GBASE

GBase 8a 分析型数据管理系统

SQL 参考手册 V8.5.1.2

www.gbase.cn

GBase 8a SQL 参考手册，南大通用数据技术股份有限公司

通讯方式
南大通用数据技术股份有限公司
天津华苑产业区海泰发展六道 6 号海泰绿色产业基地 J 座(300384)
电话：400-817-9696　　邮箱：info@gbase.cn

目 录

前言

手册简介

GBase 8a SQL 参考手册介绍 GBase 8a 中可以使用的 SQL 语句，包括数据类型、操作符和函数、DDL 和 DML 语句，以及存储过程和自定义函数，手册中还提供了示例以供读者参考。

第一章详细介绍了 GBase 8a 支持的数据类型，包括数值类型、字符类型、二进制数据类型、日期和时间类型，并对数据类型存储需求、如何选择数据类型进行了描述。

第二章详细介绍了 SQL 语言基础知识，包括数值，数据库、表、列和别名，如何设置和使用用户变量、系统变量和注释语法。

第三章详细介绍 GBase 8a 支持的操作符和函数，包括操作符、控制流函数、字符串函数、数值函数、日期和时间函数、其他函数，GROUP BY 子句的函数和修饰语、OLAP 函数和 ROWID 函数。

第四章详细介绍 SQL 语句语法，包括 DDL 语句、DML 语句、查询结果导出语句、GBase 8a 事务和锁语句、GBase 8a 其他语句和数据库管理语句。

第五章详细介绍存储过程、函数，包括存储程序（过程和函数）概述、语法格式以及存储程序（过程和函数）的使用限制。

附录对 GBase 8a 的 SQL 保留字进行了汇总。

公约

下面的文本约定用于本文档：

约　定	说　明
加粗字体	表示文档标题
大写英文（SELECT）	表示 GBase 8a 关键字
等宽字体	表示代码示例
...	表示被省略的内容。

1 数据类型

GBase 8a 支持 SQL92 中定义的大多数数据类型，同时也支持 SQL99 和 SQL2000 中定义的数据类型。

GBase 8a 支持的数据类型，如下表所示：

GBase 8a 的数据类型	
数值型	TINYINT
	INT
	BIGINT
	FLOAT
	DOUBLE
	DECIMAL
字符型	CHAR
	VARCHAR
	TEXT
二进制类型	BLOB
日期和时间型	DATE
	DATETIME
	TIME
	TIMESTAMP

1.1 数值类型

GBase 8a 支持数据类型包括严格的数值数据类型（TINYINT，INT，BIGINT，DECIMAL），以及近似的数值数据类型（FLOAT，DOUBLE）。

为了更有效地使用存储空间，请用户尽量使用最精确的类型。例如，如果一个整数列被用于在 1～127 之间的值，TINYINT 是最好的类型。

为了存储更大范围的数值，用户可以选择 BIGINT 或 DECIMAL 类型。

作为 SQL92 标准的扩展，GBase 8a 也支持整数类型 TINYINT，SMALLINT 和

BIGINT。

GBase 8a 支持的数值类型，如下表所示。

类型名称	最小值	最大值	占用字节数
TINYINT	-127	127	1
SMALLINT	-32767	32767	2
INT(INTEGER)	-2147483647	2147483647	4
BIGINT	-9223372036854775806	9223372036854775806	8
FLOAT	-3.40E+38	3.40E+38	4
DOUBLE	-1.7976931348623157E+308	1.7976931348623157E+308	8
DECIMAL[(M[,D])]	-(1E+M -1)/(1E+D)	(1E+M -1)/(1E+D)	动态计算

1.1.1 TINYINT

整数类型，TINYINT 范围是-127 到 127。TINYINT 占用 1 个字节。

1.1.2 SMALLINT

整数类型。SMALLINT 范围是-32767 到 32767。SMALLINT 占用 2 个字节。

1.1.3 INT

整数类型。INTEGER 的同义词。INT 范围是-2147483647 到 2147483647。INT 占用 4 个字节。

1.1.4 BIGINT

整数类型。

BIGINT 范围是-9223372036854775806 到 9223372036854775806。

BIGINT 占用 8 个字节。

示例 1：定义的列数据类型为 BIGINT。

示例中用到的表及数据：

```
DROP TABLE IF EXISTS products;
CREATE TABLE products(productnum BIGINT);
INSERT INTO products(productnum) VALUES(100);
```

```
gbase> SELECT productnum FROM products;
+------------+
| productnum |
+------------+
|        100 |
+------------+
1 row in set
```

1.1.5 FLOAT

FLOAT 代表一个浮点型数值，占用 4 个字节，它所存储的数值不是一个准确值。允许的值是-3.402823466E+38 到-1.175494351E-38，0，1.175494351E-38 到 3.402823466E+38。这些是理论限制，基于 IEEE 标准。实际的范围根据硬件或操作系统的不同可能稍微小些。

注意，定义 FLOAT 数据列时，M 和 D（M 是整数位数和小数位数的总位数，D 是小数的个数）可以同时定义，D 不能单独省略。

示例 1：定义的列数据类型为 FLOAT。

示例中用到的表及数据：

```
DROP TABLE IF EXISTS products;
CREATE TABLE products(productnum FLOAT);
```

```
INSERT INTO products(productnum) VALUES(-19000.44365),
(-19000.48365),(1.44365),(1.443658);

gbase> SELECT productnum FROM products;
+------------+
| productnum |
+------------+
|   -19000.4 |
|   -19000.5 |
|    1.44365 |
|    1.44366 |
+------------+
4 rows in set
```

示例 2：定义的列数据类型为 FLOAT（M），总位数小于等于 23 时，小数部分只保留一位有效数字，系统会自动对数字进行四舍五入。

定义的列数据类型为 FLOAT（M），总位数大于 23 时，小数位最大支持 15 位数字。

示例中用到的表及数据：

```
DROP TABLE IF EXISTS products;
CREATE TABLE products (a FLOAT(20),b FLOAT(28));
INSERT INTO products (a,b) VALUES(-19000.44365,-19000.44365);

gbase> SELECT * FROM products;
+----------+--------------+
| a        | b            |
+----------+--------------+
| -19000.4 | -19000.44365 |
+----------+--------------+
1 row in set
```

示例 3：定义的列数据类型为 FLOAT(7,4)，插入的数据为 999.00009 时，其近似值就是 999.0001，自动四舍五入。

示例中用到的表及数据：

```
DROP TABLE IF EXISTS products;
CREATE TABLE products(productnum FLOAT(7,4));
INSERT INTO products(productnum) VALUES(999.00009);

gbase> SELECT productnum FROM products;
+------------+
| productnum |
+------------+
|   999.0001 |
+------------+
1 row in set
```

示例 4：定义的列数据类型为 FLOAT(26,5)，指定精度为 5，则小数部分保留 5 位数字。

示例中用到的表及数据：

```
DROP TABLE IF EXISTS products;
CREATE TABLE products(productnum FLOAT(26,5));
INSERT INTO products(productnum) VALUES(19000.44365);

gbase> SELECT productnum FROM products;
+-------------+
| productnum  |
+-------------+
| 19000.44336 |
+-------------+
1 row in set
```

1.1.6 DOUBLE

DOUBLE 代表一个浮点型数值，占用 8 个字节，它所存储的数值不是一个准确值。

允许的值是-1.7976931348623157E+308 到-2.2250738585072014E-308、0 和 2.2250738585072014E-308 到 1.7976931348623157E+308。这些是理论限制，基于 IEEE 标准。实际的范围根据硬件或操作系统的不同可能稍微小些。

注意，定义 DOUBLE 数据列时，M 和 D（M 是整数位数和小数位数的总位数，D 是小数的个数），可以同时省略，D 不能单独省略。

示例 1：定义的列数据类型为 DOUBLE。

示例中用到的表及数据：

```
DROP TABLE IF EXISTS products;
CREATE TABLE products(productnum DOUBLE);
INSERT INTO products(productnum) VALUES(-19000.44365);
```

```
gbase> DESC products;
+------------+--------+------+-----+---------+-------+
| Field      | Type   | Null | Key | Default | Extra |
+------------+--------+------+-----+---------+-------+
| productnum | double | YES  |     | NULL    |       |
+------------+--------+------+-----+---------+-------+
1 row in set
```

```
gbase> SELECT productnum FROM products;
+--------------+
| productnum   |
+--------------+
| -19000.44365 |
+--------------+
1 row in set
```

1.1.7 DECIMAL

DECIMAL[(M[, D])]代表一个精确值，它所存储的数值范围是-(1E+M -1)/(1E+D)到(1E+M -1)/(1E+D)。

在 DECIMAL[(M[, D])]数据类型中，M 是总位数，支持的最大长度为 65；D 是小数点后面的位数，支持的最大长度为 30。

在不需要过高的数字精度的场景中，DECIMAL 中的 M 可以定义为 M≤18，这样可以获得更好的查询性能。

DECIMAL 用来存储那些严格要求数字精度的数据，例如货币数据，在这种情况下需要指定精度：

```
salary DECIMAL(5,2)
```

在 DECIMAL(5,2)中，5 表示总位数（整数位和小数位的位数总和），2 是小数位数。可以存储在 salary 列的最小值是-999.99，最大值是 999.99。

DECIMAL 值的最大范围受限于给定的精度和小数范围。超过小数范围时，会按四舍五入的原则截断为设定小数位数。

在定义 DECIMAL 数据列时，如果 M 和 D 同时省略，则 M 取值为 10，D 取值为 0，即 DECIMAL(10,0)，如果只指定 M 值，省略 D 值，那么插入一个非整数值的数字时，将按照四舍五入的原则截取到整数位。

示例 1：定义的列数据类型为 DECIMAL（18,5）。

示例中用到的表及数据：

```
DROP TABLE IF EXISTS products;
CREATE TABLE products(productnum DECIMAL (18,5));
INSERT INTO products(productnum) VALUES(19000.44365);
```

```
gbase> DESC products;
+------------+--------------+------+-----+---------+-------+
| Field      | Type         | Null | Key | Default | Extra |
+------------+--------------+------+-----+---------+-------+
| productnum | decimal(18,5) | YES  |     | NULL    |       |
+------------+--------------+------+-----+---------+-------+
```

```
gbase> SELECT productnum FROM products;
+-------------+
| productnum  |
```

```
+-------------+
| 19000.44365 |
+-------------+
1 row in set
```

示例 2：定义的列数据类型为 DECIMAL，M 和 D 均省略，那么 M 默认值为 10，D 默认值为 0。

```
gbase> CREATE TABLE products(productnum DECIMAL);
Query OK, 0 rows affected

gbase> DESC products;
+------------+---------------+------+-----+---------+-------+
| Field      | Type          | Null | Key | Default | Extra |
+------------+---------------+------+-----+---------+-------+
| productnum | decimal(10,0) | YES  |     | NULL    |       |
+------------+---------------+------+-----+---------+-------+
1 row in set
```

示例 3：定义列数据类型为 DECIMAL (M,D)，插入的数据超出总位数 M 时，报告错误信息；超出精度 D 时，则小数部分四舍五入。

```
gbase> CREATE TABLE products(productnum DECIMAL(8,3));
Query OK, 0 rows affected

gbase> INSERT INTO products(productnum) VALUES(191220.443);
ERROR 1264 (22003): Out of range value for column 'productnum' at row 1

gbase> INSERT INTO products(productnum) VALUES(19122.4436);
Query OK, 1 row affected

gbase> SELECT productnum FROM products;
+------------+
| productnum |
+------------+
|  19122.444 |
+------------+
```

1 row in set

1.2 字符类型

GBase 8a 目前只支持以下几种字符类型，如下表所示。

类型名称	最大长度（字符）
CHAR	255
VARCHAR	10922
TEXT	10922

1.2.1 CHAR

CHAR(M)

CHAR 类型仅仅是为了兼容 SQL 标准，因此，不建议使用者在实际的项目应用场景使用此数据类型，建议使用 VARCHAR 数据类型。

CHAR 是 CHARACTER 的缩写。m 表示该列中字符串的长度，其范围是 1 到 255 个字符。

当存储 CHAR 值时，它们会被用空格右填充到指定长度。当读取 CHAR 值时，填充的空格依旧保留。

如果给一个定义为 CHAR 类型的列插入一个超出最大长度的字符串，那么将报告报错。

1.2.2 VARCHAR

VARCHAR(M)

可变长字符串。M 表示表示该列中串的长度。M 的范围是 1 到 10922 个字符。

当存储 VARCHAR 类型的数据时，不会用空格填充补足列定义长度，存储的

数据包含空格时，保留空格。

示例 1：VARCHAR 数据类型不会补足列定义的长度，但会保留插入的数据中的空格。

示例中用到的表及数据：

```
DROP TABLE IF EXISTS products;
CREATE TABLE products (productName VARCHAR(100));
INSERT INTO products(productName) VALUES('南大通用');
INSERT INTO products(productName) VALUES('南大通用  ');
```

```
gbase> SELECT productName,  LENGTH(productName) AS length,
CHAR_LENGTH(productName) AS char_length FROM products;
+---------------+--------+-------------+
| productName   | length | char_length |
+---------------+--------+-------------+
| 南大通用      |     12 |           4 |
| 南大通用      |     14 |           6 |
+---------------+--------+-------------+
2 rows in set

gbase> SELECT productName FROM products WHERE productName = '南大通用';
+-------------+
| productName |
+-------------+
| 南大通用    |
+-------------+
1 row in set
```

查询结果中保留原始数据中的空格：

```
gbase> SELECT productName FROM products WHERE productName = '南大通用  ';
+-------------+
| productName |
+-------------+
| 南大通用    |
+-------------+
```

```
1 row in set
```

1.2.3 TEXT

TEXT 类型仅仅是为了兼容其它数据库的类型，推荐使用 VARCHAR 类型。

TEXT 类型最大支持 10922 字符的存储长度，定义 TEXT 列时，不能为它指定 DEFAULT 值。

1.3 二进制数据类型

GBase 8a 目前支持以下二进制数据类型，如下表所示：

类型名称	最大长度（字节）
BLOB	32767

BLOB 保存二进制数据，最大长度为 32767 字节。

使用 BLOB 数据类型，有如下约束：

- BLOB 列支持 32KB 的存储容量。
- 创建表时，BLOB 列不可以有 DEFAULT 值。
- 查询语句中，BLOB 列不支持过滤条件。
- 查询语句中，BLOB 列不支持 OLAP 函数。

1.4 日期和时间类型

GBase 8a 目前支持 4 种日期和时间类型，如下表所示。

类型名称	最小值	最大值	格式
DATE	0001-01-01	9999-12-31	YYYY-MM-DD
DATETIME	0001-01-01	9999-12-31	YYYY-MM-dd

类型名称	最小值	最大值	格式
	00:00:00.000000	23:59:59	HH:MI:SS.ffffff
TIME	-838:59:59	838:59:59	HHH:MI:SS
TIMESTAMP	1970-01-01 08:00:01	2038-01-01 00:59:59	YYYY-MM-DD HH:MI:SS

当使用日期和时间类型时，用户应当提供正确的格式：如，YYYY-MM-DD、YYYY-MM-DD HH:MI:SS。

1.4.1 DATE

日期类型。支持的范围是“0001-01-01”到“9999-12-31”。

GBase 8a 以“YYYY-MM-DD”格式显示 DATE 值。

示例 1：插入一个标准的 DATE 值。

```
gbase> DROP TABLE IF EXISTS products;
Query OK, 0 rows affected

gbase> CREATE TABLE products (productDate DATE);
Query OK, 0 rows affected

gbase> INSERT INTO products(productDate) VALUES('2010-09-01');
Query OK, 1 row affected

gbase> SELECT productDate FROM products;
+-------------+
| productDate |
+-------------+
| 2010-09-01  |
+-------------+
1 row in set
```

示例 2：插入一个 NULL 值。

```
gbase> DROP TABLE IF EXISTS products;
```

```
Query OK, 0 rows affected

gbase> CREATE TABLE products (productDate DATE);
Query OK, 0 rows affected

gbase> INSERT INTO products(productDate) VALUES(NULL);
Query OK, 1 row affected

gbase> SELECT productDate FROM products;
+-------------+
| productDate |
+-------------+
| NULL        |
+-------------+
1 rows in set
```

示例 3：插入一个非法的 DATE 值，系统报告错误信息。

```
gbase> DROP TABLE IF EXISTS products;
Query OK, 0 rows affected

gbase> CREATE TABLE products (productDate DATE);
Query OK, 0 rows affected

gbase> INSERT INTO products(productDate) VALUES('2010-09-31');
ERROR 1292 (22007): Incorrect date value: '2010-09-31' for column 'productDate'
at row 1
```

1.4.2 TIME

GBase 8a 以“HH:MI:SS”格式（或“HHH:MI:SS”格式）检索和显示 TIME 值，该值为字符串。

TIME 值的范围可以从“-838:59:59”到“838:59:59”。TIME 类型不仅可

以用于表示一天的时间（这一定不会超过 24 小时），而且可以用来表示所经过的时间或两个事件之间的时间间隔（这可能比 24 小时大许多或是一个负值）。

对于以字符串指定的包含时间定界符的 TIME 值，不必要为小于 10 的时、分或秒指定两位数值。“8:3:2”与“08:03:02”是一致的。

示例 1：插入一个合法的 TIME 值。

```
gbase> DROP TABLE IF EXISTS products;
Query OK, 0 rows affected

gbase> CREATE TABLE products (productDate TIME);
Query OK, 0 rows affected

gbase> INSERT INTO products(productDate) VALUES('12:09:44');
Query OK, 1 row affected

gbase> SELECT productDate FROM products;
+-------------+
| productDate |
+-------------+
| 12:09:44    |
+-------------+
1 row in set
```

示例 2：插入一个在“-838:59:59”到“838:59:59”之间，且超过 24 小时之间的 TIME 值。

```
gbase> DROP TABLE IF EXISTS products;
Query OK, 0 rows affected

gbase> CREATE TABLE products (productDate TIME);
Query OK, 0 rows affected

gbase> INSERT INTO products(productDate) VALUES('92:09:44');
Query OK, 1 row affected
```

```
gbase> SELECT productDate FROM products;
+-------------+
| productDate |
+-------------+
| 92:09:44    |
+-------------+
1 row in set
```

1.4.3 DATETIME

GBase 8a 以“YYYY-MM-DD HH:MI:SS.ffffff”格式显示 DATETIME 值。其中 ffffff 表示微秒格式。

日期和时间的组合类型。支持的范围是“0001-01-01 00:00:00.000000”到“9999-12-31 23:59:59.999999”。

示例 1：插入一个合法的 DATETIME 值。

```
gbase> DROP TABLE IF EXISTS products;
Query OK, 0 rows affected

gbase> CREATE TABLE products (productDate DATETIME);
Query OK, 0 rows affected

gbase> INSERT INTO products(productDate) VALUES('2010-09-01 12:09:44');
Query OK, 1 row affected

gbase> SELECT productDate FROM products;
+---------------------+
| productDate         |
+---------------------+
| 2010-09-01 12:09:44 |
+---------------------+
1 row in set
```

示例 2：插入系统当前的 DATATIME 值。

```
gbase> DROP TABLE IF EXISTS products;
Query OK, 0 rows affected

gbase> CREATE TABLE products (productDate DATETIME);
Query OK, 0 rows affected

gbase> INSERT INTO products(productDate) VALUES(Now());
Query OK, 1 row affected

gbase> SELECT productDate FROM products;
+---------------------+
| productDate         |
+---------------------+
| 2013-12-13 17:44:23 |
+---------------------+
1 row in set
```

示例 3：插入一个 NULL 值。

```
gbase> DROP TABLE IF EXISTS products;
Query OK, 0 rows affected

gbase> CREATE TABLE products (productDate DATETIME);
Query OK, 0 rows affected

gbase> INSERT INTO products(productDate) VALUES(NULL);
Query OK, 1 row affected

gbase> SELECT productDate FROM products;
+-------------+
| productDate |
+-------------+
| NULL        |
+-------------+
1 row in set
```

示例 4：插入一个带有微秒的 DATETIME 数值。

```
gbase> INSERT INTO products(productDate) VALUES('2013-09-15
12:09:44.123456');
Query OK, 1 row affected

gbase> SELECT productDate FROM products;
+----------------------------+
| productDate                |
+----------------------------+
| 2013-09-15 12:09:44.123456 |
+----------------------------+
1 row in set
```

示例 5：插入一个非法的 DATETIME 值，系统将报告错误信息。

```
gbase> DROP TABLE IF EXISTS products;
Query OK, 0 rows affected

gbase> CREATE TABLE products (productDate DATETIME);
Query OK, 0 rows affected

gbase> INSERT INTO products(productDate) VALUES('2010-09-31 12:09:44');
ERROR 1292 (22007): Incorrect datetime value: '2010-09-31 12:09:44' for column
'productDate' at row 1
```

1.4.4 TIMESTAMP

TIMESTAMP 类型仅仅是为了兼容 SQL 标准，因此，不建议使用者在实际的项目应用场景使用此数据类型，推荐使用 DATETIME 数据类型。

TIMESTAMP 的格式为“YYYY-MM-DD HH:MI:SS”，支持的范围是“1970-01-01 08:00:01”到“2038-01-01 00:59:59”。

使用 DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP 属性时，TIMESTAMP 列支持 INSERT、UPDATE 以及 MERGE 时，TIMESTAMP 列的值自动更新，

但是 DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP 在一张表中只能使用一次，也就是含有多个 TIMESTAMP 列时，只能给第一次出现 TIMESTAMP 的列使用 DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP 属性。

创建一张表时，如果只定义一个 TIMESTAMP 列，DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP 可以省略，系统会自动添加上。

示例：

```
gbase> CREATE TABLE t (a int,b timestamp DEFAULT CURRENT_TIMESTAMP ON UPDATE
CURRENT_TIMESTAMP, c timestamp DEFAULT '2013-01-01 00:00:01');
Query OK, 0 rows affected

gbase> SHOW CREATE TABLE t;
+-------+-----------------------------------------------------------------+
| Table | Create Table                                                    |
+-------+-----------------------------------------------------------------+
| t     | CREATE TABLE "t" (
  "a" int(11) DEFAULT NULL,
  "b" timestamp NOT NULL DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP,
  "c" timestamp NOT NULL DEFAULT '2013-01-01 00:00:01'
) ENGINE=EXPRESS DEFAULT CHARSET=utf8 TABLESPACE='sys_tablespace'          |
+-------+-----------------------------------------------------------------+
1 row in set
```

TIMESTAMP 使用限制：

以下限制说明，是针对 TIMESTAMP 数据列自动更新时的场景：

1. 在一张表中，只能自动更新表中第一个出现的 TIMESTAMP 列，并且必须使用 DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP 来定义。
2. 在一张表中，使用 DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP 来指定 TIMESTAMP 列自动更新时，只能使用一次，且用于第一个 TIMESTAMP 列。

2 SQL 语言基础

本章对 GBase 8a 的 SQL 语句的下列元素进行讨论：

数值，例如字符串和数字；

标识符，例如表和列名字；

用户和系统变量；

注释。

2.1 数值

这部分主要介绍在 GBase 8a 中使用的数值。包括字符串，数字，十六进制值，布尔值和 NULL。

2.1.1 字符串

字符串是多个字符组成的一个字符序列，由单引号“’”或双引号“″”字符包围。

例如：’a string’

在一个字符串中，确定的序列具有特殊的含义，每个序列以反斜线符号“\”开头，称为转义字符。GBase 8a 识别下列转义字符：

转义符	描　述
\0	ASCII 0 (NUL) 字符。
\’	ASCII 39 单引号“’”字符。
\″	ASCII 34 双引号“″”字符。
\b	ASCII 8 退格符。
\n	ASCII 10 换行符。
\r	ASCII 13 回车符。
\t	ASCII 9 制表符 (TAB)。
\\	ASCII 92 反斜线“\”字符。

这些符号是大小写敏感的。例如：“\b”被解释为一个退格，但是“\B”被解释为“B”。

反斜线用来解释转义字符而不是被转义。

字符串中包含引号时，可以有下列几种写法：

字符串用单引号“'”来引用的，该字符串中的单引号“'”字符可以用“''”方式转义。

用户也可以继续使用在引号前加一个转义字符“\”来转义的方式。

字符串是用单引号“'”来引用的，该字符串中的双引号“"”不需要特殊对待而且不必被重复或转义。

下面的示例说明了 SELECT 语句对引号和转义是如何工作的。

示例 1：使用单引号“'”，双引号“"”，转义字符“\”包围字符串。

```
gbase> SELECT 'hello', '"hello"', '""hello""', 'hel''lo', '\'hello' FROM t;
+-------+---------+-----------+--------+--------+
| hello | "hello" | ""hello"" | hel'lo | 'hello |
+-------+---------+-----------+--------+--------+
| hello | "hello" | ""hello"" | hel'lo | 'hello |
+-------+---------+-----------+--------+--------+
1 row in set
```

示例 2：字符串中存在转义字符“\”。

```
gbase> SELECT 'This\nIs\nFour\nLines' FROM t;
+--------------------+
| This
Is
Four
Lines |
+--------------------+
| This
Is
```

```
Four
Lines |
+-------------------+
1 row in set
```

示例 3：不存在转义含义时，忽略反斜线符号。

```
gbase> SELECT 'disappearing\ backslash' FROM t;
+-----------------------+
| disappearing backslash |
+-----------------------+
| disappearing backslash |
+-----------------------+
1 row in set
```

如果用户想要把二进制数据插入到 BLOB 字段中，下列字符必须由转义序列表示：

字　符	描　述
NUL	NUL byte　(ASCII 0)。需要用“\0”(一个反斜线和一个 ASCII“0”字符）表示。
\	反斜线 (ASCII 92)。需要用“\\”表示。
’	单引号 (ASCII 39)。需要用“\’”表示。
″	双引号 (ASCII 34)。需要用“\″”表示。

示例 4：创建的表中 productBlob 字段的类型为 BLOB，插入的数据中存在转义字符。

```
gbase> DROP TABLE IF EXISTS products;
Query OK, 0 rows affected

gbase> CREATE TABLE products (productBlob BLOB);
Query OK, 0 rows affected

gbase> INSERT INTO products values('abdcdrf\\ghi\'jklm\"nopqrs\0tuvwxyz');
Query OK, 1 row affected
```

```
gbase> SELECT productBlob FROM products;
+------------------------------+
| productBlob                  |
+------------------------------+
| abdcdrf\ghi'jklm"nopqrs tuvwxyz |
+------------------------------+
1 row in set
```

当写程序代码时，任何一个字符串都有可能包含这些特殊的字符，因此在这些字符作为SQL语句中的数据传到GBase 8a服务器之前必须进行适当的转义。

2.1.2 数字

整数被表示为一个数字序列。浮点数使用“.”作为一个十进制数的分隔符。这两个数字类型可以前置“-”以表示一个负值。

有效整数的示例：

1221、0、-32

有效浮点数的示例：

-32032.6809E+10、148.00E+13

2.1.3 十六进制值

GBase 8a支持十六进制数值。

在数字的上下文语境中，它们作为等价于整数使用。

在字符串的上下文语境中，它们作为一个字符串，每一对十六进制数字被解释为对应ASCII码的字符。

示例1：将“4742617365”转换成对应的ASCII码。

```
gbase> SELECT x'4742617365' FROM t;
```

```
+---------------+
| x'4742617365' |
+---------------+
| GBase         |
+---------------+
1 row in set
```

示例 2： 0xa 等价于整数 10。

```
gbase> SELECT 0xa+0 FROM t;
+-------+
| 0xa+0 |
+-------+
|    10 |
+-------+
1 row in set
```

示例 3：将“5061756c”转换成对应的 ASCII 码。

```
gbase> SELECT 0x5061756c FROM t;
+------------+
| 0x5061756c |
+------------+
| Paul       |
+------------+
1 row in set
```

表达式“x'hexstring'”是基于标准 SQL 的，表达式 0x 是基于 ODBC 的。二者是等价的。

示例 4：使用 HEX()函数可以将一个字符串或数值转换为一个十六进制格式的字符串。

```
gbase> SELECT HEX('cat') FROM t;
+------------+
| HEX('cat') |
+------------+
```

```
| 636174      |
+-------------+
1 row in set
```

```
gbase> SELECT 0x636174 FROM t;
+-----------+
| 0x636174 |
+-----------+
| cat       |
+-----------+
1 row in set
```

2.1.4 布尔值

常量 TURE 相当于 1，而常量 FALSE 相当于 0。

常量的名字对大小写不敏感。

示例 1：查询 TRUE 和 FALSE 对应的值。

```
gbase> SELECT TRUE, true, FALSE, false FROM t;
+------+------+-------+-------+
| TRUE | TRUE | FALSE | FALSE |
+------+------+-------+-------+
|    1 |    1 |     0 |     0 |
+------+------+-------+-------+
1 row in set
```

2.1.5 NULL 值

NULL 不区分大小写。

注意：NULL 值不同于数字类型的 0 或字符串类型的空串。

2.2 数据库、表、列和别名

数据库、表、列和别名等都是标识符，这部分描述 GBase 8a 中标识符允许的语法规则。

下面的表描述了对于每一个类型标识符允许的最大长度和可使用的字符。

标识符	最大长度(字符)	允许的字符
数据库	64	目录名允许的任何字符，除了“/” “.”
表	64	文件名允许的任何字符，除了“/” “.”
视图	64	所有字符
列	64	所有字符
别名	255	所有字符
存储过程	64	所有字符

注:

1、除了表内注明的限制，标识符不可以包含 ASCII(0)或 ASCII(255)。数据库、表和列名不应以空格结尾;

2、如果标识符是一个限制词或包含特殊字符，当用户使用它时，必须总是用'引用它，比如: SELECT * FROM 'select'.id>100;

3、如果标识符长度超过最大长度限制，数据库、表、列、视图、存储过程的命令将报错，而别名将会截断至 256 个字符进行显示。

GBase 8a 数据库支持的保留字，请参见附录部分的 GBase 8a 分析型数据库保留字。

实际应用系统中，标识符不得使用 GBase 8a 的保留字，也不能包含特殊字符。

2.3 标识符限定词

GBase 8a 允许名称由一个或多个标识符组成。组合名称的各个组成成分应

该用英文句号字符“.”分割开。组合名称的开始部分做为限定词来使用，它影响了上下文中后面的标识符的解释。

在 GBase 8a 中，用户可以使用下列表格中的任一种方式引用一个列：

列引用	含义
col_name	列 col_name 来自查询所用的任何一个表中对应字段。
table_name.col_name	列 col_name 来自当前数据库中的表 table_name。
database_name.table_name.col_name	列 col_name 来自数据库 database_name 中的表 table_name。
`column_name`	该字段是一个关键词或包含特殊字符。

组合标识符如果需要引用则标识符的各部分都要各自引用，而不是把组合标识符作为一个整体来引用。例如：`gs-table`.`gs-column`合法，whereas`gs-table.gs-column`不合法。

在一条语句的列引用中，不需要明确指定一个 *table_name* 或 *database_name.table_name* 前缀，除非这个引用存在二义性。

例如,假设表 t1 和 t2 均包含一个字段 c,当用一个使用了 t1 和 t2 的 SELECT 检索 c 时，在这种情况下，字段 c 存在二义性。因为它在这个语句所使用的表中不是唯一的，必须通过写出 t1.c 或 t2.c 来指明用户所需的是哪个表。

同样的，如果从数据库 db1 的表 t 和数据库 db2 的表 t 中检索，用户必须用 *db1.t.col_name* 和 *db2.t.col_name* 来指定引用哪个库表的列。

2.4 注释语法

GBase 8a 服务器支持三种注释风格。

到该行结束。

-- 到该行结束。注意“--”（引导号）注释风格要求第二个引导号后至少跟着一个空格（或者一个控制字符，例如，换行）。这个语法和标准的 SQL 注

释风格有点不同。

/*行中间或多个行*/。这个封闭的序列不需要在同一行表示，因此该语法支持多行注释。

示例 1：使用“#”注释。

```
gbase> SELECT 1+1 FROM t;# This comment continues to the END of line
+-----+
| 1+1 |
+-----+
|   2 |
+-----+
1 row in set
```

示例 2：使用“--”注释。

```
gbase> SELECT 1+1 FROM t;-- This comment continues to the END of line
+-----+
| 1+1 |
+-----+
|   2 |
+-----+
1 row in set
```

示例 3：使用“/*单行*/”注释。

```
gbase> SELECT 1 /* this is an in-line comment */ + 1 FROM t;
+-----+
| 1+1 |
+-----+
|   2 |
+-----+
1 row in set
```

示例 4：使用“/*多行*/”注释。

```
gbase> SELECT 1+
   /*
```

```
    this is a
    multiple-line comment
   */
    1 FROM t;
+--------------+
| 1+

    1 |
+--------------+
|            2 |
+--------------+
1 row in set
```

2.5 用户变量

GBase 8a 支持用户变量。用户变量的生命周期是会话级的，对其它会话不可见。当用户退出时，此用户的所有用户变量会自动释放。

用户变量的写法是：@var_name。一个变量名可以由 a～z、A～Z、0～9、下划线组成，必须以字母或下划线开头。

用户变量名大小写不敏感。

通过 SET 语法来定义并为变量赋值。

SET @var_name = expr [, @var_name = expr] ...

“=”是赋值操作符。赋给每一个变量的 expr 值可以是整数、实数、字符串、或 NULL。

通过 SELECT 语法查看用户变量的值。

SELECT @var_name [, @var_name] ...

示例 1：使用 SET 语句为变量赋值。

```
gbase> SET @t1='abc',@t2=null,@t3=4;
Query OK, 0 rows affected
```

```
gbase> SELECT @t1,@t2,@t3;
+------+------+------+
| @t1  | @t2  | @t3  |
+------+------+------+
| abc  | NULL |    4 |
+------+------+------+
1 row in set
```

用户变量可以用于表达式所允许的任何地方。注意，必须明确指定常量的上下文中不能使用变量，例如，在 SELECT 的 LIMIT 子句中。

如果用户使用的变量没有初始化，那么它的值就为 NULL。

3 操作符和函数

在 SQL 语句中可以使用表达式，表达式可以包含常量，字段，NULL，操作符和函数。本章描述 GBase 8a 中用于写在表达式中的操作符和函数。

包含 NULL 的表达式总是得出 NULL 值结果，除非表达式中的操作和函数在文档中有另外的说明。

本章中操作符和函数的执行示例使用表 t，并且该表已包含一行数据，此表并不是某个应用场景下的实际业务或者数据表，t 表信息如下：

```
gbase> DROP TABLE IF EXISTS t;
Query OK, 0 rows affected

gbase> CREATE TABLE t (a INT);
Query OK, 0 rows affected

gbase> INSERT INTO t VALUES(1);
Query OK, 1 row affected
```

3.1 操作符

3.1.1 操作符优先级

操作符优先级在下面列出，从最高到最低。同一行的操作符具有同样的优先级。

BINARY, COLLATE

!

-(unary minus), ~(unary bit inversion)

^

*, /, DIV, %, MOD

-, +

<<, >>

&

|

=, <=>, >=, >, <=, <, <>, !=, IS, LIKE, REGEXP, IN

BETWEEN, CASE, WHEN, THEN, ELSE

NOT

&&, AND

OR, XOR

:=

3.1.2 圆括号

(...)

括号，使用它来规定一个表达式的运算顺序，放在括号里的操作符优先执行。

示例 1：不使用括号，表达式先执行乘法操作，再执行加法操作。

```
gbase> SELECT 1+2*3 FROM t;
+-------+
| 1+2*3 |
+-------+
|     7 |
+-------+
1 row in set
```

示例 2：使用括号，表达式先执行括号中的加法操作，再执行括号外的乘法操作。

```
gbase> SELECT (1+2)*3 FROM t;
+---------+
| (1+2)*3 |
+---------+
|       9 |
+---------+
1 row in set
```

3.1.3 比较函数和操作符

比较运算的结果是 1（TRUE）、0（(FALSE）或 NULL。这些运算可用于数字和字符串上。根据需要，字符串将会自动地被转换到数字，而数字也可自动转换为字符串。

说明：本章中的一些函数（如 GREATEST()和 LEAST()）的所得值虽然不包括 1（TRUE）、0（FALSE）或 NULL，但对参数值进行比较时，也会基于下述规则。

GBase 8a 使用下列规则进行数值比较：

- 如果一个或两个参数是 NULL，比较的结果是 NULL，除了<=>比较符（含有 NULL 参数时，比较结果不是 NULL，例如，SELECT 1<=>NULL FROM t;）。
- 如果在一个比较操作中两个参数均是字符串，它们将作为字符串被比较。
- 如果两个参数均是整数，它们作为整数被比较。
- 如果比较操作中，一个参数为字符串，另一个为整数，则以操作符左侧参数的数据类型为准。
- 十六进制值如果不与一个数字进行比较，那么它将当作一个二进制字符串。
- 如果参数之一是 TIMESTAMP 或 DATETIME 列，而另一参数是一个常量，在比较执行之前，这个常量被转换为一个时间戳。需要注意的是 IN()

中的参数不是这样的。为了安全起见，建议用户在比较时使用完整的 DATETIME/DATE/TIME 字符串。

- 在所有其它情况下，参数作为浮点（REAL）数字被比较。
- 要转换一个值成为一个特殊的类型，用户可以使用 CAST()函数。

3.1.3.1 = 等于

a=b

如果两个操作数相等，则返回 1。

示例 1：两个操作数都是数字。

```
gbase> SELECT 1 = 0 FROM t;
+-------+
| 1 = 0 |
+-------+
|     0 |
+-------+
1 row in set
```

示例 2：数字与字符串进行比较。

```
gbase> SELECT '0' = 0 FROM t;
+---------+
| '0' = 0 |
+---------+
|       1 |
+---------+
1 row in set
```

示例 3：数字与字符串进行比较。

```
gbase> SELECT '0.0' = 0 FROM t;
+-----------+
| '0.0' = 0 |
```

```
+-----------+
|         1 |
+-----------+
1 row in set
```

示例 4：两个操作数都是 NULL。

```
gbase> SELECT NULL = NULL FROM t;
+--------------+
| NULL = NULL  |
+--------------+
|         NULL |
+--------------+
1 row in set
```

3.1.3.2 <=> NULL 值安全等于

NULL 值安全等于。这个操作符像“=”操作符一样执行相等比较。

如果所有的操作数是 NULL，那么返回的是 1 而不是 NULL。

如果有且只有一个操作数是 NULL，那么返回的是 0 而不是 NULL。

示例 1：所有操作数为 NULL，或部分操作数为 NULL。

```
gbase> SELECT 1 <=> 1, NULL <=> NULL, 1 <=> NULL FROM t;
+---------+---------------+-------------+
| 1 <=> 1 | NULL <=> NULL | 1 <=> NULL  |
+---------+---------------+-------------+
|       1 |             1 |           0 |
+---------+---------------+-------------+
1 row in set
```

示例 2：（=操作符的结果）

```
gbase> SELECT 1 = 1, NULL = NULL, 1 = NULL FROM t;
+-------+------------+-----------+
```

```
| 1 = 1 | NULL = NULL | 1 = NULL  |
+-------+-------------+-----------+
|     1 |        NULL |      NULL |
+-------+-------------+-----------+
1 row in set
```

3.1.3.3 <>, != 不等于

a<>b 或 a!=b

如果两个操作数不相等，则返回 1。

示例 1：操作数都为字符串。

```
gbase> SELECT '01'<>'1';
+-----------+
| '01'<>'1' |
+-----------+
|         1 |
+-----------+
1 row in set
```

示例 2：其中一个操作数为字符串。

```
gbase> SELECT 01<>'1';
+---------+
| 01<>'1' |
+---------+
|       0 |
+---------+
1 row in set
```

示例 3：操作数都为字符串。

```
gbase> SELECT 'zapp' <> 'zappp' FROM t;
+------------------+
| 'zapp' <> 'zappp' |
```

```
+------------------+
|                1 |
+------------------+
1 row in set
```

3.1.3.4 <= 小于或者等于

a<=b

如果 a 小于或等于 b，则返回 1。

示例 1：两个操作数都为数字。

```
gbase> SELECT 0.1 <= 2 FROM t;
+----------+
| 0.1 <= 2 |
+----------+
|        1 |
+----------+
1 row in set
```

3.1.3.5 < 小于

a<b

如果 a 小于 b，则返回 1。

示例 1：两个操作数都为数字。

```
gbase> SELECT 2 < 2 FROM t;
+-------+
| 2 < 2 |
+-------+
|     0 |
+-------+
1 row in set
```

3.1.3.6 >= 大于或者等于

a>=b

如果 a 大于或等于 b，则返回 1。

示例 1：两个操作数都为数字。

```
gbase> SELECT 2 >= 2 FROM t;
+--------+
| 2 >= 2 |
+--------+
|      1 |
+--------+
1 row in set
```

3.1.3.7 > 大于

a>b

如果 a 大于 b，则返回 1。

示例 1：两个操作数都为数字。

```
gbase> SELECT 2 > 2 FROM t;
+-------+
| 2 > 2 |
+-------+
|     0 |
+-------+
1 row in set
```

3.1.3.8 is boolean_value ,is not boolean_value

根据一个布尔值来检验一个值，此处的布尔值可以是 TRUE、FALSE 或 UNKNOWN。

示例 1：使用 IS 语句检验 1、0 和 NULL。

```
gbase> SELECT 1 IS TRUE, 0 IS FALSE, NULL IS UNKNOWN FROM t;
+-----------+------------+-----------------+
| 1 IS TRUE | 0 IS FALSE | NULL IS UNKNOWN |
+-----------+------------+-----------------+
|         1 |          1 |               1 |
+-----------+------------+-----------------+
1 row in set
```

示例 2：使用 IS NOT 语句检验 1、0 和 NULL。

```
gbase> SELECT 1 IS NOT UNKNOWN, 0 IS NOT UNKNOWN, NULL IS NOT UNKNOWN FROM t;
+-----------------+-----------------+---------------------+
| 1 IS NOT UNKNOWN | 0 IS NOT UNKNOWN | NULL IS NOT UNKNOWN |
+-----------------+-----------------+---------------------+
|               1 |               1 |                   0 |
+-----------------+-----------------+---------------------+
1 row in set
```

示例 3：使用 IS 语句检验一个值是否是 NULL。

```
gbase> SELECT 1 IS NULL, 0 IS NULL, NULL IS NULL FROM t;
+-----------+-----------+--------------+
| 1 IS NULL | 0 IS NULL | NULL IS NULL |
+-----------+-----------+--------------+
|         0 |         0 |            1 |
+-----------+-----------+--------------+
1 row in set
```

示例 4：使用 IS NOT 语句检验一个值是否是 NULL。

```
gbase> SELECT 1 IS NOT NULL, 0 IS NOT NULL, NULL IS NOT NULL FROM t;
+---------------+---------------+------------------+
| 1 IS NOT NULL | 0 IS NOT NULL | NULL IS NOT NULL |
+---------------+---------------+------------------+
```

```
|              1 |              1 |                  0 |
+----------------+----------------+--------------------+
1 row in set
```

3.1.3.9 expr BETWEEN min AND max

如果 expr 的值在 min 和 max 之间 (包括 min 和 max)，返回 1，否则返回 0。

若所有参数都是同一类型，则上述关系相当于表达式 min <= expr AND expr <= max。其它类型的转换根据本章开篇所述规律进行，且适用于三种参数中任意一种。

示例 1：所有参数为同一类型，expr 不在 min 和 max 中。

```
gbase> SELECT 1 BETWEEN 2 AND 3 FROM t;
+--------------------+
| 1 BETWEEN 2 AND 3 |
+--------------------+
|                  0 |
+--------------------+
1 row in set
```

示例 2：所有参数为同一类型，expr 在 min 和 max 中。

```
gbase> SELECT 'b' BETWEEN 'a' AND 'c' FROM t;
+------------------------+
| 'b' BETWEEN 'a' AND 'c' |
+------------------------+
|                       1 |
+------------------------+
1 row in set
```

示例 3：参数中包含数字和字符串。

```
gbase> SELECT 2 BETWEEN 2 AND '3' FROM t;
+--------------------+
| 2 BETWEEN 2 AND '3' |
+--------------------+
```

```
|                    1 |
+----------------------+
1 row in set
```

3.1.3.10 expr NOT BETWEEN min AND max

等同于 NOT(expr BETWEEN min AND max)。

3.1.3.11 COALESCE(value,...)

返回值为列表当中的第一个非 NULL 值，在全部为 NULL 值的情况下返回值为 NULL。

示例 1：其中一个参数值为 NULL。

```
gbase> SELECT COALESCE(NULL,1) FROM t;
+------------------+
| COALESCE(NULL,1) |
+------------------+
|                1 |
+------------------+
1 row in set
```

示例 2：每个参数值都为 NULL。

```
gbase> SELECT COALESCE(NULL,NULL,NULL) FROM t;
+--------------------------+
| COALESCE(NULL,NULL,NULL) |
+--------------------------+
|                     NULL |
+--------------------------+
1 row in set
```

3.1.3.12 GREATEST(value1,value2,...)

当有两个或多个参数时，返回值为最大的参数值。

当参数中有一个为 NULL 时，直接返回 NULL。

当参数都是字符串时，默认是不区分大小写的，如果希望字符串值的参数进行大小写敏感，则在需要敏感的字符串参数前加上 BINARY。

这些参数比较使用下列规则：

- 如果返回值在 INTEGER 上下文中或者所有的参数是整型值，那么它们使用整数比较；
- 如果返回值在 REAL 上下文中或者所有的参数是实数值，那么它们使用实数比较；
- 如果所有的参数是大小写敏感的字符串，那么参数比较也是大小写敏感的；
- 其它情况下，参数比较大小写不敏感。

示例 1：参数值为整型数字。

```
gbase> SELECT GREATEST(2,0) FROM t;
+--------------+
| GREATEST(2,0) |
+--------------+
|             2 |
+--------------+
1 row in set
```

示例 2：参数值为浮点型数字。

```
gbase> SELECT GREATEST(34.0,3.0,5.0,767.0) FROM t;
+----------------------------+
| GREATEST(34.0,3.0,5.0,767.0) |
+----------------------------+
|                       767.0 |
```

```
+----------------------------+
1 row in set
```

示例 3：参数值为字符串，不区分大小写。

```
gbase> SELECT GREATEST('B','a','C') FROM t;
+----------------------+
| GREATEST('B','a','C') |
+----------------------+
| C                    |
+----------------------+
1 row in set
```

示例 4：参数值为字符串，字符串参数前加上 BINARY，区分大小写。

```
gbase> SELECT GREATEST('B',BINARY 'a','C') FROM t;
+----------------------------+
| GREATEST('B',BINARY 'a','C') |
+----------------------------+
| a                          |
+----------------------------+
1 row in set
```

示例 5：参数值中包含 NULL，则执行结果为 NULL。

```
gbase> SELECT GREATEST('B',NULL,'C') FROM t;
+----------------------+
| GREATEST('B',NULL,'C') |
+----------------------+
| NULL                 |
+----------------------+
1 row in set
```

3.1.3.13 expr IN (value,...)

如果 expr 是 IN 列表中的任一值，它将返回 1，否则返回 0。

如果所有的值均是常量，那么所有的值被按照 expr 的类型进行计算和排序。

示例 1：expr 不是 IN 列表中的任一值。

```
gbase> SELECT 2 IN (0,3,5,'8') FROM t;
+------------------+
| 2 IN (0,3,5,'8') |
+------------------+
|                0 |
+------------------+
1 row in set
```

示例 2：expr 是 IN 列表中的值。

```
gbase> SELECT '1' IN (0,3,5,'1') FROM t;
+--------------------+
| '1' IN (0,3,5,'1') |
+--------------------+
|                  1 |
+--------------------+
1 row in set
```

如果左边的表达式是 NULL，或者在列表中没有发现相匹配的值并且列表中的一个表达式是 NULL，IN 均返回 NULL。IN()语法也可以用于子查询类型。

示例 3：expr 的值为 NULL。

```
gbase> SELECT NULL IN (0,3,5,'wefwf') FROM t;
+-------------------------+
| NULL IN (0,3,5,'wefwf') |
+-------------------------+
|                    NULL |
+-------------------------+
1 row in set
```

示例 4：子查询中包含 IN () 函数。

示例中用到的表及数据：

```
CREATE TABLE sc (sno VARCHAR(4), grade INT);
INSERT INTO sc VALUES ('101',82),('102',59),('103',90),('104',88),('106',82);
```

查询所有课程都及格的同学的学号。

```
gbase> SELECT sno FROM sc WHERE grade IN (SELECT grade FROM sc WHERE grade>60)
GROUP BY sno;
+------+
| sno  |
+------+
| 103  |
| 101  |
| 104  |
| 106  |
+------+
4 rows in set
```

3.1.3.14 expr NOT IN (value,...)

等价于 NOT(expr IN (value,...))。

3.1.3.15 ISNULL(expr)

如果 expr 为 NULL，ISNULL()的返回值为 1，否则返回值为 0。

示例 1：expr 的值不为 NULL。

```
gbase> SELECT ISNULL(1+1) FROM t;
+-------------+
| ISNULL(1+1) |
+-------------+
|           0 |
+-------------+
1 row in set
```

示例 2：1/0 的结果为 NULL，ISNULL()的返回值为 1。

```
gbase> SELECT ISNULL(1/0) FROM t;
+-------------+
| ISNULL(1/0) |
+-------------+
|           1 |
+-------------+
1 row in set
```

示例 3：对 NULL 值使用“=”进行比较，ISNULL 结果为 1。

```
gbase> SELECT ISNULL(NULL=NULL) FROM t;
+-------------------+
| ISNULL(NULL=NULL) |
+-------------------+
|                 1 |
+-------------------+
1 row in set
```

ISNULL()函数同 IS NULL 比较操作符具有一些相同的特性。IS NULL 的使用请参考“3.1.3.8 is boolean_value ,is not boolean_value”中的示例 3。

3.1.3.16 LEAST(value1,value2,...)

有两个或者更多的参数，返回最小的参数值。假如任意一个变量为 NULL，则 LEAST()的返回值为 NULL。

LEAST()对参数进行比较所依据的规则同 GREATEST()相同。

示例 1：参数值为整型数字，返回最小的参数值。

```
gbase> SELECT LEAST(2,0) FROM t;
+------------+
| LEAST(2,0) |
+------------+
|          0 |
+------------+
1 row in set
```

示例 2：参数值为浮点型数字，返回最小的参数值。

```
gbase> SELECT LEAST(34.0,3.0,5.0,767.0) FROM t;
+--------------------------+
| LEAST(34.0,3.0,5.0,767.0) |
+--------------------------+
|                      3.0 |
+--------------------------+
1 row in set
```

示例 3：参数值为字符串，不区分大小写。

```
gbase> SELECT LEAST('B','A','C') FROM t;
+--------------------+
| LEAST('B','A','C') |
+--------------------+
| A                  |
+--------------------+
1 row in set

gbase> SELECT LEAST('B','a', 'C') FROM t;
+---------------------+
| LEAST('B','a', 'C') |
+---------------------+
| a                   |
+---------------------+
1 row in set
```

示例 4：参数值为字符串，字符串参数前加上 BINARY，区分大小写。

```
gbase> SELECT LEAST(BINARY 'B',BINARY 'a', 'C') FROM t;
+----------------------------------+
| LEAST(BINARY 'B',BINARY 'a', 'C') |
+----------------------------------+
| B                                |
+----------------------------------+
1 row in set
```

示例 5：参数值中包含 NULL，则执行结果为 NULL。

```
gbase> SELECT LEAST('C',NULL,'B') FROM t;
+--------------------+
| LEAST('C',NULL,'B') |
+--------------------+
| NULL               |
+--------------------+
1 row in set
```

3.1.4 逻辑操作符

在 SQL 中,所有的逻辑操作符返回的值均为 TRUE、FALSE 或 NULL(UNKNOWN)，它们是由 1(TRUE)、0(FALSE)和 NULL 来实现的。

3.1.4.1 NOT，!逻辑非

如果操作数为 0，返回 1；如果操作数为非零，返回 0；如果操作数为 NULL，返回 NULL。

示例 1：操作数为非零，返回值为 0。

```
gbase> SELECT NOT 10 FROM t;
+--------+
| NOT 10 |
+--------+
|      0 |
+--------+
1 row in set
```

示例 2：操作数为 0，返回值为 1。

```
gbase> SELECT NOT 0 FROM t;
+-------+
```

```
| NOT 0 |
+-------+
|     1 |
+-------+
1 row in set
```

示例 3：操作数为 NULL，返回值为 NULL。

```
gbase> SELECT NOT NULL FROM t;
+-----------+
| NOT NULL  |
+-----------+
|      NULL |
+-----------+
1 row in set
```

示例 4：表达式的值为非零，返回值为 0。

```
gbase> SELECT ! (1+1) FROM t;
+---------+
| ! (1+1) |
+---------+
|       0 |
+---------+
1 row in set
```

示例 5：表达式！1+1 与(!1)+1 等价，执行结果为 1。

```
gbase> SELECT ! 1+1 FROM t;
+-------+
| ! 1+1 |
+-------+
|     1 |
+-------+
1 row in set
```

示例 6：..NOT IN...

```
gbase> SELECT 1 NOT IN (2,3,null) FROM t;
```

```
+--------------------+
| 1 NOT IN (2,3,null) |
+--------------------+
|                NULL |
+--------------------+
1 row in set
```

3.1.4.2 XOR 逻辑异或

当任意一个操作数为 NULL 时，返回值为 NULL。

对于非 NULL 的操作数:

真（1）异或假（0）的结果是真，假（0）异或真（1）的结果也是真。

真（1）异或真（1）的结果是假，假（0）异或假（0）的结果是假。

就是说两个值不相同，则异或结果为真，反之，为假。

示例 1：操作数不是 NULL，真异或真，结果为假，即返回值为 0。

```
gbase> SELECT 1 XOR 1 FROM t;
+---------+
| 1 XOR 1 |
+---------+
|       0 |
+---------+
1 row in set
```

示例 2：操作数不是 NULL，真异或假，结果为真，即返回值为 1。

```
gbase> SELECT 1 XOR 0 FROM t;
+---------+
| 1 XOR 0 |
+---------+
|       1 |
+---------+
1 row in set
```

示例 3：任意一个操作数为 NULL，则结果为 NULL。

```
gbase> SELECT 1 XOR NULL FROM t;
+-------------+
| 1 XOR NULL  |
+-------------+
|        NULL |
+-------------+
1 row in set

gbase> SELECT 0 XOR NULL FROM t;
+-------------+
| 0 XOR NULL  |
+-------------+
|        NULL |
+-------------+
1 row in set
```

示例 4：a XOR b 等价于(a AND (NOT b)) OR ((NOT a) AND b)。

```
gbase> SELECT 1 XOR 1 XOR 1 FROM t;
+---------------+
| 1 XOR 1 XOR 1 |
+---------------+
|             1 |
+---------------+
1 row in set
```

3.1.5 转换操作符和函数

3.1.5.1 BINARY

在字符串前使用 BINARY 操作符，可以使得参数值的比较区分大小写。

示例 1：字符串前不使用 BINARY，比较不区分大小写。

```
gbase> SELECT 'a' = 'A' FROM t;
+-----------+
| 'a' = 'A' |
+-----------+
|         1 |
+-----------+
1 row in set

gbase> SELECT 'a' = 'a ' FROM t;
+------------+
| 'a' = 'a ' |
+------------+
|          1 |
+------------+
1 row in set
```

示例 2：字符串前使用 BINARY，比较区分大小写。

```
gbase> SELECT BINARY 'a' = 'A' FROM t;
+------------------+
| BINARY 'a' = 'A' |
+------------------+
|                0 |
+------------------+
1 row in set
```

示例 3：字符串前使用 BINARY，对尾空格进行比较。

```
gbase> SELECT BINARY 'a' = 'a ' FROM t;
+-------------------+
| BINARY 'a' = 'a ' |
+-------------------+
|                 0 |
+-------------------+
1 row in set
```

3.1.5.2 CAST 和 CONVERT 函数

CAST(expr AS type), CONVERT(expr,type), CONVERT(expr USING transcoding_name)

CAST()和 CONVERT()函数可以用于将一个类型的数值转换到另一个类型。

type 可以是下列值之一:

CHAR、DATE、DATETIME、DECIMAL、TIME

CAST()和 CONVERT(...USING...)是标准的 SQL 语法。

CAST(str AS BINARY)等价于 BINARY str。

CAST(expr AS CHAR)把表达式看作是默认字符集中的字符串。

注意: 一个 CAST()到 DATE, DATETIME, 或 TIME 只是标识此列, 使其变为一个指定的数据类型, 而不是改变列的值。

CAST()的最终执行结果将会转化为正确的列类型。

示例 1: 将 NOW()转换为 DATE 类型。

```
gbase> SELECT CAST(NOW() AS DATE) FROM t;
+---------------------+
| CAST(NOW() AS DATE) |
+---------------------+
| 2013-12-17          |
+---------------------+
1 row in set
```

示例 2: 字符串和数字类型的转换是隐式操作, 用户使用时只要把字符串值当做一个数字即可。

```
gbase> SELECT 1+'1' FROM t;
+-------+
| 1+'1' |
```

```
+-------+
|     2 |
+-------+
1 row in set
```

示例 3：如果在一个字符串上下文中使用一个数字，该数字会被自动地转换为一个 BINARY 字符串。

```
gbase> SELECT CONCAT('hello you ',2) FROM t;
+-----------------------+
| CONCAT('hello you ',2) |
+-----------------------+
| hello you 2            |
+-----------------------+
1 row in set
```

示例 4：CAST(str AS BINARY)等价于 BINARY str。

```
gbase> SELECT CAST('a' AS BINARY) = 'a ' FROM t;
+---------------------------+
| CAST('a' AS BINARY) = 'a ' |
+---------------------------+
|                         0 |
+---------------------------+
1 row in set

gbase> SELECT BINARY 'a' = 'a ' FROM t;
+-------------------+
| BINARY 'a' = 'a ' |
+-------------------+
|                 0 |
+-------------------+
1 row in set
```

3.2 控制流函数

3.2.1 CASE

CASE value WHEN [compare-value] THEN result [WHEN [compare-value] THEN result ...] [ELSE result] END

逐一匹配，当满足 value=compare-value 时，返回对应的 result，如果未找到匹配项，则返回 ELSE 后的 result。如果没有 ELSE 子句，默认返回 NULL。

CASE WHEN [condition] THEN result [WHEN [condition] THEN result ...] [ELSE result] END

逐一判断，当 condition 为 TRUE 时，返回对应的 result，如果 condition 全为 FALSE,则返回 ELSE 后的 result。如果没有 ELSE 子句，默认返回 NULL。

一个 CASE 表达式的默认返回值类型是所有返回值的相容集合类型，具体情况视其所在语境而定：

- 如用在字符串语境中，则返回结果为字符串；
- 如用在数字语境中，则返回结果为十进制值的实数值或整数值。

示例 1：value=compare-value，返回对应的 result 值。

```
gbase> SELECT CASE 1 WHEN 1 THEN 'one' WHEN 2 THEN 'two' ELSE 'more' END FROM t;
+-------------------------------------------------------------+
| CASE 1 WHEN 1 THEN 'one' WHEN 2 THEN 'two' ELSE 'more' END  |
+-------------------------------------------------------------+
| one                                                         |
+-------------------------------------------------------------+
1 row in set
```

示例 2：condition 为 TRUE 时，返回对应的 result 值。

```
gbase> SELECT CASE WHEN 1>0 THEN 'true' ELSE 'false' END FROM t;
+---------------------------------------+
| CASE WHEN 1>0 THEN 'true' ELSE 'false' END  |
```

```
+-----------------------------------------+
| true                                    |
+-----------------------------------------+
1 row in set
```

示例 3：value 不等于 compare-value，返回值为 NULL。

```
gbase> SELECT CASE 'c' WHEN 'a' THEN 1 WHEN 'b' THEN 2 END FROM t;
+-----------------------------------------+
| CASE 'c' WHEN 'a' THEN 1 WHEN 'b' THEN 2 END  |
+-----------------------------------------+
|                                    NULL |
+-----------------------------------------+
1 row in set
```

3.2.2 DECODE

DECODE(value,value1,result1, value2,result2, value3,result3,... , result)

类似于 CASE value WHEN value1 THEN value1，唯一区别是如果 value 为 null 值，可以和后面的 null 值匹配。

示例 1：没有匹配的 value 值，返回值为 result。

```
gbase> SELECT DECODE(5,1,10,2,20,3,30,4,40, 50) FROM t;
+---------------------------------+
| DECODE(5,1,10,2,20,3,30,4,40, 50) |
+---------------------------------+
|                              50 |
+---------------------------------+
1 row in set
```

示例 2：value 为表达式，与 value1 匹配，返回值为 result1。

```
gbase> SELECT DECODE( (2 * 5) ,10,100,20,200,600) FROM t;
+---------------------------------+
```

```
| DECODE( (2 * 5) ,10,100,20,200,600) |
+-------------------------------------+
|                                 100 |
+-------------------------------------+
1 row in set
```

3.2.3 IF(expr1,expr2,expr3)

如果 *expr1* 为 TRUE（*expr1* <> 0 and *expr1* <> NULL），则 IF()的返回值为 *expr2*；否则返回值为 *expr3*。

IF()的返回值规则同 CASE 表达式返回值的规则。

示例 1：expre1 的值不为 TRUE，返回值为 expr3。

```
gbase> SELECT IF(1>2,2,3) FROM t;
+-------------+
| IF(1>2,2,3) |
+-------------+
|           3 |
+-------------+
1 row in set
```

示例 2：expre1 的值为 TRUE，返回值为 expr2。

```
gbase> SELECT IF(1<2,'yes','no') FROM t;
+-------------------+
| IF(1<2,'yes','no') |
+-------------------+
| yes               |
+-------------------+
1 row in set
```

示例 3：expr1 为表达式，值为 TRUE，返回值为 expr2。

```
gbase> SELECT IF(STRCMP('test','test1'),'no','yes') FROM t;
+-----------------------------------+
```

```
| IF(STRCMP('test','test1'),'no','yes') |
+---------------------------------------+
| no                                    |
+---------------------------------------+
1 row in set
```

示例 4：expr1 为表达式，值不为 TRUE，返回值为 expr3。

```
gbase> SELECT IF(1>2,NULL,'no') FROM t;
+------------------+
| IF(1>2,NULL,'no') |
+------------------+
| no               |
+------------------+
1 row in set
```

3.2.4 IFNULL(expr1,expr2)

如果 expr1 不为 NULL，则 IFNULL()的返回值为 expr1，否则其返回值为 expr2。

IFNULL()的返回值是数字或是字符串，具体情况取决于使用它的上下文环境。等价于 IF(expre1,expre1,expre2)。

示例 1：expr1 不为 NULL，返回值为 expr1。

```
gbase> SELECT IFNULL(1,0) FROM t;
+-------------+
| IFNULL(1,0) |
+-------------+
|           1 |
+-------------+
1 row in set
```

示例 2：expr1 为 NULL，返回值为 expr2。

```
gbase> SELECT IFNULL(NULL,10) FROM t;
```

```
+-----------------+
| IFNULL(NULL,10) |
+-----------------+
|              10 |
+-----------------+
1 row in set
```

3.2.5 NULLIF(expr1,expr2)

如果 expr1 = expr2 成立，返回值为 NULL，否则返回值为 expr1。

等价于 CASE WHEN expr1 = expr2 THEN NULL ELSE expr1 END。

示例 1：expr1 = expr2，返回值为 NULL。

```
gbase> SELECT NULLIF(1,1) FROM t;
+-------------+
| NULLIF(1,1) |
+-------------+
|        NULL |
+-------------+
1 row in set
```

示例 2：expr1 = expr2 不成立，返回值为 expr1。

```
gbase> SELECT NULLIF(1,2) FROM t;
+-------------+
| NULLIF(1,2) |
+-------------+
|           1 |
+-------------+
1 row in set
```

3.3 字符串函数

对于操作字符串位置的函数，第一个位置被标记为 1。

3.3.1 ASCII(str)

返回字符串 str 首字符的 ASCII 码值。

如果 str 是一个空字符串，那么返回值为 0。

如果 str 是一个 NULL，返回值为 NULL。

ASCII()只适合数值在 0 和 255 之间的字符。

示例 1：str 的值为“2”，返回“2”对应的 ASCII 码值。

```
gbase> SELECT ASCII('2') FROM t;
+------------+
| ASCII('2') |
+------------+
|         50 |
+------------+
1 row in set
```

示例 2：str 的值为“dx”，返回“d”对应的 ASCII 码值。

```
gbase> SELECT ASCII('dx') FROM t;
+-------------+
| ASCII('dx') |
+-------------+
|         100 |
+-------------+
1 row in set
```

3.3.2 BIN(N)

返回 N 的二进制形式，N 是 BIGINT 类型数字。

如果 N 是一个 NULL，返回值为 NULL。

示例 1：N 的值为“12”，返回“12”对应的二进制形式。

```
gbase> SELECT BIN(12) FROM t;
+---------+
| BIN(12) |
+---------+
| 1100    |
+---------+
1 row in set
```

3.3.3 BIT_LENGTH(str)

返回字符串 str 的比特长度，以比特进行计算。

示例 1：str 的值为“text”，返回其对应的比特长度。

```
gbase> SELECT BIT_LENGTH('text') FROM t;
+--------------------+
| BIT_LENGTH('text') |
+--------------------+
|                 32 |
+--------------------+
1 row in set
```

示例 2：当前字符集是 UTF8，str 为“南大通用”，返回其对应的比特长度。

```
gbase> SELECT BIT_LENGTH('南大通用') FROM t;
+---------------------------+
| BIT_LENGTH('南大通用')    |
+---------------------------+
|                        96 |
+---------------------------+
1 row in set

gbase> SHOW VARIABLES LIKE 'CHARACTER_SET_SERVER';
+----------------------+-------+
```

```
| Variable_name          | Value |
+------------------------+-------+
| character_set_server | utf8  |
+------------------------+-------+
1 row in set
```

3.3.4 CHAR(N,...)

N 是整数类型参数，返回 N 所代表的 ASCII 码值对应的字符组成的字符串，忽略参数列表中的 NULL 值。

示例 1：N 的值为“71,66,97,115,101”，各整数对应的 ASCII 码所代表的字符为“G”，“B”，“a”，“s”，“e”。

```
gbase> SELECT CHAR(71,66,97,115,101) FROM t;
+-----------------------+
| CHAR(71,66,97,115,101) |
+-----------------------+
| GBase                 |
+-----------------------+
1 row in set
```

示例 2：N 的值中包含 NULL，则 NULL 被忽略。

```
gbase> SELECT CHAR(77,72,NULL,'77') FROM t;
+----------------------+
| CHAR(77,72,NULL,'77') |
+----------------------+
| MHM                  |
+----------------------+
1 row in set
```

3.3.5 CHAR_LENGTH(str)

返回字符串 str 的字符长度，以字符进行计算。

示例 1：返回“text”的字符长度。

```
gbase> SELECT CHAR_LENGTH('text') FROM t;
+---------------------+
| CHAR_LENGTH('text') |
+---------------------+
|                   4 |
+---------------------+
1 row in set
```

示例 2：返回“南大通用”的字符长度。

```
gbase> SELECT CHAR_LENGTH('南大通用') FROM t;
+----------------------------+
| CHAR_LENGTH('南大通用')    |
+----------------------------+
|                          4 |
+----------------------------+
1 row in set
```

3.3.6 CHARACTER_LENGTH(str)

等价于 CHAR_LENGTH()。

3.3.7 CONCAT(str1,str2,...)

返回结果为连接参数产生的字符串。如有任何一个参数为 NULL，则返回值为 NULL。

示例 1：连接字符串“GB”，“a”，“se”。

```
gbase> SELECT CONCAT('GB', 'a', 'se') FROM t;
+------------------------+
| CONCAT('GB', 'a', 'se') |
```

```
+-----------------------+
| GBase                 |
+-----------------------+
1 row in set
```

示例 2：任何一个参数为 NULL，返回值为 NULL。

```
gbase> SELECT CONCAT('GB', NULL, 'se') FROM t;
+-------------------------+
| CONCAT('GB', NULL, 'se') |
+-------------------------+
| NULL                    |
+-------------------------+
1 row in set
```

示例 3：“'a'||'b'||'c'”等价于 CONCAT (str1,str2,str3)。

```
gbase> SELECT 'GB' || 'a' || 'se' FROM t;
+--------------------+
| 'GB' || 'a' || 'se' |
+--------------------+
| GBase              |
+--------------------+
1 row in set
```

3.3.8 CONCAT_WS(separator,str1,str2,...)

CONCAT_WS()代表 CONCAT With Separator，是 CONCAT()的特殊形式。第一个参数是其它参数的分隔符。

分隔符的位置放在要连接的两个字符串之间。分隔符可以是一个字符串，也可以是其它参数。

如果分隔符为 NULL，则结果为 NULL。

函数会忽略分隔符后面参数中的 NULL 值。

示例 1：分隔符为“,”。

```
gbase> SELECT CONCAT_WS(',','First name','Second name','Last Name') FROM t;
+-----------------------------------------------------+
| CONCAT_WS(',','First name','Second name','Last Name') |
+-----------------------------------------------------+
| First name,Second name,Last Name                     |
+-----------------------------------------------------+
1 row in set
```

示例 2：分隔符为“,”，其中一个 str 为 NULL。

```
gbase> SELECT CONCAT_WS(',','First name',NULL,'Last Name') FROM t;
+-------------------------------------------+
| CONCAT_WS(',','First name',NULL,'Last Name') |
+-------------------------------------------+
| First name,Last Name                       |
+-------------------------------------------+
1 row in set
```

3.3.9 CONV(N,from_base,to_base)

不同数字进制间的转换。将 N 由 from_base 进制转化为 to_base 进制，返回值为 to_base 进制形式的字符串。

如有任意一个参数为 NULL，则返回值为 NULL。

参数 N 为整数，或字符串。最小为 2 进制，最大为 36 进制。

如果 to_base 是一个负数，则 N 被看作一个带符号数。否则，N 被看作无符号数。

CONV(N,10,2)等价于 BIN (N)。

示例 1：将“a”由 16 进制转为 2 进制。

```
gbase> SELECT CONV('a',16,2) FROM t;
+----------------+
```

```
| CONV('a',16,2) |
+----------------+
| 1010           |
+----------------+
1 row in set
```

示例 2：将“6E”由 18 进制转为 8 进制。

```
gbase> SELECT CONV('6E',18,8) FROM t;
+-----------------+
| CONV('6E',18,8) |
+-----------------+
| 172             |
+-----------------+
1 row in set
```

示例 3：将“-17”由 10 进制转为-18 进制。

```
gbase> SELECT CONV(-17,10,-18) FROM t;
+------------------+
| CONV(-17,10,-18) |
+------------------+
| -H               |
+------------------+
1 row in set
```

示例 4：将“10+'10'+'10'+0xa”由 10 进制转为 10 进制。

```
gbase> SELECT CONV(10+'10'+'10'+0xa,10,10) FROM t;
+-----------------------------+
| CONV(10+'10'+'10'+0xa,10,10) |
+-----------------------------+
| 40                          |
+-----------------------------+
1 row in set
```

3.3.10ELT(N,str1,str2,str3,...)

若 N=1，则返回值为 str1，若 N=2，则返回值为 str2，以此类推。

若 N 小于 1 或大于参数的数目，则返回值为 NULL。

示例 1：N=1，则返回值为 str1。

```
gbase> SELECT ELT(1, 'ej', 'Heja', 'hej', 'foo') FROM t;
+----------------------------------+
| ELT(1, 'ej', 'Heja', 'hej', 'foo') |
+----------------------------------+
| ej                               |
+----------------------------------+
1 row in set
```

示例 2：N=4，则返回值为 str4。

```
gbase> SELECT ELT(4, 'ej', 'Heja', 'hej', 'foo') FROM t;
+----------------------------------+
| ELT(4, 'ej', 'Heja', 'hej', 'foo') |
+----------------------------------+
| foo                              |
+----------------------------------+
1 row in set
```

示例 3：N=9，大于参数的数目，返回值为 NULL。

```
gbase> SELECT ELT(9, 'ej', 'Heja', 'hej', 'foo') FROM t;
+----------------------------------+
| ELT(9, 'ej', 'Heja', 'hej', 'foo') |
+----------------------------------+
| NULL                             |
+----------------------------------+
1 row in set
```

3.3.11 EXPORT_SET

EXPORT_SET(bits,on,off[,separator[,number_of_bits]])

返回值为一个字符串，对参数 bits 的比特位，按从右到左(由低位比特到高位比特)的顺序进行检测，值中的每个比特位值，如果为 1，得到一个 on 字符串，如果为 0，得到一个 off 字符串，字符串被参数 separator 分开(默认为逗号“,”)。

number_of_bits 表示被检验的二进制位数(默认为 64)。

示例 1：参数 bits 的值为“5”，对应的二进制是 0101，按从右到左检测，输出为 1010，对应的 ON 和 OFF 值为“Y”，“N”，因此输出“Y,N,Y,N”。

```
gbase> SELECT EXPORT_SET(5,'Y','N',',',4) FROM t;
+----------------------------+
| EXPORT_SET(5,'Y','N',',',4) |
+----------------------------+
| Y,N,Y,N                    |
+----------------------------+
1 row in set
```

示例 2：number_of_bits 的位数大于 bits 值对应的二进制位数时，用 off 值补齐，即“0”。

```
gbase> SELECT EXPORT_SET(6,'1','0',',',10) FROM t;
+-----------------------------+
| EXPORT_SET(6,'1','0',',',10) |
+-----------------------------+
| 0,1,1,0,0,0,0,0,0,0         |
+-----------------------------+
1 row in set
```

3.3.12FIELD(str,str1,str2,str3,...)

如果 str 等于 str1 则返回 1，如果 str 等于 str2 则返回 2，依次向后进行比较。都不相等时，返回值为 0。

- 如果所有对于 FIELD()的参数均为字符串,则所有参数均按照字符串进

行比较。

- 如果所有的参数均为数字，则按照数字进行比较。
- 如果 str 为 NULL，则返回值为 0，原因是 NULL 不能同任何值进行同等比较。FIELD()是 ELT()的补数。

示例 1：FIELD（）的参数为字符串，所有参数按照字符串进行比较。

```
gbase> SELECT FIELD('ej','Hej', 'ej', 'Heja', 'hej', 'foo') FROM t;
+-------------------------------------------+
| FIELD('ej','Hej', 'ej', 'Heja', 'hej', 'foo') |
+-------------------------------------------+
|                                         2 |
+-------------------------------------------+
1 row in set
```

示例 2：FIELD（）的参数为数字，所有参数按照数字进行比较。

```
gbase> SELECT FIELD('112','12','112','123','213') FROM t;
+-----------------------------------+
| FIELD('112','12','112','123','213') |
+-----------------------------------+
|                                 2 |
+-----------------------------------+
1 row in set
```

示例 3：str 与 str1,...strn 都不相等，返回值为 0。

```
gbase> SELECT FIELD('fo','Hej', 'ej', 'Heja', 'hej', 'foo') FROM t;
+-------------------------------------------+
| FIELD('fo','Hej', 'ej', 'Heja', 'hej', 'foo') |
+-------------------------------------------+
|                                         0 |
+-------------------------------------------+
1 row in set
```

3.3.13FIND_IN_SET(str,strlist)

参数 strlist 由字符“,”分隔的多个子串组成。

如果字符串 str 在 strlist 中，则返回匹配的位置，从 1 开始。

如果字符串 str 不在 strlist 中或者 strlist 是一个空串，返回值为 0。

如果任何一个参数为 NULL，返回值为 NULL。

示例 1：字符串 str 在 strlist 中，返回其对应的位置。

```
gbase> SELECT FIND_IN_SET('b','a,b,c,d') FROM t;
+---------------------------+
| FIND_IN_SET('b','a,b,c,d') |
+---------------------------+
|                          2 |
+---------------------------+
1 row in set
```

3.3.14HEX(N_or_S)

如果 N_or_S 是一个数字，则返回它的十六进制字符串形式，在这里，N 是一个 BIGINT 数。相当于 CONV(N,10,16)。

如果 N_or_S 是一个字符串，则返回每个字符对应的十六进制形式，其中每个字符被转化为两个十六进制数字。

以 0xff 形式出现的字符串是此函数的反转操作。此时，将每两位十六进制转换为十进制后，对应其 ASCII 码输出一个字符。

示例 1：N_or_S 值为数字。

```
gbase> SELECT HEX(255) FROM t;
+----------+
| HEX(255) |
+----------+
```

```
| FF        |
+-----------+
1 row in set
```

示例 2：N_or_S 值为字符串。

```
gbase> SELECT HEX('abc') FROM t;
+-------------+
| HEX('abc') |
+-------------+
| 616263      |
+-------------+
1 row in set
```

示例 3：以 0xff 形式出现的字符串是 HEX(N_or_S)函数的反转操作。

```
gbase> SELECT 0x616263 FROM t;
+-----------+
| 0x616263 |
+-----------+
| abc       |
+-----------+
1 row in set
```

3.3.15 INSERT(str,pos,len,newstr)

在字符串 str 中，从 pos 位置开始，选取 len 个字符长度的子串替换为字符串 newstr。

如果 pos 值不在长度范围之内，则返回原来的字符串。

如果 len 值不在字符串剩余长度范围之内，则替换从 pos 位置开始的其余字符串。

无论哪一个参数是 NULL，则返回 NULL。

示例 1：从字符串“Quadratic”的第 3 个位置开始，将 4 个字符“drat”

替换为“What”。

```
gbase> SELECT INSERT('Quadratic', 3, 4, 'What') FROM t;
+----------------------------------+
| INSERT('Quadratic', 3, 4, 'What') |
+----------------------------------+
| QuWhattic                        |
+----------------------------------+
1 row in set
```

示例 2：pos 的值不在长度范围之内，返回原字符串。

```
gbase> SELECT INSERT('Quadratic', -1, 4, 'What') FROM t;
+----------------------------------+
| INSERT('Quadratic', -1, 4, 'What') |
+----------------------------------+
| Quadratic                        |
+----------------------------------+
1 row in set
```

示例 3：len 不在字符串剩余长度范围之内，用“What”替换从第 3 个位置开始的其余字符串。

```
gbase> SELECT INSERT('Quadratic', 3, 100, 'What') FROM t;
+-----------------------------------+
| INSERT('Quadratic', 3, 100, 'What') |
+-----------------------------------+
| QuWhat                            |
+-----------------------------------+
1 row in set
```

3.3.16 INSTR()

INSTR(str,substr)

INSTR(str,substr,start_position,nth_appearance)

第一种语法功能：

返回子串 substr 在字符串 str（左端开始）第一次出现的位置。

注意，subtr 在 sub 中的位置，以 1 开始计数。

第二种语法相对于第一种，新增如下功能：

1）查找第 N 个匹配字符串的功能。

2）从母串的第 M 个字符开始查找匹配字符串的功能。

3）支持第三个参数 start_position 为负数的情况，即从母串右端第|start_position|个位置反向查找子串的功能

参数说明：

str：字符串母串；

substr：子串；

start_position：表示从字符串（左端）第几个字符开始匹配，如是负数，则从右端往前查找子串。可选参数，默认为 1；

nth_appearance：从 start_position 开始向字符串尾方向查找第几个匹配字符串。可选参数，默认为 1；

示例 1：返回“bar”在“foobarbar”中第一次出现的位置。

```
gbase> SELECT INSTR('foobarbar', 'bar') FROM t;
+--------------------------+
| INSTR('foobarbar', 'bar') |
+--------------------------+
|                         4 |
+--------------------------+
1 row in set
```

示例 2：“foobar”不在“xbar”中。

```
gbase> SELECT INSTR('xbar', 'foobar') FROM t;
```

```
+-----------------------+
| INSTR('xbar', 'foobar') |
+-----------------------+
|                     0 |
+-----------------------+
1 row in set
```

示例 3：如有任一参数是一个二进制字符串，它是字母大小写敏感的。

```
gbase> SELECT INSTR('foobarbar', BINARY 'Bar') FROM t;
+---------------------------------+
| INSTR('foobarbar', BINARY 'Bar') |
+---------------------------------+
|                               0 |
+---------------------------------+
1 row in set
```

示例 4:从字符串“welcometobeijing”第三个字符开始匹配,查找 “bei”第一次出现的位置。

```
gbase> select INSTR ('welcometobeijing','bei',3);
+---------------------------------+
| instr('welcometobeijing','bei',3) |
+---------------------------------+
|                              10 |
+---------------------------------+
1 row in set
```

示例 5：从字符串 “112111222222333333” 第一个字符开始匹配，查找第二个 “11” 出现的位置。

```
gbase> select INSTR ('112111222222333333','11',1,2);
+-------------------------------------+
| instr('112111222222333333','11',1,2) |
+-------------------------------------+
|                                   4 |
+-------------------------------------+
1 row in set
```

示例 6：从字符串 “welcometochina” 倒数第一个字符开始匹配，查找第

一个“e”出现的位置。

```
gbase> select INSTR ('welcometochina','e',-1,1);
+---------------------------------+
| instr('welcometochina','e',-1,1) |
+---------------------------------+
|                               7 |
+---------------------------------+
1 row in set
```

3.3.17LCASE(str)

依照当前字符集设置映射，将字符串 str 中的所有字符改变为小写形式。

示例 1：将“QUADRATICALLY”转为小写形式。

```
gbase> SELECT LCASE('QUADRATICALLY') FROM t;
+------------------------+
| LCASE('QUADRATICALLY') |
+------------------------+
| quadratically          |
+------------------------+
1 row in set
```

3.3.18LEFT(str,len)

返回字符串 str 中最左边的 len 个字符。

示例 1：返回“foobarbar”左边五个字符。

```
gbase> SELECT LEFT('foobarbar', 5) FROM t;
+----------------------+
| LEFT('foobarbar', 5) |
+----------------------+
| fooba                |
+----------------------+
1 row in set
```

3.3.19 LENGTH(str)

返回字符串 str 的长度，以字节进行计算。

示例 1：返回“text”的字节长度。

```
gbase> SELECT LENGTH('text') FROM t;
+----------------+
| LENGTH('text') |
+----------------+
|              4 |
+----------------+
1 row in set
```

示例 2：返回“南大通用”的字节长度。

```
gbase> SELECT LENGTH('南大通用') FROM t;
+-----------------------+
| LENGTH('南大通用')    |
+-----------------------+
|                    12 |
+-----------------------+
1 row in set
```

3.3.20 LOCATE()

LOCATE(substr,str)，LOCATE(substr,str,pos)

第一种语法返回子串 substr 在字符串 str 中第一次出现的位置。

第二种语法返回子串 substr 在字符串 str 中的第 pos 位置后第一次出现的位置。

如果 substr 不在 str 中，则返回 0。

注意，substr 在 sub 中的位置，以 1 开始计数。

LOCATE(substr,str)与 INSTR(str,substr)相似，只是参数的位置被颠倒。

示例 1：返回“bar”在“foobarbar”中第一次出现的位置。

```
gbase> SELECT LOCATE('bar', 'foobarbar') FROM t;
+----------------------------+
| LOCATE('bar', 'foobarbar') |
+----------------------------+
|                          4 |
+----------------------------+
1 row in set
```

示例 2：“xbar”不在“foobar”中，返回值为 0。

```
gbase> SELECT LOCATE('xbar', 'foobar') FROM t;
+--------------------------+
| LOCATE('xbar', 'foobar') |
+--------------------------+
|                        0 |
+--------------------------+
1 row in set
```

示例 3：返回“bar”在“foobarbar”中的第 5 位后，第一次出现的位置。

```
gbase> SELECT LOCATE('bar', 'foobarbar',5) FROM t;
+-----------------------------+
| LOCATE('bar', 'foobarbar',5) |
+-----------------------------+
|                           7 |
+-----------------------------+
1 row in set
```

示例 4：如有任一参数是一个二进制字符串，它是字母大小写敏感的。

```
gbase> SELECT LOCATE(BINARY'bAr', 'foobarbar',5) FROM t;
+-----------------------------------+
| LOCATE(BINARY'bAr', 'foobarbar',5) |
```

```
+----------------------------------+
|                                0 |
+----------------------------------+
1 row in set
```

3.3.21 LOWER(str)

依照当前字符集设置映射，将字符串 str 中的所有字符改变为小写形式。

示例 1：LOWER(str)等价于 LCASE()。

```
gbase> SELECT LOWER('QUADRATICALLY'),LCASE('QUADRATICALLY') FROM t;
+----------------------+----------------------+
| LOWER('QUADRATICALLY') | LCASE('QUADRATICALLY') |
+----------------------+----------------------+
| quadratically          | quadratically          |
+----------------------+----------------------+
1 row in set
```

3.3.22 LPAD(str,len,padstr)

用字符串 padstr 对 str 进行左边填补，直至它的长度达到 len 个字符长度，然后返回 str。

如果 str 的长度长于 len，那么它将被截除到 len 个字符。

示例 1：将“??”补到“hi”左侧，总长度为 4 位。

```
gbase> SELECT LPAD ('hi',4,'??') FROM t;
+--------------------+
| LPAD ('hi',4,'??') |
+--------------------+
| ??hi               |
+--------------------+
1 row in set
```

示例 2: “hi”的长度大于 1，则“hi”将被截除到 1 个字符。

```
gbase> SELECT LPAD('hi',1,'??') FROM t;
+------------------+
| LPAD('hi',1,'??') |
+------------------+
| h                |
+------------------+
1 row in set
```

3.3.23LTRIM(str)

移除 str 最左边的连续多个空格。

示例 1: 移除“ barbar”左边两个空格。

```
gbase> SELECT LTRIM('  barbar') FROM t;
+------------------+
| LTRIM('  barbar') |
+------------------+
| barbar           |
+------------------+
1 row in set
```

3.3.24MAKE_SET(bits,str1,str2,...)

返回一个设定值（一个包含被“,”分开的字符串），这个值由 bits 组中具有相应的比特的字符串组成。

bits 中的比特值按照从右到左的顺序接受检验(由低位比特到高位比特)。

str1 对应第一位比特值为 1，str2 对应第二位比特值为 1，以此类推。

str1，str2，...中的 NULL 值不会被添加到结果中。

示例 1：将 1 的比特值按从右到左进行校验，“a”对应第一位比特值为 1。

```
gbase> SELECT MAKE_SET(1,'a','b','c') FROM t;
+------------------------+
| MAKE_SET(1,'a','b','c') |
+------------------------+
| a                      |
+------------------------+
1 row in set
```

示例 2：将“1|4”的比特值按从右到左进行校验，即第一位和第三位的比特值为 1，返回对应的字符串。

```
gbase> SELECT MAKE_SET(1 | 4,'hello','nice','world') FROM t;
+-------------------------------------+
| MAKE_SET(1 | 4,'hello','nice','world') |
+-------------------------------------+
| hello,world                         |
+-------------------------------------+
1 row in set
```

示例 3：字符串列中的 NULL 不被添加到结果中。

```
gbase> SELECT MAKE_SET(1 | 4,'hello','nice',NULL,'world') FROM t;
+-----------------------------------------+
| MAKE_SET(1 | 4,'hello','nice',NULL,'world') |
+-----------------------------------------+
| hello                                   |
+-----------------------------------------+
1 row in set
```

示例 4：0 没有对应的 str。

```
gbase> SELECT MAKE_SET(0,'a','b','c') FROM t;
+------------------------+
| MAKE_SET(0,'a','b','c') |
+------------------------+
|                        |
```

```
+------------------------+
1 row in set
```

3.3.25MID(str,pos,len)

返回字符串 str 中，从 pos 位置开始，长度为 len 的子串。

MID(str,pos,len)等价于 SUBSTRING(str,pos,len)。

示例 1：SUBSTRING()与 MID()等价。

```
gbase> SELECT SUBSTRING('Quadratically',5,6),MID('Quadratically',5,6) FROM
t;
+------------------------------+------------------------+
| SUBSTRING('Quadratically',5,6) | MID('Quadratically',5,6) |
+------------------------------+------------------------+
| ratica                       | ratica                 |
+------------------------------+------------------------+
1 row in set
```

3.3.26NVL(string1,replace_with)

如果 string1 为 NULL，则 NVL()函数返回 replace_with 的值，否则返回 string1 的值。

示例 1：address 列的值为 NULL，返回“UNKOWN”，否则返回 address 的值。

```
gbase> DROP TABLE IF EXISTS t_user;
Query OK, 0 rows affected

gbase> CREATE TABLE t_user (id int ,name varchar(10),address varchar(200));
Query OK, 0 rows affected

gbase> INSERT INTO t_user VALUES (1,'Tom','East
Street'),(2,'Mike',NULL),(3,'Rose','TANGREN ROAD'),(4,'White',NULL);
Query OK, 4 rows affected
```

```
Records: 4  Duplicates: 0  Warnings: 0
```

```
gbase> SELECT id,name,NVL(address,'UNKOWN') FROM t_user;
+------+-------+----------------------+
| id   | name  | NVL(address,'UNKOWN') |
+------+-------+----------------------+
|    1 | Tom   | East Street          |
|    2 | Mike  | UNKOWN               |
|    3 | Rose  | TANGREN ROAD         |
|    4 | White | UNKOWN               |
+------+-------+----------------------+
4 rows in set
```

3.3.27 OCT(N)

返回一个 N 的八进制值的字符串表示。

此处，N 是一个 BIGINT 类型的数字。

如果 N 是一个 NULL，返回值也是 NULL。

OCT(N)等价于 CONV(N,10,8)。

示例 1：返回 12 的八进制值。

```
gbase> SELECT OCT(12) FROM t;
+---------+
| OCT(12) |
+---------+
| 14      |
+---------+
1 row in set
```

示例 2：N 是 NULL，返回值为 NULL。

```
gbase> SELECT OCT(NULL) FROM t;
+-----------+
| OCT(NULL) |
```

```
+-----------+
| NULL      |
+-----------+
1 row in set
```

3.3.28 ORD(str)

如果字符串 str 的最左边的字符是一个多字节的字符，返回该字符的代码，代码的计算通过使用以下公式计算其组成字节的数值而得出。

(1st byte code)

+ (2nd byte code × 256)

+ (3rd byte code × 256^2) ...

如果最左边的字符不是一个多字节字符，返回值与 ASCII()函数的返回值相同。

示例 1：str 为“2”，返回 2 对应的 ASCII 码值。

```
gbase> SELECT ORD('2') FROM t;
+----------+
| ORD('2') |
+----------+
|       50 |
+----------+
1 row in set
```

示例 2：str 为“南大通用”，返回“南”对应的代码。

```
gbase> SELECT ORD('南大通用') FROM t;
+--------------------+
| ORD('南大通用')    |
+--------------------+
|           15043991 |
+--------------------+
```

```
1 row in set
```

3.3.29 REPEAT(str,count)

返回一个重复了 count 次的字符串 str 组成的字符串。

如果 count<=0，返回一个空字符串。

如果 str 或 count 是 NULL，返回值为 NULL。

示例 1：返回将“GBase”重复 3 次后的字符串。

```
gbase> SELECT REPEAT('GBase', 3) FROM t;
+--------------------+
| REPEAT('GBase', 3) |
+--------------------+
| GBaseGBaseGBase    |
+--------------------+
1 row in set
```

3.3.30 REPLACE(str, from_str,to_str)

返回字符串 str 中将所有出现的 from_str 替换为 to_str 的字符串。

示例 1：将“www.generaldata.com.cn”中所有出现的“w”替换为“Ww”。

```
gbase> SELECT REPLACE('www.generaldata.com.cn', 'w', 'Ww') FROM t;
+-------------------------------------------+
| REPLACE('www.generaldata.com.cn', 'w', 'Ww') |
+-------------------------------------------+
| WwWwWw.generaldata.com.cn                 |
+-------------------------------------------+
1 row in set
```

3.3.31 REVERSE(str)

返回字符顺序和 str 相反的字符串。

示例 1：将“abc”按从右到左的顺序输出。

```
gbase> SELECT REVERSE('abc') FROM t;
+----------------+
| REVERSE('abc') |
+----------------+
| cba            |
+----------------+
1 row in set
```

3.3.32 RIGHT(str,len)

返回字符串 str 中最右边的 len 个字符。

示例 1：返回“foobarbar”最右边的 4 个字符。

```
gbase> SELECT RIGHT('foobarbar', 4) FROM t;
+----------------------+
| RIGHT('foobarbar', 4) |
+----------------------+
| rbar                 |
+----------------------+
1 row in set
```

3.3.33 RPAD(str,len,padstr)

用字符串 padstr 对 str 进行右边填补，直至它的长度达到 len 个字符长度，然后返回 str。

如果 str 的长度长于 len，那么它将被截取到 len 个字符。

示例 1：在“hi”的右边补充“?”，直到长度为 5 位。

```
gbase> SELECT RPAD('hi',5,'?') FROM t;
+-----------------+
| RPAD('hi',5,'?') |
+-----------------+
| hi???           |
+-----------------+
1 row in set
```

示例 2：“hiacd”的长度大于 2，则截取到 2 个字符。

```
gbase> SELECT RPAD('hiacd',2,'?') FROM t;
+-----------------+
| RPAD('hi',2,'?') |
+-----------------+
| hi              |
+-----------------+
1 row in set
```

3.3.34RTRIM(str)

返回移除了 str 最右边多余空格后剩下的字符串。

示例 1：移除“barbar　　”最右边的多余空格。

```
gbase> SELECT RTRIM('barbar   ') FROM t;
+-------------------+
| RTRIM('barbar   ') |
+-------------------+
| barbar            |
+-------------------+
1 row in set
```

3.3.35SUBSTRING()

SUBSTRING(str,pos)

SUBSTRING(str,pos,len)

没有 len 参数的 SUBSTRING()函数从字符串 str 的 pos 位置起返回子串。

带有 len 参数的 SUBSTRING()函数从字符串 str 的 pos 位置起返回 len 个字符的子串。

pos 可以是负值。在这种情况下，子串的起始位置是从字符串的尾部 pos 位置。

如果 len 为小于 1 的值，返回结果始终为空串。

示例 1：返回“Quadratically”从第 5 位开始的子串。

```
gbase> SELECT SUBSTRING('Quadratically',5) FROM t;
+-----------------------------+
| SUBSTRING('Quadratically',5) |
+-----------------------------+
| ratically                   |
+-----------------------------+
1 row in set
```

示例 2：返回“Quadratically”从第 5 位开始的 6 个字符。

```
gbase> SELECT SUBSTRING('Quadratically',5,6) FROM t;
+-------------------------------+
| SUBSTRING('Quadratically',5,6) |
+-------------------------------+
| ratica                        |
+-------------------------------+
1 row in set
```

示例 3：pos 为“-3”，则返回的子串是“Sakila”尾部的 3 个字符。

```
gbase> SELECT SUBSTRING('Sakila', -3) FROM t;
+-------------------------+
| SUBSTRING('Sakila', -3) |
+-------------------------+
```

```
| ila                    |
+------------------------+
1 row in set
```

示例 4：pos 为“-5”，len 为“3”，则返回的子串为“Sakila”从第 2 位开始的 3 个字符。

```
gbase> SELECT SUBSTRING('Sakila', -5, 3) FROM t;
+---------------------------+
| SUBSTRING('Sakila', -5, 3) |
+---------------------------+
| aki                        |
+---------------------------+
1 row in set
```

3.3.36 SUBSTRING_INDEX(str,delim,count)

返回字符串 str 中在第 count 个分隔符 delim 之前的子串。

如果 count 是一个正数，返回从最后的（从左边开始计数）分隔符到左边所有字符。

如果 count 是负数，返回从最后的（从右边开始计数）分隔符到右边所有字符。

示例 1：count 是正数，返回从分隔符“.”到左边的所有字符。

```
gbase> SELECT SUBSTRING_INDEX('www.generaldata.com.cn', '.', 2) FROM t;
+-----------------------------------------------+
| SUBSTRING_INDEX('www.generaldata.com.cn', '.', 2) |
+-----------------------------------------------+
| www.generaldata                               |
+-----------------------------------------------+
1 row in set
```

示例 2：count 是负数，返回从分隔符“.”到右边的所有字符。

```
gbase> SELECT SUBSTRING_INDEX('www.generaldata.com.cn', '.', -3) FROM t;
+-----------------------------------------------+
| SUBSTRING_INDEX('www.generaldata.com.cn', '.', -3) |
+-----------------------------------------------+
| generaldata.com.cn                            |
+-----------------------------------------------+
1 row in set
```

3.3.37 TO_CHAR(number,[FORMAT])

将参数 number 转换为字符串，并进行格式化输出。

如果 number 的位数大于格式化参数 FORMAT 的参数，结果将以“#”显示。

格式化参数及含义如下表所示。

格式化参数	含　义
,	一般作为分组符号使用，将 number 参数格式化为数位格式字符串输出，例如千位一分组，也可以按百位、十位一分组。通常与 0、9、“.”配合使用。 示例：99,999。
.	将 number 参数，格式化为小数形式的字符串输出。只能出现一次。通常与 0、9、“,”配合使用。 示例：999.99。
$	转换为美元货币含义的字符串，只能出现在最前或最后。 示例：$999。
0	占位符，格式化 number，如果参数 number 的位数少于格式化的位数，则显示 0 补足位。注意：0 的优先级高于 9。 示例：000。
9	占位符，格式化number，一旦参数number的位数，少于格式化的位数，则用空格补足位。 示例：999。
B、b	如果 number 的值为 0，则替换为空格，可以出现在任意位置。 示例：B9.99
EEEE、eeee	按照科学计数法输出。 示例：9.99EEEE。

格式化参数	含　义
FM、fm	删除数字开头和结尾处的空格。 示例：FM909.9。
TME	按照科学计数法返回 number。
X、x	转换为 16 进制。每个 X 代表 16 进制的一位。 例如：XX，代表两位 16 进制数。 如果 number，转换成 16 进制数大于 X 的个数，则输出“#”。 注意：数值必须是大于等于 0 的整数。前面只能和 0 或者 FM 组合使用。

示例 1：以百位作为分组。

```
gbase> SELECT TO_CHAR(987654321,'999,999,999') FROM t;
+--------------------------------+
| TO_CHAR(987654321,'999,999,999') |
+--------------------------------+
|  987,654,321                   |
+--------------------------------+
1 row in set
```

示例 2：用空格位补足数值位。

```
gbase> SELECT TO_CHAR(54321,'999,999,999') FROM t;
+-----------------------------+
| TO_CHAR(54321,'999,999,999') |
+-----------------------------+
|       54,321                |
+-----------------------------+
1 row in set
```

示例 3：因为 0 的优先级高于 9，所以十万位和百万位均以 0 显示，千万位和亿位以空格显示。

```
gbase> SELECT TO_CHAR(54321,'990,999,999') FROM t;
+-----------------------------+
| TO_CHAR(54321,'990,999,999') |
+-----------------------------+
|    0,054,321                |
```

```
+----------------------------+
1 row in set

gbase> SELECT TO_CHAR(-54321,'990,999,999') FROM t;
+-----------------------------+
| TO_CHAR(-54321,'990,999,999') |
+-----------------------------+
|   -0,054,321                |
+-----------------------------+
1 row in set
```

示例 4：小数格式化输出。

```
gbase> SELECT TO_CHAR(12.97,'099.99') FROM t;
+-------------------------+
| TO_CHAR(12.97,'099.99') |
+-------------------------+
|  012.97                 |
+-------------------------+
1 row in set
```

示例 5：小数格式化输出，小数位补足三位。

因为 0 的优先级高于 9，所以不论是 090 还是 099，都按照 3 位小数格式化输出，补足位用 0 补足。

```
gbase> SELECT TO_CHAR(-12.97,'099.090') FROM t;
+---------------------------+
| TO_CHAR(-12.97,'099.090') |
+---------------------------+
| -012.970                  |
+---------------------------+
1 row in set

gbase> SELECT TO_CHAR(12.97,'099.099') FROM t;
+---------------------------+
| TO_CHAR(12.97,'099.099') |
+---------------------------+
```

```
|  012.970                |
+-------------------------+
1 row in set
```

示例 6：按美元含义格式化输出，$格式化符号只能出现在最前或最后位置。

```
gbase> SELECT TO_CHAR(84.77,'$0099.99') FROM t;
+--------------------------+
| TO_CHAR(84.77,'$0099.99') |
+--------------------------+
|  $0084.77                |
+--------------------------+
1 row in set

gbase> SELECT TO_CHAR(84.77,'0099.99$') FROM t;
+--------------------------+
| TO_CHAR(84.77,'0099.99$') |
+--------------------------+
|  $0084.77                |
+--------------------------+
1 row in set
```

示例 7：整数部分为 0，返回空格。

```
gbase> SELECT TO_CHAR(0,'B00') FROM t;
+-----------------+
| TO_CHAR(0,'B00') |
+-----------------+
|                 |
+-----------------+
1 row in set

gbase> SELECT HEX(TO_CHAR(0,'B00')) FROM t;
+---------------------+
| HEX(TO_CHAR(0,'B00')) |
+---------------------+
| 202020              |
+---------------------+
```

```
1 row in set
```

整数部分为 1 时，返回 01。

```
gbase> SELECT TO_CHAR(1,'B00') FROM t;
+------------------+
| TO_CHAR(1,'B00') |
+------------------+
|  01              |
+------------------+
1 row in set
```

整数部分为 11 时，返回 11。

```
gbase> SELECT TO_CHAR(11,'B00') FROM t;
+-------------------+
| TO_CHAR(11,'B00') |
+-------------------+
|  11               |
+-------------------+
1 row in set
```

示例 8：FORMAT 的值为“9.9EEEE”，由于是科学计算方法,所以小数位前面加一个 9 或者 0 即可,多个是没有意义的。

```
gbase> SELECT TO_CHAR(2008032001,'9.9EEEE') FROM t;
+-------------------------------+
| TO_CHAR(2008032001,'9.9EEEE') |
+-------------------------------+
|   2.0E+09                     |
+-------------------------------+
1 row in set

gbase> SELECT TO_CHAR(2008032001,'c9.99EEEE') FROM t;
+--------------------------------+
| TO_CHAR(2008032001,'c9.99EEEE') |
+--------------------------------+
|   $2.01E+09                    |
```

```
+-------------------------------+
1 row in set
```

示例 9：FORMAT 的值为“FM90.9”，删除“10.3”开头和结尾处的空格。

```
gbase> SELECT TO_CHAR(10.3,'FM90.9') FROM t;
+------------------------+
| TO_CHAR(-10.3,'FM90.9') |
+------------------------+
| 10.3                   |
+------------------------+
1 row in set

gbase> SELECT TO_CHAR(10.3,'90.9') FROM t;
+----------------------+
| TO_CHAR(10.3,'90.9') |
+----------------------+
|  10.3                |
+----------------------+
1 row in set
```

示例 10：FORMAT 的值为“TME”。

```
gbase> SELECT TO_CHAR(11,'TME') AS f_SHOW FROM t;
+---------+
| f_SHOW  |
+---------+
| 1.1E+01 |
+---------+
1 row in set
```

示例 11：FORMAT 的值为“X”，返回 number 的十六进制形式。

```
gbase> SELECT TO_CHAR(11,'X') FROM t;
+-----------------+
| TO_CHAR(11,'X') |
+-----------------+
|  B              |
```

```
+-----------------+
1 row in set
```

```
gbase> SELECT TO_CHAR(16,'XX') FROM t;
+------------------+
| TO_CHAR(16,'XX') |
+------------------+
|  10              |
+------------------+
1 row in set
```

number 转换成 16 进制数大于 X 的个数，则输出“#”。

```
gbase> SELECT TO_CHAR(16,'X') FROM t;
+-----------------+
| TO_CHAR(16,'X') |
+-----------------+
| ##              |
+-----------------+
1 row in set
```

3.3.38 TO_CHAR(datetime,[FORMAT])

将参数 datetime 转换为字符串，并进行格式化输出。

格式化参数及含义如下表所示。

格式化参数	含　义
, . ; :	除了左面标准的几个，还允许用文字作为分割符号。例如 年月日　日期分隔符。用于格式化输出日期。
AD	即拉丁文 Anno Domini 的简写，表示公元，会根据 nls 的不同转换为公元或者 AD 等。 如果是公元后的日期，显示 AD。 如果是公元前的日期，显示 BC。
AM	上午的简写，同 PM（下午），中文环境输出为上午。 如果是上午，返回 AM。

格式化参数	含　义
	如果是下午，返回 PM。
BC	即拉丁文 Before Christ 的简写，表示公元前，会根据 nls 的不同转换为公元或者 BC 等。
CC	返回世纪，以阿拉伯数字表示。
D	一周之中的第几天，返回的是序号（1～7）。
DAY	返回日期中的 DAY 部分。返回的是英文全拼形式，首字母大写。
DD	同 DAY，但是返回的是数字形式（01～31）。
DDD	日期中的日是一年当中的第几天,返回的是序号 001～366。
DY	同 DAY,但是返回的是英文形式，返回前三个字母。首字母大写。
FF[n]	就是毫秒,如果不加数字就是用默认的精度，默认 6 位精度。1 ≤ n ≤ 9。 只能用于 timestamp 类型的。
FM	删除日期开头和结尾处的空格。 示例：FM Month
FX	固定模式全局选项。 示例：FX MONTH DD DAY
HH[12 | 24]	表示小时，默认 12 小时制。 HH12，12 小时制。返回（01～12）。 HH24，24 小时制。返回（00～23）。
IW	ISO 标准的一年中的第几周（1～52,或者 1～53）。
MI	返回分钟数（00～59）。
MM	返回月份（01～12）。
MON	返回月份，英文简写形式，三个英文字母，首字母大写。
MONTH	返回月份，返回的是英文全拼。首字母大写。
PM	下午的简写,中文环境输出为下午。
Q	返回季度，取值为 1～4。
RM	用罗马数字表示的月份。罗马数字全部大写。
RR 或 RRRR	返回 2 位或者 4 位年。
SCC	返回数字形式表示的世纪。
SS	返回秒（0～59）。
SSSSS	一天从午夜开始的累积秒数（0～86399）。
TS	返回带有 AM 或者 PM 的时分秒形式的时间。
W	一个月中的第几周，其算法局限在 datetime 参数所属于的

格式化参数	含　义
	月份之内而已。
WW	同 IW。

示例 1：将 NOW()转换为 FORMAT 中对应的日期格式。

```
gbase> SELECT TO_CHAR(NOW(),'YYYY/MM/DD') FROM t;
+----------------------------+
| TO_CHAR(NOW(),'YYYY/MM/DD') |
+----------------------------+
| 2013/12/17                 |
+----------------------------+
1 row in set

gbase> SELECT TO_CHAR(NOW(),'YYYY-MM-DD') FROM t;
+----------------------------+
| TO_CHAR(NOW(),'YYYY-MM-DD') |
+----------------------------+
| 2013-12-17                 |
+----------------------------+
1 row in set

gbase> SELECT TO_CHAR(NOW(),'YYYY,MM,DD') FROM t;
+----------------------------+
| TO_CHAR(NOW(),'YYYY,MM,DD') |
+----------------------------+
| 2013,12,17                 |
+----------------------------+
1 row in set

gbase> SELECT TO_CHAR(CURDATE(),'YYYY;MM;DD') FROM t;
+-------------------------------+
| TO_CHAR(CURDATE(),'YYYY;MM;DD') |
+-------------------------------+
| 2013;12;17                    |
+-------------------------------+
1 row in set
```

```
gbase> SELECT TO_CHAR(NOW(),'YYYY"年"MM"月"DD"日"') FROM t;
+--------------------------------------+
| TO_CHAR(NOW(),'YYYY"年"MM"月"DD"日"') |
+--------------------------------------+
| 2013年12月17日                       |
+--------------------------------------+
1 row in set
```

示例 2：将 CURDATE()转换为 FORMAT 中对应的日期格式。

```
gbase> SELECT TO_CHAR(CURDATE(),'AD  YYYY-MM-DD') FROM t;
+-----------------------------------+
| TO_CHAR(CURDATE(),'AD  YYYY-MM-DD') |
+-----------------------------------+
| AD  2014-01-03                    |
+-----------------------------------+
1 row in set
```

示例 3：比较 NOW()和将 NOW()转换为“AM HH12:MM:SS”后的格式。

```
gbase> SELECT NOW(),TO_CHAR(NOW(),'AM HH12:MM:SS') FROM t;
+---------------------+-------------------------------+
| NOW()               | TO_CHAR(NOW(),'AM HH12:MM:SS') |
+---------------------+-------------------------------+
| 2013-12-17 16:45:18 | PM 04:12:18                   |
+---------------------+-------------------------------+
1 row in set

gbase> SELECT NOW(),TO_CHAR(TIMESTAMPADD(HOUR,8,NOW()),'AM HH12:MM:SS') AS
f_FormatShow FROM t;
+---------------------+-------------+
| NOW()               | f_FormatShow |
+---------------------+-------------+
| 2013-12-17 16:45:24 | AM 12:12:24  |
+---------------------+-------------+
1 row in set
```

示例 4：返回 CURDATE()的世纪数。

```
gbase> SELECT TO_CHAR(CURDATE(),'CC') FROM t;
+-------------------------+
| TO_CHAR(CURDATE(),'CC') |
+-------------------------+
| 21                      |
+-------------------------+
1 row in set
```

示例 5：系统默认周日为每周第一天，“2013-11-11”是周五。

FORMAT 的值为“D”，返回值为 6。

```
gbase> SELECT CURDATE(),TO_CHAR(CURDATE(),'D') FROM t;
+------------+-----------------------+
| CURDATE()  | TO_CHAR(CURDATE(),'D') |
+------------+-----------------------+
| 2013-12-17 | 3                     |
+------------+-----------------------+
1 row in set
```

FORMAT 的值为“DAY”，返回周五的英文全拼形式，首字母大写。

```
gbase> SELECT CURDATE(),TO_CHAR(CURDATE(),'DAY') FROM t;
+------------+------------------------+
| CURDATE()  | TO_CHAR(CURDATE(),'DAY') |
+------------+------------------------+
| 2013-12-17 | Tuesday                |
+------------+------------------------+
1 row in set
```

FORMAT 的值为“DD”，返回值为 11。

```
gbase> SELECT CURDATE(),TO_CHAR(CURDATE(),'DD') FROM t;
+------------+------------------------+
| CURDATE()  | TO_CHAR(CURDATE(),'DD') |
+------------+------------------------+
| 2013-12-17 | 17                     |
```

```
+------------+------------------------+
1 row in set
```

FORMAT 的值为“DDD”，返回“2013-10-11”是 2013 年的第几天。

```
gbase> SELECT CURDATE(),TO_CHAR(CURDATE(),'DDD') FROM t;
+------------+--------------------------+
| CURDATE()  | TO_CHAR(CURDATE(),'DDD') |
+------------+--------------------------+
| 2013-12-17 | 351                      |
+------------+--------------------------+
1 row in set
```

FORMAT 的值为“DAY”，返回周五的英文形式的前三个字母，首字母大写。

```
gbase> SELECT CURDATE(),TO_CHAR(CURDATE(),'DY') FROM t;
+------------+-------------------------+
| CURDATE()  | TO_CHAR(CURDATE(),'DY') |
+------------+-------------------------+
| 2013-12-17 | Tue                     |
+------------+-------------------------+
1 row in set
```

示例 6：查询当前时间的毫秒，默认为 6 位。

```
gbase> SELECT CURRENT_TIMESTAMP(),TO_CHAR(CURRENT_TIMESTAMP(),'FF') FROM t;
+--------------------+---------------------------------+
| CURRENT_TIMESTAMP() | TO_CHAR(CURRENT_TIMESTAMP(),'FF') |
+--------------------+---------------------------------+
| 2013-12-17 16:46:21 | 000000                          |
+--------------------+---------------------------------+
1 row in set

gbase> SELECT CURRENT_TIMESTAMP(),TO_CHAR(CURRENT_TIMESTAMP(),'FF9') FROM t;
+--------------------+----------------------------------+
| CURRENT_TIMESTAMP() | TO_CHAR(CURRENT_TIMESTAMP(),'FF9') |
+--------------------+----------------------------------+
| 2013-12-17 16:46:28 | 000000000                        |
```

```
+--------------------+---------------------------------+
1 row in set
```

示例 7：FORMAT 为“FX”或“FM”。

```
gbase> SELECT CURRENT_TIMESTAMP(),TO_CHAR(CURRENT_TIMESTAMP(),'FX
YYYY-MM-DD') FROM t;
+--------------------+-------------------------------------------+
| CURRENT_TIMESTAMP() | TO_CHAR(CURRENT_TIMESTAMP(),'FX YYYY-MM-DD') |
+--------------------+-------------------------------------------+
| 2013-12-17 16:46:35 |  2013-12-17                               |
+--------------------+-------------------------------------------+
1 row in set

gbase> SELECT CURRENT_TIMESTAMP(),TO_CHAR(CURRENT_TIMESTAMP(),'FM
YYYY-MM-DD') FROM t;
+--------------------+-------------------------------------------+
| CURRENT_TIMESTAMP() | TO_CHAR(CURRENT_TIMESTAMP(),'FM YYYY-MM-DD') |
+--------------------+-------------------------------------------+
| 2013-12-17 16:46:40 |  2013-12-17                               |
+--------------------+-------------------------------------------+
1 row in set
```

示例 8：FORMAT 为“HH”，返回小时。

```
gbase> SELECT CURRENT_TIMESTAMP(),TO_CHAR(CURRENT_TIMESTAMP(),'HH') FROM t;
+--------------------+---------------------------------+
| CURRENT_TIMESTAMP() | TO_CHAR(CURRENT_TIMESTAMP(),'HH') |
+--------------------+---------------------------------+
| 2013-12-17 16:46:48 | 04                              |
+--------------------+---------------------------------+
1 row in set
```

FORMAT 为“HH12”，返回 12 小时制的小时。

```
gbase> SELECT NOW(),TO_CHAR(NOW(),'HH12') FROM t;
+--------------------+----------------------+
| NOW()               | TO_CHAR(NOW(),'HH12') |
```

```
+---------------------+----------------------+
| 2013-12-17 16:46:55 | 04                   |
+---------------------+----------------------+
1 row in set
```

FORMAT 为“HH24”，返回 24 小时制的小时。

```
gbase> SELECT NOW(),TO_CHAR(TIMESTAMPADD(HOUR,8,NOW()),'HH24') FROM t;
+---------------------+-----------------------------------------+
| NOW()               | TO_CHAR(TIMESTAMPADD(HOUR,8,NOW()),'HH24') |
+---------------------+-----------------------------------------+
| 2013-12-17 16:47:01 | 00                                      |
+---------------------+-----------------------------------------+
1 row in set
```

示例 9：FORMAT 为“IW”，返回一年中的第几周。

```
gbase> SELECT TO_CHAR(NOW(),'IW') FROM t;
+---------------------+
| TO_CHAR(NOW(),'IW') |
+---------------------+
| 51                  |
+---------------------+
1 row in set
```

示例 10：FORMAT 为“MI”，返回分钟数。

```
gbase> SELECT NOW(),TO_CHAR(NOW(),'MI') FROM t;
+---------------------+---------------------+
| NOW()               | TO_CHAR(NOW(),'MI') |
+---------------------+---------------------+
| 2013-12-17 16:47:20 | 47                  |
+---------------------+---------------------+
1 row in set
```

示例 11：FORMAT 为“MM”，“MON”，“MONTH”，以不同形式返回月份。

```
gbase> SELECT NOW(),TO_CHAR(NOW(),'MM') FROM t;
+---------------------+---------------------+
```

```
| NOW()               | TO_CHAR(NOW(),'MM') |
+---------------------+---------------------+
| 2013-12-17 16:47:26 | 12                  |
+---------------------+---------------------+
1 row in set
```

```
gbase> SELECT NOW(),TO_CHAR(NOW(),'MON') FROM t;
+---------------------+----------------------+
| NOW()               | TO_CHAR(NOW(),'MON') |
+---------------------+----------------------+
| 2013-12-17 16:47:30 | Dec                  |
+---------------------+----------------------+
1 row in set
```

```
gbase> SELECT NOW(),TO_CHAR(NOW(),'MONTH') FROM t;
+---------------------+------------------------+
| NOW()               | TO_CHAR(NOW(),'MONTH') |
+---------------------+------------------------+
| 2013-12-17 16:47:36 | December               |
+---------------------+------------------------+
1 row in set
```

示例 12: FORMAT 为“PM HH12:MM:SS”。

```
gbase> SELECT NOW(),TO_CHAR(NOW(),'PM HH12:MM:SS') FROM t;
+---------------------+-------------------------------+
| NOW()               | TO_CHAR(NOW(),'PM HH12:MM:SS') |
+---------------------+-------------------------------+
| 2013-12-17 16:47:40 | PM 04:12:40                   |
+---------------------+-------------------------------+
1 row in set
```

```
gbase> SELECT NOW(),TO_CHAR(TIMESTAMPADD(HOUR,8,NOW()),'PM HH12:MM:SS') AS
f_FormatShow FROM t;
+---------------------+--------------+
| NOW()               | f_FormatShow |
+---------------------+--------------+
```

```
| 2013-12-17 16:47:47 | AM 12:12:47  |
+---------------------+--------------+
1 row in set
```

示例 13：FORMAT 为“Q YYYY-MM-DD”，返回 NOW()中的日期是第几季度。

```
gbase> SELECT NOW(),TO_CHAR(NOW(),'Q YYYY-MM-DD') FROM t;
+---------------------+-------------------------------+
| NOW()               | TO_CHAR(NOW(),'Q YYYY-MM-DD') |
+---------------------+-------------------------------+
| 2013-12-17 16:47:52 | 4 2013-12-17                  |
+---------------------+-------------------------------+
1 row in set
```

示例 14：FORMAT 为“RM”，返回用罗马数字表示的月份。

```
gbase> SELECT NOW(),TO_CHAR(NOW(),'RM') FROM t;
+---------------------+--------------------+
| NOW()               | TO_CHAR(NOW(),'RM') |
+---------------------+--------------------+
| 2013-12-17 16:47:59 | XII                |
+---------------------+--------------------+
1 row in set
```

示例 15：FORMAT 为“RR”、“RRRR”，返回 2 位或 4 位的年。

```
gbase> SELECT TO_CHAR(NOW(),'RR') FROM t;
+--------------------+
| TO_CHAR(NOW(),'RR') |
+--------------------+
| 13                 |
+--------------------+
1 row in set

gbase> SELECT TO_CHAR(NOW(),'RRRR') FROM t;
+---------------------+
| TO_CHAR(NOW(),'RRRR') |
+---------------------+
```

```
| 2013                 |
+----------------------+
1 row in set
```

```
gbase> SELECT TO_CHAR(TIMESTAMPADD(YEAR,-1200,NOW()),'RRRR') FROM t;
+---------------------------------------------+
| TO_CHAR(TIMESTAMPADD(YEAR,-1200,NOW()),'RRRR') |
+---------------------------------------------+
| 0813                                        |
+---------------------------------------------+
1 row in set
```

示例 16：FORMAT 为“SCC”，返回日期所属的世纪数。

```
gbase> SELECT NOW(),TO_CHAR(NOW(),'SCC') FROM t;
+---------------------+----------------------+
| NOW()               | TO_CHAR(NOW(),'SCC') |
+---------------------+----------------------+
| 2013-12-17 16:48:24 |  21                  |
+---------------------+----------------------+
1 row in set
```

```
gbase> SELECT TIMESTAMPADD(YEAR,-20,NOW()) AS
f_DATETIME,TO_CHAR(TIMESTAMPADD(YEAR,-20,NOW()),'SCC') AS f_AD FROM t;
+---------------------+------+
| f_DATETIME          | f_AD |
+---------------------+------+
| 1993-12-17 16:48:31 |  20  |
+---------------------+------+
1 row in set
```

示例 17：FORMAT 为“SSSSS”，返回一天从午夜开始的累积秒数。

```
gbase> SELECT NOW(),TO_CHAR(NOW(),'SS'),TO_CHAR(NOW(),'SSSSS') FROM t;
+---------------------+--------------------+-----------------------+
| NOW()               | TO_CHAR(NOW(),'SS') | TO_CHAR(NOW(),'SSSSS') |
+---------------------+--------------------+-----------------------+
| 2013-12-17 16:48:47 | 47                 | 60527                 |
```

```
+--------------------+--------------------+----------------------+
1 row in set
```

示例 18：FORMAT 为“TS”，返回带有 AM 或者 PM 的时分秒形式的时间。

```
gbase> SELECT NOW(),TO_CHAR(NOW(),'TS') FROM t;
+--------------------+--------------------+
| NOW()              | TO_CHAR(NOW(),'TS') |
+--------------------+--------------------+
| 2013-12-17 17:39:31 | 05:39:31 PM        |
+--------------------+--------------------+
1 row in set

gbase> SELECT TIMESTAMPADD(HOUR,8,NOW()) AS
f_now,TO_CHAR(TIMESTAMPADD(HOUR,8,NOW()),'TS') AS f_now_ts FROM t;
+--------------------+-------------+
| f_now              | f_now_ts    |
+--------------------+-------------+
| 2013-12-18 01:39:38 | 01:39:38 AM |
+--------------------+-------------+
1 row in set
```

示例 19：FORMAT 为“W”，返回日期所在月份的第几周。

```
gbase> SELECT NOW(),TO_CHAR(NOW(),'W') FROM t;
+--------------------+--------------------+
| NOW()              | TO_CHAR(NOW(),'W') |
+--------------------+--------------------+
| 2013-12-17 17:39:46 | 3                  |
+--------------------+--------------------+
1 row in set
```

3.3.39 TO_NUMBER(expr)

将字符串 expr 所包含的数据转化为 NUMBER 型数据。

expr 的形式可为任何支持格式的字符串，如“111.0023”，“23,000,000”。

示例 1：expr 为“-12.340000”。

```
gbase> SELECT TO_NUMBER('-12.340000') FROM t;
+-------------------------+
| TO_NUMBER('-12.340000') |
+-------------------------+
|                  -12.34 |
+-------------------------+
1 row in set
```

示例 2：expr 为“12.34”。

```
gbase> SELECT TO_NUMBER('12.34') FROM t;
+--------------------+
| TO_NUMBER('12.34') |
+--------------------+
|              12.34 |
+--------------------+
1 row in set
```

示例 3：expr 为“+000000123”。

```
gbase> SELECT TO_NUMBER('+000000123') FROM t;
+--------------------------+
| TO_NUMBER('+000000123') |
+--------------------------+
|                      123 |
+--------------------------+
1 row in set
```

示例 4：expr 为“1234”。

```
gbase> SELECT TO_NUMBER('1234') FROM t;
+-------------------+
| TO_NUMBER('1234') |
+-------------------+
|              1234 |
+-------------------+
1 row in set
```

3.3.40 TRANSLATE(char,from_string,to_string)

返回字符串 char 中将 from_string 中的每个字符替换为 to_string 中的相应字符以后的字符串。

to_string 不能省略。

如果 from_string 比 to_string 长，那么在 from_string 中而不在 to_string 中的额外字符将从 char 中删除，因为它们没有相应的替换字符。

如果 TRANSLATE 中的任何参数为 NULL，则结果也是 NULL。

示例 1：from_string 长度长于 to_string，在 from_string 中而不在 to_string 中的额外字符将从 char 中删除。

```
gbase> SELECT TRANSLATE('123abc','2dc','4e') FROM t;
+-------------------------------+
| TRANSLATE('123abc','2dc','4e') |
+-------------------------------+
| 143ab                         |
+-------------------------------+
1 row in set
```

因为 from_string 和 to_string 的位置是一一对应的，2 对应 4，d 对应 e，c 没有对应的值，所以 c 会被删除。字符里的 2 会替换为 4，d 因为字符串里没有，不做替换，c 由于没有对应的替换字符，所以字符串里的 c 会被删除。因此输出结果是 143ab。

示例 2：from_string 长度长于 to_string，在 from_string 中而不在 to_string 中的额外字符将从 char 中删除。

```
gbase> SELECT TRANSLATE('13579abc','13a','24') FROM t;
+---------------------------------+
| TRANSLATE('13579abc','13a','24') |
+---------------------------------+
```

```
| 24579bc                          |
+----------------------------------+
1 row in set
```

示例 3：from_string 为 NULL，返回值为 NULL。

```
gbase> SELECT TRANSLATE('23',NULL,'a') FROM t;
+--------------------------+
| TRANSLATE('23',NULL,'a') |
+--------------------------+
| NULL                     |
+--------------------------+
1 row in set
```

3.3.41 TRIM

TRIM([{BOTH | LEADING | TRAILING} [trim_char] FROM] str)

移除字符串 str 中所有的 trim_char 前缀或后缀，然后将其返回。

如果没有给出任何 BOTH、LEADING 或 TRAILING 修饰符，会假定为 BOTH。

如果没有指定 trim_char，将移除空格。

示例 1：没有指定 trim_char，将移除空格。

```
gbase> SELECT TRIM('  bar   ') FROM t;
+------------------+
| TRIM('  bar   ') |
+------------------+
| bar              |
+------------------+
1 row in set
```

示例 2：使用 LEADING 修饰符。

```
gbase> SELECT TRIM(LEADING 'x' FROM 'xxxbarxxx') FROM t;
+-----------------------------------+
```

```
| TRIM(LEADING 'x' FROM 'xxxbarxxx') |
+------------------------------------+
| barxxx                             |
+------------------------------------+
1 row in set
```

示例 3：使用 BOTH 修饰符。

```
gbase> SELECT TRIM(BOTH 'x' FROM 'xxxbarxxx') FROM t;
+-----------------------------------+
| TRIM(BOTH 'x' FROM 'xxxbarxxx') |
+-----------------------------------+
| bar                               |
+-----------------------------------+
1 row in set
```

示例 4：使用 TRAILING 修饰符。

```
gbase> SELECT TRIM(TRAILING 'xyz' FROM 'barxxyz') FROM t;
+--------------------------------------+
| TRIM(TRAILING 'xyz' FROM 'barxxyz') |
+--------------------------------------+
| barx                                 |
+--------------------------------------+
1 row in set
```

3.3.42UCASE(str)

依照当前字符集设置映射，将字符串 str 中的所有字符改变为大写，然后返回该值。

示例 1：将字符串转为大写。

```
gbase> SELECT UCASE('Gbase') FROM t;
+----------------+
| UCASE('Gbase') |
+----------------+
```

```
| GBASE          |
+----------------+
1 row in set
```

3.3.43 UNHEX(str)

HEX(str)的反运算。

它解释参数中每一对十六进制数字成一个数值，然后转换成数值表示的字符，返回的结果字符是一个二进制字符。

示例 1：将 str 转换成数值表示的字符，str 为十六进制数字。

```
gbase> SELECT UNHEX('4742617365') FROM t;
+---------------------+
| UNHEX('4742617365') |
+---------------------+
| GBase               |
+---------------------+
1 row in set
```

示例 2：将 str 转换成数值表示的字符，str 为十六进制数字。

```
gbase> SELECT 0x4742617365 FROM t;
+--------------+
| 0x4742617365 |
+--------------+
| GBase        |
+--------------+
1 row in set
```

示例 3：UNHEX(HEX())函数。

```
gbase> SELECT UNHEX(HEX('string')) FROM t;
+----------------------+
| UNHEX(HEX('string')) |
+----------------------+
```

```
| string               |
+----------------------+
1 row in set
```

示例 4：HEX(UNHEX())函数。

```
gbase> SELECT HEX(UNHEX('1267')) FROM t;
+--------------------+
| HEX(UNHEX('1267')) |
+--------------------+
| 1267               |
+--------------------+
1 row in set
```

3.3.44UPPER(str)

依照当前字符集设置映射，将字符串 str 中的所有字符改变为大写，然后返回该值。

UPPER()等价于 UCASE()。

示例 1：将字符串转换为大写。

```
gbase> SELECT UPPER('Hej') FROM t;
+--------------+
| UPPER('Hej') |
+--------------+
| HEJ          |
+--------------+
1 row in set
```

3.3.45字符串转换类型函数

GBase 8a 会自动地将数字转换到字符串，或是将字符串转换为数字。

如果将一个二进制字符串作为参数传递给一个字符串函数，结果返回也是一个二进制字符串。

一个数字被转换为字符串，该字符串被视为是一个二进制字符串，但有可能会影响最终结果。

示例 1：自动地将数字转换到字符串，或是将字符串转换为数字。

```
gbase> SELECT 1 + '1' FROM t;
+---------+
| 1 + '1' |
+---------+
|       2 |
+---------+
1 row in set

gbase> SELECT CONCAT(2,' test') FROM t;
+-------------------+
| CONCAT(2,' test') |
+-------------------+
| 2 test            |
+-------------------+
1 row in set

gbase> SELECT 38.8, CONCAT(38.8) FROM t;
+------+--------------+
| 38.8 | CONCAT(38.8) |
+------+--------------+
| 38.8 | 38.8         |
+------+--------------+
1 row in set
```

示例 2：如果明确需要将一个数字转换为字符串，可以使用 CAST()或 CONCAT()函数。建议使用 CAST()。

```
gbase> SELECT 38.8, CAST(38.8 AS CHAR) FROM t;
+------+--------------------+
| 38.8 | CAST(38.8 AS CHAR) |
```

```
+------+------------------+
| 38.8 | 38.8             |
+------+------------------+
1 row in set
```

3.3.46 expr LIKE pat [ESCAPE 'escape-char']

expr LIKE pat [ESCAPE 'escape-char']

使用 SQL 的简单的正则表达式进行比较的模式匹配。

如果表达式 expr 匹配 pat，返回 1（TRUE），否则返回 0（FALSE）。

如果 expr 或 pat 是 NULL，那么结果为 NULL。

模式未必就是文字字符串，例如，它可以使用字符串表达式或表列。

可以在模式中使用下面所示的两个通配符与 LIKE 配合使用：

字　符	含　义
%	匹配任意多个字符，甚至是零个字符。
_	严格地匹配一个字符。

示例 1：expr 与 pat 相匹配，通配符为“_”，返回 1。

```
gbase> SELECT 'David!' LIKE 'David_' FROM t;
+----------------------+
| 'David!' LIKE 'David_' |
+----------------------+
|                    1 |
+----------------------+
1 row in set
```

示例 2：expr 与 pat 相匹配，通配符为“%”，返回 1。

```
gbase> SELECT 'David!' LIKE '%D%v%' FROM t;
+----------------------+
| 'David!' LIKE '%D%v%' |
```

```
+----------------------+
|                    1 |
+----------------------+
1 row in set
```

若要对通配符的文字实例进行检验，可将转义字符放在该字符前面。如果没有指定 ESCAPE 字符，则假设为“\”。

字符串	含　义
\%	匹配一个%字符。
_	匹配一个_字符。

示例 3：expr 与 pat 不匹配，返回 0。

```
gbase> SELECT 'David!' LIKE 'David\_' FROM t;
+-------------------------+
| 'David!' LIKE 'David\_' |
+-------------------------+
|                       0 |
+-------------------------+
1 row in set
```

示例 4：转义字符“_”匹配“_”。

```
gbase> SELECT 'David_' LIKE 'David\_' FROM t;
+-------------------------+
| 'David_' LIKE 'David\_' |
+-------------------------+
|                       1 |
+-------------------------+
1 row in set
```

示例 5：为了指定一个不同的转义字符，可以使用 ESCAPE 子句。

```
gbase> SELECT 'David_' LIKE 'David|_' ESCAPE '|' FROM t;
+-----------------------------------+
| 'David_' LIKE 'David|_' ESCAPE '|' |
+-----------------------------------+
```

```
|                                    1 |
+--------------------------------------+
1 row in set
```

示例 6 和示例 7 表明，字符串比较是忽略大小写的，除非任一操作数是一个二进制字符串。

示例 6:

```
gbase> SELECT 'abc' LIKE 'ABC' FROM t;
+------------------+
| 'abc' LIKE 'ABC' |
+------------------+
|                1 |
+------------------+
1 row in set
```

示例 7:

```
gbase> SELECT 'abc' LIKE BINARY 'ABC' FROM t;
+------------------------+
| 'abc' LIKE BINARY 'ABC' |
+------------------------+
|                      0 |
+------------------------+
1 row in set
```

示例 8: LIKE 允许用在一个数字表达式上。

```
gbase> SELECT 10 LIKE '1%' FROM t;
+--------------+
| 10 LIKE '1%' |
+--------------+
|            1 |
+--------------+
1 row in set
```

注意: 由于 GBase 8a 在字符串中使用 C 转义语法(例如，用“\n”代表一

个换行字符)，在 LIKE 字符串中，必须将用到的“\”双写。

例如，若要查找“\n”,必须将其写成“\\n”。而若要查找“\”,则必须将其写成 it as ‘\\\\’ ;。原因是反斜线符号会被语法分析程序剥离一次，在进行模式匹配时，又会被剥离一次，最后会剩下一个反斜线符号接受匹配。

示例 9：如果 expr 或 pat 是 NULL，结果也是 NULL。

```
gbase> SELECT NULL LIKE 'David_', 'David!' LIKE NULL FROM t;
+------------------+--------------------+
| NULL LIKE 'David_' | 'David!' LIKE NULL  |
+------------------+--------------------+
|               NULL |                NULL |
+------------------+--------------------+
1 row in set
```

3.3.47 expr NOT LIKE pat [ESCAPE 'escape-char']

等同于 NOT (expr LIKE pat [ESCAPE 'escape-char'])。

expr NOT REGEXP pat , expr NOT RLIKE pat

等同于 NOT (expr REGEXP pat)。

如果表达式 expr 匹配 pat，返回 0；否则返回 1。

如果 expr 或 pat 是 NULL，那么结果为 NULL。

示例 1：expr 与 pat 不匹配，返回 1。

```
gbase>  SELECT NOT ('Monty!' REGEXP 'm%y%%' ) FROM t;
+------------------------------+
| not ('Monty!' REGEXP 'm%y%%' ) |
+------------------------------+
|                            1 |
+------------------------------+
1 row in set
```

示例 2：expr 与 pat 相匹配，返回 0。

```
gbase> SELECT 'David_' NOT LIKE 'David|_' ESCAPE '|';
+--------------------------------------+
| 'David_' NOT LIKE 'David|_' ESCAPE '|' |
+--------------------------------------+
|                                    0 |
+--------------------------------------+
1 row in set
```

示例 3：如果 expr 或 pat 是 NULL，结果也是 NULL。

```
gbase> SELECT NULL NOT LIKE 'David_', 'David!' NOT LIKE NULL FROM t;
+----------------------+------------------------+
| NULL NOT LIKE 'David_' | 'David!' NOT LIKE NULL  |
+----------------------+------------------------+
|                 NULL |                   NULL |
+----------------------+------------------------+
1 row in set
```

3.3.48 expr REGEXP pat，expr RLIKE pat

依照模式 pat 对字符串表达式 expr 执行一个模式比较。

模式可以是一个扩展的正则表达式，文字字符串，也可以是字符串表达式或表列。

如果表达式 expr 匹配 pat，返回 1，否则返回 0。

RLIKE 是 REGEXP 的同义词。

REGEXP 对于正常的（不是二进制）字符串是大小写不敏感的。

示例 1：expr 与 pat 不匹配，返回 0。

```
gbase> SELECT 'Monty!' REGEXP 'm%y%%' FROM t;
+-----------------------+
| 'Monty!' REGEXP 'm%y%%' |
+-----------------------+
```

```
|                        0 |
+--------------------------+
1 row in set
```

示例 2：expr 与 pat 相匹配，返回 1。

```
gbase> SELECT 'Monty!' REGEXP '.*' FROM t;
+----------------------+
| 'Monty!' REGEXP '.*' |
+----------------------+
|                    1 |
+----------------------+
1 row in set
```

示例 3：expr 与 pat 相匹配，返回 1。表达式中包含转义字符。

```
gbase> SELECT 'new*\n*line' REGEXP 'new\\*.\\*line' FROM t;
+--------------------------------------+
| 'new*\n*line' REGEXP 'new\\*.\\*line' |
+--------------------------------------+
|                                    1 |
+--------------------------------------+
1 row in set
```

示例 4：表达式前加 BINARY，区分大小写。

```
gbase> SELECT 'a' REGEXP 'A', 'a' REGEXP BINARY 'A' FROM t;
+---------------+-----------------------+
| 'a' REGEXP 'A' | 'a' REGEXP BINARY 'A' |
+---------------+-----------------------+
|              1 |                     0 |
+---------------+-----------------------+
1 row in set
```

示例 5：expr 与 pat 相匹配，返回 1。

```
gbase> SELECT 'a' REGEXP '^[a-d]' FROM t;
+--------------------+
```

```
| 'a' REGEXP '^[a-d]' |
+---------------------+
|                   1 |
+---------------------+
1 row in set
```

3.3.49 STRCMP(expr1,expr2)

STRCMP()是字符串比较函数。

如果字符串 expre1 和 expr2 相同，STRCMP()返回 0。

如果 expre1 根据当前排序次序小于 expre2，返回-1，否则返回 1。

示例 1：expre1 根据当前排序次序小于 expre2，返回-1。

```
gbase> SELECT STRCMP('text', 'text2') FROM t;
+-------------------------+
| STRCMP('text', 'text2') |
+-------------------------+
|                      -1 |
+-------------------------+
1 row in set
```

示例 2：expre1 根据当前排序次序大于 expre2，返回 1。

```
gbase> SELECT STRCMP('text2', 'text') FROM t;
+-------------------------+
| STRCMP('text2', 'text') |
+-------------------------+
|                       1 |
+-------------------------+
1 row in set
```

示例 3：expre1 根据当前排序次序等于 expre2，返回 0。

```
gbase> SELECT STRCMP('text', 'text') FROM t;
```

```
+-----------------------+
| STRCMP('text', 'text') |
+-----------------------+
|                     0 |
+-----------------------+
1 row in set
```

当进行对比时 STRCMP()使用当前的字符集，这使得默认的比较行为大小写不敏感，除非操作数中的任一个或全部为二进制字符串。

3.4 数值函数

3.4.1 算术操作符

常用的算术操作符均是可用的。

注意，如果两个参数均是整型，“-”，“+”和“*”以 BIGINT(64 位）精度运算并返回结果。

如果一个参数是无符号的整数，其他参数是整数，结果为无符号整数。

3.4.1.1 + 加法

示例 1：两个操作数都是整型。

```
gbase> SELECT 3+5 FROM t;
+-----+
| 3+5 |
+-----+
|   8 |
+-----+
1 row in set
```

3.4.1.2 - 减法

示例 1：两个操作数都是整型。

```
gbase> SELECT 3-5 FROM t;
+-----+
| 3-5 |
+-----+
|  -2 |
+-----+
1 row in set
```

3.4.1.3 - 一元减

改变参数的符号。

示例 1：操作数为整型。

```
gbase> SELECT - 2 FROM t;
+-----+
| - 2 |
+-----+
|  -2 |
+-----+
1 row in set
```

注意，如果操作数是 BIGINT 类型，那么返回值也是 BIGINT 类型。

3.4.1.4 * 乘法

示例 1：两个操作数都是整型。

```
gbase> SELECT 3*5 FROM t;
+-----+
| 3*5 |
+-----+
```

```
|  15 |
+-----+
1 row in set
```

3.4.1.5 / 除法

示例 1：两个操作数都是整型。

```
gbase> SELECT 3/5 FROM t;
+--------+
| 3/5    |
+--------+
| 0.6000 |
+--------+
1 row in set
```

示例 2：除数为 0，返回值为 NULL。

```
gbase> SELECT 102/(1-1) FROM t;
+------------+
| 102/(1-1) |
+------------+
|       NULL |
+------------+
1 row in set
```

只有当在一个结果被转换到一个整数的上下文中执行时，除法才会以 BIGINT 进行算术计算。

3.4.1.6 DIV 整数除法

示例 1：两个操作数都是整型。

```
gbase> SELECT 5 DIV 2 FROM t;
+----------+
```

```
| 5 DIV 2 |
+---------+
|       2 |
+---------+
1 row in set
```

3.4.2 数学函数

所有的数学函数在发生错误的情况下，均返回 NULL。

3.4.2.1 ABS(X)

返回 X 的绝对值。这个函数支持使用 BIGINT 值。

示例 1：X 为正数。

```
gbase> SELECT ABS(2) FROM t;
+--------+
| ABS(2) |
+--------+
|      2 |
+--------+
1 row in set
```

示例 2：X 为负数。

```
gbase> SELECT ABS(-32) FROM t;
+----------+
| ABS(-32) |
+----------+
|       32 |
+----------+
1 row in set
```

3.4.2.2 ACOS(X)

返回 X 的反余弦，即返回余弦值为 X 的值。

如果 X 不在-1 到 1 之间的范围内，返回 NULL。

示例 1：X 为正数。

```
gbase> SELECT ACOS(1) FROM t;
+---------+
| ACOS(1) |
+---------+
|       0 |
+---------+
1 row in set
```

示例 2：X 为小数。

```
gbase> SELECT ACOS(1.0001) FROM t;
+--------------+
| ACOS(1.0001) |
+--------------+
|         NULL |
+--------------+
1 row in set
```

示例 3：X 为 0。

```
gbase> SELECT ACOS(0) FROM t;
+-----------------+
| ACOS(0)         |
+-----------------+
| 1.5707963267949 |
+-----------------+
1 row in set
```

3.4.2.3 ASIN(X)

返回 X 的反正弦，返回正弦值为 X 的值。

如果 X 不在-1 到 1 之间的范围内，返回 NULL。

示例 1：X 为小数。

```
gbase> SELECT ASIN(0.2) FROM t;
+-------------------+
| ASIN(0.2)         |
+-------------------+
| 0.201357920790331 |
+-------------------+
1 row in set
```

示例 2：X 为正整数。

```
gbase> SELECT ASIN(2) FROM t;
+---------+
| ASIN(2) |
+---------+
|    NULL |
+---------+
1 row in set
```

3.4.2.4 ATAN(X)

返回 X 的反正切，即返回正切值为 X 的值。

示例 1：X 为正整数。

```
gbase> SELECT ATAN(2) FROM t;
+------------------+
| ATAN(2)          |
+------------------+
| 1.10714871779409 |
```

```
+------------------+
1 row in set
```

示例 2：X 为负整数。

```
gbase> SELECT ATAN(-2) FROM t;
+-------------------+
| ATAN(-2)          |
+-------------------+
| -1.10714871779409 |
+-------------------+
1 row in set
```

3.4.2.5 ATAN(Y,X)，ATAN2(Y,X)

返回两个变量 X 和 Y 的反正切。它类似于计算 Y/X 的反正切，两个参数的符号用于决定结果所在的象限。

示例 1：返回“-2/2”的反正切。

```
gbase> SELECT ATAN(-2,2) FROM t;
+--------------------+
| ATAN(-2,2)         |
+--------------------+
| -0.785398163397448 |
+--------------------+
1 row in set
```

示例 2：返回“PI()/0”的反正切。

```
gbase> SELECT ATAN2(PI(),0) FROM t;
+-----------------+
| ATAN2(PI(),0)   |
+-----------------+
| 1.5707963267949 |
+-----------------+
```

```
1 row in set
```

3.4.2.6 CEILING(X), CEIL(X)

返回不小于 X 的最小整数。

示例 1: X 为正数。

```
gbase> SELECT CEILING(1.23) FROM t;
+---------------+
| CEILING(1.23) |
+---------------+
|             2 |
+---------------+
1 row in set
```

示例 2: X 为负数。

```
gbase> SELECT CEIL(-1.23) FROM t;
+-------------+
| CEIL(-1.23) |
+-------------+
|          -1 |
+-------------+
1 row in set
```

3.4.2.7 COS(X)

返回 X 的余弦, X 以弧度给出。

示例 1: X 为 PI()。

```
gbase> SELECT COS(PI())FROM t;
+-----------+
| COS(PI()) |
```

```
+-----------+
|        -1 |
+-----------+
1 row in set
```

3.4.2.8 COT(X)

返回 X 的余切。

示例 1：X 为正数。

```
gbase> SELECT COT(12) FROM t;
+--------------------+
| COT(12)            |
+--------------------+
| -1.57267340639769 |
+--------------------+
1 row in set
```

示例 2：X 为 0。

```
gbase> SELECT COT(0) FROM t;
+---------+
| COT(0) |
+---------+
|   NULL |
+---------+
1 row in set
```

3.4.2.9 CRC32(expr)

计算循环冗余码校验值并返回一个 32 比特无符号值。

expr 应为一个字符串，而且在不是字符串的情况下会被作为字符串处理(若

能成功转换为字符串类型）。

若参数为 NULL，则结果为 NULL。

示例 1：expr 为字符串。

```
gbase> SELECT CRC32('GBase') FROM t;
+----------------+
| CRC32('GBase') |
+----------------+
|     3594920295 |
+----------------+
1 row in set
```

示例 2：expr 为数字。

```
gbase> SELECT CRC32(1.034) FROM t;
+--------------+
| CRC32(1.034) |
+--------------+
|   2318494194 |
+--------------+
1 row in set
```

3.4.2.10 DEGREES(X)

将参数 X 从弧度转换为角度。

示例 1：X 为 PI()。

```
gbase> SELECT DEGREES(PI())FROM t;
+---------------+
| DEGREES(PI()) |
+---------------+
|           180 |
+---------------+
1 row in set
```

3.4.2.11 EXP(X)

返回基数为 e，幂值为 X 的值，即返回 e 的 n 次幂。

示例 1：返回 e 的 2 次幂。

```
gbase> SELECT EXP(2) FROM t;
+------------------+
| EXP(2)           |
+------------------+
| 7.38905609893065 |
+------------------+
1 row in set
```

示例 2：返回 e 的-2 次幂。

```
gbase> SELECT EXP(-2) FROM t;
+--------------------+
| EXP(-2)            |
+--------------------+
| 0.135335283236613  |
+--------------------+
1 row in set
```

3.4.2.12 FLOOR(X)

返回不大于 X 的最大整数值。

如果参数 X 是 NULL，则返回结果为 NULL。

示例 1：X 为正数。

```
gbase> SELECT FLOOR(1.23) FROM t;
+-------------+
| FLOOR(1.23) |
```

```
+-------------+
|           1 |
+-------------+
1 row in set
```

示例 2：X 为负数。

```
gbase> SELECT FLOOR(-1.23) FROM t;
+--------------+
| FLOOR(-1.23) |
+--------------+
|           -2 |
+--------------+
1 row in set
```

示例 3：X 为 NULL。

```
gbase> SELECT FLOOR(NULL) FROM t;
+-------------+
| FLOOR(NULL) |
+-------------+
|        NULL |
+-------------+
1 row in set
```

3.4.2.13 LN(X)

返回 X 的自然对数。

示例 1：返回 2 的自然对数。

```
gbase> SELECT LN(2) FROM t;
+-------------------+
| LN(2)             |
+-------------------+
| 0.693147180559945 |
+-------------------+
```

```
1 row in set
```

示例 2：返回-2 的自然对数。

```
gbase> SELECT LN(-2) FROM t;
+--------+
| LN(-2) |
+--------+
|   NULL |
+--------+
1 row in set
```

3.4.2.14 LOG(X)，LOG(B,X)

如果以一个参数调用，它返回 X 的自然对数。

这个函数同 LN(X)具有相同意义。

示例 1：返回 2 的自然对数。

```
gbase> SELECT LOG(2) FROM t;
+--------------------+
| LOG(2)             |
+--------------------+
| 0.693147180559945 |
+--------------------+
1 row in set
```

示例 2：返回-2 的自然对数。

```
gbase> SELECT LOG(-2) FROM t;
+---------+
| LOG(-2) |
+---------+
|    NULL |
+---------+
1 row in set
```

示例 3：如果以两个参数调用，这个函数返回以 B 为底，X 的对数。

```
gbase> SELECT LOG(2,65536) FROM t;
+--------------+
| LOG(2,65536) |
+--------------+
|           16 |
+--------------+
1 row in set
```

示例 4：LOG(B,X)等同于 LOG(X)/LOG(B)。

```
gbase> SELECT LOG(1,100) FROM t;
+------------+
| LOG(1,100) |
+------------+
|       NULL |
+------------+
1 row in set
```

3.4.2.15 LOG2(X)

返回 X 的以 2 为底的对数。通常用于算出一个数字需要多少比特位存储。

示例 1：返回以 2 为底，“65536”的对数。

```
gbase> SELECT LOG2(65536) FROM t;
+-------------+
| LOG2(65536) |
+-------------+
|          16 |
+-------------+
1 row in set
```

示例 2：返回以 2 为底，“-100”的对数。

```
gbase> SELECT LOG2(-100) FROM t;
+------------+
| LOG2(-100) |
+------------+
|       NULL |
+------------+
1 row in set
```

3.4.2.16 LOG10(X)

返回 X 以 10 为底的对数。

示例 1：返回以 10 为底，“2”的对数。

```
gbase> SELECT LOG10(2) FROM t;
+-------------------+
| LOG10(2)          |
+-------------------+
| 0.301029995663981 |
+-------------------+
1 row in set
```

示例 2：返回以 10 为底，“100”的对数。

```
gbase> SELECT LOG10(100) FROM t;
+-------------+
| LOG10(100) |
+-------------+
|           2 |
+-------------+
1 row in set
```

示例 3：返回以 10 为底，“-100”的对数。

```
gbase> SELECT LOG10(-100) FROM t;
+--------------+
| LOG10(-100) |
```

```
+-------------+
|        NULL |
+-------------+
1 row in set
```

3.4.2.17 MOD(N,M), N % M, N MOD M

取模。返回 N 除以 M 后的余数。

示例 1：返回 234 除以 10 的余数。

```
gbase> SELECT MOD(234, 10) FROM t;
+--------------+
| MOD(234, 10) |
+--------------+
|            4 |
+--------------+
1 row in set
```

示例 2：返回 253 除以 7 的余数。

```
gbase> SELECT 253 % 7 FROM t;
+---------+
| 253 % 7 |
+---------+
|       1 |
+---------+
1 row in set
```

示例 3：MOD(29,9)与 29 MOD 9 结果相同。

```
gbase> SELECT MOD(29,9) FROM t;
+-----------+
| MOD(29,9) |
+-----------+
|         2 |
+-----------+
```

```
1 row in set

gbase> SELECT 29 MOD 9 FROM t;
+----------+
| 29 MOD 9 |
+----------+
|        2 |
+----------+
1 row in set
```

示例 4：MOD()也适用于小数部分，返回除法运算后的精确余数。

```
gbase> SELECT MOD(34.5,3) FROM t;
+-------------+
| MOD(34.5,3) |
+-------------+
|         1.5 |
+-------------+
1 row in set
```

示例 5：取模的三种表现形式。

```
gbase> SELECT 253 % 7, MOD(253,7), 253 MOD 7 FROM t;
+---------+-----------+-----------+
| 253 % 7 | MOD(253,7) | 253 MOD 7 |
+---------+-----------+-----------+
|       1 |          1 |         1 |
+---------+-----------+-----------+
1 row in set
```

3.4.2.18 PI()

返回 PI 值（圆周率）。默认显示 6 位小数，但是在 GBase 8a 内部，为 PI 使用全部的双精度。

示例 1：返回 PI 的值。

```
gbase> SELECT PI()FROM t;
+----------+
| PI()     |
+----------+
| 3.141593 |
+----------+
1 row in set
```

3.4.2.19 POW(X,Y), POWER(X,Y)

返回 X 的 Y 次幂。

示例 1：返回 2 的 2 次幂。

```
gbase> SELECT POW(2,2) FROM t;
+----------+
| POW(2,2) |
+----------+
|        4 |
+----------+
1 row in set
```

示例 2：返回 2 的-2 次幂。

```
gbase> SELECT POW(2,-2) FROM t;
+-----------+
| POW(2,-2) |
+-----------+
|      0.25 |
+-----------+
1 row in set
```

3.4.2.20 RADIANS(X)

将参数 X 从角度转换为弧度，然后返回。

示例 1：返回 90 度对应的弧度。

```
gbase> SELECT RADIANS(90) FROM t;
+-----------------+
| RADIANS(90)     |
+-----------------+
| 1.5707963267949 |
+-----------------+
1 row in set
```

3.4.2.21 RAND(), RAND(N)

返回一个范围在 0 到 1.0 之间的随机浮点值。

如果一个整数参数 N 被指定，它被当做种子值使用（用于产生一个可重复的数值）。

示例 1：返回随机浮点数。

```
gbase> SELECT RAND()FROM t;
+------------------+
| RAND()           |
+------------------+
| 0.504175471611484 |
+------------------+
1 row in set

gbase> SELECT RAND()FROM t;
+------------------+
| RAND()           |
+------------------+
| 0.499486101325123 |
+------------------+
1 row in set
```

```
gbase> SELECT RAND()FROM t;
+------------------+
| RAND()           |
+------------------+
| 0.984904122524806 |
+------------------+
1 row in set
```

示例 2：返回随机浮点数，再次运行 RAND(20)，结果与上一次相同。

```
gbase> SELECT RAND(20) FROM t;
+------------------+
| RAND(20)         |
+------------------+
| 0.158882612510475 |
+------------------+
1 row in set

gbase> SELECT RAND(20) FROM t;
+------------------+
| RAND(20)         |
+------------------+
| 0.158882612510475 |
+------------------+
1 row in set
```

在一个 ORDER BY 子句中，不可以使用 RAND()值作用于一个列，因为 ORDER BY 将多次重复计算列。用户可以以任意次序检索行。

3.4.2.22 ROUND(X)，ROUND(X,D)

ROUND(X)返回参数 X 四舍五入到最近的整数后的值。

ROUND(X,D)返回的 X 值，保留到小数点后 D 位（第 D 位的保留方式为四舍五入）。

如果 D 值为负数，则保留的 X 值为小数点左边的 D 位数字。

示例 1：X 为“-1.23”，返回结果为-1。

```
gbase> SELECT ROUND(-1.23) FROM t;
+--------------+
| ROUND(-1.23) |
+--------------+
|           -1 |
+--------------+
1 row in set
```

示例 2：X 为“-1.58”，返回结果为-2。

```
gbase> SELECT ROUND(-1.58) FROM t;
+--------------+
| ROUND(-1.58) |
+--------------+
|           -2 |
+--------------+
1 row in set
```

示例 3：X 为“1.58”，返回结果为 2。

```
gbase> SELECT ROUND(1.58) FROM t;
+-------------+
| ROUND(1.58) |
+-------------+
|           2 |
+-------------+
1 row in set
```

示例 4：对“1.298”进行四舍五入，小数点后保留 1 位数字。

```
gbase> SELECT ROUND(1.298, 1) FROM t;
+-----------------+
| ROUND(1.298, 1) |
+-----------------+
|             1.3 |
```

```
+-----------------+
1 row in set
```

示例 5：对“1.298”进行四舍五入，小数点后保留 0 位数字。

```
gbase> SELECT ROUND(1.298, 0) FROM t;
+-----------------+
| ROUND(1.298, 0) |
+-----------------+
|               1 |
+-----------------+
1 row in set
```

示例 6：对“23.298”进行四舍五入，小数点后保留“-1”位数字，即个位数字。

```
gbase> SELECT ROUND(23.298, -1) FROM t;
+-------------------+
| ROUND(23.298, -1) |
+-------------------+
|                20 |
+-------------------+
1 row in set
```

返回值类型和第一个参数的类型相同。

当第一个参数是 DECIMAL 时，ROUND()为了精确计算使用精确计算库。

对于精确值数字，ROUND()使用“四舍五入”或“舍入成最接近的数”的规则：

如果一个值的小数部分为.5 或比该值大，那么向上舍入为下一个整数，对于负数来说，向下舍入为下一个负数。

如果一个值的小数部分比.5 小，那么向下舍入为下一个整数，对于负数来说，向上舍入为下一个负数。

对于近似值数字，其结果根据 C 库而定。在很多系统中，这意味着 ROUND()

的使用遵循“舍入成最接近的偶数”的规则：

一个带有任何小数部分的值会被舍入成最接近的偶数整数。对于 25E-1，它认为 20E-1 离它最近。

示例 7：对于精确值和近似值舍入的不同之处

```
gbase> SELECT ROUND(2.5), ROUND(25E-1) FROM t;
+------------+-------------+
| ROUND(2.5) | ROUND(25E-1) |
+------------+-------------+
|          3 |           2 |
+------------+-------------+
1 row in set
```

示例 8：将数值插入到 DECIMAL 列，因为目标是精确值类型，所以舍入使用向上取原则，不考虑插入值是精确值还是近似值。

```
gbase> DROP TABLE IF EXISTS t1;
Query OK, 0 rows affected

gbase> CREATE TABLE t1 (d DECIMAL(10,0));
Query OK, 0 rows affected

gbase> INSERT INTO t1 VALUES(2.5),(2.5E0);
Query OK, 2 rows affected, 2 warnings
Records: 2  Duplicates: 0  Warnings: 2

gbase> SELECT d FROM t1;
+------+
| d    |
+------+
|    3 |
|    3 |
+------+
2 rows in set
```

示例 9：对于 DECIMAL 列和精确值的舍入使用向上舍入一半的原则。数值小

数部分为 0.5 或者更大时，则舍入到最近整数。

```
gbase> SELECT ROUND(2.5), ROUND(-2.5) FROM t;
+------------+-------------+
| ROUND(2.5) | ROUND(-2.5) |
+------------+-------------+
|          3 |          -3 |
+------------+-------------+
1 row in set
```

3.4.2.23 SIGN(X)

根据 X 值是正数、0 还是负数，分别返回-1、0 或 1。

示例 1：X 为负数，返回-1。

```
gbase> SELECT SIGN(-32) FROM t;
+-----------+
| SIGN(-32) |
+-----------+
|        -1 |
+-----------+
1 row in set
```

示例 2：X 为“0”，返回 0。

```
gbase> SELECT SIGN(0) FROM t;
+---------+
| SIGN(0) |
+---------+
|       0 |
+---------+
1 row in set
```

示例 3：X 为正数，返回 1。

```
gbase> SELECT SIGN(234) FROM t;
```

```
+-----------+
| SIGN(234) |
+-----------+
|         1 |
+-----------+
1 row in set
```

3.4.2.24 SIN(X)

返回 X 的正弦，此处，X 以弧度给出。

示例 1：返回“PI()”的正弦。

```
gbase> SELECT SIN(PI())FROM t;
+----------------------+
| SIN(PI())            |
+----------------------+
| 1.22464679914735e-16 |
+----------------------+
1 row in set
```

3.4.2.25 SQRT(X)

返回 X 的非负平方根。

示例 1：返回“4”的平方根。

```
gbase> SELECT SQRT(4) FROM t;
+---------+
| SQRT(4) |
+---------+
|       2 |
+---------+
1 row in set
```

示例 2：返回“20”的平方根。

```
gbase> SELECT SQRT(20) FROM t;
+------------------+
| SQRT(20)         |
+------------------+
| 4.47213595499958 |
+------------------+
1 row in set
```

3.4.2.26 TAN(X)

返回 X 的正切，在这里，X 以弧度给出。

示例 1：返回“PI()+1”的正切值。

```
gbase> SELECT TAN(PI()+1) FROM t;
+-----------------+
| TAN(PI()+1)     |
+-----------------+
| 1.5574077246549 |
+-----------------+
1 row in set
```

3.4.2.27 TRUNCATE(X,D)

返回数值 X 截取到 D 位小数后的数字。

如果 D 为 0，结果将不包含小数点和小数部分。

如果 D 为负数，表示截去（归零）X 值小数点左边第 D 位开始后面所有低位的值。

示例 1：X 为“1.223”，小数点后保留一位。

```
gbase> SELECT TRUNCATE(1.223,1) FROM t;
```

```
+-------------------+
| TRUNCATE(1.223,1) |
+-------------------+
|               1.2 |
+-------------------+
1 row in set
```

示例 2：X 为“1.999”，小数点后保留一位。

```
gbase> SELECT TRUNCATE(1.999,1) FROM t;
+-------------------+
| TRUNCATE(1.999,1) |
+-------------------+
|               1.9 |
+-------------------+
1 row in set
```

示例 3：D 为“0”，返回值不包含小数点和小数部分。

```
gbase> SELECT TRUNCATE(1.999,0) FROM t;
+-------------------+
| TRUNCATE(1.999,0) |
+-------------------+
|                 1 |
+-------------------+
1 row in set
```

示例 4：X 为“-1.999”，小数点后保留一位。

```
gbase> SELECT TRUNCATE(-1.999,1) FROM t;
+--------------------+
| TRUNCATE(-1.999,1) |
+--------------------+
|               -1.9 |
+--------------------+
1 row in set
```

示例 5：所有数字被四舍五入到零。

```
gbase> SELECT TRUNCATE(122,-2) FROM t;
+-----------------+
| TRUNCATE(122,-2) |
+-----------------+
|              100 |
+-----------------+
1 row in set
```

示例 6:十进值小数在计算机中通常不以精确数字存储,而是双精度型的值。

```
gbase> SELECT TRUNCATE(10.28*100,0) FROM t;
+----------------------+
| TRUNCATE(10.28*100,0) |
+----------------------+
|                 1028 |
+----------------------+
1 row in set
```

返回结果为 1028，是因为 10.28 实际上是以某些像 10.2799999999999999 的形式被存储的。

3.5 日期和时间函数

本节描述可以用来操纵时间值的函数。参考日期和时间类型来获取每种日期和时间类型在有效格式下可以表达的值的范围。

返回当前日期或者时间的函数都等于在查询开始执行时的值，仅执行一次。这意味在一个单查询中多次引用像 NOW()这样的函数总会得到一样的结果。这个原则也适用于 CURDATE(),CURTIME(),UTC_DATE(),UTC_TIME(),UTC_TIMESTAMP()和它们的同义词。

CURRENT_TIMESTAMP(), CURRENT_TIME(), CURRENT_DATE()和 FROM_UNIXTIME()返回当前时区，这和 time_zone 系统变量是一样的。还有 UNIX_TIMESTAMP()假设它的参数是当前时区的 datetime 值。

示例 1: 返回当前日期和时间。

```
gbase> SELECT NOW() FROM t;
+---------------------+
| now()               |
+---------------------+
| 2013-12-19 11:18:04 |
+---------------------+
1 row in set
```

示例 2：返回当前日期。

```
gbase> SELECT CURDATE() FROM t;
+------------+
| CURDATE()  |
+------------+
| 2013-12-19 |
+------------+
1 row in set
```

示例 3：返回当前时间。

```
gbase> SELECT CURTIME() FROM t;
+-----------+
| CURTIME() |
+-----------+
| 11:18:13  |
+-----------+
1 row in set
```

示例 4：返回当前 UTC 日期。

```
gbase> SELECT UTC_DATE() FROM t;
+------------+
| UTC_DATE() |
+------------+
| 2013-12-19 |
+------------+
1 row in set
```

示例 5：返回当前 UTC 时间。

```
gbase> SELECT UTC_TIME() FROM t;
+------------+
| UTC_TIME() |
+------------+
| 03:18:22   |
+------------+
1 row in set
```

示例 6：返回当前 UTC 时间戳（日期+时间）。

```
gbase> SELECT UTC_TIMESTAMP() FROM t;
+---------------------+
| UTC_TIMESTAMP()     |
+---------------------+
| 2013-12-19 03:18:27 |
+---------------------+
1 row in set
```

示例 7：返回当前时间戳（日期+时间）。

```
gbase> SELECT CURRENT_TIMESTAMP() FROM t;
+---------------------+
| CURRENT_TIMESTAMP() |
+---------------------+
| 2013-12-19 11:18:36 |
+---------------------+
1 row in set
```

示例 8：返回当前时间。

```
gbase> SELECT CURRENT_TIME() FROM t;
+----------------+
| CURRENT_TIME() |
+----------------+
| 11:18:42       |
+----------------+
1 row in set
```

示例 9：返回当前日期。

```
gbase> SELECT CURRENT_DATE() FROM t;
+----------------+
| CURRENT_DATE() |
+----------------+
| 2013-12-19     |
+----------------+
1 row in set
```

3.5.1 ADDDATE()

ADDDATE(date,INTERVAL expr type)，ADDDATE(expr,days)

关键词 INTERVA 及 type 分类符均不区分大小写。

当调用第二个参数 INTERVAL 时，ADDDATE()等价于 DATE_ADD()。函数 SUBDATE()等价于 DATE_SUB()。

ADDDATE(expr,days)中，expr 是一个日期或者 datetime 表达式，days 是要加入 expr 中的日期的天数。默认增加天数。

示例 1：DATE_ADD(date,INTERVAL expr type)，expr 为日期，返回增加 31 天后的日期。

```
gbase> SELECT DATE_ADD('2010-01-02', INTERVAL 31 DAY) FROM t;
+---------------------------------------+
| DATE_ADD('2010-01-02', INTERVAL 31 DAY) |
+---------------------------------------+
| 2010-02-02                            |
+---------------------------------------+
1 row in set
```

示例 2：ADDDATE(date,INTERVAL expr type)，expr 为日期，返回增加 31 天后的日期。

```
gbase> SELECT ADDDATE('2010-01-02', INTERVAL 31 DAY) FROM t;
+-------------------------------------+
| ADDDATE('2010-01-02', INTERVAL 31 DAY) |
+-------------------------------------+
| 2010-02-02                          |
+-------------------------------------+
1 row in set
```

示例 3：ADDDATE(expr,days)，返回增加 31 天后的日期。

```
gbase> SELECT ADDDATE('2010-01-02', 31 ) FROM t;
+---------------------------+
| ADDDATE('2010-01-02', 31 ) |
+---------------------------+
| 2010-02-02                |
+---------------------------+
1 row in set
```

3.5.2 ADDTIME(expr,expr2)

将 expr2 加到 expr 中并返回结果。

expr 是时间或 datetime 表达式，expr2 是一个时间表达式。

示例 1：expr 为 datetime。

```
gbase> SELECT ADDTIME('2010-01-02 23:59:59.999999','1 1:1:1.000002') FROM t;
+-----------------------------------------------------+
| ADDTIME('2010-01-02 23:59:59.999999','1 1:1:1.000002') |
+-----------------------------------------------------+
| 2010-01-04 01:01:01.000001                          |
+-----------------------------------------------------+
1 row in set
```

示例 2：expr 为时间表达式。

```
gbase> SELECT ADDTIME('01:00:00.999999', '02:00:00.999998') FROM t;
```

```
+-------------------------------------------+
| ADDTIME('01:00:00.999999', '02:00:00.999998') |
+-------------------------------------------+
| 03:00:01.999997                           |
+-------------------------------------------+
1 row in set
```

3.5.3 ADD_MONTHS(date, number)

ADD_MONTHS(date, number)函数是在一个日期上加上指定的月份数，其中，日期中的日是不变的。

如果开始日期是某月的最后一天，结果将会进行调整，返回值会对应新的一月的最后一天。

如果结果月份的天数比开始月份的天数少，返回值也会向回调整以适应有效日期。

- date：一个日期数值。
- number：加上的月份数，如果是要减去的月份数，则为负数。

示例 1：在当前日期时间上加上 3 个月，日期中的日不变。

```
gbase> SELECT NOW(),ADD_MONTHS(NOW(),3) AS f_Show FROM t;
+---------------------+---------------------+
| NOW()               | f_Show              |
+---------------------+---------------------+
| 2013-12-19 11:19:39 | 2014-03-19 11:19:39 |
+---------------------+---------------------+
1 row in set
```

示例 2：增加的月份为负数时，相当于提前月份数。

```
gbase> SELECT NOW(),ADD_MONTHS(NOW(),-3) AS f_Show FROM t;
+---------------------+---------------------+
| NOW()               | f_Show              |
```

```
+---------------------+---------------------+
| 2013-12-19 11:19:43 | 2013-09-19 11:19:43 |
+---------------------+---------------------+
1 row in set
```

示例 3：通过使用 to_date 函数，转换为日期型后，再加上指定的月份。

```
gbase> SELECT ADD_MONTHS(TO_DATE('2012-9-15','YYYY-MM-DD'),3) AS f_Show FROM
t;
+------------+
| f_Show     |
+------------+
| 2012-12-15 |
+------------+
1 row in set
```

示例 4：date 为 TO_DATE()函数。

```
gbase> SELECT ADD_MONTHS(TO_DATE('2012-9-15 12:20:31','YYYY-MM-DD
HH24:MI:SS'),3) AS f_Show FROM t;
+---------------------+
| f_Show              |
+---------------------+
| 2012-12-15 00:00:00 |
+---------------------+
1 row in set
```

示例 5：8 月 31 日是 8 月份最后一天，增加 3 个月后，是 11 月份，11 月共有 30 天，因此结果就是“2012-11-30”。

```
gbase> SELECT ADD_MONTHS(TO_DATE('2012-8-31','YYYY-MM-DD'),3) AS f_Show FROM
t;
+------------+
| f_Show     |
+------------+
| 2012-11-30 |
+------------+
1 row in set
```

3.5.4 CONVERT_TZ(dt,from_tz,to_tz)

CONVERT_TZ()将 datetime 值 dt 从 from_tz 给定的时区转化为 to_tz,并返回结果值。如果参数是不合法的，该函数返回 NULL。

如果在从 from_tz 转化到 UTC 时值超出了 TIMESTAMP 类型支持的范围，就不会进行转化。关于 TIMESTAMP 的取值范围，在“1.4 日期和时间类型”中有描述。

示例 1：要使用诸如“MET”或“Europe/Moscow”命名时区，必须适当的设置时区表。

```
gbase> SELECT CONVERT_TZ('2010-01-01 12:00:00','+00:00','-07:00') FROM t;
+---------------------------------------------------+
| CONVERT_TZ('2010-01-01 12:00:00','+00:00','-07:00') |
+---------------------------------------------------+
| 2010-01-01 05:00:00                               |
+---------------------------------------------------+
1 row in set
```

3.5.5 CURDATE()

以“YYYY-MM-DD”或“YYYYMMDD”格式返回当前的日期值，返回的格式取决于该函数是用于字符串还是数字上下文中。

示例 1：“YYYY-MM-DD”格式返回当前日期。

```
gbase> SELECT CURDATE() FROM t;
+------------+
| CURDATE()  |
+------------+
| 2013-12-19 |
+------------+
1 row in set
```

示例 2： “YYYYMMDD”格式返回当前日期。

```
gbase> SELECT CURDATE() + 0 FROM t;
+---------------+
| CURDATE() + 0 |
+---------------+
|      20131219 |
+---------------+
1 row in set
```

3.5.6 CURRENT_DATE, CURRENT_DATE()

CURRENT_DATE 和 CURRENT_DATE()等价于 CURDATE()。

示例 1：使用 CURRENT_DATE()函数返回日期。

```
gbase> SELECT CURRENT_DATE() FROM t;
+----------------+
| CURRENT_DATE() |
+----------------+
| 2013-12-19     |
+----------------+
1 row in set
```

示例 2：使用 CURRENT_DATE 函数返回日期。

```
gbase> SELECT CURRENT_DATE FROM t;
+---------------+
| CURRENT_DATE  |
+---------------+
| 2013-12-19    |
+---------------+
1 row in set
```

示例 3：使用 CURDATE()函数返回日期。

```
gbase> SELECT CURDATE() FROM t;
+------------+
| CURDATE()  |
+------------+
| 2013-12-19 |
+------------+
1 row in set
```

3.5.7 CURRENT_TIME, CURRENT_TIME()

CURRENT_TIME 和 CURRENT_TIME()等价于 CURTIME()。

示例 1：使用 CURRENT_TIME()函数返回时间。

```
gbase> SELECT CURRENT_TIME() FROM t;
+----------------+
| CURRENT_TIME() |
+----------------+
| 16:03:48       |
+----------------+
1 row in set
```

示例 2：使用 CURTIME()函数返回时间。

```
gbase> SELECT CURTIME() FROM t;
+-----------+
| CURTIME() |
+-----------+
| 16:03:53  |
+-----------+
1 row in set
```

3.5.8 CURRENT_TIMESTAMP, CURRENT_TIMESTAMP()

CURRENT_TIMESTAMP 和 CURRENT_TIMESTAMP()等价于 NOW()。

示例 1：使用 CURRENT_TIMESTAMP 函数返回“日期+时间”。

```
gbase> SELECT CURRENT_TIMESTAMP FROM t;
+---------------------+
| CURRENT_TIMESTAMP   |
+---------------------+
| 2013-12-19 16:04:01 |
+---------------------+
1 row in set
```

示例 2：使用 CURRENT_TIMESTAMP()函数返回“日期+时间”。

```
gbase> SELECT CURRENT_TIMESTAMP() FROM t;
+---------------------+
| CURRENT_TIMESTAMP() |
+---------------------+
| 2013-12-19 16:04:05 |
+---------------------+
1 row in set
```

示例 3：使用 NOW()函数返回“日期+时间”。

```
gbase> SELECT NOW() FROM t;
+---------------------+
| NOW()               |
+---------------------+
| 2013-12-19 16:04:09 |
+---------------------+
1 row in set
```

3.5.9 CURTIME()

以“HH:MI:SS”或“HHMISS”格式返回当前的时间值，返回的格式取决于该函数是用于字符串还是数字的上下文中。

示例 1：以“HH:MI:SS”格式返回当前时间。

```
gbase> SELECT CURTIME() FROM t;
+-----------+
| CURTIME() |
+-----------+
| 11:21:28  |
+-----------+
1 row in set
```

示例 2：返回“CURTIME()+0”对应的时间值。

```
gbase> SELECT CURTIME() + 0 FROM t;
+---------------+
| CURTIME() + 0 |
+---------------+
| 112136.000000 |
+---------------+
1 row in set
```

3.5.10DATE(expr)

从 date 或者 datetime 表达式 expr 中取得日期部分。

如果 expr 是一个非法日期字符串，则返回 NULL。

示例 1：从 datetime 表达式中取得日期部分。

```
gbase> SELECT DATE('2011-09-05 11:22:03') FROM t;
+----------------------------+
| DATE('2011-09-05 11:22:03') |
+----------------------------+
| 2011-09-05                 |
+----------------------------+
1 row in set
```

示例 2：expr 是一个非法日期字符串，则返回 NULL。

```
gbase> SELECT DATE('2011-09-32 11:22:03') FROM t;
+---------------------------+
| DATE('2011-09-32 11:22:03') |
+---------------------------+
| NULL                      |
+---------------------------+
1 row in set, 1 warning

gbase> SHOW WARNINGS;
+---------+------+-----------------------------------------+
| Level   | Code | Message                                 |
+---------+------+-----------------------------------------+
| Warning | 1292 | Incorrect datetime value: '2011-09-32 11:22:03' |
+---------+------+-----------------------------------------+
1 row in set
```

3.5.11 DATEDIFF(expr,expr2)

DATEDIFF()返回开始日期 expr 和结束日期 expr2 之间的天数。

expr 和 expr2 是 date 或者 datetime 表达式。只有日期部分用于计算。如果用于计算日期间隔的参数不是一个 date 或者 datetime 类型，例如，TIME 型数据，计算结果是不可信的。

示例 1：expr 晚于 expr2，返回的天数是正数。

```
gbase> SELECT DATEDIFF('2010-08-30 23:59:59','2010-08-29') FROM t;
+-----------------------------------------+
| DATEDIFF('2010-08-30 23:59:59','2010-08-29') |
+-----------------------------------------+
|                                       1 |
+-----------------------------------------+
1 row in set
```

示例 2：expr 早于 expr2，返回的天数是负数。

```
gbase> SELECT DATEDIFF('2010-07-31 23:59:59','2010-08-30') FROM t;
+-----------------------------------------+
| DATEDIFF('2010-07-31 23:59:59','2010-08-30') |
+-----------------------------------------+
|                                     -30 |
+-----------------------------------------+
1 row in set
```

3.5.12DATE_ADD(), DATE_SUB()

以下函数执行日期计算:

DATE_ADD(date,INTERVAL expr type), 加法操作

DATE_SUB(date,INTERVAL expr type), 减法操作

参数说明如下:

- date 是一个 DATETIME 或 DATE 值，指定一个日期的开始。expr 是一个表达式，用来指定从起始日期添加或减去的时间间隔值。
- expr 是一个字符串，对于负值的时间间隔，它可以以一个“-”开头。
- type 为关键词，它指示了表达式被解释的方式。
- INTERVAL 关键字和类型修饰符大小写不敏感。

相关的 type 和 expr 参数如下表所示:

type 值	期望的 expr 格式
MICROSECOND	MICROSECONDS
SECOND	SECONDS
MINUTE	MINUTES
HOUR	HOURS
DAY	DAYS
WEEK	WEEKS
MONTH	MONTHS
QUARTER	QUARTERS

type 值	期望的 expr 格式
YEAR	YEARS
SECOND_MICROSECOND	'SECONDS.MICROSECONDS'
MINUTE_MICROSECOND	'MINUTES.MICROSECONDS'
MINUTE_SECOND	'MINUTES:SECONDS'
HOUR_MICROSECOND	'HOURS.MICROSECONDS'
HOUR_SECOND	'HOURS:MINUTES:SECONDS'
HOUR_MINUTE	'HOURS:MINUTES'
DAY_MICROSECOND	'DAYS.MICROSECONDS'
DAY_SECOND	'DAYS HOURS:MINUTES:SECONDS'
DAY_MINUTE	'DAYS HOURS:MINUTES'
DAY_HOUR	'DAYS HOURS'
YEAR_MONTH	'YEARS-MONTHS'

在 expr 的格式中，GBase 8a 允许任何字符作为定界符。表中所显示的是建议的定界字符。如果 date 参数是一个 DATE 值，并且计算的间隔仅仅有 YEAR、MONTH 和 DAY 部分（没有时间部分），那么返回值也是一个 DATE 值。否则返回值是一个 DATETIME 值。

如果表达式另一边是 DATE 或 DATETIME 类型，那么 INTERVAL expr type 允许出现在“+”的任何一边。对于“-”来说，INTERVAL expr type 值只能出现在右边，因为从一个时间间隔减去一个 DATE 或 DATETIME 值没有意义。

示例 1：将“2010-08-30 23:59:59”加 1 秒。

```
gbase> SELECT '2010-08-30 23:59:59' + INTERVAL 1 SECOND FROM t;
+-----------------------------------------+
| '2010-08-30 23:59:59' + INTERVAL 1 SECOND |
+-----------------------------------------+
| 2010-08-31 00:00:00                     |
+-----------------------------------------+
1 row in set
```

示例 2：使用 DATE_ADD()函数，将“2010-08-30 23:59:59”加一秒，执行结果与示例 1 相同。

```
gbase> SELECT DATE_ADD('2010-08-30 23:59:59',INTERVAL 1 SECOND) FROM t;
+-----------------------------------------------+
| DATE_ADD('2010-08-30 23:59:59',INTERVAL 1 SECOND) |
+-----------------------------------------------+
| 2010-08-31 00:00:00                           |
+-----------------------------------------------+
1 row in set
```

示例 3：将“2010-12-31 23:59:59”加一天。

```
gbase> SELECT INTERVAL 1 DAY + '2010-12-31 23:59:59' FROM t;
+--------------------------------------+
| INTERVAL 1 DAY + '2010-12-31 23:59:59' |
+--------------------------------------+
| 2011-01-01 23:59:59                  |
+--------------------------------------+
1 row in set
```

示例 4：使用 DATE_ADD()函数，将“2010-12-31 23:59:59”加一天，执行结果与示例 3 相同。

```
gbase> SELECT DATE_ADD('2010-12-31 23:59:59',INTERVAL 1 DAY) FROM t;
+-----------------------------------------------+
| DATE_ADD('2010-12-31 23:59:59',INTERVAL 1 DAY) |
+-----------------------------------------------+
| 2011-01-01 23:59:59                           |
+-----------------------------------------------+
1 row in set
```

示例 5：DATE_ADD()函数，type 类型为“MINUTE_SECOND”。

```
gbase> SELECT DATE_ADD('2010-12-31 23:59:59', INTERVAL '1:1' MINUTE_SECOND)
FROM t;
+-----------------------------------------------------------+
| DATE_ADD('2010-12-31 23:59:59', INTERVAL '1:1' MINUTE_SECOND) |
+-----------------------------------------------------------+
| 2011-01-01 00:01:00                                       |
+-----------------------------------------------------------+
```

```
1 row in set
```

示例 6：DATE_SUB()函数，type 类型为“DAY_SECOND”。

```
gbase> SELECT DATE_SUB('2010-01-01 00:00:00', INTERVAL '1 1:1:1' DAY_SECOND)
FROM t;
+-----------------------------------------------------------+
| DATE_SUB('2010-01-01 00:00:00', INTERVAL '1 1:1:1' DAY_SECOND) |
+-----------------------------------------------------------+
| 2009-12-30 22:58:59                                       |
+-----------------------------------------------------------+
1 row in set
```

示例 7：DATE_ADD()函数，type 类型为“DAY_HOUR”。

```
gbase> SELECT DATE_ADD('2010-01-01 00:00:00', INTERVAL '-1 10' DAY_HOUR) FROM
t;
+-----------------------------------------------------------+
| DATE_ADD('2010-01-01 00:00:00', INTERVAL '-1 10' DAY_HOUR) |
+-----------------------------------------------------------+
| 2009-12-30 14:00:00                                       |
+-----------------------------------------------------------+
1 row in set
```

示例 8：DATE_ADD()函数，type 类型为“DAY_HOUR”。

因为“-1”是减去一天，所以小时“-10”，也是进行减法，这个操作，取决于最前面的操作符号。下面示例里的写法，等价于示例 7 的计算。

```
gbase> SELECT DATE_ADD('2010-01-01 00:00:00', INTERVAL '-1 10' DAY_HOUR) FROM
t;
+-----------------------------------------------------------+
| DATE_ADD('2010-01-01 00:00:00', INTERVAL '-1 10' DAY_HOUR) |
+-----------------------------------------------------------+
| 2009-12-30 14:00:00                                       |
+-----------------------------------------------------------+
1 row in set
```

示例 9：DATE_SUB()函数，type 类型为“DAY”。

```
gbase> SELECT DATE_SUB('2010-01-02', INTERVAL 31 DAY) FROM t;
+---------------------------------------+
| DATE_SUB('2010-01-02', INTERVAL 31 DAY) |
+---------------------------------------+
| 2009-12-02                            |
+---------------------------------------+
1 row in set
```

示例 10：DATE_ADD()函数，type 类型为“SECOND_MICROSECOND”。

```
gbase> SELECT DATE_ADD('2010-08-31 23:59:59.000002',INTERVAL '1.999999'
SECOND_MICROSECOND) AS DATE_ADD FROM t;
+----------------------------+
| DATE_ADD                   |
+----------------------------+
| 2010-09-01 00:00:01.000001 |
+----------------------------+
1 row in set
```

如果用户指定了一个过短的间隔值（没有包括 type 关键词所期望的所有间隔部分），GBase 8a 假设用户遗漏了间隔值的最左边部分。

例如，如果指定一个 type 为 DAY_SECOND，那么 expr 值被期望包含天、小时、分钟和秒部分。如果用户指定的值像“1:10”这样的格式，GBase 8a 假设天和小时部分被遗漏了，指定的值代表分钟和秒。也就是说，“1:10”DAY_SECOND 等价于“1:10”MINUTE_SECOND。这类似于 GBase 8a 解释 TIME 值为经过的时间而不是一天的时刻。

如果用户从一个日期类型中加或减一个包含时间的值，结果会自动调节然后转换成日期类型。

示例 11：在“2010-01-30”上加 1 天。

```
gbase> SELECT DATE_ADD('2010-08-30', INTERVAL 1 DAY) FROM t;
+--------------------------------------+
| DATE_ADD('2010-08-30', INTERVAL 1 DAY) |
```

```
+--------------------------------------+
| 2010-08-31                           |
+--------------------------------------+
1 row in set
```

示例 12：在日期类型中加 1 个小时，返回结果转换为“日期+时间”类型。

```
gbase> SELECT DATE_ADD('2010-01-01', INTERVAL 1 HOUR) FROM t;
+---------------------------------------+
| DATE_ADD('2010-01-01', INTERVAL 1 HOUR) |
+---------------------------------------+
| 2010-01-01 01:00:00                   |
+---------------------------------------+
1 row in set
```

如果用户使用了不正确的日期，返回结果将是 NULL。

如果用户增加 MONTH、YEAR_MONTH 或 YEAR，并且结果日期的天比新月份的最大天数还大，那么它将被调整到新月份的最大天数。

示例 13：在“2010-01-30”上加 1 月。

```
gbase> SELECT DATE_ADD('2010-01-30', INTERVAL 1 MONTH) FROM t;
+----------------------------------------+
| DATE_ADD('2010-01-30', INTERVAL 1 MONTH) |
+----------------------------------------+
| 2010-02-28                             |
+----------------------------------------+
1 row in set
```

3.5.13DATE_FORMAT(date,format)

依照 FORMAT 字符串格式化 date 值。

下面的格式可被用于 format 字符串中：

格　式	描　　述

格　式	描　　述
%a	星期名的英文缩写形式（Sun...Sat）
%b	月份的英文缩写形式（Jan...DEC）
%c	月份的数字形式（0...12）
%D	有英文后缀的某月的第几天（0th, 1st, 2nd, 3rd...）
%d	月份中的天数，数字形式（00...31）
%e	月份中的天数，数字形式（0...31）
%f	微秒（000000...999999）
%H	小时，24 小时制（00...23）
%h	小时，12 小时制（0,1...12）
%I	小时，12 小时制，个位数字前加 0（01...12）
%i	分钟，数字形式（00...59）
%j	一年中的天数（001...366）
%k	小时，24 小时制（0...23）
%l	小时，12 小时制（1...12）
%M	月份，英文形式全拼（January...December）
%m	月份，数字形式（00...12）
%p	AM 或 PM
%r	时间，12 小时制（HH:MI:SS 后面紧跟 AM 或 PM）
%S	秒（00...59）
%s	秒（00...59）
%T	时间，24 小时（HH:MI:SS）
%U	星期（00...53），星期日是一个星期的第一天
%u	星期（00...53），星期一是一个星期的第一天
%V	星期（01...53），星期日是一个星期的第一天。与“%X”一起使用
%v	星期（01...53），星期一是一个星期的第一天。与“%x”一起使用
%W	星期名的英文全拼形式（Sunday...Saturday）
%w	一星期中的哪一天（0=Sunday...6=Saturday）
%X	以 4 位数字形式反映周所在的年份，星期日周的第一天
%x	以 4 位数字形式反映周所在的年份，星期日周的第一天
%Y	4 位数字形式表达的年份
%y	2 位数字形式表达的年份
%%	一个字母“%”

所有其它的字符不经过解释，直接复制到结果中。

注意，“%”字符要求在格式指定符之前。

示例 1：FORMAT 格式为“%W %M %Y”。

```
gbase> SELECT DATE_FORMAT('2010-10-04 22:23:00', '%W %M %Y') FROM t;
+-----------------------------------------------+
| DATE_FORMAT('2010-10-04 22:23:00', '%W %M %Y') |
+-----------------------------------------------+
| Monday October 2010                           |
+-----------------------------------------------+
1 row in set
```

示例 2：FORMAT 格式为“%H:%i:%s”。

```
gbase> SELECT DATE_FORMAT('2010-10-04 22:23:00', '%H:%i:%s') FROM t;
+-----------------------------------------------+
| DATE_FORMAT('2010-10-04 22:23:00', '%H:%i:%s') |
+-----------------------------------------------+
| 22:23:00                                      |
+-----------------------------------------------+
1 row in set
```

示例 3：FORMAT 格式为“%D %y %a %d %m %b %j”。

```
gbase> SELECT DATE_FORMAT('2010-10-04 22:23:00','%D %y %a %d %m %b %j') FROM
t;
+-------------------------------------------------------+
| DATE_FORMAT('2010-10-04 22:23:00','%D %y %a %d %m %b %j') |
+-------------------------------------------------------+
| 4th 10 Mon 04 10 Oct 277                              |
+-------------------------------------------------------+
1 row in set
```

示例 4：FORMAT 格式为“%H %k %I %r %T %S %w”。

```
gbase> SELECT DATE_FORMAT('2010-10-04 22:23:00','%H %k %I %r %T %S %w') FROM
t;
```

```
+-----------------------------------------------------------+
| DATE_FORMAT('2010-10-04 22:23:00','%H %k %I %r %T %S %w') |
+-----------------------------------------------------------+
| 22 22 10 10:23:00 PM 22:23:00 00 1                        |
+-----------------------------------------------------------+
1 row in set
```

示例 5：FORMAT 格式为“%X %V”。

```
gbase> SELECT DATE_FORMAT('2010-01-01', '%X %V') FROM t;
+-----------------------------------+
| DATE_FORMAT('2010-01-01', '%X %V') |
+-----------------------------------+
| 2009 52                           |
+-----------------------------------+
1 row in set
```

3.5.14DAY(date)

返回 date 是一个月中的第几天，范围为 0 到 31。

示例 1：返回“2010-08-30”是 8 月中的第几天。

```
gbase> SELECT DAY('2010-08-30'),DAYOFMONTH('2010-08-30') FROM t;
+------------------+-------------------------+
| DAY('2010-08-30') | DAYOFMONTH('2010-08-30') |
+------------------+-------------------------+
|                30 |                      30 |
+------------------+-------------------------+
1 row in set
```

3.5.15DAYNAME(date)

返回给出的日期是星期几。

示例 1：返回“2010-08-30”是星期几。

```
gbase> SELECT DAYNAME('2010-08-30') FROM t;
+----------------------+
| DAYNAME('2010-08-30') |
+----------------------+
| Monday                |
+----------------------+
1 row in set
```

3.5.16DAYOFMONTH(date)

返回 date 是一个月中的第几天，范围为 0 到 31。

DAYOFMONTH()等价于 DAY()。

示例 1：返回“2010-08-30”是 8 月中的第几天。

```
gbase> SELECT DAYOFMONTH('2010-08-30') FROM t;
+-------------------------+
| DAYOFMONTH('2010-08-30') |
+-------------------------+
|                      30 |
+-------------------------+
1 row in set
```

示例 2：“2001-11-00”是 11 月的第几天。

```
gbase> SELECT DAYOFMONTH('2001-11-00') FROM t;
+-------------------------+
| DAYOFMONTH('2001-11-00') |
+-------------------------+
|                       0 |
+-------------------------+
1 row in set
```

示例 3：返回“2001-11-10”是 11 月的第几天。

```
gbase> SELECT DAYOFMONTH('2001-11-10') FROM t;
+--------------------------+
| DAYOFMONTH('2001-11-10') |
+--------------------------+
|                       10 |
+--------------------------+
1 row in set
```

3.5.17DAYOFWEEK(date)

返回 date（1 = 周日，2 = 周一，...，7 = 周六)对应的工作日索引。

示例 1：“2010-08-30”是周一，返回对应的工作日索引为 2。

```
gbase> SELECT DAYOFWEEK('2010-08-30') FROM t;
+-------------------------+
| DAYOFWEEK('2010-08-30') |
+-------------------------+
|                       2 |
+-------------------------+
1 row in set
```

3.5.18DAYOFYEAR(date)

返回 date 是一年中的第几天，范围为 1 到 366。

示例 1：返回“2010-08-30”是 2010 年的第几天。

```
gbase> SELECT DAYOFYEAR('2010-08-30') FROM t;
+-------------------------+
| DAYOFYEAR('2010-08-30') |
+-------------------------+
|                     242 |
+-------------------------+
1 row in set
```

示例 2：返回“2000-12-31”是 2000 年的第几天。

```
gbase> SELECT DAYOFYEAR('2000-12-31') FROM t;
+-------------------------+
| DAYOFYEAR('2000-12-31') |
+-------------------------+
|                     366 |
+-------------------------+
1 row in set
```

3.5.19EXTRACT(type FROM date)

EXTRACT()函数使用与 DATE_ADD()或 DATE_SUB()一致的间隔类型，但是它用于指定从日期中提取的部分，而不是进行日期算术运算。

下表为可返回的 type 类型，type 类型可组合使用。

约　定	说　明
日期	date
时间	time
年	year
季度	quarter
月	month
日	day
星期	week
小时	hour
分钟	minute
秒	second
微秒	microsecond

示例 1：返回的结果是日期中的“年”。

```
gbase> SELECT EXTRACT(YEAR FROM '2010-08-30') FROM t;
+--------------------------------+
| EXTRACT(YEAR FROM '2010-08-30') |
```

```
+------------------------------+
|                         2010 |
+------------------------------+
1 row in set
```

示例 2：返回的结果是日期中的“年月”。

```
gbase> SELECT EXTRACT(YEAR_MONTH FROM '2010-08-30 01:02:03') FROM t;
+-----------------------------------------------+
| EXTRACT(YEAR_MONTH FROM '2010-08-30 01:02:03') |
+-----------------------------------------------+
|                                        201008 |
+-----------------------------------------------+
1 row in set
```

示例 3：返回的结果是日期中“日小时分钟”的部分。

```
gbase> SELECT EXTRACT(DAY_MINUTE FROM '2010-08-30 01:02:03') FROM t;
+-----------------------------------------------+
| EXTRACT(DAY_MINUTE FROM '2010-08-30 01:02:03') |
+-----------------------------------------------+
|                                        300102 |
+-----------------------------------------------+
1 row in set
```

示例 4：返回结果是“1230”，表示 1230 微秒。

```
gbase> SELECT EXTRACT(MICROSECOND FROM '2010-08-30 10:30:00.00123') FROM t;
+-------------------------------------------------------+
| EXTRACT(MICROSECOND FROM '2010-08-30 10:30:00.00123') |
+-------------------------------------------------------+
|                                                  1230 |
+-------------------------------------------------------+
1 row in set
```

示例 5：返回结果“36”，表示 2011 年的第 36 周。

```
gbase> SELECT EXTRACT(WEEK FROM '2011-09-05 10:30:00.00123') FROM t;
+-----------------------------------------------+
```

```
| EXTRACT(WEEK FROM '2011-09-05 10:30:00.00123') |
+------------------------------------------------+
|                                             36 |
+------------------------------------------------+
1 row in set
```

3.5.20 FROM_DAYS(N)

返回天数 N 对应的 DATE 值。

示例 1: “734309”对应的 DATE 为“2010-06-21”。

```
gbase> SELECT FROM_DAYS(734309) FROM t;
+------------------+
| FROM_DAYS(734309) |
+------------------+
| 2010-06-21       |
+------------------+
1 row in set
```

3.5.21 FROM_UNIXTIME()

FROM_UNIXTIME(unix_timestamp)

FROM_UNIXTIME(unix_timestamp,FORMAT)

以“YYYY-MM-DDHH:MI:SS”或“YYYYMMDDHHMISS”格式返回一个 unix_timestamp 参数值，返回值的形式取决于它使用在字符串中还是数字中。

如果 FORMAT 已经给出，则返回值的格式依照 FORMAT 字符串的格式。FORMAT 可以包含与 DATE_FORMAT()函数同样的修饰符。

示例 1: 返回“YYYY-MM-DDHH:MI:SS”格式的日期时间值。

```
gbase> SELECT FROM_UNIXTIME(1275996580) FROM t;
+-------------------------+
```

```
| FROM_UNIXTIME(1275996580) |
+---------------------------+
| 2010-06-08 19:29:40       |
+---------------------------+
1 row in set
```

示例 2：返回“YYYYMMDDHHMISS”格式的日期时间值。

```
gbase> SELECT FROM_UNIXTIME(1275996580) + 0 FROM t;
+------------------------------+
| FROM_UNIXTIME(1275996580) + 0 |
+------------------------------+
|          20100608192940.000000 |
+------------------------------+
1 row in set
```

示例 3：FORMAT 为“%Y %D %M %h:%i:%s %x”。

```
gbase> SELECT FROM_UNIXTIME(UNIX_TIMESTAMP(),'%Y %D %M %h:%i:%s %x') FROM t;
+-----------------------------------------------------+
| FROM_UNIXTIME(UNIX_TIMESTAMP(),'%Y %D %M %h:%i:%s %x') |
+-----------------------------------------------------+
| 2013 12th October 11:10:46 2013                      |
+-----------------------------------------------------+
1 row in set
```

3.5.22 GET_FORMAT()

GET_FORMAT(DATE|TIME|DATETIME,EUR'|'USA'|'JIS'|'ISO'|'INTERNAL')

返回一个格式字符串。

这个函数可以与 DATE_FORMAT()函数或 STR_TO_DATE()函数进行组合。

对于参数 DATE，DATETIME 和 TIME，各有五种可能值，共计十五种格式字符串：

函数调用	结　果
GET_FORMAT(DATE,'USA')	'%m.%d.%Y'
GET_FORMAT(DATE,'JIS')	'%Y-%m-%d'
GET_FORMAT(DATE,'ISO')	'%Y-%m-%d'
GET_FORMAT(DATE,'EUR')	'%d.%m.%Y'
GET_FORMAT(DATE,'INTERNAL')	'%Y%m%d'
GET_FORMAT(DATETIME,'USA')	'%Y-%m-%d-%H.%i.%s'
GET_FORMAT(DATETIME,'JIS')	'%Y-%m-%d %H:%i:%s'
GET_FORMAT(DATETIME,'ISO')	'%Y-%m-%d %H:%i:%s'
GET_FORMAT(DATETIME,'EUR')	'%Y-%m-%d-%H.%i.%s'
GET_FORMAT(DATETIME,'INTERNAL')	'%Y%m%d%H%i%s'
GET_FORMAT(TIME,'USA')	'%h:%i:%s %p'
GET_FORMAT(TIME,'JIS')	'%H:%i:%s'
GET_FORMAT(TIME,'ISO')	'%H:%i:%s'
GET_FORMAT(TIME,'EUR')	'%H.%i.%S'
GET_FORMAT(TIME,'INTERNAL')	'%H%i%s'

说明：对于上述表中使用的说明符的作用，请参见“3.5.13 DATE_FORMAT(date,format)”中的表。

示例 1：DATE_FORMAT()与 GET_FORMAT()函数进行组合。

GET_FORMAT(DATE,'EUR')对应输出的格式为“%d.%m.%Y”。

```
gbase> SELECT DATE_FORMAT('2010-08-30',GET_FORMAT(DATE,'EUR')) FROM t;
+-------------------------------------------------+
| DATE_FORMAT('2010-08-30',GET_FORMAT(DATE,'EUR')) |
+-------------------------------------------------+
| 30.08.2010                                      |
+-------------------------------------------------+
1 row in set
```

示例 2：STR_TO_DATE()与 GET_FORMAT()函数进行组合。

GET_FORMAT(DATE,'USA')对应的输出格式为“%m.%d.%Y”。

```
gbase> SELECT STR_TO_DATE('08.30.2010',GET_FORMAT(DATE,'USA')) FROM t;
```

```
+------------------------------------------------+
| STR_TO_DATE('08.30.2010',GET_FORMAT(DATE,'USA')) |
+------------------------------------------------+
| 2010-08-30                                     |
+------------------------------------------------+
1 row in set
```

3.5.23HOUR(time)

返回 time 对应的小时值，范围为 0 到 23。

示例 1：返回“10:05:03”对应的小时值。

```
gbase> SELECT HOUR('10:05:03') FROM t;
+------------------+
| HOUR('10:05:03') |
+------------------+
|               10 |
+------------------+
1 row in set
```

示例 2：time 值范围实际上很大，因此 HOUR 返回值可以比 23 大。

```
gbase> SELECT HOUR('272:59:59') FROM t;
+-------------------+
| HOUR('272:59:59') |
+-------------------+
|               272 |
+-------------------+
1 row in set
```

3.5.24LAST_DAY(date)

返回 date 中当前月对应的最后一天的值。

其中，date 为日期或日期时间类型。

如果参数 date 无效，则返回 NULL。

示例 1：date 值为有效日期，返回 2011 年 8 月份的最后一天。

```
gbase> SELECT LAST_DAY('2011-08-30') FROM t;
+-----------------------+
| LAST_DAY('2011-08-30') |
+-----------------------+
| 2011-08-31            |
+-----------------------+
1 row in set
```

示例 2：date 值为有效日期，返回 2011 年 2 月份的最后一天。

```
gbase> SELECT LAST_DAY('2011-02-05') FROM t;
+-----------------------+
| LAST_DAY('2011-02-05') |
+-----------------------+
| 2011-02-28            |
+-----------------------+
1 row in set
```

示例 3：date 值为有效日期，返回 2011 年 1 月份的最后一天。

```
gbase> SELECT LAST_DAY('2011-01-01 01:01:01') FROM t;
+--------------------------------+
| LAST_DAY('2011-01-01 01:01:01') |
+--------------------------------+
| 2011-01-31                     |
+--------------------------------+
1 row in set
```

示例 4：date 值为无效日期，返回结果为 NULL。

```
gbase> SELECT LAST_DAY('2011-08-32') FROM t;
+-----------------------+
| LAST_DAY('2011-08-32') |
```

```
+----------------------+
| NULL                 |
+----------------------+
1 row in set, 1 warning

gbase> SHOW WARNINGS;
+---------+------+--------------------------------------+
| Level   | Code | Message                              |
+---------+------+--------------------------------------+
| Warning | 1292 | Incorrect datetime value: '2011-08-32' |
+---------+------+--------------------------------------+
1 row in set
```

3.5.25LOCALTIME, LOCALTIME()

LOCALTIME 和 LOCALTIME()等同于 NOW()。

示例 1：使用 LOCALTIME 函数，返回当前“日期+时间”。

```
gbase> SELECT LOCALTIME FROM t;
+---------------------+
| LOCALTIME           |
+---------------------+
| 2013-10-12 11:15:46 |
+---------------------+
1 row in set
```

示例 2：使用 LOCALTIME()函数，返回当前“日期+时间”。

```
gbase> SELECT LOCALTIME() FROM t;
+---------------------+
| LOCALTIME()         |
+---------------------+
| 2013-10-12 11:16:04 |
+---------------------+
1 row in set
```

示例 3：使用 NOW()函数，返回当前“日期+时间”。

```
gbase> SELECT NOW() FROM t;
+---------------------+
| NOW()               |
+---------------------+
| 2013-10-12 11:16:19 |
+---------------------+
1 row in set
```

3.5.26 LOCALTIMESTAMP，LOCALTIMESTAMP()

LOCALTIMESTAMP 和 LOCALTIMESTAMP()等同于 NOW()。

示例 1：使用 LOCALTIMESTAMP 函数，返回当前时间戳。

```
gbase> SELECT LOCALTIMESTAMP FROM t;
+---------------------+
| LOCALTIMESTAMP      |
+---------------------+
| 2013-10-12 11:16:33 |
+---------------------+
1 row in set
```

示例 2：使用 LOCALTIMESTAMP()函数，返回当前时间戳。

```
gbase> SELECT LOCALTIMESTAMP() FROM t;
+---------------------+
| LOCALTIMESTAMP()    |
+---------------------+
| 2013-10-12 11:16:45 |
+---------------------+
1 row in set
```

示例 3：使用 NOW()函数，返回当前“日期+时间”。

```
gbase> SELECT NOW() FROM t;
```

```
+---------------------+
| NOW()               |
+---------------------+
| 2013-10-12 11:17:09 |
+---------------------+
1 row in set
```

3.5.27 MAKEDATE(year,dayofyear)

根据给定的年份值 year 和日期在年中的天数值 dayofyear，返回日期值。

dayofyear 必须大于 0，否则返回 NULL。

示例 1：返回 2011 年第 31 天和第 32 天对应的日期。

```
gbase> SELECT MAKEDATE(2011,31), MAKEDATE(2011,32) FROM t;
+------------------+------------------+
| MAKEDATE(2011,31) | MAKEDATE(2011,32) |
+------------------+------------------+
| 2011-01-31        | 2011-02-01        |
+------------------+------------------+
1 row in set
```

示例 2：返回 2010 年和 2011 年第 365 天对应的日期。

```
gbase> SELECT MAKEDATE(2010,365), MAKEDATE(2011,365) FROM t;
+------------------+------------------+
| MAKEDATE(2010,365) | MAKEDATE(2011,365) |
+------------------+------------------+
| 2010-12-31         | 2011-12-31         |
+------------------+------------------+
1 row in set
```

示例 3：dayofyear 值等于 0，返回 NULL。

```
gbase> SELECT MAKEDATE(2011,0) FROM t;
+------------------+
```

```
| MAKEDATE(2011,0) |
+------------------+
| NULL             |
+------------------+
1 row in set
```

3.5.28 MAKETIME(hour,minute,second)

返回从 hour，minute，second 计算得到的时间值。

示例 1：返回“12,15,30”对应的时分秒的值。

```
gbase> SELECT MAKETIME(12,15,30) FROM t;
+--------------------+
| MAKETIME(12,15,30) |
+--------------------+
| 12:15:30           |
+--------------------+
1 row in set
```

3.5.29 MICROSECOND(expr)

以数字的形式返回 time 或者 datetime 表达式 expr 中的微秒值，数字取值范围是 0 到 999999。

示例 1：返回 time 中的微秒值。

```
gbase> SELECT MICROSECOND('12:00:00.123456') FROM t;
+--------------------------------+
| MICROSECOND('12:00:00.123456') |
+--------------------------------+
|                         123456 |
+--------------------------------+
1 row in set
```

示例 2：返回 datetime 中的微秒值。

```
gbase> SELECT MICROSECOND('2009-12-31 23:59:59.000010') FROM t;
+-----------------------------------------+
| MICROSECOND('2009-12-31 23:59:59.000010') |
+-----------------------------------------+
|                                      10 |
+-----------------------------------------+
1 row in set
```

示例 3：datetime 中的微秒值超过六位，结果返回前六位。

```
gbase> SELECT MICROSECOND('2009-12-31 23:59:59.1234567') FROM t;
+-----------------------------------------+
| MICROSECOND('2009-12-31 23:59:59.1234567') |
+-----------------------------------------+
|                                  123456 |
+-----------------------------------------+
1 row in set, 1 warning

gbase> SHOW WARNINGS;
+---------+------+
| Level   | Code |
+---------+------+
| Warning | 1292 |
+---------+------+
+-------------------------------------------------------------+
| Message                                                     |
+-------------------------------------------------------------+
| Truncated incorrect time value: '2009-12-31 23:59:59.1234567' |
+-------------------------------------------------------------+
1 row in set
```

3.5.30MINUTE(time)

返回 time 对应的分钟值，范围为 0 到 59。

示例 1：time 值对应的分钟值在 0～59 之内。

```
gbase> SELECT MINUTE('01-02-03 10:08:03') FROM t;
+----------------------------+
| MINUTE('01-02-03 10:08:03') |
+----------------------------+
|                          8 |
+----------------------------+
1 row in set
```

示例 2：time 值对应的分钟值大于 59，返回 NULL。

```
gbase> SELECT MINUTE('01-02-03 10:60:03') FROM t;
+----------------------------+
| MINUTE('01-02-03 10:60:03') |
+----------------------------+
|                       NULL |
+----------------------------+
1 row in set, 1 warning

gbase> SHOW WARNINGS;
+---------+------+
| Level   | Code |
+---------+------+
| Warning | 1292 |
+---------+------+
+-------------------------------------------------+
| Message                                         |
+-------------------------------------------------+
| Truncated incorrect time value: '01-02-03 10:60:03' |
+-------------------------------------------------+
1 row in set
```

3.5.31 MONTH(date)

返回 date 对应的月份值，范围为 1～12。

示例 1：date 值对应的月份值在 1 到 12 之间。

```
gbase> SELECT MONTH('2011-08-30') FROM t;
+---------------------+
| MONTH('2011-08-30') |
+---------------------+
|                   8 |
+---------------------+
1 row in set
```

示例 2：date 值对应的月份值大于 12，返回 NULL。

```
gbase> SELECT MONTH('2011-13-30') FROM t;
+---------------------+
| MONTH('2011-13-30') |
+---------------------+
|                NULL |
+---------------------+
1 row in set, 1 warning
```

3.5.32 MONTHNAME(date)

返回 date 中对应的月份，并以英文名称显示。

示例 1：返回 8 月对应的英文名称。

```
gbase> SELECT MONTHNAME('2011-08-30') FROM t;
+------------------------+
| MONTHNAME('2011-08-30') |
+------------------------+
| August                 |
+------------------------+
1 row in set
```

3.5.33 NOW()

以“YYYY-MM-DD HH:MI:SS”或“YYYYMMDDHHMISS”格式返回当前的日期时间值，返回的格式取决于该函数是用于字符串还是数字上下文中。

示例 1：返回“YYYY-MM-DD HH:MI:SS”格式的日期时间值。

```
gbase> SELECT NOW() FROM t;
+---------------------+
| NOW()               |
+---------------------+
| 2013-10-12 11:25:32 |
+---------------------+
1 row in set
```

示例 2：返回“YYYYMMDDHHMISS”格式的日期时间值。

```
gbase> SELECT NOW() + 0 FROM t;
+-----------------------+
| NOW() + 0             |
+-----------------------+
| 20131012112545.000000 |
+-----------------------+
1 row in set
```

3.5.34 PERIOD_ADD(P,N)

增加 N 个月到周期 P 中，返回值格式为 YYYYMM。

其中，P 的格式为 YYMM 或 YYYYMM。

注意，参数 P 不是日期类型的数值。

示例 1：增加 2 个月到“1108”中。

```
gbase> SELECT PERIOD_ADD(1108,2) FROM t;
+--------------------+
| PERIOD_ADD(1108,2) |
+--------------------+
|             201110 |
```

```
+--------------------+
1 row in set
```

示例 2：增加 2 个月到“201108”中。

```
gbase> SELECT PERIOD_ADD(201108,2) FROM t;
+----------------------+
| PERIOD_ADD(201108,2) |
+----------------------+
|               201110 |
+----------------------+
1 row in set
```

3.5.35PERIOD_DIFF(P1,P2)

返回 P1 和 P2 之间的月份数。

P1 和 P2 的格式以 YYMM 或 YYYYMM 指定。

注意，参数 P1 和 P2 不是日期值。

示例 1：返回“9802”和“199703”之间相隔的月份数。

```
gbase> SELECT PERIOD_DIFF(9802,199703) FROM t;
+--------------------------+
| PERIOD_DIFF(9802,199703) |
+--------------------------+
|                       11 |
+--------------------------+
1 row in set
```

3.5.36QUARTER(date)

返回 date 所代表的时间在年中的季度数，范围为 1～4。

示例 1：返回“10-08-01”在 2010 年中是第几季度。

```
gbase> SELECT QUARTER('10-08-01') FROM t;
+--------------------+
| QUARTER('10-08-01') |
+--------------------+
|                  3 |
+--------------------+
1 row in set
```

3.5.37 SECOND(time)

返回 time 对应的秒数，范围为 0～59。

示例 1：返回“10:05:03”对应的秒数。

```
gbase> SELECT SECOND('10:05:03') FROM t;
+-------------------+
| SECOND('10:05:03') |
+-------------------+
|                 3 |
+-------------------+
1 row in set
```

3.5.38 SEC_TO_TIME(seconds)

以“HH:MI:SS”或“HHMISS”格式返回参数 seconds 被转换为时分秒后的值，返回值的形式取决于该函数使用于字符串还是数字上下文。

示例 1：将“2378”转换为时分秒后，以“HH:MI:SS”格式返回该值。

```
gbase> SELECT SEC_TO_TIME(2378) FROM t;
+------------------+
| SEC_TO_TIME(2378) |
+------------------+
| 00:39:38         |
+------------------+
```

```
1 row in set
```

示例 2：将“2378”转换为时分秒后，以“HHMISS”格式返回该值。

```
gbase> SELECT SEC_TO_TIME(2378) + 0 FROM t;
+-----------------------+
| SEC_TO_TIME(2378) + 0 |
+-----------------------+
|             3938.000000 |
+-----------------------+
1 row in set
```

3.5.39 STR_TO_DATE(str,format)

STR_TO_DATE()是 DATE_FORMAT()函数的反函数。

它获得字符串 str 和一个格式化字符串 format。

如果格式字符串包含日期和时间部分，则 STR_TO_DATE()返回一个 DATETIME，否则当只包含日期或时间部分时，返回 DATE 或 TIME 值。

包含在 str 中的日期,time 或者 datetime 值应该按照 format 的格式给定。对于 format 可用的详细形式，参考“3.5.13 DATE_FORMAT(date,format)”中的表格。

如果 str 包含一个非法日期时间或者 datetime,STR_TO_DATE()返回系统当前日期。一个非法值也会给出一个警告。

示例 1：format 的值为“%d.%m.%Y %H.%i”。

```
gbase> SELECT STR_TO_DATE('30.08.2010 09.20', '%d.%m.%Y %H.%i') FROM t;
+-------------------------------------------------+
| STR_TO_DATE('30.08.2010 09.20', '%d.%m.%Y %H.%i') |
+-------------------------------------------------+
| 2010-08-30 09:20:00                             |
+-------------------------------------------------+
1 row in set
```

示例 2：将“10arp”以“%carp”格式返回。

```
gbase> SELECT STR_TO_DATE('10arp', '%carp') FROM t;
+-----------------------------+
| STR_TO_DATE('10arp', '%carp') |
+-----------------------------+
| 2013-10-01                  |
+-----------------------------+
1 row in set
```

示例 3：format 的值为“%Y-%m-%d %H:%i:%s”。

```
gbase> SELECT STR_TO_DATE('2010-15-08 00:00:00','%Y-%m-%d %H:%i:%s') FROM t;
+-----------------------------------------------------------+
| STR_TO_DATE('2010-15-08 00:00:00','%Y-%m-%d %H:%i:%s') |
+-----------------------------------------------------------+
| NULL                                                      |
+-----------------------------------------------------------+
1 row in set
```

示例 4：如果 str 取值为 0，则返回系统当前年，当前月，当前日为 1 的日期组合。

```
gbase> SELECT STR_TO_DATE('00/00/0000', '%m/%d/%Y') FROM t;
+-------------------------------------+
| STR_TO_DATE('00/00/0000', '%m/%d/%Y') |
+-------------------------------------+
| 2013-10-01                          |
+-------------------------------------+
1 row in set
```

示例 5：format 的值为“%m/%d/%Y”。

```
gbase> SELECT STR_TO_DATE('08/30/2011', '%m/%d/%Y') FROM t;
+-------------------------------------+
| STR_TO_DATE('08/30/2011', '%m/%d/%Y') |
+-------------------------------------+
```

```
| 2011-08-30                          |
+-------------------------------------+
1 row in set
```

示例 6：注意不能使用格式“%X%V”将一个“year-week”字符串转化为一个日期，原因是当一个星期跨越一个月份界限时，一个年和星期的组合不能标示一个唯一的年和月份。若要将“year-week”转化为一个日期，则也应指定具体工作日。

```
gbase> SELECT str_to_date('201135 Monday', '%X%V %W') FROM t;
+---------------------------------------+
| str_to_date('201135 Monday', '%X%V %W') |
+---------------------------------------+
| 2011-08-29                            |
+---------------------------------------+
1 row in set
```

示例 7：不存在的日期返回系统当前日期。

```
gbase> SELECT str_to_date('201135', '%X%V %W') FROM t;
+---------------------------------+
| str_to_date('201135', '%X%V %W') |
+---------------------------------+
| 2013-10-01                      |
+---------------------------------+
1 row in set
```

3.5.40 SUBDATE()

SUBDATE(date,INTERVAL expr type)

SUBDATE(expr,days)

当调用的第二个参数带有 INTERVA()时，SUBDATE()等同于 DATE_SUB()。具体信息请参见“3.5.12 DATE_ADD(), DATE_SUB”。

expr 是一个 date 或 datetime 表达式，days 用于减 expr 的天数。

示例 1：使用 DATE_SUB 函数，将“2011-01-02”减去 31 天。

```
gbase> SELECT DATE_SUB('2011-01-02', INTERVAL 31 DAY) FROM t;
+---------------------------------------+
| DATE_SUB('2011-01-02', INTERVAL 31 DAY) |
+---------------------------------------+
| 2010-12-02                            |
+---------------------------------------+
1 row in set
```

示例 2：使用 SUBDATE 函数，将“2011-01-02”减去 31 天。

```
gbase> SELECT SUBDATE('2011-01-02', INTERVAL 31 DAY) FROM t;
+---------------------------------------+
| SUBDATE('2011-01-02', INTERVAL 31 DAY) |
+---------------------------------------+
| 2010-12-02                            |
+---------------------------------------+
1 row in set
```

示例 3：使用 SUBDATE 函数，将“2011-01-02 12:00:00”减去 31 天。

```
gbase> SELECT SUBDATE('2011-01-02 12:00:00', 31) FROM t;
+-----------------------------------+
| SUBDATE('2011-01-02 12:00:00', 31) |
+-----------------------------------+
| 2010-12-02 12:00:00               |
+-----------------------------------+
1 row in set
```

3.5.41 SUBTIME(expr,expr2)

SUBTIME()将 expr2 从 expr 中减去并返回结果。

expr 是一个 time 或者 datetime 表达式，expr2 是一个时间表达式。

示例 1：返回“2009-12-31 23:59:59.999999”减去“1 1:1:1.000002”的值。

```
gbase> SELECT SUBTIME('2009-12-31 23:59:59.999999','1 1:1:1.000002') FROM t;
+----------------------------------------------------+
| SUBTIME('2009-12-31 23:59:59.999999','1 1:1:1.000002') |
+----------------------------------------------------+
| 2009-12-30 22:58:58.999997                         |
+----------------------------------------------------+
1 row in set
```

示例 2：返回“01:00:00.999999”减去“02:00:00.999998”的值。

```
gbase> SELECT SUBTIME('01:00:00.999999', '02:00:00.999998') FROM t;
+--------------------------------------------+
| SUBTIME('01:00:00.999999', '02:00:00.999998') |
+--------------------------------------------+
| -00:59:59.999999                           |
+--------------------------------------------+
1 row in set
```

3.5.42 SYSDATE()

以“YYYY-MM-DD HH:MI:SS”或“YYYYMMDDHHMISS”格式返回当前的日期时间值，返回的格式取决于该函数是用于字符串还是数字上下文中。

示例 1：使用 SYSDATE()函数返回当前的“日期+时间”。

```
gbase> SELECT SYSDATE() FROM t;
+---------------------+
| SYSDATE()           |
+---------------------+
| 2013-10-12 13:39:15 |
+---------------------+
1 row in set
```

示例 2：SYSDATE()函数与 NOW()函数的区别。

```
gbase> DROP TABLE IF EXISTS t1;
Query OK, 0 rows affected

gbase> CREATE TABLE t1(i int);
Query OK, 0 rows affected

gbase> INSERT INTO t1 VALUES(0);
Query OK, 1 row affected

gbase> SELECT s,NOW(),SYSDATE() FROM (SELECT SLEEP(5) AS s FROM t1) tt;

+---+---------------------+---------------------+
| s | NOW()               | SYSDATE()           |
+---+---------------------+---------------------+
| 0 | 2014-02-17 11:24:05 | 2014-02-17 11:24:10 |
+---+---------------------+---------------------+
1 row in set
```

从示例可以看出，NOW()函数的时间是 SQL 开始执行的时间，而 SYSDATE()的时间是执行到该函数的时间。NOW()函数和 SYSDATE()函数都可以返回当前的日期时间值，请根据实际场景进行选择。

3.5.43 TIME(expr)

将时间部分从 time 或者 datetime 表达式 expr 中取出来，并按照字符串格式返回。

示例 1：以字符串格式返回“2011-08-30 01:02:03”中的时间值。

```
gbase> SELECT TIME('2011-08-30 01:02:03') FROM t;
+----------------------------+
| TIME('2011-08-30 01:02:03') |
+----------------------------+
| 01:02:03                   |
```

```
+----------------------------+
1 row in set
```

示例 2：以字符串格式返回“2011-08-30 01:02:03.000123”中的时间值。

```
gbase> SELECT TIME('2011-08-30 01:02:03.000123') FROM t;
+-----------------------------------+
| TIME('2011-08-30 01:02:03.000123') |
+-----------------------------------+
| 01:02:03.000123                   |
+-----------------------------------+
1 row in set
```

3.5.44 TIMEDIFF(expr,expr2)

TIMEDIFF()返回开始时间 expr 和结束时间 expr2 之间的间隔时间。

expr 和 expr2 是 time 或者 datetime 表达式，但是两个表达式必须是同种类型。

示例 1：expr 和 expr2 都是 time 表达式，返回“2000:01:01 00:00:00”和“2000:01:01 00:00:00.000001”之间的间隔时间。

```
gbase> SELECT TIMEDIFF('2000:01:01 00:00:00','2000:01:01 00:00:00.000001')
FROM t;
+-------------------------------------------------------------+
| TIMEDIFF('2000:01:01 00:00:00','2000:01:01 00:00:00.000001') |
+-------------------------------------------------------------+
| -00:00:00.000001                                            |
+-------------------------------------------------------------+
1 row in set
```

示例 2：expr 和 expr2 都是 datetime 表达式，返回“2011-08-31 23:59:59.000001”和“2011-08-30 01:01:01.000002”之间的间隔时间。

```
gbase> SELECT TIMEDIFF('2011-08-31 23:59:59.000001','2011-08-30
01:01:01.000002') FROM t;
```

```
+-----------------------------------------------------------+
| TIMEDIFF('2011-08-31 23:59:59.000001','2011-08-30 01:01:01.000002') |
+-----------------------------------------------------------+
| 46:58:57.999999                                           |
+-----------------------------------------------------------+
1 row in set
```

3.5.45 TIMESTAMP

TIMESTAMP(expr),TIMESTAMP(expr,expr2)

TIMESTAMP(expr)，按照 datetime 值返回日期或者 datetime 表达式 expr。

TIMESTAMP(expr,expr2)，将时间表达式 expr2 加到时间表达式 expr 上，然后返回一个 datetime 值。

示例 1：使用 TIMESTAMP(expr)函数，返回“2011-08-30”对应的 datetime 值。

```
gbase> SELECT TIMESTAMP('2011-08-30') FROM t;
+------------------------+
| TIMESTAMP('2011-08-30') |
+------------------------+
| 2011-08-30 00:00:00     |
+------------------------+
1 row in set
```

示例 2:使用 TIMESTAMP(expr,expr2)函数,将“12:00:00”加到“2011-12-31 12:00:00”，并返回其对应的 datetime 值。

```
gbase> SELECT TIMESTAMP('2011-12-31 12:00:00','12:00:00') FROM t;
+---------------------------------------+
| TIMESTAMP('2011-12-31 12:00:00','12:00:00') |
+---------------------------------------+
| 2012-01-01 00:00:00                   |
+---------------------------------------+
1 row in set
```

3.5.46 TIMESTAMPADD

TIMESTAMPADD(interval,int_expr,datetime_expr)

将整数表达式 int_expr 加到 date 或者 datetime 表达式 datetime_expr 上。

int_expr 的单位由 interval 参数给定，应该是下列值之一：

FRAC_SECOND，SECOND，MINUTE，HOUR，DAY，WEEK，MONTH，QUARTER，或 YEAR。

interval 值可以使用上面的关键词指定,或者使用前缀 SQL_TSI_。例如 DAY 或 SQL_TSI_DAY，或者两者都可以。

示例 1：将“1”分钟加到“2011-01-02”上。

```
gbase> SELECT TIMESTAMPADD(MINUTE,1,'2011-01-02') FROM t;
+-----------------------------------+
| TIMESTAMPADD(MINUTE,1,'2011-01-02') |
+-----------------------------------+
| 2011-01-02 00:01:00               |
+-----------------------------------+
1 row in set
```

示例 2：将“1”周加到“2011-01-02”上。

```
gbase> SELECT TIMESTAMPADD(WEEK,1,'2011-01-02') FROM t;
+----------------------------------+
| TIMESTAMPADD(WEEK,1,'2011-01-02') |
+----------------------------------+
| 2011-01-09                       |
+----------------------------------+
1 row in set
```

3.5.47 TIMESTAMPDIFF

TIMESTAMPDIFF(inteval,datetime_expr1,datetime_expr2)

按照整数返回 date 或者 datetime 表达式 datetime_expr1 和 datetime_expr2 之间的差距，参数由 inteval 选项给定。

合法的 INTeval 值与 TIMESTAMPADD()函数描述相同。

示例 1：返回“2011-02-01”和“2011-05-01”之间相差的月份数。

```
gbase> SELECT TIMESTAMPDIFF(MONTH,'2011-02-01','2011-05-01') FROM t;
+-----------------------------------------------+
| TIMESTAMPDIFF(MONTH,'2011-02-01','2011-05-01') |
+-----------------------------------------------+
|                                             3 |
+-----------------------------------------------+
1 row in set
```

示例 2：返回“2011-05-01”和“2001-01-01”之间相差的年份数。

```
gbase> SELECT TIMESTAMPDIFF(YEAR,'2011-05-01','2001-01-01') FROM t;
+-----------------------------------------------+
| TIMESTAMPDIFF(YEAR,'2011-05-01','2001-01-01') |
+-----------------------------------------------+
|                                           -10 |
+-----------------------------------------------+
1 row in set
```

3.5.48 TIME_FORMAT(time,format)

此函数的作用类似 DATE_FORMAT()函数，但是 format 字符串仅处理小时，分钟，秒等格式。其它的错误格式会产生一个 NULL 值或者 0。

如果 time 值包含一个大于 23 的小时部分，%H 和%k 小时格式会产生一个大于 0～23 范围的值。其余小时格式产生的值都会用 12 取模。

示例 1：“100:00:00”包含一个大于 23 的小时部分，“%H”和”%k”返回“100”。“%h”，“%I”，“%l”格式产生的值为“100 MOD 12”。

```
gbase> SELECT TIME_FORMAT('100:00:00', '%H %k %h %I %l') FROM t;
+-----------------------------------------+
| TIME_FORMAT('100:00:00', '%H %k %h %I %l') |
+-----------------------------------------+
| 100 100 04 04 4                         |
+-----------------------------------------+
1 row in set
```

3.5.49 TIME_TO_SEC(time)

返回参数 time 对应的秒数。

示例 1：返回“22:23:00”对应的秒数。

```
gbase> SELECT TIME_TO_SEC('22:23:00') FROM t;
+-------------------------+
| TIME_TO_SEC('22:23:00') |
+-------------------------+
|                   80580 |
+-------------------------+
1 row in set
```

示例 2：返回“00:39:38”对应的秒数。

```
gbase> SELECT TIME_TO_SEC('00:39:38') FROM t;
+-------------------------+
| TIME_TO_SEC('00:39:38') |
+-------------------------+
|                    2378 |
+-------------------------+
1 row in set
```

3.5.50 TO_DATE(string,format)

将字符串 string 格式化成 format 类型的日期。

格式化参数	含　义
- , . : / 空格	一般作为参数 string 的分隔符号使用，不论使用何种分隔符号，格式化输出的格式都是以“-” 作为分隔符的。因为 GBase 8a 的日期格式就是 ISO 的格式。 注意：string 和 format 两个参数必须使用一致的分隔符。
YYYY/YY	YYYY 对应 1～4 位数字，YY 对应 1～2 位数字。 注意：YYYY 或 YY 的值不能为空。 ● 如果FORMAT参数没有指定YYYY或者YY，默认为本年，与NOW()返回的年一致。 ● 如果string和format都是“YY-MM-DD”格式，则返回结果，年补足4位，年的前两位是NOW()函数中年的前2位数值。 ● 如果string是“YY-MM-DD”格式，而format是“YYYY-MM-DD”格式，则返回结果，年也补足4位，但是年的前两位是用“00”补足。 年的范围：0～9999。
MM	如果 string 参数不包含月 MM，则 format 参数默认返回 NOW()函数中月对应的数值。
DD	如果 string 参数不包含日 DD，则 format 参数默认返回为 1 日。
DDD	一年中的第 N 天。参数 string 必须给定年和天数，foramt 参数使用 YYYY DDD 的形式，系统便能计算出对应的日期。
HH、HH12、HH24	格式化小时。 如果 string 参数中小时部分超过 12，则 format 参数中的 HH，必须使用 HH24，否则会报错。
AM	上午的简写，按照上午的 12 小时区间，格式化小时。 注意：不要使用 HH12 或者 HH24 格式， string 和 format 两个参数必须同时指定 AM 或者 PM。 如果 string 和 format 两个参数指定的 AM 或者 PM 不一致，则以 string 参数指定的是 AM 还是 PM，进行格式化输出。
PM	下午的简写，按照下午的 12 小时区间，格式化小时。 注意：不要使用 HH12 或者 HH24 格式， string 和 format 两个参数必须同时指定 AM 或者 PM。 如果 string 和 format 两个参数指定的 AM 或者 PM 不一致，则以 string 参数指定的是 AM 还是 PM，进行格式化输出。
MI	分钟。

格式化参数	含　义
	如果 string 和 foramt 参数中省略 MI，则返回分钟为 0。
SS	秒。 如果 string 和 foramt 参数中省略 SS，则返回秒为 0。

示例 1：不论 string 和 format 使用何种分隔符，格式化输出的格式都是以“-”作为分隔符。

```
gbase> SELECT TO_DATE('2011-11-15','YYYY-MM-DD') FROM t;
+-----------------------------------+
| TO_DATE('2011-11-15','YYYY-MM-DD') |
+-----------------------------------+
| 2011-11-15                        |
+-----------------------------------+
1 row in set

gbase> SELECT TO_DATE('2011/11/15','YYYY/MM/DD') FROM t;
+-----------------------------------+
| TO_DATE('2011/11/15','YYYY/MM/DD') |
+-----------------------------------+
| 2011-11-15                        |
+-----------------------------------+
1 row in set

gbase> SELECT TO_DATE('2011,11,15','YYYY,MM,DD') FROM t;
+-----------------------------------+
| TO_DATE('2011,11,15','YYYY,MM,DD') |
+-----------------------------------+
| 2011-11-15                        |
+-----------------------------------+
1 row in set

gbase> SELECT TO_DATE('2011:11:15','YYYY:MM:DD') FROM t;
+-----------------------------------+
| TO_DATE('2011:11:15','YYYY:MM:DD') |
+-----------------------------------+
```

```
| 2011-11-15                          |
+-------------------------------------+
1 row in set

gbase> SELECT TO_DATE('2011.11.15','YYYY.MM.DD') FROM t;
+-------------------------------------+
| TO_DATE('2011.11.15','YYYY.MM.DD') |
+-------------------------------------+
| 2011-11-15                          |
+-------------------------------------+
1 row in set

gbase> SELECT TO_DATE('2011 11 14','YYYY MM DD') FROM t;
+-------------------------------------+
| TO_DATE('2011 11 14','YYYY MM DD') |
+-------------------------------------+
| 2011-11-14                          |
+-------------------------------------+
1 row in set
```

示例 2：如果 string 中没有指定年，则返回本年，与 NOW()返回的年一致。

```
gbase> SELECT TO_DATE('11-15','MM-DD') FROM t;
+-------------------------+
| TO_DATE('11-15','MM-DD') |
+-------------------------+
| 2013-11-15              |
+-------------------------+
1 row in set
```

如果 string 和 format 都是“YY-MM-DD”格式，则返回结果中的“年”补足 4 位。

```
gbase> SELECT TO_DATE('11-11-15','YY-MM-DD') FROM t;
+-------------------------------+
| TO_DATE('11-11-15','YY-MM-DD') |
+-------------------------------+
| 2011-11-15                    |
```

```
+------------------------------+
1 row in set
```

如果 string 是“YY-MM-DD”格式，而 format 是“YYYY-MM-DD”格式，返回结果中的年补足 4 位。

```
gbase> SELECT TO_DATE('11-11-15','YYYY-MM-DD') FROM t;
+---------------------------------+
| TO_DATE('11-11-15','YYYY-MM-DD') |
+---------------------------------+
| 0011-11-15                      |
+---------------------------------+
1 row in set
```

示例 3：如果 string 参数不包含月 MM，则 format 参数默认返回 NOW()函数中月对应的数值。

```
gbase> SELECT TO_DATE('2011-11-15','YYYY-MM-DD') FROM t;
+---------------------------------+
| TO_DATE('2011-11-15','YYYY-MM-DD') |
+---------------------------------+
| 2011-11-15                      |
+---------------------------------+
1 row in set

gbase> SELECT TO_DATE('2011-15','YYYY-DD') FROM t;
+----------------------------+
| TO_DATE('2011-15','YYYY-DD') |
+----------------------------+
| 2011-10-15                 |
+----------------------------+
1 row in set
```

示例 4：如果 string 参数不包含日 DD，则 format 参数默认返回 1。

```
gbase> SELECT TO_DATE('2011-11-15','YYYY-MM-DD') FROM t;
+---------------------------------+
| TO_DATE('2011-11-15','YYYY-MM-DD') |
```

```
+----------------------------------+
| 2011-11-15                       |
+----------------------------------+
1 row in set
```

```
gbase> SELECT TO_DATE('2011-11','YYYY-MM') FROM t;
+-----------------------------+
| TO_DATE('2011-11','YYYY-MM') |
+-----------------------------+
| 2011-11-01                  |
+-----------------------------+
1 row in set
```

示例 5：返回一年中的第 N 天对应的日期。

```
gbase> SELECT TO_DATE('2011  318','YYYY DDD') FROM t;
+--------------------------------+
| TO_DATE('2011  318','YYYY DDD') |
+--------------------------------+
| 2011-11-14                     |
+--------------------------------+
1 row in set
```

```
gbase> SELECT TO_DATE('2010  55','YYYY DDD') FROM t;
+--------------------------------+
| TO_DATE('2010  55','YYYY DDD') |
+--------------------------------+
| 2010-02-24                     |
+--------------------------------+
1 row in set
```

```
gbase> SELECT TO_DATE('2000 366','YYYY DDD') FROM t;
+--------------------------------+
| TO_DATE('2000 366','YYYY DDD') |
+--------------------------------+
| 2000-12-31                     |
+--------------------------------+
```

```
1 row in set
```

示例 6：返回 12 小时制和 24 小时制的时间。

```
gbase> SELECT TO_DATE('2011-11-15 12:15:10','YYYY-MM-DD HH:MI:SS') FROM t;
+-----------------------------------------------------+
| TO_DATE('2011-11-15 12:15:10','YYYY-MM-DD HH:MI:SS') |
+-----------------------------------------------------+
| 2011-11-15 12:15:10                                 |
+-----------------------------------------------------+
1 row in set

gbase> SELECT TO_DATE('2011-11-15 12:15:10','YYYY-MM-DD HH12:MI:SS') FROM t;
+-----------------------------------------------------+
| TO_DATE('2011-11-15 12:15:10','YYYY-MM-DD HH12:MI:SS') |
+-----------------------------------------------------+
| 2011-11-15 12:15:10                                 |
+-----------------------------------------------------+
1 row in set

gbase> SELECT TO_DATE('2011-11-15 12:15:10','YYYY-MM-DD HH24:MI:SS') FROM t;
+-----------------------------------------------------+
| TO_DATE('2011-11-15 12:15:10','YYYY-MM-DD HH24:MI:SS') |
+-----------------------------------------------------+
| 2011-11-15 12:15:10                                 |
+-----------------------------------------------------+
1 row in set
```

小时部分超过 12，必须使用 HH24，否则报错。

```
gbase> SELECT TO_DATE('2011-11-14 13:00:00','YYYY-MM-DD HH:MI:SS') FROM t;
ERROR 1683 (HY000): incorrect parameter in function to_date.
```

小时部分超过 12，使用 HH24 后，顺利格式化输出。

```
gbase> SELECT TO_DATE('2011-11-14 13:00:00','YYYY-MM-DD HH24:MI:SS') FROM t;
+-----------------------------------------------------+
| TO_DATE('2011-11-14 13:00:00','YYYY-MM-DD HH24:MI:SS') |
```

```
+---------------------------------------------------+
| 2011-11-14 13:00:00                               |
+---------------------------------------------------+
1 row in set
```

下面的示例，用来说明 GBase 8a 的 TO_DATE()函数灵活性。

```
gbase> SELECT TO_DATE('2000 1111','YYYY MISS') FROM t;
+--------------------------------+
| TO_DATE('2000 1111','YYYY MISS') |
+--------------------------------+
| 2000-10-01 00:11:11            |
+--------------------------------+
1 row in set
```

示例 7：string 和 format 两个参数同时指定 AM 或者 PM。

```
gbase> SELECT TO_DATE('2011-11-15 8:15:20 AM','YYYY-MM-DD HH:MI:SS AM') FROM
t;
+-----------------------------------------------------+
| TO_DATE('2011-11-15 8:15:20 AM','YYYY-MM-DD HH:MI:SS AM') |
+-----------------------------------------------------+
| 2011-11-15 08:15:20                                 |
+-----------------------------------------------------+
1 row in set

gbase> SELECT TO_DATE('2011-11-15 8:15:20 PM','YYYY-MM-DD HH:MI:SS PM') FROM
t;
+-----------------------------------------------------+
| TO_DATE('2011-11-15 8:15:20 PM','YYYY-MM-DD HH:MI:SS PM') |
+-----------------------------------------------------+
| 2011-11-15 20:15:20                                 |
+-----------------------------------------------------+
1 row in set
```

如果 string 和 format 两个参数指定的 AM 或者 PM 不一致，则以 string 参数指定的是 AM 还是 PM，进行格式化输出。

```
gbase> SELECT TO_DATE('2011-11-15 8:15:20 AM','YYYY-MM-DD HH:MI:SS PM') FROM
t;
+-----------------------------------------------------------+
| TO_DATE('2011-11-15 8:15:20 AM','YYYY-MM-DD HH:MI:SS PM') |
+-----------------------------------------------------------+
| 2011-11-15 08:15:20                                       |
+-----------------------------------------------------------+
1 row in set

gbase> SELECT TO_DATE('2011-11-15 8:15:20PM','YYYY-MM-DD HH:MI:SS AM') FROM
t;
+-----------------------------------------------------------+
| TO_DATE('2011-11-15 8:15:20PM','YYYY-MM-DD HH:MI:SS AM') |
+-----------------------------------------------------------+
| 2011-11-15 20:15:20                                       |
+-----------------------------------------------------------+
1 row in set
```

示例 8：比较 NOW()和 TO_DATE(TO_CHAR(NOW(),'YYYY-MM-DD HH:MI:SS'),'YYYY-MM-DD HH:MI:SS')。

```
gbase> SELECT NOW(),TO_DATE(TO_CHAR(NOW(),'YYYY-MM-DD HH:MI:SS'),'YYYY-MM-DD
HH:MI:SS') AS f_format FROM t;
+---------------------+---------------------+
| NOW()               | f_format            |
+---------------------+---------------------+
| 2013-10-12 13:56:45 | 2013-10-12 01:56:45 |
+---------------------+---------------------+
1 row in set
```

TO_CHAR()省略掉分钟。

```
gbase> SELECT NOW(),TO_DATE(TO_CHAR(NOW(),'YYYY-MM-DD HH:SS'),'YYYY-MM-DD
HH:SS') AS f_format FROM t;
+---------------------+---------------------+
| NOW()               | f_format            |
+---------------------+---------------------+
| 2013-10-12 13:57:06 | 2013-10-12 01:00:06 |
```

```
+--------------------+--------------------+
1 row in set
```

示例 9：比较 NOW()和 TO_DATE(TO_CHAR(NOW(),'YYYY-MM-DD HH:MI:SS'),'YYYY-MM-DD HH:MI:SS')。

```
gbase> SELECT NOW(),TO_DATE(TO_CHAR(NOW(),'YYYY-MM-DD HH:MI:SS'),'YYYY-MM-DD HH:MI:SS') AS f_format FROM t;
+--------------------+--------------------+
| NOW()              | f_format           |
+--------------------+--------------------+
| 2013-10-12 13:57:37 | 2013-10-12 01:57:37 |
+--------------------+--------------------+
1 row in set
```

TO_CHAR()省略掉秒。

```
gbase> SELECT NOW(),TO_DATE(TO_CHAR(NOW(),'YYYY-MM-DD HH:MI'),'YYYY-MM-DD HH:MI') AS f_format FROM t;
+--------------------+--------------------+
| NOW()              | f_format           |
+--------------------+--------------------+
| 2013-10-12 13:57:52 | 2013-10-12 01:57:00 |
+--------------------+--------------------+
1 row in set
```

3.5.51 TO_DAYS(date)

返回日期 date 对应的天数（从年份 0 开始的天数）

示例 1：返回“950501”对应的天数。

```
gbase> SELECT TO_DAYS(950501) FROM t;
+-----------------+
| TO_DAYS(950501) |
+-----------------+
|          728779 |
```

```
+-----------------+
1 row in set
```

示例 2：返回“2010-08-30”对应的天数。

```
gbase> SELECT TO_DAYS('2010-08-30') FROM t;
+-----------------------+
| TO_DAYS('2010-08-30') |
+-----------------------+
|                734379 |
+-----------------------+
1 row in set
```

TO_DAYS()不用于阳历出现(1582)前的值，原因是当日历改变时，遗失的日期不会被考虑在内。

GBase 8a 使用日期和时间类型中的规则转化两位日期中的年值到四位。

示例 3： “2011-08-30”和“11-08-30”表示同一个日期。

```
gbase> SELECT TO_DAYS('2011-08-30'), TO_DAYS('11-08-30') FROM t;
+-----------------------+--------------------+
| TO_DAYS('2011-08-30') | TO_DAYS('11-08-30') |
+-----------------------+--------------------+
|                734744 |             734744 |
+-----------------------+--------------------+
1 row in set
```

示例 4:对于 1582 年之前的日期(或许在其它地区为下一年),结果是不可靠的。

```
gbase> SELECT TO_DAYS('1581-08-30') FROM t;
+-----------------------+
| TO_DAYS('1581-08-30') |
+-----------------------+
|                577690 |
+-----------------------+
1 row in set
```

3.5.52 TRUNC(date/datetime[,format])

TRUNC 函数返回以指定元素格式截去一部分的日期值。

date/datetime 为必选参数，表示输入的一个日期值

format 为可选参数，表示日期格式，用以指定的元素格式来截去输入的日期值。如果省略此参数，则由最近的日期截去。

format 支持的类型如下表所示。

参　数	说　明
year	返回当年第一天
yyyy	返回当年第一天
month	返回当月第一天
mm	返回当月第一天
dd	返回当天的日期
hh	得到当天凌晨 0 点 0 分 0 秒的日期
mi	得到当天凌晨 0 点 0 分 0 秒的日期

示例 1：返回当年的第一天。

```
gbase> SELECT TRUNC(current_date,'year') FROM t;
+--------------------------+
| TRUNC(current_date,'year') |
+--------------------------+
| 2013-01-01               |
+--------------------------+
1 row in set
```

示例 2：返回当年的第一天。

```
gbase> SELECT TRUNC(current_date,'yyyy') FROM t;
+--------------------------+
| TRUNC(current_date,'yyyy') |
+--------------------------+
```

```
| 2013-01-01                |
+---------------------------+
1 row in set
```

示例 3：返回当月的第一天。

```
gbase> SELECT TRUNC(current_date,'mm') FROM t;
+--------------------------+
| TRUNC(current_date,'mm') |
+--------------------------+
| 2013-10-01               |
+--------------------------+
1 row in set
```

示例 4：返回当天的日期。

```
gbase> SELECT TRUNC(current_date,'dd') FROM t;
+--------------------------+
| TRUNC(current_date,'dd') |
+--------------------------+
| 2013-10-12               |
+--------------------------+
1 row in set
```

示例 5：得到当天凌晨 0 点 0 分 0 秒的日期。

```
gbase> SELECT TRUNC(current_date,'hh') FROM t;
+--------------------------+
| TRUNC(current_date,'hh') |
+--------------------------+
| 2013-10-12 00:00:00      |
+--------------------------+
1 row in set
```

示例 6：得到当天凌晨 0 点 0 分 0 秒的日期。

```
gbase> SELECT TRUNC(current_date,'mi') FROM t;
+--------------------------+
| TRUNC(current_date,'mi') |
```

```
+-------------------------+
| 2013-10-12 00:00:00     |
+-------------------------+
1 row in set
```

3.5.53UNIX_TIMESTAMP

UNIX_TIMESTAMP(), UNIX_TIMESTAMP(date)

如果调用时没有参数，以无符号的整数形式返回一个 Unix 时间戳（从“1970-01-01 00:00:00” GMT 开始的秒数）。

如果以一个参数 date 调用 UNIX_TIMESTAMP()，它将返回该参数值从“1970-01-01 00:00:00” GMT 开始经过的秒数值。

date 可以是一个 DATE 字符串，一个 DATETIME 字符串，一个 TIMESTAMP，以一个 YYMMDD 或 YYYYMMDD 显示的本地时间。

示例 1: UNIX_TIMESTAMP()没有参数，以无符号的整数形式返回一个 Unix 时间戳。

```
gbase> SELECT UNIX_TIMESTAMP() FROM t;
+------------------+
| UNIX_TIMESTAMP() |
+------------------+
|       1381557673 |
+------------------+
1 row in set
```

示例 2: UNIX_TIMESTAMP(date)，返回从“1970-01-01 00:00:00” GMT 开始，到“2011-08-30 22:23:00”所经过的秒数值。

```
gbase> SELECT UNIX_TIMESTAMP('2011-08-30 22:23:00') FROM t;
+---------------------------------------+
| UNIX_TIMESTAMP('2011-08-30 22:23:00') |
+---------------------------------------+
```

```
|                                1314714180 |
+-------------------------------------------+
1 row in set
```

当 UNIX_TIMESTAMP 被用在 TIMESTAMP 列时,函数直接返回内部时间戳值,而不进行任何隐含的“string-to-Unix-timestamp”转化。如果向 UNIX_TIMESTAMP()传递一个溢出日期，它会返回 0,但请注意只执行基本范围检查(年份从 1970 到 2037，月份从 01 到 12，日期从 01 到 31)。

如果用户希望减去 UNIX_TIMESTAMP()列，需要将结果强制转换为一个有符号整数。

3.5.54UTC_DATE，UTC_DATE()

按照“YYYY-MM-DD”或“YYYYMMDD”格式返回当前 UTC 日期，返回值的形式取决于该函数使用于字符串还是数字上下文。

示例 1：以“YYYY-MM-DD”或“YYYYMMDD”格式返回当前 UTC 日期。

```
gbase> SELECT UTC_DATE(), UTC_DATE() + 0 FROM t;
+------------+---------------+
| UTC_DATE() | UTC_DATE() + 0 |
+------------+---------------+
| 2013-10-12 |       20131012 |
+------------+---------------+
1 row in set
```

3.5.55UTC_TIME，UTC_TIME()

按照“HH:MI:SS”或“HHMISS”格式返回当前的 UTC 时间，返回值的形式取决于该函数使用于字符串还是数字上下文。

示例 1：以“HH:MI:SS”或“HHMISS”格式返回当前的 UTC 时间。

```
gbase> SELECT UTC_TIME(), UTC_TIME() + 0 FROM t;
+------------+---------------+
| UTC_TIME() | UTC_TIME() + 0 |
+------------+---------------+
| 06:02:18   |   60218.000000 |
+------------+---------------+
1 row in set
```

3.5.56UTC_TIMESTAMP, UTC_TIMESTAMP()

按照“YYYY-MM-DD HH:MI:SS”或“YYYYMMDDHHMISS”格式返回当前 UTC 日期，返回值的形式取决于该函数使用于字符串还是数字上下文。

示例 1：以“YYYY-MM-DD HH:MI:SS”或 YYYYMMDDHHMISS 格式返回当前 UTC “日期+时间”。

```
gbase> SELECT UTC_TIMESTAMP(), UTC_TIMESTAMP() + 0 FROM t;
+---------------------+----------------------+
| UTC_TIMESTAMP()     | UTC_TIMESTAMP() + 0  |
+---------------------+----------------------+
| 2013-10-12 06:02:44 | 20131012060244.000000 |
+---------------------+----------------------+
1 row in set
```

3.5.57WEEK(date[,mode])

该函数返回日期的星期数。

两个参数形式的 WEEK()允许用户指定周是否以星期日或星期一开始，以及返回值为 0～53 还是 1～53。

如果忽略了 mode 参数，则使用系统变量 default_week_format 的值。

说明：可以使用 SHOW VARIABLES LIKE '%default_week_format%';命令查

看 default_week_format 的默认值。

下面的表显示了 mode 参数怎样工作：

模式	周的起始天	范围	一周是第一周.....
0	Sunday	0～53	一周以星期日开始
1	Monday	0～53	这种模式多用了 3 天
2	Sunday	1～53	一周以星期日开始
3	Monday	1～53	这种模式多用了 3 天
4	Sunday	0～53	这种模式多用了 3 天
5	Monday	0～53	一周以星期一开始
6	Sunday	1～53	这种模式多用了 3 天
7	Monday	1～53	一周以星期一开始

示例 1：返回“2010-08-31”对应的 2010 年中第几周。

```
gbase> SELECT WEEK('2010-08-31') FROM t;
+--------------------+
| WEEK('2010-08-31') |
+--------------------+
|                 35 |
+--------------------+
1 row in set
```

示例 2：返回“2011-08-30”对应的 2011 年中第几周，模式为“0”。

```
gbase> SELECT WEEK('2011-08-30',0) FROM t;
+----------------------+
| WEEK('2011-08-30',0) |
+----------------------+
|                   35 |
+----------------------+
1 row in set
```

示例 3：返回“2011-08-30”对应的 2011 年中第几周，模式为“1”。

```
gbase> SELECT WEEK('2011-08-30',1) FROM t;
+----------------------+
```

```
| WEEK('2011-08-30',1) |
+----------------------+
|                   35 |
+----------------------+
1 row in set
```

示例 4：返回“2011-12-31”对应的 2011 年中第几周，模式为“1”。

```
gbase> SELECT WEEK('2011-12-31',1) FROM t;
+----------------------+
| WEEK('2011-12-31',1) |
+----------------------+
|                   52 |
+----------------------+
1 row in set
```

示例 5：如果一周是上一年的最后一周，不使用可选模式 2，3，6，或 7，则函数 WEEK()返回 0。

```
gbase> SELECT YEAR('2011-01-01'), WEEK('2011-01-01',0) FROM t;
+--------------------+----------------------+
| YEAR('2011-01-01') | WEEK('2011-01-01',0) |
+--------------------+----------------------+
|               2011 |                    0 |
+--------------------+----------------------+
1 row in set
```

示例 6：返回“2011-01-01”对应的 2011 年中第几周，模式为“2”。

```
gbase> SELECT YEAR('2011-01-01'), WEEK('2011-01-01',2) FROM t;
+--------------------+----------------------+
| YEAR('2011-01-01') | WEEK('2011-01-01',2) |
+--------------------+----------------------+
|               2011 |                   52 |
+--------------------+----------------------+
1 row in set
```

示例 7：返回“2011-01-01”对应的 2011 年中第几周，模式为“3”。

```
gbase> SELECT YEAR('2011-01-01'), WEEK('2011-01-01',3) FROM t;
+--------------------+----------------------+
| YEAR('2011-01-01') | WEEK('2011-01-01',3) |
+--------------------+----------------------+
|               2011 |                   52 |
+--------------------+----------------------+
1 row in set
```

示例 8：返回“2011-01-01”对应的 2011 年中第几周，模式为“6”。

```
gbase> SELECT YEAR('2011-01-01'), WEEK('2011-01-01',6) FROM t;
+--------------------+----------------------+
| YEAR('2011-01-01') | WEEK('2011-01-01',6) |
+--------------------+----------------------+
|               2011 |                   52 |
+--------------------+----------------------+
1 row in set
```

示例 9：返回“2000-01-01”对应的 2000 年中第几周，模式为“0”。

```
gbase> SELECT YEAR('2000-01-01'), WEEK('2000-01-01',0) FROM t;
+--------------------+----------------------+
| YEAR('2000-01-01') | WEEK('2000-01-01',0) |
+--------------------+----------------------+
|               2000 |                    0 |
+--------------------+----------------------+
1 row in set
```

用户可能会提出意见，认为 GBase 8a MPP Cluster 对于 WEEK()函数应该返回“52”，原因是给定的日期实际上发生在 1999 年的第 52 周。我们决定返回“0”作为代替的原因，是希望该函数能返回“给定年份的星期数”。这使得 WEEK()函数在同其它从日期中抽取日期部分的函数结合时的使用更加可靠。

假如希望所计算的关于年份的结果包括给定日期所在周的第一天，则应使用 0、2、5 或 7 作为 mode 参数选择。

示例 10：返回“2011-01-01”对应的 2011 年中第几周，模式为“2”。

```
gbase> SELECT WEEK('2011-01-01',2) FROM t;
+----------------------+
| WEEK('2011-01-01',2) |
+----------------------+
|                   52 |
+----------------------+
1 row in set
```

示例 11：使用 YEARWEEK()函数。

```
gbase> SELECT YEARWEEK('2011-01-01') FROM t;
+-----------------------+
| YEARWEEK('2011-01-01') |
+-----------------------+
|                 201052 |
+-----------------------+
1 row in set
```

示例 12：使用 MID 函数，从“YEARWEEK('2011-01-01')”返回值中从第一个字符开始，连续提取 2 个。

```
gbase> SELECT MID(YEARWEEK('2011-01-01'),5,2) FROM t;
+--------------------------------+
| MID(YEARWEEK('2011-01-01'),5,2) |
+--------------------------------+
| 52                             |
+--------------------------------+
1 row in set
```

3.5.58 WEEKDAY(date)

返回 date 对应的星期索引(0=Monday,1=Tuesday,...6=Sunday)。

示例 1：返回“2011-08-29 22:23:00”对应的是星期几。

```
gbase> SELECT WEEKDAY('2011-08-29 22:23:00') FROM t;
+-------------------------------+
| WEEKDAY('2011-08-29 22:23:00') |
+-------------------------------+
|                             0 |
+-------------------------------+
1 row in set
```

示例 2：返回“2010-08-05”对应的是星期几。

```
gbase> SELECT WEEKDAY('2010-08-05') FROM t;
+-----------------------+
| WEEKDAY('2010-08-05') |
+-----------------------+
|                     3 |
+-----------------------+
1 row in set
```

3.5.59 WEEKOFYEAR(date)

返回 date 的周数。

date 的取值范围为 1～53。

WEEKOFYEAR(date)等价于 WEEK(date,3)。

示例 1：返回“2010-08-30”对应的是 2010 年的第几周。

```
gbase> SELECT WEEKOFYEAR('2010-08-30') FROM t;
+--------------------------+
| WEEKOFYEAR('2010-08-30') |
+--------------------------+
|                       35 |
+--------------------------+
1 row in set
```

3.5.60 YEAR(date)

返回日期 date 对应的年份值。date 的取值范围为 0～9999。

示例 1：返回“2011-02-03”对应的年份。

```
gbase> SELECT YEAR('2011-02-03') FROM t;
+-------------------+
| YEAR('2011-02-03') |
+-------------------+
|               2011 |
+-------------------+
1 row in set
```

3.5.61 YEARWEEK(date)，YEARWEEK(date,start)

返回日期 date 对应的年和周。

YEARWEEK(date,start)中的参数 start，形式和作用与 WEEK()中 mode 参数相同。

示例 1：返回“2011-02-01”对应的年和周，模式为 0。

```
gbase> SELECT YEARWEEK('2011-02-01',0) FROM t;
+-------------------------+
| YEARWEEK('2011-02-01',0) |
+-------------------------+
|                  201105 |
+-------------------------+
1 row in set
```

示例 2：当日期参数 date 是一年的第一周或最后一周时，返回的年份值可能与日期参数给出的年份不一致。

```
gbase> SELECT YEARWEEK('2011-01-01') FROM t;
+-----------------------+
```

```
| YEARWEEK('2011-01-01') |
+------------------------+
|                 201052 |
+------------------------+
1 row in set
```

当 YEARWEEK()函数的参数 start 的值为 0 或 1 时，周值的返回值不同于 WEEK()函数的返回值(0)，WEEK()是根据给定的年返回周值。

```
gbase> SELECT YEARWEEK('2011-01-01',0),WEEK('2011-01-01',0)  FROM t;
+--------------------------+----------------------+
| YEARWEEK('2011-01-01',0) | WEEK('2011-01-01',0) |
+--------------------------+----------------------+
|                   201052 |                    0 |
+--------------------------+----------------------+
1 row in set

gbase> SELECT YEARWEEK('2011-01-01',1),WEEK('2011-01-01',1) FROM t;
+--------------------------+----------------------+
| YEARWEEK('2011-01-01',1) | WEEK('2011-01-01',1) |
+--------------------------+----------------------+
|                   201052 |                    0 |
+--------------------------+----------------------+
1 row in set
```

3.6 其它函数

3.6.1 位函数

GBase 8a 使用 BIGINT（64 位）算法进行位运算，所以这些操作符最大范围是 64 位。

3.6.1.1 |按位或

示例 1：返回“29 | 15”的计算结果。

```
gbase> SELECT 29 | 15 FROM t;
+---------+
| 29 | 15 |
+---------+
|      31 |
+---------+
1 row in set
```

示例说明：29 对应的比特值为“11101”，15 对应的比特值为“1111”，逐位进行与操作，结果为“11111”，对应的十进制值为“31”。

3.6.1.2 &按位与

示例 1：返回“29 & 15”的计算结果。

```
gbase> SELECT 29 & 15 FROM t;
+---------+
| 29 & 15 |
+---------+
|      13 |
+---------+
1 row in set
```

示例说明：“29”对应的比特值为“11101”，15 对应的比特值为“1111”，逐位进行与操作，结果为“1101”，对应的十进制值为“13”。

3.6.1.3 ^按位异或

示例 1：返回“1 ^ 1”的计算结果。

```
gbase> SELECT 1 ^ 1 FROM t;
+-------+
| 1 ^ 1 |
+-------+
|     0 |
+-------+
1 row in set
```

示例 2：返回“1 ^ 0”的计算结果。

```
gbase> SELECT 1 ^ 0 FROM t;
+-------+
| 1 ^ 0 |
+-------+
|     1 |
+-------+
1 row in set
```

示例 3：返回“11 ^ 3”的计算结果。

```
gbase> SELECT 11 ^ 3 FROM t;
+--------+
| 11 ^ 3 |
+--------+
|      8 |
+--------+
1 row in set
```

示例说明：“11”对应的比特值为“1011”，“3”对应的比特值为“0011”，逐位进行异或，结果为“1000”，对应的十进制值为“8”。

3.6.1.4 <<左移操作(BIGINT)

示例 1：返回“1 << 2”的计算结果。

```
gbase> SELECT 1 << 2 FROM t;
+--------+
```

```
| 1 << 2 |
+--------+
|      4 |
+--------+
1 row in set
```

示例说明：“1”对应的比特值为“0001”，左移两位为“0100”，对应的十进制为“1”。

3.6.1.5 >>右移操作(BIGINT)

示例 1：返回“4 >> 2”的计算结果。

```
gbase> SELECT 4 >> 2 FROM t;
+--------+
| 4 >> 2 |
+--------+
|      1 |
+--------+
1 row in set
```

示例说明：“4”对应的比特值为“0100”，右移两位为“0001”，对应的十进制值为“1”。

3.6.1.6 BIT_COUNT(N)

返回在参数 N 中设置的比特位是 1 的总数量。

示例 1：返回“29”设置的比特位中 1 的个数。

```
gbase> SELECT BIT_COUNT(29) FROM t;
+---------------+
| BIT_COUNT(29) |
+---------------+
|             4 |
+---------------+
```

```
1 row in set
```

示例说明：“29”对应的比特值为“11101”，对应的十进制值为“4”。

3.6.2 加密函数

这部分函数用于加密和解密数据。

3.6.2.1 AES_ENCRYPT

AES_ENCRYPT(str,key_str)

这个函数允许使用官方的 AES 算法加密数据，曾称为“Rijndael”。该编码使用密钥的长度为 128 位。

输入参数可以是任意长度。如果参数是 NULL，函数的返回结果也是 NULL。

如果 AES_DECRYPT()探测到无效的数据或者不正确的补位，它会返回 NULL。

AES_ENCRYPT()是目前 GBase 8a 中最有加密安全性的函数。

3.6.2.2 ENCRYPT(str[,salt])

使用 Linux 的 crypt()系统调用来加密 str。

参数 salt 应该是一个两个字符的字符串。如果 salt 没有给定，会使用一个随机数值。

示例 1：因未给定 salt 值，使用随机数值对“hello”进行加密。

```
gbase> SELECT ENCRYPT('hello') FROM t;
+------------------+
| ENCRYPT('hello') |
+------------------+
| AOIaUbAL5c.NM    |
+------------------+
```

```
1 row in set
```

ENCRYPT()在一些系统上忽略除了 str 前 8 个字符之外的全部字符，这个行为通过使用 crypt()系统调用来决定。

如果 crypt()在用户的系统上不可用，ENCRYPT()总是返回 NULL。所以推荐用户使用 MD5()或 SHA1()，这两个函数存在于所有平台上。

3.6.2.3 MD5(str)

为字符串计算一个 128 位的 MD5 校验和。结果作为 32 位 16 进制字符串返回，返回值可以用作哈希密钥。

如果参数 NULL 则返回 NULL。

示例 1：使用 MD5()对“testing”进行加密。

```
gbase> SELECT MD5('testing') FROM t;
+----------------------------------+
| MD5('testing')                   |
+----------------------------------+
| ae2b1fca515949e5d54fb22b8ed95575 |
+----------------------------------+
1 row in set
```

这就是“RSA Data Security, Inc. MD5 Message-Digest Algorithm.”。

如果用户想要转换值到大写，参考在转换函数和操作符中描述的 BINARY 操作符，它可以用于二进制字符串的转换。

3.6.2.4 SHA1(str), SHA(str)

按照 RFC3174(安全哈希算法）中介绍的那样，为字符串计算一个 160 位的 SHA1 校验和。结果作为 40 位 16 进制字符串返回。

若 str 的值为 NULL，则返回 NULL。

常用的就是作为哈希密钥。用户还可以用它作为一个加密安全函数来存储密码。

示例 1：为“abc”计算一个 160 位的 SHA1 校验和，结果作为 40 位 16 进制字符串返回。

```
gbase> SELECT SHA1('abc') FROM t;
+------------------------------------------+
| SHA1('abc')                              |
+------------------------------------------+
| a9993e364706816aba3e25717850c26c9cd0d89d |
+------------------------------------------+
1 row in set
```

SHA1()可以认为加密安全性等价于 MD5().SHA()是 SHA1()的同义词。

3.6.3 信息函数

3.6.3.1 BENCHMARK(count,expr)

BENCHMARK()函数用于将表达式 expr 重复运行 count 次。

它可以用于计时 GBase 8a 处理表达式的时间，结果通常为 0。

在 gbase 客户端使用它时，它将返回查询执行所需的时间。

示例 1：将“hello”重复运行 1000000 次。

```
gbase> SELECT BENCHMARK(1000000,'hello') FROM t;
+---------------------------+
| BENCHMARK(1000000,'hello') |
+---------------------------+
|                         0 |
+---------------------------+
1 row in set
```

报告的时间是客户端操作的时间，不是服务器端的 CPU 时间。计算时间是

应当执行 BENCHMARK()多次，并注意参考服务器的负载来解释结果。

3.6.3.2 CHARSET(str)

返回字符串参数使用的字符集。

示例 1：返回“示例”使用的字符集。

```
gbase> SELECT CHARSET('示例') FROM t;
+------------------+
| CHARSET('示例')  |
+------------------+
| utf8             |
+------------------+
1 row in set
```

示例 2：返回“USER()”使用的字符集。

```
gbase> SELECT CHARSET(USER()) FROM t;
+-----------------+
| CHARSET(USER()) |
+-----------------+
| utf8            |
+-----------------+
1 row in set
```

3.6.3.3 COLLATION(str)

返回字符串参数的字符集排序规则。

示例 1：返回“abc”的字符集排序规则。

```
gbase> SELECT COLLATION('abc') FROM t;
+------------------+
| COLLATION('abc') |
+------------------+
```

```
| utf8_general_ci  |
+------------------+
1 row in set
```

示例 2：返回“_gb2312 'abc'”的字符集排序规则。

```
gbase> SELECT COLLATION(_gb2312 'abc') FROM t;
+------------------+
| COLLATION('abc') |
+------------------+
| utf8_general_ci  |
+------------------+
1 row in set
```

3.6.3.4 CONNECTION_ID()

返回当前连接的连接 ID(thread ID)。每个连接均有一个各自唯一的 ID。

示例 1：返回当前连接的连接 ID。

```
gbase> SELECT CONNECTION_ID() FROM t;
+-----------------+
| CONNECTION_ID() |
+-----------------+
|              11 |
+-----------------+
1 row in set
```

3.6.3.5 CURRENT_USER()

返回当前会话连接匹配的用户名和主机名的结合。这个值对应于决定用户访问权限的帐号。

示例 1：返回当前会话连接匹配的用户名和主机名。

```
gbase> SELECT USER() FROM t;
+----------------+
| USER()         |
+----------------+
| root@localhost |
+----------------+
1 row in set
```

示例 2：返回当前会话连接匹配的用户名和主机名的结合。

```
gbase> SELECT CURRENT_USER() FROM t;
+----------------+
| CURRENT_USER() |
+----------------+
| root@%         |
+----------------+
1 row in set
```

3.6.3.6 DATABASE()

返回当前使用的数据库名。

示例 1：当前使用的数据库为“test”。

```
gbase> SELECT DATABASE() FROM t;
+------------+
| DATABASE() |
+------------+
| test       |
+------------+
1 row in set
```

示例 2：如果没有当前数据库，系统显示提示信息。

```
gbase> SELECT DATABASE() FROM t;
ERROR 1046 (3D000): No database selected
```

3.6.3.7 SESSION_USER()

SESSION_USER()等价于 USER()。

示例 1：返回当前的 GBase 8a 用户和主机名。

```
gbase> SELECT SESSION_USER() FROM t;
+----------------+
| SESSION_USER() |
+----------------+
| root@localhost |
+----------------+
1 row in set
```

3.6.3.8 SYSTEM_USER()

SYSTEM_USER()等价于 USER()。

示例 1：返回当前的 GBase 8a 用户和主机名。

```
gbase> SELECT SYSTEM_USER() FROM t;
+----------------+
| SYSTEM_USER()  |
+----------------+
| root@localhost |
+----------------+
1 row in set
```

3.6.3.9 USER()

返回当前的 GBase 8a 用户和主机名。

示例 1：当前的用户为“root”，主机名为“localhost”

```
gbase> SELECT USER() FROM t;
+----------------+
| USER()         |
+----------------+
| root@localhost |
+----------------+
1 row in set
```

这个值是用户连接的用户名和连接的主机名。它不同于 CURRENT_USER()的返回值。

示例 2：用户可以精简到只剩用户名。

```
gbase> SELECT SUBSTRING_INDEX(USER(),'@',1) FROM t;
+-----------------------------+
| SUBSTRING_INDEX(USER(),'@',1) |
+-----------------------------+
| root                        |
+-----------------------------+
1 row in set
```

示例 3：USER()返回属于 UTF8 字符集的值，因此用户也确保了“'@'”字符串文字可以在该字符集中得到解释。

```
gbase> SELECT SUBSTRING_INDEX(USER(),_utf8'@',1) FROM t;
+----------------------------------+
| SUBSTRING_INDEX(USER(),_utf8'@',1) |
+----------------------------------+
| root                             |
+----------------------------------+
1 row in set
```

3.6.3.10 VERSION()

以字符串形式返回 GBase 8a 服务器的版本号。

示例 1：当前版本号为 8.5.1.2。

```
gbase> SELECT VERSION()FROM t;
+-----------+
| VERSION() |
+-----------+
| 8.5.1.2   |
+-----------+
1 row in set
```

3.6.4 辅助函数

3.6.4.1 FORMAT(X,D)

将数字 X 格式化为“#，###，###.##”的形式，四舍五入到 D 位小数。

如果 D 为 0，返回的结果将没有小数点和小数部分。

示例 1：将“12332.123456”进行格式化，四舍五入到 4 位小数。

```
gbase> SELECT FORMAT(12332.123456, 4) FROM t;
+-------------------------+
| FORMAT(12332.123456, 4) |
+-------------------------+
| 12,332.1235             |
+-------------------------+
1 row in set
```

示例 2：将“12332.1”进行格式化，四舍五入到 4 位小数，小数部分不足四位，用 0 补足。

```
gbase> SELECT FORMAT(12332.1,4) FROM t;
+------------------+
| FORMAT(12332.1,4) |
+------------------+
| 12,332.1000      |
```

```
+------------------+
1 row in set
```

示例 3：如果 D 为 0，返回的结果将没有小数点和小数部分。

```
gbase> SELECT FORMAT(12332.2,0) FROM t;
+------------------+
| FORMAT(12332.2,0) |
+------------------+
| 12,332           |
+------------------+
1 row in set
```

3.6.4.2 INET_ATON(expr)

给定一个用“.”分隔的字符串网络地址，即 IP 地址，函数返回一个整数，用来表示地址数值。地址可能是 4 到 8 个字节长。

示例 1：返回“209.207.224.40”对应的整数。

```
gbase> SELECT INET_ATON('209.207.224.40') FROM t;
+----------------------------+
| INET_ATON('209.207.224.40') |
+----------------------------+
|                 3520061480 |
+----------------------------+
1 row in set
```

生成的数字总是按照网络字节次序。如示例 1 所示，数值按照 $209*256^3+207*256^2+224*256^1+40*256^0$ 来计算。

示例 2：INET_ATON()还可以使用短格式的 IP 地址。

```
gbase> SELECT INET_ATON('127.0.0.1'), INET_ATON('127.1') FROM t;
+----------------------+--------------------+
| INET_ATON('127.0.0.1') | INET_ATON('127.1') |
+----------------------+--------------------+
```

```
|              2130706433 |          2130706433 |
+-------------------------+---------------------+
1 row in set
```

3.6.4.3 INET_NTOA(expr)

给定一个数字的网络地址（4 或 8 字节），以字符串形式返回点组样式表示的地址。

示例 1：返回“3520061480”对应的 IP 地址。

```
gbase> SELECT INET_NTOA(3520061480) FROM t;
+-------------------------+
| INET_NTOA(3520061480) |
+-------------------------+
| 209.207.224.40          |
+-------------------------+
1 row in set
```

3.6.4.4 SLEEP(duration)

睡眠(暂停)时间为 duration 参数给定的秒数，然后返回 0。

示例 1：暂停 10 秒后返回 0。

```
gbase> SELECT SLEEP(10) FROM t;
+------------+
| SLEEP(10) |
+------------+
|          0 |
+------------+
1 row in set
```

3.6.4.5 UUID()

返回一个通用唯一的标识符（UUID），其产生依据是公用组织在 1997.10 出版（文档号 C706）的“DCE1.1： Remote Procedure Call”CAE（通用应用程序环境）说明书。

UUID 是一个在空间和时间上全局唯一的号码。两次调用 UUID（）会返回两个不同的数值，即使这些调用是在两台独立的计算机上发生，彼此并不相关。

UUID 是一个 128 位的数字，由五位十六进制数的字符串表示这个数字，格式为 aaaaaaaa-bbbb-cccc-dddd-eeeeeeeeeeee。

前三个数字从时间戳中产生；

第四个数字在时间戳失去唯一性的时候保护时间的唯一性（例如，由于夏令时）；

第五个数字是一个 IEEE802 结点号，可以提供空间的唯一性。如果后面的部分不可用，则使用一个随机数替代（例如，由于主机没有以太网卡，或不知道怎样在操作系统上找到界面的机器地址），在这种情况下，空间唯一性不能保证。尽管如此，发生冲突的可能性还是很小。

当前，只在 Linux 上考虑 MAC 地址。在其它操作系统上，GBase 8a 使用一个随机产生的 48 位数字。

示例 1：返回一个通用唯一的标识符。

```
gbase> SELECT UUID() FROM t;
+--------------------------------------+
| UUID()                               |
+--------------------------------------+
| decb057a-6941-11e3-bd85-080027f03271 |
+--------------------------------------+
1 row in set
```

3.7 用于 GROUP BY 子句的函数和修饰语

3.7.1 GROUP BY（聚集）函数

如果用户在一条语句中使用聚集函数而不使用 GROUP BY 子句，它等价于在所有行上进行分组。

GBase 8a 扩展了 GROUP BY 的用法。在 SELECT 表达式中，用户可以使用或计算没有出现在 GROUP BY 部分中的列，它代表这个组的任何可能的值。用户可以使用它避免在不必要的分类项目上进行排序和分组，这样会得到更好的性能。

示例 1：统计欧洲的供应商，品牌为 MFGR#2221 的商品的每年的总收入。统计结果按年，品牌递增排序。

```
gbase> SELECT SUM(lo_revenue), d_year, p_brand1
    -> FROM lineorder, dwdate, part, supplier
    -> WHERE lo_orderdate = d_datekey
    -> AND lo_partkey = p_partkey
    -> AND lo_suppkey = s_suppkey
    -> AND p_brand1 = 'MFGR#2221'
    -> AND s_region = 'EUROPE'
    -> GROUP BY d_year, p_brand1
    -> ORDER BY d_year, p_brand1;
+-----------------+--------+-----------+
| SUM(lo_revenue) | d_year | p_brand1  |
+-----------------+--------+-----------+
|       584006585 |   1992 | MFGR#2221 |
|       587827610 |   1993 | MFGR#2221 |
|       598245883 |   1994 | MFGR#2221 |
|       646475117 |   1995 | MFGR#2221 |
|       618399648 |   1996 | MFGR#2221 |
|       687290315 |   1997 | MFGR#2221 |
|       590650585 |   1998 | MFGR#2221 |
+-----------------+--------+-----------+
7 rows in set
```

如果用户在 GROUP BY 部分省略的列在分组中不是唯一的，请不要使用这个特性，否则将得到不可预知的结果。

3.7.1.1 AVG([DISTINCT] expr)

返回 expr 的平均值。加上了 DISTINCT 选项用于返回 expr 中所有不同值的平均值。

示例 1：返回“lo_supplycost”的平均值。

```
gbase> SELECT lo_orderpriority, AVG(lo_supplycost) FROM ssbm.lineorder GROUP BY LO_ORDERPRIORITY;
+-----------------+--------------------+
| lo_orderpriority | AVG(lo_supplycost) |
+-----------------+--------------------+
| 2-HIGH           |         89935.4667 |
| 1-URGENT         |         89979.8854 |
| 4-NOT SPECI      |         89965.0583 |
| 5-LOW            |         89978.3022 |
| 3-MEDIUM         |         89954.0913 |
+-----------------+--------------------+
5 rows in set
```

3.7.1.2 COUNT(expr)

返回一个 SELECT 语句检索出来的记录行中非 NULL 值的记录总数目。

示例 1：检索满足条件的 c_mktsegment 列的记录中非 NULL 值的记录总数，并按 c_mktsegment 进行分组。

```
gbase> SELECT c.c_mktsegment,COUNT(*) FROM ssbm.lineorder l, ssbm.customer c WHERE l.lo_custkey = c.c_custkey GROUP BY c.c_mktsegment;
+-------------+----------+
| c_mktsegment | COUNT(*) |
+-------------+----------+
```

```
| AUTOMOBILE    |  1174124 |
| HOUSEHOLD     |  1217475 |
| MACHINERY     |  1196343 |
| BUILDING      |  1230857 |
| FURNITURE     |  1182372 |
+---------------+----------+
5 rows in set
```

COUNT(*)在它返回检索出的记录行的数目上稍微有点不同，它不管记录行中是否包括 NULL 值。因为，ssbm 是一个随机生成数据的演示数据库，因此执行结果会有差异，上述结果只是一个演示结果。

3.7.1.3 COUNT(DISTINCT expr,[expr...])

返回不同的非 NULL 的值的总数目。

示例 1：返回 lo_orderkey 列中不同的非 NULL 的值的总数目。

```
gbase> SELECT COUNT(DISTINCT lo_orderkey) FROM ssbm.lineorder;
+----------------------------+
| COUNT(DISTINCT lo_orderkey) |
+----------------------------+
|                    1500000 |
+----------------------------+
1 row in set
```

在 GBase 8a 中，用户通过给定一个表达式列表而获取不包含 NULL 不同表达式组合的数目。而在标准 SQL 中，用户必须在 COUNT(DISTINCT...)中连接所有的表达式。

3.7.1.4 MIN(), MAX()

MIN([DISTINCT] expr), MAX([DISTINCT] expr)

返回 expr 的最小或最大值。

可以为 MIN()和 MAX()设置参数；在这种情况下，它们将返回参数指定列的最小或最大值。

DISTINCT 关键词可以被用来查找 expr 的不同值的最小或最大值，然而，这样产生的结果与省略 DISTINCT 的结果相同。

MIN 和 MAX 不包括 NULL 值。

示例 1：返回 lo_supplycost 列的最大值和最小值。

```
gbase> SELECT lo_shipmode,MAX(lo_supplycost),MIN(lo_supplycost) FROM
ssbm.lineorder GROUP BY lo_shipmode;
+-------------+--------------------+--------------------+
| lo_shipmode | MAX(lo_supplycost) | MIN(lo_supplycost) |
+-------------+--------------------+--------------------+
| TRUCK       |             125939 |              54060 |
| MAIL        |             125939 |              54060 |
| REG AIR     |             125939 |              54060 |
| RAIL        |             125939 |              54060 |
| FOB         |             125939 |              54060 |
| SHIP        |             125939 |              54060 |
| AIR         |             125939 |              54060 |
+-------------+--------------------+--------------------+
7 rows in set
```

3.7.1.5 SUM([DISTINCT]expr)

返回 expr 的总和。

如果返回的结果集中没有任何记录行，它将返回 NULL。

DISTINCT 关键字用于计算 expr 中不同值的总和。

3.8 OLAP 函数

GBase 8a 提供了丰富的 OLAP 函数，辅助用户完成一些复杂的查询统计。在

使用这些函数时，请注意以下几点事项：

1、OLAP 函数中的 PARTITION BY 和 ORDER BY 的括号内不再支持使用别名。

SELECT a AS e ,RANK() OVER(ORDER BY e) FROM t1; -- a AS e 后 OVER(ORDER BY e) 引用了别名 e，不支持此种别名引用。

2、OLAP 函数中的 PARTITION BY 和 ORDER BY 的括号内的整型数值不是用于指定查询结果列的索引。

SELECT a, RANK() OVER(ORDER BY 1) FROM t1;在这个查询语句中，ORDER BY 括号里的 1 不是用于指定引用查询结果列的索引的含义，即不是指代 a 列，而是当作常量 1 来处理。

3.8.1 用于分组统计的 OLAP 函数

演示所用的表及数据。

```
DROP TABLE IF EXISTS t3;
CREATE TABLE t3 (color_type varchar(20),color_count int, in_date date);
```

插入演示数据。

```
INSERT INTO t3 (color_type,in_date,color_count)
VALUES('black','2010-09-11',18),('black','2010-10-05',18),('black','2010-10-13',31),('blue','2010-09-21',23),('blue','2010-09-30',15),('blue','2010-10-11',62),('red','2010-09-12',41),('red','2010-10-01',12),('red','2010-10-05',11);
```

3.8.1.1 GROUP BY CUBE 函数

语法：GROUP BY CUBE((…),(…),…)

功能：对 CUBE 后面括号里的 n 个字段或表达式组合做 GROUP BY 操作，最后将结果合并在一起，组合方式为 n 个字段或表达式的全部子集。

详见下面的解释：

GROUP BY CUBE(A,B,C) (A、B、C 代表语法中的“(…)”)

首先会对(A、B、C)进行 GROUP BY，然后依次是(A、B)，(A、C)，(A)，(B、C)，(B)，(C)，然后对全表进行 GROUP BY 操作，最后将所有结果合并在一起(相当于 UNION ALL 操作)，如果 n 个字段或表达式中的一个或多个在某一分组中不出现在 GROUP BY 后面，用 NULL 代替不出现的字段或表达式。

示例 1: GROUP BY CUBE(color_type,f_YearMonth)

```
gbase> SELECT color_type,in_date,color_count FROM t3 ORDER BY
color_type,in_date;
+------------+------------+-------------+
| color_type | in_date    | color_count |
+------------+------------+-------------+
| black      | 2010-09-11 |          18 |
| black      | 2010-10-05 |          18 |
| black      | 2010-10-13 |          31 |
| blue       | 2010-09-21 |          23 |
| blue       | 2010-09-30 |          15 |
| blue       | 2010-10-11 |          62 |
| red        | 2010-09-12 |          41 |
| red        | 2010-10-01 |          12 |
| red        | 2010-10-05 |          11 |
+------------+------------+-------------+
9 rows in set

gbase> SELECT NVL(color_type,'') as
color_type_show,NVL(DECODE(color_type,NULL,f_YearMonth || '合计
',NVL(f_YearMonth,color_type || ' 小计')),'总计') AS
f_YearMonth_show,SUM(color_count) FROM (SELECT
color_type,DATE_FORMAT(in_date, '%Y-%m') as f_YearMonth,color_count FROM t3)
t GROUP BY CUBE(color_type,f_YearMonth) ORDER BY color_type,f_YearMonth;
+----------------+-----------------+-----------------+
| color_type_show | f_YearMonth_show | SUM(color_count) |
+----------------+-----------------+-----------------+
| black          | 2010-09         |              18 |
| black          | 2010-10         |              49 |
```

```
| black           | black 小计       |              67 |
| blue            | 2010-09          |              38 |
| blue            | 2010-10          |              62 |
| blue            | blue 小计        |             100 |
| red             | 2010-09          |              41 |
| red             | 2010-10          |              23 |
| red             | red 小计         |              64 |
|                 | 2010-09 合计     |              97 |
|                 | 2010-10 合计     |             134 |
|                 | 总计             |             231 |
+-----------------+------------------+-----------------+
12 rows in set
```

3.8.1.2 GROUP BY ROLLUP 函数

语法：GROUP BY ROLLUP((…),(…),…)

功能：对 rollup 后面括号里的 n 个字段或表达式组合做 GROUP BY 操作，最后将结果合并在一起，组合方式为 n、n-1、n-2、…、1、0。

详见下面的解释：

GROUP BY ROLLUP(A,B,C) (A、B、C 代表语法中的“(…)”)

首先会对(A、B、C)进行 GROUP BY，然后对(A、B)进行 GROUP BY，然后是(A)进行 GROUP BY，然后对全表进行 GROUP BY 操作，最后将所有结果合并在一起(相当于 UNION ALL 操作),如果 n 个字段或表达式中的一个或多个在某一分组中不出现在 GROUP BY 后面，用 NULL 代替不出现的字段或表达式。

通常该函数用于统计例如商品的明细，小计以及最后总计的场景。

示例 1：GROUP BY ROLLUP(color_type,f_YearMonth)

```
gbase> SELECT NVL(color_type,'') as
color_type_show,DECODE(NVL(color_type,''),'','总计
',NVL(f_YearMonth,color_type || ' 小计')) AS
f_YearMonth_show,SUM(color_count) FROM (SELECT
```

```
color_type,DATE_FORMAT(in_date, '%Y-%m') as f_YearMonth,color_count FROM t3)
t GROUP BY ROLLUP(color_type,f_YearMonth) ORDER BY color_type,f_YearMonth;
+----------------+-----------------+------------------+
| color_type_show | f_YearMonth_show | SUM(color_count) |
+----------------+-----------------+------------------+
| black          | 2010-09         |               18 |
| black          | 2010-10         |               49 |
| black          | black 小计      |               67 |
| blue           | 2010-09         |               38 |
| blue           | 2010-10         |               62 |
| blue           | blue 小计       |              100 |
| red            | 2010-09         |               41 |
| red            | 2010-10         |               23 |
| red            | red 小计        |               64 |
|                | 总计            |              231 |
+----------------+-----------------+------------------+
10 rows in set
```

3.8.1.3 GROUP BY GROUPING SETS 函数

语法：GROUP BY GROUPING SETS((…),(…),…)

功能：对 GROUPING SETS 后面括号里的 n 个字段或表达式分别做 group by 操作，最后将结果合并在一起。

详见下面的解释：

GROUP BY GROUPING SETS (A,B,C) (A、B、C 代表语法中的“(…)”)

首先对(A)进行 GROUP BY，然后对(B)进行 GROUP BY，然后对(C)进行 GROUP BY，最后将所有结果合并在一起(相当于 UNION ALL 操作)，如果 n 个字段或表达式中的一个或多个在某一分组中不出现在 GROUP BY 后面，用 NULL 代替不出现的字段或表达式。

示例 1：GROUP BY GROUPING SETS(color_type,f_YearMonth)

```
gbase> SELECT color_type,in_date,color_count FROM t3 ORDER BY
color_type,in_date;
+------------+------------+-------------+
| color_type | in_date    | color_count |
+------------+------------+-------------+
| black      | 2010-09-11 |          18 |
| black      | 2010-10-05 |          18 |
| black      | 2010-10-13 |          31 |
| blue       | 2010-09-21 |          23 |
| blue       | 2010-09-30 |          15 |
| blue       | 2010-10-11 |          62 |
| red        | 2010-09-12 |          41 |
| red        | 2010-10-01 |          12 |
| red        | 2010-10-05 |          11 |
+------------+------------+-------------+
9 rows in set

gbase> SELECT NVL(color_type,'') as
color_type_show,DECODE(color_type,NULL,f_YearMonth || '合计
',NVL(f_YearMonth,color_type || ' 小计')) AS
f_YearMonth_show,SUM(color_count) FROM (SELECT
color_type,DATE_FORMAT(in_date, '%Y-%m') as f_YearMonth,color_count FROM t3)
t GROUP BY GROUPING SETS(color_type,f_YearMonth) ORDER BY
color_type,f_YearMonth;
+-----------------+-----------------+-----------------+
| color_type_show | f_YearMonth_show | SUM(color_count) |
+-----------------+-----------------+-----------------+
| black           | black 小计       |              67 |
| blue            | blue 小计        |             100 |
| red             | red 小计         |              64 |
|                 | 2010-09 合计     |              97 |
|                 | 2010-10 合计     |             134 |
+-----------------+-----------------+-----------------+
5 rows in set
```

3.8.2 非分组统计的 OLAP 函数

3.8.2.1 RANK OVER 函数

RANK() OVER([PARTITION BY col_name1,col_name2,…] ORDER BY col_name1 [ASC/DESC], col_name2 [ASC/DESC],…)

功能描述:

根据 ORDER BY 子句中表达式的值，从查询返回的每一行计算它们与其它行的相对位置。组内的数据按 ORDER BY 子句排序，然后给每一行赋一个号，从而形成一个序列，该序列从 1 开始，往后累加。

每次 ORDER BY 表达式的值发生变化时，该序列也随之增加。有同样值的行得到同样的数字序号（认为 null 是相等的）。

如果两行得到同样的排序，则后面的序数将跳跃。例如，两行序数为 1，则没有序数 2，序列将给组中的下一行分配值 3。

仅 Express 引擎支持。

在查询语句中，可以使用 RANK 函数的子句为:

1）在 SELECT 列表中: SELECT RANK() OVER(PARTITION BY i ORDER BY j) FROM t1 WHERE ...;

2）在最终 ORDER BY 子句中（通过在查询中的其它位置使用 RANK 函数的别名或位置引用）:

SELECT *,RANK() OVER (ORDER BY j) FROM t1 WHERE ... ORDER BY RANK() OVER(ORDER BY i DESC);

3）在上述两个子句中，作为表达式或标量函数的参数:

SELECT RANK() OVER (ORDER BY j DESC) + i FROM t1 WHERE ...;

SELECT CONV(RANK() OVER(PARTITION BY i ORDER BY j ASC),10,2)FROM t

使用约束（下述情况不能使用）：

1)在 WHERE 子句的搜索条件中：

```
SELECT ... FROM t1 WHERE RANK() OVER(ORDER BY i) > 3;  -- error
```

2)作为聚集函数的参数：

```
SELECT SUM(RANK() OVER(ORDER BY dollars)) FROM t1; -- error
```

3)RANK 函数不得在 HAVING 子句中使用：

```
SELECT * FROM t1 GROUP BY i HAVING RANK() OVER(ORDER BY j) < 10; --
error
```

4)RANK 函数不得在 GROUP BY LIST 中：

```
SELECT * FROM t1 GROUP BY RANK() OVER(ORDER BY i); -- error
```

5)RANK 不能嵌套在其它 RANK 内部：

```
SELECT RANK() OVER(ORDER BY RANK() OVER(ORDER BY i )); -- error
```

6)不得在 RANK 内部中使用外部引用：

```
SELECT * FROM t1 WHERE i IN (SELECT RANK () OVER(ORDER BY t1.k) FROM
t2);-- error
```

7)RANK 函数不得在 DELETE 和 UPDATE 语句的非查询部分：

```
UPDATE t1 SET i = RANK () OVER(ORDER BY j) WHERE ...; --error
```

但是用在查询部分可以：

```
UPDATE t1 SET i = i + 1 where j IN (SELECT RANK () OVER(ORDER BY t2.k)
from t2);  -- ok
```

注意：PARTITION BY 后面不能接 ASC/DESC,ORDER BY 后面可以接。

示例 1：RANK() OVER(PARTITION BY i ORDER BY j desc)

```
gbase> DROP TABLE IF EXISTS t1;
Query OK, 0 rows affected
```

```
gbase> CREATE TABLE t1(i int, j int);
Query OK, 0 rows affected

gbase> INSERT INTO t1
VALUES(2,1),(2,3),(2,3),(2,5),(3,2),(3,2),(3,2),(3,4),(3,1),(3,5);
Query OK, 10 rows affected
Records: 10  Duplicates: 0  Warnings: 0

gbase> SELECT *,RANK() OVER(PARTITION BY i ORDER BY j desc) AS rank FROM t1;
+------+------+------+
| i    | j    | rank |
+------+------+------+
|    2 |    5 |    1 |
|    2 |    3 |    2 |
|    2 |    3 |    2 |
|    2 |    1 |    4 |
|    3 |    5 |    1 |
|    3 |    4 |    2 |
|    3 |    2 |    3 |
|    3 |    2 |    3 |
|    3 |    2 |    3 |
|    3 |    1 |    6 |
+------+------+------+
10 rows in set
```

3.8.2.2 DENSE_RANK OVER 函数

DENSE_RANK( ) over([PARTITION BY col_name1,col_name2,…] ORDER BY col_name1 [ASC/DESC], col_name2 [ASC/DESC],…)

功能描述:

基本功能同 RANK 类似。

区别是如果两行得到同样的排序，则后面的序数不跳跃。例如，两行序数

为 1，序列将给组中的下一行分配值 2。

仅 Express 引擎支持。

使用说明和使用约束同 RANK() OVER()。

示例 1: RANK,DENSE_RANK() OVER (partition by i order by j desc)

```
gbase> DROP TABLE IF EXISTS t1;
Query OK, 0 rows affected

gbase> CREATE TABLE t1(i int, j int);
Query OK, 0 rows affected

gbase> INSERT INTO t1
VALUES(2,1),(2,3),(2,3),(2,5),(3,2),(3,2),(3,2),(3,4),(3,1),(3,5);
Query OK, 10 rows affected
Records: 10  Duplicates: 0  Warnings: 0

gbase> SELECT *,RANK() OVER(PARTITION BY i ORDER BY j DESC) AS RANK,DENSE_RANK()
OVER (partition by i order by j desc) AS dense_rank FROM t1;
+------+------+------+------------+
| i    | j    | rank | dense_rank |
+------+------+------+------------+
|    2 |    5 |    1 |          1 |
|    2 |    3 |    2 |          2 |
|    2 |    3 |    2 |          2 |
|    2 |    1 |    4 |          3 |
|    3 |    5 |    1 |          1 |
|    3 |    4 |    2 |          2 |
|    3 |    2 |    3 |          3 |
|    3 |    2 |    3 |          3 |
|    3 |    2 |    3 |          3 |
|    3 |    1 |    6 |          4 |
+------+------+------+------------+
10 rows in set
```

3.8.2.3 ROW_NUMBER OVER 函数

ROW_NUMBER( ) OVER([PARTITION BY col_name1,col_name2,…] ORDER BY col_name1 [asc/desc], col_name2 [asc/desc],…)

功能描述:

同 RANK 的区别就是相同的排序值序号也会依次递增。

例如，两行排序值相同，则序数为 1，2。

仅 Express 引擎支持。

使用说明和使用约束同 RANK() OVER。

示例 1: ROW_NUMBER() OVER(PARTITION BY i ORDER BY j DESC)

```
gbase> DROP TABLE IF EXISTS t1;
Query OK, 0 rows affected

gbase> CREATE TABLE t1(i int, j int);
Query OK, 0 rows affected

gbase> INSERT INTO t1
VALUES(2,1),(2,3),(2,3),(2,5),(3,2),(3,2),(3,2),(3,4),(3,1),(3,5);
Query OK, 10 rows affected
Records: 10  Duplicates: 0  Warnings: 0

gbase> SELECT *,RANK() OVER(PARTITION BY i ORDER BY j DESC) AS rank,DENSE_RANK()
OVER(PARTITION BY i ORDER BY j DESC) AS dense_rank ,ROW_NUMBER() OVER(PARTITION
BY i ORDER BY j DESC) AS row_number FROM t1;
+------+------+------+------------+------------+
| i    | j    | rank | dense_rank | row_number |
+------+------+------+------------+------------+
|    2 |    5 |    1 |          1 |          1 |
|    2 |    3 |    2 |          2 |          2 |
|    2 |    3 |    2 |          2 |          3 |
|    2 |    1 |    4 |          3 |          4 |
```

```
|    3 |    5 |    1 |           1 |           1 |
|    3 |    4 |    2 |           2 |           2 |
|    3 |    2 |    3 |           3 |           3 |
|    3 |    2 |    3 |           3 |           4 |
|    3 |    2 |    3 |           3 |           5 |
|    3 |    1 |    6 |           4 |           6 |
+------+------+------+-------------+-------------+
10 rows in set
```

3.8.2.4 SUM OVER 函数

SUM ([DISTINCT/ALL] expr) OVER ([PARTITION BY ...] [ORDER BY ... [ASC/DESC]]

功能描述:

计算组内表达式的移动累加和。

示例 1: SUM(k) OVER(PARTITION BY i ORDER BY j DESC)

```
gbase> DROP TABLE IF EXISTS t1;
Query OK, 0 rows affected

gbase> CREATE TABLE t1(i int, j int,k int);
Query OK, 0 rows affected

gbase> INSERT INTO t1
VALUES(2,1,4),(2,3,6),(2,3,4),(2,5,8),(3,2,2),(3,2,4),(3,2,2),(3,4,6),(3,1
,2),(3,5,8);
Query OK, 10 rows affected
Records: 10  Duplicates: 0  Warnings: 0

gbase> SELECT *,SUM(k) OVER(PARTITION BY i ORDER BY j DESC) AS sum FROM t1;
+------+------+------+------+
| i    | j    | k    | sum  |
+------+------+------+------+
```

```
|    2 |    5 |    8 |    8 |
|    2 |    3 |    4 |   18 |
|    2 |    3 |    6 |   18 |
|    2 |    1 |    4 |   22 |
|    3 |    5 |    8 |    8 |
|    3 |    4 |    6 |   14 |
|    3 |    2 |    2 |   22 |
|    3 |    2 |    4 |   22 |
|    3 |    2 |    2 |   22 |
|    3 |    1 |    2 |   24 |
+------+------+------+------+
10 rows in set
```

用例分析：首先会根据 i 分组，在同组内，根据 j 降序排列，从每组的第一个值开始向后累加 k 值，相同的 j 值，对应的累加和相同，都是加到最后一个 j 值对应的 k 值，如果遇到不同组，从 0 开始重新累加。

注：NULL 值的处理方式同聚合函数 sum 类似，如果全为 NULL 值，则结果为 NULL，否则 NULL 不进行累加。

以 i 值为 2、2、2、2，j 值为 5、3、3、1，K 值为 8、4、6、4，sum 值为 8、18、18、22 为例，i=2，j=5，k=8 时，sum=8，i=2，j=3，k=4 以及 i=2，j=3，k=6 时，因为 j 值相同，所以 sum 值相同，计算过程为 sum = 8 + 4 + 6 = 18。

示例 2：SUM(distinct k) OVER(PARTITION BY i)

```
gbase> SELECT *,SUM(DISTINCT k) OVER(PARTITION BY i) AS sum FROM t1;
+------+------+------+------+
| i    | j    | k    | sum  |
+------+------+------+------+
|    2 |    3 |    6 |   18 |
|    2 |    3 |    4 |   18 |
|    2 |    5 |    8 |   18 |
|    2 |    1 |    4 |   18 |
|    3 |    2 |    2 |   20 |
|    3 |    2 |    4 |   20 |
|    3 |    2 |    2 |   20 |
```

```
|    3 |    4 |    6 |   20 |
|    3 |    1 |    2 |   20 |
|    3 |    5 |    8 |   20 |
+------+------+------+------+
10 rows in set
```

用例分析：首先根据 i 分组，由于没有 ORDER BY 部分，则同组内的累加和都相等，将同组内的不重复的 k 值进行累加，如果遇到不同组，从 0 重新开始。

以 i 值为 2、2、2、2，j 值为 5、3、3、1，K 值为 8、4、6、4，sum 值为 18、18、18、18 为例，因为在这 4 组数值中，不同的 k 值为 6、4、8，所以 sum= 6 + 4 + 8 = 18。

3.8.2.5 AVG OVER 函数

AVG ([DISTINCT/ALL] expr) OVER ([PARTITION BY ...] [ORDER BY ... [ASC/DESC]]

功能描述：计算组内表达式的移动平均值。

示例 1：AVG(k) OVER(PARTITION BY i ORDER BY j DESC)

```
gbase> DROP TABLE IF EXISTS t1;
Query OK, 0 rows affected

gbase> CREATE TABLE t1(i int, j int,k int);
Query OK, 0 rows affected

gbase> INSERT INTO t1
VALUES(2,1,4),(2,3,6),(2,3,4),(2,5,8),(3,2,2),(3,2,4),(3,2,2),(3,4,6),(3,1
,2),(3,5,8);
Query OK, 10 rows affected
Records: 10  Duplicates: 0  Warnings: 0

gbase> SELECT *,AVG(k) OVER(PARTITION BY i ORDER BY j DESC) AS avg FROM t1;
+------+------+------+--------+
```

```
| i    | j    | k    | avg    |
+------+------+------+--------+
|    2 |    5 |    8 | 8.0000 |
|    2 |    3 |    4 | 6.0000 |
|    2 |    3 |    6 | 6.0000 |
|    2 |    1 |    4 | 5.5000 |
|    3 |    5 |    8 | 8.0000 |
|    3 |    4 |    6 | 7.0000 |
|    3 |    2 |    2 | 4.4000 |
|    3 |    2 |    4 | 4.4000 |
|    3 |    2 |    2 | 4.4000 |
|    3 |    1 |    2 | 4.0000 |
+------+------+------+--------+
10 rows in set
```

用例分析：首先会根据 i 分组，在同组内，根据 j 降序排列，从每组的第一个值开始向后累加 k 值，同时记录 count(k)的值，相同的 j 值，对应的累加和、count 值相同，都是计算到最后一个 j 值对应的 k 值，如果遇到不同组，从 0 开始重新累加，最后用累加和除以 count 值则是最后的 avg 值。

注：NULL 值的处理方式同聚合函数 avg 类似，如果全为 NULL 值，则结果为 NULL，否则 NULL 不进行累加，也不计算在 count 内。

以 i 值为 2、2、2、2，j 值为 5、3、3、1，K 值为 8、4、6、4，sum 值为 8、6、6、5.5 为例，i=2，j=5，k=8 时，avg=8，i=2，j=3，k=4 以及 i=2，j=3，k=6 时，因为 j 值相同，所以 avg 值相同，计算过程为 avg = (8 + 4 + 6) / 3 = 6，i=2，j=1，k=4 时，avg=5.5，计算过程为 avg = (8 + 4 + 6 + 4) / 4 = 5.5。

示例 2：AVG(DISTINCT k) OVER(PARTITION BY i)

```
gbase> SELECT *,AVG(DISTINCT k) OVER(PARTITION BY i) AS avg FROM t1;
+------+------+------+--------+
| i    | j    | k    | avg    |
+------+------+------+--------+
|    2 |    3 |    6 | 6.0000 |
|    2 |    3 |    4 | 6.0000 |
```

```
|    2 |    5 |    8 | 6.0000 |
|    2 |    1 |    4 | 6.0000 |
|    3 |    2 |    2 | 5.0000 |
|    3 |    2 |    4 | 5.0000 |
|    3 |    2 |    2 | 5.0000 |
|    3 |    4 |    6 | 5.0000 |
|    3 |    1 |    2 | 5.0000 |
|    3 |    5 |    8 | 5.0000 |
+------+------+------+--------+
10 rows in set
```

用例分析：首先根据 i 分组，由于没有 ORDER BY 部分，则同组内的累加和、COUNT 值都相等，将同组内的 k 值进行累加同时计算 COUNT 值，如果遇到不同组，从 0 重新开始。

以 i 值为 2、2、2、2，j 值为 5、3、3、1，K 值为 8、4、6、4，avg 值为 6、6、6、6 为例，因为在这 4 组数值中，不同的 k 值为 6、4、8，所以 avg= (6 + 4 + 8) / 3 = 6。

3.8.2.6 COUNT OVER 函数

COUNT(*/[DISTINCT] col) OVER([PARTITION BY col_name1,col_name2,…] [ORDER BY col_name1 [ASC/DESC], col_name2 [ASC/DESC],…])

功能描述：

该函数用于计算分组中的记录数，如果是 COUNT(*)，不用考虑 NULL 值，否则，不包含参数为 NULL 的记录，如果包含 DISTINCT，要做去重操作。

示例 1：COUNT OVER 函数示例。

```
gbase> DROP TABLE IF EXISTS t2;
Query OK, 0 rows affected

gbase> CREATE TABLE t2(i int, j int,k int);
Query OK, 0 rows affected
```

```
gbase> INSERT INTO t2
VALUES(2,1,4),(2,3,6),(2,3,4),(2,5,8),(3,2,2),(3,2,4),(3,2,2),(3,4,6),(3,1
,2),(3,5,8);
Query OK, 10 rows affected
Records: 10  Duplicates: 0  Warnings: 0

gbase> SELECT *,COUNT(k) OVER(PARTITION BY i ORDER BY j DESC) AS sum FROM t2;
+------+------+------+-----+
| i    | j    | k    | sum |
+------+------+------+-----+
|    2 |    5 |    8 |   1 |
|    2 |    3 |    4 |   3 |
|    2 |    3 |    6 |   3 |
|    2 |    1 |    4 |   4 |
|    3 |    5 |    8 |   1 |
|    3 |    4 |    6 |   2 |
|    3 |    2 |    2 |   5 |
|    3 |    2 |    4 |   5 |
|    3 |    2 |    2 |   5 |
|    3 |    1 |    2 |   6 |
+------+------+------+-----+
10 rows in set

gbase> SELECT *,COUNT(DISTINCT k) OVER(PARTITION BY i) AS sum FROM t2;
+------+------+------+-----+
| i    | j    | k    | sum |
+------+------+------+-----+
|    2 |    3 |    6 |   3 |
|    2 |    3 |    4 |   3 |
|    2 |    5 |    8 |   3 |
|    2 |    1 |    4 |   3 |
|    3 |    2 |    2 |   4 |
|    3 |    2 |    4 |   4 |
|    3 |    2 |    2 |   4 |
|    3 |    4 |    6 |   4 |
|    3 |    1 |    2 |   4 |
```

```
|    3 |    5 |    8 |    4 |
+------+------+------+------+
10 rows in set
```

3.8.2.7 LEAD/LAG OVER 函数

LEAD/LAG(expr [,offset [,DEFAULT]]) OVER([PARTITION BY col_name1,col_name2,…] ORDER BY col_name1 [ASC/DESC], col_name2 [ASC/DESC],…)

功能描述:

支持 OLAP 函数 LEAD() OVER()、LAG() OVER(),这两个函数是偏移量函数,用来查出同一字段下 N 个值或上 N 个值,并作为新的列存在表中,LEAD 向下偏移,LAG 向上偏移(N 为非负整数)。

使用说明:

1) expr: 此参数是求偏移量的表达式。

2) offset: 此参数是偏移量,可以省略,默认值是 1。

3) default: 此参数是缺省值,可以省略,默认值是 NULL。

4) 这两个函数 OVER 里面的规则同 RANK 类 OLAP 函数一样。

5) 该函数可以返回任何支持的数据类型。

使用约束和限制:

对参数的限制:

参数二和参数三必须是常量或常量表达式。

数据类型转换:

第一个参数与第三个参数的数据类型不同时,第三个参数会根据第一个参数的数据类型做隐式转换。

第一个参数	第三个参数	转换后

第一个参数	第三个参数	转换后
VARCHAR(2)	VARCHAR(20)	VARCHAR(20)
INT	DECIMAL	INT(四舍五入)
DECIMAL	INT	DECIMAL
DATE	DATETIME/TIMESTAMP	DATE
DATETIME	DATE	DATETIME
INT	INT(非数值行)	0
INT	DATE	INT OR BIGINT
INT	DATE	VARCHAR
DATE	VARCHAR	符合日期合适的 INT 数值转换成对应的日期，不符合的转成 NULL 并报 warnings
DATE	INT	符合日期合适的 INT 数值转换成对应的日期，不符合的转成 NULL 并报 warnings
INT	DATE	0

示例 1：LEAD(result, 1, NULL) OVER(PARTITION BY result ORDER BY area DESC)

```
gbase> DROP TABLE IF EXISTS t_olap;
Query OK, 0 rows affected

gbase> CREATE TABLE t_olap(result int, area varchar(200));
Query OK, 0 rows affected

gbase> INSERT INTO t_olap VALUES(550,'天津'),(600,'湖北'),(670,'湖北');
Query OK, 3 rows affected
Records: 3  Duplicates: 0  Warnings: 0

gbase> INSERT INTO t_olap VALUES(448,'天津'),(490,'北京'),(598,'天津
'),('700','湖北');
Query OK, 4 rows affected
Records: 4  Duplicates: 0  Warnings: 0

gbase> INSERT INTO t_olap VALUES(528,'天津'),(446,'北京'),(568,'天津
```

```
'),('682','湖北');
Query OK, 4 rows affected
Records: 4  Duplicates: 0  Warnings: 0

gbase> SELECT * FROM t_olap GROUP BY area,result ORDER BY area,result;
+--------+--------+
| result | area   |
+--------+--------+
|    446 | 北京   |
|    490 | 北京   |
|    448 | 天津   |
|    528 | 天津   |
|    550 | 天津   |
|    568 | 天津   |
|    598 | 天津   |
|    600 | 湖北   |
|    670 | 湖北   |
|    682 | 湖北   |
|    700 | 湖北   |
+--------+--------+
11 rows in set

gbase> SELECT *,LEAD(result, 1, NULL) OVER(PARTITION BY result ORDER BY area
DESC) AS LEAD FROM t_olap;
+--------+--------+------+
| result | area   | LEAD |
+--------+--------+------+
|    446 | 北京   | NULL |
|    448 | 天津   | NULL |
|    490 | 北京   | NULL |
|    528 | 天津   | NULL |
|    550 | 天津   | NULL |
|    568 | 天津   | NULL |
|    598 | 天津   | NULL |
|    600 | 湖北   | NULL |
|    670 | 湖北   | NULL |
|    682 | 湖北   | NULL |
```

```
|     700 | 湖北     | NULL |
+--------+--------+------+
11 rows in set
```

3.9 ROWID 函数

ROWID 为转换函数。

功能描述:

ROWID 是表中记录的唯一标识，功能与主键类似，由 server 自动维护，不实际存储。

使用说明:

实现了两种语法：伪列形式和函数形式（两种写法完全等价，不区分大小写）。其中，函数形式为 ROWID(表名)。

示例:

```
SELECT *, ROWID, ROWID(t1) FROM t1;
SELECT * FROM t1 WHERE ROWID = 1;
```

功能说明:

1)ROWID 返回类型为 BIGINT，从 0 开始排号;

2)ROWID 相当于 server 给表自动添加的伪列，可以查询及使用，由 server 自动维护，不实际存储，不需要也不允许用户进行管理（如修改或创建索引等）;

3)ROWID 作为保留字使用，不允许使用 ROWID 作为任何数据对象的名称（不论大小写及是否有单撇）;

4)DML 不会影响原有数据的 ROWID;

5)性能方面，ROWID 与常量的简单比较，如...WHERE ROWID = 1，可以使用智能索引对 DC 进行过滤，支持的比较类型包括：>，>=，<，<=，=，<>，IS NULL，IS NOT NULL，BETWEEN，NOT BETWEEN;

6)仅 Express 引擎支持 ROWID，对于 Express 引擎不支持的语句，GBase 标准引擎执行时（如含有 ROWID）会报错。外部表由于不走 Express 引擎，故外部表不支持 ROWID。

示例 1：ROWID，ROWID(t1)

```
gbase> DROP TABLE IF EXISTS t1;
Query OK, 0 rows affected

gbase> CREATE TABLE t1(i int, j int);
Query OK, 0 rows affected

gbase> INSERT INTO t1
VALUES(2,1),(2,3),(2,3),(2,5),(3,2),(3,2),(3,2),(3,4),(3,1),(3,5);
Query OK, 10 rows affected
Records: 10  Duplicates: 0  Warnings: 0

gbase> SELECT *, ROWID, ROWID(t1) FROM t1;
+------+------+-------+-------+
| i    | j    | ROWID | rowid |
+------+------+-------+-------+
|    2 |    1 |     0 |     0 |
|    2 |    3 |     1 |     1 |
|    2 |    3 |     2 |     2 |
|    2 |    5 |     3 |     3 |
|    3 |    2 |     4 |     4 |
|    3 |    2 |     5 |     5 |
|    3 |    2 |     6 |     6 |
|    3 |    4 |     7 |     7 |
|    3 |    1 |     8 |     8 |
|    3 |    5 |     9 |     9 |
+------+------+-------+-------+
10 rows in set
```

示例 2：ROWID = 1

```
gbase> DROP TABLE IF EXISTS t1;
Query OK, 0 rows affected
```

```
gbase> CREATE TABLE t1(i int, j int);
Query OK, 0 rows affected

gbase> INSERT INTO t1
VALUES(2,1),(2,3),(2,3),(2,5),(3,2),(3,2),(3,2),(3,4),(3,1),(3,5);
Query OK, 10 rows affected
Records: 10  Duplicates: 0  Warnings: 0

gbase> SELECT * FROM t1 WHERE ROWID = 1;
+------+------+
| i    | j    |
+------+------+
|    2 |    3 |
+------+------+
1 row in set
```

4 SQL 语法

本章主要介绍 GBase 8a 支持的 SQL 语法。

4.1 DDL 语句

介绍 GBase 8a 用于创建数据库对象的 DDL 语句。

4.1.1 DATABASE

4.1.1.1 CREATE DATABASE

CREATE DATABASE [IF NOT EXISTS] *database_name*

CREATE DATABASE 是以给定的名称创建一个数据库。用户需要获得创建数据库的权限，才可以使用 CREATE DATABASE。

示例 1：创建数据库。

```
gbase> CREATE DATABASE IF NOT EXISTS mydb;
Query OK, 1 row affected
```

4.1.1.2 DROP DATABASE

DROP DATABASE [IF EXISTS] *database_name*

DROP DATABASE 删除指定的数据库以及它所包含的表。请小心使用此语句！用户需要获得对数据库的 DROP 权限，才可以使用 DROP DATABASE。

使用关键字 IF EXISTS，以防止由于数据库不存在而报告错误。

示例 1：删除数据库。

```
gbase> DROP DATABASE IF EXISTS test;
Query OK, 1 row affected
```

4.1.2 TABLE

在本节中用到的术语如下。

- 临时表：创建表时使用 TEMPORARY 关键字，这样创建的表为临时表，临时表仅存在于当前 session 中。
- 预租磁盘：预租磁盘空间可以预先批量分配磁盘块，这样尽量保证了列的 DC 数据文件磁盘块连续。在顺序读取列 DC 数据时，性能会有明显提升。

4.1.2.1 CREATE TABLE

CREATE TABLE 以用户给定的名字在当前数据库创建一个表。用户必须有创建表的权限。

```
CREATE [TEMPORARY] TABLE [IF NOT EXISTS] [database_name.]table_name
(column_definition [ , column_definition],  ... [,  key_options])
[table_options]

column_definition:
column_name data_type [NOT NULL | NULL] [DEFAULT default_value]

table_options:
[COMMENT 'comment_value']
```

参数说明如下：

- TEMPORARY：该参数为可选参数，创建临时表需要使用此关键字。临时表的创建请参见“4.1.2.1.3 CREATE TEMPORARY TABLE”的内容。
- IF NOT EXISTS:该参数为可选参数,用户可以使用关键字 IF NOT EXISTS 创建表，如果表已经存在，系统将报告 WARNING 信息。
- *database_name*：该参数为可选参数，指定数据库后，在此数据库下创建表。如果没有显示指定 *database_name* 参数，创建的表隶属于 USE *database_name* 后的数据中的表。
- *table_name*:表命名规则请参见“2.2 数据库、表、索引、列和别名”。默认情况下，在当前数据库中创建表。如果没有指定当前数据库或表已经存在，则报告错误信息。
- column_name：指定表中的数据列。
- data_type：指定数据列的数据类型。数据类型参见“1 数据类型”中的内容。
- NOT NULL | NULL：指定数据列的值，是否允许为 NULL。如果既没有指定 NULL 也没有指定 NOT NULL，列被视为指定了 NULL。
- *default_value*：指定数据列的默认值。默认值必须是一个常数，而不能是一个函数或者一个表达式。举例来说，用户不能将一个数据列的默认值设置为 NOW()或者 CURRENT_DATE()之类的函数。对于给定的一个表，可以使用 SHOW CREATE TABLE 语句来查看哪些列有显式 DEFAULT 子句。
- *comment_value*：指定数据列的备注说明。例如：stu_no id COMMENT’学号’。

table_options:

COMMENT：指定表的备注说明。可以用 SHOW CREATE TABLE *table_name* 和 SHOW FULL COLUMNS FROM *table_name* 语句来显示备注信息。

示例 1：创建一张普通表。

```
gbase> DROP TABLE IF EXISTS t1;
Query OK, 0 rows affected

gbase> CREATE TABLE t1(a int , b varchar(50) NOT NULL DEFAULT 'gbase');
Query OK, 0 rows affected

gbase> DESC t1;
+-------+-------------+------+-----+---------+-------+
| Field | Type        | Null | Key | Default | Extra |
+-------+-------------+------+-----+---------+-------+
| a     | int(11)     | YES  |     | NULL    |       |
| b     | varchar(50) | NO   |     | gbase   |       |
+-------+-------------+------+-----+---------+-------+
2 rows in set
```

示例 2：创建带有列注释信息的表。

```
gbase> DROP TABLE IF EXISTS t1;
Query OK, 0 rows affected

gbase> CREATE TABLE t1 (f_username varchar(10) comment 'name' );
Query OK, 0 rows affected

gbase> DESC t1;
+------------+-------------+------+-----+---------+-------+
| Field      | Type        | Null | Key | Default | Extra |
+------------+-------------+------+-----+---------+-------+
| f_username | varchar(10) | YES  |     | NULL    |       |
+------------+-------------+------+-----+---------+-------+
1 row in set
```

```
gbase> SHOW CREATE TABLE t1;
+-------+-----------------------------------------------+
| Table | Create Table                                  |
+-------+-----------------------------------------------+
| t1    | CREATE TABLE "t1" (
  "f_username" varchar(10) DEFAULT NULL COMMENT 'name'
)                                                       |
+-------+-----------------------------------------------+
1 row in set
```

4.1.2.1.1 CREATE TABLE...AS SELECT...

语法格式:

CREATE TABLE *table_name*[(*column_definition*,...)] AS SELECT ...

功能:

根据列定义以及投影列创建表结构，并且将 SELECT 中查询的数据复制到所创建的表中。

参数说明如下:

AS: 指定 SELECT 语句，可选关键字。

其他参数说明请参见“4.1.2.1 CREATE TABLE”参数说明部分的内容。

示例 1: 复制全表表结构及数据来创建普通表。

```
gbase> DROP TABLE IF EXISTS t7;
Query OK, 0 rows affected

gbase> CREATE TABLE t7(a int, b decimal, c float, d datetime);
Query OK, 0 rows affected

gbase> INSERT INTO t7 VALUES(1,2.234,3.345,'2011-11-11
11:11:11'),(3,4.567,5.678,'2011-11-11 22:22:22');
Query OK, 2 rows affected
```

```
Records: 2  Duplicates: 0  Warnings: 2

gbase> DROP TABLE IF EXISTS t8;
Query OK, 0 rows affected

gbase> CREATE TABLE t8 SELECT * FROM t7;
Query OK, 2 rows affected
Records: 2  Duplicates: 0  Warnings: 0

gbase> SELECT * FROM t8;
+------+------+-------+---------------------+
| a    | b    | c     | d                   |
+------+------+-------+---------------------+
|    1 |    2 | 3.345 | 2011-11-11 11:11:11 |
|    3 |    5 | 5.678 | 2011-11-11 22:22:22 |
+------+------+-------+---------------------+
2 rows in set
```

示例 2：按列复制部分表结构及数据创建普通表。

```
gbase> DROP TABLE IF EXISTS t9;
Query OK, 0 rows affected

gbase> CREATE TABLE t9 SELECT a,b FROM t7;
Query OK, 2 rows affected
Records: 2  Duplicates: 0  Warnings: 0

gbase> SELECT * FROM t9;
+------+------+
| a    | b    |
+------+------+
|    1 |    2 |
|    3 |    5 |
+------+------+
2 rows in set
```

示例 3：按照条件过滤复制部分表结构及数据创建普通表。

```
gbase> DROP TABLE IF EXISTS t10;
Query OK, 0 rows affected

gbase> CREATE TABLE t10 SELECT a,b FROM t7 where d>'2011-11-11 11:11:11';
Query OK, 1 row affected
Records: 1  Duplicates: 0  Warnings: 0

gbase>  SELECT * FROM t10;
+------+------+
| a    | b    |
+------+------+
|    3 |    5 |
+------+------+
1 row in set
```

示例 4：按照联合查询的结果复制部分表结构及数据创建普通表。

```
gbase> DROP TABLE IF EXISTS t11;
Query OK, 0 rows affected

gbase> CREATE TABLE t11 SELECT a,b FROM t7 where d>'2011-11-11 11:11:11' UNION
ALL SELECT a,b FROM t7 where d='2011-11-11 11:11:11';
Query OK, 2 rows affected
Records: 2  Duplicates: 0  Warnings: 0

gbase> SELECT * FROM t11;
+------+------+
| a    | b    |
+------+------+
|    3 |    5 |
|    1 |    2 |
+------+------+
2 rows in set
```

4.1.2.1.2 CREATE TABLE...LIKE...

语法格式:

CREATE TABLE table_name1 LIKE table_name2;

功能:

复制 table_name2 的表结构来创建表 *table_name1*。

示例 1: 创建普通表。

```
gbase> DROP TABLE IF EXISTS t5;
Query OK, 0 rows affected

gbase> CREATE TABLE t5(a int,b datetime);
Query OK, 0 rows affected

gbase> INSERT INTO t5 VALUES(1,NOW());
Query OK, 1 row affected

gbase> CREATE TABLE t6 LIKE t5;
Query OK, 0 rows affected

gbase> SHOW CREATE TABLE t6;
+-------+-----------------------------------------------------------+
| Table | Create Table                                              |
+-------+-----------------------------------------------------------+
| t6    | CREATE TABLE "t6" (
  "a" int(11) DEFAULT NULL,
  "b" datetime DEFAULT NULL
) ENGINE=EXPRESS DEFAULT CHARSET=utf8 TABLESPACE='sys_tablespace' |
+-------+-----------------------------------------------------------+
1 row in set

gbase> SELECT * FROM t6;
Empty set
```

4.1.2.1.3 CREATE TEMPORARY TABLE ...

功能：

在创建一个表时，用户可以使用关键词 TEMPORARY。临时表被限制在当前连接中，当连接关闭时，临时表会自动地删除。这就意味着，两个不同的连接可以使用同一个临时表名而不会发生冲突，也不会与同名现有的表冲突（现有表将被隐藏，直到临时表被删除）。使用此种方法，一旦客户端与 GBase 8a server 断开连接，临时表将自动删除。

注意事项：

- 临时表支持除 ALTER 之外的所有 DDL 及 DML 操作。
- 临时表不能被备份。
- 临时表支持在当前连接中使用查询结果导出语句导出表中数据。

示例 1：创建临时表。

```
gbase> CREATE TEMPORARY TABLE tem_table (a int);
Query OK, 0 rows affected

gbase> INSERT INTO tem_table (a) values (1);
Query OK, 1 row affected

gbase> INSERT INTO tem_table (a) values (2);
Query OK, 1 row affected

gbase> INSERT INTO tem_table (a) values (7);
Query OK, 1 row affected

gbase> INSERT INTO tem_table (a) values (9);
Query OK, 1 row affected

gbase> SELECT * FROM tem_table;
+------+
| a    |
+------+
```

```
|    1 |
|    2 |
|    7 |
|    9 |
+------+
4 rows in set

gbase> EXIT;
Bye
$ gbase -uroot -p
Enter password:

GBase client 8.5.1.2 build 27952. Copyright (c) 2004-2013, GBase.  All Rights
Reserved.

gbase> USE test;
Query OK, 0 rows affected

gbase> SELECT * FROM tem_table;
ERROR 1146 (42S02): Table 'test.tem_table' doesn't exist
```

4.1.2.2 ALTER TABLE

```
ALTER TABLE [database_name.]table_name
    alter_specification [, alter_specification] ...
alter_specification:
  ADD [COLUMN] column_definition [FIRST | AFTER col_name ]
  | ADD [COLUMN] (column_definition,...)
  | CHANGE [COLUMN] old_col_name new_col_name column_definition
  | MODIFY [COLUMN] col_name column_definition
        [FIRST | AFTER col_name]
```

| RENAME [TO] *new_table_name*

| DROP [COLUMN] *col_name*

参数说明如下：

- | ADD [COLUMN] (column_definition,...)：用于增加新的数据列，如果使用 FIRST，则新增加的列位于所有数据列的前面；如果使用 AFTER，则新增加的列，位于指定数据列的后面。默认不使用 FIRST、AFTER，则将增加的新列追加到末尾处。
- CHANGE [COLUMN] *old_col_name new_col_name* column_definition：修改列名称。不支持修改列定义。
- MODIFY [COLUMN] col_name column_definition FIRST | AFTER *col_name* ：修改表中存在列的位置。不支持修改列定义。
- RENAME [TO] new_table_name：修改表名称为 new_table_name。
- DROP [COLUMN] *col_name*：删除表中存在的列。

注意事项：

目前已经支持的有：增加列、删除列、改表名、改列名、改变列的位置；

不支持的有：ORDER BY、改变列的数据类型、改变列的属性（NOT NULL，默认值、注释）、改变表的字符集。

新增列的限制有：对于新增加的列，如果设置了 NOT NULL，则需要同时设置默认值，否则报错；不允许非 EXPRESS 引擎的表与 EXPRESS 表互转。

示例 1：增加列。

```
gbase> DROP TABLE IF EXISTS t;
Query OK, 0 rows affected

gbase> CREATE TABLE t (a int NOT NULL DEFAULT 1, b varchar(10));
```

```
Query OK, 0 rows affected

gbase> DESC t;
+-------+-------------+------+-----+---------+-------+
| Field | Type        | Null | Key | Default | Extra |
+-------+-------------+------+-----+---------+-------+
| a     | int(11)     | NO   |     | 1       |       |
| b     | varchar(10) | YES  |     | NULL    |       |
+-------+-------------+------+-----+---------+-------+
2 rows in set

gbase> ALTER TABLE t ADD column c varchar(10) null;
Query OK, 0 rows affected
Records: 0  Duplicates: 0  Warnings: 0

gbase> DESC t;
+-------+-------------+------+-----+---------+-------+
| Field | Type        | Null | Key | Default | Extra |
+-------+-------------+------+-----+---------+-------+
| a     | int(11)     | NO   |     | 1       |       |
| b     | varchar(10) | YES  |     | NULL    |       |
| c     | varchar(10) | YES  |     | NULL    |       |
+-------+-------------+------+-----+---------+-------+
3 rows in set
```

示例 2：删除列。

```
gbase> DROP TABLE IF EXISTS t;
Query OK, 0 rows affected

gbase> CREATE TABLE t (a int NOT NULL DEFAULT 1, b varchar(10));
Query OK, 0 rows affected

gbase> DESC t;
+-------+-------------+------+-----+---------+-------+
| Field | Type        | Null | Key | Default | Extra |
+-------+-------------+------+-----+---------+-------+
```

```
| a     | int(11)      | NO   |     | 1       |       |
| b     | varchar(10)  | YES  |     | NULL    |       |
+-------+--------------+------+-----+---------+-------+
2 rows in set

gbase> ALTER TABLE t DROP column b;
Query OK, 0 rows affected
Records: 0  Duplicates: 0  Warnings: 0

gbase> DESC t;
+-------+---------+------+-----+---------+-------+
| Field | Type    | Null | Key | Default | Extra |
+-------+---------+------+-----+---------+-------+
| a     | int(11) | NO   |     | 1       |       |
+-------+---------+------+-----+---------+-------+
1 row in set
```

示例 3：变更表名称。

```
gbase> SHOW TABLES;
+--------------+
| Tables_in_ty |
+--------------+
| t            |
+--------------+
1 row in set

gbase> ALTER TABLE t RENAME ttt2;
Query OK, 0 rows affected

gbase> SHOW TABLES;
+--------------+
| Tables_in_ty |
+--------------+
| ttt2         |
+--------------+
1 row in set
```

示例 4：变更列名 b 为新列名 d。

```
gbase> CREATE TABLE t (a int not null,b varchar(10),c varchar(10));
Query OK, 0 rows affected

gbase> DESC t;
+-------+-------------+------+-----+---------+-------+
| Field | Type        | Null | Key | Default | Extra |
+-------+-------------+------+-----+---------+-------+
| a     | int(11)     | NO   |     | NULL    |       |
| b     | varchar(10) | YES  |     | NULL    |       |
| c     | varchar(10) | YES  |     | NULL    |       |
+-------+-------------+------+-----+---------+-------+
3 rows in set

gbase> ALTER TABLE t  CHANGE  b  d varchar(10);
Query OK, 0 rows affected
Records: 0  Duplicates: 0  Warnings: 0

gbase> DESC t;
+-------+-------------+------+-----+---------+-------+
| Field | Type        | Null | Key | Default | Extra |
+-------+-------------+------+-----+---------+-------+
| a     | int(11)     | NO   |     | NULL    |       |
| d     | varchar(10) | YES  |     | NULL    |       |
| c     | varchar(10) | YES  |     | NULL    |       |
+-------+-------------+------+-----+---------+-------+
3 rows in set
```

示例 5：变更列的位置至最前。

```
gbase> DROP TABLE IF EXISTS t;
Query OK, 0 rows affected

gbase> CREATE TABLE t(a int ,b varchar(10),c bool);
Query OK, 0 rows affected
```

```
gbase> DESC t;
+-------+------------+------+-----+---------+-------+
| Field | Type       | Null | Key | Default | Extra |
+-------+------------+------+-----+---------+-------+
| a     | int(11)    | YES  |     | NULL    |       |
| b     | varchar(10)| YES  |     | NULL    |       |
| c     | tinyint(1) | YES  |     | NULL    |       |
+-------+------------+------+-----+---------+-------+
3 rows in set

gbase> ALTER TABLE t  MODIFY b varchar(10) FIRST;
Query OK, 0 rows affected
Records: 0  Duplicates: 0  Warnings: 0

gbase> DESC t;
+-------+------------+------+-----+---------+-------+
| Field | Type       | Null | Key | Default | Extra |
+-------+------------+------+-----+---------+-------+
| b     | varchar(10)| YES  |     | NULL    |       |
| a     | int(11)    | YES  |     | NULL    |       |
| c     | tinyint(1) | YES  |     | NULL    |       |
+-------+------------+------+-----+---------+-------+
3 rows in set
```

示例 6：变更某列的位置到指定列的后面。

```
gbase> ALTER TABLE t MODIFY b varchar(10) AFTER c;
Query OK, 0 rows affected
Records: 0  Duplicates: 0  Warnings: 0

gbase> DESC t;
+-------+------------+------+-----+---------+-------+
| Field | Type       | Null | Key | Default | Extra |
+-------+------------+------+-----+---------+-------+
| a     | int(11)    | YES  |     | NULL    |       |
| c     | tinyint(1) | YES  |     | NULL    |       |
```

```
| b      | varchar(10) | YES  |      | NULL    |       |
+-------+-------------+------+-----+---------+-------+
3 rows in set
```

4.1.2.2.1 ALTER TABLE... SHRINK SPACE

语法格式:

ALTER TABLE [*database_names*.]*tbl_name* SHRINK SPACE

功能:

释放被删除的数据文件所占的磁盘空间。

注意事项:

索引文件及 rowid 占用的磁盘空间不会被回收。

磁盘空间回收命令仅针对表。

数据文件所占的磁盘空间回收以文件为单位，只有当这个数据文件涉及的数据都被删除后才能回收该文件占用的磁盘空间。

如果被删除的数据只命中 DC 的部分数据，则该数据文件不能被清理。

如果删除数据命中所有数据，则有尾块数据的文件不被清理。

磁盘空间回收过程中需要一定的磁盘空间来备份部分元数据文件，在没有可用空间的情况下执行该命令会报错，这时需要手工清理一部分空间（一般需要 1G 空间）再执行该命令进行空间回收。

示例:

示例 1：释放被删除的数据文件占有的磁盘空间。

示例中用到的表及数据如下:

```
DROP TABLE IF EXISTS lineitem;
CREATE TABLE lineitem (
        l_orderkey bigint,
```

```
        l_partkey bigint,
        l_suppkey bigint,
        l_linenumber bigint,
        l_quantity decimal(15,2),
        l_extendedprice decimal(15,2),
        l_discount decimal(15,2),
        l_tax decimal(15,2),
        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(50)
)DISTRIBUTED BY ('l_orderkey');
```

加载数据到 lineitem 表中：

lineitem.ctl 文件如下：

```
[lineitem]
disp_server=192.168.103.104:8888
file_list=/opt/data_gbase/tpch/lineitem.tbl
format=0
db_user=gbase
db_name=test
table_name=lineitem
delimiter='|'
extra_loader_args=--parallel=4 --timeout=0
socket=/tmp/gbase_8a_5050.sock
```

查看 node1 数据文件：

```
# ll /opt/gnode/userdata/gbase/test/sys_tablespace/lineitem_n1
总用量 15625296
-rw-rw---- 1 gbase gbase 540100801 11月 25 17:57 C00000.seg.1
-rw-rw---- 1 gbase gbase 540100801 11月 25 17:57 C00001.seg.1
```

```
-rw-rw---- 1 gbase gbase 540100801 11月 25 17:57 C00002.seg.1
-rw-rw---- 1 gbase gbase 135057661 11月 25 17:57 C00003.seg.1
-rw-rw---- 1 gbase gbase 270072041 11月 25 17:57 C00004.seg.1
-rw-rw---- 1 gbase gbase 540100801 11月 25 17:57 C00005.seg.1
-rw-rw---- 1 gbase gbase 135057661 11月 25 17:57 C00006.seg.1
-rw-rw---- 1 gbase gbase 135057661 11月 25 17:57 C00007.seg.1
-rw-rw---- 1 gbase gbase 405160617 11月 25 17:57 C00008.seg.1
-rw-rw---- 1 gbase gbase 405160617 11月 25 17:57 C00009.seg.1
-rw-rw---- 1 gbase gbase 1080158321 11月 25 17:57 C00010.seg.1
-rw-rw---- 1 gbase gbase 1080158321 11月 25 17:57 C00011.seg.1
-rw-rw---- 1 gbase gbase 1080158321 11月 25 17:57 C00012.seg.1
-rw-rw---- 1 gbase gbase 1990720125 11月 25 17:16 C00013.seg.1
-rw-rw---- 1 gbase gbase 1654785612 11月 25 17:57 C00013.seg.3
-rw-rw---- 1 gbase gbase 1620290037 11月 25 17:57 C00014.seg.1
-rw-rw---- 1 gbase gbase 1981676173 11月 25 17:14 C00015.seg.1
-rw-rw---- 1 gbase gbase 1866271509 11月 25 17:57 C00015.seg.3
```

```
gbase> SELECT COUNT(*) FROM lineitem;
+-----------+
| COUNT(*)  |
+-----------+
| 600037902 |
+-----------+
1 row in set
```

```
gbase> SELECT MIN(rowid),MAX(rowid) FROM lineitem;
+------------+------------+
| min(rowid) | max(rowid) |
+------------+------------+
|          0 | 135037279  |
+------------+------------+
1 row in set
```

删除数据后释放磁盘空间:

```
gbase> DELETE FROM lineitem WHERE rowid <130000000;
Query OK, 580028668 rows affected
```

```
gbase> SELECT COUNT(*) FROM lineitem;
+----------+
| count(*) |
+----------+
| 20009234 |
+----------+
1 row in set
```

```
gbase> ALTER TABLE lineitem SHRINK SPACE;
Query OK, 0 rows affected
```

查看 node1 数据文件：

```
# ll /opt/gnode/userdata/gbase/test/sys_tablespace/lineitem_n1
总用量 11745992
-rw-rw---- 1 gbase gbase 540100801 11 月 25 17:57 C00000.seg.1
-rw-rw---- 1 gbase gbase 540100801 11 月 25 17:57 C00001.seg.1
-rw-rw---- 1 gbase gbase 540100801 11 月 25 17:57 C00002.seg.1
-rw-rw---- 1 gbase gbase 135057661 11 月 25 17:57 C00003.seg.1
-rw-rw---- 1 gbase gbase 270072041 11 月 25 17:57 C00004.seg.1
-rw-rw---- 1 gbase gbase 540100801 11 月 25 17:57 C00005.seg.1
-rw-rw---- 1 gbase gbase 135057661 11 月 25 17:57 C00006.seg.1
-rw-rw---- 1 gbase gbase 135057661 11 月 25 17:57 C00007.seg.1
-rw-rw---- 1 gbase gbase 405160617 11 月 25 17:57 C00008.seg.1
-rw-rw---- 1 gbase gbase 405160617 11 月 25 17:57 C00009.seg.1
-rw-rw---- 1 gbase gbase 1080158321 11 月 25 17:57 C00010.seg.1
-rw-rw---- 1 gbase gbase 1080158321 11 月 25 17:57 C00011.seg.1
-rw-rw---- 1 gbase gbase 1080158321 11 月 25 17:57 C00012.seg.1
-rw-rw---- 1 gbase gbase 1654785612 11 月 25 17:57 C00013.seg.3
-rw-rw---- 1 gbase gbase 1620290037 11 月 25 17:57 C00014.seg.1
-rw-rw---- 1 gbase gbase 1866271509 11 月 25 17:57 C00015.seg.3
```

4.1.2.3 RENAME TABLE

语法格式：

RENAME TABLE [database_name.]old_table_name TO [database_name.]new_table_name;

参数说明如下：

database_name：是要修改表隶属的数据库名称，可选项；省略此参数，即为 USE 后的数据库名称。

old_table_name：是表的原有名称。

new_table_name：是表的要修改后的新名称。

功能：

RENAME TABLE 的功能上就是将一张已经存在的表的名称修改为一个不存在的新的表名称。

示例 1：更改表 n 的名称为 m。

```
gbase> SELECT table_schema,table_name,table_rows FROM
information_schema.tables WHERE table_schema='test' AND table_name = 'n';
+--------------+------------+------------+
| table_schema | table_name | table_rows |
+--------------+------------+------------+
| test         | n          |          0 |
+--------------+------------+------------+
1 row in set

gbase> USE test;
Query OK, 0 rows affected

gbase> RENAME TABLE test.n to test.m;
Query OK, 0 rows affected

gbase> SELECT table_schema,table_name,table_rows FROM
information_schema.tables WHERE table_schema='test' AND table_name = 'm';
+--------------+------------+------------+
| table_schema | table_name | table_rows |
```

```
+-------------+-----------+-----------+
| test        | m         |         0 |
+-------------+-----------+-----------+
1 row in set
```

4.1.2.4 TRUNCATE TABLE

语法格式:

TRUNCATE TABLE [database_name.]table_name;

参数说明如下:

database_name: 是要删除表隶属的数据库名称，可选项；省略此参数，即为 USE 后的数据库名称。

table_name: 是要删除其全部行的表的名称。

功能:

TRUNCATE TABLE 在功能上与不带 WHERE 子句的 DELETE 语句相同，二者均删除表中的全部行。但 TRUNCATE TABLE 比 DELETE 速度快，且使用的系统和事务日志资源少。

TRUNCATE TABLE 属于 DDL 语法，DELETE FROM table_name 属于 DML 语法。

TRUNCATE TABLE 删除表中的所有行，但表结构及其列、约束、索引等保持不变。

示例 1: 删除表 b 中的所有行。

```
gbase> USE test;
Query OK, 0 rows affected

gbase> CREATE TABLE t (a decimal(12,5) DEFAULT NULL, KEY idx_a (a) USING HASH
LOCAL);
Query OK, 0 rows affected
```

```
gbase> INSERT INTO t VALUES(1),(2),(3);
Query OK, 3 rows affected
Records: 3  Duplicates: 0  Warnings: 0

gbase> SELECT * FROM t;
+---------+
| a       |
+---------+
| 1.00000 |
| 2.00000 |
| 3.00000 |
+---------+
3 rows in set

gbase> TRUNCATE TABLE t;
Query OK, 3 rows affected

gbase> SELECT * FROM t;
Empty set
```

4.1.2.5 DROP TABLE

语法格式:

DROP [TEMPORARY] TABLE [IF EXISTS] [*database_name.*]*table_name*

参数说明如下:

TEMPORARY: 该参数为可选参数，删除临时表时建议使用此关键字。

IF EXISTS: 用户可以使用关键词 IF EXISTS 防止表不存在时报告错误。当使用 IF EXISTS 时，对于不存在的表，用户将得到一个 WARNING。

使用 DROP TABLE 移除一个表时，将移除所有的数据和表定义，用户必须有表的 DROP 权限，所以，一定要小心地使用这个命令!

4.1.3 VIEW

视图是一个虚拟表，其内容由查询定义。同真实的表一样，视图包含一系列带有名称的列。数据来自由定义视图的查询所引用的表，并且在引用视图时动态生成。对其中所引用的基础表来说，视图的作用类似于筛选。定义视图的筛选可以来自当前或其它数据库的一个或多个表，或者其它视图。分布式查询也可用于定义使用多个异类源数据的视图。

需要注意的是：GBase 8a 禁止对视图进行 INSERT、UPDATE 和 DELETE 操作。

视图具有以下的作用：

简单性。看到的就是需要的。视图不仅可以简化用户对数据的理解，也可以简化他们的操作。那些被经常使用的查询可以被定义为视图，从而使得用户不必为以后的操作每次指定全部的条件。

安全性。通过视图用户只能查询和修改他们所能见到的数据。数据库中的其它数据则既看不见也取不到。数据库授权命令可以使每个用户对数据库的检索限制到特定的数据库对象上，但不能授权到数据库特定行和特定的列上。通过视图，用户可以被限制在数据的不同子集上。

4.1.3.1 CREATE VIEW

语法格式：

CREATE [OR REPLACE] VIEW [*database_name*.]*view_name* [(column_list)] AS select_statement

此语句用来创建一个新的视图，或者使用 OR REPLACE 子句来替换已经存在的视图。

select_statement 是提供给定义视图的 SELECT 语句。本语句可以从其它表或者视图中提取数据。

此语句需要针对视图的 CREATE VIEW 权限，以及构成视图的 SELECT 语句中引用列的部分权限。对于在 SELECT 语句中要使用的列必需要有 SELECT 权限。如果使用了 OR REPLACE 子句，还必须有删除视图的权限。

视图属于数据库组件之一，在默认情况下，一个新的视图创建于当前数据库中。如果要在给定的数据库中显示地创建视图，请在创建时指定 [*database_name*.]*view_name*。

示例 1：创建视图。

```
gbase> DROP TABLE IF EXISTS product;
Query OK, 0 rows affected

gbase> CREATE TABLE product (quantity INT,price INT);
Query OK, 0 rows affected

gbase> INSERT INTO product VALUES(3,50);
Query OK, 1 row affected

gbase> CREATE VIEW product_v AS SELECT quantity,price,quantity*price FROM
product;
Query OK, 0 rows affected

gbase> SELECT * FROM product_v;
+----------+-------+----------------+
| quantity | price | quantity*price |
+----------+-------+----------------+
|        3 |    50 |            150 |
+----------+-------+----------------+
1 row in set
```

4.1.3.2 ALTER VIEW

语法格式：

ALTER VIEW [*database_name*.]*view_name* [(column_list)] AS

select_statement

示例 1：修改视图 v_t 中的列为指定列。

```
gbase> DESC v_t;
+---------+-------------+------+-----+---------+-------+
| Field   | Type        | Null | Key | Default | Extra |
+---------+-------------+------+-----+---------+-------+
| name    | varchar(20) | YES  |     | NULL    |       |
| address | varchar(40) | YES  |     | NULL    |       |
| sex     | int(11)     | YES  |     | NULL    |       |
+---------+-------------+------+-----+---------+-------+
3 rows in set

gbase> ALTER VIEW test.v_t(a, b) AS SELECT name,address FROM t;
Query OK, 0 rows affected

gbase> DESC v_t;
+-------+-------------+------+-----+---------+-------+
| Field | Type        | Null | Key | Default | Extra |
+-------+-------------+------+-----+---------+-------+
| a     | varchar(20) | YES  |     | NULL    |       |
| b     | varchar(40) | YES  |     | NULL    |       |
+-------+-------------+------+-----+---------+-------+
2 rows in set

gbase> SELECT * FROM v_t;
+------+----------------------+
| a    | b                    |
+------+----------------------+
| TOM  | east street          |
| jack | west road NO 15      |
| MIKE | DongFang road NO 22  |
| TONY | EA Street            |
| Rose | TangRen Street NO.191|
+------+----------------------+
5 rows in set
```

4.1.3.3 DROP VIEW

语法格式:

DROP VIEW [IF EXISTS] [*database_name*.]*view_name*

DROP VIEW 删除一个视图。用户必须有对每个视图的 DROP 权限。

用户可以使用关键词 IF EXISTS 防止视图不存在时报告错误。

DROP VIEW 每次只能删除一张视图。

示例 1: 删除单张视图。

```
gbase> DROP VIEW IF EXISTS student_v;
Query OK, 0 rows affected
```

4.1.4 INDEX

4.1.4.1 CREATE INDEX

语法格式:

CREATE INDEX *index_name* ON [*database_name*.]*table_name*(*column_name*) [key_block_size = *size_value*] USING HASH [GLOBAL|LOCAL]

index_name: 索引名。

database_name: 数据库名。可省略，省略时，必须首先使用 USE 命令，指定当前数据库。

table_name: 表名。

column_name: 使用索引的列名。

LOCAL: 在表的每一个 DC 上创建哈希索引。

GLOBAL: 默认创建 GLOBAL 的哈希索引。创建全局哈希索引，全局创建索引

针对整列，数据按页存储，每个数据块占用多少个页可以在创建索引时指定。

key_block_size = size_value：当使用 GLOBAL 关键字时，可以配合使用它，这个参数表示指定每个数据块的大小。size_value 值见下表：

最小值	最大值	备注
4096	32768	size_value 必须是 4096 的整数倍

一般来说，二进制类型的列不适合使用 HASH INDEX，或者该列数据量较大，但 DISTINCT 值较少时，也不适合使用 HASH INDEX。

使用 GLOBAL 还是 LOCAL 来定义索引的范围,需要根据实际业务来具体考虑。

同一表上不能创建相同名称的哈希索引，同一表的同一列上能且只能创建一个哈希索引，任何 GBase 8a 支持的数据类型的列上都可以创建哈希索引。

创建哈希索引后，基于索引列的等值查询的性能会提高，尤其是表中的数据量非常大的情况，在小数据量的情况下，哈希索引对性能的提升效果不明显。

示例 1：创建 LOCAL 哈希索引。

```
gbase> DROP TABLE IF EXISTS t1;
Query OK, 0 rows affected

gbase> CREATE TABLE t1(a int,b varchar(10));
Query OK, 0 rows affected

gbase> CREATE INDEX idx1 on t1(a) USING HASH LOCAL;
Query OK, 0 rows affected
Records: 0  Duplicates: 0  Warnings: 0
```

示例 2：创建 GLOBAL 哈希索引。

```
gbase> DROP TABLE IF EXISTS t;
Query OK, 0 rows affected

gbase> CREATE TABLE t (a INT);
Query OK, 0 rows affected
```

```
gbase> CREATE INDEX idx_t_a ON t(a) key_block_size = 4096 USING HASH GLOBAL;
Query OK, 0 rows affected
Records: 0  Duplicates: 0  Warnings: 0

gbase> SHOW CREATE TABLE t;
+-------+-----------------------------------------------------------+
| Table | Create Table                                              |
+-------+-----------------------------------------------------------+
| t     | CREATE TABLE "t" (
  "a" int(11) DEFAULT NULL,
  KEY "idx_t_a" ("a") KEY_BLOCK_SIZE=4096 USING HASH GLOBAL
) ENGINE=EXPRESS DEFAULT CHARSET=utf8 TABLESPACE='sys_tablespace' |
+-------+-----------------------------------------------------------+
1 row in set
```

示例 3：创建 GLOBAL 哈希索引，设置 key_block_size 值为 16384。

```
gbase> CREATE TABLE t1(a int,b varchar(10));
Query OK, 0 rows affected

gbase> CREATE INDEX idx3 on t1(b) key_block_size=16384 USING HASH GLOBAL;
Query OK, 0 rows affected
Records: 0  Duplicates: 0  Warnings: 0
```

4.1.4.2 DROP INDEX

语法格式：

DROP INDEX index_name ON [*database_name*.]*table_name*;

示例 1：删除索引。

```
gbase> DROP INDEX idx3 ON t1;
Query OK, 0 rows affected
Records: 0  Duplicates: 0  Warnings: 0
```

4.1.5 预租磁盘

预租磁盘空间可以预先批量分配磁盘块，这样尽量保证了列的 DC 数据文件磁盘块连续。这样在顺序读取列 DC 数据时，性能会有明显提升。

创建表时，可以指定表的自动扩展大小。当表中的存储数据超过制定的预租大小空间时，系统会自动按照预租磁盘大小空间进行自动扩展。目前预租磁盘空间大小可以按照 MB 和 GB 大小来设定，扩展大小在[1M,2G)范围内。

支持对预租磁盘的 ALTER 的 DDL 操作（关闭预租磁盘空间）。

修改预租磁盘的扩展空间（修改预租磁盘空间大小）。

预租磁盘的预租空间是按照列级增长的。

创建预租磁盘空间的语法：

CREATE TABLE [IF NOT EXISTS] *table_name*

(col type,...)

AUTOEXTEND ON NEXT *NUM*[M/G];

NUM:以 M(megabytes)，G(gigabytes)为单位。

注意以下原则：

NUM 的有效范围为 1M ≤ *NUM* < 2G。

修改预租磁盘空间大小的语法：

ALTER TABLE *table_name* AUTOEXTEND ON NEXT *NUM*[M/G] ;

关闭预租磁盘空间的语法：

ALTER TABLE *table_name* AUTOEXTEND OFF;

示例 1：创建一张表，并指定预租磁盘空间大小。

```
gbase> CREATE TABLE t(nameid int, name varchar(50)) AUTOEXTEND ON NEXT 1M;
Query OK, 0 rows affected
```

```
gbase> SHOW CREATE TABLE t;
+-------+-----------------------------------------------------------------+
| Table | Create Table                                                    |
+-------+-----------------------------------------------------------------+
| t     | CREATE TABLE "t" (
  "nameid" int(11) DEFAULT NULL,
  "name" varchar(50) DEFAULT NULL
) ENGINE=EXPRESS DEFAULT CHARSET=utf8 TABLESPACE='sys_tablespace' AUTOEXTEND
ON NEXT 1M                                                              |
+-------+-----------------------------------------------------------------+
1 row in set
```

示例 2：修改表的指定预租磁盘空间大小。

```
gbase> ALTER TABLE t AUTOEXTEND ON NEXT 2M;
Query OK, 0 rows affected
Records: 0  Duplicates: 0  Warnings: 0

gbase> SHOW CREATE TABLE t;
+-------+-----------------------------------------------------------------+
| Table | Create Table                                                    |
+-------+-----------------------------------------------------------------+
| t     | CREATE TABLE "t" (
  "nameid" int(11) DEFAULT NULL,
  "name" varchar(50) DEFAULT NULL
) ENGINE=EXPRESS DEFAULT CHARSET=utf8 TABLESPACE='sys_tablespace' AUTOEXTEND
ON NEXT 2M                                                              |
+-------+-----------------------------------------------------------------+
1 row in set
```

示例 3：关闭表的指定预租磁盘空间大小。

```
gbase> ALTER TABLE t AUTOEXTEND OFF;
Query OK, 0 rows affected
Records: 0  Duplicates: 0  Warnings: 0

gbase> SHOW CREATE TABLE t;
```

```
+-------+-------------------------------------------------------------+
| Table | Create Table                                                |
+-------+-------------------------------------------------------------+
| t     | CREATE TABLE "t" (
  "nameid" int(11) DEFAULT NULL,
  "name" varchar(50) DEFAULT NULL
) ENGINE=EXPRESS DEFAULT CHARSET=utf8 TABLESPACE='sys_tablespace' |
+-------+-------------------------------------------------------------+
1 row in set
```

示例 4：指定预租磁盘空间大小超出支持范围时，系统提示错误。

```
gbase> CREATE TABLE t1(a int) AUTOEXTEND ON NEXT 3G;
ERROR 1729 (HY000): set table extend failed: must be between 1M and 2G
```

4.1.6 列和表的压缩

4.1.6.1 列级压缩

除了使用 GBase 8a 配置参数可以指定数据存储时的压缩，系统还提供了 DDL 语法，在创建或修改表时，对表中的一列或多列进行数据压缩的定义。方便用户进行单独设置。

使用列压缩定义后，列压缩定义 > 表级定义压缩 > 系统参数全局定义压缩方式。也就是说，列压缩拥有最高级别的语法有效性。

用户在配置文件中打开压缩的时候，使用 3-1 压缩（字符型列-数值型列）时，新建的表将以 3-1 压缩算法存储数据，如果使用本章介绍的语法创建新表或者 ALTER 原有表，指定使用了 5-5 压缩，那么就会按照 5-5 压缩算法存储数据，而不会使用 3-1 压缩算法存储数据。

4.1.6.1.1 创建压缩列

语法格式：

CREATE [TEMPORARY] TABLE [IF NOT EXISTS] [*database_name*.]*table_name*

(column_definition [, column_definition], ... [, key_options])

[table_options]

column_definition:

column_name data_type [NOT NULL | NULL] [DEFAULT *default_value*] COMPRESS (compress_type)

参数说明如下：

compress_type：表中数值类型列的压缩方式。具体如下：

0：不使用压缩

1：对数字类型使用深度压缩

5：对数字类型使用轻度压缩

compress_type：表中字符串类型列的压缩方式。具体如下：

0：不使用压缩

3：对字符串类型使用深度压缩

5：对字符串类型使用轻度压缩

GBase 8a 中表的列数据类型对应使用的列压缩类型如下表所示：

GBase 8a 列类型	列压缩类型	列压缩定义值
tinyint	数值类型列	0，1，5
int	数值类型列	0，1，5
meduimint	数值类型列	0，1，5
bigint	数值类型列	0，1，5
decimal	数值类型列	0，1，5
float	数值类型列	0，1，5

GBase 8a 列类型	列压缩类型	列压缩定义值
double	数值类型列	0，1，5
date	数值类型列	0，1，5
datetime	数值类型列	0，1，5
timestamp	数值类型列	0，1，5
char	字符串类型列	0，3，5
varchar	字符串类型列	0，3，5
blob	字符串类型列	0，3，5
text	字符串类型列	0，3，5

其他参数说明请参见“4.1.2.1 CREATE TABLE”参数说明部分的内容。

示例 1：定义单列的列压缩。

```
gbase> DROP TABLE IF EXISTS t1;
Query OK, 0 rows affected


gbase> CREATE TABLE t1 (a int DEFAULT NULL,b varchar(10) COMPRESS(3));
Query OK, 0 rows affected

gbase> SHOW CREATE TABLE t1;
+-------+-----------------------------------------------------+
| Table | Create Table                                        |
+-------+-----------------------------------------------------+
| t1    | CREATE TABLE "t1" (
  "a" int(11) DEFAULT NULL,
  "b" varchar(10) DEFAULT NULL COMPRESS(3)
) ENGINE=EXPRESS DEFAULT CHARSET=utf8 TABLESPACE='sys_tablespace'|
+-------+-----------------------------------------------------+
1 row in set
```

示例 2：当定义表的列类型和数据压缩类型不一致时，系统提示错误信息。

```
gbase> DROP TABLE IF EXISTS t2;
Query OK, 0 rows affected

gbase> CREATE TABLE t2 (a int compress(3),b varchar(10) compress(5));
```

```
ERROR 1717 (HY000): incorrect compress type: 3.
```

4.1.6.1.2 修改压缩列

语法格式:

ALTER TABLE [IF NOT EXISTS] [*database_name*.]*table_name*;

ALTER [column] column_name COMPRESS (compress_type);

修改列压缩值的语法不能批更新列压缩值。

compress_type: 表中数值类型列的压缩方式。具体如下:

0: 不使用压缩

1: 对数字类型使用深度压缩

5: 对数字类型使用轻度压缩

compress_type: 表中字符串类型列的压缩方式。具体如下:

0: 不使用压缩

1: 忽略, 不压缩 (目前和 0 效果一样)

3: 对字符串类型使用深度压缩

5: 对字符串类型使用轻度压缩

修改列的压缩值后, 修改后的列数据按照新压缩算法进行存储, 修改前的列压缩值存储方式不变。

示例 1: 修改非压缩列为压缩列

```
gbase> SHOW CREATE TABLE t1;
+-------+-----------------------------------------------------+
| Table | Create Table                                        |
+-------+-----------------------------------------------------+
| t1    | CREATE TABLE "t1" (
  "a" int(11) DEFAULT NULL,
```

```
  "b" varchar(10) DEFAULT NULL COMPRESS(3)
)ENGINE=EXPRESS DEFAULT CHARSET=utf8 TABLESPACE='sys_tablespace'|
+-------+-----------------------------------------------------+
1 row in set

gbase> ALTER TABLE t1 ALTER a COMPRESS(1);
Query OK, 0 rows affected
Records: 0  Duplicates: 0  Warnings: 0

gbase> SHOW CREATE TABLE t1;
+-------+-----------------------------------------------------+
| Table | Create Table                                        |
+-------+-----------------------------------------------------+
| t1    | CREATE TABLE "t1" (
  "a" int(11) COMPRESS(1) ,
  "b" varchar(10) DEFAULT NULL COMPRESS(3)
) ENGINE=EXPRESS DEFAULT CHARSET=utf8 TABLESPACE='sys_tablespace'|
+-------+-----------------------------------------------------+
1 row in set
```

示例 2：修改压缩列的压缩类型值。

```
gbase> CREATE TABLE t2 (a int ,b varchar(10) NULL COMPRESS(5));
Query OK, 0 rows affected

gbase> SHOW CREATE TABLE t2;
+-------+-----------------------------------------------------+
| Table | Create Table                                        |
+-------+-----------------------------------------------------+
| t2    | CREATE TABLE "t2" (
  "a" int(11) DEFAULT NULL,
  "b" varchar(10) DEFAULT NULL COMPRESS(5)
) ENGINE=EXPRESS DEFAULT CHARSET=utf8 TABLESPACE='sys_tablespace'|
+-------+-----------------------------------------------------+
1 row in set

gbase> ALTER TABLE t2 ALTER b COMPRESS(3);
```

```
Query OK, 0 rows affected
Records: 0  Duplicates: 0  Warnings: 0

gbase> SHOW CREATE TABLE t2;
+-------+-----------------------------------------------------------+
| Table | Create Table                                              |
+-------+-----------------------------------------------------------+
| t2    | CREATE TABLE "t2" (
  "a" int(11) DEFAULT NULL,
  "b" varchar(10) COMPRESS(3)
) ENGINE=EXPRESS DEFAULT CHARSET=utf8 TABLESPACE='sys_tablespace'|
+-------+-----------------------------------------------------------+
1 row in set
```

示例 3：修改列压缩值为非对应的列压缩，系统报告错误信息。

```
gbase> SHOW CREATE TABLE t2;
+-------+-----------------------------------------------------------+
| Table | Create Table                                              |
+-------+-----------------------------------------------------------+
| t2    | CREATE TABLE "t2" (
  "a" int(11) DEFAULT NULL,
  "b" varchar(10) DEFAULT NULL COMPRESS(5)
) ENGINE=EXPRESS DEFAULT CHARSET=utf8 TABLESPACE='sys_tablespace' |
+-------+-----------------------------------------------------------+
1 row in set
```

--报出错误信息

```
gbase> ALTER TABLE t2 ALTER a compress(3);
ERROR 1717 (HY000): incorrect compress type: 3.
```

4.1.6.2 表级压缩

除了使用配置参数可以指定数据存储时的压缩，系统还提供了 DDL 语法，在创建表时或修改表时，进行数据压缩的定义。方便用户进行单独设置。

使用压缩表定义压缩后，表级定义压缩 > 系统参数全局定义压缩方式。也就是说，表级压缩方式覆盖全局压缩。

用户在配置文件中关闭压缩的时候，创建新表不会使用压缩算法存储。但是在创建新表时指定了表级压缩算法，就会按照指定的压缩算法存储数据。

用户在配置文件中打开压缩时，使用 compress (0,0) 创建或 ALTER 表，不指定压缩算法，则不会使用任何压缩方式存储数据。

4.1.6.2.1 创建压缩表

语法格式：

CREATE [TEMPORARY] TABLE [IF NOT EXISTS] [*database_name.*]*table_name*

(column_definition [, column_definition], ... [, key_options])

COMPRESS (compress_type1,compress_type2)

compress_type1：表中数值类型列的压缩方式。具体如下：

0：不使用压缩

1：对数字类型使用深度压缩

5：对数字类型使用轻度压缩

compress_type2：表中字符串类型列的压缩方式。具体如下：

0：不使用压缩

3：对字符串类型使用深度压缩

5：对字符串类型使用轻度压缩

使用压缩表的主要目的，就是为了节省磁盘空间，提高磁盘存储的使用率。通常 compress_type1 和 compress_type2 压缩组合值为 (1,3) 和 (5,5) 。如果表不使用压缩，compress_type1 和 compress_type2 压缩组合值为 (0,0) 。

示例 1：使用压缩表语法定义表。

```
gbase> DROP TABLE IF EXISTS t1;
Query OK, 0 rows affected

gbase> CREATE TABLE t1 (a int, b varchar(10)) COMPRESS(0,0);
Query OK, 0 rows affected
```

4.1.6.2.2 修改压缩表

语法格式:

ALTER TABLE [IF NOT EXISTS] [*database_name*.]*table_name*;

(column_definition [, column_definition], ... [, key_options])

ALTER COMPRESS (compress_type1,compress_type2);

compress_type1: 表中数值类型列的压缩方式。具体如下:

0: 不使用压缩

1: 对数字类型使用深度压缩

5: 对数字类型使用轻度压缩

compress_type2: 表中字符串类型列的压缩方式。具体如下:

0: 不使用压缩

3: 对字符串类型使用深度压缩

5: 对数字类型使用轻度压缩

示例 1: 修改压缩表的压缩类型。

```
gbase> SHOW CREATE TABLE t2;
+-------+---------------------------------------------+
| Table | Create Table                                |
+-------+---------------------------------------------+
| t2    | CREATE TABLE "t2" (
  "a" int(11) DEFAULT NULL,
```

```
  "b" varchar(10) DEFAULT NULL
) COMPRESS(1, 3)  ENGINE=EXPRESS DEFAULT CHARSET=utf8
TABLESPACE='sys_tablespace'                                    |
+-------+----------------------------------------------+
1 row in set

gbase> ALTER TABLE t2 ALTER compress(5,5);
Query OK, 0 rows affected

gbase> SHOW CREATE TABLE t2;
+-------+-------------------------------------------------------+
| Table | Create Table                                          |
+-------+-------------------------------------------------------+
| t2    | CREATE TABLE "t2" (
  "a" int(11) DEFAULT NULL,
  "b" varchar(10) DEFAULT NULL
) COMPRESS(5, 5)  ENGINE=EXPRESS DEFAULT CHARSET=utf8
TABLESPACE='sys_tablespace'                                    |
+-------+-------------------------------------------------------+
1 row in set
```

4.1.7 行列混存

4.1.7.1 行列混存的定义

由于 GBase 8a 是列存储的架构，因此当列数较多，访问的数据记录又非常离散时，会造成大量的离散 I/O，引起 I/O 性能低下，严重影响执行性能。

GBase 8a 提供行列混存功能，通过冗余行存储可以有效提高 I/O 性能。

行列混存具有以下功能：

支持 SQL 语法，包括建表时定义行存列，对已存在的表创建行存列，删除行存列；

支持快速创建，并行创建行存列；

减少冗余存储，行存列支持压缩存储；

提升 I/O 性能，行存列可以按更小粒度的 Data Page 读取数据，而不是 DC；

系统会自动判断某场景是否需要使用行存数据；

存储冗余方式灵活，用户可自定义数据存储及冗余方式；

行存列维护，DML 语句自动维护行存列，包括，INSERT、快速 UPDATE、DELETE、LOAD 等。

建表语法：

```
CREATE TABLE tablename (column-definitions,
       [GROUPED_DEFINITIONS]
);
GROUPED_DEFINITION:
     GROUPED [grouped_name](column_references)[COMPRESS(num)]
```

参数说明如下：

GROUPED：关键字，表示定义的是行存列。

grouped_name：表示行存列的名称，可以知道行存列名称。如果不指定名称，则默认为后面的 column_references 中第一个列的名称，如果该名称重名，则在名称后面加上“_#”（#为从 2 开始的一个数字）。

column_references：行存列中包含的物理列的集合，各列间以“,”分隔。

COMPRESS(num)：为行存列指定压缩方式，取值为 0、3、5 中的一个。

修改表（创建/删除行存列）语法：

ALTER TABLE *table_name* ADD GROUPED_DEF

ALTER TABLE *table_name* DROP GROUPED *grouped_name*

行存列的创建可以在 CREATE TABLE 时指定，也可以使用 ALTER TABLE...ADD GROUPED 语句。

4.1.7.2 行列混存的约束

行列混存有以下使用约束:

同一表中不允许创建同名的行存列。

同一字段不允许出现在两个行存列定义中。

除删除行存列语句外，行存列不允许在任何语句中被直接引用。

行存列的定义不允许修改，在确实需要修改的情况下，只能先删除，再根据新的定义创建。

行存列定义中包含的物理列不允许删除和修改数据类型，但可以修改列名和列在表中的顺序。

行存列只允许使用 0、3、5 压缩方式，使用其它压缩方式会发生错误。

行存列的名字，不能与行存列所创建的表中的索引名名称重名。

示例 1:建表时创建多个行存列，并指定行存列名称，行存列使用压缩算法。

```
gbase>  DROP TABLE IF EXISTS t;
Query OK, 0 rows affected

gbase> CREATE TABLE t(a int,b int,c int,d int,GROUPED a(b,c) COMPRESS(5));
Query OK, 0 rows affected

gbase> SHOW CREATE TABLE t;
+-------+-----------------------------------------------------+
| Table | Create Table                                        |
+-------+-----------------------------------------------------+
| t     | CREATE TABLE "t" (
  "a" int(11) DEFAULT NULL,
  "b" int(11) DEFAULT NULL,
  "c" int(11) DEFAULT NULL,
  "d" int(11) DEFAULT NULL,
  GROUPED "a" ("b","c") COMPRESS(5)
) ENGINE=EXPRESS DEFAULT CHARSET=utf8 TABLESPACE='sys_tablespace' |
```

```
+-------+-----------------------------------------------------+
```

```
1 row in set
```

示例 2：建表时创建行存列，不指定行存列名称。

```
gbase> DROP TABLE IF EXISTS t;
Query OK, 0 rows affected

gbase> CREATE TABLE t(a int,b int,c int,d int,GROUPED (b,c),GROUPED (d));
Query OK, 0 rows affected

gbase> SHOW CREATE TABLE t;
+-------+-----------------------------------------------------------+
| Table | Create Table                                              |
+-------+-----------------------------------------------------------+
| t     | CREATE TABLE "t" (
  "a" int(11) DEFAULT NULL,
  "b" int(11) DEFAULT NULL,
  "c" int(11) DEFAULT NULL,
  "d" int(11) DEFAULT NULL,
  GROUPED "b" ("b","c"),
  GROUPED "d" ("d")
) ENGINE=EXPRESS DEFAULT CHARSET=utf8 TABLESPACE='sys_tablespace' |
+-------+-----------------------------------------------------------+
1 row in set
```

示例 3：建表时创建行存列，指定部分行存列名称。如果行存列名称重名，则在该名称后面加上“_#”（#为从 2 开始的一个数字）

```
gbase> DROP TABLE IF EXISTS t;
Query OK, 0 rows affected

gbase> CREATE TABLE t(a int,b int,c int,d int,GROUPED a(b,c),GROUPED (a));
Query OK, 0 rows affected

gbase> SHOW CREATE TABLE t;
```

```
+-------+-----------------------------------------------------------+
| Table | Create Table                                              |
+-------+-----------------------------------------------------------+
| t     | CREATE TABLE "t" (
  "a" int(11) DEFAULT NULL,
  "b" int(11) DEFAULT NULL,
  "c" int(11) DEFAULT NULL,
  "d" int(11) DEFAULT NULL,
  GROUPED "a" ("b","c"),
  GROUPED "a_2" ("a")
) ENGINE=EXPRESS DEFAULT CHARSET=utf8 TABLESPACE='sys_tablespace' |
+-------+-----------------------------------------------------------+
1 row in set
```

示例 4：建表时创建多个行存列，重复指定行存列名称，返回错误码 1702。

```
gbase> DROP TABLE IF EXISTS t;
Query OK, 0 rows affected

gbase> CREATE TABLE t (a int,b int,c int,d int, GROUPED a(b,c), GROUPED a(d));
ERROR 1061 (42000): Duplicate key name 'a'
```

示例 5：建表时创建多个行存列，并指定行存列名称，使用压缩表语法。

```
gbase>  DROP TABLE IF EXISTS t;
Query OK, 0 rows affected

gbase> CREATE TABLE t(a int,b int,c int,d int,GROUPED a(b,c)) COMPRESS(5,5);
Query OK, 0 rows affected

gbase> SHOW CREATE TABLE t;
+-------+-----------------------------------------------------------+
| Table | Create Table                                              |
+-------+-----------------------------------------------------------+
| t     | CREATE TABLE "t" (
  "a" int(11) DEFAULT NULL,
  "b" int(11) DEFAULT NULL,
```

```
  "c" int(11) DEFAULT NULL,
  "d" int(11) DEFAULT NULL,
  GROUPED "a" ("b","c")
) COMPRESS(5, 5)  ENGINE=EXPRESS DEFAULT CHARSET=utf8
TABLESPACE='sys_tablespace'                                    |
+-------+-----------------------------------------------------+
```

示例 6：建表时创建多个行存列，并指定行存列名称，行存列和表都使用压缩语法。

```
gbase> DROP TABLE IF EXISTS t;
Query OK, 0 rows affected

gbase> CREATE TABLE t(a int,b int,c int,d int,GROUPED a(b,c) COMPRESS(3))
COMPRESS(5,5);
Query OK, 0 rows affected

gbase> SHOW CREATE TABLE t;
+-------+-------------------------------------------------------------------+
| Table | Create Table                                                      |
+-------+-------------------------------------------------------------------+
| t     | CREATE TABLE "t" (
  "a" int(11) DEFAULT NULL,
  "b" int(11) DEFAULT NULL,
  "c" int(11) DEFAULT NULL,
  "d" int(11) DEFAULT NULL,
  GROUPED "a" ("b","c") COMPRESS(3)
) COMPRESS(5, 5)  ENGINE=EXPRESS DEFAULT CHARSET=utf8 TABLESPACE='sys_tablespace' |
+-------+-------------------------------------------------------------------+
1 row in set
```

示例 7：建表时不创建行存列，使用 ALTER TABLE 语句创建行存列。

```
gbase> DROP TABLE t2;
Query OK, 0 rows affected

gbase> CREATE TABLE t2(a int,b int,c int,d int);
```

```
Query OK, 0 rows affected

gbase> ALTER TABLE t2 ADD GROUPED (a,b,c);
Query OK, 0 rows affected
Records: 0  Duplicates: 0  Warnings: 0

gbase> SHOW CREATE TABLE t2;
+-------+-----------------------------------------------------------+
| Table | Create Table                                              |
+-------+-----------------------------------------------------------+
| t2    | CREATE TABLE "t2" (
  "a" int(11) DEFAULT NULL,
  "b" int(11) DEFAULT NULL,
  "c" int(11) DEFAULT NULL,
  "d" int(11) DEFAULT NULL,
  GROUPED "a" ("a","b","c")
) ENGINE=EXPRESS DEFAULT CHARSET=utf8 TABLESPACE='sys_tablespace' |
+-------+-----------------------------------------------------------+
1 row in set
```

示例 8：建表时创建部分列为行存列，使用 ALTER TABLE 语句增加行存列。

```
gbase> DROP TABLE t2;
Query OK, 0 rows affected

gbase> CREATE TABLE t2(a int,b int,c int,d int, GROUPED a(a,b));
Query OK, 0 rows affected

gbase> ALTER TABLE t2 ADD GROUPED c(c,d);
Query OK, 0 rows affected
Records: 0  Duplicates: 0  Warnings: 0

gbase> SHOW CREATE TABLE t2;
+-------+-----------------------------------------------------+
| Table | Create Table                                        |
+-------+-----------------------------------------------------+
| t2    | CREATE TABLE "t2" (
```

```
  "a" int(11) DEFAULT NULL,
  "b" int(11) DEFAULT NULL,
  "c" int(11) DEFAULT NULL,
  "d" int(11) DEFAULT NULL,
  GROUPED "a" ("a","b"),
  GROUPED "c" ("c","d")
) ENGINE=EXPRESS DEFAULT CHARSET=utf8 TABLESPACE='sys_tablespace' |
+-------+-----------------------------------------------------+
1 row in set
```

示例 9：使用 ALTER TABLE 语句创建行存列，行存列使用压缩语法。

```
gbase> DROP TABLE t2;
Query OK, 0 rows affected

gbase> CREATE TABLE t2 (a int, b varchar(10),c int ,d int);
Query OK, 0 rows affected

gbase> ALTER TABLE t2 ADD GROUPED c(c,d) COMPRESS(3);
Query OK, 0 rows affected
Records: 0  Duplicates: 0  Warnings: 0

gbase> SHOW CREATE TABLE t2;
+-------+-----------------------------------------------------------+
| Table | Create Table                                              |
+-------+-----------------------------------------------------------+
| t2    | CREATE TABLE "t2" (
  "a" int(11) DEFAULT NULL,
  "b" varchar(10) DEFAULT NULL,
  "c" int(11) DEFAULT NULL,
  "d" int(11) DEFAULT NULL,
  GROUPED "c" ("c","d") COMPRESS(3)
) ENGINE=EXPRESS DEFAULT CHARSET=utf8 TABLESPACE='sys_tablespace' |
+-------+-----------------------------------------------------------+
1 row in set
```

示例 10：使用 ALTER TABLE 语句删除行存列。

```
gbase> ALTER TABLE t2 DROP GROUPED c;
Query OK, 0 rows affected
Records: 0  Duplicates: 0  Warnings: 0

gbase> SHOW CREATE TABLE t2;
+-------+-----------------------------------------------------------+
| Table | Create Table                                              |
+-------+-----------------------------------------------------------+
| t2    | CREATE TABLE "t2" (
  "a" int(11) DEFAULT NULL,
  "b" varchar(10) DEFAULT NULL,
  "c" int(11) DEFAULT NULL,
  "d" int(11) DEFAULT NULL
) ENGINE=EXPRESS DEFAULT CHARSET=utf8 TABLESPACE='sys_tablespace' |
+-------+-----------------------------------------------------------+
1 row in set
```

4.2 DML 语句

GBase 8a 支持标准的 DML 语句。

4.2.1 INSERT

语法格式:

INSERT [INTO] [database_name.]table_name [(col_name,...)]

VALUES ({expr | DEFAULT},...),(...),...

或

INSERT [INTO] [database_name.]table_name [(col_name,...)] SELECT ... FROM [database_name.]table_name ...

INSERT 将新行插入到一个已存在的表中。INSERT...VALUES 形式的语句基于明确的值插入记录行。INSERT ... SELECT 形式的语句从另一个或多个表中选取出值，并将其插入。

参数说明如下：

table_name：是要被插入数据的表。

col_name：指出语句指定的值赋给哪个列。

如果在 INSERT...VALUES 或 INSERT...SELECT 中没有指定 column 列表，那么所有列的值必须在 VALUES()列表中或由 SELECT 提供。如果用户不知道表的列的次序，可以使用 DESC *table_name* 来查看。

使用关键词 DEFAULT，明确地把列设置为默认值。这样，编写向所有列赋值的 INSERT 语句时可以更容易，因为使用 DEFAULT 可以避免编写出不完整的、未包含全部列值的 VALUES 清单。如果不使用 DEFAULT，用户必须注明每一个列的名称，与 VALUES 中的每个值对应。

可以使用 DEFAULT(*col_name*)作为设置列缺省值的一个更通用的形式。

如果 column 列表和 VALUES 列表都为空，INSERT 将创建一个行，它的每一列都设置为它的默认值。

可以指定一个表达式 expr 来提供列值。例如，插入一个 INT 型的数据表达式可以写为 INSERT INTO t(a) VALUES(3+5);

示例 1：INSERT INTO...

```
gbase> CREATE TABLE t0(id int);
Query OK, 0 rows affected

gbase> INSERT INTO t0 VALUES(1),(2),(3),(4),(5),(6),(2),(3),(1);
Query OK, 9 rows affected
Records: 9  Duplicates: 0  Warnings: 0

gbase> SELECT * FROM t0;
+------+
| id   |
```

```
+------+
|    1 |
|    2 |
|    3 |
|    4 |
|    5 |
|    6 |
|    2 |
|    3 |
|    1 |
+------+
9 rows in set
```

示例 2: INSERT INTO ... VALUES(DEFAULT)

```
gbase> DROP TABLE IF EXISTS t;
Query OK, 0 rows affected

gbase> CREATE TABLE t( a int NOT NULL DEFAULT 8);
Query OK, 0 rows affected

gbase> INSERT INTO t () VALUES();
Query OK, 1 row affected

gbase> INSERT INTO t(a) VALUES(DEFAULT);
Query OK, 1 row affected

gbase> SELECT a FROM t;
+---+
| a |
+---+
| 8 |
| 8 |
+---+
2 rows in set
```

示例 3: INSERT INTO VALUES(DEFAULT), INSERT INTO VALUES(DEFAULT())

```
gbase> CREATE TABLE t0(id int DEFAULT 1) REPLICATED;
Query OK, 0 rows affected

gbase> INSERT INTO t0 (id) VALUES(DEFAULT);
Query OK, 1 row affected

-- 也可以这样指定默认值进行插入
gbase> INSERT INTO t0 (id) VALUES(DEFAULT(id));
Query OK, 1 row affected

gbase> SELECT * FROM t0;
+------+
| id   |
+------+
|    1 |
|    1 |
+------+
2 rows in set
```

示例 4：INSERT 时，自动更新 TIMESTAMP 列。

示例中用到的表如下：

```
DROP TABLE IF EXISTS t1;
CREATE TABLE t1(a timestamp , b int);
INSERT INTO t1(b) VALUES(1);
INSERT INTO t1(b) VALUES(2);
INSERT INTO t1(b) VALUES(3);
INSERT INTO t1(b) VALUES(4);
INSERT INTO t1(b) VALUES(5);

gbase> SELECT * FROM t1 ORDER BY a;
+---------------------+------+
| a                   | b    |
+---------------------+------+
| 2013-12-26 15:23:01 |    1 |
| 2013-12-26 15:23:01 |    2 |
```

```
| 2013-12-26 15:23:01 |    3 |
| 2013-12-26 15:23:01 |    4 |
| 2013-12-26 15:23:01 |    5 |
+---------------------+------+
5 rows in set
```

-- 使用 INSERT INTO t1 VALUES(),...()语法时，此时 INSERT 完成后数据行对应的 TIMESTAMP 列的值自动更新为一个时间戳。

```
gbase> INSERT INTO t1(b) VALUES (6),(7);
Query OK, 1 row affected
```

-- b=6 和 b=7 对应的 TIMESTAMP 列自动更新为一个时间戳。

```
gbase> SELECT * FROM t1 ORDER BY a;
+---------------------+------+
| a                   | b    |
+---------------------+------+
| 2013-12-26 15:23:01 |    2 |
| 2013-12-26 15:23:01 |    3 |
| 2013-12-26 15:23:01 |    4 |
| 2013-12-26 15:23:01 |    5 |
| 2013-12-26 15:23:01 |    1 |
| 2013-12-26 15:23:33 |    7 |
| 2013-12-26 15:23:33 |    6 |
+---------------------+------+
7 rows in set
```

4.2.2 UPDATE

语法格式:

```
UPDATE [database_name.]table_name
    SET col_name1=expr1 [, col_name2=expr2 ...]
    [WHERE where_definition]
```

当更新列的值是一个合法的表达式时，也可以进行正确的更新赋值操作。

首先创建表 t0 和 t2，并插入数据。

```
gbase> DROP TABLE IF EXISTS t0;
Query OK, 0 rows affected

gbase> CREATE TABLE t0(id int);
Query OK, 0 rows affected

gbase> DROP TABLE IF EXISTS t2;
Query OK, 0 rows affected

gbase> CREATE TABLE t2(id int);
Query OK, 0 rows affected

gbase> INSERT INTO t0(id) VALUES(1),(2),(3),(4),(5),(6),(2),(3),(1);
Query OK, 9 rows affected
Records: 9  Duplicates: 0  Warnings: 0

gbase> INSERT INTO t2(id) VALUES(1),(2),(4);
Query OK, 3 rows affected
Records: 3  Duplicates: 0  Warnings: 0
```

示例 1：更新 t0 表的数据。

```
gbase> SELECT * FROM t0;
+------+
| id   |
+------+
|    1 |
|    2 |
|    3 |
|    4 |
|    5 |
|    6 |
|    2 |
```

```
|    3 |
|    1 |
+------+
9 rows in set
```

```
gbase> UPDATE t0 SET t0.id = t0.id+1 WHERE t0.id > 1;
Query OK, 7 rows affected
Rows matched: 7  Changed: 7  Warnings: 0
```

```
gbase> SELECT * FROM t0;
+------+
| id   |
+------+
|    1 |
|    3 |
|    4 |
|    5 |
|    6 |
|    7 |
|    3 |
|    4 |
|    1 |
+------+
9 rows in set
```

示例 2：使用 IN 的多表查询更新。

```
gbase> SELECT * FROM t0;
+------+
| id   |
+------+
|    1 |
|    2 |
|    3 |
|    4 |
|    5 |
|    6 |
```

```
|    2 |
|    3 |
|    1 |
+------+
9 rows in set

gbase> UPDATE t0 SET t0.id = 10 WHERE t0.id IN (SELECT t2.id FROM t2);
Query OK, 5 rows affected
Rows matched: 5  Changed: 5  Warnings: 0

gbase> SELECT * FROM t0;
+------+
| id   |
+------+
|   10 |
|   10 |
|    3 |
|   10 |
|    5 |
|    6 |
|   10 |
|    3 |
|   10 |
+------+
9 rows in set
```

示例 3：子查询中包含更新列，更新成功。

```
gbase> SELECT * FROM t0;
+------+
| id   |
+------+
|   10 |
|   10 |
|    3 |
|   10 |
|    5 |
```

```
|    6 |
|   10 |
|    3 |
|   10 |
+------+
9 rows in set

gbase> UPDATE t0 SET t0.id = (SELECT id FROM t0 WHERE id = 6);
Query OK, 9 rows affected
Rows matched: 9  Changed: 9  Warnings: 0

gbase> SELECT * FROM t0;
+------+
| id   |
+------+
|    6 |
|    6 |
|    6 |
|    6 |
|    6 |
|    6 |
|    6 |
|    6 |
|    6 |
+------+
9 rows in set
```

4.2.2.1 快速 UPDATE 模式

快速 UPDATE 模式，即先删除符合更新条件的数据，再在表的末尾插入需要更新的新数据。

相对于传统的行存储数据库来说，列存储的数据中 UPDATE 更新少量行时，操作效率相对来说是耗时的，因此，GBase 8a 针对此特点，专门设计了快速 UPDATE 模式，用以提高数据更新操作。

快速 UPDATE 模式目前只支持针对表对象的操作。

要使用快速 UPDATE 模式，必须在客户端使用 SET gbase_fast_update =1;的命令打开快速 UPDATE 模式。更新大批量数据的时候建议使用默认 UPDATE 模式，更新少量数据的时候建议使用快速 UPDATE 模式。

SET gbase_fast_update =0;表示关闭快速 UPDATE 模式。

SET gbase_fast_update =1;表示开启快速 UPDATE 模式。

示例 1：开启快速 UPDATE 模式。

```
gbase> DROP TABLE IF EXISTS t1;
Query OK, 0 rows affected

gbase> CREATE TABLE t1 (f_1 int);
Query OK, 0 rows affected

gbase> INSERT INTO t1 values(1),(2),(3);
Query OK, 3 rows affected
Records: 3  Duplicates: 0  Warnings: 0

gbase> SELECT * FROM t1;
+------+
| f_1  |
+------+
|    1 |
|    2 |
|    3 |
+------+
3 rows in set

gbase> SET gbase_fast_update = 1;
Query OK, 0 rows affected

gbase> UPDATE t1 SET f_1 = 10 WHERE f_1= 1;
Query OK, 1 row affected
Rows matched: 1  Changed: 1  Warnings: 0
```

```
gbase> SELECT * FROM t1;
+------+
| f_1  |
+------+
|    2 |
|    3 |
|   10 |
+------+
3 rows in set
```

4.2.3 DELETE

语法格式:

DELETE FROM [*database_name*.]*table_name* [tbl_alias] [WHERE where_definition]

其中，关键字[FROM]和表别名[tbl_alias]是可选关键字。

当 DELETE 语句中包含别名时，可以省略 FROM 关键字。

示例 1: 删除表中的数据。

--删除 id 大于 6 的数据。

```
gbase> DELETE FROM t0 WHERE t0.id > 6;
Query OK, 2 rows affected
```

--使用 IN，删除 id 值为 1，2，3 的数据。

```
gbase> DELETE FROM t0 WHERE t0.id IN ( 1,2,3);
Query OK, 3 rows affected
```

--删除全表数据。

```
gbase> DELETE FROM t0;
Query OK, 3 rows affected
```

示例 2：DELETE FROM...WHERE...IN (SELECT...FROM)

```
gbase> INSERT INTO t0 values(1),(2),(3),(4),(5),(6),(7),(8);
Query OK, 8 rows affected
Records: 8  Duplicates: 0  Warnings: 0

gbase> DELETE FROM t0 WHERE t0.ID IN (SELECT id FROM t0);
Query OK, 8 rows affected
```

示例 3：DELETE 语法中包含表的别名，可以省略 FROM 关键字。

```
gbase> INSERT INTO t0 values(1),(2),(3),(4),(5),(6),(7),(8);
Query OK, 8 rows affected
Records: 8  Duplicates: 0  Warnings: 0

gbase> DELETE FROM t0 tt WHERE tt.id=8;
Query OK, 1 row affected

gbase> DELETE t0 tt WHERE tt.id=1;
Query OK, 1 row affected
```

查看删除后的数据：

```
gbase> SELECT * FROM t0;
+------+
| id   |
+------+
|    2 |
|    3 |
|    4 |
|    5 |
|    6 |
|    7 |
+------+
6 rows in set
```

示例 4：DELETE ...WHERE...

```
gbase> DELETE t0 WHERE id = 2;
Query OK, 1 row affected
```

4.2.4 SELECT

语法格式:

```
SELECT
      [ALL | DISTINCT | DISTINCTROW ]
    select_expr, ...
    [FROM table_references
    [WHERE where_definition]
      [GROUP BY {col_name | expr | position}
    , ...]
      [HAVING where_definition]
        [ORDER BY {col_name | expr | position}
        [ASC | DESC] , ...]
      [LIMIT {[offset,] row_count | row_count OFFSET offset}]
[PROCEDURE procedure_name(argument_list)]]
      [INTO OUTFILE 'file_name' export_options]
```

在 SELECT 关键字之后可以给出大量的选项，它们会影响到语句的操作。

ALL，DISTINCT 和 DISTINCTROW 选项指定了是否返回重复的行，缺省为 ALL（所有匹配的行都返回）。DISTINCT 和 DISTINCTROW 是同义的，用于删除结果集中重复的行。

select_expr：指查询显示的列，可以使用 AS 来为 SELECT 显示的列命名别名，别名不要和 SELECT 显示的列名重复。

table_references：指定从其中找出行的一个或多个表。它的语法在 JOIN 语法中有描述。

where_definition：含有 WHERE 关键字，其后是查询所要满足的一个或多个条件的表达式。

[GROUP BY {*col_name* | expr | position}, ...] [HAVING where_definition] 参见“4.2.9.3 GROUP BY ...”。

[ORDER BY {*col_name* | expr | position} [ASC | DESC] , ...] 参见“4.2.9.4 ORDER BY...”。

[LIMIT {[offset,] row_count | row_count OFFSET *offset*}] 参见“4.2.9.5 LIMIT ...”。

[INTO OUTFILE '*file_name*' export_options]参见“4.3 分级查询语句

GBase 8a 的 SELECT 语句支持使用 START WITH...CONNECT BY 语句实现分级查询。

4.2.5 相关概念

在使用 START WITH...CONNECT BY 语句前，请先了解以下概念。

- 数据源：单物理表、单逻辑表或单个视图。
- 迭代种子：START WITH 指定的过滤条件，基于数据源过滤出来的数据称为迭代种子，迭代种子都是根节点，如果省略 START WITH，那么所有数据源都是根节点。
- 迭代关系：CONNECT BY 指定的关联条件，父子节点之间通过迭代关系进行关联。
- 分级查询：基于迭代种子和数据源，根据迭代关系遍历数据，称为分级查询，也称树形查询或层次查询。即，基于 START WITH 指定的根节点和原始表数据，根据 CONNECT BY 指定的关系遍历数据

- 环路：如果一个节点的子孙节点同时又是该节点的祖先节点，称为存在环路。
- 兄弟间排序：按 KEY 值顺序遍历数据，遍历规则是树的深度优先遍历中的先序遍历。
- 伪列：伪列并不存储在表中，但是它的行为却类似于表中的字段，可以基于伪列做查询操作，但不能更新伪列；伪列也可以理解为没有参数的函数，区别是：基于不同的行，伪列的值一般是不同的。
- 集：某些特定数据集合，如父节点集是指当前层的上层节点数据集合，子节点集是指当前层的数据集合，祖先节点集是指当前层的所有祖先节点组成的数据集合。

4.2.6 执行原理

遍历表中的每条记录，对比是否满足 START WITH 后的条件，如果不满足则继续下一条，如果满足则以该记录为根节点，然后递归寻找该节点下的子节点，查找条件是 CONNECT BY 后面指定的条件。

4.2.7 语法格式

```
SELECT column_list|[LEVEL]
FROM single_table | view
[WHERE ...]
[hierarchical_clause]
[GROUP BY ...]
[ORDER [SIBLINGS] BY ...]


hierarchical_clause:
```

```
[START WITH <conditions>] CONNECT BY <connect_conditions>
| CONNECT BY <connect_conditions> [START WITH <conditions>]
[ORDER SIBLINGS BY {col_name | expr | position} [ASC | DESC], ...]

connect_condition:
PRIOR expr1 op expr2
| expr1 op PRIOR expr2
| expr op connect_condition
| expr
```

LEVEL：可选关键字，表示等级。即该节点在树中的层数，根节点为第一层，level 为 1

WHERE：根据 CONNECT BY <connect_conditions> START WITH<connect_conditions>选择出来的记录进行过滤，是针对单条记录的过滤。WHERE 条件限制查询返回的行。

hierarchical_clause 中参数说明如下：

START WITH <conditions>：START WITH 表示开始的记录，限定作为搜索起始点的条件，如果是自上而下的搜索则是限定作为根节点的条件，如果是自下而上的搜索则是限定作为叶子节点的条件。

CONNECT BY <conditions>：这里执行的过滤会把符合条件的记录及其下所有子节点都过滤掉。

CONNECT BY PRIOR 指定与当前记录关联时的字段关系。

START WITH 和 CONNECT BY 的先后顺序不影响查询结果。

ORDER SIBLINGS BY：使用此关键字时可以进行层次查询

connect_condition 中参数说明如下：

PRIOR：表示上一条记录。可以指定与当前记录关联时的字段关系。包含此参数时表示不进行递归查询。

4.2.8 使用约束

START WITH...CONNECT BY 语句在使用时有以下约束：

1、分级查询只能基于单物理表或单隐式表（derived table）。

以下语句写法错误：

SELECT LEVEL FROM t1, t1 t2 WHERE t1.i = t2.i CONNECT BY t1.i =PRIOR t1.j GROUP BY LEVEL ORDER BY LEVEL;

2、CONNECT BY 关联条件不能包含 OR 操作，并且必须包含父子节点间的等值条件，等号的两边必须是不同的维度（一边包含 PRIOR，一边不包含 PRIOR）。

以下语句写法正确：

SELECT * FROM t1 START WITH i = 1 CONNECT BY LEVEL < 4 and PRIOR j = i ORDER SIBLINGS BY j;

以下语句写法错误：

SELECT * FROM t1 START WITH i = 1 CONNECT BY LEVEL < 4 and PRIOR j > i ORDER SIBLINGS BY j;

SELECT * FROM t1 START WITH i = 1 CONNECT BY PRIOR j = i or PRIOR i = j ORDER SIBLINGS BY j;

3、PRIOR 是一元操作符，优先级同正负号，只能用在 CONNECT BY 子句中；PRIOR 后面不可以接伪列（LEVEL、ROWID 等）、不可以接包含伪列的表达式、不可以嵌套使用 PRIOR，不可以接聚合函数。

以下语句写法错误：

```
SELECT 1 FROM t1 START WITH i = 1 CONNECT BY j = PRIOR (ROWID+1);
```

```
SELECT 1 FROM t1 START WITH i = 1 CONNECT BY j = PRIOR LEVEL;
```

```
SELECT 1 FROM t1 START WITH i = 1 CONNECT BY j = PRIOR SUM(i);
```

```
SELECT 1 FROM t1 START WITH i = 1 CONNECT BY j = PRIOR (i + PRIOR 1);
```

4、ORDER SIBLINGS BY 只能用在分级查询语句中，并且不能同聚合同时存在

以下语句写法错误：

```
SELECT * FROM t1 START WITH i = 1 CONNECT BY PRIOR j = i GROUP BY i,j ORDER SIBLINGS BY j;
```

5、LEVEL 伪列只能用于分级查询语句中。

以下语句写法错误：

```
SELECT LEVEL FROM t1;
```

6、不支持实时环路判断，只能保证最终能够检测出环路，如果数据量太大（如超过十万），会耗时很长才能检测出。

7、最大节点数为 MAX_INT(2147483647)

8、不能基于做过 DLETE 的表进行分级查询，但可以修改为隐式表的形式。

```
DELETE FROM t1 WHERE…;
```

以下语句写法错误：

```
SELECT … FROM t1 CONNECT BY…;
```

以下语句写法正确：

```
SELECT … FROM (SELECT * FROM t1) t1 CONNECT BY…;
```

9、在配置文件“gbase_8a_gbase8a.cnf”中增加参数

gbase_max_allowed_level,用于控制允许出现的最大 level 数,默认值为 1024,最大 2147483647，最小 1。

本版本不支持以下功能:

- 不支持 NOCYCLE 参数，如果使用该参数，有环路也不会报错
- 不支持 CONNECT_BY_ISCYCLE 伪列，是否为产生环路的节点
- 不支持 CONNECT_BY_ISLEAF 伪列，该节点是否为叶子节点
- 不支持 SYS_CONNECT_BY_PATH 函数
- 不支持 CONNECT_BY_ROOT 操作符

4.2.9 分级查询语句示例

4.2.9.1 表

建表语句如下:

```
DROP TABLE t1;
CREATE TABLE t1(i int, j int, l varchar(10), k double);
CREATE TABLE t2 LIKE t1;

INSERT INTO t1 VALUES
(1,2,'2013-1-2',1.2),(1,5,'2013-1-5',1.5),(1,8,'2013-1-8',1.8);
INSERT INTO t1 VALUES (5,3,'2013-5-3',5.3),(5,4,'2013-5-4',5.4);
INSERT INTO t1 VALUES (2,7,'2013-2-7',2.7),(2,6,'2013-2-6',2.6);
INSERT INTO t1 VALUES (8,9,'2013-8-9',8.9),(8,0,'2013-8-1',8.0);
INSERT INTO t1 VALUES (0,10,'2011-1-10',0.1);
INSERT INTO t2 SELECT * FROM t1;
```

示例 1: START WITH...CONNECT BY PRIOR...

```
gbase> SELECT * FROM t1 START WITH i = 1 CONNECT BY PRIOR j = i ORDER BY j,
l DESC, k;
+------+------+-----------+------+
| i    | j    | l         | k    |
+------+------+-----------+------+
|    8 |    0 | 2013-8-1  |    8 |
|    1 |    2 | 2013-1-2  |  1.2 |
|    5 |    3 | 2013-5-3  |  5.3 |
|    5 |    4 | 2013-5-4  |  5.4 |
|    1 |    5 | 2013-1-5  |  1.5 |
|    2 |    6 | 2013-2-6  |  2.6 |
|    2 |    7 | 2013-2-7  |  2.7 |
|    1 |    8 | 2013-1-8  |  1.8 |
|    8 |    9 | 2013-8-9  |  8.9 |
|    0 |   10 | 2011-1-10 |  0.1 |
+------+------+-----------+------+
10 rows in set
```

示例 2: START WITH...CONNECT BY PRIOR...ORDER SIBLINGS BY...

```
gbase> SELECT * FROM t1 START WITH i = 1 CONNECT BY PRIOR j = i AND (1 OR k
= i) > 0 ORDER SIBLINGS BY j, l DESC, k;
+------+------+-----------+------+
| i    | j    | l         | k    |
+------+------+-----------+------+
|    8 |    0 | 2013-8-1  |    8 |
|    1 |    2 | 2013-1-2  |  1.2 |
|    5 |    3 | 2013-5-3  |  5.3 |
|    5 |    4 | 2013-5-4  |  5.4 |
|    1 |    5 | 2013-1-5  |  1.5 |
|    2 |    6 | 2013-2-6  |  2.6 |
|    2 |    7 | 2013-2-7  |  2.7 |
|    1 |    8 | 2013-1-8  |  1.8 |
|    8 |    9 | 2013-8-9  |  8.9 |
|    0 |   10 | 2011-1-10 |  0.1 |
+------+------+-----------+------+
10 rows in set
```

示例 3：CONNECT BY PRIOR 与 CONNECT BY。

```
gbase> SELECT * FROM t1 CONNECT BY PRIOR j = i;
+------+------+-----------+------+
| i    | j    | l         | k    |
+------+------+-----------+------+
|    1 |    2 | 2013-1-2  |  1.2 |
|    1 |    5 | 2013-1-5  |  1.5 |
|    1 |    8 | 2013-1-8  |  1.8 |
|    5 |    3 | 2013-5-3  |  5.3 |
|    5 |    4 | 2013-5-4  |  5.4 |
|    2 |    7 | 2013-2-7  |  2.7 |
|    2 |    6 | 2013-2-6  |  2.6 |
|    8 |    9 | 2013-8-9  |  8.9 |
|    8 |    0 | 2013-8-1  |    8 |
|    0 |   10 | 2011-1-10 |  0.1 |
+------+------+-----------+------+
10 rows in set

gbase> SELECT * FROM t1 CONNECT BY j = i;
+------+------+-----------+------+
| i    | j    | l         | k    |
+------+------+-----------+------+
|    1 |    2 | 2013-1-2  |  1.2 |
|    1 |    5 | 2013-1-5  |  1.5 |
|    1 |    8 | 2013-1-8  |  1.8 |
|    5 |    3 | 2013-5-3  |  5.3 |
|    5 |    4 | 2013-5-4  |  5.4 |
|    2 |    7 | 2013-2-7  |  2.7 |
|    2 |    6 | 2013-2-6  |  2.6 |
|    8 |    9 | 2013-8-9  |  8.9 |
|    8 |    0 | 2013-8-1  |    8 |
|    0 |   10 | 2011-1-10 |  0.1 |
+------+------+-----------+------+
10 rows in set
```

示例 4: SELECT * FROM table WHERE ...START WITH...CONNECT BY PRIOR

```
gbase> SELECT * FROM t1 WHERE i < 1 START WITH i = 1 CONNECT BY PRIOR j = i
ORDER BY j, l DESC, k;
+------+------+-----------+------+
| i    | j    | l         | k    |
+------+------+-----------+------+
|    0 |   10 | 2011-1-10 |  0.1 |
+------+------+-----------+------+
1 row in set
```

4.2.9.2 视图

视图支持分级查询，即 START WITH...CONNECT BY PRIOR 的使用。

示例 1: START WITH...CONNECT BY PRIOR...

```
gbase> DROP VIEW IF EXISTS v1;
Query OK, 0 rows affected

gbase> CREATE VIEW v1 AS SELECT * FROM t1 START WITH i = 1 CONNECT BY PRIOR
j = i ORDER BY j, l DESC, k;
Query OK, 0 rows affected

gbase> SELECT * FROM v1;
+------+------+-----------+------+
| i    | j    | l         | k    |
+------+------+-----------+------+
|    8 |    0 | 2013-8-1  |    8 |
|    1 |    2 | 2013-1-2  |  1.2 |
|    5 |    3 | 2013-5-3  |  5.3 |
|    5 |    4 | 2013-5-4  |  5.4 |
|    1 |    5 | 2013-1-5  |  1.5 |
|    2 |    6 | 2013-2-6  |  2.6 |
|    2 |    7 | 2013-2-7  |  2.7 |
|    1 |    8 | 2013-1-8  |  1.8 |
```

```
|    8 |    9 | 2013-8-9  |  8.9 |
|    0 |   10 | 2011-1-10 |  0.1 |
+------+------+-----------+------+
10 rows in set

gbase> SHOW CREATE VIEW v1;
+------+-------------------+----------------------+----------------------+
| View | Create View       | character_set_client | collation_connection |
+------+-------------------+----------------------+----------------------+
| v1   | CREATE ALGORITHM=UNDEFINED DEFINER="root"@"%" SQL SECURITY DEFINER
VIEW "v1" AS select "t1"."i" AS "i","t1"."j" AS "j","t1"."l" AS "l","t1"."k"
AS "k" from "t1" start with ("t1"."i" = 1) connect by (prior "t1"."j" = "t1"."i")
order by "t1"."j","t1"."l" desc,"t1"."k" | utf8                 |
utf8_general_ci     |
+------+-------------------+----------------------+----------------------+
1 row in set
```

示例 2: START WITH...CONNECT BY PRIOR...ORDER SIBLINGS BY...

```
gbase> DROP VIEW v1;
Query OK, 0 rows affected

gbase> CREATE VIEW v1 AS SELECT * FROM t1 START WITH i = 1 CONNECT BY PRIOR
j = i AND (1 OR k = i) > 0 ORDER SIBLINGS BY j, l DESC, k;
Query OK, 0 rows affected

gbase> SELECT * FROM v1;
+------+------+-----------+------+
| i    | j    | l         | k    |
+------+------+-----------+------+
|    1 |    2 | 2013-1-2  |  1.2 |
|    2 |    6 | 2013-2-6  |  2.6 |
|    2 |    7 | 2013-2-7  |  2.7 |
|    1 |    5 | 2013-1-5  |  1.5 |
|    5 |    3 | 2013-5-3  |  5.3 |
|    5 |    4 | 2013-5-4  |  5.4 |
|    1 |    8 | 2013-1-8  |  1.8 |
```

```
|    8 |    0 | 2013-8-1  |    8 |
|    0 |   10 | 2011-1-10 |  0.1 |
|    8 |    9 | 2013-8-9  |  8.9 |
+------+------+-----------+------+
10 rows in set

gbase> SHOW CREATE VIEW v1;
+------+--------------------+----------------------+----------------------+
| View | Create View        | character_set_client | collation_connection |
+------+--------------------+----------------------+----------------------+
| v1   | CREATE ALGORITHM=UNDEFINED DEFINER="root"@"%" SQL SECURITY DEFINER
VIEW "v1" AS select "t1"."i" AS "i","t1"."j" AS "j","t1"."l" AS "l","t1"."k"
AS "k" from "t1" start with ("t1"."i" = 1) connect by ((prior "t1"."j" = "t1"."i")
and ((1 or ("t1"."k" = "t1"."i")) > 0)) order siblings by "t1"."j","t1"."l"
desc,"t1"."k" | utf8                 | utf8_general_ci      |
+------+--------------------+----------------------+----------------------+
1 row in set
```

示例 3：WHERE...START WITH...CONNECT BY PRIOR...

```
gbase> DROP VIEW v1;
Query OK, 0 rows affected

gbase> CREATE VIEW v1 AS SELECT * FROM t1 WHERE i < 1 START WITH i = 1 CONNECT
BY PRIOR j = i ORDER BY j, l DESC, k;
Query OK, 0 rows affected

gbase> SELECT * FROM v1;
+------+------+-----------+------+
| i    | j    | l         | k    |
+------+------+-----------+------+
|    0 |   10 | 2011-1-10 |  0.1 |
+------+------+-----------+------+
1 row in set

gbase> SHOW CREATE VIEW v1;
+------+--------------+----------------------+----------------------+
```

```
| View | Create View    | character_set_client | collation_connection |
+------+----------------+----------------------+----------------------+
| v1   | CREATE ALGORITHM=UNDEFINED DEFINER="root"@"%" SQL SECURITY DEFINER
VIEW "v1" AS select "t1"."i" AS "i","t1"."j" AS "j","t1"."l" AS "l","t1"."k"
AS "k" from "t1" where ("t1"."i" < 1) start with ("t1"."i" = 1) connect by (prior
"t1"."j" = "t1"."i") order by "t1"."j","t1"."l" desc,"t1"."k" | utf8
| utf8_general_ci      |
+------+----------------+----------------------+----------------------+
1 row in set
```

查询结果导出语句”。

4.2.9.3 GROUP BY ...

GROUP BY {*col_name* | expr | position}, ... [HAVING where_definition]

参数说明如下:

col_name: 指定分组的数据列，多列之间用“,”分隔。*col_name* 可以是 SELECT 中使用 AS 定义的别名。

expr: 指定分组的表达式，多列之间用“,”分隔。

注意:上面的 *col_name* 和 expr 中定义的数据列或表达式,可以不是 SELECT col_name_1,…, col_name_n FROM 之间的数据列，这一点是 GBase 8a 中比较特殊的语法。

position: 在 SELECT col_name_1,..., col_name_n FROM 之间的 col_name_1,..., col_name_n 的序号，position 是整数型数值，从 1 开始。

例如: SELECT stu_no,stu_name FROM stundent GROUP BY 1 ;1 就是指代数据列 stu_no。

HAVING where_definition: 使用 GROUP BY 子句对数据进行分组，对 GROUP BY 子句形成的分组，使用聚集函数计算各个分组的值，最后用 HAVING 子句过滤掉不符合条件的分组。

GROUP BY 的功能:

通过一定的规则将一个数据集划分成若干个小的区域（即分组），然后针对若干个小区域进行数据处理，通常 GROUP BY 和聚合函数，OLAP 函数配合使用。

注意：

HAVING 子句中的每一个元素不必出现在 SELECT 列表中。

HAVING 子句限制的是分组，而不是行，因此可以使用聚合函数。

示例 1：按照性别分组，计算不同性别的人数。

示例中用到的表及数据如下：

```
CREATE TABLE student  (stu_no int, stu_name varchar(200),stu_sex int);
INSERT INTO student (stu_no,stu_name,stu_sex)
VALUES(1,'Tom',0),(2,'Jim',0),(3,'Rose',1);
INSERT INTO student (stu_no,stu_name,stu_sex)
VALUES(4,'John',1),(5,'Mike',0),(6,'Jane',1);
INSERT INTO student (stu_no,stu_name,stu_sex)
VALUES(7,'Jack',0),(8,'Max',0),(9,'Allen',1);
INSERT INTO student (stu_no,stu_name,stu_sex) VALUES(10,'Jerry',0);

CREATE TABLE result (stu_no int,math double(4,1),english double(4,1));
INSERT INTO result VALUES(1,80.0,85.2);
INSERT INTO result VALUES(2,78.0,95.5);
INSERT INTO result VALUES(4,89.5,99.0);
INSERT INTO result VALUES(3,65.0,78.5);
INSERT INTO result VALUES(5,92.0,94.0);
INSERT INTO result VALUES(6,72.5,86.0);
INSERT INTO result VALUES(7,85.0,76.0);
INSERT INTO result VALUES(10,95.0,97.0);
INSERT INTO result VALUES(9,56.0,78.0);
INSERT INTO result VALUES(8,86.0,93.0);
```

```
gbase> SELECT CASE stu_sex WHEN 0 THEN '男' ELSE '女' END  AS
stu_sexname,COUNT(stu_sex) AS stu_count FROM student GROUP BY stu_sex;
+-------------+-----------+
| stu_sexname | stu_count |
+-------------+-----------+
```

```
| 男          |         6 |
| 女          |         4 |
+-------------+-----------+

2 rows in set
```

示例 2：查看男生和女生的数学，外语成绩总和。

```
gbase> SELECT CASE a.stu_sex WHEN 0 THEN '男' ELSE '女' END as
stu_sex ,sum(b.math+b.english) AS total FROM student a INNER JOIN result b ON
a.stu_no = b.stu_no GROUP BY a.stu_sex ORDER BY total;
+---------+--------+
| stu_sex | total  |
+---------+--------+
| 女      |  624.5 |
| 男      | 1056.7 |
+---------+--------+
2 rows in set
```

4.2.9.4 ORDER BY...

ORDER BY {*col_name* | expr | position} [ASC | DESC] , ...

参数说明如下：

ORDER BY 用于对结果集进行排序，数据列列名称或者表达式。

col_name：指定排序的数据列，多列之间用“,”分隔。*col_name* 可以是 SELECT 中使用 AS 定义的别名。

expr：指定排序的表达式，多列之间用“,”分隔。

position：在 SELECT col_name_1,……, col_name_n FROM 之间的 col_name_1,……, col_name_n 的序号，position 是整数型数值，从 1 开始。

例如：SELECT stu_no,stu_name FROM stundent ORDER BY 1 ；1 就是指代数据列 stu_no。

ASC | DESC：如果希望对记录进行排序，可以使用 ASC 或 DESC 关键字来指定排序规则，ASC 代表升序规则，DESC 代表降序规则。默认按照升序对记录进行排序。

示例 1：... ORDER BY...

```
gbase> SELECT a.stu_name,math,english,sum(math+english) AS total FROM student
a INNER JOIN result b ON a.stu_no = b.stu_no GROUP BY a.stu_no ORDER BY a.stu_no;
+----------+------+---------+-------+
| stu_name | math | english | total |
+----------+------+---------+-------+
| Tom      | 80.0 |    85.2 | 165.2 |
| Jim      | 78.0 |    95.5 | 173.5 |
| John     | 89.5 |    99.0 | 188.5 |
| Rose     | 65.0 |    75.5 | 140.5 |
| Jane     | 92.0 |    94.0 | 186.5 |
| Mike     | 72.5 |    86.0 | 158.5 |
| Jack     | 85.0 |    76.0 | 161.5 |
| Jerry    | 95.0 |    97.0 | 192.0 |
| Allen    | 56.0 |    78.0 | 134.0 |
| Max      | 86.0 |    93.0 | 179.0 |
+----------+------+---------+-------+
10 rows in set
```

4.2.9.5 LIMIT ...

LIMIT {[offset,] row_count | row_count OFFSET offset}

参数说明如下：

LIMIT row_count：row_count 是一个整数型数值，表示从记录集开始返回 row_count 行结果集。如果 row_count 指定的数值大于 SELECT 后的结果集，那么 row_count 将不起作用。

LIMIT row_count

等价于

LIMIT 0, row_count

或者等价于

LIMIT row_count OFFSET 0

LIMIT row_count OFFSET offset: row_count 指定返回结果集的行数，offset 指定结果集的偏移量，初偏移量的起始值是 0（而不是 1），即偏移量 0 对应 SELECT 返回的第一行结果集。

下面的语句含义为从 SELECT 结果集的偏移量 5 的位置开始，返回 10 行结果集。

```
SELECT * FROM ssbm.customer LIMIT 5,10;
```

示例 1: ... LIMIT...

```
gbase> SELECT SUM(lo_quantity),lo_orderkey FROM lineorder GROUP BY lo_orderkey
ORDER BY lo_orderkey  LIMIT 10;
+------------------+-------------+
| SUM(lo_quantity) | lo_orderkey |
+------------------+-------------+
|               61 |           1 |
|              149 |           2 |
|              151 |           3 |
|               30 |           4 |
|               41 |           5 |
|              191 |           6 |
|               12 |           7 |
|               66 |          32 |
|              184 |          33 |
|               75 |          34 |
+------------------+-------------+
10 rows in set
```

示例 2: t 表中包含 10 行数据，使用 LIMIT m OFFSET n 的形式，显示执行

SELECT 语句后的结果。

查看全部 10 行结果集：

```
gbase> SELECT * FROM t1 LIMIT 10;
+------+
| a    |
+------+
|    1 |
|    2 |
|    3 |
|    4 |
|    5 |
|    6 |
|    7 |
|    8 |
|    9 |
|   10 |
+------+
10 rows in set
```

从结果集中偏移量为 2 的位置开始，返回 3 行结果集，因为 SELECT 结果集的第一行的偏移值为 0，所以 SELECT 的第三行是偏移量 2 的起始位置，从此处取 3 行结果集。

```
gbase> SELECT * FROM t1 LIMIT 3 OFFSET 2;
+------+
| a    |
+------+
|    3 |
|    4 |
|    5 |
+------+
3 rows in set
```

4.2.10 JOIN

对于 SELECT 语句中的 TABLE_references 部分,GBase SQL 支持下面的 JOIN 语法:

table_reference, table_reference

table_reference [INNER | CROSS | FULL] JOIN table_reference [join_condition]

table_reference [LEFT | RIGHT [OUTER]] JOIN table_reference [join_condition]

table_reference 定义为:

[database_name.]table_name [[AS] alias]

join_condition 定义为:

[database_name.]table_name [[AS] alias]

ON 部分是用来在结果集中限定哪些行是需要的，一般不要在这部分附加任何条件，而是在 WHERE 子句中指定这些条件。这项规定也有例外。

表的引用可以通过[database_name.]table_name AS alias_name 或 [database_name.]table_name alias_name 来赋予一个别名:

```
gbase> SELECT t1.c_name,t2.lo_orderkey FROM ssbm.customer AS
t1,ssbm.lineorder AS t2 WHERE t1.c_custkey = t2.lo_custkey LIMIT 5;
+--------------------+-------------+
| c_name             | lo_orderkey |
+--------------------+-------------+
| Customer#000025738 |           1 |
| Customer#000025738 |           1 |
| Customer#000025738 |           1 |
| Customer#000018238 |           2 |
| Customer#000018238 |           2 |
+--------------------+-------------+
5 rows in set

gbase> SELECT t1.c_name,t2.lo_orderkey FROM ssbm.customer  t1,ssbm.lineorder
t2 WHERE t1.c_custkey = t2.lo_custkey LIMIT 5;
```

```
+--------------------+-------------+
| c_name             | lo_orderkey |
+--------------------+-------------+
| Customer#000025738 |           1 |
| Customer#000025738 |           1 |
| Customer#000025738 |           1 |
| Customer#000018238 |           2 |
| Customer#000018238 |           2 |
+--------------------+-------------+
5 rows in set
```

ON 条件是可以在 WHERE 子句中使用的条件表达式形式。

如果在 LEFT JOIN 的 ON 或 USING 部分的右表中没有匹配的记录，那么该右表一行中的所有列都设置为 NULL。可以用这种方法找出一个表和另一个表之间不匹配的记录:

```
gbase> SELECT t1.stu_name,t2.c_no FROM student t1 LEFT OUTER JOIN course t2
ON t1.stu_name=t2.stu_name WHERE c_no IS NULL;
+----------+---------+
| stu_name | c_no    |
+----------+---------+
| Smith    | NULL    |
| KING     | NULL    |
| Koamma   | NULL    |
+----------+---------+
2 rows in set
```

这个示例执行结果为，在 customer 中找出 stu_anme 值不在 course 的所有行（也就是，student 与 course 没有对应关系的所有行）。假设 course.c_no 都声明为 NOT NULL。

USING(column_list)子句用于为一系列的列进行命名。这些列必须同时在两个表中存在。如果表 a 和表 b 都包含列 c1，c2 和 c3，则以下联合会包含来自两个表的对应的列:

a LEFT JOIN b USING (c1,c2,c3)

a LEFT JOIN b ON a.c1 = b.c1 AND a.c2 = b.c2 AND a.c3 = b.c3

在缺乏连接条件时，INNER JOIN 和，（逗号）在语义上是等价的：都是在指定的两个表之间生成一个笛卡尔乘积（也就是，第一个表中的每一行都与第二个表中的所有行连接起来）。

GBase 8a 支持 FULL JOIN/FULL OUTER JOIN(全连接)，可以看成是 LEFT JOIN（左连接）和 RIGHT JOIN（右连接）结果的合集。但是 FULL JOIN 跟 LEFT JOIN...UNION RIGHT JOIN...并不完全等价，因为 UNION 会去除重复记录。

示例 1：FULL JOIN 与 LEFT JOIN...UNION RIGHT JOIN...

示例中用到的表及数据：

```
USE test;
CREATE TABLE t1(a int,b int);
CREATE TABLE t2(a int,b int);
INSERT INTO t1 values(11,11),(22,22),(33,33);
INSERT INTO t1 values(11,11),(22,22),(33,33);
INSERT INTO t2 values(11,11),(222,222),(333,333);
```

```
gbase> SELECT * FROM t1;
+------+------+
| a    | b    |
+------+------+
|   11 |   11 |
|   22 |   22 |
|   33 |   33 |
|   11 |   11 |
|   22 |   22 |
|   33 |   33 |
+------+------+
6 rows in set

gbase> SELECT * FROM t2;
+------+------+
```

```
| a    | b    |
+------+------+
|   11 |   11 |
|  222 |  222 |
|  333 |  333 |
+------+------+
3 rows in set
```

```
gbase> SELECT * FROM t1 FULL JOIN t2 on t1.a = t2.a;
+------+------+------+------+
| a    | b    | a    | b    |
+------+------+------+------+
|   11 |   11 |   11 |   11 |
|   11 |   11 |   11 |   11 |
| NULL | NULL |  333 |  333 |
| NULL | NULL |  222 |  222 |
|   22 |   22 | NULL | NULL |
|   33 |   33 | NULL | NULL |
|   22 |   22 | NULL | NULL |
|   33 |   33 | NULL | NULL |
+------+------+------+------+
8 rows in set
```

```
gbase> (SELECT * FROM t1 LEFT JOIN t2 ON t1.a = t2.a) UNION (SELECT * FROM t1
RIGHT JOIN t2 ON t1.a = t2.a);
+------+------+------+------+
| a    | b    | a    | b    |
+------+------+------+------+
|   11 |   11 |   11 |   11 |
|   22 |   22 | NULL | NULL |
|   33 |   33 | NULL | NULL |
| NULL | NULL |  333 |  333 |
| NULL | NULL |  222 |  222 |
+------+------+------+------+
5 rows in set
```

4.2.11UNION

SELECT ...UNION [ALL | DISTINCT]SELECT ...

[UNION [ALL | DISTINCT]

SELECT ...]

UNION 用来将多个 SELECT 语句的结果组合到一个结果集中。

在每个 SELECT 语句对应位置上的选择列，应该有相同的类型。(例如，第一个语句选出的第一列应该与其它语句选出的第一列的类型相同)。第一个 SELECT 语句中用到的列名称被作为返回结果的列名称。

SELECT 语句是一般的查询语句，但是有以下约束:

1、不同的数据类型，不能使用 UNIO 和 UNION ALL，例如: 数值型，字符型，日期和时间型之间不能使用 UNION 和 UNION ALL，但是 DATETIME 和 TIMESTAMP 类型可以使用 UNION 和 UNION ALL，其它的日期和时间类型则不行。

2、如果 UNION 和 UNION ALL 两边的数据类型为 CHAR 类型，进行 UNION 和 UNION ALL 操作时，结果返回 VARCHAR 类型。

例如:SELECT CHAR(10) UNION SELECT CHAR(255)的结果集为 VARCHAR(255)。

3、UNION 和 UNION ALL 结果由小的数据类型向大的数据类型转换，如: INT -> BIGINT -> DECIMAL -> DOUBLE。

例如: SELECT INT UNION SELECT DOUBLE 的结果集为 DOUBLE 型。

4、NULL 可以和任何类型做 UNION 和 UNION ALL。

例如: SELECT NULL UNION SELECT 1;

5、如果只是使用 UNION，那么将返回无重复记录的结果集，此时，UNION 等同于 UNION DISTINCT。如果使用 UNION ALL，将会返回所有 SELECT 后的结果集，这个结果集会存在重复的记录。

6、如果在多个 SELECT 的 UNION 查询中,同时存在 UNION [DISTINCT]和 UNION

ALL，那么 UNION ALL 会被忽略，最终返回 UNION [DISTINCT]后的结果集（过滤掉重复的记录行）。

7、如果希望使用 ORDER BY 或 LIMIT 子句来分类或限制整个 UNION 结果，可以给单独的 SELECT 语句加上括号或者把 ORDER BY 或 LIMIT 置于最后。

示例 1：SELECT 语句中使用 ORDER BY 子句分类 UNION 结果。

```
gbase> CREATE TABLE student (stu_no int, stu_name varchar(200),stu_sex int);
Query OK, 0 rows affected

gbase> INSERT INTO student VALUES(4,'King',1),(5,'Smith',1);
Query OK, 2 rows affected
Records: 2  Duplicates: 0  Warnings: 0

gbase> SELECT stu_name FROM student WHERE stu_name LIKE 'S%'
    UNION
    SELECT stu_name FROM student WHERE stu_name LIKE '%K%'
    ORDER BY stu_name LIMIT 10;
+----------+
| stu_name |
+----------+
| King     |
| Smith    |
+----------+
2 rows in set
```

这种 ORDER BY 不能使用包括表名的列引用（如，tbl_name.col_name 格式的名字）。相对的，可以在第一个 SELECT 语句中提供一个列的别名并在 ORDER BY 中引用这个别名。

示例 2：对于单独的 SELECT 应用 ORDER BY 或 LIMIT 时，将子句插入到封装 SELECT 的圆括号中。

```
gbase> (SELECT c_name FROM customer WHERE c_name LIKE '%00002%' ORDER BY c_name
LIMIT 10)
    UNION
    (SELECT c_name FROM customer WHERE c_name LIKE '%00003%'ORDER BY c_name
```

```
LIMIT 10);
+--------------------+
| c_name             |
+--------------------+
| Customer#000000002 |
| Customer#000000020 |
| Customer#000000021 |
| Customer#000000022 |
| Customer#000000023 |
| Customer#000000024 |
| Customer#000000025 |
| Customer#000000026 |
| Customer#000000027 |
| Customer#000000028 |
| Customer#000000003 |
| Customer#000000030 |
| Customer#000000031 |
| Customer#000000032 |
| Customer#000000033 |
| Customer#000000034 |
| Customer#000000035 |
| Customer#000000036 |
| Customer#000000037 |
| Customer#000000038 |
+--------------------+
20 rows in set
```

圆括号中的 SELECT 语句的 ORDER BY 只有与 LIMIT 结合才起作用。否则，ORDER BY 将被优化掉。

示例 3：UNION 结果集中列的长度和类型需要考虑到从所有 SELECT 语句中查出的值。

```
gbase> SELECT REPEAT('a',1) UNION SELECT REPEAT('b',10);
+---------------+
| REPEAT('a',1) |
+---------------+
```

```
| a             |
| bbbbbbbbbb    |
+---------------+
2 rows in set
```

示例 4：系统报告错误信息，是因为 LIMIT 1 会产生二义性，执行器不知道 LIMIT 1 是 UNION 后执行，还是 SELECT c_name FROM ssbm.customer LIMIT 1 后再 UNION SELECT c_custkey FROM ssbm.customer。

```
gbase> SELECT c_custkey FROM customer UNION SELECT c_name FROM customer LIMIT
1;
ERROR 1733 (HY000): Gbase general error: UNION/INTERSECT/MINUS of non-matching
columns: LONGLONG UNION/INTERSECT/MINUS VARCHAR
```

示例 5：datetime 和 timestamp 类型的 UNION。

```
gbase> CREATE TABLE t_student (sno int, sname varchar(100),sdate datetime);
Query OK, 0 rows affected

gbase> INSERT INTO t_student VALUES(1,'Tom',NOW()),(2,'Jim',NOW());
Query OK, 2 rows affected
Records: 2  Duplicates: 0  Warnings: 0

gbase> SELECT * FROM t_student;
+------+-------+---------------------+
| sno  | sname | sdate               |
+------+-------+---------------------+
|    1 | Tom   | 2013-12-23 14:34:36 |
|    2 | Jim   | 2013-12-23 14:34:36 |
+------+-------+---------------------+
2 rows in set

gbase> CREATE TABLE t_Result (sno int, sresult decimal(10,2),sdate timestamp);
Query OK, 0 rows affected

gbase> INSERT INTO t_result VALUES(1,99.5,NOW()),(2,100,NOW());
Query OK, 2 rows affected
```

```
Records: 2  Duplicates: 0  Warnings: 0

gbase> SELECT * FROM t_result;
+------+---------+---------------------+
| sno  | sresult | sdate               |
+------+---------+---------------------+
|    1 |   99.50 | 2013-12-23 14:34:59 |
|    2 |  100.00 | 2013-12-23 14:34:59 |
+------+---------+---------------------+
2 rows in set

gbase> SELECT sno,sdate FROM t_student UNION SELECT sno,sdate FROM t_result;
+------+---------------------+
| sno  | sdate               |
+------+---------------------+
|    1 | 2013-12-23 14:34:36 |
|    2 | 2013-12-23 14:34:36 |
|    1 | 2013-12-23 14:34:59 |
|    2 | 2013-12-23 14:34:59 |
+------+---------------------+
4 rows in set
```

示例 6：CHAR 和 CHAR 类型的 UNION。

```
gbase> USE test;
Query OK, 0 rows affected

gbase> CREATE TABLE t1(a char(10));
Query OK, 0 rows affected

gbase> CREATE TABLE t2(a char(255));
Query OK, 0 rows affected

gbase> CREATE TABLE ta AS SELECT * FROM t1 UNION SELECT * FROM t2;
Query OK, 0 rows affected
Records: 0  Duplicates: 0  Warnings: 0
```

```
gbase> DESC ta;
+-------+-------------+------+-----+---------+-------+
| Field | Type        | Null | Key | Default | Extra |
+-------+-------------+------+-----+---------+-------+
| a     | varchar(255) | YES  |     | NULL    |       |
+-------+-------------+------+-----+---------+-------+
1 row in set
```

4.2.12 INTERSECT

语法格式:

SELECT ...

INTERSECT

SELECT•••;

功能: INTERSECT(交运算符), 返回每个 SELECT 查询结果中相同的结果集, 也就是将多个查询结果集中的公共部分作为最终返回的结果集。另外交运算不忽略空值。

示例 1: SELECT ... INTERSECT SELECT ...

示例中用到的表及数据:

```
USE test;
CREATE TABLE t1 (a int , b varchar(10));
CREATE TABLE t2 (c int ,d varchar(20),e varchar(5));
INSERT INTO t1 VALUES(1,'a'),(2,'b'),(3,'c');
INSERT INTO t2 VALUES(1,'a','aa'),(2,'b','bb'),(4,'c','cc');
```

```
gbase> SELECT a ,b FROM t1;
+------+------+
| a    | b    |
+------+------+
|    1 | a    |
```

```
|    2 | b    |
|    3 | c    |
+------+------+
3 rows in set

gbase> SELECT c AS a, d AS b FROM t2;
+------+------+
| a    | b    |
+------+------+
|    1 | a    |
|    2 | b    |
|    4 | c    |
+------+------+
3 rows in set

gbase> SELECT a ,b FROM t1 INTERSECT SELECT c AS a, d AS b FROM t2;
+------+------+
| a    | b    |
+------+------+
|    1 | a    |
|    2 | b    |
+------+------+
2 rows in set
```

4.2.13MINUS

语法格式:

SELECT ... MINUS SELECT…;

功能:

MINUS(差运算符)返回结果集为第一个 SELECT 语句的结果集，并且这个结果集的查询结果所包含的信息不能出现在第二个查询语句结果集中。另外差运算不忽略空值。

示例 1: SELECT ...MINUS SELECT...

示例中用到的表及数据:

```
USE test;
CREATE TABLE t1 (a int , b varchar(10));
INSERT INTO t1 VALUES(1,'a'),(2,'b'),(3,'c');
INSERT INTO t1 VALUES(null,null);
```

```
gbase> SELECT a,b FROM t1;
+------+------+
| a    | b    |
+------+------+
|    1 | a    |
|    2 | b    |
|    3 | c    |
| NULL | NULL |
+------+------+
4 rows in set
```

```
gbase> SELECT c AS a, d AS b FROM t2;
+------+------+
| a    | b    |
+------+------+
|    1 | a    |
|    2 | b    |
|    4 | c    |
+------+------+
3 rows in set
```

```
gbase> SELECT a ,b FROM t1 MINUS SELECT c AS a, d AS b FROM t2;
+------+------+
| a    | b    |
+------+------+
|    3 | c    |
| NULL | NULL |
+------+------+
2 rows in set
```

4.2.14 MERGE

4.2.14.1 语法格式

语法格式:

MERGE [INTO] [database_name.]table_name

USING table_reference

ON conditional_exp

[WHEN MATCHED THEN UPDATE SET *col_name1*=expr1 [, *col_name2*=expr2] ...

[WHEN NOT MATCHED THEN INSERT [(*col_name3*,...)] VALUES (expr3,...)]

参数说明如下:

- MERGE [INTO] [*database_name.*]*table_name* 部分, *table_name* 必须是表, 不可以是视图, 可以使用别名。USING table_reference 部分, table_reference 可以是表、查询或视图, 只允许有一个, 可以使用别名。对于 USING 表中的数据, 如果满足指定的条件 (ON 部分), 则按照 UPDATE 部分执行 UPDATE 操作, 如果不满足, 则按照 INSERT 部分执行 INSERT 操作。
- UPDATE 部分不支持 DELETE 子句。
- INSERT 的 VALUES 部分不允许使用 MERGE 表。

使用约束与限制:

1. UPDATE 部分和 INSERT 部分位置不可以颠倒。
2. UPDATE 部分和 INSERT 部分不支持 WHERE 子句。

3. UPDATE 部分和 INSERT 部分可以省略，但不可以同时省略，否则报语法错误。

4. UPDATE 或 INSERT 中的列如果出现多次，不会报错，后指定的列生效，但建议不要依赖于该行为，避免这样使用。

5. 不允许一对多更新：如果 MERGE 表中的一行与 USING 表中的多行符合连接条件，则报错。

4.2.14.2 MERGE示例

示例 1：对 t1 表进行 MERGE 操作。

示例中用到的表及数据如下：

```
DROP TABLE IF EXISTS t1;
CREATE TABLE t1(i int,vc varchar(20),d date,dc decimal(20,3));
INSERT INTO t1 VALUES(1,'one','2013-02-03',11.21);
INSERT INTO t1 VALUES (2,'two','2013-04-03',12.21);
INSERT INTO t1 VALUES (3,'one2','2013-03-03',31.21);
INSERT INTO t1 VALUES (11,'one3','2013-08-03',41.21);
INSERT INTO t1 VALUES (14,'three','2013-07-22',161.218);
INSERT INTO t1 VALUES (33,'third','2013-09-04',11.216);
INSERT INTO t1 VALUES (5,'wto','2013-02-03',110.210);
INSERT INTO t1 VALUES (null,'first','2013-02-03',311.91);
INSERT INTO t1 VALUES (8,'five','2013-02-03',811.201);

DROP TABLE IF EXISTS t2;
CREATE TABLE t2(i int,vc varchar(20),d date,dc decimal(30,3));
INSERT INTO t2 VALUES (1,'one','2013-02-03',11.20);
INSERT INTO t2 VALUES (2,'two','2013-08-03',12.81);
INSERT INTO t2 VALUES (13,'one2','2013-09-03',31.01);
INSERT INTO t2 VALUES (110,'one3','2013-08-03',41.21);
INSERT INTO t2 VALUES (14,'three','2013-06-22',161.218);
INSERT INTO t2 VALUES (30,'third','2013-09-04',11.216);
```

-- MERGE 操作前，分别查询 t1 表和 t2 表的原始数据。

```
gbase> SELECT * FROM t1 ORDER BY i;
+------+-------+------------+---------+
| i    | vc    | d          | dc      |
+------+-------+------------+---------+
|    1 | one   | 2013-02-03 |  11.210 |
|    2 | two   | 2013-04-03 |  12.210 |
|    3 | one2  | 2013-03-03 |  31.210 |
|    5 | wto   | 2013-02-03 | 110.210 |
|    8 | five  | 2013-02-03 | 811.201 |
|   11 | one3  | 2013-08-03 |  41.210 |
|   14 | three | 2013-07-22 | 161.218 |
|   33 | third | 2013-09-04 |  11.216 |
| NULL | first | 2013-02-03 | 311.910 |
+------+-------+------------+---------+
9 rows in set

gbase> SELECT * FROM t2 ORDER BY i;
+------+-------+------------+---------+
| i    | vc    | d          | dc      |
+------+-------+------------+---------+
|    1 | one   | 2013-02-03 |  11.200 |
|    2 | two   | 2013-08-03 |  12.810 |
|   13 | one2  | 2013-09-03 |  31.010 |
|   14 | three | 2013-06-22 | 161.218 |
|   30 | third | 2013-09-04 |  11.216 |
|  110 | one3  | 2013-08-03 |  41.210 |
+------+-------+------------+---------+
6 rows in set

gbase> MERGE INTO t1 USING t2 ON t1.i=t2.i WHEN MATCHED THEN UPDATE SET
t1.vc=t2.vc WHEN NOT MATCHED THEN INSERT(t1.i,t1.vc) VALUES(t2.i,t2.vc);
Query OK, 6 rows affected
Rows matched: 6  Changed: 6  Warnings: 0

gbase> SELECT * FROM t1 ORDER BY i;
+------+-------+------------+---------+
| i    | vc    | d          | dc      |
```

```
+------+-------+------------+---------+
|    1 | one   | 2013-02-03 |  11.210 |
|    2 | two   | 2013-04-03 |  12.210 |
|    3 | one2  | 2013-03-03 |  31.210 |
|    5 | wto   | 2013-02-03 | 110.210 |
|    8 | five  | 2013-02-03 | 811.201 |
|   11 | one3  | 2013-08-03 |  41.210 |
|   13 | one2  | NULL       |    NULL |
|   14 | three | 2013-07-22 | 161.218 |
|   30 | third | NULL       |    NULL |
|   33 | third | 2013-09-04 |  11.216 |
|  110 | one3  | NULL       |    NULL |
| NULL | first | 2013-02-03 | 311.910 |
+------+-------+------------+---------+
12 rows in set
```

示例 2：t1 表使用别名 t，然后进行 MERGE 操作。

示例中用到的表及数据如下：

```
DROP TABLE IF EXISTS t1;
CREATE TABLE t1(i int,vc varchar(20),d date,dc decimal(20,3)) ;
INSERT INTO t1 VALUES(1,'one','2013-02-03',11.21);
INSERT INTO t1 VALUES (2,'two','2013-04-03',12.21);
INSERT INTO t1 VALUES (3,'one2','2013-03-03',31.21);
INSERT INTO t1 VALUES (11,'one3','2013-08-03',41.21);
INSERT INTO t1 VALUES (14,'three','2013-07-22',161.218);
INSERT INTO t1 VALUES (33,'third','2013-09-04',11.216);
INSERT INTO t1 VALUES (5,'wto','2013-02-03',110.210);
INSERT INTO t1 VALUES (null,'first','2013-02-03',311.91);
INSERT INTO t1 VALUES (8,'five','2013-02-03',811.201);

DROP TABLE IF EXISTS t2;
CREATE TABLE t2(i int,vc varchar(20),d date,dc decimal(30,3));
INSERT INTO t2 VALUES (1,'one','2013-02-03',11.20);
INSERT INTO t2 VALUES (2,'two','2013-08-03',12.81);
INSERT INTO t2 VALUES (13,'one2','2013-09-03',31.01);
INSERT INTO t2 VALUES (110,'one3','2013-08-03',41.21);
```

```
INSERT INTO t2 VALUES (14,'three','2013-06-22',161.218);
INSERT INTO t2 VALUES (30,'third','2013-09-04',11.216);
```

-- MERGE 操作前，分别查询 t1 表和 t2 表的原始数据。

```
gbase> SELECT * FROM t1 ORDER BY i;
+------+-------+------------+---------+
| i    | vc    | d          | dc      |
+------+-------+------------+---------+
|    1 | one   | 2013-02-03 |  11.210 |
|    2 | two   | 2013-04-03 |  12.210 |
|    3 | one2  | 2013-03-03 |  31.210 |
|    5 | wto   | 2013-02-03 | 110.210 |
|    8 | five  | 2013-02-03 | 811.201 |
|   11 | one3  | 2013-08-03 |  41.210 |
|   14 | three | 2013-07-22 | 161.218 |
|   33 | third | 2013-09-04 |  11.216 |
| NULL | first | 2013-02-03 | 311.910 |
+------+-------+------------+---------+
9 rows in set

gbase> SELECT * FROM t2 ORDER BY i;
+------+-------+------------+---------+
| i    | vc    | d          | dc      |
+------+-------+------------+---------+
|    1 | one   | 2013-02-03 |  11.200 |
|    2 | two   | 2013-08-03 |  12.810 |
|   13 | one2  | 2013-09-03 |  31.010 |
|   14 | three | 2013-06-22 | 161.218 |
|   30 | third | 2013-09-04 |  11.216 |
|  110 | one3  | 2013-08-03 |  41.210 |
+------+-------+------------+---------+
6 rows in set
```

-- t1 表使用别名 t 后，继续执行 MERGE 操作。

```
gbase> MERGE INTO t1 t USING t2 ON t.i=t2.i WHEN MATCHED THEN UPDATE SET
t.vc=t2.vc WHEN NOT MATCHED THEN INSERT (t.i,t.vc) VALUES (t2.i,t2.vc);
```

```
Query OK, 6 rows affected
Rows matched: 6  Changed: 6  Warnings: 0

gbase> SELECT * FROM t1 ORDER BY i;
+------+-------+------------+---------+
| i    | vc    | d          | dc      |
+------+-------+------------+---------+
|    1 | one   | 2013-02-03 |  11.210 |
|    2 | two   | 2013-04-03 |  12.210 |
|    3 | one2  | 2013-03-03 |  31.210 |
|    5 | wto   | 2013-02-03 | 110.210 |
|    8 | five  | 2013-02-03 | 811.201 |
|   11 | one3  | 2013-08-03 |  41.210 |
|   13 | one2  | NULL       |    NULL |
|   14 | three | 2013-07-22 | 161.218 |
|   30 | third | NULL       |    NULL |
|   33 | third | 2013-09-04 |  11.216 |
|  110 | one3  | NULL       |    NULL |
| NULL | first | 2013-02-03 | 311.910 |
+------+-------+------------+---------+
12 rows in set
```

示例 3：MERGE 操作后，同步更新 TIEMSTAMP 列的值。

示例中用到的表及数据如下：

```
DROP TABLE IF EXISTS  t1_bak, t2_bak;
CREATE TABLE t1_bak( a int, b int );
CREATE TABLE t2_bak( a int, b int );
INSERT INTO t1_bak VALUES ( 1, 1);
INSERT INTO t1_bak VALUES ( 2, 2);
INSERT INTO t1_bak VALUES ( 3, 3);
INSERT INTO t1_bak VALUES ( 4, 4);
INSERT INTO t2_bak VALUES ( 3, 30);
INSERT INTO t2_bak VALUES ( 4, 40);
INSERT INTO t2_bak VALUES ( 5, 50);
INSERT INTO t2_bak VALUES ( 5, 50);
INSERT INTO t2_bak VALUES ( 5, 50);
```

```
INSERT INTO t2_bak VALUES ( 5, 50);
SELECT  * from t1_bak;
SELECT  * from t2_bak;

DROP TABLE IF EXISTS t1;
DROP TABLE IF EXISTS t2;
CREATE TABLE t1(a int, b int, c timestamp, d timestamp default '2001-1-1', e
int);
CREATE TABLE t2(a int, b int, c timestamp);
INSERT INTO t1(a, b) SELECT  * FROM  t1_bak;
INSERT INTO t2(a, b) SELECT  * FROM  t2_bak;
```

-- MERGE 操作前，分别查询 t1 表和 t2 表的原始数据。

```
gbase> select * from t1;
+------+------+---------------------+---------------------+------+
| a    | b    | c                   | d                   | e    |
+------+------+---------------------+---------------------+------+
|    1 |    1 | 2013-12-26 13:07:43 | 2001-01-01 00:00:00 | NULL |
|    2 |    2 | 2013-12-26 13:07:43 | 2001-01-01 00:00:00 | NULL |
|    3 |    3 | 2013-12-26 13:07:43 | 2001-01-01 00:00:00 | NULL |
|    4 |    4 | 2013-12-26 13:07:43 | 2001-01-01 00:00:00 | NULL |
+------+------+---------------------+---------------------+------+
4 rows in set

gbase> SELECT * FROM t2;
+------+------+---------------------+
| a    | b    | c                   |
+------+------+---------------------+
|    3 |   30 | 2013-12-26 13:07:43 |
|    4 |   40 | 2013-12-26 13:07:43 |
|    5 |   50 | 2013-12-26 13:07:43 |
|    5 |   50 | 2013-12-26 13:07:43 |
|    5 |   50 | 2013-12-26 13:07:43 |
|    5 |   50 | 2013-12-26 13:07:43 |
+------+------+---------------------+
6 rows in set
```

-- 进行 MERGE 操作。

```
gbase> merge into t1 using t2 on t1.a = t2.a WHEN MATCHED THEN UPDATE SET t1.b
= t2.b WHEN NOT MATCHED THEN INSERT (t1.a, t1.b) VALUES (t2.a, t2.b);
Query OK, 6 rows affected
Rows matched: 6  Changed: 6  Warnings: 0
```

-- 查看 t1 表中的数据，TIMESTAMP 列同步更新。

```
gbase> SELECT  * FROM t1;
+------+------+---------------------+---------------------+------+
| a    | b    | c                   | d                   | e    |
+------+------+---------------------+---------------------+------+
|    1 |    1 | 2013-12-26 13:08:22 | 2001-01-01 00:00:00 | NULL |
|    2 |    2 | 2013-12-26 13:08:22 | 2001-01-01 00:00:00 | NULL |
|    3 |   30 | 2013-12-26 13:08:22 | 2001-01-01 00:00:00 | NULL |
|    4 |   40 | 2013-12-26 13:08:22 | 2001-01-01 00:00:00 | NULL |
|    5 |   50 | 2013-12-26 13:08:25 | 2001-01-01 00:00:00 | NULL |
|    5 |   50 | 2013-12-26 13:08:25 | 2001-01-01 00:00:00 | NULL |
|    5 |   50 | 2013-12-26 13:08:25 | 2001-01-01 00:00:00 | NULL |
|    5 |   50 | 2013-12-26 13:08:25 | 2001-01-01 00:00:00 | NULL |
+------+------+---------------------+---------------------+------+
8 rows in set
```

4.3 分级查询语句

GBase 8a 的 SELECT 语句支持使用 START WITH...CONNECT BY 语句实现分级查询。

4.3.1 相关概念

在使用 START WITH...CONNECT BY 语句前，请先了解以下概念。

- 数据源：单物理表、单逻辑表或单个视图。

- 迭代种子：START WITH 指定的过滤条件，基于数据源过滤出来的数据称为迭代种子，迭代种子都是根节点，如果省略 START WITH，那么所有数据源都是根节点。
- 迭代关系：CONNECT BY 指定的关联条件，父子节点之间通过迭代关系进行关联。
- 分级查询：基于迭代种子和数据源，根据迭代关系遍历数据，称为分级查询，也称树形查询或层次查询。即，基于 START WITH 指定的根节点和原始表数据，根据 CONNECT BY 指定的关系遍历数据
- 环路：如果一个节点的子孙节点同时又是该节点的祖先节点，称为存在环路。
- 兄弟间排序：按 KEY 值顺序遍历数据，遍历规则是树的深度优先遍历中的先序遍历。
- 伪列：伪列并不存储在表中，但是它的行为却类似于表中的字段，可以基于伪列做查询操作，但不能更新伪列；伪列也可以理解为没有参数的函数，区别是：基于不同的行，伪列的值一般是不同的。
- 集：某些特定数据集合，如父节点集是指当前层的上层节点数据集合，子节点集是指当前层的数据集合，祖先节点集是指当前层的所有祖先节点组成的数据集合。

4.3.2 执行原理

遍历表中的每条记录，对比是否满足 START WITH 后的条件，如果不满足则继续下一条，如果满足则以该记录为根节点，然后递归寻找该节点下的子节点，查找条件是 CONNECT BY 后面指定的条件。

4.3.3 语法格式

```
SELECT column_list|[LEVEL]
```

```
FROM single_table | view
[WHERE ...]
[hierarchical_clause]
[GROUP BY ...]
[ORDER [SIBLINGS] BY ...]

hierarchical_clause:
  [START WITH <conditions>] CONNECT BY <connect_conditions>
  | CONNECT BY <connect_conditions> [START WITH <conditions>]
  [ORDER SIBLINGS BY {col_name | expr | position} [ASC | DESC], ...]

connect_condition:
  PRIOR expr1 op expr2
  | expr1 op PRIOR expr2
  | expr op connect_condition
  | expr
```

LEVEL：可选关键字，表示等级。即该节点在树中的层数，根节点为第一层，level 为 1

WHERE：根据 CONNECT BY <connect_conditions> START WITH<connect_conditions>选择出来的记录进行过滤，是针对单条记录的过滤。WHERE 条件限制查询返回的行。

hierarchical_clause 中参数说明如下：

START WITH <conditions>: START WITH 表示开始的记录，限定作为搜索起始点的条件，如果是自上而下的搜索则是限定作为根节点的条件，如果是自下而上的搜索则是限定作为叶子节点的条件。

CONNECT BY <conditions>: 这里执行的过滤会把符合条件的记录及其下所有子节点都过滤掉。

CONNECT BY PRIOR 指定与当前记录关联时的字段关系。

START WITH 和 CONNECT BY 的先后顺序不影响查询结果。

ORDER SIBLINGS BY: 使用此关键字时可以进行层次查询

connect_condition 中参数说明如下:

PRIOR: 表示上一条记录。可以指定与当前记录关联时的字段关系。包含此参数时表示不进行递归查询。

4.3.4 使用约束

START WITH...CONNECT BY 语句在使用时有以下约束:

1、分级查询只能基于单物理表或单隐式表 (derived table) 。

以下语句写法错误:

SELECT LEVEL FROM t1, t1 t2 WHERE t1.i = t2.i CONNECT BY t1.i =PRIOR t1.j GROUP BY LEVEL ORDER BY LEVEL;

2、CONNECT BY 关联条件不能包含 OR 操作，并且必须包含父子节点间的等值条件，等号的两边必须是不同的维度 (一边包含 PRIOR，一边不包含 PRIOR) 。

以下语句写法正确:

SELECT * FROM t1 START WITH i = 1 CONNECT BY LEVEL < 4 and PRIOR j = i ORDER SIBLINGS BY j;

以下语句写法错误：

SELECT * FROM t1 START WITH i = 1 CONNECT BY LEVEL < 4 and PRIOR j > i ORDER SIBLINGS BY j;

SELECT * FROM t1 START WITH i = 1 CONNECT BY PRIOR j = i or PRIOR i = j ORDER SIBLINGS BY j;

3、PRIOR 是一元操作符，优先级同正负号，只能用在 CONNECT BY 子句中；PRIOR 后面不可以接伪列（LEVEL、ROWID 等）、不可以接包含伪列的表达式、不可以嵌套使用 PRIOR，不可以接聚合函数。

以下语句写法错误：

SELECT 1 FROM t1 START WITH i = 1 CONNECT BY j = PRIOR (ROWID+1);

SELECT 1 FROM t1 START WITH i = 1 CONNECT BY j = PRIOR LEVEL;

SELECT 1 FROM t1 START WITH i = 1 CONNECT BY j = PRIOR SUM(i);

SELECT 1 FROM t1 START WITH i = 1 CONNECT BY j = PRIOR (i + PRIOR 1);

4、ORDER SIBLINGS BY 只能用在分级查询语句中，并且不能同聚合同时存在

以下语句写法错误：

SELECT * FROM t1 START WITH i = 1 CONNECT BY PRIOR j = i GROUP BY i,j ORDER SIBLINGS BY j;

5、LEVEL 伪列只能用于分级查询语句中。

以下语句写法错误：

SELECT LEVEL FROM t1;

6、不支持实时环路判断，只能保证最终能够检测出环路，如果数据量太大（如超过十万），会耗时很长才能检测出。

7、最大节点数为 MAX_INT(2147483647)

8、不能基于做过 DLETE 的表进行分级查询，但可以修改为隐式表的形式。

DELETE FROM t1 WHERE…;

以下语句写法错误：

SELECT … FROM t1 CONNECT BY…;

以下语句写法正确：

SELECT … FROM (SELECT * FROM t1) t1 CONNECT BY…;

9、在配置文件“gbase_8a_gbase8a.cnf”中增加参数 gbase_max_allowed_level，用于控制允许出现的最大 level 数，默认值为 1024，最大 2147483647，最小 1。

本版本不支持以下功能：

- 不支持 NOCYCLE 参数，如果使用该参数，有环路也不会报错
- 不支持 CONNECT_BY_ISCYCLE 伪列，是否为产生环路的节点
- 不支持 CONNECT_BY_ISLEAF 伪列，该节点是否为叶子节点
- 不支持 SYS_CONNECT_BY_PATH 函数
- 不支持 CONNECT_BY_ROOT 操作符

4.3.5 分级查询语句示例

4.3.5.1 表

建表语句如下：

```
DROP TABLE t1;
CREATE TABLE t1(i int, j int, l varchar(10), k double);
CREATE TABLE t2 LIKE t1;
```

```
INSERT INTO t1 VALUES
(1,2,'2013-1-2',1.2),(1,5,'2013-1-5',1.5),(1,8,'2013-1-8',1.8);
INSERT INTO t1 VALUES (5,3,'2013-5-3',5.3),(5,4,'2013-5-4',5.4);
INSERT INTO t1 VALUES (2,7,'2013-2-7',2.7),(2,6,'2013-2-6',2.6);
INSERT INTO t1 VALUES (8,9,'2013-8-9',8.9),(8,0,'2013-8-1',8.0);
INSERT INTO t1 VALUES (0,10,'2011-1-10',0.1);
INSERT INTO t2 SELECT * FROM t1;
```

示例 1: START WITH...CONNECT BY PRIOR...

```
gbase> SELECT * FROM t1 START WITH i = 1 CONNECT BY PRIOR j = i ORDER BY j,
l DESC, k;
+------+------+-----------+------+
| i    | j    | l         | k    |
+------+------+-----------+------+
|    8 |    0 | 2013-8-1  |    8 |
|    1 |    2 | 2013-1-2  |  1.2 |
|    5 |    3 | 2013-5-3  |  5.3 |
|    5 |    4 | 2013-5-4  |  5.4 |
|    1 |    5 | 2013-1-5  |  1.5 |
|    2 |    6 | 2013-2-6  |  2.6 |
|    2 |    7 | 2013-2-7  |  2.7 |
|    1 |    8 | 2013-1-8  |  1.8 |
|    8 |    9 | 2013-8-9  |  8.9 |
|    0 |   10 | 2011-1-10 |  0.1 |
+------+------+-----------+------+
10 rows in set
```

示例 2: START WITH...CONNECT BY PRIOR...ORDER SIBLINGS BY...

```
gbase> SELECT * FROM t1 START WITH i = 1 CONNECT BY PRIOR j = i AND (1 OR k
= i) > 0 ORDER SIBLINGS BY j, l DESC, k;
+------+------+-----------+------+
| i    | j    | l         | k    |
+------+------+-----------+------+
|    8 |    0 | 2013-8-1  |    8 |
|    1 |    2 | 2013-1-2  |  1.2 |
|    5 |    3 | 2013-5-3  |  5.3 |
```

```
|    5 |    4 | 2013-5-4   |  5.4 |
|    1 |    5 | 2013-1-5   |  1.5 |
|    2 |    6 | 2013-2-6   |  2.6 |
|    2 |    7 | 2013-2-7   |  2.7 |
|    1 |    8 | 2013-1-8   |  1.8 |
|    8 |    9 | 2013-8-9   |  8.9 |
|    0 |   10 | 2011-1-10  |  0.1 |
+------+------+------------+------+
10 rows in set
```

示例 3：CONNECT BY PRIOR 与 CONNECT BY。

```
gbase> SELECT * FROM t1 CONNECT BY PRIOR j = i;
+------+------+------------+------+
| i    | j    | l          | k    |
+------+------+------------+------+
|    1 |    2 | 2013-1-2   |  1.2 |
|    1 |    5 | 2013-1-5   |  1.5 |
|    1 |    8 | 2013-1-8   |  1.8 |
|    5 |    3 | 2013-5-3   |  5.3 |
|    5 |    4 | 2013-5-4   |  5.4 |
|    2 |    7 | 2013-2-7   |  2.7 |
|    2 |    6 | 2013-2-6   |  2.6 |
|    8 |    9 | 2013-8-9   |  8.9 |
|    8 |    0 | 2013-8-1   |    8 |
|    0 |   10 | 2011-1-10  |  0.1 |
+------+------+------------+------+
10 rows in set

gbase> SELECT * FROM t1 CONNECT BY j = i;
+------+------+------------+------+
| i    | j    | l          | k    |
+------+------+------------+------+
|    1 |    2 | 2013-1-2   |  1.2 |
|    1 |    5 | 2013-1-5   |  1.5 |
|    1 |    8 | 2013-1-8   |  1.8 |
|    5 |    3 | 2013-5-3   |  5.3 |
```

```
|    5 |    4 | 2013-5-4  |  5.4 |
|    2 |    7 | 2013-2-7  |  2.7 |
|    2 |    6 | 2013-2-6  |  2.6 |
|    8 |    9 | 2013-8-9  |  8.9 |
|    8 |    0 | 2013-8-1  |    8 |
|    0 |   10 | 2011-1-10 |  0.1 |
+------+------+-----------+------+
10 rows in set
```

示例 4: SELECT * FROM table WHERE ...START WITH...CONNECT BY PRIOR

```
gbase> SELECT * FROM t1 WHERE i < 1 START WITH i = 1 CONNECT BY PRIOR j = i
ORDER BY j, l DESC, k;
+------+------+-----------+------+
| i    | j    | l         | k    |
+------+------+-----------+------+
|    0 |   10 | 2011-1-10 |  0.1 |
+------+------+-----------+------+
1 row in set
```

4.3.5.2 视图

视图支持分级查询，即 START WITH...CONNECT BY PRIOR 的使用。

示例 1: START WITH...CONNECT BY PRIOR...

```
gbase> DROP VIEW IF EXISTS v1;
Query OK, 0 rows affected

gbase> CREATE VIEW v1 AS SELECT * FROM t1 START WITH i = 1 CONNECT BY PRIOR
j = i ORDER BY j, l DESC, k;
Query OK, 0 rows affected

gbase> SELECT * FROM v1;
+------+------+-----------+------+
| i    | j    | l         | k    |
```

```
+------+------+-----------+------+
|    8 |    0 | 2013-8-1  |    8 |
|    1 |    2 | 2013-1-2  |  1.2 |
|    5 |    3 | 2013-5-3  |  5.3 |
|    5 |    4 | 2013-5-4  |  5.4 |
|    1 |    5 | 2013-1-5  |  1.5 |
|    2 |    6 | 2013-2-6  |  2.6 |
|    2 |    7 | 2013-2-7  |  2.7 |
|    1 |    8 | 2013-1-8  |  1.8 |
|    8 |    9 | 2013-8-9  |  8.9 |
|    0 |   10 | 2011-1-10 |  0.1 |
+------+------+-----------+------+
10 rows in set

gbase> SHOW CREATE VIEW v1;
+------+--------------------+----------------------+----------------------+
| View | Create View        | character_set_client | collation_connection |
+------+--------------------+----------------------+----------------------+
| v1   | CREATE ALGORITHM=UNDEFINED DEFINER="root"@"%" SQL SECURITY DEFINER
VIEW "v1" AS select "t1"."i" AS "i","t1"."j" AS "j","t1"."l" AS "l","t1"."k"
AS "k" from "t1" start with ("t1"."i" = 1) connect by (prior "t1"."j" = "t1"."i")
order by "t1"."j","t1"."l" desc,"t1"."k" | utf8                 |
utf8_general_ci     |
+------+--------------------+----------------------+----------------------+
1 row in set
```

示例 2: START WITH...CONNECT BY PRIOR...ORDER SIBLINGS BY...

```
gbase> DROP VIEW v1;
Query OK, 0 rows affected

gbase> CREATE VIEW v1 AS SELECT * FROM t1 START WITH i = 1 CONNECT BY PRIOR
j = i AND (1 OR k = i) > 0 ORDER SIBLINGS BY j, l DESC, k;
Query OK, 0 rows affected

gbase> SELECT * FROM v1;
+------+------+-----------+------+
```

```
| i    | j    | l          | k    |
+------+------+------------+------+
|    1 |    2 | 2013-1-2   |  1.2 |
|    2 |    6 | 2013-2-6   |  2.6 |
|    2 |    7 | 2013-2-7   |  2.7 |
|    1 |    5 | 2013-1-5   |  1.5 |
|    5 |    3 | 2013-5-3   |  5.3 |
|    5 |    4 | 2013-5-4   |  5.4 |
|    1 |    8 | 2013-1-8   |  1.8 |
|    8 |    0 | 2013-8-1   |    8 |
|    0 |   10 | 2011-1-10  |  0.1 |
|    8 |    9 | 2013-8-9   |  8.9 |
+------+------+------------+------+
10 rows in set
```

```
gbase> SHOW CREATE VIEW v1;
+------+--------------------+----------------------+----------------------+
| View | Create View        | character_set_client | collation_connection |
+------+--------------------+----------------------+----------------------+
| v1   | CREATE ALGORITHM=UNDEFINED DEFINER="root"@"%" SQL SECURITY DEFINER
VIEW "v1" AS select "t1"."i" AS "i","t1"."j" AS "j","t1"."l" AS "l","t1"."k"
AS "k" from "t1" start with ("t1"."i" = 1) connect by ((prior "t1"."j" = "t1"."i")
and ((1 or ("t1"."k" = "t1"."i")) > 0)) order siblings by "t1"."j","t1"."l"
desc,"t1"."k" | utf8                 | utf8_general_ci      |
+------+--------------------+----------------------+----------------------+
1 row in set
```

示例 3：WHERE...START WITH...CONNECT BY PRIOR...

```
gbase> DROP VIEW v1;
Query OK, 0 rows affected

gbase> CREATE VIEW v1 AS SELECT * FROM t1 WHERE i < 1 START WITH i = 1 CONNECT
BY PRIOR j = i ORDER BY j, l DESC, k;
Query OK, 0 rows affected

gbase> SELECT * FROM v1;
```

```
+------+------+-----------+------+
| i    | j    | l         | k    |
+------+------+-----------+------+
|    0 |   10 | 2011-1-10 |  0.1 |
+------+------+-----------+------+
1 row in set

gbase> SHOW CREATE VIEW v1;
+------+--------------+----------------------+----------------------+
| View | Create View  | character_set_client | collation_connection |
+------+--------------+----------------------+----------------------+
| v1   | CREATE ALGORITHM=UNDEFINED DEFINER="root"@"%" SQL SECURITY DEFINER
VIEW "v1" AS select "t1"."i" AS "i","t1"."j" AS "j","t1"."l" AS "l","t1"."k"
AS "k" from "t1" where ("t1"."i" < 1) start with ("t1"."i" = 1) connect by (prior
"t1"."j" = "t1"."i") order by "t1"."j","t1"."l" desc,"t1"."k" | utf8
| utf8_general_ci      |
+------+---------------+----------------------+----------------------+
1 row in set
```

4.4 查询结果导出语句

GBase 8a 提供了 SELECT ... INTO OUTFILE 语法，可以将数据从数据库中导出。

4.4.1 查询结果导出语法

语法格式：

SELECT ... INTO OUTFILE 'file_name' [OPTION] FROM ...;

SELECT... FROM... INTO OUTFILE 'file_name' [OPTION];

参数说明如下：

file_name：导出数据保存的路径及文件名，如果导出时，目标文件已经存在，则系统报告错误信息。

OPTION 参数说明如下表所示：

OPTION 选项	说　明
FIELDS/COLUMNS TERMINATED BY	字段分隔符，支持多个字符，如果不指定分隔符，则默认值为“\t”。
FIELDS/COLUMNS [OPTIONALLY] ENCLOSED BY	字段包围符，可以自行指定单个字符为字段包围符，指定多个字符时报错。 支持 OPTIONALLY 选项，加 OPTIONALLY 选项时仅对字符串类型起作用，否则对所有字段都起作用。 默认为无字段包围符。
FIELDS/COLUMNS ESCAPED BY	转义标识符，可以自行指定单个字符为转义标识符，指定多个字符时报错，默认为“\”，默认值在语句中写为：FIELDS ESCAPED BY '\\'。
LINES TERMINATED BY	行分隔符，支持多个字符，默认为“\n”。
LINES STARTING BY	行起始符，支持多个字符，默认为“”。

OPTION 语法约束：

1、 可指定多个字符为字段分隔符，默认为“\t”。

2、 可指定单个字符为字段包围符，指定多个包围符时报错。支持 OPTIONALLY 选项，加 OPTIONALLY 选项时仅对字符串类型起作用，否则对所有字段都起作用。默认为无字段包围符。

3、 可指定单个字符为转义标识符，指定多个转义符时报错。

4、 可指定多个字符为换行符，默认为“\n”。

5、 可指定多个字符为行首分隔符，默认为“”。

6、 如果需要同时设置多个 FIELDS/COLUMNS 和 LINES 的参数，则需要注意的是：所有 FIELDS/COLUMNS 参数的设置都应先于任何 LINES 参数，即 FIELDS/COLUMNS 在前设置，LINES 在后设置，且二者不能相互交叉混合设置；

7、 在一条完整的 SELECT ... INTO OUTFILE 语句中，FIELDS/COLUMNS 或

LINES 保留字仅可出现一次。

SELECT...INTO OUTFILE...支持导出复杂 SQL 语句的查询结果。

4.4.2 查询结果导出的保存路径及规则

SELECT...FROM TABLE_NAME INTO OUTFILE 主要是让用户能够在服务主机上快速地导出一张表或部分表（可以设定过滤条件）的内容。如果用户希望将结果文件建立在其它的主机上，而不是本地 GBase 服务器上，用户就不能使用 SELECT ... INTO OUTFILE。

导出路径支持如下分类：

- 支持相对路径。
- 支持绝对路径。

导出结果保存路径的保存规则：

绝对路径保存规则
如果指定为绝对路径则导出文件就在该路径下； 例如：’/home/save/文件名’

相对路径保存规则	
使用 use db_name，选定数据库	不使用 use db_name，选定数据库
导出文件的路径为“GBase 8a 配置文件中 datadir 字段值” + db_name + SQL 语句中设定的相对路径	导出文件的路径为“GBase 8a 配置文件中 datadir 字段值” + SQL 语句中设定的相对路径

数据导出路径及文件的使用约束：

1、 导出的文件已存在，系统提示文件已存在；

2、 在导出路径中，如果当前用户不具有创建文件的权限，则系统报错；

3、 指定路径时，如果路径前有空格，此时系统不能自动去掉路径前的空格，而是按相对路径处理，如果此路径实际不存在，则报错。例如：

```
gbase> SELECT * FROM t INTO OUTFILE '   /tmp/1.txt' FIELDS TERMINATED BY ';';
ERROR 1016 (HY000): Can't open file: 'Can't create/write to file
/home/wzx/GBase/userdata/gbase8a/   /tmp/1.txt' (errno: 1093187808)
```

4.4.3 定长导出模式

使用定长模式的前提：

1、 所有字段的最大长度均小于 16MB (16777216B)。

2、 定长导出模式参数定义如下表：

参数	参数设定值
FIELD TERMINATER BY	''
FIELD ENCLOSED BY	''
FIELD ESCAPED BY	''

示例：

```
SELECT * FROM t INTO OUTFILE '~/outfile' FIELDS TERMINATATED BY '' ENCLOSED
BY '' ESCAPED BY '';
```

定长模式的数据导出方式：

导出时各个字段都按照其字段的最大长度导出，如果该字段数据的实际长度小于该字段的最大长度，则以空格补齐。

定长模式下的转义符设置：

在定长模式下，需强制指定转义符为空，此时导出的结果是定长且对任何字符均不转义。

4.4.4 转义字符导出模式

4.4.4.1 判定 enclosed 的值是否为 TRUE

满足下列判断条件时，enclosed 的值为 TRUE：

- 如果该字段是字符串类型，且通过 FIELDS ENCLOSED BY 设定非空包围符，则 ENCLOSED 的值为 TRUE。
- 如果该字段是非字符串类型，且通过 FIELDS ENCLOSED BY 设定了非空包围符，在如下两种模式中，ENCLOSED 的值为 TRUE。
  - 指定空字段分隔符，且对非空包围符加上 OPTIONALLY 关键字，例如：FIELDS TERMINATED BY '' OPTIONALLY ENCLOSED BY '"'。
  - 非空包围符不加 OPTIONALLY 关键字，例如：FIELDS ENCLOSED BY '"'。

4.4.4.2 字符型数据的转义规则

字符型数据类型如下表所示：

字符型数据类型	
DATE	DATETIME
TIMESTAMP	TIME
CHAR	VARCHAR
BLOB	TEXT

满足下列判断条件之一，则字符 x 需要进行转义：

1、 字符 x 等于转义符首字符，

2、 字符 x 等于行分隔符首字符（FIELDS TERMINATED BY）；

3、 字符 x 等于“\0”；

4、 enclosed 的值为 TURE，并且字符 x 等于 FIELDS ENCLOSED BY 设置的字段包围符首字符；

5、 enclosed 的值为 TURE，并且字符 x 等于 FIELDS TERMINATED BY 设置的字段分隔符首字符。

说明：enclosed 值的判定规则请参见**“错误!未找到引用源。 错误!未找到引用源。”**

转义规则的说明：

1、 正常情况下，使用 FIELDS ESCAPED BY 关键字定义的转义符对字符进行转义；

2、 如果指定 FIELDS ENCLOSED BY 关键字的值为“n、t、r、b、0、Z、N”之一，并且字符与“字段包围符首字符”相同时，使用字符本身对自己进行转义。

示例中用到的表及数据：

```
DROP TABLE IF EXISTS t;
CREATE TABLE t(n int, v1 varchar(5), v2 varchar(8));
INSERT INTO t VALUES(102, 'ab', 'xmny');
```

示例：

```
gbase> SELECT * FROM t INTO OUTFILE '/home/save/1.txt' FIELDS ENCLOSED BY 'n';
Query OK, 1 row affected
```

查看导出结果，“xmny”中字符“n”使用其本身进行了转义：

```
$ cat 1.txt
n102n    nabn    nxmnnyn
```

3、 如果指定 FIELDS ENCLOSED BY 关键字的值不属于“n、t、r、b、0、Z、N”之一时，则采用转义符进行转义的方式。

示例：

```
gbase> SELECT * FROM t INTO OUTFILE '/home/save/2.txt' FIELDS ENCLOSED BY 'n';
Query OK, 1 row affected
```

查看导出结果，“xmny”中字符“m”使用默认的转义符“\”进行转义：

```
$ cat 2.txt
m102m    mabm    mx\mnym
```

4.4.4.3 非字符型数据的转义规则

正常情况下，转义符仅仅对字符型字符进行转义，但是在一些特殊的情况下，对于非字符型字符也可以进行转义。

非字符型数据类型如下表所示：

字符型数据类型	
TINYINT	NYINT
SMALLINT	MEDIUMINT
BIGINT	BOOL
FLOAT	DOUBLE
DECIMAL	YEAR

非字符型字符进行转义的情况：

1、 如果指定“字段包围符首字符”(FIELDS ENCLOSED BY)是特殊字符“.、0、1、2、3、4、5、6、7、8、9、e、+、-”之一时，才会对非字符型数据类型进行转义处理。

备注：

指定“字段包围符首字符”为某特殊字符的方式有两种：

- 直接指定某特殊字符为“字段包围符”的首字符，例如：

```
SELECT * FROM gs INTO OUTFILE '~/1.txt' FIELDS ENCLOSED BY '0';
```

- 指定“字段包围符”为空，且指定“字段分隔符”的首字符为某特殊字符，例如：

```
SELECT * FROM gs INTO OUTFILE '~/1.txt' FIELDS TERMINATED BY '0' ENCLOSED BY '';
```

2、 如果指定“字段包围符首字符”为“ntrb0ZN”之一时，实际上只能为字符“0”，再进行导出，采用的是用其本身进行转义的方式。

示例中用到的表及数据：

```
DROP TABLE IF EXISTS t;
CREATE TABLE t(n int, v1 varchar(5), v2 varchar(8));
INSERT INTO t VALUES(102, 'ab', 'xmny');
```

示例：

```
gbase> SELECT * FROM t INTO OUTFILE '/home/save/5.txt' FIELDS ENCLOSED BY '0';
Query OK, 1 row affected
```

查看导出结果，非字符型字符“102”中的“0”使用本身进行转义：

```
$ cat 5.txt
01002O   0ab0     0xmny0
```

3、 如果指定“字段包围符首字符(FIELDS ENCLOSED BY)”不属于“ntrb0ZN”之一时，则采用的是用转义符进行转义的方式。

示例：

```
gbase> SELECT * FROM t INTO OUTFILE '/home/save/6.txt' FIELDS ENCLOSED BY '2';
Query OK, 1 row affected
```

查看导出结果，非字符型数据“102”中的“2”使用默认的转义符“\”进行了转义：

```
$ cat 6.txt
210\22   2ab2     2xmny2
```

特例说明：

如果导出字符包含“\0”，则导出的结果为“转义符+0”（参见“4.4.6.3 数据中含有“\0”字符的处理”中的示例）。

4.4.4.4 两种不转义的特殊情况

字符型数据不转义的特殊情况：

如果设定字段包围符（FIELDS ENCLOSED BY）为空，导出的字段中的字符与字段分隔符首字符（FIELDS TERMINATED BY）相同，并且字段分隔符（FIELDS

TERMINATED BY）首字符是“n、t、r、b、0、Z、N”之一。这种情况下是不对导出的字符进行转义的。

示例中用到的表及数据：

```
DROP TABLE IF EXISTS t;
CREATE TABLE  t(n int, v1 varchar(5), v2 varchar(8));
INSERT INTO t VALUES(102, 'ab', 'xmny');
```

```
gbase> SELECT * FROM t INTO OUTFILE '/home/save/unescaped_1.txt' FIELDS
TERMINATED BY 'n' ENCLOSED BY '';
```

查看导出文件：

```
$ cat unescaped_1.txt
102nabnxmny
```

可见字符串中“xmny”中的字符“n”没有转义。

非字符型数据不转义的特殊情况：

如果 opt_enclosed 判定为真（判断条件请见说明部分），当导出数据中某字符与 FIELDS TERMINATED BY 设置的字段分隔符首字符相同，并且字段分隔符首字符是“.、0、1、2、3、4、5、6、7、8、9、e、+、-”之一时，这种情况下是不对导出的字符进行转义的。

说明：

满足如下判断条件之一时，opt_enclosed为真：

TERMINATED BY ';' OPTIONALLY ENCLOSED BY '"';

ENCLOSED BY '';

OPTIONALLY ENCLOSED BY '';

不写 ENCLOSED BY 子句。

示例中用到的表及数据：

```
DROP TABLE IF EXISTS t;
CREATE TABLE  t(n int, v1 varchar(5), v2 varchar(8));
INSERT INTO t VALUES(102, 'ab', 'xmny');
```

示例 1：使用“FIELDS TERMINATED BY '2' OPTIONALLY ENCLOSED BY '"'”，非字符型数据“102”中的“2”没有转义。

```
gbase> SELECT * FROM t INTO OUTFILE '/home/save/unescaped_2.txt' FIELDS
TERMINATED BY '2' OPTIONALLY ENCLOSED BY '"';
Query OK, 1 row affected
```

查看导出文件：

```
$ cat unescaped_2.txt
1022"ab"2"xmny"
```

示例 2：使用“FIELDS TERMINATED BY '2' ENCLOSED BY ''”，非字符型数据“102”中的“2”没有转义。

```
gbase> SELECT * FROM t INTO OUTFILE '/home/save/unescaped_3.txt' FIELDS
TERMINATED BY '2' ENCLOSED BY '';
Query OK, 1 row affected
```

查看导出文件：

```
$ cat unescaped_3.txt
1022ab2xmny
```

示例 3：“FIELDS TERMINATED BY '2' OPTIONALLY ENCLOSED BY ''”，非字符型数据“102”中的“2”没有转义。

```
gbase> SELECT * FROM t INTO OUTFILE '/home/save/unescaped_4.txt' FIELDS
TERMINATED BY '2' OPTIONALLY ENCLOSED BY '';
Query OK, 1 row affected
```

查看导出文件：

```
$ cat unescaped_4.txt
1022ab2xmny
```

示例 4：使用“FIELDS TERMINATED BY '2'”，非字符型数据“102”中的

“2”没有转义。

```
gbase> SELECT * FROM t INTO OUTFILE '/home/save/unescaped_5.txt' FIELDS
TERMINATED BY '2';
Query OK, 1 row affected
```

查看导出文件：

```
$ cat unescaped_5.txt
1022ab2xmny
```

4.4.5 查询结果导出示例

示例中用到的表及数据：

```
CREATE TABLE  gs (a int DEFAULT NULL, b varchar(20) DEFAULT NULL);
CREATE TABLE blob_gs (t int DEFAULT NULL, b blob);
INSERT INTO gs VALUES(3,'c'),(5,'6'),(0,'t'),(9,'\\');
INSERT INTO blob_gs VALUES (2,'1a2b'),(1,'南大通用');
```

4.4.5.1 几种常见的错误写法

示例 1：LINES 参数在 FIELDS/COLUMNS 参数之前定义。

错误写法：

```
gbase> SELECT * FROM test.t INTO OUTFILE '/home/save/a.txt' LINES TERMINATED
BY '\n' FIELDS TERMINATED BY '\t';
```

```
ERROR 1064 (42000): You have an error in your SQL syntax; check the manual that
corresponds to your GBase server version for the right syntax to use near 'FIELDS
TERMINATED BY '\t'' at line 1
```

正确写法：

```
gbase> SELECT * FROM test.t INTO OUTFILE '/home/save/a.txt' FIELDS TERMINATED
```

```
BY '\t' LINES TERMINATED BY '\n';
Query OK, 1 row affected
```

示例 2：所有的 FIELDS/COLUMNS 参数都应在所有的 LINES 参数之前定义完整，二者不可混合使用。

错误写法：

```
gbase> SELECT * FROM test.t INTO OUTFILE '/home/save/a.txt' FIELDS TERMINATED
BY '\t' LINES TERMINATED BY '\n' FIELDS ESCAPED BY '\\';
ERROR 1064 (42000): You have an error in your SQL syntax; check the manual that
corresponds to your GBase server version for the right syntax to use near 'FIELDS
ESCAPED BY '\\'' at line 1
```

正确写法：

```
gbase> SELECT * FROM test.t INTO OUTFILE '/home/save/b.txt' FIELDS TERMINATED
BY '\t' ESCAPED BY '\\' LINES TERMINATED BY '\n';
Query OK, 1 row affected
```

示例 3： FIELDS/COLUMNS 或 LINES 保留字仅可出现一次。

```
gbase> SELECT * FROM test.t INTO OUTFILE '/home/save/c.txt' FIELDS TERMINATED
BY '\t' FIELDS ESCAPED BY '\\';
ERROR 1064 (42000): You have an error in your SQL syntax; check the manual that
corresponds to your GBase server version for the right syntax to use near 'FIELDS
ESCAPED BY '\\'' at line 1
```

正确写法：

```
gbase> SELECT * FROM test.t INTO OUTFILE '/home/save/c.txt' FIELDS TERMINATED
BY '\t' ESCAPED BY '\\';
Query OK, 1 row affected
```

示例 4：如果重复设置某一参数，以最后一次设置的为准。

```
gbase> SELECT * FROM test.t INTO OUTFILE '/home/save/d.txt ' FIELDS TERMINATED
BY '\t' TERMINATED BY ';';
Query OK, 1 row affected
```

查看导出文件：

```
$ cat d.txt
102;ab;xmny
```

等价于

```
gbase> SELECT * FROM test.t INTO OUTFILE '/home/save/e.txt' FIELDS TERMINATED BY ';';
Query OK, 1 row affected
```

查看导出文件：

```
$ cat e.txt
102;ab;xmny
```

4.4.5.2 不指定字段分隔符

示例 1：导出复杂 SQL 语句的查询结果。

示例中用到的表及数据：

```
CREATE TABLE t3 (color_type varchar(20),color_count int, in_date date);
INSERT INTO t3 (color_type,in_date,color_count)
VALUES('black','2010-09-11',18),('black','2010-10-05',18),('black','2010-10-13',31),('blue','2010-09-21',23),('blue','2010-09-30',15),('blue','2010-10-11',62),('red','2010-09-12',41),('red','2010-10-01',12),('red','2010-10-05',11);
```

```
gbase> SELECT NVL(color_type,'') as color_type_show,NVL(DECODE(color_type,NULL,f_YearMonth || '合计',NVL(f_YearMonth,color_type || ' 小计')),'总计') AS f_YearMonth_show,SUM(color_count) FROM (SELECT color_type,DATE_FORMAT(in_date, '%Y-%m') as f_YearMonth,color_count FROM t3) t GROUP BY CUBE(color_type,f_YearMonth) ORDER BY color_type,f_YearMonth into outfile '/home/save/t3.txt';
Query OK, 12 rows affected
```

查看导出文件：

```
$ cat t3.txt
black    2010-09 18
black    2010-10 49
black    black 小计       67
blue     2010-09 38
blue     2010-10 62
blue     blue 小计        100
red      2010-09 41
red      2010-10 23
red      red 小计         64
         2010-09 合计      97
         2010-10 合计      134
         总计     231
```

4.4.5.3 指定字段分隔符

示例 1：指定字段分隔符为多个字符。

```
gbase> SELECT * FROM gs INTO OUTFILE '/home/save/gs_a.txt' FIELDS TERMINATED BY '@#$';
Query OK, 4 rows affected
```

查看导出文件：

```
$ cat gs_a.txt
3@#$c
5@#$6
0@#$t
9@#$\\
```

4.4.5.4 指定字段包围符

示例 1：指定单个字段包围符为“@”。

```
gbase> SELECT * FROM gs INTO OUTFILE '/home/save/gs_b.txt' FIELDS ENCLOSED BY
'@';
Query OK, 4 rows affected
```

查看导出文件：

```
$ cat gs_b.txt;
@3@     @c@
@5@     @6@
@0@     @t@
@9@     @\\@
```

示例 2：无 OPTIONALLY 选项时对所有字段都起作用。

```
gbase> SELECT * FROM gs INTO OUTFILE '/home/save/gs_c.txt' FIELDS ENCLOSED BY
'"';
Query OK, 4 rows affected
```

查看导出文件：

```
$ cat gs_c.txt;
"3"     "c"
"5"     "6"
"0"     "t"
"9"     "\\"
```

示例 3：指定 OPTIONALLY 选项时仅对字符串类型起作用。

```
gbase> SELECT * FROM gs INTO OUTFILE '/home/save/gs_d.txt' FIELDS OPTIONALLY
ENCLOSED BY '"';
Query OK, 4 rows affected
```

查看导出文件：

```
$ cat gs_d.txt;
3       "c"
5       "6"
0       "t"
9       "\\"
```

4.4.5.5 指定转义标识符

示例 1：指定单个转义标识符为“c”。

```
gbase> SELECT * FROM gs;
+------+------+
| a    | b    |
+------+------+
|    3 | c    |
|    5 | 6    |
|    0 | t    |
|    9 | \    |
+------+------+
4 rows in set

gbase> SELECT * FROM gs INTO OUTFILE '/home/save/gs_e.txt' FIELDS ESCAPED BY
'c';
Query OK, 4 rows affected
```

查看导出文件：

```
$ cat gs_e.txt;
3       cc
5       6
0       t
9       \
```

示例 2：指定转义标识符为多个字符时报错。

```
gbase> SELECT * FROM gs INTO OUTFILE '/home/save/gs_e.txt' FIELDS ESCAPED BY
```

```
'6c@#';
ERROR 1210 (HY000): Incorrect arguments to Field separator argument; check the manual
```

4.4.5.6 指定换行符

示例 1：指定多个字符为换行符。

```
gbase> SELECT * FROM gs INTO OUTFILE '/home/save/gs_f.txt' LINES TERMINATED BY '@#$';
Query OK, 4 rows affected
```

查看导出文件：

```
$ cat gs_f.txt
3       c@#$5   6@#$0   t@#$9   \\@#$
```

4.4.5.7 指定行首分隔符

示例 1：指定多个字符为行首分隔符。

```
gbase> SELECT * FROM gs INTO OUTFILE '/home/save/gs_g.txt' LINES STARTING BY '@#$';
Query OK, 4 rows affected
```

查看导出文件：

```
$ cat gs_g.txt;
@#$3    c
@#$5    6
@#$0    t
@#$9    \\
```

4.4.5.8 指定相对路径保存导出文件

示例 1：使用 USE db_name，选定数据库。

```
gbase> USE test
Query OK, 0 rows affected

gbase> SHOW VARIABLES LIKE 'datadir';
+---------------+-----------------------------------+
| Variable_name | Value                             |
+---------------+-----------------------------------+
| datadir       | /home/test/GBase/userdata/gbase8a/ |
+---------------+-----------------------------------+
1 row in set

gbase> SELECT * FROM gs INTO OUTFILE '1.txt' FIELDS TERMINATED BY ';';
Query OK, 4 rows affected
```

进入“GBase/userdata/gbase8a/test”目录下查看 1.txt 文件。

```
$ cd GBase/userdata/gbase8a/test
$ pwd
/home/test/GBase/userdata/gbase8a/test
$ cat 1.txt
3;c
5;6
0;t
9;\\
```

4.4.6 数据导出中特殊情况的示例

4.4.6.1 数据中含有 NULL 值的处理

如果待导出数据中某字段的内容为 NULL 值，则该字段导出的 NULL 文本为

“当前转义符 + N”。

默认情况下的转义符为“\”，因此字段导出的 NULL 文本为“\N”。

示例中用到的表及数据：

```
DROP TABLE IF EXISTS gs;
CREATE TABLE gs (a int DEFAULT NULL, b varchar(20) DEFAULT NULL);
INSERT INTO gs VALUES(NULL,NULL);
INSERT INTO gs VALUES(1, 'GBase');
```

示例 1：转义符默认为“\”，则“NULL”值导出的结果为“\N”。

```
gbase> SELECT * FROM gs INTO OUTFILE '/home/save/null_1.txt ';
Query OK, 2 rows affected
```

查看导出文件：

```
$ cat null_1.txt
\N      \N
1       GBase
```

示例 2：如果在导出语句中指定了字段包围符，则对 NULL 值不起作用。

```
gbase> SELECT * FROM gs INTO OUTFILE '/home/save/null_2.txt' FIELDS ENCLOSED
BY '"';
Query OK, 2 rows affected
```

查看导出文件：

```
$ cat null_2.txt
\N      \N
"1"     "GBase"
```

示例 3：设置转义符为“|”，则“NULL”值导出的结果为“|N”。

```
gbase> SELECT * FROM gs INTO OUTFILE '/home/save/null_3.txt' FIELDS ESCAPED
BY '|';
Query OK, 2 rows affected
```

查看导出文件：

```
$ cat null_3.txt
|N       |N
1        GBase
```

4.4.6.2 转义符为空字符的处理

示例中用到的表及数据：

```
DROP TABLE IF EXISTS gs;
CREATE TABLE gs (a int DEFAULT NULL, b varchar(20) DEFAULT NULL);
INSERT INTO gs VALUES(NULL,NULL);
INSERT INTO gs VALUES(1, 'GBase');
```

示例 1：如果 FIELDS ESCAPED BY '' 中的字符是空字符，则 NULL 被作为 NULL 输出，而不是作为“\N”输出。

```
gbase> SELECT * FROM gs INTO OUTFILE '/home/save/esp_1.txt' FIELDS ESCAPED BY
'';
Query OK, 2 rows affected
```

查看导出文件：

```
$ cat esp_1.txt
NULL     NULL
1        GBase
```

示例 2：如果在导出语句中指定了字段包围符，仍对 NULL 值不起作用。

```
gbase> SELECT * FROM gs INTO OUTFILE '/home/save/esp_2.txt' FIELDS ESCAPED BY
'' ENCLOSED BY '"';
Query OK, 2 rows affected
```

查看导出文件：

```
$ cat esp_2.txt
```

```
NULL    NULL
"1"     "GBase"
```

4.4.6.3 数据中含有“\0”字符的处理

示例 1：如果在导出数据中的某字段（通常为字符串类型，如 varchar）的值为“\0”，在默认情况下，该字符在导出的文本为“\0”。

示例中用到的表及数据：

```
DROP TABLE IF EXISTS gs;
CREATE TABLE gs (a int DEFAULT NULL, b varchar(20) DEFAULT NULL);
INSERT INTO gs VALUES(3,'asdf\0dv');
INSERT INTO gs VALUES(4, 'GBase');

gbase> SELECT * FROM gs INTO OUTFILE '/home/save/test_1.txt';
Query OK, 2 rows affected
```

查看导出文件：

```
$ cat test_1.txt
3       asdf\0dv
4       GBase
```

4.4.6.4 指定多字符为字段分隔符，且文本中也包含多字符分隔符时的处理

示例 1：如果在“SELECT INTO OUTFILE”语句中指定多字符作为分隔符，且字段文本中包含该分隔符串时，则只对该分隔符串的首字符进行转义。

示例中用到的表及数据：

```
DROP TABLE IF EXISTS gs;
CREATE TABLE gs (a int DEFAULT NULL, b varchar(20) DEFAULT NULL);
```

```
INSERT INTO gs VALUES(3,'nihao');
INSERT INTO gs VALUES(4, 'GBase');
```

```
gbase> SELECT * FROM gs INTO OUTFILE '/home/save/test_2.txt' FIELDS TERMINATED BY 'ih';
Query OK, 2 rows affected
```

查看导出文件：

```
$ cat test_2.txt
3ihn\ihao
4ihGBase
```

4.4.6.5 字段文本中包含“\n”或“\r”时的处理

示例中用到的表及数据：

```
DROP TABLE IF EXISTS gs;
CREATE TABLE gs (a int DEFAULT NULL, b varchar(20) DEFAULT NULL);
INSERT INTO gs values(1,'qw\ner'),(2,'as\rdf');
```

如果在导出的数据中某字段（通常为字符串类型，如 varchar）中包含“\n”或“\r”，则只对“\n”进行转义。

如果在“\n”前加转义字符（默认为“\”），“\r”不变，仍为不可见字符“\r”，使用二进制方式查看为“0x0D”。

示例 1：转义“\n”的原因是文本中的内容“\n”与默认的行分隔符(LINES TERMINATED)相同，故将文本中的“\n”转义。

```
gbase> SELECT * FROM gs INTO OUTFILE '/home/save/n_1.txt';
Query OK, 2 rows affected
```

查看导出文件：

```
$ cat -b n_1.txt
     1  1       qw\
```

```
     2  er
df   3  2       as
```

注意：cat命令中的-b参数表示对非空输出行进行编号。

使用二进制方式查看：

```
$ hexdump -C n_1.txt
00000000  31 09 71 77 5c 0a 65 72  0a 32 09 61 73 0d 64 66  |1.qw\.er.2.as.df|
00000010  0a                                                |.|
00000011
```

示例 2：如果显示的指定行分隔符为其他字符，则不发生转义。

```
gbase> SELECT * FROM gs INTO OUTFILE '/home/save/n_2.txt' LINES TERMINATED BY
';';
Query OK, 2 rows affected
```

查看导出文件：

```
$ vi n_2.txt
1       qw
er;2    as^Mdf;
```

注意：导出文件中的“^M”表示“\n”。

4.4.6.6 定长模式的导出

示例 1：定长模式导出数据。

示例中用到的表及数据：

```
DROP TABLE IF EXISTS gs;
CREATE TABLE gs (a int DEFAULT NULL, b varchar(25) DEFAULT NULL);
INSERT INTO gs values(1,'GBase 8a'),(2,'GBase 8a MPP Cluster');
```

```
gbase> SELECT * FROM gs INTO OUTFILE '/home/save/length_1.txt' FIELDS
TERMINATED BY '' ENCLOSED BY '' ESCAPED BY '';
```

```
Query OK, 2 rows affected
```

查看导出文件：

```
$ cat length_1.txt
1           GBase 8a
2           GBase 8a MPP Cluster
```

使用二进制方式查看：

```
$ hexdump -C length_1.txt
00000000  31 20 20 20 20 20 20 20  20 20 20 47 42 61 73 65  |1         GBase|
00000010  20 38 61 20 20 20 20 20  20 20 20 20 20 20 20 20  | 8a            |
00000020  20 20 20 20 20 20 20 20  20 20 20 20 20 20 20 20  |               |
*
00000050  20 20 20 20 20 20 0a 32  20 20 20 20 20 20 20 20  |      .2       |
00000060  20 20 47 42 61 73 65 20  38 61 20 4d 50 50 20 43  |  GBase 8a MPP C|
00000070  6c 75 73 74 65 72 20 20  20 20 20 20 20 20 20 20  |luster         |
00000080  20 20 20 20 20 20 20 20  20 20 20 20 20 20 20 20  |               |
*
000000a0  20 20 20 20 20 20 20 20  20 20 20 20 20 0a        |           .|
000000ae
```

从显示结果可以看出，导出数据时使用了空格补齐。

需要注意的是，一个 varchar 字符可能占用多个字节。如当前字符集为 utf8 时，一个 varchar 字符占用三个字节，如果建表时设置 varchar(10)，则该字段最大长度可能占用 30 个字节。

4.4.6.7 含有 NULL 值的定长模式的导出

示例中用到的表及数据：

```
DROP TABLE IF EXISTS gs;
CREATE TABLE gs (a int DEFAULT NULL, b varchar(25) DEFAULT NULL);
INSERT INTO gs values(NULL,'GBase 8a'),(NULL,NULL);
```

定长模式对于空值 NULL 的导出：根据字段宽度全部使用空格补齐。

例如：对该整形字段 a 插入的值为 NULL，则实际使用长度为 0，使用定长模式导出会使用 11 个空格进行填充。

示例 1：含有 NULL 值的定长模式的导出。

```
gbase> SELECT * FROM gs INTO OUTFILE '/home/save/length_2.txt' FIELDS
TERMINATED BY '' ENCLOSED BY '' ESCAPED BY '';
Query OK, 2 rows affected
```

查看导出文件：

```
$ cat length_2.txt
           GBase 8a
```

使用二进制方式查看：

```
$ hexdump -C length_2.txt
00000000  20 20 20 20 20 20 20 20  20 20 20 47 42 61 73 65  |          GBase|
00000010  20 38 61 20 20 20 20 20  20 20 20 20 20 20 20 20  | 8a            |
00000020  20 20 20 20 20 20 20 20  20 20 20 20 20 20 20 20  |               |
*
00000050  20 20 20 20 20 20 0a 20  20 20 20 20 20 20 20 20  |      .        |
00000060  20 20 20 20 20 20 20 20  20 20 20 20 20 20 20 20  |               |
*
000000a0  20 20 20 20 20 20 20 20  20 20 20 20 20 0a        |           .|
000000ae
```

从显示结果可以看出，对 NULL 值导出时全部使用了空格补齐。

4.5 GBase 8a 其他语句

4.5.1 DESCRIBE

语法格式：

{DESCRIBE | DESC} [*database_name.*]<*table_name*> [*col_name* | wild]

DESCRIBE 提供一个表中的列信息。它是 SHOW COLUMNS FROM 的简便形式。该语句也可以显示视图信息。参见 SHOW COLUMNS 语法。

col_name 可以是一个列名称，或一个包含“%”和“_”的通配符的字符串，用于获得对于带有与字符串相匹配的名称的各列的输出。没有必要在引号中包含字符串，除非其中包含空格或其它特殊字符。

示例 1：查看 customer c_custkey 的列信息。

```
gbase> DESCRIBE customer c_custkey;
+-----------+------------+------+-----+---------+-------+
| Field     | Type       | Null | Key | Default | Extra |
+-----------+------------+------+-----+---------+-------+
| c_custkey | bigint(20) | YES  |     | NULL    |       |
+-----------+------------+------+-----+---------+-------+
1 row in set
```

NULL 列表示是否可以存储 NULL 值，YES 表示可以存储。

KEY 列为空，GBase 8a 没有 key。

DEFAULT 列表示指派给该字段的默认值。

EXTRA 列包含所有附加的关于该字段的有效信息。如果列的类型与在 CREATE TABLE 语句中定义的不同。

SHOW CREATE TABLE 语句也可以提供表的信息。请参见“4.7.3 SHOW 管理语句”。

4.5.2 USE

语法格式：

USE <database_name>

使用 *database_name* 数据库作为以后查询的缺省数据库。数据库保持为当

前数据库，直到该会话结束或另一个 USE 语句发出。

示例 1：使用 ssbm 数据库作为默认数据库。

```
gbase> USE ssbm;
Query OK, 0 rows affected

gbase> SELECT COUNT(*) FROM customer;
+----------+
| COUNT(*) |
+----------+
|    30000 |
+----------+
1 row in set
```

依靠 USE 语句将一个特定数据库设为当前数据库，它并不阻止用户访问另一个数据库中的表。

下面的示例通过当前 GBase 数据库访问 ssbm 数据库中的 customer 表。

示例 2：使用 gbase 数据库作为默认数据库。

```
gbase> USE gbase;
Query OK, 0 rows affected

gbase> SELECT COUNT(*) FROM ssbm.customer;
+----------+
| COUNT(*) |
+----------+
|    30000 |
+----------+
1 row in set
```

4.5.3 KILL

语法格式：

KILL [CONNECTION | QUERY] *thread_id*

用 SHOW PROCESSLIST 语句可以查看正在运行的线程，而用 KILL *thread_id* 语句可以终止一个线程。

参数说明如下：

- KILL CONNECTION 和没选项修饰的 KILL 相同，它用给定的 thread_id 终止相关的连接。
- KILL QUERY 中止连接当前执行的语句，但是不中止该连接本身。

如果有 PROCESS 权限，可以查看所有线程。

如果有 SUPER 权限，可以终止所有线程和语句。否则，用户只能查看并终止自己的线程和语句。

当用户执行一个 KILL 命令，对应线程的 thread-specific 标志被置位。在大多数情况下，结束线程可能花费一些时间，因为只有在特定时期才检查该标志。

在 SELECT，ORDER BY 和 GROUP BY 循环中，在读取一部分行后被检查 kill 标志。如果 kill 标志被置位，该语句终止。

在 ALTER TABLE 期间，在从源表中读取表的每一个部分前检查 kill 标志。如果被置位，该语句中止并且删除临时表。

4.5.4 SET

语法格式：

SET [GLOBAL | SESSION] <*variablename*> = value

参数说明如下：

SESSION：省略掉 SESSION 关键字，也就是默认情况下，是会话（SESSION）级别的，则变量值在会话结束之前或重赋值之前是保持不变的。

GLOBAL：设置为此关键字时，新的变量值被用于新的连接当中。

示例 1：默认为会话级别，只在当前节点机器上的当前连接有效。

```
gbase> SET AUTOCOMMIT = 1;
Query OK, 0 rows affected
```

示例 2：使用 GLOBAL 关键字，设置“gbase_sql_trace”的值为“on”。

```
gbase> SHOW VARIABLES LIKE '%trace%';
+---------------------------+-------+
| Variable_name             | Value |
+---------------------------+-------+
| _gbase_sql_trace_file_mode | OFF   |
| auto_trace                | OFF   |
| gbase_sql_trace           | OFF   |
| gbase_sql_trace_level     | 0     |
+---------------------------+-------+
4 rows in set

gbase> SET GLOBAL gbase_sql_trace =on;
Query OK, 0 rows affected
```

设置成功后，查看参数值。

```
gbase> SHOW VARIABLES LIKE '%gbase_sql_trace%';
+---------------------------+-------+
| Variable_name             | Value |
+---------------------------+-------+
| _gbase_sql_trace_file_mode | OFF   |
| gbase_sql_trace           | OFF   |
| gbase_sql_trace_level     | 0     |
+---------------------------+-------+
3 rows in set
```

退出连接后，重新查看参数值。

```
gbase> QUIT
Bye
```

```
$ gbase -uroot -p
Enter password:

GBase client 8.5.1.2 build 27952. Copyright (c) 2004-2013, GBase.  All Rights
Reserved.

gbase> SHOW VARIABLES LIKE '%trace%';
+---------------------------+-------+
| Variable_name             | Value |
+---------------------------+-------+
| _gbase_sql_trace_file_mode | OFF   |
| auto_trace                | OFF   |
| gbase_sql_trace           | ON    |
| gbase_sql_trace_level     | 0     |
+---------------------------+-------+
4 rows in set
```

4.6 GBase 8a 事务和锁语句

4.6.1 START TRANSACTION, COMMIT 和 ROLLBACK 语法

```
START TRANSACTION | BEGIN [WORK]

COMMIT [WORK] [AND [NO] CHAIN] [[NO] RELEASE]

ROLLBACK [WORK] [AND [NO] CHAIN] [[NO] RELEASE]

SET AUTOCOMMIT = {0 | 1}
```

START TRANSACTION 或 BEGIN 语句可以开始一项新的事务。

COMMIT 可以提交当前事务，是变更成为永久变更。

ROLLBACK 可以回滚当前事务，取消其变更。

SET AUTOCOMMIT 语句可禁用或启用默认的 AUTOCOMMIT 模式,用于当前连接。

缺省情况下 GBase 8a 运行在 AUTOCOMMIT=1 的模式下。这就意味着，当用户执行完一个更新时，GBase 8a 将立刻将更新存储到磁盘上。

通过下面的命令，用户可以设置 GBase 8a 为非 AUTOCOMMIT 模式:

SET AUTOCOMMIT=0;

通过把 AUTOCOMMIT 变量设置为零，禁用 AUTOCOMMIT 模式之后，您必须使用 COMMIT 把变更存储到磁盘中，或者如果您想要放弃从事务开始进行以来做出的变更，使用 ROLLBACK。

如果您想要对于一个单一系列的语句禁用 AUTOCOMMIT 模式，则您可以使用 START TRANSACTION 语句。

使用 START TRANSACTION，AUTOCOMMIT 仍然被禁用，直到您使用 COMMIT 或 ROLLBACK 结束事务为止。然后 AUTOCOMMIT 模式恢复到原来的状态。

注意: GBase 8a 只支持 INSERT 级别的事务。

示例 1: 提交事务

```
gbase> USE test;
Query OK, 0 rows affected

gbase> SET AUTOCOMMIT = 0;
Query OK, 0 rows affected

gbase> START TRANSACTION;
Query OK, 0 rows affected

gbase> DROP TABLE IF EXISTS t_Year;
Query OK, 0 rows affected

gbase> CREATE TABLE t_Year(f_YearID int,f_Year year);
Query OK, 0 rows affected

gbase> INSERT INTO t_Year VALUES(1,2012);
```

```
Query OK, 1 row affected

gbase> SELECT * FROM t_Year;
Empty set

gbase> COMMIT;
Query OK, 0 rows affected

gbase> SELECT * FROM t_Year;
+----------+--------+
| f_YearID | f_Year |
+----------+--------+
|        1 |   2012 |
+----------+--------+
1 row in set
```

-- 恢复自动提交模式

```
gbase> SET AUTOCOMMIT = 1;
Query OK, 0 rows affected
```

示例 2：回滚事务

```
gbase> SET AUTOCOMMIT = 0;
Query OK, 0 rows affected

gbase> START TRANSACTION;
Query OK, 0 rows affected

gbase> INSERT INTO t_Year VALUES(2,2013);
Query OK, 1 row affected

gbase> SELECT * FROM t_Year;
+----------+--------+
| f_YearID | f_Year |
+----------+--------+
|        1 |   2012 |
+----------+--------+
```

```
1 rows in set

gbase> ROLLBACK;
Query OK, 0 rows affected

gbase> SELECT * FROM t_Year;
+----------+--------+
| f_YearID | f_Year |
+----------+--------+
|        1 |   2012 |
+----------+--------+
1 rows in set
```

4.6.1.1 INSERT 事务示例

示例 1: INSERT INTO CUSTOMER() VALUES();

```
gbase> SELECT COUNT(*) FROM customer;
+----------+
| COUNT(*) |
+----------+
|    30000 |
+----------+
1 row in set

gbase> SET AUTOCOMMIT=0;
Query OK, 0 rows affected

gbase> START TRANSACTION;
Query OK, 0 rows affected

gbase> INSERT INTO CUSTOMER() VALUES();
Query OK, 1 row affected

gbase> SELECT COUNT(*) FROM customer;
+----------+
```

```
| COUNT(*) |
+----------+
|    30000 |
+----------+
1 row in set

gbase> COMMIT;
Query OK, 0 rows affected

gbase> SELECT COUNT(*) FROM customer;
+----------+
| COUNT(*) |
+----------+
|    30001 |
+----------+
1 row in set
```

示例 2：对于 INSERT 语句，进行多次写操作后，最后提交事务。

```
gbase> DROP TABLE IF EXISTS products;
Query OK, 0 rows affected

gbase> CREATE TABLE products(nameid int, name varchar(50));
Query OK, 0 rows affected

gbase> SET AUTOCOMMIT = 0;
Query OK, 0 rows affected

gbase> INSERT INTO products (nameid,name) VALUES(1,'Tom');
Query OK, 1 row affected

gbase> INSERT INTO products (nameid,name) VALUES(2,'Tom');
Query OK, 1 row affected

gbase> SELECT * FROM products;
Empty set
```

```
-- 全部回滚

gbase> ROLLBACK;
Query OK, 0 rows affected

gbase> SELECT * FROM products;
Empty set

gbase> SET AUTOCOMMIT = 0;
Query OK, 0 rows affected

gbase> INSERT INTO products (nameid,name) VALUES(1,'Tom');
Query OK, 1 row affected

gbase> INSERT INTO products (nameid,name) VALUES(2,'Tom');
Query OK, 1 row affected

gbase> SELECT * FROM products;
Empty set

-- 全部提交

gbase> COMMIT;
Query OK, 0 rows affected

gbase> SELECT * FROM products;
+--------+------+
| nameid | name |
+--------+------+
|      1 | Tom  |
|      2 | Tom  |
+--------+------+
2 rows in set
```

4.6.1.2 UPDATE 事务示例

示例 1：UPDATE 的事务

```
gbase> SET AUTOCOMMIT = 0;
Query OK, 0 rows affected

gbase> UPDATE products SET name = 'Rose' WHERE nameid = 2;
Query OK, 1 row affected
Rows matched: 1  Changed: 1  Warnings: 0

gbase> SELECT * FROM products;
+--------+------+
| nameid | name |
+--------+------+
|      1 | Tom  |
|      2 | Tom  |
+--------+------+
2 rows in set

-- 回滚事务
gbase> ROLLBACK;
Query OK, 0 rows affected

gbase> SELECT * FROM products;
+--------+------+
| nameid | name |
+--------+------+
|      1 | Tom  |
|      2 | Tom  |
+--------+------+
2 rows in set

gbase> UPDATE products SET name = 'Rose' WHERE nameid = 2;
Query OK, 1 row affected
Rows matched: 1  Changed: 1  Warnings: 0

-- 提交事务
```

```
gbase> COMMIT;
Query OK, 0 rows affected

gbase> SELECT * FROM products;
+--------+------+
| nameid | name |
+--------+------+
|      1 | Tom  |
|      2 | Rose |
+--------+------+
2 rows in set
```

示例 2: INSERT, UPDATE 混合的事务

```
gbase> SET AUTOCOMMIT = 0;
Query OK, 0 rows affected

gbase> INSERT INTO products (nameid ,name) VALUES(3,'White');
Query OK, 1 row affected

gbase> INSERT INTO products (nameid ,name) VALUES(4,'Jerry');
Query OK, 1 row affected
```

-- 此时数据文件被锁定，不能在插入操作后，进行数据更新，因为插入数据还没有提交入库。

```
gbase> UPDATE products SET name = 'Jerry Lee' WHERE nameid = 4;
ERROR 1015 (HY000): Can't lock file (errno: 1)
```

-- 提交事务

```
gbase> COMMIT;
Query OK, 0 rows affected
```

-- 仅将多次标准的 INSERT 操作进行了提交

```
gbase> SELECT * FROM products;
+--------+-------+
```

```
| nameid | name  |
+--------+-------+
|      1 | Tom   |
|      2 | Rose  |
|      3 | White |
|      4 | Jerry |
+--------+-------+
4 rows in set
```

-- 再更新数据

```
gbase> UPDATE products SET name = 'Jerry Lee' WHERE nameid = 4;
Query OK, 1 row affected
Rows matched: 1  Changed: 1  Warnings: 0

gbase> SELECT * FROM products;
+--------+-------+
| nameid | name  |
+--------+-------+
|      1 | Tom   |
|      2 | Rose  |
|      3 | White |
|      4 | Jerry |
+--------+-------+
4 rows in set
```

-- 提交事务

```
gbase> COMMIT;
Query OK, 0 rows affected
```

-- 更新成功

```
gbase> SELECT * FROM products;
+--------+-----------+
| nameid | name      |
+--------+-----------+
|      1 | Tom       |
```

```
|      2 | Rose       |
|      3 | White      |
|      4 | Jerry Lee  |
+--------+------------+
4 rows in set
```

4.6.1.3 快速 UPDATE 模式事务示例

快速 UPDATE 模式，其本质就是先删除数据，然后在表的末尾追加新的纪录。在数据量大时，可以开启并行，以提高效率。

当设置 SET gbase_fast_update=1;时，表明开启了快速 UPDATE 模式。

示例 1：开启快速 UPDATE 模式。

```
gbase> DROP TABLE IF EXISTS t1;
Query OK, 0 rows affected

gbase> CREATE TABLE t1 (f_1 int);
Query OK, 0 rows affected

gbase> INSERT INTO t1 values(1),(2),(3);
Query OK, 3 rows affected
Records: 3  Duplicates: 0  Warnings: 0

gbase> COMMIT;
Query OK, 0 rows affected

gbase> SELECT * FROM t1;
+------+
| f_1  |
+------+
|    1 |
|    2 |
|    3 |
+------+
```

```
3 rows in set

gbase> SET gbase_fast_update = 1;
Query OK, 0 rows affected

gbase> SET AUTOCOMMIT = 0;
Query OK, 0 rows affected

gbase> UPDATE t1 SET f_1 = 10 WHERE f_1= 1;
Query OK, 1 row affected
Rows matched: 1  Changed: 1  Warnings: 0

gbase> SELECT * FROM t1;
+------+
| f_1  |
+------+
|    1 |
|    2 |
|    3 |
+------+
3 rows in set

gbase> COMMIT;
Query OK, 0 rows affected

gbase> SELECT * FROM t1;
+------+
| f_1  |
+------+
|    3 |
|    3 |
|   10 |
+------+
3 rows in set
```

4.6.1.4 DELETE 事务示例

示例 1：DELETE 的事务。

```
gbase> SET AUTOCOMMIT = 0;
Query OK, 0 rows affected

gbase> DELETE FROM products WHERE nameid = 3;
Query OK, 1 row affected

gbase> SELECT * FROM products;
+--------+----------+
| nameid | name     |
+--------+----------+
|      1 | Tom      |
|      2 | Rose     |
|      3 | White    |
|      4 | Jerry Lee |
+--------+----------+
4 rows in set

gbase> COMMIT;
Query OK, 0 rows affected

gbase> SELECT * FROM products;
+--------+----------+
| nameid | name     |
+--------+----------+
|      1 | Tom      |
|      2 | Rose     |
|      4 | Jerry Lee |
+--------+----------+
3 rows in set
```

4.6.2 不能回滚的语句

一些语句是不能被回滚的。一般地，这些语句包括数据定义的语言（DDL），比如那些创建或删除数据库的语句以及创建、删除、更改表或存储程序（过程和函数）的语句。

应使设计的事务中不包含这些语句。如果在事务中使用了不能回滚的语句，那么当后面的语句失败时，使用 ROLLBACK 语句并不能回滚事务的全部影响。

4.7 数据库管理语句

4.7.1 帐号管理语句

GBase 8a 有两种用户：

- 本地用户（user@localhost），这个用户是 GBase 服务器上的用户，所谓的 localhost 指的就是服务器本地。

可以通过这个用户在服务器上登录到 GBase 8a 系统，并执行权限下的操作，一般该用户拥有 root 权限，也可以创建一个新的本地用户，分配较低的权限。执行一些简单的操作。

- 远程用户（user@%），这个用户是客户机由于登录服务器而使用的用户，所谓的“%”指的是远程任何机器。如果把“%”写为一个具体的 IP 地址，则是指定远程某台 IP 地址的用户。

可以通过这个用户在客户机上通过客户端工具登录到服务器执行操作。一般这类用户权限较低，除非被分配高级权限。

4.7.1.1 CREATE USER

语法格式：

CREATE USER *user* [IDENTIFIED BY [*password*]]

参数说明如下：

user：帐号名称。

password：帐号密码。

CREATE USER 语句创建一个新的 GBase 8a 帐号。执行此操作，用户必须有全局 CREATE USER 权限或 system 数据库的 INSERT 权限。对每一个帐号，CREATE USER 在 gbase.user 表中创建一条新的记录，这个帐号初始只有登录数据库的权限。如果这个用户已经存在，就会发生错误。

CREATE USER 语句每次仅能添加一个用户帐号，不能同时增加多个用户帐号。

通过可选择的 IDENTIFIED BY 语句可以给帐号赋予一个密码。特别注意的是，密码设置为纯文本格式可以省略 PASSWORD 关键字。

示例 1：创建 admin 用户。

```
gbase> CREATE USER admin IDENTIFIED BY 'admin';
Query OK, 0 rows affected

gbase> EXIT
Bye
$ gbase -uadmin -p
Enter password:

GBase client 8.5.1.2 build 27952. Copyright (c) 2004-2013, GBase.  All Rights Reserved.

gbase>
```

4.7.1.2 DROP USER

语法格式：

DROP USER *user* ;

功能：

删除 GBase 8a 帐号。

要使用该语句必须有全局 DROP USER 权限或 gbase 数据库 DELETE 权限。

DROP USER 不会自动关闭任何打开的用户会话。更确切地说，当存在使用此用户打开的会话时，删除此用户并不会影响这些已打开会话的访问权限，直到这些会话关闭。

示例 1：删除 admin 用户。

```
gbase> DROP USER admin;
Query OK, 0 rows affected
```

4.7.1.3 RENAME USER

RENAME USER *old_user* TO *new_user*

RENAME USER 语句用来重命名存在的 GBase 8a 帐号。要使用该语句，必须有全局 CREATE USER 权限或对 gbase 数据库的 UPDATE 权限。如果旧的帐号不存在或是新的帐号已经存在，那么就会报错。可以按照和 GRANT 语句同样的方法给 *old_user* 和 *new_user* 赋值。

示例 1：更改 admin 为 admin1。

```
gbase> CREATE USER admin IDENTIFIED BY 'admin';
Query OK, 0 rows affected

gbase> SELECT TRIM(host) host, TRIM(user) user,password FROM gbase.user;
+-----------+---------+-------------------------------------------+
| host      | user    | password                                  |
+-----------+---------+-------------------------------------------+
| localhost | root    | *FD571203974BA9AFE270FE62151AE967ECA5E0AA |
| %         | root    |                                           |
| 127.0.0.1 | gbase   | *9C0ADBD7F08FA9D49D82760B104110C55B943B8D |
| %         | gbase   | *9C0ADBD7F08FA9D49D82760B104110C55B943B8D |
| 127.0.0.1 | sys_man | *D2B871A7CAF3F973EC6F201EFEEEE7A6D3106544 |
| %         | admin   | *4ACFE3202A5FF5CF467898FC58AAB1D615029441 |
```

```
+-----------+---------+-------------------------------------------+
6 rows in set

gbase> RENAME USER admin to admin1;
Query OK, 0 rows affected

gbase> SELECT TRIM(host) host, TRIM(user) user,password FROM gbase.user;
+-----------+---------+-------------------------------------------+
| host      | user    | password                                  |
+-----------+---------+-------------------------------------------+
| localhost | root    | *FD571203974BA9AFE270FE62151AE967ECA5E0AA |
| %         | root    |                                           |
| 127.0.0.1 | gbase   | *9C0ADBD7F08FA9D49D82760B104110C55B943B8D |
| %         | gbase   | *9C0ADBD7F08FA9D49D82760B104110C55B943B8D |
| 127.0.0.1 | sys_man | *D2B871A7CAF3F973EC6F201EFEEEE7A6D3106544 |
| %         | admin1  | *4ACFE3202A5FF5CF467898FC58AAB1D615029441 |
+-----------+---------+-------------------------------------------+
6 rows in set
```

4.7.1.4 SET PASSWORD

SET PASSWORD [FOR *user*] =

PASSWORD('*newpassword*')

FOR user: 指定修改密码的帐户名称，如果省略，就是进行修改当前登录 GBase 8a 帐户的的密码。

PASSWORD('*newpassword*'): 指定帐户的新密码；

示例 1: 首先使用 admin1 进行登录，SET PASSWORD 省略 FOR admin1 的写法，就是为当前的登录的 admin1 用户修改密码。

```
gbase> SET PASSWORD = PASSWORD('adminnew');
Query OK, 0 rows affected
gbase> EXIT
Bye
```

```
$ gbase -uadmin1 -p
Enter password:

GBase client 8.5.1.2 build 27952. Copyright (c) 2004-2013, GBase.  All Rights
Reserved.

gbase>
```

示例 2：使用 admin1 进行登录，SET PASSWORD 省略 FOR admin1 的写法，就是为当前的登录的 admin1 用户使用 OLD_PASSWORD 修改密码。为了说明 OLD_PASSWORD()和 PASSWORD()功能一样。

```
gbase> SET PASSWORD = OLD_PASSWORD('admin');
Query OK, 0 rows affected
gbase> EXIT
Bye

$ gbase -uadmin1 -p
Enter password:

GBase client 8.5.1.2 build 27952. Copyright (c) 2004-2013, GBase.  All Rights
Reserved.

gbase>
```

4.7.2 权限管理

4.7.2.1 GRANT 和 REVOKE

语法格式：

```
GRANT
    priv_type [(column_list)]
```

```
    [, priv_type [(column_list)]] ...
    ON [object_type] priv_level
    TO user [IDENTIFIED BY [PASSWORD] 'password']
    [WITH with_option ...]

object_type:
    TABLE  | FUNCTION  | PROCEDURE

priv_level:
    *  | *.*  | database_name.*  | database_name.table_name
  | table_name  | database_name.routine_name

REVOKE
    priv_type [(column_list)] [, priv_type [(column_list)]] ...
    ON [object_type] priv_level
    FROM user
REVOKE ALL PRIVILEGES, GRANT OPTION
FROM user
```

4.7.2.2 权限级别

GRANT 和 REVOKE 语句允许系统管理员创建 GBase 8a 用户帐号，并处理用户权限的赋予与收回。

对于 GRANT 和 REVOKE 语句，priv_type 指定为下列任一种。

权　限	意　义
ALL [PRIVILEGES]	设置除 GRANT OPTION 之外的所有简单权限
ALTER	允许使用 ALTER TABLE
ALTER ROUTINE	更改或取消已存储的子程序
CREATE	允许使用 CREATE TABLE
CREATE ROUTINE	创建已存储的子程序
CREATE TEMPORARY TABLES	允许使用 CREATE TEMPORARY TABLE
CREATE USER	允许使用 CREATE USER, DROP USER, RENAME USER 和 REVOKE ALL PRIVILEGES。
CREATE VIEW	允许使用 CREATE VIEW
DELETE	允许使用 DELETE
DROP	允许使用 DROP TABLE
EXECUTE	允许用户运行已存储的子程序
FILE	允许使用 SELECT...FROM TABLE_NAME INTO OUTFILE 等
GRANT OPTION	允许授予权限
INDEX	允许使用 CREATE INDEX 和 DROP INDEX
INSERT	允许使用 INSERT
PROCESS	允许使用 SHOW FULL PROCESSLIST
REFERENCES	未被实施
RELOAD	允许使用 FLUSH
SELECT	允许使用 SELECT
SHOW DATABASES	SHOW DATABASES 显示所有数据库
SHOW VIEW	允许使用 SHOW CREATE VIEW
SHUTDOWN	允许使用 gbaseadmin shutdown
SUPPER	允许使用 KILL 和 SET GLOBAL 语句
UPDATE	允许使用 UPDATE
USAGE	仅仅用于连接登录数据库，主要用来设置 with option 部分

说明:

1、 数据库权限分为数据库对象操作权限等以下 5 类:

- 数据库对象操作类权限;

- 数据操作类权限；
- 存储过程、自定义函数执行权限；
- 数据查看类权限；
- 数据库权限（包含用户管理）管理权限。

表格中的“ALL”是个特殊权限不在上述分类中，它是把 GRANT OPTION 之外的所有权限赋予指定用户。

2、 要使用 GRANT 或 REVOKE，必须拥有 GRANT OPTION 权限，并且拥有授予或收回权限。

对于 GRANT 和 REVOKE 语句，priv_level 可以授予不同级别的权限：

1、 全局级（Global level）

全局权限应用到给定服务器的所有数据库上。这些权限存储在 gbase.user 表中。GRANT ALL ON *.*和 REVOKE ALL ON *.*只可以授予和收回全局权限。

2、 数据库级（Database level）

数据库权限应用于给定数据库的所有对象上。这些权限存储在 gbase.db 和 gbase.host 表中。GRANT ALL ON db_name.*和 REVOKE ALL ON db_name.*只可以授予和收回数据库权限。

3、 表级（Table level）

表权限应用于给定表的所有列。这些权限存储在 gbase.tables_priv 表中。GRANT ALL ON db_name.tbl_name 和 REVOKE ALL ON db_name.tbl_name 只可以授予和收回表权限。

4、 列级（column level）

列权限应用于表中的指定列。这些权限存储在 gbase.tables_priv 表中。GRANT SELECT,INSERT,UPDATE(column) ON db_name.tbl_name 和 REVOKE SELECT（column） ON db_name.tbl_name 只可以授予和收回列权限。

示例 1：为 t 表中的列 a 赋予 SELECT 权限。

```
gbase> CREATE TABLE t(a int,b varchar(40));
Query OK, 0 rows affected

gbase> INSERT INTO t VALUES (1,'test'),(2,'share');
Query OK, 2 rows affected
Records: 2  Duplicates: 0  Warnings: 0

gbase> GRANT SELECT(a) ON test.t TO admin;
Query OK, 0 rows affected

gbase> SELECT * FROM gbase.tables_priv;
+-----------+----------+----------+---------------+--------------------+
| Host      | Db       | User     | Table_name    | Grantor            |
+-----------+----------+----------+---------------+--------------------+
| %         | test     | admin    | t             |  root@localhost    |
+-----------+----------+----------+---------------+--------------------+
+---------------------+------------+-------------+
| Timestamp           | Table_priv | Column_priv |
+---------------------+------------+-------------+
| 2013-12-24 09:20:06 |            | Select      |
+---------------------+------------+-------------+
```

示例 2：收回 t 表中列 a 的 SELECT 权限。

```
gbase> REVOKE SELECT(a) ON test.t FROM admin;
Query OK, 0 rows affected

gbase> SELECT * FROM gbase.tables_priv;
Empty set
```

4.7.3 SHOW 管理语句

SHOW 以多种形式提供有关服务器的数据库，表，列或状态方面的信息。这些形式描述如下：

```
SHOW [FULL] COLUMNS FROM table_name [FROM database_name] [LIKE
'pattern']

SHOW CREATE DATABASE database_name

SHOW CREATE  FUNCTION  [database_name.]fun_name

SHOW CREATE  PROCEDURE  [database_name.]proc_name

SHOW CREATE TABLE [database_name.]table_name

SHOW CREATE VIEW [database_name.]view_name

SHOW DATABASES [LIKE 'pattern']

SHOW ERRORS [LIMIT [offset,] row_count]

SHOW FUNCTION STATUS

SHOW GRANTS FOR user_name

SHOW INDEX FROM table_name [FROM database_name]

SHOW OPEN TABLES FROM database_name] [LIKE 'pattern']

SHOW PRIVILEGES

SHOW PROCEDURE STATUS

SHOW [FULL] PROCESSLIST

SHOW [GLOBAL | SESSION] STATUS [LIKE 'pattern']

SHOW TABLE STATUS

SHOW [OPEN] TABLES [FROM database_name] [LIKE 'pattern']

SHOW [GLOBAL | SESSION] VARIABLES [LIKE 'pattern']

SHOW WARNINGS [LIMIT [offset,] row_count]
```

如果一个给定的 SHOW 语句的语法包括一个 LIKE 'pattern'部分，则'pattern'是一个可以包含 SQL “%”和“_”通配符的字符串。对于把语句输出

值限定为匹配值，本样式是有用的。

4.7.3.1 SHOW COLUMNS

SHOW [FULL] COLUMNS FROM table_name [FROM *database_name*] [LIKE 'pattern']

SHOW COLUMNS 显示一个给定表中列的信息。该语句在视图中也适用。

列类型有可能与用户在 CREATE TABLE 语句中所期望的类型不同，GBase 8a 有时会在创建或更改表时改变列的类型。这种情况发生的条件在列规范的隐式变更中介绍。

关键字 FULL 产生的输出包括用户对每个列所拥有的权限。FULL 也显示所有列注释信息。

可以在 table_name FROM database_name 语法中使用 database_name.table_name，下面的两个语句是等价的：

SHOW COLUMNS FROM t FROM test;

SHOW COLUMNS FROM test.t;

SHOW FIELDS 和 SHOW COLUMNS 的作用相同。

示例 1：显示列信息。

```
gbase> SHOW COLUMNS FROM t FROM test;
+---------+-------------+------+-----+---------+-------+
| Field   | Type        | Null | Key | Default | Extra |
+---------+-------------+------+-----+---------+-------+
| name1   | varchar(20) | YES  |     | NULL    |       |
| address | varchar(40) | YES  |     | NULL    |       |
| phone   | varchar(20) | YES  |     | NULL    |       |
| sex     | int(11)     | YES  |     | NULL    |       |
+---------+-------------+------+-----+---------+-------+
4 rows in set
```

```
gbase> SHOW COLUMNS FROM test.t;
+---------+------------+------+-----+---------+-------+
| Field   | Type       | Null | Key | Default | Extra |
+---------+------------+------+-----+---------+-------+
| name1   | varchar(20) | YES  |     | NULL    |       |
| address | varchar(40) | YES  |     | NULL    |       |
| phone   | varchar(20) | YES  |     | NULL    |       |
| sex     | int(11)    | YES  |     | NULL    |       |
+---------+------------+------+-----+---------+-------+
4 rows in set
```

DESCRIBE 语句提供和 SHOW COLUMNS 语句相似的信息。请参考“4.5.1 DESCRIBE”中的内容。

4.7.3.2 SHOW CREATE DATABASE

SHOW CREATE {DATABASE | SCHEMA} *database_name*

显示创建了给定数据库的 CREATE DATABASE 语句。可以用 SHOW CREATE SCHEMA 语句实现相同的功能。

示例 1：显示创建 test 数据库的语句。

```
gbase> SHOW CREATE DATABASE test;
+----------+-----------------------------------------------------------------+
| Database | Create Database                                                 |
+----------+-----------------------------------------------------------------+
| test     | CREATE DATABASE "test" /*!40100 DEFAULT CHARACTER SET utf8 */ |
+----------+-----------------------------------------------------------------+
1 row in set
```

4.7.3.3 SHOW CREATE FUNCTION

SHOW CREATE FUNCTION [*database_name.*]*fun_name*

显示创建了给定函数的 CREATE FUNCTION 语句。

示例 1：显示创建 fn_count 函数的语句。

```
gbase> SHOW CREATE FUNCTION fn_count;
+----------+...-+----------------------------------------+-...-+
| Function | ... | Create Function                        | ... |
+----------+...-+----------------------------------------+-...-+
| fn_count | ... | CREATE ... FUNCTION "fn_count" ... END | ... |
+----------+...-+----------------------------------------+-...-+
1 row in set
```

在显示信息中包含 6 列，每列的具体信息如下。

- Function: fn_count
- sql_mode:
  PIPES_AS_CONCAT,ANSI_QUOTES,IGNORE_SPACE,NO_AUTO_VALUE_ON_ZERO,STRICT_ALL_TABLES,NO_ZERO_IN_DATE,NO_ZERO_DATE,NO_AUTO_CREATE_USER,NO_ENGINE_SUBSTITUTION,PAD_CHAR_TO_FULL_LENGTH
- Create Function: CREATE DEFINER="root"@"%" FUNCTION "fn_count"(param varchar(10)) RETURNS int

```
BEGIN
SELECT COUNT(*)/5 INTO @count FROM ssbm.customer WHERE c_nation= param;
RETURN @count;
END
```

- character_set_client: utf8
- collation_connection: utf8_general_ci
- Database Collation: utf8_general_ci

4.7.3.4 SHOW CREATE PROCEDURE

SHOW CREATE PROCEDURE [*database_name.*]*proc_name*

显示创建了给定存储过程的 CREATE PROCEDURE 语句。

示例 1：显示创建 casedemo 存储过程的语句。

```
gbase> SHOW CREATE PROCEDURE casedemo;
```

```
+-----------+-...-+-----------------------------------------+-...-+
| Procedure | ... | Create Procedure                        | ... |
+-----------+-...-+-----------------------------------------+-...-+
| casedemo  | ... | CREATE ... PROCEDURE "casedemo"() ... END| ... |
+-----------+-...-+-----------------------------------------+-...-+
1 row in set
```

在显示信息中包含 6 列，每列的具体信息如下。

- Procedure: casedemo
- sql_mode:
  PIPES_AS_CONCAT,ANSI_QUOTES,IGNORE_SPACE,NO_AUTO_VALUE_ON_ZERO,STRICT_ALL_TABLES,NO_ZERO_IN_DATE,NO_ZERO_DATE,NO_AUTO_CREATE_USER,NO_ENGINE_SUBSTITUTION,PAD_CHAR_TO_FULL_LENGTH
- Create Procedure: CREATE DEFINER="root"@"%" PROCEDURE "casedemo"()
BEGIN
  SELECT DISTINCT CASE WHEN c_nation='CHINA' THEN '中国' WHEN c_nation='MOROCCO' THEN '摩洛哥' WHEN c_nation='JORDAN' THEN '约旦' ELSE '其它国家' END 中文, c_nation FROM ssbm.customer LIMIT 10;
END
- character_set_client: utf8
- collation_connection: utf8_general_ci
- Database Collation: utf8_general_ci

4.7.3.5 SHOW CREATE TABLE

SHOW CREATE TABLE [*database_name*.]table_name

显示创建了给定表的 CREATE TABLE 语句。该语句在视图中也使用。

示例 1：显示创建 t 表的语句。

```
gbase> SHOW CREATE TABLE test.t;
+-------+-----------------------------------------------------+
| Table | Create Table                                        |
+-------+-----------------------------------------------------+
| t     | CREATE TABLE "t" (
  "a" int(11) DEFAULT NULL
```

```
) ENGINE=EXPRESS DEFAULT CHARSET=utf8 TABLESPACE='sys_tablespace' |
+-------+-----------------------------------------------------------+
1 row in set
```

SHOW CREATE TABLE 根据 SQL_QUOTE_SHOW_CREATE 选项的值显示表和列的名字。

4.7.3.6 SHOW CREATE VIEW

SHOW CREATE VIEW [database_name.]view_name

这个语句显示地使用 CREATE VIEW 创建视图。

示例 1：显示创建 student_v 视图的语句。

```
gbase> SHOW CREATE VIEW student_v;
+----------+-----------------------------------------------------+-...-+
| View     | Create View                                         | ... |
+----------+-----------------------------------------------------+-...-+
| student_v| CREATE ... VIEW `student_v` AS select .... from ... | ... |
+----------+-----------------------------------------------------+-...-+
1 row in set
```

在显示信息中包含 4 列，每列的具体信息如下。

- View: student_v
- Create View: CREATE ALGORITHM=UNDEFINED DEFINER="root"@"%" SQL SECURITY DEFINER VIEW "student_v" AS select "student"."stu_no" AS "stu_no","student"."stu_name" AS "stu_name","student"."stu_sex" AS "stu_sex" from "student"
- character_set_client: utf8
- collation_connection: utf8_general_ci

4.7.3.7 SHOW DATABASES

SHOW {DATABASES | SCHEMAS} [LIKE 'pattern']

SHOW DATABASES 列出在 GBase 8a 服务器主机上的数据库。除非拥有全局的 SHOW DATABASES 权限，用户只能看到自己拥有权限的数据库。

4.7.3.8 SHOW ERRORS

SHOW ERRORS [LIMIT [offset,] row_count]

SHOW COUNT(*) ERRORS 语句和 SHOW WARNINGS 相似，只是显示的内容不是警告和注意，而是仅仅显示错误。

LIMIT 子句与在 SELECT 语句中语法相同。参考 SELECT 语法。

SHOW COUNT(*) ERRORS 语句显示错误的数目，也可以从 error_count 变量中获得错误数：

SHOW COUNT(*) ERRORS;

SELECT @@error_count;

4.7.3.9 SHOW FUNCTION STATUS

SHOW FUNCTION STATUS 显示已经创建成功的函数的状态。

示例 1：显示已经创建成功的函数的状态。

```
gbase> SHOW FUNCTION STATUS;
+--------+----------+----------+-----------+------------------+
| Db     | Name     | Type     | Definer   | Modified         |
+--------+----------+----------+-----------+------------------+
| ssbm   | hello    | FUNCTION | root@%    | 2013-10-22 17:18:21 |
| test   | fn_count | FUNCTION | root@%    | 2013-10-22 17:56:17 |
| test   | hello    | FUNCTION | root@%    | 2013-10-22 17:22:37 |
+--------+----------+----------+-----------+------------------+
+---------------------+---------------+-----------+
| Created             | Security_type | Comment   |
+---------------------+---------------+-----------+
| 2013-10-22 17:18:21 | DEFINER       |           |
```

```
| 2013-10-22 17:56:17 | DEFINER        |            |
| 2013-10-22 17:22:37 | DEFINER        |            |
+---------------------+----------------+------------+
+----------------------+----------------------+--------------------+
| character_set_client | collation_connection | Database Collation |
+----------------------+----------------------+--------------------+
| utf8                 | utf8_general_ci      | utf8_general_ci    |
| utf8                 | utf8_general_ci      | utf8_general_ci    |
| utf8                 | utf8_general_ci      | utf8_general_ci    |
+----------------------+----------------------+--------------------+
3 rows in set
```

4.7.3.10 SHOW GRANTS

SHOW GRANTS FOR *user_name*

该语句列出允许一个 GBase 8a 用户帐号复制权限的 GRANT 语句。

示例 1：为 root 用户赋予权限。

```
gbase> SHOW GRANTS FOR 'root'@'%';
+-------------------------------------------------------------+
| Grants for root@%                                           |
+-------------------------------------------------------------+
| GRANT ALL PRIVILEGES ON *.* TO 'root'@'%' WITH GRANT OPTION |
+-------------------------------------------------------------+
1 row in set
```

可以用下面的任何语句来列出当前会话的权限：

SHOW GRANTS;

SHOW GRANTS FOR CURRENT_USER;

SHOW GRANTS FOR CURRENT_USER();

4.7.3.11 SHOW INDEX

SHOW INDEX FROM *table_name* [FROM *database_name*]

该语句列出选定数据库中指定表的索引。

示例 1：列出 test 数据库中指定表的索引。

```
gbase> USE test;
Query OK, 0 rows affected

gbase> CREATE TABLE t_index(a int, b varchar(10));
Query OK, 0 rows affected

gbase> CREATE INDEX idx_a ON t_index(a) USING HASH LOCAL;
Query OK, 0 rows affected
Records: 0  Duplicates: 0  Warnings: 0

gbase> SHOW INDEX FROM t_index FROM test;
+---------+------------+----------+--------------+-------------+
| Table   | Non_unique | Key_name | Seq_in_index | Column_name |
+---------+------------+----------+--------------+-------------+
| t_index |          1 | idx_a    |            1 | a           |
+---------+------------+----------+--------------+-------------+
+-----------+-------------+----------+--------+------+
| Collation | Cardinality | Sub_part | Packed | Null |
+-----------+-------------+----------+--------+------+
| NULL      |        NULL |     NULL | NULL   | YES  |
+-----------+-------------+----------+--------+------+
+------------+---------+
| Index_type | Comment |
+------------+---------+
| LOCAL HASH |         |
+------------+---------+
1 row in set
```

4.7.3.12 SHOW PROCEDURE STATUS

SHOW PROCEDURE STATUS

SHOW ROCEDURE STATUS 显示已经创建成功的存储过程的状态。

示例 1：显示已经创建成功的存储过程的状态。

```
gbase> SHOW PROCEDURE STATUS;
+-------+------------+---------+---------+---------------------+
| Db    | Name       | Type    | Definer | Modified            |
+-------+------------+---------+---------+---------------------+
| test  | proc_count | PROCEDURE | root@% | 2013-10-23 11:41:40 |
+-------+------------+---------+---------+---------------------+
+---------------------+---------------+-----------+
| Created             | Security_type | Comment   |
+---------------------+---------------+-----------+
| 2013-10-23 11:41:40 | DEFINER       |           |
+---------------------+---------------+-----------+
+--------------------+----------------------+------------------+
|character_set_client | collation_connection | Database Collation |
+--------------------+----------------------+------------------+
| utf8               | utf8_general_ci      | utf8_general_ci  |
+--------------------+----------------------+------------------+
1 row in set
```

4.7.3.13 SHOW PROCESSLIST

SHOW [FULL] PROCESSLIST

SHOW PROCESSLIST 显示正在运行的线程。如果有 SUPER 权限，可以看到所有线程。否则，用户只能看到自己的线程（就是与用户使用的 GBase 8a 帐号相关的线程）。如果不使用 FULL 关键字，只显示每个查询的前 100 个字符。

该语句报告 TCP/IP 连接的主机名,可以更容易看出每个客户端的运行状态，形式为 host_name:client_port。

如果用户得到“too many connections”错误消息，并且想要了解正在发

生的情况，本语句是非常有用的。GBase 8a 为拥有 SUPER 权限的帐号保留一个额外的连接，以确保管理员总是能够连接并检验系统（假设用户不给所有用户 SUPER 权限）。

SHOW PROCESSLIST 输出通常显示如下一些内容。

示例 1：显示正在运行的线程。

```
gbase> SHOW PROCESSLIST;
+----+------+---------------------+------+
| Id | User | Host                | db   |
+----+------+---------------------+------+
| 98 | root | localhost           | NULL |
+----+------+---------------------+------+
+---------+-------+-------+-----------------+
| Command | Time  | State | Info            |
+---------+-------+-------+-----------------+
| Query   |     0 | NULL  | SHOW PROCESSLIST |
+---------+-------+-------+-----------------+
1 row in set
```

4.7.3.14 SHOW STATUS

SHOW [GLOBAL | SESSION] STATUS [LIKE 'pattern']

SHOW STATUS 提供状态信息。

下面显示了部分输出。变量名称列表和变量的值可能和用户的服务器上的不同。

示例 1：查看状态信息。

```
gbase> SHOW STATUS;
+-----------------------------------+------------+
| Variable_name                     | Value      |
+-----------------------------------+------------+
| Aborted_clients                   | 0          |
```

```
| Aborted_connects                  | 17         |
| Audit_queries                     | 1          |
| Binlog_cache_disk_use             | 0          |
| Binlog_cache_use                  | 0          |
| Bytes_received                    | 120        |
| Bytes_sent                        | 3366       |
...
| Threads_running                   | 1          |
| Uptime                            | 31101      |
| Uptime_since_flush_status         | 31101      |
+-----------------------------------+------------+
261 rows in set
```

示例 2：带有 LIKE 子句的语句只显示匹配类型的变量。

```
gbase> SHOW STATUS LIKE 'Key%';
+----------------------+-------+
| Variable_name        | Value |
+----------------------+-------+
| Key_blocks_not_flushed | 0     |
| Key_blocks_unused    | 6686  |
| Key_blocks_used      | 8     |
| Key_read_requests    | 302   |
| Key_reads            | 8     |
| Key_write_requests   | 407   |
| Key_writes           | 205   |
+----------------------+-------+
7 rows in set
```

用 GLOBAL 选项可以得到所有连接到 GBase 8a 的状态值，而用 SESSION 选项可以得到当前连接的状态值。如果不用任何选项，缺省值为 SESSION。LOCAL 和 SESSION 意义相同。

注意一些状态变量只有全局值，这样无论使用 GLOBAL 还是 SESSION，都只能得到相同的值。

4.7.3.15 SHOW TABLES

SHOW [FULL | DISTRIBUTION] TABLES [FROM *database_name*] [LIKE 'pattern']

SHOW TABLES 列出一个给定数据库的非临时表。

示例 1: 列出 ssbm 数据库中以“s”开头的非临时表。

```
gbase> SHOW tables FROM ssbm like 's%';
+--------------------+
| Tables_in_ssbm (s%) |
+--------------------+
| supplier           |
+--------------------+
1 row in set
```

示例 2:支持 FULL 修饰语,SHOW FULL TABLES 输出第二个输出列 Table_type。这第二个输出列对表而言是 BASE TABLE, 对视图而言是 VIEW。

```
gbase> SHOW FULL TABLES FROM ssbm LIKE 's%';
+--------------------+------------+
| Tables_in_ssbm (s%) | Table_type |
+--------------------+------------+
| supplier           | BASE TABLE |
+--------------------+------------+
1 rows in set
```

注意: 若用户不拥有对表的权限, 在使用 SHOW TABLES 时不会输出该表。

4.7.3.16 SHOW TABLE STATUS

SHOW TABLE STATUS [FROM *database_name*] [WHERE name = '*talbe_name*']

SHOW TABLE STATUS 列出所有表或者指定数据库中表的当前状态的信息。

示例 1：查看指定数据库指定表的状态信息，由于实际的显示结果很长，因此将显示结果分为几个部分显示。

```
gbase> SHOW TABLE STATUS FROM ssbm WHERE name = 'part';
+------+---------+---------+------------+------+----------------+
| Name | Engine  | Version | Row_format | Rows | Avg_row_length |
+------+---------+---------+------------+------+----------------+
| part | EXPRESS |      10 | Compressed |    0 |              0 |
+------+---------+---------+------------+------+----------------+
+-------------+-----------------+-------------+----------+-------------+
| Data_length | Max_data_length | Index_length | Data_free |Auto_increment|
+-------------+-----------------+-------------+----------+-------------+
|           0 |                0 |            0 |         0 |         NULL |
+-------------+-----------------+-------------+----------+--------------+
+---------------------+---------------------+-----------+--------------+
| Create_time         | Update_time         | Check_time | Collation    |
+---------------------+---------------------+-----------+--------------+
| 2013-10-22 17:13:59 | 2013-10-22 17:13:59 | NULL       | utf8_general_ci |
+---------------------+---------------------+-----------+--------------+
+----------+-----------------+-----------------+--------------+
| Checksum | Create_options  | Limit_storage_size | storage_size |
+----------+-----------------+-----------------+--------------+
|     NULL | avg_row_length=306 |               0 |           0 |
+----------+-----------------+-----------------+--------------+
+----------------+---------------------------------+
| table_data_size | Comment                         |
+----------------+---------------------------------+
|               0 | Overall compression ratio: 0.000 |
+----------------+---------------------------------+
+--------------------------+---------------------------+
| local_hash_index_file_size | global_hash_index_file_size |
+--------------------------+---------------------------+
|                        0 |                         0 |
+--------------------------+---------------------------+
5 rows in set
```

4.7.3.17 SHOW VARIABLES

SHOW [GLOBAL | SESSION] VARIABLES [LIKE 'pattern']

SHOW VARIABLES 显示一些 GBase 8a 系统变量的值。

用 GLOBAL 选项，能得到连接到 GBase 8a 的新连接的变量值。用 SESSION 选项，能得到当前连接的变量值。如果不使用选项，缺省值为 SESSION。LOCAL 和 SESSION 意义相同。

下面是部分输出结果，变量和它们的值可能和用户的服务器不同。

示例 1：查看变量及其对应值。

```
gbase> SHOW VARIABLES;
+-------------------------------+-----------------------------------------+
| Variable_name                 | Value                                   |
+-------------------------------+-----------------------------------------+
| _gbase_caching_level          | 1                                       |
| _gbase_dc_block_size_limit    | 268435456                               |
| _gbase_dc_window_size         | 16                                      |
| _gbase_debug_internal         | 0                                       |
| _gbase_delete_filter_compressed| OFF                                     |
...
| version                | 8.5.1.2-37219                                          |
| version_comment        | GBase Enterprise Server - Advanced Edition (Commercial)|
| version_compile_machine| x86_64                                                 |
| version_compile_os     | unknown-linux-gnu                                      |
| wait_timeout           | 28800                                                  |
| warning_count          | 0                                                      |
+------------------------+--------------------------------------------------------+
319 rows in set
```

示例 2：使用一个 LIKE 子句，该语句仅显示匹配类型的变量。

```
gbase> SHOW VARIABLES LIKE '%buffer%';
+------------------------+-----------+
| Variable_name          | Value     |
```

```
+-------------------------+-----------+
| bulk_insert_buffer_size | 8388608   |
| gbase_buffer_distgrby   | 33554432  |
| gbase_buffer_hgrby      | 33554432  |
| gbase_buffer_hj         | 33554432  |
| gbase_buffer_insert     | 268435456 |
| gbase_buffer_result     | 16777216  |
| gbase_buffer_rowset     | 16777216  |
| gbase_buffer_sj         | 33554432  |
| gbase_buffer_sort       | 33554432  |
| gssys_sort_buffer_size  | 8388608   |
| join_buffer_size        | 131072    |
| key_buffer_size         | 8384512   |
| net_buffer_length       | 16384     |
| preload_buffer_size     | 32768     |
| read_buffer_size        | 131072    |
| read_rnd_buffer_size    | 262144    |
| sort_buffer_size        | 2097144   |
| sql_buffer_result       | OFF       |
+-------------------------+-----------+
18 rows in set
```

4.7.3.18 SHOW WARNINGS

```
SHOW WARNINGS [LIMIT [offset,] row_count]

SHOW COUNT(*) WARNINGS
```

SHOW WARNINGS 显示由最后一个语句产生的错误，警告和注意信息。当最后一个使用表的语句没有产生消息时，SHOW WARNINGS 不显示任何消息。相关的 SHOW ERRORS 语句只显示错误信息。参考 SHOW ERRORS 语法。

每个使用了表的新语句重置消息列表。

SHOW COUNT(*) WARNINGS 语句显示错误，警告和注意信息的数量，从变量 warning_count 也可以得到相同的值：

SHOW COUNT(*) WARNINGS;

SELECT @@warning_count;

warning_count 的值可能比用 SHOW WARNINGS 显示的值大，如果 max_error_count 系统变量被设定过小，以至于无法存储全部信息。

LIMIT 子句同 SELECT 语句中使用方法相同。

GBase 8a 服务器发回最后一个语句产生的错误，警告和注意信息的数目。

示例 1：查看警告信息。

```
gbase> DROP TABLE IF EXISTS no_such_table;
Query OK, 0 rows affected, 1 warning

gbase> SHOW WARNINGS;
+-------+------+-------------------------------+
| Level | Code | Message                       |
+-------+------+-------------------------------+
| Note  | 1051 | Unknown table 'no_such_table' |
+-------+------+-------------------------------+
1 row in set
```

示例 2：max_error_count 系统变量控制能存储的错误，警告和注意信息的最大数目，默认值为 64。用户可以改变该变量的值来改变可存储的信息数目。在下例中，ALTER TABLE 语句产生了三个警告信息，但是因为 max_error_count 值为 1，所以只存储了一个警告。

```
gbase> SHOW VARIABLES LIKE 'max_error_count';
+-----------------+-------+
| Variable_name   | Value |
+-----------------+-------+
| max_error_count | 64    |
+-----------------+-------+
1 row in set

gbase> SET max_error_count=1;
```

```
Query OK, 0 rows affected

gbase> SELECT 1 > '6x' FROM t;
+----------+
| 1 > '6x' |
+----------+
|        0 |
+----------+
1 row in set, 1 warning

gbase> SELECT @@warning_count;
+-----------------+
| @@warning_count |
+-----------------+
|               1 |
+-----------------+
1 row in set

gbase> SHOW WARNINGS;
+---------+------+---------------------------------------+
| Level   | Code | Message                               |
+---------+------+---------------------------------------+
| Warning | 1292 | Truncated incorrect DOUBLE value: '6x' |
+---------+------+---------------------------------------+
1 row in set
```

把 max_error_count 设为 0，则不存储警告信息。在这种情况下，warning_count 仍然指出已发生的警告数目，但是不存储任何警告内容。

5 存储过程、函数

GBase 8a 支持存储过程、自定义函数的定义和使用。

5.1 概述

存储过程是一组可以完成特定功能的 SQL 语句集，经编译后存储在数据库中。用户在执行存储过程时，需要指定存储过程的名称并给出参数（如果存储过程里包含参数）。

在如下情况中，存储过程非常有用：

- 当多个客户端应用程序是由不同的语言编写，或者运行在不同的平台，但需要执行同样的数据库操作。
- 当安全非常重要时。例如，银行对所有常用的操作都使用存储过程。这提供了一个一致的和安全的环境，并且存储过程可以保证每一个操作都正确的写入日志。如此设置，应用程序和用户将不能直接访问数据表，只能执行特定的存储过程。
- 存储过程可以提高性能，这是因为只需要在服务器和客户端之间传递更少的信息。负面影响是增加了数据库服务器的负担，因为在服务器端执行更多的任务而在客户端（应用程序）则只需执行较少的任务。这在一个或很少的数据库服务器连接有大量客户端（比如 Web 服务器）的情况下则更明显。
- 存储过程允许用户在数据库服务器中使用函数库。这正是现代应用程序语言具有的特性，例如，通过使用类来进行程序设计。这些客户端应用程序语言特性不论是否应用于数据库端的设计，对程序员来说采用这样的方法还是很有益处的。

GBase 8a 存储过程遵循 SQL：2003 标准。

GBase 8a 存储过程仍在不断地完善中。本章中所描述的所有语法都被有效

地支持，其局限性和扩展要求将被记录备案。

关于存储过程的异常处理方法请参见《GBase 8a 存储过程异常处理参考手册》。

5.2 创建存储过程、函数

存储过程和函数是由 CREATE PROCEDURE 和 CREATE FUNCTION 语句所创建的程序。存储过程通过 CALL 语句来调用，而且只能通过输出变量得到返回值。函数可以像其它函数一样从语句内部来调用（通过调用函数名），并返回一个标量值。存储程序（过程和函数）也可以调用其它存储程序（过程和函数）。

每个存储过程或函数都与一个特定的数据库相联系：

当存储程序（过程和函数）被调用时，隐含的 USE *database_name* 被执行（当存储程序（过程和函数）结束时完成），不允许在存储程序（过程和函数）中使用 USE 语句。

用户能使用数据库名来限定存储程序（过程和函数）名。这可以用来指明不在当前数据库中的存储程序（过程和函数）。例如，要调用一个与 gbase 数据库相关联的存储过程 p 或函数 f，用户可以使用 CALL gbase.p()或 gbase.f()。

当一个数据库被删除了，所有与它相关的存储程序（过程和函数）也都被删除了。

GBase 8a 允许在存储过程中使用标准的 SELECT 语句（即，不使用游标或局部变量）。这样，一个查询的结果简单直接地传送到客户端。多个 SELECT 语句产生多个结果集，所以客户端必须使用一个支持多结果集的 GBase 8a 客户端库。

要创建一个存储程序(过程和函数)，必须具有 CREATE ROUTINE 权限，ALTER ROUTINE 和 EXECUTE 权限自动的授予给它的创建者。如果开启更新日志，用户可能需要 SUPER 权限。

在默认情况下，存储程序（过程和函数）与当前的数据库相关联。要显式的将过程与数据库联系起来，那用户创建存储程序（过程和函数）时需要将它

的名字的格式写为 *database_name.sp_name*。

如果存储程序（过程和函数）的名字和内建 SQL 函数同名，当定义存储程序（过程和函数）时，用户需要在存储程序（过程和函数）名和后面的括号之间使用空格，否则会发生语法错误。同样在用户调用存储程序（过程和函数）时也要这样做。正是因为这个原因，我们建议避免使用存在的 SQL 函数名作为用户的存储程序（过程和函数）名。

在括号中必须要有参数列表。如果没有参数，应使用空的参数列表()。默认参数为 IN 参数。如果要将一个参数指定为其它类型，则请在参数名前使用关键字。

使用 RETURNS 子句（只有 FUNCTION 才能指定 RETURNS 子句）指明函数的返回类型时，函数体中必须包含一个 RETURN 语句。

如果一个存储过程或函数对同样的输入参数得到同样的结果，则被认为它是“确定的”(DETERMINISTIC)，否则就是“非确定”的(NOT DETERMINISTIC)。如果没有指明是 DETERMINISTIC 还是 NOT DETERMINISTIC，缺省是 NOT DETERMINISTIC。

当前 DETERMINISTIC 特性是可接受的，但并不被优化器所使用。然而，如果更新日志被激活，这个特性将影响到 GBase 8a 是否接受过程的定义。

几个特征参数提供了程序的数据使用信息。

- CONTAINS SQL 说明存储程序（过程和函数）不含有读取或写入数据的语句。
- NO SQL 说明过程不包含 SQL 语句。
- READS SQL DATA 表明程序包含读取数据的语句，但没有写入数据的语句。
- MODIFIES SQL DATA 指出程序包含可能写入数据的语句。如果没有这些参数被明确的指出，默认的是 CONTAINS SQL。
- SQL SECURITY 参数用来指明，此程序的执行权限是赋予创建者还是调

用者。缺省的值是 DEFINER。创建者和调用者必须要有对与程序相关的数据库的访问权。要执行存储程序（过程和函数）必须具有 EXECUTE 权限，必须具有这个权限的用户要么是定义者，要么是调用者，这依赖于如何设置 SQL SECURITY 特征。

COMMENT 语句是 GBase 8a 的扩展，可以用来描述存储过程。可以使用 SHOW CREATE PROCEDURE 和 SHOW CREATE FUNCTION 语句来显示这些信息。

GBase 8a 允许存储程序(过程和函数)包含 DDL 语句(比如 CREATE 和 DROP)和 SQL 事务语句（比如 COMMIT）。这不是标准所需要的，只是特定的实现。

返回结果集的语句不能用在函数中。这些语句包括不使用 INTO 将列值赋给变量的 SELECT 语句，SHOW 语句和比如像 EXPLAIN 这样的语句。对于在函数定义时就返回结果集的语句返回一个 Not allowed to return a result set from a function 错误（ER_SP_NO_RETSET_IN_FUNC）。对于在函数运行时才返回结果集的语句，返回 PROCEDURE%s can't return a result set in the given context 错误（ER_SP_BADSELECT）。

存储过程语法格式：

CREATE PROCEDURE <*proc_name*>([<*parameter_1*>[,...] [,*parameter_n*]])

[characteristic ...]

BEGIN

〈过程定义〉

END

函数语法格式：

CREATE FUNCTION <*func_name*>([<*parameter_1*>[,…] [,*parameter_n*]])

RETURNS type

BEGIN

〈函数定义〉

END

参数说明如下：

<*proc_name*>、<*func_name*>要创建的存储过程的名称。在同一数据库内，存储过程的名称必须唯一。存储过程名称只允许 a～z、A～Z、0～9、下划线，且不能只包含数字。

([<*parameter_1*>[,…] [,*parameter_n*]])定义存储过程的参数，每一个参数的定义格式是<参数方向><参数名称><参数数据类型>，其中过程<参数方向>确定参数是输入、输出还是输入输出，只能取 IN、OUT、INOUT 中的一个。

<参数方向><参数名称><参数数据类型>

- 存储过程的<参数方向>确定参数是输入、输出还是输入输出，只能取 IN、OUT、INOUT 中的一个。函数的<参数方向>只能是输入 IN。
- <参数名称>在同一个存储过程中必须唯一，只允许 a～z、A～Z、0～9、下划线，且不能只包含数字。
- <参数数据类型>指定参数的数据类型。

<过程定义>、<函数定义>是一系列的 SQL 语句的组合，其中包含一些数据操作以完成一定的功能逻辑。

定义存储过程时，存储过程名后面的括号是必需的，即使没有任何参数，也不能省略。

如果存储过程、函数中的<过程定义>仅包含一条 SQL 语句，则可以省略 BEGIN 和 END，否则，在定义存储过程时，必须使用 BEGIN...END 结构把相关的 SQL 语句组织在一起形成<过程定义>。

存储过程、函数可以嵌套，也可以递归调用。

type，是 GBase 8a 支持的数据类型。

下面是一个使用 IN，OUT 参数的简单的存储过程的例子。这个例子在存储过程定义前，使用 gbase 客户端定界符命令来改变语句定界符从“;”到“//”。这允许在存储体中，定界符传递给服务器而不被 gbase 解释。

示例 1：创建 proce_count 存储过程，并调用。

```
gbase> DELIMITER //
gbase> CREATE PROCEDURE proc_count (OUT param1 INT,IN param2  VARCHAR(10))
    BEGIN
    SELECT COUNT(*) INTO param1 FROM ssbm.customer WHERE c_nation= param2;
    END //
Query OK, 0 rows affected

gbase> CALL proc_count(@count1,'JORDAN')//
Query OK, 0 rows affected

gbase> DELIMITER ;
gbase> SELECT @count1;
+---------+
| @count1 |
+---------+
|    1182 |
+---------+
1 row in set
```

当使用定界符命令时，用户应该避免使用反斜杆（在 GBase 8a 中表示转义字符。

示例 2：创建含有参数的 hello 函数，使用 SQL 函数执行操作并返回结果。

```
gbase> DELIMITER //
gbase> CREATE FUNCTION hello (s CHAR(20)) RETURNS CHAR(50)
    RETURN CONCAT('Hello, ',s,'!');//
Query OK, 0 rows affected

gbase> DELIMITER ;
gbase> SET @result = hello('world');
Query OK, 0 rows affected
gbase> SELECT @result;
+-------------------------------------------------+
| @result                                         |
+-------------------------------------------------+
| Hello, world              !                     |
+-------------------------------------------------+
```

```
1 row in set
```

如果一个函数的 RETURN 语句返回的值与函数中 RETURNS 子句指明的值类型不同，返回的值强制为正确的类型。

示例 3：创建 fn_count 函数，过程定义中包含 SQL 语句。

```
gbase> DELIMITER //
gbase> CREATE FUNCTION fn_count (param  varchar(10)) RETURNS INT
       BEGIN
       SELECT COUNT(*)/5 INTO @count FROM ssbm.customer WHERE c_nation= param;
       RETURN @count;
       END//
Query OK, 0 rows affected

gbase> DELIMITER ;

gbase> SET @result = fn_count('JORDAN');
Query OK, 0 rows affected

gbase> SELECT @result;
+---------+
| @result |
+---------+
|     236 |
+---------+
1 row in set
```

5.3 修改存储过程、函数

语法格式：

```
ALTER {PROCEDURE | FUNCTION} <sp_name> [characteristic ...]

characteristic:

    { CONTAINS SQL | NO SQL | READS SQL DATA | MODIFIES SQL DATA }
```

| SQL SECURITY { DEFINER | INVOKER }

| COMMENT 'string'

示例 1：修改存储过程的注释信息。

```
gbase> ALTER PROCEDURE proc_count COMMENT 'new comment';
Query OK, 0 rows affected
```

示例 2：修改函数的注释信息。

```
gbase> ALTER FUNCTION fn_count COMMENT 'new comment';
Query OK, 0 rows affected
```

5.4 删除存储过程、函数

语法格式：

DROP {PROCEDURE | FUNCTION} [IF EXISTS] <*sp_name*>

参数说明如下：

<*sp_name*>要删除的存储过程（函数）的名称。

示例 1：删除存储过程。

```
gbase> DROP PROCEDURE IF EXISTS proc_count;
Query OK, 0 rows affected
```

示例 2：删除函数。

```
gbase> DROP FUNCTION IF EXISTS fn_count;
Query OK, 0 rows affected
```

5.5 调用存储过程、函数

GBase 8a 使用 CALL 语句调用存储过程。

语法格式:

CALL [database_name.]proc_name([<参数列表>])

说明:

调用存储过程时，如果存储过程有参数，则必须按照存储过程的定义中的顺序和类型为参数赋值，同时，对于 OUT 和 INOUT 参数，必须指明 OUT 和 INOUT 关键字。

即使存储过程没有任何参数，在调用时也必须在<存储过程名称>后面加上括号。

GBase 8a 使用 SELECT 语句调用函数。

SET @变量名 = [*database_name*.]*func_name*([<参数列表>])

GBase 8a 使用 SELECT 语句查看调用函数的执行结果。

语法格式:

SELECT @变量名;

示例 1: 调用存储过程示例。

```
gbase> USE test;
Query OK, 0 rows affected

gbase> DELIMITER //
gbase> DROP PROCEDURE proc_count//
Query OK, 0 rows affected

gbase> CREATE PROCEDURE proc_count (OUT param1 INT,IN param2 varchar(10))
       BEGIN
       SELECT COUNT(*) INTO param1 FROM ssbm.customer WHERE c_nation= param2;
       END //
Query OK, 0 rows affected

gbase> CALL proc_count(@count1, 'JORDAN')//
Query OK, 0 rows affected
```

```
gbase> DELIMITER ;
gbase> SELECT @count1 ;
+---------+
| @count1 |
+---------+
|    1182 |
+---------+
1 row in set
```

示例 2：调用函数示例。

```
gbase> USE test;
Query OK, 0 rows affected

gbase> DELIMITER //
gbase> DROP FUNCTION hello //
Query OK, 0 rows affected

gbase> CREATE FUNCTION hello (s CHAR(20)) RETURNS VARCHAR(50)
    RETURN CONCAT('Hello, ',s,'!') //
Query OK, 0 rows affected

gbase> DELIMITER ;
gbase> SET @result = hello('world');
Query OK, 0 rows affected

gbase> SELECT @result;
+----------------------------+
| @result                    |
+----------------------------+
| Hello, world               ! |
+----------------------------+
1 row in set
```

5.6 查看存储过程、函数的状态

查看创建或修改后的存储过程或函数的状态，可以使用如下语句：

SHOW {PROCEDURE | FUNCTION} STATUS

SHOW CREATE {PROCEDURE | FUNCTION} <*sp_name*>

示例 1：查看 fn_count 函数的状态。

```
gbase> SHOW CREATE FUNCTION fn_count;
+----------+...-+---------------------------------------+-...-+
| Function | ... | Create Function                       | ... |
+----------+...-+---------------------------------------+-...-+
| fn_count | ... | CREATE ... FUNCTION "fn_count"...END | ... |
+----------+...-+---------------------------------------+-...-+
1 row in set
```

在显示信息中包含 6 列，每列的具体信息如下。

- Function: fn_count
- sql_mode:
  PIPES_AS_CONCAT,ANSI_QUOTES,IGNORE_SPACE,NO_AUTO_VALUE_ON_ZERO,STRICT_ALL_TABLES,NO_ZERO_IN_DATE,NO_ZERO_DATE,NO_AUTO_CREATE_USER,NO_ENGINE_SUBSTITUTION,PAD_CHAR_TO_FULL_LENGTH
- Create Function: CREATE DEFINER="root"@"%" FUNCTION "fn_count"(param varchar(10)) RETURNS int

```
BEGIN
SELECT COUNT(*)/5 INTO @count FROM ssbm.customer WHERE c_nation= param;
RETURN @count;
END
```

- character_set_client: utf8
- collation_connection: utf8_general_ci
- Database Collation: utf8_general_ci

5.7 存储过程所支持的流程结构和语句

GBase 8a 在存储过程中除支持最基本的结构外，在<过程定义>部分还支持一些用于实现特定逻辑的流程控制结构和语句，这些结构和语句主要用于实现分支和循环。

5.7.1 DELIMITER

语法格式:

DELIMITER [Delimiter]

分隔符是通知客户端，已经完成输入一个 SQL 语句的字符或字符串符号，通常使用分号“;”，但在存储过程中，因为其中包含很多语句，每一个都需要一个分号，因此需要选择不太可能出现在语句中的符号作为分隔符，如“//”。

示例 1：使用//作为分隔符。

```
gbase> DELIMITER //
gbase> DROP PROCEDURE IF EXISTS dodeclare //
Query OK, 0 rows affected

gbase> CREATE PROCEDURE dodeclare (p1 INT)
     BEGIN
     DECLARE intX INT;
     SET intX = 0;
     REPEAT SET intX = intX + 1; UNTIL intX > p1 END REPEAT;
     SELECT intX;
     END //
Query OK, 0 rows affected

gbase> DELIMITER ;
gbase> CALL dodeclare(1000);
+------+
| intX |
+------+
| 1001 |
+------+
1 row in set
Query OK, 0 rows affected
```

5.7.2 BEGIN...END

语法格式:

```
[begin_label:] BEGIN
    [statement_list]
END [end_label]
```

存储程序（过程和函数）可能包含多个语句，这时就使用 BEGIN ... END 复合语句。

statement_list: 表示一个或多个语句的列表。多个语句之间使用分号“;”进行分隔。

复合语句可以被标记。end_label 只有在 begin_label 出现后才能使用，并且如果两者都出现，它们必须相同。

要使用多个语句，就需要客户端能发送包含语句分隔符“;”的查询字符串。这可在客户端通过 gbase 命令行使用分隔符更改命令来处理。更改查询结束的分隔符“;”（比如，改为//），允许“;”用在程序体中。

5.7.3 DECLARE

DECLARE 语句用来定义各种程序的局部项: 局部变量（参看存储过程中的变量），条件和处理器（参看条件和处理器）以及游标（参看游标）。目前不支持 SIGNAL 和 RESIGNAL 语句。

DECLARE 只能被用在 BEGIN ... END 复合语句之间，且必须位于其它语句之前。

游标必须在声明处理器变量之前被声明，并且条件必须在声明游标或处理器前声明。

语法格式一:

```
DECLARE var_name[,...] type [DEFAULT value]
```

这个语句用来声明局部变量。如果要对变量提供一个默认值，则包括一个

DEFAULT 语句。这个值可以指定为一个表达式，或一个常量。如果 DEFAULT 缺少子句，初始值为 NULL。

局部变量的作用范围在它被声明的 BEGIN ... END 块之间。变量可以在嵌套块中使用，除非在块中声明了同名的变量。

示例: DECLARE inX INT

```
gbase> DELIMITER //
gbase> DROP PROCEDURE IF EXISTS dodeclare //
Query OK, 0 rows affected

gbase> CREATE PROCEDURE dodeclare (p1 INT)
     BEGIN
     DECLARE intX INT;
     SET intX = 0;
     REPEAT SET intX = intX + 1; UNTIL intX > p1 END REPEAT;
     SELECT intX;
     END //
Query OK, 0 rows affected

gbase> DELIMITER ;
gbase> CALL dodeclare(1000);
+------+
| intX |
+------+
| 1001 |
+------+
1 row in set
Query OK, 0 rows affected
```

语法格式二:

DECLARE handler_type HANDLER FOR condition_value[,...] sp_statement

handler_type:

CONTINUE | EXIT | UNDO

condition_value:

SQLSTATE [VALUE] sqlstate_value

| condition_name | SQLWARNING | NOT FOUND | SQLEXCEPTION

| gbase_error_code

这条语句指明了处理程序，它们每个可以作为一个或多个条件来对待。如果这些条件之一发生了，具体指明的语句就会被执行。

参数说明如下：

- 对于一个 CONTINUE 命令，在处理器语句执行结束后，当前的程序继续执行。而对于一个 EXIT 处理器，当前 BEGIN...END 复合语句的执行被终止。
- SQLWARNING 是对所有以 01 开始的 SQLSTATE 代码的速记。
- NOT FOUND 是对所有以 02 开始的 SQLSTATE 代码的速记。
- SQLEXCEPTION 是对所有的没有被 SQLWARNING 或 NOT FOUND 捕获的 SQLSTATE 代码的速记。

示例：DECLARE CONTINUE HANDLER FOR SQLSTATE '02000' SET done = 1

```
gbase> DELIMITER //
gbase> DROP PROCEDURE IF EXISTS curdemo //
Query OK, 0 rows affected

gbase> CREATE PROCEDURE curdemo()
    BEGIN
    DECLARE done INT DEFAULT 0;
    DECLARE cnt INT DEFAULT 0;
    DECLARE s_region CHAR(255);
    DECLARE stmp CHAR(255) DEFAULT '';
    DECLARE cur_region CURSOR FOR SELECT DISTINCT c_region FROM ssbm.customer
ORDER BY c_region LIMIT 1000;
    DECLARE CONTINUE HANDLER FOR SQLSTATE '02000' SET done = 1;
    DROP TABLE IF EXISTS products;
```

```
CREATE TABLE products(region CHAR(255),count INT);
OPEN cur_region;
REPEAT
FETCH cur_region INTO s_region;
    IF NOT done THEN
        IF  stmp='' THEN
            SET stmp=s_region;
            SET cnt=1;
        END IF;
        IF stmp!=s_region  THEN
            INSERT INTO products(region,count) VALUES(stmp,cnt);
            SET cnt=1;
            SET stmp=s_region;
        END IF;
        SET cnt=cnt+1;
    END IF;
UNTIL done END REPEAT;
CLOSE cur_region;
INSERT INTO products(region,count) VALUES(stmp,cnt);
END//
Query OK, 0 rows affected

gbase> DELIMITER ;
gbase> CALL curdemo;
Query OK, 1 row affected

gbase> SELECT region,count FROM products;
+-----------------------------------------------------+-------+
| region                                              | count |
+-----------------------------------------------------+-------+
| AFRICA                                              |     2 |
| AMERICA                                             |     2 |
| ASIA                                                |     2 |
| EUROPE                                              |     2 |
| MIDDLE EAST                                         |     2 |
+-----------------------------------------------------+-------+
5 rows in set
```

5.7.4 SET

语法格式:

SET *var_name* = expr [, *var_name* = expr] ...

存储过程中的 SET 语句是对一般 SET 语句的扩展。引用的变量可以是在一个存储过程或全局服务器变量中声明的。

存储过程中的 SET 语句只是已存在的 SET 语法的部分实现。这允许扩展语法 SET a=x, b=y, ...这里可以混用不同的变量类型（局部变量与全局和会话服务器变量）。这也允许局部变量和一些对系统变量有意义的选项结合起来；在这种情况下，选项虽被认出但被忽略掉。

示例 1: SET intX = 0

```
gbase> DELIMITER //
gbase> DROP PROCEDURE IF EXISTS dodeclare //
Query OK, 0 rows affected

gbase> CREATE PROCEDURE dodeclare (p1 INT)
     BEGIN
     DECLARE intX INT;
     SET intX = 0;
     REPEAT SET intX = intX + 1; UNTIL intX > p1 END REPEAT;
     SELECT intX;
     END //
Query OK, 0 rows affected

gbase> DELIMITER ;
gbase> CALL dodeclare(1000);
+------+
| intX |
+------+
| 1001 |
```

```
+------+
1 row in set
Query OK, 0 rows affected
```

5.7.5 SELECT ...INTO...

语法格式:

SELECT *col_name*[,...] INTO *var_name*[,...] table_expr

功能:

将选出的列存储到变量中。只有单一行的结果才可以被取回。

示例 1: SELECT intX INTO @intResult;

```
gbase> DELIMITER //
gbase> DROP PROCEDURE IF EXISTS doselect_into //
Query OK, 0 rows affected

gbase> CREATE PROCEDURE doselect_into (p1 INT)
    BEGIN
    DECLARE intX INT;
    SET intX = 0;
    REPEAT SET intX = intX + 1; UNTIL intX > p1 END REPEAT;
    SELECT intX INTO @intResult;
    SELECT @intResult;
    END //
Query OK, 0 rows affected

gbase> DELIMITER ;
gbase> CALL doselect_into (1000);
+------------+
| @intResult |
+------------+
|       1001 |
+------------+
```

```
1 row in set
```

5.7.6 IF

GBase 8a 的 IF 结构是一个简单的条件分支结构。

语法格式:

IF 〈判断条件〉 THEN

〈执行体〉

ELSE

〈执行体〉

END IF;

GBase 8a 的 IF 结构允许嵌套。

示例 1: IF...THEN...ELSE...END IF

```
gbase> DELIMITER //
gbase> DROP FUNCTION IF EXISTS fn_count//
Query OK, 0 rows affected

gbase> CREATE FUNCTION fn_count (param  VARCHAR(10)) RETURNS INT
       BEGIN
       SELECT COUNT(*)/3 INTO @count FROM ssbm.customer WHERE c_nation=
'JORDAN';
       IF @count<=3 THEN
       RETURN @count;
       ELSE
       RETURN @count/3;
       END IF;
       END//
Query OK, 0 rows affected

gbase> DELIMITER ;
```

```
gbase> SET @result = fn_count ('JORDAN');
Query OK, 0 rows affected

gbase> SELECT @result;
+---------+
| @result |
+---------+
|     131 |
+---------+
1 row in set
```

5.7.7 ITERATE

ITERATE 语句用于实现回到指定位置重复执行，该语句只能出现在 LOOP、REPEAT 和 WHILE 结构中，并且必须为该语句定义要回到的位置的标签，之后在使用该语句处指定该标签。

语法格式:

ITERATE<标签名>

ITERATE 语句通常被放在 IF 结构中以实现根据条件重复执行。

示例 1: ITERATE...

```
gbase> DELIMITER //
gbase> CREATE PROCEDURE doiterate(p1 INT)
       BEGIN
       label1: LOOP
       SET p1 = p1 + 1;
       IF p1 < 10 THEN ITERATE label1; END IF;
       LEAVE label1;
       END LOOP label1;
       SET @x = p1;
       END //
Query OK, 0 rows affected
```

```
gbase> DELIMITER ;
gbase> CALL doiterate(1);
Query OK, 0 rows affected

gbase> SELECT @x;
+------+
| @x   |
+------+
|   10 |
+------+
1 row in set
```

5.7.8 CASE

GBase 8a 使用 CASE 结构处理多路分支的情况，其语法格式有两种，分别描述如下：

CASE

WHEN <条件 1> THEN <执行语句 1>

WHEN <条件 2> THEN <执行语句 2>

...

ELSE <执行语句 X>

END CASE ;

当<条件 1>为真值时，<执行语句 1>；当<条件 2>为真值时，<执行语句 2>；如果没有匹配的结果值，那么返回 ELSE 后的<执行语句 X>。

CASE <判定条件>

WHEN <值 1> THEN <执行语句 1>

WHEN <值 2> THEN <执行语句 2>

...

ELSE <执行语句 X>

END CASE ;

计算<判定条件>，如果<判定条件>的值等于<值 1>，<执行语句 1>；如果<判定条件>的值等于<值 2>，<执行语句 2>；如果没有相匹配的值则执行<执行语句 X>。

示例 1：CASE 后无判定条件。

```
gbase> DELIMITER //
gbase> DROP PROCEDURE IF EXISTS casedemo//
Query OK, 0 rows affected

gbase> CREATE PROCEDURE casedemo()
    BEGIN
    SELECT DISTINCT CASE WHEN c_nation='CHINA' THEN '中国' WHEN
c_nation='MOROCCO' THEN '摩洛哥' WHEN c_nation='JORDAN' THEN '约旦' ELSE '
其它国家' END 中文, c_nation FROM ssbm.customer  LIMIT 10;
    END //
Query OK, 0 rows affected

gbase> DELIMITER ;
gbase> CALL casedemo();
+------------------+
| 中文, c_nation    |
+------------------+
| 中国             |
| 摩洛哥           |
| 约旦             |
| 其它国家         |
+------------------+
4 rows in set
```

示例 2：CASE 后有判定条件。

```
gbase> DELIMITER //
gbase> DROP PROCEDURE IF EXISTS casedemo2//
```

```
Query OK, 0 rows affected

gbase> CREATE PROCEDURE casedemo2()
    BEGIN
    SELECT DISTINCT CASE  c_nation WHEN 'CHINA' THEN '中国' WHEN 'MOROCCO'
THEN '摩洛哥' WHEN 'JORDAN' THEN '约旦' ELSE '其它国家' END 中文, c_nation FROM
ssbm.customer  LIMIT 10;
    END //
Query OK, 0 rows affected

gbase> DELIMITER ;
gbase> CALL casedemo2();
+--------------------+
| 中文, c_nation     |
+--------------------+
| 中国               |
| 摩洛哥             |
| 约旦               |
| 其它国家           |
+--------------------+
4 rows in set
```

注意 CASE 计算也依靠上下文。如果是字符串上下文，返回的结果作为一个字符串，如果是数值上下文，返回结果是小数，实数或整数。

5.7.9 LOOP

LOOP 结构是 GBase 8a 中的一个简单的循环结构，用于重复执行一个或者一组语句。这个循环结构在形式上是个死循环结构，因此在执行体中通常要包括一个条件判断语句和 LEAVE 语句用于退出循环。

语法格式:

LOOP

〈执行体〉

```
END LOOP ;
```

示例 1: LOOP...END LOOP

```
gbase> DELIMITER //
gbase> DROP PROCEDURE IF EXISTS doiterate//
Query OK, 0 rows affected

gbase> CREATE PROCEDURE doiterate(p1 INT)
       BEGIN
       label1: LOOP
       SET p1 = p1 + 1;
       IF p1 < 10 THEN ITERATE label1; END IF;
       LEAVE label1;
       END LOOP label1;
       SET @x = p1;
       END //
Query OK, 0 rows affected

gbase> DELIMITER ;
gbase> CALL doiterate(1);
Query OK, 0 rows affected

gbase> SELECT @x;
+------+
| @x   |
+------+
|   10 |
+------+
1 row in set
```

5.7.10REPEAT

REPEAT 结构是 GBase 8a 中比较常见的一种循环结构，该结构会重复执行执行体直到满足退出条件。

语法格式:

REPEAT

<执行体>

UNTIL<退出条件>

END REPEAT ;

REPEAT 结构的执行体至少会执行一次，如果不允许这样可以使用 WHILE 结构代替。

示例 1: REPEAT...UNTIL...END REPEAT

```
gbase> DELIMITER //
gbase> DROP PROCEDURE IF EXISTS dorepeat//
Query OK, 0 rows affected

gbase> CREATE PROCEDURE dorepeat(p1 INT)
    BEGIN
    SET @x = 0;
    REPEAT SET @x = @x + 1; UNTIL @x > p1 END REPEAT;
    END //
Query OK, 0 rows affected

gbase> DELIMITER ;
gbase> CALL dorepeat(1000);
Query OK, 0 rows affected

gbase> SELECT @x;
+------+
| @x   |
+------+
| 1001 |
+------+
1 row in set
```

5.7.11 WHILE

WHILE 是 GBase 8a 中另一种常见的循环结构，在满足执行条件时该结构会重复执行执行体。

语法格式:

WHILE〈执行条件〉DO

〈执行体〉

END WHILE ;

WHILE 结构在逻辑上与 REPEAT 一致，唯一不同的是 until 结构中的执行体至少会执行一次，而 WHILE 结构中的执行体则可能一次也不执行。

示例: WHILE...END WHILE

```
gbase> DELIMITER //
gbase> DROP PROCEDURE IF EXISTS doWhile//
Query OK, 0 rows affected

gbase> CREATE PROCEDURE doWhile(p1 INT)
    BEGIN
    SET @x = 0;
    WHILE  @x < p1 DO  SET @x = @x + 1; END WHILE;
    END//
Query OK, 0 rows affected

gbase> DELIMITER ;
gbase> CALL dowhile(1000);
Query OK, 0 rows affected

gbase> SELECT @x;
+------+
| @x   |
+------+
| 1000 |
```

```
+------+
1 row in set
```

5.7.12LEAVE

GBase 8a 中的 LEAVE 语句用于退出循环结构，因此该语句也只能出现在 LOOP、REPEAT 和 WHILE 结构中。同样的，在使用 LEAVE 语句时必须为包含该语句的循环结构定义标签，然后在使用该语句处指定该标签。

语法格式:

LEAVE〈标签名〉

LEAVE 语句通常被放在 IF 结构中以实现根据条件退出循环结构。

示例 1: LEAVE...

```
gbase> DELIMITER //
gbase> DROP PROCEDURE IF EXISTS doiterate //
Query OK, 0 rows affected

gbase> CREATE PROCEDURE doiterate(p1 INT)
       BEGIN
       label1: LOOP
       SET p1 = p1 + 1;
       IF p1 < 10 THEN ITERATE label1; END IF;
       LEAVE label1;
       END LOOP label1;
       SET @x = p1;
       END //
Query OK, 0 rows affected

gbase> DELIMITER ;
gbase> CALL doiterate(1);
Query OK, 0 rows affected

gbase> SELECT @x;
+------+
```

```
| @x   |
+------+
|   10 |
+------+
1 row in set
```

5.7.13 静态游标 (CURSOR)

由 SELECT 语句返回的结果集通常包括一系列的记录行，但经常有一些情况下，并不总是能够将整个结果集作为一个单元来有效地处理。这时就需要一种机制以便每次处理一行记录，数据库中的游标就提供了这种机制。由于游标的意义，大多数的数据库都支持游标。GBase 8a 数据库也支持游标，但游标的定义和使用有一定的限制。

本节中描述的游标为静态游标,即在 DECLARE 时必须指定 SELECT STATEMENT 语句的结果集进行绑定；在后续操作中只能对于该结果集进行只读、仅向前的操作。

对于动态游标的定义及使用，请参见“5.7.14 动态游标 (CURSOR) ”。

游标必须在声明处理器之前被声明，变量和条件必须在声明游标或处理器之前被声明。

示例 1：静态游标完整示例。

```
gbase> DELIMITER //
gbase> DROP PROCEDURE IF EXISTS curdemo //
Query OK, 0 rows affected

gbase> CREATE PROCEDURE curdemo()
    BEGIN
    DECLARE done INT DEFAULT 0;
    DECLARE cnt INT DEFAULT 0;
    DECLARE s_region CHAR(255);
    DECLARE stmp CHAR(255) DEFAULT '';
    DECLARE cur_region CURSOR FOR SELECT DISTINCT c_region FROM ssbm.customer
```

```
ORDER BY c_region LIMIT 1000;
    DECLARE CONTINUE HANDLER FOR SQLSTATE '02000' SET done = 1;
    DROP TABLE IF EXISTS products;
    CREATE TABLE products(region CHAR(255),count INT);
    OPEN cur_region;
    REPEAT
    FETCH cur_region INTO s_region;
        IF NOT done THEN
            IF  stmp='' THEN
                SET stmp=s_region;
                SET cnt=1;
            END IF;
            IF stmp!=s_region  THEN
                INSERT INTO products(region,count) VALUES(stmp,cnt);
                SET cnt=1;
                SET stmp=s_region;
            END IF;
            SET cnt=cnt+1;
        END IF;
    UNTIL done END REPEAT;
    CLOSE cur_region;
    INSERT INTO products(region,count) VALUES(stmp,cnt);
    END//
Query OK, 0 rows affected

gbase> DELIMITER ;
gbase> CALL curdemo;
Query OK, 1 row affected

gbase> SELECT region,count FROM products;
+-------------------------------------------------------------+-------+
| region                                                      | count |
+-------------------------------------------------------------+-------+
| AFRICA                                                      |     2 |
| AMERICA                                                     |     2 |
| ASIA                                                        |     2 |
| EUROPE                                                      |     2 |
```

```
| MIDDLE EAST                                                |     2 |
+------------------------------------------------------------+-------+
5 rows in set
```

5.7.13.1 游标的定义

GBase 8a 中通过 DECLARE 声明游标，标注的主体一定是一个 SELECT 语句。

可以在一个程序中定义多个游标，但是每个块中的游标只能有一个唯一的名字。

SELECT 语句不能包含有 INTO 子句。

语法格式:

DECLARE <游标名称> CURSOR FOR <SELECT 语句>

参数说明如下:

<游标名称>要创建的游标的名称，只允许 a～z、A～Z、0～9、_(下划线)，且不能只包含数字。

<SELECT 语句>游标的内容，可以是任何合法的 SELECT 语句。

示例:

```
DECLARE cur CURSOR FOR SELECT DISTINCT c_region,1 FROM ssbm.customer  ORDER BY c_region LIMIT 1000;
```

5.7.13.2 打开游标

和其它数据库中使用游标的方式一样，在使用 GBase 8a 的游标前也需要使用 OPEN 语句打开游标。

语法格式:

OPEN <游标名称>

示例 1：以下代码是包含在游标代码块中的。

```
DECLARE cur CURSOR FOR SELECT DISTINCT c_region,1 FROM ssbm.customer  ORDER
BY c_region LIMIT 1000;
OPEN cur;
```

5.7.13.3 从游标中取得数据

使用游标的目的是为了取得游标定义中的 SELECT 语句所返回的结果集中的字段的值，在 GBase 8a 中，这一取值的过程也是通过 FETCH 语句实现的。

语法格式：

FETCH〈游标名称〉 INTO〈局部变量〉

参数说明如下：

〈游标名称〉前面定义的游标的名称，需要从该游标中取得返回值

〈局部变量〉从游标中取得的值要保存在这些局部变量中，FETCH 语句中要求局部变量的数量与游标定义语句中 SELECT 语句中的选择列表中的字段数量相同，且数据类型也要对应相同或者可以进行自动转换。这些局部变量会在后续的语句中进行处理。

示例：以下代码是包含在游标代码块中的。

```
DECLARE s_region CHAR(16);
DECLARE region INT;
DECLARE cur CURSOR FOR SELECT DISTINCT c_region,1 FROM ssbm.customer  ORDER
BY c_region LIMIT 1000;
OPEN cur;
FETCH cur  INTO s_region, region;//
```

5.7.13.4 关闭游标

游标在使用完成后需要关闭，否则游标所占用的服务器的资源不会被释放。

如果没有明确的关闭，游标则在声明它的复合语句结束处被关闭。

语法格式：

CLOSE〈游标名称〉

示例：以下代码是包含在游标代码块中的。

```
DECLARE s_region CHAR(16);
DECLARE region  INT;
DECLARE cur CURSOR FOR SELECT DISTINCT c_region,1 FROM ssbm.customer ORDER BY
c_region LIMIT 1000;
OPEN cur;
FETCH cur INTO s_region, region;
CLOSE cur;//
```

5.7.13.5 静态游标使用注意事项

GBase 8a 中的游标是一种只读、仅向前的游标。游标中包含的数据是不能在使用时被更改的，并且游标中的数据只能按照从头至尾顺序读取。

GBase 8a 中的游标需要配合处理器（handler）来使用，游标需要在处理器的声明语句之前被声明，而且，任何游标内使用的变量都需要在游标的声明语句之前被定义。

在使用游标处理数据时，通常会使用 LOOP、REPEAT 或者 WHILE 结构，并在这些结构的执行体中使用 FETCH 语句来遍历游标中的数据。

在 GBase 8a 中，同一个存储过程中可声明多个游标，但有以下使用上的限制：

- 多个游标不能嵌套，也不能相互交叉，最好是使用完一个再使用另外一个。
- 在同一个存储过程中只能定义一个处理器，这一个处理器会作用于所有的游标，因此，如果两个游标在流程上并列执行时会变得不甚合理。
- 如果使用了 LOOP、REPEAT 或者 WHILE 结构来遍历游标取得数据并进行处理，同时如果在这些循环结构的结构体中调用了存储过程，则被调

用的存储过程中不应该再包含游标和用于遍历游标的 LOOP、REPEAT 或者 WHILE 结构，否则可能会出现一些不可预期的结果。

5.7.13.6 游标示例

```
gbase> DELIMITER //
gbase> DROP PROCEDURE IF EXISTS docursor //
Query OK, 0 rows affected

gbase> CREATE PROCEDURE docursor()
     BEGIN
     DECLARE s_region VARCHAR(40);
     DECLARE DONE INT DEFAULT(0);
     DECLARE cur CURSOR FOR SELECT DISTINCT c_region FROM
ssbm.customer ORDER BY c_region LIMIT 6;
     DECLARE CONTINUE HANDLER FOR SQLSTATE '02000' SET done = 1;
     OPEN cur;
     REPEAT
     FETCH cur  INTO s_region;
     IF NOT done THEN
     SELECT s_region;
     END IF;
     UNTIL DONE END REPEAT;
     CLOSE cur;
     END //
Query OK, 0 rows affected

gbase> DELIMITER ;
gbase> CALL docursor();
+----------+
| s_region |
+----------+
| AFRICA   |
+----------+
1 row in set
```

```
+----------+
| s_region |
+----------+
| AMERICA  |
+----------+
1 row in set

+----------+
| s_region |
+----------+
| ASIA     |
+----------+
1 row in set

+----------+
| s_region |
+----------+
| EUROPE   |
+----------+
1 row in set

+-------------+
| s_region    |
+-------------+
| MIDDLE EAST |
+-------------+
1 row in set
Query OK, 0 rows affected
```

5.7.14动态游标 (CURSOR)

GBase 8a 支持动态游标的定义和使用。

作为静态游标的加强，在 DECLARE 时使用 REF CURSOR 声明为动态游标后，允许在 OPEN 时可多次绑定不同 SELECT STATEMENT 语句的结果集。

5.7.14.1 游标的定义

语法格式：

DECLARE〈游标名称〉REF CURSOR

参数说明如下：

- DECLARE：使用 DECLARE 定义动态游标时，不允许指定任何 SELECT 语句。
- 〈游标名称〉：要创建的游标的名称，只允许 a～z、A～Z、0～9、下划线，且不能只包含数字。
- REF CURSOR：表示该游标为动态游标。

示例：

```
DECLARE cur REF CURSOR;
```

5.7.14.2 打开游标

和静态游标的使用方式一样，在使用动态游标前也需要使用 OPEN 语句打开游标。

语法格式：

OPEN〈游标名称〉FOR〈SELECT 语句〉

参数说明如下：

〈SELECT 语句〉：游标的内容，可以是任何合法的 SELECT 语句，也可以是动态 SQL 语句。

示例：以下代码是包含在游标代码块中的。

```
DECLARE cur REF CURSOR;
OPEN cur FOR SELECT DISTINCT c_region,1 FROM ssbm.customer ORDER BY c_region
LIMIT 1000;
```

5.7.14.3 从游标中取得数据

通过 FETCH 语句，可取得动态游标 OPEN 语句中的 SELECT 语句返回的结果集中的字段的值。

语法格式:

FETCH〈游标名称〉 INTO〈局部变量〉

参数说明如下:

〈游标名称〉通过 OPEN 打开的游标的名称。

〈局部变量〉从游标中取得的值要保存在这些局部变量中,FETCH 语句中要求局部变量的数量与动态游标 OPEN 语句中的 SELECT 语句中的选择列表中的字段数量相同，且数据类型也要对应相同或者可以进行自动转换。

示例: 以下代码是包含在游标代码块中的。

```
DECLARE s_region CHAR(16);
DECLARE region INT;
DECLARE cur REF CURSOR;
OPEN cur FOR SELECT DISTINCT c_region,1 FROM ssbm.customer  ORDER BY c_region
LIMIT 1000;
FETCH cur INTO s_region, region;//
```

5.7.14.4 关闭游标

游标在使用完成后需要关闭，否则游标所占用的服务器的资源不会被释放。

如果没有明确的关闭，游标则在声明它的复合语句结束处被关闭。

语法格式:

CLOSE〈游标名称〉

示例：以下代码是包含在游标代码块中的。

```
DECLARE s_region CHAR(16);
DECLARE region INT;
DECLARE cur REF CURSOR;
OPEN cur FOR SELECT DISTINCT c_region,1 FROM ssbm.customer  ORDER BY c_region
LIMIT 1000;
FETCH cur INTO s_region, region;//
CLOSE cur;//
```

5.7.14.5 动态游标中使用的动态SQL语法

5.7.14.5.1 预处理语句中使用的动态SQL语法

语法格式：

```
PREPARE stmt_name FROM open_cur_stmt

EXECUTE stmt_name [USING @var_name [, @var_name] ...]

{DEALLOCATE | DROP} PREPARE stmt_name
```

参数说明如下：

<stmt_name>：预备语句名称。

<open_cur_stmt>：打开动态游标的动态 SQL 语句。

通过预处理语句，可先将动态游标 OPEN 语句进行预处理，后续通过 EXECUTE 语句进行执行。

通过动态 SQL 语句，动态游标 OPEN 语句中的 SELECT 语句可以由文本字符串或者内容为文本字符串的用户变量表示。动态 SQL 中还可包含'?'占位符表示未定参数，在执行 EXECUTE 语句时，使用通过 USING 传入的变量值为各个未定参数赋值，生成真正的 SELECT 语句。

示例：

```
DECLARE cur REF CURSOR;
```

```
  SET v = 'OPEN cur FOR SELECT * FROM hunter.t1 WHERE hunter.t1.i = ? AND
hunter.t1.j = ?';
  SET @sql_str = v;
  SET @a = 1;
  SET @b = 2;
  PREPARE stmt FROM @sql_str;
  EXECUTE stmt USING @a, @b;
  FETCH cur INTO i, j;
```

5.7.14.5.2 OPEN语句中使用的动态SQL语法

通过动态 SQL,动态游标 OPEN 语句中的 SELECT 语句可以由文本字符串或者内容为文本字符串的用户变量表示。

语法格式:

OPEN <游标名称> FOR <select_stmt>

参数说明如下:

<select_stmt>: 打开动态游标的动态 SQL 语句。

示例: SELECT * FROM hunter.t1

```
DECLARE cur REF CURSOR;
  SET v = 'SELECT * FROM hunter.t1';
  SET @sql_str = v;
  OPEN cur FOR @sql_str;
  FETCH cur INTO i, j;
```

5.7.14.6 动态游标使用注意事项

动态游标使用时需注意如下事项:

(1) 动态游标在 DECLARE 时，不允许指定任何 SELECT 语句。

(2) 动态游标允许嵌套，如果在嵌套中有 OPEN 操作，必须在同一嵌套中

有 CLOSE 操作与 OPEN 操作成对出现，避免重复 OPEN 操作的错误。

（3）动态游标只能用在存储过程中。

5.7.14.7 游标示例

示例 1：查询语句是静态 SQL 语句。

```
gbase> DELIMITER //
gbase> DROP PROCEDURE IF EXISTS docursor //
Query OK, 0 rows affected

gbase> CREATE PROCEDURE docursor()
     BEGIN
     DECLARE s_region VARCHAR(40);
     DECLARE DONE INT DEFAULT(0);
     DECLARE cur REF CURSOR;
     DECLARE CONTINUE HANDLER FOR SQLSTATE '02000' SET done = 1;
     OPEN cur FOR SELECT DISTINCT c_region FROM ssbm.customer ORDER BY c_region
LIMIT 6;
     REPEAT
     FETCH cur  INTO s_region;
     IF NOT done THEN
     SELECT s_region;
     END IF;
     UNTIL DONE END REPEAT;
     CLOSE cur;
     END //
Query OK, 0 rows affected

gbase> DELIMITER ;
gbase> CALL docursor();
+----------+
| s_region |
+----------+
| AFRICA   |
+----------+
```

```
1 row in set

+----------+
| s_region |
+----------+
| AMERICA  |
+----------+
1 row in set

+----------+
| s_region |
+----------+
| ASIA     |
+----------+
1 row in set

+----------+
| s_region |
+----------+
| EUROPE   |
+----------+
1 row in set

+-------------+
| s_region    |
+-------------+
| MIDDLE EAST |
+-------------+
1 row in set

Query OK, 0 rows affected
```

示例 2：OPEN 语句中的 SELECT 语句包含由文本字符串或者内容为文本字符串的用户变量。

```
DROP TABLE t1;
CREATE TABLE t1 (i INT, j INT);
```

```
INSERT INTO t1 VALUES(1, 1);
INSERT INTO t1 VALUES (1, 1);
INSERT INTO t1 VALUES (2, 2);
INSERT INTO t1 VALUES (3, 3);
INSERT INTO t1 VALUES (4, 4);
SELECT * FROM t1;

gbase> DELIMITER //
gbase> CREATE PROCEDURE test.test_1()
    BEGIN
      DECLARE v VARCHAR(200);
      DECLARE i INT DEFAULT (0);
      DECLARE j INT DEFAULT (0);
      DECLARE cur REF CURSOR;
      SET v = 'SELECT * FROM test.t1';
      SET @sql_str = v;
      OPEN cur FOR @sql_str;
      FETCH cur INTO i, j;
      SELECT i, j;
      CLOSE cur;
    END //
Query OK, 0 rows affected

gbase> DELIMITER ;
gbase> CALL test.test_1();
+------+------+
| i    | j    |
+------+------+
|    1 |    1 |
+------+------+
1 row in set

Query OK, 0 rows affected
```

示例 3：动态游标中的预处理语句包含动态 SQL 语句。

```
gbase> DELIMITER //
```

```
gbase> CREATE PROCEDURE test.test_1()
     BEGIN
       DECLARE v VARCHAR(200);
       SET v = 'SELECT * FROM test.t1 WHERE i = ? AND j = ?';
       SET @sql_str = v;
       SET @a = 1;
       SET @b = 2;
       PREPARE stmt FROM @sql_str;
       EXECUTE stmt USING @a, @b;
      END //
Query OK, 0 rows affected

gbase> DELIMITER ;
gbase> CALL test.test_1();
Empty set

Query OK, 0 rows affected
```

5.8 存储程序（过程、函数）的限制

函数使用限制：

不支持 DML，DDL，创建临时表。

不支持函数里面涉及 SQL 查询语句，但没有禁止。

不支持下列方式给变量赋值。

例如：

```
DECLARE res INT DEFAULT 0;

SET res=(SELECT COUNT(*) FROM t);
```

或：SELECT COUNT(*) INTO @res FROM t;

附录

GBase 8a 分析型数据库保留字

本章列出了 GBase 8a SQL 的保留字。

A

ACCESSIBLE	ADD	ALL
ALTER	ANALYZE	AND
AS	ASC	ASENSITIVE

B

BEFORE	BETWEEN	BIGINT
BINARY	BLOB	BOOLEAN
BOTH	BY	

C

CALL	CASCADE	CASE
CHANGE	CHAR	CHARACTER
CHECK	CLUSTER	COLLATE
COLUMN	COMPRESS	CONDITION
CONNECT	CONSTRAINT	CONTINUE
CONVERT	CREATE	CROSS
CURRENT_DATE	CURRENT_DATETIME	CURRENT_ROW
CURRENT_TIME	CURRENT_TIMESTAMP	CURRENT_USER
CURSOR		

D

DATABASE	DATABASES	DATACOPYMAP
DATADIR	DATASTATE	DAY_HOUR
DAY_MICROSECOND	DAY_MINUTE	DAY_SECOND
DEC	DECIMAL	DECLARE
DEFAULT	DELAYED	DELETE
DESC	DESCRIBE	DETERMINISTIC
DISTINCT	DISTINCTROW	DISTRIBUTED
DIV	DOUBLE	DROP
DUAL		

E

EACH	ELSE	ELSEIF
ENCLOSED	ENTRY	ESCAPED
EXCHANGE	EXISTS	EXIT
EXPLAIN	EXT_BAD_FILE	EXT_DATA_FILE
EXT_ESCAPE_CHARACTER	EXT_FLD_DELIM	EXT_LOG_FILE
EXT_STRING_QUALIFIER	EXT_TAB_OPT	EXT_TRIM_RIGHT_SPACE

F

FALSE	FETCH	FIRST_ROWS
FLOAT	FLOAT4	FLOAT8
FOLLOWING	FOR	FORCE
FOREIGN	FROM	FULL
FULLTEXT		

G

GBASE_ERRNO	GCLOCAL	GCLUSTER
GCLUSTER_LOCAL	GCR	GET
GRANT	GROUP	GROUPED
GROUPING		

H

HAVING	HIGH_PRIORITY	HOUR_MICROSECOND
HOUR_MINUTE	HOUR_SECOND	

I

IF	IGNORE	IN
INDEX	INDEX_DATA_PATH	INFILE
INITNODEDATAMAP	INNER	INOUT
INSENSITIVE	INSERT	INT
INT1	INT2	INT3
INT4	INT8	INTEGER
INTERSECT	INTERVAL	INTO
IS	ITERATE	

J

JOIN		

K

KEEPS	KEY	KEYS
KILL		

L

LEADING	LEAVE	LEFT
LIKE	LIMIT	LIMIT_STORAGE_SIZE
LINEAR	LINES	LINK
LOAD	LOCALTIME	LOCALTIMESTAMP
LOCK	LONG	LONGBLOB
LONGTEXT	LOOP	LOW_PRIORITY

M

MASTER_SSL_VERIFY_SERVER_CERT	MATCH	MEDIUMBLOB
MEDIUMINT	MEDIUMTEXT	MIDDLEINT
MINUS	MINUTE_MICROSECOND	MINUTE_SECOND
MOD	MODIFIES	MOVE

N

NATURAL	NOCACHE	NOCOPIES
NODE	NOLOCK	NOT
NO_WRITE_TO_BINLOG	NULL	NUMERIC

O

ON	OPTIMIZE	OPTION	OPTIONALLY
OR	ORDER	ORDERED	OUT
OUTER	OUTFILE	OVER	

P

PARALLEL	PRECEDING	PRECISION
PRIMARY	PROCEDURE	PUBLIC
PURGE		

R

RANGE	RCMAN	READ
READS	READ_WRITE	REAL
REFERENCES	REFRESH	REFRESHNODEDATAMAP
REGEXP	RELEASE	REMOTE
RENAME	REPEAT	REPLACE
REPLICATED	REQUIRE	RESTRICT
RETURN	REVOKE	RIGHT
RLIKE		

S

SAFEGROUPS	SCHEMA	SCHEMAS
SCN_NUMBER	SECOND_MICROSECOND	SELECT
SELF	SENSITIVE	SEPARATOR
SET	SETS	SHOW
SHRINK	SMALLINT	SPACE
SPATIAL	SPECIFIC	SQL
SQLEXCEPTION	SQLSTATE	SQLWARNING
SQL_BIG_RESULT	SQL_CALC_FOUND_ROWS	SQL_SMALL_RESULT
SSL	STARTING	STRAIGHT_JOIN
SYSTEM		

T

TABLE	TABLEID	TABLE_FIELDS
TARGET	TERMINATED	THEN
TID	TINYBLOB	TINYINT
TINYTEXT	TO	TO_DATE
TRAILING	TRANSACTION_LOG	TRIGGER
TRUE		

U

UNBOUNDED	UNDO	UNION
UNIQUE	UNLOCK	UNSIGNED
UPDATE	URI	USAGE
USE	USE_HASH	USING
UTC_DATE	UTC_DATETIME	UTC_TIME
UTC_TIMESTAMP		

V

VALIDATION	VALUES	VARBINARY
VARCHAR	VARCHARACTER	VARYING

W

WHEN	WHERE	WHILE
WITH	WITHOUT	WRITE

X

XOR		

Y

YEAR_MONTH		

Z

ZEROFILL		

GBASE

南大通用数据技术股份有限公司
General Data Technology Co., Ltd.

微博二维码　微信二维码

技术支持热线：400-817-9696