TDSQL 集中式开发手册

版本	作者	内容	时间
V1.0			
V1.1			
V1.2			

1 目录

1 概述 5

1.1 文档说明 5

1.2 范围 5

2 产品术语 5

3 TDSQL 支持的数据类型 5

4 TDSQL 支持的语言结构 8

5 TDSQL 的连接 11

5.1 mysql 命令行方式 11

5.2 JDBC+tomcat 连接配置 11

5.3 dbvisualizer 连接工具配置 12

6 SQL 参考 13

6.1 DDL 语句 13

6.1.1 CREATE 13

6.1.2 ALTER 41

6.1.3 DROP 56

6.1.4 TRUNCATE 58

6.2 DML 语句 59

6.2.1 DELETE 语法 59

6.2.2 INSERT 语法 63

6.2.3 REPLACE 语法 68

6.2.4 SELECT 语法 72

6.2.5 子查询语法 85

6.2.6 UPDATE 语法 89

6.3 事务控制语法 91

6.3.1 START TRANSACTION，COMMIT 和 ROLLBACK 语法 91

6.3.2 无法回滚的陈述 95

6.3.3 导致隐式提交的语句 95

6.3.4 SAVEPOINT，ROLLBACK 到 SAVEPOINT 和 RELEASE SAVEPOINT 语法 97

6.3.5 SET TRANSACTION 语法 97

6.4 Utility 语句 101

6.4.1 DESCRIBE 语句 101

6.4.2 EXPLAIN 语句 102

6.4.3 USE 语句 104

6.5 预处理语句 104

7 支持的字符集和时区 105

8 支持的函数 110

1 概述

1.1 文档说明

本手册涵盖 TDSQL 连接方式，SQL 语句开发编写等内容。目的是指导应用进行开发。

1.2 范围

本手册适用于使用 TDSQL 集中式实例的应用开发人员、数据库应用设计人员、数据库管理员等。

本手册适用于 TDSQL10.3.16.3.x 版本。

2 产品术语

- 节点：Set 或称为数据节点、分片。基于 TDSQL 数据库主从协议联结成若干组。Set 是实例中最小数据单元。
- SQL 引擎：也称为 Proxy 或网关。在 TDSQL 中位于接入层的位置，承接应用数据库请求，负责路由转发、SQL 分析、读写分离等；SQL 引擎无主备之分，本身无状态，一般采用多节点部署分担请求压力。
- 集中式实例：所有数据都在一个 set 上

3 TDSQL 支持的数据类型

TDSQL 集中式数据库支持 MySQL 所有数据类型，包括数字类型、字符类型、日期时间类型、Json 数据类型。

数字类型

集中式实例兼容整型、浮点型和定点型三种数字类型，具体兼容类型如下：

➢ 整型支持 INTEGER、 INT、SMALLINT、TINYINT、MEDIUMINT、BIGINT 七种类型，相关信息详见如下表。

类型	字节数	最小值(有符号/无符号)	最大值(有符号/无符号)

类型	字节数	最小值(有符号/无符号)	最大值(有符号/无符号)
TINYINT	1	-128/0	127/255
SMALLINT	2	-32768/0	32767/65535
MEDIUMINT	3	-8388608/0	8388607/16777215
INT	4	-2147483648/0	2147483647/4294967295
BIGINT	8	-92233720368547 75808/0	9223372036854775807/1844674 4073709551615

- 浮点型支持 FLOAT 和 DOUBLE，格式支持 FLOAT(M,D) 、REAL(M,D) 、 DOUBLE PRECISION(M,D)。
- 定点型支持 DECIMAL 和 NUMERIC，格式 DECIMAL(M,D)。

字符类型

TDSQL 支持的字符类型：CHAR、VARCHAR、BINARY、VARBINARY、BLOB、TEXT、TINYBLOB、TINYTEXT，MEDIUMBLOB、MEDIUMTEXT、LONGBLOB、LONGTEXT、ENUM、SET。

其中 CHAR 和 VARCHAR 最为常用，LOB 和 TEXT 类型不建议使用。

CHAR 和 VARCHAR 类型相似，但存储和检索的方式不同。 它们在最大长度和是否保留尾随空格方面也不同。

CHAR 和 VARCHAR 类型声明的长度指示要存储的最大字符数。 例如，CHAR(30) 最多可容纳 30 个字符。CHAR 列的长度固

定为您在创建表时声明的长度。长度可以是 0 到 255 之间的任何值。存储 CHAR 值时，它们会用空格右填充到指定的长度。

VARCHAR 列中的值是可变长度的字符串。长度可以指定为 0 到 65,535 之间的值。

日期类型

TDSQL 支持如下时间类型：

类型	日期格式	日期范围
YEAR	YYYY	1901 ~ 2155
TIME	HH:MM:SS	-838:59:59 ~ 838:59:59
DATE	YYYY-MM-DD	1000-01-01 ~ 9999-12-3
DATETIME	YYYY-MM-DD HH:MM:SS	1000-01-01 00:00:00 ~ 9999-12-31 23:59:59
TIMESTAMP	YYYY-MM-DD HH:MM:SS	1980-01-01 00:00:01 UTC ~ 2040-01-19 03:14:07 UTC

Json 数据类型

支持存储 Json 格式的数据类型，以便更加有效的对 Json 进行处理，同时又能提早检查错误。

语句如下：

注意事项：对 Json 类型的字段进行排序时，不支持混合类型排序。

例如，不能将 String 类型和 Int 类型做比较，同类型排序只支持数值类型和 String 类型，其它类型排序暂不处理。

```
mysql>  CREATE TABLE t1 (jdoc JSON,a int key);
Query OK, 0 rows affected (0.30 sec)

mysql> INSERT INTO t1 (jdoc,a)VALUES('{"key1":
"value1", "key2": "value2"}',1);
Query OK, 1 row affected (0.00 sec)

mysql> INSERT INTO t1 (jdoc,a)VALUES('{"key1":
"value1", "key2": 2}',2);

mysql> select * from t1;
+--------------------------------------+---+
| jdoc                                 | a |
+--------------------------------------+---+
| {"key1": "value1", "key2": "value2"} | 1 |
| {"key1": "value1", "key2": 2}        | 2 |
+--------------------------------------+---+
2 rows in set (0.00 sec)
```

4 TDSQL 支持的语言结构

集中式实例支持所有 MySQL 使用的文字格式，包括如下：

- String Literals
- Numeric Literals
- Date and Time Literals
- Hexadecimal Literals
- Bit-Value Literals
- Boolean Literals
- NULL Values

String Literals 格式

String Literals 是一个 bytes 或者 characters 的序列，两端被单引号 ' 或者双引号 " 包围，目前 TDSQL 不支持 ANSI_QUOTES SQL MODE，双引号 " 包围的始终认为是 String Literals，而不是 Identifier。

不支持 character set introducer 格式，即：[_charset_name]'string' [COLLATE collation_name]格式。

支持如下转义字符：

```
\0: ASCII NUL (X’00’) 字符
\‘: 单引号
\“: 双引号
\b: 退格符号
\n: 换行符
\r: 回车符
\t: tab 符（制表符）
\z: ASCII 26 (Ctrl + Z)
\\: 反斜杠 \
\%: \%
\_: _
```

Numeric Literals 格式

数值字面值包括 Integer 类型 、 Decimal 类型、浮点数字面值。

Integer 可以包括“ . ”作为小数点分隔，数字前加字符“ -” 、“ +” 来表示正数或者负数。

精确数值字面值可以表示多种格式，如格式：1, .2, 3.4, -5, -6.78, +9.10。

科学记数法，如格式：1.2E3, 1.2E-3, -1.2E3, -1.2E-3。

Date and Time Literals 格式

Date 支持如下格式：

```
'YYYY-MM-DD' or 'YY-MM-DD'
'YYYYMMDD' or 'YYMMDD'
YYYYMMDD or YYMMDD
```

例如：'2012-12-31', '2012/12/31', '2012^12^31', '2012@12@31' '20070523', '070523'

Datetime、Timestamp 支持如下格式：

```
'YYYY-MM-DD HH:MM:SS' or 'YY-MM-DD HH:MM:SS'
'YYYYMMDDHHMMSS' or 'YYMMDDHHMMSS'
YYYYMMDDHHMMSS or YYMMDDHHMMSS
```

例如：'2012-12-31 11:30:45', '2012^12^31 11+30+45', '2012/12/31 11*30*45',

```
'2012@12@31 11^30^45', 19830905132800
```

Hexadecimal Literals 格式

Hexadecimal Literals 支持的格式如下：

```
X'01AF'
X'01af'
x'01AF'
x'01af'
0x01AF
0x01af
```

Bit-Value Literals 格式

Bit-Value Literals 支持的格式如下：

```
b'01'
B'01'
0b01
```

Boolean Literals 格式

常量 True=1 和 False =0，其不区分大小写。

```
mysql>  SELECT TRUE, true, FALSE, false;
+------+------+-------+-------+
| TRUE | TRUE | FALSE | FALSE |
+------+------+-------+-------+
|    1 |    1 |     0 |     0 |
+------+------+-------+-------+
1 row in set (0.03 sec)
```

NULL Values

NULL 代表数据为空，不区分大小写，与命令 \N(不区分大小写) 同义。

注意事项: NULL 跟 0 的意义不一样，跟空字符串 '' 的意义也不一样。

5 TDSQL 的连接

5.1 mysql 命令行方式

TDSQL 通过 Proxy 接口提供和 MySQL 兼容的连接方式，用户可以通过 IP 地址、端口号、用户名以及密码连接 TDSQL 系统，连接语句如下：

语法：
mysql -hhost_ip -Pport -uusername -ppassword -c

示例：
mysql -h10.10.10.10 -P3306 -utest12 -ptest123 -c

5.2 JDBC+tomcat 连接配置

在 Tomcat 的 server.xml 中配置数据库连接时，推荐 JDBC 连接串如下：

```
jdbc:mysql://ip:port/db_name?user=your_username&password=your_pass
word&useLocalSessionStates=true&useUnicode=true&characterEncoding=
utf-8&serverTimezone=Asia/Shanghai"
```

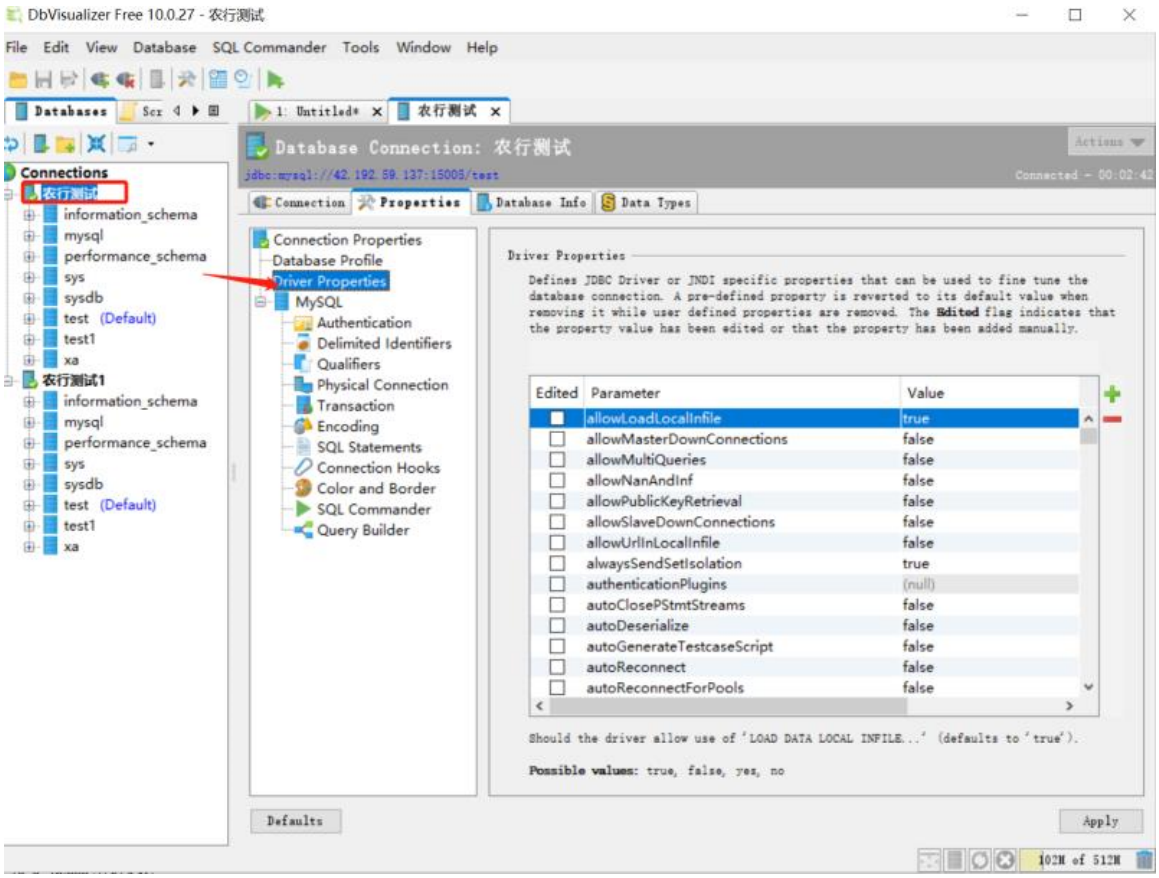
其他参数说明：

参数	含义	缺省值	推荐值
useLocalSessionState	配置驱动程序是否使用autocommit，read_only和transaction isolation的内部值(jdbc端的本地值),避免JDBC driver每次都去检查target database是否是ReadOnly,autocommit	false	true
rewriteBatchedStatements	用于保证jdbc driver可以批量执行SQL，按需配置	false	按需配置，建议true
useUnicode	是否使用Unicode字符集	false	按需配置，建议设置true

characterEncoding	字符编码格式	无	按需配置，建议设置 utf-8
serverTimezone	时区	local	按需配置，建议中国区部署设置为 Asia/Shanghai
netTimeoutForStreamingResults	当使用StremResultSet结果集时，建议配置该参数，保证使用数据库的默认超时时间	600	0（即应用端不配置，直接使用数据库服务器超时时间）
useCursorFetch	是否使用cursor来拉取数据。建议该值和TDSQL分布式保持一致。(分布式不支持游标)	false	false
useSSL	与数据库之间连接是否使用加密连接。建议互联网部署应用开启加密连接。开启后由于数据链路加密传输，影响部分性能。非互联网应用按需配置。说明：tdsql网关节点进已进行适配，默认开启usessl后，jdbc参数中无需配置 allowPublicKeyRetrieval=true	默认开启，即 useSSL=true(或 sslMode=PREFERRED)	按需配置（关闭方式：useSSL=false(或 sslMode=DISABLED)）

5.3 dbvisualizer 连接工具配置

dbvisualizer 是一个 ide 工具，使用 jdbc 连接 mysql，该 ide 默认设置 useCursorFetcht 为 true，需修改该参数为 false 访问 TDSQL。



6 SQL 参考

TDSQL 集中式实例高度兼容 MySQL 的协议和语法，以下是一些示例和演示。

6.1 DDL 语句

本节主要介绍了使用 DDL 语句创建表和常用 DDL 语句说明。

6.1.1 CREATE

6.1.1.1 CREATE DATABASE 语法

本节介绍 CREATE DATABASE 语法。

```
CREATE {DATABASE | SCHEMA} [IF NOT EXISTS] db_name
    [create_option] ...

create_option: [DEFAULT] {
    CHARACTER SET [=] charset_name
  | COLLATE [=] collation_name
}
```

注意事项：

- CREATE DATABASE 创建具有给定名称的数据库。要使用此语句，您需要对数据库具有 CREATE 权限。CREATE SCHEMA 是 CREATE DATABASE 的同义词。
- 如果数据库存在并且您没有指定 IF NOT EXISTS，则会发生错误。
- 在具有活动 LOCK TABLES 语句的会话中不允许 CREATE DATABASE。
- CHARACTER SET 选项指定默认的数据库字符集。COLLATE 选项指定默认的数据库排序规则。要查看可用的字符集和排序规则，请使用 SHOW CHARACTER SET 和 SHOW COLLATION 语句。

示例：

```
create database d2 default charset 'utf8mb4';
```

6.1.1.2 关于 CREATE TABLE

语法：

```
CREATE [TEMPORARY] TABLE [IF NOT EXISTS] tbl_name
    [(create_definition)]
    [table_options]
    [partition_options]

CREATE [TEMPORARY] TABLE [IF NOT EXISTS] tbl_name { LIKE old_tbl_n
ame | (LIKE old_tbl_name) }

create_definition: {
    col_name column_definition
  | {INDEX | KEY} [index_name] [index_type] (key_part,...)
      [index_option] ...
  | [INDEX | KEY] [index_name] (key_part,...)
      [index_option] ...
  | [CONSTRAINT [symbol]] PRIMARY KEY
      [index_type] (key_part,...)
      [index_option] ...
  | [CONSTRAINT [symbol]] UNIQUE [INDEX | KEY]
      [index_name] [index_type] (key_part,...)
      [index_option] ...
  | check_constraint_definition
}
```

```
column_definition: {
    data_type [NOT NULL | NULL] [DEFAULT]
      [AUTO_INCREMENT] [UNIQUE [KEY]] [[PRIMARY] KEY]
      [COMMENT 'string']
      [COLLATE collation_name]
      [COLUMN_FORMAT {FIXED | DYNAMIC | DEFAULT}]
      [ENGINE_ATTRIBUTE [=] 'string']
  | data_type
      [UNIQUE [KEY]] [[PRIMARY] KEY]
      [COMMENT 'string']
      [check_constraint_definition]
}

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

index_type:
USING {BTREE}

index_option: {
 index_type | COMMENT 'string'
}

[table_options]
table_option: {AUTO_INCREMENT [=] value
  | [DEFAULT] CHARACTER SET [=] charset_name
  | [DEFAULT] COLLATE [=] collation_name
  | COMMENT [=] 'string'
  | COMPRESSION [=] {'ZLIB' | 'LZ4' | 'NONE'}
  | KEY_BLOCK_SIZE [=] value
  | ENGINE [=] engine_name
  | ROW_FORMAT [=] {DEFAULT | DYNAMIC | FIXED | COMPRESSED | REDUN
DANT | COMPACT}
  | STATS_AUTO_RECALC [=] {DEFAULT | 0 | 1}
  | STATS_PERSISTENT [=] {DEFAULT | 0 | 1}
  | STATS_SAMPLE_PAGES [=] value)
}
partition_options:
    PARTITION BY
        | RANGE{(expr)}
        | LIST{(expr)}
    [SUBPARTITION BY
        {HASH(expr)
        |(column_list) }
    ]
    [(partition_definition [, partition_definition] ...)]

partition_definition:
    PARTITION partition_name
        [VALUES
```

```
            {LESS THAN {(expr | value_list) | MAXVALUE}
            |
            IN (value_list)}]
        [[STORAGE] ENGINE [=] engine_name]
        [COMMENT [=] 'string']
        [(subpartition_definition [, subpartition_definition]
 ...)]

subpartition_definition:
    SUBPARTITION logical_name
        [[STORAGE] ENGINE [=] engine_name]
        [COMMENT [=] 'string']
```

CREATE TABLE 创建一个具有给定名称的表。 您必须具有该表的 CREATE 权限。

默认情况下，表是在默认数据库中创建的，使用 InnoDB 存储引擎。 如果表存在、没有默认数据库或数据库不存在，则会发生错误。

CREATE TABLE 语句有以下几个方面：

表名

- *tbl_name*

  表名可以指定为 db_name.tbl_name 以在特定数据库中创建表。 假设数据库存在，无论是否存在默认数据库，这都有效。 如果使用带引号的标识符，请分别引用数据库和表名称。 例如，写`mydb`.`mytbl`，而不是`mydb.mytbl`。

- IF NOT EXISTS

  如果表存在，则防止发生错误。 但是，没有验证现有表具有与 CREATE TABLE 语句指示的结构相同的结构。

临时表

创建表时可以使用 TEMPORARY 关键字。 TEMPORARY 表仅在当前会话中可见，并在会话关闭时自动删除。

列数据类型和属性

- *data_type*

data_type 表示列定义中的数据类型。 某些属性不适用于所有数据类型。 AUTO_INCREMENT 仅适用于整数和浮点类型。

- NOT NULL | NULL

  如果既未指定 NULL 也未指定 NOT NULL，则将该列视为已指定 NULL。

- DEFAULT

  指定列的默认值。

  如果启用了 NO_ZERO_DATE 或 NO_ZERO_IN_DATE SQL 模式并且日期值默认值根据该模式不正确，则 CREATE TABLE 在未启用严格 SQL 模式的情况下生成警告，如果启用了严格模式，则生成错误。 例如，启用 NO_ZERO_IN_DATE 后， c1 DATE DEFAULT '2010-00-00' 会产生警告。

- COMMENT

  可以使用 COMMENT 选项指定列的注释，最长为 1024 个字符。 注释由 SHOW CREATE TABLE 和 SHOW FULL COLUMNS 语句显示。

- 表选项

  表选项用于优化表的行为。 在大多数情况下，您不必指定其中任何一个。 除非另有说明，否则这些选项适用于所有存储引擎。 不适用于给定存储引擎的选项可以作为表定义的一部分被接受和记住。 如果您稍后使用 ALTER TABLE 将表转换为使用不同的存储引擎，则此类选项将适用。

- ENGINE

  请使用 innodb 存储引擎。

- [DEFAULT] CHARACTER SET

  指定表的默认字符集。 CHARSET 是 CHARACTER SET 的同义词 。 如果字符集名称是 DEFAULT ，则使用数据库字符集。

- [DEFAULT] COLLATE

指定表的默认排序规则。

- COMMENT

  该表的注释，最多 2048 个字符。

- COMPRESSION

  用于 InnoDB 表的页面级压缩的压缩算法。 支持的值包括 Zlib、LZ4 和 None。 COMPRESSION 属性是在透明 页面压缩功能中引入的。 页面压缩仅支持驻留在 file-per-table 表空间中的 InnoDB 表，并且仅在支持稀疏文件 的 Linux 系统使用。

- KEY_BLOCK_SIZE

  对于 InnoDB 表，KEY_BLOCK_SIZE 指定用于压缩 InnoDB 表的页面大小（以千字节为单位）。 KEY_BLOCK_SIZE 值被视为提示； 如有必要，InnoDB 可以使用不同的大小。 KEY_BLOCK_SIZE 只能小 于或等于 innodb_page_size 值。 值 0 表示默认压缩页面大小，它是 innodb_page_size 值的一半。 根据 innodb_page_size，可能的 KEY_BLOCK_SIZE 值包括 0、1、2、4、8 和 16。

  建议在为 InnoDB 表指定 KEY_BLOCK_SIZE 时启用 innodb_strict_mode。 启用 innodb_strict_mode 时，指定无效的 KEY_BLOCK_SIZE 值会返回错误。 如果禁用 innodb_strict_mode，无效的 KEY_BLOCK_SIZE 值会导致警告，并且忽略 KEY_BLOCK_SIZE 选项。

  InnoDB 仅在表级别支持 KEY_BLOCK_SIZE。

  KEY_BLOCK_SIZE 不支持 32KB 和 64KB innodb_page_size 值。

  InnoDB 表压缩不支持这些页面大小。

  InnoDB 在创建临时表时不支持 KEY_BLOCK_SIZE 选项。

- ROW_FORMAT

定义存储行的物理格式。

创建禁用严格模式的表时，如果不支持指定的行格式，则使用存储引擎的默认行格式。 表的实际行格式在 Row_format 列中报告，以响应 SHOW TABLE STATUS。 create_options 列显示在 CREATE TABLE 语句中指定的行格式，SHOW CREATE TABLE 也是如此。

行格式选择因用于表的存储引擎而异。

对于 InnoDB 表：

- 默认行格式由 innodb_default_row_format 定义，其默认设置为 DYNAMIC。 当未定义 ROW_FORMAT 选项或使用 ROW_FORMAT=DEFAULT 时，将使用默认行格式。如果未定义 ROW_FORMAT 选项，或者使用 ROW_FORMAT=DEFAULT，则重建表的操作也会静 默地将表的行格式更改为 innodb_default_row_format 定义的默认值。
- 为了更有效地 InnoDB 存储数据类型，请使用 DYNAMIC 。
- 要为 InnoDB 表启用压缩，请指定 ROW_FORMAT=COMPRESSED。 创建临时表时不支持 ROW_FORMAT=COMPRESSED 选项。
- 当您指定非默认 ROW_FORMAT 子句时，请考虑启用 innodb_strict_mode 配置选项。
- 不支持 ROW_FORMAT=FIXED。 如果在禁用 innodb_strict_mode 时指定了 ROW_FORMAT=FIXED，InnoDB 会发出警告并假定 ROW_FORMAT=DYNAMIC。 如果在启用 innodb_strict_mode 时指定 ROW_FORMAT=FIXED，这是默认设置，InnoDB 将返回错误。

- STATS_AUTO_RECALC

  指定是否自动重新计算 InnoDB 表的持久统计信息。 值 DEFAULT 导致表的持久统计设置由 innodb_stats_auto_recalc 配置选项确定。 当表中 10% 的数据发生更改时，值 1 会导致重新计算统计信息。 值 0 阻止自动重新计算此表； 使用此设置，在对表进行实质性更改后，发出 ANALYZE TABLE 语句以重新计算统计信息。

- STATS_PERSISTENT

  指定是否为 InnoDB 表启用持久统计。 值 DEFAULT 导致表的持久统计设置由 innodb_stats_persistent 配置选项确定。 值 1 启用表的持久统计信息，而值 0 关闭此功能。 通过 CREATE TABLE 或 ALTER TABLE 语句启用持久统计后，在将代表性数据加载到表中后，发出 ANALYZE TABLE 语句来计算统计信息。

- STATS_SAMPLE_PAGES

  估计索引列的基数和其他统计信息时要采样的索引页数，例如由 ANALYZE TABLE 计算的那些。

表分区

partition_options 可用于控制使用 CREATE TABLE 创建的表的分区。

可以修改、合并、添加到表中以及从表中删除分区。

- PARTITION BY

  如果使用，则 partition_options 子句以 PARTITION BY 开头。 该子句包含用于确定分区的函数； 该函数返回一个从 1 到 num 的整数值，其中 num 是分区数。（一个表可以包含的用户定义分区的最大数量是 1024；子分区的数量包括在这个最大值中。）

- RANGE(*expr*)

  在这种情况下， expr 使用一组 VALUES LESS THAN 运算符显示一系列值。 使用范围分区时，您必须使用 VALUES LESS THAN 定义至少一个

分区。 您不能将 VALUES IN 与范围分区一起使用。

注意事项：

对于按 RANGE 分区的表，VALUES LESS THAN 必须与整数文字值或计算结果为单个整数值的表达式一起使用。 在 TDSQL 中，您可以在使用 PARTITION BY RANGE COLUMNS 定义的表中克服此限制。

假设您有一个表，您希望根据以下方案在包含年份值的列上进行分区。

Partition Number:	Years Range:
0	1990 and earlier
1	1991 to 1994
2	1995 to 1998
3	1999 to 2002
4	2003 to 2005
5	2006 and later

实现这种分区方案的表可以通过此处显示的 CREATE TABLE 语句实现，示例如下：

```
DROP TABLE IF EXISTS t1;
CREATE TABLE t1 (
    year_col  INT primary key,
    some_data INT
)
PARTITION BY RANGE (year_col) (
    PARTITION p0 VALUES LESS THAN (1991),
    PARTITION p1 VALUES LESS THAN (1995),
    PARTITION p2 VALUES LESS THAN (1999),
    PARTITION p3 VALUES LESS THAN (2002),
    PARTITION p4 VALUES LESS THAN (2006),
    PARTITION p5 VALUES LESS THAN MAXVALUE
);
```

VALUES LESS THAN 子句以类似于 switch ... case 块的 case 部分的方式顺序工作（在许多编程语言中都可以找到，例如 C、Java 和 PHP）。 也就是说，子句必须以这样一种方式排列，即每个

连续 VALUES LESS THAN 中指定的上限大于前一个的上限，引用 MAXVALUE 的一个在列表中排在最后。

- `RANGE COLUMNS(*column_list*)`

  RANGE 上的此变体有助于对使用多列范围条件的查询进行分区修剪（即具有 WHERE a = 1 AND b < 10 或 WHERE a = 1 AND b = 10 AND c < 10 等条件）。 它使您能够通过使用 COLUMNS 子句中的列列表和每个 PARTITION ... VALUES LESS THAN (value_list) 分区定义子句中的一组列值来指定多列中的值范围。 （在最简单的情况下，该集合由单个列组成。） column_list 和 value_list 中可以引用的最大列数为 16。

  COLUMNS 子句中使用的 column_list 可能只包含列名； 列表中的每一列必须是以下 TDSQL 数据类型之一：整数类型； 字符串类型； 和时间或日期列类型。 不允许使用 BLOB、TEXT、SET、ENUM、BIT 或空间数据类型的列； 也不允许使用浮点数类型的列。 您也不能在 COLUMNS 子句中使用函数或算术表达式。

  分区定义中使用的 VALUES LESS THAN 子句必须为出现在 COLUMNS() 子句中的每一列指定一个文字值； 也就是说，用于每个 VALUES LESS THAN 子句的值列表必须包含与 COLUMNS 子句中列出的列数相同的值。 尝试在 VALUES LESS THAN 子句中使用比 COLUMNS 子句中更多或更少的值会导致语句失败并显示错误 Inconsistency in usage of column lists for partitioning.... 您不能对出现在中的任何值使用 NULL 值小于。 对于第一列以外的给定列，可以多次使用 MAXVALUE，如下例所示：

```
DROP TABLE IF EXISTS rc;
CREATE TABLE rc (
    a INT NOT NULL,
    b INT NOT NULL,
primary key (a,b)
)
PARTITION BY RANGE COLUMNS(a,b) (
    PARTITION p0 VALUES LESS THAN (10,
5),
    PARTITION p1 VALUES LESS THAN (20,
10),
    PARTITION p2 VALUES LESS THAN (50,
MAXVALUE),
    PARTITION p3 VALUES LESS THAN (65,
MAXVALUE),
    PARTITION p4 VALUES LESS THAN (MAX
VALUE,MAXVALUE)
);
```

注意事项：

VALUES LESS THAN 值列表中使用的每个值必须与相应列的类型完全匹配； 不进行转换。 例如，不能将字符串 '1' 用于匹配使用整数类型的列的值（必须使用数字 1 代替），也不能将数字 1 用于匹配使用整数类型的列的值。 字符串类型（在这种情况下，您必须使用带引号的字符串：'1'）

- LIST(*expr*)

  这在基于具有一组受限可能值（例如州或国家/地区代码）的表列分配分区时非常有用。 在这种情况下，可以将属于某个州或国家的所有行分配给单个分区，或者可以为某个州或国家的集合保留一个分区。 它类似于 RANGE，不同之处在于只能使用 VALUES IN 来指定每个分区的允许值。

VALUES IN 与要匹配的值列表一起使用。 例如，您可以创建一个分区方案，如下所示：

```
DROP TABLE IF EXISTS client_firms;
CREATE TABLE client_firms (
    id   INT primary key,
    name VARCHAR(35)
)
PARTITION BY LIST (id) (
    PARTITION r0 VALUES IN (1, 5, 9, 13,
17, 21),
    PARTITION r1 VALUES IN (2, 6, 10, 14,
 18, 22),
    PARTITION r2 VALUES IN (3, 7, 11, 15,
 19, 23),
    PARTITION r3 VALUES IN (4, 8, 12, 16,
 20, 24)
);
```

注意事项：

使用列表分区时，您必须至少使用 VALUES IN 定义一个分区。 您不能将 VALUES LESS THAN 与 PARTITION BY LIST 一起使用。

- `LIST COLUMNS(*column_list*)`

  LIST 上的这个变体有助于使用多列上的比较条件（即具有 WHERE a = 5 AND b = 5 或 WHERE a = 1 AND b = 10 AND c = 5 等条件）的查询的分区修剪。它使您能够通过使用 COLUMNS 子句中的列列表和每个 PARTITION ... VALUES IN (value_list) 分区定义子句中的一组列值来指定多列中的值。

  LIST COLUMNS(column_list) 中使用的列列表和 VALUES IN(value_list) 中使用的值列表的数据类型规则与 RANGE COLUMNS(column_list) 中使用的列列表和值列表中使用的规则相同 VALUES LESS THAN(value_list)，除了在 VALUES IN 子句中不允许使用 MAXVALUE，您可以使用 NULL。

  用于 VALUES IN 和 PARTITION BY LIST COLUMNS 的值列表与用于 PARTITION BY LIST 的值列表之间有一个重要区别。当与 PARTITION BY LIST COLUMNS 一起使用时，VALUES IN 子句中的每个元素都必须是一组列值；每个集合中的值数必须与 COLUMNS 子句中使用的列数相同，并且这些值的数据类型必须与列的数据类型匹配（并以相同的顺序出现）。在最简

单的情况下，该集合由单个列组成。
column_list 和组成 value_list 的元素中可使用的最大列数为 16。

以下 CREATE TABLE 语句定义的表提供了一个使用 LIST COLUMNS 分区的表示例：

```
DROP TABLE IF EXISTS lc;
CREATE TABLE lc (
    a INT not NULL,
    b INT not NULL,
primary key(a,b)
)
PARTITION BY LIST COLUMNS(a,b) (
    PARTITION p0 VALUES IN( (0,0) ),
    PARTITION p1 VALUES IN( (0,1), (0,2),
(0,3), (1,1), (1,2) ),
    PARTITION p2 VALUES IN( (1,0), (2,0),
(2,1), (3,0), (3,1) ),
    PARTITION p3 VALUES IN( (1,3), (2,2),
(2,3), (3,2), (3,3) )
);
```

注意事项：

无论您在创建按 RANGE 或 LIST 分区的表时是否使用 PARTITIONS 子句，您仍然必须在表定义中至少包含一个 PARTITION VALUES 子句。

CREATE TEMPORARY TABLE 语法

创建表时可以使用 TEMPORARY 关键字。 TEMPORARY 表仅在当前会话中可见，并在会话关闭时自动删除。这意味着两个不同的会话可以使用相同的临时表名称，而不会相互冲突或与现有的同名非临时表发生冲突。（现有表是隐藏的，直到临时表被删除。）

InnoDB 不支持压缩临时表。当启用 innodb_strict_mode 时（默认值），如果指定了 ROW_FORMAT=COMPRESSED 或 KEY_BLOCK_SIZE，CREATE TEMPORARY TABLE 将返回错误。如果禁用 innodb_strict_mode，则会发出警告并使用非压缩行格式创建临时表。 innodb_file_per-table 选项不影响 InnoDB 临时表的创建。

CREATE TABLE 导致隐式提交，除非与 TEMPORARY 关键字一起使用。

TEMPORARY 表与数据库（模式）的关系非常松散。删除数据库不会自动删除在该数据库中创建的任何 TEMPORARY 表。

要创建临时表，您必须具有 CREATE TEMPORARY TABLES 权限。会话创建临时表后，服务器不会对该表执行进一步的权限检查。创建会话可以对表执行任何操作，例如 DROP TABLE、INSERT、UPDATE 或 SELECT。

这种行为的一个含义是会话可以操作其临时表，即使当前用户没有创建它们的权限。假设当前用户没有 CREATE TEMPORARY TABLES 权限，但能够执行定义者上下文存储过程，该存储过程以拥有 CREATE TEMPORARY TABLES 的用户的权限执行并创建临时表。当过程执行时，会话使用定义用户的权限。过程返回后，有效权限恢复为当前用户的有效权限，该用户仍然可以看到临时表并对其进行任何操作。

您不能使用 CREATE TEMPORARY TABLE ... LIKE 根据驻留在表空间、InnoDB 系统表空间 (innodb_system) 或通用表空间中的表的定义创建空表。此类表的表空间定义包含一个 TABLESPACE 属性，该属性定义了该表所在的表空间，上述表空间不支持临时表。要根据此类表的定义创建临时表，请改用以下语法：

```
CREATE TEMPORARY TABLE new_tbl(id int primary ke
y);
```

CREATE TABLE ... LIKE 语法

使用 CREATE TABLE ... LIKE 根据另一个表的定义创建一个空表，包括原始表中定义的任何列属性和索引，示例：

```
drop table if exists t1;
drop table if exists test2;
create table t1 (id int primary key);
create table test2 like t1;
```

副本是使用与原始表相同版本的表存储格式创建的。 原始表需要 SELECT 权限。

LIKE 仅适用于基表，不适用于视图。

重要：

当 LOCK TABLES 语句有效时，您不能执行 CREATE TABLE 或 CREATE TABLE ... LIKE。

CREATE TABLE ... LIKE 进行与 CREATE TABLE 相同的检查。这意味着如果当前的 SQL 模式与创建原始表时生效的模式不同，则表定义可能被认为对新模式无效并导致语句失败。

对于 CREATE TABLE ... LIKE，目标表保留原始表中生成的列信息。

对于 CREATE TABLE ... LIKE，目标表保留原始表中的表达式默认值。

对于 CREATE TABLE ... LIKE，目标表保留原始表中的 CHECK 约束，除了生成所有约束名称。

如果原始表是 TEMPORARY 表，则 CREATE TABLE ... LIKE 不会保留 TEMPORARY。要创建临时目标表，请使用 CREATE TEMPORARY TABLE ... LIKE。

6.1.1.3 创建普通表

示例：

```
DROP TABLE IF EXISTS t_stu;

CREATE TABLE t_stu (stu_id int primary key,first_name VARCH
AR(10),last_name VARCHAR(10),full_name VARCHAR(255)) ENGINE
=InnoDB default charset=utf8mb4;
```

6.1.1.4 创建分区表

6.1.1.4.1 支持的分区表类型

TDSQL 支持 RANGE、LIST、COLUMNS 类型的分区。

- **RANGE 分区**：这种类型的分区基于落在给定范围内的列值将行分配给分区。此类型还包含一种扩展类型为 RANGE COLUMNS 的分区类型。

  Range 分区支持类型：

  - DATE，DATETIME，TIMESTAMP
  - TINYINT, SMALLINT, MEDIUMINT, INT , BIGINT

- **LIST 分区**：类似于按 RANGE 分区，区别在于 LIST 分区是基于列值匹配一个离散值集合中的某个值来进行选择。此类型包含一种扩展类型为 LIST COLUMNS 的分区类型。

  List 分区支持类型：

  - DATE，DATETIME，TIMESTAMP
  - TINYINT, SMALLINT, MEDIUMINT, INT , BIGINT

- **COLUMN 分区**：RANGE 和 LIST 分区的扩展类型，分为 RANGE COLUMNS 和 LIST COLUMNS 两类。COLUMNS 分区允许在分区键中使用多个列，所有指定列都被考虑在内，以便在分区中放置行，以及确定在分区裁剪中检查分区的匹配行。RANGE COLUMNS 分区和 LIST COLUMNS 分区都支持使用非整数列来定义值范围或列表成员。

  COLUMN 分区支持类型：

  - DATE，DATETIME
  - TINYINT, SMALLINT, MEDIUMINT, INT , BIGINT
  - CHAR, VARCHAR

6.1.1.4.2 RANGE 分区

Range 分区表示例：

```
DROP TABLE IF EXISTS employees;

CREATE TABLE employees (

    id INT NOT NULL,
    fname VARCHAR(30),
    lname VARCHAR(30),
    hired DATE NOT NULL DEFAULT '1970-01-01',
    separated DATE NOT NULL DEFAULT '9999-12-31',
    job_code INT NOT NULL,
    store_id INT NOT NULL PRIMARY KEY
) PARTITION BY RANGE (store_id) (
PARTITION p0 VALUES LESS THAN (6),
PARTITION p1 VALUES LESS THAN (11),
PARTITION p2 VALUES LESS THAN (16),
PARTITION p3 VALUES LESS THAN (21));
```

RANGE 分区适用场景：

- 当需要删除旧的数据时。有了分区，只需要执行 alter table employees drop partition p0;就能删除以上 employees 表中 store_id<6 的所有数据，这样比 delete from employees where store_id<6;要有效得多。
- 想要使用一个包含有日期或时间值，或包含有从一些其他级数开始增长的值的列
- 经常运行直接依赖于用于分割表的列的查询。例如，当执行一个如“SELECT COUNT(*) FROM employees WHERE store_id=5 GROUP BY store_id；”这样的查询时，TDSQL 可以很迅速地确定只有分区 p0 需要扫描，这是因为余下的分区不可能包含有符合该 WHERE 子句的任何记录

RANGE COLUMNS 分区表示例：

范围列分区与范围分区类似，但是使您能够基于多个列值使用范围来定义分区。 另外，可以使用非整数类型的列来定义范围。

```
DROP TABLE IF EXISTS rcx;

CREATE TABLE rcx (
    a INT NOT NULL,
    b INT NOT NULL,
    c CHAR(3) NOT NULL,
    d INT NOT NULL,
    PRIMARY KEY (a, d , c)
) PARTITION BY RANGE COLUMNS (A, D, C) (
PARTITION p0 VALUES LESS THAN (5, 10, 'ggg'),
PARTITION p1 VALUES LESS THAN (10, 20, 'mmm'),
PARTITION p2 VALUES LESS THAN (15, 30, 'sss'),
PARTITION p3 VALUES LESS THAN (MAXVALUE, MAXVALUE, MAXVALUE)
);
```

6.1.1.4.3 LIST 分区

List 分区表示例：

```
DROP TABLE IF EXISTS employees_list;

CREATE TABLE employees_list (
    id INT NOT NULL,
    fname VARCHAR(30),
    lname VARCHAR(30),
    hired DATE NOT NULL DEFAULT '1970-01-01',
```

```
    separated DATE NOT NULL DEFAULT '9999-12-31',
    job_code INT,
    store_id INT PRIMARY KEY
) PARTITION BY LIST (store_id) (
PARTITION pNorth VALUES IN (3 , 5 , 6 , 9 , 17) ,
PARTITION pEast VALUES IN (1 , 2 , 10 , 11 , 19 , 20) ,
PARTITION pWest VALUES IN (4 , 12 , 13 , 14 , 18) ,
 PARTITION pCentral VALUES IN (7 , 8 , 15 , 16));
```

LIST COLUMNS 分区表示例：

```
DROP TABLE IF EXISTS customers_2;

CREATE TABLE customers_2 (
    first_name VARCHAR(25),
    last_name VARCHAR(25),
    street_1 VARCHAR(30),
    street_2 VARCHAR(30),
    city VARCHAR(15),
    renewal DATE primary key
)
PARTITION BY LIST COLUMNS(renewal) (
    PARTITION pWeek_1 VALUES IN('2010-02-01', '2010-02-02', '2010
-02-03',
        '2010-02-04', '2010-02-05', '2010-02-06', '2010-02-07'),
    PARTITION pWeek_2 VALUES IN('2010-02-08', '2010-02-09', '2010
-02-10',
        '2010-02-11', '2010-02-12', '2010-02-13', '2010-02-14'),
    PARTITION pWeek_3 VALUES IN('2010-02-15', '2010-02-16', '2010
-02-17',
        '2010-02-18', '2010-02-19', '2010-02-20', '2010-02-21'),
    PARTITION pWeek_4 VALUES IN('2010-02-22', '2010-02-23', '2010
-02-24',
        '2010-02-25', '2010-02-26', '2010-02-27', '2010-02-28')
);
```

6.1.1.5 CREATE INDEX 语法

通常，您在使用 CREATE TABLE 创建表本身时在表上创建所有索引。该准则对于 InnoDB 表尤其重要，其中主键决定了数据文件中行的物理布局。 CREATE INDEX 使您能够向现有表添加索引。

语法：

```
CREATE [UNIQUE ] INDEX index_name
    [index_type]
    ON tbl_name (key_part,...)
    [index_option]
    [algorithm_option | lock_option] ...

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

index_option: {
  index_type | COMMENT 'string'
}

index_type:
    USING {BTREE}

algorithm_option:
    ALGORITHM [=] {DEFAULT | INPLACE | COPY}

lock_option:
    LOCK [=] {DEFAULT | NONE | SHARED | EXCLUSIVE}
```

注意事项：

- CREATE INDEX 不能用于创建 PRIMARY KEY；对于主键，请改用 ALTER TABLE。
- 对于 INNODB 存储引擎，允许的索引类型为 BTREE。

CREATE INDEX 映射到 ALTER TABLE 语句以创建索引。CREATE INDEX 不能用于创建 PRIMARY KEY；请改用 ALTER TABLE。

以下部分描述了 CREATE INDEX 语句的不同方面：

- 列前缀索引
- 功能键索引
- 唯一索引
- 多值索引
- 索引选项

列前缀索引

对于字符串列，可以创建仅使用列值的前导部分的索引，使用 col_name(length) 语法指定索引前缀长度：

- 可以为 CHAR、VARCHAR、BINARY 和 VARBINARY 键部分指定前缀。
- 前缀限制以字节为单位。 但是，CREATE TABLE、ALTER TABLE 和 CREATE INDEX 语句中索引规范的前缀长度被解释为非二进制字符串类型（CHAR、VARCHAR、TEXT）的字符数和二进制字符串类型（BINARY、VARBINARY、BLOB）的字节数 . 在为使用多字节字符集的非二进制字符串列指定前缀长度时，请考虑这一点。前缀支持和前缀长度（如果支持）取决于存储引擎。 例如，对于使用 REDUNDANT 或 COMPACT 行格式的 InnoDB 表，前缀最长可达 767 字节。 对于使用 DYNAMIC 或 COMPRESSED 行格式的 InnoDB 表，前缀长度限制为 3072 字节。

如果指定的索引前缀超过最大列数据类型大小，则 CREATE INDEX 按如下方式处理索引：

- 对于非唯一索引，要么发生错误（如果启用了严格 SQL 模式），要么将索引长度减少到最大列数据类型大小并产生警告（如果未启用严格 SQL 模式）。
- 对于唯一索引，无论 SQL 模式如何，都会发生错误，因为减少索引长度可能会允许插入不满足指定唯一性要求的非唯一条目。

此处显示的语句使用 name 列的前 10 个字符创建索引（假设 name 具有非二进制字符串类型）：

```
CREATE INDEX part_of_name ON customer (name(10));
```

如果列中的名称通常在前 10 个字符中不同，则使用此索引执行的查找不应比使用从整个名称列创建的索引慢多少。 此外，为索引使用列前缀可以使索引文件更小，这可以节省大量磁盘空间，还可以加快 INSERT 操作。

功能键索引

“正常”索引索引列值或列值的前缀。 例如，在下表中，给定 t1 行的索引条目包括完整的 col1 值和由前 10 个字符组成的 col2 值的前缀：

```
DROP TABLE IF EXISTS t2;

CREATE TABLE t2 (col1 VARCHAR(10) primary key,col2
VARCHAR(20),INDEX(col1,col2(10)));
```

TDSQL 支持索引表达式值而不是列或列前缀值的功能键部分。 使用功能关键部件可以对未直接存储在表中的值进行索引。 例子：

```
DROP TABLE IF EXISTS t1;

CREATE TABLE t1 (col1 INT primary key, col2 INT, INDEX func_index
((ABS(col1))));

CREATE INDEX idx1 ON t1 ((col1 + col2));

CREATE INDEX idx2 ON t1 ((col1 + col2), (col1 - col2), col1);

ALTER TABLE t1 ADD INDEX ((col1 * 40) DESC);
```

具有多个关键部分的索引可以混合非功能性和功能性关键部分。

功能关键部件支持 ASC 和 DESC。

功能关键部件必须遵守以下规则。 如果关键部分定义包含不允许的构造，则会发生错误。

- 在索引定义中，将表达式括在括号内以将它们与列或列前缀区分开来。 例如，这是允许的； 表达式括在括号中：

  ```
  INDEX（（col1 + col2），（col3-col
  4））
  ```

  这会产生错误； 表达式未括在括号内：

```
INDEX（col1 + col2，col3 - col4）
```

- 功能键部分不能仅由列名组成。 例如，这是不允许的：

  ```
  INDEX（（col1），（col2））
  ```

  相反，将关键部分写为非功能性关键部分，不带括号：

  ```
  INDEX（col1，col2）
  ```

- 功能键部分表达式不能引用列前缀。

唯一索引

UNIQUE 索引创建了一个约束，使得索引中的所有值都必须是不同的。 如果您尝试添加具有与现有行匹配的键值的新行，则会发生错误。 如果为 UNIQUE 索引中的列指定前缀值，则列值在前缀长度内必须是唯一的。 UNIQUE 索引允许可以包含 NULL 的列有多个 NULL 值。

如果表的 PRIMARY KEY 或 UNIQUE NOT NULL 索引由具有整数类型的单个列组成，则可以使用 _rowid 来引用 SELECT 语句中的索引列，如下所示：

- 如果 PRIMARY KEY 由单个整数列组成，则 _rowid 指的是 PRIMARY KEY 列。 如果有一个 PRIMARY KEY 但它不包含单个整数列，则不能使用 _rowid。
- 否则，如果第一个 UNIQUE NOT NULL 索引包含单个整数列，则 _rowid 将引用该列。 如果第一个 UNIQUE NOT NULL 索引不包含单个整数列，则不能使用 _rowid。

多值索引

InnoDB 支持多值索引。 多值索引是在存储值数组的列上定义的二级索引。 “正常”索引对每个数据记录有一个索引记录 (1:1)。 多值索引可以为单个数据记录 (N:1) 包含多个索引记录。 多值索引用于索引 JSON 数组。 例如，在以下 JSON 文档中的邮政编码数组上定义的多值索引为每个邮政编码创建一个索引记录，每个索引记录引用相同的数据记录。

```
{"user":"Bob", "user_id":31, "zipcode":[94477,94536]}
```

创建多值索引

您可以在 CREATE TABLE、ALTER TABLE 或 CREATE INDEX 语句中创建多值索引。 这 SQL 数据类型数组。 然后使用 SQL 数据类型数组中的值透明地生成一个虚拟列； 最后，在虚拟列上创建功能索引（也称为虚拟索引）。 它是在形成多值索引的 SQL 数据类型数组的虚拟值列上定义的功能索引。

以下列表中的示例显示了在名为 customers 的表中的 JSON 列 custinfo 上的数组 $.zipcode 上创建多值索引 zip 的三种不同方式。 在每种情况下，JSON 数组都被转换为 UNSIGNED 整数值的 SQL 数据类型数组。

CREATE TABLE only:

```
DROP TABLE IF EXISTS customers;

CREATE TABLE customers (
      id BIGINT NOT NULL AUTO_INCREMENT PRIMARY KEY,
      modified DATETIME DEFAULT CURRENT_TIMESTAMP ON UPDATE
CURRENT_TIMESTAMP,
      custinfo JSON,
      INDEX zips((CAST(custinfo->'$.zipcode' AS UNSIGNED
ARRAY)) )
      );
```

CREATE TABLE plus CREATE INDEX:

```
DROP TABLE IF EXISTS customers;

CREATE TABLE customers (
      id BIGINT NOT NULL AUTO_INCREMENT PRIMARY KEY,
      modified DATETIME DEFAULT CURRENT_TIMESTAMP ON UPDATE
CURRENT_TIMESTAMP,
      custinfo JSON
      );

CREATE INDEX zips ON customers ((CAST(custinfo->'$.zipcode' AS
UNSIGNED ARRAY)));
```

多值索引也可以定义为复合索引的一部分。 此示例显示了一个复合索引，其中包括两个单值部分（用于 id 和 modified 列）和一个多值部分（用于 custinfo 列）：

```
DROP TABLE IF EXISTS customers;

CREATE TABLE customers (
```

```
      id BIGINT NOT NULL AUTO_INCREMENT PRIMARY KEY,
      modified DATETIME DEFAULT CURRENT_TIMESTAMP ON UPDATE
CURRENT_TIMESTAMP,
      custinfo JSON
      );

ALTER TABLE customers ADD INDEX comp(id, modified,
    (CAST(custinfo->'$.zipcode' AS UNSIGNED ARRAY)));
```

注意事项：

> 复合索引中只能使用一个多值键部分。 多值键部分可以相对于键的其他部分以任何顺序使用。 换句话说，刚刚显示的 ALTER TABLE 语句可以使用 comp(id, (CAST(custinfo->'$.zipcode' AS UNSIGNED ARRAY), modified)) （或任何其他排序）并且仍然有效。

使用多值索引

当在 WHERE 子句中指定以下函数时，优化器使用多值索引来获取记录：

- MEMBER OF()
- JSON_CONTAINS()
- JSON_OVERLAPS()

我们可以通过使用以下 CREATE TABLE 和 INSERT 语句创建和填充客户表来证明这一点：

```
DROP TABLE IF EXISTS customers;

CREATE TABLE customers (
         id BIGINT NOT NULL AUTO_INCREMENT PRIMARY KEY,
         modified DATETIME DEFAULT CURRENT_TIMESTAMP ON UPDATE
CURRENT_TIMESTAMP,
         custinfo JSON
         );

INSERT INTO customers VALUES
         (NULL, NOW(),
'{"user":"Jack","user_id":37,"zipcode":[94582,94536]}'),
         (NULL, NOW(),
'{"user":"Jill","user_id":22,"zipcode":[94568,94507,94582]}'),
         (NULL, NOW(),
'{"user":"Bob","user_id":31,"zipcode":[94477,94507]}'),
         (NULL, NOW(),
'{"user":"Mary","user_id":72,"zipcode":[94536]}'),
```

```
        (NULL, NOW(),
'{"user":"Ted","user_id":56,"zipcode":[94507,94582]}');
```

首先，我们在客户表上执行三个查询，每个查询使用 MEMBER OF()、JSON_CONTAINS() 和 JSON_OVERLAPS()，每个查询的结果如下所示：

```
mysql> SELECT * FROM customers
          WHERE 94507 MEMBER OF(custinfo->'$.zipcode');
+----+---------------------+---------------------------------
-----------------------------------+
| id | modified            | custinfo
                                   |
+----+---------------------+---------------------------------
-----------------------------------+
|  2 | 2019-06-29 22:23:12 | {"user": "Jill", "user_id": 22,
 "zipcode": [94568, 94507, 94582]} |
|  3 | 2019-06-29 22:23:12 | {"user": "Bob", "user_id": 31,
 "zipcode": [94477, 94507]}        |
|  5 | 2019-06-29 22:23:12 | {"user": "Ted", "user_id": 56,
 "zipcode": [94507, 94582]}        |
+----+---------------------+---------------------------------
-----------------------------------+
3 rows in set (0.00 sec)

mysql> SELECT * FROM customers
         WHERE JSON_CONTAINS(custinfo->'$.zipcode', CAST('
[94507,94582]' AS JSON));
+----+---------------------+---------------------------------
-----------------------------------+
| id | modified            | custinfo
                                   |
+----+---------------------+---------------------------------
-----------------------------------+
|  2 | 2019-06-29 22:23:12 | {"user": "Jill", "user_id": 22,
 "zipcode": [94568, 94507, 94582]} |
|  5 | 2019-06-29 22:23:12 | {"user": "Ted", "user_id": 56,
 "zipcode": [94507, 94582]}        |
+----+---------------------+---------------------------------
-----------------------------------+
2 rows in set (0.00 sec)

mysql> SELECT * FROM customers
         WHERE JSON_OVERLAPS(custinfo->'$.zipcode', CAST('
[94507,94582]' AS JSON));
+----+---------------------+---------------------------------
-----------------------------------+
| id | modified            | custinfo
                                   |
```

```
+----+---------------------+------------------------------
----------------------------------+
|  1 | 2019-06-29 22:23:12 | {"user": "Jack", "user_id": 37,
 "zipcode": [94582, 94536]}       |
|  2 | 2019-06-29 22:23:12 | {"user": "Jill", "user_id": 22,
 "zipcode": [94568, 94507, 94582]} |
|  3 | 2019-06-29 22:23:12 | {"user": "Bob", "user_id": 31,
 "zipcode": [94477, 94507]}       |
|  5 | 2019-06-29 22:23:12 | {"user": "Ted", "user_id": 56,
 "zipcode": [94507, 94582]}       |
+----+---------------------+------------------------------
----------------------------------+
4 rows in set (0.00 sec)
```

接下来，我们对前三个查询中的每一个运行 EXPLAIN：

```
mysql> EXPLAIN SELECT * FROM customers
          WHERE 94507 MEMBER OF(custinfo->'$.zipcode');
+----+-------------+-----------+------------+------+-------
--------+------+---------+------+------+----------+--------
-----+
| id | select_type | table     | partitions | type | possib
le_keys | key  | key_len | ref  | rows | filtered | Extra
     |
+----+-------------+-----------+------------+------+-------
--------+------+---------+------+------+----------+--------
-----+
|  1 | SIMPLE      | customers | NULL       | ALL  | NULL
        | NULL | NULL    | NULL |    5 |   100.00 | Using w
here |
+----+-------------+-----------+------------+------+-------
--------+------+---------+------+------+----------+--------
-----+
1 row in set, 1 warning (0.00 sec)

mysql> EXPLAIN SELECT * FROM customers
         WHERE JSON_CONTAINS(custinfo->'$.zipcode', CAST('
[94507,94582]' AS JSON));
+----+-------------+-----------+------------+------+-------
--------+------+---------+------+------+----------+--------
-----+
| id | select_type | table     | partitions | type | possib
le_keys | key  | key_len | ref  | rows | filtered | Extra
     |
+----+-------------+-----------+------------+------+-------
--------+------+---------+------+------+----------+--------
-----+
|  1 | SIMPLE      | customers | NULL       | ALL  | NULL
        | NULL | NULL    | NULL |    5 |   100.00 | Using w
here |
```

```
+----+-------------+-----------+------------+------+-------
--------+------+---------+------+------+----------+--------
-----+
1 row in set, 1 warning (0.00 sec)

mysql> EXPLAIN SELECT * FROM customers
         WHERE JSON_OVERLAPS(custinfo->'$.zipcode', CAST('
[94507,94582]' AS JSON));
+----+-------------+-----------+------------+------+-------
--------+------+---------+------+------+----------+--------
-----+
| id | select_type | table     | partitions | type | possib
le_keys | key  | key_len | ref  | rows | filtered | Extra
     |
+----+-------------+-----------+------------+------+-------
--------+------+---------+------+------+----------+--------
-----+
|  1 | SIMPLE      | customers | NULL       | ALL  | NULL
        | NULL | NULL    | NULL |    5 |   100.00 | Using w
here |
+----+-------------+-----------+------------+------+-------
--------+------+---------+------+------+----------+--------
-----+
1 row in set, 1 warning (0.01 sec)
```

刚刚显示的三个查询都不能使用任何键。 为了解决这个问题，我们可以在 JSON 列 (custinfo) 中的邮政编码数组上添加一个多值索引，如下所示：

```
ALTER TABLE customers
          ADD INDEX zips((CAST(custinfo->'$.zipcode' AS UNSIGNED
 ARRAY)));
```

当我们再次运行之前的 EXPLAIN 语句时，我们现在可以观察到查询可以（并且确实）使用刚刚创建的索引 zip：

```
mysql> EXPLAIN SELECT * FROM customers
          WHERE 94507 MEMBER OF(custinfo->'$.zipcode');
+----+-------------+-----------+------------+------+-------
--------+------+---------+-------+------+----------+-------
------+
| id | select_type | table     | partitions | type | possib
le_keys | key  | key_len | ref   | rows | filtered | Extra
      |
+----+-------------+-----------+------------+------+-------
--------+------+---------+-------+------+----------+-------
------+
|  1 | SIMPLE      | customers | NULL       | ref  | zips
```

```
        | zips | 9       | const |    1 |   100.00 | Using
where |
+----+-------------+-----------+------------+------+-------
--------+------+---------+-------+------+----------+-------
------+
1 row in set, 1 warning (0.00 sec)

mysql> EXPLAIN SELECT * FROM customers
         WHERE JSON_CONTAINS(custinfo->'$.zipcode', CAST('
[94507,94582]' AS JSON));
+----+-------------+-----------+------------+-------+------
---------+------+---------+------+------+----------+-------
------+
| id | select_type | table     | partitions | type  | possi
ble_keys | key  | key_len | ref  | rows | filtered | Extra
      |
+----+-------------+-----------+------------+-------+------
---------+------+---------+------+------+----------+-------
------+
|  1 | SIMPLE      | customers | NULL       | range | zips
         | zips | 9       | NULL |    6 |   100.00 | Using
where |
+----+-------------+-----------+------------+-------+------
---------+------+---------+------+------+----------+-------
------+
1 row in set, 1 warning (0.00 sec)

mysql> EXPLAIN SELECT * FROM customers
         WHERE JSON_OVERLAPS(custinfo->'$.zipcode', CAST('
[94507,94582]' AS JSON));
+----+-------------+-----------+------------+-------+------
---------+------+---------+------+------+----------+-------
------+
| id | select_type | table     | partitions | type  | possi
ble_keys | key  | key_len | ref  | rows | filtered | Extra
      |
+----+-------------+-----------+------------+-------+------
---------+------+---------+------+------+----------+-------
------+
|  1 | SIMPLE      | customers | NULL       | range | zips
         | zips | 9       | NULL |    6 |   100.00 | Using
where |
+----+-------------+-----------+------------+-------+------
---------+------+---------+------+------+----------+-------
------+
1 row in set, 1 warning (0.01 sec)
```

多值索引可以定义为唯一键。 如果定义为唯一键，则尝试插入多值索引中已存在的值会返回重复键错误。 如果已存在重复值，则尝试添加唯一的多值索引会失败，如下所示：

```
MySQL [test]> ALTER TABLE customers DROP INDEX zips;
Query OK, 0 rows affected (0.03 sec)
Records: 0  Duplicates: 0  Warnings: 0

MySQL [test]> ALTER TABLE customers
    ->          ADD UNIQUE INDEX zips((CAST(custinfo->'$.zipcode'
 AS UNSIGNED ARRAY)));
ERROR 1062 (23000): Duplicate entry '[94507, ' for key 'zips'

MySQL [test]> ALTER TABLE customers
    ->          ADD INDEX zips((CAST(custinfo->'$.zipcode' AS UNS
IGNED ARRAY)));
Query OK, 0 rows affected (0.03 sec)
Records: 0  Duplicates: 0  Warnings: 0
```

索引选项

index_option 值可以是以下任何值：

- `KEY_BLOCK_SIZE [=] *value*`

  InnoDB 表的索引级别不支持 KEY_BLOCK_SIZE

- *`index_type`*

  某些存储引擎允许您在创建索引时指定索引类型。

  Innodb 存储引擎支持的 index_type 为 BTREE 。

例如：

```
CREATE TABLE lookup (id INT primary key) ENGINE = INNODB;
CREATE INDEX id_index ON lookup (id) USING BTREE;
```

6.1.2 ALTER

6.1.2.1 ALTER TABLE

语法：

```
ALTER TABLE tbl_name
   [alter_option [, alter_option] ...]
   [partition_options]

alter_option: {
```

```
    table_options
  | ADD [COLUMN] col_name column_definition
        [FIRST | AFTER col_name]
  | ADD [COLUMN] (col_name column_definition,...)
  | ADD {INDEX | KEY} [index_name]
        [index_type] (key_part,...) [index_option] ...
  | ALGORITHM [=] {DEFAULT | INSTANT | INPLACE | COPY}
  | CHANGE [COLUMN] old_col_name new_col_name column_definition
        [FIRST | AFTER col_name]
  | [DEFAULT] CHARACTER SET [=] charset_name [COLLATE [=]
collation_name]
  | {DISABLE | ENABLE} KEYS
  | DROP [COLUMN] col_name
  | DROP {INDEX | KEY} index_name
  | LOCK [=] {DEFAULT | NONE | SHARED | EXCLUSIVE}
  | MODIFY [COLUMN] col_name column_definition
        [FIRST | AFTER col_name]
  | ORDER BY col_name [, col_name] ...
}

partition_options:
    partition_option [partition_option] ...

partition_option: {
    ADD PARTITION (partition_definition)
  | DROP PARTITION partition_names
  | TRUNCATE PARTITION {partition_names | ALL}
  | REORGANIZE PARTITION partition_names INTO (partition_definitions)
  | EXCHANGE PARTITION partition_name WITH TABLE tbl_name [{WITH |
WITHOUT} VALIDATION]
  | ANALYZE PARTITION {partition_names | ALL}
  | CHECK PARTITION {partition_names | ALL}
  | REBUILD PARTITION {partition_names | ALL}
  | REPAIR PARTITION {partition_names | ALL}
  | REMOVE PARTITIONING
}

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

index_type:
    USING {BTREE}

index_option: {
index_type | COMMENT 'string'
}
```

```
table_options:
  table_option [[,] table_option] ...

table_option: {AUTO_INCREMENT [=] value
 | [DEFAULT] CHARACTER SET [=] charset_name
 | [DEFAULT] COLLATE [=] collation_name
 | COMMENT [=] 'string'
 | COMPRESSION [=] {'ZLIB' | 'LZ4' | 'NONE'}
 | ENGINE [=] engine_name
 | KEY_BLOCK_SIZE [=] value
 | ROW_FORMAT [=] {DEFAULT | DYNAMIC | FIXED | COMPRESSED |
REDUNDANT | COMPACT}
 | STATS_AUTO_RECALC [=] {DEFAULT | 0 | 1}
 | STATS_PERSISTENT [=] {DEFAULT | 0 | 1}
 | STATS_SAMPLE_PAGES [=]
```

ALTER TABLE 更改表的结构。 例如，您可以添加或删除列、创建或销毁索引、更改现有列的类型或重命名列或表本身。 您还可以更改特征，例如用于表或表注释的存储引擎。

- 要使用 `ALTER TABLE`，你需要 `ALTER`， `CREATE` 和 `INSERT` 权限。
- 在表名后面，指定要进行的更改。 如果没有给出，`ALTER TABLE` 什么都不做。
- COLUMN 一词是可选的，可以省略，但 RENAME COLUMN 除外（用于区分列重命名操作和 RENAME 表重命名操作）。
- 该字 `COLUMN` 是可选的，可以省略，除了 `RENAME COLUMN` （区分列重命名操作和 `RENAME` 表重命名操作）。

ALTER TABLE 语句 还有其他几个方面，本节中的以下主题对此进行了描述：

- 表选项
- 性能和空间要求
- 并发控制
- 添加和删除列
- 重命名，重新定义和重新排序列

- 主键和索引
- 更改字符集
- 分区选项

表选项

table_options 表示可在 CREATE TABLE 语句中使用的表选项，例如 ENGINE、AUTO_INCREMENT。

将表选项与 ALTER TABLE 一起使用提供了一种更改单个表特征的便捷方式。 例如：

如果 t1 当前不是 InnoDB 表，则此语句将其存储引擎更改为 InnoDB：

```
ALTER TABLE t1 ENGINE = InnoDB;
```

- 指定 ENGINE 子句时， ALTER TABLE 重建表。 即使表已具有指定的存储引擎，也是如此。
- 在现有 InnoDB 表上运行 ALTER TABLE tbl_name ENGINE=INNODB 会执行“NULL”ALTER TABLE 操作，该操作可用于对 InnoDB 表进行碎片整理。在 InnoDB 表上运行 ALTER TABLE tbl_name FORCE 执行相同的功能.
- 尝试更改表的存储引擎的结果受所需存储引擎是否可用以及 NO_ENGINE_SUBSTITUTION SQL 模式设置的影响
- 要更改 InnoDB 表以使用压缩行存储格式：

```
ALTER TABLE t1 ROW_FORMAT = COMPRESSED;
```

- 要重置当前的自动增量值：

```
ALTER TABLE t1 AUTO_INCREMENT = 13;
```

注意事项：

您无法将计数器重置为小于或等于当前正在使用的值的值。对于 InnoDB，如果该值小于或等于 AUTO_INCREMENT 列中当前

的最大值，则将该值重置为当前最大 AUTO_INCREMENT 列值加 1。

- 要更改默认表格字符集：

```
ALTER TABLE t1 CHARACTER SET = utf8mb4;
```

- 添加（或更改）表注释：

```
ALTER TABLE t1 COMMENT = 'New table comment';
```

要验证表选项是否按预期更改，请使用 SHOW CREATE TABLE 或查询 INFORMATION_SCHEMA.TABLES。

性能和空间要求

ALTER TABLE 使用以下算法之一处理操作：

- COPY：对原表的一个副本进行操作，将表数据从原表逐行复制到新表中。 不允许并发 DML。
- INPLACE：操作避免复制表数据，但可能会就地重建表。 在操作的准备和执行阶段可能会短暂地对表进行独占元数据锁定。 通常，支持并发 DML
- INSTANT：操作只修改数据字典中的元数据。 在准备和执行过程中不对表采取独占元数据锁，表数据不受影响，操作瞬间完成。 允许并发 DML。

ALGORITHM 子句是可选的。 如果省略 ALGORITHM 子句，TDSQL 将 ALGORITHM=INSTANT 用于存储引擎和支持它的 ALTER TABLE 子句。 否则，使用 ALGORITHM=INPLACE。 如果不支持 ALGORITHM=INPLACE，则使用 ALGORITHM=COPY。

并发控制

对于支持它的 ALTER TABLE 操作，您可以使用 LOCK 子句来控制表被更改时并发读取和写入的级别。 为该子句指定非默认值使您能够在更改操作期间要求一定数量的并发访问或独占性，并在请求的锁定程度不可用时停止操作。

对于使用 ALGORITHM=INSTANT 的操作，只允许 LOCK = DEFAULT。 其他 LOCK 子句参数不适用。

LOCK 子句的参数是：

- LOCK = DEFAULT

  给定 ALGORITHM 子句（如果有）和 ALTER TABLE 操作的最大并发级别：如果支持，则允许并发读取和写入。 如果没有，则允许并发读取（如果支持）。 如果不是，则强制执行独占访问。

- LOCK = NONE

  如果支持，则允许并发读取和写入。 否则，会发生错误。

- LOCK = SHARED

  如果支持，则允许并发读取但阻止写入。 即使存储引擎支持给定 ALGORITHM 子句（如果有）和 ALTER TABLE 操作的并发写入，写入也会被阻止。 如果不支持并发读取，则会发生错误。

- LOCK = EXCLUSIVE

  强制独占访问。 即使存储引擎支持给定 ALGORITHM 子句（如果有）和 ALTER TABLE 操作的并发读/写，也会这样做。

添加和删除列

使用 ADD 向表添加新列，使用 DROP 删除现有列。

要在表行内的特定位置添加列，请使用 FIRST 或 AFTER col_name。 默认是最后添加列。

如果一个表只包含一列，则不能删除该列。 如果您打算删除该表，请改用 DROP TABLE 语句。

如果从表中删除列，列也会从它们所属的任何索引中删除。 如果构成索引的所有列都被删除，则该索引也会被删除。 如果您使用 CHANGE 或 MODIFY 缩短列上存在索引的列，并且生成的列长度小于索引长度，TDSQL 会自动缩短索引。

对于 ALTER TABLE ... ADD，如果列具有使用非确定性函数的表达式默认值，则该语句可能会产生警告或错误。

重命名，重新定义和重新排序列

CHANGE、MODIFY、RENAME COLUMN 和 ALTER 子句允许更改现有列的名称和定义。 它们具有以下比较特征：

- CHANGE：
  - 可以重命名列并更改其定义，或两者兼而有之。
  - 比 MODIFY 或 RENAME COLUMN 具有更多功能，但牺牲了某些操作的便利性。 CHANGE 如果不重命名列，则需要对其命名两次，如果仅重命名，则需要重新指定列定义。
  - 使用 FIRST 或 AFTER，可以对列重新排序。
- MODIFY：
  - 可以更改列定义但不能更改其名称。
  - 比 CHANGE 更方便地重命名列而不更改其定义。
  - 使用 FIRST 或 AFTER，可以对列重新排序。
- RENAME COLUMN：
  - 可以更改列名但不能更改其定义。
  - 比 CHANGE 更方便地重命名列而不更改其定义。
- ALTER：仅用于更改列默认值。

要更改列以更改其名称和定义，请使用 CHANGE，指定旧名称和新名称以及新定义。 例如，要将 INT NOT NULL 列从 a 重命名为 b 并将其定义更改为使用 BIGINT 数据类型同时保留 NOT NULL 属性，请执行以下操作：

```
DROP TABLE IF EXISTS t1;
create table t1(a int primary key,c int,d int,e int);
ALTER TABLE t1 CHANGE a b BIGINT NOT NULL;
```

要更改列定义但不更改其名称，请使用 CHANGE 或 MODIFY。 对于 CHANGE，语法需要两个列名，因此您必须两次指定相同的名称以保持名称不变。 例如，要更改 b 列的定义，请执行以下操作：

```
DROP TABLE IF EXISTS t1;
create table t1(a int primary key,b int,d int,c int);
ALTER TABLE t1 CHANGE b b INT NOT NULL;
```

MODIFY 更方便地更改定义而不更改名称，因为它只需要列名一次：

```
ALTER TABLE t1 MODIFY b INT NOT NULL;
```

要更改列名称但不更改其定义，请使用 CHANGE 或 RENAME COLUMN。 对于 CHANGE，语法需要列定义，因此要保持定义不变，您必须重新指定列当前具有的定义。 例如，要将 INT NOT NULL 列从 b 重命名为 a，请执行以下操作：

```
DROP TABLE IF EXISTS t1;
create table t1(b int primary key,c int,d int,e int);
ALTER TABLE t1 CHANGE b a INT NOT NULL;
```

RENAME COLUMN 更改名称而不更改定义更方便，因为它只需要旧名称和新名称：

```
DROP TABLE IF EXISTS t1;
create table t1(b int primary key,c int,d int,e int);
ALTER TABLE t1 RENAME COLUMN b TO a;
```

对于使用 CHANGE 或 MODIFY 的列定义更改，定义必须包括数据类型和所有应应用于新列的属性，除了索引属性（如 PRIMARY KEY 或 UNIQUE）。 原始定义中存在但未为新定义指定的属性不会被继承。 假设列 col1 被定义为 INT UNSIGNED DEFAULT 1 COMMENT 'my column' 并且您按如下方式修改该列，打算仅将 INT 更改为 BIGINT：

```
DROP TABLE IF EXISTS t1;
create table t1(col1 int primary key,col2 int,col3 int,col4
 int);
ALTER TABLE t1 MODIFY col1 BIGINT;
```

该语句将数据类型从 INT 更改为 BIGINT，但也会删除 UNSIGNED、DEFAULT 和 COMMENT 属性。 要保留它们，语句必须明确包含它们：

```
ALTER TABLE t1 MODIFY col1 BIGINT UNSIGNED DEFAULT 1 COMMEN
T'my column';
```

对于使用 CHANGE 或 MODIFY 的数据类型更改，TDSQL 尝试尽可能将现有列值转换为新类型。

注意事项：

这种转换可能会导致数据的更改。 例如，如果缩短字符串列，值可能会被截断。 如果转换到新数据类型会导致数据丢失，为了防止操作成功，请在使用 ALTER TABLE 之前启用严格 SQL 模式。

如果您使用 CHANGE 或 MODIFY 缩短列上存在索引的列，并且生成的列长度小于索引长度，TDSQL 会自动缩短索引。

要对表中的列重新排序，请在 CHANGE 或 MODIFY 操作中使用 FIRST 和 AFTER。

主键和索引

DROP PRIMARY KEY 删除主键。 如果没有主键，则会发生错误。

如果启用了 sql_require_primary_key 系统变量，则尝试删除主键会产生错误。

如果向表中添加 UNIQUE INDEX 或 PRIMARY KEY，TDSQL 会将其存储在任何非唯一索引之前，以允许尽早检测重复键。

DROP INDEX 删除索引，要确定索引名称，请使用 SHOW INDEX FROM tbl_name。

RENAME INDEX old_index_name to new_index_name 重命名索引。表的内容保持不变。 old_index_name 必须是表中未由同一 ALTER TABLE 语句删除的现有索引的名称。
new_index_name 是新的索引名称，在应用更改后不能与结果表中的索引名称重复。 两个索引名称都不能是 PRIMARY。

在 ALTER TABLE 语句之后，可能需要运行 ANALYZE TABLE 来更新索引基数信息。

ALTER INDEX 操作允许使索引可见或不可见。 优化器不使用不可见索引。 索引可见性的修改适用于主键以外的索引（显式或隐式）。 此功能与存储引擎无关（支持任何引擎）。

check 约束

ALTER TABLE 允许 CHECK 添加，删除或更改现有表的约束：

- 添加新约束：

```
ALTER TABLE tbl_name
    ADD CONSTRAINT [ symbol] CHECK（expr）[[NOT] ENFORCED];
```

- 删除名为 symbol 的现有 CHECK 约束：

```
ALTER TABLE tbl_name DROP CHECK symbol;
```

- 更改是否强制执行名为 symbol 的现有 CHECK 约束：

```
ALTER TABLE tbl_name ALTER CHECK symbol [NOT] ENFORCED;
```

如果表更改导致违反强制执行的 CHECK 约束，则会发生错误并且不会修改表。 发生错误的操作示例：

- 尝试将 AUTO_INCREMENT 属性添加到在 CHECK 约束中使用的列。
- 尝试添加强制 CHECK 约束或强制现有行违反约束条件的非强制 CHECK 约束。
- 尝试修改、重命名或删除在 CHECK 约束中使用的列，除非该约束也在同一语句中删除。 例外：如果 CHECK 约束仅引用单个列，则删除该列会自动删除该约束。

更改字符集

要将表默认字符集和所有字符列（CHAR、VARCHAR）更改为新字符集，请使用如下语句：

```
ALTER TABLE tbl_name CONVERT TO CHARACTER SET charset_name;
```

该语句还会更改所有字符列的排序规则。 如果未指定 COLLATE 子句来指示要使用的排序规则，则该语句将使用字符集的默认排序规则。

要仅更改表的默认 字符集，请使用以下语句：

```
ALTER TABLE tbl_name DEFAULT CHARACTER SET charset_name;
```

注意事项：

DEFAULT 一词是可选的。 默认字符集是在您没有为稍后添加到表中的列指定字符集时使用的字符集（例如，使用 ALTER TABLE ... ADD column）

ALTER TABLE 分区操作

ALTER TABLE 的分区相关子句可与分区表一起使用，用于重新分区、添加、删除、丢弃、导入、合并和拆分分区，以及执行分区维护。

ALTER TABLE ADD PARTITION 的 partition_definition 子句支持与 CREATE TABLE 语句的同名子句相同的选项。假设您创建了分区表，如下所示：

```
DROP TABLE IF EXISTS t1;
CREATE TABLE t1 (
      id INT,
      year_col INT primary key
)
PARTITION BY RANGE (year_col) (
    PARTITION p0 VALUES LESS THAN (1991),
    PARTITION p1 VALUES LESS THAN (1995),
    PARTITION p2 VALUES LESS THAN (1999)
);
```

您可以 p3 向此表 添加新分区 ， 以存储小于以下值的值 2002 ：

```
ALTER TABLE t1 ADD PARTITION (PARTITION p3 VALUES LESS THAN
 (2002));
```

DROP PARTITION 可用于删除一个或多个 RANGE 或 LIST 分区。存储在 partition_names 列表中命名的已删除分区中的任何数据都将被丢弃。 例如，给定先前定义的表 t1，您可以删除名为 p0 和 p1 的分区，如下所示：

```
ALTER TABLE t1 DROP PARTITION p0,p1;
```

说明：ADD PARTITION 和 DROP PARTITION 目前不支持 IF [NOT] EXISTS。

支持分区表的重命名。 您可以使用 ALTER TABLE ... REORGANIZE PARTITION; 间接重命名单个分区。 但是，此操作会复制分区的数据。

```
drop table orders_range;
CREATE TABLE orders_range (
```

```
       id INT AUTO_INCREMENT PRIMARY KEY,
       customer_surname VARCHAR(30),
       store_id INT,
        salesperson_id INT,
      order_Date DATE,
       note VARCHAR(500)
) PARTITION BY RANGE (id)
(PARTITION p0 VALUES LESS THAN (5),
PARTITION p1 VALUES LESS THAN (10),
PARTITION p3 VALUES LESS THAN (15));


alter table orders_range
reorganize partition p0 into
(partition n0 values less than(3),
partition n1 values less than(5));
```

要从选定分区中删除行，请使用 TRUNCATE PARTITION 选项。此选项采用一个或多个逗号分隔的分区名称列表。 考虑由该语句创建的表 t1：

```
drop table t1;
CREATE TABLE t1 (
    id INT,
    year_col INT primary key
)
PARTITION BY RANGE (year_col) (
    PARTITION p0 VALUES LESS THAN (1991),
    PARTITION p1 VALUES LESS THAN (1995),
    PARTITION p2 VALUES LESS THAN (1999),
    PARTITION p3 VALUES LESS THAN (2003),
    PARTITION p4 VALUES LESS THAN (2007)
);
```

要从分区中删除所有行 p0，请使用以下语句：

```
ALTER TABLE t1 TRUNCATE PARTITION p0;
```

刚才显示的语句与以下 DELETE 语句 具有相同的效果 ：

```
DELETE FROM t1 WHERE year_col <1991;
```

截断多个分区时，分区不必是连续的：这可以极大地简化分区表上的删除操作，否则如果使用 DELETE 语句完成，这些操作将需要非常复杂的 WHERE 条件。 例如，此语句删除分区 p1 和 p3 中的所有行：

```
ALTER TABLE t1 TRUNCATE PARTITION p1,p3;
```

等效的 DELETE 语句如下所示：

```
DELETE FROM t1 WHERE
  (year_col >= 1991 AND year_col < 1995)
    OR
  (year_col >= 2003 AND year_col < 2007);
```

如果使用 ALL 关键字代替分区名称列表，则该语句作用于所有表分区。

TRUNCATE PARTITION 只是删除行； 它不会改变表本身或其任何分区的定义。

要验证行是否已删除，请使用如下查询检查 INFORMATION_SCHEMA.PARTITIONS 表：

```
SELECT PARTITION_NAME, TABLE_ROWS
    FROM INFORMATION_SCHEMA.PARTITIONS
    WHERE TABLE_NAME = 't1';
```

要更改分区表使用的部分而非全部分区，您可以使用 REORGANIZE PARTITION。 该语句可以以多种方式使用：

- 将一组分区合并为一个分区。 这是通过在 partition_names 列表中命名多个分区并为 partition_definition 提供单个定义来完成的。
- 将现有分区拆分为多个分区。 通过为 partition_names 命名单个分区并提供多个 partition_definitions 来完成此操作。
- 更改使用 VALUES LESS THAN 定义的分区子集的范围或使用 VALUES IN 定义的分区子集的值列表。

注意事项：

对于没有明确命名的分区，TDSQL 会自动提供默认名称 p0、p1、p2 等。 对于子分区也是如此。

要将表分区或子分区与表交换，请使用 ALTER TABLE ... EXCHANGE PARTITION 语句——即将分区或子分区中的任何现有行移至非分区表，并将非分区表中的任何现有行移至 表分区或子分区。

另外有几个选项提供分区维护和修复功能，类似于通过 CHECK TABLE 和 REPAIR TABLE 等语句为非分区表实现的功能。 这些包括分析分区、检查分区、优化分区、重建分区和修复分区。 这些选项中的每一个都采用 partition_names 子句，该子句由一个或多个分区名称组成，以逗号分隔。 分区必须已经存在于目标表中。 您还可以使用 ALL 关键字代替 partition_names，在这种情况下，该语句作用于所有表分区。

- InnoDB 目前不支持 per-partition 优化； ALTER TABLE ... OPTIMIZE PARTITION 导致整个表重建和分析，并发出适当的警告。要解决此问题，请改用 ALTER TABLE ... REBUILD PARTITION 和 ALTER TABLE ... ANALYZE PARTITION。

- 未分区的表不支持 ANALYZE PARTITION、CHECK PARTITION、OPTIMIZE PARTITION 和 REPAIR PARTITION 选项。

- REMOVE PARTITIONING 使您能够在不影响表或其数据的情况下删除表的分区。 此选项可以与其他 ALTER TABLE 选项结合使用，例如用于添加、删除或重命名列或索引的选项。

除了其他变更规范之外，ALTER TABLE 语句可以包含 PARTITION BY 或 REMOVE PARTITIONING 子句，但必须在任何其他规范之后最后指定 PARTITION BY 或 REMOVE PARTITIONING 子句。

ADD PARTITION、DROP PARTITION、REORGANIZE PARTITION、ANALYZE PARTITION、CHECK PARTITION 和 REPAIR PARTITION 选项不能与单个 ALTER TABLE 中的其他更改规范结合使用，因为刚刚列出的选项作用于单个分区。

在给定的 ALTER TABLE 语句中只能使用以下任一选项的单个实例：PARTITION BY、ADD PARTITION、DROP PARTITION、TRUNCATE PARTITION、EXCHANGE PARTITION、REORGANIZE PARTITION 或 ANALYZE PARTITION、CHECK PARTITION、OPTIMIZE 分区，重建分区，删除分区。

例如，以下两个语句无效：

```
ALTER TABLE t1 ANALYZE PARTITION p1,ANALYZE PARTITION p2;

ALTER TABLE t1 ANALYZE PARTITION p1,CHECK PARTITION p2;
```

在第一种情况下，您可以使用带有单个 ANALYZE PARTITION 选项的单个语句同时分析表 t1 的分区 p1 和 p2，该选项列出了要分析的两个分区，如下所示：

```
ALTER TABLE t1 ANALYZE PARTITION p1,p2;
```

在第二种情况下，不能同时对同一个表的不同分区执行 ANALYZE 和 CHECK 操作。 相反，您必须发出两个单独的语句，如下所示：

```
ALTER TABLE t1 ANALYZE PARTITION p1;
ALTER TABLE t1 CHECK PARTITION p2;
```

子分区目前不支持 REBUILD 操作。 REBUILD 关键字明确禁止用于子分区，如果使用，会导致 ALTER TABLE 失败并出现错误。

当要检查或修复的分区包含任何重复的键错误时，CHECK PARTITION 和 REPAIR PARTITION 操作将失败。

ALTER TABLE 示例

从 t1 如下所示创建 的表开始：

```
CREATE TABLE t1 (a INTEGER primary key, b CHAR(10));
```

要将表重命名 t1 为 t2：

```
ALTER TABLE t1 RENAME t2;
```

要将列 a 从 INTEGER 更改为 TINYINT NOT NULL（保留名称不变），并将列 b 从 CHAR(10) 更改为 CHAR(20) 并将其从 b 重命名为 c：

```
ALTER TABLE t2 MODIFY a TINYINT NOT NULL, CHANGE b c CHAR(20);
```

要在 d 列上添加索引并在 a 列上添加 UNIQUE 索引：

```
ALTER TABLE t2 ADD INDEX(c),ADD UNIQUE(a);
```

要删除列 c：

```
ALTER TABLE t2 DROP COLUMN c;
```

添加名为 c 的新 AUTO_INCREMENT 整数列：

```
create table t2(a int primary key);
ALTER TABLE t2 ADD c INT UNSIGNED NOT NULL AUTO_INCREMENT,ADD KEY
 (c);
ALTER TABLE t2 RENAME t6;
```

6.1.3 DROP

6.1.3.1 Drop database

语法如下：

```
DROP {DATABASE | SCHEMA} [IF EXISTS] db_name
```

注意事项：

- DROP DATABASE 删除数据库中的所有表并删除数据库。 对此语句要非常小心！ 要使用 DROP DATABASE，您需要 DROP database 的 权限。 DROP SCHEMA 是 DROP DATABASE 的同义词。
- 删除数据库时， 不会自动删除专门为数据库授予的权限，必须手动删除它们。

示例：

```
DROP DATABASE test;
```

6.1.3.2 Drop index

语法如下：

```
DROP INDEX index_name ON tbl_name
    [algorithm_option | lock_option] ...

algorithm_option:
    ALGORITHM [=] {DEFAULT | INPLACE | COPY}

lock_option:
    LOCK [=] {DEFAULT | NONE | SHARED | EXCLUSIVE}
```

注意事项：

- 要删除主键，索引名称始终为 PRIMARY，必须将其指定为带引号的标识符，因为 PRIMARY 是保留字：DROP INDEX `PRIMARY` ON t;

示例：

```
MySQL [test]> show create table customer\G;
*************************** 1. row ***************************
       Table: customer
Create Table: CREATE TABLE `customer` (
  `cust_id` int(11) NOT NULL,
  `name` varchar(200) COLLATE utf8_bin DEFAULT NULL,
  `job_id` int(11) DEFAULT NULL,
  `job_name` varchar(300) COLLATE utf8_bin DEFAULT NULL,
  PRIMARY KEY (`cust_id`),
  UNIQUE KEY `uniq_idx_job_id` (`cust_id`,`job_id`),
  KEY `idx_cust` (`name`,`job_name`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8 COLLATE=utf8_bin shardkey=cust_id
1 row in set (0.00 sec)

MySQL [test]> drop index uniq_idx_job_id on customer;
Query OK, 0 rows affected (0.04 sec)

MySQL [test]> drop index idx_cust on customer;
Query OK, 0 rows affected (0.08 sec)
```

6.1.3.3 Drop table

语法如下：

```
DROP TABLE [IF EXISTS]
    tbl_name [, tbl_name] ...
    [RESTRICT | CASCADE]
```

注意事项：

- DROP TABLE 删除一个或多个表。 您必须拥有 DROP 每个表 的 权限。
- 对于每个表，它将删除表定义和所有表数据。 如果表已分区，则该语句将删除表定义，其所有分区，存储在这些分区中的所有数据以及与已删除表关联的所有分区定义。
- 删除表也会删除表的任何触发器。
- DROP TABLE 导致隐式提交。
- 删除表时，不会自动删除专门为该表授予的权限 。 必须手动删除它们。
- 所有 innodb_force_recovery 设置都不支持 DROP TABLE

- RESTRICT 和 CASCADE 关键字什么也不做。 它们被允许使从其他数据库系统移植更容易。

示例：

```
DROP TABLE test;
drop table test RESTRICT;
drop table test5 CASCADE;
```

6.1.4 TRUNCATE

语法如下：

```
TRUNCATE [TABLE] tbl_name
```

示例：

```
truncate table t1;
```

TRUNCATE TABLE 完全清空一个表。 它需要 DROP 权限。 从逻辑上讲，TRUNCATE TABLE 类似于删除所有行的 DELETE 语句，或一系列 DROP TABLE 和 CREATE TABLE 语句。

为了实现高性能，TRUNCATE TABLE 绕过了删除数据的 DML 方法。 因此，它不会导致 ON DELETE 触发器触发，不能对具有父子外键关系的 InnoDB 表执行，也不能像 DML 操作一样回滚。 但是，如果服务器在操作期间停止，则对使用原子 DDL 支持的存储引擎的表的 TRUNCATE TABLE 操作要么完全提交，要么回滚。

虽然 TRUNCATE TABLE 类似于 DELETE，但它被归类为 DDL 语句而不是 DML 语句。 它在以下方面与 DELETE 不同：

- 截断操作删除并重新创建表，这比逐行删除要快得多，特别是对于大表。
- 截断操作会导致隐式提交，因此无法回滚。
- 如果会话持有活动表锁，则无法执行截断操作。
- 截断操作不会为已删除的行数返回有意义的值。通常的结果是“0 行受影响”，这应该被解释为“没有信息”。
- 只要表定义有效，即使数据或索引文件已损坏，也可以使用 TRUNCATE TABLE 将表重新创建为空表。

- 任何 AUTO_INCREMENT 值都重置为其起始值。即使对于通常不重用序列值的 InnoDB 也是如此。
- 当与分区表一起使用时，TRUNCATE TABLE 保留分区；也就是说，删除并重新创建数据和索引文件，而分区定义不受影响。
- TRUNCATE TABLE 语句不会调用 ON DELETE 触发器。
- 支持截断损坏的 InnoDB 表。

出于二进制日志记录和复制的目的，TRUNCATE TABLE 被视为 DDL 而不是 DML，并且始终作为语句记录。

6.2 DML 语句

6.2.1 DELETE 语法

DELETE 是从表中删除行的 DML 语句。

DELETE 语句可以以 WITH 子句开头，以定义可在 DELETE 中访问的公共表表达式

单表语法

```
DELETE  [QUICK] [IGNORE] FROM tbl_name [[AS] tbl_alias]
      [PARTITION (partition_name [, partition_name] ...)]
      [WHERE where_condition]
      [ORDER BY ...]
      [LIMIT row_count]
```

DELETE 语句从 tbl_name 中删除行并返回删除的行数。 要检查已删除的行数，请调用 ROW_COUNT() 函数

主要条款

可选 WHERE 子句中的条件标识要删除的行。 如果没有 WHERE 子句，则删除所有行。

where_condition 是一个表达式，对于要删除的每一行，它的计算结果为真。

如果指定了 ORDER BY 子句，则按指定的顺序删除行。 LIMIT 子句限制了可以删除的行数。 这些子句适用于单表删除，但不适用于多表删除。

多表语法

```
DELETE [QUICK] [IGNORE]
    tbl_name[.*] [, tbl_name[.*]] ...
    FROM table_references
    [WHERE where_condition]

DELETE [QUICK] [IGNORE]
    FROM tbl_name[.*] [, tbl_name[.*]] ...
    USING table_references
    [WHERE where_condition]
```

权限

您需要对表具有 DELETE 权限才能从中删除行。 对于任何只读的列，您只需要 SELECT 特权，例如在 WHERE 子句中命名的列。

性能

当您不需要知道删除的行数时，TRUNCATE TABLE 语句是一种比不带 WHERE 子句的 DELETE 语句更快的清空表的方法。 与 DELETE 不同，TRUNCATE TABLE 不能在事务中使用，也不能在表上有锁的情况下使用。

为确保给定的 DELETE 语句不会花费太多时间，DELETE 的 LIMIT row_count 子句指定要删除的最大行数。 如果要删除的行数大于限制，则重复 DELETE 语句，直到受影响的行数小于 LIMIT 值。

分区表支持

DELETE 支持使用 PARTITION 子句进行显式分区选择，该子句采用一个或多个分区或子分区（或两者）的逗号分隔名称列表，从中选择要删除的行。 未包含在列表中的分区将被忽略。 给定一个分区表 t，其分区名为 p0，执行 DELETE FROM t PARTITION (p0) 语句对表具有与执行 ALTER TABLE t TRUNCATE PARTITION (p0) 相同的效果； 在这两种情况下，分区 p0 中的所有行都将被删除。

测试。 例如， DELETE FROM t PARTITION (p0) WHERE c < 5 仅从分区 p0 中删除条件 c < 5 为真的行； 不会检查任何其他分区中的行，因此不会受到 DELETE 的影响。

PARTITION 子句也可用于多表 DELETE 语句。 对于在 FROM 选项中命名的每个表，您最多可以使用一个这样的选项。

修饰符

该 DELETE 语句支持以下修饰符：

- IGNORE 修饰符使 MySQL 在删除行的过程中忽略可忽略的错误。（在解析阶段遇到的错误以通常的方式处理。）由于使用 IGNORE 而被忽略的错误将作为警告返回。

删除顺序

如果 DELETE 语句包含 ORDER BY 子句，则按该子句指定的顺序删除行。 这主要与 LIMIT 结合使用。 例如，以下语句查找与 WHERE 子句匹配的行，按时间戳列对它们进行排序，然后删除第一个（最旧的）行：

```
DELETE FROM somelog WHERE user = 'jcole'
ORDER BY timestamp_column LIMIT 1;
```

InnoDB 表

如果要从大表中删除许多行，则可能会超出 InnoDB 表的锁表大小。为了避免这个问题，或者只是为了尽量减少表保持锁定的时间，以下策略（根本不使用 DELETE）可能会有所帮助：

1. 选择不删除的行放入与原表结构相同的空表中：

   ```
   INSERT INTO t_copy SELECT * FROM t WHERE ...;
   ```

2. 使用 RENAME TABLE 以原子方式将原始表移开，并将副本重命名为原始名称：

   ```
   RENAME TABLE t TO t_old，t_copy TO t;
   ```

3. 删除原始表：

   ```
   DROP TABLE t_old;
   ```

在 RENAME TABLE 执行时，没有其他会话可以访问所涉及的表，因此重命名操作不受并发问题的影响。

多表删除

您可以在 DELETE 语句中指定多个表以根据 WHERE 子句中的条件从一个或多个表中删除行。 您不能在多表 DELETE 中使用 ORDER BY 或 LIMIT。

对于第一个多表语法，仅删除 FROM 子句之前列出的表中的匹配行。对于第二种多表语法，仅删除 FROM 子句中（在 USING 子句之前）

列出的表中的匹配行。 效果是您可以同时从多个表中删除行，并拥有仅用于搜索的附加表：

```
DELETE t1, t2 FROM t1 INNER JOIN t2 INNER JOIN t3
WHERE t1.id=t2.id AND t2.id=t3.id;
```

要么：

```
DELETE FROM t1, t2 USING t1 INNER JOIN t2 INNER JOIN t3
WHERE t1.id=t2.id AND t2.id=t3.id;
```

这些语句在搜索要删除的行时使用所有三个表，但仅从表 t1 和 t2 中删除匹配的行。

前面的示例使用 INNER JOIN，但多表 DELETE 语句可以使用 SELECT 语句中允许的其他类型的连接，例如 LEFT JOIN。 例如，要删除 t1 中存在但在 t2 中没有匹配项的行，请使用 LEFT JOIN：

```
DELETE t1 FROM t1 LEFT JOIN t2 ON t1.id=t2.id WHERE t2.id IS NULL;
```

如果您使用 DELETE 涉及 InnoDB 具有外键约束 的表的多表 语句，则 TDSQL 优化器可能会按照与其父/子关系不同的顺序处理表。 在这种情况下，语句失败并回滚。 相反，您应该从单个表中删除并依赖于 提供 的 ON DELETE 功能， InnoDB 以便相应地修改其他表。

注意事项：

如果声明表的别名，则在引用表时必须使用别名：

```
DELETE t1 FROM test AS t1, test2 WHERE ...
```

多表 DELETE 中的表别名应仅在语句的 table_references 部分中声明。在其他地方，允许使用别名引用，但不允许使用别名声明。

正确：

```
DELETE a1, a2 FROM t1 AS a1 INNER JOIN t2 AS a2
WHERE a1.id=a2.id;

DELETE FROM a1, a2 USING t1 AS a1 INNER JOIN t2 AS a2
WHERE a1.id=a2.id;
```

不正确：

```
DELETE t1 AS a1, t2 AS a2 FROM t1 INNER JOIN t2
WHERE a1.id=a2.id;

DELETE FROM t1 AS a1, t2 AS a2 USING t1 INNER JOIN t2
WHERE a1.id=a2.id;
```

6.2.2 INSERT 语法

语法如下：

```
INSERT [IGNORE]
    [INTO] tbl_name
    [PARTITION (partition_name [, partition_name] ...)]
    [(col_name [, col_name] ...)]
    {{VALUES | VALUE} (value_list) [, (value_list)] ...
      |
      VALUES row_constructor_list
    }
    [ON DUPLICATE KEY UPDATE assignment_list]

INSERT [IGNORE]
    [INTO] tbl_name
    [PARTITION (partition_name [, partition_name] ...)]
    SET assignment_list
    [ON DUPLICATE KEY UPDATE assignment_list]

INSERT [IGNORE]
    [INTO] tbl_name
    [PARTITION (partition_name [, partition_name] ...)]
    [(col_name [, col_name] ...)]
    {SELECT ... | TABLE table_name}
    [ON DUPLICATE KEY UPDATE assignment_list]

value:
    {expr | DEFAULT}

value_list:
    value [, value] ...

row_constructor_list:
    ROW(value_list)[, ROW(value_list)][, ...]

assignment:
    col_name = [row_alias.]value

assignment_list:
    assignment [, assignment] ...
```

INSERT 将新行插入到现有表中。语句的 INSERT ... VALUES、INSERT ... VALUES ROW() 和 INSERT ... SET 形式根据显式指定的值插入行。 INSERT ... SELECT 表单插入从另一个表或多个表中选择的行。您还可以在使用 INSERT ... TABLE 从单个表中插入行。如果要插入的

行会导致 UNIQUE 索引或 PRIMARY KEY 中的重复值，则带有 ON DUPLICATE KEY UPDATE 子句的 INSERT 可以更新现有行。

插入表需要对该表具有 INSERT 权限。如果使用 ON DUPLICATE KEY UPDATE 子句并且重复键导致执行 UPDATE，则该语句需要对要更新的列具有 UPDATE 特权。对于已读取但未修改的列，您只需要 SELECT 特权（例如，对于仅在 ON DUPLICATE KEY UPDATE 子句中 col_name=expr 赋值的右侧引用的列）。

插入分区表时，您可以控制哪些分区和子分区接受新行。 PARTITION 子句采用表的一个或多个分区或子分区（或两者）的逗号分隔名称列表。如果要由给定 INSERT 语句插入的任何行与列出的分区之一不匹配，则 INSERT 语句将失败并显示错误发现行与给定分区集不匹配。

tbl_name 是应插入行的表。 指定语句为其提供值的列，如下所示：

- 在表名后面提供一个用括号括起来的以逗号分隔的列名列表。 在这种情况下，每个命名列的值必须由 VALUES 列表、VALUES ROW() 列表或 SELECT 语句提供。 对于 INSERT TABLE 形式，源表中的列数必须与要插入的列数相匹配。

- 如果您没有为 INSERT ... VALUES 或 INSERT ... SELECT 指定列名列表，则表中每一列的值必须由 VALUES 列表、SELECT 语句或 TABLE 语句提供。 如果您不知道表中列的顺序，请使用 DESCRIBE tbl_name 查找。

- SET 子句通过名称显式指示列，以及分配给每个列的值。

列值可以通过多种方式给出：

- 如果未启用严格 SQL 模式，则任何未显式指定值的列都将设置为其默认（显式或隐式）值。例如，如果您指定的列列表未命名表中的所有列，则未命名的列将设置为其默认值。

- 如果启用了严格 SQL 模式，并且 INSERT 语句没有为每个没有默认值的列指定显式值，则会生成错误。

- 如果列列表和 VALUES 列表都为空，则 INSERT 将创建一行，其中每一列都设置为其默认值：

```
INSERT INTO tbl_name () VALUES();
```

如果未启用严格模式，TDSQL 对任何没有明确定义默认值的列使用隐式默认值。 如果启用了严格模式，如果任何列没有默认值，则会发生错误。

- 使用关键字 DEFAULT 将列显式设置为其默认值。 这使得编写将值分配给除少数列之外的所有列的 INSERT 语句变得更容易，因为它使您能够避免编写不包括表中每一列的值的不完整 VALUES 列表。 否则，您必须提供与 VALUES 列表中的每个值对应的列名称列表。

- 如果显式插入生成的列，则唯一允许的值是 DEFAULT。

- 在表达式中，您可以使用 DEFAULT(col_name) 为列 col_name 生成默认值。

- 如果表达式数据类型与列数据类型不匹配，则可能会发生提供列值的表达式 expr 的类型转换。 根据列类型，给定值的转换可能会导致不同的插入值。 例如，将字符串 '1999.0e-2' 插入 INT、FLOAT、DECIMAL(10,6) 或 YEAR 列会分别插入值 1999、19.9921、19.992100 或 1999。 INT 和 YEAR 列中存储的值是 1999，因为字符串到数字的转换只查看字符串的初始部分，因为它可能被视为有效的整数或年份。 对于 FLOAT 和 DECIMAL 列，字符串到数字的转换将整个字符串视为有效的数值。

- 表达式 expr 可以引用先前在值列表中设置的任何列。 例如，您可以这样做，因为 col2 的值引用了之前已分配的 col1：

```
INSERT INTO tbl_name (col1,col2) VALUES(15,col1*2);
```

但是以下是不合法的，因为 col1 的值指的是 col2，它是在 col1 之后赋值的：

```
INSERT INTO tbl_name (col1,col2) VALUES(col2*2,15);
```

使用 VALUES 语法的 INSERT 语句可以插入多行。 为此，请包含多个以逗号分隔的列值列表，将列表括在括号内并用逗号分隔。 例子：

```
INSERT INTO tbl_name (a,b,c)  VALUES(1,2,3), (4,5,6),
(7,8,9);
```

每个值列表必须包含与每行插入的值一样多的值。 以下语句无效，因为它包含一个包含九个值的列表，而不是三个包含三个值的列表：

```
INSERT INTO tbl_name (a,b,c) VALUES(1,2,3,4,5,6,7,8,9);
```

INSERT 语句支持以下修饰符：

- 如果使用 IGNORE 修饰符，则执行 INSERT 语句时发生的可忽略错误将被忽略。 例如，如果没有 IGNORE，复制表中现有 UNIQUE 索引或 PRIMARY KEY 值的行会导致重复键错误并且语句被中止。 使用 IGNORE，该行被丢弃并且不会发生错误。 忽略的错误会生成警告。 IGNORE 对插入到分区表中的插入有类似的影响，其中找不到与给定值匹配的分 区。如果没有 IGNORE，这 样的 INSERT 语句会因错误而中止。使用 INSERT IGNORE 时，对于包含不匹配值的行，插入操作会以静默方式 失败，但会插入匹配的行。如果未指定 IGNORE，则会触发错误的数据转换会中止语句。使用 IGNORE，将无效值调整为最接近的值并插入；产生警告但语句不会中止。您可以使用 REPLACE 而不是 INSERT 来覆盖旧行。在处理包含重复旧行的唯一键值的新行时，REPLACE 是 INSERT IGNORE 的对应物：新行替换旧行而不是被丢弃。
- 如果您指定 ON DUPLICATE KEY UPDATE，并且插入的行会导致 UNIQUE 索引或 PRIMARY

KEY 中出现重复值，则会发生旧行的 UPDATE。如果将行作为新行插入，则每行的受影响行值为 1，如果更新现有行，则为 2，如果现有行设置为其当前值，则为 0。 如果在连接到 mysqld 时为 mysql_real_connect() C API 函数指定 CLIENT_FOUND_ROWS 标志，则如果现有行设置为其当前值，则受影响的行值为 1（而不是 0）。

INSERT ... SELECT 语法

```
INSERT [IGNORE]
      [INTO] tbl_name
      [PARTITION (partition_name [, partition_name] ...)]
      [(col_name [, col_name] ...)]
      {SELECT ... | TABLE table_name}
      [ON DUPLICATE KEY UPDATE assignment_list]

value:
    {expr | DEFAULT}

assignment:
    col_name = value

assignment_list:
    assignment [, assignment] ...
```

使用 INSERT ... SELECT，您可以从 SELECT 语句的结果中快速将多行插入到一个表中，该语句可以从一个或多个表中进行选择。 例如：

```
INSERT INTO tbl_temp2 (fld_id)
SELECT tbl_temp1.fld_order_id
FROM tbl_temp1 WHERE tbl_temp1.fld_order_id > 100;
```

以下条件适用于 INSERT ... SELECT 语句，除非另有说明，否则也适用于 INSERT ... TABLE：

- 指定 IGNORE 以忽略会导致重复键违规的行。
- INSERT 语句的目标表可能出现在查询的 SELECT 部分的 FROM 子句中，或者作为由 TABLE 命名的表。但是，您不能在子查询中插入表并从同一个表中进行选择。当从同一个表中选择和插入时，TDSQL 创建一个内部临时表来保存来自 SELECT 的行，然后将这些行插入到目标表中。 但是，当 t 是 TEMPORARY 表时，不能使用 INSERT INTO t ... SELECT ... FROM t，因为在同一语句中不能两次引用 TEMPORARY 表。

出于同样的原因，当 t 是临时表时，您不能使用 INSERT INTO t ... TABLE t。

- AUTO_INCREMENT 列照常工作。
- 为确保二进制日志可用于重新创建原始表，TDSQL 不允许 INSERT ... SELECT 或 INSERT ... TABLE 语句的并发插入。
- 为避免 SELECT 和 INSERT 引用同一个表时出现歧义的列引用问题，请为 SELECT 部分中使用的每个表提供唯一的别名，并使用适当的别名限定该部分中的列名。

6.2.3 REPLACE 语法

语法如下：

```
REPLACE
    [INTO] tbl_name
    [PARTITION (partition_name [, partition_name] ...)]
    [(col_name [, col_name] ...)]
    {{VALUES | VALUE} (value_list) [, (value_list)] ...
      |
      VALUES row_constructor_list
    }

REPLACE
    [INTO] tbl_name
    [PARTITION (partition_name [, partition_name] ...)]
    SET assignment_list

REPLACE
    [INTO] tbl_name
    [PARTITION (partition_name [, partition_name] ...)]
    [(col_name [, col_name] ...)]
    {SELECT ... | TABLE table_name}

value:
    {expr | DEFAULT}

value_list:
    value [, value] ...

row_constructor_list:
    ROW(value_list)[, ROW(value_list)][, ...]

assignment:
    col_name = value

assignment_list:
    assignment [, assignment] ...
```

REPLACE 的工作方式与 INSERT 完全相同，但如果表中的旧行与 PRIMARY KEY 或 UNIQUE 索引的新行具有相同的值，则在插入新行之前删除旧行。

注意：

仅当表具有 PRIMARY KEY 或 UNIQUE 索引时，REPLACE 才有意义。否则，它就等同于 INSERT，因为没有索引可用于确定新行是否与另一行重复。

所有列的值均取自 REPLACE 语句中指定的值。任何缺失的列都设置为其默认值，就像 INSERT 一样。您不能引用当前行中的值并在新行中使用它们。如果您使用诸如 SET col_name = col_name + 1 之类的赋

值，则对右侧列名的引用将被视为 DEFAULT(col_name)，因此该赋值等效于 SET col_name = DEFAULT(col_name) + 1。

要使用 REPLACE，您必须同时拥有表的 INSERT 和 DELETE 权限。

如果显式替换生成的列，则唯一允许的值是 DEFAULT。

REPLACE 支持使用 PARTITION 子句的显式分区选择，其中包含以逗号分隔的分区、子分区或两者的名称列表。与 INSERT 一样，如果无法将新行插入到这些分区或子分区中的任何一个中，则 REPLACE 语句将失败并显示错误“找到与给定分区集不匹配的行”。

REPLACE 语句返回一个计数以指示受影响的行数。这是删除和插入的行的总和。如果单行 REPLACE 的计数为 1，则插入了一行并且没有删除任何行。如果计数大于 1，则在插入新行之前删除了一个或多个旧行。如果表包含多个唯一索引并且新行重复不同唯一索引中不同旧行的值，则单行可以替换多个旧行。

受影响的行数可以很容易地确定 REPLACE 是只添加了一行还是还替换了任何行：检查计数是 1（添加）还是更大（替换）。

您不能在子查询中替换到一个表并从同一个表中进行选择。

TDSQL 使用以下算法进行 REPLACE（和 LOAD DATA ... REPLACE）：

1. 尝试将新行插入表中
2. 由于主键或唯一索引发生重复键错误而导致插入失败：
    a.从表中删除具有重复键值的冲突行
    b.再次尝试将新行插入表中

考虑由以下 CREATE TABLE 语句创建的表：

```
DROP TABLE IF EXISTS test;
CREATE TABLE test (
       id INT UNSIGNED NOT NULL AUTO_INCREMENT,
      data VARCHAR(64) DEFAULT NULL,
      ts TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP
 ON UPDATE CURRENT_TIMESTAMP,
  PRIMARY KEY (id)
);
```

当我们创建这个表并运行 TDSQL 客户端显示的语句时，结果如下：

```
mysql> REPLACE INTO test VALUES (1, 'Old', '2014-08-20 18:47:00');
Query OK, 1 row affected (0.04 sec)

mysql> REPLACE INTO test VALUES (1, 'New', '2014-08-20 18:47:42');
Query OK, 2 rows affected (0.04 sec)

mysql> SELECT * FROM test;
+----+------+---------------------+
| id | data | ts                  |
+----+------+---------------------+
|  1 | New  | 2014-08-20 18:47:42 |
+----+------+---------------------+
1 row in set (0.00 sec)
```

现在我们创建第二个表与第一个几乎相同，除了主键现在覆盖 2 列，如下所示（强调文本）：

```
DROP TABLE IF EXISTS test2;
CREATE TABLE test2 (
       id INT UNSIGNED NOT NULL AUTO_INCREMENT,
       data VARCHAR(64) DEFAULT NULL,
      ts TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP ON UPDATE CURR
ENT_TIMESTAMP,
  PRIMARY KEY (id, ts)
);
```

当我们在 test2 上运行与在原始测试表上相同的两个 REPLACE 语句时，我们得到了不同的结果：

```
mysql> REPLACE INTO test2 VALUES (1, 'Old', '2014-08-20 18:47:00');
Query OK, 1 row affected (0.05 sec)

mysql> REPLACE INTO test2 VALUES (1, 'New', '2014-08-20 18:47:42');
Query OK, 1 row affected (0.06 sec)

mysql> SELECT * FROM test2;
+----+------+---------------------+
| id | data | ts                  |
+----+------+---------------------+
|  1 | Old  | 2014-08-20 18:47:00 |
|  1 | New  | 2014-08-20 18:47:42 |
+----+------+---------------------+
2 rows in set (0.00 sec)
```

这是因为在 test2 上运行时，id 和 ts 列值必须与要替换的行的现有行的值匹配； 否则，插入一行。

6.2.4 SELECT 语法

```
SELECT
      [ALL | DISTINCT | DISTINCTROW ]
       [HIGH_PRIORITY]
       [STRAIGHT_JOIN]
    [SQL_SMALL_RESULT] [SQL_BIG_RESULT] [SQL_BUFFER_RESULT]
    [SQL_NO_CACHE] [SQL_CALC_FOUND_ROWS]
    select_expr [, select_expr] ...
    [into_option]
    [FROM table_references
      [PARTITION partition_list]]
    [WHERE where_condition]
    [GROUP BY {col_name | expr | position}, ... [WITH ROLLUP]]
    [HAVING where_condition]
    [WINDOW window_name AS (window_spec)
        [, window_name AS (window_spec)] ...]
    [ORDER BY {col_name | expr | position}
      [ASC | DESC], ... [WITH ROLLUP]]
    [LIMIT {[offset,] row_count | row_count OFFSET offset}]
    [into_option]
    [FOR {UPDATE | SHARE}
        [OF tbl_name [, tbl_name] ...]
        [NOWAIT | SKIP LOCKED]
      | LOCK IN SHARE MODE]
```

SELECT 语句最常用的子句如下：

- 每个 select_expr 表示您要检索的列。 必须至少有一个 select_expr。
- table_references 指示要从中检索行的一个或多个表。
- SELECT 支持使用 PARTITION 子句的显式分区选择，其中包含 table_reference 中表名称后的分区或子分区（或两者）列表。 在这种情况下，仅从列出的分区中选择行，而忽略表的任何其他分区。
- WHERE 子句（如果给定）指示要选择的行必须满足的一个或多个条件。 where_condition 是一个表达式，对于要选择的每一行，它的计算结果为真。 如果没有 WHERE 子句，该语句将选择所有行。

在 WHERE 表达式中，您可以使用 TDSQL 支持的任何函数和运算符，聚合（组）函数除外。SELECT 也可用 于检索在不参考任何表的情况下计算的行。

例如：

```
MySQL [test]> select 1+1;
+-----+
| 1+1 |
+-----+
|   2 |
+-----+
1 row in set (0.00 sec)
```

在没有引用表的情况下，您可以将 DUAL 指定为虚拟表名：

```
MySQL [test]> SELECT 1 + 1 FROM DUAL;
+-------+
| 1 + 1 |
+-------+
|     2 |
+-------+
1 row in set (0.00 sec)
```

DUAL 纯粹是为了方便那些要求所有 SELECT 语句都应该有 FROM 和其他可能的子句的人。 TDSQL 可能会忽略这些子句。 如果没有引用表，TDSQL 不需要 FROM DUAL。

通常，使用的子句必须完全按照语法描述中显示的顺序给出。 例如，HAVING 子句必须在任何 GROUP BY 子句之后和任何 ORDER BY 子句之前。 INTO 子句（如果存在）可以出现在语法描述指示的任何位置，但在给定语句中只能出现一次，不能出现在多个位置。

select_expr 术语列表包含指示要检索哪些列的选择列表。 术语指定列或表达式或可以使用 *-简写：

- 仅包含一个非限定 * 的选择列表可用作从所有表中选择所有列的简写：

```
SELECT * FROM t1 INNER JOIN t2 ...
```

- tbl_name.* 可用作限定的速记以从命名表中选择所有列：

```
SELECT t1.*, t2.* FROM t1 INNER JOIN t2 ...
```

- 将不合格的 * 与选择列表中的其他项目一起使用可能会产生解析错误。 为避免此问题，请使用限定的 tbl_name.* 引用：

```
SELECT AVG(score), t1.* FROM t1 ...
```

以下列表提供了有关其他 SELECT 子句的附加信息：

- 可以使用 AS alias_name 为 select_expr 赋予一个别名。别名用作表达式的列名，可以在 GROUP BY、ORDER BY 或 HAVING 子句中使用。 例如：

```
SELECT CONCAT(last_name,', ',first_name) AS full_name
FROM mytable ORDER BY full_name;
```

当使用标识符为 select_expr 设置别名时，AS 关键字是可选的。但是，因为 AS 是可选的，如果您忘记了两个 select_expr 表达式之间的逗号，则会出现一个微妙的问题：TDSQL 将第二个解释为别名。 例如，在以下语句中，columnb 被视为别名：SELECT columna columnb FROM mytable;因此，在指定列别名时养成明确使用 AS 的习惯是一种很好的做法。

- 您可以使用 tbl_name 或 db_name.tbl_name 来引用默认数据库中的表以明确指定数据库。 您可以将列称为 col_name、tbl_name.col_name 或 db_name.tbl_name.col_name。 您不需要为列引用指定 tbl_name 或 db_name.tbl_name 前缀，除非引用不明确。

- 可以使用 tbl_name AS alias_name 或 tbl_name alias_name 为表引用设置别名。 这些语句是等效的：

```
SELECT t1.name，t2.salary FROM employee AS t1，info AS t2
 WHERE t1.name = t2.name;

SELECT t1.name，t2.salary FROM employee t1，info t2
 WHERE t1.name = t2.name;
```

- 可以使用列名、列别名或列位置在 ORDER BY 和 GROUP BY 子句中引用为输出选择的列。 列位置是整数并以 1 开头：

```
 SELECT college, region, seed FROM tournament
  ORDER BY region, seed;

SELECT college, region AS r, seed AS s FROM tournament
  ORDER BY r, s;
```

```
SELECT college, region, seed FROM tournament
 ORDER BY 2, 3;
```

要按相反顺序排序，请将 DESC（降序）关键字添加到排序依据的 ORDER BY 子句中的列名。 默认为升序； 这可以使用 ASC 关键字显式指定。不推荐使用列位置，因为该语法已从 SQL 标准中删除。

- TDSQL 支持 ORDER BY 和分组功能，这也意味着您可以在使用 GROUP BY 时对任意一列或多列进行排序，如下所示：

  ```
  SELECT a, b, COUNT(c) AS t FROM test_table GROUP BY a,b ORDER BY a,t DESC
  ```

- 当您使用 ORDER BY 或 GROUP BY 对 SELECT 中的列进行排序时，服务器仅使用 max_sort_length 系统变量指示的初始字节数对值进行排序。

- GROUP BY 允许使用 WITH ROLLUP 修饰符。

- HAVING 子句与 WHERE 子句一样，指定选择条件。 WHERE 子句指定选择列表中列的条件，但不能引用聚合函数。 HAVING 子句指定组的条件，通常由 GROUP BY 子句组成。 查询结果只包含满足 HAVING 条件的组。（如果不存在 GROUP BY，则所有行都隐式地形成一个聚合组。）HAVING 子句几乎最后应用，就在项目被发送到客户端之前，没有优化。（LIMIT 在 HAVING 之后应用。）SQL 标准要求 HAVING 必须仅引用 GROUP BY 子句中的列或聚合函数中使用的列。但是，TDSQL 支持此行为的扩展，并允许 HAVING 引用 SELECT 列表中的列和外部子查询中的列。如果 HAVING 子句引用不明确的列，则会出现警告。 在以下语句中， col2 不明确，因为它既用作别名又用作列名：

```
SELECT COUNT (col1) AS col2 FROM t GROUP BY col2 HAVING col2 = 2;
```

优先考虑标准 SQL 行为，因此如果在 GROUP BY 中使用 HAVING 列名并作为选择列列表中的别名列，则优先考虑 GROUP BY 列中的列。

- 不要对应该在 WHERE 子句中的项目使用 HAVING。例如，不要写以下内容：

```
SELECT col_name FROM tbl_nameHAVING col_name> 0;
```

改写这个：

```
SELECT col_name FROM tbl_nameWHERE col_name> 0;
```

- HAVING 子句可以引用聚合函数，而 WHERE 子句不能：

```
SELECT user, MAX(salary) FROM users
  GROUP BY user HAVING MAX(salary) > 10;
```

- TDSQL 允许重复的列名。 也就是说，可以有多个同名的 select_expr。 这是标准 SQL 的扩展。 因为 TDSQL 还允许 GROUP BY 和 HAVING 引用 select_expr 值，这可能会导致歧义：

```
SELECT 12 AS a，FROM t GROUP BY a;
```

在该语句中，两列的名称均为 a。 为确保使用正确的列进行分组，请为每个 select_expr 使用不同的名称。

- WINDOW 子句（如果存在）定义可以由窗口函数引用的命名窗口。

- TDSQL 通过在 select_expr 值中搜索，然后在 FROM 子句中的表的列中搜索来解析 ORDER BY 子句中的非限定列或别名引用。 对于 GROUP BY 或 HAVING 子句，它在搜索 select_expr 值之前先搜索 FROM 子句。

- LIMIT 子句可用于限制 SELECT 语句返回的行数。 LIMIT 接受一个或两个数字参数，它们都必须是非负整数常量，但以下情况除外：

- 在 prepare 的语句中，可以使用 ? 占位符。

- 在存储的程序中，可以使用整数值的例程参数或局部变量来指定 LIMIT 参数。

JOIN 语法

TDSQL 对 SELECT 语句和多表 DELETE 和 UPDATE 语句的 table_references 部分支持以下 JOIN 语法：

```
table_references:
      escaped_table_reference [, escaped_table_reference]
 ...

escaped_table_reference: {
    table_reference
  | { OJ table_reference }
}

table_reference: {
    table_factor
  | joined_table
}

table_factor: {
    tbl_name [PARTITION (partition_names)]
        [[AS] alias]
  | [LATERAL] table_subquery [AS] alias [(col_list)]
  | ( table_references )
}

joined_table: {
    table_reference {[INNER | CROSS] JOIN | STRAIGHT_JOIN} t
able_factor [join_specification]
  | table_reference {LEFT|RIGHT} [OUTER] JOIN table_referenc
e join_specification
  | table_reference NATURAL [INNER | {LEFT|RIGHT} [OUTER]] J
OIN table_factor
}

join_specification: {
    ON search_condition
  | USING (join_column_list)
}

join_column_list:
    column_name [, column_name] ...

index_list:
    index_name [, index_name] ...
```

表引用（当它引用分区表时）可能包含一个 PARTITION 子句，包括以逗号分隔的分区、子分区或两者的列表。 此选项跟在表的名称之后，并在任何别名声明之前。 此选项的效果是仅从列出的分区或子分区中选择行。 任何未在列表中命名的分区或子分区都将被忽略。

与标准 SQL 相比，TDSQL 中对 table_factor 的语法进行了扩展。该标准只接受 table_reference，而不是一对括号内的列表。

如果 table_reference 项列表中的每个逗号都被视为等效于内部联接，则这是一种保守的扩展。 例如：

```
SELECT * FROM t1 LEFT JOIN (t2，t3，t4)
        ON (t2.a = t1.a AND t3.b = t1.b AND t4.c = t1.c)
```

相当于：

```
SELECT * FROM t1 LEFT JOIN (t2 CROSS JOIN t3 CROSS JOIN
t4)
        ON (t2.a = t1.a AND t3.b = t1.b AND t4.c = t1.c)
```

在 TDSQL 中，JOIN、CROSS JOIN 和 INNER JOIN 在语法上是等价的（它们可以相互替换）。 在标准 SQL 中，它们不是等价的。 INNER JOIN 与 ON 子句一起使用，否则使用 CROSS JOIN。

通常，在仅包含内部连接操作的连接表达式中可以忽略括号。TDSQL 还支持嵌套连接。

以下列表描述了在编写连接时要考虑的一般因素：

- 可以使用 tbl_name AS alias_name 或 tbl_name alias_name 为表引用设置别名：

```
SELECT t1.name，t2.salary
 FROM employee AS t1 INNER JOIN info AS t2 ON t1.name =
t2.name;

SELECT t1.name，t2.salary
 FROM employee t1 INNER JOIN info t2 ON t1.name = t2.name;
```

- table_subquery 在 FROM 子句中也称为派生表或子查询。 此类子查询必须包含一个别名，以便为子查询结果提供表名，并且可以选择在括号中包含表列名列表。 一个简单的例子如下：

```
SELECT * FROM (SELECT 1,2,3) AS t1;
```

- 在没有连接条件的情况下，INNER JOIN 和 ,（逗号）在语义上是等效的：两者都在指定的表之间产生笛卡尔积（即，第一个表中的每一行都连接到第二个表中的每一行 ）。

但是，逗号运算符的优先级低于 INNER JOIN、CROSS JOIN、LEFT JOIN 等。 如果在存在连接条件时将逗号连接与其他连接类型混合使用，则可能会出现“on 子句”中的“未知列”列“col_name”形式的错误。

- 与 ON 一起使用的 search_condition 是可以在 WHERE 子句中使用的形式的任何条件表达式。 通常，ON 子句用于指定如何连接表的条件，而 WHERE 子句限制将哪些行包含在结果集中。

- 如果在 LEFT JOIN 的 ON 或 USING 部分中没有与右表匹配的行，则将所有列都设置为 NULL 的行用于右表。您可以使用此事实来查找表中在另一个表中没有对应项的行：

```
SELECT  left_tbl.*
  FROM left_tbl LEFT JOIN right_tbl ON left_tbl.id = right_tbl.id
  WHERE right_tbl.id IS NULL;
```

- USING(join_column_list) 子句命名必须存在于两个表中的列列表。 如果表 a 和 b 都包含列 c1、c2 和 c3，则以下连接比较两个表中的相应列：

```
a LEFT JOIN b USING (c1, c2, c3)
```

一些连接示例：

```
SELECT * FROM table1，table2;

SELECT * FROM table1 INNER JOIN table2 ON table1.id =
table2.id;

SELECT * FROM table1 LEFT JOIN table2 ON table1.id =
table2.id;

SELECT * FROM table1 LEFT JOIN table2 USING (id);

SELECT * FROM table1 LEFT JOIN table2 ON table1.id =
table2.id
  LEFT JOIN table3 ON table2.id = table3.id;
```

自然连接和使用 USING 的连接，包括外连接变体，根据 SQL2003 标准进行处理：

- NATURAL 连接的冗余列不会出现。 考虑这组语句：

```
CREATE TABLE t1 (i INT primary key, j INT);
CREATE TABLE t2 (k INT primary key, j INT);
INSERT INTO t1 VALUES(1, 1);
INSERT INTO t2 VALUES(1, 1);
SELECT * FROM t1 NATURAL JOIN t2;
SELECT * FROM t1 JOIN t2 USING (j);
```

在第一个 SELECT 语句中，列 j 出现在两个表中，因此成为连接列，因此，根据标准 SQL，它应该在输出中只出现一次，而不是两次。 同样，在第二个 SELECT 语句中，列 j 在 USING 子句中命名，并且在输出中应该只出现一次，而不是两次。

因此，语句产生以下输出：

```
+------+------+------+
| j    | i    | k    |
+------+------+------+
|    1 |    1 |    1 |
+------+------+------+
+------+------+------+
| j    | i    | k    |
+------+------+------+
|    1 |    1 |    1 |
+------+------+------+
```

根据标准 SQL 进行冗余列消除和列排序，产生以下显示顺序：

- 首先，按照它们在第一个表中出现的顺序合并两个连接表的公共列
- 其次，第一个表独有的列，按照它们在该表中出现的顺序
- 第三，第二个表独有的列，按照它们在该表中出现的顺序

使用合并操作定义替换两个公共列的单个结果列。 也就是说，对于两个 t1.a 和 t2.a，生成的单个连接列 a 定义为 a = COALESCE(t1.a, t2.a)，其中：

```
COALESCE(x, y) = (CASE WHEN x IS NOT NULL THEN x
ELSE y END)
```

如果连接操作是任何其他连接，则连接的结果列由连接表的所有列的串联组成。

合并列的定义的结果是，对于外连接，如果两列之一始终为 NULL，合并列包含非 NULL 列的值。 如果两个列都不为 NULL，则两个公共列具有相同的值，因此选择哪一个作为合并列的值并不重要。 解释这一点的一种简单方法是考虑外部联接的合并列由 JOIN 的内部表的公共列表示。 假设表 t1(a, b) 和 t2(a, c) 的内容如下：

```
t1   t2
---- ----
1 x  2 z
2 y  3 w
```

然后，对于此连接，列 a 包含以下值 t1.a：

```
mysql> SELECT * FROM t1 NATURAL LEFT JOIN t2;
+------+------+------+
| a    | b    | c    |
+------+------+------+
|    1 | x    | NULL |
|    2 | y    | z    |
+------+------+------+
```

相反，对于此连接，列 a 包含值 t2.a 。

```
mysql> SELECT * FROM t1 NATURAL RIGHT JOIN t2;
+------+------+------+
| a    | c    | b    |
+------+------+------+
|    2 | z    | y    |
|    3 | w    | NULL |
+------+------+------+
```

将这些结果与其他等效查询进行比较 JOIN ... ON：

```
mysql> SELECT * FROM t1 LEFT JOIN t2 ON (t1.a = t2.a);
+------+------+------+------+
| a    | b    | a    | c    |
+------+------+------+------+
|    1 | x    | NULL | NULL |
|    2 | y    |    2 | z    |
+------+------+------+------+
```

```
mysql> SELECT * FROM t1 RIGHT JOIN t2 ON (t1.a = t2.a);
+------+------+------+------+
| a    | b    | a    | c    |
+------+------+------+------+
|    2 | y    |    2 | z    |
| NULL | NULL |    3 | w    |
+------+------+------+------+
```

- USING 子句可以重写为比较相应列的 ON 子句。 但是，虽然 USING 和 ON 很相似，但它们并不完全相同。 考虑以下两个查询：

```
a LEFT JOIN b USING (c1, c2, c3)
a LEFT JOIN b ON a.c1 = b.c1 AND a.c2 = b.c2 AND a.c3 = b.c3
```

关于确定哪些行满足连接条件，两个连接在语义上是相同的。

关于确定为 SELECT * 扩展显示哪些列，这两个连接在语义上并不相同。 USING 连接选择相应列的合并值，而 ON 连接选择所有表中的所有列。 对于 USING 连接，SELECT * 选择以下值：

```
COALESCE(a.c1, b.c1), COALESCE(a.c2, b.c2),
COALESCE(a.c3, b.c3)
```

对于 ON 连接，SELECT * 选择以下值：

```
a.c1，a.c2，a.c3，b.c1，b.c2，b.c3
```

对于内部联接，COALESCE(a.c1, b.c1) 与 a.c1 或 b.c1 相同，因为两列具有相同的值。 使用外连接（例如 LEFT JOIN），两列之一可以为 NULL。 结果中省略了该列。

- ON 子句只能引用其操作数

示例：

```
drop table if exists t1,t2,t3;

CREATE TABLE t1 (i1 INT primary key);
CREATE TABLE t2 (i2 INT primary key);
CREATE TABLE t3 (i3 INT primary key);
SELECT * FROM t1 JOIN t2 ON (i1 = i3) JOIN t3;
```

该语句失败并在“on 子句”错误中出现未知列“i3”，因为 i3 是 t3 中的一个列，它不是 ON 子句的操作数。 要使连接能够被处理，请按如下方式重写语句：

```
SELECT * FROM t1 JOIN t2 JOIN t3 ON (i1 = i3);
```

- JOIN 的优先级高于逗号运算符 (,)，因此连接表达式 t1, t2 JOIN t3 被解释为 (t1, (t2 JOIN t3))，而不是 ((t1, t2) JOIN t3)。 这会影响使用 ON 子句的语句，因为该子句只能引用连接操作数中的列，并且优先级会影响对这些操作数是什么的解释。

示例：

```
drop table if exists t1,t2,t3;

CREATE TABLE t1 (i1 INT primary key,j1 INT);
CREATE TABLE t2 (i2 INT primary key,j2 INT);
CREATE TABLE t3 (i3 INT primary key,j3 INT);
INSERT INTO t1 VALUES(1,1);
INSERT INTO t2 VALUES(1,1);
INSERT INTO t3 VALUES(1,1);
SELECT * FROM t1,t2 JOIN t3 ON (t1.i1 = t3.i3);
```

JOIN 优先于逗号运算符，因此 ON 子句的操作数是 t2 和 t3。 由于 t1.i1 不是任一操作数中的列，因此结果是“on 子句”错误中的未知列“t1.i1”。

要处理连接，请使用以下任一策略：

- 用括号将前两个表显式分组，以便 ON 子句的操作数是 (t1, t2) 和 t3：

```
SELECT * FROM (t1, t2) JOIN t3 ON (t1.i1 =
t3.i3);
```

- 避免使用逗号运算符并使用 JOIN 代替：

```
SELECT * FROM t1 JOIN t2 JOIN t3 ON (t1.i1 =
t3.i3);
```

优先级相同的解释也适用于与混合逗号操作语句 INNER JOIN， CROSS JOIN， LEFT JOIN，并且 RIGHT JOIN，所有这些都具有比逗号操作符更高的优先级。

UNION 语法

语法如下：

```
SELECT ...
UNION [ALL | DISTINCT] SELECT ...
[UNION [ALL | DISTINCT] SELECT ...]
```

UNION 将来自多个 SELECT 语句的结果组合到一个结果集中。 例子：

```
MySQL [test]> SELECT 1, 2;
+---+---+
| 1 | 2 |
+---+---+
| 1 | 2 |
+---+---+
1 row in set (0.00 sec)

MySQL [test]> SELECT 'a', 'b';
+---+---+
| a | b |
+---+---+
| a | b |
+---+---+
1 row in set (0.00 sec)

MySQL [test]> SELECT 1, 2 UNION SELECT 'a', 'b';
+---+---+
| 1 | 2 |
+---+---+
| 1 | 2 |
| a | b |
+---+---+
```

```
2 rows in set (0.01 sec)
```

结果集列名和数据类型

UNION 结果集的列名取自第一个 SELECT 语句的列名。

每个 SELECT 语句对应位置列出的选定列应具有相同的数据类型。 例如，第一个语句选择的第一列应该与其他语句选择的第一列具有相同的类型。 如果对应的 SELECT 列的数据类型不匹配，则 UNION 结果中列的类型和长度会考虑所有 SELECT 语句检索到的值。 例如，考虑以下情况，其中列长度不受第一个 SELECT 值长度的限制：

```
MySQL [test]> SELECT REPEAT('a',1) UNION SELECT
REPEAT('b',20);

+----------------------+

| REPEAT('a',1)      |

+----------------------+

| a                  |

| bbbbbbbbbbbbbbbbbbbb |

+----------------------+

2 rows in set (0.02 sec)
```

UNION DISTINCT 和 UNION ALL

默认情况下，将从 UNION 结果中删除重复的行。 可选的 DISTINCT 关键字具有相同的效果，但使其明确。 使用可选的 ALL 关键字，不会发生重复行删除，结果包括来自所有 SELECT 语句的所有匹配行。

您可以在同一查询中混合使用 UNION ALL 和 UNION DISTINCT。 混合 UNION 类型的处理方式是 DISTINCT 联合覆盖其左侧的任何 ALL 联合。 可以通过使用 UNION DISTINCT 显式生成 DISTINCT 联合，也可以通过使用不带 DISTINCT 或 ALL 关键字的 UNION 隐式生成。

6.2.5 子查询语法

子查询是 SELECT 另一个语句中的语句。

支持 SQL 标准要求的所有子查询形式和操作，以及一些特定于 TDSQL 的功能。

下面是一个子查询的示例：

```
SELECT * FROM t1 WHERE column1 in (SELECT column1 FROM t2);
```

注意事项：通常不推荐使用子查询，一般需要将子查询修改为 join 或者进行优化改写。

子查询错误

有些错误仅适用于子查询。 本节介绍它们。

- 不支持的子查询语法：

```
ERROR 1235 (ER_NOT_SUPPORTED_YET)
SQLSTATE = 42000
Message = "This version of MySQL doesn't yet
support
'LIMIT & IN/ALL/ANY/SOME subquery'"
```

这意味着 TDSQL 不支持以下形式的语句：

```
SELECT * FROM t1 WHERE s1 IN (SELECT s2 FROM t2
ORDER BY s1 LIMIT 1)
```

- 来自子查询的列数不正确：

```
ERROR 1241 (ER_OPERAND_COL)
SQLSTATE = 21000
Message = "Operand should contain 1 column(s)"
```

在这种情况下会发生此错误：

```
SELECT (SELECT column1，column2 FROM t2) FROM t1;
```

如果目的是行比较，您可以使用返回多列的子查询。 在其他上下文中，子查询必须是标量操作数

- 子查询中的行数不正确：

```
ERROR 1242 (ER_SUBSELECT_NO_1_ROW)
SQLSTATE = 21000
Message = "Subquery returns more than 1 row"
```

对于子查询必须最多返回一行但返回多行的语句，会发生此错误。 考虑以下示例：

```
SELECT * FROM t1 WHERE column1 = (SELECT column1
FROM t2);
```

如果 SELECT column1 FROM t2 只返回一行，则前面的查询有效。 如果子查询返回多于一行，则会出现错误 1242。 在这种情况下，查询应改写为：

```
SELECT * FROM t1 WHERE column1 = ANY(SELECT
column1 FROM t2);
```

- 子查询中错误使用的表：

```
Error 1093 (ER_UPDATE_TABLE_USED)
SQLSTATE = HY000
Message = "You can't specify target table 'x'
for update in FROM clause"
```

在以下情况下会发生此错误，该情况尝试修改表并从子查询中的同一表中进行选择：

```
UPDATE t1 SET column2 = (SELECT MAX(column1) FROM
t1);
```

您可以使用公用表表达式来解决此问题。

优化子查询

开发正在进行中，因此从长远来看，没有优化技巧可靠。 以下列表提供了一些您可能想要使用的有趣技巧。

- 将子句从外部移动到子查询内部。 例如，使用这个查询：

```
SELECT * FROM t1
  WHERE s1 IN (SELECT s1 FROM t1 UNION ALL SELECT
s1 FROM t2);
```

而不是这个：

```
SELECT * FROM t1
  WHERE s1 IN (SELECT s1 FROM t1) OR s1 IN
(SELECT s1 FROM t2);
```

再举一个例子，使用这个查询：

```
SELECT (SELECT column1 + 5 FROM t1) FROM t2;
```

而不是这个查询：

```
SELECT (SELECT column1 FROM t1) + 5 FROM t2;
```

- 使用行子查询而不是相关子查询。 例如，使用此查询：

```
SELECT * FROM t1
  WHERE (column1，column2) IN (SELECT column1，
column2 FROM t2);
```

而不是这个查询：

```
SELECT * FROM t1
  WHERE EXISTS (SELECT * FROM t2 WHERE t2.column1
= t1.column1
                AND t2.column2 = t1.column2);
```

这些技巧可能会导致程序变得更快或更慢。 使用像 BENCHMARK() 函数 这样的工具，你可以了解在你自己的情况下有什么帮助。

将子查询重写为连接

有时，除了使用子查询之外，还有其他方法可以测试一组值中的成员资格。 此外，在某些情况下，不仅可以在没有子查询的情况下重写查询，但使用其中一些技术而不是使用子查询可能更有效。 其中一个是 IN() 构造：

例如，这个查询：

```
SELECT * FROM t1 WHERE id IN (SELECT id FROM t2);
```

可以改写为：

```
SELECT DISTINCT t1.* FROM t1，t2 WHERE t1.id =
t2.id;
```

查询：

```
SELECT * FROM t1 WHERE id NOT IN (SELECT id FROM
t2);
SELECT * FROM t1 WHERE NOT EXISTS (SELECT id FROM
t2 WHERE t1.id = t2.id);
```

可以改写为：

```
SELECT table1.*
        FROM table1 LEFT JOIN table2 ON table1.id
= table2.id
   WHERE table2.id IS NULL;
```

A LEFT [OUTER] JOIN 可以比等效的子查询更快，因为服务器可能能够更好地优化它 - 这一事实并非仅针对 TDSQL 服务器。 在 SQL-92 之前，外连接不存在，因此子查询是执行某些操作的唯一方法。

6.2.6 UPDATE 语法

UPDATE 是一个修改表中行的 DML 语句。

单表语法：

```
UPDATE [IGNORE] table_reference
    SET assignment_list
    [WHERE where_condition]
    [ORDER BY ...]
    [LIMIT row_count]

value:
    {expr | DEFAULT}

assignment:
    col_name = value

assignment_list:
    assignment [, assignment] ...
```

多表语法：

```
UPDATE [IGNORE] table_references
SET assignment_list
[WHERE where_condition]
```

对于单表语法，UPDATE 语句使用新值更新命名表中现有行的列。SET 子句指示要修改的列以及应该给它们的值。每个值都可以作为表达式给出，或者使用关键字 DEFAULT 将列显式设置为其默认值。WHERE 子句（如果给定）指定标识要更新哪些行的条件。如果没有 WHERE 子句，则更新所有行。如果指定了 ORDER BY 子句，则按指定的顺序更新行。 LIMIT 子句限制了可以更新的行数。

对于多表语法，UPDATE 更新 table_references 中命名的每个表中满足条件的行。每个匹配行都会更新一次，即使它多次匹配条件。对于多表语法，不能使用 ORDER BY 和 LIMIT。

对于分区表，该语句的单表和多表形式都支持使用 PARTITION 子句作为表引用的一部分。此选项采用一个或多个分区或子分区（或两者）的列表。只检查列出的分区（或子分区）是否匹配，并且不更新任何不在这些分区或子分区中的行，无论它是否满足 where_condition。

注意：

与将 PARTITION 与 INSERT 或 REPLACE 语句一起使用的情况不同，即使列出的分区（或子分区）中没有行与 where_condition 匹配，否则有效的 UPDATE ... PARTITION 语句也被认为是成功的。

where_condition 是一个表达式，对于要更新的每一行，它的计算结果为真。

仅对实际更新的 UPDATE 中引用的列才需要 UPDATE 权限。 对于已读取但未修改的任何列，您只需要 SELECT 权限。

如果在表达式中访问要更新的表中的列，UPDATE 将使用该列的当前值。 例如，以下语句将 col1 设置为比其当前值大 1：

```
UPDATE t1 SET col1 = col1 + 1;
```

以下语句中的第二个赋值将 col2 设置为当前（更新后的） col1 值，而不是原始 col1 值。 结果是 col1 和 col2 具有相同的值。 此行为不同于标准 SQL。

```
UPDATE t1 SET col1 = col1 + 1,col2 = col1;
```

单表 UPDATE 分配通常从左到右进行评估。对于多表更新，不能保证按任何特定顺序执行分配。

如果您将一列设置为它当前拥有的值，TDSQL 会注意到这一点并且不会更新它。

如果通过设置为 NULL 更新已声明为 NOT NULL 的列，则在启用严格 SQL 模式时会发生错误；否则，列被设置为列数据类型的隐式默认值，并且警告计数增加。数字类型的隐式默认值为 0，字符串类型为空字符串 ('')，日期和时间类型为“零”值。

如果显式更新生成的列，则唯一允许的值是 DEFAULT。

您可以使用 LIMIT row_count 来限制 UPDATE 的范围。 LIMIT 子句是行匹配限制。只要找到满足 WHERE 子句的 row_count 行，该语句就会立即停止，无论它们是否实际发生了更改。

如果 UPDATE 语句包含 ORDER BY 子句，则按该子句指定的顺序更新行。这在某些可能导致错误的情况下很有用。假设表 t 包含具有唯一索引的列 id。以下语句可能会因重复键错误而失败，具体取决于行更新的顺序：

```
UPDATE t SET id = id + 1;
```

例如，如果表的 id 列中包含 1 和 2，并且在 2 更新为 3 之前将 1 更新为 2，则会发生错误。 为了避免这个问题，添加一个 ORDER BY 子句，使具有较大 id 值的行在具有较小值的行之前更新：

```
UPDATE t SET id = id + 1 ORDER BY id DESC;
```

您还可以执行涵盖多个表的 UPDATE 操作。 但是，不能将 ORDER BY 或 LIMIT 用于多表 UPDATE。

table_references 子句列出了连接中涉及的表。 下面是一个例子：

```
UPDATE  items,month SET items.price=month.price
WHERE items.id=month.id;
```

6.3 事务控制语法

TDSQL 支持通过语句，如本地事务（一个给定的客户端会话内） SET autocommit， START TRANSACTION， COMMIT，和 ROLLBACK 。

6.3.1 START TRANSACTION，COMMIT 和 ROLLBACK 语法

语法如下：

```
START TRANSACTION
      [transaction_characteristic [, transaction_characteristi
c] ...]

transaction_characteristic: {
      WITH CONSISTENT SNAPSHOT
      | READ WRITE
       | READ ONLY
}

BEGIN [WORK]
COMMIT [WORK] [AND [NO] CHAIN] [[NO] RELEASE]
ROLLBACK [WORK] [AND [NO] CHAIN] [[NO] RELEASE]
SET autocommit = {0 | 1}
```

这些语句提供对事务使用的控制：

- START TRANSACTION 或 BEGIN 开始新的事务。
- COMMIT 提交当前事务，使其更改永久化。
- ROLLBACK 回滚当前事务，取消其更改。
- SET autocommit 禁用或启用当前会话的默认自动提交模式。

默认情况下，TDSQL 在启用自动提交模式的情况下运行。 这意味着，当不在事务内部时，每个语句都是原子的，就好像它被 START TRANSACTION 和 COMMIT 包围一样。 您不能使用 ROLLBACK 来撤消效果； 但是，如果在语句执行过程中发生错误，则该语句将被回滚。

要为一系列语句隐式禁用自动提交模式，请使用 START TRANSACTION 语句：

```
START TRANSACTION;
SELECT @A:=SUM(salary) FROM table1 WHERE type=1;
UPDATE table2 SET summary=@A WHERE type=1;
COMMIT;
```

使用 START TRANSACTION，自动提交将保持禁用状态，直到您使用 COMMIT 或 ROLLBACK 结束事务。自动提交模式然后恢复到以前的状态。

START TRANSACTION 允许使用几个控制事务特性的修饰符。要指定多个修饰符，请用逗号分隔它们。

- WITH CONSISTENT SNAPSHOT 修饰符为有能力的存储引擎启动一致读取。这仅适用于 InnoDB。效果与从任何 InnoDB 表发出 START TRANSACTION 后跟 SELECT 相同。WITH CONSISTENT SNAPSHOT 修饰符不会更改当前事务隔离级别，因此仅当当前隔离级别是允许一致读取的隔离级别时，它才提供一致快照。允许一致读取的唯一隔离级别是可重复读取。对于所有其他隔离级别，WITH CONSISTENT SNAPSHOT 子句将被忽略。忽略 WITH CONSISTENT SNAPSHOT 子句时会生成警告。

- READ WRITE 和 READ ONLY 修饰符设置事务访问模式。它们允许或禁止更改事务中使用的表。 READ ONLY 限制防止事务修改或锁定其他事务可见的事务表和非事务表；事务仍然可以修改或锁定临时表。

当已知事务为只读时，TDSQL 可以对 InnoDB 表上的查询进行额外优化。指定 READ ONLY 可确保在无法自动确定只读状态的情况下应用这些优化。

如果未指定访问模式，则应用默认模式。除非默认值已更改，否则为读/写。不允许在同一语句中同时指定 READ WRITE 和 READ ONLY。

在只读模式下，仍然可以使用 DML 语句更改使用 TEMPORARY 关键字创建的表。不允许使用 DDL 语句进行更改，就像使用永久表一样。

如果启用了 read_only 系统变量，则使用 START TRANSACTION READ WRITE 显式启动事务需要 CONNECTION_ADMIN 权限（或已弃用的 SUPER 权限）。

> 重要：
>
> 许多用于编写 TDSQL 客户端应用程序（例如 JDBC）的 API 提供了自己的方法来启动可以（有时应该）使用的事务，而不是从客户端发送 START TRANSACTION 语句。

要显式禁用自动提交模式，请使用以下语句：

SET autocommit=0;

通过将自动提交变量设置为零禁用自动提交模式后，对事务安全表（例如 InnoDB 表）的更改不会立即永久生效。 您必须使用 COMMIT 将更改存储到磁盘或 ROLLBACK 以忽略更改。

autocommit 是一个会话变量，必须为每个会话设置。

支持 BEGIN 和 BEGIN WORK 作为用于启动事务的 START TRANSACTION 的别名。 START TRANSACTION 是标准 SQL 语法，是启动临时事务的推荐方式，并允许 BEGIN 不允许的修饰符。

BEGIN 语句不同于开始 BEGIN ... END 复合语句的 BEGIN 关键字的使用。 后者不开始交易。

注意：

在所有存储程序（存储过程和函数、触发器和事件）中，解析器将 BEGIN [WORK] 视为 BEGIN ... END 块的开始。 在此上下文中使用 START TRANSACTION 开始事务。

COMMIT 和 ROLLBACK 支持可选的 WORK 关键字，CHAIN 和 RELEASE 子句也是如此。 CHAIN 和 RELEASE 可用于对事务完成进行额外控制。 completion_type 系统变量的值决定了默认的完成行为。

AND CHAIN 子句会导致新事务在当前事务结束时立即开始，并且新事务与刚刚终止的事务具有相同的隔离级别。新事务也使用与刚刚终止的事务相同的访问模式（READ WRITE 或 READ ONLY）。RELEASE 子句使服务器在终止当前事务后断开当前客户端会话。包含 NO 关键字会抑制 CHAIN 或 RELEASE 完成，如果默认情况下将 completion_type 系统变量设置为导致链接或释放完成，这会很有用。

开始事务会导致提交任何挂起的事务。

开始一个事务也会导致通过 LOCK TABLES 获得的表锁被释放，就好像你已经执行了 UNLOCK TABLES 一样。开始事务不会释放通过 FLUSH TABLES WITH READ LOCK 获得的全局读锁。

为获得最佳结果，应仅使用由单个事务安全存储引擎管理的表来执行事务。 否则，可能会出现以下问题：

- 如果您使用来自多个事务安全存储引擎（例如 InnoDB）的表，并且事务隔离级别不是 SERIALIZABLE，那么当一个事务提交时，另一个使用相同表的正在进行的事务可能只看到其中的一些第一个事务所做的更改。 也就是说，混合引擎不能保证事务的原子性，并且可能导致不一致。 （如果混合引擎事务不频繁，您可以根据需要使用 SET TRANSACTION ISOLATION LEVEL 在每个事务的基础上将隔离级别设置为 SERIALIZABLE。）
- 如果您在事务中使用非事务安全的表，则对这些表的更改会立即存储，无论自动提交模式的状态如何。
- 如果在更新事务中的非事务性表后发出 ROLLBACK 语句，则会出现 ER_WARNING_NOT_COMPLETE_ROLLBACK 警告。 对事务安全表的更改会回滚，但不会对非事务安全表的更改进行回滚。

在 COMMIT 时，每个事务都以一个块存储在二进制日志中。 不记录回滚的事务。（例外：对非事务表的修改不能回滚。如果被回滚的事务包括对非事务表的修改，整个事务将在末尾使用 ROLLBACK 语句记录，以确保复制对非事务表的修改。）

您可以使用 SET TRANSACTION 语句更改事务的隔离级别或访问模式。

回滚可能是一个缓慢的操作，可能会在没有用户明确要求的情况下隐式发生（例如，发生错误时）。 因此，SHOW PROCESSLIST 在会话的状态列中显示回滚，不仅用于使用 ROLLBACK 语句执行的显式回滚，还用于隐式回滚。

当 InnoDB 执行一个事务的完全回滚时，该事务设置的所有锁都会被释放。 如果事务中的单个 SQL 语句由于错误（例如重复键错误）而回滚，则在事务保持活动状态时保留语句设置的锁。 发生这种情况是因为 InnoDB 以一种格式存储行锁，以至于它之后无法知道哪个语句设置了哪个锁。

如果事务中的 SELECT 语句调用存储函数，并且存储函数中的语句失败，则该语句回滚。 如果随后对该事务执行 ROLLBACK，则整个事务将回滚。

6.3.2 无法回滚的陈述

有些语句不能回滚。 通常，这些包括数据定义语言 (DDL) 语句，例如创建或删除数据库的语句，创建、删除或更改表或存储例程的语句。

你应该设计你的交易不包括这样的陈述。 如果您在无法回滚的事务中早期发出一条语句，然后另一条语句随后失败，则在这种情况下，无法通过发出 ROLLBACK 语句来回滚事务的全部效果。

6.3.3 导致隐式提交的语句

本节中列出的语句（以及它们的任何同义词）隐式结束当前会话中的任何活动事务，就好像您在执行该语句之前已完成 COMMIT 一样。

大多数这些语句在执行后也会导致隐式提交。 目的是在它自己的特殊事务中处理每个这样的语句。 事务控制和锁定语句是例外：如果隐式提交在执行之前发生，另一个不会在执行之后发生。

- **定义或修改数据库对象的数据定义语言（DDL）语句。**

  ALTER EVENT， ALTER FUNCTION，ALTER PROCEDURE， ALTER SERVER，ALTER TABLE， ALTER VIEW， CREATE DATABASE， CREATE EVENT， CREATE FUNCTION， CREATE INDEX， CREATE PROCEDURE， CREATE ROLE， CREATE SERVER， CREATE SPATIAL REFERENCESYSTEM， CREATE TABLE，CREATE TRIGGER， CREATE VIEW， DROP DATABASE， DROP EVENT， DROP FUNCTION， DROP INDEX， DROP PROCEDURE， DROP ROLE， DROP SERVER，DROP SPATIAL REFERENCE SYSTEM，DROP TABLE， DROP TRIGGER， DROP VIEW， INSTALL PLUGIN， RENAME TABLE， TRUNCATE TABLE， UNINSTALL PLUGIN 。

  如果使用 TEMPORARY 关键字，CREATE TABLE 和 DROP TABLE 语句不会提交事

务。 （这不适用于临时表上的其他操作，例如 ALTER TABLE 和 CREATE INDEX，它们确实会导致提交。）但是，虽然没有发生隐式提交，但语句也不能回滚，这意味着使用此类语句 导致违反事务原子性。例如，如果您使用 CREATE TEMPORARY TABLE 然后回滚事务，该表仍然存在。

InnoDB 中的 CREATE TABLE 语句作为单个事务处理。 这意味着来自用户的 ROLLBACK 不会撤消用户在该事务期间所做的 CREATE TABLE 语句。

当您创建非临时表时，CREATE TABLE ... SELECT 会在执行语句之前和之后导致隐式提交。 （CREATE TEMPORARY TABLE ... SELECT 不会发生提交。）

- **隐式使用或修改 mysql 数据库中的表的语句 。**

```
ALTER USER ， CREATE USER ， DROP USER ，
GRANT ， RENAME USER ， REVOKE ， SET PASSWORD
```

- **事务控制和锁定语句。** BEGIN， LOCK TABLES， SET autocommit = 1（如果该值不是 1）。 START TRANSACTION UNLOCK TABLES

  仅当任何表当前已使用 LOCK TABLES 锁定以获取非事务表锁时，UNLOCK TABLES 才会提交事务。 在 FLUSH TABLES WITH READ LOCK 之后的 UNLOCK TABLES 不会发生提交，因为后一个语句不获取表级锁。

  事务不能嵌套。 这是在您发出 START TRANSACTION 语句或其同义词之一时对任何当前事务执行的隐式提交的结果。

  当事务处于 ACTIVE 状态时，不能在 XA 事务中使用导致隐式提交的语句。

  BEGIN 语句不同于开始 BEGIN ... END 复合语句的 BEGIN 关键字的使用。 后者不会导致隐式提交。

- **管理语句。** ANALYZE TABLE， CACHE INDEX，CHECK TABLE， FLUSH， LOAD INDEX INTO CACHE， OPTIMIZE TABLE， REPAIR TABLE，RESET （但不是 RESET PERSIST ）
- **复制控制语句** 。 START SLAVE， STOP SLAVE，RESET SLAVE， CHANGE MASTER TO

6.3.4 SAVEPOINT，ROLLBACK 到 SAVEPOINT 和 RELEASE SAVEPOINT 语法

语法如下：

```
SAVEPOINT identifier
ROLLBACK [WORK] TO [SAVEPOINT] identifier
RELEASE SAVEPOINT identifier
```

InnoDB 支持 SQL 语句 SAVEPOINT、ROLLBACK TO SAVEPOINT、RELEASE SAVEPOINT 和 ROLLBACK 的可选 WORK 关键字。

SAVEPOINT 语句使用标识符名称设置命名事务保存点。如果当前事务有一个同名的保存点，则删除旧的保存点并设置一个新的保存点。

ROLLBACK TO SAVEPOINT 语句将事务回滚到指定的保存点而不终止事务。当前事务在设置保存点后对行所做的修改在回滚中被撤销，但 InnoDB 不会释放保存点后存储在内存中的行锁。 （对于新插入的行，锁信息由存储在行中的事务 ID 携带；锁不单独存储在内存中。在这种情况下，行锁在 undo 中释放。）晚于指定的保存点被删除。

如果 ROLLBACK TO SAVEPOINT 语句返回以下错误，则表示不存在具有指定名称的保存点：

```
ERROR 1305 (42000): SAVEPOINT identifier does not exist
```

RELEASE SAVEPOINT 语句从当前事务的保存点集中删除指定的保存点。 不会发生提交或回滚。 如果保存点不存在，则会出错。

如果您执行 COMMIT 或未命名保存点的 ROLLBACK，则会删除当前事务的所有保存点。

当调用存储的函数或激活触发器时，会创建一个新的保存点级别。先前级别上的保存点变得不可用，因此不会与新级别上的保存点冲突。当函数或触发器终止时，它创建的任何保存点都将被释放并恢复先前的保存点级别。

6.3.5 SET TRANSACTION 语法

语法如下：

```
SET [GLOBAL | SESSION] TRANSACTION
   transaction_characteristic [, transaction_characteristic] ...

transaction_characteristic: {
     ISOLATION LEVEL level
   | access_mode
}

level: {
     REPEATABLE READ
   | READ COMMITTED
   | READ UNCOMMITTED
  | SERIALIZABLE
}

access_mode: {
     READ WRITE
   | READ ONLY
}
```

此语句指定事务特性。 它需要一个由逗号分隔的一个或多个特征值的列表。 每个特征值设置事务隔离级别或访问模式。 隔离级别用于对 InnoDB 表的操作。 访问模式指定事务是以读/写还是只读模式运行。

此外，SET TRANSACTION 可以包含一个可选的 GLOBAL 或 SESSION 关键字来指示语句的范围。

- 事务隔离级别
- 交易访问模式
- 交易特征范围

事务隔离级别

要设置事务隔离级别，请使用 ISOLATION LEVEL 级别子句。 不允许在同一个 SET TRANSACTION 语句中指定多个 ISOLATION LEVEL 子句。

默认隔离级别是 REPEATABLE READ。 其他允许的值是 READ COMMITTED、READ UNCOMMITTED 和 SERIALIZABLE。

事务访问模式

要设置事务访问模式，请使用 READ WRITE 或 READ ONLY 子句。 不允许在同一个 SET TRANSACTION 语句中指定多个访问模式子句。

默认情况下，事务以读/写模式进行，允许对事务中使用的表进行读和写。 此模式可以使用 SET TRANSACTION 和 READ WRITE 访问模式显式指定。

如果事务访问模式设置为 READ ONLY，则禁止更改表。 这可以使存储引擎能够在不允许写入时进行性能改进。

在只读模式下，仍然可以使用 DML 语句更改使用 TEMPORARY 关键字创建的表。 不允许使用 DDL 语句进行更改，就像使用永久表一样。

还可以使用 START TRANSACTION 语句为单个事务指定 READ WRITE 和 READ ONLY 访问模式。

事务特征范围

您可以为当前会话或仅为下一个事务全局设置事务特性：

- 使用 GLOBAL 关键字：
  - 该声明适用于所有后续会话。
  - 现有会话不受影响。
- 使用 SESSION 关键字：
  - 该语句适用于当前会话中执行的所有后续事务。
  - 该语句在事务中是允许的，但不影响当前正在进行的事务。
  - 如果在事务之间执行，该语句将覆盖任何先前设置命名特征的下一个事务值的语句。
- 没有任何 SESSION 或 GLOBAL 关键字：
  - 该语句仅适用于会话中执行的下一个单个事务。
  - 后续事务恢复使用命名特征的会话值。
  - 事务中不允许使用以下语句：

```
mysql> START TRANSACTION;
Query OK, 0 rows affected (0.02 sec)
```

```
mysql> SET TRANSACTION ISOLATION LEVEL
SERIALIZABLE;
ERROR 1568 (25001): Transaction characteristics
can't be changed
while a transaction is in progress
```

对全局事务特性的更改需要 CONNECTION_ADMIN 权限（或已弃用的 SUPER 权限）。 任何会话都可以自由更改其会话特性（即使在事务中间），或下一个事务的特性（在该事务开始之前）。

要在服务器启动时设置全局隔离级别，请在命令行或选项文件中使用 --transaction-isolation=level 选项。 此选项的级别值使用破折号而不是空格，因此允许的值为 READ-UNCOMMITTED、READ-COMMITTED、REPEATABLE-READ 或 SERIALIZABLE。

同样，要在服务器启动时设置全局事务访问模式，请使用 --transaction-read-only 选项。 默认值为 OFF（读/写模式），但对于只读模式可以将该值设置为 ON。

例如，要将隔离级别设置为 REPEATABLE READ 并将访问模式设置为 READ WRITE，请在选项文件的

[mysqld] 部分使用这些行：

```
[mysqld]
transaction-isolation = REPEATABLE-READ
transaction-read-only = OFF
```

在运行时，全局、会话和下一个事务范围级别的特征可以使用 SET TRANSACTION 语句间接设置，如前所述。 也可以直接使用 SET 语句设置它们，为 transaction_isolation 和 transaction_read_only 系统变量赋值：

- SET TRANSACTION 允许使用可选的 GLOBAL 和 SESSION 关键字来设置不同范围级别的事务特征。
- 用于为 transaction_isolation 和 transaction_read_only 系统变量赋值的 SET 语句具有在不同范围级别设置这些变量的语法。

下表显示了每个 SET TRANSACTION 和变量赋值语法设置的特征范围级别。

事务特征的 SET TRANSACTION 语法

Syntax	Affected Characteristic Scope
SET GLOBAL TRANSACTION *transaction_characteristic*	Global
SET SESSION TRANSACTION *transaction_characteristic*	Session
SET TRANSACTION *transaction_characteristic*	Next transaction only

事务特征的 SET 语法

Syntax	Affected Characteristic Scope
SET GLOBAL *var_name* = *value*	Global
SET @@GLOBAL.*var_name* = *value*	Global
SET PERSIST *var_name* = *value*	Global
SET @@PERSIST.*var_name* = *value*	Global
SET PERSIST_ONLY *var_name* = *value*	No runtime effect
SET @@PERSIST_ONLY.*var_name* = *value*	No runtime effect
SET SESSION *var_name* = *value*	Session
SET @@SESSION.*var_name* = *value*	Session
SET *var_name* = *value*	Session
SET @@*var_name* = *value*	Next transaction only

可以在运行时检查事务特性的全局和会话值：

```
SELECT @@GLOBAL.transaction_isolation,
@@GLOBAL.transaction_read_only;
SELECT @@SESSION.transaction_isolation,
@@SESSION.transaction_read_only;
```

6.4 Utility 语句

6.4.1 DESCRIBE 语句

DESCRIBE 用于获取表结构信息：

示例：

```
mysql> DESCRIBE City;
```

```
+------------+---------+------+-----+---------+----------------+
| Field      | Type    | Null | Key | Default | Extra          |
+------------+---------+------+-----+---------+----------------+
| Id         | int(11) | NO   | PRI | NULL    | auto_increment |
| Name       | char(35)| NO   |     |         |                |
| Country    | char(3) | NO   | UNI |         |                |
| District   | char(20)| YES  | MUL |         |                |
| Population | int(11) | NO   |     | 0       |                |
+------------+---------+------+-----+---------+----------------+
```

6.4.2 EXPLAIN 语句

语法：

```
{EXPLAIN | DESCRIBE | DESC}
    tbl_name [col_name | wild]

{EXPLAIN | DESCRIBE | DESC}
    [explain_type]
    {explainable_stmt | FOR CONNECTION connection_id}

{EXPLAIN | DESCRIBE | DESC} ANALYZE [FORMAT = TREE] select_statement

explain_type: {
    FORMAT = format_name
}

format_name: {
    TRADITIONAL
  | JSON
  | TREE
}

explainable_stmt: {
    SELECT statement
  | TABLE statement
  | DELETE statement
  | INSERT statement
  | REPLACE statement
  | UPDATE statement
}
```

注意事项：

- 查看执行计划，SQL 不会真正执行
- 在只读的 DB 上，无法查看写 SQL 的执行计划

示例：

```
DROP TABLE if exists employees;
CREATE TABLE employees (
    id INT key NOT NULL,
    fname VARCHAR(30),
    lname VARCHAR(30),
    hired date,
    separated DATE NOT NULL DEFAULT '9999-12-31',
    job_code INT,
    store_id INT
)

MySQL [test]> explain select id,fname,lname from employees where
id=20\G;
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: NULL
   partitions: NULL
         type: NULL
possible_keys: NULL
          key: NULL
      key_len: NULL
          ref: NULL
         rows: NULL
     filtered: NULL
        Extra: no matching row in const table
1 row in set (0.00 sec)
```

执行计划中各字段含义：

- id：执行行顺序，按 1,2,3,4…进行排序。在所有组中，id 值越大，优先级越高，越先执行。id 如果相同，可以认为是一组，从上往下顺序执行
- select_type：select 的类型。
- table：输出记录的表，对应行正在访问哪一个表，表名或者别名，可能是临时表或者 union 合并结果集
- partitions：符合的分区
- type：显示的是访问类型，访问类型表示以何种方式去访问数据，例如全表扫描
- possible_keys：优化器可能使用到的索引
- key：优化器实际选择的索引

- key_len：表示索引中使用的字节数，可以通过 key_len 计算查询中使用的索引长度
- ref：显示索引的哪一列被使用了，如果可能的话，是一个常数
- rows：优化器预估的记录数量，根据表的统计信息及索引使用情况，大致估算出找出所需记录需要读取的行数
- filtered：该 filtered 列指示将按表条件过滤的表行的估计百分比。 最大值为 100，这意味着不会对行进行过滤。 值从 100 开始减少表示过滤量增加
- Extra：额外的显示选项

6.4.3 USE 语句

语法如下：

```
use db_name
```

示例：

```
MySQL [test]> USE db1;
MySQL [test]> SELECT COUNT(*) FROM mytable;
MySQL [test]> USE db2; SELECT COUNT(*) FROM mytable;
```

6.5 预处理语句

TDSQL 支持预处理协议，使用方式与单机 MySQL 相同，例如：

- PREPARE Syntax
- EXECUTE Syntax

二进制协议的支持：

- COM_STMT_PREPARE
- COM_STMT_EXECUTE

注意事项：

- 目前 TDSQL 只对 Prepare/Execute 命令做语法兼容，从性能角度的话，建议用户尽量不要使用该种方式，直接使用文本协议。

示例：

```
MySQL [test]> DROP TABLE IF EXISTS test1;
Query OK, 0 rows affected (0.08 sec)

MySQL [test]> create table test1(a int not null primary key,b in
t) shardkey=a;
Query OK, 0 rows affected (1.71 sec)

MySQL [test]> insert into test1(a,b) values(5,6),(3,4),(1,2);
Query OK, 3 rows affected (0.06 sec)
Records: 3  Duplicates: 0  Warnings: 0

MySQL [test]> select a,b from test1;
+---+------+
| a | b    |
+---+------+
| 1 |    2 |
| 3 |    4 |
| 5 |    6 |
+---+------+
3 rows in set (0.02 sec)

mysql> prepare ff from "select a,b from test1 where a=?";
Query OK, 0 rows affected (0.00 sec)
Statement prepared

mysql> set @aa=3;
Query OK, 0 rows affected (0.00 sec)

mysql> execute ff using @aa;
+---+------+
| a | b    |
+---+------+
| 3 |    4 |
+---+------+
1 row in set (0.06 sec)
```

7 支持的字符集和时区

TDSQL 在后端存储支持 MySQL 的所有字符集和字符序。具体显示如下：

```
mysql> show character set;
+----------+---------------------------------+------------------
--+--------+
```

```
| Charset  | Description                     | Default collation
  | Maxlen |
+----------+---------------------------------+------------------
--+--------+
| big5     | Big5 Traditional Chinese        | big5_chinese_ci
  |      2 |
| dec8     | DEC West European               | dec8_swedish_ci
  |      1 |
| cp850    | DOS West European               | cp850_general_ci
  |      1 |
| hp8      | HP West European                | hp8_english_ci
  |      1 |
| koi8r    | KOI8-R Relcom Russian           | koi8r_general_ci
  |      1 |
| latin1   | cp1252 West European            | latin1_swedish_ci
  |      1 |
| latin2   | ISO 8859-2 Central European     | latin2_general_ci
  |      1 |
| swe7     | 7bit Swedish                    | swe7_swedish_ci
  |      1 |
| ascii    | US ASCII                        | ascii_general_ci
  |      1 |
| ujis     | EUC-JP Japanese                 | ujis_japanese_ci
  |      3 |
| sjis     | Shift-JIS Japanese              | sjis_japanese_ci
  |      2 |
| hebrew   | ISO 8859-8 Hebrew               | hebrew_general_ci
  |      1 |
| tis620   | TIS620 Thai                     | tis620_thai_ci
  |      1 |
| euckr    | EUC-KR Korean                   | euckr_korean_ci
  |      2 |
| koi8u    | KOI8-U Ukrainian                | koi8u_general_ci
  |      1 |
| gb2312   | GB2312 Simplified Chinese       | gb2312_chinese_ci
  |      2 |
| greek    | ISO 8859-7 Greek                | greek_general_ci
  |      1 |
| cp1250   | Windows Central European        | cp1250_general_ci
  |      1 |
| gbk      | GBK Simplified Chinese          | gbk_chinese_ci
  |      2 |
| latin5   | ISO 8859-9 Turkish              | latin5_turkish_ci
  |      1 |
| armscii8 | ARMSCII-8 Armenian              | armscii8_general_c
i |      1 |
| utf8     | UTF-8 Unicode                   | utf8_general_ci
  |      3 |
| ucs2     | UCS-2 Unicode                   | ucs2_general_ci
  |      2 |
```

```
| cp866    | DOS Russian                       | cp866_general_ci
  |      1 |
| keybcs2  | DOS Kamenicky Czech-Slovak        | keybcs2_general_ci
  |      1 |
| macce    | Mac Central European              | macce_general_ci
  |      1 |
| macroman | Mac West European                 | macroman_general_c
i |      1 |
| cp852    | DOS Central European              | cp852_general_ci
  |      1 |
| latin7   | ISO 8859-13 Baltic                | latin7_general_ci
  |      1 |
| utf8mb4  | UTF-8 Unicode                     | utf8mb4_general_ci
  |      4 |
| cp1251   | Windows Cyrillic                  | cp1251_general_ci
  |      1 |
| utf16    | UTF-16 Unicode                    | utf16_general_ci
  |      4 |
| utf16le  | UTF-16LE Unicode                  | utf16le_general_ci
  |      4 |
| cp1256   | Windows Arabic                    | cp1256_general_ci
  |      1 |
| cp1257   | Windows Baltic                    | cp1257_general_ci
  |      1 |
| utf32    | UTF-32 Unicode                    | utf32_general_ci
  |      4 |
| binary   | Binary pseudo charset             | binary
  |      1 |
| geostd8  | GEOSTD8 Georgian                  | geostd8_general_ci
  |      1 |
| cp932    | SJIS for Windows Japanese         | cp932_japanese_ci
  |      2 |
| eucjpms  | UJIS for Windows Japanese         | eucjpms_japanese_c
i |      3 |
| gb18030  | China National Standard GB18030   | gb18030_chinese_ci
  |      4 |
+----------+-----------------------------------+------------------
--+--------+
41 rows in set (0.02 sec)
```

查看当前连接的相关字符集：

```
mysql> show variables like "%char%";
+--------------------------+------------------------------------
----------------+
| Variable_name            | Value
                |
+--------------------------+------------------------------------
----------------+
| character_set_client     | latin1
```

```
                |
| character_set_connection | latin1
                |
| character_set_database   | utf8
                |
| character_set_filesystem | binary
                |
| character_set_results    | latin1
                |
| character_set_server     | utf8
                |
| character_set_system     | utf8
                |
| character_sets_dir       | /data/tdsql_run/8812/percona-5.7.17/
share/charsets/ |
+--------------------------+-------------------------------------
----------------+
```

设置当前连接的相关字符集：

```
mysql> set names utf8;
Query OK, 0 rows affected (0.03 sec)

mysql> show variables like "%char%";
+--------------------------+-------------------------------------
----------------+
| Variable_name            | Value
                |
+--------------------------+-------------------------------------
----------------+
| character_set_client     | utf8
                |
| character_set_connection | utf8
                |
| character_set_database   | utf8
                |
| character_set_filesystem | binary
                |
| character_set_results    | utf8
                |
| character_set_server     | utf8
                |
| character_set_system     | utf8
                |
| character_sets_dir       | /data/tdsql_run/8811/percona-5.7.17/
share/charsets/ |
+--------------------------+-------------------------------------
----------------+
```

注意事项：TDSQL 不支持通过命令行设置参数，需要通过赤兔管理台进行设置。

通过设置 time_zone 变量修改时区相关的属性：

```
mysql> show variables like '%time_zone%';
+------------------+--------+
| Variable_name    | Value  |
+------------------+--------+
| system_time_zone | CST    |
| time_zone        | SYSTEM |
+------------------+--------+
2 rows in set (0.00 sec)

mysql> create table test.tt (ts timestamp, dt datetime,c int key)
 shardkey=c;
Query OK, 0 rows affected (0.49 sec)

mysql>  insert into test.tt (ts,dt,c)values ('2017-10-01 12:12:12
', '2017-10-01 12:12:12',1);
Query OK, 1 row affected (0.09 sec)

mysql> select * from test.tt;
+---------------------+---------------------+------+
| ts                  | dt                  | c    |
+---------------------+---------------------+------+
| 2017-10-01 12:12:12 | 2017-10-01 12:12:12 |    1 |
+---------------------+---------------------+------+
1 row in set (0.04 sec)

mysql> set time_zone = '+12:00';
Query OK, 0 rows affected (0.00 sec)

mysql> show variables like '%time_zone%';
+------------------+--------+
| Variable_name    | Value  |
+------------------+--------+
| system_time_zone | CST    |
| time_zone        | +12:00 |
+------------------+--------+
2 rows in set (0.00 sec)

mysql> select * from test.tt;
+---------------------+---------------------+------+
| ts                  | dt                  | c    |
+---------------------+---------------------+------+
| 2017-10-01 16:12:12 | 2017-10-01 12:12:12 |    1 |
+---------------------+---------------------+------+
1 row in set (0.06 sec)
```

8 支持的函数

集中式实例支持以下 7 种类型的函数：

- 流程控制函数（Control Flow Functions）
- 字符串函数（String Functions）
- 数字函数（Numeric Functions and Operators）
- 日期时间函数（Date and Time Functions）
- 聚合函数（Aggregate (GROUP BY) Functions）
- 位函数（Bit Functions and Operators）
- 转换函数（Cast Functions and Operators）

以上类型的各函数具体描述如下：

流程控制函数（Control Flow Functions）

函数名	描述
CASE	Case operator
IF()	If/else construct
IFNULL()	Null if/else construct
NULLIF()	Return NULL if expr1 = expr2

字符串函数（String Functions）

函数名	描述
ASCII()	Return numeric value of left-most character
BIN()	Return a string containing binary

函数名	描述
	representation of a number
BIT_LENGTH()	Return length of argument in bits
CHAR()	Return the character for each integer passed
CHAR_LENGTH()	Return number of characters in argument
CHARACTER_LENGTH()	Synonym for CHAR_LENGTH()
CONCAT()	Return concatenated string
CONCAT_WS()	Return concatenate with separator
ELT()	Return string at index number
EXPORT_SET()	Return a string such that for every bit set in the value bits, you get an on string and for every unset bit, you get an off string
FIELD()	Return the index (position) of the first argument in the subsequent arguments
FIND_IN_SET()	Return the index position of the first argument within the second argument
FORMAT()	Return a number formatted to specified number of decimal places
FROM_BASE64()	Decode to a base-64 string and return result

函数名	描述
HEX()	Return a hexadecimal representation of a decimal or string value
INSERT()	Insert a substring at the specified position up to the specified number of characters
INSTR()	Return the index of the first occurrence of substring
LCASE()	Synonym for LOWER()
LEFT()	Return the leftmost number of characters as specified
LENGTH()	Return the length of a string in bytes
LIKE	Simple pattern matching
LOAD_FILE()	Load the named file
LOCATE()	Return the position of the first occurrence of substring
LOWER()	Return the argument in lowercase
LPAD()	Return the string argument, left-padded with the specified string
LTRIM()	Remove leading spaces

函数名	描述
MAKE_SET()	Return a set of comma-separated strings that have the corresponding bit in bits set
MATCH	Perform full-text search
MID()	Return a substring starting from the specified position
NOT LIKE	Negation of simple pattern matching
NOT REGEXP	Negation of REGEXP
OCT()	Return a string containing octal representation of a number
OCTET_LENGTH()	Synonym for LENGTH()
ORD()	Return character code for leftmost character of the argument
POSITION()	Synonym for LOCATE()
QUOTE()	Escape the argument for use in an SQL statement
REGEXP	Pattern matching using regular expressions
REPEAT()	Repeat a string the specified number of times
REPLACE()	Replace occurrences of a specified string

函数名	描述
REVERSE()	Reverse the characters in a string
RIGHT()	Return the specified rightmost number of characters
RLIKE	Synonym for REGEXP
RPAD()	Append string the specified number of times
RTRIM()	Remove trailing spaces
SOUNDEX()	Return a soundex string
SOUNDS LIKE	Compare sounds
SPACE()	Return a string of the specified number of spaces
STRCMP()	Compare two strings
SUBSTR()	Return the substring as specified
SUBSTRING()	Return the substring as specified
SUBSTRING_INDEX()	Return a substring from a string before the specified number of occurrences of the delimiter
TO_BASE64()	Return the argument converted to a base-64 string

函数名	描述
TRIM()	Remove leading and trailing spaces
UCASE()	Synonym for UPPER()
UNHEX()	Return a string containing hex representation of a number
UPPER()	Convert to uppercase
WEIGHT_STRING()	Return the weight string for a string

数字函数（**Numeric Functions and Operators**）

函数名	描述
ABS()	Return the absolute value
ACOS()	Return the arc cosine
ASIN()	Return the arc sine
ATAN()	Return the arc tangent
ATAN2(), ATAN()	Return the arc tangent of the two arguments
CEIL()	Return the smallest integer value not less than the argument
CEILING()	Return the smallest integer value not less than the

函数名	描述
	argument
CONV()	Convert numbers between different number bases
COS()	Return the cosine
COT()	Return the cotangent
CRC32()	Compute a cyclic redundancy check value
DEGREES()	Convert radians to degrees
DIV	Integer division
/	Division operator
EXP()	Raise to the power of
FLOOR()	Return the largest integer value not greater than the argument
LN()	Return the natural logarithm of the argument
LOG()	Return the natural logarithm of the first argument
LOG10()	Return the base-10 logarithm of the argument
LOG2()	Return the base-2 logarithm of the argument

函数名	描述
-	Minus operator
MOD()	Return the remainder
%, MOD	Modulo operator
PI()	Return the value of pi
+	Addition operator
POW()	Return the argument raised to the specified power
POWER()	Return the argument raised to the specified power
RADIANS()	Return argument converted to radians
RAND()	Return a random floating-point value
ROUND()	Round the argument
SIGN()	Return the sign of the argument
SIN()	Return the sine of the argument
SQRT()	Return the square root of the argument
TAN()	Return the tangent of the argument
*	Multiplication operator

函数名	描述
TRUNCATE()	Truncate to specified number of decimal places
-	Change the sign of the argument

日期时间函数（Date and Time Functions）

函数名	描述
ADDDATE()	Add time values (intervals) to a date value
ADDTIME()	Add time
CONVERT_TZ()	Convert from one time zone to another
CURDATE()	Return the current date
CURRENT_DATE(), CURRENT_DATE	Synonyms for CURDATE()
CURRENT_TIME(), CURRENT_TIME	Synonyms for CURTIME()
CURRENT_TIMESTAMP(), CURRENT_TIMESTAMP	Synonyms for NOW()
CURTIME()	Return the current time
DATE()	Extract the date part of a date or datetime expression

函数名	描述
DATE_ADD()	Add time values (intervals) to a date value
DATE_FORMAT()	Format date as specified
DATE_SUB()	Subtract a time value (interval) from a date
DATEDIFF()	Subtract two dates
DAY()	Synonym for DAYOFMONTH()
DAYNAME()	Return the name of the weekday
DAYOFMONTH()	Return the day of the month (0-31)
DAYOFWEEK()	Return the weekday index of the argument
DAYOFYEAR()	Return the day of the year (1-366)
EXTRACT()	Extract part of a date
FROM_DAYS()	Convert a day number to a date
FROM_UNIXTIME()	Format Unix timestamp as a date
GET_FORMAT()	Return a date format string

函数名	描述
HOUR()	Extract the hour
LAST_DAY	Return the last day of the month for the argument
LOCALTIME(), LOCALTIME	Synonym for NOW()
LOCALTIMESTAMP, LOCALTIMESTAMP()	Synonym for NOW()
MAKEDATE()	Create a date from the year and day of year
MAKETIME()	Create time from hour, minute, second
MICROSECOND()	Return the microseconds from argument
MINUTE()	Return the minute from the argument
MONTH()	Return the month from the date passed
MONTHNAME()	Return the name of the month
NOW()	Return the current date and time
PERIOD_ADD()	Add a period to a year-month
PERIOD_DIFF()	Return the number of months between periods

函数名	描述
QUARTER()	Return the quarter from a date argument
SEC_TO_TIME()	Converts seconds to 'HH SS' format
SECOND()	Return the second (0-59)
STR_TO_DATE()	Convert a string to a date
SUBDATE()	Synonym for DATE_SUB() when invoked with three arguments
SUBTIME()	Subtract times
SYSDATE()	Return the time at which the function executes
TIME()	Extract the time portion of the expression passed
TIME_FORMAT()	Format as time
TIME_TO_SEC()	Return the argument converted to seconds
TIMEDIFF()	Subtract time
TIMESTAMP()	With a single argument, this function returns the date or datetime expression; with two arguments, the sum of the

函数名	描述
	arguments
TIMESTAMPADD()	Add an interval to a datetime expression
TIMESTAMPDIFF()	Subtract an interval from a datetime expression
TO_DAYS()	Return the date argument converted to days
TO_SECONDS()	Return the date or datetime argument converted to seconds since Year 0
UNIX_TIMESTAMP()	Return a Unix timestamp
UTC_DATE()	Return the current UTC date
UTC_TIME()	Return the current UTC time
UTC_TIMESTAMP()	Return the current UTC date and time
WEEK()	Return the week number
WEEKDAY()	Return the weekday index
WEEKOFYEAR()	Return the calendar week of the date (1-53)
YEAR()	Return the year

函数名	描述
YEARWEEK()	Return the year and week

聚合函数（Aggregate (GROUP BY) Functions）

函数名	描述
AVG()	Return the average value of the argument
COUNT()	Return a count of the number of rows returned
MAX()	Return the maximum value
MIN()	Return the minimum value
SUM()	Return the sum

位函数（Bit Functions and Operators）

函数名	描述
BIT_COUNT()	Return the number of bits that are set
&	Bitwise AND
~	Bitwise inversion
|	Bitwise OR
^	Bitwise XOR

函数名	描述
<<	Left shift
>>	Right shift

转换函数（Cast Functions and Operators）

函数名	描述
BINARY	Cast a string to a binary string
CAST()	Cast a value as a certain type
CONVERT()	Cast a value as a certain type