GBASE

GBase 8c 插件参考手册

www.gbase.cn

GBase 8c插件参考手册，南大通用数据技术股份有限公司

版权声明

免责声明

通讯方式

南大通用数据技术股份有限公司

天津市高新区华苑产业园区工华道2号天百中心3层(300384)

电话：400-013-9696　　邮箱：info@gbase.cn

商标声明

目　录

1 通用

GBase 8c 内部支持众多插件，提供扩展功能。部分插件已预创建，其他插件可供用户手动创建、删除。语法如下：

创建插件：

```
CREATE EXTENSION exten_name;
```

升级插件：

```
ALTER EXTENSION exten_name update;
```

删除插件：

```
DROP EXTENSION exten_name;
```

2 Dolphin Extension

2.1 Dolphin 概述

GBase8c 提供 dolphin Extension Reference（版本为 dolphin-0.0）。dolphin Extension Reference 是 GBase8c 的 MySQL 兼容性数据库（dbcompatibility='B'）扩展，从关键字、数据类型、常量与宏、函数和操作符、表达式、类型转换、DDL/DML/DCL 语法、存储过程/自定义函数、系统视图等方面增强 MySQL 兼容性。

dolphin 插件继承内核原有 SQL 语法，本文档将主要介绍对于内核语法有新增、修改的内容，和内核保持一致的语法等将不再额外写出。

2.2 Dolphin 安装

插件自动安装加载，无须手动安装加载。

(1) 使用 OM 工具安装 GBase8c，详见《GBase8cV5_5.0.0_安装部署手册》。

(2) 创建 B 库并使用初始用户连接 B 库即可，Dophin 插件默认启用。

```
create database dbname with DBCOMPATIBILITY='B';
\q
$gsql -d dbname -p 15400 -U gbase
```

2.3 Dolphin 限制

- 不支持小型化版本。

- 不支持删除 dolphin 插件。
- dolphin 插件只能在 B 兼容性数据库下创建。
- dolphin 插件需要在 pg_catalog 等 schema 下创建数据类型、函数等，所以加载 dolphin 插件需要初始用户权限。GBase 8c 将在第一次通过初始用户或拥有初始用户权限的用户连接 B 数据库时自动加载 dolphin 插件。如果一个 B 兼容性数据库从来没有被初始用户或拥有初始用户权限的用户连接过，那么它也不会加载 dolphin 插件。
- dolphin 中所有新增/修改的语法不支持在 gsql 客户端通过\h 查看帮助说明，不支持在 gsql 客户端自动补齐。
- dolphin 插件的创建会删除数据库存在的插件所需的同名函数和类型以及之前存在的与之依赖的对象。
- dolphin 插件依赖于 public schema，因此不支持使用 drop schema 的方式删除 public schema。
- 连接安装有 dolphin 插件的 B 兼容性数据库时，会默认修改 GUC 参数 behavior_compat_options，增加 display_leading_zero 和 select_into_return_null 选项，以保持兼容性。

2.4 Dolphin 语法介绍

2.4.1 SQL 参考

2.4.1.1 关键字

SQL 里有保留字和非保留字之分。根据标准，保留字决不能用做其他标识符。非保留字只是在特定的环境里有特殊的含义，而在其他环境里是可以用做标识符的。

标识符的命名需要遵守如下规范：

标识符需要为字母、下划线、数字(0-9)或美元符号($)。

标识符必须以字母(a-z)或下划线(_)开头。

说明

- 此命名规范为建议项，非强制项。
- 特殊情况下可以使用双引号或者反引号(`)规避特殊字符报错。

相比于 GBase8c 原生语法，dolphin 对于关键字的修改主要为：

(1) 新增 MEDIUMINT，作为非保留关键字。

(2) 关键字 DATE 可以作为函数使用。

(3) 新增 LAST_DAY，作为保留关键字，用于在语法层面区别 GBase8c 原有 LAST_DAY 函数和 dolphin 中 LAST_DAY 函数。

(4) 新增 GET_FORMAT，作为非保留关键字，用于在语法上识别 GET_FORMAT 函数。

(5) 新增 DAY_HOUR，DAY_MINUTE，DAY_SECOND，DAY_MICROSECOND，HOUR_MINUTE，HOUR_SECOND，HOUR_MICROSECOND，MINUTE_SECOND，MINUTE_MICROSECOND，SECOND_MICROSECOND，作为非保留关键字，用于 EXTRACT 函数在语法上识别对应单位。

(6) 改变关键字 AUTHID 等级，由 RESERVED_KEYWORD 变为 COL_NAME_KEYWORD，使其可以作为表名列名使用。

(7) 改变关键字 BODY 等级，由 UNRESERVED_KEYWORD 变为 RESERVED_KEYWORD。

(8) 新增 DUAL，作为保留关键字。

表 1-1SQL 关键字

关键字	GBase8c	SQL:1999	SQL-92
FORMAT	非保留(不能是函数或类型)	-	-
IF	非保留(不能是函数或类型)	-	-
KEYS	非保留	-	-
MEDIUMINT	非保留(不能是函数或类型)	-	-
SIGNED	非保留(不能是函数或类型)	-	-
UNSIGNED	非保留(不能是函数或类型)	-	-
ZEROFILL	非保留	-	-

关键字	GBase8c	SQL:1999	SQL-92
DATE	非保留(不能是函数或类型)	-	-
LAST_DAY	保留	-	-
GET_FORMAT	非保留(不能是函数或类型)	-	-
DAY_HOUR	非保留	-	-
DAY_MINUTE	非保留	-	-
DAY_SECOND	非保留	-	-
DAY_MICROSECOND	非保留	-	-
HOUR_MINUTE	非保留	-	-
HOUR_SECOND	非保留	-	-
HOUR_MICROSECOND	非保留	-	-
MINUTE_SECOND	非保留	-	-
MINUTE_MICROSECOND	非保留	-	-
SECOND_MICROSECOND	非保留	-	-
AUTHID	非保留(不能是函数或类型)	-	-
BODY	保留	-	-
DUAL	保留	-	-

2.4.1.2 数据类型

2.4.1.2.1 数值类型

相比于 GBase8c 原生语法，dolphin 对于数值类型的修改主要为:

(1) 新增 INT/TINYINT/SMALLINT/BIGINT 支持可选的修饰符(n)，即支持 TINYINT(n)/SMALLINT(n)/BIGINT(n)的用法，n 无实际意义，不影响任何表现。

(2) 新增 MEDIUMINT(n)数据类型，是 INT4 的别名，n 无实际作用，不影响任何表现。存储空间为 4 字节，数据范围为-2,147,483,648~+2,147,483,647。

(3) 新增 FIXED[(p[,s])]数据类型，是 NUMERIC 类型的别名。用户声明精度。每四位(十进制位)占用两个字节，然后在整个数据上加上八个字节的额外开销。未指定精度的情况下，小数点前最大 131,072 位，小数点后最大 16,383 位。

(4) 新增 float4(p[,s])的方式，等价于 dec(p[,s])。

(5) 新增 double 数据类型，是 float8 的别名。

(6) 新增 float4/float 支持可选的修饰符(n)，即支持 float4(n)/float(n)的用法，当 n 在[1,24]之间时，float4(n)/float(n)代表单精度浮点数；当 n 在[25,53]之间时，float4(n)/float(n)代表双精度浮点数。

(7) 对于 decimal/dec/fixed/numeric 数据类型，在未指定精度的情况下，默认精度为(10,0)，即总位数为 10，小数位数为 0。

(8) 新增 UNSIGNEDINT/TINYINT/SMALLINT/BIGINT 类型，与普通整型相比，其最高位是数字位而非符号位；此外，在 GBase8c 中，TINYINT 默认为无符号类型，而在 B 库中则默认是有符号的。

(9) 新增 zerofill 属性修饰，只是语法上的支持，实际并没有填充零的效果。与 UNSIGNED 的作用等价。

(10) 新增 cast 函数类型转换参数 signed/unsigned，其中 castasunsigned 将类型转换为 uint8，castassigned 将类型转换为 int8.

(11) 新增 float(p,s), double(p,s), real(p,s), doubleprecision(p,s)的语法，其中 float(p,s), real(p,s), doubleprecision(p,s)大致等价于 dec(p,s)，与 dec(p,s)不同的是，float(p,s)，real(p,s)，doubleprecision(p,s)的 p 和 s 必须为整数，而 double(p,s)则完全等价于 dec(p,s)。舍入方式为四舍五入。

表 1-2 整数类型

名称	描述	存储空间	范围
TINYINT(n)	微整数，别名为 INT1。n 无实际作用，不影响任何表现。	1 字节	-128~+127
SMALLINT(n)	小范围整数，别名为 INT2。n 无实际作用，不影响任何表现。	2 字节	-32768~+32767
INTEGER(n)	常用的整数，别名为 INT4。n 无实际作用，不影响任何表现。	4 字节	-2147483648~+2147483647
MEDIUMINT(n)	INT4 的别名，n 无实际作用，不影响任何表现。	4 字节	-2147483648~+2147483647
BIGINT(n)	大范围的整数，别名为 INT8。n 无实际作用，不影响任何表现。	8 字节	-9223372036854775808~+9223372036854775807
TINYINT(n) UNSIGNED	无符号微整数，别名为 UINT1。n 无实际作用，不影响任何表现。	1 字节	0~255
SMALLINT(n) UNSIGNED	无符号小范围整数，别名为 UINT2。n 无实际作用，不影响任何表现。	2 字节	0~+65535

示例

(1) 创建具有 TINYINT(n),SMALLINT(n),MEDIUMINT(n),BIGINT(n)类型数据的表。

```
CREATE TABLE int_type_t1
(
IT_COL1 TINYINT(10),
IT_COL2 SMALLINT(20),
IT_COL3 MEDIUMINT(30),
IT_COL4 BIGINT(40),
IT_COL5 INTEGER(50)
);
```

(2) 查看表结构。

```
\d int_type_t1
```

返回结果为：

```
Table"public.int_type_t1"
Column|Type|Modifiers
--------+---------+----------
IT_COL1|tinyint|
IT_COL2|smallint|
IT_COL3|integer|
IT_COL4|bigint|
IT_COL5|integer|
```

(3) 创建带 zerofill 属性字段的表。

```
CREAT ETABLE int_type_t2
(
IT_COL1 TINYINT(10) zerofill,
IT_COL2 SMALLINT(20 )unsigned zerofill,
IT_COL3 MEDIUMINT(30) unsigned,
IT_COL4 BIGINT(40) zerofill,
IT_COL5 INTEGER(50) zerofill
);
```

(4) 查看表结构。

```
\dint_type_t2
```

返回结果为：

```
Table"public.int_type_t2"
Column|Type|Modifiers
--------+------+----------
IT_COL1|uint1|
IT_COL2|uint2|
IT_COL3|uint4|
IT_COL4|uint8|
IT_COL5|uint4|
```

(5) 利用 cast unsigned 将表达式转换为 uint8 类型。

```
select cast(1-2 as unsigned);
```

返回结果为：

```
uint8
--------------------
```

```
18446744073709551615
(1row)
```

(6) 利用 castsigned 将表达式转换为 int8 类型。

```
select cast(1-2 as signed);
```

返回结果为:

```
int8
------
-1
(1row)
```

表 1-3 任意精度型

名称	描述	存储空间	范围
NUMERIC[(p[,s])] DECIMAL[(p[,s])] FIXED[(p[,s])]	精度 p 取值范围为[1,1000]，标度 s 取值范围为[0,p]。说明：p 为总位数，s 为小数位数。	用户声明精度。每四位（十进制位）占用两个字节，然后在整个数据上加上八个字节的额外开销。	未指定精度的情况下，等价于(10,0)，即小数点前最大 10 位，小数点后 0 位。
NUMBER[(p[,s])]	NUMERIC 类型的别名。	用户声明精度。每四位（十进制位）占用两个字节，然后在整个数据上加上八个字节的额外开销。	未指定精度的情况下，小数点前最大 131,072 位，小数点后最大 16,383 位。

示例

(1) 创建具有 FIXED(p,s),FIXED,decimal,number 类型数据的表。

```
CREATE TABLE dec_type_t1
(
DEC_COL1 FIXED,
DEC_COL2 FIXED(20,5),
DEC_COL3 DECIMAL,
DEC_COL4 NUMBER
);
```

(2) 查看表结构。

```
\d dec_type_t1
```

返回结果为：

```
Table"public.dec_type_t1"
Column | Type | Modifiers
---------+--------------+-----------
DEC_COL1|numeric(10,0)|
DEC_COL2|numeric(20,5)|
DEC_COL3|numeric(10,0)|
DEC_COL4|numeric|
```

表 1-4 浮点类型

名称	描述	存储空间	范围
FLOAT[(p)] FLOAT4[(p)]	浮点数，不精准。精度 p 取值范围为[1,53]。	4 字节或 8 字节	当精度 p 在[1,24]之间时，选项 REAL 作为内部表示，当精度 p 在[25,53]之间时，选项 DOUBLEPRECISION 作为内部表示。如不指定精度，内部用 REAL 表示。
DOUBLEPRECISION FLOAT8 DOUBLE	双精度浮点数，不精准。	8 字节	-79E+308~79E+308，15 位十进制数字精度。
FLOAT4(p,s)	精度 p 取值范围为[1,1000]，标度 s 取值范围为[0,p]。 说明：p 为总位数，s 为小数位位数，等价于 dec(p,s)。	用户声明精度。每四位(十进制位）占用两个字节，然后在整个数据上加上八个字节的额外开销。	-
FLOAT(p,s) DOUBLE(p,s)	精度 p 取值范围为[1,1000]，标度 s 取值范围为[0,p]。 说明：p 为总位数，s	用户声明精度。每四位(十进制位）占用两个字节，然	-

REAL(p,s) DOUBLEPRECISION(p,s)	为小数位位数，其中 float(p,s)， real(p,s)，doubleprecision(p,s)大致等价于 dec(p,s)，但 p 和 s 都必须为整数，而 double(p,s)完全等价于 dec(p,s)。舍入方式为四舍五入。	后在整个数据上加上八个字节的额外开销。	

2.4.1.2.2 字符类型

相比于 GBase8c 原生语法，dolphin 对于字符类型的修改主要为：

(1) 修改 CHARACTER/NCHAR 类型 n 的含义，n 是指字符长度而不是字节长度。

(2) 所有的字符数据类型在对比时，均忽略尾部空格，如 where 条件过滤场景、join 场景等。例如'a'::text='a'::text 为真。需要特别注意的是，对于 VARCHAR、VARCHAR2、NVARCHAR2、NVARCHAR、TEXT、CLOB 类型，只有 GUC 参数 string_hash_compatible 为 on 的情况下，hashjoin 以及 hashagg 才会忽略尾部空格。

(3) 新增 NATIONALVARCHAR(n)，为 NVARCHAR2(n)类型的别名，n 是指字符长度。

(4) 新增 TEXT 支持可选的修饰符(n)，即支持 TEXT(n)的用法，n 无实际意义，不影响任何表现。

(5) 新增 TINYTEXT(n)/MEDIUMTEXT(n)/LONGTEXT(n)数据类型，是 TEXT 的别名，n 无实际作用，不影响任何表现。

表 1-5 字符类型

名称	描述	存储空间
CHAR(n) CHARACTER(n) NCHAR(n)	定长字符串，不足补空格。n 是指字符长度，如不带精度 n，默认精度为 1。	最大为 10MB。
NATIONAL VARCHAR(n)	变长字符串。是 NVARCHAR2(n)类型的别名。n 是指字符长度。	最大为 10MB。
TEXT(n), TINYTEXT(n),	变长字符串。n 无实际含义，不影响任何表现	最大为 1GB-1，但还需要考虑到列描述头信息的大小，以及列所在元组的大小限制(也小于 1GB-1)，因此

MEDIUMTEXT(n), LONGTEXT(n)		TEXT 类型最大大小可能小于1GB-1。

示例

(1) 创建测试表并插入数据。

```
CREATE TABLE char_type_t1
(
CT_COL1 CHARACTER(4),
CT_COL2 TEXT(10),
CT_COL3 TINYTEXT(11),
CT_COL4 MEDIUMTEXT(12),
CT_COL5 LONGTEXT(13)
);
```

(2) 查询看表结构。

```
\d char_type_t1
```

返回结果为:

```
Table"public.char_type_t1"
Column|Type|Modifiers
---------+-------------+-----------
CT_COL1|character(4)|
CT_COL2|text|
CT_COL3|text|
CT_COL4|text|
CT_COL5|text|
```

(3) 向表中插入数据。

```
INSERT INTO char_type_t1VALUES('四个字符');
```

(4) 查询数据。

```
SELECT CT_COL1,length(CT_COL1) FROM char_type_t1;
```

返回结果为:

```
CT_COL1|length
----------+--------
四个字符|12
(1row)
```

2.4.1.2.3 日期-时间类型

相比于 GBase8c 原生语法，dolphin 对于日期/时间类型的修改主要为：

(1) 修改 date/time/datetime/timestamp 类型的表现。

(2) 新增 year 数据类型。

注意：由于 GBase8c 固有的特性导致无法完全兼容 MySQL 时间数据类型的所有特性，因此需要根据本文档的要求进行特性的使用，避免使用文档描述外的特性，同时兼容后的特性已经覆盖了绝大部分场合的使用需求。

下列表格为兼容 MySQL 时间数据类型后的基本属性。

表 1-6 日期-时间类型

类型	描述	存储空间	取值范围(用户可输入范围)	精度范围	备注
date	日期	4 字节	4713BC~5874897 AD	-	(1)输入必须为有效日期，不支持月份或者天数为零值；(2)若年份大于等于 10000，必须以 'YYYY-MM-DD' 的形式进行输入；(3)若输入数据没有指定是 BC 还是 AD，则默认为 AD；
time(p)	可以用于表示一天中的时间或者一段时间(时分秒)，p 表示精度	8 字节	-838:59:59[.frac]~ 838:59:59[.frac]	p 表示小数点后的精度，取值范围为 0~6，如不指定默认为 0	-
datetime(p)	日期和时间，不带时区信息，p	8 字节	0AD~294276AD	p 表示小数点后的精度，取值范围为	(1)输入必须为有效日期，不支持月份或者天数为零

类型	描述	存储空间	取值范围(用户可输入范围)	精度范围	备注
	表示精度			0~6，如不指定默认为 0	值；(2)当输入的年份大于等于 10000 时，必须使用 'YYYY-MM-DD' 的格式进行输入；
timestamp(p)	日期和时间，带时区信息，p 表示精度	8 字节	0AD~294276AD	p 表示小数点后的精度，取值范围为 0~6，如不指定默认为 0	(1)输入必须为有效日期，不支持月份或者天数为零值；(2)注意，timestamp 类型在原来 GBase 8c 数据库中表示不带时区的时间戳，兼容后的 timestamp 类型往 MySQL 数据库靠拢，表示带时区信息的时间戳，因此存在兼容性问题；(3)当输入的年份大于等于 10000 时，必须使用 'YYYY-MM-DD' 的格式进行输入；
year(w)	年份，w 表示“displaywidth”，year(4)、year 形式输出为'YYYY'形式，year(2)形式输出为'YY'	2 字节	1901~2155	-	-

备注

对于 MySQL，在使用 CREATE TABLE 或者 ALTERTABLE 语句中，如果定义时间类型(例如 timestamp、datetime、time)列属性时不指定精度，则默认为 0。同时，使用 cast(exprastypename)语法进行类型转换时，如果目标类型没有指定精度，那么默认精度也为 0。因此，如果用户需要保留数据的输入精度，则需要显式使用 typmod。同时，兼容后的时间类型，使用::进行类型转换，如果目标类型没有指定精度，那么默认精度也为 0。

date 类型输入

支持如下格式：

格式	含义
'YYYY-MM-DD''YY-MM-DD'	年月日
'YYYYMMDD'、'YYMMDD'	年月日
YYYYMMDD、YYMMDD	年月日

备注

输入必须为有效日期，不支持月份或者日期为零值

由于 MySQL 原本的年份取值范围在 10000 以内，因此由于 MySQL 原本的年份取值范围在 10000 以内，因此如果想要输入大于等于 10000 年份的日期，请使用'YYYY-MM-DD'这种格式，例如'10100-12-12'允许输入 0000 年，同时在 GBase8c 中，认为 0000 年为闰年，可以输入 0000-2-29(MySQL 不允许)

示例

(1) 创建表，并插入 date 类型数据。

```
CREATE TABLE test_date(dt date);
INSERT INTO test_date VALUES('2020-12-21');
INSERT INTO test_date VALUES('141221');
INSERT INTO test_date VALUES(20151022);
```

(2) 查看数据。

```
SELECT * FROM test_date;
```

返回如下信息：

```
     dt
------------
 2020-12-21
```

```
 2014-12-21
 2015-10-22
(3 rows)
```

time 类型输入

支持如下格式：

格式	含义
'[-][D]hh:mm:ss[.frac]'	时分秒，前方可以指定为负，D 表示天数，取值范围为[0-34]
'[-]hhmmss[.frac]'	时分秒
[-]hhmmss[.frac]	时分秒

备注

- 对于格式'hh:mm:ss'，还支持宽松的类型'hh:mm'和'ss'的输入格式
- 当输入整数 0 时，代表的值为'00:00:00'，也是 time 类型的零值
- 由于 time 类型兼容后范围可大于 24 小时，并非仅能表示一天中的时间，请勿将 time 类型转型为 timetz 类型

示例

(1) 创建表并插入 time 类型数据。

```
CREATE TABLE test_time(ti time(2));
INSERT INTO test_time VALUES('29:12:24.1234');
INSERT INTO test_time VALUES('-34:56:59.1234');
INSERT INTO test_time VALUES(561234);
```

(2) 查看表数据

```
SELECT * FROM test_time;
      ti
--------------
 29:12:24.12
 -34:56:59.12
 56:12:34
(3 rows)
```

datetime 类型输入

支持如下格式：

格式	含义
'YYYY-MM-DDhh:mm:ss[.frac]'，'YY-MM-DDhh:mm:ss[.frac]'	时间戳
'YYYYMMDDhhmmss'，'YYMMDDhhmmss'	时间戳
YYYYMMDDhhmmss，YYMMDDhhmmss	时间戳

备注

- 输入必须为有效日期，不支持月份或者日期为零值
- 对于'YYYYMMDDhhmmss'和'YYMMDDhhmmss'格式，只有当字符串长度刚好为 8 或者 14 的时候，才会将字符串前 4 位字母识别为年的部分，其余都只会将前 2 位字母识别为年的部分
- 对于输入为 YYYYMMDDhhmmss 或 YYMMDDhhmmss 格式，输入的整数长度应该为 6/8/12/14 其中之一，如果长度不满足这个要求，则相当于往整数前方添加零，直到长度符合 6/8/12/14 其中之一(长度为 6 对应为 YYMMDD 格式，长度为 8 对应为 YYYYMMDD 格式，
- 长度为 12 对应为 YYMMDDhhmmss 格式，长度为 14 对应为 YYYYMMDDhhmmss 格式)类似兼容后的 date 类型，如果要想输入年份大于等于 10000 的时间戳，请使用'YYYY-
- MM-DDhh:mm:ss[.frac]'这种格式

示例

(1) 创建表并插入 datetime 类型数据。

```
CREATE TABLE test_datetime(dt datetime(2));
INSERT INTO test_datetime VALUES('2020-11-08 02:31:25.961');
INSERT INTO test_datetime VALUES(201112234512);
```

(2) 查看表数据。

```
SELECT * FROM test_datetime;
           dt
----------------------
 2020-11-08 02:31:25.96
 2020-11-12 23:45:12
(2 rows)
```

timestamp 类型输入

支持如下格式：

格式	含义
'YYYY-MM-DDhh:mm:ss[.frac][+/-hh:mm:ss]' 'YY-MM-DDhh:mm:ss[.frac][+/-hh:mm:ss]'	带时区信息时间戳
'YYYYMMDDhhmmss[.frac]'，'YYMMDDhhmmss[.frac]'	带时区信息时间戳
YYYYMMDDhhmmss[.frac]，YYMMDDhhmmss[.frac]	带时区信息时间戳

备注

- 输入必须为有效日期，不支持月份或者日期为零值
- 兼容的 timestamp 类型允许在格式'YYYY-MM-DDhh:mm:ss[.frac]'后面带上时区的偏移信息[+/-hh:mm:ss]
- 类似兼容后的 date 类型，如果要想输入年份大于等于 10000 的时间戳，请使用'YYYY-MM-DDhh:mm:ss[.frac]'这种格式
- 注意，timestamp 类型在 MySQL 一端为不带时区的时间戳，而在 GBase8c 一端为带时区的时间戳，实际上兼容后 timestamp 类型在内部会使用 timestamptz 类型存储，请用户在使用前注意这种区别，如想使用不带时区的时间戳，可以使用 datetime 类型。
- 注意：由于 MySQL 一端没有 timestampwith[out]timezone 这种语法，但是我们仍然在 GBase8c 保留这种语法。timestampwithtimezone 等价于直接原来 GBase8ctimestamptz 类型，timestampwithouttimezone 等价于直接使用原来 GBase8c 中的 timestamp 类型(并非兼容后的 timestamp 类型，是指 GBase8c 原有的不带时区属性的 timestamp 类型)

示例

(1) 创建表并插入 timestamp 类型数据。

```
CREATE TABLE test_timestamp(ts timestamp(2));
INSERT INTO test_timestamp VALUES('2012-10-21 23:55:23-12:12');
INSERT INTO test_timestamp VALUES(201112234512);
```

(2) 查看表数据。

```
SELECT * FROM test_timestamp;
           ts
------------------------
 2020-11-12 23:45:12+08
 2012-10-22 20:07:23+08
(2 rows)
```

(3) 变更时区，并再次查看表数据。

```
SET TIME ZONE UTC;
SELECT * FROM test_timestamp;
           ts
------------------------
 2020-11-12 15:45:12+00
 2012-10-22 12:07:23+00
(2 rows)
```

year/year(4)，year(2)类型输入

支持如下格式：

格式	含义
'YYYY' 'YY'	年，当输入两位数年份时，若值小于 70，则实际年份需要加上 2000，例如'69'表示 2069 年；若值大于等于 70，则实际年份需要加上 1900，例如'70'表示 1970 年
YYYY YY	年

备注

三种类型都接受相同的输入格式和范围，区别仅在于 year(2)类型输出格式只为 2 位数如果输入'0'，将解析为 2000 年；但是当输入的是整数 0，GBase 8c 会解析成为 0，表示 year 类型的 0 值。

示例

(1) 创建表并插入 year 类型数据。

```
CREATE TABLE test_year(y year,y2 year(2));
INSERT INTO test_year VALUES ('70', '70');
INSERT INTO test_year VALUES ('69', '69');
INSERT INTO test_year VALUES ('2069', '2069');
INSERT INTO test_year VALUES ('1970', '1970');
```

(2) 查看表数据。

```
SELECT * FROM test_year;
  y   | y2
------+----
 1970 | 70
 2069 | 69
 2069 | 69
 1970 | 70
(4 rows)
```

2.4.1.2.4 位串类型

相比于 GBase8c 原生语法，dolphin 对于位串类型的修改主要为：

(1) bit 类型的数据是最长为 n 的变长类型，超过 n 的类型会被拒绝。bit varying 类型的数据是最长为 n 的变长类型，超过 n 的类型会被拒绝。

(2) 如果用户明确地把一个位串值转换成 bit(n)，则此位串右边的内容将被截断或者在左边补齐零，直到刚好 n 位，而不会抛出任何错误。

示例

(1) 创建测试表。

```
CREATE TABLE bit_type_t1
(
    BT_COL1 INTEGER,
    BT_COL2 BIT(3),
    BT_COL3 BIT VARYING(5)
) ;
```

(2) 将不符合类型长度的数据进行转换。

```
INSERT INTO bit_type_t1 VALUES(2, B'1000'::bit(3), B'101');
```

(3) 查询数据。

```
SELECT * FROM bit_type_t1;
```

返回结果为：

```
 bt_col1 | bt_col2 | bt_col3
---------+---------+---------
       2 | 100     | 101
(1 row)
```

(4) 对长度不足的未串转换为 bit(n)，会在最左侧补齐零。

```
SELECT B'10'::bit(4);
```

返回结果为：

```
 bit
------
 0010
(1 row)
```

(5) 删除表。

```
DROP TABLE bit_type_t1;
```

2.4.1.2.5 枚举类型

ENUM 类型是一个字符串对象，其值是从创建表时在列定义时指定的枚值列表中选择的。要使用 mysql 兼容的 enum 类型，首先保证数据库是'B'类型：

```
CREATE DATABASE test_db with dbcompatibility='B';
```

创建和使用 ENUM 列

枚举值必须是字符串。例如，以创建一个包含一个类型是 ENUM 的列的表，如下所示：

```
CREATE TABLE staff(name VARCHAR(40),gender ENUM('male','female'));
INSERT INTO staff(name,gender) VALUES ('Tom','male'),('Jenny','female') SELECT
name,gender FROM staff WHERE gender='male';
name|gender
---------+--------
Tom|male
(1row)
```

枚举值字符串中不能包含'anonymous_enum'，同时不能将一个已有的类型重命名为包含'anonymous_enum'的名称，如果包含，会报错：

```
CREATE TYPE country_anonymous_enum_1 AS enum('CHINA','USA');
ERROR:enumtypename"country_anonymous_enum_1"can'tcontain"anonymous_enum"
```

枚举值的索引

根据列定义中的枚举值的的顺序，每个枚举值都会被分配一个索引值，从 1 开始。NULL 值的索引是 0。

这里的“索引”指的是枚举值在创建时位于列表中的位置，与 table 中的位置无关。例如，指定为 ENUM('male','female')的列具有以下枚举值及索引。

Value	Index

NULL	0
'male'	1
'female'	2

- 可以使用索引号在 ENUM 中插入枚举值，也可以在 WHERE 子句中使用索引号筛选枚举值，如下所示：

```
INSERT INTO staff (name, size) VALUES ('Jone',1);
SELECT name, gender FROM staff WHERE gender = 1;
```

返回如下结果：

```
   name     | gender
------------+-------
   Tom      | male
   Jone     | male
(2 rows)
```

- 如果使用的索引值超过了枚举值的个数或者为负值，则会出现错误。

```
INSERT INTO staff (name, gender) VALUES ('Lara',4);
```

返回错误信息：

```
ERROR:  enum order 4 out of the enum value size: 2
LINE 1: INSERT INTO staff (name, gender) VALUES ('Lara',4);
                                                        ^
CONTEXT:  referenced column: size
```

空值和空字符串

枚举值可以是 NULL，同时空字符串"也是会被当作是 NULL 值。

如果在 ENUM 的列中插入无效的值(即枚举列表中不存在的字符串)，则会导致错误。

枚举限制

数字不能作为枚举值。如果要将数字作为为枚举值，需要将其包含在括在引号中，即变成字符串。如果没有引号，该数字将被当作索引。

ENUM 的定义中的不能包含重复的枚举值。

ENUM 的值支持的字符串最大长度为 63。

ENUM 的枚举值没有最大元素个数的限制。

2.4.1.2.6 布尔类型

相比于 GBase8c 原生语法，dolphin 对于布尔类型的修改主要为：

(1) 将布尔类型的输出表现从't'和'f'修改为'1'和'0'。此修改仅在除了 gs_dump，gs_dumpall，gsql，gs_probackup，gs_rewind，gs_clean 以外的工具生效，如 JDBC。

示例

gsql 中，布尔类型回显仍是't'和'f'。

```
SELECT true;
bool
------
t
(1 row)
SELECT false;
bool
------
f
(1 row)
```

2.4.1.2.7 二进制类型

相比于 GBase8c 原生语法，dolphin 对于二进制类型的修改主要为:

(1) 新增 BINARY/VARBINARY/TINYBLOB/MEDIUMBLOB/LONGBLOB 类型。

(2) 对 BLOB 类型的输入函数进行了修改，在 dolphin.b_compatibility_mode 为 on 的情况下，其输入可以兼容 MySQL 数据库的普通字符串输入，输出则需要再将 bytea_output 参数设置为 escape 才能输出对应的字符串，否则会被转换成十六进制字符串的形式进行输出。

(3) 对 TINYBLOB/MEDIUMBLOB/LONGBLOB 类型，在 dolphin.b_compatibility_mode 为 off 的情况下也可以兼容 MySQL 数据库的普通字符串输入，输出则需要再将 bytea_output 参数设置为 escape 才能输出对应的字符串，否则会被转换成十六进制字符串的形式进行输出。

(4) 对 BINARY 类型的输入函数进行了修改，可支持 MySQL 数据库中的转义字符识别。

(5) 新增 BIANRYEXPR 用法，用在任意表达式前的 BINARY 关键字，表示将此表达式转化为二进制类型。

表 1-7 二进制类型

名称	描述	存储空间
BLOB	二进制大对象。 说明： 列存不支持 BLOB 类型。 BLOB 类型仅在 dolphin.b_compatibility_mode 为 on 的情况下才能够兼容 MySQL 数据库接收普通字符串输入的功能。	最大为 32TB（即 35184372088832 字节）。
TINYBLOB	二进制大对象。 说明： 列存不支持 TINYBLOB 类型。 TINYBLOB 类型不需要开启 dolphin.b_compatibility_mode 就可以兼容 MySQL 数据库接收普通字符串输入的功能。	最大为 255 字节
MEDIUMBLOB	二进制大对象。 说明： 列存不支持 MEDIUMBLOB 类型。 MEDIUMBLOB 类型不需要开启 dolphin.b_compatibility_mode 就可以兼容 MySQL 数据库接收普通字符串输入的功能。	最大为 16M-1 字节
LONGBLOB	二进制大对象。 说明： 列存不支持 LONGBLOB 类型。 LONGBLOB 类型不需要开启 dolphin.b_compatibility_mode 就可以兼容 MySQL 数据库接收普通字符串输入的功能。	最大为 4G-1 字节
RAW	变长的十六进制类型。 说明： 列存不支持 RAW 类型。	4 字节加上实际的十六进制字符串。最大为 1GB-8203 字节（即 1073733621 字节）。

名称	描述	存储空间
BYTEA	变长的二进制字符串。	4 字节加上实际的二进制字符串。最大为 1GB-8203 字节 (即 1073733621 字节）。
BINARY	定长的二进制字符串。	4 字节加上实际的二进制字符串（255 字节）。最大为 259 字节。
VARBINARY	变长的二进制字符串。	4 字节加上实际的二进制字符串(65535 字节)。最大为 65539 字节。
BYTEAWITHOUTORDERWITHEQUALCOL	变长的二进制字符串（密态特性新增的类型，如果加密列的加密类型指定为确定性加密，则该列的实际类型为 BYTEAWITHOUTORDERWITHEQUALCOL），元命令打印加密表将显示原始数据类型。	4 字节加上实际的二进制字符串。最大为 1GB 减去 53 字节（即 1073741771 字节）。
BYTEAWITHOUTORDERCOL	变长的二进制字符串（密态特性新增的类型，如果加密列的加密类型指定为随机加密，则该列的实际类型为 BYTEAWITHOUTORDERCOL），元命令打印加密表将显示原始数据类型。	4 字节加上实际的二进制字符串。最大为 1GB 减去 53 字节（即 1073741771 字节）。
_BYTEAWITHOUTORDERWITHEQUALCOL	变长的二进制字符串，密态特性新增的类型。	4 字节加上实际的二进制字符串。最大为 1GB 减去 53 字节（即 1073741771 字节）。
_BYTEAWITHOUTORDERCOL	变长的二进制字符串，密态特性新增的类型。	4 字节加上实际的二进制字符串。最大为 1GB 减去 53 字节（即 1073741771 字节）。

说明：

- 除了每列的大小限制以外，每个元组的总大小也不可超过 1GB-8203 字节(即 1073733621 字节)。
- 不支持直接使用 BYTEA WITHOUT ORDER WITH EQUAL COL、BYTEA WITHOUT ORDER COL、_BYTEA WITHOUT ORDER WITH EQUAL COL 和

_BYTEA WITHOUT ORDER COL 类型创建表。

示例

(1) 创建表

```
CREATE TABLE blob_type_t1
(
    BT_COL1 INTEGER,
    BT_COL2 BLOB,
    BT_COL3 RAW,
    BT_COL4 BYTEA
) ;
```

(2) 插入数据。

```
INSERT INTO blob_type_t1
VALUES(10,empty_blob(),HEXTORAW('DEADBEEF'),E'\\xDEADBEEF');
```

(3) 查询表中的数据。

```
SELECT * FROM blob_type_t1;
 bt_col1 | bt_col2 | bt_col3  |  bt_col4
---------+---------+----------+------------
      10 |         | DEADBEEF | \xdeadbeef
(1 row)
```

(4) 使用 BINARY 转化

```
select 'a\t'::binary;
 binary
--------
 \x6109
(1 row)
select binary 'a\b';
 binary
--------
 \x6108
(1 row)
```

2.4.1.3 函数和操作符

2.4.1.3.1 赋值操作符

相比于原始的 GBase8c，dolphin 对于赋值操作符的修改主要为:新增支持通过:=的方式赋值。如 UPDATE table_name SET col_name:=new_val;。

2.4.1.3.2 字符处理函数和操作符

相比于 GBase8c 原生语法，dolphin 对于字符处理函数和操作符的修改主要为：

(1) 新增 regexp/notregexp/rlike 操作符。

(2) 新增 locate、lcase

(3) ucase、insert、bin、char、elt、field、find_int_set、hex、space、soundex、export_set、ord、substring_index、from_base64 函数。

(4) 修改 length/bit_length/octet_length/convert/format/left/right 函数的表现。

(5) 新增^操作符的异或功能，新增 likebinary/notlikebinary 操作符。

(6) 修改 like/notlike 操作符的表现。

(7) 新增!操作符，可在表达式前使用，其效果与 NOT 一致。

(8) 新增 text_bool/varchar_bool/char_bool 函数。

- bit_length(string)

  描述：字符串的位数。对于二进制输入，将向上补齐至 8 的倍数。

  返回值类型：int

  示例：

```
SELECT bit_length('world');
bit_length
------------
40
(1row)
SELECT bit_length(b'010');
bit_length
------------
8
(1row)
```

- insert(destext,startint,lengthint,srctext)

  描述：在原字符串的指定位置插入一个新字符串，并从指定位置开始替换掉原字符串一定数量的字符。

  返回值类型：text

  示例：

```
select insert('abcdefg',2,4,'yyy');
insert
--------
ayyyfg
(1row)
```

- lcase(string)

  描述：把字符串转化为小写，等价于 lower。

  返回值类型：varchar

  示例：

```
SELECT lcase('TOM');
 lcase
-------
 tom
(1 row)
```

- length(string)

  描述：获取参数 string 中字符的数目。对于多字符编码(如中文)，返回字节数。

  返回值类型：integer

  示例：

```
SELECT length('abcd');
length
--------
4
(1row)
SELECT length('中文');
 length
--------
      6
(1 row)
```

- format(valnumber,dec_numint[,localestring])

  描述：将 val 以"x,xxx,xxx.xx"的格式返回。val 将保留 dec_num 位小数。保留的小数位数最多为 32 位，若 dec_num 大于 32，则以保留 32 位小数返回。若 dec_num 为 0，则返回内容无小数点及小数部分数字。第三个参数可选，可以根据 locale 指定返回内容的小数点及千位分隔符的格式。如果没有指定第三个参数，或第三个参数值非法，则使用默认值'en_US'。

注意：此 format 函数针对 B 兼容数据库使用，与 GBase8c 原有的 format 函数语义不同。若想使用此语义，请创建 B 兼容模式数据库，启用 MySQL 兼容性 SQL 引擎插件，并将 dolphin.b_compatibility_mode 设置为 TRUE.

返回值类型：text

执行如下命令，配置 B 兼容模式数据库：

```
CREATE DATABASE B_COMPATIBILITY_DATABASE DBCOMPATIBILITY 'B';
\c B_COMPATIBILITY_DATABASE
CREATE Extension dolphin;
SET dolphin.b_compatibility_mode = TRUE;
```

示例：

```
select format(1234.4567,2);
  format
-----------
 1,234.46
(1 row)
 select format(1234.5,4);
  format
-----------
 1,234.5000
(1 row)
select format(1234.5,0);
  format
-----------
 1,235
(1 row)
 select format(1234.5,2,'de_DE');
  format
-----------
 234,50
```

- (1 row)
- hex(numberorstringorbyteaorbit)

描述：把数字、字符、二进制字符类型或位串类型转换成十六进制表现形式

注意：GBase8c 将反斜杠"单独看做一个字符，因此对于字符串'\n'，其长度为 2。

返回值类型：text

示例：

```
SELECT hex(256);
 hex
-----
 100
(1 row)
select hex('abc');
 hex
--------
 616263
(1 row)
select hex('abc'::bytea);
 hex
--------
 616263
(1 row)
select hex(b'1111');
 hex
-----
 0f
(1 row)
select hex('\n');
 hex
-------
 5c6e
(1 row)
```

- locate(substring,string[,position])

描述：从字符串 string 的 position(缺省时为 1)所指的位置开始查找并返回第 1 次出现子串 substring 的位置的值。字符串区分大小写。

当 position 为 0 时，返回 0。

当 position 为负数时，从字符串倒数第 n 个字符往前逆向搜索。n 为 position 的绝对值。

返回值类型：integer，字符串不存在时返回 0。

示例：

```
SELECT locate('ing', 'string');
 locate
--------
```

```
     4
(1 row)
SELECT locate('ing', 'string', 0);
 locate
--------
     0
(1 row)
SELECT locate('ing', 'string', 5);
 locate
--------
     0
(1 row)
```

- octet_length(string)

  描述：等价于 length。

  返回值类型：int

  示例：

```
SELECT octet_length('中文');
 octet_length
--------------
            6
(1 row)
```

- source_string regexp pattern

  描述：正则表达式的模式匹配操作符。

  source_string 为源字符串，pattern 为正则表达式匹配模式。

  返回值类型：integer(0 或 1)

  示例：

```
SELECT 'str' regexp '[ac]' AS RESULT;
 result
--------
     0
(1 row)
```

- source_string not regexp pattern

  描述：regexp 的结果取反。

source_string 为源字符串，pattern 为正则表达式匹配模式。

返回值类型：integer(0 或 1)

示例：

```
SELECT 'str' not regexp '[ac]' AS RESULT;
 result
--------
      1
(1 row)
```

- source_string rlike pattern

  描述：等价于 regexp。

  source_string 为源字符串，pattern 为正则表达式匹配模式。

  返回值类型：integer(0 或 1)

  示例：

```
SELECT 'str' rlike '[ac]' AS RESULT;
 result
--------
      0
(1 row)
```

- ucase(string)

  描述：把字符串转化为大写。等价于 upper。

  返回值类型：varchar

  示例：

```
SELECT ucase('tom');
 ucase
-------
 TOM
(1 row)
```

- bin(number or string)

  描述：返回 N 整型或者数字字符的二进制字符串，汉字返回 0。

  返回值类型：text

  示例：

```
SELECT bin('309');
bin
------------
100110101
(1 row)
SELECT bin('你好');
bin
---
0
(1 row)
```

- char(any)

  描述：根据 ASCII 码对多个数字转换为多个字符。

  返回值类型：text

  示例：

```
select char(77,77.3,'77.3','78.8',78.8);
char
-------
MMMNO
(1 row)
```

- char_length(string)或 character_leng(string)

  描述：字符串中的字符个数,一个汉字长度为 1，并且支持二进制类型。

  返回值类型：int

  示例：

```
SELECT char_length('hello');
char_length
-------------
          5
(1 row)
SELECT char_length(B'101');
char_length
-------------
          1
(1 row)
```

- convert(exprusingtranscoding_name)

  描述：通过 transcoding_name 指定的编码方式转换 expr;

注意：默认库中支持如下格式：convert(stringbytea,src_encodingname,dest_encodingname);以 dest_encoding 指定的编码方式转换 bytea，dolphin 下支持通过 using 关键字后 transcoding_name 指定要转换的编码方式，对 expr 进行转换，不支持上述三个参数的表示方式。

返回值类型：text

示例：

```
 select convert('a' using 'utf8');
convert
---------
a
(1 row)
select convert('a' using utf8);
convert
---------
a
(1 row)
```

- elt(number, str1,str2,str3,•••)

  描述：返回后面字符串的第 N 个字符串。

  返回值类型：text

  示例：

```
select elt(3,'wo','ceshi','disange');
elt
---------
disange
(1 row)
```

- left(str text, n int)

  描述：返回字符串的前 n 个字符。当 n 是负数时，当做 0 处理。

  返回值类型：text

  示例：

```
select left('abcde', 2);
left
------
ab
```

```
(1 row)
select left('abcde', 0);
left
------
(1 row)
select left('abcde', -2);
left
------
(1 row)
```

- right(str text, n int)

  描述：返回字符串中的后 n 个字符。当 n 是负值时，当做 0 处理。

  返回值类型：text

  示例：

```
select right('abcde', 2);
right
-------
de
(1 row)
select right('abcde', 0);
right
-------
(1 row)
select right('abcde', -2);
right
-------
(1 row)
```

- field(str, str1,str2,str3,•••)

  描述：获取 str 在后面 strn 中的位置，不区分大小写。

  返回值类型：int

  示例：

```
 select field('ceshi','wo','ceshi','disange');
field
-------
    2
(1 row)
```

- find_int_set(str, strlist)

  描述：获取 str 在后面 strlist 中的位置，不区分大小写,strlist 以，分割。

  返回值类型：int

  示例：

```
select find_int_set('ceshi','wo','ceshi,ni,wo,ta');
find_int_set
-------------
           3
(1 row)
```

- space(number)

  描述：返回 N 个空格

  返回值类型：text

  示例：

```
 select space('5');
space
-------
(1 row)
```

- soundex(str)

  描述：返回描述指定字符串的语音表示的字母数字模式的算法

  返回值类型：text

  示例：

```
 select soundex('abcqwcaa');
soundex
---------
A120
(1 row)
```

- make_set(number, string1, string2, …)

  描述：返回一个由 number 中设置了相应位的字符串组成的设置值（包含子字符串的字符串，以,分隔）。string1 对应位 0，string2 对应位 1，依此类推。 string1，string2，…中的 NULL 值不添加到结果中。

  返回值类型：text

```
select make_set(1|4, 'one', 'two', NULL, 'four');
 make_set
----------
 one
(1 row)
```

- like/not like

描述：判断字符串能否匹配上 LIKE 后的模式字符串。GBase 8c 的原 like 为大小写敏感匹配，现将其改为当 dolphin.b_compatibility_mode 为 TRUE 时大小写不敏感匹配，当 dolphin.b_compatibility_mode 为 FALSE 时大小写敏感匹配。若字符串与提供的模式匹配，则 like 表达式返回真(ilike 返回假)。

注意事项：

在 dolphin 插件中增加了该操作符对 true/false 的 bool 类型兼容；

对于定长字符串 char，若插入表的字符串长度小于指定长度则会在字符串末尾自动填充空格，在 dolphin 插件中，该操作符处理定长字符串类型时忽略末尾多余空格。

返回值类型：布尔型

示例：

```
SELECT 'a' like 'A' as result;
 result
------------
         t
(1 row)
 SELECT 'abc' like 'a' as result;
 result
------------
          f
(1 row)
 SELECT 'abc' like 'A%' as result;
 result
------------
          t
(1 row)
SELECT true like true as result;
 result
------------
          t
(1 row)
```

- like binary/not like binary

描述：判断字符串能否匹配上 LIKE BINARY 后的模式字符串,like binary 采用大小写敏感模式匹配，若模式匹配则返回真(not like binary 返回假)，不匹配则返回假(not like binary 返回真)。

返回值类型：布尔型

示例：

```
SELECT 'a' like binary 'A' as result;
 result
------------
         f
(1 row)
 SELECT 'a' like binary 'a' as result;
 result
------------
         t
(1 row)
 SELECT 'abc' like binary 'a' as result;
 result
------------
          f
(1 row)
 SELECT 'abc' like binary 'a%' as result;
 result
------------
          t
(1 row)
```

- substring_index(str, delim, count)

描述：substring_index(str, delim, count)返回 str 的开始位置至匹配到第 count 次 delim 的位置之间的子字符串，count 表示匹配的次数。若 count 为正数，则从 str 的左边开始匹配，并返回匹配位置左边的子字符串；若 count 为负数，则从 str 的右边开始匹配，并返回匹配位置右边的子字符串。count 取值范围为 INT64_MIN—INT64_MAX。

返回值类型：text

示例：

```
SELECT substring_index('abcdabcdabcd', 'bcd', 2);
substring_index
```

```
----------------
 abcda
(1 row)
```

- export_set(bits, on, off, separator, number of bits)

描述：返回一个字符串，该字符串将显示位数。该函数需要 5 个自变量才能起作用。该函数将第一个参数(即整数)转换为二进制数字，如果二进制数字为 1，则返回“on”，如果二进制数字为 0，则返回“off”。

返回值类型：text

示例：

```
SELECT EXPORT_SET(5,'Y','N',',',5);
 export_set
-------------
 Y,N,Y,N,N
(1 row)
```

- FROM_BASE64

描述:根据 BASE64 编码规则，将一个 BASE64 编码的字符串解码，返回字符串的解码结果。

返回值类型：text

编码规则：

- 将输入的每三个字节（24 位）变成四个字节（32 位）：6 位为一组，高位补两个 0，组成一个字节，这样正好能将三个字节补成四个字节，其中每个字节只会对应 0(00000000)到 63(00111111)。
- 每 76 个字符加一个换行符。
- 编码 0(00000000)到 61(00111111)对应 A-Z，a-z，0-9 共 62 个字符，62(00111110)的编码是'+'，63(00111111)的编码是'/'。
- 若输入的字符串字节数不为三的倍数，那么剩余的字节根据编码规则转换，若有一个字节不满 8 位，则在低位补 0 补满 8 位，同时用'='将转换结果补满四个字节。若最后一组只有两个字节，每 6 位一组，第三组只有 4 位，低位要补两个 0，然后这三组再分别高位补两个 0，转成三个字符，末尾补一个'='；若最后一组只有一个字节，每 6 位一组，第二组只有 2 位，低位要补四个 0，然后这两组再分别高位补两个 0，转成两个字符，末尾补两个'='。

解码规则：

- 将输入的字符串用二进制表示，去掉每个字节高位的两个 0。
- 根据编码规则，正确的编码字节数必为 4 的倍数。若末尾有'='，则根据'='数量去掉最后一个除'='以外的字节低位的 0。若末尾有一个'='，即最后四个字节为'***='，则将前三个字节转二进制后再去掉最后两个 0，若末尾有两个'='，即最后四个字节为'**=='，则将前两个字节转二进制后再去掉最后四个 0。
- 将去掉高位 0 后的各个字节按顺序拼接，每 8 位转成一个字符。

例子 1：YWJj

字符串用二进制表示为：00011000(Y)00010110(W)00001001(J)00100011(j)。

去掉每个字节高位的两个 0 后变成：011000 010110 001001 100011。

将去掉高位 0 后的各个字节按顺序拼接成：01100001(a)01100010(b)01100011(c)。

故解码结果为 abc。

例子 2：YWI=

字符串用二进制表示为：00011000(Y)00010110(W)00001000(I)。

去掉每个字节高位的两个 0 后变成：011000 010110 001000。

由于末尾有一个'='，则第三个字节末尾的 0 也要去掉再拼接：01100001 01100010。

故解码结果为 ab。

示例：

```
SELECT FROM_BASE64('YWJj');
 from_base64
 -------------
 abc
(1 row)
```

- ORD(str)

描述：返回 str 的最左边的字符的数值，并使用下面公式计算该字符组成字节的对应数值：

```
 (1st byte code)
+ (2nd byte code  256)
+ (3rd byte code  256^2) ...
```

返回值类型：INT

示例：

```
select ord('1111');
ord
-----
  49
(1 row)
select ord('sss111');
ord
-----
115
(1 row)
```

- TO_BASE64(str)

描述：根据 BASE64 编码规则，将一个字符串编码成 BASE64 编码格式，返回字符串的编码结果。编码规则与解码规则和 FROM_BASE64 相同。

返回值类型：text

注意事项

如果输入的是 NULL，那么返回的结果为 NULL。

编码解码规则与函数 FROM_BASE64 相同。

例子 1：abc

将字符串用二进制表示：01100001(a)01100010(b)01100011(c)

将二进制串拆分，每 6 位一组：011000 010110 001001 100011

在每一组的高位都补上两个 0：00011000 00010110 00001001 00100011

查找 base64 编码转换表：00011000，00010110，00001001，00100011 对应的字符为：Y,W,J,j

故编码结果为 YWJj

例子 2：ab

将字符串用二进制表示：01100001(a)01100010(b)

将二进制串拆分，每 6 位一组，由于最后一组只有 4 位，在低位补 0 补够 6 位：011000 010110 0010(00)

在每一组的高位都补上两个 0：00011000 00010110 00001000

查找 base64 编码转换表：00011000，00010110，00001000 对应的字符为：Y,W,I

由于输入的字符串字节数不为三的倍数，导致转换后的字符数不为 4 的倍数，所以最后需要用=号补满 4 个字节，最终编码结果为：YWI=

示例：

```
SELECT TO_BASE64('to_base64');
  to_base64
--------------
dG9fYmFzZTY0
(1 row)
SELECT TO_BASE64('123456');
 to_base64
-----------
 MTIzNDU2
(1 row)
SELECT TO_BASE64('12345');
 to_base64
-----------
 MTIzNDU=
(1 row)
SELECT TO_BASE64('1234');
 to_base64
-----------
 MTIzNA==
(1 row)
```

- UNHEX(str)

描述：将一个十六进制编码的字符串解码，一个十六进制字符变成 4 位二进制，两个十六进制字符（8 位）解码为一个字符，返回字符串的解码结果。若十六进制字符串的字符数不为偶数，则在高位补 0。若输入的是二进制格式的字符串，则返回 NULL。

返回值类型：text

注意事项

如果输入的是 NULL 或者包含非十六进制字符，那么返回的结果为 NULL。

若输入的是数字，则将数字转成字符串后进行解码，如需将十六进制数转为十进制数，则需要使用其它的函数

编码解码规则与函数 HEX 相同。

例子 1：4142

将每个十六进制字符用 4 位二进制表示，若十六进制字符数不为偶数，则在高位补 0：0100(4)0001(1)0100(4)0010(2)

每 8 位组成一个字符：01000001 01000010

故解码结果为 AB

示例：

```
SELECT UNHEX(HEX('string'));
 unhex
--------
 string
(1 row)
SELECT HEX(UNHEX('1267'));
 hex
------
 1267
(1 row)
```

2.4.1.3.3 数字操作函数和操作符

相比于 GBase8c 原生语法，dolphin 对于时间/日期函数的修改主要为：

(1) 新增 DIV/MOD/XOR/^操作符。

(2) 新增 truncate/rand/crc32/conv/float8_bool/oct/float4_bool 函数。

- DIV

描述：除（取整）

示例：

```
SELECT 8 DIV 3 AS RESULT;
 result
--------
      2
(1 row)
```

- MOD

描述：模（求余）

示例：

```
SELECT 4 MOD 3 AS RESULT;
 result
--------
      1
(1 row)
```

- XOR

  描述：二进制 XOR

  示例：

```
SELECT 4 XOR 3 AS RESULT;
 result
--------
      0
(1 row)
```

- truncate(v numeric, s int)

  描述：截断为 s 位小数。等价于 trunc

  返回值类型：numeric

  示例：

```
SELECT truncate(44382, 2);
 truncate
----------
    443
(1 row)
```

- rand()

  描述：0.0 到 0 之间的随机数。等价于 random

  返回值类型：double precision

  示例：

```
SELECT rand();
       rand
-------------------
 0.254671605769545
(1 row)
```

- crc32(string)

  描述：计算 string 的 crc32 数值

  返回值类型：int

  示例：

```
SELECT crc32('abc');
   crc32
-----------
 891568578
(1 row)
```

- conv(input in, current_base int, new_base int)

  描述：将数字或字符串从一个数字基本系统转换为另一个数字基本系统。in 支持数字和字符串两种类型

  返回值类型：text

  示例：

```
SELECT conv(20, 10, 2);
 conv
-------
 10100
(1 row)
SELECT conv('8D', 16, 10);
 conv
------
 141
(1 row)
```

- ^

  描述：实现两个整数之间的按位异或。

  注意：

  异或操作符不支持非 0/1 的 bool 类型，也不支持 raw 类型。

  在 dolphin 插件中，^操作符不再支持对两个整数之间的幂运算。如需要使用两个整数之间的幂运算操作，可以使用 power 函数。

  返回值类型：INT

示例：

```
SELECT 1^1;
?column?
----------
       0
 (1 row)
SELECT 2 ^ 3;
 ?column?
----------
        1
(1 row)
SELECT power(2,3);
 power
-------
     8
(1 row)
```

- ::float

描述：当 set dolphin.b_compatibility_mode=0 后，可以实现对 float 数据的幂运算。当 set dolphin.b_compatibility_mode=1 后，可以实现对 float 数据的按位异或，对 float 数据四舍五入后异或。

返回值类型：DOUBLE

示例：

```
select 0.5678::float^1234::float;
 ?column?
----------
        0
(1 row)
```

- float8_bool(float)

描述：根据浮点数的取值返回布尔型（为零时返回 false，否则返回 true）。

返回值类型：boolean

示例：

```
select float8_bool(0.1);
 float8_bool
-------------
 t
```

```
(1 row)
select float8_bool(0.0);
 float8_bool
-------------
 f
(1 row)
```

- oct(input N)

  描述：将数字或字符串从一个十进制数字转换为八进制数字。

  返回值类型：text

  示例：

```
SELECT OCT(10);
 oct
-----
12
(1 row)
SELECT OCT('10');
 oct
-----
12
(1 row)
```

- float4_bool(float)

  描述：根据浮点数的取值返回布尔型（为零时返回 false，否则返回 true）。

  返回值类型：boolean

  示例：

```
select float4_bool(0.1);
 float4_bool
-------------
 t
(1 row)
select float4_bool(0.0);
 float4_bool
-------------
 f
(1 row)
```

2.4.1.3.4 Dolphin 锁

如果需要保持数据库数据的一致性，可以使用 LOCK TABLES 来阻止其他用户修改表。

例如，一个应用需要保证表中的数据在事务的运行过程中不被修改。为实现这个目的，则可以对表使用进行锁定。这样将防止数据不被并发修改。

LOCK TABLES 使用后，会让接下来的 sql 处于事务状态中，所以需要用 UNLOCK TABLES 手动释放锁并结束事务。

另外如果需要对当前 session 只允许读的话，那么还可以用 FLUSH TABLES WITH READ LOCK 实现，之后也需要用 UNLOCK TABLES 手动结束这个功能。

语法格式

上锁

```
LOCK TABLES namelist READ/WRITE
```

让当前 session 处于只读表的状态

```
FLUSH TABLES WITH READ LOCK
```

解锁

```
UNLOCK TABLES
```

参数说明

- namelist

  要锁定的表的名称，可以有多个表。

- READ/WRITE

  锁的模式。有：

  READ

  读锁，只读取表而不修改的锁模式。

  WRITE

  写锁，这个模式保证其所有者（事务）是可以访问该表的唯一事务。

  示例

  在执行删除操作时对一个表进行 WRITE 锁。

  创建示例表。

```
CREATE TABLE graderecord
            (
            number INTEGER,
            name CHAR(20),
            class CHAR(20),
            grade INTEGER
            );
```

插入数据。

```
insert into graderecord values('210101','Alan','201',92);
```

上锁。

```
LOCK TABLES graderecord WRITE;
```

删除示例表，删除锁。

```
DELETE FROM graderecord WHERE name ='Alan';
UNLOCK TABLES;
```

2.4.1.3.5 时间和日期处理函数和操作符

相比于 GBase8c 原生语法，dolphin 对于时间/日期函数的修改主要为:

(1) 新增 dayofmonth/dayofweek/dayofyear/hour/microsecond/minute/quarter/second/weekday/weekofyear/year/current_date 函数。

(2) 新增 curdate/current_time/curtime/current_timestamp/localtime/localtimestamp/now/sysdate 函数。

(3) 新增 makedate/maketime/period_add/period_diff/sec_to_time/subdate 函数。

(4) 新增 subtime/timediff/time/time_format/timestamp/timestamppadd 函数。

(5) 新增 to_days/to_seconds/unix_timestamp/utc_date/utc_time/utc_timestamp 函数。

(6) 新增 date_bool/time_bool 函数。

(7) 新增 dayname/monthname/time_to_sec/month/day/date/week/yearweek 函数，修改了 last_day 函数。

(8) 新增 datediff/from_days/convert_tz/date_add/date_sub/adddate/addtime 函数，修改了 timestampdiff 函数。

(9) 新增 get_format/date_format/from_unixtime/str_to_date 函数，修改了 extract 函数。

- curdate()

  描述：返回语句开始执行的日期。

  返回值类型：date

  示例：

```
select curdate();
  curdate
------------
 2024-02-26
(1 row)
```

- current_time

  描述：返回语句开始执行的时间。

  返回值类型：time

  示例：

```
select current_time;
current_time
--------------
16:56:02
(1 row)
```

- current_time(n)

  描述：返回语句开始执行的时间，n 为精度，最大值取 6。

  返回值类型：time

  示例：

```
select current_time(3);
 current_time(3)
-----------------
16:57:23.255
(1 row)
select current_time();
 current_time()
----------------
17:05:01
(1 row)
```

- curtime(n)

描述：返回语句开始执行的时间，n 为精度，最大值取 6。

返回值类型：time

示例：

```
select curtime(3);
  curtime(3)
--------------
 15:11:20.275
(1 row)
select curtime();
 curtime()
-----------
 15:11:38
(1 row)
```

- current_timestamp

  描述：返回语句开始执行的时间戳。

  返回值类型：datetime

  示例：

```
select current_timestamp;
  current_timestamp
---------------------
 2024-02-26 15:12:41
(1 row)
```

- current_timestamp(n)

  描述：返回语句开始执行的时间戳，n 为精度，最大值取 6。

  返回值类型：datetime

  示例：

```
select current_timestamp(3);
  current_timestamp(3)
-------------------------
 2024-02-26 15:12:15.119
(1 row)
select current_timestamp();
 current_timestamp()
---------------------
```

```
 2024-02-26 15:12:28
(1 row)
```

- dayofmonth(timestamp)

  描述：获取日期/时间值中天数的值。

  返回值类型：double precision

  示例：

```
SELECT dayofmonth(timestamp '2001-02-16 20:38:40');
 date_part
-----------
        16
(1 row)
```

- dayofweek(timestamp)

  描述：获取日期/时间值中星期的标号（1 代表星期日，2 代表星期一，以此类推，7 代表星期六）。

  返回值类型：double precision

  示例：

```
SELECT dayofweek(timestamp '2001-02-16 20:38:40');
 ?column?
----------
        6
(1 row)
```

- dayofyear(timestamp)

  描述：获取日期/时间值中一年的第几天。

  返回值类型：double precision

  示例：

```
SELECT dayofyear(timestamp '2001-02-16 20:38:40');
 date_part
-----------
        47
(1 row)
```

- hour(timestamp)

  描述：获取日期/时间值中小时的值。

返回值类型：double precision

示例：

```
SELECT hour(timestamp '2001-02-16 20:38:40');
 date_part
-----------
        20
(1 row)
```

- localtime

  描述：返回语句开始执行的时间戳。

  返回值类型：datetime

  示例：

```
select localtime;
      localtime
--------------------
 2024-02-26 15:13:55
(1 row)
```

- localtime(n)

  描述：返回语句开始执行的时间戳，n 为精度，最大值取 6。

  返回值类型：datetime

  示例：

```
select localtime(3);
      localtime(3)
-------------------------
 2024-02-26 15:14:2237
(1 row)
select localtime();
     localtime()
--------------------
 2024-02-26 15:14:35
(1 row)
```

- localtimestamp

  描述：返回语句开始执行的时间戳。

  返回值类型：datetime

示例：

```
select localtimestamp;
   localtimestamp
---------------------
 2024-02-26 15:14:59
(1 row)
```

- localtimestamp(n)

  描述：返回语句开始执行的时间戳，n 为精度，最大值取 6。

  返回值类型：datetime

  示例：

```
select localtimestamp(3);
    localtimestamp(3)
-------------------------
 2024-02-26 15:15:23.786
(1 row)
select localtimestamp();
  localtimestamp()
---------------------
 2024-02-26 15:15:38
(1 row)
```

- MAKEDATE()

  函数原型:

  DATE MAKEDATE(int8 year, int8 dayofyear)

功能描述:

给定年份和天数，返回该年份在此年份天数下日期值。

备注：

任一为 NULL，函数返回 NULL。

dayofyear 必须大于 0 否则返回 NULL。

0 <= year < 70 时：将 year 视作 20XX 年处理。 70 <= year < 100 时：将 year 视作 19XX 年处理。

返回结果限制在[0, 9999-12-31],超出范围返回 NULL。

示例:

```
SELECT MAKEDATE(2022,31), MAKEDATE(2022,32);
  makedate  |  makedate
------------+------------
2022-01-31 | 2022-02-01
(1 row)
-- 0<= year < 70 以及 70 <= year < 100
SELECT MAKEDATE(0,31), MAKEDATE(70,32);
 makedate  |  makedate
------------+------------
2000-01-31 | 1970-02-01
(1 row)
-- dayofyear <= 0 以及 超出范围 的情况
SELECT MAKEDATE(2022,0), MAKEDATE(9999,366);
 makedate | makedate
----------+----------
          |
(1 row)
```

- MAKETIME()

  函数原型:

  TIME MAKETIME(int8 hour, int8 minue, Numeric second)

功能描述:

给定小时、分钟和秒参数，返回 TIME 类型值。

备注:

当参数满足如下任一条件时，函数返回 NULL:

minue < 0 or minue >= 60

second < 0 or second >= 60

任一参数为 NULL

返回的 Time 结果保留 6 位小数，若 second 超出六位小数，则按照四舍五入进位。

返回 TIME 类型值要求在[-838:59:59, 838:59:59]中。若超出范围，则根据 hour 的正负类型，来返回指定的边界值。

示例:

```
SELECT MAKETIME(15, 15, 15.5);
 maketime
------------
 15:15:15.5
(1 row)
-- 四舍五入进位
SELECT MAKETIME(10, 15, 20.5000005);
    maketime
-----------------
 10:15:20.500001
(1 row)
-- 超出边界值
SELECT MAKETIME(839,0,0);
 maketime
-----------
 838:59:59
(1 row)
```

- microsecond(timestamp)

  描述：获取日期/时间值中微秒的值。

  返回值类型：double precision

  示例：

```
SELECT microsecond(timestamp '2001-02-16 20:38:40.123');
 date_part
-----------
    123000
(1 row)
```

- minute(timestamp)

  描述：获取日期/时间值中分钟的值。

  返回值类型：double precision

  示例：

```
SELECT minute(timestamp '2001-02-16 20:38:40.123');
 date_part
-----------
        38
(1 row)
```

- now(n)

  描述：返回语句开始执行的时间戳，n 为精度，最大值取 6。

  返回值类型：datetime

  示例：

```
select now(3);
         now(3)
-------------------------
 2024-02-26 15:17:05.573
(1 row)
select now();
        now()
--------------------
 2024-02-26 15:17:18
(1 row)
```

- PERIOD_ADD()

  函数原型:

  int8 PERIOD_ADD(int8 P, int8 N)

功能描述:

返回时期 P(格式为 YYYYMM 或 YYMM)加上 N 个月后的时期值，格式为 YYYYMM。

备注:

当任一参数为 NULL 时，函数返回 NULL。

P= 0 时，返回 0。

参数的时期 P 与返回结果的时期中的年份小于 100 时，会以 70 为边界，将年份转为 20XX 年或 19XX 年。

若入参为小数格式的字符串时，本函数会按照四舍五入进位转为整数后进行处理，而在 mysql 中，会将参数的小数部分舍去。例如:period_add('202104', '10.5')在 GBase 8c 中结果与 period_add(202101, 11)相同，而在 mysql 中，则视为 period_add(202101, 10)

示例:

```
SELECT PERIOD_ADD(202201, 2);
 period_add
------------
```

```
    202203
(1 row)

 -- p = 0
 SELECT PERIOD_ADD(0, 2);
 period_add
------------
          0
(1 row)
 -- 时期的年份处于[0,70) 或 [70, 100)范围内
 SELECT PERIOD_ADD(0101, 2), PERIOD_ADD(7001, 2);
 period_add | period_add
------------+------------
     200103 |     197003
(1 row)
```

- PERIOD_DIFF()

  函数原型:

  int8 PERIOD_DIFF(int8 P1, int8 P2)

功能描述:

返回两个时期参数 P1 与 P2 的月份数差值。

备注:

当任一参数为 NULL 时，函数返回 NULL。

时期参数 P1 和 P2 中的年份小于 100 时，会以 70 为边界，将年份转为 20XX 年或 19XX 年。

若入参为小数格式的字符串时，本函数会按照四舍五入进位转为整数后进行处理，而在 mysql 中，会将参数的小数部分舍去。例如:period_diff('202104', '202105')在 GBase 8c 中结果与 period_diff(202101, 202103)相同，而在 mysql 中，则被视为 period_diff(202101, 202102)。

示例:

```
SELECT PERIOD_DIFF(202201,202003);
 period_diff
-------------
          22
(1 row)
SELECT PERIOD_DIFF(0101,7001);
```

```
 period_diff
-------------
         372
(1 row)
```

- quarter(timestamp)

  描述：获取日期/时间值中的季度数，从 1 到 4。

  返回值类型：double precision

  示例：

```
SELECT quarter(timestamp '2001-02-16 20:38:40.123');
 date_part
-----------
         1
(1 row)
```

- second(timestamp)

  描述：获取日期/时间值中的秒数。

  返回值类型：double precision

  示例：

```
SELECT second(timestamp '2001-02-16 20:38:40.123');
 date_part
-----------
        40
(1 row)
```

- SEC_TO_TIME()

  函数原型:

  TIME SEC_TO_TIME(Numeric second)

功能描述:

给定秒数，将其转为小时、分钟与秒，返回 TIME 类型值。

备注:

当任一参数为 NULL，函数返回 NULL。

返回 TIME 类型值只保留小数点后 6 位，超出部分按照四舍五入规则进位。

返回 TIME 类型值要求在[-838:59:59, 838:59:59]中。若超出范围，则根据 second 的正负类型，来返回指定的边界值。

示例:

```
SELECT SEC_TO_TIME(4396);
 sec_to_time
-------------
 01:13:16
(1 row)
-- 四舍五入进位
 SELECT SEC_TO_TIME(2378.2222225);
   sec_to_time
-----------------
 00:39:38.222223
(1 row)
-- 返回结果超出范围
 SELECT SEC_TO_TIME(3888888);
 sec_to_time
-------------
 838:59:59
(1 row)
```

SUBDATE(expr, interval)

函数原型:

Text SUBDATE(text date, INTERVAL expr unit)

Text SUBDATE(text date, int8 days)

功能描述:

该函数执行日期运算。参数 date 指定开始 DATE 或 DATETIME 类型值。指定要从开始日期减去的 INTERVAL 值，返回相减后的结果日期值。若第二参数为整数，则将其视为减去的天数值。

备注:

函数返回格式为 DATE 或 DATETIME。一般情况下，返回类型与第一参数的类型相同。当第一参数的类型为 DATE 时且 INTERVAL 的单位包含 HOUR、MINUTE、SECOND 部分，则返回结果为 DATETIME。

本函数所支持的 interval 单位与 openguass 支持的 interval 单位保持一致，包括 YEAR，

MONTH，DAY，HOUR，MINUTE，SECOND，DAY TO HOUR，DAY TO MINUTE，DAY TO SECOND，HOUR TO MINUTE，HOUR TO SECOND，MINUTE TO SECOND。

任一参数为 NULL，函数返回 NULL。

在下列情况中，函数报错（该特性兼容此函数在 mysql 的 insert 语句中的行为）：

参数 date 的日期超出范围[0000-1-1, 9999-12-31]

当 interval 单位为仅与年或月相关时，返回结果的日期超出范围[0000-1-1, 9999-12-31]

其他 interval 单位，返回结果的日期超出范围[0001-1-1, 9999-12-31]

示例:

```
SELECT SUBDATE('2022-01-01', INTERVAL 31 DAY), SUBDATE('2022-01-01', 31);
 subdate    |  subdate
------------+------------
 2021-12-01 | 2021-12-01
(1 row)
-- 第一参数为 DATE
SELECT SUBDATE('2022-01-01', INTERVAL 1 YEAR);
 subdate
------------
 2021-01-01
(1 row)
-- 第一参数为 DATETIME
SELECT SUBDATE('2022-01-01 01:01:01', INTERVAL 1 YEAR);
       subdate
 ---------------------
 2021-01-01 01:01:01
(1 row)
-- 第一参数为 DATE 但是 INTERVAL 的单位包含 TIME 部分
SELECT SUBDATE('2022-01-01', INTERVAL 1 SECOND);
       subdate
---------------------
 2021-12-31 23:59:59
(1 row)
```

- SUBDATE(TIME, interval)

  函数原型:

  TIME SUBDATE(TIME time, INTERVAL expr unit)

  TIME SUBDATE(TIME time, int8 days)

功能描述:

该函数是为兼容 MySql 中 subdate 函数的第一参数类型可以为 TIME 的情况。该情况下，第一参数的输入必须为原始的 TIME 数据，而非由字符串的隐式转换而来。 参数 time 指定开始 TIME 类型的时间值，第二参数指定要从开始时间减去的 INTERVAL 值，返回相减后的结果日期值。若第二参数为整数，则将其视为减去的天数值。

备注:

第一参数必须为原始的 TIME 类型，而非由字符串的隐式转换而来。如 SUBDATE('1:1:1', 1)并不会进入此函数。需改为 SUBDATE(time'1:1:1', 1)。

在如下情况下，函数报错（该特性兼容此函数在 mysql 的 insert 语句中的行为）：

第二参数的 INTERVAL 单位包含年或月部分

函数返回 TIME 值超出[-838:59:59, 838:59:59]

示例:

```
SELECT SUBDATE(time'10:15:20', INTERVAL '1' DAY), SUBDATE(time'10:15:20', 1);
  subdate  |  subdate
-----------+-----------
-13:44:40 | -13:44:40
(1 row)

-- 第二参数的 INTERVAL 单位不能包含年或月部分
SELECT SUBDATE(time'838:00:00', INTERVAL '1' MONTH);
ERROR:  time field value out of range
CONTEXT:  referenced column: subdate

-- 结果超出范围
SELECT SUBDATE(time'838:59:59', INTERVAL '-1' SECOND);
ERROR:  time field value out of range
CONTEXT:  referenced column: subdate
```

- SUBTIME()

函数原型:

TEXT SUBTIME(TIME time1, TIME time2)

TEXT SUBTIME(DATETIME time1, TIME time2)

功能描述:

该函数执行日期运算，返回 DATETIME 或 TIME 表达式 time1 减去 TIME 表达式 time2 的结果。返回结果类型与 time1 传入类型保持一致。

备注:

在如下情况下，函数报错:

time1 为不为合法的 TIME 格式或者 DATETIME 格式

time2 为不为合法的 TIME 格式

返回 DATETIME 值时，结果超过[0000-01-01 00:00:00.000000, 9999-12-31 23:59:59.999999]

返回 TIME 值时，结果超过[-838:59:59, 838:59:59]

示例:

```
select subtime('11:22:33','10:20:30');
subtime
----------
01:02:03
(1 row)
select SUBTIME('2020-03-04 11:22:33', '-10:20:30');
    subtime
---------------------
2020-03-04 21:43:03
(1 row)
```

- sysdate(n)

描述：返回系统实时时间戳，n 为精度，最大值取 6。

返回值类型：datetime

示例:

```
select sysdate(3);    sysdate(3)
-------------------------
2022-07-21 17:38:23.442
(1 row)
select sysdate();    sysdate()
--------------------
2022-07-21 17:39:02
(1 row)
```

- time(expr)

功能描述:

参数指定一个 TIME 或 DATETIME 表达式 expr，提取其中的 time 表达式并返回为字符串。

备注:

返回的时间表达式中最多保留六位小数，并且不显示小数部分尾部的 0

对于异常的日期或时间格式或是域溢出的日期或时间(如 1:60:60、2022-12-32 等)，本函数兼容 mysql 中该函数于 insert 语句中的表现，即报错。

date 格式字符串将会报错，而 date 类型参数将会返回"00:00:00"。

示例:

```
select time('2022-1-1 1:1:1111116'), time('25:25:25');
      time       |   time
-----------------+----------
01:01:0111112 | 25:25:25
(1 row)
select time(date'2022-1-1');
  time
----------
00:00:00
(1 row)
```

- TIMEDIFF()

函数原型:

TIME TIMEDIFF(TIME time1, TIME time2)

TIME TIMEDIFF(DATETIME datetime1, DATETIME datetime2)

功能描述:

该函数执行 DATETIME 或 TIME 类型值间的减法运算，计算 DATETIME 或 TIME 间的时间差值，运算结果以 TIME 类型值返回。

备注:

time1, time2 所对应类型需要一致，否则返回 NULL。

若出现下列情况，函数报错（该特性兼容此函数在 mysql 的 insert 语句中的行为）：

TIME 类型入参超出[-838:59:59, 838:59:59]范围或格式不合法

DATETIME 类型入参超出[0000-01-01 00:00:00.000000, 9999-12-31 23:59:59.999999]范围或格式不合法

对于 DATETIME 格式的字符串入参，本函数支持 0 值日期，如'2000-0-1 1:1:1'

返回值超出[-838:59:59, 838:59:59]范围

示例:

```
select TIMEDIFF(time'23:59:59',time'01:01:01'), TIMEDIFF(datetime'2008-12-31
23:59:59',datetime'2008-12-30 01:01:01');
 timediff | timediff
----------+----------
 22:58:58 | 46:58:58
(1 row)

-- 两参数对应类型不一致
select timediff('2000-1-1 0:0:0', '0:0:0'), timediff(time'0:0:0',
datetime'2000-1-1 0:0:0');
 timediff | timediff
----------+----------
          |
(1 row)
-- 返回值超出范围时报错。
select timediff(time'-830:00:00', time'10:20:30');
ERROR:  time field value out of range
CONTEXT:  referenced column: timediff
```

- TIMESTAMP()

函数原型:

DATETIME TIMESTAMP(TEXT expr)

DATETIME TIMESTAMP(TEXT expr, TIME time)

功能描述:

只有一个参数时，函数将 DATE 或 DATETIME 表达式 expr 转为 DATETIME 值返回。

有两个参数时，函数计算 DATE 或 DATETIME 表达式 expr 加上 TIME 类型值 time 的结果并以 DATETIME 值返回。

备注:

expr 为不存在的 date 或 datetime 表达式时。如'2000-12-32'，'2000-1-1 24:00:00'等，函数

报错。

入参或返回值超出指定时间范围，函数报错

包含两个参数且第二参数 time 为非 TIME 格式字符串时，函数报错。

示例:

```
select TIMESTAMP('2022-01-01'), TIMESTAMP('20220101');
      timestamp       |      timestamp
----------------------+---------------------
2022-01-01 00:00:00 | 2022-01-01 00:00:00
(1 row)

select TIMESTAMP('2022-01-31 12:00:00.123456'),
TIMESTAMP('20000229120000.1234567');
        timestamp          |         timestamp
---------------------------+---------------------------
2022-01-31 12:00:00.123456 | 2000-02-29 12:00:00.123457
(1 row)
select TIMESTAMP('2022-01-31','12:00:00.123456'), TIMESTAMP('2022-01-31
12:00:00','-32:00:00');
        timestamp          |      timestamp
---------------------------+---------------------
2022-01-31 12:00:00.123456 | 2022-01-30 04:00:00
(1 row)

select TIMESTAMP('20000229','100:00:00'),
TIMESTAMP('20000229120000.123','100:00:00');
     timestamp       |       timestamp
---------------------+------------------------
 2000-03-04 04:00:00 | 2000-03-04 16:00:00.123
(1 row)
```

- timestampadd(unit, span, expr)

  函数原型:

  TEXT TIMESTAMPADD(text unit, interval span, text datetime)

功能描述:

将一段时间加到已知的时间点上。其中第一个参数 unit 是时间单位，第二个参数 span 是具体数值，第三个参数 datetime 是已知时间点。

备注：

unit 支持的单位如下：

单位	对应的输入
年	year, years, y, yr, yrs, yyyy
季度	quarter, qtr, q
月	month, months, mm, mon, mons
周	week, weeks, w
日	day, days, d, dd, ddd, j
时	hour, hours, h, hh, hr, hrs
分	minute, minutes, m, mi, min, mins
秒	second, seconds, s, sec, secs
微秒	microsecond, microsecon, us, usec, usecond, useconds, usecs

span 支持小数。当 unit 为秒时，span 根据第七位小数，四舍五入到六位小数；否则 span 四舍五入到整数。

datetime 的输入类型可以是字符串、date、datetime、time。

datetime 的输入范围和函数的计算结果均需落在区间[0001-01-01 00:00:00.000000, 9999-12-31 23:59:59.999999]内，否则报错(兼容 mysql 中该函数于 insert 语句的表现)。

对于异常的日期或时间格式或是域溢出的日期或时间(如 1:60:60、2022-12-32 等)，本函数兼容 mysql 中该函数于 insert 语句中的表现，即报错。

示例：

```
select timestampadd(day, 1, '2022-09-01');
 timestampadd
--------------
 2022-09-02
(1 row)
```

```
select timestampadd(hour, 1, '2022-09-01 08:00:00');
    timestampadd
---------------------
 2022-09-01 09:00:00
(1 row)
```

- time_format()

  函数原型:

  TEXT TIME_FORMAT(text time, text format)。

功能描述:

第一参数 time 为 time 或 datetime 表达式，函数基于第二参数 format 格式化 time 部分的值并返回为字符串。

备注:

格式	描述
%f	微秒（000000 至 999999）
%H	小时（00,24 小时格式，但能超出 23）
%h	小时（00 到 12）
%I	小时（00 到 12）
%i	分钟（00 至 59）
%p	AM or PM
%r	时间为 12 小时 AM 或 PM 格式（hh：mm：ss AM / PM）
%S	秒（00 到 59）
%s	秒（00 到 59）

%T	24 小时格式的时间(hh：mm：ss)，小时数能超出 23
%k	小时数（0,24 小时格式，但能超出 23)

format 支持的格式如下:

当第一参数 time 的 hour 部分超出[0, 23]的范围时，%H, %k 和%T 能够产生超出该范围的值，其余包含小时的 format 将转为取模 12 后的小时数。例如:

```
select time_format('100:59:59', '%H|%k|%T|%I');
    time_format
 ---------------------
100|100|100:59:59|04
(1 row)
```

对于非时分秒相关的格式，会返回 0 或 NULL，包括：

格式	返回结果
%a、%b、%D、%j、%M、%U、%u、%V、%v、%W、%w、%X、%x	NULL
%c 、%e	0
%d、%m、%y	00
%Y	0000

提取的时间值最多保留六位小数。

示例:

```
select TIME_FORMAT('83:59:59.0000009', '%T|%r|%H|%h|%I|%i|%S|%f|%p|%k');
                  time_format
-----------------------------------------------
83:59:59|11:59:59 AM|83|11|11|59|59|000001|AM|83
(1 row)
select TIME_FORMAT('2022-1-1 23:59:59.0000009',
'%T|%r|%H|%h|%I|%i|%S|%f|%p|%k');
                  time_format
-----------------------------------------------
23:59:59|11:59:59 PM|23|11|11|59|59|000001|PM|23
```

- weekday(timestamp)

描述：获取日期/时间值中是一周中的星期几（0 代表星期一，1 代表星期二，以此类推，6 代表星期日）。

返回值类型：double precision

示例：

```
SELECT weekday(timestamp '2001-02-16 20:38:40.123');
 ?column?
----------
        4
(1 row)
```

- weekofyear(timestamp)

  描述：获取日期/时间值中是一年的第几周。

  返回值类型：double precision

  示例：

```
SELECT weekofyear(timestamp '2001-02-16 20:38:40.123');
 date_part
-----------
        7
(1 row)
```

- year(timestamp)

  描述：获取日期/时间值中的年数。

  返回值类型：double precision

  示例：

```
SELECT year(timestamp '2001-02-16 20:38:40.123');
 year
-----
 2001
(1 row)
```

- current_date()

  描述：当前日期。

  返回值类型：date

  示例：

```
SELECT current_date;
    date
------------
 2017-09-01
(1 row)
```

- to_days(expr)

功能描述：接受一个 date 或 datetime 表达式作为参数，返回参数所指定的日期到 0000 年所经过的天数

返回类型：64 位整数 int8

备注:

若入参类型为 time 类型，将用于计算的日期将为当前日期加上 time 指定时间后得到的日期。

若输入日期超出[0000-01-01, 9999-12-31]的范围或入参为非法的 date 或 datetime 表达式，函数报错（兼容 mysql 中该函数于 insert 语句中的表现）。

示例:

```
select to_days('0000-01-01');
 to_days
---------
       1
(1 row)

select to_days('2022-09-05 23:59:59.5');
 to_days
---------
  738768
(1 row)

-- 当前日期为: 2022-09-05
select to_days(time'25:00:00');
 to_days
---------
  738769
(1 row)
```

- to_seconds(expr)

功能描述：参数输入一个 date 或 datetime 表达式 expr 用以指定时间点，返回 0000-01-01

00:00:00 到该时间点所经过的秒数

返回类型：64 位整数 int8

备注：

参数 datetime 支持的类型有：字符串、数值、date、datetime、time 等。当输入参数为 time 类型时，日期会被自动设置为当前日期。

返回的结果只保留整秒数，小数部分直接舍弃。

若输入日期超出[0000-01-01, 9999-12-31]的范围或入参为非法的 date 或 datetime 表达式，函数报错（兼容 mysql 中该函数于 insert 语句中的表现）。

示例：

```
select to_seconds('2022-09-01');
 to_seconds
-------------
 63829209600
(1 row)
select to_seconds('2022-09-01 12:30:30.888');
 to_seconds
-------------
 63829254630
(1 row)
select to_seconds(20220901123030);
 to_seconds
-------------
 63829254630
(1 row)
```

- unix_timestamp()

函数原型：

NUMERIC UNIX_TIMESTAMP()

NUMERIC UNIX_TIMESTAMP(text datetime)

功能描述：

不输入任何参数，直接执行该函数，返回 1970-01-01 00:00:00 UTC 到当前时间点所经过的秒数

输入一个时间点 datetime，返回 1970-01-01 00:00:00 UTC 到 datetime 所经过的秒数

备注：

参数 datetime 支持的类型有：字符串、数值、date、datetime、time 等。当输入参数为 time 类型时，日期会被自动设置为当前日期。

参数 datetime 的有效范围为 [1970-01-01 00:00:00.000000 UTC, 2038-01-19 03:14:07.999999 UTC]。

参数的输入范围会受到时区的影响，但最终计算结果不受时区影响。

计算结果最多只保留六位小数且不显示小数部分尾部的 0。

示例：

```
select unix_timestamp('2022-09-01');
 unix_timestamp
----------------
     1661961600
(1 row)
select unix_timestamp('2022-09-01 12:30:30.888');
 unix_timestamp
----------------
 1662006630.888
(1 row)
select unix_timestamp(20220901123030.6);
 unix_timestamp
----------------
   1662006630.6
(1 row)
```

- utc_date()

函数原型：DATE UTC_DATE()

功能描述：返回当前的 UTC 日期，类型为 DATE。

备注：

UTC_DATE 能够以关键词的形式识别，此时无需包含括号。

示例：

```
select UTC_DATE();
  utc_date
------------
 2024-02-26
```

```
(1 row)

select UTC_DATE;
  utc_date
------------
 2024-02-26
(1 row)
```

- utc_time()

  函数原型：

  TIME UTC_TIME()

  TIME UTC_TIME(int fsp)

功能描述：返回当前的 UTC 时间，类型为 TIME。若给定一个整数参数作为精度，则能够指定结果保留的小数数量，支持精度范围为[0-6]

  备注：

  UTC_TIME 能够以关键词的形式识别，此时无需包含括号。效果等同于无参数的 UTC_TIME()函数。

  返回的 TIME 结果不显示小数部分尾部的 0

  示例：

```
select UTC_TIME();
 utc_time
----------
 15:13:54
(1 row)
select UTC_TIME(6);
    utc_time
----------------
 15:13:56.59698
(1 row)
select UTC_TIME;
 utc_time
----------
 15:14:01
(1 row)
```

- utc_timestamp()

函数原型：

DATETIME UTC_TIMESTAMP()

DATETIME UTC_TIMESTAMP(int fsp)

功能描述：返回当前的 UTC 日期时间值，类型为 DATETIME。若给定一个整数参数作为精度，则能够指定结果保留的小数数量，支持精度范围为[0-6]。

备注:

UTC_TIMESTAMP 能够以关键词的形式识别，此时无需包含括号。效果等同于无参数的 UTC_TIMESTAMP()函数。

返回的 DATETIME 结果不显示小数部分尾部的 0

示例：

```
select UTC_TIMESTAMP();
    utc_timestamp
---------------------
 2022-09-06 15:16:28
(1 row)
select UTC_TIMESTAMP(6);
       utc_timestamp
----------------------------
 2022-09-06 15:16:34.691118
(1 row)
select UTC_TIMESTAMP;
    utc_timestamp
---------------------
 2022-09-06 15:16:39
```

- date_bool(date)

描述：根据日期值中的年数返回布尔型（为零时返回 false，否则返回 true）。

返回值类型：boolean

示例：

```
select time_bool('18:50:00');
 time_bool
-----------
 t
(1 row)
```

```
select time_bool('00:50:00');
 time_bool
-----------
 f
(1 row)
```

- time_bool(time)

  描述：根据时间值中的小时数返回布尔型（为零时返回 false，否则返回 true）。

  返回值类型：boolean

  示例：

```
select date_bool('2022-08-20');
 date_bool
-----------
 t
(1 row)
select date_bool('0000-08-20');
 date_bool
-----------
 f
(1 row)
```

- dayname(date)

  描述：返回日期对应的工作日，返回内容所在语言集受 GUC 参数 dolphin.lc_time_names 控制。

  返回值类型：text

  备注：此函数兼容 MySQL 插表时的严格模式和非严格模式表现。

  示例：

```
select dayname('2000-1-1');
dayname
----------
Saturday
(1 row)
alter system set dolphin.lc_time_names = 'zh_CN';ALTER SYSTEM SET
select dayname('2000-1-1');
dayname
---------
星期六
```

```
(1 row)
```

- monthname(date)

描述：返回日期对应月份的全称，返回内容所在语言集受 GUC 参数 dolphin.lc_time_names 控制。

返回值类型：text

备注：此函数兼容 MySQL 插表时的严格模式和非严格模式表现。

示例：

```
select monthname('2000-1-1');
monthname
-----------
January
(1 row)
alter system set dolphin.lc_time_names = 'zh_CN';ALTER SYSTEM SET
select monthname('2000-1-1');
monthname
-----------
一月
(1 row)
```

- time_to_sec(time)

描述：将时间转换为秒数。

返回值类型：integer

备注：

此函数兼容 MySQL 插表时的严格模式和非严格模式表现。

函数参数被当做 time 类型解析时，其参数约束范围为[-838:59:59, 838:59:59]，与 GBase 8c 中 time 类型插表约束相同。

示例：

```
select time_to_sec('838:59:59');time_to_sec
-------------
   3020399
(1 row)
select time_to_sec('-838:59:59');time_to_sec
-------------
   -3020399
```

```
(1 row)
```

- month(date)

  描述：返回日期中的月份。

  返回值类型：integer

  备注：此函数兼容 MySQL 插表时的严格模式和非严格模式表现。

  示例：

```
select month('2021-11-12');
month
-------
    11
    (1 row)
```

- day(date)

  描述：返回日期中的天数。

  返回值类型：integer

  备注：此函数兼容 MySQL 插表时的严格模式和非严格模式表现。

  示例：

```
select day('2021-11-12');
day
-----
12
(1 row)
```

- date(expr)

  描述：expr 识别为 date 或者 datetime 表达式，从 expr 中提取出日期部分。

  返回值类型：date

  备注：

  此函数兼容 MySQL 插表时的严格模式和非严格模式表现。

  只有当 dolphin.sql_mode 中不含'no_zero_date'时，DATE 函数才能接收'0000-00-00'输入，并且返回日期'0000-00-00'。

  示例：

```
select date('2021-11-12');
```

```
    date
------------
2021-11-12
(1 row)
select date('2021-11-12 23:59:59.9999999');
    date
------------
2021-11-13
(1 row)
```

- last_day(expr)

  描述：expr 识别为 date 或者 datetime，返回该月对应的最后一天的日期。

  返回值类型：date

  备注：

  此函数兼容 MySQL 插表时的严格模式和非严格模式表现。

  在 B 模式数据库中，当 GUC 参数 dolphin.b_compatibility_mode 为 true 时，此函数代替 GBase 8c 原有 last_day 函数。

  示例：

```
set dolphin.b_compatibility_mode = true;SET
select last_day('2021-1-30');
last_day
------------
2021-01-31
(1 row)
```

- week(date[,mode])

  描述：返回 date 参数代表的日期在一年中的第几周。mode 参数为可选整型参数，范围为[0,7]。无 mode 参数传入时，GUC 参数 default_week_format 会作为默认 mode 参数。

  mode 参数的各种取值及其意义如下：

mode	意义
0	Sunday 为一周的第一天；week 的取值范围为[0-53]；一年的第一周必须包含 Sunday
1	Monday 为一周的第一天；week 的取值范围为[0-53]；一年的第一周必须有大于

mode	意义
	等于 4 天在此年内
2	Sunday 为一周的第一天；week 的取值范围为[1-53]；一年的第一周必须包含 Sunday
3	Monday 为一周的第一天；week 的取值范围为[1-53]；一年的第一周必须有大于等于 4 天在此年内
4	Sunday 为一周的第一天；week 的取值范围为[0-53]；一年的第一周必须有大于等于 4 天在此年内
5	Monday 为一周的第一天；week 的取值范围为[0-53]；一年的第一周必须包含 Monday
6	Sunday 为一周的第一天；week 的取值范围为[1-53]；一年的第一周必须有大于等于 4 天在此年内
7	Monday 为一周的第一天；week 的取值范围为[1-53]；一年的第一周必须包含 Monday

对于一周的第一天，week 取值范围，判定一年第一周的条件的说明：

一周的第一天指一周开始的那一天，Monday 或者 Sunday 可能为一周的第一天。

week 取值范围指 WEEK 函数返回值的取值范围，有[0-53]和[1-53]两种取值范围。其中[0-53]中的 0 代表给定日期实际位于其所在年份上一年的最后一周内，但为了将返回结果与给定日期所在年份联系起来，故认为给定日期位于其所在年份的第零周（也即还未开始第一周）。若希望给定日期所在周数与其所在年份关系更紧密，则应该使用 0、1、4 或者 5 作为 mode 值，这样，当给定日期位于其所在年份上一年的最后一周时，WEEK 函数会返回 0。

判定一年第一周的条件指判定所给日期位于当前年的第一周的条件，一般而言只有日期位于年份的边界才会进行判定。此判定有两种方式，由 mode 参数决定使用哪种方式。

方式一：若 Monday 或者 Sunday 是一周的第一天，并且 Monday 或者 Sunday 在给定日期所在年内，则此周为日期所在年份的第一周。对应 mode 取值为 0、2、5 和 7。

方式二：若给定日期所在的周有大于等于 4 天位于日期所在年内，则此周为日期所在年份的第一周；否则此周为上一年的最后一周。对应 mode 取值为 1、3、4 和 6。

返回值类型：integer

备注：此函数兼容 MySQL 插表时的严格模式和非严格模式表现。

示例：

```
show dolphin.default_week_format;
dolphin.default_week_format
---------------------
0
(1 row)

-- 给定日期位于前一年的最后一周内，mode 为 0
select week('2000-1-1');
week
------
    0
(1 row)

alter system set dolphin.default_week_format = 2;
ALTER SYSTEM SET

-- 给定日期位于前一年的最后一周内，mode 为 2
select week('2000-1-1');
week
------
52
(1 row)
select week('2000-1-1', 2);
week
------
52
(1 row)
```

- yearweek(date[,mode])

描述：返回 date 参数代表的日期所在的年份和周。mode 为可选整型参数，取值范围为[0,7]。无 mode 参数传入时，0 会作为默认 mode 参数，GUC 参数 default_week_format 不会影响 yearweek 函数。mode 参数详细意义参见 week 函数。

yearweek 函数不会返回 0 周，即 week 取值范围始终为[1-53]，不受 mode 参数影响。

返回值类型：bigint

备注：此函数兼容 MySQL 插表时的严格模式和非严格模式表现。

示例：

```
select week('1987-01-01', 0);
week
------
    0
(1 row)
select yearweek('1987-01-01', 0);
yearweek
----------
198652
(1 row)
```

- datediff(expr1,expr2)

描述：expr1 和 expr2 可以是 date 或者 datetime，计算 expr1-expr2 代表的天数，只有 expr1 和 expr2 的日期部分参与计算。当输入参数不合法时，该函数返回 NULL。

返回值类型：integer（代表日期差值，单位是天）

备注：此函数兼容 MySQL 插表时的严格模式和非严格模式表现。

示例：

```
select datediff('2001-01-01','321-02-02');
  datediff
----------
  613576
(1 row)
```

- from_days(N)

描述：返回数值 N 代表的天数对应日期。

返回值类型：date

示例：

```
select from_days(365);
from_days
------------
0000-00-00
(1 row)
select from_days(366);
from_days
```

```
------------
0001-01-01
(1 row)
```

- timestampdiff(unit,datetime expr1,datetime expr2)

描述：函数返回两个日期参数 expr2 - expr1 的值，这两个参数都有可能是 datetime 或者是 date，如果参数是 date，则认为时间部分为 0。计算差值之后，将计算结果转换成指定单位显示。unit 有以下值：MICROSECOND , SECOND, MINUTE, HOUR, DAY, WEEK, MONTH, QUARTER, 或者是 YEAR。当输入参数不合法时，此函数返回 NULL。

返回值类型：bigint（代表以指定单位显示的差值）

备注：

此函数兼容 MySQL 插表时的严格模式和非严格模式表现。

在 B 模式数据库中，此函数在 GUC 参数 dolphin.b_compatibility_mode 为 true 时代替 GBase 8c 原有 timestampdiff 函数。

示例：

```
set dolphin.b_compatibility_mode = true;SET
select timestampdiff(SECOND,'2001-01-01 12:12:12','2001-01-01 12:12:11');
timestampdiff
---------------
            -1
(1 row)
select timestampdiff(MONTH,'2001-01-01 12:12:12','2001-02-01 12:12:12');
timestampdiff
---------------
            1
(1 row)
```

- convert_tz(datetime, from_tz, to_tz)

描述：将 datetime 从 from_tz 指定的时区转换到 to_tz 指定的时区。如果 datetime 从 from_tz 转换到 UTC 时区时范围超过[1970-01-01 00:00:0000000, 2038-01-19 03:14:07.999999],则不进行转换。参数无效时，函数返回 NULL。

返回值：datetime

备注：此函数兼容 MySQL 插表时的严格模式和非严格模式表现。

示例：

```
SELECT CONVERT_TZ('2004-01-01 12:00:00','GMT','MET');
      convert_tz
---------------------
  2004-01-01 13:00:00
(1 row)
```

- DATE_ADD(date/datetime/time, interval expr unit)

描述：该函数执行日期时间加法运算，返回 exrp1 加 expr2 的结果。expr1 可以为 date/datetime/time 类型的数据，expr2 代表 interval 值。expr1 为 time 类型的数据时，只能在显示声明参数类型为 time 时才能实现对 time 的加法。

返回值类型：

该函数返回值类型为 text，以便可以得到 date 或 datetime 等多种类型的结果格式。

若有函数结果参与计算需求，可以使用 cast 语句将函数结果转换为合适数据数据类型而后再参与计算，如：SELECT CAST(DATE_ADD('2021-11-12', INTERVAL 1 SECOND) AS DATETIME) + 1;

备注：

GBase 8c 目前 INTERVAL 后不支持运算表达式。

兼容 MySQL 插表时的严格模式和非严格模式表现。

一般情况下，返回结果格式与第一参数的类型相同。当第一参数的类型为 DATE 时且 INTERVAL 的单位包含 HOUR、MINUTE、SECOND 部分，则返回结果为 DATETIME 格式。

若计算结果为 datetime 并且在[0000-1-1 00:00:00.000000, 9999-12-31 23:59:59.999999]范围内，但小于'0001-1-1 00:00:00.000000'，GBase 8c 表现与 MySQL 一致：将日期部分置为'0000-00-00'，时间部分结果视具体计算结果而定。

示例：

```
SELECT DATE_ADD('2022-01-01', INTERVAL 31 DAY);
  date_add
------------
2022-02-01
(1 row)
SELECT DATE_ADD('2022-01-01 01:01:01', INTERVAL 1 YEAR);
    date_add
---------------------
2023-01-01 01:01:01
```

```
(1 row)
SELECT DATE_ADD('2022-01-01', INTERVAL 1 SECOND);
    date_add
---------------------
2022-01-01 00:00:01
(1 row)
```

- DATE_SUB(date/datetime/time, interval expr unit)

描述：该函数执行日期时间减法运算，返回 exrp1 减 expr2 的结果。expr1 可以为 date/datetime/time 类型的数据，expr2 代表 interval 值。expr1 为 time 类型的数据时，只能在显示声明参数类型为 time 时才能实现对 time 的减法。

返回值类型：

该函数返回值类型为 text，以便可以得到 date 或 datetime 等多种类型的结果格式。

若有函数结果参与计算需求，可以使用 cast 语句将函数结果转换为合适数据数据类型而后再参与计算，如：SELECT CAST(DATE_SUB('2021-11-12', INTERVAL 1 SECOND) AS DATETIME) + 1;

备注：

GBase 8c 目前 INTERVAL 后不支持运算表达式。

兼容 MySQL 插表时的严格模式和非严格模式表现。

一般情况下，返回结果格式与第一参数的类型相同。当第一参数的类型为 DATE 时且 INTERVAL 的单位包含 HOUR、MINUTE、SECOND 部分，则返回结果为 DATETIME 格式。

若计算结果为 datetime 并且在[0000-1-1 00:00:00.000000, 9999-12-31 23:59:59.999999]范围内，但小于'0001-1-1 00:00:00.000000'，GBase 8c 表现与 MySQL 一致：将日期部分置为'0000-00-00'，时间部分结果视具体计算结果而定。

示例：

```
SELECT DATE_SUB('2022-01-01', INTERVAL 31 DAY);
  date_sub
------------
2021-12-01
(1 row)
SELECT DATE_SUB('2022-01-01 01:01:01', INTERVAL 1 YEAR);
    date_sub
---------------------
```

```
2021-01-01 01:01:01
(1 row)
SELECT DATE_SUB('2022-01-01', INTERVAL 1 SECOND);
    date_sub
---------------------
2021-12-31 23:59:59
(1 row)
```

- ADDDATE(date/datetime/time, interval/days)

描述：该函数执行日期或时间加法运算。当第二参数为 interval 时，该函数表现与 DATE_ADD 函数相同，详细描述参见 DATE_ADD。当第二参数为整数时，此整数会被当作天数加在第一参数上。

备注：GBase 8c 目前 INTERVAL 后不支持运算表达式。

示例：

```
SELECT ADDDATE('2021-11-12', INTERVAL 1 SECOND);
      adddate
---------------------
2021-11-12 00:00:01
(1 row)
SELECT ADDDATE(time'12:12:12', INTERVAL 1 DAY);
adddate
----------
36:12:12
(1 row)
SELECT ADDDATE('2021-11-12', 1);
  adddate
------------
2021-11-13
(1 row)
SELECT ADDDATE(time'12:12:12', 1);
adddate
----------
36:12:12
(1 row)
```

- ADDTIME(datetime/time,time)

描述：该函数执行时间加法运算，返回 expr1 加上 expr2 的结果。expr1 可以为 datetime 或者 time 格式，expr2 只能为 time 格式。

返回值类型：

该函数返回值类型为 text，以便可以得到 time 或 datetime 的结果格式。

若有函数结果参与计算需求，可以使用 cast 语句将函数结果转换为合适数据数据类型而后再参与计算，如：SELECT CAST(ADDTIME('2021-11-12 11:11:11', '10:10:10') AS DATETIME) + 1;

备注：此函数兼容 MySQL 插表时的严格模式和非严格模式表现。

示例：

```
SELECT ADDTIME('11:22:33','10:20:30');
addtime
----------
21:43:03
(1 row)
SELECT ADDTIME('2020-03-04 11:22:33', '-10:20:30');
addtime
---------------------
2020-03-04 01:02:03
(1 row)
```

- get_format(expr1, expr2)

描述：expr1 可接收 date、datetime、timestamp、time 四种类型名字，expr2 可接收五种规格字符串：'EUR'|'USA'|'JIS'|'ISO'|'INTERNAL'。函数根据 expr1 和 expr2 返回对应类型的对应规格的字符串。

该函数返回值情况如下表所示：

类型	规格	返回值
DATE	'USA'	'%m.%d.%Y'
DATE	'JIS'	'%Y-%m-%d'
DATE	'ISO'	'%Y-%m-%d'
DATE	'EUR'	'%d.%m.%Y'
DATE	'INTERNAL'	'%Y%m%d'

类型	规格	返回值
DATETIME	'USA'	'%Y-%m-%d %H.%i.%s'
DATETIME	'JIS'	'%Y-%m-%d %H:%i:%s'
DATETIME	'ISO'	'%Y-%m-%d %H:%i:%s'
DATETIME	'EUR'	'%Y-%m-%d %H.%i.%s'
DATETIME	'INTERNAL'	'%Y%m%d%H%i%s'
TIMESTAMP	'USA'	'%Y-%m-%d %H.%i.%s'
TIMESTAMP	'JIS'	'%Y-%m-%d %H:%i:%s'
TIMESTAMP	'ISO'	'%Y-%m-%d %H:%i:%s'
TIMESTAMP	'EUR'	'%Y-%m-%d %H.%i.%s'
TIMESTAMP	'INTERNAL'	'%Y%m%d%H%i%s'
TIME	'USA'	'%h:%i:%s %p'
TIME	'JIS'	'%H:%i:%s'
TIME	'ISO'	'%H:%i:%s'
TIME	'EUR'	'%H.%i.%s'
TIME	'INTERNAL'	'%H%i%s'

返回值类型：text

示例：

```
select get_format(datetime, 'iso');
    get_format
-------------------
%Y-%m-%d %H:%i:%s
```

```
(1 row)
```

- extract(unit from expr)

  描述：从 expr 参数中提取由 unit 参数指定的部分。

  返回值：bigint

  备注：

  此函数兼容 MySQL 插表时的严格模式和非严格模式表现。

  在 B 模式数据库中，当 dolphin.b_compatibility_mode 为 true 时才会替代 GBase 8c 原有 extract 函数。

  函数参数被当做 time 类型解析时，其参数约束范围为[-838:59:59, 838:59:59]，与 GBase 8c 中 time 类型插表约束相同。

  expr 参数在解析时按照 unit 参数解析。当 unit 涉及 YEAR、WEEK、QUARTER、MONTH、DAY 时，expr 参数被解析为 date 或者 datetime；当 unit 只涉及 HOUR、MINUTE、SECOND、MICRESECOND 时，expr 参数被解析为 time。

  extract 函数可以提取复合 unit。

  复合 unit 有：DAY_HOUR，DAY_MINUTE，DAY_SECOND，DAY_MICROSECOND，HOUR_MINUTE，HOUR_SECOND，HOUR_MICROSECOND，MINUTE_SECOND，MINUTE_MICROSECOND，SECOND_MICROSECOND。

  对于复合 unit：DAY_HOUR，DAY_MINUTE，DAY_SECOND，DAY_MICROSECOND，由于这些 unit 中包含 DAY 部分，所以 GBase 8c 将 expr 当作 datetime 来解析。

  示例：

```
set dolphin.b_compatibility_mode = true;
SET
select extract(year from '2021-11-12 12:12:1000123');
extract
---------
    2021
(1 row)
select extract(day_microsecond from '2021-11-12 12:12:1000123');
    extract
----------------
12121212000123
(1 row)
```

```
select extract(hour_microsecond from '2021-11-12 12:12:1000123');
  extract
--------------
121212000123
(1 row)
set dolphin.b_compatibility_mode = false;
SET
```

- date_format(expr, format)

描述：expr 参数为输入的 date 或者 datetime 格式内容，该函数根据 format 参数格式化 expr 参数对应部分，format 参数取值如下表：

提取标志	意义	取值范围
%a	简写工作日名称	Sun…Sat
%b	简写月份名称	Jan…Dec
%c	数字形式的月份	0…12
%D	有后缀的月份中的天数	0th, 1st, 2nd, 3rd, …
%d	数字形式的月份中的天数	00…31
%e	数字形式的月份中的天数	0…31
%f	微秒	000000…999999
%H	小时	00…23
%h	小时	01…12
%I	小时	01…12
%i	分钟	00…59
%j	一年中的天数	001…366
%k	小时	0…23

提取标志	意义	取值范围
%l	小时	1…12
%M	月份全称	January…December
%m	数字形式的月份	00…12
%p	上午或者下午	AM 或者 PM
%r	范围为 12 小时的时间	%r 返回的时间格式为'hh:mm:ss AM'或者'hh:mm:ss PM'
%S	秒	00…59
%s	秒	00…59
%T	范围为 24 小时的时间	%T 返回的时间格式为'hh:mm:ss'
%U	日期在一年中对应的周数 (对应 WEEK 函数 mode 为 0 的情况)	00…53
%u	日期在一年中对应的周数 (对应 WEEK 函数 mode 为 1 的情况)	00…53
%V	日期在一年中对应的周数 (对应 WEEK 函数 mode 为 2 的情况)此标志应和'%X'一起使用	01…53
%v	日期在一年中对应的周数 (对应 WEEK 函数 mode 为 3 的情况)此标志应和'%X'一起使用	01…53
%W	工作日全称	Sunday…Saturday
%w	工作日索引	0=Sunday…6=Saturday
%X	日期所在周数对应的四位数字年份（计算方式为 Sunday 是一周的第一天），	…

提取标志	意义	取值范围
	此标志应和'%V'一起使用	
%x	日期所在周数对应的四位数字年份（计算方式为 Monday 是一周的第一天），此标志应和'%v'一起使用	…
%Y	四位数字年份	…
%y	两位数字年份	…
%%	'%'字面量	…
%x	未列出的字符 x	…

返回值：text

备注：此函数兼容 MySQL 插表时的严格模式和非严格模式表现。

示例：

```
-- 将日期格式化为指定内容
select date_format('2001-01-01 12:12:12','%Y %M %H');
date_format
 -----------------
2001 January 12
(1 row)

-- 将日期格式化为所在周数或者工作日
select date_format('2001-01-01 12:12:12','%V %v %U %u %W %w');
    date_format
    ---------------------
    53 01 00 01 Monday 1
    (1 row)
```

- from_unixtime(unix_timestamp[,format])

描述：第一个参数为数值格式的时间戳,代表距离'1970-01-01 00:00:00'UTC 的秒数；第二个参数为可选字符串参数。第二参数不传入时，函数返回'1970-01-01 00:00:00' UTC + unix_timestamp + 当前时区偏移对应的 datetime；当第二个参数给出时，函数会将 datetime 根据第二个参数进行格式化，格式化的方法与 date_format 函数相同。当 unix_timestamp 超

过最大时间戳范围后，函数返回 NULL。

返回值类型：

仅传入第一个参数时：datetime

传入两个参数时：text

示例：

```
select from_unixtime(1);
    from_unixtime
---------------------
1970-01-01 08:00:01
(1 row)
select from_unixtime(1,'%Y');
from_unixtime
 ---------------
1970
(1 row)
select from_unixtime(2147483647);
    from_unixtime
---------------------
2038-01-19 11:14:07
(1 row)
select from_unixtime(2147483648);
from_unixtime
---------------
(1 row)
```

- str_to_date(str, format)

描述：该函数是 date_format 函数的逆函数。函数会尝试将字符串 str 与字符串 format 匹配，并根据 format 中包含的标志来构造对应 date 格式、datetime 格式或者 time 格式的内容。

返回值类型：

该函数返回值类型为 text，以便可以得到 time、date 或 datetime 的结果格式。

当 format 中标志仅包含时间相关字符串'fHISThiklrs'中的字符时：time 格式内容

当 format 中标志仅包含日期相关字符串'MVUXYWabcjmvuxyw'中的字符时：date 格式内容

当 format 中标志为上述两种情况混合时：datetime 格式内容

若有函数结果参与计算需求，可以使用 cast 语句将函数结果转换为合适数据数据类型而后再参与计算，如：SELECT CAST(STR_TO_DATE('2021-11-12 12:12:12', '%Y-%m-%d %T') AS DATETIME) + 1;

备注：

此函数兼容 MySQL 插表时的严格模式和非严格模式表现。

只有当 dolphin.sql_mode 中不含'no_zero_date'时，才能构造出日期'0000-00-00'。

只有当 dolphin.sql_mode 中不含'no_zero_date'时，才能构造出单独的时间，如 select str_to_date('10', '%h');

示例：

```
-- 普通构造日期
select str_to_date('01,5,2013','%d,%m,%Y');
str_to_date
-------------
2013-05-01
(1 row)
-- 使用年份，周数，工作日构造日期
select str_to_date('200442 Monday', '%X%V %W');
str_to_date
-------------
2004-10-18
(1 row)
-- 使用年份，天数构造日期
select str_to_date('2004 100', '%Y %j');
str_to_date
-------------
2004-04-09
(1 row)
-- 构造时间
set dolphin.sql_mode =
'sql_mode_strict,sql_mode_full_group,pipes_as_concat,ansi_quotes';
SET
select str_to_date('1:12:12 pm', '%r');
str_to_date
-------------
13:12:12
```

```
(1 row)
```

- sleep(duration)

  描述：睡眠 duration 秒， 然后返回 0

  返回值：int

  备注：此函数和 mysql 略有不同，函数睡眠达到给定时间，则返回 0，如果中间发生中断，则返回 NULL。

  示例：

```
SELECT sleep(1);
sleep
 -------
    0
(1 row)
```

2.4.1.3.6 咨询锁函数

咨询锁函数用于管理咨询锁(AdvisoryLock)。

- pg_advisory_lock(keybigint)

  描述：获取会话级别的排它咨询锁。

  返回值类型：void

  备注：pg_advisory_lock 锁定应用程序定义的资源，该资源可以用一个 64 位或两个不重叠的 32 位键值标识。如果已经有另外的会话锁定了该资源，则该函数将阻塞到该资源可用为止。这个锁是排它的。多个锁定请求将会被压入栈中，因此，如果同一个资源被锁定了三次，它必须被解锁三次以将资源释放给其他会话使用。

- pg_advisory_lock(key1int,key2int)

  描述：获取会话级别的排它咨询锁。

  返回值类型：void

  备注：只允许 sysadmin 对键值对(65535,65535)加会话级别的排它咨询锁，普通用户无权限。

- pg_advisory_lock(int4,int4,Name)

  描述：获取指定数据库的排它咨询锁。

  返回值类型：void

- pg_advisory_lock_shared(keybigint)

  描述：获取会话级别的共享咨询锁。

  返回值类型：void

- pg_advisory_lock_shared(key1int,key2int)

  描述：获取会话级别的共享咨询锁。

  返回值类型：void

  备注：pg_advisory_lock_shared 类似于 pg_advisory_lock，不同之处仅在于共享锁会话可以和其他请求共享锁的会话共享资源，但排它锁除外。

- pg_advisory_unlock(keybigint)

  描述：释放会话级别的排它咨询锁。

  返回值类型：Boolean

- pg_advisory_unlock(key1int,key2int)

  描述：释放会话级别的排它咨询锁。

  返回值类型：Boolean

  备注：pg_advisory_unlock 释放先前取得的排它咨询锁。如果释放成功则返回 true。如果实际上并未持有指定的锁，将返回 false 并在服务器中产生一条 SQL 警告信息。

- pg_advisory_unlock(int4,int4,Name)

  描述：释放指定数据库上的排它咨询锁。

  返回值类型：Boolean

  备注：如果释放成功则返回 true；如果未持有锁，则返回 false。

- pg_advisory_unlock_shared(keybigint)

  描述：释放会话级别的共享咨询锁。

  返回值类型：Boolean

- pg_advisory_unlock_shared(key1int,key2int)

  描述：释放会话级别的共享咨询锁。

  返回值类型：Boolean

备注：pg_advisory_unlock_shared 类似于 pg_advisory_unlock，不同之处在于该函数释放的是共享咨询锁。

- pg_advisory_unlock_all()

  描述：释放当前会话持有的所有咨询锁。

  返回值类型：void

  备注：pg_advisory_unlock_all 将会释放当前会话持有的所有咨询锁，该函数在会话结束的时候被隐含调用，即使客户端异常地断开连接也是一样。

- pg_advisory_xact_lock(keybigint)

  描述：获取事务级别的排它咨询锁。

  返回值类型：void

- pg_advisory_xact_lock(key1int,key2int)

  描述：获取事务级别的排它咨询锁。

  返回值类型：void

  备注：pg_advisory_xact_lock 类似于 pg_advisory_lock，不同之处在于锁是自动在当前事务结束时释放，而且不能被显式的释放。只允许 sysadmin 对键值对(65535,65535)加事务级别的排它咨询锁，普通用户无权限。

- pg_advisory_xact_lock_shared(keybigint)

  描述：获取事务级别的共享咨询锁。

  返回值类型：void

- pg_advisory_xact_lock_shared(key1int,key2int)

  描述：获取事务级别的共享咨询锁。

  返回值类型：void

  备注：pg_advisory_xact_lock_shared 类似于 pg_advisory_lock_shared，不同之处在于锁是在当前事务结束时自动释放，而且不能被显式的释放。

- pg_try_advisory_lock(keybigint)

  描述：尝试获取会话级排它咨询锁。

  返回值类型：Boolean

备注：pg_try_advisory_lock 类似于 pg_advisory_lock，不同之处在于该函数不会阻塞以等待资源的释放。它要么立即获得锁并返回 true，要么返回 false 表示目前不能锁定。

- pg_try_advisory_lock(key1int,key2int)

  描述：尝试获取会话级排它咨询锁。

  返回值类型：Boolean

  备注：只允许 sysadmin 对键值对(65535,65535)加会话级别的排它咨询锁，普通用户无权限。

- pg_try_advisory_lock_shared(keybigint)

  描述：尝试获取会话级共享咨询锁。

  返回值类型：Boolean

- pg_try_advisory_lock_shared(key1int,key2int)

  描述：尝试获取会话级共享咨询锁。

  返回值类型：Boolean

  备注：pg_try_advisory_lock_shared 类似于 pg_try_advisory_lock，不同之处在于该函数尝试获得共享锁而不是排它锁。

- pg_try_advisory_xact_lock(keybigint)

  描述：尝试获取事务级别的排它咨询锁。

  返回值类型：Boolean

  描述：尝试获取事务级别的排它咨询锁。

  返回值类型：Boolean

- pg_try_advisory_xact_lock(key1int,key2int)

  描述：尝试获取事务级别的排它咨询锁。

  返回值类型：Boolean

  备注：pg_try_advisory_xact_lock 类似于 pg_try_advisory_lock，不同之处在于如果得到锁，在当前事务的结束时自动释放，而且不能被显式的释放。只允许 sysadmin 对键值对(65535,65535)加事务级别的排它咨询锁，普通用户无权限。

- pg_try_advisory_xact_lock_shared(keybigint)

描述：尝试获取事务级别的共享咨询锁。

返回值类型：Boolean

- pg_try_advisory_xact_lock_shared(key1int,key2int)

描述：尝试获取事务级别的共享咨询锁。

返回值类型：Boolean

备注：pg_try_advisory_xact_lock_shared 类似于 pg_try_advisory_lock_shared，不同之处在于如果得到锁，在当前事务结束时自动释放，而且不能被显式的释放。

- lock_cluster_ddl()

描述：尝试对 GBase8c 内所有存活的数据库主节点获取会话级别的排他咨询锁。返回值类型：Boolean

备注：只允许 sysadmin 调用，普通用户无权限。

- unlock_cluster_ddl()

描述：尝试对数据库主节点会话级别的排他咨询锁。

返回值类型：Boolean

- pg_catalog.get_lock(text,text)

描述：用指定的字符串在数据库加用户锁，第二个参数是加锁等待时间。

返回值类型：Int

- pg_catalog.get_lock(text,double)

描述：用指定的字符串在数据库加用户锁，第二个参数是加锁等待时间。

返回值类型：Int

- pg_catalog.get_lock(text)

描述：用指定的字符串在数据库加用户锁。

返回值类型：Int

- pg_catalog.release_lock(text)

描述：释放指定的锁，如果释放成功，返回 1，如果当前会话并未持有指定锁，返回 0，如果当前锁并不存在(锁必须有人持有才会出现)，返回 NULL。

返回值类型：Int

- pg_catalog.is_free_lock(text)

描述：检查字符串是否空闲，如果没有被加锁返回 1，否则返回 0，如果检查期间出现其他错误，返回 NULL。

返回值类型：Int

- pg_catalog.is_used_lock(text)

描述：检查字符串的锁被谁持有，返回对应的用户的会话 id，如果指定锁无人持有，返回

NULL。

返回值类型：Bigint

- pg_catalog.clear_all_invalid_locks()

描述：清除 locknameHASH 表中无效锁的信息，返回清除的锁的数量。

返回值类型：Bigint

- pg_catalog.release_all_locks()

描述：释放当前会话持有的所有的锁，返回释放的次数(对于单个字符串持有多个的情况，按对应数字计算而非只计算一次)。

返回值类型：Bigint

描述:查询当前库中的所有用户锁,以记录的形式返回所有用户锁的名字和持有人信息。

返回值类型：Record

2.4.1.3.7 网络地址函数和操作符

GBase8c 提供网络相关的功能函数。

- is_ipv4(string)

描述：判断是否为 ipv4 地址。

返回值类型：int

示例：

```
Select is_ipv4('19168.0.1');
is_ipv4
```

```
---------
1
(1row)
Select is_ipv4('19168.0.1'::inet);
is_ipv4
---------
1
(1row)
```

- is_ipv6(string)

  描述：判断是否为 ipv6 地址。

  返回值类型：int

  示例：

```
Select is_ipv6('2403:A200:A200:0:AFFF::3');
is_ipv6
---------
1
(1row)
Select is_ipv6('2403:A200:A200:0:AFFF::3'::inet);
is_ipv6
---------
1
(1row)
```

- inet_aton(text)

  描述：此函数将 IPv4 地址的点分十进制表示形式作为字符串，并以整数形式返回给定 IP 地址的数值。如果输入地址不是有效的 IPv4 地址或是无法识别的表达式，则此函数返回 NULL。

  返回值类型：int

  示例：

```
SELECT INET_ATON('10.0.5.9');
inet_aton
-----------
167773449
(1row)
```

- inet_ntoa(int)

描述:此函数将以整数呈现的以网络字节序给出的网络主机地址转换成以点分十进制表示的字符串(如 127.0.0.1)。如果输入整数无法转换成有效的地址，则此函数返回 NULL。

返回值类型：text

示例:

```
SELECT INET_NTOA(167773449);
inet_ntoa
-----------
10.0.5.9
(1row)
```

- inet6_aton(text)

描述：此函数将给定 IPv6 或 IPv4 网络地址作为字符串，返回一个二进制字符串，该字符串表示网络字节顺序(大端)地址的数值。

因为数字格式的 IPv6 地址需要比最大整数类型更多的字节，所以此函数返回的值具有两种长度：针对 IPv6 地址的返回的字符串长度为 16 位

针对 IPv4 地址的返回的字符串长度为 4 位。

如果参数不是有效地址，或者为 NULL，则 INET6_ATON()返回 NULL。

返回值类型：bytea

示例:

```
SELECT HEX(INET6_ATON('fdfe::5a55:caff:fefa:9089'));
hex
--------------------------------
fdfe0000000000005a55cafffefa9089
(1row)
SELECT HEX(INET6_ATON('10.0.5.9'));
hex
----------
0a000509
(1row)
```

- inet6_ntoa(bytea)

描述：MySQL 中的此函数将给定的以数字形式表示为二进制字符串的 IPv6 或 IPv4 网络地址，将地址的以分隔符分隔的字符串形式返回。如果参数不是有效地址，或者它为 NULL，则 INET6_NTOA()返回 NULL。

返回值类型：text

示例：

```
SELECT INET6_NTOA(INET6_ATON('fdfe::5a55:caff:fefa:9089'));inet6_ntoa
----------------------------
fdfe::5a55:caff:fefa:9089
(1row)
SELECT INET6_NTOA(INET6_ATON('10.0.5.9'));
inet6_ntoa
------------
10.0.5.9
(1row)
SELECT IS_IPV4_COMPAT(INET6_ATON('::10.0.5.9'));
is_ipv4_compat
----------------
1
(1row)
SELECT IS_IPV4_COMPAT(INET6_ATON('::ffff:10.0.5.9'));
is_ipv4_compat
----------------
0
(1row)
```

- is_ipv4_compat(bytea)

描述：该函数接受一个以数字形式表示的二进制字符串的IPv6地址，如同INET6_ATON()返回值。如果参数是有效的兼容 IPv4 的 IPv6 地址，则返回 1，否则返回 0。IPv4 兼容地址的格式为::ipv4_address。

返回值类型：int

示例：

```
SELECT IS_IPV4_MAPPED(INET6_ATON('::10.0.5.9'));
is_ipv4_mapped
----------------
0
(1row)
SELECT IS_IPV4_MAPPED(INET6_ATON('::ffff:10.0.5.9'));
is_ipv4_mapped
----------------
1
(1row)
```

2.4.1.3.8 条件表达式函数

注意事项

本章节仅包含 dolphin 新增的条件表达式函数。

条件表达式函数

- if(bool,expr1,expr2)

  描述：条件判断函数。若 bool 值为 true，则返回 expr1；若 bool 值为 false，则返回 expr2

  示例：

```
Select if(true,1,2);
case
------
1
(1row)

Select if(false,1,2);
case
------
2
(1row)
```

- ifnull(expr1,expr2)

  描述：

  如果 expr1 为 NULL，则返回 expr2。

  如果 expr1 非 NULL，则返回 expr1。

  示例：

```
SELECT ifnull('hello','world');
Nvl
-------
hello
(1 row)
```

  备注：参数转换逻辑与 nvl 一致。

- isnull(expr)

  如果 expr 为 NULL，则返回 true。

如果 expr 非 NULL，则返回 false。

示例：

备注：判空逻辑与 exprisnull 一致。

- interval(base_expr,expr1,expr2,...,exprn)

描述：

将 base_expr 与后面的 expr(n)逐一比较，直到 expr(n)大于 base_expr，返回 value(n-1)；如果 expr(n)均小于等于 base_expr，则返回 value(n)。

如果 base_expr 或者 expr(n)为非数值数据：

BOOL 型：TRUE 转为 1，FALSE 转为 0；

能将其截断为 float8 形式浮点数，则将其截断为 float8；

不能截断为浮点数 float8 形式，则视为 0。

示例：

```
SELECT interval(5,2,3,4,6,7);
interval
----------
3
(1 row)

SELECT interval(false,-1,0,true,2);
 interval
----------
        2
(1 row)

SELECT interval('2022-12-12'::timestamp,'asdf','2020-12-12'::date,2023);
 interval
----------
        2
(1 row)
```

- strcmp(str1,str2)

描述：将 str1 与 str2 作比较(两个字符串自左向右逐个字符比较)，若 str1=str2，则返回 0；若 str1>str2，则返回 1；若 str1<str2，则返回-1。

示例

```
SELECT strcmp('asd','asd');
 strcmp
--------
      0
(1 row)

SELECT strcmp(312,311);
 strcmp
--------
      1
(1 row)

SELECT strcmp('2021-12-12'::timestamp,20210::float8);
 strcmp
--------
     -1
(1 row)
```

2.4.1.3.9 聚集函数

- any_value(expression)

  描述：所有输入行的任意一条 expression(默认第一条)。

  参数类型：任意 set、数值、字符串、日期/时间类型等。

  返回类型：与参数数据类型相同

  示例：

```
create table test_any_value(a int, b int);
CREATE TABLE
insert into test_any_value values(1,1),(2,1),(3,2),(4,2);
INSERT 0 4
select any_value(a), b from test_any_value group by b;
any_value | b
-----------+---
        1 | 1
        3 | 2
(2 rows)
```

- default(column_name)

  描述：获取表字段的默认值输出。

返回类型：text

示例：

```
create database test dbcompatibility 'B';
\c test

CREATE TABLE TEST(id int default 100, stime timestamp default now());
insert into test values(1, now());

select default(id) from test;
mode_b_default
---------------
            100
(1 row)

select default(stime) from test;
mode_b_default
---------------

(1 row)

insert into test values(default(id) + 10);
INSERT 0 1
update test set id = default(id) - 10;
UPDATE 2
delete from test where id = default(id) - 10;
DELETE 2
```

2.4.1.3.10 系统信息函数

- database()

  描述：当前数据库模式的名称。

  返回值类型：name

  示例：

```
SELECT database();
 database
----------
 public
(1 row)
```

备注：database 返回在搜索路径中第一个顺位有效的模式名。(如果搜索路径为空则返回 NULL，没有有效的模式名也返回 NULL)。如果创建表或者其他命名对象时没有声明目标模式，则将使用这些对象的模式。

- uuid_short()

  描述：当前数据库的 uuid_short 信息。

  返回值类型：int

  示例：

```
select uuid_short();
     uuid_short
--------------------
 4616404363533877248
(1 row)
```

- dolphin_version()

  描述：dolphin 版本信息。返回一个描述 dolphin 插件版本信息的字符串。

  返回值类型：text

  示例：

```
SELECT dolphin_version();
    dolphin_version
-----------------------
 dolphin
(1 row)
```

- dolphin_types()

  描述：dolphin 新增类型信息。返回一个描述 dolphin 新增类型信息的二维字符串数组，每个数组内的信息依次是：类型名、是否支持精度、是否支持范围。

  返回值类型：text[][]

  示例：

```
SELECT dolphin_types();
                                    dolphin_types
-------------------------------------------------------------------------------
---
-------------------------------------------------------------------------------
---
```

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

{{uint1,false,false},{uint2,false,false},{uint4,false,false},{uint8,false,fal
se
},{year,true,false},{binary,true,false},{varbinary,true,false},{tinyblob,fals
e,f
alse},{mediumblob,false,false},{longblob,false,false},{set,false,false},{enum,
fa
lse,false}}
(1 row)
```

2.4.1.3.11 逻辑操作符

相比于 GBase8c 原生语法，dolphin 新增了两个逻辑操作符：

(1) 新增&&操作符。

(2) 新增||操作符。

- &&

描述：当 dolphin.b_compatibility_mode 为 TRUE 时代表逻辑与运算，支持的类型包含 boolean 型、时间型、日期型、整型、浮点型、位串型、字符型。真值表如下：

a	b	a&&b 的结果
TRUE	TRUE	TRUE
TRUE	FALSE	FALSE
TRUE	NULL	NULL
FLASE	FALSE	FALSE
FALSE	NULL	FALSE
NULL	NULL	NULL

对于不同输入类型的处理如下表：

数据类型	处理方式

布尔型	按照真值表进行逻辑与运算。
整型	仅将零转换为布尔假，其余值均转换为布尔真，再进行逻辑与运算。
浮点型	仅将零转换为布尔假，其余值均转换为布尔真，再进行逻辑与运算。
位串型	仅将全零转换为布尔假，其余值均转换为布尔真，再进行逻辑与运算。
时间型	时间类型的转换方式仅依赖于‘小时’部分，当为输入‘00：xx：xx’时，时间类型均转换成为布尔假；当输入为‘yy：xx：xx’，且 yy 不为零时，时间类型均转换为布尔真。再进行逻辑与运算。
日期型	日期类型的转换方式仅依赖于‘年’部分，当为输入‘0000-xx-xx’时，日期类型均转换成为布尔假；当输入为‘yyyy-xx-xx’，且 yy 不为零时，日期类型均转换为布尔真。再进行逻辑与运算。
字符型	字符型的转换根据字符型的首部是否为数值，若不为数值，则直接转换成为布尔假；若为数值，则等于 0 的数值转换为布尔假否则转换为布尔真，然后再进行逻辑与运算。

返回值类型：boolean

- ||

描述：当 sql_mode 参数不为'pipes_as_concat'时代表逻辑或运算，支持的类型包含

boolean 型、时间型、日期型、整型、浮点型、位串型、字符型。真值表如下：

a	**b**	**a||b 的结果**
TRUE	TRUE	TRUE
TRUE	FALSE	TRUE
TRUE	NULL	TRUE
FLASE	FALSE	FALSE
FALSE	NULL	NULL
NULL	NULL	NULL

对于不同输入类型的处理如下表：

数据类型	**处理方式**

布尔型	按照真值表进行逻辑或运算。
整型	仅将零转换为布尔假，其余值均转换为布尔真，再进行逻辑或运算。
浮点型	仅将零转换为布尔假，其余值均转换为布尔真，再进行逻辑或运算。
位串型	仅将全零转换为布尔假，其余值均转换为布尔真，再进行逻辑或运算。
时间型	时间类型的转换方式仅依赖于‘小时’部分，当为输入‘00：xx：xx’时，时间类型均转换成为布尔假；当输入为‘yy：xx：xx’，且 yy 不为零时，时间类型均转换为布尔真。再进行逻辑或运算。
日期型	日期类型的转换方式仅依赖于‘年’部分，当为输入‘0000-xx-xx’时，日期类型均转换成为布尔假；当输入为‘yyyy-xx-xx’，且 yy 不为零时，日期类型均转换为布尔真。再进行逻辑或运算。
字符型	字符型的转换根据字符型的首部是否为数值，若不为数值，则直接转换成为布尔假；若为数值，则等于 0 的数值转换为布尔假否则转换为布尔真，然后再进行逻辑或运算。

返回值类型：boolean

2.4.1.3.12 位串操作函数和操作符

相比于 GBase8c 原生语法，dolphin 对于位串函数的修改主要为:

(1) 新增 bit_bool 函数。

(2) 新增^操作符。

(3) 新增 bit_count 函数。

(4) 新增 bit_xor 函数。

- bit_bool(bit)

  描述：根据位串中的数据返回布尔型(全部为零时返回 false，否则返回 true)。

  返回值类型：boolean

  示例：

```
select bit_bool('11111');
 bit_bool
----------
 t
(1 row)
```

```
select bit_bool('00001');
 bit_bool
----------
 t
(1 row)

select bit_bool('00000');
 bit_bool
----------
 f
(1 row)
```

- ^

  描述：实现 bit 类型数据的按位异或

  返回值类型：bit

  示例：

```
select b'1001'^b'1100';
 ?column?
----------
 0101
(1 row)
```

- bit_count(N)

  描述：返回输入数据转为无符号 64 位整数时二进制字符串中位为 1 的数目

  输入类型：numerictextbit

  返回值类型：text

  注意:当输入的数字或字符串数值超过无符号 64 位整数范围，返回结果统一为 64；当输入的 bit 超过 64 个 1 时，返回结果为

  示例：

```
SELECT bit_count(29);
 bit_count
----------
 4
(1 row)
```

```
SELECT bit_count(b'101010');
 bit_count
-----------
 3
(1 row)
```

- bit_count(expr)

  描述：返回表中所有列的值异或为无符号 64 位整数时的结果

  输入类型：expr

  返回值类型：unsignedint

  注意:当表中的值的数字或字符串数值为负数时，会自动转换成无符号的数字.

  示例：

```
INSERT INTO bit_xor_int4(a) values(-1.11);
INSERT 0 1
SELECT BIT_XOR(a) from bit_xor_int4;
bit_xor
---------
1
(1 row)

INSERT INTO bit_xor_int4(a) values(1.11);
INSERT 0 1
SELECT BIT_XOR(a) from bit_xor_int4;
bit_xor
---------
0
(1 row)
```

2.4.1.3.13 JSON-JSONB 函数和操作符

在 public 中创建同名自定义函数或存储过程可能影响原函数功能，建议用户创建同名自定义函数或存储过程时指定模式名。

- json_array([val[,val]...])

  描述：输入可变长参数，输出一个 JSON 数组。

  返回类型：array-json

  示例：

- json_object([key,val[,key,val]...])

描述：输入参数为交替出现的 key、value。从一个可变参数列表构造出一个 JSON 对象，使用前需设置 GUC 参数 dolphin.b_compatibility_mode=1。

返回类型：json

备注：

由于 JSON 对象中的所有键为字符串，因此 JSON_OBJECT()会将不是字符串类型的 key 转为字符串类型。为了保证程序的稳定性，我们一般使用字符串类型的 key。

key 不能为 NULL 且入参个数应为偶数个。

- json_quote(string)

描述：输入字符串，输出 JSON 文档，并用双引号修饰。

返回类型：json

备注：

在 GBase8c 中，通过入参前加 E 来实现转译功能，转译前后与 Mysql 一致。在 GBase8c 中，json_quote 函数支持数值型。

- json_contains(target,candidate[,path])

描述：path 可选参数是 target 参数的 path，返回 target 参数是否包含 candidate 参数。返回类型：bool

备注：

当 path 在 target 中不存在时，函数返回 NULL。

- json_contains_path(json_doc,one_or_all,path[,path]...)

描述：返回目标 json_doc 参数是否存在输入的 path 参数，one_or_all 参数选择模式。返回类型：bool

备注：

- one_or_all 参数可以为 one 或 all。

one 则只要一个 path 存在即返回 true，否则返回 false；all 则全部 path 存在才返回 true，否则返回 false。

若 one_or_all 参数为 one，则按顺序检查 path，NULL 的 path 先于任一存在的 path，

则函数返回 NULL；

若 one_or_all 参数为 all，则按顺序检查 path，NULL 的 path 先于任一不存在的 path，则函数返回 NULL。

- json_extract(json_doc,path[,path]...)

  描述：在 JSON 文档提取路径表达式指定的数据并返回。

  返回类型：json

- json_unquote(json_val)

  描述：去除文本中的引号，对转义符有所处理，或者舍弃 JSON 值中的引号。

  返回类型：text

- json_unquote(json_extract(column,path))

  描述：去除文本中的引号，对转义符有所处理，或者舍弃 JSON 值中的引号。

  返回类型：text

- json_keys(json_doc[,path])

  描述：将 JSON 对象的顶级值中的键作为 JSON 数组返回，如果给定了路径参数，则返回路径所指示 JSON 对象的顶级键。

  返回类型：json

- json_search(json_doc，one_or_all，search_str[，escape_char[，path]...])

  描述：可传入一个或多个路径参数，根据转义符和 one_or_all 模式，返回目标字符串在路径限定下对应目标文件中的所在位置。

  返回类型：text

  备注：

  转义符如果为 boolean 型相当于 NULL，默认"\"为转义符，可以直接输入个位数整型作为转义符。

  目标 json 文件和目标字符串不能为空，路径不能含空，如果 path 不存在则返回空。one_or_all 只能输入 one 或 all。

  search_str 可以直接输入整型、浮点型、boolean 型进行匹配，但是只能匹配目标文件中的字符串。

search_str 可以使用模糊匹配，path 中可以使用通配符进行匹配。

- json_array_append(json,path,value[,path2,value2]...)

描述：用来修改 JSON 文档，它向指定的数组节点中追加一个元素，并返回修改后的 JSON 文档。

返回类型：json

- json_append(json_doc,path,val[,path,val]...)

描述：功能同 json_array_append 函数。

返回类型：json

备注：函数可能在将来的版本中被删除，推荐使用 json_array_append 函数。

- json_array_insert(json,path,value[,path2,value2]...)

描述：函数用来修改 JSON 文档，它向 JSON 文档中的指定的数组中的指定位置插入一个值并返回新的 JSON 文档。

返回类型：json

备注：

如果路径表达式指示的数组元素超过了数组的长度，那么新元素将插入到数组的尾部。如果 JSON 文档或者路径为 NULL，此函数将返回 NULL。

- json_insert(json_doc,path,val[,path,val]...)

描述：向一个 JSON 文档中插入数据并返回新的 JSON 文档。

返回类型：json

备注：当 JSON 文档或 path 为空时，返回空。

- json_merge(json_doc,json_doc[,json_doc]...)

描述：功能同 json_merge_preserve 函数。

返回类型：json

备注：函数可能在将来的版本中被删除，推荐使用 json_merge_preserve 函数。

- json_merge_preserve(json_doc,json_doc[,json_doc]...)

描述：合并两个及以上的 JSON，相同键值合并为一个数组。

返回类型：json

备注：

如果任何参数为 NULL 则返回 NULL。

合并规范：

若相邻的两个 JSON 参数一个为 scalar 或对象，一个为数组。将 scalar 或对象，作为数组元素，按照参数的先后顺序，加入数组参数中，合并为单个数组。

若相邻的两个 JSON 参数都是 scalar 或对象。将 scalar 或对象按照参数的先后顺序，合并为单个数组。

示例：

若相邻的两个 JSON 参数都是数组。将两个数组的各个元素，按照参数的先后顺序，合并为单个数组。

若相邻的两个 JSON 参数都是对象。将两个对象的各个成员，按照 key 的顺序，合并为单个对象。

合并 JSON 后的对象成员返回值，全部符合 key 的顺序。

- json_merge_patch(json_doc,json_doc[,json_doc]...)

  描述：合并两个及以上的 JSON，相同键值保留后者 JSON 对象键值成员。

  返回类型：json

  备注：

  若任一参数为 NULL，则之前的参数和该参数的合并结果为 NULL。

  NULL 参数后面一个参数若非 NULL，则：

  后面参数为数组、scalar，合并结果为后面参数本身。

  后面参数为对象，合并结果为 NULL。

  合并规范：

- 若相邻的两个 JSON 参数都是对象，则合并结果为单个对象。
- 若一个 JSON 对象的某一成员键在另一个 JSON 对象中没有重复，则在合并结果中保留该成员。
- 若前一个 JSON 对象的某一成员键在后一个 JSON 对象中重复，则在合并结果中，保留

后者 JSON 对象中重复键成员。特别地，当后一个相同键对应对象成员的 value 为 NULL 时，在结果中删除该键成员。

示例：

若相邻的两个 JSON 参数存在一个参数不是对象，则合并的结果直接为第二个 JSON 参数。

若任一参数为 NULL，则位于该参数之前的参数和该参数的合并结果为 NULL

NULL 参数后面一个参数若非 NULL，则：

- 后者参数为数组或 scalar，合并结果为后面参数本身。
- 后者参数为对象，合并结果为 NULL。
- json_remove(json,path[,path]...)

描述：从一个 JSON 文档中删除由路径指定的 JSON 对象并返回修改后的 JSON 文档。返回类型：json

备注：

可以通过参数提供多个路径表达式以供删除。多个路径参数会从左到右依次被执行。当执行下一个参数的时候，JSON 文档可能已经发生了变化。

如果 JSON 中不存在指定的路径，此函数返回原文档。

如果 JSON 文档或者路径为 NULL，此函数将返回 NULL。

- json_replace(json_doc,path,val[,path,val]...)

描述：在一个 JSON 文档中替换已存在的数据并返回新的 JSON 文档。第一个参数为 JSON 文档，其后为交替出现的路径和替换值。

返回类型：json

- json_set(json_doc,path,val[,path,val]...))

描述：输入 JSON 文档，路径和键值，替换 JSON 文档中已有路径对应的键值，对于新增路径，插入对应键值。

返回类型：json

- json_depth(json)

描述：返回 JSON 文档的最大深度。

返回类型：integer

备注：

空数组、空对象或标量值的深度为 1。

仅包含深度为 1 的数组或对象深度为 2。

JSON 节点的最大深度等于其所有子节点最大深度的最大值。

- json_length(json_doc[,path])

  描述：输出 JSON 长度，如果有路径，则输出该路径对应文档的长度。

  返回类型：integer

  备注：

  路径不能含有通配符*，并且只能有一个路径。

- json_valid(val)

  描述：判断输入文本是否是合法的 JSON。

  返回类型：bool

  备注：

  若输入的 val 参数是 L2ON 类型，该函数返回 true。

  若所输入字符串需要转义，在单引号前加 E，字符串转义语法是(E'...')。

- json_pretty(json)

  描述：格式化输出一个 JSON 文档，以便更易于阅读。

  返回类型：json

- json_storage_size(json)

  描述：JSON_STORAGE_SIZE()函数返回存储一个 JSON 文档的二进制表示所占用的字节数。返回类型：integer

  备注：

  json 是必需的。一个 JSON 文档。它可以是一个 JSON 字符串，或者一个 JSON 列。

  根据实际存储方式的差异，调用 GBase8c 内部函数计算 json 在 GBase8c 中的具体存储大小，其结果与 mysql 不同。

- json_arrayagg(col_or_expr)

  描述：json_arrayagg 函数返回一个 JSON_ARRAY 型数组，它将指定列中的值聚合。

  备注：

  如果结果集没有任何行，此函数将返回 NULL。

- json_objectagg(key,value)

  描述：将由第一个参数作为键和第二个参数作为值的键值对聚合为一个 JSON 对象。

  返回类型：object-json

  备注：

  如果结果集没有任何行，此函数将返回 NULL。

- column→path

  描述：相当于 json_extract 的别名，在 JSON 文档提取路径表达式指定的数据并返回，操作

  符->要在查表操作中进行。

  返回类型:json

- column→>path

  描述：功能类似于>path)，取消对 JSON 文档中提取的数据引号的引用，操作符->>要在查表操作中进行。

  返回类型:text

2.4.1.3.14 类型转换函数

- cast(xasy)

  描述：类型转换函数，将 x 转换成 y 指定的类型。如果目标类型为 char 类型，在

  dolphin.b_compatibility_mode=on 的情况下，实际转换为 varchar 类型；否则仍为 char 类型。

  在后续开发中，扩展的 CAST 转换功能有 money 到 unsigned 和 timestamp 到 unsigned 的转换。

  兼容性

  CREATE CAST 指令符合 SQL 标准，除了 SQL 没有为二进制可强制转换类型或者实现

函数的额外参数来实现功能。

2.4.1.3.15 四则运算操作符兼容

在将参数开关 dolphin.b_compatibility_mode 设置为 on 时，表示启用四则运算的 MySQL 兼容。相比于 GBase8c 原生语法，dolphin 对于四则运算的修改主要为:

(1) 支持更多类型参与四则运算，具体如下:

数字类型:tinyint(unsigned)、smallint(unsigned)、integer(unsigned)、bigint(unsigned)、float4、float8、decimal/numeric 以及 bit。

字符串类型：char、varchar、binary、varbinary、tinyblob、blob、mediumblob、longblob、enum、set、json 以及 text(目前 GBase8c 没有实际上的 tinytext、mediumtext 和 longtext，因此不考虑)。

时间日期类型：date、datetime、timestamp、time、year。

(2) 一些原操作符返回值兼容 MySQL，具体的兼容规则如下:

整型 X 整型：针对"+"、"-"、"*"这三个操作符，如果两个都是有符号的整型，则返回结果也是有符号的整型，否则返回无符号整型；针对"/"，其返回值为定点型(numeric 类型)。

整型 X 定点型：针对"+"、"-"、"*"、"/"四则运算，返回定点型。注：GBase8c 定点数未实现无符号，所以无符号整型参与运算返回的结果均为有符号的。

整型 X 浮点型：针对"+"、"-"、"*"、"/"四则运算，返回浮点型。注：GBase8c 浮点数未实现无符号，所以无符号整型参与运算返回的结果均为有符号的。

定点型 X 定点型：针对"+"、"-"、"*"、"/"四则运算，返回定点型。

定点型 X 浮点型：针对"+"、"-"、"*"、"/"四则运算，返回浮点型。

浮点型 X 浮点型：针对"+"、"-"、"*"、"/"四则运算，返回定点型。

基于上述规则，只需要运用字符串类型、时间类型等的类型转换规则，就可以推算出这些类型进行混合运算时的返回值，类型转换规则如下:

字符串类型：在进行四则运算是统一转换为浮点类型

时间日期类型:date 固定转换为有符号整型,year 固定转换为无符号整型;针对 datetime、timestamp、time 三个类型，如果没有指定 typmod(表示毫秒和微秒)的话，就转换为有符号整型，否则会转换为一个定点数，其小数位数与指定的 typmod 相同。

2.4.1.4 表达式

2.4.1.4.1 条件表达式

相比于原始的 GBase8c，dolphin 对于条件表达式的修改主要为:

新增 IFNULL/IF 表达式。

IFNULL

等价于 NVL。如果 value1 为 NULL 则返回 value2，如果 value1 非 NULL，则返回 value1。

IF

仅支持 IF(expr1,expr2,expr3)，等价于 CASE WHEN expr1 THEN expr2 ELSE expr3 END。

CASE 子句可以用于合法的表达式中。condition 是一个返回 BOOLEAN 数据类型的表达式:

如果结果为真，CASE 表达式的结果就是符合该条件所对应的 result。

如果结果为假，则以相同方式处理随后的 WHEN 或 ELSE 子句。

如果各 WHENcondition 都不为真，表达式的结果就是在 ELSE 子句执行的 result。如果省略了 ELSE 子句且没有匹配的条件，结果为 NULL。

2.4.1.5 SQL 语法

2.4.1.5.1 ALTER FUNCTION

2.4.1.5.2 ALTER FUNCTION

功能描述

修改自定义函数的属性。

注意事项

相比于原始的 GBase 8c，dolphin 对于 ALTER PROCEDURE 语法的修改为:

增加可修改 LANGUAGE 选项。

增加可修改项 { CONTAINS SQL | NO SQL | READS SQL DATA | MODIFIES SQL DATA } 。

增加可修改项 SQL SECURITY { DEFINER | INVOKER }。

语法格式

修改自定义函数的附加参数。

```
ALTER FUNCTION function_name ( [ { [ argname ] [ argmode ] argtype} [, ...] ] )
    action [ ... ] [ RESTRICT ];
```

其中附加参数 action 子句语法为。

```
{CALLED ON NULL INPUT | RETURNS NULL ON NULL INPUT | STRICT}
 | {IMMUTABLE | STABLE | VOLATILE}
 | {SHIPPABLE | NOT SHIPPABLE}
 | {NOT FENCED | FENCED}
 | [ NOT ] LEAKPROOF
 | { [ EXTERNAL|SQL ] SECURITY INVOKER | [ EXTERNAL|SQL ] SECURITY DEFINER }
 | AUTHID { DEFINER | CURRENT_USER }
 | COST execution_cost
 | ROWS result_rows
 | SET configuration_parameter { { TO | = } { value | DEFAULT }| FROM CURRENT}
 | RESET {configuration_parameter | ALL}
 | COMMENT 'text'
 | LANGUAGE lang_name
 | { CONTAINS SQL | NO SQL | READS SQL DATA | MODIFIES SQL DATA }
```

参数说明

LANGUAGE lang_name

用以实现函数的语言的名称，仅语法兼容，实际修改不会生效。

SQL SECURITY INVOKER

表明该函数将带着调用它的用户的权限执行。该参数可以省略。

SQL SECURITY INVOKER 和 SECURITY INVOKER 和 AUTHID CURRENT_USER 的功能相同。

SQL SECURITY DEFINER

声明该函数将以创建它的用户的权限执行。

SQL SECURITY DEFINER 和 AUTHID DEFINER 和 SECURITY DEFINER 的功能相同。

CONTAINS SQL | NO SQL | READS SQL DATA | MODIFIES SQL DATA

语法兼容项。

示例

```
--指定 NO SQL
postgres=# ALTER FUNCTION f1 (s char(20)) NO SQL;
--指定 CONTAINS SQL
postgres=# ALTER FUNCTION f1 (s char(20)) CONTAINS SQL;
--指定 LANGUAGE SQL
postgres=# ALTER FUNCTION f1 (s char(20)) LANGUAGE SQL ;
--指定 MODIFIES SQL DATA
postgres=# ALTER FUNCTION f1 (s char(20)) MODIFIES SQL DATA;
--指定  READS SQL DATA
postgres=# ALTER FUNCTION f1 (s char(20)) READS SQL DATA;
--指定 SECURITY INVOKER
postgres=# ALTER FUNCTION f1 (s char(20)) SQL SECURITY INVOKER;
--指定 SECURITY DEFINER
postgres=# ALTER FUNCTION f1 (s char(20)) SQL SECURITY DEFINER;
```

相关链接

ALTER FUNCTION

2.4.1.5.3 ALTER PROCEDURE

功能描述

修改自定义存储过程的属性。

注意事项

相比于原始的 GBase 8c，dolphin 对于 ALTER PROCEDURE 语法的修改为：

增加可修改 LANGUAGE 选项。

增加可修改项 { CONTAINS SQL | NO SQL | READS SQL DATA | MODIFIES SQL DATA } 。

增加可修改项 SQL SECURITY { DEFINER | INVOKER }。

语法格式

修改自定义存储过程的附加参数。

```
ALTER PROCEDURE procedure_name ( [ { [ argname ] [ argmode ] argtype} [, ...] ] )
    action [ ... ] [ RESTRICT ];
```

其中附加参数 action 子句语法为。

{CALLED ON NULL INPUT | STRICT}

```
| {IMMUTABLE | STABLE | VOLATILE}
```

```
| {SHIPPABLE | NOT SHIPPABLE}
| {NOT FENCED | FENCED}
| [ NOT ] LEAKPROOF
| { [ EXTERNAL ] SECURITY INVOKER | [ EXTERNAL ] SECURITY DEFINER }
| AUTHID { DEFINER | CURRENT_USER }
| COST execution_cost
| ROWS result_rows
| SET configuration_parameter { { TO | = } { value | DEFAULT }| FROM CURRENT}
| RESET {configuration_parameter | ALL}
| COMMENT 'text'
| LANGUAGE lang_name
| { CONTAINS SQL | NO SQL | READS SQL DATA | MODIFIES SQL DATA }
```

参数说明

LANGUAGE lang_name

用以实现存储过程的语言的名称，仅语法兼容，实际修改不会生效。

SQL SECURITY INVOKER

表明该存储过程将带着调用它的用户的权限执行。该参数可以省略。

SQL SECURITY INVOKER 和 SECURITY INVOKER 和 AUTHID CURRENT_USER 的功能相同。

SQL SECURITY DEFINER

声明该存储过程将以创建它的用户的权限执行。

SQL SECURITY DEFINER 和 AUTHID DEFINER 和 SECURITY DEFINER 的功能相同。

CONTAINS SQL | NO SQL | READS SQL DATA | MODIFIES SQL DATA

语法兼容项。

示例

```
--指定 NO SQL
postgres=# ALTER PROCEDURE proc1() NO SQL;
--指定 CONTAINS SQL
postgres=# ALTER PROCEDURE proc1() CONTAINS SQL;
--指定 LANGUAGE SQL
postgres=# ALTER PROCEDURE proc1() CONTAINS SQL LANGUAGE SQL ;
--指定 MODIFIES SQL DATA
postgres=# ALTER PROCEDURE proc1() CONTAINS SQL MODIFIES SQL DATA;
```

```
--指定 SECURITY INVOKER
postgres=# ALTER PROCEDURE proc1() SQL SECURITY INVOKER;
```

相关链接

ALTER PROCEDURE

2.4.1.5.4 ALTER SERVER

功能描述

增加、修改和删除一个现有 server 的参数。已有 server 可以从 pg_foreign_server 系统表中查询。

注意事项

本章节只包含 dolphin 新增的语法，原 GBase 8c 的语法未做删除和修改。

相比于原始的 GBase 8c，dolphin 对于 ALTER SERVER 语法的修改主要为：

对于修改的 server 其 fdw_name 为 mysql_fdw 时，增加可选 OPTIONS：DATABASE, USER, PASSWORD, SOCKET, OWNER。

对于修改的 server 其 fdw_name 为 mysql_fdw 时，若 option 未指定执行动作，且 server 的 option 已存在，则将本次语句的动作更改为 SET。

语法格式

修改外部服务的参数。

```
ALTER SERVER server_name [ VERSION 'new_version' ]
    [ OPTIONS ( {[ ADD | SET | DROP ] option ['value']} [, ... ] ) ];
```

修改外部服务的名称。

```
ALTER SERVER server_name
  RENAME TO new_name;
```

参数说明

OPTIONS

更改该服务器的选项。ADD、SET 和 DROP 指定要执行的动作。如果没有显式地指定操作， 将会假定为 ADD。选项名称必须唯一，名称和值也会使用该服务器的外部数据包装器库进行验证。

mysql_fdw 支持的 options 包括：

host （默认值为 127.0.0.1）

MySQL Server/MariaDB 的地址。

port （默认值为 3306）

MySQL Server/MariaDB 侦听的端口号。

user （默认为空）

MySQL Server/MariaDB 用于连接的用户名。若 OPTIONS 指定此选项，且不存在当前用户到给定 server 的用户映射，GBase 8c 将自动创建当前用户到新建 server 的用户映射；若 OPTIONS 指定此选项，且已存在当前用户到给定 server 的用户映射，GBase 8c 将修改该用户映射的对应 option 值。

password （默认为空）

MySQL Server/MariaDB 用于连接的用户密码。若 OPTIONS 指定此选项，且不存在当前用户到给定 server 的用户映射，GBase 8c 将自动创建当前用户到新建 server 的用户映射；若 OPTIONS 指定此选项，且已存在当前用户到给定 server 的用户映射，GBase 8c 将修改该用户映射的对应 option 值。

database （默认为空）

无实际意义，仅做语法兼容。指定 MySQL Server/MariaDB 连接的数据库请在 CREATE FOREIGN TABLE 或 ALTER FOREIGN TABLE 中完成。

owner （默认为空）

无实际意义，仅做语法兼容。

socket （默认为空）

无实际意义，仅做语法兼容。

示例

```
修改 server。
-- 当前用户到给定 server 的用户映射不存在时
postgres=# alter server server_test options(user 'my_user', password
'mypassword');
WARNING:  USER MAPPING for current user to server server_test created.
ALTER SERVER
-- 当前用户到给定 server 的用户映射已存在时
postgres=# alter server server_test options(port '3308', user 'my_user');
```

```
WARNING:  USER MAPPING for current user to server server_test altered.
ALTER SERVER
```

相关链接

CREATE SERVER ， DROP SERVER

2.4.1.5.5 ALTER TABLE

功能描述

修改表，包括修改表的定义、重命名表、重命名表中指定的列、重命名表的约束、设置表的所属模式、添加/更新多个列、打开/关闭行访问控制开关。

注意事项

本章节只包含 dolphin 新增的语法，原 GBase 8c 的语法未做删除和修改。

当一条语句下有多条子命令时，drop index 和 rename index 会优先其他子命令执行，这两种命令的优先级一致。

语法格式

修改表的定义。

```
ALTER TABLE [ IF EXISTS ] { table_name [*] | ONLY table_name | ONLY ( table_name ) }
    action [, ... ];
```

其中具体表操作 action 可以是以下子句之一：

```
column_clause
    | {DISABLE | ENABLE} KEYS
    | DROP INDEX index_name [ RESTRICT | CASCADE ]
    | DROP PRIMARY KEY [ RESTRICT | CASCADE ]
    | DROP FOREIGN KEY foreign_key_name [ RESTRICT | CASCADE ]
    | RENAME INDEX index_name to new_index_name
    | ADD table_indexclause
    | MODIFY column_name column_type ON UPDATE CURRENT_TIMESTAMP
    | AUTOEXTEND_SIZE [=] value
    | AVG_ROW_LENGTH [=] value
    | CHECKSUM [=] value
    | CONNECTION [=] 'connect_string'
    | {DATA | INDEX} DIRECTORY [=] 'absolute path to directory'
    | DELAY_KEY_WRITE [=] value
    | ENCRYPTION [=] 'encryption_string'
    | ENGINE_ATTRIBUTE [=] 'string'
```

```
    | INSERT_METHOD [=] { NO | FIRST | LAST }
    | KEY_BLOCK_SIZE [=] value
    | MAX_ROWS [=] value
    | MIN_ROWS [=] value
    | PACK_KEYS [=] value
    | PASSWORD [=] 'password'
    | START TRANSACTION
    | SECONDARY_ENGINE_ATTRIBUTE [=] 'string'
    | STATS_AUTO_RECALC [=] value
    | STATS_PERSISTENT [=] value
    | STATS_SAMPLE_PAGES [=] value
    | UNION [=] (tbl_name[,tbl_name]...)
    | TABLESPACE tablespace_name [STORAGE DISK]
    | [TABLESPACE tablespace_name] STORAGE MEMORY
```

对一个表进行重建。

```
ALTER TABLE table_name FORCE;
```

重命名表。对名称的修改不会影响所存储的数据。

```
ALTER TABLE [ IF EXISTS ] table_name
    RENAME { TO | AS } new_table_name;
```

对表 timestamp 列添加 ON UPDATE 属性。

```
""ALTER TABLE table_name
    MODIFY column_name column_type ON UPDATE CURRENT_TIMESTAMP;
```

对表 timestamp 列删除 ON UPDATE 属性。

```
""ALTER TABLE table_name
    MODIFY column_name column_type;
ADD table_indexclause
```

在表上新增一个索引

```
{[FULLTEXT] INDEX | KEY} [index_name] [index_type]
(key_part,...)[index_option]...
```

其中参数 index_type 为：

```
USING {BTREE | HASH | GIN | GIST | PSORT | UBTREE}
```

其中参数 key_part 为：

```
{col_name[(length)] | (expr)} [ASC | DESC]
```

其中参数 index_option 为：

```
index_option:{
     COMMENT 'string'
   | index_type
   | [ VISIBLE | INVISIBLE ]
   | [WITH PARSER NGRAM]
}
```

COMMENT、index_type、[VISIBLE | INVISIBLE] 的顺序和数量任意，但相同字段仅最后一个值生效。WITH PARSER NGRAM 为 FULLTEXT INDEX 指定的 ngram 解析器，前提是索引必须指定关键字 FULLTEXT，FULLTEXT 默认 WITH PARSER NGRAM。

参数说明

{DISABLE | ENABLE} KEYS

禁用和启用一个表的所有非唯一索引。

DROP INDEX index_name [RESTRICT | CASCADE]

删除一个表的索引。

DROP PRIMARY KEY [RESTRICT | CASCADE]

删除一个表的外键。

DROP FOREIGN KEY foreign_key_name [RESTRICT | CASCADE]

删除一个表的外键。

RENAME INDEX index_name to new_index_name

重命名一个表的索引。

AUTOEXTEND_SIZE [=] value

用于指定在表空间变满时扩展表空间大小；目前该特性仅有语法支持，不实现功能。参数的取值范围包括非负整数，小数，标识符，非负整数+标识符，小数+标识符。

AVG_ROW_LENGTH [=] value

用于指定表的平均行长度；目前该特性仅有语法支持，不实现功能。参数的取值范围包括非负整数，小数。

CHECKSUM [=] value

用于指定是否维护所有行的实时校验和；目前该特性仅有语法支持，不实现功能。参数的取值范围为非负整数，小数，十六进制数。

CONNECTION [=] 'connect_string'

用于指定联合表的连接字符串；目前该特性仅有语法支持，不实现功能。参数的取值范围为任意字符串。

{DATA | INDEX} DIRECTORY [=] 'absolute path to directory'

用于指定表数据数据和索引的存储目录；目前该特性仅有语法支持，不实现功能。参数的取值范围为任意字符串。

DELAY_KEY_WRITE [=] value

用于指定是否延迟表的键更新直到表关闭；目前该特性仅有语法支持，不实现功能。参数的取值范围为非负整数，小数，十六进制数。

ENCRYPTION [=] 'encryption_string'

用于指定表启用或禁用页面级数据加密；目前该特性仅有语法支持，不实现功能。参数的取值范围为任意字符串。

ENGINE_ATTRIBUTE [=] 'string'

用于指定主存储引擎的表属性；目前该特性仅有语法支持，不实现功能。参数的取值范围为任意字符串。

INSERT_METHOD [=] { NO | FIRST | LAST }

用于指定应将行插入到的表；目前该特性仅有语法支持，不实现功能。参数的取值范围为 NO，FIRST，LAST。

KEY_BLOCK_SIZE [=] value

用于指定索引键块的字节大小；目前该特性仅有语法支持，不实现功能。参数的取值范围为非负整数，小数。

MAX_ROWS [=] value

用于指定计划在表中存储的最大行数；目前该特性仅有语法支持，不实现功能。参数的取值范围为非负整数，小数。

MIN_ROWS [=] value

用于指定计划在表中存储的最小行数；目前该特性仅有语法支持，不实现功能。参数的取值范围为非负整数，小数。

PACK_KEYS [=] value

用于指定控制压缩索引的方式；目前该特性仅有语法支持，不实现功能。参数的取值范围为非负整数，小数，十六进制数，DEFAULT。

PASSWORD [=] 'password'

此选项未使用；目前该特性仅有语法支持，不实现功能。参数的取值范围为任意字符串。

SECONDARY_ENGINE_ATTRIBUTE [=] 'string'

用于指定辅助存储引擎的表属性；目前该特性仅有语法支持，不实现功能。参数的取值范围为任意字符串。

START TRANSACTION

用于开启事务模式；目前该特性仅有语法支持，不实现功能。

STATS_AUTO_RECALC [=] value

用于指定是否自动重新计算表的持久统计信息；目前该特性仅有语法支持，不实现功能。参数的取值范围为非负整数，小数，十六进制数，DEFAULT。

STATS_PERSISTENT [=] value

用于指定是否为表启用持久统计信息；目前该特性仅有语法支持，不实现功能。参数的取值范围为非负整数，小数，十六进制数，DEFAULT。

STATS_SAMPLE_PAGES [=] value

用于指定估计索引列的基数和其他统计信息时要采样的索引页数；目前该特性仅有语法支持，不实现功能。参数的取值范围为非负整数，小数，十六进制数。

UNION [=] (tbl_name[,tbl_name]•••)

用于访问一组相同的表作为一个表；目前该特性仅有语法支持，不实现功能。

TABLESPACE tablespace_name [STORAGE DISK]

用于指定表存储在磁盘；目前该特性仅有语法支持，不实现功能。

[TABLESPACE tablespace_name] STORAGE MEMORY

用于指定表存储在内存；目前该特性仅有语法支持，不实现功能。

 说明：

涉及的**参数说明**可见 ALTER TABLE。

示例

```
— 创建表、外键和非唯一索引。
postgres=# CREATE TABLE alter_table_tbl1 (a INT PRIMARY KEY, b INT);
postgres=# CREATE TABLE alter_table_tbl2 (c INT PRIMARY KEY, d INT);
postgres=# ALTER TABLE alter_table_tbl2 ADD CONSTRAINT alter_table_tbl_fk
FOREIGN KEY (d) REFERENCES alter_table_tbl1 (a);
postgres=# CREATE INDEX alter_table_tbl_b_ind ON alter_table_tbl1(b);
— 禁用和启用非唯一索引。
postgres=# ALTER TABLE alter_table_tbl1 DISABLE KEYS;
postgres=# ALTER TABLE alter_table_tbl1 ENABLE KEYS;
— 删除索引。
postgres=# ALTER TABLE alter_table_tbl1 DROP KEY alter_table_tbl_b_ind;
— 删除主键。
postgres=# ALTER TABLE alter_table_tbl2 DROP PRIMARY KEY;
— 删除外键。
postgres=# ALTER TABLE alter_table_tbl2 DROP FOREIGN KEY alter_table_tbl_fk;
— 重建表。
postgres=# ALTER TABLE alter_table_tbl1 FORCE;
— 重命名索引。
postgres=# CREATE INDEX alter_table_tbl_b_ind ON alter_table_tbl1(b);
postgres=# ALTER TABLE alter_table_tbl1 RENAME INDEX alter_table_tbl_b_ind TO
new_alter_table_tbl_b_ind;
— 修改表，创建 INVISIBLE 普通索引
postgres=# ALTER TABLE alter_table_tbl1 ADD INDEX alter_table_tbl_b_ind(b)
INVISIBLE;
— 删除表。
postgres=# DROP TABLE alter_table_tbl1, alter_table_tbl2;
— 兼容 MySQL 全文索引，添加全文索引语法，前提是兼容模式为 B 的数据库。
""test=# ALTER TABLE test ADD FULLTEXT INDEX test_index_1 (title, boby) WITH
PARSER ngram;
ALTER TABLE
test=# \d test_index_1
                  Index "fulltext_test.test_index_1"
    Column     | Type |                    Definition
---------------+------+-------------------------------------------------
 to_tsvector  | text | to_tsvector('"ngram"'::regconfig, title::text)
 to_tsvector1 | text | to_tsvector('"ngram"'::regconfig, boby)
gin, for table "fulltext_test.test"
```

相关链接

ALTER TABLE

2.4.1.5.6 ALTER TABLE-PARTITION

功能描述

修改表分区，包括增删分区、切割分区、合成分区以及修改分区属性等。

相比于内核语法，dolphin 的 rebuild,remove,check,repair,optimize,truncate,analyze,exchange,reorganize 都做了 B 兼容模式下的特色修改。

注意事项

添加分区的表空间不能是 PG_GLOBAL。

添加分区的名称不能与该分区表已有分区的名称相同。

添加分区的分区键值要和分区表的分区键的类型一致。

若添加 RANGE 分区，添加分区键值要大于分区表中最后一个范围分区的上边界。

若添加 LIST 分区，添加分区键值不能与现有分区键值重复。

不支持添加 HASH 分区。

如果目标分区表中已有分区数达到了最大值 1048575，则不能继续添加分区。

当分区表只有一个分区时，不能删除该分区。

选择分区使用 PARTITION FOR()，括号里指定值个数应该与定义分区时使用的列个数相同，并且一一对应。

Value 分区表不支持相应的 Alter Partition 操作。

列存分区表不支持切割分区。

间隔分区表不支持添加分区。

哈希分区表不支持切割分区，不支持合成分区，不支持添加和删除分区。

列表分区表不支持切割分区，不支持合成分区。

只有分区表的所有者或者被授予了分区表 ALTER 权限的用户有权限执行 ALTER TABLE PARTITION 命令，系统管理员默认拥有此权限。

语法格式

修改表分区主语法。

```
ALTER TABLE [ IF EXISTS ] { table_name  [*] | ONLY table_name | ONLY
( table_name  )}
    action [, ... ];
```

其中 action 统指如下分区维护子语法。当存在多个分区维护子句时，保证了分区的连续性，无论这些子句的排序如何，GBase 8c 总会先执行 DROP PARTITION 再执行 ADD PARTITION 操作，最后顺序执行其它分区维护操作。

```
    move_clause  |
    exchange_clause  |
    row_clause  |
    merge_clause  |
    modify_clause  |
    split_clause  |
    add_clause  |
    drop_clause  |
    truncate_clause |
    rebuild_clause |
    remove_clause |
    repair_clause |
    check_clause |
    optimize_clause
```

move_clause 子语法用于移动分区到新的表空间。

```
MOVE PARTITION { partion_name | FOR ( partition_value [, ...] ) } TABLESPACE
tablespacename
```

exchange_clause 子语法用于把普通表的数据迁移到指定的分区。

```
EXCHANGE PARTITION { ( partition_name ) | FOR ( partition_value [, ...] ) }
    WITH TABLE {[ ONLY ] ordinary_table_name | ordinary_table_name * | ONLY
( ordinary_table_name )}
    [ { WITH | WITHOUT } VALIDATION ] [ VERBOSE ] [ UPDATE GLOBAL INDEX ]
```

进行交换的普通表和分区必须满足如下条件：

普通表和分区的列数目相同，对应列的信息严格一致，包括：列名、列的数据类型、列约束、列的 Collation 信息、列的存储参数、列的压缩信息等。

普通表和分区的表压缩信息严格一致。

普通表和分区的索引个数相同，且对应索引的信息严格一致。

普通表和分区的表约束个数相同，且对应表约束的信息严格一致。

普通表不可以是临时表，分区表只能是范围分区表，列表分区表，哈希分区表。

普通表和分区表上不可以有动态数据脱敏，行访问控制约束。

列表分区表，哈希分区表不能是列存储。

List/Hash/Range 类型分区表支持 exchange_clause。

须知：

完成交换后，普通表和分区的数据被置换，同时普通表和分区的表空间信息被置换。此时，普通表和分区的统计信息变得不可靠，需要对普通表和分区重新执行 analyze。

由于非分区键不能建立本地唯一索引，只能建立全局唯一索引，所以如果普通表含有唯一索引时，会导致不能交换数据。

row_clause 子语法用于设置分区表的行迁移开关。

```
{ ENABLE | DISABLE } ROW MOVEMENT
```

merge_clause 子语法用于把多个分区合并成一个分区。

```
MERGE PARTITIONS { partition_name } [, ...] INTO PARTITION partition_name
[ TABLESPACE tablespacename ] [ UPDATE GLOBAL INDEX ]
```

modify_clause 子语法用于设置分区索引是否可用。

```
MODIFY PARTITION partition_name { UNUSABLE LOCAL INDEXES | REBUILD UNUSABLE LOCAL
INDEXES }
```

split_clause 子语法用于把一个分区切割成多个分区。

```
SPLIT PARTITION { partition_name | FOR ( partition_value [, ...] ) }
{ split_point_clause | no_split_point_clause } [ UPDATE GLOBAL INDEX ]
```

指定切割点 split_point_clause 的语法为。

```
AT ( partition_value ) INTO ( PARTITION partition_name [ TABLESPACE
tablespacename ] , PARTITION partition_name [ TABLESPACE tablespacename ] )
```

须知：

列存分区表不支持切割分区。

切割点的大小要位于正在被切割的分区的分区键范围内，指定切割点的方式只能把一个分区切割成两个新分区。

不指定切割点 no_split_point_clause 的语法为。

```
INTO { ( partition_less_than_item [, ...] ) | ( partition_start_end_item
[, ...] ) }
```

须知：

不指定切割点的方式，partition_less_than_item 指定的第一个新分区的分区键要大于正在被切割的分区的前一个分区（如果存在的话）的分区键，partition_less_than_item 指定的最后一个分区的分区键要等于正在被切割的分区的分区键大小。

不指定切割点的方式，partition_start_end_item 指定的第一个新分区的起始点（如果存在的话）必须等于正在被切割的分区的前一个分区（如果存在的话）的分区键，partition_start_end_item 指定的最后一个分区的终止点（如果存在的话）必须等于正在被切割的分区的分区键。

partition_less_than_item 支持的分区键个数最多为 4，而 partition_start_end_item 仅支持 1 个分区键，其支持的数据类型参见 PARTITION BY RANGE(parti…。

在同一语句中 partition_less_than_item 和 partition_start_end_item 两者不可同时使用；不同 split 语句之间没有限制。

分区项 partition_less_than_item 的语法为。

```
PARTITION partition_name VALUES LESS THAN ( { partition_value | MAXVALUE }
[, ...] ) | MAXVALUE
    [ TABLESPACE tablespacename ]
```

分区项 partition_start_end_item 的语法为，其约束参见 START END 语法描述。

```
PARTITION partition_name {
        {START(partition_value) END (partition_value) EVERY (interval_value)} |
        {START(partition_value) END ({partition_value | MAXVALUE}) | MAXVALUE}
|
        {START(partition_value)} |
        {END ({partition_value | MAXVALUE}) | MAXVALUE}
} [TABLESPACE tablespace_name]
```

add_clause 子语法用于为指定的分区表添加一个或多个分区。

```
    ADD PARTITION ( partition_col1_name = partition_col1_value [,
partition_col2_name = partition_col2_value ] [, ...] )
        [ LOCATION 'location1' ]
        [ PARTITION (partition_colA_name = partition_colA_value [,
partition_colB_name = partition_colB_value ] [, ...] ) ]
        [ LOCATION 'location2' ]
```

```
    ADD {partition_less_than_item | partition_start_end_item|
partition_list_item }
```

分区项 partition_list_item 的语法如下。

```
PARTITION partition_name VALUES (list_values_clause)
    [ TABLESPACE tablespacename ]
```

须知:

partition_list_item 仅支持的 1 个分区键，其支持的数据类型参见 PARTITION BY LIST(partit…。

间隔/哈希分区表不支持添加分区。

drop_clause 子语法用于删除分区表中的指定分区。

```
DROP PARTITION  { partition_name | FOR (  partition_value [, ...] )  } [ UPDATE
GLOBAL INDEX ]
```

须知： 哈希分区表不支持删除分区。

truncate_clause 子语法用于清空分区表中的指定分区。

```
TRUNCATE PARTITION  { partition_name | FOR (  partition_value [, ...] )  }
[ UPDATE GLOBAL INDEX ]
```

修改表分区名称的语法。

```
ALTER TABLE [ IF EXISTS ] { table_name [*] | ONLY table_name | ONLY ( table_name  )}
    RENAME PARTITION { partion_name | FOR ( partition_value [, ...] ) } TO
partition_new_name;
```

重建分区语法

一般用于回收分区使用空间，与删除存储在分区中的所有记录，然后重新插入它们的效果相同。这对于碎片整理很有用。

不支持列存表，不支持指定二级分区表的二级分区。

```
REBUILD PARTITION  { partition_name } [, ...]
REBUILD PARTITION ALL
```

分区表 remove partitioning 语法

移除表中 partition，但是保留所有数据。

不支持列存表和 segment 表。

REMOVE PARTITIONING

分区表 repair，check 和 optimize 语法

仅支持语法，不做实际功能支持。

```
CHECK PARTITION { partition_name } [, ...]
CHECK PARTITION ALL
REPAIR PARTITION { partition_name } [, ...]
REPAIR PARTITION ALL
OPTIMIZE PARTITION { partition_name } [, ...]
OPTIMIZE PARTITION ALL
```

Truncate 分区语法

Truncate 操作会删除当前分区对应的所有数据。

TRUNCATE PARTITION { partition_name } [, ...]

TRUNCATE PARTITION all

exchange 分区语法对齐

可以用来交换分区表和普通表的数据，普通表和分区的数据被置换，同时普通表和分区的表空间信息被置换。此时，普通表和分区的统计信息变得不可靠，需要对普通表和分区重新执行 analyze。

不支持交换二级分区。

```
exchange partition partition_name with table table_name (without/with
validation);
```

analyze 分区语法对齐

用于收集与表内容相关的统计信息。执行计划生成器会使用这些统计数据，以确定最有效的执行计划。

不支持 analyze 指定二级分区。

```
analyze partition { partition_name } [, ...]
analyze partition all;
```

add 分区语法。

```
ADD {partition_less_than_item | partition_start_end_item| partition_list_item }
[, ...]
```

drop 分区语法。

```
DROP PARTITION { { partition_name } [ UPDATE GLOBAL INDEX ] } [, ...]
DROP SUBPARTITION { { partition_name } [ UPDATE GLOBAL INDEX ] } [, ...]
```

reorganize 分区语法。

重新分割或融合指定分区，重新划分分区的定义。

以下是 ALTER TABLE … REORGANIZE PARTITION 用于重新分区一些关键点：

PARTITION 用于确定新分区方案的选项应遵循与 CREATE TABLE 语句所使用的规则相同的规则。

新的 RANGE 分区方案不能有任何重叠范围。一个新的 LIST 分区方案不能有任何重叠的值集。

partition_definitions 列表中的分区组合应与清单中命名的组合分区具有相同的范围或整体值集 partition_list。

对于由分区的表 RANGE，您只能重组相邻的分区。您不能跳过范围分区。

对于 LIST 分区，不可以删除已有对应数据的 value 值定义。

不能用于 REORGANIZE PARTITION 更改表使用的分区类型。

不可丢失原有表数据。

不支持 interval 分区，不支持 value 分区。

对于 RANGE 分区，不支持 start end 语法。

```
REORGANIZE PARTITION {{ partition_name } [, ...]} INTO {partition_less_than_item
| partition_list_item } [, ...]
```

参数说明

table_name

分区表名。

取值范围：已存在的分区表名。

partition_name

分区名。

取值范围：已存在的分区名。

tablespacename

指定分区要移动到哪个表空间。

取值范围：已存在的表空间名。

partition_value

分区键值。

通过 PARTITION FOR (partition_value [, ···])子句指定的这一组值，可以唯一确定一个分区。

取值范围：需要进行重命名的分区的分区键的取值范围。

UNUSABLE LOCAL INDEXES

设置该分区上的所有索引不可用。

REBUILD UNUSABLE LOCAL INDEXES

重建该分区上的所有索引。

ENABLE/DISABLE ROW MOVEMET

行迁移开关。

如果进行 UPDATE 操作时，更新了元组在分区键上的值，造成了该元组所在分区发生变化，就会根据该开关给出报错信息，或者进行元组在分区间的转移。

取值范围：

ENABLE：打开行迁移开关。

DISABLE：关闭行迁移开关。

默认是打开状态。

ordinary_table_name

进行迁移的普通表的名称。

取值范围：已存在的普通表名。

{ WITH | WITHOUT } VALIDATION

在进行数据迁移时，是否检查普通表中的数据满足指定分区的分区键范围。

取值范围：

WITH：对于普通表中的数据要检查是否满足分区的分区键范围，如果有数据不满足，则报错。

WITHOUT：对于普通表中的数据不检查是否满足分区的分区键范围。

默认是 WITH 状态。

由于检查比较耗时，特别是当数据量很大的情况下更甚。所以在保证当前普通表中的数据满足分区的分区键范围时，可以加上 WITHOUT 来指明不进行检查。

VERBOSE

在 VALIDATION 是 WITH 状态时，如果检查出普通表有不满足要交换分区的分区键范围的数据，那么把这些数据插入到正确的分区，如果路由不到任何分区，再报错。

须知： 只有在 VALIDATION 是 WITH 状态时，才可以指定 VERBOSE。

partition_new_name

分区的新名称。

取值范围：字符串，要符合标识符的命名规范。

示例

请参考 CREATE TABLE PARTITION 的示例。

相关链接

CREATE TABLE PARTITION，DROP TABLE

2.4.1.5.7 ALTER TABLESPACE

功能描述

修改表空间的属性。

注意事项

相比于原始的 GBase 8c，dolphin 对于 ALTER TABLESPACE 语法的修改主要为：

新增 WAIT 可选项，无实际意义，仅作语法兼容。

新增 ENGINE [=] engine_name 可选项，无实际意义，仅作语法兼容。

语法格式

重命名表空间的语法。

```
ALTER TABLESPACE tablespace_name
    RENAME TO new_tablespace_name [ alter_option_list  [ ... ] ];
```

设置表空间所有者的语法。

```
ALTER TABLESPACE tablespace_name
```

```
    OWNER TO new_owner [ alter_option_list  [ ... ] ];
```

设置表空间属性的语法。

```
ALTER TABLESPACE tablespace_name
    SET ( {tablespace_option = value} [, ... ] )
     [ alter_option_list  [ ... ] ];
```

重置表空间属性的语法。

```
ALTER TABLESPACE tablespace_name
    RESET ( { tablespace_option } [, ...] )
     [ alter_option_list  [ ... ] ];
```

设置表空间限额的语法。

```
ALTER TABLESPACE tablespace_name
    RESIZE MAXSIZE { UNLIMITED | 'space_size'}
     [ alter_option_list  [ ... ] ];
```

其中 alter_option_list 为：

WAIT

| ENGINE [=] engine_name

参数说明

tablespace_name

要修改的表空间。

取值范围：已存在的表空间名。

new_tablespace_name

表空间的新名称。

新名称不能以“PG_”开头。

取值范围：字符串，符合标识符命名规范。

new_owner

表空间的新所有者。

取值范围：已存在的用户名。

tablespace_option

设置或者重置表空间的参数。

取值范围：

seq_page_cost：设置优化器计算一次顺序获取磁盘页面的开销。缺省为 1.0。

random_page_cost：设置优化器计算一次非顺序获取磁盘页面的开销。缺省为 4.0。

说明：

random_page_cost 是相对于 seq_page_cost 的取值，等于或者小于 seq_page_cost 时毫无意义。

默认值为 4.0 的前提条件是，优化器采用索引来扫描表数据，并且表数据在 cache 中命中率可以 90%左右。

如果表数据空间要比物理内存小，那么减小该值到一个适当水平；相反地，如果表数据在 cache 中命中率要低于 90%，那么适当增大该值。

如果采用了类似于 SSD 的随机访问代价较小的存储器，可以适当减小该值，以反映真正的随机扫描代价。

value 的取值范围：正的浮点类型。

RESIZE MAXSIZE

重新设置表空间限额的数值。

取值范围：

UNLIMITED，该表空间不设置限额。

由 space_size 来确定，其格式参考 CREATE TABLESPACE。

说明：

若调整后的限额值比当前表空间实际使用的值要小，调整操作可以执行成功，后续用户需要将该表空间的使用值降低到新限额值之下，才能继续往该表空间中写入数据。

修改参数 MAXSIZE 时也可使用：

```
ALTER TABLESPACE tablespace_name RESIZE MAXSIZE
{'UNLIMITED' | 'space_size'};
```

engine_name

无实际意义。

取值范围：任意字符串。

示例

```
--创建表空间。
postgres=# CREATE TABLESPACE ds_location1 RELATIVE LOCATION
'tablespace/tablespace_1';
--创建用户 joe。
postgres=# CREATE ROLE joe IDENTIFIED BY 'xxxxxxxxx';
--创建用户 jay。
postgres=# CREATE ROLE jay IDENTIFIED BY 'xxxxxxxxx';
--创建表空间，且所有者指定为用户 joe。
postgres=# CREATE TABLESPACE ds_location2 OWNER joe RELATIVE LOCATION
'tablespace/tablespace_1';
--把表空间 ds_location1 重命名为 ds_location3，指定 option WAIT，不影响实际功能。
postgres=# ALTER TABLESPACE ds_location1 RENAME TO ds_location3 WAIT;
--改变表空间 ds_location2 的所有者，指定 option ENGINE，不影响实际功能。
postgres=# ALTER TABLESPACE ds_location2 OWNER TO jay ENGINE = 'test';
--改变表空间 ds_location2 的限额，同时指定 option ENGINE 和 WAIT，不影响实际功能。
postgres=# ALTER TABLESPACE ds_location2 RESIZE MAXSIZE UNLIMITED ENGINE = 'test'
WAIT;
--删除表空间。
postgres=# DROP TABLESPACE ds_location2 ENGINE = 'test2';
postgres=# DROP TABLESPACE ds_location3;
--删除用户。
postgres=# DROP ROLE joe;
postgres=# DROP ROLE jay;
```

相关链接

CREATE TABLESPACE，DROP TABLESPACE

2.4.1.5.8 ALTER VIEW

功能描述

ALTER VIEW 更改视图的各种辅助属性。（如果用户是更改视图的查询定义，要使用 CREATE OR REPLACE VIEW。）

注意事项

只有视图的所有者或者被授予了视图 ALTER 权限的用户才可以执行 ALTER VIEW 命令，系统管理员默认拥有该权限。针对所要修改属性的不同，对其还有以下权限约束：

修改视图的模式，当前用户必须是视图的所有者或者系统管理员，且要有新模式的

CREATE 权限，且不能与新模式中已存在的 synonym 产生命名冲突。

修改视图的所有者，当前用户必须是视图的所有者或者系统管理员，且该用户必须是新所有者角色的成员，并且此角色必须有视图所在模式的 CREATE 权限。

修改视图的命名，不能与当前模式中已存在的 synonym 产生命名冲突。

新增可以指定 ALGORITHM 选项语法。

语法格式

设置视图列的默认值。

```
ALTER [ALGORITHM = {UNDEFINED | MERGE | TEMPTABLE}] VIEW [ IF EXISTS ] view_name
    ALTER [ COLUMN ] column_name SET DEFAULT expression;
```

取消列视图列的默认值。

```
ALTER [ALGORITHM = {UNDEFINED | MERGE | TEMPTABLE}] VIEW [ IF EXISTS ] view_name
    ALTER [ COLUMN ] column_name DROP DEFAULT;
```

修改视图的所有者。

```
ALTER [ALGORITHM = {UNDEFINED | MERGE | TEMPTABLE}] VIEW [ IF EXISTS ] view_name
    OWNER TO new_owner;
```

重命名视图。

```
ALTER [ALGORITHM = {UNDEFINED | MERGE | TEMPTABLE}] VIEW [ IF EXISTS ] view_name
    RENAME TO new_name;
```

设置视图的所属模式。

```
ALTER [ALGORITHM = {UNDEFINED | MERGE | TEMPTABLE}] VIEW [ IF EXISTS ] view_name
    SET SCHEMA new_schema;
```

设置视图的选项。

```
ALTER [ALGORITHM = {UNDEFINED | MERGE | TEMPTABLE}] VIEW [ IF EXISTS ] view_name
    SET ( { view_option_name [ = view_option_value ] } [, ... ] );
```

重置视图的选项。

```
ALTER [ALGORITHM = {UNDEFINED | MERGE | TEMPTABLE}] VIEW [ IF EXISTS ] view_name
    RESET ( view_option_name [, ... ] );
```

设置视图的定义（该语法仅支持在 B 兼容模式下才能使用）

```
ALTER [ALGORITHM = {UNDEFINED | MERGE | TEMPTABLE}] [DEFINER = user] VIEW view_name
[ ( column_name [, ...] ) ]
  AS query [WITH [CASCADE | LOCAL] CHECK OPTION];
```

说明：

ALTER VIEW AS 中的 query 新查询不能改变原查询的列定义，包括顺序、列名、数据类型、类型精度等，只可在列表末尾添加其他的列。

参数说明

IF EXISTS

使用这个选项，如果视图不存在时不会产生错误，仅有会有一个提示信息。

ALGORITHM

指定算法，可选项：UNDEFINED、MERGE、TEMPTABLE。当前只做语法兼容，暂无实际功能。

view_name

视图名称，可以用模式修饰。

取值范围：字符串，符合标识符命名规范。

column_name

可选的名称列表，视图的字段名。如果没有给出，字段名取自查询中的字段名。

取值范围：字符串，符合标识符命名规范。

SET/DROP DEFAULT

设置或删除一个列的缺省值，该参数暂无实际意义。

new_owner

视图新所有者的用户名称。

new_name

视图的新名称。

new_schema

视图的新模式。

view_option_name [= view_option_value]

该子句为视图指定一个可选的参数。

security_barrier

当 VIEW 试图提供行级安全时，应使用该参数。

取值范围：Boolean 类型，TRUE、FALSE。

check_option

指定该视图的检查选项。

取值范围：LOCAL、CASCADED。

示例

```
--创建一个由 c_customer_sk 小于 150 的内容组成的视图。
postgres=# CREATE VIEW tpcds.customer_details_view_v1 AS
    SELECT * FROM tpcds.customer
    WHERE c_customer_sk < 150;
--修改视图名称。
postgres=# ALTER VIEW tpcds.customer_details_view_v1 RENAME TO
customer_details_view_v2;
--修改视图所属 schema。
postgres=# ALTER VIEW tpcds.customer_details_view_v2 SET schema public;
--删除视图。
postgres=# DROP VIEW public.customer_details_view_v2;
```

相关链接

CREATE VIEW，DROP VIEW

2.4.1.5.9 ANALYZE ANALYSE

功能描述

用于收集与数据库中普通表内容相关的统计信息，统计结果存储在系统表 PG_STATISTIC 下。执行计划生成器会使用这些统计数据，以确定最有效的执行计划。

如果没有指定参数，ANALYZE 会分析当前数据库中的每个表和分区表。同时也可以通过指定 table_name、column 和 partition_name 参数把分析限定在特定的表、列或分区表中。

ANALYZE|ANALYSE VERIFY 用于检测数据库中普通表（行存表、列存表）的数据文件是否损坏。

注意事项

说明：**注意事项**可见 ANALYZE。

语法格式

收集表的统计信息

```
""{ANALYZE | ANALYSE} [ VERBOSE ] [ NO_WRITE_TO_BINLOG | LOCAL ] TABLE
{ [schema.]table_name } [, ... ]
```

参数说明

NO_WRITE_TO_BINLOG | LOCAL

仅作语法，无实际用途

 说明：涉及的参数可见 ANALYZE。

示例

```
— 创建表。
postgres=# CREATE TABLE customer_info
(
WR_RETURNED_DATE_SK       INTEGER                       ,
WR_RETURNED_TIME_SK       INTEGER                       ,
WR_ITEM_SK                INTEGER               NOT NULL,
WR_REFUNDED_CUSTOMER_SK   INTEGER
)
;
— 创建分区表。
postgres=# CREATE TABLE customer_par
(
WR_RETURNED_DATE_SK       INTEGER                       ,
WR_RETURNED_TIME_SK       INTEGER                       ,
WR_ITEM_SK                INTEGER               NOT NULL,
WR_REFUNDED_CUSTOMER_SK   INTEGER
)
PARTITION BY RANGE(WR_RETURNED_DATE_SK)
(
PARTITION P1 VALUES LESS THAN(2452275),
PARTITION P2 VALUES LESS THAN(2452640),
PARTITION P3 VALUES LESS THAN(2453000),
PARTITION P4 VALUES LESS THAN(MAXVALUE)
)
ENABLE ROW MOVEMENT;
— 使用 ANALYZE 语句更新统计信息。
postgres=# ANALYZE TABLE customer_info, customer_par;
        Table         |   Op    | Msg_type | Msg_text
----------------------+---------+----------+----------
```

```
 public.customer_info | analyze | status   | OK
 public.customer_par  | analyze | status   | OK
(2 row)
-- 删除表。
postgres=# DROP TABLE customer_info;
postgres=# DROP TABLE customer_par;
```

相关链接

ANALYZE

2.4.1.5.10 AST

功能描述

GBase 8c 语法树校验。

对 AST 语法后的语句是否支持生成 GBase 8c 语法树作判断。

注意事项

校验不通过时，会抛出语法解析相应错误。校验通过时不作任何回显操作。

语法格式

AST [STMT] ;

参数说明

STMT

支持任意类型 SQL 语句、存储过程语句等。

示例

```
-- 建表语句校验
postgres=# AST CREATE TABLE TEST(ID INT6);
-- 不支持语句校验
postgres=# AST CREATE TABLE TEST;
ERRPR: syntax error at or near ";"
LINE 1:AST CREATE TABLE TEST;
                           ^
```

2.4.1.5.11 CHECKSUM-TABLE

功能描述

计算表数据校验和。

注意事项

不支持 QUICK 模式（返回 NULL）。

对于非普通表（例如视图）、不存在的表均返回 NULL。

不支持与异构数据库的表校验和的可比性。（例如对于相同数目，在 GBase 8c 和 mysql 中查询结果无法对比）。

非 QUICK 模式的校验和计算基于查询结果子串，暂不支持针对列的数据类型的区分。

语法格式

```
CHECKSUM TABLE tbl_name [, tbl_name] ... [QUICK | EXTENDED]
```

参数说明

tbl_name

表名，可指定表名。也可以指定 schema_name.table_name。

[QUICK | EXTENDED]

校验模式，只支持 EXTENDED（也即默认值）。

示例

```
--创建简单表
postgres=# CREATE SCHEMA tst_schema1;
postgres=# SET SEARCH_PATH TO tst_schema1;
postgres=# CREATE TABLE tst_t1
(
id int,
name VARCHAR(20),
addr text,
phone text,
addr_code text
);
postgres=# CREATE TABLE tst_t2 AS SELECT * FROM tst_t1;
INSERT 0 0
--不同插入顺序校验
postgres=# INSERT INTO tst_t1 values(2022001, 'tst_name1', 'tst_addr1',
'15600000001', '000001');
INSERT INTO tst_t1 values(2022002, 'tst_name2', 'tst_addr2', '15600000002',
'000002');
```

```
INSERT INTO tst_t1 values(2022003, 'tst_name3', 'tst_addr3', '15600000003',
'000003');
INSERT INTO tst_t1 values(2022004, 'tst_name4', 'tst_addr4', '15600000004',
'000004');
INSERT INTO tst_t2 (SELECT * FROM tst_t1 ORDER BY id DESC);
postgres=# checksum table tst_t1,tst_t2,xxx;
       Table         |  Checksum
---------------------+------------
 tst_schema1.tst_t1 | 1579899754
 tst_schema1.tst_t2 | 1579899754
 tst_schema1.xxx    | NULL

 --含大段字段的表测试
 postgres=# CREATE TABLE blog
(
id int,
title text,
content text
);
postgres=# CREATE TABLE blog_v2 AS SELECT * FROM blog;
postgres=# INSERT INTO blog values(1, 'title1', '01234567890'), (2, 'title2',
'0987654321');
postgres=# CREATE OR REPLACE FUNCTION loop_insert_result_toast(n integer)
RETURNS integer AS $$
DECLARE
    count integer := 0;
BEGIN
    LOOP
        EXIT WHEN count = n;
        UPDATE blog SET content=content||content where id = 2;
        count := count + 1;
    END LOOP;
    RETURN count;
END; $$
LANGUAGE PLPGSQL;
postgres=# select loop_insert_result_toast(16);
 loop_insert_result_toast
--------------------------
                       16
postgres=# INSERT INTO blog_v2 (SELECT * FROM blog);
postgres=# checksum table blog,blog_v2;
```

```
        Table         |  Checksum
----------------------+------------
 tst_schema1.blog     | 6249493220
 tst_schema1.blog_v2  | 6249493220
--段页式表测试
postgres=# CREATE TABLE tst_seg_t1(id int, name VARCHAR(20)) WITH (segment=on);
postgres=# CREATE TABLE tst_seg_t2(id int, name VARCHAR(20)) WITH (segment=on);
postgres=# INSERT INTO tst_seg_t1 values(2022001, 'name_example_1');
INSERT INTO tst_seg_t1 values(2022002, 'name_example_2');
INSERT INTO tst_seg_t1 values(2022003, 'name_example_3');
postgres=# INSERT INTO tst_seg_t2 (SELECT * FROM tst_seg_t1);
postgres=# checksum table tst_seg_t1,tst_seg_t2;
         Table          |  Checksum
------------------------+------------
 tst_schema1.tst_seg_t1 | 5620410817
 tst_schema1.tst_seg_t2 | 5620410817
```

2.4.1.5.12 CREATE-FUNCTION

功能描述

创建一个函数。

注意事项

相比于原始的 GBase 8c，dolphin 对于 CREATE FUNCTION 语法的修改为：

增加 LANGUAGE 默认值 plpgsql。

增加语法兼容项 [NOT] DETERMINISTIC。

增加语法兼容项 { CONTAINS SQL | NO SQL | READS SQL DATA | MODIFIES SQL DATA } 。

增加语法兼容项 SQL SECURITY { DEFINER | INVOKER }。

增加 MySQL 风格**语法格式**。

语法格式

dolphin 加载后，CREATE FUNCTION 语法的格式为

兼容 PostgreSQL 风格的创建自定义函数语法。

```
CREATE [ OR REPLACE ] FUNCTION function_name
    ( [ { argname [ argmode ] argtype [ { DEFAULT | := | = } expression ]
} [, ...] ] )
```

```
    [ RETURNS rettype
        | RETURNS TABLE (  { column_name column_type  }  [, ...] )]
    [
        {IMMUTABLE  | STABLE  | VOLATILE}
        | {SHIPPABLE | NOT SHIPPABLE}
        | [ NOT  ] LEAKPROOF
        | WINDOW
        | {CALLED ON NULL INPUT | RETURNS NULL ON NULL INPUT | STRICT}
        | {[ EXTERNAL| SQL  ] SECURITY INVOKER  | [ EXTERNAL| SQL  ] SECURITY
DEFINER | AU
THID DEFINER  | AUTHID CURRENT_USER}
        | {FENCED | NOT FENCED}
        | {PACKAGE}
        | COST execution_cost
        | ROWS result_rows
        | SET configuration_parameter { {TO | =} value | FROM CURRENT }
        | COMMENT 'text'
        | {DETERMINISTIC | NOT DETERMINISTIC}
        | LANGUAGE lang_name
        | { CONTAINS SQL | NO SQL | READS SQL DATA | MODIFIES SQL DATA }
    ] [...]
    {
        AS 'definition'
        | AS 'obj_file', 'link_symbol'
    }
```

O 风格的创建自定义函数的语法。

```
CREATE [ OR REPLACE  ] FUNCTION function_name
    ( [  { argname [ argmode  ] argtype [  { DEFAULT | := | =  } expression  ] }
  [, ...]  ] )
    RETURN rettype
    [
        {IMMUTABLE  | STABLE  | VOLATILE }
        | {SHIPPABLE | NOT SHIPPABLE}
        | {PACKAGE}
        | [ NOT  ] LEAKPROOF
        | {CALLED ON NULL INPUT  | RETURNS NULL ON NULL INPUT  | STRICT }
        | {[ EXTERNAL| SQL  ] SECURITY INVOKER  | [ EXTERNAL| SQL  ] SECURITY
DEFINER | |
AUTHID DEFINER  | AUTHID CURRENT_USER}
        | COST execution_cost
```

```
        | ROWS result_rows
        | SET configuration_parameter { {TO | =} value | FROM CURRENT }
        | COMMENT 'text'
        | {DETERMINISTIC | NOT DETERMINISTIC}
        | LANGUAGE lang_name
        | { CONTAINS SQL | NO SQL | READS SQL DATA | MODIFIES SQL DATA }
    ][...]
    {
       IS | AS
    } plsql_body
/
```

MySQL 风格**语法格式**。

```
CREATE [ OR REPLACE  ] FUNCTION function_name
    ( [  { argname [ argmode  ] argtype [  { DEFAULT | := | =  } expression  ] }
  [, ...]  ] )
    RETURNS rettype
    [
        SQL SECURITY { DEFINER | INVOKER }
        | COMMENT 'text'
        | {DETERMINISTIC | NOT DETERMINISTIC}
        | LANGUAGE lang_name
        | { CONTAINS SQL | NO SQL | READS SQL DATA | MODIFIES SQL DATA }
    ][...] plsql_body
```

参数说明

LANGUAGE lang_name

用以实现函数的语言的名称。PostgreSQL 风格函数默认值 sql, O 风格默认值 plpgsql。MySQL 风格**语法格式**下，LANGUAGE 选项仅做语法兼容，可填入其他值，但最终将使用 plpgsql 作为实现函数的语言。在 MySQL 风格**语法格式**下，此选项允许重复。

SQL SECURITY INVOKER

表明该函数将带着调用它的用户的权限执行。该参数可以省略。

SQL SECURITY INVOKER 和 SECURITY INVOKER 和 AUTHID CURRENT_USER 的功能相同。

在 MySQL 风格**语法格式**下，此选项允许重复，且与 SQL SECURITY DEFINER 同类别。该类别的函数选项以最后一个输入为准。

SQL SECURITY DEFINER

声明该函数将以创建它的用户的权限执行。

SQL SECURITY DEFINER 和 AUTHID DEFINER 和 SECURITY DEFINER 的功能相同。

在 MySQL 风格**语法格式**下，此选项允许重复，且与 SQL SECURITY INVOKER 同类别。该类别的函数选项以最后一个输入为准。

CONTAINS SQL|NO SQL|READS SQL DATA|MODIFIES SQL DATA

语法兼容项。此选项允许重复。

示例

```
--指定 CONTAINS SQL
postgres=# CREATE FUNCTION func_test (s CHAR(20)) RETURNS int
CONTAINS SQL AS $$ select 1 $$ ;
--指定 DETERMINISTIC
postgres=# CREATE FUNCTION func_test (s int) RETURNS int
CONTAINS SQL DETERMINISTIC  AS $$ select s; $$ ;
--指定 LANGUAGE SQL
postgres=# CREATE FUNCTION func_test (s int) RETURNS int
CONTAINS SQL LANGUAGE SQL AS $$ select s; $$ ;
--指定 NO SQL
postgres=# CREATE FUNCTION func_test (s int) RETURNS int
NO SQL AS $$ select s; $$ ;
--指定  READS SQL DATA
postgres=# CREATE FUNCTION func_test (s int) RETURNS int
CONTAINS SQL  READS SQL DATA  AS $$ select s; $$ ;
--指定 MODIFIES SQL DATA
postgres=# CREATE FUNCTION func_test (s int) RETURNS int
CONTAINS SQL LANGUAGE SQL NO SQL  MODIFIES SQL DATA AS $$ select s; $$ ;
--指定 SECURITY DEFINER
postgres=# CREATE FUNCTION func_test (s int) RETURNS int
NO SQL SQL SECURITY DEFINER AS $$ select s; $$ ;
--指定 SECURITY INVOKER
postgres=# CREATE FUNCTION func_test (s int) RETURNS int
SQL SECURITY INVOKER  READS SQL DATA LANGUAGE SQL AS $$ select s; $$ ;
--MySQL 风格语法格式
postgres=# create function func(n int) returns varchar(50) return (select n+1);
CREATE FUNCTION
postgres=# select func(1);
 func
------
```

```
 2
(1 row)
postgres=# delimiter //
SET
postgres=# create function func10(b int) returns int
postgres-# begin
postgres-#     if b > 0 then return b + 10;
postgres-#     else return -1;
postgres-#     end if;
postgres-# end//
CREATE FUNCTION
postgres=# delimiter ;
SET
postgres=# select func10(9);
 func10
--------
     19
(1 row)
```

相关链接

CREATE FUNCTION

2.4.1.5.13 CREATE-INDEX

功能描述

在指定的表上创建索引。

索引可以用来提高数据库查询性能，但是不恰当的使用将导致数据库性能下降。建议仅在匹配如下某条原则时创建索引：

经常执行查询的字段。

在连接条件上创建索引，对于存在多字段连接的查询，建议在这些字段上建立组合索引。例如，select * from t1 join t2 on t1.a=t2.a and t1.b=t2.b，可以在 t1 表上的 a、b 字段上建立组合索引。

where 子句的过滤条件字段上（尤其是范围条件）。

在经常出现在 order by、group by 和 distinct 后的字段。

在分区表上创建索引与在普通表上创建索引的语法不太一样，使用时请注意，如分区表上不支持并行创建索引，不支持创建部分索引。

新增可以指定 ALGORITHM 选项语法。

注意事项

本章节只包含 dolphin 新增的语法，原 GBase 8c 的语法未做删除和修改。 新增支持 option 的无序排列。

语法格式

在表上创建索引。

```
CREATE [ UNIQUE | FULLTEXT ] INDEX [ CONCURRENTLY ] [ [schema_name.]index_name ]
    { ON table_name [ USING method ] | [ USING method ] ON table_name }
    ({ { column_name | ( expression ) } [ COLLATE collation ] [ opclass ] [ ASC
| DESC ] [ NULLS { FIRST | LAST } ] }[, ...] )
    [ index_option ]
    [ WHERE predicate | ALGORITHM [=] {DEFAULT | INPLACE | COPY} ];
CREATE [UNIQUE] INDEX index_name
    ON tbl_name (key_part,...)
    [USING {BTREE | HASH}]
```

在分区表上创建索引。

```
CREATE [ UNIQUE ] INDEX [ [schema_name.]index_name ]
    { ON table_name [ USING method ] | [ USING method ] ON table_name }
    ( {{ column_name | ( expression ) } [ COLLATE collation ] [ opclass ] [ ASC
| DESC ] [ NULLS LAST ] }[, ...] )
    [ LOCAL [ ( { PARTITION index_partition_name [ ( SUBPARTITION
index_subpartition_name [, ...] ) ] [ TABLESPACE index_partition_tablespace ] }
[, ...] ) ] | GLOBAL ]
    [ index_option ] [ALGORITHM [=] {DEFAULT | INPLACE | COPY} ]
```

参数说明

FULLTEXT

该关键字为创建兼容 MySQL 的全文索引的语法。该全文索引主要用于字符串的搜索匹配。包含局部匹配搜索，支持中文，韩文，日文。与 MATCH () AGAINST ()配合使用。

column_name (length)

创建一个基于该表一个字段的前缀键索引，column_name 为前缀键的字段名，length 为前缀长度。

前缀键将取指定字段数据的前缀作为索引键值，可以减少索引占用的存储空间。含有前缀键字段的过滤条件和连接条件可以使用索引。

说明：

前缀键支持的索引方法：btree、ubtree。

前缀键的字段的数据类型必须是二进制类型或字符类型（不包括特殊字符类型）。

前缀长度必须是不超过 2676 的正整数，并且不能超过字段的最大长度。对于二进制类型，前缀长度以字节数为单位。对于非二进制字符类型，前缀长度以字符数为单位。键值的实际长度受内部页面限制，若字段中含有多字节字符、或者一个索引上有多个键，索引行长度可能会超限，导致报错，设定较长的前缀长度时请考虑此情况。

CREATE INDEX 语法中，不支持以下关键字作为前缀键的字段名称：COALESCE、CONVERT、DAYOFMONTH、DAYOFWEEK、DAYOFYEAR、DB_B_FORMAT、EXTRACT、GREATEST、HOUR_P、IFNULL、LEAST、LOCATE、MICROSECOND_P、MID、MINUTE_P、NULLIF、NVARCHAR、NVL、OVERLAY、POSITION、QUARTER、SECOND_P、SUBSTR、SUBSTRING、TEXT_P、TIME、TIMESTAMP、TIMESTAMPDIFF、TREAT、TRIM、WEEKDAY、WEEKOFYEAR、XMLCONCAT、XMLELEMENT、XMLEXISTS、XMLFOREST、XMLPARSE、XMLPI、XMLROOT、XMLSERIALIZE。若含有上述关键字的前缀键所在的索引是通过 ALTER TABLE 或 CREATE TABLE 语法创建的，导出的 CREATE INDEX 语句可能无法成功执行，请尽量不要使用上述关键字作为前缀键的列名称。

index_option

创建索引时可指定选项，其语法为：

```
INCLUDE ( column_name [, ...] )
| WITH ( { storage_parameter = value } [, ...] )
| TABLESPACE tablespace_name
```

其中，TABLESPACE 选项允许输入多次，以最后一次的输入为准。

ALGORITHM

指定算法，可选项：DEFAULT、INPLACE、COPY。当前只做语法兼容，暂无实际功能。

示例

```
--创建表 tpcds.ship_mode_t1。
postgres=# create schema tpcds;
postgres=# CREATE TABLE tpcds.ship_mode_t1
(
```

```
    SM_SHIP_MODE_SK           INTEGER               NOT NULL,
    SM_SHIP_MODE_ID           CHAR(16)              NOT NULL,
    SM_TYPE                   CHAR(30)                      ,
    SM_CODE                   CHAR(10)                      ,
    SM_CARRIER                CHAR(20)                      ,
    SM_CONTRACT               CHAR(20)
)
;
--在表 tpcds.ship_mode_t1 上的 SM_SHIP_MODE_SK 字段上创建普通的唯一索引。
postgres=# CREATE UNIQUE INDEX ds_ship_mode_t1_index1 ON
tpcds.ship_mode_t1(SM_SHIP_MODE_SK);
--在表 tpcds.ship_mode_t1 上的 SM_SHIP_MODE_SK 字段上创建指定 B-tree 索引。
postgres=# CREATE INDEX ds_ship_mode_t1_index4 ON tpcds.ship_mode_t1 USING
btree(SM_SHIP_MODE_SK);
--在表 tpcds.ship_mode_t1 上 SM_CODE 字段上创建表达式索引。
postgres=# CREATE INDEX ds_ship_mode_t1_index2 ON
tpcds.ship_mode_t1(SUBSTR(SM_CODE,1 ,4));
--在表 tpcds.ship_mode_t1 上的 SM_SHIP_MODE_SK 字段上创建 SM_SHIP_MODE_SK 大于 10
的部分索引。
postgres=# CREATE UNIQUE INDEX ds_ship_mode_t1_index3 ON
tpcds.ship_mode_t1(SM_SHIP_MODE_SK) WHERE SM_SHIP_MODE_SK>10;
--重命名一个现有的索引。
postgres=# ALTER INDEX tpcds.ds_ship_mode_t1_index1 RENAME TO
ds_ship_mode_t1_index5;
--设置索引不可用。
postgres=# ALTER INDEX tpcds.ds_ship_mode_t1_index2 UNUSABLE;
--重建索引。
postgres=# ALTER INDEX tpcds.ds_ship_mode_t1_index2 REBUILD;
--删除一个现有的索引。
postgres=# DROP INDEX tpcds.ds_ship_mode_t1_index2;
--删除表。
postgres=# DROP TABLE tpcds.ship_mode_t1;
--创建表空间。
postgres=# CREATE TABLESPACE example1 RELATIVE LOCATION
'tablespace1/tablespace_1';
postgres=# CREATE TABLESPACE example2 RELATIVE LOCATION
'tablespace2/tablespace_2';
postgres=# CREATE TABLESPACE example3 RELATIVE LOCATION
'tablespace3/tablespace_3';
postgres=# CREATE TABLESPACE example4 RELATIVE LOCATION
'tablespace4/tablespace_4';
```

```
--创建表 tpcds.customer_address_p1。
postgres=# CREATE TABLE tpcds.customer_address_p1
(
    CA_ADDRESS_SK             INTEGER               NOT NULL,
    CA_ADDRESS_ID             CHAR(16)              NOT NULL,
    CA_STREET_NUMBER          CHAR(10)                      ,
    CA_STREET_NAME            VARCHAR(60)                   ,
    CA_STREET_TYPE            CHAR(15)                      ,
    CA_SUITE_NUMBER           CHAR(10)                      ,
    CA_CITY                   VARCHAR(60)                   ,
    CA_COUNTY                 VARCHAR(30)                   ,
    CA_STATE                  CHAR(2)                       ,
    CA_ZIP                    CHAR(10)                      ,
    CA_COUNTRY                VARCHAR(20)                   ,
    CA_GMT_OFFSET             DECIMAL(5,2)                  ,
    CA_LOCATION_TYPE          CHAR(20)
)
TABLESPACE example1
PARTITION BY RANGE(CA_ADDRESS_SK)
(
   PARTITION p1 VALUES LESS THAN (3000),
   PARTITION p2 VALUES LESS THAN (5000) TABLESPACE example1,
   PARTITION p3 VALUES LESS THAN (MAXVALUE) TABLESPACE example2
)
ENABLE ROW MOVEMENT;
--创建分区表索引 ds_customer_address_p1_index1，不指定索引分区的名称。
postgres=# CREATE INDEX ds_customer_address_p1_index1 ON
tpcds.customer_address_p1(CA_ADDRESS_SK) LOCAL;
--创建分区表索引 ds_customer_address_p1_index2，并指定索引分区的名称。
postgres=# CREATE INDEX ds_customer_address_p1_index2 ON
tpcds.customer_address_p1(CA_ADDRESS_SK) LOCAL
(
    PARTITION CA_ADDRESS_SK_index1,
    PARTITION CA_ADDRESS_SK_index2 TABLESPACE example3,
    PARTITION CA_ADDRESS_SK_index3 TABLESPACE example4
)
TABLESPACE example2;
--创建 GLOBAL 分区索引
postgres=#CREATE INDEX ds_customer_address_p1_index3 ON
tpcds.customer_address_p1(CA_ADDRESS_ID) GLOBAL;
--不指定关键字，默认创建 GLOBAL 分区索引
```

```
postgres=#CREATE INDEX ds_customer_address_p1_index4 ON
tpcds.customer_address_p1(CA_ADDRESS_ID);
--修改分区表索引 CA_ADDRESS_SK_index2 的表空间为 example1。
postgres=# ALTER INDEX tpcds.ds_customer_address_p1_index2 MOVE PARTITION
CA_ADDRESS_SK_index2 TABLESPACE example1;
--修改分区表索引 CA_ADDRESS_SK_index3 的表空间为 example2。
postgres=# ALTER INDEX tpcds.ds_customer_address_p1_index2 MOVE PARTITION
CA_ADDRESS_SK_index3 TABLESPACE example2;
--重命名分区表索引。
postgres=# ALTER INDEX tpcds.ds_customer_address_p1_index2 RENAME PARTITION
CA_ADDRESS_SK_index1 TO CA_ADDRESS_SK_index4;
--删除索引和分区表。
postgres=# DROP INDEX tpcds.ds_customer_address_p1_index1;
postgres=# DROP INDEX tpcds.ds_customer_address_p1_index2;
postgres=# DROP TABLE tpcds.customer_address_p1;
--删除表空间。
postgres=# DROP TABLESPACE example1;
postgres=# DROP TABLESPACE example2;
postgres=# DROP TABLESPACE example3;
postgres=# DROP TABLESPACE example4;
--创建列存表以及列存表 GIN 索引。
postgres=# create table cgin_create_test(a int, b text) with (orientation =
column);
CREATE TABLE
postgres=# create index cgin_test on cgin_create_test using
gin(to_tsvector('ngram', b));
CREATE INDEX
##全文索引
postgres=# CREATE SCHEMA fulltext_test;
CREATE SCHEMA
postgres=# set current_schema to 'fulltext_test';
SET
postgres=# CREATE TABLE test (
id int unsigned auto_increment not null primary key,
title varchar,
boby text,
name name
);
NOTICE:  CREATE TABLE will create implicit sequence "test_id_seq" for serial
column "test.id"
```

```
NOTICE:  CREATE TABLE / PRIMARY KEY will create implicit index "test_pkey" for
table "test"
CREATE TABLE
postgres=# \d test
             Table "fulltext_test.test"
 Column |       Type        |        Modifiers
--------+-------------------+-------------------------
 id     | uint4             | not null AUTO_INCREMENT
 title  | character varying |
 boby   | text              |
 name   | name              |
Indexes:
    "test_pkey" PRIMARY KEY, btree (id) TABLESPACE pg_default
postgres=# CREATE FULLTEXT INDEX test_index_1 ON test (title, boby) WITH PARSER
ngram;
\d test_index_1
                  Index "fulltext_test.test_index_1"
    Column     | Type |                    Definition
---------------+------+-----------------------------------------------
 to_tsvector  | text | to_tsvector('"ngram"'::regconfig, title::text)
 to_tsvector1 | text | to_tsvector('"ngram"'::regconfig, boby)
gin, for table "fulltext_test.test"
postgres=# CREATE FULLTEXT INDEX test_index_2 ON test (title, boby, name);
CREATE INDEX
```

相关链接

CREATE INDEX

2.4.1.5.14 CREATE-PROCEDURE

功能描述

创建一个新的存储过程。

注意事项

相比于原始的 GBase 8c，dolphin 对于 CREATE PROCEDURE 语法的修改为：

增加 LANGUAGE 选项。

增加语法兼容项 [NOT] DETERMINISTIC。

增加语法兼容项 { CONTAINS SQL | NO SQL | READS SQL DATA | MODIFIES SQL DATA } 。

增加语法兼容项 SQL SECURITY { DEFINER | INVOKER }。

兼容 MySQL 的创建存储过程的语法格式

兼容创建存储过程紧跟单条查询语句

语法格式

GBase 8c 原始创建存储过程的语法。

```
CREATE [ OR REPLACE ] PROCEDURE procedure_name
    [ ( {[ argname ] [ argmode ] argtype [ { DEFAULT | := | = } expression ]}[,...]) ]
    [
       { IMMUTABLE | STABLE | VOLATILE }
       | { SHIPPABLE | NOT SHIPPABLE }
       | {PACKAGE}
       | [ NOT ] LEAKPROOF
       | { CALLED ON NULL INPUT | RETURNS NULL ON NULL INPUT | STRICT }
       | {[ EXTERNAL |SQL ] SECURITY INVOKER | [ EXTERNAL|SQL ] SECURITY DEFINER
| AUTHID DEFINER | AUTHID CURRENT_USER}
       | COST execution_cost
       | SET configuration_parameter { TO value | = value | FROM CURRENT }
       | COMMENT text
       | {DETERMINISTIC | NOT DETERMINISTIC}
       | LANGUAGE lang_name
       | { CONTAINS SQL | NO SQL | READS SQL DATA | MODIFIES SQL DATA }
    ][ ... ]
 { IS | AS }
plsql_body
/
```

使用 MySQL 的格式进行创建存储过程。

注意：使用 MMySQL 的格式创建时，需要在客户端使用 delimiter 命令设置结束符。

```
CREATE [ OR REPLACE ] PROCEDURE procedure_name
    ( [ {[ argname ] [ argmode ] argtype [ { DEFAULT | := | = }
expression ]}[,...] ] )
    [
       { IMMUTABLE | STABLE | VOLATILE }
       | { SHIPPABLE | NOT SHIPPABLE }
       | {PACKAGE}
       | [ NOT ] LEAKPROOF
       | { CALLED ON NULL INPUT | RETURNS NULL ON NULL INPUT | STRICT }
       | {[ EXTERNAL |SQL ] SECURITY INVOKER | [ EXTERNAL|SQL ] SECURITY DEFINER
| AUTHID DEFINER | AUTHID CURRENT_USER}
```

```
      | COST execution_cost
      | SET configuration_parameter { TO value | = value | FROM CURRENT }
      | COMMENT text
      | {DETERMINISTIC | NOT DETERMINISTIC}
      | LANGUAGE lang_name
      | { CONTAINS SQL | NO SQL | READS SQL DATA | MODIFIES SQL DATA }
   ][ ... ]
  routine_body
```

创建存储过程紧跟单条查询语句。

```
CREATE [ OR REPLACE ] PROCEDURE procedure_name
    ( [ {[ argname ] [ argmode ] argtype [ { DEFAULT | := | = }
expression ]}[,...] ] )
    [
       { IMMUTABLE | STABLE | VOLATILE }
       | { SHIPPABLE | NOT SHIPPABLE }
       | {PACKAGE}
       | [ NOT ] LEAKPROOF
       | { CALLED ON NULL INPUT | RETURNS NULL ON NULL INPUT | STRICT }
       | {[ EXTERNAL |SQL ] SECURITY INVOKER | [ EXTERNAL|SQL ] SECURITY DEFINER
| AUTHID DEFINER | AUTHID CURRENT_USER}
       | COST execution_cost
       | SET configuration_parameter { TO value | = value | FROM CURRENT }
       | COMMENT text
       | {DETERMINISTIC | NOT DETERMINISTIC}
       | LANGUAGE lang_name
       | { CONTAINS SQL | NO SQL | READS SQL DATA | MODIFIES SQL DATA }
    ][ ... ]
   select_stmt
```

参数说明

LANGUAGE lang_name

用以实现存储过程的语言的名称。默认值 plpgsql。

SQL SECURITY INVOKER

表明该存储过程将带着调用它的用户的权限执行。该参数可以省略。

SQL SECURITY INVOKER 和 SECURITY INVOKER 和 AUTHID CURRENT_USER 的功能相同。

SQL SECURITY DEFINER

声明该存储过程将以创建它的用户的权限执行。

SQL SECURITY DEFINER 和 AUTHID DEFINER 和 SECURITY DEFINER 的功能相同。

CONTAINS SQL|NO SQL|READS SQL DATA|MODIFIES SQL DATA

语法兼容项。

示例

```
--创建存储过程使用单条查询语句，显示为 CREATE FUNCTION
postgres=#  create procedure procxx() select a from t1;
CREATE FUNCTION
--结果集类似 RETURNS SETOF RECORD
postgres=# select procxx();
 procxx
--------
 (1)
 (2)
(2 rows)
```

相关链接

CREATE PROCEDURE

2.4.1.5.15 CREATE-SERVER

功能描述

定义一个新的外部服务器。

注意事项

本章节只包含 dolphin 新增的语法，原 GBase 8c 的语法未做删除和修改。

相比于原始的 GBase 8c，dolphin 对于 CREATE SERVER 语法的修改主要为：

新增 fdw_name 可选值 mysql，其功能与 mysql_fdw 一致。

对于 fdw_name 为 mysql_fdw 时，增加可选 OPTIONS：DATABASE, USER, PASSWORD, SOCKET, OWNER。

语法格式

```
CREATE SERVER server_name
    FOREIGN DATA WRAPPER fdw_name
    OPTIONS ( { option_name ' value ' } [, ...] ) ;
```

参数说明

fdw_name

指定外部数据封装器的名称。

取值范围：dist_fdw，hdfs_fdw，log_fdw，file_fdw，mot_fdw，oracle_fdw，mysql_fdw，mysql, postgres_fdw。

OPTIONS ({ option_name ' value ' } [, …])

这个子句为服务器指定选项。这些选项通常定义该服务器的连接细节，但是实际的名称和值取决于该服务器的外部数据包装器。

mysql_fdw 支持的 options 包括：

host（默认值为 127.0.0.1）

MySQL Server/MariaDB 的地址。

port（默认值为 3306）

MySQL Server/MariaDB 侦听的端口号。

user（默认为空）

MySQL Server/MariaDB 用于连接的用户名。若 OPTIONS 指定此选项，GBase 8c 将自动创建当前用户到新建 server 的用户映射。

password（默认为空）

MySQL Server/MariaDB 用于连接的用户密码。若 OPTIONS 指定此选项，GBase 8c 将自动创建当前用户到新建 server 的用户映射。

database（默认为空）

无实际意义，仅做语法兼容。指定 MySQL Server/MariaDB 连接的数据库请在 CREATE FOREIGN TABLE 或 ALTER FOREIGN TABLE 中完成。

owner（默认为空）

无实际意义，仅做语法兼容。

socket（默认为空）

无实际意义，仅做语法兼容。

示例

```
创建 server。
postgres=# create server server_test foreign data wrapper mysql options(host
'192.108.0.1', port '3306', user 'foreign_server_test',
```

```
password 'password@123', database 'my_db', owner 'test_user');
WARNING:  Option database will be deprecated for CREATE SERVER.
WARNING:  Option owner will be deprecated for CREATE SERVER.
WARNING:  USER MAPPING for current user to server server_test created.
CREATE SERVER
```

相关链接

ALTER SERVER，DROP SERVER

2.4.1.5.16 CREATE-TABLE

功能描述

在当前数据库中创建一个新的空白表，该表由命令执行者所有。

注意事项

本章节只包含 dolphin 新增的语法，原 GBase 8c 的语法未做删除和修改。

语法格式

通过无括号 like 创建表。

```
CREATE [ [ GLOBAL | LOCAL ] [ TEMPORARY | TEMP ] | UNLOGGED ] TABLE [ IF NOT EXISTS ]
table_name LIKE source_table [ like_option [...] ]
```

like 后不能添加普通建表的额外可选语句。

table 前不能添加 foreign 选项，包括外表、mot 表的创建。

默认复制源表的索引，若不希望复制索引，需要手动指定 EXCLUDING INDEXES。

默认复制源分区表的分区，若不希望复制分区，需要手动指定 EXCLUDING PARTITION。

对于含索引的分区表，若只指定 EXCLUDING PARTITION，由于默认复制分区，将会报错，因为普通表不支持分区索引。

只支持复制 range 分区表的分区，对于 hash、list 分区表，由于默认复制分区，会直接报错，需要手动指定 EXCLUING PARTITION。二级分区只支持复制 range-range 分区，处理方法同上。

创建表。

```
CREATE [ [ GLOBAL | LOCAL ] [ TEMPORARY | TEMP ] | UNLOGGED ] TABLE [ IF NOT EXISTS ]
table_name
    ({ column_name data_type [ compress_mode ] [ COLLATE collation ]
[ column_constraint [ ... ] ]
```

```
      | table_constraint
      | table_indexclause
      | LIKE source_table [ like_option [...] ] }
      [, ... ])
  [ AUTO_INCREMENT [ = ] value ]
  [ WITH ( {storage_parameter = value} [, ... ] ) ]
  [ ON CgbaseIT { PRESERVE ROWS | DELETE ROWS | DROP } ]
  [ COMPRESS | NOCOMPRESS ]
  [ TABLESPACE tablespace_name ]
  [ COMMENT {=| } 'text' ];
  [ create_option ]
```

其中 create_option 为：

```
      [ WITH ( {storage_parameter = value} [, ... ] ) ]
      [ ON CgbaseIT { PRESERVE ROWS | DELETE ROWS | DROP } ]
      [ COMPRESS | NOCOMPRESS ]
      [ TABLESPACE tablespace_name ]
      [ COMPRESSION [=] compression_arg ]
      [ ENGINE [=] engine_name ]
      [ COLLATE [=] collation_name ]
      [ [DEFAULT] { CHARSET | CHARACTER SET } [=] charset_name ]
      [ ROW_FORMAT [=] row_format_name ]
      [ AUTOEXTEND_SIZE [=] value ]
      [ AVG_ROW_LENGTH [=] value ]
      [ CHECKSUM [=] value ]
      [ CONNECTION [=] 'connect_string' ]
      [ {DATA | INDEX} DIRECTORY [=] 'absolute path to directory' ]
      [ DELAY_KEY_WRITE [=] value ]
      [ ENCRYPTION [=] 'encryption_string' ]
      [ ENGINE_ATTRIBUTE [=] 'string' ]
      [ INSERT_METHOD [=] { NO | FIRST | LAST } ]
      [ KEY_BLOCK_SIZE [=] value ]
      [ MAX_ROWS [=] value ]
      [ MIN_ROWS [=] value ]
      [ PACK_KEYS [=] value ]
      [ PASSWORD [=] 'password' ]
      [ START TRANSACTION ]
      [ SECONDARY_ENGINE_ATTRIBUTE [=] 'string' ]
      [ STATS_AUTO_RECALC [=] value ]
      [ STATS_PERSISTENT [=] value ]
      [ STATS_SAMPLE_PAGES [=] value ]
```

```
        [ UNION [=] (tbl_name[,tbl_name]...) ]
        [ TABLESPACE tablespace_name STORAGE DISK ]
        [ [TABLESPACE tablespace_name] STORAGE MEMORY ]
```

除了 WITH 选项外允许输入多次同一种 create_option，以最后一次的输入为准。

创建表上索引 table_indexclause：

```
""{[FULLTEXT] INDEX | KEY} [index_name] [index_type]
(key_part,...)[index_option]...
```

该语法不支持 CREATE FOREIGN TABLE (MOT 表等) 创建。

其中参数 index_type 为：

```
USING {BTREE | HASH | GIN | GIST | PSORT | UBTREE}
```

其中参数 key_part 为：

```
{col_name[(length)] | (expr)} [ASC | DESC]
```

length 为前缀索引。

其中参数 index_option 为：

```
index_option:{
     COMMENT 'string'
   | index_type
   | [ VISIBLE | INVISIBLE ]
  | [WITH PARSER NGRAM]
}
```

COMMENT、index_type、[VISIBLE | INVISIBLE] 的顺序和数量任意，但相同字段仅最后一个值生效。WITH PARSER NGRAM 为 FULLTEXT INDEX 指定的 ngram 解析器，前提是索引必须指定关键字 FULLTEXT，FULLTEXT 默认 WITH PARSER NGRAM。

其中 like 选项 like_option 为：

```
{ INCLUDING | EXCLUDING } { DEFAULTS | GENERATED | CONSTRAINTS | INDEXES | STORAGE
| COMMENTS | PARTITION | RELOPTIONS | ALL }
```

参数说明

data_type

字段的数据类型。

对枚举类型 ENUM，以及 CHAR, CHARACTER, VARCHAR, TEXT 等字符类型，创建表格时可使用关键字 CHARSET 或 CHARACTER SET 声明列字符集。目前该特性仅做语法

支持，不实现功能。

column_constraint

字段的类型约束中，添加了 mysql 的 ON UPDATE 特性，归类于字段类型约束。与 DEFAULT 属性属于同类约束。该 ON UPDATE 属性用于，执行 UPDATE 操作 timestamp 字段为缺省时，则自动更新 timestamp 字段的时间截。如果更新字段的数据内容与原来的数据内容一致，则其他含有 ON UPDATE 的字段的时间截不会自动更新。

CREATE TABLE table_name(column_name timestamp ON UPDATE CURRENT_TIMESTAMP);

COLLATE collation

COLLATE 子句指定列的排序规则（该列必须是可排列的数据类型）。如果没有指定，则使用默认的排序规则。排序规则可以使用 **“**select * from pg_collation;**”** 命令从 pg_collation 系统表中查询，默认的排序规则为查询结果中以 default 开始的行。

对未被支持的排序规则，数据库将发出警告，并将该列设置为默认的排序规则。

{ [DEFAULT] CHARSET | CHARACTER SET } [=] charset_name

用于选择表所使用的字符集；目前该特性仅有语法支持，不实现功能。

COLLATE [=] collation_name

用于选择表所使用的排序规则；目前该特性仅有语法支持，不实现功能。

ROW_FORMAT [=] row_format_name

用于选择表所使用的行存储格式；目前该特性仅有语法支持，不实现功能。

AUTO_INCREMENT

该关键字将字段指定为自动增长列。自动增长列必须是某个索引的第一个字段。

AUTOEXTEND_SIZE [=] value

用于指定在表空间变满时扩展表空间大小；目前该特性仅有语法支持，不实现功能。参数的取值范围包括非负整数，小数，标识符，非负整数+标识符，小数+标识符。

AVG_ROW_LENGTH [=] value

用于指定表的平均行长度；目前该特性仅有语法支持，不实现功能。参数的取值范围包括非负整数，小数。

CHECKSUM [=] value

用于指定是否维护所有行的实时校验和；目前该特性仅有语法支持，不实现功能。参数的取值范围为非负整数，小数，十六进制数。

CONNECTION [=] 'connect_string'

用于指定联合表的连接字符串；目前该特性仅有语法支持，不实现功能。参数的取值范围为任意字符串。

{DATA | INDEX} DIRECTORY [=] 'absolute path to directory'

用于指定表数据数据和索引的存储目录；目前该特性仅有语法支持，不实现功能。参数的取值范围为任意字符串。

DELAY_KEY_WRITE [=] value

用于指定是否延迟表的索引更新直到表关闭；目前该特性仅有语法支持，不实现功能。参数的取值范围为非负整数，小数，十六进制数。

ENCRYPTION [=] 'encryption_string'

用于指定表启用或禁用页面级数据加密；目前该特性仅有语法支持，不实现功能。参数的取值范围为任意字符串。

ENGINE_ATTRIBUTE [=] 'string'

用于指定主存储引擎的表属性；目前该特性仅有语法支持，不实现功能。参数的取值范围为任意字符串。

INSERT_METHOD [=] { NO | FIRST | LAST }

用于指定应将行插入到的表；目前该特性仅有语法支持，不实现功能。参数的取值范围为 NO，FIRST，LAST。

KEY_BLOCK_SIZE [=] value

用于指定索引键块的字节大小；目前该特性仅有语法支持，不实现功能。参数的取值范围为非负整数，小数。

MAX_ROWS [=] value

用于指定计划在表中存储的最大行数；目前该特性仅有语法支持，不实现功能。参数的取值范围为非负整数，小数。

MIN_ROWS [=] value

用于指定计划在表中存储的最小行数；目前该特性仅有语法支持，不实现功能。参数的

取值范围为非负整数，小数。

PACK_KEYS [=] value

用于指定控制压缩索引的方式；目前该特性仅有语法支持，不实现功能。参数的取值范围为非负整数，小数，十六进制数，DEFAULT。

PASSWORD [=] 'password'

此选项未使用；目前该特性仅有语法支持，不实现功能。参数的取值范围为任意字符串。

SECONDARY_ENGINE_ATTRIBUTE [=] 'string'

用于指定辅助存储引擎的表属性；目前该特性仅有语法支持，不实现功能。参数的取值范围为任意字符串。

START TRANSACTION

用于开启事务模式；目前该特性仅有语法支持，不实现功能。

STATS_AUTO_RECALC [=] value

用于指定是否自动重新计算表的持久统计信息；目前该特性仅有语法支持，不实现功能。参数的取值范围为非负整数，小数，十六进制数，DEFAULT。

STATS_PERSISTENT [=] value

用于指定是否为表启用持久统计信息；目前该特性仅有语法支持，不实现功能。参数的取值范围为非负整数，小数，十六进制数，DEFAULT。

STATS_SAMPLE_PAGES [=] value

用于指定估计索引列的基数和其他统计信息时要采样的索引页数；目前该特性仅有语法支持，不实现功能。参数的取值范围为非负整数，小数，十六进制数。

UNION [=] (tbl_name[,tbl_name]•••)

用于访问一组相同的表作为一个表；目前该特性仅有语法支持，不实现功能。

TABLESPACE tablespace_name STORAGE DISK

用于指定表存储在磁盘；目前该特性仅有语法支持，不实现功能。

[TABLESPACE tablespace_name] STORAGE MEMORY

用于指定表存储在内存；目前该特性仅有语法支持，不实现功能。

示例

```
--创建表上索引
postgres=# CREATE TABLE tpcds.warehouse_t24
(
    W_WAREHOUSE_SK            INTEGER               NOT NULL,
    W_WAREHOUSE_ID            CHAR(16)              NOT NULL,
    W_WAREHOUSE_NAME          VARCHAR(20)                   ,
    W_WAREHOUSE_SQ_FT         INTEGER                       ,
    W_STREET_NUMBER           CHAR(10)                      ,
    W_STREET_NAME             VARCHAR(60)                   ,
    W_STREET_TYPE             CHAR(15)                      ,
    W_SUITE_NUMBER            CHAR(10)                      ,
    W_CITY                    VARCHAR(60)                   ,
    W_COUNTY                  VARCHAR(30)                   ,
    W_STATE                   CHAR(2)                       ,
    W_ZIP                     CHAR(10)                      ,
    W_COUNTRY                 VARCHAR(20)                   ,
    W_GMT_OFFSET              DECIMAL(5,2)                  ,
    key (W_WAREHOUSE_SK)                                    ,
    index idx_ID using btree (W_WAREHOUSE_ID)
);
--创建表上组合索引、表达式索引、函数索引
postgres=# CREATE TABLE tpcds.warehouse_t25
(
    W_WAREHOUSE_SK            INTEGER               NOT NULL,
    W_WAREHOUSE_ID            CHAR(16)              NOT NULL,
    W_WAREHOUSE_NAME          VARCHAR(20)                   ,
    W_WAREHOUSE_SQ_FT         INTEGER                       ,
    W_STREET_NUMBER           CHAR(10)                      ,
    W_STREET_NAME             VARCHAR(60)                   ,
    W_STREET_TYPE             CHAR(15)                      ,
    W_SUITE_NUMBER            CHAR(10)                      ,
    W_CITY                    VARCHAR(60)                   ,
    W_COUNTY                  VARCHAR(30)                   ,
    W_STATE                   CHAR(2)                       ,
    W_ZIP                     CHAR(10)                      ,
    W_COUNTRY                 VARCHAR(20)                   ,
    W_GMT_OFFSET              DECIMAL(5,2)                  ,
    key using btree (W_WAREHOUSE_SK, W_WAREHOUSE_ID desc)   ,
    index idx_SQ_FT using btree ((abs(W_WAREHOUSE_SQ_FT)))  ,
    key idx_SK using btree ((abs(W_WAREHOUSE_SK)+1))
);
```

```
--创建带 INVISIBLE 普通索引的表
postgres=# CREATE TABLE tpcds.warehouse_t26
(
    W_WAREHOUSE_SK            INTEGER               NOT NULL,
    W_WAREHOUSE_ID            CHAR(16)              NOT NULL,
    W_WAREHOUSE_NAME          VARCHAR(20)                   ,
    W_WAREHOUSE_SQ_FT         INTEGER                       ,
    W_STREET_NUMBER           CHAR(10)                      ,
    W_STREET_NAME             VARCHAR(60)                   ,
    W_STREET_TYPE             CHAR(15)                      ,
    W_SUITE_NUMBER            CHAR(10)                      ,
    W_CITY                    VARCHAR(60)                   ,
    W_COUNTY                  VARCHAR(30)                   ,
    W_STATE                   CHAR(2)                       ,
    W_ZIP                     CHAR(10)                      ,
    W_COUNTRY                 VARCHAR(20)                   ,
    W_GMT_OFFSET              DECIMAL(5,2)                  ,
    index idx_ID using btree (W_WAREHOUSE_ID) INVISIBLE
);
--包含 index_option 字段
postgres=# create table test_option(a int, index idx_op using btree(a) cgbaseent
'idx cgbaseent');
--创建表格时对列指定字符集。
postgres=# CREATE TABLE t_column_charset(c text CHARSET test_charset);
WARNING:  character set "test_charset" for type text is not supported yet. default
value set
CREATE TABLE
--创建表格时对表格指定字符序。
postgres=# CREATE TABLE t_table_collate(c text) COLLATE test_collation;
WARNING:  COLLATE for TABLE is not supported for current version. skipped
CREATE TABLE
--创建表格时对表格指定字符集。
postgres=# CREATE TABLE t_table_charset(c text) CHARSET test_charset;
WARNING:  CHARSET for TABLE is not supported for current version. skipped
CREATE TABLE
--创建表格时对表格指定行记录格式。
postgres=# CREATE TABLE t_row_format(c text) ROW_FORMAT test_row_format;
WARNING:  ROW_FORMAT for TABLE is not supported for current version. skipped
CREATE TABLE
--创建表时对表指定在表空间变满时扩展表空间大小。
postgres=# CREATE TABLE t_autoextend_size(c text) AUTOEXTEND_SIZE 4M;
```

```
WARNING:  AUTOEXTEND_SIZE for TABLE is not supported for current version. skipped
CREATE TABLE
--创建表时对表指定表的平均行长度。
postgres=# CREATE TABLE t_avg_row_length(c text) AVG_ROW_LENGTH 10;
WARNING:  AVG_ROW_LENGTH for TABLE is not supported for current version. skipped
CREATE TABLE
--创建表时对表指定是否维护所有行的实时校验和。
postgres=# CREATE TABLE t_checksum(c text) CHECKSUM 0;
WARNING:  CHECKSUM for TABLE is not supported for current version. skipped
CREATE TABLE
--创建表时对表指定联合表的连接字符串。
postgres=# CREATE TABLE t_connection(c text) CONNECTION 'connect_string';
WARNING:  CONNECTION for TABLE is not supported for current version. skipped
CREATE TABLE
--创建表时对表指定表数据数据和索引的存储目录。
postgres=# CREATE TABLE t_data_directory(c text) DATA DIRECTORY
'data_directory';
WARNING:  DIRECTORY for TABLE is not supported for current version. skipped
CREATE TABLE
postgres=# CREATE TABLE t_index_directory(c text) INDEX DIRECTORY
'index_directory';
WARNING:  DIRECTORY for TABLE is not supported for current version. skipped
CREATE TABLE
--创建表时对表指定是否延迟表的索引更新直到表关闭。
postgres=# CREATE TABLE t_delay_key_write(c text) DELAY_KEY_WRITE 1;
WARNING:  DELAY_KEY_WRITE for TABLE is not supported for current version. skipped
CREATE TABLE
--创建表时对表指定表启用或禁用页面级数据加密。
postgres=# CREATE TABLE t_encryption(c text) ENCRYPTION 'Y';
WARNING:  ENCRYPTION for TABLE is not supported for current version. skipped
CREATE TABLE
--创建表时对表指定主存储引擎的表属性。
postgres=# CREATE TABLE t_engine_attribute(c text) ENGINE_ATTRIBUTE
'engine_attribute';
WARNING:  ENGINE_ATTRIBUTE for TABLE is not supported for current version.
skipped
CREATE TABLE
--创建表时对表指定应将行插入到的表。
postgres=# CREATE TABLE t_insert_method(c text) INSERT_METHOD NO;
WARNING:  INSERT_METHOD for TABLE is not supported for current version. skipped
CREATE TABLE
```

```
--创建表时对表指定索引键块的字节大小。
postgres=# CREATE TABLE t_key_block_size(c text) KEY_BLOCK_SIZE 10;
WARNING:  KEY_BLOCK_SIZE for TABLE is not supported for current version. skipped
CREATE TABLE
--创建表时对表指定计划在表中存储的最大行数。
postgres=# CREATE TABLE t_max_rows(c text) MAX_ROWS 20;
WARNING:  MAX_ROWS for TABLE is not supported for current version. skipped
CREATE TABLE
--创建表时对表指定计划在表中存储的最小行数。
postgres=# CREATE TABLE t_min_rows(c text) MIN_ROWS 5;
WARNING:  MIN_ROWS for TABLE is not supported for current version. skipped
CREATE TABLE
--创建表时对表指定控制压缩索引的方式。
postgres=# CREATE TABLE t_pack_keys(c text) PACK_KEYS DEFAULT;
WARNING:  PACK_KEYS for TABLE is not supported for current version. skipped
CREATE TABLE
postgres=# CREATE TABLE t_password(c text) PASSWORD 'password';
WARNING:  PASSWORD for TABLE is not supported for current version. skipped
CREATE TABLE
--创建表时对表指定开启事务模式。
postgres=# CREATE TABLE t_start_transaction(c text) START TRANSACTION;
WARNING:  START TRANSACTION for TABLE is not supported for current version.
skipped
CREATE TABLE
--创建表时对表指定辅助存储引擎的表属性。
postgres=# CREATE TABLE t_secondary_engine_attribute(c text)
SECONDARY_ENGINE_ATTRIBUTE 'secondary_engine_attribute';
WARNING:  SECONDARY_ENGINE_ATTRIBUTE for TABLE is not supported for current
version. skipped
CREATE TABLE
--创建表时对表指定是否自动重新计算表的持久统计信息。
postgres=# CREATE TABLE t_stats_auto_recalc(c text) STATS_AUTO_RECALC DEFAULT;
WARNING:  STATS_AUTO_RECALC for TABLE is not supported for current version.
skipped
CREATE TABLE
--创建表时对表指定是否为表启用持久统计信息。
postgres=# CREATE TABLE t_stats_persistent(c text) STATS_PERSISTENT DEFAULT;
WARNING:  STATS_PERSISTENT for TABLE is not supported for current version.
skipped
CREATE TABLE
--创建表时对表指定估计索引列的基数和其他统计信息时要采样的索引页数。
```

```
postgres=# CREATE TABLE t_stats_sample_pages(c text) STATS_SAMPLE_PAGES 1;
WARNING:  STATS_SAMPLE_PAGES for TABLE is not supported for current version.
skipped
CREATE TABLE
--创建表时访问一组相同的表作为一个表。
postgres=# CREATE TABLE t_union(c text) UNION(a, b);
WARNING:  UNION for TABLE is not supported for current version. skipped
CREATE TABLE
--创建表时对表指定表存储在磁盘。
postgres=# CREATE TABLESPACE test ADD DATAFILE 'data.ibd';
WARNING:  Suffix ".ibd" of datafile path detected. The actual path will be renamed
as "data_ibd"
CREATE TABLESPACE
postgres=# CREATE TABLE t_tablespace_storage_disk(c text) TABLESPACE test
STORAGE DISK;
WARNING:  TABLESPACE_OPTION for TABLE is not supported for current version.
skipped
CREATE TABLE
--创建表时对表指定表存储在内存。
postgres=# CREATE TABLESPACE test ADD DATAFILE 'data.ibd';
WARNING:  Suffix ".ibd" of datafile path detected. The actual path will be renamed
as "data_ibd"
CREATE TABLESPACE
postgres=# CREATE TABLE t_tablespace_storage_memory(c text) TABLESPACE test
STORAGE MEMORY;
WARNING:  TABLESPACE_OPTION for TABLE is not supported for current version.
skipped
CREATE TABLE
 -创建兼容 MySQL 全文索引语法的表。前提是兼容模式为 B 的数据库。
postgres=# CREATE TABLE test (
postgres(# id int unsigned auto_increment not null primary key,
postgres(# title varchar,
postgres(# boby text,
postgres(# name name,
postgres(# FULLTEXT (title, boby) WITH PARSER ngram
postgres(# );
NOTICE:  CREATE TABLE will create implicit sequence "test_id_seq" for serial
column "test.id"
NOTICE:  CREATE TABLE / PRIMARY KEY will create implicit index "test_pkey" for
table "test"
CREATE TABLE
```

```
postgres=# drop table if exists articles;
NOTICE:  table "articles" does not exist, skipping
DROP TABLE
postgres=# CREATE TABLE articles (
postgres(# ID int,
postgres(# title VARCHAR(100),
postgres(# FULLTEXT INDEX ngram_idx(title)WITH PARSER ngram
postgres(# );
CREATE TABLE
postgres=# \d articles
       Table "fulltext_test.articles"
 Column |          Type          | Modifiers
--------+------------------------+-----------
 ID     | integer                |
 title  | character varying(100) |
Indexes:
    "ngram_idx" gin (to_tsvector('ngram'::regconfig, title::text)) TABLESPACE
pg_default
postgres=# drop table if exists articles;
DROP TABLE
postgres=# CREATE TABLE articles (
postgres(# ID int,
postgres(# title VARCHAR(100),
postgres(# FULLTEXT INDEX (title)WITH PARSER ngram
postgres(# );
CREATE TABLE
postgres=# \d articles
       Table "fulltext_test.articles"
 Column |          Type          | Modifiers
--------+------------------------+-----------
 ID     | integer                |
 title  | character varying(100) |
Indexes:
    "articles_to_tsvector_idx" gin (to_tsvector('ngram'::regconfig,
title::text)) TABLESPACE pg_default
postgres=# drop table if exists articles;
DROP TABLE
postgres=# CREATE TABLE articles (
postgres(# ID int,
postgres(# title VARCHAR(100),
postgres(# FULLTEXT KEY keyngram_idx(title)WITH PARSER ngram
```

```
postgres(# );
CREATE TABLE
postgres=# \d articles
       Table "fulltext_test.articles"
 Column |          Type          | Modifiers
--------+------------------------+-----------
 ID     | integer                |
 title  | character varying(100) |
Indexes:
    "keyngram_idx" gin (to_tsvector('ngram'::regconfig, title::text))
TABLESPACE pg_default
postgres=# drop table if exists articles;
DROP TABLE
postgres=# CREATE TABLE articles (
postgres(# ID int,
postgres(# title VARCHAR(100),
postgres(# FULLTEXT KEY (title)WITH PARSER ngram
postgres(# );
CREATE TABLE
postgres=# \d articles
       Table "fulltext_test.articles"
 Column |          Type          | Modifiers
--------+------------------------+-----------
 ID     | integer                |
 title  | character varying(100) |
Indexes:
    "articles_to_tsvector_idx" gin (to_tsvector('ngram'::regconfig,
title::text)) TABLESPACE pg_default
postgres=# create table table_ddl_0154(col1 int,col2 varchar(64), FULLTEXT
idx_ddl_0154(col2));
CREATE TABLE
```

2.4.1.5.17 CREATE-TABLE-AS

功能描述

根据查询结果创建表。

CREATE TABLE AS 创建一个表并且用来自 SELECT 命令的结果填充该表。该表的字段和 SELECT 输出字段的名称及数据类型相关。不过用户可以通过明确地给出一个字段名称列表来覆盖 SELECT 输出字段的名称。

CREATE TABLE AS 对源表进行一次查询，然后将数据写入新表中，而查询视图结果会

根据源表的变化而有所改变。相比之下，每次做查询的时候，视图都重新计算定义它的 SELECT 语句。

注意事项

本章节只包含 dolphin 新增的语法，原 GBase 8c 的语法未做删除和修改。

语法格式

```
CREATE [ [ GLOBAL | LOCAL ] [ TEMPORARY | TEMP ] | UNLOGGED ] TABLE table_name
    [ (column_name [, ...] ) ]
    [ WITH ( {storage_parameter = value} [, ... ] ) ]
    [ ON CgbaseIT { PRESERVE ROWS | DELETE ROWS | DROP } ]
    [ COMPRESS | NOCOMPRESS ]
    [ TABLESPACE tablespace_name ]
    [ AS ] query
    [ WITH [ NO ] DATA ];
```

参数说明

[AS] query

一个 SELECT VALUES 命令。

AS 关键字可选，但若 query 中带有 WITH 语句，则必须使用括号将 query 包围，参考语句：

CREATE TABLE t_new (WITH temp_t(a, b) AS (SELECT a, b FROM t_old) SELECT * FROM temp_t);

示例

```
--创建一个表 tpcds.store_returns 表。
postgres=# CREATE TABLE tpcds.store_returns
(
    W_WAREHOUSE_SK            INTEGER                NOT NULL,
    W_WAREHOUSE_ID            CHAR(16)               NOT NULL,
    sr_item_sk                VARCHAR(20)                    ,
    W_WAREHOUSE_SQ_FT         INTEGER
);
--创建一个表 tpcds.store_returns_t1 并插入 tpcds.store_returns 表中 sr_item_sk 字段中大于 16 的数值。
postgres=# CREATE TABLE tpcds.store_returns_t1 AS SELECT * FROM tpcds.store_returns WHERE sr_item_sk > '4795';
--使用 tpcds.store_returns 拷贝一个新表 tpcds.store_returns_t2。
```

```
postgres=# CREATE TABLE tpcds.store_returns_t2 AS table tpcds.store_returns;
--删除表。
postgres=# DROP TABLE tpcds.store_returns_t1 ;
postgres=# DROP TABLE tpcds.store_returns_t2 ;
postgres=# DROP TABLE tpcds.store_returns;
```

相关链接

CREATE TABLE，SELECT

2.4.1.5.18 CREATE-TABLE-PARTITION

功能描述

创建分区表。分区表是把逻辑上的一张表根据某种方案分成几张物理块进行存储，这张逻辑上的表称之为分区表，物理块称之为分区。分区表是一张逻辑表，不存储数据，数据实际是存储在分区上的。

常见的分区方案有范围分区（Range Partitioning）、间隔分区（Interval Partitioning）、哈希分区（Hash Partitioning）、列表分区（List Partitioning）、数值分区（Value Partition）等。目前行存表支持范围分区、间隔分区、哈希分区、列表分区，列存表仅支持范围分区。

范围分区是根据表的一列或者多列，将要插入表的记录分为若干个范围，这些范围在不同的分区里没有重叠。为每个范围创建一个分区，用来存储相应的数据。

范围分区的分区策略是指记录插入分区的方式。目前范围分区仅支持范围分区策略。

范围分区策略：根据分区键值将记录映射到已创建的某个分区上，如果可以映射到已创建的某一分区上，则把记录插入到对应的分区上，否则给出报错和提示信息。这是最常用的分区策略。

间隔分区是一种特殊的范围分区，相比范围分区，新增间隔值定义，当插入记录找不到匹配的分区时，可以根据间隔值自动创建分区。

间隔分区只支持基于表的一列分区，并且该列只支持 TIMESTAMP[(p)] [WITHOUT TIME ZONE]、TIMESTAMP[(p)] [WITH TIME ZONE]、DATE 数据类型。

间隔分区策略：根据分区键值将记录映射到已创建的某个分区上，如果可以映射到已创建的某一分区上，则把记录插入到对应的分区上，否则根据分区键值和表定义信息自动创建一个分区，然后将记录插入新分区中，新创建的分区数据范围等于间隔值。

哈希分区是根据表的一列，为每个分区指定模数和余数，将要插入表的记录划分到对应的分区中，每个分区所持有的行都需要满足条件：分区键的值除以为其指定的模数将产生为

其指定的余数。

哈希分区策略：根据分区键值将记录映射到已创建的某个分区上，如果可以映射到已创建的某一分区上，则把记录插入到对应的分区上，否则返回报错和提示信息。

列表分区是根据表的一列，将要插入表的记录通过每一个分区中出现的键值划分到对应的分区中，这些键值在不同的分区里没有重叠。为每组键值创建一个分区，用来存储相应的数据。

列表分区策略：根据分区键值将记录映射到已创建的某个分区上，如果可以映射到已创建的某一分区上，则把记录插入到对应的分区上，否则给出报错和提示信息。

分区可以提供若干好处：

某些类型的查询性能可以得到极大提升。特别是表中访问率较高的行位于一个单独分区或少数几个分区上的情况下。分区可以减少数据的搜索空间，提高数据访问效率。

当查询或更新一个分区的大部分记录时，连续扫描那个分区而不是访问整个表可以获得巨大的性能提升。

如果需要大量加载或者删除的记录位于单独的分区上，则可以通过直接读取或删除那个分区以获得巨大的性能提升，同时还可以避免由于大量 DELETE 导致的 VACUUM 超载（仅范围分区）。

相比于内核语法，dolphin 的 rebuild,remove,check,repair,optimize,truncate,analyze,exchange,reorganize 都做了 B 兼容模式下的特色修改。

注意事项

唯一约束和主键约束的约束键包含所有分区键将为约束创建 LOCAL 索引，否则创建 GLOBAL 索引。

目前哈希分区和列表分区仅支持单列构建分区键，暂不支持多列构建分区键。

只需要有间隔分区表的 INSERT 权限，往该表 INSERT 数据时就可以自动创建分区。

对于分区表 PARTITION FOR (values)语法，values 只能是常量。

对于分区表 PARTITION FOR (values)语法，values 在需要数据类型转换时，建议使用强制类型转换，以防隐式类型转换结果与预期不符。

分区数最大值为 1048575 个，一般情况下业务不可能创建这么多分区，这样会导致内存不足。应参照参数 local_syscache_threshold 的值合理创建分区，分区表使用内存大致为（分

区数 * 3 / 1024）MB。理论上分区占用内存不允许大于 local_syscache_threshold 的值，同时还需要预留部分空间以供其他功能使用。

使用 table_indexclause 创建分区表上的索引为 LOCAL 索引，不支持选择 GLOBAL 索引。

支持使用表达式当作分区键，允许分区键使用算术运算符 “+"、”-"、"*"。

只支持部分函数允许在分区键中使用，支持的函数为: ABS()、CEILING()、DATEDIFF()、DAY()、DAYOFMONTH()、DAYOFWEEK()、DAYOFYEAR()、EXTRACT() 、FLOOR()、HOUR()、MICROSECOND()、MINUTE()、MOD()、MONTH()、QUARTER()、SECOND()、TIME_TO_SEC()、TO_DAYS()、TO_SECONDS()、UNIX_TIMESTAMP()、WEEKDAY()、YEAR()、YEARWEEK()。

表达式用作分区键时，只支持设置一个 partition key，且分区为 range、hash 和 list 分区，另外暂不支持列存表。

语法格式

```
CREATE TABLE [ IF NOT EXISTS ] partition_table_name
( [
    { column_name data_type [ COLLATE collation ] [ column_constraint [ ... ] ]
    | table_constraint
    | table_indexclause
    | LIKE source_table [ like_option [...] ] }[, ... ]
] )
    [create_option]
     PARTITION BY {
        {RANGE (partition_key) [ INTERVAL ('interval_expr') [ STORE IN (tablespace_name [, ... ] ) ] ] ( partition_less_than_item [, ... ] )} |
        {RANGE (partition_key) [ INTERVAL ('interval_expr') [ STORE IN (tablespace_name [, ... ] ) ] ] ( partition_start_end_item [, ... ] )} |
        {LIST | HASH (partition_key) (PARTITION partition_name [VALUES [IN] (list_values_clause)] opt_table_space )}
    } [ { ENABLE | DISABLE } ROW MOVEMENT ];
```

[create_option]

其中 create_option 为：

[WITH ({storage_parameter = value} [, ...])]

[COMPRESS | NOCOMPRESS]

[TABLESPACE tablespace_name]

[COMPRESSION [=] compression_arg]

[ENGINE [=] engine_name]

除了 WITH 选项外允许输入多次同一种 create_option，以最后一次的输入为准。

列约束 column_constraint：

[CONSTRAINT constraint_name]

{ NOT NULL |

NULL |

CHECK (expression) |

DEFAULT default_e xpr |

GENERATED ALWAYS AS (generation_expr) STORED |

UNIQUE index_parameters |

PRIMARY KEY index_parameters |

REFERENCES reftable [(refcolumn)] [MATCH FULL | MATCH PARTIAL | MATCH SIMPLE]

[ON DELETE action] [ON UPDATE action] }

[DEFERRABLE | NOT DEFERRABLE | INITIALLY DEFERRED | INITIALLY IMMEDIATE]

表约束 table_constraint：

[CONSTRAINT constraint_name]

{ CHECK (expression) |

UNIQUE (column_name [, ...]) index_parameters |

PRIMARY KEY (column_name [, ...]) index_parameters |

FOREIGN KEY (column_name [, ...]) REFERENCES reftable [(refcolumn [, ...])]

[MATCH FULL | MATCH PARTIAL | MATCH SIMPLE] [ON DELETE action] [ON UPDATE action] }

[DEFERRABLE | NOT DEFERRABLE | INITIALLY DEFERRED | INITIALLY IMMEDIATE]

创建表上索引| table_indexclause：

{INDEX | KEY} [index_name] [index_type] (key_part,...)[index_option]...

其中参数 index_type 为：

USING {BTREE | HASH | GIN | GIST | PSORT | UBTREE}

其中参数 key_part 为：

{col_name [(length)] | (expr)} [ASC | DESC]

其中 col_name (length)为前缀键，column_name 为前缀键的字段名，length 为前缀长度。前缀键将取指定字段数据的前缀作为索引键值，可以减少索引占用的存储空间。含有前缀键字段的过滤条件和连接条件可以使用索引。

说明：

前缀键支持的索引方法：btree、ubtree。

前缀键的字段的数据类型必须是二进制类型或字符类型（不包括特殊字符类型）。

前缀长度必须是不超过 2676 的正整数，并且不能超过字段的最大长度。对于二进制类型，前缀长度以字节数为单位。对于非二进制字符类型，前缀长度以字符数为单位。键值的实际长度受内部页面限制，若字段中含有多字节字符、或者一个索引上有多个键，索引行长度可能会超限，导致报错，设定较长的前缀长度时请考虑此情况。

其中参数 index_option 为：

index_option:{

COMMENT 'string'

| index_type

}

COMMENT、index_type 的顺序和数量任意，但相同字段仅最后一个值生效。

like 选项 like_option：

{ INCLUDING | EXCLUDING } { DEFAULTS | GENERATED | CONSTRAINTS | INDEXES | STORAGE | COMMENTS | RELOPTIONS| ALL }

索引存储参数 index_parameters：

[WITH ({storage_parameter = value} [, ...])]

[USING INDEX TABLESPACE tablespace_name]

partition_less_than_item：

PARTITION partition_name VALUES LESS THAN ({ partition_value | MAXVALUE }) | MAXVALUE [TABLESPACE tablespace_name]

partition_start_end_item：

PARTITION partition_name {

{START(partition_value) END (partition_value) EVERY (interval_value)} |

{START(partition_value) END ({partition_value | MAXVALUE}) | MAXVALUE} |

{START(partition_value)} |

{END ({partition_value | MAXVALUE}) | MAXVALUE}

} [TABLESPACE tablespace_name]

参数说明

IF NOT EXISTS

如果已经存在相同名称的表，不会抛出一个错误，而会发出一个通知，告知表关系已存在。

partition_table_name

分区表的名称。

取值范围：字符串，要符合标识符的命名规范。

column_name

新表中要创建的字段名。

取值范围：字符串，要符合标识符的命名规范。

data_type

字段的数据类型。

COLLATE collation

COLLATE 子句指定列的排序规则（该列必须是可排列的数据类型）。如果没有指定，则使用默认的排序规则。排序规则可以使用“select * from pg_collation;”命令从 pg_collation 系统表中查询，默认的排序规则为查询结果中以 default 开始的行。

CONSTRAINT constraint_name

列约束或表约束的名称。可选的约束子句用于声明约束，新行或者更新的行必须满足这些约束才能成功插入或更新。

定义约束有两种方法：

列约束：作为一个列定义的一部分，仅影响该列。

表约束：不和某个列绑在一起，可以作用于多个列。

LIKE source_table [like_option •••]

LIKE 子句声明一个表，新表自动从这个表里面继承所有字段名及其数据类型和非空约束。

和 INHERITS 不同，新表与原来的表之间在创建动作完毕之后是完全无关的。在源表做的任何修改都不会传播到新表中，并且也不可能在扫描源表的时候包含新表的数据。

字段缺省表达式只有在声明了 INCLUDING DEFAULTS 之后才会包含进来。缺省是不包含缺省表达式的，即新表中所有字段的缺省值都是 NULL。

如果指定了 INCLUDING GENERATED，则源表列的生成表达式会复制到新表中。默认不复制生成表达式。

非空约束将总是复制到新表中，CHECK 约束则仅在指定了 INCLUDING CONSTRAINTS 的时候才复制，而其他类型的约束则永远也不会被复制。此规则同时适用于表约束和列约束。

和 INHERITS 不同，被复制的列和约束并不使用相同的名称进行融合。如果明确的指定了相同的名称或者在另外一个 LIKE 子句中，将会报错。

如果指定了 INCLUDING INDEXES，则源表上的索引也将在新表上创建，默认不建立

索引。

如果指定了 INCLUDING STORAGE，则拷贝列的 STORAGE 设置也将被拷贝，默认情况下不包含 STORAGE 设置。

如果指定了 INCLUDING COMMENTS，则源表列、约束和索引的注释也会被拷贝过来。默认情况下，不拷贝源表的注释。

如果指定了 INCLUDING RELOPTIONS，则源表的存储参数（即源表的 WITH 子句）也将拷贝至新表。默认情况下，不拷贝源表的存储参数。

INCLUDING ALL 包含了 INCLUDING DEFAULTS、INCLUDING CONSTRAINTS、INCLUDING INDEXES、INCLUDING STORAGE、INCLUDING COMMENTS、INCLUDING PARTITION 和 INCLUDING RELOPTIONS 的内容。

WITH (storage_parameter [= value] [, •••])

这个子句为表或索引指定一个可选的存储参数。参数的详细描述如下所示：

FILLFACTOR

一个表的填充因子（fillfactor）是一个介于 10 和 100 之间的百分数。100（完全填充）是默认值。如果指定了较小的填充因子，INSERT 操作仅按照填充因子指定的百分率填充表页。每个页上的剩余空间将用于在该页上更新行，这就使得 UPDATE 有机会在同一页上放置同一条记录的新版本，这比把新版本放置在其他页上更有效。对于一个从不更新的表将填充因子设为 100 是最佳选择，但是对于频繁更新的表，选择较小的填充因子则更加合适。该参数对于列存表没有意义。

取值范围：10~100

ORIENTATION

决定了表的数据的存储方式。

取值范围：

COLUMN：表的数据将以列式存储。

ROW（缺省值）：表的数据将以行式存储。

 须知：orientation 不支持修改。

STORAGE_TYPE

指定存储引擎类型，该参数设置成功后就不再支持修改。

取值范围：

- USTORE，表示表支持 Inplace-Update 存储引擎。特别需要注意，使用 USTORE 表，必须要开启 track_counts 和 track_activities 参数，否则会引起空间膨胀。

- ASTORE，表示表支持 Append-Only 存储引擎。

- 默认值，不指定表时，默认是 Append-Only 存储。

COMPRESSION

列存表的有效值为 LOW/MIDDLE/HIGH/YES/NO，压缩级别依次升高，默认值为 LOW。

行存表不支持压缩。

MAX_BATCHROW

指定了在数据加载过程中一个存储单元可以容纳记录的最大数目。该参数只对列存表有效。

取值范围：10000~60000，默认 60000。

PARTIAL_CLUSTER_ROWS

指定了在数据加载过程中进行将局部聚簇存储的记录数目。该参数只对列存表有效。

取值范围：大于等于 MAX_BATCHROW，建议取值为 MAX_BATCHROW 的整数倍数。

DELTAROW_THRESHOLD

预留参数。该参数只对列存表有效。

取值范围：0￣9999

segment

使用段页式的方式存储。本参数仅支持行存表。不支持列存表、临时表、unlog 表。不支持 ustore 存储引擎。

取值范围：on/off

默认值：off

COMPRESS / NOCOMPRESS

创建一个新表时，需要在创建表语句中指定关键字 COMPRESS，这样，当对该表进行批量插入时就会触发压缩特性。该特性会在页范围内扫描所有元组数据，生成字典、压缩元组数据并进行存储。指定关键字 NOCOMPRESS 则不对表进行压缩。行存表不支持压缩。

该参数已废弃，列存表请使用 COMPRESSION 修改压缩等级。

缺省值为 NOCOMPRESS，即不对元组数据进行压缩。

TABLESPACE tablespace_name

指定新表将要在 tablespace_name 表空间内创建。如果没有声明，将使用默认表空间。

PARTITION BY RANGE(partition_key)

创建范围分区。partition_key 为分区键的名称。

（1）对于从句是 VALUES LESS THAN 的语法格式：

 须知：对于从句是 VALUE LESS THAN 的语法格式，范围分区策略的分区键最多支持 4 列。

该情形下，分区键支持的数据类型为：TINYINT[UNSIGNED]、SMALLINT[UNSIGNED]、INTEGER[UNSIGNED]、BIGINT[UNSIGNED]、DECIMAL、NUMERIC、REAL、DOUBLE PRECISION、CHARACTER VARYING(n)、VARCHAR(n)、CHARACTER(n)、CHAR(n)、CHARACTER、CHAR、TEXT、NVARCHAR、NVARCHAR2、NAME、TIMESTAMP[(p)] [WITHOUT TIME ZONE]、TIMESTAMP[(p)] [WITH TIME ZONE]、DATE。

（2）对于从句是 START END 的语法格式：

 须知：对于从句是 START END 的语法格式，范围分区策略的分区键仅支持 1 列。

该情形下，分区键支持的数据类型为：TINYINT[UNSIGNED]、SMALLINT[UNSIGNED]、INTEGER[UNSIGNED]、BIGINT[UNSIGNED]、DECIMAL、NUMERIC、REAL、DOUBLE PRECISION、TIMESTAMP[(p)] [WITHOUT TIME ZONE]、TIMESTAMP[(p)] [WITH TIME ZONE]、DATE。

（3）对于指定了 INTERVAL 子句的语法格式：

 须知：对于指定了 INTERVAL 子句的语法格式，范围分区策略的分区键仅支持 1 列。

该情形下，分区键支持的数据类型为：TIMESTAMP[(p)] [WITHOUT TIME ZONE]、TIMESTAMP[(p)] [WITH TIME ZONE]、DATE。

PARTITION partition_name VALUES LESS THAN ({ partition_value | MAXVALUE }) | MAXVALUE

指定各分区的信息。partition_name 为范围分区的名称。partition_value 为范围分区的上边界，取值依赖于 partition_key 的类型。MAXVALUE 表示分区的上边界，它通常用于设置最后一个范围分区的上边界。

 须知：

每个分区都需要指定一个上边界。

分区上边界的类型应当和分区键的类型一致。

分区列表是按照分区上边界升序排列的，值较小的分区位于值较大的分区之前。

PARTITION partition_name {START (partition_value) END (partition_value) EVERY (interval_value)}|{START (partition_value) END (partition_value|MAXVALUE) | MAXVALUE} | {START(partition_value)**} |{END (partition_value | MAXVALUE) | MAXVALUE}

指定各分区的信息，各参数意义如下：

partition_name：范围分区的名称或名称前缀，除以下情形外（假定其中的 partition_name 是 p1），均为分区的名称。

若该定义是 START+END+EVERY 从句，则语义上定义的分区的名称依次为 p1_1, p1_2, …。例如对于定义“PARTITION p1 START(1) END(4) EVERY(1)”，则生成的分区是：[1, 2), [2, 3) 和 [3, 4)，名称依次为 p1_1, p1_2 和 p1_3，即此处的 p1 是名称前缀。

若该定义是第一个分区定义，且该定义有 START 值，则范围（MINVALUE, START）将自动作为第一个实际分区，其名称为 p1_0，然后该定义语义描述的分区名称依次为 p1_1, p1_2, …。例如对于完整定义“PARTITION p1 START(1), PARTITION p2 START(2)”，则生成的分区是：(MINVALUE, 1), [1, 2) 和 [2, MAXVALUE)，其名称依次为 p1_0, p1_1 和 p2，即此处 p1 是名称前缀，p2 是分区名称。这里 MINVALUE 表示最小值。

partition_value：范围分区的端点值（起始或终点），取值依赖于 partition_key 的类型，不可是 MAXVALUE。

interval_value：对[START，END）表示的范围进行切分，interval_value 是指定切分后每个分区的宽度，不可是 MAXVALUE；如果（END-START）值不能整除以 EVERY 值，则仅最后一个分区的宽度小于 EVERY 值。

MAXVALUE：表示最大值，它通常用于设置最后一个范围分区的上边界。

 须知：

在创建分区表若第一个分区定义含 START 值，则范围（MINVALUE，START）将自动作为实际的第一个分区。

START END 语法需要遵循以下限制：

每个 partition_start_end_item 中的 START 值（如果有的话，下同）必须小于其 END 值。

相邻的两个 partition_start_end_item，第一个的 END 值必须等于第二个的 START 值；

每个 partition_start_end_item 中的 EVERY 值必须是正向递增的，且必须小于（END-START）值；

每个分区包含起始值，不包含终点值，即形如：[起始值，终点值)，起始值是 MINVALUE 时则不包含；

一个 partition_start_end_item 创建的每个分区所属的 TABLESPACE 一样；

partition_name 作为分区名称前缀时，其长度不要超过 57 字节，超过时自动截断；

在创建、修改分区表时请注意分区表的分区总数不可超过最大限制（1048575）；

在创建分区表时 START END 与 LESS THAN 语法不可混合使用。

即使创建分区表时使用 START END 语法，备份（gs_dump）出的 SQL 语句也是 VALUES LESS THAN 语法格式。

INTERVAL ('interval_expr') [STORE IN (tablespace_name [, …])]

间隔分区定义信息。

interval_expr：自动创建分区的间隔，例如：1 day、1 month。

STORE IN (tablespace_name [, …])：指定存放自动创建分区的表空间列表，如果有指定，则自动创建的分区从表空间列表中循环选择使用，否则使用分区表默认的表空间。

 须知：列存表不支持间隔分区。

PARTITION BY LIST(partition_key)

创建列表分区。partition_key 为分区键的名称。

对于 partition_key，列表分区策略的分区键仅支持 1 列。

对于从句是 VALUES (list_values_clause)的语法格式，list_values_clause 中包含了对应分区存在的键值，推荐每个分区的键值数量不超过 64 个。

分区键支持的数据类型为：INT1[UNSIGNED]、INT2[UNSIGNED]、INT4[UNSIGNED]、

INT8[UNSIGNED] 、 NUMERIC 、 VARCHAR(n) 、 CHAR 、 BPCHAR 、 NVARCHAR 、 NVARCHAR2、TIMESTAMP[(p)] [WITHOUT TIME ZONE]、TIMESTAMP[(p)] [WITH TIME ZONE]、DATE。分区个数不能超过 1048575 个。

PARTITION BY HASH(partition_key)

创建哈希分区。partition_key 为分区键的名称。

对于 partition_key，哈希分区策略的分区键仅支持 1 列。

分区键支持的数据类型为：INT1[UNSIGNED]、INT2[UNSIGNED]、INT4[UNSIGNED]、INT8[UNSIGNED]、NUMERIC、VARCHAR(n)、CHAR、BPCHAR、TEXT、NVARCHAR、NVARCHAR2、TIMESTAMP[(p)] [WITHOUT TIME ZONE]、TIMESTAMP[(p)] [WITH TIME ZONE]、DATE。分区个数不能超过 1048575 个。

{ ENABLE | DISABLE } ROW MOVEMENT

行迁移开关。

如果进行 UPDATE 操作时，更新了元组在分区键上的值，造成了该元组所在分区发生变化，就会根据该开关给出报错信息，或者进行元组在分区间的转移。

取值范围：

ENABLE（缺省值）：行迁移开关打开。

DISABLE：行迁移开关关闭。

 须知：列表/哈希分区表暂不支持 ROW MOVEMENT。

NOT NULL

字段值不允许为 NULL。ENABLE 用于语法兼容，可省略。

NULL

字段值允许 NULL ，这是缺省。

这个子句只是为和非标准 SQL 数据库兼容。不建议使用。

CHECK (condition) [NO INHERIT]

CHECK 约束声明一个布尔表达式，每次要插入的新行或者要更新的行的新值必须使表达式结果为真或未知才能成功，否则会抛出一个异常并且不会修改数据库。

声明为字段约束的检查约束应该只引用该字段的数值，而在表约束里出现的表达式可以

引用多个字段。

用 NO INHERIT 标记的约束将不会传递到子表中去。

ENABLE 用于语法兼容，可省略。

DEFAULT default_expr

DEFAULT 子句给字段指定缺省值。该数值可以是任何不含变量的表达式(不允许使用子查询和对本表中的其他字段的交叉引用)。缺省表达式的数据类型必须和字段类型匹配。

缺省表达式将被用于任何未声明该字段数值的插入操作。如果没有指定缺省值则缺省值为 NULL 。

GENERATED ALWAYS AS (generation_expr) STORED

该子句将字段创建为生成列，生成列的值在写入(插入或更新)数据时由 generation_expr 计算得到，STORED 表示像普通列一样存储生成列的值。

说明：

生成表达式不能以任何方式引用当前行以外的其他数据。生成表达式不能引用其他生成列，不能引用系统列。生成表达式不能返回结果集，不能使用子查询，不能使用聚集函数，不能使用窗口函数。生成表达式调用的函数只能是不可变（IMMUTABLE）函数。

不能为生成列指定默认值。

生成列不能作为分区键的一部分。

生成列不能和 ON UPDATE 约束字句的 CASCADE,SET NULL,SET DEFAULT 动作同时指定。生成列不能和 ON DELETE 约束字句的 SET NULL、SET DEFAULT 动作同时指定。

修改和删除生成列的方法和普通列相同。删除生成列依赖的普通列，生成列被自动删除。不能改变生成列所依赖的列的类型。

生成列不能被直接写入。在 INSERT 或 UPDATE 命令中，不能为生成列指定值，但是可以指定关键字 DEFAULT。

生成列的权限控制和普通列一样。

列存表、内存表 MOT 不支持生成列。外表中仅 postgres_fdw 支持生成列。

UNIQUE index_parameters

UNIQUE (column_name [, ···]) index_parameters

UNIQUE 约束表示表里的一个字段或多个字段的组合必须在全表范围内唯一。

对于唯一约束，NULL 被认为是互不相等的。

PRIMARY KEY index_parameters

PRIMARY KEY (column_name [, •••]) index_parameters

主键约束声明表中的一个或者多个字段只能包含唯一的非 NULL 值。

一个表只能声明一个主键。

DEFERRABLE | NOT DEFERRABLE

这两个关键字设置该约束是否可推迟。一个不可推迟的约束将在每条命令之后马上检查。可推迟约束可以推迟到事务结尾使用 SET CONSTRAINTS 命令检查。缺省是 NOT DEFERRABLE。目前，UNIQUE 约束、主键约束、外键约束可以接受这个子句。所有其他约束类型都是不可推迟的。

INITIALLY IMMEDIATE | INITIALLY DEFERRED

如果约束是可推迟的，则这个子句声明检查约束的缺省时间。

如果约束是 INITIALLY IMMEDIATE（缺省），则在每条语句执行之后就立即检查它；

如果约束是 INITIALLY DEFERRED ，则只有在事务结尾才检查它。

约束检查的时间可以用 SET CONSTRAINTS 命令修改。

USING INDEX TABLESPACE tablespace_name

为 UNIQUE 或 PRIMARY KEY 约束相关的索引声明一个表空间。如果没有提供这个子句，这个索引将在 default_tablespace 中创建，如果 default_tablespace 为空，将使用数据库的缺省表空间。

示例

示例 1：创建范围分区表 tpcds.web_returns_p1，含有 8 个分区，分区键为 integer 类型。分区的范围分别为：wr_returned_date_sk< 2450815、2450815<= wr_returned_date_sk< 2451179、2451179<=wr_returned_date_sk< 2451544 、 2451544 <= wr_returned_date_sk< 2451910 、2451910 <= wr_returned_date_sk< 2452275 、 2452275 <= wr_returned_date_sk< 2452640 、2452640 <= wr_returned_date_sk< 2453005、wr_returned_date_sk>=2453005。

```
--创建表 tpcds.web_returns。
postgres=# CREATE TABLE tpcds.web_returns
(
```

```
    W_WAREHOUSE_SK            INTEGER               NOT NULL,
    W_WAREHOUSE_ID            CHAR(16)              NOT NULL,
    W_WAREHOUSE_NAME          VARCHAR(20)                   ,
    W_WAREHOUSE_SQ_FT         INTEGER                       ,
    W_STREET_NUMBER           CHAR(10)                      ,
    W_STREET_NAME             VARCHAR(60)                   ,
    W_STREET_TYPE             CHAR(15)                      ,
    W_SUITE_NUMBER            CHAR(10)                      ,
    W_CITY                    VARCHAR(60)                   ,
    W_COUNTY                  VARCHAR(30)                   ,
    W_STATE                   CHAR(2)                       ,
    W_ZIP                     CHAR(10)                      ,
    W_COUNTRY                 VARCHAR(20)                   ,
    W_GMT_OFFSET              DECIMAL(5,2)
);
--创建分区表 tpcds.web_returns_p1。
postgres=# CREATE TABLE tpcds.web_returns_p1
(
    WR_RETURNED_DATE_SK       INTEGER                       ,
    WR_RETURNED_TIME_SK       INTEGER                       ,
    WR_ITEM_SK                INTEGER               NOT NULL,
    WR_REFUNDED_CUSTOMER_SK   INTEGER                       ,
    WR_REFUNDED_CDEMO_SK      INTEGER                       ,
    WR_REFUNDED_HDEMO_SK      INTEGER                       ,
    WR_REFUNDED_ADDR_SK       INTEGER                       ,
    WR_RETURNING_CUSTOMER_SK  INTEGER                       ,
    WR_RETURNING_CDEMO_SK     INTEGER                       ,
    WR_RETURNING_HDEMO_SK     INTEGER                       ,
    WR_RETURNING_ADDR_SK      INTEGER                       ,
    WR_WEB_PAGE_SK            INTEGER                       ,
    WR_REASON_SK              INTEGER                       ,
    WR_ORDER_NUMBER           BIGINT                NOT NULL,
    WR_RETURN_QUANTITY        INTEGER                       ,
    WR_RETURN_AMT             DECIMAL(7,2)                  ,
    WR_RETURN_TAX             DECIMAL(7,2)                  ,
    WR_RETURN_AMT_INC_TAX     DECIMAL(7,2)                  ,
    WR_FEE                    DECIMAL(7,2)                  ,
    WR_RETURN_SHIP_COST       DECIMAL(7,2)                  ,
    WR_REFUNDED_CASH          DECIMAL(7,2)                  ,
    WR_REVERSED_CHARGE        DECIMAL(7,2)                  ,
    WR_ACCOUNT_CREDIT         DECIMAL(7,2)                  ,
```

```
    WR_NET_LOSS               DECIMAL(7,2)
)
WITH (ORIENTATION = COLUMN,COMPRESSION=MIDDLE)
PARTITION BY RANGE(WR_RETURNED_DATE_SK)
(
        PARTITION P1 VALUES LESS THAN(2450815),
        PARTITION P2 VALUES LESS THAN(2451179),
        PARTITION P3 VALUES LESS THAN(2451544),
        PARTITION P4 VALUES LESS THAN(2451910),
        PARTITION P5 VALUES LESS THAN(2452275),
        PARTITION P6 VALUES LESS THAN(2452640),
        PARTITION P7 VALUES LESS THAN(2453005),
        PARTITION P8 VALUES LESS THAN(MAXVALUE)
);
--从示例数据表导入数据。
postgres=# INSERT INTO tpcds.web_returns_p1 SELECT * FROM tpcds.web_returns;
--删除分区 P8。
postgres=# ALTER TABLE tpcds.web_returns_p1 DROP PARTITION P8;
--增加分区 WR_RETURNED_DATE_SK 介于 2453005 和 2453105 之间。
postgres=# ALTER TABLE tpcds.web_returns_p1 ADD PARTITION P8 VALUES LESS THAN
(2453105);
--增加分区 WR_RETURNED_DATE_SK 介于 2453105 和 MAXVALUE 之间。
postgres=# ALTER TABLE tpcds.web_returns_p1 ADD PARTITION P9 VALUES LESS THAN
(MAXVALUE);
--删除分区 P8。
postgres=# ALTER TABLE tpcds.web_returns_p1 DROP PARTITION FOR (2453005);
--分区 P7 重命名为 P10。
postgres=# ALTER TABLE tpcds.web_returns_p1 RENAME PARTITION P7 TO P10;
--分区 P6 重命名为 P11。
postgres=# ALTER TABLE tpcds.web_returns_p1 RENAME PARTITION FOR (2452639) TO
P11;
--查询分区 P10 的行数。
postgres=# SELECT count(*) FROM tpcds.web_returns_p1 PARTITION (P10);
 count
--------
 0
(1 row)
--查询分区 P1 的行数。
postgres=# SELECT COUNT(*) FROM tpcds.web_returns_p1 PARTITION FOR (2450815);
 count
--------
```

```
 0
(1 row)
```

示例 2：创建范围分区表 tpcds.web_returns_p2，含有 8 个分区，分区键类型为 integer 类型，其中第 8 个分区上边界为 MAXVALUE。

八个分区的范围分别为： wr_returned_date_sk< 2450815 、 2450815<= wr_returned_date_sk< 2451179 、 2451179<=wr_returned_date_sk< 2451544 、 2451544 <= wr_returned_date_sk< 2451910 、 2451910 <= wr_returned_date_sk< 2452275 、 2452275 <= wr_returned_date_sk< 2452640 、 2452640 <= wr_returned_date_sk< 2453005 、 wr_returned_date_sk>=2453005。

分区表 tpcds.web_returns_p2 的表空间为 example1；分区 P1 到 P7 没有声明表空间，使用采用分区表 tpcds.web_returns_p2 的表空间 example1；指定分区 P8 的表空间为 example2。

假定数据库节点的数据目录/pg_location/mount1/path1，数据库节点的数据目录/pg_location/mount2/path2，数据库节点的数据目录/pg_location/mount3/path3，数据库节点的数据目录/pg_location/mount4/path4 是 dwsadmin 用户拥有读写权限的空目录。

```
postgres=# CREATE TABLESPACE example1 RELATIVE LOCATION
'tablespace1/tablespace_1';
postgres=# CREATE TABLESPACE example2 RELATIVE LOCATION
'tablespace2/tablespace_2';
postgres=# CREATE TABLESPACE example3 RELATIVE LOCATION
'tablespace3/tablespace_3';
postgres=# CREATE TABLESPACE example4 RELATIVE LOCATION
'tablespace4/tablespace_4';
postgres=# CREATE TABLE tpcds.web_returns_p2
(
    WR_RETURNED_DATE_SK       INTEGER                       ,
    WR_RETURNED_TIME_SK       INTEGER                       ,
    WR_ITEM_SK                INTEGER                NOT NULL,
    WR_REFUNDED_CUSTOMER_SK   INTEGER                       ,
    WR_REFUNDED_CDEMO_SK      INTEGER                       ,
    WR_REFUNDED_HDEMO_SK      INTEGER                       ,
    WR_REFUNDED_ADDR_SK       INTEGER                       ,
    WR_RETURNING_CUSTOMER_SK  INTEGER                       ,
    WR_RETURNING_CDEMO_SK     INTEGER                       ,
    WR_RETURNING_HDEMO_SK     INTEGER                       ,
    WR_RETURNING_ADDR_SK      INTEGER                       ,
    WR_WEB_PAGE_SK            INTEGER                       ,
    WR_REASON_SK              INTEGER                       ,
```

```
    WR_ORDER_NUMBER           BIGINT                NOT NULL,
    WR_RETURN_QUANTITY        INTEGER                       ,
    WR_RETURN_AMT             DECIMAL(7,2)                  ,
    WR_RETURN_TAX             DECIMAL(7,2)                  ,
    WR_RETURN_AMT_INC_TAX     DECIMAL(7,2)                  ,
    WR_FEE                    DECIMAL(7,2)                  ,
    WR_RETURN_SHIP_COST       DECIMAL(7,2)                  ,
    WR_REFUNDED_CASH          DECIMAL(7,2)                  ,
    WR_REVERSED_CHARGE        DECIMAL(7,2)                  ,
    WR_ACCOUNT_CREDIT         DECIMAL(7,2)                  ,
    WR_NET_LOSS               DECIMAL(7,2)
)
TABLESPACE example1
PARTITION BY RANGE(WR_RETURNED_DATE_SK)
(
        PARTITION P1 VALUES LESS THAN(2450815),
        PARTITION P2 VALUES LESS THAN(2451179),
        PARTITION P3 VALUES LESS THAN(2451544),
        PARTITION P4 VALUES LESS THAN(2451910),
        PARTITION P5 VALUES LESS THAN(2452275),
        PARTITION P6 VALUES LESS THAN(2452640),
        PARTITION P7 VALUES LESS THAN(2453005),
        PARTITION P8 VALUES LESS THAN(MAXVALUE) TABLESPACE example2
)
ENABLE ROW MOVEMENT;
--以 like 方式创建一个分区表。
postgres=# CREATE TABLE tpcds.web_returns_p3 (LIKE tpcds.web_returns_p2
INCLUDING PARTITION);
--修改分区 P1 的表空间为 example2。
postgres=# ALTER TABLE tpcds.web_returns_p2 MOVE PARTITION P1 TABLESPACE
example2;
--修改分区 P2 的表空间为 example3。
postgres=# ALTER TABLE tpcds.web_returns_p2 MOVE PARTITION P2 TABLESPACE
example3;
--以 2453010 为分割点切分 P8。
postgres=# ALTER TABLE tpcds.web_returns_p2 SPLIT PARTITION P8 AT (2453010) INTO
(
        PARTITION P9,
        PARTITION P10
);
--将 P6，P7 合并为一个分区。
```

```
postgres=# ALTER TABLE tpcds.web_returns_p2 MERGE PARTITIONS P6, P7 INTO
PARTITION P8;
--修改分区表迁移属性。
postgres=# ALTER TABLE tpcds.web_returns_p2 DISABLE ROW MOVEMENT;
--删除表和表空间。
postgres=# DROP TABLE tpcds.web_returns_p1;
postgres=# DROP TABLE tpcds.web_returns_p2;
postgres=# DROP TABLE tpcds.web_returns_p3;
postgres=# DROP TABLESPACE example1;
postgres=# DROP TABLESPACE example2;
postgres=# DROP TABLESPACE example3;
postgres=# DROP TABLESPACE example4;
```

示例 3：START END 语法创建、修改 Range 分区表。

假定/home/gbase/startend_tbs1、/home/gbase/startend_tbs2、/home/gbase/startend_tbs3、/home/gbase/startend_tbs4 是 gbase 用户拥有读写权限的空目录。

```
""-- 创建表空间
postgres=# CREATE TABLESPACE startend_tbs1 LOCATION
'/home/gbase/startend_tbs1';
postgres=# CREATE TABLESPACE startend_tbs2 LOCATION
'/home/gbase/startend_tbs2';
postgres=# CREATE TABLESPACE startend_tbs3 LOCATION
'/home/gbase/startend_tbs3';
postgres=# CREATE TABLESPACE startend_tbs4 LOCATION
'/home/gbase/startend_tbs4';
-- 创建临时 schema
postgres=# CREATE SCHEMA tpcds;
postgres=# SET CURRENT_SCHEMA TO tpcds;
-- 创建分区表，分区键是 integer 类型
postgres=# CREATE TABLE tpcds.startend_pt (c1 INT, c2 INT)
TABLESPACE startend_tbs1
PARTITION BY RANGE (c2) (
    PARTITION p1 START(1) END(1000) EVERY(200) TABLESPACE startend_tbs2,
    PARTITION p2 END(2000),
    PARTITION p3 START(2000) END(2500) TABLESPACE startend_tbs3,
    PARTITION p4 START(2500),
    PARTITION p5 START(3000) END(5000) EVERY(1000) TABLESPACE startend_tbs4
)
ENABLE ROW MOVEMENT;
-- 查看分区表信息
```

```
postgres=# SELECT relname, boundaries, spcname FROM pg_partition p JOIN
pg_tablespace t ON p.reltablespace=t.oid and
p.parentid='tpcds.startend_pt'::regclass ORDER BY 1;
   relname    | boundaries |    spcname
--------------+------------+---------------
 p1_0         | {1}        | startend_tbs2
 p1_1         | {201}      | startend_tbs2
 p1_2         | {401}      | startend_tbs2
 p1_3         | {601}      | startend_tbs2
 p1_4         | {801}      | startend_tbs2
 p1_5         | {1000}     | startend_tbs2
 p2           | {2000}     | startend_tbs1
 p3           | {2500}     | startend_tbs3
 p4           | {3000}     | startend_tbs1
 p5_1         | {4000}     | startend_tbs4
 p5_2         | {5000}     | startend_tbs4
 startend_pt  |            | startend_tbs1
(12 rows)
-- 导入数据，查看分区数据量
postgres=# INSERT INTO tpcds.startend_pt VALUES (GENERATE_SERIES(0, 4999),
GENERATE_SERIES(0, 4999));
postgres=# SELECT COUNT(*) FROM tpcds.startend_pt PARTITION FOR (0);
 count
-------
     1
(1 row)
postgres=# SELECT COUNT(*) FROM tpcds.startend_pt PARTITION (p3);
 count
-------
   500
(1 row)
-- 增加分区：[5000, 5300), [5300, 5600), [5600, 5900), [5900, 6000)
postgres=# ALTER TABLE tpcds.startend_pt ADD PARTITION p6 START(5000) END(6000)
EVERY(300) TABLESPACE startend_tbs4;
-- 增加 MAXVALUE 分区：p7
postgres=# ALTER TABLE tpcds.startend_pt ADD PARTITION p7 END(MAXVALUE);
-- 重命名分区 p7 为 p8
postgres=# ALTER TABLE tpcds.startend_pt RENAME PARTITION p7 TO p8;
-- 删除分区 p8
postgres=# ALTER TABLE tpcds.startend_pt DROP PARTITION p8;
-- 重命名 5950 所在的分区为：p71
```

```
postgres=# ALTER TABLE tpcds.startend_pt RENAME PARTITION FOR(5950) TO p71;
-- 分裂 4500 所在的分区[4000, 5000)
postgres=# ALTER TABLE tpcds.startend_pt SPLIT PARTITION FOR(4500)
INTO(PARTITION q1 START(4000) END(5000) EVERY(250) TABLESPACE startend_tbs3);
-- 修改分区 p2 的表空间为 startend_tbs4
postgres=# ALTER TABLE tpcds.startend_pt MOVE PARTITION p2 TABLESPACE
startend_tbs4;
-- 查看分区情形
postgres=# SELECT relname, boundaries, spcname FROM pg_partition p JOIN
pg_tablespace t ON p.reltablespace=t.oid and
p.parentid='tpcds.startend_pt'::regclass ORDER BY 1;
   relname    | boundaries |    spcname
--------------+------------+---------------
 p1_0         | {1}        | startend_tbs2
 p1_1         | {201}      | startend_tbs2
 p1_2         | {401}      | startend_tbs2
 p1_3         | {601}      | startend_tbs2
 p1_4         | {801}      | startend_tbs2
 p1_5         | {1000}     | startend_tbs2
 p2           | {2000}     | startend_tbs4
 p3           | {2500}     | startend_tbs3
 p4           | {3000}     | startend_tbs1
 p5_1         | {4000}     | startend_tbs4
 p6_1         | {5300}     | startend_tbs4
 p6_2         | {5600}     | startend_tbs4
 p6_3         | {5900}     | startend_tbs4
 p71          | {6000}     | startend_tbs4
 q1_1         | {4250}     | startend_tbs3
 q1_2         | {4500}     | startend_tbs3
 q1_3         | {4750}     | startend_tbs3
 q1_4         | {5000}     | startend_tbs3
 startend_pt  |            | startend_tbs1
(19 rows)
-- 删除表和表空间
postgres=# DROP SCHEMA tpcds CASCADE;
postgres=# DROP TABLESPACE startend_tbs1;
postgres=# DROP TABLESPACE startend_tbs2;
postgres=# DROP TABLESPACE startend_tbs3;
postgres=# DROP TABLESPACE startend_tbs4;
```

示例 4：创建间隔分区表 sales，初始包含 2 个分区，分区键为 DATE 类型。 分区的范围分别为：time_id < '2019-02-01 00:00:00'、

```
'2019-02-01 00:00:00' <= time_id < '2019-02-02 00:00:00' 。
--创建表 sales
postgres=# CREATE TABLE sales
(prod_id NUMBER(6),
 cust_id NUMBER,
 time_id DATE,
 channel_id CHAR(1),
 promo_id NUMBER(6),
 quantity_sold NUMBER(3),
 amount_sold NUMBER(10,2)
)
PARTITION BY RANGE (time_id)
INTERVAL('1 day')
( PARTITION p1 VALUES LESS THAN ('2019-02-01 00:00:00'),
  PARTITION p2 VALUES LESS THAN ('2019-02-02 00:00:00')
);
-- 数据插入分区 p1
postgres=# INSERT INTO sales VALUES(1, 12, '2019-01-10 00:00:00', 'a', 1, 1, 1);
-- 数据插入分区 p2
postgres=# INSERT INTO sales VALUES(1, 12, '2019-02-01 00:00:00', 'a', 1, 1, 1);
-- 查看分区信息
postgres=# SELECT t1.relname, partstrategy, boundaries FROM pg_partition t1,
pg_class t2 WHERE t1.parentid = t2.oid AND t2.relname = 'sales' AND t1.parttype
= 'p';
 relname | partstrategy |       boundaries
---------+--------------+------------------------
 p1      | r            | {"2019-02-01 00:00:00"}
 p2      | r            | {"2019-02-02 00:00:00"}
(2 rows)
-- 插入数据没有匹配的分区，新创建一个分区，并将数据插入该分区
-- 新分区的范围为 '2019-02-05 00:00:00' <= time_id < '2019-02-06 00:00:00'
postgres=# INSERT INTO sales VALUES(1, 12, '2019-02-05 00:00:00', 'a', 1, 1, 1);
-- 插入数据没有匹配的分区，新创建一个分区，并将数据插入该分区
-- 新分区的范围为 '2019-02-03 00:00:00' <= time_id < '2019-02-04 00:00:00'
postgres=# INSERT INTO sales VALUES(1, 12, '2019-02-03 00:00:00', 'a', 1, 1, 1);
-- 查看分区信息
postgres=# SELECT t1.relname, partstrategy, boundaries FROM pg_partition t1,
pg_class t2 WHERE t1.parentid = t2.oid AND t2.relname = 'sales' AND t1.parttype
= 'p';
 relname | partstrategy |       boundaries
---------+--------------+------------------------
```

```
 sys_p1  | i            | {"2019-02-06 00:00:00"}
 sys_p2  | i            | {"2019-02-04 00:00:00"}
 p1      | r            | {"2019-02-01 00:00:00"}
 p2      | r            | {"2019-02-02 00:00:00"}
(4 rows)
```

示例 5：创建 LIST 分区表 test_list，初始包含 4 个分区，分区键为 INT 类型。4 个分区的范围分别为：2000、3000、4000、5000。

```
--创建表 test_list
postgres=# create table test_list (col1 int, col2 int)
partition by list(col1)
(
partition p1 values (2000),
partition p2 values (3000),
partition p3 values (4000),
partition p4 values (5000)
);
-- 数据插入
postgres=# INSERT INTO test_list VALUES(2000, 2000);
INSERT 0 1
postgres=# INSERT INTO test_list VALUES(3000, 3000);
INSERT 0 1
-- 查看分区信息
postgres=# SELECT t1.relname, partstrategy, boundaries FROM pg_partition t1,
pg_class t2 WHERE t1.parentid = t2.oid AND t2.relname = 'test_list' AND
t1.parttype = 'p';
 relname | partstrategy | boundaries
---------+--------------+------------
 p1      | l            | {2000}
 p2      | l            | {3000}
 p3      | l            | {4000}
 p4      | l            | {5000}
(4 rows)
-- 插入数据没有匹配到分区，报错处理
postgres=# INSERT INTO test_list VALUES(6000, 6000);
ERROR:  inserted partition key does not map to any table partition
-- 添加分区
postgres=# alter table test_list add partition p5 values (6000);
ALTER TABLE
```

```
postgres=# SELECT t1.relname, partstrategy, boundaries FROM pg_partition t1,
pg_class t2 WHERE t1.parentid = t2.oid AND t2.relname = 'test_list' AND
t1.parttype = 'p';
 relname | partstrategy | boundaries
---------+--------------+------------
 p5      | l            | {6000}
 p4      | l            | {5000}
 p1      | l            | {2000}
 p2      | l            | {3000}
 p3      | l            | {4000}
(5 rows)
postgres=# INSERT INTO test_list VALUES(6000, 6000);
INSERT 0 1
-- 分区表和普通表交换数据
postgres=# create table t1 (col1 int, col2 int);
CREATE TABLE
postgres=# select * from test_list partition (p1);
 col1 | col2
------+------
 2000 | 2000
(1 row)
postgres=# alter table test_list exchange partition (p1) with table t1;
ALTER TABLE
postgres=# select * from test_list partition (p1);
 col1 | col2
------+------
(0 rows)
postgres=# select * from t1;
 col1 | col2
------+------
 2000 | 2000
(1 row)
-- truncate 分区
postgres=# select * from test_list partition (p2);
 col1 | col2
------+------
 3000 | 3000
(1 row)
postgres=# alter table test_list truncate partition p2;
ALTER TABLE
postgres=# select * from test_list partition (p2);
```

```
 col1 | col2
------+------
(0 rows)
-- 删除分区
postgres=# alter table test_list drop partition p5;
ALTER TABLE
postgres=# SELECT t1.relname, partstrategy, boundaries FROM pg_partition t1,
pg_class t2 WHERE t1.parentid = t2.oid AND t2.relname = 'test_list' AND
t1.parttype = 'p';
 relname | partstrategy | boundaries
---------+--------------+------------
 p4      | l            | {5000}
 p1      | l            | {2000}
 p2      | l            | {3000}
 p3      | l            | {4000}
(4 rows)
postgres=# INSERT INTO test_list VALUES(6000, 6000);
ERROR:  inserted partition key does not map to any table partition

-- 删除分区表
postgres=# drop table test_list;
```

示例 6：创建 HASH 分区表 test_hash，初始包含 2 个分区，分区键为 INT 类型。

```
""--创建表 test_hash
postgres=# create table test_hash (col1 int, col2 int)
partition by hash(col1)
(
partition p1,
partition p2
);
-- 数据插入
postgres=# INSERT INTO test_hash VALUES(1, 1);
INSERT 0 1
postgres=# INSERT INTO test_hash VALUES(2, 2);
INSERT 0 1
postgres=# INSERT INTO test_hash VALUES(3, 3);
INSERT 0 1
postgres=# INSERT INTO test_hash VALUES(4, 4);
INSERT 0 1
-- 查看分区信息
```

```
postgres=# SELECT t1.relname, partstrategy, boundaries FROM pg_partition t1,
pg_class t2 WHERE t1.parentid = t2.oid AND t2.relname = 'test_hash' AND
t1.parttype = 'p';
 relname | partstrategy | boundaries
---------+--------------+------------
 p1      | h            | {0}
 p2      | h            | {1}
(2 rows)
-- 查看数据
postgres=# select * from test_hash partition (p1);
 col1 | col2
------+------
    3 |    3
    4 |    4
(2 rows)
postgres=# select * from test_hash partition (p2);
 col1 | col2
------+------
    1 |    1
    2 |    2
(2 rows)
-- 分区表和普通表交换数据
postgres=# create table t1 (col1 int, col2 int);
CREATE TABLE
postgres=# alter table test_hash exchange partition (p1) with table t1;
ALTER TABLE
postgres=# select * from test_hash partition (p1);
 col1 | col2
------+------
(0 rows)
postgres=# select * from t1;
 col1 | col2
------+------
    3 |    3
    4 |    4
(2 rows)
-- truncate 分区
postgres=# alter table test_hash truncate partition p2;
ALTER TABLE
postgres=#  select * from test_hash partition (p2);
 col1 | col2
```

```
------+------
(0 rows)
-- 删除分区表
postgres=# drop table test_hash;

--rebuild,remove,check,repair,optimize 语法示例
--创建分区表 test_part
CREATE TABLE IF NOT EXISTS test_part
(
a int primary key not null default 5,
b int,
c int,
d int
)
PARTITION BY RANGE(a)
(
    PARTITION p0 VALUES LESS THAN (100000),
    PARTITION p1 VALUES LESS THAN (200000),
    PARTITION p2 VALUES LESS THAN (300000)
);
create unique index idx_c on test_part (c);
create index idx_b on test_part using btree(b) local;
alter table test_part add constraint uidx_d unique(d);
alter table test_part add constraint uidx_c unique using index idx_c;
--向分区表插入数据
insert into test_part (with RECURSIVE t_r(i,j,k,m) as(values(0,1,2,3) union all
select i+1,j+2,k+3,m+4 from t_r where i < 250000) select * from t_r);
--检查分区表系统信息
select relname from pg_partition where (parentid in (select oid from pg_class
where relname = 'test_part')) and parttype = 'p' and oid != relfilenode order by
relname;
--通过索引从分区表 select 数据
explain select * from test_part where ((99990 < c and c < 100000) or (219990 <
c and c < 220000));
select * from test_part where ((99990 < c and c < 100000) or (219990 < c and c
< 220000));
select * from test_part where ((99990 < d and d < 100000) or (219990 < d and d
< 220000));
select * from test_part where ((99990 < b and b < 100000) or (219990 < b and b
< 220000));
```

```
--测试 rebuild 分区表语法
ALTER TABLE test_part REBUILD PARTITION p0, p1;
--检查分区表系统信息和真实数据
select relname from pg_partition where (parentid in (select oid from pg_class
where relname = 'test_part')) and parttype = 'p' and oid != relfilenode order by
relname;
explain select * from test_part where ((99990 < c and c < 100000) or (219990 <
c and c < 220000));
select * from test_part where ((99990 < c and c < 100000) or (219990 < c and c
< 220000));
select * from test_part where ((99990 < d and d < 100000) or (219990 < d and d
< 220000));
select * from test_part where ((99990 < b and b < 100000) or (219990 < b and b
< 220000));

--测试 rebuild partition all 分区表语法
ALTER TABLE test_part REBUILD PARTITION all;
--检查分区表系统信息和真实数据
select relname from pg_partition where (parentid in (select oid from pg_class
where relname = 'test_part')) and parttype = 'p' and oid != relfilenode order by
relname;
explain select * from test_part where ((99990 < c and c < 100000) or (219990 <
c and c < 220000));
select * from test_part where ((99990 < c and c < 100000) or (219990 < c and c
< 220000));
select * from test_part where ((99990 < d and d < 100000) or (219990 < d and d
< 220000));
select * from test_part where ((99990 < b and b < 100000) or (219990 < b and b
< 220000));

--测试 repair check optimize 分区表语法
ALTER TABLE test_part repair PARTITION p0,p1;
ALTER TABLE test_part check PARTITION p0,p1;
ALTER TABLE test_part optimize PARTITION p0,p1;
ALTER TABLE test_part repair PARTITION all;
ALTER TABLE test_part check PARTITION all;
ALTER TABLE test_part optimize PARTITION all;

--测试 remove partitioning 语法
select relname, boundaries from pg_partition where parentid in (select parentid
from pg_partition where relname = 'test_part') order by relname;
```

```
select parttype,relname from pg_class where relname = 'test_part' and
relfilenode != oid;
ALTER TABLE test_part remove PARTITIONING;
--检查分区表移除分区信息后的系统信息和真实数据
explain select * from test_part where ((99990 < c and c < 100000) or (219990 <
c and c < 220000));
select * from test_part where ((99990 < c and c < 100000) or (219990 < c and c
< 220000));
select relname, boundaries from pg_partition where parentid in (select parentid
from pg_partition where relname = 'test_part') order by relname;
select parttype,relname from pg_class where relname = 'test_part' and
relfilenode != oid;
--truncate,analyze,exchange 语法示例
CREATE TABLE IF NOT EXISTS test_part1
(
a int,
b int
)
PARTITION BY RANGE(a)
(
    PARTITION p0 VALUES LESS THAN (100),
    PARTITION p1 VALUES LESS THAN (200),
    PARTITION p2 VALUES LESS THAN (300)
);
create table test_no_part1(a int, b int);
insert into test_part1 values(99,1),(199,1),(299,1);
select * from test_part1;
--truncate partition 语法
ALTER TABLE test_part1 truncate PARTITION p0, p1;
select * from test_part1;
insert into test_part1 (with RECURSIVE t_r(i,j) as(values(0,1) union all select
i+1,j+2 from t_r where i < 20) select * from t_r);
select * from test_part1;
ALTER TABLE test_part1 truncate PARTITION all;
select * from test_part1;
--测试 GBase 8c truncate partition 语法
insert into test_part1 values(99,1),(199,1);
select * from test_part1;
ALTER TABLE test_part1 truncate PARTITION p0, truncate PARTITION p1;
select * from test_part1;
--exchange partition 语法
```

```
insert into test_part1 values(99,1),(199,1),(299,1);
alter table test_part1 exchange partition p2 with table test_no_part1 without
validation;
select * from test_part1;
select * from test_no_part1;
alter table test_part1 exchange partition p2 with table test_no_part1 without
validation;
select * from test_part1;
select * from test_no_part1;
--测试 GBase 8c exchange partition 语法
alter table test_part1 exchange partition (p2) with table test_no_part1 without
validation;
select * from test_part1;
select * from test_no_part1;
alter table test_part1 exchange partition (p2) with table test_no_part1 without
validation;
select * from test_part1;
select * from test_no_part1;
--analyze partition 语法
alter table test_part1 analyze partition p0,p1;
alter table test_part1 analyze partition all;
--测试 GBase 8c analyze partition 语法
analyze test_part1 partition (p1);
--add, drop 语法示例
CREATE TABLE IF NOT EXISTS test_part2
(
a int,
b int
)
PARTITION BY RANGE(a)
(
    PARTITION p0 VALUES LESS THAN (100),
    PARTITION p1 VALUES LESS THAN (200),
    PARTITION p2 VALUES LESS THAN (300),
    PARTITION p3 VALUES LESS THAN (400)
);
CREATE TABLE IF NOT EXISTS test_subpart2
(
a int,
b int
)
```

```
PARTITION BY RANGE(a) SUBPARTITION BY RANGE(b)
(
    PARTITION p0 VALUES LESS THAN (100)
    (
        SUBPARTITION p0_0 VALUES LESS THAN (100),
        SUBPARTITION p0_1 VALUES LESS THAN (200),
        SUBPARTITION p0_2 VALUES LESS THAN (300)
    ),
    PARTITION p1 VALUES LESS THAN (200)
    (
        SUBPARTITION p1_0 VALUES LESS THAN (100),
        SUBPARTITION p1_1 VALUES LESS THAN (200),
        SUBPARTITION p1_2 VALUES LESS THAN (300)
    ),
    PARTITION p2 VALUES LESS THAN (300)
    (
        SUBPARTITION p2_0 VALUES LESS THAN (100),
        SUBPARTITION p2_1 VALUES LESS THAN (200),
        SUBPARTITION p2_2 VALUES LESS THAN (300)
    ),
    PARTITION p3 VALUES LESS THAN (400)
    (
        SUBPARTITION p3_0 VALUES LESS THAN (100),
        SUBPARTITION p3_1 VALUES LESS THAN (200),
        SUBPARTITION p3_2 VALUES LESS THAN (300)
    )
);
--test b_compatibility drop and add partition syntax
select relname, boundaries from pg_partition where parentid in (select parentid
from pg_partition where relname = 'test_part2');
ALTER TABLE test_part2 DROP PARTITION p3;
select relname, boundaries from pg_partition where parentid in (select parentid
from pg_partition where relname = 'test_part2');
ALTER TABLE test_part2 add PARTITION (PARTITION p3 VALUES LESS THAN
(400),PARTITION p4 VALUES LESS THAN (500),PARTITION p5 VALUES LESS THAN (600));
select relname, boundaries from pg_partition where parentid in (select parentid
from pg_partition where relname = 'test_part2');
ALTER TABLE test_part2 add PARTITION (PARTITION p6 VALUES LESS THAN
(700),PARTITION p7 VALUES LESS THAN (800));
ALTER TABLE test_part2 DROP PARTITION p4,p5,p6;
```

```
select relname, boundaries from pg_partition where parentid in (select parentid
from pg_partition where relname = 'test_part2');
ALTER TABLE test_part2 add PARTITION (PARTITION p4 VALUES LESS THAN (500));
select relname, boundaries from pg_partition where parentid in (select oid from
pg_partition where parentid in (select parentid from pg_partition where relname
= 'test_subpart2'));
ALTER TABLE test_subpart2 DROP SUBPARTITION p0_0;
ALTER TABLE test_subpart2 DROP SUBPARTITION p0_2, p1_0, p1_2;
select relname, boundaries from pg_partition where parentid in (select oid from
pg_partition where parentid in (select parentid from pg_partition where relname
= 'test_subpart2'));
--reorganize 分区语法示例
CREATE TABLE test_range_subpart
(
    a INT4 PRIMARY KEY,
    b INT4
)
PARTITION BY RANGE (a) SUBPARTITION BY HASH (b)
(
    PARTITION p1 VALUES LESS THAN (200)
    (
        SUBPARTITION s11,
        SUBPARTITION s12,
        SUBPARTITION s13,
        SUBPARTITION s14
    ),
    PARTITION p2 VALUES LESS THAN (500)
    (
        SUBPARTITION s21,
        SUBPARTITION s22
    ),
    PARTITION p3 VALUES LESS THAN (800),
    PARTITION p4 VALUES LESS THAN (1200)
    (
        SUBPARTITION s41
    )
);
insert into test_range_subpart values(199,1),(499,1),(799,1),(1199,1);
--test test_range_subpart
```

```
alter table test_range_subpart reorganize partition p1,p2 into (partition m1
values less than(100),partition m2 values less than(500)(subpartition
m21,subpartition m22));
select pg_get_tabledef('test_range_subpart');
select * from test_range_subpart subpartition(m22);
select * from test_range_subpart subpartition(m21);
select * from test_range_subpart partition(m1);
explain select /*+ indexscan(test_range_subpart test_range_subpart_pkey) */ *
from test_range_subpart where a > 0;
select * from test_range_subpart;
-- 分区表建索引，在 create table 中 index 默认为 local,不支持指定 global/local
CREATE TABLE test_partition_btree
(
    f1  INTEGER,
    f2  INTEGER,
    f3  INTEGER,
    key part_btree_idx using btree(f1)
)
PARTITION BY RANGE(f1)
(
        PARTITION P1 VALUES LESS THAN(2450815),
        PARTITION P2 VALUES LESS THAN(2451179),
        PARTITION P3 VALUES LESS THAN(2451544),
        PARTITION P4 VALUES LESS THAN(MAXVALUE)
);
-- 分区表建组合索引
CREATE TABLE test_partition_index
(
    f1  INTEGER,
    f2  INTEGER,
    f3  INTEGER,
    key part_btree_idx2 using btree(f1 desc, f2 asc)
)
PARTITION BY RANGE(f1)
(
        PARTITION P1 VALUES LESS THAN(2450815),
        PARTITION P2 VALUES LESS THAN(2451179),
        PARTITION P3 VALUES LESS THAN(2451544),
        PARTITION P4 VALUES LESS THAN(MAXVALUE)
);
-- 分区表列存创建索引
```

```
CREATE TABLE test_partition_column
(
    f1  INTEGER,
    f2  INTEGER,
    f3  INTEGER,
    key part_column(f1)
) with (ORIENTATION = COLUMN)
PARTITION BY RANGE(f1)
(
        PARTITION P1 VALUES LESS THAN(2450815),
        PARTITION P2 VALUES LESS THAN(2451179),
        PARTITION P3 VALUES LESS THAN(2451544),
        PARTITION P4 VALUES LESS THAN(MAXVALUE)
);
-- 分区表创建表达式索引
CREATE TABLE test_partition_expr
(
    f1  INTEGER,
    f2  INTEGER,
    f3  INTEGER,
    key part_expr_idx using btree((abs(f1)+1))
)
PARTITION BY RANGE(f1)
(
        PARTITION P1 VALUES LESS THAN(2450815),
        PARTITION P2 VALUES LESS THAN(2451179),
        PARTITION P3 VALUES LESS THAN(2451544),
        PARTITION P4 VALUES LESS THAN(MAXVALUE)
);
```

示例 7：创建分区键为表达式分区的分区表。

```
postgres=# create table testrangepart(a int, b int) partition by range(abs(a*2))
(
    partition p0 values less than(100),
    partition p1 values less than(200)
);
CREATE TABLE
postgres=# select partkeyexpr from pg_partition where (parttype = 'r') and
(parentid in (select oid from pg_class where relname = 'testrangepart'));

partkeyexpr
```

```
----------------------------------------------------------------------------
----------------------------------------------------------------------------
----------------------------------------------------------------------------
----------------------------------------------------------------------------
----------------------------------------------------------------------------
----------------------------------------------------------------------------
----------------------------------------------------------------------------
----------------------------------------------------------------------------
----------------------------------------------------------
{FUNCEXPR :funcid 1397 :funcresulttype 23 :funcresulttype_orig -1 :funcretset
false :funcformat 0 :funccollid 0 :inputcollid 0 :args ({OPEXPR :opno
514 :opfuncid 141 :opresulttype 23 :opretset false :opcollid 0 :inputcollid
0 :args ({VAR :varno 1 :varattno 1 :vartype 23 :vartypmod -1 :varcollid
0 :varlevelsup 0 :varnoold 1 :varoattno 1 :location 64} {CONST :consttype
23 :consttypmod -1 :constcollid 0 :constlen 4 :constbyval true :constisnull
false :ismaxvalue false :location 66 :constvalue 4 [ 2 0 0 0 0 0 0
0 ] :cursor_data  :row_count 0 :cur_dno -1 :is_open false :found false :not_found
false :null_open false :null_fetch false}) :location 65}) :location 60 :refSynOid
0}
(1 row)
postgres=# insert into testrangepart values(-51,1),(49,2);
INSERT 0 2
postgres=# insert into testrangepart values(-101,1);
ERROR:  inserted partition key does not map to any table partition
postgres=# select * from testrangepart partition(p0);
a  | b
----+---
49 | 2
(1 row)
postgres=# select * from testrangepart partition(p1);
a  | b
-----+---
-51 | 1
(1 row)
postgres=# select * from testrangepart where a = -51;
a  | b
-----+---
-51 | 1
(1 row)
```

相关链接

ALTER TABLE PARTITION，DROP TABLE

2.4.1.5.19 CREATE-TABLESPACE

功能描述

在数据库中创建一个新的表空间。

注意事项

本章节只包含 dolphin 新增的语法，原 GBase 8c 的语法未做删除和修改。

由于路径的特殊字符校验，在使用 ADD DATAFILE 创建表空间时，若输入路径以.ibd 结尾，dolphin 插件会将路径名称更改为以_ibd 结尾。

语法格式

```
CREATE TABLESPACE tablespace_name
    tablespace_details;
```

其中创建表空间的详细信息 tablespace_details 为：

```
[ OWNER user_name ] [RELATIVE] LOCATION 'directory' [ MAXSIZE 'space_size' ]
[with_option_clause] [ ENGINE [=] engine_name ]
| ADD DATAFILE 'directory' [ ENGINE [=] engine_name ]
```

参数说明

ENGINE [=] engine_name

指定存储引擎；该特性目前只有语法没有功能。

示例

```
--使用 ADD DATAFILE 语法创建表空间。
postgres=# CREATE TABLESPACE t_tbspace ADD DATAFILE 'my_tablespace' ENGINE =
test_engine;
CREATE TABLESPACE
--使用 ADD DATAFILE 语法创建表空间，输入路径以.ibd 结尾
postgres=# CREATE TABLESPACE test_tbspace_ibd ADD DATAFILE 'test_tbspace1.ibd';
WARNING:  Suffix ".ibd" of datafile path detected. The actual path will be renamed
as "test_tbspace1_ibd"
CREATE TABLESPACE
postgres=# CREATE TABLE t_tbspace(num int) TABLESPACE test_tbspace_ibd;
CREATE TABLE
postgres=# \d t_tbspace
   Table "public.t_tbspace"
```

```
 Column |  Type   | Modifiers
--------+---------+-----------
 num    | integer |
Tablespace: "test_tbspace_ibd"
```

2.4.1.5.20 CREATE-TRIGGER

功能描述

创建一个触发器。 触发器将与指定的表或视图关联，并在特定条件下执行指定的函数。

对比原始 GBase 8c 语法，新增了使用 MySQL 的格式创建触发器的语法。

新增了使用单条 sql 创建触发器的语法。

注意事项

当前仅支持在普通行存表上创建触发器，不支持在列存表、临时表、unlogged 表等类型表上创建触发器。

如果为同一事件定义了多个相同类型的触发器，则按触发器的名称字母顺序触发它们。

触发器常用于多表间数据关联同步场景，对 SQL 执行性能影响较大，不建议在大数据量同步及对性能要求高的场景中使用。

执行创建触发器操作的用户需要拥有指定表的 TRIGGER 权限或被授予了 CREATE ANY TRIGGER 权限。

语法格式

o 风格创建触发器的语法

```
CREATE [ CONSTRAINT ] TRIGGER trigger_name { BEFORE | AFTER | INSTEAD OF } { event
[ OR ... ] }
    ON table_name
    [ FROM referenced_table_name ]
    { NOT DEFERRABLE | [ DEFERRABLE ] { INITIALLY IMMEDIATE | INITIALLY
DEFERRED } }
    [ FOR [ EACH ] { ROW | STATEMENT } ]
    [ WHEN ( condition ) ]
    EXECUTE PROCEDURE function_name ( arguments );
```

兼容 mysql 兼容风格的创建触发器的语法

```
CREATE [ CONSTRAINT ] [ DEFINER=user ] TRIGGER [ IF NOT EXISTS ] trigger_name
{ BEFORE | AFTER | INSTEAD OF } { event [ OR ... ] }
    ON table_name
```

```
    [ FROM referenced_table_name ]
    { NOT DEFERRABLE | [ DEFERRABLE ] { INITIALLY IMMEDIATE | INITIALLY
DEFERRED } }
    [ FOR [ EACH ] { ROW | STATEMENT } ]
    [ WHEN ( condition ) ]
    [ trigger_order ]
    trigger_body
```

其中 event 包含以下几种：

INSERT

UPDATE [OF column_name [, ...]]

DELETE

TRUNCATE

其中 trigger_order 是：

{ FOLLOWS|PRECEDES } other_trigger_name

参数说明

CONSTRAINT

可选项，指定此参数将创建约束触发器，即触发器作为约束来使用。除了可以使用 SET CONSTRAINTS 调整触发器触发的时间之外，这与常规触发器相同。 约束触发器必须是 AFTER ROW 触发器。

DEFINER

可选项，指定此参数可以影响触发器内引用对象的权限控制。

IF NOT EXISTS

可选项，指定此参数如果触发器在相同的模式中具有相同的名称、相同的表、相同的表，则防止发生错误。

trigger_name

触发器名称，该名称不能限定模式，因为触发器自动继承其所在表的模式，且同一个表的触发器不能重名。 对于约束触发器，使用 SET CONSTRAINTS 修改触发器行为时也使用此名称。

取值范围：符合标识符命名规范的字符串，且最大长度不超过 63 个字符。

BEFORE

触发器函数是在触发事件发生前执行。

AFTER

触发器函数是在触发事件发生后执行，约束触发器只能指定为 AFTER。

INSTEAD OF

触发器函数直接替代触发事件。

event

启动触发器的事件，取值范围包括：INSERT、UPDATE、DELETE 或 TRUNCATE，也可以通过 OR 同时指定多个触发事件。

对于 UPDATE 事件类型，可以使用下面语法指定列：

UPDATE OF column_name1 [, column_name2 ...]

表示当这些列作为 UPDATE 语句的目标列时，才会启动触发器，但是 INSTEAD OF UPDATE 类型不支持指定列信息。

table_name

需要创建触发器的表名称。

取值范围：数据库中已经存在的表名称。

referenced_table_name

约束引用的另一个表的名称。 只能为约束触发器指定，常见于外键约束。

取值范围：数据库中已经存在的表名称。

DEFERRABLE | NOT DEFERRABLE

约束触发器的启动时机，仅作用于约束触发器。这两个关键字设置该约束是否可推迟。

详细介绍请参见 CREATE TABLE。

INITIALLY IMMEDIATE| INITIALLY DEFERRED

如果约束是可推迟的，则这个子句声明检查约束的缺省时间，仅作用于约束触发器。

详细介绍请参见 CREATE TABLE。

FOR EACH ROW | FOR EACH STATEMENT

触发器的触发频率。

FOR EACH ROW 是指该触发器是受触发事件影响的每一行触发一次。

FOR EACH STATEMENT 是指该触发器是每个 SQL 语句只触发一次。

未指定时默认值为 FOR EACH STATEMENT。约束触发器只能指定为 FOR EACH ROW。

condition

决定是否实际执行触发器函数的条件表达式。当指定 WHEN 时，只有在条件返回 true 时才会调用该函数。

在 FOR EACH ROW 触发器中，WHEN 条件可以通过分别写入 OLD.column_name 或 NEW.column_name 来引用旧行或新行值的列。 当然，INSERT 触发器不能引用 OLD 和 DELETE 触发器不能引用 NEW。

INSTEAD OF 触发器不支持 WHEN 条件。

WHEN 表达式不能包含子查询。

对于约束触发器，WHEN 条件的评估不会延迟，而是在执行更新操作后立即发生。 如果条件返回值不为 true，则触发器不会排队等待延迟执行。

function_name

用户定义的函数，必须声明为不带参数并返回类型为触发器，在触发器触发时执行。

arguments

执行触发器时要提供给函数的可选的以逗号分隔的参数列表。参数是文字字符串常量，简单的名称和数字常量也可以写在这里，但它们都将被转换为字符串。 请检查触发器函数的实现语言的描述，以了解如何在函数内访问这些参数。

trigger_order

可选项，trigger_order 特征中的{FOLLOWS|PRECEDES}控制触发器的优先触发顺序，B 兼容性模式下允许对同一个表，在同一触发事件定义多个触发器，会按照触发器创建的先后顺序来决定触发的优先顺序（先创建的优先）。可以通过{FOLLOWS|PRECEDES}来调整优先级。使用 FOLLOWS，最后一次使用的触发器与原始触发器最紧挨着，其他的触发器的优先级都顺序向后挤压；使用 PRECEDES，最后一次使用的触发器与原始触发器最紧挨着，其他的触发器的优先级都顺序向前挤压。

trigger_body

直接通过在 begin…end 之间书写代码块，定义触发器之后要完成的工作。

也可以是单条 sql 语句，目前支持的语句：insert、update、delete、set、call。

当设置了分隔符后，使用 MySQL 风格的创建触发器的语法，trigger_body 的格式是按照 MySQL 的格式规定书写的，declare 段落需要写在 begin … end 段落之间。

说明：

关于触发器种类：

INSTEAD OF 的触发器必须标记为 FOR EACH ROW，并且只能在视图上定义。

BEFORE 和 AFTER 触发器作用在视图上时，只能标记为 FOR EACH STATEMENT。

TRUNCATE 类型触发器仅限 FOR EACH STATEMENT。

表 1 表和视图上支持的触发器种类：

触发时机	触发事件	行级	语句级
BEFORE	INSERT/UPDATE/DELETE	表	表和视图
	TRUNCATE	不支持	表
AFTER	INSERT/UPDATE/DELETE	表	表和视图
	TRUNCATE	不支持	表
INSTEAD OF	INSERT/UPDATE/DELETE	视图	不支持
	TRUNCATE	不支持	不支持

表 2PLPGSQL 类型触发器函数特殊变量：

变量名	变量含义
NEW	INSERT 及 UPDATE 操作涉及 tuple 信息中的新值，对 DELETE 为空。
OLD	UPDATE 及 DELETE 操作涉及 tuple 信息中的旧值，对 INSERT 为空。
TG_NAME	触发器名称。
TG_WHEN	触发器触发时机（BEFORE/AFTER/INSTEAD OF）。
TG_LEVEL	触发频率（ROW/STATEMENT）。
TG_OP	触发操作（INSERT/UPDATE/DELETE/TRUNCATE）。
TG_RELID	触发器所在表 OID。
TG_RELNAME	触发器所在表名（已废弃，现用 TG_TABLE_NAME 替代）。
TG_TABLE_NAME	触发器所在表名。
TG_TABLE_SCHEMA	触发器所在表的 SCHEMA 信息。
TG_NARGS	触发器函数参数个数。

变量名	变量含义
TG_ARGV[]	触发器函数参数列表。

示例

```
--创建源表及触发表
postgres=# CREATE TABLE test_trigger_src_tbl(id1 INT, id2 INT, id3 INT);
postgres=# CREATE TABLE test_trigger_des_tbl(id1 INT, id2 INT, id3 INT);
--创建触发器函数
postgres=# CREATE OR REPLACE FUNCTION tri_insert_func() RETURNS TRIGGER AS
           $$
           DECLARE
           BEGIN
                   INSERT INTO test_trigger_des_tbl VALUES(NEW.id1, NEW.id2,
NEW.id3);
                   RETURN NEW;
           END
           $$ LANGUAGE PLPGSQL;
postgres=# CREATE OR REPLACE FUNCTION tri_update_func() RETURNS TRIGGER AS
           $$
           DECLARE
           BEGIN
                   UPDATE test_trigger_des_tbl SET id3 = NEW.id3 WHERE
id1=OLD.id1;
                   RETURN OLD;
           END
           $$ LANGUAGE PLPGSQL;
postgres=# CREATE OR REPLACE FUNCTION TRI_DELETE_FUNC() RETURNS TRIGGER AS
           $$
           DECLARE
           BEGIN
                   DELETE FROM test_trigger_des_tbl WHERE id1=OLD.id1;
                   RETURN OLD;
           END
           $$ LANGUAGE PLPGSQL;
--创建 INSERT 触发器
postgres=# CREATE TRIGGER insert_trigger
           BEFORE INSERT ON test_trigger_src_tbl
```

```
            FOR EACH ROW
            EXECUTE PROCEDURE tri_insert_func();
--创建 UPDATE 触发器
postgres=# CREATE TRIGGER update_trigger
            AFTER UPDATE ON test_trigger_src_tbl
            FOR EACH ROW
            EXECUTE PROCEDURE tri_update_func();
--创建 DELETE 触发器
postgres=# CREATE TRIGGER delete_trigger
            BEFORE DELETE ON test_trigger_src_tbl
            FOR EACH ROW
            EXECUTE PROCEDURE tri_delete_func();
--执行 INSERT 触发事件并检查触发结果
postgres=# INSERT INTO test_trigger_src_tbl VALUES(100,200,300);
postgres=# SELECT * FROM test_trigger_src_tbl;
postgres=# SELECT * FROM test_trigger_des_tbl;  //查看触发操作是否生效。
--执行 UPDATE 触发事件并检查触发结果
postgres=# UPDATE test_trigger_src_tbl SET id3=400 WHERE id1=100;
postgres=# SELECT * FROM test_trigger_src_tbl;
postgres=# SELECT * FROM test_trigger_des_tbl;  //查看触发操作是否生效
--执行 DELETE 触发事件并检查触发结果
postgres=# DELETE FROM test_trigger_src_tbl WHERE id1=100;
postgres=# SELECT * FROM test_trigger_src_tbl;
postgres=# SELECT * FROM test_trigger_des_tbl;  //查看触发操作是否生效
--修改触发器
postgres=# ALTER TRIGGER delete_trigger ON test_trigger_src_tbl RENAME TO
delete_trigger_renamed;
--禁用 insert_trigger 触发器
postgres=# ALTER TABLE test_trigger_src_tbl DISABLE TRIGGER insert_trigger;
--禁用当前表上所有触发器
postgres=# ALTER TABLE test_trigger_src_tbl DISABLE TRIGGER ALL;
--删除触发器
postgres=# DROP TRIGGER insert_trigger ON test_trigger_src_tbl;
postgres=# DROP TRIGGER update_trigger ON test_trigger_src_tbl;
postgres=# DROP TRIGGER delete_trigger_renamed ON test_trigger_src_tbl;
--创建 B 兼容性数据库
postgres=# create database db_mysql dbcompatibility 'B';
--创建触发器定义用户
postgres=# create user test_user password 'Gauss@123';
--创建原表及触发表
db_mysql=# create table test_mysql_trigger_src_tbl (id INT);
```

```
db_mysql=# create table test_mysql_trigger_des_tbl (id INT);
db_mysql=# create table animals (id INT, name CHAR(30));
db_mysql=# create table food (id INT, foodtype VARCHAR(32), remark VARCHAR(32),
time_flag TIMESTAMP);
--创建 MySQL 兼容 definer 语法触发器
db_mysql=# create definer=test_user trigger trigger1
                  after insert on test_mysql_trigger_src_tbl
                  for each row
                  begin
                   insert into test_mysql_trigger_des_tbl values(1);
                  end;
                  /
--创建 MySQL 兼容 trigger_order 语法触发器
db_mysql=# create trigger animal_trigger1
                  after insert on animals
                  for each row
                  begin
                   insert into food(id, foodtype, remark, time_flag) values
(1,'ice cream', 'sdsdsdsd', now());
                  end;
                  /
--创建 MySQL 兼容 FOLLOWS 触发器
db_mysql=# create trigger animal_trigger2
                  after insert on animals
                  for each row
                  follows animal_trigger1
                  begin
                   insert into food(id, foodtype, remark, time_flag) values
(2,'chocolate', 'sdsdsdsd', now());
                  end;
                  /
db_mysql=# create trigger animal_trigger3
          after insert on animals
          for each row
          follows animal_trigger1
          begin
              insert into food(id, foodtype, remark, time_flag) values (3,'cake',
'sdsdsdsd', now());
          end;
          /
db_mysql=# create trigger animal_trigger4
```

```
          after insert on animals
          for each row
          follows animal_trigger1
          begin
              insert into food(id, foodtype, remark, time_flag) values
(4,'sausage', 'sdsdsdsd', now());
          end;
          /
--执行 insert 触发事件并检查触发结果
db_mysql=# insert into animals (id, name) values(1,'lion');
db_mysql=# select * from animals;
db_mysql=# select id, foodtype, remark from food;
--创建 MySQL 兼容 PROCEDES 触发器
db_mysql=# create trigger animal_trigger5
          after insert on animals
          for each row
          precedes animal_trigger3
          begin
              insert into food(id, foodtype, remark, time_flag) values (5,'milk',
'sdsds', now());
          end;
          /
db_mysql=# create trigger animal_trigger6
          after insert on animals
          for each row
          precedes animal_trigger2
          begin
              insert into food(id, foodtype, remark, time_flag) values
(6,'strawberry', 'sdsds', now());
          end;
          /
--执行 insert 触发事件并检查触发结果
db_mysql=# insert into animals (id, name) values(2, 'dog');
db_mysql=# select * from animals;
db_mysql=# select id, foodtype, remark from food;
--创建 MySql
--创建 MySQL 兼容 if not exists 语法触发器
db_mysql=# create trigger if not exists animal_trigger1
          after insert on animals
          for each row
          begin
```

```
            insert into food(id, foodtype, remark, time_flag) values (1,'ice
cream', 'sdsdsdsd', now());
         end;
         /
--创建 MySQL 格式触发器
db_mysql=# delimiter //
db_mysql=# create trigger animal_d_trigger1
         after insert on animals
         for each row
         begin
             insert into food (id ,foodtype, remark, time_flag)
values(1,'ice','avcs', now());
         end;
         //
db_mysql=# delimiter ;
--创建 MySQL 兼容 trigger_body 为单条 sql 语法触发器
db_mysql=# create trigger animal_trigger_single
         after insert on animals
         for each row
         insert into food(id, foodtype, remark, time_flag) values (1,'ice cream',
'sdsdsdsd', now());
```

相关链接

ALTER TRIGGER，DROP TRIGGER，ALTER TABLE

2.4.1.5.21 CREATE-INDEX

功能描述

在指定的表上创建索引。

索引可以用来提高数据库查询性能，但是不恰当的使用将导致数据库性能下降。建议仅在匹配如下某条原则时创建索引：

经常执行查询的字段。

在连接条件上创建索引，对于存在多字段连接的查询，建议在这些字段上建立组合索引。例如，select * from t1 join t2 on t1.a=t2.a and t1.b=t2.b，可以在 t1 表上的 a、b 字段上建立组合索引。

where 子句的过滤条件字段上（尤其是范围条件）。

在经常出现在 order by、group by 和 distinct 后的字段。

在分区表上创建索引与在普通表上创建索引的语法不太一样，使用时请注意，如分区表上不支持并行创建索引，不支持创建部分索引。

新增可以指定 ALGORITHM 选项语法。

注意事项

本章节只包含 dolphin 新增的语法，原 GBase 8c 的语法未做删除和修改。 新增支持 option 的无序排列。

语法格式

在表上创建索引。

```
CREATE [ UNIQUE | FULLTEXT ] INDEX [ CONCURRENTLY ] [ [schema_name.]index_name ]
    { ON table_name [ USING method ] | [ USING method ] ON table_name }
    ({ { column_name | ( expression ) } [ COLLATE collation ] [ opclass ] [ ASC
| DESC ] [ NULLS { FIRST | LAST } ] }[, ...] )
    [ index_option ]
    [ WHERE predicate | ALGORITHM [=] {DEFAULT | INPLACE | COPY} ];
CREATE [UNIQUE] INDEX index_name
    ON tbl_name (key_part,...)
    [USING {BTREE | HASH}]
```

在分区表上创建索引。

```
CREATE [ UNIQUE ] INDEX [ [schema_name.]index_name ]
    { ON table_name [ USING method ] | [ USING method ] ON table_name }
    ( {{ column_name | ( expression ) } [ COLLATE collation ] [ opclass ] [ ASC
| DESC ] [ NULLS LAST ] }[, ...] )
    [ LOCAL [ ( { PARTITION index_partition_name [ ( SUBPARTITION
index_subpartition_name [, ...] ) ] [ TABLESPACE index_partition_tablespace ] }
[, ...] ) ] | GLOBAL ]
    [ index_option ] [ALGORITHM [=] {DEFAULT | INPLACE | COPY} ]
```

参数说明

FULLTEXT

该关键字为创建兼容 MySQL 的全文索引的语法。该全文索引主要用于字符串的搜索匹配。包含局部匹配搜索，支持中文，韩文，日文。与 MATCH () AGAINST ()配合使用。

column_name (length)

创建一个基于该表一个字段的前缀键索引，column_name 为前缀键的字段名，length 为前缀长度。

前缀键将取指定字段数据的前缀作为索引键值，可以减少索引占用的存储空间。含有前缀键字段的过滤条件和连接条件可以使用索引。

说明：

前缀键支持的索引方法：btree、ubtree。

前缀键的字段的数据类型必须是二进制类型或字符类型（不包括特殊字符类型）。

前缀长度必须是不超过 2676 的正整数，并且不能超过字段的最大长度。对于二进制类型，前缀长度以字节数为单位。对于非二进制字符类型，前缀长度以字符数为单位。键值的实际长度受内部页面限制，若字段中含有多字节字符、或者一个索引上有多个键，索引行长度可能会超限，导致报错，设定较长的前缀长度时请考虑此情况。

CREATE INDEX 语法中，不支持以下关键字作为前缀键的字段名称：COALESCE、CONVERT、DAYOFMONTH、DAYOFWEEK、DAYOFYEAR、DB_B_FORMAT、EXTRACT、GREATEST、HOUR_P、IFNULL、LEAST、LOCATE、MICROSECOND_P、MID、MINUTE_P、NULLIF、NVARCHAR、NVL、OVERLAY、POSITION、QUARTER、SECOND_P、SUBSTR、SUBSTRING、TEXT_P、TIME、TIMESTAMP、TIMESTAMPDIFF、TREAT、TRIM、WEEKDAY、WEEKOFYEAR、XMLCONCAT、XMLELEMENT、XMLEXISTS、XMLFOREST、XMLPARSE、XMLPI、XMLROOT、XMLSERIALIZE。若含有上述关键字的前缀键所在的索引是通过 ALTER TABLE 或 CREATE TABLE 语法创建的，导出的 CREATE INDEX 语句可能无法成功执行，请尽量不要使用上述关键字作为前缀键的列名称。

index_option

创建索引时可指定选项，其语法为：

INCLUDE (column_name [, ...])

| WITH ({ storage_parameter = value } [, ...])

| TABLESPACE tablespace_name

其中，TABLESPACE 选项允许输入多次，以最后一次的输入为准。

ALGORITHM

指定算法，可选项：DEFAULT、INPLACE、COPY。当前只做语法兼容，暂无实际功能。

示例

```
--创建表 tpcds.ship_mode_t1。
postgres=# create schema tpcds;
postgres=# CREATE TABLE tpcds.ship_mode_t1
(
    SM_SHIP_MODE_SK           INTEGER               NOT NULL,
    SM_SHIP_MODE_ID           CHAR(16)              NOT NULL,
    SM_TYPE                   CHAR(30)                      ,
    SM_CODE                   CHAR(10)                      ,
    SM_CARRIER                CHAR(20)                      ,
    SM_CONTRACT               CHAR(20)
)
;
--在表 tpcds.ship_mode_t1 上的 SM_SHIP_MODE_SK 字段上创建普通的唯一索引。
postgres=# CREATE UNIQUE INDEX ds_ship_mode_t1_index1 ON
tpcds.ship_mode_t1(SM_SHIP_MODE_SK);
--在表 tpcds.ship_mode_t1 上的 SM_SHIP_MODE_SK 字段上创建指定 B-tree 索引。
postgres=# CREATE INDEX ds_ship_mode_t1_index4 ON tpcds.ship_mode_t1 USING
btree(SM_SHIP_MODE_SK);
--在表 tpcds.ship_mode_t1 上 SM_CODE 字段上创建表达式索引。
postgres=# CREATE INDEX ds_ship_mode_t1_index2 ON
tpcds.ship_mode_t1(SUBSTR(SM_CODE,1 ,4));
--在表 tpcds.ship_mode_t1 上的 SM_SHIP_MODE_SK 字段上创建 SM_SHIP_MODE_SK 大于 10
的部分索引。
postgres=# CREATE UNIQUE INDEX ds_ship_mode_t1_index3 ON
tpcds.ship_mode_t1(SM_SHIP_MODE_SK) WHERE SM_SHIP_MODE_SK>10;
--重命名一个现有的索引。
postgres=# ALTER INDEX tpcds.ds_ship_mode_t1_index1 RENAME TO
ds_ship_mode_t1_index5;
--设置索引不可用。
postgres=# ALTER INDEX tpcds.ds_ship_mode_t1_index2 UNUSABLE;
--重建索引。
postgres=# ALTER INDEX tpcds.ds_ship_mode_t1_index2 REBUILD;
--删除一个现有的索引。
postgres=# DROP INDEX tpcds.ds_ship_mode_t1_index2;
--删除表。
postgres=# DROP TABLE tpcds.ship_mode_t1;
--创建表空间。
postgres=# CREATE TABLESPACE example1 RELATIVE LOCATION
'tablespace1/tablespace_1';
postgres=# CREATE TABLESPACE example2 RELATIVE LOCATION
'tablespace2/tablespace_2';
```

```
postgres=# CREATE TABLESPACE example3 RELATIVE LOCATION
'tablespace3/tablespace_3';
postgres=# CREATE TABLESPACE example4 RELATIVE LOCATION
'tablespace4/tablespace_4';
--创建表 tpcds.customer_address_p1。
postgres=# CREATE TABLE tpcds.customer_address_p1
(
    CA_ADDRESS_SK             INTEGER               NOT NULL,
    CA_ADDRESS_ID             CHAR(16)              NOT NULL,
    CA_STREET_NUMBER          CHAR(10)                      ,
    CA_STREET_NAME            VARCHAR(60)                   ,
    CA_STREET_TYPE            CHAR(15)                      ,
    CA_SUITE_NUMBER           CHAR(10)                      ,
    CA_CITY                   VARCHAR(60)                   ,
    CA_COUNTY                 VARCHAR(30)                   ,
    CA_STATE                  CHAR(2)                       ,
    CA_ZIP                    CHAR(10)                      ,
    CA_COUNTRY                VARCHAR(20)                   ,
    CA_GMT_OFFSET             DECIMAL(5,2)                  ,
    CA_LOCATION_TYPE          CHAR(20)
)
TABLESPACE example1
PARTITION BY RANGE(CA_ADDRESS_SK)
(
   PARTITION p1 VALUES LESS THAN (3000),
   PARTITION p2 VALUES LESS THAN (5000) TABLESPACE example1,
   PARTITION p3 VALUES LESS THAN (MAXVALUE) TABLESPACE example2
)
ENABLE ROW MOVEMENT;
--创建分区表索引 ds_customer_address_p1_index1，不指定索引分区的名称。
postgres=# CREATE INDEX ds_customer_address_p1_index1 ON
tpcds.customer_address_p1(CA_ADDRESS_SK) LOCAL;
--创建分区表索引 ds_customer_address_p1_index2，并指定索引分区的名称。
postgres=# CREATE INDEX ds_customer_address_p1_index2 ON
tpcds.customer_address_p1(CA_ADDRESS_SK) LOCAL
(
    PARTITION CA_ADDRESS_SK_index1,
    PARTITION CA_ADDRESS_SK_index2 TABLESPACE example3,
    PARTITION CA_ADDRESS_SK_index3 TABLESPACE example4
)
TABLESPACE example2;
```

```
--创建 GLOBAL 分区索引
Postgres=#CREATE INDEX ds_customer_address_p1_index3 ON
tpcds.customer_address_p1(CA_ADDRESS_ID) GLOBAL;
--不指定关键字，默认创建 GLOBAL 分区索引
postgres=#CREATE INDEX ds_customer_address_p1_index4 ON
tpcds.customer_address_p1(CA_ADDRESS_ID);
--修改分区表索引 CA_ADDRESS_SK_index2 的表空间为 example1。
postgres=# ALTER INDEX tpcds.ds_customer_address_p1_index2 MOVE PARTITION
CA_ADDRESS_SK_index2 TABLESPACE example1;
--修改分区表索引 CA_ADDRESS_SK_index3 的表空间为 example2。
postgres=# ALTER INDEX tpcds.ds_customer_address_p1_index2 MOVE PARTITION
CA_ADDRESS_SK_index3 TABLESPACE example2;
--重命名分区表索引。
postgres=# ALTER INDEX tpcds.ds_customer_address_p1_index2 RENAME PARTITION
CA_ADDRESS_SK_index1 TO CA_ADDRESS_SK_index4;
--删除索引和分区表。
postgres=# DROP INDEX tpcds.ds_customer_address_p1_index1;
postgres=# DROP INDEX tpcds.ds_customer_address_p1_index2;
postgres=# DROP TABLE tpcds.customer_address_p1;
--删除表空间。
postgres=# DROP TABLESPACE example1;
postgres=# DROP TABLESPACE example2;
postgres=# DROP TABLESPACE example3;
postgres=# DROP TABLESPACE example4;
--创建列存表以及列存表 GIN 索引。
postgres=# create table cgin_create_test(a int, b text) with (orientation =
column);
CREATE TABLE
postgres=# create index cgin_test on cgin_create_test using
gin(to_tsvector('ngram', b));
CREATE INDEX
##全文索引
postgres=# CREATE SCHEMA fulltext_test;
CREATE SCHEMA
postgres=# set current_schema to 'fulltext_test';
SET
postgres=# CREATE TABLE test (
id int unsigned auto_increment not null primary key,
title varchar,
boby text,
name name
```

```
);
NOTICE:  CREATE TABLE will create implicit sequence "test_id_seq" for serial
column "test.id"
NOTICE:  CREATE TABLE / PRIMARY KEY will create implicit index "test_pkey" for
table "test"
CREATE TABLE
postgres=# \d test
              Table "fulltext_test.test"
 Column |       Type        |        Modifiers
--------+-------------------+-----------------------
 id     | uint4             | not null AUTO_INCREMENT
 title  | character varying |
 boby   | text              |
 name   | name              |
Indexes:
    "test_pkey" PRIMARY KEY, btree (id) TABLESPACE pg_default
postgres=# CREATE FULLTEXT INDEX test_index_1 ON test (title, boby) WITH PARSER
ngram;
\d test_index_1
                  Index "fulltext_test.test_index_1"
    Column     | Type |                    Definition
---------------+------+-------------------------------------------------
 to_tsvector  | text | to_tsvector('"ngram"'::regconfig, title::text)
 to_tsvector1 | text | to_tsvector('"ngram"'::regconfig, boby)
gin, for table "fulltext_test.test"
postgres=# CREATE FULLTEXT INDEX test_index_2 ON test (title, boby, name);
CREATE INDEX
```

相关链接

CREATE INDEX

2.4.1.5.22 CREATE-VIEW

功能描述

创建一个视图。视图与基本表不同，是一个虚拟的表。数据库中仅存放视图的定义，而不存放视图对应的数据，这些数据仍存放在原来的基本表中。若基本表中的数据发生变化，从视图中查询出的数据也随之改变。从这个意义上讲，视图就像一个窗口，透过它可以看到数据库中用户感兴趣的数据及变化。

注意事项

被授予 CREATE ANY TABLE 权限的用户，可以在 public 模式和用户模式下创建视图。不可与同一模式下已存在的 synonym 产生命名冲突。

新增可以指定 ALGORITHM 选项语法。

语法格式

```
CREATE [ OR REPLACE ] [ALGORITHM = {UNDEFINED | MERGE | TEMPTABLE}] [DEFINER =
user] [ TEMP | TEMPORARY ] VIEW view_name [ ( column_name [, ...] ) ]
    [ WITH ( {view_option_name [= view_option_value]} [, ... ] ) ]
    AS query
    [ WITH [ CASCADED | LOCAL ] CHECK OPTION ];
```

说明：

创建视图时使用 WITH(security_barrier)可以创建一个相对安全的视图，避免攻击者利用低成本函数的 RAISE 语句打印出隐藏的基表数据。

当视图创建后，不允许使用 REPLACE 修改本视图当中的列名，也不允许删除列。

参数说明

OR REPLACE

当 CREATE VIEW 中存在 OR REPLACE 时，表示若以前存在该视图就进行替换，但新查询不能改变原查询的列定义，包括顺序、列名、数据类型、类型精度等，只可在列表末尾添加其他的列。

ALGORITHM

指定算法，可选项：UNDEFINED、MERGE、TEMPTABLE。当前只做语法兼容，暂无实际功能。

DEFINER = user

指定 user 作为视图的属主。该选项仅在 B 兼容模式下使用。

TEMP | TEMPORARY

创建临时视图。

view_name

要创建的视图名称。可以用模式修饰。

取值范围：字符串，符合标识符命名规范。

column_name

可选的名称列表，用作视图的字段名。如果没有给出，字段名取自查询中的字段名。

取值范围：字符串，符合标识符命名规范。

view_option_name [= view_option_value]

该子句为视图指定一个可选的参数。

security_barrier

当 VIEW 试图提供行级安全时，应使用该参数。

取值范围：Boolean 类型，TRUE、FALSE。

check_option

指定该视图的检查选项。

取值范围：LOCAL、CASCADED。

query

为视图提供行和列的 SELECT 或 VALUES 语句。

WITH [CASCADED | LOCAL] CHECK OPTION

该选项控制自动更新视图的行为，对视图的 insert 和 update，要检查确保新行满足视图定义的条件，即新行可以通过视图看到。如果没有通过检查，则拒绝修改。如果没有添加该选项，则允许通过对视图的 insert 和 update 来创建该视图不可见的行。支持下列检查选项：

LOCAL

只检查视图本身直接定义的条件，除非底层视图也定义了 CHECK OPTION，否则它们定义的条件都不检查。

CASCADED

检查该视图和所有底层视图定义的条件。如果仅声明了 CHECK OPTION，没有声明 LOCAL 和 CASCADED，默认是 CASCADED。

注意:

只有在可自动更新、没有 INSTEAD OF 触发器或者 INSTEAD 规则的视图上才支持 CHECK OPTION。如果一个自动更新的视图被定义在一个具有 INSTEAD OF 触发器的视图上，那么 CHECK OPTION 可以被用来检查该自动更新视图上的条件，但具有 INSTEAD OF

触发器的视图上的条件不会被检查。如果该视图或者任何底层关系具有导致 INSERT 或 UPDATE 命令被重写的 INSTEAD 规则，那么在被重写的查询中将忽略所有检查选项，包括任何来自定义在有 STEAD 规则关系上的可自动更新视图的检查。

基于 MySQL 外表的视图不支持 CHECK OPTION 选项。

可自动更新视图

简单视图是可自动更新的，系统允许在这类视图上执行 INSERT、UPDATE 和 DELETE 语句，如果一个视图满足以下条件，那么它就是可自动更新的。

视图的 FROM 列表中只有一项，并且必须是一个表或者是另一个可自动更新视图。

视图定义的顶层不能包含 WITH、DISTINCT、GROUP BY、HAVING、LIMIT、OFFSET 子句的视图

视图定义的顶层不能包含集合操作（UNION、INTERSET、EXCEPT）的视图。

视图的目标列表中不能包含聚集函数、窗口函数或者返回集合的函数。

一个可自动更新的视图可以混合可更新列以及不可更新列。如果一个列是对底层关系中一个可更新列的简单引用，则它是可更新的。否则该列是只读的，并且在一个 INSERT 或者 UPDATE 语句尝试对它赋值时会报错。

如果视图是可自动更新的，系统将把视图上的任何 INSERT、UPDATE 或者 DELETE 语句转换成在底层关系上的对应语句。

如果一个可自动更新的视图包含一个 WHERE 条件，该条件会限制底层关系的哪些行可以被该视图上的 UPDATE 以及 DELETE 语句修改。不过，一个允许被 UPDATE 修改的行可能让该行不再满足 WHERE 条件，并且因此也不再能从视图中可见。类似的，一个 INSERT 命令可能插入不满足 WHERE 条件的行，因此从该视图中看不到这些行。CHECK OPTION 可以用来阻止 INSERT 和 UPDATE 命令创建这类从视图中无法看到的行。

一个更加复杂的、不满足上述条件的视图默认是只读的，系统不允许在该视图上执行 INSERT、UPDATE 和 DELETE 语句。可以通过在该视图上创建一个 INSTEAD OF 触发器来获得可更新视图的效果，该触发器必须把该视图上尝试的插入转换成其他表上合法的动作，更多信息请见 CREATE TRIGGER。另一种方式是创建规则（见 CREATE RULE）。

注意在视图上执行插入、更新或删除的用户必须具有该视图上相应的插入、更新或删除特权。此外，视图的所有者必须拥有底层关系上对应的权限，但执行的用户并不需要底层关系上的任何权限。

示例

```
--创建字段 spcname 为 pg_default 组成的视图。
postgres=# CREATE VIEW myView AS
    SELECT * FROM pg_tablespace WHERE spcname = 'pg_default';
--查看视图。
postgres=# SELECT * FROM myView ;
--删除视图 myView。
postgres=# DROP VIEW myView;
```

相关链接

ALTER VIEW，DROP VIEW

2.4.1.5.23 DESCRIBE-TABLE

功能描述

DESCRIBE 和 EXPLAIN 互为同义词，可以用于查看指定表结构，或查看指定 SQL 的执行计划。

查看执行计划部分内容详见 EXPLAIN。

注意事项

临时表需要指定临时表对应的 schema 查询。

复合主键索引所有参与列都会在 Key 字段中显示为 PRI。

复合唯一索引所有参与列都会在 Key 字段中显示为 UNI。

如果一个列参与了多个索引的创建，将按 PRI、UNI、MUL 的优先级顺序显示。

生成列会在 Default 中显示生成式。

不支持表同义词。

语法格式

```
{DESCRIBE | DESC | EXPLAIN} tbl_name
```

参数说明

{DESCRIBE | DESC | EXPLAIN}

使用 DESCRIBE、DESC 和 EXPLAIN 效果是等价的。

tbl_name

表名，可指定表名。也可以指定 schema_name.table_name。

示例

```
--创建 test 表
postgres=# CREATE TABLE test2
postgres-# (
postgres(# id int PRIMARY KEY
postgres(# );
NOTICE:  CREATE TABLE / PRIMARY KEY will create implicit index "test2_pkey" for
table "test2"
CREATE TABLE
postgres=# create table test
postgres-# (
postgres(# a SERIAL,
postgres(# b varchar(10),
postgres(# c varchar(10),
postgres(# d varchar(10),
postgres(# e varchar(10),
postgres(# f varchar(10),
postgres(# g varchar(10) DEFAULT 'g',
postgres(# h varchar(10) NOT NULL,
postgres(# i int DEFAULT 0,
postgres(# j int DEFAULT 0,
postgres(# k int GENERATED ALWAYS AS (i + j) STORED,
postgres(# l int DEFAULT 0,
postgres(# m int CHECK (m < 50),
postgres(# PRIMARY KEY (a, b),
postgres(# FOREIGN KEY(l) REFERENCES test2(id)
postgres(# );
NOTICE:  CREATE TABLE will create implicit sequence "test_a_seq" for serial
column "test.a"
NOTICE:  CREATE TABLE / PRIMARY KEY will create implicit index "test_pkey" for
table "test"
CREATE TABLE
postgres=# CREATE UNIQUE INDEX idx_c on test (c);
CREATE INDEX
postgres=# CREATE UNIQUE INDEX idx_d_e on test (d, e);
CREATE INDEX
postgres=# CREATE INDEX idx_f on test (f);
CREATE INDEX
--查看 test 表结构
postgres=# desc test;
```

```
 Field |         Type          | Null | Key |              Default              |
Extra
-------+-----------------------+------+-----+-----------------------------------
-+-------
 a     | integer               | NO   | PRI | nextval('test_a_seq'::regclass) |
 b     | character varying(10) | NO   | PRI | NULL                              |
 c     | character varying(10) | YES  | UNI | NULL                              |
 d     | character varying(10) | YES  | UNI | NULL                              |
 e     | character varying(10) | YES  | UNI | NULL                              |
 f     | character varying(10) | YES  | MUL | NULL                              |
 g     | character varying(10) | YES  |     | 'g'::character varying            |
 h     | character varying(10) | NO   |     | NULL                              |
 i     | integer               | YES  |     | 0
|
 j     | integer               | YES  |     | 0
|
 k     | integer               | YES  |     | (i + j)                           |
 l     | integer               | YES  | MUL | 0                                 |
 m     | integer               | YES  |     | NULL
|
(13 rows)

postgres=# desc public.test;
 Field |         Type          | Null | Key |              Default              |
Extra
-------+-----------------------+------+-----+-----------------------------------
-+-------
 a     | integer               | NO   | PRI | nextval('test_a_seq'::regclass) |
 b     | character varying(10) | NO   | PRI | NULL                              |
 c     | character varying(10) | YES  | UNI | NULL                              |
 d     | character varying(10) | YES  | UNI | NULL                              |
 e     | character varying(10) | YES  | UNI | NULL                              |
 f     | character varying(10) | YES  | MUL | NULL                              |
 g     | character varying(10) | YES  |     | 'g'::character varying            |
 h     | character varying(10) | NO   |     | NULL                              |
 i     | integer               | YES  |     | 0
|
 j     | integer               | YES  |     | 0
|
 k     | integer               | YES  |     | (i + j)                           |
 l     | integer               | YES  | MUL | 0                                 |
```

```
 m      | integer                | YES  |      | NULL
|
(13 rows)
postgres=# describe public.test;
 Field |         Type           | Null | Key |             Default             |
Extra
-------+------------------------+------+-----+---------------------------------
-+-------
 a     | integer                | NO   | PRI | nextval('test_a_seq'::regclass) |
 b     | character varying(10)  | NO   | PRI | NULL                            |
 c     | character varying(10)  | YES  | UNI | NULL                            |
 d     | character varying(10)  | YES  | UNI | NULL                            |
 e     | character varying(10)  | YES  | UNI | NULL                            |
 f     | character varying(10)  | YES  | MUL | NULL                            |
 g     | character varying(10)  | YES  |     | 'g'::character varying          |
 h     | character varying(10)  | NO   |     | NULL                            |
 i     | integer                | YES  |     | 0
|
 j     | integer                | YES  |     | 0
|
 k     | integer                | YES  |     | (i + j)                         |
 l     | integer                | YES  | MUL | 0                               |
 m      | integer                | YES  |      | NULL
|
(13 rows)
```

相关链接

EXPLAIN

2.4.1.5.24 DO

功能描述

执行匿名代码块。

代码块被看做是没有参数的一段函数体，返回值类型是 void。它的解析和执行是同一时刻发生的。

或，执行表达式并不返回结果

注意事项

相比于原始的 GBase 8c，dolphin 对于 DO 语法的修改为：

在原始语法的基础上增加 DO expr_list 语法，用于执行表达式并不返回结果。

语法格式

```
DO [ LANGUAGE lang_name ] code;
```

或

```
DO expr[,expr...];
```

参数说明

lang_name

用来解析代码的程序语言的名称，如果缺省，默认的语言是 plpgsql。

code

程序语言代码可以被执行的。程序语言必须指定为字符串才行。

expr

表达式，多个表达式使用逗号进行分隔，表达式支持的内容参考表达式。

示例

```
--执行并不返回结果
postgres=# DO 1;
postgres=# DO pg_sleep(1);
--执行多个表达式，不返回结果
postgres=# DO 1+2;
```

2.4.1.5.25 DROP-DATABASE

功能描述

删除一个数据库或一个模式。

注意事项

相比于原始的 GBase 8c，dolphin 对于 DROP DATABASE 语法的修改为：

增加 DATABASE 解析为 SCHEMA 含义。

语法格式

```
DROP DATABASE [ IF EXISTS ] database_name ;
```

参数说明

IF EXISTS

如果指定的数据库不存在，则发出一个 notice 而不是抛出一个错误。

database_name

要删除的数据库名称。

取值范围：字符串，已存在的数据库名称。

说明：

B 兼容性下， dolphin.b_compatibility_mode 为 on 时，语法等同为无 dolphin 时的 DROP SCHEMA 语法； dolphin.b_compatibility_mode 为 off 时， 语法为无 dolphin 时的 DROP DATABASE 语法。

示例

请参见 CREATE DATABASE 的示例。

相关链接

CREATE DATABASE,DROP DATABASE

2.4.1.5.26 DROP-INDEX

功能描述

删除索引。

注意事项

只有索引的所有者或者拥有索引所在表的 INDEX 权限的用户有权限执行 DROP INDEX 命令，系统管理员默认拥有此权限。

对于全局临时表，当某个会话已经初始化了全局临时表对象（包括创建全局临时表和第一次向全局临时表内插入数据）时，其他会话不能够执行该表上索引的删除操作。

新增可以指定 ALGORITHM 选项语法。

语法格式

```
DROP INDEX [ CONCURRENTLY ] [ IF EXISTS ]
    index_name [, ...] [ CASCADE | RESTRICT ] [ALGORITHM [=] {DEFAULT | INPLACE
| COPY}];
```

参数说明

CONCURRENTLY

以不加锁的方式删除索引。删除索引时，一般会阻塞其他语句对该索引所依赖表的访问。加此关键字，可实现删除过程中不做阻塞。

此选项只能指定一个索引的名称，并且 CASCADE 选项不支持。

普通 DROP INDEX 命令可以在事务内执行，但是 DROP INDEX CONCURRENTLY 不可以在事务内执行。

IF EXISTS

如果指定的索引不存在，则发出一个 notice 而不是抛出一个错误。

index_name

要删除的索引名。

取值范围：已存在的索引。

CASCADE | RESTRICT

CASCADE：表示允许级联删除依赖于该索引的对象。

RESTRICT（缺省值）：表示有依赖与此索引的对象存在，则该索引无法被删除。

ALGORITHM

指定算法，可选项：DEFAULT、INPLACE、COPY。当前只做语法兼容，暂无实际功能。

示例

请参见 CREATE INDEX 的示例。

相关链接

ALTER INDEX，CREATE INDEX

2.4.1.5.27 DROP-TABLESPACE

功能描述

删除一个表空间。

注意事项

相比于原始的 GBase 8c，dolphin 对于 DROP TABLESPACE 语法的修改主要为：

新增 ENGINE [=] engine_name 可选项，无实际意义，仅作语法兼容。

语法格式

```
DROP TABLESPACE [ IF EXISTS ] tablespace_name [ENGINE [=] engine_name];
```

参数说明

IF EXISTS

如果指定的表空间不存在，则发出一个 notice 而不是抛出一个错误。

tablespace_name

表空间的名称。

取值范围：已存在的表空间的名称。

engine_name

无实际意义。

取值范围：任意字符串。

示例

请参见 CREATE TABLESPACE 的示例。

相关链接

ALTER TABLESPACE，CREATE TABLESPACE

2.4.1.5.28 EXECUTE

功能描述

执行一个前面准备好的预备语句。因为一个预备语句只在会话的生命期里存在，那么预备语句必须是在当前会话的前些时候用 PREPARE 语句创建的。

注意事项

如果创建预备语句的 PREPARE 语句声明了一些参数，那么传递给 EXECUTE 语句的必须是一个兼容的参数集，否则就会生成一个错误。

相比于原始的 GBase 8c，dolphin 对于 PREPARE 语法的修改为：支持 EXECUTE USING 语法。

语法格式

```
EXECUTE name [ ( parameter [, ...] ) ];
EXECUTE name USING parameter [, ...];
```

参数说明

name

要执行的预备语句的名称。

parameter

给预备语句的一个参数的具体数值。它必须是一个和生成与创建这个预备语句时指定参数的数据类型相兼容的值的表达式。

示例

```
--创建表 reason。
postgres=# CREATE TABLE tpcds.reason (
    CD_DEMO_SK            INTEGER           NOT NULL,
    CD_GENDER             character(16)               ,
    CD_MARITAL_STATUS     character(100)
)
;
--插入数据。
postgres=# INSERT INTO tpcds.reason VALUES(51, 'AAAAAAAADDAAAAAA', 'reason 51');
--创建表 reason_t1。
postgres=# CREATE TABLE tpcds.reason_t1 AS TABLE tpcds.reason;
--为一个 INSERT 语句创建一个预备语句然后执行它。
postgres=# PREPARE insert_reason(integer,character(16),character(100)) AS
INSERT INTO tpcds.reason_t1 VALUES($1,$2,$3);
postgres=# EXECUTE insert_reason(52, 'AAAAAAAADDAAAAAA', 'reason 52');
postgres=# EXECUTE insert_reason USING 52, 'AAAAAAAADDAAAAAA', 'reason 52';
--删除表 reason 和 reason_t1。
postgres=# DROP TABLE tpcds.reason;
postgres=# DROP TABLE tpcds.reason_t1;
```

2.4.1.5.29 EXPLAIN

功能描述

EXPLAIN 和 DESCRIBE 互为同义词，可以用于查看指定表结构，或查看指定 SQL 的执行计划。

查看表格结构的语法说明请参考 DESCRIBE 语法，以下内容仅介绍查看执行计划部分。

执行计划将显示 SQL 语句所引用的表会采用什么样的扫描方式，如：简单的顺序扫描、索引扫描等。如果引用了多个表，执行计划还会显示用到的 JOIN 算法。

执行计划的最关键的部分是语句的预计执行开销，这是计划生成器估算执行该语句将花费多长的时间。

若指定了 ANALYZE 选项，则该语句会被执行，然后根据实际的运行结果显示统计数据，包括每个计划节点内时间总开销（毫秒为单位）和实际返回的总行数。这对于判断计划生成器的估计是否接近现实非常有用。

注意事项

在指定 ANALYZE 选项时，语句会被执行。如果用户想使用 EXPLAIN 分析 INSERT、UPDATE、DELETE、CREATE TABLE AS 或 EXECUTE 语句，而不想改动数据（执行这些语句会影响数据），请使用如下方法。

```
START TRANSACTION;
EXPLAIN ANALYZE ...;
ROLLBACK;
```

语法格式

显示 SQL 语句的执行计划，支持多种选项，对选项顺序无要求。

```
{EXPLAIN | DESCRIBE | DESC} [ (  option  [, ...] )  ] statement;
或
{EXPLAIN | DESCRIBE | DESC} [FORMAT = format_name] statement;
或
{EXPLAIN | DESCRIBE | DESC} [EXTENDED] statement;
```

其中 {EXPLAIN | DESCRIBE | DESC} 表示使用 DESCRIBE、DESC 和 EXPLAIN 效果是等价的。

选项 option 子句的语法为：

```
ANALYZE [ boolean ] |
    ANALYSE [ boolean ] |
    VERBOSE [ boolean ] |
    COSTS [ boolean ] |
    CPU [ boolean ] |
    DETAIL [ boolean ] |(不可用)
    NODES [ boolean ] |(不可用)
    NUM_NODES [ boolean ] |(不可用)
    BUFFERS [ boolean ] |
    TIMING [ boolean ] |
    PLAN [ boolean ] |
    FORMAT { TEXT | XML | JSON | YAML }
```

显示 SQL 语句的执行计划，且要按顺序给出选项。

```
{EXPLAIN | DESCRIBE | DESC} { [  { ANALYZE  | ANALYSE  }  ] [ VERBOSE  ]  |
PERFORMANCE  } statement;
```

参数说明

statement

指定要分析的 SQL 语句。

ANALYZE boolean | ANALYSE boolean

显示实际运行时间和其他统计数据。

取值范围：

TRUE（缺省值）：显示实际运行时间和其他统计数据。

FALSE：不显示。

VERBOSE boolean

显示有关计划的额外信息。

取值范围：

TRUE（缺省值）：显示额外信息。

FALSE：不显示。

COSTS boolean

包括每个规划节点的估计总成本，以及估计的行数和每行的宽度。

取值范围：

TRUE（缺省值）：显示估计总成本和宽度。

FALSE：不显示。

CPU boolean

打印 CPU 的使用情况的信息。

取值范围：

TRUE（缺省值）：显示 CPU 的使用情况。

FALSE：不显示。

DETAIL boolean（不可用）

打印数据库节点上的信息。

取值范围：

TRUE（缺省值）：打印数据库节点的信息。

FALSE：不打印。

NODES boolean（不可用）

打印 query 执行的节点信息。

取值范围：

TRUE（缺省值）：打印执行的节点的信息。

FALSE：不打印。

NUM_NODES boolean（不可用）

打印执行中的节点的个数信息。

取值范围：

TRUE（缺省值）：打印数据库节点个数的信息。

FALSE：不打印。

BUFFERS boolean

包括缓冲区的使用情况的信息。

取值范围：

TRUE：显示缓冲区的使用情况。

FALSE（缺省值）：不显示。

TIMING boolean

包括实际的启动时间和花费在输出节点上的时间信息。

取值范围：

TRUE（缺省值）：显示启动时间和花费在输出节点上的时间信息。

FALSE：不显示。

PLAN

是否将执行计划存储在 plan_table 中。当该选项开启时，会将执行计划存储在 PLAN_TABLE 中，不打印到当前屏幕，因此该选项为 on 时，不能与其他选项同时使用。

取值范围：

ON（缺省值）：将执行计划存储在 plan_table 中，不打印到当前屏幕。执行成功返回 EXPLAIN SUCCESS。

OFF：不存储执行计划，将执行计划打印到当前屏幕。

FORMAT

指定输出格式。

取值范围：TEXT、XML、JSON 和 YAML。

默认值：TEXT。

PERFORMANCE

使用此选项时，即打印执行中的所有相关信息。

format_name

指定输出格式。

取值范围：JSON 或 TRADITIONAL。

默认值：TRADITIONAL

EXTENDED

可选，无区别。

示例

```
-- 1、首先创建一个兼容性为 B 模式的数据库，并切换
postgres=# create database postgres with dbcompatibility 'B';
CREATE DATABASE
postgres=# \c postgres
Non-SSL connection (SSL connection is recgbaseended when requiring high-security)
You are now connected to database "postgres" as user "gbase".
-- 2、在新的数据库上创建一个表
postgres=# create table test_t(c1 int, c2 varchar(30));
CREATE TABLE
-- 3、查看 SQL 的执行计划
postgres=# explain select * from test_t;
```

```
                        QUERY PLAN
-----------------------------------------------------------
 Seq Scan on test_t  (cost=0.00..17.29 rows=729 width=82)
(1 row)
-- 4、在查看计划时可以指定输出格式
-- 注意：只有当 explain_perf_mode 为 normal 时，才支持 json 格式
postgres=# SET explain_perf_mode=normal;
SET
postgres=# explain (format json) select * from test_t;
            QUERY PLAN
---------------------------------
 [                              +
   {                            +
     "Plan": {                  +
       "Node Type": "Seq Scan",  +
       "Relation Name": "test_t",+
       "Alias": "test_t",        +
       "Startup Cost": 0.00,     +
       "Total Cost": 17.29,      +
       "Plan Rows": 729,         +
       "Plan Width": 82          +
     }                          +
   }                            +
 ]
(1 row)
postgres=# explain format=json select * from test_t;
            QUERY PLAN
---------------------------------
 [                              +
   {                            +
     "Plan": {                  +
       "Node Type": "Seq Scan",  +
       "Relation Name": "test_t",+
       "Alias": "test_t",        +
       "Startup Cost": 0.00,     +
       "Total Cost": 17.29,      +
       "Plan Rows": 729,         +
       "Plan Width": 82          +
     }                          +
   }                            +
 ]
```

```
(1 row)
-- 5、如果一个查询中的 where 子句的列有索引，在条件不一样或数据量等不一样时，可能会显示不同的执行计划
postgres=# create index idx_test_t_c1 on test_t(c1);
CREATE INDEX
postgres=# insert into test_t values(generate_series(1, 200), 'hello GBase 8c');
INSERT 0 200
postgres=# explain select c1, c2 from test_t where c1=100;
                                QUERY PLAN
---------------------------------------------------------------------------
 Bitmap Heap Scan on test_t  (cost=4.28..12.74 rows=4 width=82)
   Recheck Cond: (c1 = 100)
   ->  Bitmap Index Scan on idx_test_t_c1  (cost=0.00..4.28 rows=4 width=0)
         Index Cond: (c1 = 100)
(4 rows)
-- 6、可以通过 costs 选项，指定是否显示开销
postgres=# explain (costs false) select * from test_t where c1=100;
               QUERY PLAN
----------------------------------------
 Bitmap Heap Scan on test_t
   Recheck Cond: (c1 = 100)
   ->  Bitmap Index Scan on idx_test_t_c1
         Index Cond: (c1 = 100)
(4 rows)
-- 7、在兼容性为 B 的数据库下，explain 和 desc (describe) 是等价的，还可以用来查看表结构信息
postgres=# explain test_t;
 Field |         Type          | Null | Key | Default | Extra
-------+-----------------------+------+-----+---------+-------
 c1    | integer               | YES  | MUL | NULL    |
 c2    | character varying(30) | YES  |     | NULL    |
(2 rows)
```

相关链接

ANALYZE|ANALYSE，DESCRIBE

2.4.1.5.30 FLUSH-BINARY-LOGS

功能描述

手动归档 pg_xlog 日志。

注意事项

N/A

语法格式

FLUSH BINARY LOGS

参数说明

N/A

示例

```
postgres=# flush binary logs;
 pg_switch_xlog
----------------
 0/FE8DD60
(1 row)
```

2.4.1.5.31 GRANT

功能描述

GRANT 用于授予一个或多个角色的权限。

注意事项

本章节只包含 dolphin 新增的语法，原 GBase 8c 的语法未做删除和修改。 增加 ALTER ROUTINE、CRAETE ROUTINE、CREATE TEMPORARY TABLES、CREATE USER、CREATE TABLESPACE、INDEX 权限

语法格式

新增 ALTER ROUTINE 权限

与 function 和 procedure 的 alter 权限基本一致

修改后的语法说明为：

```
GRANT { { EXECUTE | ALTER ROUTINE | ALTER | DROP | COMMENT } [, ...] | ALL
[ PRIVILEGES ] }
ON {FUNCTION {function_name ( [ {[ argmode ] [ arg_name ] arg_type} [, ...] ] )}
| PROCEDURE {proc_name ( [ {[ argmode ] [ arg_name ] arg_type} [, ...] ] )} [, ...]
| ALL FUNCTIONS IN SCHEMA schema_name [, ...] | ALL PROCEDURE IN SCHEMA schema_name
[, ...] | schema_name.*}
TO { [ GROUP ] role_name | PUBLIC } [, ...]
```

```
[ WITH GRANT OPTION ];
```

新增 CREATE ROUTINE 权限

与 CREATE ANY FUNCTION 权限基本一致

修改后的语法说明为：

```
GRANT { CREATE ANY TABLE | ALTER ANY TABLE | DROP ANY TABLE | SELECT ANY TABLE
| INSERT ANY TABLE | UPDATE ANY TABLE |
  DELETE ANY TABLE | CREATE ANY SEQUENCE | CREATE ANY INDEX | CREATE ANY FUNCTION
| CREATE ROUTINE | EXECUTE ANY FUNCTION |
  CREATE ANY PACKAGE | EXECUTE ANY PACKAGE | CREATE ANY TYPE } [, ...]
  [ON *.*]
  TO [ GROUP ] role_name [, ...]
  [ WITH ADMIN OPTION ];
```

新增 CREATE TEMPORARY TABLES 权限

与 TEMPORARY 权限基本一致

修改后的语法说明为：

```
GRANT { { CREATE | CONNECT | CREATE TEMPORARY TABLES | TEMPORARY | TEMP | ALTER
| DROP | COMMENT } [, ...]
    | ALL [ PRIVILEGES ] }
    ON { DATABASE database_name [, ...] | database_name.* }
    TO { [ GROUP ] role_name | PUBLIC } [, ...]
    [ WITH GRANT OPTION ];
```

新增 CREATE USER 权限

控制用户创建新用户的权限，与用户的 CREATEROLE 和 NOCREATEROLE 权限基本一致

新增的语法说明为：

```
GRANT CREATE USER ON *.* TO ROLE_NAME;
```

新增 CREATE TABLESPACE 权限

控制用户创建新表空间的权限

新增的语法说明为：

```
GRANT CREATE TABLESPACE ON *.* TO ROLE_NAME;
```

新增 INDEX 权限

与 CREATE ANY INDEX 权限基本一致

修改后的语法说明为：

```
GRANT INDEX
  ON *.*
  TO [ GROUP ] role_name [, ...]
  [ WITH ADMIN OPTION ];
```

参数说明

N/A

示例

```
GRANT ALTER ROUTINE ON FUNCTION TEST TO USER_TESTER;
GRANT CREATE ANY FUNCTION TO USER_TESTER;
GRANT CREATE TEMPORARY TABLES ON DATABASE DATABASE_TEST TO USER_TESTER;
GRANT CREATE USER ON *.* TO USER_TESTER;
GRANT CREATE TABLESPACE ON *.* TO USER_TESTER;
GRANT INDEX TO TEST_USER;
```

相关链接

GRANT

2.4.1.5.32 GRANT-REVOKE-PROXY

功能描述

授予、召回代理者权限。

注意事项

N/A

语法格式

```
GRANT PROXY ON user
    TO user [, user] ...
    [WITH GRANT OPTION]
REVOKE PROXY ON user
    FROM user [, user] ...
```

参数说明

{PROXY}

语法关键词。

user

用户（角色）名。

示例

```
--创建简单表
postgres=# CREATE SCHEMA tst_schema1;
postgres=# SET SEARCH_PATH TO tst_schema1;
postgres=# CREATE TABLE tst_t1
(
id int,
name varchar(20)
);
INSERT INTO tst_t1 values(20220101, 'proxy_example');
--创建用户
postgres=# DROP ROLE if EXISTS test_proxy_u1;
postgres=# CREATE ROLE test_proxy_u1 IDENTIFIED BY 'test_proxy_u1@123';
postgres=# DROP ROLE if EXISTS test_proxy_u2;
postgres=# CREATE ROLE test_proxy_u3 IDENTIFIED BY 'test_proxy_u2@123';
postgres=# DROP ROLE if EXISTS test_proxy_u3;
postgres=# CREATE ROLE test_proxy_u3 IDENTIFIED BY 'test_proxy_u3@123';
--schema、表权限授予
postgres=# GRANT ALL ON SCHEMA tst_schema1 TO test_proxy_u2;
postgres=# GRANT ALL ON SCHEMA tst_schema1 TO test_proxy_u2;
postgres=# GRANT ALL ON SCHEMA tst_schema1 TO test_proxy_u2;
postgres=# GRANT ALL ON tst_t1 to test_proxy_u1;
--权限检测（无权限）
postgres=# SET ROLE test_proxy_u2 PASSWORD 'test_proxy_u2@123';
postgres=#> SELECT * FROM tst_schema1.tst_t1;
ERROR:  permission denied for relation tst_t1
DETAIL:  N/A
--权限检测（拥有代理者权限）
postgres=#> RESET ROLE;
postgres=# GRANT PROXY ON test_proxy_u1 TO test_proxy_u2;
postgres=# SET ROLE test_proxy_u2 PASSWORD 'test_proxy_u2@123';
postgres=#>  SELECT * FROM tst_schema1.tst_t1;
    id    |     name
----------+--------------
 20220101 | proxy_example

 --权限检测（级联式检测 usr_1->usr_2->usr_3）
 postgres=#> RESET ROLE;
```

```
postgres=# GRANT PROXY ON test_proxy_u2 TO test_proxy_u3;
postgres=# SET ROLE test_proxy_u3 PASSWORD 'test_proxy_u3@123';
postgres=#>  SELECT * FROM tst_schema1.tst_t1;
    id    |     name
----------+--------------
 20220101 | proxy_example

--对被代理者授予的权限检测 (with grant option)
postgres=#> RESET ROLE;
postgres=# SET ROLE test_proxy_u2 PASSWORD 'test_proxy_u2@123';
postgres=#> grant proxy on test_proxy_u1 to test_proxy_u3;
ERROR:  must have admin option on role "test_proxy_u1"
postgres=#> RESET ROLE;
RESET
postgres=# SET ROLE test_proxy_u2 PASSWORD 'test_proxy_u2@123';
SET
postgres=#> grant proxy on test_proxy_u1 to test_proxy_u3;
ERROR:  must have admin option on role "test_proxy_u1"
postgres=#> RESET ROLE;
postgres=# grant proxy on test_proxy_u1 to test_proxy_u2 with grant option;
postgres=# SET ROLE test_proxy_u2 PASSWORD 'test_proxy_u2@123';
postgres=#> grant proxy on test_proxy_u1 to test_proxy_u3;
--召回代理权限测试
postgres=#> revoke proxy on test_proxy_u1 from test_proxy_u3;
postgres=#> revoke proxy on test_proxy_u1 from test_proxy_u2;
postgres=#>  SET ROLE test_proxy_u3 password 'test_proxy_u3@123';
postgres=#> SELECT * FROM tst_schema1.tst_t1;
ERROR:  permission denied for relation tst_t1
DETAIL:  N/A
```

2.4.1.5.33 INSERT

功能描述

向表中添加一行或多行数据。

注意事项

本章节只包含 dolphin 新增的语法，原 GBase 8c 的语法未做删除和修改。

增加 values()空值插入操作，根据 sql_mode 的不同会有不一样的插入效果。

新增 set_clause_values 子句。

语法格式

[WITH [RECURSIVE] with_query [, ...]]

INSERT [/*+ plan_hint */] [IGNORE] [INTO] table_name [partition_clause] [AS alias] [(column_name [, ...])]

{ DEFAULT VALUES

| [VALUES | VALUE] [{({ expression | DEFAULT } [, ...]) }][, ...]

| query

| set_clause_values }

[ON DUPLICATE KEY UPDATE { NOTHING | { column_name = { expression | DEFAULT } } [, ...] [WHERE condition] }]

[RETURNING {* | {output_expression [[AS] output_name] }[, ...]}];

参数说明

IGNORE

若带有 IGNORE 关键字的 INSERT 语句执行时在指定场景引发了 Error，则会将 Error 降级为 Warning，且继续语句的执行，不会影响其他数据的操作。能使 Error 降级的场景有：

1.违反非空约束时：

若执行的 SQL 语句违反了表的非空约束，使用此 hint 可将 Error 降级为 Warning，并根据 GUC 参数 sql_ignore_strategy 的值采用以下策略的一种继续执行：

sql_ignore_startegy 为 ignore_null 时，忽略违反非空约束的行的 INSERT 操作，并继续执行剩余数据操作。

sql_ignore_startegy 为 overwrite_null 时，将违反约束的 null 值覆写为目标类型的默认值，并继续执行剩余数据操作。


说明：

GUC 参数 sql_ignore_strategy 为枚举类型，可选值有：ignore_null, overwrite_null

2.违反唯一约束时：

若执行的 SQL 语句违反了表的唯一约束，使用此 hint 可将 Error 降级为 Warning，忽略违反约束的行的 INSERT 操作，并继续执行剩余数据操作。

3.分区表无法匹配到合法分区时：

在对分区表进行 INSERT 操作时，若某行数据无法匹配到表格的合法分区，使用此 hint 可将 Error 降级为 Warning，忽略该行操作，并继续执行剩余数据操作。

4.插入值向目标列类型转换失败时：

执行 INSERT 语句时，若发现新值与目标列类型不匹配，使用此 hint 可将 Error 降级为 Warning，并根据新值与目标列的具体类型采取以下策略的一种继续执行：

当新值类型与列类型同为数值类型时：

若新值在列类型的范围内，则直接进行插入；若新值在列类型范围外，则以列类型的最大/最小值替代。

当新值类型与列类型同为字符串类型时：

若新值长度在列类型限定范围内，则以直接进行插入；若新值长度在列类型的限定范围外，则保留列类型长度限制的前 n 个字符。

若遇到新值类型与列类型不可转换时：

插入列类型的默认值。

IGNORE 关键字不支持列存，无法在列存表中生效。

VALUES()

当 GUC 参数 sql_mode 为 stric_all_tables 时，为所有列插入 NULL，否则如果对应字段名有缺省值，插入缺省值，如果没有缺省值，判断对应字段是否有 not_null 约束，没有插入 NULL，有则插入类型基础值，具体基础值参考视图 pg_type_basic_value。

set_clause_values

一种 insert into table_name set column_name = value, column_name = value, …依次类推。set_clause_values 是指 set column_name = value，多个列插入值用逗号分隔。 该是 insert into 的一种扩展语法。为防止 insert into 时字段顺序与值顺序混乱造成写入错误。

示例

IGNORE 关键字

为使用 ignore_error hint，需要创建 B 兼容模式的数据库，名称为 db_ignore。

```
create database db_ignore dbcompatibility 'B';
\c db_ignore
忽略非空约束
db_ignore=# create table t_not_null(num int not null);
```

```
CREATE TABLE
-- 采用忽略策略
db_ignore=# set sql_ignore_strategy = 'ignore_null';
SET
db_ignore=# insert /*+ ignore_error */ into t_not_null values(null), (1);
WARNING:  null value in column "num" violates not-null constraint
DETAIL:  Failing row contains (null).
INSERT 0 1
db_ignore=# select * from t_not_null ;
 num
-----
   1
(1 row)
-- 采用覆写策略
db_ignore=# delete from t_not_null;
db_ignore=# set sql_ignore_strategy = 'overwrite_null';
SET
db_ignore=# insert /*+ ignore_error */ into t_not_null values(null), (1);
WARNING:  null value in column "num" violates not-null constraint
DETAIL:  Failing row contains (null).
INSERT 0 2
db_ignore=# select * from t_not_null ;
 num
-----
   0
   1
(2 rows)
忽略唯一约束
db_ignore=# create table t_unique(num int unique);
NOTICE:  CREATE TABLE / UNIQUE will create implicit index "t_unique_num_key" for
table "t_unique"
CREATE TABLE
db_ignore=# insert into t_unique values(1);
INSERT 0 1
db_ignore=# insert /*+ ignore_error */ into t_unique values(1),(2);
WARNING:  duplicate key value violates unique constraint in table "t_unique"
INSERT 0 1
db_ignore=# select * from t_unique;
 num
-----
   1
```

```
   2
(2 rows
忽略分区表无法匹配到合法分区
db_ignore=# CREATE TABLE t_ignore
db_ignore-# (
db_ignore(#     col1 integer NOT NULL,
db_ignore(#     col2 character varying(60)
db_ignore(# ) WITH(segment = on) PARTITION BY RANGE (col1)
db_ignore-# (
db_ignore(#     PARTITION P1 VALUES LESS THAN(5000),
db_ignore(#     PARTITION P2 VALUES LESS THAN(10000),
db_ignore(#     PARTITION P3 VALUES LESS THAN(15000)
db_ignore(# );
CREATE TABLE
db_ignore=# insert /*+ ignore_error */ into t_ignore values(20000);
WARNING:  inserted partition key does not map to any table partition
INSERT 0 0
db_ignore=# select * from t_ignore ;
 col1 | col2
------+------
(0 rows)
插入值向目标列类型转换失败
-- 当新值类型与列类型同为数值类型
db_ignore=# create table t_tinyint(num tinyint);
CREATE TABLE
db_ignore=# insert /*+ ignore_error */ into t_tinyint values(10000);
WARNING:  tinyint out of range
CONTEXT:  referenced column: num
INSERT 0 1
db_ignore=# select * from t_tinyint;
 num
-----
 127
(1 row)
-- 当新值类型与列类型同为字符类型时
db_ignore=# create table t_varchar5(content varchar(5));
CREATE TABLE
db_ignore=# insert /*+ ignore_error */ into t_varchar5 values('abcdefghi');
WARNING:  value too long for type character varying(5)
CONTEXT:  referenced column: content
INSERT 0 1
```

```
db_ignore=# select * from t_varchar5 ;
 content
---------
 abcde
(1 row)
--创建表 value_test。
postgres=# create table value_test(a int not null, b int default 3);
--向表中插入 VALUES()。
postgres=# insert into value_test values();
ERROR:  null value in column "a" violates not-null constraint
--关闭 sql_mode,向表中插入 VALUES()。
postgres=# set dolphin.sql_mode = '';
postgres=# insert into value_test values();
--查看表数据
postgres=# select * from value_test;
 a | b
---+---
 0 | 3
(1 row)
--删除表 value_test。
postgres=# DROP TABLE 删除表 value_test。;
```

相关链接

INSERT

2.4.1.5.34 KILL

功能描述

终止指定连接或该连接下执行的 SQL 语句。

注意事项

KILL 语法在非线程池模式和线程池模式下均有效。

一般结合 SHOW PROCESSSLIST 的查询结果 Id 字段使用。

也可以结合 select sessionid from pg_stat_activity where (过滤条件) 使用

语法格式

KILL [CONNECTION | QUERY] processlist_id

参数说明

CONNECTION

使用 CONNECTION 关键字修饰 KILL 语句时，效果等价于 KILL processlist_id，终止当前连接。

QUERY

使用 QUERY 关键字修饰 KILL 语句时，终止当前连接执行的 SQL 语句，连接本身不受影响。

processlist_id

连接 Id。

示例

```
--查看当前连接
postgres=# show processlist;
       Id        |       Pid       |      QueryId      | UniqueSqlId |   User
| Host |    db    |        Cgbaseand         |
     BackendStart          |          XactStart           |  Time  | State
|                 Info
---------------+---------------+-----------------+------------+-------
----+------+---------+----------------------+---
-------------------------+-----------------------------+--------+-------
-+-----------------------------------
 139653370304256 | 139653370304256 |                0 |           0 | GBase 8c
|       | postgres | ApplyLauncher          | 20
22-06-21 16:46:19.656076+08 |                              |        |
|
 139653319255808 | 139653319255808 |                0 |           0 | GBase 8c
|       | postgres | Asp                    | 20
22-06-21 16:46:19.728521+08 |                              |      1 | active |
 139653336483584 | 139653336483584 |                0 |           0 | GBase 8c
|       | postgres | PercentileJob          | 20
22-06-21 16:46:19.728527+08 |                              |      8 | active |
 139653302175488 | 139653302175488 |                0 |           0 | GBase 8c
|       | postgres | statement flush thread | 20
22-06-21 16:46:19.728558+08 |                              | 508507 | idle   |
 139653198239488 | 139653198239488 |                0 |           0 | GBase 8c
|       | postgres | WorkloadMonitor        | 20
22-06-21 16:46:19.750133+08 |                              |        |
|
 139653181298432 | 139653181298432 |                0 |           0 | GBase 8c
|       | postgres | WLMArbiter             | 20
```

```
22-06-21 16:46:19.750976+08 |                                  |         |
|
 139653215110912 | 139653215110912 |                 0 |           0 | GBase 8c
|       | postgres | workload                | 20
22-06-21 16:46:19.754504+08 | 2022-06-21 16:46:19.769585+08 | 508507 | active |
WLM fetch collect info from data nodes
 139653421840128 | 139653421840128 |                 0 |           0 | GBase 8c
|       | postgres | JobScheduler            | 20
22-06-27 10:00:54.754007+08 |                                  |       0 | active |
 139653044328192 | 139653044328192 | 48976645947655327 |  1772643515 | GBase 8c
| -1    | dolphin  | gsql                    | 20
22-06-27 14:00:53.163338+08 | 2022-06-27 14:01:26.794658+08 |       0 | active |
show processlist;
 139653027546880 | 139653027546880 | 48976645947655326 |  1775585557 | GBase 8c
| -1    | postgres | gsql                    | 20
22-06-27 14:01:03.969962+08 | 2022-06-27 14:01:19.967521+08 |       7 | active |
select pg_sleep(100);
(10 rows)
--终止 139653027546880 连接执行的 SQL 语句
postgres=# kill query 139653027546880;
 result
--------
 t
(1 row)
--查看 processlist 的 139653027546880 连接状态，已经变为 idle
postgres=# show processlist;
       Id        |       Pid        |      QueryId       | UniqueSqlId |   User
| Host |    db     |        Cgbaseand         |
      BackendStart          |           XactStart           |  Time   | State
|                 Info
----------------+----------------+------------------+------------+-------
----+------+---------+-----------------------+---
--------------------------+-------------------------------+--------+-------
-+----------------------------------------
 139653370304256 | 139653370304256 |                 0 |           0 | GBase 8c
|       | postgres | ApplyLauncher           | 20
22-06-21 16:46:19.656076+08 |                                  |         |
|
 139653319255808 | 139653319255808 |                 0 |           0 | GBase 8c
|       | postgres | Asp                     | 20
22-06-21 16:46:19.728521+08 |                                  |       0 | active |
```

```
 139653336483584 | 139653336483584 |                 0 |           0 | GBase 8c
|       | postgres | PercentileJob           | 20
22-06-21 16:46:19.728527+08 |                                 |       5 | active |
 139653302175488 | 139653302175488 |                 0 |           0 | GBase 8c
|       | postgres | statement flush thread | 20
22-06-21 16:46:19.728558+08 |                                 | 508573 | idle   |
 139653198239488 | 139653198239488 |                 0 |           0 | GBase 8c
|       | postgres | WorkloadMonitor         | 20
22-06-21 16:46:19.750133+08 |                                 |        |
|
 139653181298432 | 139653181298432 |                 0 |           0 | GBase 8c
|       | postgres | WLMArbiter              | 20
22-06-21 16:46:19.750976+08 |                                 |        |
|
 139653215110912 | 139653215110912 |                 0 |           0 | GBase 8c
|       | postgres | workload                | 20
22-06-21 16:46:19.754504+08 | 2022-06-21 16:46:19.769585+08 | 508573 | active |
WLM fetch collect info from data nodes
 139653421840128 | 139653421840128 |                 0 |           0 | GBase 8c
|       | postgres | JobScheduler            | 20
22-06-27 10:00:54.754007+08 |                                 |       1 | active |
 139653044328192 | 139653044328192 | 48976645947655329 |  1772643515 | GBase 8c
| -1    | dolphin  | gsql                    | 20
22-06-27 14:00:53.163338+08 | 2022-06-27 14:02:33.180256+08 |       0 | active |
show processlist;
 139653027546880 | 139653027546880 |                 0 |           0 | GBase 8c
| -1    | postgres | gsql                    | 20
22-06-27 14:01:03.969962+08 |                                 |      11 | idle   |
select pg_sleep(100);
(10 rows)
--终止 139653027546880 连接
postgres=# kill 139653027546880;
 result
--------
 t
(1 row)
--或
postgres=# kill connection 139653027546880;
 result
--------
 t
```

```
(1 row)
--查看 processlist 中已经不存在该连接
postgres=# show processlist;
       Id        |       Pid       |      QueryId      | UniqueSqlId |   User
| Host |    db    |        Cgbaseand         |
     BackendStart          |          XactStart          |  Time   | State
|                 Info
----------------+---------------+-----------------+------------+-------
----+------+---------+----------------------+---
------------------------+----------------------------+--------+-------
-+---------------------------------------
 139653370304256 | 139653370304256 |                0 |          0 | GBase 8c
|       | postgres | ApplyLauncher          | 20
22-06-21 16:46:19.656076+08 |                                |        |
|
 139653319255808 | 139653319255808 |                0 |          0 | GBase 8c
|       | postgres | Asp                    | 20
22-06-21 16:46:19.728521+08 |                                |      1 | active |
 139653336483584 | 139653336483584 |                0 |          0 | GBase 8c
|       | postgres | PercentileJob          | 20
22-06-21 16:46:19.728527+08 |                                |      7 | active |
 139653302175488 | 139653302175488 |                0 |          0 | GBase 8c
|       | postgres | statement flush thread | 20
22-06-21 16:46:19.728558+08 |                                | 508696 | idle   |
 139653198239488 | 139653198239488 |                0 |          0 | GBase 8c
|       | postgres | WorkloadMonitor        | 20
22-06-21 16:46:19.750133+08 |                                |        |
|
 139653181298432 | 139653181298432 |                0 |          0 | GBase 8c
|       | postgres | WLMArbiter             | 20
22-06-21 16:46:19.750976+08 |                                |        |
|
 139653215110912 | 139653215110912 |                0 |          0 | GBase 8c
|       | postgres | workload               | 20
22-06-21 16:46:19.754504+08 | 2022-06-21 16:46:19.769585+08 | 508696 | active |
WLM fetch collect info from data nodes
 139653421840128 | 139653421840128 |                0 |          0 | GBase 8c
|       | postgres | JobScheduler           | 20
22-06-27 10:00:54.754007+08 |                                |      1 | active |
 139653044328192 | 139653044328192 | 48976645947655331 |  1772643515 | GBase 8c
| -1    | dolphin  | gsql                   | 20
```

```
22-06-27 14:00:53.163338+08 | 2022-06-27 14:04:35.418518+08 |       0 | active |
show processlist;
(9 rows)
```

2.4.1.5.35 LOAD-DATA

功能描述

通过 LOAD DATA 命令实现从一个文件拷贝数据到一个表。

注意事项

当参数 enable_copy_server_files 关闭时，只允许初始用户执行 LOAD DATA 命令，当参数 enable_copy_server_files 打开，允许具有 SYSADMIN 权限的用户或继承了内置角色 gs_role_copy_files 权限的用户执行，但默认禁止对数据库配置文件、密钥文件、证书文件和审计日志执行，以防止用户越权查看或修改敏感文件。

只能用于表，不能用于视图。

不支持列存表和外表。

需要插入的表的 insert 权限，replace 选项还需要表的 delete 权限。

如果声明了一个字段列表，LOAD 将只在文件和表之间拷贝已声明字段的数据。如果表中有任何不在字段列表里的字段，将为那些字段插入缺省值。

声明的数据源文件，服务器必须可以访问该文件。

如果数据文件的任意行包含比预期多或者少的字段，dolphin.sql_mode 为严格模式时将抛出一个错误，宽松模式时缺少的字段将插入 NULL,如果字段有 NOT NULL 约束则会插入类型基础值。

\N 为 NULL，如果要输入实际数据值\N ，使用\\N。

语法格式

```
LOAD DATA
INFILE 'file_name'
[REPLACE | IGNORE]
INTO TABLE tbl_name
[CHARACTER SET charset_name]
[{FIELDS | COLUMNS}
[TERMINATED BY 'string']
[[OPTIONALLY] ENCLOSED BY 'char']
[ESCAPED BY 'char']
```

```
        ]
        [LINES
        [STARTING BY 'string']
        [TERMINATED BY 'string']
        ]
        [IGNORE number {LINES | ROWS}]
        [(col_name_or_user_var
        [, col_name_or_user_var] ...)]
```

参数说明

REPLACE

插入的数据发生主键或唯一键冲突时才会起作用，会先将表中冲突的行进行删除，之后继续插入的数据。

IGNORE

插入的数据发生主键或唯一键冲突时才会起作用，会忽略冲突行文件数据，继续插入后续的数据。

tbl_name

表的名称（可以有模式修饰）。

取值范围：已存在的表名。

col_name

可选的待拷贝字段列表。

取值范围：如果没有声明字段列表，将使用所有字段。

ESCAPED BY 'char'

用来指定逃逸字符，逃逸字符只能指定为单字节字符。

默认值为双引号。当与 ENCLOSED BY 值相同时，会被替换为'\0'。

LINES TERMINATED BY 'string'

指定导出数据文件换行符样式。

取值范围：支持多字符换行符，但换行符不能超过 10 个字节。常见的换行符，如\r、\n、\r\n（设成 0x0D、0x0A、0x0D0A 效果是相同的），其他字符或字符串，如$、#。

说明：

LINES TERMINATED BY 参数不能和分隔符、null 参数相同。

LINES TERMINATED BY 参数不能包含：.abcdefghijklmnopqrstuvwxyz0123456789。

CHARACTER SET 'charset_name'

指定文件编码格式名称。

取值范围：有效的编码格式。

缺省值：当前编码格式。

[OPTIONALLY] ENCLOSED BY 'char'

指定包裹符，完整包裹符内的数据将被当成一列的参数进行解析，OPTIONALLY 没有实际意义。

缺省值：双引号。

📖 说明：

ENCLOSED BY 参数不能和分隔符参数相同。

ENCLOSED BY 参数只能是单字节的字符。

FIELDS | COLUMNS TERMINATED BY 'string'

在文件中分隔各个字段的字符串，分隔符最大长度不超过 10 个字节。

缺省值：缺省是水平制表符。

IGNORE number {LINES | ROWS}

指定数据导出时，跳过数据文件的前 number 行。

示例

```
--创建 load_t1 表。
postgres=# CREATE TABLE load_t1
(
    SM_SHIP_MODE_SK           INTEGER               NOT NULL,
    SM_SHIP_MODE_ID           CHAR(16)              NOT NULL,
    SM_TYPE                   CHAR(30)                      ,
    SM_CODE                   CHAR(10)                      ,
    SM_CARRIER                CHAR(20)                      ,
    SM_CONTRACT               CHAR(20)
);
```

```
--/home/gbase/test.csv 文件
1,a,b,c,d,e
,a,b,c,d,e
3,\N,a,b,c,d
\N,a,b,c,d,e
--从/home/gbase/test.csv 文件拷贝数据到表 load_t1。
postgres=# LOAD DATA INFILE '/home/gbase/test.csv' INTO TABLE load_t1;
--从/home/gbase/test.csv 文件拷贝数据到表 load_t1，使用参数如下：字段分隔符为'\t'（fields terminated by E'\t'）换行符为'\r'（lines terminated by E'\r'）跳过前两行（IGNORE 2 LINES)。
postgres=# LOAD DATA INFILE '/home/gbase/test.csv' INTO TABLE load_t1 fields terminated by ',' lines terminated by E'\n' IGNORE 2 LINES;
postgres=# select * from load_t1;
 sm_ship_mode_sk | sm_ship_mode_id  |            sm_type             |
sm_code    |      sm_carrier      |     sm_contract
----------------+------------------+--------------------------------+-------
-----+----------------------+----------------------
               3 |                  | a                              | b
| c                    | d
               0 | a                | b                              | c
| d                    | e
(2 rows)
--删除 load_t1。
postgres=# DROP TABLE load_t1;
```

2.4.1.5.36 OPTIMIZE-TABLE

功能描述

重建表和索引的物理空间，释放可回收空间给操作系统，并更新相关表的统计信息。

注意事项

需要表 vacuum/owner 或 superuser 权限。

b 数据库的 optimize 操作支持多表，GBase 8c 的 optimize 操作只支持单表。

b 数据库的 optimize 属于 Online DDL 操作，处理过程的主要阶段不影响表的读写；但 GBase 8c 的 optimize 会阻塞表的读写，表数据量较大时可能会存在长时间锁表情况，请谨慎操作。

optimize 也会被其他事务或两阶段事务阻塞。

尽量不要并发执行 optimize 多张表，如果需要并发执行，请降低并发数量，一般在 3

以下。

执行 optimize 时需要确保数据目录当前剩余空间大于该表当前占用空间，不然可能会失败。

表在短时间内删除大量数据后不要立即执行 optimize，尝试等待几秒或执行若干其他事务后再执行，不然可能出现元组处于 HEAPTUPLE_RECENTLY_DEAD 状态无法正常回收的情况。

语法格式

```
OPTIMIZE [VERBOSE] [NO_WRITE_TO_BINLOG | LOCAL] TABLE tbl_name
```

参数说明

[VERBOSE]

查看 optimize 处理详情，可缺省。

[NO_WRITE_TO_BINLOG | LOCAL]

仅兼容语法，无实际效果，可缺省。

tbl_name

表名，可指定表名。也可以指定 schema_name.table_name。

示例

```
--创建 doc 表
postgres=# create table doc(id serial primary key, content varchar(255));
NOTICE:  CREATE TABLE will create implicit sequence "doc_id_seq" for serial
column "doc.id"
NOTICE:  CREATE TABLE / PRIMARY KEY will create implicit index "doc_pkey" for
table "doc"
CREATE TABLE
--插入 10000 条数据
postgres=# insert into doc(content) select 'abcd1234' from
generate_series(1,10000) as content;
INSERT 0 100000
--删除 9000 条数据
postgres=# delete from doc where id <= 9000;
DELETE 9000
--optimize 表
postgres=# optimize table doc;
VACUUM
```

```
--optimize 表（查看处理详情）
postgres=# optimize verbose table doc;
INFO:  vacuuming "public.doc"(primary pid=24692)
INFO:  "doc": found 9000 removable, 1000 nonremovable row versions in 55
pages(primary pid=24692)
DETAIL:  0 dead row versions cannot be removed yet.
CPU 0.00s/0.04u sec elapsed 0.04 sec.
INFO:  analyzing "public.doc"(primary pid=24692)
INFO:  ANALYZE INFO : "doc": scanned 6 of 6 pages, containing 1000 live rows and
0 dead rows; 1000 rows in sample, 1000 estimated total rows(primary pid=24692)
VACUUM
```

2.4.1.5.37 PREPARE

功能描述

创建一个预备语句。

预备语句是服务端的对象，可以用于优化性能。在执行 PREPARE 语句的时候，指定的查询被解析、分析、重写。当随后发出 EXECUTE 语句的时候，预备语句被规划和执行。这种设计避免了重复解析、分析工作。PREPARE 语句创建后在整个数据库会话期间一直存在，一旦创建成功，即便是在事务块中创建，事务回滚，PREPARE 也不会删除。只能通过显式调用 DEALLOCATE 进行删除，会话结束时，PREPARE 也会自动删除。

注意事项

相比于原始的 GBase 8c，dolphin 对于 PREPARE 语法的修改为：

支持 PREPARE FROM 语法。

statement 支持加单引号，且单引号内的 statement 必须是单个 query。加单引号的场景，statement 除了 SELECT、INSERT、UPDATE、DELETE、MERGE INTO 或 VALUES 语句外，还支持其他最终会转换成 SelectStmt 的语句，如部分 SHOW 系列语句等。

statement 中的绑定参数支持使用?，需要先将 dolphin.b_compatibility_mode 设置为 on，且不能同时在一个语句中同时使用$和?作为参数占位符。将 dolphin.b_compatibility_mode 设置为 on 后，?将不能作为操作符使用。

语法格式

```
PREPARE name [ ( data_type [, ...] ) ] { AS | FROM } statement;
PREPARE name [ ( data_type [, ...] ) ] { AS | FROM } 'statement';
```

参数说明

name

指定预备语句的名称。它必须在该会话中是唯一的。

data_type

参数的数据类型。

statement

是 SELECT INSERT、UPDATE、DELETE、MERGE INTO 或 VALUES 语句之一。

示例

```
postgres=# CREATE TABLE test(name text, age int);
CREATE TABLE
postgres=# INSERT INTO test values('a',18);
INSERT 0 1
postgres=# PREPARE stmt FROM SELECT * FROM test;
PREPARE
postgres=# EXECUTE stmt;
 name | age
------+-----
 a    |  18
(1 row)
postgres=# set dolphin.b_compatibility_mode to on;
SET
postgres=# PREPARE stmt1 FROM 'SELECT sqrt(pow(?,2) + pow(?,2)) as test';;
PREPARE
postgres=# EXECUTE stmt1 USING 6,8;
 test
------
   10
(1 row)
```

2.4.1.5.38 RENAME-TABLE

功能描述

修改表名，包括修改表的 schema、重命名表、删除表的权限。

```
RENAME TABLE old_table to new_table;
```

注意事项

本章节只包含 dolphin 新增的语法，原 GBase 8c 的语法未做删除和修改。

当 rename table 语句下有多条修改表名命令时，该语法会对要修改的表名进行排序，然后按 顺序加锁，再按从左往右顺序修改表名，当 rename table a to b, b to c，中间表 b 不存在时，则不加锁跳过。

对于表不存在，和目标表与存在的表发生冲突时，则报相应的错误信息。当表有同义词时， 则原始表不能有同义词依赖，目标表不能存在有与之同名的同义词。

对目标表修改表名和模式时，会判断当前用户是否对该表拥有权限。

RENAME TABLE 语法格式

- 修改表的定义。

```
RENAME TABLE old_schema.table_name TO new_schema.new_table_name [,
old_schema.table_name TO new_schema.new_table_name ...];
```

参数说明

RENAME TABLE 可以同时重命名一张或者多张表。但必须有对旧表 ALTER 和 DROP 的权限，和对 新表 CREATE 和 INSERT 的权限。且必须对 old_schema 和 new_schema 有权限。

- 修改表名

```
RENAME TABLE old_table to new_table;
```

- 当旧表和新表在同一 schema 下，其等同于

```
ALTER TABLE old_table RENAME TO new_table;
```

- RENAME TABLE 支持在一条 sql 语法中修改多张表名，且其执行顺序是从左到右。

```
RENAME TABLE A TO B, B TO C, C TO A;
```

- RENAME TABLE 包含有锁表操作，其对表加锁顺序，是根据旧表的 schema.table 来排序，

然后依次进行对排序后的表加锁。 GBase 8c 中的跨 schema 修改表名，相当于 mysql 中跨 库修改表名。

```
RENAME TABLE old_schema.old_table TO new_schema.new_table;
```

old_table 中不能是同义词，且不能存储同义词依赖。 new_table 不能是同义词。RENAME TABLE 在修改表名，同时也会修改系统表 pg_type 里与 old_table 同名的数据类 型。和系统表 pg_depend 中的依赖。

不支持临时表和全局临时表。

不支持视图的跨 schema 修改表名，只支持在同一 schema 下表名的修改。

RENAME TABLE 改完表名后， new_table 不具备 old_table 的权限。必须用超级用户重新给 new_table 分配权限。

REANME TABLE 语法， old_table 如果具备触发器，则 old_table 不能跨 schema 修改表名。 old_table 和 new_table 不能前后一样。

RENAME TABLE 语法如果 old_table 不指定 schema，则从 search_ path 中遍历，直到找到 old_table 为止，否则报错 old_table 不存在。如果 new_table 不指定 schema，则表示 new_table 与 old_table 处于同一 schema 中。

相关链接

ALTER TABLE

2.4.1.5.39 RENAME-USER

功能描述

修改数据库中的用户名。

注意事项

RENAME USER 会修改用户名，并且只能修改当前表的 user 的名称。

如果修改多个用户，并且其中一个用户名不存在或其他原因导致执行失败，整条语句都会失败，所有用户名都会保持不变。

与 ALTER USER ... RENAME TO ...等价。

语法格式

```
RENAME USER
    old_user1 TO new_user1,
    old_user2 TO new_user2,
    ...
```

参数说明

old_user 旧的用户名，必须已存在

new_user 新的用户名

示例

```
rename user
    user1 to user4,
```

```
    user2 to user5,
    user3 to user6;
```

相关链接

ALTER USER

2.4.1.5.40 REVOKE

功能描述

REVOKE 用于撤销一个或多个角色的权限。

注意事项

本章节只包含 dolphin 新增的语法，原 GBase 8c 的语法未做删除和修改。 增加 ALTER ROUTINE、CRAETE ROUTINE、CREATE TEMPORARY TABLES、CREATE USER、CREATE TABLESPACE、INDEX 权限

语法格式

新增 ALTER ROUTINE 权限

与 function 和 procedure 的 alter 权限基本一致

修改后的语法说明为：

```
REVOKE { { EXECUTE | ALTER ROUTINE | ALTER | DROP | COMMENT } [, ...] | ALL
[ PRIVILEGES ] }
ON {FUNCTION {function_name ( [ {[ argmode ] [ arg_name ] arg_type} [, ...] ] )}
| PROCEDURE {proc_name ( [ {[ argmode ] [ arg_name ] arg_type} [, ...] ] )} [, ...]
| ALL FUNCTIONS IN SCHEMA schema_name [, ...] | ALL PROCEDURE IN SCHEMA schema_name
[, ...] | schema_name.*}
FROM { [ GROUP ] role_name | PUBLIC } [, ...]
[ WITH GRANT OPTION ];
```

新增 CREATE ROUTINE 权限

与 CREATE ANY FUNCTION 权限基本一致

修改后的语法说明为：

```
REVOKE { CREATE ANY TABLE | ALTER ANY TABLE | DROP ANY TABLE | SELECT ANY TABLE
| INSERT ANY TABLE | UPDATE ANY TABLE |
  DELETE ANY TABLE | CREATE ANY SEQUENCE | CREATE ANY INDEX | CREATE ANY FUNCTION
| CREATE ROUTINE | EXECUTE ANY FUNCTION |
  CREATE ANY PACKAGE | EXECUTE ANY PACKAGE | CREATE ANY TYPE } [, ...]
```

```
  [ON *.*]
  FROM [ GROUP ] role_name [, ...]
  [ WITH ADMIN OPTION ];
```

新增 CREATE TEMPORARY TABLES 权限

与 TEMPORARY 权限基本一致

修改后的语法说明为：

```
REVOKE { { CREATE | CONNECT | CREATE TEMPORARY TABLES | TEMPORARY | TEMP | ALTER
| DROP | COMMENT } [, ...]
    | ALL [ PRIVILEGES ] }
    ON { DATABASE database_name [, ...] | database_name.* }
    FROM { [ GROUP ] role_name | PUBLIC } [, ...]
    [ WITH GRANT OPTION ];
```

新增 CREATE USER 权限

控制用户创建新用户的权限，与用户的 CREATEROLE 和 NOCREATEROLE 权限基本一致

新增的语法说明为：

```
REVOKE CREATE USER ON *.* FROM ROLE_NAME;
```

新增 CREATE TABLESPACE 权限

控制用户创建新表空间的权限

新增的语法说明为：

```
REVOKE CREATE TABLESPACE ON *.* FROM ROLE_NAME;
```

新增 INDEX 权限

与 CREATE ANY INDEX 权限基本一致

修改后的语法说明为：

```
REVOKE { CREATE ANY TABLE | ALTER ANY TABLE | DROP ANY TABLE | SELECT ANY TABLE
| INSERT ANY TABLE | UPDATE ANY TABLE |
  DELETE ANY TABLE | CREATE ANY SEQUENCE | CREATE ANY INDEX | INDEX | CREATE ANY
FUNCTION | EXECUTE ANY FUNCTION |
  CREATE ANY PACKAGE | EXECUTE ANY PACKAGE | CREATE ANY TYPE } [, ...]
  { ON *.* }
  FROM [ GROUP ] role_name [, ...]
  [ WITH ADMIN OPTION ];
```

参数说明

N/A

示例

```
REVOKE ALTER ROUTINE ON FUNCTION TEST FROM USER_TESTER;
REVOKE CREATE ANY FUNCTION FROM USER_TESTER;
REVOKE CREATE TEMPORARY TABLES ON DATABASE DATABASE_TEST FROM USER_TESTER;
REVOKE CREATE USER ON *.* FROM USER_TESTER;
REVOKE CREATE TABLESPACE ON *.* FROM USER_TESTER;
REVOKE INDEX FROM TEST_USER;
```

相关链接

REVOKE

2.4.1.5.41 SELECT

功能描述

SELECT 用于从表或视图中取出数据。

SELECT 语句就像叠加在数据库表上的过滤器，利用 SQL 关键字从数据表中过滤出用户需要的数据。

注意事项

对比原 GBase 8c 的 SELECT 语法，新增了 WHERE 子句下的 sounds like 语法。

新增 join 不带 on/using,效果与 cross join 一致。

新增 PARTITION 子句可指定多个分区。

新增 UNION 子句列如果没有相似的数据类型，会采取转换为 text 类型的方式进行处理。

新增 FROM DUAL 语法，含义等同于不写 FROM 子句，是为了满足那些要求所有 SELECT 语句都应该包含 FROM 的情况。

语法格式

查询数据

```
[ WITH [ RECURSIVE ] with_query [, ...] ]
SELECT [/*+ plan_hint */] [ ALL | DISTINCT [ ON ( expression [, ...] ) ] ]
{ * | {expression [ [ AS ] output_name ]} [, ...] }
[ FROM from_item [, ...] ]
[ WHERE condition ]
```

```
[ [ START WITH condition ] CONNECT BY [NOCYCLE] condition [ ORDER SIBLINGS BY
expression ] ]
[ GROUP BY grouping_element [, ...] ]
[ HAVING condition [, ...] ]
[ WINDOW {window_name AS ( window_definition )} [, ...] ]
[ { UNION | INTERSECT | EXCEPT | MINUS } [ ALL | DISTINCT ] select ]
[ ORDER BY {expression [ [ ASC | DESC | USING operator ] |
nlssort_expression_clause ] [ NULLS { FIRST | LAST } ]} [, ...] ]
[ LIMIT { [offset,] count | ALL } ]
[ OFFSET start [ ROW | ROWS ] ]
[ FETCH { FIRST | NEXT } [ count ] { ROW | ROWS } ONLY ]
[ {FOR { UPDATE | NO KEY UPDATE | SHARE | KEY SHARE } [ OF table_name [, ...] ]
[ NOWAIT ]} [...] ];
```

其中指定查询源 from_item 为：

```
{[ ONLY ] table_name [ * ] [ partition_clause ] [ [ AS ] alias [ ( column_alias
[, ...] ) ] ]
[ TABLESAMPLE sampling_method ( argument [, ...] ) [ REPEATABLE ( seed ) ] ]
[TIMECAPSULE {TIMESTAMP|CSN} expression]
|( select ) [ AS ] alias [ ( column_alias [, ...] ) ]
|with_query_name [ [ AS ] alias [ ( column_alias [, ...] ) ] ]
|function_name ( [ argument [, ...] ] ) [ AS ] alias [ ( column_alias [, ...] |
column_definition [, ...] ) ]
|function_name ( [ argument [, ...] ] ) AS ( column_definition [, ...] )
|from_item [ NATURAL ] join_type from_item [ ON join_condition | USING
( join_column [, ...] ) ]}
```

其中不写 FROM 子句的情况等价于：

```
FROM DUAL
```

其中 group 子句为：

```
( )
| expression
| ( expression [, ...] )
| rollup_clause
| CUBE ( { expression | ( expression [, ...] ) } [, ...] )
| GROUPING SETS ( grouping_element [, ...] )
```

其中指定分区 partition_clause 为：

```
PARTITION { ( partition_name [, ...] ) |
       FOR (  partition_value [, ...] ) }
```

说明：指定分区只适合分区表。

rollup_clause 子句为：

```
ROLLUP ( { expression | ( expression [, ...] ) } [, ...] )
| { expression | ( expression [, ...] ) } WITH ROLLUP
```

JOIN 语法

```
[JOIN | INNER JOIN] {ON join_condition | USING ( join_column [, ...] ) }
```

参数说明

WHERE 子句

sounds like 是 condition 的一种语法，用法如：column_name sounds like '字符'；相当于 soundex(column_name) = soundex('字符')的对比结果，是一个 boolean 的值。用于通过 soundex 处理来查询满足条件的数据。

where 子句可以包含兼容 MySQL 全文索引的查询语法。match(column_name [, …]) against ('匹配字符')也是 condition 的一种语法。

where match(column_name [, ...]) against ('匹配字符');

column_name 可以是多列，列名之间用逗号分隔。 against()的匹配字符只能是字符内容(即全文索引支持字段类型只能是这三种 char, varchar, text)，不包含 int, bool, 特殊字符(!,#, 空格等)与正规功能。 注意： mysql 的全文索引查询语法 match(column_name)允许无序，但该功能底层用的是 GBase 8c 的 to_tsvector(),他的要求是字段顺序必须有序(与表的字段顺序一致)。

用于安装了 dolphin 插件，处于 MySQL 兼容性场景下的全文索引查询。其语法结构相当于

to_tsvector('ngram', col_name [|| col_name]) @@ to_tsquery('字符串')

UNION 子句 UNION 计算多个 SELECT 语句返回行集合的并集。UNION 内部的 SELECT 语句必须拥有相同数量的列，列如果没有相似的数据类型或者为 UNKNOWN 类型，会采取转换为 text 类型的方式进行处理。

UNION 子句有如下约束条件：

除非声明了 ALL 子句，否则缺省的 UNION 结果不包含重复的行。

同一个 SELECT 语句中的多个 UNION 操作符是从左向右计算的，除非用圆括弧进行了

标识。

FOR UPDATE，FOR NO KEY UPDATE，FOR SHARE 和 FOR KEY SHARE 不能在 UNION 的结果或输入中声明。

一般表达式：

select_statement UNION [ALL] select_statement

select_statement 可以是任何没有 ORDER BY、LIMIT、FOR UPDATE，FOR NO KEY UPDATE，FOR SHARE 或 FOR KEY SHARE 子句的 SELECT 语句。

如果用圆括弧包围，ORDER BY 和 LIMIT 可以附着在子表达式里。

 说明：

涉及的其它参数说明可见 SELECT。

示例

SOUNDS LIKE 子句示例：同音字段查询

```
postgres=# CREATE TABLE TEST(id int, name varchar);
postgres=# INSERT INTO TEST VALUES(1, 'too');
postgres=# SELECT * FROM TEST WHERE name SOUNDS LIKE 'two';
 id | name
----+------
  1 | too
(1 row)
SELECT GROUP BY 子句中使用 ROLLUP
postgres=# CREATE TABLESPACE t_tbspace ADD DATAFILE 'my_tablespace' ENGINE =
test_engine;
CREATE TABLESPACE
postgres=# CREATE TABLE t_with_rollup(id int, name varchar(20), area varchar(50),
count int);
CREATE TABLE
postgres=# INSERT INTO t_with_rollup values(1, 'a', 'A', 10);
INSERT 0 1
postgres=# INSERT INTO t_with_rollup values(2, 'b', 'B', 15);
INSERT 0 1
postgres=# INSERT INTO t_with_rollup values(2, 'b', 'B', 20);
INSERT 0 1
postgres=# INSERT INTO t_with_rollup values(3, 'c', 'C', 50);
INSERT 0 1
```

```
postgres=# INSERT INTO t_with_rollup values(3, 'c', 'C', 15);
INSERT 0 1
postgres=# SELECT name, sum(count) FROM t_with_rollup GROUP BY ROLLUP(name);
 name | sum
------+-----
 a    |  10
 b    |  35
 c    |  65
      | 110
(4 rows)
postgres=# SELECT name, sum(count) FROM t_with_rollup GROUP BY (name) WITH ROLLUP;
 name | sum
------+-----
 a    |  10
 b    |  35
 c    |  65
      | 110
(4 rows)
postgres=# create table join_1(col1 int4, col2 int8);
postgres=# create table join_2(col1 int4, col2 int8);
postgres=# insert into join_1 values(1, 2), (3, 3);
postgres=# insert into join_2 values(1, 1), (2, 3), (4, 4);
postgres=# select join_1 join join_2;
 col1 | col2 | col1 | col2
------+------+------+------
    1 |    2 |    1 |    1
    1 |    2 |    2 |    3
    1 |    2 |    4 |    4
    3 |    3 |    1 |    1
    3 |    3 |    2 |    3
    3 |    3 |    4 |    4
postgres=# select join_1 inner join join_2;
 col1 | col2 | col1 | col2
------+------+------+------
    1 |    2 |    1 |    1
    1 |    2 |    2 |    3
    1 |    2 |    4 |    4
    3 |    3 |    1 |    1
    3 |    3 |    2 |    3
    3 |    3 |    4 |    4
```

SELECT 语句中使用 FROM DUAL 示例

```
postgres=# select 1 as col;
 col
-----
   1
(1 row)
postgres=# select 1 as col FROM DUAL;
 col
-----
   1
(1 row)
SELECT FROM PARTITION 子句指定多个分区
postgres=# create table multi_partition_select_test(C_INT INTEGER) partition by
range(C_INT)
postgres-# (
postgres(#     partition test_part1 values less than (400),
postgres(#     partition test_part2 values less than (700),
postgres(#     partition test_part3 values less than (1000)
postgres(# );
CREATE TABLE
postgres=# insert into multi_partition_select_test values(111);
INSERT 0 1
postgres=# insert into multi_partition_select_test values(555);
INSERT 0 1
postgres=# insert into multi_partition_select_test values(888);
INSERT 0 1
postgres=# select a.* from multi_partition_select_test partition (test_part1,
test_part2) a;
 c_int
-------
   111
   555
(2 rows)
UNION 子句非相似数据类型按 TEXT 类型进行转换示例:
-- 首先创建兼容模式为 B 的数据库
CREATE DATABASE mydb WITH DBCOMPATIBILITY 'B';
\c mydb
-- 创建两个表并插入测试数据
CREATE TABLE tbl_date(col DATE);
INSERT INTO tbl_date VALUES('2000-02-16');
CREATE TABLE tbl_json(col JSON);
INSERT INTO tbl_json VALUES('{"id":1,"dbname":"GBase 8c","language":"C++"}');
```

```
-- UNION 查询，将会使用 TEXT 类型进行转换
SELECT * FROM tbl_date UNION SELECT * FROM tbl_json;
兼容 MySQL 兼容性全文索引语法查询，前提是兼容模式为 B 的数据库。
postgres=# CREATE SCHEMA fulltext_test;
CREATE SCHEMA
postgres=# set current_schema to 'fulltext_test';
SET
postgres=# CREATE TABLE test (
postgres(# id int unsigned auto_increment not null primary key,
postgres(# title varchar,
postgres(# boby text,
postgres(# name name,
postgres(# FULLTEXT (title, boby) WITH PARSER ngram
postgres(# );
NOTICE:  CREATE TABLE will create implicit sequence "test_id_seq" for serial
column "test.id"
NOTICE:  CREATE TABLE / PRIMARY KEY will create implicit index "test_pkey" for
table "test"
CREATE TABLE
postgres=# \d test
              Table "fulltext_test.test"
 Column |       Type        |        Modifiers
--------+-------------------+-------------------------
 id     | uint4             | not null AUTO_INCREMENT
 title  | character varying |
 boby   | text              |
 name   | name              |
Indexes:
    "test_pkey" PRIMARY KEY, btree (id) TABLESPACE pg_default
    "test_to_tsvector_to_tsvector1_idx" gin (to_tsvector('ngram'::regconfig,
title::text), to_tsvector('ngram'::regconfig, boby)) TABLESPACE pg_default
postgres=# \d test_to_tsvector_to_tsvector1_idx
      Index "fulltext_test.test_to_tsvector_to_tsvector1_idx"
    Column     | Type |                    Definition
---------------+------+-----------------------------------------------
 to_tsvector  | text | to_tsvector('ngram'::regconfig, title::text)
 to_tsvector1 | text | to_tsvector('ngram'::regconfig, boby)
gin, for table "fulltext_test.test"
postgres=# DROP INDEX test_to_tsvector_to_tsvector1_idx;
DROP INDEX
```

```
postgres=# ALTER TABLE test ADD FULLTEXT INDEX test_index_1 (title, boby) WITH
PARSER ngram;
ALTER TABLE
postgres=# DROP INDEX test_index_1;
DROP INDEX
postgres=# CREATE FULLTEXT INDEX test_index_1 ON test (title, boby) WITH PARSER
ngram;
CREATE INDEX
postgres=# \d test_index_1
                 Index "fulltext_test.test_index_1"
    Column     | Type |                   Definition
---------------+------+-----------------------------------------------
 to_tsvector  | text | to_tsvector('ngram'::regconfig, title::text)
 to_tsvector1 | text | to_tsvector('ngram'::regconfig, boby)
gin, for table "fulltext_test.test"
postgres=# INSERT INTO test(title, boby, name) VALUES('test', '&67575@gauss',
'GBase 8c');
INSERT 0 1
postgres=# INSERT INTO test(title, boby, name) VALUES('test1', 'gauss', 'GBase
8c');
INSERT 0 1
postgres=# INSERT INTO test(title, boby, name) VALUES('test2', 'gauss2', 'GBase
8c');
INSERT 0 1
postgres=# INSERT INTO test(title, boby, name) VALUES('test3', 'test', 'GBase
8c');
INSERT 0 1
postgres=# INSERT INTO test(title, boby, name) VALUES('gauss_123_@', 'test',
'GBase 8c');
INSERT 0 1
postgres=# INSERT INTO test(title, boby, name) VALUES('', '', 'GBase 8c');
INSERT 0 1
postgres=# INSERT INTO test(title, boby, name) VALUES(' ', ' ', ' ');
INSERT 0 1
postgres=# SELECT * FROM TEST;
 id |    title    |     boby     |   name
----+-------------+--------------+----------
 1  | test        | &67575@gauss | GBase 8c
 2  | test1       | gauss        | GBase 8c
 3  | test2       | gauss2       | GBase 8c
 4  | test3       | test         | GBase 8c
```

```
 5  | gauss_123_@ | test         | GBase 8c
 6  |             |              | GBase 8c
 7  |             |              |
(7 rows)
postgres=# SELECT * FROM TEST WHERE MATCH (title, boby) AGAINST ('test');
 id |    title    |     boby     |   name
----+-------------+--------------+----------
 1  | test        | &67575@gauss | GBase 8c
 2  | test1       | gauss        | GBase 8c
 3  | test2       | gauss2       | GBase 8c
 4  | test3       | test         | GBase 8c
 5  | gauss_123_@ | test         | GBase 8c
(5 rows)
postgres=# SELECT * FROM TEST WHERE MATCH (title, boby) AGAINST ('gauss');
 id |    title    |     boby     |   name
----+-------------+--------------+----------
 1  | test        | &67575@gauss | GBase 8c
 2  | test1       | gauss        | GBase 8c
 3  | test2       | gauss2       | GBase 8c
 5  | gauss_123_@ | test         | GBase 8c
(4 rows)
postgres=# DROP INDEX test_index_1;
DROP INDEX
postgres=# CREATE FULLTEXT INDEX test_index_1 ON test (boby) WITH PARSER ngram;
CREATE INDEX
postgres=# \d test_index_1
             Index "fulltext_test.test_index_1"
   Column    | Type |                Definition
-------------+------+-----------------------------------
 to_tsvector | text | to_tsvector('ngram'::regconfig, boby)
gin, for table "fulltext_test.test"

postgres=# SELECT * FROM test WHERE MATCH (boby) AGAINST ('test');
 id |    title    | boby |   name
----+-------------+------+----------
 4  | test3       | test | GBase 8c
 5  | gauss_123_@ | test | GBase 8c
(2 rows)
postgres=# SELECT * FROM test WHERE MATCH (boby) AGAINST ('gauss');
 id | title |     boby     |   name
----+-------+--------------+----------
```

```
 1  | test   | &67575@gauss | GBase 8c
 2  | test1  | gauss        | GBase 8c
 3  | test2  | gauss2       | GBase 8c
(3 rows)
postgres=# DROP INDEX test_index_1;
DROP INDEX
postgres=# CREATE FULLTEXT INDEX test_index_1 ON test (title, boby, name) WITH
PARSER ngram;
CREATE INDEX
postgres=# \d test_index_1
                 Index "fulltext_test.test_index_1"
    Column     | Type |                    Definition
---------------+------+-----------------------------------------------
 to_tsvector   | text | to_tsvector('ngram'::regconfig, title::text)
 to_tsvector1  | text | to_tsvector('ngram'::regconfig, boby)
 to_tsvector2  | text | to_tsvector('ngram'::regconfig, name::text)
gin, for table "fulltext_test.test"
postgres=# SELECT * FROM test WHERE MATCH (title, boby, name) AGAINST ('test');
 id |    title     |     boby     |   name
----+--------------+--------------+----------
 1  | test         | &67575@gauss | GBase 8c
 2  | test1        | gauss        | GBase 8c
 3  | test2        | gauss2       | GBase 8c
 4  | test3        | test         | GBase 8c
 5  | gauss_123_@  | test         | GBase 8c
(5 rows)
postgres=# SELECT * FROM test WHERE MATCH (title, boby, name) AGAINST ('gauss');
 id |    title     |     boby     |   name
----+--------------+--------------+----------
 1  | test         | &67575@gauss | GBase 8c
 2  | test1        | gauss        | GBase 8c
 3  | test2        | gauss2       | GBase 8c
 4  | test3        | test         | GBase 8c
 5  | gauss_123_@  | test         | GBase 8c
 6  |              |              | GBase 8c
(6 rows)
postgres=# SELECT * FROM test WHERE MATCH (title, boby, name) AGAINST ('GBase 8c');
 id |    title     |     boby     |   name
----+--------------+--------------+----------
 1  | test         | &67575@gauss | GBase 8c
 2  | test1        | gauss        | GBase 8c
```

```
 3  | test2       | gauss2       | GBase 8c
 4  | test3       | test         | GBase 8c
 5  | gauss_123_@ | test         | GBase 8c
 6  |             |              | GBase 8c
(6 rows)
postgres=# drop table if exists articles;
NOTICE:  table "articles" does not exist, skipping
DROP TABLE
postgres=# CREATE TABLE articles (
postgres(# ID int,
postgres(# title VARCHAR(100),
postgres(# FULLTEXT INDEX ngram_idx(title)WITH PARSER ngram
postgres(# );
CREATE TABLE
postgres=# \d articles
       Table "fulltext_test.articles"
 Column |          Type          | Modifiers
--------+------------------------+-----------
 ID     | integer                |
 title  | character varying(100) |
Indexes:
    "ngram_idx" gin (to_tsvector('ngram'::regconfig, title::text)) TABLESPACE
pg_default
postgres=# drop table if exists articles;
DROP TABLE
postgres=# CREATE TABLE articles (
postgres(# ID int,
postgres(# title VARCHAR(100),
postgres(# FULLTEXT INDEX (title)WITH PARSER ngram
postgres(# );
CREATE TABLE
postgres=# \d articles
       Table "fulltext_test.articles"
 Column |          Type          | Modifiers
--------+------------------------+-----------
 ID     | integer                |
 title  | character varying(100) |
Indexes:
    "articles_to_tsvector_idx" gin (to_tsvector('ngram'::regconfig,
title::text)) TABLESPACE pg_default
postgres=# drop table if exists articles;
```

```
DROP TABLE
postgres=# CREATE TABLE articles (
postgres(# ID int,
postgres(# title VARCHAR(100),
postgres(# FULLTEXT KEY keyngram_idx(title)WITH PARSER ngram
postgres(# );
CREATE TABLE
postgres=# \d articles
       Table "fulltext_test.articles"
 Column |          Type          | Modifiers
--------+------------------------+-----------
 ID     | integer                |
 title  | character varying(100) |
Indexes:
    "keyngram_idx" gin (to_tsvector('ngram'::regconfig, title::text))
TABLESPACE pg_default
postgres=# drop table if exists articles;
DROP TABLE
postgres=# CREATE TABLE articles (
postgres(# ID int,
postgres(# title VARCHAR(100),
postgres(# FULLTEXT KEY (title)WITH PARSER ngram
postgres(# );
CREATE TABLE
postgres=# \d articles
       Table "fulltext_test.articles"
 Column |          Type          | Modifiers
--------+------------------------+-----------
 ID     | integer                |
 title  | character varying(100) |
Indexes:
    "articles_to_tsvector_idx" gin (to_tsvector('ngram'::regconfig,
title::text)) TABLESPACE pg_default
postgres=# create table table_ddl_0154(col1 int,col2 varchar(64), FULLTEXT
idx_ddl_0154(col2));
CREATE TABLE
postgres=# create table table_ddl_0085(
postgres(# id int(11) not null,
postgres(# username varchar(50) default null,
postgres(# sex varchar(5) default null,
postgres(# address varchar(100) default null,
```

```
postgres(# score_num int(11));
CREATE TABLE
postgres=# create fulltext index idx_ddl_0085_02 on table_ddl_0085(username);
CREATE INDEX
postgres=# insert into table_ddl_0085 values (1,'test','m','xi''an changanqu',
10001), (2,'tst','w','xi''an beilingqu', 10002),
(3,'es','w','xi''an yangtaqu', 10003),(4,'s','m','beijingchaoyangqu', 10004);
INSERT 0 4
postgres=# SELECT * FROM table_ddl_0085 WHERE MATCH (username) AGAINST ('te' IN
NATURAL LANGUAGE MODE);
 id | username | sex |     address      | score_num
----+----------+-----+------------------+-----------
  1 | test     | m   | xi'an changanqu |     10001
(1 row)
postgres=# SELECT * FROM table_ddl_0085 WHERE MATCH (username) AGAINST ('ts' IN
NATURAL LANGUAGE MODE WITH QUERY EXPANSION);
 id | username | sex |     address      | score_num
----+----------+-----+------------------+-----------
  2 | tst      | w   | xi'an beilingqu |     10002
(1 row)
postgres=# SELECT * FROM table_ddl_0085 WHERE MATCH (username) AGAINST ('test'
IN BOOLEAN MODE);
 id | username | sex |     address      | score_num
----+----------+-----+------------------+-----------
  1 | test     | m   | xi'an changanqu |     10001
(1 row)
postgres=# SELECT * FROM table_ddl_0085 WHERE MATCH (username) AGAINST ('es' WITH
QUERY EXPANSION);
 id | username | sex |     address      | score_num
----+----------+-----+------------------+-----------
  1 | test     | m   | xi'an changanqu |     10001
  3 | es       | w   | xi'an yangtaqu  |     10003
(2 rows)
postgres=# SELECT * FROM table_ddl_0085 WHERE MATCH (username) AGAINST ('s');
 id | username | sex |      address       | score_num
----+----------+-----+--------------------+-----------
  4 | s        | m   | beijingchaoyangqu |     10004
(1 row)
postgres=# insert into table_ddl_0085 select * from table_ddl_0085 where match
(username) against ('te' IN NATURAL LANGUAGE MODE);
INSERT 0 1
```

```
postgres=# select * from table_ddl_0085;
 id | username | sex |      address      | score_num
----+----------+-----+-------------------+-----------
  1 | test     | m   | xi'an changanqu   |     10001
  2 | tst      | w   | xi'an beilingqu   |     10002
  3 | es       | w   | xi'an yangtaqu    |     10003
  4 | s        | m   | beijingchaoyangqu |     10004
  1 | test     | m   | xi'an changanqu   |     10001
(5 rows)
postgres=# create fulltext index idx_ddl_0085_03 on table_ddl_0085(username)
with parser ngram visible;
CREATE INDEX
postgres=# create fulltext index idx_ddl_0085_04 on table_ddl_0085(username)
visible with parser ngram;
CREATE INDEX
postgres=# create fulltext index idx_ddl_0085_05 on table_ddl_0085(username)
visible;
CREATE INDEX
postgres=# create fulltext index idx_ddl_0085_06 on table_ddl_0085(username)
with parser ngram cgbaseent 'TEST FULLTEXT INDEX COMMENT';
CREATE INDEX
postgres=# create fulltext index idx_ddl_0085_07 on table_ddl_0085(username)
cgbaseent 'TEST FULLTEXT INDEX COMMENT' with parser ngram;
CREATE INDEX
postgres=# create fulltext index idx_ddl_0085_08 on table_ddl_0085(username)
cgbaseent 'TEST FULLTEXT INDEX COMMENT';
CREATE INDEX
postgres=# drop schema fulltext_test cascade;
NOTICE:  drop cascades to 4 other objects
DETAIL:  drop cascades to table test
drop cascades to table articles
drop cascades to table table_ddl_0154
drop cascades to table table_ddl_0085
DROP SCHEMA
postgres=# reset current_schema;
RESET
```

相关链接

SELECT

2.4.1.5.42 SET-CHARSET

功能描述

设置客户端的字符编码类型。

注意事项

GBase 8c 中该语句等价于 set client_encoding。

请根据前端业务的情况确定，客户端编码和服务器端编码尽量保持一致，提高效率。

兼容 PostgreSQL 所有的字符编码类型。

语法格式

```
SET {CHARACTER SET | CHARSET} {'charset_name' | DEFAULT}
```

参数说明

{CHARACTER SET | CHARSET}

两者是等价的。

{'charset_name' | DEFAULT}

charset_name 支持 GBase 8c 可设置的字符编码类型，如 utf8、gbk 等；指定 DEFAULT 时会将字符集重置为默认的字符集。

charset_name 支持以下形式：

1. utf8

2. 'utf8'

3. "utf8"

示例

```
postgres=# show client_encoding;
-[ RECORD 1 ]---+----
client_encoding | GBK
postgres=# set charset gbk;
SET
db_show=# show client_encoding;
-[ RECORD 1 ]---+----
client_encoding | GBK
postgres=# set charset default;
SET
```

```
postgres=# show client_encoding;
-[ RECORD 1 ]---+-----
client_encoding | UTF8
postgres=# set character set 'gbk';
SET
postgres=# show client_encoding;
-[ RECORD 1 ]---+----
client_encoding | GBK
postgres=# set character set default;
SET
postgres=# show client_encoding;
-[ RECORD 1 ]---+-----
client_encoding | UTF8
```

2.4.1.5.43 SET-PASSWORD

功能描述

修改用户密码。

注意事项

不指定用户则修改当前连接用户密码。

初始用户可以修改任何用户的密码（包括自身密码），不需要指定 REPLACE 校验当前密码。

非初始用户不能修改初始用户的密码。

sysadmin 和拥有 createrole 权限的用户可以修改其他（非初始化、非 sysadmin，非 createrole 权限）用户密码，不需要指定 REPLACE 校验当前密码。

sysadmin 和拥有 createrole 权限的用户修改自身密码时，需要指定 REPLACE 校验当前密码。

语法格式

```
SET PASSWORD [FOR user] = password_option [REPLACE 'current_auth_string']
password_option: {
    'auth_string'
  | PASSWORD('auth_string')
}
```

参数说明

[FOR user]

user 支持以下形式：

user（不区分大小写）。

'user'（区分大小写）。

“user”（区分大小写）。

'user'@'host'（区分大小写）。

current_user()/current_user。

auth_string

需要设置的密码。

current_auth_string

当前密码。

示例

```
--修改指定用户密码
postgres=# create user user1 with password 'abcd@123';
CREATE ROLE
postgres=# set password for user1 = 'abcd@124';
ALTER ROLE
--修改当前用户密码
postgres=# set password = 'abcd@123';
ALTER ROLE
postgres=# set password for current_user = 'abcd@123';
ALTER ROLE
postgres=# set password for current_user() = 'abcd@123';
ALTER ROLE
```

2.4.1.5.44 SHOW-CHARSET

注意事项

N/A

功能描述

显示所有支持的字符集。

语法格式

```
SHOW {CHARACTER SET | CHARSET} [LIKE 'pattern' | WHERE expr]
```

参数说明

WHERE expr

筛选表达式。

LIKE 'pattern'

pattern 正则表达式匹配字符集的名称。

返回结果集

字段	说明	备注
charset	字符集名称	
Description	字符集的描述	
default collation	字符集的默认排序规则	该字段内容为空
maxlen	存储一个字符所需的最大字节数	

示例

```
postgres=# SHOW CHARACTER SET LIKE 'a%';
 charset |    Description    | default collation | maxlen
---------+-------------------+-------------------+--------
 abc     | alias for WIN1258 |                   |      1
 alt     | alias for WIN866  |                   |      1
(2 rows)
```

2.4.1.5.45 SHOW-CHARACTER-SET

注意事项

N/A

功能描述

显示所有支持的字符集。

语法格式

```
SHOW {CHARACTER SET | CHARSET} [LIKE 'pattern' | WHERE expr]
```

参数说明

WHERE expr

筛选表达式。

LIKE 'pattern'

pattern 正则表达式匹配字符集的名称。

返回结果集

字段	说明	备注
charset	字符集名称	
Description	字符集的描述	
default collation	字符集的默认排序规则	该字段内容为空
maxlen	存储一个字符所需的最大字节数	

示例

```
postgres=# SHOW CHARACTER SET LIKE 'a%';
 charset |    Description    | default collation | maxlen
---------+-------------------+-------------------+--------
 abc     | alias for WIN1258 |                   |      1
 alt     | alias for WIN866  |                   |      1
(2 rows)
```

2.4.1.5.46 SHOW-COLLATION

注意事项

N/A

功能描述

显示所有支持的服务器的字符序。

语法格式

```
SHOW COLLATION [LIKE 'pattern' | WHERE expr]
```

参数说明

WHERE expr

筛选表达式。

LIKE 'pattern'

pattern 正则表达式匹配排序的名称。

返回结果集

字段	说明	备注
collation	排序集名字	
charset	排序集关联的字符集	
id	字符集的描述	该字段对应 pg_collation 表中的行对应行的 OID
default	是否是字符集对应的排序集	GBase 8c 无默认排序，此字段内容为空
compiled	排序集是否已编译	该字段内容为 Yes
sortlen	排序字符集时需要的内存大小	该字段内容为空

示例

```
postgres=# SHOW COLLATION LIKE 'aa%';
    collation     | charset |  id   | default | compiled | sortlen
```

```
-----------------+--------+-------+--------+---------+----------
 aa_DJ           | utf8   | 13450 |        | Yes     |
 aa_DJ           | latin1 | 13451 |        | Yes     |
 aa_DJ.iso88591  | latin1 | 13452 |        | Yes     |
 aa_DJ.utf8      | utf8   | 13453 |        | Yes     |
 aa_ER           | utf8   | 13454 |        | Yes     |
 aa_ER.utf8      | utf8   | 13455 |        | Yes     |
 aa_ER.utf8@saaho | utf8  | 13456 |        | Yes     |
 aa_ER@saaho     | utf8   | 13457 |        | Yes     |
 aa_ET           | utf8   | 13458 |        | Yes     |
 aa_ET.utf8      | utf8   | 13459 |        | Yes     |
(10 rows)
```

2.4.1.5.47 SHOW_COLUMNS

功能描述

查看指定表的列元信息。

注意事项

临时表需要指定临时表对应的 schema 查询。

复合主键索引所有参与列都会在 Key 字段中显示为 PRI。

复合唯一索引所有参与列都会在 Key 字段中显示为 UNI。

如果一个列参与了多个索引的创建，以该列第一个创建的索引为准显示 Key 字段。

生成列会在 Default 中显示生成式。

表名中含 schemaname/dbname 并且同时指定 dbname 时，仅匹配指定的 dbname。

结果仅显示当前查询用户具有 SELECT 权限的列信息。

语法格式

```
SHOW [FULL] {COLUMNS | FIELDS}
   {FROM | IN} tbl_name
   [{FROM | IN} db_name]
   [LIKE 'pattern' | WHERE expr]
```

参数说明

{COLUMNS | FIELDS}

使用 COLUMNS 和 FIELDS 效果是等价的。

tbl_name

表名，可指定表名。也可以指定 schema_name.table_name。

db_name

库名（或 schema），当 tbl_name 中也指定库名（或 schema 名）时，优先选择本选项。

LIKE 'pattern'

pattern 匹配显示结果的 Field 列。

示例

```
--创建简单表
postgres=# CREATE SCHEMA tst_schema1;
postgres=# SET SEARCH_PATH TO tst_schema1;
postgres=# CREATE TABLE tst_t1
postgres-# (
postgres(# id int primary key,
postgres(# name varchar(20) NOT NULL,
postgres(# addr text COLLATE "de_DE",
postgres(# phone text COLLATE "es_ES",
postgres(# addr_code text
postgres(# );
postgres=# COMMENT ON COLUMN tst_t1.id IS 'identity';
--查看表的列元信息
postgres=# SHOW COLUMNS FROM tst_t1;
   Field   |         Type          | Null | Key | Default | Extra
-----------+-----------------------+------+-----+---------+-------
 id        | integer               | NO   | PRI | NULL    |
 name      | character varying(20) | NO   |     | NULL    |
 addr      | text                  | YES  |     | NULL    |
 phone     | text                  | YES  |     | NULL    |
 addr_code | text                  | YES  |     | NULL    |
postgres=# show FULL COLUMNS FROM tst_t1;
   Field   |         Type          | Collation | Null | Key | Default | Extra |
Privileges                | comment
-----------+-----------------------+-----------+------+-----+---------+------
-+-----------------------------------+---------
 id        | integer               | NULL      | NO   | PRI | NULL    |       |
UPDATE,SELECT,REFERENCES,INSERT,COMMENT | identity
 name      | character varying(20) | NULL      | NO   |     | NULL    |       |
UPDATE,SELECT,REFERENCES,INSERT,COMMENT |
```

```
 addr       | text                   | de_DE      | YES  |      | NULL    |        |
UPDATE,SELECT,REFERENCES,INSERT,COMMENT |
 phone      | text                   | es_ES      | YES  |      | NULL    |        |
UPDATE,SELECT,REFERENCES,INSERT,COMMENT |
 addr_code | text                    | NULL       | YES  |      | NULL    |        |
UPDATE,SELECT,REFERENCES,INSERT,COMMENT |

 postgres=# show FULL COLUMNS FROM tst_schema1.tst_t1;
   Field    |          Type          | Collation | Null | Key | Default | Extra |
Privileges                | comment
------------+------------------------+-----------+------+-----+---------+------
-+---------------------------------------+---------
 id         | integer                | NULL      | NO   | PRI | NULL    |       |
UPDATE,SELECT,REFERENCES,INSERT,COMMENT | identity
 name       | character varying(20)  | NULL      | NO   |     | NULL    |       |
UPDATE,SELECT,REFERENCES,INSERT,COMMENT |
 addr       | text                   | de_DE      | YES  |      | NULL    |        |
UPDATE,SELECT,REFERENCES,INSERT,COMMENT |
 phone      | text                   | es_ES      | YES  |      | NULL    |        |
UPDATE,SELECT,REFERENCES,INSERT,COMMENT |
 addr_code | text                    | NULL       | YES  |      | NULL    |        |
UPDATE,SELECT,REFERENCES,INSERT,COMMENT |

--模糊匹配、过滤
postgres=# show full columns from tst_t1 like '%addr%';
   Field    | Type | Collation | Null | Key | Default | Extra |
Privileges                | comment
------------+------+-----------+------+-----+---------+-------+--------------
-------------------------+---------
 addr       | text | de_DE     | YES  |     | NULL    |       |
UPDATE,SELECT,REFERENCES,INSERT,COMMENT |
 addr_code | text | NULL       | YES  |     | NULL    |       |
UPDATE,SELECT,REFERENCES,INSERT,COMMENT |

postgres=# show full columns from tst_t1 where Type='text';
   Field    | Type | Collation | Null | Key | Default | Extra |
Privileges                | comment
------------+------+-----------+------+-----+---------+-------+--------------
-------------------------+---------
 addr       | text | de_DE     | YES  |     | NULL    |       |
UPDATE,SELECT,REFERENCES,INSERT,COMMENT |
```

```
 phone      | text | es_ES      | YES  |      | NULL    |        |
UPDATE,SELECT,REFERENCES,INSERT,COMMENT |
 addr_code | text | NULL       | YES  |      | NULL    |        |
UPDATE,SELECT,REFERENCES,INSERT,COMMENT |
--显示权限过滤
postgres=# CREATE USER tst_u1 PASSWORD 'tst_u1@123';
postgres=# SET ROLE tst_u1 PASSWORD 'tst_u1@123';
postgres=#> SET SEARCH_PATH TO tst_schema1;
postgres=#> show full columns from tst_t1;
 Field | Type | Collation | Null | Key | Default | Extra | Privileges | comment
-------+------+-----------+------+-----+---------+-------+------------+------
---
(0 rows)
postgres=# RESET ROLE;
postgres=# GRANT SELECT (addr, phone) on tst_t1 to tst_u1;
postgres=# SET ROLE tst_u1 PASSWORD 'tst_u1@123';
postgres=#> SET SEARCH_PATH TO tst_schema1;
postgres=#> show full columns from tst_t1;
  Field   | Type | Collation | Null | Key | Default | Extra | Privileges   | comment
----------+------+-----------+------+-----+---------+-------+--------------
---+---------
 addr      | text | de_DE     | YES  |     | NULL    |       | SELECT       |
 phone     | text | es_ES     | YES  |     | NULL    |       | SELECT       |
```

2.4.1.5.48 SHOW-CREATE-DATABASE

功能描述

显示 CREATE DATABASE 创建命名数据库的语句。

如果该 SHOW 语句包含一个 IF NOT EXISTS 子句，则输出也包含这样的子句。SHOW CREATE SCHEMA 是的同义词 SHOW CREATE DATABASE。

注意事项

在 b 数据库中 database 和 schema 是等价的，所以在 GBase 8c 中进行语句拼装时都按 create schema 拼装。

在 GBase 8c 中,创建 schema 支持 with blockchain 子句，所以在拼装时也会根据 schema 在系统表中的信息判断是否拼接该子句。

语法格式

```
SHOW CREATE {DATABASE | SCHEMA} [IF NOT EXISTS] db_name
```

参数说明

db_name

目标实例名。

示例

```
--查询数据库创建语句
postgres=# show create database test_get_database;
     Database      |                  Create Database
-------------------+-----------------------------------------------------
 test_get_database | CREATE SCHEMA test_get_database AUTHORIZATION gbase
(1 row)
```

2.4.1.5.49 SHOW-CREATE-FUNCTION

功能描述

它返回可用于重新创建命名函数的确切字符串。 类似的语句 SHOW CREATE PROCEDURE 显示有关存储函数的信息。 要使用任一语句，您必须具有全局 SELECT 特权。

注意事项

sql_mode 是查询时的会话值,b 数据库在这里展示的是创建例程时绑定的 sql_mode,GBase 8c 这里展示的是会话的值,因为 GBase 8c 在创建例程时不会将例程与 sql_mode 绑定。

character_set_client 是 client_encoding 创建例程时系统变量 的会话值 。

collation_connection 是 lc_collate 创建数据库时指定的值。

Database Collation 是 lc_collate 创建数据库时指定的值。

语法格式

```
SHOW CREATE FUNCTION func_name
```

参数说明

func_name

函数名。

示例

```
--创建函数
postgres=# CREATE FUNCTION functest_A_1(text, date) RETURNS bool LANGUAGE 'sql'
       AS 'SELECT $1 = ''abcd'' AND $2 > ''2001-01-01''';
CREATE FUNCTION
--查询函数创建语句
postgres=# show create function functest_A_1;
   Function   |                          Create Function
|              sql_mode               | character_set_client |
collation_connection
| Database Collation
-------------+-------------------------------------------------------------------
----+-----------------------------------+----------------------+----------
----------+--------------------
 functest_a_1 | CREATE OR REPLACE FUNCTION public.functest_a_1(text, date)
+| sql_mode_strict,sql_mode_full_group | UTF8                 | en_US.UTF-8
| en_US.UTF-8
              |  RETURNS boolean
+|                                     |                      |
|
              |  LANGUAGE sql
+|                                     |                      |
|
              |  NOT FENCED NOT SHIPPABLE
+|                                     |                      |
|
              | AS $function$SELECT $1 = 'abcd' AND $2 >
'2001-01-01'$function$;+|                                    |
|
|
              |
|                                     |                      |
|
(1 row)
```

2.4.1.5.50 SHOW-CREATE-PROCEDURE

功能描述

它返回可用于重新创建命名存储过程的确切字符串。 类似的语句 SHOW CREATE FUNCTION 显示有关存储函数的信息。 要使用任一语句，您必须具有全局 SELECT 特权。

注意事项

sql_mode 是查询时的会话值,b 数据库在这里展示的是创建例程时绑定的 sql_mode,GBase 8c 这里展示的是会话的值，因为 GBase 8c 在创建例程时不会将例程与 sql_mode 绑定。

character_set_client 是 client_encoding 创建例程时系统变量 的会话值 。

collation_connection 是 lc_collate 创建数据库时指定的值。

Database Collation 是 lc_collate 创建数据库时指定的值。

语法格式

```
SHOW CREATE PROCEDURE proc_name
```

参数说明

proc_name

存储过程名。

示例

```
--创建存储过程
postgres=# create procedure test_procedure_test(int,int)
postgres-# SHIPPABLE IMMUTABLE
postgres-# as
postgres$# begin
postgres$# select $1 + $2;
postgres$# end;
postgres$# /
CREATE PROCEDURE
--查询存储过程创建语句
postgres=# show create procedure test_procedure_test;
      Procedure      |                        Create Procedure
|              sql_mode               | character_set_client |
collation_connection | Database Collation
--------------------+-------------------------------------------------------
---------+-------------------------------------+--------------------+-----
----------------+------------------
 test_procedure_test | CREATE OR REPLACE PROCEDURE
public.test_procedure_test(int,int)+| sql_mode_strict,sql_mode_full_group |
UTF8                | en_US.UTF-8
      | en_US.UTF-8
                    |  IMMUTABLE SHIPPABLE
+|                                     |                      |
```

```
      |
                      | AS  DECLARE
+|                                  |                      |
      |
                      | begin
+|                                  |                      |
      |
                      | select $1 + $2;
+|                                  |                      |
      |
                      | end;
+|                                  |                      |
      |
                      | /
+|                                  |                      |
      |
                      |
|                                  |                      |
      |
(1 row)
```

2.4.1.5.51 SHOW-CREATE-TABLE

功能描述

显示创建表 tbl_name

显示 CREATE TABLE 创建命名表的语句。此语法也可用于查询视图(view)的创建语句。

注意事项

此语法不支持查询临时表。

字段字符集和字符序将从表上继承，字段和表的字符集、字符序在 SHOW CREATE TABLE 中均会全部显示。若表的默认字符集或字符序不存在，当 b_format_behavior_compat_options = 'default_collation'时，字符集和字符序将继承当前数据库的字符集及其对应的默认字符序，未设置 b_format_behavior_compat_options 时，无字符集和字符序显示。

语法格式

```
show create table tbl_name;
```

参数说明

tbl_name

表名。

示例

```
postgres=# CREATE TABLE t1 (c1 INT PRIMARY KEY);
NOTICE:  CREATE TABLE / PRIMARY KEY will create implicit index "t1_pkey" for table
"t1"
CREATE TABLE
postgres=# show create table t1;
 Table |                      Create Table
-------+-----------------------------------------------------
 t1    | SET search_path = public;                           +
       | CREATE TABLE t1 (                                   +
       |     c1 integer NOT NULL                             +
       | )                                                   +
       | WITH (orientation=row, compression=no);             +
       | ALTER TABLE t1 ADD CONSTRAINT t1_pkey PRIMARY KEY (c1);
(1 row)
```

2.4.1.5.52 SHOW-CREATE-TRIGGER

功能描述

它返回可用于重新创建命名触发器的确切字符串。

注意事项

sql_mode 是查询时的会话值,b 数据库在这里展示的是创建例程时绑定的 sql_mode，GBase 8c 这里展示的是会话的值，因为 GBase 8c 在创建例程时不会将例程与 sql_mode 绑定。

character_set_client 是 client_encoding 创建例程时系统变量 的会话值 。

collation_connection 是 lc_collate 创建数据库时指定的值。

Database Collation 是 lc_collate 创建数据库时指定的值。

语法格式

```
SHOW CREATE TRIGGER trigger_name
```

参数说明

trigger_name

触发器名。

示例

```
--查询触发器创建语句
postgres=# show create trigger before_ins_stmt_trig;
       Trigger        |              sql_mode               |
SQL Original Statement
                              | character_set_client | collation_connection |
Database Collation
--------------------+-------------------------------------+---------------
-------------------------------------------------------------------------
-----------------------------------+--------------------+-----------
---------+--------------------
 before_ins_stmt_trig | sql_mode_strict,sql_mode_full_group | CREATE TRIGGER
before_ins_stmt_trig BEFORE INSERT ON main_table FOR EACH STATEMENT EXECUTE
PROCEDURE trigger_func('before_ins_stmt') | UTF8                   | en_US.UTF-8
| en_US.UTF-8
(1 row)
```

2.4.1.5.53 SHOW-CREATE-VIEW

功能描述

它返回可用于重新创建命名视图的确切字符串。

注意事项

character_set_client 是 client_encoding 创建例程时系统变量的会话值。

collation_connection 是 lc_collate 创建数据库时指定的值。

语法格式

```
SHOW CREATE VIEW view_name
```

参数说明

view_name

视图名。

示例

```
--创建视图
postgres=# create view tt19v as
postgres-# select 'foo'::text = any(array['abc','def','foo']::text[]) c1,
postgres-#        'foo'::text = any((select
array['abc','def','foo']::text[])::text[]) c2;
```

```
CREATE VIEW
--查询视图创建语句
postgres=# show create view tt19v;
 View  |
Create View
                          | character_set_client | collation_connection
------+------------------------------------------------------------------------
------------------------------------------------------------------------------
-----------------------------------+--------------------+--------------
------
 tt19v | CREATE OR REPLACE VIEW public.tt19v AS
                          +| UTF8                 | en_US.UTF-8
       | SELECT ('foo'::text = ANY (ARRAY['abc'::text, 'def'::text,
'foo'::text])) AS c1, ('foo'::text = ANY ((SELECT ARRAY['abc'::text, 'def'::text,
'foo'::text] AS "array")::text[])) AS c2; |                      |
(1 row)
```

2.4.1.5.54 SHOW-DATABASES

功能描述

列出所有或按条件查询相关 schema。

注意事项

b 数据库的 show databases 是查查询据库操作，GBase 8c 的 show databases 是查询 schema 操作。

schema 会按名称顺序展示。

语法格式

```
SHOW {DATABASES | SCHEMAS} [LIKE 'pattern' | WHERE expr]
```

参数说明

{DATABASES | SCHEMAS}

两者是等价的。

[LIKE 'pattern' | WHERE expr]

pattern 支持 like 语法，可以是 schema_name 的全称或者一部分，用于模糊查询；expr 支持任意表达式，通常的用法是：show database where database = 'name'

示例

```
--查看当前数据库下所有 schema
postgres=# create schema a1;
CREATE SCHEMA
postgres=# show databases;
      Database
--------------------
 a1
 blockchain
 cstore
 db4ai
 dbe_perf
 dbe_pldebugger
 dbe_pldeveloper
 information_schema
 pg_catalog
 pg_toast
 pkg_service
 public
 snapshot
 sqladvisor
(14 rows)
--按条件查询 schema
postgres=# create schema abb1;
CREATE SCHEMA
postgres=# create schema abb2;
CREATE SCHEMA
postgres=# create schema abb3;
CREATE SCHEMA
postgres=# show databases like '%bb%';
 Database
----------
 abb1
 abb2
 abb3
(3 rows)
postgres=# show databases like 'a%';
 Database
----------
 a1
 abb1
 abb2
```

```
 abb3
(4 rows)
postgres=# show schemas where database = 'a1';
 Database
----------
 a1
(1 row)
```

2.4.1.5.55 SHOW-FUNCTION-STATUS

注意事项

N/A

功能描述

显示有关存储函数的信息。

语法格式

```
SHOW FUNCTION STATUS [LIKE 'pattern' | WHERE expr]
```

参数说明

WHERE expr

筛选表达式。

LIKE 'pattern'

pattern 正则表达式匹配函数名。

返回结果集

字段	说明	备注
Db	schema 名字	按照 schema 展示
Name	函数名称	
TYPE	类型	FUNCTION/PROCEDURE

字段	说明	备注
Deinfer	用户	
Modified	修改时间	
Created	创建时间	
Security_type	安全类型	
comment	注释	
character_set_client	创建时客户端的字符集	显示为空
collation_connection	创建时客户端的排序规则	显示为空
Database Collation	数据库的排序集	

示例

```
-- 创建函数
postgres=# CREATE FUNCTION func_add_sql(integer, integer) RETURNS integer AS
'select $1 + $2;' LANGUAGE SQL IMMUTABLE RETURNS NULL ON NULL INPUT;
CREATE FUNCTION
postgres=# show function status like 'func_add_s%';
   Db    |     Name     |   Type    | Definer |           Modified            |
Created            | Security_type | comment | character_set_client |
collation_connection | Database Collation
--------+--------------+-----------+---------+-------------------------------+
-------------------------+--------------+---------+----------------------
---+----------------------+------------------
```

```
 public | func_add_sql | FUNCTION | wyc      | 2022-09-24 14:42:29.427382+08 |
2022-09-24 14:42:29.427382+08 | INVOKER       |          |
|                       | en_US.UTF-8
(1 row)
```

2.4.1.5.56 SHOW-GRANTS

功能描述

显示 GBase 8c 中用户的权限信息。

注意事项

若不指定用户，则显示当前用户的权限信息。

语法格式

```
""SHOW GRANTS [FOR user]
```

参数说明

user

用户名。如不指定，则显示当前执行语句的用户的权限信息

参考示例

```
mysql=# show grants for test;
                                          Grants
-------------------------------------------------------------------------------
--------------
 GRANT INSERT, SELECT, UPDATE, DELETE, TRUNCATE, REFERENCES, TRIGGER ON TABLE test
TO test
 GRANT SELECT ON TABLE test TO test
 ALTER ROLE test WITH LOGIN
(3 rows)
```

2.4.1.5.57 SHOW-INDEX

功能描述

查看表的索引信息。

注意事项

若不指定 schema_name，查询的是当前 schema 下的表。

若指定的表是 schema_name.table_name 格式，且显示指定了 schema_name，则实际上取后者的 schema。

语法格式

```
SHOW { INDEX | INDEXES | KEYS }
   { FROM | IN } table_name
   [{FROM | IN} schema_name ]
   [ WHERE expr ]
```

参数说明

table_name

表名，可指定表名，也可以指定 schema_name.table_name

schema_name

schema 名，可选项，若不指定，则查询的是当前 schema

输出字段说明

字段	含义
Table	索引所属表名
Non_unique	是否是非唯一索引
Key_name	索引名
Seq_in_index	索引列在索引中的序号
Column_name	索引列的列名
Collation	取值有 A（默认，升序），D（降序）、NULL（索引不支持排序）
Cardinality	根据 pg_statistic.stadistinct 和 pg_class.reltuples 计算得到：stadistinct > 0: stadistinct

字段	含义
	stadistinct = 0: NULL stadistinct < 0: reltuples * stadistinct * -1
Sub_part	索引前缀。如果该列仅被部分索引，则是索引字符的数量；如果整个列都被索引，则是 NULL。当前不支持前缀索引，NULL
Packed	如何打包 key 值，create table 时指定 pack_keys；否则返回 NULL。当前不支持，为 NULL
Null	可能包含 NULL 值则是 YES，否则为"
Index_type	使用的索引方法：BTREE、HASH 等
comment	pg_index 表中记录的 indisusable 为 true 则显示 disabled，false 则显示"
Index_cgbaseent	创建索引时 COMMENT 指定的注释信息

示例

```
--创建表和索引
postgres=# CREATE SCHEMA tst_schema;
postgres=# SET SEARCH_PATH TO tst_schema;
postgres=# CREATE TABLE tst_t1
postgres-# (
postgres(# id int primary key,
postgres(# name varchar(20) NOT NULL
postgres(# );
postgres=# CREATE INDEX tst_t1_name_ind on tst_t1(name);
--查看表的索引
postgres=# show index from tst_t1 ;
```

```
 table   | non_unique |    key_name     | seq_in_index | column_name | collation
| cardinality | sub_part | packed | null | index_type | cgbaseent | index_cgbaseent
--------+------------+-----------------+--------------+-------------+-----------
---+-------------+----------+--------+------+------------+---------+---------
------
 tst_t1 | t          | tst_t1_name_ind |            1 | name        | A
|             |          |        |      | btree      |         |
 tst_t1 | f          | tst_t1_pkey     |            1 | id          | A
|             |          |        |      | btree      |         |
(2 rows)
```

相关链接

N/A

2.4.1.5.58 SHOW-MASTER-STATUS

功能描述

查看当前 wal（xlog）日志的相关进度。

注意事项

该语句在非主库也可以执行。

在主库执行时，Xlog_Lsn 和 pg_current_xlog_location 的结果一致；在非主库执行时，Xlog_Lsn 和 pg_last_xlog_replay_location 的结果一致。

主库用该语句查询当前 xlog 写入的实时进度。

备库用该语句查询当前 xlog 回放的实时进度。

语法格式

```
SHOW MASTER STATUS
```

参数说明

Xlog_File_Name

当前处理的 xlog 文件名。

Xlog_File_Offset

当前处理的 xlog 的文件偏移位置。

Xlog_Lsn

当前 xlog 的 LSN。

示例

```
postgres=# show master status;
      Xlog_File_Name       | Xlog_File_Offset |  Xlog_Lsn
---------------------------+------------------+------------
 000000010000000000000010 |          7142672 | 0/106CFD10
(1 row)
```

2.4.1.5.59 SHOW_PLUGINS

功能描述

查看当前数据库中插件清单。

注意事项

N/A

语法格式

SHOW PLUGINS

参数说明

N/A

示例

```
--查看插件清单
postgres=# SHOW PLUGINS;
      Name        |  Status  | Type | Library | License |
comment
----------------+----------+------+---------+---------+--------------------
------------------------------
 roach_api_stub  | DISABLED |      | NULL    |         | roach api stub
 file_fdw        | ACTIVE   |      | NULL    |         | foreign-data wrapper
for flat file access
 security_plugin | ACTIVE   |      | NULL    |         | provides security
functionality
 hdfs_fdw        | ACTIVE   |      | NULL    |         | foreign-data wrapper
for flat file access
 plpgsql         | ACTIVE   |      | NULL    |         | PL/pgSQL procedural
language
 dolphin         | ACTIVE   |      | NULL    |         | sql engine
 dist_fdw        | ACTIVE   |      | NULL    |         | foreign-data wrapper
for distfs access
```

```
 postgres_fdw     | DISABLED |       | NULL     |          | foreign-data wrapper
for remote PostgreSQL servers
 hstore           | ACTIVE   |       | NULL     |          | data type for storing
sets of (key, value) pairs
 log_fdw          | ACTIVE   |       | NULL     |          | Foreign Data Wrapper
for accessing logging data
--插件状态的更新显示
postgres=# drop Extension hstore;
postgres=# SHOW PLUGINS;
      Name        |  Status  | Type | Library | License |
comment
------------------+----------+------+---------+---------+--------------------
-------------------------------
 roach_api_stub   | DISABLED |       | NULL     |          | roach api stub
 file_fdw         | ACTIVE   |       | NULL     |          | foreign-data wrapper
for flat file access
 security_plugin  | ACTIVE   |       | NULL     |          | provides security
functionality
 hdfs_fdw         | ACTIVE   |       | NULL     |          | foreign-data wrapper
for flat file access
 plpgsql          | ACTIVE   |       | NULL     |          | PL/pgSQL procedural
language
 dolphin          | ACTIVE   |       | NULL     |          | sql engine
 dist_fdw         | ACTIVE   |       | NULL     |          | foreign-data wrapper
for distfs access
 postgres_fdw     | DISABLED |       | NULL     |          | foreign-data wrapper
for remote PostgreSQL servers
 hstore           | DISABLED |       | NULL     |          | data type for storing
sets of (key, value) pairs
 log_fdw          | ACTIVE   |       | NULL     |          | Foreign Data Wrapper
for accessing logging data
postgres=# CREATE Extension hstore;
postgres=# show plugins;
      Name        |  Status  | Type | Library | License |
comment
------------------+----------+------+---------+---------+--------------------
-------------------------------
 roach_api_stub   | DISABLED |       | NULL     |          | roach api stub
 file_fdw         | ACTIVE   |       | NULL     |          | foreign-data wrapper
for flat file access
```

```
 security_plugin | ACTIVE   |       | NULL    |          | provides security
functionality
 hdfs_fdw        | ACTIVE   |       | NULL    |          | foreign-data wrapper
for flat file access
 plpgsql         | ACTIVE   |       | NULL    |          | PL/pgSQL procedural
language
 dolphin         | ACTIVE   |       | NULL    |          | sql engine
 dist_fdw        | ACTIVE   |       | NULL    |          | foreign-data wrapper
for distfs access
 postgres_fdw    | DISABLED |       | NULL    |          | foreign-data wrapper
for remote PostgreSQL servers
 hstore          | ACTIVE   |       | NULL    |          | data type for storing
sets of (key, value) pairs
 log_fdw         | ACTIVE   |       | NULL    |          | Foreign Data Wrapper
for accessing logging data
```

2.4.1.5.60 SHOW_PRIVILEGES

功能描述

查看当前数据库中的权限信息清单。

注意事项

N/A

语法格式

SHOW PRIVILEGES

参数说明

N/A

示例

```
-- 显示当前数据库中的权限信息清单
postgres=# show privileges;
     Privileges      |
Context                                                |
comment
---------------------+------------------------------------------------------
-----------------------------------------------------+----------------------
-----------------------------------------
```

```
 Alter                | Large object, Sequence, Database, Foreign
Server, Function, Node group, Schema, Tablespace, Type, Directory, Package | To alter
the 'objects'
 Alter any index      | Index
| To alter any index
 Alter any sequence   | Sequence
| To alter any sequence
 Alter any table      | Table
| To alter any table
 Alter any trigger    | Trigger
| To alter any trigger
 Alter any type       | Type
| To alter any type
 comment              | Table
| To cgbaseent on table
 Compute              | Node group
| To compute on node group
 Connect              | Database
| To connect database
 Create               | Database, Schema, Tablespace, Node group
| To create database, schema, tablespace, node group
 Create any function  | Function
| To create any function
 Create any index     | Index
| To create any index
 Create any package   | Package
| To create any package
 Create any sequence  | Sequence
| To create any sequence
 Create any synonym   | Synonym
| To create any synonym
 Create any table     | Table
| To create any table
 Create any trigger   | Trigger
| To create any trigger
 Create any type      | Type
| To create any type
 Delete               | Table
| To delete table
 Delete any table     | Table
| To delete any table
```

```
 Drop any sequence     | Sequence
| To drop any sequence
 Drop any synonym      | Synonym
| To drop any synonym
 Drop any table        | Table
| To drop any table
 Drop any trigger      | Trigger
| To drop any trigger
 Drop any type         | Type
| To drop any type
 Execute               | Function,Procedure,Package
| To execute function,  procedure,Package
 Execute any function  | Function
| To execute any function
 Execute any package   | Package
| To execute any package
 Index                 | Table
| To create index on table
 Insert                | Table
| To insert into table
 Insert any table      | Table
| To insert any table
 References            | Table
| To have references on table
 Select                | Large object,Sequence,Table
| To select on large object,sequence and table
 Select any sequence   | Sequence
| To select any sequence
 Select any table      | Table
| To select on any table
 Temporary             | Database
| To create temporary table in database
 Temp                  | Database
| To create temporary table in database
 Truncate              | Table
| To truncate table
 Update                | Large object,Sequence,Table
| To update large object,Sequence,Table
 Update any table      | Table
| To update any table
```

```
 Usage                | Domain,Foreign data wrapper,Foreign
server,Language,Schema,Sequence,Type                                  | To
use domain,fdw,foreign server,language,schema,sequence and type
 Vacuum               | Table
| To vacuum table
(42 rows)
```

2.4.1.5.61 SHOW-PROCEDURE-STATUS

注意事项

N/A

功能描述

显示有关存储过程的信息。

语法格式

SHOW PROCEDURE STATUS [LIKE 'pattern' | WHERE expr]

参数说明

参考 SHOW FUNCTION STATUS

展示内容

参考 SHOW FUNCTION STATUS

示例

```
postgres=# -- 创建存储过程
postgres=# create or replace procedure proc1() as declare genre_rec record; --
声明记录类型
postgres$# begin
postgres$# for genre_rec in (select e1.ename from public.emp1 e1 join
public.emp1mot e1m on e1.mgr = e1m.mgr)
postgres$# loop
postgres$# raise notice '%', genre_rec."ename"; --打印
postgres$# end loop;
postgres$# end;
postgres$# /
CREATE PROCEDURE
    postgres=# -- 查看信息
postgres=# show procedure status like 'proc%';
```

```
   Db    | Name  |   Type    | Definer |           Modified           |
Created             | Security_type | comment | character_set_client |
collation_connection |Database Collation
--------+-------+-----------+---------+------------------------------+------
------------------------+--------------+---------+----------------------+--
--------------------+--------------------
 public | proc1 | PROCEDURE | wyc     | 2022-09-24 14:46:40.868293+08 | 2022-09-24
14:46:40.868293+08 | INVOKER       |         |                      |
|en_US.UTF-8
(1 row)
```

2.4.1.5.62 SHOW-PROCESSLIST

功能描述

查当前外部连接（或内部线程）相关信息。

注意事项

Id 字段对应 pg_stat_activity 视图中的 sessionid

Info 字段记录的是该连接最后一次执行的 SQL，这和 B 数据库有一些差异（B 数据库显示当前执行中的 SQL），但可以结合 State 字段查看 SQL 是否是在执行中，State 字段为 active 时，对应 Info 字段的 SQL 则在执行中。

语法格式

```
""SHOW [FULL] PROCESSLIST
```

参数说明

FULL

不使用 FULL 选项，Info 字段只展示 SQL 长度不超过 100 的部分。

使用 FULL 选项，Info 字段可完全展示长度不超过 1024 的 SQL 语句，如果 SQL 长度超过 1024，会截断 1024 长度之外的部分。

示例

```
--方式 1
postgres=# show processlist;
       Id        |       Pid       |      QueryId       | UniqueSqlId |   User
| Host |    db    |        Cgbaseand         |
     BackendStart          |          XactStart          |  Time  | State
|                Info
```

```
---------------+---------------+-----------------+-----------+-------
----+------+---------+-----------------------+---
-------------------------+-------------------------------+--------+-------
-+--------------------------------------
 139653370304256 | 139653370304256 |               0 |         0 | GBase 8c
|       | postgres | ApplyLauncher          | 20
22-06-21 16:46:19.656076+08 |                               |        |
|
 139653319255808 | 139653319255808 |               0 |         0 | GBase 8c
|       | postgres | Asp                    | 20
22-06-21 16:46:19.728521+08 |                               |      1 | active |
 139653336483584 | 139653336483584 |               0 |         0 | GBase 8c
|       | postgres | PercentileJob          | 20
22-06-21 16:46:19.728527+08 |                               |      8 | active |
 139653302175488 | 139653302175488 |               0 |         0 | GBase 8c
|       | postgres | statement flush thread | 20
22-06-21 16:46:19.728558+08 |                               | 508507 | idle   |
 139653198239488 | 139653198239488 |               0 |         0 | GBase 8c
|       | postgres | WorkloadMonitor        | 20
22-06-21 16:46:19.750133+08 |                               |        |
|
 139653181298432 | 139653181298432 |               0 |         0 | GBase 8c
|       | postgres | WLMArbiter             | 20
22-06-21 16:46:19.750976+08 |                               |        |
|
 139653215110912 | 139653215110912 |               0 |         0 | GBase 8c
|       | postgres | workload               | 20
22-06-21 16:46:19.754504+08 | 2022-06-21 16:46:19.769585+08 | 508507 | active |
WLM fetch collect info from data nodes
 139653421840128 | 139653421840128 |               0 |         0 | GBase 8c
|       | postgres | JobScheduler           | 20
22-06-27 10:00:54.754007+08 |                               |      0 | active |
 139653044328192 | 139653044328192 | 48976645947655327 |  1772643515 | GBase 8c
| -1    | dolphin  | gsql                   | 20
22-06-27 14:00:53.163338+08 | 2022-06-27 14:01:26.794658+08 |      0 | active |
show processlist;
 139653027546880 | 139653027546880 | 48976645947655326 |  1775585557 | GBase 8c
| -1    | postgres | gsql                   | 20
22-06-27 14:01:03.969962+08 | 2022-06-27 14:01:19.967521+08 |      7 | active |
select pg_sleep(100);
(10 rows)
```

```
--方式 2
postgres=# show full processlist;
       Id        |       Pid       |      QueryId      | UniqueSqlId |   User
| Host |    db    |        Cgbaseand         |
     BackendStart          |          XactStart          |  Time  | State
|                 Info
---------------+---------------+-----------------+------------+-------
----+------+---------+----------------------+---
------------------------+-----------------------------+--------+-------
-+-------------------------------------
 139653370304256 | 139653370304256 |                0 |          0 | GBase 8c
|       | postgres | ApplyLauncher          | 20
22-06-21 16:46:19.656076+08 |                               |        |
|
 139653319255808 | 139653319255808 |                0 |          0 | GBase 8c
|       | postgres | Asp                    | 20
22-06-21 16:46:19.728521+08 |                               |      1 | active |
 139653336483584 | 139653336483584 |                0 |          0 | GBase 8c
|       | postgres | PercentileJob          | 20
22-06-21 16:46:19.728527+08 |                               |      8 | active |
 139653302175488 | 139653302175488 |                0 |          0 | GBase 8c
|       | postgres | statement flush thread | 20
22-06-21 16:46:19.728558+08 |                               | 508507 | idle   |
 139653198239488 | 139653198239488 |                0 |          0 | GBase 8c
|       | postgres | WorkloadMonitor        | 20
22-06-21 16:46:19.750133+08 |                               |        |
|
 139653181298432 | 139653181298432 |                0 |          0 | GBase 8c
|       | postgres | WLMArbiter             | 20
22-06-21 16:46:19.750976+08 |                               |        |
|
 139653215110912 | 139653215110912 |                0 |          0 | GBase 8c
|       | postgres | workload               | 20
22-06-21 16:46:19.754504+08 | 2022-06-21 16:46:19.769585+08 | 508507 | active |
WLM fetch collect info from data nodes
 139653421840128 | 139653421840128 |                0 |          0 | GBase 8c
|       | postgres | JobScheduler           | 20
22-06-27 10:00:54.754007+08 |                               |      0 | active |
 139653044328192 | 139653044328192 | 48976645947655327 |  1772643515 | GBase 8c
| -1    | dolphin  | gsql                   | 20
```

```
22-06-27 14:00:53.163338+08 | 2022-06-27 14:01:26.794658+08 |       0 | active |
show processlist;
 139653027546880 | 139653027546880 | 48976645947655326 |  1775585557 | GBase 8c
| -1    | postgres | gsql                     | 20
22-06-27 14:01:03.969962+08 | 2022-06-27 14:01:19.967521+08 |       7 | active |
select pg_sleep(100);
(10 rows)
```

相关链接

N/A

2.4.1.5.63 SHOW-SLAVE-HOSTS

功能描述

查看 wal（xlog）日志同步状态信息，例如发起端发送日志位置，接收端接收日志位置等。

注意事项

在主库执行有效。

展示结果和 select * from pg_stat_replication 一致。

语法格式

SHOW SLAVE HOSTS

参数说明

pid

线程的 PID。

usesysid

用户系统 ID。

usename

用户名。

application_name

程序名称。

client_addr

客户端地址。

client_port

客户端端口。

backend_start

程序启动时间。

state

日志复制的状态：

追赶状态、

一致的流状态。

sender_sent_location

发送端发送日志位置。

receiver_write_location

接收端 write 日志位置。

receiver_flush_location

接收端 flush 日志位置。

receiver_replay_location

接收端 replay 日志位置。

sync_priority

同步复制的优先级（0 表示异步）。

sync_priority

同步状态：

异步复制、

同步复制、

潜在同步者。

示例

```
postgres=# show slave hosts;
```

```
-[ RECORD 1 ]----------+------------------------------
pid                    | 140395615176448
usesysid               | 10
usename                | GBase 8c
application_name       | WalSender to Standby[walreceiver]
client_addr            | 127.0.0.1
client_hostname        |
client_port            | 43174
backend_start          | 2022-08-23 18:41:12.398717+08
state                  | Streaming
sender_sent_location   | 0/1098BB08
receiver_write_location | 0/1098BB08
receiver_flush_location | 0/1098BB08
receiver_replay_location | 0/1098BB08
sync_priority          | 1
sync_state             | Sync
```

2.4.1.5.64 SHOW_STATUS

功能描述

SHOW STATUS 显示系统当前统计状态所有值的数据。 该语句不需要任何特权。它仅需要连接到服务器的能力。

注意事项

N/A

语法格式

```
SHOW [GLOBAL | SESSION] STATUS
    [LIKE 'pattern' | WHERE expr]
```

示例

```
postgres=# show status;
            Variable_name              |              Value
---------------------------------------+---------------------------------
 apply_counter                         | 0
 apply_total_dur                       | 0
 avgiotim                              | 11
 bg_cgbaseit_counter                     | 14
 bg_resp_avg                           | 2380520
 bg_resp_max                           | 7032370
 bg_resp_min                           | 160
```

bg_resp_total	33327281
bg_rollback_counter	0
blk_read_time	0
blks_hit	475019
blks_read	1296
blk_write_time	0
buffers_alloc	0
buffers_backend	0
buffers_backend_fsync	0
buffers_checkpoint	0
buffers_clean	0
checkpoints_req	4
checkpoints_timed	27
checkpoint_sync_time	6376
checkpoint_write_time	5
ckpt_clog_flush_num	0
ckpt_csnlog_flush_num	0
ckpt_multixact_flush_num	0
ckpt_predicate_flush_num	0
ckpt_redo_point	0/28DDB58
ckpt_twophase_flush_num	0
cgbaseit_counter	5
confl_bufferpin	0
confl_deadlock	0
conflicts	0
confl_lock	0
confl_snapshot	0
confl_tablespace	0
curr_dwn	0
current_xlog_insert_lsn	0/28DDF28
curr_start_page	1661
curr_time	1
datid	16384
datname	tt
dcl_count	0
ddl_count	0
deadlocks	0
delete_count	0
dml_count	8
file_id	0
file_reset_num	0

file_trunc_num	0
global_instance_time_count	10
high_threshold_pages	0
high_threshold_writes	0
insert_count	0
last_replayed_read_ptr	0
local_max_ptr	0
locks_count	43
login_counter	3
logout_counter	2
low_threshold_pages	17
low_threshold_writes	2
lstiotim	10
maxiowtm	32
maxwritten_clean	0
miniotim	3
min_recovery_point	0
node_name	single_node1
numbackends	1
os_runtime_count	19
os_threads_count	23
p80	0
p95	0
pgwr_actual_flush_total_num	7
pgwr_last_flush_num	7
phyblkrd	501
phyblkwrt	7
phyblkwrt	6
phyrds	501
phywrts	6
phywrts	7
primary_flush_ptr	42852136
process_pending_counter	0
process_pending_total_dur	0
queue_head_page_rec_lsn	0/28DDB58
queue_rec_lsn	0/28DDB58
read_data_io_counter	0
read_data_io_total_dur	0
read_ptr	42851160
read_xlog_io_counter	2
read_xlog_io_total_dur	471

recovery_done_ptr	0
redo_done_time	0
redo_start_ptr	42851160
redo_start_time	732333220674305
remain_dirty_page_num	3
resp_avg	8090
resp_max	35491
resp_min	140
resp_total	40449
rollback_counter	0
select_count	5
single_node1-backend_used_memory	1
single_node1-cstore_used_memory	0
single_node1-dynamic_peak_memory	562
single_node1-dynamic_peak_shrctx	180
single_node1-dynamic_used_memory	561
single_node1-dynamic_used_shrctx	180
single_node1-gpu_dynamic_peak_memory	0
single_node1-gpu_dynamic_used_memory	0
single_node1-gpu_max_dynamic_memory	0
single_node1-max_backend_memory	348
single_node1-max_cstore_memory	512
single_node1-max_dynamic_memory	8142
single_node1-max_process_memory	12288
single_node1-max_sctpcgbase_memory	0
single_node1-max_shared_memory	3285
single_node1-other_used_memory	0
single_node1-pooler_conn_memory	0
single_node1-pooler_freeconn_memory	0
single_node1-process_used_memory	800
single_node1-sctpcgbase_peak_memory	0
single_node1-sctpcgbase_used_memory	0
single_node1-shared_used_memory	215
single_node1-storage_compress_memory	0
single_node1-udf_reserved_memory	0
speed	0
stats_reset	2023-03-16 17:34:43.424584+08
stats_reset	2023-03-16 17:34:25.277803+08
summary_file_iostat_count	62
temp_bytes	0
temp_files	0

```
 total_pages                          | 17
 total_writes                         | 2
 tup_deleted                          | 61
 tup_fetched                          | 64396
 tup_inserted                         | 9906
 tup_returned                         | 55952
 tup_updated                          | 305
 update_count                         | 0
 user_id                              | 10
 user_name                            | hlv
 wait_events_count                    | 401
 worker_info                          | no redo worker
 workload                             | default_pool
 write_data_io_counter                | 0
 write_data_io_total_dur              | 0
 writetim                             | 75
 xact_cgbaseit                          | 1235
 xact_rollback                        | 28
(148 rows)

postgres=# show global status;
            Variable_name             |             Value
--------------------------------------+-------------------------------
 apply_counter                        | 0
 apply_total_dur                      | 0
 avgiotim                             | 10
 bg_cgbaseit_counter                    | 168
 bg_resp_avg                          | 53200083
 bg_resp_max                          | 122185319
 bg_resp_min                          | 79
 bg_resp_total                        | 8937613869
 bg_rollback_counter                  | 2
 blk_read_time                        | 0
 blks_hit                             | 504050
 blks_read                            | 1444
 blk_write_time                       | 0
 buffers_alloc                        | 0
 buffers_backend                      | 0
 buffers_backend_fsync                | 0
 buffers_checkpoint                   | 0
 buffers_clean                        | 0
```

checkpoints_req	4
checkpoints_timed	29
checkpoint_sync_time	6794
checkpoint_write_time	5
ckpt_clog_flush_num	1
ckpt_csnlog_flush_num	0
ckpt_multixact_flush_num	1
ckpt_predicate_flush_num	0
ckpt_redo_point	0/28DE060
ckpt_twophase_flush_num	0
cgbaseit_counter	10
confl_bufferpin	0
confl_deadlock	0
conflicts	0
confl_lock	0
confl_snapshot	0
confl_tablespace	0
curr_dwn	0
current_xlog_insert_lsn	0/28DE180
curr_start_page	1684
curr_time	1
datid	16384
datname	tt
dcl_count	0
ddl_count	0
deadlocks	0
delete_count	0
dml_count	21
file_id	0
file_reset_num	0
file_trunc_num	2
global_instance_time_count	10
high_threshold_pages	0
high_threshold_writes	0
insert_count	0
last_replayed_read_ptr	0
local_max_ptr	0
locks_count	44
login_counter	9
logout_counter	8
low_threshold_pages	27

low_threshold_writes	4
lstiotim	8
maxiowtm	32
maxwritten_clean	0
miniotim	3
min_recovery_point	0
node_name	single_node1
numbackends	1
os_runtime_count	19
os_threads_count	23
p80	0
p95	0
pgwr_actual_flush_total_num	10
pgwr_last_flush_num	3
phyblkrd	520
phyblkwrt	10
phyblkwrt	8
phyrds	520
phywrts	8
phywrts	9
primary_flush_ptr	42852736
process_pending_counter	0
process_pending_total_dur	0
queue_head_page_rec_lsn	0/0
queue_rec_lsn	0/28DE060
read_data_io_counter	0
read_data_io_total_dur	0
read_ptr	42851160
read_xlog_io_counter	2
read_xlog_io_total_dur	471
recovery_done_ptr	0
redo_done_time	0
redo_start_ptr	42851160
redo_start_time	732333220674305
remain_dirty_page_num	0
resp_avg	47907
resp_max	254914
resp_min	116
resp_total	479066
rollback_counter	0
select_count	15

single_node1-backend_used_memory	1
single_node1-cstore_used_memory	0
single_node1-dynamic_peak_memory	571
single_node1-dynamic_peak_shrctx	181
single_node1-dynamic_used_memory	558
single_node1-dynamic_used_shrctx	181
single_node1-gpu_dynamic_peak_memory	0
single_node1-gpu_dynamic_used_memory	0
single_node1-gpu_max_dynamic_memory	0
single_node1-max_backend_memory	348
single_node1-max_cstore_memory	512
single_node1-max_dynamic_memory	8142
single_node1-max_process_memory	12288
single_node1-max_sctpcgbase_memory	0
single_node1-max_shared_memory	3285
single_node1-other_used_memory	0
single_node1-pooler_conn_memory	0
single_node1-pooler_freeconn_memory	0
single_node1-process_used_memory	806
single_node1-sctpcgbase_peak_memory	0
single_node1-sctpcgbase_used_memory	0
single_node1-shared_used_memory	220
single_node1-storage_compress_memory	0
single_node1-udf_reserved_memory	0
speed	0
stats_reset	2023-03-16 17:34:25.277803+08
stats_reset	2023-03-16 17:34:43.424584+08
summary_file_iostat_count	65
temp_bytes	0
temp_files	0
total_pages	27
total_writes	4
tup_deleted	61
tup_fetched	68794
tup_inserted	9906
tup_returned	59299
tup_updated	305
update_count	0
user_id	10
user_name	hlv
wait_events_count	401

```
 worker_info                          | no redo worker
 workload                             | default_pool
 write_data_io_counter                | 0
 write_data_io_total_dur              | 0
 writetim                             | 93
 xact_cgbaseit                          | 1324
 xact_rollback                        | 30
(148 rows)

postgres=# show session status;
            Variable_name             |             Value
--------------------------------------+-------------------------------
 apply_counter                        | 0
 apply_total_dur                      | 0
 avgiotim                             | 10
 bg_cgbaseit_counter                    | 168
 bg_resp_avg                          | 53200083
 bg_resp_max                          | 122185319
 bg_resp_min                          | 79
 bg_resp_total                        | 8937613869
 bg_rollback_counter                  | 2
 blk_read_time                        | 0
 blks_hit                             | 504050
 blks_read                            | 1444
 blk_write_time                       | 0
 buffers_alloc                        | 0
 buffers_backend                      | 0
 buffers_backend_fsync                | 0
 buffers_checkpoint                   | 0
 buffers_clean                        | 0
 checkpoints_req                      | 4
 checkpoints_timed                    | 29
 checkpoint_sync_time                 | 6794
 checkpoint_write_time                | 5
 ckpt_clog_flush_num                  | 1
 ckpt_csnlog_flush_num                | 0
 ckpt_multixact_flush_num             | 1
 ckpt_predicate_flush_num             | 0
 ckpt_redo_point                      | 0/28DE060
 ckpt_twophase_flush_num              | 0
 cgbaseit_counter                       | 10
```

confl_bufferpin	0
confl_deadlock	0
conflicts	0
confl_lock	0
confl_snapshot	0
confl_tablespace	0
curr_dwn	0
current_xlog_insert_lsn	0/28DE180
curr_start_page	1684
curr_time	1
datid	16384
datname	tt
dcl_count	0
ddl_count	0
deadlocks	0
delete_count	0
dml_count	21
file_id	0
file_reset_num	0
file_trunc_num	2
global_instance_time_count	10
high_threshold_pages	0
high_threshold_writes	0
insert_count	0
last_replayed_read_ptr	0
local_max_ptr	0
locks_count	44
login_counter	9
logout_counter	8
low_threshold_pages	27
low_threshold_writes	4
lstiotim	8
maxiowtm	32
maxwritten_clean	0
miniotim	3
min_recovery_point	0
node_name	single_node1
numbackends	1
os_runtime_count	19
os_threads_count	23
p80	0

p95	0
pgwr_actual_flush_total_num	10
pgwr_last_flush_num	3
phyblkrd	520
phyblkwrt	10
phyblkwrt	8
phyrds	520
phywrts	8
phywrts	9
primary_flush_ptr	42852736
process_pending_counter	0
process_pending_total_dur	0
queue_head_page_rec_lsn	0/0
queue_rec_lsn	0/28DE060
read_data_io_counter	0
read_data_io_total_dur	0
read_ptr	42851160
read_xlog_io_counter	2
read_xlog_io_total_dur	471
recovery_done_ptr	0
redo_done_time	0
redo_start_ptr	42851160
redo_start_time	732333220674305
remain_dirty_page_num	0
resp_avg	47907
resp_max	254914
resp_min	116
resp_total	479066
rollback_counter	0
select_count	15
single_node1-backend_used_memory	1
single_node1-cstore_used_memory	0
single_node1-dynamic_peak_memory	571
single_node1-dynamic_peak_shrctx	181
single_node1-dynamic_used_memory	558
single_node1-dynamic_used_shrctx	181
single_node1-gpu_dynamic_peak_memory	0
single_node1-gpu_dynamic_used_memory	0
single_node1-gpu_max_dynamic_memory	0
single_node1-max_backend_memory	348
single_node1-max_cstore_memory	512

```
 single_node1-max_dynamic_memory       | 8142
 single_node1-max_process_memory       | 12288
 single_node1-max_sctpcgbase_memory    | 0
 single_node1-max_shared_memory        | 3285
 single_node1-other_used_memory        | 0
 single_node1-pooler_conn_memory       | 0
 single_node1-pooler_freeconn_memory   | 0
 single_node1-process_used_memory      | 806
 single_node1-sctpcgbase_peak_memory   | 0
 single_node1-sctpcgbase_used_memory   | 0
 single_node1-shared_used_memory       | 220
 single_node1-storage_compress_memory  | 0
 single_node1-udf_reserved_memory      | 0
 speed                                 | 0
 stats_reset                           | 2023-03-16 17:34:25.277803+08
 stats_reset                           | 2023-03-16 17:34:43.424584+08
 summary_file_iostat_count             | 65
 temp_bytes                            | 0
 temp_files                            | 0
 total_pages                           | 27
 total_writes                          | 4
 tup_deleted                           | 61
 tup_fetched                           | 68794
 tup_inserted                          | 9906
 tup_returned                          | 59299
 tup_updated                           | 305
 update_count                          | 0
 user_id                               | 10
 user_name                             | hlv
 wait_events_count                     | 401
 worker_info                           | no redo worker
 workload                              | default_pool
 write_data_io_counter                 | 0
 write_data_io_total_dur               | 0
 writetim                              | 93
 xact_cgbaseit                         | 1324
 xact_rollback                         | 30
(148 rows)

postgres=# show status like 'xact%';
 Variable_name | Value
```

```
--------------+-------
 xact_cgbaseit   | 1390
 xact_rollback | 31
(2 rows)

postgres=# show status where variable_name = 'ckpt_redo_point';
  Variable_name  |   Value
-----------------+-----------
 ckpt_redo_point | 0/28DE2A0
(1 row)
```

2.4.1.5.65 SHOW_TABLES

功能描述

查看当前库（或 schema)的表清单。

注意事项

若不指定 db_name，查询的是当前库(或 schema)下的表清单。

语法格式

```
SHOW [FULL] TABLES
   [{FROM | IN} db_name]
   [LIKE 'pattern' | WHERE expr]
```

参数说明

db_name

库名（或 schema）,可选项，若不指定，则查询的是当前库或 schema。

LIKE 'pattern'

pattern 匹配显示结果第一列(列名为'Tables_in_dbname [pattern]'）。

示例

```
--创建简单表
postgres=# CREATE SCHEMA tst_schema;
postgres=# SET SEARCH_PATH TO tst_schema;
postgres=# CREATE TABLE tst_t1
postgres-# (
postgres(# id int primary key,
postgres(# name varchar(20) NOT NULL,
postgres(# addr text COLLATE "de_DE",
```

```
postgres(# phone text COLLATE "es_ES",
postgres(# addr_code text
postgres(# );
postgres=# CREATE VIEW tst_v1 AS SELECT * FROM tst_t1;
postgres=# CREATE TABLE t_t2(id int);
--查看库（或 SCHEMA）下表清单信息
postgres=# show tables;
 Tables_in_tst_schema
----------------------
 tst_t1
 tst_v1
 t_t2
postgres=# show full tables;
 Tables_in_tst_schema | Table_type
----------------------+------------
 tst_t1               | BASE TABLE
 tst_v1               | VIEW
 t_t2                 | BASE TABLE
postgres=# show full tables in tst_schema;
 Tables_in_tst_schema | Table_type
----------------------+------------
 tst_t1               | BASE TABLE
 tst_v1               | VIEW
 t_t2                 | BASE TABLE

--模糊匹配、过滤
postgres=# show full tables like '%tst%';
 Tables_in_tst_schema (%tst%) | Table_type
------------------------------+------------
 tst_t1                       | BASE TABLE
 tst_v1                       | VIEW
postgres=# show full tables where Table_type='VIEW';
 Tables_in_tst_schema | Table_type
----------------------+------------
 tst_v1               | VIEW
```

2.4.1.5.66 SHOW-TABLE-STATUS

功能描述

查看当前库（或 schema）的表状态。

注意事项

若不指定 db_name，查询的是当前库（或 schema）下的表状态。

语法格式

```
SHOW TABLE STATUS
    [{FROM | IN} db_name]
    [LIKE 'pattern' | WHERE expr]
```

参数说明

db_name

库名（或 schema），可选项，若不指定，则查询的是当前库或 schema。

LIKE 'pattern'

pattern 匹配显示结果第一列（列名为'Name [`pattern`]'）。

输出字段说明

字段	含义
Name	表名
Engine	存储引擎类型。取值范围：USTORE，表示表支持 Inplace-Update 存储引擎。ASTORE，表示表支持 Append-Only 存储引擎。
Version	默认值 NULL
Row_format	存储方式。取值范围：ROW，表示表的数据将以行式存储。COLUMN，表示表的数据将以列式存储。
Rows	行数

字段	含义
Avg_row_length	默认值 NULL
Data_length	数据大小，由 pg_relation_size(oid)获得
Max_data_length	默认值 NULL
Index_length	索引大小，由 pg_indexes_size(oid)获得
Data_free	默认值 NULL
Auto_increment	当 primary key 为 sequence 时获取其 last 值
Create_time	创建时间
Update_time	更新时间
Check_time	默认值 NULL
Collation	排序集
Checksum	默认值 NULL
Create_options	建表选项

字段	含义
comment	注释

示例

```
postgres=# CREATE SCHEMA tst_schema;
postgres=#
postgres=# SET SEARCH_PATH TO tst_schema;
postgres=#
postgres=# CREATE TABLE tst_t1
postgres-# (
postgres(# id serial primary key,
postgres(# name varchar(20),
postgres(# phone text
postgres(# )WITH(ORIENTATION=ROW, STORAGE_TYPE=USTORE);
postgres=#
postgres=# COMMENT ON TABLE tst_t1 IS 'this is cgbaseent';
postgres=#
postgres=# CREATE VIEW tst_v1 AS SELECT * FROM tst_t1;
postgres=#
postgres=# CREATE TABLE tst_t2
postgres-# (
postgres(# id serial primary key,
postgres(# name varchar(20),
postgres(# phone text
postgres(# )WITH(ORIENTATION=COLUMN);
postgres=#

--查看表状态
postgres=# show table status;
  Name   | Engine | Version | Row_format | Rows | Avg_row_length | Data_length |
Max_data_length | Index_length | Data_free | Auto_increment |     Create_time
|     Update_time     | Check_time |  Collation   | Checksum |
Create_options                    |     comment
-------+-------+---------+------------+------+----------------+-------------
-+-----------------+-------------+-----------+----------------+-------------
-------+--------------------+-----------+------------+----------+--------
-----------------------------------------+----------------
```

```
 tst_t1 | USTORE |          | ROW         |     0 |               0 |            0 |
0 |         57344 |         0 |               1 | 2022-10-18 09:04:24 | 2022-10-18
09:04:24 |             | en_US.UTF-8 |           |
{orientation=row,storage_type=ustore,compression=no} | this is cgbaseent
 tst_t2 | ASTORE |          | COLUMN      |     0 |               0 |        24576 |
0 |          8192 |         0 |               1 | 2022-10-18 09:04:24 | 2022-10-18
09:04:24 |             | en_US.UTF-8 |           |
{orientation=column,compression=low}                |
 tst_v1 |        |          |             |     0 |               0 |            0 |
0 |            0 |         0 |                 | 2022-10-18 09:04:24 | 2022-10-18
09:04:24 |             | en_US.UTF-8 |           |
|
(3 rows)

--like 模糊匹配
postgres=# show table status in tst_schema like '%tst_t%';
  Name   | Engine | Version | Row_format | Rows | Avg_row_length | Data_length |
Max_data_length | Index_length | Data_free | Auto_increment |     Create_time
|     Update_time     | Check_time |  Collation   | Checksum |
Create_options                    |     comment
--------+--------+---------+------------+------+----------------+-------------
-+-----------------+--------------+-----------+----------------+-------------
--------+---------------------+-----------+------------+----------+--------
-----------------------------------------------+-----------------
 tst_t1 | USTORE |          | ROW         |     0 |               0 |            0 |
0 |         57344 |         0 |               1 | 2022-10-18 09:04:24 | 2022-10-18
09:04:24 |             | en_US.UTF-8 |           |
{orientation=row,storage_type=ustore,compression=no} | this is cgbaseent
 tst_t2 | ASTORE |          | COLUMN      |     0 |               0 |        24576 |
0 |          8192 |         0 |               1 | 2022-10-18 09:04:24 | 2022-10-18
09:04:24 |             | en_US.UTF-8 |           |
{orientation=column,compression=low}                |
(2 rows)

--where 条件筛选
postgres=# show table status from tst_schema where Engine='ASTORE';
  Name   | Engine | Version | Row_format | Rows | Avg_row_length | Data_length |
Max_data_length | Index_length | Data_free | Auto_increment |     Create_time
|     Update_time     | Check_time |  Collation   | Checksum |
Create_options            | comment
```

```
--------+--------+---------+-----------+------+--------------+-----------
-+----------------+-------------+----------+--------------+-------------
--------+--------------------+-----------+------------+---------+--------
----------------------------+---------
 tst_t2 | ASTORE |           | COLUMN     |    0 |            0 |       24576 |
0 |          8192 |          0 |               1 | 2022-10-18 09:04:24 | 2022-10-18
09:04:24 |             | en_US.UTF-8 |           |
{orientation=column,compression=low} |
(1 row)
```

2.4.1.5.67 SHOW-TRIGGERS

功能描述

显示有关存储函数的信息。

注意事项

N/A

语法格式

```
SHOW TRIGGERS {FROM | IN} db_name [LIKE 'pattern' | WHERE expr]
```

参数说明

db_name

库名（或 schema）。

WHERE expr

筛选表达式。

LIKE 'pattern'

pattern 正则表达式匹配触发器名字。

返回结果集

字段名	类型	说明
Trigger	触发器名称	

字段名	类型	说明
Event	触发器事件（Insert、delete、update、truncate）	
Table	触发器定义的表	
Statement	触发器内容	
Timing	触发器时机（触发器之前或之后）	
Created	触发器创建时间	此处为空
sql_mode	触发器创建时的 sql mode	此处为空
Definer	创建者	
character_set_client	创建时客户端的字符集	此处为空
collation_connection	创建时客户端的排序规则	此处为空
Database Collation	数据库的排序集	

实例

```
postgres=# -- 创建触发器表和触发器函数
postgres=# CREATE TABLE test_trigger_src_tbl(id1 INT, id2 INT, id3 INT);
```

```
CREATE OR REPLACE FUNCTION tri_insert_func() RETURNS TRIGGER AS $$ DECLARE BEGIN
INSERT INTO test_trigger_des_tbl VALUES(NEW.id1, NEW.id2, NEW.id3); RETURN NEW;
END $$ LANGUAGE PLPGSQL;
-- 创建触发器
CREATE TRIGGER insert_trigger BEFORE INSERT ON test_trigger_src_tbl FOR EACH ROW
EXECUTE PROCEDURE tri_insert_func();
-- 查看信息
show triggers;
CREATE TABLE
postgres=# CREATE TABLE test_trigger_des_tbl(id1 INT, id2 INT, id3 INT);
CREATE TABLE
postgres=# CREATE OR REPLACE FUNCTION tri_insert_func() RETURNS TRIGGER AS
$$ DECLARE BEGIN INSERT INTO test_trigger_des_tbl VALUES(NEW.id1, NEW.id2,
NEW.id3); RETURN NEW; END $$ LANGUAGE PLPGSQL;
CREATE FUNCTION
postgres=# -- 创建触发器
postgres=# CREATE TRIGGER insert_trigger BEFORE INSERT ON test_trigger_src_tbl
FOR EACH ROW EXECUTE PROCEDURE tri_insert_func();
CREATE TRIGGER
postgres=# -- 查看信息
postgres=# show triggers;
    Trigger     | Event  |        Table         |              Statement
| Timing | Created | sql_mode | Definer | character_set_client |
collation_connection | Database Collation
---------------+--------+----------------------+--------------------------
--------+--------+---------+---------+---------+----------------------+----
-----------------+--------------------
 insert_trigger | INSERT | test_trigger_src_tbl | EXECUTE PROCEDURE
tri_insert_func() | BEFORE |          |          | wyc     |
|                      | en_US.UTF-8
(1 row)
```

2.4.1.5.68 SHOW-VARIABLES

功能描述

SHOW VARIABLES 显示系统变量的值。 该语句不需要任何特权。它仅需要连接到服务器的能力。 可以追加 like 和 where 子句来进行更进一步的匹配。

SHOW VARIABLES 接受可选 GLOBAL 或 SESSION 可变范围修饰符：

使用 GLOBAL 修饰符，该语句显示全局系统变量值。这些是用于初始化与 GBase 8c 的新连接的相应会话变量的值。如果变量没有全局值，则不会显示任何值。

使用 SESSION 修饰符，该语句显示对当前连接有效的系统变量值。

如果不存在修饰符，则默认值为 SESSION。

注意事项

N/A

语法格式

```
SHOW [GLOBAL | SESSION] VARIABLES [LIKE 'pattern' | WHERE expr];
```

参数说明

[GLOBAL | SESSION]

global 表示查询 guc 参数的默认值。 session 表示查询 guc 参数的会话值。

[LIKE 'pattern' | WHERE expr]

匹配表达式。

示例

```
--查询以 v 开头的 guc 参数
postgres=# show variables like 'v%';
       Variable_name        |   Value
----------------------------+-----------
 vacuum_cost_delay          | 0
 vacuum_cost_limit          | 200
 vacuum_cost_page_dirty     | 20
 vacuum_cost_page_hit       | 1
 vacuum_cost_page_miss      | 10
 vacuum_defer_cleanup_age   | 0
 vacuum_freeze_min_age      | 2000000000
 vacuum_freeze_table_age    | 4000000000
 vacuum_gtt_defer_check_age | 10000
 var_eq_const_selectivity   | off
 version_retention_age      | 0
(11 rows)
```

2.4.1.5.69 SHOW-WARNINGS

功能描述

显示有关存储函数的信息。

注意事项

添加的系统参数 sql_note，是设置 show warnings 是否显示 Note 级别的信息开关。

Code 字段是信息的错误码。其数字含义对应 ERRCODE 中的宏定义。其中各种信息的状态宏都是由 MAKE_SQLSTATE(ch1, ch2, ch3, ch4, ch5)生成。MAKE_SQLSTATE 作用是把字符 ch1 ~ ch5 的 ascii 码减去'0'，再取其二进制的后六位得到 res1 ~ res5，然后把这 5 个数据从低到高位组成一个 30 位二进制结果(res5res4res3res2res1)，转换成一个十进制数字，即就是错误码的数字。 不同的错误码数字对应不同的状态宏。

语法格式

```
SHOW WARNINGS [LIMIT [offset,] row_count]
SHOW COUNT(*) WARNINGS
SHOW ERRORS [LIMIT [offset,] row_count]
SHOW COUNT(*) ERRORS
```

参数说明

row_count

输出上条 sql，生成的 warnings/errors 信息的行数限制。

offset

从第几行信息开始显示。

添加系统参数

sql_note 该参数是设置 show warnings 是否显示 Note 级别的信息开关

返回结果集

字段名	类型	说明
Level	字符类型	信息的级别(Note/Warning/Error)
Code	整数类型	信息状态对应的错误码
Message	字符类型	信息内容

示例

```
postgres=# show sql_note;
 sql_note
```

```
----------
 on
(1 row)

postgres=# create table test(id int, name varchar default 11);
CREATE TABLE
postgres=# create table test(id int, name varchar default 11);
ERROR:  relation "test" already exists in schema "public"
DETAIL:  creating new table with existing name in the same schema
postgres=# show warnings limit 1;
 level |   code    |                      message
-------+-----------+-------------------------------------------------
 Error | 117571716 | relation "test" already exists in schema "public"
(1 row)

postgres=# show count(*) warnings;
 count
-------
     1
(1 row)

postgres=# CREATE OR REPLACE FUNCTION TEST_FUNC(tempdata char) RETURNS VOID AS
$$
postgres$# BEGIN
postgres$# raise info'TEST CHAR VALUE IS %',tempdata;
postgres$# END;
postgres$# $$ LANGUAGE plpgsql;
CREATE FUNCTION
postgres=# select TEST_FUNC('abc'::clob);
INFO:  TEST CHAR VALUE IS abc
CONTEXT:  referenced column: test_func
 test_func
-----------


(1 row)

postgres=# show warnings;
 level | code |        message
-------+------+-----------------------
 Note  |    0 | TEST CHAR VALUE IS abc
(1 row)
```

```
postgres=# set sql_note=false;
SET
postgres=# select TEST_FUNC('abc'::clob);
INFO:  TEST CHAR VALUE IS abc
CONTEXT:  referenced column: test_func
 test_func
-----------

(1 row)

postgres=# show warnings;
 level | code | message
-------+------+---------
(0 rows)

postgres=# SELECT pg_advisory_unlock(1), pg_advisory_unlock_shared(2),
pg_advisory_unlock(1, 1), pg_advisory_unlock_shared(2, 2);
WARNING:  you don't own a lock of type ExclusiveLock
CONTEXT:  referenced column: pg_advisory_unlock
WARNING:  you don't own a lock of type ShareLock
CONTEXT:  referenced column: pg_advisory_unlock_shared
WARNING:  you don't own a lock of type ExclusiveLock
CONTEXT:  referenced column: pg_advisory_unlock
WARNING:  you don't own a lock of type ShareLock
CONTEXT:  referenced column: pg_advisory_unlock_shared
 pg_advisory_unlock | pg_advisory_unlock_shared | pg_advisory_unlock |
pg_advisory_unlock_shared
--------------------+--------------------------+--------------------+-------
--------------------
 f                  | f                        | f                  | f
(1 row)

postgres=# show warnings;
  level   | code |                  message
---------+------+-----------------------------------------
 Warning |   64 | you don't own a lock of type ExclusiveLock
 Warning |   64 | you don't own a lock of type ShareLock
 Warning |   64 | you don't own a lock of type ExclusiveLock
 Warning |   64 | you don't own a lock of type ShareLock
(4 rows)
```

```
postgres=# show warnings limit 2, 4;
  level   | code |                  message
---------+------+-------------------------------------------
 Warning |   64 | you don't own a lock of type ExclusiveLock
 Warning |   64 | you don't own a lock of type ShareLock
(2 rows)
(3 rows)
```sql
--用 sql_note 控制存储 note 信息的开关。
CREATE OR REPLACE FUNCTION TEST_FUNC(tempdata char) RETURNS VOID AS $$
BEGIN
   raise info'TEST CHAR VALUE IS %',tempdata;
END;
$$ LANGUAGE plpgsql;
select TEST_FUNC('abc'::clob);
INFO:  TEST CHAR VALUE IS abc
CONTEXT:  referenced column: test_func
 test_func
-----------

(1 row)

show warnings;
 level | code |        message
-------+------+-----------------------
 Note  |    0 | TEST CHAR VALUE IS abc
(1 row)

set sql_note=false;
select TEST_FUNC('abc'::clob);
INFO:  TEST CHAR VALUE IS abc
CONTEXT:  referenced column: test_func
 test_func
-----------

(1 row)

show warnings;
 level | code | message
-------+------+---------
```
```

```
(0 rows)
```

2.4.1.5.70 UPDATE

功能描述

更新表中的数据。UPDATE 修改满足条件的所有行中指定的字段值，WHERE 子句声明条件，SET 子句指定的字段会被修改，没有出现的字段则保持它们的原值。

注意事项

本章节仅包含 dolphin 新增语法，原 GBase 8c 的 UPDATE 语法未作修改。原 GBase 8c 的 UPDATE 语法请参考章节 UPDATE

语法格式

单表更新：

```
[ WITH [ RECURSIVE ] with_query [, ...] ]
UPDATE [/*+ plan_hint */] [IGNORE] [ ONLY ] table_name [ partition_clause ] [ * ]
[ [ AS ] alias ]
SET {column_name = { expression | DEFAULT }
    |( column_name [, ...] ) = {( { expression | DEFAULT } [, ...] )
|sub_query }}[, ...]
    [ FROM from_list] [ WHERE condition ]
    [ ORDER BY {expression [ [ ASC | DESC | USING operator ]
    [ LIMIT { count } ]
    [ RETURNING {*
                | {output_expression [ [ AS ] output_name ]} [, ...] }];
```

多表更新：

```
[ WITH [ RECURSIVE ] with_query [, ...] ]
UPDATE [/*+ plan_hint */] [IGNORE] table_list
SET {column_name = { expression | DEFAULT }
    |( column_name [, ...] ) = {( { expression | DEFAULT } [, ...] )
|sub_query }}[, ...]
    [ FROM from_list] [ WHERE condition ];
where sub_query can be:
SELECT [ ALL | DISTINCT [ ON ( expression [, ...] ) ] ]
{ * | {expression [ [ AS ] output_name ]} [, ...] }
[ FROM from_item [, ...] ]
[ WHERE condition ]
[ GROUP BY grouping_element [, ...] ]
[ HAVING condition [, ...] ]
```

```
[ ORDER BY {expression [ [ ASC | DESC | USING operator ] |
nlssort_expression_clause ] [ NULLS { FIRST | LAST } ]} [, ...] ]
[ LIMIT { [offset,] count | ALL } ]
```

参数说明

IGNORE

若带有 IGNORE 关键字的 UPDATE 语句执行时在指定场景引发了 Error，则会将 Error 降级为 Warning，且继续语句的执行，不会影响其他数据的操作。能使 Error 降级的场景有：

1.违反非空约束时：

若执行的 SQL 语句违反了表的非空约束，使用此 hint 可将 Error 降级为 Warning，并根据 GUC 参数 sql_ignore_strategy 的值采用以下策略的一种继续执行：

sql_ignore_startegy 为 ignore_null 时，忽略违反非空约束的行的 UPDATE 操作，并继续执行剩余数据操作。

sql_ignore_startegy 为 overwrite_null 时，将违反约束的 null 值覆写为目标类型的默认值，并继续执行剩余数据操作。

说明：

GUC 参数 sql_ignore_strategy 为枚举类型，可选值有：ignore_null, overwrite_null

2.违反唯一约束时：

若执行的 SQL 语句违反了表的唯一约束，使用此 hint 可将 Error 降级为 Warning，忽略违反约束的行的 UPDATE 操作，并继续执行剩余数据操作。

3.分区表无法匹配到合法分区时

在对分区表进行 UPDATE 操作时，若某行数据无法匹配到表格的合法分区，使用此 hint 可将 Error 降级为 Warning，忽略该行操作，并继续执行剩余数据操作。

4.更新值向目标列类型转换失败时：

执行 UPDATE 语句时，若发现新值与目标列类型不匹配，使用此 hint 可将 Error 降级为 Warning，并根据新值与目标列的具体类型采取以下策略的一种继续执行：

- 当新值类型与列类型同为数值类型时：

若新值在列类型的范围内，则直接进行更新；若新值在列类型范围外，则以列类型的最大/最小值替代。

- 当新值类型与列类型同为字符串类型时：

  若新值长度在列类型限定范围内，则以直接进行更新；若新值长度在列类型的限定范围外，则保留列类型长度限制的前 n 个字符。

- 若遇到新值类型与列类型不可转换时：

  更新为列类型的默认值。

IGNORE 关键字不支持列存，无法在列存表中生效。

示例

IGNORE 关键字

为使用 ignore_error hint，需要创建 B 兼容模式的数据库，名称为 db_ignore。

```
create database db_ignore dbcompatibility 'B';
\c db_ignore
忽略非空约束
db_ignore=# create table t_not_null(num int not null);
CREATE TABLE
-- 采用忽略策略
db_ignore=# set sql_ignore_strategy = 'ignore_null';
SET
db_ignore=# insert into t_not_null values (1);
INSERT 0 1
db_ignore=# select * from t_not_null ;
 num
-----
   1
(1 row)
db_ignore=# update /*+ ignore_error */ t_not_null set num = null where num = 1;
WARNING:  null value in column "num" violates not-null constraint
DETAIL:  Failing row contains (null).
UPDATE 0
db_ignore=# select * from t_not_null ;
 num
-----
   1
(1 row)
-- 采用覆写策略
db_ignore=# delete from t_not_null;
db_ignore=# set sql_ignore_strategy = 'overwrite_null';
```

```
SET
db_ignore=# insert into t_not_null values (1);
WARNING:  null value in column "num" violates not-null constraint
DETAIL:  Failing row contains (null).
INSERT 0 1
db_ignore=# select * from t_not_null ;
 num
-----
   1
(1 rows)
db_ignore=# update /*+ ignore_error */ t_not_null set num = null where num = 1;
WARNING:  null value in column "num" violates not-null constraint
DETAIL:  Failing row contains (null).
UPDATE 1
db_ignore=# select * from t_not_null ;
 num
-----
   0
(1 rows)
```

忽略唯一约束

```
db_ignore=# create table t_unique(num int unique);
NOTICE:  CREATE TABLE / UNIQUE will create implicit index "t_unique_num_key" for
table "t_unique"
CREATE TABLE
db_ignore=# insert into t_unique values(1), (2);
INSERT 0 1
db_ignore=# update /*+ ignore_error */ t_unique set num = 1 where num = 2;
WARNING:  duplicate key value violates unique constraint in table "t_unique"
UPDATE 0
db_ignore=# select * from t_unique ;
 num
-----
   1
   2
(2 rows)
```

忽略分区表无法匹配到合法分区

```
db_ignore=# CREATE TABLE t_ignore
db_ignore-# (
db_ignore(#     col1 integer NOT NULL,
db_ignore(#     col2 character varying(60)
db_ignore(# ) WITH(segment = on) PARTITION BY RANGE (col1)
```

```
db_ignore-# (
db_ignore(#     PARTITION P1 VALUES LESS THAN(5000),
db_ignore(#     PARTITION P2 VALUES LESS THAN(10000),
db_ignore(#     PARTITION P3 VALUES LESS THAN(15000)
db_ignore(# );
CREATE TABLE
db_ignore=# insert into t_ignore values(3000);
INSERT 0 1
db_ignore=# select * from t_ignore ;
 col1 | col2
------+------
 3000 |
(1 row)
db_ignore=# update /*+ ignore_error */ t_ignore set col1 = 20000 where col1 = 3000;
WARNING:  fail to update partitioned table "t_ignore".new tuple does not map to
any table partition.
UPDATE 0
db_ignore=# select * from t_ignore ;
 col1 | col2
------+------
 3000 |
(1 row)
```

更新值向目标列类型转换失败

```
-- 当新值类型与列类型同为数值类型
db_ignore=# create table t_tinyint(num tinyint);
CREATE TABLE
db_ignore=# insert into t_tinyint values(1);
INSERT 0 1
db_ignore=# select * from t_tinyint;
 num
-----
 1
(1 row)
db_ignore=# update /*+ignore_error */ t_tinyint set num = 10000 where num = 1;
WARNING:  tinyint out of range
CONTEXT:  referenced column: num
UPDATE 1
db_ignore=# select * from t_tinyint;
 num
-----
 127
```

```
(1 row)
-- 当新值类型与列类型同为字符类型时
db_ignore=# create table t_varchar5(content varchar(5));
CREATE TABLE
db_ignore=# insert into t_varchar5 values('abc');
INSERT 0 1
db_ignore=# select * from t_varchar5;
 content
---------
 abc
(1 row)
db_ignore=# update /*+ ignore_error */ t_varchar5 set content = 'abcdefghijklmn'
where content = 'abc';
WARNING:  value too long for type character varying(5)
CONTEXT:  referenced column: content
UPDATE 1
db_ignore=# select * from t_varchar5;
 content
---------
 abcde
(1 row)
```

2.4.1.5.71 USE-DB_NAME

功能描述

USE db_name 语句将 db_name 数据库作为默认（当前）数据库使用，用于后续语句。该数据库保持为默认数据库，直到语段的结尾，或者直到发布一个不同的 USE 语句。

注意事项

N/A

语法格式

USE db_name

参数说明

db_name

数据库名。

示例

```
--切换到 db1 库
```

```
postgres=# USE db1;
SET
postgres=# CREATE TABLE test(a text);
CREATE TABLE
postgres=# INSERT INTO test VALUES('db1');
INSERT 0 1
--切换到 db2 库
postgres=# USE db2;
SET
postgres=# CREATE TABLE test(a text);
CREATE TABLE
postgres=# INSERT INTO test VALUES('db2');
INSERT 0 1
postgres=# select a from db1.test;
  a
-----
 db1
(1 row)
postgres=# select a from db2.test;
  a
-----
 db2
(1 row)
postgres=# select a from test;
  a
-----
 db2
(1 row)
--切换到 db1 库
postgres=# USE db1;
SET
postgres=# select a from test;
  a
-----
 db1
(1 row)
```

相关链接

N/A

2.4.1.5.72 SELECT-HINT

功能描述

设置本次查询执行内生效的查询优化相关 GUC 参数。本章节涉及的是 dolphin 关于 select-hint 的语法增强，对内核支持的 hint 详见使用 Plan Hint 进行调优.

语法格式

set_var(param = value)

参数说明

param

参数名。

目前支持使用 Hint 设置生效的参数有

布尔类:

enable_bitmapscan, enable_hashagg , enable_hashjoin, enable_indexscan , enable_indexonlyscan, enable_material , enable_mergejoin, enable_nestloop , enable_index_nestloop, enable_seqscan , enable_sort, enable_tidscan , partition_iterator_elimination , partition_page_estimation , enable_functional_dependency , var_eq_const_selectivity,

整形类:

query_dop

浮点类:

cost_weight_index、default_limit_rows、seq_page_cost、random_page_cost、cpu_tuple_cost、cpu_index_tuple_cost、cpu_operator_cost、effective_cache_size

枚举类型:

try_vector_engine_strategy

字符串类型:

dolphin.optimizer_switch

value

参数的取值。

2.4.2 系统视图

2.4.2.1 PG TYPE NONSTRICT BASIC VALUE

PG_TYPE_NONSTRICT_BASIC_VALUE 视图存储类型的基础值，用于 insert values() 的插入。默认只有系统管理员权限才可以访问此系统表，普通用户需要授权才可以访问。

表 1-8 PG_TYPE_NONSTRICT_BASIC_VALUE 字段

名称	类型	描述
typename	text	类型名称
basic_value	text	类型基础值

2.4.2.2 INDEX STATISTIC

INDEX_STAITISTIC 视图存储当前数据库的索引信息。

表 1-9 INDEX_STAITISTIC 字段

名称	类型	描述
namespace	name	索引所属表空间
table	name	索引所属表
non_unique	boolean	是否是唯一索引
key_name	name	索引名
seq_in_index	smallint	索引列在索引中的序号

名称	类型	描述
column_name	name	索引列的列名
collation	text	取值有 A（默认，升序），D（降序）、NULL（索引不支持排序）
cardinality	double precision	根据 pg_statistic.stadistinct 和 pg_class.reltuples 计算得到： stadistinct > 0: stadistinct stadistinct = 0: NULL stadistinct < 0: reltuples * stadistinct * -1
sub_part	text	索引前缀。如果该列仅被部分索引，则是索引字符的数量；如果整个列都被索引，则是 NULL。当前不支持前缀索引，NULL
packed	text	如何打包 key 值，create table 时指定 pack_keys；否则返回 NULL。当前不支持，为 NULL
null	text	可能包含 NULL 值则是 YES，否则为
index_type	name	使用的索引方法：BTREE、HASH 等
cgbaseent	text	pg_index 表中记录的 indisusable 为 true 则显示 disabled，false 则显示"
index_cgbaseent	text	创建索引时 COMMENT 指定的注释信息

2.4.3 GUC 参数说明

2.4.3.1 dolphin.sql_mode

取值范围：字符串

默认值：'sql_mode_strict,sql_mode_full_group,pipes_as_concat,ansi_quotes,no_zero_date,pad_char_to_full_length'

参数说明：参数值为逗号间隔的字符串，仅允许合法字符串设定，不合法情况下，启动后报 warning。同样，设置时候，如果新值非法，则报 warning 并且不修改老值。当前有几种场景会用到 sql_mode：

- sql_mode_strict：插入不符合当前列类型的值时,进行数据转换;分两种场景，insert into table values(…) 和 insert into table select … 主要涉及到各种数据类型之间的互相转换，目前涉及的类型有 tinyint[unsigned],smallint[unsigned],int[unsigned],bigint[unsigned],float,double,numeric,clob,char 和 varchar；
- sql_mode_strict：插入的列值长度超过此列所限定的长度时,赋予该列最大或最小值，涉及的类型有 tinyint[unsigned],smallint[unsigned],int[unsigned],bigint[unsigned],float,double,numeric,clob,char 和 varchar;
- sql_mode_strict：insert 时，属性是非空且没有默认值的列，且没有在 insert 的列表中，则为其添加默认值；(涉及的类型同上)
- sql_mode_strict：支持对属性是非空且没有默认值的列显式插入 default；(涉及的类型同上)
- sql_mode_full_group：
- 出现在 select 列表中的列（不使用聚合函数），是否一定要出现在 group by 子句中。当处在 sql_mode_full_group 模式（默认模式）下，如果 select 列表中的列没有使用聚合函数，也没有出现在 group by 子句，那么会报错，如果不在此模式下，则会执行成功，并在所有符合条件的元组中选取第一个元组。
- 出现在 order by 中的列，是否一定要出现在 distinct 中(注意是 distinct，不是 distinct on)。当处在 sql_mode_full_group 模式（默认模式）下，不允许没有出现在 distinct 中的列出

现在 order by 子句中，否则允许。

- pipes_as_concat：控制 || 当成连接符还是 或操作符
- ansi_quotes：主要是针对出现在各种需要使用双引号表示字符串值的地方。当 ansi_quotes 打开，就表示此时的双引号中的内容要作为对象引用看待；当 ansi_quotes 关闭时，表示双引号中的内容要作为字符串的值看待。当关闭 ansi_quotes 时，会导致部分元命令失效，失效的元命令如下表所示：

参数	参数说明
\d[S+]	列出当前 search_path 中模式下所有的表、视图和序列。当 search_path 中不同模式存在同名对象时，只显示 search_path 中位置靠前模式下的同名对象。
\d+ [PATTERN]	列出所有表、视图和索引。
\da[S] [PATTERN]	列出所有可用的聚集函数以及它们操作的数据类型和返回值类型。
\db[+] [PATTERN]	列出所有可用的表空间。
\dc[S+] [PATTERN]	列出所有字符集之间的可用转换。
\dC[+] [PATTERN]	列出所有类型转换。
\dd[S] [PATTERN]	显示所有匹配 PATTERN 的描述。
\ddp [PATTERN]	显示所有默认的使用权限。

参数	参数说明
\dD[S+] [PATTERN]	列出所有可用域。
\ded[+] [PATTERN]	列出所有的 Data Source 对象。
\det[+] [PATTERN]	列出所有的外部表。
\des[+] [PATTERN]	列出所有的外部服务器。
\deu[+] [PATTERN]	列出用户映射信息。
\dew[+] [PATTERN]	列出封装的外部数据。
\df[antw][S+] [PATTERN]	列出所有可用函数以及它们的参数和返回的数据类型。a 代表聚集函数，n 代表普通函数，t 代表触发器，w 代表窗口函数。
\dF[+] [PATTERN]	列出所有的文本搜索配置信息。
\dFd[+] [PATTERN]	列出所有的文本搜索字典。
\dFp[+] [PATTERN]	列出所有的文本搜索分析器。
\dFt[+] [PATTERN]	列出所有的文本搜索模板。

参数	参数说明
\dl	\lo_list 的别名，显示一个大对象的列表。
\dL[S+] [PATTERN]	列出可用的程序语言。
\dm[S+] [PATTERN]	列出物化视图
\dn[S+] [PATTERN]	列出所有的模式（名称空间）。
\do[S] [PATTERN]	列出所有可用的操作符以及它们的操作数和返回的数据类型。
\dO[S+] [PATTERN]	列出排序规则
\dp [PATTERN]	列出一列可用的表、视图以及相关的权限信息。
\dT[S+] [PATTERN]	列出所有的数据类型。
\dE[S+] [PATTERN]	这一组命令，字母 E、i、s、t 和 v 分别代表着外部表、索引、序列、表和视图。可以以任意顺序指定其中一个或者它们的组合来列出这些对象。例如：\dit 列出所有的索引和表。在命令名称后面追加+，则每一个对象的物理尺寸以及相关的描述也会被列出。
\dx[+] [PATTERN]	列出安装数据库的扩展信息。
\l[+]	列出服务器上所有数据库的名称、所有者、字符集编码以及使用权限。

参数	参数说明
\z [PATTERN]	列出数据库中所有表、视图和序列以及它们相关的访问特权。

ansi_quotes 关闭情况下，如果需要用到数据库关键字作为对象标识符，或者出于规范性要求需要包裹所有对象标识符，可以使用反引号(`)替代双引号。

使用反引号(`)的情况下，表名称（受到参数 lower_case_table_names 控制）之外，其他包围的列名称，索引名称等都会做自动的小写化处理，返回数据的列也均为小写名称。

no_zero_date：控制 '0000-00-00' 是否为合法日期，支持 DATE、DATETIME 类型

参数	表现
no_zero_date, sql_mode_strict	非法日期，报错（使用 update/insert ignore 时告警）
no_zero_date	非法日期，告警
sql_mode_strict	合法日期，无告警
-	合法日期，无告警

pad_char_to_full_length：控制 char 类型查询时是否删除尾部空格。

该参数属于 USERSET 类型参数，请参考表 1 中对应设置方法进行设置。

示例：

```
--创建表 test1。
postgres=# CREATE TABLE test1
(
  a1 smallint not null,
  a2 int not null,
  a3 bigint not null,
  a4 float not null,
  a5 double not null,
```

```
  a6 numeric not null,
  a7 varchar(5) not null
);
CREATE TABLE
--向表中插入记录失败。
postgres=# insert into test1(a1,a2) values(123412342342314,3453453453434324);
ERROR:  smallint out of range
CONTEXT:  referenced column: a1
--查询表失败
postgres=# select a1,a2 from test1 group by a1;
ERROR:  column "test1.a2" must appear in the GROUP BY clause or be used in an
aggregate function
LINE 1: select a1,a2 from test1 group by a1;

--向表中插入记录成功。
postgres=# set dolphin.sql_mode = '';
SET
postgres=# insert into test1(a1,a2) values(123412342342314,3453453453434324);
WARNING:  invalid input syntax for numeric: ""
CONTEXT:  referenced column: a6
WARNING:  smallint out of range
CONTEXT:  referenced column: a1
WARNING:  integer out of range
CONTEXT:  referenced column: a2
INSERT 0 1
--查询表成功
postgres=# select a1,a2 from test1 group by a1;
  a1   |     a2
-------+------------
 32767 | 2147483647
(1 row)

--删除表
postgres=# DROP TABLE test1;
DROP TABLE

--ansi_quotes 效果展示
postgres=# create database test_db dbcompatibility 'B';
CREATE DATABASE
postgres=# \c test_db
Non-SSL connection (SSL connection is recgbaseended when requiring high-security)
```

```
You are now connected to database "test_db" as user "luozihao".
test_db=# show dolphin.sql_mode ;
                                          dolphin.sql_mode
------------------------------------------------------------------------------
-------------------------

sql_mode_strict,sql_mode_full_group,pipes_as_concat,ansi_quotes,no_zero_date,
pad_char_to_full_length
(1 row)

test_db=# create table test(a varchar(20));
CREATE TABLE
test_db=# insert into test values('test');
INSERT 0 1
test_db=# select "a" from test;
  a
------
 test
(1 row)

test_db=# set dolphin.sql_mode to
'sql_mode_strict,sql_mode_full_group,pipes_as_concat,no_zero_date,pad_char_to
_full_length';
SET
test_db=# select "a" from test;
 ?column?
----------
 a
(1 row)
```

2.4.3.2 dolphin.b_db_timestamp

参数说明：参数值为浮点数，该参数影响 dolphin 中的 curdate/current_time/curtime/current_timestamp/localtime/localtimestamp/now 函数。当此参数值为 0 时，以上函数返回当前日期或时间；若参数值位于区间[1,2147483647]，则上述函数以该 GUC 参数的值作为秒数偏移，返回 1970 年 01 月 01 日 00:00:00 UTC + 秒数偏移 + 当前时区偏移的对应日期或时间。设置此参数时，若值不在上述合法区间内，会报错。

该参数属于 USERSET 类型参数，请参考表 1 中对应设置方法进行设置。

取值范围：0∪[1.0, 2147483647.0]

默认值：0

示例：

```
postgres=# show dolphin.b_db_timestamp;
 b_db_timestamp
----------------
 0
(1 row)

postgres=# select now();
        now()
--------------------
 2022-09-18 19:52:23
(1 row)

postgres=# set dolphin.b_db_timestamp = 1.0;
SET
postgres=# select now();
        now()
--------------------
 1970-01-01 08:00:01
(1 row)
```

2.4.3.3 dolphin.default_week_format

参数说明：参数值为整数，该参数影响 dolphin 插件中的 week 函数，该参数的取值范围为[0,7]，分别对应 8 种不同的计算策略，这些策略的详细内容参见时间/日期函数中的 week 函数说明。当此 GUC 参数设置的值超过对应边界值时，会报 warning，并且将此 GUC 参数的值设置为对应边界值。

该参数属于 SIGHUP 类型参数，请参考表 1 中对应设置方法进行设置。

取值范围：[0, 7]

默认值：0

示例：

```
postgres=# show dolphin.default_week_format;
dolphin.default_week_format
---------------------
0
(1 row)
```

```
postgres=# select week('2000-1-1');
week
------
    0
(1 row)

postgres=# alter system set dolphin.default_week_format = 2;
ALTER SYSTEM SET

postgres=# select week('2000-1-1');
week
------
52
(1 row)
```

2.4.3.4 dolphin.lc_time_names

参数说明：参数值为字符串，该参数控制 dolphin 插件中 dayname/monthname 函数以何种语言输出结果。该参数的取值有 111 种。设置参数时，若值不在合法取值范围内，则会报错。

该参数属于 SIGHUP 类型参数，请参考表 1 中对应设置方法进行设置。

取值范围： lc_time_names 语言集有如下可供选择的值：

参数值	语言集
ar_AE	Arabic - United Arab Emirates
ar_BH	Arabic - Bahrain
ar_DZ	Arabic - Algeria
ar_EG	Arabic - Egypt

参数值	语言集
ar_IN	Arabic - India
ar_IQ	Arabic - Iraq
ar_JO	Arabic - Jordan
ar_KW	Arabic - Kuwait
ar_LB	Arabic - Lebanon
ar_LY	Arabic - Libya
ar_MA	Arabic - Morocco
ar_OM	Arabic - Oman
ar_QA	Arabic - Qatar
ar_SA	Arabic - Saudi Arabia
ar_SD	Arabic - Sudan
ar_SY	Arabic - Syria

参数值	语言集
ar_TN	Arabic - Tunisia
ar_YE	Arabic - Yemen
be_BY	Belarusian - Belarus
bg_BG	Bulgarian - Bulgaria
ca_ES	Catalan - Spain
cs_CZ	Czech - Czech Republic
da_DK	Danish - Denmark
de_AT	German - Austria
de_BE	German - Belgium
de_CH	German - Switzerland
de_DE	German - Germany
de_LU	German - Luxembourg

参数值	语言集
el_GR	Greek - Greece
en_AU	English - Australia
en_CA	English - Canada
en_GB	English - United Kingdom
en_IN	English - India
en_NZ	English - New Zealand
en_PH	English - Philippines
en_US	English - United States
en_ZA	English - South Africa
en_ZW	English - Zimbabwe
es_AR	Spanish - Argentina
es_BO	Spanish - Bolivia

参数值	语言集
es_CL	Spanish - Chile
es_CO	Spanish - Colombia
es_CR	Spanish - Costa Rica
es_DO	Spanish - Dominican Republic
es_EC	Spanish - Ecuador
es_ES	Spanish - Spain
es_GT	Spanish - Guatemala
es_HN	Spanish - Honduras
es_MX	Spanish - Mexico
es_NI	Spanish - Nicaragua
es_PA	Spanish - Panama
es_PE	Spanish - Peru

参数值	语言集
es_PR	Spanish - Puerto Rico
es_PY	Spanish - Paraguay
es_SV	Spanish - El Salvador
es_US	Spanish - United States
es_UY	Spanish - Uruguay
es_VE	Spanish - Venezuela
et_EE	Estonian - Estonia
eu_ES	Basque - Spain
fi_FI	Finnish - Finland
fo_FO	Faroese - Faroe Islands
fr_BE	French - Belgium
fr_CA	French - Canada

参数值	语言集
fr_CH	French - Switzerland
fr_FR	French - France
fr_LU	French - Luxembourg
gl_ES	Galician - Spain
gu_IN	Gujarati - India
he_IL	Hebrew - Israel
hi_IN	Hindi - India
hr_HR	Croatian - Croatia
hu_HU	Hungarian - Hungary
id_ID	Indonesian - Indonesia
is_IS	Icelandic - Iceland
it_CH	Italian - Switzerland

参数值	语言集
it_IT	Italian - Italy
ja_JP	Japanese - Japan
ko_KR	Korean - Republic of Korea
lt_LT	Lithuanian - Lithuania
lv_LV	Latvian - Latvia
mk_MK	Macedonian - North Macedonia
mn_MN	Mongolia - Mongolian
ms_MY	Malay - Malaysia
nb_NO	Norwegian(Bokmål) - Norway
nl_BE	Dutch - Belgium
nl_NL	Dutch - The Netherlands
no_NO	Norwegian - Norway

参数值	语言集
pl_PL	Polish - Poland
pt_BR	Portugese - Brazil
pt_PT	Portugese - Portugal
rm_CH	Romansh - Switzerland
ro_RO	Romanian - Romania
ru_RU	Russian - Russia
ru_UA	Russian - Ukraine
sk_SK	Slovak - Slovakia
sl_SI	Slovenian - Slovenia
sq_AL	Albanian - Albania
sr_RS	Serbian - Serbia
sv_FI	Swedish - Finland

参数值	语言集
sv_SE	Swedish - Sweden
ta_IN	Tamil - India
te_IN	Telugu - India
th_TH	Thai - Thailand
tr_TR	Turkish - Turkey
uk_UA	Ukrainian - Ukraine
ur_PK	Urdu - Pakistan
vi_VN	Vietnamese - Vietnam
zh_CN	Chinese - China
zh_HK	Chinese - Hong Kong
zh_TW	Chinese - Taiwan

默认值：'en_US'

示例：

```
postgres=# select dayname('2000-1-1');
```

```
dayname
----------
Saturday
(1 row)

postgres=# alter system set dolphin.lc_time_names = 'zh_CN';
ALTER SYSTEM SET

postgres=# select dayname('2000-1-1');
dayname
---------
星期六
(1 row)
```

2.4.3.5 dolphin.b_compatibility_mode

参数说明：参数值为布尔类型，该参数影响 dolphin 插件中的部分冲突的函数和操作符等，参数开启时这些函数和操作符会执行兼容性逻辑，关闭时则保持 GBase 8c 原有的逻辑。

当前影响的操作符有：

LIKE/NOT LIKE

(字符类型异或)^

(数字类型异或)^

&&

#

影响的函数有：

LAST_DAY

TIMESTAMPDIFF

FORMAT

EXTRACT

CAST

其他影响的参数：

?

该参数属于 USERSET 类型参数，请参考表 1 中对应设置方法进行设置。

取值范围：布尔型

on 表示使用新增兼容性功能。

off 表示关闭兼容性功能，使用内核原有功能。

默认值：off

2.4.3.6 version_cgbaseent

参数说明：该参数目前为只读参数，且未实现其具体意义。参数值为字符串类型，表示数据库服务端及许可证信息。

该参数目前属于 INTERNAL 类型参数，用户无法对其进行设置。

取值范围：字符串

默认值：GBase 8c Server(MulanPSL-2.0)

2.4.3.7 auto_increment_increment

参数说明：该参数目前为只读参数，且未实现其具体意义。参数值为整数类型，表示自增列的自增步长。

该参数目前属于 INTERNAL 类型参数，用户无法对其进行设置。

取值范围：[1, 65535]

默认值：1

2.4.3.8 character_set_client

参数说明：该参数目前为只读参数，且未实现其具体意义。参数值为字符串类型，表示客户端使用该字符集。

该参数目前属于 INTERNAL 类型参数，用户无法对其进行设置。

取值范围：字符串

默认值：utf8

2.4.3.9 character_set_connection

参数说明：该参数目前为只读参数，且未实现其具体意义。参数值为字符串类型，表示没有字符集引入程序时，使用该字符集。

该参数目前属于 INTERNAL 类型参数，用户无法对其进行设置。

取值范围：字符串

默认值：utf8

2.4.3.10 character_set_results

参数说明：该参数目前为只读参数，且未实现其具体意义。参数值为字符串类型，表示服务端使用该字符集向客户端返回查询结果。

该参数目前属于 INTERNAL 类型参数，用户无法对其进行设置。

取值范围：字符串

默认值：utf8

2.4.3.11 character_set_server

参数说明：该参数目前为只读参数，且未实现其具体意义。参数值为字符串类型，表示服务端使用该字符集。

该参数目前属于 INTERNAL 类型参数，用户无法对其进行设置。

取值范围：字符串

默认值：latin1

2.4.3.12 collation_server

参数说明：该参数目前为只读参数，且未实现其具体意义。参数值为字符串类型，表示服务端使用该排序规则。

该参数目前属于 INTERNAL 类型参数，用户无法对其进行设置。

取值范围：字符串

默认值：latin1_swedish_ci

2.4.3.13 collation_connection

参数说明：该参数目前为只读参数，且未实现其具体意义。参数值为字符串类型，表示连接字符集使用该排序规则。

该参数目前属于 INTERNAL 类型参数，用户无法对其进行设置。

取值范围：字符串

默认值：无

2.4.3.14 init_connect

参数说明：该参数目前为只读参数，且未实现其具体意义。参数值为字符串类型，表示

连接初始化时执行的 SQL 语句。

该参数目前属于 INTERNAL 类型参数，用户无法对其进行设置。

取值范围：字符串

默认值：无

2.4.3.15 interactive_timeout

参数说明：该参数目前为只读参数，且未实现其具体意义。参数值为整数类型，表示交互式连接在持续无活动该秒数后，服务端会将其关闭。

该参数目前属于 INTERNAL 类型参数，用户无法对其进行设置。

取值范围：[1, 31536000]

默认值：28800

2.4.3.16 license

参数说明：该参数目前为只读参数，且未实现其具体意义。参数值为字符串类型，表示服务端使用该许可证。

该参数目前属于 INTERNAL 类型参数，用户无法对其进行设置。

取值范围：字符串

默认值：MulanPSL-2.0

2.4.3.17 max_allowed_packet

参数说明：该参数目前为只读参数，且未实现其具体意义。参数值为整数类型，表示数据包的大小上限（字节）。

该参数目前属于 INTERNAL 类型参数，用户无法对其进行设置。

取值范围：[1024, 1073741824]

默认值：4194304

2.4.3.18 net_buffer_length

参数说明：该参数目前为只读参数，且未实现其具体意义。参数值为整数类型，表示缓冲区的默认大小，缓冲区的大小可以动态的扩张到 max_allowed_packet，并在 SQL 语句结束后还原。

该参数目前属于 INTERNAL 类型参数，用户无法对其进行设置。

取值范围：[1024, 1048576]

默认值：16384

2.4.3.19 net_write_timeout

参数说明：该参数目前为只读参数，且未实现其具体意义。参数值为整数类型，表示在等待写入该秒数后，服务端会将其中止。

该参数目前属于 INTERNAL 类型参数，用户无法对其进行设置。

取值范围：[1, 31536000]

默认值：60

2.4.3.20 query_cache_size

参数说明：该参数目前为只读参数，且未实现其具体意义。参数值为整数类型，表示在缓存查询结果时，分配的内存大小（字节）。

该参数目前属于 INTERNAL 类型参数，用户无法对其进行设置。

取值范围：[0, 9223372036854775807]

默认值：1048576

2.4.3.21 system_time_zone

参数说明：该参数目前为只读参数，且未实现其具体意义。参数值为字符串类型，表示服务器系统时区。

该参数目前属于 INTERNAL 类型参数，用户无法对其进行设置。

取值范围：字符串

默认值：无

2.4.3.22 query_cache_type

参数说明：该参数目前为只读参数，且未实现其具体意义。参数值为整数类型，表示查询缓存的类型。

该参数目前属于 INTERNAL 类型参数，用户无法对其进行设置。

取值范围：[0, 2]

默认值：0

2.4.3.23 time_zone

参数说明：该参数目前为只读参数，且未实现其具体意义。参数值为字符串类型，表示当前时区。

该参数目前属于 INTERNAL 类型参数，用户无法对其进行设置。

取值范围：[-12:59, +13:00]

默认值：SYSTEM

2.4.3.24 wait_timeout

参数说明：该参数目前为只读参数，且未实现其具体意义。参数值为字符串类型，表示非交互式连接在持续无活动该秒数后，服务端会将其关闭。

该参数目前属于 INTERNAL 类型参数，用户无法对其进行设置。

取值范围：[1, 31536000]

默认值：28800

2.4.3.25 dolphin.lower_case_table_names

参数说明：该参数用于控制用户名、表名、视图名、模式名的大小写敏感；为 0 时大小写敏感，>0 时为大小写不敏感

该参数属于 USERSET 类型参数，请参考表 1 中对应设置方法进行设置。

取值范围：[0, 2]

默认值：1

2.4.3.26 dolphin.default_database_name

参数说明：dolphin 协议插件默认使用的 GBase 8c 数据库实例名称

该参数属于 SIGHUP 类型参数，请参考表 1 中对应设置方法进行设置。

须知：

当加载了 dophin 插件，并且开启了 dolphin 数据库协议后，可以使用此功能。

由于 GBase 8c 的 database 同 mysql 的 database 体系不一致，因此 dophin 需要选择一个 GBase 8c 的数据库实例。

取值范围：字符串

默认值：加载 dolphin 协议插件时，当前会话的 database_name

2.4.3.27 dolphin.optimizer_switch

参数说明：控制优化器行为，该参数是一系列控制选项的集合。当前支持的控制选项如下：

选项名	默认	功能
use_invisible_index	off	控制是否使用不可见索引

该参数属于 USERSET 类型参数。

取值范围：字符串

有效取值：各选项以逗号隔开，如下。

optimizer_switch='cgbaseand[,cgbaseand]…'

cgbaseand	描述
default	将所有控制选项设为其默认值
opt_name = default	将指定控制选项设为其默认值
opt_name = off	将指定控制选项设为关闭
opt_name = on	将指定控制选项设为打开

默认值：default

示例：

```
-- 设置 use_invisible_index 为 on
postgres=# set dolphin.optimizer_switch = 'use_invisible_index = on';

-- 设置 dolphin.optimizer_switch 为 defalut，表示所有控制选项都设置为默认值
postgres=# set dolphin.optimizer_switch = 'default';
```

```
-- 表示仅设置 use_invisible_index 选项为默认值
postgres=# set dolphin.optimizer_switch = 'use_invisible_index = default';
```

2.4.3.28 dolphin.dolphin.div_precision_increment

参数说明：此变量指定使用/运算符执行除法运算的结果的小数位数。

该参数用于提供除法运算结果的小数位数，在通过该参数以及入参计算出小数位数之后，与 GBase 8c 原生计算的小数位数进行对比，获取两者中的较大值作为计算结果的小数位数。大多数情况下 GBase 8c 的小数位数会比 MySQL 高，所以该参数在设置的值不大的情况下效果暂时不明显。

取值范围：[0, 30]

默认值：4

2.4.3.29 sql_note

参数说明：设置 show warnings 是否显示 Note 级别的信息开关。

取值范围：布尔型

默认值：ON

示例：

```
postgres=# show sql_note;
 sql_note
----------
 on
(1 row)
```

2.4.4 存储过程

2.4.4.1 基本语句

2.4.4.1.1 赋值语句

注意事项

相比于原始的 GBase8c，dolphin 对于赋值语法的修改为：

增加在 begin…end 之间可以通过 set 对变量进行赋值的语法功能。

B 模式下支持：

```
setvariable_name:=value;
```

对以上语法格式的解释如下：

variable_name：变量名。

value：可以是值或表达式。值 value 的类型需要和变量 variable_name 的类型兼容才能正确赋值。

示例：

```
DECLARE
emp_idINTEGER:=7788;--赋值
BEGIN
emp_id:=5;--赋值
emp_id:=5*7784;
END;
/
B 模式下:
DECLARE
emp_idINTEGER:=7788;--赋值
BEGIN
setemp_id:=5;--赋值
setemp_id:=5*7784;
END;
/
```

须知：

在 begin…end 之间可以通过 setvariable_name:=(=)value 来对变量进行赋值。

2.4.5 标识符说明

2.4.5.1 列名标识符

注意事项

相比于原始的 GBase8c，dolphin 对于列名标识符的修改为：

列名及别名信息敏感存储、显示，不考虑是否使用双引号包围列名标识符。

列名及别名信息不敏感的比较，即列名“aAa”和“AAa”标识相同列。

示例：

```
create database col_name dbcompatibility 'B';
CREATED ATABASE
```

```
\ccol_name
col_name=#CREATE TABLE t1(aAaint);
CREATE TABLE
col_name=#INSERT INTO t1 values(1);
INSERT01
col_name=#SELECT * FROM t1;
aAa
-----
1
(1row)
col_name=#select "AAa" from t1;
AAa
-----
1
(1row)
col_name=#select aaaASAaA from t1;
AaA
-----
1
(1row)
```

3 PostGIS Extension

3.1 PostGIS 概述

GBase 8c 提供 PostGIS ExtensionReference（版本为 PostGIS-2.4.2）。PostGIS ExtensionReference 是 PostgreSQL 的空间数据库扩展，提供如下空间信息服务功能：空间对象、空间索引、空间操作函数和空间操作符。PostGIS ExtensionReference 完全遵循 OpenGIS 规范。

PostGIS ExtensionReference 依赖第三方开源软件如下：

- Geos 3.6.2
- Proj 4.9.2
- Json 0.12.1
- Libxml2 2.7.1
- Gdal 1.11.0

3.2 PostGIS 安装

GBase 8c 安装包内集成 PostGIS 插件，但须手动安装加载。

(1) 使用 OM 工具安装 GBase8c，详见《GBase8cV5_5.0.0_安装部署手册》。

(2) 创建 postgis 插件。

3.3 PostGIS 使用

- 创建 Extension

```
CREATE Extension Reference postgis;
```

- 使用 Extension

PostGIS Extension 函数调用格式为：

```
SELECT GisFunction (Param1, Param2,.....);
```

其中 GisFunction 为函数名，Param1、Param2 等为函数参数名。下列 SQL 语句展示 PostGIS 的简单使用。具体请参考 https://download.osgeo.org/postgis/docs/postgis-2.4.2.pdf。

示例 1：几何表的创建。

```
CREATE TABLE cities ( id integer, city_name varchar(50) );
SELECT AddGeometryColumn('cities', 'position', 4326, 'POINT', 2);
```

示例 2：几何数据的插入。

```
INSERT INTO cities (id, position, city_name) VALUES
(1,ST_GeomFromText('POINT(-9.5 23)',4326),'CityA');
INSERT INTO cities (id, position, city_name) VALUES
(2,ST_GeomFromText('POINT(-10.6 40.3)',4326),'CityB');
INSERT INTO cities (id, position, city_name) VALUES
(3,ST_GeomFromText('POINT(20.8 30.3)',4326), 'CityC');
```

示例 3：计算三个城市间任意两个城市距离。

```
SELECT p1.city_name,p2.city_name,ST_Distance(p1.position,p2.position) FROM
cities AS p1, cities AS p2 WHERE p1.id > p2.id;
```

- 删除 ExtensionReference

在 GBase 8c 中删除 PostGIS ExtensionReference 的方法如下所示：

```
DROP ExtensionReference postgis [CASCADE];
```

说明

- 如果 Extension 被其它对象依赖(如创建的几何表)，需要加入 CASCADE(级联)关键字，删除所有依赖对象。

若要完全删除 PostGIS，则需由 gbase 用户使用 gs_om 工具移除 PostGIS 及 其依赖的动态链接库，格式如下：

```
gs_om -t postgis -m rmlib
```

3.4 PostGIS 支持和限制

支持数据类型

GBase 8c 的 PostGIS Extension 支持如下数据类型：

- box2d
- box3d
- geometry_dump
- geometry
- geography

支持的操作符和函数列表

表 2-1 PostGIS Extension 支持的操作符和函数列表

函数分类	包含函数
Management Functions	AddGeometryColumn、DropGeometryColumn、DropGeometryTable、PostGIS_Full_Version、PostGIS_GEOS_Version、PostGIS_Liblwgeom_Version、PostGIS_Lib_Build_Date、PostGIS_Lib_Version、PostGIS_PROJ_Version、PostGIS_Scripts_Build_Date、PostGIS_Scripts_Installed、PostGIS_Version、PostGIS_LibXML_Version、PostGIS_Scripts_Released、Populate_Geometry_Columns 、UpdateGeometrySRID
Geometry Constructors	ST_BdPolyFromText 、 ST_BdMPolyFromText 、 ST_Box2dFromGeoHash 、 ST_GeogFromText 、 ST_GeographyFromText 、 ST_GeogFromWKB 、 ST_GeomCollFromText 、 ST_GeomFromEWKB 、 ST_GeomFromEWKT 、 ST_GeometryFromText 、 ST_GeomFromGeoHash 、 ST_GeomFromGML 、 ST_GeomFromGeoJSON 、 ST_GeomFromKML 、 ST_GMLToSQL 、 ST_GeomFromText 、 ST_GeomFromWKB 、 ST_LineFrgbaseultiPoint 、 ST_LineFromText 、 ST_LineFromWKB 、 ST_LinestringFromWKB 、 ST_MakeBox2D 、 ST_3DMakeBox 、 ST_MakeEnvelope 、 ST_MakePolygon 、 ST_MakePoint 、 ST_MakePointM 、 ST_MLineFromText 、 ST_MPointFromText 、 ST_MPolyFromText 、 ST_Point 、 ST_PointFromGeoHash 、 ST_PointFromText 、 ST_PointFromWKB 、 ST_Polygon 、 ST_PolygonFromText 、 ST_WKBToSQL、ST_WKTToSQL

函数分类	包含函数
Geometry Accessors	GeometryType 、 ST_Boundary 、 ST_CoordDim 、 ST_Dimension 、 ST_EndPoint 、 ST_Envelope 、 ST_ExteriorRing 、 ST_GeometryN 、 ST_GeometryType 、 ST_InteriorRingN 、 ST_IsClosed 、 ST_IsCollection 、 ST_IsEmpty 、 ST_IsRing 、 ST_IsSimple 、 ST_IsValid 、 ST_IsValidReason 、 ST_IsValidDetail 、 ST_M 、 ST_NDims 、 ST_NPoints 、 ST_NRings 、 ST_NumGeometries 、 ST_NumInteriorRings 、 ST_NumInteriorRing 、 ST_NumPatches 、 ST_NumPoints 、 ST_PatchN 、 ST_PointN 、 ST_SRID 、 ST_StartPoint 、 ST_Summary 、 ST_X 、 ST_XMax 、 ST_XMin 、 ST_Y 、 ST_YMax 、 ST_YMin、ST_Z、ST_ZMax、ST_Zmflag、ST_ZMin
Geometry Editors	ST_AddPoint 、 ST_Affine 、 ST_Force2D 、 ST_Force3D 、 ST_Force3DZ 、 ST_Force3DM 、 ST_Force4D 、 ST_ForceCollection 、 ST_ForceSFS 、 ST_ForceRHR 、 ST_LineMerge 、 ST_CollectionExtract 、 ST_CollectionHomogenize 、 ST_Multi 、 ST_RemovePoint 、 ST_Reverse 、 ST_Rotate 、 ST_RotateX 、 ST_RotateY 、 ST_RotateZ 、 ST_Scale 、 ST_Segmentize 、 ST_SetPoint 、 ST_SetSRID 、 ST_SnapToGrid 、 ST_Snap 、 ST_Transform、ST_Translate、ST_TransScale
Geometry Outputs	ST_AsBinary 、 ST_AsEWKB 、 ST_AsEWKT 、 ST_AsGeoJSON 、 ST_AsGML 、 ST_AsHEXEWKB 、 ST_AsKML 、 ST_AsLatLonText 、 ST_AsSVG、ST_AsText、ST_AsX3D、ST_GeoHash
Operators	&&、&&&、&<、&<|、&>、 <<、 <<|、 =、 >>、@ 、 |&> 、 |>>、 ~、 ~=、 ↔ 、 <#>

函数分类	包含函数
Spatial Relationships and Measurements	ST_3DClosestPoint 、 ST_3DDistance 、 ST_3DDWithin 、 ST_3DDFullyWithin 、 ST_3DIntersects 、 ST_3DLongestLine 、 ST_3DMaxDistance 、 ST_3DShortestLine 、 ST_Area 、 ST_Azimuth 、 ST_Centroid 、 ST_ClosestPoint 、 ST_Contains 、 ST_ContainsProperly 、 ST_Covers 、 ST_CoveredBy 、 ST_Crosses 、 ST_LineCrossingDirection 、 ST_Disjoint 、 ST_Distance 、 ST_HausdorffDistance 、 ST_MaxDistance 、 ST_DistanceSphere 、 ST_DistanceSpheroid 、 ST_DFullyWithin 、 ST_DWithin 、 ST_Equals 、 ST_HasArc 、 ST_Intersects 、 ST_Length 、 ST_Length2D 、 ST_3DLength 、 ST_Length_Spheroid 、 ST_Length2D_Spheroid 、 ST_3DLength_Spheroid 、 ST_LongestLine 、 ST_OrderingEquals 、 ST_Overlaps 、 ST_Perimeter 、 ST_Perimeter2D 、 ST_3DPerimeter 、 ST_PointOnSurface 、 ST_Project 、 ST_Relate 、 ST_RelateMatch 、 ST_ShortestLine、ST_Touches、ST_Within
Geometry Processing	ST_Buffer 、 ST_BuildArea 、 ST_Collect 、 ST_ConcaveHull 、 ST_ConvexHull 、 ST_CurveToLine 、 ST_DelaunayTriangles 、 ST_Difference 、 ST_Dump 、 ST_DumpPoints 、 ST_DumpRings 、 ST_FlipCoordinates 、 ST_Intersection 、 ST_LineToCurve 、 ST_MakeValid 、 ST_MemUnion 、 ST_MinimumBoundingCircle 、 ST_Polygonize 、 ST_Node 、 ST_OffsetCurve 、 ST_RemoveRepeatedPoints 、 ST_SharedPaths 、 ST_Shift_Longitude、ST_Simplify、ST_SimplifyPreserveTopology、ST_Split、 ST_SymDifference、ST_Union、ST_UnaryUnion
Linear Referencing	ST_LineInterpolatePoint 、 ST_LineLocatePoint 、 ST_LineSubstring 、 ST_LocateAlong 、 ST_LocateBetween 、 ST_LocateBetweenElevations 、 ST_InterpolatePoint 、 ST_AddMeasure
Miscellaneous Functions	ST_Accum、 Box2D、 Box3D、ST_Expand、ST_Extent、 ST_3Dextent、 Find_SRID、ST_MemSize

函数分类	包含函数
Exceptional Functions	PostGIS_AddBBox、 PostGIS_DropBBox、 PostGIS_HasBBox

空间索引

GBase 8c 数据库的 PostGIS ExtensionReference 支持 GIST (Generalized Search Tree) 空间索引(分区表除外)。相比于 B-tree 索引， GIST 索引适应于任意类型的非常规数据结构，可有效 提高几何和地理数据信息的检索效率。

使用如下命令创建 GIST 索引:

```
CREATE INDEX indexname ON tablename USING GIST ( geometryfield );
```

扩展限制

只支持行存表。

不支持拓扑对象管理模块 Topology 和栅格数据处理模块 Raster。

不支持 BRIN 索引。

spatial_ ref_sys 表在扩容期间只支持查询操作。

4 assessment

4.1 概述

提供 assessment 工具来评估数据 SQL 文本在 GBase 8c 中的兼容性。支持从多个场景下获取 sql 语句进行兼容性评估，并输出评估报告。

支持从 mysql 端采集 sql 语句，支持从 mysql 端采集以下几种 sql 类型：

- 对象定义语句(ddl)。
- 慢 sql 语句。此类 sql 通过查询 mysql.slow_log 表获得，需要用户开启慢 sql 记录至该表中。
- 全量 sql 语句。此类 sql 通过查询 mysql.general_log 表获得，同样需要用户开启 general log 记录至该表中。

包含但不限于以下限制：

- 仅支持 SQL 文本文件输入，且 SQL 之间以;分割。
- 不使用 dolphin、whale 等兼容性插件场景，不兼容语句的报错信息可能不准确。如果使用对应插件，需遵循插件使用约束。
- 暂不支持#注释，请将文本内的#注释替换为--注释或直接删除。
- 存储过程、函数语句仅支持：创建体的合法性校验和函数体的语法兼容校验。
- 对于评估结果的准确率：
  - 完全兼容：完全支持该语法。兼容结果可能依赖于传入 SQL 语句的前置执行结果，因此实际在执行时不一定完全兼容。
  - 语法兼容：支持该语法，但是实际使用过程中可能包含字段类型不支持、函数不存在等问题。
  - 语句不兼容：不支持该语法。
  - 不支持评估：未考虑的语句。后续会陆续支持语句评估（例如 create database 等跨数据库影响语句）。
  - 忽略语句：注释等。

对于 A 兼容数据库，在作 SQL 语句导出时，最好提前作如下设置：

```
EXECUTE
DBMS_METADATA.SET_TRANSFORM_PARAM(DBMS_METADATA.SESSION_TRANSFORM,'SEGMENT_ATTRIBUTES',false);
EXECUTE
DBMS_METADATA.SET_TRANSFORM_PARAM(DBMS_METADATA.SESSION_TRANSFORM,'SQLTERMINATOR',true);
EXECUTE
DBMS_METADATA.SET_TRANSFORM_PARAM(DBMS_METADATA.SESSION_TRANSFORM,'STORAGE',false);
EXECUTE
DBMS_METADATA.SET_TRANSFORM_PARAM(DBMS_METADATA.SESSION_TRANSFORM,'TABLESPACE',false);
```

4.2 运行

确保有一个正在运行的数据库，且当前支持通过 gsql 连接

运行命令如下：

```
assessment_database[args]
```

参数	描述	使用	示例
连接参数	p	端口（必选）	-p 15400
	d	数据库（可选）	-d evaluation
	U	用户名（可选），如果支持本地连接，可不填	-U user
	W	密码（可选），如果支持本地连接，可不填	-W ******
兼容性评估	c	指定兼容类型（A\B\C\PG)，如果指定 d 参数，该参数不可设置。	-c B
文件参数	f	评估 SQL 文件（必填）	-f intput.sql
	o	输出文件（必填），一般输入 html 结尾文	-o result.html

		件	

4.3 示例

示例一：通过 gs_initdb 初始化数据库并启动，假设启动端口为 15400。此时可以通过 gsql -dpostgres -p15400 方式直接连接数据库。假设输入文件为 test.sql，输出报告路径为 result.html，评估的源数据库类型为 B，则评估使用的命令为：

```
assessment_database -p15400 -cB -ftest.sql -oresult.html
```

此时回显信息为：

```
assessment_database: create database "assessment_197561" automatically.
assessment_database: Create plugin[dolphin] automatically.
assessment_database: Create extension[assessment] automatically.
assessment_database: 解析[100.00%]:35/35
assessment_database: Create database assessment_197561 automatically, clear it
manually!
```

示例二：假设远程已经有数据库节点，在兼容性评估节点可以通过 gsql -dpostgres -p15400 -h127.0.0.2 -Utest -W***** 接数据库。假设输入文件为 test.sql，输出报告路径为 result.html，评估的源数据库类型为 B，则评估使用的命令为：

```
assessment_database -p15400 -cB -h127.0.0.2 -Utest -W***** -ftest.sql
-oresult.html
```

示例三：假设远程已经有数据库节点，且自己创建了 evaluation 数据库用于兼容性评估。在兼容性评估节点可以通过 gsql -devalution -p15400 -h127.0.0.2 -Utest -W*****连接数据库。期望通过假设输入文件为 test.sql，输出报告路径为 result.html，则评估使用的命令为：

```
assessment_database -p15400 -devaluation -h127.0.0.2 -Utest -W***** -ftest.sql
-oresult.html
```

即将 case 2 中的-cB 换成-devaluation 指定数据库即可。

结果

评估工具最终会生成评估报告，以 html 形式展示。包含如下内容：语句、兼容类型、失败原因。兼容类型包括：语法兼容、完全兼容、语法不兼容、不支持评估。

5 pg_trgm

pg_trgm 插件提供函数和操作符，用以测定字母数字文本基于三元模型匹配的相似性，还有支持快速搜索相似字符串的索引操作符类。

在 GBase 8c 中，插件已内部集成，如需启用执行命令：

```
create extension pg_trgm;
```

如需停止使用请执行命令：

```
drop extension pg_trgm;
```

三元模型概念

三元模型是一组从一个字符串中获得的三个连续的字符。 我们可以通过计数两个字符串共享的三元模型的数量来测量它们的相似性。 这个简单的想法证明在测量许多自然语言词汇的相似性时是非常有效的。

注意

- pg_trgm 从一个字符串提取三元模型时忽略非文字字符（非字母）。当确定包含在字符串中的三元模型集合时，每个单词被认为有两个空格前缀和一个空格后缀。例如，字符串"cat"中的三元模型的集合是"c"，"ca"，"cat"和"at"。 字符串"foo|bar"中的三元模型的集合是"f"，"fo"，"foo"，"oo"，"b"，"ba"，"bar"和"ar"。

5.1 函数和操作符

pg_trgm 模块提供的函数如表 3-1 所示，提供的操作符如表 3-2 所示。

表 2-1 pg_trgm 函数

函数	返回	描述
similarity(text, text)	real	返回一个数字表明两个参数是多么相似。结果的范围是 0（表明两个字符串完全不相似） 到 1（表明两个字符串是完全相同的）。
show_trgm(text)	text[]	返回一个给定字符串中所有三元模型的数组。（实际上这个除了调试之外很少有用。）
show_limit()	real	返回%操作符使用的当前相似性阈值。例如，这个设置两个单词间的最小相似性， 这两个单词被认为足够相似以致相互之间拼写错误。
set_limit(real)	real	设置%操作符使用的当前相似性阈值。该阈值必须在 0 和 1 之间（缺省是 0.3）。 返回传递进来的相同的值。

表 2-2 pg_trgm 函数

操作符	返回	描述
text%text	boolean	如果它的参数的相似性高于 set_limit 设置的当前相似性阈值则返回 true。
text<->text	real	返回参数之间的"距离"，这是 1 减去 similarity()值。

5.2 索引支持

pg_trgm 模块提供 GiST 和 GIN 索引操作符类。为了相似性搜索高效，该操作符类允在文本字段上创建一个索引。这些索引类型支持上面描述的相似性操作符，并且额外支持基于三元模型的索引搜索：LIKE，ILIKE，~和~*查询。

示例：

```
CREATE TABLE test_trgm (t text);
CREATE INDEX trgm_idx ON test_trgm USING gist (t gist_trgm_ops);
```

或

```
CREATE INDEX trgm_idx ON test_trgm USING gin (t gin_trgm_ops);
```

可以用 t 列来做相似性搜索：

```
SELECT t, similarity(t, 'word') AS sml
  FROM test_trgm
  WHERE t % 'word'
  ORDER BY sml DESC, t;
```

这将返回文本列上与“word”足够相似的所有值，从最佳匹配到最坏匹配排序。 该索引将用来做一个快速操作，即使是在非常大的数据集上面。

上面查询的一个变体是：

```
SELECT t, t <-> 'word' AS dist
  FROM test_trgm
  ORDER BY dist LIMIT 10;
```

这个可以通过 GiST 索引更有效的实现，而不是 GIN 索引。当只想要少量最靠近的匹配时， 通常会使用第一个公式。

这些索引类型也支持 LIKE 和 ILIKE 索引搜索，例如：

```
SELECT * FROM test_trgm WHERE t LIKE '%foo%bar';
```

该索引搜索通过从搜索字符串中提取三元模型然后在索引中查找这些三元模型来工作。搜索字符串中的三元模型越多，索引搜索越有效率。不像基于 B-tree 的搜索， 搜索字符串不需要是左边固定的。

支持正则表达式匹配的索引搜索 （~和~*操作符），例如：

```
SELECT * FROM test_trgm WHERE t ~ '(foo|bar)';
```

该索引搜索通过从正则表达式中提取三元模型然后在索引中查找这些三元模型来工作。从正则表达式中提取出来的三元模型越多，索引搜索越有效率。不像基于 B-tree 的搜索， 搜索字符串不需要是左边固定的。

对于 LIKE 和正则表达式搜索，请记住，没有可提取的三元模型将降级为全文索引搜索。

一般来说，GIN 索引比 GiST 索引搜索起来要快，但是在建立或更新时要慢；索引 GIN 更适合于静态数据而 GiST 适合于经常更新的数据。请根据实际情况来选择索引类型。

5.3 文本搜索集成

三元模型匹配在用于与全文本索引相协调时是一个非常有用的工具。 尤其是它可以帮助识别错误拼写的输入单词，这样的单词将不能直接通过全文本搜索机制匹配。

第一步是要生成一个包含文档中的所有唯一词的辅助表：

```
CREATE TABLE words AS SELECT word FROM
        ts_stat('SELECT to_tsvector(''simple'', bodytext) FROM documents');
```

这里的 documents 是含有我们希望搜索的文本字段 bodytext 的表。 使用 simple 配置 to_tsvector 函数的原因， 不是使用一个语言特定的配置，是我们想要一个原始（未修改）词的列表。

下一步，在该词的列上创建一个三元模型索引：

```
CREATE INDEX words_idx ON words USING gin(word gin_trgm_ops);
```

现在，一个类似于之前示例的 SELECT 查询可以用来在用户搜索词中建议错误拼写的词的拼写。 需要一个有用的额外的文本，选中的词和错误拼写的词有相似的长度。

注意

- 因为上述 words 表是作为一个单独的、静态表生成的，需要定期重新生成，这样才能与文档集合保持合理的更新。通常不需要实时更新。

6 pgcrypto

pgcrypto 插件提供的加密函数能够实现服务器端的数据加密，以便在 SQL 语句中调用这些函数来完成数据的加密。

在 GBase 8c 中，插件已内部集成，如需启用执行命令：

```
create extension pgcrypto;
```

如需停止使用请执行命令：

```
drop extension pgcrypto;
```

6.1 普通哈希函数

6.1.1 digest()

digest(data text|bytea, type text) 返回 bytea

计算一个给定 data 的一个二进制哈希值。type 是要使用的算法。标准算法是 md5、sha1、sha224、sha256、sha384 和 sha512。如果使用 OpenSSL 编译了 pgcrypto，如表 4-4 中所述，有更多算法可用。

如果你想摘要成为一个十六进制字符串，可以在结果上使用 encode()。例如：

```
CREATE OR REPLACE FUNCTION sha1(bytea) returns text AS $$
    SELECT encode(digest($1, 'sha1'), 'hex')
$$ LANGUAGE SQL STRICT IMMUTABLE;
```

6.1.2 hmac()

hmac(data text|bytea, key text, type text) 返回 bytea

为带有密钥 key 的 data 计算哈希过的 MAC。type 与 digest()中相同。

这与 digest()相似，但是该哈希只能在知晓密钥的情况下被重新计算出来。这阻止了某人修改数据且还想更改哈希以匹配之的企图。

如果该密钥大于哈希块的尺寸，它将先被哈希然后把结果用作密钥。

6.2 口令哈希函数

函数 crypt()和 gen_salt()是特别设计用来做口令哈希的。crypt()完成哈希，而 gen_salt()负责为前者准备算法参数。

crypt()中的算法在以下方面不同于通常的 MD5 或 SHA1 哈希算法：

- 速度慢。由于数据量很小，这是增加蛮力口令破解难度的唯一方法。
- 使用一个随机值（称为 salt），这样具有相同口令的用户将得到不同的密文口令。这也是针对逆转算法的一种额外保护。
- 会在结果中包括算法类型，这样用不同算法哈希的口令能共存。
- 其中一些是自适应的 — 这意味着当计算机变得更快时，你可以调整该算法变得更慢，而不会产生与现有口令的不兼容。

下表列出了 crypt()函数所支持的算法。

表 4-1 crypt()支持的算法

算法	最大口令长度	自适应？	Salt 位数	输出长度	描述
bf	72	yes	128	60	基于 Blowfish，变体 2a
md5	unlimited	no	48	34	基于 MD5 的加密
xdes	8	yes	24	20	扩展的 DES
des	8	no	12	13	原生 UNIX 加密

6.2.1 crypt()

crypt(password text, salt text) 返回 text

计算 password 的一个 crypt(3) 风格的哈希。在存储一个新口令时，你需要使用 gen_salt()产生一个新的 salt 值。要检查一个口令，把存储的哈希值作为 salt，并且测试结果是否匹配存储的值。

设置一个新口令的例子：

```
UPDATE ... SET pswhash = crypt('new password', gen_salt('md5'));
```

认证的例子：

```
SELECT (pswhash = crypt('entered password', pswhash)) AS pswmatch FROM ... ;
```

如果输入的口令正确，这会返回 true。

6.2.2 gen_salt()

gen_salt(type text [, iter_count integer]) 返回 text

产生一个在 crypt()中使用的新随机 salt 字符串。该 salt 字符串也告诉了 crypt()要使用哪种算法。

type 参数指定哈希算法。可接受的类型是：des、xdes、md5 以及 bf。

iter_count 参数让用户可以为使用迭代计数的算法指定迭代计数。计数越高，哈希口令花的时间更长并且因而需要更多时间去攻破它。不过使用太高的计数会导致计算一个哈希的时间高达数年 — 这并不使用。如果 iter_count 参数被忽略，将使用默认的迭代计数。允许的 iter_count 值与算法相关，如表 4-2 所示。

表 4-2 crypt()的迭代计数

算法	默认值	最小值
xdes	725	1
bf	6	4

对 xdes 算法还有额外的限制：迭代计数必须是一个奇数。

要选取一个合适的迭代计数，考虑最初的 DES 加密被设计成在当时的硬件上每秒钟完成 4 次哈希。低于每秒 4 次哈希的速度很可能会损害可用性。而超过每秒 100 次哈希又可能太快了。

表 4-3 给出了不同哈希算法的相对慢度的综述。该表展示了在假设口令只含有小写字母或者大小写字母及数字的情况下，在一个 8 字符口令中尝试所有字符组合所需要的时间。在 crypt-bf 项中，在一个斜线之后的数字是 gen_salt 的 iter_count 参数

表 4-3 哈希算法速度

算法	次哈希/秒	对于[a-z]	对于[A-Za-z0-9]	相对于 md5 hash 的持续时间
crypt-bf/8	1792	4 年	3927 年	100k
crypt-bf/7	3648	2 年	1929 年	50k
crypt-bf/6	7168	1 年	982 年	25k

crypt-bf/5	13504	188 天	521 年	12.5k
crypt-md5	171584	15 天	41 年	1k
crypt-des	23221568	157.5 分	108 天	7
sha1	37774272	90 分	68 天	4
md5(hash)	150085504	22.5 分	17 天	1

6.3 加密函数

一个加密的 PGP 消息由两个部分或者包组成：

- 包含一个会话密钥的包 — 加密过的对称密钥或者公钥。
- 包含用会话密钥加密过的数据的包。

- 当用一个对称密钥（即一个口令）加密时：

给定的口令被使用一个 String2Key (S2K) 算法哈希。这更像 crypt()算法 — 有目的地慢并且使用随机 salt — 但是它会产生一个全长度的二进制密钥。

如果要求一个独立的会话密钥，将会生成一个新的随机密钥。否则该 S2K 密钥将被直接用作会话密钥。

如果直接使用 S2K 密钥，那么只有 S2K 设置将被放入会话密钥包中。否则会话密钥会用 S2K 密钥加密并且放入会话密钥包中。

- 当使用一个公共密钥加密时：

一个新的随机会话密钥会被生成。

它被用公共密钥加密并且放入到会话密钥包中。

在两种情况下，要被加密的数据按下列步骤被处理：

可选的数据操纵：压缩、转换成 UTF-8 或者行末转换。

数据会被加上一个随机字节的块作为前缀。这等效于使用一个随机 IV。

追加一个随机前缀和数据的 SHA1 哈希。

所有这些都用会话密钥加密并且放在数据包中。

6.3.1 pgp_sym_encrypt()

pgp_sym_encrypt(data text, psw text [, options text]) 返回 bytea

pgp_sym_encrypt_bytea(data bytea, psw text [, options text]) 返回 bytea

使用一个对称 PGP 密钥 psw 加密 data。options 参数可以包含下文所述的选项设置。

6.3.2 pgp_sym_decrypt()

pgp_sym_decrypt(msg bytea, psw text [, options text]) 返回 text

pgp_sym_decrypt_bytea(msg bytea, psw text [, options text]) 返回 bytea

解密一个用对称密钥加密过的 PGP 消息。

不允许使用 pgp_sym_decrypt 解密 bytea 数据。这是为了避免输出非法的字符数据。使用 pgp_sym_decrypt_bytea 解密原始文本数据是好的。

options 参数可以包含下文所述的选项设置。

6.3.3 pgp_pub_encrypt()

pgp_pub_encrypt(data text, key bytea [, options text]) 返回 bytea

pgp_pub_encrypt_bytea(data bytea, key bytea [, options text]) 返回 bytea

用一个公共 PGP 密钥 key 加密 data。给这个函数一个私钥会产生一个错误。

options 参数可以包含下文所述的选项设置。

6.3.4 pgp_pub_decrypt()

pgp_pub_decrypt(msg bytea, key bytea [, psw text [, options text]]) 返回 text

pgp_pub_decrypt_bytea(msg bytea, key bytea [, psw text [, options text]]) 返回 bytea

解密一个公共密钥加密的消息。key 必须是对应于用来加密的公钥的私钥。如果私钥是用口令保护的，你必须在 psw 中给出该口令。如果没有口令，但你想要指定选项，你需要给出一个空口令。

不允许使用 pgp_pub_decrypt 解密 bytea 数据。这是为了避免输出非法的字符数据。使用 pgp_pub_decrypt_bytea 解密原始文本数据是好的。

options 参数可以包含下文所述的选项设置。

6.3.5 pgp_key_id()

pgp_key_id(bytea) 返回 text

pgp_key_id 抽取一个 PGP 公钥或私钥的密钥 ID。或者如果给定了一个加密过的消息，它给出一个用来加密数据的密钥 ID。

它能够返回 2 个特殊密钥 ID：

SYMKEY

该消息是用一个对称密钥加密的。

ANYKEY

该消息是用公钥加密的，但是密钥 ID 已经被移除。这意味着你将需要尝试你所有的密钥来看看哪个能解密该消息。pgcrypto 本身不产生这样的消息。

注意不同的密钥可能具有相同的 ID。这很少见但是是一种正常事件。客户端应用则应该尝试用每一个去解密，看看哪个合适 — 像处理 ANYKEY 一样。

6.3.6 armor(),dearmor()

armor(data bytea [, keys text[], values text[]]) 返回 text

dearmor(data text) 返回 bytea

这些函数把二进制数据包装/解包成 PGP ASCII-armored 格式，其基本上是带有 CRC 和额外格式化的 Base64。

如果指定了 keys 和 values 数组，每一个 键/值对的 armored 格式上会增加一个 armor header。两个 数组都必须是单一维度的，并且它们的长度必须相同。键和值不能包含任何 非 ASCII 字符。

6.3.7 pgp_armor_headers

pgp_armor_headers(data text, key out text, value out text) returns setof record

pgp_armor_headers()从 data 中抽取 armor header。返回值是一个有两列的行集合，包括键和值。如果键或值 包含任何非-ASCII 字符，它们会被视作 UTF-8。

6.3.8 PGP 函数的选项

选项被命名为与 GnuPG 类似的形式。一个选项的值应该在一个等号后给出，各个选项之间用逗号分隔。例如：

```
pgp_sym_encrypt(data, psw, 'compress-algo=1, cipher-algo=aes256')
```

除了 convert-crlf 之外所有这些选项只适用于加密函数。解密函数会从 PGP 数据中得到

这些参数。

最有趣的选项可能是 compress-algo 和 unicode-mode。其余的应该可以使用合理的默认值。

6.3.8.1 cipher-algo

要用哪个密码算法。

值：bf,aes128,aes192,aes256(只用于 OpenSSL：3des,cast5)

默认：aes128

适用于：pgp_sym_encrypt,pgp_pub_encrypt

6.3.8.2 compress-algo

要使用哪种压缩算法。只有 PostgreSQL 编译时使用了 zlib 时才可用。

值：

0-不压缩

1-ZIP 压缩

2-ZLIB 压缩（=ZIP 外加元数据和块 CRC）

默认：0

适用于：pgp_sym_encrypt,pgp_pub_encrypt

6.3.8.3 compress-level

压缩多少。级别越高压缩得越小但是速度也越慢。0 表示禁用压缩。

值：0,1-9

默认：6

适用于：pgp_sym_encrypt,pgp_pub_encrypt

6.3.8.4 convert-crlf

加密时是否把\n 转换成\r\n 以及解密时是否把\r\n 转换成\n。RFC 4880 指定文本数据存储时应该使用\r\n 换行。使用这个选项能够得到完全 RFC 兼容的行为。

值：0,1

默认：0

适用于：pgp_sym_encrypt,pgp_pub_encrypt,pgp_sym_decrypt,pgp_pub_decrypt

6.3.8.5 disable-mdc

不用 SHA-1 保护数据。使用这个选项的唯一好的理由是实现与古董级别 PGP 产品的兼容，这些产品在受 SHA-1 保护的包被加入到 RFC 4880 之前就已经存在了。最近的 gnupg.org 和 pgp.com 软件能很好地支持它。

值：0，1

默认：0

适用于：pgp_sym_encrypt,pgp_pub_encrypt

6.3.8.6 sess-key

使用单独的会话密钥。公钥加密总是使用一个单独的会话密钥。这个选项是用于对称密钥加密的，对称密钥加密默认直接使用 S2K 密钥。

值：0，1

默认：0

适用于：pgp_sym_encrypt

6.3.8.7 s2k-mode

要使用哪一种 S2K 算法。

值：

0-不用 salt。危险！

1-1-用 salt 但是使用固定的迭代计数。

2-3-可变的迭代计数。

3-默认：3

4-适用于：pgp_sym_encrypt

6.3.8.8 s2k-count

S2K 算法要使用的迭代次数。它必须是一个位于 1024 和 65011712 之间的值， 首尾两个值包括在内。

默认：65536 和 253952 之间的一个随机值

适用于：pgp_sym_encrypt，只能用于 s2k-mode=3

6.3.8.9 s2k-digest-algo

要在 S2K 计算中使用哪种摘要算法。

值：md5,sha1

默认：sha1

适用于：pgp_sym_encrypt

6.3.8.10 s2k-cipher-algo

要用哪种密码来加密独立的会话密钥。

值：bf,aes,aes128,aes192,aes256

默认：usecipher-algo

适用于：pgp_sym_encrypt

6.3.8.11 unicode-mode

是否把文本数据在数据库内部编码和 UTF-8 之间来回转换。如果你的数据库已经是 UTF-8，将不会转换，但是消息将被标记为 UTF-8。没有这个选项它将不会被标记。

值：0，1

默认：0

适用于：pgp_sym_encrypt,pgp_pub_encrypt

6.3.8.12 用 GnuPG 生成 PGP 密钥

要生成一个新密钥：

```
gpg --gen-key
```

更好的密钥类型是“DSA 和 Elgamal”。

对于 RSA 密钥，你必须创建仅用于签名的 DSA 或 RSA 密钥作为主控密钥，然后用 gpg --edit-key 增加一个 RSA 加密子密钥。

要列举密钥：

```
gpg --list-secret-keys
```

要以 ASCII-保护格式导出一个公钥：

```
gpg -a --export KEYID > public.key
```

要以 ASCII-保护格式导出一个私钥：

```
gpg -a --export-secret-keys KEYID > secret.key
```

在把这些密钥交给 PGP 函数之前，你需要对它们使用 dearmor()。或者如果你能处理二进制数据，可以从命令中去掉-a。

6.3.8.13 PGP 代码的限制

不支持签名。这也意味着它不检查加密子密钥是否属于主控密钥。

不支持加密密钥作为主控密钥。由于通常并不鼓励那种用法，这应该不是问题。

不支持多个子密钥。这可能看起来像一个问题，因为在实践中普遍需要多个子密钥。在另一方面，你不能把你的常规 GPG/PGP 密钥用于 pgcrypto，而是创建一些新的密钥，因为使用场景相当不同。

6.4 F.26.4.原始的加密函数

这些函数只在数据上运行一次加密，不具有 PGP 加密的任何先进特性。因此这些函数有一些主要的问题：

- 直接把用户密钥用作加密密钥。
- 不提供任何完整性检查来查看被加密数据是否被修改。
- 希望用户自己管理所有加密参数，甚至是 IV。
- 无法处理文本。

因此，在介绍了 PGP 加密后，我们不鼓励使用原始的加密函数。

```
encrypt(data bytea, key bytea, type text) 返回 bytea
decrypt(data bytea, key bytea, type text) 返回 bytea

encrypt_iv(data bytea, key bytea, iv bytea, type text) 返回 bytea
decrypt_iv(data bytea, key bytea, iv bytea, type text) 返回 bytea
```

使用 type 指定的密码方法加密/解密数据。type 字符串的语法是：

```
algorithm [ - mode ] [ /pad: padding ]
```

其中 algorithm 是下列之一：

bf— Blowfish

aes— AES (Rijndael-128)

并且 mode 是下列之一：

cbc— 下一个块依赖前一个（默认）

ecb— 每一个块被独立加密（只用于测试）

并且 padding 是下列之一：

pkcs— 数据可以是任意长度（默认）

none— 数据必须是密码块尺寸的倍数

因此，例如这些是等效的：

```
encrypt(data, 'fooz', 'bf')
encrypt(data, 'fooz', 'bf-cbc/pad:pkcs')
```

在 encrypt_iv 和 decrypt_iv 中，iv 参数是 CBC 模式的初始值，ECB 会忽略它。如果不是准确的块尺寸，它会被修剪或用零填充。在没有这个参数的函数中，它的值都被默认为零。

6.5 随机数据函数

```
gen_random_bytes(count integer) 返回 bytea
```

返回 count 个密码上强壮的随机字节。一次最多可以抽取 1024 个字节。这是为了避免耗尽随机数发生池。

```
gen_random_uuid() 返回 uuid
```

返回一个版本 4 的（随机的）UUID。

6.6 注解

6.6.1 配置

pgcrypto 会根据查找主 PostgreSQLconfigure 脚本配置自身。影响选项是--with-zlib 以及--with-openssl。

在编译了 zlib 时，PGP 加密函数能够在加密前压缩数据。

在编译了 OpenSSL 时，会有更多可用算法。公钥加密函数也会更快，因为 OpenSSL 有优化得更好的 BIGNUM 函数。

表 4-4 使用和不用 OpenSSL 的功能总结

功能	内建	使用 OpenSSL
MD5	yes	yes
SHA1	yes	yes
SHA224/256/384/512	yes	yes
其他摘要算法	no	yes (注意 1)
Blowfish	yes	yes
AES	yes	yes
DES/3DES/CAST5	no	yes
原始加密	yes	yes
PGP 对称加密	yes	yes
PGP 公钥加密	yes	yes

注意：

OpenSSL 支持的任何摘要算法都是自动选取的。这对于使用密码来说是不可能的，因为需要被显式地支持。

6.6.2 NULL 处理

按照 SQL 中的标准，只要任何参数是 NULL， 所有的函数都会返回 NULL。在不当使用时这可能会导致安全风险。

6.6.3 安全性限制

所有 pgcrypto 函数都在数据库服务器内部运行。这意味着在 pgcrypto 和客户端应用之间移动的所有数据和口令都是明文。因此必须：

- 本地连接或者使用 SSL 连接。
- 信任系统管理员和数据库管理员。
- 在客户端应用中进行加密。

7 pgvector

向量扩展可以为用户提供强大的向量功能，用于存储、查询和处理向量数据。具有以下特点；

直接集成：作为扩展添加到现有数据库中，方便用户获得适量数据库的功能，而无需重大系统变动；

支持多种向量距离：内置多种距离度量，包括欧几里得距离、余弦距离和曼哈顿距离，可以根据具体应用需求来定制相似性的搜索和分析；

索引支持：提供高效的索引选项，例如 k-最近邻搜索。即使数据集大小增长，用户也可以实现快速查询执行，并保持较高的搜索准确性；

易于查询语法访问：可以使用 SQL 查询语法进行向量操作。

8 uuid-ossp

该插件用于储存全局唯一标识符 UUID。

在 GBase 8c 中，插件已内部集成，如需启用执行命令：

```
create extension "uuid-ossp"
```

如需停止使用请执行命令：

```
drop extension "uuid-ossp"
```

表 5-1 UUID 生成函数

函数	描述
uuid_generate_v1()	这个函数生成版本 1 的 UUID。它的算法使用了计算机的 MAC 地址和时间戳。注意这种 UUID 泄露了生成它的计算机标识和生成它的时间，所以它可能不太适合对安全性要求较高的应用。
uuid_generate_v1mc()	这个函数生成一个版本 1 的 UUID，但是使用一个随机多播 MAC 地址而不是计算机的真实的 MAC 地址。
uuid_generate_v3(namespace uuid, name text)	这个函数使用给定的输入名字(name)在给定的命名空间(namespace)中生成一个版本 3 的 UUID。给定的命名空间应该是调用表 5-2 中的函数 uuid_ns_*()返回的常量。(理论上他可能是任何 UUID)参数 name 是一个选定命名空间

	(namespace)中的标识符。 例如: `SELECT uuid_generate_v3(uuid_ns_url(), 'http://www.postgresql.org');` 参数 name 会被使用 MD5 算法做哈希，所以从产生的 UUID 中不可能反向获得明文。利用这个方法生成的 UUID 不需要随机算法不依赖任何运行相关的环境因素，因此生成过程是可重复的。
uuid_generate_v4()	这个函数生成一个版本 4 的 UUID，它完全依靠随机数。
uuid_generate_v5(namespace uuid, name text)	这个函数生成一个版本 5 的 UUID，它个工作过程类似于版本 3 的 UUID，但是它使用的 SHA-1 的哈希算法。因为 SHA-1 算法被认为比 MD5 算法更安全，所有应该尽量使用版本 5 而不版本 3。

表 5-1 返回 UUID 常量的函数

uuid_nil()	一个"nil"UUID 常量，它不应该看作一个真正的 UUID。
uuid_ns_dns()	代表 DNS 命令空间的 UUID 常量。
uuid_ns_url()	代表 URL 命名空间的 UUID 常量。
uuid_ns_oid()	代表 ISO 对象标识符(OID)命名空间的 UUID 常量。(它是 ASN.1 的 OID，和 PostgreSQL 用的 OID 没有关系）。
uuid_ns_x500()	代表 X.500 识别名字(DN)命名空间的 UUID 常量

9 pgpool

Pgpool-II 是数据库客户端与服务器之间的代理软件，也就是说客户端通过 Pgpool 连接数据库。它提供以下功能：

- 连接池

与 PostgreSQL 服务器建立连接后会由 Pgpool-II 维护该连接，并当新连接连接时，如果存在具有相同连接信息（即用户名，数据库，协议版本和其他连接参数）的连接则直接使用它们。它减少了连接开销并改善了系统的整体吞吐量。

- 负载均衡

如果数据库是复制模式的（以复制模式或主/从模式运行），则在任何服务器上执行 SELECT 查询都将返回相同的结果。Pgpool-II 利用复制功能来减少每台 PostgreSQL 服务器上的负载。它通过将 SELECT 查询分配到可用的服务器之间来做到这一点，从而提高了系统的整体吞吐量。在理想情况下，读取性能可以与 PostgreSQL 服务器的数量成正比。在只读查询并发比较高的情况下，负载平衡效果最佳。

- 自动故障转移

如果其中一台数据库服务器出现故障或无法访问， Pgpool-II 会将其分离，并将继续使用其余的数据库服务器提供服务。提供了很多自动故障转移配置来使该功能到最优效果，例如包括超时和重试。

- 在线恢复

Pgpool-II 可以通过执行一个命令来执行数据库节点的在线恢复。当在线恢复与自动故障转移一起使用时，可以通过故障转移将失败的节点分离，同时使用在线恢复自动附加为备用节点。也可以同步附加新的 PostgreSQL 服务器节点。

- 复制

Pgpool-II 可以管理多台 PostgreSQL 服务器。复制功能可以在两台或者多台 PostgreSQL 服务器之间创建实时备份，因此，如果其中一台 PostgreSQL 服务器发生故障，服务可以继续运行而不会中断。Pgpool-II 具有内置复制（本机复制）。但是，用户可以使用外部复制功能，包括 PostgreSQL 的流复制（目前大多数的主备集群都采用 PostgreSQL 的流复制来实现）。

- 限制超出连接

PostgreSQL 的最大连接数是有限制的（postgres.conf 中 max_connections），达到此数量

时，新连接将被拒绝。但是，增加最大连接数会增加资源消耗，并对整体系统性能产生负面影响。Pgpool-II 对最大连接数也有限制，但是将对额外的连接进行排队，而不是立即返回错误。当然也可以配置为在超过连接限制（4.1 或更高版本）时返回错误。

- 看门狗

看门狗可以协调多个 Pgpool-II 节点，建立强壮的集群系统，并避免单点故障或脑裂。看门狗可以对其他 pgpool-II 节点执行生命检查，以检测 Pgpoll-II 的故障。如果活跃中的 Pgpool-II 节点发生故障，则备用 Pgpool-II 节点可以升级为活跃状态，变为主节点，并接管虚拟 IP。

- 内存查询缓存

内存查询缓存允许保存一对 SELECT 语句及其结果。如果出现相同的 SELECT，则 Pgpool-II 从缓存中返回该值。由于不涉及 SQL 解析或对 PostgreSQL 的访问，因此内存缓存将非常快。但另一方面，在某些情况下，它可能会比正常途径慢，因为它增加了存储缓存数据的开销。

10 wal2json

wal2json 是用于逻辑解码的输出插件。这意味着插件可以访问由 INSERT 和 UPDATE 生成的元组。此外，还可以根据配置的复制标识（REPLICA IDENTITY）访问更新/删除旧行版本。可以使用流协议（逻辑复制插槽）或专有的 SQL API。

配置

(1) 修改 postgresql.conf 参数，请修改以下参数：

```
gs_guc reload -N all -l all -c "wal_level=logical"
gs_guc reload -N all -l all -c "wal_sender_timeout=60s"
gs_guc reload -N all -l all -h "host  replication all  0.0.0.0/0 sha256"
```

并酌情修改 max_replication_slots 和 max_wal_senders 参数。

(2) 重启数据库：

```
gs_om -t restart
```

(3) 创建用户并退出数据库

```
gsql -d postgres -p 15400
postgres=# CREATE USER test SYSADMIN PASSWORD "Gbase,123";
postgres=# \q
```

示例

通过 SQL 调用函数消费更改：

```
postgres=# CREATE TABLE table2_with_pk (a SERIAL, b VARCHAR(30), c TIMESTAMP NOT
NULL, PRIMARY KEY(a, c));CREATE TABLE table2_without_pk (a SERIAL, b NUMERIC(5,2),
c TEXT);
CREATE TABLE
postgres=# CREATE TABLE table2_without_pk (a SERIAL, b NUMERIC(5,2), c TEXT);
CREATE TABLE
postgres=# SELECT 'init' FROM pg_create_logical_replication_slot('test_slot',
'wal2json');
 ?column?
----------
 init
(1 row)

postgres=# BEGIN;
BEGIN
```

```
postgres=# INSERT INTO table2_with_pk (b, c) VALUES('Backup and Restore', now());
INSERT 0 1
postgres=# INSERT INTO table2_with_pk (b, c) VALUES('Tuning', now());
INSERT 0 1
postgres=# INSERT INTO table2_with_pk (b, c) VALUES('Replication', now());
INSERT 0 1
postgres=# SELECT pg_logical_emit_message(true, 'wal2json', '此消息将被传递');
 pg_logical_emit_message
-------------------------
 81/C4F33608
(1 row)
postgres=# DELETE FROM table2_with_pk WHERE a < 3;
DELETE 2
postgres=# SELECT pg_logical_emit_message(false, 'wal2json', '即使回滚事务，也会传递此非事务性消息');
 pg_logical_emit_message
-------------------------
 81/C4F33738
(1 row)
postgres=# INSERT INTO table2_without_pk (b, c) VALUES(2.34, 'Tapir');
INSERT 0 1
--这条修改不会添加到流中，因为表没有主键或复制标识
postgres=# UPDATE table2_without_pk SET c = 'Anta' WHERE c = 'Tapir';
UPDATE 1
postgres=# COMMIT;
COMMIT

--通过 SQL 调用函数消费更改（使用格式版本 1）
postgres=# SELECT data FROM pg_logical_slot_get_changes('test_slot', NULL, NULL,
'pretty-print', '1', 'add-msg-prefixes', 'wal2json');
WARNING:  Due to DDL in the same top transaction, partial modification may be lost
for <XID, FIRST_LSN, FINAL_LSN, SKIP_START_LSN> = { <91352, 0/4AF3A8E0,
0/4AF41560, 0/4AF3A928> }
WARNING:  Due to DDL in the same top transaction, partial modification may be lost
for <XID, FIRST_LSN, FINAL_LSN, SKIP_START_LSN> = { <91353, 0/4AF41D40,
0/4AF4A538, 0/4AF41D88> }
                                                    data
--------------------------------------------------------------------------------
------------------------------
```

```
{
                                +
        "change":
[
           +
        ]
+
 }

{
                                +
        "change":
[
           +
        ]
+
 }

{
                                +
        "change":
[
           +

{
              +
                        "kind": "insert",
+
                        "schema": "public",
+
                        "table": "table2_with_pk",
+
                        "columnnames": ["a", "b", "c"],
+
                        "columntypes": ["integer", "character varying(30)",
"timestamp without time zone"],+
                        "columnvalues": [4, "Backup and Restore", "2024-03-07
15:49:06.185422"]            +
                 }
+
```

```
                ,{
                                +
                        "kind": "insert",
+
                        "schema": "public",
+
                        "table": "table2_with_pk",
+
                        "columnnames": ["a", "b", "c"],
+
                        "columntypes": ["integer", "character varying(30)",
"timestamp without time zone"],+
                        "columnvalues": [5, "Tuning", "2024-03-07
15:49:06.185422"]                       +
                }
+
                ,{
                                +
                        "kind": "insert",
+
                        "schema": "public",
+
                        "table": "table2_with_pk",
+
                        "columnnames": ["a", "b", "c"],
+
                        "columntypes": ["integer", "character varying(30)",
"timestamp without time zone"],+
                        "columnvalues": [6, "Replication", "2024-03-07
15:49:06.185422"]                       +
                }
+
                ,{
                                +
                        "kind": "insert",
+
                        "schema": "public",
+
                        "table": "table2_without_pk",
+
```

```
                        "columnnames": ["a", "b", "c"],
+
                        "columntypes": ["integer", "numeric(5,2)", "text"],
+
                        "columnvalues": [1, 2.34, "Tapir"]
+
                }
+
                ,{
                        +
                        "kind": "update",
+
                        "schema": "public",
+
                        "table": "table2_without_pk",
+
                        "columnnames": ["a", "b", "c"],
+
                        "columntypes": ["integer", "numeric(5,2)", "text"],
+
                        "columnvalues": [1, 2.34, "Anta"],
+
                        "oldkeys":
{                                                                       +
                                "keynames": ["a", "b", "c"],
+
                                "keytypes": ["integer", "numeric(5,2)",
"text"],                           +
                                "keyvalues": [1, 2.34, "Anta"]
+
                        }
+
                }
+
        ]
+
 }
(3 rows)

--清理测试数据
postgres=# SELECT 'stop' FROM pg_drop_replication_slot('test_slot');
```

```
 ?column?
 ----------
 stop
(1 row)

postgres=#
postgres=# DROP TABLE table2_with_pk;
DROP TABLE
postgres=# DROP TABLE table2_without_pk;
DROP TABLE
```

- 通过 SQL 调用函数消费更改（使用格式版本 2）

```
postgres=# CREATE TABLE table3_with_pk (a SERIAL, b VARCHAR(30), c TIMESTAMP NOT
NULL, PRIMARY KEY(a, c));
CREATE TABLE
postgres=# CREATE TABLE table3_without_pk (a SERIAL, b NUMERIC(5,2), c TEXT);
CREATE TABLE
postgres=#
postgres=# SELECT 'init' FROM pg_create_logical_replication_slot('test_slot',
'wal2json');
 ?column?
----------
 init
(1 row)

postgres=# BEGIN;
BEGIN
postgres=# INSERT INTO table3_with_pk (b, c) VALUES('Backup and Restore', now());
INSERT 0 1
postgres=# INSERT INTO table3_with_pk (b, c) VALUES('Tuning', now());
INSERT 0 1
postgres=# INSERT INTO table3_with_pk (b, c) VALUES('Replication', now());
INSERT 0 1

postgres=# DELETE FROM table3_with_pk WHERE a < 3;
DELETE 2
postgres=# INSERT INTO table3_without_pk (b, c) VALUES(2.34, 'Tapir');
INSERT 0 1

 --这条修改不会添加到流中，因为表没有主键或复制标识
postgres=# UPDATE table3_without_pk SET c = 'Anta' WHERE c = 'Tapir';
```

```
UPDATE 1
postgres=# COMMIT;
COMMIT
postgres=# SELECT data FROM pg_logical_slot_get_changes('test_slot', NULL, NULL,
'format-version', '2', 'add-msg-prefixes', 'wal2json');
WARNING:  no tuple identifier for UPDATE in table "public"."table3_without_pk"
CONTEXT:  slot "test_slot", output plugin "wal2json", in the change callback,
associated LSN 0/4B411EB0

                data

------------------------------------------------------------------------------
----------------------------------------
------------------------------------------------------------------------------
----------------------------------------
----------------------------------------
 {"action":"B"}

{"action":"I","schema":"public","table":"table3_with_pk","columns":[{"name":"
a","type":"integer","value":4},{"name"
:"b","type":"character varying(30)","value":"Backup and
Restore"},{"name":"c","type":"timestamp without time zone","
value":"2024-03-07 16:16:24.579427"}]}

{"action":"I","schema":"public","table":"table3_with_pk","columns":[{"name":"
a","type":"integer","value":5},{"name"
:"b","type":"character
varying(30)","value":"Tuning"},{"name":"c","type":"timestamp without time
zone","value":"2024
-03-07 16:16:24.579427"}]}

{"action":"I","schema":"public","table":"table3_with_pk","columns":[{"name":"
a","type":"integer","value":6},{"name"
:"b","type":"character
varying(30)","value":"Replication"},{"name":"c","type":"timestamp without time
zone","value":
"2024-03-07 16:16:24.579427"}]}

{"action":"I","schema":"public","table":"table3_without_pk","columns":[{"name
":"a","type":"integer","value":1},{"na
```

```
me":"b","type":"numeric(5,2)","value":2.34},{"name":"c","type":"text","value":
"Tapir"}]}
 {"action":"C"}
(6 rows)

postgres=# SELECT * FROM table3_with_pk;
 a |         b          |             c
---+--------------------+---------------------------
 4 | Backup and Restore | 2024-03-07 16:16:24.579427
 5 | Tuning             | 2024-03-07 16:16:24.579427
 6 | Replication        | 2024-03-07 16:16:24.579427
(3 rows)

--获取需要的操作数据
postgres=# BEGIN;
BEGIN
postgres=# INSERT INTO table3_with_pk (b, c) VALUES('joan', now());
INSERT 0 1
postgres=# UPDATE table3_with_pk SET b = 'hexq' WHERE b = 'joan';
UPDATE 1
postgres=# COMMIT;
COMMIT
postgres=# SELECT * FROM table3_with_pk;
 a |         b          |             c
---+--------------------+---------------------------
 4 | Backup and Restore | 2024-03-07 16:16:24.579427
 5 | Tuning             | 2024-03-07 16:16:24.579427
 6 | Replication        | 2024-03-07 16:16:24.579427
 7 | hexq               | 2024-03-07 16:18:22.365904
(4 rows)

--只获取 insert 动作的数据
postgres=# SELECT data FROM pg_logical_slot_peek_changes('test_slot', NULL, NULL,
'format-version', '2', 'actions', 'insert');

          data

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

```
------------------------
 {"action":"B"}

{"action":"I","schema":"public","table":"table3_with_pk","columns":[{"name":"
a","type":"integer","value":7},{"name"
:"b","type":"character
varying(30)","value":"joan"},{"name":"c","type":"timestamp without time
zone","value":"2024-0
3-07 16:18:22.365904"}]}
 {"action":"C"}
(3 rows)

--只获取 update 动作的数据
postgres=# SELECT data FROM pg_logical_slot_peek_changes('test_slot', NULL, NULL,
'format-version', '2', 'actions', 'update');


data


------------------------------------------------------------------------------
----------------------------------------
------------------------------------------------------------------------------
----------------------------------------
------------------------------------------------------------------------------
----------------------------------------
------------------------------------------------
 {"action":"B"}

{"action":"U","schema":"public","table":"table3_with_pk","columns":[{"name":"
a","type":"integer","value":7},{"name"
:"b","type":"character
varying(30)","value":"hexq"},{"name":"c","type":"timestamp without time
zone","value":"2024-0
3-07
16:18:22.365904"}],"identity":[{"name":"a","type":"integer","value":7},{"name
":"c","type":"timestamp without ti
me zone","value":"2024-03-07 16:18:22.365904"}]}
 {"action":"C"}
(3 rows)
```

注意

- 需监控槽的消费情况，如果中断消费，且槽位未删除会导致 WAL 无法自动清理
- 表必须要有主键或复制标识

11 pg_bulkload

pg_bulkload 是一种用于 GBase 8c 的高速数据加载工具，相比 copy 命令。最大的优势就是速度。优势在让我们跳过 shared buffer,wal buffer。直接写文件。pg_bulkload 的 direct 模式就是这种思路来实现的，它还包含了数据恢复功能，即导入失败的话，需要恢复。

pg_bulkload 旨在将大量数据加载到数据库中。用户可以选择是否检查数据库约束以及在加载期间忽略多少错误。例如，将数据从另一个数据库复制到 GBase 8c，可以跳过性能完整性检查。另一方面，可以在加载不干净的数据时启用约束检查。

pg_bulkload 的最初目标是 COPY 在 PostgreSQL 中更快地替代命令，但 3.0 或更高版本具有一些 ETL 功能，例如输入数据验证和具有过滤功能的数据转换。

pg_bulkload 加载数据时，在内部调用用户定义函数 pg_bulkload() 并执行加载。pg_bulkload() 函数将在 pg_bulkload 安装期间安装。

11.1 语法

```
pg_bulkload is a bulk data loading tool for PostgreSQL

Usage:
  Dataload: pg_bulkload [dataload options] control_file_path
  Recovery: pg_bulkload -r [-D DATADIR]

Dataload options:
  -i, --input=INPUT         INPUT path or function
  -O, --output=OUTPUT       OUTPUT path or table
  -l, --logfile=LOGFILE     LOGFILE path
  -P, --parse-badfile=*     PARSE_BADFILE path
  -u, --duplicate-badfile=* DUPLICATE_BADFILE path
  -o, --option="key=val"    additional option

Recovery options:
  -r, --recovery            execute recovery
  -D, --pgdata=DATADIR      database directory

Connection options:
```

```
  -d, --dbname=DBNAME       database to connect
  -h, --host=HOSTNAME       database server host or socket directory
  -p, --port=PORT           database server port
  -U, --username=USERNAME   user name to connect as
  -w, --no-password         never prompt for password
  -W, --password            force password prompt

Generic options:
  -e, --echo                echo queries
  -E, --elevel=LEVEL        set output message level
  --help                    show this help, then exit
  --version                 output version information, then exit

Read the website for details. <http://pgbulkload.projects.postgresql.org/>
Report bugs to <pgbulkload-general@pgfoundry.org>.
```

11.2 使用

(1) 初始化数据

```
postgres=# create database testdb ;
CREATE DATABASE
postgres=# \c testdb
You are now connected to database "testdb" as user "postgres".
testdb=#
testdb=# create table tb_asher (id int,name text);
CREATE TABLE
testdb=# \d
          List of relations
 Schema |   Name    | Type  |  Owner
--------+----------+-------+----------
 public | tb_asher | table | postgres
(1 row)

testdb=# create extension pg_bulkload;  #如果连接指定到单个库时，需要创建扩展以生成 pgbulkload.pg_bulkload() 函数
CREATE EXTENSION

test=# \q
```

模拟 CSV 文件

```
postgres@s2ahumysqlpg01-> seq 100000| awk '{print $0"|asher"}' > bulk_asher.txt
```

```
postgres@s2ahumysqlpg01-> more   bulk_asher.txt
1|asher
2|asher
3|asher
4|asher
5|asher
6|asher
```

(2) 加载到指定表

```
# 将 bulk_asher.txt 里的数据加载到 testdb 库下的 tb_asher 表中
postgres@s2ahumysqlpg01->  pg_bulkload -i home/postgres/bulk_asher.txt -O
tb_asher  -l home/postgres/tb_asher_output.log -P
home/postgres/tb_asher_bad.txt  -o "TYPE=CSV" -o "DELIMITER=|" -d testdb -U
postgres  -h 127.0.0.1
NOTICE: BULK LOAD START
NOTICE: BULK LOAD END
        0 Rows skipped.
        100000 Rows successfully loaded.
        0 Rows not loaded due to parse errors.
        0 Rows not loaded due to duplicate errors.
        0 Rows replaced with new rows.
```

查看导入日志：

```
postgres@s2ahumysqlpg01-> cat home/postgres/tb_asher_output.log

pg_bulkload 3.1.19 on 2022-02-20 16:32:28.642071+08

INPUT = home/postgres/bulk_asher.txt
PARSE_BADFILE = home/postgres/tb_asher_bad.txt
LOGFILE = home/postgres/tb_asher_output.log
LIMIT = INFINITE
PARSE_ERRORS = 0
CHECK_CONSTRAINTS = NO
TYPE = CSV
SKIP = 0
DELIMITER = |
QUOTE = "\""
ESCAPE = "\""
NULL =
OUTPUT = public.tb_asher
```

```
MULTI_PROCESS = NO
VERBOSE = NO
WRITER = DIRECT
DUPLICATE_BADFILE =
u01/postgresql/data_bak/pg_bulkload/20220220163228_testdb_public_tb_asher.dup.
csv
DUPLICATE_ERRORS = 0
ON_DUPLICATE_KEEP = NEW
TRUNCATE = NO


  0 Rows skipped.
  100000 Rows successfully loaded.
  0 Rows not loaded due to parse errors.
  0 Rows not loaded due to duplicate errors.
  0 Rows replaced with new rows.

Run began on 2022-02-20 16:32:28.642071+08
Run ended on 2022-02-20 16:32:28.743741+08

CPU 0.02s/0.04u sec elapsed 0.10 sec
```

(3) 先清空在加载

```
#增加了 -o "TRUNCATE=YES" 参数
postgres@s2ahumysqlpg01-> pg_bulkload -i home/postgres/bulk_asher.txt -O
tb_asher  -l home/postgres/tb_asher_output.log -P
home/postgres/tb_asher_bad.txt  -o "TYPE=CSV" -o "DELIMITER=|" -o
"TRUNCATE=YES"  -d testdb -U postgres -h 127.0.0.1
4NOTICE: BULK LOAD START
NOTICE: BULK LOAD END
        0 Rows skipped.
        100000 Rows successfully loaded.
        0 Rows not loaded due to parse errors.
        0 Rows not loaded due to duplicate errors.
        0 Rows replaced with new rows.

# 数据查询
postgres@s2ahumysqlpg01-> psql -h127.0.0.1 -d testdb  -c "select count(1) from
tb_asher ;"
 count
--------
```

```
 100000
(1 row)
```

(4) 使用控制文件

```
# 新建控制文件 ，可以根据之前加载时，产生的日志文件 tb_asher_output.log 来更改，去掉里面没有值的参数  NULL =
vi  asher.ctl
INPUT = home/postgres/bulk_asher.txt
PARSE_BADFILE = home/postgres/tb_asher_bad.txt
LOGFILE = home/postgres/tb_asher_output.log
LIMIT = INFINITE
PARSE_ERRORS = 0
CHECK_CONSTRAINTS = NO
TYPE = CSV
SKIP = 0
DELIMITER = |
QUOTE = "\""
ESCAPE = "\""
OUTPUT = public.tb_asher
MULTI_PROCESS = NO
VERBOSE = NO
WRITER = DIRECT
DUPLICATE_BADFILE =
u01/postgresql/data_bak/pg_bulkload/20220220164822_testdb_public_tb_asher.dup.csv
DUPLICATE_ERRORS = 0
ON_DUPLICATE_KEEP = NEW
TRUNCATE = YES

# 使用控制文件来加载
postgres@s2ahumysqlpg01-> pg_bulkload  home/postgres/asher.ctl -d testdb -U postgres -h 127.0.0.1
NOTICE: BULK LOAD START
NOTICE: BULK LOAD END
        0 Rows skipped.
        100000 Rows successfully loaded.
        0 Rows not loaded due to parse errors.
        0 Rows not loaded due to duplicate errors.
        0 Rows replaced with new rows.

# 数据查询
```

```
postgres@s2ahumysqlpg01-> psql -h127.0.0.1 -d testdb  -c "select count(1) from
tb_asher ;"
 count
--------
 100000
(1 row)
```

(5) 强制写 wal 日志

```
# pg_bulkload 默认是跳过 buffer 直接写文件 ，但时如果有复制 ，或者需要基本 wal
日志恢复时没有 wal 日志是不行的，这是我们可以强制让其写 wal 日志 ，只需要加载 -o
"WRITER=BUFFERED" 参数就可以了
 pg_bulkload -i home/postgres/bulk_asher.txt -O tb_asher  -l
home/postgres/tb_asher_output.log -P home/postgres/tb_asher_bad.txt  -o
"TYPE=CSV" -o "DELIMITER=|" -o "TRUNCATE=YES" -o "WRITER=BUFFERED"  -d testdb -U
postgres -h 127.0.0.1

NOTICE: BULK LOAD START
NOTICE: BULK LOAD END
        0 Rows skipped.
        100000 Rows successfully loaded.
        0 Rows not loaded due to parse errors.
        0 Rows not loaded due to duplicate errors.
        0 Rows replaced with new rows.
```

GBASE®

南大通用数据技术股份有限公司
General Data Technology Co., Ltd.

微信二维码

技术支持热线：400-013-9696