云数据库 GaussDB

M-Compatibility 开发指南（集中式）

文档版本	01
发布日期	2025-10-23

华为云计算技术有限公司

商标声明


和其他华为商标均为华为技术有限公司的商标。
本文档提及的其他所有商标或注册商标，由各自的所有人拥有。

注意

您购买的产品、服务或特性等应受华为云计算技术有限公司商业合同和条款的约束，本文档中描述的全部或部分产品、服务或特性可能不在您的购买或使用范围之内。除非合同另有约定，华为云计算技术有限公司对本文档内容不做任何明示或暗示的声明或保证。

由于产品版本升级或其他原因，本文档内容会不定期进行更新。除非另有约定，本文档仅作为使用指导，本文档中的所有陈述、信息和建议不构成任何明示或暗示的担保。

华为云计算技术有限公司

地址： 贵州省贵安新区黔中大道交兴功路华为云数据中心 邮编：550029

网址： https://www.huaweicloud.com/

目 录

1 M-Compatibility 数据库系统概述

M-Compatibility数据库是GaussDB数据库的增强版本，新增了数据类型、语法、函数等功能，能够更好地满足企业级客户对高性能、高可靠、高安全、高智能、高扩展及易迁移数据库诉求。

M-Compatibility数据库逻辑架构继承了GaussDB原有数据库逻辑架构，具体请参考《开发指南》的“数据库系统概述 > 数据库逻辑结构图”章节。

Database和Schema设计请参考《兼容性说明》中的“MySQL兼容性说明 > MySQL数据库兼容性概述 > MySQL兼容性M-Compatibility模式概述”章节的Database和Schema设计部分内容。

1.1 数据查询请求处理过程

图 1-1 GaussDB 服务响应流程

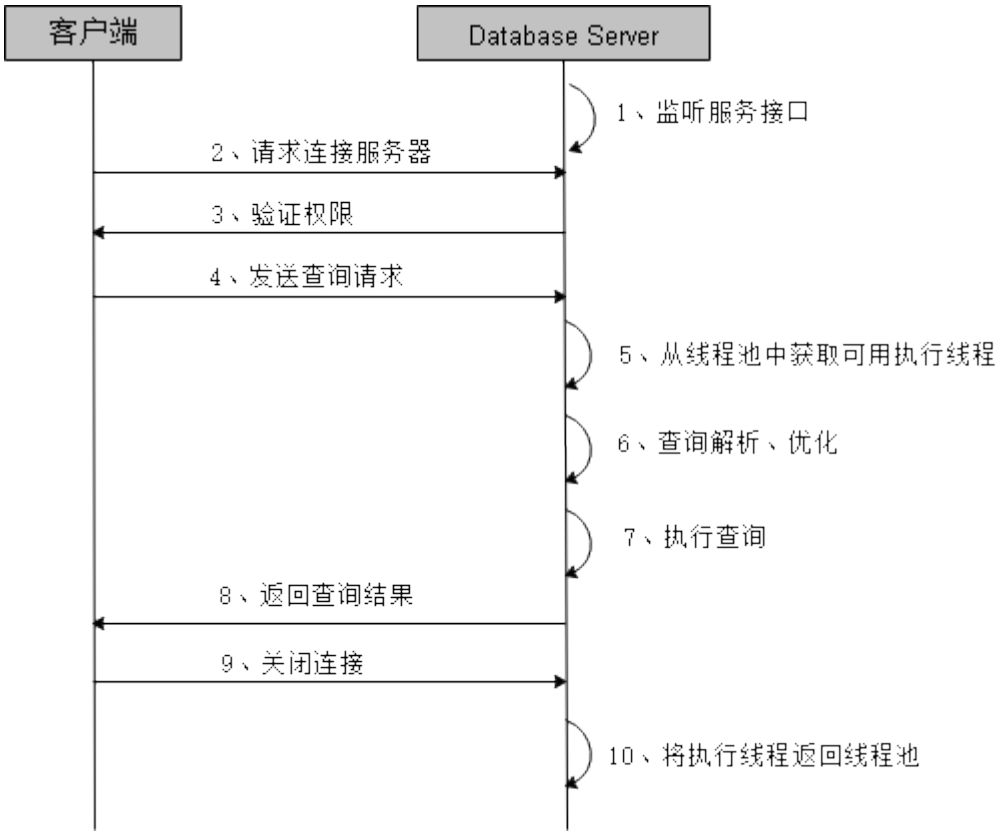
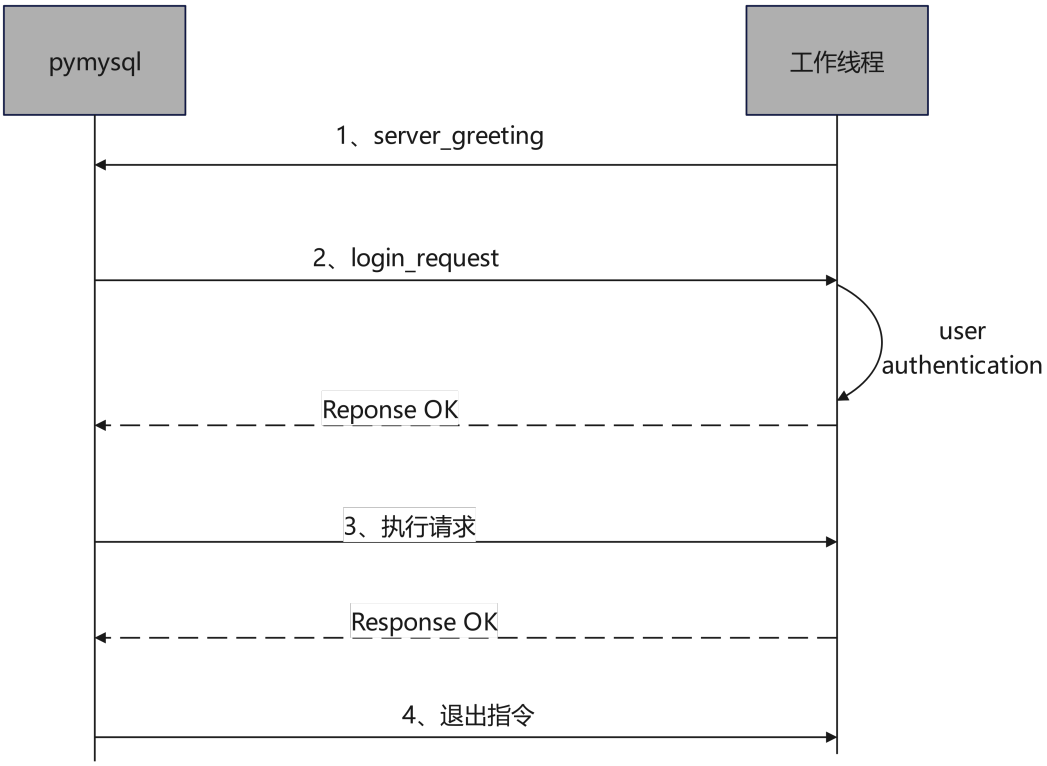


图 1-2 pymysql 与 M-Compatibility 数据库交互流程



1.2 M-Compatibility 数据库不支持的 GaussDB 特性

M-Compatibility数据库当前暂不支持《特性指南》内如下特性：

1. 物化视图。
2. 全密态数据库相关特性，包括“设置密态等值查询”和“内存解密逃生通道”。
3. SPM计划管理。
4. DB4AI特性。

2 创建 M-Compatibility 数据库及用户

创建M-Compatibility数据库需要在非M-Compatibility数据库下进行，需要确保当前环境中已有templatem模板数据库。

```
gaussdb=# \l
                                 List of databases
   Name    | Owner | Encoding |   Collate   |    Ctype    | Access privileges
-----------+-------+----------+-------------+-------------+-------------------
 postgres  | omm   | UTF8     | en_US.UTF-8 | en_US.UTF-8 |
 template0 | omm   | UTF8     | en_US.UTF-8 | en_US.UTF-8 | =c/omm           +
           |       |          |             |             | omm=CTc/omm
 template1 | omm   | UTF8     | en_US.UTF-8 | en_US.UTF-8 | =c/omm           +
           |       |          |             |             | omm=CTc/omm
 templatem | omm   | UTF8     | en_US.UTF-8 | en_US.UTF-8 | =c/omm           +
           |       |          |             |             | omm=CTc/omm
(4 rows)
```

从505.1.0之前的版本升级到最新版本，基础包中如果templatem模板库未创建，升级之后templatem模板库也不会创建。

使用M-Compatibility数据库，需要先通过以下示例创建数据库。

```
gaussdb=# CREATE DATABASE m_db dbcompatibility = 'M' template=templatem;
gaussdb=# \c m_db
Non-SSL connection (SSL connection is recommended when requiring high-security)
You are now connected to database "m_db" as user "omm".
m_db=# CREATE USER omm_test password '********';
m_db=# GRANT ALL privileges to omm_test;
```

创建M-Compatibility数据库后，其它操作请参考**SQL参考**。

须知

建议在升级至505.1版本之前检查是否存在非M-Compatibility的templatem数据库，若用户已创建templatem数据库，请联系华为技术工程师提供技术支持。如不处理，会影响到M-Compatibility数据库的相关特性。

3 应用程序开发教程

3.1 数据库访问路由

Database和Schema设计请参考《兼容性说明》中的“MySQL兼容性说明 > MySQL数据库兼容性概述 > MySQL兼容性M-Compatibility模式概述”章节的Database和Schema设计部分内容。

步骤1 参考**创建M-Compatibility数据库及用户**创建数据库和用户。

步骤2 m_db数据库下会默认生成omm_test同名Schema和其他默认的Schema，这些Schema就是工作空间。可以使用以下三种方式进行数据表的访问。

1. 指定database和schema直接使用table名进行访问。
2. 指定database不指定schema，系统默认路由到连接用户默认创建的同名schema，使用schema.table的方式进行访问。
3. 指定database不指定schema，使用SQL或者函数接口的方式切换schema，直接使用table名进行访问。

----结束

3.2 开发规范

如果用户在APP的开发中，使用了连接池机制，那么需要遵循如下规范：

- 如果在连接中设置了GUC参数，那么在将连接归还连接池之前，必须使用“SET SESSION AUTHORIZATION DEFAULT;RESET ALL;”将连接的状态清空。
- 如果使用了临时表，那么在将连接归还连接池之前，必须将临时表删除。

否则，连接池里面的连接就是有状态的，会对用户后续使用连接池进行操作的正确性带来影响。

应用程序开发驱动兼容性说明如**表3-1**所示：

表 3-1 兼容性说明

驱动	兼容性说明
pymysql	驱动前向兼容数据库，若需使用驱动与数据库同步增加的新特性，须升级数据库。

须知

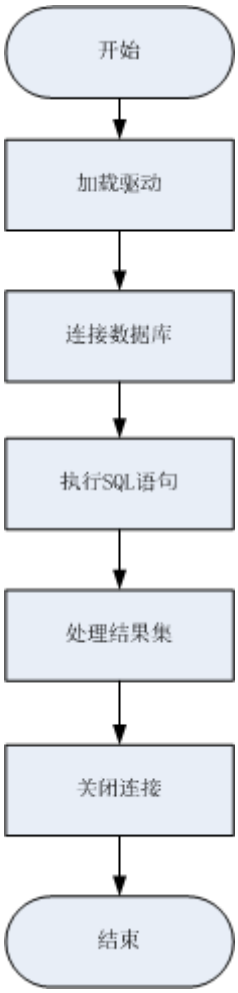
- plat_compat_server_port参数合法设置且成功启动后才可以进行应用程序连接。
- 应用程序连接时需要保证使用SSL进行安全的TCP/IP连接或者使用RSA加密登录密码的方式通信以保证用户信息安全。

3.3 基于 pymysql 开发

pymysql是一个Python库，旨在为数据库提供SQL语句执行的统一访问接口。pymysql提供了诸如客户端游标和服务器端游标、异步通信和通知等功能。pymysql对Unicode的兼容性良好，因此能够处理多种字符编码的数据，确保数据的完整性和准确性。此外，M-Compatibility数据库支持pymysql通过SSL模式进行安全连接，保障数据传输的安全性和可靠性。

3.3.1 开发流程

图 3-1 采用 pymysql 开发应用程序的流程



3.3.2 开发步骤

注：操作前需按照**创建M-Compatibility数据库及用户**创建好数据库和用户。

步骤1 配置好GUC参数并开启数据库。

步骤2 使用pip命令安装pymysql库（版本1.0.2及以上）。如果使用到RSA，还需使用pip命令安装cryptography库，并设置GUC参数 plat_compat_allow_public_key_retrieval为on。

步骤3 加载pymysql库。

在创建数据库连接之前，需要先import pymysql。

```
import  pymysql
```

步骤4 连接数据库。

非SSL方式连接数据库：

1. 使用pymysql.connect函数获得connection对象。
2. 使用connection对象创建cursor对象。

SSL方式连接数据库（tls版本1.2及以上）：

用户通过pymysql连接M-Compatibility时，可以通过开启SSL加密客户端和服务器之间的通讯。在使用SSL时，默认用户已经获取了客户端所需要的证书和私钥文件，关于证书等文件的获取请参考Openssl相关文档和命令。

1. 使用*.ini文件（python的configparser包可以解析这种类型的配置文件）保存数据库连接的配置信息。
2. 在连接选项中添加SSL连接相关参数：ca, key, cert。
   a. ssl_disabled: SSL通信开关。
   b. ssl_cert：客户端公钥路径。
   c. ssl_ca: SSL通信ca证书路径。
   d. ssl_key：客户端私钥路径。
   e. ssl_verify_cert：客户端是否校验服务端证书。
   f. ssl_verify_identity: 客户端是否校验服务端地址。
3. 使用pymysql.connect函数获得connection对象。
4. 使用connection对象创建cursor对象。

步骤5 执行SQL语句。

1. 构造操作语句，使用%s作为占位符，执行时pymysql会用参数值智能替换掉占位符。可以添加RETURNING子句，来得到自动生成的字段值。
2. 使用cursor.execute方法来操作一行SQL语句，使用cursor.executemany方法来操作多行SQL语句。

步骤6 处理结果集。

1. cursor.fetchone()：这种方法提取的查询结果集的下一行，返回一个序列，没有数据可用时则返回空。
2. cursor.fetchall()：这个例程获取所有查询结果（剩余）行，返回一个列表。空行时则返回空列表。
3. cursor.fetchmany(size=None)：这个例程获取size行查询结果行，返回一个列表。未设置size值则返回所有剩余行。空行时则返回空列表。

说明

对于数据库特有数据类型，如tinyint类型，查询结果中相应字段为字符串形式。

步骤7 关闭连接。

在使用数据库连接完成相应的数据操作后，需要关闭数据库连接。关闭数据库连接可以直接调用其close方法，如connection.close()。

须知

此方法关闭数据库连接，并不自动调用commit()。如果只是关闭数据库连接而不调用commit()方法，那么所有更改将会丢失。

----结束

3.3.3 示例：常用操作

```
import pymysql

# 创建连接对象（非SSL连接），客户端和服务端的字符集需保持一致，否则会报错。
conn=pymysql.connect(database="database",user="user",password="********",host="localhost",port=port,char
set="utf8")
# 创建连接对象（SSl连接），客户端和服务端的字符集需保持一致，否则会报错。
conn=pymysql.connect(database="database_name",
                     user="user",
                     password="********",
                     host="IP_address",
                     port=port,
                     ssl_disabled=False,
                     ssl_ca="file1",
                     ssl_key="file2",
                     ssl_cert="file3"
)
# 注意：用户使用SSL远程连接到数据库节点时需要使用sha256的认证方式，并需要使用有效的CA根证书、服务
器公私钥、客户端公私钥。

cur=conn.cursor() # 创建指针对象
conn.set_character_set('utf8', collation=None) # 设置字符集。客户端和服务端的字符集需保持一致，否则会报
错。

# 开启事务
conn.begin()

# 创建表
cur.execute("CREATE TABLE student(id integer,name varchar(256),gender varchar(256));")

# 插入数据
cur.execute("INSERT INTO student(id,name,gender) VALUES(%s,%s,%s);",(1,'Aspirin','M'))
cur.execute("INSERT INTO student(id,name,gender) VALUES(%s,%s,%s);",(2,'Taxol','F'))
cur.execute("INSERT INTO student(id,name,gender) VALUES(%s,%s,%s);",(3,'Dixheral','M'))

# 批量插入数据
stus = ((4,'John','M'),(5,'Alice','F'),(6,'Peter','M'))
cur.executemany("INSERT INTO student(id,name,gender) VALUES(%s,%s,%s);",stus)

# 获取结果
cur.execute("SELECT * FROM student;")
results=cur.fetchall()
print(results)

cur.execute("SELECT * FROM student;")
results=cur.fetchmany()
print(results)

# 提交操作
conn.commit()
# 插入一条数据
cur.execute("INSERT INTO student(id,name,gender) VALUES(%s,%s,%s);",(7,'Lucy','F'))

# 回退操作
conn.rollback()

conn.select_db(dbname)

# 尝试重连
conn.ping()

# 关闭连接
cur.close()
conn.close()

# pymysql常用连接方式
# 常用RSA连接方式
conn = pymysql.connect(dbname="dbname", user="user", password="********", host="localhost", port=port)
```



```
# 常用SSL连接方式
conn = pymysql.connect(dbname="dbname", user="user", password="********", host="localhost",
port=port,'ssl_disabled':False,'ssl_ca': './ca.pem','ssl_key': './client-key.pem','ssl_cert': './client-cert.pem')
```

3.3.4 pymysql 接口参考

pymysql库是一套提供给用户的API方法，本节将对部分常用接口做具体描述。

3.3.4.1 pymysql.connect()

功能描述

此方法创建新的数据库会话并返回新的Connection对象。

原型

```
conn=pymysql.connect(database="databasename",user="username",password="********",host="127.0.0.1",port=portnum)
```

参数

表 3-2 pymysql.connect 参数

关键字	参数说明
database	数据库名称。
user	用户名。
password	密码。
host	数据库IP地址。
port	连接端口号。
ssl_disabled	客户端是否开启SSL，默认是None。
ssl_ca	客户端CA根证书，SSL连接时用。
ssl_cert	客户端证书路径，SSL连接时用。
ssl_key	客户端密钥路径，SSL连接时用。
charset	客户端编码格式。
autocommit	autocommit模式（默认是False）。

返回值

Connection对象（连接数据库实例的对象）。

示例

请参见 **示例：常用操作**。

3.3.4.2 connection.cursor()

功能描述

此方法用于返回新的Cursor对象。

原型

```
conn=pymysql.connect(database="databasename",user="username",password="********",host="127.0.0.1",port=portnum)
conn.cursor()
```

返回值

Cursor对象（用于整个数据库使用Python编程的cursor）。

示例

请参见**示例：常用操作**。

3.3.4.3 cursor.execute(query, args)

功能描述

此方法执行被参数化的SQL语句（即占位符，而不是SQL文字）。

原型

```
cursor.execute(query,args)
```

参数

表 3-3 cursor.execute 参数

关键字	参数说明
query	待执行的SQL语句。
args	变量列表，匹配query中%s占位符。

返回值

无。

约束

cursor.execute(query,args)，query一次只输入SQL语句，禁止多条SQL堆叠输入。

cursor.execute(query,args)，输入的SQL语句中禁止使用\0。

示例

请参见**示例：常用操作**。

3.3.4.4 cursor.executemany(query,args)

功能描述

此方法可以执行多条被参数化的SQL语句。

原型

```
cursor.executemany(query,args)
```

参数

表 3-4 cursor.executemany 参数

关键字	参数说明
query	待执行的SQL语句。
args	变量列表，匹配query中%s占位符。

返回值

无。

约束

cursor.executemany(query,args)，输入的SQL语句中禁止使用\0。

示例

请参见**示例：常用操作**。

3.3.4.5 connection.begin()

功能描述

此方法将开启一个事务。

须知

默认情况下，不调用begin()，pymysql也会在执行第一个命令之前打开一个事务。begin() 方法提供了手动控制事务的能力。

原型

```
connection.begin()
```

参数

无。

返回值

无。

示例

请参见**示例：常用操作**。

3.3.4.6 connection.commit()

功能描述

此方法将当前挂起的事务提交到数据库。

须知

默认情况下，pymysql在执行第一个命令之前打开一个事务：如果不调用commit()，任何数据操作的效果都将丢失。

原型

```
connection.commit()
```

参数

无。

返回值

无。

示例

请参见**示例：常用操作**。

3.3.4.7 connection.rollback()

功能描述

此方法回滚当前挂起事务。

须知

执行关闭连接“close()”而不先提交更改“commit()”将导致执行隐式回滚。

原型

```
connection.rollback()
```

参数

无。

返回值

无。

示例

请参见**示例：常用操作**。

3.3.4.8 cursor.fetchone()

功能描述

此方法提取查询结果集的下一行，并返回一个元组。

原型

```
cursor.fetchone()
```

参数

无。

返回值

单个元组，为结果集的第一条结果，当没有更多数据可用时，返回为“None”。

示例

请参见**示例：常用操作**。

3.3.4.9 cursor.fetchall()

功能描述

此方法获取查询结果的所有（剩余）行，并将它们作为元组列表返回。

原型

```
cursor.fetchall()
```

参数

无。

返回值

元组列表，为结果集的所有结果。空行时则返回空列表。

示例

请参见**示例：常用操作**。

3.3.4.10 cursor.fetchmany(size=None)

功能描述

此方法返回一个包含多行数据的列表，每行数据通常表示为一个元组或类似结构的对象。如果指定了 size 参数，返回的列表将包含指定数量的行；如果未指定 size 参数，则返回尽可能多的行，直到结果集耗尽。

原型

```
cursor.fetchmany(size=None)
```

参数

表 3-5 cursor.fetchmany 参数

关键字	参数说明
size	返回数据的行数

返回值

元祖列表。

示例

请参见**示例：常用操作**。

3.3.4.11 cursor.close()

功能描述

此方法关闭当前连接的游标。

原型

```
cursor.close()
```

参数

无。

返回值

无。

示例

请参见**示例：常用操作**。

3.3.4.12 connection.close()

功能描述

此方法关闭数据库连接。

须知

此方法关闭数据库连接，并不自动调用commit()。如果只是关闭数据库连接而不调用commit()方法，那么所有更改将会丢失。M-Compatibility数据库不支持connection.kill(thred_id)接口。

原型

```
connection.close()
```

参数

无。

返回值

无。

示例

请参见**示例：常用操作**。

3.3.4.13 connection.ping(reconnect = True)

功能描述

用于检测数据库连接是否可用。如果连接断开或不可用，根据参数设置决定是否重新连接。如果重连失败则报错。

原型

```
connection.ping(reconnect = True)
```

参数

表 3-6 connection.ping(reconnect = True)参数

关键字	参数说明
reconnect	是否重连（默认为True）

返回值

无。

示例

请参见**示例：常用操作**。

3.3.4.14 connection.select_db(dbname)

功能描述

切换了当前活动的数据库。

原型

```
connection.select_db(dbname)
```

参数

表 3-7 cursor.select_db(dbname)参数

关键字	参数说明
dbname	要连接的数据库名

返回值

无。

示例

请参见**示例：常用操作**。

3.3.4.15 connection.set_character_set(charset, collation=None)

功能描述

用于设置数据库连接的字符集和字符序。

须知

客户端和服务端的字符集需保持一致，否则会报错。

原型

```
connection.set_character_set(charset, collation=None)
```

参数

表 3-8 connection.set_character_set(charset, collation=None)参数

关键字	参数说明
charset	数据库连接的字符集
collation	数据库连接的字符序（默认为None）

返回值

无。

示例

请参见**示例：常用操作**。

3.3.4.16 connection.autocommit(arg)

功能描述

启用或关闭数据库连接的自动提交功能。通过pymysql连接时，默认情况下自动提交功能是关闭的，可以使用connection.autocommit(arg)方法启用或关闭。

原型

```
connection.autocommit(arg)
```

参数

arg：启用或关闭数据库连接的自动提交功能，参数取值为True或False。

返回值

无

示例

```
import pymysql
# 创建连接对象（非SSL连接），客户端和服务端的字符集需保持一致，否则会报错。
conn=pymysql.connect(database="database",user="user",password="********",host="localhost",port=port,char
set="utf8")
# 创建连接对象（SSl连接），客户端和服务端的字符集需保持一致，否则会报错。
conn=pymysql.connect(database="database_name",
                     user="user",
                     password="********",
                     host="IP_address",
                     port=port,
                     ssl_disabled=False,
                     ssl_ca="file1",
                     ssl_key="file2",
                     ssl_cert="file3"
)
# 注意：用户使用SSL远程连接到数据库节点时需要使用sha256的认证方式，并需要使用有效的CA根证书、服务
器公私钥、客户端公私钥。
```

```
assert conn.get_autocommit() == True

# 启用自动提交。
conn.autocommit(False)
assert conn.get_autocommit() == False

# 关闭自动提交。
conn.autocommit(True)
assert conn.get_autocommit() == True

# 关闭数据库连接。
conn.close()
```

3.4 基于 GaussDB JDBC 开发

JDBC（Java Database Connectivity，Java数据库连接）是一种用于执行SQL语句的Java API，可以为多种关系数据库提供统一访问接口，应用程序可基于它操作数据。

M-Comaptibility数据库支持基于GaussDB JDBC的开发，具体使用方法参见《开发指南》中的“应用程序开发 > 基于JDBC开发”章节，部分功能和GaussDB存在差异，具体参见“GaussDB JDBC连接M-Compatibility数据库使用约束”。

GaussDB JDBC 连接 M-Compatibility 数据库使用约束

1. 使用DriverManager.getConnection(String url, Properties info)方法创建数据库连接时，info参数不支持如下属性：binaryTransfer、binaryTransferEnable、binaryTransferDisable；
2. 使用DriverManager.getConnection(String url, Properties info)方法创建数据库连接时，当开启info参数mCompatible后，部分功能和行为会产生变更，变更兼容MySQL JDBC 5.1版本驱动，具体变更请参见《开发指南》中“应用程序开发 > 基于JDBC开发 > 开发步骤 > 连接数据库 > 连接参数参考”章节。
3. 其他不支持的GaussDB特性，具体参见**M-Compatibility数据库不支持的GaussDB特性**。
4. 在使用GaussDB JDBC连接数据库，在PBE场景执行参数绑定时，推荐按照**表1 各个数据类型推荐的JDBC写入方式**中的JDBC写入方式进行绑定，否则可能导致实际的操作符选择与GSQL执行时不一致。

表1 各个数据类型推荐的JDBC写入方式

数值数据类型	MYSQL数据类型	DRS JDBC写入方式
整数类型	BOOL	prepStatement.setBoolean
	BOOLEAN	prepStatement.setBoolean
	TINYINT[(M)] [UNSIGNED]	prepStatement.setShort/ prepStatement.setInt
	SMALLINT[(M)] [UNSIGNED]	prepStatement.setShort/ prepStatement.setInt
	MEDIUMINT[(M)] [UNSIGNED]	prepStatement.setInt/ prepStatement.setInt

数值数据类型	MYSQL数据类型	DRS JDBC写入方式
	INT[(M)] [UNSIGNED]	prepStatement.setInt/ prepStatement.setLong
	INTEGER[(M)] [UNSIGNED]	prepStatement.setInt/ prepStatement.setLong
	BIGINT[(M)] [UNSIGNED]	prepStatement.setLong/ prepStatement.setBigDecimal
任意精度类型	DECIMAL[(M[,D])] [UNSIGNED]	prepStatement.setBigDecimal
	NUMERIC[(M[,D])] [UNSIGNED]	prepStatement.setBigDecimal
	DEC[(M[,D])] [UNSIGNED]	prepStatement.setBigDecimal
	FIXED[(M[,D])]	prepStatement.setBigDecimal
	FLOAT[(M,D)] [UNSIGNED]	prepStatement.setFloat(index, Float.parseFloat(val))
	FLOAT(p) [UNSIGNED]	prepStatement.setFloat(index, Float.parseFloat(val))
	DOUBLE[(M,D)] [UNSIGNED]	prepStatement.setDouble
	DOUBLE PRECISION[(M,D)]	prepStatement.setDouble
	REAL[(M,D)] [UNSIGNED]	prepStatement.setDouble
日期与时间数据类型	DATE	prepStatement.setDate
	DATETIME[(fsp)]	prepStatement.setString/ prepStatement.setTimestamp(1970/1/1 0:00:00)
	TIMESTAMP[(fsp)]	prepStatement.setString/ prepStatement.setTimestamp
	TIME[(fsp)]	prepStatement.setString
	YEAR	prepStatement.setDate
字符串数据类型	CHAR[(M)]	prepStatement.setObject(index, val, 1);
	VARCHAR(M)	prepStatement.setString
	TINYTEXT	prepStatement.setString
	TEXT	prepStatement.setString

数值数据类型	MYSQL数据类型	DRS JDBC写入方式
	MEDIUMTEXT	prepStatement.setString
	LONGTEXT	prepStatement.setString
	ENUM('value1','value2',...)	prepStatement.setString
	SET('value1','value2',...)	prepStatement.setString
二进制数据类型	BINARY[(M)]	prepStatement.setBytes(index, val);
	VARBINARY(M)	prepStatement.setBytes(index, val);
	TINYBLOB	prepStatement.setBlob(index,PGBlob);
	BLOB	prepStatement.setBlob(index,PGBlob);
	MEDIUMBLOB	prepStatement.setBlob(index,PGBlob);
	LONGBLOB	prepStatement.setBlob(index,PGBlob);
	BIT[(M)]	prepStatement.setBigDecimal
JSON 数据类型	JSON	prepStatement.setString

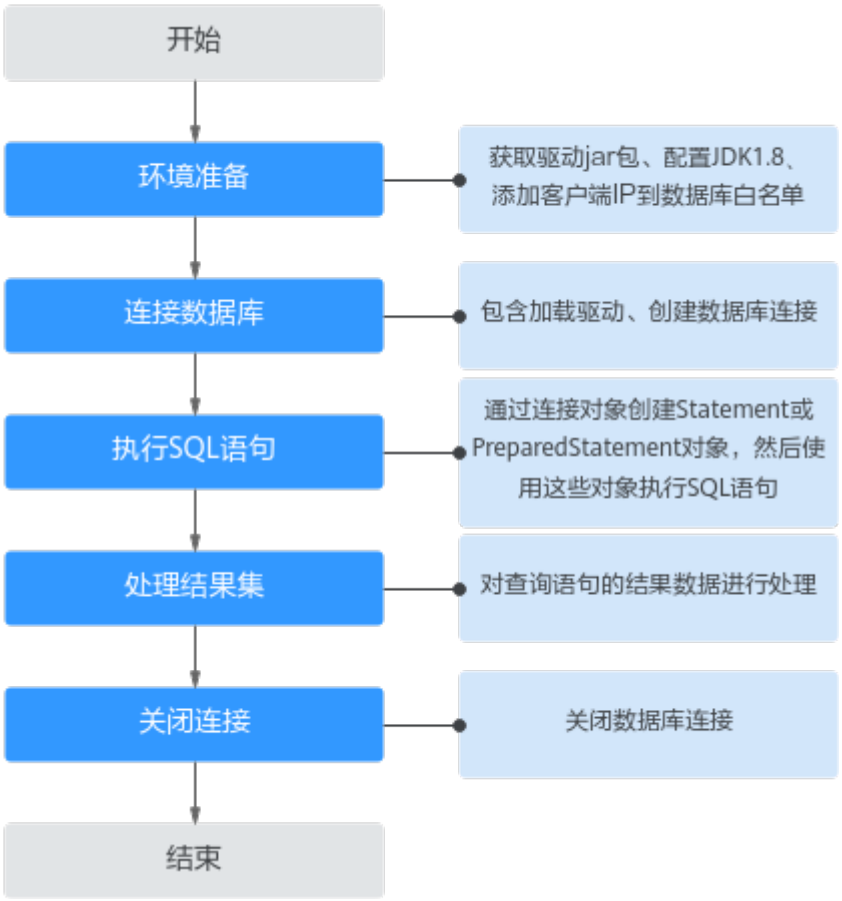
5. 使用getGeneratedKeys方法获取批量插入的自增值时，返回的结果是由数据库驱动根据首条自增值、自增步长以及插入的条目数决定。如果在批量插入操作中用户指定了自增列的值，那么返回的结果可能与预期的结果存在差异。
6. 当GUC参数m_err_sql_dialect的值为"MySQL"时，可以启用错误码替换功能，将GaussDB内核的错误码和SQLSTATE替换为M-Compatibility模式数据库的错误码和SQLSTATE。

3.5 基于 MySQL JDBC 开发

M-Compatibility模式数据库支持基于MySQL JDBC的开发。应用程序通过加载MySQL Connector/J 驱动，连接M-Compatibility模式数据库，实现数据操作与管理。

3.5.1 开发流程

图 3-2 采用 JDBC 开发应用程序的流程



3.5.2 开发步骤

3.5.2.1 环境准备

获取驱动 jar 包

- 从MySQL官方网站下载MySQL Connector/J驱动，如：mysql-connector-java-5.1.49.jar（本章节示例均以此版本驱动为例）。
- 支持的MySQL Connector/J版本5.1.47版本及以后。

安装 Java 环境

安装准备：客户端需配置JDK1.8软件包。JDK支持Windows、Linux等多种平台。下面以Windows为例，配置方法如下。

步骤1 DOS窗口（Windows环境中的命令提示符）输入“java -version”，查看JDK版本，确认为JDK1.8版本。如果未安装JDK，请从官方网站下载安装包并安装。

步骤2 配置系统环境变量。

1. 右键单击“我的电脑”，选择“属性”。
2. 在“系统”页面左侧导航栏单击“高级系统设置”。
3. 在“系统属性”页面，“高级”页签上单击“环境变量”。
4. 在“环境变量”页面上，“系统变量”区域单击“新建”或“编辑”配置系统变量。变量说明如表3-9所示。

表 3-9 变量说明

变量名	操作	变量值
JAVA_HOME	– 若存在，则单击“编辑”。 – 若不存在，则单击“新建”。	JAVA的安装目录。 例如：C:\Program Files\Java\jdk1.8.0_131。
Path	单击“编辑”。	– 若配置了JAVA_HOME，则在变量值的最前面加上： %JAVA_HOME%\bin。 – 若未配置JAVA_HOME，则在变量值的最前面加上 JAVA安装的全路径：C:\Program Files\Java\jdk1.8.0_131\bin。
CLASSPATH	单击“新建”。	%JAVA_HOME%\lib;%JAVA_HOME%\lib\tools.jar。

----结束

添加客户端 IP 到数据库白名单

将客户端IP添加到数据库白名单，具体操作请联系管理员处理。

设置兼容性参数

步骤1 配置协议监听的TCP端口号，设置GUC参数plat_compat_server_port为合理端口。

步骤2 如果使用到RSA，需要设置GUC参数plat_compat_allow_public_key_retrieval为on。

步骤3 将GUC参数m_format_dev_version设置为“s2”。

----结束

3.5.2.2 连接数据库

连接数据库前需先加载驱动。本章节介绍如何加载驱动、数据库连接参数及连接数据库的方式。

3.5.2.2.1 加载驱动

在创建数据库连接之前，需要先加载数据库驱动程序“com.mysql.jdbc.Driver”。

加载驱动有以下方法：

- 在代码中创建连接之前，在任意位置隐含装载：Class.forName("com.mysql.jdbc.Driver");。
- 在JVM启动时参数传递：java -Djdbc.drivers=com.mysql.jdbc.Driver *jdbctest*（jdbctest表示测试用例程序的名称）。

说明

MySQL 8.0以上驱动版本，包名由com.mysql.jdbc.Driver变更为com.mysql.cj.jdbc.Driver，使用时需根据对应驱动版本适配修改。

3.5.2.2.2 数据库连接参数

函数原型

JDBC提供了三个方法，用于创建数据库连接。

- DriverManager.getConnection(String url)。
- DriverManager.getConnection(String url, Properties info)。
- DriverManager.getConnection(String url, String user, String password)。

连接参数

表 3-10 数据库连接参数

参数	描述

url	MySQL JDBC驱动的数据库连接描述符。格式如下：
	jdbc:mysql://host:port/database
	jdbc:mysql://host:port/database?param1=value1&param2=value2
	说明
	• database为需要连接的数据库名称，不可缺省，需要明确指定。
	• host为数据库服务器名称或IP地址，支持IPv4格式地址。
	  • 由于安全原因，数据库主节点禁止数据库内部其他节点无认证接入。如果要在数据库内部访问数据库主节点，请将JDBC程序部署在数据库主节点所在机器，host使用“127.0.0.1”。否则可能会出现“FATAL: Forbid remote connection with trust method!”错误。
	  • 建议业务系统单独部署在数据库外部，否则可能会影响数据库运行性能。
	  • 缺省情况下，连接服务器为localhost。
	• port为数据库服务器端口，不可缺省，需要在服务端设置GUC参数plat_compat_server_port。
	• param为参数名称，即数据库连接属性。
	参数可以配置在URL中，以?开始配置，以=为参数赋值，以&作为不同参数的间隔。也可以采用Properties对象的属性方式进行配置。
	• value为参数值，即数据库连接属性值。
	• 连接时需配置connectTimeout和socketTimeout，如果未配置，默认为0，即不会超时。在DN与客户端出现网络故障时，客户端一直未收到DN侧ACK确认报文时，会启动超时重传机制，不断地进行重新传递。当超时时间达到系统默认的600秒后才会报超时错误，这会导致RTO时间很高。
	• 建议使用JDBC标准接口建立连接时，确保URL格式的合法性，不合法的URL会导致异常，且异常中包含原始URL字符串，可能造成敏感信息泄漏。

info	数据库连接属性。具体请参见**info参数的连接属性**。

参数	描述
user	数据库用户。
password	数据库用户的密码。

info 参数的连接属性

info参数连接的所有属性名称对大小写敏感。常用的属性如表3-11所示。

表 3-11 info 参数的连接属性

属性名称	属性说明	属性值
user	表示创建连接的数据库用户。	属性类型：STRING
password	表示数据库用户的密码。	属性类型：STRING
useSSL	表示以SSL方式连接数据库。	属性类型：BOOLEAN 取值范围： • true表示以SSL方式连接数据库。 • false表示不以SSL方式连接数据库。
verifyServerCertificate	表示以SSL方式连接数据库时，客户端是否校验服务端证书。	属性类型：BOOLEAN 取值范围： • true表示客户端需要校验服务端证书。 • false表示客户端不校验服务端证书。 默认值：true
trustCertificateKeyStoreUrl	表示受信任的根证书密钥库的URL。	属性类型：STRING
trustCertificateKeyStoreType	表示受信任的根证书的密钥存储类型。	属性类型：STRING 默认值：JKS
trustCertificateKeyStorePassword	表示受信任的根证书密钥库的密码。	属性类型：STRING
clientCertificateKeyStoreUrl	表示客户端证书密钥库的URL。	属性类型：STRING
clientCertificateKeyStoreType	表示客户端证书的密钥存储类型。	属性类型：STRING 默认值：JKS

属性名称	属性说明	属性值
clientCertificateKeyStorePassword	表示客户端证书密钥库的密码。	属性类型：STRING
allowPublicKeyRetrieval	表示是否允许客户端从服务端获取公钥。	属性类型：BOOLEAN 取值范围： ● true表示允许客户端从服务端获取公钥。 ● false表示禁止客户端从服务端获取公钥。 默认值：false
rewriteBatchedStatements	表示是否可对批量执行的SQL重写。	属性类型：BOOLEAN 取值范围： ● true表示批量插入时，可将N条插入语句合并为一条：insert into TABLE_NAME values(values1, ..., valuesN), ..., (values1, ..., valuesN)。 ● false表示批量插入时，不重写SQL语句。 默认值：false
allowMultiQueries	表示是否可批量执行多条SQL语句。	属性类型：BOOLEAN 取值范围： ● true表示允许批量执行多条SQL语句。 ● false表示禁止批量执行多条SQL语句。 默认值：false
autoReconnect	表示客户端断开连接后是否会自动重连。	属性类型：BOOLEAN 取值范围： ● true表示客户端断连后，可自动重连。 ● false表示客户端断连后，不会自动重连。 默认值：false

属性名称	属性说明	属性值
useUnicode	表示是否使用Unicode字符编码。	属性类型：BOOLEAN 取值范围： • true表示客户端可使用unicode字符编码。 • false表示不使用unicode字符编码。 默认值：false
characterEncoding	表示设置字符集。 **说明** 需要保持客户端字符集与服务端一致。若未设置该参数，对于8.0.25及更早驱动版本，客户端将尝试使用服务端的默认字符集；对于8.0.26及更高驱动版本，客户端将使用utf8mb4的默认排序规则。	属性类型：STRING
connectTimeout	用于连接服务器操作系统的超时值。如果连接到服务器操作系统消耗的时间超过此值，则连接断开。当URL配置多IP时，表示连接单个IP的超时时间。	属性类型：INTEGER 属性单位：ms（毫秒） 取值范围：0 ~ 2147483647，0表示没有超时。 默认值：0
socketTimeout	用于socket读取操作的超时值。如果从服务器读取所消耗的时间超过此值，则连接关闭。	属性类型：INTEGER 属性单位：ms（毫秒） 取值范围：0 ~ 2147483647，0表示没有超时。 默认值：0

3.5.2.2.3 以常规方式连接

以下2个示例分别采用DriverManager.getConnection(String url, String user, String password)和DriverManager.getConnection(String url, Properties info)方法，以不加密方式（即常规方式）创建数据库连接。

示例1：连接数据库

```
//以下代码将获取数据库连接操作封装为一个接口，可通过给定用户名和密码来连接数据库。
public static Connection getConnect(String username, String passwd)
  {
    //驱动类。
    String driver = "com.mysql.jdbc.Driver";
    //数据库连接描述符。
    String sourceURL = "jdbc:mysql://$ip:$port/database?useSSL=false&allowPublicKeyRetrieval=true";
    Connection conn = null;

    try
    {
      //加载驱动。
      Class.forName(driver);
```

```
    }
    catch( Exception e )
    {
        e.printStackTrace();
        return null;
    }

    try
    {
         //创建连接。
        conn = DriverManager.getConnection(sourceURL, username, passwd);
        System.out.println("Connection succeed!");
    }
    catch(Exception e)
    {
        e.printStackTrace();
        return null;
    }

    return conn;
}
```

示例2：使用Properties对象作为参数建立连接

```
// 以下代码将使用Properties对象作为参数建立连接。
public static Connection getConnectUseProp(String username, String passwd)
{
    //驱动类。
    String driver = "com.mysql.jdbc.Driver";
    //数据库连接描述符。
    String sourceURL = "jdbc:mysql://$ip:$port/database?useSSL=false&allowPublicKeyRetrieval=true";
    Connection conn = null;
    Properties info = new Properties();

    try
    {
        //加载驱动。
        Class.forName(driver);
    }
    catch( Exception e )
    {
        e.printStackTrace();
        return null;
    }

    try
    {
         info.setProperty("user", username);
         info.setProperty("password", passwd);
         //创建连接。
         conn = DriverManager.getConnection(sourceURL, info);
         System.out.println("Connection succeed!");
    }
    catch(Exception e)
    {
        e.printStackTrace();
        return null;
    }

    return conn;
}
```

3.5.2.2.4 以 SSL 方式连接

用户通过JDBC连接GaussDB服务器时，可以通过开启SSL加密客户端和服务器之间的通讯，为敏感数据在Internet上的传输提供一种安全保障手段。

本节主要介绍应用程序通过JDBC如何采用SSL的方式对客户端进行配置（服务端配置请联系管理员）。

在使用本节所描述的方法前，默认用户已经获取了服务端和客户端所需要的证书和私钥文件。关于证书生成和获取，请联系管理员或参见Openssl相关文档和命令。

客户端配置

获取client.key、client.crt、cacert.pem证书文件，用于之后操作的配置。

设置服务端身份验证，需要使用到已生成的服务端cacert.pem证书文件。

步骤1 使用Java的keytool工具导入服务器证书。

```
>keytool -importcert -alias MyCACert -file cacert.pem -keystore truststore -storepass mypassword
```

步骤2 在Java或者应用程序中配置信任库。可以选择以下方法中的一种。

方法一：使用Java命令行参数。

```
-Djavax.net.ssl.trustStore=path_to_truststore_file
-Djavax.net.ssl.trustStorePassword=mypassword
```

方法二：在客户端代码中设置系统属性。

```
System.setProperty("javax.net.ssl.trustStore","path_to_truststore_file");
System.setProperty("javax.net.ssl.trustStorePassword","mypassword");
```

方法三：在连接属性中设置。

```
trustCertificateKeyStoreUrl=file:path_to_truststore_file
trustCertificateKeyStorePassword=mypassword
```

----结束

设置客户端身份验证，需要使用到已生成的clientcert.pem和clientkey.pem证书文件。

步骤1 将客户端证书文件client.crt和密钥client.key，转为pem格式。

```
>openssl x509 -in client.crt -out client.pem -outform PEM
>openssl rsa -in client.key -out clientkey.pem -outform PEM
```

步骤2 将客户端密钥和证书文件转换为PKCS12格式。

```
>openssl pkcs12 -export -in client.pem -inkey clientkey.pem -name "myclient" -passout pass:mypassword -
out clientkeystore.p12
```

步骤3 将客户端密钥和证书导入Java密钥库。

```
>keytool -importkeystore -srckeystore clientkeystore.p12 -srcstoretype pkcs12 -srcstorepass mypassword -
destkeystore keystore -deststoretype JKS -deststorepass mypassword
```

步骤4 在Java或者应用程序中配置信任库。可以选择以下三种方法中的一种。

方法一：使用Java命令行参数。

```
-Djavax.net.ssl.keyStore=path_to_keystore_file
-Djavax.net.ssl.keyStorePassword=mypassword
```

方法二：在客户端代码中设置系统属性。

```
System.setProperty("javax.net.ssl.keyStore","path_to_keystore_file");
System.setProperty("javax.net.ssl.keyStorePassword","mypassword");
```

方法三：在连接属性中设置。

```
clientCertificateKeyStoreUrl=file:path_to_truststore_file
clientCertificateKeyStorePassword=mypassword
```

----结束

示例


```
// 认证用的用户名和密码直接写到代码中有很大的安全风险，建议在配置文件或者环境变量中存放（密码应密文存放，使用时解密），确保安全。
// 本示例以用户名和密码保存在环境变量中为例，运行本示例前请先在本地环境中设置环境变量（环境变量名称请根据自身情况进行设置）EXAMPLE_USERNAME_ENV和EXAMPLE_PASSWORD_ENV。
import java.sql.Connection;
import java.sql.DriverManager;
import java.util.Properties;

public class SSL{
    public static void main(String[] args) {
        String userName = System.getenv("EXAMPLE_USERNAME_ENV");
        String password = System.getenv("EXAMPLE_PASSWORD_ENV");
        String urls = "jdbc:gaussdb://$ip.$port/database?useSSL=true&verifyServerCertificate=true"+
            "&trustCertificateKeyStoreUrl=file:path_to_truststore_file" +
            "&trustCertificateKeyStorePassword=mypassword"+
            "&clientCertificateKeyStoreUrl=file:path_to_truststore_file" +
            "&clientCertificateKeyStorePassword=mypassword";


        // 创建数据库连接。
        try {
            Class.forName("com.mysql.jdbc.Driver").newInstance();
        } catch (Exception e) {
            e.printStackTrace();
        }
        try {
            Connection conn;
            conn = DriverManager.getConnection(urls,userName, password);
            conn.close();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

3.5.2.3 执行 SQL 语句

执行普通 SQL 语句

应用程序通过执行SQL语句来操作数据库的数据（不用传递参数的语句），需要按以下步骤执行。

支持对XML类型数据进行SELECT、UPDATE、INSERT、DELETE等操作。

步骤1 调用Connection的createStatement方法创建语句对象。

```
// 认证用的用户名和密码直接写到代码中有很大的安全风险，建议在配置文件或者环境变量中存放（密码应密文存放，使用时解密），确保安全。
// 本示例以用户名和密码保存在环境变量中为例，运行本示例前请先在本地环境中设置环境变量（环境变量名称请根据自身情况进行设置）EXAMPLE_USERNAME_ENV和EXAMPLE_PASSWORD_ENV。
String userName = System.getenv("EXAMPLE_USERNAME_ENV");
String password = System.getenv("EXAMPLE_PASSWORD_ENV");
String sourceURL = "jdbc:mysql://$ip.$port/database";
Connection conn = DriverManager.getConnection(sourceURL,userName,password);
Statement stmt = conn.createStatement();
```

步骤2 调用Statement的executeUpdate方法执行SQL语句。

```
int rc = stmt.executeUpdate("CREATE TABLE customer_t1(c_customer_sk INTEGER, c_customer_name VARCHAR(32));");
```

步骤3 关闭语句对象。

```
stmt.close();
```

----结束

执行预编译 SQL 语句

预编译语句是只编译和优化一次，可以通过设置不同的参数值多次使用。由于已经预先编译好，后续使用会减少执行时间。因此，如果多次执行一条语句，请选择使用预编译语句。可以按以下步骤执行：

步骤1 调用Connection的prepareStatement方法创建预编译语句对象。

```
PreparedStatement pstmt = con.prepareStatement("UPDATE customer_t1 SET c_customer_name = ? WHERE c_customer_sk = 1");
```

步骤2 调用PreparedStatement的setShort设置参数。

```
pstmt.setShort(1, (short)2);
```

步骤3 调用PreparedStatement的executeUpdate方法执行预编译SQL语句。

```
int rowcount = pstmt.executeUpdate();
```

步骤4 调用PreparedStatement的close方法关闭预编译语句对象。

```
pstmt.close();
```

----结束

执行批处理

用一条预处理语句处理多条相似的数据，数据库只创建一次执行计划，节省了语句的编译和优化时间。可以按如下步骤执行：

步骤1 调用Connection的prepareStatement方法创建预编译语句对象。

```
// 认证用的用户名和密码直接写到代码中有很大的安全风险，建议在配置文件或者环境变量中存放（密码应密文存放，使用时解密），确保安全。
// 本示例以用户名和密码保存在环境变量中为例，运行本示例前请先在本地环境中设置环境变量（环境变量名称请根据自身情况进行设置）EXAMPLE_USERNAME_ENV和EXAMPLE_PASSWORD_ENV。
String userName = System.getenv("EXAMPLE_USERNAME_ENV");
String password = System.getenv("EXAMPLE_PASSWORD_ENV");
String sourceURL = "jdbc:mysql://$ip:$port/database";
Connection conn = DriverManager.getConnection(sourceURL,userName,password);
PreparedStatement pstmt = conn.prepareStatement("INSERT INTO customer_t1 VALUES (?)");
```

步骤2 针对每条数据要调用对应接口设置参数，以及调用addBatch将SQL语句添加到批处理中。

```
pstmt.setShort(1, (short)2);
pstmt.addBatch();
```

步骤3 调用PreparedStatement的executeBatch方法执行批处理。

```
int[] rowcount = pstmt.executeBatch();
```

步骤4 调用PreparedStatement的close方法关闭预编译语句对象。

```
pstmt.close();
```

说明

在实际的批处理过程中，通常不终止批处理程序的执行，否则会降低数据库的性能。因此在批处理程序时，应该关闭自动提交功能，每几行提交一次。关闭自动提交功能的语句为：conn.setAutoCommit(false);。

----结束

说明

暂不支持服务端PBE，应用程序在连接参数中设置useServerPrepStmts参数为true，则不生效。

3.5.2.4 处理结果集

设置结果集类型

不同类型的结果集有各自的应用场景，应用程序需要根据实际情况选择相应的结果集类型。在执行SQL语句过程中，需要先创建相应的语句对象，而部分创建语句对象的方法提供了设置结果集类型的功能。具体的参数设置如表3-12所示。涉及的Connection的方法如下：

```
//创建一个Statement对象，该对象将生成具有给定类型和并发性的ResultSet对象。
createStatement(int resultSetType, int resultSetConcurrency);

//创建一个PreparedStatement对象，该对象将生成具有给定类型和并发性的ResultSet对象。
prepareStatement(String sql, int resultSetType, int resultSetConcurrency);

//创建一个CallableStatement对象，该对象将生成具有给定类型和并发性的ResultSet对象。
prepareCall(String sql, int resultSetType, int resultSetConcurrency);
```

表 3-12 结果集类型

参数	描述
resultSetType	表示结果集的类型，具体有以下类型： ● ResultSet.TYPE_FORWARD_ONLY：ResultSet只能向前移动。是缺省值。 ● ResultSet.TYPE_SCROLL_SENSITIVE：在修改后重新滚动到修改所在行，可以看到修改后的结果。 ● ResultSet.TYPE_SCROLL_INSENSITIVE：对可修改例程所做的编辑不进行显示。 **说明** 结果集从数据库中读取了数据之后，即使类型是ResultSet.TYPE_SCROLL_SENSITIVE，也不会看到由其他事务在这之后引起的改变。调用ResultSet的refreshRow()方法，可进入数据库并从其中取得当前游标所指记录的最新数据。
resultSetConcurrency	表示结果集的并发，具体有以下类型： ● ResultSet.CONCUR_READ_ONLY：如果不从结果集中的数据建立一个新的更新语句，不能对结果集中的数据进行更新。 ● ResultSet.CONCUR_UPDATEABLE：可改变的结果集。对于可滚动的结果集，可对结果集进行适当的改变。

在结果集中定位

ResultSet对象具有指向其当前数据行的光标。最初，光标被置于第一行之前，next方法将光标移动到下一行。因为该方法在ResultSet对象没有下一行时返回false，所以可以在while循环中使用它来迭代结果集。但对于可滚动的结果集，JDBC驱动程序提供更多的定位方法，使ResultSet指向特定的行。定位方法如表3-13所示。

表 3-13 在结果集中定位的方法

方法	描述
next()	把ResultSet向下移动一行。
previous()	把ResultSet向上移动一行。
beforeFirst()	把ResultSet定位到第一行之前。
afterLast()	把ResultSet定位到最后一行之后。
first()	把ResultSet定位到第一行。
last()	把ResultSet定位到最后一行。
absolute(int row)	把ResultSet移动到参数指定的行数。
relative(int rows)	rows为正数表示把ResultSet向下移动rows行，rows为负数表示把ResultSet向上移动（-rows）行。

获取结果集中光标的位置

对于可滚动的结果集，可调用定位方法来改变光标的位置。JDBC驱动程序提供了获取结果集中光标所处位置的方法。获取光标位置的方法如表3-14所示。

表 3-14 获取结果集光标的位置

方法	描述
isFirst()	是否在第一行。
isLast()	是否在最后一行。
isBeforeFirst()	是否在第一行之前。
isAfterLast()	是否在最后一行之后。
getRow()	获取当前在第几行。

获取结果集中的数据

ResultSet对象提供了丰富的方法，以获取结果集中的数据。获取数据常用的方法如表3-15所示，其他方法请参见JDK官方文档。

表 3-15 ResultSet 对象的常用方法

方法	描述
getInt(int columnIndex)	按列标获取int型数据。
getInt(String columnLabel)	按列名获取int型数据。

方法	描述
getString(int columnIndex)	按列标获取String型数据。
getString(String columnLabel)	按列名获取String型数据。
getDate(int columnIndex)	按列标获取Date型数据。
getDate(String columnLabel)	按列名获取Date型数据。

3.5.2.5 关闭数据库连接

在使用数据库连接完成相应的数据操作后，需要关闭数据库连接。

关闭数据库连接可以直接调用close方法。

```
// 认证用的用户名和密码直接写到代码中有很大的安全风险，建议在配置文件或者环境变量中存放（密码应密文存放，使用时解密），确保安全。
// 本示例以用户名和密码保存在环境变量中为例，运行本示例前请先在本地环境中设置环境变量（环境变量名称请根据自身情况进行设置）EXAMPLE_USERNAME_ENV和EXAMPLE_PASSWORD_ENV。
String userName = System.getenv("EXAMPLE_USERNAME_ENV");
String password = System.getenv("EXAMPLE_PASSWORD_ENV");
String sourceURL = "jdbc:mysql://$ip:$port/database";
Connection conn = DriverManager.getConnection(sourceURL, userName, password);
conn.close();
```

3.5.3 示例

3.5.3.1 常用操作示例

示例 1 创建数据库连接、创建表、插入数据

此示例将演示如何基于GaussDB提供的JDBC接口开发应用程序。执行示例前，需要先加载驱动，驱动加载方法请参见**加载驱动**。

```
// DBTest.java
// 演示基于JDBC开发的主要步骤，会涉及创建数据库连接、创建表、插入数据等。
// 认证用的用户名和密码直接写到代码中有很大的安全风险，建议在配置文件或者环境变量中存放（密码应密文存放，使用时解密），确保安全。
// 本示例以用户名和密码保存在环境变量中为例，运行本示例前请先在本地环境中设置环境变量（环境变量名称请根据自身情况进行设置）EXAMPLE_USERNAME_ENV和EXAMPLE_PASSWORD_ENV。

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.SQLException;
import java.sql.Statement;
import java.sql.CallableStatement;
import java.sql.Types;

public class DBTest {

 // 创建数据库连接。
 public static Connection GetConnection(String username, String passwd) {
   String driver = "com.mysql.jdbc.Driver";
   String sourceURL = "jdbc:mysql://$ip:$port/database?useSSL=false&allowPublicKeyRetrieval=true";
   Connection conn = null;
   try {
     // 加载数据库驱动。
     Class.forName(driver).newInstance();
```



```
  } catch (Exception e) {
    e.printStackTrace();
    return null;
  }

  try {
    // 创建数据库连接。
    conn = DriverManager.getConnection(sourceURL, username, passwd);
    System.out.println("Connection succeed!");
  } catch (Exception e) {
    e.printStackTrace();
    return null;
  }

  return conn;
};

// 执行普通SQL语句，创建customer_t1表。
public static void CreateTable(Connection conn) {
  Statement stmt = null;
  try {
    stmt = conn.createStatement();

    // 执行普通SQL语句。
    int rc = stmt
        .executeUpdate("CREATE TABLE customer_t1(c_customer_sk INTEGER, c_customer_name
VARCHAR(32));");

    stmt.close();
  } catch (SQLException e) {
    if (stmt != null) {
      try {
        stmt.close();
      } catch (SQLException e1) {
        e1.printStackTrace();
      }
    }
    e.printStackTrace();
  }
}

// 执行预处理语句，批量插入数据。
public static void BatchInsertData(Connection conn) {
  PreparedStatement pst = null;

  try {
    // 生成预处理语句。
    pst = conn.prepareStatement("INSERT INTO customer_t1 VALUES (?,?)");
    for (int i = 0; i < 3; i++) {
      // 添加参数。
      pst.setInt(1, i);
      pst.setString(2, "data " + i);
      pst.addBatch();
    }
    // 执行批处理。
    pst.executeBatch();
    pst.close();
  } catch (SQLException e) {
    if (pst != null) {
      try {
        pst.close();
      } catch (SQLException e1) {
      e1.printStackTrace();
      }
    }
    e.printStackTrace();
  }
}
```



```
 // 执行预编译语句，更新数据。
 public static void ExecPreparedSQL(Connection conn) {
   PreparedStatement pstmt = null;
   try {
     pstmt = conn
         .prepareStatement("UPDATE customer_t1 SET c_customer_name = ? WHERE c_customer_sk = 1");
     pstmt.setString(1, "new Data");
     int rowcount = pstmt.executeUpdate();
     pstmt.close();
   } catch (SQLException e) {
     if (pstmt != null) {
       try {
         pstmt.close();
       } catch (SQLException e1) {
         e1.printStackTrace();
       }
     }
     e.printStackTrace();
   }
 }

 /**
  * 主程序，逐步调用各静态方法。
  * @param args
 */
 public static void main(String[] args) {
   // 创建数据库连接。
   String userName = System.getenv("EXAMPLE_USERNAME_ENV");
   String password = System.getenv("EXAMPLE_PASSWORD_ENV");
   Connection conn = GetConnection(userName, password);

   // 创建表。
   CreateTable(conn);

   // 批量插入数据。
   BatchInsertData(conn);

   // 执行预编译语句，更新数据。
   ExecPreparedSQL(conn);

   // 关闭数据库连接。
   try {
     conn.close();
   } catch (SQLException e) {
     e.printStackTrace();
   }

 }

}
```

上述示例的运行结果为：

```
Connection succeed!
```

示例 2 批量查询

此示例主要使用setFetchSize调整客户端内存使用，原理是通过数据库游标来分批获取服务端数据，但会加大网络交互，可能会损失部分性能。

由于游标事务内有效，故需要先关闭自动提交，最后执行手动提交。

```
// 认证用的用户名和密码直接写到代码中有很大的安全风险，建议在配置文件或者环境变量中存放（密码应密文存放，使用时解密），确保安全。
// 本示例以用户名和密码保存在环境变量中为例，运行本示例前请先在本地环境中设置环境变量（环境变量名称请根据自身情况进行设置）EXAMPLE_USERNAME_ENV和EXAMPLE_PASSWORD_ENV。

import java.sql.Connection;
```



```
import java.sql.DriverManager;
import java.sql.Statement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.PreparedStatement;

public class Batch {
    public static void main(String[] args) throws SQLException {
        String driver = "com.mysql.jdbc.Driver";
        String username = System.getenv("EXAMPLE_USERNAME_ENV");
        String passwd = System.getenv("EXAMPLE_PASSWORD_ENV");
        String sourceURL = "jdbc:mysql://$ip:$port/database?useSSL=false&allowPublicKeyRetrieval=true";
        Connection conn = null;

        try {
            // 加载数据库驱动。
            Class.forName(driver).newInstance();
        } catch (Exception e) {
            e.printStackTrace();
        }

        try {
            // 创建数据库连接。
            conn = DriverManager.getConnection(sourceURL, username, passwd);
            System.out.println("Connection succeed!");
        } catch (Exception e) {
            e.printStackTrace();
        }

        // 关闭自动提交。
        conn.setAutoCommit(false);

        // 新建表。
        Statement st = conn.createStatement();
        st.execute("create table mytable (cal1 int);");

        // 表中插入200行数据。
        PreparedStatement pstm = conn.prepareStatement("insert into mytable values (?)");
        for (int i = 0; i < 200; i++) {
            pstm.setInt(1, i + 1);
            pstm.addBatch();
        }
        pstm.executeBatch();
        conn.commit();
        pstm.close();

        // 打开游标，每次获取50行数据。
        st.setFetchSize(50);
        int fetchCount = 0;
        ResultSet rs = st.executeQuery("SELECT * FROM mytable");
        while (rs.next()) {
            fetchCount++;
        }
        System.out.println(fetchCount == 200);
        conn.commit();
        rs.close();

        // 关闭服务器游标。
        st.setFetchSize(0);
        fetchCount = 0;
        rs = st.executeQuery("SELECT * FROM mytable");
        while (rs.next()) {
            fetchCount++;
        }
        System.out.println(fetchCount == 200);
        conn.commit();
        rs.close();

        // Close the statement.
```



```
        st.close();
        conn.close();

    }
}
```

上述示例的运行结果为：

```
Connection succeed!
true
true
```

执行完毕后可执行如下命令恢复自动提交。

```
conn.setAutoCommit(true);
```

示例 3 常用数据类型使用

```
// 认证用的用户名和密码直接写到代码中有很大的安全风险，建议在配置文件或者环境变量中存放（密码应密文存放，使用时解密），确保安全。
// 本示例以用户名和密码保存在环境变量中为例，运行本示例前请先在本地环境中设置环境变量（环境变量名称请根据自身情况进行设置）EXAMPLE_USERNAME_ENV和EXAMPLE_PASSWORD_ENV。

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.Statement;
import java.sql.CallableStatement;
import java.sql.SQLException;
import java.sql.PreparedStatement;
import java.sql.Types;

public class Type {
    public static void main(String[] args) throws SQLException {
        String driver = "com.mysql.jdbc.Driver";
        String username = System.getenv("EXAMPLE_USERNAME_ENV");
        String passwd = System.getenv("EXAMPLE_PASSWORD_ENV");
        String sourceURL = "jdbc:mysql://$ip:$port/database?useSSL=false&allowPublicKeyRetrieval=true";
        Connection conn = null;

        try {
            // 加载数据库驱动。
            Class.forName(driver).newInstance();
        } catch (Exception e) {
            e.printStackTrace();
        }

        try {
            // 创建数据库连接。
            conn = DriverManager.getConnection(sourceURL, username, passwd);
            System.out.println("Connection succeed!");
        } catch (Exception e) {
            e.printStackTrace();
        }

        Statement st = conn.createStatement();
        // bit类型使用示例。
        st.execute("create table if not exists t_bit(col_bit bit)");
        PreparedStatement pstm1 = conn.prepareStatement("insert into t_bit values(?)");
        pstm1.setBoolean(1,true);
        pstm1.execute();
        pstm1.close();
        ResultSet rs1 = st.executeQuery(" select col_bit from t_bit;");
        while (rs1.next()) {
            System.out.println(rs1.getBoolean(1));
        }
        rs1.close();

        // float8类型使用示例。
        st.execute("create table if not exists t_float(col1 float8)");
```



```
        PreparedStatement pstm2 = conn.prepareStatement("insert into t_float values(?)");
        pstm2.setDouble(1, 123456.123);
        pstm2.execute();
        pstm2.close();
        ResultSet rs2 = st.executeQuery(" select col1 from t_float;");
        while (rs2.next()) {
            System.out.println(rs2.getDouble(1));
        }
        rs2.close();
        st.close();

        // 关闭数据库连接。
        try {
            conn.close();
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
}
```

上述示例的运行结果为：

```
Connection succeed!
false
123456.123
```

示例 4 使用流式读功能

流式读功能：读取数据时，服务端一次获取全部数据，发送到客户端socket缓冲区中，缓冲区占满则暂停，有空余则继续向缓冲区中发送数据，同时JVM逐行从缓冲区中读取数据。

优势是处理结果快，不会造成JVM内存溢出。劣势是只能向后遍历，数据处理完毕之前或者statement关闭之前，当前连接不能执行其他操作。

```
// 认证用的用户名和密码直接写到代码中有很大的安全风险，建议在配置文件或者环境变量中存放（密码应密文存放，使用时解密），确保安全。
// 本示例以用户名和密码保存在环境变量中为例，运行本示例前请先在本地环境中设置环境变量（环境变量名称请根据自身情况进行设置）EXAMPLE_USERNAME_ENV和EXAMPLE_PASSWORD_ENV。

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;

public class Stream {
    // 创建数据库连接。
    public static Connection getConnection(String username, String passwd) {
        String driver = "com.mysql.jdbc.Driver";
        String sourceURL = "jdbc:mysql://$ip:$port/database?useSSL=false&allowPublicKeyRetrieval=true";
        Connection conn = null;
        try {
            //加载驱动。
            Class.forName(driver);
        } catch (Exception e) {
            e.printStackTrace();
            return null;
        }
        try {
            //创建连接。
            conn = DriverManager.getConnection(sourceURL, username, passwd);
            System.out.println("Connection succeed!");
        } catch (Exception e) {
            e.printStackTrace();
            return null;
        }
```



```
    return conn;
}

// 执行普通SQL语句，创建t_user表。
public static void CreateTable(Connection conn) {
    Statement stmt = null;
    try {
        stmt = conn.createStatement();

        //执行普通SQL语句。
        stmt.executeUpdate("DROP TABLE IF EXISTS t_user");
        stmt.executeUpdate("CREATE TABLE t_user(id int, name VARCHAR(20));");
        stmt.close();
    } catch (SQLException e) {
        if (stmt != null) {
            try {
                stmt.close();
            } catch (SQLException e1) {
                e1.printStackTrace();
            }
        }
        e.printStackTrace();
    }
}

//执行预处理语句，批量插入数据。
public static void BatchInsertData(Connection conn) {
    PreparedStatement pst = null;

    try {
        //生成预处理语句。
        pst = conn.prepareStatement("INSERT INTO t_user VALUES (?,?)");
        for (int i = 0; i < 20; i++) {
            //添加参数。
            pst.setInt(1, i + 1);
            pst.setString(2, "name " + (i + 1));
            pst.addBatch();
        }
        //执行批处理。
        pst.executeBatch();
        pst.close();
    } catch (SQLException e) {
        if (pst != null) {
            try {
                pst.close();
            } catch (SQLException e1) {
                e1.printStackTrace();
            }
        }
        e.printStackTrace();
    }
}

// 开启流式读，查询t_user表中内容。
public static void StreamingQuery(Connection conn) {
    PreparedStatement pst = null;
    ResultSet resultSet = null;

    try {
        // 查询表t_user中的所有值。
        pst = conn.prepareStatement("SELECT * FROM t_user");
        pst.setFetchSize(Integer.MIN_VALUE);
        resultSet = pst.executeQuery();
        while (resultSet.next()) {
            System.out.println(resultSet.getInt(1));
        }
    } catch (SQLException e) {
        throw new RuntimeException(e);
    } finally {
```



```
            if (resultSet != null) {
                try {
                    resultSet.close();
                } catch (SQLException e) {
                    throw new RuntimeException(e);
                }
            }

            if (pst != null) {
                try {
                    pst.close();
                } catch (SQLException e) {
                    throw new RuntimeException(e);
                }
            }
        }
    }

    public static void main(String[] args) throws Exception {
        String userName = System.getenv("EXAMPLE_USERNAME_ENV");
        String password = System.getenv("EXAMPLE_PASSWORD_ENV");

        Connection conn = getConnection(userName, password);

        CreateTable(conn);

        BatchInsertData(conn);

        StreamingQuery(conn);

        //关闭数据库连接。
        try {
            conn.close();
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
}
```

上述示例的运行结果为：

```
Connection succeed!
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
```

须知

使用流式读功能时，结果集使用完之后，需要执行resultSet.close()或者statement.close()操作，否则当前连接将不可用。

3.5.4 JDBC 接口参考

JDBC接口是一套提供给用户的API方法，本节将对部分常用接口做具体描述，若涉及其他接口可参考JDK1.8（软件包）/JDBC4.2中相关内容。

3.5.4.1 java.sql.Connection

java.sql.Connection是数据库连接接口，具体信息如表3-16所示。

表 3-16 对 java.sql.Connection 接口的支持情况

方法名	描述	返回值类型	throws	是否支持
abort(Executor executor)	终止打开的连接。	VOID	SQLException	Yes
clearWarnings()	清除为此Connection对象报告的所有警告。	VOID	SQLException	Yes
close()	立即释放此Connection对象的数据库和JDBC资源，而不是等待系统自动释放。	VOID	SQLException	Yes
commit()	使自上次提交/回滚以来所做的所有更改永久化，并释放此Connection对象当前持有的任何数据库锁。仅当禁用了自动提交模式时，才应使用此方法。	VOID	SQLException	Yes
createArrayOf(String typeName, Object[] elements)	用于创建ARRAY对象的工厂方法。	ARRAY	SQLException	Yes
createBlob()	构造一个实现BLOB接口的对象。返回的对象最初不包含数据。BLOB接口的setBinaryStream和setBytes方法可以用于向BLOB添加数据。	BLOB	SQLException	Yes
createClob()	构造一个实现CLOB接口的对象。返回的对象最初不包含数据。CLOB接口的setAsciiStream、setCharacterStream和setString方法可以用于向CLOB添加数据。	CLOB	SQLException	Yes
createSQLXML()	构造一个实现SQLXML接口的对象。返回的对象最初不包含数据。可以使用SQLXML接口的createXmlStreamWriter对象和setString方法向SQLXML对象添加数据。	SQLXML	SQLException	Yes

方法名	描述	返回值类型	throws	是否支持
createStatement()	创建一个用于将SQL语句发送到数据库的语句对象。	STATEMENT	SQLException	Yes
createStatement(int resultSetType, int resultSetConcurrency)	创建一个语句对象，该对象将生成具有给定类型和并发性的ResultSet对象。	STATEMENT	SQLException	Yes
createStatement(int resultSetType, int resultSetConcurrency, int resultSetHoldability)	创建一个语句对象，该对象将生成具有给定类型和并发性的ResultSet对象。	STATEMENT	SQLException	Yes
getAutoCommit()	检索此连接对象的当前自动提交模式。	BOOLEAN	SQLException	Yes
getCatalog()	检索此Connection对象的当前目录名称。	STRING	SQLException	Yes
getClientInfo()	返回一个列表，其中包含驱动程序支持的每个客户端信息属性的名称和当前值。如果客户端信息属性尚未设置且没有默认值，则该属性的值可能为NULL。	PROPERTIES	SQLException	Yes
getClientInfo(String name)	返回由name指定的客户端信息属性的值。如果尚未设置指定的客户端信息属性，并且没有默认值，此方法可能会返回NULL。如果驱动程序不支持指定的客户端信息属性名称，此方法也将返回NULL。	STRING	SQLException	Yes
getHoldability()	检索使用此Connection对象创建的ResultSet对象的当前保持性。	INT	SQLException	Yes
getMetaData()	检索包含有关此Connection对象表示连接的数据库的元数据的DatabaseMetaData对象。元数据包括有关数据库表、其支持的SQL语法、其存储过程、此连接的功能等的信息。	DATABASEMETADATA	SQLException	Yes
getNetworkTimeout()	检索驱动程序等待数据库请求完成的毫秒数。如果超过限制，将导致SQLException。	INT	SQLException	Yes

方法名	描述	返回值类型	throws	是否支持
getSchema()	检索此Connection对象的当前架构名称。	STRING	SQLException	Yes
getTransactionIsolation()	检索此Connection对象的当前事务隔离级别。	INT	SQLException	Yes
getTypeMap()	检索与此Connection对象关联的Map对象。仅当应用程序添加了条目，否则返回的类型映射将为空。	MAP<STRING, CLASS<?>>	SQLException	Yes
getWarnings()	检索此Connection对象上的调用报告的第一个警告。如果有多个警告，则后续警告将链接到第一个警告，并可以通过对先前检索的警告调用方法SQLWarning.getNextWarning来检索。	SQLWARNING	SQLException	Yes
isClosed()	检索此Connection对象是否已关闭。如果已对连接调用了方法关闭，或者发生了某些重大错误，则连接将关闭。只有在调用closection.Close方法之后调用此方法时，才保证返回true。	BOOLEAN	SQLException	Yes
isReadOnly()	检索此Connection对象是否处于只读模式。	BOOLEAN	SQLException	Yes
isValid(int timeout)	如果连接尚未关闭并且仍然有效，则返回true。驱动程序应提交对连接的查询，或使用其他机制，以肯定地验证连接在调用此方法时仍然有效。	BOOLEAN	SQLException	Yes
nativeSQL(String sql)	将给定的SQL语句转换为系统的本机SQL语法。驱动程序可以在发送JDBC SQL语法之前将其转换为其系统的本机SQL语法。此方法返回驱动程序本应发送的语句的本机形式。	STRING	SQLException	Yes
prepareCall(String sql)	创建用于调用数据库存储过程的CallableStatement对象。CallableStatement对象提供了用于设置其IN和OUT参数的方法，以及用于执行对存储过程调用的方法。	CALLABLESTATEMENT	SQLException	No

方法名	描述	返回值类型	throws	是否支持
prepareCall(String sql, int resultSetType, int resultSetConcurrency)	创建一个CallableStatement对象，该对象将生成具有给定类型和并发性的ResultSet对象。此方法与上面的准备呼叫方法相同，但它允许覆盖默认的结果集类型和并发。	CALLABLESTATEMENT	SQLException	No
prepareCall(String sql, int resultSetType, int resultSetConcurrency, int resultSetHoldability)	创建一个CallableStatement对象，该对象将生成具有给定类型和并发性的ResultSet对象。此方法与上面的准备呼叫方法相同，但它允许覆盖默认的结果集类型、结果集并发类型和保持性。	CALLABLESTATEMENT	SQLException	No
prepareStatement(String sql)	创建一个用于将参数化SQL语句发送到数据库的准备语句对象。	PREPAREDSTATEMENT	SQLException	Yes
prepareStatement(String sql, int autoGeneratedKeys)	创建一个默认的准备语句对象，该对象能够检索自动生成的键。给定的常量反馈驱动程序，它是否应使自动生成的密钥可供检索。如果SQL语句不是INSERT语句，也不是能够返回自动生成键的SQL语句，则将忽略此参数。	PREPAREDSTATEMENT	SQLException	Yes
prepareStatement(String sql, int[] columnIndexes)	创建一个默认的准备语句对象，该对象能够返回由给定数组指定的自动生成的键。此数组包含目标表中包含应可用的自动生成键的列的索引。如果SQL语句不是INSERT语句，也不是能够返回自动生成键的SQL语句，驱动程序将忽略数组。	PREPAREDSTATEMENT	SQLException	Yes
prepareStatement(String sql, int resultSetType, int resultSetConcurrency)	创建一个准备语句对象，该对象将生成具有给定类型和并发性的ResultSet对象。此方法与上面的准备语句方法相同，但它允许覆盖默认的结果集类型和并发。	PREPAREDSTATEMENT	SQLException	Yes

方法名	描述	返回值类型	throws	是否支持
prepareStatement(String sql, int resultSetType, int resultSetConcurrency, int resultSetHoldability)	创建一个准备语句对象，该对象将生成具有给定类型、并发性和保持性的ResultSet对象。	PREPAREDSTATEMENT	SQLException	Yes
prepareStatement(String sql, String[] columnNames)	创建一个默认的准备语句对象，该对象能够返回由给定数组指定的自动生成的键。此数组包含目标表中包含应返回的自动生成键的列的名称。如果SQL语句不是INSERT语句，也不是能够返回自动生成键的SQL语句，驱动程序将忽略数组。	PREPAREDSTATEMENT	SQLException	Yes
releaseSavepoint(Savepoint savepoint)	从当前事务中删除指定的Savepoint和后续Savepoint对象。删除保存点后对保存点的任何引用都将引发SQLException。	VOID	SQLException	Yes
rollback()	撤销在当前事务中所做的所有更改，并释放此Connection对象当前持有的任何数据库锁。仅当禁用自动提交模式时，才应使用此方法。	VOID	SQLException	Yes
rollback(Savepoint savepoint)	撤销设置给定Savepoint对象后所做的所有更改。仅当禁用自动提交模式时，才应使用此方法。	VOID	SQLException	Yes
setAutoCommit(boolean autoCommit)	将此连接的自动提交模式设置为给定状态。如果连接处于自动提交模式，则其所有SQL语句都将作为单个事务执行和提交。否则，其SQL语句将分组为事务，这些事务由对方法提交或方法回滚的调用终止。默认情况下，新连接处于自动提交模式。	VOID	SQLException	Yes
setCatalog(String catalog)	设置此Connection对象的当前目录名称。	VOID	SQLException	Yes
setClientInfo(Properties properties)	设置连接的客户端信息属性的值。Properties对象包含要设置的客户端信息属性的名称和值。属性列表中包含的客户端信息属性集替换连接上的当前客户端信息属性集。如果当前在连接上设置的属性不在属性列表中，则将清除该属性。	VOID	SQLClientInfoException	Yes

方法名	描述	返回值类型	throws	是否支持
setClientInfo(String name,String value)	将name指定的客户端info属性的值设置为value指定的值。	VOID	SQLClientInfoException	Yes
setHoldability(int holdability)	将使用此Connection对象创建的ResultSet对象的默认保持性更改为给定的保持性。ResultSet对象的默认保持性可以通过调用DatabaseMetaData.getResultSetHoldability来确定。	VOID	SQLException	Yes
setNetworkTimeout(Executor executor, int milliseconds)	设置连接或从连接创建的对象等待数据库回复任何一个请求的最长时间。如果任何请求都未应答，则等待方法将返回SQLException，并且从Connection创建的Connection或对象将标记为已关闭。除关闭、isCabled或Connection.isValid方法外，对对象的任何后续使用都将引发SQLException。	VOID	SQLException	Yes
setReadOnly(boolean readOnly)	将此连接置于只读模式，作为驱动程序的提示，以启用数据库优化。	VOID	SQLException	Yes
setSavepoint()	在当前事务中创建一个未命名的保存点，并返回表示它的新保存点对象。	SAVEPOINT	SQLException	Yes
setSavepoint(String name)	在当前事务中使用给定名称创建保存点，并返回表示它的新保存点对象。	SAVEPOINT	SQLException	Yes
setSchema(String schema)	将给定的架构名称设置为访问。	VOID	SQLException	Yes
setTransactionIsolation(int level)	尝试将此Connection对象的事务隔离级别更改为给定的级别。接口Connection中定义的常量是可能的事务隔离级别。	VOID	SQLException	Yes
setTypeMap(Map<String,Class<?>> map)	安装给定的TypeMap对象作为此Connection对象的类型映射。类型映射将用于SQL结构化类型和不同类型的自定义映射。	VOID	SQLException	Yes

3.5.4.2 java.sql.DatabaseMetaData

java.sql.DatabaseMetaData是数据库对象定义接口，具体信息如**表3-17**所示。

表 3-17 对 java.sql.DatabaseMetaData 的支持情况

方法名	返回值类型	是否支持
allProceduresAreCallable()	BOOLEAN	Yes
allTablesAreSelectable()	BOOLEAN	Yes
autoCommitFailureClosesAllResultSets()	BOOLEAN	Yes
dataDefinitionCausesTransactionCommit()	BOOLEAN	Yes
dataDefinitionIgnoredInTransactions()	BOOLEAN	Yes
deletesAreDetected(int type)	BOOLEAN	Yes
doesMaxRowSizeIncludeBlobs()	BOOLEAN	Yes
generatedKeyAlwaysReturned()	BOOLEAN	Yes
getBestRowIdentifier(String catalog, String schema, String table, int scope, boolean nullable)	RESULTSET	Yes
getCatalogs()	RESULTSET	Yes
getCatalogSeparator()	STRING	Yes
getCatalogTerm()	STRING	Yes
getClientInfoProperties()	RESULTSET	Yes
getColumnPrivileges(String catalog, String schema, String table, String columnNamePattern)	RESULTSET	No
getColumns(String catalog, String schemaPattern, String tableNamePattern, String columnNamePattern)	RESULTSET	No
getConnection()	CONNECTION	Yes
getCrossReference(String parentCatalog, String parentSchema, String parentTable, String foreignCatalog, String foreignSchema, String foreignTable)	RESULTSET	No

方法名	返回值类型	是否支持
getDatabaseMajorVersion()	INT	Yes
getDatabaseMinorVersion()	INT	Yes
getDatabaseProductName()	STRING	Yes
getDatabaseProductVersion()	STRING	Yes
getDefaultTransactionIsolation()	INT	Yes
getDriverMajorVersion()	INT	Yes
getDriverMinorVersion()	INT	Yes
getDriverName()	STRING	Yes
getDriverVersion()	STRING	Yes
getExportedKeys(String catalog, String schema, String table)	RESULTSET	No
getExtraNameCharacters()	STRING	Yes
getFunctionColumns(String catalog, String schemaPattern, String functionNamePattern, String columnNamePattern)	RESULTSET	No
getFunctions(String catalog, String schemaPattern, String functionNamePattern)	RESULTSET	No
getIdentifierQuoteString()	STRING	Yes
getImportedKeys(String catalog, String schema, String table)	RESULTSET	Yes
getIndexInfo(String catalog, String schema, String table, boolean unique, boolean approximate)	RESULTSET	Yes
getJDBCMajorVersion()	INT	Yes
getJDBCMinorVersion()	INT	Yes
getMaxBinaryLiteralLength()	INT	Yes
getMaxCatalogNameLength()	INT	Yes
getMaxCharLiteralLength()	INT	Yes

方法名	返回值类型	是否支持
getMaxColumnNameLength()	INT	Yes
getMaxColumnsInGroupBy()	INT	Yes
getMaxColumnsInIndex()	INT	Yes
getMaxColumnsInOrderBy()	INT	Yes
getMaxColumnsInSelect()	INT	Yes
getMaxColumnsInTable()	INT	Yes
getMaxConnections()	INT	Yes
getMaxCursorNameLength()	INT	Yes
getMaxIndexLength()	INT	Yes
getMaxLogicalLobSize()	DEFAULT LONG	Yes
getMaxProcedureNameLength()	INT	Yes
getMaxRowSize()	INT	Yes
getMaxSchemaNameLength()	INT	Yes
getMaxStatementLength()	INT	Yes
getMaxStatements()	INT	Yes
getMaxTableNameLength()	INT	Yes
getMaxTablesInSelect()	INT	Yes
getMaxUserNameLength()	INT	Yes
getNumericFunctions()	STRING	Yes
getPrimaryKeys(String catalog, String schema, String table)	RESULTSET	Yes
getProcedureColumns(String catalog, String schemaPattern, String procedureNamePattern, String columnNamePattern)	RESULTSET	No
getProcedures(String catalog, String schemaPattern, String procedureNamePattern)	RESULTSET	No
getProcedureTerm()	STRING	Yes

方法名	返回值类型	是否支持
getResultSetHoldability()	INT	Yes
getSchemas()	RESULTSET	Yes
getSchemas(String catalog, String schemaPattern)	RESULTSET	Yes
getSchemaTerm()	STRING	Yes
getSearchStringEscape()	STRING	Yes
getSQLKeywords()	STRING	Yes
getSQLStateType()	INT	Yes
getStringFunctions()	STRING	Yes
getSystemFunctions()	STRING	Yes
getTablePrivileges(String catalog, String schemaPattern, String tableNamePattern)	RESULTSET	No
getTables(String catalog, String schemaPattern, String tableNamePattern, String[] types)	RESULTSET	No
getTableTypes()	RESULTSET	Yes
getTimeDateFunctions()	STRING	Yes
getTypeInfo()	RESULTSET	Yes
getUDTs(String catalog, String schemaPattern, String typeNamePattern, int[] types)	RESULTSET	Yes
getURL()	STRING	Yes
getUserName()	STRING	No
getVersionColumns(String catalog, String schema, String table)	RESULTSET	Yes
insertsAreDetected(int type)	BOOLEAN	Yes
isCatalogAtStart()	BOOLEAN	Yes
isReadOnly()	BOOLEAN	Yes
locatorsUpdateCopy()	BOOLEAN	Yes
nullPlusNonNullIsNull()	BOOLEAN	Yes

方法名	返回值类型	是否支持
nullsAreSortedAtEnd()	BOOLEAN	Yes
nullsAreSortedAtStart()	BOOLEAN	Yes
nullsAreSortedHigh()	BOOLEAN	Yes
nullsAreSortedLow()	BOOLEAN	Yes
othersDeletesAreVisible(int type)	BOOLEAN	Yes
othersInsertsAreVisible(int type)	BOOLEAN	Yes
othersUpdatesAreVisible(int type)	BOOLEAN	Yes
ownDeletesAreVisible(int type)	BOOLEAN	Yes
ownInsertsAreVisible(int type)	BOOLEAN	Yes
ownUpdatesAreVisible(int type)	BOOLEAN	Yes
storesLowerCaseIdentifiers()	BOOLEAN	Yes
storesLowerCaseQuotedIdentifiers()	BOOLEAN	Yes
storesMixedCaseIdentifiers()	BOOLEAN	Yes
storesMixedCaseQuotedIdentifiers()	BOOLEAN	Yes
storesUpperCaseIdentifiers()	BOOLEAN	Yes
storesUpperCaseQuotedIdentifiers()	BOOLEAN	Yes
supportsAlterTableWithAddColumn()	BOOLEAN	Yes
supportsAlterTableWithDropColumn()	BOOLEAN	Yes
supportsANSI92EntryLevelSQL()	BOOLEAN	Yes
supportsANSI92FullSQL()	BOOLEAN	Yes
supportsANSI92IntermediateSQL()	BOOLEAN	Yes
supportsBatchUpdates()	BOOLEAN	Yes

方法名	返回值类型	是否支持
supportsCatalogsInDataManipulation()	BOOLEAN	Yes
supportsCatalogsInIndexDefinitions()	BOOLEAN	Yes
supportsCatalogsInPrivilegeDefinitions()	BOOLEAN	Yes
supportsCatalogsInProcedureCalls()	BOOLEAN	Yes
supportsCatalogsInTableDefinitions()	BOOLEAN	Yes
supportsColumnAliasing()	BOOLEAN	Yes
supportsConvert()	BOOLEAN	Yes
supportsConvert(int fromType, int toType)	BOOLEAN	Yes
supportsCoreSQLGrammar()	BOOLEAN	Yes
supportsCorrelatedSubqueries()	BOOLEAN	Yes
supportsDataDefinitionAndDataManipulationTransactions()	BOOLEAN	Yes
supportsDataManipulationTransactionsOnly()	BOOLEAN	Yes
supportsDifferentTableCorrelationNames()	BOOLEAN	Yes
supportsExpressionsInOrderBy()	BOOLEAN	Yes
supportsExtendedSQLGrammar()	BOOLEAN	Yes
supportsFullOuterJoins()	BOOLEAN	Yes
supportsGetGeneratedKeys()	BOOLEAN	Yes
supportsGroupBy()	BOOLEAN	Yes
supportsGroupByBeyondSelect()	BOOLEAN	Yes
supportsGroupByUnrelated()	BOOLEAN	Yes
supportsIntegrityEnhancementFacility()	BOOLEAN	Yes

方法名	返回值类型	是否支持
supportsLikeEscapeClause()	BOOLEAN	Yes
supportsLimitedOuterJoins()	BOOLEAN	Yes
supportsMinimumSQLGrammar()	BOOLEAN	Yes
supportsMixedCaseIdentifiers()	BOOLEAN	Yes
supportsMixedCaseIdentifiers()	BOOLEAN	Yes
supportsMixedCaseQuotedIdentifiers()	BOOLEAN	Yes
supportsMultipleOpenResults()	BOOLEAN	Yes
supportsMultipleResultSets()	BOOLEAN	Yes
supportsMultipleTransactions()	BOOLEAN	Yes
supportsNamedParameters()	BOOLEAN	Yes
supportsNonNullableColumns()	BOOLEAN	Yes
supportsOpenCursorsAcrossCommit()	BOOLEAN	Yes
supportsOpenCursorsAcrossRollback()	BOOLEAN	Yes
supportsOpenStatementsAcrossCommit()	BOOLEAN	Yes
supportsOpenStatementsAcrossRollback()	BOOLEAN	Yes
supportsOrderByUnrelated()	BOOLEAN	Yes
supportsOuterJoins()	BOOLEAN	Yes
supportsPositionedDelete()	BOOLEAN	Yes
supportsPositionedUpdate()	BOOLEAN	Yes
supportsResultSetConcurrency(int type, int concurrency)	BOOLEAN	Yes
supportsResultSetHoldability(int holdability)	BOOLEAN	Yes
supportsResultSetType(int type)	BOOLEAN	Yes

方法名	返回值类型	是否支持
supportsSavepoints()	BOOLEAN	Yes
supportsSchemasInDataManipulation()	BOOLEAN	Yes
supportsSchemasInIndexDefinitions()	BOOLEAN	Yes
supportsSchemasInPrivilegeDefinitions()	BOOLEAN	Yes
supportsSchemasInProcedureCalls()	BOOLEAN	Yes
supportsSchemasInTableDefinitions()	BOOLEAN	Yes
supportsSelectForUpdate()	BOOLEAN	Yes
supportsStatementPooling()	BOOLEAN	Yes
supportsStoredFunctionsUsingCallSyntax()	BOOLEAN	Yes
supportsStoredProcedures()	BOOLEAN	Yes
supportsSubqueriesInComparisons()	BOOLEAN	Yes
supportsSubqueriesInExists()	BOOLEAN	Yes
supportsSubqueriesInIns()	BOOLEAN	Yes
supportsSubqueriesInQuantifieds()	BOOLEAN	Yes
supportsTableCorrelationNames()	BOOLEAN	Yes
supportsTransactionIsolationLevel(int level)	BOOLEAN	Yes
supportsTransactions()	BOOLEAN	Yes
supportsUnion()	BOOLEAN	Yes
supportsUnionAll()	BOOLEAN	Yes
updatesAreDetected(int type)	BOOLEAN	Yes
usesLocalFilePerTable()	BOOLEAN	Yes
usesLocalFiles()	BOOLEAN	Yes

3.5.4.3 java.sql.PreparedStatement

java.sql.PreparedStatement是预处理语句接口，具体信息如表3-18所示。

表 3-18 对 java.sql.PreparedStatement 的支持情况

方法名	返回值类型	支持JDBC4
addBatch()	VOID	Yes
executeBatch()	INT[]	Yes
clearParameters()	VOID	Yes
execute()	BOOLEAN	Yes
executeQuery()	RESULTSET	Yes
excuteUpdate()	INT	Yes
executeLargeUpdate()	LONG	Yes
getMetaData()	RESULTSETMETADATA	No
getParameterMetaData()	PARAMETERMETADATA	Yes
setArray(int parameterIndex, Array x)	VOID	Yes
setAsciiStream(int parameterIndex, InputStream x, int length)	VOID	Yes
setBinaryStream(int parameterIndex, InputStream x)	VOID	Yes
setBinaryStream(int parameterIndex, InputStream x, int length)	VOID	Yes
setBinaryStream(int parameterIndex, InputStream x, long length)	VOID	Yes
setBlob(int parameterIndex, InputStream inputStream)	VOID	Yes
setBlob(int parameterIndex, InputStream inputStream, long length)	VOID	Yes

方法名	返回值类型	支持JDBC4
setBlob(int parameterIndex, Blob x)	VOID	Yes
setCharacterStream(int parameterIndex, Reader reader)	VOID	Yes
setCharacterStream(int parameterIndex, Reader reader, int length)	VOID	Yes
setClob(int parameterIndex, Reader reader)	VOID	Yes
setClob(int parameterIndex, Reader reader, long length)	VOID	Yes
setClob(int parameterIndex, Clob x)	VOID	Yes
setDate(int parameterIndex, Date x, Calendar cal)	VOID	Yes
setNull(int parameterIndex, int sqlType)	VOID	Yes
setNull(int parameterIndex, int sqlType, String typeName)	VOID	Yes
setObject(int parameterIndex, Object x)	VOID	Yes
setObject(int parameterIndex, Object x, int targetSqlType)	VOID	Yes
setObject(int parameterIndex, Object x, int targetSqlType, int scaleOrLength)	VOID	Yes
setSQLXML(int parameterIndex, SQLXML xmlObject)	VOID	Yes
setTime(int parameterIndex, Time x)	VOID	Yes

方法名	返回值类型	支持JDBC4
setTime(int parameterIndex, Time x, Calendar cal)	VOID	Yes
setTimestamp(int parameterIndex, Timestamp x)	VOID	Yes
setTimestamp(int parameterIndex, Timestamp x, Calendar cal)	VOID	Yes
setUnicodeStream(int parameterIndex, InputStream x, int length)	VOID	Yes
setURL(int parameterIndex, URL x)	VOID	Yes
setBoolean(int parameterIndex, boolean x)	VOID	Yes
setBigDecimal(int parameterIndex, BigDecimal x)	VOID	Yes
setByte(int parameterIndex, byte x)	VOID	Yes
setBytes(int parameterIndex, byte[] x)	VOID	Yes
setDate(int parameterIndex, Date x)	VOID	Yes
setDouble(int parameterIndex, double x)	VOID	Yes
setFloat(int parameterIndex, float x)	VOID	Yes
setInt(int parameterIndex, int x)	VOID	Yes
setLong(int parameterIndex, long x)	VOID	Yes
setShort(int parameterIndex, short x)	VOID	Yes

方法名	返回值类型	支持JDBC4
setString(int parameterIndex, String x)	VOID	Yes
setNString(int parameterIndex, String x)	VOID	Yes
addBatch()	VOID	Yes
executeBatch()	INT[]	Yes

3.5.4.4 java.sql.ResultSet

java.sql.ResultSet是执行结果集接口，具体信息如表3-19所示。

表 3-19 对 java.sql.ResultSet 的支持情况

方法名	返回值类型	支持JDBC4
absolute(int row)	BOOLEAN	Yes
afterLast()	VOID	Yes
beforeFirst()	VOID	Yes
cancelRowUpdates()	VOID	Yes
clearWarnings()	VOID	Yes
close()	VOID	Yes
deleteRow()	VOID	Yes
findColumn(String columnLabel)	INT	Yes
first()	BOOLEAN	Yes
getArray(int columnIndex)	ARRAY	Yes
getArray(String columnLabel)	ARRAY	Yes
getAsciiStream(int columnIndex)	INPUTSTREAM	Yes
getAsciiStream(String columnLabel)	INPUTSTREAM	Yes
getBigDecimal(int columnIndex)	BIGDECIMAL	Yes

方法名	返回值类型	支持JDBC4
getBigDecimal(String columnLabel)	BIGDECIMAL	Yes
getBinaryStream(int columnIndex)	INPUTSTREAM	Yes
getBinaryStream(String columnLabel)	INPUTSTREAM	Yes
getBlob(int columnIndex)	BLOB	Yes
getBlob(String columnLabel)	BLOB	Yes
getBoolean(int columnIndex)	BOOLEAN	Yes
getBoolean(String columnLabel)	BOOLEAN	Yes
getByte(int columnIndex)	BYTE	Yes
getByte(String columnLabel)	BYTE	Yes
getBytes(int columnIndex)	BYTE[]	Yes
getBytes(String columnLabel)	BYTE[]	Yes
getCharacterStream(int columnIndex)	READER	Yes
getCharacterStream(Strin g columnLabel)	READER	Yes
getClob(int columnIndex)	CLOB	Yes
getClob(String columnLabel)	CLOB	Yes
getConcurrency()	INT	Yes
getCursorName()	STRING	Yes
getDate(int columnIndex)	DATE	Yes
getDate(int columnIndex, Calendar cal)	DATE	Yes
getDate(String columnLabel)	DATE	Yes
getDate(String columnLabel, Calendar cal)	DATE	Yes

方法名	返回值类型	支持JDBC4
getDouble(int columnIndex)	DOUBLE	Yes
getDouble(String columnLabel)	DOUBLE	Yes
getFetchDirection()	INT	Yes
getFetchSize()	INT	Yes
getFloat(int columnIndex)	FLOAT	Yes
getFloat(String columnLabel)	FLOAT	Yes
getInt(int columnIndex)	INT	Yes
getInt(String columnLabel)	INT	Yes
getLong(int columnIndex)	LONG	Yes
getLong(String columnLabel)	LONG	Yes
getMetaData()	RESULTSETMETADATA	Yes
getNString(int columnIndex)	STRING	Yes
getNString(String columnLabel)	STRING	Yes
getObject(int columnIndex)	OBJECT	Yes
getObject(int columnIndex, Class<T> type)	<T> T	Yes
getObject(int columnIndex, Map<String,Class<?>> map)	OBJECT	Yes
getObject(String columnLabel)	OBJECT	Yes
getObject(String columnLabel, Class<T> type)	<T> T	Yes
getObject(String columnLabel, Map<String,Class<?>> map)	OBJECT	Yes

方法名	返回值类型	支持JDBC4
getRow()	INT	Yes
getShort(int columnIndex)	SHORT	Yes
getShort(String columnLabel)	SHORT	Yes
getSQLXML(int columnIndex)	SQLXML	Yes
getSQLXML(String columnLabel)	SQLXML	Yes
getStatement()	STATEMENT	Yes
getString(int columnIndex)	STRING	Yes
getString(String columnLabel)	STRING	Yes
getTime(int columnIndex)	TIME	Yes
getTime(int columnIndex, Calendar cal)	TIME	Yes
getTime(String columnLabel)	TIME	Yes
getTime(String columnLabel, Calendar cal)	TIME	Yes
getTimestamp(int columnIndex)	TIMESTAMP	Yes
getTimestamp(int columnIndex, Calendar cal)	TIMESTAMP	Yes
getTimestamp(String columnLabel)	TIMESTAMP	Yes
getTimestamp(String columnLabel, Calendar cal)	TIMESTAMP	Yes
getType()	INT	Yes
getWarnings()	SQLWARNING	Yes
insertRow()	VOID	No
isAfterLast()	BOOLEAN	Yes
isBeforeFirst()	BOOLEAN	Yes

方法名	返回值类型	支持JDBC4
isClosed()	BOOLEAN	Yes
isFirst()	BOOLEAN	Yes
isLast()	BOOLEAN	Yes
last()	BOOLEAN	Yes
moveToCurrentRow()	VOID	Yes
moveToInsertRow()	VOID	Yes
next()	BOOLEAN	Yes
previous()	BOOLEAN	Yes
refreshRow()	VOID	Yes
relative(int rows)	BOOLEAN	Yes
rowDeleted()	BOOLEAN	Yes
rowInserted()	BOOLEAN	Yes
rowUpdated()	BOOLEAN	Yes
setFetchDirection(int direction)	VOID	Yes
setFetchSize(int rows)	VOID	Yes
updateArray(int columnIndex, Array x)	VOID	Yes
updateArray(String columnLabel, Array x)	VOID	Yes
updateAsciiStream(int columnIndex, InputStream x, int length)	VOID	Yes
updateAsciiStream(String columnLabel, InputStream x, int length)	VOID	Yes
updateBigDecimal(int columnIndex, BigDecimal x)	VOID	Yes
updateBigDecimal(String columnLabel, BigDecimal x)	VOID	Yes
updateBinaryStream(int columnIndex, InputStream x, int length)	VOID	Yes

方法名	返回值类型	支持JDBC4
updateBinaryStream(String columnLabel, InputStream x, int length)	VOID	Yes
updateBoolean(int columnIndex, boolean x)	VOID	Yes
updateBoolean(String columnLabel, boolean x)	VOID	Yes
updateByte(int columnIndex, byte x)	VOID	Yes
updateByte(String columnLabel, byte x)	VOID	Yes
updateBytes(int columnIndex, byte[] x)	VOID	Yes
updateBytes(String columnLabel, byte[] x)	VOID	Yes
updateCharacterStream(int columnIndex, Reader x, int length)	VOID	Yes
updateCharacterStream(String columnLabel, Reader reader, int length)	VOID	Yes
updateDate(int columnIndex, Date x)	VOID	Yes
updateDate(String columnLabel, Date x)	VOID	Yes
updateDouble(int columnIndex, double x)	VOID	Yes
updateDouble(String columnLabel, double x)	VOID	Yes
updateFloat(int columnIndex, float x)	VOID	Yes
updateFloat(String columnLabel, float x)	VOID	Yes
updateInt(int columnIndex, int x)	VOID	Yes
updateInt(String columnLabel, int x)	VOID	Yes
updateLong(int columnIndex, long x)	VOID	Yes

方法名	返回值类型	支持JDBC4
updateLong(String columnLabel, long x)	VOID	Yes
updateNull(int columnIndex)	VOID	Yes
updateNull(String columnLabel)	VOID	Yes
updateObject(int columnIndex, Object x)	VOID	Yes
updateObject(int columnIndex, Object x, int scaleOrLength)	VOID	Yes
updateObject(String columnLabel, Object x)	VOID	Yes
updateObject(String columnLabel, Object x, int scaleOrLength)	VOID	Yes
updateRow()	VOID	No
updateShort(int columnIndex, short x)	VOID	Yes
updateShort(String columnLabel, short x)	VOID	Yes
updateSQLXML(int columnIndex, SQLXML xmlObject)	VOID	Yes
updateSQLXML(String columnLabel, SQLXML xmlObject)	VOID	Yes
updateString(int columnIndex, String x)	VOID	Yes
updateString(String columnLabel, String x)	VOID	Yes
updateTime(int columnIndex, Time x)	VOID	Yes
updateTime(String columnLabel, Time x)	VOID	Yes
updateTimestamp(int columnIndex, Timestamp x)	VOID	Yes

方法名	返回值类型	支持JDBC4
updateTimestamp(String columnLabel, Timestamp x)	VOID	Yes
wasNull()	BOOLEAN	Yes

3.5.4.5 java.sql.ResultSetMetaData

java.sql.ResultSetMetaData是对ResultSet对象相关信息的具体描述，具体信息如表3-20所示。

表 3-20 对 java.sql.ResultSetMetaData 的支持情况

方法名	返回值类型	支持JDBC4
getCatalogName(int column)	STRING	Yes
getColumnClassName(int column)	STRING	Yes
getColumnCount()	INT	Yes
getColumnDisplaySize(int column)	INT	No
getColumnLabel(int column)	STRING	Yes
getColumnName(int column)	STRING	Yes
getColumnType(int column)	INT	Yes
getColumnTypeName(int column)	STRING	Yes
getPrecision(int column)	INT	No
getScale(int column)	INT	No
getSchemaName(int column)	STRING	Yes
getTableName(int column)	STRING	Yes
isAutoIncrement(int column)	BOOLEAN	Yes
isCaseSensitive(int column)	BOOLEAN	Yes
isCurrency(int column)	BOOLEAN	Yes
isDefinitelyWritable(int column)	BOOLEAN	Yes
isNullable(int column)	INT	Yes

方法名	返回值类型	支持JDBC4
isReadOnly(int column)	BOOLEAN	Yes
isSearchable(int column)	BOOLEAN	Yes
isSigned(int column)	BOOLEAN	Yes
isWritable(int column)	BOOLEAN	Yes

3.5.4.6 java.sql.Statement

java.sql.Statement是SQL语句接口，具体信息如表3-21所示。

表 3-21 对 java.sql.Statement 的支持情况

方法名	返回值类型	支持JDBC4
addBatch(String sql)	VOID	Yes
clearBatch()	VOID	Yes
clearWarnings()	VOID	Yes
close()	VOID	Yes
closeOnCompletion()	VOID	Yes
execute(String sql)	BOOLEAN	Yes
execute(String sql, int autoGeneratedKeys)	BOOLEAN	Yes
execute(String sql, int[] columnIndexes)	BOOLEAN	Yes
execute(String sql, String[] columnNames)	BOOLEAN	Yes
executeBatch()	BOOLEAN	Yes
executeQuery(String sql)	RESULTSET	Yes
executeUpdate(String sql)	INT	Yes
executeUpdate(String sql, int autoGeneratedKeys)	INT	Yes
executeUpdate(String sql, int[] columnIndexes)	INT	Yes

方法名	返回值类型	支持JDBC4
executeUpdate(String sql, String[] columnNames)	INT	Yes
getConnection()	CONNECTION	Yes
getFetchDirection()	INT	Yes
getFetchSize()	INT	Yes
getGeneratedKeys()	RESULTSET	Yes
getMaxFieldSize()	INT	Yes
getMaxRows()	INT	Yes
getMoreResults()	BOOLEAN	Yes
getMoreResults(int current)	BOOLEAN	Yes
getResultSet()	RESULTSET	Yes
getResultSetConcurrency()	INT	Yes
getResultSetHoldability()	INT	Yes
getResultSetType()	INT	Yes
getQueryTimeout()	INT	Yes
getUpdateCount()	INT	Yes
getWarnings()	SQLWARNING	No
isClosed()	BOOLEAN	Yes
isCloseOnCompletion()	BOOLEAN	Yes
isPoolable()	BOOLEAN	Yes
setCursorName(String name)	VOID	Yes
setEscapeProcessing(boolean enable)	VOID	Yes
setFetchDirection(int direction)	VOID	Yes
setMaxFieldSize(int max)	VOID	No
setMaxRows(int max)	VOID	No

方法名	返回值类型	支持JDBC4
setPoolable(boolean poolable)	VOID	Yes
setQueryTimeout(int seconds)	VOID	Yes
setFetchSize(int rows)	VOID	Yes
cancel()	VOID	No
executeLargeUpdate(String sql)	LONG	Yes
getLargeUpdateCount()	LONG	Yes
executeLargeBatch()	LONG	Yes
executeLargeUpdate(String sql, int autoGeneratedKeys)	LONG	Yes
executeLargeUpdate(String sql, int[] columnIndexes)	LONG	Yes
executeLargeUpdate(String sql, String[] columnNames)	LONG	Yes

3.5.5 附录

3.5.5.1 JDBC 数据类型映射关系

在M-Compatibility模式数据库下，数据类型、JAVA变量类型以及JDBC类型索引关系如表3-22所示。

表 3-22 JDBC 数据类型映射关系

Gauss Kernel数据类型	JAVA变量类型	JDBC类型索引
TINYINT	java.lang.Integer	Types.TINYINT
TINYINT UNSIGNED	java.lang.Integer	Types.TINYINT
SMALLINT	java.lang.Integer	Types.SMALLINT
SMALLINT UNSIGNED	java.lang.Integer	Types.SMALLINT

Gauss Kernel数据类型	JAVA变量类型	JDBC类型索引
INT	java.lang.Integer	Types.INTEGER
INT UNSIGNED	java.lang.Long	Types.INTEGER
MEDIUMINT	java.lang.Integer	Types.INTEGER
MEDIUMINT UNSIGNED	java.lang.Integer	Types.INTEGER
BIGINT	java.lang.Long	Types.BIGINT
BIGINT UNSIGNED	java.math.BigInteger	Types.BIGINT
NUMERIC	java.math.BigDecimal	Types.DECIMAL
FLOAT4	java.lang.Float	Types.REAL
FLOAT8	java.lang.Double	Types.DOUBLE
BIT	java.lang.Boolean	Types.BIT
BOOL/BOOLEAN	java.lang.Boolean	Types.BIT
CHAR	java.lang.String	Types.CHAR
VARCHAR	java.lang.String	Types.VARCHAR
TEXT	java.lang.String	Types.LONGVARCHAR
TINYTEXT	java.lang.String	Types.LONGVARCHAR
MEDIUMTEXT	java.lang.String	Types.LONGVARCHAR
LONGTEXT	java.lang.String	Types.LONGVARCHAR
BINARY	byte[]	Types.BINARY
VARBINARY	byte[]	Types.VARBINARY
TINYBLOB	byte[]	Types.VARBINARY
BLOB	byte[]	Types.LONGVARBINAR
MEDIUMBLOB	byte[]	Types.LONGVARBINAR
LONGBLOB	byte[]	Types.LONGVARBINAR
DATE	java.sql.Date	Types.DATE
TIME	java.sql.Time	Types.TIME
TIMESTAMP	java.sql.Timestamp	Types.TIMESTAMP
DATETIME	java.sql.Timestamp	Types.TIMESTAMP
YEAR	java.sql.Date	Types.DATE

Gauss Kernel数据类型	JAVA变量类型	JDBC类型索引
SET	java.lang.String	Types.CHAR
ENUM	java.lang.String	Types.CHAR

3.6 基于 ODBC 开发

ODBC（Open Database Connectivity，开放数据库互连）是由Microsoft公司基于X/OPEN CLI提出的用于访问数据库的应用程序编程接口。提供了一种统一的方法，让应用程序可以访问各种数据库管理系统（DBMS），而不用考虑具体的数据库类型或者操作系统平台。ODBC允许应用程序使用SQL来查询、插入、更新和删除数据库中的数据。应用程序通过ODBC提供的API接口与数据库进行交互，增强了应用程序的可移植性、扩展性和可维护性。

M-Comaptibility模式数据库支持基于GaussDB ODBC的开发，具体使用方法请参见《开发指南》中的“应用程序开发 > 基于ODBC开发”章节，部分功能和GaussDB存在差异点。

GaussDB ODBC连接M-Compatibility模式数据库使用约束：

- 创建数据库连接时，UseDeclareFetch参数不支持。
- 当GUC参数m_err_sql_dialect的值为"MySQL"时，可以启用错误码替换功能，将GaussDB内核的错误码和SQLSTATE替换为M-Compatibility模式数据库的错误码和SQLSTATE。
- 其他不支持的GaussDB特性，具体参见**M-Compatibility数据库不支持的GaussDB特性**。

4 SQL 参考

4.1 SQL 简介

什么是 SQL

SQL是用于访问和处理数据库的标准计算机语言。

SQL提供了各种任务的语句，包括：

- 查询数据。
- 在表中插入、更新和删除行。
- 创建、替换、更改和删除对象。
- 控制对数据库及其对象的访问。
- 保证数据库的一致性和完整性。

SQL语言由用于处理数据库和数据库对象的命令和函数组成。该语言还会强制实施有关数据类型、表达式和文本使用的规则。因此在**SQL参考**章节，除了SQL语法参考外，还会看到有关数据类型、表达式、函数和操作符等信息。

SQL 发展简史

SQL发展简史如下：

- 1986年，ANSI X3.135-1986，ISO/IEC 9075:1986，SQL-86
- 1989年，ANSI X3.135-1989，ISO/IEC 9075:1989，SQL-89
- 1992年，ANSI X3.135-1992，ISO/IEC 9075:1992，SQL-92
- 1999年，ISO/IEC 9075:1999，SQL:1999
- 2003年，ISO/IEC 9075:2003，SQL:2003
- 2011年，ISO/IEC 9075:2011，SQL:2011
- 2016年，ISO/IEC 9075:2016，SQL:2016
- 2019年，ISO/IEC 9075:2019，SQL:2019

GaussDB 支持的 SQL 标准

GaussDB默认支持SQL:2016的大部分特性。

4.2 关键字

SQL里有保留关键字和非保留关键字之分。根据标准，保留关键字绝不能用作其他标识符。非保留关键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作标识符的。在保留关键字和非保留关键字的基础上，M-Compatibility进一步细化了关键字作为标识符的使用范围，由此扩展出以下四类关键字：

- 保留：一般的保留关键字，不可以作为任何数据库对象（表、列、函数、类型、视图、索引以及变量等）的标识符。
- 非保留：一般的非保留关键字，可作为任何数据库对象的标识符。
- 保留（可以是函数或类型）：特殊的保留关键字，与一般的保留关键字相比，该类关键字可以用作函数和变量的标识符。
- 非保留（不能是函数或类型）：特殊的非保留关键字，与一般的非保留关键字相比，该类关键字不能被用作函数和变量的标识符。

约束差异：

- 关键字实现的约束差异参见《兼容性说明》中的“MySQL兼容性说明 > MySQL兼容性M-Compatibility模式 > SQL > 关键字”章节。

须知

1. 目前“非保留”关键字在作为数据库对象的标识符时存在如下限制：

   不带反引号的BEGIN、BY、CLOSE、CURSOR、DECLARE、DELETE、EXECUTE、FUNCTION、IF、IMMEDIATE、INSERT、LOOP、MOVE、OF、REF、RELEASE、RETURN、SAVEPOINT、STRICT、TYPE以及UPDATE等关键字不支持作为变量名使用。
2. 保留关键字以及以下关键字不能直接作为列别名使用，即类似SELECT 1 BIGINT的用法会导致错误：

   BETWEEN、BIGINT、BLOB、CHAR、CHARACTER、CROSS、DEC、DECIMAL、DIV、DOUBLE、EXISTS、FLOAT、FLOAT4、FLOAT8、GROUPING、INNER、INOUT、INT、INT1、INT2、INT3、INT4、INT8、INTEGER、JOIN、LEFT、LIKE、LONGBLOB、LONGTEXT、MEDIUMBLOB、MEDIUMINT、MEDIUMTEXT、MOD、NATURAL、NUMERIC、OUT、OUTER、PRECISION、REAL、RIGHT、ROW、ROW_NUMBER、SIGNED、SMALLINT、SOUNDS、TINYBLOB、TINYINT、TINYTEXT、VALUES、VARCHAR、VARYING、WITHOUT

SQL 关键字

表 4-1 SQL 关键字

关键字	类型
ABSOLUTE	非保留

关键字	类型
ACCESSIBLE	保留
ACCOUNT	非保留
ACTION	非保留
ACTIVE	非保留
ADD	非保留
ADDDATE	非保留
ADMIN	非保留
AFTER	非保留
AGAINST	非保留
AGGREGATE	非保留
ALGORITHM	非保留
ALL	保留
ALTER	非保留
ALWAYS	非保留
ANALYZE	保留
AND	保留
ANY	保留
APPEND	非保留
ARRAY	保留
AS	保留
ASC	保留
ASCII	非保留
ASENSITIVE	保留
ASSIGN_GTIDS_TO_ANONYMOUS_TRANSACTIONS	非保留
AT	非保留
ATTRIBUTE	非保留
AUTHID	非保留
AUTHENTICATION	非保留
AUTHORIZATION	非保留

关键字	类型
AUTO_INCREMENT	非保留
AUTOEXTEND_SIZE	非保留
AVG	非保留
AVG_ROW_LENGTH	非保留
BACKUP	非保留
BEFORE	非保留
BEGIN	非保留
BEGIN_NON_ANOYBLOCK	非保留
BETWEEN	非保留（不能是函数或类型）
BIGINT	非保留（不能是函数或类型）
BINARY	保留
BINLOG	非保留
BIT	非保留（不能是函数或类型）
BLOB	非保留（不能是函数或类型）
BLOCK	非保留
BOOL	非保留（不能是函数或类型）
BOOLEAN	非保留（不能是函数或类型）
BOTH	保留
BTREE	非保留
BUCKETS	非保留
BULK	非保留
BY	非保留
BYTE	非保留
CACHE	非保留
CALL	非保留
CALLED	非保留
CASCADE	非保留
CASCADED	非保留
CASE	保留
CAST	非保留

关键字	类型
CATALOG_NAME	非保留
CHAIN	非保留
CHALLENGE_RESPONSE	非保留
CHANGE	非保留
CHANGED	非保留
CHANNEL	非保留
CHAR	非保留（不能是函数或类型）
CHARACTER	非保留（不能是函数或类型）
CHARSET	非保留
CHECK	保留
CHECKPOINT	非保留
CHECKSUM	非保留
CIPHER	非保留
CLASS_ORIGIN	非保留
CLEAN	非保留
CLIENT	非保留
CLONE	非保留
CLOSE	非保留
CLUSTER	非保留
COALESCE	非保留
CODE	非保留
COLLATE	保留
COLLATION	非保留
COLUMN	保留
COLUMN_FORMAT	非保留
COLUMN_NAME	非保留
COLUMNS	非保留
COMMENT	非保留
COMMIT	非保留
COMMITTED	非保留

关键字	类型
COMPACT	保留（可以是函数或类型）
COMPLETION	非保留
COMPONENT	非保留
COMPRESSED	非保留
COMPRESSION	非保留
CONCURRENT	非保留
CONCURRENTLY	保留（可以是函数或类型）
CONDITION	非保留
CONNECTION	非保留
CONSISTENT	非保留
CONSTRAINT	保留
CONSTRAINT_CATALOG	非保留
CONSTRAINT_NAME	非保留
CONSTRAINT_SCHEMA	非保留
CONSTRAINTS	非保留
CONTAINS	非保留
CONTEXT	非保留
CONTINUE	非保留
CONVERT	非保留
COORDINATOR	非保留
COORDINATORS	非保留
COPY	非保留
COST	非保留
COUNT	非保留
CPU	非保留
CREATE	保留
CROSS	保留（可以是函数或类型）
CSV	非保留
CUBE	非保留
CUME_DIST	保留

关键字	类型
CURRENT	非保留
CURRENT_DATE	保留
CURRENT_SCHEMA	保留（可以是函数或类型）
CURRENT_TIME	保留
CURRENT_TIMESTAMP	保留
CURRENT_USER	保留
CURSOR	非保留
CURSOR_NAME	非保留
CURTIME	非保留
CYCLE	非保留
DATA	非保留
DATABASE	非保留
DATABASES	保留
DATAFILE	非保留
DATANODE	非保留
DATANODES	非保留
DATE	非保留（不能是函数或类型）
DATE_ADD	非保留
DATE_FORMAT	非保留
DATE_SUB	非保留
DATETIME	非保留
DAY	非保留
DAY_HOUR	保留
DAY_MICROSECOND	保留
DAY_MINUTE	保留
DAY_SECOND	保留
DEALLOCATE	非保留
DEC	非保留（不能是函数或类型）
DECIMAL	非保留（不能是函数或类型）
DECLARE	非保留

关键字	类型
DECODE	非保留
DEFAULT	保留
DEFAULT_AUTH	非保留
DEFINER	非保留
DEFINITION	非保留
DELAY_KEY_WRITE	非保留
DELAYED	保留
DELETE	非保留
DELIMITER	非保留
DENSE_RANK	保留
DESC	保留
DESCRIBE	保留
DESCRIPTION	非保留
DETERMINISTIC	非保留
DIAGNOSTICS	非保留
DICTIONARY	非保留
DIRECT	非保留
DIRECTORY	非保留
DISABLE	非保留
DISCARD	非保留
DISK	非保留
DISTINCT	保留
DISTINCTROW	保留
DISTRIBUTE	非保留
DISTRIBUTED	非保留
DIV	保留（可以是函数或类型）
DO	非保留
DOCUMENT	非保留
DOUBLE	非保留（不能是函数或类型）
DROP	非保留

关键字	类型
DUAL	保留
DUMPFILE	非保留
DUPLICATE	非保留
DYNAMIC	非保留
EACH	非保留
ELSE	保留
ELSEIF	保留
EMPTY	保留
ENABLE	非保留
ENCLOSED	非保留
ENCRYPTION	非保留
END	非保留
ENDS	非保留
ENFORCED	非保留
ENGINE	非保留
ENGINE_ATTRIBUTE	非保留
ENGINES	非保留
ENUM	非保留
ERROR	非保留
ERRORS	非保留
ESCAPE	非保留
ESCAPED	非保留
EVENT	非保留
EVENTS	非保留
EVERY	非保留
EXCEPT	保留
EXCHANGE	非保留
EXCLUDE	非保留
EXECUTE	非保留
EXISTS	非保留（不能是函数或类型）

关键字	类型
EXIT	保留
EXPANSION	非保留
EXPIRE	非保留
EXPIRED	非保留
EXPLAIN	非保留
EXPORT	非保留
EXTENDED	非保留
EXTENSION	非保留
EXTENT_SIZE	非保留
EXTERNAL	非保留
EXTRACT	非保留
FACTOR	非保留
FAILED_LOGIN_ATTEMPTS	非保留
FALSE	保留
FAST	非保留
FAULTS	非保留
FENCED	非保留
FETCH	保留
FIELDS	非保留
FILE	非保留
FILE_BLOCK_SIZE	非保留
FILTER	非保留
FINISH	非保留
FIRST	非保留
FIRST_VALUE	保留
FIXED	非保留（不能是函数或类型）
FLOAT	非保留（不能是函数或类型）
FLOAT4	非保留（不能是函数或类型）
FLOAT8	非保留（不能是函数或类型）
FLUSH	非保留

关键字	类型
FOLLOWING	非保留
FOLLOWS	非保留
FOR	保留
FORCE	保留
FOREIGN	保留
FORMAT	非保留
FORWARD	非保留
FOUND	非保留
FREEZE	保留（可以是函数或类型）
FROM	保留
FULL	非保留
FULLTEXT	保留
FUNCTION	非保留
GENERAL	非保留
GENERATE	非保留
GENERATED	非保留
GEOMCOLLECTION	非保留
GEOMETRY	非保留
GEOMETRYCOLLECTION	非保留
GET	保留
GET_FORMAT	非保留
GET_MASTER_PUBLIC_KEY	非保留
GET_SOURCE_PUBLIC_KEY	非保留
GLOBAL	非保留
GRANT	保留
GRANTED	非保留
GRANTS	非保留
GROUP	保留
GROUP_REPLICATION	非保留
GROUPING	非保留（不能是函数或类型）

关键字	类型
GROUPS	保留
GTID_ONLY	非保留
HANDLER	非保留
HASH	非保留
HAVING	保留
HELP	非保留
HIGH_PRIORITY	保留
HISTOGRAM	非保留
HISTORY	非保留
HOLD	非保留
HOST	非保留
HOSTS	非保留
HOUR	非保留
HOUR_MICROSECOND	保留
HOUR_MINUTE	保留
HOUR_SECOND	保留
IDENTIFIED	非保留
IDENTITY	非保留
IF	保留
IGNORE	保留
IGNORE_SERVER_IDS	非保留
IMMUTABLE	非保留
IMPORT	非保留
IN	保留
INACTIVE	非保留
INCLUDING	非保留
INCREMENT	非保留
INDEX	保留
INDEXES	非保留
INFILE	非保留

关键字	类型
INITIAL	非保留
INITIAL_SIZE	非保留
INITIATE	非保留
INNER	保留（可以是函数或类型）
INOUT	非保留（不能是函数或类型）
INPUT	非保留
INSENSITIVE	非保留
INSERT	非保留
INSERT_METHOD	非保留
INSTALL	非保留
INSTANCE	非保留
INT	非保留（不能是函数或类型）
INT1	非保留（不能是函数或类型）
INT2	非保留（不能是函数或类型）
INT3	非保留（不能是函数或类型）
INT4	非保留（不能是函数或类型）
INT8	非保留（不能是函数或类型）
INTEGER	非保留（不能是函数或类型）
INTERSECT	保留
INTERVAL	保留
INTO	保留
INVISIBLE	非保留
INVOKER	非保留
IO	非保留
IO_AFTER_GTIDS	保留
IO_BEFORE_GTIDS	保留
IO_THREAD	非保留
IPC	非保留
IS	保留
ISNULL	非保留

关键字	类型
ISOLATION	非保留
ISSUER	非保留
ITERATE	保留
JOIN	保留（可以是函数或类型）
JSON	非保留
JSON_TABLE	保留
JSON_VALUE	非保留
KEY	保留
KEY_BLOCK_SIZE	非保留
KEYRING	非保留
KEYS	保留
KILL	非保留
LAG	保留
LANGUAGE	非保留
LAST	非保留
LAST_VALUE	保留
LATERAL	保留
LC_COLLATE	非保留
LEAD	保留
LEADING	保留
LEAKPROOF	非保留
LEAVE	保留
LEAVES	非保留
LEFT	保留（可以是函数或类型）
LESS	非保留
LEVEL	非保留
LIKE	保留（可以是函数或类型）
LIMIT	保留
LINEAR	保留
LINES	非保留

关键字	类型
LINESTRING	非保留
LIST	非保留
LOAD	非保留
LOAD_BAD	非保留
LOAD_DISCARD	非保留
LOCAL	非保留
LOCALTIME	保留
LOCALTIMESTAMP	保留
LOCATION	非保留
LOCK	保留
LOCKED	非保留
LOCKS	非保留
LOGFILE	非保留
LOGS	非保留
LONG	保留
LONGBLOB	非保留（不能是函数或类型）
LONGTEXT	非保留（不能是函数或类型）
LOOP	非保留
LOW_PRIORITY	保留
MASTER	非保留
MASTER_AUTO_POSITION	非保留
MASTER_BIND	保留
MASTER_COMPRESSION_ALGORITHMS	非保留
MASTER_CONNECT_RETRY	非保留
MASTER_DELAY	非保留
MASTER_HEARTBEAT_PERIOD	非保留
MASTER_HOST	非保留
MASTER_LOG_FILE	非保留
MASTER_LOG_POS	非保留

关键字	类型
MASTER_PASSWORD	非保留
MASTER_PORT	非保留
MASTER_PUBLIC_KEY_PATH	非保留
MASTER_RETRY_COUNT	非保留
MASTER_SSL	非保留
MASTER_SSL_CA	非保留
MASTER_SSL_CAPATH	非保留
MASTER_SSL_CERT	非保留
MASTER_SSL_CIPHER	非保留
MASTER_SSL_CRL	非保留
MASTER_SSL_CRLPATH	非保留
MASTER_SSL_KEY	非保留
MASTER_SSL_VERIFY_SERVER_CERT	保留
MASTER_TLS_CIPHERSUITES	非保留
MASTER_TLS_VERSION	非保留
MASTER_USER	非保留
MASTER_ZSTD_COMPRESSION_LEVEL	非保留
MATCH	保留
MAX_CONNECTIONS_PER_HOUR	非保留
MAX_QUERIES_PER_HOUR	非保留
MAX_ROWS	非保留
MAX_SIZE	非保留
MAX_UPDATES_PER_HOUR	非保留
MAX_USER_CONNECTIONS	非保留
MAXVALUE	保留
MEDIUM	非保留
MEDIUMBLOB	非保留（不能是函数或类型）
MEDIUMINT	非保留（不能是函数或类型）
MEDIUMTEXT	非保留（不能是函数或类型）
MEMBER	非保留

关键字	类型
MEMORY	非保留
MERGE	非保留
MESSAGE_TEXT	非保留
MICROSECOND	非保留
MIDDLEINT	保留
MIGRATE	非保留
MIN_ROWS	非保留
MINUTE	非保留
MINUTE_MICROSECOND	保留
MINUTE_SECOND	保留
MINVALUE	非保留
MOD	保留（可以是函数或类型）
MODE	非保留
MODIFIES	保留
MODIFY	非保留
MONTH	非保留
MOVE	非保留
MULTILINESTRING	非保留
MULTIPOINT	非保留
MULTIPOLYGON	非保留
MUTEX	非保留
MYSQL_ERRNO	非保留
NAME	非保留
NAMES	非保留
NATIONAL	非保留（不能是函数或类型）
NATURAL	保留（可以是函数或类型）
NCHAR	非保留（不能是函数或类型）
NDB	非保留
NDBCLUSTER	非保留
NESTED	非保留

关键字	类型
NETWORK_NAMESPACE	非保留
NEVER	非保留
NEW	非保留
NEXT	非保留
NO	非保留
NO_WAIT	非保留
NO_WRITE_TO_BINLOG	保留
NOCYCLE	非保留
NODE	非保留
NODEGROUP	非保留
NOMAXVALUE	非保留
NOMINVALUE	非保留
NONE	非保留
NOT	保留
NOTHING	非保留
NOW	非保留
NOWAIT	非保留
NTH_VALUE	保留
NTILE	保留
NULL	保留
NULLS	非保留
NUMBER	非保留（不能是函数或类型）
NUMERIC	非保留（不能是函数或类型）
NVARCHAR	非保留（不能是函数或类型）
OF	非保留
OFF	非保留
OFFSET	保留
OIDS	非保留
OJ	非保留
OLD	非保留

关键字	类型
ON	保留
ONE	非保留
ONLY	保留
OPEN	非保留
OPERATOR	非保留
OPTIMIZE	保留
OPTIMIZE_COSTS	保留
OPTION	非保留
OPTIONAL	非保留
OPTIONALLY	非保留
OPTIONS	非保留
OR	保留
ORDER	保留
ORDINALITY	非保留
ORGANIZATION	非保留
OTHERS	非保留
OUT	非保留（不能是函数或类型）
OUTER	保留（可以是函数或类型）
OUTFILE	非保留
OVER	非保留
OWNED	非保留
OWNER	非保留
PACK_KEYS	非保留
PAGE	非保留
PARSER	非保留
PARTIAL	非保留
PARTITION	非保留
PARTITIONING	非保留
PARTITIONS	非保留
PASSWORD	非保留

关键字	类型
PASSWORD_LOCK_TIME	非保留
PATH	非保留
PERCENT_RANK	保留
PERSIST	非保留
PERSIST_ONLY	非保留
PHASE	非保留
PLUGIN	非保留
PLUGINS	非保留
PLUGIN_DIR	非保留
POINT	非保留
POLYGON	非保留
PORT	非保留
PRECEDES	非保留
PRECEDING	非保留
PRECISION	非保留（不能是函数或类型）
PREPARE	非保留
PREPARED	非保留
PRESERVE	非保留
PREV	非保留
PRIMARY	保留
PRIVILEGES	非保留
PRIVILEGE_CHECKS_USER	非保留
PROCEDURE	保留
PROCESS	非保留
PROCESSLIST	非保留
PROFILE	非保留
PROFILES	非保留
PROXY	非保留
PURGE	非保留
QUARTER	非保留

关键字	类型
QUERY	非保留
QUICK	非保留
RANDOM	非保留
RANGE	非保留
RANK	保留
READ	保留
READS	保留
READ_ONLY	非保留
READ_WRITE	保留
REAL	非保留（不能是函数或类型）
REBUILD	非保留
RECOVER	非保留
RECURSIVE	非保留
RECYCLEBIN	非保留
REDO_BUFFER_SIZE	非保留
REDUNDANT	非保留
REFERENCE	非保留
REFERENCES	保留
REGEXP	保留
REGISTRATION	非保留
REINDEX	非保留
RELATIVE	非保留
RELAY	非保留
RELAYLOG	非保留
RELAY_LOG_FILE	非保留
RELAY_LOG_POS	非保留
RELAY_THREAD	非保留
RELEASE	非保留
RELOAD	非保留
REMOVE	非保留

关键字	类型
RENAME	非保留
REORGANIZE	非保留
REPAIR	非保留
REPEAT	保留
REPEATABLE	非保留
REPLACE	非保留
REPLICA	非保留
REPLICAS	非保留
REPLICATE_DO_DB	非保留
REPLICATE_DO_TABLE	非保留
REPLICATE_IGNORE_DB	非保留
REPLICATE_IGNORE_TABLE	非保留
REPLICATE_REWRITE_DB	非保留
REPLICATE_WILD_DO_TABLE	非保留
REPLICATE_WILD_IGNORE_TABLE	非保留
REPLICATION	非保留
REQUIRE	保留
REQUIRE_ROW_FORMAT	非保留
REQUIRE_TABLE_PRIMARY_KEY_CHECK	非保留
RESET	非保留
RESIGNAL	保留
RESOURCE	非保留
RESPECT	非保留
RESTART	非保留
RESTORE	非保留
RESTRICT	非保留
RESUME	非保留
RETAIN	非保留
RETURN	非保留
RETURNED_SQLSTATE	非保留

关键字	类型
RETURNING	保留
RETURNS	非保留
REUSE	非保留
REVERSE	非保留
REVOKE	非保留
RIGHT	保留（可以是函数或类型）
RLIKE	保留
ROLE	非保留
ROLLBACK	非保留
ROLLUP	非保留
ROTATE	非保留
ROUTINE	非保留
ROW	非保留（不能是函数或类型）
ROWS	非保留
ROW_COUNT	非保留
ROW_FORMAT	非保留
ROW_NUMBER	保留（可以是函数或类型）
RTREE	非保留
SAVEPOINT	非保留
SCHEDULE	非保留
SCHEMA	非保留
SCHEMAS	保留
SCHEMA_NAME	非保留
SCROLL	非保留
SECOND	非保留
SECOND_MICROSECOND	保留
SECONDARY	非保留
SECONDARY_ENGINE	非保留
SECONDARY_ENGINE_ATTRIBUTE	非保留
SECONDARY_LOAD	非保留

关键字	类型
SECONDARY_UNLOAD	非保留
SECURITY	非保留
SELECT	保留
SENSITIVE	保留
SEPARATOR	非保留
SEQUENCE	非保留
SERIAL	非保留
SERIALIZABLE	非保留
SERVER	非保留
SESSION	非保留
SET	非保留
SETOF	非保留（不能是函数或类型）
SHARE	非保留
SHIPPABLE	非保留
SHOW	非保留
SHUTDOWN	非保留
SIGNAL	保留
SIGNED	非保留
SIMPLE	非保留
SIZE	非保留
SKIP	非保留
SLAVE	非保留
SLICE	非保留
SLICEGROUP	非保留
SLOW	非保留
SMALLDATETIME	非保留（不能是函数或类型）
SMALLDATETIME_FORMAT	非保留
SMALLINT	非保留（不能是函数或类型）
SNAPSHOT	非保留
SOCKET	非保留

关键字	类型
SOME	保留
SONAME	非保留
SOUNDS	非保留
SOURCE	非保留
SOURCE_AUTO_POSITION	非保留
SOURCE_BIND	非保留
SOURCE_COMPRESSION_ALGORITHMS	非保留
SOURCE_CONNECTION_AUTO_FAILOVER	非保留
SOURCE_CONNECT_RETRY	非保留
SOURCE_DELAY	非保留
SOURCE_HEARTBEAT_PERIOD	非保留
SOURCE_HOST	非保留
SOURCE_LOG_FILE	非保留
SOURCE_LOG_POS	非保留
SOURCE_PASSWORD	非保留
SOURCE_PORT	非保留
SOURCE_PUBLIC_KEY_PATH	非保留
SOURCE_RETRY_COUNT	非保留
SOURCE_SSL	非保留
SOURCE_SSL_CA	非保留
SOURCE_SSL_CAPATH	非保留
SOURCE_SSL_CERT	非保留
SOURCE_SSL_CIPHER	非保留
SOURCE_SSL_CRL	非保留
SOURCE_SSL_CRLPATH	非保留
SOURCE_SSL_KEY	非保留
SOURCE_SSL_VERIFY_SERVER_CERT	非保留
SOURCE_TLS_CIPHERSUITES	非保留
SOURCE_TLS_VERSION	非保留

关键字	类型
SOURCE_USER	非保留
SOURCE_ZSTD_COMPRESSION_LEVEL	非保留
SPATIAL	保留
SPECIFIC	保留
SPLIT	非保留
SQL	非保留
SQLEXCEPTION	保留
SQLSTATE	保留
SQLWARNNING	保留
SQL_AFTER_GTIDS	非保留
SQL_AFTER_MTS_GAPS	非保留
SQL_BEFORE_GTIDS	非保留
SQL_BIG_RESULT	保留
SQL_BUFFER_RESULT	非保留
SQL_CALC_FOUND_ROWS	保留
SQL_NO_CACHE	保留
SQL_SMALL_RESULT	非保留
SQL_THREAD	非保留
SQL_TSI_DAY	非保留
SQL_TSI_HOUR	非保留
SQL_TSI_MINUTE	非保留
SQL_TSI_MONTH	非保留
SQL_TSI_QUARTER	非保留
SQL_TSI_SECOND	非保留
SQL_TSI_WEEK	非保留
SQL_TSI_YEAR	非保留
SRID	非保留
SSL	保留
STABLE	非保留
STACKED	非保留

关键字	类型
START	非保留
STARTING	非保留
STARTS	非保留
STATS_AUTO_RECALC	非保留
STATS_PERSISTENT	非保留
STATS_SAMPLE_PAGES	非保留
STATUS	非保留
STDIN	非保留
STDOUT	非保留
STOP	非保留
STORAGE	非保留
STORED	非保留
STRAIGHT_JOIN	保留
STREAM	非保留
STRICT	非保留
STRING	非保留
SUBCLASS_ORIGIN	非保留
SUBDATE	非保留
SUBJECT	非保留
SUBPARTITION	非保留
SUBPARTITIONS	非保留
SUBSTR	非保留
SUBSTRING	非保留
SUPER	非保留
SUSPEND	非保留
SWAPS	非保留
SWITCHES	非保留
SYSDATE	非保留
SYSTEM	非保留
TABLE	保留

关键字	类型
TABLES	非保留
TABLESPACE	非保留
TABLE_CHECKSUM	非保留
TABLE_NAME	非保留
TEMPORARY	非保留
TEMPTABLE	非保留
TERMINATED	非保留
TEXT	非保留（不能是函数或类型）
THAN	非保留
THEN	保留
THREAD_PRIORITY	非保留
TIES	非保留
TIME	非保留（不能是函数或类型）
TIMESTAMP	非保留（不能是函数或类型）
TIMESTAMPADD	非保留
TIMESTAMPDIFF	非保留
TINYBLOB	非保留（不能是函数或类型）
TINYINT	非保留（不能是函数或类型）
TINYTEXT	非保留（不能是函数或类型）
TLS	非保留
TO	保留
TRAILING	保留
TRANSACTION	非保留
TRIGGER	保留
TRIGGERS	非保留
TRIM	非保留
TRUE	保留
TRUNCATE	非保留
TYPE	非保留
TYPES	非保留

关键字	类型
UNBOUNDED	非保留
UNCOMMITTED	非保留
UNDEFINED	非保留
UNDO	保留
UNDOFILE	非保留
UNDO_BUFFER_SIZE	非保留
UNICODE	非保留
UNINSTALL	非保留
UNION	保留
UNIQUE	保留
UNKNOWN	非保留
UNLOCK	非保留
UNLOGGED	非保留
UNREGISTER	非保留
UNSIGNED	保留
UNTIL	非保留
UNUSABLE	非保留
UPDATE	非保留
UPGRADE	非保留
URL	非保留
USAGE	非保留
USE	保留
USER	保留
USER_RESOURCES	非保留
USE_FRM	非保留
USING	保留
UTC_DATE	保留
UTC_TIME	保留
UTC_TIMESTAMP	保留
VACUUM	非保留

关键字	类型
VALID	非保留
VALIDATION	非保留
VALUE	非保留
VALUES	非保留（不能是函数或类型）
VARBINARY	保留
VARCHAR	非保留（不能是函数或类型）
VARCHAR2	非保留（不能是函数或类型）
VARCHARACTER	非保留（不能是函数或类型）
VARIABLES	非保留
VARYING	非保留
VCPU	非保留
VERBOSE	保留（可以是函数或类型）
VERSION	非保留
VIEW	非保留
VIRTUAL	保留
VISIBLE	非保留
VOLATILE	非保留
WAIT	非保留
WARNINGS	非保留
WEEK	非保留
WEIGHT_STRING	非保留
WHEN	保留
WHERE	保留
WHILE	保留
WINDOW	保留
WITH	保留
WITHIN	非保留
WITHOUT	非保留
WORKComments	非保留
WRAPPER	非保留

关键字	类型
WRITE	保留
X509	非保留
XA	非保留
XID	非保留
XML	非保留
XOR	保留
YEAR	非保留（不能是函数或类型）
YEAR_MONTH	保留
ZEROFILL	保留
ZONE	非保留

4.3 字符集与字符序

字符集（Character Set）是字符的编码规则，字符序（Collation）是字符的排序规则，本章主要对M-Compatibility下的字符集、字符序进行介绍，以下介绍的字符集、字符序规则和语法仅在M-Compatibility下支持。

- M-Compatibility的字符集和字符序存在以下要求：
  - 每一个字符集都有一个或多个字符序，只有一个字符序为字符集的默认字符序。
  - 每一个字符序仅有一个相关联的字符集。
  - 相同数据使用不同的字符序排序结果可能会不同。
  - utf8mb4与utf8为同一字符集。
  - 建议表字段和server_encoding选择相同字符集，避免转码带来的性能损耗。
- M-Compatibility的字符集和字符序支持以下功能：
  - 支持多种字符集存储字符串。
  - 支持使用字符序比较字符串。
  - 支持数据库级、模式级、表级、字段级字符集和字符序。

说明

除SQL_ASCII库外，其他字符集的数据库支持多字符集混用。

支持的字符集

M-Compatibility下支持的字符集如下：

表 4-2 M-Compatibility 字符集列表

字符集	说明	默认字符序
utf8	针对Unicode的可变长度字符编码，字符编码长度1~4字节。	utf8mb4_general_ci
utf8mb4	与utf8为同一字符集。	utf8mb4_general_ci
gbk	国标汉字编码扩展字符集。	gbk_chinese_ci
gb18030	国标汉字编码字符集。	gb18030_chinese_ci
binary	二进制伪字符集。	binary
latin1	拉丁字符集。	latin1_swedish_ci

说明

- 当前数据库级、模式级、表级和字段级语法，仅支持指定为上述的字符集。
- binary字符集实际通过已有字符集SQL_ASCII实现。
- 当前GaussDB字符集之间的转换逻辑相比于mysql存在差异，因此可能存在一些特殊字符在M-Compatibility下可以转换，而mysql下转换失败；不建议使用这些生僻的特殊字符。
- 目前GaussDB对不属于当前字符集的非法字符未执行严格的编码逻辑校验，可能导致此类非法字符成功输入。

支持的字符序

M-Compatibility下支持的字符序如下：

表 4-3 M-Compatibility 字符序列表

字符序	所属字符集	说明	空白填充
utf8_bin	utf8	使用二进制排序规则。	支持
utf8_general_ci	utf8	使用通用排序规则。	支持
utf8_unicode_ci	utf8	使用基于 Unicode Collation Algorithm (UCA) 的排序规则。	支持
utf8mb4_bin	utf8mb4	同utf8_bin。	支持
utf8mb4_general_ci	utf8mb4	同utf8_general_ci。	支持
utf8mb4_unicode_ci	utf8mb4	同utf8_unicode_ci。	支持

字符序	所属字符集	说明	空白填充
utf8mb4_0900_ai_ci	utf8mb4	使用基于 Unicode Collation Algorithm (UCA) 的排序规则。	支持
gbk_bin	gbk	使用二进制排序规则。	支持
gbk_chinese_ci	gbk	使用中文语言(拼音)排序规则。	支持
gb18030_bin	gb18030	使用二进制排序规则。	支持
gb18030_chinese_ci	gb18030	使用中文语言(拼音)排序规则。	支持
binary	binary	使用二进制排序规则。	不支持
latin1_swedish_ci	latin1	使用瑞典语排序规则。	支持
latin1_bin	latin1	使用二进制排序规则。	支持

说明

- 字符序名称是以和它们相关联的字符集的名称作为开头，通常后面加一个或多个表示其他字符序特征的后缀，例如: _bin表示二进制排序规则， _ci表示不区分大小写。
- 当字符序支持空白填充属性时，字符串进行比较时忽略末尾空格，例如 : 'A ' = 'A'。

4.3.1 客户端连接的字符集和字符序

含有字符集属性的数据在服务端和客户端之间传输会自动进行编码转换。当服务端接收到客户端发送的SQL语句后，会将其字符集编码由客户端字符集client_encoding向数据库字符集server_encoding进行编码转换。查询结果数据发送到客户端之前也会将数据向客户端字符集character_set_results进行编码转换。

系统参数说明

- server_encoding
  创建M-Compatibility时指定的字符集，详见**创建M-Compatibility数据库及用户**。
- client_encoding
  客户端的字符集。可以通过SET NAMES语句进行修改，详见**SET**。
- character_set_connection
  SQL语句中未指定字符集的字符串常量的默认字符集。
- collation_connection
  SQL语句中未指定字符序的字符串常量的默认字符序。

- character_set_results
  返回结果的字符集。
- character_set_database
  显示当前database、schema的字符集。
- collation_database
  显示当前database、schema的字符序。
- character_set_server
  用于指定创建database、schema时的字符集。
- collation_server
  用于指定创建database、schema时的字符序。

  说明

  - 对于一个非字符类型对象转换为字符类型的表达式，其结果的字符集和字符序为character_set_connection和collation_connection。
  - 字符类型的绑定参数的字符集和字符序默认为系统参数character_set_connection和collation_connection设置的值，绑定参数输入任何值将认作上述字符集。
  - 字符数据转换编码的过程中会校验字符的编码，如果不符合，将会上报"invalid byte sequence for encoding xxx"的异常。

4.3.2 库级字符集和字符序

创建库/模式并指定默认字符集和字符序。

```
CREATE {DATABASE | SCHEMA} [IF NOT EXISTS] db_name
    [create_option] ...
create_option: [DEFAULT] {
    CHARACTER SET [=] default_charset
  | CHAR SET [=] default_charset
  | CHARSET [=] default_charset
  | COLLATE [=] default_collation
}
```

修改库/模式的默认字符集、字符序属性。

```
ALTER {DATABASE | SCHEMA} db_name
    [create_option] ...
create_option: [DEFAULT] {
    CHARACTER SET [=] default_charset
  | CHAR SET [=] default_charset
  | CHARSET [=] default_charset
  | COLLATE [=] default_collation
}
```

语法说明：

- **db_name**
  模式名称。
  取值范围：字符串，要符合标识符的命名规范。
- **default_charset**
  指定模式的默认字符集，单独指定时会将模式的默认字符序设置为指定的字符集的默认字符序。
- **default_collation**
  指定模式的默认字符序，单独指定时会将模式的默认字符集设置为指定的字符序对应的字符集。

GaussDB通过以下方式选择模式的字符集和字符序：

- 如果同时指定了default_charset和default_collation，则使用字符集default_charset和字符序default_collation，且default_charset和default_collation需要对应，不对应时报错。
- 如果仅指定了default_charset，则使用字符集default_charset及其默认字符序。
- 如果仅指定了default_collation，则使用default_collation字符序和其对应的字符集。
- 如果既不指定default_charset也不指定default_collation，则该库/模式字符集同server_encoding，字符序为该字符集的默认字符序。

说明

- default_charset仅支持指定为带有默认字符序的字符集，如果指定的字符集没有默认字符序则报错。
- default_collation仅支持**支持的字符序**中的字符序，指定其他字符序会报错。
- 除SQL_ASCII库外，其他字符集的数据库支持多字符集混用。

示例：

```
-- 仅设置字符集，字符序为字符集的默认字符序
m_db=# CREATE SCHEMA test CHARSET utf8;

-- 仅设置字符序，字符集为字符序关联的字符集
m_db=# CREATE SCHEMA test COLLATE utf8_bin;

-- 同时设置字符集与字符序，字符集和字符序需对应
m_db=# CREATE SCHEMA test CHARSET utf8 COLLATE utf8_bin;

-- 将test的默认字符集修改为utf8mb4，默认字符序修改为utf8mb4_bin。
m_db=# ALTER SCHEMA test CHARSET utf8mb4 COLLATE utf8mb4_bin;
```

4.3.3 表级字符集和字符序

设置表的默认字符集和默认字符序，创建表操作请参见**CREATE TABLE**。

```
CREATE TABLE table_name (column_list)
    [ [DEFAULT] {CHARACTER SET | CHAR SET | CHARSET}[ = ] default_charset ]
    [ [DEFAULT] COLLATE [ = ] default_collation ]
```

修改表的默认字符集和默认字符序，修改不会影响表中当前已经存在的列，修改表操作请参见**ALTER TABLE**。

```
ALTER TABLE table_name
    [ [DEFAULT] {CHARACTER SET | CHAR SET | CHARSET}[ = ] default_charset ]
    [ [DEFAULT] COLLATE [ = ] default_collation ]
```

修改表的默认字符集和默认字符序为指定的值，同时将表中的所有字符类型的字段的字符集和字符序设置为指定的值，并将字段里的数据转换为新字符集编码。

```
ALTER TABLE table_name
    CONVERT TO {CHARACTER SET | CHAR SET | CHARSET}charset [ COLLATE collation ]
```

参数说明

- **table_name**

  表名称。
- **default_charset**

指定表的默认字符集，单独指定时会将表的默认字符序设置为指定的字符集的默认字符序。

- **default_collation**

  指定表的默认字符序，单独指定时会将表的默认字符集设置为指定的字符序对应的字符集。

GaussDB通过以下方式选择表的字符集和字符序：

- 如果同时指定了default_charset和default_collation，则使用字符集default_charset和字符序default_collation，且default_charset和default_collation需要对应，不对应会产生报错。
- 如果仅指定了default_charset，则使用字符集default_charset及其默认字符序。
- 如果仅指定了default_collation，则使用default_collation字符序和其对应的字符集。
- 如果既不指定default_charset也不指定default_collation，则使用该表所在的模式的默认字符集和默认字符序作为表的默认字符集和表的默认字符序。

说明

- default_charset仅支持指定为带有默认字符序的字符集，如果指定的字符集没有默认字符序则报错。
- default_collation仅支持**支持的字符序**中的字符序，指定其他字符序报错。
- 如果表的默认字符序为BINARY ，则表中未指定字符序的文本类型会转换为对应的二进制类型，且字符序设置为binary。
- 外键与外键引用列字符序不同时，禁止创建外键。
- 如果表或者列上有外键/外键引用列时，无法转换/modify 字符集/字符序。
- 除SQL_ASCII库外，其他字符集的数据库支持多字符集混用。

示例：

```
-- 仅设置字符集，字符序为字符集的默认字符序
m_db=# CREATE TABLE test(c1 text) CHARSET utf8;

-- 仅设置字符序，字符集为字符序关联的字符集
m_db=# CREATE TABLE test(c1 text) COLLATE utf8_bin;

-- 同时设置字符集与字符序，字符集和字符序需对应
m_db=# CREATE TABLE test(c1 text)  CHARSET utf8 COLLATE utf8_bin;

-- 将表中字符类型字段的数据转化为utf8mb4编码，并设置表和字段的字符序为utf8mb4_bin
m_db=# ALTER TABLE test CONVERT TO CHARSET utf8mb4 COLLATE utf8mb4_bin;

-- 修改表的默认字符集为utf8mb4，默认字符序为utf8mb4_bin
m_db=# ALTER TABLE test CHARSET utf8mb4 COLLATE utf8mb4_bin;
```

4.3.4 列级字符集和字符序

每个字符串类型（即类型为CHAR 、VARCHAR、 TEXT等字符串类型）的列都可以设置列的字符集和列的字符序。

```
CREATE TABLE table_name (
    column_name data_type
    [ {CHARACTER SET | CHAR SET | CHARSET} charset ]
    [ COLLATE collation ]);
```

语法说明：

- **table_name**

  表名称。

- **data_type**

  字段的数据类型，字符串类型支持字符集、字符序语法。

- **CHARACTER SET | CHAR SET | CHARSET charset**

  指定表字段的字符集，单独指定时会将字段的字符序设置为指定的字符集的默认字符序。

- **COLLATE collation**

  COLLATE子句指定列的字符序（该列的数据类型必须支持字符序）。如果没有指定，则使用默认的排序规则。

设置方式：

M-Compatibility通过以下方式选择表字段的字符集和字符序：

- 如果同时指定了charset和collation ，则使用字符集charset和字符序collation，且charset和collation需要一一对应，不对应会产生报错。
- 如果仅指定了charset，则使用字符集charset及其默认字符序。
- 如果仅指定了collation，则使用与collation关联的字符集和指定的字符序。
- 如果既不指定charset也不指定collation，则使用表的默认字符集和默认字符序。

说明

- default_charset仅支持指定为带有默认字符序的字符集，如果指定的字符集没有默认字符序则报错。
- default_collation仅支持**支持的字符序**中的字符序，指定其他字符序时会提示报错。
- 如果列的字符序为BINARY ，则表中未指定字符序的文本类型会转换为对应的二进制类型，且字符序设置为binary。
- 分区表的分区键的字符集必须与数据库字符集相同。
- 除SQL_ASCII库外，其他字符集的数据库支持多字符集混用。

示例：

```
-- 仅设置字符集，字符序为字符集的默认字符序
m_db=# CREATE TABLE test(c1 text CHARSET utf8);

-- 仅设置字符序，字符集为字符序关联的字符集
m_db=# CREATE TABLE test(c1 text COLLATE utf8_bin);

-- 同时设置字符集与字符序，字符集和字符序需对应
m_db=# CREATE TABLE test(c1 text CHARSET utf8 COLLATE utf8_bin);
```

4.3.5 字符类型表达式的字符集和字符序

每一个字符类型的表达式含有字符集和字符序属性。字符串常量的默认字符集与字符序由系统参数character_set_connection和collation_connection决定。

字符集语法：

```
[_charset_name]'string'
```

语法说明：

charset_name

指定一个字符集，解析后面的string表达式，charset_name为合法的字符集名。

示例：

```
--指定一个字符集，解析后面的string表达式
m_db=# select _utf8'GaussDB';
```

字符序语法：

其他字符串类型的表达式也可以指定字符序。

```
expression [COLLATE collation_name]
```

语法说明：

COLLATE collation_name

指定字符序的名称，用于设置这个字符串的字符序属性。指定的字符序必须是表达式的字符集允许的字符序。表达式的数据类型只能是支持字符序的数据类型。

示例：

```
--使用collate语句指定字符序
m_db=# SELECT 'a' COLLATE utf8mb4_general_ci = 'A';
```

说明

如不支持则会隐式转换成字符串类型，没有隐式转换路径则报错。

4.3.6 字符集和字符序合并规则

不同字符集字符序的表达式按一定优先级处理，来确定字符串比较运算时的使用的字符序和表达式的字符集。

字符序优先级

不同表达式字符序优先级由高到低排列如下：

- COLLATE语法拥有最高优先级。
- 含有字符序冲突的表达式（如：两个不同字符序的字符串拼接表达式）。
- 支持字符序的数据类型的列、用户自定义变量、存储过程参数、CASE表达式等。
- 特定的系统函数（如：version()和opengauss_version()函数表达式）。
- 字符串常量和绑定参数。
- NULL表达式以及表达式的数据类型不支持字符序的表达式。

当两个表达式字符序不同时，使用字符序优先级最高的表达式的字符序。

示例：

```
m_db=# CREATE TABLE t_utf8(c1 varchar(16) CHARACTER SET utf8mb4 COLLATE utf8mb4_bin);
m_db=# INSERT INTO t_utf8 VALUES('STRING');

-- 比较时使用utf8mb4_bin字符序，结果为false。
m_db=#  SELECT c1 = 'string' AS result FROM t_utf8;
 result
--------
 f
(1 row)

-- 比较时使用utf8mb4_general_ci字符序，结果为true。
m_db=# SELECT c1 = 'string' COLLATE utf8mb4_general_ci AS result FROM t_utf8;
```



```
 result
--------
 t
(1 row)

-- 将绑定参数“?”的字符序定义为collation_connection。
m_db=# PREPARE test_collation FROM 'SELECT c1 = ? AS result FROM t_utf8';

-- 绑定参数字符序与字符串常量同级别，在上一步prepare时已经确定了执行时使用c1的字符序，因此结果为false。
m_db=# SET @aa = 'string' COLLATE utf8mb4_general_ci;
m_db=# EXECUTE test_collation using @aa;
 result
--------
 f
(1 row)

-- CASE表达式与c1列同级别，即使表达式含有显式的字符序，比较时仍然采用c1的字符序，二者不相等，输出“same level”。
m_db=# SELECT CASE 'string' COLLATE utf8mb4_general_ci WHEN c1 THEN 'different level' ELSE 'same level' END AS result FROM t_utf8;
   result
------------
 same level
(1 row)

-- IN子查询与c1列同级别，即使表达式含有显式的字符序，比较时仍然采用c1的字符序，二者不相等。
m_db=# SELECT c1 FROM t_utf8 WHERE c1 in (SELECT 'string' COLLATE utf8mb4_general_ci);
 c1
----
(0 rows)
```

当两个相同优先级的表达式字符序不同时，采用以下方式处理：

- 如果两者字符集相同，优先使用后缀为_bin的字符序。
- 如果两者字符集相同，优先不使用default字符序。
- 如果两者字符集不同，优先使用BINARY字符序。
- 如果两者字符集不同，且一个为Unicode字符集，另一个不为Unicode字符集，非Unicode字符集的表达式需要转码为Unicode字符集，最终使用Unicode字符集的表达式的字符序。
- 如果不符合上述情况，两表达式将被标记为字符序冲突，字符序将被标记为无效。
  - 因COLLATE语法指定同字符集不同字符序产生的冲突，将提示异常。
  - 若相同字符集的字符序由于没有_bin字符序产生冲突，在字符串拼接函数中会使用该字符集的_bin字符序，并且该_bin字符序也仅能用于字符串拼接，其他使用场景将提示异常。

示例：

```
m_db=# CREATE TABLE t_utf8mb4_charset(
    c_utf8_bin varchar(16) character set utf8mb4 collate utf8mb4_bin,
    c_utf8_uni varchar(16) character set utf8mb4 collate utf8mb4_unicode_ci,
    c_utf8_gen varchar(16) character set utf8mb4 collate utf8mb4_general_ci);
m_db=# INSERT INTO t_utf8mb4_charset VALUES('STRING', 'String', 'string');

-- 优先使用utf8mb4_bin字符序比较，结果为false。
m_db=# SELECT c_utf8_bin = c_utf8_uni FROM t_utf8mb4_charset;
 ?column?
----------
 f
(1 row)

-- 字符序冲突，进行二进制比较，结果为false。
```



```
m_db=# SELECT c_utf8_uni = c_utf8_gen FROM t_utf8mb4_charset;
ERROR:  Collation mismatch between collations "utf8mb4_unicode_ci" and "utf8mb4_general_ci".
LINE 1: SELECT c_utf8_uni = c_utf8_gen FROM t_utf8mb4_charset;
                          ^

-- 显式指定的字符序冲突，抛出异常。
m_db=# SELECT c_utf8_uni COLLATE utf8mb4_unicode_ci = c_utf8_gen COLLATE utf8mb4_general_ci FROM
t_utf8mb4_charset;
ERROR:  collation mismatch between explicit collations "utf8mb4_unicode_ci" and "utf8mb4_general_ci"
LINE 1: ..._utf8_uni COLLATE utf8mb4_unicode_ci = c_utf8_gen COLLATE ut...
                                                  ^
```

说明

- 只有字符串类型（包括SET、ENUM类型）的对象和表达式的字符集可以和数据库的字符集不同。
- 在字符集和字符序合并规则中，认为 C、POSIX、DEFAULT字符序对应的字符集为server_encoding。

4.4 SQL 语法

4.4.1 标识符说明

M-Compatibility标识符的命名需要遵守如下规范：

- 标识符支持默认使用反引号（``）：
  - 当sql_mode设置为ANSI_QUOTES时标识符同时支持反引号（``）和双引号（""）。
  - 特殊情况下可以使用引号规避特殊字符报错。
- 无引号标识符中允许的字符：
  - ASCII：字母（a-zA-Z）、数字（0-9）、下划线（_）、美元符号（$）、井号（#）。
  - 扩展：U+0080~U+00FF。
  - 只允许字母、数字、下划线以及U+0080~U+00FF扩展字符作为开头。
- 有引号标识符中允许的字符：
  - ASCII：U+0001~U+007F。
  - 扩展：U+0080~U+FFFF。
- 标识符不支持使用U+0000或U+10000以上编码的字符。
- 标识符支持以数字开头，但不能全部由数字组成，除非在引号标识符中。

M-Compatibility标识符的大小写敏感情况如下：

- 在GUC参数lower_case_table_names为0时，完全区分大小写的标识符有库、Schema、表、视图，在创建和使用的过程中，严格区分大小写。
- 在GUC参数lower_case_table_names为1时，库、Schema、表（包括auto_increment自增列生成的序列名）、视图，在创建和使用的时候不区分大小写，且在反引号或双引号包裹以上对象时也不区分大小写。
- 列名存储区分大小写，比较时候不区分大小写。列名存储在内部按照真实的大小写存储，而正常使用则不区分大小写。
- 其余标识符：

- 在内部是全小写的，如果标识符使用双引号（需要设置SQL_MODE为ANSI_QUOTES）或反引号，则区分大小写。
- 如果指定Schema或者Table修饰时，在GUC参数lower_case_table_names为0时，Schema和Table是区分大小写的；在GUC参数lower_case_table_names为1时，Schema和Table不区分大小写。

4.4.2 SQL 语句

4.4.2.1 SQL 语法格式说明

表 4-4 SQL 语法格式说明

格式	意义
[ ]	表示用“[ ]”括起来的部分是可选的。
...	表示前面的元素可重复出现。
[x | y | ...]	表示从两个或多个选项中选取一个或者不选。
{ x | y | ... }	表示从两个或多个选项中选取一个。
[x | y | ...] [...]	表示可选多个参数或者不选，如果选择多个参数，则参数之间用空格分隔。
[x | y | ...] [,...]	表示可选多个参数或者不选，如果选择多个参数，则参数之间用逗号分隔。
{ x | y | ... } [...]	表示可选多个参数，至少选一个，如果选择多个参数，则参数之间以空格分隔。
{ x | y | ... } [,...]	表示可选多个参数，至少选一个，如果选择多个参数，则参数之间用逗号分隔。

4.4.2.2 DCL 语法一览表

DCL（Data Control Language，数据控制语言），是用来设置或更改数据库用户或角色权限的语句。

授权

M-Compatibility提供了针对数据对象和角色授权的语句，请参见**14.2.5.2.12.1-GRANT**。

收回权限

M-Compatibility提供了收回权限的语句，请参见**14.2.5.2.17.9-REVOKE**。

显示指定表中列的信息

M-Compatibility提供显示指定表中列的信息的语句，请参见**DESCRIBE**。

4.4.2.3 DDL 语法一览表

DDL（Data Definition Language，数据定义语言），用于定义或修改数据库中的对象。如：表、索引、视图等。

说明

M-Compatibility不支持数据库主节点不完整时进行DDL操作。例如：数据库中有1个数据库主节点故障时执行新建数据库、表等操作都会失败。

定义角色

角色是用来管理权限的，从数据库安全的角度考虑，可以把所有的管理和操作权限划分到不同的角色上。所涉及的SQL语句，如表4-5所示。

表 4-5 角色定义相关 SQL

功能	相关SQL
创建角色	CREATE ROLE
修改角色属性	ALTER ROLE
删除角色	DROP ROLE
删除一个数据库角色所拥有的数据库对象	DROP OWNED

定义用户

用户是用来登录数据库的，通过对用户赋予不同的权限，可以方便地管理用户对数据库的访问及操作。所涉及的SQL语句，如表4-6所示。

表 4-6 用户定义相关 SQL

功能	相关SQL
创建用户	CREATE USER
修改用户属性	ALTER USER
删除用户	DROP USER

定义数据库或模式

M-Compatibility中数据库（DATABASE）和模式（SCHEMA）是同义词，是组织、存储和管理数据的仓库，而数据库定义主要包括：创建数据库、修改数据库属性，以及删除数据库。所涉及的SQL语句，如表4-7所示。

表 4-7 数据库定义相关 SQL

功能	相关SQL
创建数据库/模式	CREATE DATABASE，CREATE SCHEMA
修改数据库/模式属性	ALTER DATABASE，ALTER SCHEMA
删除数据库/模式	DROP DATABASE，DROP SCHEMA
指定当前模式	USE

定义表

表是数据库中的一种特殊数据结构，用于存储数据对象以及对象之间的关系。所涉及的SQL语句，如表4-8所示。

表 4-8 表定义相关 SQL

功能	相关SQL
创建表	CREATE TABLE
通过查询结果集创建表	CREATE TABLE SELECT
修改表属性	ALTER TABLE
修改一个或多个表名称	RENAME TABLE
删除表	DROP TABLE

定义分区表

分区表是一种逻辑表，数据是由普通表存储的，主要用于提升查询性能。所涉及的SQL语句，如表4-9所示。

表 4-9 分区表定义相关 SQL

功能	相关SQL
创建分区表	CREATE TABLE PARTITION
创建分区	ALTER TABLE PARTITION
修改分区表属性	ALTER TABLE PARTITION
删除分区	ALTER TABLE PARTITION
删除分区表	DROP TABLE
创建二级分区表	CREATE TABLE SUBPARTITION
修改二级分区表分区	ALTER TABLE SUBPARTITION

定义索引

索引是对数据库表中一列或多列的值进行排序的一种结构，使用索引可快速访问数据库表中的特定信息。所涉及的SQL语句，如表4-10所示。

表 4-10 索引定义相关 SQL

功能	相关SQL
创建索引	CREATE INDEX
修改索引属性	ALTER INDEX
删除索引	DROP INDEX
重建索引	REINDEX

定义插件扩展

该特性为内部使用，不建议用户使用。

表 4-11 插件扩展定义相关 SQL

功能	相关SQL
创建一个新的插件扩展	CREATE EXTENSION
修改插件扩展	ALTER EXTENSION
删除插件扩展	DROP EXTENSION

定义用户组

表 4-12 用户组定义相关 SQL

功能	相关SQL
创建一个新用户组	CREATE GROUP
修改一个用户组的属性	ALTER GROUP
删除用户组	DROP GROUP

定义序列

表 4-13 序列定义相关 SQL

功能	相关SQL
向当前数据库增加一个新的序列	CREATE SEQUENCE
修改一个现有的序列的参数	ALTER SEQUENCE
从当前数据库里删除序列	DROP SEQUENCE

定义视图

表 4-14 视图定义相关 SQL

功能	相关SQL
创建一个视图	CREATE VIEW
更改视图的各种辅助属性	ALTER VIEW
数据库中强制删除已有的视图	DROP VIEW

定义函数

表 4-15 函数定义相关 SQL

功能	相关SQL
创建函数	CREATE FUNCTION
删除函数	DROP FUNCTION

定义资源标签

表 4-16 资源标签定义相关 SQL

功能	相关SQL
创建资源标签	CREATE RESOURCE LABEL
修改资源标签	ALTER RESOURCE LABEL
删除资源标签	DROP RESOURCE LABEL

定义审计策略

表 4-17 审计策略定义相关 SQL

功能	相关SQL
创建统一审计策略	**CREATE AUDIT POLICY**
修改统一审计策略	**ALTER AUDIT POLICY**
删除一个审计策略	**DROP AUDIT POLICY**

收集统计信息

收集与数据库中普通表内容相关的统计信息，请参见**ANALYZE**。

清理回收站

M-Compatibility提供清理回收站的语句，请参见**PURGE**。

定义一个对象的注释

M-Compatibility支持定义或修改一个对象的注释的语句，请参见**COMMENT**。

根据查询结果创建新表

M-Compatibility支持用于根据查询结果创建一个新表，并且将查询到的数据插入到新表的语句，请参见**SELECT INTO**。

清理表数据

M-Compatibility支持在快速地从表中删除所有行的语句，请参见**TRUNCATE**。

回收存储空间

M-Compatibility支持回收表或B-Tree索引中已经删除的行所占据的存储空间的语句，请参见**VACUUM**。

4.4.2.4 DML 语法一览表

DML（Data Manipulation Language，数据操作语言），用于对数据库表中的数据进行操作。如：插入、更新、查询、删除。

插入数据

插入数据是往数据库表中添加一条或多条记录，请参见**INSERT**。

修改数据

修改数据是修改数据库表中的一条或多条记录，请参见**UPDATE**。

查询数据

数据库查询语句SELECT是用于在数据库中检索适合条件的信息，请参见**SELECT**。

删除数据

M-Compatibility提供了删除表中指定条件的数据，请参见**DELETE**。

复制数据

M-Compatibility提供了在表和文件之间复制数据的语句，请参见**COPY**。

锁

M-Compatibility提供了多种锁模式用于控制对表中数据的并发访问，请参见**LOCK**。

操作会话

用户与数据库之间建立的连接称为会话，修改会话请参见**ALTER SESSION**。

执行匿名代码块

M-Compatibility提供执行匿名代码块的语句，请参见**DO**。

插入或替换数据

M-Compatibility提供在表中插入或者替换新的数据的语句，请参见**REPLACE**。

从外部文件导入数据

M-Compatibility提供将外部文件数据导入到数据库指定表中的语句，请参见**LOAD DATA**。

展示信息

M-Compatibility提供展示信息的语句，请参见**SHOW**。

4.4.2.5 其他语法一览表

除了DCL、DDL和DML语法，M-Compatibility还提供了其他功能的语法。

显示 SQL 语句的执行计划

M-Compatibility提供显示SQL语句的执行计划的语句，请参见**EXPLAIN**。

创建预备语句

M-Compatibility提供创建一个预备语句的语句，请参见**PREPARE**。

预备语句

表 4-18 预备语句相关 SQL

功能	相关SQL
执行一个前面准备好的预备语句	EXECUTE
删除前面编写的预备语句	DEALLOCATE，DROP PREPARE

事务相关 SQL

表 4-19 事务相关 SQL

功能	相关SQL
回滚当前事务并且撤销所有当前事务中所做的更改。	ROLLBACK
开始一个事务/启动事务。	BEGIN
	SET TRANSACTION
	START TRANSACTION
提交当前事务。	COMMIT
删除一个当前事务先前定义的保存点。	RELEASE SAVEPOINT
回滚到一个保存点。	ROLLBACK TO SAVEPOINT
在当前事务里建立一个新的保存点。	SAVEPOINT

修改/显示/恢复运行参数

表 4-20 修改/显示/恢复运行参数相关 SQL

功能	相关SQL
修改运行时配置参数	SET
显示当前运行时参数的数值	SHOW
将指定的运行时参数恢复为缺省值	RESET

设置用户标识符

表 4-21 设置用户标识符相关 SQL

功能	相关SQL
设置当前会话的当前用户标识符	SET ROLE
把当前会话里的会话用户标识和当前用户标识都设置为指定的用户	SET SESSION AUTHORIZATION

4.4.2.6 A

4.4.2.6.1 ALTER AUDIT POLICY

功能描述

修改统一审计策略。

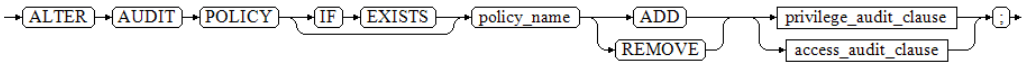
注意事项

- 审计策略的维护有权限限制，只有POLADMIN，SYSADMIN或初始用户有权限进行此操作。
- 需要打开enable_security_policy开关统一审计策略才可以生效。

语法格式

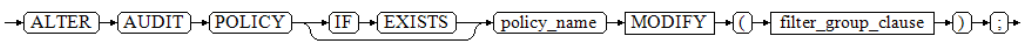
添加/删除审计策略中的操作类型。

```
ALTER AUDIT POLICY [ IF EXISTS ] policy_name { ADD | REMOVE } { [ privilege_audit_clause ]
[ access_audit_clause ] };
```



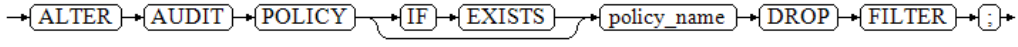
修改审计策略中的过滤条件。

```
ALTER AUDIT POLICY [ IF EXISTS ] policy_name MODIFY ( filter_group_clause );
```



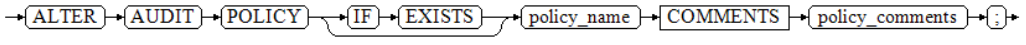
将审计策略中的过滤条件删除。

```
ALTER AUDIT POLICY [ IF EXISTS ] policy_name DROP FILTER;
```



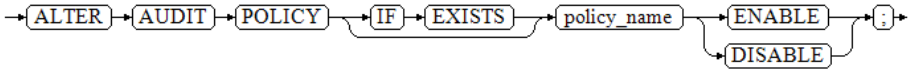
修改审计策略描述。

```
ALTER AUDIT POLICY [ IF EXISTS ] policy_name COMMENTS policy_comments;
```



打开或者关闭审计策略。

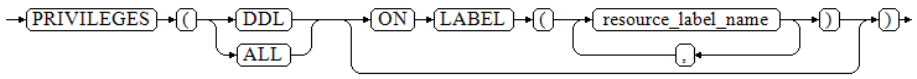
```
ALTER AUDIT POLICY [ IF EXISTS ] policy_name { ENABLE | DISABLE };
```



- privilege_audit_clause：

  审计策略中具体的DDL操作类型及目标资源标签。

  ```
  PRIVILEGES ({ DDL | ALL } [ ON LABEL ( resource_label_name [, ... ] ) ])
  ```

- access_audit_clause：

  审计策略中具体的DML操作类型及目标资源标签。

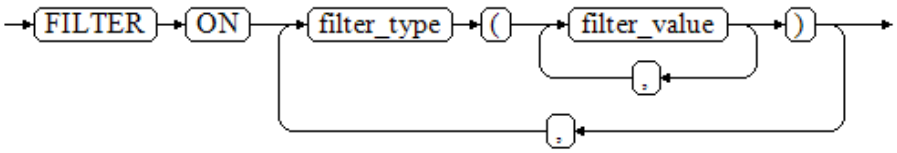
  ```
  ACCESS ({ DML | ALL } [ ON LABEL ( resource_label_name [, ... ] ) ])
  ```

- filter_group_clause：

  审计策略中的过滤条件。

  ```
  FILTER ON { filter_type ( filter_value [, ... ] ) } [, ... ]
  ```

- DDL：

  指的是针对数据库执行如下操作时进行审计。

  ```
  { ( ALTER | ANALYZE | COMMENT | CREATE | DROP | GRANT | REVOKE | SET | SHOW ) }
  ```

- DML：

  指的是针对数据库执行如下操作时进行审计。

  ```
  { ( COPY | DEALLOCATE | DELETE | EXECUTE | REINDEX | INSERT | PREPARE | SELECT | TRUNCATE | UPDATE ) }
  ```

参数说明

- **policy_name**

  审计策略名称，需要唯一，不可重复。

  取值范围：字符串，要符合**标识符命名规范**。

- **resource_label_name**

  资源标签名称。

- **DDL**

  指的是针对数据库执行如下操作时进行审计，目前支持：CREATE、ALTER、DROP、ANALYZE、COMMENT、GRANT、REVOKE、SET、SHOW。

- **DML**

  指的是针对数据库执行如下操作时进行审计，目前支持：SELECT、COPY、DEALLOCATE、DELETE、EXECUTE、INSERT、PREPARE、REINDEX、TRUNCATE、UPDATE。

- **ALL**

指的是上述DDL或DML中支持的所有对数据库的操作。当形式为{ DDL | ALL }时，ALL指所有DDL操作；当形式为{ DML | ALL }时，ALL指所有DML操作。

- **filter_type**

  指定审计策略的过滤信息，过滤类型包括：IP、ROLES、APP。

- **filter_value**

  指具体过滤信息内容。

- **policy_comments**

  用于记录策略相关的描述信息。

- **ENABLE|DISABLE**

  可以打开或关闭统一审计策略。

示例

- 添加/删除审计策略中的操作类型。

```
--创建一个对数据库执行CREATE的审计策略adt1。
m_db=# CREATE AUDIT POLICY adt1 PRIVILEGES CREATE;

--添加adt1审计策略中的DROP。
m_db=# ALTER AUDIT POLICY adt1 ADD PRIVILEGES (DROP);

--删除adt1审计策略中的DROP。
m_db=# ALTER AUDIT POLICY adt1 REMOVE PRIVILEGES (DROP);
```

- 修改审计策略的注释信息。

```
--修改adt1审计策略的注释信息为adt1_comments。
m_db=# ALTER AUDIT POLICY adt1 COMMENTS 'adt1_comments';
```

- 修改审计策略的过滤信息。

```
--创建bob_audit用户。
m_db=# CREATE USER bob_audit PASSWORD '********';

--修改adt1审计策略的过滤用户为bob_audit。
m_db=# ALTER AUDIT POLICY adt1 MODIFY (FILTER ON (ROLES(bob_audit)));

--删除bob_audit用户。
m_db=# DROP USER bob_audit;
```

- 删除审计策略的过滤条件。

```
--删除adt1审计策略的过滤条件。
m_db=# ALTER AUDIT POLICY adt1 DROP FILTER;
```

- 关闭审计策略。

```
--关闭adt1审计策略。
m_db=# ALTER AUDIT POLICY adt1  DISABLE;

--删除adt1审计策略。
m_db=# DROP AUDIT POLICY adt1;
```

相关链接

CREATE AUDIT POLICY，**DROP AUDIT POLICY**。

4.4.2.6.2 ALTER DATABASE

功能描述

M-Compatibility中DATABASE和SCHEMA是同义词，该语法允许修改数据库或模式的字符集、字符序、和所有者。

注意事项

- 只有模式的所有者或者被授予了模式ALTER权限的用户有权限执行ALTER DATABASE命令，三权分立开关关闭时，系统管理员默认拥有此权限。但要修改模式的所有者，当前用户必须是该模式的所有者或者系统管理员，且该用户是新所有者角色的成员。
- 对于除public以外的系统模式，如pg_catalog、sys等，只允许初始用户修改模式的所有者。修改系统自带模式的名称可能会导致部分功能不可用甚至影响数据库正常运行，默认情况下不允许修改系统自带模式的名称，考虑到前向兼容性，仅允许当系统在启动或升级过程中或参数allow_system_table_mods为on时修改。
- 除初始用户外，其他用户无法将schema的所有者修改为运维管理员。

语法格式

- 修改数据库所有者。

```
ALTER DATABASE database_name
    OWNER TO new_owner;
```

- 修改数据库字符集、字符序。

```
ALTER {DATABASE | SCHEMA} database_name
    [create_option] ...
create_option: [DEFAULT] {
    CHARACTER SET [=] default_charset
  | CHAR SET [=] default_charset
  | CHARSET [=] default_charset
  | COLLATE [=] default_collation
}
```

参数说明

- **database_name**

  需要修改属性的数据库名称。

  取值范围：字符串，要符合**标识符说明**。

- **new_owner**

  数据库的新所有者。

  取值范围：字符串，有效的用户名，用户名要求详见**•user_name**。

- **COLLATE [=] default_collation**

  可选。指定数据库使用的字符集。例如，通过collate = 'zh_CN.gbk'设定该参数。

  该参数的使用会影响到对字符串的排序顺序（如使用ORDER BY执行，以及在文本列上使用索引的顺序）。默认是使用模板数据库的字符集。

  取值范围：参考**库级字符集和字符序**。

- **{CHAR SET | CHARSET | CHARACTER SET} [=] default_charset**

  可选。指定数据库使用的字符分类。例如，通过CHARSET = 'zh_CN.gbk'设定该参数。该参数的使用会影响到字符的分类，如大写、小写和数字。默认是使用模板数据库的字符分类。

  取值范围：参考**库级字符集和字符序**。

示例

- 修改数据库字符集、字符序

```
--创建数据库test_db1。
m_db=# CREATE DATABASE test_db1;
--将test_db1的字符集改为utf8。
```



```
m_db=# ALTER DATABASE test_db1 CHARSET utf8;
--将test_db1的字符序改为utf8mb4_bin。
m_db=# ALTER DATABASE test_db1 COLLATE utf8mb4_bin;
```

- 修改数据库所有者。

```
--创建用户scott。
m_db=# CREATE USER scott PASSWORD '********';
--将test_db1的所有者修改为scott。
m_db=# ALTER DATABASE test_db1 OWNER TO scott;
```

相关链接

CREATE DATABASE，**DROP DATABASE**。

4.4.2.6.3 ALTER EXTENSION

功能描述

修改插件扩展。当前不支持用户使用此语法。该特性为内部使用，不建议用户使用。

注意事项

ALTER EXTENSION 修改一个已安装的扩展的定义，要使用该接口，需要设置support_extended_features为true。这里有几种方式：

- UPDATE

  这种方式更新这个扩展到一个新的版本。这个扩展必须满足一个适用的更新脚本(或者一系列脚本) 这样就能修改当前安装版本到一个要求的版本。

- SET SCHEMA

  这种方式移动扩展对象到另一个模式。这个扩展必须relocatable才能使命令成功。

- ADD member_object

  这种方式添加一个已存在对象到扩展。这主要在扩展更新脚本上有用。这个对象接着会被视为扩展的成员; 显而易见，该对象只能通过取消扩展来取消 。

- DROP member_object

  这个方式从扩展上移除一个成员对象。主要在扩展更新脚本上有用，这个对象没有被取消，只是从扩展里分开了。

  您必须拥有扩展来使用 ALTER EXTENSION。这个 ADD/DROP 方式要求 添加/删除对象的所有权。

语法格式

- 修改扩展的版本

```
ALTER EXTENSION name UPDATE [ TO new_version ];
```

- 修改扩展的模式

```
ALTER EXTENSION name SET SCHEMA new_schema;
```

- 添加或删除扩展的成员对象

```
ALTER EXTENSION name { ADD | DROP } member_object;
```

其中成员对象member_object写法为：

```
{ COLLATION object_name |
  FOREIGN DATA WRAPPER object_name |
```



```
SCHEMA object_name |
SEQUENCE object_name |
SERVER object_name |
TABLE object_name |
VIEW object_name}
```

参数说明

- **name**

  已安装扩展的名称。

- **new_version**

  扩展的新版本。可以通过被标识符和字面字符重写。如果不指定的扩展的新版本，ALTER EXTENSION UPDATE会更新到扩展的控制文件中显示的默认版本。

- **new_schema**

  扩展的新模式。

- **object_name**

  agg_name

  function_name

  operator_name

  从扩展里被添加或移除的对象的名称。包含表、聚合 、域、外链表、函数、操作符、操作符类、操作符族、序列、文本搜索对象、类型和能被模式合格的视图的名称。

- **agg_type**

  在聚合函数操作上的一个输入数据类型，去引用一个零参数聚合函数，写 * 代替这些输入数据类型列表。

- **source_type**

  强制转换的源数据类型的名称。

- **target_type**

  强制转换的目标数据类型的名称。

- **argmode**

  这个函数参数的模型：IN、OUT、INOUT或者 VARIADIC。如果省略的话，默认值为IN。ALTER EXTENSION 不关心OUT参数 ，因为确认函数的一致性只需要输入参数，因此列出 IN、INOUT和 VARIADIC参数就足够了。

- **argname**

  函数参数的名称。ALTER EXTENSION不关心参数名称，确认函数的一致性只需要参数数据类型。

- **argtype**

  函数参数的数据类型（可以有模式修饰）。

- **left_type**

  right_type

  操作符参数的数据类型（可以有模式修饰），为前缀或后缀运算符的丢失参数写 NONE 。

示例

更新 plpgsql 扩展到版本 2.0：


```
m_db=# ALTER EXTENSION plpgsql UPDATE TO '2.0';
```

更新 plpgsql 扩展的模式为utils：

```
m_db=# ALTER EXTENSION plpgsql SET SCHEMA utils;
```

相关链接

CREATE EXTENSION，**DROP EXTENSION**

4.4.2.6.4 ALTER GROUP

功能描述

更改角色名称或者成员关系。

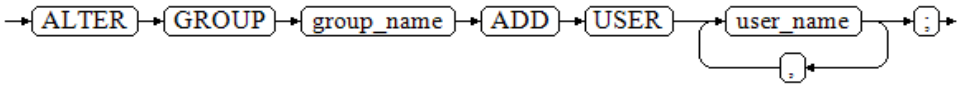
注意事项

- ALTER GROUP为非SQL标准语法，不推荐使用。
- 其中ADD USER、DROP USER两个子句用于向用户组增加或删除用户（任何用户都可以是“用户”或者“用户组”）。这两个子句等效于授予或者回收成员关系，因此建议使用GRANT或者REVOKE语句。
- RENAME TO子句修改用户组名称，等效于ALTER USER命名角色。

语法格式

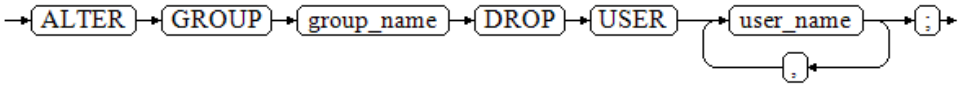
- 向用户组中添加用户。

```
ALTER GROUP group_name
    ADD USER user_name [, ... ];
```



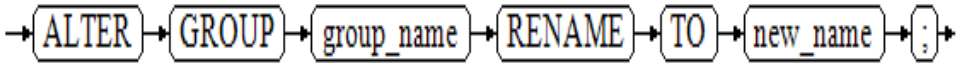
- 从用户组中删除用户。

```
ALTER GROUP group_name
    DROP USER user_name [, ... ];
```



- 修改用户组的名称。

```
ALTER GROUP group_name
    RENAME TO new_name;
```



参数说明

- **user_name**
  现有角色名。
  取值范围：已存在的角色名，角色名要求详见**•role_name**。
- **group_name**
  现有用户组名。

取值范围：已存在的角色名，角色名要求详见•role_name。

- **new_name**

  新角色名称。

  取值范围：字符串，要符合标识符命名规范，角色名要求详见•role_name。

示例

- 重命名用户组。

```
--创建用户组test。
m_db=# CREATE GROUP test PASSWORD '********';

--修改用户名，等效于ALTER GROUP RENAME。
m_db=# ALTER GROUP test RENAME TO tu_a1;
```

- 向用户组中添加或者删除用户。

```
--创建用户tu_a2、tu_a3。
m_db=# CREATE ROLE tu_a2 PASSWORD '********';
m_db=# CREATE ROLE tu_a3 PASSWORD '********';

--向用户组tu_a1中添加用户tu_a2。
m_db=# ALTER GROUP tu_a1 ADD USER tu_a2;

--如上SQL等效于GRANT语句。
m_db=# GRANT tu_a1 TO tu_a3;

--查询。
m_db=# SELECT groname, grolist FROM pg_group WHERE groname = 'tu_a1';
 groname |    grolist
---------+---------------
 tu_a1   | {25590,25593}
(1 row)
m_db=# SELECT rolname, oid FROM pg_roles WHERE oid IN (25590,25593);
 rolname |  oid
---------+-------
 tu_a2   | 25590
 tu_a3   | 25593
(2 rows)

--删除。
m_db=# DROP ROLE tu_a1,tu_a2,tu_a3;
```

相关链接

CREATE GROUP，DROP GROUP，ALTER ROLE

4.4.2.6.5 ALTER INDEX

功能描述

用于修改已有索引的定义。

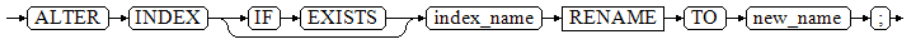
注意事项

- 索引的所有者、拥有索引所在表的INDEX权限的用户或者被授予了ALTER ANY INDEX权限的用户有权限执行此命令，当三权分立开关关闭时，系统管理员默认拥有此权限。
- 请勿在同一基表上保持大量的不可见索引，否则可能会对INSERT、UPDATE、DELETE等DML操作的性能产生影响。

语法格式

- 重命名表索引的名称。

```
ALTER INDEX [ IF EXISTS ] index_name
    RENAME TO new_name;
```

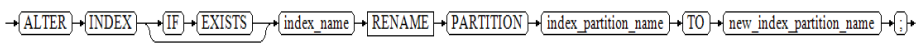


- 设置表索引不可用。

```
ALTER INDEX [ IF EXISTS ] index_name UNUSABLE;
```

- 重命名索引分区。

```
ALTER INDEX [ IF EXISTS ] index_name
    RENAME PARTITION index_partition_name TO new_index_partition_name;
```



参数说明

- **index_name**

  要修改的索引名称。

- **IF EXISTS**

  如果指定的索引不存在，则发出一个notice而不是error。

- **RENAME TO new_name**

  只改变索引的名称。对存储的数据没有影响。

  new_name

  新的索引名。

  取值范围：字符串，且符合**标识符说明**。

- **RENAME PARTITION index_partition_name TO new_index_partition_name**

  用于重命名索引分区。

- **new_index_partition_name**

  新索引分区名。

- **index_partition_name**

  索引分区名。

示例

- 重命名索引。

```
--创建test1表并为其创建索引。
m_db=# CREATE TABLE test1(col1 INT, col2 INT);
m_db=# CREATE INDEX aa ON test1(col1);

--将索引aa重命名为idx_test1_col1。
m_db=# ALTER INDEX aa RENAME TO idx_test1_col1;

--查询test1表上的索引信息。
m_db=# SELECT tablename,indexname,tablespace FROM pg_indexes WHERE tablename = 'test1';
 tablename |   indexname    | tablespace
-----------+----------------+------------
 test1     | idx_test1_col1 |
(1 row)
```

- 重命名索引分区。

```
--创建分区表test2。
m_db=# CREATE TABLE test2(col1 int, col2 int) PARTITION BY RANGE (col1)(
    PARTITION p1 VALUES LESS THAN (100),
```



```
    PARTITION p2 VALUES LESS THAN (200)
);

--创建分区索引。
m_db=# CREATE INDEX idx_test2_col1 ON test2(col1) LOCAL(
    PARTITION p1,
    PARTITION p2
);
--重命名索引分区。
m_db=# ALTER INDEX idx_test2_col1 RENAME PARTITION p1 TO p1_test2_idx;
m_db=# ALTER INDEX idx_test2_col1 RENAME PARTITION p2 TO p2_test2_idx;

--查询索引idx_test2_col1分区的名称。
m_db=# SELECT relname FROM pg_partition WHERE parentid = 'idx_test2_col1'::regclass;
   relname
--------------
 p1_test2_idx
 p2_test2_idx
(2 rows)
```

相关链接

CREATE INDEX，**DROP INDEX**，**REINDEX**

4.4.2.6.6 ALTER RESOURCE LABEL

功能描述

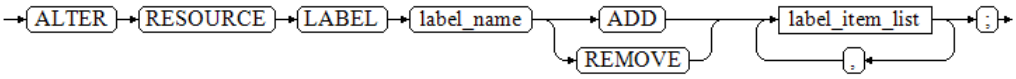
ALTER RESOURCE LABEL语句用于修改资源标签。

注意事项

只有poladmin、sysadmin或初始用户才能执行此操作。

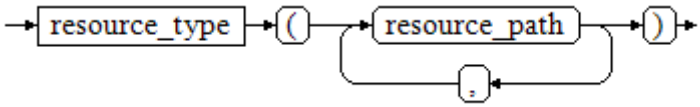
语法格式

```
ALTER RESOURCE LABEL label_name {ADD|REMOVE}
  label_item_list[, ...];
```



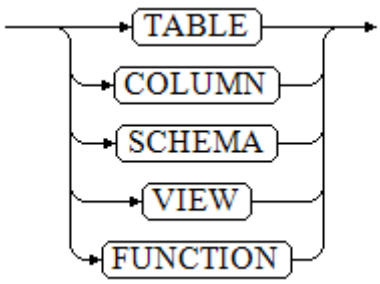
- label_item_list：

  ```
  resource_type(resource_path[, ...])
  ```

- resource_type：

  ```
  TABLE | COLUMN | SCHEMA | VIEW | FUNCTION
  ```

参数说明

- **label_name**
  资源标签名称。
  取值范围：字符串，要符合**标识符命名规范**。
- **resource_type**
  指的是要标记的数据库资源类型。
- **resource_path**
  指的是描述具体的数据库资源的路径。

示例

```
--创建基本表table_for_label。
m_db=# CREATE TABLE table_for_label(col1 int, col2 text);

--创建资源标签table_label。
m_db=# CREATE RESOURCE LABEL table_label ADD COLUMN(table_for_label.col1);

--将col2添加至资源标签table_label中。
m_db=# ALTER RESOURCE LABEL table_label ADD COLUMN(table_for_label.col2);

--将资源标签table_label中的一项移除。
m_db=# ALTER RESOURCE LABEL table_label REMOVE COLUMN(table_for_label.col1);

--删除资源标签table_label。
m_db=# DROP RESOURCE LABEL table_label;

--删除基本表table_for_label。
m_db=# DROP TABLE table_for_label;
```

相关链接

CREATE RESOURCE LABEL，**DROP RESOURCE LABEL**。

4.4.2.6.7 ALTER ROLE

功能描述

修改角色属性。

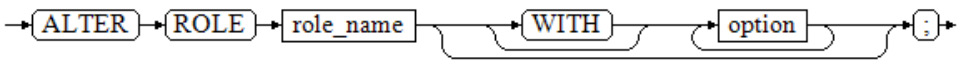
注意事项

无。

语法格式

- 修改角色的权限。

```
ALTER ROLE role_name [ [ WITH ] option [ ... ] ];
```



  - 其中权限项子句option为：

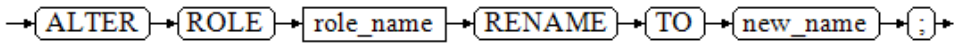
```
{CREATEDB | NOCREATEDB}
  | {CREATEROLE | NOCREATEROLE}
  | {AUDITADMIN | NOAUDITADMIN}
  | {SYSADMIN | NOSYSADMIN}
  | {MONADMIN | NOMONADMIN}
```



```
    | {OPRADMIN | NOOPRADMIN}
    | {POLADMIN | NOPOLADMIN}
    | {USEFT | NOUSEFT}
    | {INHERIT | NOINHERIT}
    | {LOGIN | NOLOGIN}
    | {PERSISTENCE | NOPERSISTENCE}
    | CONNECTION LIMIT connlimit
    | PASSWORD { 'password' [EXPIRED] | DISABLE }
    | IDENTIFIED BY { 'password' [ REPLACE 'old_password' | EXPIRED ] | DISABLE }
    | VALID BEGIN 'timestamp'
    | VALID UNTIL 'timestamp'
    | USER GROUP 'groupuser'
    | NODE GROUP logic_cluster_name
    | ACCOUNT { LOCK | UNLOCK }
```

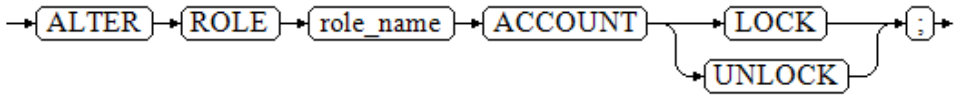
- 修改角色的名称：

```
ALTER ROLE role_name
    RENAME TO new_name;
```



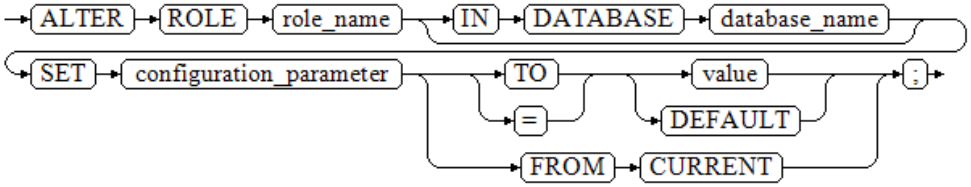
- 锁定或解锁：

```
ALTER ROLE role_name
    ACCOUNT { LOCK | UNLOCK };
```



- 设置角色的配置参数：

```
ALTER ROLE role_name [ IN DATABASE database_name ]
    SET configuration_parameter {{ TO | = } { value | DEFAULT } | FROM CURRENT};
```



- 重置角色的所有配置参数：

```
ALTER ROLE role_name
    [ IN DATABASE database_name ] RESET ALL;
```

参数说明

- **role_name**

  现有角色名。

  取值范围：已存在的角色名，角色名要求详见•role_name。

- **IN DATABASE database_name**

  表示修改角色在指定数据库上的参数。

- **SET configuration_parameter {{ TO | = } { value | DEFAULT } | FROM CURRENT}**

  设置角色的参数。ALTER ROLE中修改的会话参数只针对指定的角色，且在下一次该角色启动的会话中有效。

  取值范围：

  configuration_parameter和value的取值请参见SET。

  DEFAULT：表示清除configuration_parameter参数的值，configuration_parameter参数的值将继承本角色新产生的SESSION的默认值。

FROM CURRENT：取当前会话中的值设置为configuration_parameter参数的值。

- **RESET ALL**

  把所有configuration_parameter参数的恢复到默认。

- **ACCOUNT LOCK | ACCOUNT UNLOCK**
  - ACCOUNT LOCK：锁定账户，禁止登录数据库。
  - ACCOUNT UNLOCK：解锁账户，允许登录数据库。
- **PGUSER**

  当前版本不允许修改角色的PGUSER属性

- **{PASSWORD|IDENTIFIED BY} 'password'**

  重置或修改用户密码。除了初始用户外其他管理员或普通用户修改自己的密码需要输入正确的旧密码。只有初始用户、三权分立关闭时的系统管理员（sysadmin）或拥有创建用户（CREATEROLE）权限的用户才可以重置普通用户密码，无需输入旧密码。初始用户可以重置系统管理员的密码，系统管理员不允许重置其他系统管理员的密码。应当使用单引号将用户密码括起来。

- **EXPIRED**

  设置密码失效。只有初始用户、系统管理员（sysadmin）或拥有创建用户（CREATEROLE）权限的用户才可以设置用户密码失效，其中系统管理员只有在三权分立关闭时，才可以设置自己或其他系统管理员密码失效。不允许设置初始用户密码失效。

  密码失效的用户可以登录数据库但不能执行查询操作，只有修改密码或由管理员重置密码后才可以恢复正常查询操作。

其他参数请参见CREATE ROLE的**参数说明**。

示例

请参见CREATE ROLE的**示例**。

相关链接

CREATE ROLE，**DROP ROLE**，**SET ROLE**

4.4.2.6.8 ALTER SCHEMA

功能描述

M-Compatibility中DATABASE和SCHEMA是同义词，允许修改数据库或模式的字符集、字符序、所有者。

注意事项

- 只有模式的所有者或者被授予了模式ALTER权限的用户有权限执行ALTER SCHEMA命令，三权分立开关关闭时，系统管理员默认拥有此权限。但要修改模式的所有者，当前用户必须是该模式的所有者或者系统管理员，且该用户是新所有者角色的成员。
- 对于除public以外的系统模式，如pg_catalog、sys等，只允许初始用户修改模式的所有者。修改系统自带模式的名称可能会导致部分功能不可用甚至影响数据库正常运行，默认情况下不允许修改系统自带模式的名称，考虑到前向兼容性，仅允许当系统在启动或升级过程中或参数allow_system_table_mods为on时修改。

- 除初始用户外，其他用户无法将schema的所有者修改为运维管理员。

语法格式

- 修改数据库所有者。

```
ALTER SCHEMA schema_name
    OWNER TO new_owner;
```

- 修改数据库字符集、字符序

```
ALTER {DATABASE | SCHEMA} schema_name
    [create_option] ...
create_option: [DEFAULT] {
    CHARACTER SET [=] default_charset
  | CHAR SET [=] default_charset
  | CHARSET [=] default_charset
  | COLLATE [=] default_collation
}
```

参数说明

- **schema_name**

  需要修改属性的数据库名称。

  取值范围：字符串，要符合**标识符说明**。

- **new_owner**

  数据库的新所有者。

  取值范围：字符串，有效的用户名，用户名要求详见**•user_name**。

- **COLLATE [=] default_collation**

  可选。指定数据库使用的字符集。例如，通过collate = 'zh_CN.gbk'设定该参数。

  该参数的使用会影响到对字符串的排序顺序（如使用ORDER BY执行，以及在文本列上使用索引的顺序）。默认是使用模板数据库的字符集。

  取值范围：参考**库级字符集和字符序**。

- **{CHAR SET | CHARSET | CHARACTER SET} [=] default_charset**

  可选。指定数据库使用的字符分类。例如，通过CHARSET = 'zh_CN.gbk'设定该参数。该参数的使用会影响到字符的分类，如大写、小写和数字。默认是使用模板数据库的字符分类。

  取值范围：参考**库级字符集和字符序**。

示例

- 修改数据库字符集、字符序。

```
--创建数据库test_db1。
m_db=# CREATE SCHEMA test_db1;
--将test_db1的字符集改为utf8。
m_db=# ALTER SCHEMA test_db1 CHARSET utf8;
--将test_db1的字符序改为utf8mb4_bin
m_db=# ALTER SCHEMA test_db1 COLLATE utf8mb4_bin;
```

- 修改数据库所有者。

```
--创建用户scott。
m_db=# CREATE USER scott PASSWORD '********';
--将test_db1的所有者修改为scott。
m_db=# ALTER SCHEMA test_db1 OWNER TO scott;
```

相关链接

CREATE SCHEMA，**DROP SCHEMA**。

4.4.2.6.9 ALTER SEQUENCE

功能描述

修改一个现有的序列的参数。

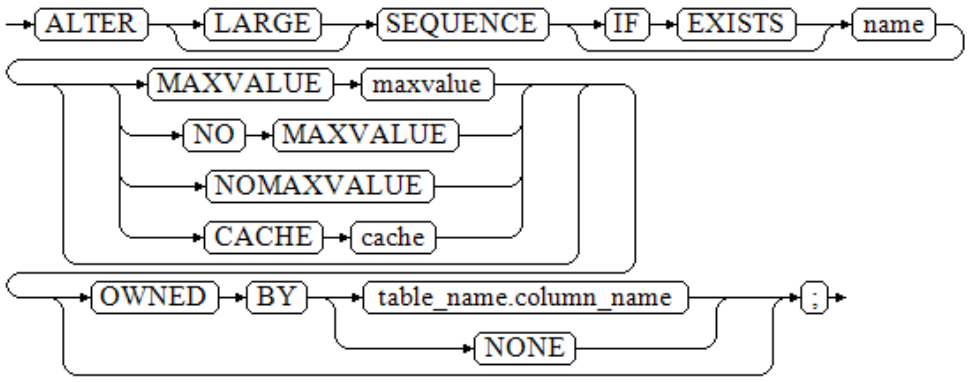
注意事项

- 序列的所有者或者被授予了序列ALTER权限的用户或者被授予了ALTER ANY SEQUENCE权限的用户才能执行ALTER SEQUENCE命令，三权分立开关关闭时，系统管理员默认拥有该权限。但要修改序列的所有者，当前用户必须是该序列的所有者或者系统管理员，且该用户是新所有者角色的成员。
- 当前版本仅支持修改拥有者、归属列、最大值和cache缓存值。若要修改其他参数，可以删除重建，并用Setval函数恢复当前值。
- ALTER SEQUENCE MAXVALUE不支持在事务、函数和存储过程中使用。
- 修改序列的最大值后，会清空该序列在所有会话的cache。
- ALTER SEQUENCE会阻塞nextval、setval、currval和lastval的调用。

语法格式

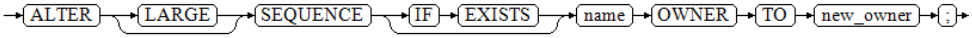
- 修改序列最大值、归属列和缓存值：

```
ALTER SEQUENCE [ IF EXISTS ] name
    [MAXVALUE maxvalue | NO MAXVALUE | NOMAXVALUE | CACHE cache]
    [ OWNED BY { table_name.column_name | NONE } ] ;
```



- 修改序列的拥有者：

```
ALTER SEQUENCE [ IF EXISTS ] name OWNER TO new_owner;
```



参数说明

- **name**

  将要修改的序列名称。

- **IF EXISTS**

  当序列不存在时使用该选项，不会显示ERROR，而是返回一个NOTICE信息。

- **MAXVALUE maxvalue | NO MAXVALUE | NOMAXVALUE**

  执行序列的最大值。新修改的最大值必须大于当前的last_value；如果没有指定，将保持旧的最大值。

  取值范围：(last_value, 2^{63}-1] ，使用LARGE标识时则为(last_value, 2^{127}-1]。

- **CACHE**

  为了快速访问，而在内存中预先存储序列号的个数。如果没有指定，将保持旧的缓冲值。

  取值范围：[1, 2^63-1]，使用LARGE标识时则为：[1, 2^{127}-1]。

- **OWNED BY**

  将序列和一个表的指定字段进行关联。这样，在删除指定字段或其所在表的时候会自动删除已关联的序列。

  如果序列已经和表有关联后，使用这个选项后新的关联关系会覆盖旧的关联。

  关联的表和序列的所有者必须是同一个用户，并且在同一个模式中。

  使用OWNED BY NONE将删除任何已经存在的关联。

- **new_owner**

  序列新所有者的用户名。用户要修改序列的所有者，必须是新角色的直接或者间接成员，并且该角色必须有序列所在模式上的CREATE权限，用户名要求详见**•user_name**。

示例

```
--创建一个名为serial的递增序列，从101开始。
m_db=# CREATE SEQUENCE serial START 101;

--查询serial的下个值
m_db=# SELECT pg_catalog.nextval('serial');
 nextval
---------
     101
(1 row)

--将序列serial的最大值改为101。
m_db=# ALTER SEQUENCE serial MAXVALUE 101;

--查询serial的下个值
m_db=# SELECT pg_catalog.nextval('serial');
ERROR:  nextval: reached maximum value of sequence "serial" (101)
CONTEXT:  referenced column: nextval

--删除序列。
m_db=# DROP SEQUENCE serial CASCADE;
```

相关链接

CREATE SEQUENCE，**DROP SEQUENCE**

4.4.2.6.10 ALTER SESSION

功能描述

ALTER SESSION命令用于定义或修改那些对当前会话有影响的条件或参数。修改后的会话参数会一直保持，直到断开当前会话。

注意事项

- 如果执行SET TRANSACTION之前没有执行START TRANSACTION，则事务立即结束，命令无法显示效果。
- 可以用START TRANSACTION里面声明所需要的transaction_mode(s)的方法来避免使用SET TRANSACTION。具体请参见：**START TRANSACTION**。

语法格式

- 设置会话的事务参数。

```
ALTER SESSION SET TRANSACTION
   { ISOLATION LEVEL { READ COMMITTED | READ UNCOMMITTED | REPEATABLE READ |
SERIALIZABLE} } | { READ ONLY  | READ WRITE } } [, ...] ;
```

- 设置会话的其他运行时参数。

```
ALTER SESSION SET
   {{config_parameter { { TO  | =  }  { value | DEFAULT } | FROM CURRENT }}
     | TIME ZONE time_zone
     | CURRENT_SCHEMA schema
     | SCHEMA 'schema'
     | NAMES encoding_name
     | NAMES encoding_name COLLATE collate_name
     | ROLE role_name PASSWORD 'password'
     | SESSION AUTHORIZATION { role_name PASSWORD 'password' | DEFAULT }
   } ;
```

参数说明

- **config_parameter**

  可设置的运行时参数的名称。可用的运行时参数可以使用SHOW ALL命令查看。

  - **value**

    config_parameter的新值。可以声明为字符串常量、标识符、数字，或者逗号分隔的列表。DEFAULT用于把这些参数设置为它们的缺省值。

    - DEFAULT
    - OFF
    - RESET
    - 用户指定的值：需要满足修改参数的取值限制

  - **FROM CURRENT**

    取当前会话中的值设置为configuration_parameter的值。

- **TIME ZONE timezone**

  用于指定当前会话的本地时区。

  取值范围：有效的本地时区。该选项对应的运行时参数名称为TimeZone，DEFAULT缺省值为PRC。

- **CURRENT_SCHEMA schema**

  CURRENT_SCHEMA用于指定当前的模式。

  取值范围：已存在模式名称。如果模式名不存在，会导致CURRENT_SCHEMA值为空。

- **SCHEMA 'schema'**

  同CURRENT_SCHEMA。此处的schema是个字符串。

- **NAMES encoding_name**

  用于设置客户端的字符编码。等价于set client_encoding to encoding_name。

  取值范围：有效的字符编码。该选项对应的运行时参数名称为client_encoding，默认编码为UTF8。

- **ROLE role_name**

  role_name取值范围：已存在的角色名，角色名要求详见**•role_name**。

- **PASSWORD 'password'**

  password为角色的密码。要求符合密码的命名规则。

- **SESSION AUTHORIZATION**

  当前会话的用户表示符。

- **ISOLATION LEVEL**

  指定事务隔离级别，该参数决定当一个事务中存在其他并发运行事务时能够看到什么数据。

  取值范围：

  - READ COMMITTED：读已提交隔离级别，只能读到已经提交的数据，而不会读到未提交的数据。这是缺省值。
  - READ UNCOMMITTED：读未提交隔离级别，指定后的行为和READ COMMITTED行为一致。

示例

```
-- 创建模式ds。
m_db=# CREATE SCHEMA ds;

--设置模式搜索路径。
m_db=# SET SEARCH_PATH TO ds, public;

--设置日期时间风格为传统的POSTGRES风格（日在月前）。
m_db=# SET DATESTYLE TO postgres, dmy;

--设置当前会话的字符编码为UTF8。
m_db=# ALTER SESSION SET NAMES 'UTF8';

--设置时区为加州伯克利。
m_db=# SET TIME ZONE 'PST8PDT';

--设置时区为意大利。
m_db=# SET TIME ZONE 'Europe/Rome';

--设置当前模式。
m_db=# ALTER SESSION SET CURRENT_SCHEMA TO ds;
--删除ds模式。
m_db=# DROP SCHEMA ds;
--开启事务,设置事务级别。
m_db=# START TRANSACTION;
m_db=# ALTER SESSION SET TRANSACTION READ ONLY;
m_db=# ROLLBACK;
```

相关链接

SET

4.4.2.6.11 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、设置表的所属模式、添加/更新多个列。

注意事项

- 表的所有者、被授予了表ALTER权限的用户或被授予ALTER ANY TABLE权限的用户有权限执行ALTER TABLE命令，系统管理员默认拥有此权限。但要修改表的所

有者或者修改表的模式，当前用户必须是该表的所有者或者系统管理员，且该用户是新所有者角色的成员。

- 不能修改分区表的TABLESPACE，但可以修改分区的TABLESPACE。
- 不支持修改存储参数ORIENTATION。
- SET SCHEMA操作不支持修改为系统内部模式，当前仅支持用户模式之间的修改。
- 不支持增加DEFAULT值中包含nextval()表达式的列。
- 通过约束名删除PRIMARY KEY约束时，不会删除NOT NULL约束，如果有需要，请手动删除NOT NULL约束。
- 使用JDBC时，支持通过PrepareStatement对DEFAULT值进行参数化设置。
- 如果用ADD COLUMN增加一个字段，那么所有表中现有行都初始化为该字段的缺省值（如果没有声明DEFAULT子句，默认值为NULL）。
  - 新增列没有声明DEFAULT值时，默认值为NULL，不会触发全表更新。
  - 新增列如果有DEFAULT值，必须符合以下所有要求，否则会带来全表更新的操作，影响在线业务：
    - 数据类型必须为以下类型：TINYINT、SMALLINT、BIGINT、INTEGER、NUMERIC、DECIMAL、BOOL、FLOAT、DOUBLE、CHAR、VARCHAR、TEXT、TIMESTAMP、DATE、TIME。
    - 新增列的DEFAULT值长度不超过128个字节。
    - 新增列DEFAULT值不包含易变（volatile）函数。
    - 新增列设置有DEFAULT值，且DEFAULT值不为NULL。
    - 如果不确定是否满足新增列DEFAULT值不包含易变（volatile）函数的条件，可以查询pg_rpoc系统表中函数的provolatile属性是否为'v'进行判断。
- 使用FIRST | AFTER column_name新增列或修改列，或修改字段的字符集，会带来全表更新的操作，影响在线业务。

语法格式

- 修改表的定义。

```
ALTER TABLE { 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
    | ADD [CONSTRAINT] table_constraint
    | DROP CONSTRAINT [ IF EXISTS ]  constraint_name [ RESTRICT | CASCADE ]
    | SET ( {storage_parameter = value} [, ... ] )
    | RESET ( storage_parameter [, ... ] )
    | OWNER TO new_owner
    | TO { GROUP groupname | NODE ( nodename [, ... ] ) }
    | ADD NODE ( nodename [, ... ] )
    | DELETE NODE ( nodename [, ... ] )
    | AUTO_INCREMENT [ = ] value
    | COMMENT [ = ] 'string'
    | [ [ DEFAULT ] {CHARACTER SET | CHAR SET | CHARSET} [ = ] charset_name] [ [ DEFAULT ]
COLLATE [ = ] collation_name]
    | CONVERT TO {CHARACTER SET | CHAR SET | CHARSET} charset_name [COLLATE collation_name ]
    | ADD index_clause
    | DROP {INDEX | KEY} index_name
    | DROP PRIMARY KEY
```



```
  | DROP FOREIGN KEY fk_name
  | REPLICA IDENTITY { DEFAULT | USING INDEX index_name | FULL | NOTHING }
```

说明

- **ADD [CONSTRAINT] table_constraint**
  给表增加一个新的约束。
- **DROP CONSTRAINT [IF EXISTS] constraint_name [RESTRICT | CASCADE]**
  删除一个表上的约束。
- **SET ({storage_parameter = value} [, ...])**
  修改表的一个或多个存储参数。
- **RESET (storage_parameter [, ...])**
  重置表的一个或多个存储参数。
- **OWNER TO new_owner**
  将表、序列、视图的所有者改变成指定的用户，用户名要求详见•user_name。
- **TO { GROUP groupname | NODE (nodename [, ...]) }**
  此语法仅在扩展模式（GUC参数support_extended_features为on时）下可用。该模式谨慎打开，主要供内部扩容工具使用，一般用户不应使用该模式。
- **ADD NODE (nodename [, ...])**
  此语法主要供内部扩容工具使用，一般用户不建议使用。
- **DELETE NODE (nodename [, ...])**
  此语法主要供内部缩容工具使用，一般用户不建议使用。
- **AUTO_INCREMENT [=] value**
  设置自动增长列下一次的自增值。设置的值只有大于当前自增计数器时才会生效。
  value必须是非负数，且不得大于2^{127}-1。
- **[[DEFAULT] {CHARACTER SET | CHAR SET | CHARSET} [=] default_charset] [[DEFAULT] COLLATE [=] default_collation]**
  将表的默认字符集和默认字符序修改为指定的值。修改不会影响表中当前已经存在的列。
- **CONVERT TO {CHARACTER SET | CHAR SET | CHARSET} charset [COLLATE collation]**
  将表的默认字符集和默认字符序修改为指定的值，同时将表中的所有字符类型的字段的字符集和字符序设置为指定的值，并将字段里的数据转换为新字符集编码。
  该语法不支持分区表修改字符序。
- **DROP {INDEX | KEY} index_name**
  删除表上的索引。INDEX和KEY为同义词。
- **DROP PRIMARY KEY**
  删除表上的主键约束。
- **DROP FOREIGN KEY fk_name**
  删除指定的外键约束。
- **REPLICA IDENTITY { DEFAULT | USING INDEX index_name | FULL | NOTHING }**
  在逻辑复制场景下，指定该表的UPDATE和DELETE操作中旧元组的记录级别。
  - DEFAULT记录主键的列的旧值，没有主键则不记录。
  - USING INDEX记录命名索引覆盖的列的旧值，这些值必须是唯一的、不局部的、不可延迟的，并且仅包括标记为NOT NULL的列。
  - FULL记录该行中所有列的旧值。
  - NOTHING不记录有关旧行的信息。

  在逻辑复制场景，解析该表的UPDATE和DELETE操作语句时，解析出的旧元组由以此方法记录的信息组成。对于有主键表该选项可设置为DEFAULT或FULL。对于无主键表该选项需设置为FULL，否则解码时旧元组将解析为空。一般场景不建议设置为NOTHING，旧元组会始终解析为空。

针对ustore表，选项NOTHING无效，实际效果等同于FULL；DEFAULT没有主键时，记录该行所有列。

- 其中列相关的操作column_clause可以是以下子句之一：

```
ADD [ COLUMN ] column_name data_type [ {CHARACTER SET | CHAR SET | CHARSET}
charset ][ COLLATE collation ] [ column_constraint [ ... ] ] [ FIRST | AFTER column_name ]
| ADD ({ column_name data_type [ compress_mode ]} [, ...] )
| MODIFY column_name [ CONSTRAINT constraint_name ] NOT NULL [ ENABLE ]
| MODIFY column_name [ CONSTRAINT constraint_name ] NULL
| MODIFY [ COLUMN ] column_name data_type [ {CHARACTER SET | CHAR SET | CHARSET}
charset ] [{[ COLLATE collation ] | [ column_constraint ]} [ ... ] ] [ FIRST | AFTER column_name ]
| CHANGE [ COLUMN ] old_column_name new_column_name data_type [ {CHARACTER SET |
CHAR SET | CHARSET}  charset ] [{[ COLLATE collation ] | [ column_constraint ]} [ ... ] ] [ FIRST
| AFTER column_name ]
| DROP [ COLUMN ]  column_name [ RESTRICT | CASCADE ]
| ALTER [ COLUMN ] column_name { SET DEFAULT default_expr | DROP DEFAULT }
| MODIFY column_name data_type [GENERATED ALWAYS] AS generation_expr [STORED]
```


说明

- **ADD [COLUMN] column_name data_type [{CHARACTER SET | CHAR SET | CHARSET} charset] [COLLATE collation] [column_constraint [...]] [FIRST | AFTER column_name]**

  向表中增加一个新的字段。用ADD COLUMN增加一个字段，所有表中现有行都初始化为该字段的缺省值（如果没有声明DEFAULT子句，值为NULL）。其中FIRST | AFTER column_name表示新增字段到某个位置。

- **ADD ({ column_name data_type [compress_mode] } [, ...])**

  向表中增加多列。

- **MODIFY ({ column_name data_type | column_name [CONSTRAINT constraint_name] NOT NULL [ENABLE] | column_name [CONSTRAINT constraint_name] NULL } [, ...])**

  修改表已存在字段的数据类型。此命令会导致该字段的统计信息清空，建议在修改后重新收集该列的统计信息。

- **MODIFY [COLUMN] column_name data_type [{CHARACTER SET | CHAR SET | CHARSET} charset] [{[COLLATE collation] | [column_constraint]} [...]] [FIRST | AFTER column_name]**
  - 修改表已存在字段的定义，将用新定义替换字段原定义，原字段上的索引、独立对象约束（例如：主键、唯一键、CHECK约束等）不会被删除。[FIRST | AFTER column_name]语法表示修改字段定义的同时修改字段在表中的位置。
  - 不支持修改分区键信息，即不支持指定column_name为分区键。
  - 被修改数据类型或排序规则的字段如果被一个生成列引用，这个生成列的数据将会重新生成。
  - 被修改字段若被一些对象依赖（比如：索引、独立对象约束、视图等），修改字段过程中将会重建这些对象。若被修改后字段定义违反此类对象的约束，修改操作会失败，比如：修改作为视图结果列的字段的数据类型。请修改字段前评估这类影响。
  - 被修改字段若被一些对象调用，修改字段不会处理这些对象。修改字段完毕后，这些对象有可能出现不可用的情况，请修改字段前评估这类影响。
  - 修改字段的字符集或字符序会将字段中的数据转换为新的字符集进行编码。
  - 此命令会导致该字段的统计信息清空，建议在修改后重新收集该列的统计信息。
- **CHANGE [COLUMN] old_column_name new_column_name data_type [{CHARACTER SET | CHAR SET | CHARSET} charset] [{[COLLATE collation] | [column_constraint]} [...]] [FIRST | AFTER column_name]**
  - 修改表已存在字段的名称和定义，字段新名称不能是已有字段的名称，将用新名称和定义替换字段原名称和定义。原字段上的索引、独立对象约束（例如：主键、唯一键、CHECK约束）等不会被删除。[FIRST | AFTER column_name]语法表示修改字段名称和定义的同时修改字段在表中的位置。
  - 不支持修改分区键字段的数据类型和排序规则，不支持修改规则引用的字段的数据类型和排序规则。
  - 被修改数据类型或排序规则的字段如果被一个生成列引用，这个生成列的数据将会重新生成。
  - 被修改字段若被一些对象依赖（比如：索引、独立对象约束、视图等），修改字段过程中将会重建这些对象。若被修改后字段定义违反此类对象的约束，修改操作会失败，比如：修改作为视图结果列的字段的数据类型。请修改字段前评估这类影响。
  - 被修改字段若被一些对象调用，修改字段不会处理这些对象。修改字段名称后，这些对象有可能出现不可用的情况，请修改字段前评估这类影响。
  - 修改字段的字符集或字符序会将字段中的数据转换为新的字符集进行编码。

- **DROP [COLUMN] column_name [RESTRICT | CASCADE]**
  - 从表中删除一个字段，和这个字段相关的索引和表约束也会被自动删除。CASCADE选项在数据库M-compatibility模式兼容版本控制开关s1及以上版本（如m_format_dev_version = 's1'）时仅语法支持，但实际不生效。
  - DROP COLUMN命令并不是物理上把字段删除，而只是简单地把它标记为对SQL操作不可见。随后对该表的插入和更新将在该字段存储一个NULL。因此，删除一个字段是很快的，但是它不会立即释放表在磁盘上的空间，因为被删除了的字段占据的空间还没有回收。这些空间将在执行VACUUM时而得到回收。
- **ALTER [COLUMN] column_name { SET DEFAULT default_expr | DROP DEFAULT }**

  为一个字段设置或者删除缺省值。请注意缺省值只应用于随后的INSERT命令，它们不会修改表中已经存在的行。也可以为视图创建缺省，这个时候它们是在视图的ON INSERT规则应用之前插入到INSERT句中的。

  此处的default_expr支持范围请参考**•DEFAULT default_expr**
- **MODIFY column_name data_type [GENERATED ALWAYS] AS generation_expr [STORED]**

  修改生成列的数据类型和表达式。详见**[GENERATED ALWAYS] AS (generation_expr) [STORED]**。

  - 其中列约束column_constraint为：

```
AUTO_INCREMENT
| COMMENT 'string'
|[ CONSTRAINT constraint_name ]
   { NOT NULL |
     NULL |
     CHECK ( expression ) |
     DEFAULT default_expr  |
     ON UPDATE update_expr |
     [GENERATED ALWAYS] AS ( generation_expr ) [STORED] |
     UNIQUE [KEY] index_parameters |
     [PRIMARY] KEY index_parameters |
     REFERENCES reftable [ ( refcolumn ) ] [ MATCH FULL  | MATCH SIMPLE ]
        [ ON DELETE action ] [ ON UPDATE action ] }
```

  - 其中列的压缩可选项compress_mode为：

```
[ DICTIONARY ]
```

  - 其中update_expr为：

```
{ CURRENT_TIMESTAMP | LOCALTIMESTAMP | NOW() }
```

- index_clause

```
{INDEX | KEY} [ [schema_name.] index_name ] [ USING method ] ({column_name [(length)] | (expr) [ASC | DESC]}[,...]) [[COMMENT 'string' | USING method][...]]
```

  参数说明，详见**CREATE INDEX**语法。

- 其中表约束table_constraint为：

```
[ CONSTRAINT [ constraint_name ] ]
   { CHECK ( expression ) |
     UNIQUE [INDEX|KEY][index_name] [USING access_method] ( { { column_name
[ ( length ) ] | ( expression ) } [ ASC | DESC ] } [, ... ] ) index_parameters [USING
access_method] [comment 'string']|
     PRIMARY KEY [ USING access_method ] [index_name] ( { column_name [ ASC | DESC ] }
[, ... ] ) index_parameters [USING access_method] [comment 'string']
     }
     FOREIGN KEY [ idx_name ] ( column_name [, ... ] ) REFERENCES reftable [ ( refcolumn
[, ... ] ) ]
       [ MATCH FULL  | MATCH SIMPLE ] [ ON DELETE action ] [ ON UPDATE action ] }
```

  access_method可取值为：

```
BTREE
```

  其中索引参数index_parameters为：


```
    [ WITH ( {storage_parameter = value} [, ... ] ) ]
```

- 重命名表。对名称的修改不会影响所存储的数据。

```
ALTER TABLE table_name
    RENAME [TO | AS | =] new_table_name;
```

- 重命名表上的索引。

```
ALTER TABLE table_name
    RENAME {INDEX | KEY} old_index_name TO new_index_name
```

- 设置表的所属模式。

```
ALTER TABLE [ IF EXISTS ] table_name
    SET SCHEMA new_schema;
```

说明

- 这种形式把表移动到另外一个模式。相关的索引、约束都跟着移动。目前序列不支持改变schema。 若该表拥有序列，需要将序列删除，重建，或者取消拥有关系， 才能将表schema更改成功。
- 要修改一个表的模式，用户必须在新模式上拥有CREATE权限。要把该表添加为一个父表的新子表，用户必须同时又是父表的所有者。要修改所有者，用户还必须是新的所有角色的直接或间接成员，并且该成员必须在此表的模式上有CREATE权限。这些限制规定了该用户不能做出了重建和删除表之外的事情。不过，系统管理员可以以任何方式修改任意表的所有权限。
- 除了RENAME和SET SCHEMA之外所有动作都可以捆绑在一个经过多次修改的列表中并行使用。比如，可以在一个命令里增加几个字段或修改几个字段的类型。对于大表，此种操作带来的效率提升更明显，原因在于只需要对该大表做一次处理。
- 增加一个CHECK或NOT NULL约束将会扫描该表，以保证现有的行符合约束要求。
- 用一个非空缺省值增加一个字段或者改变一个字段的现有类型会重写整个表。对于大表来说，这个操作可能会花很长时间，并且它还临时需要两倍的磁盘空间。

- 添加多个列。

```
ALTER TABLE table_name
    ADD ( { column_name data_type [ compress_mode ] [ COLLATE collation ] [ column_constraint
[ ... ] ]} [, ...] );
```

参数说明

- **IF EXISTS**

  如果不存在相同名称的表，不会抛出一个错误，而会发出一个通知，告知表不存在。

- **table_name [*] | ONLY table_name | ONLY (table_name)**

  table_name是需要修改的表名。

  若声明了ONLY选项，则只有该表被更改。若未声明ONLY，该表及其所有子表都将会被更改。另外，可以在表名称后面显示地增加*选项来指定包括子表，即表示所有后代表都被扫描，这是默认行为。目前ONLY和增加*选项保留语法，但功能不支持。

- **constraint_name**
  - 在DROP CONSTRAINT操作中表示要删除的现有约束的名称。
  - 在ADD CONSTRAINT操作中表示新增的约束名称。

- **index_name**

  索引名称。

须知

在ADD CONSTRAINT操作中：

- 对于外键约束，constraint_name和index_name同时指定时，索引名为constraint_name。
- 对于唯一键约束，constraint_name和index_name同时指定时，索引名为index_name。

- **USING method**

  指定创建索引的方法。

  取值范围参考**参数说明**中的USING method。

须知

在ADD CONSTRAINT操作中：

- 未指定USING method时，对于ASTORE的存储方式，默认索引方法为btree；对于USTORE的存储方式，默认索引方法为ubtree。
- 当表的存储方式为USTORE时，SQL语句中约束指定为using btree，底层会自动将约束建立为using ubtree。

- **ASC | DESC**

  ASC表示指定按升序排序（默认）。DESC指定按降序排序。

- **expression**

  创建一个基于该表的一个或多个字段的表达式索引约束，必须写在圆括弧中。

- **storage_parameter**

  存储参数的名称。

  支持使用ALTER TABLE SET/RESET语法修改的存储参数如下：

  - parallel_workers

    表示创建索引时起的bgworker线程数量，例如2就表示将会起2个bgworker线程并发创建索引。

    取值类型：int类型

    取值范围：[0,32]，0表示关闭并行建索引。

    默认值：不设置该参数，表示未开启并行建索引功能。

- **new_owner**

  表新拥有者的名称，用户名要求详见**•user_name**。

- **column_name，column_1_name， column_2_name**

  现存的或新字段的名称。

- **data_type**

  新字段的类型，或者现存字段的新类型。

- **compress_mode**

  表字段的压缩可选项。该子句指定该字段优先使用的压缩算法。行存表不支持压缩。当前M-Compatibility下无法使用。

- **charset**

  指定表字段的字符集。单独指定时会将字段的字符序设置为指定的字符集的默认字符序。

- **collation**

  字段排序规则（字符序）名称。可选字段COLLATE指定了新字段的排序规则，如果省略，排序规则为新字段的默认类型。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。

  还支持utf8mb4_bin、utf8mb4_general_ci、utf8mb4_unicode_ci、binary、gbk_chinese_ci、gbk_bin、gb18030_chinese_ci、gb18030_bin字符序，请参见**表级字符集和字符序**。

  说明

  - 仅字符类型支持指定字符集，指定为BINARY字符集或字符序实际是将字符类型转化为对应的二进制类型，若类型映射不存在则报错。当前仅有TEXT类型转化为BLOB的映射。
  - 除BINARY字符集、字符序外，当前仅支持指定与数据库编码相同的字符集。
  - 字段字符集或字符序未显式指定时，若指定了表的默认字符集或字符序，字段字符集和字符序将从表上继承。
  - 当修改的字符集或字符序对应的字符集与当前字段字符集不同时，会将字段中的数据转换为指定的字符集进行编码。

- **NOT NULL | NULL**

  设置列是否允许空值。

- **ENABLE**

  表示启动该约束，缺省时默认启用。

- **CHECK (expression)**

  每次将要插入的新行或者将要被更新的行必须使表达式结果为真才能成功，否则会抛出一个异常并且不会修改数据库。

  目前，CHECK表达式不能包含子查询。

- **DEFAULT default_expr**
  - 使用DEFAULT子句给字段指定缺省表达式，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 。
  - 当未在缺省表达式外嵌套括号时，支持指定以下内容：常量、带正负号的数值常量、**update_expr**。
  - 当在缺省表达式外嵌套括号时，支持指定以下内容：常量、带正负号的数值常量、**update_expr**、CURRENT_TIME/CURTIME函数、CURRENT_DATE/CURDATE函数（CURRENT_TIME/CURRENT_DATE支持不带括号形式的调用）。
  - 仅支持在TIMESTAMP、DATETIME类型的字段上指定update_expr作为默认值，且字段的精度与update_expr的精度须保持一致。

- **ON UPDATE update_expr**

  ON UPDATE子句为字段的一种属性约束。

  当对表中某元组执行UPDATE操作时，若更新字段的新值和表中旧值不相同，则表中该元组上具有该属性且不在更新字段内的字段值自动更新为当前时间戳；若更新字段的新值和表中旧值相同，则表中该元组上具有该属性且不在更新字段内的字段值不变，保持原有值；若具有该属性的字段在更新字段内，则对应这些字段值直接按指定更新的值更新。

说明

- 语法上update_expr支持CURRENT_TIMESTAMP 、LOCALTIMESTAMP 、LOCALTIME、NOW()四种关键字，也支持关键字带括号指定或不指定精度。例如：ON UPDATE CURRENT_TIMESTAMP()、ON UPDATE CURRENT_TIMESTAMP(5)、ON UPDATE LOCALTIMESTAMP()、ON UPDATE LOCALTIMESTAMP(6)等。不带括号或空括号时精度为0，其中NOW关键字不支持不带括号。四种关键字互为同义词，属性效果相同。
- 该属性支持在如下类型的列上指定: TIMESTAMP、DATETIME。
- 该属性指定的精度和对应列上类型指定的精度必须一致，否则会触发报错。例如：CREATE TABLE t1 (col1 timestamp(6) ON UPDATE CURRENT_TIMESTAMP(6)); 若精度不一致，会产生ERROR: Invalid ON UPDATE clause for "col1" 报错。
- 该属性和生成列约束不能同时指定同一列。
- 分区表中的分区键不支持指定该属性。

- **[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

  - STORED关键字可省略，与不省略STORED语义相同。
  - 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。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约束子句的CASCADE,SET NULL,SET DEFAULT动作同时指定。生成列不能和ON DELETE约束子句的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。

- **AUTO_INCREMENT**

  指定列为自动增长列。

  详见：**•AUTO_INCREMENT**。

- **COMMENT [=] 'string'**
  - COMMENT [=] 'string'子句表示给表添加注释。
  - 在column_constraint中的COMMENT 'string'表示给列添加注释。
  - 在table_constraint中的COMMENT 'string'表示给主键和唯一键对应的索引添加注释。

  具体请参见：**•COMMENT [=] 'string'**

- **列级唯一约束：UNIQUE [KEY] index_parameters**

  UNIQUE KEY与UNIQUE语义相同。

  UNIQUE约束表示表里的一个或多个字段的组合必须在全表范围内唯一。

- **表级唯一约束：UNIQUE [INDEX | KEY] [index_name][USING method] ({{ column_name [(length)] | (expression) } [ASC | DESC] }[, ...]) index_parameters**

UNIQUE约束表示表里的一个或多个字段的组合必须在全表范围内唯一。

column_name (length)是前缀键，详见：**•column_name (length)**。

index_name为索引名。

须知

对于唯一键约束，constraint_name和index_name同时指定时，索引名为index_name。

- **列级主键约束：[PRIMARY] KEY index_parameters**

  表级主键约束：PRIMARY KEY [index_name] [USING method] ({ column_name [ASC | DESC] } [, ...] ） index_parameters

  主键约束表明表中的一个或者一些字段只能包含唯一（不重复）的非NULL值。

- **列级外键约束：REFERENCES reftable [(refcolumn)] [MATCH matchtype] [ON DELETE action] [ON UPDATE action]**

  表级外键约束：FOREIGN KEY (column_name [, ...]) REFERENCES reftable [(refcolumn [, ...])] [MATCH matchtype] [ON DELETE action] [ON UPDATE action]

  表列字段约束REFERENCES使用时语法不报错也没有提示，但实际不生效。

  外键约束要求新表中一列或多列构成的组应该只包含、匹配被参考表中被参考字段值。若省略refcolumn，则将使用reftable的主键。被参考列应该是被参考表中的唯一字段或主键。外键约束不能被定义在临时表和永久表之间。

  参考字段与被参考字段之间存在三种类型匹配，分别是：

  - MATCH FULL：不允许一个多字段外键的字段为NULL，除非全部外键字段都是NULL。
  - MATCH SIMPLE（缺省）：允许任意外键字段为NULL。

  另外，当被参考表中的数据发生改变时，某些操作也会在新表对应字段的数据上执行。ON DELETE子句声明当被参考表中的被参考行被删除时要执行的操作。ON UPDATE子句声明当被参考表中的被参考字段数据更新时要执行的操作。对于ON DELETE子句、ON UPDATE子句的可能动作：

  - NO ACTION（缺省）：删除或更新时，创建一个表明违反外键约束的错误。
  - RESTRICT：删除或更新时，创建一个表明违反外键约束的错误。与NO ACTION相同。
  - CASCADE：删除新表中任何引用了被删除行的行，或更新新表中引用行的字段值为被参考字段的新值。
  - SET NULL：设置引用字段为NULL。
  - SET DEFAULT：设置引用字段为它们的缺省值。

  说明

  - 外键约束的完整性检查由GUC参数foreign_key_checks进行控制，foreign_key_checks可以设置为on/off，分别表示启用/关闭外键约束的完整性检查，默认情况下为on。

- **WITH ({storage_parameter = value} [, ...])**

  索引指定一个可选的存储参数。

- **new_table_name**

  修改后新的表名称。

- **new_column_name**
  表中指定列修改后新的列名称。
- **new_index_name**
  修改后表上索引的新名称。
- **new_schema**
  修改后新的模式名称。
- **CASCADE | RESTRICT**
  CASCADE： 级联删除依赖于被依赖字段或者约束的对象（比如引用该字段的视图）。
  RESTRICT：如果字段或者约束还有任何依赖的对象，则拒绝删除该字段。这是缺省行为。

  说明

  该属性在数据库M-compatibility模式兼容版本控制开关s1及以上版本（如m_format_dev_version = 's1'）时仅语法支持，但实际不生效。
- **FIRST**
  新增列或修改列到第一位。
- **AFTER column_name**
  新增列或修改列到column_name之后。

  说明

  有规则依赖的表不支持改变表列的位置（包括新增和修改导致列位置的变化）。
- **schema_name**
  表所在的模式名称。

修改表示例

- **重命名表**
```
m_db=# CREATE TABLE aa(c1 int, c2 int);
m_db=# ALTER TABLE aa RENAME TO test_alt1;
```
- **修改表所属模式**
```
--创建模式test_schema。
m_db=# CREATE SCHEMA test_schema;
--把表test_alt1的所属模式修改为test_schema。
m_db=# ALTER TABLE test_alt1 SET SCHEMA test_schema;
--查询表信息。
m_db=# SELECT schemaname,tablename FROM pg_tables WHERE tablename = 'test_alt1';
 schemaname  | tablename
-------------+-----------
 test_schema | test_alt1
(1 row)
```
- **修改表的所有者**
```
--创建用户test_user。
m_db=# CREATE USER test_user PASSWORD 'XXXXXXXXXX';
-- 修改test_alt1表的所有者为test_user。
m_db=# ALTER TABLE test_schema.test_alt1 OWNER TO test_user;
-- 查看。
m_db=# SELECT tablename, schemaname, tableowner FROM pg_tables WHERE tablename =
'test_alt1';
 tablename | schemaname  | tableowner
-----------+-------------+------------
 test_alt1 | test_schema | test_user
(1 row)
```


修改列示例

- **修改列名**

```
-- 建表。
m_db=# CREATE TABLE test_alt2(c1 INT,c2 INT);
-- 修改列名。
m_db=# ALTER TABLE test_alt2 CHANGE COLUMN c1 id INT;
m_db=# ALTER TABLE test_alt2 CHANGE COLUMN c2 areaid INT;
-- 查看。
m_db=# \d test_alt2
   Table "public.test_alt2"
 Column |  Type   | Modifiers
--------+---------+-----------
 id     | integer |
 areaid | integer |
```

- **增加列**

```
-- 表test_alt2增加列。
m_db=# ALTER TABLE test_alt2 ADD COLUMN name VARCHAR(20);
-- 查看。
m_db=# \d test_alt2
           Table "public.test_alt1"
  Column  |         Type          | Modifiers
----------+-----------------------+-----------
 id       | integer               |
 areacode | integer               |
 name     | varchar(20)           |
```

- **增加AUTO_INCREMENT自增列**

```
-- 建表，新增自增列。
m_db=# CREATE TABLE test_autoinc(col1 int);

-- 插入一条数据。
m_db=# INSERT INTO test_autoinc(col1) VALUES(1);

-- 添加一个本地自增列，从1开始自增。
m_db=# ALTER TABLE test_autoinc ADD COLUMN col int AUTO_INCREMENT;

m_db=# SELECT col,col1 FROM test_autoinc ORDER BY 2,1;
 col | col1
-----+------
   1 |    1
(1 row)

-- 将下一个自增值设为10。
m_db=# ALTER TABLE test_autoinc AUTO_INCREMENT = 10;

-- NULL触发自增，自增值为10。
m_db=# INSERT INTO test_autoinc(col, col1) VALUES(NULL,2);

-- 0触发自增，自增值为11。
m_db=# INSERT INTO test_autoinc(col, col1) VALUES(0,3);

m_db=# SELECT col,col1 FROM test_autoinc ORDER BY 2,1;
 col | col1
-----+------
   1 |    1
  10 |    2
  11 |    3
(3 rows)
```

- **修改列的数据类型**

```
-- 修改test_alt2表中name字段的类型。
m_db=# ALTER TABLE test_alt2 MODIFY name VARCHAR(50);
-- 查看。
m_db=# \d test_alt2
          Table "public.test_alt1"
 Column |         Type          | Modifiers
--------+-----------------------+-----------
 c1     | integer               |
```



```
 c2     | integer               |
 name   | varchar(50)           |
```

- **删除列**

```
-- 删除test_alt2中areaid字段。
m_db=# ALTER TABLE test_alt2 DROP COLUMN areaid;
-- 查看
m_db=# \d test_alt2
          Table "public.test_alt2"
 Column |          Type          | Modifiers
--------+------------------------+-----------
 id     | integer                |
 name   | varchar(50)            |
```

修改约束示例

- **修改字段默认值**

```
--建表。
m_db=# CREATE TABLE test_alt3(pid INT, areaid CHAR(5), name VARCHAR(20));
--修改test_alt1表中id的默认值。
m_db=# ALTER TABLE test_alt3 ALTER COLUMN areaid SET DEFAULT '00000';
--查看。
m_db=# \d test_alt3
                Table "public.test_alt3"
 Column |          Type          |         Modifiers
--------+------------------------+------------------------
 pid    | integer                |
 areaid | char(5)                | default '00000'::bpchar
 name   | varchar(20)            |
--删除id的默认值。
m_db=# ALTER TABLE test_alt3 ALTER COLUMN areaid DROP DEFAULT;
--查看。
m_db=# \d test_alt3
          Table "public.test_alt3"
 Column |          Type          | Modifiers
--------+------------------------+-----------
 pid    | integer                |
 areaid | char(5)                |
 name   | varchar(20)            |
```

- **添加表级约束**

  直接添加约束

```
--给表添加主键约束。
m_db=# ALTER TABLE test_alt3 ADD CONSTRAINT pk_test3_pid PRIMARY KEY (pid);
--查看。
m_db=# \d test_alt3
          Table "public.test_alt3"
 Column |          Type          | Modifiers
--------+------------------------+-----------
 pid    | integer                | not null
 areaid | integer                |
 name   | varchar(20)            |
Indexes:
    "pk_test3_pid" PRIMARY KEY, btree (pid) TABLESPACE pg_default
```

修改索引示例

删除表中索引

```
-- 创建表和索引
m_db=# CREATE TABLE t01(c1 int PRIMARY KEY, c2 int UNIQUE);
m_db=# CREATE INDEX idx1 ON t01(c1);
m_db=# CREATE INDEX idx2 ON t01(c1,c2);
m_db=# \d+ t01
                    Table "public.t01"
 Column |  Type   | Modifiers | Storage | Stats target | Description
--------+---------+-----------+---------+--------------+-------------
 c1     | integer | not null  | plain   |              |
```



```
 c2     | integer |           | plain   |              |
Indexes:
    "t01_pkey" PRIMARY KEY, ubtree ON t01 (c1) WITH (storage_type=USTORE) TABLESPACE pg_default
    "t01_c2_key" UNIQUE CONSTRAINT, ubtree ON t01 (c2) WITH (storage_type=USTORE) TABLESPACE
pg_default
    "idx1" ubtree ON t01 (c1) WITH (storage_type=USTORE) TABLESPACE pg_default
    "idx2" ubtree ON t01 (c1, c2) WITH (storage_type=USTORE) TABLESPACE pg_default
Has OIDs: no
Options: orientation=row, compression=no, storage_type=USTORE, collate=1537, segment=off

-- 删除表中索引
m_db=# ALTER TABLE t01 DROP INDEX idx1, DROP INDEX idx2;
m_db=# ALTER TABLE t01 DROP INDEX t01_pkey, DROP INDEX t01_c2_key;
m_db=# \d+ t01
                      Table "public.t01"
 Column |  Type   | Modifiers | Storage | Stats target | Description
--------+---------+-----------+---------+--------------+-------------
 c1     | integer | not null  | plain   |              |
 c2     | integer |           | plain   |              |
Has OIDs: no
Options: orientation=row, compression=no, storage_type=USTORE, collate=1537, segment=off

m_db=# DROP TABLE t01;
```

相关链接

CREATE TABLE，**DROP TABLE**

4.4.2.6.12 ALTER TABLE PARTITION

功能描述

修改表分区，包括增加/删除分区、切割/合并分区、清空分区、移动分区表空间、交换分区、重命名分区、收集分区、开启/关闭分区自动扩展功能，以及修改分区属性等。

注意事项

- 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，当三权分立开关关闭时，系统管理员默认拥有此权限。
- 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。
- 不支持添加哈希分区。
- 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。
- 哈希分区表不支持切割分区，不支持合并分区，不支持添加/删除分区。
- 删除、切割、合并、清空、交换分区的操作会使Global索引失效，可以申明UPDATE GLOBAL INDEX子句同步更新索引。
- 如果删除、切割、合并、清空、交换分区操作不申明UPDATE GLOBAL INDEX子句，并发的DML业务有可能因为索引不可用而报错。

- 若设置参数enable_gpi_auto_update为on，即使不申明UPDATE GLOBAL INDEX子句，也会自动更新Global索引。

语法格式

修改分区表分区包括修改表分区主语法、修改表分区名称的语法。

- 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，M-Compatibility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 |
    merge_clause |
    split_clause |
    add_clause |
    drop_clause |
    truncate_clause |
    analyze_clause |
```

  - move_clause子语法用于移动分区到新的表空间。

```
MOVE PARTITION { partition_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 ]  [ 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。
    - 普通表索引和分区Local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。

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，可能会导致无法交换数据。

  如果需要进行数交换数据操作可以通过创建中间表的方式。先将分区数据插入到中间表，truncate分区，普通表数据插入分区表，drop普通表，重命名中间表的方式完成数据交换操作。
- 如果在普通表/分区表上进行了DROP COLUMN操作，被删除的列依然物理存在，则需要保证普通表和分区的被删除列严格对齐才能交换成功。
- EXCHANGE PARTITION partition_name：
  - 当partition_name为一级分区名时，进行交换的是一级分区和普通表；当partition_name为二级分区时， 进行交换的是二级分区和普通表。
  - 不支持在二级分区表中交换一级分区和普通表。
  - 进行分区交换后，自增列不会被重置。

- merge_clause子语法用于把多个分区合并成一个分区。一个命令中合并的源分区上限为300。

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

须知

- 对于范围分区，MERGE分区要求源分区的范围连续递增，且MERGE后的分区名可以与最后一个源分区名相同；对于列表分区，则源分区无顺序要求，且MERGE后的分区名可以与任一源分区名相同。如果MERGE后的分区名与源分区名相同，视为同一个分区。
- USTORE存储引擎表不支持在事务块/存储过程中执行ALTER TABLE MERGE PARTITIONS的操作。

- 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 ]
```

须知

- SPLIT后的分区名可以与源分区名相同，将视为不同的分区。
- 未打开guc参数enable_ilm的情况下，如果使用split_clause子语法把一个带有ilm policy的分区分割成多个分区，新分区不继承ilm policy。

  - 范围分区表指定切割点split_point_clause的语法为：

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


须知

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。

- 范围分区表不指定切割点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。
- 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）必须等于正在被切割的分区的前一个分区（如果存在）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）必须等于正在被切割的分区的分区键。
  no_split_point_clause
- partition_less_than_item支持的分区键个数最多为16，而partition_start_end_item仅支持1个分区键，其支持的数据类型请参见**PARTITION BY RANGE [COLU...**。
- 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的语法如下，其中最后一个分区可以不写分区范围定义，即VALUES LESS THAN (partition_value)部分，默认继承源分区范围定义的上界值。

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

须知

RANGE分区时支持MAXVALUE关键字不带括号，不支持在二级分区的子分区中使用，不支持在分区字段为多列的场景使用。

- 分区项partition_start_end_item的语法如下，其约束请参见**START END语法描述**。

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

- 列表分区表指定切割点split_point_clause的语法如下。

```
VALUES ( partition_value_list ) INTO ( PARTITION partition_name[ TABLESPACE tablespacename ] , PARTITION partition_name [ TABLESPACE tablespacename ] )
```


须知

切割点必须是源分区的一个非空真子集，指定切割点的方式只能把一个分区切割成两个新分区。

- 列表分区表不指定切割点no_split_point_clause的语法如下，其中最后一个分区不能写分区范围定义，即VALUES (partition_value_list)部分，其范围等于源分区去掉其他子分区后的剩余集合。

```
INTO ( PARTITION partition_name VALUES (partition_value_list) [ TABLESPACE tablespacename ][, ...] )
```

须知

- 最后一个新分区不能写分区范围定义，其范围等于源分区去掉其他子分区后的剩余集合。
- 不指定切割点的方式，每一个新分区都必须是源分区的一个非空真子集，且互不交叉。

– 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_less_than_item | partition_start_end_item | partition_list_item}[, ...])
```

分区项partition_list_item的语法为：

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

须知

- partition_list_item支持最多16个分区键，其支持的数据类型请参见**PARTITIONBYLIST[COLUMNS](partition_key)**。
- 间隔/哈希分区表不支持添加分区。
- IN不支持在二级分区的子分区中使用。

– drop_clause子语法用于删除分区表中的指定分区。

```
DROP PARTITION partition_name
```

须知

- 哈希分区表不支持删除分区。
- 当分区表只有一个分区时，不能删除该分区。

– truncate_clause子语法用于清空分区表中的指定分区。

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

– analyze_clause子语法用于收集分区表中的指定分区的统计信息。

```
ANALYZE PARTITION  { { ALL | partition_name [, ...] }
```

- 修改表分区名称的语法。

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


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一个表空间。

  取值范围：已存在的表空间名。

  注：需要在非M-Compatibility数据库中创建或删除tablespace。

- **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：需要进行操作的分区的分区键的取值范围。

- **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。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。

- **partition_new_name**

  分区的新名称。

  取值范围：字符串，要符合**标识符说明**。

- **{ ALL | partition_name [, ...] }**

  ALL：所有分区数据。

  partition_name ： 目标分区表的分区名，支持一级分区名和二级分区名。

  取值范围：已存在的分区名。

- **UPDATE GLOBAL INDEX**

  如果使用该参数，则会更新分区表上的所有全局索引，以确保使用全局索引可以查询出正确的数据。如果不使用该参数，则分区表上的所有全局索引将会失效。

示例

- **修改表分区名称**


```
--创建前置分区表。
m_db=# CREATE TABLE test_p1 (col1 INT, col2 INT) PARTITION BY RANGE (col1)
(
    PARTITION p1 VALUES LESS THAN (10),
    PARTITION p2 VALUES LESS THAN (20),
    PARTITION p3 VALUES LESS THAN (MAXVALUE)
);

--修改分区名称。
m_db=# ALTER TABLE test_p1 RENAME PARTITION p3 TO pmax;

--查询分区信息。
m_db=# SELECT relname, boundaries, oid FROM pg_partition WHERE parentid='test_p1'::regclass AND
parttype <> 'r';
 relname | boundaries |  oid
---------+------------+-------
 p1      | {10}       | 17066
 p2      | {20}       | 17067
 pmax    | {NULL}     | 17068
(3 rows)
```

- **分区交换**

```
--创建普通表，插入数据。
m_db=# CREATE TABLE test_ep1(col1 INT,col2 INT);
m_db=# INSERT INTO test_ep1 VALUES (GENERATE_SERIES(1,9), 1000);

--迁移普通表数据到指定分区。
m_db=# ALTER TABLE test_p1 EXCHANGE PARTITION p1 WITH TABLE test_ep1;

--查询。
m_db=# SELECT COUNT(*) FROM test_p1 PARTITION (p1);
 count
-------
     9
(1 row)

--删除表test_ep1
m_db=# DROP TABLE test_ep1;
```

- **分区合并**

```
--将test_p1表中p2,pmax分区合并到pmax中。
m_db=# ALTER TABLE test_p1 MERGE PARTITIONS p2,pmax INTO PARTITION pmax;

--查看分区。
m_db=# SELECT relname, boundaries, oid FROM pg_partition WHERE parentid='test_p1'::regclass AND
parttype <> 'r' order by 1;
 relname | boundaries |  oid
---------+------------+-------
 p1      | {10}       | 17066
 pmax    | {NULL}     | 17070
(2 rows)

--删除表和表空间。
m_db=# DROP TABLE test_p1;
```

- **切割分区**

```
--建表。
m_db=# CREATE TABLE test_r1 (col1 INT,col2 INT) PARTITION BY RANGE (col1)(
    PARTITION p1   VALUES LESS THAN (10),
    PARTITION pmax VALUES LESS THAN (MAXVALUE)
);

--切割分区。
m_db=# ALTER TABLE test_r1 SPLIT PARTITION pmax AT (20) INTO (PARTITION p2, PARTITION
pmax);
m_db=# ALTER TABLE test_r1 SPLIT PARTITION pmax INTO (
    PARTITION p3   VALUES LESS THAN (30),
    PARTITION pmax VALUES LESS THAN (MAXVALUE)
);
```



```
--查询。
m_db=# SELECT relname, boundaries, oid FROM pg_partition WHERE parentid='test_r1'::regclass AND
parttype <> 'r' order by 1;
 relname | boundaries |  oid
---------+------------+-------
 p1      | {10}       | 17088
 p2      | {20}       | 17090
 p3      | {30}       | 17092
 pmax    | {NULL}     | 17093
(4 rows)

--删除表test_r1。
m_db=# DROP TABLE test_r1;
--建表。
m_db=# CREATE TABLE test_r2(col1 INT, col2 INT) PARTITION BY RANGE (col1)(
    PARTITION p1   START(1) END(10),
    PARTITION p2   START(10) END(20),
    PARTITION pmax START(20) END(MAXVALUE)
);

--切割分区。
m_db=# ALTER TABLE test_r2 SPLIT PARTITION pmax INTO (
    PARTITION p3 START(20) END(30),
    PARTITION pmax START(30) END (MAXVALUE)
);

--查看。
m_db=# SELECT relname, boundaries, oid FROM pg_partition WHERE parentid='test_r2'::regclass AND
parttype <> 'r' order by 1;
 relname | boundaries |  oid
---------+------------+-------
 p1_0    | {1}        | 17112
 p1_1    | {10}       | 17113
 p2      | {20}       | 17114
 p3      | {30}       | 17116
 pmax    | {NULL}     | 17117
(5 rows)

--删除表test_r2。
m_db=# DROP TABLE test_r2;
--建表。
m_db=# CREATE TABLE test_l1(col1 INT, col2 INT) PARTITION BY LIST(col1)(
    PARTITION p1   VALUES (10,20),
    PARTITION p2 VALUES (30,40)
);

--切割分区。
m_db=# ALTER TABLE test_l1 SPLIT PARTITION p1 VALUES (10) INTO (PARTITION p1_1, PARTITION
p1_2);
m_db=# ALTER TABLE test_l1 SPLIT PARTITION p2 INTO (PARTITION p3_1 VALUES(30), PARTITION
p3_2);

--查看。
m_db=# SELECT relname, boundaries, oid FROM pg_partition WHERE parentid='test_l1'::regclass AND
parttype <> 'r' order by 1;
 relname | boundaries |  oid
---------+------------+-------
 p1_1    | {10}       | 17132
 p1_2    | {20}       | 17133
 p3_1    | {30}       | 17134
 p3_2    | {40}       | 17135
(4 rows)

--删除表test_l1。
m_db=# DROP TABLE test_l1;
```

- **添加分区**

```
--建表。
m_db=# CREATE TABLE test_p2 (col1 INT, col2 INT) PARTITION BY RANGE (col1)(
    PARTITION p1 VALUES LESS THAN (10),
```



```
    PARTITION p2 VALUES LESS THAN (20)
);

--添加分区。
m_db=# ALTER TABLE test_p2 ADD PARTITION (PARTITION p3 VALUES LESS THAN (30));

--删除表test_p2。
m_db=# DROP TABLE test_p2;
--建表。
m_db=# CREATE TABLE test_p3 (col1 INT, col2 INT) PARTITION BY LIST(col1)(
    PARTITION p1 VALUES (1),
    PARTITION p2 VALUES (2)
);

--添加分区。
m_db=# ALTER TABLE test_p3 ADD PARTITION (PARTITION p3 VALUES (3));

--删除表test_p3。
m_db=# DROP TABLE test_p3;

--进入M模式。
--建表
m_db=# CREATE TABLE addpart1
(
   month_code int,
   num        int,
   name       varchar(30)
)PARTITION BY RANGE COLUMNS(month_code)
(
   PARTITION p_202301 VALUES LESS THAN(202302),
   PARTITION p_202302 VALUES LESS THAN (202303)
);
--添加分区
m_db=# ALTER TABLE addpart1 ADD PARTITION (PARTITION p_202303 VALUES LESS THAN(202304));
--删除表
m_db=# DROP TABLE addpart1;
```

- **删除分区**

```
--建表。
m_db=# CREATE TABLE test_p4 (col1 INT, col2 INT) PARTITION BY LIST(col1)(PARTITION p1 VALUES
(1),PARTITION p2 VALUES (2));

--删除test_p3表的p2分区。
m_db=# ALTER TABLE test_p4 DROP PARTITION p2;

--查看。
m_db=# SELECT relname, boundaries, oid FROM pg_partition WHERE parentid='test_p4'::regclass;
 relname | boundaries |  oid
---------+------------+-------
 test_p4 |            | 17187
 p1      | {1}        | 17188
(2 rows)

--删除表test_p4。
m_db=# DROP TABLE test_p4;
```

- **清空分区**

```
--建表。
m_db=# CREATE TABLE test_p5 (col1 INT, col2 INT) PARTITION BY RANGE (col1)(
    PARTITION p1 VALUES LESS THAN (5),
    PARTITION p2 VALUES LESS THAN (10)
);

--插入数据。
m_db=# INSERT INTO test_p5 VALUES (GENERATE_SERIES(1,9), 100);

--查看p2分区数据。
m_db=# SELECT * FROM test_p5 PARTITION (p2);
 col1 | col2
------+------
```



```
   5 |  100
   6 |  100
   7 |  100
   8 |  100
   9 |  100
(5 rows)

--清空p2分区的数据。
m_db=# ALTER TABLE test_p5 TRUNCATE PARTITION p2;

--查看p2分区数据。
m_db=# SELECT * FROM test_p5 PARTITION (p2);
 col1 | col2
------+------
(0 rows)

--清空p1、p2分区数据。
m_db=# ALTER TABLE test_p5 TRUNCATE PARTITION p1,p2;

--清空所有分区数据。
m_db=# ALTER TABLE test_p5 TRUNCATE PARTITION ALL;

--删除表test_p5。
m_db=# DROP TABLE test_p5;
```

- 收集分区统计信息

```
--建立一级分区表。
m_db=# CREATE TABLE sales (
    id INT,
    region VARCHAR(100),
    amount DECIMAL(10,2),
    sale_date DATE
)
PARTITION BY RANGE(sale_date) (
    PARTITION p0 VALUES LESS THAN (20100101),
    PARTITION p1 VALUES LESS THAN (20110101),
    PARTITION p2 VALUES LESS THAN (20120101),
    PARTITION p3 VALUES LESS THAN MAXVALUE
);
--插入数据。
m_db=# INSERT INTO sales(sale_date) VALUES('2005-1-1'),('2010-1-1'),('2011-1-1'),('2012-1-1');
--收集分区统计信息。
m_db=# ALTER TABLE sales ANALYZE PARTITION p0;
m_db=# ALTER TABLE sales ANALYZE PARTITION p1,p2;
m_db=# ALTER TABLE sales ANALYZE PARTITION all;
m_db=# DROP TABLE sales;

--建立二级分区表。
CREATE TABLE range_hash1 (id INT, purchased INT)
PARTITION BY RANGE(purchased)
SUBPARTITION BY HASH(purchased)
(
    PARTITION p0 VALUES LESS THAN (19900101)
    (
        SUBPARTITION s0,
        SUBPARTITION s1
    ),
    PARTITION p1 VALUES LESS THAN (20000101)
    (
        SUBPARTITION s2,
        SUBPARTITION s3
    ),
    PARTITION p2 VALUES LESS THAN MAXVALUE
    (
        SUBPARTITION s4,
        SUBPARTITION s5
    )
);
--插入数据。
m_db=# insert into range_hash1 values(1,19800101),(2,19800601),(3,19850101),(4,19850601),
```



```
(5,19900102);
--混合收集一级、二级分区统计信息
m_db=# ALTER TABLE range_hash1 ANALYZE PARTITION s0,s1;
m_db=# ALTER TABLE range_hash1 ANALYZE PARTITION p0,p1;
m_db=# ALTER TABLE range_hash1 ANALYZE PARTITION all;
m_db=# ALTER TABLE range_hash1 ANALYZE PARTITION p2,s4,s5;
m_db=# ALTER TABLE range_hash1 ANALYZE PARTITION all;
m_db=# DROP TABLE range_hash1;
```

相关链接

CREATE TABLE PARTITION，DROP TABLE

4.4.2.6.13 ALTER TABLE SUBPARTITION

功能描述

修改二级分区表分区，包括增删分区、清空分区、切割/合并分区、移动分区表空间、交换分区、重命名分区、开启/关闭分区自动扩展功能，以及修改分区属性等。

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一级分区和二级分区的名称相同。
- 添加分区的分区键值要和分区表的分区键的类型一致。
- 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。
- 当分区表只有一个一级分区或二级分区时，不能删除该分区。
- 选择分区使用PARTITION FOR()或SUBPARTITION 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。
- 在M-Compatibility模式下，SUBPARTITION只支持HASH/KEY分区，但不支持添加HASH/KEY分区。只有一种情况例外，二级分区表的二级分区方式为HASH且一级分区方式不是HASH，此时支持新增一级分区并创建对应的二级分区；不支持删除HASH/KEY分区；不支持切割和合并HASH/KEY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。
- 删除、切割、清空、交换分区的操作会使Global索引失效，可以申明UPDATE GLOBAL INDEX子句同步更新索引。
- 如果删除、切割、清空、交换分区操作不申明UPDATE GLOBAL INDEX子句，并发的DML业务有可能因为索引不可用而报错。
- 若设置参数enable_gpi_auto_update为on，即使不申明UPDATE GLOBAL INDEX子句，也会自动更新Global索引。
- 开启一级/二级列表分区自动扩展要求对应层级的分区中不能存在分区键值为DEFAULT的分区。

语法格式

修改二级分区表分区包括修改表分区主语法、修改表分区名称的语法、重置分区ID和开启/关闭分区自动扩展功能的语法。

- 修改表分区主语法。

```
ALTER TABLE { 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，M-Compatibility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 |
truncate_clause |
set_partitioning_clause
```

- move_clause子语法用于移动分区到新的表空间。

  ```
  MOVE SUBPARTITION { subpartition_name | FOR ( subpartition_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 SUBPARTITION { ( subpartition_name ) | FOR ( subpartition_value [, ...] ) }
     WITH TABLE {[ ONLY ] ordinary_table_name | ordinary_table_name * | ONLY ( ordinary_table_name )}
     [ { WITH | WITHOUT } VALIDATION ] [ 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。
  - 普通表索引和分区Local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，分区表只能是二级分区表。

  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，可能会导致不能交换数据。

    如果需要进行数交换数据操作可以通过创建中间表的方式，先将分区数据插入到中间表，truncate分区，普通表数据插入分区表，drop普通表，重命名中间表的方式完成数据交换操作。
  - 如果在普通表/分区表上进行了drop column操作，被删除的列依然物理存在，所以需要保证普通表和分区的被删除列也严格对齐才能交换成功。

- truncate_clause子语法用于清空分区表中的指定分区。语法可以作用在一级分区上。

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

  也可以作用在二级分区上。

  ```
  TRUNCATE SUBPARTITION { subpartition_name | FOR ( subpartition_value [, ...] ) } [ UPDATE GLOBAL INDEX ]
  ```

- 修改表分区名称的语法。可以修改分区表的一级分区。

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

  也可以修改分区表的二级分区。


```
ALTER TABLE [ IF EXISTS ] { table_name [*] | ONLY table_name | ONLY ( table_name )}
    RENAME SUBPARTITION { subpartition_name | FOR ( subpartition_value [, ...] ) } TO
subpartition_new_name;
```

参数说明

- **table_name**

  分区表名。

  取值范围：已存在的分区表名。

- **subpartition_name**

  二级分区名。

  取值范围：已存在的二级分区名。

- **tablespacename**

  指定分区要移动到哪一个表空间。

  取值范围：已存在的表空间名。

  注：需要在非M-Compatibility数据库中创建或删除tablespace。

- **{ ALL | partition_name [, ...] }**

  ALL：清除所有分区数据。

  partition_name ： 目标分区表的分区名，支持一级分区名和二级分区名。

  取值范围：已存在的分区名。

- **partition_value**

  一级分区键值。

  通过PARTITION FOR (partition_value [, ...])子句指定的这一组值，可以唯一确定一个一级分区。

  取值范围：需要进行操作的一级分区的分区键的取值范围。

- **subpartition_value**

  一级分区键值和二级分区键值。

  通过SUBPARTITION FOR (subpartition_value [, ...])子句指定的这一组值，可以唯一确定一个二级分区。

  取值范围：对于需要进行操作的二级分区，需要同时有其一级分区分区键和二级分区分区键的取值范围。

- **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。

- **partition_new_name**

  分区的新名称。

  取值范围：字符串，要符合**标识符说明**。

- **subpartition_new_name**

  二级分区的新名称。

  取值范围：字符串，要符合**标识符说明**。

- **UPDATE GLOBAL INDEX**

  如果使用该参数，则会更新分区表上的所有全局索引，以确保使用全局索引可以查询出正确的数据；如果不使用该参数，则分区表上的所有全局索引将会失效。

示例

请参考**CREATE TABLE SUBPARTITION**的示例。

相关链接

CREATE TABLE SUBPARTITION

4.4.2.6.14 ALTER USER

功能描述

修改数据库用户的属性。

注意事项

ALTER USER中修改的会话参数只针对指定的用户，且在下一次会话中有效。

语法格式

- 修改用户的权限等信息。

```
ALTER USER user_name [ [ WITH ] option [ ... ] ];
ALTER USER user_name [ IN DATABASE database_name ]
    SET configuration_parameter {{ TO | = } { value | DEFAULT }|FROM CURRENT};
ALTER USER user_name
    [ IN DATABASE database_name ] RESET ALL;
```

  其中option子句为。


```
{ CREATEDB | NOCREATEDB }
  | { CREATEROLE | NOCREATEROLE }
  | { AUDITADMIN | NOAUDITADMIN }
  | { SYSADMIN | NOSYSADMIN }
  | {MONADMIN | NOMONADMIN}
  | {OPRADMIN | NOOPRADMIN}
  | {POLADMIN | NOPOLADMIN}
  | { USEFT | NOUSEFT }
  | {INHERIT | NOINHERIT}
  | { LOGIN | NOLOGIN }
  | {PERSISTENCE | NOPERSISTENCE}
  | CONNECTION LIMIT connlimit
  | VALID BEGIN 'timestamp'
  | VALID UNTIL 'timestamp'
  | USER GROUP 'groupuser'
  | NODE GROUP logic_cluster_name
  | ACCOUNT { LOCK | UNLOCK }
```

- 修改与用户关联的指定会话参数值。

```
ALTER USER user_name [ IN DATABASE database_name ]
  SET configuration_parameter { { TO | = } { value | DEFAULT } | FROM CURRENT };
```

- 重置与用户关联的所有会话参数值。

```
ALTER USER user_name
  [ IN DATABASE database_name ] RESET ALL;
```

参数说明

- **user_name**

  现有用户名。

  取值范围：已存在的用户名，用户名要求详见•**user_name**。

- **new_password**

  新密码。

  密码规则如下：

  - 不能与当前密码相同。
  - 密码默认不少于8个字符。
  - 不能与用户名及用户名倒序相同。
  - 至少包含大写字母（A-Z），小写字母（a-z），数字（0-9），非字母数字字符（限定为~!@#$%^&*()-_=+\|[{}];:,<.>/?）四类字符中的三类字符。
  - 应当使用单引号将用户密码括起来。

  取值范围：字符串。

- **old_password**

  旧密码。

- **ACCOUNT { LOCK | UNLOCK }**
  - ACCOUNT LOCK：锁定账户，禁止登录数据库。
  - ACCOUNT UNLOCK：解锁账户，允许登录数据库。
- **PGUSER**

  当前版本不允许修改用户的PGUSER属性。

其他参数请参见**CREATE ROLE**和**ALTER ROLE**的参数说明。

示例

请参考CREATE USER的**示例**。

相关链接

CREATE ROLE，CREATE USER，DROP USER

4.4.2.6.15 ALTER VIEW

功能描述

ALTER VIEW更改视图的各种辅助属性。

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，三权分立开关关闭时，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。三权分立开关打开时，系统管理员不能修改视图模式。0
- 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。三权分立开关打开时，系统管理员不能修改视图的所有者。
- 禁止修改不可更新视图当中的列名、列类型以及删除列；允许修改可更新视图列名、列类型以及删除列。
- 不建议对嵌套视图中底层视图做修改列名、列类型以及删除列操作，会导致上层视图不可用。

语法格式

- 设置视图列的默认值。
```
ALTER VIEW [ IF EXISTS ] view_name
    ALTER [ COLUMN ] column_name SET DEFAULT expression;
```
- 取消列视图列的默认值。
```
ALTER VIEW [ IF EXISTS ] view_name
    ALTER [ COLUMN ] column_name DROP DEFAULT;
```
- 修改视图的所有者。
```
ALTER VIEW [ IF EXISTS ] view_name
    OWNER TO new_owner;
```
- 重命名视图。
```
ALTER VIEW [ IF EXISTS ] view_name
    RENAME TO new_name;
```
- 设置视图的所属模式。
```
ALTER VIEW [ IF EXISTS ] view_name
    SET SCHEMA new_schema;
```
- 设置视图的定义。
```
ALTER VIEW view_name [ ( column_name [, ...] ) ]
    AS query [WITH [CASCADED | LOCAL] CHECK OPTION];
```

  说明

  更改视图的查询定义实际使用的是CREATE OR REPLACE VIEW，两者功能相同。

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。

- **view_name**
  视图名称，可以用模式修饰。
  取值范围：字符串，已经存在的视图名。
- **column_name**
  字段名称。
  取值范围：字符串，已经存在的视图字段名。
- **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，用户名要求详见**•user_name**。
- **new_name**
  视图的新名称。
- **new_schema**
  视图的新模式。
- **expression**
  常量、函数或SQL表达式。
- **WITH [CASCADED | LOCAL] CHECK OPTION**
  设置控制更新视图的行为，取值如下：
  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。
  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，否则底层视图定义的条件都不检查。

示例

- 重命名视图。

```
--创建test_tbl表。
m_db=# CREATE TABLE test_tb1(col1 INT,col2 INT);
--创建视图。
m_db=# CREATE VIEW abc AS SELECT * FROM test_tb1;
--重命名视图。
m_db=# ALTER VIEW IF EXISTS abc RENAME TO test_v1;
--查看视图。
m_db=# \dv
            List of relations
 Schema |  Name   | Type | Owner | Storage
--------+---------+------+-------+---------
 public | test_v1 | view | omm   |
(1 row)
```

- 修改视图所有者。

```
--创建用户。
m_db=# CREATE ROLE role_test PASSWORD '********';
--修改视图所有者。
m_db=# ALTER VIEW IF EXISTS test_v1 OWNER TO role_test;
--查看视图信息。
m_db=# \dv
              List of relations
 Schema |  Name   | Type |   Owner   | Storage
--------+---------+------+-----------+---------
 public | test_v1 | view | role_test |
(1 row)
```


- 设置视图所属模式。

```
--创建模式。
m_db=# CREATE SCHEMA tcpds;
--修改视图所属模式。
m_db=# ALTER VIEW test_v1 SET SCHEMA tcpds;
--查看视图信息。
m_db=# \dv tcpds.test_v1;
              List of relations
 Schema |  Name   | Type |   Owner   | Storage
--------+---------+------+-----------+---------
 tcpds  | test_v1 | view | role_test |
(1 row)
```

相关链接

CREATE VIEW，**DROP VIEW**

4.4.2.6.16 ANALYZE

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、pg_statistic_ext下，执行ANALYZE命令后，可在上述系统表中查询收集到的统计信息，也可以通过系统视图pg_stats、pg_ext_stats查询信息。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。

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_name和partition_name参数把分析限定在特定的表、列或分区表中。

每次收集的统计信息，都会存入统计信息历史表（gs_statistic_history、gs_statistic_ext_history、gs_statistic_history）中，历史表存放的数量和统计信息的保留时间由GUC参数stats_history_record_limit和stats_history_retention_time控制。

注意事项

- ANALYZE非临时表不能在一个匿名块、事务块、函数或存储过程内被执行。支持存储过程中ANALYZE临时表，不支持统计信息回滚操作。
- ANALYZE VERIFY操作处理的大多为异常场景检测需要使用RELEASE版本。ANALYZE VERIFY 场景不触发远程读，因此远程读参数不生效。对于关键系统表出现错误被系统检测出页面损坏时，将直接报错不再继续检测。
- 如果没有指定参数，ANALYZE处理当前数据库里用户拥有相应权限的每个表。如果参数中指定了表，ANALYZE只处理指定的表。
- 要对一个表进行ANALYZE操作，通常用户必须是表的所有者或者被授予了指定表ANALYZE权限的用户，三权分立开关关闭时，默认系统管理员有该权限。数据库的所有者允许对数据库中除了共享目录以外的所有表进行ANALYZE操作（该限制意味着只有系统管理员才能真正对一个数据库进行ANALYZE操作）。ANALYZE命令会跳过那些用户没有权限的表。
- ANALYZE不收集无法做比较或等值运算的列，例如cursor类型。

语法格式

- 收集表的统计信息。

```
ANALYZE [ VERBOSE ]
    [ table_name [ ( column_name [, ...] ) ] ] ;
```


- 收集分区表的分区统计信息。

```
ANALYZE [ VERBOSE ]
    table_name [ ( column_name [, ...] ) ] PARTITION ( partition_name ) ;
```

  说明

  使用关键字PARTITION，partition_name必须为一级分区名字。

- 手动收集多列统计信息。

```
ANALYZE [ VERBOSE ]
    table_name (( column_1_name, column_2_name [, ...] ));
```

  说明

  - 如果关闭GUC参数enable_functional_dependency，每组多列统计信息最多支持32列；如果开启GUC参数enable_functional_dependency，每组多列统计信息最多支持4列。
  - 不支持收集多列统计信息的表：系统表，全局临时表。

- 自动收集多列统计信息。

  打开auto_statistic_ext_columns参数后执行ANALYZE，自动根据该表的索引前缀创建多列统计信息，多列统计信息的列数不超过设置的auto_statistic_ext_columns值。

  例：表t存在索引(a,b,c,d)，设置auto_statistic_ext_columns参数为4，则analyze t将创建关于(a,b)、(a,b,c)、(a,b,c,d)的多列统计信息。

```
ANALYZE [ VERBOSE ] table_name;
```

参数说明

- **VERBOSE**

  启用显示进度信息。

  说明

  如果指定了VERBOSE，ANALYZE发出进度信息，表明目前正在处理的表。各种有关表的统计信息也会打印出来。

- **table_name**

  需要分析的特定表的表名（可能会带模式名），如果省略，将对数据库中的所有表（非外部表）进行分析。

  对于ANALYZE收集统计信息，目前仅支持行存表。

  取值范围：已有的表名。

- **column_name，column_1_name，column_2_name**

  需要分析特定列的列名，默认为所有列。

  取值范围：已有的列名。

- **partition_name**

  如果table为分区表，在关键字PARTITION后面指定分区名partition_name表示分析该分区表的统计信息。

  取值范围：表的某一个分区名。

示例

--- 创建表。

```
m_db=# 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。

```
m_db=# 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)
);
```

--- 使用ANALYZE语句更新统计信息。

```
m_db=# ANALYZE customer_info;
m_db=# ANALYZE customer_par;
```

--- 使用ANALYZE VERBOSE语句更新统计信息，并输出表的相关信息。

```
m_db=# ANALYZE VERBOSE customer_info;
INFO:  analyzing "public.customer_info"(cn_5002 pid=53078)
INFO:  analyzing "public.customer_info" inheritance tree(cn_5002 pid=53078)
ANALYZE
```

说明

环境若有故障，需查看数据库主节点的log。

--- 删除表。

```
m_db=# DROP TABLE customer_info;
m_db=# DROP TABLE customer_par;
```

4.4.2.7 B

4.4.2.7.1 BEGIN

功能描述

通过BEGIN启动事务。如果声明了隔离级别、读写模式，那么新事务就使用这些特性，类似执行了**SET TRANSACTION**。

注意事项

无。

语法格式

开启事务

```
BEGIN [ WORK | TRANSACTION ]
 [
   {
```



```
    ISOLATION LEVEL { READ COMMITTED | READ UNCOMMITTED | SERIALIZABLE | REPEATABLE READ }
    | { READ WRITE | READ ONLY }
  } [, ...]
];
```

参数说明

- **WORK | TRANSACTION**

  BEGIN格式中的可选关键字，没有实际作用。

- **ISOLATION LEVEL**

  指定事务隔离级别，它决定当一个事务中存在其他并发运行事务时它能够看到什么数据。

  说明

  在事务中第一个数据修改语句（SELECT、INSERT、DELETE、UPDATE、COPY）执行之后，事务隔离级别就不能再次设置。

  取值范围：

  - READ COMMITTED：读已提交隔离级别，只能读到已经提交的数据，而不会读到未提交的数据。这是缺省值。
  - READ UNCOMMITTED：读未提交隔离级别，指定后的行为和READ COMMITTED行为一致。
  - REPEATABLE READ：可重复读隔离级别，仅仅看到事务开始之前提交的数据，它不能看到未提交的数据，以及在事务执行期间由其它并发事务提交的修改。
  - SERIALIZABLE：M-Compatibility目前功能上不支持此隔离级别，设置该隔离级别时，等价于REPEATABLE READ。

- **READ WRITE | READ ONLY**

  指定事务访问模式（读/写或者只读）。

示例

```
--创建SCHEMA。
m_db=# CREATE SCHEMA tpcds;

--创建表tpcds.reason。
m_db=# CREATE TABLE tpcds.reason (c1 int, c2 int);

--以默认方式启动事务。
m_db=# START TRANSACTION;
m_db=# SELECT * FROM tpcds.reason;
m_db=# COMMIT;

--以默认方式启动事务。
m_db=# BEGIN;
m_db=# SELECT * FROM tpcds.reason;
m_db=# COMMIT;

--以隔离级别为READ COMMITTED，读/写方式启动事务。
m_db=# BEGIN ISOLATION LEVEL READ COMMITTED READ WRITE;
m_db=# SELECT * FROM tpcds.reason;
m_db=# COMMIT;

--删除表tpcds.reason。
m_db=# DROP TABLE tpcds.reason;

--删除SCHEMA。
m_db=# DROP SCHEMA tpcds;
```


相关链接

START TRANSACTION

4.4.2.8 C

4.4.2.8.1 CLEAN CONNECTION

功能描述

用来清理数据库连接。允许在节点上清理指定数据库的指定用户的相关连接。

注意事项

- M-Compatibility下不支持指定节点，仅支持TO ALL。
- 该功能仅在force模式下，可以清理正在使用的正常连接。

语法格式

```
CLEAN CONNECTION
    TO { COORDINATOR ( nodename [, ... ] ) | NODE ( nodename [, ... ] )| ALL [ CHECK ] [ FORCE ] }
    [ FOR DATABASE dbname ]
    [ TO USER username ];
```

参数说明

- **CHECK**

  仅在节点列表为TO ALL时可以指定。如果指定该参数，会在清理连接之前检查数据库是否被其他会话连接访问。此参数主要用于DROP DATABASE之前的连接访问检查，如果发现有其他会话连接，则将报错并停止删除数据库。

- **FORCE**

  仅在节点列表为TO ALL时可以指定，如果指定该参数，所有和指定dbname和username相关的线程都会收到SIGTERM信号，然后被强制关闭。

- **COORDINATOR (nodename [, ...]) | NODE (nodename [, ...]) | ALL**

  仅支持TO ALL，必须指定该参数，节点上的指定连接会被全部删除。

- **dbname**

  删除指定数据库上的连接。如果不指定，则删除所有数据库的连接。

  取值范围：已存在数据库名。

- **username**

  删除指定用户上的连接。如果不指定，则删除所有用户的连接。

  取值范围：已存在的用户。

示例

```
--创建数据库test_clean_connection。
m_db=# CREATE DATABASE test_clean_connection;

--创建jack用户。
m_db=# CREATE USER jack PASSWORD '********';

--在另一个会话用jack用户登录该数据库之后，通过视图查询到该连接信息。
m_db=# SELECT datname,usename,application_name,waiting,state
```



```
         FROM pg_stat_activity
         WHERE datname = 'test_clean_connection';
       datname        | usename | application_name | waiting | state
----------------------+---------+------------------+---------+-------
 test_clean_connection | jack    | gsql             | f       | idle
(1 row)

--此时直接删除数据库test_clean_connection会有如下报错：
m_db=# DROP DATABASE test_clean_connection;
ERROR:  Database "test_clean_connection" is being accessed by other users. You can stop all connections by command: "clean connection to all force for database XXXX;" or wait for the sessions to end by querying view: "pg_stat_activity".
DETAIL:  There is 1 other session using the database.

--删除登录数据库数据库test_clean_connection的所有节点的连接。
--如果不使用FORCE参数是无法删除stat状态为其他的状态的连接。
m_db=# CLEAN CONNECTION TO ALL FORCE FOR DATABASE test_clean_connection;

--查询登录数据库test_clean_connection的连接。
m_db=# SELECT datname,usename,application_name,waiting,state
         FROM pg_stat_activity
         WHERE datname = 'test_clean_connection';
 datname | usename | application_name | waiting | state
---------+---------+------------------+---------+-------
(0 rows)


--删除数据库test_clean_connection。
m_db=# DROP DATABASE test_clean_connection;

--删除用户jack。
m_db=# DROP USER jack;
```

4.4.2.8.2 CHECKPOINT

功能描述

检查点（CHECKPOINT）是一个事务日志中的点，所有数据文件都在该点被更新以反映日志中的信息，所有数据文件都将被刷新到磁盘。

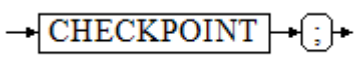
设置事务日志检查点。预写式日志（WAL）缺省时在事务日志中每隔一段时间放置一个检查点。可以使用gs_guc命令设置相关运行时参数（checkpoint_segments，checkpoint_timeout和incremental_checkpoint_timeout）来调整这个原子化检查点的间隔。

注意事项

- 只有系统管理员和运维管理员可以调用CHECKPOINT。
- CHECKPOINT强制立即进行检查，而不是等到下一次调度时的检查点。

语法格式

```
CHECKPOINT;
```



参数说明

无。

示例

```
--设置检查点。
m_db=# CHECKPOINT;
```

4.4.2.8.3 COMMENT

功能描述

定义或修改一个对象的注释。

注意事项

- 每个对象只存储一条注释，因此要修改一个注释，对同一个对象发出一条新的COMMENT命令即可。要删除注释，在文本字符串的位置写上NULL即可。当删除对象时，注释自动被删除。
- 目前注释浏览没有安全机制，任何连接到某数据库上的用户都可以看到所有该数据库对象的注释。共享对象（比如数据库、角色、表空间）的注释是全局存储的，连接到任何数据库的任何用户都可以看到它们。因此，不要在注释里存放与安全有关的敏感信息。
- 对大多数对象，只有对象的所有者或者被授予了对象COMMENT权限的用户可以设置注释，系统管理员默认拥有该权限。
- 角色没有所有者，所以COMMENT ON ROLE命令仅可以由系统管理员对系统管理员角色执行，有CREATE ROLE权限的角色也可以为非系统管理员角色设置注释。系统管理员可以对所有对象进行注释。

语法格式

```
COMMENT ON
{ COLLATION object_name |
  COLUMN { table_name.column_name | view_name.column_name } |
  CONSTRAINT constraint_name ON table_name |
  DATABASE object_name |
  INDEX object_name |
  SEQUENCE seq_name |
  USER user_name |
  ROLE object_name |
  SCHEMA object_name |
  TABLE object_name |
  TABLESPACE object_name |
  VIEW object_name |
}
   IS 'text';
```

参数说明

- **user_name**

  取值范围：已存在的用户名，用户名要求详见•**user_name**。
- **object_name**

  对象名。
- **table_name.column_name**

  view_name.column_name

  列名称。前缀可加表名称或者视图名称。
- **constraint_name**

  表约束的名称。

- **table_name**
  表的名称。
- **text**
  注释。

示例

```
--建表。
m_db=# CREATE TABLE emp(
    empno varchar(7),
    ename varchar(50),
    job varchar(50),
    mgr varchar(7),
    deptno int
);
--表添加注释。
m_db=# COMMENT ON TABLE emp IS '部门表';
--字段添加注释。
m_db=# COMMENT ON COLUMN emp.empno  IS  '员工编号';
m_db=# COMMENT ON COLUMN emp.ename  IS  '员工姓名';
m_db=# COMMENT ON COLUMN emp.job    IS  '职务';
m_db=# COMMENT ON COLUMN emp.mgr    IS  '上司编号';
m_db=# COMMENT ON COLUMN emp.deptno IS  '部门编号';

--查看表的注释。
m_db=# \d+
 Schema |    Name     |   Type   | Owner |    Size    |              Storage               | Description
--------+-------------+----------+-------+------------+------------------------------------+-------------
 public | emp         | table    | omm   | 0 bytes    | {orientation=row,compression=no} | 部门表

--查看字段注释。
m_db=# \d+ emp
                            Table "public.emp"
 Column |          Type          | Modifiers | Storage  | Stats target | Description
--------+------------------------+-----------+----------+--------------+-------------
 empno  | character varying(7)   |           | extended |              | 员工编号
 ename  | character varying(50)  |           | extended |              | 员工姓名
 job    | character varying(50)  |           | extended |              | 职务
 mgr    | character varying(7)   |           | extended |              | 上司编号
 deptno | integer                |           | plain    |              | 部门编号
Has OIDs: no
Options: orientation=row, compression=no, storage_type=USTORE

--删除表emp。
m_db=# DROP TABLE emp;
```

4.4.2.8.4 COMMIT

功能描述

通过COMMIT可完成提交事务的功能，即提交事务的所有操作。

注意事项

执行COMMIT这个命令的时候，命令执行者必须是该事务的创建者或系统管理员，且创建和提交操作可以不在同一个会话中。

语法格式

```
COMMIT [ WORK | TRANSACTION ];
```


参数说明

- **COMMIT**

  提交当前事务，让所有当前事务的更改为其他事务可见。

- **WORK | TRANSACTION**

  可选关键字，除了增加可读性没有其他任何作用。

示例

```
-- 创建SCHEMA。
m_db=# CREATE SCHEMA tpcds;

--创建表。
m_db=# CREATE TABLE tpcds.customer_demographics_t2
(
    CD_DEMO_SK               INTEGER               NOT NULL,
    CD_GENDER                CHAR(1)                     ,
    CD_MARITAL_STATUS        CHAR(1)                     ,
    CD_EDUCATION_STATUS      CHAR(20)                    ,
    CD_PURCHASE_ESTIMATE     INTEGER                     ,
    CD_CREDIT_RATING         CHAR(10)                    ,
    CD_DEP_COUNT             INTEGER                     ,
    CD_DEP_EMPLOYED_COUNT    INTEGER                     ,
    CD_DEP_COLLEGE_COUNT     INTEGER
)
;

--开启事务。
m_db=# START TRANSACTION;

--插入数据。
m_db=# INSERT INTO tpcds.customer_demographics_t2 VALUES(1,'M', 'U', 'DOCTOR DEGREE', 1200,
'GOOD', 1, 0, 0);
m_db=# INSERT INTO tpcds.customer_demographics_t2 VALUES(2,'F', 'U', 'MASTER DEGREE', 300, 'BAD', 1,
0, 0);

--提交事务，让所有更改永久化。
m_db=# COMMIT;

--查询数据。
m_db=# SELECT * FROM tpcds.customer_demographics_t2;

--删除表tpcds.customer_demographics_t2。
m_db=# DROP TABLE tpcds.customer_demographics_t2;

-- 删除SCHEMA。
m_db=# DROP SCHEMA tpcds;
```

相关链接

ROLLBACK

4.4.2.8.5 COPY

功能描述

通过COPY命令实现在表和文件之间复制数据。

COPY FROM从一个文件复制数据到一个表，COPY TO把一个表的数据复制到一个文件。

注意事项

- 当参数enable_copy_server_files关闭时，只允许初始用户执行COPY FROM FILENAME或COPY TO FILENAME命令，当参数enable_copy_server_files打开时，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COPY FROM FILENAME或COPY TO FILENAME，以防止用户越权查看或修改敏感文件。同时enable_copy_server_files打开时，管理员可以通过guc参数safe_data_path设置普通用户可以导入导出的路径必须为设置路径的子路径，未设置此guc参数时候（默认情况），不对普通用户使用的路径进行拦截。该参数会对copy使用路径中的...进行报错处理。
- COPY只能用于表，不能用于视图。
- COPY TO需要读取的表的SELECT权限，COPY FROM需要插入的表的INSERT权限。
- 如果声明了一个字段列表，COPY将只在文件和表之间复制已声明字段的数据。如果表中有任何不在字段列表里的字段，COPY FROM将为那些字段插入缺省值。
- 如果声明了数据源文件，服务器必须可以访问该文件；如果指定了STDIN，数据将在客户前端和服务器之间流动，输入时，表的列与列之间使用TAB键分隔，在新的一行中以反斜杠和句点（\.）表示输入结束。
- 如果数据文件的任意行包含比预期多或者少的字段，COPY FROM将抛出一个错误。
- 数据的结束可以用一个只包含反斜杠和句点（\.）的行表示。如果从文件中读取数据，数据结束的标记是不必要的；如果在客户端应用之间复制数据，必须要有结束标记。
- COPY FROM中\N为空字符串，如果要输入实际数据值\N ，使用\\N。
- COPY FROM不支持在导入过程中对数据做预处理（比如说表达式运算，填充指定默认值等）。如果需要在导入过程中对数据做预处理，用户需先把数据导入到临时表中，然后执行SQL语句通过运算插入到表中，但此方法会导致I/O膨胀，降低导入性能。
- COPY FROM在遇到数据格式错误时会回滚事务，但没有足够的错误信息，不方便用户从大量的原始数据中定位错误数据。
- COPY FROM/TO适合低并发，本地小数据量导入导出。
- COPY命令中，生成列不能出现在指定列的列表中。使用COPY… TO导出数据时，如果没有指定列的列表，则该表的所有列除了生成列都会被导出。COPY… FROM导入数据时，生成列会自动更新，并像普通列一样保存。
- COPY是服务端命令，执行环境和数据库服务端进程保持一致；\COPY是客户端元命令，执行环境和客户端gsql保持一致。需要注意的是，当在云上环境中使用数据库和gsql时，COPY命令和\COPY命令都使用云上环境的路径；当在云上环境中使用数据库，在非云上环境使用gsql时，COPY命令使用云上环境内的路径，\COPY命令则使用其他环境的路径。
- 在导出float类型的数据时，推荐将extra_float_digits设为3，避免有效位数丢失造成导出导入前后数据不一致。
- M-Compatibility模式数据库下，不建议在lower_case_table_names参数不同的库之间进行COPY FROM/TO操作。

语法格式

- 从一个文件复制数据到一个表。


```
COPY table_name [ ( column_name [, ...] ) ] [WITH OIDS]
    FROM { 'filename' | STDIN } [LOAD] [ LOAD_DISCARD 'discard_path'] [LOAD_BAD 'bad_path']
    [ [ WITH ] ( option [, ...] ) | copy_option ];
```

说明

上述语法中copy_option与option语法不兼容。

- 把一个表的数据复制到一个文件。

```
COPY table_name [ ( column_name [, ...] ) ] [WITH OIDS]
    TO { 'filename' | STDOUT } [LOAD]
  [ [ WITH ] ( option [, ...] ) | copy_option ];

COPY query
    TO { 'filename' | STDOUT }
  [ [ WITH ] ( option [, ...] ) | copy_option ];
```

说明

copy_option是指COPY原生的参数形式，而option是兼容外表导入的参数形式。

其中可选参数option子句语法为：

```
FORMAT 'format_name'
| FORMAT binary
| DELIMITER 'delimiter_character'
| NULL  'null_string'
| QUOTE 'quote_character'
| HEADER [ boolean ]
| USEEOF [ boolean ]
| ESCAPE 'escape_character'
| FORCE_NOT_NULL ( column_name [, ...] )
| FORCE_QUOTE { column_name [, ...] | * }
| DATE_FORMAT 'date_format_string'
| SMALLDATETIME_FORMAT 'smalldatetime_format_string'
```

其中可选参数copy_option子句语法为：

```
NULL [ AS ] 'null_string'
| FORCE_NOT_NULL column_name [, ...]
| BINARY
| CSV
| ESCAPE [ AS ] 'escape_character'
| ENCODING 'encoding_name'
| DELIMITER [ AS ] 'delimiter_character'
| ROWS 'string'
| DATE_FORMAT 'date_format_string'
| SMALLDATETIME_FORMAT 'smalldatetime_format_string'
| SKIP int_number
| WHEN { ( start - end ) | column_name } { = | != } 'string'
| SEQUENCE ( { column_name ( integer [, incr] ) [, ...] } )
```

参数说明

- **query**

  其结果将被复制。

  取值范围：一个必须用圆括弧包围的SELECT或VALUES命令。

- **table_name**

  表的名称（可以有模式修饰）。

  取值范围：已存在的表名。

- **column_name**

  可选的待复制字段列表。

  取值范围：如果没有声明字段列表，将使用所有字段。

- **STDIN**

声明输入是来自标准输入。输入时，表的列与列之间使用TAB键分隔，在新的一行中以反斜杠和句点（\.）表示输入结束。

- **STDOUT**
  声明输出打印到标准输出。
- **LOAD**
  区分COPY语法为gs_loader调用还是用户直接调用，非主动调用接口，不推荐使用。
- **LOAD_DISCARD 'discard_path'**
  指定gs_loader客户端discard文件生成路径，非主动调用接口，不推荐使用。
- **LOAD_BAD 'bad_path'**
  指定gs_loader客户端bad文件生成路径，非主动调用接口，不推荐使用。
- **OPTION { option_name ' value ' }**
  用于指定兼容外表的各类参数。
  - FORMAT
    数据源文件的格式。
    取值范围：CSV、TEXT、BINARY。
    - CSV格式的文件，可以有效处理数据列中的换行符，但对一些特殊字符处理有欠缺。
    - TEXT格式的文件，可以有效处理一些特殊字符，但无法正确处理数据列中的换行符。
    - BINARY形式的选项会使得所有的数据被存储/读作二进制格式而不是文本。 这比TEXT和CSV格式的要快一些，但是一个BINARY格式文件可移植性比较差。

    缺省值：TEXT
  - DELIMITER
    指定数据文件行数据的字段分隔符。

    说明
    - 分隔符不能是\r和\n。
    - 分隔符不能和null参数相同，CSV格式数据的分隔符不能和quote参数相同。
    - TEXT格式数据的分隔符不能包含： 小写字母、数字和特殊字符.\。
    - 数据文件中单行数据长度需<1GB，如果分隔符较长且数据列较多的情况下，会影响导出有效数据的长度。
    - 分隔符推荐使用多字符和不可见字符。多字符例如'$^&'；不可见字符例如0x07，0x08，0x1b等。

    取值范围：支持多字符分隔符，但分隔符不能超过10个字节。
    缺省值：
    - TEXT格式的默认分隔符是水平制表符（tab）。
    - CSV格式的默认分隔符为“,”。
  - NULL
    用来指定数据文件中空值的表示。
    取值范围：

- null值不能是\r和\n，最大为100个字符。
- null值不能和分隔符、quote参数相同。

缺省值：

- CSV格式下默认值是一个没有引号的空字符串。
- 在TEXT格式下默认值是\N。

– HEADER

指定导出数据文件是否包含标题行，标题行一般用来描述表中每个字段的信息。header只能用于CSV格式的文件中。

在导入数据时，如果header选项为on，则数据文本第一行会被识别为标题行，会忽略此行。如果header为off，而数据文件中第一行会被识别为数据。

在导出数据时，如果header选项为on，则需要指定fileheader。如果header为off，则导出数据文件不包含标题行。

取值范围：true/on，false/off。

缺省值：false

– ESCAPE

CSV格式下，用来指定逃逸字符，逃逸字符只能指定为单字节字符。

缺省值：''。当与quote值相同时，会被替换为'\0'。

– FORCE_NOT_NULL (column_name [, ...])

在CSV COPY FROM模式下，指定的字段输入不能为空。

取值范围：已存在的字段。

– DATE_FORMAT

导入对于DATE类型指定格式。此参数不支持BINARY格式，会报“cannot specify bulkload compatibility options in BINARY mode”错误信息。此参数仅对COPY FROM导入有效。

取值范围：合法DATE格式。可参考**日期时间函数**。

说明

对于DATE类型内建为TIMESTAMP类型的数据库，在导入的时候，若需指定格式，可以参考下面的timestamp_format参数。

- **COPY_OPTION { option_name ' value ' }**

用于指定COPY原生的各类参数。

– NULL null_string

用来指定数据文件中空值的表示。

须知

在使用COPY FROM的时候，任何匹配这个字符串的字符串将被存储为NULL值，所以应该确保指定的字符串和COPY TO相同。

取值范围：

- null值不能是\r和\n，最大为100个字符。

- null值不能和分隔符、quote参数相同。

缺省值：

- 在TEXT格式下默认值是\N。
- CSV格式下默认值是一个没有引号的空字符串。

– FORCE NOT NULL column_name [, ...]

在CSV COPY FROM模式下，指定的字段不为空。若输入为空，则将视为长度为0的字符串。

取值范围：已存在的字段。

– BINARY

使用二进制格式存储和读取，而不是以文本的方式。

说明

- 在二进制模式下，不能声明DELIMITER、NULL、CSV选项。
- 指定BINARY类型后，不能再通过option或copy_option指定CSV、TEXT等类型。
- 如果设置了GUC参数support_binary_copy_version='header_encoding'，则通过COPY TO导出的二进制文件头中包含数据库的服务端编码信息，COPY FROM导入二进制文件时对编码信息的一致性进行校验，文件中编码信息须与服务端编码保持一致。
- 如果设置了GUC参数copy_special_character_version='no_error'，则会在导入的过程中屏蔽非法字符编码的校验，将非法编码字符按原样导入，查询时以乱码显示。在了解后果的情况下，请谨慎选择开启此参数。可以在COPY语句中使用LOG ERRORS或LOG ERRORS DATA参数，将错误编码数据记录到错误表中，便于记录与查看。
- 二进制模式下copy_special_character_version='no_error'，仅对TEXT、CHAR、VARCHAR、NVARCHAR2类型的字段生效。

– CSV

打开逗号分隔变量（CSV）模式。指定CSV类型后，不能再通过option或copy_option指定BINARY、TEXT等类型。

– ESCAPE [AS] 'escape_character'

CSV格式下，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''。当与quote值相同时，会被替换为'\0'。

– ENCODING 'encoding_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。

– DATE_FORMAT 'date_format_string'

导入对于DATE类型指定格式。此参数不支持BINARY格式，会报“cannot specify bulkload compatibility options in BINARY mode”错误信息。此参数仅对COPY FROM导入有效。

取值范围：合法DATE格式。可参考**日期时间函数**。

说明

对于DATE类型内建为TIMESTAMP类型的数据库，在导入的时候，若需指定格式，可以参考下面的timestamp_format参数。

- SKIP int_number

  指定数据导入时，跳过数据文件的前 int_number行。
- WHEN { (start - end) | column_name } { = | != } 'string'

  数据导入时，检查导入的每一行数据，只有符合WHEN条件的数据行才导入表中。
- SEQUENCE ({ column_name (integer [, incr]) [, ...] })

  数据导入时，SEQUENCE修饰的列，不从数据文件读取数据，通过指定的integer，按照incr递增数值；不指定incr则默认从1开始递增。

COPY FROM能够识别的特殊反斜杠序列如下所示。

- **\b**：反斜杠 （ASCII 8）
- **\f**：换页（ASCII 12）
- **\n**：换行符 （ASCII 10）
- **\r**：回车符 （ASCII 13）
- **\t**：水平制表符 （ASCII 9）
- **\v**：垂直制表符 （ASCII 11）
- **\digits**：反斜杠后面跟着一到三个八进制数，表示ASCII值为该数的字符。
- **\xdigits**：反斜杠x后面跟着一个或两个十六进制位声明指定数值编码的字符。

权限控制示例

```
m_db=# copy t1 from '/home/xy/t1.csv';
ERROR:  COPY to or from a file is prohibited for security concerns
HINT:  Anyone can COPY to stdout or from stdin. gsql's \copy command also works for anyone.
m_db=# grant gs_role_copy_files to xxx;
```

此错误为非初始用户没有使用copy的权限示例，解决方式为打开enable_copy_server_files参数，则管理员可以使用copy功能，普通用户需要在此基础上加入gs_role_copy_files群组。

示例

```
--创建SCHEMA。
m_db=# CREATE SCHEMA tpcds;

--创建tpcds.ship_mode表。
m_db=# CREATE TABLE tpcds.ship_mode
(
    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)
)
;

--向tpcds.ship_mode表插入一条数据。
m_db=# INSERT INTO tpcds.ship_mode VALUES (1,'a','b','c','d','e');

--将tpcds.ship_mode中的数据复制到/home/omm/ds_ship_mode.dat文件中。
m_db=# COPY tpcds.ship_mode TO '/home/omm/ds_ship_mode.dat';

--将tpcds.ship_mode 输出到STDOUT。
m_db=# COPY tpcds.ship_mode TO STDOUT;

--将tpcds.ship_mode 的数据输出到STDOUT，使用参数如下：分隔符为','(delimiter ',')，编码格式为
```



```
UTF8(encoding 'utf8')。
m_db=# COPY tpcds.ship_mode TO STDOUT WITH (delimiter ',', encoding 'utf8');

--将tpcds.ship_mode 的数据输出到STDOUT，使用参数如下：导入格式为CSV（format 'CSV'），引号包围SM_SHIP_MODE_SK字段的导出内容(force_quote(SM_SHIP_MODE_SK))。
m_db=# COPY tpcds.ship_mode TO STDOUT WITH (format 'CSV', force_quote(SM_SHIP_MODE_SK));

--创建tpcds.ship_mode_t1表。
m_db=# 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)
)
;

--从STDIN复制数据到表tpcds.ship_mode_t1。
m_db=# COPY tpcds.ship_mode_t1 FROM STDIN;

--从/home/omm/ds_ship_mode.dat文件复制数据到表tpcds.ship_mode_t1。
m_db=# COPY tpcds.ship_mode_t1 FROM '/home/omm/ds_ship_mode.dat';

--从/home/omm/ds_ship_mode.dat文件复制数据到表tpcds.ship_mode_t1，使用参数如下：导入格式为TEXT（format 'text'），分隔符为'\t'（delimiter E'\t'），忽略多余列（ignore_extra_data 'true'），不指定转义（noescaping 'true'）。
m_db=# COPY tpcds.ship_mode_t1 FROM '/home/omm/ds_ship_mode.dat' WITH(format 'text', delimiter E'\t', ignore_extra_data 'true', noescaping 'true');

--删除表和SCHEMA。
m_db=# DROP TABLE tpcds.ship_mode;
m_db=# DROP TABLE tpcds.ship_mode_t1;
m_db=# DROP SCHEMA tpcds;
```

4.4.2.8.6 CREATE AUDIT POLICY

功能描述

创建统一审计策略。

注意事项

- 审计策略的创建与维护有权限限制，只有poladmin、sysadmin或初始用户有权限进行此操作。
- 在创建审计策略之前，需要确保已经开启安全策略开关，即设置GUC参数enable_security_policy=on后，脱敏策略才会生效。
- 系统管理员或安全策略管理员可以访问GS_AUDITING_POLICY、GS_AUDITING_POLICY_ACCESS、GS_AUDITING_POLICY_PRIVILEGES和GS_AUDITING_POLICY_FILTERS系统表，查询已创建的审计策略。
- 审计策略名称应具有唯一性，避免与现有策略产生冲突。可以使用IF NOT EXISTS来检查指定的审计策略是否存在，以避免重复创建。

语法格式

```
CREATE AUDIT POLICY [ IF NOT EXISTS ] policy_name { { privilege_audit_clause | access_audit_clause } [, ... ] [ filter_group_clause ] [ ENABLE | DISABLE ] };
```

- privilege_audit_clause：
  ```
  PRIVILEGES { DDL | ALL } [ ON LABEL ( resource_label_name [, ... ] ) ]
  ```


- access_audit_clause：
  ```
  ACCESS { DML | ALL } [ ON LABEL ( resource_label_name [, ... ] ) ]
  ```
- filter_group_clause：
  ```
  FILTER ON { FILTER_TYPE ( filter_value [, ... ] ) } [, ... ]
  ```
- DDL：
  ```
  { ALTER | ANALYZE | COMMENT | CREATE | DROP | GRANT | REVOKE | SET | SHOW }
  ```
- DML：
  ```
  { COPY | DEALLOCATE | DELETE | EXECUTE | REINDEX | INSERT | PREPARE | SELECT | TRUNCATE | UPDATE }
  ```
- FILTER_TYPE：
  ```
  { APP | ROLES | IP }
  ```

参数说明

- **policy_name**

  审计策略名称，需要唯一，不可重复。

  取值范围：字符串，要符合**标识符命名规范**，且最大长度不超过63个字符。若超过63个字符，数据库会截断并保留前63个字符当作审计策略名称。当审计策略名称中包含大写字母时，数据库会自动转换为小写字母，如果需要创建包含大写字母的审计策略名称则需要使用双引号括起来。

  说明

  标识符需要为小写字母（a-z）、大写字母(A-Z)、下划线（_）、数字（0~9）或美元符号（$），且必须以字母或下划线开头。

- **resource_label_name**

  资源标签名称。

- **DDL**

  指的是针对数据库执行如下操作时进行审计，目前支持：CREATE、ALTER、DROP、ANALYZE、COMMENT、GRANT、REVOKE、SET、SHOW。

  取值为ANALYZE时，ANALYZE和VACUUM操作都会被审计。

- **DML**

  指的是针对数据库执行如下操作时进行审计，目前支持：SELECT、COPY、DEALLOCATE、DELETE、EXECUTE、INSERT、PREPARE、REINDEX、TRUNCATE、UPDATE。

- **ALL**

  指的是上述DDL或DML中支持的所有对数据库的操作。当形式为{ DDL | ALL }时，ALL指所有DDL操作；当形式为{ DML | ALL }时，ALL指所有DML操作。

- **FILTER_TYPE**

  描述策略过滤的条件类型，包括APP、ROLES、IP。

- **filter_value**

  指具体过滤信息内容。

- **ENABLE|DISABLE**

  可以打开或关闭统一审计策略。若不指定ENABLE|DISABLE，语句默认为ENABLE。

示例

- 创建一个对数据库执行CREATE的审计策略。

```
--创建adt1策略。
m_db=# CREATE AUDIT POLICY adt1 PRIVILEGES CREATE;

--查看adt1策略。
m_db=# SELECT * FROM GS_AUDITING_POLICY;
 polname | polcomments |         modifydate         | polenabled
---------+-------------+----------------------------+------------
 adt1    |             | 2023-11-06 16:41:40.947417 | t

--查看审计策略的存放位置。
m_db=# SHOW audit_directory;

--删除审计策略adt1。
m_db=# DROP AUDIT POLICY adt1;
```

- 创建一个审计策略，仅审计用户dev_audit进行CREATE操作 。

```
--创建dev_audit用户。
m_db=# CREATE USER dev_audit PASSWORD '********';

--创建一个表tb_for_audit。
m_db=# CREATE TABLE tb_for_audit(col1 text, col2 text, col3 text);

--创建基于tb_for_audit表的adt_lb0资源标签。
m_db=# CREATE RESOURCE LABEL adt_lb0 add TABLE(public.tb_for_audit);

--创建针对adt_lb0资源进行CREATE操作的adt2审计策略。
m_db=# CREATE AUDIT POLICY adt2 PRIVILEGES CREATE ON LABEL(adt_lb0) FILTER ON
ROLES(dev_audit);

--删除审计策略adt2。
m_db=# DROP AUDIT POLICY adt2;

--删除表tb_for_audit。
m_db=# DROP TABLE tb_for_audit;

--删除dev_audit用户。
m_db=# DROP USER dev_audit;
```

- 创建一个仅审计记录用户dev_audit，客户端工具为gsql，IP地址为'10.20.30.40', '127.0.0.0/24'，在执行针对adt_lb0资源进行的SELECT、INSERT、DELETE操作数据库的审计策略。

```
--创建dev_audit用户。
m_db=# CREATE USER dev_audit PASSWORD '********';

--创建审计策略adt3。
m_db=# CREATE AUDIT POLICY adt3 ACCESS SELECT ON LABEL(adt_lb0), INSERT ON
LABEL(adt_lb0), DELETE FILTER ON ROLES(dev_audit), APP(gsql), IP('10.20.30.40', '127.0.0.0/24');

--删除审计策略adt3。
m_db=# DROP AUDIT POLICY adt3;

--删除dev_audit用户。
m_db=# DROP USER dev_audit;
```

相关链接

ALTER AUDIT POLICY，**DROP AUDIT POLICY**。

4.4.2.8.7 CREATE DATABASE

功能描述

DATABASE与SCHEMA相同，此语句创建一个新的SCHEMA。

注意事项

- 只有拥有CREATEDB权限的用户才可以创建新数据库，系统管理员默认拥有此权限。
- 在创建数据库过程中，出现类似“Permission denied”的错误提示，可能是由于文件系统上数据目录的权限不足。出现类似“No space left on device”的错误提示，可能是由于磁盘满引起的。

语法格式

```
CREATE {DATABASE | SCHEMA} [IF NOT EXISTS] database_name
    [[create_option] [,...]]
create_option: [DEFAULT] {
    CHARACTER SET [=] default_charset
  | CHAR SET [=] default_charset
  | CHARSET [=] default_charset
  | COLLATE [=] default_collation
}
```

参数说明

- **database_name**

  数据库名称。

  取值范围：字符串，要符合**标识符说明**。

- **COLLATE [=] collation_name**

  可选。指定新数据库使用的字符集。例如，通过collate = 'zh_CN.gbk'设定该参数。

  该参数的使用会影响到对字符串的排序顺序（如使用ORDER BY执行，以及在文本列上使用索引的顺序）。默认是使用模板数据库的字符集。

  取值范围：参考**库级字符集和字符序**。

- **{CHAR SET | CHARSET | CHARACTER SET} [=] charset_name**

  可选。指定新数据库使用的字符分类。例如，通过CHARSET = 'zh_CN.gbk'设定该参数。该参数的使用会影响到字符的分类，如大写、小写和数字。默认是使用模板数据库的字符分类。

  取值范围：参考**库级字符集和字符序**。

有关字符编码的一些限制：

- 若区域设置为C（或POSIX），则允许所有的编码类型，但是对于其他的区域设置，字符编码必须和区域设置相同。
- 若字符编码方式是SQL_ASCII，并且修改者为管理员用户时，则字符编码可以和区域设置不相同。
- 编码和区域设置必须匹配模板数据库，除了将template0当作模板。 因为其他数据库可能会包含不匹配指定编码的数据，或者可能包含排序顺序受LC_COLLATE和LC_CTYPE影响的索引。复制这些数据会导致在新数据库中的索引失效。template0是不包含任何会受到影响的数据或者索引。

示例

```
--创建并切换至测试数据库。
m_db=# CREATE DATABASE test1;
m_db=# USE test1;
--创建表a。
m_db=# CREATE TABLE a(id int);
```



```
m_db=# DROP TABLE a;
--删除DATABASE test1。
m_db=# USE public;
m_db=# DROP DATABASE test1;
```

相关链接

ALTER DATABASE，**DROP DATABASE**

优化建议

- **create database**

  事务中不支持创建database。

4.4.2.8.8 CREATE EXTENSION

功能描述

当前不支持用户使用此语法。安装一个扩展。该特性为内部使用，不建议用户使用。

注意事项

- 在使用CREATE EXTENSION载入扩展到数据库中之前，必须先安装好该扩展的支持文件。
- CREATE EXTENSION命令安装一个新的扩展到一个数据库中，必须保证没有同名的扩展已经被安装。
- 安装一个扩展意味着执行一个扩展的脚本文件，这个脚本会创建一个新的SQL实体，例如函数、数据类型、操作符和索引支持的方法。
- 安装扩展需要有和创建他的组件对象相同的权限。对于大多数扩展这意味着需要超户或者数据库所有者的权限，对于后续的权限检查和该扩展脚本所创建的实体，运行CREATE EXTENSION命令的角色将变为扩展的所有者。
- CREATE EXTENSION时如果数据库中存在与EXTENSION内同名的同义词、操作符、目录、函数、存储过程、视图、表这些数据库对象，将会导致CREATE EXTENSION失败。
- 数据库禁止直接创建扩展，因为扩展可能会引起非预期的错误，且在升级后面临不兼容的问题。如果需要创建扩展，需要设置**enable_extension**为true才能够创建。
- CREATE EXTENSION时，如果GUC参数enable_object_special_character为off，且扩展的脚本文件中使用"@extschema@"，那么扩展的支持文件中schema参数的值不能包含["$'\]中任意特殊字符。

语法格式

```
CREATE EXTENSION [ IF NOT EXISTS ] extension_name
[ WITH ] [ SCHEMA schema_name ]
[ VERSION version ]
[ FROM old_version ];
```

参数说明

- **IF NOT EXISTS**

  如果系统已经存在一个同名的扩展，不会报错。这种情况下会给出一个提示。请注意该参数不保证系统存在的扩展和现在脚本创建的扩展相同。

- **extension_name**

  将被安装扩展的名字，数据库将使用文件SHAREDIR/extension/extension_name.control中的详细信息创建扩展。

- **schema_name**

  扩展的实例被安装在该模式下，扩展的内容可以被重新安装。指定的模式必须已经存在，如果没有指定，扩展的控制文件也不指定一个模式，这样将使用默认模式。

  ⚠ 注意

  扩展不认为它在任何模式里面：扩展在一个数据库范围内的名字是不受限制的，但是一个扩展的实例是属于一个模式的。

- **version**

  安装扩展的版本，可以写为一个标识符或者字符串.默认的版本在扩展的控制文件中指定。

- **old_version**

  当你想升级安装"old style" 模块中没有的内容时,你必须指定FROM old_version。这个选项使CREATE EXTENSION 运行一个安装脚本将新的内容安装到扩展中，而不是创建一个新的实体。注意：SCHEMA指定了包括这些已存在实体的模式。

示例

在当前数据库安装扩展。例如安装security_plugin：

```
m_db=# CREATE EXTENSION IF NOT EXISTS security_plugin;
m_db=# DROP EXTENSION security_plugin;
```

相关链接

ALTER EXTENSION，**DROP EXTENSION**

4.4.2.8.9 CREATE FUNCTION

功能描述

创建一个函数。

说明

内部使用功能，不支持用户使用。

注意事项

- 如果创建函数时参数或返回值带有精度，不进行精度检测。
- 创建函数时，函数定义中对表对象的操作建议都显式指定模式，否则可能会导致函数执行异常。
- 如果函数参数中带有出参，想要出参生效，必须打开guc参数 set behavior_compat_options = 'proc_outparam_override'； SELECT、CALL调用函数时，必须要在出参位置提供实参进行调用，否则函数调用失败。

- 兼容PostgreSQL风格的函数支持重载。
- 不能创建仅形参名字不同（函数名和参数列表类型都一样）的重载函数。
- 不支持形参仅在自定义ref cursor类型和sys_refcursor类型不同的重载。
- 不支持仅返回的数据类型不同的函数重载。
- 不支持仅默认值不同的函数重载。
- 重载的函数在调用时变量需要明确具体的类型。
- 在函数内部使用未声明的变量，函数被调用时会报错。
- SELECT调用可以指定不同参数来进行同名函数调用。
- 在创建function时，不能在avg函数外面嵌套其他agg函数，或者其他系统函数。
- 新创建的函数默认会给PUBLIC授予执行权限。用户默认继承PUBLIC角色权限，因此其他用户也会有函数的执行权限并可以查看函数的定义，另外执行函数时还需要具备函数所在schema的USAGE权限。用户在创建函数时可以选择收回PUBLIC默认执行权限，然后根据需要将执行权限授予其他用户，为了避免出现新函数能被所有人访问的时间窗口，应在一个事务中创建函数并且设置函数执行权限。开启数据库对象隔离属性后，普通用户只能查看有权限执行的函数定义。
- 在函数内部调用其它无参数的函数时，可以省略括号，直接使用函数名进行调用。
- 在函数内部调用其他有出参的函数，如果在赋值表达式中调用时，需要打开guc参数 set behavior_compat_options = 'proc_outparam_override' ，并提前定义与出参类型相同的变量，然后将变量作为出参调用带有出参的其他函数，出参才能生效。否则，被调函数的出参会被忽略。
- 被授予CREATE ANY FUNCTION权限的用户，可以在用户模式下创建/替换函数。
- 函数默认为AUTHID CURRENT_USER权限，如果想将默认行为改为AUTHID DEFINER权限，需要设置guc参数 behavior_compat_options='plsql_security_definer'。
- 对于PL/SQL函数，打开参数 behavior_compat_options='proc_outparam_override'后，out/inout的行为会改变，函数中同时返回return和out/inout，而参数打开前仅返回return，见**示例**。
- 对于PL/SQL函数，打开参数 behavior_compat_options='proc_outparam_override'后，有以下限制：
  - 如果同一Schema和package中已存在带有out/inout参数函数，不能再次创建带有out/inout参数的同名函数。
  - 部分场景不支持函数参与表达式（与参数打开前相比），如CALL function等。
  - 不支持调用无RETURN的函数、PERFORM function调用。
  - 打开GUC参数proc_outparam_override后，函数返回值为SETOF类型时，out出参不会生效。
- 创建函数时，不支持使用函数自身作为入参默认值。
- 带OUT模式参数的函数不能在SQL语句中被调用。
- 带OUT模式参数的函数不能被SELECT INTO语法调用。
- 带OUT模式参数的函数不支持嵌套调用。

  比如：

  ```
  b := func(a,func(c,1));
  ```

  建议改为：


```
tmp := func(c,1); b := func(a,tmp);
```

- 在创建函数时，不会检查函数内返回值的类型。
- 如果将定义者权限的函数创建到其他用户Schema下，则会以其他用户的权限执行该函数，有越权风险，请谨慎使用。
- 在运维管理员的Schema下，只允许初始用户，系统管理员和Schema的属主自己创建对象，不允许其他用户在运维管理员的Schema下创建对象或修改对象的Schema为运维管理员的Schema。
- 在表达式中使用out参数作为出参时，如下情况不会生效，例如：使用execute immediate sqlv using func语法执行函数、使用select func into语法执行函数、使用insert、update等DML语句执行、使用select where a=func()；带out出参的函数，作为入参时，fun（func（out b），a），out出参b未生效等。
- 在FUNCTION的RETURN语句中，返回复合类型的构造器的调用时，实际返回类型与定义返回类型不一致时，可以隐式转换为定义返回类型时则对结果进行类型转换，支持跨schema调用，如：RETURN schema.record；不支持跨database调用，如：RETURN package.schema.record；在FUNCTION的RETURN语句中，返回FUNCTION的调用时，不支持在运算操作的表达式中带OUT参数的FUNCTION，如RETURN func(c out) + 1。
- 函数复杂调用，如：func(x).a，函数调用返回复合类型，支持跨schema调用，不支持通过database.schema.package.func(x).b的方式调用。
- 函数中存在通过guc参数控制特性的语法、函数等，如果在会话内更改相关guc参数，可能存在与预期结果不符，修改参数后，调用函数可能会维持修改前的行为，请谨慎变更guc参数。

语法格式

```
CREATE OR REPLACE FUNCTION function_name
   ( [ { argname [ argmode ] argtype [ { DEFAULT | := | = } expression ]} [, ...] ] )
   [ RETURNS rettype | RETURNS TABLE ( { column_name column_type } [, ...] ) ]
   LANGUAGE lang_name
   [
     {IMMUTABLE | STABLE | VOLATILE}
      | {SHIPPABLE | NOT SHIPPABLE}
      | [ NOT ] LEAKPROOF
      | STRICT
      | {AUTHID DEFINER | AUTHID CURRENT_USER}
      | not fenced
      | COST execution_cost
      | ROWS result_rows
    ][...]
   {
      AS 'definition'
   };
```

参数说明

- **function_name**

  要创建的函数名称（可以用模式修饰）。

  取值范围：字符串，要符合**标识符说明**。且最多为63个字符。若超过63个字符，数据库会截断并保留前63个字符当做函数名称。

  说明

  建议不要创建和系统函数重名的函数，否则调用时需要指定函数的Schema。

- **argname**

  函数参数的名称。

取值范围：字符串，要符合**标识符说明**。且最多为63个字符。若超过63个字符，数据库会截断并保留前63个字符当做函数参数名称。

- **argmode**

  函数参数的模式。

  取值范围：IN，OUT，INOUT或VARIADIC。缺省值是IN。只有OUT模式的参数后面能跟VARIADIC。并且OUT和INOUT模式的参数不能用在RETURNS TABLE的函数定义中。

  说明

  VARIADIC用于声明数组类型的参数。

- **argtype**

  函数参数的类型。可以使用%TYPE或%ROWTYPE间接引用变量或表的类型。

- **expression**

  参数的默认表达式。

  说明

  - 在参数a_format_version值为10c和a_format_dev_version值为s2的情况下，函数参数为INOUT模式时不支持默认表达式。
  - 推荐使用方式：将所有默认值参数定义在所有非默认值参数后。
  - 调用带有默认参数的函数时，入参从左往右排入函数，如果有非默认参数的入参缺失则报错。
  - 打开 proc_uncheck_default_param 参数，调用带有默认参数的函数时，入参从左往右排入函数，允许缺省默认参数个入参，如果有非默认参数的入参缺失，则会用错位的默认值填充该参数。
  - 在参数a_format_version值为10c、a_format_dev_version值为s1和关闭proc_outparam_override，函数参数同时包括out出参和default时，默认值不可缺省。

- **rettype**

  函数返回值的数据类型。与argtype相同，同样可以使用%TYPE或%ROWTYPE间接引用类型。

  如果存在OUT或INOUT参数，可以省略RETURNS子句。如果没有省略，则该子句必须和输出参数所表示的结果类型一致，如果有多个输出参数，则为RECORD，否则与单个输出参数的类型相同。

  SETOF修饰词表示该函数将返回一个集合，而不是单独一项。

  说明

  PACKAGE外FUNCTION argtype和rettype中%TYPE不支持引用PACKAGE变量的类型。

- **column_name**

  字段名称。

- **column_type**

  字段类型。

- **definition**

  一个定义函数的字符串常量，含义取决于语言。它可以是一个内部函数名称、一个指向某个目标文件的路径、一个SQL查询、一个过程语言文本。

- **LANGUAGE lang_name**

用以实现函数的语言的名称。可以是SQL，internal，或者是用户定义的过程语言名称。为了保证向下兼容，该名称可以用单引号（包围）。若采用单引号，则引号内必须为大写。

说明

- internel函数在定义时，如果AS指定为内部系统函数，则新创建函数的参数类型，参数个数，与返回值类型需要与内部系统函数保持一致，且需要有执行此内部系统函数的权限。
- internal函数只支持拥有sysadmin权限的用户创建。

- **IMMUTABLE**

  表示该函数在给出同样的参数值时总是返回同样的结果。

- **STABLE**

  表示该函数不能修改数据库，对相同参数值，在同一次表扫描里，该函数的返回值不变，但是返回值可能在不同SQL语句之间变化。

- **VOLATILE**

  表示该函数值可以在一次表扫描内改变，因此不会做任何优化。

- **SHIPPABLE|NOT SHIPPABLE**

  表示该函数是否可以下推执行。预留接口，不推荐使用。

- **NOT FENCED**

  声明用户定义的C函数是否在非保护模式下执行。预留接口，不推荐使用。

- **LEAKPROOF**

  指出该函数的参数只包括返回值。LEAKPROOF只能由系统管理员设置。

- **STRICT**

  STRICT用于指定如果函数的某个参数是NULL，此函数总是返回NULL。如果声明了这个参数，当有NULL值参数时该函数不会被执行；而只是自动返回一个NULL结果。

- **AUTHID CURRENT_USER**

  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。

- **AUTHID DEFINER**

  声明该函数将以创建它的用户的权限执行。

- **COST execution_cost**

  用来估计函数的执行成本。

  execution_cost以cpu_operator_cost为单位。

  取值范围：>=0的数值

- **ROWS result_rows**

  估计函数返回的行数。用于函数返回的是一个集合。

  取值范围：>=0的数值，默认值是1000行。

须知

当在函数体中进行创建用户、修改密码或加解密等涉及密码或密钥相关操作时，系统表及日志中会记录密码或密钥的明文信息。为防止敏感信息泄露，不建议用户在函数体中进行涉及密码或密钥等敏感信息的相关操作。

示例

```
--定义函数为SQL查询。
m_db=# CREATE OR REPLACE FUNCTION func_add_sql(integer, integer) RETURNS bigint
    AS 'select $1 + $2;'
    LANGUAGE SQL
    IMMUTABLE
    SHIPPABLE
    STRICT
    AUTHID DEFINER
    not fenced
    COST 10000;

--利用参数名用plpgsql自增一个整数。
m_db=# CREATE OR REPLACE FUNCTION func_increment_plsql(i integer) RETURNS integer AS $$
    BEGIN
        RETURN i + 1;
    END;
    $$ LANGUAGE plpgsql;

--创建internal函数。
m_db=# CREATE OR REPLACE FUNCTION func_test(name) RETURNS text LANGUAGE INTERNAL AS
'series_internal';

--创建同名函数。
m_db=# CREATE OR REPLACE FUNCTION func_add(int) RETURNS int AS $$
    BEGIN
        RETURN $1+10;
    END;
    $$ LANGUAGE PLPGSQL;

m_db=# CREATE OR REPLACE FUNCTION func_add(int,int) RETURNS int AS $$
    BEGIN
        RETURN $1+$2;
    END;
    $$ LANGUAGE PLPGSQL;

--调用函数。
m_db=# SELECT func_add_sql(3, 7);
 func_add_sql
--------------
           10
(1 row)

m_db=# SELECT func_increment_plsql(5);
 func_increment_plsql
----------------------
                    6
(1 row)

m_db=# SELECT func_add(4);
 func_add
----------
       14
(1 row)

m_db=# SELECT func_add(2,7);
 func_add
----------
        9
(1 row)

--删除函数。
m_db=# DROP FUNCTION func_add_sql;
m_db=# DROP FUNCTION IF EXISTS func_increment_plsql;
m_db=# DROP FUNCTION func_test() CASCADE;
m_db=# DROP FUNCTION func_add(int);
m_db=# DROP FUNCTION IF EXISTS func_add(int,int) RESTRICT;
```


相关链接

DROP FUNCTION

4.4.2.8.10 CREATE GROUP

功能描述

创建一个新用户组。

注意事项

CREATE GROUP是CREATE ROLE的别名，非SQL标准语法，不推荐使用，建议用户直接使用CREATE ROLE替代。

语法格式

```
CREATE GROUP group_name [ [ WITH ] option [ ... ] ] { PASSWORD | IDENTIFIED BY } { 'password'
[ EXPIRED ] | DISABLE };
```

其中可选项option子句语法为：

```
{SYSADMIN | NOSYSADMIN}
    | {MONADMIN | NOMONADMIN}
    | {OPRADMIN | NOOPRADMIN}
    | {POLADMIN | NOPOLADMIN}
    | {AUDITADMIN | NOAUDITADMIN}
    | {CREATEDB | NOCREATEDB}
    | {USEFT | NOUSEFT}
    | {CREATEROLE | NOCREATEROLE}
    | {INHERIT | NOINHERIT}
    | {LOGIN | NOLOGIN}
    | {REPLICATION | NOREPLICATION}
    | {PERSISTENCE | NOPERSISTENCE}
    | CONNECTION LIMIT connlimit
    | VALID BEGIN 'timestamp'
    | VALID UNTIL 'timestamp'
    | USER GROUP 'groupuser'
    | NODE GROUP logic_cluster_name
    | IN ROLE role_name [, ...]
    | IN GROUP role_name [, ...]
    | ROLE role_name [, ...]
    | ADMIN role_name [, ...]
    | USER role_name [, ...]
    | DEFAULT TABLESPACE tablespace_name
    | PROFILE DEFAULT
    | PROFILE profile_name
    | PGUSER
```

参数说明

请参考CREATE ROLE的**参数说明**。

示例

```
--创建用户组。
m_db=# CREATE GROUP super_users WITH PASSWORD "********";

--创建用户。
m_db=# CREATE ROLE lche WITH PASSWORD "********";

--创建用户。
m_db=# CREATE ROLE jim WITH PASSWORD "********";
```



```
--向用户组中添加用户。
m_db=# ALTER GROUP super_users ADD USER lche, jim;

--从用户组中删除用户。
m_db=# ALTER GROUP super_users DROP USER jim;

--修改用户组的名称。
m_db=# ALTER GROUP super_users RENAME TO normal_users;

--删除用户。
m_db=# DROP ROLE lche, jim;

--删除用户组。
m_db=# DROP GROUP normal_users;
```

相关链接

ALTER GROUP，**DROP GROUP**，**CREATE ROLE**

4.4.2.8.11 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，分区表上创建索引会根据如下规则进行判断：如果创建索引时申明了GLOBAL/LOCAL关键字，则创建对应类型的索引;在未申明的情况下，默认创建GLOBAL索引，但以下情况例外：如果是unique索引且包含全部分区键会创建LOCAL索引。

注意事项

- 索引自身也占用存储空间、消耗计算资源，创建过多的索引将对数据库性能造成负面影响（尤其影响数据导入的性能，建议在数据导入后再建索引）。因此，仅在必要时创建索引。
- 索引定义里的所有函数和操作符都必须是immutable类型的，即它们的结果必须只能依赖于它们的输入参数，而不受任何外部的影响（如另外一个表的内容或者当前时间）。这个限制可以确保该索引的行为是定义良好的。要在一个索引上或WHERE中使用用户定义函数，请把它标记为immutable类型函数。
- 分区表索引分为LOCAL索引与GLOBAL索引，LOCAL索引与某个具体分区绑定，而GLOBAL索引则对应整个分区表。
- 被授予CREATE ANY INDEX权限的用户，可以在public模式和用户模式下创建索引。
- 禁止其他用户在初始用户的表上创建包含用户自定义函数的表达式索引。

- 如果表达式索引中调用的是用户自定义函数，按照函数创建者权限执行表达式索引函数。
- 不支持XML类型数据作为普通索引、UNIQUE索引、GLOBAL索引、LOCAL索引。
- 在线创建索引的类型只支持BTREE索引和UBTREE索引，不支持HASH索引。索引创建形式只支持非分区表普通索引及分区表GLOBAL索引、LOCAL索引，不支持在线索引字段增删改、PCR UBTREE索引、二级分区与GSI。在线并行创建索引只支持Astore及Ustore的普通索引、GLOBAL索引、LOCAL索引。

说明

在未开启精度传递参数（m_format_behavior_compat_options不开启enable_precision_decimal选项）情况下创建的索引在开启精度传递参数后会失效，如果打开精度传递需要重新创建索引。

语法格式

- 在表上创建索引。

```
CREATE [ UNIQUE ] INDEX [ [schema_name.] index_name ] [index_type]
ON table_name ( key_part,... )
    [ WITH ( {storage_parameter = value} [, ... ] ) ]
    [ TABLESPACE tablespace_name ]
    [index_option]
    [algorithm_option | lock_option] ...

key_part:
    { { column_name [ ( length ) ] | ( expression ) } [ COLLATE collation ] [ opclass ] [ ASC | DESC ]
[ NULLS { FIRST | LAST } ] }[, ...]

index_option: {
 index_type
  | COMMENT 'string'
}

index_type:
    USING {BTREE | UBTREE}

algorithm_option:
    ALGORITHM [=] {DEFAULT | INPLACE | COPY}

lock_option:
    LOCK [=] {DEFAULT | NONE | SHARED | EXCLUSIVE};
```

- 在分区表上创建索引。

```
CREATE [ UNIQUE ] INDEX [ [schema_name.] index_name ][index_type]
ON table_name
    [ LOCAL
        [ ( { PARTITION index_partition_name [ TABLESPACE index_partition_tablespace ]
                [ ( [SUBPARTITION index_subpartition_name] [ TABLESPACE index_partition_tablespace ]
                    [, ...] ) ]
            [, ...] } ) ]
       | GLOBAL ]
    [ WITH ( { storage_parameter = value } [, ...] ) ]
    [ TABLESPACE tablespace_name ];
    [index_option]
    [algorithm_option | lock_option] ...

index_option: {
 index_type
  | COMMENT 'string'
}

index_type:
    USING {BTREE | UBTREE}

algorithm_option:
```



```
    ALGORITHM [=] {DEFAULT | INPLACE | COPY}

lock_option:
    LOCK [=] {DEFAULT | NONE | SHARED | EXCLUSIVE};
```

参数说明

- **UNIQUE**

  创建唯一性索引，每次添加数据时检测表中是否有重复值。如果插入或更新的值会引起重复的记录时，将导致一个错误。

  目前只有B-tree及UBtree索引支持唯一索引。

- **schema_name**

  模式的名称。

  取值范围：已存在模式名。

- **index_name**

  要创建的索引名，索引的模式与表相同。

  取值范围：字符串，要符合**标识符说明**。

- **table_name**

  需要为其创建索引的表的名称，可以用模式修饰。

  取值范围：已存在的表名。

- **USING method**

  指定创建索引的方法。

  取值范围：

  - BTREE：BTREE索引使用一种类似于B+树的结构来存储数据的键值，通过这种结构能够快速地查找索引。BTREE适合支持比较查询以及查询范围。
  - UBTREE：仅供ustore表使用的多版本btree索引，索引页面上包含事务信息，并能自主回收页面。ubtree索引默认开启insertpt功能。

  行存表（ASTORE存储引擎）支持的索引类型：BTREE（行存表缺省值）。行存表（USTORE存储引擎）支持的索引类型：BTREE（实际建立为UBTREE）。

  须知

  BTREE与表的存储类型astore/ustore强相关，在创建索引时指定索引类型与主表不对应时会自动进行转换。在指定创建索引时，ASTORE创建的为BTREE，而USTORE创建创建的为UBTREE。

- **column_name**

  表中需要创建索引的列的名称（字段名）。

  如果索引方式支持多字段索引，可以声明多个字段。全局索引最多可以声明31个字段，其他索引最多可以声明32个字段。

- **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。含有前缀键字段的部分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。
- 前缀键属于一种特殊的表达式键，部分未说明的约束和限制，与表达式键一致，请参考表达式索引的说明。

- **expression**

  创建一个基于该表的一个或多个字段的表达式索引，通常必须写在圆括弧中。

  表达式索引可用于获取对基本数据的某种变形的快速访问。比如，一个在upper(col)上的函数索引将允许WHERE upper(col) = 'JIM'子句使用索引。

  在创建表达式索引时，如果表达式中包含IS NULL子句，则这种索引是无效的。

- **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。

- **opclass**

  操作符类的名称。对于索引的每一列可以指定一个操作符类，操作符类标识了索引那一列的使用的操作符。例如一个btree索引在一个四字节整数上可以使用int4_ops；这个操作符类包括四字节整数的比较函数。实际上对于列上的数据类型默认的操作符类是足够用的。操作符类主要用于一些有多种排序的数据。例如，用户想按照绝对值或者实数部分排序一个复数。能通过定义两个操作符类然后在建立索引时选择合适的类。另外，如果包含字符串类型（varchar、varchar2、text等）的索引的COLLATE的值不是C或者POSIX，但希望索引能够支持前缀匹配，则需要指定varchar_pattern_ops选项。

- **LOCAL**

  指定创建的分区索引为LOCAL索引。

- **GLOBAL**

  指定创建的分区索引为GLOBAL索引，当不指定LOCAL、GLOBAL关键字时，默认创建GLOBAL索引。

- **NULLS FIRST**

  指定空值在排序中排在非空值之前，当指定DESC排序时，本选项为默认的。

- **NULLS LAST**

  指定空值在排序中排在非空值之后，未指定DESC排序时，本选项为默认的。

- **PARTITION子句**

  指定一级分区分区信息语法如下：

  ```
  PARTITION index_partition_name [ TABLESPACE index_partition_tablespace ]
  ```

  指定二级分区分区信息语法如下：

  ```
  SUBPARTITION index_subpartition_name [ TABLESPACE index_partition_tablespace ]
  ```

  - **PARTITION index_partition_name**

    索引分区的名称。

    取值范围：字符串，要符合**标识符说明**。

- **SUBPARTITION index_subpartition_name**

  索引二级分区的名称。

  取值范围：字符串，要符合**标识符说明**。

- **TABLESPACE index_partition_tablespace**

  索引分区的表空间。

  取值范围：如果没有声明，将使用分区表索引的表空间index_tablespace。

  注：需要在非M-Compatibility数据库中创建或删除tablespace。

- **WITH ({storage_parameter = value} [, ...])**

  指定索引方法的存储参数。

  取值范围：

  只有UBTREE索引支持INDEXSPLIT参数。只有非分区表的BTREE索引支持DEDUPLICATION参数。只有UBTREE索引支持INDEX_TXNTYPE参数。

  - STORAGE_TYPE

    表示索引所在的表的存储引擎类型，当索引指定的storage_type与索引类型冲突时，会自动修改为正确的存储类型。该参数设置成功后无法再次进行修改。

    取值范围：

    - USTORE，表示索引所在的表为Inplace-Update存储引擎。
    - ASTORE，表示索引所在的表为Append-Only存储引擎。

    默认值：UStore表创建的索引默认为USTORE，AStore表创建的索引默认为ASTORE。

  - FILLFACTOR

    索引的填充因子（fillfactor）是一个介于10和100之间的百分数。对于大并发插入且键值范围比较密集的场景，插入时同一个索引页面的竞争比较大，选择较小的填充因子更加合适。

    取值范围：10~100

  - INDEXSPLIT

    UBTREE索引选择采取哪种分裂策略。其中DEFAULT策略指的是与BTREE相同的分裂策略。INSERTPT策略能在某些场景下显著降低索引空间占用。

    取值范围：INSERTPT, DEFAULT

    默认值：INSERTPT

  - INDEX_TXNTYPE

    UBTree索引类型（只有UBTree索引支持INDEX_TXNTYPE），当该值取值为PCR时，可支持通过UBTree进行闪回查询。PCR版本UBTree索引当前不支持在线创建索引、全局二级索引、极致RTO回放和备机读的功能。当不指定index_txntype时，具体创建哪种类型的索引通过GUC参数index_txntype进行控制。INDEX_TXNTYPE不支持ALTER INDEX INDEX_NAME SET (INDEX_TXNTYPE=PCR或RCR)进行修改。

    类型：字符串（不区分大小写）

    取值范围： RCR, PCR

    默认值：RCR

    举例：

    ```
    CREATE UNIQUE INDEX t2_b_pkey ON t(b) WITH(index_txntype='pcr');
    ```


- ACTIVE_PAGES

  表示索引的页面数量，可能比实际的物理文件页面少，可以用于优化器调优。目前只对ustore的分区表local索引生效，且会被vacuum、analyze更新（包括auto vacuum）。不建议用户手动设置该参数。

- DEDUPLICATION

  索引参数，设置索引是否对键值重复的元组进行去重压缩。在重复键值的索引较多时，开启参数可以有效降低索引占用空间。对主键索引和唯一索引不生效。非唯一索引且索引键值重复度很低或者唯一的场景，开启参数会使索引插入性能小幅度劣化。暂不支持分区表的local/global索引。

  取值范围：布尔值，默认取GUC参数中enable_default_index_deduplication的值（默认为off）。

- stat_state

  标识该索引的统计信息是否被锁定，如果被锁定了，该索引的统计信息无法更新。

  取值范围：locked、unlock。

  默认值：unlock。

- LPI_PARALLEL_METHOD

  索引参数，设置分区表LOCAL索引并行创建的方式。PAGE为页面级并行创建索引，开启多个子线程执行数据的扫描和排序，每个子线程一次处理一个数据页面，扫描排序后，在主线程串行合并排序结果并将元组插入到索引中；PARTITION为分区级并行创建索引，开启多个子线程，每个子线程负责一个分区中数据的扫描、排序、索引插入；AUTO会根据分区表统计信息来预估页面级和分区级两种并行创建索引方式的代价，并选择代价较小的并行创建方式（统计信息可能会和实际数据情况有误差而导致计算结果不准确）。当分区表数据在各分区分布均匀时，建议指定该参数为PARTITION。该参数仅支持Astore分区表的BTREE LOCAL索引，不支持分区表GLOBAL索引、非分区表索引、段页式表索引、在线创建索引。

  类型：字符串（不区分大小写）。

  取值范围：PAGE、PARTITION、AUTO。

  默认值：PAGE。

  举例：

  ```
  CREATE INDEX idx ON tbl(col) WITH (lpi_parallel_method = 'partition');
  ```

- **TABLESPACE tablespace_name**

  指定索引的表空间，如果没有声明则使用默认的表空间。

  取值范围：已存在的表空间名。

  注：需要在非M-Compatibility数据库中创建或删除tablespace。

示例

- **普通索引**

```
--创建tbl_test1表。
m_db=# CREATE TABLE tbl_test1(
    id int,           --用户id
    name varchar(50), --用户姓名
    postcode char(6)  --邮编
);

--为表tbl_test1创建索引idx_test1指定表空间。
m_db=# CREATE INDEX idx_test1 ON tbl_test1(name) TABLESPACE pg_default;
```



```
--查询索引idx_test1信息。
m_db=# SELECT indexname,tablename,tablespace FROM pg_indexes WHERE indexname = 'idx_test1';
 indexname | tablename | tablespace
-----------+-----------+------------
 idx_test1 | tbl_test1 |
(1 row)

--删除索引。
m_db=# DROP INDEX idx_test1;
```

- **唯一索引**

```
--为表tbl_test1创建唯一索引idx_test2。
m_db=# CREATE UNIQUE INDEX idx_test2 ON tbl_test1(id);

--查询索引信息。
m_db=# \d tbl_test1
          Table "public.tbl_test1"
  Column  |         Type          | Modifiers
----------+-----------------------+-----------
 id       | integer               |
 name     | character varying(50) |
 postcode | character(6)          |
Indexes:
    "idx_test2" UNIQUE, btree (id) TABLESPACE pg_default

--删除索引。
m_db=# DROP INDEX idx_test2;
```

- **分区索引**

```
--建表。
m_db=# CREATE TABLE student(id int, name varchar(20)) PARTITION BY RANGE (id) (
    PARTITION p1 VALUES LESS THAN (200),
    PARTITION pmax VALUES LESS THAN (MAXVALUE)
);

--创建LOCAL分区索引不指定索引分区的名称。
m_db=# CREATE INDEX idx_student1 ON student(id) LOCAL;

--查看索引分区信息，发现LOCAL索引分区数和表的分区数一致。
m_db=# SELECT relname FROM pg_partition WHERE parentid = 'idx_student1'::regclass;
   relname
-------------
 p1_id_idx
 pmax_id_idx
(2 rows)

--删除LOCAL分区索引。
m_db=# DROP INDEX idx_student1;

--创建GLOBAL索引。
m_db=# CREATE INDEX idx_student2 ON student(name) GLOBAL;

--查看索引分区信息，发现GLOBAL索引分区数和表的分区数不一致。
m_db=# SELECT relname FROM pg_partition WHERE parentid = 'idx_student2'::regclass;
 relname
---------
(0 rows)

--删除GLOBAL分区索引。
m_db=# DROP INDEX idx_student2;

--删除表。
m_db=# DROP TABLE student;
```

相关链接

ALTER INDEX，**DROP INDEX**

优化建议

- create index

  建议仅在匹配如下条件之一时创建索引：

  - 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。

  约束限制：

  - 普通表的索引支持最大列数为32列；分区表的GLOBAL索引支持最大列数为31列。
  - 单个索引大小不能超过索引页面大小（8k），其中B-tree、UBtree不能超过页面大小的三分之一。
  - 分区表创建GLOBAL索引时，存在以下约束条件：
    - 不支持表达式索引。
    - 仅支持B-tree索引（ASTORE），USTORE下为UBTREE。
  - 在顺序不同的索引列上，支持同时创建GLOBAL和LOCAL索引。顺序相同时，则不支持同时创建。
  - 如果alter语句不带有UPDATE GLOBAL INDEX，那么原有的GLOBAL索引将失效，查询时将使用其他索引进行查询；如果alter语句带有UPDATE GLOBAL INDEX，原有的GLOBAL索引仍然有效，并且索引功能正确。

4.4.2.8.12 CREATE RESOURCE LABEL

功能描述

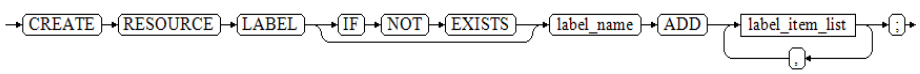
CREATE RESOURCE LABEL语句用于创建资源标签。

注意事项

只有POLADMIN、SYSADMIN或初始用户能正常执行此操作。

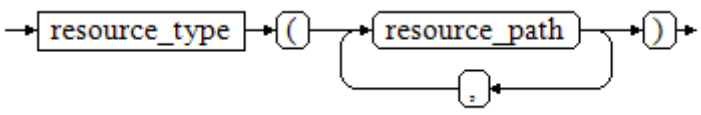
语法格式

```
CREATE RESOURCE LABEL [IF NOT EXISTS] label_name ADD label_item_list[, ...];
```



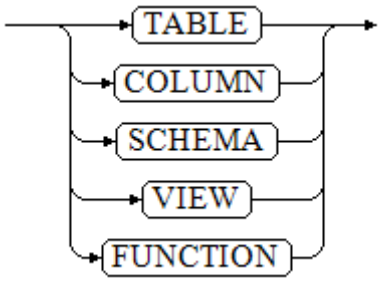
- label_item_list：

  ```
  resource_type(resource_path[, ...])
  ```



- resource_type：

  ```
  { TABLE | COLUMN | SCHEMA | VIEW | FUNCTION }
  ```

参数说明

- **IF NOT EXISTS**

  如果已经存在相同名称的资源标签，不会抛出错误，而是发出一个通知，告知此资源标签已存在。

- **label_name**

  资源标签名称，创建时要求不能与已有标签重名。

  取值范围：字符串，要符合**标识符命名规范**，且最大长度不超过63个字符。若超过63个字符，数据库会截断并保留前63个字符当作资源标签名称。当资源标签名称中包含大写字母时数据库会自动转换为小写字母，如果需要创建包含大写字母的资源标签名称则需要使用双引号括起来。

  说明

  标识符需要为小写字母（a-z）、大写字母(A-Z)、下划线（_）、数字（0~9）或美元符号（$），且必须以字母或下划线开头。

- **resource_type**

  指的是要标记的数据库资源的类型。

  取值范围：表（TABLE）、列（COLUMN）、模式（SCHEMA）、视图（VIEW）、函数（FUNCTION）。

- **resource_path**

  指的是描述具体的数据库资源的路径。

示例

```
--创建一个表tb_for_label。
m_db=# CREATE TABLE tb_for_label(col1 text, col2 text, col3 text);

--基于表创建资源标签。
m_db=# CREATE RESOURCE LABEL IF NOT EXISTS table_label add TABLE(public.tb_for_label);

--再次创建已存在的表资源标签，对比加参数IF NOT EXISTS与不加IF NOT EXISTS参数的区别。
m_db=# CREATE RESOURCE LABEL IF NOT EXISTS table_label add TABLE(public.tb_for_label);
NOTICE:  table_label label already defined, skipping
CREATE RESOURCE LABEL
m_db=# CREATE RESOURCE LABEL table_label add TABLE(public.tb_for_label);
ERROR:  table_label label already defined

--基于列创建资源标签。
m_db=# CREATE RESOURCE LABEL IF NOT EXISTS column_label add COLUMN(public.tb_for_label.col1);

--创建一个模式schema_for_label。
m_db=# CREATE SCHEMA schema_for_label;

--基于模式创建资源标签。
m_db=# CREATE RESOURCE LABEL IF NOT EXISTS schema_label add SCHEMA(schema_for_label);
```



```
--创建一个视图view_for_label。
m_db=# CREATE VIEW view_for_label AS SELECT 1;

--基于视图创建资源标签。
m_db=# CREATE RESOURCE LABEL IF NOT EXISTS view_label add VIEW(view_for_label);

--创建一个函数func_for_label。
m_db=# CREATE FUNCTION func_for_label RETURNS TEXT AS $$ SELECT col1 FROM tb_for_label; $$
LANGUAGE SQL;

--基于函数创建资源标签。
m_db=# CREATE RESOURCE LABEL IF NOT EXISTS func_label add FUNCTION(func_for_label);

--删除表资源标签table_label。
m_db=# DROP RESOURCE LABEL IF EXISTS table_label;

--删除列资源资源标签column_label。
m_db=# DROP RESOURCE LABEL IF EXISTS column_label;

--删除函数资源标签func_for_label。
m_db=# DROP FUNCTION func_for_label;

--删除视图资源标签view_for_label。
m_db=# DROP VIEW view_for_label;

--删除模式资源标签schema_for_label。
m_db=# DROP SCHEMA schema_for_label;

--删除表tb_for_label。
m_db=# DROP TABLE tb_for_label;
```

相关链接

ALTER RESOURCE LABEL，**DROP RESOURCE LABEL**。

4.4.2.8.13 CREATE ROLE

功能描述

创建角色。

角色是拥有数据库对象和权限的实体。在不同的环境中角色可以认为是一个用户，一个组或者兼顾两者。

注意事项

- 在数据库中添加一个新角色，角色无登录权限。
- 创建角色的用户必须具备CREATE ROLE的权限或者是系统管理员。

语法格式

```
CREATE ROLE role_name [ [ WITH ] option [ ... ] ] { PASSWORD | IDENTIFIED BY } { 'password' [EXPIRED] |
DISABLE };
```

其中角色信息设置子句option语法为：

```
{SYSADMIN | NOSYSADMIN}
    | {MONADMIN | NOMONADMIN}
    | {OPRADMIN | NOOPRADMIN}
    | {POLADMIN | NOPOLADMIN}
    | {AUDITADMIN | NOAUDITADMIN}
    | {CREATEDB | NOCREATEDB}
    | {USEFT | NOUSEFT}
    | {CREATEROLE | NOCREATEROLE}
```



```
| {INHERIT | NOINHERIT}
| {LOGIN | NOLOGIN}
| {REPLICATION | NOREPLICATION}
| {PERSISTENCE | NOPERSISTENCE}
| CONNECTION LIMIT connlimit
| VALID BEGIN 'timestamp'
| VALID UNTIL 'timestamp'
| USER GROUP 'groupuser'
| NODE GROUP logic_cluster_name
| IN ROLE role_name [, ...]
| IN GROUP role_name [, ...]
| ROLE role_name [, ...]
| ADMIN role_name [, ...]
| USER role_name [, ...]
| DEFAULT TABLESPACE tablespace_name
| PROFILE DEFAULT
| PROFILE profile_name
| PGUSER
```

参数说明

- **role_name**

  角色名称。

  取值范围：字符串，要符合**标识符说明**，且最多为63个字符。若超过63个字符，数据库会截断并保留前63个字符当做角色名称。在创建角色时，数据库的时候会给出提示信息。角色名支持使用反引号括起来，当sql_mode设置为ANSI_QUOTES，也可以用双引号括起来。不用反引号、双引号括起来的角色名会被转换为小写。

  说明

  标识符需要为字母、下划线、数字（0-9）或美元符号（$），且必须以字母（a-z）或下划线（_）开头。

- **password**

  登录密码。

  密码规则如下：

  - 密码默认不少于8个字符。
  - 不能与用户名及用户名倒序相同。
  - 至少包含大写字母（A-Z），小写字母（a-z），数字（0-9），非字母数字字符（限定为~!@#$%^&*()-_=+\|[{}];:,<.>/?）四类字符中的三类字符。
  - 密码也可以是符合格式要求的密文字符串，这种情况主要用于用户数据导入场景，不推荐用户直接使用。如果直接使用密文密码，用户需要知道密文密码对应的明文，并且保证密码复杂度，数据库不会校验密文密码复杂度，直接使用密文密码的安全性由用户保证。
  - 创建角色时，应当使用单引号将用户密码括起来。

  取值范围：不为空的字符串。

- **EXPIRED**

  在创建用户时可指定EXPIRED参数，即创建密码失效用户，该用户不允许执行简单查询和扩展查询。只有在修改自身密码后才可正常执行语句。

- **DISABLE**

  默认情况下，用户可以更改自己的密码，除非密码被禁用。要禁用用户的密码，请指定DISABLE。禁用某个用户的密码后，将从系统中删除该密码，此类用户只能通过外部认证来连接数据库，例如：kerberos认证。只有管理员才能启用或禁

用密码。普通用户不能禁用初始用户的密码。要启用密码，请运行ALTER USER并指定密码。

- **SYSADMIN | NOSYSADMIN**

  决定一个新角色是否为“系统管理员”，具有SYSADMIN属性的角色拥有系统最高权限。

  缺省为NOSYSADMIN。

  三权分立关闭时，具有SYSADMIN属性的用户有权限创建具有SYSADMIN、CREATEROLE、AUDITADMIN、MONADMIN、POLADMIN、CREATEDB属性的用户和普通用户。

  三权分立打开时，具有SYSADMIN属性的用户无权创建用户。

- **MONADMIN | NOMONADMIN**

  定义角色是否是监控管理员。

  缺省为NOMONADMIN。

- **OPRADMIN | NOOPRADMIN**

  定义角色是否是运维管理员。

  缺省为NOOPRADMIN。

- **POLADMIN | NOPOLADMIN**

  定义角色是否是安全策略管理员。

  缺省为NOPOLADMIN。

- **AUDITADMIN | NOAUDITADMIN**

  定义角色是否有审计管理属性。

  缺省为NOAUDITADMIN。

- **CREATEDB | NOCREATEDB**

  决定一个新角色是否能创建数据库。

  新角色没有创建数据库的权限。

  缺省为NOCREATEDB。

- **USEFT | NOUSEFT**

  该参数为保留参数，暂未启用。

- **CREATEROLE | NOCREATEROLE**

  决定一个角色是否可以创建新角色（也就是执行CREATE ROLE和CREATE USER）。 一个拥有CREATEROLE权限的角色也可以修改和删除其他角色。

  缺省为NOCREATEROLE。

  三权分立关闭时，具有CREATEROLE属性的用户有权限创建具有CREATEROLE、AUDITADMIN、MONADMIN、POLADMIN、CREATEDB属性的用户和普通用户。

  三权分立打开时，具有CREATEROLE属性的用户有权限创建具有CREATEROLE、MONADMIN、POLADMIN、CREATEDB属性的用户和普通用户。

- **INHERIT | NOINHERIT**

  这些子句决定一个角色是否“继承”它所在组的角色的权限。不推荐使用。

- **LOGIN | NOLOGIN**

  具有LOGIN属性的角色才可以登录数据库。一个拥有LOGIN属性的角色可以认为是一个用户。

缺省为NOLOGIN。

- **REPLICATION | NOREPLICATION**

  定义角色是否允许流复制或设置系统为备份模式。REPLICATION属性是特定的角色，仅用于复制。

  缺省为NOREPLICATION。

- **PERSISTENCE | NOPERSISTENCE**

  定义永久用户。仅允许初始用户创建、修改和删除具有PERSISTENCE属性的永久用户。

- **CONNECTION LIMIT connlimit**

  声明该角色可以使用的并发连接数量。

  须知

  - 系统管理员不受此参数的限制。
  - connlimit数据库主节点单独统计，数据库M-Compatibility整体的连接数 = connlimit * 当前正常数据库主节点个数。

  取值范围：[-1, 2^31-1]的整数。缺省值为-1，表示没有限制。

- **VALID BEGIN 'timestamp'**

  设置角色生效的时间戳。如果省略了该子句，角色无有效开始时间限制，timestamp为生效时间，格式为'YYYY-MM-DD HH:mm:ss'。

- **VALID UNTIL 'timestamp'**

  设置角色失效的时间戳。如果省略了该子句，角色无有效结束时间限制，timestamp为失效时间，格式为'YYYY-MM-DD HH:mm:ss'。

- **USER GROUP 'groupuser'**

  创建一个user的子用户。当前版本暂不支持。

- **IN ROLE role_name**

  新角色立即拥有IN ROLE子句中列出的一个或多个现有角色拥有的权限。不推荐使用。

- **IN GROUP role_name**

  IN GROUP是IN ROLE过时的拼法。不推荐使用。

- **ROLE role_name**

  ROLE子句列出一个或多个现有的角色，它们将自动添加为这个新角色的成员，拥有新角色所有的权限。

- **ADMIN role_name**

  ADMIN子句类似ROLE子句，不同的是ADMIN后的角色可以把新角色的权限赋给其他角色。

- **USER role_name**

  USER子句是ROLE子句过时的拼法。

- **DEFAULT TABLESPACE tablespace_name**

  DEFAULT TABLESPACE子句将被忽略，无实际意义。

- **PROFILE profile_name**

PROFILE子句将被忽略，无实际意义。

- **PGUSER**

  当前版本该属性没有实际意义，仅为了语法的前向兼容而保留。

示例

```
--创建一个角色，名为manager，密码为********。
m_db=# CREATE ROLE manager IDENTIFIED BY '********';

--创建一个角色，从2015年1月1日开始生效，到2026年1月1日失效。
m_db=# CREATE ROLE miriam WITH LOGIN PASSWORD '********' VALID BEGIN '2015-01-01' VALID UNTIL
'2026-01-01';

--修改角色manager的密码为********。
m_db=# ALTER ROLE manager IDENTIFIED BY '********' REPLACE '**********';

--修改角色manager为系统管理员。
m_db=# ALTER ROLE manager SYSADMIN;

--删除角色manager。
m_db=# DROP ROLE manager;

--删除角色miriam。
m_db=# DROP GROUP miriam;
```

相关链接

SET ROLE，**ALTER ROLE**，**DROP ROLE**，**GRANT**

4.4.2.8.14 CREATE SCHEMA

功能描述

创建模式。DATABASE与SCHEMA同义。

访问命名对象时可以使用模式名作为前缀进行访问，如果无模式名前缀，则访问当前模式下的命名对象。创建命名对象时也可用模式名作为前缀修饰。

另外，CREATE SCHEMA可以包括在新模式中创建对象的子命令，这些子命令和那些在创建完模式后发出的命令没有任何区别。如果使用了AUTHORIZATION子句，则所有创建的对象都将被该用户所拥有。

注意事项

- 只要用户对当前数据库有CREATE权限，就可以创建模式。
- 系统管理员在普通用户同名schema下创建的对象，所有者为schema的同名用户（非系统管理员）。

语法格式

```
CREATE {DATABASE | SCHEMA} [IF NOT EXISTS] schema_name
    [[create_option] [,...]]
create_option: [DEFAULT] {
    CHARACTER SET [=] default_charset
  | CHAR SET [=] default_charset
  | CHARSET [=] default_charset
  | COLLATE [=] default_collation
}
```


参数说明

- **schema_name**

  模式名称。

  须知

  - 模式名不能和当前数据库里其他的模式重名。
  - 模式名不能和当前数据库的初始用户重名。
  - 模式的名称不可以“pg_”开头。
  - 模式的名称不可以“gs_role_”开头。

  取值范围：字符串，要符合**标识符说明**。

  说明

  如果当前搜索路径上的模式中存在同名对象时，需要明确指定引用对象所在的模式。可以通过命令SHOW SEARCH_PATH来查看当前搜索路径上的模式。

- **COLLATE [=] collation_name**

  可选。指定新数据库使用的字符集。例如，通过collate = 'zh_CN.gbk'设定该参数。

  该参数的使用会影响到对字符串的排序顺序（如使用ORDER BY执行，以及在文本列上使用索引的顺序）。默认是使用模板数据库的字符集。

  取值范围：参考**库级字符集和字符序**。

- **{CHAR SET | CHARSET | CHARACTER SET} [=] charset_name**

  可选。指定新数据库使用的字符分类。例如，通过CHARSET = 'zh_CN.gbk'设定该参数。该参数的使用会影响到字符的分类，如大写、小写和数字。默认是使用模板数据库的字符分类。

  取值范围：参考**库级字符集和字符序**。

示例

```
--创建并切换至测试数据库。
m_db=# CREATE SCHEMA test1;
m_db=# USE test1;
--创建表a。
m_db=# CREATE TABLE a(id int);
m_db=# DROP TABLE a;
--删除SCHEMA test1。
m_db=# USE public;
m_db=# DROP SCHEMA test1;
```

相关链接

ALTER SCHEMA，**DROP SCHEMA**

4.4.2.8.15 CREATE SEQUENCE

功能描述

用于向当前数据库增加一个新的序列。序列的Owner为创建此序列的用户。

注意事项

- Sequence是一个存放等差数列的特殊表。这个表没有实际意义，通常用于为行或者表生成唯一的标识符。
- 如果给出一个模式名，则该序列就在给定的模式中创建，否则会在当前模式中创建。序列名必须和同一个模式中的其他序列、表、索引、视图或外表的名称不同。
- 创建序列后，在表中使用序列的nextval()函数和generate_series(1,N)函数对表插入数据，请保证nextval的可调用次数大于等于N+1次，否则会因为generate_series()函数会调用N+1次而导致报错。
- Sequence默认最大值为2^63-1。
- 被授予CREATE ANY SEQUENCE权限的用户，可以在public模式和用户模式下创建序列。

语法格式

```
CREATE  SEQUENCE name [ INCREMENT [ BY ] increment ]
    [ MINVALUE minvalue | NO MINVALUE | NOMINVALUE ] [ MAXVALUE maxvalue | NO MAXVALUE |
NOMAXVALUE]
    [ START [ WITH ] start ] [ CACHE cache ] [ [ NO ] CYCLE | NOCYCLE ]
    [ RESTART [ WITH ] value | RESTART ]
    [ OWNED BY { table_name.column_name | NONE } ];
```

参数说明

- **name**

  将要创建的序列名称。

  取值范围: 仅可以使用小写字母（a~z）、 大写字母（A~Z），数字和特殊字符“#” 、“_”、“$”的组合。

- **increment**

  可选。指定序列的步长。一个正数将生成一个递增的序列，一个负数将生成一个递减的序列。

  缺省值为1。

- **MINVALUE minvalue | NO MINVALUE| NOMINVALUE**

  可选。执行序列的最小值。如果没有声明minvalue或者声明了NO MINVALUE，则递增序列的缺省值为1，递减序列的缺省值为-2^{63}-1。NOMINVALUE等价于NO MINVALUE

- **MAXVALUE maxvalue | NO MAXVALUE| NOMAXVALUE**

  可选。执行序列的最大值。如果没有声明maxvalue或者声明了NO MAXVALUE，则递增序列的缺省值为2^{63}-1，递减序列的缺省值为-1。NOMAXVALUE等价于NO MAXVALUE

- **start**

  可选。指定序列的起始值。缺省值：对于递增序列为minvalue，递减序列为maxvalue。

- **cache**

  可选。为了快速访问，而在内存中预先存储序列号的个数。

  缺省值为1，表示一次只能生成一个值，也就是没有缓存。

说明

不建议同时定义cache和maxvalue或minvalue。因为定义cache后不能保证序列的连续性，可能会产生空洞，造成序列号段浪费。

- **[NO] CYCLE | NOCYCLE**

  可选。用于使序列达到maxvalue或者minvalue后可循环并继续下去。

  如果声明了NO CYCLE，则在序列达到其最大值后任何对nextval的调用都会返回一个错误。

  - NOCYCLE的作用等价于NO CYCLE。缺省值为NO CYCLE。
  - 若定义序列为CYCLE，则不能保证序列的唯一性。

- **OWNED BY**

  可选。将序列和一个表的指定字段进行关联。这样，在删除指定字段或其所在表的时候会自动删除已关联的序列。关联的表和序列的所有者必须是同一个用户，并且在同一个模式中。需要注意的是，通过指定OWNED BY，仅仅是建立了表的对应列和sequence之间关联关系，并不会在插入数据时在该列上产生自增序列。

  缺省值为OWNED BY NONE，表示不存在这样的关联。

须知

通过OWNED BY创建的Sequence不建议用于其他表，如果希望多个表共享Sequence，该Sequence不应该从属于特定表。

示例

创建一个名为seq1的递增序列，从101开始，步长为10。

```
m_db=# CREATE SEQUENCE seq1
    START 101
    INCREMENT 10;

--从序列中选出下一个数字：
m_db=# SELECT nextval('seq1');
 nextval
---------
     101
(1 row)
m_db=# SELECT nextval('seq1');
 nextval
---------
     111

--删除序列。
m_db=# DROP SEQUENCE seq1;
```

相关链接

DROP SEQUENCE，**ALTER SEQUENCE**

4.4.2.8.16 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

- 如果在建表过程中数据库系统发生故障，系统恢复后可能无法自动清除之前已创建的、大小为0的磁盘文件。此种情况出现概率小，不影响数据库系统的正常运行。
- 使用JDBC时，支持通过PrepareStatement对DEFAULT值进行参数化设置。
- 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表。
- 主键不支持前缀索引。

语法格式

创建表。

```
CREATE [ [LOCAL | GLOBAL] TEMPORARY | UNLOGGED ] TABLE [ IF NOT EXISTS ] table_name
   {({ column_name data_type [ {CHARACTER SET | CHAR SET | CHARSET} charset_name ]
[ compress_mode ] [ COLLATE collation_name][AUTO_INCREMENT][COMMENT [=] 'string']
[ column_constraint [ ... ] ]
       | table_constraint
       | {INDEX | KEY} [ [schema_name.] index_name ] [ USING method ] ({column_name [(length)] | (expr)
[ASC | DESC]}[,...]) [[COMMENT 'string' | USING method][...]]
       | LIKE source_table }
       [, ... ])}
   [ table_option ][[,...]|[...]]
   [ WITH ( {storage_parameter = value} [, ... ] ) ]
   [ ON COMMIT { PRESERVE ROWS | DELETE ROWS } ]
   [ TABLESPACE tablespace_name ];

CREATE [ [LOCAL | GLOBAL] TEMPORARY ] TABLE [IF NOT EXISTS] table_name LIKE source_table;
```

- 其中table_option为：

  ```
  {
     AUTO_INCREMENT [=] value
   | [DEFAULT] {CHARACTER SET | CHAR SET | CHARSET} [=] charset_name
   | [DEFAULT] COLLATE [=] collation_name
   | COMMENT [=] 'string'
   | ENGINE [=] engine_name
   | ROW_FORMAT [=] {DEFAULT | DYNAMIC | FIXED | COMPRESSED | REDUNDANT | COMPACT}
  }
  ```

  说明

  - 同一个地方多次设置COMMENT时仅最后一个生效。
  - 同一个地方多次设置AUTO_INCREMENT时仅最后一个生效。
  - 同一个地方多次设置COLLATE时仅最后一个生效。
  - 同一个地方多次设置CHARSET时仅最后一个生效。
  - ENGINE、ROW_FORMAT使用时语法不报错也没有提示，但实际不生效。
  - 不支持WITH (ORIENTATION = column)

- 其中列约束column_constraint为：

  ```
  [ CONSTRAINT constraint_name ]
  { NOT NULL |
    NULL |
    CHECK ( expression ) |
    DEFAULT default_expr |
    ON UPDATE now_expr |
    [ GENERATED ALWAYS ] AS ( generation_expr ) [STORED | VIRTUAL] |
    UNIQUE [KEY] index_parameters |
    [ PRIMARY ] KEY index_parameters |
    REFERENCES reftable [ ( refcolumn ) ] [ MATCH FULL | MATCH SIMPLE ]
       [ ON DELETE action ] [ ON UPDATE action ] }
  ```

- 其中列的压缩可选项compress_mode为：

  ```
  {  DICTIONARY  }
  ```


- 其中表约束table_constraint为：

```
[ CONSTRAINT [ constraint_name ] ]
{ CHECK ( expression ) |
  UNIQUE [ INDEX | KEY ] [ index_name ][ USING method ] ( { { column_name [ ( length ) ] |
（ expression ） } [ ASC | DESC ] } [, ... ] ) index_parameters [USING method| COMMENT 'string']|
  PRIMARY KEY [index_name][ USING method ] ( { column_name  }[ ASC | DESC ] } [, ... ] )
index_parameters [USING method| COMMENT 'string']|
  FOREIGN KEY [ index_name ] ( column_name [, ... ] ) REFERENCES reftable [ (refcolumn [, ... ] ) ]
      [ MATCH FULL | MATCH SIMPLE ] [ ON DELETE action ] [ ON UPDATE action ]
  }
```

- 其中索引参数index_parameters为：

```
[ WITH ( {storage_parameter = value} [,...]
[ USING INDEX TABLESPACE tablespace_name ]
```

- 其中now_expr为：

```
{ CURRENT_TIMESTAMP | LOCALTIMESTAMP  | LOCALTIME | NOW() }
```

参数说明

- **UNLOGGED**

  如果指定该关键字，则创建的表为非日志表。

  - 在非日志表中写入的数据不会被写入到预写日志中，当执行操作系统重启、数据库重启、主备切换、切断电源操作或异常关机后，非日志表上的写入会被自动截断，会造成数据丢失的风险。非日志表中的内容也不会被复制到备服务器中。在非日志表中创建的索引也不会被自动记录。
  - 备服务器上非日志表无数据，且在备机查询非日志表时会报错。

  使用场景：非日志表无法保证数据的安全性，用户应该在确保数据已经做好备份的前提下使用非日志表，例如系统升级时进行数据的备份。

  故障处理：当异常关机等操作导致非日志表上的索引发生数据丢失时，用户应该对发生错误的索引进行重建。

- **GLOBAL | LOCAL**

  创建临时表时可以在TEMP或TEMPORARY前指定GLOBAL或LOCAL关键字。如果指定GLOBAL关键字，M-Compatibility会创建全局临时表，否则M-Compatibility会创建本地临时表。

- **TEMPORARY**

  如果指定该关键字，则创建的表为临时表。

  临时表分为全局临时表和本地临时表两种类型。创建临时表时如果指定GLOBAL关键字则为全局临时表，否则为本地临时表。

  全局临时表的元数据对所有会话可见，会话结束后元数据继续存在。会话与会话之间的用户数据、索引和统计信息相互隔离，每个会话只能看到和更改自己提交的数据。全局临时表有两种模式：一种是基于会话级别的(ON COMMIT PRESERVE ROWS), 当会话结束时自动清空用户数据；一种是基于事务级别的(ON COMMIT DELETE ROWS), 当执行commit或rollback时自动清空用户数据。建表时如果没有指定ON COMMIT选项，则缺省为会话级别。与本地临时表不同，全局临时表建表时可以指定非pg_temp_开头的schema。

  本地临时表只在当前会话可见，本会话结束后会自动删除。因此，在除当前会话连接的数据库节点故障时，仍然可以在当前会话上创建和使用临时表。由于临时表只在当前会话创建，对于涉及对临时表操作的DDL语句，会产生DDL失败的报错。因此，建议DDL语句中不要对临时表进行操作。

须知

- 本地临时表通过每个会话独立的以pg_temp开头的schema来保证只对当前会话可见，因此，不建议用户在日常操作中手动删除以pg_temp、pg_toast_temp开头的schema。
- 如果建表时不指定TEMPORARY关键字，而指定表的schema为当前会话的pg_temp_开头的schema，则此表会被创建为临时表。
- ALTER/DROP全局临时表和索引，如果其它会话正在使用它，禁止操作。
- 全局临时表的DDL只会影响当前会话的用户数据和索引。例如truncate、reindex、analyze只对当前会话有效。
- 全局临时表功能可以通过设置GUC参数max_active_global_temporary_table控制是否启用。如果max_active_global_temporary_table=0，关闭全局临时表功能。
- 临时表只对当前会话可见，因此不支持与\parallel on并行执行一起使用。
- 临时表不支持主备切换。
- 全局临时表不响应自动清理，在长连接场景使用时尽量使用on commit delete rows的全局临时表，或定期手动执行vacuum，否则可能导致clog日志不回收。
- 全局临时表不支持以下场景：
  - 不支持创建全局临时Sequence，各个会话的全局临时表使用共享的Sequence，只能保证唯一性，不保证连续性。
  - 不支持创建全局临时视图。
  - 不支持创建分区表。
  - 不支持创建Hash bucket表。
  - 不支持扩展统计信息。

- **IF NOT EXISTS**

  如果已经存在相同名称的表，不会报出错误，而会发出通知，告知通知此表已存在。
- **table_name**

  要创建的表名。
- **column_name**

  新表中要创建的字段名。
- **constraint_name**

  建表时指定的约束名称。
- **index_name**

  索引名。

须知

- 对于外键约束，constraint_name和index_name同时指定时，索引名为constraint_name。
- 对于唯一键约束，constraint_name和index_name同时指定时，索引名为index_name。

- **USING method**

  指定创建索引的方法。

  取值范围请参见**参数说明**中的USING method。

  须知

  目前只支持索引方法为btree，但是不支持ustore表创建btree索引。

- **ASC | DESC**

  ASC表示指定按升序排序（默认）。DESC指定按降序排序。

- **expression**

  创建一个基于该表的一个或多个字段的表达式索引约束，必须写在圆括弧中。

- **data_type**

  字段的数据类型。

- **compress_mode**

  表字段的压缩选项。该选项指定表字段优先使用的压缩算法。行存表不支持压缩。当前M-Compatibility模式数据库下无法使用。

  取值范围：DICTIONARY

- **{CHARACTER SET | CHAR SET | CHARSET} charset_name**

  指定表字段的字符集。单独指定时会将字段的字符序设置为指定的字符集的默认字符序。

- **COLLATE collation_name**

  COLLATE子句指定列的排序规则（字符序）（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。支持的字符序具体请参见**表级字符集和字符序**

  说明

  - 仅字符类型支持指定字符集。指定为BINARY字符集或字符序实际是将字符类型转化为对应的二进制类型，若类型映射不存在则报错。当前仅有TEXT类型转化为BLOB的映射。
  - 字段字符集或字符序未显式指定时，若指定了表的默认字符集或字符序，字段字符集和字符序将从表上继承。
  - 除SQL_ASCII库外，其他字符集的数据库支持多字符集混用。

- **LIKE source_table**

  LIKE子句声明一个表，新表将从源表继承以下信息：

  - 字段名及其数据类型。
  - 非空约束、CHECK约束、主键约束、唯一键约束、索引、默认值以及ON UPDATE属性。
  - 表、字段、索引以及约束上的注释。
  - 分区信息。
  - 表的存储参数以及字段的存储参数。

  新表与源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在新表中产生影响，并且在扫描源表的时候也不会包含新表的数据。

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。

说明

如果建表语句中包括LIKE子句和WITH子句、PARTITION BY子句时，建表将会使用SQL中指定的WITH信息和分区信息，而不再使用源表的信息。对于CHARSET、COMMENT等信息，同时指定时使用源表的信息。

- **AUTO_INCREMENT [=] value**

  这个子句为自动增长列指定一个初始值，value必须为正数，不得超过2^{127}-1。

- **COMMENT [=] 'string'**
  - COMMENT [=] 'string'子句表示给表添加注释。
  - 在column_constraint中的COMMENT 'string'表示给列添加注释。
  - 在table_constraint中的COMMENT 'string'表示给主键和唯一键对应的索引添加注释。

须知

- 表级注释支持的最大字符串长度为2048字符，列级和索引级注释支持的最大长度为1024字符。
- table_constraint中的COMMENT仅支持主键和唯一键，其他约束不支持。

- **WITH ({ storage_parameter = value } [, ...])**

  这个子句为表或索引指定一个可选的存储参数。用于表的WITH子句还可以包含OIDS=FALSE表示不分配OID。

  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。在Ustore存储引擎下，该值得默认值为92，在Astore存储引擎下默认值为100（完全填充）。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。

    取值范围：10~100
  - **ORIENTATION**

    指定表数据的存储方式，该参数设置成功后就不再支持修改，当前M-Compatibility仅支持行存方式。

    取值范围：

    - ROW，表示表的数据将以行式存储。

      行存储适合于OLTP业务，适用于点查询或者增删操作较多的场景。

    默认值：

    若指定表空间为普通表空间，默认值为ROW。
  - **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ORIENTATION和STORAGE_TYPE时创建表，默认是USTORE存储引擎（表示表支持Inplace-Update存储引擎）。

- **INIT_TD**

  创建UStore表时，指定初始化的TD个数，该参数可以通过alter table进行修改。特别需要注意，该参数会影响数据页面存放的单个元组的最大大小，具体换算方法为MAX_TUPLE_SIZE = BLCKSZ - INIT_TD * TD_SIZE，例如用户将INIT_TD数量从4修改为8，单个元组最大大小会减小4 * INIT_TD大小。

  取值范围：2~128，默认值为4。

- **COMPRESSION**

  指定表数据的压缩级别，它决定了表数据的压缩比以及压缩时间。一般来讲，压缩级别越高，压缩比也越大，压缩时间也越长；反之亦然。实际压缩比取决于加载的表数据的分布特征。行存表不支持压缩。

  取值范围：行存表不支持压缩，默认值为NO。

- **COMPRESSLEVEL**

  指定表数据同一压缩级别下的不同压缩水平，它决定了同一压缩级别下表数据的压缩比以及压缩时间。对同一压缩级别进行了更加详细的划分，为用户选择压缩比和压缩时间提供了更多的空间。总体来讲，此值越大，表示同一压缩级别下压缩比越大，压缩时间越长；反之亦然。

  取值范围：0~3，默认值为0。

- **segment**

  使用段页式的方式存储。本参数仅支持行存表。不支持1-5号物理文件非法删除破坏场景的防护。

  取值范围：on/off

  默认值：off

- **enable_tde**

  指定该表为加密表。数据库会自动将加密表中的数据先加密再存储。使用该参数前，请确保已通过GUC参数enable_tde开启透明加密功能，并通过GUC参数tde_key_info设置访问密钥服务的信息，在《特性指南》中“透明数据加密”章节可获取该参数的详细使用方法。本参数仅支持行存表、段页式表、临时表和unlogged表。

  取值范围：on/off。设置enable_tde=on时，key_type、tde_cmk_id、dek_cipher参数由数据库自动生成，用户无法手动指定或更改。

  默认值：off

- **parallel_workers**

  表示创建索引时起的bgworker线程数量，例如2就表示将会起2个bgworker线程并发创建索引。

  取值范围：[0,32]，int类型，0表示关闭并行建索引。

  默认值：不设置该参数，表示未开启并行建索引功能。

- encrypt_algo

指定加密表的加密算法，需与enable_tde结合使用。

取值范围：字符串，有效值为：AES_128_CTR，SM4_CTR。

默认值：为tde_encrypt_config参数中table_algorithm的赋值。

- dek_cipher

  数据密钥的密文。用户为表设置enable_tde参数后，数据库自动生成数据密钥。

  取值范围：字符串

  默认值：空

- key_type

  主密钥的类型。用户为表设置enable_tde参数后，数据库自动从GUC参数tde_key_info中获取主密钥的类型。

  取值范围：字符串

  默认值：空

- cmk_id

  主密钥的ID。用户为表设置enable_tde参数后，数据库自动从GUC参数tde_key_info中获取主密钥的ID。

  取值范围：字符串

  默认值：空

- **hasuids**

  参数开启：更新表元组时，为元组分配表级唯一标识id。

  取值范围：on/off。

  默认值：off。

- **collate**

  用于记录表的默认字符序，一般只用于内部存储和导入导出，不推荐用户指定或修改。

  取值范围：支持的字符序的oid。

  默认值：0。

- **stat_state**

  标识该表的统计信息是否被锁定，如果被锁定了，该表的统计信息无法更新。

  取值范围：locked、unlock。

  默认值：unlock。

- **ON COMMIT { PRESERVE ROWS | DELETE ROWS }**

  ON COMMIT选项决定在事务中执行创建临时表操作，当事务提交时，此临时表的后续操作。有以下两个选项，当前支持PRESERVE ROWS和DELETE ROWS选项。

  - PRESERVE ROWS（缺省值）：提交时不对临时表做任何操作，临时表及其表数据保持不变。
  - DELETE ROWS：提交时删除临时表中数据。

- **TABLESPACE tablespace_name**

  创建新表时指定此关键字，表示新表将要在指定表空间内创建。如果没有声明，将使用默认表空间。

  注：需要在非M-Compatibility数据库中创建或删除tablespace。

- **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。

- **NOT NULL**

  字段值不允许为NULL。

- **NULL**

  字段值允许为NULL ，这是缺省值。

  这个子句只是为和非标准SQL数据库兼容。不建议使用。

- **CHECK (expression)**

  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。

  说明

  - expression表达式中，如果存在“<>NULL”或“！=NULL”，这种写法是无效的，需要修改为“is NOT NULL”。
  - 不开启精度传递开关（m_format_behavior_compat_options不开启enable_precision_decimal选项）时创建的带有CHECK约束的表，与开启精度传递开关后创建的带有CHECK约束的表在浮点数的比较结果等方面可能出现不一致的情况。如果用户需要在开启精度传递开关的情况下使用未开启精度开关时创建的带有CHECK约束的表，建议在开启精度传递开关后使用ALTER TABLE语法重新定义CHECK约束。

  ```
  m_db=# SET m_format_behavior_compat_options='';
  SET
  m_db=# CREATE TABLE mm1(a float(10, 4), b float(5, 3), CHECK(a/b=1.7142858) STORED);
  CREATE TABLE
  m_db=# INSERT INTO mm1 VALUES(1.2, 0.7);
  ERROR:  New row in relation "mm1" violates check constraint "mm1_check".
  DETAIL:  N/A
  m_db=# SET m_format_behavior_compat_options='enable_precision_decimal';
  SET
  m_db=# INSERT INTO mm1 VALUES(1.2, 0.7);
  INSERT 0 1
  m_db=# CREATE TABLE mm2(a float(10, 4), b floaT(5, 3), CHECK(a/b=1.7142858) STORED);
  CREATE TABLE
  m_db=# INSERT INTO mm2 VALUES(1.2, 0.7);
  ERROR:  New row in relation "mm2" violates check constraint "mm2_check".
  DETAIL:  N/A

  m_db=# DROP TABLE mm1, mm2;
  CREATE TABLE
  ```

- **DEFAULT default_expr**

  - 使用DEFAULT子句给字段指定缺省表达式，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 。
  - 当未在缺省表达式外嵌套括号时，支持指定以下内容：常量、带正负号的数值常量、**update_expr**。
  - 当在缺省表达式外嵌套括号时，支持指定以下内容：常量、带正负号的数值常量、**update_expr**、CURRENT_TIME/CURTIME函数、CURRENT_DATE/CURDATE函数（CURRENT_TIME/CURRENT_DATE支持不带括号形式的调用）。
  - 仅支持在TIMESTAMP、DATETIME类型的字段上指定update_expr作为默认值，且字段的精度与update_expr的精度须保持一致。

- **ON UPDATE now_expr**

  ON UPDATE子句为字段的一种属性约束。

  当对表中某元组执行UPDATE操作时，若更新字段的新值和表中旧值不相同，则表中该元组上具有该属性且不在更新字段内的字段值自动更新为当前时间戳；若更新字段的新值和表中旧值相同，则表中该元组上具有该属性且不在更新字段内的字段值不变，保持原有值；若具有该属性的字段在更新字段内，则对应这些字段值直接按指定更新的值更新。

  说明

  - 语法上update_expr支持CURRENT_TIMESTAMP 、LOCALTIMESTAMP 、LOCALTIME、NOW()四种关键字，也支持关键字带括号指定或不指定精度。例如：ON UPDATE CURRENT_TIMESTAMP()、ON UPDATE CURRENT_TIMESTAMP(5)、ON UPDATE LOCALTIMESTAMP()、ON UPDATE LOCALTIMESTAMP(6)等。不带括号或空括号时精度为0，其中NOW关键字不支持不带括号。四种关键字互为同义词，属性效果相同。
  - 该属性支持在如下类型的列上指定: TIMESTAMP、DATETIME。
  - 该属性指定的精度和对应列上类型指定的精度必须一致，否则会触发报错。例如：CREATE TABLE t1 (col1 timestamp(6) ON UPDATE CURRENT_TIMESTAMP(6)); 若精度不一致，会产生ERROR: Invalid ON UPDATE clause for "col1" 报错。
  - 该属性和生成列约束不能同时指定同一列。
  - 分区表中的分区键不支持指定该属性。

- **[GENERATED ALWAYS] AS (generation_expr) [STORED | VIRTUAL]**

  该子句将字段创建为生成列，通过指定STORED和VIRTUAL关键字表明生成列的列值存储方式：

  - STORED：创建存储生成列，列值需要存储，在插入或更新元组时，由生成列表达式计算存储生成列的列值。
  - VIRTUAL：创建虚拟生成列，列值不需要存储，在查询语句涉及虚拟生成列时，由生成列表达式计算列值。

说明

- STORED和VIRTUAL关键字可省略，缺省为STORED，在参数m_format_dev_version为s2条件下缺省为VIRTUAL。
- 生成表达式不能以任何方式引用当前行以外的其他数据。生成表达式支持引用早定义于当前生成列的其他生成列，不能引用系统列。生成列表达式不能引用系统变量、用户自定义变量、存储过程中的变量。生成列表达式不支持引用自增列，且生成列不支持定义自增列属性。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、窗口函数和自定义函数。生成表达式调用的函数只能是不可变（IMMUTABLE）函数。
- 不支持为生成列指定默认值。
- 存储生成列支持作为分区键的一部分，当作为分区键的一部分时，不支持ALTER TABLE修改存储生成列。
- 虚拟生成列不支持作为分区键的一部分。
- 虚拟生成列不支持被外键约束引用。
- 不支持在虚拟生成列上创建索引。
- 存储生成列不能和ON UPDATE约束子句的CASCADE,SET NULL,SET DEFAULT动作同时指定。存储生成列不能和ON DELETE约束子句的SET NULL,SET DEFAULT动作同时指定。
- 在参数m_format_dev_version为s2条件下，存储生成列的基列不能和ON UPDATE或者ON DELETE约束子句的CASCADE, 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。删除生成列依赖的基列，生成列被自动删除。在参数m_format_dev_version为s2条件下，删除生成列依赖的基列之前需要先删除对应的生成列。
- 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。向可被更新的视图中插入生成列数据时，不能为生成列指定值，但是可以指定关键字DEFAULT。
- 当存储生成列带CHECK约束时，在参数m_format_dev_version为s2的条件下，ALTER TABLE修改存储生成列定义，如果存在生成列不满足CHECK约束，ALTER TABLE语句执行失败，系统报ERROR。
- 生成列的权限控制和普通列一样。
- 不开启精度传递开关（m_format_behavior_compat_options不开启enable_precision_decimal选项）时创建的带有生成列的表，与开启精度传递开关后创建的带有生成列的表在浮点数的比较结果等方面可能出现不一致的情况。如果用户需要在开启精度传递开关的情况下使用未开启精度开关时创建的带有生成列的表，建议在开启精度传递开关后使用ALTER TABLE语法重新定义生成列。

```
m_db=# SET m_format_behavior_compat_options='';
SET
m_db=# CREATE TABLE mm1(a float(10, 4), b float(5, 3), c boolean AS ((a/b)=1.7142858)
STORED);
CREATE TABLE
m_db=# INSERT INTO mm1 VALUES(1.2, 0.7);
INSERT 0 1
m_db=# SELECT * FROM mm1;
  a    |   b   | c
--------+-------+---
 1.2000 | 0.700 | 0
(1 row)

m_db=# SET m_format_behavior_compat_options='enable_precision_decimal';
SET
m_db=# CREATE TABLE mm2(a float(10, 4), b float(5, 3), c boolean AS ((a/b)=1.7142858)
STORED);
CREATE TABLE
m_db=# INSERT INTO mm1 VALUES(1.2, 0.7);
INSERT 0 1
m_db=# INSERT INTO mm2 VALUES(1.2, 0.7);
INSERT 0 1
m_db=# SELECT * FROM mm1;
  a    |   b   | c
```



```
--------+-------+---
 1.2000 | 0.700 | 0
 1.2000 | 0.700 | 1
(2 rows)

m_db=# SELECT * FROM mm2;
   a    |   b   | c
--------+-------+---
 1.2000 | 0.700 | 0
(1 row)

m_db=# DROP TABLE mm1, mm2;
DROP TABLE
```

- **AUTO_INCREMENT**

  该关键字将字段指定为自动增长列。

  自动增长列建议设置索引，并需要作为索引的第一个字段，否则建表时产生警告。

  若在插入时不指定此列的值（或指定此列的值为0、NULL、DEFAULT），此列的值将由自增计数器自动增长得到。

  若插入或更新此列为一个大于当前自增计数器的值，执行成功后，自增计数器将刷新为此值。

  自增初始值由“AUTO_INCREMENT [=] value”子句设置，若不设置，默认为1。

  说明

  - 自动增长列数据类型只能为整数类型、4字节或8字节浮点类型、布尔类型。当自增值已经达到字段数据类型的最大值时，继续自增将产生错误。
  - 每个表只能有一个自动增长列。
  - 自动增长列建议设置索引，并需要作为索引的第一个字段，否则建表时产生警告，含有自动增长列的表进行某些操作时会产生错误，例如：ALTER TABLE EXCHANGE PARTITION。
  - 自动增长列不能指定DEFAULT缺省值。
  - CHECK约束的表达式中不能含有自动增长列，生成列的表达式中不能含有自动增长列。
  - 可以指定自动增长列允许NULL，若不指定，默认自动增长列含有NOT NULL约束。
  - 含有自动增长列的表创建时，会创建一个依赖于此列的序列作为自增计数器，不允许通过序列相关功能修改或删除此序列，可以查看序列的值。请勿使其他序列依赖或关联此自动增长列。
  - 本地临时表中的自动增长列不会创建序列。
  - 自增计数器自增和刷新操作不会回滚。
    - 数据插入到表之前，0/NULL会触发自增。数据插入或更新到表之后，会更新自增计数器。如果在自增之后出现了报错，数据没有插入或更新到表中，此时自增计数器不会回滚。后续插入语句基于自增计数器触发自增，会出现表中自动增长列的值不连续的情况。
    - 批量插入或导入预留自增缓存值也有可能产生自动增长列的值不连续的情况，详见auto_increment_cache参数说明。

- **[DEFAULT] {CHARACTER SET | CHAR SET | CHARSET} [=] default_charset**

  指定表的默认字符集。单独指定时会将表的默认字符序设置为指定的字符集的默认字符序。

- **[DEFAULT] COLLATE [=] default_collation**

  指定表的默认字符序。单独指定时会将表的默认字符集设置为指定的字符序对应的字符集。

字符序请参见**字符集与字符序**。

说明

表的字符集或字符序未显式指定时，若指定了模式的默认字符集或字符序，表字符集和字符序将从模式上继承。

- **列级唯一约束：UNIQUE [KEY] index_parameters**

  UNIQUE约束表示表里的一个或多个字段的组合必须在全表范围内唯一。

  UNIQUE KEY与UNIQUE语义相同。

- **表级唯一约束：UNIQUE [INDEX | KEY][index_name][USING method] ({{ column_name [(length)] | (expression) } [ASC | DESC] }[, ...]) 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。

  column_name(length)是前缀键，详见：**•column_name (length)**。

  index_name为索引名。

须知

对于唯一键约束，constraint_name和index_name同时指定时，索引名为index_name。

- **列级主键约束：[PRIMARY] KEY index_parameters**

  表级主键约束：PRIMARY KEY [index_name] [USING method] ({ column_name [ASC | DESC] } [, ...] ） 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。

- **表列字段约束：REFERENCES reftable [(refcolumn)] [MATCH matchtype] [ON DELETE action] [ON UPDATE action]**

  表级别约束：FOREIGN KEY (column_name [, ...]) REFERENCES reftable [(refcolumn [, ...])] [MATCH matchtype] [ON DELETE action] [ON UPDATE action]

  说明

  表列字段约束REFERENCES在数据库M-compatibility模式兼容版本控制开关s1及以上版本时，语法不报错也没有提示，但实际不生效（如m_format_dev_version = 's1'）。

  外键约束要求新表中一列或多列构成的组应该只包含、匹配被参考表中被参考字段值。若省略refcolumn，则将使用reftable的主键。被参考列应该是被参考表中的唯一字段或主键。外键约束不能被定义在临时表和永久表之间。

  参考字段与被参考字段之间存在三种类型匹配，分别是：

  - MATCH FULL：不允许一个多字段外键的字段为NULL，除非全部外键字段都是NULL。
  - MATCH SIMPLE（缺省）：允许任意外键字段为NULL。

  另外，当被参考表中的数据发生改变时，某些操作也会在新表对应字段的数据上执行。ON DELETE子句声明当被参考表中的被参考行被删除时要执行的操作。ON UPDATE子句声明当被参考表中的被参考字段数据更新时要执行的操作。对于ON DELETE子句、ON UPDATE子句的可能动作：

  - NO ACTION（缺省）：删除或更新时，创建一个表明违反外键约束的错误。
  - RESTRICT：删除或更新时，创建一个表明违反外键约束的错误。
  - CASCADE：删除新表中任何引用了被删除行的行，或更新新表中引用行的字段值为被参考字段的新值。
  - SET NULL：设置引用字段为NULL。
  - SET DEFAULT：设置引用字段为它们的缺省值。

  说明

  - 外键约束的完整性检查由GUC参数foreign_key_checks进行控制，foreign_key_checks可以设置为on/off，分别表示启用/关闭外键约束的完整性检查，默认情况下为on。

- **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。

  说明

  该属性在数据库M-compatibility模式兼容版本控制开关为s1时不支持（即m_format_dev_version = 's1'）。

建表示例

- **临时表**

```
--创建表临时表。
m_db=# CREATE TEMPORARY TABLE test_t1(
    id       CHAR(7),
    name     VARCHAR(20),
    province VARCHAR(60),                    --省
    country   VARCHAR(30) DEFAULT 'China'        --国籍
);

-- 在当前会话中插入数据。
m_db=# INSERT INTO test_t1 VALUES ('0000009','Jack','Guangzhou','China');

--临时表里面的数据只在当前事务中有效，所以在另一个会话中查看该表中没有数据。
m_db=# SELECT * FROM test_t1;
 id | name | age
----+------+-----
(0 rows)
--事务中创建表临时表，并指定提交事务时删除该临时表数据。
m_db=# START TRANSACTION;
m_db=# CREATE TEMPORARY TABLE test_t2(
    id       CHAR(7),
    name     VARCHAR(20),
    province VARCHAR(60),                    --省
    country   VARCHAR(30) DEFAULT 'China'        --国籍
) ON COMMIT DELETE ROWS;
m_db=# INSERT test_t2 VALUES('aid','aname','aprovince','China');
m_db=# COMMIT;
m_db=# SELECT * FROM test_t2;
 id | name | province | country
----+------+----------+---------
(0 rows)
--删除表。
m_db=# DROP TABLE test_t1;
m_db=# DROP TABLE test_t2;
```

- **IF NOT EXIST关键字**

  使用该关键字，表不存在时报NOTICE；如不用该关键字，则报ERROR。两种情况下表都不会创建成功。


```
m_db=# CREATE TABLE test_t3(id INT);
--创建一个已经存在同名的表test_t3。
m_db=# CREATE TABLE test_t3(id INT);
ERROR:  relation "test_t3" already exists in schema "public"
DETAIL:  creating new table with existing name in the same schema

--使用IF NOT EXISTS关键字。
m_db=# CREATE TABLE IF NOT EXISTS test_t3(id INT);
NOTICE:  relation "test_t3" already exists, skipping
CREATE TABLE

--删除表。
m_db=# DROP TABLE test_t3;
```

- **建表时指定AUTO_INCREMENT自增列**

```
-- 创建表指定自增列，从10开始自增
m_db=# CREATE TABLE test_autoinc(col int AUTO_INCREMENT, col1 int) AUTO_INCREMENT = 10;

-- 建议自增列作为索引首列，创建一个索引
m_db=# CREATE INDEX test_autoinc_ai ON test_autoinc(col);

-- NULL触发自增，自增值为10
m_db=# INSERT INTO test_autoinc(col, col1) VALUES(NULL,1);

-- 100不触发自增，插入成功后，自增计数更新为100
m_db=# INSERT INTO test_autoinc(col, col1) VALUES(100,2);

-- 0触发自增，自增值为101
m_db=# INSERT INTO test_autoinc(col, col1) VALUES(0,3);

m_db=# SELECT col,col1 FROM test_autoinc ORDER BY 2,1;
 col | col1
-----+------
  10 |    1
 100 |    2
 101 |    3
(3 rows)
```

- **CREATE TABLE ... LIKE建表**

```
-- 创建源表t1
m_db=# CREATE TABLE t1(col INT);
CREATE TABLE

m_db=# \d t1
      Table "public.t1"
 Column |  Type   | Modifiers
--------+---------+-----------
 col    | integer |

-- 创建目标表t2
m_db=# CREATE TABLE t2(LIKE t1);
CREATE TABLE

m_db=# \d t2
      Table "public.t2"
 Column |  Type   | Modifiers
--------+---------+-----------
 col    | integer |

-- 创建目标表t3
m_db=# CREATE TABLE t3 LIKE t1;
CREATE TABLE

m_db=# \d t3
      Table "public.t3"
 Column |  Type   | Modifiers
--------+---------+-----------
 col    | integer |
```

- 建表时创建生成列


```
-- 创建虚拟生成列。
m_db=# CREATE TABLE triangle (
  a DOUBLE,
  b DOUBLE,
  c DOUBLE AS (SQRT(a * a + b * b)) VIRTUAL
);

m_db=# INSERT INTO triangle(a, b) VALUES (3, 4);

m_db=# SELECT * FROM triangle;
 a | b | c
---+---+---
 3 | 4 | 5
(1 row)

-- 创建存储生成列。
m_db=# CREATE TABLE triangle (
  a DOUBLE,
  b DOUBLE,
  c DOUBLE AS (SQRT(a * a + b * b)) STORED
);

m_db=# INSERT INTO triangle(a, b) VALUES (3, 4);

m_db=# SELECT * FROM triangle;
 a | b | c
---+---+---
 3 | 4 | 5
(1 row)

-- 创建生成列，生成列表达式引用已定义的生成列。
m_db=# CREATE TABLE triangle (
  a DOUBLE,
  b DOUBLE,
  c DOUBLE AS (SQRT(a * a + b * b)) STORED,
  d DOUBLE AS (c/100) VIRTUAL
);

m_db=# INSERT INTO triangle(a, b) VALUES (3, 4);

m_db=# SELECT * FROM triangle;
 a | b | c |  d
---+---+---+------
 3 | 4 | 5 | 0.05
(1 row)
```

建表添加约束示例

- **非空约束**

  非空约束的字段，如果在添加数据时没有指定值，就会报错。可以为表中多个字段添加非空约束。

```
--建表并给id字段添加非空约束。
m_db=# CREATE TABLE test_t4(
    id       CHAR(7) NOT NULL,
    name     VARCHAR(20),
    province VARCHAR(60),                       --省
    country   VARCHAR(30) DEFAULT 'China'       --国籍
);

--插入数据时,如果不指定id的值或者id为NULL，会触发非空约束,导致插入失败。
m_db=# INSERT INTO test_t4 (name,province) VALUES ('scott','Shanghai');
ERROR:  null value in column "id" violates not-null constraint
DETAIL:  Failing row contains (null, scott, Shanghai, China)

--删除表。
m_db=# DROP TABLE test_t4;
```

- **唯一约束**

关键字UNIQUE给字段添加一个唯一约束，插入数据时该字段如有重复则触发约束，多个NULL不算重复，添加唯一约束时，会自动增加一个唯一索引。可以为表中多个字段添加唯一约束。

```
--建表添加唯一约束。
m_db=# CREATE TABLE test_t5(
    id       CHAR(7) UNIQUE,
    name     VARCHAR(20),
    province VARCHAR(60),                       --省
    country   VARCHAR(30) DEFAULT 'China'       --国籍
);

--也可以用如下写法，人工为唯一约束命名，以及为多个字段添加约束。
m_db=# CREATE TABLE test_t6(
    id       CHAR(7),
    name     VARCHAR(20),
    province VARCHAR(60),                        --省
    country   VARCHAR(30) DEFAULT 'China',       --国籍
    CONSTRAINT unq_test_id UNIQUE (id,name)
);
--插入id重复的数据，触发约束，导致插入失败。
m_db=# INSERT INTO test_t5(id) VALUES('0000010');
INSERT 0 1
m_db=# INSERT INTO test_t5(id) VALUES('0000010');
ERROR:  duplicate key value violates unique constraint "test_t5_id_key"
DETAIL:  Key (id)=(0000010) already exists.
--多次插入id是NULL的数据不会触发约束。
m_db=# INSERT INTO test_t5(id) VALUES (NULL);
INSERT 0 1
m_db=# INSERT INTO test_t5(id) VALUES (NULL);
INSERT 0 1

m_db=# SELECT * FROM test_t5;
   id    | name | province | country
---------+------+----------+---------
 0000010 |      |          | China
         |      |          | China
         |      |          | China

--删除表。
m_db=# DROP TABLE test_t5;
m_db=# DROP TABLE test_t6;
```

- **主键约束**

  关键字PRIMARY KEY给字段添加主键约束，要求字段唯一且不为空。添加主键约束时自动为该表创建唯一索引，也会为该字段自动增加一个非空约束。

  每个表里面只能定义一个主键约束，不能定义多个。

```
--建表添加主键约束。
m_db=# CREATE TABLE test_t6(
    id       CHAR(7) PRIMARY KEY,
    name     VARCHAR(20),
    province VARCHAR(60),                       --省
    country   VARCHAR(30) DEFAULT 'China'       --国籍
);
m_db=# INSERT INTO test_t6 (id,name,province) VALUES ('0000001','july','Beijing');

--也可以用如下写法，人工为唯一约束命名，以及为多个字段添加约束。
m_db=# CREATE TABLE test_t7(
    id       CHAR(7),
    name     VARCHAR(20),
    province VARCHAR(60),                       --省
    country   VARCHAR(30) DEFAULT 'China',       --国籍
    CONSTRAINT pk_test_t6_id PRIMARY KEY (id,name)
);
--插入id为NULL的数据，触发约束。
m_db=# INSERT INTO test_t6 (id,name,province) VALUES (NULL,'july','Beijing');
ERROR:  null value in column "id" violates not-null constraint
DETAIL:  Failing row contains (null, july, Beijing, China).
```



```
--插入id重复的数据，触发约束。
m_db=# INSERT INTO test_t6 (id,name,province) VALUES ('0000001','ben','Shanghai');
ERROR:  duplicate key value violates unique constraint "test_t6_pkey"
DETAIL:  Key (id)=(0000001) already exists.
--删除表。
m_db=# DROP TABLE test_t6;
m_db=# DROP TABLE test_t7;
```

- **检查约束**

  关键字CHECK给字段添加检查约束，在检查约束中必须引用表中的一个或多个字段，并且表达式返回结果必须是一个布尔值。在表达式中不能包含子查询。对同一个字段可以同时定义检查约束和非空约束。

```
--建表，添加检查约束。
m_db=# CREATE TABLE test_t8 (
    id       CHAR(7),
    name     VARCHAR(20),
    age      INT CHECK(age > 0 AND age < 150)
);

--也可以使用如下SQL，人工为检查约束命名以及为一个或者多个字段添加检查约束。
m_db=# CREATE TABLE test_t9 (
    id       CHAR(7),
    name     VARCHAR(20),
    age      INT,
    CONSTRAINT chek_test_t8_age CHECK(age > 0 AND age < 150)
);
--插入不符合表达式的值，会触发检查约束导致插入失败。
m_db=# INSERT INTO test_t8 (id,name,age) VALUES ('0000007','scott',200);
ERROR:  new row for relation "test_t8" violates check constraint "test_t8_age_check"
DETAIL:  N/A
--删除表。
m_db=# DROP TABLE test_t8;
m_db=# DROP TABLE test_t9;
```

- **外键约束**

  当两个表包含一个或多个公共列时，可以通过外键约束来强制关联两个表。

  - FOREIGN KEY：用来指定该表中和被引用的表有关系的字段。
  - REFERENCES：用来指定被引用的表和原表有关系的字段。

  外键约束的特点:

  - 定义为外键约束的字段中只能包含相应的其他表中引用字段的值或NULL。
  - 可以为一个字段或者多个字段定义外键约束。
  - 定义了外键约束的字段和相应的引用字段可以存在同一个表中，称为自引用。
  - 对同一个字段可以同时定义外键和非空约束。
  - 主表中被应用的列，必须有主键约束或唯一约束。

```
--创建部门表。
m_db=# CREATE TABLE dept(
    deptno INT PRIMARY KEY,
    loc VARCHAR(200)
);

--创建员工表，添加外键约束。
m_db=# CREATE TABLE emp(
    empno INT,
    name VARCHAR(50),
    deptno INT,
    CONSTRAINT fk_emp FOREIGN KEY (deptno) REFERENCES dept(deptno)
);
--部门表插入数据。
m_db=# INSERT INTO dept VALUES (10,'Beijing');
```



```
m_db=# INSERT INTO dept VALUES (20,'Beijing');
m_db=# INSERT INTO dept VALUES (30,'Shanghai');

--员工表中插入deptno在部门表中能找到的数据。
m_db=# INSERT INTO emp VALUES (1,'Bob',10);

--员工表中插入deptno为NULL的数据。
m_db=# INSERT INTO emp VALUES (2,'Scott',NULL);

--员工表中插入deptno在部门表中找不到的数据。
m_db=# INSERT INTO emp VALUES (1,'Jack',999);
ERROR:  insert or update on table "emp" violates foreign key constraint "fk_emp"
DETAIL:  Key (deptno)=(999) is not present in table "dept".
--查看数据。
m_db=# SELECT * FROM emp;
 empno | name  | deptno
-------+-------+--------
     1 | Bob   |     10
     2 | Scott |
(2 rows)
--删除表。
m_db=# DROP TABLE emp;
m_db=# DROP TABLE dept;
```

相关链接

ALTER TABLE，**DROP TABLE**

优化建议

- UNLOGGED
  - UNLOGGED表和表上的索引因为数据写入时不通过WAL日志机制，写入速度远高于普通表。因此，可以用于缓冲存储复杂查询的中间结果集，增强复杂查询的性能。
  - UNLOGGED表无主备机制，在系统故障或异常断点等情况下，会有数据丢失风险，因此，不可用来存储基础数据。
- TEMPORARY
  - 非全局临时表只在当前会话可见，会话结束后会自动删除。
- LIKE
  - 新表自动从这个表中继承所有字段名及其数据类型和非空约束，新表与源表之间在创建动作完毕之后是完全无关的。
- ORIENTATION ROW
  - 创建行存表，行存储适合于OLTP业务，此类型的表上交互事务比较多，一次交互会涉及表中的多个列，用行存查询效率较高。

4.4.2.8.17 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）、哈希分区（Hash/Key Partitioning）、列表分区（List Partitioning）、列分区（Range/List Columns Partitioning）等。目前行存表支持范围分区、哈希分区、列表分区。

范围分区（Range Partitioning）：

- 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：最常用的分区策略为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。

哈希分区（Hash/Key Partitioning）：

- 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（List Partitioning）：

- 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，否则返回报错和提示信息。

分区存在以下优势：

- 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，连续扫描该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该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（哈希分区不支持删除分区）。

注意事项

- 当为分布表添加唯一约束或主键约束时，如果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。
- 目前哈希分区仅支持单列构建分区键，暂不支持多列构建分区键。
- 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。

- 考虑性能影响，一般建议单表最大分区数不超过2000，子分区数 *（local索引个数 + 1）不超过10000。
- 当分区数太多导致内存不足时，会间接导致性能急剧下降。
- 指定分区语句目前不能走全局索引扫描。
- 不支持如下数据类型作为分区键、二级分区键：BLOB、TINYBLOB、MEDIUMBLOB、LONGBLOB、TEXT、TINYTEXT、MEDIUMTEXT、LONGTEXT、BIT。

语法格式

```
CREATE TABLE [ IF NOT EXISTS ] partition_table_name
 ( [
   { column_name data_type [ {CHARACTER SET | CHAR SET | CHARSET} charset ] [ COLLATE collation ]
[ column_constraint [ ... ] ]
   | table_constraint
   | LIKE source_table
] )
   [ table_option [ [ , ] ... ] ]
   [ WITH ( {storage_parameter = value} [, ... ] ) ]
   [ TABLESPACE tablespace_name ]
    PARTITION BY {
      {RANGE [COLUMNS] (partition_key) [ PARTITIONS integer ] ( partition_less_than_item [, ... ] )
[partition_options] } |
      {RANGE [COLUMNS] (partition_key) [ PARTITIONS integer ] ( partition_start_end_item [, ... ] )
[partition_options] } |
      {LIST [COLUMNS] (partition_key)  [ PARTITIONS integer ] ( PARTITION partition_name VALUES [IN]
(list_values)  [TABLESPACE [=] tablespace_name][, ... ] [partition_options]  ) } |
      {{ HASH | KEY } (partition_key) [ PARTITIONS integer ] ( PARTITION partition_name ] [TABLESPACE
[=] tablespace_name][partition_options] [, ... ] ) }
   };
```

- 其中table_option为：

  ```
  AUTO_INCREMENT [=] value
    | [DEFAULT] CHARACTER SET [=] charset_name
    | [DEFAULT] COLLATE [=] collation_name
    | COMMENT [=] 'string'
    | ENGINE [=] engine_name
    | ROW_FORMAT [=] {DEFAULT | DYNAMIC | FIXED | COMPRESSED | REDUNDANT | COMPACT}
  }
  ```

  说明

  - 同一个地方多次设置COMMENT时仅最后一个生效。
  - 同一个地方多次设置AUTO_INCREMENT时仅最后一个生效。
  - 同一个地方多次设置COLLATE时仅最后一个生效。
  - ENGINE、ROW_FORMAT使用时语法不报错也没有提示，但实际不生效。

- 列约束column_constraint：

  ```
  [ CONSTRAINT constraint_name ]
  { NOT NULL |
    NULL |
    CHECK ( expression ) |
    DEFAULT default_expr |
    ON UPDATE update_expr |
    [GENERATED ALWAYS] AS ( generation_expr ) [STORED | VIRTUAL] |
    AUTO_INCREMENT |
    COMMENT 'string' |
    UNIQUE [KEY] index_parameters |
    PRIMARY KEY index_parameters |
    REFERENCES reftable [ ( refcolumn ) ] [ MATCH FULL  | MATCH SIMPLE ]
          [ ON DELETE action ] [ ON UPDATE action ] }
  ```

- 表约束table_constraint：

  ```
  [ CONSTRAINT [ constraint_name ] ]
  { CHECK ( expression ) |
  ```



```
  UNIQUE [ index_name ][ USING method ] ( { column_name [ ASC | DESC ] } [, ... ] )
index_parameters |
  PRIMARY KEY [ USING method ] ( { column_name [ ASC | DESC ] } [, ... ] ) index_parameters |
  FOREIGN KEY [ index_name ] ( column_name [, ... ] ) REFERENCES reftable [ ( refcolumn [, ... ] ) ]
      [ MATCH FULL  | MATCH SIMPLE ] [ ON DELETE action ] [ ON UPDATE action ] }
```

- 索引存储参数index_parameters：

```
[ WITH ( {storage_parameter = value} [, ... ] ) ]
[ USING method ]
[ COMMENT 'string']
```

- 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 | MAXVALUE})} |
      {START(partition_value)} |
      {END({partition_value | MAXVALUE})}
} [TABLESPACE [=] tablespace_name]
```

- partition_options为：

```
ENGINE [=] ’ string’ |
STORAGE ENGINE [=] ’ string’
```

  当前仅语法上不报错，但此语法无实际作用。

- 其中update_expr为：

```
{ CURRENT_TIMESTAMP | LOCALTIMESTAMP | NOW() }
```

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**标识符说明**。

- **column_name**

  新表中要创建的字段名。

  取值范围：字符串，要符合**标识符说明**。

- **data_type**

  字段的数据类型。

- **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。constraint_name为可选项

- **index_name**

  索引名。

须知

- 对于外键约束，constraint_name和index_name同时指定时，索引名为constraint_name。
- 对于唯一键约束，constraint_name和index_name同时指定时，索引名以index_name。

- **USING method**

  指定创建索引的方法。

  取值范围参考**参数说明**中的USING method。

须知

目前只支持索引方法为btree。

- **ASC | DESC**

  ASC表示指定按升序排序（默认）。DESC指定按降序排序。

- **LIKE source_table**

  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。

- **AUTO_INCREMENT [=] value**

  这个子句为自动增长列指定一个初始值，value必须为正整数，不得超过2^{127}-1。

- **COMMENT [=] 'string'**
  - COMMENT [=] 'string'子句表示给表添加注释。
  - 在column_constraint中的COMMENT 'string'表示给列添加注释。
  - 在table_constraint中的COMMENT 'string'表示给主键和唯一键对应的索引添加注释。

  具体请参见：**•COMMENT[=]'string'**。

- **{CHARACTER SET | CHAR SET | CHARSET} charset**

  指定表字段的字符集。单独指定时会将字段的字符序设置为指定的字符集的默认字符序。

- **COLLATE collation**

  COLLATE子句指定列的排序规则（字符序）（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。支持utf8mb4_bin、utf8mb4_general_ci、utf8mb4_unicode_ci、binary、gbk_chinese_ci、gbk_bin、gb18030_chinese_ci、gb18030_bin字符序。具体请参见**表级字符集和字符序**

说明

- 仅字符类型支持指定字符集，指定为BINARY字符集或字符序实际是将字符类型转化为对应的二进制类型，若类型映射不存在则报错。当前仅有TEXT类型转化为BLOB的映射。
- 字段字符集或字符序未显式指定时，若指定了表的默认字符集或字符序，字段字符集和字符序将从表上继承。
- 除BINARY字符集、字符序外，当前仅支持指定与数据库编码相同的字符集。

- **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10~100的百分数。在Ustore存储引擎下，该值得默认值为92，在Astore存储引擎下默认值为100（完全填充）。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。

  取值范围：10~100

  – ORIENTATION

  决定了表的数据的存储方式。当前M-compatibility模式数据库下只支持行存储方式。

  取值范围：

  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Inplace-Update存储。

  – COMPRESSION

  行存表不支持压缩。

  – segment

  使用段页式的方式存储。本参数仅支持行存表。不支持临时表、unlog表。

  取值范围：on/off

  默认值：off

- statistic_granularity

  记录该表在分析统计信息时的默认PARTITION_MODE。此参数对非分区表设置无效。

  取值范围：请参见PARTITION_MODE选项说明。

  默认值：AUTO。

  表 4-22 PARTITION_MODE 选项说明

PARTITION_MODE选项	含义
ALL	收集整表、一级分区的统计信息。
GLOBAL	收集整表的统计信息。
PARTITION	收集一级分区的统计信息。
GLOBAL AND PARTITION	收集整表、一级分区的统计信息。
ALL COMPLETE	收集整表、一级分区的统计信息。
AUTO	缺省值。

- **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。

  注：需要在非M-Compatibility数据库中创建或删除tablespace。

- **PARTITION BY RANGE [COLUMNS] (partition_key)**

  创建范围分区。partition_key为分区键的名称。

  “PARTITION BY RANGE COLUMNS” 语义同 “PARTITION BY RANGE”。

  （1）对于从句是VALUES LESS THAN的语法格式：

  须知

  对于从句是VALUES LESS THAN的语法格式，范围分区策略的分区键最多支持16列。

  该情形下，分区键支持的数据类型为：TINYINT、SMALLINT、INTEGER、BIGINT、TINYINT UNSIGNED、SMALLINT UNSIGNED、INTEGER UNSIGNED、BIGINT UNSIGNED、DATE、YEAR、CHAR[(n)]、VARCHAR[(n)]、MEDIUMINT、MEDIUMINT UNSIGNED、BINARY[(n)]、VARBINARY(n)、DATETIME[(p)]、TIME[(p)]。

  （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、SMALLINT、INTEGER、BIGINT、TINYINT UNSIGNED、SMALLINT UNSIGNED、INTEGER UNSIGNED、BIGINT UNSIGNED、DATE。

- **PARTITION partition_name VALUES LESS THAN {({ partition_value | MAXVALUE } [,...])** | MAXVALUE } [TABLESPACE tablespace_name]

  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。
  - 不在括号内的MAXVALUE只能有一个分区键。
  - partition_value不支持使用表达式。

- **PARTITION partition_name {START (partition_value) END (partition_value|MAXVALUE)} | {START(partition_value)} | {END (partition_value | MAXVALUE)}** [TABLESPACE tablespace_name]

  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。
  - 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

1. 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。
2. 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）；
3. 在创建分区表时START END与LESS THAN语法不可混合使用。
4. 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。

- **PARTITION BY LIST [COLUMNS] (partition_key)**

  创建列表分区。partition_key为分区键的名称。

  “PARTITION BY LIST COLUMNS” 语义同 “PARTITION BY LIST”。

  - 对于partition_key，不支持表达式。当不指定二级分区时，列表分区策略的分区键最多支持16列，当指定二级分区时，列表分区策略的分区键最多支持1列，且不支持表达式。
  - 对于从句是VALUES [IN] (list_values)的语法格式，list_values中包含了对应分区存在的键值，每个分区的键值数量不超过64个。
  - 从句"VALUES IN"语义同"VALUES"。

  分区键支持的数据类型为：TINYINT、SMALLINT、INTEGER、BIGINT、TINYINT UNSIGNED、SMALLINT UNSIGNED、INTEGER UNSIGNED、BIGINT UNSIGNED、DATE、YEAR、CHAR[(n)]、VARCHAR[(n)]、MEDIUMINT、MEDIUMINT UNSIGNED、BINARY[(n)]、VARBINARY(n)、DATETIME[(p)]、TIME[(p)]。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，且不支持表达式。

  分区键支持的数据类型为：TINYINT、SMALLINT、INTEGER、BIGINT、TINYINT UNSIGNED、SMALLINT UNSIGNED、INTEGER UNSIGNED、BIGINT UNSIGNED、DATE、YEAR、CHAR[(n)]、VARCHAR[(n)]、MEDIUMINT、MEDIUMINT UNSIGNED、BINARY[(n)]、VARBINARY(n)、DATETIME[(p)]、TIME[(p)]、TIMESTAMP[(p)]、NUMERIC[(p[,s])]、FLOAT4[(p, s)]、FLOAT8[(p,s)]。分区个数不能超过1048575个。

- **PARTITION BY KEY(partition_key)**

  语义同“PARTITION BY HASH(partition_key)”。

- **PARTITIONS integer**

  指定分区个数。

  integer为分区数，必须为大于0的整数，且不得大于1048575。

  - 当在RANGE和LIST分区后指定此子句时，必须显式定义每个分区，且定义分区的数量必须与integer值相等。只能在RANGE和LIST分区后指定此子句。
  - 当在HASH和KEY分区后指定此子句时，若不列出各个分区定义，将自动生成integer个分区，自动生成的分区名为“p+数字”，数字依次为0到integer-1，分区的表空间默认为此表的表空间；也可以显式列出每个分区定义，此时定义分区的数量必须与integer值相等。若既不列出分区定义，也不指定分区数量，将创建唯一一个分区。

- **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)**

  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。

- **DEFAULT default_expr**

  - 使用DEFAULT子句给字段指定缺省表达式，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 。
  - 当未在缺省表达式外嵌套括号时，支持指定以下内容：常量、带正负号的数值常量、**update_expr**。
  - 当在缺省表达式外嵌套括号时，支持指定以下内容：常量、带正负号的数值常量、**update_expr**、CURRENT_TIME/CURTIME函数、CURRENT_DATE/CURDATE函数（CURRENT_TIME/CURRENT_DATE支持不带括号形式的调用）。
  - 仅支持在TIMESTAMP、DATETIME类型的字段上指定update_expr作为默认值，且字段的精度与update_expr的精度须保持一致。

- **ON UPDATE update_expr**

  ON UPDATE子句为字段的一种属性约束。

  当对表中某元组执行UPDATE操作时，若更新字段的新值和表中旧值不相同，则表中该元组上具有该属性且不在更新字段内的字段值自动更新为当前时间戳；若更新字段的新值和表中旧值相同，则表中该元组上具有该属性且不在更新字段内的字段值不变，保持原有值；若具有该属性的字段在更新字段内，则对应这些字段值直接按指定更新的值更新。

说明

- 语法上update_expr支持CURRENT_TIMESTAMP 、LOCALTIMESTAMP 、LOCALTIME、NOW()四种关键字，也支持关键字带括号指定或不指定精度。例如：ON UPDATE CURRENT_TIMESTAMP()、ON UPDATE CURRENT_TIMESTAMP(5)、ON UPDATE LOCALTIMESTAMP()、ON UPDATE LOCALTIMESTAMP(6)等。不带括号或空括号时精度为0，其中NOW关键字不支持不带括号。四种关键字互为同义词，属性效果相同。
- 该属性支持在如下类型的列上指定: TIMESTAMP、DATETIME。
- 该属性指定的精度和对应列上类型指定的精度必须一致，否则会触发报错。例如：CREATE TABLE t1 (col1 timestamp(6) ON UPDATE CURRENT_TIMESTAMP(6)); 若精度不一致，会产生ERROR: Invalid ON UPDATE clause for "col1" 报错。
- 该属性和生成列约束不能同时指定同一列。
- 分区表中的分区键不支持指定该属性。

- **[GENERATED ALWAYS] AS (generation_expr) [STORED | VIRTUAL]**

  该子句将字段创建为生成列，通过指定STORED和VIRTUAL关键字表明生成列的列值存储方式：

  - STORED：创建存储生成列，列值需要存储，在插入或更新元组时，由生成列表达式计算存储生成列的列值。
  - VIRTUAL：创建虚拟生成列，列值不需要存储，在查询语句涉及虚拟生成列时，由生成列表达式计算列值。

说明

- STORED和VIRTUAL关键字可省略，缺省为STORED，在参数m_format_dev_version为s2条件下缺省为VIRTUAL。
- 生成表达式不能以任何方式引用当前行以外的其他数据。生成表达式支持引用早定义于当前生成列的其他生成列，不能引用系统列。生成列表达式不能引用系统变量、用户自定义变量、存储过程中的变量。生成列表达式不支持引用自增列，且生成列不支持定义自增列属性。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、窗口函数和自定义函数。生成表达式调用的函数只能是不可变（IMMUTABLE）函数。
- 不支持为生成列指定默认值。
- 存储生成列支持作为分区键的一部分，当作为分区键的一部分时，不支持ALTER TABLE修改存储生成列。
- 虚拟生成列不支持作为分区键的一部分。
- 虚拟生成列不支持被外键约束引用。
- 不支持在虚拟生成列上创建索引。
- 存储生成列不能和ON UPDATE约束子句的CASCADE,SET NULL,SET DEFAULT动作同时指定。存储生成列不能和ON DELETE约束子句的SET NULL,SET DEFAULT动作同时指定。
- 在参数m_format_dev_version为s2条件下，存储生成列的基列不能和ON UPDATE或者ON DELETE约束子句的CASCADE, 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。删除生成列依赖的基列，生成列被自动删除。在参数m_format_dev_version为s2条件下，删除生成列依赖的基列之前需要先删除对应的生成列。
- 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。向可被更新的视图中插入生成列数据时，不能为生成列指定值，但是可以指定关键字DEFAULT。
- 当存储生成列带CHECK约束时，在参数m_format_dev_version为s2的条件下，ALTER TABLE修改存储生成列定义，如果存在生成列不满足CHECK约束，ALTER TABLE语句执行失败，系统报ERROR。
- 生成列的权限控制和普通列一样。
- 不开启精度传递开关（m_format_behavior_compat_options不开启enable_precision_decimal选项）时创建的带有生成列的表，与开启精度传递开关后创建的带有生成列的表在浮点数的比较结果等方面可能出现不一致的情况。如果用户需要在开启精度传递开关的情况下使用未开启精度开关时创建的带有生成列的表，建议在开启精度传递开关后使用ALTER TABLE语法重新定义生成列。

```
m_db=# SET m_format_behavior_compat_options='';
SET
m_db=# CREATE TABLE mm1(a float(10, 4), b float(5, 3), c boolean AS ((a/b)=1.7142858)
STORED);
CREATE TABLE
m_db=# INSERT INTO mm1 VALUES(1.2, 0.7);
INSERT 0 1
m_db=# SELECT * FROM mm1;
  a    |   b   | c
--------+-------+---
 1.2000 | 0.700 | 0
(1 row)

m_db=# SET m_format_behavior_compat_options='enable_precision_decimal';
SET
m_db=# CREATE TABLE mm2(a float(10, 4), b float(5, 3), c boolean AS ((a/b)=1.7142858)
STORED);
CREATE TABLE
m_db=# INSERT INTO mm1 VALUES(1.2, 0.7);
INSERT 0 1
m_db=# INSERT INTO mm2 VALUES(1.2, 0.7);
INSERT 0 1
m_db=# SELECT * FROM mm1;
  a    |   b   | c
```



```
--------+-------+---
 1.2000 | 0.700 | 0
 1.2000 | 0.700 | 1
(2 rows)

m_db=# SELECT * FROM mm2;
   a    |   b   | c
--------+-------+---
 1.2000 | 0.700 | 0
(1 row)

m_db=# DROP TABLE mm1, mm2;
DROP TABLE
```

- **AUTO_INCREMENT**
  指定列为自动增长列。
  详见：**•AUTO_INCREMENT**。
- **列级唯一约束：UNIQUE [KEY] index_parameters**
  UNIQUE约束表示表里的一个或多个字段的组合必须在全表范围内唯一。
  UNIQUE KEY与UNIQUE语义相同。
- **表级唯一约束：UNIQUE [INDEX | KEY][index_name][USING method] ({{ column_name [(length)] | (expression) } [ASC | DESC] }[, ...]) 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。
  column_name(length)是前缀键，详见：**•column_name (length)**。
  index_name为索引名。

须知

对于唯一键约束，constraint_name和index_name同时指定时，索引名为index_name。

- **列级主键约束：[PRIMARY] KEY index_parameters**
  表级主键约束：PRIMARY KEY [index_name] [USING method] ({ column_name [ASC | DESC] } [, ...] ） 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。

范围分区

- VALUES LESS THAN

```
--创建分区表test_range1。
m_db=# CREATE TABLE test_range1(
    id INT,
    info VARCHAR(20)
) PARTITION BY RANGE (id) (
    PARTITION p1 VALUES LESS THAN (200),
    PARTITION p2 VALUES LESS THAN (400),
    PARTITION p3 VALUES LESS THAN (600),
    PARTITION pmax VALUES LESS THAN (MAXVALUE)
);

--插入1000条数据
m_db=# INSERT INTO test_range1 VALUES(GENERATE_SERIES(1,1000),'abcd');
```



```
--查看p1分区的行数199条，[1,200)。
m_db=# SELECT COUNT(*) FROM test_range1 PARTITION (p1);
 count
-------
   199
(1 row)

--查看p2分区的行数200条，[200,400)。
m_db=# SELECT COUNT(*) FROM test_range1 PARTITION (p2);
 count
-------
   200
(1 row)

--查看分区信息。
m_db=# SELECT a.relname, a.boundaries
FROM pg_partition a
WHERE a.parentid = 'test_range1'::regclass and a.parttype = 'p';
 relname | boundaries
---------+------------
 pmax    | {NULL}
 p3      | {600}
 p2      | {400}
 p1      | {200}
(4 rows)

--删除
m_db=# DROP TABLE test_range1;
```

- START END

```
--创建分区表。
m_db=# CREATE TABLE test_range2(
    id INT,
    info VARCHAR(20)
) PARTITION BY RANGE (id) (
    PARTITION p1 START(1) END(600),
    PARTITION p2 START(600) END(800),
    PARTITION pmax START(800) END(MAXVALUE)
);

--查看分区信息。
m_db=# SELECT relname, boundaries FROM pg_partition WHERE parentid = 'test_range2'::regclass
AND parttype = 'p' ORDER BY 1;
relname | boundaries
---------+------------
 p1_0    | {1}
 p1_1    | {600}
 p2      | {800}
 pmax    | {NULL}
(4 rows)

--删除。
m_db=# DROP TABLE test_range2;
```

列表分区

```
--创建列表分区表。
m_db=# CREATE TABLE test_list ( NAME VARCHAR ( 50 ), area VARCHAR ( 50 ) )
PARTITION BY LIST (area) (
    PARTITION p1 VALUES ('Beijing'),
    PARTITION p2 VALUES ('Shanghai'),
    PARTITION p3 VALUES ('Guangzhou'),
    PARTITION p4 VALUES ('Shenzhen'),
    PARTITION pdefault VALUES (DEFAULT)
);

--插入数据。
m_db=# INSERT INTO test_list VALUES ('bob', 'Shanghai'),('scott', 'Sichuan');
```



```
--查询分区数据。
m_db=# SELECT * FROM test_list PARTITION (p2);
 name |   area
------+----------
 bob  | Shanghai
(1 row)
m_db=# SELECT * FROM test_list PARTITION (pdefault);
 name  |  area
-------+---------
 scott | Sichuan
(1 row)

--删除。
m_db=# DROP TABLE test_list;
```

哈希分区

```
--创建哈希分区表，指定分区数。
m_db=# CREATE TABLE test_hash1(c1 int) PARTITION BY HASH(c1) PARTITIONS 3;

--创建哈希分区表，并指定分区名。
m_db=# CREATE TABLE test_hash2(c1 int) PARTITION BY HASH(c1)(
    PARTITION pa,
    PARTITION pb,
    PARTITION pc
);

--查看分区信息。
m_db=# SELECT b.relname AS table_name,
       a.relname AS partition_name
FROM pg_partition a,
     pg_class b
WHERE b.relname LIKE 'test_hash%'
  AND a.parttype = 'p'
  AND a.parentid = b.oid;
 table_name | partition_name
------------+----------------
 test_hash1 | p2
 test_hash1 | p1
 test_hash1 | p0
 test_hash2 | pc
 test_hash2 | pb
 test_hash2 | pa
(6 rows)

--删除。
m_db=# DROP TABLE test_hash1,test_hash2;
```

相关链接

ALTER TABLE PARTITION，**DROP TABLE**

4.4.2.8.18 CREATE TABLE SUBPARTITION

功能描述

创建二级分区表。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。对于二级分区表，顶层节点表和一级分区都是逻辑表，不存储数据，只有二级分区（叶子节点）存储数据。

二级分区表的分区方案是由两个一级分区的分区方案组合而来的，一级分区的分区方案详见章节**CREATE TABLE PARTITION**。

二级分区表组合方案在M-Compatibility模式下只支持4种：Range-Hash分区、Range-Key分区、List-Hash分区、List-Key分区。目前二级分区仅支持行存表。

注意事项

- 二级分区表有两个分区键，每个分区键只能支持1列，且不支持表达式。
- 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。如果指定创建local唯一索引，必须包含所有分区键。
- 创建二级分区表时，如果在其一级分区下不显示指定二级分区，会自动创建一个同范围的二级分区。
- 二级分区表的二级分区（叶子节点）个数不能超过1048575个，一级分区无限制，但一级分区下面至少有一个二级分区。
- 二级分区表的总分区数（包括一级分区和二级分区）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，二级分区表使用内存大致为（总分区数 * 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。
- 考虑性能影响，一般建议单表最大分区数不超过2000，子分区数 *（local索引个数 + 1） 不超过10000。
- 当分区数太多导致内存不足时，会间接导致性能急剧下降。
- 二级分区表只支持行存。
- 不支持cluster。
- 指定分区查询时，如select * from tablename partition/subpartition (partitionname)，关键字partition和subpartition注意不要写错。如果写错，查询不会报错，这时查询会变为对表起别名进行查询。
- 不支持密态数据库、账本数据库。
- 对于二级分区表PARTITION/SUBPARTITION FOR (VALUES)语法，VALUES只能是常量。
- 对于二级分区表PARTITION/SUBPARTITION 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。
- 指定分区语句目前不能走全局索引扫描。

语法格式

```
CREATE TABLE [ IF NOT EXISTS ] subpartition_table_name
(
   { column_name data_type [ COLLATE collation ] [ column_constraint [ ... ] ]
   | table_constraint
   | LIKE source_table
)
   [ table_option [ [ , ] ... ] ]
   [ WITH ( {storage_parameter = value} [, ... ] ) ]
   [ TABLESPACE tablespace_name ]
   PARTITION BY { RANGE [ COLUMNS ] | LIST [ COLUMNS ] } (partition_key) [ PARTITIONS integer ]
      SUBPARTITION BY { HASH | KEY} (subpartition_key) [ SUBPARTITIONS integer ]
   (
      PARTITION partition_name1 [ VALUES LESS THAN {(val1) | MAXVALUE} | VALUES [IN] (val1[, …]) ]
      [ TABLESPACE [=] tablespace ]
      [(
        { SUBPARTITION subpartition_name1
          [ TABLESPACE [=] tablespace ] } [, ...]
      )] [partition_options] [, ...]
   );
```


- 其中table_option为：

```
AUTO_INCREMENT [=] value
  | [DEFAULT] CHARACTER SET [=] charset_name
  | [DEFAULT] COLLATE [=] collation_name
  | COMMENT [=] 'string'
  | ENGINE [=] engine_name
  | ROW_FORMAT [=] {DEFAULT | DYNAMIC | FIXED | COMPRESSED | REDUNDANT | COMPACT}
}
```

  说明

  - 同一个地方多次设置COMMENT时仅最后一个生效。
  - 同一个地方多次设置AUTO_INCREMENT时仅最后一个生效。
  - 同一个地方多次设置COLLATE时仅最后一个生效。
  - ENGINE、ROW_FORMAT使用时语法不报错也没有提示，但实际不生效。

- 其中列约束column_constraint为：

```
[ CONSTRAINT constraint_name ]
{ NOT NULL |
  NULL |
  CHECK ( expression ) |
  DEFAULT default_expr |
  ON UPDATE now_expr |
  [ GENERATED ALWAYS ] AS ( generation_expr ) [STORED | VIRTUAL] |
  UNIQUE [KEY] index_parameters |
  [ PRIMARY ] KEY index_parameters |
  REFERENCES reftable [ ( refcolumn ) ] [ MATCH FULL | MATCH SIMPLE ]
     [ ON DELETE action ] [ ON UPDATE action ] }
```

- 其中表约束table_constraint为：

```
[ CONSTRAINT [ constraint_name ] ]
{ CHECK ( expression ) |
  UNIQUE [ INDEX | KEY ] [ index_name ][ USING method ] ( { { column_name [ ( length ) ] |
 ( expression ) } [ ASC | DESC ] } [, ... ] ) index_parameters [USING method| COMMENT 'string']|
  PRIMARY KEY [index_name][ USING method ] ( { column_name  }[ ASC | DESC ] } [, ... ] )
index_parameters [USING method| COMMENT 'string']|
  FOREIGN KEY [ index_name ] ( column_name [, ... ] ) REFERENCES reftable [ (refcolumn [, ... ] ) ]
     [ MATCH FULL | MATCH SIMPLE ] [ ON DELETE action ] [ ON UPDATE action ]
  }
```

- 其中索引参数index_parameters为：

```
[ WITH ( {storage_parameter = value} [,...]
[ USING INDEX TABLESPACE tablespace_name ]
```

- partition_options为：

```
ENGINE [=] ’ string’ |
STORAGE ENGINE [=] ’ string’
```

  当前仅语法上不报错，但此语法无实际作用。

- 其中update_expr为：

```
{ CURRENT_TIMESTAMP | LOCALTIMESTAMP | NOW() }
```

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。

- **subpartition_table_name**

  二级分区表的名称。

  取值范围：字符串，要符合**标识符说明**。

- **column_name**

  新表中要创建的字段名。

取值范围：字符串，要符合**标识符说明**。

- **data_type**

  字段的数据类型。

- **COLLATE collation**

  COLLATE子句指定列的排序规则（字符序）（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。支持utf8mb4_bin、utf8mb4_general_ci、utf8mb4_unicode_ci、binary、gbk_chinese_ci、gbk_bin、gb18030_chinese_ci、gb18030_bin字符序。具体请参见**表级字符集和字符序**

  说明

  - 仅字符类型支持指定字符集，指定为BINARY字符集或字符序实际是将字符类型转化为对应的二进制类型，若类型映射不存在则报错。当前仅有TEXT类型转化为BLOB的映射。
  - 除BINARY字符集、字符序外，当前仅支持指定与数据库编码相同的字符集。
  - 字段字符集或字符序未显式指定时，若指定了表的默认字符集或字符序，字段字符集和字符序将从表上继承。

- **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。

- **index_name**

  索引名。

  须知

  - 对于外键约束，constraint_name和index_name同时指定时，索引名为constraint_name。
  - 对于唯一键约束，constraint_name和index_name同时指定时，索引名以index_name。

- **USING method**

  指定创建索引的方法。

  取值范围参考**参数说明**中的USING method。

  须知

  - 目前只支持索引方法为btree

- **ASC | DESC**

  ASC表示指定按升序排序（默认）。DESC指定按降序排序。

- **LIKE source_table**

LIKE子句声明一个表，新表自动从这个表中继承所有字段名及其数据类型和非空约束。

新表与源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。

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。

- **AUTO_INCREMENT [=] value**

  这个子句为自动增长列指定一个初始值，value必须为正整数，不得超过2^{127}-1。

- **COMMENT [=] 'string'**
  - COMMENT [=] 'string'子句表示给表添加注释。
  - 在column_constraint中的COMMENT 'string'表示给列添加注释。
  - 在table_constraint中的COMMENT 'string'表示给主键和唯一键对应的索引添加注释。

  具体请参见：**•COMMENT [=] 'string'**

- **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。在Ustore存储引擎下，该值得默认值为92，在Astore存储引擎下默认值为100（完全填充）。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。

    取值范围：10~100

  - ORIENTATION

    决定了表的数据的存储方式。当前M-compatibility只支持行存。

    取值范围：

    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Inplace-Update存储。

- COMPRESSION

  行存表不支持压缩。

- segment

  使用段页式的方式存储。本参数仅支持行存表。不支持临时表、unlog表。

  取值范围：on/off

  默认值：off

- statistic_granularity

  记录该表在分析统计信息时的默认PARTITION_MODE。此参数对非分区表设置无效。

  取值范围：请参见PARTITION_MODE选项说明。

  默认值：AUTO。

  表 4-23 PARTITION_MODE 选项说明

PARTITION_MODE选项	含义
ALL	收集整表、一级分区的统计信息。
GLOBAL	收集整表的统计信息。
PARTITION	收集一级分区的统计信息。
GLOBAL AND PARTITION	收集整表、一级分区的统计信息。
ALL COMPLETE	收集整表、一级分区的统计信息。
AUTO	缺省值。

- **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。

  注：需要在非M-Compatibility数据库中创建或删除tablespace。

- **PARTITION BY {RANGE [COLUMNS] | LIST [COLUMNS]} (partition_key)**
  - 对于partition_key，分区策略的分区键仅支持1列，且不支持表达式。
  - 分区键支持的数据类型和一级分区表约束保持一致。
  - COLUMNS关键字只能加在RANGE或LIST之后，“RANGE COLUMNS” 语义同 “RANGE”，“LIST COLUMNS” 语义同 “LIST”。
  - KEY与HASH同义。
- **SUBPARTITION BY {HASH | KEY} (subpartition_key)**
  - 对于subpartition_key，分区策略的分区键仅支持1列，且不支持表达式。
  - 分区键支持的数据类型和一级分区表约束保持一致。
- **PARTITIONS integer**

  指定分区个数。

  integer为分区数，必须为大于0的整数，且不得大于1048575。

  - 当在RANGE和LIST分区后指定此子句时，必须显式定义每个分区，且定义分区的数量必须与integer值相等。只能在RANGE和LIST分区后指定此子句。

  - 当在HASH和KEY分区后指定此子句时，若不列出各个分区定义，将自动生成integer个分区，自动生成的分区名为“p+数字”，数字依次为0到integer-1，分区的表空间默认为此表的表空间；也可以显式列出每个分区定义，此时定义分区的数量必须与integer值相等。若既不列出分区定义，也不指定分区数量，将创建唯一一个分区。

- **SUBPARTITIONS integer**

  指定二级分区数量。

  integer为二级分区个数，必须为大于0的整数，且不得大于1048575。

  - 只能在HASH和KEY二级分区后指定此子句。

    - 若不列出各个二级分区定义，将在每个一级分区内自动生成integer个二级分区，自动生成的二级分区名为“一级分区名+sp+数字”，数字依次为0到integer-1，分区的表空间默认为此表的表空间。
    - 也可以列出每个二级分区定义，此时二级分区的数量必须与integer值相等。
    - 若既不列出每个二级分区定义，也不指定二级分区数量，将创建唯一一个二级分区。

- **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 (expression)**

  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。

  说明

  - expression表达式中，如果存在“<>NULL”或“！=NULL”，这种写法是无效的，需要修改为“is NOT NULL”。
  - 不开启精度传递开关（m_format_behavior_compat_options不开启enable_precision_decimal选项）时创建的带有CHECK约束的表，与开启精度传递开关后创建的带有CHECK约束的表在浮点数的比较结果等方面可能出现不一致的情况。如果用户需要在开启精度传递开关的情况下使用未开启精度开关时创建的带有CHECK约束的表，建议在开启精度传递开关后使用ALTER TABLE语法重新定义CHECK约束。

```
m_db=# SET m_format_behavior_compat_options='';
SET
m_db=# CREATE TABLE mm1(a float(10, 4), b float(5, 3), CHECK(a/b=1.7142858) STORED);
CREATE TABLE
m_db=# INSERT INTO mm1 VALUES(1.2, 0.7);
ERROR:  New row in relation "mm1" violates check constraint "mm1_check".
DETAIL:  N/A
m_db=# SET m_format_behavior_compat_options='enable_precision_decimal';
SET
m_db=# INSERT INTO mm1 VALUES(1.2, 0.7);
INSERT 0 1
m_db=# CREATE TABLE mm2(a float(10, 4), b floaT(5, 3), CHECK(a/b=1.7142858) STORED);
CREATE TABLE
m_db=# INSERT INTO mm2 VALUES(1.2, 0.7);
ERROR:  New row in relation "mm2" violates check constraint "mm2_check".
DETAIL:  N/A

m_db=# DROP TABLE mm1, mm2;
CREATE TABLE
```


- **DEFAULT default_expr**
  - 使用DEFAULT子句给字段指定缺省表达式，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 。
  - 当未在缺省表达式外嵌套括号时，支持指定以下内容：常量、带正负号的数值常量、**update_expr**。
  - 当在缺省表达式外嵌套括号时，支持指定以下内容：常量、带正负号的数值常量、**update_expr**、CURRENT_TIME/CURTIME函数、CURRENT_DATE/CURDATE函数（CURRENT_TIME/CURRENT_DATE支持不带括号形式的调用）。
  - 仅支持在TIMESTAMP、DATETIME类型的字段上指定update_expr作为默认值，且字段的精度与update_expr的精度须保持一致。
- **ON UPDATE update_expr**

  ON UPDATE子句为字段的一种属性约束。

  当对表中某元组执行UPDATE操作时，若更新字段的新值和表中旧值不相同，则表中该元组上具有该属性且不在更新字段内的字段值自动更新为当前时间戳；若更新字段的新值和表中旧值相同，则表中该元组上具有该属性且不在更新字段内的字段值不变，保持原有值；若具有该属性的字段在更新字段内，则对应这些字段值直接按指定更新的值更新。

  说明

  - 语法上update_expr支持CURRENT_TIMESTAMP 、LOCALTIMESTAMP 、LOCALTIME、NOW()四种关键字，也支持关键字带括号指定或不指定精度。例如：ON UPDATE CURRENT_TIMESTAMP()、ON UPDATE CURRENT_TIMESTAMP(5)、ON UPDATE LOCALTIMESTAMP()、ON UPDATE LOCALTIMESTAMP(6)等。不带括号或空括号时精度为0，其中NOW关键字不支持不带括号。四种关键字互为同义词，属性效果相同。
  - 该属性支持在如下类型的列上指定: TIMESTAMP、DATETIME。
  - 该属性指定的精度和对应列上类型指定的精度必须一致，否则会触发报错。例如：CREATE TABLE t1 (col1 timestamp(6) ON UPDATE CURRENT_TIMESTAMP(6)); 若精度不一致，会产生ERROR: Invalid ON UPDATE clause for "col1" 报错。
  - 该属性和生成列约束不能同时指定同一列。
  - 分区表中的分区键不支持指定该属性。
- **[GENERATED ALWAYS] AS (generation_expr) [STORED | VIRTUAL]**

  该子句将字段创建为生成列，通过指定STORED和VIRTUAL关键字表明生成列的列值存储方式：
  - STORED：创建存储生成列，列值需要存储，在插入或更新元组时，由生成列表达式计算存储生成列的列值。
  - VIRTUAL：创建虚拟生成列，列值不需要存储，在查询语句涉及虚拟生成列时，由生成列表达式计算列值。

说明

- STORED和VIRTUAL关键字可省略，缺省为STORED，在参数m_format_dev_version为s2条件下缺省为VIRTUAL。
- 生成表达式不能以任何方式引用当前行以外的其他数据。生成表达式支持引用早定义于当前生成列的其他生成列，不能引用系统列。生成列表达式不能引用系统变量、用户自定义变量、存储过程中的变量。生成列表达式不支持引用自增列，且生成列不支持定义自增列属性。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、窗口函数和自定义函数。生成表达式调用的函数只能是不可变（IMMUTABLE）函数。
- 不支持为生成列指定默认值。
- 存储生成列支持作为分区键的一部分，当作为分区键的一部分时，不支持ALTER TABLE修改存储生成列。
- 虚拟生成列不支持作为分区键的一部分。
- 虚拟生成列不支持被外键约束引用。
- 不支持在虚拟生成列上创建索引。
- 存储生成列不能和ON UPDATE约束子句的CASCADE,SET NULL,SET DEFAULT动作同时指定。存储生成列不能和ON DELETE约束子句的SET NULL,SET DEFAULT动作同时指定。
- 在参数m_format_dev_version为s2条件下，存储生成列的基列不能和ON UPDATE或者ON DELETE约束子句的CASCADE, 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。删除生成列依赖的基列，生成列被自动删除。在参数m_format_dev_version为s2条件下，删除生成列依赖的基列之前需要先删除对应的生成列。
- 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。向可被更新的视图中插入生成列数据时，不能为生成列指定值，但是可以指定关键字DEFAULT。
- 当存储生成列带CHECK约束时，在参数m_format_dev_version为s2的条件下，ALTER TABLE修改存储生成列定义，如果存在生成列不满足CHECK约束，ALTER TABLE语句执行失败，系统报ERROR。
- 生成列的权限控制和普通列一样。
- 不开启精度传递开关（m_format_behavior_compat_options不开启enable_precision_decimal选项）时创建的带有生成列的表，与开启精度传递开关后创建的带有生成列的表在浮点数的比较结果等方面可能出现不一致的情况。如果用户需要在开启精度传递开关的情况下使用未开启精度开关时创建的带有生成列的表，建议在开启精度传递开关后使用ALTER TABLE语法重新定义生成列。

```
m_db=# SET m_format_behavior_compat_options='';
SET
m_db=# CREATE TABLE mm1(a float(10, 4), b float(5, 3), c boolean AS ((a/b)=1.7142858)
STORED);
CREATE TABLE
m_db=# INSERT INTO mm1 VALUES(1.2, 0.7);
INSERT 0 1
m_db=# SELECT * FROM mm1;
  a    |   b   | c
--------+-------+---
 1.2000 | 0.700 | 0
(1 row)

m_db=# SET m_format_behavior_compat_options='enable_precision_decimal';
SET
m_db=# CREATE TABLE mm2(a float(10, 4), b float(5, 3), c boolean AS ((a/b)=1.7142858)
STORED);
CREATE TABLE
m_db=# INSERT INTO mm1 VALUES(1.2, 0.7);
INSERT 0 1
m_db=# INSERT INTO mm2 VALUES(1.2, 0.7);
INSERT 0 1
m_db=# SELECT * FROM mm1;
  a    |   b   | c
```



```
--------+-------+---
 1.2000 | 0.700 | 0
 1.2000 | 0.700 | 1
(2 rows)

m_db=# SELECT * FROM mm2;
   a    |   b   | c
--------+-------+---
 1.2000 | 0.700 | 0
(1 row)

m_db=# DROP TABLE mm1, mm2;
DROP TABLE
```

- **AUTO_INCREMENT**
  指定列为自动增长列。
  详见：**•AUTO_INCREMENT**。
- **列级唯一约束：UNIQUE [KEY] index_parameters**
  UNIQUE约束表示表里的一个或多个字段的组合必须在全表范围内唯一。
  UNIQUE KEY与UNIQUE语义相同。
- **表级唯一约束：UNIQUE [INDEX | KEY][index_name][USING method] ({{ column_name [(length)] | (expression) } [ASC | DESC] }[, ...]) 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。
  column_name(length)是前缀键，详见：**•column_name (length)**。
  index_name为索引名。

须知

对于唯一键约束，constraint_name和index_name同时指定时，索引名为index_name。

- **列级主键约束：[PRIMARY] KEY index_parameters**
  表级主键约束：PRIMARY KEY [index_name] [USING method] ({ column_name [ASC | DESC] } [, ...] ） 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。

示例

- 示例1：创建各种组合类型的二级分区表

```
m_db=# CREATE TABLE list_hash
(
   month_code VARCHAR ( 30 ) NOT NULL ,
   dept_code  VARCHAR ( 30 ) NOT NULL ,
   user_no    VARCHAR ( 30 ) NOT NULL ,
   sales_amt  int
)
PARTITION BY LIST (month_code) SUBPARTITION BY HASH (dept_code)
(
 PARTITION p_201901 VALUES ( '201902' )
 (
  SUBPARTITION p_201901_a,
  SUBPARTITION p_201901_b
```



```
 ),
 PARTITION p_201902 VALUES ( '201903' )
 (
  SUBPARTITION p_201902_a,
  SUBPARTITION p_201902_b
 )
);
m_db=# INSERT INTO list_hash VALUES('201902', '1', '1', 1);
m_db=# INSERT INTO list_hash VALUES('201902', '2', '1', 1);
m_db=# INSERT INTO list_hash VALUES('201902', '3', '1', 1);
m_db=# INSERT INTO list_hash VALUES('201903', '4', '1', 1);
m_db=# INSERT INTO list_hash VALUES('201903', '5', '1', 1);
m_db=# INSERT INTO list_hash VALUES('201903', '6', '1', 1);
m_db=# SELECT * from list_hash;
 month_code | dept_code | user_no | sales_amt
------------+-----------+---------+-----------
 201903     | 4         | 1       |         1
 201903     | 5         | 1       |         1
 201903     | 6         | 1       |         1
 201902     | 2         | 1       |         1
 201902     | 3         | 1       |         1
 201902     | 1         | 1       |         1
(6 rows)

m_db=# DROP TABLE list_hash;
m_db=# CREATE TABLE range_hash
(
    month_code VARCHAR ( 30 ) NOT NULL ,
    dept_code  VARCHAR ( 30 ) NOT NULL ,
    user_no    VARCHAR ( 30 ) NOT NULL ,
    sales_amt  int
)
PARTITION BY RANGE (month_code) SUBPARTITION BY HASH (dept_code)
(
  PARTITION p_201901 VALUES LESS THAN( '201903' )
  (
    SUBPARTITION p_201901_a,
    SUBPARTITION p_201901_b
  ),
  PARTITION p_201902 VALUES LESS THAN( '201904' )
  (
    SUBPARTITION p_201902_a,
    SUBPARTITION p_201902_b
  )
);
m_db=# INSERT INTO range_hash VALUES('201902', '1', '1', 1);
m_db=# INSERT INTO range_hash VALUES('201902', '2', '1', 1);
m_db=# INSERT INTO range_hash VALUES('201902', '1', '1', 1);
m_db=# INSERT INTO range_hash VALUES('201903', '2', '1', 1);
m_db=# INSERT INTO range_hash VALUES('201903', '1', '1', 1);
m_db=# INSERT INTO range_hash VALUES('201903', '2', '1', 1);
m_db=# SELECT * from range_hash;
 month_code | dept_code | user_no | sales_amt
------------+-----------+---------+-----------
 201902     | 2         | 1       |         1
 201902     | 1         | 1       |         1
 201902     | 1         | 1       |         1
 201903     | 2         | 1       |         1
 201903     | 2         | 1       |         1
 201903     | 1         | 1       |         1
(6 rows)

m_db=# DROP TABLE range_hash;
```

- 示例2：对二级分区表进行DML指定分区操作

```
m_db=# CREATE TABLE range_hash
(
    month_code VARCHAR ( 30 ) NOT NULL ,
    dept_code  VARCHAR ( 30 ) NOT NULL ,
    user_no    VARCHAR ( 30 ) NOT NULL ,
```



```
  sales_amt  int
)
PARTITION BY RANGE (month_code) SUBPARTITION BY HASH (dept_code)
(
 PARTITION p_201901 VALUES LESS THAN( '201903' )
 (
  SUBPARTITION p_201901_a,
  SUBPARTITION p_201901_b
 ),
 PARTITION p_201902 VALUES LESS THAN( '201904' )
 (
  SUBPARTITION p_201902_a,
  SUBPARTITION p_201902_b
 )
);
--指定一级分区插入数据
m_db=# INSERT INTO range_hash partition (p_201901) VALUES('201902', '1', '1', 1);
--实际分区和指定分区不一致，报错
m_db=# INSERT INTO range_hash partition (p_201902) VALUES('201902', '1', '1', 1);
ERROR:  inserted partition key does not map to the table partition
DETAIL:  N/A.

--指定分区查询数据
m_db=# SELECT * from range_hash partition (p_201901);
 month_code | dept_code | user_no | sales_amt
------------+-----------+---------+-----------
 201902     | 1         | 1       |         1
(1 row)

m_db=# SELECT * from range_hash partition for ('201902');
 month_code | dept_code | user_no | sales_amt
------------+-----------+---------+-----------
 201902     | 1         | 1       |         1
( rows)

--指定分区更新数据
m_db=# UPDATE range_hash partition (p_201901) SET user_no = '2';
m_db=# SELECT * from range_hash;
 month_code | dept_code | user_no | sales_amt
------------+-----------+---------+-----------
 201902     | 1         | 2       |         1
(1 row)
m_db=# UPDATE range_hash partition for ('201902') SET user_no = '4';
m_db=# SELECT * from range_hash;
 month_code | dept_code | user_no | sales_amt
------------+-----------+---------+-----------
 201902     | 1         | 4       |         1
(1 row)

--指定分区删除数据
m_db=# DELETE FROM range_hash partition (p_201901);
DELETE 1
```

4.4.2.8.19 CREATE TABLE SELECT

功能描述

CREATE TABLE SELECT会创建一个表并且使用来自SELECT命令的结果填充该表。该表的字段名和字段类型来自于自定义列和SELECT输出字段。

- 只出现在自定义列中字段排在最前面，否则按照SELECT输出列的顺序依次排列。
- 列名相同但大小写不同会被识别为同一列，如果存在相同列，保留自定义列的数据类型，保留SELECT列名的大小写。

- 如果SELECT列和自定义列存在相同字段，始终保留SELECT列的COMMENT。如果SELECT列为直接表列或者视图展开后的直接表列，保留旧表中该列的COMMENT，否则COMMENT设置为NULL。
- 如果SELECT列为直接表列或者视图展开后的直接表列，并且未自定义该字段，保留旧表列的NULL、NOT NULL、DEFAULT VALUE、ON UPDATE、CHARSET和COLLATE。

说明

CREATE TABLE SELECT支持FOR UPDATE选项，锁机制与INSERT SELECT FOR UPDATE保持一致。

注意事项

- 不支持创建分区表。
- 不支持REPLACE/IGNORE。
- 如果SELECT列非直接表列，则默认允许NULL且无默认值。如：CREATE TABLE t1 SELECT unix_timestamp('2008-01-02 09:08:07.3465') AS a;，创建的新表的字段a允许为NULL且无默认值。
- 如果需要使用完整的功能，需要将GUC参数m_format_behavior_compat_options开启enable_precision_decimal选项，否则由于版本兼容性问题，涉及数据类型精度相关的类型将导致行为报错。如果SELECT列包含非直接表列（表达式、函数、常量等），以及UNION场景（因为涉及结果类型推导），都会报错。
- 如果在建表过程中数据库系统发生故障，系统恢复后可能无法自动清除之前已创建的、大小非0的磁盘文件。此种情况出现概率小，不影响数据库系统的正常运行。

语法格式

```
CREATE [ TEMPORARY | UNLOGGED ] TABLE [ IF NOT EXISTS ] table_name
   {({ column_name data_type [ {CHARACTER SET | CHAR SET | CHARSET} charset_name ] [ COLLATE
collation_name][AUTO_INCREMENT][COMMENT [=] 'string'] [ column_constraint [ ... ] ]
       | table_constraint
       | {INDEX | KEY} [ [schema_name.] index_name ] [ USING method ] ({column_name [(length)] | (expr)
[ASC | DESC]}[,...]) [[COMMENT 'string' | USING method][...]]
}
       [, ... ])}
   [ table_option ][[,...]|[...]]
   [ WITH ( {storage_parameter = value} [, ... ] ) ]
   [AS] query
;
```

- 其中table_option为：

```
{
    AUTO_INCREMENT [=] value
  | [DEFAULT] {CHARACTER SET | CHAR SET | CHARSET} [=] charset_name
  | [DEFAULT] COLLATE [=] collation_name
  | COMMENT [=] 'string'
  | ENGINE [=] engine_name
  | ROW_FORMAT [=] {DEFAULT | DYNAMIC | FIXED | COMPRESSED | REDUNDANT | COMPACT}
}
```


说明

- 同一个地方多次设置COMMENT时仅最后一个生效。
- 同一个地方多次设置AUTO_INCREMENT时仅最后一个生效。
- 同一个地方如果多次设置不同的COLLATE，COLLATE对应的字符集必须相同，且只有最后一个COLLATE生效。
- 不支持同一个地方多次设置不同的CHARSET。
- ENGINE、ROW_FORMAT使用时语法不报错也没有提示，但实际不生效。
- 不支持WITH (ORIENTATION = column)。
- 不支持PREPARE语句中嵌套CREATE TABLE AS。

- 其中列约束column_constraint为：

```
{ NOT NULL |
  NULL |
  CHECK ( expression ) |
  DEFAULT default_expr |
  ON UPDATE now_expr |
  [ GENERATED ALWAYS ] AS ( generation_expr ) [STORED | VIRTUAL] ] |
  UNIQUE [KEY] |
  [ PRIMARY ] KEY |
  REFERENCES reftable [ ( refcolumn ) ] [ MATCH FULL | MATCH SIMPLE ]
    [ ON DELETE action ] [ ON UPDATE action ] }
```

- 其中表约束table_constraint为：

```
[ CONSTRAINT [ constraint_name ] ]
{ CHECK ( expression ) |
  UNIQUE [ INDEX | KEY ] [ index_name ][ USING method ] ( { { column_name [ ( length ) ] |
( expression ) } [ ASC | DESC ] } [, ... ] ) [USING method| COMMENT 'string']|
  PRIMARY KEY [index_name][ USING method ] ( { column_name  }[ ASC | DESC ] } [, ... ] ) [USING
method| COMMENT 'string']|
  FOREIGN KEY [ index_name ] ( column_name [, ... ] ) REFERENCES reftable [ (refcolumn [, ... ] ) ]
    [ MATCH FULL | MATCH SIMPLE ] [ ON DELETE action ] [ ON UPDATE action ]
}
```

- 其中now_expr为：

```
{ CURRENT_TIMESTAMP | LOCALTIMESTAMP  | LOCALTIME | NOW() }
```

参数说明

- **UNLOGGED**

  如果指定此关键字，则创建的表为非日志表。

  在非日志表中写入的数据不会被写入到预写日志中，当执行操作系统重启、数据库重启、主备切换、切断电源操作或异常关机后，非日志表上的写入会被自动截断，会造成数据丢失的风险。非日志表中的内容也不会被复制到备服务器中。在非日志表中创建的索引也不会被自动记录。

  使用场景：非日志表无法保证数据的安全性，用户应该在确保数据已经做好备份的前提下使用，例如系统升级时进行数据的备份。

  故障处理：当异常关机等操作导致非日志表上的索引发生数据丢失时，用户应该对发生错误的索引进行重建。

- **TEMPORARY**

  如果指定TEMPORARY关键字，则创建的表为本地临时表。

  本地临时表只在当前会话可见，本会话结束后会自动删除。因此，在除当前会话连接的数据库节点故障时，仍然可以在当前会话上创建和使用临时表。由于临时表只在当前会话创建，对于涉及对临时表操作的DDL语句，会产生DDL失败的报错。因此，建议DDL语句中不要对临时表进行操作。

须知

- 本地临时表通过每个会话独立的以pg_temp开头的schema来保证只对当前会话可见，因此，不建议用户在日常操作中手动删除以pg_temp、pg_toast_temp开头的Schema。
- 如果建表时不指定TEMPORARY关键字，而指定表的Schema为当前会话的pg_temp_开头的Schema，则此表会被创建为临时表。
- 临时表只对当前会话可见，因此不支持与\parallel on并行执行一起使用。
- 临时表不支持主备切换。

- **IF NOT EXISTS**

  如果已经存在相同名称的表，不会报出错误，而会发出通知，告知通知此表已存在。
- **table_name**

  要创建的表名。
- **column_name**

  新表中要创建的字段名。
- **constraint_name**

  建表时指定的约束名称。
- **index_name**

  索引名。

须知

- 对于外键约束，constraint_name和index_name同时指定时，索引名为constraint_name。
- 对于唯一键约束，constraint_name和index_name同时指定时，索引名为index_name。

- **USING method**

  指定创建索引的方法。

  取值范围请参见**参数说明**中的USING method。

须知

目前只支持索引方法为btree，但是不支持ustore表创建btree索引。

- **ASC | DESC**

  ASC表示指定按升序排序（默认）。DESC指定按降序排序。
- **expression**

  创建一个基于该表的一个或多个字段的表达式索引约束，必须写在圆括弧中。
- **data_type**

  字段的数据类型。

- **{CHARACTER SET | CHAR SET | CHARSET} charset_name**

  指定表字段的字符集。单独指定时会将字段的字符序设置为指定的字符集的默认字符序。

- **COLLATE collation_name**

  COLLATE子句指定列的排序规则（字符序）（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。支持的字符序具体请参见**表级字符集和字符序**

  说明

  - 仅字符类型支持指定字符集。指定为BINARY字符集或字符序实际是将字符类型转化为对应的二进制类型，若类型映射不存在则报错。当前仅有TEXT类型转化为BLOB的映射。
  - 字段字符集或字符序未显式指定时，若指定了表的默认字符集或字符序，字段字符集和字符序将从表上继承。
  - 除SQL_ASCII库外，其他字符集的数据库支持多字符集混用。

- **AUTO_INCREMENT [=] value**

  这个子句为自动增长列指定一个初始值，value必须为正数，不得超过2^{127}-1。

- **COMMENT [=] 'string'**
  - COMMENT [=] 'string'子句表示给表添加注释。
  - 在column_constraint中的COMMENT 'string'表示给列添加注释。
  - 在table_constraint中的COMMENT 'string'表示给主键和唯一键对应的索引添加注释。

  须知

  - 表级注释支持的最大字符串长度为2048字符，列级和索引级注释支持的最大长度为1024字符。
  - table_constraint中的COMMENT仅支持主键和唯一键，其他约束不支持。

- **WITH ({ storage_parameter = value } [, ...])**

  这个子句为表或索引指定一个可选的存储参数。用于表的WITH子句还可以包含OIDS=FALSE表示不分配OID。

  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。在Ustore存储引擎下，该值得默认值为92，在Astore存储引擎下默认值为100（完全填充）。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。

    取值范围：10~100

  - **ORIENTATION**

    指定表数据的存储方式，该参数设置成功后就不再支持修改，当前M-Compatibility仅支持行存方式。

取值范围：

- ROW，表示表的数据将以行式存储。

  行存储适合于OLTP业务，适用于点查询或者增删操作较多的场景。

默认值：

若指定表空间为普通表空间，默认值为ROW。

- **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ORIENTATION和STORAGE_TYPE时创建表，默认是USTORE存储引擎（表示表支持Inplace-Update存储引擎）。

- **INIT_TD**

  创建UStore表时，指定初始化的TD个数，该参数可以通过alter table进行修改。特别需要注意，该参数会影响数据页面存放的单个元组的最大大小，具体换算方法为MAX_TUPLE_SIZE = BLCKSZ - INIT_TD * TD_SIZE，例如用户将INIT_TD数量从4修改为8，单个元组最大大小会减小4 * INIT_TD大小。

  取值范围：2~128，默认值为4。

- **COMPRESSION**

  指定表数据的压缩级别，它决定了表数据的压缩比以及压缩时间。一般来讲，压缩级别越高，压缩比也越大，压缩时间也越长；反之亦然。实际压缩比取决于加载的表数据的分布特征。行存表不支持压缩。

  取值范围：行存表不支持压缩，默认值为NO。

- **COMPRESSLEVEL**

  指定表数据同一压缩级别下的不同压缩水平，它决定了同一压缩级别下表数据的压缩比以及压缩时间。对同一压缩级别进行了更加详细的划分，为用户选择压缩比和压缩时间提供了更多的空间。总体来讲，此值越大，表示同一压缩级别下压缩比越大，压缩时间越长；反之亦然。

  取值范围：0~3，默认值为0。

- **COMPRESSTYPE**

  行存表参数，设置行存表压缩算法。1代表pglz算法，2代表zstd算法，3代表TurboDB（自研改进压缩），默认不压缩。

  取值范围：0~23，默认值为0。

- **COMPRESS_LEVEL**

  行存表参数，设置行存表压缩算法等级，仅当COMPRESSTYPE为2时生效。压缩等级越高，表的压缩效果越好，表的访问速度越慢。

  取值范围：-31~31，默认值为0。

- **COMPRESS_CHUNK_SIZE**

  行存表参数，设置行存表压缩chunk块大小。chunk数据块越小，预期能达到的压缩效果越好，同时数据越离散，影响表的访问速度。

取值范围：与页面大小有关。在页面大小为8k场景，取值范围为：512、1024、2048、4096。

默认值：4096

- **COMPRESS_PREALLOC_CHUNKS**

  行存表参数，设置行存表压缩chunk块预分配数量。预分配数量越大，表的压缩率相对越差，离散度越小，访问性能越好。

  取值范围：0~7，默认值为0。

- **COMPRESS_BYTE_CONVERT**

  行存表参数，设置行存表压缩字节转换预处理。在一些场景下可以提升压缩效果，同时会导致一定性能劣化。仅在COMPRESSTYPE为2时允许配置，用于指定是否需要进行预处理中的行列转换处理，其他压缩算法禁止配置。

  取值范围：布尔值，默认关闭。

- **COMPRESS_DIFF_CONVERT**

  行存表参数，设置行存表压缩字节差分预处理。仅在COMPRESSTYPE为2且COMPRESS_BYTE_CONVERT为true时允许配置为true。在一些场景下可以提升压缩效果，同时会导致一定性能劣化。

  取值范围：布尔值，默认关闭。

- **segment**

  使用段页式的方式存储。本参数仅支持行存表。不支持1-5号物理文件非法删除破坏场景的防护。

  取值范围：on/off

  默认值：off

- **enable_tde**

  创建透明加密表。前提是开启透明数据加密开关GUC参数**enable_tde**，同时启用了KMS密钥管理服务，并正确配置了数据库实例主密钥ID GUC参数**tde_cmk_id**。本参数仅支持行存表、段页式表和unlogged表。不支持临时表。

  取值范围：on/off。当前配置为on时表示开启透明数据加密；当前配置为off时，表示当前不开启加密但是保留后期打开加密功能，在创建表时会向KMS申请创建数据加密密钥。

  默认值：off

- **parallel_workers**

  表示创建索引时起的bgworker线程数量，例如2就表示将会起2个bgworker线程并发创建索引。

  取值范围：[0,32]，int类型，0表示关闭并行建索引。

  默认值：不设置该参数，表示未开启并行建索引功能。

- **encrypt_algo**

  指定透明数据加密算法。前提是需要对该表设置enable_tde选项。加密算法只能在创建表时指定，不同的表允许使用不同的加密算法，创建表成功后算法不可修改。

  取值范围：字符串，有效值为：AES_128_CTR，SM4_CTR。

  默认值：不设置enable_tde选项时默认为空；当enable_tde选项设置为on或off时，如果不设置encrypt_algo则算法默认为AES_128_CTR。

- **dek_cipher**

透明数据加密密钥的密文。当开启enable_tde选项时会自动申请创建，用户不可单独指定。通过密钥轮转功能可以对密钥进行更新。

取值范围：字符串。

默认值：不开启加密时默认为空。

- **key_type**

  透明加密密钥类型。当开启enable_tde选项时会自动申请创建，用户不可单独指定。

  取值范围：字符串。

  默认值：不开启加密时默认为空。

- **cmk_id**

  透明数据加密使用的数据库实例主密钥ID。当开启enable_tde选项时通过GUC参数**tde_cmk_id**获取，用户单独不可指定或修改。

  取值范围：字符串。

  默认值：不开启加密时默认为空。

- **hasuids**

  参数开启：更新表元组时，为元组分配表级唯一标识id。

  取值范围：on/off。

  默认值：off。

- **collate**

  用于记录表的默认字符序，一般只用于内部存储和导入导出，不推荐用户指定或修改。

  取值范围：支持的字符序的oid。

  默认值：0。

- **stat_state**

  标识该表的统计信息是否被锁定，如果被锁定了，该表的统计信息无法更新。

  取值范围：locked、unlock。

  默认值：unlock。

- **CONSTRAINT constraint_name**

  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。

  表约束：不和某个列绑在一起，可以作用于多个列。

- **NOT NULL**

  字段值不允许为NULL。

- **NULL**

  字段值允许为NULL ，这是缺省值。

  这个子句只是为和非标准SQL数据库兼容。不建议使用。

- **CHECK (expression)**

  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。

说明

expression表达式中，如果存在“<>NULL”或“！=NULL”，这种写法不生效，需要修改为“is NOT NULL”。

- **DEFAULT default_expr**
  - 使用DEFAULT子句给字段指定缺省表达式，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 。
  - 当未在缺省表达式外嵌套括号时，支持指定以下内容：常量、带正负号的数值常量、**update_expr**。
  - 当在缺省表达式外嵌套括号时，支持指定以下内容：常量、带正负号的数值常量、**update_expr**、CURRENT_TIME/CURTIME函数、CURRENT_DATE/CURDATE函数（CURRENT_TIME/CURRENT_DATE支持不带括号形式的调用）。
  - 仅支持在TIMESTAMP、DATETIME类型的字段上指定update_expr作为默认值，且字段的精度与update_expr的精度须保持一致。
- **ON UPDATE now_expr**

  ON UPDATE子句为字段的一种属性约束。

  当对表中某元组执行UPDATE操作时，若更新字段的新值和表中旧值不相同，则表中该元组上具有该属性且不在更新字段内的字段值自动更新为当前时间戳；若更新字段的新值和表中旧值相同，则表中该元组上具有该属性且不在更新字段内的字段值不变，保持原有值；若具有该属性的字段在更新字段内，则对应这些字段值直接按指定更新的值更新。

  说明

  - 语法上update_expr支持CURRENT_TIMESTAMP 、LOCALTIMESTAMP 、LOCALTIME和NOW()四种关键字，同时支持关键字带括号指定或不指定精度。例如：ON UPDATE CURRENT_TIMESTAMP()、ON UPDATE CURRENT_TIMESTAMP(5)、ON UPDATE LOCALTIMESTAMP()以及ON UPDATE LOCALTIMESTAMP(6)等。不带括号或空括号时精度为0，其中NOW关键字不支持不带括号。四种关键字互为同义词，属性效果相同。
  - 该属性支持在如下类型的列上指定: TIMESTAMP、DATETIME。
  - 该属性指定的精度和对应列上类型指定的精度必须一致，否则会触发报错。例如：CREATE TABLE t1 (col1 timestamp(6) ON UPDATE CURRENT_TIMESTAMP(6)); 若精度不一致，会产生ERROR: Invalid ON UPDATE clause for "col1" 报错。
  - 该属性和生成列约束不能同时指定同一列。
- **[GENERATED ALWAYS] AS (generation_expr) [STORED | VIRTUAL]**

  该子句将字段创建为生成列，通过指定STORED和VIRTUAL关键字表明生成列的列值存储方式：
  - STORED：创建存储生成列，列值需要存储，在插入或更新元组时，由生成列表达式计算存储生成列的列值。
  - VIRTUAL：创建虚拟生成列，列值不需要存储，在查询语句涉及虚拟生成列时，由生成列表达式计算列值。

说明

- STORED和VIRTUAL关键字可省略，缺省为STORED，在参数m_format_dev_version为s2条件下缺省为VIRTUAL。
- 生成表达式不能以任何方式引用当前行以外的其他数据。生成表达式支持引用早定义于当前生成列的其他生成列，不能引用系统列。生成列表达式不能引用系统变量、用户自定义变量、存储过程中的变量。生成列表达式不支持引用自增列，且生成列不支持定义自增列属性。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、窗口函数和自定义函数。生成表达式调用的函数只能是不可变（IMMUTABLE）函数。
- 不支持为生成列指定默认值。
- 虚拟生成列不支持被外键约束引用。
- 存储生成列不能和ON UPDATE约束子句的CASCADE,SET NULL,SET DEFAULT动作同时指定。存储生成列不能和ON DELETE约束子句的SET NULL,SET DEFAULT动作同时指定。
- 在参数m_format_dev_version为s2条件下，存储生成列的基列不能和ON UPDATE或者ON DELETE约束子句的CASCADE, 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。删除生成列依赖的基列，生成列被自动删除。在参数m_format_dev_version为s2条件下，删除生成列依赖的基列之前需要先删除对应的生成列。
- 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。向可被更新的视图中插入生成列数据时，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。

- **AUTO_INCREMENT**

  该关键字将字段指定为自动增长列。

  自动增长列建议设置索引，并需要作为索引的第一个字段，否则建表时产生警告。

  若在插入时不指定此列的值（或指定此列的值为0、NULL、DEFAULT），此列的值将由自增计数器自动增长得到。

  若插入或更新此列为一个大于当前自增计数器的值，执行成功后，自增计数器将刷新为此值。

  自增初始值由“AUTO_INCREMENT [=] value”子句设置，若不设置，默认为1。

说明

- 自动增长列数据类型只能为整数类型、4字节或8字节浮点类型、布尔类型。当自增值已经达到字段数据类型的最大值时，继续自增将产生错误。
- 每个表只能有一个自动增长列。
- 自动增长列建议设置索引，并需要作为索引的第一个字段，否则建表时产生警告，含有自动增长列的表进行某些操作时会产生错误。
- 自动增长列不能指定DEFAULT缺省值。
- CHECK约束的表达式中不能含有自动增长列，生成列的表达式中不能含有自动增长列。
- 可以指定自动增长列允许NULL，若不指定，默认自动增长列含有NOT NULL约束。
- 含有自动增长列的表创建时，会创建一个依赖于此列的序列作为自增计数器，不允许通过序列相关功能修改或删除此序列，可以查看序列的值。
- 本地临时表中的自动增长列不会创建序列。
- 自增计数器自增和刷新操作不会回滚。
  - 数据插入到表之前，0/NULL会触发自增。数据插入或更新到表之后，会更新自增计数器。如果在自增之后出现了报错，数据没有插入或更新到表中，此时自增计数器不会回滚。后续插入语句基于自增计数器触发自增，会出现表中自动增长列的值不连续的情况。
  - 批量插入或导入预留自增缓存值也有可能产生自动增长列的值不连续的情况，详见auto_increment_cache参数说明。

- **[DEFAULT] {CHARACTER SET | CHAR SET | CHARSET} [=] default_charset**

  指定表的默认字符集。单独指定时会将表的默认字符序设置为指定的字符集的默认字符序。

- **[DEFAULT] COLLATE [=] default_collation**

  指定表的默认字符序。单独指定时会将表的默认字符集设置为指定的字符序对应的字符集。

  字符序请参见**字符集与字符序**。

  说明

  表的字符集或字符序未显式指定时，若指定了模式的默认字符集或字符序，表字符集和字符序将从模式上继承。

- **列级唯一约束：UNIQUE [KEY]**

  UNIQUE约束表示表里的一个或多个字段的组合必须在全表范围内唯一。

  UNIQUE KEY与UNIQUE语义相同。

- **表级唯一约束：UNIQUE [INDEX | KEY][index_name][USING method] ({{ column_name [(length)] | (expression) } [ASC | DESC] }[, ...])**

  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。

  column_name(length)是前缀键，详见：**•column_name (length)**。

  index_name为索引名。

须知

对于唯一键约束，constraint_name和index_name同时指定时，索引名为index_name。

- **列级主键约束：[PRIMARY] KEY**

  表级主键约束：PRIMARY KEY [index_name] [USING method] ({ column_name [ASC | DESC] } [, ...])

  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。

- **表列字段约束：REFERENCES reftable [(refcolumn)] [MATCH matchtype] [ON DELETE action] [ON UPDATE action]**

  表级别约束：FOREIGN KEY (column_name [, ...]) REFERENCES reftable [(refcolumn [, ...])] [MATCH matchtype] [ON DELETE action] [ON UPDATE action]

  说明

  表列字段约束REFERENCES在数据库M-compatibility模式兼容版本控制开关s1及以上版本时，语法不报错也没有提示，但实际不生效（如m_format_dev_version = 's1'）。

  外键约束要求新表中一列或多列构成的组应该只包含、匹配被参考表中被参考字段值。若省略refcolumn，则将使用reftable的主键。被参考列应该是被参考表中的唯一字段或主键。外键约束不能被定义在临时表和永久表之间。

  参考字段与被参考字段之间存在三种类型匹配，分别是：

  - MATCH FULL：不允许一个多字段外键的字段为NULL，除非全部外键字段都是NULL。
  - MATCH SIMPLE（缺省）：允许任意外键字段为NULL。

  另外，当被参考表中的数据发生改变时，某些操作也会在新表对应字段的数据上执行。ON DELETE子句声明当被参考表中的被参考行被删除时要执行的操作。ON UPDATE子句声明当被参考表中的被参考字段数据更新时要执行的操作。对于ON DELETE子句、ON UPDATE子句的可能动作：

  - NO ACTION（缺省）：删除或更新时，创建一个表明违反外键约束的错误。
  - RESTRICT：删除或更新时，创建一个表明违反外键约束的错误。
  - CASCADE：删除新表中任何引用了被删除行的行，或更新新表中引用行的字段值为被参考字段的新值。
  - SET NULL：设置引用字段为NULL。
  - SET DEFAULT：设置引用字段为它们的缺省值。

  说明

  - 外键约束的完整性检查由GUC参数foreign_key_checks进行控制，foreign_key_checks可以设置为on/off，分别表示启用/关闭外键约束的完整性检查，默认情况下为on。

- **AS query**

  一个SELECT语句。若SELECT语句首部带有CTE，AS后面的部分需要使用括号进行包含，如：AS (query) 。

示例

```
-- 删除表
m_db=# DROP TABLE IF EXISTS test1, test2, test3, test4;
DROP TABLE
m_db=# CREATE TABLE test1(col1 int, col2 int);
CREATE TABLE

-- 无自定义列，仅复制test1字段建表
m_db=# CREATE TABLE test2 AS SELECT * FROM test1;
INSERT 0 0
```



```
m_db=# SHOW CREATE TABLE test2;
 Table |                          Create Table
-------+-----------------------------------------------------------------------------
 test2 | SET search_path = public;                                                  +
       | CREATE TABLE test2 (                                                       +
       |     col1 integer,                                                          +
       |     col2 integer                                                           +
       | )                                                                          +
       | CHARACTER SET = "UTF8" COLLATE = "utf8mb4_general_ci"                      +
       | WITH (orientation=row, compression=no, storage_type=USTORE, segment=off);
(1 row)

-- 包含自定义列，新表字段包含自定义列和select结果列
m_db=# CREATE TABLE test3(col0 int, col2 varchar(55) not null) AS SELECT * FROM test1;
INSERT 0 0
m_db=# SHOW CREATE TABLE test3;
 Table |                          Create Table
-------+-----------------------------------------------------------------------------
 test3 | SET search_path = public;                                                  +
       | CREATE TABLE test3 (                                                       +
       |     col0 integer,                                                          +
       |     col1 integer,                                                          +
       |     col2 varchar(55) CHARACTER SET `UTF8` COLLATE utf8mb4_general_ci NOT NULL+
       | )                                                                          +
       | CHARACTER SET = "UTF8" COLLATE = "utf8mb4_general_ci"                      +
       | WITH (orientation=row, compression=no, storage_type=USTORE, segment=off);
(1 row)

-- 包含建表选项示例
m_db=# CREATE TABLE test4(col0 int, col2 varchar(55)) CHARACTER SET gb18030 COLLATE gb18030_bin
WITH(storage_type=astore) AS SELECT * FROM test1;
INSERT 0 0
m_db=# SHOW CREATE TABLE test4;
 Table |                         Create Table
-------+-------------------------------------------------------------------------
 test4 | SET search_path = public;                                              +
       | CREATE TABLE test4 (                                                   +
       |     col0 integer,                                                      +
       |     col1 integer,                                                      +
       |     col2 varchar(55) CHARACTER SET `GB18030` COLLATE gb18030_bin+
       | )                                                                      +
       | CHARACTER SET = "GB18030" COLLATE = "gb18030_bin"                      +
       | WITH (orientation=row, storage_type=astore, compression=no);
(1 row)

-- 删除表
m_db=# DROP TABLE IF EXISTS test1, test2, test3, test4;
DROP TABLE
```

4.4.2.8.20 CREATE USER

功能描述

创建一个用户。

注意事项

- 通过CREATE USER创建的用户，默认具有LOGIN权限。
- 通过CREATE USER创建用户的同时，系统会在执行该命令的数据库中，为该用户创建一个同名的SCHEMA。
- 系统管理员在普通用户同名schema下创建的对象，所有者为schema的同名用户（非系统管理员）。

语法格式

```
CREATE USER user_name [ [ WITH ] option [ ... ] ]{ PASSWORD | IDENTIFIED BY } { 'password' [EXPIRED] | DISABLE };
```

其中option子句用于设置权限及属性等信息。

```
{SYSADMIN | NOSYSADMIN}
    | {MONADMIN | NOMONADMIN}
    | {OPRADMIN | NOOPRADMIN}
    | {POLADMIN | NOPOLADMIN}
    | {AUDITADMIN | NOAUDITADMIN}
    | {CREATEDB | NOCREATEDB}
    | {USEFT | NOUSEFT}
    | {CREATEROLE | NOCREATEROLE}
    | {INHERIT | NOINHERIT}
    | {LOGIN | NOLOGIN}
    | {REPLICATION | NOREPLICATION}
    | {PERSISTENCE | NOPERSISTENCE}
    | CONNECTION LIMIT connlimit
    | VALID BEGIN 'timestamp'
    | VALID UNTIL 'timestamp'
    | USER GROUP 'groupuser'
    | NODE GROUP logic_cluster_name
    | IN ROLE role_name [, ...]
    | IN GROUP role_name [, ...]
    | ROLE role_name [, ...]
    | ADMIN role_name [, ...]
    | USER role_name [, ...]
    | DEFAULT TABLESPACE tablespace_name
    | PROFILE DEFAULT
    | PROFILE profile_name
    | PGUSER
```

参数说明

- **user_name**

  用户名称。

  取值范围：字符串，要符合**标识符说明**，且最多为63个字符。若超过63个字符，数据库会截断并保留前63个字符当做用户名称。在创建用户时，数据库的时候会给出提示信息。用户名支持使用反引号括起来，当sql_mode设置为ANSI_QUOTES，也可以用双引号括起来。不用反引号、双引号括起来的用户名会被转换为小写。

  说明

  - 标识符需要为字母、下划线、数字（0~9）或美元符号（$），且必须以字母（a~z）或下划线（_）开头。
  - 通过CREATE USER创建用户时会自动创建一个同名的SCHEMA，在GUC参数lower_case_table_names为1大小写不敏感时，Schema名会自动转为小写，所以不建议用户创建含有大写字母的用户名。

- **password**

  登录密码。

  密码规则如下：

  - 密码默认不少于8个字符。
  - 不能与用户名及用户名倒序相同。
  - 至少包含大写字母（A-Z），小写字母（a-z），数字（0-9），非字母数字字符（限定为~!@#$%^&*()-_=+\|[{}];:,<.>/?）四类字符中的三类字符。
  - 密码也可以是符合格式要求的密文字符串，这种情况主要用于用户数据导入场景，不推荐用户直接使用。如果直接使用密文密码，用户需要知道密文密

码对应的明文，并且保证密码复杂度，数据库不会校验密文密码复杂度，直接使用密文密码的安全性由用户保证。

- 创建用户时，应当使用单引号将用户密码括起来。

取值范围：字符串。

CREATE USER的其他参数值请参考**CREATE ROLE**。

示例

```
--创建用户jim，登录密码为********。
m_db=# CREATE USER jim PASSWORD '********';

--下面语句与上面的等价。
m_db=# CREATE USER kim IDENTIFIED BY '********';

--如果创建有“创建数据库”权限的用户，则需要加CREATEDB关键字。
m_db=# CREATE USER dim CREATEDB PASSWORD '********';

--将用户jim的登录密码由********修改为**********。
m_db=# ALTER USER jim IDENTIFIED BY '**********' REPLACE '********';

--为用户jim追加CREATEROLE权限。
m_db=# ALTER USER jim CREATEROLE;

--将enable_seqscan的值设置为on， 设置成功后，在下一会话中生效。
m_db=# ALTER USER jim SET enable_seqscan TO on;

--重置jim的enable_seqscan参数。
m_db=# ALTER USER jim SET enable_seqscan TO DEFAULT;

--锁定jim账户。
m_db=# ALTER USER jim ACCOUNT LOCK;

--删除用户。
m_db=# DROP USER kim CASCADE;
m_db=# DROP USER jim CASCADE;
m_db=# DROP USER dim CASCADE;
```

相关链接

ALTER USER，**CREATE ROLE**，**DROP USER**

4.4.2.8.21 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。

- 将经常使用的数据定义为视图，可以将复杂的查询SQL语句进行封装。简化操作。
- 安全性，用户只能查询视图定义的数据。隐藏基表字段，保护数据库的数据结构。
- 简化用户权限的管理，只授予用户使用视图的权限。

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。

语法格式

```
CREATE [ OR REPLACE ] [ 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。

当不开启精度传递开关（m_format_behavior_compat_options不开启enable_precision_decimal选项）时，含有以下数据类型的字段，在query中不允许进行计算操作（如函数调用、使用操作符计算），仅允许直接的字段调用（如 SELECT col1 FROM table1），精度传递开关打开（m_format_behavior_compat_options开启enable_precision_decimal选项）时可以使用：

- BINARY[(n)]
- VARBINARY(n)
- CHAR[(n)]
- VARCHAR(n)
- TIME[(p)]
- DATETIME[(p)]
- TIMESTAMP[(p)]
- BIT[(n)]
- NUMERIC[(p[,s])]
- DECIMAL[(p[,s])]
- DEC[(p[,s])]
- FIXED[(p[,s])]
- FLOAT4[(p, s)]
- FLOAT8[(p,s)]
- FLOAT[(p)]
- REAL[(p, s)]
- FLOAT[(p, s)]
- DOUBLE[(p,s)]
- DOUBLE PRECISION[(p,s)]
- TEXT
- TINYTEXT
- MEDIUMTEXT
- LONGTEXT
- BLOB
- TINYBLOB
- MEDIUMBLOB
- LONGBLOB

对视图和子查询的更新、视图的插入和删除的约束涉及到的一些概念，解释如下：

- 连接视图：多张表JOIN创建的视图。
- 保留键表：对多表连接视图进行插入、更新、删除受到键保留表的限制。在多表视图中，若源表的每一行与视图中的每一行一一对应， 而不存在源表中一行数据在JOIN连接后在视图中对应多行数据的情况，则源表为保留键表。
- 顶层与底层关系：视图可能有多层嵌套，如一个视图由一个或多个视图或子查询构成。将当前DML直接操作的视图称为顶层，将构成视图的表、视图、及WITH子句中的表、视图等称为对应的底层关系。
- 可更新列：不是系统列或whole-row reference，直接引用基表中的用户列的列可更新。
- 可更新视图：可以对视图做插入、更新、删除操作的，称为可更新视图。可更新视图不包含DISTINCT、GROUP BY、HAVING、LIMIT、OFFSET子句 ，不包含集合运算（UNION

以及EXCEPT），并且不包含聚集函数、窗口函数、返回集合函数（array_agg、json_agg、generate_series等）。

- 当视图创建后，不允许使用REPLACE修改不可更新视图当中的列名、列类型，也不允许删除列；可更新视图允许更新视图列名、列类型，允许删除列。
- 不建议对嵌套视图中底层视图做修改列名、列类型以及删除列操作，会导致上层视图不可用。
- 不开启精度传递开关（m_format_behavior_compat_options不开启enable_precision_decimal选项）时创建的视图，与开启精度传递开关的视图在小数位数、计算结果小数值、浮点数的比较结果等方面可能出现不一致的情况。精度开关未开启情况下创建的视图，如果在精度开关开启后使用，建议打开精度开关重建视图。

参数说明

- **OR REPLACE**

  可选。如果视图已存在，则重新定义。

- **TEMPORARY**

  可选。创建一个临时视图。在当前会话结束时会自动删除掉视图。如果视图引用的任何表是临时表，视图将被创建为临时视图（不管SQL中有没有指定TEMPORARY）。

- **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**标识符说明**。

- **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**标识符说明**。

- **view_option_name [= view_option_value]**

  该子句为视图指定一个可选的参数。

  目前view_option_name支持的参数仅有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。

- **query**

  为视图提供行和列的SELECT或VALUES语句。

须知

若query包含指定分区表分区的子句，创建视图会将所指定分区的OID固化到系统表中。如果使用导致指定分区的OID发生变更的分区DDL语法，如DROP/SPLIT/MERGE该分区，则会导致视图不可用。需要重新创建视图。

- **WITH [CASCADED | LOCAL] CHECK OPTION**

  控制更新视图的行为，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查询显示。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。WITH CHECK OPTION选项可以指定为CASCADED或LOCAL，语义分别如下：

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。

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，否则底层视图定义的条件都不检查。

说明

- 如果指定了CHECK OPTION，无法对多表连接视图或多表连接子查询中的连接列进行插入、更新操作。
- 如果指定了CHECK OPTION，若多表连接视图或多表连接子查询中出现重复基表，且重复的基表不都是保留键表，则无法对该视图或子查询进行删除操作。

示例

- 普通视图：

```
--创建test_tb1表,并向表中插入100条数据。
m_db=# CREATE TABLE test_tb1(col1 int, col2 int);
m_db=# INSERT INTO test_tb1 VALUES (generate_series(1,100),generate_series(1,100));
--创建一个col1小于5的视图。
m_db=# CREATE VIEW test_v1 AS SELECT * FROM test_tb1 WHERE col1 < 3;
--查看视图。
m_db=# SELECT * FROM test_v1;
 col1 | col2
------+------
    1 |    1
    2 |    2
(2 rows)

--删除表和视图。
m_db=# DROP VIEW test_v1;
m_db=# DROP TABLE test_tb1;
```

- 临时视图：

```
--创建表和临时视图。
m_db=# CREATE TABLE test_tb2(c1 int, c2 int);
m_db=# CREATE TEMPORARY VIEW test_v2 AS SELECT * FROM test_tb2;
--查看表和视图信息（临时表所属模式不是public，而是以pg_temp开头的模式）。
m_db=# \d
                                    List of relations
        Schema          |   Name   | Type  | Owner |                      Storage
------------------------+----------+-------+-------+-----------------------------------------------------
 pg_temp_dn_6001_1_1_9473 | test_v2  | view  | omm   |
 public                  | test_tb2 | table | omm   |
{orientation=row,compression=no,storage_type=USTORE}
(2 rows)

--退出当前会话重新登录后，再次查看临时视图已经被删除。
m_db=# \d
                                 List of relations
 Schema |   Name   | Type  | Owner |                      Storage
--------+----------+-------+-------+-----------------------------------------------------
 public | test_tb2 | table | omm   | {orientation=row,compression=no,storage_type=USTORE}
(1 row)

--删除视图和表。
m_db=# DROP VIEW test_v2 ;
m_db=# DROP TABLE test_tb2 CASCADE;
```

相关链接

ALTER VIEW，**DROP VIEW**

4.4.2.9 D

4.4.2.9.1 DROP AUDIT POLICY

功能描述

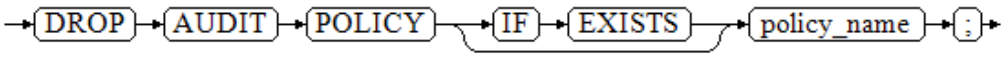
删除一个审计策略。

注意事项

只有POLADMIN、SYSADMIN或初始用户才能进行此操作。

语法格式

```
DROP AUDIT POLICY [IF EXISTS] policy_name;
```



参数说明

- **policy_name**
  审计策略名称，需要唯一，不可重复。
  取值范围：字符串，要符合**标识符命名规范**。
- **IF EXISTS**
  判断审计策略是否存在。如果存在则删除成功，否则发出一个NOTICE信息。

示例

```
--创建adt1策略。
m_db=# CREATE AUDIT POLICY adt1 PRIVILEGES CREATE;
CREATE AUDIT POLICY

--删除审计策略adt1。
m_db=# DROP AUDIT POLICY adt1;
DROP AUDIT POLICY

--删除一个不存在的审计策略adt0,提示删除失败，该审计策略不存在。
m_db=# DROP AUDIT POLICY adt0;
ERROR:  adt0 policy does not exist, drop failed
```

相关链接

ALTER AUDIT POLICY，**CREATE AUDIT POLICY**。

4.4.2.9.2 DEALLOCATE

功能描述

用于删除前面编写的预备语句。

注意事项

如果用户没有明确删除一个预备语句，那么它将在会话结束的时候被删除。

语法格式

```
{DEALLOCATE | DROP}  PREPARE name;
```

参数说明

name

将要删除的预备语句。

示例

```
--创建q1,q2,q3三个预备语句，先删除q1，再删除所有预备语句。
m_db=# SELECT name, statement, parameter_types FROM pg_prepared_statements;
 name | statement | parameter_types
------+-----------+-----------------
(0 rows)

m_db=# PREPARE q1 from  'SELECT 1 AS a';
PREPARE
m_db=# PREPARE q2 from 'SELECT 1 AS a';
PREPARE
m_db=# PREPARE q3 from 'SELECT 1 AS a';
PREPARE
m_db=# SELECT name, statement, parameter_types FROM pg_prepared_statements;
 name |          statement           | parameter_types
------+------------------------------+-----------------
 q1   | PREPARE q1 AS SELECT 1 AS a; | {}
 q3   | PREPARE q3 AS SELECT 1 AS a; | {}
 q2   | PREPARE q2 AS SELECT 1 AS a; | {}
(3 rows)

m_db=# DEALLOCATE PREPARE q1;
DEALLOCATE
m_db=# SELECT name, statement, parameter_types FROM pg_prepared_statements;
 name |          statement           | parameter_types
------+------------------------------+-----------------
 q3   | PREPARE q3 AS SELECT 1 AS a; | {}
 q2   | PREPARE q2 AS SELECT 1 AS a; | {}
(2 rows)
```

4.4.2.9.3 DELETE

功能描述

DELETE从指定的表里删除满足WHERE子句的行。如果WHERE子句不存在，将删除表中所有行，结果只保留表结构。

注意事项

- 表的所有者、被授予表DELETE权限的用户或被授予DELETE ANY TABLE权限的用户有权删除表中数据，当三权分立开关关闭时，系统管理员默认拥有此权限。同时也必须有USING子句引用的表以及condition上读取表的SELECT权限。
- 对于多表删除语法，暂时不支持对视图进行多表删除。
- 对于子查询是STREAM计划的DELETE语句，不支持删除的行数据同时进行UPDATE更新操作。

语法格式

单表删除：

```
[ WITH [ RECURSIVE ] with_query [, ...] ]
DELETE [/*+ plan_hint */] [IGNORE] FROM {table_name [partition_clause] | view_name}
   [ WHERE condition ]
   [ ORDER BY { expression  [ ASC | DESC ] } [ NULLS { FIRST | LAST } ] ]
   [ LIMIT { count } ];
```


多表删除：

```
[ WITH [ RECURSIVE ] with_query [, ...] ]
DELETE [/*+ plan_hint */] [IGNORE]
    FROM table_name1[ .* ] [,table_name2[ .* ]] ...
    USING using_list
    [ WHERE condition ];
```

或：

```
[ WITH [ RECURSIVE ] with_query [, ...] ]
DELETE [/*+ plan_hint */] [IGNORE]
    table_name1[ .* ][,table_name2[ .* ]] ...
    FROM using_list
    [ WHERE condition ];
```

参数说明

- **WITH [RECURSIVE] with_query [, ...]**

  用于声明一个或多个可以在主查询中通过名称引用的子查询，相当于临时表。这种子查询语句结构称为CTE（Common Table Expression）结构，应用这种结构时，执行计划中将存在CTE SCAN的内容。

  如果声明了RECURSIVE，那么允许SELECT子查询通过名称引用其本身。

  其中with_query的详细格式为：

  ```
  with_query_name [ ( column_name [, ...] ) ] AS ( {select} )
  ```

  - with_query_name指定子查询生成的结果集名称，在查询中可使用该名称访问子查询的结果集。
  - column_name指定子查询结果集中显示的列名。
  - 每个子查询支持SELECT语句。
  - RECURSIVE只能出现在WITH后面，多个CTE的情况下，只需要在第一个CTE处声明RECURSIVE。
  - 使用RECURSIVE时，CTE子查询中UNION ALL和EXCEPT ALL或UNION [DISTINCT]和EXCEPT [DISTINCT]两侧的子查询结果，其数据类型必须使用cast函数转换成相同的数据类型，且两侧子查询结果的精度和字符序也要相同。如：WITH RECURSIVE cte (n) AS (SELECT cast(id as signed int) from table_1 UNION ALL SELECT cast((n + 1) as signed int) FROM cte WHERE n < 5) SELECT * FROM cte。由操作符产生的类型转换具体请参见**逻辑操作符规格约束**、**位运算操作符规格约束**和**算术操作符规格约束**。

- **plan_hint**

  以/*+ */的形式在DELETE关键字后，用于对DELETE对应的语句块生成的计划进行hint调优。每条语句中只有第一个/*+ plan_hint */注释块会作为hint生效，里面可以写多条hint。

- **IGNORE**

  DELETE语句使用IGNORE关键字时，可将部分ERROR级别的错误降级为WARNING级别，并根据不同的错误场景将无效值调整为最接近的值。GaussDB支持错误降级的场景如下：

  sql_mode为宽松模式的场景。

  说明

  升级观察期不支持IGNORE。

- **table_name**

  目标表的名称（可以有模式修饰）。

取值范围：已存在的表名。

- **partition_clause**

  指定分区删除操作。

  PARTITION { (partition_name | subpartition_name [, ...])

  关键字详见**SELECT**章节介绍。

  示例详见**CREATE TABLE SUBPARTITION**。

- **view_name**

  目标视图的名称。

  说明

  对视图的删除，有如下约束：

  - 只有直接引用基表用户列的列可进行DELETE操作。
  - 视图必须至少包含一个可更新列，关于可更新列请参见**CREATE VIEW**。
  - 不支持在顶层包含DISTINCT、GROUP BY、HAVING、LIMIT、OFFSET子句的视图。
  - 不支持在顶层包含集合运算（UNION以及EXCEPT）的视图。
  - 不支持目标列表中包含聚集函数、窗口函数、返回集合函数（array_agg、json_agg、generate_series等）的视图。
  - 视图中支持的表类型包括普通表、临时表、全局临时表、分区表、二级分区表、ustore表、astore表。
  - 连接视图只能对视图中的保留键表做删除操作，如果只存在一张保留键表，则删除该表数据，如果存在多张保留键表，仅删除FROM后的第一张保留键表的数据。关于保留键表请参见**CREATE VIEW**。
  - 不支持对系统视图进行DELETE操作。
  - 不支持多表删除功能。

- **NULLS FIRST**

  指定空值在排序中排在非空值之前，当指定DESC排序时，本选项为默认的。

- **NULLS LAST**

  指定空值在排序中排在非空值之后，未指定DESC排序时，本选项为默认的。

- **using_list**

  using子句。

须知

当删除多张目标表时，using_list指定关联表的集合时可以同时出现目标表，并且可以定义表的别名并在目标表中使用。其他情况下则目标表不可重复出现在using_list中。

设置GUC兼容性参数m_format_dev_version为's2'后，所有目标表需出现在using_list中，且两者的表名、别名、所属库名均相等时才满足匹配条件。匹配规则如下所示，如果不存在唯一匹配，语法执行失败：

1. 如果目标表使用库名.表名的形式，则表名字段只能是真实表名，而非别名引用，且只能与using_list中未指定别名的对象进行匹配。
2. 如果目标表未指定库名，默认属于当前正在使用的库名；
3. 如果using_list中对象指定别名，按别名匹配。

- **condition**

  一个返回Boolean值的表达式，用于判断哪些行需要被删除。不建议使用int等数值类型作为condition，因为int等数值类型可以隐式转换为bool值（非0值隐式转换为true，0转换为false），可能导致非预期的结果。

- **ORDER BY**

  关键字详见**SELECT**章节介绍。

- **LIMIT**

  关键字详见**SELECT**章节介绍。

示例

- **删除部分数据**

```
--建表。
m_db=# CREATE TABLE test_t1(col1 INT,col2 INT);
m_db=# INSERT INTO test_t1 VALUES (1, 1), (2, 2), (3, 3), (4, 4), (4, 6);

--删除表中所有记录。
m_db=# DELETE FROM test_t1 WHERE col1 = 4;

--查询。
m_db=# SELECT * FROM test_t1;
 col1 | col2
------+------
    1 |    1
    2 |    2
    3 |    3
(3 rows)
```

- **删除所有数据**

```
--删除所有的数据。
m_db=# DELETE FROM test_t1;

--查询。
m_db=# SELECT * FROM test_t1;
 col1 | col2
------+------
(0 rows)

--删除表。
m_db=# DROP TABLE test_t1;
```

- **删除视图**

```
--创建SCHEMA。
m_db=# CREATE SCHEMA del_view;
CREATE SCHEMA
m_db=# SET CURRENT_SCHEMA = 'del_view';
SET

--创建表并插入数据。
m_db=# CREATE TABLE t1 (x1 int, y1 int);
CREATE TABLE
m_db=# CREATE TABLE t2 (x2 int PRIMARY KEY, y2 int);
NOTICE:  CREATE TABLE / PRIMARY KEY will create implicit index "t2_pkey" for table "t2"
CREATE TABLE
m_db=# CREATE TABLE tdata (x INT PRIMARY KEY, y INT);
NOTICE:  CREATE TABLE / PRIMARY KEY will create implicit index "tdata_pkey" for table "tdata"
CREATE TABLE
m_db=# CREATE TABLE tinfo (z INT PRIMARY KEY, comm VARCHAR(20));
NOTICE:  CREATE TABLE / PRIMARY KEY will create implicit index "tinfo_pkey" for table "tinfo"
CREATE TABLE
m_db=# INSERT INTO t1 VALUES (1, 1), (2, 2), (3, 3), (5, 5);
INSERT 0 4
m_db=# INSERT INTO t2 VALUES (1, 1), (2, 2), (3, 3), (5, 5);
INSERT 0 4
m_db=# INSERT INTO tdata VALUES (1, 1), (2, 2), (3, 3);
```



```
INSERT 0 3
m_db=# INSERT INTO tinfo VALUES (1,'one'), (2, 'two'), (3, 'three'), (5, 'wrong three');
INSERT 0 4

--创建单表视图。
m_db=# CREATE VIEW v_del1 AS SELECT * FROM t1;
CREATE VIEW

--通过视图删除t1中的数据。
m_db=# DELETE FROM v_del1 where y1 = 3;
DELETE 1

--创建多表连接视图。
m_db=# CREATE VIEW vvt1t2 AS SELECT * FROM t1, t2 WHERE x1 = x2;
CREATE VIEW
m_db=# CREATE VIEW vv_dup AS SELECT td1.x x1, td1.y y1, td2.x x2, td2.y y2 FROM tdata td1, tdata
td2, tinfo WHERE td2.y=tinfo.z AND td1.x=td2.y;
CREATE VIEW

--对多表连接的视图做删除操作。
m_db=# SELECT * FROM vvt1t2;
 x1 | y1 | x2 | y2
----+----+----+----
  1 |  1 |  1 |  1
  2 |  2 |  2 |  2
  5 |  5 |  5 |  5
(3 rows)

m_db=# DELETE FROM vvt1t2 WHERE y2 = 5;
DELETE 1
m_db=# SELECT * FROM vvt1t2;
 x1 | y1 | x2 | y2
----+----+----+----
  1 |  1 |  1 |  1
  2 |  2 |  2 |  2
(2 rows)

m_db=# SELECT * FROM vv_dup;
 x1 | y1 | x2 | y2
----+----+----+----
  1 |  1 |  1 |  1
  2 |  2 |  2 |  2
  3 |  3 |  3 |  3
(3 rows)

m_db=# DELETE FROM vv_dup WHERE y1 = 2;
DELETE 1
m_db=# SELECT * FROM vv_dup;
 x1 | y1 | x2 | y2
----+----+----+----
  1 |  1 |  1 |  1
  3 |  3 |  3 |  3
(2 rows)

--删除SCHEMA。
m_db=# DROP SCHEMA del_view;
NOTICE:  drop cascades to 7 other objects
DETAIL:  drop cascades to table t1
drop cascades to table t2
drop cascades to table tdata
drop cascades to table tinfo
drop cascades to view v_del1
drop cascades to view vvt1t2
drop cascades to view vv_dup
DROP SCHEMA
```

优化建议

- delete

如果要删除表中的所有记录，建议使用truncate语法。

4.4.2.9.4 DESCRIBE

功能描述

DESCRIBE显示指定表中列的信息，同时也适用于视图。

显示的列信息包括：

- Field：列名。
- Type：列数据类型。
- Null：列值是否可以为NULL，可以为YES，否则为NO。
- Key：列是否被索引，取值范围：
  - PRI：表示该列被主键索引。
  - UNI：表示该列是唯一索引中的第一列。
  - MUL：表示该列是非唯一索引中的第一列。
  - NULL：表示该列没有被索引，或者不是唯一索引或非唯一索引的第一列。
- Default：列默认值。
- Extra：列的附加信息，取值范围：
  - auto_increment：表示该列为自增列。
  - on update CURRENT_TIMESTAMP：表示该列具有ON UPDATE约束。
  - VIRTUAL GENERATED：表示该列为虚拟生成列。
  - STORED GENERATED：表示该列为存储生成列。

注意事项

- DESCRIBE是**SHOW COLUMNS**语法的快捷键，同时DESCRIBE还可以显示视图的信息。
- 默认情况下，DESCRIBE显示表中所有列的信息。若给出表中列的名称，在这种情况下该语句仅显示指定列的信息；若给出带有%或_的通配符的字符串，在这种情况下该语句仅显示名称与字符串匹配的列的输出。除非字符串包含空格或其他特殊字符，否则无需将字符串括在引号内。

语法格式

```
{DESCRIBE | DESC}
  tbl_name [col_name | wild]
```

参数说明

wild

一个模式字符串，它可以包含%和_通配符字符。

示例

```
-- 创建表t01
m_db=# CREATE TABLE t01 (
    c1 int AUTO_INCREMENT COMMENT 'this is c1',
    c2 int NOT NULL CHECK (c2 > 0),
    c3 text CHARACTER SET utf8mb4 COLLATE utf8mb4_bin NULL COMMENT 'this is c3',
```



```
  c4 int DEFAULT 5 COMMENT 'this is c4',
  c5 datetime ON UPDATE current_timestamp COMMENT 'this is c5',
  c6 int GENERATED ALWAYS AS (c2 + c4) STORED COMMENT 'this is c6',
  c7 int UNIQUE COMMENT 'this is c7',
  c8 int,
  c9 text CHARSET utf8mb4 COLLATE utf8mb4_bin COMMENT 'this is c9',
  c10 int COMMENT 'this is c10',
  c11 int COMMENT 'this is c11',
  c12 int COMMENT 'this is c12',
  CONSTRAINT PRIMARY KEY (c1) USING btree COMMENT 'this is primary key',
  CONSTRAINT FOREIGN KEY (c8) REFERENCES t01(c1),
  CONSTRAINT CHECK (c10 > 0),
  CONSTRAINT UNIQUE (c11) COMMENT 'this is unique constraint',
  c13 text COMMENT 'this is c13',
  c14 int COMMENT 'this is c14',
  INDEX USING btree (c13(10)) COMMENT 'this is index',
  INDEX (c13(10),c14) USING btree COMMENT 'this is index too'
);

-- 显示列信息
m_db=# DESCRIBE t01;
 Field |   Type   | Null | Key | Default |           Extra
-------+----------+------+-----+---------+----------------------------
 c1    | integer  | NO   | PRI |         | auto_increment
 c2    | integer  | NO   |     |         |
 c3    | text     | YES  |     |         |
 c4    | integer  | YES  |     | 5       |
 c5    | datetime | YES  |     |         | on update CURRENT_TIMESTAMP
 c6    | integer  | YES  |     |         | STORED GENERATED
 c7    | integer  | YES  | UNI |         |
 c8    | integer  | YES  |     |         |
 c9    | text     | YES  |     |         |
 c10   | integer  | YES  |     |         |
 c11   | integer  | YES  | UNI |         |
 c12   | integer  | YES  |     |         |
 c13   | text     | YES  | MUL |         |
 c14   | integer  | YES  |     |         |
(14 rows)

-- 模糊匹配列信息
m_db=# DESCRIBE t01 c_;
 Field |   Type   | Null | Key | Default |           Extra
-------+----------+------+-----+---------+----------------------------
 c1    | integer  | NO   | PRI |         | auto_increment
 c2    | integer  | NO   |     |         |
 c3    | text     | YES  |     |         |
 c4    | integer  | YES  |     | 5       |
 c5    | datetime | YES  |     |         | on update CURRENT_TIMESTAMP
 c6    | integer  | YES  |     |         | STORED GENERATED
 c7    | integer  | YES  | UNI |         |
 c8    | integer  | YES  |     |         |
 c9    | text     | YES  |     |         |
(9 rows)

m_db=# DESCRIBE t01 c%;
 Field |   Type   | Null | Key | Default |           Extra
-------+----------+------+-----+---------+----------------------------
 c1    | integer  | NO   | PRI |         | auto_increment
 c2    | integer  | NO   |     |         |
 c3    | text     | YES  |     |         |
 c4    | integer  | YES  |     | 5       |
 c5    | datetime | YES  |     |         | on update CURRENT_TIMESTAMP
 c6    | integer  | YES  |     |         | STORED GENERATED
 c7    | integer  | YES  | UNI |         |
 c8    | integer  | YES  |     |         |
 c9    | text     | YES  |     |         |
 c10   | integer  | YES  |     |         |
 c11   | integer  | YES  | UNI |         |
 c12   | integer  | YES  |     |         |
```



```
 c13   | text     | YES  | MUL |         |
 c14   | integer  | YES  |     |         |
(14 rows)

m_db=# DROP TABLE t01;
```

相关链接

EXPLAIN、**SHOW**

4.4.2.9.5 DO

功能描述

该语法仅内部工具使用。

执行匿名代码块。

代码块被看作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。

注意事项

- 目前只支持plpgsql安装语言。
- 如果语言是不受信任的，用户必须有使用程序语言的USAGE权限，或者是系统管理员权限。

语法格式

```
DO code;
```

参数说明

- **code**
  可以被执行的程序语言代码，必须指定为字符串。

示例

```
--创建用户webuser。
m_db=# CREATE USER webuser PASSWORD '********';

--授予用户webuser对模式tpcds下视图的所有操作权限。
m_db=# DO $$DECLARE r record;
BEGIN
    FOR r IN SELECT c.relname table_name,n.nspname table_schema FROM pg_class c,pg_namespace n
            WHERE c.relnamespace = n.oid AND n.nspname = 'tpcds' AND relkind IN ('r','v')
    LOOP
        EXECUTE 'GRANT ALL ON ' || quote_ident(r.table_schema) || '.' || quote_ident(r.table_name) || ' TO
webuser';
    END LOOP;
END$$;

--删除用户webuser。
m_db=# DROP USER webuser CASCADE;
```

4.4.2.9.6 DROP DATABASE

功能描述

M-Compatibility中DATABASE与SCHEMA是同义词，删除数据库就是删除模式。

注意事项

- 只有数据库所有者或者被授予了数据库DROP权限的用户有权限执行DROP DATABASE命令，系统管理员默认拥有此权限。
- 除初始用户和运维管理员外，其他用户无法DROP掉运维管理员的DATABASE。

须知

- DROP DATABASE一旦执行将无法撤销，请谨慎使用。
- 不要随意删除pg_temp或pg_toast_temp开头的模式，这些模式是系统内部使用的，如果删除，可能导致无法预知的结果。

语法格式

```
DROP DATABASE [ IF EXISTS ] database_name;
```

参数说明

- **IF EXISTS**

  如果指定的数据库/模式不存在，则发出一个notice而不是抛出一个错误。
- **database_name**

  要删除的数据库/模式名称。

  取值范围：字符串，已存在的数据库名称。

示例

请参见CREATE DATABASE的**示例**。

相关链接

ALTER DATABASE，**CREATE DATABASE**

4.4.2.9.7 DROP EXTENSION

功能描述

当前不支持用户使用此语法。删除一个扩展。该特性为内部使用，不建议用户使用。

注意事项

- DROP EXTENSION 命令从数据库中删除一个扩展。 在删除扩展的过程中，构成扩展的组件也会一起删除。
- 必须是扩展的拥有者才能够使用DROP EXTENSION命令。

语法格式

```
DROP EXTENSION [ IF EXISTS ] name [, ...] [ CASCADE | RESTRICT ];
```


参数说明

- **IF EXISTS**

  当使用IF EXISTS参数，如果扩展不存在时，不会抛出错误，而是产生一个通知。

- **name**

  已经安装的扩展模块的名称。

- **CASCADE**

  自动删除依赖于该扩展的对象。

- **RESTRICT**

  如果有依赖于扩展的对象，则不允许删除此扩展（除非它所有的成员对象和其它扩展对象在一条 DROP命令一起删除）。 这是缺省行为。

示例

从当前数据库中删除扩展plpgsql，plpgsql为数据库默认创建extension。

```
m_db=# DROP EXTENSION IF EXISTS plpgsql;
```

在当前数据库中，如果有使用plpgsql的对象的，这条命令就会失败，比如任一表中的字段使用plpgsql类型。这时增加CASCADE选项会强制删除扩展和依赖于扩展的对象。

相关链接

ALTER EXTENSION，**CREATE EXTENSION**

4.4.2.9.8 DROP FUNCTION

功能描述

删除一个已存在的函数。

说明

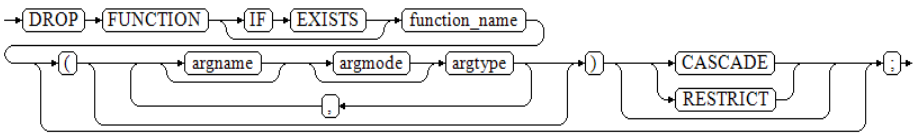
内部使用功能，不支持用户使用。

注意事项

- 如果函数中涉及对临时表的相关操作，则无法使用DROP FUNCTION删除函数。
- 只有函数的所有者或者被授予了函数DROP权限的用户才能执行DROP FUNCTION命令，系统管理员默认拥有该权限。

语法格式

```
DROP FUNCTION [ IF EXISTS ] function_name
[ ( [ {[ argname ] [ argmode ] argtype} [, ...] ] ) [ CASCADE | RESTRICT ] ];
```



参数说明

- **IF EXISTS**

  表示如果函数存在则执行删除操作，函数不存在也不会报错，只是产出一个NOTICE。

- **function_name**

  要删除的函数名称。

  取值范围：已存在的函数名。

- **argmode**

  函数参数的模式。

- **argname**

  函数参数的名称。

- **argtype**

  函数参数的类型

- **CASCADE | RESTRICT**
  - CASCADE：级联删除依赖于函数的对象 。
  - RESTRICT：如果有任何依赖对象存在，则拒绝删除该函数（缺省行为）。

示例

- 删除函数可省略参数列表

```
--创建函数。
m_db=# CREATE OR REPLACE FUNCTION func_add_sql(integer, integer) RETURNS bigint
AS 'select $1 + $2;'
LANGUAGE SQL
IMMUTABLE
SHIPPABLE
STRICT
AUTHID DEFINER
not fenced
COST 10000;
/

--删除函数。
m_db=# DROP FUNCTION func_add_sql;
```

- 删除存在同名的函数

  如果存在同名函数，删除时需加上参数列表，否则报错。

```
--创建函数。
m_db=# CREATE OR REPLACE FUNCTION func_increment_plsql(i integer) RETURNS integer AS $$
BEGIN
    RETURN i + 1;
END;
$$ LANGUAGE plpgsql;


--重载函数func_add。
m_db=# CREATE OR REPLACE FUNCTION func_add(int) RETURNS int AS $$
BEGIN
    RETURN $1+10;
END;
$$ LANGUAGE PLPGSQL;

--删除函数。
m_db=# DROP FUNCTION func_add(int);
m_db=# DROP FUNCTION func_add(int,int);
```


相关链接

CREATE FUNCTION

4.4.2.9.9 DROP GROUP

功能描述

删除用户组。

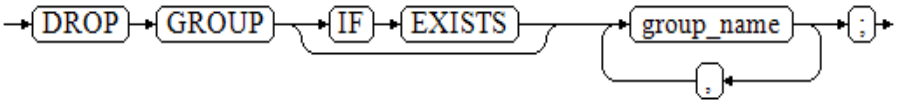
DROP GROUP是DROP ROLE的别名。

注意事项

DROP GROUP是M-Compatibility管理工具封装的接口，用来实现M-Compatibility管理。该接口不建议用户直接使用，以免对M-Compatibility状态造成影响。

语法格式

```
DROP GROUP [ IF EXISTS ] group_name [, ...];
```



参数说明

- **IF EXISTS**
  如果不存在该角色，不会报出错误，而会发出通知，告知通知该角色不存在。
- **group_name**
  要删除的角色名称。
  取值范围：已存在的角色。

示例

请参见CREATE GROUP的**示例**。

相关链接

ALTER GROUP，**CREATE GROUP**，**DROP ROLE**

4.4.2.9.10 DROP INDEX

功能描述

删除索引。

注意事项

- 索引的所有者、索引所在模式的所有者、拥有索引所在表的INDEX权限的用户或者被授予了DROP ANY INDEX权限的用户有权限执行DROP INDEX命令，三权分立关闭时的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，其他会话无法执行该表上索引的删除操作。

语法格式

```
DROP INDEX [ CONCURRENTLY ] [ IF EXISTS ]
   index_name [, ...] [ CASCADE | RESTRICT];

DROP INDEX index_name [, ...] [CASCADE | RESTRICT] ON table_name [algorithm_option | lock_option];
```

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，在删除过程中可避免阻塞。

  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ERROR。

- **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：表示有依赖于此索引的对象存在，则该索引无法被删除。此选项为缺省值。

- **table_name**

  表名。

- **algorithm_option**

  参考**CREATE INDEX**。

- **lock_option**

  参考**CREATE INDEX**。

示例

```
-- 建表
m_db=# CREATE TABLE test1_index (id INT, name VARCHAR(20));
m_db=# CREATE INDEX idx_test1 on test1_index (id);

-- 删除索引
m_db=# DROP INDEX IF EXISTS idx_test1 CASCADE;

-- 删除表
m_db=# DROP TABLE test1_index;
```

相关链接

ALTER INDEX，**CREATE INDEX**

4.4.2.9.11 DROP OWNED

功能描述

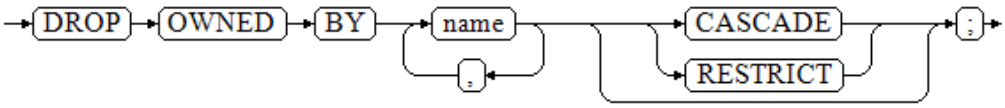
删除一个数据库角色所拥有的数据库对象的权限。

注意事项

- 所有该角色在当前数据库里和共享对象（数据库，表空间）上的所有对象上的权限都将被撤销。
- DROP OWNED常常被用来为移除一个或者多个角色做准备。因为DROP OWNED只影响当前数据库中的对象，通常需要在包含将被移除角色所拥有的对象的每一个数据库中都执行这个命令。
- 使用CASCADE选项可能导致这个命令递归去删除由其他用户所拥有的对象。
- 角色所拥有的数据库、表空间将不会被移除。

语法格式

```
DROP OWNED BY name [, ...] [ CASCADE | RESTRICT ];
```



参数说明

- **name**

  角色名。
- **CASCADE | RESTRICT**
  - CASCADE：级联删除所有依赖于被删除对象的对象。
  - RESTRICT：拒绝删除那些有任何依赖对象存在的对象。此选项为缺省值。

示例

```
--创建jim用户。
m_db=# CREATE USER jim PASSWORD '********';

--撤销jim在当前数据库里和共享对象（数据库，表空间）上的所有对象上的权限。
m_db=# DROP OWNED BY jim;

--删除jim用户。
m_db=# DROP USER jim;
```

相关链接

DROP ROLE

4.4.2.9.12 DROP PREPARE

功能描述

用于删除已编写的预备语句。

注意事项

如果用户没有明确删除一个预备语句，那么它将在会话结束的时候被删除。

语法格式

```
{DEALLOCATE | DROP}  PREPARE name;
```

示例

请参见DEALLOCATE的**示例**。

相关链接

PREPARE

4.4.2.9.13 DROP RESOURCE LABEL

功能描述

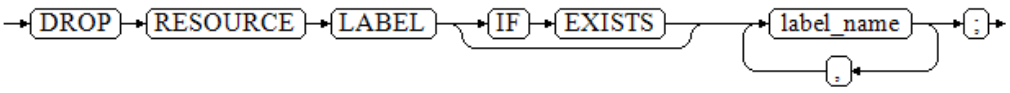
DROP RESOURCE LABEL语句用于删除资源标签。

注意事项

只有POLADMIN、SYSADMIN或初始用户才能执行此操作。

语法格式

```
DROP RESOURCE LABEL [IF EXISTS] label_name[, ...];
```



参数说明

- **label_name**
  资源标签名称。
  取值范围：字符串，要符合**标识符命名规范**。

示例

```
--创建一个表tb_for_label。
m_db=# CREATE TABLE tb_for_label(col1 text, col2 text, col3 text);

--基于表创建资源标签。
m_db=# CREATE RESOURCE LABEL IF NOT EXISTS table_label add TABLE(public.tb_for_label);

--再次创建已存在的表资源标签，对比加参数IF NOT EXISTS与不加IF NOT EXISTS参数的区别。
m_db=# CREATE RESOURCE LABEL IF NOT EXISTS table_label add TABLE(public.tb_for_label);
NOTICE:  table_label label already defined, skipping
CREATE RESOURCE LABEL
m_db=# CREATE RESOURCE LABEL table_label add TABLE(public.tb_for_label);
ERROR:  table_label label already defined

--基于列创建资源标签。
m_db=# CREATE RESOURCE LABEL IF NOT EXISTS column_label add COLUMN(public.tb_for_label.col1);
```



```
--创建一个模式schema_for_label。
m_db=# CREATE SCHEMA schema_for_label;

--基于模式创建资源标签。
m_db=# CREATE RESOURCE LABEL IF NOT EXISTS schema_label add SCHEMA(schema_for_label);

--创建一个视图view_for_label。
m_db=# CREATE VIEW view_for_label AS SELECT 1;

--基于视图创建资源标签。
m_db=# CREATE RESOURCE LABEL IF NOT EXISTS view_label add VIEW(view_for_label);

--创建一个函数func_for_label。
m_db=# CREATE FUNCTION func_for_label RETURNS TEXT AS $$ SELECT col1 FROM tb_for_label; $$
LANGUAGE SQL;

--基于函数创建资源标签。
m_db=# CREATE RESOURCE LABEL IF NOT EXISTS func_label add FUNCTION(func_for_label);

--删除表资源标签table_label。
m_db=# DROP RESOURCE LABEL IF EXISTS table_label;

--删除列资源资源标签column_label。
m_db=# DROP RESOURCE LABEL IF EXISTS column_label;

--删除函数资源标签func_for_label。
m_db=# DROP FUNCTION func_for_label;

--删除视图资源标签view_for_label。
m_db=# DROP VIEW view_for_label;

--删除模式资源标签schema_for_label。
m_db=# DROP SCHEMA schema_for_label;

--删除表tb_for_label。
m_db=# DROP TABLE tb_for_label;
```

相关链接

ALTER RESOURCE LABEL，**CREATE RESOURCE LABEL**。

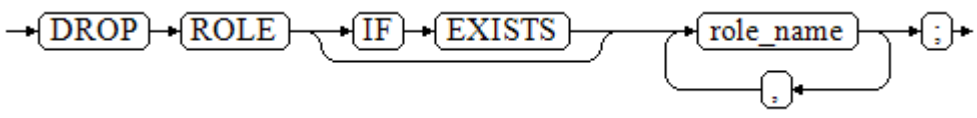
4.4.2.9.14 DROP ROLE

功能描述

删除指定的角色。

语法格式

```
DROP ROLE [ IF EXISTS ] role_name [, ...];
```



参数说明

- **IF EXISTS**
  如果指定的角色不存在，则发出一个notice而不是抛出一个错误。
- **role_name**
  要删除的角色名称。

取值范围：已存在的角色名，角色名要求详见•role_name。

示例

请参见CREATE ROLE的示例。

相关链接

CREATE ROLE，ALTER ROLE，SET ROLE

4.4.2.9.15 DROP SCHEMA

功能描述

从数据库中删除模式。

注意事项

- 只有模式的所有者或者被授予了模式DROP权限的用户有权限执行DROP SCHEMA命令，当三权分立开关关闭时，系统管理员默认拥有此权限。
- 除初始用户和运维管理员外，其他用户无法DROP掉运维管理员的SCHEMA。

语法格式

```
DROP SCHEMA [ IF EXISTS ] schema_name [, ...];
```

参数说明

- **IF EXISTS**

  如果指定的模式不存在，发出一个notice而不是抛出一个错误。

- **schema_name**

  模式的名称。

  取值范围：已存在模式名。

须知

不要随意删除pg_temp或pg_toast_temp开头的模式，这些模式是系统内部使用的，如果删除，可能导致无法预知的结果。

示例

请参见CREATE SCHEMA的示例。

相关链接

ALTER SCHEMA，CREATE SCHEMA。

4.4.2.9.16 DROP SEQUENCE

功能描述

从当前数据库里删除序列。

注意事项

序列的所有者、序列所在模式的所有者、被授予了序列DROP权限的用户或者被授予了DROP ANY SEQUENCE权限的用户才能删除，当三权分立开关关闭时，系统管理员默认拥有该权限。

语法格式

```
DROP SEQUENCE [ IF EXISTS ] {[schema.]sequence_name} [ , ... ] [ CASCADE | RESTRICT ];
```

参数说明

- **IF EXISTS**
  如果指定的序列不存在，则发出一个notice而不是抛出一个错误。
- **sequence_name**
  序列名称。
- **CASCADE**
  级联删除依赖序列的对象。
- **RESTRICT**
  如果存在任何依赖的对象，则拒绝删除序列。此项是缺省值。

示例

```
--创建一个名为serial的递增序列，从101开始。
m_db=# CREATE SEQUENCE serial START 101;

--删除序列。
m_db=# DROP SEQUENCE serial;
```

相关链接

ALTER SEQUENCE， **CREATE SEQUENCE**

4.4.2.9.17 DROP TABLE

功能描述

删除指定的表。

注意事项

- DROP TABLE会强制删除指定的表，删除表后，依赖该表的索引会被删除。删除分区表，会同时删除分区表中的所有分区。
- 表的所有者、表所在模式的所有者、被授予了表的DROP权限的用户或被授予DROP ANY TABLE权限的用户，有权删除指定表 ，三权分立关闭时的系统管理员默认拥有该权限。

语法格式

```
DROP [TEMPOARARY] TABLE [ IF EXISTS ]
   { [schema.]table_name } [, ...] [ CASCADE | RESTRICT ] [ PURGE ];
```

参数说明

- **IF EXISTS**

  如果指定的表不存在，则发出一个notice而不是抛出一个错误。

- **schema**

  模式名称。

- **table_name**

  表名称。

- **CASCADE | RESTRICT**
  - CASCADE：表示允许级联删除依赖于该表的对象（比如视图）。
  - RESTRICT（缺省项）：如果存在依赖对象，则拒绝删除该表。这个是缺省。

    说明

    该属性在数据库M-compatibility模式兼容版本控制开关s1及以上版本（如m_format_dev_version = 's1'）时仅语法支持，但实际不生效。

- **PURGE**

  该参数表示即使开启回收站功能，drop表时，也会直接物理删除表，而不是将其放入回收站中。

示例

```
--创建test表。
m_db=# CREATE TABLE test(c1 int, c2 int);
--删除test表。
m_db=# DROP TABLE IF EXISTS test;

--视图依赖表时，删除表失败。
m_db=# create schema s1;
m_db=# create table s1.t1(a int);
m_db=# create view s1.v1 as select * from s1.t1;
m_db=# create view s1.v2 as select * from s1.v1;
m_db=# drop table s1.t1;
ERROR:  cannot drop table s1.t1 because other objects depend on it
DETAIL:  view s1.v1 depends on table s1.t1
view s1.v2 depends on view s1.v1
HINT:  Please drop the dependent first.
m_db=# drop table s1.t1 cascade;
ERROR:  cannot drop table s1.t1 because other objects depend on it
DETAIL:  view s1.v1 depends on table s1.t1
view s1.v2 depends on view s1.v1
HINT:  Please drop the dependent first.
```

相关链接

ALTER TABLE，**CREATE TABLE**

4.4.2.9.18 DROP USER

功能描述

删除用户，同时会删除同名的schema。

注意事项

- 须使用CASCADE级联删除依赖用户的对象（除数据库外）。当删除用户的级联对象时，如果级联对象处于锁定状态，则此级联对象无法被删除，直到对象被解锁或锁定级联对象的线程被终止。
- 在M-Compatibility中，存在一个配置参数enable_kill_query，此参数在配置文件gaussdb.conf中。此参数影响级联删除用户对象的行为：
  - 当参数enable_kill_query为on ，且使用CASCADE模式删除用户时，会自动kill锁定用户级联对象的线程，并删除用户。
  - 当参数enable_kill_query为off，且使用CASCADE模式删除用户时，会等待锁定级联对象的线程结束之后再删除用户。
- 在数据库中删除用户时，如果依赖用户的对象在其他数据库中或者依赖用户的对象是其他数据库，请用户先手动删除其他数据库中的依赖对象或直接删除依赖数据库，再删除用户。即DROP USER不支持跨数据库进行级联删除。
- 在删除用户时，需要先删除该用户拥有的所有对象并且收回该用户在其他对象上的权限，或者通过指定CASCADE级联删除该用户拥有的对象和被授予的权限。

语法格式

```
DROP USER [ IF EXISTS ] user_name [, ...] [ CASCADE | RESTRICT ];
```

参数说明

- **IF EXISTS**

  如果指定的用户不存在，发出一个notice而不是抛出一个错误。
- **user_name**

  待删除的用户名。

  取值范围：已存在的用户名，用户名要求详见**•user_name**。
- **CASCADE | RESTRICT**
  - CASCADE：级联删除依赖用户的对象，并收回授予该用户的权限。
  - RESTRICT：如果用户还有任何依赖的对象或被授予了其他对象的权限，则拒绝删除该用户（缺省行为）。

    说明

    在M-Compatibility中，存在一个配置参数enable_kill_query，此参数在配置文件gaussdb.conf中。此参数影响级联删除用户对象的行为：
    - 当参数enable_kill_query为on ，且使用CASCADE模式删除用户时，会自动kill锁定用户级联对象的线程，并删除用户。
    - 当参数enable_kill_query为off，且使用CASCADE模式删除用户时，会等待锁定级联对象的线程结束之后再删除用户。

示例

请参考CREATE USER的**示例**。

相关链接

ALTER USER，**CREATE USER**

4.4.2.9.19 DROP VIEW

功能描述

数据库中强制删除已有的视图。

注意事项

视图的所有者、视图所在模式的所有者、被授予了视图DROP权限的用户或拥有DROP ANY TABLE权限的用户，有权限执行DROP VIEW的命令，三权分立关闭时的系统管理员默认拥有此权限。

语法格式

```
DROP VIEW [ IF EXISTS ] view_name [, ...] [ CASCADE | RESTRICT ];
```

参数说明

- **IF EXISTS**

  如果指定的视图不存在，则发出一个notice而不是抛出一个错误。

- **view_name**

  要删除的视图名称。

  取值范围：已存在的视图。

- **CASCADE | RESTRICT**
  - CASCADE：表示允许级联删除依赖于该表的对象（比如视图）。
  - RESTRICT：如果有依赖对象存在，则拒绝删除此视图。此选项为缺省值。

    说明

    该属性在M兼容版本控制开关s1及以上版本（如m_format_dev_version = 's1'）时仅语法支持，但实际不生效。

示例

请参见CREATE VIEW的**示例**。

相关链接

ALTER VIEW，**CREATE VIEW**

4.4.2.10 E

4.4.2.10.1 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，所以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

如果创建预备语句时，PREPARE语句声明了一些参数，那么传递给EXECUTE语句必须是一个兼容的参数集，否则会出现错误。

语法格式

```
EXECUTE name;
```

参数说明

- **name**

  要执行的预备语句的名称。

- **parameter**

  给预备语句的参数的变量。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变量。

示例

```
--创建SCHEMA。
m_db=# CREATE SCHEMA tpcds;

--创建表reason。
m_db=# CREATE TABLE tpcds.reason (
   CD_DEMO_SK          INTEGER          NOT NULL,
   CD_GENDER           character(16)            ,
   CD_MARITAL_STATUS   character(100)
);

--插入数据。
m_db=# INSERT INTO tpcds.reason VALUES(51, 'AAAAAAAADDAAAAAA', 'reason 51');

--创建表reason_t1。
m_db=# CREATE TABLE tpcds.reason_t1 LIKE tpcds.reason;

--为一个INSERT语句创建一个预备语句然后执行它。
m_db=# PREPARE insert_reason FROM 'INSERT INTO tpcds.reason_t1
VALUES(52,''AAAAAAAADDAAAAAA'',''reason 52'')';

m_db=# EXECUTE insert_reason;

--删除表reason和reason_t1。
m_db=# DROP TABLE tpcds.reason;
m_db=# DROP TABLE tpcds.reason_t1;

--删除SCHEMA。
m_db=# DROP SCHEMA tpcds;
```

4.4.2.10.2 EXPDP PLUGGABLE DATABASE

功能描述

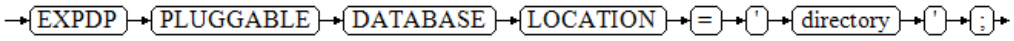
导出PDB的全部物理文件。

说明

EXPDP PLUGGABLE DATABASE语法用于细粒度备份恢复，由备份恢复工具调用，直接调用可能提示目录不存在等报错，不推荐用户直接调用该SQL。

语法格式

```
EXPDP PLUGGABLE DATABASE LOCATION = 'directory';
```



参数说明

directory

导出文件的存储目录。

示例

- 前置条件

  请参见《开发指南》中“SQL参考 > SQL语法 > C > CREATE PLUGGABLE DATABASE”章节中的示例，完成PDB的创建与打开。

- 导出PDB。

```
--创建导出目录
mkdir /data1/expdp/my_pdb

--连入pdb并执行导出操作。
gaussdb=# \c my_pdb;
my_pdb=# EXPDP PLUGGABLE DATABASE LOCATION = '/data1/expdp/my_pdb';
```

4.4.2.10.3 EXPLAIN

功能描述

显示SQL语句的执行计划。

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值是否接近实际值非常有用。

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;
```

- 由于参数DETAIL，NODES，NUM_NODES是分布式模式下的功能，在单机模式中是被禁止使用的。假如使用，会产生如下错误。

```
m_db=# CREATE TABLE student(id int, name char(20));
CREATE TABLE
m_db=# EXPLAIN (NODE true) INSERT INTO student VALUES(5,'a'),(6,'b');
ERROR:  unrecognized EXPLAIN option "nodes"
m_db=# EXPLAIN (NUM_NODES true) INSERT INTO student VALUES(5,'a'),(6,'b');
ERROR:  unrecognized EXPLAIN option "num_nodes"
```


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 [ ( option [, ...] ) ] statement;
```

其中选项option子句的语法为。

```
ANALYZE [ boolean ] |
    VERBOSE [ boolean ] |
    COSTS [ boolean ] |
    CPU [ boolean ] |
    DETAIL [ boolean ] |
    NODES [ boolean ] |(仅分布式模式可用，集中式模式不可用)
    NUM_NODES [ boolean ] |(仅分布式模式可用，集中式模式不可用)
    BUFFERS [ boolean ] |
    TIMING [ boolean ] |
    PLAN [ boolean ] |
    BLOCKNAME [ boolean ] |
    OUTLINE [ boolean ] |
    ADAPTCOST [ boolean ] |
    FORMAT { TEXT | XML | JSON | YAML }
    OPTEVAL [ boolean ]
```

- 显示SQL语句的执行计划，且要按顺序给出选项。

```
EXPLAIN { [ ANALYZE ] [ VERBOSE ] } statement;
```

参数说明

- **statement**

  指定要分析的SQL语句。

- **ANALYZE boolean**

  显示实际运行时间和其他统计数据。当两个参数同时使用时，在option中排在后面的一个生效。

  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。需要结合ANALYZE选项一起使用。

  取值范围：

  - TRUE（缺省值）：显示CPU的使用情况。
  - FALSE：不显示。

- **DETAIL boolean**

  打印数据库节点上的信息。需要结合ANALYZE选项一起使用。

取值范围：

- TRUE（缺省值）：打印数据库节点的信息。
- FALSE：不打印。

- **NODES boolean**（仅分布式模式可用，集中式模式不可用）

  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**（仅分布式模式可用，集中式模式不可用）

  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。需要结合ANALYZE选项一起使用。

  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。需要结合ANALYZE选项一起使用。

  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。

- **BLOCKNAME boolean**

  是否显示计划的每个操作所处于的查询块。当该选项开启时，会将每个操作所处于的查询块的名字输出在Query Block列上，方便用户获取查询块名字，并使用Hint修改执行计划：

  - TRUE（缺省值）：显示计划时，将每个操作所处于的查询块的名字输出在新增列Query Block列上。该选项需要在pretty模式下使用。
  - FALSE：不对计划显示产生影响。

- **OUTLINE boolean**

  是否显示计划的Outline Hint信息。

  - ON：显示计划时，将Outline Hint显示在计划下方 。

  - OFF（缺省值）：不显示计划的Outline Hint信息。
- **ADAPTCOST boolean**

  在Normal模式下是否显示计划的基数估计方式信息。
  - ON（缺省值）：Normal模式下，在计划节点上展示基数估计的方式，包含默认方式和反馈方式，不对预备语句生效。
  - OFF：不展示基数估计的方式信息。
- **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。下述为部分信息描述：
  - ex c/r：代表平均每行使用cpu周期数，等于（ex cyc）/（ex row）。
  - ex row：执行行数。
  - ex cyc：代表使用的cpu周期数。
  - inc cyc：代表包含子节点使用的总cpu周期数。
  - shared hit：代表算子的share buffer命中情况。
  - loops：算子循环执行次数。
  - total_calls：生成元素总数。
  - remote query poll time stream gather：算子用于侦听各DN数据到达CN的网络poll时间。
  - deserialize time：反序列化所需时间。
  - estimated time：估计时间。
- **OPTEVAL boolean**

  是否显示SCAN算子（当前仅支持seqscan、indexscan、indexonlyscan、bitmapheapscan）的代价淘汰明细，当开启此开关的时候，会在执行计划中显示一个名字为Cost Evaluation Info (identified by plan id)的计划块，该选项仅仅可以和COSTS、VERBOSE、FORMAT三个选项共存。此计划块中的具体参数明细，请参考**示例2**。

  取值范围：
  - TRUE：显示SCAN算子的代价淘汰明细。
  - FALSE（缺省值）：不显示。

示例 1

```
--创建SCHEMA。
m_db=# CREATE SCHEMA tpcds;

--创建表tpcds.customer_address。
m_db=# CREATE TABLE tpcds.customer_address
(
ca_address_sk         INTEGER           NOT NULL,
ca_address_id         CHARACTER(16)     NOT NULL
);

--向表中插入多条记录。
m_db=# INSERT INTO tpcds.customer_address VALUES (5000, 'AAAAAAAABAAAAAAA'),(10000,
'AAAAAAAACAAAAAAA');
```



```
--创建一个表tpcds.customer_address_p1。
m_db=# CREATE TABLE tpcds.customer_address_p1 LIKE tpcds.customer_address;

--修改explain_perf_mode为normal。
m_db=# SET explain_perf_mode=normal;

--显示表简单查询的执行计划。
m_db=# EXPLAIN SELECT * FROM tpcds.customer_address_p1;
                          QUERY PLAN
------------------------------------------------------------------------
 Seq Scan on customer_address_p1  (cost=0.00..18.01 rows=801 width=72)
(1 row)

--以JSON格式输出的执行计划（explain_perf_mode为normal时）。
m_db=# EXPLAIN(FORMAT JSON) SELECT * FROM tpcds.customer_address_p1;
               QUERY PLAN
------------------------------------------------
 [                                             +
   {                                           +
     "Plan": {                                   +
       "Node Type": "Seq Scan",                  +
       "Relation Name": "customer_address_p1",+
       "Alias": "customer_address_p1",           +
       "Startup Cost": 0.00,                     +
       "Total Cost": 18.01,                      +
       "Plan Rows": 801,                         +
       "Plan Width": 72                          +
     }                                           +
   }                                             +
 ]
(1 row)

--如果有一个索引，当使用一个带索引WHERE条件的查询，可能会显示一个不同的计划。
m_db=# EXPLAIN SELECT * FROM tpcds.customer_address_p1 WHERE ca_address_sk=10000;
                          QUERY PLAN
------------------------------------------------------------------------
 Seq Scan on customer_address_p1  (cost=0.00..20.01 rows=4 width=72)
   Filter: (ca_address_sk = 10000)
(2 rows)

--以YAML格式输出的执行计划（explain_perf_mode为normal时）。
m_db=# EXPLAIN(FORMAT YAML) SELECT * FROM tpcds.customer_address_p1 WHERE
ca_address_sk=10000;
               QUERY PLAN
-----------------------------------------
 - Plan:                                 +
     Node Type: "Seq Scan"               +
     Relation Name: "customer_address_p1"+
     Alias: "customer_address_p1"        +
     Startup Cost: 0.00                  +
     Total Cost: 20.01                   +
     Plan Rows: 4                        +
     Plan Width: 72                      +
     Filter: "(ca_address_sk = 10000)"
(1 row)

--禁止开销估计的执行计划。
m_db=# EXPLAIN(COSTS FALSE) SELECT * FROM tpcds.customer_address_p1 WHERE ca_address_sk=10000;
           QUERY PLAN
---------------------------------
 Seq Scan on customer_address_p1
   Filter: (ca_address_sk = 10000)
(2 rows)

--带有聚集函数查询的执行计划。
m_db=# EXPLAIN SELECT SUM(ca_address_sk) FROM tpcds.customer_address_p1 WHERE
ca_address_sk<10000;
                             QUERY PLAN
------------------------------------------------------------------------------
```



```
 Aggregate  (cost=20.68..20.69 rows=1 width=12)
   ->  Seq Scan on customer_address_p1  (cost=0.00..20.01 rows=267 width=4)
         Filter: (ca_address_sk < 10000)
(3 rows)

--创建一个二级分区表。
m_db=# CREATE TABLE range_hash ( ID INT, purchased YEAR ) PARTITION BY RANGE ( ID ) SUBPARTITION
BY HASH ( purchased ) (
    PARTITION p0
    VALUES
        LESS THAN ( 1000 ) ( SUBPARTITION s0, SUBPARTITION s1 ),
        PARTITION p1
    VALUES
        LESS THAN ( 2000 ) ( SUBPARTITION s2, SUBPARTITION s3 ),
        PARTITION p2
    VALUES
    LESS THAN MAXVALUE ( SUBPARTITION s4, SUBPARTITION s5 )
);

--执行带有二级分区表的查询语句。
m_db=# EXPLAIN SELECT * FROM range_hash WHERE purchased  = '1';
                                QUERY PLAN
---------------------------------------------------------------------------
 Partition Iterator  (cost=0.00..36.86 rows=11 width=8)
   Iterations: 3, Sub Iterations: 6
   ->  Partitioned Seq Scan on range_hash  (cost=0.00..36.86 rows=11 width=8)
         Filter: (purchased = '2001'::year)
         Selected Partitions:  1..3
         Selected Subpartitions:  ALL
(6 rows)

--删除表tpcds.customer_address_p1。
m_db=# DROP TABLE tpcds.customer_address_p1;

--删除表tpcds.customer_address。
m_db=# DROP TABLE tpcds.customer_address;

--删除表range_list。
m_db=# DROP TABLE range_hash;

--删除SCHEMA。
m_db=# DROP SCHEMA tpcds;
```

示例 2

```
--创建tb_a，tb_b。
m_db=# CREATE TABLE tb_a(c1 int);
m_db=# INSERT INTO tb_a VALUES(1),(2),(3);
m_db=# CREATE TABLE tb_b AS SELECT * FROM tb_a;
m_db=# EXPLAIN (OPTEVAL on )SELECT * FROM tb_a a, tb_b b WHERE a.c1=b.c1 AND a.c1=1;
                            QUERY PLAN
------------------------------------------------------------------
 Nested Loop  (cost=0.00..81.88 rows=144 width=8)
   ->  Seq Scan on tb_a a  (cost=0.00..40.03 rows=12 width=4)
         Filter: (c1 = 1)
   ->  Materialize  (cost=0.00..40.09 rows=12 width=4)
         ->  Seq Scan on tb_b b  (cost=0.00..40.03 rows=12 width=4)
               Filter: (c1 = 1)
(6 rows)

--删除表tb_a，tb_b。
m_db=# DROP TABLE tb_a;
m_db=# DROP TABLE tb_b;
```


说明

针对Cost Evaluation Info (identified by plan id)计划块：

1. “2 --”与“4 --”所在行表示的是当前胜选算子，它们各自下面的缩进计划块表示的是当前胜选算子直接淘汰的算子。如算子2淘汰的算子有Seq Scan算子和Bitmap Heap Scan算子。
2. 如上所示，对上述看到的关键参数进行说明：
    1. id表示当前算子从指定id的path转换而来，该值主要用于在debug2日志中方便定位某一条路径。
    2. rpage表示在代价模型计算过程中使用的基表的页面数。
    3. ipage表示在代价模型计算过程中使用的索引的页面数。
    4. tuples表示在代价模型计算过程中使用的tuple的数量。
    5. selec表示在代价模型计算过程中使用的索引选择率，-1表示当前算子的索引选率无效。
    6. ml表示在代价模型计算过程中，当前触发的ML模型事件，1表示effective_cache_size指定的缓存足够，2表示effective_cache_size指定的缓存不足，3表示effective_cache_size指定的缓存严重不足。
    7. iscost表示在模型代价计算过程中是否发生过忽略启动代价的事件。
    8. lossy表示在代价模型计算过程中是否触发bitmap heap scan的lossy机制。
    9. uidx表示在代价模型计算过程中是否触发唯一索引优先规则。

相关链接

ANALYZE

4.4.2.11 G

4.4.2.11.1 GRANT

功能描述

对角色和用户进行授权操作。

使用GRANT命令进行用户授权包括以下场景：

- **将系统权限授权给角色或用户**

  系统权限又称为用户属性，包括SYSADMIN、CREATEDB、CREATEROLE、AUDITADMIN、MONADMIN、OPRADMIN、POLADMIN、INHERIT、REPLICATION和LOGIN等。

  系统权限一般通过CREATE/ALTER ROLE语法来指定。其中，SYSADMIN权限可以通过GRANT/REVOKE ALL PRIVILEGES授予或撤销。但系统权限无法通过ROLE和USER的权限被继承，也无法授予PUBLIC。

- **将数据库对象授权给角色或用户**

  将数据库对象（表、视图、指定字段、数据库、函数、模式、表空间等）的相关权限授予特定角色或用户。

  GRANT命令将数据库对象的特定权限授予一个或多个角色，这些权限会追加到已有的权限上。

  关键字PUBLIC表示该权限要赋予所有角色，包括以后创建的用户。PUBLIC可以看做是一个隐含定义好的组，它总是包括所有角色。任何角色或用户都将拥有通过GRANT直接赋予的权限和所属的权限，再加上PUBLIC的权限。

如果声明了WITH GRANT OPTION，则被授权的用户也可以将此权限赋予他人，否则就不能授权给他人。这个选项不能赋予PUBLIC，这是M-Compatibility特有的属性。

M-Compatibility会将某些类型的对象上的权限授予PUBLIC。默认情况下，对表、表字段、序列、外部服务器、模式或表空间对象的权限不会授予PUBLIC，而以下这些对象的权限会授予PUBLIC：数据库的CONNECT权限和CREATE TEMP TABLE权限、函数的EXECUTE特权、语言和数据类型（包括域）的USAGE特权。对象拥有者可以撤销默认授予PUBLIC的权限并专门授予权限给其他用户。为了更安全，建议在同一个事务中创建对象并设置权限，这样其他用户就没有时间窗口使用该对象。

对象的所有者缺省具有该对象上的所有权限，出于安全考虑所有者可以舍弃部分权限，但ALTER、DROP、COMMENT、INDEX、VACUUM以及对象的可再授予权限属于所有者固有的权限，隐式拥有。

- **将角色或用户的权限授权给其他角色或用户**

  将一个角色或用户的权限授予一个或多个其他角色或用户。在这种情况下，每个角色或用户都可视为拥有一个或多个数据库权限的集合。

  当声明了WITH ADMIN OPTION，被授权的用户可以将该权限再次授予其他角色或用户，以及撤销所有由该角色或用户继承到的权限。当授权的角色或用户发生变更或被撤销时，所有继承该角色或用户权限的用户拥有的权限都会随之发生变更。

  三权分立关闭时，系统管理员可以赋予或者撤销任何非永久用户、运维管理员和私用用户角色的权限，安全管理员可以赋予或者撤销任何非系统管理员、内置角色、永久用户、运维管理员和私用用户角色的权限。

- **将ANY权限授予给角色或用户**

  将ANY权限授予特定的角色和用户，ANY权限的取值范围参见语法格式。当声明了WITH ADMIN OPTION，被授权的用户可以将该ANY权限再次授予其他角色/用户，或从其他角色/用户处回收该ANY权限。ANY权限可以通过角色被继承，但不能赋予PUBLIC。初始用户和三权分立关闭时的系统管理员用户可以给任何角色/用户授予或撤销ANY权限。

  目前支持以下ANY权限：CREATE ANY TABLE、ALTER ANY TABLE、DROP ANY TABLE、SELECT ANY TABLE、INSERT ANY TABLE、UPDATE ANY TABLE、DELETE ANY TABLE、CREATE ANY SEQUENCE、ALTER ANY SEQUENCE、DROP ANY SEQUENCE、SELECT ANY SEQUENCE、ALTER ANY INDEX、DROP ANY INDEX。详细的ANY权限范围描述参考**表4-24**

注意事项

- 不允许将ANY权限授予PUBLIC，也不允许从PUBLIC回收ANY权限。
- ANY权限属于数据库内的权限，只对授予该权限的数据库内的对象有效，例如SELECT ANY TABLE只允许用户查看当前数据库内的所有用户表数据，对其他数据库内的用户表无查看权限。
- ANY权限与原有的权限相互无影响。
- 如果用户被授予了CREATE ANY TABLE权限，在同名模式下创建表的属主是该模式的所有者，用户对表进行其他操作时，需要授予相应的操作权限。与此类似的还有CREATE ANY SEQUENCE和CREATE ANY INDEX，在同名模式下创建的对象的所有者是同名模式的所有者。
- 通过GRANT授予用户使用表的权限时，如果用户使用不当，可能会通过ALTER语法在表的默认值、约束增加表达式、通过创建索引在索引上增加表达式等操作导致权限被利用的风险。

语法格式

- 将表或视图的访问权限赋予指定的用户或角色。

```
GRANT { { SELECT | INSERT | UPDATE | DELETE | TRUNCATE | REFERENCES | ALTER | DROP |
COMMENT | INDEX | VACUUM } [, ...]
    | ALL [ PRIVILEGES ] }
  ON { [ TABLE ] table_name [, ...]
    | ALL TABLES IN SCHEMA schema_name [, ...] }
  TO { [ GROUP ] role_name | PUBLIC } [, ...]
  [ WITH GRANT OPTION ];
```

说明

用户如果想要对某张表进行某种操作，除了拥有对该表的操作权限外，还需要拥有表所在模式的USAGE权限。除此之外，如果想要在表上创建索引，则必须拥有模式的CREATE权限和表上的INDEX权限。

- 将表中字段的访问权限赋予指定的用户或角色。

```
GRANT { {{ SELECT | INSERT | UPDATE | REFERENCES | COMMENT } ( column_name [, ...] )} [, ...]
    | ALL [ PRIVILEGES ] ( column_name [, ...] ) }
  ON [ TABLE ] table_name [, ...]
  TO { [ GROUP ] role_name | PUBLIC } [, ...]
  [ WITH GRANT OPTION ];
```

说明

如果拥有表的访问权限，则默认拥有表中所有字段的访问权限。如果要仅赋予表中某个字段的访问权限，需要先撤销所属表的访问权限。

- 将序列的访问权限赋予指定的用户或角色，LARGE字段属性可选，赋权语句不区分序列是否为LARGE。

```
GRANT { { SELECT | UPDATE | USAGE | ALTER | DROP | COMMENT } [, ...]
    | ALL [ PRIVILEGES ] }
  ON { [ [ LARGE ] SEQUENCE ] sequence_name [, ...] }
  TO { [ GROUP ] role_name | PUBLIC } [, ...]
  [ WITH GRANT OPTION ];
```

- 将数据库的访问权限赋予指定的用户或角色。

```
GRANT { { CREATE | CONNECT | TEMPORARY | TEMP | ALTER | DROP | COMMENT } [, ...]
    | ALL [ PRIVILEGES ] }
  ON DATABASE database_name [, ...]
  TO { [ GROUP ] role_name | PUBLIC } [, ...]
  [ WITH GRANT OPTION ];
```

说明

GUC参数m_format_behavior_compat_options开启grant_database_nomapping选项时该语法生效，将数据库的访问权限赋予指定的用户或角色；未开启grant_database_nomapping选项时，GRANT ON DATABASE映射为GRANT ON SCHEMA，将模式的访问权限赋予指定的用户或角色，GRANT ON DATABASE也仅支持GRANT ON SCHEMA支持的语法选项。

- 将模式的访问权限赋予指定的用户或角色。

```
GRANT { { CREATE | USAGE | ALTER | DROP | COMMENT } [, ...] | ALL [ PRIVILEGES ] }
  ON SCHEMA schema_name [, ...]
  TO { [ GROUP ] role_name | PUBLIC } [, ...]
  [ WITH GRANT OPTION ];
```

说明

将模式中的表或者视图对象授权给其他用户时，需要将表或视图所属的模式的USAGE权限同时授予该用户，若没有该权限，则只能看到这些对象的名称，并不能实际进行对象访问。 同名模式下创建表的权限无法通过此语法赋予，可以通过将角色的权限赋予其他用户或角色的语法，赋予同名模式下创建表的权限。

- 将角色的权限赋予其他用户或角色的语法。


```
GRANT role_name [, ...]
  TO role_name [, ...]
  [ WITH ADMIN OPTION ];
```

- 将SYSADMIN权限赋予指定的角色。

```
GRANT ALL PRIVILEGES
  TO role_name;
```

- 将ANY权限赋予其他用户或角色的语法。

```
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 |
ALTER ANY SEQUENCE | DROP ANY SEQUENCE |
  SELECT ANY SEQUENCE | ALTER ANY INDEX | DROP ANY INDEX |  } [, ...]
  TO [ GROUP ] role_name [, ...]
  [ WITH ADMIN OPTION ];
```

参数说明

GRANT的权限分类如下所示。

- **SELECT**

  允许对指定的表、视图、序列执行SELECT命令，update或delete时也需要对应字段上的select权限。

- **INSERT**

  允许对指定的表执行INSERT命令。

- **UPDATE**

  允许对声明的表中任意字段执行UPDATE命令。通常，update命令也需要select权限来查询出哪些行需要更新。SELECT… FOR UPDATE和SELECT… FOR SHARE除了需要SELECT权限外，还需要UPDATE权限。

- **DELETE**

  允许执行DELETE命令删除指定表中的数据。通常，delete命令也需要select权限来查询出哪些行需要删除。

- **TRUNCATE**

  允许执行TRUNCATE语句删除指定表中的所有记录。

- **REFERENCES**

  创建一个外键约束，必须拥有参考表和被参考表的REFERENCES权限。

- **CREATE**
  - 对于模式，允许在模式中创建新的对象。如果要重命名一个对象，用户除了必须是该对象的所有者外，还必须拥有该对象所在模式的CREATE权限。
  - 对于表空间，允许在表空间中创建表，允许在创建数据库和模式的时候把该表空间指定为缺省表空间。

- **EXECUTE**

  允许使用指定的函数，以及利用这些函数实现的操作符。

- **USAGE**
  - 对于过程语言，允许用户在创建函数的时候指定过程语言。
  - 对于模式，USAGE允许访问包含在指定模式中的对象，若没有该权限，则只能看到这些对象的名称。
  - 对于序列，USAGE允许使用nextval函数。

- **ALTER**

允许用户修改指定对象的属性，但不包括修改对象的所有者和修改对象所在的模式。

- **DROP**
  允许用户删除指定的对象。
- **COMMENT**
  允许用户定义或修改指定对象的注释。
- **INDEX**
  允许用户在指定表上创建索引，并管理指定表上的索引，还允许用户对指定表执行REINDEX操作。
- **VACUUM**
  允许用户对指定的表执行ANALYZE和VACUUM操作。
- **ALL PRIVILEGES**
  一次性给指定用户/角色赋予所有可赋予的权限。只有系统管理员有权执行GRANT ALL PRIVILEGES。

GRANT的参数说明如下所示。

- **role_name**
  已存在用户名称，用户名要求详见**•user_name**。
- **table_name**
  已存在表名称。
- **column_name**
  已存在字段名称。
- **schema_name**
  已存在模式名称。
- **sequence_name**
  已存在序列名称。
- **type_name**
  已存在类型名称。
- **WITH GRANT OPTION**
  如果声明了WITH GRANT OPTION，则被授权的用户也可以将此权限赋予他人，否则就不能授权给他人。这个选项不能赋予PUBLIC。

非对象所有者给其他用户授予对象权限时，命令按照以下规则执行：

- 如果用户没有该对象上指定的权限，命令立即失败。
- 如果用户有该对象上的部分权限，则GRANT命令只授予他有授权选项的权限。
- 如果用户没有可用的授权选项，GRANT ALL PRIVILEGES形式将发出一个警告信息，其他命令形式将发出在命令中提到的且没有授权选项的相关警告信息。

说明

三权分立关闭时的数据库系统管理员可以访问所有对象，而不会受对象的权限设置影响。这个特点类似Unix系统的root的权限。和root一样，除了必要的情况外，建议不要总是以系统管理员身份进行操作。

- **WITH ADMIN OPTION**

对于角色，当声明了WITH ADMIN OPTION，被授权的用户可以将该角色再授予其他角色/用户，或从其他角色/用户回收该角色。

对于ANY权限，当声明了WITH ADMIN OPTION，被授权的用户可以将该ANY权限再授予其他角色/用户，或从其他角色/用户回收该ANY权限。

表 4-24 ANY 权限列表

系统权限名称	描述
CREATE ANY TABLE	用户能够在public模式和用户模式下创建表或视图。
ALTER ANY TABLE	用户拥有对public模式和用户模式下表或视图的ALTER权限。如果想要修改表的唯一索引为表增加主键约束或唯一约束，还需要授予该表的索引权限。
DROP ANY TABLE	用户拥有对public模式和用户模式下表或视图的DROP权限。
SELECT ANY TABLE	用户拥有对public模式和用户模式下表或视图的SELECT权限。
UPDATE ANY TABLE	用户拥有对public模式和用户模式下表或视图的UPDATE权限。
INSERT ANY TABLE	用户拥有对public模式和用户模式下表或视图的INSERT权限。
DELETE ANY TABLE	用户拥有对public模式和用户模式下表或视图的DELETE权限。
CREATE ANY SEQUENCE	用户能够在public模式和用户模式下创建序列。
CREATE ANY INDEX	用户能够在public模式和用户模式下创建索引。如果在某表空间创建分区表索引，需要授予用户该表空间的创建权限。
ALTER ANY SEQUENCE	用户拥有对public模式和用户模式下序列的ALTER权限，但不包括修改序列的所有者。
DROP ANY SEQUENCE	用户拥有对public模式和用户模式下序列的DROP权限。
SELECT ANY SEQUENCE	用户拥有对public模式和用户模式下序列的SELECT、USAGE和UPDATE权限。
ALTER ANY INDEX	用户拥有对public模式和用户模式下索引的ALTER权限。如果要重命名索引，还需要索引所在模式下创建对象的权限。如果涉及表空间的操作，还需要对应表空间的相应操作权限。如果设置索引不可用（UNUSABLE），还需要DROP ANY INDEX权限。
DROP ANY INDEX	用户拥有对public模式和用户模式下索引的DROP权限。

说明

用户被授予任何一种ANY权限后，用户对public模式和用户模式具有USAGE权限，对GaussDB支持的Schema中除public之外的系统模式没有USAGE权限，GaussDB支持的Schema详见《开发指南》的“Schema”章节。

示例

示例1：将系统权限授权给用户或者角色。

创建名为joe的用户，并将sysadmin权限授权给他。

```
m_db=# CREATE USER joe PASSWORD '********';
m_db=# GRANT ALL PRIVILEGES TO joe;
```

授权成功后，用户joe会拥有sysadmin的所有权限。

示例2：将对象权限授权给用户或者角色。

1. 撤销joe用户的sysadmin权限，然后将模式tpcds的使用权限和表tpcds.reason的所有权限授权给用户joe。

   ```
   m_db=# CREATE SCHEMA tpcds;
   CREATE SCHEMA
   m_db=# CREATE TABLE tpcds.reason
   (
   r_reason_sk         INTEGER      NOT NULL,
   r_reason_id         CHAR(16)     NOT NULL,
   r_reason_desc       VARCHAR(20)
   );
   CREATE TABLE
   m_db=# REVOKE ALL PRIVILEGES FROM joe;
   m_db=# GRANT USAGE ON SCHEMA tpcds TO joe;
   m_db=# GRANT ALL PRIVILEGES ON tpcds.reason TO joe;
   ```

   授权成功后，joe用户就拥有了tpcds.reason表的所有权限，包括增删改查等权限。

2. 将tpcds.reason表中r_reason_sk、r_reason_id、r_reason_desc列的查询权限，r_reason_desc的更新权限授权给joe。

   ```
   m_db=# GRANT SELECT (r_reason_sk,r_reason_id,r_reason_desc),UPDATE (r_reason_desc) ON tpcds.reason TO joe;
   ```

   授权成功后，用户joe对tpcds.reason表中r_reason_sk，r_reason_id, r_reason_desc的查询权限会立即生效。如果joe用户需要拥有将这些权限授权给其他用户的权限，可以通过以下语法对joe用户进行授权。

   ```
   m_db=# GRANT SELECT (r_reason_sk, r_reason_id) ON tpcds.reason TO joe WITH GRANT OPTION;
   ```

   创建角色tpcds_manager，将模式tpcds的访问权限授权给角色tpcds_manager，并授予该角色在tpcds下创建对象的权限，不允许该角色中的用户将权限授权给其他人。

   ```
   m_db=# CREATE ROLE tpcds_manager PASSWORD '********';
   m_db=# GRANT USAGE,CREATE ON SCHEMA tpcds TO tpcds_manager;
   ```

示例3：将用户或者角色的权限授权给其他用户或角色。

1. 创建角色manager，将joe的权限授权给manager，并允许该角色将权限授权给其他人。

   ```
   m_db=# CREATE ROLE manager PASSWORD '********';
   m_db=# GRANT joe TO manager WITH ADMIN OPTION;
   ```

2. 创建用户senior_manager，将用户manager的权限授权给该用户。

   ```
   m_db=# CREATE ROLE senior_manager PASSWORD '********';
   m_db=# GRANT manager TO senior_manager;
   ```

3. 撤销权限，并清理用户。


```
m_db=# REVOKE joe FROM manager;
m_db=# REVOKE manager FROM senior_manager;
m_db=# DROP USER manager;
```

示例4：撤销上述授予的权限，并清理角色和用户。

```
m_db=# REVOKE ALL PRIVILEGES ON SCHEMA tpcds FROM joe;
m_db=# REVOKE USAGE,CREATE ON SCHEMA tpcds FROM tpcds_manager;
m_db=# DROP ROLE tpcds_manager;
m_db=# DROP ROLE senior_manager;
m_db=# DROP USER joe CASCADE;
m_db=# DROP TABLE tpcds.reason;
m_db=# DROP SCHEMA tpcds;
```

相关链接

REVOKE

4.4.2.12 I

4.4.2.12.1 IMPDP PLUGGABLE DATABASE RECOVER

功能描述

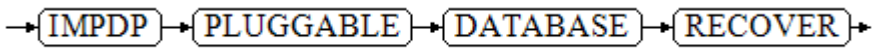
导入PDB的修复阶段。

说明

IMPDP PLUGGABLE DATABASE RECOVER语法用于细粒度备份恢复，由备份恢复工具调用，不推荐用户直接调用该SQL。

语法格式

```
IMPDP PLUGGABLE DATABASE RECOVER;
```



示例

- 前置条件

  请参见《开发指南》中“SQL参考 > SQL语法 > I > IMPDP PLUGGABLE DATABASE CREATE”章节中的示例，完成PDB的导入执行阶段。

- 导入恢复PDB

```
--使用系统管理员连接postgres数据库，为导入执行阶段创建出的PDB分配资源计划指令。
gaussdb=# call resource_manager.create_pending_area();
 create_pending_area
---------------------


(1 row)
--由于该接口是创建资源，是否成功与当前实例资源规格有关。如遇到接口报错，请根据错误描述进行参数调整参数后重新尝试。
gaussdb=# CALL resource_manager.create_resource_plan_directive(pdb_name =>'my_pdb1', min_cpu
=> 4, max_dynamic_memory => '18022MB', max_shared_memory => '14745MB', io_limits=> 300,
io_priority => 'High', max_connections => 880);
 create_resource_plan_directive
--------------------------------


(1 row)
gaussdb=# CALL resource_manager.submit_pending_area();
```



```
 submit_pending_area
---------------------

(1 row)

--使用系统管理员连接postgres数据库，开启导入执行阶段创建出的PDB。
gaussdb=# alter pluggable database my_pdb1 open;
ALTER PLUGGABLE DATABASE

--连入pdb并执行导入恢复操作。
gaussdb=# \c my_pdb1;
my_pdb=# IMPDP PLUGGABLE DATABASE RECOVER;
IMPDP PLUGGABLE DATABASE RECOVER
```

4.4.2.12.2 INSERT

功能描述

向表中插入一行或多行数据。

注意事项

- 只有拥有表INSERT权限的用户，才可以向表中插入数据。用户被授予insert any table权限，相当于用户对除系统模式之外的任何模式具有USAGE权限，并且拥有这些模式下表的INSERT权限。
- 如果使用ON DUPLICATE KEY UPDATE，用户必须要有该表的INSERT、UPDATE权限，UPDATE子句中列的SELECT权限。
- 如果使用query子句插入来自查询里的数据行，用户还需要拥有在查询里使用的表的SELECT权限。
- 生成列不能被直接写入。在INSERT命令中不能为生成列指定值，但是可以指定关键字DEFAULT。

语法格式

```
INSERT [/*+ plan_hint */] [IGNORE]
[INTO] tbl_name
[PARTITION (partition_name [, partition_name] ...)]
   [(col_name [, col_name] ...)]
   {VALUES | VALUE} ([value_list]) [, (value_list)] ...
   [ON DUPLICATE KEY UPDATE column_name = {expression | DEFAULT}[, ...]];

INSERT [/*+ plan_hint */] [IGNORE] [INTO] tbl_name
   [PARTITION (partition_name [, partition_name] ...)]
   SET column_name = {expression | DEFAULT}[, ...]
   [ON DUPLICATE KEY UPDATE column_name = {expression | DEFAULT}[, ...]];

INSERT [/*+ plan_hint */] [IGNORE] [INTO] tbl_name
   [PARTITION (partition_name [, partition_name] ...)]
   [(col_name [, col_name] ...)]
   query
   [ON DUPLICATE KEY UPDATE column_name = {expression | DEFAULT}[, ...]];

INSERT [/*+ plan_hint */] [IGNORE] [INTO] view_name
   [(col_name [, col_name] ...)]
   {VALUES | VALUE} ([value_list]) [, (value_list), ...];

INSERT [/*+ plan_hint */] [IGNORE] [INTO] view_name
   SET column_name = {expression | DEFAULT}[, ...];

INSERT [/*+ plan_hint */] [IGNORE] [INTO] view_name
   [(col_name [, col_name] ...)]
   query;
```


参数说明

- **plan_hint子句**

  以/*+ */的形式在INSERT关键字后，用于对INSERT对应的语句块生成的计划进行hint调优。每条语句中只有第一个/*+ plan_hint */注释块会作为hint生效，里面可以写多条hint。

- **IGNORE**

  INSERT语句使用IGNORE关键字时，可将部分ERROR级别的错误降级为WARNING级别，并根据不同的错误场景将无效值调整为最接近的值。GaussDB支持错误降级的场景如下：

  - 破坏NOT NULL约束。
  - 唯一键冲突。
  - 插入的值无法匹配对应的分区。
  - 指定分区插入时，插入的数据与指定分区不匹配。
  - 子查询返回多行。
  - sql_mode为宽松模式的场景。

- **table_name**

  要插入数据的目标表名。

  取值范围：已存在的表名。

- **partition_name**

  分区名。

- **column_name**

  目标表中的字段名称：

  - 字段名可以有子字段名或者数组下标修饰。
  - 没有在字段列表中出现的每个字段，将由系统默认值，或者声明时的默认值填充，若都没有则用NULL填充。例如，向一个复合类型中的某些字段插入数据的话，其他字段将是NULL。
  - 目标字段（column_name）可以按顺序排列。如果没有列出任何字段，则默认全部字段，且顺序为表声明时的顺序。
  - 如果value子句和query中只提供了N个字段，则目标字段为前N个字段。
  - value子句和query提供的值在表中从左到右关联到对应列。

  取值范围：已存在的字段名。

- **expression**

  赋予对应column的一个有效表达式或值：

  - 如果是在INSERT ON DUPLICATE KEY UPDATE语句下，expression可以为VALUES(column_name)，用来表示引用冲突行对应的column_name字段的值。其中VALUES(column_name)支持嵌套在复合表达式中，例如：VALUES(column_name) +1、VALUES(column_name) + VALUES(column_name)、function_name(VALUES(column_name))等。

    说明

    VALUES(column_name)特性仅支持在ON DUPLICATE KEY UPDATE子句中使用。

  - 向表中字段插入单引号 " ' "时需要使用单引号自身进行转义。
  - 如果插入行的表达式不是正确的数据类型，系统试图进行类型转换，若转换不成功，则插入数据失败，系统返回错误信息。

- **DEFAULT**

  对应字段名的缺省值。如果没有缺省值，则为NULL。

- **query**

  一个查询语句（SELECT语句），将查询结果作为插入的数据。

- **ON DUPLICATE KEY UPDATE**

  对于带有唯一约束（UNIQUE INDEX或PRIMARY KEY）的表，如果插入数据违反唯一约束，则对冲突行执行UPDATE子句完成更新，对于不带唯一约束的表，则仅执行插入。UPDATE时，可通过"VALUES()" 来选择源数据相应的列。

  - 如果表中存在多个唯一约束，如果所插入数据违反多个唯一约束，对于检测到冲突的第一行进行更新，其他冲突行不更新（检查顺序与索引维护具有强相关性，一般先创建的索引先进行冲突检查）。
  - 如果插入多行，这些行均与表中同一行数据存在唯一约束冲突，则按照顺序，第一条执行插入或更新，之后依次执行更新。
  - expression不支持使用子查询表达式。
  - 设置GUC兼容性参数m_format_dev_version为's2'后，UPDATE子句更新多列时，如果有引用前边的字段名，则使用其更新后的数据，且支持同一字段名可被修改多次。
  - 如果query为子查询，SQL语句满足以下条件时，UPDATE子句支持引用子查询中的列名：

    - 子查询为UNION查询时，不支持引用子查询中的列。
    - 第一层子查询中包含聚集函数、GROUP BY聚集操作时，不支持引用子查询中的列。
    - 仅支持引用子查询中第一个FROM返回的所有字段，不支持引用子查询投影列或多层嵌套子查询的列。
    - 子查询FROM子句的字段存在列别名时，UPDATE子句只能引用列别名。
    - 子查询FROM子句的表存在表别名时，UPDATE子句只能引用表别名，原始表名无效。
    - 支持子查询投影列为函数、表达式场景下引用子查询列。
    - UPDATE子句支持列名、表名.列名以及库名.表名.列名格式。

- **view_name**

  要插入的目标视图。

说明

对视图的插入，有如下约束：

1. 只有直接引用基表用户列的列可插入。
2. 视图必须至少包含一个可更新列，关于可更新列请参见**CREATE VIEW**。
3. 不支持在顶层包含DISTINCT、GROUP BY、HAVING、LIMIT、OFFSET子句的视图。
4. 不支持在顶层包含集合运算（UNION以及EXCEPT）的视图。
5. 不支持目标列表中包含聚集函数、窗口函数、返回集合函数（array_agg、json_agg、generate_series等）的视图。
6. 不支持ON DUPLICATE KEY UPDATE功能。
7. 视图中支持的表类型包括普通表、临时表、全局临时表、分区表、二级分区表、ustore表、astore表。
8. 对多表连接视图，一次只能对一张基表做插入。
9. 对连接视图插入时显式指定的插入列或隐式指定（即创建视图中指定的列）不能引用非保留键表的列。关于保留键表请参见**CREATE VIEW**。
10. 不支持对系统视图插入。

示例

- **插入一条数据**

  示例：

```
--建表。
m_db=# CREATE TABLE test_t1(col1 INT,col2 VARCHAR(20));

--插入数据。
m_db=# INSERT INTO test_t1 (col1, col2) VALUES (1,'AB');

--只给表中部分列插入数据。
m_db=# INSERT INTO test_t1 (col1) VALUES (2);

--VALUES关键字左边没有括号，右边括号里面必须严格按照表结构的顺序给所有的字段添加值。
m_db=# INSERT INTO test_t1 VALUES (3,'AC');

--查询表。
m_db=# SELECT * FROM test_t1;
 col1 | col2
------+------
    1 | AB
    2 |
    3 | AC
(3 rows)

--删除表。
m_db=# DROP TABLE test_t1;
```

- **插入多条数据**

  示例：

```
--建表。
m_db=# CREATE TABLE test_t2(col1 INT,col2 VARCHAR(20));
m_db=# CREATE TABLE test_t3(col1 INT,col2 VARCHAR(20));

--插入多条数据。
m_db=# INSERT INTO test_t2 (col1, col2) VALUES (10,'AA'),(20,'BB'),(30,'CC');

--查询表。
m_db=# SELECT * FROM test_t2;
 col1 | col2
------+------
   10 | AA
   20 | BB
   30 | CC
```



```
(3 rows)

--把test_t2中的数据插入到test_t3中。
m_db=# INSERT INTO test_t3 SELECT * FROM test_t2;

--查询表。
m_db=# SELECT * FROM test_t3;
 col1 | col2
------+------
   10 | AA
   20 | BB
   30 | CC
(3 rows)

--删除表。
m_db=# DROP TABLE test_t2;
m_db=# DROP TABLE test_t3;
```

- **ON DUPLICATE KEY UPDATE**

  示例：

```
-- 建表。
m_db=# CREATE TABLE test_t4 (id INT PRIMARY KEY, info VARCHAR(10));
m_db=# INSERT INTO test_t4 VALUES (1, 'AA'), (2,'BB'), (3, 'CC');

-- 使用ON DUPLICATE KEY UPDATE关键字。
m_db=# INSERT INTO test_t4 VALUES (3, 'DD'), (4, 'EE') ON DUPLICATE KEY UPDATE info =
VALUES(info);

-- 查询表。
m_db=# SELECT * FROM test_t4;
 id | info
----+------
  1 | AA
  2 | BB
  4 | EE
  3 | DD

-- 删除表。
m_db=# DROP TABLE test_t4;

-- 如果query为子查询，ON DUPLICATE KEY UPDATE子句支持使用子查询中的列名。
m_db=# CREATE TABLE t1(a INT PRIMARY KEY, b INT);
CREATE TABLE t2(c INT, d INT);

m_db=# INSERT INTO t1 VALUES(2, 3), (5, 6);
INSERT INTO t2 VALUES(2, 4), (5, 7);

m_db=# INSERT INTO t1 (a, b) SELECT c, d FROM t2 ON DUPLICATE KEY UPDATE b = b + d;
SELECT * FROM t1;
 a | b
---+----
 2 |  7
 5 | 13
(2 rows)

m_db=# DROP TABLE t1;
DROP TABLE t2;
```

- **INSERT IGNORE**

  示例1：破坏NOT NULL约束

```
--建表。
m_db=# CREATE TABLE test_t5(f1 INT NOT NULL);
CREATE TABLE

--使用IGNORE关键字。
m_db=# INSERT IGNORE INTO test_t5 VALUES(NULL);
WARNING:  null value in column "f1" violates not-null constraint
DETAIL:  Failing row contains (null).
```



```
INSERT 0 1

--查询表。
m_db=# SELECT * FROM test_t5;
 f1
----
  0
(1 row)

--删除表。
m_db=# DROP TABLE test_t5;
```

示例2：唯一键冲突

```
--建表。
m_db=# CREATE TABLE test_t6(f1 INT PRIMARY KEY);
NOTICE:  CREATE TABLE / PRIMARY KEY will create implicit index "test_t6_pkey" for table "test_t6"
CREATE TABLE

--插入数据。
m_db=# INSERT INTO test_t6 VALUES(1);
INSERT 0 1

--使用IGNORE关键字。
m_db=# INSERT IGNORE INTO test_t6 VALUES(1);
WARNING:  duplicate key value violates unique constraint "test_t6_pkey"
INSERT 0 0

--查询表。
m_db=# SELECT * FROM test_t6;
 f1
----
  1
(1 row)

--删除表。
m_db=# DROP TABLE test_t6;
DROP TABLE
```

示例3：插入的值没有找到对应的分区

```
--建表。
m_db=# CREATE TABLE test_t7(f1 INT, f2 INT) PARTITION BY LIST(f1) (PARTITION p0 VALUES(1, 4,
7), PARTITION p1 VALUES (2, 5, 8));
CREATE TABLE

--使用IGNORE关键字。
m_db=# INSERT IGNORE INTO test_t7 VALUES(3, 5);
WARNING:  inserted partition key does not map to any table partition
INSERT 0 0

--查询表。
m_db=# SELECT * FROM test_t7;
 f1 | f2
----+----
(0 rows)

--删除表。
m_db=# DROP TABLE test_t7;
DROP TABLE
```

示例4：指定分区插入时，插入的数据与指定的分区不匹配

```
--建表。
m_db=# CREATE TABLE test_t8(f1 INT NOT NULL, f2 TEXT, f3 INT) PARTITION BY RANGE(f1)
(PARTITION p0 VALUES LESS THAN(5), PARTITION p1 VALUES LESS THAN(10), PARTITION p2
VALUES LESS THAN(15), PARTITION p3 VALUES LESS THAN(MAXVALUE));
CREATE TABLE

--使用IGNORE关键字。
m_db=# INSERT IGNORE INTO test_t8 PARTITION(p2) VALUES(20, 'Jan', 1);
```



```
WARNING:  inserted partition key does not map to the table partition
DETAIL:  N/A.
INSERT 0 0

--查询表。
m_db=# SELECT * FROM test_t8;
 f1 | f2 | f3
----+----+----
(0 rows)

--删除表。
m_db=# DROP TABLE test_t8;
DROP TABLE
```

示例5：子查询返回多行

```
--建表。
m_db=# CREATE TABLE test_t9(f1 INT, f2 INT);
CREATE TABLE

--插入数据。
m_db=# INSERT INTO test_t9 VALUES(1, 1), (2, 2), (3, 3);
INSERT 0 3

--使用IGNORE关键字。
m_db=# INSERT IGNORE INTO test_t9 VALUES((SELECT f1 FROM test_t9), 0);
WARNING:  more than one row returned by a subquery used as an expression
CONTEXT:  referenced column: f1
INSERT 0 1

--查询表。
m_db=# SELECT * FROM test_t9 WHERE f2 = 0;
 f1 | f2
----+----
    |  0
(1 row)

--删除表。
m_db=# DROP TABLE test_t9;
DROP TABLE
```

- **插入视图**

  示例：

```
--创建SCHEMA。
m_db=# CREATE SCHEMA ins_view;
CREATE SCHEMA
m_db=# SET CURRENT_SCHEMA = 'ins_view';
SET

--创建表。
m_db=# CREATE TABLE t1 (x1 int, y1 int);
CREATE TABLE
m_db=# CREATE TABLE t2 (x2 int PRIMARY KEY, y2 int);
NOTICE:  CREATE TABLE / PRIMARY KEY will create implicit index "t2_pkey" for table "t2"
CREATE TABLE

--创建单表视图。
m_db=# CREATE VIEW v_ins1 AS SELECT * FROM t1;
CREATE VIEW
m_db=# CREATE VIEW v_ins2 AS SELECT * FROM t1 WHERE y1 < 3;
CREATE VIEW

--通过视图对t1插入。
m_db=# INSERT INTO v_ins1 VALUES (1, 1);
INSERT 0 1
m_db=# INSERT INTO v_ins2 VALUES (5, 5);
INSERT 0 1

--创建多表视图。
m_db=# CREATE VIEW vv_ins AS SELECT * FROM t1, t2 WHERE x1 = x2;
```



```
CREATE VIEW

--通过多表视图对t1插入。
m_db=# INSERT INTO vv_ins (x1, y1) VALUES (2, 2);
INSERT 0 1

--删除SCHEMA。
m_db=# DROP SCHEMA ins_view;
NOTICE:  drop cascades to 5 other objects
DETAIL:  drop cascades to table t1
drop cascades to table t2
drop cascades to view v_ins1
drop cascades to view v_ins2
drop cascades to view vv_ins
DROP SCHEMA
```

优化建议

VALUES

通过insert语句批量插入数据时，建议将多条记录合并入一条语句中执行插入，以提高数据加载性能。

例如：INSERT INTO sections VALUES (30, 'Administration', 31, 1900),(40, 'Development', 35, 2000), (50, 'Development' , 60 , 2001);

4.4.2.13 L

4.4.2.13.1 LOAD DATA

功能描述

将文件中的数据导入到数据库指定表中。

注意事项

- LOAD DATA语法需要具有表的INSERT和DELETE权限。
- 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打开时，允许具有SYSADMIN权限的用户或继承内置角色gs_role_copy_files权限的用户执行LOAD DATA命令，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LOAD DATA命令，防止用户越权查看或修改敏感文件。
- 当参数enable_copy_server_files打开时，管理员可以通过GUC参数safe_data_path设置普通用户可以导入/导出的路径，但设置的路径必须为设置的safe_data_path的子路径；未设置GUC参数safe_data_path时（默认情况），不对普通用户使用的路径进行拦截。
- 执行LOAD DATA语法写入表中的数据若无法转换为表中数据类型格式时，LOAD DATA导入数据将执行失败。
- LOAD DATA只能用于表，不能用于视图。
- LOAD DATA在与COPY TO导出配合使用时，针对bit类型可能存在结果准确性的问题。为保证bit类型结果在导出操作后再进行导入操作时与源数据保持一致，LOAD DATA需要与SELECT INTO OUTFILE语法配合使用。

语法格式

```
LOAD DATA
   [LOCAL]
```



```
INFILE 'file_name'
[REPLACE | IGNORE]
INTO TABLE tbl_name
[PARTITION (partition_name [, partition_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] ...)]
[SET col_name={expr | DEFAULT}
    [, col_name={expr | DEFAULT}] ...]
```

参数说明

- **LOCAL**

  指定导入文件的位置。

  不指定LOCAL时，则从服务端所在环境中导入数据，若'file_name'为相对路径，则默认导入路径为数据目录。

  若指定LOCAL参数，且GUC参数m_format_behavior_compat_options中包含值enable_load_data_remote_transmission，则从客户端所在环境导入数据；若GUC参数m_format_behavior_compat_options中不包含值enable_load_data_remote_transmission，则从服务端所在环境导入数据，当指定为相对路径时默认导入路径为数据库二进制所在路径，即$GAUSSHOME/bin/。

  说明

  - 当导入数据与表中数据冲突或文件中字段数小于指定表中字段数时，指定LOCAL与指定IGNORE作用一致。
  - 当指定m_format_behavior_compat_options中包含值enable_load_data_remote_transmission时，LOAD DATA 命令后不支持和其他SQL一起执行。
  - 若导入文件位置为客户端，则需将GaussDB服务端与客户端（JDBC驱动、ODBC驱动）更新至505.2.1版本。

- **REPLACE | IGNORE**

  当导入数据与表中原有数据冲突时，若指定REPLACE，则替换冲突行数据；若指定IGNORE则跳过冲突行数据，继续导入。若数据冲突但不指定REPLACE，IGNORE或LOCAL中任意一个则终止导入并报错。

  说明

  若文件字段数小于指定表列数，指定LOCAL或IGNORE参数会为剩余列赋默认值。不指定IGNORE或LOCAL参数会报错。

- **PARTITION**

  当导入表为分区表时，此参数用来指定分区。若数据与指定分区范围不一致则报错。

- **CHARACTER SET**

  指定数据文件的编码格式名称，缺省为当前客户端编码格式。

- **FIELDS | COLUMNS**
  - **TERMINATED BY**
    指定两列之间分隔符，缺省为'\t'。

    说明

    指定换行符不能与分隔符相同。
  - **[OPTIONALLY] ENCLOSED BY**
    指定引号字符，缺省为''。
    OPTIONALLY参数为可选参数，无实际作用。
    引号符仅支持单字符，不支持字符串。
  - **ESCAPED BY**
    指定转义符，缺省为'\'。
    转义字符仅支持单字符，不支持字符串。
- **LINES**
  - **STARTING BY**
    指定导入数据文件起始字段样式。
  - **TERMINATED BY**
    指定导入数据文件换行符样式。
- **IGNORE**
  指定数据导入时，跳过数据文件的前 number行。
- **col_name_or_user_var**
  可选的待复制字段列表。
  取值范围：如果没有声明字段列表，将使用所有字段。

  说明

  - 指定列参数不支持重复指定列。
  - LOAD DATA语法指定列时，col_name_or_user_var支持指定为表中存在列或用户变量。
- **SET**
  指定列值，可以指定为表达式或DEFAULT。

  说明

  - 表达式中不支持列名。
  - 若表达式结果类型与被赋值列对应类型之间不存在隐式转换函数则报错。

示例

```
-- 准备待导入的文件。
m_db=# CREATE TABLE load_data_tmp1(a int, b int);
m_db=# INSERT INTO load_data_tmp1 VALUES(1,1),(2,2),(3,3);
m_db=# \copy load_data_tmp1 to '/home/omm/load1.csv';
m_db=# CREATE TABLE load_data_tmp2(a int, b int);
m_db=# INSERT INTO load_data_tmp2 VALUES(1,2);
m_db=# \copy load_data_tmp2 to '/home/omm/load2.csv';
m_db=# CREATE TABLE load_data_tmp3(a int, b int);
m_db=# INSERT INTO load_data_tmp3 VALUES(4,4),(5,5);
m_db=# \copy load_data_tmp3 to '/home/omm/load3.csv';
m_db=# CREATE TABLE load_data_tmp4(a char(50));
```



```
m_db=# INSERT INTO load_data_tmp4 VALUES('"load test quote"'), ('\'load test single_quote\'');
m_db=# \copy load_data_tmp4 to '/home/omm/load4.csv';

-- 创建表。
m_db=# CREATE TABLE load_data_tbl1(load_col1 INT UNIQUE, load_col2 INT, load_col3 CHAR(10));

-- 向表中插入一条数据。
m_db=# INSERT INTO load_data_tbl1 VALUES(0,0,'load0');

-- 从文件/home/omm/load1.csv中复制数据到load_data_tbl表，指定列名，设置load_col3列值统一为load。
m_db=# LOAD DATA INFILE '/home/omm/load1.csv' INTO TABLE load_data_tbl1(load_col1, load_col2) SET
load_col3 = 'load';

-- 后续操作导入数据，load_col3列值均为'load'。
m_db=# SELECT * FROM load_data_tbl1 ORDER BY load_col1;
 load_col1 | load_col2 | load_col3
-----------+-----------+-----------
         0 |         0 | load0
         1 |         1 | load
         2 |         2 | load
         3 |         3 | load
(4 rows)

-- 从文件/home/omm/load2.csv中复制数据到load_data_tbl表，指定IGNORE忽略冲突。
m_db=# LOAD DATA INFILE '/home/omm/load2.csv' IGNORE INTO TABLE load_data_tbl1;

-- 表load_data_tbl1中数据不变，冲突数据跳过。
m_db=# SELECT * FROM load_data_tbl1 ORDER BY load_col1;
 load_col1 | load_col2 | load_col3
-----------+-----------+-----------
         0 |         0 | load0
         1 |         1 | load
         2 |         2 | load
         3 |         3 | load
(4 rows)

-- 创建分区表。
m_db=# CREATE TABLE load_data_tbl2
(
    load_col_col1 INT,
    load_col_col2 INT
) PARTITION BY RANGE (load_col_col2)
(
    PARTITION load_p1 VALUES LESS THAN(3),
    PARTITION load_p2 VALUES LESS THAN(9),
    PARTITION load_p3 VALUES LESS THAN(MAXVALUE)
);

-- 从文件/home/omm/load3.csv中复制数据到load_data_tbl2表，指定PARTITION。
m_db=# LOAD DATA INFILE '/home/omm/load3.csv' INTO TABLE load_data_tbl2 PARTITION (load_p2);

-- 数据导入到load_data_tbl2表中指定分区。
m_db=# SELECT * FROM load_data_tbl2;
 load_col_col1 | load_col_col2
---------------+---------------
             4 |             4
             5 |             5
(2 rows)

-- 创建表。
m_db=# CREATE TABLE load_data_tbl3(load_col_col1 CHAR(30));

-- 从文件/home/omm/load4.csv中复制数据到load_data_tbl3表，指定FIELDS ENCLOSED BY。
m_db=# LOAD DATA INFILE '/home/omm/load4.csv' INTO TABLE load_data_tbl3 FIELDS ENCLOSED BY '"';

-- 数据"load test quote"双引号被去掉，'load test single_quote'单引号保留。
m_db=# select * from load_data_tbl3;
         load_col_col1
--------------------------------
```



```
 load test quote
 'load test single_quote'
(2 rows)

-- 删除表。
m_db=# DROP TABLE load_data_tmp1;
m_db=# DROP TABLE load_data_tmp2;
m_db=# DROP TABLE load_data_tmp3;
m_db=# DROP TABLE load_data_tmp4;
m_db=# DROP TABLE load_data_tbl1;
m_db=# DROP TABLE load_data_tbl2;
m_db=# DROP TABLE load_data_tbl3;
```

4.4.2.13.2 LOCK

功能描述

LOCK TABLE获取表级锁。

M-Compatibility模式数据库在为一个引用了表的命令自动请求锁时，尽可能选择最小限制的锁模式。如果用户需要一种更为严格的锁模式，可以使用LOCK命令。例如，一个应用是在Read Committed隔离级别上运行事务，并且它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，则可以在查询之前对表使用READ锁模式进行锁定。这样将防止数据不被并发修改，从而保证后续的查询可以读到已提交的持久化的数据。因为READ锁模式与任何写操作需要的ROW EXCLUSIVE模式冲突，语句将等到所有当前持有ROW EXCLUSIVE模式锁的事务提交或回滚后才能执行。因此，一旦获得该锁，就不会存在未提交的写操作，并且其他操作也只能等到该锁释放之后才能开始。

注意事项

- LOCK TABLE只能在一个事务块的内部有用，因为锁在事务结束时就会被释放。出现在任意事务块外面的LOCK TABLE都会报错。
- 如果没有声明锁模式，缺省为最严格的模式WRITE。
- 没有UNLOCK TABLE命令，锁总是在事务结束时释放。M-Compatibility模式数据库暂不支持UNLOCK TABLE命令。
- LOCK TABLE只处理表级的锁，锁与锁之间是否冲突，规则请参见**表4-25**。
- 如果没有打开xc_maintenance_mode参数，那么对系统表申请WRITE级别锁将报错。

语法格式

```
LOCK [ TABLE | TABLES ]
   tbl_name [lock_type]
    [, tbl_name [lock_type] ] ...
    [ NOWAIT] ;

lock_type: {
    READ
  | WRITE
  | IN ACCESS SHARE MODE
}
```


参数说明

表 4-25 冲突的锁模式

请求的锁模式/当前锁模式	ACCESS SHARE	ROW SHARE	ROW EXCLUSIVE	SHARE UPDATE EXCLUSIVE	READ	SHARE ROW EXCLUSIVE	EXCLUSIVE	WRITE
ACCESS SHARE	-	-	-	-	-	-	-	X
ROW SHARE	-	-	-	-	-	-	X	X
ROW EXCLUSIVE	-	-	-	-	X	X	X	X
SHARE UPDATE EXCLUSIVE	-	-	-	X	X	X	X	X
READ	-	-	X	X	-	X	X	X
SHARE ROW EXCLUSIVE	-	-	X	X	X	X	X	X
EXCLUSIVE	-	X	X	X	X	X	X	X
WRITE	X	X	X	X	X	X	X	X

LOCK的参数说明如下所示：

- **table_name**
  要锁定的表的名称，可以有模式修饰。
  LOCK TABLE命令中声明的表的顺序就是上锁的顺序。
  取值范围：已存在的表名。
- **READ**
  SHARE锁允许并发的查询，但是禁止对表进行修改。
  CREATE INDEX（不带CONCURRENTLY选项）语句会自动请求这种锁。
- **WRITE**
  这个模式保证其所有者（事务）是可以访问该表的唯一事务。

ALTER TABLE、DROP TABLE、TRUNCATE、REINDEX命令会自动请求这种锁。

在LOCK TABLE命令没有明确声明需要的锁模式时，它是缺省锁模式。

- **ACCESS SHARE**

  ACCESS SHARE锁只允许对表进行读取，而禁止对表进行修改。所有对表进行读取而不修改的SQL语句都会自动请求这种锁。例如，SELECT命令会自动在被引用的表上请求一个ACCESS SHARE锁。

- **ROW SHARE | ROW EXCLUSIVE | SHARE UPDATE EXCLUSIVE | SHARE ROW EXCLUSIVE | EXCLUSIVE**

  M-Compatibility模式数据库下暂不支持使用LOCK TABLE获取这些模式的锁。

- **NOWAIT**

  声明LOCK TABLE不去等待任何冲突的锁释放，如果无法立即获取该锁，该命令退出并且发出一个错误信息。

  在不指定NOWAIT的情况下获取表级锁时，如果有其他互斥锁存在的话，则等待其他锁的释放。

示例

```
--创建SCHEMA。
m_db=# CREATE SCHEMA tpcds;

--创建表tpcds.reason。
m_db=# CREATE TABLE tpcds.reason
(
r_reason_sk         INTEGER      NOT NULL,
r_reason_id         CHAR(16)     NOT NULL,
r_reason_desc       INTEGER
);

--向表中插入多条记录。
m_db=# INSERT INTO tpcds.reason VALUES (1, 'AAAAAAAABAAAAAAA', '18'),(5, 'AAAAAAAACAAAAAAA',
'362'),(7, 'AAAAAAAADAAAAAAA', '585');

m_db=# START TRANSACTION;

m_db=# LOCK TABLE tpcds.reason WRITE;

m_db=# COMMIT;

--删除表。
m_db=# DROP TABLE tpcds.reason;

--删除SCHEMA。
m_db=# DROP SCHEMA tpcds;
```

4.4.2.14 P

4.4.2.14.1 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

不支持创建同名的PREPARE语句。

语法格式

```
PREPARE name FROM string
```

参数说明

- **name**
  预备查询标识符。
- **string**
  字符串，为预备SQL语句，不支持使用用户变量@var_name，支持以下语句：
  - SELECT
  - INSERT
  - UPDATE
  - DELETE
  - ALTER [SCHEMA|DATABASE]

    说明

    支持修改数据库字符集字符序，不支持修改数据库所有者。
  - ALTER [TABLE|VIEW|INDEX]
  - ALTER USER
  - CREATE TABLE
  - CREATE USER
  - CREATE [SCHEMA|DATABASE]
  - DROP [TABLE|VIEW|INDEX|SCHEMA|DATABASE]
  - DROP USER
  - [GRANT|REVOKE]
  - CREATE INDEX
  - COMMIT
  - TRUNCATE TABLE
  - SET
  - CREATE VIEW

示例

```
m_db=# PREPARE p1 FROM 'SELECT 10 < 52';
PREPARE
m_db=# EXECUTE p1;
 ?column?
----------
 t
(1 row)
```


4.4.2.14.2 PURGE

功能描述

使用PURGE语句可以实现如下功能：

- 从回收站中清理表或索引，并释放对象相关的全部空间。
- 清理回收站。

注意事项

- 清除（PURGE）操作支持：表（PURGE TABLE）、索引（PURGE INDEX）、回收站（PURGE RECYCLEBIN）。
- 执行PURGE操作的权限要求如下：
  - PURGE TABLE：用户必须是表的所有者，且用户必须拥有表所在模式的USAGE权限，当三权分立开关关闭时，系统管理员默认拥有此权限。
  - PURGE INDEX：用户必须是索引的所有者，用户必须拥有索引所在模式的USAGE权限，当三权分立开关关闭时，系统管理员默认拥有此权限。
  - PURGE RECYCLEBIN：普通用户只能清理回收站中当前用户拥有的对象，且用户必须拥有对象所在模式的USAGE权限，当三权分立开关关闭时，系统管理员默认可以清理回收站所有对象。

前提条件

- 开启enable_recyclebin参数，启用回收站，参数使用请联系管理员处理。
- recyclebin_retention_time参数用于设置回收站对象保留时间，超过该时间的回收站对象将被自动清理，参数使用请联系管理员处理。

语法格式

```
PURGE { TABLE [schema_name.]table_name
      | INDEX index_name
      | RECYCLEBIN
    };
```

参数说明

- **schema_name**

  模式名。

- **TABLE [schema_name.] table_name**

  清空回收站中指定的表，可用模式名修饰。

- **INDEX index_name**

  清空回收站中指定的索引。

- **RECYCLEBIN**

  清空回收站中的对象。

示例

```
-- 创建SCHEMA。
m_db=# CREATE SCHEMA tpcds;
```



```
-- 创建表tpcds.reason_t1
m_db=# CREATE TABLE tpcds.reason_t1
 (
  r_reason_sk    integer,
  r_reason_id    character(16),
  r_reason_desc  character(100)
  ) ;
-- 创建表tpcds.reason_t2
m_db=# CREATE TABLE tpcds.reason_t2
 (
  r_reason_sk    integer,
  r_reason_id    character(16),
  r_reason_desc  character(100)
  ) ;
-- 对表tpcds.reason_t1创建索引
m_db=# CREATE INDEX index_t1 on tpcds.reason_t1(r_reason_id);
m_db=# CREATE INDEX index_t2 on tpcds.reason_t2(r_reason_id);
m_db=# DROP TABLE tpcds.reason_t1;
m_db=# DROP TABLE tpcds.reason_t2;

-- 查看回收站。
m_db=# SELECT rcyname,rcyoriginname,rcytablespace FROM pg_catalog.gs_recyclebin;
           rcyname             | rcyoriginname | rcytablespace
-------------------------------+---------------+---------------
 BIN$41512338419D$87EAEB8==$0 | reason_t1     |             0
 BIN$4151233841A3$87EBBE0==$0 | index_t1      |             0
 BIN$4151233841A0$87EC6A0==$0 | reason_t2     |             0
 BIN$4151233841A4$87ED0F0==$0 | index_t2      |             0
(4 rows)

-- PURGE清除表
m_db=# PURGE TABLE tpcds.reason_t2;
m_db=# SELECT rcyname,rcyoriginname,rcytablespace FROM pg_catalog.gs_recyclebin;

-- PURGE清除索引
m_db=# PURGE INDEX tpcds.index_t1;
m_db=# SELECT rcyname,rcyoriginname,rcytablespace FROM pg_catalog.gs_recyclebin;
           rcyname             | rcyoriginname | rcytablespace
-------------------------------+---------------+---------------
 BIN$41512338419D$87EAEB8==$0 | reason_t1     |             0
 BIN$4151233841A3$87EBBE0==$0 | index_t1      |             0
(2 rows)

-- 回收所有对象
m_db=# PURGE recyclebin;
m_db=# SELECT rcyname,rcyoriginname,rcytablespace FROM pg_catalog.gs_recyclebin;
 rcyname | rcyoriginname | rcytablespace
---------+---------------+---------------
(0 rows)

-- 删除SCHEMA。
 m_db=# DROP SCHEMA tpcds;
```

4.4.2.15 R

4.4.2.15.1 REINDEX

功能描述

为表中的数据重建索引。

在以下几种情况下需要使用REINDEX重建索引：

- 索引崩溃，并且不再包含有效的数据。
- 索引变得“臃肿”，包含大量的空页或接近空页。

- 为索引更改了存储参数（例如填充因子），并且希望这个更改完全生效。

注意事项

- REINDEX DATABASE和SYSTEM这种形式的重建索引不能在事务块中执行。
- 若索引带有lpi_parallel_method选项，取值为PARTITION且表的parallel_workers选项大于0时，不支持对该索引并行重建；无该选项或选项取值为AUTO时，并行重建时会默认走页面级并行重建索引。详见**LPI_PARALLEL_METHOD**。

语法格式

- 重建普通索引。
```
REINDEX { INDEX |  TABLE | DATABASE} [CONCURRENTLY] name [ FORCE ];
REINDEX SYSTEM name [FORCE];
```
- 重建索引分区。
```
REINDEX { INDEX | TABLE} name PARTITION partition_name [ FORCE ];
```

参数说明

- **INDEX**

  重新建立指定的索引。
- **TABLE**

  重新建立指定表的所有索引，如果表有从属的“TOAST”表，则这个表也会重建索引。如果表上有索引已经被alter unusable失效，则这个索引无法被重新创建。当指定CONCURRENTLY选项时，暂不支持重建从属“TOAST”表上的索引。
- **DATABASE**

  重建当前数据库里的所有索引。当指定CONCURRENTLY选项时，暂不支持重建数据库中表的从属“TOAST”表上的索引。
- **SYSTEM**

  在当前数据库上重建所有系统表上的索引。不会处理在用户表上的索引。
- **CONCURRENTLY**

  以不阻塞DML的方式重建索引（加ShareUpdateExclusiveLock锁）。重建索引时，一般会阻塞其他语句对该索引所依赖表的访问。指定此关键字，可以实现重建过程中不阻塞DML。不支持在线重建系统表上的索引。不支持REINDEX INTERNAL TABLE CONCURRENTLY和REINDEX SYSTEM CONCURRENTLY，不支持REINDEX INVALID INDEX CONCURRENTLY。当执行REINDEX DATABASE CONCURRENTLY时，在线重建当前数据库中用户表上的所有索引（不会处理系统表上的索引）。REINDEX CONCURRENTLY不可以在事务内执行。在线重建索引只支持B-tree索引和UB-tree索引，只支持普通索引、GLOBAL索引、LOCAL索引，不支持在线索引字段增删改，不支持PCR ubtree索引，不支持二级分区与GSI。在线并行重建索引只支持Astore及Ustore的普通索引、GLOBAL索引、LOCAL索引，其他继承当前版本在线重建索引规格约束。如果在线重建索引失败，对于用户手动取消、唯一索引键值重复、资源不足、启动线程失败、锁超时等场景，为避免占用资源，系统会自动清理新索引，在系统无法自动清理失败新索引的情况下（比如数据库宕机、FATAL、PANIC），需要尽快手动清除（使用DROP INDEX语句）非法新索引及（使用DROP TABLE语句）临时表，以防占用更多资源。一般来说，非法的新索引的后缀名为_ccnew。REINDEX INDEX CONCURRENTLY对表加4级会话锁，且其前几个阶段与CREATE INDEX CONCURRENTLY相似，因此也可能产生卡住或死锁的问题，具体场景与CREATE INDEX CONCURRENTLY相似（比如两个会话同时对同一个索引或表进行REINDEX CONCURRENTLY操作，会引发死锁问题）。

> **说明**
>
> 重建索引时指定此关键字，Astore需要执行先后两次对全表的扫描来完成build，第一次扫描时创建新索引，不阻塞读写操作，第二次扫描时合并更新第一次扫描到目前为止发生的变更；Ustore需完成一次全表扫描，在扫描过程中并发DML产生的数据会被插入到以“index_oid_cctmp“命名的临时表中，扫描结束后合并临时表到以“_ccnew{n}“为后缀名的新索引中并删除临时表，交换新旧索引，旧索引标记为死亡，启用新索引，重建索引完成。

- **name**

  需要重建索引的索引、表、数据库的名称。表和索引可以有模式修饰。

  > **说明**
  >
  > REINDEX DATABASE和SYSTEM只能重建当前数据库的索引，所以name必须和当前数据库名称相同。

- **FORCE**

  无效选项，会被忽略。

- **partition_name**

  需要重建索引的分区的名称或者索引分区的名称。

  取值范围：

  - 如果前面是REINDEX INDEX，则这里应该指定索引分区的名称。
  - 如果前面是REINDEX TABLE，则这里应该指定分区的名称。

须知

REINDEX DATABASE和SYSTEM这种形式的重建索引不能在事务块中执行。

示例

```
--创建SCHEMA。
m_db=# CREATE SCHEMA tpcds;

--创建表tpcds. customer。
m_db=# CREATE TABLE tpcds.customer
(
c_customer_sk        INTEGER       NOT NULL,
c_customer_id        CHAR(16)       NOT NULL
);

--向表中插入多条记录。
m_db=# INSERT INTO tpcds.customer VALUES (1, 'AAAAAAAABAAAAAAA'),(5, 'AAAAAAAACAAAAAAA'),(10,
'AAAAAAAADAAAAAAA');

--创建一个行存表tpcds.customer_t1，并在tpcds.customer_t1表上的c_customer_sk字段创建索引。
m_db=# CREATE TABLE tpcds.customer_t1
(
    c_customer_sk             integer               not null,
    c_customer_id             char(16)              not null,
    c_current_cdemo_sk        integer                       ,
    c_current_hdemo_sk        integer                       ,
    c_current_addr_sk         integer                       ,
    c_first_shipto_date_sk    integer                       ,
    c_first_sales_date_sk     integer                       ,
    c_salutation              char(10)                      ,
    c_first_name              char(20)                      ,
    c_last_name               char(30)                      ,
    c_preferred_cust_flag     char(1)                       ,
```

```
    c_birth_day               integer                   ,
    c_birth_month             integer                   ,
    c_birth_year              integer                   ,
    c_birth_country           varchar(20)               ,
    c_login                   char(13)                  ,
    c_email_address           char(50)                  ,
    c_last_review_date        char(10)
)
WITH (orientation = row);

m_db=# CREATE INDEX tpcds_customer_index1 ON tpcds.customer_t1 (c_customer_sk);

m_db=# INSERT INTO tpcds.customer_t1 SELECT * FROM tpcds.customer WHERE c_customer_sk < 10;

--重建一个单独索引。
m_db=# REINDEX INDEX tpcds.tpcds_customer_index1;

--重建表tpcds.customer_t1上的所有索引。
m_db=# REINDEX TABLE tpcds.customer_t1;

--删除tpcds.customer_t1表。
m_db=# DROP TABLE tpcds.customer_t1;

--删除表。
m_db=# DROP TABLE tpcds.customer;

--删除SCHEMA。
m_db=# DROP SCHEMA tpcds;
```

优化建议

- INTERNAL TABLE

  此种情况大多用于故障恢复，不建议进行并发操作。
- DATABASE

  不能在事务中REINDEX DATABASE。
- SYSTEM

  不能在事务中REINDEX系统表。

4.4.2.15.2 RELEASE SAVEPOINT

功能描述

RELEASE SAVEPOINT删除一个当前事务先前定义的保存点。

把一个保存点删除就令其无法作为回滚点使用，除此之外它没有其它用户可见的行为。它并不能撤销在保存点建立起来之后执行的命令的影响，要撤销那些命令可以使用ROLLBACK TO SAVEPOINT 。当不再需要的时候，删除一个保存点可以令系统在事务结束之前提前回收一些资源。

RELEASE SAVEPOINT也删除所有在指定的保存点建立之后的所有保存点。

注意事项

- 不能RELEASE一个没有定义的保存点，语法上会报错。
- 如果事务在回滚状态，则不能释放保存点。
- 如果多个保存点拥有同样的名称，只有最近定义的保存点才被释放。

语法格式

```
RELEASE [ SAVEPOINT ] savepoint_name;
```

参数说明

- **savepoint_name**
  要删除的保存点的名称

示例

```
--创建SCHEMA。
m_db=# CREATE SCHEMA tpcds;

--创建一个新表。
m_db=# CREATE TABLE tpcds.table1(a int);

--开启事务。
m_db=# START TRANSACTION;

--插入数据。
m_db=# INSERT INTO tpcds.table1 VALUES (3);

--建立保存点。
m_db=# SAVEPOINT my_savepoint;

--插入数据。
m_db=# INSERT INTO tpcds.table1 VALUES (4);

--删除保存点。
m_db=# RELEASE SAVEPOINT my_savepoint;

--提交事务。
m_db=# COMMIT;

--查询表的内容，会同时看到3和4。
m_db=# SELECT * FROM tpcds.table1;

--删除表。
m_db=# DROP TABLE tpcds.table1;

--删除SCHEMA。
m_db=# DROP SCHEMA tpcds;
```

相关链接

SAVEPOINT，**ROLLBACK TO SAVEPOINT**

4.4.2.15.3 RENAME TABLE

功能描述

用于修改单个表或多个表的名称。对名称的修改不会影响所存储的数据。

语法格式

```
RENAME { TABLE | TABLES } table_name TO new_table_name [, table_name2 TO new_table_name2, ...];
```

参数说明

- **TABLE | TABLES**
  TABLE和TABLES可以互相替换使用，与语句中操作表的个数无关。

- **table_name TO new_table_name [, table_name2 TO new_table_name2, ...]**

  table_name、table_name2 ...：需要修改的表名。

  new_table_name、new_table_name2...：修改后的新表名。

  TO：连接词，无实际含义。

修改表示例

- 单表修改名称

```
gaussdb=# CREATE TABLE aa(c1 int, c2 int);
gaussdb=# RENAME TABLE aa TO test_alt1;
gaussdb=# DROP TABLE test_alt1;
```

- 多个表修改名称

```
gaussdb=# CREATE TABLE aa(c1 int, c2 int);
gaussdb=# CREATE TABLE bb(c1 int, c2 int);
gaussdb=# RENAME TABLE aa TO test_alt1, bb TO test_alt2;
gaussdb=# DROP TABLE test_alt1,test_alt2;
```

相关链接

ALTER TABLE

4.4.2.15.4 REPLACE

功能描述

在表中插入或者替换新的数据。当插入的数据与原有数据存在主键/唯一键冲突时，执行REPLACE语句会先删除原有数据，再插入新的数据。

REPLACE语句有如下三种形式：

- 值替换插入。即通过VALUES或VALUE构造一行记录，并插入到表中。
- 查询替换插入。通过SELECT子句返回的结果集构造一行或多行记录，并插入到表中。
- 设定指定字段值。与值插入类似，对于没有指定的列取其默认值。

注意事项

- 执行该语句的用户需要有表的DELETE和INSERT权限。
- 主键/唯一键不冲突的情况下，可直接插入；对于主键/唯一键冲突的情况，先删除原有数据，再插入新的数据。
- REPLACE操作返回格式为REPLACE 0 X，X表示删除和插入的操作数。
- REPLACE...SELECT形式，select_list列数必须与待插入的字段数保持一致。
- REPLACE...SET语法中：
  - 若col_name有默认值，则SET col_name = col_name + 1等同于 SET col_name = col_name的默认值 + 1；
  - 若col_name无默认值且没有NOT NULL约束时，则SET col_name = col_name + 1等同于col_name = NULL；
  - 若col_name无默认值但存在NOT NULL约束时，各数据类型的默认零值如下表：

数据类型	默认零值
tinyint [unsigned | signed] 、smallint [unsigned | signed]、mediumint [unsigned | signed]、int [unsigned | signed]、bigint [unsigned | signed]、float4、float8、numeric、bool	0，如果numeric指定小数位，则显示小数位。如NUMERIC(10, 3)：0.000。
char、varchar、tinytext、text、mediumtext、longtext、tinyblob、blob、mediumblob、longblob、bit、varbinary	''
binary	''，空串长度等于指定的长度，未指定长度时长度为1。
time	00:00:00。
date	0000-00-00。
timestamp	0000-00-00 00:00:00。
datetime	0000-00-00 00:00:00。
year	0000。

说明

- date、timestamp和datetime需在宽松模式下才能插入时间零值。
- 函数表达式的默认值，如果在解析时可计算出结果，则默认值为计算出的常量，否则为NULL。
- 除以上场景，默认值为NULL。

- REPLACE ... SET语法中，后面设置的col_name依赖于前面col_name的值，如果前面没有设置，则取默认值。如SET f1 = f1 + 1, f2 = f1场景下，f1等于f1的默认值（假设为0）+1，f2等于f1计算后的值1。
- 不支持延迟生效（DEFERRABLE）的唯一约束或主键。
- 如果表中存在多个唯一约束，如果所插入数据违反多个唯一约束， 则会删除所有违反约束的数据，插入新的数据。此场景需要注意，可能会误删不需要删除的数据，该场景需要慎重操作。
- 如果插入多行，这些行均与表中同一行数据存在唯一约束冲突，则按照执行顺序，依次进行REPLACE操作。
- SET col_name = col_name + 1，col_name的长度不支持超过1。如SET col_name = table_name.col_name + 1中，table_name.col_name的长度为2，形如B.A、C.B.A、D.C.B.A格式都不支持。
- 浮点型比较时，需要注意浮点型可能存在精度丢失。
- 不支持密态表。
- 不支持内存表。

语法格式

- 值替换插入。

```
REPLACE [ INTO ] table_name
    [ PARTITION ( partition_name [, ... ] ) ]
    [ ( col_name [, ... ] ) ]
    { VALUES | VALUE } ( value [, ... ] ) [, ... ];

--其中value格式为：
{ expression | DEFAULT }
```

- 查询替换插入。

```
REPLACE [ INTO ] table_name
    [ PARTITION ( partition_name [, ... ] ) ]
    [ ( col_name [, ... ]) ]
    query;
```

- 设置指定字段值。

```
REPLACE [ INTO ] table_name
    [ PARTITION ( partition_name [, ... ] ) ]
    SET col_name = value [, ... ];

--其中value格式为：
{ expression | DEFAULT }
```

参数说明

- **table_name**

  要插入数据的目标表名。

  取值范围：已存在的表名。

- **col_name**

  目标表中的字段名。

  - 字段名可以有子字段名或者数组下标修饰。
  - 没有在字段列表中出现的每个字段，将由系统默认值，或者声明时的默认值填充，若都没有则用NULL填充。例如，向一个复合类型中的某些字段插入数据，那么其他字段将是NULL。
  - 目标字段（col_name）可以按顺序排列。如果没有列出任何字段，则默认全部字段，且顺序为表声明时的顺序。
  - 如果VALUE子句和QUERY中只提供了N个字段，则目标字段为前N个字段。
  - VALUE子句和QUERY提供的值在表中从左到右关联到对应列。

  取值范围：已存在的字段名。

- **PARTITION (partition_name [, ...])**

  指定分区插入操作。其中partition_name为分区名。

  如果VALUE子句的值和指定分区不一致，结果会提示异常。

- **value**

  待插入的值，value格式为：

  ```
  { expression | DEFAULT }
  ```

  a. expression表示赋予对应列一个有效表达式或值

     向表中字段插入单引号 " ' "时需要使用单引号自身进行转义。

     如果插入行的表达式不是正确的数据类型，系统试图进行类型转换，若转换不成功，则插入数据失败，系统返回错误信息。

  b. DEFAULT表示对应字段名的缺省值。如果没有缺省值，则为NULL。

- **query**
  一个查询语句（SELECT语句），将查询结果作为插入的数据。

示例

```
--创建一个新表。
m_db=# CREATE TABLE test(f1 int primary key, f2 int, f3 int);

--插入数据。
m_db=# INSERT INTO test VALUES(1, 1, 1), (2, 2, 2), (3, 3, 3);
INSERT 0 3

--值替换插入数据。
m_db=# REPLACE INTO test VALUES(1, 11, 11);
REPLACE 0 2

--查询值替换插入的结果
m_db=# SELECT * FROM test WHERE f1 = 1;
 f1 | f2 | f3
----+----+----
  1 | 11 | 11
(1 row)

--查询替换插入数据。
m_db=# REPLACE INTO test SELECT 2, 22, 22;
REPLACE 0 2

--查询查询替换插入的结果
m_db=# SELECT * FROM test WHERE f1 = 2;
 f1 | f2 | f3
----+----+----
  2 | 22 | 22
(1 row)

--设置指定字段替换插入数据。
m_db=# REPLACE INTO test SET f1 = f1 + 3, f2 = f1 * 10 + 3, f3 = f2;
REPLACE 0 2

--查询设置指定字段替换插入数据的结果
m_db=# SELECT * FROM test WHERE f1 = 3;
 f1 | f2 | f3
----+----+----
  3 | 33 | 33
(1 row)

--删除表格
m_db=# DROP TABLE test;
```

4.4.2.15.5 RESET

功能描述

RESET将指定的运行时参数恢复为缺省值。这些参数的缺省值是指gaussdb.conf配置文件中所描述的参数缺省值。

当前版本不支持 RESET configuration_parameter，只支持RESET ALL。

注意事项

- RESET ALL的事务性行为和SET相同，它的影响将会被事务回滚撤销。
- RESET ALL不能对role和session authorization恢复为缺省值。

语法格式

```
RESET ALL;
```

参数说明

- **ALL**
  所有运行时参数。

示例

```
--把所有参数设置为缺省值。
m_db=# RESET ALL;
```

相关链接

SET，**SHOW**

4.4.2.15.6 REVOKE

功能描述

REVOKE用于撤销一个或多个角色的权限。

注意事项

非对象所有者试图在对象上REVOKE权限，命令按照以下规则执行：

- 如果授权用户没有该对象上的权限，则命令立即失败。
- 如果授权用户有部分权限，则只撤销那些有授权选项的权限。
- 如果授权用户没有授权选项，REVOKE ALL PRIVILEGES形式将发出一个错误信息，而对于其他形式的命令而言，如果是命令中指定名称的权限没有相应的授权选项，该命令将发出一个警告。

语法格式

- 回收指定表或视图上权限。
```
REVOKE [ GRANT OPTION FOR ]
    { { SELECT | INSERT | UPDATE | DELETE | TRUNCATE | REFERENCES | ALTER | DROP | COMMENT |
INDEX | VACUUM }[, ...]
    | ALL [ PRIVILEGES ] }
    ON { [ TABLE ] table_name [, ...]
        | ALL TABLES IN SCHEMA schema_name [, ...] }
    FROM { [ GROUP ] role_name | PUBLIC } [, ...]
    [ CASCADE | RESTRICT ];
```
- 回收表上指定字段权限。
```
REVOKE [ GRANT OPTION FOR ]
    { {{ SELECT | INSERT | UPDATE | REFERENCES | COMMENT } ( column_name [, ...] )}[, ...]
    | ALL [ PRIVILEGES ] ( column_name [, ...] ) }
    ON [ TABLE ] table_name [, ...]
    FROM { [ GROUP ] role_name | PUBLIC } [, ...]
    [ CASCADE | RESTRICT ];
```
- 回收指定序列上权限，LARGE字段属性可选，回收语句不区分序列是否为LARGE。
```
REVOKE [ GRANT OPTION FOR ]
    { { SELECT | UPDATE | ALTER | DROP | COMMENT }[, ...]
    | ALL [ PRIVILEGES ] }
    ON { [ [ LARGE ] SEQUENCE ] sequence_name [, ...] }
```



```
  FROM { [ GROUP ] role_name | PUBLIC } [, ...]
  [ CASCADE | RESTRICT ];
```

- 回收指定数据库上权限。

```
REVOKE [ GRANT OPTION FOR ]
  { { CREATE | CONNECT | TEMPORARY | TEMP | ALTER | DROP | COMMENT } [, ...]
  | ALL [ PRIVILEGES ] }
  ON DATABASE database_name [, ...]
  FROM { [ GROUP ] role_name | PUBLIC } [, ...]
  [ CASCADE | RESTRICT ];
```

  说明

  GUC参数m_format_behavior_compat_options开启grant_database_nomapping选项时该语法生效，回收指定的用户或角色对数据库层级对象的访问权限；未开启grant_database_nomapping选项时，REVOKE ON DATABASE映射为REVOKE ON SCHEMA，回收指定的用户或角色对模式层级对象的访问权限，REVOKE ON DATABASE也仅支持REVOKE ON SCHEMA支持的语法选项。

- 回收指定模式上权限。

```
REVOKE [ GRANT OPTION FOR ]
  { { CREATE | USAPRIVILEGEGE | ALTER | DROP | COMMENT } [, ...] | ALL [ PRIVILEGES ] }
  ON SCHEMA schema_name [, ...]
  FROM { [ GROUP ] role_name | PUBLIC } [, ...]
  [ CASCADE | RESTRICT ];
```

- 按角色回收角色上的权限。

```
REVOKE [ ADMIN OPTION FOR ]
  role_name [, ...] FROM role_name [, ...]
  [ CASCADE | RESTRICT ];
```

- 回收角色上的sysadmin权限。

```
REVOKE ALL PRIVILEGES FROM role_name;
```

- 回收ANY权限。

```
REVOKE [ ADMIN OPTION FOR ]
  { 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 |
ALTER ANY SEQUENCE | DROP ANY SEQUENCE |
  SELECT ANY SEQUENCE | ALTER ANY INDEX | DROP ANY INDEX
} [, ...]
  FROM [ GROUP ] role_name [, ...];
```

参数说明

关键字PUBLIC表示一个隐式定义的拥有所有角色的组。

权限类别和参数说明，请参见GRANT的**参数说明**。

任何特定角色拥有的特权包括直接授予该角色的特权、从该角色作为其成员的角色中得到的权限以及授予给PUBLIC的权限。因此，从PUBLIC收回SELECT特权并不一定会意味着所有角色都会失去在该对象上的SELECT特权，那些直接被授予的或者通过另一个角色被授予的角色仍然会拥有它。类似地，从一个用户收回SELECT后，如果PUBLIC仍有SELECT权限，该用户还是可以使用SELECT。

指定GRANT OPTION FOR时，只撤销对该权限授权的权力，而不撤销该权限本身。

如用户A拥有某个表的UPDATE权限，及WITH GRANT OPTION选项，同时A把这个权限赋予了用户B，则用户B持有的权限称为依赖性权限。当用户A持有的权限或者授权选项被撤销时，必须声明CASCADE，将所有依赖性权限都撤销。

一个用户只能撤销由它自己直接赋予的权限。例如，如果用户A被指定授权（WITH ADMIN OPTION）选项，且把一个权限赋予了用户B，然后用户B又赋予了用户C，则用户A不能直接将C的权限撤销。但是，用户A可以撤销用户B的授权选项，并且使用

CASCADE。这样，用户C的权限就会自动被撤销。另外一个例子：如果A和B都赋予了C同样的权限，则A可以撤销他自己的授权选项，但是不能撤销B的，因此C仍然拥有该权限。

如果执行REVOKE的角色持有的权限是通过多层成员关系获得的，则具体是哪一个包含的角色执行的该命令是不确定的。在这种场合下，建议使用SET ROLE成为特定角色，然后执行REVOKE，否则可能导致删除了不想删除的权限，或者是任何权限都没有删除。

示例

请参考GRANT的**示例**。

相关链接

GRANT

4.4.2.15.7 ROLLBACK

功能描述

回滚当前事务并取消当前事务中的所有更新。

在事务运行的过程中发生了某种故障，事务不能继续执行，系统将事务中对数据库的所有已完成的操作全部撤销，数据库状态回到事务开始时。

注意事项

如果不在一个事务内部发出ROLLBACK不会有问题，但是将抛出一个NOTICE信息。

语法格式

```
ROLLBACK [ WORK | TRANSACTION ];
```

参数说明

- **WORK | TRANSACTION**

  可选关键字。除了增加可读性，没有任何其他作用。

示例

```
--开启一个事务
m_db=# START TRANSACTION;

--取消所有更改
m_db=# ROLLBACK;
```

相关链接

COMMIT

4.4.2.15.8 ROLLBACK TO SAVEPOINT

功能描述

ROLLBACK TO SAVEPOINT用于回滚到一个保存点，隐含地删除所有在该保存点之后建立的保存点。

回滚所有指定保存点建立之后执行的命令。保存点仍然有效，并且需要时可以再次回滚到该点。

注意事项

- 不能回滚到一个未定义的保存点，语法上会报错。
- 使用ROLLBACK TO SAVEPOINT回滚到一个保存点。使用RELEASE SAVEPOINT删除一个保存点，但是保留该保存点建立后执行的命令的效果。

语法格式

```
ROLLBACK [ WORK | TRANSACTION ] TO [ SAVEPOINT ] savepoint_name;
```

参数说明

- **savepoint_name**

  回滚截至的保存点

示例

```
--撤销 my_savepoint 建立之后执行的命令的影响。
m_db=# START TRANSACTION;
m_db=# SAVEPOINT my_savepoint;
m_db=# ROLLBACK TO SAVEPOINT my_savepoint;
m_db=# RELEASE SAVEPOINT my_savepoint;
m_db=# COMMIT;
```

相关链接

SAVEPOINT，**RELEASE SAVEPOINT**

4.4.2.16 S

4.4.2.16.1 SAVEPOINT

功能描述

SAVEPOINT用于在当前事务里建立一个新的保存点。

保存点是事务中的一个特殊记号，它允许将那些在它建立后执行的命令全部回滚，把事务的状态恢复到保存点所在的时刻。

注意事项

- 使用ROLLBACK TO SAVEPOINT回滚到一个保存点。使用RELEASE SAVEPOINT删除一个保存点，但是保留该保存点建立后执行的命令的效果。
- 保存点只能在一个事务块里面建立。在一个事务里面可以定义多个保存点。

- 由于节点故障或者通信故障引起的节点线程或进程退出导致的报错，以及由于COPY FROM操作中源数据与目标表的表结构不一致导致的报错，均不能正常回滚到保存点之前，而是整个事务回滚。
- SQL标准要求，使用SAVEPOINT建立一个同名保存点时，需要自动删除前面的同名保存点。在M-Compatibility数据库里，将保留旧的保存点，但是在回滚或者释放的时候，只使用最近的保存点。释放了新的保存点将导致旧的再次成为ROLLBACK TO SAVEPOINT和RELEASE SAVEPOINT可以访问的保存点。除此之外，SAVEPOINT是完全符合SQL标准的。

语法格式

```
SAVEPOINT savepoint_name;
```

参数说明

- **savepoint_name**
  新建保存点的名称。

示例

```
--创建一个新表。
m_db=# CREATE TABLE table1(a int);

--开启事务。
m_db=# START TRANSACTION;

--插入数据。
m_db=# INSERT INTO table1 VALUES (1);

--建立保存点。
m_db=# SAVEPOINT my_savepoint;

--插入数据。
m_db=# INSERT INTO table1 VALUES (2);

--回滚保存点。
m_db=# ROLLBACK TO SAVEPOINT my_savepoint;

--插入数据。
m_db=# INSERT INTO table1 VALUES (3);

--提交事务。
m_db=# COMMIT;

--查询表的内容，会同时看到1和3,不能看到2，因为2被回滚。
m_db=# SELECT * FROM table1;

--删除表。
m_db=# DROP TABLE table1;

--创建一个新表。
m_db=# CREATE TABLE table2(a int);

--开启事务。
m_db=# START TRANSACTION;

--插入数据。
m_db=# INSERT INTO table2 VALUES (3);

--建立保存点。
m_db=# SAVEPOINT my_savepoint;

--插入数据。
m_db=# INSERT INTO table2 VALUES (4);
```



```
--回滚保存点。
m_db=# RELEASE SAVEPOINT my_savepoint;

--提交事务。
m_db=# COMMIT;

--查询表的内容，会同时看到3和4。
m_db=# SELECT * FROM table2;

--删除表。
m_db=# DROP TABLE table2;
```

相关链接

RELEASE SAVEPOINT，**ROLLBACK TO SAVEPOINT**

4.4.2.16.2 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

- 表的所有者、拥有表SELECT权限的用户或拥有SELECT ANY TABLE权限的用户，有权读取表或视图中数据，当三权分立开关关闭时，系统管理员默认拥有此权限。
- 必须对每个在SELECT命令中使用的字段有SELECT权限。
- 使用FOR UPDATE、FOR SHARE还要求UPDATE权限。

语法格式

查询数据

```
[ WITH [ RECURSIVE ] with_query [, ...] ]
SELECT [/*+ plan_hint */]
    [ALL | DISTINCT | DISTINCTROW]
    [STRAIGHT_JOIN]
    [SQL_NO_CACHE]
    [SQL_CALC_FOUND_ROWS]
    select_expr [, select_expr] ...
    [into_option]
    [FROM from_item [,...]]
    [WHERE where_condition]
    [GROUP BY {col_name | expr | position} [WITH ROLLUP]]
    [HAVING condition]
    [WINDOW {window_name AS (window_definition)} [, ...]]
    [ORDER BY {col_name | expression | position}
      [ASC | DESC][ NULLS { FIRST | LAST } ], ...]
    [[LIMIT {[offset,] row_count | row_count OFFSET offset}] | FETCH {FIRST | NEXT} [(expr)] {ROW| ROWS}
ONLY]
    [into_option]
    [ FOR READ ONLY |
        [ {FOR { UPDATE | SHARE } [ OF table_name [, ...] ] [ NOWAIT | WAIT N | SKIP LOCKED ]} [...] ] |
        LOCK IN SHARE MODE]

select_expr：
```



```
    expression [ [AS] output_name ]

where_condition:
expr

into_option:
INTO {
  [[[LOCAL | GLOBAL] TEMPORARY] [TABLE] table_name] |
  OUTFILE file_name [CHARACTER SET encoding_name] [FIELDS fields_items] [LINES lines_items] |
  DUMPFILE file_name
}
```

SELECT.. UNION/EXCEPT语法：

```
SELECT ...
    UNION [ALL | DISTINCT] SELECT ...
    [UNION [ALL | DISTINCT] SELECT ...]
SELECT ...
    EXCEPT[ALL | DISTINCT] SELECT ...
    [EXCEPT[ALL | DISTINCT] SELECT ...]
```

说明

WHERE子句、HAVING子句、GROUP BY子句中使用SELECT子句中的别名，存在如下限制：

- 初始化建表语句。
  ```
  m_db=# DROP TABLE IF EXISTS t0,t1;
  m_db=# CREATE TABLE t0 (c0 INT, c1 INT);
  m_db=# CREATE TABLE t1 (c0 INT, c1 INT);
  ```
- 只能同一层引用。
  ```
  m_db=# SELECT r FROM t0 WHERE (select c0 as r from t0) = 1; -- error
  ```
- 只能引用targetlist中的别名。
  ```
  m_db=# SELECT * FROM t0 WHERE (SELECT c0 AS r FROM t0) = 1 AND r = 2; -- error
  ```
- 只能是后面的表达式引用前面的表达式。
  ```
  m_db=# SELECT r AS d, c0 AS r FROM t0; -- error
  ```
- 不能包含volatile函数。
  ```
  m_db=# SELECT uuid() AS r FROM t0 WHERE r = 1; -- error
  ```
- 不能包含Window function函数。
  ```
  m_db=# SELECT pg_catalog.rank() over(PARTITION BY c0 ORDER BY c1) AS r FROM t0 WHERE r =
  1; -- error
  ```
- 不支持在join on条件中引用别名。
  ```
  m_db=# SELECT uuid() AS r FROM t0 JOIN t1 ON r = 1; -- error
  ```
- targetlist中有多个要引用的别名则报错。
  ```
  m_db=# SELECT c0 AS r AS u AS v FROM t0; -- error
  ```

- 其中子查询with_query为：
  ```
  with_query_name [ ( column_name [, ...] ) ]
     AS ( {select} )
  ```
- 其中INTO子句为：
  ```
  into_option: {
        INTO [[[LOCAL | GLOBAL] TEMPORARY] [TABLE]] table_name
     | INTO OUTFILE 'file_name'
        [CHARACTER SET charset_name]
        export_options
     | INTO DUMPFILE 'file_name'
  }
  export_options:
     [FIELDS
   [TERMINATED BY 'string']
   [[OPTIONALLY] ENCLOSED BY 'char']
   [ESCAPED BY 'char' ]
     ]
     [LINES
   [STARTING BY 'string']
   [TERMINATED BY 'string']
  ```



```
  ]
}
```

- 其中指定查询源from_item为：

```
{ table_name  [ partition_clause ] [ [ AS ] alias [ ( column_alias [, ...] ) ] ]
|( 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 [, ...] )
|joined_table
```

说明

table_name包含dual虚拟表。M-Compatibility允许在没有引用表的情况下，将dual指定为虚拟表名。

- 其中joined_table为：

```
joined_table: {
   table_reference [INNER | CROSS] JOIN table_factor [join_specification]
  | table_reference STRAIGHT_JOIN table_factor [join_specification]
  | table_reference {LEFT|RIGHT} [OUTER] JOIN table_reference join_specification
  | table_reference NATURAL [{LEFT|RIGHT} [OUTER]] JOIN table_factor
}
join_specification: {
   ON join_condition
  | USING (join_column_list)
}
join_column_list:
    column_name [, column_name] ...
```

- 其中group子句为：

```
| expression
| ( expression [, ...] )
```

- from_item中指定分区partition_clause为：

```
PARTITION { ( { partition_name | subpartition_name } [, ...] ) | FOR (  partition_value [, ...] ) } |
SUBPARTITION { ( subpartition_name ) | FOR (  subpartition_value [, ...] )}
```

说明

- 指定分区只适合分区表。
- PARTITION指定多个分区名时，一级分区名和二级分区名可同时存在，且可以存在相同的分区名，最终分区范围取其并集。

参数说明

- **plan_hint**

  以/*+ */的形式在SELECT关键字后，用于对SELECT对应的语句块生成的计划进行hint调优。每条语句中只有第一个/*+ plan_hint */注释块会作为hint生效，里面可以写多条hint。

- **WITH [RECURSIVE] with_query [, ...]**

  用于声明一个或多个可以在主查询中通过名称引用的子查询，相当于临时表。这种子查询语句结构称为CTE（Common Table Expression）结构，应用这种结构时，执行计划中将存在CTE SCAN的内容。

  如果声明了RECURSIVE，那么允许SELECT子查询通过名称引用其本身。

  其中with_query的详细格式为：

  ```
  with_query_name [ ( column_name [, ...] ) ] AS ( {select} )
  ```

  - with_query_name指定子查询生成的结果集名称，在查询中可使用该名称访问子查询的结果集。
  - column_name指定子查询结果集中显示的列名。

- 每个子查询支持SELECT语句。
- RECURSIVE只能出现在WITH后面，多个CTE的情况下，只需要在第一个CTE处声明RECURSIVE。
- 使用RECURSIVE时，CTE子查询中UNION ALL和EXCEPT ALL或UNION [DISTINCT]和EXCEPT [DISTINCT]两侧的子查询结果，其数据类型必须使用cast函数转换成相同的数据类型，且两侧子查询结果的精度和字符序也要相同。如：WITH RECURSIVE cte (n) AS (SELECT cast(id as signed int) from table_1 UNION ALL SELECT cast((n + 1) as signed int) FROM cte WHERE n < 5) SELECT * FROM cte。由操作符产生的类型转换具体请参见**逻辑操作符规格约束**、**位运算操作符规格约束**和**算术操作符规格约束**。

- **ALL**

  声明返回所有符合条件的行，是默认行为，可以省略该关键字。

- **DISTINCT | DISTINCTROW**

  从SELECT的结果集中删除所有重复的行，使结果集中的每行都是唯一的。

  须知

  DISTINCT|DISTINCTROW表达式是使用与ORDER BY相同的规则进行解释的。除非使用了ORDER BY来保证需要的行首先出现，否则，"第一行" 是不可预测的。

- **STRAIGHT_JOIN**

  对于内联接，可改变优化器对于联表查询的执行顺序，使左边的表始终在右右边表之前读取。

- **SQL_CALC_FOUND_ROWS**

  使用found_rows函数计算结果集中有多少行时，忽略任何LIMIT子句。

- **SELECT列表**

  指定查询表中列名，可以是部分列或者是全部（使用通配符*表示）。

  通过使用子句AS output_name可以为输出字段取个别名，这个别名通常用于输出字段的显示。支持关键字name、value和type作为列别名。

  列名可以用下面几种形式表达：

  - 手动输入列名，多个列之间用英文逗号“,”分隔。
  - 可以是FROM子句里面计算出来的字段。

- **INTO子句**

  将SELECT出的结果输出到指定用户自定义变量或文件。

  - OUTFILE
    - CHARACTER SET 指定编码格式。
    - FIELDS 指定每个字段的属性：

      TERMINATED 指定间隔符。

      [OPTIONALLY] ENCLOSED 指定引号符，指定OPTIONALLY时只对字符串数据类型起作用。

      ESCAPED 指定转义符。
    - LINES 指定行属性：

      STARTING 指定行开头。

TERMINATED 指定行结尾。

- DUMPFILE

  导出无间隔符，无换行符的单行数据到文件。

- file_name

  指定文件的绝对路径。

```
导出到文件：
m_db=# SELECT * FROM t;
 a | b
---+---
 1 | a
(1 row)
--导出数据到outfile文件。
m_db=# SELECT * FROM t INTO OUTFILE '/home/gaussdb/t.txt' FIELDS TERMINATED BY '~'
ENCLOSED BY 't' ESCAPED BY '^' LINES STARTING BY '$' TERMINATED BY '&\n';
文件内容：$t1t~tat&，其中LINES STARTING BY($),FIELDS TERMINATED BY(~),ENCLOSED
BY(t),LINES TERMINATED BY(&\n)。
--导出数据到dumpfile文件。
m_db=# SELECT * FROM t INTO DUMPFILE '/home/gaussdb/t.txt';
文件内容：1a
```

- **FROM子句**

  为SELECT声明一个或多个源表。

  FROM子句涉及的元素如下所示。

  - table_name

    表名或视图名，名称前可加上模式名，如：schema_name.table_name。

  - alias

    给表或复杂的表引用起一个临时的表别名，以便被其余的查询引用。

    别名用于缩写或者在自连接中消除歧义。如果提供了别名，它就会完全隐藏表的实际名称。

  - column_alias

    列别名

  - PARTITION

    查询分区表的某个分区的数据。

  - partition_name

    分区名。

  - partition_value

    指定的分区键值。在创建分区表时，如果指定了多个分区键，可以通过PARTITION FOR子句指定的这一组分区键的值，唯一确定一个分区。

  - SUBPARTITION

    查询分区表的某个二级分区的数据。

  - subpartition_name

    二级分区名。

  - subpartition_value

    指定的一级分区和二级分区键值。可以通过SUBPARTITION FOR子句指定的两个分区键的值，唯一确定一个二级分区。

  - subquery

    FROM子句中可以出现子查询，创建一个临时表保存子查询的输出。

  - with_query_name

WITH子句同样可以作为FROM子句的源，可以通过WITH查询的名称对其进行引用。

- function_name

  函数名称。函数调用也可以出现在FROM子句中。

- join有以下类型：

  - [INNER] JOIN

    一个JOIN子句组合两个FROM项。可使用圆括弧以决定嵌套的顺序。如果没有圆括弧，JOIN从左向右嵌套。

    在任何情况下，JOIN都比逗号分隔的FROM项绑定得更紧。

  - LEFT [OUTER] JOIN

    返回笛卡尔积中所有符合连接条件的行，再加上左表中通过连接条件没有匹配到右表行的那些行。这样，左边的行将扩展为生成表的全长，方法是在那些右表对应的字段位置填上NULL。请注意，只在计算匹配的时候，才使用JOIN子句的条件，外层的条件是在计算完毕之后施加的。

  - RIGHT [OUTER] JOIN

    返回所有内连接的结果行，加上每个不匹配的右边行（左边用NULL扩展）。

    这只是一个符号上的方便，因为总是可以把它转换成一个LEFT OUTER JOIN，只要把左边和右边的输入互换位置即可。

  - CROSS JOIN

    CROSS JOIN等效于INNER JOIN ON(TRUE)，即没有被条件删除的行。这种连接类型与简单的FROM和WHERE的效果相同，相对来说在符号使用上更加便捷。

    说明

    必须为INNER和OUTER连接类型声明一个连接条件，即NATURAL ON、join_condition、USING (join_column [， ...]) 之一。

  - STRAIGHT_JOIN

    与INNER JOIN功能相似。STRAIGHT_JOIN场景下，左边的表始终在右右边的表之前读取，可改变优化器对于联表查询的执行顺序。

  其中CROSS JOIN和INNER JOIN生成一个简单的笛卡尔积，和在FROM的顶层列出两个项的结果相同。

  允许连续使用多次JOIN。

- ON join_condition

  连接条件，用于限定连接中的哪些行是匹配的。如：ON left_table.a = right_table.a。不建议使用int等数值类型作为join_condition，因为int等数值类型可以隐式转换为bool值（非0值隐式转换为true，0转换为false），可能导致非预期的结果。

- USING(join_column[，...])

  ON left_table.a = right_table.a AND left_table.b = right_table.b ... 的简写。要求对应的列必须同名。

- NATURAL

  NATURAL是具有相同名称的两个表的所有列的USING列表的简写。

- from item

  用于连接的查询源对象的名称。

- **WHERE子句**

  WHERE子句构成一个行选择表达式，用来缩小SELECT查询的范围。condition是返回值为布尔型的任意表达式，任何不满足该条件的行都不会被检索。不建议使用int等数值类型作为condition，因为int等数值类型可以隐式转换为bool值（非0值隐式转换为true，0转换为false），可能导致非预期的结果。

  须知

  对于WHERE子句的LIKE操作符，当LIKE中要查询特殊字符“%”、“_”、“\”的时候需要使用反斜杠“\”来进行转义。

- **GROUP BY子句**

  将查询结果按某一列或多列的值分组，值相等的为一组。

  WITH ROLLUP：

  若想对分组的查询结果进行统计，GROUP BY语句结尾添加WITH ROLLUP语句。WITH ROLLUP也可以对多列分组查询结果进行统计

  须知

  若sql_mode中包含ONLY_FULL_GROUP_BY选项。

  1. SELECT列表中非聚合函数列与GROUP BY字段一致时，不会报错。
  2. SELECT列表中非聚合函数列与GROUP BY字段不一致时，分两种情况：

     1）GROUP BY列表包含主键或唯一非空键时，SQL语句不会报错

     2）SELECT列表中非聚合函数列，都出现GROUP BY列表或WHERE列表中。且WHERE子句中的列需要等于某一常量的形式，不会报错
  3. 对于同时含有join子句和GROUP BY子句时，

     1）LEFT JOIN

     语法的ON子句中，若条件为两列等值判断，如t1.col1=t2.col2 ...，则GROUP BY子句中可无右连接表列，不报错。如t1 LEFT JOIN t2，GROUP BY 子句只需包含t1.col1

     2）RIGHT JOIN

     语法的ON子句中，若条件为两列等值判断，如t1.col1=t2.col2 ...，则GROUP BY子句中可无左连接表列，不报错。如t1 RIGHT JOIN t2，GROUP BY 子句只需包含t2.col2

     3）INNER JOIN/CROSS JOIN/STRAIGHT_JOIN

     语法的ON子句中，若条件为两列等值判断，如t1.col1=t2.col2 ...，则GROUP BY子句中包含任一列即可，不报错。

  若sql_mode中不包含ONLY_FULL_GROUP_BY选项。无上述约束，以上报错场景均不报错。

  当查询表为空表时，带有WITH ROLLUP语句会查询出一条空行数据。

- **HAVING子句**

与GROUP BY子句配合用来选择特殊的组。HAVING子句将组的一些属性与一个常数值比较，只有满足HAVING子句中的逻辑表达式的组才会被提取出来。HAVING必须且只能引用GROUP BY子句中的列或聚合函数中使用的列。

- **WINDOW子句**

  一般形式为WINDOW window_name AS（window_definition）[, ...]，window_name是可以被随后的窗口定义所引用的名称，window_definition可以是以下的形式：

  ```
  [ existing_window_name ]
  [ PARTITION BY expression [, ...] ]
  [ ORDER BY expression [ ASC | DESC ] [ NULLS { FIRST | LAST } ] [, ...] ]
  [ frame_clause ]
  ```

  frame_clause为窗函数定义一个窗口框架window frame，窗函数（并非所有）依赖于框架，window frame是当前查询行的一组相关行。frame_clause可以是以下的形式：

  ```
  [ RANGE | ROWS ] frame_start
  [ RANGE | ROWS ] BETWEEN frame_start AND frame_end
  ```

  frame_start和frame_end形式为：

  ```
  UNBOUNDED PRECEDING
  VALUE PRECEDING
  CURRENT ROW
  VALUE FOLLOWING
  UNBOUNDED FOLLOWING
  ```

  说明

  针对使用到VALUE场景存在如下约束：

  - VALUE的值一定为非负数。
  - VALUE PRECEDING/FOLLOWING仅支持与ROWS连用，不支持与RANGE连用。

- **UNION子句**

  UNION计算多个SELECT语句返回行集合的并集。

  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 SHARE不能在UNION的结果或输入中声明。
  - 通过GUC参数enable_union_all_order控制UNION ALL子查询是否保序：
    - 开启方式：set enable_union_all_order to on。
    - 关闭方式：set enable_union_all_order to off。
    - 查询方式：show enable_union_all_order。

    在开启enable_union_all_order参数时，在UNION声明了ALL并且主查询未排序的情况下，支持子查询保序。其他情况均不支持子查询保序。

  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 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。

- **EXCEPT子句**

从第一个查询结果中去除两个查询结果的交集。

EXCEPT子句有以下约束条件:

- 仅在声明了ALL子句情况下，查询的结果不进行去重操作；缺省的EXCEPT会对查询结果进行去重操作。
- 同一个SELECT语句中，如果包含多个EXCEPT操作符，运算逻辑从左向右计算。仅当在语句中使用圆括号进行标识后，运算逻辑会根据优先级进行查询。
- EXCEPT的查询结果或输入中，不能指定FOR UPDATE或FOR SHARE操作。

一般表达式：

```
select_statement EXCEPT [ALL | DISTINCT] select_statement
```

- select_statement可以是任何没有ORDER BY、LIMIT、FOR UPDATE或FOR SHARE子句的SELECT语句。
- 当select_statement中包含ORDER BY或LIMIT子句时，需在select_statement前后添加圆括号。

- **ORDER BY子句**

  对SELECT语句检索得到的数据进行升序或降序排序。对于ORDER BY表达式中包含多列的情况：

  - 首先根据最左边的列进行排序，如果这一列的值相同，则根据下一个表达式进行比较直至比较完成。
  - 如果对于所有声明的表达式都相同，则按随机顺序返回。
  - 在与DISTINCT关键字一起使用的情况下，ORDER BY中排序的列必须包括在SELECT语句所检索的结果集的列中。
  - 在与GROUP BY子句一起使用的情况下，ORDER BY中排序的列必须包括在GROUP BY子句所分组的列中。

  对于不与GROUP BY子句一起使用，且在SELECT结果集中包含聚合函数时，ORDER BY只对 返回集合的函数 所在的列生效：

  ```
  m_db=# CREATE TABLE t1(c1 int, c2 int);
  m_db=# EXPLAIN SELECT sum(c1) c FROM t1 ORDER BY c, c2; --忽略orderby。
  m_db=# EXPLAIN SELECT sum(c1) c, generate_series(1,5) s FROM t1 ORDER BY c, c2; --忽略orderby。
  m_db=# EXPLAIN SELECT sum(c1) c, generate_series(1,5) s FROM t1 ORDER BY c, s; --结果对s列进行排序。
  ```

  - **NULLS FIRST**

    指定空值在排序中排在非空值之前，当指定DESC排序时，本选项为默认的。

  - **NULLS LAST**

    指定空值在排序中排在非空值之后，未指定DESC排序时，本选项为默认的。

须知

如果要支持中文拼音排序，需要在初始化M-Compatibility数据库时指定gb18030、GBK字符集，详见：**库级字符集和字符序**；并在比较时指定gb18030_chinese_ci, gbk_chinese_ci字符序。

- **LIMIT子句**

  LIMIT子句由两个独立的子句组成：

  LIMIT { count | ALL } 限制返回行数，count为指定行数，LIMIT ALL的效果和省略LIMIT子句一样。

LIMIT start, count声明返回的最大行数，而start声明开始返回行之前忽略的行数。如果两个都指定了，会在开始计算count个返回行之前先跳过start行。

LIMIT子句不支持ROWNUM作为count或者offset。

- **FETCH {FIRST | NEXT} [(expr)] {ROW| ROWS} ONLY**

  如果不指定count，默认值为1，FETCH子句限定返回查询结果从第一行开始的总行数。

  说明

  该属性在数据库M-Compatibility模式兼容控制开关s1及以上版本时不支持（如m_format_dev_version = 's1'）。

- **OFFSET子句**

  SQL：2008开始提出一种不同的语法：

  OFFSET start { ROW | ROWS }

  start声明开始返回行之前忽略的行数。

- **锁定子句**

  FOR UPDATE子句将对SELECT检索出来的行进行加锁。这样避免它们在当前事务结束前被其他事务修改或者删除，即其他企图UPDATE、 DELETE、 SELECT FOR UPDATE、SELECT FOR SHARE这些行的事务将被阻塞，直到当前事务结束。任何在一行上的DELETE命令也会获得FOR UPDATE锁模式，在主键列上修改值的UPDATE也会获得该锁模式。反过来，SELECT FOR UPDATE将等待已经在相同行上运行以上这些命令的并发事务，并且接着锁定并且返回被更新的行（或者没有行，因为行可能已被删除）。

  FOR SHARE的行为类似，只是它在每个检索出来的行上要求一个共享锁，而不是一个排他锁。一个共享锁阻塞其它事务执行UPDATE、DELETE、SELECT FOR UPDATE，不阻塞SELECT FOR SHARE。

  为了避免操作等待其他事务提交，可使用NOWAIT选项，如果被选择的行不能立即被锁住，将会立即汇报一个错误，而不是等待；WAIT n选项，如果被选择的行不能立即被锁住，等待n秒（其中，n为int类型，取值范围：0 <= n <= 2147483），n秒内获取锁则正常执行，否则报错；SKIP LOCKED选项，对表加锁时跳过已经加锁的行，SKIP LOCKED只能跳过行锁，对不同事务中锁与锁不互相阻塞的场景（如SELECT FOR SHARE - SELECT FOR SHARE, 指定SKIP LOCKED时）不会跳过锁。

  如果在锁定子句中明确指定了表名称，则只有这些指定的表被锁定，其他在SELECT中使用的表将不会被锁定。否则，将锁定该命令中所有使用的表。

  如果锁定子句应用于一个视图或者子查询，它同样将锁定所有该视图或子查询中使用到的表。

  多个锁定子句可以用于为不同的表指定不同的锁定模式。

  如果一个表中同时出现（或隐含同时出现）在多个子句中，则按照最强的锁处理。类似的，如果影响一个表的任意子句中出现了NOWAIT，该表将按照NOWAIT处理。

须知

- 对ustore表的查询只支持FOR SHARE/FOR UPDATE/FOR READ ONLY/LOCK IN SHARE MODE。
- 对于子查询是stream计划的FOR UPDATE/SHARE语句，不支持加锁的同一行被并发更新。
- SELECT FOR UPDATE、SELECT FOR SHARE不支持与UNION/EXCEPT/DISTINCT/GROUP BY/HAVING一起使用。

- **PARTITION子句**

  查询某个分区表中相应分区的数据。

简单查询

简单查询指从一个或多个表或视图中检索一个或多个列数据的操作。

```
--建表并插入数据。
m_db=# CREATE TABLE student(
    sid INT PRIMARY KEY,
    class INT,
    name  VARCHAR(50),
    gender INT CHECK(gender = 0 OR gender = 1) --性别，1为男，0为女
);
m_db=# INSERT INTO student (sid, class, name, gender) VALUES (1, 1, 'Michael', 0);
m_db=# INSERT INTO student (sid, class, name, gender) VALUES (2, 2, 'Bob', 1);
m_db=# INSERT INTO student (sid, class, name, gender) VALUES (3, 2, 'Gary', 0);

--查询部分列。
m_db=# SELECT sid,name FROM student;
 sid |  name
-----+---------
   1 | michael
   2 | bob
   3 | Gary
(3 rows)

--查询所有列。
m_db=# SELECT * FROM student;
 sid | class |  name   | gender
-----+-------+---------+-----
   1 |     1 | michael |   0
   2 |     2 | bob     |   1
   3 |     2 | Gary    |   0
(3 rows)

--给列取别名。
m_db=# SELECT sid student_id, name FROM student;
 student_id |  name
------------+---------
          1 | michael
          2 | bob
          3 | Gary
(3 rows)

--删除。
m_db=# DROP TABLE student;
```

条件查询

条件查询指查询出符合条件的数据。

```
-- 建表并插入数据
m_db=# CREATE TABLE test_grade(
```



```
    id INT,
    name VARCHAR(20),
    score FLOAT       -- 成绩
);
m_db=# INSERT INTO test_grade VALUES (1,'Scott',90),(2,'Jack',87.5),(3,'Ben',48);

-- 查询出成绩不及格的学生信息
m_db=# SELECT * FROM test_grade WHERE score < 60;
 id | name | score
----+------+-------
  3 | Ben  |    48
(1 row)

-- 删除
m_db=# DROP TABLE test_grade;
```

说明

条件查询中允许使用列别名。

分组查询

分组查询通常用于配合聚合函数，查询分类统计的信息。常见的聚合函数有总行数count()、求和sum()、平均值avg()、最小值min()、最大值max()。一般和GROUP BY联合使用。

若想对分组的查询结果进行统计，则可以将WITH ROLLUP添加到GROUP BY语句中，WITH ROLLUP可以对多列分组查询结果进行统计。

```
-- 建表并插入数据。
m_db=# CREATE TABLE student (id INT,name VARCHAR(20),class INT);
m_db=# INSERT INTO student VALUES (1, 'Scott', 1), (2, 'Ben', 1);
m_db=# INSERT INTO student VALUES (3, 'Jack', 2),(4, 'Anna', 2),(5, 'Judy', 2);
m_db=# INSERT INTO student VALUES (6, 'Sally', 3), (7, 'Jordan', 3);

--使用聚合函数COUNT()统计每个班级有多少人。
m_db=# SELECT class, count(*) AS cnt FROM student GROUP BY class;
 class | cnt
-------+-----
     2 |   3
     1 |   2
     3 |   2
(3 rows)

--使用WITH ROLLUP语法对按班级分组统计后的人数进行汇总，汇总数据在最后一列。
m_db=# SELECT class, count(*) AS cnt FROM student GROUP BY class WITH ROLLUP;
 class | cnt
-------+-----
     1 |   2
     2 |   3
     3 |   2
       |   7
(4 rows)

--使用WITH ROLLUP语法对按班级,姓名分组统计后的人数进行汇总，汇总数据在最后一列。
解释输出：对每个班级的姓名分组的后面，产生一个额外的统计行，意为每个班的人数。最后在其他行后面，产生一个额外的统计行，意为所有班的总人数。
m_db=# SELECT class,name, count(*) AS cnt FROM student GROUP BY class,name WITH ROLLUP;
 class |  name  | cnt
-------+--------+-----
     1 | Ben    |   1
     1 | Scott  |   1
     1 |        |   2
     2 | Anna   |   1
     2 | Jack   |   1
     2 | Judy   |   1
     2 |        |   3
     3 | Jordan |   1
```



```
    3 | Sally  |   1
    3 |        |   2
      |        |   7
(11 rows)

--使用HAVING子句过滤人数少于3个的班级数据。
m_db=# SELECT class, COUNT(*) AS num FROM student GROUP BY class HAVING COUNT(*) < 3;
 class | num
-------+-----
    1 |   2
    3 |   2
(2 rows)

--删除。
m_db=# DROP TABLE student;
```

sql_mode中包含ONLY_FULL_GROUP_BY选项，用于校验GROUP BY 列表与SELECT列表。

```
-- 建表并插入数据。
m_db=# CREATE TABLE test (
          id int  primary key NOT NULL,
          id_2 int NOT NULL,
          s1 varchar(10),
          s2 int,
          s3 varchar(12)
        );
m_db=# ALTER TABLE test ADD CONSTRAINT unique_id_2 UNIQUE(id_2);
m_db=# INSERT INTO test value(1,101,'a',1000,'aa');
m_db=# INSERT INTO test value(2,102,'b',2000,'aaa');
m_db=# INSERT INTO test value(3,103,'c',3000,'aaaa');
m_db=# INSERT INTO test value(4,104,'b',4000,'aaa');
m_db=# INSERT INTO test value(5,105,'c',5000,'aaaaa');
m_db=# CREATE TABLE tb1(a int, b int, c int);
m_db=# INSERT INTO tb1 VALUES(1, 4, 11);
m_db=# INSERT INTO tb1 VALUES(2, 5, 12);
m_db=# INSERT INTO tb1 VALUES(2, 6, 13);
m_db=# INSERT INTO tb1 VALUES(2, 2, 13);
m_db=# CREATE TABLE tb2(a int, b int);
m_db=# INSERT INTO tb2 VALUES(2, 7);
m_db=# INSERT INTO tb2 VALUES(2, 8);
m_db=# INSERT INTO tb2 VALUES(3, 9);

-- 设置SQL MODE。
m_db=# SET SQL_MODE =
'ONLY_FULL_GROUP_BY,STRICT_TRANS_TABLES,NO_ZERO_IN_DATE,NO_ZERO_DATE,ERROR_FOR_DIVISION_
BY_ZERO,NO_AUTO_CREATE_USER,NO_ENGINE_SUBSTITUTION';

--SELECT列表中非聚合函数列与GROUP BY字段一致时，不会报错。
m_db=# SELECT s1, s3 FROM test GROUP BY s1,s3;
 s1 |  s3
----+-------
 c  | aaaaa
 b  | aaa
 c  | aaaa
 a  | aa
(4 rows)

m_db=# SELECT s1, s3 FROM test GROUP BY s3;
ERROR:  Column test.s1 must appear in the GROUP BY clause or be used in an aggregate function.
LINE 1: SELECT s1, s3 FROM test GROUP BY s3;
               ^
--SELECT列表中非聚合函数列与GROUP BY字段不一致时，分两种情况：
--1）GROUP BY列表包含主键或唯一非空键时，SQL语句不会报错。
m_db=# SELECT s1, MAX(LENGTH(s3)) FROM test GROUP BY id;
 s1 | max
----+-----
 c  |   4
 c  |   5
 a  |   2
```



```
 b |  3
 b |  3
(5 rows)

--2）SELECT列表中非聚合函数列，都出现GROUP BY列表或WHERE列表中。且WHERE子句中的列需要等于某一常量。
m_db=# SELECT s1, SUM(s2) FROM test WHERE s1 = 'b' GROUP BY s1;
 s1 | sum
----+------
 b  | 6000
(1 row)

--对于同时含有JOIN子句和GROUP BY子句时：
--1）LEFT JOIN语法的ON子句中，若条件为两列等值判断，如t1.col1=t2.col2 ...，则GROUP BY子句中可无右连接表列，不报错。如t1 LEFT JOIN t2，GROUP BY 子句只需包含t1.col1。
m_db=# SELECT tb1.a, tb2.a FROM tb1 LEFT JOIN tb2 ON tb1.a = tb2.a GROUP BY tb1.a;
 a | a
---+---
 1 |
 2 | 2
(2 rows)

m_db=# SELECT tb1.a, tb2.a FROM tb1 LEFT JOIN tb2 ON tb1.a = tb2.a GROUP BY tb2.a;
ERROR:  Column tb1.a must appear in the GROUP BY clause or be used in an aggregate function.
LINE 1: SELECT tb1.a, tb2.a FROM tb1 LEFT JOIN tb2 ON tb1.a = tb2.a ...
               ^
--2）RIGHT JOIN语法的ON子句中，若条件为两列等值判断，如t1.col1=t2.col2 ...，则GROUP BY子句中可无左连接表列，不报错。如t1 RIGHT JOIN t2，GROUP BY 子句只需包含t2.col2。
m_db=# SELECT tb1.a, tb2.a FROM tb1 RIGHT JOIN tb2 ON tb1.a = tb2.a GROUP BY tb2.a;
 a | a
---+---
   | 3
 2 | 2
(2 rows)

m_db=# SELECT tb1.a, tb2.a FROM tb1 RIGHT JOIN tb2 ON tb1.a = tb2.a GROUP BY tb1.a;
ERROR:  Column tb2.a must appear in the GROUP BY clause or be used in an aggregate function.
LINE 1: SELECT tb1.a, tb2.a FROM tb1 RIGHT JOIN tb2 ON tb1.a = tb2.a...
                     ^
--3）INNER JOIN/CROSS JOIN/STRAIGHT_JOIN语法的ON子句中，若条件为两列等值判断，如t1.col1=t2.col2 ...，则GROUP BY子句中包含任一列即可，不报错。
m_db=# SELECT tb1.a, tb2.a FROM tb1 INNER JOIN tb2 ON tb1.a = tb2.a GROUP BY tb1.a;
 a | a
---+---
 2 | 2
(1 row)

m_db=# SELECT tb1.a, tb2.a FROM tb1 INNER JOIN tb2 ON tb1.a = tb2.a GROUP BY tb2.a;
 a | a
---+---
 2 | 2
(1 row)

--删除。
m_db=# DROP TABLE test, tb1, tb2;
```

分页查询

语法如下：

```
SELECT query_list FROM table_name [ LIMIT { [offset,] count | ALL } ]
```

- offset ： 表示从第几行向后开始。
- count：表示向后查询几条数据。
- ALL：表示向后查询所有的数据。

```
-- 建表并插入100条数据。
m_db=# CREATE TABLE testl(id int PRIMARY KEY, flag varchar(10));
```



```
m_db=# INSERT INTO testl (id, flag) VALUES (generate_series(1,100),generate_series(1,100));

-- 查询前5条数据。
m_db=# SELECT * FROM testl ORDER BY 1  LIMIT 5;
 id | flag
----+-------
  1 | 1
  2 | 2
  3 | 3
  4 | 4
  5 | 5
(5 rows)

-- 从第20条向后查询4条数据。
m_db=# SELECT * FROM testl ORDER BY 1 LIMIT 20,4;
 id |  flag
----+--------
 21 | 21
 22 | 22
 23 | 23
 24 | 24
(4 rows)

-- 从第96条向后查询出所有数据。
m_db=# SELECT * FROM testl ORDER BY 1 LIMIT 96,ALL;
 id  |  flag
-----+---------
  97 | 97
  98 | 98
  99 | 99
 100 | 100
(4 rows)

-- 删除。
m_db=# DROP TABLE testl;
```

分区查询

查询指定分区的数据。

```
-- 创建范围分区表
m_db=# CREATE TABLE test_range1(
    id INT,
    info VARCHAR(20)
) PARTITION BY RANGE (id) (
    PARTITION p1 VALUES LESS THAN (200),
    PARTITION p2 VALUES LESS THAN (400),
    PARTITION p3 VALUES LESS THAN (600),
    PARTITION p4 VALUES LESS THAN (800),
    PARTITION pmax VALUES LESS THAN (MAXVALUE)
);
-- 插入1000数据
m_db=# INSERT INTO test_range1 VALUES(GENERATE_SERIES(1,1000),'abcd');

-- 查询p1分区有多少条数据
m_db=# SELECT COUNT(*) FROM test_range1 PARTITION (p1);
 count
-------
   199
(1 row)

-- 删除。
m_db=# DROP TABLE test_range1;
```

连接查询

连接查询也可称为跨表查询，需要关联多个表进行查询。


```
-- 建表并插入数据。
m_db=# CREATE TABLE emp(
    id int,             --员工编号
    name varchar(20),   --员工姓名
    deptno int          --所属部门编号
);

m_db=# CREATE TABLE dept(
    deptno int,         --部门编号
    depname varchar(20) --部门名
);

m_db=# INSERT INTO emp VALUES (1, 'Joe', 10), (2, 'Scott', 20), (3, 'Ben', 999);   --Ben还没有确认是哪一个部门所以部门编号是999
m_db=# INSERT INTO dept VALUES (10, 'hr'), (20, 'it'), (30, 'sal');                 --sal部门没有员工
```

- 内连接（INNER JOIN）

  ```
  m_db=# SELECT t1.id,t1.name,t2.depname FROM emp t1 JOIN dept t2 ON t1.deptno = t2.deptno;
   id | name  | depname
  ----+-------+---------
    1 | Joe   | hr
    2 | Scott | it
  (2 rows)
  ```

- 左连接（LEFT JOIN）

  ```
  m_db=# SELECT t1.id,t1.name,t2.depname FROM emp t1 LEFT JOIN dept t2 ON t1.deptno = t2.deptno;
   id | name  | depname
  ----+-------+---------
    1 | Joe   | hr
    2 | Scott | it
    3 | Ben   |
  (3 rows)
  ```

- 右连接（RIGHT JOIN）

  ```
  m_db=# SELECT t1.id,t1.name,t2.depname FROM emp t1 RIGHT JOIN dept t2 ON t1.deptno =
  t2.deptno;
   id | name  | depname
  ----+-------+---------
    1 | Joe   | hr
    2 | Scott | it
      |       | sal
  (3 rows)
  ```

子查询

- 按照子查询的返回记录数分类可分为单行子查询，多行子查询。
- 按照子查询是否被执行多次分类可分为关联子查询，非关联子查询。

  一个查询可以嵌套在另一个查询中，其结果作为另一个查询的数据来源或判断条件。其中外层查询也叫父查询，内层查询也叫子查询。
- 子查询允许不添加别名。

说明

M-Compatibility模式数据库不支持子查询结果包含多列，当包含多列时执行会报错。

```
m_db=# SELECT row(1,2) = (SELECT 1,2);
ERROR:  subquery must return only one column
LINE 1: SELECT row(1,2) = (SELECT 1,2);
                          ^
```

单行子查询

单行子查询操作符有>=、>、<=、<、<>。

```
-- 创建学生表并插入数据。
m_db=# CREATE TABLE student(
```



```
    sid VARCHAR(5),     -- 学号
    grade INT,          -- 年级
    name VARCHAR(20),   -- 姓名
    height INT          -- 身高
);
m_db=# INSERT INTO student VALUES ('00001',1,'Scott',135),('00002',1,'Jack',95),('00003',1,'Ben',100);
m_db=# INSERT INTO student VALUES ('00004',2,'Henry',115),('00005',2,'Jordan',130),('00006',2,'Bob',126);
m_db=# INSERT INTO student VALUES ('00007',3,'Bella',128),('00008',3,'Alicia',136);
-- 创建老师表并插入数据。
m_db=# CREATE TABLE teacher (
    name VARCHAR(20),    -- 教师姓名
    grade INT            -- 班级
);
-- 插入数据
m_db=# INSERT INTO teacher VALUES ('Bill',1),('Sally',2),('Luke',3);

-- 查询出身高比Bella高的学生。
m_db=# SELECT * FROM student
WHERE height > (SELECT height FROM student WHERE name = 'Bella');
  sid  | grade |  name  | height
-------+-------+--------+--------
 00001 |     1 | Scott  |    135
 00005 |     2 | Jordan |    130
 00008 |     3 | Alicia |    136
(3 rows)
```

多行子查询

多行子查询操作符：

- IN：等于列表中的任意一个。
- ANY：需要和单行比较符一起使用，和子查询返回的任意值比较。
- ALL：需要和单行比较符一起使用，和子查询返回的所有值比较。
- SOME：ANY的别名，作用相同。

示例：查询出Sally和Luke的学生

```
m_db=# SELECT * FROM student t1 WHERE t1.grade IN (
    SELECT grade FROM teacher WHERE name = 'Sally' OR name = 'Luke'
);
  sid  | grade |  name  | height
-------+-------+--------+--------
 00004 |     2 | Henry  |    115
 00005 |     2 | Jordan |    130
 00006 |     2 | Bob    |    126
 00007 |     3 | Bella  |    128
 00008 |     3 | Alicia |    136
(5 rows)
```

示例：查询出2年级比3年级任意一个人都高的学生。

```
m_db=# SELECT * FROM student
WHERE grade = 2 AND
      height > ANY (SELECT height FROM student WHERE grade = 3);
  sid  | grade |  name  | height
-------+-------+--------+--------
 00005 |     2 | Jordan |    130
(1 row)

--查询结果相当于：
m_db=# SELECT * FROM student
WHERE grade = 2 AND
      height > (SELECT MIN(height) FROM student WHERE grade = 3);
```

示例：查询出1年级比2年级所有人都高的学生。


```
m_db=# SELECT * FROM student
WHERE grade = 1 AND
      height > ALL (SELECT height FROM student WHERE grade = 2);
 sid  | grade | name  | height
-------+-------+-------+--------
 00001 |     1 | Scott |    135
(1 row)

--查询结果相当于：
m_db=# SELECT * FROM student
WHERE grade = 1 AND
      height > (SELECT MAX(height) FROM student WHERE grade = 2);
```

关联子查询

特点：子查询不能单独运行，是和父查询相关的。先执行父查询，再执行子查询。每执行一次父查询，子查询都要重新计算一次。

示例：查询身高大于本班级平均身高的学生。

```
m_db=# SELECT * FROM student out
WHERE height > (SELECT AVG(height) FROM student
               WHERE grade = out.grade);
 sid  | grade |  name  | height
-------+-------+--------+--------
 00001 |     1 | Scott  |    135
 00005 |     2 | Jordan |    130
 00006 |     2 | Bob    |    126
 00008 |     3 | Alicia |    136
(4 rows)
```

非关联子查询

特点：子查询先将值查询出来，再返回给外层查询。

示例：查询身高大于本班级平均身高的学生。

```
m_db=# SELECT t1.*
FROM student t1,
    (SELECT grade, AVG(height) avg_hei FROM student GROUP BY grade) t2
WHERE t1.grade = t2.grade AND
      t1.height > t2.avg_hei;
 sid  | grade |  name  | height
-------+-------+--------+--------
 00001 |     1 | Scott  |    135
 00005 |     2 | Jordan |    130
 00006 |     2 | Bob    |    126
 00008 |     3 | Alicia |    136
(4 rows)
-- 删除
m_db=# DROP TABLE student;
m_db=# DROP TABLE teacher;
```

复合查询

包含复合运算符的查询，即复合查询。所有的复合查询都具有相同的优先级，参加集合操作的各查询结果的列数、表达式的数量都必须一致，类型必须兼容。

UNION

常见的集合运算有：

- UNION：两个查询结果集的并集，对结果进行去重。

- UNION ALL：两个查询的并集，只将两个查询的结果合并。

UNION 示例

```
-- 建表并插入数据
m_db=# CREATE TABLE test1(c11 INT, c12 VARCHAR(20));
m_db=# INSERT INTO test1 VALUES (1,'a'),(2,'b'),(4,'d');

m_db=# CREATE TABLE test2(c21 INT, c22 VARCHAR(20));
m_db=# INSERT INTO test2 VALUES (1,'a'),(3,'c');
```

- UNION

```
m_db=# SELECT * FROM test1 UNION SELECT * FROM test2;
 c11 | c12
-----+-----
   4 | d
   3 | c
   1 | a
   2 | b
(4 rows)
```

- UNION ALL

```
m_db=# SELECT * FROM test1 UNION ALL SELECT * FROM test2;
 c11 | c12
-----+-----
   1 | a
   2 | b
   4 | d
   1 | a
   3 | c
(5 rows)
```

EXCEPT

常见的交集运算有：

- EXCEPT：在第一个查询结果中去除两个查询结果的交集，并对结果进行去重。
- EXCEPT ALL：在第一个查询结果中去除两个查询结果的交集，不进行去重。
- EXCEPT DISTINCT：在第一个查询结果中去除两个查询结果的交集，并对结果进行去重。

EXCEPT 示例

```
-- 建表并插入数据
m_db=# CREATE TABLE a(m int,n int);
m_db=# INSERT INTO a(m,n) VALUES(1,2);
m_db=# INSERT INTO a(m,n) VALUES(2,3);
m_db=# INSERT INTO a(m,n) VALUES(3,4);

m_db=# CREATE TABLE b(m int,n int);
m_db=# INSERT INTO b(m,n) VALUES(1,2);
m_db=# INSERT INTO b(m,n) VALUES(1,3);
m_db=# INSERT INTO b(m,n) VALUES(3,4);

m_db=# CREATE TABLE c(m int,n int);
m_db=# INSERT INTO c(m,n) VALUES(1,3);
m_db=# INSERT INTO c(m,n) VALUES(1,3);
m_db=# INSERT INTO c(m,n) VALUES(3,4);
```

- EXCEPT

```
m_db=# SELECT * FROM a EXCEPT SELECT * FROM b;
 m | n
---+---
 2 | 3
(1 row)
```



```
m_db=# SELECT * FROM a EXCEPT SELECT * FROM c;
 m | n
---+---
 2 | 3
 1 | 2
(2 rows)

m_db=# SELECT * FROM b EXCEPT SELECT * FROM a;
 m | n
---+---
 1 | 3
(1 row)
-- 对查询结果进行去重。
m_db=# SELECT * FROM c EXCEPT SELECT * FROM a;
 m | n
---+---
 1 | 3
(1 row)
```

- EXCEPT ALL

```
m_db=# SELECT * FROM a EXCEPT ALL SELECT * FROM b;
 m | n
---+---
 2 | 3
(1 row)

m_db=# SELECT * FROM a EXCEPT ALL SELECT * FROM c;
 m | n
---+---
 2 | 3
 1 | 2
(2 rows)

m_db=# SELECT * FROM b EXCEPT ALL SELECT * FROM a;
 m | n
---+---
 1 | 3
(1 row)

m_db=# SELECT * FROM c EXCEPT ALL SELECT * FROM a;
 m | n
---+---
 1 | 3
 1 | 3
(2 rows)
```

- EXCEPT DISTINCT

```
m_db=# SELECT * FROM a EXCEPT DISTINCT SELECT * FROM b;
 m | n
---+---
 2 | 3
(1 row)

m_db=# SELECT * FROM a EXCEPT DISTINCT SELECT * FROM c;
 m | n
---+---
 2 | 3
 1 | 2
(2 rows)

m_db=# SELECT * FROM b EXCEPT DISTINCT SELECT * FROM a;
 m | n
---+---
 1 | 3
(1 row)
-- 对查询结果进行去重。
m_db=# SELECT * FROM c EXCEPT DISTINCT SELECT * FROM a;
 m | n
---+---
 1 | 3
```



```
(1 row)

m_db=# DROP TABLE a,b,c;
```

4.4.2.16.3 SELECT INTO

功能描述

SELECT INTO用于根据查询结果创建一个新表，并且将查询到的数据插入到新表中。

数据并不返回给客户端，这一点和普通的SELECT不同。新表的字段具有和SELECT的输出字段相同的名称和数据类型。

语法格式

```
[ WITH [ RECURSIVE ] with_query [, ...] ]
SELECT
    [ALL | DISTINCT | DISTINCTROW]
    [SQL_NO_CACHE]
    select_expr [, select_expr] ...
    INTO [ [ GLOBAL | LOCAL ] [ TEMPORARY ]  ] [ TABLE ] new_table
    [FROM from_item]
    [WHERE where_condition]
    [GROUP BY {col_name | expr | position}]
    [HAVING condition]
    [ORDER BY {col_name | expression | position}
      [ASC | DESC], ...]
    [LIMIT {[offset,] row_count | row_count OFFSET offset}]
    INTO [ [ GLOBAL | LOCAL ] [ TEMPORARY ]  ] [ TABLE ] new_table
    [ FOR READ ONLY |
        [ {FOR { UPDATE | SHARE | } [ OF table_name [, ...] ] [ NOWAIT | WAIT N ]} [...] ] |
        LOCK IN SHARE MODE];
```

参数说明

- **new_table**

  new_table指定新建表的名称。

- **GLOBAL | LOCAL**

  创建临时表时可以在TEMP或TEMPORARY前指定GLOBAL或LOCAL关键字。如果指定GLOBAL关键字，M-Compatibility会创建全局临时表，否则M-Compatibility会创建本地临时表。

- **TEMPORARY**

  如果指定TEMPORARY关键字，则创建的表为临时表。临时表分为全局临时表和本地临时表两种类型。创建临时表时如果指定GLOBAL关键字则为全局临时表，否则为本地临时表。

  全局临时表的元数据对所有会话可见，会话结束后元数据继续存在。会话与会话之间的用户数据、索引和统计信息相互隔离，每个会话只能看到和更改自己提交的数据。全局临时表有两种模式：一种是基于会话级别的(ON COMMIT PRESERVE ROWS), 当会话结束时自动清空用户数据；一种是基于事务级别的(ON COMMIT DELETE ROWS), 当执行commit或rollback时自动清空用户数据。建表时如果没有指定ON COMMIT选项，则缺省为会话级别。与本地临时表不同，全局临时表建表时可以指定非pg_temp_开头的schema。

  本地临时表只在当前会话可见，本会话结束后会自动删除。因此，在除当前会话连接的数据库节点故障时，仍然可以在当前会话上创建和使用临时表。由于临时表只在当前会话创建，对于涉及对临时表操作的DDL语句，会产生DDL失败的报错。因此，建议DDL语句中不要对临时表进行操作。TEMP和TEMPORARY等价。

须知

- 本地临时表通过每个会话独立的以pg_temp开头的schema来保证只对当前会话可见，因此，不建议用户在日常操作中手动删除以pg_temp、pg_toast_temp开头的schema。
- 如果建表时不指定TEMPORARY/TEMP关键字，而指定表的schema为当前会话的pg_temp_开头的schema，则此表会被创建为临时表。
- ALTER/DROP全局临时表和索引，如果其它会话正在使用它，禁止操作。
- 全局临时表的DDL只会影响当前会话的用户数据和索引。例如truncate、reindex、analyze只对当前会话有效。

说明

SELECT INTO的其它参数可参考SELECT的**参数说明**。

示例

```
--创建SCHEMA。
m_db=# CREATE SCHEMA tpcds;

--创建表tpcds.reason。
m_db=# CREATE TABLE tpcds.reason
(
  r_reason_sk       integer,
  r_reason_id       character(16),
  r_reason_desc     character(100)
);

--向表中插入多条记录。
m_db=# INSERT INTO tpcds.reason values(1,'AAAAAAAABAAAAAAA','reason 1'),
(2,'AAAAAAAABAAAAAAA','reason 2'),(3,'AAAAAAAABAAAAAAA','reason 3'),
(4,'AAAAAAAABAAAAAAA','reason 4'),(4,'AAAAAAAABAAAAAAA','reason 5'),
(4,'AAAAAAAACAAAAAAA','reason 6'),(5,'AAAAAAAACAAAAAAA','reason 7');

--将tpcds.reason表中r_reason_sk小于5的值加入到新建表中。
m_db=# SELECT * INTO tpcds.reason_t1 FROM tpcds.reason WHERE r_reason_sk < 5;
INSERT 0 6

--删除tpcds.reason_t1表。
m_db=# DROP TABLE tpcds.reason_t1;

--删除表。
m_db=# DROP TABLE tpcds.reason;

--删除SCHEMA。
m_db=# DROP SCHEMA tpcds;
```

相关链接

SELECT

4.4.2.16.4 SET

功能描述

用于修改运行时配置参数。

注意事项

大多数运行时参数都可以用SET在运行时设置，但有些则在服务运行过程中或会话开始之后不能修改。

语法格式

- 设置所处的时区。
```
SET [ SESSION | LOCAL ] TIME ZONE { timezone | LOCAL | DEFAULT };
```
- 设置所属的模式。
```
SET [ SESSION | LOCAL ]
  {CURRENT_SCHEMA { TO | = } { schema | DEFAULT }
  | SCHEMA 'schema'};
```
- 设置客户端编码集。
```
SET [ SESSION | LOCAL ] NAMES {'charset_name' [COLLATE 'collation_name'] | DEFAULT};
```
- 设置自定义用户变量。
```
SET @var_name := expr [, @var_name := expr] ...
SET @var_name = expr [, @var_name = expr] ...
```
- 设置其他运行时的参数。
```
SET [ SESSION | @@SESSION. | @@ | LOCAL | @@LOCAL.]
    {config_parameter { TO | = } { expr | DEFAULT } | FROM CURRENT };
```
- 设置字符集。同时设置character_set_connection、character_set_results、character_set_client的值，把character_set_client、character_set_results设置成指定charset，把character_set_connection设置成character_set_database的值。DEFAULT表示把character_set_client、character_set_results设置为默认值，character_set_connection设置成character_set_database的值。
```
SET {CHARSET | CHARACTER SET} {DEFAULT | charset_name}
```

参数说明

- **SESSION**

  声明的参数只对当前会话起作用。如果SESSION和LOCAL都没出现，则SESSION为缺省值。

  如果在事务中执行了此命令，命令的产生影响将在事务回滚之后消失。如果该事务已提交，影响将持续到会话的结束，除非被另外一个SET命令重置参数。

- **LOCAL**

  声明的参数只在当前事务中有效。在COMMIT或ROLLBACK之后，会话级别的设置将再次生效。

  不论事务是否提交，此命令的影响只持续到当前事务结束。一个特例是：在一个事务里面，既有SET命令，又有SET LOCAL命令，且SET LOCAL在SET后面，则在事务结束之前，SET LOCAL命令会起作用，但事务提交之后，则是SET命令会生效。

- **TIME ZONE**

  用于指定当前会话的本地时区。

  取值范围：有效的本地时区。该选项对应的运行时参数名称为TimeZone，DEFAULT缺省值为PRC。

- **CURRENT_SCHEMA**

  CURRENT_SCHEMA用于指定当前的模式。

  取值范围：已存在模式名称。如果模式名不存在，会导致CURRENT_SCHEMA值为空。

- **SCHEMA**

  同CURRENT_SCHEMA。此处的schema是个字符串。

  例如：set schema 'public';

- **NAMES {'charset_name' [COLLATE 'collation_name'] | DEFAULT};**

  用于设置客户端的字符编码。

  等价于：

  ```
  set client_encoding = charset_name; set character_set_connection = charset_name;
  set collation_connection = collation_name; set character_set_results = charset_name;
  ```

  取值范围：M-compatibility兼容模式下支持的字符集，字符序。暂不支持指定charset_name与数据库字符集不同。

- **config_parameter**

  可设置的运行时参数的名称。可用的运行时参数可以使用SHOW ALL命令查看。

  说明

  部分通过SHOW ALL查看的参数不能通过SET设置。如max_datanodes。

- **value**

  config_parameter的新值。可以声明为字符串常量、标识符、数字，或者逗号分隔的列表。DEFAULT用于把这些参数设置为它们的缺省值。

- **SESSION | @@SESSION. | @@| LOCAL | @@LOCAL.**

  声明的参数生效方式为superuser、user，可通过pg_settings系统视图的context字段确定，如果没有出现GLOBAL /SESSION，则SESSION为缺省值。支持config_parameter赋值为表达式。

  说明

  1. 使用@@config_parameter进行操作符运算时，尽量使用空格隔开。比如set @@config_parameter1=@@config_parameter1*2; 命令中，会将=@@当做操作符，可将其修改为set @@config_parameter1= @@config_parameter1 * 2 。
  2. GaussDB中仅支持通过数据库运维平台下发全局参数，避免错误使用时影响其他用户或应用，带来安全隐患，因此不支持@@global语法修改参数值。

- **var_name**

  自定义变量名。变量名只能由数字、字母、下划线（_），点（.）、$组成，如果使用单引号（'）、双引号（"）、反引号（`）引用时，则可以使用其他字符，但不能使用相同引号。形如，单引号内可以使用除单引号外的其他字符。

  说明

  - 自定义变量只会存储数值类型、字符类型、二进制类型和NULL。对于布尔类型，有符号整型，bit和year类型会转为int8存储；无符号整型转为无符号int8存储；float4、float8类型转为float8存储；numeric类型转为numeric存储；二进制类型转为longblob类型；除year类型的时间类型和字符类型转为longtext存储。
  - prepare语句不支持自定义用户变量。
  - 对于连续赋值的场景，只支持@var_name1 := @var_name2 := ⋯ := expr和@var_name1 = @var_name2 := ⋯ := expr，等号只有放在首位才表示赋值，形如set @var_name1 = @var_name2 := @var_name3 = @var_name4 := expr这种等号放在中间的赋值场景不支持，因为除首位其他位置等号表示比较操作符。
  - 非SET语句中不支持对用户变量赋值，只支持使用或查询。
  - 对于无法转为用户变量支持的数据类型，则报错。

- **expr**

表达式，支持可直接或间接转为整型、浮点型、字符串、位串和NULL的表达式。

说明

- 字符串表达式中避免包含口令等敏感信息的函数，如加解密类函数gs_encrypt，gs_decrypt等，防止敏感信息泄露。
- 当expr为子查询表达式或为case when表达式且最终结果为子查询表达式时，自定义变量结果与直接查询结果一致。MySQL使用中间计算结果，自定义变量和直接查询结果可能不一致。

示例

```
--设置模式搜索路径。
m_db=# SET search_path TO tpcds, public;

--把日期时间风格设置为传统的 POSTGRES 风格(日在月前)。
m_db=# SET datestyle TO postgres,dmy;

--设置用户变量
m_db=# SET @a := 1;
```

相关链接

RESET，**SHOW**

4.4.2.16.5 SET ROLE

功能描述

设置当前会话的当前用户标识符。

注意事项

- 当前会话的用户必须是指定的rolename角色的成员，但系统管理员可以选择任何角色。
- 使用这条命令，它可能会增加一个用户的权限，也可能会限制一个用户的权限。如果会话用户的角色有INHERITS属性，则它自动拥有它能SET ROLE变成的角色的所有权限；在这种情况下，SET ROLE实际上是删除了所有直接赋予会话用户的权限，以及它的所属角色的权限，只剩下指定角色的权限。另一方面，如果会话用户的角色有NOINHERITS属性，SET ROLE删除直接赋予会话用户的权限，而获取指定角色的权限。

语法格式

- 设置当前会话的当前用户标识符。
  ```
  SET [ SESSION | LOCAL ] ROLE role_name PASSWORD 'password';
  ```
- 重置当前用户标识为当前会话用户标识符。
  ```
  SET ROLE = DEFAULT;
  ```

参数说明

- **SESSION**

  声明这个命令只对当前会话起作用，此参数为缺省值。
- **LOCAL**

  声明该命令只在当前事务中有效。

- **role_name**
  取值范围：已存在的角色名，角色名要求详见•role_name。
- **password**
  角色的密码。要求符合密码的命名规则。不支持直接使用密文密码。

示例

```
--创建角色paul。
m_db=# CREATE ROLE paul IDENTIFIED BY '********';

--设置当前用户为paul。
m_db=# SET ROLE paul PASSWORD '********';

--查看当前用户。
m_db=# SELECT CURRENT_USER;

--重置当前用户。
m_db=# SET role = DEFAULT;

--删除用户。
m_db=# DROP USER paul;
```

4.4.2.16.6 SET SESSION AUTHORIZATION

功能描述

把当前会话里的会话用户标识和当前用户标识都设置为指定的用户。

注意事项

只有在初始会话用户有系统管理员权限的时候，会话用户标识符才能改变。否则，只有在指定了被认证的用户名的情况下，系统才接受该命令。

语法格式

- 为当前会话设置会话用户标识符和当前用户标识符。
  ```
  SET [ SESSION | LOCAL ] SESSION AUTHORIZATION role_name PASSWORD 'password';
  ```
- 重置会话和当前用户标识符为初始认证的用户名。
  ```
  {SET [ SESSION | LOCAL ] SESSION AUTHORIZATION DEFAULT
  | SET SESSION_AUTHORIZATION = DEFAULT};
  ```

参数说明

- **SESSION**
  声明这个命令只对当前会话起作用。
- **LOCAL**
  声明该命令只在当前事务中有效。
- **role_name**
  用户名。
  取值范围：字符串，用户名要求详见•user_name。
- **password**
  角色的密码。要求符合密码的命名规则。不支持直接使用密文密码。
- **DEFAULT**

重置会话和当前用户标识符为初始认证的用户名。

示例

```
--创建角色paul。
m_db=# CREATE ROLE paul IDENTIFIED BY '********';

--设置当前用户为paul。
m_db=# SET SESSION AUTHORIZATION paul password '********';

--查看当前用户。
m_db=# SELECT CURRENT_USER;

--重置当前用户。
m_db=# SET SESSION_AUTHORIZATION = DEFAULT;

--删除用户。
m_db=# DROP USER paul;
```

相关参考

SET ROLE

4.4.2.16.7 SET TRANSACTION

功能描述

为事务设置特性。事务特性包括事务隔离级别、事务访问模式(读/写或者只读)。可以设置会话的默认事务特性（SESSION），也可以设置当前数据库的全局会话的事务特性（GLOBAL）。

注意事项

设置当前数据库全局会话（GLOBAL）的事务特性在重新连接后生效。开启m_format_dev_version='s2'参数时，SET TRANSACTION默认是设置下一个事务特性，不允许在事务内使用下一个事务特性；如果隐式事务报错，即单个SQL语句报错，继续保持下一个事务特性。

语法格式

设置事务的隔离级别、读写模式。

```
{ SET [ LOCAL | SESSION | GLOBAL ] TRANSACTION }
  { ISOLATION LEVEL { READ COMMITTED | READ UNCOMMITTED | SERIALIZABLE | REPEATABLE READ }
  | { READ WRITE | READ ONLY } };
```

参数说明

- **LOCAL**

  声明这个命令只对当前会话起作用。效果等价于SESSION。
- **SESSION**

  声明这个命令只对当前会话起作用。
- **GLOBAL**

  声明这个命令对当前数据库的全局会话生效。
- **ISOLATION LEVEL**

  指定事务隔离级别，该参数决定当一个事务中存在其他并发运行事务时能够看到什么数据。

说明

在事务中第一个数据修改语句（SELECT、INSERT、DELETE、UPDATE、COPY）执行之后，当前事务的隔离级别就不能再次设置。

取值范围：

- READ COMMITTED：读已提交隔离级别，只能读到已经提交的数据，而不会读到未提交的数据。这是缺省值。
- READ UNCOMMITTED：读未提交隔离级别，指定后的行为和READ COMMITTED行为一致。
- REPEATABLE READ：可重复读隔离级别，仅仅能看到事务开始之前提交的数据，不能看到未提交的数据，以及在事务执行期间由其它并发事务提交的修改。
- SERIALIZABLE：M-Compatibility目前功能上不支持此隔离级别，等价于REPEATABLE READ。

- **READ WRITE | READ ONLY**

  指定事务访问模式（读/写或者只读）。

示例

```
--开启一个事务，设置事务的隔离级别为READ COMMITTED，访问模式为READ ONLY。
m_db=# START TRANSACTION;
m_db=# SET LOCAL TRANSACTION ISOLATION LEVEL READ COMMITTED READ ONLY;
m_db=# COMMIT;

--设置当前会话的事务隔离级别、读写模式。
m_db=# SET SESSION TRANSACTION ISOLATION LEVEL READ COMMITTED;
m_db=# SET SESSION TRANSACTION READ ONLY;

--给数据库设置全局会话的事务隔离级别、读写模式。
m_db=# SET GLOBAL TRANSACTION ISOLATION LEVEL READ COMMITTED;
m_db=# SET GLOBAL TRANSACTION READ ONLY;

--开启m_format_dev_version='s2'参数，设置下一个事务隔离级别、读写模式。
m_db=# SET m_format_dev_version='s2';
m_db=# SET TRANSACTION ISOLATION LEVEL REPEATABLE READ READ ONLY;

--开启m_format_dev_version='s2'参数,不允许在事务内设置下一个事务特性。
m_db=# SET m_format_dev_version='s2';
m_db=# START TRANSACTION;
m_db=# SET TRANSACTION ISOLATION LEVEL REPEATABLE READ READ ONLY; -- ERROR:  Transaction characteristics can't be changed while a transaction is in progress.
m_db=# COMMIT;
```

4.4.2.16.8 SHOW

功能描述

SHOW显示当前运行时参数的数值，同时还可以用于显示数据库对象的信息，如库信息、表信息、列信息或服务器的状态信息等。

注意事项

SHOW语法显示数据库对象信息时需要用户拥有相应权限，具体请参见**参数说明**。

语法格式

```
SHOW
 {
```



```
  [VARIABLES LIKE] configuration_parameter |
  TIME ZONE |
  TRANSACTION ISOLATION LEVEL |
  SESSION AUTHORIZATION |
  ALL
};

SHOW {CHARACTER SET | CHARSET} [like_or_where];
SHOW COLLATION [like_or_where];
SHOW [FULL] COLUMNS FROM [tbl_name | db_name.tbl_name] [FROM db_name] [like_or_where];
SHOW CREATE {DATABASE | SCHEMA} [IF NOT EXISTS] db_name;
SHOW CREATE TABLE [tbl_name | db_name.tbl_name];
SHOW CREATE VIEW [view_name | db_name.view_name];
SHOW [DATABASES | SCHEMAS] [like_or_where];
SHOW [STORAGE] ENGINES;
SHOW {INDEX | INDEXES | KEYS} {FROM | IN} tbl_name [{FROM | IN} db_name] [WHERE expr];
SHOW [FULL] PROCESSLIST;
SHOW [GLOBAL | SESSION ] STATUS [like_or_where];
SHOW [GLOBAL | SESSION] VARIABLES [like_or_where];
SHOW [FULL] TABLES [{FROM | IN} db_name][LIKE 'pattern' | WHERE expr];
SHOW TABLE STATUS [{FROM | IN}db_name][LIKE 'pattern' | WHERE expr];

like_or_where:
 {
  LIKE 'pattern' |
  WHERE expr
 };
```

参数说明

- **configuration_parameter**

  运行时参数的名称。

  取值范围：可以使用SHOW ALL命令查看运行时参数。

- **TIME ZONE**

  时区。

- **TRANSACTION ISOLATION LEVEL**

  事务的隔离级别。

- **SESSION AUTHORIZATION**

  当前会话的用户标识符。

- **{CHARACTER SET | CHARSET}**

  显示所有可用的字符集。可以使用LIKE子句对列名进行模糊匹配，也可以使用WHERE子句指定更通用的选择条件。

  SHOW CHARACTER SET显示的列信息包括：

  - Charset：字符集名称。
  - Description：字符集的说明。
  - Default collation：字符集的默认排序规则。
  - Maxlen：存储一个字符所需的最大字节数。

- **COLLATION**

  显示服务器支持的排序规则。可以使用LIKE子句对列名进行模糊匹配，也可以使用WHERE子句指定更通用的选择条件。

  SHOW COLLATION显示的列信息包括：

  - Collation：排序规则名称。
  - Charset：与排序规则相关联的字符集的名称。

  - Id：排序规则id。
  - Default：排序规则是否为其字符集的默认值。
  - Compiled：兼容字段，默认为Yes。
  - Sortlen：这与对以字符集表示的字符串进行排序所需的内存量有关。

- **[FULL] COLUMNS FROM [tbl_name | db_name.tbl_name] [FROM db_name] [like_or_where]**

  显示指定表中的列信息，同时适用于视图。需要拥有指定表所在Schema的USAGE权限，同时还需要拥有指定表的任意表级权限或列级权限。仅显示拥有SELECT、INSERT、UPDATE、REFERENCES和COMMENT权限的列信息。可以使用LIKE子句对列名进行模糊匹配，也可以使用WHERE子句指定更通用的选择条件。

  FULL：可选，使输出额外包含列排序规则、注释及权限信息。可以使用db_name.tbl_name或FROM db_name语句的形式指定表所在的Schema，同时使用时，FROM db_name语句优先级高于db_name.tbl_name。

  SHOW COLUMNS显示的列信息包括：

  - Field：列名。
  - Type：列数据类型。
  - Collation：字符类型的排序规则，仅声明FULL关键字时才会显示。
  - Null：列值是否可以为NULL，可以为YES，否则为NO。
  - Key：列是否被索引，取值范围：
    - PRI：表示该列被主键索引。
    - UNI：表示该列是唯一索引中的第一列。
    - MUL：表示该列是非唯一索引中的第一列。
    - NULL：表示该列没有被索引，或者不是唯一索引或非唯一索引的第一列。
  - Default：列默认值。
  - Extra：列的附加信息，取值范围：
    - auto_increment：表示该列为自增列。
    - on update CURRENT_TIMESTAMP：表示该列具有ON UPDATE约束。
    - VIRTUAL GENERATED：表示该列为虚拟生成列。
    - STORED GENERATED：表示该列为存储生成列。
  - Privilege：用户对该列拥有的权限，只有使用FULL关键字时才会显示。
  - Comment：列注释，只有使用FULL关键字时才会显示。

- **CREATE {DATABASE | SCHEMA} [IF NOT EXISTS] db_name**

  显示创建指定数据库的CREATE DATABASE语句。需要拥有指定Schema的USAGE权限。

  DATABASE和SCHEMA为同义词。如果SHOW语句包含IF NOT EXISTS语句，则输出的CREATE DATABASE语句中也包含IF NOT EXISTS。

- **CREATE TABLE [tbl_name | db_name.tbl_name]**

显示创建指定表的CREATE TABLE语句，同时适用于视图，作用于视图时与SHOW CREATE VIEW功能相同。需要拥有指定表所在Schema的USAGE权限，同时还需要拥有指定表的任意表级权限。

可以使用db_name.tbl_name语句的形式指定表所在的Schema。设置GUC兼容性参数m_format_dev_version为's2'后，返回的结果集可能存在变化，具体信息请参见《兼容性说明》中的“MySQL兼容性说明 > MySQL兼容性M-Compatibility模式 > SQL > DML”章节的“SHOW CREATE TABLE”语法。

- **CREATE VIEW [view_name | db_name.view_name]**

  显示创建指定视图的CREATE VIEW语句。需要拥有指定视图所在Schema的USAGE权限，同时还需要拥有指定视图的任意表级权限。可以使用db_name.view_name语句的形式指定视图所在的Schema。精度传递开关打开（m_format_behavior_compat_options开启enable_precision_decimal选项）时，不展示隐式类型转换信息。

- **[STORAGE] ENGINES**

  当前数据库服务端支持的存储引擎。

- **[GLOBAL | SESSION] STATUS**

  显示数据库连接数以及运行时间，如不显式指定GLOBAL或SESSION，默认为当前会话。

  - GLOBAL：显示全局范围内的数据库连接数以及运行时间。
  - SESSION：显示当前会话的数据库连接数以及运行时间。

- **DATABASES | SCHEMAS**

  当前数据库中所有Schema。查询结果使用字符集utf8mb4、字符序utf8mb4_bin。

- **INDEX | INDEXES | KEYS**

  显示索引信息。GaussDB中临时表存储于独立的临时Schema中，在使用FROM或IN db_name条件来展示指定临时表索引信息时，须指明db_name为临时表所在的Schema才能展示临时表索引信息，否则会提示不存在该临时表。查询结果中Table、Index_type、Index_comment字段使用字符集utf8mb4、字符序utf8mb4_bin；字段Key_name、Column_name、Collation、Null、Comment字段使用字符集utf8mb4、字符序utf8mb4_general_ci。

- **[FULL] PROCESSLIST**

  显示当前服务器内的线程信息。如果没有SYSADMIN权限，用户只能看到当前用户线程的信息。显示的内容同information_schema.processlist视图，具体返回结果请参见**processlist**。如果未指定FULL，返回结果中的Info列仅显示前100个字符。

- **ALL**

  所有运行时参数。

- **[GLOBAL | SESSION] VARIABLES**

  显示数据库系统变量的值，如不显式指定GLOBAL或SESSION，默认为当前会话。可以使用LIKE子句对列名进行模糊匹配，也可以使用WHERE子句指定更通用的选择条件。

  - GLOBAL：显示全局系统变量值。这些值用于为数据库的新连接初始化相应的会话变量。如果变量没有全局值，则不显示任何值。
  - SESSION：显示当前会话的系统变量值。如果变量没有会话值，则显示全局值。

- **[FULL] TABLES [{FROM | IN} db_name][LIKE 'pattern' | WHERE expr]**

  查看指定数据库中的表或者视图，默认是用户查询时使用的数据库。不包含临时表。如果指定数据库，用户必须要有数据库的权限。查询结果按照升序排序。查询结果使用字符集utf8mb4、字符序utf8mb4_bin。

  - FULL：FULL可选，若指定FULL，显示信息中包含Table_type列。
  - {FROM | IN} db_name：指定查询数据库db_name中的表。
  - LIKE 'pattern': LIKE语法筛选符合pattern条件的表，当目标database是information_schema时，pattern被转为小写再进行匹配。
  - WHERE expr: WHERE语法筛选符合expr条件的表。

- **TABLE STATUS [{FROM | IN}db_name][LIKE 'pattern' | WHERE expr]**

  M-compatibility模式数据库中SHOW TABLE STATUS语法功能为显示表和视图的信息，未指定Schema时默认为当前所在Schema，不包含临时表。查询结果使用的字符集和字符序在information_schema.tables的对应字段中体现。

  - {FROM | IN} db_name：指定查询数据库db_name中的表。
  - LIKE 'pattern': LIKE语法筛选符合pattern条件的表，当目标database是information_schema时，pattern被转为小写再进行匹配。
  - WHERE expr: WHERE语法筛选符合expr条件的表。

表 4-26 SHOW TABLE STATUS 查询结果内容

展示列	字段含义	映射字段
Name	表名称	information_schema.tables.TABLE_NAME
Engine	表所用的引擎	information_schema.tables.ENGINE
Version	所属版本	information_schema.tables.VERSION
Row_format	行格式	information_schema.tables.ROW_FORMAT
Rows	表中的行数	information_schema.tables.TABLE_ROWS
Avg_row_length	平均长度	information_schema.tables.AVG_ROW_LENGTH
Data_length	数据文件长度	information_schema.tables.DATA_LENGTH
Max_data_length	数据文件最大长度	information_schema.tables.MAX_DATA_LENGTH
Index_length	索引长度	information_schema.tables.INDEX_LENGTH
Data_free	已分配但未使用的字节数	information_schema.tables.DATA_FREE

展示列	字段含义	映射字段
Auto_increment	下一次自增值	information_schema.tables.AUTO_INCREMENT
Create_time	创建表时间	information_schema.tables.CREATE_TIME
Update_time	上一次更新数据的时间	information_schema.tables.UPDATE_TIME
Check_time	表的最后一次check时间	information_schema.tables.CHECK_TIME
Collation	表排序规则	information_schema.tables.TABLE_COLLATION
Checksum	校验和	information_schema.tables.CHECKSUM
Create_options	创建表的选项	information_schema.tables.CREATE_OPTIONS
Comment	表注释	information_schema.tables.TABLE_COMMENT

示例

- SHOW语法示例

```
--显示 timezone 参数值。
m_db=# SHOW timezone;

--显示所有参数。
m_db=# SHOW ALL;

--显示参数名中包含”var”的所有参数。
m_db=# SHOW VARIABLES LIKE var;
```

- SHOW CHARACTER SET语法示例

```
-- 显示所有可用的字符集。
m_db=# SHOW CHARACTER SET;
 charset | description | default collation  | maxlen
---------+-------------+--------------------+--------
 UTF8    |             | utf8mb4_general_ci |
(1 row)
```

- SHOW COLLATION语法示例

```
--显示服务器支持的排序规则。
m_db=# SHOW COLLATION;
     collation      | charset | id | default | compiled | sortlen
--------------------+---------+----+---------+----------+---------
 C                  | UTF8    |    |         | Yes      |
 default            | UTF8    |    |         | Yes      |
 POSIX              | UTF8    |    |         | Yes      |
 utf8_bin           | UTF8    |    |         | Yes      |
 utf8_general_ci    | UTF8    |    |         | Yes      |
 utf8mb4_0900_ai_ci | UTF8    |    |         | Yes      |
 utf8mb4_bin        | UTF8    |    |         | Yes      |
 utf8mb4_general_ci | UTF8    |    | Yes     | Yes      |
 utf8mb4_unicode_ci | UTF8    |    |         | Yes      |
 utf8_unicode_ci    | UTF8    |    |         | Yes      |
(10 rows)
```


- SHOW COLUMNS语法示例

```
-- 创建表t01
m_db=# CREATE TABLE t01 (
    c1 int AUTO_INCREMENT COMMENT 'this is c1',
    c2 int NOT NULL CHECK (c2 > 0),
    c3 text CHARACTER SET utf8mb4 COLLATE utf8mb4_bin NULL COMMENT 'this is c3',
    c4 int DEFAULT 5 COMMENT 'this is c4',
    c5 datetime ON UPDATE current_timestamp COMMENT 'this is c5',
    c6 int GENERATED ALWAYS AS (c2 + c4) STORED COMMENT 'this is c6',
    c7 int UNIQUE COMMENT 'this is c7',
    c8 int,
    c9 text CHARSET utf8mb4 COLLATE utf8mb4_bin COMMENT 'this is c9',
    c10 int COMMENT 'this is c10',
    c11 int COMMENT 'this is c11',
    c12 int COMMENT 'this is c12',
    CONSTRAINT PRIMARY KEY (c1) USING btree COMMENT 'this is primary key',
    CONSTRAINT FOREIGN KEY (c8) REFERENCES t01(c1),
    CONSTRAINT CHECK (c10 > 0),
    CONSTRAINT UNIQUE (c11) COMMENT 'this is unique constraint',
    c13 text COMMENT 'this is c13',
    c14 int COMMENT 'this is c14',
    INDEX USING btree (c13(10)) COMMENT 'this is index',
    INDEX (c13(10),c14) USING btree COMMENT 'this is index too'
);

-- 显示列信息
m_db=# SHOW FULL COLUMNS FROM t01;
 Field |   Type   |     Collation      | Null | Key | Default |           Extra            |           Privileges
|   Comment
-------+----------+--------------------+------+-----+---------+----------------------------
+--------------------------------+-------------
 c1    | integer  |                    | NO   | PRI |         | auto_increment             |
select,insert,update,references | this is c1
 c2    | integer  |                    | NO   |     |         |                            | select,insert,update,references |
 c3    | text     | utf8mb4_bin        | YES  |     |         |                            | select,insert,update,references
| this is c3
 c4    | integer  |                    | YES  |     | 5       |                            | select,insert,update,references |
this is c4
 c5    | datetime |                    | YES  |     |         | on update CURRENT_TIMESTAMP |
select,insert,update,references | this is c5
 c6    | integer  |                    | YES  |     |         | STORED GENERATED           |
select,insert,update,references | this is c6
 c7    | integer  |                    | YES  | UNI |         |                            | select,insert,update,references |
this is c7
 c8    | integer  |                    | YES  |     |         |                            | select,insert,update,references |
 c9    | text     | utf8mb4_bin        | YES  |     |         |                            | select,insert,update,references
| this is c9
 c10   | integer  |                    | YES  |     |         |                            | select,insert,update,references |
this is c10
 c11   | integer  |                    | YES  | UNI |         |                            | select,insert,update,references |
this is c11
 c12   | integer  |                    | YES  |     |         |                            | select,insert,update,references |
this is c12
 c13   | text     | utf8mb4_general_ci | YES  | MUL |         |                            |
select,insert,update,references | this is c13
 c14   | integer  |                    | YES  |     |         |                            | select,insert,update,references |
this is c14
(14 rows)

m_db=# DROP TABLE t01;
```

- SHOW CREATE DATABASE语法示例

```
-- 创建db1
m_db=# CREATE DATABASE IF NOT EXISTS db1 DEFAULT CHARACTER SET utf8mb4 DEFAULT
COLLATE utf8mb4_bin;

-- 显示db1创建语句
m_db=# SHOW CREATE DATABASE db1;
 Database |                      Create Database
```



```
----------+------------------------------------------------------------------------
 db1      | CREATE DATABASE db1 DEFAULT CHARACTER SET UTF8 COLLATE utf8mb4_bin
(1 row)

-- 带IF NOT EXISTS
m_db=# SHOW CREATE DATABASE IF NOT EXISTS db1;
 Database |                              Create Database
----------+-------------------------------------------------------------------------------------
 db1      | CREATE DATABASE IF NOT EXISTS db1 DEFAULT CHARACTER SET UTF8 COLLATE
utf8mb4_bin
(1 row)

m_db=# DROP DATABASE db1;
```

- SHOW CREATE TABLE语法示例

```
-- 创建表t01
m_db=# CREATE GLOBAL TEMPORARY TABLE t01(
    c1 int COMMENT 'this is c1',
    c2 int PRIMARY KEY USING INDEX TABLESPACE pg_default,
    c3 int,
    c4 int,
    CONSTRAINT UNIQUE (c3) COMMENT 'this is unique',
    INDEX idx (c3) COMMENT 'this is index'
) COMMENT 'this is table'
ON COMMIT PRESERVE ROWS;

-- 设置兼容性参数m_format_dev_version为s2
m_db=# SET m_format_dev_version='s2';
-- 显示表t01的建表语句
m_db=# SHOW CREATE TABLE t01;
 Table |                                          Create Table
-------
+-----------------------------------------------------------------------------------------------------------------
 t01   | SET search_path = public;                                                                                +
       | CREATE GLOBAL TEMPORARY TABLE t01
(                                                                                                                +
       |     c1 integer COMMENT 'this is c1',                                                                      +
       |     c2 integer NOT NULL,                                                                                 +
       |     c3 integer,                                                                                          +
       |     c4 integer,                                                                                          +
       |     CONSTRAINT t01_pkey PRIMARY KEY (c2) WITH (storage_type=USTORE) USING
ubtree,                                +
       |     CONSTRAINT t01_c3_key UNIQUE (c3) WITH (storage_type=USTORE) USING ubtree
COMMENT 'this is unique'             +
       | )                                                                                                        +
       | CHARACTER SET = "UTF8" COLLATE = "utf8mb4_general_ci" COMMENT 'this is
table'                                  +
       | WITH (orientation=row, compression=no, storage_type=USTORE,
segment=off)                                        +
       | ON COMMIT PRESERVE ROWS;                                                                                +
       | CREATE INDEX idx USING ubtree ON t01 (c3) WITH (storage_type=USTORE) TABLESPACE
pg_default COMMENT 'this is index';
(1 row)

-- 设置兼容性参数m_format_dev_version为空
m_db=# SET m_format_dev_version='';
-- 显示表t01的建表语句
m_db=# SHOW CREATE TABLE t01;
 Table |                                          Create Table
-------
+-----------------------------------------------------------------------------------------------------------------
 t01   | SET search_path = public;                                                                                +
       | CREATE GLOBAL TEMPORARY TABLE t01
(                                                                                                                +
       |     c1 integer,                                                                                          +
       |     c2 integer NOT NULL,                                                                                +
       |     c3 integer,                                                                                          +
       |     c4 integer                                                                                           +
       | )                                                                                                       +
       | CHARACTER SET = "UTF8" COLLATE =
```



```
"utf8mb4_general_ci"                                                                +
      | WITH (orientation=row, compression=no, storage_type=USTORE, on_commit_delete_rows=false,
segment=off);              +
      | COMMENT ON TABLE t01 IS 'this is table';                                                                +
      | COMMENT ON COLUMN t01.c1 IS 'this is c1';
+
      | CREATE INDEX idx USING ubtree ON t01 (c3) WITH (storage_type=USTORE) TABLESPACE
pg_default COMMENT 'this is index';+
      | COMMENT ON INDEX idx IS 'this is index';                                                                +
      | ALTER TABLE t01 ADD CONSTRAINT t01_c3_key UNIQUE USING ubtree (c3) WITH
(storage_type=USTORE);                    +
      | COMMENT ON INDEX t01_c3_key IS 'this is
unique';                                                  +
      | ALTER TABLE t01 ADD CONSTRAINT t01_pkey PRIMARY KEY USING ubtree  (c2) WITH
(storage_type=USTORE);
(1 row)

m_db=# DROP TABLE t01;
```

- SHOW CREATE VIEW语法示例

```
-- 创建视图
m_db=# CREATE TABLE t01 (c1 int, c2 int, c3 int);
m_db=# CREATE VIEW v01 AS SELECT c1 AS col1, c3 AS col2 FROM t01;

-- 显示视图v01的创建语句
m_db=# SHOW CREATE VIEW v01;
 View |                  Create View                   | character_set_client | collation_connection
------+------------------------------------------------+----------------------+----------------------
 v01  | CREATE VIEW v01 AS                            +|                      |
      | SELECT t01.c1 AS col1, t01.c3 AS col2 FROM t01; |                      |
(1 row)

m_db=# DROP VIEW v01;
m_db=# DROP TABLE t01;
```

- SHOW PROCESSLIST语法示例

```
-- 用test用户连接M兼容库db_m，显示线程信息
m_db=# SHOW FULL PROCESSLIST;
       Id        | User | Host |  db  | Command | Time | State |          Info
-----------------+------+------+------+---------+------+-------+------------------------
 139880022669056 | test |      | m_db |         |    0 |       | SHOW FULL PROCESSLIST;
(1 row)
```

- SHOW INDEX语法示例

```
-- 创建表和索引
m_db=# CREATE TABLE t01 (c1 int PRIMARY KEY);

-- 显示索引信息的语句
m_db=# SHOW INDEX from t01;
 Table | Non_unique | Key_name | Seq_in_index | Column_name | Collation | Cardinality | Sub_part |
Packed | Null | Index_type | Comment | Index_comment
-------+------------+----------+--------------+-------------+-----------+-------------+----------+--------+------
+------------+---------+---------------
 t01   |          0 | PRIMARY  |            1 | c1          | A         |           0 |          |        |      | UBTREE
|            |         |
(1 row)
```

- SHOW VARIABLES语法示例

```
-- 显示全局系统变量值
m_db=# SHOW GLOBAL VARIABLES LIKE 'a_format%';
              Variable_name               | Value
------------------------------------------+-------
 a_format_copy_version                    |
 a_format_date_timestamp                  | off
 a_format_dev_version                     |
 a_format_disable_func                    |
 a_format_load_with_constraints_violation |
 a_format_version                         |
(6 rows)

-- 显示对当前会话的系统变量值
```



```
m_db=# SHOW SESSION VARIABLES LIKE 'query_%';
  Variable_name   |  Value
------------------+---------
 query_max_mem    | 0
 query_dop        | 1
 query_mem        | 0
 query_cache_type | 0
 query_band       |
 query_cache_size | 1048576
(6 rows)
```

- SHOW TABLES语法示例

```
m_db=# CREATE SCHEMA sc_test;
CREATE SCHEMA
m_db=# USE sc_test;
SET
m_db=# CREATE TABLE t1(a int, b int);
CREATE TABLE
m_db=# CREATE TABLE t2(a int, b int);
CREATE TABLE
m_db=# SHOW TABLES;
 Tables_in_sc_test
-------------------
 t1
 t2
(2 rows)
m_db=# SHOW FULL TABLES;
 Tables_in_sc_test | Table_type
-------------------+------------
 t1                | BASE TABLE
 t2                | BASE TABLE
(2 rows)

m_db=# SHOW TABLES FROM sc_test;
 Tables_in_sc_test
-------------------
 t1
 t2
(2 rows)
m_db=# SHOW TABLES LIKE "%1";
 Tables_in_sc_test (%1)
------------------------
 t1
(1 row)

m_db=# SHOW TABLES LIKE "t%";
 Tables_in_sc_test (t%)
------------------------
 t1
 t2
(2 rows)

m_db=# SHOW TABLES WHERE Tables_in_sc_test='t1';
 Tables_in_sc_test
-------------------
 t1
(1 row)
m_db=# DROP SCHEMA sc_test;
NOTICE:  drop cascades to 2 other objects
DETAIL:  drop cascades to table sc_test.t1
drop cascades to table sc_test.t2
DROP SCHEMA
```

- SHOW TABLE STATUS语法示例

```
m_db=# CREATE SCHEMA sc_test;
CREATE SCHEMA
m_db=# USE sc_test;
SET
m_db=# CREATE TABLE t1(a int, b int);
CREATE TABLE
```



```
m_db=# CREATE TABLE t2(a int, b int);
CREATE TABLE
m_db=# 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
------+--------+---------+------------+------+----------------+-------------+-----------------+--------------
+-----------+----------------+--------------------+-------------+------------+--------------------+----------
+-------------------------------------------------------------------------+---------
 t1   | Ustore |         |            | 0    |              | 0           |               | 0           |         |              |
2024-04-21 15:56:25 |            |           | utf8mb4_general_ci |         | orientation=row
compression=no storage_type=USTORE collate=1537 segment=off |
 t2   | Ustore |         |            |      |              | 0           |               | 0           |         |              |
2024-04-21 15:56:32 |            |           | utf8mb4_general_ci |         | orientation=row
compression=no storage_type=USTORE collate=1537 segment=off |
(2 rows)

m_db=# SHOW TABLE STATUS FROM sc_test;
 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
------+--------+---------+------------+------+----------------+-------------+-----------------+--------------
+-----------+----------------+--------------------+-------------+------------+--------------------+----------
+-------------------------------------------------------------------------+---------
 t1   | Ustore |         |            | 0    |              | 0           |               | 0           |         |              |
2024-04-21 15:56:25 |            |           | utf8mb4_general_ci |         | orientation=row
compression=no storage_type=USTORE collate=1537 segment=off |
 t2   | Ustore |         |            | 0    |              | 0           |               | 0           |         |              |
2024-04-21 15:56:32 |            |           | utf8mb4_general_ci |         | orientation=row
compression=no storage_type=USTORE collate=1537 segment=off |
(2 rows)

m_db=# SHOW TABLE STATUS FROM sc_test LIKE '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
------+--------+---------+------------+------+----------------+-------------+-----------------+--------------
+-----------+----------------+--------------------+-------------+------------+--------------------+----------
+-------------------------------------------------------------------------+---------
 t1   | Ustore |         |            | 0    |              | 0           |               | 0           |         |              |
2024-04-21 15:56:25 |            |           | utf8mb4_general_ci |         | orientation=row
compression=no storage_type=USTORE collate=1537 segment=off |
 t2   | Ustore |         |            | 0    |              | 0           |               | 0           |         |              |
2024-04-21 15:56:32 |            |           | utf8mb4_general_ci |         | orientation=row
compression=no storage_type=USTORE collate=1537 segment=off |
(2 rows)

m_db=# SHOW TABLE STATUS FROM sc_test LIKE '%1';
 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
------+--------+---------+------------+------+----------------+-------------+-----------------+--------------
+-----------+----------------+--------------------+-------------+------------+--------------------+----------
+-------------------------------------------------------------------------+---------
 t1   | Ustore |         |            | 0    |              | 0           |               | 0           |         |              |
2024-04-21 15:56:25 |            |           | utf8mb4_general_ci |         | orientation=row
compression=no storage_type=USTORE collate=1537 segment=off |
(1 row)
                                          ^
m_db=# SHOW TABLE STATUS FROM sc_test WHERE Name='t1';
 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
------+--------+---------+------------+------+----------------+-------------+-----------------+--------------
+-----------+----------------+--------------------+-------------+------------+--------------------+----------
+-------------------------------------------------------------------------+---------
 t1   | Ustore |         |            | 0    |              | 0           |               | 0           |         |              |
2024-04-21 15:56:25 |            |           | utf8mb4_general_ci |         | orientation=row
compression=no storage_type=USTORE collate=1537 segment=off |
```



```
(1 row)

m_db=# SHOW TABLE STATUS FROM sc_test WHERE Name LIKE '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
------+--------+---------+------------+------+----------------+-------------+-----------------+--------------
+-----------+----------------+--------------------+-------------+------------+--------------------+----------
+-------------------------------------------------------------------------------+---------
 t1   | Ustore |         |            | 0    |              | 0           |               | 0           |         |              |
2024-04-21 15:56:25 |             |           | utf8mb4_general_ci |          | orientation=row
compression=no storage_type=USTORE collate=1537 segment=off |
 t2   | Ustore |         |            | 0    |              | 0           |               | 0           |         |              |
2024-04-21 15:56:32 |             |           | utf8mb4_general_ci |          | orientation=row
compression=no storage_type=USTORE collate=1537 segment=off |
(2 rows)
m_db=# DROP SCHEMA sc_test;
NOTICE:  drop cascades to 2 other objects
DETAIL:  drop cascades to table sc_test.t1
drop cascades to table sc_test.t2
DROP SCHEMA
```

相关链接

SET，**RESET**

4.4.2.16.9 START TRANSACTION

功能描述

通过START TRANSACTION启动事务。如果声明了隔离级别、读写模式，那么新事务就使用这些特性，类似执行了**SET TRANSACTION**。

语法格式

格式一：START TRANSACTION格式

```
START TRANSACTION
  [
    {
      ISOLATION LEVEL { READ COMMITTED | READ UNCOMMITTED | SERIALIZABLE | REPEATABLE READ }
      | { READ WRITE | READ ONLY } | WITH CONSISTENT SNAPSHOT
    } [ ...]
  ];
```

格式二：BEGIN格式

```
BEGIN [ WORK | TRANSACTION ]
  [
    {
      ISOLATION LEVEL { READ COMMITTED | READ UNCOMMITTED | SERIALIZABLE | REPEATABLE READ }
      | { READ WRITE | READ ONLY }
     } [ ...]
  ];
```

参数说明

- **WORK | TRANSACTION**

  BEGIN格式中的可选关键字，没有实际作用。

- **ISOLATION LEVEL**

  指定事务隔离级别，它决定当一个事务中存在其他并发运行事务时它能够看到什么数据。

> **说明**
>
> 在事务中第一个数据修改语句（SELECT, INSERT，DELETE，UPDATE，COPY）执行之后，事务隔离级别就不能再次设置。

取值范围：

- READ COMMITTED：读已提交隔离级别，只能读到已经提交的数据，而不会读到未提交的数据。这是缺省值。
- READ UNCOMMITTED：读未提交隔离级别，指定后的行为和READ COMMITTED行为一致。
- REPEATABLE READ：可重复读隔离级别，仅仅看到事务开始之前提交的数据，它不能看到未提交的数据，以及在事务执行期间由其它并发事务提交的修改。
- SERIALIZABLE：M-Compatibility目前功能上不支持此隔离级别，设置该隔离级别时，等价于REPEATABLE READ。

- **READ WRITE | READ ONLY**

  指定事务访问模式（读/写或者只读）。

- **WITH CONSISTENT SNAPSHOT**

  仅在可重复读隔离级别下生效，开启事务时即生成快照，其余隔离级别不生效，并产生告警。

> **说明**
>
> 如果关闭此参数，在可重复读隔离级别，事务中第一个数据修改语句（DML、DDL或DCL）执行之后，即会建立事务的一致性读快照。支持多次设置此参数。

示例

```
--创建SCHEMA。
m_db=# CREATE SCHEMA tpcds;

--创建表tpcds.reason。
m_db=# CREATE TABLE tpcds.reason (c1 int, c2 int);

--以默认方式启动事务。
m_db=# START TRANSACTION;
m_db=# SELECT * FROM tpcds.reason;
m_db=# COMMIT;

--以默认方式启动事务。
m_db=# BEGIN;
m_db=# SELECT * FROM tpcds.reason;
m_db=# COMMIT;

--以隔离级别为READ COMMITTED，读/写方式启动事务。
m_db=# START TRANSACTION ISOLATION LEVEL READ COMMITTED READ WRITE;
m_db=# SELECT * FROM tpcds.reason;
m_db=# COMMIT;

--可重复读隔离级别下，带WITH CONSISTENT SNAPSHOT开启事务。
m_db=# START TRANSACTION WITH CONSISTENT SNAPSHOT ISOLATION LEVEL REPEATABLE READ;
m_db=# SELECT * FROM tpcds.reason;
m_db=# COMMIT;

--非可重复读隔离级别下，带WITH CONSISTENT SNAPSHOT开启事务。
m_db=# START TRANSACTION WITH CONSISTENT SNAPSHOT ISOLATION LEVEL READ COMMITTED;
WARNING:  WITH CONSISTENT SNAPSHOT was ignored because this phrase can only be used with REPEATABLE READ isolation level.
m_db=# SELECT * FROM tpcds.reason;
m_db=# COMMIT;
```



```
--删除表tpcds.reason。
m_db=# DROP TABLE tpcds.reason;

--删除SCHEMA。
m_db=# DROP SCHEMA tpcds;
```

相关链接

COMMIT，**ROLLBACK**，**SET TRANSACTION**

4.4.2.17 T

4.4.2.17.1 TRUNCATE

功能描述

清理表数据，TRUNCATE快速地从表中删除所有行。

它和在目标表上进行无条件的DELETE有同样的效果，但由于TRUNCATE不做表扫描，因而快得多。在大表上操作效果更明显。

注意事项

- TRUNCATE TABLE在功能上与不带WHERE子句DELETE语句相同：二者均删除表中的全部行。
- TRUNCATE TABLE比DELETE速度快且使用系统和事务日志资源少：
  - DELETE语句每次删除一行，并在事务日志中为所删除每行记录一项。
  - TRUNCATE TABLE通过释放存储表数据所用数据页来删除数据，并且只在事务日志中记录页的释放。
- TRUNCATE，DELETE，DROP三者的差异如下：
  - TRUNCATE TABLE，删除内容，释放空间，但不删除定义。
  - DELETE TABLE，删除内容，不删除定义，不释放空间。
  - DROP TABLE，删除内容和定义，释放空间。

语法格式

- 清理表数据。

```
TRUNCATE [ TABLE ] [ONLY] {table_name [ * ]} [, ... ]
   [CASCADE | RESTRICT][ PURGE ]};
```

- 清理表分区的数据。

```
ALTER TABLE { [ ONLY  ] table_name
          | table_name *
          | ONLY ( table_name )  }
   TRUNCATE PARTITION { { ALL | partition_name [, ...] }
                    | FOR (  partition_value  [, ...] )  } [ UPDATE GLOBAL INDEX ];
```

参数说明

- **ONLY**

  如果声明ONLY，只有指定的表会被清空。如果没有声明ONLY，这个表以及其所有子表（若有）会被清空。目前ONLY和增加*选项保留语法，但功能不支持。

- **table_name**
  目标表的名称（可以有模式修饰）。
  取值范围：已存在的表名。
- **CASCADE | RESTRICT**
  - CASCADE：级联删除依赖此视图的对象（比如其他视图）。
  - RESTRICT：如果有依赖对象存在，则拒绝删除此视图。此选项为缺省值。

  说明

  该属性在数据库M-compatibility模式兼容版本控制开关s1及以上版本时不支持（如m_format_dev_version = 's1'）。
- **PURGE**
  默认将表数据放入回收站中，PURGE直接清理。
- **{ ALL | partition_name [, ...] }**
  ALL：清除所有分区数据。
  partition_name ： 目标分区表的分区名，支持一级分区名和二级分区名。
  取值范围：已存在的分区名。
- **partition_value**
  指定的分区键值。
  通过PARTITION FOR子句指定的这一组值，可以唯一确定一个分区。
  取值范围：需要进行删除数据分区的分区键的取值范围。

须知

1. 如果是其他表的外表，使用truncate清空数据时会失败并且报错。
2. 使用PARTITION FOR子句时，partition_value所在的整个分区会被清空。

- **UPDATE GLOBAL INDEX**
  如果使用该参数，则会更新分区表上的所有全局索引，以确保使用全局索引可以查询出正确的数据；如果不使用该参数，则分区表上的所有全局索引将会失效。

示例

```
--创建SCHEMA。
m_db=# CREATE SCHEMA tpcds;

--创建表tpcds.reason。
m_db=# CREATE TABLE tpcds.reason
(
  r_reason_sk       integer,
  r_reason_id       character(16),
  r_reason_desc     character(100)
);

--向表中插入多条记录。
m_db=# INSERT INTO tpcds.reason values(1,'AAAAAAAABAAAAAAA','reason 1'),
(5,'AAAAAAAABAAAAAAA','reason 2'),(15,'AAAAAAAABAAAAAAA','reason 3'),
(25,'AAAAAAAABAAAAAAA','reason 4'),(35,'AAAAAAAABAAAAAAA','reason 5'),
(45,'AAAAAAAACAAAAAAA','reason 6'),(55,'AAAAAAAACAAAAAAA','reason 7');

--创建表。
m_db=# CREATE TABLE tpcds.reason_t1  LIKE tpcds.reason;
```



```
--清空表tpcds.reason_t1。
m_db=# TRUNCATE TABLE tpcds.reason_t1;

--删除表。
m_db=# DROP TABLE tpcds.reason_t1;
--创建分区表。
m_db=# CREATE TABLE tpcds.reason_p
(
  r_reason_sk integer,
  r_reason_id character(16),
  r_reason_desc character(100)
)PARTITION BY RANGE (r_reason_sk)
(
  partition p_05_before values less than (05),
  partition p_15 values less than (15),
  partition p_25 values less than (25),
  partition p_35 values less than (35),
  partition p_45_after values less than (MAXVALUE)
);

--插入数据。
m_db=# INSERT INTO tpcds.reason_p SELECT * FROM tpcds.reason;

--清空分区p_05_before。
m_db=# ALTER TABLE tpcds.reason_p TRUNCATE PARTITION p_05_before;

--清空分区p_15。
m_db=# ALTER TABLE tpcds.reason_p TRUNCATE PARTITION for (15);

--清空分区表。
m_db=# TRUNCATE TABLE tpcds.reason_p;

--删除表。
m_db=# DROP TABLE tpcds.reason_p;

--删除表。
m_db=# DROP TABLE tpcds.reason;

--删除SCHEMA。
m_db=# DROP SCHEMA tpcds;
```

4.4.2.18 U

4.4.2.18.1 UPDATE

功能描述

更新表中的数据，UPDATE对满足条件的所有行中指定的字段值进行修改。WHERE子句表示声明条件；SET子句中指定的字段将会被修改，没有出现的字段则保持原有的字段值。

注意事项

- 表的所有者、拥有表UPDATE权限的用户、拥有UPDATE ANY TABLE权限的用户，皆有权限更新表中的数据，当三权分立开关关闭时，系统管理员默认拥有此权限。
- 对expression或condition条件里涉及到的任何表需要有SELECT权限。
- 生成列不能被直接写入。在UPDATE命令中不能为生成列指定值，但是可以指定关键字DEFAULT。
- 对于多表更新语法，暂时不支持对视图进行多表更新。

- 对于子查询是STREAM计划的UPDATE语句，不支持并发更新同一行。
- 不支持用户通过UPDATE系统表的方式对数据库字符编码进行修改，该操作会导致存在存量数据或其他部分操作异常的情况。如需更改数据库的字符集编码，应当遵循切库流程，进行相关的数据迁移操作。

语法格式

```
单表更新：
[ WITH [ RECURSIVE ] with_query [, ...] ]
UPDATE [/*+ plan_hint */] [IGNORE] [ ONLY ] {table_name [ partition_clause ] | subquery | view_name}
[ * ] [ [ AS ] alias ]
SET {column_name = { expression | DEFAULT }
}[, ...]
   [ WHERE condition ]
   [ORDER BY {col_name | expression | position}
     [ASC | DESC][ NULLS { FIRST | LAST } ], ...]
    [ LIMIT { count } ];

多表更新：
[ WITH [ RECURSIVE ] with_query [, ...] ]
UPDATE [/*+ plan_hint */] [IGNORE] table_list
SET {column_name = { expression | DEFAULT }
}[, ...]
    [ WHERE condition ];
```

参数说明

- **WITH [RECURSIVE] with_query [, ...]**

  用于声明一个或多个可以在主查询中通过名称引用的子查询，相当于临时表。这种子查询语句结构称为CTE（Common Table Expression）结构，应用这种结构时，执行计划中将存在CTE SCAN的内容。

  如果声明了RECURSIVE，那么允许SELECT子查询通过名称引用其本身。

  其中with_query的详细格式为：

  ```
  with_query_name [ ( column_name [, ...] ) ] AS ( {select} )
  ```

  - with_query_name指定子查询生成的结果集名称，在查询中可使用该名称访问子查询的结果集。
  - column_name指定子查询结果集中显示的列名。
  - 每个子查询支持SELECT语句。
  - RECURSIVE只能出现在WITH后面，多个CTE的情况下，只需要在第一个CTE处声明RECURSIVE。
  - 使用RECURSIVE时，CTE子查询中UNION ALL和EXCEPT ALL或UNION [DISTINCT]和EXCEPT [DISTINCT]两侧的子查询结果，其数据类型必须使用cast函数转换成相同的数据类型，且两侧子查询结果的精度和字符序也要相同。如：WITH RECURSIVE cte (n) AS (SELECT cast(id as signed int) from table_1 UNION ALL SELECT cast((n + 1) as signed int) FROM cte WHERE n < 5) SELECT * FROM cte。由操作符产生的类型转换具体请参见**逻辑操作符规格约束**、**位运算操作符规格约束**和**算术操作符规格约束**。

- **plan_hint**

  以/*+ */的形式在UPDATE关键字后，用于对UPDATE对应的语句块生成的计划进行hint调优。每条语句中只有第一个/*+ plan_hint */注释块会作为hint生效，里面可以写多条hint。

- **IGNORE**

UPDATE语句使用IGNORE关键字时，可将部分ERROR级别的错误降级为WARNING级别，并根据不同的错误场景将无效值调整为最接近的值。GaussDB支持错误降级的场景如下：

- 破坏NOT NULL约束
- 唯一键冲突
- 插入的值没有找到对应的分区
- 指定分区插入时，插入的数据与指定分区不匹配
- 子查询返回多行
- sql_mode为宽松模式的场景

说明

升级观察期不支持IGNORE。

- **table_name**

  要更新的表名，可以使用模式修饰。如果在表名前指定了ONLY，只会更新表中匹配的行。如果未指定，任何从该表继承得到的表中的匹配行也会被更新。目前ONLY和增加*选项保留语法，功能不支持

  取值范围：已存在的表名称。

- **partition_clause**

  指定分区更新操作

  PARTITION { ({ partition_name | subpartition_name } [, ...]) | FOR (partition_value [, ...]) } |

  SUBPARTITION { (subpartition_name) | FOR (subpartition_value [, ...]) }

  关键字请参见**SELECT**。

  示例请参见**CREATE TABLE SUBPARTITION**。

  说明

  PARTITION指定多个分区名时，一级分区名和二级分区名可同时存在，且可以存在相同的分区名，最终分区范围取其并集。

- **subquery**

  要更新的子查询，在对子查询进行更新时，会将子查询当成一个临时视图。

```
[ WITH [ RECURSIVE ] with_query [, ...] ]
SELECT [/*+ plan_hint */] [ ALL ]
    { * | {expression [ [ AS ] output_name ]} [, ...] }
    [ into_option ]
    [ FROM from_item [, ...] ]
    [ WHERE condition ]
    [ORDER BY {col_name | expression | position} [ ASC | DESC ] [ NULLS { FIRST | LAST } ], ...]
    [ into_option ];

into_option:
INTO {
  [[[LOCAL | GLOBAL] TEMPORARY] [TABLE] table_name] |
  OUTFILE file_name [CHARACTER SET encoding_name] [FIELDS fields_items] [LINES lines_items] |
  DUMPFILE file_name
}
```

  其中指定子查询源from_item为：

```
{[ ONLY ] {table_name | view_name} [ * ] [ partition_clause ] [ [ AS ] alias [ ( column_alias [, ...] ) ] ]
|( select ) [ AS ] alias [ ( column_alias [, ...] ) ]
|with_query_name [ [ AS ] alias [ ( column_alias [, ...] ) ] ]
|joined_table
```

  其中joined_table为：


```
joined_table: {
    table_reference [INNER | CROSS] JOIN table_factor [join_specification]
  | table_reference {LEFT|RIGHT} [OUTER] JOIN table_reference join_specification
  | table_reference NATURAL [{LEFT|RIGHT} [OUTER]] JOIN table_factor
}
join_specification: {
    ON join_condition
  | USING (join_column_list)
}
join_column_list:
    column_name [, column_name] ...
```

如果子查询中只有一张表，则对该表更新数据；如果子查询中有多张表或有嵌套关系，则通过判断是否有保留键表确定是否可更新。关于保留键表请参见**CREATE VIEW**。

- **view_name**

  要更新的目标视图。

  说明

  对视图和子查询的更新，有如下约束：

  - 只有直接引用基表用户列的列可进行UPDATE操作。
  - 子查询或视图必须至少包含一个可更新列，关于可更新列请参见**CREATE VIEW**。
  - 不支持在顶层包含DISTINCT、GROUP BY、HAVING、LIMIT、OFFSET子句的视图和子查询。
  - 不支持在顶层包含集合运算（UNION以及EXCEPT）的视图和子查询。
  - 不支持目标列表中包含聚集函数、窗口函数、返回集合函数（array_agg、json_agg、generate_series等）的视图和子查询。
  - 视图和子查询中支持的表类型包括普通表、临时表、全局临时表、分区表、二级分区表、ustore表、astore表。
  - 多表连接视图或连接子查询中一次只能更新一张基表。
  - 连接视图或子查询只能更新保留键表。关于保留键表请参见**CREATE VIEW**。
  - 不支持更新系统视图。

- **alias**

  目标表的别名。

  取值范围：字符串，符合**标识符说明**。

- **table_list**

  一个表的表达式列表，与from_list类似，但可以同时声明目标表和关联表，仅在多表更新语法中使用。

- **column_name**

  要修改的字段名。

  支持使用目标表的别名加字段名来引用这个字段。例如：UPDATE foo AS f SET f.col_name = 'namecol'。

  支持使用库名.别名或库名.表名加字段名来引用这个字段。例如：UPDATE foo AS f SET public.f.col_name = 'namecol'。

  取值范围：已存在的字段名。

  设置GUC兼容性参数m_format_dev_version为's2'后：

  - 如果table_name指定了别名，且只能通过别名引用此表。
  - 仅在单表场景多个字段下，各字段的表达式计算会按从左到右的顺序；如果有引用前面的字段名，则使用其更新后的数据，且支持同一字段名可被修改多次。

- **expression**

  赋给字段的值或表达式。

- **DEFAULT**

  用对应字段的缺省值填充该字段。

  如果没有缺省值，则为NULL。

- **from_list**

  一个表的表达式列表，允许在WHERE条件里使用其他表的字段。与在一个SELECT语句的FROM子句里声明表列表类似。

  须知

  目标表不能出现在from_list里，除非在使用一个自连接（此时它必须以from_list的别名出现）。

- **condition**

  一个返回Boolean类型结果的表达式。只有这个表达式返回true的行才会被更新。不建议使用int等数值类型作为condition，因为int等数值类型可以隐式转换为bool值（非0值隐式转换为true，0转换为false），可能导致非预期的结果。

- **ORDER BY**

  关键字详见**SELECT**章节介绍。

- **LIMIT**

  关键字详见**SELECT**章节介绍。

示例

- 修改表中所有数据。

```
--创建tbl_test1表并插入数据。
m_db=# CREATE TABLE tbl_test1(id int, info varchar(10));
m_db=# INSERT INTO tbl_test1 VALUES (1, 'A'), (2, 'B');

--查询。
m_db=# SELECT * FROM tbl_test1;
 id | info
----+------
  1 | A
  2 | B
(2 rows)

--修改tbl_test1表中info列的信息。
m_db=# UPDATE tbl_test1 SET info = 'aa';

--查询tbl_test1表。
m_db=# SELECT * FROM tbl_test1;
 id | info
----+------
  1 | aa
  2 | aa
(2 rows)
```

- 修改表中部分数据。

```
--修改tbl_test1表中id=2的数据。
m_db=# UPDATE tbl_test1 SET info = 'bb' WHERE id = 2;

--查询tbl_test1表。
m_db=# SELECT * FROM tbl_test1;
 id | info
```



```
----+------
  1 | aa
  2 | bb
(2 rows)
```

- 修改数据，并返回修改后的数据。

```
--修改tbl_test1表中id=1的数据，并指定返回info列。
m_db=# UPDATE tbl_test1 SET info = 'ABC' WHERE id = 1;
UPDATE 1

--删除tbl_test1表。
m_db=# DROP TABLE tbl_test1;
```

- 更新子查询

```
--创建SCHEMA。
m_db=# CREATE SCHEMA upd_subqry;
CREATE SCHEMA
m_db=# SET CURRENT_SCHEMA = 'upd_subqry';
SET

--创建表并插入数据。
m_db=# CREATE TABLE t1 (x1 int, y1 int);
CREATE TABLE
m_db=# CREATE TABLE t2 (x2 int PRIMARY KEY, y2 int);
NOTICE:  CREATE TABLE / PRIMARY KEY will create implicit index "t2_pkey" for table "t2"
CREATE TABLE
m_db=# INSERT INTO t1 VALUES (1, 1), (2, 2), (3, 3), (5, 5);
INSERT 0 4
m_db=# INSERT INTO t2 VALUES (1, 1), (2, 2), (3, 3), (5, 5);
INSERT 0 4

--通过子查询更新t1。
m_db=# UPDATE (SELECT * FROM t1) SET y1 = 13 where y1 = 3;
UPDATE 1
m_db=# UPDATE (SELECT * FROM t1 WHERE y1 < 2) SET y1 = 12 WHERE y1 = 2;
UPDATE 0

--插入多表连接的子查询。
m_db=# UPDATE (SELECT * FROM t1, t2 WHERE x1 = x2) SET y1 = 11 WHERE y2 = 1;
UPDATE 1

--删除SCHEMA。
m_db=# DROP SCHEMA upd_subqry;
NOTICE:  drop cascades to 2 other objects
DETAIL:  drop cascades to table t1
drop cascades to table t2
DROP SCHEMA
```

- 更新视图

```
--创建SCHEMA。
m_db=# CREATE SCHEMA upd_view;
CREATE SCHEMA
m_db=# SET CURRENT_SCHEMA = 'upd_view';
SET

--创建表并插入数据。
m_db=# CREATE TABLE t1 (x1 int, y1 int);
CREATE TABLE
m_db=# CREATE TABLE t2 (x2 int PRIMARY KEY, y2 int);
NOTICE:  CREATE TABLE / PRIMARY KEY will create implicit index "t2_pkey" for table "t2"
CREATE TABLE
m_db=# INSERT INTO t1 VALUES (1, 1), (2, 2), (3, 3), (5, 5);
INSERT 0 4
m_db=# INSERT INTO t2 VALUES (1, 1), (2, 2), (3, 3), (5, 5);
INSERT 0 4

--创建单表视图。
m_db=# CREATE VIEW v_upd1 AS SELECT * FROM t1;
CREATE VIEW
```



```
--通过视图对t1更新。
m_db=# UPDATE v_upd1 SET y1 = 13 where y1 = 3;
UPDATE 1

--创建多表视图。
m_db=# CREATE VIEW vv_upd AS SELECT * FROM t1, t2 WHERE x1 = x2;
CREATE VIEW

--通过连接视图对t1更新。
m_db=# UPDATE vv_upd SET y1 = 1 WHERE y2 = 1;
UPDATE 1

--删除SCHEMA。
m_db=# DROP SCHEMA upd_view;
NOTICE:  drop cascades to 4 other objects
DETAIL:  drop cascades to table t1
drop cascades to table t2
drop cascades to view v_upd1
drop cascades to view vv_upd
DROP SCHEMA
```

4.4.2.18.2 USE

功能描述

USE用于指定当前的模式，使数据库使用指定的模式作为默认（当前）模式。

注意事项

使用USE指定模式时，需要拥有模式的USAGE权限。当前用户可以通过执行“SHOW databases;”命令查看其拥有USAGE权限的模式。当用户没有对应模式的USAGE权限且进行指定模式操作时，会将当前模式置为空，且不会产生错误信息提示。用户可以通过调用database函数查看模式指定是否实际生效。

```
m_db=> USE test;
SET
m_db=> SELECT database();
ERROR:  function returned NULL
CONTEXT:  referenced column: database
```

语法格式

```
USE schema_name
```

参数说明

schema_name

要指定的模式名称。

取值范围：已存在的模式名称。

示例

```
m_db=# SELECT database();
 database
----------
 public
(1 row)

m_db=# SHOW DATABASES;
     Database
--------------------
```



```
 information_schema
 blockchain
 db4ai
 dbe_perf
 dbe_pldebugger
 dbe_pldeveloper
 dbe_sql_util
 m_schema
 pg_catalog
 pg_toast
 pkg_service
 public
 snapshot
 sqladvisor
 sys
(16 rows)

m_db=# USE m_schema;
SET
m_db=# SELECT database();
 database
----------
 m_schema
(1 row)
```

4.4.2.19 V

4.4.2.19.1 VACUUM

功能描述

VACUUM回收表或B-Tree索引中已经删除的行所占据的存储空间。在一般的数据库操作里，那些已经DELETE的行并没有从它们所属的表中物理删除；在完成VACUUM之前它们仍然存在。因此有必要周期地运行VACUUM，特别是在经常更新的表上。

注意事项

- 如果没有参数，VACUUM处理当前数据库里用户拥有相应权限的每个表。如果参数指定了一个表，VACUUM只处理指定的表。
- 要对一个表进行VACUUM操作，通常用户必须是表的所有者或者被授予了指定表VACUUM权限的用户，三权分立开关关闭时，默认系统管理员有该权限。数据库的所有者允许对数据库中除了共享目录以外的所有表进行VACUUM操作（该限制意味着只有系统管理员才能真正对一个数据库进行VACUUM操作）。VACUUM命令会跳过那些用户没有权限的表进行垃圾回收操作。
- VACUUM不能在事务块内执行。
- 建议生产数据库经常清理（至少每晚一次），以保证不断地删除失效的行。尤其是在增删了大量记录之后，对受影响的表执行VACUUM ANALYZE命令是一个很好的习惯。这样将更新系统目录为最近的更改，并且允许查询优化器在规划用户查询时有更好地选择。
- 不建议日常使用FULL选项，但是可以在特殊情况下使用。例如在用户删除了一个表的大部分行之后，希望从物理上缩小该表以减少磁盘空间占用。VACUUM FULL通常要比单纯的VACUUM收缩更多的表尺寸。FULL选项并不清理索引，所以推荐周期性的运行**REINDEX**命令。如果执行此命令后所占用物理空间无变化（未减少），请确认是否有其他活跃事务（删除数据事务开始之前开始的事务，并在VACUUM FULL执行前未结束）存在，如果有等其他活跃事务退出进行重试。

- VACUUM FULL通过重建表的方式将表内空闲空间归还给表空间，重建过程需要额外申请表中有效数据相当的存储空间。对于非段页式表，VACUUM FULL执行结束后，原表所占物理文件会被删除，原表所占的物理文件的空间会归还给操作系统；对于段页式表，VACUUM FULL执行结束后，原表所占的物理空间，会被归还给段页式数据文件，不会归还给操作系统。
- VACUUM会导致I/O流量的大幅增加，这可能会影响其他活动会话的性能。因此，有时候会建议使用基于开销的VACUUM延迟特性。
- 如果指定了VERBOSE选项，VACUUM将打印处理过程中的信息，以表明当前正在处理的表。各种有关当前表的统计信息也会打印出来。
- 当含有带括号的选项列表时，选项可以以任何顺序写入。如果没有括号，则选项必须按语法显示的顺序给出。
- VACUUM和VACUUM FULL时，会根据参数vacuum_defer_cleanup_age延迟清理行存表记录，即不会立即清理刚刚删除的元组。
- VACUUM ANALYZE先执行一个VACUUM操作，然后给每个选定的表执行一个ANALYZE。对于日常维护脚本而言，这是一个很方便的组合。
- 简单的VACUUM（不带FULL选项）只是简单地回收空间并且令其可以再次使用。这种形式的命令可以和对表的普通读写并发操作，因为没有请求排他锁。
  VACUUM FULL执行更广泛的处理，包括跨块移动行，以便把表压缩到最少的磁盘块数目里。这种形式要慢许多并且在处理的时候需要在表上施加一个排他锁。
- 同时执行多个VACUUM FULL可能出现死锁。
- 如果没有打开xc_maintenance_mode参数，那么VACUUM FULL操作将跳过所有系统表。
- 执行DELETE后立即执行VACUUM FULL命令不会回收空间。执行DELETE后再执行1000个非SELECT事务，或者等待1s后再执行1个事务，之后再执行VACUUM FULL命令空间才会回收。
- VACUUM FULL期间会对表加排他锁，不建议在业务高峰期运行VACUUM FULL，可能导致等待时间过长或者业务中断。

语法格式

- 回收空间并更新统计信息，对关键字顺序无要求。

```
VACUUM [ ( { FULL | FREEZE | VERBOSE | {ANALYZE  }} [,...] ) ]
    [ table_name [ (column_name [, ...] ) ] [ PARTITION ( partition_name ) | SUBPARTITION
( subpartition_name ) ] ];
```

- 仅回收空间，不更新统计信息。

```
VACUUM [ FULL ] [ FREEZE ] [ VERBOSE ] [ table_name
[ PARTITION ( partition_name ) | SUBPARTITION ( subpartition_name ) ] ];
```

- 回收空间并更新统计信息，且对关键字顺序有要求。

```
VACUUM [ FULL ] [ FREEZE ] [ VERBOSE ] { ANALYZE } [ VERBOSE ]
    [ table_name [ (column_name [, ...] ) ] ] [ PARTITION ( partition_name ) ];
```

参数说明

- **FULL**

  选择“FULL”清理，这样可以恢复更多的空间，但是需要耗时更多，并且在表上施加了排他锁。

  说明

  使用FULL参数会导致统计信息丢失，如果需要收集统计信息，请在VACUUM FULL语句中加上analyze关键字。

- **FREEZE**

  指定FREEZE相当于执行VACUUM时将vacuum_freeze_min_age参数设为0。

- **VERBOSE**

  为每个表打印一份详细的清理工作报告。

- **ANALYZE**

  更新用于优化器的统计信息，以决定执行查询的最有效方法。

  说明

  ustore分区表会触发vacuum。

- **table_name**

  要清理的表的名称（可以有模式修饰）。

  取值范围：要清理的表的名称。缺省时为当前数据库中的所有表。

- **column_name**

  要分析的具体的字段名称，需要配合analyze选项使用。

  取值范围：要分析的具体的字段名称。缺省时为所有字段。

  说明

  由于VACUUM ANALYZE语句的机制是依次执行VACUUM和ANALYZE，因此当column_name错误时，会存在VACUUM执行成功但ANALYZE执行失败的情况；对于分区表，则会出现对某个分区VACUUM执行成功之后ANALYZE执行失败的情况。

- **partition_name**

  要清理的表的一级分区名称。缺省时为所有一级分区。

- **subpartition_name**

  要清理的表的二级分区名称。缺省时为所有二级分区。

示例

```
--创建SCHEMA。
m_db=# CREATE SCHEMA tpcds;

--创建表tpcds.reason。
m_db=# CREATE TABLE tpcds.reason
(
  r_reason_sk       integer,
  r_reason_id       character(16),
  r_reason_desc     character(100)
);

--向表中插入多条记录。
m_db=# INSERT INTO tpcds.reason values(1,'AAAAAAAABAAAAAAA','reason 1'),
(2,'AAAAAAAABAAAAAAA','reason 2');

--在表tpcds.reason上创建索引。
m_db=# CREATE UNIQUE INDEX ds_reason_index1 ON tpcds.reason(r_reason_sk);

--对带索引的表tpcds.reason执行VACUUM操作。
m_db=# VACUUM (VERBOSE, ANALYZE) tpcds.reason;

--删除索引。
m_db=# DROP INDEX tpcds.ds_reason_index1 CASCADE;
m_db=# DROP TABLE tpcds.reason;
m_db=# DROP SCHEMA tpcds;
```


优化建议

- vacuum
  - VACUUM不能在事务块内执行。
  - 建议生产数据库经常清理（至少每晚一次），以保证不断地删除失效的行。尤其是在增删了大量记录后，对相关表执行VACUUM ANALYZE命令。
  - 不建议日常使用FULL选项，但是可以在特殊情况下使用。例如，一个例子就是在用户删除了一个表的大部分行之后，希望从物理上缩小该表以减少磁盘空间占用。

4.5 函数和操作符

本章介绍M-Compatibility下的函数和操作符。

⚠ 注意

《开发指南》中的“SQL参考 > 函数和操作符”章节中的函数和操作符在M-Compatibility下属于内部接口，禁止使用，使用这些函数和操作符可能产生预期外的结果。

《开发指南》中的“SQL参考 > 函数和操作符”章节中的以下函数在M-Compatibility中暂不支持：

表 4-27 和 MySQL 同名且 M-Compatibility 尚未支持的函数

isEmpty	overlaps	point	regexp_instr	regexp_like
regexp_replace	regexp_substr	stddev_pop	stddev_samp	var_pop
var_samp	variance	-	-	-

说明

函数regexp_instr、regexp_like、regexp_replace、regexp_substr在参数m_format_dev_version值为's2'及以上版本并且参数m_format_behavior_compat_options值包含'enable_conflict_funcs'的情况下使用会报错，并提示M-Compatibility数据库不支持；其他行为和《开发指南》中的“SQL参考 > 函数和操作符 > 字符串处理函数和操作符”章节中的同名函数保持一致。

表 4-28 系统信息函数

adm_hist_snapshot_func	adm_hist_sqlstat_func	adm_hist_sqltext_func	session_context

表 4-29 统计信息函数

get_global_transactions_prepared_xacts	get_global_transactions_running_xacts

表 4-30 系统管理函数

dbe_perf.get_full_sql_by_parent_id_and_timestamp	dbe_perf.get_active_session_profile	dbe_perf.get_global_slow_sql_by_timestamp	dbe_perf.get_global_full_sql_by_timestamp

表 4-31 其他函数

information_schema_pg_truetypid	information_schema._pg_truetypmod	summary_create	sys_context

表 4-32 内部函数

db4ai.archive_snapshot	db4ai.create_snapshot	db4ai.create_snapshot_internal	db4ai.manage_snapshot_internal	db4ai.prepare_snapshot
db4ai.prepare_snapshot_internal	db4ai.publish_snapshot	db4ai.purge_snapshot	db4ai.purge_snapshot_internal	db4ai.sample_snapshot

表 4-33 废弃函数

pgxc_get_stat_dirty_tables	gs_wlm_persistent_user_resource_info

4.5.1 操作符概述

操作符用于连接操作数或参数并返回结果。从语法上讲，运算符可以出现在操作数之前、操作数之后或两个操作数之间，因此，操作符可以分为一元运算符和二元运算符。

- 一元运算符
  ```
  运算符 操作数
  ```
- 二元运算符
  ```
  操作数1 运算符 操作数2
  ```

M-Compatibility模式下当前支持操作符及各操作符的简短说明如**表4-34**所示。

表 4-34 操作符描述

操作类型	操作符名称	操作数类型	描述
逻辑操作	NOT (!)	一元	逻辑非。
	AND（&&）	二元	逻辑与。
	OR（||）	二元	逻辑或。
	XOR	二元	逻辑异或。
位操作	&	二元	按位与。
	|	二元	按位或。
	^	二元	按位异或。
	<<	二元	左移。
	>>	二元	右移。
	~	一元	按位非。
模式匹配	[NOT] LIKE	二元	简单模式匹配。
比较操作	<	二元	小于。
	>	二元	大于。
	<=	二元	小于或等于。
	>=	二元	大于或等于。
	=	二元	等于。
	<>或!=	二元	不等于。
	<=>	二元	NULL安全等于。
	[NOT] BETWEEN AND	三元	判断操作数1是否大于等于操作数2且小于等于操作数3。
	IS	二元	判断是否与布尔值等价。
	IS NOT	二元	判断是否与布尔值不等价（IS的否定）。
	IS NOT NULL	一元（操作数 运算符）	判断是否非NULL。
	IS NULL	一元（操作数 运算符）	判断是否为NULL。
算术操作	+	一元/二元	一元表示正数，二元表示加法。
	-	一元/二元	一元表示负数，二元表示减法。

操作类型	操作符名称	操作数类型	描述
	*	二元	乘法操作符。
	/	二元	除法操作符。
	%或MOD	二元	整数除法，返回余数。
	DIV	二元	整数除法，返回商值。
类型转换	BINARY	一元	转换为BINARY类型。

操作符支持的数据类型

表 4-35 数据类型

类型分类	类型
布尔类型	BOOLEAN/BOOL
二进制类型	BINARY、VARBINARY、TINYBLOB、BLOB、MEDIUMBLOB、LONGBLOB
字符类型	CHAR、VARCHAR、TINYTEXT、TEXT、MEDIUMTEXT、LONGTEXT
日期时间类型	DATE、TIME、TIMESTAMP、DATETIME、YEAR
位串类型	BIT
数值类型	TINYINT、SMALLINT、MEDIUMINT、INT/INTEGER、BIGINT、NUMERIC/DECIMAL、DEC、FIXED、FLOAT、FLOAT4、FLOAT8、DOUBLE、TINYINT UNSIGNED、SMALLINT UNSIGNED、MEDIUMINT UNSIGNED、INT/INTEGER UNSIGNED、BIGINT UNSIGNED

数据类型详情参考**数据类型**。

操作符优先级

运算符从优先级从上到下越来越高，同一行上的运算符具有相同的优先级，具体见**表 4-36**。

表 4-36 操作符优先级列表

结合性	符号
left	OR
left	XOR
left	AND

结合性	符号
right	NOT
right	=
nonassoc	<、>、>=、<=
nonassoc	LIKE
nonassoc	BETWEEN
nonassoc	IN
left	|
left	&
left	<<、>>
left	+、-
left	* 、/ 、%、DIV、MOD
left	^
left	~
right	!

4.5.2 逻辑操作符

M-Compatibility模式下支持AND（&&）、OR（||）、NOT（!）、XOR四个逻辑操作符，详见**表4-34**。运算结果有三种结果，分别为TRUE、FALSE和空值。

表 4-37 逻辑操作符详情

运算符	操作数	含义
NOT/!	一元	逻辑非
AND/&&	二元	逻辑与
OR/||	二元	逻辑或
XOR	二元	逻辑异或

运算规则请参见**NOT操作符**和**AND、OR、XOR操作符**，表中的a和b代表逻辑表达式。

NOT 操作符

如果操作数为真，则计算结果为假。如果操作数为假，则计算结果为真。

取值范围：布尔类型

运算规则：

表 4-38 NOT 运算规则表（以 TRUE、FALSE、NULL 进行运算）

a	NOT a
TRUE	FALSE
FALSE	TRUE
NULL	空值

说明

“!”和“NOT”功能相同，但是“!”的优先级高于“NOT”。

示例：

```
m_db=# SELECT NOT TRUE;
 ?column?
----------
 f
(1 row)

m_db=# SELECT NOT FALSE;
 ?column?
----------
 t
(1 row)

m_db=# SELECT NOT NULL;
 ?column?
----------

(1 row)
```

AND、OR、XOR 操作符

AND、OR、XOR的运算逻辑如下：

表 4-39 AND、OR、XOR 运算规则表（以 TRUE、FALSE、NULL 进行运算）

a	b	a AND b的结果	a OR b的结果	a XOR b的结果
TRUE	TRUE	TRUE	TRUE	FALSE
TRUE	FALSE	FALSE	TRUE	TRUE
TRUE	NULL	空值	TRUE	空值
FALSE	FALSE	FALSE	FALSE	FALSE
FALSE	NULL	FALSE	空值	空值
NULL	NULL	空值	空值	空值

说明

操作符AND、XOR和OR具有交换性，即交换左右两个操作数，不影响其结果。

示例：

```
m_db=# SELECT TRUE AND TRUE;
 ?column?
----------
 t
(1 row)

m_db=# SELECT TRUE OR FALSE;
 ?column?
----------
 t
(1 row)

m_db=# SELECT TRUE XOR NULL;
 ?column?
----------

(1 row)
```

逻辑操作符规格约束

逻辑运算支持当前M-Compatibility数据库所有已支持的数据类型。不同大类型之间的数据操作时，先提升到大类型，再根据提升后的大类型的结果是否为0进行计算。提升规则如下表所示。

表 4-40 逻辑运算类型归类

操作符	左右入参类型	归类类型
XOR	所有类型	BIGINT
AND、OR、NOT	TINYINT、SMALLINT、MEDIUMINT、INT/INTEGER、BIGINT、TINYINT UNSIGNED、SMALLINT UNSIGNED、MEDIUMINT UNSIGNED、INT/INTEGER UNSIGNED、BIGINT UNSIGNED、BIT、YEAR、BOOL/LBOOLEAN	BIGINT
	其它	DOUBLE

4.5.3 位运算操作符

M-Compatibility支持的位运算符，包含按位与（&）、按位或（|）、按位异或（^）、左移（<<）、右移（>>）、按位取反（~）六个位运算运算符，详见表4-34。

优先级：按位取反 > 按位异或 > 左移 = 右移 > 按位与 > 按位或。

&操作符

描述：按位进行与操作。

示例：

```
m_db=# SELECT 1 & 1;
 ?column?
----------
 1
(1 row)
```

|操作符

描述：按位进行或操作。

示例：

```
m_db=# SELECT 1 | 1;
 ?column?
----------
 1
(1 row)
```

^操作符

描述：按位进行异或操作。

示例：

```
m_db=# SELECT -1 ^ -1;
 ?column?
----------
 0
(1 row)
```

<<操作符

描述：左移操作。

示例：

```
m_db=# SELECT 1 << 1;
 ?column?
----------
 2
(1 row)
```

>>操作符

描述：右移操作。

示例：

```
m_db=# SELECT -1 >> -1;
 ?column?
```



```
----------
 0
(1 row)
```

~操作符

描述：按位进行非操作。

示例：

```
m_db=# SELECT ~1;
      ?column?
----------------------
 18446744073709551614
(1 row)
```

位运算操作符规格约束

位运算操作符支持当前M-Compatibility数据库所有已支持的数据类型。如果操作数不为NULL则最终运算结果的类型为BIGINT UNSIGNED；如果其中一个操作数为NULL，则运算结果为NULL。

4.5.4 模式匹配操作符

数据库提供了[NOT] LIKE操作符和正则表达式模式匹配。

[NOT] LIKE

描述：判断字符串是否能匹配上LIKE后的模式字符串。如果字符串与提供的模式匹配，则LIKE表达式返回为真（NOT LIKE表达式返回假），否则返回为假（NOT LIKE表达式返回真）。支持含有0字符的操作数（字符串类型、二进制类型、位串类型）进行模式匹配操作。

匹配规则：

1. 此操作符只有在它的模式匹配整个串的时候才能成功。如果要匹配在串内任何位置的序列，该模式必须以百分号开头和结尾。
2. 下划线 _代表匹配任何单个字符； 百分号%代表任意串的通配符。
3. 要匹配文本里的下划线或者百分号，在提供的模式里相应字符必须前导逃逸字符。逃逸字符的作用是禁用元字符的特殊含义，缺省的逃逸字符是反斜线，也可以用ESCAPE子句指定一个不同的逃逸字符。
4. 要匹配逃逸字符本身，写两个逃逸字符。例如要写一个包含反斜线的模式常量，需要在SQL语句里写两个反斜线。
5. 是否区分大小写，与字符序有关，比如，默认的utf8mb4_general_ci不区分大小写。
6. 操作符~~等效于LIKE，!~~等效于NOT LIKE。

示例：

- 一般字符串匹配，字符序为utf8mb4_general_ci，不区分大小写。

```
m_db=# INSERT INTO t VALUES
('ab^CD&eF','ab^CD&eF','ab^CD&eF','ab^CD&eF','ab^CD&eF','ab^CD&eF');
m_db=# SELECT c FROM t WHERE c LIKE '%B^c%';
    c
----------
 ab^CD&eF
(1 row)
```


- BINARY/BLOB相关字段，字符序为utf8mb4_bin，区分大小写。

```
m_db=# INSERT INTO t VALUES (b'0110001', 'ab^CD&eF', 'ab^CD&eF');
m_db=# SELECT b FROM t WHERE b LIKE 'ab%';
     b
------------
 ab^CD&eF
(1 row)
```

- 用ESCAPE字符匹配_及%。

```
m_db=# INSERT INTO t VALUES ('ab_CD%eF','ab_CD%eF','ab_CD%eF','ab_CD%eF','ab_CD%eF','ab_CD
%eF');
m_db=# SELECT c FROM t WHERE c LIKE '%B\_c%';
    c
----------
 ab_CD%eF
(1 row)
```

- 指定ESCAPE字符。

```
m_db=# INSERT INTO t VALUES ('ab_CD%eF','ab_CD%eF','ab_CD%eF','ab_CD%eF','ab_CD%eF','ab_CD
%eF');
m_db=# SELECT c FROM t WHERE c LIKE '%B^_c%' ESCAPE '^';
    c
----------
 ab_CD%eF
(1 row)
```

[NOT] REGEXP

描述：REGEXP操作符用于正则表达式匹配。根据自己的模式是否匹配给定字符串而返回真或者假。下表描述了正则表达式操作符：

操作符名称	描述	语法说明
REGEXP	字符串是否与正则表达式匹配。	expr REGEXP pat
NOT REGEXP	字符串是否与正则表达式不匹配，REGEXP的否定。	expr NOT REGEXP pat

正则表达式语法：

- ^：匹配字符串的开头。
- $：匹配字符串的末尾。
- .：匹配任意字符（包括回车和换行符）。
- a*：匹配零个或多个“a”字符。
- a+：匹配一个或多个“a”字符。
- a?：匹配零个或一个“a”字符。
- de | abc：匹配“de”字符或“abc”字符。
- (abc)*：匹配零个或多个“abc”字符。
- a{1}，a{2,3}：a{1}表示重复“a”字符一次；a{2,3}表示重复“a”2~3次。
- [a-dX]，[^a-dX]：范围匹配。[a-dX]匹配a、b、c、d或X的字符。[^a-dX]表示匹配不是a、b、c、d或X的字符。
- [.characters.]：匹配用于校对元素的字符序列。允许的字符名称以及匹配的字符如下表所示。

Name	Character	Name	Character
NUL	0	SOH	1
STX	2	ETX	3
EOT	4	ENQ	5
ACK	6	BEL	7
alert	7	BS	10
backspace	\b'	HT	11
tab	\t'	LF	12
newline	\n'	VT	13
vertical-tab	\v'	FF	14
form-feed	\f'	CR	15
carriage-return	\r'	SO	16
SI	17	DLE	20
DC1	21	DC2	22
DC3	23	DC4	24
NAK	25	SYN	26
ETB	27	CAN	30
EM	31	SUB	32
ESC	33	IS4	34
FS	34	IS3	35
GS	35	IS2	36
RS	36	IS1	37
US	37	space	'
exclamation-mark	!'	quotation-mark	"'
number-sign	#'	dollar-sign	$'
percent-sign	%'	ampersand	&'
apostrophe	\''	left-parenthesis	('
right-parenthesis	)'	asterisk	*'
plus-sign	+'	comma	,'
hyphen	-'	hyphen-minus	-'

Name	Character	Name	Character
period	.'	full-stop	.'
slash	/'	solidus	/'
zero	0'	one	1'
two	2'	three	3'
four	4'	five	5'
six	6'	seven	7'
eight	8'	nine	9'
colon	:'	semicolon	;'
less-than-sign	<'	equals-sign	='
greater-than-sign	>'	question-mark	?'
commercial-at	@'	left-square-bracket	['
backslash	\\'	reverse-solidus	\\'
right-square-bracket	]'	circumflex	^'
circumflex-accent	^'	underscore	_'
low-line	_'	grave-accent	`'
left-brace	{'	left-curly-bracket	{'
vertical-line	|'	right-brace	}'
right-curly-bracket	}'	tilde	~'
DEL	177	-	-

- [=character_class=]：表示等价类。匹配具有相同排序校对值的所有字符，包括其本身。例如，如果o和(+)是等同类的成员，那么[[=o=]]、[[=(+)=]]和[o(+)]是同义词。
- [:character_class:]：表示匹配属于该类的所有字符的字符类。标准的类名如下表所示。

字符类名	描述	字符范围
alnum	文字数字字符	[0-9a-zA-Z]
alpha	文字字符	[a-zA-Z]
blank	空白字符	空白字符[\t]

cntrl	控制字符	[\x01-\x1F]
digit	数字字符	[0-9]
graph	图形字符	[^\x01-\x20]
lower	小写文字字符	[a-z]
print	图形字符	[^\x01-\x20]
punct	标点字符	[-!"#$%&'( )*+,./:;<=>?@[\ \]^_`{|}~]
space	空格、制表符、新行、和回车	[\n\r\t\x0B]
upper	大写文字字符	[A-Z]
xdigit	十六进制数字字符	[0-9a-fA-F]

- [[:<:]]，[[:>:]]：分别匹配字符串的开始和结束。

示例：

```
m_db=# SELECT 'fo\nfo' REGEXP '^fo$';
 ?column?
----------
 f
(1 row)

m_db=# SELECT 'fofo' REGEXP '^fo';
 ?column?
----------
 t
(1 row)

m_db=# SELECT 'Ban' REGEXP '^Ba*n';
 ?column?
----------
 t
(1 row)

m_db=# SELECT 'Bn' REGEXP '^Ba?n';
 ?column?
----------
 t
(1 row)

m_db=# SELECT 'pi' REGEXP 'pi|apa';
 ?column?
----------
 t
(1 row)

m_db=# SELECT 'aXbc' NOT REGEXP '^[a-dXYZ]$';
 ?column?
----------
 t
(1 row)
```

当m_format_behavior_compat_options开启enable_escape_string配置开关，支持的转义字符匹配包括：

转义字符	描述
\'	一个单引号字符。
\"	一个双引号字符。
\b	一个退格符。
\n	一个换行符。
\r	一个回车符。
\t	一个制表符。
\Z	ASCII 26（Control Z）。
\\	一个反斜杠符。
\%	一个%符。
_	一个_符。
\0	GaussDB报错：invalid byte sequence for encoding "UTF8": 0x00。

当m_format_behavior_compat_options关闭enable_escape_string配置开关，并设置standard_conforming_strings='off', escape_string_warning='off', backslash_quote='on'，支持的转义字符匹配包括：

转义字符	描述
\'	一个单引号字符。
\b	退格键。
\f	换页符，如C语言。
\n	换行符，如C语言。
\r	回车符，如C语言。
\t	水平制表符，如C语言
\uwxyz	（其中wxyz 是四个十六进制数字）十六进制值为0xwxyz的字符。
\xhhh	（其中hhh是十六进制数字的任何序列）十六进制值为0xhhh的字符。
\0	GaussDB报错：invalid byte sequence for encoding "UTF8": 0x00。
\xy	（其中xy是两个八进制数字）八进制值为0xy的字符。
\xyz	（其中xyz是三个八进制数字）八进制值为0xyz的字符。

⚠ 注意

词法语法解析按照字节流解析，当多字节字符中包含与'\', '\'', '\\'等符号一致的编码时，会导致与mysql行为不一致，建议暂时关闭转义符开关进行规避。

4.5.5 比较函数和比较操作符

比较操作符用于对两个数据的大小进行比较，并返回一个布尔类型的值。

比较操作符均为二元运算符，详见表4-34。M-Compatibility模式下提供的比较操作符请参见表4-34。

说明

- 不等于的计算优先级高于等于。
- M-Compatibility模式下目前只支持btree索引和ubtree索引。
- <>和<=>不支持索引扫描。
- <=>不支持hash连接和合并连接。
- <=>不支持行比较。
- <、<=、=、>=、>、<>操作符支持行表达式与NULL值比较。
- 支持含有0字符的操作数（字符串类型、二进制类型、位串类型）进行<、>、<=、>=、=、!=、<>、<=>及BETWEEN AND操作。
- IS NULL、ISNULL操作支持行表达式作为参数使用。

```
m_db=# SELECT (1,2) <=> row(2,3);
ERROR:  could not determine interpretation of row comparison operator <=>
LINE 1: select (1,2) <=> row(2,3);
                     ^
HINT:  unsupported operator.
m_db=# SELECT (1,2) < NULL;
 ?column?
----------

(1 row)

m_db=# SELECT (1,2) <> NULL;
 ?column?
----------

(1 row)
m_db=# SELECT (1, 2)  IS NULL;
 ?column?
----------
 f
(1 row)

m_db=# SELECT ISNULL((1, 2));
 ?column?
----------
 f
(1 row)
```

比较操作符

- <操作符

  描述：小于操作。

  示例：

  ```
  m_db=# SELECT 2 < 2;
   ?column?
  ```



```
----------
 f
(1 row)
```

- >操作符

  描述：大于操作符。

  示例：

```
m_db=# SELECT 2 > 2;
 ?column?
----------
 f
(1 row)
```

- <=操作符

  描述：小于等于操作符。

  示例：

```
m_db=# SELECT 0.1 <= 0.2;
 ?column?
----------
 t
(1 row)
```

- >=操作符

  描述：大于等于操作符。

  示例：

```
m_db=# SELECT 2 >= 2;
 ?column?
----------
 t
(1 row)
```

- =操作符

  描述：等于操作符。

  示例：

```
m_db=# SELECT 2 = 2;
 ?column?
----------
 t
(1 row)
```

- <>或!=操作符

  描述：不等于操作符。

  示例：

```
m_db=# SELECT 1 <> 2;
 ?column?
----------
 t
(1 row)
```

- <=>操作符

  描述：NULL安全等于操作符，此操作符执行与=操作符类似的相等比较，但NULL与NULL进行NULL安全等于比较时返回t，否则返回f。

  示例：

```
m_db=# SELECT 1 <=> 1, 1 <=> NULL, NULL <=> NULL;
 ?column? | ?column? | ?column?
----------+----------+----------
 t        | f        | t
(1 row)
```


- BETWEEN AND操作符

  描述：判断操作数1是否大于等于操作数2且小于等于操作数3

  示例：

```
m_db=# select 2 between 1 and 3;
 ?column?
----------
 t
(1 row)
```

- IS操作符

```
expr IS boolean_value
```

  描述：判断expr是否等价于boolean_value，如果等价则返回真（t），否则返回假（f）。其中boolean_value可以是TRUE、FALSE、NULL或UNKNOWN。

  示例：

```
m_db=# SELECT 1 IS TRUE, 0 IS FALSE, NULL IS UNKNOWN;
 is true | is false | ?column?
---------+----------+----------
 t       | t        | t
(1 row)
```

- IS NOT操作符

```
expr IS NOT boolean_value
```

  描述：判断expr是否不等价于boolean_value，如果不等价则返回真（t），否则返回假（f）。其中boolean_value可以是TRUE、FALSE、NULL或UNKNOWN。

  示例：

```
m_db=# SELECT 1 IS NOT UNKNOWN, 0 IS NOT UNKNOWN, NULL IS NOT UNKNOWN;
 ?column? | ?column? | ?column?
----------+----------+----------
 t        | t        | f
(1 row)
```

- IS NOT NULL操作符

```
expr IS NOT NULL
```

  描述：判断expr是否为非NULL。如果是则返回真（t），否则返回假（f）。

  示例：

```
m_db=# SELECT 1 IS NOT NULL, 0 IS NOT NULL, NULL IS NOT NULL;
 ?column? | ?column? | ?column?
----------+----------+----------
 t        | t        | f
(1 row)
```

- IS NULL操作符

```
expr IS NULL
```

  描述：判断expr是否为NULL。如果是则返回真（t），否则返回假（f）。

  示例：

```
m_db=# SELECT 1 IS NULL, 0 IS NULL, NULL IS NULL;
 ?column? | ?column? | ?column?
----------+----------+----------
 f        | f        | t
(1 row)
```

比较函数

- COALESCE(value1, value2, ...)

  描述：返回参数列表中第一个非NULL值。如果没有非NULL值，该函数返回空值。

  返回值类型：由类型推导矩阵决定，会根据入参的类型决定返回值类型。

示例：

```
m_db=# SELECT COALESCE(NULL, 1);
 coalesce
----------
        1
(1 row)

m_db=# SELECT COALESCE(NULL, NULL, NULL);
 coalesce
----------

(1 row)
```

- GREATEST(value1, value2, ...)

  描述：返回参数列表中的最大值。

  参数：至少需要两个参数。

  示例：

```
m_db=# SELECT GREATEST(34.0, 3.0, 5.0, 767.0);
 greatest
----------
    767.0
(1 row)

m_db=# SELECT GREATEST('B', 'A', 'C');
 greatest
----------
 C
(1 row)
```

- INTERVAL(N, N1, N2, N3, ...)

  描述：返回N与其他参数组成的列表(N1，N2，N3，...）的比较情况。

  参数：至少需要两个参数，所有传入的参数被视为数字类型。

  返回值类型：SMALLINT

  - 如果N为NULL，则返回-1。
  - 如果N<N1，则返回0。
  - 如果存在正整数j，对于任意正整数i≤j，都有$N_i \le N \le N_{j+1}$，则返回j。

  示例：

```
m_db=# SELECT INTERVAL(24, 2, 5, 14, 26, 28, 46);
 interval
----------
        3
(1 row)

m_db=# SELECT INTERVAL(24, 29, 5, 14);
 interval
----------
        0
(1 row)
```

- ISNULL(expr)

  描述：如果参数expr为NULL，那么该函数返回t，否则返回f。

  参数：任意入参。

  返回值类型：BOOLEAN

  示例：

```
m_db=# SELECT ISNULL('2023-01-01');
 isnull
--------
 f
```



```
(1 row)

m_db=# SELECT ISNULL(NULL);
 isnull
--------
 t
(1 row)
```

- LEAST(value1, value2, ...)

  描述：返回参数列表中的最小值。

  参数：至少需要两个参数。

  示例：

```
m_db=# SELECT LEAST(34.0, 3.0, 5.0, 767.0);
 least
-------
   3.0
(1 row)

m_db=# SELECT LEAST('B', 'A', 'C');
 least
-------
 A
(1 row)
```

4.5.6 数字操作函数和算术操作符

算术操作符

M-Compatibility模式下的算术操作符包含+、-、*、/、%、-（取负操作）、DIV，详见表4-34。

- +

  描述：进行加法操作。

  示例：

```
/* plus constant */
m_db=# SELECT 2+3;
 ?column?
----------
        5
(1 row)

m_db=# SELECT +3;
 ?column?
----------
        3
(1 row)
```

- -

  描述：进行减法操作或表示负数。

  说明

  如果两个操作数都是整数并且其中任何一个是无符号的，则结果是无符号整数。对于减法，如果启用 NO_UNSIGNED_SUBTRACTION SQL 模式，则即使任何操作数是无符号数，结果也会是有符号数。

  示例：

```
/* minus constant */
m_db=# SELECT 3-2;
 ?column?
----------
        1
```



```
(1 row)

m_db=# SELECT -3;
 ?column?
----------
       -3
(1 row)
```

- *

  描述：进行乘法操作。

  示例：

```
/* multiply constant */
m_db=# SELECT 2*3;
 ?column?
----------
        6
(1 row)

m_db=# SELECT -2*3;
 ?column?
----------
       -6
(1 row)
```

- /

  描述：进行除法操作。

  说明

  在使用 / 进行除法运算时，使用两个精确值操作数时结果的小数位数是第一个操作数的小数位数加上 div_precision_increment 系统变量的值（默认为 4）。例如，表达式 8.85/0.093 的结果有六个小数位 (95.161290)。

  示例：

```
/* division constant */
m_db=# SELECT 4/2;
 ?column?
----------
   2.0000
(1 row)

m_db=# SELECT 8.85/0.093;
 ?column?
------------
 95.161290
(1 row)
```

- %或MOD

  描述：进行取余操作。

  示例：

```
/* mod constant */
m_db=# SELECT 7%4;
 ?column?
----------
        3
(1 row)

m_db=# SELECT 7 MOD 4;
 ?column?
----------
        3
(1 row)
```

- DIV

  描述：进行整除操作。

示例：

```
/* DIV test */
m_db=# SELECT 2.1 DIV 1;
 ?column?
----------
        2
(1 row)

m_db=# SELECT 1 DIV 2;
 ?column?
----------
        0
(1 row)
```

算术操作符规格约束

操作符支持不同大类之间的数据进行操作。数据进行操作前，先将数据提升到相应的大类后再计算出相应的结果。操作符中操作数提升规则以及结果类型规格如下文所述。

表 4-41 参数提升规则

左右入参类型	归类类型
TINYINT、SMALLINT、MEDIUMINT、INT/INTEGER、BIGINT、DATE	BIGINT
TINYINT UNSIGNED、SMALLINT UNSIGNED、MEDIUMINT UNSIGNED、INT/INTEGER UNSIGNED、BIGINT UNSIGNED、BIT、YEAR	BIGINT UNSIGNED
TIME、TIMESTAMP、DATETIME	不带毫秒时，对应BIGINT 带毫秒时，对应NUMERIC
UNKNOWN、CHAR、VARCHAR、BINARY、VARBINARY、TINYTEXT TEXT、MEDIUMTEXT、LONGTEXT, TINYBLOB、BLOB、MEDIUMBLOB、LONGBLOB, FLOAT、FLOAT4、FLOAT8	FLOAT8
NUMERIC	NUMERIC
其他默认	BIGINT

表 4-42 加法、乘法、取模结果类型

归类类型组合（从上到下优先级从高到低）	最终结果类型
FLOAT8 + 任意	FLOAT8
NUMERIC + 任意	NUMERIC
BIGINT UNSIGNED + 任意	BIGINT UNSIGNED
其他	BIGINT

表 4-43 减法结果类型

归类类型组合（从上到下优先级从高到低）	最终结果类型
FLOAT8 + 任意	FLOAT8
NUMERIC + 任意	NUMERIC
BIGINT UNSIGNED + 任意	BIGINT UNSIGNED或BIGINT (如果GUC参数SQL_MODE开启了NO_UNSIGNED_SUBTRACTION时，结果类型是BIGINT、否则返回BIGINT UNSIGNED)
其他	BIGINT

表 4-44 除法结果类型

归类类型组合（从上到下优先级从高到低）	最终结果类型
FLOAT8 + 任意	float8
其他	numeric

表 4-45 取负结果类型

入参类型	结果类型
TINYINT、SMALLINT、MEDIUMINT、INT/INTEGER、BIGINT、TINYINT UNSIGNED、SMALLINT UNSIGNED、MEDIUMINT UNSIGNED、INT/INTEGER UNSIGNED、BIGINT UNSIGNED、BIT、YEAR	BIGINT
UNKNOWN、CHAR、VARCHAR、BINARY、VARBINARY、TINYTEXT TEXT、MEDIUMTEXT、LONGTEXT, TINYBLOB、BLOB、MEDIUMBLOB、LONGBLOB、DATE、DATETIME、TIMESTAMP、TIME、FLOAT、FLOAT4、FLOAT8	FLOAT8
NUMERIC	NUMERIC
其他默认	FLOAT8

表 4-46 整除结果类型

入参类型	结果类型
TINYINT、SMALLINT、MEDIUMINT、INT/INTEGER、BIGINT、 TINYINT UNSIGNED、SMALLINT UNSIGNED、MEDIUMINT UNSIGNED、INT/INTEGER UNSIGNED、BIGINT UNSIGNED、BIT、YEAR	BIGINT(入参转BIGINT大类型作除法运算，结果取整返回BIGINT类型)
UNKNOWN、CHAR、VARCHAR、BINARY、VARBINARY、TINYTEXT TEXT、MEDIUMTEXT、LONGTEXT、 TINYBLOB、BLOB、MEDIUMBLOB、LONGBLOB、DATE、DATETIME、TIMESTAMP、TIME、FLOAT、FLOAT4、FLOAT8、NUMERIC	BIGINT (入参转DOUBLE大类型作除法运算，结果取整返回BIGINT类型)

说明

在当算数操作符的操作数为返回值是datetime、timestamp、time类型的函数，或是datetime和time类型的字面量，如timestamp'2000-01-01 00:00:00'、time'08:00:00'的时候，会忽略小数点后的内容，即忽略掉毫秒信息。

数字操作函数

- ABS(expr)

  描述：求数值表达式的绝对值。

  参数：expr，数值表达式。

  返回值类型：BIGINT UNSIGNED、BIGINT、DOUBLE、DECIMAL。与数值表达式的数据类型相同。

  示例：

  ```
  m_db=# SELECT ABS(-1.3);
   abs
  -----
   1.3
  (1 row)
  ```

- ACOS(X)

  描述：返回X的反余弦值，即余弦为X的值。

  参数：X，取值范围不在[-1、1]区间内，该函数返回空值。

  返回值类型：DOUBLE

  示例：

  ```
  m_db=# SELECT ACOS(0);
          acos
  --------------------
   1.5707963267948966
  (1 row)
  ```

- ASIN(X)

  描述：返回X的反正弦值，即正弦为X的值。

  参数：X，取值范围不在[-1、1]区间内，该函数返回空值。

  返回值类型：DOUBLE

示例：

```
m_db=# SELECT ASIN(0);
 asin
------
    0
(1 row)
```

- ATAN(X)

  描述：返回X的反正切值，即正切值为X的值。

  返回值类型：DOUBLE

  示例：

  ```
  m_db=# SELECT ATAN(-10);
         atan
  ---------------------
   -1.4711276743037347
  (1 row)
  ```

- ATAN(Y、X)

  描述：返回X与Y的反正切值。它类似于计算Y/X的反正切值，两个参数的符号用于确定结果所在的象限。

  返回值类型：DOUBLE

  示例：

  ```
  m_db=# SELECT ATAN(-2、2);
         atan
  ---------------------
   -0.7853981633974483
  (1 row)
  ```

- ATAN2(Y、X)

  描述：该函数的功能用法完全等同于**ATAN(Y,X)**。

  示例：

  ```
  m_db=# SELECT ATAN2(-2、2);
         atan2
  ---------------------
   -0.7853981633974483
  (1 row)
  ```

- CEIL(INT X)

  描述：CEILING函数的别名，返回不小于X的最小整数值。如果X为NULL，则返回NULL。

  返回值类型：BIGINT、BIGINT UNSIGNED、DOUBLE、DECIMAL

  说明

  - 此函数在参数m_format_dev_version值为's2'或以上版本且参数m_format_behavior_compat_options值包含'enable_conflict_funcs'的情况下为M-Compatibility兼容模式数据库实现行为，即本章节描述的行为；其他行为和《开发指南》中的“SQL参考 > 函数和操作符 > 数字操作函数和操作符”章节中的ceil函数保持一致。
  - 在非M-Compatibility兼容模式数据库实现行为下，入参类型为M-Compatibility兼容模式所独有的数据类型时（例如BIGINT UNSIGNED），使用该函数结果可能存在异常。

  示例：

  ```
  m_db=# SELECT CEIL(-5.5);
   ceil
  ------
     -5
  (1 row)
  ```


- CEILING(X)

  描述：返回大于等于X的最小整数。

  返回值类型：BIGINT、DOUBLE、DECIMAL。

  示例：

```
m_db=# SELECT CEILING(-5.5);
 ceiling
---------
      -5
(1 row)
```

- CONV(TEXT N, INT from_base, INT to_base)

  描述：在不同数基之间转换数字，比如从10进制转为2进制。输入与输出的进制值在2~36范围内（包括2和36）。如果from_base是负数，则N被作为有符号数输入，否则作为无符号数输入；若to_base是负数，将输出指定为有符号数，否则指定为无符号数。

  返回值类型：TEXT

  示例：

```
m_db=# select CONV(10010011, 2, 4);
 conv
------
 2103
(1 row)

m_db=# select CONV('111ypf9z', 36, 2);
                 conv
---------------------------------------
 1001011000111101100010110111110010111
(1 row)
```

- COS(X)

  描述：返回指定弧度X的余弦值。

  返回值类型：DOUBLE

  示例：

```
m_db=# SELECT COS(0);
 cos
-----
   1
(1 row)
```

- COT(X)

  描述：返回指定弧度X的余切值。

  返回值类型：DOUBLE

  示例：

```
m_db=# SELECT COT(1);
        cot
--------------------
 0.6420926159343306
(1 row)
```

- CRC32(TEXT str)

  描述：用于计算循环冗余值。如果str为NULL，则返回NULL；否则，在计算冗余操作后，返回INT UNSIGNED值。

  返回值类型：INT UNSIGNED

  示例：

```
m_db=# SELECT CRC32('a');
   crc32
```



```
------------
 3904355907
(1 row)
```

- DEGREES(X)

  描述：将参数X从弧度转换为度数并返回。

  返回值类型：DOUBLE

  示例：

```
m_db=# SELECT DEGREES(PI());
 degrees
---------
     180
(1 row)
```

- EXP(X)

  描述：返回自然底数e（2.7182818...）的X次幂。

  返回值类型：DOUBLE

```
m_db=# SELECT EXP(2);
       exp
------------------
 7.38905609893065
(1 row)
```

- FLOOR(X)

  描述：返回小于等于X的最大整数。

  返回值类型：BIGINT UNSIGNED、BIGINT、DOUBLE、DECIMAL。

  示例：

```
m_db=# SELECT FLOOR(5.5);
 floor
-------
     5
(1 row)
```

- LN(X)

  描述：返回X的自然对数，即X的以e（2.7182818...）为底的对数。

  返回值类型：DOUBLE

  示例：

```
m_db=# SELECT LN(2);
         ln
--------------------
 0.6931471805599453
(1 row)
```

- LOG([B、]X)

  描述：返回以B为底X的对数。缺省B参数时，以e（2.7182818...）为底。

  返回值类型：DOUBLE

  示例：

```
m_db=# SELECT LOG(2);
        log
--------------------
 0.6931471805599453
(1 row)

m_db=# SELECT LOG(10、2);
         log
---------------------
 0.30102999566398114
(1 row)
```


- LOG10(X)

  描述：返回以10为底X的对数。

  返回值类型：DOUBLE

```
m_db=# SELECT LOG10(2);
       log10
--------------------
 0.3010299956639812
(1 row)
```

- LOG2(X)

  描述：返回以2为底X的对数。

  返回值类型：DOUBLE

```
m_db=# SELECT LOG2(2);
 log2
------
    1
(1 row)
```

- MOD(X, Y)

  描述：进行取余操作。

  返回值类型：INT、DOUBLE、DECIMAL。

  示例：

```
m_db=# SELECT MOD(5, 3);
 ?column?
----------
        2
(1 row)

m_db=# SELECT MOD(7, 4);
 ?column?
----------
        3
(1 row)
```

- PI()

  描述：返回π的值（圆周率）。

  返回值类型：DOUBLE

  示例：

```
m_db=# SELECT PI();
        pi
-------------------
 3.141592653589793
(1 row)
```

- POW(X、Y)

  描述：返回X的Y次方。

  返回值类型：DOUBLE

  示例：

```
m_db=# SELECT POW(2、3);
 pow
-----
   8
(1 row)
```

- POWER(X、Y)

  描述：返回X的Y次方。该函数是**POW(X, Y)**函数的别名。

  返回值类型：DOUBLE

示例：

```
m_db=# SELECT POWER(2、3);
 power
-------
     8
(1 row)
```

- RADIANS(X)

  描述：将参数X从度数转换为弧度并返回。

  返回值类型：DOUBLE

  示例：

```
m_db=# SELECT RADIANS(180);
      radians
--------------------
 3.141592653589793
(1 row)
```

- RAND([seed])

  描述：返回随机浮点数，范围[0、1)。

  参数：seed，随机数种子。指定相同的seed值该函数会生成相同的随机数。

  返回值类型：DOUBLE

  示例：

```
m_db=# SELECT RAND();
        rand
--------------------
 0.5320404642261565
(1 row)

m_db=# SELECT RAND(1);
        rand
---------------------
 0.40540353712197724
(1 row)
```

- SIGN(X)

  描述：返回X的符号对应的值，即正值返回1，零值返回0，负值返回-1。

  返回值类型：BIGINT

  示例：

```
m_db=# SELECT SIGN(5.2);
 sign
------
    1
(1 row)

m_db=# SELECT SIGN(0);
 sign
------
    0
(1 row)

m_db=# SELECT SIGN(-5.2);
 sign
------
   -1
(1 row)
```

- SIN(X)

  描述：返回指定弧度X的正弦值。

  返回值类型：DOUBLE

  示例：


```
m_db=# SELECT SIN(PI());
          sin
------------------------
 1.2246467991473532e-16
(1 row)
```

- SQRT(X)

  描述：返回X的平方根。

  返回值类型：DOUBLE

  示例：

```
m_db=# SELECT SQRT(0.64);
 sqrt
------
  0.8
(1 row)
```

- TAN(X)

  描述：返回指定弧度X的正切值。

  返回值类型：DOUBLE

  示例：

```
m_db=# SELECT TAN(PI());
           tan
-------------------------
 -1.2246467991473532e-16
(1 row)
```

- TRUNCATE(X、D)

  描述：返回X保留D位小数的结果。

  参数：如果参数D为0，则返回X的整数部分。如果参数D为负数，则使X的小数点左侧D位数变为0。

  返回值类型：BIGINT、DOUBLE、DECIMAL。

  示例：

```
m_db=# SELECT TRUNCATE(3.2312、1);
 truncate
----------
      3.2
(1 row)

m_db=# SELECT TRUNCATE(123.2312、-1);
 truncate
----------
      120
(1 row)
```

扩展场景

- 算术操作符优先级。取负 > 乘、除、取余、整除 > 加、减。

  示例1：乘法操作优先级高于加法操作

```
m_db=# SELECT 1 + 2 * 3;
 ?column?
----------
        7
(1 row)
```

  示例2：取余操作优先级高于加法操作

```
m_db=# SELECT 1 + 2 % 3;
 ?column?
----------
        3
(1 row)
```


示例3：取模操作优先级高于加法操作

```
m_db=# SELECT 1 + 2 MOD 3;
 ?column?
----------
        3
(1 row)
```

示例4：整除操作优先级高于加法操作

```
m_db=# SELECT 1 + 7 DIV 3;
 ?column?
----------
        3
(1 row)
```

4.5.7 字符串函数

M-Compatibility提供字符串函数对字符串进行处理。

- 对于对字符串位置进行操作的函数，第一个位置的编号为1。
- 对于使用了长度作为参数的函数，非整数的参数将四舍五入到最接近的整数。
- 如果数据库字符集是SQL_ASCII，数据库将把字节值0-127根据 ASCII标准解释，而字节值128-255则当作无法解析的字符。如果设置为SQL_ASCII，数据库将无法转换或者校验非ASCII字符，导致CHAR_LENGTH、FIND_IN_SET、LEFT、LOWER、REVERSE、RIGHT、STRCMP、TRIM等字符串函数产生未预期的结果。如果使用了任意非ASCII数据，都不建议将数据库的字符集设置为SQL_ASCII。

ASCII

ASCII(str)

描述：返回字符串str最左侧字符的ASCII码值，str为空字符串时返回0，str为NULL时返回NULL。

返回值类型：INT

示例：

```
m_db=# SELECT ASCII('');
 ascii
-------
     0
(1 row)

m_db=# SELECT ASCII('abc');
 ascii
-------
    97
(1 row)

m_db=# SELECT ASCII(NULL);
 ascii
-------

(1 row)
```

BIN

BIN(INT N)

描述：返回N的二进制值的字符串表示形式，与CONV(N,10,2)功能相同。

返回值类型：TEXT

示例：

```
m_db=# SELECT BIN(12);
 bin
------
 1100
(1 row)
```

BIT_LENGTH

BIT_LENGTH(str)

描述：返回字符串str的比特位长度。

返回值类型：BIGINT

示例：

```
m_db=# SELECT BIT_LENGTH('text');
 bit_length
------------
         32
(1 row)
```

CHAR

CHAR(INT N,... [USING charset_name])

描述：将每个参数N通过ASCII码转换成对应的字符。其中charset_name为字符集名称。

返回值类型：TEXT/BLOB

示例：

```
m_db=# SELECT CHAR(71,97,117,115,115);
 char
-------
 Gauss
(1 row)

m_db=# SELECT CHAR(71,97,117,115,115 using utf8);
 char
-------
 Gauss
(1 row)
```

CHAR_LENGTH

CHAR_LENGTH(str)

描述：返回字符串str包含的字符数，多字节的字符会被计为1个字符。例如，输入2个2字节的字符，LENGTH()返回4，而CHAR_LENGTH()返回2。

返回值类型：BIGINT

示例：

```
m_db=# SELECT CHAR_LENGTH('text');
 char_length
-------------
           4
(1 row)
```


CHARACTER_LENGTH

CHARACTER_LENGTH(str)

描述：与**CHAR_LENGTH**功能、用法相同。

返回值类型：BIGINT

示例：

```
m_db=# SELECT CHARACTER_LENGTH('text');
 character_length
------------------
                4
(1 row)
```

COMPRESS

COMPRESS (TEXT string_to_compress)

描述：压缩一个字符串，返回二进制字符串类型的结果。

返回值类型：LONGBLOB

示例：

```
m_db=# SELECT HEX(COMPRESS('abcde'));
       HEX(COMPRESS('abcde'))
------------------------------------
 05000000789C4B4C4A4E49050005C801F0
(1 row)
```

CONCAT

CONCAT(str1, str2, ..., strN)

描述：接受若干个字符串，按顺序将其拼接并返回拼接后的字符串。如果有一个字符串为NULL，则返回NULL。

返回值类型：TEXT/BLOB

示例：

```
m_db=# SELECT CONCAT('hello', 'world', '!');
   concat
-------------
 helloworld!
(1 row)

m_db=# SELECT CONCAT('hello', NULL, '!');
 concat
--------

(1 row)
```

CONCAT_WS

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

描述：把多个字符串按顺序进行拼接，相邻字符串使用separator进行分割。当separator为NULL时，返回NULL。当待拼接的字符串为NULL时，将被忽略。

返回值类型：TEXT/BLOB

示例：

```
m_db=# SELECT CONCAT_WS(',', 'hello', NULL, 'world', '!');
   concat_ws
---------------
 hello,world,!
(1 row)

m_db=# SELECT CONCAT_WS(NULL, 'hello', NULL, 'world', '!');
 concat_ws
-----------

(1 row)
```

ELT

ELT(INT N, TEXT str1, TEXT str2, TEXT str3, ...)

描述：返回字符串列表的第N个元素。如果N为空、小于1、大于字符串总个数或者N位置处的字符串为NULL，则返回NULL。

参数说明：

N：用于指定位置。

str1, str2, str3, …：表示字符串参数列表，参数最大个数为8192。

返回值类型：TEXT/BLOB

示例：

```
m_db=# SELECT ELT (1,'a');
 elt
-----
 a
(1 row)

m_db=# SELECT ELT (2,'a', 'b');
 elt
-----
 b
(1 row)
```

EXPORT_SET

EXPORT_SET (INT bits, TEXT on, TEXT off[, TEXT separator[, INT length]])

描述：根据参数的二进制位生成一个使用指定分隔符拼接的字符串，如果length不指定或者指定范围不在[0,64]，length采用64位。将bits转换成二进制，如果bit位是1，采用on指定的字符串，如果bit位是0，采用off指定的字符串，中间使用separator指定的字符串隔离，separator不指定默认使用逗号（,）为隔离符。返回值的长度受GUC参数max_allowed_packet的影响，最大取值为Min(max_allowed_packet, 1073741819)。

参数说明：如表4-47所示。

表 4-47 EXPORT_SET 函数参数说明

名称	描述
bits	必选，一个数字， bits中的比特值按照从右到左的顺序依次决定on或者off出现在该位置。

名称	描述
on	必选，当对应bit的比特位值为1时，使用的字符串。
off	必选，当对应bit的比特位值为0时，使用的字符串。
separator	可选，分隔字符串，默认值为逗号（，）。
length	可选，集合的元素个数，默认值为64。

返回值类型：TEXT

示例：

```
m_db=# SELECT EXPORT_SET(5, 'Y', 'N');
                                                          export_set
----------------------------------------------------------------------------------------------------------------------------------
---

Y,N,Y,N,N,N,N,N,N,N,N,N,N,N,N,N,N,N,N,N,N,N,N,N,N,N,N,N,N,N,N,N,N,N,N,N,N,N,N,N,N,N,N,N,N,N,N,N,N,N,
N,N,N,N,N,N,N,N,N,N,N,N,N,N

m_db=# SELECT EXPORT_SET(5,'Y','N','|');
                                                          export_set
----------------------------------------------------------------------------------------------------------------------------------
---
 Y|N|Y|N|N|N|N|N|N|N|N|N|N|N|N|N|N|N|N|N|N|N|N|N|N|N|N|N|N|N|N|N|N|N|N|N|N|N|N|N|N|N|N|N|N|N|N|N|N|N|
N|N|N|N|N|N|N|N|N|N|N|N|N|N
(1 row)

m_db=# SELECT EXPORT_SET(5,'Y','N','|', 5);
 export_set
------------
 Y|N|Y|N|N
(1 row)
```

FIELD

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

描述：field函数返回str在str1、str2、str3等列表中的位置。从1开始递增，返回0表示str不在str1、str2、str3等列表中。

返回值类型：bigint

- 如果str为NULL或者不在str1、str2、str3等列表时，直接返回0。
- 函数入参全部为数字时按照数字进行比较，入参全部为非数字类型时按照字符串进行比较，入参存在数字和非数字混合时按照DOUBLE类型进行比较。

示例：

```
m_db=# SELECT FIELD(1,2,1);
 field
-------
     2
(1 row)

m_db=# SELECT FIELD('a','b','a');
 field
-------
     2
(1 row)
```

```
m_db=# SELECT FIELD('a',1,'b','a');
 field
------
     2
(1 row)
```

FIND_IN_SET

FIND_IN_SET(str, strlist)

描述：返回字符串str在字符串列表strlist中的位置，从1开始。字符串列表是1个包含若干个子字符串的字符串，各个子字符串之间使用逗号“,”进行分隔。

返回值类型：INT

- 如果strlist中不包含str，返回0。
- 如果str或strlist为NULL，则返回NULL。
- 如果str中包含逗号，则返回0。

示例：

```
m_db=# SELECT FIND_IN_SET('e', 'a,b,c;d,e');
 find_in_set
-------------
           4
(1 row)

m_db=# SELECT FIND_IN_SET('a,', 'a,b,c;d,e');
 find_in_set
-------------
           0
(1 row)

m_db=# SELECT FIND_IN_SET(NULL, 'a,b,c;d,e');
 find_in_set
-------------

(1 row)
```

FORMAT

FORMAT(INT X , INT D [, TEXT locale])

描述：将数字X格式化为指定格式（#,###,###.##），并按照指定的D进行四舍五入到小数点后D位，将结果作为字符串返回。如果D为0，则结果没有小数点或小数部分，如果X或D为NULL，则返回结果NULL。

参数：

X：必选，表示被格式化的数字。

D：必选，表示保留的小数位数。

locale：可选，表示特定的语言环境。

返回值类型：TEXT

说明

- 此函数在参数m_format_dev_version值为's2'或以上版本且参数m_format_behavior_compat_options值包含'enable_conflict_funcs'的情况下为M-Compatibility兼容模式数据库实现行为，即本章节描述的行为；其他行为和《开发指南》中的“SQL参考 > 函数和操作符 > 数字操作函数和操作符”章节中的ceil函数保持一致。
- 在非M-Compatibility兼容模式数据库实现行为下，入参类型为M-Compatibility兼容模式所独有的数据类型时（例如BIGINT UNSIGNED），使用该函数结果可能存在异常。

示例：

```
m_db=# SELECT FORMAT(12332.123456, 4);
   format
-------------
 12,332.1235
(1 row)

m_db=# SELECT FORMAT(12332.123456,4,'de_DE');
   format
-------------
 12.332,1235
(1 row)
```

FROM_BASE64

FROM_BASE64(TEXT str)

描述：对通过TO_BASE64()返回的字符串进行解码，并将结果作为二进制字符串返回。返回值的长度受GUC参数max_allowed_packet的影响，最大取值为Min(max_allowed_packet, 1073741819)。

返回值类型：LONGBLOB

示例：

```
m_db=# SELECT FROM_BASE64('R2F1c3M=');
 from_base64
-------------
 Gauss
(1 row)
```

HEX

- HEX(str)

  描述：将字符串str转换为十六进制字符串，str中的每个字符的每个字节都转换为两个十六进制数字，其逆运算为**UNHEX**。

  返回值类型：TEXT

  示例：

  ```
  m_db=# SELECT HEX('abc'), UNHEX(HEX('abc'));
    hex   | unhex
  --------+-------
   616263 | abc
  (1 row)
  ```

- HEX(N)

  描述：将数字N转换为十六进制字符串。

  返回值类型：TEXT

  示例：

  ```
  m_db=# SELECT HEX(255);
   hex
  ```



```
-----
 FF
(1 row)
```

INSERT

INSERT(str,pos,len,newstr)

描述：使用newstr将str内从pos位置开始的len个字符长度的内容进行替换后输出。

参数	类型	描述
str	TEXT	原始字符串
pos	INT	替换起始位置
len	INT	替换长度
newstr	TEXT	替换的子字符串

返回值类型：VARCHAR/VARBINARY/LONGTEXT/LONBLOB

示例：

```
m_db=# SELECT INSERT('abcdefg',2,3,'mmm');
 insert
---------
 ammmefg
(1 row)

--入参中任一参数为NULL时，函数返回NULL。
m_db=# SELECT INSERT(NULL, 2, 4, 'yyy');
 insert
--------
(1 row)

--位置信息异常时，返回原字符串。
m_db=# SELECT INSERT('abcdefg', -100, 4, 'yyy');
 insert
---------
 abcdefg
(1 row)
```

INSTR

INSTR(str,substr)

描述：返回substr在str中第一次出现的位置。

参数	类型	描述
str	TEXT	原始字符串
substr	TEXT	待查找的子字符串

返回值类型：INT

示例：


```
m_db=# SELECT INSTR('absbbsod','bs');
 instr('absbbsod','bs')
------------------------
                      2
(1 row)

--入参含NULL时，函数返回NULL。
m_db=# SELECT INSTR('abcd12abc',null);
 instr
-------
(1 row)
```

LCASE

LCASE(str)

描述：将字符串中的大写字符转换为小写字母。LCASE与**LOWER**功能、用法一致。

返回值类型：TEXT/BLOB

示例：

```
m_db=# SELECT LCASE('Hello,world!');
    lcase
--------------
 hello,world!
(1 row)
```

LEFT

LEFT(str, len)

描述：返回字符串str最左侧的len个字符，如果str或len为NULL，返回NULL。

返回值类型：TEXT/BLOB

示例：

```
m_db=# SELECT LEFT('abcdef', 3);
 left
------
 abc
(1 row)

m_db=# SELECT LEFT('abcdef', NULL);
 left
------

(1 row)

m_db=# SELECT LEFT('abcdef', 11);
  left
--------
 abcdef
(1 row)
```

LENGTH

LENGTH(str)

描述：返回字符串str包含的字节数，多字节的字符会被计为多个字节。例如，输入2个2字节的字符，LENGTH()返回4，而CHAR_LENGTH()返回2。

返回值类型：BIGINT

示例：

```
m_db=# SELECT LENGTH('text');
 char_length
-------------
           4
(1 row)
```

LOCATE

LOCATE(TEXT substr, TEXT str[, INT pos])

描述：返回字符串substr在字符串str中第一次出现的位置，从位置pos（默认为1）开始计算。如果在str中无法匹配到substr，则返回0。

参数：

substr：必选，表示要搜索的子字符串。

str：必选，表示将被搜索的字符串。

pos：可选，表示搜索的起始位置。

返回值类型：BIGINT

示例：

```
m_db=# SELECT LOCATE('bar', 'foobarbar');
 locate
--------
      4
(1 row)

m_db=# SELECT LOCATE('bar', 'foobarbar',5);
 locate
--------
      7
(1 row)
```

LOWER

LOWER(str)

描述：将字符串中的大写字母转换为小写字母。LOWER的功能、用法和**LCASE**一致。

返回值类型：TEXT/BLOB

示例：

```
m_db=# SELECT LOWER('Hello,world!');
    lower
--------------
 hello,world!
(1 row)
```

LPAD

LPAD(str, len, padstr)

描述：使用字符串padstr从左侧填充字符串str，直到长度为len。返回值的长度受GUC参数max_allowed_packet的影响，最大取值为Min(max_allowed_packet/函数返回值字符集单字符最大长度, 1073741819)。

参数：参数具体说明见下表。

参数	类型	描述
str	TEXT	待被填充的字符串
len	INT	填充后字符串的长度
padstr	TEXT	用于填充的字符串

返回值类型：TEXT/BLOB

- 如果str长度大于len，则返回str左侧的len个字符。
- 如果str、len、padstr中有一个为NULL，则返回NULL。
- 如果len为负数，则返回NULL。

示例：

```
m_db=# SELECT LPAD('hello', 10, 'abc');
    lpad
------------
 abcabhello
(1 row)

m_db=# SELECT LPAD('hello', 3, 'abc');
 lpad
------
 hel
(1 row)

m_db=# SELECT LPAD('hello', 10, NULL);
 lpad
------

(1 row)

m_db=# SELECT LPAD('hello', -10, 'abc');
 lpad
------

(1 row)
```

LTRIM

LTRIM(str)

描述：删除字符串str左侧的空格。

返回值类型：TEXT/BLOB

示例：

```
m_db=# SELECT LTRIM('  hello   ');
  ltrim
----------
 hello
(1 row)
```

MAKE_SET

MAKE_SET(INT bits, TEXT str1, TEXT str2,...)

描述：返回一个分隔子字符串的字符串，字符串由参数集合中相应位置的字符串组成，str1对应bit0，str2对应bit1。函数最多能处理64项str入参，当超过64项时只截取前64项。bits获取最右侧N位bit值。

返回值类型：TEXT

示例：

```
m_db=# SELECT MAKE_SET(5,'hello','nice','world');
  make_set
-------------
 hello,world
(1 row)
```

MD5

MD5(TEXT str)

描述：用于计算一个给定字符串的MD5摘要，并将结果作为一个32位的（由十六进制字符组成的）字符串返回。如果参数为NULL，则返回NULL。

返回值类型：TEXT

示例：

```
m_db=# SELECT MD5('Gauss');
                md5
----------------------------------
 edc41fb7bf8bdac012523d1bcd949a4f
(1 row)
```

说明

- MD5加密算法安全性低，存在安全风险，不建议使用。
- MD5函数会在日志中记录哈希的明文，因此不建议用户用该函数加密密钥等敏感信息。

MID

MID(TEXT str, INT pos)

描述：输出str从pos位置开始到str结尾的字符串，MID函数与SUBSTR函数完全一致，内部沿用SUBSTR函数。在升级观察期期间，调用MID函数创建视图，回滚之后由于SUBSTR函数没有被删除，导致该视图不会被删除。

返回值类型：TEXT/BLOB

示例：

```
m_db=# SELECT MID('foobarbarfoobar', 4);
     mid
---------------
 barbarfoobar
(1 row)
```

MID(str FROM pos)

描述：输出str从pos位置开始到结尾的字符串。

返回值类型：TEXT/BLOB

示例：

```
m_db=# SELECT MID('foobarbarfoobar' FROM 4);
     mid
```



```
--------------
 barbarfoobar
(1 row)
```

MID(TEXT str, INT pos, INT len)

描述：输出str从pos位置开始之后len个字符长度所组成的字符串。

返回值类型：TEXT/BLOB

示例：

```
m_db=# SELECT MID('foobarbarfoobar', 4, 9);
    mid
-----------
 barbarfoo
(1 row)
```

MID(str FROM pos FOR len)

描述：输出str从pos位置开始之后len个字符长度所组成的字符串。

返回值类型：TEXT/BLOB

示例：

```
m_db=# SELECT MID('foobarbarfoobar' FROM 4 FOR 9);
    mid
-----------
 barbarfoo
(1 row)
```

ORD

ORD(TEXT str)

描述：返回字符串参数中的第一个字符的字符代码。

- 如果第一个字符为单字节字符，函数返回字符的ASCII值。
- 如果第一个字符是多字节字符，返回逻辑为：第一个字节代码 + 第二个字节的代码 * 256 + 第三个字节的代码 * 256 * 256，即result += char[n] << (8*n)，其中n为字节下标。

返回值类型：BIGINT

示例：

```
m_db=# SELECT ORD('gauss');
 ord
-----
 103
(1 row)
```

OCTET_LENGTH

OCTET_LENGTH(str)

描述：以字节为单位返回字符串str的长度，即字节的数量。如果str不是字符串类型，以其标准输出格式的字符串的长度为准。

返回值类型：BIGINT

示例：


```
m_db=# SELECT OCTET_LENGTH('text');
 octet_length
--------------
            4
(1 row)
```

POSITION

POSITION(substr in str)

描述：返回字符串substr在字符串str中第一次出现的位置，与LOCATE(substr,str)功能相同。

返回值类型：BIGINT

说明

- 此函数在参数m_format_dev_version值为's2'或以上版本且参数m_format_behavior_compat_options值包含'enable_conflict_funcs'的情况下为M-Compatibility兼容模式数据库实现行为，即本章节描述的行为；其他行为和《开发指南》中的“SQL参考 > 函数和操作符 > 数字操作函数和操作符”章节中的ceil函数保持一致。
- 在非M-Compatibility兼容模式数据库实现行为下，入参类型为M-Compatibility兼容模式所独有的数据类型时（例如BIGINT UNSIGNED），使用该函数结果可能存在异常。

示例：

```
m_db=# SELECT POSITION('bs' in 'absbbsod');
 position
----------
        2
(1 row)
```

QUOTE

QUOTE(TEXT str)

描述：在字符串str两端分别添加单引号后进行输出。同时可以将str中的特殊字符进行转义，包括\\、\'或Control+Z（\Z）。如果参数为NULL，则返回值为字符串“NULL”，而不括起单引号。

返回值类型：VARCHAR/LONGTEXT/VARBINARY/LONGBLOB

示例：

```
m_db=# SELECT QUOTE('Do not do that.');
       quote
--------------------
 'Do not do that.'
(1 row)
-- 字符串中含有也是字符\\、\'或Control+Z（\Z）。
m_db=# SELECT QUOTE('Don\\t do that.');
       quote
--------------------
 'Don\\t do that.'
(1 row)

m_db=# select quote('Don\Zt do that.');
       quote
--------------------
 'Don\Zt do that.'
(1 row)

m_db=# select quote('Don\'t do that.');
       quote
```



```
-------------------
 'Don\'t do that.'
(1 row)

m_db=# SELECT QUOTE(null);
 quote
-------
 NULL
(1 row)
```

RANDOM_BYTES

RANDOM_BYTES(INT len)

描述：返回指定长度len的随机字符串。如果入参为NULL，则返回NULL，如果入参值不在[1, 1024]，则返回“ Length value is out of range ”的错误信息。

返回值类型：BLOB

示例：

```
m_db=# SELECT RANDOM_BYTES(1);
 random_bytes
--------------
 q
(1 row)
```

REPEAT

REPEAT(str, count)

描述：返回字符串str重复count次组成的字符串。如果count小于等于0，返回NULL。如果str或者count为NULL，返回NULL。返回值的长度受GUC参数max_allowed_packet的影响，最大取值为Min(max_allowed_packet, 1073741819)。

返回值类型：TEXT/BLOB

示例：

```
m_db=# SELECT REPEAT('abc', 3);
  repeat
-----------
 abcabcabc
(1 row)

m_db=# SELECT REPEAT('abc', NULL);
 repeat
--------

(1 row)

m_db=# SELECT REPEAT('abc', -1);
 repeat
--------

(1 row)
```

REPLACE

REPLACE(str, from_str, to_str)

描述：使用字符串to_str替换字符串str中的字符串from_str，替换时区分大小写。当有任意一个参数为NULL时，返回NULL。返回值的长度受GUC参数max_allowed_packet的影响，最大取值为Min(max_allowed_packet, 1073741819)。

参数：参数具体说明见下表。

参数	类型	描述
str	TEXT	待被替换的字符串
from_str	TEXT	查找目标
to_str	TEXT	替换成

返回值类型：TEXT/BLOB

示例：

```
m_db=# SELECT REPLACE('abc,efg,hijk,abcde,dabc,e', 'abc', 'xyz');
          replace
----------------------------
 xyz,efg,hijk,xyzde,dxyz,e
(1 row)

m_db=# SELECT REPLACE('abc,efg,hijk,abcde,dabc,e', NULL, 'xyz');
 replace
---------

(1 row)
```

REVERSE

REVERSE(str)

描述：返回字符串str的倒序排列。

返回值类型：TEXT/BLOB

示例：

```
m_db=# SELECT REVERSE('abcd');
 reverse
---------
 dcba
(1 row)
```

RIGHT

RIGHT(str, len)

描述：返回字符串str最右边len个字符，如果str或len为NULL，则返回NULL。

返回值类型：TEXT/BLOB

示例：

```
m_db=# SELECT RIGHT('abcdef', 4);
 right
-------
 cdef
(1 row)

m_db=# SELECT RIGHT('abcdef', NULL);
 right
-------
```



```
(1 row)
```

RPAD

RPAD(str, len, padstr)

描述：使用字符串padstr从右侧填充字符串str，直到长度为len。返回值的长度受GUC参数max_allowed_packet的影响，最大取值为Min(max_allowed_packet/函数返回值字符集单字符最大长度, 1073741819)。

参数：参数具体说明见下表。

参数	类型	描述
str	TEXT	待被填充的字符串
len	INT	填充后字符串的长度
padstr	TEXT	用于填充的字符串

返回值类型：TEXT/BLOB

- 如果str长度大于len，则返回str左侧的len个字符。
- 如果str、len、padstr中有一个为NULL，则返回NULL。
- 如果len为负数，则返回NULL。

示例：

```
m_db=#  SELECT RPAD('hello', 10, 'abc');
    rpad
------------
 helloabcab
(1 row)

m_db=# SELECT RPAD('hello', 3, 'abc');
 rpad
------
 hel
(1 row)

m_db=# SELECT RPAD('hello', 10, NULL);
 rpad
------

(1 row)

m_db=# SELECT RPAD('hello', -10, 'abc');
 rpad
------

(1 row)
```

RTRIM

RTRIM(str)

描述：删除字符串str右侧的空格。

返回值类型：TEXT/BLOB

示例：

```
m_db=# SELECT RTRIM('  hello   ');
  rtrim
---------
   hello
(1 row)
```

SHA

SHA(str)

描述：计算字符串str的SHA-1 160位检验和。

参数：str，字符串或数字。

返回值：字符串类型。40个十六进制数字组成的字符串。

示例：

```
m_db=# SELECT SHA('abc');
                 sha
------------------------------------------
 a9993e364706816aba3e25717850c26c9cd0d89d
(1 row)
```

说明

- SHA1加密算法安全性低，存在安全风险，不建议使用。
- SHA函数会在日志中记录哈希的明文，因此不建议用户用该函数加密密钥等敏感信息。

SHA1

SHA1(str)

描述：SHA1是SHA函数的别名，功能用法相同。

示例：

```
m_db=# SELECT SHA1('abc');
                 sha1
------------------------------------------
 a9993e364706816aba3e25717850c26c9cd0d89d
(1 row)
```

说明

- SHA1加密算法安全性低，存在安全风险，不建议使用。
- SHA1函数会在日志中记录哈希的明文，因此不建议用户用该函数加密密钥等敏感信息。

SHA2

SHA2(str, hash_length)

描述：计算字符串str的SHA-2检验和。

参数：

str：字符串或数字。

hash_length：对应相应的SHA2算法，可选值为 0(SHA-256)、224(SHA-224)、256(SHA-256)、384(SHA-384)、512(SHA-512)，其他值将返回NULL。

返回值：字符串类型。40个十六进制数字组成的字符串。

示例：

```
m_db=# SELECT SHA2('abc', 224);
                            sha2
------------------------------------------------------------
 23097d223405d8228642a477bda255b32aadbce4bda0b3f7e36c9da7
(1 row)
```

说明

- SHA224加密算法安全性低，存在安全风险，不建议使用。
- SHA2函数会在日志中记录哈希的明文，因此不建议用户用该函数加密密钥等敏感信息。

SPACE

SPACE(N)

描述：返回由N个空格组成的字符串,如果N小于等于0，返回空字符串。返回值的长度受GUC参数max_allowed_packet的影响，最大取值为Min(max_allowed_packet, 1073741819)。

返回值类型：TEXT

示例：

```
m_db=# SELECT CONCAT('a', SPACE(6), 'b');
  concat
-----------
 a      b
(1 row)

m_db=# SELECT CONCAT('a', SPACE(-1), 'b');
 concat
--------
 ab
(1 row)
```

STRCMP

STRCMP(str1, str2)

描述：比较str1与str2是否相同。

返回值类型：INT

- 如果 str1 等于 str2， STRCMP函数将返回 0。
- 如果 str1 小于 str2， STRCMP函数将返回 -1。
- 如果 str1 大于 str2， STRCMP函数将返回 1。
- 当任意个参数为 NULL 时， STRCMP函数将返回 NULL。

示例：

```
m_db=# SELECT STRCMP('abc', 'abc');
 strcmp
--------
      0
(1 row)

m_db=# SELECT STRCMP('abc1', 'abc');
 strcmp
```



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

m_db=# SELECT STRCMP('abc', 'abc1');
 strcmp
--------
     -1
(1 row)

m_db=# SELECT STRCMP('abc1', 'abc2');
 strcmp
--------
     -1
(1 row)

m_db=# SELECT STRCMP('abc1', NULL);
 strcmp
--------

(1 row)
```

SUBSTR

SUBSTR函数有以下四种原型，各原型中str、pos以及len的含义一致。

- SUBSTR(str, pos)
- SUBSTR(str FROM pos)
- SUBSTR(str, pos, len)
- SUBSTR(str FROM pos FOR len)
- SUBSTR(str FOR len FROM pos)
- SUBSTR(str FOR len)

描述：返回str的子字符串，起始位置为pos，长度为len。不指定len时，返回的子字符串从pos位置开始，一直到str字符串的结尾。

- pos的位置从1开始计数。
- 参数中包含NULL时，返回NULL。
- pos为负数时，从str尾部向头部倒序确定起始位置。
- len小于等于0、pos的位置为0、pos的位置越界时，返回空字符串。

参数：参数具体说明见下表。

参数	类型	描述
str	TEXT	待被截取的字符串
pos	INT	起始位置
len	INT	子字符串的长度

返回值类型：TEXT/BLOB

示例：

```
m_db=# SELECT SUBSTR('abcdefg', 2, 3);
 substr
```



```
--------
 bcd
(1 row)

m_db=# SELECT SUBSTR('abcdefg', NULL);
 substr
--------

(1 row)

m_db=# SELECT SUBSTR('abcdefg', 2);
 substr
--------
 bcdefg
(1 row)

m_db=# SELECT SUBSTR('abcdefg', -2, 3);
 substr
--------
 fg
(1 row)

m_db=# SELECT SUBSTR('abcdefg', -2);
 substr
--------
 fg
(1 row)

m_db=# SELECT SUBSTR('abcdefg', 0, 3);
 substr
--------

(1 row)
```

SUBSTRING

SUBSTRING函数有以下四种原型，各原型中str、pos以及len的含义一致。

- SUBSTRING(str, pos)
- SUBSTRING(str FROM pos)
- SUBSTRING(str, pos, len)
- SUBSTRING(str FROM pos FOR len)
- SUBSTRING(str FOR len FROM pos)
- SUBSTRING(str FOR len)

描述：SUBSTRING与**SUBSTR**功能、用法一致。

参数：参数具体说明见下表。

参数	类型	描述
str	TEXT	待被截取的字符串
pos	INT	起始位置
len	INT	子字符串的长度

返回值类型：TEXT/BLOB

示例：


```
m_db=# SELECT SUBSTRING('abcdefg', 2, 3);
 substring
-----------
 bcd
(1 row)

m_db=# SELECT SUBSTRING('abcdefg', NULL);
 substring
-----------

(1 row)

m_db=# SELECT SUBSTRING('abcdefg', 2);
 substring
-----------
 bcdefg
(1 row)

m_db=# SELECT SUBSTRING('abcdefg', -2, 3);
 substring
-----------
 fg
(1 row)

m_db=# SELECT SUBSTRING('abcdefg', -2);
 substring
-----------
 fg
(1 row)

m_db=# SELECT SUBSTRING('abcdefg', 0, 3);
 substring
-----------

(1 row)
```

SUBSTRING_INDEX

SUBSTRING_INDEX(str, delim, count)

描述：使用分隔符delim和次数count确定字符串str中最终分隔符的位置，返回分最终隔符左侧（或右侧）的全部字符。匹配分隔符delim时区分大小写。

- 如果count为正数，从字符串的左侧开始计数，直到delim出现了count次，返回最终分隔符左侧的全部字符。
- 如果count为负数，从字符串的右侧开始计数，直到delim出现了|count|次，返回最终分隔符右侧的全部字符。
- 如果任意一个参数为NULL，返回NULL。
- 如果str或delim为空字符串，返回空字符串
- 如果count等于0，返回空字符串。

参数：参数具体说明见下表。

参数	类型	描述
str	TEXT	待被截取的字符串
delim	TEXT	分隔符
count	INT	分隔符出现的次数

返回值类型：TEXT/BLOB

示例：

```
m_db=# SELECT SUBSTRING_INDEX('abc..def..ghi..jkl....', '..', 3);
 substring_index
-----------------
 abc..def..ghi
(1 row)

m_db=# SELECT SUBSTRING_INDEX('abc..def..ghi..jkl....', '..', -3);
 substring_index
-----------------
 jkl....
(1 row)

m_db=# SELECT SUBSTRING_INDEX('abc..def..ghi..jkl....', '..', NULL);
 substring_index
-----------------

(1 row)

m_db=# SELECT SUBSTRING_INDEX('abc..def..ghi..jkl....', '..' ,0);
 substring_index
-----------------

(1 row)
```

TO_BASE64

TO_BASE64(TEXT str)

描述：将字符串转换为base-64编码形式，并将结果作为字符串返回。如果参数不为字符串，则在转换前，将其转换为字符串。返回值的长度受GUC参数max_allowed_packet的影响，最大取值为Min(max_allowed_packet, 1073741819)。

返回值类型：LONGTEXT

示例：

```
m_db=# SELECT TO_BASE64('Gauss');
 to_base64
-----------
 R2F1c3M=
(1 row)
```

TRIM

TRIM([[BOTH | LEADING | TRAILING] [remstr] FROM] str)

描述：删除字符串str中的所有前缀/后缀子字符串remstr，当任意一个参数为NULL时，返回NULL。匹配remstr时区分大小写。

- remstr：默认为空格。
- [BOTH | LEADING | TRAILING]：
  - BOTH：删除前缀和后缀；
  - LEADING：删除前缀；
  - TRAILING：删除后缀；
  - 默认为BOTH。

返回值类型：TEXT/BLOB

示例：

```
m_db=# SELECT TRIM('   abc   ');
 trim
------
 abc
(1 row)

m_db=# SELECT TRIM(TRAILING FROM '   abc   ');
  trim
--------
    abc
(1 row)

m_db=# SELECT TRIM(BOTH 'xyz' FROM 'xyzxyabcxyzxy');
    trim
------------
 xyabcxyzxy
(1 row)
```

UCASE

UCASE(str)

描述：将字符串中的小写字母转换为大写字母。UCASE的功能、用法和**UPPER**一致。

返回值类型：TEXT/BLOB

示例：

```
m_db=# SELECT UCASE('Hello,world!');
    ucase
--------------
 HELLO,WORLD!
(1 row)
```

UNCOMPRESS

UNCOMPRESS (LONGBLOB string_to_uncompress)

描述：对COMPRESS函数压缩的字符串进行解压。如果函数入参不是压缩值，结果返回NULL。

返回值类型：LONGBLOB

示例：

```
m_db=# SELECT UNCOMPRESS(COMPRESS('abcde'));
 UNCOMPRESS(COMPRESS('abcde'))
-------------------------------
 abcde
(1 row)
```

UNCOMPRESSED_LENGTH

UNCOMPRESSED_LENGTH(LONGBLOB compressed_string)

描述：返回压缩字符串在压缩之前的长度。

返回值类型：INT

示例：

```
m_db=# SELECT UNCOMPRESSED_LENGTH(COMPRESS('abcde'));
 UNCOMPRESSED_LENGTH(COMPRESS('abcde'))
```



```
----------------------------------------
                                      5
(1 row)
```

UNHEX

UNHEX(str)

描述：将字符串str中的每个字符解释为十六进制数字，将其转换为该数字表示的字节，返回二进制字符串，其逆运算为**HEX(str)**。str中的字符必须是合法的十六进制数字，如果参数包含任何非法的数字，则返回NULL。

返回值类型：LONGBLOB

示例：

```
m_db=# SELECT UNHEX(616263), HEX(UNHEX('616263'));
 unhex |  hex
-------+--------
 abc   | 616263
(1 row)
```

UPPER

UPPER(str)

描述：将字符串中的小写字母转换为大写字母。UPPER的功能、用法和**UCASE**一致。

返回值类型：TEXT/BLOB

示例：

```
m_db=# SELECT UPPER('Hello,world!');
    upper
--------------
 HELLO,WORLD!
(1 row)
```

4.5.8 日期时间函数

ADDDATE

- ADDDATE(date, INTERVAL expr unit)

  描述：用于在指定的日期上添加一段时间间隔，并返回计算结果。该函数这种形式的功能用法与**DATE_ADD**函数相同。

  参数：参数具体说明见下表。

参数	类型	描述	取值范围
date	时间类型表达式、TEXT、DATETIME、DATE、TIME等类型。	表示日期时间基准。	和类型范围一致。
expr	整数、浮点数、字符串、表达式等。	表示时间间隔，可以是负值。	和类型范围一致。

参数	类型	描述	取值范围
unit	关键字。	表示时间间隔的单位。	YEAR、QUARTER、MONTH、WEEK、DAY、HOUR、MINUTE、SECOND、MICROSECOND等，详见**时间间隔表达式**。

示例：

```
m_db=# SELECT ADDDATE('2018-05-01', INTERVAL 1 DAY);
  adddate
------------
 2018-05-02
(1 row)
```

- ADDDATE(date, days)

  描述：用于将days指定的天数与date指定的日期相加，返回计算结果。

  参数：参数具体说明见下表。

参数	类型	描述	取值范围
date	时间类型表达式、TEXT、DATETIME、DATE、TIME等类型。	表示日期时间基准。	和类型范围一致。
days	整数、浮点数、字符串等。	表示时间间隔，可以是负值。	和类型范围一致。

  返回值类型：TEXT、DATE、DATETIME或TIME。

  示例：

```
m_db=# SELECT ADDDATE('2018-05-01', 1);
  adddate
------------
 2018-05-02
(1 row)
```

ADDTIME

ADDTIME(expr1, expr2)

描述：返回时间或时间日期表达式expr1与时间表达式expr2相加后的值，返回值的格式与expr1保持一致。

参数：

- expr1：时间或时间日期表达式。
- expr2：时间表达式。

返回值类型：TEXT、DATETIME或TIME。返回值类型与expr1类型有关，expr1解析为DATETIME，则返回DATETIME，expr1解析为TIME，则返回TIME。

示例：

```
m_db=# SELECT ADDTIME('2000-03-01 20:59:59', '00:00:01');
       addtime
---------------------
 2000-03-01 21:00:00
(1 row)

m_db=# SELECT ADDTIME('20:59:59', '00:00:01');
 addtime
----------
 21:00:00
(1 row)
```

CONVERT_TZ

CONVERT_TZ(dt, from_tz, to_tz)

描述：CONVERT_TZ函数用于将日期时间值dt从from_tz时区转换到to_tz时区。如果参数dt无效，则返回空。如果dt超出TIMESTAMP类型支持范围，则不会发生转换。

参数：参数具体说明见下表。

参数	类型	描述	取值范围
dt	TEXT、DATETIME、DATE、TIME和其他可以隐式转换为DATETIME的类型。	表示日期时间值。	请参见表4-55。
from_tz/to_tz	±hh:mm格式的字符串。	表示相较于UTC时间的偏移，如'+08:00'。	[-12:59, +13:00]，正负号必须加。
	命名时区。	如'MET'、'UTC'等。	具体请参考PG_TIMEZONE_NAMES系统视图。

返回值类型：DATETIME

示例：

```
m_db=# SELECT CONVERT_TZ('2023-01-01 10:10:10', '+00:00', '+08:00');
    convert_tz
---------------------
 2023-01-01 18:10:10
(1 row)
```

CURDATE

CURDATE()

描述：返回本地函数调用开始时刻的系统日期。支持在同一连接内修改时区，返回的日期受时区的影响。

返回值类型：DATE

示例：

```
m_db=# SELECT CURDATE();
  curdate
------------
 2023-12-09
(1 row)
```

CURRENT_DATE

CURRENT_DATE/CURRENT_DATE()

描述：返回本地函数调用开始时刻的系统日期。支持在同一连接内修改时区，返回的日期受时区的影响。该函数是**CURDATE**的别名。

返回值类型：DATE

示例：

```
m_db=# SELECT CURRENT_DATE;
 current_date
--------------
 2023-12-09
(1 row)

m_db=# SELECT CURRENT_DATE();
 current_date
--------------
 2023-12-09
(1 row)
```

CURRENT_TIME

CURRENT_TIME/CURRENT_TIME([scale])

描述：返回本地函数调用开始时刻的时间，不含日期的部分。

参数：scale，表示微秒部分的精度，有效值为0到6的整数，默认值为0。

返回值类型：TIME

示例：

```
m_db=# SELECT CURRENT_TIME;
 current_time
--------------
 10:14:12
(1 row)

m_db=# SELECT CURRENT_TIME(6);
  current_time
-----------------
 10:14:15.512064
(1 row)
```

CURRENT_TIMESTAMP

CURRENT_TIMESTAMP/CURRENT_TIMESTAMP([scale])

描述：返回本地函数调用开始时刻的日期时间。该函数是**NOW**的别名。

参数：scale，表示微秒部分的精度，有效值为0到6的整数，默认值为0。

返回值类型：DATETIME

示例：

```
m_db=# SELECT CURRENT_TIMESTAMP;
  current_timestamp
---------------------
 2023-12-09 11:20:04
(1 row)

m_db=# SELECT CURRENT_TIMESTAMP(6);
     current_timestamp
----------------------------
 2023-12-09 11:20:07.059582
(1 row)
```

CURTIME

CURTIME([scale])

描述：返回本地函数调用开始时刻的时间，不含日期的部分。该函数是**CURRENT_TIME**的别名。

参数：scale，表示微秒部分的精度，有效值为0到6的整数，默认值为0。

返回值类型：TIME

示例：

```
m_db=# SELECT CURTIME();
 curtime
----------
 11:24:03
(1 row)

m_db=# SELECT CURTIME(6);
     curtime
-----------------
 11:24:05.590022
(1 row)
```

DATE()

DATE(expr)

描述：返回时间或日期表达式expr的日期部分。

参数：expr，可取的类型有TEXT、DATETIME、DATE、TIME、时间或日期表达式等。

返回值类型：DATE

示例：

```
m_db=# SELECT DATE('2023-01-01 10:11:12');
    date
------------
 2023-01-01
(1 row)
```

DATEDIFF

DATEDIFF(date1, date2)

描述：返回date1和date2之间的天数。计算中只用到参数的日期部分，忽略时间部分。

参数：

- date1：时间类型表达式、TEXT、DATETIME、DATE、TIME等类型。
- date2：时间类型表达式、TEXT、DATETIME、DATE、TIME等类型。

返回值类型：INT

示例：

```
m_db=# SELECT DATEDIFF('2018-05-01', '2018-04-01');
 datediff
----------
       30
(1 row)
```

DATE_ADD

DATE_ADD(date, INTERVAL expr unit)

描述：用于在指定的日期上添加一段时间间隔，并返回计算结果。

参数：参数具体说明见下表。

参数	类型	描述	取值范围
date	时间类型表达式、TEXT、DATETIME、DATE、TIME等类型。	表示日期时间基准。	和类型范围一致。
expr	整数、浮点数、字符串、表达式等。	表示时间间隔，可以是负值。	和类型范围一致。
unit	关键字。	表示时间间隔的单位。	YEAR、QUARTER、MONTH、WEEK、DAY、HOUR、MINUTE、SECOND、MICROSECOND等，详见**时间间隔表达式**。

返回值类型：TEXT、DATE、DATETIME或TIME。

示例：

```
m_db=# SELECT DATE_ADD('2018-05-01', INTERVAL 1 DAY);
  adddate
------------
 2018-05-02
(1 row)
```

DATE_FORMAT

DATE_FORMAT(date, format)

描述：将日期时间以指定格式输出。

参数：参数具体说明见下表。

参数	类型	描述	取值范围
date	时间类型表达式、TEXT、DATETIME、DATE、TIME等类型。	需要格式化的日期。	和类型范围一致。
format	TEXT。	格式化字符串。	详见表4-48

表 4-48 格式化字符串的具体取值和含义

格式符	含义
%a	星期的缩写（Sun..Sat）。
%b	月份的缩写（Jan..Dec）。
%c	月份数字（0..12）。
%D	带有英语前缀的月份中的每天（0th, 1st, 2nd, 3rd, …）。
%d	月份中的每天的两位数字表示（00..31）。
%e	月份中的每天的数字表示（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	一年中的每天（001..366）。
%k	小时（0..23）。
%l	小时（1..12）。
%M	月份名称（January..December）。
%m	两位数字月份（00..12）。
%p	AM或者PM。
%r	十二小时制时间（hh:mm:ss后跟AM或PM）。
%S	秒（00..59）。
%s	秒（00..59）。
%T	二十四小时制时间（hh:mm:ss）。
%U	一年中的星期（00..53），每周的开始是星期天。

%u	一年中的星期（00..53），每周的开始是星期一。
%V	一年中的星期（01..53），每周的开始是星期天。
%v	一年中的星期（01..53），每周的开始是星期一。
%W	星期的名称（Sunday..Saturday）。
%w	星期中的每天（0=星期天..6=星期六）。
%X	一年中的星期，每周的开始是星期天，四位数字，用于%V。
%x	一年中的星期，每周的开始是星期一，四位数字，用于%v。
%Y	四位数字年份。
%y	两位数字年份。
%%	文字字符%。
%*x*	*x*，对于以上未列出的任何"*x*"。

返回值类型：TEXT

示例：

```
m_db=# SELECT DATE_FORMAT('2023-10-11 12:13:14.151617','%b %c %M %m');
    date_format
--------------------
 Oct 10 October 10
(1 row)
```

DATE_SUB

DATE_SUB(date, INTERVAL expr unit)

描述：用于在指定的日期上减去一段时间间隔，并返回计算结果。

参数：参数具体说明见下表。

参数	类型	描述	取值范围
date	时间类型表达式、TEXT、DATETIME、DATE、TIME等类型。	表示日期时间基准。	和类型范围一致。
expr	整数、浮点数、字符串、表达式等。	表示时间间隔，可以是负值。	和类型范围一致。

参数	类型	描述	取值范围
unit	关键字。	表示时间间隔的单位。	YEAR、QUARTER、MONTH、WEEK、DAY、HOUR、MINUTE、SECOND、MICROSECOND等，详见**时间间隔表达式**。

返回值类型：TEXT、DATE、DATETIME或TIME。

示例：

```
m_db=# SELECT DATE_SUB('2018-05-01', INTERVAL 1 DAY);
  date_sub
------------
 2018-04-30
(1 row)
```

DAY

DAY(date)

描述：提取日期时间的天数部分，将结果返回。该函数是**DAYOFMONTH**的别名。

参数：date，表示指定提取的日期时间，可以是时间类型表达式、TEXT、DATETIME、DATE、TIME等类型。

返回值类型：INT

示例：

```
m_db=# SELECT DAY('2018-05-12');
 day
-----
  12
(1 row)
```

DAYNAME

DAYNAME(date)

描述：返回日期的星期几名称。返回值使用的语言由GUC参数lc_time_names控制。

参数：date，表示日期时间，可以是时间类型表达式、TEXT、DATETIME、DATE、TIME等类型。

返回值类型：TEXT

示例：

```
m_db=# SELECT DAYNAME('2018-05-12');
 dayname
----------
 Saturday
(1 row)
```

DAYOFMONTH

DAYOFMONTH(date)

描述：提取日期时间的天数部分，将结果返回。该函数是**DAY**的别名。

参数：date，表示指定提取的日期时间，可以是时间类型表达式、TEXT、DATETIME、DATE、TIME等类型。

返回值类型：INT

示例：

```
m_db=# SELECT DAYOFMONTH('2018-05-12');
 dayofmonth
------------
         12
(1 row)
```

DAYOFWEEK

DAYOFWEEK(date)

描述：返回日期的工作日索引（1=周日，2=周一，…，7=周六）。

参数：date，表示日期时间，可以是时间类型表达式、TEXT、DATETIME、DATE、TIME等类型。

返回值类型：BIGINT

示例：

```
m_db=# SELECT DAYOFWEEK('2023-04-16');
 dayofweek
-----------
         1
(1 row)
```

DAYOFYEAR

DAYOFYEAR(date)

描述：返回一年中的第几天。

参数：date，表示日期时间，可以是时间类型表达式、TEXT、DATETIME、DATE、TIME等类型。

返回值类型：INT，范围[1, 366]。

示例：

```
m_db=# SELECT DAYOFYEAR('2000-12-31');
 dayofyear
-----------
       366
(1 row)
```

EXTRACT

EXTRACT(unit FROM date)

描述：以整数类型返回date的unit指定部分值。如果unit指定多个部分，则将所有的值按顺序拼接。

参数：

- unit：表示时间间隔单位，详见**时间间隔表达式**。
- date：表示日期时间，可以是时间类型表达式、TEXT、DATETIME、DATE、TIME等类型。

返回值类型：BIGINT

示例：

```
m_db=# SELECT EXTRACT(DAY FROM '2000-12-31');
 extract
---------
      31
(1 row)

m_db=# SELECT EXTRACT(WEEK FROM '2000-12-31');
 extract
---------
      53
(1 row)
m_db=# SELECT EXTRACT(YEAR_MONTH FROM '2000-12-31');
 extract
---------
  200012
(1 row)
```

FROM_DAYS

FROM_DAYS(days)

描述：返回指定天数days对应的日期值。天数指距离“0000-01-01”的天数。

参数：days，指定的天数。

返回值类型：DATE

示例：

```
m_db=# SELECT FROM_DAYS(50000);
 from_days
------------
 0136-11-23
(1 row)
```

FROM_UNIXTIME

FROM_UNIXTIME(unix_timestamp[, format])

描述：将Unix时间戳转换为日期时间格式的函数。Unix时间戳是指从1970年1月1日08:00:00 UTC到指定时间的秒数。

参数：

- unix_timestamp：表示UNIX时间戳。
- format：返回指定格式的日期时间字符串。
  - 具体返回指定格式参考**DATE_FORMAT**函数。

返回值类型：TEXT/DATETIME

示例：

```
m_db=# SELECT FROM_UNIXTIME(1111885200);
    from_unixtime
---------------------
```



```
 2005-03-27 09:00:00
(1 row)

m_db=# SELECT FROM_UNIXTIME(1447430881, '%Y %D %M %h:%i:%s %x');
          from_unixtime
-----------------------------------
 2015 14th November 12:08:01 2015
(1 row)
```

GET_FORMAT

GET_FORMAT({DATE | TIME | DATETIME | TIMESTAMP}, {'EUR' | 'USA' | 'JIS' | 'ISO' | 'INTERNAL'})

描述：返回指定格式的字符串，即不同地区的“年月日 时分秒”格式和排序标准。

参数：

- DATE | TIME | DATETIME | TIMESTAMP：表示时间类型，为关键字。
- 'EUR' | 'USA' | 'JIS' | 'ISO' | 'INTERNAL'：五种时间格式，TEXT类型。

返回值类型：TEXT

示例：

```
m_db=# SELECT GET_FORMAT(DATE, 'EUR');
 get_format
------------
 %d.%m.%Y
(1 row)

m_db=# SELECT GET_FORMAT(DATE, 'USA');
 get_format
------------
 %m.%d.%Y
(1 row)
```

HOUR

HOUR(expr)

描述：返回时间或日期时间表达式expr中的的小时部分的数值。

参数：time，表示时间，可以是时间类型表达式、TEXT、DATETIME、DATE、TIME等类型。

返回值类型：INT

示例：

```
m_db=# SELECT HOUR('10:11:12');
 hour
------
   10
(1 row)
```

LAST_DAY

LAST_DAY(date)

描述：返回日期时间date当月最后一天的日期值。

参数：date，表示日期时间，可以是时间类型表达式、TEXT、DATETIME、DATE、TIME等类型。

返回值类型：DATE

示例：

```
m_db=# SELECT LAST_DAY('2023-01-01');
  last_day
------------
 2023-01-31
(1 row)

m_db=# SELECT LAST_DAY('2023-01-01 10:11:12');
  last_day
------------
 2023-01-31
(1 row)
```

LOCALTIME

LOCALTIME/LOCALTIME([scale])

描述：返回本地函数调用开始时刻的日期时间。LOCALTIME和LOCALTIME()是**NOW**的同义词。

参数：scale，表示微秒部分的精度，有效值为0到6的整数，默认值为0。

返回值类型：DATETIME

示例：

```
m_db=# SELECT LOCALTIME;
      localtime
---------------------
 2023-12-09 17:58:18
(1 row)

m_db=# SELECT LOCALTIME(6);
         localtime
----------------------------
 2023-12-09 17:58:21.283227
(1 row)
```

LOCALTIMESTAMP

LOCALTIMESTAMP/LOCALTIMESTAMP([scale])

描述：返回当前的日期时间。LOCALTIMESTAMP和LOCALTIMESTAMP()是**NOW**的同义词。

参数：scale，表示微秒部分的精度，有效值为0到6的整数，默认值为0。

返回值类型：DATETIME

示例：

```
m_db=# SELECT LOCALTIMESTAMP;
   localtimestamp
---------------------
 2023-12-09 18:00:34
(1 row)

m_db=# SELECT LOCALTIMESTAMP(6);
       localtimestamp
----------------------------
 2023-12-09 18:01:03.053427
(1 row)
```


MAKEDATE

MAKEDATE(year, dayofyear)

描述：根据年份和天数值返回日期。

参数：

- year：BIGINT，指定的年份。
- dayofyear：BIGINT，该年的第若干天，允许跨年，小于等于0时返回null。

返回值类型：DATE

示例：

```
m_db=# SELECT MAKEDATE(2000, 60);
  makedate
------------
 2000-02-29
(1 row)
```

MAKETIME

MAKETIME(hour, minute, second)

描述：通过入参hour、minute、second生成指定的时间值。

参数：

- hour：小时部分对应的数值。
- minute：分钟部分对应的数值。
- second：秒数部分对应的数值，可以有小数部分。

返回值类型：TIME

示例：

```
m_db=# SELECT MAKETIME(8, 15, 26.53);
  maketime
-------------
 08:15:26.53
(1 row)

m_db=# SELECT MAKETIME(-838, 15, 26.53);
   maketime
---------------
 -838:15:26.53
(1 row)
```

MICROSECOND

MICROSECOND(expr)

描述：返回时间或日期时间表达式expr的微秒数。

参数：expr，可取的类型有TEXT、DATETIME、DATE、TIME、时间或日期表达式等。

返回值类型：BIGINT

示例：


```
m_db=# SELECT MICROSECOND('2023-5-5 10:10:10.24485');
 microsecond
-------------
      244850
(1 row)
```

MINUTE

MINUTE(expr)

描述：返回时间或日期时间表达式expr中的分钟数。

参数：expr，可取的类型有TEXT、DATETIME、DATE、TIME、时间或日期表达式等。

返回值类型：INT

示例：

```
m_db=# SELECT MINUTE('10:11:12');
 minute
--------
     11
(1 row)
```

MONTH

MONTH(date)

描述：提取日期时间的月份部分，将结果返回。

参数：date，表示指定提取的日期时间，可以是时间类型表达式、TEXT、DATETIME、DATE、TIME等类型。

返回值类型：INT

示例：

```
m_db=# SELECT MONTH('2018-05-12');
 month
-------
     5
(1 row)
```

MONTHNAME

MONTHNAME(date)

描述：返回日期的月份全名称。

参数：date，表示日期时间，可以是时间类型表达式、TEXT、DATETIME、DATE、TIME等类型。

返回值类型：TEXT

示例：

```
m_db=# SELECT MONTHNAME('2018-05-12');
 monthname
-----------
 May
(1 row)
```


NOW

NOW([scale])

描述：返回本地函数调用开始时刻的日期时间，受时区设置的影响。

参数：scale，表示微秒部分的精度，有效值为0到6的整数，默认值为0。

返回值类型：DATETIME

示例：

```
m_db=# SELECT NOW();
         now
---------------------
 2023-12-11 09:46:56
(1 row)

m_db=# SELECT NOW(6);
            now
----------------------------
 2023-12-11 09:46:58.404811
(1 row)
```

PERIOD_ADD

PERIOD_ADD(period, month_number)

描述：在指定格式的日期上添加指定的月数，并按照YYYYMM格式进行返回。

参数：

- period：使用YYYYMM或YYMM格式表示的日期。
- month_number：需要添加的月数，允许为负数。

返回值类型：BIGINT。

示例：

```
m_db=# SELECT PERIOD_ADD(202205, -12);
 period_add
------------
     202105
(1 row)
```

PERIOD_DIFF

PERIOD_DIFF(P1, P2)

描述：计算两个日期的月份差。

参数：P1，P2均为YYMM或YYYYMM格式的日期。

返回值类型：BIGINT，月份差。

示例：

```
m_db=# SELECT PERIOD_DIFF('202101', '202102');
 period_diff
-------------
          -1
(1 row)
```


QUARTER

QUARTER(date)

描述：返回指定日期的季度值，取值范围为[1, 4]。

参数：date，表示指定的日期时间，可以是时间类型表达式、TEXT、DATETIME、DATE、TIME等类型。

返回值类型：INT

示例：

```
m_db=# SELECT QUARTER('2018-05-12');
 quarter
---------
       2
(1 row)
```

SECOND

SECOND(expr)

描述：返回时间或日期时间表达式expr中的秒数。

参数：expr，表示时间，可以是时间类型表达式、TEXT、DATETIME、DATE、TIME等类型。

返回值类型：INT

示例：

```
m_db=# SELECT SECOND('10:11:12');
 second
--------
     12
(1 row)
```

SEC_TO_TIME

SEC_TO_TIME(seconds)

描述：将seconds表示的秒数转换为TIME类型的时间。

参数：seconds，指定的秒数，取值范围[-3020399, +3020399]。

返回值类型：TIME

示例：

```
m_db=# SELECT SEC_TO_TIME(3020399);
   sec_to_time
------------------
 838:59:59.000000
(1 row)
```

STR_TO_DATE

STR_TO_DATE(str, format)

描述：将指定的字符串根据指定日期格式转为日期或时间。

参数：

- str：需要格式化成日期的字符串。
- format：格式化字符串，详见**表4-48**。

返回值类型：DATETIME、DATE、TIME。

示例：

```
m_db=# SELECT STR_TO_DATE('May 1, 2013','%M %d,%Y');
 str_to_date
-------------
 2013-05-01
(1 row)
```

SUBDATE

- SUBDATE(date, INTERVAL expr unit)

  描述：用于在指定的日期上减去一段时间间隔，并返回计算结果。该函数这种形式的功能用法与**DATE_SUB**函数相同。

  参数：参数具体说明见下表。

参数	类型	描述	取值范围
date	时间类型表达式、TEXT、DATETIME、DATE、TIME等类型。	表示日期时间基准。	和类型范围一致。
expr	整数、浮点数、字符串、表达式等。	表示时间间隔，可以是负值。	和类型范围一致。
unit	关键字。	表示时间间隔的单位。	YEAR、QUARTER、MONTH、WEEK、DAY、HOUR、MINUTE、SECOND、MICROSECOND等，详见**时间间隔表达式**。

  返回值类型：TEXT、DATE、DATETIME或TIME。

  示例：

```
m_db=# SELECT SUBDATE('2018-05-01', INTERVAL 1 DAY);
  subdate
------------
 2018-04-30
(1 row)
```

- SUBDATE(date, days)

  描述：用于将days指定的天数与date指定的日期相减，返回计算结果。

  参数：参数具体说明见下表。

参数	类型	描述	取值范围
date	时间类型表达式、TEXT、DATETIME、DATE、TIME等类型。	表示日期时间基准。	和类型范围一致。
days	整数、浮点数、字符串等。	表示时间间隔，可以是负值。	和类型范围一致。

返回值类型：TEXT、DATE、DATETIME或TIME。

示例：

```
m_db=# SELECT SUBDATE('2018-05-01', 1);
  subdate
------------
 2018-04-30
(1 row)
```

SUBTIME

SUBTIME(expr1, expr2)

描述：返回时间或时间日期表达式expr1与时间表达式expr2相减后的值，返回值的格式与expr1保持一致。

参数：

- expr1：时间或时间日期表达式。
- expr2：时间表达式。

返回值类型：TEXT、DATETIME或TIME。返回值类型与expr1类型有关，expr1解析为DATETIME，则返回DATETIME，expr1解析为TIME，则返回TIME。

示例：

```
m_db=# SELECT SUBTIME('2000-03-01 20:59:59', '00:00:01');
       subtime
---------------------
 2000-03-01 20:59:58
(1 row)

m_db=# SELECT SUBTIME('20:59:59', '00:00:01');
 subtime
----------
 20:59:58
(1 row)
```

SYSDATE

SYSDATE([precision])

描述：返回函数执行时刻的系统日期和时间。

参数：precision，表示微秒部分的精度，有效值为0到6的整数，默认值为0。

返回值类型：DATETIME

示例：

```
m_db=# SELECT SYSDATE();
       sysdate
---------------------
 2023-12-11 11:28:29
(1 row)

m_db=# SELECT SYSDATE(6);
          sysdate
----------------------------
 2023-12-11 11:28:31.535374
(1 row)
```

TIME

TIME(expr)

描述：返回时间或日期表达式expr的时间部分。

参数：expr，可取的类型有TEXT、DATETIME、DATE、TIME、时间或日期表达式等。

返回值类型：TIME

示例：

```
m_db=# SELECT TIME('2023-01-01 10:11:12');
   time
----------
 10:11:12
(1 row)
```

TIMEDIFF

TIMEDIFF(date1, date2)

描述：以TIME类型返回两个日期的时间间隔。

参数：date1或date2，时间类型表达式、TEXT、DATETIME、DATE或TIME等类型。

返回值类型：TIME

示例：

```
m_db=# SELECT TIMEDIFF(date'2022-12-30',20221229);
 timediff
----------
 24:00:00
(1 row)
```

TIMESTAMP

- TIMESTAMP(expr)

  描述：返回日期时间表达式expr的日期时间值。

  参数：时间类型表达式、TEXT、DATETIME、DATE或TIME等类型。

  返回值类型：DATETIME

  示例：

  ```
  m_db=# SELECT TIMESTAMP('2022-12-30');
       timestamp
  ---------------------
  ```



```
 2022-12-30 00:00:00
(1 row)
```

- TIMESTAMP(expr1, expr2)

  描述：返回日期时间表达式expr1与时间表达式expr2相加后的日期时间值。如果expr2中包含日期，返回NULL空值。

  参数：时间类型表达式、TEXT、DATETIME、DATE或TIME等类型。

  返回值类型：DATETIME

  示例：

```
m_db=# SELECT TIMESTAMP('2022-12-30', '12:00:00');
     timestamp
---------------------
 2022-12-30 12:00:00
(1 row)
```

TIMESTAMPADD

TIMESTAMPADD(unit, interval, datetime_expr)

描述：返回一个新的日期时间，该日期时间是通过将unit的多个interval添加到datetime_expr计算得到的。

参数：参数的具体说明如下表。

参数	类型	描述	取值范围
unit	关键字。	时间间隔单位。	YEAR、QUARTER、MONTH、WEEK、DAY、HOUR、MINUTE、SECOND、MICROSECOND。
interval	数值表达式。	时间间隔数值。	和类型范围一致。
datetime_expr	时间类型表达式、TEXT、DATETIME、DATE或TIME等类型。	时间日期基准值。	和类型范围一致。

返回值类型：TEXT、DATE、DATETIME或TIME。

示例：

```
m_db=#  SELECT TIMESTAMPADD(DAY,-2,'2022-07-27');
 timestampadd
--------------
 2022-07-25
(1 row)
```

TIMESTAMPDIFF

TIMESTAMPDIFF(unit, date1, date2)

描述：以unit为单位返回两个日期时间的间隔。

参数：参数的具体说明如下表。

参数	类型	描述	取值范围
unit	关键字。	时间间隔单位。	YEAR、QUARTER、MONTH、WEEK、DAY、HOUR、MINUTE、SECOND、MICROSECOND。
date1	时间类型表达式、TEXT、DATETIME、DATE或TIME等类型。	表示日期时间。	和类型范围一致。
date2	时间类型表达式、TEXT、DATETIME、DATE或TIME等类型。	表示日期时间。	和类型范围一致。

返回值类型：BIGINT

示例：

```
m_db=# SELECT TIMESTAMPDIFF(SECOND, '2023-01-01 10:11:12', '2023-01-01 05:00:00');
 timestampdiff
---------------
        -18672
(1 row)
```

TIME_FORMAT

TIME_FORMAT(time, format)

描述：根据格式说明符format格式化time入参。

参数：

- time：时间类型表达式、TEXT、DATETIME、DATE、TIME等类型。
- format：格式化字符串，具体取值和含义如下表所示。

格式符	含义
%f	微秒（000000至999999）。
%H	小时（00到23）。
%h、%I	小时（00到12）。
%l	小时（0到12）。
%k	小时（0到838）。

格式符	含义
%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、%s	秒（00到59）。
%T	24小时格式的时间（hh：mm：ss）。
%a、%b、%D、%j、%M、%U、%u、%V、%v、%W、%w、%X、%x	NULL。
%c、%e	0。
%d、%m、%y	00。
%Y	0000。
%其他字符/其他字符，如%A/A	返回字符本身，如A。
%单个字符+字符串s	解析（%单个字符），拼接s。

返回值类型：TEXT

示例：

```
m_db=# SELECT TIME_FORMAT('25:30:30', '%T|%r|%H|%h|%I|%i|%S|%f|%p|%k');
                time_format
-----------------------------------------------------
 25:30:30|01:30:30 AM|25|01|01|30|30|000000|AM|25
(1 row)
```

TIME_TO_SEC

TIME_TO_SEC(time)

描述：将TIME类型入参转化为秒数。

参数：time，时间类型表达式、TEXT、DATETIME、DATE或TIME等类型。时间表达式按照TIME来解析。

返回值类型：INT

示例：

```
m_db=# SELECT TIME_TO_SEC('10:11:12');
 time_to_sec
-------------
       36672
(1 row)
```

TO_DAYS

TO_DAYS(date)

描述：返回指定日期距离“0000-01-01”的天数。

参数：入参为时间类型表达式、TEXT、DATETIME、DATE或TIME等类型。时间表达式按照DATE来解析。

返回值类型：INT

示例：

```
m_db=# SELECT TO_DAYS('2000-1-1');
 to_days
---------
  730485
(1 row)
```

TO_SECONDS

TO_SECONDS(expr)

描述：返回时间或日期时间表达式expr自公元零年以来的秒数。

参数：入参为时间或日期时间表达式、TEXT、DATETIME、DATE或TIME等类型。时间表达式按照DATETIME来解析。

返回值类型：BIGINT

示例：

```
m_db=# SELECT TO_SECONDS('2009-11-29 13:43:32');
 to_seconds
-------------
 63426721412
(1 row)
```

UNIX_TIMESTAMP

UNIX_TIMESTAMP([date])

描述：

- 不指定date参数时，返回当前时间距离“1970-01-01 00:00:00”的秒数，受时区参数设置的影响。
- 指定参数date参数时，返回指定时间距离“1970-01-01 00:00:00”的秒数，受时区参数设置的影响。

参数：时间类型表达式、TEXT、DATETIME、DATE或TIME等类型。

返回值类型：DECIMAL/BIGINT。当入参存在小数部分或者入参为非法的时间类型字符串时，返回DECIMAL，否则返回BIGINT。

示例：

```
m_db=# SELECT UNIX_TIMESTAMP('2022-12-22');
 unix_timestamp
----------------
     1671638400
(1 row)
```

UTC_DATE

UTC_DATE/UTC_DATE()

描述：将函数执行时的当前UTC日期作为“YYYY-MM-DD”格式的值返回。

返回值类型：DATE

示例：

```
m_db=# SELECT UTC_DATE;
  utc_date
------------
 2023-12-11
(1 row)

m_db=# SELECT UTC_DATE();
  utc_date
------------
 2023-12-11
(1 row)
```

UTC_TIME

UTC_TIME/UTC_TIME([scale])

描述：将函数执行时的当前UTC时间作为“HH:MM:SS”格式的值返回。

参数：scale，表示微秒部分的精度，有效值为0到6的整数，默认值为0。

返回值类型：TIME

示例：

```
m_db=# SELECT UTC_TIME;
 utc_time
----------
 07:10:49
(1 row)

m_db=# SELECT UTC_TIME();
 utc_time
----------
 07:10:49
(1 row)

m_db=# SELECT UTC_TIME(6);
    utc_time
-----------------
 07:10:51.445213
(1 row)
```

UTC_TIMESTAMP

UTC_TIMESTAMP/UTC_TIMESTAMP([scale])

描述：将函数执行时的当前UTC时间戳作为“YYYY-MM-DD HH:MM:SS”格式的值返回。

参数：scale，表示微秒部分的精度，有效值为0到6的整数，默认值为0。

返回值类型：DATETIME

示例：

```
m_db=# SELECT UTC_TIMESTAMP;
    utc_timestamp
---------------------
 2023-12-11 07:12:02
(1 row)

m_db=# SELECT UTC_TIMESTAMP();
```



```
   utc_timestamp
---------------------
 2023-12-11 07:12:04
(1 row)

m_db=# SELECT UTC_TIMESTAMP(3);
      utc_timestamp
--------------------------
 2023-12-11 07:12:06.413
(1 row)
```

WEEK

WEEK(date[, mode])

描述：返回日期的周数。周数根据ISO 8601:1988计算，在ISO定义里，一月份的前几天可能是前一年的第52或者第53周，十二月份的后几天可能是下一年的第1周。

参数：

- date：指定日期时间，时间类型表达式、TEXT、DATETIME、DATE、TIME等类型。
- 当省略mode参数时，使用系统变量default_week_format的值，默认值为0。当指定mode参数时，可以指定一周的开始，以及返回值的范围。可选参数mode的含义如下表所示，默认值为0。

  表 4-49 mode 参数的取值及含义说明

mode	一周的第一天	返回值范围	一月份的哪一个周是第一周
0	周日	0~53	第一个有周日的周。
1	周一	0~53	第一个有4天及以上的周。
2	周日	1~53	第一个有周日的周。
3	周一	1~53	第一个有4天及以上的周。
4	周日	0~53	第一个有4天及以上的周。
5	周一	0~53	第一个有周一的周。
6	周日	1~53	第一个有4天及以上的周。
7	周一	1~53	第一个有周一的周。

返回值类型：BIGINT

示例：

```
m_db=# SELECT WEEK('2000-01-01');
 week
------
    0
```



```
(1 row)

m_db=# SELECT WEEK('2000-01-01', 2);
 week
------
   52
(1 row)
```

WEEKDAY

WEEKDAY(date)

描述：返回一个日期的工作日索引值，即星期一为0，星期二为1，星期三为2，星期四为3，星期五为4，星期六为5，星期日为6。

参数：时间类型表达式、TEXT、DATETIME、DATE或TIME等类型。

返回值类型：INT

示例：

```
m_db=# SELECT WEEKDAY('1970-01-01 12:00:00');
 weekday
---------
 3
(1 row)
```

WEEKOFYEAR

WEEKOFYEAR(date)

描述：返回日期时间的日历周，范围[1, 53]。等同于**WEEK(date, 3)**。

参数：时间类型表达式、TEXT、DATETIME、DATE或TIME等类型。

返回值类型：INT

示例：

```
m_db=# SELECT WEEKOFYEAR('1970-05-22');
 weekofyear
------------
         21
(1 row)
```

YEAR

YEAR(date)

描述：提取日期时间的年份部分，将结果返回。

参数：date，表示指定提取的日期时间，可以是时间类型表达式、TEXT、DATETIME、DATE、TIME等类型。

返回值类型：INT。

示例：

```
m_db=# SELECT YEAR('2018-05-12');
 year
------
 2018
(1 row)
```


YEARWEEK

YEARWEEK(date[, mode])

描述：返回日期的年份和周数。

参数：

- date：指定日期时间，时间类型表达式、TEXT、DATETIME、DATE、TIME等类型。
- 可选参数mode详见**表4-49**。

返回值类型：BIGINT

示例：

```
m_db=#  SELECT YEARWEEK('2000-01-01', 3);
 yearweek
----------
   199952
(1 row)
```

4.5.9 类型转换函数

CAST

CAST(expr AS type)

描述：将表达式expr显式转换为数据类型type，等价于**CONVERT(expr, type)**。type支持如下数据类型：

- BINARY[(N)]

  当expr为NULL时，返回值类型为BINARY(0)；expr不为NULL时，返回值类型为VARBINARY。如果指定了可选长度N，超出N的部分会被截断，不足N的部分使用0x00进行填充。如果未指定可选长度N，将根据表达式的结果确定返回值的长度。如果返回值的长度超出BINARY的上限，则返回值类型为BLOB，仍不足则返回值类型为LONGBLOB。N的取值受GUC参数max_allowed_packet的影响，最大取值为Min(max_allowed_packet, 1073741819)。
- CHAR[(N)]

  当expr为NULL时，返回值类型为CHAR(0)；expr不为NULL时，返回值类型为VARCHAR。如果指定了可选长度N，超出N的部分会被截断，不足N的部分不会进行填充。如果未指定可选长度N，将根据表达式的结果确定返回值的长度。如果返回值的长度超出VARCHAR的上限，则返回值类型为TEXT，仍不足则返回值类型为LONGTEXT。
- DATE

  返回值类型为DATE。
- DATETIME[(M)]

  返回值类型为DATETIME，M用于确定秒的精度，范围为[0,6]。
- TIME[(N)]

  返回值类型为TIME，M用于确定秒的精度，范围为[0,6]。
- DECIMAL[M [, D])]

  返回值类型为DECIMAL，M用于确定最大位数（精度），最大值为65，默认为10。D用于确定小数点后的位数，最大值为30，默认为0。

- DOUBLE
  返回值类型为DOUBLE。
- FLOAT[(p)]
  当没有指定精度p时，返回值类型为FLOAT。当p >=0且p < 24时，返回值类型为FLOAT。当p >= 24时，返回值类型为DOUBLE。 当p < 0时，返回错误。
- JSON
  返回值类型为JSON。
- SIGNED[INTEGER]
  返回值类型为带符号的BIGINT。
- UNSIGNED[INTEGER]
  返回值类型为不带符号的BIGINT。

示例：

```
m_db=# SELECT CAST('abc' AS BINARY(2));
WARNING:  Truncated incorrect binary(2) value: 'abc'
CONTEXT:  referenced column: cast
 cast
------
 ab
(1 row)

m_db=# SELECT CAST('abc' AS CHAR(10));
 cast
------
 abc
(1 row)

m_db=# SELECT CAST('2023/1/1' AS DATE);
    cast
------------
 2023-01-01
(1 row)

m_db=# SELECT CAST('2023/01/01' AS DATETIME(2));
          cast
-------------------------
 2023-01-01 00:00:00.00
(1 row)

m_db=#  SELECT CAST('09:23:14' AS TIME(2));
    cast
-------------
 09:23:14.00
(1 row)

m_db=# SELECT CAST(12.34567 AS DECIMAL(10,4));
  cast
---------
 12.3457
(1 row)

m_db=# SELECT CAST(-12 AS SIGNED);
 cast
------
  -12
(1 row)

m_db=# SELECT CAST(-12 AS UNSIGNED);
         cast
----------------------
 18446744073709551604
(1 row)
```


CONVERT

- CONVERT(expr, type)

  描述：将表达式expr显式转换为数据类型type，等价于**CAST(expr, type)**。

  示例：

```
m_db=# SELECT CONVERT('abc', BINARY(2));
WARNING:  Truncated incorrect binary(2) value: 'abc'
CONTEXT:  referenced column: convert
 convert
---------
 ab
(1 row)

m_db=# SELECT CONVERT('abc', CHAR(10));
 convert
---------
 abc
(1 row)

m_db=# SELECT CONVERT('2023/1/1', DATE);
  convert
------------
 2023-01-01
(1 row)

m_db=# SELECT CONVERT('2023/01/01', DATETIME(2));
        convert
------------------------
 2023-01-01 00:00:00.00
(1 row)

m_db=# SELECT CONVERT('09:23:14', TIME(2));
   convert
-------------
 09:23:14.00
(1 row)

m_db=# SELECT CONVERT(12.34567, DECIMAL(10,4));
 convert
---------
 12.3457
(1 row)

m_db=# SELECT CONVERT(-12, SIGNED);
 convert
---------
     -12
(1 row)

m_db=# SELECT CONVERT(-12, UNSIGNED);
       convert
----------------------
 18446744073709551604
(1 row)
```

- CONVERT(expr USING transcoding_name)

  描述：将表达式expr转换为transcoding_name指定的字符集。

  示例：

```
m_db=# SELECT CONVERT('abc' USING 'latin1');
 convert
---------
 abc
(1 row)
```


4.5.10 网络地址函数

INET_ATON

INET_ATON(TEXT str)

描述：指定一个字符串形式的IPv4网络地址，函数返回以网络字节顺序表示的地址数值。如果入参不是有效地址，则返回NULL。

返回值类型：BIGINT UNSIGNED

示例：

```
m_db=# SELECT INET_ATON('127'),INET_ATON('127.1'),INET_ATON('127.0.1'),INET_ATON('127.0.0.1');
 INET_ATON('127') | INET_ATON('127.1') | INET_ATON('127.0.1') | INET_ATON('127.0.0.1')
------------------+--------------------+----------------------+------------------------
              127 |         2130706433 |           2130706433 |             2130706433
(1 row)
```

INET_NTOA

INET_NTOA (INT expr)

描述：指定一个按网络字节顺序排列的数字IPv4网络地址，函数返回网络地址的字符串形式。如果入参不是有效地址，则返回NULL。

返回值类型：VARCHAR

示例：

```
m_db=# SELECT INET_NTOA(4294902271),INET_NTOA(4294967295),INET_NTOA(511);
 INET_NTOA(4294902271) | INET_NTOA(4294967295) | INET_NTOA(511)
-----------------------+-----------------------+----------------
 255.255.1.255         | 255.255.255.255       | 0.0.1.255
(1 row)
```

INET6_ATON

INET6_ATON(TEXT str)

描述：指定字符串形式的IPv4网络地址，函数返回网络地址的二进制字符串形式。如果入参不是有效地址，则返回NULL。

返回值类型：VARBINARY

示例：

```
m_db=# SELECT HEX(INET6_ATON('1.2.3.4'));
 HEX(INET6_ATON('1.2.3.4'))
----------------------------
 01020304
(1 row)
```

INET6_NTOA

INET6_NTOA(VARBINARY expr)

描述：指定二进制字符串形式的IPv4网络地址，函数返回以网络字节顺序表示的地址数值。当入参为有效地址，且为VARBINARY或BINARY类型时，函数返回有效结果，否则返回NULL。

返回值类型：VARCHAR

示例：

```
m_db=# SELECT INET6_NTOA(INET6_ATON('1:2:3:4:5:6:7:8'));
 INET6_NTOA(INET6_ATON('1:2:3:4:5:6:7:8'))
-------------------------------------------
 1:2:3:4:5:6:7:8
(1 row)
```

IS_IPV6

IS_IPV6(ANY expr)

描述：判断输入的字符串是否为合法的IPv6地址，合法返回1，不合法返回0，如果输入的字符串为NULL，则返回0。

返回值类型：INT

示例：

```
m_db=# SELECT IS_IPV6('1:2:3:4:5:6:7:8');
 is_ipv6
---------
       1
(1 row)

m_db=# SELECT IS_IPV6('1');
 is_ipv6
---------
       0
(1 row)
```

IS_IPV4

IS_IPV4(ANY expr)

描述：判断输入的字符串是否为合法的IPv4地址，合法返回1，不合法返回0，如果输入的字符串为NULL，则返回0。

返回值类型：INT

示例：

```
m_db=# SELECT IS_IPV4('1.2.3.4');
 is_ipv4
---------
       1
(1 row)

m_db=# SELECT IS_IPV4('1');
 is_ipv4
---------
       0
(1 row)
```

4.5.11 聚合函数

聚合函数对一组值进行计算，并返回单个的值。除了COUNT(*)外，聚合函数都会忽略NULL值。聚合函数通常与GROUP BY子句一起使用。

说明

- 聚合函数的结果与数据输入顺序相关，不同的数据输入顺序会导致结果的精度差异。

```
m_db=# CREATE TABLE IF NOT EXISTS t1 (name VARCHAR(20), c1 INT(100), c2 FLOAT(7,5));
CREATE TABLE
m_db=# INSERT INTO t1 VALUES
m_db-# ('计算机', 666,-55.155),
m_db-# ('计算机', 789,-15.593),
m_db-# ('计算机', 928,-53.963),
m_db-# ('计算机', 666,-54.555),
m_db-# ('计算机', 666,-55.555),
m_db-# ('数据库', 666,-55.155),
m_db-# ('数据库', 789,-15.593),
m_db-# ('数据库', 928,-53.963),
m_db-# ('数据库', 666,-54.555),
m_db-# ('数据库', 666,-55.555);
INSERT 0 10

m_db=# select std(c1/c2) from t1;
        std
--------------------
 15.02396266299967
(1 row)

# ORDER BY改变了聚合函数的执行顺序，导致结果精度差异
m_db=# select std(c1/c2 order by c2) from t1;
        std
--------------------
 15.023962662999669
(1 row)

m_db=# select std(c1/c2) from t1 group by name;
        std
--------------------
 15.023962662999669
 15.023962662999669
(2 rows)

# 与WITH ROLLUP使用，也会改变聚合函数的执行顺序，导致结果精度差异
m_db=# select std(c1/c2) from t1 group by name with rollup;
        std
--------------------
  15.02396266299967
 15.023962662999669
  15.02396266299967
(3 rows)
```

- 聚合函数与GROUP BY同时使用时，若推导类型为DECIMAL场景时，GaussDB保留完整精度的数据。

  例如：

```
m_db=#  SELECT avg(col_znumeric3/col_znumeric2) FROM fun_op_case_tb_1 WHERE id<=10
GROUP BY name;
avg
---------------
1.45980942957
1.45980942957
1.45980942957
(3 rows)
m_db=# SELECT sum(col_bit1/col_time2) FROM  fun_op_case_tb_1 WHERE id<=10 GROUP BY
name;
sum
--------
0.0006
0.0006
0.0006
(3 rows)
```


AVG

AVG([ALL | DISTINCT] expr) [over_clause]

描述：返回输入expr的平均值，NULL值将被忽略。如果指定了over_clause，此函数将作为窗口函数进行执行。over_clause参数说明请参见**窗口函数**说明。

参数类型：可以是TINYINT、SMALLINT、MEDIUMINT、INT、BIGINT、DECIMAL、FLOAT、DOUBLE。

返回值类型：

- 返回值类型视入参数据类型而定。当入参为精确值（INT、DECIMAL）时，返回DECIMAL；当入参为近似值（FLOAT、DOUBLE）时返回DOUBLE类型。
- 如果expr中的值均为NULL，则AVG()返回NULL。
- 如果指定ALL或者不指定ALL与DISTINCT，则返回expr中所有非空值的平均值。
- 如果指定DISTINCT，则返回expr中不同值的平均值。

示例：

```
m_db=# CREATE TABLE m_test(age INT, salary INT);
CREATE TABLE

m_db=# INSERT INTO m_test VALUES(1, NULL), (1, 1), (1, 2), (1, 2), (2, NULL), (2, NULL);
INSERT 0 6

m_db=# SELECT * FROM m_test;
 age | salary
-----+--------
   1 |
   1 |      1
   1 |      2
   1 |      2
   2 |
   2 |
(6 rows)

m_db=# SELECT age, AVG(salary) FROM m_test GROUP BY (age);
 age |  avg
-----+--------
   1 | 1.6667
   2 |
(2 rows)

m_db=# SELECT age, AVG(DISTINCT salary) FROM m_test GROUP BY (age);
 age |  avg
-----+--------
   1 | 1.5000
   2 |
(2 rows)

m_db=# SELECT age, AVG(salary) OVER(PARTITION BY age ORDER by salary RANGE BETWEEN
UNBOUNDED PRECEDING AND CURRENT ROW) FROM m_test;
 age |  avg
-----+--------
   1 | 1.0000
   1 | 1.6667
   1 | 1.6667
   1 | 1.6667
   2 |
   2 |
(6 rows)

m_db=# DROP TABLE m_test;
DROP TABLE
```


说明

当输入是字符类型时，会触发字符类型到数值类型的隐式转换，并输出WARNING日志，导致性能变差。

```
m_db=# SELECT AVG('xxxx');
WARNING:  Truncated incorrect double value: 'xxxx'
 avg
-----
   0
(1 row)
```

BIT_AND

BIT_AND([ALL] expr) [over_clause]

描述：返回expr中所有值的按位与，忽略NULL。如果指定了over_clause，此函数将作为窗口函数进行执行。over_clause参数说明请参见**窗口函数**说明。

参数类型：BIT_AND需要使用BIGINT作为参数，其他类型的参数会被转换为BIGINT，并且可能发生截断。

返回值类型：

- 返回值类型为BIGINT。
- 如果expr中的值均为NULL，则BIT_AND()返回一个中性值（所有位都被设置为1）。
- ALL为可选项，不影响结果。

示例：

```
m_db=# CREATE TABLE m_test(age INT, salary INT);
CREATE TABLE

m_db=# INSERT INTO m_test VALUES(1, NULL), (1, 1), (1, 2), (1, 2), (2, NULL), (2, NULL);
INSERT 0 6

m_db=# SELECT * FROM m_test;
 age | salary
-----+--------
   1 |
   1 |      1
   1 |      2
   1 |      2
   2 |
   2 |
(6 rows)

m_db=# SELECT BIT_AND(ALL salary) FROM m_test GROUP BY age;
       bit_and
----------------------
 0
 18446744073709551615
(2 rows)

m_db=# SELECT age, BIT_AND(salary) OVER(PARTITION BY age ORDER by salary ROWS CURRENT ROW)
FROM m_test;
 age |       bit_and
-----+----------------------
   1 | 1
   1 | 2
   1 | 2
   1 | 18446744073709551615
   2 | 18446744073709551615
   2 | 18446744073709551615
```



```
(6 rows)

m_db=# DROP TABLE m_test;
DROP TABLE
```

说明

当输入是字符类型时，会触发字符类型到数值类型的隐式转换，并输出WARNING日志，导致性能变差。

```
m_db=# SELECT BIT_AND('xxxx');
WARNING:  Truncated incorrect INTEGER value: 'xxxx'
 bit_and
---------
 0
(1 row)
```

BIT_OR

BIT_OR([ALL] expr) [over_clause]

描述：返回expr中所有值的按位或，忽略NULL。如果指定了over_clause，此函数将作为窗口函数进行执行。over_clause参数说明请参见**窗口函数**说明。

参数类型：BIT_OR需要使用BIGINT作为参数，其他类型的参数会被转换为BIGINT，并且可能发生截断。

返回值类型：

- 返回值类型为BIGINT。
- 如果expr中的值均为NULL，则BIT_OR()返回一个中性值（所有位都被设置为0）。
- ALL为可选项，不影响结果。

示例：

```
m_db=# CREATE TABLE m_test(age INT, salary INT);
CREATE TABLE

m_db=# INSERT INTO m_test VALUES(1, NULL), (1, 1), (1, 2), (1, 2), (2, NULL), (2, NULL);
INSERT 0 6

m_db=# SELECT * FROM m_test;
 age | salary
-----+--------
   1 |
   1 |      1
   1 |      2
   1 |      2
   2 |
   2 |
(6 rows)

m_db=# SELECT BIT_OR(ALL salary) FROM m_test GROUP BY age;
 bit_or
--------
 3
 0
(2 rows)

m_db=# SELECT age, BIT_OR(salary) OVER(PARTITION BY age ORDER by salary ROWS CURRENT ROW)
FROM m_test;
 age | bit_or
-----+--------
   1 | 1
```



```
  1 | 2
  1 | 2
  1 | 0
  2 | 0
  2 | 0
(6 rows)

m_db=# DROP TABLE m_test;
DROP TABLE
```

说明

当输入是字符类型时，会触发字符类型到数值类型的隐式转换，并输出WARNING日志，导致性能变差。

```
m_db=# SELECT BIT_OR('xxxx');
WARNING:  Truncated incorrect INTEGER value: 'xxxx'
 bit_or
--------
 0
(1 row)
```

BIT_XOR

BIT_XOR([ALL] expr) [over_clause]

描述：返回expr中所有值的按位异或，忽略NULL。如果指定了over_clause，此函数将作为窗口函数进行执行。over_clause参数说明请参见**窗口函数**说明。

参数类型：BIT_XOR需要使用BIGINT作为参数，其他类型的参数会被转换为BIGINT，并且可能发生截断。

返回值类型：

- 返回值类型为BIGINT。
- 如果expr中的值均为NULL，则BIT_OR()返回一个中性值（所有位都被设置为0）。
- ALL为可选项，不影响结果。

示例：

```
m_db=# CREATE TABLE m_test(age INT, salary INT);
CREATE TABLE

m_db=# INSERT INTO m_test VALUES(1, NULL), (1, 1), (1, 2), (1, 2), (2, NULL), (2, NULL);
INSERT 0 6

m_db=# SELECT * FROM m_test;
 age | salary
-----+--------
   1 |
   1 |      1
   1 |      2
   1 |      2
   2 |
   2 |
(6 rows)

m_db=# SELECT BIT_XOR(ALL salary) FROM m_test GROUP BY age;
 bit_xor
---------
 1
 0
(2 rows)
```



```
m_db=# SELECT age, BIT_XOR(salary) OVER(PARTITION BY age ORDER by salary ROWS CURRENT ROW)
FROM m_test;
 age | bit_xor
-----+---------
   1 | 1
   1 | 2
   1 | 2
   1 | 0
   2 | 0
   2 | 0
(6 rows)

m_db=# DROP TABLE m_test;
DROP TABLE
```

说明

当输入是字符类型时，会触发字符类型到数值类型的隐式转换，并输出WARNING日志，导致性能变差。

```
m_db=# SELECT BIT_XOR('xxxx');
WARNING:  Truncated incorrect INTEGER value: 'xxxx'
 bit_xor
---------
 0
(1 row)
```

COUNT

- COUNT([ALL] expr)

  描述：返回expr中非空值的数量。

  参数类型：入参可以是任意类型。

  返回值类型：

  - 返回值类型为BIGINT。
  - 如果expr中的值均为NULL，则COUNT()返回0。
  - ALL为可选项，不影响结果。
  - COUNT(*)比较特殊，它返回expr中所有值的数量，无论其是否为NULL。

  示例：

```
m_db=# CREATE TABLE m_test(age INT, salary INT);
CREATE TABLE

m_db=# INSERT INTO m_test VALUES(1, NULL), (1, 1), (1, 2), (1, 2), (2, NULL), (2, NULL);
INSERT 0 6

m_db=# SELECT * FROM m_test;
 age | salary
-----+--------
   1 |
   1 |      1
   1 |      2
   1 |      2
   2 |
   2 |
(6 rows)

m_db=# SELECT COUNT(ALL salary) FROM m_test GROUP BY age;
 count
-------
     3
     0
(2 rows)

m_db=# SELECT COUNT(*) FROM m_test GROUP BY age;
```



```
 count
-------
     4
     2
(2 rows)

m_db=# DROP TABLE m_test;
DROP TABLE
```

- COUNT(DISTINCT expr1 [, expr2, ..., exprN])

  描述：返回expr中不同非空值的数量。

  参数类型：入参可以是任意类型。

  返回值类型：

  - 返回值类型为BIGINT。
  - 如果expr中的值均为NULL，则返回0。

  示例：

```
m_db=# CREATE TABLE m_test(age INT, salary INT);
CREATE TABLE

m_db=# INSERT INTO m_test VALUES(1, NULL), (1, 1), (1, 2), (1, 2), (2, NULL), (2, NULL);
INSERT 0 6

m_db=# SELECT * FROM m_test;
 age | salary
-----+--------
   1 |
   1 |      1
   1 |      2
   1 |      2
   2 |
   2 |
(6 rows)

m_db=# SELECT COUNT(DISTINCT salary) FROM m_test GROUP BY age;
 count
-------
     2
     0
(2 rows)

m_db=# DROP TABLE m_test;
DROP TABLE
```

- COUNT(expr) over_clause

  描述：返回根据over_clause分区的expr的非空总行数，若输入为空，输出NULL。指定了over_clause，此函数作为窗口函数进行执行。over_clause参数说明请参见**窗口函数**说明。

  返回值类型：

  - 返回值类型为BIGINT。
  - 如果expr中的值均为NULL，则返回0。

  示例：

```
m_db=# CREATE TABLE count_t1(group_id INT, val INT);
CREATE TABLE
m_db=# INSERT INTO count_t1 VALUES(1,1),(1,2),(1, 3),(2, 4),(2, 5),(2, 6),(3,7),(3,8);
INSERT 0 8
m_db=# SELECT group_id, val, COUNT(val) OVER(partition by group_id) FROM count_t1;
 groupid | val | count
---------+-----+-------
       1 |   1 |     3
       1 |   2 |     3
       1 |   3 |     3
```



```
     2 |  4 |   3
     2 |  5 |   3
     2 |  6 |   3
     3 |  7 |   2
     3 |  8 |   2
(8 rows)

m_db=# DROP TABLE count_t1;
DROP TABLE
```

GROUP_CONCAT

GROUP_CONCAT语法如下：

```
GROUP_CONCAT([DISTINCT] expr [,expr ...]
         [ORDER BY {unsigned_integer | col_name | expr}
             [ASC | DESC] [,col_name ...]]
         [SEPARATOR str_val])
```

描述：将expr中的各个值进行拼接，返回一个字符串结果，NULL值会被忽略。

- 指定DISTINCT时消除expr中的重复值。
- ORDER BY：通过指定列对expr中的值进行排序。
- ASC | DESC：ASC为升序排序，DESC为倒序排序，默认为升序排序。
- SEPARATOR：拼接字符串时使用的分隔符，默认为逗号","。
- 当同时指定DISTINCT和ORDER BY时，DISTINCT表达式必须包含ORDER BY表达式，否则会报错。
- GROUP_CONCAT(... ORDER BY 数字）不代表按照第几个参数的顺序，数字只是一个常量表达式，相当于不排序。
- 使用参数group_concat_max_len限制GROUP_CONCAT最大返回长度，超长截断，目前能返回的最大长度是1073741823。

返回值类型：TEXT或BLOB

示例：

```
m_db=# CREATE TABLE m_test(id INT, name CHAR);
CREATE TABLE

m_db=# INSERT INTO m_test VALUES (1, 'a'), (1, 'b'), (2, 'd'), (2, 'c'), (3, 'a'), (3, NULL);
INSERT 0 6

m_db=# SELECT * FROM m_test;
 id | name
----+------
  1 | a
  1 | b
  2 | d
  2 | c
  3 | a
  3 |
(6 rows)

m_db=# SELECT GROUP_CONCAT(name SEPARATOR '') FROM m_test;
 group_concat
--------------
 abdca
(1 row)

m_db=# SELECT GROUP_CONCAT(DISTINCT id, name ORDER BY id DESC) FROM m_test;
   group_concat
------------------
 3a,2c,2d,1a,1b
```



```
(1 row)

m_db=# DROP TABLE m_test;
DROP TABLE
```

MAX

MAX([ALL | DISTINCT] expr) [over_clause]

描述：返回输入expr的最大值，NULL值将被忽略。如果指定了over_clause，此函数将作为窗口函数进行执行。over_clause参数说明请参见**窗口函数**说明。

参数类型：任意数值、字符串、日期时间类型。

返回值类型：

- 返回值类型与参数数据类型相同，当参数非表字段时，max返回值类型和mysql 8.0保持一致，与mysql 5.7不一样。
- 如果expr中的值均为NULL，则MAX()返回NULL。
- 如果指定ALL或者不指定ALL与DISTINCT，则返回expr中所有非空值的最大值。
- 如果指定了DISTINCT，则返回expr中不同非空值的最大值，与不指定DISTINCT的结果相同。

示例：

```
m_db=# CREATE TABLE m_test(age INT, salary INT);
CREATE TABLE

m_db=# INSERT INTO m_test VALUES(1, NULL), (1, 1), (1, 2), (1, 2), (2, NULL), (2, NULL);
INSERT 0 6

m_db=# SELECT * FROM m_test;
 age | salary
-----+--------
   1 |
   1 |      1
   1 |      2
   1 |      2
   2 |
   2 |
(6 rows)

m_db=# SELECT MAX(DISTINCT salary) FROM m_test GROUP BY age;
 max
-----
   2

(2 rows)

m_db=# SELECT age, MAX(salary) OVER(PARTITION BY age ORDER by salary ROWS CURRENT ROW)
FROM m_test;
 age | max
-----+-----
   1 |   1
   1 |   2
   1 |   2
   1 |
   2 |
   2 |
(6 rows)

m_db=# DROP TABLE m_test;
DROP TABLE
```


MIN

MIN([ALL | DISTINCT] expr) [over_clause]

描述：返回输入expr的最小值，NULL值将被忽略。如果指定了over_clause，此函数将作为窗口函数进行执行。over_clause参数说明请参见**窗口函数**说明。

参数类型：任意数值、字符串、日期时间类型。

返回值类型：

- 返回值类型与参数数据类型相同，当参数非表字段时，min返回值类型和MySQL 8.0保持一致，与MySQL 5.7不一样。
- 如果expr中的值均为NULL，则MIN()返回NULL。
- 如果指定ALL或者不指定ALL与DISTINCT，则返回expr中所有非空值的最小值。
- 如果指定了DISTINCT，则返回expr中不同非空值的最小值，与不指定DISTINCT的结果相同。

示例：

```
m_db=# CREATE TABLE m_test(age INT, salary INT);
CREATE TABLE

m_db=# INSERT INTO m_test VALUES(1, NULL), (1, 1), (1, 2), (1, 2), (2, NULL), (2, NULL);
INSERT 0 6

m_db=# SELECT * FROM m_test;
 age | salary
-----+--------
   1 |
   1 |      1
   1 |      2
   1 |      2
   2 |
   2 |
(6 rows)

m_db=# SELECT MIN(DISTINCT salary) FROM m_test GROUP BY age;
 min
-----
   1

(2 rows)

m_db=# SELECT age, MIN(salary) OVER(PARTITION BY age ORDER by salary ROWS CURRENT ROW) FROM
m_test;
 age | min
-----+-----
   1 |   1
   1 |   2
   1 |   2
   1 |
   2 |
   2 |
(6 rows)

m_db=# DROP TABLE m_test;
DROP TABLE
```

SUM

SUM([ALL | DISTINCT] expr) [over_clause]

描述：返回输入expr的和，NULL值将被忽略。如果指定了over_clause，此函数将作为窗口函数进行执行。over_clause参数说明请参见**窗口函数**说明。

参数类型：可以是TINYINT、SMALLINT、MEDIUMINT、INT、BIGINT、DECIMAL、FLOAT、DOUBLE。

返回值类型：

- 返回值类型视入参数据类型而定。当入参为精确值（INT、DECIMAL）时，返回DECIMAL；当入参为近似值（FLOAT、DOUBLE）时返回DOUBLE类型。
- 如果expr中的值均为NULL，则SUM()返回NULL。
- 如果指定ALL或者不指定ALL与DISTINCT，则返回expr中所有非空值的和。
- 如果指定DISTINCT，则返回expr中不同非空值的和。

示例：

```
m_db=# CREATE TABLE m_test(age INT, salary INT);
CREATE TABLE

m_db=# INSERT INTO m_test VALUES(1, NULL), (1, 1), (1, 2), (1, 2), (2, NULL), (2, NULL);
INSERT 0 6

m_db=# SELECT * FROM m_test;
 age | salary
-----+--------
   1 |
   1 |      1
   1 |      2
   1 |      2
   2 |
   2 |
(6 rows)

m_db=# SELECT age, SUM(salary) FROM m_test GROUP BY (age);
 age | sum
-----+-----
   1 |   5
   2 |
(2 rows)

m_db=# SELECT age, SUM(DISTINCT salary) FROM m_test GROUP BY (age);
 age | sum
-----+-----
   1 |   3
   2 |
(2 rows)

m_db=# SELECT age, SUM(salary) OVER(PARTITION BY age ORDER by salary ROWS BETWEEN
UNBOUNDED PRECEDING AND CURRENT ROW) FROM m_test;
 age | sum
-----+-----
   1 |   1
   1 |   3
   1 |   5
   1 |   5
   2 |
   2 |
(6 rows)

m_db=# DROP TABLE m_test;
DROP TABLE
```


说明

当输入是字符类型时，会触发字符类型到数值类型的隐式转换，并输出WARNING日志，导致性能变差。

```
m_db=# SELECT SUM('xxxx');
WARNING:  Truncated incorrect double value: 'xxxx'
 sum
-----
   0
(1 row)
```

STD

STD(str) [over_clause]

描述：计算所有非NULL输入值的总体标准差并返回结果。如果指定了over_clause，此函数将作为窗口函数进行执行。over_clause参数说明请参见**窗口函数**说明。

返回值类型：DOUBLE

示例：

```
m_db=# CREATE TABLE students (name VARCHAR(20) NOT NULL, math INT);
CREATE TABLE
m_db=# INSERT INTO students (name, math) VALUES('a', 80),('b', 85),('c', 90),('d', 95),('e', 88),('f', 92),('g',
86),('h', 89),('i', 91),('j', 87);
INSERT 0 10
m_db=# SELECT STD(math) AS std_math FROM students;
      std_math
--------------------
 3.9509492530276824
(1 row)

m_db=# SELECT name, STD(math) OVER(ORDER by math ROWS BETWEEN UNBOUNDED PRECEDING
AND CURRENT ROW) FROM students;
 name |        std
------+--------------------
 a    |                  0
 b    |                2.5
 g    | 2.6246692913372693
 j    | 2.6925824035672505
 e    | 2.7856776554368228
 h    | 2.9107081994288277
 c    |  3.063944369932457
 i    | 3.2403703492039275
 f    | 3.4354721852756214
 d    | 3.9509492530276806
(10 rows)


m_db=# DROP TABLE students;
DROP TABLE
```

4.5.12 JSON 函数和操作符

4.5.12.1 JSON 函数

JSON_APPEND

JSON_APPEND(json_doc, path, val[, path, val] ...)

描述：向json_doc中path指定的路径追加一个值，并返回修改后的json。JSON_APPEND是JSON_ARRAY_APPEND的别名，行为与JSON_ARRAY_APPEND一致。

- path必须为一个数组或对象。如果指定的是一个数组，会在数组末尾添加新值；如果指定的是一个对象的key，会在对象value的末尾添加新值；如果对象是单独的一个值，则将其转换为一个数组，并在末尾添加新值。
- 如果json_doc或path路径为null，则函数返回null。
- 如果json_doc中不存在path指定的路径，则不修改。
- 如果json_doc格式错误、path并非是有效的路径表达式或者path中包含*或**时，会产生报错。
- 如果有null值和格式错误场景同时在可变参数列表中，则按异常的先后顺序处理，若null值顺序在前则返回null，否则产生报错。
- 如果json_doc深度超过100时，则产生报错

参数说明：与**表4-50**相同。

返回值：JSON

示例：

```
m_db=# SELECT json_append('[1, [2, 3]]', '$[1]', 4, '$[0]', false, '$[0]', null, '$[0]', 1);
 json_append
-------------------------------------------------------------
 [[1, false, null, 1], [2, 3, 4]]
(1 row)
```

JSON_ARRAY

JSON_ARRAY([val[, val] ...])

描述：从一个可变参数列表构造出一个数组，并返回数组形式的JSON。如果该函数没有任何参数，则返回一个空的数组形式的JSON。

返回值：JSON

示例：

```
-- 不输入入参，返回一个空的数组形式的JSON。
m_db=# SELECT json_array();
 json_array
------------
 []
(1 row)

m_db=# SELECT json_array(TRUE, FALSE, NULL, 114, 'text');
 json_array
-----------------------------------------------------------------
 [true, false, null, 114, "text"]
(1 row)
```

JSON_ARRAY_APPEND

JSON_ARRAY_APPEND(json_doc, path, val[, path, val] ...)

描述：向json_doc中path指定的路径添加一个值，并返回修改后的JSON。

- path必须为一个数组或对象。如果指定的是一个数组，会在数组末尾添加新值；如果指定的是一个对象的key，会在对象value的末尾添加新值；如果对象是单独的一个值，则将其转换为一个数组，并在末尾添加新值。

- 如果json_doc或path路径为null，则函数返回null。
- 如果json_doc中不存在path指定的路径，则不修改。
- 如果json_doc格式错误、path并非是有效的路径表达式或者path中包含*或**时，会产生报错。
- 如果有null值和格式错误场景同时在可变参数列表中，则按异常的先后顺序处理，若null值顺序在前则返回null，否则产生报错。
- 如果json_doc深度超过100时，则产生报错。

参数说明：如表4-50所示。

表 4-50 JSON_ARRAY_APPEND 参数说明

名称	描述
json_doc	表示要添加值的JSON。
path	表示要向JSON中添加值的位置。用“$”表示通过json_doc输入的JSON。
val	表示要添加的值。

返回值：JSON

示例：

```
-- 同时向对象和数组追加值。
m_db=# SELECT json_array_append('[1, [2, 3]]', '$[1]', 4, '$[0]',3);
 json_array_append
----------------------
 [[1, 3], [2, 3, 4]]
(1 row)

-- 向字典对象的value追加值。
m_db=# SELECT json_array_append('{"name":"Tom"}','$.name', 2);
 json_array_append
-------------------------------------------------------------------
 {"name": ["Tom", 2]}
(1 row)

-- 向字典对象本身追加值。
m_db=# SELECT json_array_append('{"name":"Tom"}','$', 2);
 json_array_append
-------------------------------------------------------------------
 [{"name": "Tom"}, 2]
(1 row)
```

JSON_ARRAY_INSERT

JSON_ARRAY_INSERT(json_doc, path, val[, path, val] ...)

描述：向json_doc中path指定的数组中的指定位置插入一个值，并返回修改后的JSON。

- 如果指定的路径不是一个JSON数组，则会产生报错。
- 如果指定的路径中已存在值，则在该路径上插入新值并将已存在的值后移。
- 如果json_doc中不存在path指定的路径，则不修改。

- 如果json_doc或path路径为null，则函数返回null。
- 如果json格式错误、path并非是有效的路径表达式或者path路径中包含*或**时，则会产生报错。
- 如果有null值和格式错误场景同时在可变参数列表中，则按异常的先后顺序处理，若null值顺序在前则返回null，否则产生报错。
- 如果json_doc深度超过100时，则产生报错。

参数说明：与**表4-50**相同。

返回值：JSON

示例：

```
--一对json_path-value用例。
m_db=# SELECT json_array_insert('[1, [2, 3]]', '$[1]', 4);
 json_array_insert
-------------------
 [1, 4, [2, 3]]
(1 row)

--多个json_path-value用例。
m_db=# SELECT json_array_insert('[1, [2, 3]]', '$[1]', 4, '$[0]', false, '$[0]', null, '$[0]', 1);
 json_array_insert
-----------------------------------------------------------------------
 [1, null, false, 1, 4, [2, 3]]
(1 row)

-- 指定路径为JSON数组
m_db=# select json_array_insert('{"a":[1]}','$.a[1]',2);
 json_array_insert
--------------------
 {"a": [1, 2]}
(1 row)

-- 指定的路径不存在
m_db=# select json_array_insert('{"a":1}','$[1]',2);
 json_array_insert
--------------------
 {"a": 1}
(1 row)
```

JSON_CONTAINS

JSON_CONTAINS(target_json_doc, candidate_json_doc[, path])

描述：检查target_json_doc是否包含candidate_json_doc。如果指定path参数，则通过path匹配到target_json_doc中的子JSON数据后，再检查target_json_doc是否包含candidate_json_doc。

- 若target_json_doc格式错误则报错，若target_json_doc为null则返回null。
- 若candidate_json_doc格式错误则报错，若candidate_json_doc为null则返回null。
- 若path格式错误则报错，若path为null则返回null。
- 如果有null值和格式错误场景同时在可变参数列表中，则按异常的先后顺序处理，若null值顺序在前则返回null，否则产生报错。
- 如果json_doc深度超过100时，则产生报错。

返回值：BIGINT

示例：


```
m_db=# SELECT json_contains('[1, [2, 3], 4]', '[1, 3]');
 json_contains
---------------
             1
(1 row)

m_db=# SELECT json_contains('{"a": 1, "b": {"c": 3, "d": 4}}', '{"d": 4}', '$.b');
 json_contains
---------------
             1
(1 row)

m_db=# SELECT json_contains('{"a": 1, "b": {"c": 3, "d": 4}}', '{"e": 4}', '$.b');
 json_contains
---------------
             0
(1 row)
```

JSON_CONTAINS_PATH

JSON_CONTAINS_PATH(json_doc, one_or_all, path [, path [, path …]])

描述：在json_doc中查询path指定的位置是否能匹配到子JSON数据。

- 当one_or_all为one时，path参数中只要存在一个可以匹配到子JSON的数据，即返回1。
- 当one_or_all为all时，path参数需要全部匹配到子JSON才可以返回1，否则返回0。
- 若json_doc格式错误则报错，若为null则返回null。
- 若one_or_all格式错误则报错，若为null则返回null。
- 当one_or_all为one时：
  - 如果前面path的格式均正确，且至少含有一个json_doc中存在的路径，则不会去检查之后的path是否为null或报错，直接返回1。
  - 如果前面path的格式均正确，且均为json_doc中不存在的路径，则后面的path先出现null值则返回null，先出现格式错误则产生报错。
- 在one_or_all为all时：
  - 如果前面path的格式均正确，且至少含有一个json_doc中不存在的路径，则不会去检查之后的path是否为null或报错，直接返回0。
  - 如果前面path的格式均正确，且均为json_doc中存在的路径，则后面的path先出现null值则返回null，先出现格式错误则产生报错。
- 如果json_doc深度超过100时，则产生报错。

返回值：BIGINT

示例：

```
m_db=# SELECT json_contains_path('{"a": {"b": 2, "c": 3}, "d": null}', 'one', '$.d', '$.e');
 json_contains_path
--------------------
                  1
(1 row)


m_db=# SELECT json_contains_path('{"a": {"b": 2, "c": 3}, "d": null}', 'all', '$.d', '$.e');
 json_contains_path
--------------------
                  0
(1 row)
```


JSON_DEPTH

JSON_DEPTH(json_doc)

描述：返回一个JSON的最大深度。如果json_doc为null，则返回null。如果json_doc存在不合法、空字符串或JSON深度超过100时，则产生报错。

返回值：BIGINT

示例：

```
m_db=# SELECT JSON_DEPTH('[]');
 json_depth
------------
          1
(1 row)

m_db=# SELECT JSON_DEPTH('{"s":1}');
 json_depth
------------
          2
(1 row)

m_db=# SELECT JSON_DEPTH('{"s":1, "x":2,"y":3}');
 json_depth
------------
          2
(1 row)

m_db=# SELECT JSON_DEPTH('{"s":1, "x":2,"y":[1]}');
 json_depth
------------
          3
(1 row)
```

JSON_EXTRACT

JSON_EXTRACT(json_doc, path[, path…])

描述：在json_doc中根据path提取子JSON数据，path参数可以有多个，该函数会将提取的子JSON数据拼接成JSON数组返回，但若仅匹配到一个子JSON数据，则会直接返回该子JSON数据。

- 若json_doc格式错误则报错，若为null则返回null。
- 若path格式错误则报错，为null则返回null。
- 如果有null值和格式错误场景同时在可变参数列表中，若null值顺序在前则返回null，否则产生报错。
- 如果json_doc深度超过100时，则产生报错。

返回值：JSON

示例：

```
m_db=# SELECT json_extract('{"a": {"b": 2, "c": 3}, "d": 4}', '$.a');
   json_extract
------------------
 {"b": 2, "c": 3}
(1 row)

m_db=# SELECT json_extract('{"a": {"b": 2, "c": 3}, "d": 4}', '$.a', '$.a');
              json_extract
---------------------------------------
 [{"b": 2, "c": 3}, {"b": 2, "c": 3}]
(1 row)
```


JSON_INSERT

JSON_INSERT(json_doc, path, val[, path, val] ...)

描述：向json_doc中path指定的路径插入一个值，并返回修改后的JSON。JSON_INSERT只能将数据插入到不存在的路径中，如果指定的路径上有值，则不修改。

- 如果json_doc或path路径为null，则返回null。
- 如果json_doc格式错误、path并非是有效的路径表达式或者path中包含*或**时，则产生报错。
- 如果有null值和格式错误场景同时在可变参数列表中，则按异常的先后顺序处理，若null值顺序在前则返回null，否则产生报错。
- 如果json_doc深度超过100时，则产生报错。

参数说明：与**表4-50**相同。

返回值：JSON

示例：

```
-- 如果传入的path路径中，已存在值，则插入不生效。
m_db=# SELECT json_insert('{"x": 1, "y": [1, 2]}', '$.x', '2');
 json_insert
-------------------
 {"x": 1, "y": [1, 2]}
(1 row)

-- 多个路径，按顺序追加。
m_db=# SELECT json_insert('[1, [2, 3], {"a": [4, 5]}]', '$[10]', 10,'$[5]', 5);
 json_insert
-----------------------------------------------------------------------
 [1, [2, 3], {"a": [4, 5]}, 10, 5]
(1 row)
```

JSON_KEYS

JSON_KEYS(json_doc[, path])

描述：以JSON数组的形式返回json_doc中顶级对象的键key，若存在path参数，则以JSON数组的形式返回用path匹配之后的json_doc中顶级对象的键key。

- json_doc或path路径为null时，或者json_doc中不存在path指定的路径时，JSON_KEYS返回null。
- 如果JSON格式错误、path并非是有效的路径表达式或者path中包含*或**时，会产生报错。
- 如果有null值和格式错误场景同时在可变参数列表中，则按异常的先后顺序处理，若null值顺序在前则返回null，否则产生报错。
- 如果json_doc深度超过100时，则产生报错。

返回值：JSON

示例：

```
m_db=# SELECT json_keys('{"a": {"b": 2, "c": 3}, "d": 4}');
 json_keys
------------
 ["a", "d"]
(1 row)
```



```
m_db=# SELECT json_keys('{"a": {"b": 2, "c": 3}, "d": 4}', '$.a');
 json_keys
------------
 ["b", "c"]
(1 row)
```

JSON_LENGTH

JSON_LENGTH(json_doc[,path])

描述：返回JSON中指定路径的值的长度，如果没有指定路径，则返回JSON的长度。

- 如果json_doc或path路径为null时，或者json_doc中不存在path指定的路径时，结果返回null。
- 如果JSON格式错误、path并非是有效的路径表达式或者path中包含*或**时，会产生报错。
- 如果有null值和格式错误场景同时在可变参数列表中，则按异常的先后顺序处理，若null值顺序在前则返回null，否则产生报错。
- 如果json_doc深度超过100时，则产生报错。

返回值：BIGINT

示例：

```
-- 内嵌的数组或对象不参与计算长度。比如，{“x”：[1, 2]}的长度是1。
m_db=# SELECT json_length('{"name":"Tom", "age":24, "like":["football", "basketball"]}');
 json_length
-------------
           3
(1 row)

-- 对象的长度是对象成员的数量，比如，{“x”：1}的长度是1。
m_db=# SELECT json_length('{"a":["abc","bcd"],"b":"abc"}','$.a');
 json_length
-------------
           2
(1 row)
```

JSON_MERGE

JSON_MERGE(json_doc, json_doc[, json_doc] ...)

描述：入参为多个json_doc，当json_doc的个数大于等于2，JSON对象从可变参数列表中构造。此函数将所有传入的json_doc进行合并，并返回合并后的结果。

- 如果入参中有null，结果返回null。
- 如果有null值和格式错误场景同时在可变参数列表中，则按异常的先后顺序处理，若null值顺序在前则返回null，否则产生报错。
- 如果json_doc深度超过100时，则产生报错。
- JSON_MERGE是JSON_MERGE_PRESERVE的别名，行为与JSON_MERGE_PRESERVE一致。每次调用时都会出现“'JSON_MERGE' is deprecated and will be removed in a future release. Please use JSON_MERGE_PRESERVE/JSON_MERGE_PATCH instead.”的warning信息。

返回值：JSON

示例：


```
-- 两个数组合并为一个数组，保留所有数组中的元素。
m_db=# SELECT json_merge('[1, 2]','[2]');
 json_merge
---------------------
 [1, 2, 2]
(1 row)

-- 一个纯值会被包装成一个数组并作为数组进行合并，这个纯值作为入参必须用单引号引起来。如果表示字符则需要加双引号。
m_db=# SELECT json_merge('[3, 2]','1');
 json_merge
-------------------------------
 [3, 2, 1]
(1 row)

-- 两个对象合并为一个对象，保留所有的键和值。对象组合时会重新排序。对象和数组合并时，会将对象包装到一个数组中并作为数组进行合并。
m_db=# SELECT json_merge('{"b":"2"}','{"a":"1"}','[1,2]');
 json_merge
-------------------------------
 [{"a": "1", "b": "2"}, 1, 2]
(1 row)
```

JSON_MERGE_PATCH

JSON_MERGE_PATCH(json_doc, json_doc[, json_doc] ...)

描述：对两个或多个json_doc执行符合RFC 7396标准的合并，并返回合并后的结果，不保留具有重复键的成员。

- 如果作为参数传递的json_doc中至少有一个无效，则产生报错。
- 如果json_doc深度超过100时，则产生报错。

返回值：JSON

示例：

```
m_db=# SELECT json_merge_patch('{"b":"2"}','{"a":"1"}');
   json_merge_patch
----------------------
 {"a": "1", "b": "2"}
(1 row)

m_db=# SELECT json_merge_patch('{"b":"2"}','{"a":"1"}','[1,2]');
 json_merge_patch
------------------
 [1, 2]
(1 row)
```

JSON_MERGE_PRESERVE

JSON_MERGE_PRESERVE(json_doc, json_doc[, json_doc] ...)

描述：入参为多个json_doc，当json_doc的个数大于等于2，JSON对象从可变参数列表中构造；此函数将所有传入的json进行合并，并返回合并后的结果。

- 如果入参中有null，返回null。
- 如果有null值和格式错误场景同时在可变参数列表中，则按异常的先后顺序处理，若null值顺序在前则返回null，否则产生报错。
- 如果json_doc深度超过100时，则产生报错。

返回值：JSON

示例：

```
-- 两个数组合并为一个数组，保留所有数组中的元素。
m_db=# SELECT json_merge_preserve('[1, 2]','[2]');
 json_merge_preserve
---------------------
 [1, 2, 2]
(1 row)

-- 一个纯值会被包装成一个数组并作为数组进行合并，这个纯值作为入参必须用单引号引起来。如果表示字符则需要加双引号。
m_db=# SELECT json_merge_preserve('[3, 2]','1');
 json_merge_preserve
---------------------
 [3, 2, 1]
(1 row)

-- 两个对象合并为一个对象，保留所有的键和值。对象组合时会重新排序。对象和数组合并时，会将对象包装到一个数组中并作为数组进行合并。
m_db=# SELECT json_merge_preserve('{"b":"2"}','{"a":"1"}','[1,2]');
     json_merge_preserve
-------------------------------
 [{"a": "1", "b": "2"}, 1, 2]
(1 row)
```

JSON_OBJECT

JSON_OBJECT([key, val[, key1, val1] ...])

描述：从一个可变参数列表构造出一个JSON对象，入参必须为偶数个，key和val彼此组成键值对。当key为null时或入参个数为奇数个时报错。

返回值：JSON

示例：

```
m_db=# SELECT json_object('d',2,'c','name','b',true,'a',2,'a',NULL,'d',1);
 json_object
-----------------------------------------
 {"a": 2, "b": true, "c": "name", "d": 2}
(1 row)

m_db=# SELECT json_object();
 json_object
-------------
 {}
(1 row)
```

JSON_QUOTE

JSON_QUOTE(str)

描述：使用双引号将入参str引用后输出，使其成为一个JSON字符串值。

返回值：TEXT

示例：

```
m_db=# SELECT json_quote('123');
 json_quote
------------
 "123"
(1 row)

m_db=# SELECT json_quote('"1"');
 json_quote
```



```
------------
 "\"1\""
(1 row)

m_db=# SELECT json_quote('"NULL"');
 json_quote
------------
 "\"NULL\""
(1 row)
```

JSON_REMOVE

JSON_REMOVE(json_doc, path[, path] ...)

描述：将json_doc中path指定路径上的值删除，并返回修改后的JSON。如果json_doc中不存在指定的路径，则不修改；入参中存在多个path时，较后的path会以删除过的json_doc为基准，继续执行删除操作。

- 如果json_doc或path路径为null，则函数返回null。
- 如果json_doc格式错误、path并非是有效的路径表达式或者path中包含*或**时，则产生报错。
- 如果有path路径不存在、null值和格式错误场景同时在可变参数列表中，则按异常的先后顺序处理。
- 如果json_doc深度超过100时，则产生报错。

参数说明：json_doc、path与**表4-50**中的参数说明相同。

返回值：JSON

示例：

```
-- 可以接受多个path路径，并且从左向右依次执行，执行多个path时，以已经删除过的json为基准，来追加json_remove操作。
m_db=# SELECT json_remove('[0, 1, 2, [3, 4]]', '$[0]', '$[0]','$[0]');
 json_remove
-------------
 [[3, 4]]
(1 row)

-- 删除不存在的路径，直接返回。
m_db=# SELECT json_remove('{"A":1, "B":2}', '$.C');
 json_remove
-------------
 {"A": 1, "B": 2}
(1 row)
```

JSON_REPLACE

JSON_REPLACE(json_doc, path, val[, path, val] ...)

描述：将json_doc中path指定路径上的值替换成新值，并返回修改后的JSON。

- 如果path为json_doc中不存在的路径时，则不修改，原样返回json_doc。
- 如果json_doc或path为null，则函数返回null。
- 如果json_doc格式错误、path并非是有效的路径表达式或者path中包含*或**时，则产生报错。
- 如果有null值和格式错误场景同时在可变参数列表中，则按异常的先后顺序处理，若null值顺序在前则返回null，否则产生报错。

- 如果json_doc深度超过100时，则产生报错。

参数说明：与**表4-50**相同。

返回值：JSON

示例：

```
m_db=# SELECT json_replace('{"x": 1}', '$.x', true);
 json_replace
--------------
 {"x": true}
(1 row)

-- 如果value为字符串， json_replace()函数会将其作为字符串写入到 JSON 中。
m_db=# SELECT json_replace('{"x": 1}', '$.x', 'true');
 json_replace
--------------
 {"x": "true"}
(1 row)

-- 可以对同一个路径做多次修改，按照参数从左往右的顺序执行，最终的结果以最后一次的处理为准。
m_db=# SELECT json_replace('{"x": 1}', '$.x', true, '$.x', 123, '$.x', 'asd', '$.x', null);
 json_replace
--------------
 {"x": null}
(1 row)

-- 如果path为json_doc中不存在指定的路径时，则不修改，原样返回。
m_db=# SELECT json_replace('[1]','$.a',2);
 json_replace
--------------
 [1]
(1 row)
```

JSON_SEARCH

JSON_SEARCH(json_doc, one_or_all, search_str[, escape_char[, path]...])

描述：返回json_doc中出现指定字符串search_str的位置。

- one_or_all为one时，仅返回search_str第一次出现时的位置。
- one_or_all为all时，返回search_str所有出现时的位置。
- escape_char为单个字符用以转义search_str中出现的通配符，为null时的转义字符默认为'\'。
- path表示要返回该路径下search_str出现的位置，且path可以有多个。
- 首先对escape_char进行判断，若格式错误则报错；再对json_doc进行判断，若json_doc格式错误则报错，若为null则返回null；若one_or_all格式错误则报错，若为null则返回null；若path格式错误则报错，为null则返回null，如果有null值和格式错误场景同时在可变参数列表中，若null值顺序在前则返回null，否则产生报错。
- 如果json_doc深度超过100时，则产生报错。

返回值：JSON

示例：

```
m_db=# SELECT json_search('{"a": "444%", "d": [[[44, "444%", "4444"]]], "h": {"hh": {"hhh": "4444%"}}}',
'all', '44%4\%', null, '$.a', '$.h');
      json_search
-----------------------
 ["$.a", "$.h.hh.hhh"]
```



```
(1 row)

m_db=# SELECT json_search('{"a": "444%", "d": [[[44, "444%", "4444"]]], "h": {"hh": {"hhh": "4444%"}}}',
'one', '44%4\%', null, '$.a', '$.h');
 json_search
-------------
 "$.a"
(1 row)
```

JSON_SET

JSON_SET(json_doc, path, val[, path, val] ...)

描述：向json_doc中path指定的路径插入一个值，或将指定路径上的值替换成新值，并返回修改后的JSON。如果指定的路径上有值，则将对应的值替换成新值；如果指定的路径不存在，则在对应的路径上插入数据。

- 如果json_doc或path路径为null，则函数返回null。
- 如果json_doc格式错误、path并非是有效的路径表达式或者path中包含*或**时，则产生报错。
- 如果有null值和格式错误场景同时在可变参数列表中，则按异常的先后顺序处理，若null值顺序在前则返回null，否则产生报错。
- 如果json_doc深度超过100时，则产生报错。

参数说明：与**表4-50**相同。

返回值：JSON

示例：

```
m_db=# SELECT json_set('{"s":3}','$.s','d');
 json_set
----------
 {"s": "d"}
(1 row)

-- 输入多个path-value对时，json_set()函数会按照顺序进行set操作。比如，在同一个不存在的路径进行两次set操作，则在第一次set时新增，第二次set时修改值。
m_db=# SELECT json_set('{"s":3}','$.a','d','$.a','1');
 json_set
----------
 {"a": "1", "s": 3}
(1 row)
```

JSON_TYPE

JSON_TYPE(json_doc)

描述：返回一个给定的JSON值的类型，入参必须带单引号。不接受数值类型的入参，如果入参为字符类型，需要添加双引号。如果json_doc深度超过100时，则产生报错。

返回值：TEXT

示例：

```
-- JSON对象类型
m_db=# SELECT json_type('{"name":"tom"}');
 json_type
------------
 OBJECT
(1 row)
```



```
-- 数值类型
m_db=# SELECT json_type('123');
 json_type
------------
 INTEGER
(1 row)

-- 字符串类型
m_db=# SELECT json_type('"123"');
 json_type
------------
 STRING
(1 row)
```

JSON_UNQUOTE

JSON_UNQUOTE(json_val)

描述：取消双引号引用JSON值，并将结果作为字符串返回。

返回值：TEXT

示例：

```
m_db=# SELECT json_unquote('"NULL"');
 json_unquote
--------------
 NULL
(1 row)

m_db=# SELECT json_unquote('"abc"');
 json_unquote
--------------
 abc
(1 row)

m_db=# SELECT json_unquote('');
 json_unquote
--------------
 ''
(1 row)
```

JSON_VALID

JSON_VALID(str)

描述：判断指定入参是否是一个合法的JSON。如果指定的参数是有效的JSON，JSON_VALID函数返回1；如果不是有效的JSON，则返回0；入参为null时，则返回null。

返回值：BIGINT

示例：

```
m_db=# SELECT json_valid('{"name":"tom"}');
 json_valid
------------
          1
(1 row)

m_db=# SELECT json_valid('[}');
 json_valid
------------
          0
(1 row)
```



```
m_db=# SELECT json_valid(1);
 json_valid
------------
          0
(1 row)
```

4.5.12.2 JSON 操作符

->操作符

描述：查询JSON中指定位置的数据，返回带引号的查询结果。

示例：

```
-- 创建数据表。
m_db=# CREATE TABLE muscleape
(
  id       TINYINT UNSIGNED NOT NULL AUTO_INCREMENT,
  category JSON,
  tags     JSON,
  PRIMARY KEY (id)
);

-- 插入数据。
m_db=# INSERT INTO muscleape (category, tags) VALUES ('{"id": 1,"name": "muscleape"}','[1,2,3]');
m_db=# INSERT INTO muscleape (category, tags) VALUES
(JSON_OBJECT("id",2,"name","muscleape_q"),JSON_ARRAY(1,3,5));

-- 查询表。
m_db=# SELECT * FROM muscleape;
 id |             category              |   tags
----+-----------------------------------+-----------
 1  | {"id": 1, "name": "muscleape"}    | [1, 2, 3]
 2  | {"id": 2, "name": "muscleape_q"}  | [1, 3, 5]
(2 rows)

-- 查询json中的数据。
m_db=# SELECT id,category->'$.id',category->'$.name',tags->'$[0]',tags->'$[2]' FROM muscleape;
 id | ?column? |    ?column?     | ?column? | ?column?
----+----------+-----------------+----------+----------
 1  | 1        | "muscleape"     | 1        | 3
 2  | 2        | "muscleape_q"   | 1        | 5
(2 rows)



-- 删除表。
m_db=# DROP TABLE muscleape;
```

说明

如果->操作符左侧的数据执行了::JSON强制类型转换，其实际转换的JSON类型与GUC参数m_format_behavior_compat_options兼容性选项是否设置了cast_as_new_json有关。

->>操作符

描述：查询JSON中指定位置的数据，返回不带引号的查询结果。

示例：

```
-- 创建数据表。
m_db=# CREATE TABLE muscleape
(
  id       TINYINT UNSIGNED NOT NULL AUTO_INCREMENT,
  category JSON,
```



```
  tags     JSON,
  PRIMARY KEY (id)
);

-- 插入数据。
m_db=# INSERT INTO muscleape (category, tags) VALUES ('{"id": 1,"name": "muscleape"}','[1,2,3]');
m_db=# INSERT INTO muscleape (category, tags) VALUES
(JSON_OBJECT("id",2,"name","muscleape_q"),JSON_ARRAY(1,3,5));

-- 查询表。
m_db=# SELECT * FROM muscleape;
 id |             category              |   tags
----+-----------------------------------+-----------
 1  | {"id": 1, "name": "muscleape"}    | [1, 2, 3]
 2  | {"id": 2, "name": "muscleape_q"} | [1, 3, 5]
(2 rows)

-- 查询json中的数据。

m_db=# SELECT id,category->>'$.id',category->>'$.name',tags->>'$[0]',tags->>'$[2]' FROM muscleape;
 id | ?column? |  ?column?    | ?column? | ?column?
----+----------+--------------+----------+----------
 1  | 1        | muscleape    | 1        | 3
 2  | 2        | muscleape_q | 1        | 5
(2 rows)


-- 删除表。
m_db=# DROP TABLE muscleape;
```

说明

如果->>操作符左侧的数据执行了::JSON强制类型转换，其实际转换的JSON类型与GUC参数m_format_behavior_compat_options兼容性选项是否设置了cast_as_new_json有关。

比较操作符

描述：JSON类型之间的比较操作符支持=、<=>、<>、<、<=、>和>=运算，返回结果为true或false。如果是JSON类型和其他类型进行比较，会将其他类型转换为JSON类型后再进行比较。

JSON类型的排序规则：

- 首先比较JSON数据的类型，OPAQUE > TIMESTAMP = DATETIME > TIME > DATE > BOOL > 数组 > 对象 > 字符串类型 > DOUBLE = UINT = INT = DECIMAL > NULL，结果相等则比较内容。
- 标量之间比较值的大小，OPAQUE先进行类型之间的比较，TYPE_STRING > TYPE_VAR_STRING > TYPE_BLOB > TYPE_BIT > TYPE_VARCHAR > TYPE_YEAR，类型相同时依次比较每个字节的大小。
- 数组之间依次比较元素的大小，当其中一个数组的元素比较完之后，则元素个数多的大。对象之间比较长度，长度相等则依次比较每个键值对，先比较键key，再比较值value。

示例：

```
mydb=# SELECT CAST(TIME'12:12:00' AS JSON) < CAST(BINARY'100100100' AS JSON);
 ?column?
----------
 t
(1 row)

mydb=# SELECT CAST('{"jsnid": [true, "abc"], "tag": {"ab": 1, "b": null, "a": 2}}' AS JSON) > CAST('[1, 2,
"json", null, [[]], {}]' AS JSON);
```



```
 ?column?
----------
 f
(1 row)

mydb=# SELECT CAST('true' AS JSON) <= CAST('-1.5e2' AS JSON);
 ?column?
----------
 f
(1 row)

mydb=# SELECT CAST('"abc"' AS JSON) >= CAST('null' AS JSON);
 ?column?
----------
 t
(1 row)

mydb=# SELECT CAST('0.5e3' AS JSON) = CAST('500' AS JSON);
 ?column?
----------
 t
(1 row)
```

算术运算符

JSON类型与任意类型的运算操作符支持-、+、*及/运算，返回DOUBLE类型。对于数字类型标量、布尔类型标量和字符串类型标量的JSON类型数据，会转换为相应的DOUBLE类型后进行计算，其他类型的JSON数据无法正常转换。

示例：

```
mydb=# SELECT CAST('true' AS JSON) + CAST('1.5e2' AS JSON);
 ?column?
----------
      151
(1 row)

mydb=# SELECT CAST('"123"' AS JSON) - 1;
 ?column?
----------
      122
(1 row)

mydb=# SELECT CAST('5' AS JSON) / 2;
 ?column?
----------
   2.5000
(1 row)

mydb=# SELECT CAST('3' AS JSON) * 3.33;
 ?column?
----------
     9.99
(1 row)
```

4.5.13 窗口函数

窗口函数必须通过over_clause指定OVER子句，以确定如何将查询行划分为多个组。

须知

窗口函数在未开启精度传递参数时，其结果的精度可能会存在问题，请在开启精度GUC参数（即m_format_behavior_compat_options值包含'enable_precision_decimal'）后使用。

over_clause语法形式：

```
{ OVER (window_spec) | OVER existing_window_name }
```

其中window_spec的具体语法形式：

```
[ existing_window_name ] [ partition_clause ] [ order_clause ] [ frame_clause ]
```

- existing_window_name：引用的已存在的窗口名称。
- partition_clause：指定如何将查询行进行分组。具体语法形式：

  ```
  PARTITION BY expr [, ...]
  ```

- order_clause：指定如何对每个分组中的行进行排序。具体语法形式：

  ```
  ORDER BY expr [ ASC | DESC ] [ NULLS { FIRST | LAST } ] [, ...]
  ```

  - NULLS FIRST：指定空值在排序中位于非空值之前，指定DESC排序时的默认行为。
  - NULLS LAST：指定空值在排序中位于非空值之后，未指定DESC排序时的默认行为。
- frame_clause：为窗口函数定义一个窗口框架window frame，window frame为当前查询行的一组相关行。frame_clause可以是以下的形式：

  ```
  [ RANGE | ROWS ] frame_start
  [ RANGE | ROWS ] BETWEEN frame_start AND frame_end
  ```

  其中frame_start和frame_end形式为：

  ```
  UNBOUNDED PRECEDING
  VALUE PRECEDING
  CURRENT ROW
  VALUE FOLLOWING
  UNBOUNDED FOLLOWING
  ```

  说明

  针对使用到VALUE场景存在如下约束：

  - VALUE的值仅支持为非负数。
  - VALUE PRECEDING以及VALUE FOLLOWING仅支持在指定为ROWS场景时使用，不支持在指定为RANGE场景使用。

DENSE_RANK

DENSE_RANK() over_clause

描述：在一次查询中，按照over_clause指定的行顺序，为每一行分配一个编号，编号从1开始。排序列数值相同时，保留重复编号，遇到下一个不同值时，依然按照连续数字排列。

返回值类型：BIGINT UNSIGNED

说明

此函数在参数m_format_dev_version值为's2'或以上版本且参数m_format_behavior_compat_options值包含'enable_conflict_funcs'的情况下为M-Compatibility模式数据库实现行为，即本章节描述的行为；其他行为和《开发指南》中的“SQL参考 > 函数和操作符 > 窗口函数”章节中的dense_rank函数保持一致。

示例：

```
m_db=# CREATE TABLE employee(dname VARCHAR(20),eid VARCHAR(20),ename VARCHAR(20),hiredate
DATE,salary INT);
CREATE TABLE
m_db=# INSERT INTO employee VALUES('研发部','1001','研发1','2021-11-01',NULL);
INSERT 0 1
m_db=# INSERT INTO employee VALUES('研发部','1002','研发2','2021-11-02',5000);
```



```
INSERT 0 1
m_db=# INSERT INTO employee VALUES('研发部','1003','研发3','2021-11-03',7000);
INSERT 0 1
m_db=# INSERT INTO employee VALUES('研发部','1004','研发4','2021-11-04',7000);
INSERT 0 1
m_db=# INSERT INTO employee VALUES('研发部','1005','研发5','2021-11-05',4000);
INSERT 0 1
m_db=# INSERT INTO employee VALUES('研发部','1006','研发6','2021-11-06',4000);
INSERT 0 1
m_db=# INSERT INTO employee VALUES('销售部','1007','销售1','2021-11-01',2000);
INSERT 0 1
m_db=# INSERT INTO employee VALUES('销售部','1008','销售2','2021-11-02',NULL);
INSERT 0 1
m_db=# INSERT INTO employee VALUES('销售部','1009','销售3','2021-11-03',5000);
INSERT 0 1
m_db=# INSERT INTO employee VALUES('销售部','1010','销售4','2021-11-04',6000);
INSERT 0 1
m_db=# INSERT INTO employee VALUES('销售部','1011','销售5','2021-11-05',9000);
INSERT 0 1
m_db=# INSERT INTO employee VALUES('销售部','1012','销售6','2021-11-06',6000);
INSERT 0 1
m_db=# SELECT dname,ename,salary,DENSE_RANK() OVER(PARTITION BY dname ORDER BY salary DESC)
as rk FROM employee;
 dname  | ename | salary | rk
--------+-------+--------+----
 研发部 | 研发1 |        | 1
 研发部 | 研发3 |   7000 | 2
 研发部 | 研发4 |   7000 | 2
 研发部 | 研发2 |   5000 | 3
 研发部 | 研发5 |   4000 | 4
 研发部 | 研发6 |   4000 | 4
 销售部 | 销售2 |        | 1
 销售部 | 销售5 |   9000 | 2
 销售部 | 销售6 |   6000 | 3
 销售部 | 销售4 |   6000 | 3
 销售部 | 销售3 |   5000 | 4
 销售部 | 销售1 |   2000 | 5
(12 rows)

m_db=# DROP TABLE employee;
DROP TABLE
```

FIRST_VALUE

FIRST_VALUE(expr) [null_treatment] over_clause

描述：在一次查询中，按照over_clause指定的行顺序，返回当前窗口第一行数据中expr的值。

返回值类型：任意类型（与expr的类型有关）

示例：

```
m_db=# CREATE TABLE t_stu(id INT, name VARCHAR(30), gender VARCHAR(5), class INT, math INT, eng
INT, his INT);
CREATE TABLE
m_db=# INSERT INTO t_stu VALUES (1, 'Mike', 'm', 1, 99, 86, 67), (2, 'Judy', 'f', 1, 85, 86, 85), (3, 'Jonh', 'm',
2, 79, 59, 95), (4, 'Lily', 'f', 2, 88, 81, 80), (5, 'Ethan', 'm', 2, 77, 70, 69), (6, 'Chris', 'f', 3, 65, 96, 94), (7, 'Vio',
'f', 3, 66, 69, 93);
INSERT 0 7
m_db=# SELECT id, name, math, class, FIRST_VALUE(name) OVER (PARTITION BY class ORDER BY math
DESC) FROM t_stu;
 id | name  | math | class | first_value
----+-------+------+-------+-------------
  1 | Mike  |   99 |     1 | Mike
  2 | Judy  |   85 |     1 | Mike
  4 | Lily  |   88 |     2 | Lily
  3 | Jonh  |   79 |     2 | Lily
```



```
 5 | Ethan |   77 |     2 | Lily
 7 | Vio   |   66 |     3 | Vio
 6 | Chris |   65 |     3 | Vio
(7 rows)

m_db=# DROP TABLE t_stu;
DROP TABLE
```

LAG

LAG(expr[, N[, default]]) [null_treatment] over_clause

描述：在一次查询中取出当前行的前N行中expr对应的值。如果没有这样的行，则返回值为default。

参数：如表4-51所示。

表 4-51 LAG 函数参数说明

名称	类型	描述	取值范围
expr	任意类型	表示要进行对比的字段。	任意类型值。
N	BIGINT	表示expr的偏移量，缺省值为1。可以是以下取值形式： • 常量字面量的无符号整数。 • 用户自定义的变量。	只允许是在范围[0, 2^{31}-1]的整数值。
default	任意类型	表示默认值，缺省值为NULL。	任意类型值。
[null_treatment]	-	是否忽略NULL值。	RESPECT NULLS或IGNORE NULLS（可选，但不支持）。

返回值类型：任意类型（与default和expr有关）。

示例：

```
m_db=# CREATE TABLE employee (id INT PRIMARY KEY, name VARCHAR(20) NOT NULL, age INT);
NOTICE:  CREATE TABLE / PRIMARY KEY will create implicit index "employee_pkey" for table "employee"
CREATE TABLE
m_db=# INSERT INTO employee VALUES(2, 'Bob', 36),(3, 'John', 25),(5, 'Mary', 25),(4, 'Michael', 36),(1,
'Tom', 25);
INSERT 0 5
m_db=# SELECT id, name, LAG(name) OVER() FROM employee;
 id |  name   |   lag
----+---------+---------
  2 | Bob     |
  3 | John    | Bob
  5 | Mary    | John
  4 | Michael | Mary
  1 | Tom     | Michael
(5 rows)
```



```
m_db=# SELECT id, name, LAG(name, 2) OVER() FROM employee;
 id |  name   | lag
----+---------+------
  2 | Bob     |
  3 | John    |
  5 | Mary    | Bob
  4 | Michael | John
  1 | Tom     | Mary
(5 rows)

m_db=# SELECT id, name, LAG(name, 2, 'N/A') OVER() FROM employee;
 id |  name   | lag
----+---------+------
  2 | Bob     | N/A
  3 | John    | N/A
  5 | Mary    | Bob
  4 | Michael | John
  1 | Tom     | Mary
(5 rows)

m_db=# SELECT id, name, LAG(name, 2, 'N/A') OVER(ORDER BY id) FROM employee;
 id |  name   | lag
----+---------+------
  1 | Tom     | N/A
  2 | Bob     | N/A
  3 | John    | Tom
  4 | Michael | Bob
  5 | Mary    | John
(5 rows)

m_db=# SELECT id, age,name, LAG(name, 1, 'NA') OVER(PARTITION BY age) FROM employee;
 id | age |  name   | lag
----+-----+---------+------
  3 |  25 | John    | NA
  5 |  25 | Mary    | John
  1 |  25 | Tom     | Mary
  2 |  36 | Bob     | NA
  4 |  36 | Michael | Bob
(5 rows)

m_db=# DROP TABLE employee;
DROP TABLE
```

LAST_VALUE

LAST_VALUE(expr) [null_treatment] over_clause

描述：在一次查询中，按照over_clause指定的行顺序，返回当前窗口最后一行数据中expr的值。

返回值类型：任意类型（与expr的类型有关）

示例：

```
m_db=# CREATE TABLE t_stu(id INT, name VARCHAR(30), gender VARCHAR(5), class INT, math INT, eng INT, his INT);
CREATE TABLE
m_db=# INSERT INTO t_stu VALUES (1, 'Mike', 'm', 1, 99, 86, 67), (2, 'Judy', 'f', 1, 85, 86, 85), (3, 'Jonh', 'm', 2, 79, 59, 95), (4, 'Lily', 'f', 2, 88, 81, 80), (5, 'Ethan', 'm', 2, 77, 70, 69), (6, 'Chris', 'f', 3, 65, 96, 94), (7, 'Vio', 'f', 3, 66, 69, 93);
INSERT 0 7
m_db=# SELECT id, name, math, class, LAST_VALUE(name) OVER (PARTITION BY class ORDER BY math DESC RANGE BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING) FROM t_stu;
 id | name  | math | class | last_value
----+-------+------+-------+------------
  1 | Mike  |   99 |     1 | Judy
  2 | Judy  |   85 |     1 | Judy
```



```
 4 | Lily  |   88 |     2 | Ethan
 3 | Jonh  |   79 |     2 | Ethan
 5 | Ethan |   77 |     2 | Ethan
 7 | Vio   |   66 |     3 | Chris
 6 | Chris |   65 |     3 | Chris
(7 rows)

m_db=# DROP TABLE t_stu;
DROP TABLE
```

LEAD

LEAD(expr[, N[, default]]) [null_treatment] over_clause

描述：在一次查询中获取当前行之后N行数据中expr对应的值。如果不存在相应的行，则返回值为default。

参数：如表4-51所示。

返回值类型：任意类型（与default和expr有关）。

示例：

```
m_db=# CREATE TABLE employee (id INT PRIMARY KEY, name VARCHAR(20) NOT NULL, age INT);
NOTICE:  CREATE TABLE / PRIMARY KEY will create implicit index "employee_pkey" for table "employee"
CREATE TABLE
m_db=# INSERT INTO employee VALUES(2, 'Bob', 36),(3, 'John', 25),(5, 'Mary', 25),(4, 'Michael', 36),(1,
'Tom', 25);
INSERT 0 5
m_db=# SELECT id, name, LEAD(name) OVER() FROM employee;
 id |  name   |  lead
----+---------+---------
  2 | Bob     | John
  3 | John    | Mary
  5 | Mary    | Michael
  4 | Michael | Tom
  1 | Tom     |
(5 rows)

m_db=# SELECT id, name, LEAD(name, 2) OVER() FROM employee;
 id |  name   |  lead
----+---------+---------
  2 | Bob     | Mary
  3 | John    | Michael
  5 | Mary    | Tom
  4 | Michael |
  1 | Tom     |
(5 rows)

m_db=# SELECT id, name, LEAD(name, 2, 'N/A') OVER() FROM employee;
 id |  name   |  lead
----+---------+---------
  2 | Bob     | Mary
  3 | John    | Michael
  5 | Mary    | Tom
  4 | Michael | N/A
  1 | Tom     | N/A
(5 rows)

m_db=# SELECT id, name, LEAD(name, 2, 'N/A') OVER(ORDER BY id) FROM employee;
 id |  name   |  lead
----+---------+---------
  1 | Tom     | John
  2 | Bob     | Michael
  3 | John    | Mary
  4 | Michael | N/A
  5 | Mary    | N/A
(5 rows)
```



```
m_db=# SELECT id, age,name, LEAD(name, 1, 'NA') OVER(PARTITION BY age) FROM employee;
 id | age |  name   |  lead
----+-----+---------+---------
  3 |  25 | John    | Mary
  5 |  25 | Mary    | Tom
  1 |  25 | Tom     | NA
  2 |  36 | Bob     | Michael
  4 |  36 | Michael | NA
(5 rows)

m_db=# DROP TABLE employee;
DROP TABLE
```

NTILE

NTILE(n) over_clause

描述：将排序分区中的行尽可能平均分配给n个桶，并将桶号分配给每一行，其中n为正整数。

返回值类型：BIGINT UNSIGNED

示例：

```
m_db=# CREATE TABLE ntile_t1(val INT);
CREATE TABLE
m_db=# INSERT INTO ntile_t1 VALUES(1),(2),(3),(4),(5),(6),(7),(8),(9),(10);
INSERT 0 10
m_db=# SELECT val, NTILE(4) OVER() as groupid FROM ntile_t1;
 val | groupid
-----+---------
   1 | 1
   2 | 1
   3 | 1
   4 | 2
   5 | 2
   6 | 2
   7 | 3
   8 | 3
   9 | 4
  10 | 4
(10 rows)

m_db=# DROP TABLE ntile_t1;
DROP TABLE
```

PERCENT_RANK

PERCENT_RANK() over_clause

描述：为各组内对应值生成相对序号，即根据公式 (rank - 1) / (totalrows - 1)计算所得的值。其中rank为该值依据**RANK**函数所生成的对应序号，totalrows为该分组内的总元素个数。

返回值类型：DOUBLE PRECISION

示例：

```
m_db=# CREATE TABLE percent_rank_t1(group_id INT, data INT);
CREATE TABLE
m_db=# INSERT INTO percent_rank_t1 VALUES(1,1),(1,2),(1, 4),(1, 5),(2, 6),(2, 7),(2,8),(2,9),(2,10);
INSERT 0 9
m_db=# SELECT group_id,data,PERCENT_RANK() OVER(PARTITION BY group_id ORDER BY data) FROM
percent_rank_t1;
 group_id | data |    percent_rank
```



```
----------+------+--------------------
       1 |    1 |                  0
       1 |    2 | 0.3333333333333333
       1 |    4 | 0.6666666666666666
       1 |    5 |                  1
       2 |    6 |                  0
       2 |    7 |               0.25
       2 |    8 |                0.5
       2 |    9 |               0.75
       2 |   10 |                  1
(9 rows)

m_db=# DROP TABLE percent_rank_t1;
DROP TABLE
```

RANK

RANK() over_clause

描述：在一次查询中，按照over_clause指定的行顺序，为每一行分配一个编号，编号从1开始。排序列数值相同时，保留重复编号，遇到下一个不同值时，跳跃到总体的排名。

返回值类型：BIGINT UNSIGNED

说明

此函数在参数m_format_dev_version值为's2'或以上版本且参数m_format_behavior_compat_options值包含'enable_conflict_funcs'的情况下，为M-Compatibility模式数据库实现行为，即当前函数描述的行为；其他场景下，此函数的行为和《开发指南》中的“SQL参考 > 函数和操作符 > 窗口函数”章节中的rank函数保持一致。

示例：

```
m_db=# CREATE TABLE employee(dname VARCHAR(20),eid VARCHAR(20),ename VARCHAR(20),hiredate
DATE,salary INT);
CREATE TABLE
m_db=# INSERT INTO employee VALUES('研发部','1001','研发1','2021-11-01',NULL);
INSERT 0 1
m_db=# INSERT INTO employee VALUES('研发部','1002','研发2','2021-11-02',5000);
INSERT 0 1
m_db=# INSERT INTO employee VALUES('研发部','1003','研发3','2021-11-03',7000);
INSERT 0 1
m_db=# INSERT INTO employee VALUES('研发部','1004','研发4','2021-11-04',7000);
INSERT 0 1
m_db=# INSERT INTO employee VALUES('研发部','1005','研发5','2021-11-05',4000);
INSERT 0 1
m_db=# INSERT INTO employee VALUES('研发部','1006','研发6','2021-11-06',4000);
INSERT 0 1
m_db=# INSERT INTO employee VALUES('销售部','1007','销售1','2021-11-01',2000);
INSERT 0 1
m_db=# INSERT INTO employee VALUES('销售部','1008','销售2','2021-11-02',NULL);
INSERT 0 1
m_db=# INSERT INTO employee VALUES('销售部','1009','销售3','2021-11-03',5000);
INSERT 0 1
m_db=# INSERT INTO employee VALUES('销售部','1010','销售4','2021-11-04',6000);
INSERT 0 1
m_db=# INSERT INTO employee VALUES('销售部','1011','销售5','2021-11-05',9000);
INSERT 0 1
m_db=# INSERT INTO employee VALUES('销售部','1012','销售6','2021-11-06',6000);
INSERT 0 1
m_db=# SELECT dname,ename,salary,RANK() over (PARTITION BY dname ORDER BY salary DESC) AS rk
FROM employee;
 dname  | ename | salary | rk
--------+-------+--------+----
 研发部 | 研发1 |        | 1
 研发部 | 研发3 |   7000 | 2
```



```
 研发部 | 研发4 |   7000 | 2
 研发部 | 研发2 |   5000 | 4
 研发部 | 研发5 |   4000 | 5
 研发部 | 研发6 |   4000 | 5
 销售部 | 销售2 |        | 1
 销售部 | 销售5 |   9000 | 2
 销售部 | 销售6 |   6000 | 3
 销售部 | 销售4 |   6000 | 3
 销售部 | 销售3 |   5000 | 5
 销售部 | 销售1 |   2000 | 6
(12 rows)

m_db=# DROP TABLE employee;
DROP TABLE
```

ROW_NUMBER

ROW_NUMBER() over_clause

描述：按照over_clause指定的行顺序，为每一行分配一个编号。

返回值类型：BIGINT UNSIGNED

说明

此函数在参数m_format_dev_version值为's2'或以上版本且参数m_format_behavior_compat_options值包含'enable_conflict_funcs'的情况下为M-Compatibility模式数据库实现行为，即本章节描述的行为；其他行为和《开发指南》中的“SQL参考 > 函数和操作符 > 窗口函数”章节中的row_number函数保持一致。

示例：

```
m_db=# CREATE TABLE employee (id INT PRIMARY KEY, name VARCHAR(20) NOT NULL, age INT);
NOTICE:  CREATE TABLE / PRIMARY KEY will create implicit index "employee_pkey" for table "employee"
CREATE TABLE
m_db=# INSERT INTO employee VALUES(2, 'Bob', 36),(3, 'John', 25),(5, 'Mary', 25),(4, 'Michael', 36),(1,
'Tom', 25);
INSERT 0 5
m_db=# SELECT id, name, ROW_NUMBER() OVER() FROM employee;
 id |  name   | row_number
----+---------+------------
  2 | Bob     |          1
  3 | John    |          2
  5 | Mary    |          3
  4 | Michael |          4
  1 | Tom     |          5
(5 rows)

m_db=# SELECT id, name, ROW_NUMBER() OVER(ORDER BY id) FROM employee;
 id |  name   | row_number
----+---------+------------
  1 | Tom     |          1
  2 | Bob     |          2
  3 | John    |          3
  4 | Michael |          4
  5 | Mary    |          5
(5 rows)

m_db=# SELECT id, age,name, ROW_NUMBER() OVER(PARTITION BY age) FROM employee;
 id | age |  name   | row_number
----+-----+---------+------------
  3 |  25 | John    |          1
  5 |  25 | Mary    |          2
  1 |  25 | Tom     |          3
  2 |  36 | Bob     |          1
  4 |  36 | Michael |          2
(5 rows)
```



```
m_db=# DROP TABLE employee;
DROP TABLE
```

4.5.14 加解密函数

AES_DECRYPT

AES_DECRYPT(crypt_str, key_str[, init_vector][, kdf_name][, salt][, info | iterations])

描述：基于AES算法，使用解密口令key_str、初始化向量init_vector和KDF相关选项（kdf_name、salt、info/iterations）对加密后的字符串crypt_str进行解密。

参数解释：

- crypt_str：需要被解密的字符串。若crypt_str为NULL，函数返回NULL。
- key_str：解密口令。若key_str为NULL，函数返回NULL。为了安全，对于128bit/192bit/256bit的密钥长度（由块加密模式block_encryption_mode确定），建议用户使用128bit/192bit/256bit的安全随机数作为密钥字符串。
- init_vector：为需要它的块加密模式提供初始化变量，长度大于等于16字节（大于16字节的部分会被自动忽略）。str和key_str均不为NULL时，该参数不可为NULL，否则报错。为了安全，建议用户在OFB模式下，保证每次加密IV值的唯一性；在CBC模式和CFB模式下，保证每次加密的IV值不可被预测。
- kdf_name：指定使用的KDF算法，取值为PBKDF2或HKDF。
- salt：用来与原密码组合生成密钥。
- info/iterations：对于HKDF，INFO是该字段指定的info参数。对于PBKDF2，ITERATIONS是该字段指定的迭代次数。

返回值类型：统一返回LONGBLOB类型。

示例：

```
m_db=# SELECT
AES_DECRYPT(AES_ENCRYPT('huwei123','123456vfhex4dyu','vdaladhjsadadabcdef','hkdf'),'123456vfhex4dyu',
'vdaladhjsadadabcdef','hkdf');
 aes_decrypt
-------------
 huwei123
(1 row)
```


说明

- 由于该函数的执行过程需要传入解密口令，为了安全起见，gsql工具不会将包含该函数名字样的SQL记录入执行历史；即无法在gsql里通过上下翻页功能找到该函数的执行历史。
- 在存储过程的相关操作中需尽量避免调用该函数，避免敏感参数信息在日志中泄露的风险。同时建议用户在使用包含该函数的存储过程相关操作时，将该函数的参数信息过滤后再提供给外部维护人员定位，日志使用完后请及时删除。
- 在打开debug_print_plan开关的情况下需尽量避免调用该函数，避免敏感参数信息在日志中泄露的风险。同时建议用户在打开debug_print_plan开关生成的日志中对该函数的参数信息进行过滤后再提供给外部维护人员定位，日志使用完后请及时删除。
- 若想成功解密，需要保证block_encryption_mode，key_str，iv值与加密时一致。
- 由于编码差异，不支持从gsql客户端直接复制加密后的数据进行解密，此场景解密出的结果不一定是加密前的字符串。
- 由于SQL_ASCII设置与其他设置表现得相当不同。如果服务器字符集是SQL_ASCII，服务器把字节值0~127根据 ASCII标准解释，而字节值128~255则当作无法解析的字符。如果设置为SQL_ASCII，就不会有编码转换。该函数调用openssl三方库返回的数据的编码为非ASCII数据，因此当数据库服务端字符集设置为SQL_ASCII时，客户端编码也需设置为SQL_ASCII ，否则会报错。因为数据库不会帮助转换或者校验非ASCII字符。

AES_ENCRYPT

AES_ENCRYPT(str, key_str[, init_vector][, kdf_name][, salt][, info | iterations])

描述：基于AES算法，使用加密口令key_str、初始化向量init_vector和KDF相关选项（kdf_name、salt、info/iterations）对字符串str进行加密。

参数解释：

- str：需要被加密的字符串。若str为NULL，函数返回NULL。
- key_str：加密口令。若key_str为NULL，函数返回NULL。为了安全，对于128bit/192bit/256bit的密钥长度（由块加密模式block_encryption_mode确定），建议用户使用128bit/192bit/256bit的安全随机数作为密钥字符串。
- init_vector：为需要它的块加密模式提供初始化变量，长度大于等于16字节（大于16字节的部分会被自动忽略）。str和key_str均不为NULL时，该参数不可为NULL，否则报错。为了安全，建议用户在OFB模式下，保证每次加密IV值的唯一性；在CBC模式和CFB模式下，保证每次加密的IV值不可被预测。
- kdf_name：指定使用的KDF算法，取值为PBKDF2或HKDF。
- salt：用来与原密码组合生成密钥。
- info/iterations：对于HKDF，INFO是该字段指定的info参数。对于PBKDF2，ITERATIONS是该字段指定的迭代次数。

返回值类型：统一返回LONGBLOB类型。

示例：

```
m_db=# SELECT AES_ENCRYPT('huwei123','123456vfhex4dyu,vdaladhjsadad','1234567890123456');
   aes_encrypt
-------------------
 Z_Z\x7F\\x7Fy\x1D
(1 row)
```


说明

- 由于该函数的执行过程需要传入加密口令，为了安全起见，gsql工具不会将包含该函数名字样的SQL记录入执行历史；即无法在gsql里通过上下翻页功能找到该函数的执行历史。
- 在存储过程的相关操作中需尽量避免调用该函数，避免敏感参数信息在日志中泄露的风险。同时建议用户在使用包含该函数的存储过程相关操作时，将该函数的参数信息过滤后再提供给外部维护人员定位，日志使用完后请及时删除。
- 在打开debug_print_plan开关的情况下需尽量避免调用该函数，避免敏感参数信息在日志中泄露的风险。同时建议用户在打开debug_print_plan开关生成的日志中对该函数的参数信息进行过滤后再提供给外部维护人员定位，日志使用完后请及时删除。
- 由于SQL_ASCII设置与其他设置表现得相当不同。如果服务器字符集是SQL_ASCII，服务器把字节值0~127根据 ASCII标准解释，而字节值128~255则当作无法解析的字符。如果设置为SQL_ASCII，就不会有编码转换。该函数调用openssl三方库返回的数据的编码为非ASCII数据，因此当数据库服务端字符集设置为SQL_ASCII时，客户端编码也需设置为SQL_ASCII ，否则会报错。因为数据库不会帮助转换或者校验非ASCII字符。

PASSWORD

PASSWORD(TEXT str_input)

描述：对输入的字符串进行两次SHA-1加密输出。

返回值类型：TEXT

示例：

```
m_db=# SELECT PASSWORD('Gauss');
                 password
-------------------------------------------
 *DAC4E366F59D0427EFB4CCE84C7EA705EEA2FCC6
(1 row)
```

说明

PASSWORD函数使用的SHA-1加密算法安全性低，存在安全风险，不建议使用。

4.5.15 流程控制函数

IF

IF(expr1, expr2, expr3)

描述：如果expr1为TRUE（即expr1不等于0且expr1不为NULL），IF()返回expr2；否则，IF()返回expr3。

示例：

```
m_db=# SELECT IF(true, 'ab', 1);
 case
------
 ab
(1 row)

m_db=# SELECT IF(1>2, 1, NULL);
 case
------

(1 row)

m_db=# SELECT IF(NULL, 'TRUE', 'FALSE');
 case
```



```
-------
 FALSE
(1 row)
```

IFNULL

IFNULL(expr1, expr2)

描述：如果expr1不为null，IFNULL()返回expr1；如果expr1为null，IFNULL()返回expr2。

示例：

```
m_db=# select ifnull(12, 123);
 ifnull
--------
     12
(1 row)

m_db=# select ifnull(null, 123);
 ifnull
--------
    123
(1 row)
```

NULLIF

NULLIF(expr1, expr2)

描述：如果expr1等于expr2，返回空值；否则，返回expr1。

示例：

```
m_db=# select nullif(123, 123);
 nullif
--------

(1 row)

m_db=# select nullif(12, 123);
 nullif
--------
     12
(1 row)
```

4.5.16 内部函数

M-Compatibility数据库的内部函数，不建议用户直接使用。

attname_eq_mysql	avg_any	avg_collect	avg_final	binary_binary_mysql	binary_blob_mysql	binary_bool_m
binary_char_mysql	binary_datetime_mysql	binary_float_mysql	binary_in_mysql	binary_int4_mysql	binary_int8_mysql	binary_longblob_mysql
binary_mediumblob_mysql	binary_mysql	binary_numeric_mysql	binary_out_mysql	binary_recv_mysql	binary_send_mysql	binary_str_eq_mysql

binary_str_equal_mysql	binary_str_ge_mysql	binary_str_gt_mysql	binary_str_le_mysql	binary_str_lt_mysql	binary_str_ne_mysql	binary_text_mysql
binary_tinyblob_mysql	binary_typmodin_mysql	binary_typmodout_mysql	binary_uint8_mysql	binarycmp_mysql	binarylikebinary	bit
bit_and_any	bit_binary_mysql	bit_blob_mysql	bit_bool_m	bit_char_mysql	bit_date_mysql	bit_datetime_mysql
bit_double_mysql	bit_equal_mysql	bit_float_mysql	bit_func_binary_mysql	bit_func_char_mysql	bit_func_varbinary_mysql	bit_func_varchar_mysql
bit_in	bit_in_for_copy	bit_in_for_load	bit_int4_mysql	bit_int8_mysql	bit_mediumint_mysql	bit_neg_any
bit_numeric_mysql	bit_or_any	bit_out	bit_out_for_select_into	bit_recv	bit_send	bit_set_m
bit_shift_left_any	bit_shift_right_any	bit_smallint_mysql	bit_text_mysql	bit_time_mysql	bit_timestamp_mysql	bit_tinyint_mysql
bit_uint1_mysql	bit_uint2_mysql	bit_uint3_mysql	bit_uint4_mysql	bit_uint8_mysql	bit_varbinary_mysql	bit_varchar_mysql
bit_xor_any	bit_year_mysql	bittypmodin	bittypmodout	blob_binary_mysql	blob_bit_mysql	blob_bool_m
blob_char_mysql	blob_date_mysql	blob_datetime_mysql	blob_int4_mysql	blob_int8_mysql	blob_numeric_mysql	blob_recv_mysql
blob_send_mysql	blob_text_mysql	blob_uint8_mysql	bloblikebinary	blobtypmodin	blobtypmodout	bool_binary_mysql
bool_bit_mysql	bool_bool_equal_mysql	bool_char_mysql	bool_double_mysql	bool_int1_mysql	bool_int2_mysql	bool_int4_mysql
bool_int8_mysql	bool_mediumint_mysql	bool_numeric_mysql	bool_real_mysql	bool_set_m	bool_uint1_mysql	bool_uint2_mysql
bool_uint3_mysql	bool_uint4_mysql	bool_uint8_mysql	booltext_mysql	bpchar	bpchar_binary_mysql	bpchar_bit_mysql

bpchar_bpchar_mysql	bpchar_date	bpchar_datetime_mysql	bpchar_equal_mysql	bpchar_json_m	bpchar_longtext_eq_m	bpchar_longtext_ge_m
bpchar_longtext_gt_m	bpchar_longtext_le_m	bpchar_longtext_lt_m	bpchar_mediumtext_eq_m	bpchar_mediumtext_ge_m	bpchar_mediumtext_gt_m	bpchar_mediumtext_le_m
bpchar_mediumtext_lt_m	bpchar_numeric	bpchar_text_cmp_m	bpchar_text_eq_m	bpchar_text_ge_m	bpchar_text_gt_m	bpchar_text_le_m
bpchar_text_lt_m	bpchar_time	bpchar_timestamp	bpchar_tinytext_eq_m	bpchar_tinytext_ge_m	bpchar_tinytext_gt_m	bpchar_tinytext_le_m
bpchar_tinytext_lt_m	bpchar_year	bpcharcmp	bpchareq	bpcharge	bpchargt	bpcharin
bpcharle	bpcharlike	bpcharlt	bpcharne	bpcharout	bpcharrecv	bpcharsend
bpchartypmodin	btequalimage	btfloat48cmp	btfloat4cmp	btfloat84cmp	btfloat8cmp	btint1sortsupport_m
byteawithoutorderwithequalcoltypmodin	cast_to_binary	cast_to_cstring	cast_to_datetime	cast_to_float8	cast_to_int8	cast_to_json
cast_to_numeric	cast_to_time	cast_to_uint8	convert_column_to_json_m	date_binary_mysql	date_bit_mysql	date_blob
date_bool_m	date_bpchar	date_cmp_datetime	date_date_mysql	date_datetime_eq_mysql	date_datetime_equal_mysql	date_datetime_ge_mysql
date_datetime_gt_mysql	date_datetime_le_mysql	date_datetime_lt_mysql	date_datetime_mysql	date_datetime_ne_mysql	date_eq_timestamp	date_equal_mysql
date_float4_mysql	date_float8_mysql	date_func_bit_mysql	date_func_int1_mysql	date_func_int2_mysql	date_func_int3_mysql	date_func_int4_mysql
date_func_int8_mysql	date_func_uint1_mysql	date_func_uint2_mysql	date_func_uint3_mysql	date_func_uint4_mysql	date_func_uint8_mysql	date_func_year_mysql

date_ge_timestamp	date_gt_timestamp	date_in	date_int1_mysql	date_int2_mysql	date_int3_mysql	date_int4_mysql
date_int8_mysql	date_le_timestamp	date_literal_mysql	date_longblob	date_longtext	date_mediumblob	date_mediumtext
date_ne_timestamp	date_numeric_mysql	date_out	date_row_bit_mysql	date_row_float4_mysql	date_row_float8_mysql	date_row_int1_mysql
date_row_int2_mysql	date_row_int3_mysql	date_row_int4_mysql	date_row_int8_mysql	date_row_numeric_mysql	date_row_time_mysql	date_row_uint1_mysql
date_row_uint2_mysql	date_row_uint3_mysql	date_row_uint4_mysql	date_row_uint8_mysql	date_row_year_mysql	date_set_m	date_text
date_time_mysql	date_timestamp_equal_mysql	date_timestamp_lt_mysql	date_timestamp_mysql	date_tinyblob	date_tinytext	date_uint1_mysql
date_uint2_mysql	date_uint3_mysql	date_uint4_mysql	date_uint8_mysql	date_varbinary_mysql	date_varchar	date_year_mysql
datetime_binary_mysql	datetime_bit_mysql	datetime_blob	datetime_bool_m	datetime_bpchar_mysql	datetime_cmp_date	datetime_cmp_time
datetime_cmp_timestamp	datetime_date_eq_mysql	datetime_date_equal_mysql	datetime_date_ge_mysql	datetime_date_gt_mysql	datetime_date_le_mysql	datetime_date_lt_mysql
datetime_date_mysql	datetime_date_ne_mysql	datetime_float4_mysql	datetime_float8_mysql	datetime_func_bit_mysql	datetime_func_year_mysql	datetime_in
datetime_index	datetime_int1_mysql	datetime_int2_mysql	datetime_int3_mysql	datetime_int4_mysql	datetime_int8_mysql	datetime_literal_mysql
datetime_longblob	datetime_longtext	datetime_mediumblob	datetime_mediumtext	datetime_mi	datetime_mi_interval	datetime_numeric_mysql
datetime_out	datetime_pl_interval	datetime_recv	datetime_row_bit_mysql	datetime_row_date_mysql	datetime_row_float4_mysql	datetime_row_float8_mysql

datetime_row_int1_mysql	datetime_row_int2_mysql	datetime_row_int3_mysql	datetime_row_int4_mysql	datetime_row_int8_mysql	datetime_row_numeric_mysql	datetime_row_time_mysql
datetime_row_uint1_mysql	datetime_row_uint2_mysql	datetime_row_uint3_mysql	datetime_row_uint4_mysql	datetime_row_uint8_mysql	datetime_row_year_mysql	datetime_scale_mysql
datetime_send	datetime_set_m	datetime_text_mysql	datetime_time_mysql	datetime_timestamp_eq_mysql	datetime_timestamp_equal_mysql	datetime_timestamp_ge_mysql
datetime_timestamp_gt_mysql	datetime_timestamp_le_mysql	datetime_timestamp_lt_mysql	datetime_timestamp_mysql	datetime_timestamp_ne_mysql	datetime_tinyblob	datetime_tinytext
datetime_uint1_mysql	datetime_uint2_mysql	datetime_uint3_mysql	datetime_uint4_mysql	datetime_uint8_mysql	datetime_varbinary_mysql	datetime_varchar_mysql
datetime_year	datetimetypmodin	datetimetypmodout	default_internal	div_any	double_bool_m	dtof
dtoi2	dtoi4	dtoi8	eq_binary	eq_datetime	equal_binary	equal_datetime
equal_float	equal_int	equal_numeric	equal_string	equal_time	f4toi1	f8toi1
float4_binary_mysql	float4_bit_mysql	float4_blob_mysql	float4_bpchar	float4_int3_mysql	float4_mysql	float4_numeric
float4_row_blob_mysql	float4_row_text_mysql	float4_set_m	float4_text	float4_typmod_in_mysql	float4_typmod_out_mysql	float4_uint1_mysql
float4_uint2_mysql	float4_uint3_mysql	float4_uint4_mysql	float4_uint8_mysql	float4_varbinary_mysql	float4_varchar	float4_year
float48eq	float48equal_mysql	float48ge	float48gt	float48le	float48lt	float48ne
float4eq	float4equal_mysql	float4ge	float4gt	float4in	float4le	float4lt
float4ne	float4out	float4recv	float4send	float8	float8_binary_mysql	float8_bit_mysql

float8_blob_mysql	float8_bpchar	float8_date_mysql	float8_datetime_mysql	float8_int3_mysql	float8_mysql	float8_numeric
float8_set_m	float8_text	float8_time_mysql	float8_timestamp_mysql	float8_typmod_in_mysql	float8_typmod_out_mysql	float8_uint1_mysql
float8_uint2_mysql	float8_uint3_mysql	float8_uint4_mysql	float8_uint8_mysql	float8_varbinary_mysql	float8_varchar	float8_year
float84eq	float84equal_mysql	float84ge	float84gt	float84le	float84lt	float84ne
float8eq	float8ge	float8gt	float8in	float8le	float8lt	float8ne
float8out	float8recv	float8send	ftod	ftoi2	ftoi4	ftoi8
ge_binary	ge_datetime	get_instr_rt_percentile	group_concat_finalfn_mysql	group_concat_transfn_mysql	gs_get_col_attribute	gs_get_schemadef
gs_get_viewdef	gs_logical_decode_lock	gt_binary	gt_datetime	hashfloat4	hashfloat8	hashint1
hashjsonm	hashsettext_m	i1tobinary_mysql	i1tobit_mysql	i1toblob_mysql	i1tof4	i1tof8
i1toi2	i1toi3_mysql	i1toi4	i1toi8	i1toui1_mysql	i1toui2_mysql	i1toui3_mysql
i1toui4_mysql	i1toui8_mysql	i1tovarbinary_mysql	i1typmodin_mysql	i1typmodout_mysql	i2tobinary_mysql	i2tobit_mysql
i2toblob_mysql	i2tod_mysql	i2tof_mysql	i2toi1	i2toi3_mysql	i2toi4_mysql	i2toui1_mysql
i2toui2_mysql	i2toui3_mysql	i2toui4_mysql	i2toui8_mysql	i2tovarbinary_mysql	i2typmodin_mysql	i2typmodout_mysql
i3tobinary_mysql	i3tobit_mysql	i3toblob_mysql	i3tobpchar_mysql	i3tof4_mysql	i3tof8_mysql	i3toi1_mysql
i3toi2_mysql	i3toi4_mysql	i3toi8_mysql	i3tonum_mysql	i3totext_mysql	i3toui3_mysql	i3tovarbinary_mysql
i3tovarchar_mysql	i4tobinary_mysql	i4toblob_mysql	i4tod_mysql	i4tof_mysql	i4toi1	i4toi3_mysql

i4toui1_mysql	i4toui2_mysql	i4toui3_mysql	i4toui4_mysql	i4toui8_mysql	i4tovarbinary_mysql	i4typmodin_mysql
i4typmodout_mysql	i8tobinary_mysql	i8toblob_mysql	i8tod_mysql	i8tof_mysql	i8toi1	i8toi3_mysql
i8toui1_mysql	i8toui2_mysql	i8toui3_mysql	i8toui4_mysql	i8toui8_mysql	i8tovarbinary_mysql	i8typmodin_mysql
i8typmodout_mysql	int_div_any	int1_bpchar	int1_equal_mysql	int1_int16_mysql	int1_int2_cmp_m	int1_int2_eq_m
int1_int2_equal_m	int1_int2_ge_m	int1_int2_gt_m	int1_int2_le_m	int1_int2_lt_m	int1_int2_ne_m	int1_int3_cmp_m
int1_int3_eq_m	int1_int3_equal_m	int1_int3_ge_m	int1_int3_gt_m	int1_int3_le_m	int1_int3_lt_m	int1_int3_ne_m
int1_int4_cmp_m	int1_int4_eq_m	int1_int4_equal_m	int1_int4_ge_m	int1_int4_gt_m	int1_int4_le_m	int1_int4_lt_m
int1_int4_ne_m	int1_int8_cmp_m	int1_int8_eq_m	int1_int8_equal_m	int1_int8_ge_m	int1_int8_gt_m	int1_int8_le_m
int1_int8_lt_m	int1_int8_ne_m	int1_numeric	int1_set_m	int1_text	int1_uint_eq_m	int1_uint_equal_m
int1_uint_ge_m	int1_uint_gt_m	int1_uint_le_m	int1_uint_lt_m	int1_uint_ne_m	int1_varchar	int16_int1_mysql
int16_int3_mysql	int16_set_m	int16_uint1_mysql	int16_uint2_mysql	int16_uint3_mysql	int16_uint4_mysql	int16_uint8_mysql
int1cmp	int1in	int1out	int1recv	int1send	int2	int2
int2	int2_bpchar	int2_int1_cmp_m	int2_int1_eq_m	int2_int1_equal_m	int2_int1_ge_m	int2_int1_gt_m
int2_int1_le_m	int2_int1_lt_m	int2_int1_ne_m	int2_int2_equal_mysql	int2_int3_cmp_mysql	int2_int3_eq_mysql	int2_int3_equal_mysql
int2_int3_ge_mysql	int2_int3_gt_mysql	int2_int3_le_mysql	int2_int3_lt_mysql	int2_int3_ne_mysql	int2_int4_equal_mysql	int2_int8_equal_mysql
int2_numeric_mysql	int2_set_m	int2_text	int2_uint_eq_mysql	int2_uint_equal_mysql	int2_uint_ge_mysql	int2_uint_gt_mysql

int2_uint_le_mysql	int2_uint_lt_mysql	int2_uint_ne_mysql	int2_varchar	int28_mysql	int2in	int2out
int3_int1_cmp_m	int3_int1_eq_m	int3_int1_equal_m	int3_int1_ge_m	int3_int1_gt_m	int3_int1_le_m	int3_int1_lt_m
int3_int1_ne_m	int3_int16_mysql	int3_int2_cmp_mysql	int3_int2_eq_mysql	int3_int2_equal_mysql	int3_int2_ge_mysql	int3_int2_gt_mysql
int3_int2_le_mysql	int3_int2_lt_mysql	int3_int2_ne_mysql	int3_int4_cmp_mysql	int3_int4_eq_mysql	int3_int4_equal_mysql	int3_int4_ge_mysql
int3_int4_gt_mysql	int3_int4_le_mysql	int3_int4_lt_mysql	int3_int4_ne_mysql	int3_int8_cmp_mysql	int3_int8_eq_mysql	int3_int8_equal_mysql
int3_int8_ge_mysql	int3_int8_gt_mysql	int3_int8_le_mysql	int3_int8_lt_mysql	int3_int8_ne_mysql	int3_set_m	int3_uint_eq_mysql
int3_uint_equal_mysql	int3_uint_ge_mysql	int3_uint_gt_mysql	int3_uint_le_mysql	int3_uint_lt_mysql	int3_uint_ne_mysql	int4
int4_bit_mysql	int4_bpchar	int4_int1_cmp_m	int4_int1_eq_m	int4_int1_equal_m	int4_int1_ge_m	int4_int1_gt_m
int4_int1_le_m	int4_int1_lt_m	int4_int1_ne_m	int4_int2_equal_mysql	int4_int3_cmp_mysql	int4_int3_eq_mysql	int4_int3_equal_mysql
int4_int3_ge_mysql	int4_int3_gt_mysql	int4_int3_le_mysql	int4_int3_lt_mysql	int4_int3_ne_mysql	int4_int4_equal_mysql	int4_int8_equal_mysql
int4_numeric_mysql	int4_set_m	int4_text	int4_uint_eq_mysql	int4_uint_equal_mysql	int4_uint_ge_mysql	int4_uint_gt_mysql
int4_uint_le_mysql	int4_uint_lt_mysql	int4_uint_ne_mysql	int4_varchar	int4_year_mysql	int48_mysql	int4in
int4out	int8	int8_bit_mysql	int8_bpchar	int8_date_mysql	int8_datetime_mysql	int8_int1_cmp_m
int8_int1_eq_m	int8_int1_equal_m	int8_int1_ge_m	int8_int1_gt_m	int8_int1_le_m	int8_int1_lt_m	int8_int1_ne_m
int8_int2_equal_mysql	int8_int3_cmp_mysql	int8_int3_eq_mysql	int8_int3_equal_mysql	int8_int3_ge_mysql	int8_int3_gt_mysql	int8_int3_le_mysql

int8_int3_lt_mysql	int8_int3_ne_mysql	int8_int4_equal_mysql	int8_numeric_mysql	int8_set_m	int8_text	int8_time_mysql
int8_timestamp_mysql	int8_uint_eq_mysql	int8_uint_equal_mysql	int8_uint_ge_mysql	int8_uint_gt_mysql	int8_uint_le_mysql	int8_uint_lt_mysql
int8_uint_ne_mysql	int8_varchar	int84_mysql	int8eq	int8ge	int8gt	int8in
int8le	int8lt	int8ne	int8out	integer_time	interval_pl_datetime	is_false
is_not_false	is_not_true	is_true	json_binary_m	json_bit_m	json_blob_m	json_bool_m
json_char_m	json_cmp_m	json_date_m	json_datetime_m	json_eq_m	json_equal_m	json_float4_m
json_float8_m	json_ge_m	json_gt_m	json_in_m	json_int1_m	json_int2_m	json_int3_m
json_int4_m	json_int8_m	json_le_m	json_longblob_m	json_longtext_m	json_lt_m	json_mediumblob_m
json_mediumtext_m	json_ne_m	json_numeric_m	json_out_m	json_recv_m	json_send_m	json_text_m
json_time_m	json_timestamp_m	json_tinyblob_m	json_tinytext_m	json_to_set_m	json_uint1_m	json_uint2_m
json_uint3_m	json_uint4_m	json_uint8_m	json_varbinary_m	json_varchar_m	json_year_m	jsonm_array_element_m
jsonm_array_element_text_m	jsonm_object_field_m	jsonm_object_field_text_m	le_binary	le_datetime	like	like_any
logical_oper_bool	longblob_binary_mysql	longblob_char_mysql	longblob_date_mysql	longblob_datetime_mysql	longblob_in_mysql	longblob_int4_mysql
longblob_int8_mysql	longblob_numeric_mysql	longblob_out_mysql	longblob_recv_mysql	longblob_send_mysql	longblob_text_mysql	longblob_uint8_mysql
longbloblikebinary	longtext_binary_mysql	longtext_bpchar_eq_m	longtext_bpchar_ge_m	longtext_bpchar_gt_m	longtext_bpchar_le_m	longtext_bpchar_lt_m

longtext_char_mysql	longtext_date_mysql	longtext_datetime_mysql	longtext_float_mysql	longtext_in_mysql	longtext_int4_mysql	longtext_numeric_mysql
longtext_out_mysql	longtext_recv_mysql	longtext_send_mysql	longtext_uint8_mysql	longtextlike	lt_binary	lt_datetime
max_any	mediumblob_binary_mysql	mediumblob_char_mysql	mediumblob_date_mysql	mediumblob_datetime_mysql	mediumblob_in_mysql	mediumblob_int4_mysql
mediumblob_int8_mysql	mediumblob_numeric_mysql	mediumblob_out_mysql	mediumblob_recv_mysql	mediumblob_send_mysql	mediumblob_text_mysql	mediumblob_uint8_mysql
mediumbloblikebinary	mediumblobtypmodin	mediumblobtypmodout	mediumint_bool_m	mediumint_eq_mysql	mediumint_equal_mysql	mediumint_ge_mysql
mediumint_gt_mysql	mediumint_le_mysql	mediumint_lt_mysql	mediumint_ne_mysql	mediumintin_mysql	mediumintout_mysql	mediumintrecv_mysql
mediumintsend_mysql	mediumint_typmodin_mysql	mediumint_typmodout_mysql	mediumtext_binary_mysql	mediumtext_bpchar_eq_m	mediumtext_bpchar_ge_m	mediumtext_bpchar_gt_m
mediumtext_bpchar_le_m	mediumtext_bpchar_lt_m	mediumtext_char_mysql	mediumtext_date_mysql	mediumtext_datetime_mysql	mediumtext_float_mysql	mediumtext_in_mysql
mediumtext_int4_mysql	mediumtext_numeric_mysql	mediumtext_out_mysql	mediumtext_recv_mysql	mediumtext_send_mysql	mediumtext_uint8_mysql	mediumtextlike
mediumtexttypmodin	mediumtexttypmodout	min_any	minus_any	mod_any	mul_any	mysql_compare_binary
mysql_compare_datetime	mysql_compare_float	mysql_compare_int	mysql_compare_numeric	mysql_compare_string	mysql_compare_time	ne_binary
ne_datetime	neg_any	nlike_any	numeric	numeric_bit_mysql	numeric_blob_mysql	numeric_bool
numeric_bpchar	numeric_date_mysql	numeric_datetime_mysql	numeric_eq	numeric_float4	numeric_float8	numeric_ge

numeric_gt	numeric_in	numeric_int1	numeric_int2	numeric_int4	numeric_int8	numeric_le
numeric_lt	numeric_mediumint_mysql	numeric_ne	numeric_out	numeric_recv	numeric_send	numeric_set_m
numeric_text	numeric_time_mysql	numeric_timestamp_mysql	numeric_uint1_mysql	numeric_uint2_mysql	numeric_uint3_mysql	numeric_uint4_mysql
numeric_uint8_mysql	numeric_varbinary_m	numeric_varchar_m	numeric_year_mysql	numerictypmodin	numerictypmodout	plus_any
rawin	rawout	real_bool_m	real_date	real_datetime	real_time	real_timestamp
schema	set_bit_m	set_bool_m	set_date_m	set_datetime_m	set_float4_m	set_float8_m
set_in_m	set_int1_m	set_int16_m	set_int2_m	set_int3_m	set_int4_m	set_int8_m
set_numeric_m	set_out_m	set_recv_m	set_send_m	set_set_m	set_text_m	set_time_m
set_timestamp_m	set_to_json_m	set_typmod_in_m	set_typmod_out_m	set_uint1_m	set_uint2_m	set_uint3_m
set_uint4_m	set_uint8_m	set_year_m	seteq_m	setge_m	setgt_m	setle_m
setlt_m	setne_m	std_any_m	std_collect_m	std_final_m	str_eq_mysql	str_equal_mysql
str_ge_mysql	str_gt_mysql	str_le_mysql	str_lt_mysql	str_ne_mysql	sum_any	sum_final
text	text_binary_mysql	text_bit_mysql	text_blob_mysql	text_bool_m	text_bpchar_cmp_m	text_bpchar_eq_m
text_bpchar_ge_m	text_bpchar_gt_m	text_bpchar_le_m	text_bpchar_lt_m	text_date	text_datetime_mysql	text_float4_mysql
text_float8_mysql	text_ge	text_gt	text_int1_mysql	text_int2_mysql	text_int4_mysql	text_int8_mysql
text_json_m	text_le	text_longblob_mysql	text_longtext_mysql	text_lt	text_mediumblob_mysql	text_mediumint_mysql

text_mediumtext_mysql	text_mysql	text_numeric	text_set_m	text_text_mysql	text_time_mysql	text_timestamp
text_tinyblob_mysql	text_tinytext_mysql	text_uint1_mysql	text_uint2_mysql	text_uint3_mysql	text_uint4_mysql	text_uint8_mysql
text_varbinary_mysql	text_varchar_mysql	text_year_mysql	texteq	texticregexeq_mysql	texticregexne_mysql	textin
textlike	textne	textout	textrecv	textsend	texttypmodin	texttypmodout
time_binary_mysql	time_bit_mysql	time_blob	time_bool_m	time_bpchar_mysql	time_cmp_datetime	time_date_mysql
time_datetime_mysql	time_eq	time_float4_mysql	time_float8_mysql	time_func_bit_mysql	time_func_year_mysql	time_ge
time_gt	time_in	time_int1_mysql	time_int2_mysql	time_int3_mysql	time_int4_mysql	time_int8_mysql
time_le	time_literal_mysql	time_longblob	time_longtext	time_lt	time_mediumblob	time_mediumtext
time_ne	time_numeric_mysql	time_out	time_recv	time_row_bit_mysql	time_row_date_mysql	time_row_datetime_mysql
time_row_float4_mysql	time_row_float8_mysql	time_row_int1_mysql	time_row_int2_mysql	time_row_int3_mysql	time_row_int4_mysql	time_row_int8_mysql
time_row_numeric_mysql	time_row_timestamp_mysql	time_row_uint1_mysql	time_row_uint2_mysql	time_row_uint3_mysql	time_row_uint4_mysql	time_row_uint8_mysql
time_row_year_mysql	time_scale_mysql	time_set_m	time_text_mysql	time_timestamp_mysql	time_tinyblob	time_tinytext
time_typmodout_mysql	time_uint1_mysql	time_uint2_mysql	time_uint3_mysql	time_uint4_mysql	time_uint8_mysql	time_varbinary_mysql
time_varchar_mysql	time_year_mysql	timestamp_binary_mysql	timestamp_bit_mysql	timestamp_blob	timestamp_bool_m	timestamp_bpchar_mysql

timestamp_cmp_date	timestamp_cmp_datetime	timestamp_date_equal_mysql	timestamp_date_lt_mysql	timestamp_datetime_eq_mysql	timestamp_datetime_equal_mysql	timestamp_datetime_ge_mysql
timestamp_datetime_gt_mysql	timestamp_datetime_le_mysql	timestamp_datetime_lt_mysql	timestamp_datetime_mysql	timestamp_datetime_ne_mysql	timestamp_eq_date	timestamp_equal_mysql
timestamp_float4_mysql	timestamp_float8_mysql	timestamp_ge_date	timestamp_gt_date	timestamp_in	timestamp_int1_mysql	timestamp_int2_mysql
timestamp_int3_mysql	timestamp_int4_mysql	timestamp_int8_mysql	timestamp_le_date	timestamp_longblob	timestamp_longtext	timestamp_mediumblob
timestamp_mediumtext	timestamp_ne_date	timestamp_numeric_mysql	timestamp_out	timestamp_row_bit_mysql	timestamp_row_date_mysql	timestamp_row_float4_mysql
timestamp_row_float8_mysql	timestamp_row_int1_mysql	timestamp_row_int2_mysql	timestamp_row_int3_mysql	timestamp_row_int4_mysql	timestamp_row_int8_mysql	timestamp_row_numeric_mysql
timestamp_row_time_mysql	timestamp_row_uint1_mysql	timestamp_row_uint2_mysql	timestamp_row_uint3_mysql	timestamp_row_uint4_mysql	timestamp_row_uint8_mysql	timestamp_row_year_mysql
timestamp_set_m	timestamp_text	timestamp_time_mysql	timestamp_tinyblob	timestamp_tinytext	timestamp_typmodout_mysql	timestamp_uint1_mysql
timestamp_uint2_mysql	timestamp_uint3_mysql	timestamp_uint4_mysql	timestamp_uint8_mysql	timestamp_varbinary_mysql	timestamp_varchar	timestamp_year_mysql
timestamptz_datetime_mysql	tinyblob_binary_mysql	tinyblob_char_mysql	tinyblob_date_mysql	tinyblob_datetime_mysql	tinyblob_in_mysql	tinyblob_int4_mysql
tinyblob_int8_mysql	tinyblob_numeric_mysql	tinyblob_out_mysql	tinyblob_recv_mysql	tinyblob_send_mysql	tinyblob_text_mysql	tinyblob_uint8_mysql
tinyblobblikebinary	tinyblobtypmodin	tinyblobtypmodout	tinytext_binary_mysql	tinytext_bpchar_eq_m	tinytext_bpchar_ge_m	tinytext_bpchar_gt_m

tinytext_bpchar_le_m	tinytext_bpchar_lt_m	tinytext_char_mysql	tinytext_date_mysql	tinytext_datetime_mysql	tinytext_float_mysql	tinytext_in_mysql
tinytext_int4_mysql	tinytext_numeric_mysql	tinytext_out_mysql	tinytext_recv_mysql	tinytext_send_mysql	tinytext_uint8_mysql	tinytextlike
tinytexttypmodin	tinytexttypmodout	to_nchar	ui1tof4_mysql	ui1tof8_mysql	ui1toi1_mysql	ui1toi2_mysql
ui1toi3_mysql	ui1toi4_mysql	ui1toi8_mysql	ui1toui2_mysql	ui1toui3_mysql	ui1toui4_mysql	ui1toui8_mysql
ui2tof4_mysql	ui2tof8_mysql	ui2toi1_mysql	ui2toi2_mysql	ui2toi3_mysql	ui2toi4_mysql	ui2toi8_mysql
ui2toui1_mysql	ui2toui3_mysql	ui2toui4_mysql	ui2toui8_mysql	ui3tof4_mysql	ui3tof8_mysql	ui3toi1_mysql
ui3toi2_mysql	ui3toi3_mysql	ui3toi4_mysql	ui3toi8_mysql	ui3toui1_mysql	ui3toui2_mysql	ui3toui4_mysql
ui3toui8_mysql	ui4tof4_mysql	ui4tof8_mysql	ui4toi1_mysql	ui4toi2_mysql	ui4toi3_mysql	ui4toi4_mysql
ui4toi8_mysql	ui4toui1_mysql	ui4toui2_mysql	ui4toui3_mysql	ui4toui8_mysql	ui8tof4_mysql	ui8tof8_mysql
ui8toi1_mysql	ui8toi2_mysql	ui8toi3_mysql	ui8toi4_mysql	ui8toi8_mysql	ui8toui1_mysql	ui8toui2_mysql
ui8toui3_mysql	ui8toui4_mysql	uint_index	uint_int1_eq_m	uint_int1_equal_m	uint_int1_ge_m	uint_int1_gt_m
uint_int1_le_m	uint_int1_lt_m	uint_int1_ne_m	uint_int2_eq_mysql	uint_int2_equal_mysql	uint_int2_ge_mysql	uint_int2_gt_mysql
uint_int2_le_mysql	uint_int2_lt_mysql	uint_int2_ne_mysql	uint_int3_eq_mysql	uint_int3_equal_mysql	uint_int3_ge_mysql	uint_int3_gt_mysql
uint_int3_le_mysql	uint_int3_lt_mysql	uint_int3_ne_mysql	uint_int4_eq_mysql	uint_int4_equal_mysql	uint_int4_ge_mysql	uint_int4_gt_mysql
uint_int4_le_mysql	uint_int4_lt_mysql	uint_int4_ne_mysql	uint_int8_eq_mysql	uint_int8_equal_mysql	uint_int8_ge_mysql	uint_int8_gt_mysql
uint_int8_le_mysql	uint_int8_lt_mysql	uint_int8_ne_mysql	uint_uint_eq_mysql	uint_uint_equal_mysql	uint_uint_ge_mysql	uint_uint_gt_mysql

uint_uint_le_mysql	uint_uint_lt_mysql	uint_uint_ne_mysql	uint1_bool_m	uint1_in_mysql	uint1_int16_mysql	uint1_numeric_mysql
uint1_out_mysql	uint1_recv_mysql	uint1_send_mysql	uint1_set_m	uint1_typmodin_mysql	uint1_typmodout_mysql	uint2_bool_m
uint2_in_mysql	uint2_int16_mysql	uint2_numeric_mysql	uint2_out_mysql	uint2_recv_mysql	uint2_send_mysql	uint2_set_m
uint2_typmodin_mysql	uint2_typmodout_mysql	uint3_bool_m	uint3_in_mysql	uint3_int16_mysql	uint3_numeric_mysql	uint3_out_mysql
uint3_recv_mysql	uint3_send_mysql	uint3_set_m	uint3_typmodin_mysql	uint3_typmodout_mysql	uint4_bool_m	uint4_in_mysql
uint4_int16_mysql	uint4_numeric_mysql	uint4_out_mysql	uint4_recv_mysql	uint4_send_mysql	uint4_set_m	uint4_typmodin_mysql
uint4_typmodout_mysql	uint8_binary_m	uint8_bit_mysql	uint8_blob_m	uint8_bool_m	uint8_bpchar_m	uint8_in_mysql
uint8_int16_mysql	uint8_numeric_mysql	uint8_out_mysql	uint8_recv_mysql	uint8_send_mysql	uint8_set_m	uint8_set_m
uint8_text_m	uint8_typmodin_mysql	uint8_typmodout_mysql	uint8_varbinary_m	uint8_varchar_m	unsupported_json_transfer_m	varbinary_char_mysql
varbinary_datetime_mysql	varbinary_float_mysql	varbinary_in_mysql	varbinary_int_mysql	varbinary_mysql	varbinary_out_mysql	varbinary_recv_mysql
varbinary_send_mysql	varbinary_text_mysql	varbinary_typmodin_mysql	varbinary_typmodout_mysql	varbinary_uint8_mysql	varbinary likebinary	varchar
varchar_binary_mysql	varchar_date	varchar_datetime_mysql	varchar_json_m	varchar_numeric	varchar_time	varchar_timestamp
varchar_uint8_mysql	varchar_year	varcharin	varcharlike	varcharout	varcharrecv	varcharsend
varchartypmodin	xor_bool	year_bigint	year_binary	year_bit_mysql	year_blob	year_bool_m

year_bpchar	year_date	year_datetime	year_eq_mysql	year_equal_mysql	year_float4	year_float8
year_ge_mysql	year_gt_mysql	year_in_mysql	year_index	year_int1_mysql	year_int3_mysql	year_integer
year_le_mysql	year_lt_mysql	year_ne_mysql	year_numeric	year_out_mysql	year_recv_mysql	year_send_mysql
year_set_m	year_smallint	year_text	year_time	year_timestamp	year_uint1_mysql	year_uint2
year_uint3_mysql	year_uint4	year_uint8	year_varbinary	year_varchar	yeartypmodin_mysql	yeartypmodout_mysql

4.5.17 其他函数

ANY_VALUE

ANY_VALUE(ANY arg)

描述：返回每个分组中指定列的第一条数据。

返回值类型：任意类型

示例：

```
m_db=# CREATE TABLE t1 (c1 INT, c2 INT);
CREATE TABLE

m_db=# INSERT INTO t1 VALUES(1, NULL), (1, 18), (2, 17), (2, 22);
INSERT 0 4

m_db=# SELECT * FROM t1;
 c1 | c2
----+----
  1 |
  1 | 18
  2 | 17
  2 | 22
(4 rows)

m_db=# SELECT ANY_VALUE(c2) FROM t1 GROUP BY c1;
 any_value
-----------

        17
(2 rows)

m_db=# DROP TABLE t1;
```

ATTNAME_EQ_MYSQL

attname_eq_mysql(name1, name2)

描述：如果name1等于name2（忽略大小写），返回true；否则，返回false。

返回值类型：BOOL

示例：

```
m_db=# SELECT attname_eq_mysql('abc', 'ABC');
 attname_eq_mysql
------------------
 t
(1 row)
```

BENCHMARK

BENCHMARK(INT count, expr)

描述：重复计算expr表达式count次，通过这种方式评估expr表达式的执行效率。正常场景下，函数的返回值始终为0，但可以根据客户端提示的执行时间控制BENCHMARK总共执行所消耗的时间。BENCHMARK函数只能测量数字表达式（scalar expression）的性能，表达式可以为一个子查询，但子查询返回的结果只能是单值。

返回值类型：INT

示例：

```
m_db=# CREATE TABLE employee (id INT PRIMARY KEY, name VARCHAR(20) NOT NULL, age INT DEFAULT
18);
NOTICE:  CREATE TABLE / PRIMARY KEY will create implicit index "employee_pkey" for table "employee"
CREATE TABLE
m_db=# INSERT INTO employee VALUES(2, 'Bob', 36),(3, 'John', 25),(5, 'Mary', 25),(4, 'Michael', 36),(1,
'Tom', 25);
INSERT 0 5
m_db=# \timing on
Timing is on.
m_db=# SELECT BENCHMARK(1000000, ABS(age)) FROM employee;
 benchmark
-----------
         0
         0
         0
         0
         0
(5 rows)

Time: 2769.498 ms
m_db=# \timing off
Timing is off.
m_db=# DROP TABLE employee;
DROP TABLE
```

COLLATION

COLLATION (TEXT str)

描述：返回指定的字符串的排序规则。

返回值类型：TEXT

示例：

```
mdb=# SELECT COLLATION('Gauss');
     collation
--------------------
 utf8mb4_general_ci
(1 row)

mdb=# SELECT COLLATION(_BINARY'Gauss');
 collation
-----------
 binary
```



```
(1 row)

mdb=# SELECT COLLATION(_GBK'Gauss');
   collation
----------------
 gbk_chinese_ci
(1 row)
```

CONNECTION_ID

CONNECTION_ID()

描述：返回当前连接的客户端会话ID。

返回值类型：BIGINT UNSIGNED

示例：

```
m_db=# SELECT connection_id();
  connection_id
-----------------
 140598681532160
(1 row)
```

DATABASE

DATEBASE()

描述：返回当前数据库（Schema）的名称。

返回值类型：TEXT

示例：

```
m_db=# SELECT DATABASE();
 database
----------
 public
(1 row)
```

DEFAULT

DEFAULT(col_name)

描述：返回指定列col_name的默认值。如果col_name没有设置默认值且没有指定NOT NULL，则返回NULL；如果col_name无默认值且定义NOT NULL则报错。

返回值类型：任意类型

示例：

```
m_db=# CREATE TABLE employee (id INT PRIMARY KEY, name VARCHAR(20) NOT NULL, age INT DEFAULT
18);
NOTICE:  CREATE TABLE / PRIMARY KEY will create implicit index "employee_pkey" for table "employee"
CREATE TABLE
m_db=# INSERT INTO employee VALUES(2, 'Bob', 36),(3, 'John', 25),(5, 'Mary', 25),(4, 'Michael', 36),(1,
'Tom', 25);
INSERT 0 5
m_db=# SELECT DEFAULT(age) FROM employee;
 default
---------
      18
      18
      18
      18
```



```
    18
(5 rows)

m_db=# DROP TABLE employee;
DROP TABLE
```

FOUND_ROWS

FOUND_ROWS()

描述：返回当前连接前一条查询语句执行结果的行数，当前一条查询语句使用SQL_CALC_FOUND_ROWS关键字配合使用时，前一条查询语句中的即使存在LIMIT子句，FOUND_ROWS函数也将返回所有行数，前一条语句不是SELECT形式的查询语句时，FOUND_ROWS函数的返回值是未定义的。

返回值类型：BIGINT

示例：

```
m_db=# CREATE TABLE employee (id INT PRIMARY KEY, name VARCHAR(20) NOT NULL, age INT);
NOTICE:  CREATE TABLE / PRIMARY KEY will create implicit index "employee_pkey" for table "employee"
CREATE TABLE
m_db=# INSERT INTO employee VALUES(2, 'Bob', 36),(3, 'John', 25),(5, 'Mary', 25),(4, 'Michael', 36),(1,
'Tom', 25);
INSERT 0 5
m_db=# SELECT * FROM employee;
 id |  name   | age
----+---------+-----
  2 | Bob     |  36
  3 | John    |  25
  5 | Mary    |  25
  4 | Michael |  36
  1 | Tom     |  25
(5 rows)

m_db=# SELECT FOUND_ROWS();
 found_rows
------------
          5
(1 row)

m_db=# SELECT * FROM employee LIMIT 3;
 id | name | age
----+------+-----
  2 | Bob  |  36
  3 | John |  25
  5 | Mary |  25
(3 rows)

m_db=# SELECT FOUND_ROWS();
 found_rows
------------
          3
(1 row)

m_db=# SELECT * FROM employee LIMIT 7;
 id |  name   | age
----+---------+-----
  2 | Bob     |  36
  3 | John    |  25
  5 | Mary    |  25
  4 | Michael |  36
  1 | Tom     |  25
(5 rows)

m_db=# SELECT FOUND_ROWS();
 found_rows
```



```
------------
         5
(1 row)

m_db=# SELECT SQL_CALC_FOUND_ROWS * FROM employee LIMIT 3;
 id | name | age
----+------+-----
  2 | Bob  |  36
  3 | John |  25
  5 | Mary |  25
(3 rows)

m_db=# SELECT FOUND_ROWS();
 found_rows
------------
         5
(1 row)

m_db=# DROP TABLE employee;
DROP TABLE
```

LAST_INSERT_ID

- last_insert_id()

  描述：返回当前会话最近一次执行的INSERT语句中，成功插入AUTO_INCREMENT列的第一个自动生成的值。

  返回值类型：BIGINT UNSIGNED

  示例：

  ```
  m_db=# CREATE TABLE t1(c1 INT PRIMARY KEY, c2 INT AUTO_INCREMENT UNIQUE)
  AUTO_INCREMENT=10;
  NOTICE:  CREATE TABLE will create implicit sequence "t1_c2_seq" for serial column "t1.c2"
  NOTICE:  CREATE TABLE / PRIMARY KEY will create implicit index "t1_pkey" for table "t1"
  NOTICE:  CREATE TABLE / UNIQUE will create implicit index "t1_c2_key" for table "t1"
  CREATE TABLE
                                  ^
  m_db=# INSERT INTO t1(c1) VALUES(1), (2), (3);
  INSERT 0 3

  m_db=# SELECT last_insert_id();
   last_insert_id
  ----------------
   10
  (1 row)
  m_db=# DROP TABLE t1;
  DROP TABLE
  ```

- last_insert_id(expr)

  描述：将last_insert_id(expr)函数的返回值作为再次执行last_insert_id()函数的返回值进行返回。若expr为NULL，则last_insert_id(expr)返回NULL，再次执行last_insert_id()函数，返回值为0。

  返回值类型：BIGINT UNSIGNED

  示例：

  ```
  m_db=# SELECT last_insert_id(100);
   last_insert_id
  ----------------
   100
  (1 row)

  m_db=# SELECT last_insert_id();
   last_insert_id
  ----------------
   100
  (1 row)
  ```


ROW_COUNT

ROW_COUNT()

描述：返回上一条语句影响行数。对于DDL语句，直接返回0；除SELECT之外的DML语句，返回影响的实际行数，SELECT语句返回-1，其中SELECT INTO返回实际影响行数。

返回值类型：BIGINT

示例：

```
m_db=# CREATE TABLE employee (id INT PRIMARY KEY, name VARCHAR(20) NOT NULL, age INT);
NOTICE:  CREATE TABLE / PRIMARY KEY will create implicit index "employee_pkey" for table "employee"
CREATE TABLE
m_db=# INSERT INTO employee VALUES(2, 'Bob', 36),(3, 'John', 25),(5, 'Mary', 25),(4, 'Michael', 36),(1,
'Tom', 25);
INSERT 0 5
m_db=# SELECT ROW_COUNT();
 row_count
-----------
         5
(1 row)

m_db=# UPDATE employee SET age=26 WHERE age=25;
UPDATE 3
m_db=# SELECT ROW_COUNT();
 row_count
-----------
         3
(1 row)

m_db=# DELETE FROM employee WHERE age=36;
DELETE 2
m_db=# SELECT ROW_COUNT();
 row_count
-----------
         2
(1 row)

m_db=# DROP TABLE employee;
DROP TABLE
```

SCHEMA

SCHEMA()

描述：返回当前Schema（数据库）的名称。

返回值类型：TEXT

示例：

```
m_db=# SELECT SCHEMA();
 schema
--------
 public
(1 row)
```

SLEEP

SLEEP(INT duration)

描述：使执行该函数的会话按照指定的秒数休眠。

返回值类型：BIGINT

示例：

```
m_db=# SELECT SLEEP(1);
 sleep
-------
     0
(1 row)
```

SYSTEM_USER

SYSTEM_USER()

描述：返回一个UTF8编码的字符串，该字符串内容包含当前客户端使用的当前账户的用户名和当前客户端主机的ip地址，格式为username@ip。

返回值类型：TEXT

示例：

```
m_db=# SELECT SYSTEM_USER();
  system_user
----------------
 oms@localhost
(1 row)
```

UUID

UUID()

描述：返回一个根据RFC 4122、ISO/IEF 9834-8:2005以及相关标准定义的通用唯一标识符（UUID）。这个标识符是一个小写十六进制数字组成的字符串，该字符串由一组8位数字、三组4位数字和一组12位数字组成，总共32个数字代表128位。

返回值类型：VARCHAR

示例：

```
m_db=# SELECT UUID();
                 uuid
--------------------------------------
 b28d6aaa-7a3a-c03c-f812-fa163e6cc336
(1 row)
```

UUID_SHORT

UUID_SHORT()

描述：返回一个在一定条件下具有唯一性的短通用标识符。这个标识符是一个64位无符号类型的整数。

在满足下列条件时，返回的值是唯一的：

- 当前集群下的服务节点数不能超过256个。
- 不能在节点重新启动之间设置服务器主机的系统时间。
- 在节点重新启动之间平均每秒调用UUID_SHORT()少于1600万次。

返回值类型：UINT64

示例：

```
m_db=# SELECT UUID_SHORT();
     uuid_short
```



```
----------------------
 1386314461119561729
(1 row)
```

4.5.18 安全函数

- pg_query_audit()

  描述：查看数据库主节点审计日志。

  多租场景：non-PDB调用该函数时返回全部审计日志，PDB下调用该函数仅返回该PDB相关审计日志。

  返回值类型：record

  函数返回字段如下：

名称	类型	描述
time	timestamp with time zone	操作时间
type	text	操作类型
result	text	操作结果
userid	oid	用户id
username	text	执行操作的用户名
database	text	数据库名称
client_conninfo	text	客户端连接信息
object_name	text	操作对象名称
detail_info	text	执行操作详细信息
node_name	text	节点名称
thread_id	text	线程id
local_port	text	本地端口
remote_port	text	远端端口

- pg_delete_audit()

  描述：删除指定时间段的审计日志。多租场景下，non-PDB调用该函数时可以删除全局指定时间段的审计日志，PDB下调用该函数仅允许删除该PDB相关指定时间段的审计日志。

  返回值类型：void

4.6 数据类型

4.6.1 概述

M-Compatibility支持的数据类型有：

- 布尔类型：BOOLEAN/BOOL。详细介绍请参见 **布尔类型**。
- 二进制类型：BINARY、VARBINARY、TINYBLOB、BLOB、MEDIUMBLOB、LONGBLOB。详细介绍请参见**表4-53**。
- 字符类型：CHAR、VARCHAR、TINYTEXT、TEXT、MEDIUMTEXT、LONGTEXT。详细介绍请参见**表4-54**。
- 时间类型：DATE、TIME、TIMESTAMP、DATETIME、YEAR。详细介绍请参见**表4-55**。
- 位串类型：BIT。详细介绍请参见**表4-56**。
- 数值类型：TINYINT、SMALLINT、MEDIUMINT、INT/INTEGER、BIGINT、NUMERIC/DECIMAL、FLOAT、DOUBLE。详情介绍请见**数值类型**。

M-Compatibility支持数据类型间的类型转换，具体转化关系请参见**类型转换**。

M-Compatibility支持数据类型的相关约束具体参见《兼容性说明》中的“MySQL兼容性说明 > MySQL兼容性M-Compatibility模式 ”章节。

4.6.2 布尔类型

BOOLEAN数据类型用来存储布尔值true和false。BOOLEAN或BOOL作为TINYINT类型的同义词，可以在BOOL或BOOLEAN列中插入1和0之外的值。M-Compatibility支持的布尔类型请参见**表4-52**。

表 4-52 布尔类型

名称	描述	存储空间	取值
BOOLEAN BOOL	布尔类型。取值范围为[-128,127]	1字节。	• 真：TRUE、1 、true、以及所有非0数值。 • 假：FALSE、0、false。 • 空值：null。

说明

SQL中建议BOOLEAN值使用true和false。

示例

显示用数字1和0输出BOOLEAN值。

```
--创建表。
m_db=# CREATE TABLE bool_type_t1
(
    BT_COL1 BOOLEAN,
    BT_COL2 TEXT
);

--插入数据。
```



```
m_db=# INSERT INTO bool_type_t1 VALUES (TRUE, 'sic est');

m_db=# INSERT INTO bool_type_t1 VALUES (FALSE, 'non est');

m_db=# INSERT INTO bool_type_t1 VALUES (123, 'sic est');

--查看数据。
m_db=# SELECT * FROM bool_type_t1;
 bt_col1 | bt_col2
---------+---------
 1       | sic est
 0       | non est
 123     | sic est
(3 rows)

m_db=# SELECT * FROM bool_type_t1 WHERE bt_col1 = 't';
WARNING:  Truncated incorrect double value: 't'
 bt_col1 | bt_col2
---------+---------
 0       | non est

m_db=# SELECT * FROM bool_type_t1 WHERE bt_col1 = '123';
 bt_col1 | bt_col2
---------+---------
 123     | sic est
(1 row)

--删除表。
m_db=# DROP TABLE bool_type_t1;
```

4.6.3 二进制类型

M-Compatibility支持的二进制类型请参见**表4-53**。

表 4-53 二进制类型

名称	描述	存储空间
BINARY[(n)]	二进制定长字符串，不足补空格。n是指字节长度，如不带精度n，默认精度为1。	最大为255个字节。
VARBINARY(n)	二进制变长字符串。n是指字节长度。	最大为65532个字节。
BLOB[(n)]	二进制大对象。n是指该类型的可选字节长度，指定n后将列创建为最小的但足以容纳n字节的BLOB类型。	最大为65535个字节。
TINYBLOB	二进制大对象。	最大为255个字节。
MEDIUMBLOB	二进制大对象。	最大为16777215个字节。
LONGBLOB	二进制大对象。	最大为1GB-1个字节。

说明

- sql_mode参数值包含“strict_trans_tables”时，非法输入或者输入超过范围时，会报error。
- sql_mode参数值不包含“strict_trans_tables”时，非法输入或者输入超过范围时，会报warning信息，并返回截断后的值。
- 不支持‘||’连接符连接数据。

示例

显示用字符串输出二进制类型。

```
--创建表。
m_db=# CREATE TABLE blob_type_t1
(
    BT_COL1 BINARY(10),
    BT_COL2 VARBINARY(10),
    BT_COL3 BLOB,
    BT_COL4 BLOB(10)
);

--插入数据。
m_db=# INSERT INTO blob_type_t1 VALUES ('blob_test1', 'blob_test2','blob_test3','blob_test4');

--查询表中的数据。
m_db=# SELECT *  FROM blob_type_t1;
  bt_col1   |  bt_col2   |  bt_col3   |  bt_col4
------------+------------+------------+------------
 blob_test1 | blob_test2 | blob_test3 | blob_test4
(1 row)

--删除表。
m_db=# DROP TABLE blob_type_t1;

--示例：BINARY类型
--创建表。
m_db=# CREATE TABLE blob_type_t2
(
    BT_COL1 BINARY(5)
);

--严格模式下插入的数据长度超过类型规定的长度报错。
m_db=# SET SQL_MODE =
'strict_trans_tables,only_full_group_by,no_zero_in_date,no_zero_date,error_for_division_by_zero,
no_auto_create_user,no_engine_substitution';
SET
m_db=# INSERT INTO blob_type_t2 VALUES ('too long');
ERROR:  value too long for type binary(5)
CONTEXT:  referenced column: bt_col1

--宽松模式下插入的数据长度超过类型规定的长度提示warning,数据截断插入。
m_db=# SET SQL_MODE = '';
SET
m_db=# INSERT INTO blob_type_t2 VALUES ('too long');
WARNING:  value too long for type binary(5)
CONTEXT:  referenced column: bt_col1
INSERT 0 1

--查询表中的数据。
m_db=# SELECT * FROM blob_type_t2;
 bt_col1
---------
 too l
(1 row)

--删除表。
m_db=# DROP TABLE blob_type_t2;
```


4.6.4 字符类型

M-Compatibility支持的字符类型请参见表4-54。

表 4-54 字符类型

名称	描述	存储空间
CHAR[(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255个字符。
VARCHAR(n)	变长字符串。n是指字符长度。	最大为65532个字节。
TEXT[(n)]	变长字符串。n是指该类型的可选字符长度，指定n后将列创建为最小的但足以容纳n字符对应的字节长度的TEXT类型。	最大为65535个字节。
TINYTEXT	变长字符串。	最大为255个字节。
MEDIUMTEXT	变长字符串。	最大为16777215个字节。
LONGTEXT	变长字符串。	最大为1GB-1个字节。

说明

- sql_mode参数值包含“strict_trans_tables”时，非法输入或者输入超过存储空间范围时，会报error。
- sql_mode参数值不包含“strict_trans_tables”时，非法输入或者输入超过存储空间范围时，会报warning信息，并返回截断后的值。
- sql_mode参数值包含"pad_char_to_full_length", CHAR类型作为表列和临时变量的数据类型时，输出带有尾部空格的字符串，否则输出不带尾部空格的字符串。
- 不支持‘||’连接符连接数据。

示例

显示用字符串输出字符类型。

```
--创建表。
m_db=# CREATE TABLE text_type_t1
(
    BT_COL1 CHAR(10),
    BT_COL2 VARCHAR(10),
    BT_COL3 TEXT,
    BT_COL4 TEXT(10)
);

--插入数据。
m_db=# INSERT INTO text_type_t1 VALUES ('text_test1', 'text_test2','text_test3','text_test4');

--查询表中的数据。
m_db=# SELECT *  FROM text_type_t1;
  bt_col1   |  bt_col2   |  bt_col3   |  bt_col4
------------+------------+------------+------------
 text_test1 | text_test2 | text_test3 | text_test4
(1 row)
```



```
--删除表。
m_db=# DROP TABLE text_type_t1;

--示例：CHAR类型。
--创建表。
m_db=# CREATE TABLE char_type_t2
(
    BT_COL1 CHAR(5)
);

--严格模式下插入的数据长度超过类型规定的长度报错。
m_db=# SET SQL_MODE =
'strict_trans_tables,only_full_group_by,no_zero_in_date,no_zero_date,error_for_division_by_zero,
no_auto_create_user,no_engine_substitution';
SET
m_db=# INSERT INTO char_type_t2 VALUES ('too long');
ERROR:  value too long for type character(5)
CONTEXT:  referenced column: BT_COL1

--宽松模式下插入的数据长度超过类型规定的长度提示warning,数据截断插入。
m_db=# SET SQL_MODE = '';
SET
m_db=# INSERT INTO char_type_t2 VALUES ('too long');
WARNING:  value too long for type character(5)
CONTEXT:  referenced column: BT_COL1
INSERT 0 1

--查询表中的数据。
m_db=# SELECT * FROM char_type_t2 ;
 bt_col1
---------
 too l
(1 row)

--删除表。
m_db=# DROP TABLE blob_type_t2;
```

4.6.5 日期时间类型

M-Compatibility支持的日期时间类型请参见**表4-55**。

表 4-55 日期时间类型

名称	描述	存储空间
DATE	用于保存年月日信息，即日期信息。 ● 输入格式：支持有分隔符的字符串'YYYY-MM-DD'和无分隔符的字符串'YYYYMMDD'以及数字YYYYMMDD的输入格式，其中年份可以只输入2位，此时70~99默认为1970~1999，00~69默认为2000~2069。 – 有分隔符场景下，任意标点符号可作为分隔符。 ● 输出格式：仅支持YYYY-MM-DD。 ● 范围：1000-01-01 ~ 9999-12-31。 – sql_mode参数值包含“allow_invalid_dates”时，月1-12与日1-31可以自由组合，即2月31日也可以存入。 – sql_mode参数值不包含“no_zero_in_date”时，允许月和日为0，即1999-00-00为合法值。 – sql_mode参数值不包含“no_zero_date”时，允许年月日均为0，即0000-00-00为合法值。	4个字节。
TIME[(p)]	用于保存时分秒和毫秒信息，即时间信息。 ● 输入格式：支持有分隔符的字符串'hh:mm:ss'和无分隔符的字符串'hhmmss'以及数字hhmmss的输入格式，支持输入日即天数信息的字符串'D hh:mm:ss'，支持毫秒输入，最终会四舍五入截断至指定精度。 – 有分隔符场景下，仅单个英文冒号可作为分隔符。 ● 输出格式：支持hh:mm:ss或hhh:mm:ss。 ● 范围：-838:59:59.000000 ~ 838:59:59.000000。 ● 精度：p表示小数点后的精度，取值范围为0~6。当p设置超过6时，按照p为6进行处理。	8个字节。

名称	描述	存储空间
DATETIME[(p)]	用于保存年月日信息，即日期信息。 ● 输入格式：支持有分隔符的字符串'YYYY-MM-DD'和无分隔符的字符串'YYYYMMDD'以及数字YYYYMMDD的输入格式，其中年份可以只输入2位，此时70~99默认为1970~1999，00~69默认为2000~2069，支持毫秒输入，最终会四舍五入截断至指定精度。 – 有分隔符场景下，在日期部分或时间部分中任意标点符号可作为分隔符，毫秒前的分隔符只能是小数点。 – 年月日与时分秒之间允许若干空格或单个大写字母T作为间隔，其中空格仅应用于有分隔符的字符串。 ● 输出格式：仅支持YYYY-MM-DD hh:mm:ss。 ● 范围：1000-01-01 00:00:00 ~ 9999-12-31 23:59:59。 – sql_mode参数值包含“allow_invalid_dates”时，月1-12与日1-31可以自由组合，即2月31日也可以存入。 – sql_mode参数值不包含“no_zero_in_date”时，允许月和日为0，即1999-00-00 10:00:00为合法值。 – sql_mode参数值不包含“no_zero_date”时，允许年月日均为0，即0000-00-00 10:00:00为合法值。 ● 精度：p表示小数点后的精度，取值范围为0~6。当p设置超过6时，按照p为6进行处理。	8个字节。

名称	描述	存储空间
TIMESTAMP[(p)]	用于保存同时包含日期信息和时间信息的数据，受到数据库时区的影响。 ● 输入格式：支持有分隔符的字符串'YYYY-MM-DD hh:mm:ss'和无分隔符的字符串'YYYYMMDDhhmmss'以及数字YYYYMMDDhhmmss的输入格式，其中年份可以只输入2位时，此时70~99默认为1970~1999，00~69默认为2000~2069，支持毫秒输入，最终会四舍五入截断至指定精度。 – 有分隔符场景下，在日期部分或时间部分中任意标点符号可作为分隔符，毫秒前的分隔符只能是小数点。 – 年月日与时分秒之间允许若干空格或单个大写字母T作为间隔，其中空格仅应用于有分隔符的字符串。 ● 输出格式：仅支持YYYY-MM-DD hh:mm:ss。 ● 范围：1970-01-01 00:00:01' UTC ~ 2038-01-19 03:14:07' UTC。 ● 精度：p表示小数点后的精度，取值范围为0~6。当p设置超过6时，按照p为6进行处理。	8个字节。
YEAR	用于保存年份信息。 ● 输入格式：支持4位的YYYY和至多两位的YY，此时70~99默认为1970~1999，00~69默认为2000~2069。 ● 输出格式：仅支持YYYY。 ● 范围：1901 ~ 2155。	4个字节。

说明

- M-Compatibility不支持ODBC语法的字面量：
  { d 'str' }
  { t 'str' }
  { ts 'str' }
- M-Compatibility支持标准SQL字面量：
  DATE 'str'
  TIME 'str'
  TIMESTAMP 'str'
- 时间类型的数据在显示的时候会自动忽略末尾的所有零。
- 有分隔符的字符串输入中反斜杠“\”在M-Compatibility中视作普通标点符号而非转义字符标志。
- 精度p默认取值为0。
- sql_mode参数值包含“strict_trans_tables”时，非法输入或者超过范围会报错，不包含时非法输入或者超过范围存入0值。

示例:

```
--创建表
m_db=# CREATE TABLE temporal_date(a DATE);
m_db=# CREATE TABLE temporal_time(b TIME);
m_db=# CREATE TABLE temporal_datetime(c DATETIME(5));
m_db=# CREATE TABLE temporal_timestamp(d TIMESTAMP);
m_db=# CREATE TABLE temporal_year(e YEAR);
--插入数据
m_db=# INSERT INTO temporal_date VALUES ('2020-02-02');
m_db=# INSERT INTO temporal_date VALUES (date'2020-02-02');
m_db=# INSERT INTO temporal_date VALUES (20200202);
m_db=# INSERT INTO temporal_time VALUES ('20 10:00:00');
m_db=# INSERT INTO temporal_time VALUES ('800:00:00');
m_db=# INSERT INTO temporal_time VALUES (time'200:00:00');
m_db=# INSERT INTO temporal_datetime VALUES ('2020-02-02T04:04:04');
m_db=# INSERT INTO temporal_datetime VALUES (timestamp'2020-02-02 10:00:00');
m_db=# INSERT INTO temporal_datetime VALUES (20201010010101);
m_db=# INSERT INTO temporal_timestamp VALUES ('2020-02-02 10:00:00');
m_db=# INSERT INTO temporal_timestamp VALUES (20200220101010);
m_db=# INSERT INTO temporal_year VALUES (2020);
m_db=# INSERT INTO temporal_year VALUES (20);
--查看数据
m_db=# SELECT * FROM temporal_date;
     a
------------
 2020-02-02
 2020-02-02
 2020-02-02
(3 rows)

m_db=# SELECT * FROM temporal_time;
     b
-----------
 490:00:00
 800:00:00
 200:00:00
(3 rows)

m_db=# SELECT * FROM temporal_datetime;
            c
----------------------------
 2020-02-02 04:04:04.00000
 2020-02-02 10:00:00.00000
 2020-10-10 01:01:01.00000
(3 rows)

m_db=# SELECT * FROM temporal_timestamp;
          d
---------------------
 2020-02-02 10:00:00
 2020-02-20 10:10:10
(2 rows)

m_db=# SELECT * FROM temporal_year;
  e
------
 2020
 2020
(2 rows)

--删除表
m_db=# DROP TABLE temporal_date;
m_db=# DROP TABLE temporal_time;
m_db=# DROP TABLE temporal_datetime;
m_db=# DROP TABLE temporal_timestamp;
m_db=# DROP TABLE temporal_year;
```


4.6.6 位串类型

位串就是一串由1和0组成的字符串。它们可以用于存储位掩码。M-Compatibility支持的位串类型请参见表4-56。

表 4-56 位串类型

名称	描述	存储空间
BIT[(n)]	位串就是一串由1和0组成的字符串。它们可以用于存储位掩码。 n是指位串长度, 取值范围为[1,64]，如不带精度n，默认精度为1。	最大为64字节。

说明

- 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右边补齐零，直到刚好n位，而不会返回出任何错误。
- 如果用户输出位串值，将会把原有数据的位串值以字符串形式输出。参见示例。
- BIT类型支持位串输入和普通字符串输入。参见示例。
- sql_mode参数值包含“strict_trans_tables”时，非法输入或者超过范围会报错，不包含时非法输入或者超过范围存入当前长度的最大值。
- JDBC&&PBE场景下通过setBytes或setBinaryStream向BIT列插入数据时，如果字节流不是4字节对齐的。M-Compatibility中会因为数据不完整，无法成功插入并提示报错。使用setString方法替代setBytes方法。

示例

显示用字符串输出位串类型。

```
--创建表。
m_db=# CREATE TABLE bit_type_t1
(
    BIT_COL BIT(64)
);

-- 插入数据。
m_db=# INSERT INTO bit_type_t1 VALUES (b'101');

-- 查询表中数据。
m_db=# SELECT *  FROM bit_type_t1;
 bit_col
---------
 101
(1 row)

--删除表。
m_db=# DROP TABLE bit_type_t1;

--创建表。
m_db=# CREATE TABLE bit_type_t2
(
    BIT_COL BIT(3)
);

--严格模式下插入的数据长度超过类型规定的长度报错。
```



```
m_db=# SET SQL_MODE =
'strict_trans_tables,only_full_group_by,no_zero_in_date,no_zero_date,error_for_division_by_zero,
no_auto_create_user,no_engine_substitution';
SET
m_db=# INSERT INTO bit_type_t2 VALUES (b'10101');
ERROR:  bit string too long for type bit varying(3)
CONTEXT:  referenced column: bit_col

--宽松模式下插入的数据长度超过类型规定的长度提示warning,数据取长度内最大值插入。
m_db=# SET SQL_MODE = '';
SET
m_db=# INSERT INTO bit_type_t2 VALUES (b'10101');
WARNING:  bit string too long for type bit varying(3)
CONTEXT:  referenced column: bit_col
INSERT 0 1

--查询表中的数据。
m_db=# SELECT * FROM bit_type_t2;
 bit_col
---------
 111
(1 row)

--删除表。
m_db=# DROP TABLE bit_type_t2;
```

4.6.7 数值类型

M-Compatibility数值类型包括有符号整数类型、无符号整数类型、定点数数值类型、浮点数数值类型。

4.6.7.1 有符号整数类型

M-Compatibility支持的有符号整数类型请参见表4-57。

表 4-57 有符号整数类型

名称	描述	存储空间	范围
TINYINT [SIGNED] [ZEROFILL]	微整数，类型别名为INT1。	1字节。	[-128, 127]。
SMALLINT [SIGNED] [ZEROFILL]	小范围整数，类型别名为INT2。	2字节。	[-32,768, 32,767] 。
MEDIUMINT [SIGNED] [ZEROFILL]	中等范围整数。	4字节。	[-8,388,608, 8,388,607] 。
INT/INTEGER [SIGNED] [ZEROFILL]	常用整数，类型别名为INT4。	4字节。	[-2,147,483,648, 2,147,483,647] 。
BIGINT [SIGNED] [ZEROFILL]	大范围整数，类型别名为INT8。	8字节。	[-2^63, 2^63-1]。

说明

- sql_mode参数值包含“strict_trans_tables”时，有符号整数类型列输入非数值类型的字符，会报error。
- sql_mode参数值不包含“strict_trans_tables”时，有符号整数类型列输入非数值类型的字符，会报warning信息，并返回截断后的值。
- 创建表列时，整数类型后增加SIGNED语法与直接使用有符号整数类型功能含义相同。
- 支持有符号整数类型设置ZEROFILL属性：
  - 设置ZEROFILL属性时，会按照显示宽度在数值前添零。如果没有设置显示宽度，会按照数据类型的默认显示宽度作为具体显示宽度。例如，对于声明为INT(4) ZEROFILL的列，将检索5的值作为0005。
  - 设置ZEROFILL属性时，将自动添加UNSIGNED属性，即使设置为SIGNED属性，也会转成UNSIGNED属性。

示例

```
-- 创建列类型为有符号整数类型的表。
m_db=# CREATE TABLE test_int
(
    a TINYINT,
    b SMALLINT,
    c MEDIUMINT,
    d INTEGER,
    e BIGINT
);

-- 插入数据。
m_db=# INSERT INTO test_int VALUES (100, 200, 300, 400, 500);

-- 查询表中的数据。
m_db=# SELECT * FROM test_int;
  a  |  b  |  c  |  d  |  e
-----+-----+-----+-----+-----
 100 | 200 | 300 | 400 | 500
(1 row)

-- 严格模式下插入非数值类型的字符报错。
m_db=# SET SQL_MODE =
'strict_trans_tables,only_full_group_by,no_zero_in_date,no_zero_date,error_for_division_by_zero,
no_auto_create_user,no_engine_substitution';
SET
m_db=# INSERT INTO test_int VALUES ('1@2@3', '200&20&2', '4%4', '200$30', '1@3');
ERROR:  invalid input syntax for integer: "1@2@3"
LINE 1: INSERT INTO test_int VALUES ('1@2@3', '200&20&2', '4%4', '20...
                                     ^
CONTEXT:  referenced column: a

-- 宽松模式下插入非数值类型的字符提示warning，自动截断。
m_db=# SET SQL_MODE = '';
SET
m_db=# INSERT INTO test_int VALUES ('1@2@3', '200&20&2', '4%4', '200$30', '1@3');
WARNING:  invalid input syntax for integer: "1@2@3"
LINE 1: INSERT INTO test_int VALUES ('1@2@3', '200&20&2', '4%4', '20...
                                     ^
CONTEXT:  referenced column: a
WARNING:  invalid input syntax for integer: "200&20&2"
LINE 1: INSERT INTO test_int VALUES ('1@2@3', '200&20&2', '4%4', '20...
                                           ^
CONTEXT:  referenced column: b
WARNING:  invalid input syntax for integer: "4%4"
LINE 1: INSERT INTO test_int VALUES ('1@2@3', '200&20&2', '4%4', '20...
                                                 ^
CONTEXT:  referenced column: c
WARNING:  invalid input syntax for integer: "200$30"
LINE 1: ...INTO test_int VALUES ('1@2@3', '200&20&2', '4%4', '200$30', ...
```



```
                                                ^
CONTEXT:  referenced column: d
WARNING:  invalid input syntax for integer: "1@3"
LINE 1: ...st_int VALUES ('1@2@3', '200&20&2', '4%4', '200$30', '1@3');
                                                             ^
CONTEXT:  referenced column: e
INSERT 0 1

m_db=# SELECT * FROM test_int;
  a  |  b  |  c  |  d  |  e
-----+-----+-----+-----+-----
 100 | 200 | 300 | 400 | 500
 1   | 200 | 4   | 200 |   1
(2 rows)

-- 设置zerofill属性。
m_db=# DROP TABLE IF EXISTS t1;
DROP TABLE
m_db=# CREATE TABLE t1 (a int zerofill);
CREATE TABLE
m_db=# INSERT INTO t1 values(1);
INSERT 0 1
m_db=# SELECT * FROM t1;
     a
------------
 0000000001
(1 row)

-- 删除表。
m_db=# DROP TABLE test_int;
```

4.6.7.2 无符号整数类型

M-Compatibility支持的无符号整数类型请参见**表4-58**。

表 4-58 无符号整数类型

名称	描述	存储空间	范围
TINYINT UNSIGNED [ZEROFILL]	无符号微整数	1字节。	[0, 255]。
SMALLINT UNSIGNED [ZEROFILL]	无符号小范围整数	2字节。	[0, 65,535] 。
MEDIUMINT UNSIGNED [ZEROFILL]	无符号中等范围整数	4字节。	[0, 16,777,215] 。
INT/INTEGER UNSIGNED [ZEROFILL]	无符号常用整数	4字节。	[0, 4,294,967,295] 。
BIGINT UNSIGNED [ZEROFILL]	无符号大范围整数	8字节。	[0, 18,446,744,073,709,551,615]。

说明

- sql_mode参数值包含“strict_trans_tables”时，无符号整数类型列输入负值，会报error。
- sql_mode参数值包含“strict_trans_tables”时，无符号整数类型列输入非数值类型的字符，会报error。
- sql_mode参数值不包含“strict_trans_tables”时，无符号整数类型列输入负值，会报warning信息，并返回数值0。
- sql_mode参数值不包含“strict_trans_tables”时，无符号整数类型列输入非数值类型的字符，会报warning信息，并返回截断后的值。
- 支持无符号整数类型设置ZEROFILL属性：
  - 设置ZEROFILL属性时，会按照显示宽度在数值前添零。如果没有设置显示宽度，会按照数据类型的默认显示宽度作为具体显示宽度。例如，对于声明为INT(4) UNSIGNED ZEROFILL的列，将检索5的值作为0005。
  - 设置ZEROFILL属性时，将自动添加UNSIGNED属性。

示例

```
-- 创建列类型为无符号整数类型的表。
m_db=# CREATE TABLE test_uint
(
    a TINYINT UNSIGNED,
    b SMALLINT UNSIGNED,
    c MEDIUMINT UNSIGNED,
    d INTEGER UNSIGNED,
    e BIGINT UNSIGNED
);

-- 插入数据。
m_db=# INSERT INTO test_uint VALUES (100, 200, 300, 400, 500);

-- 查询表中的数据。
m_db=# SELECT * FROM test_uint;
  a  |  b  |  c  |  d  |  e
-----+-----+-----+-----+-----
 100 | 200 | 300 | 400 | 500
(1 row)

-- 严格模式下插入负值报错。
m_db=# SET SQL_MODE =
'strict_trans_tables,only_full_group_by,no_zero_in_date,no_zero_date,error_for_division_by_zero,
no_auto_create_user,no_engine_substitution';
SET
m_db=# INSERT INTO test_uint VALUES (-1, -1, -1, -1, -1);
ERROR:  unsigned tinyint out of range
CONTEXT:  referenced column: a

-- 宽松模式下插入负值提示warning，返回数值0。
m_db=# SET SQL_MODE = '';
SET
m_db=# INSERT INTO test_uint VALUES (-1, -1, -1, -1, -1);
WARNING:  unsigned tinyint out of range
CONTEXT:  referenced column: a
WARNING:  unsigned smallint out of range
CONTEXT:  referenced column: b
WARNING:  unsigned mediumint out of range
CONTEXT:  referenced column: c
WARNING:  unsigned int out of range
CONTEXT:  referenced column: d
WARNING:  unsigned bigint out of range
CONTEXT:  referenced column: e
INSERT 0 1
m_db=# select * from test_uint;
 a | b | c | d | e
---+---+---+---+---
```



```
 0 | 0 | 0 | 0 | 0
(1 row)

-- 设置zerofill属性。
m_db=# DROP TABLE IF EXISTS t1;
DROP TABLE
m_db=# CREATE TABLE t1 (a int UNSIGNED zerofill);
CREATE TABLE
m_db=# INSERT INTO t1 VALUES(1);
INSERT 0 1
m_db=# SELECT * FROM t1;
     a
------------
 0000000001
(1 row)

-- 删除表。
m_db=# DROP TABLE test_uint;
```

4.6.7.3 定点数数值类型

定点数数值类型用于精确存储数值，M-Compatibility支持的定点数数值类型请参见**表 定点数数值类型**。

表 4-59 定点数数值类型

名称	描述	存储空间	范围
NUMERIC[(p[,s])] [ZEROFILL]	• 精度p取值范围为[1,65]，标度s取值范围为[0,30]，且标度s不大于精度p。 • 在未指定精度或标度的情况下，默认精度p为10，默认标度s为0。	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。	（十进制位）最大81位数值。
DECIMAL[(p[,s])] [ZEROFILL]	该类型映射为NUMERIC。	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。	（十进制位）最大81位数值。
DEC[(p[,s])] [ZEROFILL]	该类型映射为NUMERIC。	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。	（十进制位）最大81位数值。
FIXED[(p[,s])] [ZEROFILL]	该类型映射为NUMERIC。	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。	（十进制位）最大81位数值。

说明

- 精度p为数值总位数，标度s为数值小数位数。
- 输入超过标度的数值时，会报notice信息，并返回截断后的值。
- sql_mode参数值包含“strict_trans_tables”时，非法输入或者输入超过精度的数值时，会报error。
- sql_mode参数值不包含“strict_trans_tables”时，非法输入或者输入超过精度的数值时，会报warning信息，并返回截断后的值或当前精度最大值。
- 支持定点数数值类型设置ZEROFILL属性：
  - 设置ZEROFILL属性时，会按照精度和标度对数值的整数位和小数位添零。如果没有设置精度、标度，会按照数据类型的默认精度、标度作为具体精度、标度。例如，对于声明为NUMERIC(10,5) ZEROFILL的列，将检索2.1的值作为00002.10000。
  - 设置ZEROFILL属性时，将自动添加UNSIGNED属性。

示例

```
-- 创建列类型为定点数数值类型的表。
m_db=# CREATE TABLE num_tab1
(
    a NUMERIC(10,5),
    b DECIMAL(10,5),
    c DEC(10,5),
    d FIXED(10,5)
);

-- 插入指定精度和标度范围内的数据。
m_db=# INSERT INTO num_tab1 VALUES(1234.56, 1234.56, 1234.56, 1234.56);

-- 插入超过指定标度的数据，提示notice，返回按指定标度截取小数位的数值。
m_db=# INSERT INTO num_tab1 VALUES(12.3456789, 12.3456789, 12.3456789, 12.3456789);
NOTICE:  Truncated incorrect DECIMAL value: '12.3456789'
CONTEXT:  referenced column: a
NOTICE:  Truncated incorrect DECIMAL value: '12.3456789'
CONTEXT:  referenced column: b
NOTICE:  Truncated incorrect DECIMAL value: '12.3456789'
CONTEXT:  referenced column: c
NOTICE:  Truncated incorrect DECIMAL value: '12.3456789'
CONTEXT:  referenced column: d
INSERT 0 1

-- 严格模式下插入超过指定精度的数据，报错。
m_db=# SET SQL_MODE =
'strict_trans_tables,only_full_group_by,no_zero_in_date,no_zero_date,error_for_division_by_zero,
no_auto_create_user,no_engine_substitution';
SET
m_db=# INSERT INTO num_tab1 VALUES(12345678999, 12345678999, 12345678999, 12345678999);
ERROR:  numeric field overflow
DETAIL:  A field with precision 10, scale 5 must round to an absolute value less than 10^5.
CONTEXT:  referenced column: a

-- 宽松模式下插入超过指定精度的数据,提示warning，返回指定精度下最大值。
m_db=# SET SQL_MODE = '';
SET
m_db=# INSERT INTO num_tab1 VALUES(12345678999, 12345678999, 12345678999, 12345678999);
WARNING:  numeric field overflow
DETAIL:  A field with precision 10, scale 5 must round to an absolute value less than 10^5.
CONTEXT:  referenced column: a
WARNING:  numeric field overflow
DETAIL:  A field with precision 10, scale 5 must round to an absolute value less than 10^5.
CONTEXT:  referenced column: b
WARNING:  numeric field overflow
DETAIL:  A field with precision 10, scale 5 must round to an absolute value less than 10^5.
CONTEXT:  referenced column: c
WARNING:  numeric field overflow
```



```
DETAIL:  A field with precision 10, scale 5 must round to an absolute value less than 10^5.
CONTEXT:  referenced column: d
INSERT 0 1

-- 查询表中的数据。
m_db=# SELECT * FROM num_tab1;
     a     |     b     |     c     |     d
-----------+-----------+-----------+-----------
  1234.56000 |  1234.56000 |  1234.56000 |  1234.56000
    12.34568 |    12.34568 |    12.34568 |    12.34568
  99999.99999 | 99999.99999 | 99999.99999 | 99999.99999
(3 rows)

-- 设置zerofill属性。
m_db=# DROP TABLE IF EXISTS t1;
DROP TABLE
m_db=# CREATE TABLE t1 (a numeric zerofill);
CREATE TABLE
m_db=# INSERT INTO t1 values(1);
INSERT 0 1
m_db=# SELECT * FROM t1;
     a
------------
 0000000001
(1 row)

-- 删除表。
m_db=# DROP TABLE num_tab2;
```

4.6.7.4 浮点数数值类型

浮点数数值类型用于存储数值，存储的是近似值。M-Compatibility支持的浮点数数值类型请参见**表 浮点数数值类型**。

表 4-60 浮点数数值类型

名称	描述	存储空间	范围
FLOAT4[(p, s)] [ZEROFILL]	单精度浮点数，不精准。 精度p取值范围为[1,255]，标度s取值范围为[0,min(p, 30)]。	4字节。	-3.402E+38~3.402E+38，6位十进制数字精度。
FLOAT8[(p,s)] [ZEROFILL]	双精度浮点数，不精准。 精度p取值范围为[1,255]，标度s取值范围为[0,min(p, 30)]。	8字节。	-1.79E+308~1.79E+308，15位十进制数字精度。
FLOAT[(p)] [ZEROFILL]	p取值范围为[0,53]。 • 当p取值为[0, 25)时，FLOAT(p)同FLOAT4. • 当p取值为[25, 54)时，FLOAT(p)同FLOAT8。	当p取值为[0, 25)时，4字节，当p取值为[25, 54)时，8字节。	当p取值为[0, 25)时，取值范围同FLOAT4，当p取值为[25, 54)时，取值范围同FLOAT8。

名称	描述	存储空间	范围
REAL[(p, s)] [ZEROFILL]	按DOUBLE[(p, s)]存储，行为同DOUBLE[(p, s)]。 sql_mode设置REAL_AS_FLOAT时，按FLOAT[(p, s)]存储，行为同FLOAT8[(p, s)]。	8字节。 sql_mode设置REAL_AS_FLOAT时，4字节。	-1.79E+308~1.79E+308，15位十进制数字精度。 sql_mode设置REAL_AS_FLOAT时，-3.402E+38~3.402E+38，6位十进制数字精度。
FLOAT[(p, s)] [ZEROFILL]	该类型映射为FLOAT4[(p,s)]。精度p取值范围为[1,255]，标度s取值范围为[0,min(p, 30)]。	4字节。	-3.402E+38~3.402E+38，6位十进制数字精度。
DOUBLE[(p, s)] [ZEROFILL]	该类型映射为FLOAT8[(p,s)]。精度p取值范围为[1,255]，标度s取值范围为[0,min(p, 30)]。	8字节。	-1.79E+308~1.79E+308，15位十进制数字精度。
DOUBLE PRECISION[(p,s)] [ZEROFILL]	该类型映射为FLOAT8[(p,s)]。精度p取值范围为[1,255]，标度s取值范围为[0,min(p, 30)]。	8字节。	-1.79E+308~1.79E+308，15位十进制数字精度。

说明

- 精度p为数值总位数，标度s为数值小数位数。
- 输入超过标度的数值时，会返回截断后的值。
- 未设置精度时，字段的范围如**表4-60**所示的范围，设置了精度时字段的范围以设置的精度为准。
- sql_mode参数值包含“strict_trans_tables”时，非法输入或者输入超过精度的数值时，会报error。
- sql_mode参数值不包含“strict_trans_tables”时，非法输入或者输入超过精度的数值时，会报warning信息，并返回截断后的值或当前精度最大值。
- sql_mode参数包含“real_as_float”时，real类型按float类型存储，行为和float类型一致。
- sql_mode参数不包含“real_as_float”时，real类型按double类型存储，行为和double类型一致。
- 支持浮点数数值类型设置ZEROFILL属性：
  - 设置ZEROFILL属性时，会按照精度和标度对数值的整数位和小数位添零。例如，对于声明为FLOAT(10,5) ZEROFILL的列，将检索2.1的值作为0002.10000。
  - 设置ZEROFILL属性时，将自动添加UNSIGNED属性。

示例

```
-- 创建列类型为浮点数数值类型的表。
m_db=# CREATE TABLE float_tab1
```

```
(
    a FLOAT(6,2),
    b DOUBLE(6,2)
);

-- 插入指定精度和标度范围内的数据。
m_db=# INSERT INTO float_tab1 VALUES(1234.56, 1234.56);

-- 插入超过指定标度的数据，返回按指定标度截取小数位的数值。
m_db=# INSERT INTO float_tab1 VALUES(12.3456789, 12.3456789);

-- 严格模式下插入超过指定精度的数据，报错。
m_db=# SET SQL_MODE =
'strict_trans_tables,only_full_group_by,no_zero_in_date,no_zero_date,error_for_division_by_zero,
no_auto_create_user,no_engine_substitution';
SET
m_db=# INSERT INTO float_tab1 VALUES(12345678999, 12345678999);
ERROR:  float field overflow
DETAIL:  A field with precision 6, scale 2 must round to an absolute value less than 9.9999900e+03.
CONTEXT:  referenced column: a


-- 宽松模式下插入超过指定精度的数据,提示warning，返回指定精度下最大值。
m_db=# SET SQL_MODE = '';
SET
m_db=# INSERT INTO float_tab1 VALUES(12345678999, 12345678999);
WARNING:  float field overflow
DETAIL:  A field with precision 6, scale 2 must round to an absolute value less than 9.9999900e+03.
CONTEXT:  referenced column: a
WARNING:  float field overflow
DETAIL:  A field with precision 6, scale 2 must round to an absolute value less than 9.9999899999999998e
+03.
CONTEXT:  referenced column: b
INSERT 0 1

-- 查询表中的数据。
m_db=# SELECT * FROM float_tab1;
    a    |    b
---------+---------
 1234.56 | 1234.56
   12.35 |   12.35
 9999.99 | 9999.99
(3 rows)

-- 删除表。
m_db=# DROP TABLE float_tab1;

-- SQL_MODE没设置REAL_AS_FLOAT时创建列类型为real类型的表。
m_db=# SET SQL_MODE='';
SET
m_db=# CREATE TABLE real_tab1(a REAL);

-- 插入数据。
m_db=# INSERT INTO real_tab1 VALUES(12345.678999999999);

-- 查询表中的数据，结果同DOUBLE类型，精确显示12位。
m_db=# SELECT * FROM real_tab1;
    a
---------
 12345.678999999998
 (1 row)

-- 删除表。
m_db=# DROP TABLE real_tab1;

-- SQL_MODE设置REAL_AS_FLOAT时创建列类型为real类型的表。
m_db=# SET SQL_MODE='REAL_AS_FLOAT';
SET
m_db=# CREATE TABLE real_tab2(a REAL);
```



```
-- 插入数据。
m_db=# INSERT INTO real_tab2 VALUES(12345.678999999999);

-- 查询表中的数据，结果同FLOAT类型，精确显示6位。
m_db=# SELECT * FROM real_tab2;
    a
---------
 12345.7
 (1 row)

-- 设置zerofill属性。
m_db=# DROP TABLE IF EXISTS t1;
DROP TABLE
m_db=# CREATE TABLE t1 (a float zerofill);
CREATE TABLE
m_db=# INSERT INTO t1 VALUES(1);
INSERT 0 1
m_db=# SELECT * FROM t1;
      a
--------------
 000000000001
(1 row)

-- 删除表。
m_db=# DROP TABLE real_tab2;
```

4.6.8 JSON 类型

M-Compatibility支持的JSON类型请参见**表4-61**。

表 4-61 JSON 类型

名称	描述	存储空间
JSON	非结构化的数据类型，用于存储和操作JSON数据。 ● 输入格式： 标量： – 空类型：null，全小写。 – bool类型：仅true和false，全小写。 – 数字类型：正整数、负整数、小数和0，支持科学计数法。不支持多余的前导0、正数前的+号、NaN和inf。 – 字符串类型：必须加双引号。 数组：[]结构，存放的元素可以是任意类型的JSON，不要求数组内所有元素类型一致。 对象：{}结构，存储{key:value}的键值对，key只能是带有双引号的字符串，value可以是任意类型的JSON。对于重复的key，以第一个键值对{key:value}为准。 ● 输出格式： 文本解析成JSON类型后，会忽略无关语义的细节，如空格。 对象类型的格式归一化： – 删除key重复的键值对{key:value}，只保留第一个出现的键值对{key:value}。 – 键值对{key:value}会重新进行排序，排序规则：key长度长的位于较后位置，key长度相等则ASCII码较大的位于较后位置。 ● 取值范围： – 数字类型取值范围与FLOAT8类型相同：最小值-1.797693e+308，最大值1.797693e+308，超出取值范围则报错。 – 数组和对象类型的最大嵌套深度为100。	最大为1073741817个字节。

说明

- JSON数据类型的输入合法校验不受GUC参数sql_mode影响。sql_mode为任意取值时，JSON类型列输入不合法的字符时，都会产生ERROR。
- JSON数据类型不支持作为主键和索引键以及作为分区键。
- 当插入JSON类型列的整数大于无符号整数类型最大值2^64 - 1或小于有符号整数类型最小值-2^63时，会作为DOUBLE类型存储，对精度会产生部分差异。
- 为了与GaussDB非M-Compatibility模式数据库（即GaussDB原有）的JSON类型区分，M-Compatibility模式数据库内新增GUC参数m_format_behavior_compat_options的兼容性选项cast_as_new_json。
  - 该选项开启时，执行::JSON、CREATE TABLE <tablename> AS <包含JSON类型的SELECT查询>; 等涉及通过::JSON方式执行JSON类型转换的场景，实际转换为M-Compatibility模式数据库的JSON类型。
  - 如果不开启该选项，相关JSON类型转换的场景实际转换为非M-Compatibility模式数据库（即GaussDB原有）的JSON类型。
  - 该选项在数据库实例初始化后默认开启，但如果需要显式指定m_format_behavior_compat_options兼容性参数的值并使用M-Compatibility模式数据库内的JSON类型，为保证功能符合预期，需要补充设置cast_as_new_json参数。如：
    ```
    SET m_format_behavior_compat_options = 'xxx,xxx,cast_as_new_json';
    ```
- 为了与原有的JSON类型区分，M-Compatibility中的JSON类型在pg_type系统表中的typname名称为“jsonm”。
  - 在SQL语法中，JSON相当于JSONM的别名，但JSONM不可作为数据类型名称使用。
  - “jsonm”字样可能会在错误信息等文本中出现，用户可将其看作JSON类型。
  - pg_type系统表中typname名称为“json”和“jsonb”的数据类型为原有的JSON类型，在M-Compatibility中已废弃，请勿使用相关的内部函数，例如：“json_in”、“jsonb_in”等。

示例

```
-- 创建列类型为JSON数值类型的表。
m_db=# CREATE TABLE test_json (id INT, json_value JSON);
CREATE TABLE
-- 插入空类型标量
m_db=# INSERT INTO test_json VALUES(1, 'null');
INSERT 0 1
-- 插入数字类型标量
m_db=# INSERT INTO test_json VALUES(2, '-1.5e+2');
INSERT 0 1
-- 插入bool类型标量
m_db=# INSERT INTO test_json VALUES(3, 'true');
INSERT 0 1
m_db=# INSERT INTO test_json VALUES(4, 'false');
INSERT 0 1
-- 插入字符串类型标量
m_db=# INSERT INTO test_json VALUES(5, '"abc"');
INSERT 0 1
-- 插入数组
m_db=# INSERT INTO test_json VALUES(6, '[1, 2, "json", null, [[]], {}]');
INSERT 0 1
-- 插入对象
m_db=# INSERT INTO test_json VALUES(7, '{"jsnid": [true, "abc"], "tag": {"ab": 1, "b": null, "a": 2}}');
INSERT 0 1
-- 查询表中的数据
m_db=# SELECT * FROM test_json;
 id |                          json_value
----+--------------------------------------------------------------
  1 | null
  2 | -150
  3 | true
  4 | false
  5 | "abc"
```



```
  6 | [1, 2, "json", null, [[]], {}]
  7 | {"tag": {"a": 2, "b": null, "ab": 1}, "jsnid": [true, "abc"]}
(7 rows)

-- 非法输入报错
m_db=# INSERT INTO test_json VALUES(1, '+20');
ERROR:  invalid input syntax for type json
LINE 1: INSERT INTO test_json VALUES(1, '+20');
                                    ^
DETAIL:  Token "+" is invalid.
CONTEXT:  JSON data, line 1: +...
referenced column: json_value
m_db=# INSERT INTO test_json VALUES(1, 'NaN');
ERROR:  invalid input syntax for type json
LINE 1: INSERT INTO test_json VALUES(1, 'NaN');
                                    ^
DETAIL:  Token "NaN" is invalid.
CONTEXT:  JSON data, line 1: NaN
referenced column: json_value
m_db=# INSERT INTO test_json VALUES(1, 'inf');
ERROR:  invalid input syntax for type json
LINE 1: INSERT INTO test_json VALUES(1, 'inf');
                                    ^
DETAIL:  Token "inf" is invalid.
CONTEXT:  JSON data, line 1: inf
referenced column: json_value
m_db=# INSERT INTO test_json VALUES(1, 'NULL');
ERROR:  invalid input syntax for type json
LINE 1: INSERT INTO test_json VALUES(1, 'NULL');
                                    ^
DETAIL:  Token "NULL" is invalid.
CONTEXT:  JSON data, line 1: NULL
referenced column: json_value
m_db=# INSERT INTO test_json VALUES(1, 'TRUE');
ERROR:  invalid input syntax for type json
LINE 1: INSERT INTO test_json VALUES(1, 'TRUE');
                                    ^
DETAIL:  Token "TRUE" is invalid.
CONTEXT:  JSON data, line 1: TRUE
referenced column: json_value
m_db=# INSERT INTO test_json VALUES(1, '000123');
ERROR:  invalid input syntax for type json
LINE 1: INSERT INTO test_json VALUES(1, '000123');
                                    ^
DETAIL:  Token "000123" is invalid.
CONTEXT:  JSON data, line 1: 000123
referenced column: json_value
m_db=# INSERT INTO test_json VALUES(1, 'abc');
ERROR:  invalid input syntax for type json
LINE 1: INSERT INTO test_json VALUES(1, 'abc');
                                    ^
DETAIL:  Token "abc" is invalid.
CONTEXT:  JSON data, line 1: abc
referenced column: json_value
m_db=# INSERT INTO test_json VALUES(1, '{12:"abc"}');
ERROR:  invalid input syntax for type json
LINE 1: INSERT INTO test_json VALUES(1, '{12:"abc"}');
                                    ^
DETAIL:  Expected string or "}", but found "12".
CONTEXT:  JSON data, line 1: {12...
referenced column: json_value

-- 删除表
m_db=# DROP TABLE test_json;
```


4.6.9 枚举类型

ENUM是一个字符串类型，该类型只在建表时可以创建，创建的ENUM类型会提供一个列表包含所有可选的枚举值。ENUM的合法值为列表中的一个值，该值可以由枚举值的字符串形式或者枚举的下标表示。

ENUM创建语法如下：

```
ENUM('val1','val2',...) [CHARACTER SET charset_name] [COLLATE collation_name]
```

ENUM类型的枚举值的字符串值和整型值一一对应，在操作时含义相同。以ENUM('beijing', 'shanghai', 'nanjing', 'wuhan') 为例，枚举值的字符串和整型值对应关系如表4-62所示。

表 4-62 枚举类型成员字符串值和整型值（下标）对应关系

字符串值	整型下标
NULL	NULL
''	0
'beijing'	1
'shanghai'	2
'nanjing'	3
'wuhan'	4

说明

1. NULL值的下标为NULL。
2. 宽松模式下或者INSERT IGNORE语法插入非法值时，会插入0值，即空串。当有枚举成员为空串时，空串为合法值，且整型值为空串的位置下标，非法值的字符串值也为空串，但整型值为0。

M-Compatibility支持的ENUM类型请如表4-63所示。

表 4-63 枚举类型

名称	描述	存储空间
ENUM	枚举类型，用于存储和操作可选的枚举值。 ● 输入格式: – 字符串，可选的枚举成员之一。 – 整型，枚举成员对应的位置下标之一。 ● 输出格式: – 枚举创建时的枚举成员字符串值。 ● 取值范围: – 最多支持创建65535个枚举成员。 – 枚举成员所代表的字符串值或整型值之一。	6字节

说明

- ENUM数据类型的输入合法校验受sql_mode影响。宽松模式下，ENUM类型列输入不合法的值，会插入0值，即空串，严格模式下，插入不合法的值，会报错。
- ENUM数据类型创建时的每个成员值需要唯一。出现重复成员值，严格模式下报错，宽松模式下支持创建，但默认使用重复值中第一个出现的。
- ENUM数据类型支持字符集与字符序特性，且校验成员值是否重复与ENUM列上的字符序特性有关。
- ENUM数据类型不支持作为分区键。
- ENUM数据类型上的索引仅支持通过整型查找，使用字符串值无法走索引。
- 不建议ENUM成员值为数值，容易与下标值出现混淆。
- 当前ENUM类型与其它类型比较不支持走索引扫描，需要在ENUM类型的列上，根据需要查询的数据类型，创建函数表达式索引来支持走索引扫描。

示例:

```
--创建表和ENUM数据类型
m_db=# CREATE TABLE test_enum(c1 ENUM('abc','efg','中文','汉字'));
CREATE TABLE

-- 查看表结构
m_db=# select pg_get_tabledef('test_enum');
                                  pg_get_tabledef
-------------------------------------------------------------------------------------------
 SET search_path = public;                                                                +
 CREATE TABLE test_enum (                                                                 +
    c1 ENUM('abc', 'efg', '中文', '汉字') CHARACTER SET `UTF8` COLLATE utf8mb4_general_ci+
 )                                                                                        +
 CHARACTER SET = "UTF8" COLLATE = "utf8mb4_general_ci"                                    +
 WITH (orientation=row, compression=no, storage_type=USTORE, segment=off);
(1 row)

-- 插入字符串数据
m_db=# insert into test_enum values('abc');
```



```
INSERT 0 1
m_db=# insert into test_enum values('中文');
INSERT 0 1

-- 插入整型数据
m_db=# insert into test_enum values(2),(4);
INSERT 0 2

-- 字符串形式查询数据
m_db=# select * from test_enum where c1 = 'efg' or c1 = '中文';
  c1
------
 中文
 efg
(2 rows)

-- 整型值形式查询数据
m_db=# select * from test_enum where c1 = 1 or c1 = 4;
  c1
------
 abc
 汉字
(2 rows)

--删除表
m_db=# drop table test_enum;
DROP TABLE

--创建表和ENUM数据类型
m_db=# create table t1(id int, c1 enum('a','bb','ccc'));
CREATE TABLE

-- 直接创建ENUM类型的索引无法支持走索引扫描
m_db=# create index idx_1 on t1(c1);
CREATE INDEX

m_db=# explain select c1 from t1 where c1 = 'a';
                    QUERY PLAN
-----------------------------------------------------
 Seq Scan on t1  (cost=0.00..59.00 rows=15 width=32)
   Filter: (cast_to_cstring(c1) = 'a'::text)
(2 rows)

-- 字符串查询，创建cast_to_cstring 函数索引，支持索引扫描
m_db=# create index id_t1 on t1((cast_to_cstring(c1)));
CREATE INDEX

m_db=# explain select c1 from t1 where c1 = 'a';
                            QUERY PLAN
-------------------------------------------------------------------
 Bitmap Heap Scan on t1  (cost=4.30..13.78 rows=6 width=32)
   Recheck Cond: (cast_to_cstring(c1) = 'a'::text)
   ->  Bitmap Index Scan on id_t1  (cost=0.00..4.30 rows=6 width=0)
         Index Cond: (cast_to_cstring(c1) = 'a'::text)
(4 rows)

-- 整型查询，创建cast_to_int8 函数索引，支持索引扫描
m_db=# create index id_t2 on t1((cast_to_int8(c1)));
CREATE INDEX

m_db=# explain select c1 from t1 where c1 = 5;
                            QUERY PLAN
-------------------------------------------------------------------
 Bitmap Heap Scan on t1  (cost=4.30..13.78 rows=6 width=32)
   Recheck Cond: (cast_to_int8(c1) = 5)
   ->  Bitmap Index Scan on id_t2  (cost=0.00..4.30 rows=6 width=0)
         Index Cond: (cast_to_int8(c1) = 5)
(4 rows)
```


4.6.10 集合类型

SET数据类型是一种常量字符串类型数据，该类型只在建表时可以创建，适用于存储确定集合的子集的场景，这些字符串常量必须是SET定义时指定的常量字符串集合的子集。

SET创建语法如下：

```
SET('val1','val2',...) [CHARACTER SET charset_name] [COLLATE collation_name]
```

SET类型底层存储是以bitmap的形式存储，每个集合成员对应于一个比特位。每个集合的子集有字符串形式和整型两种表示方法，在操作时含义相同，以SET('beijing', 'shanghai', 'nanjing', 'wuhan') 为例，集合的字符串值和整型值对应的关系如**表4-64**所示。

表 4-64 集合子集字符串值和整型值对应关系

集合字符串值	集合对应二进制值	整型值
'beijing'	0001	1
'shanghai'	0010	2
'nanjing'	0100	4
'wuhan'	1000	8
'beijing,nanjing'	0101	5

M-Compatibility支持的SET类型如**表4-65**所示。

表 4-65 集合类型

名称	描述	存储空间
SET	集合类型，用于存储确定集合的子集的场景。 ● 输入格式: – 字符串，子集合之一，成员间以','分隔。 – 整型，子集合对应的整型值。 ● 输出格式: – 子集合的字符串，成员间以','分隔。 ● 取值范围: – 最多支持创建64个成员。 – 创建时给出集合的子集之一。	4字节~8字节

说明

- SET数据类型的输入合法校验受sql_mode影响。宽松模式下，SET类型列输入不合法的值，会插入0值，即空串，严格模式下，插入不合法的值，会报错。
- SET数据类型创建时的每个成员值需要唯一。出现重复成员值，严格模式下报错，宽松模式下支持创建，但默认使用重复值中第一个出现的。
- SET数据类型支持字符集与字符序特性，且校验成员值是否重复与SET列上的字符序特性有关。
- SET数据类型不支持作为分区键。
- SET数据类型上的索引仅支持通过整型查找，使用字符串值无法走索引。
- 当前SET类型与其它类型比较不支持走索引扫描，需要在SET类型的列上，根据需要查询的数据类型，创建函数表达式索引来支持走索引扫描。

示例:

```
-- 创建表和SET数据类型
m_db=# CREATE TABLE test_set(c1 SET('abc','efg','中文','汉字'));
CREATE TABLE

-- 查看表结构
m_db=# SELECT pg_get_tabledef('test_set');
                                pg_get_tabledef
-------------------------------------------------------------------------------------------
 SET search_path = public;                                                                +
 CREATE TABLE test_set (                                                                  +
     c1 SET('abc', 'efg', '中文', '汉字') CHARACTER SET `UTF8` COLLATE utf8mb4_general_ci+
 )                                                                                        +
 CHARACTER SET = "UTF8" COLLATE = "utf8mb4_general_ci"                                    +
 WITH (orientation=row, compression=no, storage_type=USTORE, segment=off);
(1 row)

-- 插入字符串数据
m_db=# INSERT INTO test_set VALUES('abc,efg'),('efg,汉字,中文'), ('abc');
INSERT 0 3

-- 插入整型数据
m_db=# INSERT INTO test_set VALUES(5),(12);
INSERT 0 2

-- 查询数据
m_db=# SELECT * FROM test_set;
      c1
---------------
 abc,efg
 efg,中文,汉字
 abc
 abc,中文
 中文,汉字
(5 rows)

-- 字符串形式查询数据
m_db=# SELECT * FROM test_set WHERE c1 = 'abc,efg';
   c1
---------
 abc,efg
(1 row)

-- 整型值形式查询数据
m_db=# SELECT * FROM test_set WHERE c1 = 12;
    c1
-----------
 中文,汉字
(1 row)

-- 删除表
```



```
m_db=# DROP TABLE test_set;
DROP TABLE

-- 创建表和SET数据类型
m_db=# CREATE TABLE t1(id int, c1 set('a','bb','ccc'));
CREATE TABLE

-- 直接创建SET类型的索引无法支持走索引扫描
m_db=# CREATE INDEX idx_1 ON t1(c1);
CREATE INDEX

m_db=# EXPLAIN SELECT c1 FROM t1 WHERE c1 = 'a';
                        QUERY PLAN
-----------------------------------------------------------
 Seq Scan on t1  (cost=0.00..59.00 rows=15 width=32)
   Filter: (cast_to_cstring(c1) = 'a'::text)
(2 rows)

-- 字符串查询，创建cast_to_cstring 函数索引，支持索引扫描
m_db=# CREATE INDEX id_t1 ON t1((cast_to_cstring(c1)));
CREATE INDEX

m_db=# EXPLAIN SELECT c1 FROM t1 WHERE c1 = 'a';
                              QUERY PLAN
----------------------------------------------------------------------
 Bitmap Heap Scan on t1  (cost=4.30..13.78 rows=6 width=32)
   Recheck Cond: (cast_to_cstring(c1) = 'a'::text)
   ->  Bitmap Index Scan on id_t1  (cost=0.00..4.30 rows=6 width=0)
         Index Cond: (cast_to_cstring(c1) = 'a'::text)
(4 rows)

-- 整型查询，创建cast_to_int8函数索引，支持索引扫描
m_db=# CREATE INDEX id_t2 ON t1((cast_to_int8(c1)));
CREATE INDEX

m_db=#  EXPLAIN SELECT c1 FROM t1 WHERE c1 = 5;
                              QUERY PLAN
----------------------------------------------------------------------
 Bitmap Heap Scan on t1  (cost=4.30..13.78 rows=6 width=32)
   Recheck Cond: (cast_to_int8(c1) = 5)
   ->  Bitmap Index Scan on id_t2  (cost=0.00..4.30 rows=6 width=0)
         Index Cond: (cast_to_int8(c1) = 5)
(4 rows)
```

4.7 类型转换

背景信息

在SQL中，每个数据都与一个决定其行为和用法的数据类型相关。在M-Compatibility里，有四种基本的SQL结构需要独立的类型转换规则。

- 函数调用

  多数SQL类型系统是构筑在一套丰富的函数上的。函数调用可以有一个或多个参数。因为SQL允许函数重载，所以不能通过函数名直接找到要调用的函数，分析器必须根据函数提供的参数类型选择正确的函数。

- 操作符

  SQL允许在表达式上使用前缀或后缀（一元）操作符，也允许表达式内部使用二元操作符（两个参数）。像函数一样，操作符也可以被重载，因此操作符的选择也和函数一样取决于参数类型。

- 值存储

INSERT和UPDATE语句将表达式结果存入表中。语句中的表达式类型必须和目标字段的类型一致或者可以转换为一致。

- UNION，EXCEPT，CASE和相关构造

  因为联合SELECT语句中的所有查询结果必须在一列里显示出来，所以每个SELECT子句中的元素类型必须相互匹配并转换成一个统一类型。类似地，一个CASE构造的结果表达式必须转换成统一的类型，这样整个CASE表达式会有一个统一的输出类型。

四种SQL结构在M-Compatibility类型转换规则下分为三种场景：隐式类型转换；显式类型转换；UNION，EXCEPT，CASE和相关构造。M-Compatibility中针对三种场景有不同的转换行为与转换关系。

语义分析阶段会决定表达式的返回值类型并选择适当的转换行为。M-Compatibility数据类型的基本类型分类包括：INTEGER、DECIMAL、STRING、BITSTRING、FLOAT和TEMPORAL。为了解决表达式的类型选择问题，M-Compatibility中每种基本类型之间都存在隐式转换。根据类型与可用的隐式转换，就可能保证有歧义的表达式得到有效的解决方式。

4.7.1 隐式类型转换

隐式类型转换主要适用于函数调用、操作符和值存储场景。下面主要介绍值存储场景的类型解析。其他场景请见**函数和操作符**章节。

值存储数据类型解析

1. 查找与目标字段准确的匹配。
2. 尝试将表达式直接转换成目标类型。如果已知这两种类型之间存在一个已注册的转换函数，那么直接调用该转换函数即可。如果表达式是一个未知类型文本，该文本字符串的内容将交给目标类型的输入转换过程。
3. 检查目标类型是否有长度转换。长度转换是一个从某类型到自身的转换。如果目标类型存在长度转换逻辑，那么在存储到目标字段之前先在表达式上应用。这样的转换函数总是接受一个额外的类型为integer的参数，它接收目标字段的atttypmod值（实际上是其声明长度，atttypmod的解释随不同的数据类型而不同），并且它可能会接受一个BOOL类型的第三个参数，表示转换是显式的还是隐式的。转换函数负责赋予那些长度相关的语义，比如长度检查或者截断。

示例：

CHAR存储类型转换。对一个目标列定义为CHAR(20)的语句，下面的语句显示存储值的长度正确：

```
m_db=# CREATE TABLE char_storage (
    VS_COL1 CHAR(20)
);
m_db=# INSERT INTO char_storage VALUES('abcdef');
m_db=# SELECT VS_COL1, length(VS_COL1) FROM char_storage;
 VS_COL1 | length
---------+--------
 abcdef  |      6
(1 row)

m_db=# DROP TABLE char_storage;
```

说明

这里输入的字段被解析为UNKNOWN，UNKNOWN类型会根据目标列在系统表中找到对应CHAR类型的输入函数。最后，在系统表里找到长度转换函数bpchar(bpchar, integer, bool) 并且应用于结果和存储的字段长。这个类型相关的函数执行所需的长度检查和额外的空白填充。

FLOAT存储类型向BINARY存储类型转换, 对目标列为BINARY(5)的表插入从FLOAT列取出的值：

```
m_db=# CREATE TABLE varchar_storage (
    VS_COL1 BINARY(5)
);
m_db=# CREATE TABLE float_storage (
    VS_COL1 FLOAT
);
m_db=# INSERT INTO float_storage VALUES(1234567);
m_db=# INSERT INTO varchar_storage SELECT * FROM float_storage ;
INSERT 0 1
m_db=# SELECT * FROM varchar_storage;
 VS_COL1
---------
 1.2e6
(1 row)
m_db=# DROP TABLE varchar_storage, float_storage;
```

说明

从FLOAT列向BINARY列进行数据插入时，会搜索系统表中FLOAT到BINARY类型的隐式转换规则，调用对应的类型转换函数。最后，在系统表里找到长度转换函数binary(bpchar, integer, bool)并且应用于结果和存储的字段长。 这个类型相关的函数执行所需的长度检查和额外的空白填充。

4.7.2 显式类型转换

显式类型转换主要适用于类型转换函数调用场景，类型转换函数请参考**类型转换函数**。

显式类型转换语法为：

```
SELECT CAST(expression AS data_type);
```

其中expression是要转换的表达式及data_type要转换成的数据类型如**表4-66**和**表4-67**所示。

表 4-66 显式 cast 转换

目标类型	用于承接输入值的数据类型（返参类型）
SIGNED [INTEGER]	BIGINT
UNSIGNED [INTEGER]	BIGINT UNSIGNED
DATE	DATE
TIME	TIME
DATETIME	DATETIME
BINARY	TEXT
CHAR	TEXT
DECIMAL	NUMERIC
FLOAT	FLOAT4
DOUBLE	FLOAT8

目标类型	用于承接输入值的数据类型（返参类型）
JSON	JSON

示例：

```
-- signed显式cast用法。
m_db=# SELECT CAST(9223372036854775808 AS SIGNED);
         cast
----------------------
 -9223372036854775808
(1 row)

-- 边界值处理。
m_db=# SELECT CAST(9223372036854775808.0 AS SIGNED);
WARNING:  Truncated incorrect DECIMAL value: '9223372036854775808.0'
CONTEXT:  referenced column: cast
        cast
---------------------
 9223372036854775807
(1 row)

-- char显式cast用法（修饰符限定边界）。
m_db=# SELECT CAST(123 AS CHAR(3));
 cast
------
 123
(1 row)
```

表 4-67 JSON 类型显式转换行为

参数类型	CAST(expression AS JSON)
TEXT、TINYTEXT、MEDIUMTEXT、LONGTEXT、VARCHAR、CHAR、字符串常量	作为输入文本解析成JSON格式，返回相应的JSON类型（包括NULL、BOOLEAN、INTEGER，UNSIGNED INTEGER、DOUBLE、STRING、ARRAY、OBJECT类型），如果不符合JSON语法，会产生ERROR。
INT、TINYINT、SMALLINT、MEDIUMINT、BIGINT	转换为INTEGER类型的JSON数字标量。
INT UNSIGNED、TINYINT UNSIGNED、SMALLINT UNSIGNED、MEDIUMINT UNSIGNED、BIGINT UNSIGNED	转换为UNSIGNED INTEGER类型的JSON数字标量。
FLOAT4、FLOAT8	转换为DOUBLE类型的JSON数字标量。
NUMERIC	转换为DECIMAL类型的JSON数字标量。
BLOB、TINYBLOB、MEDIUMBLOB、LONGBLOB、BINARY、VARBINARY	转换为BLOB类型的JSON字符串标量。
BIT	转换为BIT类型的JSON字符串标量。

参数类型	CAST(expression AS JSON)
YEAR	转换为OPAQUE类型的JSON字符串标量。
TIME	转换为TIME类型的JSON字符串标量。
DATE	转换为DATE类型的JSON字符串标量。
TIMESTAMP	转换为TIMESTAMP类型的JSON字符串标量。
DATETIME	转换为DATETIME类型的JSON字符串标量。
JSON	返回值为输入的JSON参数。
SET、ENUM	转换为STRING类型的JSON字符串标量。

说明

- 显式类型转换还可以使用双冒号语法，例如“SELECT 1::INT”；此方法不建议用户使用（可能导致结果不符合预期）。
- 设置兼容性参数m_format_dev_version='s2'后双冒号显式转换行为变更为默认隐式转换逻辑且支持M-Compatibility数据库中新增数据类型（除SET、ENUM类型，该两种类型不支持双冒号类型转换方法）；不设置兼容性参数情况下支持GaussDB原有的双冒号显式转换逻辑，M-Compatibility数据库中新增数据类型暂不支持。

4.7.3 UNION，EXCEPT, CASE 和相关构造

SQL UNION/CASE构造必须把那些可能不太相似的类型匹配起来成为一个结果集。解析算法分别应用于联合查询的每个输出字段。EXCEPT构造对不相同的类型使用和UNION相同的算法进行解析。

UNION/EXCEPT/CASE将两种类型的表达式转换为统一的目标类型，并通过隐式类型转换将数据转换为目标类型，实现聚合。

UNION 和相关构造解析

UNION分为UNION和UNION ALL。UNION将数据聚合并进行去重处理，UNION ALL只聚合数据，不做去重处理。

示例：

```
-- 创建表
m_db=# CREATE TABLE test_union1 (union_col1 INT);
m_db=# CREATE TABLE test_union2 (union_col2 CHAR(5));

-- 插入数据
m_db=# INSERT INTO test_union1 VALUES (123);
m_db=# INSERT INTO test_union2 VALUES ('123');

-- union用法
m_db=# SELECT union_col1 FROM test_union1 UNION SELECT union_col2 FROM test_union2;
 union_col1
------------
```



```
 123
(1 row)

-- union all用法
m_db=# SELECT union_col1 FROM test_union1 UNION ALL SELECT union_col2 FROM test_union2;
 union_col1
------------
 123
 123
(2 rows)

-- 非列值UNION应用
m_db=# SELECT '234' FROM test_union1 UNION SELECT '234';
 ?column?
----------
 234
(1 row)
```

EXCEPT 和相关构造解析

EXCEPT分为EXCEPT、EXCEPT ALL和EXCEPT DISTINCT。EXCEPT和EXCEPT DISTINCT会在第一个查询结果中去除两个查询结果的交集并对结果进行去重处理，EXCEPT ALL不做去重处理。

示例：

```
-- 创建表
m_db=# CREATE TABLE data_types (
  col_int1 TINYINT,
  col_int2 SMALLINT,
  col_int4 INT,
  col_int8 BIGINT,
  col_uint1 TINYINT UNSIGNED,
  col_uint2 SMALLINT UNSIGNED,
  col_uint4 INT UNSIGNED,
  col_uint8 BIGINT UNSIGNED,
  col_float FLOAT
);

-- 插入数据
INSERT INTO data_types
VALUES
  (
   1,
   2,
   3,
   4,
   1,
   2,
   3,
   4,
   1.1
);

-- except用法
m_db=# SELECT col_int4 FROM data_types EXCEPT SELECT col_float FROM data_types;
 col_int4
----------
        3
(1 row)

m_db=# SELECT PG_TYPEOF(col_int4) FROM (SELECT col_int4 FROM data_types EXCEPT SELECT col_float
FROM data_types);
 pg_typeof
-----------
 double
(1 row)
```



```
-- except all用法
m_db=# SELECT col_int4 FROM data_types EXCEPT ALL SELECT col_float FROM data_types;
 col_int4
----------
        3
(1 row)

m_db=# SELECT PG_TYPEOF(col_int4) FROM (SELECT col_int4 FROM data_types EXCEPT ALL SELECT
col_float FROM data_types);
 pg_typeof
-----------
 double
(1 row)

-- except distinct用法
m_db=# SELECT col_int4 FROM data_types EXCEPT DISTINCT SELECT col_float FROM data_types;
 col_int4
----------
        3
(1 row)

m_db=# SELECT PG_TYPEOF(col_int4) FROM (SELECT col_int4 FROM data_types EXCEPT DISTINCT SELECT
col_float FROM data_types);
 pg_typeof
-----------
 double
(1 row)

-- 非列值except应用
m_db=# SELECT col_int4 FROM data_types EXCEPT SELECT '4';
 col_int4
----------
 3
(1 row)

m_db=# SELECT PG_TYPEOF(col_int4) FROM (SELECT col_int4 FROM data_types EXCEPT SELECT '4');
 pg_typeof
-----------
 varchar
(1 row)

-- 删除表
m_db=# DROP TABLE data_types;
```

CASE 和相关构造解析

CASE语法一共有两种表达方式，两种方式功能含义相同：

```
CASE WHEN col = A THEN ... ELSE ...
CASE col WHEN A THEN ... ELSE ...
```

示例：

```
-- 创建表
m_db=# CREATE TABLE test_case1 (case_col1 INT);

-- 插入数据
m_db=# INSERT INTO test_case1 VALUES (1), (23), (45), (1);

-- case用法
m_db=# SELECT CASE WHEN case_col1 = 1 THEN 'a' ELSE 'b' END FROM test_case1;
 case
------
 a
 b
 b
 a
(4 rows)
```



```
m_db=# SELECT CASE case_col1 WHEN 1 THEN 'a' ELSE 'b' END FROM test_case1;
 case
------
 a
 b
 b
 a
(4 rows)
```

说明

在精度传递场景下，使用CASE WHEN语句时，会进行类型转换和精度重新计算，导致最终的输出结果与CASE子句对比会出现末尾多零场景或末尾少零场景：

- 补零场景：CASE节点会根据CASE子句的精度计算CASE节点精度，当THEN子句的精度比CASE节点的精度小时，会在CASE节点末尾补零。在CASE WHEN语句中出现的用户变量，用户变量将使用所保存类型的默认精度作为用户变量的精度。
- 末尾少零场景：多层CASE WHEN嵌套时，内层CASE执行类型转换之后，只保留内层CASE的精度，外层CASE无法得到THEN子句的精度信息，因此外层CASE会根据内层CASE的精度计算的精度进行类型转换。当外层CASE类型转换时内层CASE精度比THEN子句少，会出现末尾少零场景。

示例：

```
-- 末尾补零场景
m_db=# SELECT 15.6 AS result;
 result
--------
   15.6
(1 row)

m_db=# SELECT CASE WHEN 1 < 2 THEN 15.6 ELSE  23.578 END AS result;
 result
--------
 15.600
(1 row)

m_db=# SELECT  greatest(12, 3.4, 15.6) AS result;
 result
--------
   15.6
(1 row)

m_db=# SELECT CASE WHEN 1 < 2 THEN greatest(12, 3.4, 15.6) ELSE greatest(123.4, 23.578, 36) END AS
result;
 result
--------
 15.600
(1 row)

-- 末尾零省略场景
m_db=# CREATE TABLE t1 AS SELECT (false/-timestamp '2008-12-31 23:59:59.678') AS result;
INSERT 0 1
m_db=# desc t1;
 Field  |    Type     | Null | Key | Default | Extra
--------+-------------+------+-----+---------+-------
 result | double(8,7) | YES  |     |         |
(1 row)

m_db=# SELECT (false/-timestamp '2008-12-31 23:59:59.678') AS result;
   result
------------
 -0.0000000
(1 row)

m_db=# cREATE TABLE t1 AS SELECT (case when 1<2 THEN false/-timestamp '2008-12-31 23:59:59.678'
ELSE 0016.11e3/'22.2' END) AS result;
INSERT 0 1
m_db=# DESC t1;
```



```
 Field  |  Type  | Null | Key | Default | Extra
--------+--------+------+-----+---------+-------
 result | double | YES  |     |         |
(1 row)

m_db=# SELECT (CASE WHEN 1<2 THEN false/-timestamp '2008-12-31 23:59:59.678' ELSE 0016.11E3/'22.2'
END) AS result;
 result
--------
     -0
(1 row)

m_db=# DROP TABLE t1;
DROP TABLE
m_db=# CREATE TABLE t1 AS SELECT (CASE WHEN 1+1=2 THEN CASE WHEN 1<2 THEN false/-timestamp
'2008-12-31 23:59:59.678' ELSE 0016.11e3/'22.2' END ELSE 'test' END) AS result;
INSERT 0 1
m_db=# DESC t1;
 Field  |     Type     | Null | Key | Default | Extra
--------+--------------+------+-----+---------+-------
 result | varchar(23) | YES  |     |         |
(1 row)

m_db=# SELECT (CASE WHEN 1+1=2 THEN CASE WHEN 1<2 THEN false/-timestamp '2008-12-31
23:59:59.678' ELSE 0016.11e3/'22.2' END ELSE 'test' END) AS result;
 result
--------
 -0
(1 row)
```

4.8 表达式

4.8.1 简单表达式

逻辑表达式

逻辑表达式的操作符和运算规则，请参见**逻辑操作符**。

比较表达式

常用的比较操作符，请参见**比较函数和比较操作符**。

除比较操作符外，还可以使用以下句式结构：

BETWEEN操作符

```
a  BETWEEN  x  AND  y
```

一般情况下等效于a >= x AND a <= y

```
a  NOT BETWEEN  x  AND y
```

一般情况下等效于a < x OR a > y

说明

- BETWEEN操作符在进行转换时，会先判断a和x的比较类型，根据a和x的比较类型和y判断最终的比较类型，这样在复杂类型的场景下会和a >= x AND a <= y的结果不相等。
- a BETWEEN x AND y，x和y不允许使用表达式，例如1 < 1。

示例

```
m_db=# SELECT 2 BETWEEN 1 AND 3 AS RESULT;
 result
----------
 t
(1 row)

m_db=# SELECT 2 >= 1 AND 2 <= 3 AS RESULT;
 result
----------
 t
(1 row)

m_db=# SELECT 2 NOT BETWEEN 1 AND 3 AS RESULT;
 result
----------
 f
(1 row)

m_db=# SELECT 2 < 1 OR 2 > 3 AS RESULT;
 result
----------
 f
(1 row)

m_db=# SELECT '2' between 'a' AND 3.0 AS RESULT;
WARNING:  Truncated incorrect double value: 'a'
CONTEXT:  referenced column: RESULT
 result
----------
 t
(1 row)

m_db=# SELECT '2' >= 'a' AND '2' <= 3.0 AS RESULT;
 result
----------
 f
(1 row)
```

4.8.2 条件表达式

在执行SQL语句时，可通过条件表达式筛选出符合条件的数据。

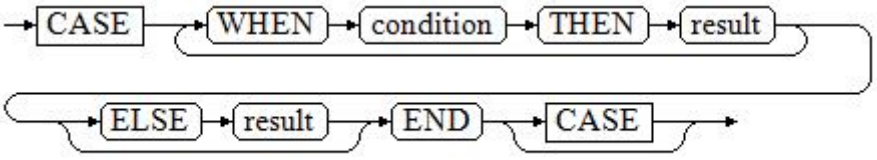
条件表达式主要有以下几种：

- CASE

  CASE表达式是条件表达式，类似于其他编程语言中的CASE语句。

  CASE表达式的语法图请参考图4-1。

  图 4-1 case::=

  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WHEN condition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。
- 支持对XML类型数据操作。

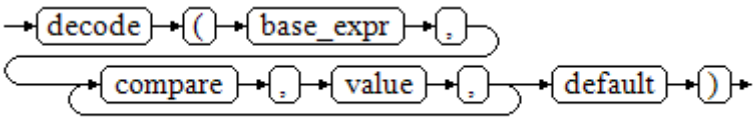
示例：

```
m_db=# CREATE SCHEMA tpcds;

m_db=# CREATE TABLE tpcds.case_when_t1(CW_COL1 INT);

m_db=# INSERT INTO tpcds.case_when_t1 VALUES (1), (2), (3);

m_db=# SELECT * FROM tpcds.case_when_t1;
cw_col1
---------
 1
 2
 3
(3 rows)

m_db=# SELECT CW_COL1, CASE WHEN CW_COL1=1 THEN 'one' WHEN CW_COL1=2 THEN 'two'
ELSE 'other' END FROM tpcds.case_when_t1 ORDER BY 1;
 cw_col1 | case
---------+-------
       1 | one
       2 | two
       3 | other
(3 rows)

m_db=# DROP TABLE tpcds.case_when_t1;

m_db=# DROP SCHEMA tpcds;
```

- DECODE

  DECODE的语法图请参见图4-2。

  图 4-2 decode::=

  将表达式base_expr与后面的每个compare(n) 进行比较，如果匹配返回相应的value(n)。如果没有发生匹配，则返回default。

  支持对XML类型数据操作。

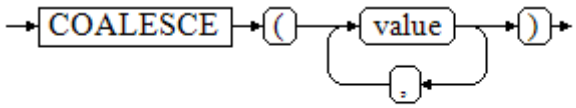
  示例请参见《开发指南》的“SQL参考 > 函数和操作符 > 条件表达式函数”章节。

```
m_db=# SELECT DECODE('A','A',1,'B',2,0);
 case
------
    1
(1 row)
```

- COALESCE

  COALESCE的语法图请参见图4-3。

图 4-3 coalesce::=



COALESCE返回它的第一个非NULL的参数值。如果参数都为NULL，则返回NULL。它常用于在显示数据时用缺省值替换NULL。和CASE表达式一样，COALESCE只计算用来判断结果的参数，即在第一个非空参数右边的参数不会被计算。

支持对XML类型数据操作。

示例

```
m_db=# CREATE SCHEMA tpcds;
m_db=# CREATE TABLE tpcds.c_tabl(description varchar(10), short_description varchar(10),
final_value varchar(10));

m_db=# INSERT INTO tpcds.c_tabl VALUES('abc', 'efg', '123');
m_db=# INSERT INTO tpcds.c_tabl VALUES(NULL, 'efg', '123');

m_db=# INSERT INTO tpcds.c_tabl VALUES(NULL, NULL, '123');

m_db=# SELECT description, short_description, final_value, COALESCE(description, short_description,
final_value) FROM tpcds.c_tabl ORDER BY 1, 2, 3, 4;
 description | short_description | final_value| coalesce
-------------+-------------------+------------+----------
 abc         | efg               | 123        | abc
             | efg               | 123        | efg
             |                   | 123        | 123
(3 rows)

m_db=# DROP TABLE tpcds.c_tabl;
m_db=# DROP SCHEMA tpcds;
```

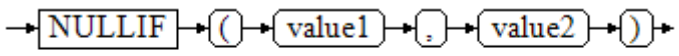
如果description不为NULL，则返回description的值，否则计算下一个参数short_description；如果short_description不为NULL，则返回short_description的值，否则计算下一个参数final_value；如果final_value不为NULL，则返回final_value的值，否则返回（none）。

```
m_db=# SELECT COALESCE(NULL,'Hello World');
   coalesce
---------------
 Hello World
(1 row)
```

- NULLIF

  NULLIF的语法图请参见**图4-4**。

  图 4-4 nullif::=

  只有当value1和value2相等时，NULLIF才返回NULL。否则它返回value1。支持对XML类型数据操作。

  示例

```
m_db=# CREATE SCHEMA tpcds;
m_db=# CREATE TABLE tpcds.null_if_t1 (
```



```
    NI_VALUE1 VARCHAR(10),
    NI_VALUE2 VARCHAR(10)
);

m_db=# INSERT INTO tpcds.null_if_t1 VALUES('abc', 'abc');
m_db=# INSERT INTO tpcds.null_if_t1 VALUES('abc', 'efg');

m_db=# SELECT NI_VALUE1, NI_VALUE2, NULLIF(NI_VALUE1, NI_VALUE2) FROM tpcds.null_if_t1
ORDER BY 1, 2, 3;

 ni_value1 | ni_value2 | nullif
-----------+-----------+--------
 abc       | abc       |
 abc       | efg       | abc
(2 rows)
m_db=# DROP TABLE tpcds.null_if_t1;
m_db=# DROP SCHEMA tpcds;
```

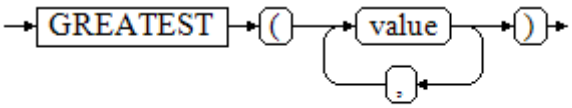
如果value1等于value2则返回NULL，否则返回value1。

```
m_db=# SELECT NULLIF('Hello','Hello World');
 nullif
--------
 Hello
(1 row)
```

- GREATEST（最大值），LEAST（最小值）

  GREATEST的语法图请参见图4-5。

  图 4-5 greatest::=

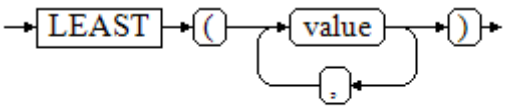
  

  从一个任意数字表达式的列表里选取最大的数值。支持对XML类型数据操作。

```
m_db=# SELECT greatest(9000,155555,2.01);
 greatest
----------
   155555
(1 row)
```

  LEAST的语法图请参见图4-6。

  图 4-6 least::=

  从一个任意数字表达式的列表里选取最小的数值。

  以上的数字表达式必须都可以转换成一个普通的数据类型，该数据类型将是结果类型。

  列表中的NULL值将被忽略。只有所有表达式的结果都是NULL的时候，结果才是NULL。

  支持对XML类型数据操作。


```
m_db=# SELECT least(9000,2);
 least
-------
     2
(1 row)
```

示例请参见《开发指南》的“SQL参考 > 函数和操作符 > 条件表达式函数”章节。

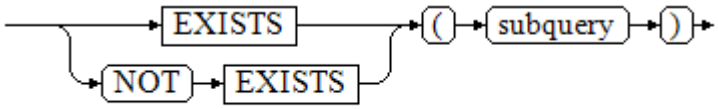
4.8.3 子查询表达式

子查询表达式主要有以下几种。

EXISTS/NOT EXISTS

EXISTS/NOT EXISTS的语法图请参见图4-7。

图 4-7 EXISTS/NOT EXISTS::=



- EXISTS的参数是一个任意的SELECT语句，或者是子查询。
- 系统对子查询进行运算以判断它是否返回行。如果它至少返回一行，则EXISTS结果就为真；如果子查询没有返回任何行， EXISTS的结果是假。
- 这个子查询通常只运行到能判断它是否可以生成至少一行为止，而不是等到全部运算结束。

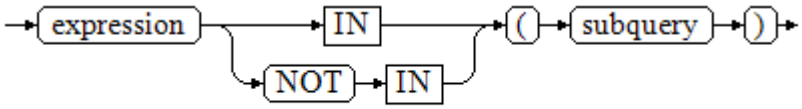
示例：

```
m_db=# CREATE SCHEMA tpcds;
m_db=# CREATE TABLE tpcds.store_returns
(
  sr_reason_sk     integer,
  sr_reason_id      character(16),
  sr_reason_desc    character(100),
  sr_customer_sk    integer
);
m_db=# CREATE TABLE tpcds.date_dim
(
  d_dom            integer,
  d_reason_desc     character(100)
);
m_db=# INSERT INTO tpcds.store_returns VALUES(1, 'test_001', 'test_desc_001', 5), (12, 'test_002',
'test_desc_002', 8), (23, 'test_003', 'test_desc_003', 9), (14, 'test_004', 'test_desc_004', 10), (45, 'test_005',
'test_desc_005', 12), (55, 'test_006', 'test_desc_006', 1), (43, 'test_007', 'test_desc_007', 12);
m_db=# INSERT INTO tpcds.date_dim VALUES(1, 'test_dim_desc_001'), (6, 'test_dim_desc_002'), (9,
'test_dim_desc_003'), (19, 'test_dim_desc_004'), (12, 'test_dim_desc_005'), (23, 'test_dim_desc_006');

m_db=# SELECT sr_reason_sk,sr_customer_sk FROM tpcds.store_returns WHERE EXISTS (SELECT d_dom
FROM tpcds.date_dim WHERE d_dom = store_returns.sr_reason_sk AND sr_customer_sk <10);
 sr_reason_sk | sr_customer_sk
--------------+----------------
            1 |              5
           12 |              8
           23 |              9
(3 rows)
```


IN/NOT IN

IN/NOT IN的语法请参见图4-8。

图 4-8 IN/NOT IN::=



- IN/NOT IN右侧为一个圆括号括起来的子查询，它只返回一个字段。
- IN/NOT IN左侧表达式对子查询结果的每一行进行一次计算和比较。
- 当IN/NOT IN语句中如果含有任何相等的子查询行，则IN结果为真；如果没有相等行，则结果为假（包括子查询没有返回任何行的情况）。
- 表达式或子查询行里的NULL遵照SQL处理布尔值和NULL组合时的规则。如果两个行对应的字段都相等且非空，则这两行相等；如果任意对应字段不等且非空，则这两行不等；否则结果是未知（NULL）。如果每一行的结果都是不等或NULL，并且至少有一个NULL，则IN的结果是NULL。
- 数组表达式IN操作符支持字符串类型、二进制类型、位串类型含零字符串不截断。

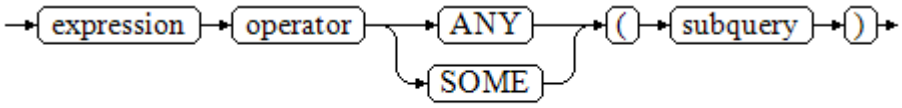
示例：

```
m_db=# SELECT sr_reason_sk,sr_customer_sk  FROM tpcds.store_returns WHERE sr_customer_sk IN (SELECT
d_dom FROM tpcds.date_dim WHERE d_dom < 10);
sr_reason_sk | sr_customer_sk
--------------+----------------
          55 |              1
          23 |              9
(2 rows)
```

ANY/SOME

ANY/SOME的语法图请参见图4-9。

图 4-9 any/some::=



- ANY/SOME右边是一个圆括号括起来的子查询，它只返回一个字段。
- ANY/SOME左边表达式使用operator对子查询结果的每一行进行一次计算和比较，其结果是布尔值。如果至少获得一个真值，则ANY结果为真；如果全部获得假值，则结果是假（包括子查询没有返回任何行的情况）。SOME是ANY的同义词。IN与ANY可以等效替换。

示例：

```
m_db=# SELECT sr_reason_sk,sr_customer_sk  FROM tpcds.store_returns WHERE sr_customer_sk < ANY
(SELECT d_dom FROM tpcds.date_dim WHERE d_dom < 10);
```

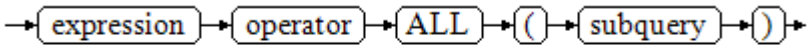


```
sr_reason_sk | sr_customer_sk
--------------+----------------
           1 |              5
          12 |              8
          55 |              1
(3 rows)
```

ALL

ALL的语法请参见图4-10。

图 4-10 all::=



- ALL右边是一个圆括号括起来的子查询，它只返回一个字段。
- ALL左边表达式使用operator对子查询结果的每一行进行一次计算和比较，其结果是布尔值。如果全部获得真值，ALL结果为真（包括子查询没有返回任何行的情况）；如果至少获得一个假值，则结果是假。

示例：

```
m_db=# SELECT sr_reason_sk,sr_customer_sk  FROM tpcds.store_returns WHERE sr_customer_sk <
all(SELECT d_dom FROM tpcds.date_dim WHERE d_dom < 10);
 sr_reason_sk | sr_customer_sk
--------------+----------------
(0 rows)

m_db=# DROP TABLE tpcds.store_returns;
m_db=# DROP TABLE tpcds.date_dim;
m_db=# DROP SCHEMA tpcds;
```

4.8.4 行表达式

行比较允许使用以下操作符=、<>、<、<=、>、>=。暂不支持<=>等操作符，或其中一个相似的语义符。

语法如下：

```
row_constructor operator row_constructor
```

说明

- 操作符两边都是一个行构造器，两行值必须具有相同数目的字段，每一行都进行比较。
- 支持的数据类型和比较操作符一致。

<、<=、>、>=

在<、<=、>、>=的情况下，行中元素从左到右依次比较，直到遇到一对不相等的元素或者一对为空的元素。如果这对元素中存在至少一个null值，则比较结果是未知的null，否则这对元素的比较结果为最终的结果。如果最终没有遇到不相等或者为空的元素，则认为这两行值相等，根据操作符含义判断最终结果。

示例：

```
m_db=# SELECT (1,2,NULL) < (1,3,0) AS RESULT;
 result
```



```
----------
 t
(1 row)

m_db=# SELECT (4,5,6) > (3,2,1) AS RESULT;
 result
--------
 t
(1 row)

m_db=# SELECT (4,1,1) > (3,2,1) AS RESULT;
 result
--------
 t
(1 row)

m_db=# SELECT ('TEST','DATA') > ('DATA','DATA') AS RESULT;
 result
--------
 t
(1 row)

m_db=# SELECT (4,1,1) > (3,2,NULL) AS RESULT;
 result
--------
 t
(1 row)

m_db=# SELECT (NULL,1,1) > (3,2,1) AS RESULT;
 result
--------

(1 row)

m_db=# SELECT (NULL,5,6) > (NULL,5,6) AS RESULT;
 result
--------

(1 row)

m_db=# SELECT (4,5,6) > (4,5,6) AS RESULT;
 result
--------
 f
(1 row)

m_db=# SELECT (2,2,5) >= (2,2,3) AS RESULT;
 result
--------
 t
(1 row)

m_db=# SELECT (2,2,1) <= (2,2,3) AS RESULT;
 result
--------
 t
(1 row)
```

=、<>

=、<>和别的操作符使用略有不同。如果两行值的所有字段都是非空并且符合操作符条件，则认为两行是符合操作符条件的；如果两行值的任意字段为非空并且不符合操作符条件，则认为两行是不符合操作符条件的；如果两行值的任意字段为空，则比较的结果是未知的null。

示例：


```
m_db=# SELECT (1,2,3) = (1,2,3) AS RESULT;
 result
--------
 t
(1 row)

m_db=# SELECT (1,2,3) <> (2,2,3) AS RESULT;
 result
--------
 t
(1 row)

m_db=# SELECT (2,2,3) <> (2,2,NULL) AS RESULT;
 result
--------

(1 row)

m_db=#  SELECT (NULL,5,6) <> (NULL,5,6) AS RESULT;
 result
--------

(1 row)
```

4.8.5 时间间隔表达式

语法：INTERVAL EXPR UNIT

说明：时间间隔表达式interval_expr表示一个时间间隔，其语法如上，INTERVAL为关键字，EXPR为任意表达式，将解释为一个时间类型的数值，UNIT为特定关键字，用于表示数值的单位，如HOUR、DAY、WEEK等。关键字INTERVAL和说明符不区分大小写。

时间间隔表达式中UNIT的取值范围如下**表4-68**所示。

说明

- interval_expr一般用于与另一个时间类型数值做运算，可作为函数DATE_ADD()、DATE_SUB()、ADDDATE()、SUBDATE()和TIMESTAMPADD()的入参，
- 也可直接用加减运算符+、-与另一个时间类型数值进行计算。
- 在做减法运算的时候不可以将interval_expr放在左侧，因一个时间间隔减去一个时间不符合逻辑。
- 对于**表1 时间间隔表达式UNIT取值范围**所示的时间单位UNIT可分为单时间单位和多时间单位两部分：
  - 单时间单位从MICROSECOND到YEAR，会将EXPR解释为单一单位的时间数值，如'1'YEAR 表示1年。
  - 多时间单位从SECOND_MICROSECOND到YEAR_MONTH，会将EXPR解释为跨时间单位的时间数值，例如DAY_MINUTE会将EXPR解释为xxx日xxx时xxx分三部分。
- EXPR中一段连续的数字字符会被解析为一个有效数字，可用任意字符隔开，例如'1:1:1' DAY_MINUTE表示1日1小时1分。若解析的有效数字个数不足跨时间单位中单一时间单位数，会将高位补0，例如'1:1' DAY_MINUTE表示0日1小时1分。

表 4-68 时间间隔表达式 UNIT 取值范围

UNIT取值范围	预期EXPR格式
MICROSECOND	MICROSECOND
SECOND	SECOND

UNIT取值范围	预期EXPR格式
MINUTE	MINUTE
HOUR	HOUR
DAY	DAY
WEEK	WEEK
MONTH	MONTH
QUARTER	QUARTER
YEAR	YEAR
SECOND_MICROSECOND	SECOND_MICROSECOND
MINUTE_MICROSECOND	MINUTE_MICROSECOND
MINUTE_SECOND	MINUTE_SECOND
HOUR_MICROSECOND	HOUR_MICROSECOND
HOUR_SECOND	HOUR_SECOND
HOUR_MINUTE	HOUR_MINUTE
DAY_MICROSECOND	DAY_MICROSECOND
DAY_SECOND	DAY_SECOND
DAY_MINUTE	DAY_MINUTE
DAY_HOUR	DAY_HOUR
YEAR_MONTH	YEAR_MONTH

示例：

```
m_db=# select date_add('1999-12-31',interval '1' day);
  date_add
------------
 2000-01-01
(1 row)

m_db=# select adddate('1998-12-03', interval '1' YEAR);
  adddate
------------
 1999-12-03
(1 row)

m_db=# select date_sub('1999-12-31',interval '1' day);
  date_sub
------------
 1999-12-30
(1 row)

m_db=# select subdate('1998-12-03', interval '1' YEAR);
  subdate
------------
 1997-12-03
```



```
(1 row)

m_db=# select timestampadd(day, '1', '1999-12-31');
 timestampadd
--------------
 2000-01-01
(1 row)

m_db=# select '1998-12-03' + interval '1' YEAR;
  ?column?
------------
 1999-12-03
(1 row)

m_db=# select '1998-12-03' - interval '1' YEAR;
  ?column?
------------
 1997-12-03
(1 row)

m_db=# select interval '1' YEAR + '1998-12-03';
  ?column?
------------
 1999-12-03
(1 row)

m_db=# select interval '1' YEAR - '1998-12-03';
ERROR:  syntax error at or near "-"
LINE 1: select interval '1' YEAR - '1998-12-03';
                            ^
```

4.9 注释

4.9.1 单行注释

默认情况下，数据库中“--”后的内容被视作单行注释内容，不进行语法解析。设置参数m_format_behavior_compat_options包含forbid_none_space_comment选项时，--后面必须添加一个空格，空格后的内容才会被视为单行注释。

示例

```
-- 单行注释基本场景：
m_db=# SET m_format_behavior_compat_options='forbid_none_space_comment';
SET
m_db=# SELECT 'abc' -- 1;
m_db-# ;
 ?column?
----------
 abc
(1 row)

-- 单行注释特殊场景：
m_db=# select 'a'--;
 ?column?
----------
 a
(1 row)

-- 注释优先级低于引号优先级：
m_db=# select '-- abc';
 ?column?
----------
 -- abc
(1 row)
```


5 系统表和系统视图

M-Compatibility在查询系统表和系统视图的时候，需要使用小写的系统表和系统视图名称，information_schema中的系统表和系统视图除外。

5.1 系统表

M-Compatibility继承了GaussDB原有系统表，具体请参考《开发指南》中“系统表和系统视图 > 系统表”章节。

5.2 系统视图

M-Compatibility继承了GaussDB原有的部分系统视图，继承的视图如表5-1所示，视图的具体定义及视图信息请参见《开发指南》中“系统表和系统视图”章节。

说明

- 当前M-Compatibility模式数据库不支持审计能力，gs_auditing、gs_auditing_access和gs_auditing_privilege视图数据为空。
- 当前M-Compatibility模式数据库不支持脱敏能力，gs_masking视图数据为空。

表 5-1 M-Compatibility 模式数据库继承 GaussDB 数据库的视图

视图分类	视图名称
系统视图	database links
	gs_db_links
分区表	gs_stat_all_partitions
	gs_stat_xact_all_partitions
	gs_statio_all_partitions
审计	gs_auditing
	gs_auditing_access

视图分类	视图名称
	gs_auditing_privilege
用户和权限管理	gs_db_privileges
	gs_labels
	pg_rlspolicies
	pg_roles
	pg_seclabels
	pg_shadow
	pg_user
	pg_user_mappings
动态脱敏	gs_masking
透明加密	pg_tde_info
通信	gs_comm_listen_address_ext_info
	gs_get_listen_address_ext_info
	pg_comm_delay
	pg_comm_recv_stream
	pg_comm_send_stream
	pg_comm_status
	pg_get_invalid_backends
段页式存储	gs_seg_datafile_layout
	gs_seg_datafiles
	gs_seg_extents
	gs_seg_segments
	gs_seg_segment_layout
	gs_seg_spc_segments
	gs_seg_spc_extents
	gs_seg_spc_remain_extents
	gs_seg_spc_remain_segments
SPM计划管理	gs_spm_sql_baseline
	gs_spm_sql_param
	gs_spm_sql_evolution

视图分类	视图名称
	gs_spm_sys_baseline
负载管理	gs_wlm_cgroup_info
	gs_wlm_plan_operator_history
	gs_wlm_rebuild_user_resource_pool
	gs_wlm_user_info
多租户	mtd_user
其他系统视图	gs_all_control_group_info
	gs_all_prepared_statements
	gs_comm_proxy_thread_status
	gs_file_stat
	gs_get_control_group_info
	gs_glc_memory_detail
	gs_gsc_memory_detail
	gs_instance_time
	gs_lsc_memory_detail
	gs_my_plan_trace
	gs_os_run_info
	gs_redo_stat
	gs_session_all_settings
	gs_session_memory
	gs_session_memory_context
	gs_session_memory_detail
	gs_session_stat
	gs_session_time
	gs_sql_count
	gs_stat_session_cu
	gs_thread_memory_context
	gs_total_memory_detail
	gs_total_nodegroup_memory_detail
	pg_available_extension_versions

视图分类	视图名称
	pg_available_extensions
	pg_control_group_config
	pg_cursors
	pg_ext_stats
	pg_get_senders_catchup_time
	pg_group
	pg_gtt_attached_pids
	pg_gtt_relstats
	pg_gtt_stats
	pg_indexes
	pg_locks
	pg_node_env
	pg_os_threads
	pg_prepared_statements
	pg_prepared_xacts
	pg_replication_origin_status
	pg_replication_slots
	pg_running_xacts
	pg_session_iostat
	pg_settings
	pg_stat_activity
	pg_stat_activity_ng
	pg_stat_all_indexes
	pg_stat_all_tables
	pg_stat_bad_block
	pg_stat_bgwriter
	pg_stat_database
	pg_stat_database_conflicts
	pg_stat_replication
	pg_stat_sys_indexes

视图分类	视图名称
	pg_stat_sys_tables
	pg_stat_user_functions
	pg_stat_user_indexes
	pg_stat_user_tables
	pg_stat_xact_all_tables
	pg_stat_xact_sys_tables
	pg_stat_xact_user_functions
	pg_stat_xact_user_tables
	pg_statio_all_indexes
	pg_statio_all_sequences
	pg_statio_all_tables
	pg_statio_sys_indexes
	pg_statio_sys_sequences
	pg_statio_sys_tables
	pg_statio_user_indexes
	pg_statio_user_sequences
	pg_statio_user_tables
	pg_stats
	pg_tables
	pg_thread_wait_status
	pg_timezone_abbrevs
	pg_timezone_names
	pg_total_memory_detail
	pg_total_user_resource_info
	pg_total_user_resource_info_oid
	pg_variable_info
	pg_views
	gs_global_archive_status
	gs_workload_rule_stat

M-Compatibility继承了pg_available_extension_versions、pg_available_extensions、pg_comm_delay、pg_comm_recv_stream等系统视图，继承了原有GaussDB系统视图的同时，在对应系统视图下建立了以GS_前缀的别名，具体如表5-2所示。 为了保证兼容性，请优先使用GS_前缀的更名视图。

表 5-2 M-Compatibility 模式数据库系统视图别名

视图名称	视图别名
pg_available_extension_versions	gs_available_extension_versions
pg_available_extensions	gs_available_extensions
pg_comm_delay	gs_comm_delay
pg_comm_recv_stream	gs_comm_recv_stream
pg_comm_send_stream	gs_comm_send_stream
pg_comm_status	gs_comm_status
pg_control_group_config	gs_control_group_config
pg_cursors	gs_cursors
pg_ext_stats	gs_ext_stats
pg_get_invalid_backends	gs_get_invalid_backends
pg_get_senders_catchup_time	gs_get_senders_catchup_time
pg_group	gs_group
pg_gtt_attached_pids	gs_gtt_attached_pids
pg_gtt_relstats	gs_gtt_relstats
pg_gtt_stats	gs_gtt_stats
pg_indexes	gs_indexes
pg_locks	gs_locks
pg_node_env	gs_node_env
pg_os_threads	gs_os_threads
pg_prepared_statements	gs_prepared_statements
pg_prepared_xacts	gs_prepared_xacts
pg_replication_origin_status	gs_replication_origin_status
pg_replication_slots	gs_replication_slots
pg_rlspolicies	gs_rlspolicies
pg_roles	gs_roles
pg_rules	gs_rules

视图名称	视图别名
pg_running_xacts	gs_running_xacts
pg_seclabels	gs_seclabels
pg_session_iostat	gs_session_iostat
pg_session_wlmstat	gs_session_wlmstat
pg_settings	gs_settings
pg_shadow	gs_shadow
pg_stat_activity	gs_stat_activity
pg_stat_activity_ng	gs_stat_activity_ng
pg_stat_all_indexes	gs_stat_all_indexes
pg_stat_all_tables	gs_stat_all_tables
pg_stat_bad_block	gs_stat_bad_block
pg_stat_bgwriter	gs_stat_bgwriter
pg_stat_database	gs_stat_database
pg_stat_database_conflicts	gs_stat_database_conflicts
pg_stat_replication	gs_stat_replication
pg_stat_sys_indexes	gs_stat_sys_indexes
pg_stat_sys_tables	gs_stat_sys_tables
pg_stat_user_functions	gs_stat_user_functions
pg_stat_user_indexes	gs_stat_user_indexes
pg_stat_user_tables	gs_stat_user_tables
pg_stat_xact_all_tables	gs_stat_xact_all_tables
pg_stat_xact_sys_tables	gs_stat_xact_sys_tables
pg_stat_xact_user_functions	gs_stat_xact_user_functions
pg_stat_xact_user_tables	gs_stat_xact_user_tables
pg_statio_all_indexes	gs_statio_all_indexes
pg_statio_all_sequences	gs_statio_all_sequences
pg_statio_all_tables	gs_statio_all_tables
pg_statio_sys_indexes	gs_statio_sys_indexes
pg_statio_sys_sequences	gs_statio_sys_sequences
pg_statio_sys_tables	gs_statio_sys_tables

视图名称	视图别名
pg_statio_user_indexes	gs_statio_user_indexes
pg_statio_user_sequences	gs_statio_user_sequences
pg_statio_user_tables	gs_statio_user_tables
pg_stats	gs_stats
pg_tables	gs_tables
pg_tde_info	gs_tde_info
pg_thread_wait_status	gs_thread_wait_status
pg_timezone_abbrevs	gs_timezone_abbrevs
pg_timezone_names	gs_timezone_names
pg_total_memory_detail	gs_total_memory_detail
pg_total_user_resource_info	gs_total_user_resource_info
pg_total_user_resource_info_oid	gs_total_user_resource_info_oid
pg_user	gs_user
pg_user_mappings	gs_user_mappings
pg_variable_info	gs_variable_info
pg_views	gs_views
pg_wlm_statistics	gs_wlm_statistics

6 Schema

继承了DBE_PLDEVELOPER、SYS Schema，具体请参考《开发指南》的“Schema”章节。继承了DBE_PERF和PG_CATALOG Schema的部分视图同时，在此Schema下新增gs_前缀的更名视图。 其对应关系如表6-1所示。 为了保证兼容性，请优先使用带有gs_前缀的别名视图。

表 6-1 M-Compatibility 模式数据库系统视图别名

schema名称	原视图名称	视图别名
dbe_perf	ai_watchdog_detection_warnings	gs_ai_watchdog_detection_warnings
dbe_perf	ai_watchdog_monitor_status	gs_ai_watchdog_monitor_status
dbe_perf	ai_watchdog_parameters	gs_ai_watchdog_parameters
dbe_perf	bgwriter_stat	gs_bgwriter_stat
dbe_perf	class_vital_info	gs_class_vital_info
dbe_perf	config_settings	gs_config_settings
dbe_perf	file_iostat	gs_file_iostat
dbe_perf	file_redo_iostat	gs_file_redo_iostat
dbe_perf	global_bgwriter_stat	gs_global_bgwriter_stat
dbe_perf	global_candidate_status	gs_global_candidate_status
dbe_perf	global_ckpt_status	gs_global_ckpt_status
dbe_perf	global_config_settings	gs_global_config_settings
dbe_perf	global_double_write_status	gs_global_double_write_status
dbe_perf	global_file_iostat	gs_global_file_iostat
dbe_perf	global_file_redo_iostat	gs_global_file_redo_iostat

schema名称	原视图名称	视图别名
dbe_perf	global_instance_time	gs_global_instance_time
dbe_perf	global_locks	gs_global_locks
dbe_perf	global_memory_node_detail	gs_global_memory_node_detail
dbe_perf	global_operator_history	gs_global_operator_history
dbe_perf	global_operator_history_table	gs_global_operator_history_table
dbe_perf	global_operator_runtime	gs_global_operator_runtime
dbe_perf	global_os_runtime	gs_global_os_runtime
dbe_perf	global_os_threads	gs_global_os_threads
dbe_perf	global_pagewriter_status	gs_global_pagewriter_status
dbe_perf	global_parallel_decode_status	gs_global_parallel_decode_status
dbe_perf	global_parallel_decode_thread_info	gs_global_parallel_decode_thread_info
dbe_perf	global_plancache_clean	gs_global_plancache_clean
dbe_perf	global_plancache_status	gs_global_plancache_status
dbe_perf	global_record_reset_time	gs_global_record_reset_time
dbe_perf	global_recovery_status	gs_global_recovery_status
dbe_perf	global_redo_status	gs_global_redo_status
dbe_perf	global_rel_iostat	gs_global_rel_iostat
dbe_perf	global_replication_slots	gs_global_replication_slots
dbe_perf	global_replication_stat	gs_global_replication_stat
dbe_perf	global_rto_status	gs_global_rto_status
dbe_perf	global_session_memory	gs_global_session_memory
dbe_perf	global_session_memory_detail	gs_global_session_memory_detail
dbe_perf	global_session_stat	gs_global_session_stat
dbe_perf	global_session_stat_activity	gs_global_session_stat_activity
dbe_perf	global_session_time	gs_global_session_time
dbe_perf	global_shared_memory_detail	gs_global_shared_memory_detail

schema名称	原视图名称	视图别名
dbe_perf	global_single_flush_dw_status	gs_global_single_flush_dw_status
dbe_perf	global_slow_query_history	gs_global_slow_query_history
dbe_perf	global_slow_query_info	gs_global_slow_query_info
dbe_perf	global_stat_all_indexes	gs_global_stat_all_indexes
dbe_perf	global_stat_all_tables	gs_global_stat_all_tables
dbe_perf	global_stat_bad_block	gs_global_stat_bad_block
dbe_perf	global_stat_database	gs_global_stat_database
dbe_perf	global_stat_database_conflicts	gs_global_stat_database_conflicts
dbe_perf	global_stat_db_cu	gs_global_stat_db_cu
dbe_perf	global_stat_session_cu	gs_global_stat_session_cu
dbe_perf	global_stat_sys_indexes	gs_global_stat_sys_indexes
dbe_perf	global_stat_sys_tables	gs_global_stat_sys_tables
dbe_perf	global_stat_user_functions	gs_global_stat_user_functions
dbe_perf	global_stat_user_indexes	gs_global_stat_user_indexes
dbe_perf	global_stat_user_tables	gs_global_stat_user_tables
dbe_perf	global_stat_xact_all_tables	gs_global_stat_xact_all_tables
dbe_perf	global_stat_xact_sys_tables	gs_global_stat_xact_sys_tables
dbe_perf	global_stat_xact_user_functions	gs_global_stat_xact_user_functions
dbe_perf	global_stat_xact_user_tables	gs_global_stat_xact_user_tables
dbe_perf	global_statement_complex_history	gs_global_statement_complex_history
dbe_perf	global_statement_complex_history_table	gs_global_statement_complex_history_table
dbe_perf	global_statement_complex_runtime	gs_global_statement_complex_runtime
dbe_perf	global_statement_count	gs_global_statement_count
dbe_perf	global_statio_all_indexes	gs_global_statio_all_indexes

schema名称	原视图名称	视图别名
dbe_perf	global_statio_all_sequences	gs_global_statio_all_sequences
dbe_perf	global_statio_all_tables	gs_global_statio_all_tables
dbe_perf	global_statio_sys_indexes	gs_global_statio_sys_indexes
dbe_perf	global_statio_sys_sequences	gs_global_statio_sys_sequences
dbe_perf	global_statio_sys_tables	gs_global_statio_sys_tables
dbe_perf	global_statio_user_indexes	gs_global_statio_user_indexes
dbe_perf	global_statio_user_sequences	gs_global_statio_user_sequences
dbe_perf	global_statio_user_tables	gs_global_statio_user_tables
dbe_perf	global_streaming_hadr_rto_and_rpo_stat	gs_global_streaming_hadr_rto_and_rpo_stat
dbe_perf	global_thread_wait_status	gs_global_thread_wait_status
dbe_perf	global_threadpool_status	gs_global_threadpool_status
dbe_perf	global_transactions_prepared_xacts	gs_global_transactions_prepared_xacts
dbe_perf	global_transactions_running_xacts	gs_global_transactions_running_xacts
dbe_perf	global_user_transaction	gs_global_user_transaction
dbe_perf	global_wait_events	gs_global_wait_events
dbe_perf	global_workload_transaction	gs_global_workload_transaction
dbe_perf	slow_query_history	gs_slow_query_history
dbe_perf	slow_query_info	gs_slow_query_info
dbe_perf	instance_time	gs_instance_time
dbe_perf	local_rel_iostat	gs_local_rel_iostat
dbe_perf	local_threadpool_status	gs_local_threadpool_status
dbe_perf	locks	gs_locks
dbe_perf	memory_node_detail	gs_memory_node_detail
dbe_perf	node_name	gs_node_name
dbe_perf	operator_history	gs_operator_history

schema名称	原视图名称	视图别名
dbe_perf	operator_history_table	gs_operator_history_table
dbe_perf	operator_runtime	gs_operator_runtime
dbe_perf	os_runtime	gs_os_runtime
dbe_perf	os_threads	gs_os_threads
dbe_perf	parallel_decode_status	gs_parallel_decode_status
dbe_perf	parallel_decode_thread_info	gs_parallel_decode_thread_info
dbe_perf	replication_slots	gs_replication_slots
dbe_perf	replication_stat	gs_replication_stat
dbe_perf	session_cpu_runtime	gs_session_cpu_runtime
dbe_perf	session_memory_detail	gs_session_memory_detail
dbe_perf	session_memory_runtime	gs_session_memory_runtime
dbe_perf	session_stat	gs_session_stat
dbe_perf	session_stat_activity	gs_session_stat_activity
dbe_perf	session_time	gs_session_time
dbe_perf	shared_memory_detail	gs_shared_memory_detail
dbe_perf	stat_all_indexes	gs_stat_all_indexes
dbe_perf	stat_all_tables	gs_stat_all_tables
dbe_perf	stat_bad_block	gs_stat_bad_block
dbe_perf	stat_database	gs_stat_database
dbe_perf	stat_database_conflicts	gs_stat_database_conflicts
dbe_perf	stat_sys_indexes	gs_stat_sys_indexes
dbe_perf	stat_sys_tables	gs_stat_sys_tables
dbe_perf	stat_user_functions	gs_stat_user_functions
dbe_perf	stat_user_indexes	gs_stat_user_indexes
dbe_perf	stat_user_tables	gs_stat_user_tables
dbe_perf	stat_xact_all_tables	gs_stat_xact_all_tables
dbe_perf	stat_xact_sys_tables	gs_stat_xact_sys_tables
dbe_perf	stat_xact_user_functions	gs_stat_xact_user_functions
dbe_perf	stat_xact_user_tables	gs_stat_xact_user_tables

schema名称	原视图名称	视图别名
dbe_perf	statement	gs_statement
dbe_perf	statement_complex_history	gs_statement_complex_history
dbe_perf	statement_complex_history_table	gs_statement_complex_history_table
dbe_perf	statement_complex_runtime	gs_statement_complex_runtime
dbe_perf	statement_count	gs_statement_count
dbe_perf	statement_history	gs_statement_history
dbe_perf	statement_iostat_complex_runtime	gs_statement_iostat_complex_runtime
dbe_perf	statement_responsetime_percentile	gs_statement_responsetime_percentile
dbe_perf	statement_wlmstat_complex_runtime	gs_statement_wlmstat_complex_runtime
dbe_perf	statio_all_indexes	gs_statio_all_indexes
dbe_perf	statio_all_sequences	gs_statio_all_sequences
dbe_perf	statio_all_tables	gs_statio_all_tables
dbe_perf	statio_sys_indexes	gs_statio_sys_indexes
dbe_perf	statio_sys_sequences	gs_statio_sys_sequences
dbe_perf	statio_sys_tables	gs_statio_sys_tables
dbe_perf	statio_user_indexes	gs_statio_user_indexes
dbe_perf	statio_user_sequences	gs_statio_user_sequences
dbe_perf	statio_user_tables	gs_statio_user_tables
dbe_perf	summary_file_iostat	gs_summary_file_iostat
dbe_perf	summary_file_redo_iostat	gs_summary_file_redo_iostat
dbe_perf	summary_rel_iostat	gs_summary_rel_iostat
dbe_perf	summary_stat_all_indexes	gs_summary_stat_all_indexes
dbe_perf	summary_stat_all_tables	gs_summary_stat_all_tables
dbe_perf	summary_stat_bad_block	gs_summary_stat_bad_block
dbe_perf	summary_stat_database	gs_summary_stat_database
dbe_perf	summary_stat_database_conflicts	gs_summary_stat_database_conflicts

schema名称	原视图名称	视图别名
dbe_perf	summary_stat_sys_indexes	gs_summary_stat_sys_indexes
dbe_perf	summary_stat_sys_tables	gs_summary_stat_sys_tables
dbe_perf	summary_stat_user_functions	gs_summary_stat_user_functions
dbe_perf	summary_stat_user_indexes	gs_summary_stat_user_indexes
dbe_perf	summary_stat_user_tables	gs_summary_stat_user_tables
dbe_perf	summary_stat_xact_all_tables	gs_summary_stat_xact_all_tables
dbe_perf	summary_stat_xact_sys_tables	gs_summary_stat_xact_sys_tables
dbe_perf	summary_stat_xact_user_functions	gs_summary_stat_xact_user_functions
dbe_perf	summary_stat_xact_user_tables	gs_summary_stat_xact_user_tables
dbe_perf	summary_statement	gs_summary_statement
dbe_perf	summary_statement_count	gs_summary_statement_count
dbe_perf	summary_statio_all_indexes	gs_summary_statio_all_indexes
dbe_perf	summary_statio_all_sequences	gs_summary_statio_all_sequences
dbe_perf	summary_statio_all_tables	gs_summary_statio_all_tables
dbe_perf	summary_statio_sys_indexes	gs_summary_statio_sys_indexes
dbe_perf	summary_statio_sys_sequences	gs_summary_statio_sys_sequences
dbe_perf	summary_statio_sys_tables	gs_summary_statio_sys_tables
dbe_perf	summary_statio_user_indexes	gs_summary_statio_user_indexes
dbe_perf	summary_statio_user_sequences	gs_summary_statio_user_sequences
dbe_perf	summary_statio_user_tables	gs_summary_statio_user_tables

schema名称	原视图名称	视图别名
dbe_perf	summary_transactions_prepared_xacts	gs_summary_transactions_prepared_xacts
dbe_perf	summary_transactions_running_xacts	gs_summary_transactions_running_xacts
dbe_perf	summary_user_login	gs_summary_user_login
dbe_perf	summary_workload_sql_count	gs_summary_workload_sql_count
dbe_perf	summary_workload_sql_elapse_time	gs_summary_workload_sql_elapse_time
dbe_perf	summary_workload_transaction	gs_summary_workload_transaction
dbe_perf	thread_wait_status	gs_thread_wait_status
dbe_perf	transactions_prepared_xacts	gs_transactions_prepared_xacts
dbe_perf	transactions_running_xacts	gs_transactions_running_xacts
dbe_perf	user_login	gs_user_login
dbe_perf	user_transaction	gs_user_transaction
dbe_perf	wait_events	gs_wait_events
dbe_perf	wlm_user_resource_config	gs_wlm_user_resource_config
dbe_perf	wlm_user_resource_runtime	gs_wlm_user_resource_runtime
dbe_perf	workload_sql_count	gs_workload_sql_count
dbe_perf	workload_sql_elapse_time	gs_workload_sql_elapse_time
dbe_perf	workload_transaction	gs_workload_transaction
dbe_perf	global_adio_completer_status	gs_global_adio_completer_status
dbe_perf	global_aio_slot_usage_status	gs_global_aio_slot_usage_status
dbe_perf	ai_watchdog_ftask_status	gs_ai_watchdog_ftask_status
dbe_perf	perf_query	gs_perf_query
pg_catalog	dv_sessions	gs_dv_sessions
pg_catalog	dv_session_longops	gs_dv_session_longops
pg_catalog	mpp_tables	gs_mpp_tables

schema名称	原视图名称	视图别名
pg_catalog	get_global_prepared_xacts	gs_get_global_prepared_xacts
pg_catalog	pgxc_prepared_xacts	gs_pgxc_prepared_xacts
pg_catalog	pgxc_thread_wait_status	gs_pgxc_thread_wait_status
pg_catalog	pg_comm_delay	gs_comm_delay
pg_catalog	pg_comm_recv_stream	gs_comm_recv_stream
pg_catalog	pg_comm_send_stream	gs_comm_send_stream
pg_catalog	pg_comm_status	gs_comm_status
pg_catalog	session_memory	gs_session_memory

6.1 Information Schema

M-Compatibility继承了GaussDB的Information_Schema下的原有视图，具体请参考《开发指南》中“Schema > Information_Schema”章节。

M-Compatibility继承并改名的GaussDB的Information Schema视图见下表。

继承改名后的视图名称
_gs_foreign_data_wrappers
_gs_foreign_servers
_gs_foreign_table_columns
_gs_foreign_tables
_gs_user_mappings
gs_administrable_role_authorizations
gs_applicable_roles
gs_attributes
gs_character_sets
gs_check_constraint_routine_usage
gs_check_constraints
gs_collation_character_set_applicability
gs_collations
gs_column_domain_usage
gs_column_options

继承改名后的视图名称
gs_column_privileges
gs_column_udt_usage
gs_columns
gs_constraint_column_usage
gs_constraint_table_usage
gs_data_type_privileges
gs_domain_constraints
gs_domain_udt_usage
gs_domains
gs_element_types
gs_enabled_roles
gs_foreign_data_wrapper_options
gs_foreign_data_wrappers
gs_foreign_server_options
gs_foreign_servers
gs_foreign_table_options
gs_foreign_tables
gs_information_schema_catalog_name
gs_key_column_usage
gs_parameters
gs_referential_constraints
gs_role_column_grants
gs_role_routine_grants
gs_role_table_grants
gs_role_udt_grants
gs_role_usage_grants
gs_routine_privileges
gs_routines
gs_schemata
gs_sequences

继承改名后的视图名称
gs_table_constraints
gs_table_privileges
gs_tables
gs_triggered_update_columns
gs_triggers
gs_udt_privileges
gs_usage_privileges
gs_user_defined_types
gs_user_mapping_options
gs_user_mappings
gs_view_column_usage
gs_view_routine_usage
gs_view_table_usage
gs_views

6.1.1 character_sets

character_sets视图提供有关可用字符集的信息。该视图为只读，不允许修改。所有用户对这个视图有“读取”权限。

表 6-2 information_schema.character_sets 字段

名称	类型	描述
CHARACTER_SET_NAME	varchar(32)	字符集名称。
DEFAULT_COLLATE_NAME	varchar(32)	字符集的默认排序规则。
DESCRIPTION	varchar(60)	字符集的描述主要说明字符集使用的默认字符序。
MAXLEN	bigint	存储一个字符所需的最大字节数。

GUC参数m_format_dev_version控制行为保持前向兼容，当m_format_dev_version取值为s2时，可使用最新功能，其他取值时保持前向兼容。

当m_format_dev_version取值不为s2时，视图仅展示当前数据库的字符集信息：


```
m_db=# SET m_format_dev_version=s1;
SET
m_db=# SELECT * FROM information_schema.character_sets;
 CHARACTER_SET_NAME | DEFAULT_COLLATE_NAME |          DESCRIPTION          | MAXLEN
--------------------+----------------------+-------------------------------+--------
 UTF8               | utf8mb4_general_ci   | utf8mb4_general_ci collation |      4
(1 row)
```

m_format_dev_version取值为s2时，视图可以查询所有可用字符集信息：

```
m_db=# SET m_format_dev_version=s2;
SET
m_db=# SELECT * FROM information_schema.character_sets;
 CHARACTER_SET_NAME | DEFAULT_COLLATE_NAME |          DESCRIPTION          | MAXLEN
--------------------+----------------------+-------------------------------+--------
 binary             | binary               | binary collation              |      1
 gbk                | gbk_chinese_ci       | gbk_chinese_ci collation      |      2
 utf8mb4            | utf8mb4_general_ci   | utf8mb4_general_ci collation |      4
 gb18030            | gb18030_chinese_ci   | gb18030_chinese_ci collation |      4
 latin1             | latin1_swedish_ci    | latin1_swedish_ci collation  |      1
(5 rows)
```

6.1.2 collations

collations视图提供了有关每个字符集的排序规则的信息。这个视图为只读，不允许修改。所有用户对这个视图有“读取”权限。

表 6-3 information_schema.collations 字段

名称	类型	描述
COLLATION_NAME	varchar(32)	排序规则名称。
CHARACTER_SET_NAME	varchar(32)	与排序规则关联的字符集的名称。
ID	bigint	排序规则id，具体可参考pg_collation.id。
IS_DEFAULT	varchar(3)	排序规则是否是其字符集的默认值。
IS_COMPILED	varchar(3)	字符集是否被编译到服务器中。
SORTLEN	bigint	当前版本暂不支持，默认为null。

GUC参数m_format_dev_version控制行为保持前向兼容，m_format_dev_version取值为s2时，可使用最新功能，其他取值保持前向兼容。

m_format_dev_version取值不为s2时，视图仅展示当前数据库的字符集相关的字符集排序规则信息：

```
m_db=# SET m_format_dev_version=s1;
SET
m_db=#
m_db=# SELECT * FROM information_schema.collations;
   COLLATION_NAME   | CHARACTER_SET_NAME | ID  | IS_DEFAULT | IS_COMPILED | SORTLEN
--------------------+--------------------+-----+------------+-------------+---------
 utf8mb4_0900_ai_ci | UTF8               | 255 |            | Yes         |
 utf8_bin           | UTF8               |  83 |            | Yes         |
 utf8_unicode_ci    | UTF8               | 192 |            | Yes         |
```



```
 utf8_general_ci    | UTF8              |  33 |           | Yes        |
 utf8mb4_bin        | UTF8              |  46 |           | Yes        |
 utf8mb4_unicode_ci | UTF8              | 224 |           | Yes        |
 utf8mb4_general_ci | UTF8              |  45 | Yes       | Yes        |
 POSIX              | UTF8              |     |           | Yes        |
 C                  | UTF8              |     |           | Yes        |
 default            | UTF8              |     |           | Yes        |
(10 rows)
```

m_format_dev_version取值为s2时，视图可以查询所有可用字符排序规则信息：

```
m_db=# SET m_format_dev_version=s2;
SET
m_db=# SELECT * FROM information_schema.collations;
   COLLATION_NAME    | CHARACTER_SET_NAME | ID  | IS_DEFAULT | IS_COMPILED | SORTLEN
---------------------+--------------------+-----+------------+-------------+---------
 binary              | binary             |  63 | Yes        | Yes         |
 gbk_chinese_ci      | gbk                |  28 | Yes        | Yes         |
 gbk_bin             | gbk                |  87 |            | Yes         |
 utf8mb4_general_ci  | utf8mb4            |  45 | Yes        | Yes         |
 utf8mb4_unicode_ci  | utf8mb4            | 224 |            | Yes         |
 utf8mb4_bin         | utf8mb4            |  46 |            | Yes         |
 utf8_general_ci     | utf8               |  33 |            | Yes         |
 utf8_unicode_ci     | utf8               | 192 |            | Yes         |
 utf8_bin            | utf8               |  83 |            | Yes         |
 utf8mb4_0900_ai_ci  | utf8mb4            | 255 |            | Yes         |
 gb18030_chinese_ci  | gb18030            | 248 | Yes        | Yes         |
 gb18030_bin         | gb18030            | 249 |            | Yes         |
 latin1_swedish_ci   | latin1             |   8 | Yes        | Yes         |
 latin1_bin          | latin1             |  47 |            | Yes         |
(14 rows)
```

说明

m_format_dev_version=s2时，查询结果不包含C，POSIX和default。

6.1.3 collation_character_set_applicability

collation_character_set_applicability视图指示哪种字符集适用于哪种排序规则。这个视图为只读，不允许修改。所有用户对这个视图有“读取”权限。

表 6-4 information_schema.collation_character_set_applicability 字段

名称	类型	描述
COLLATION_NAME	varchar(32)	排序规则名称。
CHARACTER_SET_NAME	varchar(32)	与排序规则关联的字符集的名称。

GUC参数m_format_dev_version控制行为保持前向兼容，m_format_dev_version取值为s2时，可使用最新功能，其他取值保持前向兼容。

m_format_dev_version取值不为s2时，视图仅展示当前数据库的字符集相关的字符集排序规则信息：

```
m_db=# SET m_format_dev_version=s1;
SET
m_db=# SELECT * FROM information_schema.collation_character_set_applicability;
   COLLATION_NAME    | CHARACTER_SET_NAME
---------------------+--------------------
```



```
 utf8mb4_0900_ai_ci | UTF8
 utf8_bin           | UTF8
 utf8_unicode_ci    | UTF8
 utf8_general_ci    | UTF8
 utf8mb4_bin        | UTF8
 utf8mb4_unicode_ci | UTF8
 utf8mb4_general_ci | UTF8
 POSIX              | UTF8
 C                  | UTF8
 default            | UTF8
(10 rows)
```

m_format_dev_version取值为s2时，视图可以查询所有可用排序规则和字符集的对应关系：

```
m_db=# SET m_format_dev_version=s2;
SET
m_db=# SELECT * FROM  information_schema.collation_character_set_applicability;
   COLLATION_NAME   | CHARACTER_SET_NAME
--------------------+--------------------
 binary             | binary
 gbk_chinese_ci     | gbk
 gbk_bin            | gbk
 utf8mb4_general_ci | utf8mb4
 utf8mb4_unicode_ci | utf8mb4
 utf8mb4_bin        | utf8mb4
 utf8_general_ci    | utf8
 utf8_unicode_ci    | utf8
 utf8_bin           | utf8
 utf8mb4_0900_ai_ci | utf8mb4
 gb18030_chinese_ci | gb18030
 gb18030_bin        | gb18030
 latin1_swedish_ci  | latin1
 latin1_bin         | latin1
(14 rows)
```

说明

- m_format_dev_version=s2，查询结果不包含C、POSIX和default。
- utf8和utf8mb4在M-compatibility兼容模式数据库下为同一字符集，所以utf8可以使用utf8mb4的字符序，utf8mb4可以使用utf8的字符序。

6.1.4 columns

columns视图提供有关表中列的信息。相关st_geometry_columns表提供有关存储空间数据的表列的信息。这个视图为只读，不允许修改。所有用户对这个视图有“读取”权限。

表 6-5 information_schema.columns 字段

名称	类型	描述
TABLE_CATALOG	varchar(512)	包含该列的表所属目录（数据库）的名称。该值为当前数据库名。该字段的值在lower_case_table_names为0时大小写敏感，在lower_case_table_names为1时大小写不敏感。
TABLE_SCHEMA	varchar(64)	包含该列的表所属的Schema名称。该字段的值在lower_case_table_names为0时大小写敏感，在lower_case_table_names为1时大小写不敏感。

名称	类型	描述
TABLE_NAME	varchar(64)	包含该列的表的名称。该字段的值在lower_case_table_names为0时大小写敏感，在lower_case_table_names为1时大小写不敏感。
COLUMN_NAME	varchar(64)	列的名称。
ORDINAL_POSITION	bigint unsigned	列在表中的位置。
COLUMN_DEFAULT	longtext	列的默认值。
IS_NULLABLE	varchar(3)	列的可空性。
DATA_TYPE	varchar(64)	列数据类型。
CHARACTER_MAXIMUM_LENGTH	bigint unsigned	对于字符串列，以字符为单位的最大长度。
CHARACTER_OCTET_LENGTH	bigint unsigned	对于字符串列，最大长度（以字节为单位）。
NUMERIC_PRECISION	bigint unsigned	对于数字列，数字精度。
NUMERIC_SCALE	bigint unsigned	对于数字列，数字刻度。
DATETIME_PRECISION	bigint unsigned	对于时间列，小数秒精度。
CHARACTER_SET_NAME	varchar(32)	对于字符串列，字符集名称。
COLLATION_NAME	varchar(32)	对于字符串列，排序规则名称。
COLUMN_TYPE	longtext	列数据类型。该column_type值包含类型名称和可能的其他信息，例如精度或长度。
COLUMN_KEY	varchar(3)	该列是否被索引： ● pri：代表是主键，或者是一个多列主键的其中一个。 ● uni：代表是一个唯一索引的第一个列。 ● mul：代表该列是一个非唯一索引的第一个列。 ● 空：代表没有被索引，或者非pri、uni、mul场景。

名称	类型	描述
EXTRA	varchar(30)	有关指定列的任何其他可用信息。在以下情况下，该值是非NULL的： ● auto_increment：用于具有AUTO_INCREMENT属性的列。 ● on update CURRENT_TIMESTAMP：对于时间列，具有on update CURRENT_TIMESTAMP属性的列。 ● STORED GENERATED：对于生成列，具有STORED GENERATED属性的列。 ● VIRTUAL GENERATED：对于生成列，具有VIRTUAL GENERATED属性的列。
PRIVILEGES	varchar(80)	当前用户拥有该列的权限。
COLUMN_COMMENT	varchar(1024)	列定义中包含的注释信息。
GENERATION_EXPRESSION	longtext	对于生成的列，显示用于计算列值的表达式。对于非生成列为空。

说明

当m_format_dev_version的设置不为s2时，查询结果包含当前会话创建的临时表但不包含其他会话创建的临时表；当m_format_dev_version设置为s2时，查询结果既不包含其他会话创建的临时表，也不包含当前会话创建的临时表。

6.1.5 column_privileges

column_privileges视图提供有关列权限的信息。它从m_schema.columns_priv视图中获取其值。这个视图为只读，不允许修改。所有用户对这个视图有“读取”权限。

表 6-6 information_schema.column_privileges 字段

名称	类型	描述
GRANTEE	varchar(81)	授予权限的账户名称。
TABLE_CATALOG	varchar(512)	包含该列的表所属目录（数据库）的名称。
TABLE_SCHEMA	varchar(64)	包含该列的表所属的Schema名称。
TABLE_NAME	varchar(64)	包含该列的表的名称。
COLUMN_NAME	varchar(64)	列的名称。
PRIVILEGE_TYPE	varchar(64)	授予的特权。该值是在列级别授予的任何权限。

名称	类型	描述
IS_GRANTABLE	varchar(3)	如果用户有GRANT OPTION权限为yes，否则为no。

6.1.6 engines

engines视图提供了有关存储引擎的信息。用于查看是否支持存储引擎或查看默认引擎。这个视图为只读，不允许修改。所有用户对这个视图有“读取”权限。

表 6-7 information_schema.engines 字段

名称	类型	描述
ENGINE	varchar(64)	引擎的名称。 返回值：Ustore和Astore。
SUPPORT	varchar(8)	服务器对存储引擎的支持级别。 返回值：YES和YES。
COMMENT	varchar(80)	引擎的简要说明。 返回值：In-place update store和Append update store。
TRANSACTIONS	varchar(3)	引擎是否支持事务。 返回值：YES和YES。
XA	varchar(3)	引擎是否支持XA事务。 返回值：YES和YES。
SAVEPOINTS	varchar(3)	引擎是否支持保存点。 返回值：YES和YES。

6.1.7 events

events视图显示GaussDB中用户创建的定时任务信息。这个视图为只读，不允许修改。所有用户对这个视图有“读取”权限。用户只能看到自己有访问权限的记录（Schema）。

表 6-8 information_schema.events 字段

名称	类型	描述
EVENT_CATALOG	varchar(64)	标识作业要在哪一个数据库执行的数据库名称。该字段的值在lower_case_table_names为0时大小写敏感，在lower_case_table_names为1时大小写不敏感。

名称	类型	描述
EVENT_SCHEMA	varchar(64)	标识作业执行时的Schema名称。该字段的值在lower_case_table_names为0时大小写敏感，在lower_case_table_names为1时大小写不敏感。
EVENT_NAME	varchar(64)	定时任务或程序名称。
DEFINER	varchar(93)	创建者的用户名。
TIME_ZONE	varchar(64)	作业调度执行的时区设置，值总是设为system。
EVENT_BODY	varchar(8)	该字段当前版本暂不支持，置null。
EVENT_DEFINITION	longtext	作业内容，定时任务中的程序内容。
EVENT_TYPE	varchar(9)	作业类型，该字段当前版本暂不支持，置null。
EXECUTE_AT	datetime	单次执行类作业开始时间。
INTERVAL_VALUE	varchar(256)	重复执行类作业执行时间间隔。
INTERVAL_FIELD	varchar(18)	该字段当前版本暂不支持，置null。
SQL_MODE	varchar(8192)	创建或更改作业时生效的SQL模式，该字段当前版本暂不支持，置null。
STARTS	datetime	重复执行类作业开始时间。
ENDS	datetime	重复执行类作业结束时间。
STATUS	varchar(18)	当前作业的执行状态，取值范围：('r', 's', 'f', 'd')，默认为's'。 取值含义： ● r=running。 ● s=successfully finished。 ● f=job failed。 ● d=disabled。
ON_COMPLETION	varchar(12)	该字段当前版本暂不支持，置null。
CREATED	datetime	作业创建时间，该字段当前版本暂不支持，置null。
LAST_ALTERED	datetime	作业最近一次被修改的时间，该字段当前版本暂不支持，置null。
LAST_EXECUTED	datetime	作业上次执行的时间。
EVENT_COMMENT	varchar(64)	该字段当前版本暂不支持，置null。
ORIGINATOR	bigint	该字段当前版本暂不支持，置null。

名称	类型	描述
CHARACTER_SET_CLIENT	varchar(32)	该字段当前版本暂不支持，置null。
COLLATION_CONNECTION	varchar(32)	该字段当前版本暂不支持，置null。
DATABASE_COLLATION	varchar(32)	该字段当前版本暂不支持，置null。

6.1.8 files

files视图提供了有关存储表空间数据的文件的信息。这个视图为只读，不允许修改。所有用户对这个视图有“读取”权限。

表 6-9 information_schema.files 字段

名称	类型	描述
FILE_ID	bigint	文件标识符，该字段当前版本暂不支持，置null。
FILE_NAME	varchar(4000)	文件名称，该字段当前版本暂不支持，置null。
FILE_TYPE	varchar(20)	文件类型，该字段当前版本暂不支持，置null。
TABLESPACE_NAME	varchar(64)	与文件关联的表空间名称，该字段当前版本暂不支持，置null。
TABLE_CATALOG	varchar(64)	该字段当前版本暂不支持，置null。
TABLE_SCHEMA	varchar(64)	该字段当前版本暂不支持，置null。
TABLE_NAME	varchar(64)	该字段当前版本暂不支持，置null。
LOGFILE_GROUP_NAME	varchar(64)	该字段当前版本暂不支持，置null。
LOGFILE_GROUP_NUMBER	bigint	该字段当前版本暂不支持，置null。
ENGINE	varchar(64)	存储引擎，该字段当前版本暂不支持，置null。
FULLTEXT_KEYS	varchar(64)	该字段当前版本暂不支持，置null。
DELETED_ROWS	bigint	该字段当前版本暂不支持，置null。
UPDATE_COUNT	bigint	该字段当前版本暂不支持，置null。
FREE_EXTENTS	bigint	该字段当前版本暂不支持，置null。

名称	类型	描述
TOTAL_EXTENTS	bigint	该字段当前版本暂不支持，置null。
EXTENT_SIZE	bigint	该字段当前版本暂不支持，置null。
INITIAL_SIZE	bigint unsigned	该字段当前版本暂不支持，置null。
MAXIMUM_SIZE	bigint unsigned	该字段当前版本暂不支持，置null。
AUTOEXTEND_SIZE	bigint unsigned	该字段当前版本暂不支持，置null。
CREATION_TIME	datetime	该字段当前版本暂不支持，置null。
LAST_UPDATE_TIME	datetime	该字段当前版本暂不支持，置null。
LAST_ACCESS_TIME	datetime	该字段当前版本暂不支持，置null。
RECOVER_TIME	bigint	该字段当前版本暂不支持，置null。
TRANSACTION_COUNTER	bigint	该字段当前版本暂不支持，置null。
VERSION	bigint unsigned	该字段当前版本暂不支持，置null。
ROW_FORMAT	varchar(10)	该字段当前版本暂不支持，置null。
TABLE_ROWS	bigint unsigned	该字段当前版本暂不支持，置null。
AVG_ROW_LENGTH	bigint unsigned	该字段当前版本暂不支持，置null。
DATA_LENGTH	bigint unsigned	该字段当前版本暂不支持，置null。
MAX_DATA_LENGTH	bigint unsigned	该字段当前版本暂不支持，置null。
INDEX_LENGTH	bigint unsigned	该字段当前版本暂不支持，置null。
DATA_FREE	bigint unsigned	该字段当前版本暂不支持，置null。
CREATE_TIME	datetime	该字段当前版本暂不支持，置null。
UPDATE_TIME	datetime	该字段当前版本暂不支持，置null。
CHECK_TIME	datetime	该字段当前版本暂不支持，置null。
CHECKSUM	bigint unsigned	该字段当前版本暂不支持，置null。
STATUS	varchar(20)	该字段当前版本暂不支持，置null。
EXTRA	varchar(255)	该字段当前版本暂不支持，置null。

6.1.9 global_status

global_status视图提供了有关服务器的系统状态信息。这个视图为只读，不允许修改。所有用户对这个视图有“读取”权限。当前仅支持Threads_connected和Uptime，用于展示全局范围内的数据库连接数和运行时间。

表 6-10 information_schema.global_status 字段

名称	类型	描述
VARIABLE_NAME	varchar(64)	服务器系统状态信息的变量名称。
VARIABLE_VALUE	varchar(1024)	服务器系统状态信息的变量值。

6.1.10 global_variables

global_variables视图提供了全局变量信息。这个视图为只读，不允许修改。所有用户对这个视图有“读取”权限。

表 6-11 information_schema.global_variables 字段

名称	类型	描述
VARIABLE_NAME	varchar(64)	变量名称。
VARIABLE_VALUE	varchar(1024)	变量值。

6.1.11 key_column_usage

key_column_usage视图描述了哪些键列具有约束。由于功能键部分的信息是表达式，因此该视图不提供相关内容，该视图仅提供有关列的信息。这个视图为只读，不允许修改。所有用户对这个视图有“读取”权限。

约束：该视图对于表达式约束（包括组合表达式约束）仅显示一条数据，且不显示列名。

表 6-12 information_schema.key_column_usage 字段

名称	类型	描述
CONSTRAINT_CATALOG	varchar(512)	约束所属目录的名称。该值为当前数据库名。该字段的值在lower_case_table_names为0时大小写敏感，在lower_case_table_names为1时大小写不敏感。
CONSTRAINT_SCHEMA	varchar(64)	约束所属的Schema名称。该字段的值在lower_case_table_names为0时大小写敏感，在lower_case_table_names为1时大小写不敏感。
CONSTRAINT_NAME	varchar(64)	约束的名称。

名称	类型	描述
TABLE_CATALOG	varchar(512)	表所属目录的名称。该值为当前数据库名。该字段的值在lower_case_table_names为0时大小写敏感，在lower_case_table_names为1时大小写不敏感。
TABLE_SCHEMA	varchar(64)	表所属的模式（数据库）的名称。该字段的值在lower_case_table_names为0时大小写敏感，在lower_case_table_names为1时大小写不敏感。
TABLE_NAME	varchar(64)	具有约束的表的名称。该字段的值在lower_case_table_names为0时大小写敏感，在lower_case_table_names为1时大小写不敏感。
COLUMN_NAME	varchar(64)	具有约束的列的名称。如果约束是外键，那么这是外键的列，而不是外键引用的列。
ORDINAL_POSITION	bigint	列在约束中的位置。从1开始编号。
POSITION_IN_UNIQUE_CONSTRAINT	bigint	NULL用于唯一和主键约束。对于外键约束，此列是被引用表的键中的序号位置。
REFERENCED_TABLE_SCHEMA	varchar(64)	约束引用的schema名称。该字段的值在lower_case_table_names为0时大小写敏感，在lower_case_table_names为1时大小写不敏感。
REFERENCED_TABLE_NAME	varchar(64)	约束引用的表的名称。该字段的值在lower_case_table_names为0时大小写敏感，在lower_case_table_names为1时大小写不敏感。
REFERENCED_COLUMN_NAME	varchar(64)	约束引用的列的名称。

6.1.12 optimizer_trace

optimizer_trace视图提供优化器跟踪功能为跟踪语句生成的信息。这个视图为只读，不允许修改。所有用户对这个视图有“读取”权限。

表 6-13 information_schema.optimizer_trace 字段

名称	类型	描述
QUERY	longtext	暂不支持，置null。
TRACE	longtext	暂不支持，置null。
MISSING_BYTES_BEYOND_MAX_MEM_SIZE	integer	暂不支持，置null。
INSUFFICIENT_PRIVILEGES	tinyint	暂不支持，置null。

6.1.13 parameters

parameters视图提供有关存储例程（存储过程和存储函数）的参数，以及有关存储函数的返回值的信息。该视图不包括内置函数或可加载函数。这个视图为只读，不允许修改。所有用户对这个视图有“读取”权限。

表 6-14 information_schema.parameters 字段

名称	类型	描述
SPECIFIC_CATALOG	varchar(512)	包含参数的例程所属目录的名称。该值为当前数据库名。该字段的值在lower_case_table_names为0时大小写敏感，在lower_case_table_names为1时大小写不敏感。
SPECIFIC_SCHEMA	varchar(64)	包含参数的例程所属的Schema名称。该字段的值在lower_case_table_names为0时大小写敏感，在lower_case_table_names为1时大小写不敏感。
SPECIFIC_NAME	varchar(64)	包含参数的例程的名称。
ORDINAL_POSITION	integer	存储过程或函数的参数排序，ordinal_position值为 1、2、3等。
PARAMETER_MODE	varchar(5)	参数的模式。该值是in、out或inout。对于存储的函数返回值，此值为null。
PARAMETER_NAME	varchar(64)	参数的名称。对于存储的函数返回值，此值为null。
DATA_TYPE	varchar(64)	参数数据类型。
CHARACTER_MAXIMUM_LENGTH	integer	对于字符串参数，以字符为单位的最大长度。
CHARACTER_OCTET_LENGTH	integer	对于字符串参数，以字节为单位的最大长度。

名称	类型	描述
NUMERIC_PRECISION	bigint	对于数字参数，表示数字精度。
NUMERIC_SCALE	integer	对于数字参数，表示数字标度。
DATETIME_PRECISION	bigint	对于时间参数，表示小数秒精度。
CHARACTER_SET_NAME	varchar(64)	对于字符串参数，表示字符集名称。
COLLATION_NAME	varchar(64)	对于字符串参数，表示排序规则名称。
DTD_IDENTIFIER	longtext	参数数据类型。
ROUTINE_TYPE	varchar(9)	• 存储过程：procedure。 • 存储函数：function。

6.1.14 partitions

显示GaussDB中分区表（partitioned table）和分区（table partition）信息，一级分区和二级分区分开记录。这个视图为只读，不允许修改。所有用户对这个视图有“读取”权限。 用户只能看到自己有访问权限的记录。由于视图中部分信息基于统计信息获取，执行ANALYZE后再查看（如果数据库中更新数据，建议延迟执行ANALYZE）。

表 6-15 information_schema.partitions 字段

名称	类型	描述
TABLE_CATALOG	varchar(512)	所在catalog名称。该字段的值在lower_case_table_names为0时大小写敏感，在lower_case_table_names为1时大小写不敏感。
TABLE_SCHEMA	varchar(64)	Schema名称。该字段的值在lower_case_table_names为0时大小写敏感，在lower_case_table_names为1时大小写不敏感。
TABLE_NAME	varchar(64)	表名。该字段的值在lower_case_table_names为0时大小写敏感，在lower_case_table_names为1时大小写不敏感。
PARTITION_NAME	varchar(64)	分区名称。
SUBPARTITION_NAME	varchar(64)	子分区名称。如果不是子分区，则为null。

名称	类型	描述
PARTITION_ORDINAL_POSITION	bigint unsigned	分区在表中的位置。
SUBPARTITION_ORDINAL_POSITION	bigint unsigned	子分区在分区中的位置。如果不是子分区，则为null。
PARTITION_METHOD	varchar(18)	分区策略。如果分区不是一级分区，则为null。 ● 'r'：范围分区。 ● 'l'：list分区。 ● 'h'：hash分区。
SUBPARTITION_METHOD	varchar(12)	子分区策略。如果分区不是二级分区，则为null。 ● 'r'：范围分区。 ● 'l'：list分区。 ● 'h'：hash分区。
PARTITION_EXPRESSION	longtext	暂不支持，值为null。
SUBPARTITION_EXPRESSION	longtext	暂不支持，值为null。
PARTITION_DESCRIPTION	longtext	● 对于范围分区，显示各分区的上边界值。 ● 对于列表分区，显示各分区的取值列表。 ● 对于哈希分区，显示各分区的编号。
TABLE_ROWS	bigint unsigned	分区中的记录数。
AVG_ROW_LENGTH	bigint unsigned	暂不支持，值为null。
DATA_LENGTH	bigint unsigned	暂不支持，值为null。
MAX_DATA_LENGTH	bigint unsigned	暂不支持，值为null。
INDEX_LENGTH	bigint unsigned	暂不支持，值为null。
DATA_FREE	bigint unsigned	暂不支持，值为null。
CREATE_TIME	datetime	暂不支持，值为null。
UPDATE_TIME	datetime	暂不支持，值为null。
CHECK_TIME	datetime	暂不支持，值为null。
CHECKSUM	bigint unsigned	暂不支持，值为null。

名称	类型	描述
PARTITION_COMMENT	varchar(80)	暂不支持，值为null。
NODEGROUP	varchar(12)	暂不支持，置null。
TABLESPACE_NAME	varchar(64)	分区所属表空间的名称。

6.1.15 plugins

plugins视图提供有关服务器插件的信息。

表 6-16 information_schema.plugins 字段

名称	类型	描述
PLUGIN_NAME	varchar(64)	用于在语句中引用插件的名称，例如install plugin和uninstall plugin。
PLUGIN_VERSION	varchar(20)	来自插件的通用类型描述符的版本。
PLUGIN_STATUS	varchar(10)	插件状态： ● active：已安装。 ● inactive：未安装。
PLUGIN_TYPE	varchar(80)	该字段当前版本暂不支持，置null。
PLUGIN_TYPE_VERSION	varchar(20)	该字段当前版本暂不支持，置null。
PLUGIN_LIBRARY	varchar(64)	该字段当前版本暂不支持，置null。
PLUGIN_LIBRARY_VERSION	varchar(20)	该字段当前版本暂不支持，置null。
PLUGIN_AUTHOR	varchar(64)	该字段当前版本暂不支持，置null。
PLUGIN_DESCRIPTION	longtext	插件的描述。
PLUGIN_LICENSE	varchar(80)	该字段当前版本暂不支持，置null。
LOAD_OPTION	varchar(64)	该字段当前版本暂不支持，置null。

6.1.16 processlist

processlist视图显示当前由服务器内执行的一组线程执行的操作过程信息。这个视图为只读，不允许修改。所有用户对这个视图有“读取”权限。

表 6-17 information_schema.processlist 字段

名称	类型	描述
ID	bigint	线程标识符。当GUC参数enable_thread_pool为on时，如果后台处于空闲状态，则值为0。
USER	varchar(32)	发出语句的用户。
HOST	varchar(64)	发出语句的客户端的主机名和端口号（当通过远程客户端登录时才会显示，其他情况下值为NULL）。
DB	varchar(64)	显示当前M-Compatibility数据库的名称。
COMMAND	varchar(16)	代表客户端执行的命令类型，若空闲则为sleep。该字段当前版本暂不支持，值为NULL。
TIME	integer	线程处于其当前状态的时间（以秒为单位）。
STATE	varchar(64)	指示线程正在做的操作、事件或状态。该字段当前版本暂不支持，值为NULL。
INFO	longtext	线程正在执行的语句，如果没有执行任何语句则为NULL。

6.1.17 profiling

profiling视图提供语句分析信息。这个视图为只读，不允许修改。所有用户对这个视图有“读取”权限。当前版本该视图未实现，所有值均为NULL。

表 6-18 information_schema.profiling 字段

名称	类型	描述
QUERY_ID	integer	语句标识符。
SEQ	integer	序列号，指示具有相同query_id值的行的显示顺序。
STATE	varchar(30)	指各个分析步骤所属的状态。
DURATION	decimal(9,6)	语句在给定状态下保持多长时间，以秒为单位。

名称	类型	描述
CPU_USER	decimal(9,6)	用户CPU使用情况，以秒为单位。
CPU_SYSTEM	decimal(9,6)	系统CPU使用情况，以秒为单位。
CONTEXT_VOLUNTARY	integer	自愿的上下文切换次数。
CONTEXT_INVOLUNTARY	integer	非自愿的上下文切换次数。
BLOCK_OPS_IN	integer	块输入操作的数量。
BLOCK_OPS_OUT	integer	块输出操作的数量。
MESSAGES_SENT	integer	发送的通信消息数。
MESSAGES_RECEIVED	integer	接收的通信消息数。
PAGE_FAULTS_MAJOR	integer	主要页面错误的数量。
PAGE_FAULTS_MINOR	integer	次要页面错误的数量。
SWAPS	integer	交换次数。
SOURCE_FUNCTION	varchar(30)	分析状态在源代码中执行位置的信息。
SOURCE_FILE	varchar(20)	分析状态在源代码中执行位置的信息。
SOURCE_LINE	integer	分析状态在源代码中执行位置的信息。

6.1.18 referential_constraints

referential_constraints视图提供有关外键的信息。这个视图为只读，不允许修改。所有用户对这个视图有“读取”权限。

表 6-19 information_schema.referential_constraints 字段

名称	类型	描述
CONSTRAINT_CATALOG	varchar(512)	约束所属数据库的名称。该值为约束所属的数据库。该字段的值在lower_case_table_names为0时大小写敏感，在lower_case_table_names为1时大小写不敏感。

名称	类型	描述
CONSTRAINT_SCHEMA	varchar(64)	约束所属的Schema名称。该字段的值在lower_case_table_names为0时大小写敏感，在lower_case_table_names为1时大小写不敏感。
CONSTRAINT_NAME	varchar(64)	约束的名称。
UNIQUE_CONSTRAINT_CATALOG	varchar(512)	约束引用的唯一约束的数据库的名称。该值为唯一约束所属的数据库。该字段的值在lower_case_table_names为0时大小写敏感，在lower_case_table_names为1时大小写不敏感。
UNIQUE_CONSTRAINT_SCHEMA	varchar(64)	约束引用的唯一约束的Schema的名称。该字段的值在lower_case_table_names为0时大小写敏感，在lower_case_table_names为1时大小写不敏感。
UNIQUE_CONSTRAINT_NAME	varchar(64)	约束引用的唯一约束的名称。
MATCH_OPTION	varchar(64)	约束match属性的值。
UPDATE_RULE	varchar(64)	约束on update属性的值。可能的值为cascade、set null、set default、restrict、no action。
DELETE_RULE	varchar(64)	约束on delete属性的值。可能的值为cascade、set null、set default、restrict、no action。
TABLE_NAME	varchar(64)	约束所属表的名称。该字段的值在lower_case_table_names为0时大小写敏感，在lower_case_table_names为1时大小写不敏感。
REFERENCED_TABLE_NAME	varchar(64)	约束引用的表的名称。该字段的值在lower_case_table_names为0时大小写敏感，在lower_case_table_names为1时大小写不敏感。

6.1.19 routines

routines视图提供有关存储例程（存储过程和存储函数）的信息。这个视图为只读，不允许修改。所有用户对这个视图有“读取”权限。

表 6-20 information_schema.routines 字段

名称	类型	描述
SPECIFIC_NAME	varchar(64)	例程的名称。

名称	类型	描述
ROUTINE_CATALOG	varchar(512)	例程所属目录的名称。该字段的值在lower_case_table_names为0时大小写敏感，在lower_case_table_names为1时大小写不敏感。
ROUTINE_SCHEMA	varchar(64)	例程所属的Schema名称。该字段的值在lower_case_table_names为0时大小写敏感，在lower_case_table_names为1时大小写不敏感。
ROUTINE_NAME	varchar(64)	例程的名称。
ROUTINE_TYPE	varchar(9)	• 存储过程：procedure。 • 存储函数：function。
DATA_TYPE	varchar(64)	如果例程是存储函数，返回值数据类型。如果例程是存储过程，则此值为void。
CHARACTER_MAXIMUM_LENGTH	integer	对于存储的函数字符串返回值，最大字符长度。如果例程是存储过程，则此值为null。
CHARACTER_OCTET_LENGTH	integer	对于存储的函数字符串返回值，以字节为单位的最大长度。如果例程是存储过程，则此值为null。
NUMERIC_PRECISION	bigint unsigned	对于存储函数数字返回值，数字精度。如果例程是存储过程，则此值为null。
NUMERIC_SCALE	integer	对于存储的函数数字返回值，数字比例。如果例程是存储过程，则此值为null。
DATETIME_PRECISION	bigint unsigned	对于存储函数时间返回值，小数秒精度。如果例程是存储过程，则此值为null。
CHARACTER_SET_NAME	varchar(64)	对于存储的函数字符串返回值，字符集名称。如果例程是存储过程，则此值为null。
COLLATION_NAME	varchar(64)	对于存储的函数字符串返回值，排序规则名称。如果例程是存储过程，则此值为null。
DTD_IDENTIFIER	longtext	如果例程是存储函数，返回值数据类型。如果例程是存储过程，则此值为void。
ROUTINE_BODY	varchar(8	用于例程定义的语言。该值始终为SQL。
ROUTINE_DEFINITION	longtext	例程执行的SQL语句的文本。
EXTERNAL_NAME	varchar(64)	该值始终为null。
EXTERNAL_LANGUAGE	varchar(64)	存储例程的语言。

名称	类型	描述
PARAMETER_STYLE	varchar(8)	该值始终为SQL。
IS_DETERMINISTIC	varchar(3)	yes或no，取决于例程是否定义有deterministic特征。
SQL_DATA_ACCESS	varchar(64)	例程的数据访问特征。
SQL_PATH	varchar(64)	该值始终为null。
SECURITY_TYPE	varchar(7)	视图查询数据时的安全验证方式。该值为definer或者invoker。
CREATED	datetime	创建例程的日期和时间，置null。
LAST_ALTERED	datetime	上次修改例程的日期和时间，置null。
SQL_MODE	varchar(8192)	创建或更改例程时有效的SQL模式，置null。
ROUTINE_COMMENT	longtext	存储程序的注释信息，置null。
DEFINER	varchar(93)	definer子句中指定的账户（通常是创建例程的用户），置null。
CHARACTER_SET_CLIENT	varchar(32)	创建例程时系统变量的会话值，置null。
COLLATION_CONNECTION	varchar(32)	创建例程时系统变量的会话值，置null。
DATABASE_COLLATION	varchar(32)	与例程关联的数据库的排序规则，置null。

6.1.20 schemata

schemata视图提供有关数据库的信息。这个视图为只读，不允许修改。所有用户对这个视图有“读取”权限。

表 6-21 information_schema.schemata 字段

名称	类型	描述
CATALOG_NAME	varchar（512）	数据库名称。该字段的值在lower_case_table_names为0时大小写敏感，在lower_case_table_names为1时大小写不敏感。

名称	类型	描述
SCHEMA_NAME	varchar（64）	Schema的名称。该字段的值在lower_case_table_names为0时大小写敏感，在lower_case_table_names为1时大小写不敏感。
DEFAULT_CHARACTER_SET_NAME	varchar（32）	Schema默认字符集名称。
DEFAULT_COLLATION_NAME	varchar（32）	Schema默认排序规则。
SQL_PATH	varchar（32）	该值始终为null。

6.1.21 schema_privileges

显示用户在GaussDB中对Schema所拥有的权限。该视图为只读，不允许修改。所有用户对这个视图有“读取”权限。

表 6-22 information_schema.schema_privileges 字段

名称	类型	描述
GRANTEE	varchar(81)	被授予权限的用户或角色的名称。
TABLE_CATALOG	varchar(512)	对象所在catalog名称。
TABLE_SCHEMA	varchar(64)	对象的Schema名称。
PRIVILEGE_TYPE	varchar(64)	被授予的权限。
IS_GRANTABLE	varchar(3)	表示是否授予特权： • 是：yes。 • 否：no。

6.1.22 session_status

session_status视图显示当前连接的服务器状态信息。该视图为只读，不允许修改。所有用户对这个视图有“读取”权限。当前仅支持Threads_connected和Uptime，用于展示会话范围内的数据库连接数和运行时间。

表 6-23 information_schema.session_status 字段

名称	类型	描述
VARIABLE_NAME	varchar(64)	当前会话的服务器系统状态信息的变量名称。
VARIABLE_VALUE	varchar(1024)	当前会话的服务器系统状态信息的变量值。

6.1.23 session_variables

session_variables视图显示对当前连接有效的系统变量值。该视图为只读，不允许修改。所有用户对这个视图有“读取”权限。

表 6-24 information_schema.session_variables 字段

名称	类型	描述
VARIABLE_NAME	varchar(64)	变量名称。
VARIABLE_VALUE	varchar(1024)	变量值。

6.1.24 statistics

statistics视图提供表索引完整列相关列信息。该视图为只读，不允许修改。所有用户对这个视图有“读取”权限。由于视图中部分信息基于统计信息获取，执行analyze后再查看（如果数据库中更新数据，建议延迟执行analyze）。如果索引列不是表中的完整列，不在该视图中进行记录。

表 6-25 information_schema.statistics 字段

名称	类型	描述
TABLE_CATALOG	varchar(512)	数据库名称。该字段的值在lower_case_table_names为0时大小写敏感，在lower_case_table_names为1时大小写不敏感。
TABLE_SCHEMA	varchar(64)	索引关联的表所在schema名称。该字段的值在lower_case_table_names为0时大小写敏感，在lower_case_table_names为1时大小写不敏感。
TABLE_NAME	varchar(64)	索引关联的表名称。该字段的值在lower_case_table_names为0时大小写敏感，在lower_case_table_names为1时大小写不敏感。

名称	类型	描述
NON_UNIQUE	bigint	是否是非唯一索引。如果是非唯一索引，则为1，如果是唯一索引，则为0。
INDEX_SCHEMA	varchar(64)	索引所属的模式的名称。该字段的值在lower_case_table_names为0时大小写敏感，在lower_case_table_names为1时大小写不敏感。
INDEX_NAME	varchar(64)	索引的名称。如果索引是主键，则名称始终是primary，否则为索引名称。
SEQ_IN_INDEX	bigint	索引列在索引中的序列号。
COLUMN_NAME	varchar(64)	索引列名称。
COLLATION	varchar(1)	列在索引中的排序方式： A（升序）、D（降序）。
CARDINALITY	bigint	索引中唯一值数值的数量的估计值。
SUB_PART	bigint	索引前缀。如果列仅被部分索引，则为索引的字符数；如果整列都被索引，则为NULL。
PACKED	varchar(10)	该字段当前版本暂不支持，置null。
NULLABLE	varchar(3)	索引列是否可以存在null。 yes(可以存在null), ''(不可以存在null)。
INDEX_TYPE	varchar(16)	索引方法（如：BTREE、UBTREE)。
COMMENT	varchar(16)	该字段当前版本暂不支持，置null。
INDEX_COMMENT	varchar(1024)	索引注释 。

执行analyze更新统计信息后再查看视图：

```
my_db=# create table t1(a int);
CREATE TABLE
my_db=# insert into t1 values (1),(2);
INSERT 0 2
my_db=# create index t1_index_1 on t1(a);
CREATE INDEX
my_db=# analyze;
ANALYZE
my_db=# select table_catalog,table_name,index_name,index_type, cardinality from
information_schema.statistics where table_name='t1';
 table_catalog | table_name | index_name | index_type | cardinality
---------------+------------+------------+------------+-------------
 my_db         | t1         | t1_index_1 | UBTREE     |           2
(1 row)
```


6.1.25 tables

该视图提供有关数据库中表的信息。这个视图为只读，不允许修改。所有用户对这个视图有“读取”权限。由于视图中部分信息基于统计信息获取，执行analyze后再查看（如果数据库中更新数据，建议延迟执行analyze）。

表 6-26 information_schema.tables 字段

名称	类型	描述
TABLE_CATALOG	varchar(512)	数据库名称。该字段的值在lower_case_table_names为0时大小写敏感，在lower_case_table_names为1时大小写不敏感。
TABLE_SCHEMA	varchar(64)	表所在的schema名称。该字段的值在lower_case_table_names为0时大小写敏感，在lower_case_table_names为1时大小写不敏感。
TABLE_NAME	varchar(64)	表的名称。该字段的值在lower_case_table_names为0时大小写敏感，在lower_case_table_names为1时大小写不敏感。
TABLE_TYPE	varchar(64)	表类型。BASE TABLE表示表，VIEW表示视图，SYSTEM VIEW表示information_schema下的表。information_schema下的视图和表都为SYSTEM VIEW。
ENGINE	varchar(64)	表的存储引擎。
VERSION	bigint unsigned	该字段当前版本暂不支持，默认为null。
ROW_FORMAT	varchar(10)	该字段当前版本暂不支持，默认为null。
TABLE_ROWS	bigint unsigned	如果表指定的是视图，则为null，否则为活元组数量。
AVG_ROW_LENGTH	bigint unsigned	如果表中没有活元组和死元组，则为null，否则计算表平均行长度。
DATA_LENGTH	bigint unsigned	如果表指定的是视图，则为null，否则计算数据文件长度。
MAX_DATA_LENGTH	bigint unsigned	该字段当前版本暂不支持，默认为null。
INDEX_LENGTH	bigint unsigned	如果表指定的是视图，则为null，否则为索引文件大小。
DATA_FREE	bigint unsigned	如果表中没有活元组和死元组，则为null，否则为已分配但未使用的字节数。
AUTO_INCREMENT	bigint unsigned	下一次自增值。不设置则为null，否则为具体值。

名称	类型	描述
CREATE_TIME	datetime	创建表时间。对于创建视图的情形，显示实际的创建表时间。 M-Compatibility模式数据库自带的表、视图设置null。
UPDATE_TIME	datetime	上一次表更新数据的时间。 M-Compatibility模式数据库自带的表、视图设置null。
CHECK_TIME	datetime	该字段当前版本暂不支持，默认为null。
TABLE_COLLATION	varchar(32)	表排序规则。如果表指定的是视图，则为null。如果指定的表在创建时，未使用COLLATE子句指定列的排序规则，则为null。
CHECKSUM	bigint unsigned	该字段当前版本暂不支持，默认为null。
CREATE_OPTIONS	varhcar(255)	创建表的选项。
TABLE_COMMENT	varchar(2048)	表注释。

执行ANALYZE更新统计信息后再查看视图：

```
m_db=#  CREATE TABLE t1(a int);
CREATE TABLE
m_db=# INSERT INTO t1 VALUES (1),(2);
INSERT 0 2
m_db=# ANALYZE;
ANALYZE
m_db=# SELECT TABLE_CATALOG,TABLE_SCHEMA,TABLE_NAME,TABLE_TYPE,TABLE_ROWS,ENGINE FROM
information_schema.tables WHERE TABLE_NAME='t1';
 TABLE_CATALOG | TABLE_SCHEMA | TABLE_NAME | TABLE_TYPE | TABLE_ROWS | ENGINE
---------------+--------------+------------+------------+------------+--------
 m_db          | public       | t1         | BASE TABLE | 2          | Ustore
(1 row)
m_db=# DROP TABLE t1;
DROP TABLE
```

6.1.26 table_constraints

table_constraints视图描述了哪些表具有约束。该视图为只读，不允许修改。所有用户对这个视图有“读取”权限。

表 6-27 information_schema.table_constraints 字段

名称	类型	描述
CONSTRAINT_CATALOG	varchar(512)	约束所属数据库的名称。该字段的值在lower_case_table_names为0时大小写敏感，在lower_case_table_names为1时大小写不敏感。
CONSTRAINT_SCHEMA	varchar(64)	约束所属的Schema名称。该字段的值在lower_case_table_names为0时大小写敏感，在lower_case_table_names为1时大小写不敏感。
CONSTRAINT_NAME	varchar(64)	约束的名称。如果是主键约束，则显示为PRIMARY。
TABLE_SCHEMA	varchar(64)	表所属的Schema的名称。该字段的值在lower_case_table_names为0时大小写敏感，在lower_case_table_names为1时大小写不敏感。
TABLE_NAME	varchar(64)	表的名称。该字段的值在lower_case_table_names为0时大小写敏感，在lower_case_table_names为1时大小写不敏感。
CONSTRAINT_TYPE	varchar(64)	约束的类型。取值：unique、primary key、 foreign key、check。

6.1.27 table_privileges

table_privileges视图描述了表的权限。该视图为只读，不允许修改。所有用户对这个视图有“读取”权限。

表 6-28 information_schema.table_privileges 字段

名称	类型	描述
GRANTEE	varchar(81)	授予权限的账户的名称。
TABLE_CATALOG	varchar(512)	表所属目录的名称。
TABLE_SCHEMA	varchar(64)	表所属的Schema名称。
TABLE_NAME	varchar(64)	表的名称。
PRIVILEGE_TYPE	varchar(64)	授予的特权。该值可以是可以在表级别授予的任何权限。
IS_GRANTABLE	varchar(3)	如果用户有GRANT OPTION权限，则为yes，否则为no。

6.1.28 triggers

triggers视图提供了有关触发器的信息。该视图为只读，不允许修改。所有用户对这个视图有“读取”权限。

表 6-29 information_schema.triggers 字段

名称	类型	描述
TRIGGER_CATALOG	varchar(512)	视图所属目录的名称。该字段的值在lower_case_table_names为0时大小写敏感，在lower_case_table_names为1时大小写不敏感。
TRIGGER_SCHEMA	varchar(64)	触发器所属的Schema名称。该字段的值在lower_case_table_names为0时大小写敏感，在lower_case_table_names为1时大小写不敏感。
TRIGGER_NAME	varchar(64)	触发器的名称。
EVENT_MANIPULATION	varchar(6)	触发事件。值为INSERT（插入一行）、DELETE（删除一行）或 UPDATE（修改一行）。
EVENT_OBJECT_CATALOG	varchar(512)	触发器关联表所在的目录。该字段的值在lower_case_table_names为0时大小写敏感，在lower_case_table_names为1时大小写不敏感。
EVENT_OBJECT_SCHEMA	varchar(64)	触发器关联表的模式。该字段的值在lower_case_table_names为0时大小写敏感，在lower_case_table_names为1时大小写不敏感。
EVENT_OBJECT_TABLE	varchar(64)	触发器关联表的名称。该字段的值在lower_case_table_names为0时大小写敏感，在lower_case_table_names为1时大小写不敏感。
ACTION_ORDER	bigint	event_manipulation触发器操作在具有相同action_timing值的同一表上的触发器列表中的顺序位置 。该字段当前版本暂不支持，置null。
ACTION_CONDITION	longtext	该值始终为null。
ACTION_STATEMENT	longtext	触发体，即触发器激活时执行的语句。此文本使用UTF8编码。
ACTION_ORIENTATION	varchar(9)	该值始终为row。

名称	类型	描述
ACTION_TIMING	varchar(6)	触发器是在触发事件之前还是之后激活。值为before或 after。
ACTION_REFERENCE_OLD_TABLE	varchar(64)	该值始终为null。
ACTION_REFERENCE_NEW_TABLE	varchar(64)	该值始终为null。
ACTION_REFERENCE_OLD_ROW	varchar(3)	旧列标识符。该值始终为old。
ACTION_REFERENCE_NEW_ROW	varchar(3)	新列标识符。该值始终为new。
CREATED	datetime	创建触发器的日期和时间。这是timestamp(2)触发器的一个值（以百分之一秒为单位的小数部分）。该字段当前版本暂不支持，置null。
SQL_MODE	varchar(8192)	创建触发器时有效的SQL模式，触发器在该模式下执行。该字段当前版本暂不支持，置null。
DEFINER	varchar(93)	创建触发器的用户。该字段当前版本暂不支持，置null。
CHARACTER_SET_CLIENT	varchar(32)	创建触发器时character_set_client系统变量的会话值。该字段当前版本暂不支持，置null。
COLLATION_CONNECTION	varchar(32)	创建触发器时collation_connection系统变量的会话值。该字段当前版本暂不支持，置null。
DATABASE_COLLATION	varchar(32)	与触发器关联的数据库的排序规则。该字段当前版本暂不支持，置null。

6.1.29 user_privileges

user_privileges视图显示用户在GaussDB中对表、视图、Schema等数据库对象所拥有的权限。该视图为只读，不允许修改。所有用户对这个视图有“读取”权限。用户只能看到自己有访问权限的记录。

表 6-30 information_schema.user_privileges 字段

名称	类型	描述
GRANTEE	varchar(81)	被授予权限的用户或角色的名称。
TABLE_CATALOG	varchar(512)	对象所在目录（数据库）的名称。

名称	类型	描述
PRIVILEGE_TYPE	varchar(64)	被授予的权限。
IS_GRANTABLE	varchar(3)	yes表示有授予权限，no表示没有。

6.1.30 views

views视图提供有关数据库中视图的信息。

表 6-31 information_schema.views 字段

名称	类型	描述
TABLE_CATALOG	varchar(512)	视图所属目录（数据库）的名称。该字段的值在lower_case_table_names为0时大小写敏感，在lower_case_table_names为1时大小写不敏感。
TABLE_SCHEMA	varchar(64)	视图所属的Schema名称。该字段的值在lower_case_table_names为0时大小写敏感，在lower_case_table_names为1时大小写不敏感。
TABLE_NAME	varchar(64)	视图的名称。该字段的值在lower_case_table_names为0时大小写敏感，在lower_case_table_names为1时大小写不敏感。
VIEW_DEFINITION	longtext	视图定义的语句。
CHECK_OPTION	varchar(8)	check_option属性的值。该值是NONE、CASCADE或LOCAL中的一个。
IS_UPDATABLE	varchar(3)	视图是否能更新。
DEFINER	varchar(93)	该字段当前版本暂不支持，置null。
SECURITY_TYPE	varchar(7)	该字段当前版本暂不支持，置null。
CHARACTER_SET_CLIENT	varchar(32)	该字段当前版本暂不支持，置null。
COLLATION_CONNECTION	varchar(32)	该字段当前版本暂不支持，置null。

6.2 m_schema

新 增的m_schema schema封装了新增的table_priv、columns_priv、procs_priv、proc、和func视图。

6.2.1 tables_priv

m_schema.tables_priv视图显示GaussDB用户对表拥有的权限。

表 6-32 m_schema.tables_priv 字段

名称	类型	描述
Host	char(60)	主机名。
Db	char(64)	数据库（Schema）名。
User	char(32)	用户名。
Table_name	char(64)	表名。
Grantor	char(93)	执行授权的用户名。
Timestamp	timestamp	current_timestamp。 暂不支持，值为null。

说明

- sql_mode参数值包含ansi_quotes时，查询user列时需要添加双引号，例如：
  ```
  SELECT "user" FROM m_schema.tables_priv WHERE "user" = 'PUBLIC';
  ```
- sql_mode参数值不包含ansi_quotes时，查询user列时要添加反引号，例如：
  ```
  SELECT `user` FROM m_schema.tables_priv WHERE `user` = 'PUBLIC';
  ```
- 本版本M-Compatibility中，本视图中的Table_priv和Column_priv字段暂不予显示。

6.2.2 columns_priv

m_schema.columns_priv视图显示GaussDB用户对字段所拥有的权限信息。

表 6-33 m_schema.columns_priv 字段

名称	类型	描述
Host	char(60)	主机名。
Db	char(64)	数据库（Schema）名。
User	char(32)	用户名。
Table_name	char(64)	表名。
Column_name	char(64)	列名。
Timestamp	timestamp	current_timestamp。 暂不支持，值为null。

说明

- sql_mode参数值包含ansi_quotes时，查询user列时需要添加双引号，例如：
  ```
  SELECT "user" FROM m_schema.tables_priv WHERE "user" = 'PUBLIC';
  ```
- sql_mode参数值不包含ansi_quotes时，查询user列时要添加反引号，例如：
  ```
  SELECT `user` FROM m_schema.tables_priv WHERE `user` = 'PUBLIC';
  ```
- 本版本M-Compatibility模式数据库中，本视图中的Column_priv字段暂不予显示。

6.2.3 procs_priv

m_schema.procs_priv视图显示GaussDB用户对存储过程和函数拥有的权限信息。

这个视图为只读，不允许修改。所有用户对这个视图都有“读取”权限。用户只能看到自己有访问权限的记录。

表 6-34 m_schema.procs_priv 字段

名称	类型	描述
Host	char(60)	主机名。
Db	char(64)	数据库（Schema）名。
User	char(32)	用户名。
Routine_name	char(64)	routine名称。
Grantor	char(93)	执行授权的用户名。
Timestamp	timestamp	current_timestamp。 暂不支持，值为null。

说明

- sql_mode参数值包含ansi_quotes时，查询user列时需要添加双引号，例如：
  ```
  SELECT "user" FROM m_schema.tables_priv WHERE "user" = 'PUBLIC';
  ```
- sql_mode参数值不包含ansi_quotes时，查询user列时要添加反引号，例如：
  ```
  SELECT `user` FROM m_schema.tables_priv WHERE `user` = 'PUBLIC';
  ```
- 本版本M-Compatibility模式数据库中，本视图中的Routine_type和Proc_priv字段暂不予显示。

6.2.4 proc

m_schema.proc视图显示GaussDB存储过程的信息。

这个视图为只读，不允许修改。所有用户对这个视图都有“读取”权限。用户只能看到自己有访问权限的记录。

表 6-35 m_schema.proc 字段

名称	类型	描述
db	char(64)	数据库（Schema）名。
name	char(64)	routine名。
specific_name	char(64)	routine精确名称。
param_list	blob	参数列表。
returns	longblob	routine返回值类型。 array/user_defined等。
body	longblob	routine定义。
definer	char(93)	该值指示哪一个用户定义了该routine。
created	timestamp	创建routine的日期和时间。 暂不支持，值为null。
modified	timestamp	修改routine的日期和时间。 暂不支持，值为null。
comment	text	暂不支持，值为null。
character_set_client	char(32)	创建routine的客户端使用的字符集。
collation_connection	char(32)	创建routine的连接使用的排序规则（和字符集）。 暂不支持，值为null。
db_collation	char(32)	创建routine时数据库的默认排序规则（和字符集）。 暂不支持，值为null。
body_utf8	longblob	routine的定义（utf8）。 暂不支持，值为null。

说明

本版本M-Compatibility模式数据库中，本视图中的type、language、sql_data_access、is_deterministic、security_type和sql_mode字段暂不予显示。

6.2.5 func

m_schema.func视图显示GaussDB函数的信息。

这个视图为只读，不允许修改。所有用户对这个视图有“读取”权限。用户只能看到自己有访问权限的记录。

表 6-36 m_schema.func 字段

名称	类型	描述
name	char(64)	UDF名称。
ret	tinyint	返回值类型。暂不支持，默认值为null。
dl	char(128)	shared library名。

说明

本版本M-Compatibility模式数据库中，本视图中的type字段暂不予显示。

7 存储过程

当前版本不支持，后续支持。