ADD/ALTER/RENAME/DROP CONSTRAINT

* ALTER TABLE for physical property changes

* ALTER TABLE ADD/DROP SUPPLEMENTAL LOG

* LOCK TABLE

* ALTER TABLE .. ADD/ALTER/DROP GLOBAL SECONDARY INDEX

* ALTER DATABASE MOVE SHARD

* ALTER TABLE .. MOVE SHARD

* ALTER TABLE .. SPLIT SHARD

* ALTER TABLE .. MERGE SHARD

* DROP TABLESPACE

* DROP SCHEMA

* DROP USER

> * DROP CLUSTER GROUP

使用示例

以下为执行CREATE IMMUTABLE TABLE语句的示例

```
gSQL> CREATE IMMUTABLE TABLE t1
(
    id INTEGER PRIMARY KEY,
    name VARCHAR(128),
    addr VARCHAR(128)
);

Table created.
```

以下为执行CREATE IMMUTABLE TABLE ... AS SELECT语句的示例

```
gSQL> CREATE IMMUTABLE TABLE T2
       AS SELECT *
             FROM T1;

Table created.
```

兼容性

标准SQL未定义CREATE IMMUTABLE TABLE 语句和 CREATE IMMUTABLE TABLE AS SELECT语句

参考

相关内容参考下文

- **CREATE TABLE**
- **CREATE TABLE AS SELECT**

9.10 CREATE INDEX

功能

创建索引

语句

```
<index definition> ::=
    CREATE [ UNIQUE ] INDEX index_name
        ON table_name ( <index column element> [, ...] )
        [ <index attributes> [...] ]
        [ TABLESPACE tablespace_name ]
    ;

<index column element> ::=
    column_name [ ASC | DESC ] [ NULLS FIRST | NULLS LAST ]

<index attributes> ::=
      <physical attribute clause>
    | STORAGE ( <segment attr clause> [...] )
    | <parallel clause>
```

```
<physical attribute clause> ::=

      PCTFREE integer

    | INITRANS integer

    | MAXTRANS integer


<segment attr clause> ::=

      INITIAL <size_clause>

    | NEXT <size_clause>

    | MINSIZE <size_clause>

    | MAXSIZE <size_clause>


<size clause> ::=

      integer [ K | M | G | T ]


<parallel clause> ::=

      NOPARALLEL

    | PARALLEL [ integer ]
```

使用范围及访问权限

用户应满足以下条件才能执行<index definition>语句

- 对创建索引的表需要有以下权限中的一个
  - 对表有(INDEX或CONTROL TABLE) ON TABLE
  - 对表所在的SCHEMA有CONTROL SCHEMA ON SCHEMA

    - CREATE ANY INDEX ON DATABASE
- 对创建索引的SCHEMA需要有以下权限中的一个
    - 对SCHEMA有(CREATE INDEX或CONTROL SCHEMA) ON SCHEMA
    - CREATE ANY INDEX ON DATABASE
- 对创建索引的表空间需要有以下权限中的一个
    - 对表空间有CREATE OBJECT ON TABLESPACE
    - USAGE TABLESPACE ON DATABASE
- 如下决定索引的所有者
    - 索引所属的schema的所有者
    - 索引所属的schema为PUBLIC时执行语句的用户

> Note:
>
> 集群中的unique索引应包含所有sharding key

语句规则及参数

UNIQUE

组成索引的column不能有重复值

index_name

要创建的索引名并且在SCHEMA内唯一

省略SCHEMA名时在引用的表所属的SCHEMA上创建索引

索引名称的长度应小于128byte

table_name

要创建索引的表名称

与schema_name.table_name相同可定义表所在的SCHEMA省略schema_name时使用执行语句的用户的默认SCHEMA名

column_name

用作索引key的Column名

应定义一个以上的Column最多可将32个column用作索引key

根据情况会出现如下约束

- 包含在索引的column数据类型为LONG CHARACTER VARYINGLONG BINARY VARYING时无法创建索引
- Column的precision之和小于1200字节时才可创建索引

ASC | DESC

定义column的排序模式

- ASC：升序排列
- DESC：降序排列
- 省略时默认值为ASC

NULLS FIRST | NULLS LAST

定义空值的排序模式

- NULLS FIRST : 把NULL排在最前面
- NULLS LAST : 把NULL排在最后面
- 未指定时默认为NULLS LAST

<physical attribute clause>

定义索引的物理属性信息

- PCTFREE integer
  - 定义
    - 为了调整key的插入引起的页分割频率而预留的空间
    - 只应用于以bottom-up方式创建索引
  - 可使用0-99之间的值
  - 省略时使用DEFAULT_INDEX_PCTFREE property中设置的值
- INITRANS integer
  - 定义
    - 可同时访问页的初始事务的数量
    - 访问索引的用户少时设置低的INITRANS访问索引的用户多时设置高的INITRANS
    - 必要时会自动扩展至已设置的MAXTRANS
  - 可使用1-32之间的值
  - 省略时默认值为4
- MAXTRANS integer

- 定义
  - 可同时访问页的事务的最大数量
- 可使用1-32之间的值
- 省略时默认值为8

<segment attr clause>

描述索引的存储空间信息

- INITIAL integer
  - 定义
    - 描述索引创建初期分配的物理空间大小
    - 该大小align到表所属的TABLESPACE的EXTENT大小后使用（ex：EXT大小为8192bytes时‘INITIAL 100’实际以8192bytes运行）
    - 该大小（align到TABLESPACE的EXTENT大小的大小）应大于或等于MINEXTENTS的大小小于或等于MAXEXTENTS的大小
  - 最小值为1最大值根据系统环境有所不同
  - 省略时默认值与表所属的TABLESPACE的一个EXTENT的大小相同
- NEXT integer
  - 定义
    - 描述添加索引的物理空间时分配的物理空间大小
    - 该大小align到所属的TABLESPACE的EXTENT大小后使用（ex：EXT大小为8192bytes时‘NEXT 100’实际以8192bytes运行）
    - NEXT根据当前索引可使用的剩余空间的大小（从MAXEXTENTS的大小减去当前使用中的空间的大小）如下运行

        - 剩余空间的大小为0时无法再扩展空间
        - 剩余空间的大小大于0且小于NEXT时分配与剩余空间大小相同的空间
        - 剩余空间的大小大于NEXT时分配与NEXT大小相同的空间
    - 最小值为1最大值根据系统环境有所不同
    - 省略时默认值与表所属的TABLESPACE的一个EXTENT的大小相同
- MINSIZE integer
    - 定义
        - 索引需维持的最小空间大小
        - 该值应小于或等于MAXSIZE的值
    - 该大小align到索引所属的TABLESPACE的EXTENT大小后使用
    - 最小值为1最大值根据系统环境有所不同
    - 如小于2个EXTENT的大小则确定为2个EXTENT的大小
    - 省略时默认值为2个EXTENT的大小
- MAXSIZE integer
    - 定义
        - 可在索引中分配到的最大空间大小
        - 该值应大于或等于MINSIZE的值
    - 该大小align到索引所属的TABLESPACE的EXTENT大小后使用
    - 最小值为1最大值根据系统环境有所不同
    - 省略时默认值为32 terabyte（35,184,372,088,832）
    - 即使指定大于32 terabyte的值也会修改成32 terabyte进行设置

<size clause>

指定文件的字节大小（省略单位时默认为bytes）

- K : kilobytes
- M : megabytes
- G : gigabytes
- T : terabytes

NOPARALLEL | PARALLEL [integer]

指定创建索引时使用的线程数量

- NOPARALLEL
  - 不并行创建索引
- PARALLEL [integer]
  - 并行创建索引
  - 省略integer或指定为0时遵循参数(INDEX_BUILD_PARALLEL_FACTOR)
  - integer可使用0-16范围内的值
  - 参数值为0时系统决定最优值
- 未指定时默认值为PARALLEL

TABLESPACE tablespace_name

指定存储索引的表空间名称

- 指定tablespace_name时
  - tablespace_name为data tablespace时创建LOGGING索引
  - tablespace_name为temporary tablespace 或nologging tablespace时创建NOLOGGING索引

- 省略TABLESPACE子句时
  - 指定USER的INDEX TABLESPACE tablespace_name时
    - 使用定义的表空间
  - USER的INDEX TABLESPACE为NULL时
    - 磁盘表的索引使用用户的默认数据表空间
    - 内存表的索引使用用户的默认临时表空间

说明

LOGGING索引与NOLOGGING索引有以下trade-off

- LOGGING索引
  - 优点：启动系统时使用日志自动恢复索引不需要另外创建
  - 缺点：变更相关row时用日志记录索引的变更内容引发磁盘I/O
- NOLOGGING索引
  - 优点：变更相关row时不会发生索引变更内容的磁盘I/O
  - 缺点：启动数据库时在无索引日志信息的情况下自动重建索引

使用示例

以下为创建unique index的示例

```
gSQL> CREATE UNIQUE INDEX idx_t1_id ON t1( id );
```

```
Index created.
```

以下为对多个column创建索引的示例

```
gSQL> CREATE INDEX idx_t1_id_name ON t1( id, name );


Index created.
```

以下为指定索引column排序模式的示例

```
gSQL> CREATE INDEX idx_t1_dept_id ON t1( dept_id DESC );


Index created.
```

以下为指定索引column的NULL排序模式的示例

```
gSQL> CREATE INDEX idx_t1_name ON t1( name NULLS FIRST );


Index created.
```

以下为设置索引存储空间信息的示例

```
gSQL> CREATE INDEX idx_t1_id ON t1( id )
             STORAGE ( INITIAL 10M NEXT 1M MINSIZE 10M MAXSIZE 100M );


Index created.
```

以下为对索引创建重做日志的示例

```
gSQL> CREATE INDEX idx_t1_id ON t1( id );


Index created.
```

以下为并行执行创建索引的示例

```
gSQL> CREATE INDEX idx_t1_name ON t1( name ) PARALLEL;


Index created.
```

以下为创建索引时指定表空间的示例

```
gSQL> CREATE INDEX idx_t1_name ON t1( name ) TABLESPACE mem_temp_tbs;


Index created.
```

兼容性

标准SQL未定义索引的概念

参考

相关内容参考**DROP INDEX**

9.11 CREATE MEMORY DATA TABLESPACE

功能

定义内存数据表空间

语句

```
<memory data tablespace statement> ::=
    CREATE [ MEMORY ] [ DATA ] TABLESPACE tablespace_name
        DATAFILE <memory datafile clause> [, ...]
        [ <data tablespace management clause> [, ...] ]

<memory datafile clause> ::=
     'filename'
        [ SIZE <size clause> | REUSE | SIZE <size clause> REUSE ]
        [ AT <domain_name> ]
<size clause> ::=
    integer [ K | M | G | T ]

<data tablespace management clause> ::=
      { ONLINE | OFFLINE }
    | EXTSIZE <size clause>
```

使用范围及访问权限

用户需要有CREATE TABLESPACE ON DATABASE权限才能执行<memory data tablespace definition>语句

执行语句的用户对创建的表空间有CREATE OBJECT ON TABLESPACE权限

需要有以下权限中的一个才能在创建的表空间生成对象

- 对表空间有CREATE OBJECT ON TABLESPACE
- USAGE TABLESPACE ON DATABASE

语句规则及参数

[MEMORY] [DATA]

存储表索引等永久性对象的内存表空间

可以省略MEMORY与DATA保留字

tablespace_name

要创建的表空间的名称

表空间名的长度应小于128 byte

<memory datafile clause>

- 'filename'
  - 存储并管理数据的文件名
  - 存储内存数据CHECKPOINT image的空间
  - filename是新的文件或已有的文件
  - filename的长度应小于1024byte
- SIZE <size clause>
  - 为新的文件时通过SIZE指定初始大小
  - 文件已存在时报错
  - 文件大小范围为1M ~ 30G
- REUSE
  - 对于已有的文件使用REUSE
  - 文件不存在时生成新的文件
  - 新创建的文件大小：
    - 数据表空间取决于USER_DATA_TABLESPACE_SIZE参数
    - 临时表空间取决于USER_TEMP_TABLESPACE_SIZE参数
- SIZE <size clause> REUSE
  - 同时指定SIZE与REUSE时根据filename的存在与否如下执行
  - 新的filename使用SIZE指定文件的初始大小
  - 已有的filename使用现有文件将大小调整为SIZE的值

<size clause>

指定文件的byte大小（省略时默认值为bytes）

- K : kilobytes
- M : megabytes
- G : gigabytes
- T : terabytes

<domain_name>

执行语句的成员或组的名称

未指定时在所有组执行

ONLINE | OFFLINE

设置表空间的ONLINE或OFFLINE状态

- ONLINE：表空间创建后可立即使用的状态
- OFFLINE：无法使用的状态因此需要变更为ONLINE状态后才能使用

EXTSIZE <size clause>

指定表空间的extent大小

- extent大小以byte为单位可选择六种(64K, 128K, 256K, 512K, 1M, 2M)中的一个
- 如果extent大小指定为64K ~ 128K之间的值则被设置为128K指定为2M以上则被设置为2M

说明

数据表空间是提供存储tableindex(LOGGING)等SQL schema对象的物理空间的对象

使用示例

以下为创建内存数据表空间的示例

```
gSQL> CREATE TABLESPACE space1 DATAFILE 'test_file_1.dbf' SIZE 10M REUSE;


Tablespace created.
```

以下为创建由多个数据文件组成的表空间的示例

```
gSQL> CREATE TABLESPACE space1
             DATAFILE 'test_file_3_1.dbf' SIZE 10M REUSE,
                      'test_file_3_2.dbf' SIZE 10M REUSE;


Tablespace created.
```

兼容性

标准SQL未定义表空间的概念

参考

相关内容参考下文

- **DROP TABLESPACE**
- **ALTER TABLESPACE**

9.12 CREATE MEMORY TEMPORARY TABLESPACE

功能

定义内存临时表空间

语句

```
<memory temporary tablespace statement> ::=
    CREATE [ MEMORY ] TEMPORARY TABLESPACE tablespace_name
        MEMORY <memory clause> [, ...]
        <temporary tablespace management clause>

<memory clause>
     'memory_name' { SIZE <size clause> } [ AT <domain_name> ]

<temporary tablespace management clause> ::=
    EXTSIZE <size clause>
```

使用范围及访问权限

用户需要有CREATE TABLESPACE ON DATABASE权限才能执行<memory temporary tablespace definition>语句

执行语句的用户对创建的表空间有CREATE OBJECT ON TABLESPACE权限

用户需要有以下权限中的一个才能在创建的表空间创建对象

- 对表空间有CREATE OBJECT ON TABLESPACE
- USAGE TABLESPACE ON DATABASE

语句规则及参数

[MEMORY] TEMPORARY

存储语句处理过程中的中间结果等临时对象或no logging索引的内存临时表空间

可以省略MEMORY保留字

tablespace_name

要创建的表空间的名称

表空间名的长度应小于128 byte

<memory clause>

- 'memory_name'
  - 存储临时数据的内存名
  - memory_name应在对应表空间中是唯一的
  - memory_name的长度应小于1024 byte
- SIZE <size clause>
  - 指定初始大小
  - 指定范围为1M ~ 30G

<size clause>

指定共享内存空间的byte大小（省略单位时默认为bytes）

临时内存数据不以文件形式管理对应image

- K : kilobytes
- M : megabytes
- G : gigabytes
- T : terabytes

<domain_name>

执行语句的成员或组的名称

未指定时在所有组执行

EXTSIZE <size clause>

指定表空间的extent大小

- extent大小以byte为单位可选择六种(64K, 128K, 256K, 512K, 1M, 2M)中的一个
- 如果extent大小指定为64K ~ 128K之间的值则被设置为128K指定为2M以上则被设置为2M

说明

临时表空间是提供存储index(NOLOGGING)等SQL schema对象与处理语句时用于sortinghashing的中间结果的物理空间的对象

使用示例

以下为创建临时表空间的示例

```
gSQL> CREATE TEMPORARY TABLESPACE temp_space1 MEMORY 'test_memory_1' SIZE
10M;


Tablespace created.
```

以下为创建有多个内存空间的临时表空间的示例

```
gSQL> CREATE TEMPORARY TABLESPACE temp_space1
             MEMORY 'test_memory_3_1' SIZE 10M,
```

```
                    'test_memory_3_2' SIZE 10M;


Tablespace created.
```

兼容性

标准SQL未定义表空间的概念

参考

相关内容参考下文

- **DROP TABLESPACE**
- **ALTER TABLESPACE**

9.13 CREATE PROFILE

功能

创建profile的语句可设置密码管理方法

向用户分配profile根据profile定义的方法管理用户密码

语句

```
<profile definition> ::=


    CREATE PROFILE profile_name LIMIT
    { <password_parameters>, ...}
    ;


<password parameters> ::=
      FAILED_LOGIN_ATTEMPTS { integer | UNLIMITED | DEFAULT }
    | PASSWORD_LOCK_TIME  { password_parameter_number_interval | UNLIMITED
| DEFAULT }
    | PASSWORD_LIFE_TIME  { password_parameter_number_interval | UNLIMITED
| DEFAULT }
    | PASSWORD_GRACE_TIME { password_parameter_number_interval | UNLIMITED
| DEFAULT }
```

```
    | PASSWORD_REUSE_MAX  { integer | UNLIMITED | DEFAULT }
    | PASSWORD_REUSE_TIME { password_parameter_number_interval | UNLIMITED
| DEFAULT }
    | PASSWORD_VERIFY_FUNCTION { <verify_policy> | NULL | DEFAULT }


<verify_policy> ::=
      KISA_VERIFY_FUNCTION
    | ORA12C_VERIFY_FUNCTION
    | ORA12C_STRONG_VERIFY_FUNCTION
    | VERIFY_FUNCTION_11G
    | VERIFY_FUNCTION


<password_parameter_number_interval> ::=
   integer
 | integer / integer
```

使用范围及访问权限

用户需要有CREATE PROFILE ON DATABASE权限才能执行<profile definition>语句

语句规则及参数

profile_name

指定要创建的profile名称

password_parameters

设置密码管理相关的参数

- 可以设置为以下参数
  - FAILED_LOGIN_ATTEMPTS
  - PASSWORD_LOCK_TIME
  - PASSWORD_LIFE_TIME
  - PASSWORD_GRACE_TIME
  - PASSWORD_REUSE_MAX
  - PASSWORD_REUSE_TIME
  - PASSWORD_VERIFY_FUNCTION

省略的参数遵循"DEFAULT" profile的策略

FAILED_LOGIN_ATTEMPTS

设置可连续登录失败的次数

超出指定次数将锁定账号

- FAILED_LOGIN_ATTEMPTS integer
  - 应为大于0的正整数值
- FAILED_LOGIN_ATTEMPTS UNLIMITED
  - 不会因登录失败而锁定用户
- FAILED_LOGIN_ATTEMPTS DEFAULT
  - 遵循"DEFAULT" profile策略

PASSWORD_LOCK_TIME

设置连续登录失败后锁定账号的时间（day）

- PASSWORD_LOCK_TIME constant_expression
  - 锁定持续的期限(day)
  - 默认单位为天(day)
  - 为了测试可以指定时(n/24)分(n/1440)秒(n/86400)
  - 值的范围为1秒(1/86400) ~ 100000天
- PASSWORD_LOCK_TIME UNLIMITED
  - 账号被锁定后执行ALTER USER user_name ACCOUNT UNLOCK语句之前不会解锁
- PASSWORD_LOCK_TIME DEFAULT
  - 遵循"DEFAULT" profile策略

PASSWORD_LIFE_TIME

设置密码的有效期限（day）

- PASSWORD_LIFE_TIME constant_expression

  - 密码的有效期限（day）
  - 默认单位为天（day）
  - 为了测试可以指定时（n/24）分（n/1440）秒（n/86400）
  - 值的范围为1秒（1/86400） ~ 100000天
- PASSWORD_LIFE_TIME UNLIMITED
  - 密码永不过期
- PASSWORD_LIFE_TIME DEFAULT
  - 遵循"DEFAULT" profile策略

PASSWORD_GRACE_TIME

设置PASSWORD_LIFE_TIME后登录时可继续使用当前密码的宽限期

- PASSWORD_GRACE_TIME constant_expression
  - 密码的宽限期（day）
  - 默认单位为天（day）
  - 为了测试可以指定时（n/24）分（n/1440）秒（n/86400）
  - 值的范围为1秒（1/86400） ~ 100000天
- PASSWORD_GRACE_TIME UNLIMITED
  - 持续宽限密码有效期
- PASSWORD_GRACE_TIME DEFAULT
  - 遵循"DEFAULT" profile策略

密码到期后第一次登录时开始计算PASSWORD_GRACE_TIME如在此期间内不变更密码则密码到期

PASSWORD_REUSE_MAX

设置要重复使用之前的密码时无法重复使用的最近密码的数量

PASSWORD_REUSE_MAX应与PASSWORD_REUSE_TIME同时使用

- PASSWORD_REUSE_MAX integer
  - 值的范围应为大于0的正整数
- PASSWORD_REUSE_MAX UNLIMITED
  - PASSWORD_REUSE_TIME为UNLIMITED时可重复使用之前的所有密码
  - PASSWORD_REUSE_TIME不是UNLIMITED时不能重复使用之前的任何密码
- PASSWORD_REUSE_MAX DEFAULT
  - 遵循"DEFAULT" profile策略

PASSWORD_REUSE_TIME

设置要重复使用之前的密码时无法重复使用该密码的时间

PASSWORD_REUSE_TIME应与PASSWORD_REUSE_MAX同时使用

- PASSWORD_REUSE_TIME constant_expression
  - 无法重复使用此密码的期限（day）
  - 默认单位为天（day）
  - 为了测试可以指定时（n/24）分（n/1440）秒（n/86400）
  - 值的范围为1秒（1/86400） ~ 100000天
- PASSWORD_REUSE_TIME UNLIMITED
  - PASSWORD_REUSE_MAX为UNLIMITED时可重复使用之前的所有密码

    - PASSWORD_REUSE_MAX不是UNLIMITED时不能重复使用之前的任何密码
- PASSWORD_REUSE_TIME DEFAULT
    - 遵循"DEFAULT" profile策略

PASSWORD_VERIFY_FUNCTION

设置密码复杂度验证方法

- PASSWORD_VERIFY_FUNCTION null
    - 不执行密码复杂度验证
- PASSWORD_VERIFY_FUNCTION DEFAULT
    - 遵循"DEFAULT" profile策略
- PASSWORD_VERIFY_FUNCTION <verify policy>
    - 可如下指定复杂度验证方法
        - KISA_VERIFY_FUNCTION
        - ORA12C_VERIFY_FUNCTION
        - ORA12C_STRONG_VERIFY_FUNCTION
        - VERIFY_FUNCTION_11G
        - VERIFY_FUNCTION

KISA_VERIFY_FUNCTION

Korea Internet & Security Agency (KISA)的密码验证方法

- 8个字符以上
- 1个以上的字母

- 1个以上的数字
- 1个以上的特殊字符

ORA12C_VERIFY_FUNCTION

Oracle的ORA12C_VERIFY_FUNCTION密码验证方法

- 8个字符以上
- 1个以上字母
- 1个以上数字
- 不能包含database name
- 不能包含用户名或倒序的用户名
- 不能包含'sundb'
- 不能包含'oracle'
- 不能使用如下简单的密码
  - welcome1, database1, account1, user1234, password1, oracle123, computer1, abcdefg1, change_on_intall
- 与之前的密码至少有3个不同的字符

ORA12C_STRONG_VERIFY_FUNCTION

Oracle的ORA12C_STRONG_VERIFY_FUNCTION密码验证方法

- 9个字符以上
- 2个以上大写字母
- 2个以上小写字母
- 2个以上数字

- 2个以上的特殊字符
- 与之前的密码至少有4个不同的字符

VERIFY_FUNCTION_11G

Oracle的VERIFY_FUNCTION_11G密码验证方法

- 8个以上字符
- 1个以上的字母
- 1个以上的数字
- 不能包含用户名
- 与之前的密码至少有3个不同的字符

VERIFY_FUNCTION

Oracle的VERIFY_FUNCTION密码验证方法

- 不能与用户名相同
- 4个以上字符
- 1个以上的字母
- 1个以上的数字
- 1个以上的特殊字符
- 不能使用如下简单的密码
    - welcome, database, account, user, password, oracle, computer, abcd
- 与之前的密码至少有3个不同的字符

说明

锁定账号

以下为影响账号锁定的参数

- FAILED_LOGIN_ATTEMPTS
- PASSWORD_LOCK_TIME

例如如下创建profile与user时

```
CREATE PROFILE prof LIMIT
    FAILED_LOGIN_ATTEMPTS 4
    PASSWORD_LOCK_TIME 30;

ALTER USER u1 PROFILE prof;
```

u1用户登录失败超过4次时被锁定30天

并在30天后解锁账号锁定

PASSWORD_LOCK_TIME为UNLIMITED时需要通过ALTER USER语句明确解锁用户锁定

```
ALTER USER user1 ACCOUNT UNLOCK;
```

密码过期

以下为会影响密码的有效期的参数

- PASSWORD_LIFE_TIME
- PASSWORD_GRACE_TIME

密码过期按照如下顺序产生

1. 设置密码
    - 以变更密码的时间点为基准将经过PASSWORD_LIFE_TIME的时间设置为密码过期时间点
    - 密码的过期状态为OPEN可以正常登录
2. 密码过期后的登录
    - 登录成功但密码的过期状态为EXPIRED(GRACE)并显示以下警告
        - ERR-28000(16310): The password will expire in n days
        - ERR-28000(16311): The password will expire soon
        - SQL标准未定义password expire概念
        - 28000为authentication warning或error的SQL标准状态编号(16310, 16311)为SUNDB的error code
    - 以登陆时间为准将经过PASSWORD_GRACE_TIME的时间重新设置为密码的过期时间点
3. 宽限期后的登录
    - 密码过期状态为EXPIRED而无法登录并显示以下警告
        - ERR-28000(16312) : The password has expired
        - SQL标准未定义password expire概念
        - 28000为authentication warning或error的SQL标准状态编号(16312)为SUNDB的error code
        - 为了使用程序控制密码的重新输入需要使用16312值的SUNDB internal error code

阶段	时间点	登录成功与否	用户的状态
1	变更密码	Success	OPEN
2	经过PASSWORD_LIFE_TIME后	Success with warning	EXPIRED(GRACE)
3	经过PASSWORD_GRACE_TIME后	Error	EXPIRED

Table 9-5 分阶段密码过期状态

如下所示

```
CREATE PROFILE prof LIMIT
   PASSWORD_LIFE_TIME 90
   PASSWORD_GRACE_TIME 3;


ALTER USER u1 PROFILE prof;
```

如上所示u1用户在90天后依然可以登录但将收到3天内密码将过期的警告信息

如果3天内不变更密码此密码将过期

密码过期后登录时提醒输入新的密码并拒绝账号访问

是否可重新使用密码

以下为影响密码的重复使用与否的参数

- PASSWORD_REUSE_MAX
- PASSWORD_REUSE_TIME

下表为两个参数的密码的重复使用与否

PASSWORD_REUSE_MAX	PASSWORD_REUSE_TIME	可重复使用的条件
value	value	需满足PASSWORD_REUSE_TIME与PASSWORD_REUSE_MAX条件
value	UNLIMITED	始终不可以
UNLIMITED	value	始终不可以
UNLIMITED	UNLIMITED	始终可以

Table 9-6 可重复使用密码的条件

创建如下profile时

```
CREATE PROFILE prof LIMIT
   PASSWORD_REUSE_MAX 5
   PASSWORD_REUSE_TIME 3;
```

无法重新使用最近的5个密码与近3天变更的的密码

以下为u1用户的密码变更记录假设当前密码为P#_000007当前日期为2015-08-08时重复使用之前密码的可行与否如下

password	password_date	是否可重复使用
P#_000001	2015-08-01	可以
P#_000002	2015-08-02	可以

password	password_date	是否可重复使用
P#_000003	2015-08-03	违反REUSE_MAX
P#_000004	2015-08-04	违反REUSE_MAX
P#_000005	2015-08-05	违反REUSE_MAX, REUSE_TIME
P#_000006	2015-08-06	违反REUSE_MAX, REUSE_TIME
P#_000007	2015-08-07	违反REUSE_MAX, REUSE_TIME

Table 9-7 可否重复使用的示例

为了检测是否可重新使用密码可以通过以下语句清除累积的密码变更历史记录

```
ALTER DATABASE CLEAR PASSWORD HISTORY;
```

DEFAULT profile

创建数据库时将自动创建如下"DEFAULT" profile创建的password parameter信息如下

parameter	value
FAILED_LOGIN_ATTEMPTS	10
PASSWORD_LOCK_TIME	1
PASSWORD_LIFE_TIME	180
PASSWORD_GRACE_TIME	7

parameter	value
PASSWORD_REUSE_MAX	UNLIMITED
PASSWORD_REUSE_TIME	UNLIMITED
PASSWORD_VERIFY_FUNCTION	NULL

Table 9-8 DEFAULT profile的构成

"DEFAULT" profile的默认值有以下特点

- 锁定账号
  - 连续10(FAILED_LOGIN_ATTEMPTS)次登录失败时账号将被锁定1天(PASSWORD_LOCK_TIME)
- 密码过期
  - 180天(PASSWORD_LIFE_TIME)后将有7天(PASSWORD_GRACE_TIME)的宽限期之后密码过期
- 密码重复使用与否
  - 可以重复使用之前的密码
- 检查密码复杂度
  - 不检查

无法清除DEFAULT profile并且可通过以下语句更改

```
ALTER PROFILE DEFAULT LIMIT ...
```

使用示例

以下为创建控制用户锁定的profile的示例连续3次登录失败时此账号将被锁定3天

```
gSQL> CREATE PROFILE prof1 LIMIT
        FAILED_LOGIN_ATTEMPTS 3
        PASSWORD_LOCK_TIME 3;

Profile created.

gSQL> COMMIT;

Commit complete.
```

以下为创建控制密码过期的profile的示例密码有效期为90天有7天的宽限期

```
gSQL> CREATE PROFILE prof1 LIMIT
        PASSWORD_LIFE_TIME 90
        PASSWORD_GRACE_TIME 7;

Profile created.

gSQL> COMMIT;

Commit complete.
```

以下为创建控制密码是否可重复使用的profile的示例如下示例中在变更密码时不检查之前的密码

```
gSQL> CREATE PROFILE prof1 LIMIT
        PASSWORD_REUSE_MAX  DEFAULT
        PASSWORD_REUSE_TIME DEFAULT;

Profile created.

gSQL> COMMIT;

Commit complete.
```

以下为创建控制密码复杂度检查的profile的示例

```
gSQL> CREATE PROFILE prof1 LIMIT
        PASSWORD_VERIFY_FUNCTION KISA_VERIFY_FUNCTION;

Profile created.

gSQL> COMMIT;

Commit complete.
```

以下为设置所有参数后创建profile的示例

```
gSQL> CREATE PROFILE prof1 LIMIT
        FAILED_LOGIN_ATTEMPTS 3
        PASSWORD_LOCK_TIME 3
        PASSWORD_LIFE_TIME 90
        PASSWORD_GRACE_TIME 7
        PASSWORD_REUSE_MAX  DEFAULT
        PASSWORD_REUSE_TIME DEFAULT
        PASSWORD_VERIFY_FUNCTION KISA_VERIFY_FUNCTION;

Profile created.

gSQL> COMMIT;

Commit complete.
```

兼容性

标准SQL未定义profile概念

参考

相关内容参考下文

- **DROP PROFILE**

- ALTER PROFILE
- CREATE USER
- ALTER USER
- ALTER DATABASE CLEAR PASSWORD HISTORY

9.14 CREATE SCHEMA

功能

定义SCHEMA

语句

```
<schema definition> ::=
    CREATE SCHEMA <schema name clause>
        [ <schema element> [...] ]
    ;

<schema name clause> ::=
      schema_name
    | AUTHORIZATION user_identifier
    | schema_name AUTHORIZATION user_identifier

<schema element> ::=
      <table definition>
    | <view definition>
    | <index definition>
    | <sequence generator definition>
```

```
    | <grant privilege statement>
    | <comment statement>
```

使用范围及访问权限

用户应满足以下条件才能执行<schema definition>语句

- 需要有CREATE SCHEMA ON DATABASE权限才能创建SCHEMA
- 如果有<schema element>时为了执行<schema element>语句需要有相应的权限详细内容参考以下各语句的访问权限
  - **CREATE TABLE**
  - **CREATE VIEW**
  - **CREATE INDEX**
  - **CREATE SEQUENCE**
  - **GRANT privileges TO**
  - **COMMENT ON name IS**
- user_identifier用户对创建的SCHEMA有以下权限
  - 创建的schema_name SCHEMA所有者
  - 用<schema element>创建的对象的所有者
- 生成的SCHEMA不会有额外的权限因此为了创建对象需另赋合适的Schema权限

  SCHEMA权限类型参考GRANT privileges TO语句的**<schema privilege>**

  相关使用示例参考CREATE USER的使用示例

语句规则及参数

schema_name

要创建的SCHEMA名称

应为数据库内唯一的SCHEMA名称

SCHEMA名称的长度应小于128 byte

AUTHORIZATION user_identifier

省略Schema名称时创建与user_identifier相同名称的SCHEMA

不指定AUTHORIZATION时使用执行语句的用户的user_identifier

schema_name AUTHORIZATION user_identifier

指定创建的SCHEMA名称与SCHEMA所有者

所有者不能为role或PUBLIC

<schema element>

创建schema的同时定义schema内创建的对象

schema_element以列出的顺序执行没有逗号(,)仅用空白区分

不能在与创建的schema不同名称的schema中定义对象

- 使用<grant privilege statement>语句时只能定义以下<privilege>

    - <schema priviliege>
    - <table privilege>
    - <sequence privilege>
- <comment statement>语句只能定义以下对象
    - SCHEMA schema_name
    - TABLE [schema_name].table_name
    - COLUMN [schema_name].table_name.column_name
    - INDEX [schema_name].index_name
    - SEQUENCE [schema_name].sequence_name
    - CONSTRAINT [schema_name].constraint_name

说明

SCHEMA是逻辑分类tableviewindexsequenceconstraint等SQL schema的对象

SUNDB的user与schema是1:N的关系即用户没有自己的SCHEMA或可以有多个SCHEMA

标准SQL未明确定义userschemadatabase等non-schema对象之间的关系不同的数据库对non-schema对象之间的关系有不同的定义

> Note:
>
> DBMS中user与schema的关系
>
> * Oracle
>
> ** User : schema = 1 : 1

* DB2

** User : schema = 1 : N

* Postgres

** User : schema = 1 : N

* MySQL

** Database : schema = 1 : 1

** User为database (schema)的下层对象

使用示例

以下为创建schema的示例

```
gSQL> CREATE SCHEMA s1;

Schema created.
```

以下为创建schema并指定其所有者的示例

```
gSQL> CREATE SCHEMA s1 AUTHORIZATION test;

Schema created.
```

以下为创建schema的同时创建其对象的示例

```
gSQL> CREATE SCHEMA s1
            CREATE TABLE t1 ( id INTEGER, name VARCHAR(128) )
            CREATE INDEX idx_t1_id ON t1 ( id )
            COMMENT ON TABLE t1 IS 'comment on s1.t1'
;

Schema created.
```

兼容性

Feature ID	说明	是否支持
S071	SQL paths in function and type name resolution	X
F461	Named character sets	X
F171	Multiple schemas per user	O
T332	Extended roles	X

Table 9-9 标准SQL兼容性

参考

相关内容参考下文

- **DROP SCHEMA**
- **CREATE USER**
- **CREATE TABLE**
- **CREATE VIEW**
- **CREATE INDEX**
- **CREATE SEQUENCE**
- **GRANT privileges TO**
- **COMMENT ON name IS**

9.15 CREATE SEQUENCE

功能

创建序列（sequence）

语句

```
<sequence generator definition> ::=
    CREATE SEQUENCE [schema_name.] sequence_name
        [ <sequence generator option> [, ...] ]
    ;

<sequence generator option> ::=
      <sequence generator start with option>
    | <basic sequence generator option>

<sequence generator start with option> ::=
    START WITH integer

<basic sequence generator option> ::=
      <sequence generator increment by option>
    | <sequence generator maxvalue option>
```

```
    | <sequence generator minvalue option>
    | <sequence generator cycle option>
    | <sequence generator cache option>

<sequence generator increment by option> ::=
    INCREMENT BY integer

<sequence generator maxvalue option> ::=
      MAXVALUE integer
    | (NO MAXVALUE | NOMAXVALUE)

<sequence generator minvalue option> ::=
      MINVALUE integer
    | (NO MINVALUE | NOMINVALUE)

<sequence generator cycle option> ::=
      CYCLE
    | (NO CYCLE | NOCYCLE)

<sequence generator cache option> ::=
      CACHE integer
    | (NO CACHE | NOCACHE)
```

使用范围及访问权限

用户需要有以下权限中的一个才能执行<sequence generator definition>语句

- 对序列所属的SCHEMA有(CREATE SEQUENCE或CONTROL SCHEMA) ON SCHEMA
- CREATE ANY SEQUENCE ON DATABASE

如下确定序列的所有者

- 序列所属的schema的所有者
- 序列所属的schema为PUBLIC时执行语句的用户

序列所有者有USAGE ON SEQUENCE WITH GRANT OPTION权限

需要有以下权限中的一个才能使用创建的序列

- 对序列有USAGE ON SEQUENCE
- 对序列所在的SCHEMA有 (USAGE SEQUENCE或CONTROL SCHEMA) ON SCHEMA
- USAGE ANY SEQUENCE ON DATABASE

语句规则及参数

sequence_name

要创建的序列名应为SCHEMA内唯一的名称

与schema_name.sequence_name相同可定义序列所在的SCHEMA省略schema_name时使用执行

语句的用户的默认SCHEMA名

序列名的长度应小于128 byte

<sequence generator option>

不使用<sequence generator option>时以下两条语句意义相同

- CREATE SEQUENCE test_seq;
- CREATE SEQUENCE test_seq START WITH 1 INCREMENT BY 1 NO MINVALUE NO MAXVALUE NO CYCLE CACHE 20;

<sequence generator start with option>

定义首次创建的序列号

根据升序或降序有以下特点

- 升序序列时（INCREMENT BY 正数）
  - 用于以大于最小值的序列值开始的情况
  - 省略START WITH时默认为最小值（MINVALUE value）
- 降序序列时（INCREMENT BY 负数）
  - 用于以小于最大值的序列值开始的情况
  - 省略START WITH时默认为最大值（MAXVALUE value）

<sequence generator increment by option>

定义序列号的间隔

有以下约束及特点

- 可以使用正数或负数但不能使用0

- 间隔的绝对值应小于MINVALUE与MAXVALUE之差
- 为正数时创建升序序列为负数时创建降序系列
- 省略INCREMENT BY时默认值为正数1

<sequence generator maxvalue option>

定义序列可创建的最大值

- MAXVALUE integer
  - 最大值的范围为64bit整数的最小值(−9,223,372,036,854,775,808)到64bit整数的最大值(+9,223,372,036,854,775,807)之间
  - 应大于或等于START WITH的值大于MINVALUE值
- NO MAXVALUE | NOMAXVALUE
  - 如下定义最大值
    - 升序序列时为64bit整数的最大值(+9,223,372,036,854,775,807)
    - 降序序列时为-1
  - NO MAXVALUE (标准SQL)与NOMAXVALUE为相同意义的保留字因此可任意使用
- 未指定MAXVALUE (标准SQL)与NO MAXVALUE时默认为NO MAXVALUE

<sequence generator minvalue option>

定义序列可创建的最小值

- MINVALUE integer
  - 最小值的范围为64bit整数的最小值(−9,223,372,036,854,775,808)到64bit整数的最大值(+9,223,372,036,854,775,807)之间

    - 应小于或等于START WITH的值小于MAXVALUE值
- NO MINVALUE | NOMINVALUE
    - 如下定义最小值
        - 升序序列时为1
        - 降序序列时为64bit整数的最小值(-9,223,372,036,854,775,808)
    - NO MINVALUE(标准SQL)与NOMINVALUE为相同意义的保留字因此可任意使用
- 未指定MINVALUE (标准SQL)与NO MINVALUE时默认为NO MINVALUE

<sequence generator cycle option>

指定序列值达到最大值或最小值时是否继续创建值

- CYCLE
    - 升序序列已达到最大值时从最小值开始重新创建
    - 降序序列已达到最小值时从最大值开始重新创建
- NO CYCLE | NOCYCLE
    - 已达到最大值或最小值时不能再创建序列值
    - NO CYCLE (SQL 亚圣)与NOCYCLE为相同意义的保留字因此可任意使用
- 未指定CYCLE或NO CYCLE时默认为NO CYCLE

<sequence generator cache option>

定义为了快速访问序列而在内存上预加载的序列值的数量

重启数据库时丢失之前在内存上加载的序列值序列值从加载后的值开始

- CACHE integer

    - CACHE值应大于或等于2
    - 有CYCLE时CACHE值不能大于CYCLE长度
        - CYCLE长度：CEIL(MAXVALUE - MINVALUE) / ABS(INCREMENT)
- NO CACHE | NOCACHE
    - 不在内存上预加载序列值
- 未指定CACHE或NO CACHE时默认为CACHE 20

说明

可通过**NEXTVAL**函数与**CURRVAL** 函数使用创建的序列对象的序列值

序列值没有事务属性因此使用序列函数的SQL语句发生错误或执行回滚时序列值也会维持最新值

CURRVAL函数返回会话中最近调用的NEXTVAL值
使用此特性可在其他SQL语句继续使用通过一次NEXTVAL获取的序列值但会话中未先调用NEXTVAL时使用CURRVAL会报错

使用示例

如下未定义序列选项的seq1对象为与seq2对象相同意义的升序序列

```
gSQL> CREATE SEQUENCE seq1;


Sequence created.
```

```
gSQL> CREATE SEQUENCE seq2 START WITH 1 INCREMENT BY 1 NO MINVALUE NO
MAXVALUE NO CYCLE CACHE 20;

Sequence created.
```

以下为创建奇数序列值的序列

```
gSQL> CREATE SEQUENCE seq1 START WITH 1 INCREMENT BY 2;

Sequence created.
```

以下为从0开始到1000反复创建偶数序列值的示例

```
gSQL> CREATE SEQUENCE seq1 START WITH 0 MINVALUE 0 MAXVALUE 1000 INCREMENT
BY 2 CYCLE;

Sequence created.
```

以下为从-1开始创建降序序列的示例

```
gSQL> CREATE SEQUENCE seq1 INCREMENT BY -1;

Sequence created.
```

兼容性

标准SQL未定义<sequence generator cache option>语句

Feature ID	说明	是否支持
T176	Sequence generator support	O

Table 9-10 标准SQL兼容性

参考

相关内容参考下文

- **DROP SEQUENCE**
- **ALTER SEQUENCE**
- **NEXTVAL**
- **CURRVAL**

9.16 CREATE SYNONYM

功能

创建同义词同义词是表视图序列其他同义词的别名可用于以下语句

- DML : SELECT, INSERT, UPDATE, DELETE, LOCK TABLECALL
- DDL : GRANT, REVOKE, COMMENT

语句

```
<table definition> ::=
    CREATE [OR REPLACE] [PUBLIC] SYNONYM [schema_name.]synonym_name
    FOR [schema_name.]object_name
    ;
```

使用范围及访问权限

用户应满足以下条件才能执行<synonym definition>语句

- 指定PUBLIC为了创建public synonym需要有CREATE PUBLIC SYNONYM ON DATABASE权限
- Public synonym的所有者为PUBLIC创建的用户没有任何权限
- 需要有以下权限中的一个才能创建private synonym

  - 对SCHEMA有(CREATE SYNONYM或CONTROL SCHEMA) ON SCHEMA
  - CREATE ANY SYNONYM ON DATABASE
- 如下决定private synonym的所有者
  - Private synonym所属的schema的所有者
  - Private synonym所属的schema为PUBLIC时执行语句的用户
- 即使创建同义词如果没有对应对象的权限则无法执行使用此同义词的语句
- 如果给同义词赋予权限则也会给同义词所指的对象赋予权限因此赋予权限时需注意

语句规则及参数

[OR REPLACE]

如果有同义词则代替原有的同义词

[PUBLIC]

为了创建public synonym而指定

省略时创建private synonym

synonym_name

要创建的同义词名称并且应为在SCHEMA内唯一的名称

与schema_name.synonym_name相同可定义同义词所属的SCHEMA省略schema_name时使用执行语句的用户的默认SCHEMA名

同义词名称的长度应小于128 byte

Public synonym为non-schema对象因此指定'PUBLIC'创建Public Synonym时无法指定SCHEMA名

object_name

与schema_name.object_name相同可定义对象所在的SCHEMA省略schema_name时使用执行语句的用户的默认SCHEMA名

可指定object_name的对象类型如下

- Table
- View
- Sequence
- 其他synonym

对应对象的存在与否cycle check检查权限等均在执行使用同义词的语句时执行

说明

同义词是表视图序列其他同义词的别名

使用同义词时即使变更底层的对象也不需要修改应用程序仅需要重新定义该同义词因此很方便另外同义词隐藏对象的真实名称与schema可改善数据库的安全性而且可缩短数据库的名称提高可用性

由于同义词只是一个别名即使创建同义词也无法使用同义词直接访问对应对象需要对此对象有相应的权限才能访问

执行使用同义词的语句时按以下顺序访问对象

1. 查找对应名称的表
2. 没有表时查找对应名称的Private Synonym
3. 没有private synonym时查找对应名称的public synonym

```
gSQL> CREATE PUBLIC SYNONYM syn1 FOR u1.t1;


Synonym created.


gSQL> CREATE PUBLIC SYNONYM syn2 FOR syn1;


Synonym created.


gSQL> SELECT * FROM syn2;
```

以上SELECT语句的访问对象顺序如下

1. 检索syn2表但该表不存在
2. 检索syn2 private synonym但该synonym不存在
3. 检索syn2 public synonym找到该synonym
    A. 检索syn1表但该表不存在
    B. 检索syn1 private synonym但该synonym不存在
    C. 检索syn1 public synonym找到该synonym
        i. 通过检索找到u1.t1表

使用示例

以下为创建private synonym的示例

```
gSQL> CREATE SYNONYM MyEmp FOR branch.Employee;

Synonym created.

gSQL> SELECT * FROM MyEmp;
```

以下为创建public synonym的示例

```
gSQL> CREATE PUBLIC SYNONYM MainEmp FOR main.Employee;

Synonym created.

gSQL> SELECT * FROM MainEmp;
```

兼容性

标准SQL未定义CREATE SYNONYM语句

参考

相关内容参考**DROP SYNONYM**

9.17 CREATE TABLE

功能

定义表

语句

```
<table definition> ::=
    CREATE TABLE table_name
        ( <table element> [, ...] )
        [ <table sharding strategy> ]
        [ <table attribute clause> [...] ]
        [ TABLESPACE tablespace_name ]
        [ <table global secondary index clause> ]
    ;

<table element> ::=
      <column definition>
    | <table constraint definition>

<column definition> ::=
    column_name <data type>
```

```
        [ <default clause> | <identity column specification> ]
        [ <column constraint definition> ]


<data type> ::=
      <character string type>
    | <binary string type>
    | <numeric type>
    | <boolean type>
    | <datetime type>
    | <interval type>


<character string type> ::=
      CHARACTER [ ( integer [ <character length units> ] ) ]
    | CHAR [ ( integer [ <character length units> ] ) ]
    | CHARACTER VARYING ( integer [ <character length units> ] )
    | CHAR VARYING ( integer [ <character length units> ] )
    | VARCHAR ( integer [ <character length units> ] )
    | CHARACTER LONG VARYING
    | LONG VARCHAR


<character length units> ::=
      CHARACTERS
    | CHAR
    | OCTETS
    | BYTE
```

```
<binary string type> ::=
      BINARY [ ( length ) ]
    | BINARY VARYING ( length )
    | VARBINARY ( length )
    | LONG BINARY VARYING
    | LONG VARBINARY

<numeric type> ::=
      <exact numeric type>
    | <approximate numeric type>
    | <native numeric type>

<exact numeric type> ::=
      NUMERIC [ ( precision [, scale ] ) ]
    | SMALLINT
    | INTEGER
    | INT
    | BIGINT

<approximate numeric type> ::=
    | FLOAT [ ( precision ) ]
    | REAL
    | DOUBLE PRECISION
```

```
<native numeric type> ::=

      NATIVE_SMALLINT

    | NATIVE_INTEGER

    | NATIVE_BIGINT

    | NATIVE_REAL

    | NATIVE_DOUBLE


<boolean type> ::=

    BOOLEAN


<datetime type> ::=

      DATE

    | TIME [ ( time_precision ) ] [ WITH TIME ZONE | WITHOUT TIME ZONE ]

    | TIMESTAMP [ ( timestamp_precision ) ] [ WITH TIME ZONE | WITHOUT
TIME ZONE ]


<interval type> ::=

    INTERVAL <interval qualifier>


<interval qualifier> ::=

      <non-second primary datetime field>
[ ( interval_leading_field_precision ) ]

          TO { <non-second primary datetime field> | SECOND
[ ( interval_fractional_seconds_precision ) ] }

    | <non-second primary datetime field>
```

```
[ ( interval_leading_field_precision ) ]
    | SECOND [ ( interval_leading_field_precision [,
interval_fractional_seconds_precision ] ) ]

<non-second primary datetime field> ::=
      YEAR
    | MONTH
    | DAY
    | HOUR
    | MINUTE

<default clause> ::=
    DEFAULT <default option>

<default option> ::=
      constant
    | NULL
    | expression

<identity column specification> ::=
    GENERATED { ALWAYS | BY DEFAULT } AS IDENTITY
    [ ( <common sequence generator option> [, ...] ) ]

<common sequence generator option> ::=
      START WITH integer_constant
```

```
    | <basic sequence generator option>


<basic sequence generator option> ::=

      INCREMENT BY integer_constant

    | { MAXVALUE integer_constant | NO MAXVALUE }

    | { MINVALUE integer_constant | NO MINVALUE }

    | { CYCLE | NO CYCLE }

    | { CACHE integer_constant | NO CACHE }


<column constraint definition> ::=

    [ CONSTRAINT constraint_name ] <column constraint> [ <constraint
characteristics> ]


<column constraint> ::=

      NOT NULL

    | { UNIQUE | PRIMARY KEY } [ <index name clause> [ <index
attributes> ] [ TABLESPACE index_tablespace_name ] ]


<index name clause> ::=

    INDEX index_name


<index attributes> ::=

      <index physical attribute clause>

    | STORAGE ( <segment attr clause> [...] )
```

```
<table constraint definition> ::=
    [ CONSTRAINT constraint_name ] <table constraint> [ <constraint
characteristics> ]

<table constraint> ::=
      <unique constraint definition> [ <index name clause> [ <index
attributes> ] [ TABLESPACE index_tablespace_name ] ]

<unique constraint definition> ::=
    { UNIQUE | PRIMARY KEY } ( <key column element> [, ...] )

<key column element> ::=
    column_name [ ASC | DESC ] [ NULLS FIRST | NULLS LAST ]


<table sharding strategy> ::=
      <cloned strategy>
    | <hash sharding strategy>
    | <range sharding strategy>
    | <list sharding strategy>

<cloned strategy> ::=
    CLONED [ <clone placement> ]
```

```
<clone placement> ::=
      AT CLUSTER WIDE
    | AT CLUSTER GROUP group_list

<hash sharding strategy> ::=
    SHARDING BY [HASH] ( column_list )
    [ <hash shard count> ]
    [ <hash shard placement> ]

<hash shard count> ::=
    SHARD COUNT integer

<hash shard placement> ::=
      AT CLUSTER WIDE
    | AT CLUSTER GROUP group_list

<range sharding strategy> ::=
    SHARDING BY RANGE ( column_list )
    { <cluster-wide range shard placement> | <group-specific range shard
placement> }

<cluster-wide range shard placement> ::=
    AT CLUSTER WIDE
    <range shard definition> [, ...]
```

```
<group-specific range shard placement> ::=

    <group-specific range shard definition> [, ...]


<group-specific range shard definition> ::=

    <range shard definition> AT CLUSTER GROUP group_name


<range shard definition> ::=

    SHARD range_name VALUES LESS THAN ( <range value clause> )


<range value clause> ::=

    <range value> [, ...]


<range value> ::=

      constant

    | MAXVALUE


<list sharding strategy> ::=

    SHARDING BY LIST ( column_name )

    { <cluster-wide list shard placement> | <group-specific list shard
placement> }


<cluster-wide list shard placement> ::=

    AT CLUSTER WIDE

    <list shard definition> [, ...]
```

```
<group-specific list shard placement> ::=

    <group-specific list shard definition> [, ...]


<group-specific list shard definition> ::=

    <list shard definition> AT CLUSTER GROUP group_name


<list shard definition> ::=

      SHARD shard_name VALUES IN ( <list value clause> )


<list value clause> ::=

    <list value> [, ...]


<list value> ::=

      constant

    | NULL

    | DEFAULT




<table attribute clause> ::=

      [ <table physical attribute clause> ]

    | [ STORAGE ( <segment attr clause> [...] ) ]


<table physical attribute clause> ::=

      PCTFREE integer

    | PCTUSED integer
```

```
    | INITRANS integer
    | MAXTRANS integer

<index physical attribute clause> ::=
      PCTFREE integer
    | INITRANS integer
    | MAXTRANS integer

<segment attr clause> ::=
      INITIAL <size_clause>
    | NEXT <size_clause>
    | MINSIZE <size_clause>
    | MAXSIZE <size_clause>

<size clause> ::=
      integer [ K | M | G | T ]


<constraint characteristics> ::=
      [ NOT ] DEFERRABLE [ <constraint check time> ]
    | <constraint check time> [ [ NOT ] DEFERRABLE ]

<constraint check time> ::=
      INITIALLY DEFERRED
    | INITIALLY IMMEDIATE
```

```
<table global secondary index clause> ::=

      WITH GLOBAL SECONDARY INDEX [ <index attributes> [...] ]

[ TABLESPACE tablespace_name ]

    |  WITHOUT GLOBAL SECONDARY INDEX
```

使用范围及访问权限

根据数据库为单机版还是集群版有如下区别

- 单机版
  - 无法定义<table sharding strategy>
  - 无法定义<table global secondary index clause>
- 集群版
  - 定义UNIQUEPRIMARY KEY约束条件时需包含所有sharding key
  - 无法定义可延时的约束条件

用户应满足以下条件才能执行<table definition>语句

- 对创建表的SCHEMA需要有以下权限中的一个
  - 对SCHEMA有(CREATE TABLE或CONTROL SCHEMA) ON SCHEMA
  - CREATE ANY TABLE ON DATABASE
- 对创建表的表空间需要有以下权限中的一个
  - 对表空间有CREATE OBJECT ON TABLESPACE
  - USAGE TABLESPACE ON DATABASE

- 有同时创建的约束条件时对创建约束条件的SCHEMA需要有以下权限中的一个
  - 对SCHEMA有(ADD CONSTRAINT或CONTROL SCHEMA) ON SCHEMA
  - ALTER ANY TABLE ON DATABASE
- 同时创建的约束条件为key约束条件时对创建索引的表空间需要有以下权限中的一个
  - 对表空间有CREATE OBJECT ON TABLESPACE
  - USAGE TABLESPACE ON DATABASE
- 如下确定表的所有者
  - 表所属的schema的所有者
  - 表所属的schema为PUBLIC时执行语句的用户
- 表所有者对创建的表拥有以下权限
  - 对表的权限
    - SELECT ON TABLE WITH GRANT OPTION
    - INSERT ON TABLE WITH GRANT OPTION
    - UPDATE ON TABLE WITH GRANT OPTION
    - DELETE ON TABLE WITH GRANT OPTION
    - TRIGGER ON TABLE WITH GRANT OPTION
    - REFERENCES ON TABLE WITH GRANT OPTION
    - LOCK ON TABLE WITH GRANT OPTION
    - INDEX ON TABLE WITH GRANT OPTION
    - ALTER ON TABLE WITH GRANT OPTION
  - 对表的所有Column的权限
    - SELECT(columns) ON TABLE WITH GRANT OPTION
    - INSERT(columns) ON TABLE WITH GRANT OPTION
    - UPDATE(columns) ON TABLE WITH GRANT OPTION
    - REFERENCES(columns) ON TABLE WITH GRANT OPTION

- 对同时创建的约束条件的权限
  - 约束条件的所有者
  - 与约束条件同时创建的索引的所有者

无法在集群系统使用<table sharding strategy>语句

语句规则及参数

table_name

要创建的表的名称应为SCHEMA内唯一的名称

与schema_name.table_name相同可定义表所在的SCHEMA省略schema_name时使用执行语句的用户的默认SCHEMA名

表名长度应小于128 byte

<column definition>

定义构成表的column

表应包含一个以上的column定义

可定义column的数据类型默认值自动创建值约束条件等

column_name

构成表的column名所有column名应为在表内唯一的名称

Column名长度应小于128 byte

<data type>

- <character string type>
- <binary string type>
- <numeric type>
- <exact numeric type>
- <approximate numeric type>
- <boolean type>
- <datetime type>
- <interval type>
- <interval qualifier>
- <non-second primary datetime field>

定义column的数据类型

定义拥有自动创建值(<identity column specification>)的Column时数据类型应为

SMALLINTINTEGERBIGINT中的一个

数据类型相关内容参考**Data Types**

<character length units>

指定character类型的一个文字的长度单位

- CHARACTERS/CHAR是将一个文字的最大byte指定为一个文字的长度因此汉字等multi-bytes文字的长度也处理为1
- OCTETS/BYTE将1bytes指定为一个文字的长度因此汉字等multi-bytes文字的长度处理为multi-bytes

- 省略时遵循数据库创建时使用的CHAR_LENGTH_UNITS属性值

标准SQL的默认值为CHARACTERS

> Note:
>
> 其他DBMS的char lenth unit默认值如下
>
> • Oracle, DB2: OCTETS
>
> • MS-SQL, MySQL, PostgreSQL: CHARACTERS

[<default clause> | <identity column specification>]

指定column的默认值

<default clause>与<identity column specification>不能一起使用

省略时默认值为NULL

<default clause>

DEFAULT子句定义INSERTUPDATE等语句中指定DEFAULT或省略对应column名时使用的默认值

- 使用DEFAULT时
  - 例: CREATE TABLE t1 (id INTEGER, name VARCHAR(32) DEFAULT 'anonymous');
  - 省略Column时
    - INSERT INTO t1(id) VALUES (1);
    - INSERT INTO t1(id) SELECT id FROM other_table;
  - 指定DEFAULT时

- INSERT INTO t1 DEFAULT VALUES;
- INSERT INTO t1 VALUES (2, DEFAULT);
- UPDATE t1 SET name = DEFAULT;

DEFAULT expression的数据类型应可以与Column的数据类型进行转换

类型之间不可转换或空间不足时如果在INSERTUPDATE语句中使用DEFAULT则如下报错

- CREATE TABLE t1 (user_name VARCHAR(1) DEFAULT CURRENT_USER);
- 例: error: INSERT INTO t1 VALUES (DEFAULT);
- 例: error: UPDATE t1 SET user_name = DEFAULT;

DEFAULT expression可以使用所有built-in函数但不能使用如下

- 逻辑运算符(AND, OR, NOT)比较运算符(=, >, ..)
- Stored Function
- Column名
- subquery expression

<identity column specification>

定义拥有自动创建值的Column

一张表只能有一个identity column

即使不指定NOT NULL约束条件identity column也会成为not nullable column

<identity column specification>与DEFAULT不能同时使用

<identity column specification>与DEFAULT相同定义在INSERTUPDATE语句中指定DEFAULT或

省略Column名时使用的默认值

如下定义创建方式

- GENERATED BY DEFAULT AS IDENTITY

  用户指定值时应用其值与DEFAULT相同需要使用默认值时创建自动值

  - CREATE TABLE t1 (id INTEGER GENERATED BY DEFAULT AS IDENTITY, name VARCHAR(32));
  - (O) INSERT INTO t1 VALUES (12345, 'SUNDB');
    - 输入用户指定的值（12345）
  - (O) INSERT INTO t1(name) VALUES ('SUNDB');
    - 在id column输入自动创建值
  - (O) INSERT INTO t1(id, name) SELECT other_id, other_name FROM other_table;
    - 输入用户指定的值
  - (O) INSERT INTO t1(name) SELECT other_name FROM other_table;
    - 在id column输入自动创建值
  - (O) UPDATE t1 SET id = 10000 WHERE id = 12345;
    - 输入用户指定的值
  - (O) UPDATE t1 SET id = DEFAULT WHERE id = 12345;
    - 在id column输入自动创建值
- GENERATED ALWAYS AS IDENTITY

  用户无法指定值与DEFAULT相同应可创建默认值

  - CREATE TABLE t1 (id INTEGER GENERATED ALWAYS AS IDENTITY, name VARCHAR(32));
  - (X) INSERT INTO t1 VALUES (12345, 'SUNDB');
    - 报错用户无法指定值

- (O) INSERT INTO t1(name) VALUES ('SUNDB');
  - 在id column输入自动创建值
- (X) INSERT INTO t1(id, name) SELECT other_id, other_name FROM other_table;
  - 报错用户无法指定值
- (O) INSERT INTO t1(name) SELECT other_name FROM other_table;
  - 在id column输入自动创建值
- (X) UPDATE t1 SET id = 10000 WHERE id = 12345;
  - 报错用户无法指定值
- (O) UPDATE t1 SET id = DEFAULT WHERE id = 12345;
  - 在id column输入自动创建值

关于identity column的创建选项<common sequence generator option>与<basic sequence generator option>的详细内容参考**CREATE SEQUENCE**

<column constraint definition>

对column定义如下约束条件

- NOT NULL约束条件
- UNIQUE约束条件
- PRIMARY KEY约束条件

constraint_name

约束条件名称可省略

省略constraint_name时如下自动命名约束条件

当自动生成的名称重复时需要指定constraint_name

- NOT NULL约束条件
  - "table_name" + "_" + "NOT_NULL" + "_" + "column_name"
- UNIQUE约束条件
  - "table_name" + "_" + "UNIQUE" + "_" + "column_name"
- PRIMARY KEY约束条件
  - "table_name" + "_" + "PRIMARY_KEY"

约束条件名的长度应小于128 byte

NOT NULL约束条件

Column不允许有空值

UNIQUE约束条件

Column不允许有相同值

但允许空值

PRIMARY KEY约束条件

Column不允许有空值或相同值

一个表只能定义一个PRIMARY KEY约束条件

<index name clause>

定义UNIQUE约束条件 PRIMARY KEY约束条件时创建的对应索引名

- INDEX index_name
  - 定义约束条件的索引名
  - 无法与SCHEMA名称一起使用在与约束条件相同的SCHEMA中生成

定义UNIQUE约束条件PRIMARY KEY约束条件时省略INDEX子句的情况下自动生成符合约束条件的索引

自动生成的索引名为"constraint_name" + "_INDEX"

- <index attributes>
  - 指定要创建的索引的物理属性
  - 详细内容参考**CREATE INDEX**
- TABLESPACE index_tablespace_name
  - 指定要创建索引的表空间
  - 详细内容参考**CREATE INDEX**

<table constraint definition>

- <unique constraint definition>
- 定义表时根据约束条件在语句中的位置可分为以下两种
  - 定义column时可通过column约束定义（<column constraint definition>）定义对一个column的约束条件
  - 相反表约束定义（<table constraint definition>）可与column定义分开另行指定可指定

一个以上column的约束条件

对表的约束定义与对column的约束定义在语句上有以下区别

- NOT NULL约束
  - 无法通过表约束定义指定
- 与column的约束定义不同需要指定Column
  - UNIQUE约束<unique constraint definition>
    - UNIQUE (column_name [, ...])
  - PRIMARY KEY约束<unique constraint definition>
    - PRIMARY KEY (column_name [, ...])

key column element

指定作为key对象的column

- Column name
  - 创建Key的column的名称
- ASC | DESC
  - ASC : 升序排列
  - DESC : 降序排列
  - 未指定时默认值为ASC
- NULLS FIRST | NULLS LAST
  - NULLS FIRST : 把NULL排在最前面
  - NULLS LAST : 把NULL排在最后面
  - 未指定时默认为NULLS LAST

<table sharding strategy>

定义表的sharding策略

可定义如下四种策略中的一个

- <cloned strategy>
- <hash sharding strategy>
- <range sharding strategy>
- <list sharding strategy>

省略时取决于**DEFAULT_SHARDING**参数值

- DEFAULT_SHARDING值为0时
  - <cloned strategy>
- DEFAULT_SHARDING值为1时
  - <hash sharding strategy>

<cloned strategy>

复制表的所有数据

<clone placement>

定义Clone的分配策略

- AT CLUSTER WIDE

- 在集群系统的所有集群组的所有集群成员中分配clone
- 添加集群组集群成员时可通过**ALTER TABLE name REBALANCE**重新分配clone

- AT CLUSTER GROUP group_list
  - 在指定集群组的所有集群成员中分配clone
  - 在指定集群组添加集群成员时可通过**ALTER TABLE name REBALANCE**重新分配clone
  - 添加集群组不影响clone的重新分配
- 省略时默认为AT CLUSTER WIDE

<hash sharding strategy>

将表的数据以sharding key的hash值为准分割shard

SHARDING BY [HASH] (column_list)

定义hash sharding的sharding key

- 最多可列出32个column
- 无法使用相同的column
- 无法使用LONG VARCHARLONG VARBINARY类型的column

<hash shard count>

定义要分割的hash shard的数量

Shard的数量可定义1~512

省略时默认值为24

<hash shard placement>

定义hash shard的分配策略

- AT CLUSTER WIDE
  - 在集群系统的所有集群组的所有集群成员中分配shard
  - 添加集群组集群成员时可通过**ALTER TABLE name REBALANCE**重新分配shard
- AT CLUSTER GROUP group_list
  - 在指定集群组的所有集群成员中分配hash shard
  - group_list的数量应小于或等于<hash shard count>的值
  - 与range shardlist shard不同hash shard无法指定分配特定shard的集群组系统自动决定分配shard的集群组
  - 在指定集群组添加集群成员时可通过**ALTER TABLE name REBALANCE**重新分配shard
  - 添加集群组不影响hash shard的重新分配
- 省略时默认值为AT CLUSTER WIDE

<range sharding strategy>

将表的数据以sharding key的范围值为准分割shard

SHARDING BY RANGE (column_list)

定义range sharding的sharding key

- 最多可列出32个column
- 无法使用相同的column

- 无法使用LONG VARCHARLONG VARBINARY类型的column

<cluster-wide range shard placement>

将Range shard自动分配到集群系统的所有集群组

描述<range shard definition>之前描述AT CLUSTER WIDE语句

添加集群组和集群成员时使用**ALTER TABLE name REBALANCE** 语句可自动重新分配shard

- 创建range sharded table
- 在原有cluster group (g1, g2, g3)分配6个shard

```
CREATE TABLE t1
(
   id   INTEGER,
   name VARCHAR(32)
)
SHARDING BY RANGE (id)
    AT CLUSTER WIDE
    SHARD s1 VALUES LESS THAN ( 200000 ),
    SHARD s2 VALUES LESS THAN ( 400000 ),
    SHARD s3 VALUES LESS THAN ( 500000 ),
    SHARD s4 VALUES LESS THAN ( 600000 ),
    SHARD s5 VALUES LESS THAN ( 800000 ),
    SHARD s6 VALUES LESS THAN ( MAXVALUE )
;
```

- 添加cluster group

```
CREATE CLUSTER GROUP g4

       CLUSTER MEMBER g4n1 HOST '192.168.0.41' PORT 10401

;
```

- 重新分配range shard
- 包括已添加的g4在cluster group g1g2g3g4中重新分配6个shard

```
ALTER TABLE t1 REBALANCE;
```

<group-specific range shard placement>

将range shard分配到指定集群组中

与<range shard definition>同时描述分配该shard的AT CLUSTER GROUP group_name语句

在指定的集群组中添加集群成员时可使用**ALTER TABLE name REBALANCE**自动重新分配shard

添加集群组不影响range shard的重新分配

- 创建range sharded table
- 在指定cluster group分配各个range shard

```
CREATE TABLE t1

(

   id   INTEGER,

   name VARCHAR(32)

)
```

```
SHARDING BY RANGE (id)
    SHARD s1 VALUES LESS THAN ( 200000 )   AT CLUSTER GROUP g1,
    SHARD s2 VALUES LESS THAN ( 400000 )   AT CLUSTER GROUP g2,
    SHARD s3 VALUES LESS THAN ( 500000 )   AT CLUSTER GROUP g3,
    SHARD s4 VALUES LESS THAN ( 600000 )   AT CLUSTER GROUP g2,
    SHARD s5 VALUES LESS THAN ( 800000 )   AT CLUSTER GROUP g3,
    SHARD s6 VALUES LESS THAN ( MAXVALUE ) AT CLUSTER GROUP g1
;
```

- 添加cluster group

```
CREATE CLUSTER GROUP g4
       CLUSTER MEMBER g4n1 HOST '192.168.0.41' PORT 10401
;
```

- 重新部署range shard
- 新创建的cluster group g4中不分配shard

```
ALTER TABLE t1 REBALANCE;
```

<range shard definition>

SHARD range_name应在表中是唯一的

最多可定义512个<range shard definition>

列出的<range shard definition>按照<range value clause>的顺序排列使用互不相同的<range

value clause>

将所有值定义为MAXVALUE的的<range shard definition>叫做MAX shard

MAX shard必须存在并且只能存在一个

- 应包含MAX shard

```
gSQL>
CREATE TABLE t1
(
   id INTEGER,
   name VARCHAR(32)
)
SHARDING BY RANGE (id)
   AT CLUSTER WIDE
   SHARD s1 VALUES LESS THAN ( 100000 ),
   SHARD s2 VALUES LESS THAN ( 200000 ),
   SHARD s3 VALUES LESS THAN ( MAXVALUE )
;

Table created.
```

- 不包含MAX shard时报错

```
gSQL>
CREATE TABLE t1
```

```
(
   id INTEGER,
   name VARCHAR(32)
)
SHARDING BY RANGE (id)
   AT CLUSTER WIDE
   SHARD s1 VALUES LESS THAN ( 100000 ),
   SHARD s2 VALUES LESS THAN ( 200000 ),
   SHARD s3 VALUES LESS THAN ( 300000 )
;

ERR-42000(16377): MAX shard not defined :
   SHARD s3 VALUES LESS THAN ( 300000 )
   *
ERROR at line 10:
```

<range value clause>

<range value>应为常数值或最大值的MAXVALUE

<range value>不可使用NULL值

- (O) SHARD s1 VALUES LESS THAN (1)
- (O) SHARD s2 VALUES LESS THAN (1 + 1)
- (O) SHARD s3 VALUES LESS THAN (MAXVALUE)
- (X) SHARD s4 VALUES LESS THAN (SYSDATE)

- (X) SHARD s5 VALUES LESS THAN (NULL)

MAXVALUE为始终比其他值大的值包含null值

有多个sharding key时MAXVALUE后仅可指定MAXVALUE

- (O) SHARD s1 VALUES LESS THAN (100, MAXVALUE)
- (X) SHARD s2 VALUES LESS THAN (MAXVALUE, 100)
- (O) SHARD s3 VALUES LESS THAN (MAXVALUE, MAXVALUE)

使用多个column定义sharding key时与如下SHARD s3相同必须要有一个将所有值以MAXVALUE列出的MAX shard

```
CREATE TABLE t1
(
   id INTEGER,
   name VARCHAR(32)
)
SHARDING BY RANGE (id, name)
   AT CLUSTER WIDE
   SHARD s1 VALUES LESS THAN ( 100000, MAXVALUE ),
   SHARD s2 VALUES LESS THAN ( 200000, 20000 ),
   SHARD s3 VALUES LESS THAN ( MAXVALUE, MAXVALUE )
;
```

<list sharding strategy>

将表数据以sharding key的罗列值为准进行shard分割

SHARDING BY LIST (column_name)

定义list sharding的sharding key

- 仅可使用一个column
- 无法使用LONG VARCHARLONG VARBINARY类型的column

<cluster-wide list shard placement>

将list shard自动分配到集群系统的所有集群组

描述<list shard definition>之前描述AT CLUSTER WIDE语句

添加集群组集群成员时可使用**ALTER TABLE name REBALANCE**自动重新配置shard

- 创建list sharded table
- 在原有cluster group的g1g2g3中分配5个shard

```
CREATE TABLE city
(
   id   INTEGER,
   name VARCHAR(32)
)
SHARDING BY LIST (name)
```

```
    AT CLUSTER WIDE
    SHARD s1 VALUES IN ( 'SEOUL' ),
    SHARD s2 VALUES IN ( 'PUSAN', 'ULSAN', 'DAEGU' ),
    SHARD s3 VALUES IN ( 'DAEJEON', 'GWANGJU' ),
    SHARD s4 VALUES IN ( 'ANSAN', 'GOYANG' ),
    SHARD s5 VALUES IN ( DEFAULT )
;
```

- 添加集群组

```
CREATE CLUSTER GROUP g4
       CLUSTER MEMBER g4n1 HOST '192.168.0.41' PORT 10401
;
```

- 重新配置list shard
- 包括已添加的g4在g1g2g3g4集群组配置5个shard

```
ALTER TABLE t1 REBALANCE;
```

<group-specific list shard placement>

将list shard分配到指定集群组

与<list shard definition>一起描述分配该shard的AT CLUSTER GROUP group_name语句

在指定集群添加集群成员时可通过**ALTER TABLE name REBALANCE**语句自动重新分配shard

添加集群组不影响list shard的重新分配

- 创建list sharded table
- 将各list shard配置到指定cluster group

```
CREATE TABLE city
(
   id   INTEGER,
   name VARCHAR(32)
)
SHARDING BY LIST (name)
    SHARD s1 VALUES IN ( 'SEOUL' )                   AT CLUSTER GROUP g1,
    SHARD s2 VALUES IN ( 'PUSAN', 'ULSAN', 'DAEGU' ) AT CLUSTER GROUP g2,
    SHARD s3 VALUES IN ( 'DAEJEON', 'GWANGJU' )      AT CLUSTER GROUP g3,
    SHARD s4 VALUES IN ( 'ANSAN', 'GOYANG' )         AT CLUSTER GROUP g2,
    SHARD s5 VALUES IN ( DEFAULT )                   AT CLUSTER GROUP g1
;
```

- 添加集群组

```
CREATE CLUSTER GROUP g4
       CLUSTER MEMBER g4n1 HOST '192.168.0.41' PORT 10401
;
```

- 重新分配list shard
- 已添加的集群组g4中不分配shard

```
ALTER TABLE t1 REBALANCE;
```

<list shard definition>

LIST list_name应在表中是唯一的

最多可定义512个<list shard definition>

列出的<list shard definition>的所有<list value>值应为互不相同的值

DEFAULT指除了列出的所有<list value>外剩余的所有值

DEFAULT无法与其他值同时指定

包含DEFAULT的shard叫做DEFAULT shard

必须要有DEFALULT shard并且只能有一个

- 应包含DEFAULT shard

```
gSQL>
CREATE TABLE t1
(
   category INTEGER,
   name     VARCHAR(32)
)
SHARDING BY LIST (category)
   AT CLUSTER WIDE
   SHARD s1 VALUES IN ( 1, 3, 5, 7 ),
   SHARD s2 VALUES IN ( 2, 4, 6, 8 ),
   SHARD s3 VALUES IN ( DEFAULT )
;
```

```
Table created.
```

- 未包含DEFAULT shard时报错

```
gSQL>

CREATE TABLE t1
(
   category INTEGER,
   name     VARCHAR(32)
)
SHARDING BY LIST (category)
   AT CLUSTER WIDE
   SHARD s1 VALUES IN ( 1, 3, 5, 7 ),
   SHARD s2 VALUES IN ( 2, 4, 6, 8 ),
   SHARD s3 VALUES IN ( 9, 10 )
;

ERR-42000(16385): DEFAULT shard not defined :
   SHARD s3 VALUES IN ( 9, 10 )
   *
ERROR at line 10:
```

<list value clause>

<list value>应为常数值

<list value>可使用NULL值或DEFAULT值

DEFAULT无法与其他值同时指定

- (O) SHARD s1 VALUES IN (1, 1 + 1, 3, 4)
- (O) SHARD s2 VALUES IN (5, 6, 7, NULL)
- (O) SHARD s3 VALUES IN (DEFAULT)
- (X) SHARD s4 VALUES IN (DEFAULT, 8, 9, 10)
- (X) SHARD s5 VALUES IN (current_timestamp, systimestamp)
- (X) SHARD s6 VALUES IN (c1, c2)

<table physical attribute clause>

定义表的物理属性信息

- PCTFREE integer
  - 定义
    - 在页中修改或更新row时为行大小的增加而预留的空间
    - 最初输入时不使用此空间
    - PCTFREE不足时修改或更新数据时将发生ROW MIGRATION
  - 可使用0-99之间的值
  - 省略时默认值为10
- PCTUSED integer

    - 定义
        - 在数据页中添加新的行之前行数据与overhead可使用的页的最小百分比
        - 即由于更新或删除原有数据值等导致其小于PCTUSED时仅限于这些页可以输入数据
    - 可使用0-99之间的值
    - 省略时默认值为40
- INITRANS integer
    - 定义
        - 可同时访问页的初始事务的数量
        - 访问索引的用户少时INITRANS设置的小访问索引的用户多时INITRANS设置的大
        - 必要时会自动扩展到已设置的MAXTRANS
    - 可使用1~32之间的值
    - 省略时默认值为4
- MAXTRANS integer
    - 定义
        - 可同时访问页的事务的最大数量
    - 可使用1~32之间的值
    - 省略时默认值为8

<index physical attribute clause>

定义索引的物理属性信息

- PCTFREE integer
    - 定义

        - 为了调整因key的插入引起的页分割频率而预留的空间
        - 只应用于bottom-up创建索引时
    - 可使用0~99之间的值
    - 省略时使用DEFAULT_INDEX_PCTFREE property中设置的值
- INITRANS integer
    - 与<table physical attribute clause>的INITRANS相同
- MAXTRANS integer
    - 与<table physical attribute clause>的MAXTRANS相同

<segment attr clause>

描述表的存储空间的信息

- INITIAL integer
    - 定义
        - 创建表时初期分配的物理空间的大小
        - 此大小align到表所属的表空间的EXTENT的大小后使用（ex：EXT大小为8192bytes时'INITIAL100'实际以8192bytes运行）
        - 此大小（align到表空间的EXTENT大小的大小）应大于或等于MINEXTENTS的大小小于或等于MAXEXTENTS的大小
    - 最小值为1最大值根据系统环境有所不同
    - 省略时默认为表所在的表空间的一个EXTENT大小
- NEXT integer
    - 定义
        - 添加表的物理空间时分配的物理空间大小

        - 此大小align到表所在的表空间的EXTENT大小后使用（例: EXT大小为8192 bytes时'NEXT 100'实际以8192 bytes运行）
        - NEXT根据当前表可用的剩余空间大小（MAXEXTENTS大小减去当前使用中的空间大小）如下运行
            - 剩余空间的大小为0时无法再扩展空间
            - 剩余空间的大小大于0且小于NEXT时分配与剩余空间大小相同的空间
            - 剩余空间的大小大于NEXT时分配与NEXT大小相同的空间
    - 最小值为1最大值根据系统环境有所不同
    - 省略时默认为表所在的表空间的一个EXTENT大小
- MINSIZE integer
    - 定义
        - 表需维持的最小空间大小
        - 该值应小于或等于MAXSIZE的值
    - 此大小align到表所在的表空间的EXTENT大小后使用
    - 最小值为1最大值根据系统环境有所不同
    - 小于2个EXTENT的大小时设置为2个EXTENT的大小
    - 省略时默认值为2个EXTENT的大小
- MAXSIZE integer
    - 定义
        - 表可分配的最大空间的大小
        - 该值应大于或等于MINSIZE
    - 此大小align到表所在的表空间的EXTENT大小
    - 最小值为1最大值根据系统环境有所不同
    - 省略时默认值为32 terabyte（35,184,372,088,832）
    - 即使指定大于32 terabyte的值也会修改成32 terabyte进行设置

<size clause>

指定文件的byte大小（未指定单位时默认为bytes）

- K : kilobytes
- M : megabytes
- G : gigabytes
- T : terabytes

TABLESPACE tablespace_name

指定存储表的表空间名称

省略TABLESPACE时使用执行语句的用户的默认tablespace_name

TABLESPACE index_tablespace_name

指定存储索引的表空间名称

省略TABLESPACE子句时使用用户的索引表空间

用户的索引表空间为NULL时DISK表使用用户的数据表空间内存表使用用户的默认临时表空间

<constraint characteristics>

定义约束条件的特点

定义约束条件时可以设置以下特点

- 约束条件的可延时与否(DEFERRABLE | NOT DEFERRABLE)

- 约束条件的检查时间点(<constraint check time>)

省略<constraint characteristics>时设置为NOT DEFERRABLE INITIALLY IMMEDIATE

DEFERRABLE | NOT DEFERRABLE

执行DML时不检查约束条件设置是否可以延时到执行COMMIT时检查约束条件

可延时的约束条件的检查时间点通过**SET CONSTRAINTS**语句控制

- NOT DEFERRABLE
  - 无法延长检查时间点执行INSERTDELETEUPDATE时检查约束条件
- DEFERRABLE
  - 通过**SET CONSTRAINTS**语句控制检查时间点
  - SET CONSTRAINTS constraint_name IMMEDIATE
    - 执行DML时检查约束条件
  - SET CONSTRAINTS constraint_name DEFERRED
    - 执行COMMIT时检查约束条件
- 未指定时默认值取决于<constraint check time>
  - 指定INITIALLY IMMEDIATE时为NOT DEFERRABLE
  - 指定INITIALLY DEFERRED时为DEFERRABLE
  - 未指定<constraint check time>时为NOT DEFERRABLE

<constraint check time>

为可延时的(DEFERRABLE)约束条件时设置检查时间点的初始值

- INITIALLY IMMEDIATE
  - 执行DML时检查约束条件
- INITIALLY DEFERRED
  - 执行COMMIT时检查约束条件
  - 不能与NOT DEFERRABLE一起使用
- 未指定时默认值为INITIALLY IMMEDIATE

可延时的约束条件相关详细说明参考**SET CONSTRAINTS**语句

<table global secondary index clause>

定义表的全局二级索引

- WITH GLOBAL SECONDARY INDEX [<index attributes> [...]] [TABLESPACE tablespace_name]
  - 在集群环境中创建表时创建全局二级索引
  - <index attribute>
    - 设置全局二级索引的index attribute
  - TABLESPACE tablespace_name
    - 指定创建全局二级索引的表空间
- WITHOUT GLOBAL SECONDARY INDEX
  - 在集群环境创建表时不创建全局二级索引

说明

约束条件的特性

SUNDB创建key约束条件时为了检查唯一性（uniqueness）自动创建索引

以下column不允许空值

- 有NOT NULL约束条件的column
- Column属于primary key约束条件时
- Column为identity column时

Cluster Table

在集群环境中表按照以下sharding策略中的一个策略管理数据

- Cloned table
  - 复制表的所有数据后部署到集群系统
- Hash sharded table
  - 将表的数据以sharding key的hash值为准分割为多个shard后部署到集群系统
- Range sharded table
  - 将表的数据以shrading key的范围值为准分割为多个shard后部署到集群系统
- List sharded table
  - 将表的数据以sharding key的list值为准分割为多个shard后部署到集群系统

创建表时考虑以下事项后决定sharding策略在集群系统中运行的表根据其特性分为code table与

fact table

- Code table
  - 产品目录供应者目录等数据变更少数据量少的表
  - 经常与fact table同时引用的表
- Fact table
  - 交易记录通话记录等数据变更多数据量多的表
  - 数据量多而需要sharding的表

建议在code table中使用<cloned strategy>fact table根据表的访问模式确定<table sharding strategy>

使用示例

以下为创建普通表的示例

```
gSQL> CREATE TABLE region
(
    r_regionkey    INTEGER
  , r_name         CHAR(25)
  , r_comment      VARCHAR(152)
);


Table created.
```

以下为创建表时定义column约束条件的示例

```
gSQL> CREATE TABLE supplier
(
    s_suppkey     INTEGER PRIMARY KEY
  , s_name        CHAR(25) NOT NULL
  , s_address     VARCHAR(40)
  , s_nationkey   INTEGER
  , s_phone       CHAR(15)
  , s_acctbal     NUMERIC(12,2)
  , s_comment     VARCHAR(101)
);

Table created.
```

以下为创建表时定义包含多个Column的约束条件的示例

```
gSQL> CREATE TABLE partsupp
(
    ps_partkey    INTEGER
  , ps_suppkey    INTEGER
  , ps_availqty   INTEGER
  , ps_supplycost NUMERIC(12,2)
  , ps_comment    VARCHAR(199)
  , CONSTRAINT ps_unique_key UNIQUE(ps_partkey, ps_suppkey)
);
```

```
Table created.
```

以下为创建表时定义包含可延时与否的约束条件的示例

```
gSQL> CREATE TABLE t1
(
    id      NUMBER         PRIMARY KEY
                           NOT DEFERRABLE INITIALLY IMMEDIATE
  , name    VARCHAR(128)   CONSTRAINT t1_nn NOT NULL
                           DEFERRABLE INITIALLY IMMEDIATE
  , addr    VARCHAR(1024)
  , CONSTRAINT t1_uk UNIQUE ( id, name )
                     DEFERRABLE INITIALLY DEFERRED
);

Table created.

gSQL> COMMIT;

Commit complete.
```

以下为创建表时定义拥有自动创建值与默认值的column的示例

```
CREATE TABLE customer
(
```

```
    c_custkey     INTEGER   GENERATED BY DEFAULT AS IDENTITY
  , c_name        VARCHAR(25)
  , c_address     VARCHAR(40) DEFAULT 'N/A'
  , c_nationkey   INTEGER
  , c_phone       CHAR(15)
  , c_acctbal     NUMERIC(12,2)
  , c_mktsegment  CHAR(10)
  , c_comment     VARCHAR(117)
);


Table created.
```

以下为创建表时指定要存储的表空间的示例

```
gSQL> CREATE TABLE lineitem
(
    l_orderkey      INTEGER
  , l_partkey       INTEGER
  , l_suppkey       INTEGER
  , l_linenumber    INTEGER
  , l_quantity      NUMERIC(12,2)
  , l_extendedprice NUMERIC(12,2)
  , l_discount      NUMERIC(12,2)
  , l_tax           NUMERIC(12,2)
  , l_returnflag    CHAR(1)
  , l_linestatus    CHAR(1)
```

```
  , l_shipdate      DATE

  , l_commitdate    DATE

  , l_receiptdate   DATE

  , l_shipinstruct  CHAR(25)

  , l_shipmode      CHAR(10)

  , l_comment       VARCHAR(44)

  , PRIMARY KEY (l_orderkey, l_linenumber) INDEX lineitem_pk_idx

TABLESPACE mem_temp_tbs

) TABLESPACE mem_data_tbs;


Table created.
```

以下为定义cluster-wide cloned table的示例在下例中将表的数据复制并部署到整个集群系统中

```
gSQL>

CREATE TABLE region

(

    r_regionkey   INTEGER

  , r_name        CHAR(25)

  , r_comment     VARCHAR(152)

)

CLONED

AT CLUSTER WIDE

;


Table created.
```

以下为定义group-specific cloned table的示例在下例中将表的数据复制并部署到用户指定的g1g2集群组中

```
gSQL>
CREATE TABLE region
(
    r_regionkey    INTEGER
  , r_name         CHAR(25)
  , r_comment      VARCHAR(152)
)
CLONED
AT CLUSTER GROUP g1, g2
;

Table created.
```

以下为定义cluster-wide hash sharded table的示例在下例中表的数据由ps_partkey column的hash值分割为24个shard所有shard自动部署到整个集群系统

```
gSQL>
CREATE TABLE partsupp
(
    ps_partkey     INTEGER
  , ps_suppkey     INTEGER
  , ps_availqty    INTEGER
  , ps_supplycost NUMERIC(12,2)
```

```
  , ps_comment    VARCHAR(199)
)
SHARDING BY HASH ( ps_partkey )
SHARD COUNT 24
AT CLUSTER WIDE
;

Table created.
```

以下为定义group-specific hash sharded table的示例在下例中表的数据由ps_partkey column的hash值分割为24个shard所有shard自动部署到指定的集群组g2g3

```
gSQL>
CREATE TABLE partsupp
(
    ps_partkey    INTEGER
  , ps_suppkey    INTEGER
  , ps_availqty   INTEGER
  , ps_supplycost NUMERIC(12,2)
  , ps_comment    VARCHAR(199)
)
SHARDING BY HASH ( ps_partkey )
SHARD COUNT 24
AT CLUSTER GROUP g2, g3
;
```

```
Table created.
```

以下为定义cluster-wide range sharded table的示例下例以D_ID column的范围值为准将表数据分割为8个shard所有shard自动部署到整个集群系统

```
gSQL>
CREATE TABLE DISTRICT (
    D_ID        INTEGER,
    D_W_ID      INTEGER,
    D_NAME      VARCHAR(10),
    D_STREET_1  VARCHAR(20),
    D_STREET_2  VARCHAR(20),
    D_CITY      VARCHAR(20),
    D_STATE     CHAR(2),
    D_ZIP       CHAR(9),
    D_TAX       NUMERIC(4,4),
    D_YTD       NUMERIC(15,2),
    D_NEXT_O_ID INTEGER,

    PRIMARY KEY (D_W_ID, D_ID) INDEX DISTRICT_PK_IDX
)
    SHARDING BY RANGE (D_ID)
    AT CLUSTER WIDE
    SHARD s1 VALUES LESS THAN ( 100 ),
    SHARD s2 VALUES LESS THAN ( 200 ),
    SHARD s3 VALUES LESS THAN ( 300 ),
```

```
    SHARD s4 VALUES LESS THAN ( 400 ),
    SHARD s5 VALUES LESS THAN ( 500 ),
    SHARD s6 VALUES LESS THAN ( 600 ),
    SHARD s7 VALUES LESS THAN ( 700 ),
    SHARD s8 VALUES LESS THAN ( MAXVALUE );
;

Table created.
```

以下为定义group-specific range sharded table的示例下例以NO_D_ID column的范围值为准将表数据分割为3个shard指定s1 shard部署到g1集群组s2 shard部署到g2集群组s3 shard部署到g3集群组

```
gSQL>
CREATE TABLE NEW_ORDER
(
    NO_O_ID INTEGER,
    NO_D_ID INTEGER,
    NO_W_ID INTEGER,

    PRIMARY KEY(NO_W_ID, NO_D_ID, NO_O_ID) INDEX NEW_ORDER_PK_IDX
)
    SHARDING BY RANGE (NO_D_ID)
    SHARD s1 VALUES LESS THAN ( 5 )        AT CLUSTER GROUP g1,
    SHARD s2 VALUES LESS THAN ( 8 )        AT CLUSTER GROUP g2,
    SHARD s3 VALUES LESS THAN ( MAXVALUE ) AT CLUSTER GROUP g3
```

```
;


Table created.
```

以下为定义cluster-wide list sharded table的示例下例以city column为准将list shard分割为5个shard所有shard自动部署到整个集群系统中

```
gSQL>
CREATE TABLE t1
(
    id   INTEGER
  , name VARCHAR(32)
  , city VARCHAR(128)
)
   SHARDING BY LIST (city)
      AT CLUSTER WIDE
      SHARD s1 VALUES IN ( 'seoul' ),
      SHARD s2 VALUES IN ( 'busan', 'ulsan' ),
      SHARD s3 VALUES IN ( 'suwon', 'ansan', 'osan' ),
      SHARD s4 VALUES IN ( 'goyang', 'paju', 'guri' ),
      SHARD s5 VALUES IN ( DEFAULT )
;


Table created.
```

以下为定义group-specific list sharded table的示例下例以city column为准将list shard分割为5个

shard各个shard自动部署到指定的集群组中

```
gSQL>
CREATE TABLE t1
(
    id   INTEGER
  , name VARCHAR(32)
  , city VARCHAR(128)
)
   SHARDING BY LIST (city)
      SHARD s1 VALUES IN ( 'seoul' )                  AT CLUSTER GROUP g1,
      SHARD s2 VALUES IN ( 'busan', 'ulsan' )         AT CLUSTER GROUP g2,
      SHARD s3 VALUES IN ( 'suwon', 'ansan', 'osan' ) AT CLUSTER GROUP g1,
      SHARD s4 VALUES IN ( 'goyang', 'paju', 'guri' ) AT CLUSTER GROUP g2,
      SHARD s5 VALUES IN ( DEFAULT )                  AT CLUSTER GROUP g3
;

Table created.
```

创建没有全局二级索引的表T1

```
gSQL> CREATE TABLE T1 ( I1 INTEGER, I1 CHAR(32) )  WITHOUT GLOBAL
SECONDARY INDEX;

Table created.
```

创建表T1后创建表T1的全局二级索引

```
gSQL> CREATE TABLE T1 ( I1 INTEGER, I1 CHAR(32) )  WITH GLOBAL SECONDARY
INDEX;


Table created.
```

创建表T1后在tablespace USER_DATA_TBS以logging option创建表T1的全局二级索引

```
gSQL> CREATE TABLE T1 ( I1 INTEGER, I1 CHAR(32) )
      WITH GLOBAL SECONDARY INDEX
      TABLESPACE USER_DATA_TBS;


Table created.
```

创建表T1后在tablespace USER_TEMP_TBS以nologging option创建表T1的全局二级索引

```
gSQL> CREATE TABLE T1 ( I1 INTEGER, I1 CHAR(32) )
      WITH GLOBAL SECONDARY INDEX
      TABLESPACE USER_TEMP_TBS;


Table created.
```

兼容性

标准SQL未定义下文

- TABLESPACE<physical attribute clause>等物理概念
- 标准SQL的DEFAULT中不能使用运算

Feature ID	说明	是否支持
T171	LIKE clause in table definition	X
F531	Temporary tables	X
S051	Create table of type	X
S043	Enhanced reference types	X
S081	Subtables	X
T173	Extended LIKE clause in table definition	X
T180	System-versioned tables	X
F692	Extended collation support	X
T174	Identity columns	O
T175	Generated columns	X
S071	SQL paths in function and type name resolution	X
F321	User authorization	O
T322	Extended roles	X
F762	CURRENT_CATALOG	O
F763	CURRENT_SCHEMA	O

Table 9-11 标准SQL兼容性

参考

相关内容参考下文

- **DROP TABLE**
- **ALTER TABLE**
- **CREATE TABLESPACE**
- **CREATE SCHEMA**
- **CREATE INDEX**
- **CREATE SEQUENCE**
- **SET CONSTRAINTS**
- **ALTER TABLE name ADD GLOBAL SECONDARY INDEX**

9.18 CREATE TABLE AS SELECT

功能

从查询结果中创建新表

语句

```
<table definition: AS query expression> ::=
    CREATE TABLE table_name
        [ ( column_name [, ...] ) ]
        [ <table sharding strategy> ]
        [ <table attribute clause> [, ...] ]
        [ TABLESPACE tablespace_name ]
        [ <table global secondary index clause> ]
        AS <query expression> [ WITH [ NO ] DATA ]
    ;


<table sharding strategy> ::=
      <cloned strategy>
    | <hash sharding strategy>
    | <range sharding strategy>
    | <list sharding strategy>
```

```
<cloned strategy> ::=
    CLONED [ <clone placement> ]

<clone placement> ::=
      AT CLUSTER WIDE
    | AT CLUSTER GROUP group_list

<hash sharding strategy> ::=
    SHARDING BY [HASH] ( column_list )
    [ <hash shard count> ]
    [ <hash shard placement> ]

<hash shard count> ::=
    SHARD COUNT integer

<hash shard placement> ::=
      AT CLUSTER WIDE
    | AT CLUSTER GROUP group_list

<range sharding strategy> ::=
    SHARDING BY RANGE ( column_list )
    { <cluster-wide range shard placement> | <group-specific range shard
placement> }
```

```
<cluster-wide range shard placement> ::=
    AT CLUSTER WIDE
    <range shard definition> [, ...]

<group-specific range shard placement> ::=
    <group-specific range shard definition> [, ...]

<group-specific range shard definition> ::=
    <range shard definition> AT CLUSTER GROUP group_name

<range shard definition> ::=
    SHARD range_name VALUES LESS THAN ( <range value clause> )

<range value clause> ::=
    <range value> [, ...]

<range value> ::=
      constant
    | MAXVALUE

<list sharding strategy> ::=
    SHARDING BY LIST ( column_name )
    { <cluster-wide list shard placement> | <group-specific list shard
placement> }
```

```
<cluster-wide list shard placement> ::=

    AT CLUSTER WIDE

    <list shard definition> [, ...]


<group-specific list shard placement> ::=

    <group-specific list shard definition> [, ...]


<group-specific list shard definition> ::=

    <list shard definition> AT CLUSTER GROUP group_name


<list shard definition> ::=

      SHARD shard_name VALUES IN ( <list value clause> )


<list value clause> ::=

    <list value> [, ...]


<list value> ::=

      constant

    | NULL

    | DEFAULT




<table attribute clause> ::=

      [ <table physical attribute clause> ]

    | [ STORAGE ( <segment attr clause> [...] ) ]
```

```
<table physical attribute clause> ::=

      PCTFREE integer

    | PCTUSED integer

    | INITRANS integer

    | MAXTRANS integer


<index physical attribute clause> ::=

      PCTFREE integer

    | INITRANS integer

    | MAXTRANS integer


<segment attr clause> ::=

      INITIAL <size_clause>

    | NEXT <size_clause>

    | MINSIZE <size_clause>

    | MAXSIZE <size_clause>


<size clause> ::=

      integer [ K | M | G | T ]


<table global secondary index clause> ::=

      WITH GLOBAL SECONDARY INDEX [ <index attributes> [...] ]
[ TABLESPACE tablespace_name ]

    |  WITHOUT GLOBAL SECONDARY INDEX
```

使用范围及访问权限

用户应满足以下条件才能执行<table definition:AS query expression>语句

- 创建表的权限
  - 参考**CREATE TABLE**语句的访问权限
- SELECT访问权限
  - 参考**SELECT**语句的访问权限

语句规则及参数

table_name

要创建的表的名称

详细内容参考**table_name**语句

column_name_list

构成表的column名应在表内是唯一的名称而且column数量应与SELECT结果的column数量相同

未指定时使用<query expression>的SELECT语句的column名

但是SELECT语句有不是column的expression(function, operation, subquery等)时应指定alias或column name

Column名长度应小于128 byte

WITH [NO] DATA

指定为WITH DATA时SELECT结果将插入到要创建的表中

指定为WITH NO DATA时SELECT结果不会插入到要创建的表中

未指定时与指定WITH DATA运行方式相同

other syntax

其他语句规则参考**CREATE TABLE**语句的syntax

说明

执行CREATE TABLE AS SELECT语句时在SELECT list中指定有NOT NULL约束条件的column时新的表中也会创建NOT NULL约束条件但是可延时的NOT NULL约束条件不会在新的表中创建NOT NULL约束条件

非指定生成NOT NULL约束条件而是像primary keyidentity column等本身有NOT NULL属性时新的表中不会创建NOT NULL约束条件

使用示例

以下为执行CREATE TABLE AS SELECT语句的示例

```
gSQL> CREATE TABLE recent_orders
```

```
                AS SELECT order_id, order_item, order_date
                   FROM orders
                   WHERE order_date >= '2015-03-03'
Table created.
```

以下为指定column名的示例

```
gSQL> CREATE TABLE recent_orders ( order_id, order_item, order_date )
                AS SELECT order_id, order_item, order_date
                   FROM orders
                   WHERE order_date >= '2015-03-03'
Table created.
```

以下为SELECT list中有函数的示例

```
gSQL> CREATE TABLE recent_orders ( order_id, order_item, order_date )
                AS SELECT order_date, COUNT(*)
                   FROM orders
                   WHERE order_date >= '2015-03-03'
                   GROUP BY order_date;
Table created.
```

以下为有WITH DATA的情况

```
gSQL>CREATE TABLE orders
(
    order_id   NUMBER
```

```
  , order_item VARCHAR(128)
  , order_date DATE
);
gSQL> COMMIT;
gSQL> INSERT INTO orders VALUES ( 1, 'Pen', '2010-01-01' );
gSQL> INSERT INTO orders VALUES ( 2, 'Book', '2015-03-03' );
gSQL> COMMIT;
gSQL> CREATE TABLE recent_orders
                AS SELECT order_id, order_item, order_date
                   FROM orders
                   WHERE order_date >= '2015-03-03'
      WITH DATA;
Table created.
gSQL> SELECT COUNT(*) FROM recent_orders;

COUNT(*)
--------
       1

1 row selected.
```

以下为有WITH NO DATA的情况

```
gSQL>CREATE TABLE orders
(
    order_id   NUMBER
```

```
  , order_item VARCHAR(128)
  , order_date DATE
);
gSQL> COMMIT;
gSQL> INSERT INTO orders VALUES ( 1, 'Pen', '2010-01-01' );
gSQL> INSERT INTO orders VALUES ( 2, 'Book', '2015-03-03' );
gSQL> COMMIT;
gSQL> CREATE TABLE recent_orders
                AS SELECT order_id, order_item, order_date
                   FROM orders
                   WHERE order_date >= '2015-03-03'
      WITH NO DATA;
Table created.
gSQL> SELECT COUNT(*) FROM recent_orders;

COUNT(*)
--------
       0

1 row selected.
```

兼容性

CREATE TABLE AS SELECT语句遵循SQL标准但以下为标准的扩展

- 标准中SELECT语句外面的括号为必选但Sundb中为可选
- 标准中WITH [NO] DATA语句为必选但Sundb中为可选
- Sundb表空间概念是标准中没有的扩展概念

Feature ID	说明	是否支持
T172	AS subquery clause in table definition	O

Table 9-12 标准SQL兼容性

参考

相关内容参考下文

- **CREATE TABLE**
- **SELECT**

9.19 CREATE TABLESPACE

功能

创建表空间

语句

```
<create tablespace statement> ::=
      <memory data tablespace statement>
    | <memory temporary tablespace statement>
    ;
```

使用范围及访问权限

用户需要有CREATE TABLESPACE ON DATABASE权限才能执行<create tablespace statement>语句

执行语句的用户对创建的表空间有CREATE OBJECT ON TABLESPACE权限

为了在创建的表空间创建对象需要有以下权限中的一个

- 对表空间有CREATE OBJECT ON TABLESPACE
- USAGE TABLESPACE ON DATABASE

语句规则及参数

<memory data tablespace statement>

- [MEMORY] TEMPORARY

存储语句处理过程中的中间结果等临时对象或no logging索引的临时表空间

可以省略MEMORY保留字

<memory data tablespace clause>

定义内存数据表空间

详细内容参考**CREATE MEMORY DATA TABLESPACE**

<memory temporary tablespace definition>

定义内存临时表空间

详细内容参考**CREATE MEMORY TEMPORARY TABLESPACE**

说明

参考各语句的详细说明

使用示例

参考各语句的详细使用示例

兼容性

标准SQL未定义表空间概念

参考

相关内容参考下文

- **DROP TABLESPACE**
- **ALTER TABLESPACE**

9.20 CREATE USER

功能

定义数据库用户

语句

```
<user definition> ::=
    CREATE USER user_identifier IDENTIFIED BY password
    [ PROFILE { profile_name | DEFAULT | NULL } ]
    [ PASSWORD EXPIRE ]
    [ ACCOUNT { LOCK | UNLOCK } ]
    [ DEFAULT TABLESPACE tablespace_name ]
    [ TEMPORARY TABLESPACE tablespace_name ]
    [ INDEX TABLESPACE { tablespace_name | NULL } ]
    [ <schema clause> ]
    ;


<schema clause> ::=
      WITH SCHEMA [schema_name]
    | WITHOUT SCHEMA
```

使用范围及访问权限

用户需要有CREATE USER ON DATABASE权限才能执行<user definition>语句

创建的user_identifier用户有通过<schema clause>创建的SCHEMA的所有者权限

> Note:
>
> 创建的user_identifier不会有额外的权限
>
> user_identifier用户为了连接数据库并执行SQL语句需有相应权限

语句规则及参数

user_identifier

要创建的用户的名称

用户名(user_identifier)与角色名(role_name)应为唯一

user_identifier的长度应小于128 byte

password

加密存储为要创建的用户的密码

密码长度应小于128 byte

密码区分大小写

password应以英文字母开头可包含英文字母数字underscore(_)$

为了使用此外的特殊符号应以双引号(")括住

PROFILE { profile_name | DEFAULT | NULL }

分配密码管理策略的profile

- PROFILE profile_name
  - 分配用户生成的profile_name
- PROFILE DEFAULT
  - 分配默认profile "DEFAULT"
- PROFILE NULL
  - 不分配profile

省略PROFILE时与PROFILE NULL相同不应用profile

密码管理策略相关详细内容参考**CREATE PROFILE**

PASSWORD EXPIRE

使用户密码过期

用于强制用户在登录之前修改密码

ACCOUNT { LOCK | UNLOCK }

- ACCOUNT LOCK
  - 锁定用户账号
- ACCOUNT UNLOCK

- 解除账号锁定

DEFAULT TABLESPACE tablespace_name

指定存储用户创建的表索引(LOGGING)等对象的默认表空间

省略DEFAULT TABLESPACE时指定为数据库创建时定义的default data tablespace (MEM_DATA_TBS)

TEMPORARY TABLESPACE tablespace_name

指定存储用户生成的临时表索引(NO LOGGING)语句处理过程中的中间结果的表空间

省略TEMPORARY TABLESPACE时指定为数据库创建时定义的default temporary tablespace (MEM_TEMP_TBS)

INDEX TABLESPACE { tablespace_name | NULL }

指定存储用户创建的索引对象的默认表空间

- 指定INDEX TABLESPACE tablespace_name
  - 指定数据表空间时成为LOGGING索引
  - 指定临时表空间时成为NOLOGGING索引
- INDEX TABLESPACE NULL
  - 不指定索引表空间

省略INDEX TABLESPACE时为INDEX TABLESPACE NULL

<schema clause>

创建用户默认使用的SCHEMA

应为数据库中唯一的SCHEMA名

- WITH SCHEMA [schema_name]
  - 无schema_name时创建与user_identifier相同名称的SCHEMA
  - user的SCHEMA PATH设置如下
    - schema_name, PUBLIC
- WITHOUT SCHEMA
  - 不创建user拥有的SCHEMA
  - user的SCHEMA PATH设置为PUBILC

不指定<schema clause>时默认值为WITH SCHEMA创建与user_identifier相同名称的SCHEMA

用户拥有的SCHEMA可以通过**CREATE SCHEMA**语句追加创建

说明

用户为由权限集合构成的授权（authorization ）对象

最初执行<user definition>语句时创建没有任何权限的用户需要赋予适当的权限

SUNDB的user与schema是1:N的关系

即用户可以没有自己的SCHEMA或可以有多个SCHEMA

标准SQL未明确定义userschemadatabase等non-schema对象之间的关系不同的数据库对non-

schema对象之间的关系有不同的定义

Note:

各DBMS的user与schema的关系

* Oracle

** User：schema = 1：1的关系

* DB2

** 与OS user相同

** 无创建并删除User的额外的SQL语句

* Postgres

** User：Schema = 1：N的关系

* MySQL

** Database：schema = 1：1的关系

** User为database（schema）的下层对象

使用示例

创建用户后为了使此用户创建对象并操作数据需要赋予如下权限

以下为创建用户并对其用户赋予权限的示例

- Create a user.

```
gSQL> CREATE USER u1 IDENTIFIED BY u1_password
             DEFAULT   TABLESPACE mem_data_tbs
             TEMPORARY TABLESPACE mem_temp_tbs
             INDEX TABLESPACE NULL;

User created.

gSQL> COMMIT;

Commit complete.
```

- Grant database privileges.

```
gSQL> GRANT CREATE SESSION ON DATABASE TO u1;

Grant succeeded.

COMMIT;

Commit complete.
```

- Grant schema privileges.

```
GRANT CREATE TABLE, CREATE VIEW, CREATE INDEX, CREATE SEQUENCE, ADD
```

```
CONSTRAINT
      ON SCHEMA u1 TO u1;

Grant succeeded.

COMMIT;

Commit complete.
```

- Grant tablespace privileges.

```
GRANT CREATE OBJECT ON TABLESPACE mem_data_tbs TO u1;

Grant succeeded.

GRANT CREATE OBJECT ON TABLESPACE mem_temp_tbs TO u1;

Grant succeeded.

COMMIT;

Commit complete.
```

以下为用户创建对象的示例

• It needs CREATE SESSION ON DATABASE.

```
gSQL> \connect u1 u1_password
```

• It needs CREATE TABLE ON SCHEMA u1.

• It needs CREATE OBJECT ON TABLESPACE mem_data_tbs.

```
gSQL> CREATE TABLE u1.t1 ( c1 INTEGER, c2 INTEGER ) TABLESPACE
mem_data_tbs;

Table created.

gSQL> COMMIT;
```

• It needs CREATE INDEX ON SCHEMA u1.

• It needs CREATE OBJECT ON TABLESPACE mem_temp_tbs.

```
gSQL> CREATE INDEX u1.idx ON t1 (c2) TABLESPACE mem_temp_tbs;

Index created.

gSQL> COMMIT;
```

• It needs ADD CONSTRAINT ON SCHEMA u1.

```
gSQL> ALTER TABLE t1 ADD CONSTRAINT u1.t1_pk PRIMARY KEY (c1) ;

Table altered.
```

```
gSQL> COMMIT;
```

• It needs CREATE SEQUENCE ON SCHEMA u1.

```
gSQL> CREATE SEQUENCE u1.seq;


Sequence created.


gSQL> COMMIT;


gSQL> INSERT INTO u1.t1 VALUES ( u1.seq.NEXTVAL, u1.seq.NEXTVAL );


1 row created


gSQL> COMMIT;
```

兼容性

标准SQL定义用户的概念但未定义生成及删除用户的相关SQL语句

参考

相关内容参考下文

- DROP USER
- ALTER USER
- CREATE SCHEMA

9.21 CREATE VIEW

功能

定义视图

语句

```
<view definition> ::=
    CREATE [ OR REPLACE ] [ FORCE | NO FORCE ]
        VIEW view_name [ ( column_name [, ...] ) ]
        AS <query expression>
    ;
```

使用范围及访问权限

用户应满足以下条件才能执行<view definition>语句

- 为了创建视图需要有以下权限中的一个
  - 对视图所在的SCHEMA有(CREATE VIEW或CONTROL SCHEMA) ON SCHEMA
  - CREATE ANY VIEW ON DATABASE
- 使用OR REPLACE时如果视图已存在则需要有删除现有视图的以下权限中的一个
  - 该视图的所有者

    - 对该视图有CONTROL TABLE ON TABLE
    - 对视图所在的SCHEMA有(DROP VIEW或CONTROL SCHEMA) ON SCHEMA
    - DROP ANY VIEW ON DATABASE
- 对<query expression>语句中使用的所有表需要有以下权限中的一个
    - 在表的column中用于语句的所有Column有SELECT(columns) ON TABLE
    - 对表有(SELECT或CONTROL TABLE) ON TABLE
    - 对表所在的SCHEMA有(SELECT TABLE或CONTROL SCHEMA) ON SCHEMA
    - SELECT ANY TABLE ON DATABASE
- 如下决定创建的视图的所有者
    - 视图所属的schema的所有者
    - 视图所属的schema为PUBLIC时执行语句的用户
- 视图的所有者对创建的视图有以下权限
    - SELECT ON TABLE WITH GRANT OPTION
    - INSERT ON TABLE WITH GRANT OPTION
    - UPDATE ON TABLE WITH GRANT OPTION
    - DELETE ON TABLE WITH GRANT OPTION
    - TRIGGER ON TABLE
    - LOCK ON TABLE WITH GRANT OPTION
    - ALTER ON TABLE WITH GRANT OPTION

语句规则及参数

[OR REPLACE]

如果视图已存在代替现有视图

[FORCE | NO FORCE]

- FORCE
  - 与<query expression>的有效性与否无关创建视图
- NO FORCE
  - <query expression>为有效时创建视图
- 默认值为NO FORCE

view_name

要创建的视图名应在SCHEMA内是唯一的

与schema_name.view_name相同可定义视图所在的SCHEMA省略schema_name时使用执行语句的用户的默认SCHEMA名

视图名的长度应小于128 byte

[(column_name [, ...])]

定义构成视图的column名

各column名应在视图内是唯一的

Column数量应与SELECT的结果column的数量相同

省略column名的列表时使用<query expression>的SELECT的column名

AS <query expression>

创建视图的**SELECT**语句

<query expression>不能包含以下变量

- host parameter
- SQL parameter
- dynamic parameter
- embedded variable
- SEQUENCE对象

说明

视图是向查询赋予名称的对象使用方式与table类似

执行包含视图的查询时视图将解析为视图定义中的查询例如如下视图参考的表结构发生变化时视图定义中的星号(*)将自动以变更的表结构为准重新解析

```
gSQL> CREATE VIEW v1 AS SELECT * FROM t1;

gSQL> COMMIT;

gSQL> SELECT * FROM v1;


ID NAME

-- ---------

 1 leekmo
```

```
 2 mkkim

 3 egonspace


3 rows selected.


gSQL> ALTER TABLE t1 ADD COLUMN ( dept_id  INTEGER,  addr

VARCHAR(1024) );

gSQL> COMMIT;


gSQL> select * from v1;


ID NAME      DEPT_ID ADDR

-- --------- ------- ----

 1 leekmo       null null

 2 mkkim        null null

 3 egonspace    null null


3 rows selected.
```

通过FORCE选项在语句有错的情况下创建视图或视图引用的表或视图发生变更删除的情况时对应的视图将受到影响

这些信息可以从INFORMATION_SCHEMA.VIEWS信息中查询

- IS_COMPILED Column
    - TRUE：正常创建视图

- FALSE: 使用FORCE选项在有错的情况下创建了视图

- IS_AFFECTED Column
  - TRUE : 视图参照的表或视图发生变更
  - FALSE: 创建视图并编译后视图引用的表或视图未发生变更

未限制视图的最大创建数量与可在视图内部生成的最大column数量因此在存储空间范围内可持续创建

使用示例

以下为创建视图的示例

```
gSQL> CREATE VIEW v1 AS SELECT * FROM t1 WHERE dept_id = 101;


View created.
```

以下为定义视图时定义column名的示例

```
gSQL> CREATE VIEW v1 ( v_id, v_name )
          AS SELECT id, name FROM t1 WHERE dept_id = 101;


View created.
```

以下为使用REPLACE选项删除现有视图后创建新视图的示例

```
gSQL> CREATE OR REPLACE VIEW v1(id, name)
```

```
            AS SELECT id, name FROM t1;


View created.
```

以下为使用FORCE选项在没有视图参照的对象的情况下强行创建该视图的示例

```
gSQL> CREATE FORCE VIEW v1
          AS SELECT * FROM t1 WHERE dept_id = 101;


ERR-01000(16243): Warning: View created with compilation errors
ERR-42000(16040): table or view does not exist :
    AS SELECT * FROM t1 WHERE dept_id = 101
                      *
ERROR at line 2:


View created.
```

兼容性

标准SQL未定义以下语句

- [OR REPLACE]语句
- [FORCE | NO FORCE]语句

Feature ID	说明	是否支持
T131	Recursive query	O
F751	View CHECK enhancements	X
S043	Enhanced reference types	X
T111	Updatable joins, unions, and columns	X
F852	Top-level <order by clause> in views	O
F864	Top-level <result offset clause> in views	O
F859	Top-level <fetch first clause> in views	O
S081	Subtables	X

Table 9-13 标准SQL兼容性

参考

相关内容参考下文

- **DROP VIEW**
- **ALTER VIEW**
- **SELECT**

9.22 DECLARE cursor_name

功能

声明游标

语句

```
<declare cursor> ::=

    DECLARE cursor_name <cursor properties> { FOR | IS } <cursor

specification>

    ;


<cursor properties> ::=

      [ <cursor sensitivity> ] [ <cursor scrollability>] ] CURSOR

[ <cursor holdability> ]

    | [ <odbc cursor type] CURSOR [ <cursor holdability> ]


<cursor sensitivity> ::=

      INSENSITIVE

    | SENSITIVE

    | ASENSITIVE
```

```
<cursor scrollability> ::=

      NO SCROLL

    | SCROLL


<cursor holdability> ::=

      WITH HOLD

    | WITHOUT HOLD


<odbc cursor type> ::=

      STATIC

    | KEYSET


<cursor specification> ::=

      statement_name

    | <cursor query>  [ <updatability clause> ]


<cursor query> ::=

      <select statement>

    | <insert returning query statement>

    | <update returning query statement>

    | <delete returning query statement>


<updatability clause> ::=

      FOR READ ONLY

    | FOR UPDATE [ OF <column name list> ] [ <lock wait mode> ]
```

```
<lock wait mode> ::=
    | WAIT
    | WAIT second
    | NOWAIT
```

使用范围及访问权限

使用statement_name的动态游标(dynamic cursor)可以在Embedded SQL中使用

根据<cursor query>的类型需要有以下访问权限

访问权限相关内容参考下文

- 参考**SELECT**语句的访问权限
- 参考**SELECT .. FOR UPDATE**语句的访问权限
- 参考**INSERT INTO name RETURNING**语句的访问权限
- 参考**UPDATE name RETURNING**语句的访问权限
- 参考**DELETE FROM name RETURNING**语句的访问权限

语句规则及参数

cursor_name

要声明的游标名

一个会话内游标名应该是唯一的

游标名的长度应小于128 byte

{ FOR | IS }

在标准SQL中作为语句的关键字使用FOR或IS中的一个

<cursor properties>

定义游标的属性

- 不指定<cursor sensitivity>时默认值为INSENSITIVE
- 不指定<cursor scrollability>时默认值为NO SCROLL
- 不指定<cursor holdability>时<cursor updatability>确定默认值

updatable query

游标的语句应为识别基础表的行变化或对行可获取锁的updatable query才能使用游标属性的SENSITIVE或FOR UPDATE

updatable query需要满足以下所有条件

- 最上层查询中不能有DISTINCT
  - (X) SELECT DISTINCT * FROM t1;
- 最上层查询中不能有GROUP BYHAVINGaggregation function
  - (X) SELECT MAX(c1) FROM t1;

- 不能为Returning query
  - (X) DELETE FROM t1 RETURNING c1;
- 不能有Set运算符
  - (X) SELECT * FROM t1 UNION ALL SELECT * FROM t2;
- FROM语句的表中应至少有一个updatable column
  - Join的表中不属于cross join的表的Column不是updatable column
    - FULL OUTER JOIN不是cross join
    - NATURAL JOIN不是cross join
    - INNER JOIN中使用USING时也不是cross join
  - 以下表的Column不是updatable column
    - Dictionary Table, Fixed Table, Performance View
  - View的Column不是updatable table

<cursor sensitivity>

指定是否能查看使用游标时影响查询结果的以下数据变化

- INSENSITIVE
  - 无法识别游标运行中发生变更的数据内容
- SENSITIVE
  - <cursor query>应为updatable query
  - 识别在与游标相同的事务中变更（UPDATE）删除（DELETE）的数据
  - 识别通过其他事务的COMMIT完成变更（UPDATE）删除（DELETE）的数据
- ASENSITIVE
  - 根据<cursor query>的类型决定INSENSITIVE或SENSITIVE

    - 为updatable query时为SENSITIVE
    - 非updatable query时为INSENSITIVE
- 省略时默认值为INSENSITIVE

<cursor scrollability>

指定是否按照顺序或不按照顺序Fetch游标的结果集

- NO SCROLL
    - 仅可按照顺序FETCH (FETCH NEXT)
- SCROLL
    - 可不按照顺序FETCH
- 省略时默认值为NO SCROLL

<cursor holdability>

设置打开游标并提交事务后是否维持游标

- WITH HOLD
    - 即使提交事务也维持游标
    - 不能与FOR UPDATE语句一起使用
    - 不能与**INSERT INTO name RETURNING**语句一起使用
    - 不能与**UPDATE name RETURNING**语句一起使用
    - 不能与**DELETE FROM name RETURNING**语句一起使用
    - 不能用于包括table commit action为ON COMMIT DELETE ROWS的global temporary table的查询

- WITHOUT HOLD
  - commit/rollback事务后关闭游标
- Rollback与Cursor
  - 回滚事务时关闭事务中的游标
  - 回滚至Savepoint则关闭savepoint之后创建的游标
- 未指定时<cursor holdability>的默认值取决于<cursor updatability>
  - 为FOR READ ONLY或不指定<cursor updatability>时默认值为WITH HOLD
  - 与FOR UPDATE语句一起使用时默认值为WITHOUT HOLD

<odbc cursor type>

作为标准ODBC的游标类型具有SCROLL属性

- STATIC CURSOR
  - 与标准SQL的INSENSITIVE SCROLL相同
  - 可不按顺序进行FETCH
  - 标准ODBC的static scroll cursor
- KEYSET CURSOR
  - 与标准SQL的ASENSITIVE SCROLL相同
  - 标准ODBC的keyset-driven scroll cursor
  - 根据以下特性确定sensitivity属性

Updatability	Query 类型	Sensitivity
FOR UPDATE	updatable query	SENSITIVE

Updatability	Query 类型	Sensitivity
FOR UPDATE	non-updatable query	query error
FOR READ ONLY	any query	INSENSITIVE
N/A	updatable query	SENSITIVE
N/A	non-updatable query	INSENSITIVE

Table 9-14 根据FOR [UPDATE / READ ONLY]语句与query类型的sensivitity确定

<cursor specification>

定义游标的对象query

使用statement_name时声明未指定query的动态游标(dynamic cursor)使用<cursor query>时声明指定query的静态游标(standing cursor)

statement_name

游标参照的statement_name可在embedded SQL中使用

statement_name应存在于执行<declare cursor>语句之前statement_name参照的SQL语句应为由**PREPARE statement_name**语句事先准备的query

不是query时执行**OPEN cursor_name**语句则报错

<cursor query>

游标中可使用的query类型如下

- **SELECT**
- **SELECT .. FOR UPDATE**
- **INSERT INTO name RETURNING**
- **UPDATE name RETURNING**
- **DELETE FROM name RETURNING**

<updatability clause>

指定是否使用游标变更row

- FOR READ ONLY
  - 声明读取专用游标
- FOR UPDATE
  - 声明可写的游标
  - 为了直到事务结束不被其他事务变更数据打开游标时获取该行数据的x lock
  - 不能与WITH HOLD语句一起使用
  - <cursor query>应为updatable query
- 未指定时默认为FOR READ ONLY

FOR UPDATE OF ...

打开游标时列出与获取lock相关的column

- FOR UPDATE OF中列出的column
  - 应为<select statement>的FROM子句中列出的表的可更新的column
  - 获取列出column的表的锁（lock）
- 仅使用FOR UPDATE时
  - 与列出<select statement>的FROM子句中列出的表的可更新的column有相同意义
  - 获取所有column的表的锁

<lock wait mode>

与FOR UPDATE语句一起使用并指定锁的获取方法

- WAIT
  - 打开游标时对查询结果的所有行获取锁
  - 一直等到可获取锁为止
- WAIT second
  - 打开游标时对查询结果的所有行获取锁
  - 指定时间内未获取锁则报错
  - 以秒为单位可以使用0 ~ 1000000000之间的值
- NOWAIT
  - 打开游标时对查询结果的所有行获取锁
  - 不能立即获取锁时报错
- 未指定时默认值为WAIT

说明

控制query的属性时如果使用DECLARE CURSOR语句与OPENFETCHCLOSE语句会控制服务器的游标因此相比使用ODBC statement或JDBC statement的游标使用性能会受到影响

执行query前可使用ODBC statement与JDBC statement控制游标属性通过DECLARE CURSOR语句的SQL游标属性控制方法与对应的标准ODBC和标准JDBC的游标属性控制方法如下

Property 分类	SUNDB cursor property	标准ODBC的游标属性设置	标准JDBC的游标属性设置
Sensitivity	INSENSITIVE	SQLSetStmtAttr(stmt, SQL_ATTR_CURSOR_SENSITIVITY, SQL_INSENSITIVE, len)	无法设置
	SENSITIVE	SQLSetStmtAttr(stmt, SQL_ATTR_CURSOR_SENSITIVITY, SQL_SENSITIVE, len)	无法设置
	ASENSITIVE	SQLSetStmtAttr(stmt, SQL_ATTR_CURSOR_SENSITIVITY, SQL_UNSPECIFIED, len)	无法设置
Scrollability	NO SCROLL	SQLSetStmtAttr(stmt, SQL_ATTR_CURSOR_SCROLLABLE, SQL_NONSCROLLABLE, len)	无法设置

Property 分类	SUNDB cursor property	标准ODBC的游标属性设置	标准JDBC的游标属性设置
	SCROLL	SQLSetStmtAttr(stmt, SQL_ATTR_CURSOR_SCROLLABLE, SQL_SCROLLABLE, len)	无法设置
Holdability	WITHOUT HOLD	无法设置	java.sql.Connection::prepareStatement(query, type, conc, ResultSet.CLOSE_CURSORS_AT_COMMIT)
	WITH HOLD	无法设置	java.sql.Connection::prepareStatement(query, type, conc, ResultSet.HOLD_CURSORS_OVER_COMMIT)

Table 9-15 ODBC/JDBC的游标属性控制

ODBC cursor type对应的SQL游标声明如下

ODBC cursor type	声明SQL cursor
SQLSetStmtAttr(stmt, SQL_ATTR_CURSOR_TYPE, SQL_CURSOR_FORWARD_ONLY, len)	NO SCROLL CURSOR
SQLSetStmtAttr(stmt, SQL_ATTR_CURSOR_TYPE, SQL_CURSOR_STATIC, len)	STATIC CURSOR
SQLSetStmtAttr(stmt, SQL_ATTR_CURSOR_TYPE, SQL_CURSOR_KEYSET_DRIVEN, len)	KEYSET CURSOR

Table 9-16 ODBC cursor type对应的SQL游标声明

JDBC cursor type对应的SQL游标声明如下

JDBC cursor type	声明SQL cursor
java.sql.Connection::prepareStatement(query, ResultSet.TYPE_FORWARD_ONLY, conc, hold)	INSENSITIVE NO SCROLL CURSOR
java.sql.Connection::prepareStatement(query, ResultSet.TYPE_SCROLL_INSENSITIVE, conc, hold)	INSENSITIVE SCROLL CURSOR
java.sql.Connection::prepareStatement(query, ResultSet.TYPE_SCROLL_SENSITIVE, conc, hold)	KEYSET CURSOR

Table 9-17 JDBC cursor type对应的SQL游标声明

使用示例

以下为使用interactive sql(gsql)声明游标并使用的简单示例

```
gSQL> DECLARE cur1 CURSOR FOR SELECT id, data FROM t1;


Cursor declared.


gSQL> OPEN cur1;
```

```
Cursor is open.


gSQL> \var v_id   INTEGER

gSQL> \var v_data VARCHAR(128)


gSQL> FETCH cur1 INTO :v_id, :v_data;


V_ID V_DATA

---- ------

   1 data_1


1 row fetched.


gSQL> FETCH cur1 INTO :v_id, :v_data;


V_ID V_DATA

---- ------

   2 data_2


1 row fetched.


gSQL> FETCH cur1 INTO :v_id, :v_data;


V_ID V_DATA

---- ------
```

```
   3 data_3


1 row fetched.


gSQL> FETCH cur1 INTO :v_id, :v_data;


V_ID V_DATA
---- ------
   4 data_4


1 row fetched.


gSQL> FETCH cur1 INTO :v_id, :v_data;


V_ID V_DATA
---- ------
   5 data_5


1 row fetched.


gSQL> FETCH cur1 INTO :v_id, :v_data;


no rows fetched.


gSQL> CLOSE cur1;
```

```
Cursor closed.
```

以下为声明KEYSET游标并按照顺序检索后完成对UPDATEDELETE语句的事务后反方向检索的示例

```
gSQL> DECLARE cur_keyset KEYSET CURSOR FOR SELECT id, data FROM t1;

Cursor declared.

gSQL> OPEN cur_keyset;

Cursor is open.

gSQL> \var v_id   INTEGER
gSQL> \var v_data VARCHAR(128)

gSQL> FETCH NEXT cur_keyset INTO :v_id, :v_data;

V_ID V_DATA
---- ------
   1 data_1

1 row fetched.
```

```
gSQL> FETCH NEXT cur_keyset INTO :v_id, :v_data;


V_ID V_DATA

---- ------

   2 data_2


1 row fetched.


gSQL> FETCH NEXT cur_keyset INTO :v_id, :v_data;


V_ID V_DATA

---- ------

   3 data_3


1 row fetched.


gSQL> FETCH NEXT cur_keyset INTO :v_id, :v_data;


V_ID V_DATA

---- ------

   4 data_4


1 row fetched.


gSQL> FETCH NEXT cur_keyset INTO :v_id, :v_data;
```

```
V_ID V_DATA
---- ------
   5 data_5

1 row fetched.

gSQL> FETCH NEXT cur_keyset INTO :v_id, :v_data;

no rows fetched.

gSQL> UPDATE t1 SET data = 'new data_2' WHERE id = 2;

1 row updated.

gSQL> COMMIT;

Commit complete.

gSQL> DELETE FROM t1 WHERE id = 4;

1 row deleted.

gSQL> COMMIT;
```

```
Commit complete.

gSQL> FETCH PRIOR cur_keyset INTO :v_id, :v_data;

V_ID V_DATA
---- ------
   5 data_5

1 row fetched.

gSQL> FETCH PRIOR cur_keyset INTO :v_id, :v_data;

no rows fetched.

gSQL> FETCH PRIOR cur_keyset INTO :v_id, :v_data;

V_ID V_DATA
---- ------
   3 data_3

1 row fetched.

gSQL> FETCH PRIOR cur_keyset INTO :v_id, :v_data;
```

```
V_ID V_DATA
---- ----------
   2 new data_2

1 row fetched.

gSQL> FETCH PRIOR cur_keyset INTO :v_id, :v_data;

V_ID V_DATA
---- ------
   1 data_1

1 row fetched.

gSQL> CLOSE cur_keyset;

Cursor closed.
```

以下为声明SCROLL游标并通过fetch orientation使用游标的示例

```
gSQL> DECLARE cur_scroll SCROLL CURSOR FOR SELECT id, data FROM t1;

Cursor declared.

gSQL> OPEN cur_scroll;
```

```
Cursor is open.


gSQL> \var v_id   INTEGER
gSQL> \var v_data VARCHAR(128)


gSQL> FETCH LAST cur_scroll INTO :v_id, :v_data;
V_ID V_DATA
---- ------
   5 data_5


1 row fetched.



gSQL> FETCH PRIOR cur_scroll INTO :v_id, :v_data;


V_ID V_DATA
---- ------
   4 data_4


1 row fetched.



gSQL> FETCH FIRST cur_scroll INTO :v_id, :v_data;
```

```
V_ID V_DATA

---- ------

   1 data_1


1 row fetched.



gSQL> FETCH ABSOLUTE 3 cur_scroll INTO :v_id, :v_data;


V_ID V_DATA

---- ------

   3 data_3


1 row fetched.



gSQL> FETCH RELATIVE -1 cur_scroll INTO :v_id, :v_data;


V_ID V_DATA

---- ------

   2 data_2


1 row fetched.
```

```
gSQL> FETCH ABSOLUTE 3 cur_scroll INTO :v_id, :v_data;


V_ID V_DATA
---- ------
   3 data_3


1 row fetched.


gSQL> CLOSE cur_scroll;


Cursor closed.
```

兼容性

<declare cursor>语句与标准SQL之间有以下区别

- 标准SQL的<cursor sensitivity>默认值为ASENSITIVESUNDB的默认值为INSENSITIVE
- 标准SQL中未定义以下<odbc cursor type>
  - STATIC CURSOR
  - KEYSET CURSOR
- 标准SQL的<cursor holdability>默认值为WITHOUT HOLDSUNDB的默认值根据<cursor updatability>有所不同
- 标准SQL中的<cursor query>仅使用<select statement>但SUNDB可以使用以下returning query
  - **INSERT INTO name RETURNING**

- **UPDATE name RETURNING**
- **DELETE FROM name RETURNING**

- 标准SQL<cursor updatability>的默认值取决于<select statement>但SUNDB的默认值为FOR READ ONLY
- 标准SQL中没有<lock wait mode>语句

Feature ID	说明	是否支持
F831	Full cursor update	O
T231	Sensitive cursors	O
F791	Insensitive cursors	O
F431	Read-only scrollable cursors	O
T471	Result sets return value	X
T551	Optional key words for default syntax	O
T111	Updatable joins, unions, and columns	X
B031	Basic dynamic SQL	O

Table 9-18 标准SQL兼容性

参考

相关内容参考下文

- **OPEN cursor_name**
- **FETCH cursor_name**
- **CLOSE cursor_name**
- **PREPARE statement_name**
- **SELECT**
- **SELECT .. FOR UPDATE**
- **INSERT INTO name RETURNING**
- **UPDATE name RETURNING**
- **DELETE FROM name RETURNING**

9.23 DELETE FROM

功能

删除表的row

语句

```
<delete statement: searched> ::=
    DELETE [ FROM ] table_name [ [ AS ] alias_name ]
        [ WHERE <search condition> ]
        [ <result offset clause> ]
        [ <fetch limit clause> ]
    ;

<result offset clause> ::=
    OFFSET skip_count [ ROW | ROWS ]

<fetch limit clause> ::=
      <fetch first clause>
    | <limit clause>

<fetch first clause> ::=
```

```
    FETCH [ FIRST | NEXT ] [ row_count ] [ ROW ONLY | ROWS ONLY ]


<limit clause>
    LIMIT { fetch_row_count | offset_row_count, fetch_row_count | ALL }
```

使用范围及访问权限

用户需要有以下权限中的一个才能执行<delete statement: searched>语句

- 对表有(DELETE或CONTROL TABLE) ON TABLE
- 对表所在的SCHEMA有(DELETE TABLE或CONTROL SCHEMA) ON SCHEMA
- DELETE ANY TABLE ON DATABASE

语句规则及参数

table_name

要删除row的对象表名

与schema_name.table_name相同可定义表所在的SCHEMA省略schema_name时使用执行语句的用户的默认SCHEMA名

[AS alias_name]

table_name的别名

WHERE <search condition>

删除满足WHERE条件的row

未指定WHERE条件时删除所有row

WHERE条件的详细内容参考**SELECT**语句的**where clause**

<result offset clause>

指定查询结果中要跳过的row数量

详细内容参考**SELECT**语句的**offset limit clause**

<fetch limit clause>

作为指定要fetch的row数量的语句有以下两种方法

- <fetch first clause>
  - 指定要fetch的row数量
  - 详细内容参考**SELECT**语句的**<fetch first clause>**
- <limit clause>
  - 指定要fetch的row数量或同时指定查询结果中要跳过的row数量与fetch的row数量
  - 详细内容参考**SELECT**语句的**<limit clause>**

说明

DELETE相关语句之间的区别

- **DELETE FROM**
  - 删除满足条件的多条行数据
  - 例: DELETE FROM t1 WHERE c1 = 0;
- **DELETE FROM name WHERE CURRENT OF cursor_name**
  - 删除游标当前所指的行数据
  - 例: DELETE FROM t1 WHERE CURRENT OF cursor;
- **DELETE FROM name RETURNING**
  - 删除满足条件的多条行数据可以通过与**SELECT**语句相同的方式(SQLFetch()等API)检索删除的行数据
  - 例: DELETE FROM t1 WHERE c1 = 0 RETURNING c2;
- **DELETE FROM name RETURNING .. INTO**
  - 删除一条以下的行数据删除的行数据为一条时通过RETURNING INTO语句的主机变量获取值
  - 例: DELETE FROM t1 WHERE c1 = 0 RETURNING c2 INTO :v1;

使用示例

以下为DELETE语句的简单示例

```
gSQL> DELETE FROM t1 WHERE id > 3;
```

```
2 rows deleted.
```

以下为使用<result offset clause>与<fetch first clause>语句在满足条件的行数据中跳过部分(2条)行数据后仅删除部分(2条)行数据的示例

```
gSQL> DELETE FROM t1 OFFSET 2 FETCH 2;

2 rows deleted.



gSQL> SELECT * FROM t1 ORDER BY 1;

ID DATA
-- ------
 1 data_1
 2 data_2
 5 data_5

3 rows selected.
```

兼容性

标准SQL的DELETE语句中未定义以下语句

- <result offset clause>
- <fetch limit clause>

Feature ID	说明	是否支持
F781	Self-referencing operations	X
T111	Updatable joins, unions, and columns	X

Table 9-19 标准SQL兼容性

参考

相关内容参考下文

- **DELETE FROM name WHERE CURRENT OF cursor_name**
- **DELETE FROM name RETURNING**
- **DELETE FROM name RETURNING .. INTO**
- **SELECT**

9.24 DELETE FROM name RETURNING

功能

删除表的行数据并检索删除的行数据

语句

```
<delete returning query statement> ::=
    DELETE [ FROM ] table_name [ [ AS ] alias_name ]
        [ WHERE <search condition> ]
        [ <result offset clause> ]
        [ <fetch limit clause> ]
        <returning clause>
    ;

<result offset clause> ::=
    OFFSET skip_count [ ROW | ROWS ]

<fetch limit clause> ::=
      <fetch first clause>
    | <limit clause>
```

```
<fetch first clause> ::=

    FETCH [ FIRST | NEXT ] [ row_count ] [ ROW ONLY | ROWS ONLY ]


<limit clause>

    LIMIT { fetch_row_count | offset_row_count, fetch_row_count | ALL }


<returning clause> ::=

    { RETURN | RETURNING } { * | { <value expression> [ [AS] alias_name] }

[, ...] }
```

使用范围及访问权限

用户应满足以下条件才能执行<delete returning query statement>语句

- 用户应满足以下条件才能执行DELETE语句
  - 对表有(DELETE或CONTROL TABLE) ON TABLE
  - 对表所在的SCHEMA有(DELETE TABLE或CONTROL SCHEMA) ON SCHEMA
  - DELETE ANY TABLE ON DATABASE
- 对用于RETURNING的所有column需要有以下权限中的一个
  - 对RETURNING语句中使用的所有Column有SELECT(columns) ON TABLE
  - 对表有(SELECT或CONTROL TABLE) ON TABLE
  - 对表所在的SCHEMA有(SELECT TABLE或CONTROL SCHEMA) ON SCHEMA
  - SELECT ANY TABLE ON DATABASE

语句规则及参数

table_name

要删除行数据的对象表名

[AS alias_name]

table_name的别名

WHERE <search condition>

删除满足WHERE条件的行数据

详细内容参考**DELETE FROM**语句

<result offset clause>

指定查询结果中要跳过的行数据条数

详细内容参考**DELETE FROM**语句

<fetch first clause>

指定要fetch的行数据条数

详细内容参考**DELETE FROM**语句

<limit clause>

指定要fetch的行数据条数或同时指定查询结果中要跳过的行数据条数与fetch的行数据条数

详细内容参考**DELETE FROM**语句

<returning clause>

删除的行数据为结果集在其中指定要检索的Column

- RETURNING语句作为结果返回通过DELETE语句删除的行数据
- <value expression>
  - 与 SELECT语句的<select list>相同但不能使用Aggregation等
- [[AS] alias_name]
  - 使用AS可指定value expression名

RETURN与RETURNING为相同意义的关键字

说明

详细内容参考**DELETE相关语句之间的区别**

使用示例

以下为删除满足条件的行数据并检索删除的行数据的示例

```
gSQL> DELETE FROM t1 WHERE id > 3 RETURNING *;


ID DATA
-- ------
 4 data_4
 5 data_5


2 rows deleted.
```

以下为使用RETURNING的运算查询删除的行数据信息的示例

```
gSQL> DELETE FROM t1
             WHERE id > 3
             RETURNING 'ID: ' || id || ', DATA: ' || data AS id_data;


ID_DATA
-------------------
ID: 4, DATA: data_4
ID: 5, DATA: data_5


2 rows deleted.
```

兼容性

标准SQL无<delete returning query statement>语句

参考

相关内容参考下文

- **DELETE FROM**
- **DELETE FROM name WHERE CURRENT OF cursor_name**
- **DELETE FROM name RETURNING .. INTO**
- **SELECT**

9.25 DELETE FROM name RETURNING .. INTO

功能

删除表的一条行数据并通过主机变量获取删除的行数据的值

语句

```
<delete returning query statement> ::=
    DELETE [ FROM ] table_name [ [ AS ] alias_name ]
        [ WHERE <search condition> ]
        [ <result offset clause> ]
        [ <fetch limit clause> ]
        <returning into clause>
    ;


<result offset clause> ::=
    OFFSET skip_count [ ROW | ROWS ]



<fetch limit clause> ::=
      <fetch first clause>
    | <limit clause>
```

```
<fetch first clause> ::=
    FETCH [ FIRST | NEXT ] [ row_count ] [ ROW ONLY | ROWS ONLY ]


<limit clause>
    LIMIT { fetch_row_count | offset_row_count, fetch_row_count | ALL }


<returning into clause> ::=
    { RETURN | RETURNING } { * | { <value expression> [ [AS] alias_name] }
[, ...] } INTO variable_name [, ...]
```

使用范围及访问权限

用户应满足以下条件才能执行<delete returning into statement>语句

- 用户需要有以下权限中的一个才能执行DELETE语句
  - 对表有(DELETE或CONTROL TABLE) ON TABLE
  - 对表所在的SCHEMA有(DELETE TABLE或CONTROL SCHEMA) ON SCHEMA
  - DELETE ANY TABLE ON DATABASE
- 对RETURNING中使用的所有Column需要有以下权限中的一个
  - 对RETURNING语句中使用的所有Column有SELECT(columns) ON TABLE
  - 对表有(SELECT或CONTROL TABLE) ON TABLE

- 对表所在的SCHEMA有(SELECT TABLE或CONTROL SCHEMA) ON SCHEMA
- SELECT ANY TABLE ON DATABASE

语句规则及参数

table_name

要删除行数据的对象表名

[AS alias_name]

table_name的别名

WHERE <search condition>

删除满足WHERE条件的行数据

详细内容参考**DELETE FROM**语句

<result offset clause>

指定查询结果中要跳过的行数据条数

详细内容参考**DELETE FROM**语句

<fetch first clause>

指定要fetch的行数据条数

详细内容参考**DELETE FROM**语句

<limit clause>

指定要fetch的行数据条数或同时指定查询结果中要跳过的行数与fetch的行数

详细内容参考**DELETE FROM**语句

<returning into clause>

- RETURNING .. AS ..
  - 参考**DELETE FROM name RETURNING**语句的<returning clause>
- INTO variable_name [, ...]
  - INTO中指定的变量数量应与RETURNING中指定的expression数量相同

说明

要删除的行数据应为一条以下

删除两条以上行数据时将报错

详细内容参考**DELETE相关语句之间的区别**

使用示例

以下为在interactive sql(gsql)中删除行数据并通过主机变量获取删除的行数据的示例

```
gSQL> \var v_id    INTEGER
gSQL> \var v_data  VARCHAR(128)

gSQL> DELETE FROM t1 WHERE id = 3 RETURNING id, data INTO :v_id, :v_data;

V_ID V_DATA
---- ------
   3 data_3

1 row deleted.
```

兼容性

标准SQL没有<delete returning into statement>

参考

相关内容参考下文

- **DELETE FROM**

- **DELETE FROM name WHERE CURRENT OF cursor_name**
- **DELETE FROM name RETURNING**
- **SELECT**

9.26 DELETE FROM name WHERE CURRENT OF cursor_name

功能

删除游标所指的一条行数据

语句

```
<delete statement: positioned> ::=
    DELETE [ FROM ] table_name [ [ AS ] alias_name ]
        WHERE CURRENT OF cursor_name
    ;
```

使用范围及访问权限

用户需要有可执行**DELETE FROM**语句的权限才能执行<delete statement: positioned>语句

语句规则及参数

table_name

要删除行数据的对象表名

[AS alias_name]

table_name的别名

cursor_name

cursor_name对应的游标需要满足以下条件

- 应为已打开的游标(参考**OPEN cursor_name**)
- 需要有通过游标FETCH的行数据(参考**FETCH cursor_name**）
- 游标使用的查询应可识别table_name(参考 **DECLARE cursor_name**）
- 游标应为可以更新table_name的游标(参考**DECLARE cursor_name**)

说明

详细内容参考**DELETE相关语句之间的区别**

使用示例

以下为在interactive sql(gsql)中声明FOR UPDATE游标并使用此游标删除行数据的示例

```
gSQL> DECLARE cur1 CURSOR FOR SELECT id, data FROM t1 FOR UPDATE;

Cursor declared.


gSQL> OPEN cur1;

Cursor is open.


gSQL> \var v_id   INTEGER
gSQL> \var v_data VARCHAR(128)

gSQL> FETCH cur1 INTO :v_id, :v_data;

V_ID V_DATA
---- ------
   1 data_1

1 row fetched.
```

```
gSQL> FETCH cur1 INTO :v_id, :v_data;


V_ID V_DATA

---- ------

   2 data_2


1 row fetched.


gSQL> DELETE FROM t1 WHERE CURRENT OF cur1;


1 row deleted.


gSQL> FETCH cur1 INTO :v_id, :v_data;


V_ID V_DATA

---- ------

   3 data_3


1 row fetched.


gSQL> FETCH cur1 INTO :v_id, :v_data;


V_ID V_DATA

---- ------

   4 data_4
```

```
1 row fetched.

gSQL> DELETE FROM t1 WHERE CURRENT OF cur1;

1 row deleted.

gSQL> FETCH cur1 INTO :v_id, :v_data;

V_ID V_DATA
---- ------
   5 data_5

1 row fetched.

gSQL> FETCH cur1 INTO :v_id, :v_data;

no rows fetched.

gSQL> CLOSE cur1;

Cursor closed.

gSQL> SELECT id, data FROM t1 ORDER BY 1;
```

```
ID DATA

-- ------

 1 data_1

 3 data_3

 5 data_5


3 rows selected.
```

兼容性

Feature ID	说明	是否支持
S111	ONLY in query expressions	X
B031	Basic dynamic SQL	O

Table 9-20 标准SQL兼容性

参考

相关内容参考下文

- **DECLARE cursor_name**
- **OPEN cursor_name**
- **FETCH cursor_name**

- **DELETE FROM**
- **DELETE FROM name RETURNING**
- **DELETE FROM name RETURNING .. INTO**

9.27 DROP AUDIT POLICY

功能

删除audit policy

语句

```
<drop audit policy statement> ::=

    DROP AUDIT POLICY [ IF EXISTS ] policy_name

;
```

使用范围及访问权限

用户需要有AUDIT SYSTEM ON DATABASE权限才能执行<drop audit policy statement>语句

语句规则及参数

IF EXISTS

无policy_name也不报错

policy_name

要删除的audit policy对象的名称

说明

无法删除已激活的audit policy对象此时应使用NOAUDIT POLICY语句禁用audit policy

使用示例

以下为删除audit policy的示例

```
DROP AUDIT POLICY policy_table;
```

兼容性

标准SQL没有audit policy

参考

相关内容参考下文

- Audit policy对象管理

    - **CREATE AUDIT POLICY**
    - **DROP AUDIT POLICY**
    - **ALTER AUDIT POLICY**
- Audit policy 激活/禁用
    - **AUDIT POLICY**
    - **NOAUDIT POLICY**
- Audit trail 查询: **AUDIT_TRAIL**
- Audit trail 清除: **ALTER DATABASE CLEAR AUDIT TRAIL**

9.28 DROP CLUSTER GROUP

功能

从集群系统删除集群组

语句

```
<drop cluster group statement> ::=

    DROP CLUSTER GROUP [IF EXISTS] group_name

    ;
```

使用范围及访问权限

可在集群系统中执行

用户需要有ADMINISTRATION ON DATABASE权限才能执行<drop cluster group statement>语句

语句规则及参数

[IF EXISTS]

没有集群组也不报错

group_name

集群组的名称

可删除没有shard的集群组

说明

即使删除集群组在不产生data loss的情况下可删除集群组

> Caution:
>
> 目标cluster group的所有成员都必须处于inactive状态
>
> 否则会发生以下错误

```
gSQL> DROP CLUSTER GROUP g3;


ERR-42000(16582): there are active cluster members in the target cluster
group 'G3'
```

使用示例

以下为删除集群组的示例

```
gSQL> DROP CLUSTER GROUP g3;
```

```
Cluster Group dropped.
```

兼容性

标准SQL未定义集群相关概念

参考

相关内容参考**CREATE CLUSTER GROUP**

9.29 DROP CLUSTER LOCATION

功能

删除集群成员的访问信息

语句

```
<drop cluster location statement> ::=
    DROP CLUSTER LOCATION member_name
    ;
```

使用范围及访问权限

可在集群系统中执行

用户需要有ADMINISTRATION ON DATABASE权限才能执行<drop cluster location statement>语句

语句规则及参数

member_name

集群成员的名称

需要有与注册的集群位置信息相同的集群成员名称

名称的长度应小于128byte

说明

默认集群位置信息通过生成集群组或添加集群成员时提供的访问信息自动生成生成的信息在删除集群成员或集群组时同时被删除

若集群位置的访问信息发生变更不需要删除集群成员并重新生成可使用**ALTER CLUSTER LOCATION**变更访问信息

使用示例

```
gSQL>
DROP CLUSTER LOCATION g1n2
;


Created
```

兼容性

标准SQL未定义集群相关概念

参考

相关内容参考下文

- **CREATE CLUSTER LOCATION**
- **ALTER CLUSTER LOCATION**

9.30 DROP INDEX

功能

删除索引

语句

```
<drop index statement> ::=

    DROP INDEX [ IF EXISTS ] index_name

    ;
```

使用范围及访问权限

用户需要有以下权限中的一个才能执行<drop index statement>语句

- 索引的所有者
- 索引所在的表的所有者
- 对索引所在的表有CONTROL TABLE ON TABLE
- 对索引所在的SCHEMA有(DROP INDEX或CONTROL SCHEMA) ON SCHEMA
- DROP ANY INDEX ON DATABASE

语句规则及参数

IF EXISTS

索引不存在时不报错

index_name

要删除的索引名

与schema_name.index_name相同可定义索引所在的SCHEMA省略schema_name时使用执行语句的用户的默认SCHEMA名

不能删除为UNIQUE约束条件PRIMARY KEY约束条件生成的索引

删除上述为了约束条件生成的索引需要通过**ALTER TABLE name DROP CONSTRAINT**语句删除相关约束条件

说明

DROP INDEX等Data Definition Language (DDL)语句在事务没有提交的情况下也可以ROLLBACK

使用示例

以下为删除索引的简单示例

```
gSQL> DROP INDEX idx_t1_id;
```

```
Index dropped.
```

以下为使用IF EXISTS语句在即使没有对应索引的情况下也不报错的示例

```
gSQL> DROP INDEX IF EXISTS not_exist_index;


Index dropped.
```

兼容性

标准SQL未定义索引的概念

参考

相关内容参考下文

- **CREATE INDEX**
- **DROP TABLE**
- **ALTER TABLE name DROP CONSTRAINT**

9.31 DROP PROFILE

功能

删除profile

语句

```
<drop profile statement> ::=

    DROP PROFILE [ IF EXISTS ] profile_name [ CASCADE ] ;
```

使用范围及访问权限

用户需要有DROP PROFILE ON DATABASE权限才能执行<drop profile statement>语句

语句规则及参数

IF EXISTS

即使没有profile也不报错

profile_name

指定要删除的profile名称

不能删除DEFAULT profile

CASCADE

如果存在已分配的用户为了删除profile则必须指定该语句

分配到要删除的profile的用户的profile将变更为DEFAULT profile

使用示例

以下为使用CASCADE语句删除profile的示例

```
gSQL> DROP PROFILE prof CASCADE;


Profile dropped.


gSQL> COMMIT;


Commit complete.
```

兼容性

标准SQL未定义profile概念

参考

相关内容参考下文

- **CREATE PROFILE**
- **ALTER PROFILE**

9.32 DROP SCHEMA

功能

删除SCHEMA

语句

```
<drop schema statement> ::=

    DROP SCHEMA [ IF EXISTS ] schema_name

        [ <drop behavior> ]

    ;


<drop behavior> ::=

      RESTRICT

    | CASCADE
```

使用范围及访问权限

用户需要有以下权限中的一个才能执行<drop schema statement>语句

- SCHEMA的所有者
- 对SCHEMA有CONTROL SCHEMA ON SCHEMA

- DROP SCHEMA ON DATABASE

语句规则及参数

IF EXISTS

没有对应SCHEMA时不报错

schema_name

要删除的SCHEMA名

但不能删除生成数据库时自动创建的DICTIONARY_SCHEMAINFORMATION_SCHEMAPUBLIC等built-in SCHEMA

<drop behavior>

- RESTRICT
  - Schema内不能有对象
- CASCADE
  - 同时删除schema内的所有对象
- 省略时默认值为RESTRICT

说明

DROP SCHEMA等Data Definition Language (DDL)语句在事务没有提交的情况下也可以ROLLBACK

此时也会删除包含在删除的schema中的垃圾桶对象

使用示例

以下为删除schema和schema内所有对象的示例

```
gSQL> DROP SCHEMA s1 CASCADE;


Schema dropped.
```

以下为使用IF EXISTS语句在即使没有对应SCHEMA的情况下也不报错的示例.

```
gSQL> DROP SCHEMA IF EXISTS not_exist_schema;


Schema dropped.
```

兼容性

标准SQL未定义IF EXISTS语句

Feature ID	说明	是否支持
F032	CASCADE drop behavior	O
F381	Extended schema manipulation	O

Table 9-21 标准SQL兼容性

参考

相关内容参考**CREATE SCHEMA**

9.33 DROP SEQUENCE

功能

删除序列

语句

```
<drop sequence generator statement> ::=

    DROP SEQUENCE [ IF EXISTS ] [schema_name.] sequence_name

    ;
```

使用范围及访问权限

用户需要有以下权限中的一个才能执行<drop sequence generator statement>语句

- 对应序列的所有者
- 对序列所在的SCHEMA有(DROP SEQUENCE或CONTROL SCHEMA) ON SCHEMA
- DROP ANY SEQUENCE ON DATABASE

语句规则及参数

IF EXISTS

没有对应序列也不报错

sequence_name

要删除的序列名

与schema_name.sequence_name相同可定义序列所在的SCHEMA省略schema_name时使用执行语句的用户的默认SCHEMA名

说明

DROP SEQUENCE等Data Definition Language (DDL)语句在事务没有提交的情况下也可以ROLLBACK

使用示例

以下为删除序列的使用示例

```
gSQL> DROP SEQUENCE seq1;


Sequence dropped.
```

以下为使用IF EXISTS语句在即使没有对应序列的情况下也不报错的示例

```
gSQL> DROP SEQUENCE invalid_sequence;


ERR-42000(16044): sequence does not exist :
DROP SEQUENCE invalid_sequence
              *
ERROR at line 1:



gSQL> DROP SEQUENCE IF EXISTS invalid_sequence;


Sequence dropped.
```

兼容性

标准SQL未定义IF EXISTS语句

Feature ID	说明	是否支持
T176	Sequence generator support	O

Table 9-22 标准SQL兼容性

参考

相关内容参考下文

- **CREATE SEQUENCE**
- **ALTER SEQUENCE**

9.34 DROP SYNONYM

功能

删除同义词

语句

```
<drop synonym statement> ::=

    DROP [ PUBLIC ] SYNONYM [ IF EXISTS ] [schema_name.]synonym_name

    ;
```

使用范围及访问权限

指定PUBLIC后删除public synonym需要有DROP PUBLIC SYNONYM ON DATABASE权限

用户需要有以下权限中的一个才能删除private synonym

- 此同义词的所有者
- 对同义词所在的SCHEMA有(DROP SYNONYM或CONTROL SCHEMA) ON SCHEMA
- DROP ANY SYNONYM ON DATABASE

语句规则及参数

[PUBLIC]

删除public synonym时指定

省略时删除private synonym

IF EXISTS

没有同义词时不报错

synonym_name

要删除的同义词名称

与schema_name.synonym_name相同可以定义同义词所属的SCHEMA省略schema_name时使用

执行语句的用户的默认SCHEMA名

指PUBLIC时无法指定SCHEMA名

说明

DROP SYNONYM等Data Definition Language (DDL)语句在事务没有提交的情况下也可以

ROLLBACK

使用示例

以下为删除private synonym的示例

```
gSQL> DROP SYNONYM MyEmp;


Synonym dropped.
```

以下为删除public synonym的示例

```
gSQL> DROP PUBLIC SYNONYM MainEmp;


Synonym dropped.
```

兼容性

标准SQL未定义DROP SYNONYM语句

参考

相关内容参考**CREATE SYNONYM**

9.35 DROP TABLE

功能

删除表

> Note:
>
> 激活垃圾桶功能时不会立即删除表而保管在垃圾桶

语句

```
<drop table statement> ::=

    DROP TABLE [ IF EXISTS ] table_name

    [ <drop behavior> ]

    [ PURGE ]

    ;


<drop behavior> ::=

      RESTRICT

    | CASCADE

    | CASCADE CONSTRAINTS
```

使用范围及访问权限

用户需要有以下权限中的一个才能执行<drop table statement>语句

- 表的所有者
- 对表有CONTROL TABLE ON TABLE
- 对表所在的SCHEMA有(DROP TABLE或CONTROL SCHEMA) ON SCHEMA
- DROP ANY TABLE ON DATABASE

语句规则及参数

IF EXISTS

没有对应表也不报错

table_name

要删除的表名

与schema_name.table_name相同可定义表所在的SCHEMA省略schema_name时使用执行语句的用户的默认SCHEMA名

不能删除如下数据库创建时自动创建的表

- "DEFINITION_SCHEMA"SCHEMA的表
- "FIXED_TABLE_SCHEMA"SCHEMA的表

同时删除在表创建的约束条件和索引

drop behavior

目前RESTRICT/CASCADE的运行方式相同

省略时默认值为RESTRICT

purge

即使激活垃圾桶功能时表也不会保存至垃圾桶而是立即删除

说明

DROP TABLE等Data Definition Language (DDL)语句在事务没有提交的情况下也可以ROLLBACK

使用示例

以下为删除普通表的示例

```
gSQL> DROP TABLE region;


Table dropped.
```

以下为使用IF EXISTS语句即使在没有对应表的情况下也不报错的示例

```
gSQL> DROP TABLE IF EXISTS invalid_table;


Table dropped.
```

以下为回滚被DROP的表的示例

```
gSQL> SELECT r_regionkey, r_name FROM region;


R_REGIONKEY R_NAME

----------- -------------------------

          0 AFRICA

          1 AMERICA

          2 ASIA

          3 EUROPE

          4 MIDDLE EAST


5 rows selected.



gSQL> DROP TABLE region;


Table dropped.



gSQL> SELECT r_regionkey, r_name FROM region;
```

```
ERR-42000(16040): table or view does not exist :
SELECT r_regionkey, r_name FROM region
                                *
ERROR at line 1:



gSQL> ROLLBACK;

Rollback complete.



gSQL> SELECT r_regionkey, r_name FROM region;

R_REGIONKEY R_NAME
----------- -------------------------
          0 AFRICA
          1 AMERICA
          2 ASIA
          3 EUROPE
          4 MIDDLE EAST

5 rows selected.
```

兼容性

标准SQL未定义以下语句

- IF EXISTS
- CASCADE CONSTRAINTS

Feature ID	说明	是否支持
F032	CASCADE drop behavior	O

Table 9-23 标准SQL兼容性

参考

相关内容参考**CREATE TABLE**

9.36 DROP TABLESPACE

功能

删除表空间

语句

```
<drop tablespace statement> ::=

    DROP TABLESPACE [ IF EXISTS ] tablespace_name

        [ INCLUDING CONTENTS ]

        [ { AND | KEEP } DATAFILES ]

        [ <drop behavior> ]

    ;


<drop behavior> ::=

      RESTRICT

    | CASCADE

    | CASCADE CONSTRAINTS
```

使用范围及访问权限

用户需要有DROP TABLESPACE ON DATABASE权限才能执行<drop tablespace definition>语句

语句规则及参数

IF EXISTS

没有对应表空间也不报错

tablespace_name

要删除的表空间的名称

不能删除如下数据库生成时自动生成的系统表空间

- DICTIONARY_TBS : system tablespace for dictionary management
- MEM_UNDO_TBS : system tablespace for default undo tablespace
- MEM_DATA_TBS : system tablespace for default user data tablespace
- MEM_TEMP_TBS : system tablespace for default temporary tablespace

> Note:
>
> tablespace_name为用户的默认表空间时删除表空间后对象不能再分配到空间因此删除表空间后需要通过**ALTER USER**语句变更用户的默认表空间

INCLUDING CONTENTS

删除表空间内的对象(table, index, key constraints)参考属于表空间的表的索引与key constraint

存在于表空间外部时也一起删除

不使用INCLUDING CONTENTS时不能有任何属于表空间的对象

[{ AND | KEEP } DATAFILES]

指定是否同时删除构成表空间的数据文件

内存临时表空间中没有数据文件因此忽略此语句

- AND DATAFILES
  - 同时删除数据文件
- KEEP DATAFILES
  - 保留数据文件不删除
- 未指定时默认为KEEP DATAFILES

drop behavior

目前RESTRICT/CASCADE的运行方式相同

省略时默认为RESTRICT

说明

DROP TABLESPACE 与其他Data Definition Language (DDL)不同不能回滚自动提交执行语句的事务此时同时删除要删除的表空间中的垃圾桶对象

使用示例

以下为删除表空间的同时删除表空间中的所有对象以及构成表空间的所有数据文件的示例

```
gSQL> DROP TABLESPACE space1 INCLUDING CONTENTS AND DATAFILES CASCADE
CONSTRAINTS;


Tablespace dropped.
```

以下为使用IF EXISTS语句在即使没有对应表空间的情况下也不报错的示例

```
gSQL> DROP TABLESPACE IF EXISTS not_exist_tablespace;


Tablespace dropped.
```

兼容性

标准SQL未定义表空间的概念

参考

相关内容参考下文

- **CREATE MEMORY DATA TABLESPACE**
- **CREATE MEMORY TEMPORARY TABLESPACE**
- **ALTER TABLESPACE**

9.37 DROP USER

功能

删除数据库用户

语句

```
<drop user statement> ::=

    DROP USER [ IF EXISTS ] user_identifier [ <drop behavior> ]

    ;


<drop behavior> ::=

      RESTRICT

    | CASCADE
```

使用范围及访问权限

用户需要有DROP USER ON DATABASE权限才能执行<drop user statement>语句

> Note:
>
> 不能有user_identifier拥有的SCHEMA

> 删除SCHEMA的详细内容参考**DROP SCHEMA**

语句规则及参数

IF EXISTS

没有对应用户也不报错

user_identifier

要删除的数据库用户的名称

但无法删除数据库生成时自动创建的"SYS"等用户

如下不删除由user_identifier生成但不是其所有者的对象

- Role
- Tablespace

<drop behavior>

- RESTRICT
  - 不能拥有User所有的如下SQL schema object
    - table, view
    - index
    - sequence

        - table constraint
- CASCADE
    - 删除User的拥有的所有如下SQL schema object
        - table, view
        - index
        - sequence
        - table constraint
- 省略时默认值为RESTRICT

> Note:
>
> DBMS中user与schema关系
>
> * Oracle
>
> ** User : schema = 1 : 1
>
> ** CASCADE时同时删除schema
>
> * DB2
>
> ** 与OS user相同
>
> ** 没有创建删除用户的额外SQL语句
>
> * Postgres
>
> ** User : schema = 1 : N
>
> ** 没有CASCADE选项删除用户拥有的所有对象以及赋予其他用户的所有权限后才能删除用户

> * MySQL
>
> ** Database : schema = 1 : 1
>
> ** user为schema(database) 的下层对象没有CASCADE选项

说明

SUNDB的user与schema是1:N的关系即用户可能没有自己的SCHEMA或也可能有多个SCHEMA

为了删除用户对象需要删除user拥有的所有schema此时也同时删除要删除的user对象的垃圾桶对象

使用示例

以下为删除用户拥有的所有schema后删除该用户的示例

```
gSQL> DROP SCHEMA u1 CASCADE;


Schema dropped.


gSQL> DROP USER u1 CASCADE;


User dropped.
```

以下为使用IF EXISTS语句在即使没有对应用户的情况下也不报错的示例

```
gSQL> DROP USER IF EXISTS not_exist_user;


User dropped.
```

兼容性

标准SQL定义了用户的概念但未定义用户的创建及删除相关的SQL语句

参考

相关内容参考下文

- **CREATE USER**
- **ALTER USER**
- **DROP SCHEMA**

9.38 DROP VIEW

功能

删除视图

语句

```
<drop view statement> ::=

    DROP VIEW [ IF EXISTS ] view_name

    ;
```

使用范围及访问权限

用户需要有以下权限中的一个才能执行<drop view statement>语句

- 视图的所有者
- 对视图有CONTROL TABLE ON TABLE
- 对视图所在的SCHEMA有(DROP VIEW或CONTROL SCHEMA) ON SCHEMA
- DROP ANY VIEW ON DATABASE

语句规则及参数

IF EXISTS

即使没有对应视图时也不报错

view_name

要删除的视图名

与schema_name.view_name相同可定义视图所在的SCHEMA省略schema_name时使用执行语句的用户的默认SCHEMA名

说明

DROP VIEW等Data Definition Language (DDL)语句在事务没有提交的情况下也可以ROLLBACK

使用示例

以下为删除视图的示例

```
gSQL> DROP VIEW v1;


View dropped.
```

以下为使用IF EXISTS语句在即使没有对应视图的情况下也不报错的示例

```
gSQL> DROP VIEW IF EXISTS not_exist_view;


View dropped.
```

兼容性

标准SQL未定义IF EXISTS语句

Feature ID	说明	是否支持
F032	CASCADE drop behavior	X

Table 9-24 标准SQL兼容性

参考

相关内容参考下文

- **CREATE VIEW**
- **ALTER VIEW**

9.39 EXECUTE IMMEDIATE 'sql_string'

功能

执行编写应用程序时未定义的动态SQL语句

语句

```
<execute immediate statement> ::=

    EXECUTE IMMEDIATE <SQL statement variable>

    ;


<SQL statement variable> ::=

      variable_name

    | 'sql statement'

    | "sql statement"

    | sql statement
```

使用范围及访问权限

可在embedded SQL中使用

需要有满足动态SQL类型的执行权限

语句规则及参数

<SQL statement variable>

<SQL statement variable>参照的动态SQL语句不能使用主机变量(:var)或parameter marker(?)

可使用如下四种类型的<SQL statement variable>

- variable_name : 存储SQL的变量
- 'sql statement' : 用单引号(')引住的SQL
- "sql statement" : 用双引号('')引住的SQL
- sql statement : 没有引号的SQL

在单引号内表示字符串数据时如下使用两次单引号

```
{
    ...
    EXEC SQL EXECUTE IMMEDIATE 'INSERT INTO t1 VALUES ( ''literal
data'' )';
    ...
}
```

SQL语句为有查询结果的query时执行成功但不返回结果

variable_name

对应variable_name的类型应为character string

variable_name中定义的动态SQL语句应为有效语句

sql statement

sql statement中定义的动态SQL语句应为有效语句

说明

EXECUTE IMMEDIATE 'sql_string'可用于dynamic embedded SQL应用程序中没有host variable的non-query SQL不需要额外的准备过程因此应用于一次性执行的DDLDML等

详细内容参考**Embedded Dynamic SQL**

使用示例

以下为在embedded SQL源代码中使用EXECUTE IMMEDIATE 'sql_string'的示例

```
{
    ...
    sprintf(sSqlStmt, "INSERT INTO EMP_RND\n"
            "SELECT *\n"
            "FROM   EMP\n"
```

```
            "WHERE  JOB = 'RND'\n" );
    EXEC SQL EXECUTE IMMEDIATE :sSqlStmt;
    if(sqlca.sqlcode != 0)
    {
        goto fail_exit;
    }
    ...
}
```

使用EXECUTE IMMEDIATE 'sql_string'的全部源代码可以在**Example Program**中查看

兼容性

Feature ID	说明	是否支持
B031	Basic Dynamic SQL	O

Table 9-25 标准SQL兼容性

参考

相关内容参考下文

- **PREPARE statement_name**
- **EXECUTE statement_name**

- **Embedded Dynamic SQL**

9.40 EXECUTE statement_name

功能

执行准备好的statement

语句

```
<execute statement> ::=
    EXECUTE statement_name [ <parameter using clause> ] [ <result into
clause> ]
    ;

<parameter using clause> ::=
      <using parameter arguments>

<using parameter arguments> ::=
    USING variable_name [, ...]

<result into clause> ::=
      <into result arguments>

<into result arguments> ::=
```

```
    INTO variable_name [, ...]
```

使用范围及访问权限

可在embedded SQL中使用

需要有符合dynamic SQL语句类型的执行权限

语句规则及参数

statement_name

准备好的statement的名称

需要通过**PREPARE statement_name**语句准备statement_name

如果statement_name参照的动态SQL语句包含dynamic parameter时应指定<parameter using clause>

```
{
    ...
    EXEC SQL PREPARE stmt1 FROM 'DELETE FROM t1 WHERE c1 > ?';
    EXEC SQL EXECUTE stmt1 USING :sValue;
    ...
}
```

```
{

    ...
    EXEC SQL PREPARE stmt1 FROM 'SELECT COUNT(*) INTO :v1 FROM t1';
    EXEC SQL EXECUTE stmt1 USING :sValue;
    ...
}
```

如果statement_name参照的动态SQL语句为query或有结果的stored function时应指定<result into clause>

```
{
    ...
    EXEC SQL PREPARE stmt1 FROM 'SELECT COUNT(*) FROM t1';
    EXEC SQL EXECUTE stmt1 INTO :sValue;
    ...
}
```

有多条查询时可正常执行但只能返回一条最初的记录

如果要获取多条记录应使用如下游标相关语句

- **DECLARE cursor_name**
- **OPEN cursor_name**
- **FETCH cursor_name**

- **CLOSE cursor_name**

没有查询结果时返回NO DATA

[<parameter using clause>] [<result into clause>]

可随意描述<parameter using clause>与<result into clause>的顺序但不能重复描述

<parameter using clause>

如果statement_name参照的动态SQL语句包含参数时应通过<using parameter arguments>指定参数信息

<using parameter arguments>

使用<using parameter arguments>语句时variable_name的数量应与statement_name参照的动态SQL语句中的参数数量相同

按照列出的variable_name的顺序对应dynamic parameter顺序

```
{

    ...
    EXEC SQL PREPARE stmt1 FROM 'DELETE FROM t1 WHERE c1 IN ( ?, ?, ? )';
    EXEC SQL EXECUTE stmt1 USING :sValue1, :sValue2, :sValue3;
    ...
```

```
}
```

<result into clause>

如果statement_name参照的动态SQL语句为query时应通过<into result arguments>指定结果Column的信息

结果为空值时不指定INDICATOR则报[DATA EXCEPTION, NULL VALUE, NO INDICATOR PARAMETER]错误

<into result arguments>

使用<into result arguments>语句时variable_name的数量应与statement_name参照的动态SQL语句的结果Column的数量相同

按照列出的variable_name的顺序对应dynamic parameter顺序

```
{

    ...
    EXEC SQL PREPARE stmt1 FROM 'SELECT MIN(salary), MAX(salary),
AVG(salary) FROM employee';
    EXEC SQL EXECUTE stmt1 INTO :sMinValue, :sMaxValue, :sAvgValue;
    ...
}
```

说明

statement_name是embedded SQL源代码中给预编译器（precompiler）的statement标识符不是host variable因此不需要额外的类型或声明EXECUTE statement_name语句应放在PREPARE statement_name语句之后

详细内容参考**Embedded Dynamic SQL**

使用示例

以下为在embedded SQL源代码中使用EXECUTE statement_name的示例

```
{
    ...
    sprintf( sUpdateSql, "UPDATE EMP SET sal = sal * :v1 WHERE JOB =
'SALES'");
    EXEC SQL PREPARE UPDATE_STMT FROM :sUpdateSql;
    if(sqlca.sqlcode != 0)
    {
        goto fail_exit;
    }

    sRatio = 1.1;
    EXEC SQL EXECUTE UPDATE_STMT USING :sRatio;
    if(sqlca.sqlcode != 0)
```

```
    {
        goto fail_exit;
    }
    ...
}
```

使用EXECUTE statement_name的全部源代码可以在**Example Program**中查看

兼容性

Feature ID	说明	是否支持
B031	Basic Dynamic SQL	O
B032	Extended dynamic SQL	X

Table 9-26 标准SQL兼容性

参考

相关内容参考下文

- **PREPARE statement_name**
- **DECLARE cursor_name**
- **OPEN cursor_name**

- **FETCH cursor_name**
- **CLOSE cursor_name**
- **Embedded Dynamic SQL**

9.41 FETCH cursor_name

功能

将游标位于结果集的特定行并通过主机变量获取对应行的值

语句

```
<fetch statement> ::=
    FETCH [ <fetch orientation> ] [ FROM ] cursor_name
        <result into clause>
    ;

<fetch orientation> ::=
      NEXT
    | PRIOR
    | FIRST
    | LAST
    | CURRENT
    | ABSOLUTE position
    | RELATIVE position

<result into clause> ::=
```

```
      <into result arguments>


<into result arguments> ::=

    INTO variable_name [, ...]
```

语句规则及参数

[FROM] cursor_name

应为在会话内open的游标

可以省略FROM

<fetch orientation>

为了使用FETCH NEXT外的<fetch orientation>应使用scrollable cursor

省略<fetch orientation>时默认为NEXT

Open的游标对结果集有如下游标位置信息

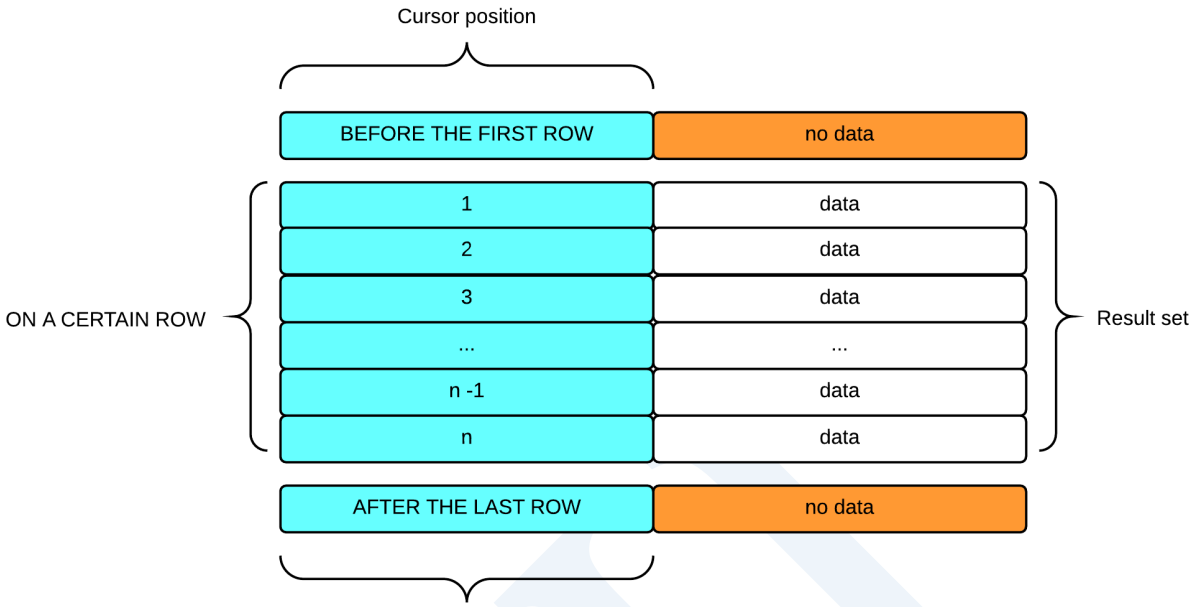


Figure 9-1 游标的位置信息

游标的位置	说明
BEFORE THE FIRST ROW	结果集的第一行之前的位置OPEN时间点的位置也属于此
ON A CERTAIN ROW	通过FETCH位于结果集的特定行的状态
AFTER THE LAST ROW	位于结果集的最后一行之后的状态

Table 9-27 游标的位置

以游标的当前位置为准<fetch orientation>如下运行

- NEXT : 检索当前位置的下一行
- PRIOR : 检索当前位置的前一行
- FIRST : 检索结果集的第一行
- LAST : 检索结果集的最后一行

- CURRENT : 检索当前位置的row
- ABSOLUTE position
  - 检索结果集的position位置对应的row
  - position为负数时检索从AFTER THE LAST ROW之前位置对应的row
- RELATIVE position
  - 检索离当前位置相隔position距离的row

<result into clause>

使用<into result arguments>描述获取结果Column的变量信息

结果值为空值时不指定INDICATOR则报[DATA EXCEPTION, NULL VALUE, NO INDICATOR PARAMETER]错误

<into result arguments>

INTO语句的变量数量应与游标的结果集的Column数量相同

说明

执行FETCH后的游标位置为BEFORE THE FIRST ROW或AFTER THE LAST LOW时与<fetch orientation>中输入的位置无关位于相同位置

使用示例

以下为在interactive sql(gsql)中声明SCROLL游标并显示多种<fetch orientation>运行的示例

```
gSQL> DECLARE cur_scroll SCROLL CURSOR FOR SELECT id, data FROM t1;


Cursor declared.


gSQL> OPEN cur_scroll;


Cursor is open.


gSQL> \var v_id   INTEGER
gSQL> \var v_data VARCHAR(128)


gSQL> FETCH NEXT cur_scroll INTO :v_id, :v_data;


V_ID V_DATA
---- ------
   1 data_1


1 row fetched.


gSQL> FETCH NEXT cur_scroll INTO :v_id, :v_data;
```

```
V_ID V_DATA

---- ------

   2 data_2


1 row fetched.


gSQL> FETCH PRIOR cur_scroll INTO :v_id, :v_data;


V_ID V_DATA

---- ------

   1 data_1


1 row fetched.


gSQL> FETCH PRIOR cur_scroll INTO :v_id, :v_data;


no rows fetched.


gSQL> FETCH FIRST cur_scroll INTO :v_id, :v_data;


V_ID V_DATA

---- ------

   1 data_1


1 row fetched.
```

```
gSQL> FETCH FIRST cur_scroll INTO :v_id, :v_data;

V_ID V_DATA
---- ------
   1 data_1

1 row fetched.

gSQL> FETCH LAST cur_scroll INTO :v_id, :v_data;

V_ID V_DATA
---- ------
   5 data_5

1 row fetched.

gSQL> FETCH LAST cur_scroll INTO :v_id, :v_data;

V_ID V_DATA
---- ------
   5 data_5

1 row fetched.
```

```
gSQL> FETCH FIRST cur_scroll INTO :v_id, :v_data;


V_ID V_DATA

---- ------

   1 data_1


1 row fetched.


gSQL> FETCH CURRENT cur_scroll INTO :v_id, :v_data;


V_ID V_DATA

---- ------

   1 data_1


1 row fetched.


gSQL> FETCH LAST cur_scroll INTO :v_id, :v_data;


V_ID V_DATA

---- ------

   5 data_5


1 row fetched.


gSQL> FETCH CURRENT cur_scroll INTO :v_id, :v_data;
```

```
V_ID V_DATA
---- ------
   5 data_5

1 row fetched.

gSQL> FETCH ABSOLUTE 3 cur_scroll INTO :v_id, :v_data;

V_ID V_DATA
---- ------
   3 data_3

1 row fetched.

gSQL> FETCH CURRENT cur_scroll INTO :v_id, :v_data;

V_ID V_DATA
---- ------
   3 data_3

1 row fetched.

gSQL> FETCH ABSOLUTE 1 cur_scroll INTO :v_id, :v_data;
```

```
V_ID V_DATA

---- ------

   1 data_1


1 row fetched.


gSQL> FETCH ABSOLUTE -1 cur_scroll INTO :v_id, :v_data;


V_ID V_DATA

---- ------

   5 data_5


1 row fetched.


gSQL> FETCH ABSOLUTE 6 cur_scroll INTO :v_id, :v_data;


no rows fetched.


gSQL> FETCH ABSOLUTE -6 cur_scroll INTO :v_id, :v_data;


no rows fetched.


gSQL> FETCH ABSOLUTE 3 cur_scroll INTO :v_id, :v_data;


V_ID V_DATA
```

```
---- ------
   3 data_3


1 row fetched.


gSQL> FETCH ABSOLUTE -3 cur_scroll INTO :v_id, :v_data;


V_ID V_DATA
---- ------
   3 data_3


1 row fetched.


gSQL> FETCH RELATIVE 1 cur_scroll INTO :v_id, :v_data;


V_ID V_DATA
---- ------
   4 data_4


1 row fetched.


gSQL> FETCH RELATIVE -1 cur_scroll INTO :v_id, :v_data;


V_ID V_DATA
---- ------
```

```
   3 data_3


1 row fetched.


gSQL> FETCH RELATIVE 5 cur_scroll INTO :v_id, :v_data;


no rows fetched.


gSQL> FETCH RELATIVE -5 cur_scroll INTO :v_id, :v_data;


V_ID V_DATA
---- ------
   1 data_1


1 row fetched.


gSQL> CLOSE cur_scroll;


Cursor closed.
```

兼容性

标准SQL未定义<fetch orientation>中的CURRENT内容

Feature ID	说明	是否支持
F431	Read-only scrollable cursors	O
B031	Basic dynamic SQL	O

Table 9-28 标准SQL兼容性

参考

相关内容参考下文

- **DECLARE cursor_name**
- **OPEN cursor_name**
- **CLOSE cursor_name**

9.42 FLASHBACK TABLE

功能

恢复保管在回收站的表对象

语句

```
<flashback table statement> ::=

    FLASHBACK TABLE table_name

    TO BEFORE DROP [ RENAME TO new_table_name ]

    ;
```

使用范围及访问权限

用户需要有一下权限中的一个才能执行<flashback table statement>语句

- 相应表的所有者
- 对相应表有CONTROL TABLE ON TABLE
- 对表所在的SCHEMA有(DROP TABLE或CONTROL SCHEMA) ON SCHEMA
- DROP ANY TABLE ON DATABASE

语句规则及参数

table_name

回收站中保存的对象的名称或删除的表的名称

删除的表名称与 schema_name.table_namei相同可定义schema省略时使用执行语句的用户的默认schema名称

new_table_name

恢复的表的新名称

Schema中不能有相同的表名称

说明

使用回收站中保存的对象名或删除的表名恢复保管在回收站的表对象如果有与删除的表相同的名称时恢复最新的表对象

要恢复的表对象的名称已存在时报错可使用RENAME TO语句恢复为新的表名恢复的表的约束条件和索引恢复为被删除之前的名称如果有与被删除之前的约束条件和索引名称相同的名称时恢复为回收站中保存的名称

与其他Data Definition Language (DDL)不同无法回滚FLASHBACK TABLE自动提交执行语句的事务

使用示例

以下为以回收站中保存的对象名恢复表的示例

```
gSQL> SELECT SCHEMA_NAME, OBJECT_NAME, ORIGINAL_NAME, OBJECT_TYPE FROM
USER_RECYCLEBIN;

SCHEMA_NAME OBJECT_NAME                          ORIGINAL_NAME OBJECT_TYPE
----------- ------------------------------------ ------------- -----------
PUBLIC      BIN$106A4F90165D11EA9C5C835D3E4BBBF7 T1            TABLE

1 row selected.

gSQL> FLASHBACK TABLE "BIN$106A4F90165D11EA9C5C835D3E4BBBF7" TO BEFORE
DROP;

Flashback complete.
```

以下为在回收站恢复为被删除之前的表名的示例

```
gSQL> SELECT SCHEMA_NAME, OBJECT_NAME, ORIGINAL_NAME, OBJECT_TYPE FROM
USER_RECYCLEBIN;

SCHEMA_NAME OBJECT_NAME                          ORIGINAL_NAME OBJECT_TYPE
----------- ------------------------------------ ------------- -----------
```

```
PUBLIC      BIN$106A4F90165D11EA9C5C835D3E4BBBF7 T1            TABLE


gSQL> FLASHBACK TABLE T1 TO BEFORE DROP;


Flashback complete.
```

兼容性

标准SQL中未定义<flashback table statement>

参考

相关内容参考如下

- 表回收站管理
- PURGE

9.43 GRANT privileges TO

功能

给用户赋权限

语句

```
<grant privilege statement> ::=
    GRANT <privilege> TO <grantee> [, ...]
        [ WITH GRANT OPTION ]
    ;

<grantee> ::=
      PUBLIC
    | user_identifier
    ;

<privilege> ::=
      <database privilege>
    | <tablespace privilege>
    | <schema privilege>
    | <table privilege>
```

```
    | <sequence privilege>
    | <procedure privilege>


<database privilege> ::=
      ALL [ PRIVILEGES ] [ON DATABASE]
    | <database action> [, ...] [ON DATABASE]


<database action> ::=
      ADMINISTRATION
    | ANALYZE ANY
    | ALTER DATABASE
    | ALTER SYSTEM
    | AUDIT SYSTEM
    | ACCESS CONTROL
    | CREATE SESSION
    | CREATE PROFILE
    | ALTER PROFILE
    | DROP PROFILE
    | CREATE USER
    | ALTER USER
    | DROP USER
    | CREATE ROLE
    | ALTER ROLE
    | DROP ROLE
    | CREATE TABLESPACE
```

```
| ALTER TABLESPACE
| DROP TABLESPACE
| USAGE TABLESPACE
| CREATE SCHEMA
| ALTER SCHEMA
| DROP SCHEMA
| CREATE PUBLIC SYNONYM
| DROP PUBLIC SYNONYM
| CREATE ANY TABLE
| ALTER ANY TABLE
| DROP ANY TABLE
| SELECT ANY TABLE
| INSERT ANY TABLE
| DELETE ANY TABLE
| UPDATE ANY TABLE
| LOCK ANY TABLE
| CREATE ANY VIEW
| DROP ANY VIEW
| CREATE ANY SEQUENCE
| ALTER ANY SEQUENCE
| DROP ANY SEQUENCE
| USAGE ANY SEQUENCE
| CREATE ANY INDEX
| ALTER ANY INDEX
| DROP ANY INDEX
```

```
    | CREATE ANY SYNONYM

    | DROP ANY SYNONYM

    | CREATE ANY PROCEDURE

    | ALTER ANY PROCEDURE

    | DROP ANY PROCEDURE

    | EXECUTE ANY PROCEDURE

    | CREATE ANY PACKAGE

    | ALTER ANY PACKAGE

    | DROP ANY PACKAGE

    | EXECUTE ANY PACKAGE

    | PURGE DBA_RECYCLEBIN


<tablespace privilege> ::=

      ALL [ PRIVILEGES ] ON TABLESPACE tablespace_name

    | <tablespace action> [, ...] ON TABLESPACE tablespace_name


<tablespace action> ::=

    CREATE OBJECT


<schema privilege> ::=

      ALL [ PRIVILEGES ] ON SCHEMA schema_name

    | <schema action> [, ...] [ON SCHEMA schema_name]


<schema action> ::=

      CONTROL SCHEMA
```

```
| CREATE TABLE
| ALTER TABLE
| DROP TABLE
| SELECT TABLE
| INSERT TABLE
| DELETE TABLE
| UPDATE TABLE
| LOCK TABLE
| CREATE VIEW
| DROP VIEW
| CREATE SEQUENCE
| ALTER SEQUENCE
| DROP SEQUENCE
| USAGE SEQUENCE
| CREATE INDEX
| ALTER INDEX
| DROP INDEX
| ADD CONSTRAINT
| CREATE SYNONYM
| DROP SYNONYM
| CREATE PROCEDURE
| ALTER PROCEDURE
| DROP PROCEDURE
| EXECUTE PROCEDURE
| CREATE PACKAGE
```

```
    | ALTER PACKAGE

    | DROP PACKAGE

    | EXECUTE PACKAGE


<table privilege> ::=

      ALL [ PRIVILEGES ] ON [TABLE] table_name

    | { <table action> | <column action> } [, ...] ON [TABLE] table_name


<table action> ::=

      CONTROL TABLE

    | SELECT

    | INSERT

    | UPDATE

    | DELETE

    | REFERENCES

    | LOCK

    | INDEX

    | ALTER


<column action> ::=

      SELECT ( column_name [, ...] )

    | INSERT ( column_name [, ...] )

    | UPDATE ( column_name [, ...] )

    | REFERENCES ( column_name [, ...] )
```

```
<sequence privilege> ::=

      ALL [ PRIVILEGES ] ON SEQUENCE sequence_name

    | <sequence action> ON SEQUENCE sequence_name


<sequence action> ::=

    USAGE


<procedure privilege> ::=

      ALL [ PRIVILEGES ] ON PROCEDURE procedure_name

    | <procedure action> ON PROCEDURE procedure_name


<procedure action> ::=

    EXECUTE


<package privilege> ::=

      ALL [ PRIVILEGES ] ON PACKAGE package_name

    | <package action> ON PACKAGE package_name


<package action> ::=

    EXECUTE
```

语句规则及参数

<grantee>

被赋予权限的用户

- user_identifier
  - 向对应用户赋予权限
- PUBLIC
  - 指所有用户的authorization对象

WITH GRANT OPTION

使grantee（被赋予权限的用户）可以向其他用户赋予对应权限

如下赋予相同的<privilege>的权限时会继续维持WITH GRANT OPTION

- GRANT SELECT ON t1 TO u1 WITH GRANT OPTION;
- GRANT SELECT ON t1 TO u1;

<privilege>

要向grantee（被赋予权限的用户）赋予的权限

Grantor（执行语句的用户）需要满足以下条件中的一个

- Grantor使用WITH GRANT OPTION拥有该<privilege>

    - Grantor成为执行语句的用户
- Grantor拥有ACCESS CONTROL ON DATABASE权限
    - Grantor成为对象的所有者
        - <database privilege> : _SYSTEM账号
        - <tablespace privilege> : _SYSTEM账号
        - <schema privilege> : _SYSTEM账号
        - <table privilege> : table的所有者
        - <sequence privilege> : sequence的所有者
        - <procedure privilege> : procedure/function的所有者
        - <package privilege>: package的所有者

<database privilege>

对数据库对象的权限

可以省略[ON DATABASE]语句

可用database privilege定义的database action如下

- ALL [PRIVILEGES] [ON DATABASE]
    - grantor（执行语句的用户）使用WITH GRANT OPTION拥有的数据库的所有权限

<database action>	说明
ADMINISTRATION	启动关闭服务器的权限
ALTER DATABASE	执行ALTER DATABASE语句的权限

<database action>	说明
ALTER SYSTEM	执行ALTER SYSTEM语句的权限
AUDIT SYSTEM	控制Audit Policy的权限
ACCESS CONTROL	控制所有权限的权限
CREATE SESSION	连接数据库的权限
CREATE PROFILE	在数据库中生成profile的权限
ALTER PROFILE	变更数据库的所有profile的权限
DROP PROFILE	删除数据库的所有profile的权限
CREATE USER	在数据库中生成用户的权限
ALTER USER	变更数据库的所有用户的权限
DROP USER	删除数据库的所有用户的权限
CREATE ROLE	在数据库中生成角色的权限
ALTER ROLE	变更数据库的所有角色的权限
DROP ROLE	删除数据库的所有角色的权限
CREATE TABLESPACE	在数据库中生成表空间的权限
ALTER TABLESPACE	在数据库中变更所有表空间的权限
DROP TABLESPACE	在数据库中删除所有表空间的权限
USAGE TABLESPACE	在数据库中使用所有表空间的权限
CREATE SCHEMA	在数据库中生成SCHEMA的权限

<database action>	说明
ALTER SCHEMA	变更数据库的所有SCHEMA的权限
DROP SCHEMA	删除数据库的所有SCHEMA的权限
CREATE PUBLIC SYNONYM	在数据库中生成PUBLIC SYNONYM的权限
DROP PUBLIC SYNONYM	删除数据库的所有PUBLIC SYNONYM的权限
CREATE ANY TABLE	在数据库的所有SCHEMA中生成表的权限
ALTER ANY TABLE	变更数据库的所有表的权限
DROP ANY TABLE	删除数据库的所有表的权限
SELECT ANY TABLE	检索数据库的所有表的行数据的权限
INSERT ANY TABLE	在数据库的所有表中插入行数据的权限
DELETE ANY TABLE	在数据库的所有表中删除行数据的权限
UPDATE ANY TABLE	在数据库的所有表中更新行数据的权限
LOCK ANY TABLE	对数据库的所有表执行LOCK语句的权限
CREATE ANY VIEW	在数据库的所有SCHEMA里生成视图的权限
DROP ANY VIEW	删除数据库的所有视图的权限
CREATE ANY SEQUENCE	在数据库的所有SCHEMA里生成序列的权限
ALTER ANY SEQUENCE	变更数据库的所有序列的权限
DROP ANY SEQUENCE	删除数据库的所有序列的权限
USAGE ANY SEQUENCE	使用数据库的所有序列的权限

<database action>	说明
CREATE ANY INDEX	在数据库的所有SCHEMA里生成索引的权限
ALTER ANY INDEX	变更数据库的所有索引的权限
DROP ANY INDEX	删除数据库的所有索引的权限
CREATE ANY SYNONYM	生成数据库的所有同义词的权限
DROP ANY SYNONYM	删除数据库的所有同义词的权限
CREATE ANY PROCEDURE	在数据库的所有schema中生成procedure/function的权限
ALTER ANY PROCEDURE	变更数据库的所有procedure/function的权限
DROP ANY PROCEDURE	删除数据库的所有procedure/function的权限
EXECUTE ANY PROCEDURE	执行数据库的所有procedure/function的权限
CREATE ANY PACKAGE	在数据库的所有schema中生成package的权限
ALTER ANY PACKAGE	变更数据库的所有package的权限
DROP ANY PACKAGE	删除数据库的所有package的权限
EXECUTE ANY PACKAGE	执行数据库的所有package的权限
PURGE DBA_RECYCLEBIN	可删除数据库的所有回收站的权限

Table 9-29 Database Privilege

<tablespace privilege>

对表空间对象的权限

可用tablespace privilege定义的tablespace action如下

- ALL [PRIVILEGES] ON TABLESPACE tablespace_name
    - grantor（执行语句的用户）使用WITH GRANT OPTION拥有的该表空的所有权限

<tablespace action>	说明
CREATE OBJECT	在表空间中生成对象的权限

Table 9-30 Tablespace Privilege

<schema privilege>

对SCHEMA对象的权限

- 是否省略[ON SCHEMA schema_name]
    - 有多个<grantee>时不能省略[ON SCHEMA schema_name]语句
    - 使用ALL [PRIVILEGES]时不能省略[ON SCHEMA schema_name]语句
    - 省略[ON SCHEMA schema_name]时<grantee>只能使用一个user_identifier在<grantee>的schema检索路径中赋予第一个SCHEMA的权限

以下为可用schema privilege定义的schema action

- ALL [PRIVILEGES] ON SCHEMA schema_name
    - grantor（执行语句的用户）使用WITH GRANT OPTION拥有的该schema的所有权限

<schema action>	说明
CONTROL SCHEMA	对对应SCHEMA的所有权限
CREATE TABLE	在SCHEMA中生成表的权限
ALTER TABLE	变更SCHEMA的所有表的权限
DROP TABLE	删除SCHEMA的所有表的权限
SELECT TABLE	查询SCHEMA的所有表的行数据的权限
INSERT TABLE	生成SCHEMA的所有表的行数据的权限
DELETE TABLE	删除SCHEMA的所有表的行数据的权限
UPDATE TABLE	更新SCHEMA的所有表的行数据的权限
LOCK TABLE	对SCHEMA的所有表执行LOCK语句的权限
CREATE VIEW	在SCHEMA中生成视图的权限
DROP VIEW	删除SCHEMA的所有视图的权限
CREATE SEQUENCE	在SCHEMA中生成序列的权限
ALTER SEQUENCE	变更SCHEMA的所有序列的权限
DROP SEQUENCE	删除SCHEMA的所有序列的权限
USAGE SEQUENCE	使用SCHEMA的所有序列的权限
CREATE INDEX	在SCHEMA中生成索引的权限
ALTER INDEX	变更SCHEMA的所有索引的权限
DROP INDEX	删除SCHEMA的所有索引的权限

<schema action>	说明
ADD CONSTRAINT	在SCHEMA中生成约束条件的权限
CREATE SYNONYM	在SCHEMA中生成同义词的权限
DROP SYNONYM	删除SCHEMA的所有同义词的权限
CREATE PROCEDURE	在SCHEMA中生成procedure/function的权限
ALTER PROCEDURE	变更SCHEMA的所有procedure/function的权限
DROP PROCEDURE	删除SCHEMA的所有procedure/function的权限
EXECUTE PROCEDURE	执行SCHEMA的所有procedure/function的权限
CREATE PACKAGE	在SCHEMA中生成package的权限
ALTER PACKAGE	变更SCHEMA的所有package的权限
DROP PACKAGE	删除SCHEMA的所有package的权限
EXECUTE PACKAGE	执行SCHEMA的所有package的权限

Table 9-31 Schema privilege

<table privilege>

对表或视图对象的权限

可以省略[TABLE]语句

可用table privilege定义的table action如下

- ALL [PRIVILEGES] ON [TABLE] table_name
    - grantor（执行语句的用户）使用WITH GRANT OPTION拥有的对应表的所有权限

<table action>	说明
CONTROL TABLE	对对应表的全部权限
SELECT	查询表的行数据的权限
INSERT	生成表的行数据的权限
UPDATE	更新表的行数据的权限
DELETE	删除表的行数据的权限
REFERENCES	生成参考对应表的参考约束条件的权限
LOCK	对表执行LOCK语句的权限
INDEX	对表生成索引的权限
ALTER	变更表的权限

Table 9-32 Table privilege

SELECTINSERTUPDATEREFERENCES对表的所有column赋予额外的权限

可用table privilege定义的column action如下但column action仅应用于base table

<column action>	说明
SELECT (columns)	查询对应column的权限

<column action>	说明
INSERT (columns)	生成包含对应column的行数据的权限
UPDATE (columns)	更新对应column的权限
REFERENCES (columns)	生成参考对应column的参考约束条件的权限

Table 9-33 Column privilege

<sequence privilege>

对序列对象的权限

可用sequence privilege定义的sequence action如下

- ALL [PRIVILEGES] ON SEQUENCE sequence_name
    - grantor（执行语句的用户）使用WITH GRANT OPTION拥有的对应序列的所有权限

<sequence action>	说明
USAGE	使用序列的权限

Table 9-34 Sequence privilege

<procedure privilege>

对procedure/ function对象的权限

可用procedure privilege定义的action如下

- ALL [PRIVILEGES] ON PROCEDURE procedure_name
    - grantor（执行语句的用户）使用WITH GRANT OPTION拥有的对应procedure/ function的所有权限

<sequence action>	说明
EXECUTE	执行procedure/ function的权限

Table 9-35 Procedure privilege

<package privilege>

对package对象的权限

可用package privilege定义的action如下

- ALL [PRIVILEGES] ON PACKAGE package_name
    - grantor（执行语句的用户）使用WITH GRANT OPTION拥有的对应package的所有权限

<package action>	说明
EXECUTE	执行package的权限

Table 9-36 Package privilege

说明

GRANT privilege等Data Definition Language (DDL)语句在事务没有提交的情况下也可以ROLLBACK

创建TableSequence等SQL schema object的owner即使未被赋予额外的权限其本身拥有一定的权限

详细说明参考如下CREATE语句

- **CREATE TABLE**
- **CREATE VIEW**
- **CREATE SEQUENCE**
- **ALTER TABLE name ADD COLUMN**
- **CREATE FUNCTION**
- **CREATE PROCEDURE**
- **CREATE PACKAGE**

创建schema, tablespace等non-schema object的owner不会自动被赋予对应对象的任何权限因此需要另赋权限

详细说明参考如下CREATE语句

- **CREATE SCHEMA**
- **CREATE TABLESPACE**
- **CREATE USER**

使用示例

以下为向u1用户赋予SELECT ON TABLE t1权限的示例

```
gSQL> GRANT SELECT ON t1 TO u1;


Grant succeeded.
```

以下为向指所有用户的PUBLIC账号赋予SELECT ON TABLE t1权限的示例

```
gSQL> GRANT SELECT ON t1 TO PUBLIC;


Grant succeeded.
```

以下为u1用户使用WITH GRANT OPTION向其他用户赋予对应权限的示例

```
gSQL> GRANT SELECT ON t1 TO u1 WITH GRANT OPTION;


Grant succeeded.
```

以下为执行语句的用户使用WITH GRANT OPTION向u1用户赋予TABLE t1对象的所有权限的示例

```
gSQL> GRANT ALL PRIVILEGES ON TABLE t1 TO u1;


Grant succeeded.
```

以下为赋予可连接数据库的CREATE SESSION ON DATABASE权限的示例

```
gSQL> GRANT CREATE SESSION ON DATABASE TO u1;


Grant succeeded.
```

以下为向u1用户赋予可在SCHEMA s1中创建tableviewindexsequenceconstraint对象的多个权限的示例

```
gSQL> GRANT CREATE TABLE, CREATE VIEW, CREATE INDEX, CREATE SEQUENCE, ADD
CONSTRAINT ON SCHEMA s1 TO u1;


Grant succeeded.
```

以下为向u1用户赋予可在TABLESPACE mem_data_tbs中创建对象的权限的示例

```
gSQL> GRANT CREATE OBJECT ON TABLESPACE mem_data_tbs TO u1;


Grant succeeded.
```

以下为向u1用户赋予可查询t1表的部分Column的权限的示例

```
gSQL> GRANT SELECT( id, name ) ON TABLE t1 TO u1;


Grant succeeded.
```

以下为向u1用户赋予可对seq1序列使用NEXTVAL()CURRVAL()函数的权限的示例

```
gSQL> GRANT USAGE ON SEQUENCE seq1 TO u1;


Grant succeeded.
```

兼容性

标准SQL未定义以下权限

- <database privilege>
- <tablespace privilege>
- <schema privilege>

Feature ID	说明	是否支持
S023	Basic structured types	X
S024	Enhanced structured types	X
S081	Subtables	X
T211	Basic trigger capability	X
T281	SELECT privilege with column granularity	O
T332	Extended Roles	X
F731	INSERT column privileges	O

Table 9-37 标准SQL兼容性

参考

相关内容参考下文

- **REVOKE privileges FROM**
- **CREATE USER**
- **DROP USER**
- **ALTER USER**

10. SQL References (H~Z)

10.1 INSERT INTO

功能

在表中创建新的行数据

语句

```
<insert statement> ::=
    INSERT INTO table_name [ ( column_name [, ...] ) ]
        <insert source>
    ;

<insert source> ::=
      <values clause>
    | <from subquery>
    | <from default>

<values clause> ::=
    VALUES { ( { <value expression> | DEFAULT } [, ...] ) } [, ...]
```

```
<from subquery> ::=
    <query expression>

<from default> ::=
    DEFAULT VALUES
```

使用范围及访问权限

用户应满足以下条件才能执行<insert statement>语句

- 用户需要有以下权限中的一个才能执行INSERT语句
  - 对作为插入对象的所有Column有INSERT(columns) ON TABLE
  - 对表有(INSERT或CONTROL TABLE) ON TABLE
  - 对表所在的SCHEMA有(INSERT TABLE或CONTROL SCHEMA) ON SCHEMA
  - INSERT ANY TABLE ON DATABASE
- 使用<from subquery>时对使用的所有表需要有以下权限中的一个
  - 对表的Column中用于语句的所有column有SELECT(columns) ON TABLE
  - 对表有(SELECT或CONTROL TABLE) ON TABLE
  - 对表所在的SCHEMA有(SELECT TABLE或CONTROL SCHEMA) ON SCHEMA
  - SELECT ANY TABLE ON DATABASE

语句规则及参数

table_name

创建行数据的对象表的名称

与schema_name.table_name相同可定义表所在的SCHEMA省略schema_name时使用执行语句的用户的默认SCHEMA名

[(column_name [, ...])]

表的column名

可以省略column列表

Column数量应与<insert source>值的数量相同向省略的column分配默认值

<values clause>

要向对应column分配的值的列表

- <value expression>
  - 向对应的column的分配的值或运算式
- DEFAULT
  - 对应column的值使用通过**CREATE TABLE**语句定义的默认值
  - 未定义时分配空值

如下可以创建多条row

```
INSERT INTO table_name VALUES ( 1, 'A' ), ( 2, 'B' ), ( 3, 'C' )
```

<from subquery>

要创建行数据的查询

详细内容参考**SELECT**的**query expression**

DEFAULT VALUES

所有Column都使用默认值

DEFAULT VALUES与以下语句意义相同

```
VALUES ( DEFAULT, DEFAULT, ..., DEFAULT )
```

说明

INSERT相关语句之间的区别

- **INSERT INTO**
  - 在表创建一条或多条Row
  - 例: INSERT INTO t1 SELECT * FROM t1;
- **INSERT INTO name RETURNING**
  - 在表创建一条或多条Row创建的行可以与SELECT语句相同的方式(SQLFetch()等API)进行检索

- 例: INSERT INTO t1 SELECT * FROM t1 RETURNING c1;

- **INSERT INTO name RETURNING .. INTO**
  - 可创建一条以下的row仅创建一条时通过RETURNING INTO的主机变量获取值
  - 例: INSERT INTO t1 DEFAULT VALUES RETURNING c1 INTO :v1;

使用示例

以下为使用INSERT语句创建一条row的示例

```
gSQL> INSERT INTO region VALUES ( 0, 'AFRICA' );


1 row created.
```

以下为在INSERT语句中使用column的默认值或identity值的示例

```
gSQL> CREATE TABLE region
(
    r_regionkey    BIGINT    GENERATED BY DEFAULT AS IDENTITY
  , r_name         CHAR(25)  DEFAULT 'N/A'
);


Table created.


gSQL> COMMIT;
```

```
Commit complete.
```

- 所有column输入DEFAULT

```
gSQL> INSERT INTO region DEFAULT VALUES;


1 row created.
```

- 所有column输入DEFAULT

```
gSQL> INSERT INTO region VALUES (DEFAULT, DEFAULT);


1 row created.
```

- 省略column时使用r_name column的DEFAULT值

```
gSQL> INSERT INTO region(r_regionkey) VALUES (-100);


1 row created.
```

- 省略column时使用r_regionkey column的identity值

```
gSQL> INSERT INTO region(r_name) VALUES ('ASIA');


1 row created.
```

```
gSQL> SELECT * FROM region;


R_REGIONKEY R_NAME

----------- -------------------------

          1 N/A

          2 N/A

       -100 N/A

          3 ASIA


4 rows selected.
```

以下为在VALUES语句描述并创建多条row的示例

```
gSQL> INSERT INTO region

       VALUES ( 1, 'AFRICA' ),

              ( 2, 'ASIA'   ),

              ( 3, 'EUROPE' );


3 rows created.
```

以下为使用子查询创建多条row的示例

```
gSQL> INSERT INTO region SELECT r_regionkey, r_name FROM tmp_region WHERE

r_regionkey < 3;


3 rows created.
```

兼容性

Feature ID	说明	是否支持
F781	Self-referencing operations	X
F222	INSERT statement: DEFAULT VALUES clause	O
S204	Enhanced structured types	X
S043	Enhanced reference types	X
T111	Updatable joins, unions, and columns	X

Table 10-1 标准SQL兼容性

参考

相关内容参考下文

- **SELECT**
- **INSERT INTO name RETURNING**
- **INSERT INTO name RETURNING .. INTO**

10.2 INSERT INTO name RETURNING

功能

在表中创建新的行数据并查询创建的行数据

语句

```
<insert statement> ::=
    INSERT INTO table_name [ ( column_name [, ...] ) ]
        <insert source>
        <returning clause>
    ;

<insert source> ::=
      <values clause>
    | <from subquery>
    | <from default>

<values clause> ::=
    VALUES { ( { <value expression> | DEFAULT } [, ...] ) } [, ...]

<from subquery> ::=
```

```
    <query expression>


<from default> ::=
    DEFAULT VALUES


<returning clause> ::=
      [ RETURN | RETURNING ] { * | { <value expression> [ [AS]
alias_name ] } [, ...]
```

使用范围及访问权限

用户应满足以下条件才能执行<insert returning query statement>语句

- 用户需要有以下权限中的一个才能执行INSERT语句
  - 对作为插入对象的所有column有INSERT(columns) ON TABLE
  - 对表有(INSERT或CONTROL TABLE) ON TABLE
  - 对表所在的SCHEMA有(INSERT TABLE或CONTROL SCHEMA) ON SCHEMA
  - INSERT ANY TABLE ON DATABASE
- 使用<from subquery>时对语句中使用的所有表需要有以下权限中的一个
  - 对表的column中用于语句的所有column有SELECT(columns) ON TABLE
  - 对表有(SELECT或CONTROL TABLE) ON TABLE
  - 对表所在的SCHEMA有(SELECT TABLE或CONTROL SCHEMA) ON SCHEMA
  - SELECT ANY TABLE ON DATABASE
- 对用于RETURNING的所有column需要有以下权限中的一个
  - 对用于RETURNING语句的所有Column有SELECT(columns) ON TABLE

- 对表有(SELECT或CONTROL TABLE) ON TABLE
- 对表所在的SCHEMA有(SELECT TABLE或CONTROL SCHEMA) ON SCHEMA
- SELECT ANY TABLE ON DATABASE

语句规则及参数

table_name

创建行数据的对象表名

[(column_name [, ...])]

表的Column名

详细内容参考**INSERT INTO**

<values clause>

要向对应Column分配的值的列表

详细内容参考**INSERT INTO**

<from subquery>

创建row的查询

详细内容参考**INSERT INTO**

DEFAULT VALUES

所有Column使用默认值

详细内容参考**INSERT INTO**

<returning clause>

返回插入的行数据

- 结果集为创建的行数据描述其中要查询的Column
  - RETURNING返回用INSERT语句插入的row作为结果集的结果
  - <value expression>
    - 与SELECT语句的<select list>相同但不能使用Aggregation
  - [[AS] alias_name]
    - 使用AS可指定value expression的名称

RETURN与RETURNING为相同意义的关键字

说明

详细内容参考**INSERT相关语句之间的区别**

使用示例

以下为查询用INSERT语句创建的column值的示例

```
gSQL> CREATE TABLE region
(
    r_regionkey   BIGINT    GENERATED BY DEFAULT AS IDENTITY
  , r_name        CHAR(25)  DEFAULT 'N/A'
);

Table created.

gSQL> COMMIT;

Commit complete.
```

- RETURNING创建的DEFAULT值的情况

```
gSQL> INSERT INTO region VALUES ( DEFAULT, DEFAULT ) RETURNING
r_regionkey, r_name;

R_REGIONKEY R_NAME
----------- -------------------------
          1 N/A
```

```
1 row created.
```

- RETURNING忽略的column值的情况

```
gSQL> INSERT INTO region(r_name) VALUES ('ASIA') RETURNING r_regionkey;


R_REGIONKEY
-----------
          2


1 row created.
```

以下为查询从子查询中创建的行数据的示例

```
gSQL> INSERT INTO region
      SELECT r_regionkey, r_name FROM tmp_region WHERE r_regionkey < 3
      RETURNING r_regionkey, r_name;


R_REGIONKEY R_NAME
----------- -------------------------
          0 AFRICA
          1 AMERICA
          2 ASIA


3 rows created.
```

兼容性

标准SQL未定义<insert returning query statement>

参考

相关内容参考以下内容

- **INSERT INTO**
- **INSERT INTO name RETURNING .. INTO**

10.3 INSERT INTO name RETURNING .. INTO

功能

在表创建一个行数据并通过主机变量获取此行的值

语句

```
<insert statement> ::=
    INSERT INTO table_name [ ( column_name [, ...] ) ]
        <insert source>
        <returning into clause>
    ;

<insert source> ::=
      <values clause>
    | <from subquery>
    | <from default>

<values clause> ::=
    VALUES { ( { <value expression> | DEFAULT } [, ...] ) } [, ...]

<from subquery> ::=
```

```
    <query expression>


<from default> ::=

    DEFAULT VALUES


<returning into clause> ::=

      [ RETURN | RETURNING ] { * | { <value expression> [ [AS]
alias_name ] } [, ...] INTO variable_name [, ...]
```

使用范围及访问权限

用户应满足以下条件才能执行<insert returning into statement>语句

- 用户需要有以下权限中的一个才能执行INSERT语句
  - 对作为插入对象的所有column有INSERT(columns) ON TABLE
  - 对表有(INSERT或CONTROL TABLE) ON TABLE
  - 对表所在的SCHEMA有(INSERT TABLE或CONTROL SCHEMA) ON SCHEMA
  - INSERT ANY TABLE ON DATABASE
- 使用<from subquery>时对语句中使用的所有表需要有以下权限中的一个
  - 对表的column中用于语句的所有column有SELECT(columns) ON TABLE
  - 对表有(SELECT或CONTROL TABLE) ON TABLE
  - 对表所在的SCHEMA有(SELECT TABLE或CONTROL SCHEMA) ON SCHEMA
  - SELECT ANY TABLE ON DATABASE
- 对用于RETURNING的所有column需要有以下权限中的一个
  - 对用于RETURNING语句的所有column有SELECT(columns) ON TABLE

- 对表有(SELECT或CONTROL TABLE) ON TABLE
- 对表所在的SCHEMA有(SELECT TABLE或CONTROL SCHEMA) ON SCHEMA
- SELECT ANY TABLE ON DATABASE

语句规则及参数

table_name

创建行数据的对象表的名称

[(column_name [, ...])]

表的column名

详细内容参考**INSERT INTO**

<values clause>

要向对应Column分配的值的列表

详细内容参考**INSERT INTO**

<from subquery>

生成row的查询

详细内容参考**INSERT INTO**

DEFAULT VALUES

所有Column使用默认值

详细内容参考**INSERT INTO**

<returning clause>

返回插入的行数据

参考**INSERT INTO name RETURNING**的**<returning clause>**

INTO variable_name [, ...]

INTO中定义的变量数量应与RETURNING中定义的expression数量相同

创建的行数据应为一条以下插入两条以上row时报错

说明

详细内容参考**INSERT相关语句之间的区别**

使用示例

以下为通过主机变量获取创建的行的值的示例

```
gSQL> CREATE TABLE region
```

```
(
    r_regionkey   BIGINT    GENERATED BY DEFAULT AS IDENTITY
  , r_name        CHAR(25)  DEFAULT 'N/A'
);

Table created.

gSQL> COMMIT;

Commit complete.
```

- 声明主机变量

```
\VAR v_key  BIGINT
\VAR v_name VARCHAR(128)
```

- 通过主机变量获取创建的DEFAULT值

```
gSQL> INSERT INTO region
      VALUES ( DEFAULT, DEFAULT )
      RETURNING r_regionkey, r_name
      INTO :v_key, :v_name;

V_KEY V_NAME
----- -------------------------
    1 N/A
```

```
1 row created.
```

- 通过主机变量获取省略的column值

```
gSQL> INSERT INTO region(r_name)
      VALUES ('ASIA')
      RETURNING r_regionkey
      INTO :v_key;


V_KEY
-----
    2


1 row created.
```

兼容性

标准SQL未定义<insert returning into statement>语句

参考

相关内容参考以下内容

- INSERT INTO
- INSERT INTO name RETURNING

10.4 INSERT INTO name ... UPDATE

功能

在表中创建新的row如果有违unique约束条件则更新原来的row

语句

```
<upsert statement> ::=
    INSERT INTO table_name [ ( column_name [, ...] ) ]
        <insert source>
        <duplicate key clause>
    ;

<insert source> ::=
      <values clause>
    | <from subquery>
    | DEFAULT VALUES

<values clause> ::=
    VALUES { ( { <value expression> | DEFAULT } [, ...] ) } [, ...]

<from subquery> ::=
```

```
    <query expression>


<duplicate key clause>
    ON DUPLICATE KEY { DO NOTHING | <do update clause> }


<do update clause> ::=
    [DO] UPDATE [SET] <set clause>  [, ...]


<set value clause> ::=
      <value expression>
    | DEFAULT
    | VALUES( column_name )


<set clause> ::=
      column_name = <set value clause>
    | ( column_name [, ...] ) = ( <set value clause> [, ...] )
    | ( column_name [, ...] ) = ( <query expression> )
```

使用范围及访问权限

用户需要满足以下条件才能执行<upsert statement>语句

- 用户需要拥有以下权限中的一个才能执行该语句
  - 对作为insert对象的所有column有INSERT(columns) ON TABLE
  - 对表有(INSERT或者CONTROL TABLE) ON TABLE

    - 对表所属架构有(INSERT TABLE或CONTROL SCHEMA) ON SCHEMA
    - INSERT ANY TABLE ON DATABASE
    - 对作为update对象的所有column有UPDATE(columns) ON TABLE
    - 对表有(UPDATE或者CONTROL TABLE) ON TABLE
    - 对表所属SCHEMA有(UPDATE TABLE或CONTROL SCHEMA) ON SCHEMA
    - UPDATE ANY TABLE ON DATABASE
- 使用<from subquery>时对语句中使用的所有表需要以下权限中的一个
    - 表的column中被用于语句的所有column有SELECT(columns) ON TABLE
    - 对表有(SELECT或者CONTROL TABLE) ON TABLE
    - 对表所属SCHEMA有(SELECT TABLE或CONTROL SCHEMA) ON SCHEMA
    - SELECT ANY TABLE ON DATABASE
- 对RETURNING子句中使用的所有column需要以下权限中的一个
    - 对RETURNING语句中使用的所有column有SELECT(columns) ON TABLE
    - 对表有(SELECT或者CONTROL TABLE) ON TABLE
    - 对表所属SCHEMA有(SELECT TABLE或CONTROL SCHEMA) ON SCHEMA
    - SELECT ANY TABLE ON DATABASE

语句规则及参数

table_name

创建row的目标表名

如果有违unique约束条件的情况下执行update时则为要变更的目标表名

可以像schema_name.table_name一样定义表所属的SCHEMA如果省略schema_name则会使用执

行语句的用户的默认SCHEMA名

[(column_name [, ...])]

表的column名

详细内容参考**INSERT INTO**语句的**[(column_name [, ...])]**

<values clause>

要向对应column分配的值的列表

详细内容参考**INSERT INTO**语句的**<values clause>**

<from subquery>

创建row的查询

详细内容参考**SELECT**语句的**query expression**

DEFAULT VALUES

所有column使用默认值

详细内容参考**INSERT INTO**语句的**DEFAULT VALUES**

<duplicate key clause>

定义当违反unique约束条件时要采取的动作

DO NOTHING

定义当违反unique约束条件时不采取任何动作

<do update clause>

当违反unique约束条件时按照<set clause>更新column的值

<set value clause>

定义要向更新的column分配的值

可以使用如下方式进行定义

- column_name = <value expression>

```
DO UPDATE SET column1 = value1, column2 = value2, column3 = value3
```

- column_name = DEFAULT

```
DO UPDATE SET column1 = DEFAULT, column2 = DEFAULT, column3 = DEFAULT
```

- column_name = VALUES(column_name)

将<insert source>的值作为更新的值

```
DO UPDATE SET column1 = VALUES(column1), column2 = VALUES(column2),
```

```
column3 = VALUES(column2)
```

<set clause>

定义要更新的column和要分配的值<set clause>的column数和值的数相同

可使用如下的方式进行定义

- column_name = { <set value clause> }

```
ON DUPLICATE KEY
   DO UPDATE SET column1 = value1, column2 = value2, column3 = value3
```

- (column_name [, ...]) = (<set value clause> } [, ...])

```
ON DUPLICATE KEY
   DO UPDATE SET ( column1, column2, column3 ) = ( value1, value2,
value3 )
```

- (column_name [, ...]) = (<query expression>)

```
ON DUPLICATE KEY
   DO UPDATE SET column1 = ( SELECT max(value1) FROM other_table_name )
```

<query expression>必须是创建单个row的查询

Column的值如果使用默认值那么执行**CREATE TABLE**时将使用定义的默认值（参考**<default clause>**）未定义时使用NULL值

说明

INSERT INTO name ... UPDATE相关语句之间的差异

- **INSERT INTO name ... UPDATE**
  - 在表中创建row如果有违unique约束条件时则更新原row
  - 例：INSERT INTO t1 VALUES (1, 1) ON DUPLICATE KEY UPDATE c2 = c2 + 1;
- **INSERT INTO name ... UPDATE RETURNING**
  - 在表中创建row或者更新原row插入或者更新的row可以使用与SELECT语句相同的方式(SQLFetch()等API)进行查询
  - 例：INSERT INTO t1 VALUES (1, 1) ON DUPLICATE KEY UPDATE c2 = c2 + 1 RETURNING c2;
- **INSERT INTO name ... UPDATE RETURNING ... INTO**
  - 创建或更新一条以下的row创建或更新的row为一条时将从RETURNING INTO的host变量获取值
  - 例：INSERT INTO t1 VALUES (1, 1) ON DUPLICATE KEY UPDATE c2 = c2 + 1 RETURNING c2 INTO :v1;

<upsert statement>是deterministic statement

如下所示同值的两个不同UPSERT语句需要得出相同的结果

- INSERT INTO t1 VALUES(1),(2),(3) ON DUPLICATE KEY UPDATE c1 = c1 + 1;
- INSERT INTO t1 VALUES(3),(2),(1) ON DUPLICATE KEY UPDATE c1 = c1 + 1;

```
gSQL> CREATE TABLE t1 ( c1 INTEGER UNIQUE );


Table created.


gSQL> INSERT INTO t1 VALUES( 1 ),( 2 ),( 3 );


3 rows created.


gSQL> INSERT INTO t1 VALUES( 1 ),( 2 ),( 3 ) ON DUPLICATE KEY UPDATE c1 =
c1 + 1;


3 rows created.


gSQL> SELECT * FROM t1;


C1
--
 2
 3
 4


3 rows selected.
```

```
gSQL> CREATE TABLE t1 ( c1 INTEGER UNIQUE );
```

```
Table created.

gSQL> INSERT INTO t1 VALUES( 1 ),( 2 ),( 3 );

3 rows created.

gSQL> INSERT INTO t1 VALUES( 3 ),( 2 ),( 1 ) ON DUPLICATE KEY UPDATE c1 =
c1 + 1;

3 rows created.

gSQL> SELECT * FROM t1;

C1
--
 2
 3
 4

3 rows selected.
```

使用示例

以下为违背unique约束条件更新单个row的示例

```
gSQL> CREATE TABLE t1 ( c1 INTEGER UNIQUE );

Table created.

gSQL> INSERT INTO t1 VALUES( 1 );

1 row created.

gSQL> INSERT INTO t1 VALUES( 1 ) ON DUPLICATE KEY UPDATE c1 = c1 + 1;

1 row created.

gSQL> SELECT * FROM t1;

C1
--
 2

1 row selected.
```

以下为违背unique约束条件不更新row的示例

```
gSQL> CREATE TABLE t1 ( c1 INTEGER UNIQUE );

Table created.

gSQL> INSERT INTO t1 VALUES( 1 );

1 row created.

gSQL> INSERT INTO t1 VALUES( 1 ) ON DUPLICATE KEY DO NOTHING;

no rows created.

gSQL> SELECT * FROM t1;

C1
--
 1

1 row selected.
```

以下为使用subquery插入或更新多个row的示例

```
gSQL> CREATE TABLE t1 ( c1 INTEGER UNIQUE );

Table created.
```

```
gSQL> INSERT INTO t1 VALUES( 1 ),( 2 ),( 3 ),( 4 );


4 rows created.


gSQL> INSERT INTO t1 ( SELECT c1 FROM t1 ) ON DUPLICATE KEY UPDATE c1 = c1
+ 1;


4 rows created.


gSQL> SELECT * FROM t1;


C1
--
 2
 3
 4
 5


4 rows selected.
```

兼容性

SQL标准中没有定义<upsert statement>语句

参考

相关内容参考以下内容

- **20.1 INSERT INTO**
- **20.5 INSERT INTO name ... UPDATE RETURNING**
- **20.6 INSERT INTO name ... UPDATE RETURNING ... INTO**
- **20.27 UPDATE**

10.5 INSERT INTO name ... UPDATE RETURNING

功能

在表中创建新的row如果违背unique约束条件则更新原来的row之后再查询创建或变更的row

语句

```
<upsert returning statement> ::=

    INSERT INTO table_name [ ( column_name [, ...] ) ]

        <insert source>

        <duplicate key clause>

        <returning clause>

    ;

<insert source> ::=

      <values clause>

    | <from subquery>

    | DEFAULT VALUES


<values clause> ::=

    VALUES { ( { <value expression> | DEFAULT } [, ...] ) } [, ...]
```

```
<from subquery> ::=

    <query expression>


<duplicate key clause>

    ON DUPLICATE KEY { DO NOTHING | <do update clause> }


<do update clause> ::=

    [DO] UPDATE [SET] <set clause>  [, ...]


<set value clause> ::=

      <value expression>

    | DEFAULT

    | VALUES( column_name )


<set clause> ::=

      column_name = <set value clause>

    | ( column_name [, ...] ) = ( <set value clause> [, ...] )

    | ( column_name [, ...] ) = ( <query expression> )


<returning clause> ::=

    [ RETURN | RETURNING ] { * | { <value expression> [ [AS]

alias_name ] } [, ...]
```

使用范围及访问权限

用户需要满足以下条件才能执行<upsert returning statement>语句

- 用户需要以下权限中的一个才能执行该语句
  - 对作为Insert对象的所有column有INSERT(columns) ON TABLE
  - 对表有(INSERT或者CONTROL TABLE) ON TABLE
  - 对表所属SCHEMA有(INSERT TABLE或CONTROL SCHEMA) ON SCHEMA
  - INSERT ANY TABLE ON DATABASE
  - 对作为Update对象的所有column有UPDATE(columns) ON TABLE
  - 对表有(UPDATE或者CONTROL TABLE) ON TABLE
  - 对表所属SCHEMA有(UPDATE TABLE或CONTROL SCHEMA) ON SCHEMA
  - UPDATE ANY TABLE ON DATABASE
- 使用<from subquery>时对语句中使用的所有表需要以下权限中的一个
  - 表的column中被用于语句的所有column有SELECT(columns) ON TABLE
  - 对表有(SELECT或者CONTROL TABLE) ON TABLE
  - 对表所属SCHEMA有(SELECT TABLE或CONTROL SCHEMA) ON SCHEMA
  - SELECT ANY TABLE ON DATABASE
- 对RETURNING子句中使用的所有column需要以下权限中的一个
  - 对RETURNING语句中使用的所有column有SELECT(columns) ON TABLE
  - 对表有(SELECT或者CONTROL TABLE) ON TABLE
  - 对表所属SCHEMA有(SELECT TABLE或CONTROL SCHEMA) ON SCHEMA
  - SELECT ANY TABLE ON DATABASE

语句规则及参数

table_name

创建row的目标表名

如果在违背unique约束条件的情况下执行update时则为要变更的目标表名

详细内容参考**INSERT INTO name ... UPDATE**语句的**table_name**

[(column_name [, ...])]

是表的column名

详细内容参考**INSERT INTO**语句的**[(column_name [, ...])]**

<values clause>

要向对应column分配的值的列表

详细内容参考**INSERT INTO**语句的**<values clause>**

<from subquery>

创建row的查询

详细内容参考**SELECT**语句的**query expression**

DEFAULT VALUES

所有column使用默认值

详细内容参考**INSERT INTO**语句的**DEFAULT VALUES**

<duplicate key clause>

定义当违反unique约束条件时要采取的动作

DO NOTHING

定义当违反unique约束条件时不采取任何动作

<do update clause>

当违反unique约束条件时按照<set clause>更新column的值

<set value clause>

定义向要更新的column分配的值

详细内容参考**20.4 INSERT INTO name ... UPDATE**语句的**<set value clause>**

<set clause>

定义要更新的column和要分配的值<set clause>的column数和值的数相同

详细内容参考**20.4 INSERT INTO name ... UPDATE**语句的**<set clause>**

<returning clause>

返回插入或者变更的row

- 将创建的row作为结果集记述其中需要查询的column
  - RETURNING语句返回将已插入或变更的row作为结果集的结果
  - <value expression>
    - 与SELECT语句的<select list>相同但是不能使用aggregation等
  - [[AS] alias_name]
    - 可使用AS语句指定value expression的名称

说明

详细内容参考**INSERT INTO name ... UPDATE相关语句之间的差异**

以下为插入四个row后再返回插入的结果的示例

```
gSQL> CREATE TABLE t1 ( c1 INTEGER UNIQUE );


Table created.


gSQL> INSERT INTO t1 VALUES( 1 ), ( 2 ), ( 3 ), ( 4 ) ON DUPLICATE KEY
UPDATE c1 = c1 + 1 RETURNING c1;
```

```
C1
--
 1
 2
 3
 4

4 rows created.
```

以下为违背unique约束条件更新row后返回更新后结果的示例

```
gSQL> CREATE TABLE t1 ( c1 INTEGER UNIQUE );

Table created.

gSQL> INSERT INTO t1 VALUES( 1 ),( 2 ),( 3 ),( 4 );

4 rows created.

gSQL> INSERT INTO t1 ( SELECT c1 FROM t1 ) ON DUPLICATE KEY UPDATE c1 = c1
+ 1 RETURNING c1;

C1
--
 2
 3
```

```
4

5


4 rows created.
```

兼容性

SQL标准中没有定义<upsert returning statement>语句

参考

参考以下相关内容

- **INSERT INTO name ... UPDATE**
- **INSERT INTO name ... UPDATE RETURNING ... INTO**

10.6 INSERT INTO name ... UPDATE RETURNING ... INTO

功能

在表中创建一条row如果违背unique约束条件则更新原来的row之后通过host变量获取创建或者变更的row的值

语句

```
<upsert returning into statement> ::=
    INSERT INTO table_name [ ( column_name [, ...] ) ]
        <insert source>
        <duplicate key clause>
        <returning clause>
        <into clause>
    ;
<insert source> ::=
      <values clause>
    | <from subquery>
    | DEFAULT VALUES

<values clause> ::=
```

```
    VALUES { ( { <value expression> | DEFAULT } [, ...] ) } [, ...]


<from subquery> ::=

    <query expression>


<duplicate key clause>

    ON DUPLICATE KEY { DO NOTHING | <do update clause> }


<do update clause> ::=

    [DO] UPDATE [SET] <set clause>  [, ...]


<set value clause> ::=

      <value expression>

    | DEFAULT

    | VALUES( column_name )


<set clause> ::=

      column_name = <set value clause>

    | ( column_name [, ...] ) = ( <set value clause> [, ...] )

    | ( column_name [, ...] ) = ( <query expression> )


<returning clause> ::=

    [ RETURN | RETURNING ] { * | { <value expression> [ [AS]
alias_name ] } [, ...]
```

```
<into clause> ::= INTO variable_name [, ...]
```

使用范围及访问权限

用户需要满足以下条件才能执行<upsert returning into statement>语句

- 用户需要以下权限中的一个才能执行该语句
  - 对作为Insert对象的所有column有INSERT(columns) ON TABLE
  - 对表有(INSERT或者CONTROL TABLE) ON TABLE
  - 对表所属SCHEMA有(INSERT TABLE或CONTROL SCHEMA) ON SCHEMA
  - INSERT ANY TABLE ON DATABASE
  - 对作为Update对象的所有column有UPDATE(columns) ON TABLE
  - 对表有(UPDATE或者CONTROL TABLE) ON TABLE
  - 对表所属SCHEMA有(UPDATE TABLE或CONTROL SCHEMA) ON SCHEMA
  - UPDATE ANY TABLE ON DATABASE
- 使用<from subquery>时对语句中使用的所有表需要以下权限中的一个
  - 表的column中被用于语句的所有column有SELECT(columns) ON TABLE
  - 对表有(SELECT或者CONTROL TABLE) ON TABLE
  - 对表所属SCHEMA有(SELECT TABLE或CONTROL SCHEMA) ON SCHEMA
  - SELECT ANY TABLE ON DATABASE
- 对RETURNING子句中使用的所有column需要以下权限中的一个
  - 对RETURNING语句中使用的所有column有SELECT(columns) ON TABLE
  - 对表有(SELECT或者CONTROL TABLE) ON TABLE
  - 对表所属SCHEMA有(SELECT TABLE或CONTROL SCHEMA) ON SCHEMA
  - SELECT ANY TABLE ON DATABASE

语句规则及参数

table_name

创建row的目标表名

如果在违背unique约束条件的情况下执行update时则为要变更的目标表名

详细内容参考**INSERT INTO name ... UPDATE**语句的**table_name**

[(column_name [, ...])]

是表的column名

详细内容参考**INSERT INTO**语句的**[(column_name [, ...])]**

<values clause>

要向对应column分配的值的列表

详细内容参考**INSERT INTO**语句的**<values clause>**

<from subquery>

创建row的查询

详细内容参考**SELECT**语句的**query expression**

DEFAULT VALUES

所有column使用默认值

详细内容参考**INSERT INTO**语句的**DEFAULT VALUES**

<duplicate key clause>

定义当违反unique约束条件时要采取的动作

DO NOTHING

定义当违反unique约束条件时不采取任何动作

<do update clause>

当违反unique约束条件时按照<set clause>更新column的值

<set value clause>

定义向要更新的column分配的值

详细内容参考**INSERT INTO name ... UPDATE**语句的**<set value clause>**

<set clause>

定义要更新的column和要分配的值<set clause>的column数和值的数应相同

详细内容参考**INSERT INTO name ... UPDATE**语句的**<set clause>**

<returning clause>

返回插入或者变更的row

详细内容参考**INSERT INTO name ... UPDATE RETURNING**语句的**<returning clause>**

<into clause>

INTO语句中记述的变量数需与RETURNING语句中记述的expression的个数相同

要创建的row需少于一条生成2条以上Row则会出现错误

说明

详细内容参考**INSERT INTO name ... UPDATE相关语句之间的差异**

以下是插入单个row后通过主机变量获得插入的结果的示例

```
gSQL> \VAR v_c1 INTEGER;


gSQL> CREATE TABLE t1 ( c1 INTEGER UNIQUE );


Table created.


gSQL> INSERT INTO t1 VALUES( 1 ) ON DUPLICATE KEY UPDATE c1 = c1 + 1
RETURNING c1 INTO :v_c1;
```

```
V_C1
----
   1

1 row created.
```

以下为违背unique约束条件更新了一个row后通过主机变量获取更新结果的示例

```
gSQL> \VAR v_c1 INTEGER;

gSQL> CREATE TABLE t1 ( c1 INTEGER UNIQUE );

Table created.

gSQL> INSERT INTO t1 VALUES( 1 );

1 row created.

gSQL> INSERT INTO t1 VALUES( 1 ) ON DUPLICATE KEY UPDATE c1 = c1 + 1
RETURNING c1 INTO :v_c1;

V_C1
----
   2
```

```
1 row created.
```

兼容性

SQL标准中没有定义<upsert returning into statement>语句

参考

相关内容参考下文

- **INSERT INTO name ... UPDATE**
- **INSERT INTO name ... UPDATE RETURNING**

10.7 LOCK TABLE

功能

对一个以上的表设置LOCK

语句

```
<lock table statement> ::=
    LOCK TABLE lock target [, ...]
    IN <lock mode> MODE [<wait clause>]
    ;

<lock mode> ::=
    SHARE
    | EXCLUSIVE
    | ROW SHARE
    | ROW EXCLUSIVE
    | SHARE ROW EXCLUSIVE

<wait clause> ::=
    NOWAIT
```

```
    | WAIT time
```

使用范围及访问权限

用户需要有以下权限中的一个才能执行<lock table statement>语句

- 对表有(LOCK或CONTROL TABLE) ON TABLE
- 对表所在的SCHEMA有(LOCK TABLE或CONTROL SCHEMA) ON SCHEMA
- LOCK ANY TABLE ON DATABASE

语句规则及参数

<lock target>

指定锁定的对象表

<lock mode>

指定锁定模式

- SHARE
  - 对locked table允许并发执行查询但不允许update表
- EXCLUSIVE
  - 允许对locked table执行排他性查询处理
- ROW SHARE

- 允许并发访问locked table但不允许为了exclusive access而对整个表的锁定

- ROW EXCLUSIVE
  - 允许并发访问locked table但不允许为了exclusive access而对整个表的锁定
  - 设置为ROW EXCLUSIVE mode时拒绝SHARE mode的锁定
  - Updateinsertdelete时自动赋予ROW EXCLUSIVE mode
- SHARE ROW EXCLUSIVE
  - 全表扫描或向其他用户检索表的row时使用
  - 拒绝其他用户访问以SHARE mode锁定的表或更新中的行

<wait clause>

指定为了获取锁而等待的时间

- NOWAIT
  - 立即获取对象的lock控制
  - 已被其他用户锁定时立即接收控制权
    - 这时数据库将产生message
- WAIT time
  - 设置为了获取锁而等待的时间
  - 以秒为单位使用0 ~ 1000000000之间的值
- 未指定时无限等待直到获取lock

说明

提交或回滚事务时自动解除所有获取的锁使用ROLLBACK TO SAVEPOINT语句时解除对应

savepoint以后获取的所有锁

使用示例

以下为使其他事务无法对t1表执行任何变更运算的示例

```
gSQL> LOCK TABLE t1 IN EXCLUSIVE MODE;


Table locked.
```

以下为对多个表执行LOCK语句的示例

```
gSQL> LOCK TABLE t1, t2 IN EXCLUSIVE MODE;


Table locked.
```

以下为对t1表获取SHARE ROW EXCLUSIVE lock的示例

```
gSQL> LOCK TABLE t1 IN SHARE ROW EXCLUSIVE MODE;


Table locked.
```

以下为可对对应TABLE立即获取lock时执行的语句无法及时获取lock时报错

```
gSQL> LOCK TABLE t1 IN EXCLUSIVE MODE NOWAIT;
```

```
Table locked.
```

以下为为了获取lock而等待10秒的示例

```
gSQL> LOCK TABLE t1 IN EXCLUSIVE MODE WAIT 10;


Table locked.
```

兼容性

标准SQL未定义lock table的概念

参考

相关内容参考下文

- **COMMIT**
- **ROLLBACK**

10.8 NOAUDIT POLICY

功能

禁用audit policy

语句

```
<noaudit policy statement> ::=
    NOAUDIT POLICY policy_name
    [ <specified_user_option> ]
    ;

<specified_user_option> ::=
      BY user_name [, ...]
```

使用范围及访问权限

用户需要有AUDIT SYSTEM ON DATABASE权限才能执行<noaudit policy statement>

语句规则及参数

policy_name

要禁用的audit policy的对象的名称

禁用的audit policy不影响原有的会话只影响新生成的会话

<specified_user_option>

指定从审计对象中排除的用户

NOAUDIT POLICY语句与AUDIT POLICY语句不同没有EXCEPT选项

使用AUDIT POLICY name BY时使用NOAUDIT POLICY name BY语句禁用

使用AUDIT POLICY name EXCEPT时使用NOAUDIT POLICY name语句禁用

根据AUDIT POLICY语句的使用方法如下使用NOAUDIT POLICY语句禁用相应选项

类型	AUDIT POLICY语句	NOAUDIT POLICY语句
所有用户	AUDIT POLICY p1	NOAUDIT POLICY p1
使用BY	AUDIT POLICY p1 BY u1	NOAUDIT POLICY p1 BY u1
使用EXCEPT	AUDIT POLICY p1 EXCEPT u1	NOAUDIT POLICY p1

Table 10-2 Audit policy的激活/禁用

禁用已激活的所有用户时将完全禁用audit policy对象

说明

如下查询audit policy对象的激活信息

```
SELECT policy_name
     , enabled_opt
     , user_name
  FROM audit_policy_enabled
 WHERE policy_name = 'P1';
```

NOAUDIT POLICY语句删除根据AUDIT POLICY指定方式创建的个别激活信息

没有通过以上查询激活的信息时将全面禁用audit policy

如下激活所有用户时NOAUDIT POLICY BY不产生任何影响

```
AUDIT POLICY p1;
```

- 不产生任何影响

```
NOAUDIT POLICY p1 BY u1;
```

- 应如下禁用

```
NOAUDIT POLICY p1;
```

对一个以上的用户个别进行激活时根据AUDIT POLICY设置方法使用NOAUDIT POLICY语句

使用BY激活的情况

如下激活audit policy时

```
AUDIT POLICY p1 WHENEVER NOT SUCCESSFUL;

AUDIT POLICY p1 BY u1;

AUDIT POLICY p1 BY u2;
```

激活信息的查询结果如下

```
SELECT policy_name
     , enabled_opt
     , user_name
     , when_success
     , when_failure
  FROM audit_policy_enabled
 WHERE policy_name = 'P1';


POLICY_NAME  ENABLED_OPT  USER_NAME    WHEN_SUCCESS  WHEN_FAILURE
-----------  -----------  ---------    ------------  ------------
P1           BY           ALL USERS    NO            YES
P1           BY           U1           YES           YES
P1           BY           U2           YES           YES
```

以下为执行NOAUDIT语句并查询激活信息的示例

```
NOAUDIT POLICY p1;


SELECT policy_name
     , enabled_opt
     , user_name
     , when_success
     , when_failure
  FROM audit_policy_enabled
 WHERE policy_name = 'P1';


POLICY_NAME  ENABLED_OPT  USER_NAME    WHEN_SUCCESS    WHEN_FAILURE
-----------  -----------  ---------    ------------    ------------
P1           BY           U1           YES             YES
P1           BY           U2           YES             YES
```

禁用对ALL USERS的failure的审计仍然激活对u1u2用户的审计

如下通过BY选项追加使用NOAUDIT POLICY语句则全面禁用audit policy p1

```
NOAUDIT POLICY p1 BY u1, u2;


SELECT policy_name
     , enabled_opt
     , user_name
     , when_success
     , when_failure
  FROM audit_policy_enabled
```

```
 WHERE policy_name = 'P1';


no rows selected.
```

使用EXCEPT激活的情况

如下激活audit policy时

```
AUDIT POLICY p1 EXCEPT u1, sys;
```

激活信息的查询结果如下

```
SELECT policy_name
     , enabled_opt
     , user_name
     , when_success
     , when_failure
  FROM audit_policy_enabled
 WHERE policy_name = 'P1';


POLICY_NAME  ENABLED_OPT  USER_NAME    WHEN_SUCCESS    WHEN_FAILURE
-----------  -----------  ---------    ------------    ------------
P1           EXCEPT       U1           YES             YES
P1           EXCEPT       SYS          YES             YES
```

NOAUDIT POLICY语句与AUDIT POLICY语句不同没有EXCEPT option因此如下执行语句

```
NOAUDIT POLICY p1;


SELECT policy_name
     , enabled_opt
     , user_name
     , when_success
     , when_failure
  FROM audit_policy_enabled
 WHERE policy_name = 'P1';


no rows selected.
```

即使用EXCEPT选项激活audit policy时无法通过NOAUDIT POLICY语句再次激活个别用户

使用示例

以下为禁用所有用户的示例

```
NOAUDIT POLICY table_pol;
```

以下为禁用通过BY激活的特定用户的示例

```
NOAUDIT POLICY table_pol BY u1;
```

兼容性

标准SQL没有audit policy

参考

相关内容参考下文

- Audit policy 对象管理
  - **CREATE AUDIT POLICY**
  - **DROP AUDIT POLICY**
  - **ALTER AUDIT POLICY**
- Audit policy 激活/禁用
  - **AUDIT POLICY**
  - **NOAUDIT POLICY**
- Audit trail 查询: **AUDIT_TRAIL**
- Audit trail 清除: **ALTER DATABASE CLEAR AUDIT TRAIL**

10.9 OPEN cursor_name

功能

打开游标

语句

```
<open statement> ::=
    OPEN cursor_name [ <parameter using clause> ]
    ;

<parameter using clause> ::=
      <using parameter arguments>

<using parameter arguments> ::=
    USING variable_name [, ...]
```

使用范围及访问权限

cursor_name为通过**PREPARE statement_name**与**DECLARE cursor_name**语句声明的动态游标

时可在Embedded SQL中使用

与声明cursor_name的**DECLARE cursor_name**语句包含的**<cursor query>**的权限相同

语句规则及参数

cursor_name

应为在会话内用**DECLARE cursor_name**语句声明的游标

<parameter using clause>

可在Embedded SQL中可使用

使用<parameter using clause>语句时cursor_name应为使用**PREPARE statement_name**语句与**DECLARE cursor_name**语句声明的动态游标

<using parameter arguments>

使用<using parameter arguments>时variable_name数量应与**PREPARE statement_name**参考的query包含的parameter的数量相同

variable_name根据列出的顺序对应dynamic parameter顺序

```
{
    ...
    EXEC SQL PREPARE stmt1 FROM 'SELECT c1, c2 FROM t1 WHERE c1 IN
( ?, ?, ? )';
```

```
    EXEC SQL DECLARE cur1 CURSOR FOR stmt1;
    EXEC SQL OPEN cur1 USING :sValue1, :sValue2, :sValue3;
    ...
    EXEC SQL WHENEVER NOT FOUND DO break;
    for(;;)
    {
        EXEC SQL FETCH cur1 INTO :sC1, :sC2;
    }
    EXEC SQL WHENEVER NOT FOUND CONTINUE;
    ...
    EXEC SQL CLOSE cur1;
    ...
}
```

说明

游标是在会话内可区分的对象其他会话内使用的游标与当前会话内使用的游标无关

使用OPEN cursor_name语句时应为通过**DECLARE cursor_name**语句声明的游标而且游标应是关闭的状态

使用示例

以下为在interactive sql(gsql)中声明游标并使用OPEN cursor语句的示例

```
gSQL> DECLARE cur1 CURSOR FOR SELECT id, data FROM t1;


Cursor declared.


gSQL> OPEN cur1;


Cursor is open.


gSQL> \var v_id   INTEGER
gSQL> \var v_data VARCHAR(128)


gSQL> FETCH cur1 INTO :v_id, :v_data;


V_ID V_DATA
---- ------
   1 data_1


1 row fetched.


gSQL> FETCH cur1 INTO :v_id, :v_data;


V_ID V_DATA
---- ------
   2 data_2
```

```
1 row fetched.

gSQL> FETCH cur1 INTO :v_id, :v_data;

V_ID V_DATA
---- ------
   3 data_3

1 row fetched.

gSQL> FETCH cur1 INTO :v_id, :v_data;

V_ID V_DATA
---- ------
   4 data_4

1 row fetched.

gSQL> FETCH cur1 INTO :v_id, :v_data;

V_ID V_DATA
---- ------
   5 data_5

1 row fetched.
```

```
gSQL> FETCH cur1 INTO :v_id, :v_data;

no rows fetched.

gSQL> CLOSE cur1;

Cursor closed.
```

兼容性

Feature ID	说明	是否支持
B031	Basic Dynamic SQL	O

Table 10-3 标准SQL兼容性

参考

相关内容参考下文

- **DECLARE cursor_name**
- **FETCH cursor_name**
- **CLOSE cursor_name**

- **PREPARE statement_name**

10.10 PREPARE statement_name

功能

准备反复执行的dynamic SQL语句

语句

```
<prepare statement> ::=

    PREPARE statement_name FROM <SQL statement variable>

    ;


<SQL statement variable> ::=

      variable_name

    | 'sql statement'

    | "sql statement"

    | sql statement
```

使用范围及访问权限

可在embedded SQL中使用

需要有符合dynamic SQL语句类型的执行权限

语句规则及参数

statement_name

要准备的statement名

statement名的长度应小于128 byte

后续执行的**EXECUTE statement_name**或**DECLARE cursor_name**语句参考statement_name

有相同的statement_name时删除之前准备的dynamic SQL

```
{
    ...

    EXEC SQL PREPARE stmt1 FROM 'DELETE FROM t1';
    ...
    EXEC SQL PREPARE stmt1 FROM 'UPDATE t1 SET c1 = c1 + 10';
    ...
}
```

<SQL statement variable>

<SQL statement variable>的使用分以下四种

- variable_name : 存储SQL的变量
- 'sql statement' : 用单引号(')引住的SQL
- "sql statement" : 用双引号(")引住的SQL

- sql statement : 没有引号的SQL

单引号内使用字符串数据时如下使用两次单引号

```
{
    ...
    PREPARE stmt_name FROM 'INSERT INTO t1 VALUES ( ''literal data'' )';
    ...
}
```

<SQL statement variable>参照的动态SQL语句可使用主机变量(:var)或parameter marker(?)

但使用没有引号的SQL语句时不能使用parameter marker(?)

根据参照的dynamic SQL语句的特性变量为input或output dynamic parameter

dynamic SQL语句里定义的dynamic parameter对变量名没有特殊的意义与类型无关按照描述的顺序识别

- 例 1

```
{
    ...
    int sValue1;
    int sValue2;
    ...
    EXEC SQL PREPARE stmt1 FROM 'DELETE FROM t1 WHERE c1 BETWEEN ? AND ?';
    EXEC SQL EXECUTE stmt1 USING :sValue1, :sValue2;
    ...
```

```
}
```

- 所有parameter marker为input dynamic parameter
- 识别顺序
  - 第一个 - BETWEEN ?
    - input dynamic parameter
    - 使用:sValue1值
  - 第二个 - AND ?
    - input dynamic parameter
    - 使用:sValue2值
- 例 2

```
{
    ...
    int sValue1;
    int sValue2;
    ...
    EXEC SQL PREPARE stmt1 FROM 'SELECT SUM(c2) INTO :v1 FROM t1 WHERE
c1 > :v2';
    EXEC SQL EXECUTE stmt1 USING :sValue1, :sValue2;
    ...
}
```

- 存在input dynamic parameter与output dynamic parameter
- 识别顺序
  - 第一个 - :v1

- output dynamic parameter
- 在:sValue1中存储值

- 第二个 - :v2
  - input dynamic parameter
  - 使用:sValue2的值

variable_name

variable_name对应的类型应为character string

variable_name中定义的动态SQL语句应为有效语句

sql statement

sql statement中定义的动态SQL语句应为有效语句

说明

为了使用EXECUTE或cursorPREPARE statement_name FROM sql_string语句分析对应SQL语句

statement_name是embedded SQL源代码中给预编译器提示的statement标识符由于不是host variable因此不需要额外的类型或声明

详细内容参考**Embedded Dynamic SQL**

使用示例

PREPARE statement_name在Embedded SQL源代码中的使用示例如下

```
{
    ...
    sprintf( sUpdateSql, "UPDATE EMP SET sal = sal * :v1 WHERE JOB =
'SALES'");
    EXEC SQL PREPARE UPDATE_STMT FROM :sUpdateSql;
    if(sqlca.sqlcode != 0)
    {
        goto fail_exit;
    }

    sRatio = 1.1;
    EXEC SQL EXECUTE UPDATE_STMT USING :sRatio;
    if(sqlca.sqlcode != 0)
    {
        goto fail_exit;
    }
    ...
}
```

使用PREPARE statement_name的所有源代码可以在**Example Program**中查看

兼容性

Feature ID	说明	是否支持
B031	Basic Dynamic SQL	O
B034	Dynamic specification of cursor attributes	X

Table 10-4 标准SQL兼容性

参考

相关内容查看下文

- **EXECUTE statement_name**
- **DECLARE cursor_name**
- **EXECUTE IMMEDIATE 'sql_string'**
- **Embedded Dynamic SQL**

10.11 PURGE

功能

永久删除存储在回收站的对象

语句

```
<purge statement> :==
    PURGE <purge action>
    ;

<purge action> :==
     TABLE table_name
   | INDEX index_name
   | CONSTRAINT constraint_name
   | TABLESPACE tablespace_name [ USER user_name ]
   | RECYCLEBIN
   | USER_RECYCLEBIN
   | DBA_RECYCLEBIN
```

使用范围及访问权限

用户需要有以下权限中的一个才能执行<purge statement>语句

- 该表的所有者
- 对该表有CONTROL TABLE ON TABLE
- 对该表所属的schema有(DROP TABLE或者CONTROL SCHEMA) ON SCHEMA
- DROP ANY TABLE ON DATABASE

语句规则及参数

table_name

存储在回收站的对象名称或被删除的表的名称

schema_name.table_name相同被删除的表的名称可定义表所属的schema省略schema_name时使用执行语句的用户的默认schema名称

同时删除与表相关的索引与约束条件

index_name

存储在回收站的对象名称或被删除的索引的名称

与schema_name.index_name相同可定义索引所属的schema省略schema_name时使用执行语句的用户的默认schema名称

通过约束条件创建的key索引需要通过约束条件删除

constraint_name

存储在回收站的对象名或删除的约束条件的名称

tablespace_name

表空间的名称

指定USER时需要DROP ANY TABLE ON DATABASE权限

user_name

用户的名称

recyclebin

user_recyclebin的别名

user_recyclebin

删除用户拥有的所有回收站

dba_recyclebin

删除数据库的所有回收站

需要PURGE DBA_RECYCLEBIN ON DATABASE权限

说明

使用保存在回收站的对象名或删除的表名永久删除保管在回收站的对象

如果有与删除对象表相同名称的表时删除最久的对象

可在用户拥有的回收站对象中指定表空间后删除表空间所包含的对象此时指定用户则可以仅删除包含在该用户的表空间中的对象

提交事务前可回滚PURGE TABLEINDEXCONSTRAINT语句而无法回滚 PURGE TABLESPACERECYCLEBINDBA_RECYCLEBIN语句并自动提交执行语句的事务

使用示例

以下为删除保存在回收站中的表的示例

```
gSQL> SELECT OBJECT_NAME, ORIGINAL_NAME, OBJECT_TYPE FROM USER_RECYCLEBIN;


OBJECT_NAME                          ORIGINAL_NAME        OBJECT_TYPE
------------------------------------ -------------------- -----------
BIN$135B9908166111EA9C5C835D3E4BBBF7 T1                   TABLE
BIN$135B993A166111EA9C5C835D3E4BBBF7 T1_PRIMARY_KEY       CONSTRAINT
BIN$135B991C166111EA9C5C835D3E4BBBF7 T1_PRIMARY_KEY_INDEX INDEX
BIN$135B9926166111EA9C5C835D3E4BBBF7 T1_IDX1              INDEX


4 rows selected.
```

```
gSQL> PURGE TABLE t1;


Table purged.
```

以下为删除保存在回收站中的索引的示例

```
gSQL> SELECT OBJECT_NAME, ORIGINAL_NAME, OBJECT_TYPE FROM USER_RECYCLEBIN;


OBJECT_NAME                          ORIGINAL_NAME        OBJECT_TYPE
------------------------------------ -------------------- -----------
BIN$135B9908166111EA9C5C835D3E4BBBF7 T1                   TABLE
BIN$135B993A166111EA9C5C835D3E4BBBF7 T1_PRIMARY_KEY       CONSTRAINT
BIN$135B991C166111EA9C5C835D3E4BBBF7 T1_PRIMARY_KEY_INDEX INDEX
BIN$135B9926166111EA9C5C835D3E4BBBF7 T1_IDX1              INDEX


4 rows selected.


gSQL> PURGE INDEX t1_idx1;


Index purged.
```

以下为删除保存在回收站中的约束条件的示例

```
gSQL> SELECT OBJECT_NAME, ORIGINAL_NAME, OBJECT_TYPE FROM USER_RECYCLEBIN;


OBJECT_NAME                          ORIGINAL_NAME        OBJECT_TYPE
```

```
---------------------------------- -------------------- -----------
BIN$135B9908166111EA9C5C835D3E4BBBF7 T1                   TABLE
BIN$135B993A166111EA9C5C835D3E4BBBF7 T1_PRIMARY_KEY       CONSTRAINT
BIN$135B991C166111EA9C5C835D3E4BBBF7 T1_PRIMARY_KEY_INDEX INDEX

3 rows selected.

gSQL> PURGE CONSTRAINT t1_primary_key;

Constraints purged.
```

以下为删除保存在回收站的表空间中的对象的示例

```
gSQL> SELECT OBJECT_NAME, ORIGINAL_NAME, OBJECT_TYPE, TABLESPACE_NAME FROM
USER_RECYCLEBIN;

OBJECT_NAME                          ORIGINAL_NAME OBJECT_TYPE
TABLESPACE_NAME
---------------------------------- ------------- ----------- -----------
----
BIN$02C76B24166311EA9C5C835D3E4BBBF7 T1            TABLE
MEM_DATA_TBS

1 row selected.

gSQL> PURGE TABLESPACE MEM_DATA_TBS;
```

```
Tablespace purged.
```

以下为删除用户拥有的所有回收站的示例

```
gSQL> SELECT OBJECT_NAME, ORIGINAL_NAME, OBJECT_TYPE FROM USER_RECYCLEBIN;

OBJECT_NAME                          ORIGINAL_NAME        OBJECT_TYPE
------------------------------------ -------------------- -----------
BIN$64F6BFFC166311EA9C5C835D3E4BBBF7 T1                   TABLE
BIN$64F6C042166311EA9C5C835D3E4BBBF7 T1_PRIMARY_KEY       CONSTRAINT
BIN$64F6C010166311EA9C5C835D3E4BBBF7 T1_PRIMARY_KEY_INDEX INDEX
BIN$64F6C024166311EA9C5C835D3E4BBBF7 T1_IDX1              INDEX

4 rows selected.

gSQL> PURGE USER_RECYCLEBIN;

Recyclebin purged.
```

以下为删除系统的所有回收站的示例

```
gSQL> SELECT OWNER, OBJECT_NAME, ORIGINAL_NAME, OBJECT_TYPE FROM
USER_RECYCLEBIN;

OWNER OBJECT_NAME                          ORIGINAL_NAME
```

```
OBJECT_TYPE

----- ----------------------------------- ------------------- ----------

-

TEST  BIN$F0FB26F0166311EAA7C5D51B86D72AB6 T1                  TABLE

TEST  BIN$F0FB272C166311EAA7C5D51B86D72AB6 T1_PRIMARY_KEY      CONSTRAINT

TEST  BIN$F0FB2704166311EAA7C5D51B86D72AB6 T1_PRIMARY_KEY_INDEX INDEX

TEST  BIN$F0FB2718166311EAA7C5D51B86D72AB6 T1_IDX1             INDEX


4 rows selected.


gSQL> PURGE DBA_RECYCLEBIN;


DBA Recyclebin purged.
```

兼容性

标准SQL未定义<purge statement>

参考

相关内容参考下文

- 表回收站管理
- FLASHBACK TABLE

10.12 RELEASE SAVEPOINT savepoint_specifier

功能

删除保存点

语句

```
<release savepoint statement> ::=

    RELEASE SAVEPOINT savepoint_name

    ;
```

语句规则及参数

savepoint_name

保存点的名称必需要有

保存点名的长度应小于128 byte

说明

定义了多个保存点的情况下执行RELEASE SAVEPOINT savepoint_name时同时删除savepoint_name之后定义的保存点

使用示例

以下为删除保存点的示例

```
gSQL> RELEASE SAVEPOINT sp2;


Savepoint dropped.
```

兼容性

Feature ID	说明	是否支持
T271	Savepoints	O

Table 10-5 标准SQL兼容性

参考

相关内容参考下文

- **COMMIT**
- **ROLLBACK**
- **SAVEPOINT savepoint_specifier**

10.13 REVOKE privileges FROM

功能

收回向用户赋予的权限

语句

```
<revoke privilege statement> ::=
    REVOKE [ <revoke option extention> ] <privilege>
      FROM <grantee> [, ...]
      [ <revoke behavior> ]
    ;

<revoke option extention> ::=
      GRANT OPTION FOR

<revoke behavior> ::=
      RESTRICT
    | CASCADE
    | CASCADE CONSTRAINTS
```

语句规则及参数

<privilege>

要从revokee（被取消权限的用户）收回的权限

Revoker（执行语句的用户）需要满足以下条件中的一个

- Revoker向revokee赋予<privilege>时
  - 仅收回revoker向revokee赋予的<privilege>
- Revoker拥有ACCESS CONTROL ON DATABASE权限时
  - 取消其他grantor向revokee赋予的对应<privilege>

使用ALL [PRIVILEGES]时即使没有满足的<privilege>也执行成功

<privilege>类型相关内容参考**GRANT privileges TO**的 **<privilege>**

<grantee>

被取消权限的用户

- user_identifier
  - 取消对应用户的权限
- PUBLIC
  - 表示所有用户的authorization对象

GRANT OPTION FOR

删除权限中的WITH GRANT OPTION

同时删除dependent privilege的WITH GRANT OPTION

权限不变

<revoke behavior>

- Dependent privilege: 与通过WITH GRANT OPTION被赋予<privilege>的revokee向其他用户赋予的相同<privilege>
- RESTRICT
  - 如果有dependent privilege则不能revoke
- CASCADE
  - 同时revoke dependent privilege
- CASCADE CONSTRAINTS
  - 同时revoke dependent privilege
- 省略时默认值为CASCADE

说明

REVOKE privilege等Data Definition Language (DDL)语句在事务没有提交的情况下也可以ROLLBACK

执行如下DROP语句时不需要执行额外的REVOKE语句也会删除对应对象相关所有的权限信息

- SQL schema object相关DROP语句
  - **DROP TABLE**
  - **DROP VIEW**
  - **DROP SEQUENCE**
  - **ALTER TABLE name SET UNUSED COLUMN**
  - **DROP FUNCTION**
  - **DROP PROCEDURE**
  - **DROP PACKAGE**
- Non-schema object相关DROP语句
  - **DROP SCHEMA**
  - **DROP TABLESPACE**
  - **DROP USER**

使用示例

以下为REVOKE对t1表的多个权限的示例

```
gSQL> REVOKE INSERT, UPDATE, DELETE, LOCK, ALTER, INDEX ON t1 FROM u1;


Revoke succeeded.
```

以下为REVOKE表示所有用户的PUBLIC账号的SELECT ON TABLE t1权限的示例但仅收回PUBLIC账号的权限不收回向特定用户明确赋予的SELECT ON TABLE t1权限

```
gSQL> REVOKE SELECT ON t1 FROM PUBLIC;
```

```
Revoke succeeded.
```

以下为u1用户的SELECT ON TABLE t1权限不变仅收回可向其他用户赋予该权限的GRANT OPTION的示例

```
gSQL> REVOKE GRANT OPTION FOR SELECT ON t1 FROM u1;

Revoke succeeded.
```

以下为使用RESTRICT选项收回u1用户的权限并u1用户向其他用户赋予权限时报错的情况如果要同时删除dependent privilege则使用CASCADE选项

```
gSQL> REVOKE SELECT ON t1 FROM u1 RESTRICT;

ERR-2B000(16235): dependent privilege descriptors still exist

gSQL> REVOKE SELECT ON t1 FROM u1 CASCADE;

Revoke succeeded.
```

兼容性

标准SQL未定义以下privilege

- <database privilege>

- <tablespace privilege>
- <schema privilege>

与标准SQL的<revoke behavior>有以下区别

- 标准SQL的默认值为RESTRICT
- 标准SQL没有CASCADE CONSTRAINTS

Feature ID	说明	是否支持
T311	Basic Roles	X
F034	Extended REVOKE statement	X
S081	Subtables	X

Table 10-6 标准SQL兼容性

参考

相关内容参考下文

- **GRANT privileges TO**
- **<database privilege>**
- **<tablespace privilege>**
- **<schema priviliege>**
- **<table privilege>**

- **Column privilege**
- **<sequence privilege>**

10.14 ROLLBACK

功能

取消事务或保存点以后的操作

语句

```
<rollback statement> ::=
    ROLLBACK [ WORK ] [ <rollback force clause> | <savepoint clause> ]
    ;

<rollback force clause> ::=
    FORCE 'xid_string' [ COMMENT 'comment_string' ]

<savepoint clause> ::=
    TO SAVEPOINT savepoint_name
```

语句规则及参数

WORK

不影响操作的保留字

<rollback force clause>

用于手动回滚分布式事务

- FORCE 'xid_string'
  - 回滚属于'xid_string'的分布式事务
  - 'xid_string'由'format_id.transaction_id.branch_id'构成
- COMMENT 'comment_string'
  - 回滚分布式事务时指定事务的注释

<savepoint clause>

定义当前事务的回滚范围

- 未指定时
  - 取消当前事务的所有操作
  - 终止事务
  - 删除所有保存点
  - 解除所有锁
- TO SAVEPOINT savepoint_name
  - 取消当前事务的savepoint_name之后的操作
  - 不终止事务
  - 删除savepoint_name之后的保存点
  - 解除savepoint_name以后获取的锁

说明

ROLLBACK语句回滚事务内执行的以下语句

- Data Manipulation Language (DML)语句
  - 变更数据的INSERTUPDATEDELETE等语句
- Data Definition Language (DDL)语句
  - 变更对象的结构和定义的CREATEDROPALTERTRUNCATEGRANTREVOKE等语句

也有例外情况DDL中使用操作系统资源或变更数据类型的以下语句不会被回滚执行语句时自动提交

- **CREATE TABLESPACE**
- **DROP TABLESPACE**
- **ALTER TABLESPACE**
- ALTER TABLE .. ALTER COLUMN .. SET DATA TYPE: **<alter column data type clause>**

使用示例

以下为对INSERT语句执行ROLLBACK的示例

```
gSQL> INSERT INTO t1 VALUES ( 1, 'anonymous' );


1 row created.
```

```
gSQL> SELECT * FROM t1;


ID DATA
-- ---------
 1 anonymous


1 row selected.


gSQL> ROLLBACK;


Rollback complete.


gSQL> SELECT * FROM t1;


no rows selected.
```

以下为执行DROP TABLE语句后将其ROLLBACK的示例

```
gSQL> DROP TABLE t1;


Table dropped.


gSQL> SELECT * FROM t1;


ERR-42000(16040): table or view does not exist :
SELECT * FROM t1
```

```
              *
ERROR at line 1:


gSQL> ROLLBACK;


Rollback complete.


gSQL> SELECT * FROM t1;


ID DATA
-- ---------
 1 anonymous


1 row selected.
```

兼容性

Feature ID	说明	是否支持
T271	Savepoints	O
T261	Chained transactions	X

Table 10-7 标准SQL兼容性

参考

相关内容参考下文

- **COMMIT**
- **SAVEPOINT savepoint_specifier**

10.15 SAVEPOINT savepoint_specifier

功能

定义保存点

语句

```
<savepoint statement> ::=

    SAVEPOINT savepoint_name

    ;
```

语句规则及参数

savepoint_name

保存点的名称

与原有的保存点名称相同时则删除原有的保存点

保存点名称的长度应小于128 byte

说明

定义的保存点用于ROLLBACK TO SAVEPOINT语句(参考**ROLLBACK**)执行到对应保存点的DMLDDL语句被撤回同时解除该语句获取的lock

当提交或回滚事务时自动删除定义的保存点可通过**RELEASE SAVEPOINT savepoint_specifier**语句删除定义的保存点

使用示例

以下为定义保存点并执行ROLLBACK TO SAVEPOINT语句的示例

```
gSQL> SAVEPOINT sp1;

Savepoint created.

gSQL> INSERT INTO t1 VALUES ( 1, 'anonymous' );

1 row created.

gSQL> SAVEPOINT sp2;

Savepoint created.

gSQL> INSERT INTO t1 VALUES ( 2, 'someone' );
```

```
1 row created.

gSQL> SAVEPOINT sp3;

Savepoint created.

gSQL> INSERT INTO t1 VALUES ( 3, 'anyone' );

1 row created.

gSQL> SELECT * FROM t1;

ID DATA
-- ---------
 1 anonymous
 2 someone
 3 anyone

3 rows selected.

gSQL> ROLLBACK TO SAVEPOINT sp3;

Rollback complete.
```

```
gSQL> SELECT * FROM t1;


ID DATA
-- ---------
 1 anonymous
 2 someone


2 rows selected.


gSQL> ROLLBACK TO SAVEPOINT sp2;


Rollback complete.


gSQL> SELECT * FROM t1;


ID DATA
-- ---------
 1 anonymous


1 row selected.


gSQL> ROLLBACK TO SAVEPOINT sp1;


Rollback complete.
```

```
gSQL> SELECT * FROM t1;


no rows selected.
```

兼容性

Feature ID	说明	是否支持
T271	Savepoints	O

Table 10-8 标准SQL兼容性

参考

相关内容参考下文

- **COMMIT**
- **ROLLBACK**
- **RELEASE SAVEPOINT savepoint_specifier**

10.16 SELECT

query expression

功能

在一个以上的表或视图中检索row

语句

```
<query expression> ::=
    [ <with clause> ] <query expression body> [ <order by clause> ]
[ <offset limit clause> ]


<query expression body> ::=
      <query term>
    | <set operator>


<query term> ::=
      <query specification>
    | <left paren> <query expression body> [ <order by clause> ] [ <offset
limit clause> ] <right paren>
```

使用范围及访问权限

用户需要对语句中使用的所有表具有以下权限中的一个才能执行<query expression>语句

- 对表的column中用于语句的所有column有SELECT(columns) ON TABLE
- 对表有(SELECT或CONTROL TABLE) ON TABLE
- 对表所在的SCHEMA有(SELECT TABLE或CONTROL SCHEMA) ON SCHEMA
- SELECT ANY TABLE ON DATABASE

语句规则及参数

<with clause>

<with clause>定义临时结果集并可参考其结果集

详细内容参考**with clause**

<set operator>

执行子查询之间的集合运算

详细内容参考**set operator**

<query specification>

定义一个子查询

详细内容参考**query specification**

<order by clause>

描述检索结果的排列信息

详细内容参考**order by clause**

<offset limit clause>

描述检索结果集合中要跳过的row数量与要获取的row数量

详细内容参考**offset limit clause**

说明

以SELECT语句描述query

可以省略<with clause><order by clause><offset limit clause>

可以使用<set operator>拥有两个以上的子查询

使用示例

以下为SELECT语句示例

```
gSQL> SELECT s_name, s_nation FROM supplier;


S_NAME                    S_NATION

------------------------- -------------

Supplier#1                FRANCE

Supplier#2                KOREA
```

```
Supplier#3                GERMANY

Supplier#4                UNITED STATES

Supplier#5                CANADA


5 rows selected.
```

以下为使用<order by clause>的SELECT语句示例

```
gSQL> SELECT s_name, s_nation FROM supplier ORDER BY s_name DESC;


S_NAME                    S_NATION

------------------------- -------------

Supplier#5                CANADA

Supplier#4                UNITED STATES

Supplier#3                GERMANY

Supplier#2                KOREA

Supplier#1                FRANCE


5 rows selected.
```

以下为使用<offset limit clause>的SELECT语句示例

```
gSQL> SELECT s_name, s_nation FROM supplier OFFSET 1;


S_NAME                    S_NATION

------------------------- -------------
```

```
Supplier#2                KOREA

Supplier#3                GERMANY

Supplier#4                UNITED STATES

Supplier#5                CANADA


4 rows selected.


gSQL> SELECT s_name, s_nation FROM supplier LIMIT 1;


S_NAME                    S_NATION

------------------------- --------

Supplier#1                FRANCE


1 row selected.
```

以下为使用<order by clause>与<offset limit clause>的SELECT语句示例

```
gSQL> SELECT s_name, s_nation FROM supplier ORDER BY s_name DESC OFFSET 3

LIMIT 1;


S_NAME                    S_NATION

------------------------- --------

Supplier#2                KOREA


1 row selected.
```

以下为使用<with clause>的SELECT语句示例

```
* Non Recursive CTE


gSQL>
WITH revenue ( supplier_no, total_revenue ) AS
      (
            SELECT
                  l_suppkey,
                  SUM(l_extendedprice * (1 - l_discount))
              FROM lineitem
             WHERE l_shipdate >= DATE '1996-01-01'
               AND l_shipdate < DATE '1996-01-01' + INTERVAL '3' MONTH
             GROUP BY
                  l_suppkey
      )
select
       s_suppkey,
       s_name,
       s_address,
       s_phone,
       ROUND( total_revenue, 2 ) as total_revenue
  from
       supplier,
       revenue
```

```
 where
       s_suppkey = supplier_no
   and total_revenue = (
                           select
                                  max(total_revenue)
                             from
                                  revenue
                       )
order by
      s_suppkey;


S_SUPPKEY S_NAME                    S_ADDRESS         S_PHONE
TOTAL_REVENUE
--------- ------------------------- ----------------- --------------- ----
---------
     8449 Supplier#000008449        Wp34zim9qYFbVctdW 20-469-856-8873
1772627.21


1 row selected.


* Recursive CTE


gSQL>
WITH GenerateRecord ( c1, c2 ) AS
     (
```

```
          SELECT 1, 11
            FROM dual
          UNION ALL
          SELECT c1 + 1, c2 + 1
            FROM GenerateRecord
           WHERE c1 < 10
     )
SELECT c1, c2 FROM GenerateRecord;


C1 C2
-- --
 1 11
 2 12
 3 13
 4 14
 5 15
 6 16
 7 17
 8 18
 9 19
10 20


10 rows selected.
```

兼容性

Feature ID	说明	是否支持
T121	WITH (excluding RECURSIVE) in query expression	O
T122	WITH (excluding RECURSIVE) in subquery	O
T131	Recursive query	O
T132	Recursive query in subquery	O
F661	Simple tables	O
F302	INTERSECT table operator	O
F301	CORRESPONDING in query expressions	X
T551	Optional key words for default syntax	O
F304	EXCEPT ALL table operator	O
F850	Top-level <order by clause>in <query expression>	O
F851	<order by clause>in subqueries	O
F855	Nested <order by clause>in <query expression>	O
F856	Nested <fetch first clause>in <query expression>	O
F857	Top-level <fetch first clause>in <query expression>	O
F858	<fetch first clause>in subqueries	O
F860	dynamic <fetch first row count>in <fetch first clause>	X

Feature ID	说明	是否支持
F861	Top-level <result offset clause>in <query expression>	O
F862	<result offset clause>in subqueries	O
F863	Nested <result offset clause>in <query expression>	O
F865	dynamic <offset row count>in <result offset clause>	X
F866	FETCH FIRST clause: PERCENT option	X
F867	FETCH FIRST clause: WITH TIES option	X

Table 10-9 标准SQL兼容性

with clause

功能

<with clause>定义临时结果集并可参考其结果集

其在SELECT语句中被定义并参照是赋予名称的临时结果集叫做Common Table Expression (CTE)

语句

```
<with clause> ::=
    WITH <with list>

<with list> ::=
    <with list element> [ { <comma> <with list element> }... ]

<with list element> ::=
    <query name> [ <left paren> <with column list> <right paren> ]
        AS <table subquery> [ <search or cycle clause> ]

<with column list> ::=
    <column name list>

<search or cycle clause> ::=
    <search clause>
  | <cycle clause>
```

```
  | <search clause> <cycle clause>


<search clause> ::=

    SEARCH <recursive search order> SET <sequence column>


<recursive search order> ::=

    DEPTH FIRST BY <ordering column list>

  | BREADTH FIRST BY <ordering column list>


<ordering column list> ::=

    <ordering column> [ { <comma> <ordering column> }... ]


<ordering column> ::=

    <column name> [ ASC | DESC ] [ NULLS FIRST | NULLS LAST ]


<sequence column> ::=

    <column name>


<cycle clause> ::=

    CYCLE <cycle column list> SET <cycle mark column> TO <cycle mark
value>

        DEFAULT <non-cycle mark value>


<cycle column list> ::=

    <cycle column> [ { <comma> <cycle column> }... ]
```

```
<cycle column> ::=
    <column name>

<cycle mark column> ::=
    <column name>

<cycle mark value> ::=
    <value expression>

<non-cycle mark value> ::=
    <value expression>
```

使用范围及访问权限

在<query expression>语句支持用户需要满足<query expression>的访问权限才能执行

详细内容参考**query expression**

语句规则及参数

<with list>

可定义多个<with list element>

<with list element>

定义描述的<query name>的临时结果集

<with list element>叫做Common Table Expression (CTE)

CTE分为recursive CTE和non-recursive CTE

详细内容参考**说明**部分

- CTE参照
  - 根据描述的顺序限制CTE参照
  - 可参照当前CTE之前描述的CTE
  - 无法参照当前CTE之后描述的CTE
- non-recursive CTE
  - 在<with list element>可参照当前CTE之前描述的CTE
- recursive CTE
  - 在<with list element>可参照当前CTE之前描述的CTE或self CTE
  - Self-reference CTE在CTE内仅允许一次

```
--# success : non recursive CTE
WITH CTE_1( c1 ) AS
    (
        SELECT i1
          FROM t1
    ),
    CTE_2( c2 ) AS
    (
        SELECT c1
```

```
          FROM CTE_1        ❶ 参照上一行CTE
    )
SELECT c2 FROM CTE_2;


--# error : non recursive CTE
WITH CTE_1( c1 ) AS
    (
         SELECT c2
           FROM CTE_2       ❷ 参照下一行CTE
    ),
    CTE_2( c2 ) AS
    (
         SELECT i1
           FROM t1
    )
SELECT c1 FROM CTE_1;
```

```
--# success : recursive CTE
WITH CTE_1( c1 ) AS
     (
          SELECT i1
            FROM t1
     ),
     CTE_2( c2 ) AS
```

```
    (
         SELECT c1
           FROM CTE_1          ❶ 参照上一行CTE
    ),
    CTE_3( c3 ) AS
    (
         SELECT 1
           FROM CTE_1
         UNION ALL
         SELECT 1
          FROM CTE_2, CTE_3     ❷ 参照上一行CTE或self CTE
    )
SELECT c3 FROM CTE_3;


--# error : recursive CTE
WITH CTE_RECURSIVE( c1 ) AS
   (
         SELECT i1
           FROM t1
         WHERE i1 IS NULL
         UNION ALL
         SELECT 1
           FROM CTE_RECURSIVE A, CTE_RECURSIVE B    ❸ self-reference CTE
仅允许一次
         WHERE 1 = 0
```

```
    )
SELECT c1 FROM CTE_RECURSIVE;
```

<query name>

<query name>在WITH clause中不可以重复

<with column list>

Recursive CTE不可以省略<with column list>

<search clause>

- Non-recursive CTE
  - 无法描述<search clause>
- Recursive CTE
  - 描述CTE结果record的排列顺序
- <recursive search order>
  - DEPTH FIRST BY
    - 返回sibling rows之前返回child rows
  - BREADTH FIRST BY
    - 返回child rows之前返回sibling rows
- <ordering column list>
  - 指定column list的排列方式
  - 排列顺序
    - ASC
    - DESC

        - 未指定时默认值为ASC
    - Null ordering
        - NULLS FIRST
        - NULLS LAST
        - 未指定时默认值为NULLS LAST
    - 要描述<with list element>中声明的<with column list>
    - 不支持LONG type (LONG VARCHAR, LONG VARBINARY)
- <sequence column>
    - 存储CTE结果record的顺序
    - <column name>不得与以下项目重复
        - <with list element>的<with column list>中声明的column name
        - <cycle clause>的<cycle column list>中声明的column name

<cycle clause>

- Non-recursive CTE
    - 无法描述<cycle clause>
- Recursive CTE
    - 省略<cycle clause>时产生cycle时返回error

根据cycle产生与否在<cycle mark column>存储<cycle mark value>或<non-cycle mark value>

- <cycle column list>
    - 描述在<with list element>声明的<with column list>
- <cycle mark column>
    - <column name>不得与以下项目重复

    - <with list element>的<with column list>中声明的column name
    - <search clause>的<sequence column>中声明的column name
- <cycle mark value>或<non-cycle mark value>
    - 仅可描述1byte字符

说明

<with clause>定义临时结果集并可参照其结果集

在SELECT语句内定义并参照是赋予名称的临时结果集

其叫做Common Table Expression(CTE)

CTE分为recursive CTE和non-recursive CTE

- Recursive CTE：参照CTE中当前定义的CTE（self-reference CTE）的情况

```
WITH RECURSIVE_CTE ( c1 ) AS
     (
          SELECT 1
            FROM dual
          UNION ALL
          SELECT c1 + 1
            FROM RECURSIVE_CTE
           WHERE c1 < 10
     )
SELECT c1 FROM RECURSIVE_CTE;
```

- Non-recursive CTE：不是recursive CTE的情况

```
WITH NON_RECURSIVE_CTE ( c1 ) AS
     (
          SELECT i1
            FROM t1
          UNION ALL
          SELECT i1
            FROM t2
     )
SELECT c1 FROM NON_RECURSIVE_CTE;
```

<with clause>可在SELECT, INSERT, UPDATE, DELETE, CREATE TABLE AS SELECT, CREATE VIEW语句中描述

<with list element>

定义描述的<query name>的临时结果集

<with list element>叫做Common Table Expression（CTE）

CTE分为recursive CTE和non-recursive CTE

- Recursive CTE
  - 参照CTE内当前定义的CTE（self-reference CTE）的情况
  - 包含self-reference CTE的query block叫做recursive member query
  - 非recursive member query的其他query block叫做anchor member query
  - Recursive member query和anchor member query要以UNION ALL构成
  - Recursive member query只能描述一个
- Non-recursive CTE：不是Recursive CTE的情况

```
WITH CTE_RECURSIVE( c1, c2 ) AS
    (
        SELECT i1, i2                          ❶ Anchor member query
          FROM t1
         WHERE i2 IS NULL
        UNION ALL
        SELECT i1, i2                          ❷ Recursive member query
          FROM CTE_RECURSIVE, t1               ❸ Self reference
         WHERE CTE_RECURSIVE.c1 = t1.i2
    )
SELECT c1, c2 FROM CTE_RECURSIVE;
```

- Recursive member query不得包含以下项目
  - GROUP BY或DISTINCT clause
    - (X) SELECT i1, i2 FROM CTE_RECURSIVE, t1 WHERE CTE_RECURSIVE.c1 = t1.i2 GROUP BY i1, i2
    - (X) SELECT DISTINCT i1, i2 FROM CTE_RECURSIVE, t1 WHERE CTE_RECURSIVE.c1 = t1.i2
  - 在LEFTRIGHTOUTER JOIN的 inner part参照CTE
    - (X) SELECT i1, i2 FROM t1 LEFT OUTER JOIN CTE_RECURSIVE ON t1.i2 = CTE_RECURSIVE.c1
  - Aggregation function
    - (X) SELECT MAX(i1), MAX(i2) FROM CTE_RECURSIVE, t1 WHERE CTE_RECURSIVE.c1 = t1.i2
  - 包含self-reference CTE的subquery

- (X) SELECT i1, i2 FROM (SELECT * FROM CTE_RECURSIVE) cte, t1 WHERE cte.c1 = t1.i2

<search clause>

描述CTE结果record的排列顺序

sibling rows通过<ordering column list>排列对排列的record指定sibling rows和child rows的返回顺序

<sequence column>存储结果record的顺序

- DEPTH FIRST BY
  - 返回sibling rows之前返回child rows
- BREADTH FIRST BY
  - 返回child rows之前返回sibling rows

```
gSQL>
SELECT * FROM t1;


I1  I2
--- ---
A   ---
AA  A
AB  A
AC  A
AAX AA
ABX AB
```

```
ACX AC


7 rows selected.


* SEARCH BREADTH FIRST BY


gSQL>
WITH w1( w_i1, w_i2 ) AS
    (
          SELECT i1, i2
            FROM t1
           WHERE i1 = 'A'
          UNION ALL
          SELECT i1, i2
            FROM w1, t1
           WHERE w_i1 = i2
    ) SEARCH BREADTH FIRST BY w_i1, w_i2 SET w_seq
SELECT w_i1, w_i2, w_seq
 FROM w1;


W_I1 W_I2 W_SEQ
---- ---- -----
A    ---      1
AA   A        2
AB   A        3
```

```
AC   A        4

AAX  AA       5

ABX  AB       6

ACX  AC       7


7 rows selected.


* SEARCH DEPTH FIRST BY


gSQL>
WITH w1( w_i1, w_i2 ) AS
    (
         SELECT i1, i2
           FROM t1
          WHERE i1 = 'A'
         UNION ALL
         SELECT i1, i2
           FROM w1, t1
          WHERE w_i1 = i2
    ) SEARCH DEPTH FIRST BY w_i1, w_i2 SET w_seq
SELECT w_i1, w_i2, w_seq
  FROM w1;


W_I1 W_I2 W_SEQ
---- ---- -----
```

```
A    ---      1

AA   A        2

AAX  AA       3

AB   A        4

ABX  AB       5

AC   A        6

ACX  AC       7


7 rows selected.
```

<cycle clause>

未描述<cycle clause> 语句时产生cycle时报错

<cycle column list>用于检查cycle

根据cycle的产生与否在<cycle mark column>存储<cycle mark value>或<no-cycle mark value>

<cycle mark value>或<non-cycle mark value>中仅可描述1 byte字符

在产生cycle的record的<cycle mark column>存储<cycle mark value>此时不再有recursion返回截

止产生cycle的record

不产生cycle的sibling rows继续进行recursion

```
gSQL>

SELECT * FROM t1;


I1  I2

--- ---
```

```
A   ---
AA  A
AB  A
AC  A
AA  AA
AAX AA
ABX AB
ACX AC


8 rows selected.
```

- 产生cycle并未描述cycle clause的情况

```
gSQL>
WITH w1( w_i1, w_i2 ) AS
     (
          SELECT i1, i2
            FROM t1
           WHERE i1 = 'A'
          UNION ALL
          SELECT i1, i2
            FROM w1, t1
           WHERE w_i1 = i2
     )
SELECT w_i1, w_i2
  FROM w1;
```

```
ERR-42000(16511): cycle detected while executing recursive WITH query
```

- 产生cycle并描述cycle clause的情况

```
gSQL>
WITH w1( w_i1, w_i2 ) AS
     (
          SELECT i1, i2
            FROM t1
           WHERE i1 = 'A'
          UNION ALL
          SELECT i1, i2
            FROM w1, t1
           WHERE w_i1 = i2
     ) CYCLE w_i1, w_i2 SET c_cycle TO 'T' DEFAULT 'F'
SELECT w_i1, w_i2, c_cycle
  FROM w1;


W_I1 W_I2 C_CYCLE
---- ---- -------
A    ---  F
AC   A    F
AB   A    F
AA   A    F
ACX  AC   F
```

```
ABX   AB    F

AAX   AA    F

AA    AA    F

AAX   AA    F

AA    AA    T


10 rows selected.
```

使用示例

以下为使用WITH子句的SELECT语句的示例

- Non-recursive CTE

```
gSQL>
WITH revenue ( supplier_no, total_revenue ) AS
      (
            SELECT
                  l_suppkey,
                  SUM(l_extendedprice * (1 - l_discount))
              FROM lineitem
             WHERE l_shipdate >= DATE '1996-01-01'
               AND l_shipdate < DATE '1996-01-01' + INTERVAL '3' MONTH
             GROUP BY
                  l_suppkey
      )
```

```
select
       s_suppkey,
       s_name,
       s_address,
       s_phone,
       ROUND( total_revenue, 2 ) as total_revenue
  from
       supplier,
       revenue
 where
       s_suppkey = supplier_no
   and total_revenue = (
                           select
                                  max(total_revenue)
                             from
                                  revenue
                       )
order by
      s_suppkey;


S_SUPPKEY S_NAME                    S_ADDRESS         S_PHONE
TOTAL_REVENUE
--------- ------------------------- ----------------- --------------- ----
---------
     8449 Supplier#000008449        Wp34zim9qYFbVctdW 20-469-856-8873
```

```
1772627.21


1 row selected.
```

- Recursive CTE

```
gSQL>
WITH GenerateRecord ( c1, c2 ) AS
     (
          SELECT 1, 11
            FROM dual
          UNION ALL
          SELECT c1 + 1, c2 + 1
            FROM GenerateRecord
           WHERE c1 < 10
     )
SELECT c1, c2 FROM GenerateRecord;


C1 C2
-- --
 1 11
 2 12
 3 13
 4 14
 5 15
 6 16
```

```
 7 17
 8 18
 9 19
10 20


10 rows selected.
```

以下为要用于WITH子句示例的emp表的record检索结果

```
gSQL>
SELECT * FROM emp;


NAME    MGR
------- -------
Kelly   null
Bill    Kelly
Jackson Kelly
Joe     Kelly
Scott   Bill
Larry   Bill
Paul    Jackson
Bill    Bill


8 rows selected.
```

以下为使用SEARCH BREADTH FIRST BY的示例

- 产生cycle

```
gSQL>
WITH w_emp( w_name, w_mgr ) AS
     (
         SELECT name, mgr
           FROM emp
          WHERE mgr IS NULL
         UNION ALL
         SELECT name, mgr
           FROM emp, w_emp
          WHERE mgr = w_emp.w_name
     ) SEARCH BREADTH FIRST BY w_name SET w_seq
SELECT w_name, w_mgr, w_seq
  FROM w_emp;


ERR-42000(16511): cycle detected while executing recursive WITH query
```

- 在产生cycle的上一个语句描述cycle clause并检索

```
gSQL>
WITH w_emp( w_name, w_mgr ) AS
     (
         SELECT name, mgr
           FROM emp
          WHERE mgr IS NULL
```

```
         UNION ALL
         SELECT name, mgr
           FROM emp, w_emp
          WHERE mgr = w_emp.w_name
     ) SEARCH BREADTH FIRST BY w_name SET w_seq
       CYCLE w_name SET w_cycle TO 'T' DEFAULT 'F'
SELECT w_name, w_mgr, w_seq, w_cycle
  FROM w_emp;

W_NAME  W_MGR   W_SEQ W_CYCLE
------- ------- ----- -------
Kelly   null        1 F
Bill    Kelly       2 F
Jackson Kelly       3 F
Joe     Kelly       4 F
Bill    Bill        5 T
Larry   Bill        6 F
Paul    Jackson     7 F
Scott   Bill        8 F

8 rows selected.
```

以下为使用SEARCH DEPTH FIRST BY的示例

```
gSQL>
WITH w_emp( w_name, w_mgr ) AS
```

```
    (
          SELECT name, mgr
            FROM emp
           WHERE mgr IS NULL
          UNION ALL
          SELECT name, mgr
            FROM emp, w_emp
           WHERE mgr = w_emp.w_name
    ) SEARCH DEPTH FIRST BY w_name SET w_seq
      CYCLE w_name SET w_cycle TO 'T' DEFAULT 'F'
SELECT w_name, w_mgr, w_seq, w_cycle
  FROM w_emp;

W_NAME  W_MGR   W_SEQ W_CYCLE
------- ------- ----- -------
Kelly   null        1 F
Bill    Kelly       2 F
Bill    Bill        3 T
Larry   Bill        4 F
Scott   Bill        5 F
Jackson Kelly       6 F
Paul    Jackson     7 F
Joe     Kelly       8 F

8 rows selected.
```

以下为在CREATE TABLE AS SELECT语句使用with clause的示例

```
gSQL>
CREATE TABLE new_emp AS
WITH w_emp( w_name, w_mgr ) AS
     (
          SELECT name, mgr
            FROM emp
           WHERE mgr IS NULL
          UNION ALL
          SELECT name, mgr
            FROM emp, w_emp
           WHERE mgr = w_emp.w_name
     ) SEARCH BREADTH FIRST BY w_name SET w_seq
       CYCLE w_name SET w_cycle TO 'T' DEFAULT 'F'
SELECT w_name, w_mgr, w_seq, w_cycle
  FROM w_emp;


Table created.
```

以下为在INSERT语句使用with clasuse的示例

```
gSQL>
INSERT INTO new_emp
WITH w_emp( w_name, w_mgr ) AS
     (
```

```
          SELECT name, mgr
            FROM emp
           WHERE mgr = 'Bill'
          UNION ALL
          SELECT name, mgr
            FROM emp, w_emp
           WHERE mgr = w_emp.w_name
     ) SEARCH BREADTH FIRST BY w_name SET w_seq
       CYCLE w_name SET w_cycle TO 'T' DEFAULT 'F'
SELECT w_name, w_mgr, w_seq, w_cycle
  FROM w_emp;


6 rows created.
```

以下为在UPDATE 语句使用with clasuse的示例

```
gSQL>
UPDATE new_emp SET w_name = NULL
 WHERE ( w_name, w_mgr )
       IN ( WITH w_emp( w_name, w_mgr ) AS
                (
                    SELECT name, mgr
                      FROM emp
                     WHERE mgr = 'Bill'
                    UNION ALL
                    SELECT name, mgr
```

```
                    FROM emp, w_emp
                   WHERE mgr = w_emp.w_name
                ) SEARCH BREADTH FIRST BY w_name SET w_seq
                  CYCLE w_name SET w_cycle TO 'T' DEFAULT 'F'
            SELECT w_name, w_mgr
              FROM w_emp );


9 rows updated.
```

以下为在 DELETE 语句使用with clause的示例

```
gSQL>
DELETE FROM new_emp
WHERE ( w_mgr )
      IN ( WITH w_emp( w_name, w_mgr ) AS
                (
                    SELECT name, mgr
                      FROM emp
                     WHERE mgr = 'Bill'
                    UNION ALL
                    SELECT name, mgr
                      FROM emp, w_emp
                     WHERE mgr = w_emp.w_name
                ) SEARCH BREADTH FIRST BY w_name SET w_seq
                  CYCLE w_name SET w_cycle TO 'T' DEFAULT 'F'
           SELECT w_mgr
```

```
            FROM w_emp );


9 rows deleted.
```

以下为在CREATE VIEW 语句使用with clasuse的示例

```
gSQL>
CREATE VIEW v_emp AS
WITH w_emp( w_name, w_mgr ) AS
     (
          SELECT name, mgr
            FROM emp
           WHERE mgr IS NULL
          UNION ALL
          SELECT name, mgr
            FROM emp, w_emp
           WHERE mgr = w_emp.w_name
     ) SEARCH BREADTH FIRST BY w_name SET w_seq
       CYCLE w_name SET w_cycle TO 'T' DEFAULT 'F'
SELECT w_name, w_mgr, w_seq, w_cycle
  FROM w_emp;


View created.
```

query specification

功能

描述从<table expression>的结果衍生的table

语句

```
<query specification> ::=
    SELECT [ <hint clause> ] [ <set quantifier> ] <select list> <table
expression>


<set quantifier> ::=
      ALL
    | DISTINCT


<table expression> ::=
      <from clause> [ <where clause> ] [ <hierarchical query clause> ]
[ <group by clause> ] [ <having clause> ] [ <window clause> ]
```

使用范围及访问权限

用户需要满足以下条件中的一个才能执行<query expression>语句

- 表的所有者

- 对表有SELECT权限
- 对表所在的SCHEMA有SELECT TABLECONTROL TABLECONTROL权限中的一个
- 对数据库有SELECT TABLE权限

语句规则及参数

<hint clause>

描述执行语句所需的hint

详细内容参考**SQL Hint**

<set quantifier>

描述是否重复删除查询结果

省略时运行方式与ALL相同

<select list>

描述要从查询结果中搜索的column

详细内容参考**select list**

<from clause>

描述要查询的表

详细内容参考**from clause**

<where clause>

描述查询条件

详细内容参考**where clause**

<hierarchical query clause>

描述使其按照层次结构搜索层次模式数据

详细内容参考**hierarchical query clause**

<group by clause>

描述检索结果的grouping

详细内容参考**group by clause**

<having clause>

描述grouping的结果的条件

详细内容参考**having clause**

<window clause>

描述window function的执行范围

详细内容参考**window clause**

说明

<hint clause>

<hint clause>是用户为了向optimizer直接指示SQL语句执行方法而使用的comment

SUNDB的查询优化器优先应用用户指定的<hint clause>

如果不能应用再根据成本计算选择最佳执行计划

即使<hint clause>语句报错SUNDB默认设置忽略此错误如果想查看<hint clause>在语句上是否存在错误可以把**HINT_ERROR**参数设置为on并执行语句

<set quantifier>

<set quantifier>设置是否删除由<select list>的expression构成的结果集中的重复记录

- ALL：不删除结果集的重复记录
- DISTINCT：删除结果集的重复记录
- 省略时默认为ALL

<select list>

描述要从查询结果检索的column

此目录以逗号(,)列表区分

如果要描述<from clause>的所有Column则使用星号('*')

<from clause>

<from clause>描述要查询的表或视图

<where clause>

<where clause>描述查询条件以使在从<from clause>获得的结果集中仅返回满足条件的结果

<hierarchical query clause>

描述使其按照层次结构搜索层次模式数据

使用开始条件和下级连接条件将表的record以depth-first顺序的层次结构返回

<group by clause>

<group by clause>描述使用 <where clause>的结果集的grouping方法

描述<group by clause>时<select list>可以为如下表达式

- 常数
- 描述在group by的表达式
- 描述中group by的表达式的运算式
- 属于group的表达式的统计函数

<having clause>

<having clause>描述被grouping的结果集的查询条件

通常与<group by clause>一起使用

<window clause>

描述select list和order by clause中描述的window function的执行范围

使用示例

以下为使用<hint clause>的SELECT语句的示例

```
gSQL> SELECT /*+ INDEX_DESC(supplier, supplier_pk_index) */ s_name,
s_nation FROM supplier;

S_NAME                    S_NATION
------------------------- -------------
Supplier#5                CANADA
Supplier#4                UNITED STATES
Supplier#3                GERMANY
Supplier#2                KOREA
Supplier#1                FRANCE

5 rows selected.
```

以下为使用<set quantifier>的SELECT语句的示例

```
gSQL> SELECT ALL p_type FROM part;

P_TYPE
```

```
------

COPPER

NICKEL

STEEL

NICKEL

STEEL


5 rows selected.


gSQL> SELECT DISTINCT p_type FROM part;


P_TYPE

------

COPPER

STEEL

NICKEL


3 rows selected.
```

以下为使用<where clause>的SELECT语句的示例

```
gSQL> SELECT p_name, p_brand, p_type, p_size FROM part where p_size < 10;


P_NAME P_BRAND    P_TYPE P_SIZE

------ ---------- ------ ------

Part#1 Brand#1    COPPER      7
```

```
Part#2 Brand#1    NICKEL      1


2 rows selected.
```

以下为使用<hierarchical query clause>查询SELECT语句的层次结构数据的示例

```
gSQL>
SELECT *
  FROM emp
START WITH mgr IS NULL
CONNECT BY NOCYCLE mgr = PRIOR name
ORDER SIBLINGS BY name;


NAME    MGR
------- -------
Kelly   null
Bill    Kelly
Larry   Bill
Scott   Bill
Jackson Kelly
Paul    Jackson
Joe     Kelly


7 rows selected.
```

以下为使用<group by clause>的SELECT语句的示例

```
gSQL> SELECT ps_partkey, SUM(ps_availqty) FROM partsupp GROUP BY
ps_partkey;

PS_PARTKEY SUM(PS_AVAILQTY)
---------- ----------------
         1            11401
         2             8025
         3            13864
         4            11564
         5             8744

5 rows selected.
```

以下为使用<having clause>的SELECT语句的示例

```
gSQL> SELECT ps_partkey, SUM(ps_availqty) FROM partsupp GROUP BY
ps_partkey having SUM(ps_availqty) > 10000;

PS_PARTKEY SUM(PS_AVAILQTY)
---------- ----------------
         1            11401
         3            13864
         4            11564

3 rows selected.
```

以下为使用<window clause>的SELECT语句的示例

```
gSQL>
SELECT item_no,
       sales_date,
       sales,
       SUM( sales ) OVER W1 cumulative_sales,
       AVG( sales ) OVER w1 avg_sales
  FROM store
WINDOW w1 AS ( PARTITION BY item_no
               ORDER BY sales_date
               ROWS BETWEEN UNBOUNDED PRECEDING
                        AND CURRENT ROW );

ITEM_NO SALES_DATE SALES CUMULATIVE_SALES AVG_SALES
------- ---------- ----- ---------------- ---------
    100 2001-01-01   150              150       150
    100 2001-01-02   100              250       125
    100 2001-01-03   170              420       140
    100 2001-01-04    90              510     127.5
    100 2001-01-05   200              710       142
    235 2001-01-01    70               70        70
    235 2001-01-02   130              200       100
    235 2001-01-03   190              390       130
    235 2001-01-04   150              540       135
```

```
    235 2001-01-05    50              590       118


10 rows selected.
```

兼容性

Feature ID	说明	是否支持
F801	Full set function	X
T051	Row types	X
T301	Functional dependencies	X
T325	Qualified SQL parameter references	X
T053	Explicit aliases for all-fields reference	O
T285	Enhanced derived column names	O

Table 10-10 标准SQL兼容性

参考

相关内容参考**query expression**

select list

功能

描述在查询结果中检索的column

语句

```
<select list> ::=
      <asterisk>
    | <select sublist> [ { <comma> <select sublist> } ... ]

<select sublist> ::=
      <derived column>
    | <qualified asterisk>

<qualified asterisk> ::=
      <asterisked identifier chain> <period> <asterisk>

<asterisked identifier chain> ::=
    <asterisked identifier> [ { <period> <asterisked identifier> } ... ]

<derived column> ::=
    <value expression> [ <as clause> ]
```

```
<as clause> ::=

    [ AS ] <column name>
```

使用范围及访问权限

<select list>语句中有column或子查询时应满足如下

- 访问column的权限
- 访问子查询中的表以及column的权限

语句规则及参数

<select list>

拥有<asterisk>或<select sublist>

<asterisk>

- <asterisk>只能单独使用于<select list>
  - (O) SELECT * FROM t1;
  - (X) SELECT *, c1 FROM t1;

<select sublist>

- 拥有<derived column>或<qualified asterisk>
  - SELECT c1, c2 FROM t1;
  - SELECT t1.* FROM t1;

- <derived column>可使用AS变更输出名称AS可省略
  - SELECT c1 AS col1, c2 AS col2 AS FROM t1;
  - SELECT c1 col1, c2 col2 FROM t1;
- 描述两个以上的<select sublist>时以逗号（）区分各个<select sublist>
  - (O) SELECT c1, c2 FROM t1;
  - (O) SELECT c1, c2, t1.* FROM t1;
  - (X) SELECT c1 c2 FROM t1;
    - c2处理为ALIAS
  - (X) SELECT c1 c2 c3 FROM t1;

说明

<select list>

<select list>描述包含在结果集的Column

<asterisk>

<asterisk>将<from clause>中的所有column设置为select list

<select sublist>

<select sublist>拥有<derived column>或<qualified asterisk>

- <qualified asterisk>
  - 将属于特定表或视图的所有Column设置为select list
- <derived column>

- 可以描述Column或<value expression>
- 可使用<as clause>变更column name此时可省略AS
- <from clause>中有拥有相同column name的表时为了参考此column必须指定table name或table alias
  - SELECT t1.c1, t2.c1 FROM t1, t2;
  - SELECT a.c1, b.c1 FROM t1 a, t2 b;

描述两个以上的<select sublist>时必须要用逗号（）区分

select list中设置的名称

- <derived column>中指定<column name>时该名称设置为select list名
  - SELECT i1 AS name FROM t1;
- <derived column>中未指定<column name>时
  - <derived column>为single column reference时
    - single column的column name设置为select list名
    - SELECT i1 FROM t1;
  - <derived column>为非column的表达式时
    - 不设置select list
    - SELECT i1 + 100 FROM t1;
    - 用于CREATE TABLE AS SELECT语句时应描述column name
    - CREATE TABLE t2 AS SELECT i1 + 100 AS sum_i1 FROM t1;

使用示例

以下为使用<asterisk>的SELECT语句的示例

```
gSQL> SELECT * FROM supplier;


S_SUPPKEY S_NAME                    S_NATION      S_PHONE

--------- ------------------------- ------------- ---------------

        1 Supplier#1                FRANCE        27-918-335-1736

        2 Supplier#2                KOREA         15-679-861-2259

        3 Supplier#3                GERMANY       11-383-516-1199

        4 Supplier#4                UNITED STATES 25-843-787-7479

        5 Supplier#5                CANADA        21-151-690-3663


5 rows selected.
```

以下为使用<select sublist>的SELECT语句的示例

```
gSQL> SELECT revenue.* FROM revenue;


SUPPLIER_NO TOTAL_REVENUE

----------- -------------

          1      11978.64

          2       20321.5

          3      41844.68


3 rows selected.


gSQL> SELECT supplier_no suppno, total_revenue AS TOTAL FROM revenue;
```

```
SUPPNO    TOTAL

------ --------

     1 11978.64

     2  20321.5

     3 41844.68


3 rows selected.


gSQL> SELECT 1, revenue.*, CAST( total_revenue AS NATIVE_INTEGER ) TOTAL

FROM revenue;


1 SUPPLIER_NO TOTAL_REVENUE TOTAL

- ----------- ------------- -----

1           1      11978.64 11979

1           2       20321.5 20322

1           3      41844.68 41845


3 rows selected.
```

参考

相关内容参考**query specification**

from clause

功能

定义从一个或多个表衍生的表

语句

```
<from clause> ::=
    FROM <table reference list>

<table reference list> ::=
    <table reference> [ { , <table reference> } ... ]

<table reference> ::=
      <table factor>
    | <joined table>

<table factor> ::=
    <table primary>

<table primary> ::=
      <table name> [ <cluster domain> ] [ [ AS ] <correlation name> ]
    | <derived table> [ <cluster domain> ] [ [ AS ] <correlation name>
[ <left paren> <derived column list> <right paren> ] ]
```

```
    | <lateral derived table> [ <cluster domain> ] [ [ AS ] <correlation
name> [ <left paren> <derived column list> <right paren> ] ]
    | <table function derived table> [ [ AS ] <correlation name> ]
    | <parenthesized joined table>

<derived table> ::=
    <table subquery>

<lateral derived table> ::=
    LATERAL <table subquery>

<parenthesized joined table> ::=
      <left paren> <parenthesized joined table> <right paren>
    | <left paren> <joined table> <right paren>

<derived column list> ::=
    <column name list>

<cluster domain> ::=
    @ <cluster domain name>

<cluster domain name> ::=
      GLOBAL
    | LOCAL
    | LOCAL_OFFLINE
```

```
    | <identifier>


<table function derived table> ::=

      TABLE <left paren> <table function expression> <right paren>


<table function expression> ::=

      <table function name> <left paren> [ <table function argument
list> ] <right paren>


<table function argument list> ::=

      <value expression> [ <comma> ... ]
```

使用范围及访问权限

需要有访问<table reference list>中的表或视图的权限

语句规则及参数

<table reference list>

- <table reference list>可使用逗号(,)描述一个以上的表
- 描述2个以上的表时
    - 从左向右评估（evaluation）表
    - <select list>中使用*时从左侧表的Column到右侧表的column的顺序映射到<select list>

<table primary>

- 可使用<correlation name>定义别名(alias name)
  - SELECT * FROM t1 AS a, t2 AS b;
  - SELECT * FROM (SELECT i1 FROM t1) AS a;
- <derived table>中 即<table subquery>
  - 可使用<correlation name>描述别名(alias name)
    - SELECT * FROM (SELECT i1, i2, i3 FROM t1) AS a;
  - 可描述<derived column list>
    - SELECT * FROM (SELECT i1, i2, i3 FROM t1) AS a(col1, col2, col3);
    - <derived column list>的<column name>数量应与<table subquery>中描述的<select list>的目标数量相同
    - 与<table subquery>中描述的<select list>的目标按照顺序1:1映射
    - 为了参照<derived table>内<table subquery>的<select list>必须使用在<derived column list>中描述的<column name>

```
SELECT col1, col2
FROM ( SELECT i1, i2 FROM t1 ) AS a( col1, col2 )
WHERE col1 = 1 AND col2 = 1;
```

- <lateral derived table>
  - 如果在<table subquery>前指定LATERAL则变为lateral inline view
  - lateral inline view可以引用<table subquery>内所有clause中main query的FROM句中列出的table
    - 但是只能引用先于lateral inline view指定的table即使为先指定的table当为RIGHT OUTER JOIN, FULL OUTER JOIN时也无法引用

- <table function derived table>
  - table function derived table是由执行table function的结果集构成的逻辑表关于table function derived table的详细内容参考**Table Function Derived Table**
  - table function derived table指定TABLE和要执行的table function名称
  - 可以在<table function expression>的<table function argument list>中引用FROM子句中<table function derived table>之前列出的table的column
    - 但是即使是预先指定的table在为RIGHT OUTER JOIN, FULL OUTER JOIN时也无法引用

```
SELECT t1.col1, ft.rf2
FROM t1, TABLE( tablefunc( t1.col1 ) );
```

<correlation name>

- <table reference list>中不能存在2个以上相同的<correlation name>
- 描述了<table reference list>时如果要参照该<table name>或<derived table>则必须使用<correlation name>
  - SELECT a.i1 FROM t1 AS a WHERE a.i1 > 3;
  - (X) SELECT t1.i1 FROM t1 AS a WHERE t1.i1 > 3;
- 描述<correlation name>时可以省略前面的AS
  - SELECT a.i1 FROM t1 a;

<derived column list>

<derived column list>中不能有2个以上相同的<column name>

<cluster domain>

- <cluster domain>作为对象可描述表或视图表子查询
  - SELECT * FROM t1@G1;
  - 无法在<parenthesized joined table>描述
    - (X) SELECT * FROM (t1 INNER JOIN t2 ON t1.sk = t2.sk)@G2;
- 无法在要变更结构或数据的表或视图中描述<cluster domain>
  - (X) DELETE FROM t2@GLOBAL;
  - (X) UPDATE t1@GLOBAL SET i1 = 1;
  - (X) INSERT INTO t1@GLOBAL VALUES (1, 10);
  - (X) SELECT * FROM t1@GLOBAL FOR UPDATE;
  - (X) CREATE INDEX t1_idx ON t1@GLOBAL(i1);

<cluster domain name>

<cluster domain name>的<identifier>可以包含cluster group name或者cluster member name

- cluster group name
  - SELECT * FROM t1@G1;
- cluster member name
  - SELECT * FROM t1@G1N1;

说明

<table reference list>

<table reference list>中可以使用逗号(,)描述2个以上的表

- 描述2个以上的表时该表从左向右侧均以与cross join相同的方式运行
  - SELECT * FROM t1, t2;
  - <=> SELECT * FROM t1 CROSS JOIN t2;
- <where clause>中有2个表的join条件时与<where clause>为join条件的inner join一样运行
  - SELECT * FROM t1, t2 WHERE t1.I1 = t2.I1;
  - <=> SELECT * FROM t1 INNER JOIN t2 ON t1.i1 = t2.i1;
- <where clause>中使用outer join operator (+)时与outer join的运行方式相同
  - outer join operator (+)的详细内容参考**OUTER JOIN**

<table reference>

<table reference>可以为单张表或视图table subqueryjoined table等除joined table外其他可以拥有correlation name

joined table的详细内容参考**joined table**

<table primary>

<table primary>可以是表视图table subquery或<parenthesized joined table>

表或视图table subquery可以有correlation name这时可以省略AS如果使用correlation name应在<select list>及<where clause>等参考该表以及视图table subquery的所有地方使用correlation name

Table subquery可以描述<derived column list>与correlation name相同在参照该table subquery的column的所有地方使用<derived column list>中描述的名称为了在table subquery中使用<derived column list>必须使用correlation name

<parenthesized joined table>描述参与join运算的表的逻辑join顺序这时用括号括住的所有表的join全部为cross join和inner join时查询优化器可变更join顺序

<cluster domain>

省略<cluster domain>时与以<cluster domain name>使用GLOBAL的意义相同

详细内容参考**Cluster Domain**

<cluster domain name>

<cluster domain name>中定义的保留字有如下意义

- GLOBAL
  - 将所有集群组选定为Cluster Domain
- LOCAL
  - 仅将执行用户查询的服务器选定为Cluster Domain
    - 在G2N1执行如下查询时获取G2N1的数据
    - SELECT * FROM t1@LOCAL;
- LOCAL_OFFLINE
  - 为了查询offline状态的表数据仅将执行用户查询的服务器选定为Cluster Domain
    - 在G2N1执行如下查询时获取offline table T1的G2N1的数据
    - SELECT * FROM t1@LOCAL_OFFLINE;
  - 在Online Table描述LOCAL_OFFLINE domain时则报错

在<cluster domain name>描述<identifier>时将该名称的集群组或集群成员选定为**Cluster Domain**

使用示例

以下为使用<table name>查询单张表的SELECT语句的示例

```
gSQL> SELECT c_name, c_nation FROM customer;


C_NAME     C_NATION

---------- -------------

Customer#1 KOREA

Customer#2 CANADA

Customer#3 KOREA

Customer#4 GERMANY

Customer#5 UNITED STATES


5 rows selected.
```

以下为使用<derived table>的SELECT语句的示例

```
gSQL> SELECT * FROM (SELECT c_name, c_nation FROM customer);


C_NAME     C_NATION

---------- -------------

Customer#1 KOREA

Customer#2 CANADA

Customer#3 KOREA

Customer#4 GERMANY
```

```
Customer#5 UNITED STATES


5 rows selected.



gSQL> SELECT * FROM (SELECT c_name, c_nation FROM customer) AS CUST

("CUSTOMER_NAME", "CUSTOMER_NATION");


CUSTOMER_NAME CUSTOMER_NATION

------------- ---------------

Customer#1    KOREA

Customer#2    CANADA

Customer#3    KOREA

Customer#4    GERMANY

Customer#5    UNITED STATES


5 rows selected.
```

以下为使用<lateral derived table>的SELECT语句的示例

```
gSQL> SELECT r_name, n_name

  FROM region, (  SELECT n_name

                    FROM nation

                   WHERE n_regionkey = r_regionkey

               ) v_nation

 WHERE r_name = 'ASIA';
```

```
ERR-42000(16036): 'R_REGIONKEY': invalid identifier :
                  WHERE n_regionkey = r_regionkey
                                      *


gSQL> SELECT r_name, n_name
  FROM region, LATERAL (  SELECT n_name
                            FROM nation
                           WHERE n_regionkey = r_regionkey
                       ) v_nation
 WHERE r_name = 'ASIA';
R_NAME                    N_NAME
------------------------- -------------------------
ASIA                      INDIA
ASIA                      INDONESIA
ASIA                      JAPAN
ASIA                      CHINA
ASIA                      VIETNAM


5 rows selected.
```

以下为使用括号的joined table的SELECT语句的示例

```
gSQL> SELECT customer.c_name, o_totalprice FROM (customer INNER JOIN
orders ON customer.c_custkey = orders.o_custkey);
```

```
C_NAME     O_TOTALPRICE
---------- ------------
Customer#1    173665.47
Customer#2     46929.18
Customer#4    193846.25
Customer#3     32151.78
Customer#5     144659.2


5 rows selected.
```

以下为使用以逗号(,)区分的2个表的SELECT语句的示例

```
gSQL> SELECT c_name, o_totalprice FROM customer, orders;


C_NAME     O_TOTALPRICE
---------- ------------
Customer#1    173665.47
Customer#1     46929.18
Customer#1    193846.25
Customer#1     32151.78
Customer#1     144659.2
Customer#2    173665.47
Customer#2     46929.18
Customer#2    193846.25
Customer#2     32151.78
Customer#2     144659.2
```

```
Customer#3    173665.47

Customer#3     46929.18

Customer#3    193846.25

Customer#3     32151.78

Customer#3     144659.2

Customer#4    173665.47

Customer#4     46929.18

Customer#4    193846.25

Customer#4     32151.78

Customer#4     144659.2


C_NAME     O_TOTALPRICE

---------- ------------

Customer#5    173665.47

Customer#5     46929.18

Customer#5    193846.25

Customer#5     32151.78

Customer#5     144659.2


25 rows selected.
```

以下为使用<cluster domain>的SELECT语句的示例

- 使用GLOBAL保留字

```
gSQL> SELECT * FROM (SELECT c_name, c_nation FROM customer@GLOBAL);
```

```
C_NAME     C_NATION
---------- -------------
Customer#1 KOREA
Customer#2 CANADA
Customer#3 KOREA
Customer#4 GERMANY
Customer#5 UNITED STATES

5 rows selected.
```

- 使用LOCAL保留字

```
gSQL> SELECT * FROM (SELECT c_name, c_nation FROM customer)@LOCAL;

C_NAME     C_NATION
---------- -------------
Customer#1 KOREA
Customer#2 CANADA

2 rows selected.
```

- 使用名为G1的集群组名

```
gSQL> SELECT * FROM (SELECT c_name, c_nation FROM customer@G1);
```

```
C_NAME     C_NATION

---------- -------------

Customer#1 KOREA

Customer#2 CANADA


2 rows selected.
```

- 使用名为G2N1的集群成员名

```
gSQL> SELECT * FROM (SELECT c_name, c_nation FROM customer@G2N1);


C_NAME     C_NATION

---------- -------------

Customer#3 KOREA

Customer#4 GERMANY


2 rows selected.
```

参考

相关内容参考**subquery**

joined table

功能

描述从cartesian productinner joinouter join等中衍生的表

语句

```
<joined table> ::=

      <cross join>

    | <qualified join>

    | <natural join>


<cross join> ::=

    <table reference> CROSS JOIN <table factor>


<qualified join> ::=

    <table reference> [ <join type> ] JOIN <table reference> <join

specification>


<natural join> ::=

    <table reference> NATURAL [ <join type> ] JOIN <table factor>


<join specification> ::=

      <join condition>
```

```
    | <named columns join>


<join condition> ::=

    ON <search condition>


<named columns join> ::=

    USING ( <join column list> )


<join type> ::=

      INNER

    | { LEFT | RIGHT | FULL } [ OUTER ]


<join column list> ::=

    <column name list>
```

使用范围及访问权限

需要有访问joined table中的所有表和视图的权限

语句规则及参数

<cross join>

指定join条件的<join specification>不会在<cross join>位置

<join specification>的右侧可以为单张表或<table subquery><parenthesized joined table>

<qualified join>

- 必须描述指定join条件的<join specification>
  - SELECT * FROM t1 INNER JOIN t2 ON t1.i1 = t2.i1;
  - SELECT * FROM t1 INNER JOIN t2 USING (i1);
- 可以省略<join type>省略时处理为INNER
  - SELECT * FROM t1 JOIN t2 ON t1.i1 = t2.i1;
  - <=> SELECT * FROM t1 INNER JOIN t2 ON t1.i1 = t2.i1;
- <join type>中可以省略OUTER
  - SELECT * FROM t1 LEFT JOIN t2 ON t1.i1 = t2.i1;
  - <=> SELECT * FROM t1 LEFT OUTER JOIN t2 ON t1.i1 = t2.i1;
- <join type>为OUTER JOIN时<join specification>只能是<join condition>
  - SELECT * FROM t1 FULL OUTER JOIN t2 ON t1.i1 = t2.i1;
  - (X) SELECT * FROM t1 FULL OUTER JOIN t2 USING (i1);

<natural join>

- 指定join条件的<join specification>不会在<natural join>位置
- <natural join>右侧可以是单张表或<table subquery><parenthesized joined table>
- 可以省略<join type>省略时默认处理为INNER
  - SELECT * FROM t1 NATURAL JOIN t2;
  - <=> SELECT * FROM t1 NATURAL INNER JOIN t2;
- <join type>不支持OUTER
  - (X) SELECT * FROM t1 NATURAL LEFT OUTER JOIN t2;
- NATURAL JOIN的左侧row与右侧row中没有相同的<column name>时处理为<cross join>
  - t1(c1 INTEGER, c2 INTEGER);

    - t2(c3 INTEGER, c4 INTEGER);
    - SELECT * FROM t1 NATURAL INNER JOIN t2;
    - <=> SELECT * FROM t1 CROSS JOIN t2;
- NATURAL JOIN的左侧row与右侧row中有相同的<column name>时与使用USING语句的运行方式相同
    - t1(c1 INTEGER, c2 INTEGER);
    - t2(c2 INTEGER, c3 INTEGER);
    - SELECT * FROM t1 NATURAL INNER JOIN t2;
    - <=> SELECT * FROM t1 INNER JOIN t2 USING(c2);

<join specification>

- 只能描述<join condition>或<named columns join>中的一个
    - <join condition>
        - SELECT * FROM t1 INNER JOIN t2 ON t1.i1 = t2.i1;
    - <named columns join>
        - SELECT * FROM t1 INNER JOIN t2 USING (i1);
- 描述<named columns join>时
    - <join column list>必须描述一个以上的column name
        - SELECT * FROM t1 INNER JOIN t2 USING (i1);
    - column name不能与<table name>.<column name>一起描述
        - (X) SELECT * FROM t1 INNER JOIN t2 USING (t1.i1);
    - <join column list>列出的column应存在于JOIN的左侧row与右侧row而且可进行比较
        - t1(c1 INTEGER, c2 INTEGER);
        - t2(c2 INTEGER, c3 INTEGER);
        - SELECT * FROM t1 INNER JOIN t2 USING (c2);

- <select list>中使用*的记录结构

  1）<join column list>中描述的column

  2）左侧row中不属于<join column list>的column

  3）右侧row中不属于<join column list>的column

  - t1(c1 INTEGER, c2 INTEGER);
  - t2(c2 INTEGER, c3 INTEGER);
  - SELECT * FROM t1 INNER JOIN t2 USING (c2);
  - 记录结构：C2C1C3

- <join column list>中描述的<column name>不能同时与<table name>.<column name>参照仅可以<column name>参照

  - SELECT c2 FROM t1 INNER JOIN t2 USING (c2) WHERE c2 > 3;
  - (X) SELECT t1.c2 FROM t1 INNER JOIN t2 USING (c2);
  - (X) SELECT * FROM t1 INNER JOIN t2 USING (c2) WHERE t1.c2 > 3;

- <join column list>的join条件处理

  - 关于<join column list>中列出的各个column
  - 生成<left table name>.<column name> = <right table name>.<column name>条件
  - 生成以AND处理各个<column name>的条件的条件
  - t1(c1 INTEGER, c2 INTEGER);
  - t2(c1 INTEGER, c2 INTEGER);
  - SELECT * FROM t1 INNER JOIN t2 USING (c1, c2);
  - join条件：t1.c1 = t2.c1 AND t1.c2 = t2.c2

- <select list>中不可使用返回特定表的所有column的<table name>.*语句

  - (X) SELECT t1.*, t2.* FROM t1 INNER JOIN t2 USING (c1, c2);

说明

<cross join>

<cross join>返回结合左侧的各个row与右侧的所有row的结果

```
T1 ( 1, 1 ), ( 2, 2 )

T2 ( 2, 2 ), ( 3, 3 )


gSQL> SELECT * FROM t1 CROSS JOIN t2;

C1 C2 C1 C2

-- -- -- --

 1  1  2  2

 1  1  3  3

 2  2  2  2

 2  2  3  3

4 rows selected.
```

<cross join>不能指定join条件但可以通过<where clause>描述两张表的join条件此时与inner join运行方式相同

- SELECT * FROM t1 CROSS JOIN t2 WHERE t1.c1 = t2.c1;
- <=> SELECT * FROM t1 INNER JOIN t2 ON t1.c1 = t2.c1;

```
T1 ( 1, 1 ), ( 2, 2 )

T2 ( 2, 2 ), ( 3, 3 )
```

```
gSQL> SELECT * FROM t1 CROSS JOIN t2 WHERE t1.c1 = t2.c1;

C1 C2 C1 C2

-- -- -- --

 2  2  2  2

1 row selected.
```

<qualified join>

<qualified join>结合左侧各个row与右侧所有row后仅返回满足join条件的结果

如果<table expression>中有<where clause>则在<qualified join>的结果集合应用<where clause>条件

Inner join将<where clause>的条件处理为与join条件相同的方式进行处理其结果仍相同但outer join把<where clause>的条件处理为与join条件相同的处理方式则其结果会发生变化

INNER JOIN

```
t1 ( 1, 1 ), ( 2, 2 ), ( 3, 3 ), ( 4, 4 ), ( 5, 5 )

t2 ( 2, 2 ), ( 3, 3 )
```

- 只在ON子句中有条件时

```
gSQL> SELECT * FROM t1 INNER JOIN t2 ON t1.c1 = t2.c1 AND t1.c2 = t2.c2;

C1 C2 C1 C2

-- -- -- --

 2  2  2  2

 3  3  3  3
```

```
2 rows selected.
```

- ON子句和WHERE子句中有条件时
  - 在应用JOIN条件ON t1.c1 = t2.c1的结果集合应用WHERE条件t1.c2 = t2.c2

```
gSQL> SELECT * FROM t1 INNER JOIN t2 ON t1.c1 = t2.c1 WHERE t1.c2 = t2.c2;
C1 C2 C1 C2
-- -- -- --
 2  2  2  2
 3  3  3  3
2 rows selected.
```

  - 应用JOIN条件ON t1.c1 = t2.c1的结果集合→应用WHERE 条件 t1.c2 = t2.c2

```
( 2,  2,    2,    2 )                    ( 2,  2,    2,    2 )
  ( 3,  3,    3,    3 )                →   ( 3,  3,    3,    3 )
```

OUTER JOIN

```
t1 ( 1, 1 ), ( 2, 2 ), ( 3, 3 ), ( 4, 4 ), ( 5, 5 )
t2 ( 2, 2 ), ( 3, 3 )
```

- 只在ON子句中有条件时

```
gSQL> SELECT * FROM t1 LEFT OUTER JOIN t2 ON t1.c1 = t2.c1 AND t1.c2 =
t2.c2;
C1 C2   C1   C2
```

```
-- -- ---- ----
 1  1 null null
 2  2    2    2
 3  3    3    3
 4  4 null null
 5  5 null null
5 rows selected.
```

- ON子句和WHERE子句中有条件时
  - 在应用JOIN条件ON t1.c1 = t2.c1的结果集合应用WHERE条件t1.c2 = t2.c2

```
gSQL> SELECT * FROM t1 LEFT OUTER JOIN t2 ON t1.c1 = t2.c1 WHERE t1.c2 =
t2.c2;
C1 C2 C1 C2
-- -- -- --
 2  2  2  2
 3  3  3  3
2 rows selected.
```

  - 应用JOIN条件ON t1.c1 = t2.c1的结果集合→应用WHERE 条件 t1.c2 = t2.c2

```
( 1,  1, null, null )
  ( 2,  2,    2,    2 )                      ( 2,  2,    2,    2 )
  ( 3,  3,    3,    3 )                  →   ( 3,  3,    3,    3 )
  ( 4,  4, null, null )
  ( 5,  5, null, null )
```

Left outer join是当右侧row中存在满足左侧row的join条件的row时返回对应row的组合若右侧row中没有满足join条件的row时则维持左侧row的值并把右侧row的值均设置为NULL后返回

LEFT OUTER JOIN

```
t1 ( 1, 1 ), ( 2, 2 )
t2 ( 2, 2 ), ( 3, 3 )


gSQL> SELECT * FROM t1 LEFT OUTER JOIN t2 ON t1.c1 = t2.c1;
C1 C2   C1   C2
-- -- ---- ----
 1  1 null null
 2  2    2    2
2 rows selected.
```

Right outer join的操作方法正好与left outer join相反

```
RIGHT OUTER JOIN


t1 ( 1, 1 ), ( 2, 2 )
t2 ( 2, 2 ), ( 3, 3 )


gSQL> SELECT * FROM t1 RIGHT OUTER JOIN t2 ON t1.c1 = t2.c1;
  C1   C2 C1 C2
---- ---- -- --
   2    2  2  2
```

```
null null  3  3

2 rows selected.
```

Full outer join对不满足left outer join的结果与join条件的右侧row返回将左侧row的值均设置为NULL的row

```
FULL OUTER JOIN


t1 ( 1, 1 ), ( 2, 2 )

t2 ( 2, 2 ), ( 3, 3 )


gSQL> SELECT * FROM t1 FULL OUTER JOIN t2 ON t1.c1 = t2.c1;

  C1   C2   C1   C2

---- ---- ---- ----

   1    1 null null

   2    2    2    2

null null    3    3

3 rows selected.
```

<natural join>

<natural join>是对参与join的两张表中拥有相同名称的所有Column以等值（equal）join连接即在参与join的两张表中拥有相同名称的所有column等同于在inner join描述'USING'语句

```
t1 ( C1 INTEGER, C2 INTEGER )

t2 ( C1 INTEGER, C3 INTEGER )
```

```
t1 ( 1, 10 ), ( 2, 20 ), ( 3, 30 )

t2 ( 1, 100 ), ( 2, 200 ), ( 3, 300 )


gSQL> SELECT * FROM t1 NATURAL JOIN t2;

C1 C2  C3

-- -- ---

 1 10 100

 2 20 200

 3 30 300

3 rows selected.


gSQL> SELECT * FROM t1 INNER JOIN t2 USING ( c1 );

C1 C2  C3

-- -- ---

 1 10 100

 2 20 200

 3 30 300

3 rows selected.
```

<join specification>

描述join条件

<join condition>描述连接join语句左侧row与右侧row的条件

<named columns join>是当左侧row与右侧row存在相同的<column name>时列出其并描述join条件

```
t1 ( C1 INTEGER, C2 INTEGER )

t2 ( C1 INTEGER, C3 INTEGER )


t1 ( 1, 10 ), ( 2, 20 ), ( 3, 30 )

t2 ( 1, 100 ), ( 2, 200 ), ( 3, 300 )


• <join condition>

gSQL> SELECT * FROM t1 INNER JOIN t2 ON t1.c1 = t2.c1;

C1 C2 C1  C3

-- -- -- ---

 1 10  1 100

 2 20  2 200

 3 30  3 300

3 rows selected.


• <named columns join>

gSQL> SELECT * FROM t1 INNER JOIN t2 USING ( c1 );

C1 C2  C3

-- -- ---

 1 10 100

 2 20 200

 3 30 300

3 rows selected.
```

使用示例

以下为使用<cross join>的SELECT语句的示例

```
gSQL> SELECT c_name, o_totalprice FROM customer CROSS JOIN orders;


C_NAME     O_TOTALPRICE
---------- ------------
Customer#1    173665.47
Customer#1     46929.18
Customer#1    193846.25
Customer#1     32151.78
Customer#1     144659.2
Customer#2    173665.47
Customer#2     46929.18
Customer#2    193846.25
Customer#2     32151.78
Customer#2     144659.2
Customer#3    173665.47
Customer#3     46929.18
Customer#3    193846.25
Customer#3     32151.78
Customer#3     144659.2
Customer#4    173665.47
Customer#4     46929.18
Customer#4    193846.25
```

```
Customer#4     32151.78

Customer#4     144659.2


C_NAME     O_TOTALPRICE

---------- ------------

Customer#5    173665.47

Customer#5     46929.18

Customer#5    193846.25

Customer#5     32151.78

Customer#5     144659.2


25 rows selected.
```

以下为使用inner join的SELECT语句的示例

```
gSQL> SELECT c_name, o_totalprice FROM customer INNER JOIN orders ON

c_custkey = o_custkey;


C_NAME     O_TOTALPRICE

---------- ------------

Customer#1    173665.47

Customer#2     46929.18

Customer#4    193846.25

Customer#3     32151.78

Customer#5     144659.2
```

```
5 rows selected.
```

以下为使用outer join的SELECT语句的示例

```
gSQL> SELECT c_name, o_totalprice FROM customer LEFT OUTER JOIN orders ON
c_custkey = o_custkey AND o_orderdate < '1996-01-01';

C_NAME     O_TOTALPRICE
---------- ------------
Customer#1         null
Customer#2         null
Customer#3     32151.78
Customer#4    193846.25
Customer#5     144659.2

5 rows selected.

gSQL> SELECT c_name, o_totalprice FROM customer RIGHT OUTER JOIN orders ON
c_custkey = o_custkey AND c_nation = 'KOREA';

C_NAME     O_TOTALPRICE
---------- ------------
Customer#1    173665.47
null           46929.18
null          193846.25
Customer#3     32151.78
```

```
null           144659.2


5 rows selected.


gSQL> SELECT c_name, o_totalprice FROM customer FULL OUTER JOIN orders ON
c_custkey = o_custkey AND c_nation = 'KOREA' AND o_orderdate < '1996-01-
01';


C_NAME     O_TOTALPRICE
---------- ------------
Customer#1         null
Customer#2         null
Customer#3     32151.78
Customer#4         null
Customer#5         null
null          173665.47
null           46929.18
null          193846.25
null           144659.2


9 rows selected.
```

以下为使用natural join的SELECT语句的示例

```
gSQL> SELECT c_name, o_totalprice FROM (SELECT c_custkey custkey, c_name
FROM customer) NATURAL JOIN (SELECT o_custkey custkey, o_totalprice FROM
```

```
orders);


C_NAME     O_TOTALPRICE
---------- ------------
Customer#1    173665.47
Customer#2     46929.18
Customer#4    193846.25
Customer#3     32151.78
Customer#5     144659.2


5 rows selected.
```

兼容性

Feature ID	说明	是否支持
F401	Extended joined table	O
F402	Named column joins for LOBs, arrays, and multisets	X
F403	Partitioned join tables	X

Table 10-11 标准SQL兼容性

参考

相关内容参考**from clause**

where clause

功能

对<from clause>的结果应用<search condition>

语句

```
<where clause> ::=

    WHERE <search condition>
```

语句规则及参数

<where clause>

WHERE关键字后面应为返回boolean type的<search condition>

说明

<where clause>的详细内容参考**Conditions**

使用示例

以下为使用<where clause>的SELECT语句的示例

```
gSQL> SELECT s_name, s_nation FROM supplier WHERE s_nation = 'KOREA';
```

```
S_NAME                    S_NATION
------------------------- --------
Supplier#2                KOREA

1 row selected.

gSQL> SELECT s_name, ps_availqty, ps_supplycost FROM supplier, partsupp
WHERE s_nation = 'KOREA' AND s_suppkey = ps_suppkey;

S_NAME                    PS_AVAILQTY PS_SUPPLYCOST
------------------------- ----------- -------------
Supplier#2                       8076        993.49
Supplier#2                       4069        357.84

2 rows selected.
```

兼容性

Feature ID	说明	是否支持
F441	Extended set function support	O

Table 10-12 标准SQL兼容性

参考

相关内容参考**query specification**

hierarchical query clause

功能

描述使以层次结构检索层次模式数据

使用开始条件和下级连接条件将表的record返回为depth-first顺序的层次结构

语句

```
<hierarchical query clause> ::=

    <start with connect by clause> [ <order siblings by clause> ]


<start with connect by clause> ::=

    <start with clause> <connect by clause>

    | <connect by clause> <start with clause>

    | <connect by clause>


<start with clause> ::=

    START WITH <start_with_condition>


<connect by clause> ::=

    CONNECT BY [NOCYCLE] <connect_by_condition>


<order siblings by clause> ::=

    ORDER SIBLINGS BY <ordering element> [ { <comma> <ordering
```

```
element> }... ]


<ordering element> ::=
    <value expression> [ASC | DESC] [NULLS FIRST | NULLS LAST]


<hierarchy expression> ::=
    LEVEL
    | CONNECT_BY_ISCYCLE
    | CONNECT_BY_ISLEAF
    | PRIOR <value expression>
    | CONNECT_BY_ROOT <value expression>
    | SYS_CONNECT_BY_PATH <left paren> <value expression> <comma>
<character string literal> <right paren>
```

使用范围及访问权限

在<query specification>语句支持执行的用户需要满足 <query specification>的访问权限

详细内容参考**query specification**

语句规则及参数

<hierarchical query clause>

必须描述<connect by clause>

有必要时描述<start with clause>或<order siblings by clause>

<start with clause>

在数据层次描述root（最上级）record的条件

未描述时from子句的所有record成为root record对象

在SELECT语句中仅可描述一次

<connect by clause>

描述上级（parent）record和下级（child）record的关系

使用表示上级record的column值的PRIOR operator表示上级record和下级record的关系

未使用PRIOR operator描述上级record和下级record的连接条件时可能会产生无限loop

在SELECT语句中仅可描述一次

- NOCYCLE
  - 未描述NOCYCLE选项时
    - 产生cycle时该查询产生error后执行中断
  - 描述NOCYCLE选项时
    - 产生cycle时该查询不报错
    - 引起cycle的record暂停检索下级record并不包含在结果record中
    - 对未产生cycle的sibling rows继续执行查询
    - 引起cycle的record的上级record的CONNECT_BY_ISCYCLE中存储1
    - 未引起cycle的record的上级record的CONNECT_BY_ISCYCLE中存储0

<order siblings by clause>

指定拥有相同上级（parent）record的sibling record之间的fetch顺序

- 排列顺序
  - ASC
  - DESC
  - 未指定时默认值为ASC
- Null ordering
  - NULL FIRST
  - NULLS LAST
  - 未指定时默认值为NULLS LAST

<hierarchy expression>

- 构成hierarchy query时可获得如下hierarchy信息
  - LEVEL
  - CONNECT_BY_ISCYCLE
  - CONNECT_BY_ISLEAF
  - PRIOR
  - CONNECT_BY_ROOT
  - SYS_CONNECT_BY_PATH

Expression	Result DataType
LEVEL	NATIVE_BIGINT
CONNECT_BY_ISCYCLE	NATIVE_BIGINT
CONNECT_BY_ISLEAF	NATIVE_BIGINT
PRIOR expr	expr的DataType

Expression	Result DataType
CONNECT_BY_ROOT expr	expr的DataType
SYS_CONNECT_BY_PATH(expr, literal)	VARCHAR(4000 characters)

Table 10-13 <hierarchy expression>的结果类型

可描述<hierarchy expression>的语句为如下

Expression\clause	FROM	START WITH	CONNECT BY	ORDER SIBLINGS BY	WHERE/ GROUP BY/ HAVING	ORDER BY/ SELECT TARGET
LEVEL	X	O	O	X	O	O
CONNECT_BY_ISCYCLE	X	X	X	X	O	O
CONNECT_BY_ISLEAF	X	X	X	X	O	O
PRIOR	X	X	O	X	O	O
CONNECT_BY_ROOT	X	X	X	X	O	O
SYS_CONNECT_BY_PATH	X	X	X	X	O	O

<hierarchy expression>是否可以作为<hierarchy expression>的参数使用如下

Expression\Argument(expr)	LEVEL	CONNECT_BY_ISCYCLE	CONNECT_BY_ISLEAF	PRIOR	CONNEC
PRIOR expr	X	X	X	X	

Expression\Argument(expr)	LEVEL	CONNECT_BY_ISCYCLE	CONNECT_BY_ISLEAF	PRIOR	CONNEC
CONNECT_BY_ROOT expr	X	X	X	X	
SYS_CONNECT_BY_PARTH(expr,literal)	O	O	O	O	

说明

<hierarchical query clause>是以层次结构检索层次模式数据的语句

使用开始条件和下级连接条件以depth-first顺序的层次结构返回表的record

在SELECT描述<hierarchical query clause>时按照如下顺序进行处理

1. FROM子句的ON条件
2. START WITH
3. CONNECT BY
4. WHERE

- FROM子句中仅有单张表时

```
SELECT *
 FROM r_region
WHERE r_population > 10000000              ❸ WHERE子句的条件
START WITH r_name = 'EARTH'                ❶ START WITH
CONNECT BY r_domain = PRIOR r_name         ❷ CONNECT BY
```

- FROM子句中有由多张表构成的join条件时
  - 在FROM子句的ON子句描述join条件时

```
SELECT *
  FROM r_region INNER JOIN s_region
       ON r_id = s_id                    ❶ ON子句的join条件
 WHERE r_population > 10000000           ❹ WHERE子句的条件
START WITH r_name = 'EARTH'              ❷ START WITH
CONNECT BY r_domain = PRIOR r_name       ❸ CONNECT BY
```

- 在WHERE子句描述join条件时

```
SELECT *
  FROM r_region, s_region
 WHERE r_population > 10000000           ❸ WHERE子句的条件
   AND r_id = s_id                       ❸ WHERE子句的join条件
START WITH r_name = 'EARTH'              ❶ START WITH
CONNECT BY r_domain = PRIOR r_name       ❷ CONNECT BY
```

- 在FROM的ON子句和WHERE子句均描述join条件时

```
SELECT *
  FROM r_region INNER JOIN s_region
       ON r_name = s_name                ❶ ON子句的join条件
 WHERE r_population > 10000000           ❹ WHERE子句的条件
   AND r_id = s_id                       ❹ WHERE子句的join条件
START WITH r_name = 'EARTH'              ❷ START WITH
CONNECT BY r_domain = PRIOR r_name       ❸ CONNECT BY
```

<order siblings by clause>

指定在<hierarchical query clause>内拥有相同上级（parent）record的sibling record之间的fetch顺序

<order siblings by clause>与<order by clause>是独立的语句

```
gSQL>
SELECT * FROM t1;


I1  I2
--- ----
A   null
AA  A
AB  A
fAA AA
eAA AA
bAA AA
dAB AB
cAB AB
aAB AB


9 rows selected.
```

- 以下为指定拥有相同上级record的sibling record之间的fetch顺序之间的示例

```
gSQL>
```

```
SELECT LEVEL, i1, i2
  FROM t1
START WITH i1 = 'A'
CONNECT BY i2 = PRIOR i1
ORDER SIBLINGS BY i1;


LEVEL I1  I2
----- --- ----
    1 A   null
    2 AA  A
    3 bAA AA
    3 eAA AA
    3 fAA AA
    2 AB  A
    3 aAB AB
    3 cAB AB
    3 dAB AB


9 rows selected.
```

- 以下为使用ORDER BY子句按照LEVEL顺序对通过层次结构检索的所有结果进行排列的示例

```
gSQL>
SELECT LEVEL, i1, i2
  FROM t1
```

```
START WITH i1 = 'A'

CONNECT BY i2 = PRIOR i1

ORDER SIBLINGS BY i1

ORDER BY LEVEL;


LEVEL I1  I2

----- --- ----

    1 A   null

    2 AA  A

    2 AB  A

    3 bAA AA

    3 eAA AA

    3 fAA AA

    3 aAB AB

    3 cAB AB

    3 dAB AB


9 rows selected.
```

<hierarchy expression>

hierarchy expression的功能为如下

- PRIOR
  - 以当前record的上级record为基础获取信息
  - PRIOR是单行运算符与单行运算符 +- 等拥有相同的优先级

```
gSQL>
SELECT i1, i2
  FROM t1
START WITH i1 = 'X'
CONNECT BY i2 = PRIOR i1;


I1    I2
----- ----
X     null
XA    X
XXA   XA
XXXA  XXA
XXXXA XXXA


5 rows selected.
```

- LEVEL
  - Record所属的层级值
  - root record的LEVEL为1root的下级（child）record LEVEL为2
  - 下级（child）record的LEVEL逐级增加1

```
gSQL>
SELECT LEVEL, i1, i2
  FROM t1
START WITH i1 = 'X'
CONNECT BY i2 = prior i1;
```

```
LEVEL I1    I2
----- ----- ----
   1 X     null
   2 XA    X
   3 XXA   XA
   4 XXXA  XXA
   5 XXXXA XXXA

5 rows selected.
```

- CONNECT_BY_ISCYCLE
  - 获取当前record和相关的下级record中是否存在产生cycle的record的信息
  - 仅可在CONNECT BY语句中有NOCYCLE时使用

```
gSQL>
SELECT * FROM t1;

I1 I2
-- ----
A  null
AA A
AB A
AC A
AA AA
AB AA
```

```
6 rows selected.


gSQL>
SELECT i1, i2, CONNECT_BY_ISCYCLE
  FROM t1
START WITH i1 = 'A'
CONNECT BY NOCYCLE i2 = prior i1;


I1 I2   CONNECT_BY_ISCYCLE
-- ---- ------------------
A  null                  0
AA A                     1
AB AA                    0
AB A                     0
AC A                     0


5 rows selected.
```

- CONNECT_BY_ISLEAF
  - 获取是否存在与当前record相关的下级record的信息
  - 没有与当前record相关的下级record时返回1

```
gSQL>
SELECT i1, i2, CONNECT_BY_ISLEAF
  FROM t1
```

```
START WITH i1 = 'X'

CONNECT BY i2 = prior i1;


I1    I2   CONNECT_BY_ISLEAF

----- ---- -----------------

X     null                 0

XA    X                    0

XXA   XA                   0

XXXA  XXA                  0

XXXXA XXXA                 1


5 rows selected.
```

- CONNECT_BY_ROOT
  - 以当前record的最上级record为基础获取信息

```
gSQL>

SELECT i1, i2, CONNECT_BY_ROOT i1

  FROM t1

START WITH i1 = 'X'

CONNECT BY i2 = prior i1;


I1    I2   CONNECT_BY_ROOT I1

----- ---- ------------------

X     null X

XA    X    X
```

```
XXA   XA   X

XXXA  XXA  X

XXXXA XXXA X


5 rows selected.
```

- SYS_CONNECT_BY_PATH
  - 跟随当前record的上级record递归性的搜索并获取信息

```
gSQL>
SELECT i1, i2, SYS_CONNECT_BY_PATH( i1, '/' )
  FROM t1
START WITH i1 = 'X'
CONNECT BY i2 = prior i1;


I1    I2   SYS_CONNECT_BY_PATH( I1, '/' )
----- ---- ------------------------------
X     null /X
XA    X    /X/XA
XXA   XA   /X/XA/XXA
XXXA  XXA  /X/XA/XXA/XXXA
XXXXA XXXA /X/XA/XXA/XXXA/XXXXA


5 rows selected.
```

使用示例

以下为用于hierarchical query clause示例的emp表的record检索结果

```
gSQL>
SELECT * FROM emp;


NAME    MGR
------- -------
Kelly   null
Bill    Kelly
Jackson Kelly
Joe     Kelly
Scott   Bill
Larry   Bill
Paul    Jackson
Bill    Bill


8 rows selected.
```

以下为产生cycle的示例

```
gSQL>
SELECT *
  FROM emp
START WITH mgr IS NULL
```

```
CONNECT BY mgr = PRIOR name
ORDER SIBLINGS BY name;


ERR-42000(16511): cycle detected while executing recursive WITH query
```

以下为通过CONNECT BY NOCYCLE语句执行查询的示例

```
gSQL>
SELECT *
  FROM emp
START WITH mgr IS NULL
CONNECT BY NOCYCLE mgr = PRIOR name
ORDER SIBLINGS BY name;

NAME    MGR
------- -------
Kelly   null
Bill    Kelly
Larry   Bill
Scott   Bill
Jackson Kelly
Paul    Jackson
Joe     Kelly

7 rows selected.
```

以下为使用hierarchy expression查询层次结构数据的信息的示例

```
gSQL>
SELECT name,
       mgr,
       PRIOR name AS prior_mgr,
       LEVEL,
       CONNECT_BY_ISCYCLE AS iscycle,
       CONNECT_BY_ISLEAF AS isleaf,
       CONNECT_BY_ROOT mgr AS root_mgr,
       SYS_CONNECT_BY_PATH( mgr, '/' ) AS path
  FROM emp
START WITH mgr IS NULL
CONNECT BY NOCYCLE mgr = PRIOR name
ORDER SIBLINGS BY name;


NAME    MGR     PRIOR_MGR LEVEL ISCYCLE ISLEAF ROOT_MGR PATH
------- ------- --------- ----- ------- ------ -------- ---------------
Kelly   null    null          1       0      0 null     /
Bill    Kelly   Kelly         2       1      0 null     //Kelly
Larry   Bill    Bill          3       0      1 null     //Kelly/Bill
Scott   Bill    Bill          3       0      1 null     //Kelly/Bill
Jackson Kelly   Kelly         2       0      0 null     //Kelly
Paul    Jackson Jackson       3       0      1 null     //Kelly/Jackson
Joe     Kelly   Kelly         2       0      1 null     //Kelly
```

```
7 rows selected.
```

group by clause

功能

描述对之前语句处理的结果应用<group by clause>的grouped table

语句

```
<group by clause> ::=

    GROUP BY <grouping element list>


<grouping element list> ::=

    <grouping element> [ { , <grouping element> } ... ]


<grouping element> ::=

      <ordinary grouping set>

    | <empty grouping set>


<ordinary grouping set> ::=

      <grouping column reference>


<grouping column reference> ::=

    <column reference>

    | <value_expression>
```

```
<empty grouping set> ::=

    <left paren> <right paren>
```

使用范围及访问权限

执行<group by clause>时不需要额外的访问权限

语句规则及参数

<ordinary grouping set>

由一个以上的<grouping column reference>构成

不支持LONG type (LONG VARCHAR, LONG VARBINARY)

- SELECT c1, sum(c2) FROM t1 GROUP BY c1;
- SELECT sum(c1) FROM t1 GROUP BY NULL;

<empty grouping set>

只能使用括号描述

- SELECT sum(c1) FROM t1 GROUP BY ();

说明

<grouping element list>

执行将<group by clause>中描述的<grouping element list>捆绑为一个GROUPING SET的grouping与存在于GROUPING SET的所有<grouping element>值一致时处理为相同的group

- 描述<group by clause>时<select list>中可以有如下表达式
  - 常数
  - <group by clause>中描述的<grouping column reference>
  - 包含<group by clause>中描述的<grouping column reference>的运算式
  - 未描述在<group by clause>的column的集合函数
  - SELECT c1, sum(c2) FROM t1 GROUP BY c1;

<grouping column reference>

<grouping column reference>可以包含<column reference>或<value expression>

- <column reference>
  - 只能引用属于<query specification>的<from clause>的Column
    - SELECT c1 FROM t1 GROUP BY c1;
  - 存在相同的column名时需要使用表名等明确描述Column名
    - SELECT t1.c1, t2.c1 FROM t1, t2 GROUP BY t1.c1, t2.c1;
- <value expression>
  - 包含<column reference>的expression
    - 可使用<column reference>划分为多个group
    - SELECT sum(c2) FROM t1 GROUP BY c1 + 10;

- 不包含<column reference>的表达式
  - <value expression>值均为相同的常数值因此所有记录由单个group构成
  - <value expression>中描述null值时null值视为相同的值所有记录由单个group构成
  - SELECT sum(c1), sum(c2) FROM t1 GROUP BY NULL;

<empty grouping set>

<empty grouping set>的所有记录由单个group构成

• SELECT sum(c1), sum(c2) FROM t1 GROUP BY ();

使用示例

以下为使用GROUP BY的SELECT语句的示例

```
gSQL> SELECT c_nation, COUNT(c_name) FROM customer GROUP BY c_nation;


C_NATION      COUNT(C_NAME)

------------- -------------

UNITED STATES             1

CANADA                    1

KOREA                     2

GERMANY                   1


4 rows selected.


gSQL> SELECT COUNT(c_name) FROM customer GROUP BY NULL;
```

```
COUNT(C_NAME)

-------------

            5


1 row selected.


gSQL> SELECT COUNT(c_name) FROM customer GROUP BY ();


COUNT(C_NAME)

-------------

            5


1 row selected.
```

兼容性

Feature ID	说明	是否支持
T431	Extended grouping capabilities	X
T432	Nested and concatenated GROUPING SETS	X
T434	GROUP BY DISTINCT	X

Table 10-14 标准SQL兼容性

参考

相关内容参考下文

- **having clause**
- **query specification**

having clause

功能

描述删除不满足<search condition>的group的grouped table

语句

```
<having clause> ::=

    HAVING <search condition>
```

使用范围及访问权限

执行<having clause>时不需要额外的访问权限

语句规则及参数

<having clause>

- 只有<group by clause>中描述的<grouping column reference>在<search condition>中可在没有集合函数的情况下使用
    - SELECT c1, sum(c2) FROM t1 GROUP BY c1 HAVING c1 > 3;
- 没有在<group by clause>中定义的column可以使用集合函数进行描述
    - SELECT c1, sum(c2) FROM t1 GROUP BY c1 HAVING sum(c2) > 100;

说明

<having clause>

<having clause>定义grouping的数据的检索条件

通常与<group by clause>一起使用在没有<group by clause>的情况下使用<having clause>时视为有<empty grouping set>

- SELECT sum(c1), sum(c2) FROM t1 HAVING sum(c1) > 0;
- <=> SELECT sum(c1), sum(c2) FROM t1 GROUP BY () HAVING sum(c1) > 0;

<having clause>中只能描述<group by clause>中描述的<grouping column reference>

未在<group by clause>中描述的column可使用集合函数进行描述

- SELECT c1, sum(c2) FROM t1 GROUP BY c1 HAVING c1 > 3 AND sum(c2) > 100;

使用示例

以下为使用<having clause>的SELECT语句的示例

```
gSQL> SELECT c_nation, COUNT(c_name) FROM customer GROUP BY c_nation
HAVING COUNT(c_name) > 1;


C_NATION COUNT(C_NAME)
-------- -------------
KOREA                2
```

```
1 row selected.


gSQL> SELECT COUNT(c_name) FROM customer HAVING COUNT(c_name) > 1;


COUNT(C_NAME)
-------------
            5


1 row selected.
```

兼容性

Feature ID	说明	是否支持
T301	Functional dependencies	O

Table 10-15 标准SQL兼容性

参考

相关内容参考下文

- **group by clause**
- **Conditions**

window clause

功能

定义select list和order by clause中描述的window function的执行范围

语句

```
<window clause> ::=

     WINDOW <window definition list>


<window definition list> ::=

     <window definition> [ { <comma> <window definition> }... ]


<window definition> ::=

     <new window name> AS <window specification>


<new window name> ::=

     <window name>


<window specification> ::=

     <left paren> <window specification details> <right paren>


<window specification details> ::=

     [ <existing window name> ]
```

```
          [ <window partition clause> ]

          [ <window order clause> ]

          [ <window frame clause> ]


<existing window name> ::=

     <window name>


<window partition clause> ::=

     PARTITION BY <window partition column reference list>


<window partition column reference list> ::=

     <window partition column reference>

          [ { <comma> <window partition column reference> }... ]


<window partition column reference> ::=

     <column reference>


<window order clause> ::=

     ORDER BY <sort specification list>


<sort specification list> ::=

     <sort specification> [ { <comma> <sort specification> }... ]


<sort specification> ::=

     <sort key> [ <ordering specification> ] [ <null ordering> ]
```

```
<sort key> ::=
     <value expression>

<ordering specification> ::=
       ASC
     | DESC

<null ordering> ::=
      NULLS FIRST
    | NULLS LAST

<window frame clause> ::=
     <window frame units> <window frame extent>
          [ <window frame exclusion> ]

<window frame units> ::=
       ROWS
     | RANGE
     | GROUPS

<window frame extent> ::=
       <window frame start>
     | <window frame between>
```

```
<window frame start> ::=

       UNBOUNDED PRECEDING

     | <window frame preceding>

     | CURRENT ROW


<window frame preceding> ::=

     <unsigned value specification> PRECEDING


<window frame between> ::=

     BETWEEN <window frame bound 1> AND <window frame bound 2>


<window frame bound 1> ::=

     <window frame bound>


<window frame bound 2> ::=

     <window frame bound>


<window frame bound> ::=

       <window frame start>

     | UNBOUNDED FOLLOWING

     | <window frame following>


<window frame following> ::=

     <unsigned value specification> FOLLOWING
```

```
<window frame exclusion> ::=
       EXCLUDE CURRENT ROW
     | EXCLUDE GROUP
     | EXCLUDE TIES
     | EXCLUDE NO OTHERS
```

使用范围及访问权限

当window clause中存在column时需要设置column的访问权限

语句规则及参数

<window clause>

无法在window clause中描述window function

<window definition list>

可以定义多个<window definition>

<window definition>

通过<new window name>描述window function的执行范围

<window clause>内<new window name>不能重复

可以在window function的over子句引用<new window name>

```
SELECT SUM(i2) OVER w1
```

```
  FROM t1
WINDOW w1 AS ( PARTITION BY i1 ORDER BY i2 );
```

<window specification>

定义window function的执行范围

可以引用<existing window name>在原来定义信息基础上重定义<window specification>

<existing window name>仅可引用<window definition list>中已定义的<new window name>

- 在window子句中引用

```
SELECT SUM(i2) OVER w2
  FROM t1
WINDOW w1 AS ( PARTITION BY i1
               ORDER BY i2 ),
       w2 AS ( w1 ROWS BETWEEN UNBOUNDED PRECEDING  <---
                           AND CURRENT ROW );
```

- 在window function的over子句引用

```
SELECT SUM(i2) OVER ( w1 ROWS BETWEEN UNBOUNDED PRECEDING  <---
                                  AND CURRENT ROW )
  FROM t1
WINDOW w1 AS ( PARTITION BY i1
               ORDER BY i2 );
```

引用<existing window name>重定义<window specification>时

- 无法在重定义部分描述<window partition clause>

```
SELECT SUM(i2) OVER ( w1 PARTITION BY i1 )  <--- ( X )
  FROM t1
WINDOW w1 AS ( );
```

- 在<existing window name>中描述order by clause时

  无法在重定义部分描述order by clause

```
SELECT SUM(i2) OVER ( w1 ORDER BY i3 ) <--- ( X )
  FROM t1
WINDOW w1 AS ( PARTITION BY i1
               ORDER BY i2 );
```

- 无法在<existing window name>中描述window frame clause

```
SELECT SUM(i2) OVER ( w1 )
  FROM t1
WINDOW w1 AS ( PARTITION BY i1
               ORDER BY i2
               ROWS BETWEEN UNBOUNDED PRECEDING   <--- ( X )
                        AND CURRENT ROW );
```

<window frame start>

省略frame end时<window frame start>和<window frame start> AND CURRENT ROW相同

- UNBOUNDED PRECEDING
  - UNBOUNDED PRECEDING AND CURRENT ROW
- *offset* PRECEDING
  - *offset* PRECEDING AND CURRENT ROW
    - 3 PRECEDING AND CURRENT ROW
- CURRENT ROW
  - CURRENT ROW AND CURRENT ROW

<window frame between>

- 无法在frame start中描述UNBOUNDED FOLLOWING
  - BETWEEN *UNBOUNDED FOLLOWING* AND UNBOUNDED FOLLOWING (X)
- 无法在frame end中描述UNBOUNDED PRECEDING
  - BETWEEN UNBOUNDED PRECEDING AND *UNBOUNDED PRECEDING* (X)
- frame start为CURRENT ROW时无法在frame end中描述<window frame preceding>
  - BETWEEN CURRENT ROW AND *1 PRECEDING* (X)
- frame start为<window frame following>时无法在frame end中描述<window frame preceding>或CURRENT ROW
  - BETWEEN 3 FOLLOWING AND *1 PRECEDING* (X)
  - BETWEEN 3 FOLLOWING AND *CURRENT ROW* (X)

<window frame following> / <window frame preceding>

offset PRECEDING / *offset* FOLLOWING

- 无法在offset中描述负数或NULL
  - *NULL* PRECEDING (X)
  - *-1* PRECEDING (X)
  - *NULL* FOLLOWING (X)
  - *-1* FOLLOWING (X)
- frame unit为RANGE时
  - 如果window ORDER BY的sort key为数字类型则在offset中以数字类型进行描述
    - ORDER BY orderkey RANGE BETWEEN 3 PRECEDING AND 5 FOLLOWING
  - 如果window ORDER BY的sort key为datetime或interval类型则在offset中以interval类型进行描述
    - ORDER BY orderdate RANGE BETWEEN INTERVAL'3'DAY PRECEDING AND INTERVAL'5'day FOLLOWING
- frame unit为ROWS/GROUPS时则在offset中以常数型数字进行描述
  - ORDER BY orderkey ROWS BETWEEN 3 PRECEDING AND 5 FOLLOWING
  - ORDER BY orderkey GROUPS BETWEEN 3 PRECEDING AND 5 FOLLOWING

说明

WINDOW clause描述select list和order by clause中描述的window function的执行范围

使用<window partition clause>进行分组

使用<window order clause>给组内记录排序

使用<window frame clause>定义组内已排序记录的window function对象记录的范围

WINDOW clause对执行FROMWHEREGROUP BYHAVING子句后的结果集执行

查询语句中使用aggregateGROUP BYHAVING子句时应在WINDOW子句中描述组的column而不是描述原表的column

<window specification>

描述各记录的window function的执行范围

可以引用<existing window name>在原来定义信息基础上重定义<window specification>

```
SELECT SUM(i2) OVER ( w1
                      ORDER BY i2
                      ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW )
  FROM t1
WINDOW w1 AS ( PARTITION BY i1 ),
       w2 AS ( w1 ORDER BY i3 );
```

→ 相同语句

```
SELECT SUM(i2) OVER ( PARTITION BY i1
                      ORDER BY i2
                      ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW )
  FROM t1
WINDOW w1 AS ( PARTITION BY i1 ),
       w2 AS ( PARTITION BY i1
               ORDER BY i3 );
```

在window function OVER子句中引用<window name> wname时OVER wname和OVER (wname) 不同

- OVER wname

```
* 引用定义为wname的<window specification>信息


ex) 引用w1
SELECT SUM(i2) OVER w1
  FROM t1
WINDOW w1 AS ( PARTITION BY i1
               ORDER BY i2
               ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW );
```

- OVER(wname)

```
* 在原来定义的信息的基础上重定义<window specification>
  无法在原有信息上定义<window frame clause>


ex) 引用w1
SELECT SUM(i2) OVER ( w1 ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT
ROW )
  FROM t1
WINDOW w1 AS ( PARTITION BY i1
               ORDER BY i2 );
```

```
ex) 引用w1 ( 错误情况：无法在原有信息上定义<window frame clause> )
SELECT SUM(i2) OVER ( w1 )
  FROM t1
WINDOW w1 AS ( PARTITION BY i1
               ORDER BY i2
               ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW );  <---
```

<window partition clause>

使用PARTITION BY基于<window partition column reference list>将查询结果集分组

若省略此子句函数将把查询结果集的所有row处理为单个群组

- 描述<window partition clause>时

```
gSQL>
SELECT orderdate,
       orderkey,
       totalprice,
       SUM( totalprice ) OVER( PARTITION BY orderdate ) AS SUM_OVER_RESULT
FROM orders;

ORDERDATE  ORDERKEY TOTALPRICE SUM_OVER_RESULT
---------- -------- ---------- ---------------
1998-07-24     1730     204656          520630
1998-07-24    17056     289620          520630
1998-07-24    19937      26354          520630      partition ❶
```

```
------------------------------------------------------------------------

1998-07-25     2400     150304          368523

1998-07-25    11204      27165          368523

1998-07-25    11938     191054          368523      partition ❷

------------------------------------------------------------------------

1998-07-26    35655      13698          362691

1998-07-26    53377     185930          362691

1998-07-26    55010     163063          362691      partition ❸

------------------------------------------------------------------------


9 rows selected.
```

- 省略<window partition clause>时

```
gSQL>
SELECT orderdate,
       orderkey,
       totalprice,
       SUM( totalprice ) OVER() AS SUM_OVER_RESULT
  FROM orders;


ORDERDATE  ORDERKEY TOTALPRICE SUM_OVER_RESULT
---------- -------- ---------- ---------------
1998-07-24     1730     204656         1251844

1998-07-24    17056     289620         1251844

1998-07-24    19937      26354         1251844
```

```
1998-07-25     2400     150304         1251844

1998-07-25    11204      27165         1251844

1998-07-25    11938     191054         1251844

1998-07-26    35655      13698         1251844

1998-07-26    53377     185930         1251844

1998-07-26    55010     163063         1251844      partition ❶

------------------------------------------------------------------------


9 rows selected.
```

<window order clause>

使用ORDER BY基于<sort specification list>在分区内指定数据排序方式

- <ordering specification>
  - 可以指定升序或降序
    - ASC
    - DESC
    - 未指定时默认值为ASC
- <null ordering>
  - 可以指定NULL值和非NULL值的顺序
    - NULLS FIRST
    - NULLS LAST
    - 未指定时默认值为NULLS LAST
- <window frame clause>的<window frame units>为RANGE时
  - 描述offset PRECEDING或offset FOLLOWING时仅可指定一个sort key

        - ORDER BY I1 RANGE 3 PRECEDING
        - ORDER BY I1 RANGE BETWEEN CURRENT ROW AND 3 FOLLOWING
    - 除此以外的情况可以指定多个sort key
        - ORDER BY I1, I2 RANGE UNBOUNDED PRECEDING
        - ORDER BY I1, I2 RANGE CURRENT ROW
        - ORDER BY I1, i2 RANGE BETWEEN CURRENT ROW AND UNBOUNDED FOLLOWING
        - ORDER BY I1, i2 RANGE BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING

<window frame clause>

指定作为window function的对象记录范围的window frame

window frame是与查询的各row (current row)相关的记录范围

window frame的对象是当前分区内已排序的记录

window frame可以定义要应用的单位(ROWS/ RANGE/ GROUPS)开始和结束点以及要排除的记录

省略时应用RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW

- RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW
    - 开始点：从分区开始记录开始
    - 结束点：到当前记录的所有peer记录为止
- peer: window ORDER BY子句的排序相同的记录

```
gSQL>
SELECT orderdate AS O_DATE,
```

```
       orderkey AS O_KEY,
       custkey,
       totalprice,
       SUM( totalprice ) OVER( PARTITION BY orderdate
                               ORDER BY custkey ) AS SUM_OVER_RESULT
  FROM orders;

O_DATE     O_KEY CUSTKEY TOTALPRICE SUM_OVER_RESULT
---------- ----- ------- ---------- ---------------
2000-01-01   101 3088161     180000          180000
2000-01-01   102 3088163      42000          269000   <- peer ( custkey值
相同 )
2000-01-01   103 3088163      47000          269000      peer
2000-01-01   104 3088165     217000          486000
2000-01-01   105 3088167     108000          734000   <- peer ( custkey值
相同 )
2000-01-01   106 3088167      60000          734000      peer
2000-01-01   107 3088167      80000          734000      peer
...
15 rows selected.
```

- 省略<window frame clause>时

```
gSQL>
SELECT orderdate AS O_DATE,
       orderkey AS O_KEY,
```

```
       custkey,
       totalprice,
       SUM( totalprice ) OVER( PARTITION BY orderdate
                               ORDER BY custkey ) AS SUM_OVER_RESULT
  FROM orders;


O_DATE     O_KEY CUSTKEY TOTALPRICE    SUM_OVER_RESULT
---------- ----- ------- ----------    ---------------
2000-01-01   101 3088161     180000 ❶      180000 ❶
2000-01-01   102 3088163      42000 ❷      269000 ❶+❷+❸
2000-01-01   103 3088163      47000 ❸      269000 ❶+❷+❸
2000-01-01   104 3088165     217000 ❹      486000 ❶+❷+❸+❹
2000-01-01   105 3088167     108000 ❺      734000 ❶+❷+❸+❹+❺+❻+❼
2000-01-01   106 3088167      60000 ❻      734000 ❶+❷+❸+❹+❺+❻+❼
2000-01-01   107 3088167      80000 ❼      734000 ❶+❷+❸+❹+❺+❻+❼
2000-01-01   108 3088169      32000 ❽      766000 ❶+❷+❸+❹+❺+❻+❼+❽
2000-01-01   109 3088170      30000 ❾      816000
❶+❷+❸+❹+❺+❻+❼+❽+❾+❿
2000-01-01   110 3088170      20000 ❿      816000
❶+❷+❸+❹+❺+❻+❼+❽+❾+❿
-------------------------------------------------------------------------
2000-03-03   301 3088161     180000 ❶      222000 ❶+❷
2000-03-03   302 3088161      42000 ❷      222000 ❶+❷
2000-03-03   303 3088165      47000 ❸      269000 ❶+❷+❸
2000-03-03   304 3088167     217000 ❹      594000 ❶+❷+❸+❹+❺
```

```
2000-03-03    305 3088167      108000 ❺       594000 ❶+❷+❸+❹+❺


15 rows selected.
```

<window frame units>

ROWS/ RANGE/ GROUPS为应用window frame的单位

<window frame extent>

定义window frame start （开始点）和window frame end（结束点）

- UNBOUNDED PRECEDING
  - 从分区开始记录开始
- UNBOUNDED FOLLOWING
  - 到分区最后记录为止
- CURRENT ROW
  - 为<window frame units> ROWS时
    - 当前记录
  - 为<window frame units> RANGE/ GROUPS时
    - 当前记录的所有peer记录
- offset PRECEDING/ offset FOLLOWING
  - 为<window frame units> ROWS时
    - 当前记录的前/后offset记录数量范围
  - 为<window frame units> GROUPS时
    - 当前记录的group前/后offset group数量范围
  - 为<window frame units> RANGE时

    - 当前记录的前/后offset值的范围
    - 以ASC方式排列ORDER BY column时
    - ... offset PRECEDING → (当前记录的sort key value - offset) 以上的value的记录
    - ... offset FOLLOWING → (当前记录的sort key value + offset) 以下的value的记录
    - 以DESC 方式排列ORDER BY column时
    - ... offset PRECEDING → (当前记录的sort key value + offset)以上的value的记录
    - ... offset FOLLOWING → (当前记录的sort key value - offset)以下的value的记录
- 详细内容参考 **<window frame extent>使用示例**

<window frame exclusion>

定义要在window frame中排除的记录

- EXCLUDE CURRENT ROW: 排除当前记录
- EXCLUDE GROUP: 排除当前记录和所有peer记录
- EXCLUDE TIES: 保持当前记录排除所有peer记录
- EXCLUDE NO OTHERS: 不排除任何记录
- 未指定时默认值为EXCLUDE NO OTHERS
- 详细内容参考 **<window frame exclusion> 使用示例**

<window frame extent>使用示例

- BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW

```
# ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW


gSQL>
```

```
SELECT orderdate AS O_DATE,
       orderkey AS O_KEY,
       custkey,
       totalprice,
       SUM( totalprice ) OVER( PARTITION BY orderdate
                               ORDER BY custkey
                               ROWS BETWEEN UNBOUNDED PRECEDING
                                        AND CURRENT ROW ) AS
SUM_OVER_RESULT
  FROM orders;


O_DATE     O_KEY CUSTKEY TOTALPRICE    SUM_OVER_RESULT
---------- ----- ------- ----------    ---------------
2000-01-01   101 3088161     180000 ❶      180000 ❶
2000-01-01   102 3088163      42000 ❷      222000 ❶+❷
2000-01-01   103 3088163      47000 ❸      269000 ❶+❷+❸
2000-01-01   104 3088165     217000 ❹      486000 ❶+❷+❸+❹
2000-01-01   105 3088167     108000 ❺      594000 ❶+❷+❸+❹+❺
2000-01-01   106 3088167      60000 ❻      654000 ❶+❷+❸+❹+❺+❻
2000-01-01   107 3088167      80000 ❼      734000 ❶+❷+❸+❹+❺+❻+❼
2000-01-01   108 3088169      32000 ❽      766000 ❶+❷+❸+❹+❺+❻+❼+❽
2000-01-01   109 3088170      30000 ❾      796000
❶+❷+❸+❹+❺+❻+❼+❽+❾
2000-01-01   110 3088170      20000 ❿      816000
❶+❷+❸+❹+❺+❻+❼+❽+❾+❿
```

```
------------------------------------------------------------------------

2000-03-03   301 3088161     180000 ❶      180000 ❶

2000-03-03   302 3088161      42000 ❷      222000 ❶+❷

2000-03-03   303 3088165      47000 ❸      269000 ❶+❷+❸

2000-03-03   304 3088167     217000 ❹      486000 ❶+❷+❸+❹

2000-03-03   305 3088167     108000 ❺      594000 ❶+❷+❸+❹+❺


15 rows selected.
```

```
# RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW


gSQL>
SELECT orderdate AS O_DATE,
       orderkey AS O_KEY,
       custkey,
       totalprice,
       SUM( totalprice ) OVER( PARTITION BY orderdate
                               ORDER BY custkey
                               RANGE BETWEEN UNBOUNDED PRECEDING
                                         AND CURRENT ROW ) AS
SUM_OVER_RESULT
  FROM orders;


O_DATE     O_KEY CUSTKEY TOTALPRICE    SUM_OVER_RESULT
```

```
---------- ----- ------- ----------    ---------------

2000-01-01   101 3088161     180000 ❶      180000 ❶

2000-01-01   102 3088163      42000 ❷      269000 ❶+❷+❸

2000-01-01   103 3088163      47000 ❸      269000 ❶+❷+❸

2000-01-01   104 3088165     217000 ❹      486000 ❶+❷+❸+❹

2000-01-01   105 3088167     108000 ❺      734000 ❶+❷+❸+❹+❺+❻+❼

2000-01-01   106 3088167      60000 ❻      734000 ❶+❷+❸+❹+❺+❻+❼

2000-01-01   107 3088167      80000 ❼      734000 ❶+❷+❸+❹+❺+❻+❼

2000-01-01   108 3088169      32000 ❽      766000 ❶+❷+❸+❹+❺+❻+❼+❽

2000-01-01   109 3088170      30000 ❾      816000

❶+❷+❸+❹+❺+❻+❼+❽+❾+❿

2000-01-01   110 3088170      20000 ❿      816000

❶+❷+❸+❹+❺+❻+❼+❽+❾+❿

-------------------------------------------------------------------------

2000-03-03   301 3088161     180000 ❶      222000 ❶+❷

2000-03-03   302 3088161      42000 ❷      222000 ❶+❷

2000-03-03   303 3088165      47000 ❸      269000 ❶+❷+❸

2000-03-03   304 3088167     217000 ❹      594000 ❶+❷+❸+❹+❺

2000-03-03   305 3088167     108000 ❺      594000 ❶+❷+❸+❹+❺


15 rows selected.
```

```
# GROUPS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW
```

```
gSQL>
SELECT orderdate AS O_DATE,
       orderkey AS O_KEY,
       custkey,
       totalprice,
       SUM( totalprice ) OVER( PARTITION BY orderdate
                               ORDER BY custkey
                               GROUPS BETWEEN UNBOUNDED PRECEDING
                                         AND CURRENT ROW ) AS
SUM_OVER_RESULT
  FROM orders;

O_DATE     O_KEY CUSTKEY TOTALPRICE    SUM_OVER_RESULT
---------- ----- ------- ----------    ---------------
2000-01-01   101 3088161     180000 ❶      180000 ❶
2000-01-01   102 3088163      42000 ❷      269000 ❶+❷+❸
2000-01-01   103 3088163      47000 ❸      269000 ❶+❷+❸
2000-01-01   104 3088165     217000 ❹      486000 ❶+❷+❸+❹
2000-01-01   105 3088167     108000 ❺      734000 ❶+❷+❸+❹+❺+❻+❼
2000-01-01   106 3088167      60000 ❻      734000 ❶+❷+❸+❹+❺+❻+❼
2000-01-01   107 3088167      80000 ❼      734000 ❶+❷+❸+❹+❺+❻+❼
2000-01-01   108 3088169      32000 ❽      766000 ❶+❷+❸+❹+❺+❻+❼+❽
2000-01-01   109 3088170      30000 ❾      816000
❶+❷+❸+❹+❺+❻+❼+❽+❾+❿
2000-01-01   110 3088170      20000 ❿      816000
```

```
❶+❷+❸+❹+❺+❻+❼+❽+❾+❿

------------------------------------------------------------------------

2000-03-03   301 3088161     180000 ❶      222000 ❶+❷

2000-03-03   302 3088161      42000 ❷      222000 ❶+❷

2000-03-03   303 3088165      47000 ❸      269000 ❶+❷+❸

2000-03-03   304 3088167     217000 ❹      594000 ❶+❷+❸+❹+❺

2000-03-03   305 3088167     108000 ❺      594000 ❶+❷+❸+❹+❺


15 rows selected.
```

- BETWEEN offset PRECEDING AND offset FOLLOWING

```
# ROWS BETWEEN 1 PRECEDING AND 2 FOLLOWING


gSQL>
SELECT orderdate AS O_DATE,
       orderkey AS O_KEY,
       custkey,
       totalprice,
       SUM( totalprice ) OVER( PARTITION BY orderdate
                               ORDER BY custkey
                               ROWS BETWEEN 1 PRECEDING
                                        AND 2 FOLLOWING ) AS
SUM_OVER_RESULT
  FROM orders;
```

```
O_DATE     O_KEY CUSTKEY TOTALPRICE    SUM_OVER_RESULT

---------- ----- ------- ----------    ---------------

2000-01-01   101 3088161     180000 ❶     269000 ❶+❷+❸

2000-01-01   102 3088163      42000 ❷     486000 ❶+❷+❸+❹

2000-01-01   103 3088163      47000 ❸     414000 ❷+❸+❹+❺

2000-01-01   104 3088165     217000 ❹     432000 ❸+❹+❺+❻

2000-01-01   105 3088167     108000 ❺     465000 ❹+❺+❻+❼

2000-01-01   106 3088167      60000 ❻     280000 ❺+❻+❼+❽

2000-01-01   107 3088167      80000 ❼     202000 ❻+❼+❽+❾

2000-01-01   108 3088169      32000 ❽     162000 ❼+❽+❾+❿

2000-01-01   109 3088170      30000 ❾      82000 ❽+❾+❿

2000-01-01   110 3088170      20000 ❿      50000 ❾+❿

------------------------------------------------------------------------

2000-03-03   301 3088161     180000 ❶     269000 ❶+❷+❸

2000-03-03   302 3088161      42000 ❷     486000 ❶+❷+❸+❹

2000-03-03   303 3088165      47000 ❸     414000 ❷+❸+❹+❺

2000-03-03   304 3088167     217000 ❹     372000 ❸+❹+❺

2000-03-03   305 3088167     108000 ❺     325000 ❹+❺


15 rows selected.
```

```
# RANGE BETWEEN 1 PRECEDING AND 2 FOLLOWING


##########################################################
```

```
# 使用ASC方式排列ORDER BY column时

#######################################################
```

- 1 PRECEDING

  --> (current row的sortkey value - 1) 以上的value

       = *(custkey - 1) 以上的value*

- 2 FOLLOWING

  --> (current row的sortkey value + 2) 以下的value

       = *(custkey + 2) 以下的value*

```
gSQL>
SELECT orderdate AS O_DATE,
       orderkey AS O_KEY,
       custkey,
       totalprice,
       SUM( totalprice ) OVER( PARTITION BY orderdate
                               ORDER BY custkey
                               RANGE BETWEEN 1 PRECEDING
                                         AND 2 FOLLOWING ) AS
SUM_OVER_RESULT
  FROM orders;

O_DATE     O_KEY CUSTKEY TOTALPRICE    SUM_OVER_RESULT
---------- ----- ------- ----------    ---------------
```

```
2000-01-01   101 3088161     180000 ❶      269000 ❶+❷+❸

2000-01-01   102 3088163      42000 ❷      306000 ❷+❸+❹

2000-01-01   103 3088163      47000 ❸      306000 ❷+❸+❹

2000-01-01   104 3088165     217000 ❹      465000 ❹+❺+❻+❼

2000-01-01   105 3088167     108000 ❺      280000 ❺+❻+❼+❽

2000-01-01   106 3088167      60000 ❻      280000 ❺+❻+❼+❽

2000-01-01   107 3088167      80000 ❼      280000 ❺+❻+❼+❽

2000-01-01   108 3088169      32000 ❽       82000 ❽+❾+❿

2000-01-01   109 3088170      30000 ❾       82000 ❽+❾+❿

2000-01-01   110 3088170      20000 ❿       82000 ❽+❾+❿

------------------------------------------------------------------------

2000-03-03   301 3088161     180000 ❶      222000 ❶+❷

2000-03-03   302 3088161      42000 ❷      222000 ❶+❷

2000-03-03   303 3088165      47000 ❸      372000 ❸+❹+❺

2000-03-03   304 3088167     217000 ❹      325000 ❹+❺

2000-03-03   305 3088167     108000 ❺      325000 ❹+❺


15 rows selected.



#########################################################

# 使用DESC方式排列ORDER BY column时

#########################################################
```

- 1 PRECEDING

--> (current row的sortkey value + 1) 以下的value

= *(custkey + 1) 以下的value*

- 2 FOLLOWING

--> (current row的sortkey value - 2) 以上的value

= *(custkey - 2) 以上的value*

```
gSQL>
SELECT orderdate AS O_DATE,
       orderkey AS O_KEY,
       custkey,
       totalprice,
       SUM( totalprice ) OVER( PARTITION BY orderdate
                               ORDER BY custkey DESC
                               RANGE BETWEEN 1 PRECEDING
                                         AND 2 FOLLOWING ) AS
SUM_OVER_RESULT
  FROM orders;

O_DATE     O_KEY CUSTKEY TOTALPRICE    SUM_OVER_RESULT
---------- ----- ------- ----------    ---------------
2000-01-01   109 3088170      30000 ❶       82000 ❶+❷+❸
2000-01-01   110 3088170      20000 ❷       82000 ❶+❷+❸
2000-01-01   108 3088169      32000 ❸      330000 ❶+❷+❸+❹+❺+❻
2000-01-01   105 3088167     108000 ❹      465000 ❹+❺+❻+❼
```

```
2000-01-01   106 3088167      60000 ❺      465000 ❹+❺+❻+❼

2000-01-01   107 3088167      80000 ❻      465000 ❹+❺+❻+❼

2000-01-01   104 3088165     217000 ❼      306000 ❼+❽+❾

2000-01-01   102 3088163      42000 ❽      269000 ❽+❾+❿

2000-01-01   103 3088163      47000 ❾      269000 ❽+❾+❿

2000-01-01   101 3088161     180000 ❿      180000 ❿

-------------------------------------------------------------------------

2000-03-03   304 3088167     217000 ❶      372000 ❶+❷+❸

2000-03-03   305 3088167     108000 ❷      372000 ❶+❷+❸

2000-03-03   303 3088165      47000 ❸       47000 ❸

2000-03-03   301 3088161     180000 ❹      222000 ❹+❺

2000-03-03   302 3088161      42000 ❺      222000 ❹+❺


15 rows selected.
```

```
# GROUPS BETWEEN 1 PRECEDING AND 2 FOLLOWING


gSQL>
SELECT orderdate AS O_DATE,
       orderkey AS O_KEY,
       custkey,
       totalprice,
       SUM( totalprice ) OVER( PARTITION BY orderdate
                               ORDER BY custkey
```

```
                                GROUPS BETWEEN 1 PRECEDING

                                            AND 2 FOLLOWING ) AS

SUM_OVER_RESULT

  FROM orders;


O_DATE     O_KEY CUSTKEY TOTALPRICE    SUM_OVER_RESULT

---------- ----- ------- ----------    ---------------

2000-01-01   101 3088161     180000 ❶      486000 ❶+❷+❸+❹

2000-01-01   102 3088163      42000 ❷      734000 ❶+❷+❸+❹+❺+❻+❼

2000-01-01   103 3088163      47000 ❸      734000 ❶+❷+❸+❹+❺+❻+❼

2000-01-01   104 3088165     217000 ❹      586000 ❷+❸+❹+❺+❻+❼+❽

2000-01-01   105 3088167     108000 ❺      547000 ❹+❺+❻+❼+❽+❾+❿

2000-01-01   106 3088167      60000 ❻      547000 ❹+❺+❻+❼+❽+❾+❿

2000-01-01   107 3088167      80000 ❼      547000 ❹+❺+❻+❼+❽+❾+❿

2000-01-01   108 3088169      32000 ❽      330000 ❺+❻+❼+❽+❾+❿

2000-01-01   109 3088170      30000 ❾       82000 ❽+❾+❿

2000-01-01   110 3088170      20000 ❿       82000 ❽+❾+❿

-------------------------------------------------------------------------

2000-03-03   301 3088161     180000 ❶      594000 ❶+❷+❸+❹+❺

2000-03-03   302 3088161      42000 ❷      594000 ❶+❷+❸+❹+❺

2000-03-03   303 3088165      47000 ❸      594000 ❶+❷+❸+❹+❺

2000-03-03   304 3088167     217000 ❹      372000 ❸+❹+❺

2000-03-03   305 3088167     108000 ❺      372000 ❸+❹+❺


15 rows selected.
```

- BETWEEN CURRENT ROW AND UNBOUNDED FOLLOWING

```
# ROWS BETWEEN CURRENT ROW AND UNBOUNDED FOLLOWING


gSQL>
SELECT orderdate AS O_DATE,
       orderkey AS O_KEY,
       custkey,
       totalprice,
       SUM( totalprice ) OVER( PARTITION BY orderdate
                               ORDER BY custkey
                               ROWS BETWEEN CURRENT ROW
                                        AND UNBOUNDED FOLLOWING ) AS
SUM_OVER_RESULT
  FROM orders;


O_DATE     O_KEY CUSTKEY TOTALPRICE SUM_OVER_RESULT
---------- ----- ------- ---------- ---------------
2000-01-01   101 3088161     180000          816000
2000-01-01   102 3088163      42000          636000
2000-01-01   103 3088163      47000          594000
2000-01-01   104 3088165     217000          547000
2000-01-01   105 3088167     108000          330000
2000-01-01   106 3088167      60000          222000
2000-01-01   107 3088167      80000          162000
```

```
2000-01-01   108 3088169      32000           82000

2000-01-01   109 3088170      30000           50000

2000-01-01   110 3088170      20000           20000

2000-03-03   301 3088161     180000          594000

2000-03-03   302 3088161      42000          414000

2000-03-03   303 3088165      47000          372000

2000-03-03   304 3088167     217000          325000

2000-03-03   305 3088167     108000          108000


15 rows selected.
```

```
# RANGE BETWEEN CURRENT ROW AND UNBOUNDED FOLLOWING


gSQL>
SELECT orderdate AS O_DATE,
       orderkey AS O_KEY,
       custkey,
       totalprice,
       SUM( totalprice ) OVER( PARTITION BY orderdate
                               ORDER BY custkey
                               RANGE BETWEEN CURRENT ROW
                                         AND UNBOUNDED FOLLOWING ) AS
SUM_OVER_RESULT
  FROM orders;
```

```
O_DATE     O_KEY CUSTKEY TOTALPRICE SUM_OVER_RESULT
---------- ----- ------- ---------- ---------------
2000-01-01   101 3088161     180000          816000
2000-01-01   102 3088163      42000          636000
2000-01-01   103 3088163      47000          636000
2000-01-01   104 3088165     217000          547000
2000-01-01   105 3088167     108000          330000
2000-01-01   106 3088167      60000          330000
2000-01-01   107 3088167      80000          330000
2000-01-01   108 3088169      32000           82000
2000-01-01   109 3088170      30000           50000
2000-01-01   110 3088170      20000           50000
2000-03-03   301 3088161     180000          594000
2000-03-03   302 3088161      42000          594000
2000-03-03   303 3088165      47000          372000
2000-03-03   304 3088167     217000          325000
2000-03-03   305 3088167     108000          325000

15 rows selected.
```

```
# GROUPS BETWEEN CURRENT ROW AND UNBOUNDED FOLLOWING

gSQL>
```

```
SELECT orderdate AS O_DATE,
       orderkey AS O_KEY,
       custkey,
       totalprice,
       SUM( totalprice ) OVER( PARTITION BY orderdate
                               ORDER BY custkey
                               GROUPS BETWEEN CURRENT ROW
                                          AND UNBOUNDED FOLLOWING ) AS
SUM_OVER_RESULT
  FROM orders;


O_DATE     O_KEY CUSTKEY TOTALPRICE SUM_OVER_RESULT
---------- ----- ------- ---------- ---------------
2000-01-01   101 3088161     180000          816000
2000-01-01   102 3088163      42000          636000
2000-01-01   103 3088163      47000          636000
2000-01-01   104 3088165     217000          547000
2000-01-01   105 3088167     108000          330000
2000-01-01   106 3088167      60000          330000
2000-01-01   107 3088167      80000          330000
2000-01-01   108 3088169      32000           82000
2000-01-01   109 3088170      30000           50000
2000-01-01   110 3088170      20000           50000
2000-03-03   301 3088161     180000          594000
2000-03-03   302 3088161      42000          594000
```

```
2000-03-03    303 3088165      47000          372000

2000-03-03    304 3088167     217000          325000

2000-03-03    305 3088167     108000          325000


15 rows selected.
```

<window frame exclusion>使用示例

- EXCLUDE CURRENT ROW

```
# ROWS


gSQL>
SELECT orderdate AS O_DATE,
       orderkey AS O_KEY,
       custkey,
       totalprice,
       SUM( totalprice ) OVER( PARTITION BY orderdate
                               ORDER BY custkey
                               ROWS BETWEEN UNBOUNDED PRECEDING
                                        AND CURRENT ROW
                               EXCLUDE CURRENT ROW ) AS SUM_OVER_RESULT
  FROM orders;


O_DATE     O_KEY CUSTKEY TOTALPRICE    SUM_OVER_RESULT
---------- ----- ------- ----------    ---------------
```

```
2000-01-01   101 3088161     180000 ❶       null
2000-01-01   102 3088163      42000 ❷     180000 ❶
2000-01-01   103 3088163      47000 ❸     222000 ❶+❷
2000-01-01   104 3088165     217000 ❹     269000 ❶+❷+❸
2000-01-01   105 3088167     108000 ❺     486000 ❶+❷+❸+❹
2000-01-01   106 3088167      60000 ❻     594000 ❶+❷+❸+❹+❺
2000-01-01   107 3088167      80000 ❼     654000 ❶+❷+❸+❹+❺+❻
2000-01-01   108 3088169      32000 ❽     734000 ❶+❷+❸+❹+❺+❻+❼
2000-01-01   109 3088170      30000 ❾     766000 ❶+❷+❸+❹+❺+❻+❼+❽
2000-01-01   110 3088170      20000 ❿     796000
❶+❷+❸+❹+❺+❻+❼+❽+❾
------------------------------------------------------------------------
2000-03-03   301 3088161     180000 ❶       null
2000-03-03   302 3088161      42000 ❷     180000 ❶
2000-03-03   303 3088165      47000 ❸     222000 ❶+❷
2000-03-03   304 3088167     217000 ❹     269000 ❶+❷+❸
2000-03-03   305 3088167     108000 ❺     486000 ❶+❷+❸+❹


15 rows selected.
```

```
# RANGE


gSQL>
SELECT orderdate AS O_DATE,
```

```
       orderkey AS O_KEY,
       custkey,
       totalprice,
       SUM( totalprice ) OVER( PARTITION BY orderdate
                               ORDER BY custkey
                               RANGE BETWEEN UNBOUNDED PRECEDING
                                         AND CURRENT ROW
                               EXCLUDE CURRENT ROW ) AS SUM_OVER_RESULT
  FROM orders;


O_DATE     O_KEY CUSTKEY TOTALPRICE    SUM_OVER_RESULT
---------- ----- ------- ----------    ---------------
2000-01-01   101 3088161     180000 ❶        null
2000-01-01   102 3088163      42000 ❷      227000 ❶+❸
2000-01-01   103 3088163      47000 ❸      222000 ❶+❷
2000-01-01   104 3088165     217000 ❹      269000 ❶+❷+❸
2000-01-01   105 3088167     108000 ❺      626000 ❶+❷+❸+❹+❻+❼
2000-01-01   106 3088167      60000 ❻      674000 ❶+❷+❸+❹+❺+❼
2000-01-01   107 3088167      80000 ❼      654000 ❶+❷+❸+❹+❺+❻
2000-01-01   108 3088169      32000 ❽      734000 ❶+❷+❸+❹+❺+❻+❼
2000-01-01   109 3088170      30000 ❾      786000
❶+❷+❸+❹+❺+❻+❼+❽+❿
2000-01-01   110 3088170      20000 ❿      796000
❶+❷+❸+❹+❺+❻+❼+❽+❾
--------------------------------------------------------------------------
```

```
2000-03-03    301 3088161     180000 ❶       42000 ❷
2000-03-03    302 3088161      42000 ❷      180000 ❶
2000-03-03    303 3088165      47000 ❸      222000 ❶+❷
2000-03-03    304 3088167     217000 ❹      377000 ❶+❷+❸+❺
2000-03-03    305 3088167     108000 ❺      486000 ❶+❷+❸+❹

15 rows selected.
```

```
# GROUPS

gSQL>
SELECT orderdate AS O_DATE,
       orderkey AS O_KEY,
       custkey,
       totalprice,
       SUM( totalprice ) OVER( PARTITION BY orderdate
                               ORDER BY custkey
                               GROUPS BETWEEN UNBOUNDED PRECEDING
                                          AND CURRENT ROW
                               EXCLUDE CURRENT ROW ) AS SUM_OVER_RESULT
  FROM orders;

O_DATE     O_KEY CUSTKEY TOTALPRICE    SUM_OVER_RESULT
---------- ----- ------- ----------    ---------------
```

```
2000-01-01   101 3088161     180000 ❶       null
2000-01-01   102 3088163     42000 ❷      227000 ❶+❸
2000-01-01   103 3088163     47000 ❸      222000 ❶+❷
2000-01-01   104 3088165     217000 ❹      269000 ❶+❷+❸
2000-01-01   105 3088167     108000 ❺      626000 ❶+❷+❸+❹+❻+❼
2000-01-01   106 3088167     60000 ❻      674000 ❶+❷+❸+❹+❺+❼
2000-01-01   107 3088167     80000 ❼      654000 ❶+❷+❸+❹+❺+❻
2000-01-01   108 3088169      32000 ❽      734000 ❶+❷+❸+❹+❺+❻+❼
2000-01-01   109 3088170     30000 ❾      786000
❶+❷+❸+❹+❺+❻+❼+❽+❿
2000-01-01   110 3088170     20000 ❿      796000
❶+❷+❸+❹+❺+❻+❼+❽+❾
------------------------------------------------------------------------
2000-03-03   301 3088161     180000 ❶       42000 ❷
2000-03-03   302 3088161     42000 ❷      180000 ❶
2000-03-03   303 3088165      47000 ❸      222000 ❶+❷
2000-03-03   304 3088167     217000 ❹      377000 ❶+❷+❸+❺
2000-03-03   305 3088167     108000 ❺      486000 ❶+❷+❸+❹


15 rows selected.
```

- EXCLUDE GROUP

```
# ROWS


gSQL>
```

```
SELECT orderdate AS O_DATE,
       orderkey AS O_KEY,
       custkey,
       totalprice,
       SUM( totalprice ) OVER( PARTITION BY orderdate
                               ORDER BY custkey
                               ROWS BETWEEN UNBOUNDED PRECEDING
                                        AND CURRENT ROW
                               EXCLUDE GROUP ) AS SUM_OVER_RESULT
  FROM orders;


O_DATE     O_KEY CUSTKEY TOTALPRICE    SUM_OVER_RESULT
---------- ----- ------- ----------    ---------------
2000-01-01   101 3088161     180000 ❶       null
2000-01-01   102 3088163      42000 ❷     180000 ❶
2000-01-01   103 3088163      47000 ❸     180000 ❶
2000-01-01   104 3088165     217000 ❹     269000 ❶+❷+❸
2000-01-01   105 3088167     108000 ❺     486000 ❶+❷+❸+❹
2000-01-01   106 3088167      60000 ❻     486000 ❶+❷+❸+❹
2000-01-01   107 3088167      80000 ❼     486000 ❶+❷+❸+❹
2000-01-01   108 3088169      32000 ❽     734000 ❶+❷+❸+❹+❺+❻+❼
2000-01-01   109 3088170      30000 ❾     766000 ❶+❷+❸+❹+❺+❻+❼+❽
2000-01-01   110 3088170      20000 ❿     766000 ❶+❷+❸+❹+❺+❻+❼+❽
-------------------------------------------------------------------------
2000-03-03   301 3088161     180000 ❶       null
```

```
2000-03-03    302 3088161      42000 ❷        null

2000-03-03    303 3088165      47000 ❸      222000 ❶+❷

2000-03-03    304 3088167     217000 ❹      269000 ❶+❷+❸

2000-03-03    305 3088167     108000 ❺      269000 ❶+❷+❸


15 rows selected.
```

```
# RANGE


gSQL>
SELECT orderdate AS O_DATE,
       orderkey AS O_KEY,
       custkey,
       totalprice,
       SUM( totalprice ) OVER( PARTITION BY orderdate
                               ORDER BY custkey
                               RANGE BETWEEN UNBOUNDED PRECEDING
                                         AND CURRENT ROW
                               EXCLUDE GROUP ) AS SUM_OVER_RESULT
  FROM orders;


O_DATE     O_KEY CUSTKEY TOTALPRICE    SUM_OVER_RESULT
---------- ----- ------- ----------    ---------------
2000-01-01   101 3088161     180000 ❶       null
```

```
2000-01-01   102 3088163      42000 ❷      180000 ❶

2000-01-01   103 3088163      47000 ❸      180000 ❶

2000-01-01   104 3088165     217000 ❹      269000 ❶+❷+❸

2000-01-01   105 3088167     108000 ❺      486000 ❶+❷+❸+❹

2000-01-01   106 3088167      60000 ❻      486000 ❶+❷+❸+❹

2000-01-01   107 3088167      80000 ❼      486000 ❶+❷+❸+❹

2000-01-01   108 3088169      32000 ❽      734000 ❶+❷+❸+❹+❺+❻+❼

2000-01-01   109 3088170      30000 ❾      766000 ❶+❷+❸+❹+❺+❻+❼+❽

2000-01-01   110 3088170      20000 ❿      766000 ❶+❷+❸+❹+❺+❻+❼+❽

-------------------------------------------------------------------------

2000-03-03   301 3088161     180000 ❶        null

2000-03-03   302 3088161      42000 ❷        null

2000-03-03   303 3088165      47000 ❸      222000 ❶+❷

2000-03-03   304 3088167     217000 ❹      269000 ❶+❷+❸

2000-03-03   305 3088167     108000 ❺      269000 ❶+❷+❸


15 rows selected.
```

```
# GROUPS


gSQL> SELECT orderdate AS O_DATE,
       orderkey AS O_KEY,
       custkey,
       totalprice,
```

```
       SUM( totalprice ) OVER( PARTITION BY orderdate
                               ORDER BY custkey
                               GROUPS BETWEEN UNBOUNDED PRECEDING
                                         AND CURRENT ROW
                               EXCLUDE GROUP ) AS SUM_OVER_RESULT
  FROM orders;


O_DATE     O_KEY CUSTKEY TOTALPRICE    SUM_OVER_RESULT
---------- ----- ------- ----------    ---------------
2000-01-01   101 3088161     180000 ❶       null
2000-01-01   102 3088163      42000 ❷     180000 ❶
2000-01-01   103 3088163      47000 ❸     180000 ❶
2000-01-01   104 3088165     217000 ❹     269000 ❶+❷+❸
2000-01-01   105 3088167     108000 ❺     486000 ❶+❷+❸+❹
2000-01-01   106 3088167      60000 ❻     486000 ❶+❷+❸+❹
2000-01-01   107 3088167      80000 ❼     486000 ❶+❷+❸+❹
2000-01-01   108 3088169      32000 ❽     734000 ❶+❷+❸+❹+❺+❻+❼
2000-01-01   109 3088170      30000 ❾     766000 ❶+❷+❸+❹+❺+❻+❼+❽
2000-01-01   110 3088170      20000 ❿     766000 ❶+❷+❸+❹+❺+❻+❼+❽
-------------------------------------------------------------------------
2000-03-03   301 3088161     180000 ❶       null
2000-03-03   302 3088161      42000 ❷       null
2000-03-03   303 3088165      47000 ❸     222000 ❶+❷
2000-03-03   304 3088167     217000 ❹     269000 ❶+❷+❸
2000-03-03   305 3088167     108000 ❺     269000 ❶+❷+❸
```

```
15 rows selected.
```

- EXCLUDE TIES

```
# ROWS


gSQL>
SELECT orderdate AS O_DATE,
       orderkey AS O_KEY,
       custkey,
       totalprice,
       SUM( totalprice ) OVER( PARTITION BY orderdate
                               ORDER BY custkey
                               ROWS BETWEEN UNBOUNDED PRECEDING
                                        AND CURRENT ROW
                               EXCLUDE TIES ) AS SUM_OVER_RESULT
  FROM orders;


O_DATE     O_KEY CUSTKEY TOTALPRICE    SUM_OVER_RESULT
---------- ----- ------- ----------    ---------------
2000-01-01   101 3088161     180000 ❶      180000 ❶
2000-01-01   102 3088163      42000 ❷      222000 ❶+❷
2000-01-01   103 3088163      47000 ❸      227000 ❶+❸
2000-01-01   104 3088165     217000 ❹      486000 ❶+❷+❸+❹
2000-01-01   105 3088167     108000 ❺      594000 ❶+❷+❸+❹+❺
```

```
2000-01-01   106 3088167      60000 ❻      546000 ❶+❷+❸+❹+❻

2000-01-01   107 3088167      80000 ❼      566000 ❶+❷+❸+❹+❼

2000-01-01   108 3088169      32000 ❽      766000 ❶+❷+❸+❹+❺+❻+❼+❽

2000-01-01   109 3088170      30000 ❾      796000
❶+❷+❸+❹+❺+❻+❼+❽+❾

2000-01-01   110 3088170      20000 ❿      786000
❶+❷+❸+❹+❺+❻+❼+❽+❿

------------------------------------------------------------------------

2000-03-03   301 3088161     180000 ❶      180000 ❶

2000-03-03   302 3088161      42000 ❷       42000 ❷

2000-03-03   303 3088165      47000 ❸      269000 ❶+❷+❸

2000-03-03   304 3088167     217000 ❹      486000 ❶+❷+❸+❹

2000-03-03   305 3088167     108000 ❺      377000 ❶+❷+❸+❺


15 rows selected.
```

```
# RANGE


gSQL>
SELECT orderdate AS O_DATE,
       orderkey AS O_KEY,
       custkey,
       totalprice,
       SUM( totalprice ) OVER( PARTITION BY orderdate
```

```
                                ORDER BY custkey
                                RANGE BETWEEN UNBOUNDED PRECEDING
                                          AND CURRENT ROW
                                EXCLUDE TIES ) AS SUM_OVER_RESULT
  FROM orders;


O_DATE     O_KEY CUSTKEY TOTALPRICE    SUM_OVER_RESULT
---------- ----- ------- ----------    ---------------
2000-01-01   101 3088161     180000 ❶      180000 ❶
2000-01-01   102 3088163      42000 ❷      222000 ❶+❷
2000-01-01   103 3088163      47000 ❸      227000 ❶+❸
2000-01-01   104 3088165     217000 ❹      486000 ❶+❷+❸+❹
2000-01-01   105 3088167     108000 ❺      594000 ❶+❷+❸+❹+❺
2000-01-01   106 3088167      60000 ❻      546000 ❶+❷+❸+❹+❻
2000-01-01   107 3088167      80000 ❼      566000 ❶+❷+❸+❹+❼
2000-01-01   108 3088169      32000 ❽      766000 ❶+❷+❸+❹+❺+❻+❼+❽
2000-01-01   109 3088170      30000 ❾      796000
❶+❷+❸+❹+❺+❻+❼+❽+❾
2000-01-01   110 3088170      20000 ❿      786000
❶+❷+❸+❹+❺+❻+❼+❽+❿
------------------------------------------------------------------------
2000-03-03   301 3088161     180000 ❶      180000 ❶
2000-03-03   302 3088161      42000 ❷       42000 ❷
2000-03-03   303 3088165      47000 ❸      269000 ❶+❷+❸
2000-03-03   304 3088167     217000 ❹      486000 ❶+❷+❸+❹
```

```
2000-03-03    305 3088167     108000 ❺      377000 ❶+❷+❸+❺


15 rows selected.
```

```
# GROUPS


gSQL>
SELECT orderdate AS O_DATE,
       orderkey AS O_KEY,
       custkey,
       totalprice,
       SUM( totalprice ) OVER( PARTITION BY orderdate
                               ORDER BY custkey
                               GROUPS BETWEEN UNBOUNDED PRECEDING
                                          AND CURRENT ROW
                               EXCLUDE TIES ) AS SUM_OVER_RESULT
  FROM orders;


O_DATE     O_KEY CUSTKEY TOTALPRICE    SUM_OVER_RESULT
---------- ----- ------- ----------    ---------------
2000-01-01   101 3088161     180000 ❶      180000 ❶
2000-01-01   102 3088163      42000 ❷      222000 ❶+❷
2000-01-01   103 3088163      47000 ❸      227000 ❶+❸
2000-01-01   104 3088165     217000 ❹      486000 ❶+❷+❸+❹
```

```
2000-01-01   105 3088167     108000 ❺      594000 ❶+❷+❸+❹+❺

2000-01-01   106 3088167     60000 ❻       546000 ❶+❷+❸+❹+❻

2000-01-01   107 3088167     80000 ❼       566000 ❶+❷+❸+❹+❼

2000-01-01   108 3088169      32000 ❽      766000 ❶+❷+❸+❹+❺+❻+❼+❽

2000-01-01   109 3088170     30000 ❾       796000

❶+❷+❸+❹+❺+❻+❼+❽+❾

2000-01-01   110 3088170     20000 ❿       786000

❶+❷+❸+❹+❺+❻+❼+❽+❿

-------------------------------------------------------------------------

2000-03-03   301 3088161     180000 ❶      180000 ❶

2000-03-03   302 3088161     42000 ❷        42000 ❷

2000-03-03   303 3088165      47000 ❸      269000 ❶+❷+❸

2000-03-03   304 3088167     217000 ❹      486000 ❶+❷+❸+❹

2000-03-03   305 3088167     108000 ❺      377000 ❶+❷+❸+❺


15 rows selected.
```

使用示例

```
gSQL>

SELECT item_no,

       sales_date,

       sales,

       SUM( sales ) OVER W1 cumulative_sales,

       AVG( sales ) OVER w1 avg_sales
```

```
  FROM store
WINDOW w1 AS ( PARTITION BY item_no
               ORDER BY sales_date
               ROWS BETWEEN UNBOUNDED PRECEDING
                        AND CURRENT ROW );


ITEM_NO SALES_DATE SALES CUMULATIVE_SALES AVG_SALES
------- ---------- ----- ---------------- ---------
    100 2001-01-01   150              150       150
    100 2001-01-02   100              250       125
    100 2001-01-03   170              420       140
    100 2001-01-04    90              510     127.5
    100 2001-01-05   200              710       142
    235 2001-01-01    70               70        70
    235 2001-01-02   130              200       100
    235 2001-01-03   190              390       130
    235 2001-01-04   150              540       135
    235 2001-01-05    50              590       118


10 rows selected.
```

兼容性

Feature ID	说明	是否支持
T611	Elementary OLAP operations	X
T612	Advanced OLAP operations	X
T301	Functional dependencies	X
T620	WINDOW clause: GROUPS option	O

Table 10-16 SQL标准兼容性

参考

相关内容参考下文

- **group by clause**
- **order by clause**
- **Window Function**

order by clause

功能

描述查询结果的排列顺序

语句

```
<order by clause> ::=

    ORDER BY <sort specification list>


<sort specification list> ::=

    <sort specification> [ { <comma> <sort specification> }... ]


<sort specification> ::=

    <sort key> [ <ordering specification> ] [ <null ordering> ]


<sort key> ::=

    <value expression>


<ordering specification> ::=

      ASC

    | DESC


<null ordering> ::=
```

```
    NULLS FIRST
  | NULLS LAST
```

使用范围及访问权限

为了排序而描述的 <sort key>中有column时用户需要有对column的访问权限

语句规则及参数

<order by clause>

- <query specification>中描述了<set quantifier> DISTINCT时<sort key>中只能有 <select list>中描述的表达式
  - SELECT DISTINCT c1, c2 FROM t1 ORDER BY c1;
  - (X) SELECT DISTINCT c1, c2 FROM t1 ORDER BY c5;
- <query specification>的<select list>中描述了一个以上的<set function specification>时<sort key>中只能有<select list>中描述的表达式
  - SELECT c1, sum(c2) FROM t1 GROUP BY c1 ORDER BY c1;
  - (X) SELECT c1, sum(c2) FROM t1 GROUP BY c1 ORDER BY c5;
- 在<set operator>语句指定了<order by clause>时以最先描述的<query specification>为准分析<sort key>
  - SELECT c1, c2 FROM t1 UNION SELECT i1, i2 FROM t3 ORDER BY c1, c2;
  - (X) SELECT c1, c2 FROM t1 UNION SELECT i1, i2 FROM t3 ORDER BY i1, i2;

<sort specification list>

- <ordering specification>
  - ASC
  - DESC
  - 未指定时默认为ASC
- <null ordering>
  - NULLS FIRST
  - NULLS LAST
  - 未指定时默认为NULLS LAST

<sort key>

- <sort key>的<value expression>为正整数值时将其值用作sort key index
  - 将对应该值的<query specification>的第i个<select sublist>用作sort key
    - SELECT c1, c2 FROM t1 ORDER BY 1;
    - 以sort key排列C1
  - 不存在对应该值的<query specification>的第i个<select sublist>时返回错误
    - (X) SELECT c1, c2 FROM t1 ORDER BY 3;
- <value expression>不支持row subquery或relation subquery
  - (X) SELECT c1, c2 FROM t1 ORDER BY (SELECT i1, i2 FROM t2 FETCH FIRST ROW ONLY);
  - T2中有多个记录
    - (X) SELECT c1, c2 FROM t1 ORDER BY (SELECT i1 FROM t2);
- 其余<value expression>用作sort key

说明

<order by clause>

<order by clause>描述排序检索结果的方法

<order by clause>中可用逗号(,)列表列出<sort key>按照列出的顺序比较记录的<sort key>并按照顺序排列

```
SELECT c1, c2 FROM t1 ORDER BY c1, c2;
```

<sort key>中可以描述指定升序或降序的<ordering specification>省略时排列为升序

```
gSQL> SELECT c1 FROM t1;
C1
--
 2
 3
 1
3 rows selected.
```

- 升序（ASC）

```
gSQL> SELECT c1 FROM t1 ORDER BY c1;
C1
--
 1
```

```
 2

 3

3 rows selected.


gSQL> SELECT c1 FROM t1 ORDER BY c1 ASC;

C1

--

 1

 2

 3

3 rows selected.
```

- 降序（DESC）

```
gSQL> SELECT c1 FROM t1 ORDER BY c1 DESC;

C1

--

 3

 2

 1

3 rows selected.
```

<sort key>中可以使用<null ordering>指定NULL值和非NULL值的顺序省略时排列为NULLS LAST

```
gSQL> SELECT c1 FROM t1;

  C1
```

```
----

   2

null

   1

3 rows selected.
```

- NULLS LAST

```
gSQL> SELECT c1 FROM t1 ORDER BY c1;

  C1

----

   1

   2

null

3 rows selected.


gSQL> SELECT c1 FROM t1 ORDER BY c1 NULLS LAST;

  C1

----

   1

   2

null

3 rows selected.
```

- NULLS FIRST

```
gSQL> SELECT c1 FROM t1 ORDER BY c1 NULLS FIRST;
  C1
----
null
   1
   2
3 rows selected.
```

在<sort key>中描述常数值时将在<select list>中位于对应值的顺序的表达式视为<sort key>另外此时描述的常数值是大于0的整数应小于或等于<select list>中描述的表达式的所有数量

```
gSQL> SELECT c1 FROM t1 ORDER BY 1;
  C1
----
   1
   2
null
3 rows selected.
```

<sort key>中不能描述LONG type (LONG VARCHAR, LONG VARBINARY)

与null value的比较

- null value之间进行比较时视为相同值
- null value与非null value的值之间进行比较时遵循以下规则
    - 为NULLS FIRST且为ASC时 ：null value < not null value

  - 为NULLS LAST且为ASC时 : null value > not null value
  - 为NULLS FIRST且为DESC时 : null value > not null value
  - 为NULLS LAST且为DESC时 : null value < not null value
- null value之间的比较结果为UNKNOWN时根据检索顺序排序

排序拥有相同sort key值的row

不能以 sort key区分的行称为peerpeer根据检索顺序排序

用于<sort key>的<aggregation function>

<query specification>中使用<aggregation function>或描述<group by clause>时<aggregation function>可以作为<sort key>使用

但只在描述<group by clause>的情况下重叠的<aggregation function>可以作为<sort key>使用

```
gSQL> SELECT c1, c2 FROM t1;
C1 C2
-- --
 2  1
 3  5
 1  2
 2 10
 3 10
5 rows selected.

gSQL> SELECT sum(c1) FROM t1 ORDER BY sum(c1);
```

```
SUM(C1)

-------

     11

1 row selected.


gSQL> SELECT c1, sum(c2) FROM t1 GROUP BY c1 ORDER BY sum(c2);

C1 SUM(C2)

-- -------

 1       2

 2      11

 3      15

3 rows selected.


gSQL> SELECT sum(c1) FROM t1 GROUP BY c1 ORDER BY sum(sum(c1));

SUM(C1)

-------

      6

1 row selected.
```

使用示例

以下为使用ORDER BY的SELECT语句的示例

```
gSQL> SELECT c_name, c_nation FROM customer ORDER BY c_nation;


C_NAME     C_NATION
```

```
---------- -------------

Customer#2 CANADA

Customer#4 GERMANY

Customer#1 KOREA

Customer#3 KOREA

Customer#5 UNITED STATES


5 rows selected.


gSQL> SELECT c_name, c_nation FROM customer ORDER BY c_nation DESC;


C_NAME     C_NATION

---------- -------------

Customer#5 UNITED STATES

Customer#1 KOREA

Customer#3 KOREA

Customer#4 GERMANY

Customer#2 CANADA


5 rows selected.


gSQL> SELECT c_name, c_nation FROM customer ORDER BY 2 DESC;


C_NAME     C_NATION

---------- -------------
```

```
Customer#5 UNITED STATES

Customer#1 KOREA

Customer#3 KOREA

Customer#4 GERMANY

Customer#2 CANADA


5 rows selected.
```

兼容性

Feature ID	说明	是否支持
F850	Top-level <order by clause>in <query expression>	O
F851	<order by clause>in subqueries	O
F852	Top-level <order by clause>in views	O
F855	Nested <order by clause>in <query expression>	O

Table 10-17 标准SQL兼容性

参考

相关内容参考**query expression**

offset limit clause

功能

描述检索结果中要跳过的行数与要fetch的行数

语句

```
<offset limit clause> ::=
      <result offset clause>
    | <fetch limit clause>
    | <result offset clause> <fetch limit clause>

<result offset clause> ::=
    OFFSET <offset row count> [ { ROW | ROWS } ]

<fetch limit clause> ::=
      <fetch first clause>
    | <limit clause>

<fetch first clause> ::=
    FETCH [ FIRST | NEXT ] [ <fetch row count> ] [ ROW ONLY | ROWS ONLY ]

<limit clause> ::=
    LIMIT { <fetch row count> | <offset row count> , <fetch row count> |
```

```
ALL }
```

使用范围及访问权限

<offset limit clause>不需要访问权限

语句规则及参数

<result offset clause>

- <offset row count>值应为大于或等于0的正整数
- ROW与ROWS为相同意义的关键字可以省略
- 省略语句时含义为OFFSET 0 ROWS

<fetch limit clause>

- 指定检索结果中要跳过的的行数
- 省略语句时含义为LIMIT ALL

<fetch first clause>

- 指定要fetch的行数
- 不能与<limit clause>一起使用
- FIRST与NEXT为相同意义的关键字可以省略
- ROW ONLY与ROWS ONLY为相同意义的关键字可以省略
- <fetch row count>

- 应为大于0的正整数
- 可省略省略时其值为1

<limit clause>

- 指定要fetch的行数
- 指定查询结果中要跳过的行数与要fetch的行数
- 不能与<fetch first clause>一起使用
- 以LIMIT <fetch row count>使用时
  - <fetch row count>应为大于0的正整数
  - 此语句与FETCH FIRST <fetch row count> ROWS ONLY有相同的意义
- 以LIMIT <offset row count><fetch row count>使用时
  - 不能与<result offset clause>同时使用
  - <offset row count>应为大于或等于0的正整数
  - <fetch row count>应为大于0的正整数
  - 此语句与OFFSET <offset row count> ROWS FETCH FIRST <fetch row count> ROWS ONLY有相同的意义
- 以LIMIT ALL使用时不限制要fetch的行数

说明

<result offset clause>

在检索结果中从第<offset row count>的行开始fetch如果<offset row count>检索的结果大于或等于行数则fetch row的数量为0条

```
gSQL> SELECT c1 FROM t1;

C1

--

 1

 2

 3

3 rows selected.


gSQL> SELECT c1 FROM t1 OFFSET 1;

C1

--

 2

 3

2 rows selected.


gSQL> SELECT c1 FROM t1 OFFSET 3;

no rows selected.
```

<fetch first clause>

在检索结果中fetch与<fetch row count>数量相同数量的结果

```
gSQL> SELECT c1 FROM t1;

C1

--

 1
```

```
 2

 3

3 rows selected.


gSQL> SELECT c1 FROM t1 FETCH FIRST 2 ROWS ONLY;

C1

--

 1

 2

2 rows selected.
```

<limit clause>

使用LIMIT <fetch_row_count>时在检索结果中fetch与<fetch row count>数量相同数量的结果

使用LIMIT <offset row count><fetch row count>时在检索结果中从第<offset row count>个行开始fetch与<fetch row count>数量相同数量的结果

使用LIMIT ALL使用时无数量限制的fetch检索的结果

```
gSQL> SELECT c1 FROM t1;

C1

--

 1

 2

 3

3 rows selected.
```

• LIMIT <fetch_row_count>

```
gSQL> SELECT c1 FROM t1 LIMIT 2;
C1
--
 1
 2
2 rows selected.
```

• LIMIT <offset row count>, <fetch_row_count>

```
gSQL> SELECT c1 FROM t1 LIMIT 1, 1;
C1
--
 2
1 row selected.
```

• LIMIT ALL

```
gSQL> SELECT c1 FROM t1 LIMIT ALL;
C1
--
 1
 2
 3
3 rows selected.
```

使用示例

以下为使用<result offset clause>的SELECT语句的示例

```
gSQL> SELECT c_name, c_nation FROM customer OFFSET 1;


C_NAME     C_NATION

---------- -------------

Customer#2 CANADA

Customer#3 KOREA

Customer#4 GERMANY

Customer#5 UNITED STATES


4 rows selected.
```

以下为使用<fetch first clause>的SELECT语句的示例

```
gSQL> SELECT c_name, c_nation FROM customer FETCH FIRST ROW ONLY;


C_NAME     C_NATION

---------- --------

Customer#1 KOREA


1 row selected.


gSQL> SELECT c_name, c_nation FROM customer FETCH FIRST 2 ROW ONLY;
```

```
C_NAME     C_NATION
---------- --------
Customer#1 KOREA
Customer#2 CANADA

2 rows selected.
```

以下为使用<limit clause>的SELECT语句的示例

```
gSQL> SELECT c_name, c_nation FROM customer LIMIT 1;

C_NAME     C_NATION
---------- --------
Customer#1 KOREA

1 row selected.

gSQL> SELECT c_name, c_nation FROM customer LIMIT 1, 2;

C_NAME     C_NATION
---------- --------
Customer#2 CANADA
Customer#3 KOREA

2 rows selected.
```

```
gSQL> SELECT c_name, c_nation FROM customer LIMIT ALL;

C_NAME     C_NATION
---------- -------------
Customer#1 KOREA
Customer#2 CANADA
Customer#3 KOREA
Customer#4 GERMANY
Customer#5 UNITED STATES

5 rows selected.
```

以下为使用<result offset clause>与<fetch limit clause>的SELECT语句的示例

```
gSQL> SELECT c_name, c_nation FROM customer OFFSET 1 FETCH 2;

C_NAME     C_NATION
---------- --------
Customer#2 CANADA
Customer#3 KOREA

2 rows selected.

gSQL> SELECT c_name, c_nation FROM customer OFFSET 1 LIMIT 2;
```

```
C_NAME      C_NATION

---------- --------

Customer#2 CANADA

Customer#3 KOREA


2 rows selected.
```

兼容性

Feature ID	说明	是否支持
F861	Top-level <result offset clause>in <query expression>	O
F862	<result offset clause>in subqueries	O
F863	Nested <result offset clause>in <query expression>	O
F864	Top-level <result offset clause>in views	O
F865	dynamic <offset row count>in <result offset clause>	X

Table 10-18 标准SQL兼容性

set operator

功能

对子查询结果的集合执行运算

语句

```
<set operator> ::=
      <set operator term>
    | <query expression body> UNION [ ALL | DISTINCT ] <set operator term>
    | <query expression body> EXCEPT [ ALL | DISTINCT ] <set operator
term>
    | <query expression body> MINUS [ ALL | DISTINCT ] <set operator term>


<set operator term> ::=
      <query term>
    | <set operator term> INTERSECT [ ALL | DISTINCT ] <set operator term>
```

使用范围及访问权限

需要拥有各<set operator term>的<query expression>的访问权限才能执行<set operator>语句

语句规则及参数

<set operator>

- 描述子查询之间的集合运算
- 各子查询的<select list>的目标数量应相同而且匹配的目标类型应均属于相同的data type group
- 第一个子查询的<select list>的目标名成为<set operator>结果目标的代表名
    - gSQL> SELECT c1 AS NAME FROM t1 UNION SELECT i1 FROM t2;

      NAME

      1

      2

      2 rows selected.
- 不使用括号等明确描述执行顺序时从左侧子查询到右侧子查询的顺序评估并执行
- <set operator>的各operator意义如下
    - UNION
        - UNION ALL：不删除重复的子查询结果并处理为合集
        - UNION DISTINCT：删除重复的子查询结果并处理为合集
        - 未描述ALL/DISTINCT时与描述DISTINCT的运行方法相同
    - EXCEPT
        - EXCEPT ALL ：不删除重复的子查询结果并处理为差集
        - EXCEPT DISTINCT：删除重复数据的子查询结果并处理为差集
        - 未描述ALL/DISTINCT时与描述DISTINCT的运行方式相同
    - MINUS

        - 是EXCEPT的别名与EXCEPT运行方式相同
    - INTERSECT
        - INTERSECT ALL：不删除重复的子查询结果并处理为交集
        - INTERSECT DISTINCT：删除重复数据的子查询结果并处理为交集
        - 未描述ALL/DISTINCT时与描述DISTINCT的运行方式相同

<query term>

描述一个子查询

详细内容参考**query expression**

说明

<set operator>的ALL与DISTINCT区别

例如R1表与R2表的数据如下时执行各个<set operator>的结果如下

- TABLE 数据
    - R1 TABLE = {1, 1, 1, 2, 2, 2, 3, 4, 4, 5}
    - R2 TABLE = {1, 1, 3, 3, 4}
- SELECT * FROM R1 UNION ALL SELECT * FROM R2;
    - result = {1, 1, 1, 1, 1, 2, 2, 2, 3, 3, 3, 4, 4, 4, 5}
- SELECT * FROM R1 UNION DISTINCT SELECT * FROM R2;
    - result = {1, 2, 3, 4, 5}
- SELECT * FROM R1 MINUS ALL SELECT * FROM R2;
    - result = {1, 2, 2, 2, 4, 5}

- SELECT * FROM R1 MINUS DISTINCT SELECT * FROM R2;
  - result = {2, 5}
- SELECT * FROM R1 INTERSECT ALL SELECT * FROM R2;
  - result = {1, 1, 3, 4}
- SELECT * FROM R1 INTERSECT DISTINCT SELECT * FROM R2;
  - result = {1, 3, 4}

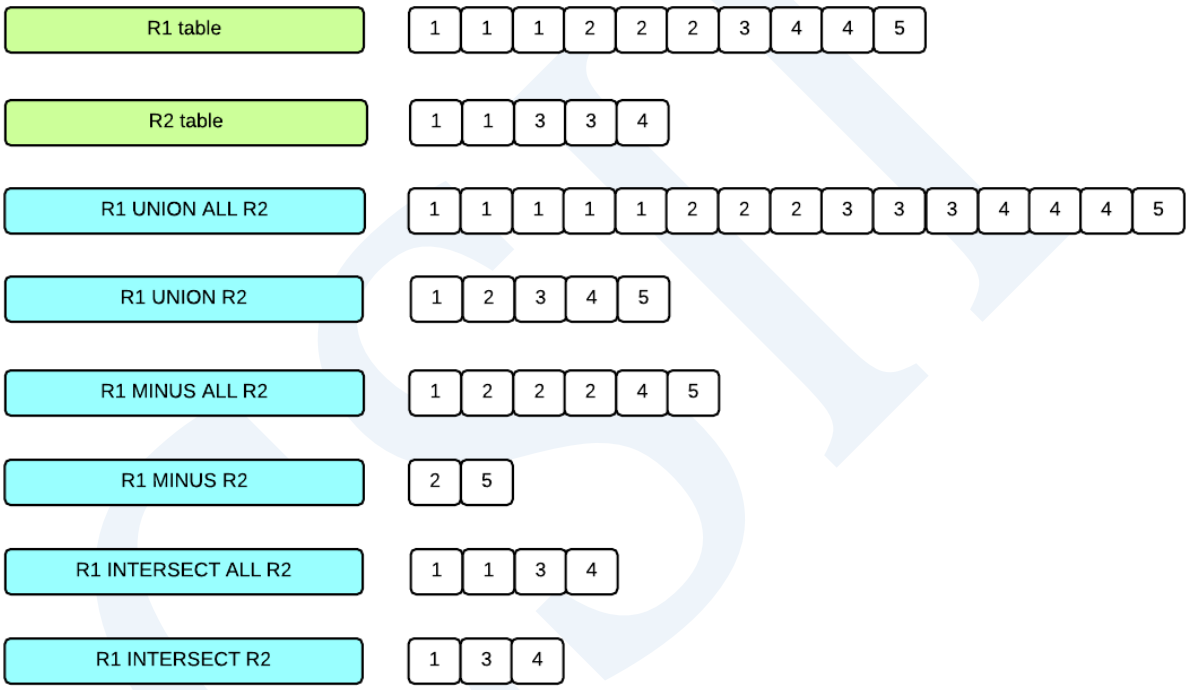


Figure 10-1 SET运算结果

运算符的优先顺序

<set operator>的运算符优先顺序如下

- 括号（）优先
- INTERSECT优先

- UNION, EXCEPT以left-right描述的顺序优先

<set operator>的结果类型

<set operator>的所有子查询的第i个Column应为同一个系列的数据类型根据结果类型组合规则决定结果类型

但LONG VARCHAR与LONG VARBINARY类型只能使用UNION ALL

ORDER BY语句

<set operator>与ORDER BY一起使用时子查询之间的Column名不同时可如下使用

- ORDER BY indicator
  - 描述结果Column的顺序

    SELECT c1 FROM t1

    UNION ALL

    SELECT c2 FROM t2

    ORDER BY 1;
- ORDER BY left_column_name
  - 描述第一个子查询的Column名

    SELECT c1 FROM t1

    UNION ALL

    SELECT c2 FROM t2

    ORDER BY c1;

使用示例

以下为使用UNION运算的SELECT语句的示例

```
gSQL> SELECT s_nation nation FROM supplier UNION ALL SELECT c_nation FROM
customer;

NATION
-------------
FRANCE
KOREA
GERMANY
UNITED STATES
CANADA
KOREA
CANADA
KOREA
GERMANY
UNITED STATES

10 rows selected.

gSQL> SELECT s_nation nation FROM supplier UNION DISTINCT SELECT c_nation
FROM customer;

NATION
```

```
-------------
UNITED STATES
CANADA
KOREA
GERMANY
FRANCE


5 rows selected.
```

以下为使用EXCEPT运算的SELECT语句的示例

```
gSQL> SELECT c_nation nation FROM customer EXCEPT ALL SELECT s_nation FROM
supplier;

NATION
------
KOREA

1 row selected.

gSQL> SELECT c_nation nation FROM customer EXCEPT DISTINCT SELECT s_nation
FROM supplier;

no rows selected.
```

以下为使用INTERSECT运算的SELECT语句的示例

```
gSQL> SELECT c_nation nation FROM customer INTERSECT ALL SELECT s_nation
FROM supplier;


NATION
-------------
UNITED STATES
CANADA
KOREA
GERMANY


4 rows selected.


gSQL> SELECT c_nation nation FROM customer INTERSECT DISTINCT SELECT
s_nation FROM supplier;


NATION
-------------
UNITED STATES
CANADA
KOREA
GERMANY


4 rows selected.
```

兼容性

Feature ID	说明	是否支持
F302	INTERSECT table operator	O
F301	CORRESPONDING	X
T551	Optional key words for default syntax	O
F304	EXCEPT ALL table operator	O

Table 10-19 标准SQL兼容性

参考

相关内容参考**query expression**

subquery

功能

描述<query expression>中衍生的scalar valuerowtable等

语句

```
<scalar subquery> ::=

    <subquery>


<row subquery> ::=

    <subquery>


<table subquery> ::=

    <subquery>


<subquery> ::=

    ( <query expression> )
```

使用范围及访问权限

需要拥有对<subquery>中的<query expression>的访问权限

语句规则及参数

<scalar subquery>

- <query expression>中的目标数量应为1个
- 根据<query expression>中返回的行数结果值如下
  - 返回0条row时结果值为NULL
  - 返回1条row时结果值为包含在对应row的结果值
  - 返回2条以上row时发生exception error

<row subquery>

- <query expression>中的目标数量应为2个以上
- 根据<query expression>中返回的行数结果值如下
  - 返回0条row时结果值为所有column为NULL的row
  - 返回0条row时结果值为对应row
  - 返回2条以上row时发生exception error

<table subquery>

- <query expression>中的目标数量应为1个以上
- 根据<query expression>中返回的行数结果值如下
  - 返回0条row时结果值为no rows
  - 返回1条以上row时结果值为对应row

说明

<scalar subquery>

<scalar subquery>是返回有一个Column的一条row作为结果值的子查询<scalar subquery>的目标只能有一个结果的数据类型遵循目标的数据类型

<scalar subquery>可以在<select list>的目标中单独使用可用于仅拥有单个Column的运算符

<row subquery>

<row subquery>为返回有两个以上Column的一条row作为结果值的子查询<row subquery>的目标应为两个以上结果的数据类型遵循目标的各个数据类型

<row subquery>不能在<select list>的目标中单独使用仅可用于拥有两个以上Column的row运算符

<table subquery>

<table subquery>为返回有一个以上Column的一条以上row作为结果值的子查询<table subquery>的目标应为1个以上结果的数据类型遵循目标的各个数据类型

<table subquery>不能在<select list>的目标中单独使用可用于INNOT INEXISTSNOT EXISTSQuantify Operator等运算符

使用示例

以下为使用<scalar subquery>的SELECT语句的示例

```
gSQL> SELECT (SELECT c_name FROM dual)  FROM customer;


(SELECT C_NAME FROM DUAL)

-------------------------

Customer#1

Customer#2

Customer#3

Customer#4

Customer#5


5 rows selected.


gSQL> SELECT c_name, c_nation FROM customer WHERE c_nation = (SELECT

'CANADA' FROM dual);


C_NAME     C_NATION

---------- --------

Customer#2 CANADA


1 row selected.
```

以下为使用<row subquery>的SELECT语句的示例

```
gSQL> SELECT p_name, p_brand, p_type FROM part WHERE (p_brand, p_type) =

(SELECT 'Brand#1', 'NICKEL' FROM dual);
```

```
P_NAME P_BRAND    P_TYPE

------ ---------- ------

Part#2 Brand#1    NICKEL


1 row selected.
```

以下为使用<table subquery>的SELECT语句的示例

```
gSQL> SELECT s_name, s_nation FROM supplier WHERE s_nation IN (SELECT

c_nation FROM customer);


S_NAME                    S_NATION

------------------------- -------------

Supplier#2                KOREA

Supplier#3                GERMANY

Supplier#4                UNITED STATES

Supplier#5                CANADA


4 rows selected.


gSQL> SELECT * FROM (SELECT s_name, s_nation FROM supplier);


S_NAME                    S_NATION

------------------------- -------------

Supplier#1                FRANCE

Supplier#2                KOREA
```

```
Supplier#3              GERMANY

Supplier#4              UNITED STATES

Supplier#5              CANADA


5 rows selected.
```

兼容性

Feature ID	说明	是否支持
F471	Scalar subquery values	O
F641	Row and table constructors	X
T501	Enhanced EXISTS predicate	O
E061-11	Subqueries in IN predicate	O
E061-12	Subqueries in quantified comparison predicate	O
E061-12	Correlated subqueries	O

Table 10-20 标准SQL兼容性

参考

相关内容参考下文

- **from clause**
- **where clause**

hint clause

描述执行语句时使用的hint

详细内容参考**SQL Hint**

10.17 SELECT .. FOR UPDATE

功能

设置是否对SELECT语句的结果集执行更新操作

语句

```
<select for update statement> ::=
    <query expression>  <updatability clause>
    ;

<updatability clause> ::=
      FOR READ ONLY
    | FOR UPDATE [ OF <column name list> ] [ <lock wait mode> ]

<lock wait mode> ::=
    | WAIT
    | WAIT second
    | NOWAIT
```

使用范围及访问权限

用户应满足以下条件才能执行<select for update statement>语句

- 用户需要对语句中使用的所有表有以下权限中的一个才能执行<query expression>语句
  - 对表的column中用于语句的所有Column有SELECT(columns) ON TABLE
  - 对表有(SELECT或CONTROL TABLE) ON TABLE
  - 对表所在的SCHEMA有(SELECT TABLE或CONTROL SCHEMA) ON SCHEMA
  - SELECT ANY TABLE ON DATABASE
- 使用FOR UPDATE语句时对为锁定对象的表需要有以下权限中的一个
  - 对对应表有(LOCK或CONTROL TABLE) ON TABLE
  - 对表所在的SCHEMA有(LOCK TABLE或CONTROL SCHEMA) ON SCHEMA
  - LOCK ANY TABLE ON DATABASE

语句规则及参数

<query expression>

SELECT语句里不能有INTO子句

为了执行FOR UPDATEquery应为可识别base table的行数据变化或在row中可获得锁的updatable query

updatable query应满足以下所有条件

- 最上层查询里不能有DISTINCT

    - (X) SELECT DISTINCT * FROM t1;
- 最上层查询里不能有GROUP BYHAVINGaggregation function
    - (X) SELECT MAX(c1) FROM t1;
- 不能有Set运算符
    - (X) SELECT * FROM t1 UNION ALL SELECT * FROM t2;
- FROM子句的表里应存在一个以上的updatable column
    - Join的表中不属于cross join的表的column不是updatable column
        - FULL OUTER JOIN不是cross join
        - NATURAL JOIN不是cross join
        - INNER JOIN中使用USING时也不是cross join
    - 以下表的column不是updatable column
        - Dictionary table, fixed table, performance view
    - View的column不是updatable table

SELECT语句的详细内容参考**query expression**

<updatability clause>

指定是否变更结果集的行数据

- FOR READ ONLY
    - 声明只读专用查询
- FOR UPDATE
    - 声明可写的查询
    - 执行查询时为了直到事务结束不被其他事务变更数据对行数据加x lock
    - <query expression> 应为updatable query

FOR UPDATE OF ...

列出执行查询时与获取lock相关的column

- FOR UPDATE OF语句中列出的column
  - 应为<query expression>的FROM子句中列出的表的可更新的column
  - 对列出的column的表获取lock
- 仅使用FOR UPDATE时
  - 与列出<query expression>的FROM子句中列出的表的所有可更新的column的意义相同
  - 对所有column的表获取锁

<lock wait mode>

与FOR UPDATE语句一起使用指定锁的获取方法

- WAIT
  - 获取查询结果之前获取对查询结果的所有行的锁
  - 一直等到获取锁
- WAIT second
  - 获取查询结果之前获取对查询结果的所有行的锁
  - 指定时间内无法获取锁则报错
  - 以秒为单位可以使用0 ~ 1000000000之间的值
- NOWAIT
  - 获取查询结果之前获取对查询结果的所有行的锁
  - 不能立即获取锁则报错
- 省略时默认值为WAIT

说明

SELECT语句与事务结束与否无关可以持续对行执行fetch相反SELECT .. FOR UPDATE语句获取对行数据的锁因此事务结束后无法执行fetch

> Note:
>
> Cursor holdability
>
> * WITH HOLD
> ** 与事务结束与否无关可持续fetch
> ** 也叫fetch across commit
>
> * WITHOUT HOLD
> ** 事务结束后无法fetch

使用示例

以下为使用FOR UPDATE语句锁定行数据的示例

```
gSQL> SELECT id, data FROM t1 WHERE id = 3 FOR UPDATE;


ID DATA
-- ------
 3 data_3
```

```
1 row selected.
```

如下即使使用join与ORDER BY语句为updatable query时可以使用FOR UPDATE语句

```
gSQL> SELECT t1.id, t1.name, t2.addr
        FROM t1, t2
       WHERE t1.id = t2.id
       ORDER BY 1
         FOR UPDATE;

ID NAME    ADDR
-- ------- -------------
 1 someone somewhere
 2 anyone  anywhere
 3 unknown N/A
 4 leekmo  leekmo's home
 5 mkkim   seoul

5 rows selected.
```

如下不为updatable query时不能使用FOR UPDATE语句

```
gSQL> SELECT id, COUNT(*)
        FROM t1
       GROUP BY id
```

```
        FOR UPDATE;


ERR-42000(16112): query expression is not updatable
```

兼容性

标准SQL未定义<select for update statement>可以通过**DECLARE cursor_name**语句定义

10.18 SELECT .. INTO

功能

通过查询检索一条row并通过主机变量获得检索的row值

语句

```
<select statement: single row> ::=
    SELECT [ <hint clause> ] [ <set quantifier> ] <select list>
        INTO <select target list>
        <table expression>
    ;

<select target list> ::=
    variable_name [, ...]
```

使用范围及访问权限

用户需要对语句中使用的所有表有以下权限中的一个才能执行<select statement: single row>语句

- 对表的column中用于语句的所有column有SELECT(columns) ON TABLE

- 对表有(SELECT或CONTROL TABLE) ON TABLE
- 对表所在的SCHEMA有(SELECT TABLE或CONTROL SCHEMA) ON SCHEMA
- SELECT ANY TABLE ON DATABASE

语句规则及参数

<hint clause>

描述执行查询时的Hint

详细内容参考**SELECT**语句的**hint clause**子句

<set quantifier>

描述是否删除查询结果的重复数据

详细内容参考**query specification**

<select list>

描述要在查询结果中检索的column

详细内容参考**select list**

INTO <select target list>

INTO子句中的变量数量应与<select list>中的expression数量相同

<table expression>

定义检索条件等查询内容

详细内容参考**query specification**

说明

检索的row应为一条以下

检索两条以上row时将报错

SELECT语句的区别

- <select statement>
  - 检索满足条件的多个row通过SQLFetch()等API检索查询的row
  - 例: SELECT c1 FROM t1 WHERE c1 > 0;
- <select statement: single row>
  - 可检索满足条件的一条以下的row检索的row为一条时通过INTO的主机变量获取其记录值
  - 例: SELECT c2 INTO :v1 FROM t1 WHERE c1 = 0;

使用示例

以下为使用interactive sql(gsql)获取主机变量值的示例

```
gSQL> \var v_id   INTEGER

gSQL> \var v_data VARCHAR(128)


gSQL> SELECT id, data INTO :v_id, :v_data FROM t1 WHERE id = 3;


V_ID V_DATA

---- ------

   3 data_3


1 row selected.
```

10.19 SELECT .. INTO .. FOR UPDATE

功能

通过查询检索一个row并设置是否执行更新后从主机变量获取检索的row的值

语句

```
<select for update statement: single row> ::=
    SELECT [ <hint clause> ] [ <set quantifier> ] <select list>
        INTO <select target list>
        <table expression>  <updatability clause>
    ;

<select target list> ::=
    variable_name [, ...]

<updatability clause> ::=
      FOR READ ONLY
    | FOR UPDATE [ OF <column name list> ] [ <lock wait mode> ]

<lock wait mode> ::=
    | WAIT
```

```
| WAIT second
| NOWAIT
```

使用范围及访问权限

用户需要对语句中使用的所有表有以下权限中的一个才能执行<select statement: single row>语句

- 对表的column中用于语句的所有Column有SELECT(columns) ON TABLE
- 对表有(SELECT或CONTROL TABLE) ON TABLE
- 对表所在的SCHEMA有(SELECT TABLE或CONTROL SCHEMA) ON SCHEMA
- SELECT ANY TABLE ON DATABASE

使用FOR UPDATE语句时对为锁定对象的表需要有以下权限中的一个

- 对对应表有(LOCK或CONTROL TABLE) ON TABLE
- 对表所在的SCHEMA有(LOCK TABLE或CONTROL SCHEMA) ON SCHEMA
- LOCK ANY TABLE ON DATABASE

语句规则及参数

<select for update statement: single row>

为了执行FOR UPDATEquery应为可识别base table的行数据变化或可在row中获取锁的updatable query

Updatable query应满足以下所有条件

- 最上层查询里不能有DISTINCT
  - (X) SELECT DISTINCT * FROM t1;
- 最上层里不能有GROUP BYHAVINGaggregation function
  - (X) SELECT MAX(c1) FROM t1;
- 不能有Set运算符
  - (X) SELECT * FROM t1 UNION ALL SELECT * FROM t2;
- FROM子句的表里应存在一个以上的updatable column
  - Join的表中不属于cross join的表的column不是updatable column
    - FULL OUTER JOIN不是cross join
    - NATURAL JOIN不是cross join
    - INNER JOIN中使用USING时不是cross join
  - 以下表的column不是updatable column
    - Dictionary table, fixed table, performance view
  - View的column不是updatable table

<updatability clause>

指定是否变更结果集的行数据

- FOR READ ONLY
  - 声明为只读专用查询
- FOR UPDATE
  - 声明为可写的查询
  - 执行查询时为了直到事务结束不被其他事务变更数据对行数据加x lock

- <query expression> 应为updatable query

FOR UPDATE OF ...

列出执行查询时与获取lock相关的column

- 在FOR UPDATE OF语句中列出的column
  - 应为<query expression>的FROM子句中列出的表的可更新的column
  - 对列出的column的表获取lock
- 仅使用FOR UPDATE时
  - 与列出<query expression>的FROM子句中列出的表的所有可更新的column的意义相同
  - 对所有column的表获取锁

<lock wait mode>

与FOR UPDATE语句一起使用指定锁的获取方法

- WAIT
  - 获取查询结果之前获取对查询结果的所有行的锁
  - 一直等到获取锁
- WAIT second
  - 获取查询结果之前获取对查询结果的所有行的锁
  - 指定时间内无法获取锁则报错
  - 以秒为单位可以使用0 ~ 1000000000之间的值
- NOWAIT
  - 获取查询结果之前获取对查询结果的所有行的锁

- 不能立即获取锁则报错

- 省略时默认值为WAIT

<hint clause>

描述用于执行查询的hint

详细内容参考**SELECT**语句的**<hint clause>**子句

<set quantifier>

定义是否删除查询结果的重复内容

详细内容参考**query specification**

<select list>

描述要从查询结果检索的column

详细内容参考**<select list>**

INTO <select target list>

INTO中定义的变量的数量应与<select list>中定义的expression数量相同

<table expression>

描述检索条件等查询内容

详细内容参考**query specification**

说明

检索的row应为一条以下

检索两条以上row时报错

SELECT语句与事务结束与否无关可以持续对row的fetch相反SELECT .. FOR UPDATE语句获取对行数据的锁因此事务结束后无法执行fetch

> Note:
>
> Cursor holdability
>
> * WITH HOLD
> ** 与事务结束与否无关可持续fetch
> ** 也称为fetch across commit
>
> * WITHOUT HOLD
> ** 事务结束后无法fetch

SELECT语句的区别

- <select statement>
    - 检索满足条件的多个row设置是否执行更新通过SQLFetch()等API检索查询的row

- 例: SELECT c1 FROM t1 WHERE c1 > 0 FOR UPDATE;

- <select for update statement: single row>
  - 可检索满足条件的一条以下的row设置是否执行更新检索的row为一条时通过INTO的主机变量获取其记录值
  - 例: SELECT c2 INTO :v1 FROM t1 WHERE c1 = 0 FOR UPDATE;

使用示例

以下为使用FOR UPDATE语句锁定行数据后使用 interactive sql(gsql)获取host变量值的示例

```
gSQL> \var v_id   INTEGER
gSQL> \var v_data VARCHAR(128)

gSQL> SELECT id, data INTO :v_id, :v_data FROM t1 WHERE id = 3 FOR UPDATE;

V_ID V_DATA
---- ------
   3 data_3

1 row selected.
```

如下即使使用join与ORDER BY语句为updatable query时可使用FOR UPDATE

```
gSQL> \var v_id   INTEGER
gSQL> \var v_name VARCHAR(128)
```

```
gSQL> \var v_addr VARCHAR(128)



gSQL> SELECT t1.id, t1.name, t2.addr
        INTO :v_id, :v_name, :v_addr
        FROM t1, t2
       WHERE t1.id = t2.id
       ORDER BY 1
       LIMIT 1
         FOR UPDATE;


ID NAME    ADDR
-- ------- -------------
 1 someone somewhere


1 row selected.
```

如下不是updatable query时无法使用FOR UPDATE语句

```
gSQL> \var v_id    INTEGER
gSQL> \var v_count INTEGER


gSQL> SELECT id, COUNT(*)
        INTO :v_id, :v_count
        FROM t1
       GROUP BY id
```

```
        FOR UPDATE;


ERR-42000(16112): query expression is not updatable
```

参考

相关内容参考下文

- **SELECT .. FOR UPDATE**
- **SELECT .. INTO**

10.20 SET CONSTRAINTS

功能

将事务中可延时的约束条件的检查点设置为IMMEDIATE或DEFERRED

语句

```
<set constraints mode statement> ::=
    SET { CONSTRAINT | CONSTRAINTS } <constraint name list> { DEFERRED |
IMMEDIATE }
    ;

<constraint name list> ::=
      ALL
    | <constraint name> [, ...]
```

使用范围及访问权限

无需访问权限即可执行SET CONSTRAINTS

> Caution:

> 集群系统中不支持

语句规则及参数

CONSTRAINT | CONSTRAINTS

CONSTRAINT与CONSTRAINTS是相同意义的关键字标准SQL是CONSTRAINTS

<constraint name list>

描述约束条件名目录或可以用ALL关键字指定所有可延时的约束条件

指定<constraint name>时约束条件应为可延时的约束条件

ALL表示所有可延时的约束条件

DEFERRED | IMMEDIATE

设置指定的可延时的约束条件的检查点

- IMMEDIATE
  - 执行DML时检查对应的约束条件
  - 事务违反对应约束条件时报错
- DEFERRED
  - 执行COMMIT时检查对应的约束条件

事务进行时在当前事务中设置检查点事务未进行则设置于下一个事务

事务结束后不影响下一个事务

说明

可延时的约束条件

可延时的(DEFERRABLE)约束条件可变更检查点

以下为创建具有可延时的约束条件的表并添加数据的示例

```
gSQL> CREATE TABLE t1
(
    id   INTEGER,
    name VARCHAR(128) CONSTRAINT t1_uk UNIQUE
                      DEFERRABLE INITIALLY IMMEDIATE
);

Table created.

gSQL> COMMIT;

Commit complete.

gSQL> INSERT INTO t1 VALUES ( 1, 'leekmo' );

1 row created.
```

```
gSQL> INSERT INTO t1 VALUES ( 2, 'mkkim' );

1 row created.

gSQL> COMMIT;

Commit complete.
```

上述示例中在name column创建了可延时的UNIQUE约束条件初始的检查点设置为INITIALLY IMMEDIATE因此每次执行DML时均检查约束条件

如下要互相替换两条row的name值时由于检查点为IMMEDIATE因此均违反约束条件

```
gSQL> UPDATE t1 SET name = 'mkkim' WHERE id = 1;

ERR-23000(16057): unique constraint (PUBLIC.T1_UK) violated

gSQL> UPDATE t1 SET name = 'leekmo' WHERE id = 2;

ERR-23000(16057): unique constraint (PUBLIC.T1_UK) violated
```

如下把检查点变更为DEFERRED后因在COMMIT时检查约束条件因此即使执行同样的如上操作也均成功

```
gSQL> SET CONSTRAINTS t1_uk DEFERRED;
```

```
Constraints set.

gSQL> UPDATE t1 SET name = 'mkkim' WHERE id = 1;

1 row updated.

gSQL> UPDATE t1 SET name = 'leekmo' WHERE id = 2;

1 row updated.

gSQL> COMMIT;

Commit complete.
```

把检查点设置为DEFERRED时由于在COMMIT的时候检查约束条件因此在违反约束条件的情况下COMMIT事务时事务会如下失败并回滚

```
gSQL> SET CONSTRAINTS t1_uk DEFERRED;

Constraints set.

gSQL> INSERT INTO t1 VALUES ( 3, 'leekmo' );

1 row created.

gSQL> COMMIT;
```

```
ERR-40002(16291): transaction rollback: integrity constraint violation :
PUBLIC.T1_UK(1)
```

违反可延时约束条件的事务

事务违反设置为DEFFERED的约束条件时执行以下语句将报如下错误

- COMMIT
  - 报错并回滚事务
- SET CONSTRAINTS ALL IMMEDIATE
  - 语句报错
- DDL
  - 语句报错

执行COMMIT时可能会发生回滚因此执行SET CONSTRAINTS ALL IMMEDIATE语句需提前查看事务是否违反约束条件

```
gSQL> SET CONSTRAINTS t1_uk DEFERRED;

Constraints set.

gSQL> INSERT INTO t1 VALUES ( 3, 'leekmo' );

1 row created.
```

```
gSQL> SET CONSTRAINTS ALL IMMEDIATE;


ERR-23000(16038): integrity constraint violation : PUBLIC.T1_UK(1)


gSQL> SELECT * FROM t1 ORDER BY id;


ID NAME
-- ------
 1 mkkim
 2 leekmo
 3 leekmo


3 rows selected.


gSQL> UPDATE t1 SET name = 'xcom73' WHERE id = 3;


1 row updated.


gSQL> SET CONSTRAINTS ALL IMMEDIATE;


Constraints set.


gSQL> COMMIT;


Commit complete.
```

事务控制语句

SET CONSTRAINTS语句与**SAVEPOINT savepoint_specifier**语句相同是事务执行过程中使用的事务控制语句

SET CONSTRAINTS应用COMMITROLLBACKROLLBACK TO SAVEPOINT语句等事务控制

以下为创建有多个可延时约束条件的表的示例

```
CREATE TABLE t1
(
   id1 INTEGER CONSTRAINT t1_uk1 UNIQUE DEFERRABLE INITIALLY IMMEDIATE,
   id2 INTEGER CONSTRAINT t1_uk2 UNIQUE DEFERRABLE INITIALLY IMMEDIATE,
   id3 INTEGER CONSTRAINT t1_uk3 UNIQUE DEFERRABLE INITIALLY IMMEDIATE
);
```

如下在事务进行中执行<set constraints mode statement>语句时根据不同的时间点可延时约束条件的检查点也发生变化

- result: success

```
INSERT INTO t1 VALUES ( 1, 1, 1 );


1 row created.


COMMIT;
```

```
Commit complete.
```

- result: success

```
SAVEPOINT sp1;


Savepoint created.
```

- result: success
- t1_uk1 constraint is DEFERRED

```
SET CONSTRAINTS t1_uk1 DEFERRED;


Constraints set.
```

- result: success

```
SAVEPOINT sp2;


Savepoint created.
```

- result: success
- t1_uk1, t1_uk2 constraints are DEFERRED

```
SET CONSTRAINTS t1_uk2 DEFERRED;


Constraints set.
```

- result: success

```
SAVEPOINT sp3;


Savepoint created.
```

- result: success
- ALL constraints are DEFERRED

```
SET CONSTRAINTS ALL DEFERRED;


Constraints set.
```

- result: success

```
SAVEPOINT sp4;


Savepoint created.
```

- result: success
- ALL constraints are IMMEDIATE

```
SET CONSTRAINTS ALL IMMEDIATE;


Constraints set.
```

如下使用ROLLBACK TO SAVEPOINT语句回滚部分事务时SET CONSTRAINTS语句也同时部分回

滚随之检查点也发生变化

- result: error

```
INSERT INTO t1 VALUES ( 1, 2, 2 );


ERR-23000(16057): unique constraint (PUBLIC.T1_UK1) violated
```

- result: error

```
INSERT INTO t1 VALUES ( 3, 1, 3 );


ERR-23000(16057): unique constraint (PUBLIC.T1_UK2) violated
```

- result: error

```
INSERT INTO t1 VALUES ( 4, 4, 1 );


ERR-23000(16057): unique constraint (PUBLIC.T1_UK3) violated
```

- result: success
- t1_uk1, t1_uk2 constraints are DEFERRED

```
ROLLBACK TO SAVEPOINT sp4;


Rollback complete.
```

- result: success

```
INSERT INTO t1 VALUES ( 1, 2, 2 );
```

```
1 row created.
```

- result: success

```
INSERT INTO t1 VALUES ( 3, 1, 3 );
```

```
1 row created.
```

- result: success

```
INSERT INTO t1 VALUES ( 4, 4, 1 );
```

```
1 row created.
```

- result: success
- t1_uk1, t1_uk2 constraints are DEFERRED

```
ROLLBACK TO SAVEPOINT sp3;
```

```
Rollback complete.
```

- result: success

```
INSERT INTO t1 VALUES ( 1, 2, 2 );


1 row created.
```

- result: success

```
INSERT INTO t1 VALUES ( 3, 1, 3 );


1 row created.
```

- result: error

```
INSERT INTO t1 VALUES ( 4, 4, 1 );


ERR-23000(16057): unique constraint (PUBLIC.T1_UK3) violated
```

- result: success
- t1_uk1 constraint is DEFERRED

```
ROLLBACK TO SAVEPOINT sp2;


Rollback complete.
```

- result: success

```
INSERT INTO t1 VALUES ( 1, 2, 2 );
```

```
1 row created.
```

- result: error

```
INSERT INTO t1 VALUES ( 3, 1, 3 );


ERR-23000(16057): unique constraint (PUBLIC.T1_UK2) violated
```

- result: error

```
INSERT INTO t1 VALUES ( 3, 1, 3 );


ERR-23000(16057): unique constraint (PUBLIC.T1_UK2) violated
```

- result: success
- all constraint are IMMEDIATE

```
ROLLBACK TO SAVEPOINT sp1;


Rollback complete.
```

- result: error

```
INSERT INTO t1 VALUES ( 1, 2, 2 );


ERR-23000(16057): unique constraint (PUBLIC.T1_UK1) violated
```

- result: error

```
INSERT INTO t1 VALUES ( 3, 1, 3 );


ERR-23000(16057): unique constraint (PUBLIC.T1_UK2) violated
```

- result: error

```
INSERT INTO t1 VALUES ( 4, 4, 1 );


ERR-23000(16057): unique constraint (PUBLIC.T1_UK3) violated
```

- result: 1 row
- 1 1 1

```
SELECT * FROM t1;


ID1 ID2 ID3
--- --- ---
  1   1   1


1 row selected.
```

对事务进行COMMIT或ROLLBACK时结束SET CONSTRAINTS语句的影响所有可延时约束条件根据约束条件的特性遵循设置的INITIALLY IMMEDIATE或INITIALLY DEFERRED的值

使用示例

以下为描述约束条件名变更检查点的示例

```
gSQL> SET CONSTRAINTS t1_uk1 DEFERRED;

Constraints set.
```

以下为变更所有可延时约束条件的检查点的示例

```
gSQL> SET CONSTRAINTS ALL DEFERRED;

Constraints set.
```

兼容性

标准SQL未定义CONSTRAINT关键字

Feature ID	说明	是否支持
F721	Deferrable constraints	O

Table 10-21 标准SQL兼容性

参考

相关内容参考下文

- 创建约束条件
  - **CREATE TABLE**
  - **ALTER TABLE name ADD CONSTRAINT**
  - **ALTER TABLE name ADD COLUMN**
  - **ALTER TABLE name ALTER COLUMN**
- 变更约束条件: **ALTER TABLE name ALTER CONSTRAINT**
- 控制约束条件检查点: **SET CONSTRAINTS**

10.21 SET SCHEMA schema_name

功能

设置要在当前session使用的默认schema名

语句

```
<set schema statement> ::=

    SET SCHEMA schema_name

    ;
```

使用范围及访问权限

无

语句规则及参数

schema_name

要在当前session设置的默认schema名

说明

设置要在当前session使用的默认schema名

未指定对象的schema名时成为在当前session使用的默认schema名

- 以u1用户访问时使用u1用户的schema path检索R relation

```
% gsql u1 u1

gsql> SELECT * FROM r;
```

- 设置SET SCHEMA语句时使用NEW_SCHEMA.R检索R relation

```
gsql> SET SCHEMA new_schema;


gsql> SELECT * FROM r;
```

使用示例

以下为u1用户拥有s1s2 schema的示例

```
CREATE USER u1 IDENTIFIED BY u1 WITHOUT SCHEMA;

CREATE SCHEMA s1 AUTHORIZATION u1;

CREATE SCHEMA s2 AUTHORIZATION u1;

COMMIT;
```

```
ALTER USER u1 SCHEMA PATH ( s1, s2 );

GRANT ALL PRIVILEGES TO u1;

COMMIT;


CREATE TABLE s1.t1 ( c1 VARCHAR(32) );

INSERT INTO s1.t1 VALUES ( 'S1.T1' );

COMMIT;


CREATE TABLE s2.t1 ( c1 VARCHAR(32) );

INSERT INTO s2.t1 VALUES ( 'S2.T1' );

COMMIT;
```

首次访问时使用用户u1的schema path解析t1表后查询S1.T1表

```
% gsql u1 u1


gSQL> SELECT current_schema FROM dual;


CURRENT_SCHEMA

--------------

S1


1 row selected.



gSQL> SELECT * FROM t1;
```

```
C1

-----

S1.T1


1 row selected.
```

使用SET SCHEMA语句后使用session的schema名解析t1表后查询S2.T1表

```
gSQL> SET SCHEMA s2;

Session set.


gSQL> SELECT current_schema FROM dual;

CURRENT_SCHEMA

--------------

S2


1 row selected.


gSQL> SELECT * FROM t1;

C1
```

```
-----

S2.T1


1 row selected.
```

兼容性

Feature ID	说明	是否支持
F761	Session management	O

Table 10-22 标准SQL兼容性

10.22 SET SESSION AUTHORIZATION user_identifier

功能

变更session user与current user

语句

```
<set session user identifier statement> ::=
    SET SESSION AUTHORIZATION user_identifier
    ;
```

使用范围及访问权限

执行logon用户需要有ACCESS CONTROL ON DATABASE权限才能执行<set session user identifier statement>语句

通过以下三种形式管理用户信息

- logon user
  - 执行login的用户维持到关闭连接
- session user

  - 与最初的logon user相同可通过SET SESSION AUTHORIZATION语句变更
- current user
  - 通常与session user相同但使用PSM或View时为了控制访问等在系统内部暂时变更
  - session user与current user的概念类似于unix system的real user与effective user之间的区别

语句规则及参数

user_identifier

要变更的用户的名称

说明

执行SET SESSION AUTHORIZATION语句后的所有语句均以session user为准执行因此检查session user的权限并创建对象时所有者也成为session user

使用示例

以下为有ACCESS CONTROL ON DATABASE权限的test用户将session user变更为u1用户的示例

```
gSQL> SET SESSION AUTHORIZATION u1;
```

```
Session set.


gSQL> SELECT LOGON_USER(), SESSION_USER(), CURRENT_USER FROM dual;


LOGON_USER() SESSION_USER() CURRENT_USER
------------ -------------- ------------
TEST         U1             U1


1 row selected.
```

兼容性

Feature ID	说明	是否支持
F321	User authorization	O

Table 10-23 标准SQL兼容性

10.23 SET SESSION CHARACTERISTICS AS transaction_mode

功能

设置会话的事务属性

语句

```
<set session characteristics statement> ::=
    SET SESSION CHARACTERISTICS AS TRANSACTION <transaction_mode>
    ;

<transaction_mode> ::=
    { <transaction_access_mode> | ISOLATION LEVEL < isolation_level > }

<transaction_access_mode> ::=
    READ { ONLY | WRITE }

< isolation_level > ::=
    { READ COMMITTED | SERIALIZABLE }
```

语句规则及参数

<transaction_access_mode>

后续事务的ACCESS MODE

- READ ONLY
- READ WRITE

<isolation_level>

后续事务的ISOLATION LEVEL

- READ COMMITTED
- SERIALIZABLE

说明

SET SESSION CHARACTERISTICS设置会话的事务属性即会话内生成的所有事务依照此属性

另外**SET TRANSACTION transaction_mode**语句只变更后续执行的一个事务的属性

使用示例

以下为在会话内创建的所有事务设置为READ ONLY的示例

```
gSQL> SET SESSION CHARACTERISTICS AS TRANSACTION READ ONLY;


Session set.
```

以下为在会话内创建的所有事务的isolation level设置为READ COMMITTED的示例

```
gSQL> SET SESSION CHARACTERISTICS AS TRANSACTION ISOLATION LEVEL READ
COMMITTED;


Session set.
```

兼容性

Feature ID	说明	是否支持
F761	Session management	O

Table 10-24 标准SQL兼容性

参考

相关内容参考**SET TRANSACTION transaction_mode**

10.24 SET TIME ZONE

功能

设置会话的TIMEZONE

语句

```
<set local time zone statement> ::=
    SET TIME ZONE <set time zone value>
    ;

<set time zone value> ::=
    { '[+|-]hh:mm' | LOCAL }
```

语句规则及参数

<set time zone value>

要设置的TIMEZONE的值

- hh:mm : 要设置的TIMEZONE的GMT OFFSET
  - offset值的范围为'-14:00' ~ '+14:00'

- LOCAL : 创建会话的时间点的TIME ZONE
    - 创建会话时TIME ZONE设置为client OS的TIME ZONE

说明

会话的time zone变更影响**CURRENT_TIMECURRENT_TIMESTAMP**等函数的结果值

使用示例

以下为将会话的time zone变更为'+09:00'的示例

```
gSQL> SET TIME ZONE '+09:00';


Session set.
```

兼容性

Feature ID	说明	是否支持
F411	Time zone specification	O

Table 10-25 标准SQL兼容性

10.25 SET TRANSACTION transaction_mode

功能

设置下一个事务的属性

语句

```
<set transaction statement> ::=
    SET TRANSACTION <transaction_mode>
    ;

<transaction_mode> ::=
    { <transaction_access_mode> | ISOLATION LEVEL < isolation_level > }

<transaction_access_mode> ::=
    READ { ONLY | WRITE }

< isolation_level > ::=
    { READ COMMITTED | SERIALIZABLE }
```

语句规则及参数

<transaction_access_mode>

下一个事务的ACCESS MODE

- READ ONLY
- READ WRITE

<isolation_level>

下一个事务的ISOLATION LEVEL

- READ COMMITTED
- SERIALIZABLE

说明

SET TRANSACTION设置下一个事务的属性事务结束后事务的属性将恢复到默认值

使用示例

以下为将下一个执行的事务设置为只读模式的示例

```
gSQL> SET TRANSACTION READ ONLY;


Transaction set.
```

兼容性

Feature ID	说明	是否支持
T251	SET TRANSACTION statement: LOCAL option	X

Table 10-26 标准SQL兼容性

参考

相关内容参考**SET SESSION CHARACTERISTICS AS transaction_mode**

10.26 TRUNCATE TABLE

功能

删除表的所有row

语句

```
<truncate table statement> ::=
    TRUNCATE TABLE table_name
        [ RESTART IDENTITY | CONTINUE IDENTITY ]
        [ DROP STORAGE | DROP ALL STORAGE ]
    ;
```

使用范围及访问权限

用户需要有以下权限中的一个才能执行<truncate table statement>语句

- 表的所有者
- 对表有CONTROL TABLE ON TABLE
- 对表所在的SCHEMA有(DROP TABLE或CONTROL SCHEMA) ON SCHEMA
- DROP ANY TABLE ON DATABASE

语句规则及参数

table_name

要删除row的对象表名

与schema_name.table_name相同可定义表所在的SCHEMA省略schema_name时使用执行语句的用户的默认SCHEMA名

[RESTART IDENTITY | CONTINUE IDENTITY]

- RESTART IDENTITY
  - 表中有拥有自动生成值的Column(identity column)时自动重新开始
- CONTINUE IDENTITY
  - 表中有拥有自动生成值的Column(identity column)时不变更原有值
- 未指定时默认值为CONTINUE IDENTITY

[DROP STORAGE | DROP ALL STORAGE]

- DROP STORAGE
  - 分配给表的extent中删除除MINSIZE大小外的extent
- DROP ALL STORAGE
  - 删除分配给表的所有extent
- 未指定时默认值为DROP STORAGE

说明

TRUNCATE TABLE等Data Definition Language (DDL)语句在事务没有提交的情况下也可以ROLLBACK

使用示例

以下为执行TRUNCATE TABLE语句的示例

```
gSQL> TRUNCATE TABLE t1;


Table truncated.
```

以下为执行TRUNCATE TABLE时重新开始identity column值的示例

```
TRUNCATE TABLE t1 RESTART IDENTITY;


Table truncated.
```

兼容性

标准SQL未定义[DROP STORAGE | DROP ALL STORAGE] 语句

Feature ID	说明	是否支持
F200	TRUNCATE TABLE statement	O
F202	TRUNCATE TABLE: identity column restart option	O

Table 10-27 标准SQL兼容性

10.27 UPDATE

功能

更新表的row

语句

```
<update statement: searched> ::=
    UPDATE table_name [ [ AS ] alias_name ]
        SET <set clause> [, ...]
        [ WHERE <search condition> ]
        [ <result offset clause> ]
        [ <fetch limit clause> ]
    ;

<set clause> ::=
      column_name = { <value expression> | DEFAULT }
    | ( column_name [, ...] ) = ( { <value expression> | DEFAULT }
[, ...] )
    | ( column_name [, ...] ) = ( <query expression> )
```

```
<result offset clause> ::=

    OFFSET skip_count [ ROW | ROWS ]



<fetch limit clause> ::=

      <fetch first clause>

    | <limit clause>



<fetch first clause> ::=

    FETCH [ FIRST | NEXT ] [ row_count ] [ ROW ONLY | ROWS ONLY ]



<limit clause>

    LIMIT { fetch_row_count | offset_row_count, fetch_row_count | ALL }
```

使用范围及访问权限

用户需要有以下权限中的一个才能执行<update statement: searched>语句

- 对update的所有Column有UPDATE(columns) ON TABLE
- 对表有(UPDATE或CONTROL TABLE) ON TABLE
- 对表所在的SCHEMA有(UPDATE TABLE或CONTROL SCHEMA) ON SCHEMA
- UPDATE ANY TABLE ON DATABASE

语句规则及参数

table_name

要更新row的对象表名

与schema_name.table_name相同可定义表所在的SCHEMA省略schema_name时使用执行语句的用户的默认SCHEMA名

[AS alias_name]

table_name的别名

<set clause>

定义要变更的column与要分配的值<set clause>的Column数量应与值的数量相同

用如下方法定义

- column_name = { <value expression> | DEFAULT }

```
UPDATE table_name
   SET column1 = value1, column2 = value2, column3 = value3
```

- (column_name [, ...]) = ({ <value expression> | DEFAULT } [, ...])

```
UPDATE table_name
```

```
    SET ( column1, column2, column3 ) = ( value1, value2, value3 )
```

- (column_name [, ...]) = (<query expression>)

```
UPDATE table_name
    SET column1 = ( SELECT max(value1) FROM other_table_name )
```

<query expression>应为生成一条行数据的语句

Column值使用默认值时使用执行**CREATE TABLE**时定义的默认值(参考:**<default clause>**)未定义时被分配空值

WHERE <search condition>

变更满足WHERE条件的row

不指定WHERE条件时变更所有的row

WHERE条件的详细内容参考**SELECT**的**where clause**

<result offset clause>

指定查询结果中要跳过的row数

详细内容参考**SELECT**语句的**<result offset clause>**

<fetch limit clause>

指定要Fetch的row的数量的语句可使用以下两种方法

- <fetch first clause>
  - 指定要fetch的row数
  - 详细内容参考**SELECT**语句的**<fetch first clause>**
- <limit clause>
  - 指定要fetch的row数或同时指定查询结果中要跳过的row数与要fetch的row数
  - 详细内容参考**SELECT**语句的**<limit clause>**

说明

UPDATE相关语句之间的区别

- **UPDATE**
  - 更新满足条件的多个row
  - 例: UPDATE t1 SET c2 = c2 + 1 WHERE c1 > 0;
- **UPDATE name WHERE CURRENT OF cursor_name**
  - 更新当前游标指向的row
  - 例: UPDATE t1 WHERE CURRENT OF cursor;
- **UPDATE name RETURNING**
  - 更新满足条件的多个row对更新的row可以与**SELECT**语句相同的方式通过(SQLFetch() 等API)进行检索
  - 例: UPDATE t1 SET c2 = c2 + 1 WHERE c1 > 0 RETURNING c2;
- **UPDATE name RETURNING .. INTO**
  - 更新一条以下的row更新的rwo为一条时通过RETURNING INTO的主机变量获取值
  - 例: UPDATE t1 SET c2 = c2 + 1 WHERE c1 = 0 RETURNING c2 INTO :v1;

使用示例

以下为更新满足条件的多个row的示例

```
gSQL> UPDATE lineitem
         SET l_shipdate = CURRENT_DATE
       WHERE l_returnflag = 'R';


5 rows updated.
```

以下为更新多个column值的示例

```
gSQL> UPDATE lineitem
         SET l_shipdate   = CURRENT_DATE
           , l_returnflag = 'A'
       WHERE l_returnflag = 'R';


5 rows updated.
```

以下为用括号括住多个column并更新的示例

```
gSQL> UPDATE lineitem
         SET ( l_shipdate  , l_returnflag )
           = ( CURRENT_DATE, 'A' )
       WHERE l_returnflag = 'R';
```

```
5 rows updated.
```

以下为使用子查询更新column值的示例

```
gSQL> UPDATE lineitem
         SET l_discount = ( SELECT MAX(l_discount) + 0.01 FROM lineitem )
       WHERE l_returnflag = 'R';


5 rows updated.
```

以下为使用OFFSET与FETCH子句在满足条件的row中更新部分row的示例

```
gSQL> UPDATE lineitem
         SET l_discount = l_discount + 0.01
       WHERE l_returnflag = 'R'
      OFFSET 3
      FETCH 2;


2 rows updated.
```

兼容性

标准SQL未定义UPDATE的以下语句

- <result offset clause>
- <fetch limit clause>

Feature ID	说明	是否支持
F781	Self-referencing operations	X
T111	Updatable joins, unions, and columns	X

Table 10-28 标准SQL兼容性

10.28 UPDATE name RETURNING

功能

更新表的row检索变更前后的row

语句

```
<update statement: searched> ::=
    UPDATE table_name [ [ AS ] alias_name ]
        SET <set clause> [, ...]
        [ WHERE <search condition> ]
        [ <result offset clause> ]
        [ <fetch limit clause> ]
        <returning clause>

<set clause> ::=
      column_name = { <value expression> | DEFAULT }
    | ( column_name [, ...] ) = ( { <value expression> | DEFAULT }
[, ...] )
    | ( column_name [, ...] ) = ( <query expression> )
```

```
<result offset clause> ::=

    OFFSET skip_count [ ROW | ROWS ]



<fetch limit clause> ::=

      <fetch first clause>

    | <limit clause>



<fetch first clause> ::=

    FETCH [ FIRST | NEXT ] [ row_count ] [ ROW ONLY | ROWS ONLY ]



<limit clause>

    LIMIT { fetch_row_count | offset_row_count, fetch_row_count | ALL }



<returning clause> ::=

    { RETURN | RETURNING } [ NEW | OLD ] { * | { <value expression> [ [AS]
alias_name] } [, ...] }
```

使用范围及访问权限

用户应满足以下条件才能执行<update returning query statement>语句

- 用户需要有以下权限中的一个才能执行UPDATE语句
  - 对update的所有column有UPDATE(columns) ON TABLE
  - 对表有(UPDATE或CONTROL TABLE) ON TABLE
  - 对表所在的SCHEMA有(UPDATE TABLE或CONTROL SCHEMA) ON SCHEMA
  - UPDATE ANY TABLE ON DATABASE
- 对用于RETURNING语句的所有Column需要有以下权限中的一个
  - 对用于RETURNING语句的所有Column有SELECT(columns) ON TABLE
  - 对表有(SELECT或CONTROL TABLE) ON TABLE
  - 对表所在的SCHEMA有(SELECT TABLE或CONTROL SCHEMA) ON SCHEMA
  - SELECT ANY TABLE ON DATABASE

语句规则及参数

table_name

要更新row的对象表名

[AS alias_name]

table_name的别名

<set clause>

定义要变更的column与要分配的值<set clause>的column数量应与值的数量相同

详细内容参考**UPDATE**

WHERE <search condition>

变更满足WHERE条件的row

不指定WHERE条件时变更所有的row

WHERE条件的详细内容参考**SELECT**的**where clause**

<result offset clause>

指定查询结果中要跳过的row数

详细内容参考**SELECT**语句的**<result offset clause>**

<fetch limit clause>

指定要fetch的row数的语句可以使用以下两种方法

- <fetch first clause>
  - 指定要fetch的row数
  - 详细内容参考**SELECT**语句的**<fetch first clause>**
- <limit clause>
  - 指定要fetch的row数或同时指定查询结果中要跳过的row数与要fetch的row数
  - 详细内容参考**SELECT**语句的**<limit clause>**

<returning clause>

将更新的row作为结果集并描述要在其中搜索的column

- RETURN与RETURNING为相同意义的关键字
- NEW | OLD
  - NEW：在更新后的row中以更新后的row为准检索
  - OLD：在更新后的row中以变更前的row为准检索
  - 省略时默认值为NEW
- <value expression>
  - 与SELECT语句的<select list>相同但不能使用Aggregation等
- [[AS] alias_name]
  - 使用AS子句可指定<value expression>的名称

说明

详细内容参考**UPDATE相关语句之间的区别**

使用示例

以下为通过RETURNING获得更新后的row值的示例

```
gSQL> UPDATE lineitem
         SET l_discount = l_discount + 0.01
       WHERE l_returnflag = 'R'
   RETURNING l_orderkey, l_linenumber, l_discount;


L_ORDERKEY L_LINENUMBER L_DISCOUNT
```

```
---------- ------------ ----------
         8            1        .07
         9            2        .11
        12            5        .05
        15            1        .03
        16            2        .08


5 rows updated.
```

以下为通过RETURNING OLD获取更新后的row的更新前的值的示例

```
gSQL> UPDATE lineitem
         SET l_discount = l_discount + 0.01
       WHERE l_returnflag = 'R'
   RETURNING OLD l_orderkey, l_linenumber, l_discount;


L_ORDERKEY L_LINENUMBER L_DISCOUNT
---------- ------------ ----------
         8            1        .06
         9            2         .1
        12            5        .04
        15            1        .02
        16            2        .07


5 rows updated.
```

兼容性

标准SQL中没有<update returning query statement>

10.29 UPDATE name RETURNING .. INTO

功能

更新表的一条row并通过主机变量获取更新的row值

语句

```
<update statement: searched> ::=
    UPDATE table_name [ [ AS ] alias_name ]
        SET <set clause> [, ...]
        [ WHERE <search condition> ]
        [ <result offset clause> ]
        [ <fetch limit clause> ]
        <returning into clause>
    ;



<set clause> ::=
      column_name = { <value expression> | DEFAULT }
    | ( column_name [, ...] ) = ( { <value expression> | DEFAULT }
[, ...] )
    | ( column_name [, ...] ) = ( <query expression> )
```

```
<result offset clause> ::=
    OFFSET skip_count [ ROW | ROWS ]


<fetch limit clause> ::=
      <fetch first clause>
    | <limit clause>


<fetch first clause> ::=
    FETCH [ FIRST | NEXT ] [ row_count ] [ ROW ONLY | ROWS ONLY ]


<limit clause>
    LIMIT { fetch_row_count | offset_row_count, fetch_row_count | ALL }


<returning into clause> ::=
    { RETURN | RETURNING } [ NEW | OLD ] { * | { <value expression> [ [AS]
alias_name] } [, ...] } INTO variable_name [, ...]
```

使用范围及访问权限

用户应满足以下条件才能执行<update returning query statement>语句

- 用户需要有以下权限中的一个才能执行UPDATE语句
  - 对update的所有column有UPDATE(columns) ON TABLE
  - 对表有(UPDATE或CONTROL TABLE) ON TABLE
  - 对表所在的SCHEMA有(UPDATE TABLE或CONTROL SCHEMA) ON SCHEMA
  - UPDATE ANY TABLE ON DATABASE
- 对用于RETURNING语句的所有Column需要有以下权限中的一个
  - 对用于RETURNING语句的所有Column有SELECT(columns) ON TABLE
  - 对表有(SELECT或CONTROL TABLE) ON TABLE
  - 对表所在的SCHEMA有(SELECT TABLE或CONTROL SCHEMA) ON SCHEMA
  - SELECT ANY TABLE ON DATABASE

语句规则及参数

table_name

要更新row的对象表名

[AS alias_name]

table_name的别名

<set clause>

定义要更新的column与要分配的值<set clause>的column数量应与值的数量相同

详细内容参考**UPDATE**

WHERE <search condition>

更新满足WHERE条件的row

不指定WHERE条件时更新所有的row

WHERE条件的详细内容参考**SELECT**语句的**where clause**

<result offset clause>

指定查询结果中要跳过的row数

详细内容参考**SELECT**语句的**<result offset clause>**

<fetch limit clause>

指定要fetch的row数的语句可使用以下两种方法

- <fetch first clause>
  - 指定要fetch的row数
  - 详细内容参考**SELECT**语句的**<fetch first clause>**
- <limit clause>
  - 指定要fetch的row数或同时指定查询结果中要跳过的row数与要fetch的row数

- 详细内容参考**SELECT**语句的**<limit clause>**

RETURNING .. AS ..

将更新的row作为结果集并描述要在其中搜索的column

详细内容参考**UPDATE name RETURNING**语句的**<returning clause>**

INTO variable_name [, ...]

INTO中描述的变量数量应与RETURNING中描述的expression数量相同

更新的行数据应为一条以下

更新两条以上row时报错

说明

详细内容参考**UPDATE相关语句之间的区别**

使用示例

以下为在通过主机变量获取更新的行的Column值的示例

- 声明主机变量

```
gSQL> \VAR v_discount NUMBER
```

```
gSQL> UPDATE lineitem
         SET l_discount = l_discount + 0.01
       WHERE l_orderkey = 12 AND l_linenumber = 5
   RETURNING l_discount INTO :v_discount;

V_DISCOUNT
----------
       .05

1 row updated.
```

兼容性

标准SQL中没有<update returning into statement>语句

10.30 UPDATE name WHERE CURRENT OF cursor_name

功能

更新游标指向的一条row

语句

```
<update statement: positioned> ::=
    UPDATE table_name [ [ AS ] alias_name ]
        SET <set clause> [, ...]
        WHERE CURRENT OF cursor_name
    ;
```

使用范围及访问权限

用户应满足以下条件才能执行<update statement: positioned>语句

- 用户需要有以下权限中的一个才能执行UPDATE语句
    - 对update的所有column有UPDATE(columns) ON TABLE
    - 对表有(UPDATE或CONTROL TABLE) ON TABLE
    - 对表所在的SCHEMA有(UPDATE TABLE或CONTROL SCHEMA) ON SCHEMA

- UPDATE ANY TABLE ON DATABASE

语句规则及参数

table_name

要更新row的对象表名

[AS alias_name]

table_name的别名

<set clause>

定义变更的Column与分配的值<set clause>的Column数量应与值的数量应该相同

详细内容参考**UPDATE**

cursor_name

cursor_name对应的游标需要满足以下条件

- 应为已打开的游标(参考**OPEN cursor_name**)
- 需要有通过游标FETCH的row(参考**FETCH cursor_name**)
- 游标使用的语句应可识别table_name(参考**DECLARE cursor_name**)
- 应为可以更新table_name的游标(参考**DECLARE cursor_name**)

说明

详细内容参考**UPDATE相关语句之间的区别**

使用示例

以下为interactive sql(gsql)中使用游标执行<update statement: positioned> 语句的示例

- 声明主机变量

```
gSQL> \VAR v_discount NUMBER
```

- 声明游标

```
gSQL> DECLARE update_cursor CURSOR FOR
        SELECT l_discount
          FROM lineitem
         WHERE l_orderkey = 8 AND l_linenumber = 1
           FOR UPDATE;


Cursor declared.
```

- 打开游标

```
gSQL> OPEN update_cursor;
```

```
Cursor is open.
```

- Fetch行

```
gSQL> FETCH update_cursor INTO :v_discount;

V_DISCOUNT
----------
       .06

1 row fetched.
```

- 更新current row

```
gSQL> UPDATE lineitem
         SET l_discount = l_discount + 0.01
       WHERE CURRENT OF update_cursor;

1 row updated.
```

- 关闭游标

```
gSQL> CLOSE update_cursor;

Cursor closed.
```

```
gSQL> COMMIT;


Commit complete.
```

以下为embedded SQL程序中使用游标执行<update statement: positioned>语句的示例

```
{
    ...
    EXEC SQL BEGIN DECLARE SECTION;
        ...
        double v_discount;
        ...
    EXEC SQL END DECLARE SECTION;
    ...
    EXEC SQL DECLARE update_cursor CURSOR FOR
              SELECT l_discount
                FROM lineitem
               WHERE l_orderkey = 8 AND l_linenumber = 1
                 FOR UPDATE;
    ...
    EXEC SQL OPEN update_cursor;
    ...
    EXEC SQL FETCH NEXT update_cursor INTO :v_discount;
    ...
    EXEC SQL UPDATE lineitem
                SET l_discount = l_discount + 0.01
```

```
            WHERE CURRENT OF update_cursor;

    ...

    EXEC SQL CLOSE update_cursor;

    ...

    EXEC SQL COMMIT WORK;

    ...

}
```

兼容性

Feature ID	说明	是否支持
F831	Full cursor update	O
B031	Basic dynamic SQL	O

Table 10-29 标准SQL兼容性

参考

相关内容参考**CLOSE cursor_name**