腾讯云数据仓库 TCHouse-X

语法手册

腾讯云

文档目录

语法手册
数据类型
数据类型概览

最近更新时间：2026-05-06 16:28:12

TCHouse-X 支持一组核心数据类型，这些类型用于定义表列、表达式值，以及函数的参数和返回值。

隐式类型转换原则

TCHouse-X 仅支持有限的隐式类型转换。这种限制是为了防止因意外的转换行为而产生不可预期的结果。

禁止隐式转换（需要 CAST()）

字符串与数值/布尔类型： TCHouse-X 不会在 `STRING` 类型与任何数值类型（如 `INT`，`BIGINT` 等）或 `BOOLEAN` 类型之间执行隐式转换。

> **⚠ 注意：**
> 遇到此类转换需求时，请始终显式使用 `CAST()` 函数。

允许隐式转换（自动提升）

数值类型之间的提升： TCHouse-X 会在数值类型之间，从“较小或精度较低”的类型向“较大或精度更高”的类型执行隐式转换。

- 示例： `SMALLINT` 可以隐式转换为 `BIGINT` 或 `FLOAT`。
- **降级限制：** 相反地，从“较大或精度较高”的类型向“较小或精度较低”的类型转换，如将 `DOUBLE` 转换为 `FLOAT`，或将 `INT` 转换为 `TINYINT`，必须在查询中显式调用 `CAST()`。

TCHouse-X 内置数据类型一览表

类型	范围/约束	描述及说明
BOOLEAN	`TRUE` 或 `FALSE`	TCHouse-X 兼容 MySQL 协议：布尔值 `TRUE` 和 `FALSE` 在查询结果中分别以数值 `1` 和 `0` 形式展示。
TINYINT	[`-128`, `127`]	TCHouse-X 内表暂不支持 `TINYINT` 类型，建表时该类型将自动转换为 `INT` 类型。
SMALLINT	[`-32768`, `32767`]	TCHouse-X 内表暂不支持 `SMALLINT` 类型，建表时该类型将自动转换为 `INT` 类型。

INT	[-2147483648 , 2147483647]	
BIGINT	[-9223372036854775808 , 9223372036854775807]	
FLOAT	• **正数范围：** [1.401298464632481707e-45 , 3.4028234663852886e+38] • **负数范围：** [-3.4028234663852886e+38 , -1.401298464632481707e-45]	单精度浮点数，6 到 9 位有效数字。
DOUBLE	• **正数范围：** [4.94065645841246544e-324 , 1.7976931348623157e+308] • **负数范围：** [-1.7976931348623157e+308 , -4.94065645841246544e-324]	双精度浮点数，15 到 17 位有效数字。
DECIMAL	[-10^38 + 1 , 10^38 - 1]	高精度定点数。 – 精度 (M): 有效数字总数 [1, 27] – 小数位 (D): 小数位数字数量 [0, 9] – 约束：M >= D，整数位最多 18 位
CHAR	长度范围：1-255	定长字符串，TCHouse-X 内表暂不支持 CHAR 类型，建表时该类型将自动转换为 STRING 类型。
VARCHAR	长度范围：1-65535	定长字符串，TCHouse-X 内表暂不支持 VARCHAR 类型，建表时该类型将自动转换为 STRING 类型。
STRING	最大支持 2 GB	变长字符串，使用 UTF-8 编码存储。
DATE	['0001-01-01' , '9999-12-31']	仅包含日期信息。

TIMESTAMP/TIMESTAMPTZ	[0001-01-01 , 9999-12-31]	包含日期和时间（年、月、日、时、分、秒），包含本地时区信息。精度为微秒。
TIMESTAMPNTZ	[0001-01-01 , 9999-12-31]	包含日期和时间（年、月、日、时、分、秒），不包含本地时区信息。精度为微秒。

TINYINT

最近更新时间：2026-05-06 16:28:12

`TINYINT` 是一种 1 字节整数数据类型，用于 `CREATE TABLE` 和 `ALTER TABLE` 语句中。

语法

在 `CREATE TABLE` 语句的列定义中：

```
column_name  TINYINT
```

> **注意：**
> 由于 TCHouse-X 内表暂不支持 `TINYINT`，系统在建表阶段会自动完成类型提升（Type Promotion），将 `TINYINT` 字段统一处理为 `INT` 类型。

范围

- **取值范围：** `-128` 到 `127`。
- **注意：** TCHouse-X 不支持无符号（`UNSIGNED`）子类型。

数据转换

隐式转换

TCHouse-X 会自动将 `TINYINT` 转换为更大范围的整数类型（`SMALLINT`，`INT`，`BIGINT`）或浮点类型（`FLOAT`，`DOUBLE`）。

显式转换

- 转换为 `STRING` 或 `TIMESTAMP` 必须使用 `CAST()`。
- **TIMESTAMP/TIMESTAMPTZ 转换规则：** 将整数值 N 转换为 `TIMESTAMP` / `TIMESTAMPTZ` 时，系统会生成一个以 Unix 纪元（1970-01-01 00:00:00 UTC）为基准、增加 N 秒后的时间戳。在查询展示时，该值会自动转换为当前系统时区对应的本地时间。
- **TIMESTAMPNTZ 转换规则：** 将整数值 N 转换为 `TIMESTAMPNTZ` 时，系统会生成一个以 Unix 纪元（1970-01-01 00:00:00 UTC）为基准、增加 N 秒后的时间戳。

溢出处理

系统将溢出结果当作 `NULL` 返回，例如，`CAST(-200 AS TINYINT)` 返回 `NULL`，`CAST(200 AS TINYINT)` 返回 `NULL`）。

使用说明与限制

- **选型建议：**如果数值可能超出 127，请改用 `SMALLINT` 类型。
- **NULL 处理：**将任何非数字值（Non-numeric）强制转换为此类型将产生 `NULL` 。

SMALLINT

最近更新时间：2026-05-06 16:28:12

`SMALLINT` 是一种 2 字节整数数据类型，用于 `CREATE TABLE` 和 `ALTER TABLE` 语句。

语法

在 `CREATE TABLE` 语句的列定义中：

```
column_name SMALLINT
```

> **注意：**
> 由于 TCHouse-X 内表暂不支持 `SMALLINT`，系统在建表阶段会自动完成类型提升（Type Promotion），将 `SMALLINT` 字段统一处理为 `INT` 类型。

取值范围

- 取值范围：`-32,768` 到 `32,767`。
- 注意：TCHouse-X 不支持无符号（UNSIGNED）子类型。

数据转换

隐式转换

TCHouse-X 会自动将 `SMALLINT` 转换为更大范围的整数类型（`INT`，`BIGINT`）或浮点类型（`FLOAT`，`DOUBLE`）。

显式转换

- 转换为 `STRING` 或 `TIMESTAMP` 必须使用 `CAST()`。
- **TIMESTAMP/TIMESTAMPTZ 转换规则：**将整数值 N 转换为 `TIMESTAMP` / `TIMESTAMPTZ` 时，系统会生成一个以 Unix 纪元（1970-01-01 00:00:00 UTC）为基准、增加 N 秒后的时间戳。在查询展示时，该值会自动转换为当前系统时区对应的本地时间。
- **TIMESTAMPNTZ 转换规则：**将整数值 N 转换为 `TIMESTAMPNTZ` 时，系统会生成一个以 Unix 纪元（1970-01-01 00:00:00 UTC）为基准、增加 N 秒后的时间戳。

使用说明与限制

- **选型建议：**如果数值可能超出 32,767，请改用 `INT` 类型。
- **NULL 处理：**将任何非数字值（Non-numeric）强制转换为此类型将产生 `NULL`。

- **分区 (Partitioning)：优先使用此类型作为分区键列。相比字符串形式，TCHouse-X 处理数值类型的效率更高。**

INT

最近更新时间：2026-05-06 16:28:12

`INT` 是一种 4 字节整数数据类型，用于 `CREATE TABLE` 和 `ALTER TABLE` 语句。

语法

在 `CREATE TABLE` 语句的列定义中：

```
column_name INT
```

范围

- **取值范围：** `-2,147,483,648` 到 `2,147,483,647`。
- **注意：** TCHouse-X 不支持无符号（`UNSIGNED`）子类型。

数据转换

隐式转换

TCHouse-X 会自动将 `INT` 转换为更大范围的整数类型（`BIGINT`）或浮点类型（`FLOAT`，`DOUBLE`）。

显式转换

- 转换为 `TINYINT`、`SMALLINT`、`STRING` 或 `TIMESTAMP` 必须使用 `CAST()`。
- **TIMESTAMP/TIMESTAMPTZ 转换规则：** 将整数值 N 转换为 `TIMESTAMP` / `TIMESTAMPTZ` 时，系统会生成一个以 Unix 纪元（1970-01-01 00:00:00 UTC）为基准、增加 N 秒后的时间戳。在查询展示时，该值会自动转换为当前系统时区对应的本地时间。
- **TIMESTAMPNTZ 转换规则：** 将整数值 N 转换为 `TIMESTAMPNTZ` 时，系统会生成一个以 Unix 纪元（1970-01-01 00:00:00 UTC）为基准、增加 N 秒后的时间戳。

使用说明与限制

- **选型建议：** 如果数值可能超出 `2,147,483,647`，请改用 `BIGINT` 类型。
- **NULL 处理：** 将任何非数字值（Non-numeric）强制转换为此类型将产生 `NULL`。
- **分区 (Partitioning)：** 优先使用此类型作为分区键列。相比字符串形式，TCHouse-X 处理数值类型的效率更高。

BIGINT

最近更新时间：2026-05-06 16:28:12

`BIGINT` 是一种 8 字节整数数据类型，用于 `CREATE TABLE` 和 `ALTER TABLE` 语句。

语法

在 `CREATE TABLE` 语句的列定义中：

```
column_name BIGINT
```

范围

- **取值范围：** `-9,223,372,036,854,775,808` 到 `9,223,372,036,854,775,807` 。
- **注意：** TCHouse-X 不支持无符号（ `UNSIGNED` ）子类型。

数据转换

隐式转换

TCHouse-X 会自动将 `BIGINT` 转换为浮点类型（ `FLOAT` 或 `DOUBLE` ）。

显式转换

- 转换为 `TINYINT` 、 `SMALLINT` 、 `INT` 、 `STRING` 或 `TIMESTAMP` 必须使用 `CAST()` 。
- **TIMESTAMP/TIMESTAMPTZ 转换规则：** 将整数值 N 转换为 `TIMESTAMP` / `TIMESTAMPTZ` 时，系统会生成一个以 Unix 纪元（1970-01-01 00:00:00 UTC）为基准、增加 N 秒后的时间戳。在查询展示时，该值会自动转换为当前系统时区对应的本地时间。
- **TIMESTAMPNTZ 转换规则：** 将整数值 N 转换为 `TIMESTAMPNTZ` 时，系统会生成一个以 Unix 纪元（1970-01-01 00:00:00 UTC）为基准、增加 N 秒后的时间戳。

使用说明与建议

- **选型权衡：** BIGINT 在声明列时非常方便，因为它可以容纳任何整数值，且在 `INSERT` 时会自动进行类型提升。然而，它是占用空间最大的整数类型（磁盘和内存均占 8 字节）。过度使用会导致查询效率下降，扩展性变差。
- **最佳实践：** 优先使用能容纳输入值的最小整数类型。仅在必要时使用 `CAST()` 转换为大类型。
- **溢出处理：** 如果数值超过了 `BIGINT` 的范围，请使用具有足够精度的 `DECIMAL` 类型。
- **NULL 处理：** 将任何非数字值强制转换为此类型将产生 NULL。
- **分区 (Partitioning)：** 优先使用此类型作为分区键列。相比字符串形式，Impala 处理数值类型的效率更高。

示例

```
-- 创建包含 BIGINT 的表
CREATE TABLE t1 (x BIGINT);

-- 显式转换示例
SELECT CAST(1000 AS BIGINT);
```

FLOAT

最近更新时间：2026-05-06 16:28:12

`FLOAT` 是一种单精度浮点数据类型，用于 `CREATE TABLE` 和 `ALTER TABLE` 语句。

语法

在 `CREATE TABLE` 的列定义中：

```
column_name  FLOAT
```

数值特性

- **取值范围：**
  - **正数范围：**[`1.40129846432481707e-45` ， `3.40282346638528860e+38`]
  - **负数范围：**[`-3.40282346638528860e+38` ，`-1.40129846432481707e-45`]
- **精度：**6 到 9 位有效数字，具体取决于使用情况。有效数字的位数与小数点的位置无关。
- **存储表示：**占用 4 字节 存储空间，遵循 IEEE 754 单精度二进制浮点数 格式。

数据转换

隐式转换

TCHouse-X 会自动将 `FLOAT` 转换为更高精度的 `DOUBLE` 值，但不支持反向隐式转换。

显式转换

- 可使用 `CAST()` 将 `FLOAT` 转换为 `TINYINT` 、 `SMALLINT` 、 `INT` 、 `BIGINT` 、 `STRING` 。

```
SELECT
    CAST(123.45 AS TINYINT) AS to_tinyint,
    CAST(1234.56 AS SMALLINT) AS to_smallint,
    CAST(123456.7 AS INT) AS to_int,
    CAST(9876543210.1 AS BIGINT) AS to_bigint,
    CAST(1.0 AS BOOLEAN)  AS true_case,
    CAST(123.45 AS STRING) AS to_string;
-- 结果：123,234,123456,9876543210,1,'123.45'
```

- 科学计数法：在 `FLOAT` 字面量或从 `STRING` 转换时，可以使用指数表示法（例如 1.0e6 表示一百万）。

- **TIMESTAMP/TIMESTAMPTZ 转换规则：**将整数值 N 转换为 `TIMESTAMP` / `TIMESTAMPTZ` 时，系统会生成一个以 Unix 纪元（1970-01-01 00:00:00 UTC）为基准、增加 N 秒后的时间戳。在查询展示时，该值会自动转换为当前系统时区对应的本地时间。
- **TIMESTAMPNTZ 转换规则：**将整数值 N 转换为 `TIMESTAMPNTZ` 时，系统会生成一个以 Unix 纪元（1970-01-01 00:00:00 UTC）为基准、增加 N 秒后的时间戳。

使用说明与限制

聚合计算的差异性

由于 `FLOAT` 和 `DOUBLE` 采用高性能硬件指令进行算术运算，且分布式查询在处理海量数据时，各节点执行运算的顺序可能不同，导致对大批量数据进行 `SUM()` 或 `AVG()` 等聚合操作时，结果可能会有微小差异。若要求结果严格一致且可重复，请使用 `DECIMAL` 类型。

精度缺失风险

浮点数无法精确表示某些小数值。在对精度要求极高的场景（尤其是从其他使用不同存储格式的数据库系统迁移数据时），`DECIMAL` 通常是比 `FLOAT` 或 `DOUBLE` 更好的选择。

DOUBLE

最近更新时间：2026-05-06 16:28:12

`DOUBLE` 是一种双精度浮点数据类型，用于 `CREATE TABLE` 和 `ALTER TABLE` 语句。

语法

在 `CREATE TABLE` 语句的列定义中：

```
column_name DOUBLE
```

数值特性

- **取值范围：**
  - **正数范围：** [`4.94065645841246544e-324` , `1.79769313486231570e+308`]
  - **负数范围：** [`-1.79769313486231570e+308` , `-4.94065645841246544e-324`]
- **精度：** 15 到 17 位有效数字，具体取决于使用情况。有效数字的位数与小数点的位置无关。
- **存储表示：** 占用 8 字节 存储空间，遵循 IEEE 754 双精度二进制浮点数 格式。

数据转换

隐式转换

TCHouse-X 不会自动将 DOUBLE 转换为任何其他类型。

显式转换

- 可使用 `CAST()` 将 `DOUBLE` 转换为 `FLOAT` 、 `TINYINT` 、 `SMALLINT` 、 `INT` 、 `BIGINT` 、 `STRING` 、 `TIMESTAMP` 或 `BOOLEAN` 。
- **科学计数法：** 在 `DOUBLE` 字面量或从 `STRING` 转换时，可以使用指数表示法（例如 `1.0e6` 表示一百万）。
- **TIMESTAMP/TIMESTAMPTZ 转换规则：** 将整数值 N 转换为 `TIMESTAMP` / `TIMESTAMPTZ` 时，系统会生成一个以 Unix 纪元（1970-01-01 00:00:00 UTC）为基准、增加 N 秒后的时间戳。在查询展示时，该值会自动转换为当前系统时区对应的本地时间。
- **TIMESTAMPNTZ 转换规则：** 将整数值 N 转换为 `TIMESTAMPNTZ` 时，系统会生成一个以 Unix 纪元（1970-01-01 00:00:00 UTC）为基准、增加 N 秒后的时间戳。

使用说明与限制

聚合计算的差异性

由于 `FLOAT` 和 `DOUBLE` 采用高性能硬件指令进行算术运算，且分布式查询在处理海量数据时，各节点执行运算的顺序可能不同，导致对大批量数据进行 `SUM()` 或 `AVG()` 等聚合操作时，结果可能会有微小差异。若要求结果严格一致且可重复，请使用 `DECIMAL` 类型。

精度缺失风险

浮点数无法精确表示某些小数值。在对精度要求极高的场景（尤其是从其他使用不同存储格式的数据库系统迁移数据时），`DECIMAL` 通常是比 `FLOAT` 或 `DOUBLE` 更好的选择。

DECIMAL

最近更新时间：2026-05-06 16:28:12

DECIMAL 是一种具有**固定精度（Precision）和标度（Scale）**的数值数据类型。它非常适合存储和计算需要精确结果的数值，例如货币、科学计量等。

语法与定义

```
DECIMAL[(precision[, scale])]
```

- **Precision（精度）：**表示该数值的总位数（不含小数点）。
  - 范围：1 到 38。
  - 默认值：9。
- **Scale（标度）：**表示小数点后的位数。
  - 必须小于或等于精度。
  - 默认值：0。
- **默认状态：**省略参数时，默认为 `DECIMAL(9, 0)`。

范围与极值

- **取值范围：** `-10^38+1` 到 `10^38-1`。
- **最大值：**由 `DECIMAL(38, 0)` 表示。
- **最精密小数值：**由 `DECIMAL(38, 38)` 表示（小数点后 38 位）。其最接近 0 的值为 `0.000...1`（37 个零），最接近 1 的值为 `0.999...`（38 个九）。

内存占用

DECIMAL 的内存占用仅由**精度（Precision）**决定，与标度无关。

精度 (Precision)	内存占用 (In-memory Storage)
1 – 9	4 字节
10 – 18	8 字节
19 – 38	16 字节

算术运算规则

TCHouse-X 的 `DECIMAL` 运算遵循严格的精度推导公式，最大精度为 38。若推导出的精度超过 38，TCHouse-X 将优先保留至少 6 位小数并进行四舍五入。

运算	结果精度 (Resulting Precision)	结果标度 (Resulting Scale)
加法 / 减法	`max(L1, L2) + max(S1, S2) + 1`	`max(S1, S2)`
乘法	`P1 + P2 + 1`	`S1 + S2`
除法	`L1 + S2 + max(S1 + P2 + 1, 6)`	`max(S1 + P2 + 1, 6)`
取模	`min(L1, L2) + max(S1, S2)`	`max(S1, S2)`

说明：

变量定义：

- p1, p2：输入值的精度（Precision）。
- S1, S2：输入值的标度（Scale）。
- L1, L2：输入值的整数部分位数，计算公式为 L = P − S。

类型转换

隐式转换

- **DECIMAL → DOUBLE/FLOAT：**必要时会自动转换，但会丢失精度（故不推荐大量使用浮点型）。
- **DOUBLE/FLOAT → DECIMAL：**禁止隐式转换，必须显式使用 `CAST()`。
- **整数 → DECIMAL：**仅当目标 `DECIMAL` 有足够空间容纳该整数类型的最大位数时才允许。
  - BI`G` INT 需要左侧保留 19 位；`INT` 10 位；`SMALLINT` 5 位；`TINYINT` 3 位。

显式转换

- **CAST(STRING AS DECIMAL):**
  - 如果字符串小数位多于目标标度，将执行四舍五入。
  - 支持科学计数法，如 `CAST('1.0e6' AS DECIMAL(32, 0))`。
- `CAST(DECIMAL AS TIMESTAMP)`：返回 Unix 纪元（1970-01-01）之后 N 秒的时间点。在查询展示时，该值会自动转换为当前系统时区对应的本地时间。

与 FLOAT/DOUBLE 的对比

特性	DECIMAL	FLOAT / DOUBLE
精确性	绝对精确（适合金融计算）	不精确，存在舍入误差
性能	运算开销略高	高性能（硬件指令加速）

聚合一致性	结果可重复且稳定	大数据量 SUM/AVG 可能产生微小波动
分区	推荐，目录名与数值严格匹配	不推荐，目录名可能不符合预期

特殊场景注意事项

分区

使用 `DECIMAL` 列作为分区键比浮点数更可靠，因为其目录名称与数值完全对应，不会出现浮点数表示不精确导致的目录名误差。

CHAR

最近更新时间：2026-05-06 16:28:12

> **注意：**
> TCHouse-X 内表和 Iceberg 外表并不强制区分 `CHAR` 与 `STRING`，两者处理逻辑相同。若需利用 `CHAR` 的特定标准特性，请确保在 Hive 外表场景下使用，本章内容亦仅针对此类外表场景。

固定长度字符类型，如果值长度不足，则在尾部用空格填充到指定长度；如果值长度超过指定长度，则截断多余的尾部字符。

语法

在 `CREATE TABLE` 语句的列定义中：

```
column_name  CHAR ( length )
```

- 最大长度：可指定的 length 上限为 255。

尾部空格语义

- 当存储的 `CHAR` 值短于指定长度时，查询返回该值时会在必要时以尾部空格填充，最终返回的字符串长度与定义长度相同。
- 在数据文件中， `CHAR` 值的前导空格会被保留；
- 如果值本身包含尾部空格，这些空格不会存储在数据文件中，但查询时会补足到定义长度。
- 比较两个仅尾部空格数量不同的 `CHAR` 值时，会被视为相等。
- 比较或处理 CHAR 值时，如果转换后超过定义长度，会截断；如果不足，则填充
  - `SELECT CAST('x' AS CHAR(4)) = CAST('x ' AS CHAR(4)); --` 返回 `TRUE`。
  - `CHAR_LENGTH()` 返回包括尾部空格在内的长度。
  - `LENGTH()` 返回不包括尾部空格的长度。
  - `CONCAT()` 返回包括尾部空格在内的字符串。

分区

`CHAR` 类型可用于分区键列。但由于数值类型处理效率更高，如果分区键表示数字，建议使用合适范围的整数类型（如 `INT` 、 `BIGINT` ）。

内部细节

在内存中， `CHAR` 值表示为与定义长度相同大小的字节数组，值不足时右侧补空格。

列统计信息

`CHAR` 类型长度固定，其列统计信息中的最大和平均大小字段在运行 `COMPUTE STATS` 之前即已填充。

限制

- `CHAR` 和 `VARCHAR` 列中的所有数据必须采用与 UTF-8 兼容的编码。如需存储二进制 BLOB，请使用 `STRING` 列。
- 表达式比较 `CHAR` 与 `STRING` 或 `VARCHAR` 时，`CHAR` 会先隐式转换为 `STRING`，并保留尾部空格。该行为与其他数据库系统不同。如需得到 TRUE，需将两侧都 `CAST` 为相同长度的 `CHAR`：

```
-- 如下 case 返回 1，即 true
SELECT CAST("foo  " AS CHAR(5)) = CAST('foo' AS CHAR(3));
```

VARCHAR

最近更新时间：2026-05-06 16:28:12

> **注意：**
> TCHouse-X 内表和 Iceberg 外表并不强制区分 `VARCHAR` 与 `STRING`，两者处理逻辑相同。若需利用 `VARCHAR` 的特定标准特性，请确保在 Hive 外表场景下使用，本章内容亦仅针对此类外表场景。

`VARCHAR` 是一种变长字符类型。如果存储的值超过指定长度，在处理时会根据需要进行截断。

语法

在 `CREATE TABLE` 语句的列定义中：

```
column_name VARCHAR(max_length)
```

- 最大长度：可指定的 max_length 上限为 65,535。
- 内存占用：在内存中表示为字节数组，仅占用表示实际值所需的最小空间（不含填充）。

模式演变

你可以使用 `ALTER TABLE ... CHANGE` 语句在以下类型间自由转换：

- `STRING` ↔ `VARCHAR(n)`
- `CHAR(n)` ↔ `VARCHAR(n)`
- 转换时可以更改长度值。即使原表中存在超长值，TCHouse-X 也不会报错，而是选择在查询时截断。

开发建议与限制

分区

`VARCHAR` 可用作分区键列。如果分区键是数字的字符串表示，出于性能和扩展性考虑，建议优先使用范围足够的整数类型（如 `INT`，`BIGINT`），因为数值计算比字符匹配更高效。

编码限制

`CHAR` 和 `VARCHAR` 列中的所有数据必须使用与 UTF-8 兼容的字符编码。如果你需要存储来自其他数据库系统的二进制数据（即 `BLOB` 类型），请改用 `STRING` 列。

示例

截断行为演示

```
CREATE TABLE varchar_1 (s VARCHAR(1));
CREATE TABLE varchar_4 (s VARCHAR(4));
CREATE TABLE varchar_20 (s VARCHAR(20));

-- 插入数据（超长部分会被截断）
INSERT INTO varchar_1 VALUES (CAST('hello' AS VARCHAR(1))); -- 存储为 'h'
INSERT INTO varchar_4 VALUES (CAST('hello' AS VARCHAR(4))); -- 存储为
'hell'

-- 查询对比
SELECT * FROM varchar_1; -- 输出 'h'
SELECT CONCAT('[', s, ']') FROM varchar_20; -- 输出 '[hello]'，两边无多余空
格
```

兼容性与转换

`VARCHAR` 与 `STRING` 在比较运算符和内置函数中可以互换使用：

```
-- 在内置函数中使用
SELECT LENGTH(CAST('foo' AS VARCHAR(100))); -- 返回 3

-- 不同长度定义的 VARCHAR 进行比较
SELECT CAST('xyz' AS VARCHAR(5)) > CAST('abc' AS VARCHAR(10)); -- 返回 1
```

STRING

最近更新时间：2026-05-06 16:28:12

`STRING` 是一种用于 `CREATE TABLE` 和 `ALTER TABLE` 语句的数据类型，用于存储可变长度的字符序列。

语法

在 `CREATE TABLE` 或 `ALTER TABLE` 语句的列定义中：

```
column_name STRING
```

长度限制与性能建议

虽然 TCHouse-X 支持 `VARCHAR(max_length)` 或 `CHAR(length)` 以实现精确的长度控制，但出于性能最佳化考虑，建议在实际应用中尽可能使用 `STRING`。
关于 `STRING` 长度的注意事项：

- 硬限制：单个 `STRING` 值及单行总大小的上限为 2 GB。超过此限制的查询将报错。
- 性能阈值：
  - 处理 32 KB 或更小的字符串时，系统运行最稳定，且不会出现明显的性能或内存问题。
  - 当字符串超过 32 KB 时，性能和内存消耗可能会下降。

字符集支持

为了确保在 TCHouse-X 所有子系统中获得最稳定、完整的支持，**强烈建议将字符串值限制在 ASCII 字符集范围内**。

> ⚠ **注意：**
> 虽然 TCHouse-X 支持存储和查询 UTF-8 数据，但在处理非 ASCII 字符（如中文、特殊符号）时，以下功能可能无法按预期工作：

风险场景	具体表现
长度处理	`CHAR` 或 `VARCHAR` 的自动截断（Truncating）和填充（Padding）可能出现逻辑错误（因字节数与字符数不一致）。
排序与比较	`ORDER BY` 子句及比较运算符（如 `>`，`<`）可能无法按语言习惯正确排序。
分区管理	将非 ASCII 字符串作为分区键（Partition Key）可能导致数据检索异常。

> 若您的业务涉及多国语言特性（如特定的排序规则、扩展 ASCII 变体 ISO-8859-1 等），将排序、格式化或显示逻辑移至应用程序端实现，而非依赖数据库引擎。

数据转换

隐式转换

TCHouse-X 不会自动将 STRING 转换为任何数值类型。如果格式匹配，TCHouse-X 会自动将 `STRING` 转换为 `TIMESTAMP` 。

显式转换

- 可使用 `CAST()` 将 `STRING` 转换为 `TINYINT` , `SMALLINT` , `INT` , `BIGINT` , `FLOAT` , `DOUBLE` , `DECIMAL` 或 `TIMESTAMP` 。
- 不能直接将 `STRING` 转换为 `BOOLEAN` 。
- `BOOLEAN` 转 `STRING` 时，true 返回 'true'，false 返回 'false'。

特殊注意事项

分区

虽然使用 `STRING` 列作为分区键很方便（即使内容是数字），但出于性能和可扩展性考虑，强烈建议使用数值列（如 `INT` , `SMALLINT` ）作为分区键（例如 `YEAR` , `MONTH` , `DAY` ）。数值类型的内存占用更少，计算速度更快。

空字符串与 NULL

在 `DISTINCT` 和 `GROUP BY` 等子句中，TChouse-X 将 零长度字符串 (""）、NULL 和 空格 视为三个不同的值。

示例

字面量与符号转义

```
SELECT  'I am a single-quoted string';
SELECT  "I am a double-quoted string";
SELECT  'I\'m a single-quoted string with an apostrophe';   -- 使用单引号，并对内部单引号进行转义
SELECT  "I\'m a double-quoted string with an apostrophe";   -- 使用双引号，并对内部单引号进行转义
SELECT  'I am a "short" single-quoted string containing quotes';  -- 使用双引号，内部单引号不需要转义
```

```
SELECT  "I am a \"short\" double-quoted string containing quotes";-- 使
用双引号，并对内部双引号进行转义
```

字符串函数

```
-- 拼接与转换
SELECT CONCAT("Counting: ", CAST(3 AS STRING), ' little pigs.');
-- 截取子串
SELECT SUBSTR("hello world", 7, 5); -- 返回 'world'
```

表操作

```
CREATE TABLE t1 (s1 STRING, s2 STRING);
INSERT INTO t1 VALUES ("hello", 'world'), ('7', "wonders");
-- 使用 LIKE 过滤
SELECT s1, s2, length(s1) FROM t1 WHERE s2 LIKE 'w%';
```

BOOLEAN

最近更新时间：2026-05-06 16:28:12

`BOOLEAN` 数据类型用于 `CREATE TABLE` 和 `ALTER TABLE` 语句中，表示单一的“真/假”选择。

语法

在 `CREATE TABLE` 语句的列定义中：

```
column_name BOOLEAN
```

取值范围

- 取值： `TRUE` 或 `FALSE` 。
- 书写规范：字面量 `TRUE` 和 `FALSE` 不需要加引号。你可以使用大写、小写或混写。
- 查询显示：TCHouse-X 遵循 MySQL 协议规范，布尔值以整数表示：
  - 1：表示 TRUE
  - 0：表示 FALSE

类型转换

TCHouse-X 不会自动将任何其他类型隐式转换为 `BOOLEAN` 。所有转换必须使用 `CAST()` 函数进行显式调用。

数值类型与 BOOLEAN 的转换

- 从数值转 BOOLEAN：任何整数或浮点类型都可以转换为 `BOOLEAN` 。0 表示 `false` ，任何非零值都会转换为 `true` 。

```
SELECT CAST(42 AS BOOLEAN) AS nonzero_int,
       CAST(99.44 AS BOOLEAN) AS nonzero_decimal,
       CAST(000 AS BOOLEAN) AS zero_int,
       CAST(0.0 AS BOOLEAN) AS zero_decimal;
-- 结果依次为: 1, 1, 0, 0
```

- 从 BOOLEAN 转数值：结果将变为 1 (true) 或 0 (false)。

```
SELECT CAST(true AS INT) AS true_int,
       CAST(false AS DOUBLE) AS false_double;
```

```
-- 结果依次为: 1, 0
```

- **DECIMAL 类型：**可以将 `DECIMAL` 转换为 `BOOLEAN`（遵循非零即真的规则），但不能直接将 `BOOLEAN` 转换为 `DECIMAL`。

```
select cast(cast (1 as decimal(3,1)) as boolean); -- 结果为1
select cast(true as decimal); -- 报错
```

其他类型转换

- **STRING：**不能将 STRING 转换为 BOOLEAN。但可以将 BOOLEAN 转换为 STRING，其中 true 返回 'true'，false 返回 'false'。

```
select cast('abc' as boolean); -- 报错
select cast(false as string); -- 返回 'false'
select cast(true as string); -- 返回 'true'
```

- **TIMESTAMP：**虽然支持互转，但结果通常没有实际意义。

NULL 值处理

如果表达式中的任何参数为 `NULL`，则该类型的表达式结果也为 `NULL`。

DATE

最近更新时间：2026-05-06 16:28:12

`DATE` 类型用于存储不包含时间部分的日期值（年、月、日）。

范围与定义

- 取值范围：`0001-01-01` 至 `9999-12-31`。
- 标准格式：遵循 ISO 8601 标准的 `YYYY-MM-DD`。

语法与字面量 (Literals)

声明一个 `DATE` 字面量时，必须使用关键字 `DATE` 后紧跟单引号括起来的字符串：`DATE '2013-01-01'`

显式转换

通过显式转换，可以精准控制 `DATE` 与其他类型的交互。

转换方向	规则说明	示例
`TIMESTAMP` → `DATE`	提取日期部分，丢弃所有时间信息（时、分、秒、微秒）。	`CAST(TIMESTAMP '2023-10-01 12:00:00' AS DATE)` → `2023-10-01`
`STRING` → `DATE`	字符串必须符合 `yyyy-MM-dd` 或带有时间的完整格式。时间部分将被静默忽略。格式错误将报错。	`CAST('2023-10-01 10:30:00' AS DATE)` → `2023-10-01`
`DATE` → `TIMESTAMP`	补全时间部分，默认设为当天凌晨 00:00:00。	`CAST(DATE '2023-10-01' AS TIMESTAMP)` → `'2023-10-01 00:00:00'`
`DATE` → `STRING`	输出格式固定的日期字符串。	`CAST(DATE '2023-10-01' AS STRING)` → `'2023-10-01'`

隐式转换

在特定上下文中，系统会自动执行以下转换：

- `STRING` → `DATE`：当字符串符合 `yyyy-MM-dd` 或 `yyyy-MM-dd HH:mm:ss.SSSSSSSSS` 模式且目标字段为 `DATE` 类型时。
- `DATE` → `TIMESTAMP`：当 `DATE` 值参与需要 `TIMESTAMP` 类型的运算（如与时间戳对比）时，会自动补全时间戳。

TIMESTAMP/TIMESTAMPTZ

最近更新时间：2026-05-06 16:28:12

`TIMESTAMPTZ`（Time Stamp with Time Zone）用于存储包含日期、时间的完整数值，并能够自动处理时区转换。

> **说明：**
> 提示：在 TCHouse-X 中，`TIMESTAMP` 是 `TIMESTAMPTZ` 的别名。

核心属性

- **字段构成：**可拆分为年、月、日、时、分、秒。
- **时区支持：**包含本地时区信息。默认情况下，系统以 UTC 格式存储，但在写入、读取或与系统时间交互时，使用本地时区进行解释。
- **存储精度：**支持 微秒（Microsecond）级别。
- **取值范围：**`0001-01-01` 至 `9999-12-31`。超出此范围的值将转换为 NULL。

语法说明

列定义

在 `CREATE TABLE` 语句中，以下两种写法等效：

```
column_name TIMESTAMP
column_name TIMESTAMPTZ
```

运算表达式

支持使用 `INTERVAL` 关键字或日期函数进行时间偏移计算：

```
timestamp_value [+|-] INTERVAL 'interval_expression'
```

- `interval_expression` 定义见下方 INTERVAL 表达式介绍。

INTERVAL 表达式

可以使用 `INTERVAL` 配合运算符实现灵活的时间加减。

- **支持单位：**`YEAR[S]`，`MONTH[S]`，`WEEK[S]`，`DAY[S]`，`HOUR[S]`，`MINUTE[S]`，`SECOND[S]`。
- **限制条件：**
  - `INTERVAL` 表达式仅能指定一种单位。

- `INTERVAL` 数值需要为整数。

- **复合运算示例：**

```
-- 通过连续计算实现任意粒度的偏移
SELECT CAST('2025-01-01 01:00:00' as TIMESTAMP) + INTERVAL '3 WEEKS' -
INTERVAL '1 DAY';
```

数据转换与格式规范

字符串解析

TCHouse-X 会自动解析符合特定格式的字符串。

- **标准格式：** `'yyyy-MM-dd HH:mm:ss.SSSSSS'`。
- **弹性规则：**
  - 可仅包含日期（如 `'1966-07-30'`）。
  - 小数秒部分可省略。
  - 各部分数字的前导零可省略（如 `'2018-1-1 1:2:3'`）。
  - 时区指定：支持带时区名的字面量，如 `'2024-05-20 14:00:00 America/New_York'`。

转换时的特殊处理

1. **空白字符：**自动忽略字符串两端的空格、制表符（`\t`）、换行（`\n`）或回车（`\r`）。
2. **分隔符：**在 `CAST()` 显式转换中，日期与时间之间支持使用“一个或多个空格”或“字符 T”。
3. **安全性：**如果字符串格式不符合要求，转换结果为 `NULL` 而不会触发系统报错。
4. **2038年问题：**在涉及 UNIX 时间戳的函数（如 `FROM_UNIXTIME`）中，系统内部使用 `BIGINT` 处理，彻底规避了传统 32 位 `INT` 溢出的风险。

分区注意事项

`TIMESTAMP` 列不可以作为分区键。

基础转换与函数示例

```
-- 显式转换示例
SELECT CAST('1966-07-30' AS TIMESTAMP);

SET TIMEZONE = 'Asia/Shanghai';
SELECT CAST('2024-05-20 14:00:00 America/New_York' AS TIMESTAMP); -- 返回
2024-05-21 02:00:00
```

```
-- 将按本地时区(如 Asia/Shanghai)返回对应时间，纽约使用东部时区（ET），且2024年5月
20日处于夏令时（EDT，UTC-4），
-- 北京时间是 UTC+8，所以纽约时间比北京时间晚12个小时

-- 常用日期函数
SELECT HOUR('1970-01-01 15:30:00');     -- 返回 15
SELECT DAYOFWEEK('2004-06-13');         -- 返回 1 （星期日）
SELECT DATE_ADD('2004-06-13', 365);     -- 返回 '2005-06-13 00:00:00'
SELECT DATEDIFF('1989-12-31', '1984-09-01'); -- 计算间隔天数
SELECT NOW();                           -- 返回当前本地时间
```

表操作示例

```
CREATE TABLE dates_and_times (t TIMESTAMP);
INSERT INTO dates_and_times VALUES
  ('1966-07-30'),
  ('1985-09-25 17:45:30.005'),
  ('08:30:00');

SELECT * FROM dates_and_times;
+-------------------------+
| t                       |
+-------------------------+
| 1966-07-30 00:00:00     |
| 1985-09-25 17:45:30.005 |
| NULL                    |
+-------------------------+
```

TIMESTAMPNTZ

最近更新时间：2026-05-06 16:28:12

`TIMESTAMPNTZ` 用于存储纯粹的日期和时间值。它不包含时区信息，也不随系统时区的改变而变化，仅代表字面上的时间点。

核心属性

- **字段构成：**包含年、月、日、时、分、秒。
- **时区无关性：**在写入、读取或函数转换时，均忽略时区。即使输入字符串包含时区后缀，也会被自动截断/忽略。
- **存储精度：**支持 微秒 (Microsecond) 级别。
- **取值范围：** `0001-01-01` 至 `9999-12-31` 。

语法说明

在 `CREATE TABLE` 语句中，以下两种写法等效：

```
column_name TIMESTAMPNTZ
column_name TIMESTAMP_NTZ
```

运算表达式

支持使用 `INTERVAL` 关键字或日期函数进行时间偏移计算：

```
timestamp_value [+|-] INTERVAL 'interval_expression'
```

INTERVAL 表达式

可以使用 `INTERVAL` 配合运算符实现灵活的时间加减。

- **支持单位：** `YEAR[S]` , `MONTH[S]` , `WEEK[S]` , `DAY[S]` , `HOUR[S]` , `MINUTE[S]` , `SECOND[S]` 。
- **限制条件：**每个 `INTERVAL` 表达式仅能指定一种单位。
- **复合运算示例：**

```
-- 通过连续计算实现任意粒度的偏移
timestamp_value + INTERVAL '3 WEEKS' - INTERVAL '1 DAY'
```

数据转换规范

字符串解析

- **标准格式：** `'yyyy-MM-dd HH:mm:ss.SSSSSS'`
- **兼容性：** 支持前导零省略（如 `2018-1-1`）以及日期或时间的单独输入。
- **弹性规则：**
  - **忽略时区：** 例如 `'1999-12-01 01:02:03 America/New_York'` 会被解析为 `'1999-12-01 01:02:03'`。
  - **忽略空白：** 自动忽略字符串两端的空格、制表符（`\t`）、换行（`\n`）或回车（`\r`）。
  - **灵活分隔符：** `CAST` 转换时，日期与时间之间可用 `空格` 或 `T` 分隔

转换时的特殊处理

- **NULL 处理：** 若字符串格式无法识别或数值非法（如小时超出 23），结果返回 `NULL` 而不报错。
- **2038 问题：** 相关函数底层采用 BIGINT 处理，避免了传统 32 位整型的时间溢出风险。

分区注意事项

`TIMESTAMPNTZ` 列不可以作为分区键。

查询与操作示例

基础转换与函数示例

```
-- 自动忽略字符串中的时区信息
SELECT CAST('2024-05-20 14:00:00 America/New_York' AS TIMESTAMPNTZ);
-- 返回: 2024-05-20 14:00:00 (不会进行时区转换)

-- 非法格式返回 NULL
SELECT HOUR('1970-01-01 27:30:00'); -- NULL (小时越界)

-- 日期加减
SELECT DATE_ADD('2004-06-13', 365); -- 2005-06-13 00:00:00
```

表操作示例

```
CREATE TABLE dates_and_times (t TIMESTAMPNTZ);
INSERT INTO dates_and_times VALUES
  ('1966-07-30'),
  ('1985-09-25 17:45:30.005'),
  ('08:30:00');

SELECT * FROM dates_and_times;
```

```
+-------------------------+
| t                       |
+-------------------------+
| 1966-07-30 00:00:00     |
| 1985-09-25 17:45:30.005 |
| NULL                    |
+-------------------------+
```

TCHouse-X 语法
DDL
DDL 语法概览

最近更新时间：2026-05-06 16:28:12

DDL (Data Definition Language) 是 SQL 的核心子集，专门用于定义或修改数据库的逻辑结构（Schema）。它不直接操作具体的业务数据，而是管理存储数据的“容器”及其元数据。
在 TCHouse-X 中，DDL 操作通常涉及对象的创建、变更和清理，绝大多数语句以 `CREATE`、`ALTER` 或 `DROP` 关键字开头。

TCHouse-X DDL 语句分类表

操作对象	创建 (Create)	修改 (ALTER)	删除 (DROP)
数据库 (DATABASE)	CREATE DATABASE 语句	-	DROP DATABASE 语句
表 (TABLE)	CREATE TABLE 语句	ALTER TABLE 语句	DROP TABLE 语句
视图 (VIEW)	CREATE VIEW 语句	ALTER VIEW 语句	DROP VIEW 语句
统计信息 (Stats)	COMPUTE STATS 语句	-	DROP STATS 语句

CREATE DATABASE 语句

最近更新时间：2026-05-06 16:28:12

`CREATE DATABASE`（或 `CREATE SCHEMA`）语句用于在 TCHouse-X 中创建新的数据库。

在 TCHouse-X 中，数据库具有双重性质：

- 逻辑构造：用于在独立的命名空间内将相关的表、视图和函数进行分组。通常为每个应用、相关表集或实验轮次使用不同的数据库。
- 物理构造：在底层存储上对应一个目录树。该目录内部用于存放数据库下的内表、分区和数据文件。

> **注意：**
> 如果要使用与 TCHouse-X 保留关键字 相同的标识符作为数据库、表、列或视图名称，必须用反引号 `` ` `` 包裹该标识符。

语法

```
CREATE (DATABASE | SCHEMA) [IF NOT EXISTS] database_name [COMMENT
'database_comment']
```

语法说明

关键字/参数	说明
`IF NOT EXISTS`	可选。如果指定的数据库已存在，则语句不会抛出错误，而是成功执行并发出警告。
`database_name`	必需。 新数据库的名称。
`COMMENT`	可选。为数据库添加描述性注释。

使用说明

- **默认数据库**：初次连接 TCHouse-X 时，默认所在的数据库是 `default`。
- **切换数据库**：创建数据库后，可以使用 `USE database_name` 语句切换当前操作的数据库。切换后，在引用该数据库下的表时，可以省略数据库名前缀。
- **数据库引用**：即使不切换，您也可以始终使用完全限定名（`db_name.table_name`）来引用任何数据库中的表。
- **删除数据库限制**：

- 您不能删除当前正在使用（`USE`）的数据库。
- 在删除数据库之前，必须先删除其中的所有表。或者，可以使用 `CASCADE` 子句强制删除数据库及其所有内容。

示例

```
-- 创建并使用第一个数据库
CREATE DATABASE first_db;
USE first_db;
CREATE TABLE t1 (x int);

-- 创建并使用第二个数据库，展示命名空间隔离 (Namespace Isolation)
CREATE DATABASE second_db;
USE second_db;
-- 每个数据库都有独立的表命名空间，因此表名可以重复使用
CREATE TABLE t1 (s string);

-- 创建一个临时数据库
CREATE DATABASE temp_db;

-- 引用表的两种方式：
-- 方法 1：使用全限定名（数据库名.表名）在 temp_db 中创建表 t2
CREATE TABLE temp_db.t2 (x int, y int);

-- 方法 2：先切换数据库上下文，然后在 temp_db 中创建表 t3
USE temp_db;
CREATE TABLE t3 (s string);

-- 尝试删除当前正在使用的数据库（操作会失败）
DROP DATABASE temp_db;
-- 错误示例：ERROR: AnalysisException: Cannot drop current default
database: temp_db

-- 切换到 default 数据库，以便删除其他数据库
USE default;

-- 尝试删除一个非空数据库（操作会失败；需要先删除表，或使用 CASCADE 关键字）
DROP DATABASE temp_db;
-- 错误示例：CAUSED BY: InvalidOperationException: Database temp_db is not
empty
```

```
SHOW TABLES IN temp_db;
/*
+------+
| name |
+------+
| t2   |
| t3   |
+------+
*/

-- 推荐方法：使用 CASCADE 关键字进行级联强制删除
DROP DATABASE temp_db CASCADE;

-- 传统或谨慎的方法：先手动删除所有表，然后再删除数据库
DROP TABLE temp_db.t2;
DROP TABLE temp_db.t3;
DROP DATABASE temp_db;
```

DROP DATABASE 语句

最近更新时间：2026-05-06 16:28:12

`DROP DATABASE`（或 `DROP SCHEMA`）语句用于将一个数据库从系统中永久删除。
此操作会执行以下物理操作：
1. 移除元数据： 从元存储（Metastore）中移除数据库的所有元数据信息。
2. 删除数据： 物理删除与该数据库关联的目录以及其中存储的内部表数据文件。

语法

```
DROP (DATABASE|SCHEMA) [IF EXISTS] database_name [RESTRICT | CASCADE];
```

语法说明

关键字	说明
`IF EXISTS`	可选。如果指定的数据库不存在，则避免抛出错误。
`database_name`	必需。 要删除的数据库名称。
`RESTRICT`	默认行为。 强制要求数据库在删除前必须为空。如果数据库包含任何对象（表、视图等），操作将失败。
`CASCADE`	强制删除数据库中的所有对象（表、视图等），然后删除数据库本身。推荐用于快速清理。

使用说明

默认行为 (RESTRICT)

默认情况下，数据库必须是空的才能被删除，以防止意外数据丢失。
手动清理要求： 如果不使用 `CASCADE`，您必须手动删除或移动数据库中的所有对象：

- 使用 `SHOW TABLES` 定位所有对象，然后使用 `DROP TABLE` 和 `DROP VIEW` 删除它们。
- 如需保留某些对象，可使用 `ALTER TABLE` 或 `ALTER VIEW` 将它们移动到另一个数据库。

级联删除 (CASCADE)

使用 `CASCADE` 子句时，TCHouse-X 会自动删除数据库中的所有表和对象。
注意外部表： `CASCADE` 导致的自动删除遵循标准的 `DROP TABLE` 和 `DROP VIEW` 规则。特别是，任何外部表的目录和底层数据文件在删除时会被保留。

当前数据库限制

您不能删除当前会话正在使用的数据库。

- 操作要求： 在执行 `DROP DATABASE` 之前，必须先使用 `USE` 语句切换到另一个数据库。
- 便捷切换： 始终可用的 `default` 数据库是退出当前数据库并执行删除操作的便捷目标，即执行 `USE default;` 。

示例

```
-- 准备工作：创建并填充两个示例数据库
CREATE DATABASE temp_to_drop;
USE temp_to_drop;
CREATE TABLE t1 (x int);
CREATE TABLE t2 (s string);

USE default; -- 切换到 default 数据库，以便执行删除 temp_to_drop 的操作

-- 1. 尝试使用默认行为 (RESTRICT) 删除一个非空数据库（操作会失败）
DROP DATABASE temp_to_drop;
-- 错误示例：ERROR: InvalidOperationException: Database temp_to_drop is
not empty, One or more tables exist.

-- 2. 推荐方法：使用 CASCADE（级联）关键字强制删除数据库及其所有内部对象
DROP DATABASE temp_to_drop CASCADE;
-- 成功删除数据库及其内部的表 t1 和 t2。

-- 3. 传统或谨慎的方法（手动清理）
CREATE DATABASE another_temp;
USE another_temp;
CREATE TABLE t3 (y int);
USE default;

-- 步骤 3a：手动删除该数据库中的所有表
DROP TABLE another_temp.t3;

-- 步骤 3b：删除此时已经为空的数据库
DROP DATABASE another_temp;
```

CREATE VIEW 语句

最近更新时间：2026-05-06 16:28:12

`CREATE VIEW` 语句允许您为复杂的查询创建一个简洁的别名（即视图）。视图纯粹是一个逻辑构造，不包含任何底层数据。

- 逻辑构造： 视图是基础查询的别名。基础查询可以包含连接（`JOIN`）、表达式、列重命名、列别名等复杂的 SQL 特性。
- 元数据操作： 由于视图没有物理数据，CREATE VIEW 或后续的 ALTER VIEW 操作仅修改元存储数据库中的元数据，不会影响存储系统中的任何数据文件。
- 平台兼容性提示 (TCHouse-X)：在使用与 TCHouse-X 保留关键字相同的数据库、表、视图或列名称时，必须使用反引号 (``) 将该标识符括起来。

语法

```
CREATE VIEW [IF NOT EXISTS] view_name
    [(column_name [COMMENT 'column_comment'][, ...])]
    [COMMENT 'view_comment']

  AS select_statement
```

语法说明

关键字/参数	说明
`IF NOT EXISTS`	可选。如果视图已存在，则使用此选项可以避免抛出错误。语句会成功执行，但会发出一个警告。
`view_name`	必需。要创建的视图的名称。
`column_name`	可选。为视图的列指定自定义名称，也可以添加列级注释。如果省略，则使用 `select_statement` 结果集中的列名。
`COMMENT 'view_comment'`	可选。为视图添加描述性注释。
`AS select_statement`	必需。定义视图的基础查询。

使用说明

`CREATE VIEW` 语句在以下场景中提供了显著优势：

1. **简化复杂查询：**
   - 将冗长、复杂的 SQL 查询（涉及多表连接、复杂表达式等）简化为一行简单的 `SELECT * FROM view_name;` 查询，提高可读性和维护性。
   - 应用程序只需对视图发出简单查询：

```
select * from view_name;
select * from view_name order by c1 desc limit 10;
```

2. **降低维护成本（抽象层）：**
   - 隐藏底层结构： 隐藏基础表和列的名称。
   - 如果底层表名或列名发生更改，只需重新创建视图。所有使用该视图的应用程序查询仍可继续运行，无需任何修改。
3. **集中优化策略：**
   - 集中实现复杂的优化技术（如最佳的 WHERE 条件、连接顺序、连接提示等）。
   - 应用程序只需查询视图，即可受益于这些最优策略。如果找到更优的优化方法，只需重新创建视图，所有应用程序即可即时获得性能提升。
4. **模块化和代码复用：**
   - 简化需要多次重复复杂子句（如多表连接、复杂过滤）的一整类相关查询。视图可以作为这些复杂逻辑的封装模块。

> **说明：**
> 对于仅在单个查询中需要复用复杂子句的场景，可以考虑使用 WITH 子句（Common Table Expression, CTE）作为创建视图的替代方案。

示例

```
-- 创建一个与底层表完全相同的视图
CREATE VIEW v1 AS SELECT * FROM t1;

-- 创建一个仅包含底层表中特定列的视图（列级过滤）
CREATE VIEW v2 AS SELECT c1, c3, c7 FROM t1;

-- 创建一个包含过滤条件的视图（行级与去重过滤）
CREATE VIEW v3 AS SELECT DISTINCT c1, c3, c7 FROM t1 WHERE c1 IS NOT
NULL AND c5 > 0;
```

```
-- 创建一个对底层表列进行重排序和重命名的视图
CREATE VIEW v4 AS SELECT c4 AS last_name, c6 AS address, c2 AS
birth_date FROM t1;

-- 创建一个通过运行函数来转换或加工特定列的视图
CREATE VIEW v5 AS SELECT c1, CAST(c3 AS STRING) c3, CONCAT(c4,c5) c5,
TRIM(c6) c6, "Constant" c8 FROM t1;

-- 创建一个用于隐藏复杂查询逻辑（如多表关联）的视图
CREATE VIEW v6 AS SELECT t1.c1, t2.c2 FROM t1 JOIN t2 ON t1.id = t2.id;

-- 创建一个包含列注释和表注释的视图
CREATE VIEW v7 (c1 COMMENT 'c1 的注释', c2) COMMENT 'v7 视图的注释' AS
SELECT t1.c1, t1.c2 FROM t1;
```

ALTER VIEW 语句

最近更新时间：2026-05-06 16:28:12

`ALTER VIEW` 语句用于更改现有视图的定义或特征。

核心概念与影响

- **逻辑操作：** 视图是纯粹的逻辑构造（基础查询的别名），没有底层物理数据。
- **元数据更改：** `ALTER VIEW` 操作仅更改元存储数据库中的元数据，不会影响 TCHouse-X 中的任何数据文件。
- **验证更改：** 要查看视图更新后的定义和元数据，请执行 `DESCRIBE FORMATTED [database_name.]view_name` 语句。

语法

`ALTER VIEW` 语句支持以下两种主要操作：

```
-- 1. 修改视图的底层查询和列定义
ALTER VIEW [database_name.]view_name
    [(column_name [COMMENT 'column_comment'][, ...])]
    AS select_statement;

-- 2. 重命名或移动视图
ALTER VIEW [database_name.]view_name
    RENAME TO [database_name.]new_view_name;
```

修改视图定义

用途说明

使用 `AS` 子句可以更改视图的基础查询，同时可选择性地指定新的列名和列级注释。

示例

```
-- 更改视图 v1 的底层查询（列名默认为跟随查询结果）
ALTER VIEW v1 AS
    SELECT x, UPPER(s) AS new_s_column
    FROM t2;

-- 更改视图 v1 的底层查询并显式指定新的列名
```

```
ALTER VIEW v1 (c1, c2) AS
    SELECT x, UPPER(s) AS s_upper
    FROM t2;

-- 更改视图 v7 的底层查询并指定带有注释的列名
ALTER VIEW v7 (c1 COMMENT 'New comment for c1', c2) AS
    SELECT t1.c1, t1.c2
    FROM t1;
```

重命名或移动视图

用途说明

`RENAME TO` 子句用于更改视图的名称，或将其移动到另一个数据库，或同时执行这两项操作。

示例

操作	语句	描述
重命名并移动	`ALTER VIEW db1.v1 RENAME TO db2.v2;`	将视图 `db1.v1` 移动到 `db2`，并重命名为 `v2`。
仅重命名	`ALTER VIEW db1.v1 RENAME TO db1.v2;`	在数据库 `db1` 内，将视图从 `v1` 重命名为 `v2`。
仅移动	`ALTER VIEW db1.v1 RENAME TO db2.v1;`	将视图 `v1` 从 `db1` 移动到 `db2`，名称保持不变。

DROP VIEW 语句

最近更新时间：2026-05-06 16:28:12

`DROP VIEW` 语句用于删除一个先前通过 `CREATE VIEW` 语句创建的视图。

核心概念

视图（View）本质上是一个逻辑构造（一个查询的别名），它本身不包含任何物理数据。因此，执行 DROP VIEW 操作：

- 仅涉及元存储数据库（Metastore）中的元数据更改。
- 不会影响存储系统中的任何数据文件。
- 不会删除视图所引用的基础表中的数据。

删除指定的视图，该视图最初是通过 `CREATE VIEW` 语句创建的。由于视图纯粹是一个逻辑构造（查询的别名），没有任何物理数据，因此 `DROP VIEW` 仅涉及元存储数据库中的元数据更改，而不会影响存储系统中的任何数据文件。

语法

```
DROP VIEW [IF EXISTS] [db_name.]view_name
```

关键字/参数	说明
`IF EXISTS`	可选。如果视图不存在，则使用此选项可以避免抛出错误。语句会成功执行，但会发出一个警告。
`db_name`	可选。指定视图所在的数据库名称。如果省略，则默认在当前数据库中查找视图。
`view_name`	必需。要删除的视图的名称。

示例

以下示例演示了如何创建和删除视图，并说明了如何通过数据库限定符（db_name.）来引用或操作不同数据库中的视图。

```
-- 在当前数据库中创建并删除视图
CREATE VIEW few_rows_from_t1 AS SELECT * FROM t1 LIMIT 10;
DROP VIEW few_rows_from_t1;
```

```
-- 创建并删除一个引用其他数据库中表的视图
CREATE VIEW table_from_other_db AS SELECT x FROM db1.foo WHERE x IS NOT
NULL;
DROP VIEW table_from_other_db;

USE db1;
-- 在另一个数据库中创建视图
CREATE VIEW db2.v1 AS SELECT * FROM db2.foo;
-- 切换到另一个数据库并删除视图
USE db2;
DROP VIEW v1;

USE db1;
-- 在另一个数据库中创建视图
CREATE VIEW db2.v1 AS SELECT * FROM db2.foo;
-- 直接删除另一个数据库中的视图（无需切换）
DROP VIEW db2.v1;
```

CREATE TABLE 语句

最近更新时间：2026-05-06 16:28:12

CREATE TABLE 语句用于创建新表并指定其所有属性。
在创建表时，您可以根据需要指定以下关键方面：

- 表的类型：内表还是外表，区别参见 内表和外表说明 。
- 表的结构：列名、数据类型、注释、约束。
- 表的物理组织：分区和排序策略。
- 对于外表：数据文件的存储格式和位置。

> **注意：**
> 如果要使用与 TCHouse-X 保留关键字相同的数据库、表、列或视图名称，请务必使用反引号 (``) 将该标识符括起来。

创建内表

CREATE TABLE

内表是 TCHouse-X 默认创建的表类型，由 TCHouse-X 完全管理底层数据文件。执行 DROP TABLE 时，数据文件也会被物理删除。

语法

```
CREATE TABLE [IF NOT EXISTS] [db_name.]table_name
  (col_name data_type [constraint_specification] [COMMENT 'col_comment']
[, ...]
)
  [PARTITIONED BY SPEC partition_clause]
  [SORT BY ([column [, column ...]])]
  [COMMENT 'table_comment']
```

语法说明

关键字/参数	说明
`IF NOT EXISTS`	可选。如果表已存在，则使用此选项可以避免抛出错误。语句会成功执行，但会发出一个警告。
`db_name`	可选。若未指定，则默认使用当前会话连接的数据库。

`table_name`	必需。要创建的表的名称。
`column_name`	必选。为表的列指定名称。
`data_type`	必选。为表的列指定类型。
`COMMENT 'col_comment'`	可选。为表的列添加描述性注释。

属性详解

表约束 (constraint_specification)

```
constraint_specification:
    PRIMARY KEY (col_name, ...)
```

- **当前支持：** 内表目前仅支持主键约束 (`PRIMARY KEY`)。
- **限制：** 不支持 `FLOAT` 和 `DOUBLE` 类型的列作为主键， `BOOLEAN` 不能单独作为主键。

分区和分桶 (PARTITIONED BY SPEC)

内表使用 `PARTITIONED BY SPEC` 子句创建分区表，支持分区和分桶两种方式。

```
partition_clause:
      column_name                                   -- 仅分区
    | BUCKET(bucket_number, column_name)            -- 分桶
```

目的： TCHouse-X 利用定义的分区和分桶键，对数据文件进行物理组织。查询时可利用分区元数据，仅扫描匹配的分区，尤其在关联查询时能显著减少 I/O。

分区限制：

- `TIMESTAMP/TIMESTAMPTZ` 、 `TIMESTAMPNTZ` 类型的列不能作为分区键

分桶限制 (BUCKET):

- 只能指定一个列作为分桶键。
- `BOOLEAN` 、 `FLOAT` 、 `DOUBLE` 类型的列不能作为分桶键。
- 如果表定义了主键，则该主键必须包含分桶键。

排序 (SORT BY)

使用 `SORT BY` 子句定义表数据在存储时按照指定字段进行有序存储。

效益： 排序后，数据文件内部记录的列最小/最大值分布更集中。TCHouse-X 可根据 `WHERE` 条件快速跳过不包含目标范围的文件，并提升数据存储的编码和压缩效果。

默认行为：

- 如果定义了主键，但未指定排序键，则默认使用主键作为排序键。
- 如果主键和排序键都未指定，则没有排序键。
- 如果没有显示指定分桶键，则默认以下为分桶键：
  - 如果指定了主键，则主键为分桶键
  - 如果没有主键，则默认为随机分桶

示例

```
-- 创建非主键表
CREATE TABLE test_simple(c1 int, c2 string);

-- 创建按 c2 和 c3 分区的非主键表
CREATE TABLE test_partitioned(c1 int, c2 string, c3 string)
    PARTITIONED BY SPEC (c2, c3);

-- 创建分区、分桶并排序的表：按 c2 分区，按 c3 分桶，按 c1 排序
CREATE TABLE test_all(c1 int, c2 string, c3 string, c4 date)
    PARTITIONED BY SPEC (c2, BUCKET(16, c3))
    SORT BY(c1);

-- 创建单列主键表
CREATE TABLE test_pk_simple(c1 int primary key, c2 string);

-- 创建复合主键表
CREATE TABLE test_pk_composite(c1 int, c2 string, c3 int, primary
key(c1, c3));

-- 创建包含主键、分区、分桶和排序的复杂内表
CREATE TABLE test_pk_complex(c1 int, c2 string, c3 string, c4 date,
primary key(c1,c2))
    PARTITIONED BY SPEC(c3, BUCKET(16, c1))
    SORT BY(c4);
```

CREATE TABLE AS SELECT

`CREATE TABLE AS SELECT`（简称 `CTAS`）是一种高效的简写语法。它允许您基于另一个查询的结果创建一张新表，并同时将源数据复制到新表中，从而省去了单独执行 `INSERT` 语句的步骤。

功能

`CTAS` 可用于以下目的：

- 复制表结构和数据： 快速克隆现有表。
- 创建数据子集： 使用 `WHERE` 子句仅复制满足特定条件的行。
- 转换和清洗数据： 在 `SELECT` 语句中应用函数、连接和表达式来转换数据。
- 动态分区： 根据源表查询结果的列值来定义新表的分区。

语法

```
CREATE TABLE [IF NOT EXISTS] [db_name.]table_name
  [PRIMARY KEY (col_name [, ...])]
  [PARTITIONED BY SPEC partition_clause]
  [SORT BY ([column [, column ...]])]
  [COMMENT 'table_comment']
AS
  select_statement
```

语法说明

关键字/参数	说明
`IF NOT EXISTS`	可选。如果表已存在，则使用此选项可以避免抛出错误。语句会成功执行，但会发出一个警告。
`db_name`	可选。若未指定，则默认使用当前会话连接的数据库。
`table_name`	必需。要创建的表的名称。
`PRIMARY KEY`	可选。新表指定主键。
`PARTITIONED BY SPEC`	可选。新表指定分区键。
`SORT BY`	可选。新表指定排序键。
`COMMENT 'table_comment'`	可选。为表添加描述性注释。
`select_statement`	必选。定义了新表的结构和数据来源。有关查询语法的详细信息，请参阅 SELECT 语句 。

⚠ 注意：

列名继承：新创建的表会继承 SELECT 语句结果集的列名。您可以使用列别名（AS new_name）来覆盖它们。
元数据继承：原表的列注释和表注释不会被带到新表中。
原子性：
CTAS 不是原子操作。它实际分两步执行：
1. `CREATE TABLE`
2. `INSERT INTO SELECT`

如果 `INSERT INTO` 步骤失败，只会回滚 `INSERT INTO` 操作，而 `CREATE TABLE` 语句依然是成功的。

示例

以下示例演示了 `CTAS` 的各种用法，包括数据复制、转换和动态分区创建。

```
-- 准备源表数据
CREATE TABLE t1 (x INT, y STRING);
INSERT INTO t1 VALUES (1, 'one'), (2, 'two'), (3, 'three');

-- 1. 克隆所有列和数据（全量复制）
CREATE TABLE clone_of_t1 AS
    SELECT * FROM t1;

-- 2. 按条件复制行（创建子集）
CREATE TABLE subset_of_t1 AS
    SELECT * FROM t1 WHERE x >= 2;

-- 3. 只克隆表结构，不复制数据（空表克隆）
CREATE TABLE empty_clone_of_t1 AS
    SELECT * FROM t1 WHERE 1=0;

-- 4. 重新组织、重命名列并转换数据
CREATE TABLE t5 AS
    SELECT
        upper(y) AS s,                    -- 数据转换和重命名
        x+1 AS a,                         -- 表达式计算
        'Entirely new column' AS n        -- 插入新的常量列
    FROM t1;
SELECT * FROM t5;
/*
+-------+---+--------------------+
```

```
| s     | a | n                    |
+-------+---+----------------------+
| ONE   | 2 | Entirely new column  |
| TWO   | 3 | Entirely new column  |
| THREE | 4 | Entirely new column  |
+-------+---+----------------------+
*/

-- 5. 基于源数据创建分区表
CREATE TABLE partitions_no (year smallint, month tinyint, s string);
INSERT INTO partitions_no VALUES
  (2016, 1, 'January 2016'),
  (2016, 2, 'February 2016'),
  (2016, 3, 'March 2016');

-- 创建一个新表，并使用源表中的 year 和 month 列作为分区键
CREATE TABLE partitions_yes PARTITIONED BY SPEC (year, month)
  AS SELECT s, year, month FROM partitions_no;
```

CREATE TABLE LIKE

核心功能

- 克隆结构： 新创建的表会继承源表的完整结构，包括列定义、数据类型、主键、分区规范、排序属性以及所有相关的注释。
- 创建空表： 此语句不会复制源表中的任何数据。

语法

```
CREATE TABLE [IF NOT EXISTS] [target_db_name.]target_table_name
LIKE [source_db_name.]source_table_name
[COMMENT 'table_comment'];
```

关键字说明

关键字/参数	说明
`IF NOT EXISTS`	可选。如果表已存在，则避免抛出错误。

`target_db_name`	可选。目标表的 database 名，若未指定，则默认使用当前会话连接的数据库。
`target_table_name`	必选。目标表的 table 名
`LIKE { ... }`	必需。 指定结构来源：一个已存在的 TCHouse-X 表。
`source_db_name`	可选。源表的 database 名，若未指定，则默认使用当前会话连接的数据库。
`source_table_name`	必选。源表的 table 名
`COMMENT`	可选。唯一支持的附加子句。 用于为新表添加或覆盖表级注释。

> ⚠ **注意：**
> 如果需要在一次操作中**同时克隆表结构和复制数据**，请使用 `CREATE TABLE AS SELECT (CTAS)` 语法。

示例

以下示例演示了如何创建一个继承现有表结构的新表，并为其添加注释。

```
-- 1. 创建源表，包含主键、分区和排序属性
CREATE TABLE test_source(c1 int, c2 string, PRIMARY KEY(c1))
    PARTITIONED BY SPEC(c2)
    SORT BY(c1);

-- 2. 使用 LIKE 创建一个空表 test2，继承 test_source 的所有结构属性，并指定一个新的注释
CREATE TABLE test2 LIKE test_source
    COMMENT 'This is a clone of the structure of test_source.';
```

创建外表

外表 (External Table) 的数据文件通常在 TCHouse-X 外部管理和存储。TCHouse-X 提供了对这些外部数据的查询能力，但不支持修改外部数据（如 `INSERT`，`UPDATE`，`DELETE`）。
TCHouse-X 目前支持两种主要的外部表类型：Hive 外表和 Iceberg 外表。

Hive 外表

Hive 外表用于访问存储在 HDFS 或兼容文件系统上的传统 Hive 格式数据（如 Parquet、Textfile）。

语法

```
CREATE EXTERNAL TABLE [IF NOT EXISTS] [db_name.]table_name
  (col_name data_type [COMMENT 'col_comment']     [, ...]
)
  [PARTITIONED BY partition_clause]
  [SORT BY ([column [, column ...]])]
  [COMMENT 'table_comment']
  [ROW FORMAT row_format]
  STORED AS file_format
  LOCATION 'table_path'

partition_clause
  (col_name data_type, [, ...])

row_format:
    DELIMITED [FIELDS TERMINATED BY 'char' [ESCAPED BY 'char']] [LINES
TERMINATED BY 'char']

file_format:
    PARQUET
  | TEXTFILE
```

关键字说明

关键字	详细定义	适用性/说明
`PARTITIONED BY`	`(col_name data_type, [, ...])`	Hive 外表的分区列需在主列定义后单独指定。
`STORED AS`	`PARQUET`	仅支持 `PARQUET`、`TEXTFILE`
`ROW FORMAT`	`DELIMITED ...`	仅对 `TEXTFILE` 文件格式有效，用于指定字段和行分隔符。
`LOCATION`	`'table_path'`	数据文件在文件系统上的绝对路径。

对于 TEXTFILE 如果数据文件包含 header， 可以通过 skip.header.line.count 表属性，跳过 header 的解析，比如 csv 文件内容如下：

```
TINYINT,SMALLINT,INT,BIGINT,BOOLEAN,FLOAT,DOUBLE,DECIMAL,STRING,CHAR,VAR
CHAR,TIMESTAMP
1,100,1000,1000000,true,3.14,3.1415926535,123.456,Hello World,A,Short
Text,2023-10-10 12:34:56
2,200,2000,2000000,false,2.71,2.7182818284,654.321,Data Lake,B,Longer
Text,2023-10-11 13:45:00
3,300,3000,3000000,true,1.23,1.2345678901,789.123,Big Data,C,Very Long
Text,2023-10-12 14:56:30
```

建表语句如下：

```
CREATE EXTERNAL TABLE text_t (a TINYINT,b SMALLINT, c INT, d BIGINT,e
BOOLEAN,f FLOAT,g DOUBLE,h DECIMAL(10,4),i STRING,j CHAR(30),k
VARCHAR(30),l TIMESTAMP)
ROW FORMAT DELIMITED FIELDS TERMINATED BY ',' STORED AS textfile
LOCATION 'cosn://douglaspeng-cq-
1375622477/shawbozhang_external_table/text_t'
TBLPROPERTIES ('skip.header.line.count'='1');
```

Iceberg 外表

Iceberg 外表用于访问遵循 Apache Iceberg 规范的表。TCHouse-X 支持 Iceberg 的 v1 和 v2 格式，并且目前支持基于 Hadoop 的两种 Catalog 类型和 Hive 的 Catalog 类型。

语法

```
CREATE EXTERNAL TABLE [IF NOT EXISTS] [db_name.]table_name
  [COMMENT 'table_comment']
  STORED AS ICEBERG
  [LOCATION 'table_path']
  [TBLPROPERTIES ('key1'='value1', 'key2'='value2', ...)]
```

HadoopTables 模式

适用于 Iceberg 表的元数据文件（metadata）位于文件系统中的某个特定位置。

示例

```
CREATE EXTERNAL TABLE ice_hadoop_tbl
STORED AS ICEBERG
LOCATION '/path/to/table'
TBLPROPERTIES('iceberg.catalog'='hadoop.tables');
```

示例说明

- `LOCATION` 字段：指定包含 Iceberg 表元数据（如 metadata 文件夹）的路径
- `'iceberg.catalog'` 属性：需设置为 `'hadoop.tables'`

HadoopCatalog 模式

适用于 Iceberg 表存储在一个公共的目录位置下，并使用命名空间 (namespace) 和表标识符 (table identifier) 来引用。

示例

```
CREATE EXTERNAL TABLE ice_hadoop_cat
STORED AS ICEBERG
TBLPROPERTIES('iceberg.catalog'='hadoop.catalog',
'iceberg.catalog_location'='/path/to/catalog',
'iceberg.table_identifier'='namespace.table');
```

示例属性说明

属性	说明
`iceberg.catalog`	设置为 `'hadoop.catalog'`。
`iceberg.catalog_location`	指定 Iceberg Catalog 存储的根目录位置。
`iceberg.table_identifier`	使用 namespace.table 格式指定表标识符。

HiveCatalog 模式

适用于 Iceberg 表的 catalog 为 hive 类型的 Iceberg 表。

示例

```
CREATE EXTERNAL TABLE ice_hive_cat STORED AS ICEBERG
```

```
TBLPROPERTIES ('iceberg.catalog'='hive.catalog',
  'hive.metastore.uris'='thrift://127.0.0.1:9083',
  'metastore.catalog.default'='hive',
  'fs.defaultFS'='cosn://cos_bucket',
  'iceberg.table_identifier'='namespace.table');
```

示例属性说明

属性	说明
`iceberg.catalog`	设置为 `'hive.catalog'`。
`hive.metastore.uris`	指定 Hive metastore 的 uris。
`metastore.catalog.default`	指定 Iceberg 表在 hive 中的 catalog，一般默认在'hive'中。
`fs.defaultFS`	指定 Iceberg 表数据所在的文件系统，目前只支持 COS，格式：`cosn://cos_bucket_name`
`iceberg.table_identifier`	使用 namespace.table 格式指定表标识符。

ALTER TABLE 语句

最近更新时间：2026-05-06 16:28:12

`ALTER TABLE` 语句用于更改现有 TCHouse-X 表的结构或属性。

核心概念

- 逻辑操作： 在 TCHouse-X 中， `ALTER TABLE` 主要是一个逻辑操作，用于更新元存储中存储的表元数据。
- 物理影响： 大多数 `ALTER TABLE` 操作不会重写或移动实际的数据文件。然而，某些操作（例如更改文件格式、添加/删除列导致的数据不一致）可能需要用户自行执行相应的物理文件系统操作（如使用 `INSERT OVERWRITE` 重写数据文件）以确保数据兼容性。

`ALTER TABLE` 语句更改现有 TCHouse-X 表的结构或属性。
在 TCHouse-X 中，这主要是一个逻辑操作，用于更新 TCHouse-X 中存储的表元数据。大多数 `ALTER TABLE` 操作并不会真正重写或移动实际数据文件。执行 `ALTER TABLE` 操作时，通常还需要执行相应的物理文件系统操作，例如重写数据文件以包含额外字段，或将其转换为不同的文件格式。

> **注意：**
> Iceberg 外表不支持 任何ALTER TABLE 操作。

修改表属性

语法

```
-- 1. 重命名或移动表
ALTER TABLE [old_db_name.]old_table_name RENAME TO
[new_db_name.]new_table_name;

-- 2. 更改外部表的存储属性 （仅限外部表）
ALTER TABLE name
  SET { FILEFORMAT file_format
  | ROW FORMAT row_format
  | LOCATION 'table_path_of_directory' };

-- 3. 设置表属性
ALTER TABLE name SET TBLPROPERTIES ('key'='val', ...);
```

语法说明

子句	描述	TCHouse-X 内表是否支持
RENAME TO	更改表名，或将表移动到另一个数据库。	不支持
SET FILEFORMAT	更改底层数据文件的格式（如 PARQUET, TEXTFILE）。	不支持
SET ROW FORMAT	更改行数据的格式（如 DELIMITED 分隔符）。	不支持
SET LOCATION	更改外部表数据文件所在的目录。	不支持
SET TBLPROPERTIES	用于设置表自定义属性。	支持

修改列

`ALTER TABLE` 提供了多种修改表列定义的方法，主要影响元数据。

语法

```
-- 添加新列
ALTER TABLE name ADD [IF NOT EXISTS] COLUMNS (col_spec[, col_spec ...]);
ALTER TABLE name ADD COLUMN [IF NOT EXISTS] col_spec;

-- 替换所有现有列（注意：内部 TCHouse-X 表不支持此操作）
ALTER TABLE name REPLACE COLUMNS (col_spec[, col_spec ...]);

-- 修改单个列的名称、类型或注释
ALTER TABLE name CHANGE column_name col_spec;

-- 删除列
ALTER TABLE name DROP [COLUMN] column_name;

-- 修改列的注释
ALTER TABLE name ALTER [COLUMN] column_name SET COMMENT 'comment_text';

-- 设置列的统计信息
ALTER TABLE name SET COLUMN STATS col_name('statsKey'='val', ...);

-- 列规范（col_spec）的定义：
```

```
-- 列名 类型名称 [COMMENT '列注释']
col_spec ::= col_name type_name [COMMENT 'column-comment']

-- 列统计信息键（statsKey）的定义：
-- numDVs（去重值数量）| numNulls（空值数量）| avgSize（平均长度）| maxSize（最大长度）
statsKey ::= numDVs | numNulls | avgSize | maxSize
```

⚠ 注意：

- TCHouse-X 内表不支持 `REPLACE COLUMNS` 子句。
- 变更列类型支持情况如下：

类型分类	变更方向	结果	示例
整数类型	扩大存储范围	支持	`INT` → `BIGINT`
	缩小存储范围	不支持	`BIGINT` → `INT`
浮点类型	增加精度	支持	`FLOAT` → `DOUBLE`
	降低精度	不支持	`DOUBLE` → `FLOAT`
DECIMAL	扩大精度 (p)	支持 (需保持 s 不变)	`DECIMAL(9,0)` → `DECIMAL(18,0)`
	缩小精度 (p)	不支持	`DECIMAL(18,0)` → `DECIMAL(9,0)`

添加列

- 您可以一次添加多个列。
- 如果使用 `IF NOT EXISTS`，当列已存在时 TCHouse-X 会忽略请求，不会返回错误。
- 数据兼容性： 对于底层数据文件中不存在的新增列，所有旧数据行在该列的值都将被视为 `NULL` 。

示例

```
-- 准备工作：创建表并插入初始数据
CREATE TABLE t1 (x int);
INSERT INTO t1 VALUES (1), (2);
```

```
-- 添加列 s 和 t
ALTER TABLE t1 ADD COLUMNS (s string, t timestamp);
INSERT INTO t1 VALUES (3, 'three', '2014-11-02 01:30:00');

-- 再次添加列 b
ALTER TABLE t1 ADD COLUMNS (b boolean);
INSERT INTO t1 VALUES (4, 'four', '2014-11-02 01:03:00', true);

-- 查询结果显示：旧数据在新增列上的值为 NULL
SELECT * FROM t1 ORDER BY x;
/*
+---+-------+---------------------+------+
| x | s     | t                   | b    |
+---+-------+---------------------+------+
| 1 | NULL  | NULL                | NULL |
| 2 | NULL  | NULL                | NULL |
| 3 | three | 2014-11-02 01:30:00 | NULL |
| 4 | four  | 2014-11-02 01:03:00 | 1    |
+---+-------+---------------------+------+
*/
```

设置列统计信息

您可以使用不区分大小写的符号名称指定统计类型：`numDVs`（不同值数量）、`numNulls`（空值数量）、`avgSize`（平均大小）、`maxSize`（最大大小）。键名称和值都需要用引号括起来。

示例

```
-- 准备工作：创建表并插入初始数据
CREATE TABLE t1 (x int, s string);
INSERT INTO t1 VALUES (1, 'one'), (2, 'two'), (2, 'deux');

-- 查看初始统计信息
-- 注意：-1 通常表示尚未计算或缺失统计数据
SHOW COLUMN STATS t1;
/*
+--------+--------+------------------+--------+----------+----------+
| Column | Type   | #Distinct Values | #Nulls | Max Size | Avg Size |
+--------+--------+------------------+--------+----------+----------+
```

```
| x       | INT     | -1                | -1      | 4         | 4         |
| s       | STRING  | -1                | -1      | -1        | -1        |
+---------+---------+-------------------+---------+-----------+-----------+
*/

-- 手动设置列统计信息
-- numDVs: 去重后的值数量, numNulls: 空值数量
ALTER TABLE t1 SET COLUMN STATS x ('numDVs'='2','numNulls'='0');
ALTER TABLE t1 SET COLUMN STATS s ('numdvs'='3','maxsize'='4');

-- 检查更新后的统计信息
SHOW COLUMN STATS t1;
/*
+---------+---------+-------------------+---------+-----------+-----------+
| Column  | Type    | #Distinct Values  | #Nulls  | Max Size  | Avg Size  |
+---------+---------+-------------------+---------+-----------+-----------+
| x       | INT     | 2                 | 0       | 4         | 4         |
| s       | STRING  | 3                 | -1      | 4         | -1        |
+---------+---------+-------------------+---------+-----------+-----------+
*/
```

修改分区

> **注意：**
>
> 1. 本节内容主要适用于 Hive 外部表。
> 2. 表必须是分区表（使用 PARTITIONED BY 子句创建）。分区是文件系统中的物理目录，其名称编码了特定的列值

语法

```
-- 添加分区
-- 可以指定分区列及其对应的值，可选是否指定存储路径 (location_spec)
ALTER TABLE name ADD [IF NOT EXISTS] PARTITION (partition_spec)
[location_spec];

-- 删除分区
-- 使用 PURGE 关键字可以跳过回收站，直接从文件系统中物理删除数据
ALTER TABLE name DROP [IF EXISTS] PARTITION (partition_spec) [PURGE];
```

```
-- 修改分区属性
-- 可以单独为某个分区设置文件格式、行格式或存储目录路径
ALTER TABLE name PARTITION (partition_spec)
  SET { FILEFORMAT file_format
  | ROW FORMAT row_format
  | LOCATION 'table_path_of_directory' };

-- 分区规范 (partition_spec) 的定义：
-- 格式为：分区列名 = 常量值
partition_spec ::= partition_col=constant_value
```

添加分区

TCHouse-X 的 `INSERT` 语句会在必要时自动创建分区，因此 `ALTER TABLE ... ADD PARTITION` 主要用于数据导入场景。
目的： 手动在元存储中注册一个新的分区元数据，通常指向一个已经包含数据文件的文件系统目录。
数据导入方式：

- 使用 `LOAD DATA` 语句将文件移动到新创建的分区目录中。
- 使用 `ALTER TABLE ... PARTITION (...) SET LOCATION` 将分区元数据指向一个已经包含数据的目录。

删除分区

`DROP PARTITION` 子句用于删除某组分区键值对应的文件系统目录及其关联的数据文件。
使用场景：

- **数据治理：** 删除不再需要的旧分区（例如，只分析最近三个月的数据）。
- **性能提升：** 减少与表关联的元数据量和查询计划的复杂性，从而简化并加速对分区表的执行。

使用 IF [NOT] EXISTS 确保 SQL 成功执行

为了在添加或删除分区时避免因分区已存在或不存在而抛出错误，可以使用可选的 `IF [NOT] EXISTS` 或 `IF EXISTS` 子句，以确保语句能够成功执行。

示例

```
-- 准备工作：创建分区表并创建 1990 和 2000 分区
CREATE EXTERNAL TABLE partition_t (s string) PARTITIONED BY (y int)
STORED AS TEXTFILE LOCATION 'cosn://your_path';
ALTER TABLE partition_t ADD PARTITION (y=1990);
ALTER TABLE partition_t ADD PARTITION (y=2000);
```

```
-- 尝试添加已存在的分区（将失败）
ALTER TABLE partition_t ADD PARTITION (y=2000);
-- ERROR: AnalysisException: Partition spec already exists: (y=2000).

-- 使用 IF NOT EXISTS 成功执行（即使 2000 已存在，2010 是新创建）
ALTER TABLE partition_t ADD IF NOT EXISTS PARTITION (y=2000);
ALTER TABLE partition_t ADD IF NOT EXISTS PARTITION (y=2010);

-- 使用 IF EXISTS 成功删除（即使 1950 不存在，2000 被删除）
ALTER TABLE partition_t DROP IF EXISTS PARTITION (y=2000);
ALTER TABLE partition_t DROP IF EXISTS PARTITION (y=1950);
```

多分区批量操作

对于 `DROP` 或 `SET` 操作， `PARTITION` 子句的表达式可以包含比较运算符（如 `<`，`IN`，`BETWEEN`）和布尔运算符（如 `AND`，`OR`），允许一次性针对多个分区执行操作。

- 分区键值可以是常量表达式： 可以使用不引用任何列的任意常量表达式来指定分区键的值。

示例

```
-- 示例：使用常量表达式添加分区
ALTER TABLE time_data ADD PARTITION (month=concat('Decem','ber'));

-- 示例：删除过期数据
ALTER TABLE historical_data DROP PARTITION (year < 1995);

-- 示例：复合条件删除分区（逗号分隔等同于 AND）
ALTER TABLE historical_data DROP PARTITION (year < 1995, last_name like
'A%');

-- 示例：批量更改文件格式
ALTER TABLE fast_growing_data PARTITION (year = 2016, month in
(10,11,12))
  SET FILEFORMAT parquet;
```

说明：

- 对多个分区生效的 DDL 语句即使没有分区匹配条件也视为成功（不产生任何更改），其结果等同于指定了 `IF EXISTS` 子句。
- 此方法的性能类似于快速连续执行一系列单分区 `ALTER TABLE` 语句。

自动检测新分区目录

> ⚠ **注意：**
>
> 1. 本节内容主要适用于 Hive 外部表。
> 2. 表必须是分区表（使用 PARTITIONED BY 子句创建）。分区是文件系统中的物理目录，其名称编码了特定的列值

当新的分区目录在 TCHouse-X 外部被创建和添加数据（例如通过 Hive `ALTER TABLE` 语句或直接的 HDFS/COS 命令操作）时，`RECOVER PARTITIONS` 子句会扫描表的数据目录，自动检测这些新添加的分区目录及其中的数据文件，并将相应的元数据注册到 TCHouse-X。

语法

```
ALTER TABLE name RECOVER PARTITIONS
```

示例

```
-- 准备工作（创建外部表并创建一个分区）
CREATE EXTERNAL TABLE t1 (s string) PARTITIONED BY (year int, month int)
STORED AS TEXTFILE LOCATION 'cosn://path';
ALTER TABLE t1 ADD PARTITION (year = 2016, month = 1);

-- t1 看到一个分区
SHOW PARTITIONS t1

-- 删除分区
ALTER TABLE t1 DROP PARTITION (year = 2016, month = 1);

-- t1 下没有分区
SHOW PARTITIONS t1

-- 执行 RECOVER PARTITIONS 扫描 COS 目录并注册新的分区元数据
ALTER TABLE t1 RECOVER PARTITIONS;

-- t1 看到一个分区
SHOW PARTITIONS t1
```

SYNC DATA

> ⚠ **注意：**
> 该操作仅 TCHouse-X 内表支持。

```
ALTER TABLE name EXECUTE SYNC_DATA();
```

- TCHouse-X 内表在数据写入后，会先写 WAL 日志，WAL日志持久化成功后，会立即返回成功。TCHouse-X 存储引擎后台会把 WAL 日志转换成内表存储格式。在 WAL 转成内表存储格式之前，这次写入的数据是无法查询的。
- 可以使用 SYNC DATA 语句强制存储引擎把 WAL 日志数据转换成内表存储格式，这样就可以查询到新写入的数据。
- 目前 SYNC DATA 默认超时是60s，如果一次写入的数据非常多，可能会出现超时的情况，重试即可。

DROP TABLE 语句

最近更新时间：2026-05-06 16:28:12

`DROP TABLE` 语句用于从 TCHouse-X 系统中删除一个表及其元数据。

核心影响

DROP TABLE 操作对底层数据文件的影响取决于表的类型：

表类型	数据文件	元数据 (Metastore)
内表	删除	删除
外表	保留	删除

> **注意：**
> 外部表适用于数据由其他 Hadoop/数据湖组件控制的情况。TCHouse-X 仅用于查询这些数据文件，`DROP TABLE` 不会影响数据的原始存储。

语法

```
DROP TABLE [IF EXISTS] [db_name.]table_name
```

语法说明

`IF EXISTS` 子句：可选的 `IF EXISTS` 子句使得该语句执行成功。

- 如果表存在： 正常删除该表。
- 如果表不存在： 语句成功执行，但不做任何操作，也不会返回错误。

> **说明：**
> 此功能在自动化脚本（如设置脚本）中非常有用。通过结合使用 DROP TABLE IF EXISTS 和 CREATE TABLE IF NOT EXISTS，可以确保脚本在第一次运行时和后续重复运行时都能成功执行。

> **注意：**
> 在删除表之前，请确保您位于正确的数据库中（通过 `USE db_name` 语句），或者使用完全限定名 `db_name.table_name` 来指定要删除的表。

示例

```
-- 准备工作
CREATE DATABASE temporary;
USE temporary;
CREATE TABLE unimportant (x int);
CREATE TABLE trivial (s string);

-- 1. 删除当前数据库中的表
DROP TABLE unimportant;

-- 2. 切换到另一个数据库
USE default;

-- 3. 尝试删除其他数据库中的表（将失败，因为未指定数据库名）
DROP TABLE trivial;
-- 错误示例: ERROR: AnalysisException: Table does not exist:
default.trivial

-- 4. 使用完全限定名成功删除
DROP TABLE temporary.trivial;
```

强制删除表

在某些情况下（例如元数据不一致或损坏），标准的 DROP TABLE 语句可能会失败。在这种情况下，可以通过设置会话变量来强制执行删除操作。

用法

```
-- 启用强制删除模式
SET force_drop_table = true;

-- 执行删除操作
DROP TABLE test_corrupted;
```

COMPUTE STATS 语句

最近更新时间：2026-05-06 16:28:12

`COMPUTE STATS` 语句用于收集表及其所有关联列和分区中关于数据量和分布的信息。这些统计信息存储在元数据中，TCHouse-X 的查询优化器依靠这些统计信息来准确判断表的大小、不同值的数量（基数）、数据倾斜程度等，从而为 连接（`JOIN`）、聚合（`Aggregation`）或 插入（`INSERT`）等操作生成更高效的执行计划。

完整统计信息

语法

```
COMPUTE STATS [db_name.]table_name  [ ( column_list ) ]
column_list ::= column_name [ , column_name, ... ]
```

使用说明

- 完整扫描 (默认)：如果未提供 `column_list`，`COMPUTE STATS` 会为表中的所有列计算统计信息。
- 指定列：可通过可选的逗号分隔列列表指定要计算统计信息的列。这适用于宽表或大型字符串列，可避免不必要的开销。
- 错误处理：当指定的列不存在或列类型（如复杂类型）不受支持时，语句将报错。如果提供空列表，则不分析任何列。

性能考量

`COMPUTE STATS` 是 ETL 过程结束时的一个关键步骤。准确的统计信息是生成高效查询计划、提升性能和降低内存使用的基础。

- 优化连接： 准确的统计信息能帮助 TCHouse-X 为连接查询构建高效的执行计划。
- 资源管理： 统计信息有助于 TCHouse-X 估算每个查询所需的内存，这对于使用准入控制 (Admission Control) 尤其重要。
- 性能调优： 对于大表，`COMPUTE STATS` 语句本身可能耗时较长。
- 直方图： `COMPUTE STATS` 还可以收集列的直方图信息，但出于性能考虑，该功能默认关闭。

示例

这个示例展示了两张表 `T1` 和 `T2`，它们通过 T1.ID 与 T2.PARENT 之间的父子关系关联，且具有较少的不同值。`T1` 的数据量很小，而 `T2` 大约有 100 000 行。最初，统计信息仅包含物理度量，例如文件数、总大小，以及针对固定长度列（如 `INT` 类型）的大小度量。未知值用 -1 表示。对每张表运行 `COMPUTE STATS` 之后，通过 `SHOW STATS` 语句可以获取更多详细信息。如果要对这两张表进行连接查询，就需要同时拥有它们的统计信息，才能获得最有效的查询优化。

```
-- 查看表 t1 的统计信息
show table stats t1;
-- 结果
-- +-------+--------+------+--------+
-- | #Rows | #Files | Size | Format |
-- +-------+--------+------+--------+
-- | -1    | 1      | 33B  | TEXT   |
-- +-------+--------+------+--------+

-- 查看表 t2 的统计信息
show table stats t2;
-- 结果
-- +-------+--------+----------+--------+
-- | #Rows | #Files | Size     | Format |
-- +-------+--------+----------+--------+
-- | -1    | 28     | 960.00KB | TEXT   |
-- +-------+--------+----------+--------+

-- 查看表 t1 的列统计信息
show column stats t1;
-- 结果
-- +--------+--------+------------------+--------+----------+----------+
-- | Column | Type   | #Distinct Values | #Nulls | Max Size | Avg Size |
-- +--------+--------+------------------+--------+----------+----------+
-- | id     | INT    | -1               | -1     | 4        | 4        |
-- | s      | STRING | -1               | -1     | -1       | -1       |
-- +--------+--------+------------------+--------+----------+----------+

-- 查看表 t2 的列统计信息
show column stats t2;
-- +--------+--------+------------------+--------+----------+----------+
-- | Column | Type   | #Distinct Values | #Nulls | Max Size | Avg Size |
-- +--------+--------+------------------+--------+----------+----------+
-- | parent | INT    | -1               | -1     | 4        | 4        |
-- | s      | STRING | -1               | -1     | -1       | -1       |
-- +--------+--------+------------------+--------+----------+----------+

-- 对 t1 执行统计信息收集
compute stats t1;
```

```
-- 结果
-- +----------------------------------------+
-- | summary                                |
-- +----------------------------------------+
-- | Updated 1 partition(s) and 2 column(s). |
-- +----------------------------------------+

-- 查看表 t1 的统计信息
show table stats t1;
-- 结果
-- +-------+--------+------+--------+
-- | #Rows | #Files | Size | Format |
-- +-------+--------+------+--------+
-- | 3     | 1      | 33B  | TEXT   |
-- +-------+--------+------+--------+

-- 查看表 t1 的列统计信息
show column stats t1;
-- 结果
-- +--------+--------+------------------+--------+----------+----------+
-- | Column | Type   | #Distinct Values | #Nulls | Max Size | Avg Size |
-- +--------+--------+------------------+--------+----------+----------+
-- | id     | INT    | 3                | -1     | 4        | 4        |
-- | s      | STRING | 3                | -1     | -1       | -1       |
-- +--------+--------+------------------+--------+----------+----------+

-- 对 t2 执行统计信息收集
compute stats t2;
-- 结果
-- +----------------------------------------+
-- | summary                                |
-- +----------------------------------------+
-- | Updated 1 partition(s) and 2 column(s). |
-- +----------------------------------------+

-- 查看表 t2 的统计信息
show table stats t2;
-- 结果
-- +-------+--------+----------+--------+
-- | #Rows | #Files | Size     | Format |
```

```
-- +-------+--------+----------+--------+
-- | 98304 | 1      | 960.00KB | TEXT   |
-- +-------+--------+----------+--------+

-- 查看表 t2 的列统计信息
show column stats t2;
-- 结果
-- +--------+--------+------------------+--------+----------+----------+
-- | Column | Type   | #Distinct Values | #Nulls | Max Size | Avg Size |
-- +--------+--------+------------------+--------+----------+----------+
-- | parent | INT    | 3                | -1     | 4        | 4        |
-- | s      | STRING | 6                | -1     | 14       | 9.3      |
-- +--------+--------+------------------+--------+----------+----------+
```

DROP STATS 语句

最近更新时间：2026-05-06 16:28:12

`DROP STATS` 用于从表或分区中删除由 COMPUTE STATS 语句生成的统计信息。

> **注意：**
> 删除统计信息后，查询优化器将无法基于准确的统计数据进行执行计划优化，可能导致查询性能下降。建议仅在必要时执行此操作。

语法

```
DROP STATS [database_name.]table_name
```

参数说明

参数	说明
`database_name`	（可选）目标表所在的数据库名称。如未指定，则默认使用当前数据库。
`table_name`	要删除统计信息的表名。

使用示例

```
-- 1. 删除当前数据库中指定表的统计信息
DROP STATS my_table;

-- 2. 删除指定数据库中表的统计信息
DROP STATS my_db.my_table;
```

DML
DML 语法概览

最近更新时间：2026-05-06 16:28:12

DML (Data Manipulation Language) 是 SQL 语言的核心子集，专门用于对数据库表中的数据内容进行增、删、改等操作。
在 TCHouse-X 中，常用的 DML 语句及其功能如下表所示：

语句类型	功能描述	常见应用场景
INSERT 语句	向表中插入新行	手动录入数据或将查询结果导入目标表。
DELETE 语句	删除表中的特定行	根据指定条件（WHERE 子句）移除过时或错误的数据。
UPDATE 语句	修改表中的现有数据	更新一个或多个字段的值。
TRUNCATE 语句	清空表中的所有数据	快速移除表中全部行，通常比 DELETE 更高效，但不支持按条件删除。

DELETE 语句

最近更新时间：2026-05-06 16:28:12

`DELETE` 语句用于从 TCHouse-X 表中删除满足指定条件的行。

> ⚠ **注意：**
> **仅限主键表**：TCHouse-X 目前仅支持对定义了主键（Primary Key）的内表进行 `DELETE` 操作。非主键表不支持此操作。

语法

```
DELETE [FROM] [database_name.]table_name [ WHERE where_conditions ]
```

- 如果提供 `WHERE` 子句，则删除匹配条件的行。
- 如果省略 `WHERE` 子句，则删除表中的所有行（清空表内容）。

使用说明

- **谓词限制**：WHERE 子句中的条件仅在目标表为主键表时有效。
- **条件灵活性**：`WHERE` 子句右侧的表达式规则与 `SELECT` 语句相同，支持子查询。
- **结果反馈**：执行完成后，系统会在消息栏中报告受影响（被删除）的行数。
- **统计信息更新**：
  - 在执行大规模 `DELETE` 操作后，建议执行 `COMPUTE STATS` 语句。
  - **原因**：保持统计信息最新对优化器生成高效的连接（Join）查询执行计划至关重要，特别是对于大表。

示例

准备测试表

```
-- 创建主键表并插入数据
CREATE TABLE test_pk (id int PRIMARY KEY, content int);
INSERT INTO test_pk VALUES (1,1), (2,2), (3,3), (4,4);

-- 创建非主键表
CREATE TABLE test_nopk (id int, content int);
```

基于条件删除

```
-- 删除指定 ID 的行
DELETE FROM test_pk WHERE id = 1;

-- 使用子查询删除：删除 ID 大于“ID为2的行”的所有行
DELETE FROM test_pk WHERE id > (SELECT id FROM test_pk WHERE id = 2);

-- 基于非主键字段条件删除
DELETE FROM test_pk WHERE content = 2;
```

清空表数据

```
-- 删除主键表中的所有行
DELETE FROM test_pk;
```

错误示例

```
-- 尝试删除非主键表数据（将触发报错）
DELETE FROM test_nopk;
-- 报错信息: ERROR: AnalysisException: Tci NOT SUPPORT: Currently only
support delete with primary key
```

注意事项

- **内表限制**：仅 TCHouse-X 内表支持 `DELETE` 操作，外表（如 Hive 映射表）不支持。
- **性能建议**：若需清空整个大表，使用 `TRUNCATE TABLE` 通常比不带条件的 `DELETE` 更快。

UPDATE 语句

最近更新时间：2026-05-06 16:28:13

`UPDATE` 语句用于修改 TCHouse-X 表中现有行的列值。

核心约束

1. **仅限主键表**：目前仅支持对定义了主键（Primary Key）的内表进行 `UPDATE` 操作。
2. **主键不可更改**：`SET` 子句中不能包含主键列，即不支持直接更新主键的值。
3. **内表限制**：仅 TCHouse-X 内表支持此操作。

语法

```
UPDATE [database_name.]table_name SET col = val [, col = val ... ]
  [ FROM joined_table_refs ]
  [ WHERE where_conditions ]
```

使用说明

- **条件过滤**：`WHERE` 子句决定了更新的范围。若省略 `WHERE`，则表中的所有行都会被更新。
- **谓词规则**：`WHERE` 子句支持与 `SELECT` 语句相同的复杂条件，包括子查询。
- **FROM 子句**：通过 `FROM` 进行多表连接（Join），可根据关联表的匹配结果来限制待更新的行。
- **并发行为**：
  - `UPDATE` 可与其他 DML（`INSERT` / `UPSERT`）并发运行。
  - 由于并发执行时数据状态可能发生变化，最终受影响的行数可能与预期存在微小差异。
- 反馈信息：执行成功后，系统会报告受影响的行数（Modified rows）。
- 统计信息维护：
  - 执行大规模更新后，务必运行 `COMPUTE STATS`。这对于保持执行计划的准确性、优化后续的连接查询性能至关重要。

示例

数据准备

```
-- 创建主键表
CREATE TABLE test_pk (id int PRIMARY KEY, content int);
INSERT INTO test_pk VALUES (1,1), (2,2), (3,3), (4,4);
```

```
-- 创建非主键表
CREATE TABLE test_nopk (id int, content int);
INSERT INTO test_nopk VALUES (1,1), (2,2), (3,3), (4,4);
```

常规更新与限制

```
-- 【推荐】基于主键更新特定行
UPDATE test_pk SET content = 11 WHERE id = 1;

-- 【报错】非主键表不支持更新
-- ERROR: AnalysisException: Currently only support update with primary
key
UPDATE test_nopk SET content = 11 WHERE id = 1;

-- 【报错】禁止修改主键列的值
-- ERROR: AnalysisException: setting expr can't be primary key
UPDATE test_pk SET id = 100 WHERE id = 1;
```

复杂条件更新

```
-- 使用子查询作为 WHERE 谓词
UPDATE test_pk SET content = 34
WHERE id > (SELECT id FROM test_pk WHERE id = 2);

-- 全表更新（省略 WHERE 子句）
UPDATE test_pk SET content = 100;
```

带 FROM 子句的关联更新

通过连接（Join）其他表来确定更新范围：

```
UPDATE test_pk
SET test_pk.content = 33
FROM test_pk JOIN test_nopk ON test_pk.content > test_nopk.content
WHERE test_pk.id > (SELECT id FROM test_pk WHERE id = 2);
```

优化建议

- **数据安全**：在执行不带 `WHERE` 条件的 `UPDATE` 前，建议先用相同的条件执行 `SELECT`，核对受影响的数据范围。
- **性能优化**：对于超大规模的全表数据更新，有时重新加载数据（ `TRUNCATE` + `INSERT` ）可能比逐行 `UPDATE` 更具性能优势。

TRUNCATE 语句

最近更新时间：2026-05-06 16:28:13

`TRUNCATE TABLE` 用于高效地清空表中的所有数据，同时保留表的定义（Schema）和结构。

语法

```
TRUNCATE [TABLE] [IF EXISTS] [db_name.]table_name
```

使用说明

- **核心用途：**常用于 ETL 流程中清空中间表或临时表。相比于删除并重建表，或使用 `INSERT OVERWRITE`，`TRUNCATE` 是一种低开销、高性能的方法。
- **适用范围：**
  - 删除表中的所有数据及其相关的物理数据文件。
  - 支持内表、外表、分区表。
  - 操作适用于全表，会自动处理所有分区的数据。
- **副作用：**执行后，由 `COMPUTE STATS` 生成的所有表统计信息和列统计信息将被重置（清空）。
- **安全建议：**
  - 在执行前确保位于正确的数据库（使用 `USE` 切换），或使用全限定名（`db.table`）。
  - `IF EXISTS`：此子句可防止因表不存在而报错，非常适合放在自动化脚本或标准化 ETL 设置脚本中。
- **可选关键字：**`TABLE` 关键字是可选的，不影响执行逻辑。

示例

清空数据并重置统计信息

以下示例展示了 `TRUNCATE` 如何在清空数据的同时重置统计信息。

```
-- 1. 准备数据并收集统计信息
CREATE TABLE truncate_demo (x INT);
INSERT INTO truncate_demo VALUES (1), (2), (4), (8);

-- TCHouse-X 数据写入后可见性约为 10s，建议至少等待 10s 后再执行后续操作。
COMPUTE STATS truncate_demo;

-- 查看统计信息：此时有 4 行数据
SHOW TABLE STATS truncate_demo;
```

```
-- +-------+--------+------+--------+
-- | #Rows | #Files | Size | Format |
-- +-------+--------+------+--------+
-- | 4     | 1      | 420B | TCI    |
-- +-------+--------+------+--------+

-- 2. 执行清空操作
TRUNCATE TABLE truncate_demo;

-- 3. 验证结果：数据和统计信息均被清空
SELECT COUNT(*) FROM truncate_demo; -- 返回 0

SHOW TABLE STATS truncate_demo;
-- +--------+--------+------+--------+
-- | #Rows  | #Files | Size | Format |
-- +--------+--------+------+--------+
-- | -1     | 0      | 0B   | TCI    |
-- +--------+--------+------+--------+
```

在自动化脚本中使用 IF EXISTS

该子句允许脚本在表可能不存在的情况下继续运行而不中断。

```
-- 即使 staging_table2 和 3 不存在，脚本也不会报错
TRUNCATE TABLE IF EXISTS staging_table1;
TRUNCATE TABLE IF EXISTS staging_table2;
TRUNCATE TABLE IF EXISTS staging_table3;
```

TRUNCATE vs DELETE 快速对比

特性	TRUNCATE TABLE	DELETE
操作粒度	全表清空	可根据条件精确删除 (WHERE)
性能	极高（直接物理删除文件）	较低（逐行或按主键范围删除）
主键要求	无要求，支持各类表	仅支持主键表
统计信息	自动重置	保留统计信息（需手动重置）

INSERT 语句

最近更新时间：2026-05-06 16:28:13

`INSERT` 语句用于向 TCHouse-X 的内表或外表中添加、追加或替换数据。

语法

```
[WITH with_clause]
INSERT { INTO | OVERWRITE } [TABLE] table_name
  [(column_list)]
  [ PARTITION (partition_clause)]
{
    select_statement
  | VALUES (value [, value ...]) [, (value [, value ...]) ...]
}

partition_clause ::= col_name [= constant] [, col_name [= constant] ...]
```

核心子句说明：

- **INTO**：将数据追加到表中，保留现有数据。
- **OVERWRITE**：替换表中的现有数据。旧数据文件会被立即物理删除。
- **column_list（列排列）**：可选。指定插入的目标列子集及其顺序。未提及的列将自动填充为 NULL。

使用限制与特性

不同表类型（内表 vs 外表）在执行 `INSERT` 时有显著差异：

内表 (Internal Table)

- **不支持的操作**：不支持 OVERWRITE 语法，不支持 `PARTITION` 子句。
- **可见性延迟**：插入数据后不会立即生效。若需立即可见，需执行：`ALTER TABLE table_name EXECUTE SYNC_DATA()`；
- **主键去重**：若插入的主键值已存在，新数据将覆盖旧数据。
- **注意**：由于分布式查询结果顺序的不确定性，对主键表使用 `INSERT SELECT` 可能导致结果不唯一。

外表 (External Table)

- **环境设置**：必须关闭向量化引擎：`SET ENABLE_VECTORIZED_EXECUTOR = false;`。
- **支持情况**：支持 `OVERWRITE` 填充和 `PARTITION` 分区插入。
- **格式支持**：目前仅支持写入 `TEXT` 和 `PARQUET` 格式。不支持 `RCFile` 或 `SequenceFile`。

分区插入 (仅限外表)

静态分区插入

在 `PARTITION` 子句中明确指定分区键的常量值。

```
-- 关闭向量化引擎
SET ENABLE_VECTORIZED_EXECUTOR = false;

-- 此时 SELECT 只需要提供非分区列的值
INSERT INTO t_partitioned PARTITION (year=2024, month=1)
SELECT id, name FROM source_table;
```

动态分区插入

分区键在 `PARTITION` 子句中列举但不赋值，由 `SELECT` 或 `VALUES` 结果集的最后几列动态填充。

```
-- 关闭向量化引擎
SET ENABLE_VECTORIZED_EXECUTOR = false;

-- 分区键 x, y 的值将取自 VALUES 结果集的最后两个位置
INSERT INTO t1 PARTITION (x, y) VALUES (1, 2, 'c');
-- 等价于: w=1, x=2, y='c'
```

关键注意事项

- **数据类型匹配**：系统不会自动将大类型转换为小类型。例如，将 `DOUBLE` 函数结果存入 `FLOAT` 列时，必须手动执行 `CAST()`：`INSERT INTO t1 SELECT CAST(COS(angle) AS FLOAT) ...`
- **VALUES 性能**：`INSERT ... VALUES` 每次操作都会生成一个小文件且无法并行，**严禁用于大规模 ETL 导入。该语法仅适用于小型维度表或语法测试。**
- **排序忽略**：`INSERT ... SELECT` 中的 `ORDER BY` 子句会被优化器忽略，因为并行写入无法保证全局有序性。

示例

主键内表的数据覆盖

```
CREATE TABLE pk_table (id INT PRIMARY KEY, val STRING);
INSERT INTO pk_table VALUES (1, 'old');
```

```
-- 插入相同主键
INSERT INTO pk_table VALUES (1, 'new');
ALTER TABLE pk_table EXECUTE SYNC_DATA();

-- 结果 id=1 的值为 'new'
SELECT * FROM pk_table;
```

外表的分区列排列操作

```
-- 列排列：只插入 c1, c2 两列，跳过 c3
INSERT INTO target_table (c1, c2) SELECT x, y FROM source_table;

-- 动态分区：w取1, x取2, y取'c'
INSERT INTO t_ext_partitioned (w) PARTITION (x, y) VALUES (1, 2, 'c');
```

DQL
DQL 语法概览

最近更新时间：2026-05-06 16:28:13

DQL (Data Query Language) 是 SQL 语言中应用最广泛的子集，主要用于从数据库模式对象（如表、视图）中检索数据。DQL 的核心目标是根据用户指定的过滤、聚合及排序规则，返回结构化的结果集。

TCHouse-X DQL 语句分类

在 TCHouse-X 中，DQL 不仅包含对业务数据的查询，还包含对数据库元数据的探索。

语句类型	功能描述	典型应用场景
SELECT 语句	核心查询语句	从表中提取、过滤、聚合、连接以及排序数据。
SHOW 语句	元数据概览	查看数据库列表、表列表、分区信息、统计信息或查询计划。
DESCRIBE 语句	表结构描述	查看表的列定义、数据类型、注释以及主键等结构化细节。

DESCRIBE 语句

最近更新时间：2026-05-06 16:28:13

`DESCRIBE` 语句用于查看数据库、表、视图的元数据信息，包括列定义、分区列、表属性等。

> **说明：**
> `DESCRIBE` 可简写为 `DESC`，两者功能完全等价。

语法

```
-- 查看表结构（列名、类型、注释）
{DESCRIBE | DESC} [database_name.]table_name

-- 查看表的详细属性
{DESCRIBE | DESC} FORMATTED [database_name.]table_name

-- 查看数据库属性
{DESCRIBE | DESC} DATABASE [database_name]
```

语法说明

形式	说明
`DESCRIBE [database_name.]table_name`	查看基本列信息：列名、数据类型、注释 等。
`DESCRIBE FORMATTED [database_name.]table_name`	查看详细信息：列定义、分区信息、存储格式、表属性、SerDe 等。
`DESCRIBE DATABASE database_name`	查看数据库属性：注释、位置、路径等。

SELECT 语句
SELECT 语法概览

最近更新时间：2026-05-06 16:28:13

`SELECT` 语句用于执行查询，从一个或多个表中检索数据，并生成由行和列组成的结果集。
`INSERT` 语句通常也以 `SELECT` 语句结束，用于定义从一个表复制到另一个表的数据。

语法：

```
[WITH name AS (select_expression) [, ...] ]
SELECT
  [ALL | DISTINCT]
  [STRAIGHT_JOIN]
  expression [, expression ...]
FROM table_reference [, table_reference ...]
[[FULL | [LEFT | RIGHT] INNER | [LEFT | RIGHT] OUTER | [LEFT | RIGHT]
SEMI | [LEFT | RIGHT] ANTI | CROSS]
  JOIN table_reference
  [ON join_equality_clauses | USING (col1[, col2 ...]] ...
WHERE conditions
GROUP BY { column | expression [, ...] }
HAVING conditions
ORDER BY { column | expression [ASC | DESC] [NULLS FIRST | NULLS LAST]
[, ...] }
LIMIT expression [OFFSET expression]
[UNION [ALL] select_statement] ...]

table_reference := { table_name | (subquery) }
```

`SELECT` 查询支持：

- SQL标量数据类型： `BOOLEAN` 、 `TINYINT` 、 `SMALLINT` 、 `INT` 、 `BIGINT` 、 `DECIMAL` 、 `FLOAT` 、 `DOUBLE` 、 `TIMESTAMP` 、 `STRING` 、 `VARCHAR` 、 `CHAR` 。
- 可选的 `WITH` 子句，位于 `SELECT` 关键字之前，用于定义一个子查询，子查询的名称或列名可以在主查询中稍后引用。此子句允许你抽象化重复的子句，如聚合函数，这些子句在同一个查询中多次引用。
- 可以在 `FROM` 子句中使用子查询，也可以在 `WHERE` 子句中使用，例如使用 `IN()` 、 `EXISTS` 和 `NOT EXISTS` 操作符。
- 支持 `WHERE` 、 `GROUP BY` 、 `HAVING` 子句。

- 可以通过它们的序号引用 `SELECT` 列表项。支持在 `GROUP BY`、`HAVING` 和 `ORDER BY` 子句中使用序号。序号只能在顶层使用。例如，以下语句是允许的：

```
-- 有效的例子：
  SELECT int_col / 2, sum(x)
  FROM t
  GROUP BY 1;

-- 在这个查询中，GROUP BY 1表示按第一列（即int_col / 2）分组。
  SELECT int_col / 2
  FROM t
  ORDER BY 1;

  SELECT int_col / 2
  FROM t
  ORDER BY 1 + 2;
-- 以上语句也是有效的，但ORDER BY 1 + 2没有任何效果，因为1 + 2的结果是3，表示排序
依据是第三列，而不是按第一列排序。

  SELECT NOT bool_col
  FROM t
  GROUP BY 1
  HAVING not 1;
-- 这个查询在解析时会抛出错误，因为not 1是无效的表达式，not运算符不能直接作用于数字
1。
```

总结：允许在 `GROUP BY`、`ORDER BY` 等子句中使用数字作为列序号，但需要注意的是，子表达式中的数字不会被解释为列序号，并且某些运算（如not 1）会导致语法错误。

- 支持多种 `JOIN` 子句。左连接、右连接、半连接、全连接和外连接在所有TCHouse-X版本中都得到支持。`CROSS JOIN` 操作符也可用。在性能调优过程中，你可以通过在 `SELECT` 语句及任何 `DISTINCT` 或 `ALL` 关键字后面立即添加 `STRAIGHT_JOIN` 关键字，来覆盖TCHouse-X内部的连接子句重排序。

  有关连接查询的详细信息和示例，请参见TCHouse-X `SELECT` 语句中的连接。
- `UNION ALL`。
- `LIMIT`。
- 外部表。
- 关系运算符，如大于、小于或等于。
- 算术运算符，如加法或减法。
- 逻辑/布尔运算符 `AND`、`OR` 和 `NOT`。TCHouse-X不支持相应的符号&&、||和!。

- 常见的SQL内置函数，如 `COUNT` 、 `SUM` 、 `CAST` 、 `LIKE` 、 `IN` 、 `BETWEEN` 和 `COALESCE` 。TCHouse-X特别支持在TCHouse-X内置函数中描述的内置函数。

DISTINCT 操作符

最近更新时间：2026-05-06 16:28:13

`DISTINCT` 用于在 `SELECT` 查询中过滤重复数据，仅保留唯一的行或值组合。

核心用法

单列去重

检索某一列中所有不重复的值。

- **NULL 值处理**：如果列中包含 `NULL`，`DISTINCT` 会将其作为一个独立的值返回。

```
-- 返回客户所属的所有不重复国家
SELECT DISTINCT c_birth_country FROM customer;
```

多列组合去重

检索多列字段的**唯一组合**。只有当所有指定列的值都完全相同时，才会被视为重复行。

```
-- 返回称谓与姓氏的唯一组合
SELECT DISTINCT c_salutation, c_last_name FROM customer;
```

与聚合函数结合 (COUNT DISTINCT)

常用于统计不重复项的数量。

- **统计单列唯一值**：`COUNT(DISTINCT col)` 会忽略 `NULL` 值，仅计算非空唯一值的个数。

```
SELECT COUNT(DISTINCT c_birth_country) FROM customer;
```

- **统计多列组合唯一值**：

```
SELECT COUNT(DISTINCT c_salutation, c_last_name) FROM customer;
```

数据区分度说明

在 TCHouse-X 中，`DISTINCT` 和 `GROUP BY` 对字符的敏感度极高，以下三者会被视为完全不同的独立值：

- **零长度字符串 (`""`)**
- **NULL 值 (`NULL`)**
- **空格字符串 (" ")**

注意事项与性能建议

- **无序性预警：**TCHouse-X 不会自动对 `DISTINCT` 后的结果进行排序。如果业务需要有序输出，必须显式添加 `ORDER BY` 子句。
- **性能优化：**在某些场景下，如果子查询中使用了 `DISTINCT`，可以考虑通过 `JOIN` 或 `EXISTS` 改写，以获得更优的执行计划。

GROUP BY 子句

最近更新时间：2026-05-06 16:28:13

`GROUP BY` 子句用于将结果集划分为多个逻辑组，并对每个组执行聚合计算（如 `COUNT()`、`SUM()`、`AVG()`、`MIN()` 和 `MAX()`）。

核心规则

- **非聚合列强制分组：**在 `SELECT` 列表中出现的、未参与聚合计算的所有列名，必须全部列在 `GROUP BY` 子句中。
- **空值处理：**TCHouse-X 对数据的区分度极高。在分组逻辑中，零长度字符串（`""`）、`NULL` 和 空格（`" "`）被视为三个完全不同的独立值。

使用数字序号 GROUP BY

TCHouse-X 支持通过正整数来引用 `SELECT` 列表中的列位置，从而简化代码。

- **对应关系：**正整数表示 SELECT 输出列表中的列序号（从 1 开始），而非原始物理表的列序号。这与 MySQL、PostgreSQL 和 Doris 的行为保持一致。
- **截断规则：**如果使用小数（如 `1.1`），系统会将其截断为整数（`1`）处理。
- **不支持负数：**使用 `-1` 等负数会导致错误。

示例

```
-- 以下两者等价
SELECT category, COUNT(*) FROM t1 GROUP BY category;
SELECT category, COUNT(*) FROM t1 GROUP BY 1;

-- 小数会被截断为整数
SELECT category, COUNT(*) AS cnt FROM t1 GROUP BY 1.1;    -- 等价于 GROUP
BY 1

-- 负数不支持
SELECT category, COUNT(*) FROM t1 GROUP BY -1;
-- Error: Expression 'CATEGORY' is not being grouped
```

浮点数分组的风险提示

在处理科学或金融数据时，需特别注意 `FLOAT` 或 `DOUBLE` 类型的列。

- **精度不确定性：**由于硬件浮点数的存储特性，它们无法精确表示所有十进制分数（例如 `96.94` 在底层可能存储为 `96.940002441...`）。

- **分组结果偏差：**当对浮点数列进行 `GROUP BY` 时，查询结果可能会因为这种微小的精度偏差产生“看似重复但值略有不同”的行。

解决方案：

1. 使用 `ROUND()` 函数对列进行四舍五入后再分组。
2. 使用 `DECIMAL` 类型存储货币或需要高精度的金融数据，以避免舍入误差。

综合案例

基础聚合与排序

查找销售总数量前 5 的商品及其销售次数：

```
SELECT
  ss_item_sk AS Item,
  COUNT(ss_item_sk) AS Times_Purchased,
  SUM(ss_quantity) AS Total_Quantity
FROM store_sales
GROUP BY ss_item_sk
ORDER BY Total_Quantity DESC
LIMIT 5;
```

配合 HAVING 进行结果过滤

`WHERE` 无法过滤聚合后的结果，必须使用 `HAVING` 。以下查询查找销售次数 >= 100 且销售总量最低的 5 个商品：

```
SELECT
  ss_item_sk AS Item,
  COUNT(ss_item_sk) AS Times_Purchased,
  SUM(ss_quantity) AS Total_Quantity
FROM store_sales
GROUP BY ss_item_sk
HAVING Times_Purchased >= 100
ORDER BY Total_Quantity ASC
LIMIT 5;
```

常见问题 (FAQ)

可以在 `GROUP BY` 中使用别名吗?

是的，TCHouse-X 支持在 `GROUP BY` 中引用 `SELECT` 列表中定义的别名。

为什么我的分组结果比预期多?

请检查数据中是否存在不可见的空格或空值。如前文所述，TCHouse-X 会将它们视为不同的组。

HAVING 子句

最近更新时间：2026-05-06 16:28:13

`HAVING` 子句用于对 `SELECT` 查询的汇总结果进行过滤。与 `WHERE` 子句在分组前过滤单条记录不同，`HAVING` 是在数据分组及聚合计算完成后，对“组”级别的数据进行筛选。

核心特性

- **聚合依赖：**主要用于检查聚合函数（如 `COUNT()`、`SUM()`、`AVG()`、`MIN()`、`MAX()`）的结果。
- **分组配合：**通常与 `GROUP BY` 子句结合使用。
- **支持子查询：**`HAVING` 条件中可以嵌入标量子查询，实现动态过滤。

HAVING 与 WHERE 对比

特性	WHERE 子句	HAVING 子句
执行阶段	在分组前（`GROUP BY` 之前）执行	在分组后（`GROUP BY` 之后）执行
作用对象	原始表中的每一行记录	聚合后的每一个分组（Group）
支持函数	不支持聚合函数	专门处理聚合函数

代码示例

基础用法：筛选特定规模的分组

统计各类别下的记录数，并仅保留记录数超过 1 条的类别：

```
SELECT category, COUNT(*) AS cnt
FROM t1
GROUP BY category
HAVING cnt > 1;
```

结合标量子查询

`HAVING` 子句可以根据其他查询返回的单一数值进行过滤。

示例 1：静态标量子查询对比

```
-- 筛选记录数大于子查询指定值（此处为 1）的分组
SELECT category, COUNT(*) AS cnt
FROM t1
```

```
GROUP BY category
HAVING COUNT(*) > (SELECT 1);
```

示例 2：使用聚合函数子查询实现动态过滤

```
-- 查找类别编号（或数值）大于全局平均类别的分组
SELECT category, COUNT(*) AS cnt
FROM t1
GROUP BY category
HAVING category > (SELECT avg(category) from t1);
```

结合标量子查询

```
-- HAVING 子句中使用标量子查询
SELECT category, COUNT(*) AS cnt
FROM t1
GROUP BY category
HAVING COUNT(*) > (SELECT 1);

-- 使用聚合函数作为子查询
SELECT category, COUNT(*) AS cnt
FROM t1
GROUP BY category
HAVING category > (SELECT avg(category) from t1);
```

优化建议与注意事项

- **性能原则：** 如果过滤条件不涉及聚合函数（例如 `WHERE status = 'Active'`），请务必将其写在 `WHERE` 子句中。这样可以在分组计算前排除无关数据，显著提升查询效率。
- **别名引用：** `HAVING` 可以直接引用 `SELECT` 列表中的列别名（如示例中的 `cnt`），这增加了代码的可读性。

JOIN 子句

最近更新时间：2026-05-06 16:28:13

`JOIN` 子句是 `SELECT` 语句的核心，用于将两个或多个表的数据基于相关标识符（如外键）组合在一起，实现跨表数据的交叉引用。

语法概览与连接类型

TCHouse-X 支持多种标准的 SQL 连接方式：

- **基本连接：** `INNER JOIN`（默认），`LEFT [OUTER] JOIN`，`RIGHT [OUTER] JOIN`，`FULL [OUTER] JOIN`。
- **高级连接：**
  - **SEMI JOIN：** 半连接（左/右），仅返回在一侧表中存在匹配的行，且每行仅返回一次。
  - **ANTI JOIN：** 反连接，返回在一侧表中找不到匹配项的行。
- **笛卡尔积：** `CROSS JOIN` 或不带 `ON` 条件的 `JOIN`。

```
SELECT select_list FROM
  table_or_subquery1 [INNER] JOIN table_or_subquery2 |
  table_or_subquery1 {LEFT [OUTER] | RIGHT [OUTER] | FULL [OUTER]} JOIN
table_or_subquery2 |
  table_or_subquery1 {LEFT | RIGHT} SEMI JOIN table_or_subquery2 |
  table_or_subquery1 {LEFT | RIGHT} ANTI JOIN table_or_subquery2 |
    [ ON col1 = col2 [AND col3 = col4 ...] |
      USING (col1 [, col2 ...]) ]
  [other_join_clause ...]
[ WHERE where_clauses ]

SELECT select_list FROM
  table_or_subquery1, table_or_subquery2 [, table_or_subquery3 ...]
  [other_join_clause ...]
WHERE
    col1 = col2 [AND col3 = col4 ...]

SELECT select_list FROM
  table_or_subquery1 CROSS JOIN table_or_subquery2
  [other_join_clause ...]
[ WHERE where_clauses ]
```

连接风格：SQL-92 vs. SQL-89

SQL-92 (推荐)

使用显式的 `JOIN` 关键字，逻辑清晰，易于维护。

- **ON 子句：**用于不同名列的比较。
- **USING 子句：**当两表连接列名相同时的简写（如 USING (id)）。

```
-- 使用 ON
SELECT t1.c1, t2.c2 FROM t1 JOIN t2 ON t1.id = t2.id;
-- 使用 USING
SELECT t1.c1, t2.c2 FROM t1 JOIN t2 USING (id);
```

SQL-89 (传统)

使用逗号分隔表名，连接条件置于 `WHERE` 中。

- **风险：**若误删 `WHERE` 中的连接条件，会意外产生巨大的笛卡尔积。
- **限制：**TCHouse-X 的外连接（OUTER JOIN）必须使用 SQL-92 语法，不支持 `(+)` 或 `*=`。

通用连接场景

在 TCHouse-X 中，通用连接场景主要分为内连接（ `INNER JOIN` ）和外连接（ `OUTER JOIN` ），它们决定了如何处理两表中不匹配的数据。

内连接 (Inner Join)

内连接是最常用的连接类型。它仅返回两个表中连接列值完全匹配的行组合。

- **默认行为：**TCHouse-X 默认执行内连接。
- **命名冲突：**若两表存在同名列，必须使用表前缀（如 t1.id）或别名来消除歧义。
- **语法等价性：**以下三种写法在 TCHouse-X 中效果相同。

```
-- 1. SQL-89 风格（隐含连接）
SELECT t1.id, c1, c2 FROM t1, t2 WHERE t1.id = t2.id;

-- 2. SQL-92 风格（显式连接）
SELECT t1.id, c1, c2 FROM t1 JOIN t2 ON t1.id = t2.id;

-- 3. 显式内连接（语义最清晰）
SELECT t1.id, c1, c2 FROM t1 INNER JOIN t2 ON t1.id = t2.id;
```

外连接 (Outer Join)

外连接用于保留不满足连接条件的行。当一侧表没有匹配数据时，结果集中对应的另一侧列将填充为 `NULL`。

- **左外连接 (LEFT OUTER JOIN)**：返回左表所有行。右表无匹配则补 `NULL`。
- **右外连接 (RIGHT OUTER JOIN)**：返回右表所有行。左表无匹配则补 `NULL`。
- **全外连接 (FULL OUTER JOIN)**：返回左、右表中的所有行。任何一侧缺失匹配均补 `NULL`。

```
-- 左外连接
SELECT * FROM t1 LEFT OUTER JOIN t2 ON t1.id = t2.id;

-- 右外连接
SELECT * FROM t1 RIGHT OUTER JOIN t2 ON t1.id = t2.id;

-- 全外连接
SELECT * FROM t1 FULL OUTER JOIN t2 ON t1.id = t2.id;
```

警告：

- 强制风格：执行外连接时，TCHouse-X 必须使用 SQL-92 语法（即 `JOIN` 关键字）。不支持通过逗号分隔表名的方式实现外连接。
- 不支持扩展符号：TCHouse-X 不支持某些传统数据库（如 Oracle 或 SQL Server 老版本）中的厂商专用扩展符，如 `(+)` 或 `*=`。

特殊连接场景

自连接 (Self-joins)

用于处理树形结构（如父子关系）。通过为同一张表分配不同的别名实现。

```
SELECT lhs.id AS child, rhs.id AS parent
FROM tree_data lhs, tree_data rhs
WHERE lhs.parent_id = rhs.id;
```

等值与非等值连接

- **等值连接 (Equijoins)**：使用 `=` 比较。这是最高效的连接方式，支持哈希连接（Hash Join）。
- **非等值连接 (Non-Equijoins)**：使用 `!=`，`<`，`>` 等。

警告：

非等值连接通常通过 `CROSS JOIN` 配合 `WHERE` 实现，或在 `OUTER JOIN` 中使用。由于涉及嵌套循环连接（Nested Loop Join），可能非常耗费内存且不支持溢写磁盘。

半连接 (Semi-joins)

半连接是一种高效的关联变体，主要用于判断“存在性”。它与普通内连接（ `Inner Join` ）最大的区别在于：**即使右表中有多个匹配项，左表的每一行也只会被返回一次。**这有效地避免了结果集的冗余膨胀。

核心特性

- **左半连接 (LEFT SEMI JOIN)：**仅当右表中存在匹配数据时，才返回左表的记录。
- **右半连接 (RIGHT SEMI JOIN)：**反转比较逻辑，仅返回右表中有匹配项的记录（TCHouse-X 2.2 及更高版本支持）。
- **去重机制：**半连接在逻辑上等价于 EXISTS 子句。它不会合并右表的列到结果集中，只返回原始表的数据。

示例：左半连接 (LEFT SEMI JOIN)

以下查询将检索出所有“在表 `t2` 中拥有匹配 ID”的 `t1` 记录：

```
-- 仅返回 t1 的列，且 t1 的每一行在结果集中最多出现一次
SELECT t1.c1, t1.c2
FROM t1 LEFT SEMI JOIN t2
ON t1.id = t2.id;
```

半连接与内连接对比：

特性	内连接 (INNER JOIN)	半连接 (SEMI JOIN)
结果集大小	可能因为右表的多条匹配产生笛卡尔积	绝不产生重复行，保持左表原有的行数或更少
返回列	包含左右两表的所有请求列	仅包含单侧表的列
典型场景	获取两个表的组合信息	检查数据是否存在（类似 `IN` 或 `EXISTS` ）

自然连接 (Natural Joins)

自然连接是一种特殊的等值连接，它通过隐式匹配两个表中所有名称相同的列来建立关联。

工作原理

- **自动匹配：**自然连接会自动寻找左表和右表中命名的所有列，并执行等值比较，无需手动编写 ON 或 USING 子句。
- **列去重：**在结果集中，同名的连接列仅会保留一份（虽然在 SELECT * 时表现明显，但在指定列名查询时需注意别名使用）。

语法样例

```
-- 假设 t1 和 t2 都有名为 'id' 和 'type' 的列
-- 系统会自动执行 ON t1.id = t2.id AND t1.type = t2.type
SELECT t1.c1, t2.c2
```

```
FROM t1 NATURAL JOIN t2;
```

> **警告：**
> 尽管 `NATURAL JOIN` 简化了代码，但在生产环境中需谨慎使用：
> - **结构依赖性强：**查询结果高度依赖于表结构。如果未来在其中一个表中添加了与另一表同名的无关列，连接条件会意外增加，导致结果集大幅缩小或为空。
> - **维护成本高：**TCHouse-X 场景，数据模型演进较快。列名的微小变动可能会导致下游报表或逻辑发生不可预知的变化。
> - **可读性较差：**对于不熟悉表结构的维护者来说，仅通过 SQL 语句很难直观判断连接的具体字段。

反连接 (Anti-joins)

反连接（Anti-join）是半连接的逻辑反面。它用于筛选出那些在另一张表中找不到匹配项的记录。这在查找“孤立数据”或执行“排除逻辑”时非常高效。

核心特性

- **左反连接 (LEFT ANTI JOIN)：**返回左表中所有与右表不匹配的行。
- **负向逻辑：**它表达的是“不存在”的关系，即：返回所有在右表中没有对应关联键的左表记录。
- **版本支持：**TCHouse-X 目前全面支持 `LEFT ANTI JOIN`；`RIGHT ANTI JOIN` 目前尚不支持，计划在后续版本中推出。

语法示例

以下查询将返回所有在 `t2` 表中没有对应项的 `t1` 记录：

```
SELECT t1.id, t1.c1
FROM t1 LEFT ANTI JOIN t2
ON t1.id = t2.id;
```

NULL 值的连接处理

默认情况下 `NULL = NULL` 为假。

> **说明：**
> 使用 `<=>`（NULL-Safe Equal）操作符。它比 `OR (A IS NULL AND B IS NULL)` 更简洁且性能更高，因为它能利用哈希连接。

性能调优

连接查询是资源密集型操作，优化顺序至关重要。

强制顺序控制

- `STRAIGHT_JOIN`：紧跟 `SELECT` 关键字，强制按 `FROM` 子句中的表顺序执行连接。

```
SELECT STRAIGHT_JOIN t1.id, t1.c1, t2.c2
FROM t1 JOIN t2
ON t1.id = t2.id;
```

- `LEADING` Hint：在复杂查询中精确指定顺序。

```
SELECT /*+ LEADING(t1 t2 t3) */ ... FROM t1 JOIN t2 ... JOIN t3 ...
```

统计信息与 CBO

TCHouse-X 的自研优化器 (Horn) 依赖统计信息进行 基于成本的优化 (CBO)。

统计信息维护命令

- 查看：`SHOW TABLE STATS [table_name]`
- 收集：`COMPUTE STATS [table_name]` (非分区) 或 `COMPUTE INCREMENTAL STATS` (分区表)。

手动策略（无统计信息时）

1. 先连接最大的表。
2. 优先连接过滤性（返回行数最少）的表。

LIMIT 子句

最近更新时间：2026-05-06 16:28:13

`LIMIT` 子句用于指定 `SELECT` 查询返回的最大行数。在分布式查询处理中，预先设定结果集上限能显著优化协调节点的内存分配，提升整体吞吐量。

语法规范

```
LIMIT constant_integer_literal
```

核心限制：

- **类型限制：**当前版本仅支持非负整数字面量（如 10）。
- **不支持表达式：**无法使用算术运算（2+2）、函数（LENGTH()）、类型转换（CAST()）或科学计数法（1e6）。
- **不支持子查询：**参数必须是静态确定的。
- **风格限制：**不支持 MySQL 风格的 LIMIT start, n。

场景	错误写法 (Unsupported)	正确写法 (Supported)
表达式	`LIMIT 10 + 5`	`LIMIT 15`
函数	`LIMIT LENGTH('abc')`	`LIMIT 3`
偏移量	`LIMIT 2, 10`	`LIMIT 10 OFFSET 2`

执行逻辑

在 TCHouse-X 中，`LIMIT` 不仅仅是过滤数据，它具有显著的性能加速作用：

- 本地限制：每个计算节点在本地扫描时，达到 `LIMIT` 数量后即停止扫描。
- 减少网络开销：仅传输必要的行数到协调节点，大幅降低节点间的通信延迟。
- 内存保护：防止由于 `WHERE` 条件过宽导致的海量数据涌入内存。

典型使用场景

- **Top-N / Bottom-N 查询：**必须配合 `ORDER BY` 使用，例如获取“评分最高的前 10 件商品”。
- **语法验证：**使用 `LIMIT 0`。系统会解析并生成执行计划，但不会执行实际的 I/O 或返回数据，适合测试 SQL 合法性。
- **数据采样：**不带 `ORDER BY` 使用 `LIMIT`。由于不需要全局排序，这种方式开销极低，适合预览表数据结构。
- **分页处理：**配合 `OFFSET` 使用。

> ⚠ **注意：**
> **分页建议：**在大数据场景下，高偏移量的 `OFFSET` 会导致查询变慢（因为系统仍需读取并丢弃前面的行）。最佳实践是尽可能在应用端进行数据缓存。

示例演示

基础用法

```
-- 获取满足条件的前 2 条记录
SELECT x FROM numbers WHERE x > 2 LIMIT 2;
```

Top-N 排序

```
-- 获取数值最高的前 3 名
SELECT x AS "Top 3" FROM numbers ORDER BY x DESC LIMIT 3;
```

偏移量分页

```
-- 跳过前 2 行，获取接下来的 3 行记录
SELECT * FROM numbers LIMIT 3 OFFSET 2;
```

特殊限制与说明

- **子查询限制：**在 `EXISTS` 和 `IN` 操作符中使用的相关子查询内部，严禁使用 `LIMIT` 子句。
- **顺序保证：**若不配合 `ORDER BY` 使用，`LIMIT` 返回的行具有不确定性，即多次执行结果可能因分布式扫描顺序不同而改变。

OFFSET 子句

最近更新时间：2026-05-06 16:28:13

`OFFSET` 子句用于跳过结果集中的前 N 条记录，从指定的偏移位置开始返回数据。在 TCHouse-X 中，结果集的行号从 0 开始计数，因此 `OFFSET 0` 在逻辑上等同于省略该子句。

核心用法与最佳实践

为了确保查询结果的确定性和实用性， `OFFSET` 必须与以下子句配合使用：

- **ORDER BY**：明确记录的物理/逻辑顺序，否则偏移的起点将具有随机性。
- **LIMIT**：界定返回的窗口大小（如获取第 101-110 条记录）。

性能警示

在 TCHouse-X 中，使用 `OFFSET` 进行深分页是一项高开销操作：

- **I/O 浪费**：即使结果集只需要 10 条数据，若设置了 `OFFSET 1000000`，系统仍需处理并排序前 100 万条记录，然后将其丢弃。
- **单点压力**：Coordinator 必须汇集所有节点产生的偏移前数据进行全局排序和计数，容易导致内存溢出或响应延迟。

应用场景与迁移建议

传统应用兼容

主要用于迁移那些深度依赖传统数据库分页逻辑（如 Web UI 分页控件）的旧系统。

推荐的替代架构

由于系统查询通常涉及大规模磁盘 I/O，**不建议**频繁发起多次带有不同 `OFFSET` 的查询。建议使用应用端缓存：一次性查询所需的所有潜在记录，将其缓存在应用层（如 Redis 或内存），由前端逻辑处理分页。

语法示例

以下展示了如何实现典型的分页查询：

```
-- 第 1 页：获取前 5 条记录
SELECT x FROM numbers ORDER BY x LIMIT 5 OFFSET 0;

-- 第 2 页：跳过前 5 条，获取接下来的 5 条记录
SELECT x FROM numbers ORDER BY x LIMIT 5 OFFSET 5;
```

注意事项

- **OFFSET 越界：**如果 `OFFSET` 的值超过了总结果集的行数，查询将成功执行但返回**空结果集**，不会报错。
- **数据一致性：**在多次分页请求之间，如果底层表数据发生了增删，可能会导致不同页面间出现重复或缺失的行。

ORDER BY 子句

最近更新时间：2026-05-06 16:28:13

`ORDER BY` 子句用于根据一个或多个列的值对 `SELECT` 语句的结果集进行升序（ASC）或降序（DESC）排列。

执行原理

TCHouse-X 中，执行 `ORDER BY` 是一项 高开销 操作，其过程如下：

1. **局部排序：**各计算节点并行对其负责的数据分片进行本地排序。
2. **结果汇聚：**排序后的数据流传输至协调器节点（Coordinator）。
3. **全局合并：**协调器通过归并算法将各节点的结果集串行合并。

> **注意：**
> - 资源消耗：排序相比普通查询更消耗内存。
> - 响应延迟：普通查询可以在匹配到行时即时返回，而 `ORDER BY` 必须等待所有节点处理完成并由协调器完成全局排序后，才能开始向客户端返回首行数据。

语法规范

```
ORDER BY col_ref [, col_ref ...] [ASC | DESC] [NULLS FIRST | NULLS LAST]

col_ref ::= column_name | integer_literal
```

列引用方式：

- **名称引用：**最常用的方式，如 `ORDER BY price`。
- **位置索引（列号）：**可以使用整数常量。例如 `ORDER BY 1` 表示按 `SELECT` 列表中的第一列排序。

> **说明：**
> 列号必须是显式列出的列，不能在 `SELECT *` 语句中使用。必须是数字常量，不可使用复杂的表达式或字符串。

排序行为

升序与降序

- `ASC`（默认）：从小到大排列。
- `DESC`：从大到小排列。

性能优化

ORDER BY 与 LIMIT 结合使用

原理：如果包含 `LIMIT 10`，每个计算节点仅需将本地排序后的前 10 条数据发送给协调器。Coordinator 只需在极小的中间结果集中选取前 10 条，极大地降低了网络传输压力和内存需求。

```
-- 执行高效的 Top 10 查询
SELECT user_id, SUM(page_views) `sum`
FROM web_stats
GROUP BY user_id
ORDER BY `sum` DESC
LIMIT 10;
```

避免无效排序

子查询忽略：TCHouse-X 通常会忽略子查询或视图定义内部的 `ORDER BY`。若要保证最终输出有序，必须将 `ORDER BY` 应用于最外层的 `SELECT`。

分页

虽然可以使用 `LIMIT` 与 `OFFSET` 实现分页，但在处理大表时，由于越往后的页面需要跳过的数据越多，效率会逐渐降低。

```
-- 分页示例
SELECT page_title FROM search_content
ORDER BY page_title
LIMIT 10 OFFSET 20; -- 获取第 3 页数据
```

UNION 子句

最近更新时间：2026-05-06 16:28:13

`UNION` 子句用于将两个或多个 `SELECT` 语句的结果集组合成一个单一的结果集。它在逻辑上是对数据集的“并集”操作。

语法规范

```
query_1 UNION [DISTINCT | ALL] query_2
```

- **UNION DISTINCT（默认）：** 合并结果集并移除重复行。
- **UNION ALL：** 合并结果集并保留所有行（包括重复行）。

关键对比与性能建议

特性	UNION ALL	UNION (DISTINCT)
重复值处理	保留所有重复记录	自动应用 DISTINCT 去重
内存消耗	**低。**数据直接流式传输	**高。**需在内存中维护哈希表以检测重复
处理速度	**快。**几乎无额外开销	**慢。**涉及昂贵的去重计算
推荐场景	确定数据无重复，或业务允许重复	必须保证结果集唯一时

> ⚠ **注意：**
> 在 TCHouse-X 中，处理数百万行数据时，去重操作会导致巨大的 CPU 和内存开销。若业务逻辑允许，强烈建议优先使用 `UNION ALL`。

高级用法说明

全局排序与分页：如果直接对单个查询使用 ORDER BY，它仅作用于该子部分。若要对**合并后的最终结果集**进行全局排序，必须将 UNION 语句包装在子查询中：

```
-- 推荐：对合并后的所有结果进行全局排序
SELECT * FROM (
  SELECT x FROM t1
  UNION ALL
  SELECT x FROM t2
) AS combined_results
```

```
ORDER BY x
LIMIT 10;
```

示例演示

UNION ALL：高性能合并（保留重复）

```
-- 即使两个表中都有数值 1，也会全部返回
SELECT x FROM few_ints
UNION ALL
SELECT x FROM few_ints;
-- 结果包含原始数据集的两倍行数
```

UNION DISTINCT：去重合并（默认行为）

```
-- 返回两个表中不重复的唯一值
SELECT x FROM few_ints
UNION
SELECT 10;
-- 结果集会自动去重，耗时相对较长
```

注意事项

- **列匹配：**所有 `SELECT` 语句中的列数量必须相同，且对应位置的数据类型必须兼容。
- **列名定义：**结果集的列名通常由**第一个** `SELECT` 语句决定。
- **LIMIT 子句：**在 TCHouse-X 中，对 `UNION` 的结果集使用 **ORDER BY** 时，不再强制要求配合 **LIMIT**，但为了性能考虑，建议在最外层按需添加 LIMIT。

WITH 子句

最近更新时间：2026-05-06 16:28:13

`WITH` 子句允许在 `SELECT` 、 `INSERT` 或 `UPDATE` 语句之前定义一个或多个临时结果集，这些结果集被称为公用表表达式 (CTE)。它类似于定义了仅在当前查询范围内有效的“局部视图”。

核心价值

- **消除冗余（方便维护）：**将多次引用的复杂子查询提取到顶部，减少代码重复，降低维护成本。
- **提升可读性：**将嵌套过深、逻辑复杂的 SQL “扁平化”，使其更接近人类阅读习惯（从上到下执行逻辑）。
- **冲突隔离：**定义的别名仅在当前查询结束后即销毁，不会与数据库中的实际表名或视图名产生冲突。
- **跨系统兼容：**广泛支持 ANSI SQL 标准（如 Oracle、PostgreSQL、MySQL 8.0+ 等），增强了代码的可移植性。

使用限制

递归支持：当前版本**不支持递归查询**（即 `WITH RECURSIVE` ），而某些其他数据库系统支持该功能。

代码示例

多重定义与引用

```
-- 定义两个临时表 t1 和 t2，并执行合并插入
WITH
  t1 AS (SELECT 1 AS val),
  t2 AS (SELECT 2 AS val)
INSERT INTO tab
SELECT * FROM t1
UNION ALL
SELECT * FROM t2;
```

嵌套作用域定义

CTE 也支持在不同查询层级进行定义。以下示例展示了外层定义与内层嵌套定义的结合使用：

```
-- 在外层定义 t1，同时在 UNION ALL 的分支内局部定义 t2
WITH t1 AS (SELECT 1)
(
  WITH t2 AS (SELECT 2)
  SELECT * FROM t2
```

```
)
UNION ALL
SELECT * FROM t1;
```

与传统子查询对比

特性	传统子查询 (Subquery)	WITH 子句 (CTE)
逻辑顺序	由内而外（嵌套式）	由上而下（流式）
复用性	无法直接复用，需多次书写逻辑	一次定义，多次引用
代码维护	随着嵌套加深，代码极难维护	逻辑清晰，易于调试

子查询

最近更新时间：2026-05-06 16:28:13

子查询是嵌套在另一个查询中的 `SELECT` 语句。它允许查询逻辑根据另一个表的内容动态调整，为 SQL 提供了强大的表达能力和灵活性。

子查询分类

标量子查询 (Scalar Subqueries)

- **定义：**返回结果仅为 **单行单列** 的子查询。
- **特性：**常用于聚合计算（如 `MAX()`）。该单一值可以替换到任何需要标量值的上下文中（如比较运算符）。
- **空值处理：**如果子查询未匹配到任何行，结果视为 `NULL`。
- **示例：** `SELECT x FROM t1 WHERE x > (SELECT MAX(y) FROM t2);`

非关联子查询 (Non-Correlated Subqueries)

- **定义：**子查询内部不引用外层查询的任何表。
- **执行逻辑：**子查询仅计算一次，其结果集作为常量提供给外层查询遍历使用。
- **示例：** `SELECT x FROM t1 WHERE x IN (SELECT y FROM t2);`

关联子查询 (Correlated Subqueries)

- **定义：**子查询内部引用了外层查询的列。
- **执行逻辑：**外层查询的每一行都会驱动子查询重新计算，具有极强的动态过滤能力。
- **示例：**

```
-- 查找工资高于所在部门平均水平的员工
SELECT name FROM employees e1
WHERE salary > (SELECT AVG(salary) FROM employees e2 WHERE e1.dept_id =
e2.dept_id);
```

语法与位置

```
SELECT select_list FROM table_ref [, table_ref ...]

table_ref ::= table_name | (select_statement)
```

子查询可以出现在以下位置：

1. **FROM 子句：**作为“派生表”或“内联视图”。
2. **WHERE 子句：**配合比较运算符（ `=` 、 `>` ）、 `[NOT] IN` 或 `[NOT] EXISTS` 。
3. **HAVING 子句：**用于过滤聚合后的组数据。

开发建议与调优

- **表别名引用：**如果内层和外层引用了同一张表，必须使用别名来区分列的作用域。

```
SELECT * FROM t1 `one` WHERE id IN (SELECT parent FROM t1 `two` WHERE
`one`.parent = `two`.id);
```

- **连接顺序控制：** `STRAIGHT_JOIN` 提示仅作用于其所在的当前查询块。如果需要优化嵌套视图或子查询的连接顺序，需在对应的嵌套块内分别添加提示。
- **性能权衡：**关联子查询虽然功能强大，但在处理大数据量时可能会导致频繁的子查询评估。在性能敏感场景下，优先考虑将其重写为 `JOIN` 形式。

限制

为保证分布式执行的稳定性，TCHouse-X 对子查询施加了以下限制：

- 运算符限制：标量子查询不能与 `ANY` 、 `ALL` 或 `SOME` 运算符一起使用。
- 比较限制：对于 `EXISTS` 子句，外层与内层之间的比较必须包含至少一次相等比较。
- 上下文限制：
  - 标量子查询仅支持数值上下文。不能用于 `LIKE` 、 `REGEXP` 或与 `TIMESTAMP` 等非数值类型的比较。
  - 不能在 `CASE` 表达式中作为比较值或返回值。
  - 不能用作 `BETWEEN` 运算符的参数。
- 逻辑连接限制：子查询不能出现在 `OR` 连接条件的外部（即 `WHERE (subquery) OR col = 1` 是不允许的）。
- 引用限制：在子查询中引用外层列时，必须使用完全限定名。

SHOW 语句

最近更新时间：2026-05-06 16:28:13

`SHOW` 语句用于获取数据库、表、视图、分区、列、会话变量等信息。

语法

```
-- 展示数据库列表
SHOW DATABASES [[LIKE] 'pattern']
SHOW SCHEMAS [[LIKE] 'pattern']        -- SHOW DATABASES 的别名

-- 展示指定数据库下的表列表
SHOW TABLES [IN database_name] [[LIKE] 'pattern']
SHOW TABLES [FROM database_name] [LIKE 'pattern']
SHOW FULL TABLES FROM database_name

-- 展示表、视图的建表语句
SHOW CREATE TABLE [database_name.]table_name
SHOW CREATE VIEW [database_name.]view_name

-- 展示表、列统计信息
SHOW TABLE STATS [database_name.]table_name
SHOW COLUMN STATS [database_name.]table_name

-- 展示表分区（仅适用于外表）
SHOW PARTITIONS [database_name.]table_name

-- 展示数据库建库语句
SHOW CREATE DATABASE [database_name]

-- 展示表的列信息
SHOW [FULL] COLUMNS FROM [database_name.]table_name

-- 展示库的表信息
SHOW TABLE STATUS [FROM database_name] [LIKE 'pattern']

-- 展示会话属性
SHOW VARIABLES [LIKE 'pattern']
```

通配符规则

`pattern` 参数是一个带引号的字符串字面量，用于模糊匹配对象名称：

符号	含义	示例	匹配结果
`*` 或者 `%`	匹配任意多个字符	`'a*'`	以 a 开头的所有名称
`|`	表示"或"条件	`'*dim*|*fact*'`	包含 dim 或 fact 的名称

其他语句
COMMENT 语句

最近更新时间：2026-05-06 16:28:13

`COMMENT` 语句用于为数据库、表或列添加、修改或删除描述性注释。这些注释存储在元数据中，方便通过查询工具查看。

语法

```
-- 为数据库设置或移除注释
COMMENT ON DATABASE db_name IS {'comment_string' | NULL}

-- 为表设置或移除注释
COMMENT ON TABLE [db_name.]table_name IS {'comment_string' | NULL}

-- 为列设置或移除注释
COMMENT ON COLUMN [db_name.]table_name.column_name IS {'comment_string'
| NULL}
```

参数说明

- **db_name**：可选。如果目标数据库不是当前所在的数据库，则必须指定。
- **NULL**：如果指定为 `NULL`，则会从指定对象中移除已有的注释。
- `'comment_string'`：注释内容。字符串必须用单引号包裹。
  - 长度限制：注释最长可达 256 个字符。

操作示例

添加或修改注释

```
COMMENT ON DATABASE sales_db IS '该库包含所有销售交易数据';
COMMENT ON TABLE sales_db.orders IS '订单主表，按日期分区';
COMMENT ON COLUMN sales_db.orders.order_id IS '订单唯一识别码';
```

移除已有注释

```
COMMENT ON COLUMN orders.order_id IS NULL;
```

其他设置与查看方式

除了 `COMMENT ON` 语句，您还可以通过以下方式管理注释：

1. 创建对象时添加：在 `CREATE TABLE` 时使用 `COMMENT` 关键字。

```
CREATE TABLE t1 (id INT COMMENT '用户ID') COMMENT '用户信息表';
```

2. 修改表属性时添加：使用 `ALTER TABLE` 修改属性。

```
ALTER TABLE t1 SET TBLPROPERTIES('comment' = '新的表描述');
```

3. 查看注释：使用 `DESCRIBE` 命令查看列注释，或使用 `SHOW CREATE TABLE` 查看表注释。

```
DESCRIBE table_name;            -- 查看列的 Comment 字段
DESCRIBE FORMATTED table_name; -- 查看表的详细元数据和注释
SHOW CREATE TABLE table_name;   -- 查看完整的建表 SQL 及注释
```

VALUES 语句

最近更新时间：2026-05-06 16:28:13

`VALUES` 子句不仅可以作为 `INSERT` 语句的一部分，还可以作为独立的语句或与 `SELECT` 结合使用。它允许在不创建物理表的情况下，动态构造一个内存中的数据集。

语法

独立使用

```
VALUES (row)[, (row), ...];

row ::= column [, column, ...]
```

派生表使用

```
SELECT select_list
  FROM (VALUES (row)[, (row), ...]) [AS alias[(col_alias1, ...)]];

row ::= column [, column, ...]
```

使用规则与约束

- 结构一致性： 每一行 (row) 必须包含相同数量的列 (column)。
- 类型兼容性： 各行中对应列的数据类型必须相互兼容（如数值型与浮点型可兼容，日期字符串与时间戳可兼容）。
- 命名机制：
  - 若未显式指定派生表名，系统默认分配别名 `expr$0`。建议使用 AS 关键字显式命名以增强代码可读性。
  - 默认列名遵循 `派生表名.expr$N` 规范（其中 `N` 为从 0 开始起始的列索引）。
- 内容灵活： 列的内容可以是常量、变量或表达式。

示例

构造匿名数据集（默认命名）：

```
-- 构造一个 2 行 3 列的派生表，使用默认列命名
SELECT * FROM (VALUES (4,5,6), (7,8,9));
-- 输出：
```

```
-- +------------------+------------------+------------------+
-- | `expr$0`.`expr$0` | `expr$0`.`expr$1` | `expr$0`.`expr$2` |
-- +------------------+------------------+------------------+
-- |                4 |                5 |                6 |
-- |                7 |                8 |                9 |
-- +------------------+------------------+------------------+
```

指定派生表别名：

```
-- 通过 AS 关键字定义表名，简化引用路径。
SELECT * FROM (VALUES (4,5,6), (7,8,9)) AS t(x,y,z);

-- 输出结果：
-- +--------------+--------------+--------------+
-- | `t`.`expr$0` | `t`.`expr$1` | `t`.`expr$2` |
-- +--------------+--------------+--------------+
-- |            4 |            5 |            6 |
-- |            7 |            8 |            9 |
-- +--------------+--------------+--------------+
```

自定义列别名：

```
-- 在定义表名的同时，通过括号显式指定每一列的名称。
SELECT * FROM (VALUES (4,5,6), (7,8,9)) AS t(x, y, z);

-- 输出结果：
-- +------+------+------+
-- | x    | y    | z    |
-- +------+------+------+
-- |    4 |    5 |    6 |
-- |    7 |    8 |    9 |
-- +------+------+------+
```

USE 语句

最近更新时间：2026-05-06 16:28:13

`USE` 语句用于将当前会话切换到指定的数据库。当前数据库是执行 `CREATE TABLE`、`INSERT`、`SELECT` 或其他语句时，默认作用的对象所在的数据库（除非您在对象名前显式加上了数据库名称）。
新的当前数据库在会话期间内有效，直到执行另一个 `USE` 语句。

语法

使用 `USE` 语句时，您只需指定要切换到的数据库名称。

```
USE db_name
```

> ⚠ **注意：**
> 默认情况下，当您连接到 TCHouse-X 实例时，会话的初始数据库名为 `default`。

使用说明

切换默认数据库在以下情况下非常方便且能提高效率：

- 简化表引用：避免在每次引用表时都加上数据库名称（如 `database_name.table_name`）。
  - 优化前：`SELECT * FROM db.t1 JOIN db.t2`
  - 优化后：`SELECT * FROM t1 JOIN t2`
- 连续操作便捷性：当您需要在同一数据库内进行一系列操作（例如创建表、插入数据和查询表）时，无需重复指定数据库名，保持操作的聚焦性。

内置函数
内置函数概览

最近更新时间：2026-05-06 16:28:13

TCHouse-X 在线引擎提供了丰富的内置函数库。通过这些函数，您可以在 SQL 语句中直接实现复杂的数学计算、文本处理、日期转换以及高级数据分析，极大地简化了业务逻辑的开发。

函数分类指南

我们将函数按功能划分为以下类别，您可以根据处理数据的需求选择合适的函数：

类别	功能描述	常用示例
数学函数	执行基础及高级数学运算，如三角函数、对数、取整等。	`ABS()`，`ROUND()`，`CEIL()`
位操作函数	针对二进制位进行操作，适用于高性能的标记位计算。	`BIT_AND()`，`BIT_XOR()`
比较函数	用于数值、字符串或日期的逻辑比较。	`LEAST()`，`GREATEST()`，`COALESCE()`
字符串函数	实现文本提取、拼接、大小写转换及格式化。	`CONCAT()`，`SUBSTR()`，`TRIM()`
日期与时间函数	处理日期、时间的格式转换、计算与提取，支持时间窗口统计和时区转换。	`CURRENT_DATE()`，`DATEDIFF()`
正则表达式函数	基于正则规则进行复杂的文本匹配、提取与替换。	`REGEXP_LIKE()`，`REGEXP_EXTRACT()`
聚合函数	对一组值进行计算并返回单个结果，常用于 GROUP BY。	`SUM()`，`AVG()`，`COUNT_DISTINCT()`
窗口函数	在不分组的情况下对行集进行计算，常用于排名和移动平均。	`ROW_NUMBER()`，`RANK()`，`LAG()`
URL 函数	专门用于解析 URL 字符串，提取域名、查询参数等信息。	`URL_EXTRACT_HOST()`，`URL_DECODE()`

数学函数

最近更新时间：2026-05-06 16:28:13

> **说明：**
> 适用版本：TCHouse-X 内核版本 2.0.0及以上版本。

数学函数（Arithmetic Functions）用于执行比基础加减乘除更复杂的数值计算。TCHouse-X 提供了丰富的函数库，涵盖三角学、对数、指数、进制转换及统计分桶等操作。

基础算术与算术扩展

函数	返回类型	说明	示例
`abs(x)`	与 `x` 相同	返回 `x` 的绝对值。	`select abs(-10); --10`
`add(x, y)`	与 `x` 相同	返回 `x + y`。`x` 与 `y` 类型须一致。	`select add(5,2); --7`
`checked_add(x, y)`	与 `x` 相同	安全加法。溢出时报错。	`select checked_add(5,2); --7`
`subtract(x, y)`	与 `x` 相同/`Decimal`	返回 `x - y`。`x` 与 `y` 类型须一致。	`select subtract(10, 2); -- 8`
`checked_subtract(x, y)`	与 `x` 相同/`Decimal`	安全减法，在溢出时报错。	-
`multiply(x, y)`	与 `x` 相同/`Decimal`	返回 `x * y`。`x` 与 `y` 类型须一致。	`select multiply(2, 4); -- 8`
`checked_multiply(x, y)`	与 `x` 相同/`Decimal`	安全乘法，在溢出时报错。	-
`divide(x, y)`	`Double` / `Decimal`	返回 `x ÷ y`。`x` 与 `y` 类型须一致。执行浮点除法，结果保留 6 位小数，除以零返回 `NULL`。	`select divide(2,4); -- 0.500000`
`checked_divide(x, y)`	`Double` / `Decimal`	安全除法，在溢出时报错。	-

`x div y`	`Bigint`	整数除法，结果向下取整。溢出或除以零返回 `NULL`。	`select 1 div 2; -- 0`
			`select 3 div 2; -- 1`
			`select -3 div 2; -- -2`

`pmod(n, m)`	与 `n` 相同	返回 `n ÷ m` 的正余数。	`select pmod(-10, 3); -- 2` `select pmod(10, 3); -- 1`
`remainder(n, m)`	与 `n` 相同	返回 `n % m` 的余数（同 Spark `%`）。	`select remainder(-10, 3); -- -1` `select remainder(10, 3); -- 1`

`unaryminus(x)`	与 `x` 相同	返回 `-x`。	`select unaryminus(1); -- -1`
			`select unaryminus(-1); -- 1`
			`select unaryminus(0); -- 0`

数值修约与比较函数

函数	返回类型	说明	示例
`ceil(x)`	与 `x` 相同	向上取整。支持 `Bigint`，`Double`，`Decimal`。	`select ceil(1.23); --2`
`floor(x)`	与 `x` 相同	向下取整。支持 `Bigint`，`Double`，`Decimal`。	`select floor(1.23); --1`
`round(x, d)`	与 `x` 相同	四舍五入到 `d` 位（使用 HALF_UP 模式）。	`select round(1.2345, 2); --1.23`
`rint(x)`	`Double`	返回最接近 `x` 的整数（双精度）。	`select rint(12.34); --12` `select rint(12.54); --13`

greatest(x, ...)	与 x 相同	返回参数列表中的最大值，忽略 NULL。	select greatest(1, 5, 2); --5
least(x, ...)	与 x 相同	返回参数列表中的最小值，忽略 NULL。	select least(1, 5, 2); --1
sign(x) / signum	Double	返回符号：正数 1，负数 -1，零 0。	select sign(100);--1 select sign(-100);--1 select sign(0);-- 0

指数、对数与幂运算

函数	返回类型	说明	示例
e()	Double	返回欧拉常数 e。	select e(); --2.718281828 4590451
exp(x)	Double	返回欧拉常数 e 的 x 次幂。	select exp(1); --2.718281 8284590451 select exp(2); --7.389056 09893065
expm1(x)	Double	返回 e^x - 1。在 x 接近 0 时比 exp(x)-1 更精确。	select expm1(1); --1.718 2818284590451
ln(x) / log(x)	Double	返回 x 的自然对数（以 e 为底）。	select log(e()); --1 select ln(e()); --1
log(base, x)	Double	返回以 base 为底的 x 的对数。	select log(2, 8); --3
log10(x)	Double	返回以 10 为底的对数。	select log10(100); --2
log2(x)	Double	返回以 2 为底的对数。	select log2(8); --3
log1p(x)	Double	返回 ln(x + 1)。	select log1p(0); --0

pow(x, p)	Double	返回 x 的 p 次幂。	select pow(2, 3); --8
sqrt(x)	Double	返回 x 的平方根。	select sqrt(9); --3
cbrt(x)	Double	返回 x 的立方根。	select cbrt(8);--2
factorial(x)	Bigint	返回 x 的阶乘（x 须在 0-20 之间）。	select factorial(5); --120

三角函数

函数	返回类型	说明	示例
sin(x)	Double	返回 x（弧度）的正弦值。	SELECT sin(PI()/2); -- 结果：1 (90° 的正弦)
cos(x)	Double	返回 x（弧度）的余弦值。	SELECT cos(PI()); -- 结果：-1 (180° 的余弦)
tan(x)	Double	返回 x（弧度）的正切值。	SELECT tan(PI() / 4); -- 预期 1.0，实际返回 0.9999999999999989（浮点精度误差，非 bug）
asin(x)	Double	返回 x（弧度）的反正弦值。	SELECT asin(1.0); --1.57079... (π/2)
acos(x)	Double	返回 x（弧度）的反余弦值。	SELECT acos(-1.0); --3.14159... (π)
atan(x)	Double	返回 x（弧度）的反正切值。	SELECT atan(1.0); --0.78539... (π/4)
sinh(x)	Double	返回 x（弧度）的双曲正弦 。	SELECT sinh(1.0); --1.1752...
cosh(x)	Double	返回 x（弧度）的双曲余弦。	SELECT cosh(0); --1

`tanh(x)`	`Double`	返回 `x`（弧度）的双曲正切值。	`SELECT tanh(0.5); --0.4621...`
`asinh(x)`	`Double`	返回 `x`（弧度）的反双曲正弦值。	`SELECT asinh(1.0); --0.8813...`
`acosh(x)`	`Double`	返回 `x`（弧度）的反双曲余弦值。	`SELECT acosh(1.0); --0`
`atanh(x)`	`Double`	返回 `x`（弧度）的反双曲正切值。	`SELECT atanh(0.5); --0.5493...`
`cot(x)`	`Double`	返回 `x`（弧度）的余切值。	`SELECT cot(PI() / 4); --` 预期结果 `1.0`，实际返回 `1.0000000000000002`（浮点精度误差，非 `bug`）
`csc(x)`	`Double`	返回 `x`（弧度）的余割值。	`SELECT csc(PI()/2);--1`（余割，`1/sin`）
`sec(x)`	`Double`	返回 `x`（弧度）的正割值。	`SELECT sec(0); --` 结果：`1.0`（正割，`1/cos`）
`atan2(y, x)`	`Double`	返回 `y/x` 的反正切值。兼容 Spark 极端情况。	`SELECT atan2(1, 1); --0.78539...`（坐标 `(1,1)` 的极角）
`degrees(x)`	`Double`	将弧度转换为角度。	`SELECT degrees(PI()); -- 180`
`radians(x)`	`Double`	将角度转换为弧度。	`SELECT radians(180); --3.14159...`（π）
`hypot(a, b)`	`Double`	返回 `sqrt(a^2 + b^2)` 的值 。	`SELECT hypot(3, 4); --5`

进制转换与位操作

函数	返回类型	说明	示例
`bin(x)`	`Varchar`	返回 `Long` 类型 `x` 的二进制字符串。	`SELECT bin(10); --'1010'`

`hex(x)`	`Varchar`	返回 `x` 的十六进制字符串。支持数字、字符串和二进制。	`SELECT hex(255); --'FF'`
`unhex(x)`	`Varbinary`	将十六进制字符串转回二进制。匹配 Spark 3.5.3 行为。	`SELECT hex('ABC'); --'414243'`
`conv(n, f, t)`	`Varchar`	将数字从 `f` 进制转换为 `t` 进制。	`SELECT conv('1010', 2, 10); --'10'` `SELECT conv('FF', 16, 2); --'11111111'`
`unscaled_value(x)`	`Bigint`	返回短小数（`Short Decimal`）的原始 `Bigint` 值。	`SELECT unscaled_value(CAST(12.34 AS DECIMAL(10, 2))); --1234`

其他特殊函数

函数	返回类型	说明	示例
`isnan(x)`	`Boolean`	检查 `x` 是否为 `NaN`。支持 `Float` / `Double`。	`SELECT isnan(cast('nan' as double)); -- 1`
`rand([seed])` / `random([seed])`	`Double`	返回 `[0, 1)` 范围内的随机数。指定 `seed` 可获得确定性结果。	`SELECT rand(); -- 结果随机` `SELECT rand(123); -- 结果固定`
`not(x)`	`Boolean`	`not true` 返回 `false`	`SELECT not(1 = 1); --0`
`width_bucket(x, b1, b2, n)`	`Bigint`	将 `x` 分配到从 `b1` 到 `b2` 划分为 `n` 个等宽桶中的桶编号。	```SELECT val, width_bucket(val, 0, 100, 4) as bucket_id FROM (```

```
        VALUES
            (-5),
            (20),
            (50),
            (90),
            (110)
    ) AS t(val);
+------+-----------+
| val  | bucket_id |
+------+-----------+
|   -5 |         0 |
|   20 |         1 |
|   50 |         3 |
|   90 |         4 |
|  110 |         5 |
+------+-----------+
```

位操作函数

最近更新时间：2026-05-06 16:28:13

> **说明：**
> 适用版本：TCHouse-X 内核版本 2.0.0及以上版本。

逻辑位运算符

此类函数通过比较两个整数对应的二进制位来计算结果。

函数	说明	SQL 样例	结果 (二进制)
`bitwise_and(x, y)`	返回 `x` 和 `y` 的按位与结果	`SELECT bitwise_and(3, 5);`	1 (0011 & 0101 = 0001)
`bitwise_or(x, y)`	返回 `x` 和 `y` 的按位或结果	`SELECT bitwise_or(3, 5);`	7 (0011 | 0101 = 0111)
`bitwise_xor(x, y)`	返回 `x` 和 `y` 的按位异或结果	`SELECT bitwise_xor(3, 5);`	6 (0011 ^ 0101 = 0110)
`bitwise_not(x)`	返回 `x` 的按位非值 (NOT)	`SELECT bitwise_not(0);`	-1

位统计与提取

用于分析数值内部的 1 分布情况。

函数	说明	SQL 样例	结果
`bit_count(x)`	统计参数 `x` 中被设置为 `1` 的位数。	`SELECT bit_count(7);`	`3` (`7` 的二进制是 `111` ，共有 3 个 "1"。)
`bit_get(x, pos)`	返回指定位置 `pos` 的位值 (`0` 或 `1`)。	`SELECT bit_get(11, 3);`	`1` (`11` 二进制为 `1011` 。第 3 位 (从右往左，0 开始) 是 `1` 。)
`getbit(x, pos)`	同上	`SELECT getbit(11, 2);`	`0` (功能同上。第 2 位是 `0` 。)

> ⚠ **注意：**
> `bit_get` 的 `pos` 不能超过类型的最大位数（如 `INTEGER` 是 31，`BIGINT` 是 63），否则会报错。

位移操作

在底层算法中常用于快速乘、除以 2^n。

函数	SQL 样例	结果	计算过程
`shiftleft(x, n)`	`SELECT shiftleft(2, 2);`	`8`	`2` (0010) 左移 `2` 位变为 8 (1000)。相当于 2 * (2^2)。
`shiftright(x, n)`	`SELECT shiftright(8, 1);`	`4`	`8` (1000) 右移 1 位变为 4 (0100)。相当于 8 / (2^1)。

比较函数

最近更新时间：2026-05-06 16:28:13

说明：

- 适用版本：TCHouse-X 内核版本 2.0.0及以上版本。
- TCHouse-X 遵循 MySQL 协议规范，布尔值以整数表示：
  - 1：表示 TRUE
  - 0：表示 FALSE

相等性判断

函数名称	运算符	说明	示例
`equalto`	`=`	`NULL = NULL` 结果为 `NULL`。	`SELECT equalto(null, null); -- NULL`
`equalnullsafe`	`<=>`	`NULL <=> NULL` 结果为 `1`	`SELECT equalnullsafe(null, null); -- 1`
`neq`	`!=`	任意操作数为 `NULL` 则结果为 `NULL`。	`SELECT neq(1,2); -- 1`
`isnull`	`IS NULL`	判断输入是否为 `NULL`。	`SELECT isnull(1); -- 0`
`isnotnull`	`IS NOT NULL`	判断输入是否不为 `NULL`。	`SELECT isnotnull(1); -- 1`

范围与大小比较

函数	运算符	说明	示例
`between(x, min, max)`	`BETWEEN`	检查 `x` 是否在闭区间 `[min, max]` 内。	`SELECT `between`(15, 10, 20);-- 1`
`greaterthan(x,y)`	`>`	`x` 严格大于 `y` 时返回 `1`。	`SELECT greaterthan(10, 5); --1`

greaterthanorequal(x, y)	>=	x 大于或等于 y 时返回 1 。	SELECT greaterthanorequal(10, 5); --1
lessthan(x, y)	<	x 严格小于 y 时返回 1 。	SELECT lessthan(10, 5); --0
lessthanorequal(x, y)	<=	x 小于或等于 y 时返回 1 。	SELECT lessthanorequal(10, 5); --0

空值与异常处理

函数名称	说明	示例
coalesce(x, ...)	返回参数列表中的第一个非空值。若全为 NULL 则返回 NULL 。	SELECT coalesce(NULL, NULL, 'Spark', 'SQL'); -- 'Spark'
ifnull(x, y) / nvl(x, y)	若 x 为 NULL 则返回 y ；否则返回 x 。	SELECT ifnull(NULL, 0), nvl('A', 'B'); --0, 'A'
nvl2(x, y, z)	若 x 不为 NULL 返回 y ；否则返回 z 。	SELECT nvl2('data', 'Valid', 'Missing'); -- 'Valid'
nullif(x, y)	若 x = y 返回 NULL ；否则返回 x 。	SELECT nullif(100, 100), nullif(100, 200); --NULL, 100
nanvl(x, y)	若 x 不是 NaN 返回 x ；否则返回 y 。通常用于将浮点异常值替换为默认值。	SELECT nanvl(CAST('NaN' AS DOUBLE), 0.0); --0

字符串函数

最近更新时间：2026-05-06 16:28:13

基础转换与编码

用于处理字符编码、进制转换及 Base64 编解码。

函数	说明	示例
`ascii(str)`	返回字符串 `s` 首字符的 Unicode/ASCII 码位。若为空串则返回 0。	`SELECT ascii('123'); -- 49`
`chr(n)` / `char(n)`	`ascii` 的逆操作。将 Unicode 码点 `n` 转换为字符。若 `n > 255` 通常执行取模运算。	`SELECT char(97); -- 'a'`
`hex(x)`	转十六进制。支持将数字、字符串或二进制转换为十六进制字符串。	`SELECT hex('Spark'); -- '537061726b'`
`unhex(str)`	十六进制反转。将十六进制字符串还原为原始二进制/字符串格式。	`SELECT unhex('537061726b'); -- 'Spark'`
`base64(bin)`	将二进制数据转换为 Base64 编码的文本字符串。	`SELECT base64(unhex('537061')); -- 'U3Bh'`
`unbase64(str)`	将 Base64 编码的字符串解码回二进制数据。	`SELECT unbase64('U3BhcmsgU1FM'); -- 'Spark SQL'`
`conv(n, f, t)`	通用进制转换。将数值 `n` 从 `f` 进制转换为 `t` 进制（支持 2 到 36 进制）。	`SELECT conv('100', 2, 10); -- '4'`

长度与位置检索

用于测量长度或定位子串。

函数	说明	示例
`char_length(s)`	返回字符串中的字符个数（包含末尾空格）。	`SELECT char_length('SQL '); -- 4`

`length(s)`	返回字符数（在某些库中等同于 `char_length`，但在部分数据库指字节数）。	`SELECT length('Spark '); -- 6`
`bit_length(s)`	返回字符串占用的位长度（1 字符 = 8 bit）。	`SELECT bit_length('123'); -- 24`
`instr(s, sub)`	返回子串 `sub` 在 `s` 中首次出现的位置（索引从 1 开始）。	`SELECT instr('Spark', 'p'); -- 2`
`locate(sub, s, p)`	从位置 `p` 开始，查找 `sub` 在 s 中出现的位置。	`SELECT locate('aa', 'aaads', 2); -- 2`
`find_in_set(s, list)`	在以逗号分隔的列表 `list` 中查找 `s` 的索引位置。	`SELECT find_in_set('b', 'a,b,c'); -- 2`
`levenshtein(s1, s2)`	编辑距离。计算将 `s1` 转换为 `s2` 所需的最少单字符编辑次数（插入/删除/替换）。	`SELECT levenshtein('kitten', 'sitting'); -- 3`
`soundex(s)`	语音特征码。根据读音返回 4 位代码，用于查找读音相似但拼写不同的单词。	`SELECT soundex('Miller'); -- 'M460'`

子串提取与修剪

函数	说明	示例
`left(s, len)`	返回字符串 `s` 最左侧的 `len` 个字符。	`SELECT left('Spark', 2); -- 'Sp'`
`right(s, len)`	返回字符串 `s` 最右侧的 len 个字符。	`SELECT right('Spark', 1); -- 'k'`
`substring(s, p)`	从位置 `p` 开始截取到末尾（注意：SQL 索引通常从 1 开始）。	`SELECT substring('Spark', 2); -- 'park'`
`substring(s, p, l)`	从位置 `p` 开始，截取长度为 `l` 的子串。	`SELECT substring('Spark', 2, 2); -- 'pa'`
`substring_index(s, d, c)`	返回分隔符 `d` 第 `c` 次出现之前（`c` 为正）或之后（`c` 为负）的所有内容。	`SELECT substring_index('a.b.c', '.', 2); -- 'a.b'`

`ltrim(s)`	删除字符串左侧的所有空格。	`SELECT ltrim('  data'); -- 'data'`
`rtrim(s)`	删除字符串右侧的所有空格。	`SELECT rtrim('data  '); -- 'data'`
`trim(s) / btrim(s)`	同时删除字符串两侧的空格。	`SELECT trim('  spark  '); -- 'spark'`
`empty2null(s)`	如果字符串是空字符串 `''`，则返回 `NULL`；否则返回原值。	`SELECT empty2null(''); -- NULL`

字符串修改与填充

函数	说明	示例
`initcap(s)`	将字符串中每个单词的首字母转换为大写，其余小写。	`SELECT initcap('hi world'); -- 'Hi World'`
`lower(s) / lcase(s)`	将字符串 `s` 中的所有字母转换为小写。	`SELECT lower('SQL'); -- 'sql'`
`lpad(s, l, p)`	在字符串 `s` 左侧填充字符 `p` 直到总长度达到 `l`。	`SELECT lpad('hi', 4, '?'); -- '??hi'`
`mask(s, U, L, D, O)`	数据脱敏。按类别替换字符： • `U` (upperChar): 大写替换符 (默认'X') • `L` (lowerChar): 小写替换符 (默认'x') • `D` (digitChar): 数字替换符 (默认'n') • `O` (otherChar): 其他字符替换符 (默认NULL, 即保留)	`SELECT mask('Ab-12', 'X', 'x', 'n'); -- 'Xx-nn'`
`overlay(s, r, p, l)`	覆盖替换。从字符串 `s` 的第 `p` 位开始，将长度为 `l` 的内容替换为字符串 `r`。	`SELECT overlay('Spark', '_', 3, 1); -- 'Sp_rk'`

`repeat(s, n)`	将字符串 `s` 重复拼接 `n` 次。	`SELECT repeat('12', 2); -- '1212'`
`replace(str, src, dst)`	将 `str` 中所有的 `src` 子串替换为 `dst` 。	`SELECT replace('ABCabc', 'abc', 'DEF'); -- 'ABCDEF'`
`reverse(s)`	将字符串 `s` 中的字符顺序反转。	`SELECT reverse('ABC'); -- 'CBA'`
`rpad(s, l, p)`	在字符串 `s` 右侧填充字符 `p` 直到总长度达到 `l` 。	`SELECT rpad('hi', 4, '?'); -- 'hi??'`
`space(n)`	返回由 `n` 个空格组成的字符串。	`SELECT space(5); -- '  '`
`translate(s, from, to)`	字符级映射。按位置将 `from` 中的每个字符替换为 `to` 中对应位置的字符。	`SELECT translate('abc', 'ab', '12'); -- '12c'`
`upper(s) / ucase(s)`	将字符串 `s` 中的所有字母转换为大写。	`SELECT upper('sql'); -- 'SQL'`

> ⚠ **注意：**
> `overlay` 是系统保留关键字。在 SQL 脚本中调用该函数时，必须使用反引号将其括起来，例如：`` `overlay`(...) ``。

拼接、正则与哈希

函数	说明	示例
`concat(str1, str2, ...)`	将多个字符串参数拼接为一个字符串。若其中包含 `NULL` 值，该值将被忽略。	`SELECT concat('Hi', space(1), 'SQL'); -- 'Hi SQL'`
`concat_ws(sep[, string]+)`	使用指定的分隔符 `sep` 连接多个字符串。连接过程中会自动跳过 `NULL` 值。	`SELECT concat_ws('-', '2025', NULL, '01'); -- '2025-01'`
`regexp_extract(str, regexp[, idx])`	根据正则表达式 `regexp` 匹配字符串 `str` ，并返回指定捕获组索引 `idx` 对应的子串。	`-- 正则提取：提取第一个连续数字` `SELECT regexp_extract('ID: 100-200', '(\\d+)', 1); -- '100'`

`regexp_replace(str, regexp, rep[, position])`	在字符串 `str` 中搜索所有符合正则表达式 `regexp` 的子串，并将其统一替换为字符串 `rep` 。	`SELECT regexp_replace('100-200', '(\\d+)', 'num'); -- num-num`
`md5(binary)`	计算输入数据的 MD5 消息摘要（128 位）。结果以 32 位十六进制小写字符串形式返回。	`SELECT md5(cast('Spark' as binary)); -- 8cde774d6f7333752ed72cacddb05126`
`sha(binary)` / `sha1(binary)`	计算输入数据的 SHA-1 消息摘要（160 位）。结果以 40 位十六进制小写字符串形式返回。	`SELECT sha(CAST('another_binary' AS BINARY)); -- 798db8f340ba0f832ba7a96e183c2b03f8257b30`
`sha2(binary, bitLength)`	计算 SHA-2 系列哈希值。支持的位数 `bitLength` 包括 224, 256, 384, 512（输入 0 等同于 256）。若指定的位数不支持，则返回 `NULL` 。	`SELECT sha2(CAST('binary_data_sha2' AS BINARY), 256); -- 641573c871686fd3f6da77421329ce81a8f6977465d52eef06fd4379fdcf6b19`

特殊逻辑判断

函数	说明	示例
`contains(left, right)`	判断 `right` 是否为 `left` 的子串。	`SELECT `contains`('Spark SQL', 'Spark'); -- 1` `SELECT `contains`('Spark SQL', 'SPARK'); -- 0`
`startswith(left, right)`	检测 `left` 是否符合前缀匹配规则 `right` 。	`SELECT startswith('js SQL', 'js'); -- 1` `SELECT startswith('js SQL', 'SQL'); -- 0`
`endswith(left, right)`	检测 `left` 是否符合后缀匹配规则 `right` 。	`SELECT endswith('js SQL', 'js'); -- 0` `SELECT endswith('js SQL', 'SQL'); -- 1`
`luhn_check(str)`	基于 Luhn 算法执行校验和检查，用于识别输入错误或无效的标识号码。	`SELECT luhn_check('79927398713'); --1`

日期与时间函数

最近更新时间：2026-05-06 16:28:13

> **说明：**
> 适用版本：TCHouse-X 内核版本 2.0.0及以上版本。

当前系统时间

这些函数用于获取查询执行时刻的系统时间。

函数	说明	示例
`current_date()`	返回当前日期。	`select current_date(); -- 2026-01-01`
`current_timestamp()`	返回当前时间戳（含微秒）。	`select current_timestamp(); -- 2026-01-01 11:30:43.554431`
`current_timezone()`	返回当前会话的本地时区。	`select current_timezone(); -- Asia/Shanghai`
`unix_timestamp()`	以秒为单位返回当前 UNIX 时间戳。	`select unix_timestamp(); -- 1767468503`

日期与时间戳构造

函数	说明	示例
`make_date(y, m, d)`	根据年、月、日整数创建日期。	`select make_date(2024, 12, 31); -- 2024-12-31`
`make_timestamp(y,m,d,h,mi,s[,tz])`	创建带时区的时间戳。秒可含小数，tz 可选。	`select make_timestamp(2024, 12, 28, 6, 30, 45.8); -- 2024-12-28 06:30:45.8`
`make_timestamp_ntz(...)`	创建无时区时间戳（NTZ）。	`select make_timestamp_ntz(2024, 12, 31, 23, 59, 59); -- 2024-12-31 23:59:59`
`date_from_unix_date(n)`	返回 1970-01-01 之后第 n 天的日期。	`select date_from_unix_date(1); -- 1970-01-02`

日期加减与差值

函数	说明	示例
`add_months(d, n)`	加 `n` 月。自动处理月末（如 1-30 加一月为 2-28）。	`select add_months('2015-01-30', 1); -- 2015-02-28`
`date_add(d, n)`	在日期 `d` 上增加 `n` 天。	`select date_add('2024-12-31', 5); -- 2025-01-05`
`date_sub(d, n)`	在日期 `d` 上减去 `n` 天。	`select date_sub('2024-12-31', 5); -- 2024-12-26`
`datediff(end, start)`	计算两个日期之间的天数差。	`select datediff('2024-12-31', '2024-12-30'); -- 1`
`months_between(t1, t2)`	返回两个时间戳间的月数（返回 Double）。	`select months_between('2025-01-18', '2024-12-15'); -- 1.09677419`
`next_day(d, dayOfWeek)`	返回指定日期之后的第一个“星期几”。	`select next_day('2025-07-23', 'Mon'); -- 2025-07-28`

属性提取与截断

类别	函数	说明	示例
字段提取	`year()`，`month()`，`quarter()`，`day()`，`hour()`，`minute()`，`second()`	提取对应的整数部分。	`select hour('2009-07-30 12:58:59'); -- 12`
星期相关	`dayofweek()` / `weekday()`	`dayofweek`：1=Sun, 7=Sat `weekday`：0=Mon, 6=Sun	`select dayofweek('2023-08-22'); -- 3`
年内统计	`dayofyear()` / `week_of_year()`	返回一年中的第几天/第几周。	`select dayofyear('2016-04-09'); -- 100`
灵活提取	`extract(field FROM obj)`	从对象中提取 `YEAR`，`DAY`，`SECOND` 等字段。	`select extract(DAY FROM '2016-04-09'); -- 9`

截断	`date_trunc(fmt, ts)`	将时间戳按单位截断（如截断至月份）。	`select date_trunc('MONTH', '2015-03-05'); -- 2015-03-01 00:00:00`
月末	`last_day(d)`	返回该月最后一天的日期。	`select last_day(DATE '2024-02-15') -- 2024-02-29`

格式化与转换

函数	说明	示例
`to_date(s[, fmt])`	将字符串转为日期。	`select to_date('2016-12-31', 'yyyy-MM-dd'); -- 2016-12-31`
`to_timestamp(s, fmt)`	将字符串转为带时区的时间戳。	`select to_timestamp('2024-12-31', 'yyyy-MM-dd'); -- 2024-12-31 00:00:00`
`date_format(ts, fmt)`	将时间戳按格式转为字符串。	`select date_format(now(), 'yyyy/MM/dd'); -- 2026/01/01`
`from_unixtime(sec, fmt)`	将 UNIX 秒数转为格式化字符串（受时区影响）。	`select from_unixtime(1767239624, 'yyyy-MM-dd');  -- 2026/01/01`
`unix_date(d)`	返回自 1970-01-01 以来的天数。	`select unix_date('1970-01-02'); -- 1`

`unix_seconds/unix_millis/unix_micros`	返回自 Epoch 以来的秒/毫秒/微秒。	`SET TIMEZONE = 'UTC'; -- 为保证样例结果稳定，先设置时区为UTC`
		`select unix_millis('1970-01-01 00:00:01'); -- 1000`
		`select unix_seconds('1970-01-01 00:00:01'); -- 1`
		`select unix_micros('1970-01-01 00:00:01'); -- 1000000`

`from_utc_timestamp/to_utc_timestamp`	**在指定时区与 UTC 之间转换时间戳。**	`SELECT from_utc_timestamp('2015-07-24 07:00:00', 'America/Los_Angeles'); --2015-07-24 00:00:00`

正则表达式函数

最近更新时间：2026-05-06 16:28:13

> **说明：**
> 适用版本：TCHouse-X 内核版本 2.0.0及以上版本。

like

语法

```
like(string, pattern[, escape]) -> boolean
```

功能说明

评估字符串是否与给定的模式（Pattern）匹配。模式可以包含普通字符和通配符。

- **%**：匹配 0 个、1 个或多个字符。
- **_**：匹配单个字符。
- **escape**：可选，指定一个 ASCII 字符用于转义通配符。
- **匹配规则**：区分大小写。

性能说明

- **缓存机制**：每个执行线程最多允许编译 100 个正则表达式。
- **优化策略**：简单模式如 `hello`、`hello%`、`%hello%`、`_hello_` 等将直接评估，不计入正则编译限制。只有复杂模式才会触发正则编译。

示例

```
SELECT like('abc', '%b%');              -- true
SELECT like('a_c', '%#_%', '#');        -- true (使用 # 转义 _)
```

rlike

语法

```
rlike(string, pattern) -> boolean
```

功能说明

基于正则表达式模式进行匹配。

- **包含匹配**：与 `like` 不同，`rlike` 只需模式包含在字符串内即可返回 `true`（无需匹配整个字符串）。

- **锚定：若需完全匹配，请使用 `^` 和 `$` 锚定符。**
- **限制： `pattern` 必须为常量字符串，不支持列引用。**

示例

```
SELECT rlike('1a 2b 14m', '\d+b');      -- true
```

regexp_extract

语法

- `regexp_extract(string, pattern) -> varchar`
- `regexp_extract(string, pattern, group) -> varchar`

功能说明

在字符串中查找正则表达式的第一次出现。

- **无 `group` 参数：返回第一个完整匹配的子串。**
- **带 `group` 参数：返回指定捕获组（Capture Group）编号的内容。**
- **限制： `pattern` 必须为常量字符串，不支持列引用。**

示例

```
SELECT regexp_extract('1a 2b 14m', '\d+');              -- '1'
SELECT regexp_extract('1a 2b 14m', '(\d+)([a-z]+)', 2); -- 'a'
```

regexp_replace

语法

- `regexp_replace(string, pattern, overwrite) -> varchar`
- `regexp_replace(string, pattern, overwrite, position) -> varchar`

功能说明

将字符串中匹配正则表达式的部分替换为 `overwrite` 字符串。

替换逻辑

- **支持捕获组引用：使用 `$g` 或 `${g}` 引用编号组，使用 `${name}` 引用命名组。**
- **转义符：使用 `\$` 表示原义美元符号。若 `\` 后紧跟非数字或非反斜杠字符，则忽略该 `\` 。**

参数说明

- `string`：源字符串。
- `pattern`：正则表达式。
- `overwrite`：替换后的内容。
- `position`：可选，起始搜索位置（从 1 开始）。若 `position` 大于字符串长度，返回原字符串；小于 1 则抛出异常。

限制说明

编译限制：单次函数调用中唯一正则表达式的编译上限为 20 个，超过将抛出异常。

示例

```
-- 基础替换
SELECT regexp_replace('Hello, World!', 'l', 'L');           -- 'HeLLo,
WorLd!'
-- 捕获组替换
SELECT regexp_replace('300-300', '(\\d+)-(\\d+)', '400');  -- '400'
-- 指定起始位置
SELECT regexp_replace('Hello, World!', 'l', 'L', 6);        -- 'Hello,
WorLd!' （从位置6开始替换）
SELECT regexp_replace('Hello, World!', 'l', 'L', 5);        -- 'Hello,
World!' （o之后没有l，不匹配）
```

聚合函数

最近更新时间：2026-05-06 16:28:13

基础统计与数值聚合

此类函数用于对数据集进行最基本的计数、求和及平均值计算。

语法	返回类型	说明
`count(*)`	`BIGINT`	统计总行数，包含 `NULL` 值。
`count(expr1, ...)`	`BIGINT`	统计指定列中非空的行数。
`count(DISTINCT expr, ...)`	`BIGINT`	统计唯一且非空的数值个数（去重计数）。
`sum(x)`	数值类型	计算总和。忽略 NULL。
`avg(x)` / `mean(x)`	`DOUBLE` / `DECIMAL`	计算算术平均值。
`any_value(x)`	同 `x`	随机返回组内一个非空值。

近似计算

适用于大数据场景，通过牺牲少量精度来大幅提升基数统计或分位数计算的性能。

语法	返回类型	说明
`approx_distinct(x[, e])`	`BIGINT`	利用 HyperLogLog++ 估算不同值的数量。`e` 为标准误差（默认 2.3%），取值范围 [0.0040625, 0.26]。
`approx_count_distinct(expr[, relativeSD])`	`BIGINT`	`relativeSD` 定义允许的最大相对标准偏差。
`approx_percentile(x, percentile)`	`DOUBLE`	使用 T-Digest 算法计算近似百分位数。`percentile` 取值 0 ~ 1

median(x)	DOUBLE	返回数值或区间列的中位数。

示例

```
-- 统计海量日志中独立访问用户（UV）的近似数量
WITH user_logs AS (
    SELECT 1 AS user_id UNION ALL SELECT 1 UNION ALL
    SELECT 2 AS user_id UNION ALL SELECT 3 UNION ALL
    SELECT NULL AS user_id
)
SELECT
    -- 1. 默认误差估算（约 2.3%）
    approx_distinct(user_id) AS uv_default,
    -- 2. 指定更小的标准误差（e=0.01 表示 1% 误差）
    approx_distinct(user_id, 0.01) AS uv_precise,
    -- 3. 使用 approx_count_distinct
    approx_count_distinct(user_id) AS uv_count_style
FROM user_logs;

-- 结果:
-- uv_default,uv_precise,uv_count_style
-- 3,3,3


-- 分析用户响应时间，获取中位数、P90（90% 分位数）和 P99 指标，以评估系统性能。
WITH response_times AS (
    SELECT 10 AS latency UNION ALL SELECT 20 AS latency UNION ALL
    SELECT 30 AS latency UNION ALL SELECT 100 AS latency UNION ALL
    SELECT 500 AS latency
)
SELECT
    -- median: 计算 50% 位置的值
    median(latency) AS p50_median,
    -- approx_percentile: 计算 50% 分位数（等同于中位数）
    approx_percentile(latency, 0.5) AS p50_approx,
    -- approx_percentile: 计算 90% 位置的值（P90）
    approx_percentile(latency, 0.9) AS p90_latency,
    -- approx_percentile: 计算 99% 位置的值（P99）
```

```
    approx_percentile(latency, 0.99) AS p99_latency
FROM response_times;

-- 结果：
-- p50_median,p50_approx,p90_latency,p99_latency
-- 30,30,500,500
```

逻辑与位运算聚合

用于对布尔值或整数进行逻辑合并。

语法	入参类型	返回类型	说明
`bool_and(expr)` / `every(expr)`	`BOOLEAN`	`BOOLEAN`	逻辑与聚合，只有当所有值均为 `TRUE` 时返回 `TRUE` 。
`bool_or(expr)`	`BOOLEAN`	`BOOLEAN`	逻辑或聚合，只要有一个值为 `TRUE` 即返回 `TRUE` 。
`bitwise_and_agg(x)`	`TINYINT` / `SMALLINT` / `INTEGER` / `BIGINT`	same as `x`	返回所有非空输入值按补码表示法进行位与 (AND) 运算的结果。
`bitwise_or_agg(x)`	`TINYINT` / `SMALLINT` / `INTEGER` / `BIGINT`	same as `x`	返回所有非空输入值按补码表示法进行位或 (OR) 运算的结果。
`bitwise_xor_agg(x)`	`TINYINT` / `SMALLINT` / `INTEGER` / `BIGINT`	same as `x`	返回所有非空输入值按补码表示法进行位异或 (XOR) 运算的结果。

示例

```
-- 假设有一张订单明细表 order_items，我们需要判断每个订单是否已经完成支付（所有商品都已发货）以及是否存在退款项。
-- 创建临时数据
WITH order_items AS (
    SELECT 101 AS order_id, TRUE AS is_shipped, FALSE AS is_refunded
UNION ALL
    SELECT 101 AS order_id, TRUE AS is_shipped, TRUE AS is_refunded
UNION ALL
```

```
    SELECT 102 AS order_id, TRUE AS is_shipped, FALSE AS is_refunded
UNION ALL
    SELECT 102 AS order_id, FALSE AS is_shipped, FALSE AS is_refunded
)
-- 执行聚合查询
SELECT
    order_id,
    -- 只有当订单下所有商品都已发货时，结果才为 TRUE
    bool_and(is_shipped) AS all_shipped,
    -- 只要订单中有一个商品发生退款，结果即为 TRUE
    bool_or(is_refunded) AS has_refund
FROM order_items
GROUP BY order_id;
-- 结果：
-- order_id,all_shipped,has_refund
-- 101,1,1
-- 102,0,0

-- 创建权限数据
WITH user_roles AS (
    -- 角色 A: Read (1) + Write (2) = 3
    SELECT 'User_A' AS user_name, 3 AS permission_mask UNION ALL
    -- 角色 B: Read (1) + Execute (4) = 5
    SELECT 'User_A' AS user_name, 5 AS permission_mask
)
-- 执行位运算聚合
SELECT
    user_name,
    -- 位或：获取用户拥有的所有权限 (1 | 2 | 4 = 7)
    bitwise_or_agg(permission_mask) AS total_permissions,
    -- 位与：获取用户在所有角色中共同拥有的权限 (3 & 5 = 1)
    bitwise_and_agg(permission_mask) AS common_permissions,
    -- 位异或：获取角色间差异的权限位 (3 ^ 5 = 6)
    bitwise_xor_agg(permission_mask) AS diff_permissions
FROM user_roles
GROUP BY user_name;
-- 结果：
-- user_name,total_permissions,common_permissions,diff_permissions
-- User_A,7,1,6
```

极值与位置聚合

语法	支持入参类型	返回类型	说明
`min(x)`	`x` 必须是可排序类型	same as `x`	–
`max(x)`	`x` 必须是可排序类型	same as `x`	–
`first(x)`	`String`	same as x	返回其在组内遇到的第一个值。
`last(x)`	`String`	same as x	返回其在组内遇到的最后一个值。
`first_ignore_null(x)`	`T`	same as x	返回其在组内遇到的第一个非空值，如果所有值均为 `NULL`，则返回 `NULL`。
`last_ignore_null(x)`	`T`	same as `x`	返回其在组内遇到的第一个非空值。如果所有值均为 `NULL`，则返回 `NULL`。

```
-- 假设我们有一台设备，在不同时间点上报了温度（temperature）和固件版本（version）。
其中版本号并不是每次都上报（存在 NULL）。
WITH sensor_readings AS (
    -- 模拟数据：ID, 时间, 温度, 固件版本
    SELECT 'Sensor_01' AS device_id, '2026-01-01 08:00' AS ts, 20.5 AS
temp, 'v1.0' AS ver UNION ALL
    SELECT 'Sensor_01' AS device_id, '2026-01-01 08:15' AS ts, 22.1 AS
temp, NULL AS ver   UNION ALL
    SELECT 'Sensor_01' AS device_id, '2026-01-01 08:30' AS ts, 19.8 AS
temp, 'v1.1' AS ver UNION ALL
    SELECT 'Sensor_01' AS device_id, '2026-01-01 08:45' AS ts, 21.0 AS
temp, NULL AS ver
)
SELECT
    device_id,
    -- 1. 极值聚合
    min(temp) AS min_temp,              -- 最低温度
    max(temp) AS max_temp,              -- 最高温度

    -- 2. 位置聚合（包含NULL）
```

```
    `first`(ver) AS first_ver_raw,        -- 遇到的第一个值（v1.0）
    `last`(ver) AS last_ver_raw,          -- 遇到的最后一个值（NULL）

    -- 3. 忽略空值的位置聚合
    first_ignore_null(ver) AS first_valid_ver, -- 第一个非空版本（v1.0）
    last_ignore_null(ver) AS latest_ver        -- 最新有效版本（v1.1）
FROM sensor_readings
GROUP BY device_id;
-- 结果：
--
device_id,min_temp,max_temp,first_ver_raw,last_ver_raw,first_valid_ver,l
atest_ver
-- Sensor_01,19.8,22.1,v1.0,NULL,v1.0,v1.1
```

统计学与线性回归

语法	入参类型	返回类型	说明
`std(x)` / `stddev(x)` / `stddev_samp(x)`	`TINYINT` / `SMALLINT` / `INT` / `BIGINT` / `FLOAT` / `DOUBLE` / `DECIMAL`	`DOUBLE`	返回根据一组值计算出的样本标准差。
`stddev_pop(x)`	`TINYINT` / `SMALLINT` / `INT` / `BIGINT` / `FLOAT` / `DOUBLE` / `DECIMAL`	`DOUBLE`	总体标准差计算。
`variance(x)` / `var_samp(x)`	`TINYINT` / `SMALLINT` / `INT` / `BIGINT` / `FLOAT` / `DOUBLE` / `DECIMAL`	`DOUBLE`	返回所有输入值计算出的样本方差。x 的类型应为 DOUBLE。当 x 的数量大于或等于 2 时，将生成非空输出。
`var_pop(x)`	`TINYINT` / `SMALLINT` / `INT` / `BIGINT` / `FLOAT` / `DOUBLE` / `DECIMAL`	`DOUBLE`	总体方差计算。
`skewness(x)`	`TINYINT` / `SMALLINT` / `INT` / `BIGINT` / `FLOAT` / `DOUBLE` / `DECIMAL`	`DOUBLE`	返回所有输入值的偏度。当 x 的计数大于或等于 1 时，将生成非空输出。当累加器中 m2 的值为 0 时，将生成空输出。

kurtosis(x)	TINYINT / SMALLINT / INT / BIGINT / FLOAT / DOUBLE / DECIMAL	DOUBLE	返回所有输入值的偏度。当 x 的计数大于或等于 1 时，将生成非空输出。当累加器中 m2 的值为 0 时，将生成空输出。

```
-- 构建一个模拟的生产线零件重量数据集，通过这些统计函数来评估生产工艺的稳定性。
WITH production_data AS (
    -- 模拟 5 个零件的重量（单位：克）
    SELECT 'Line_A' AS line_id, 10.2 AS weight UNION ALL
    SELECT 'Line_A' AS line_id, 9.8 AS weight  UNION ALL
    SELECT 'Line_A' AS line_id, 10.0 AS weight UNION ALL
    SELECT 'Line_A' AS line_id, 10.5 AS weight UNION ALL
    SELECT 'Line_A' AS line_id, 9.5 AS weight
)
SELECT
    line_id,
    -- 1. 标准差：衡量数据偏离平均值的程度
    stddev_samp(weight) AS sample_std,     -- 样本标准差 （常用）
    stddev_pop(weight) AS pop_std,         -- 总体标准差

    -- 2. 方差：标准差的平方
    var_samp(weight) AS sample_variance,   -- 样本方差
    var_pop(weight) AS pop_variance,       -- 总体方差

    -- 3. 分布形态
    skewness(weight) AS data_skew,         -- 偏度：反映分布的对称性
    kurtosis(weight) AS data_kurt          -- 峰度：反映分布的陡峭程度/尾部厚度
FROM production_data
GROUP BY line_id;
-- 结果：
--
line_id,sample_std,pop_std,sample_variance,pop_variance,data_skew,data_k
urt
--
Line_A,0.38078865529319522,0.34058768251719912,0.14499999999999985,0.115
99996948243643,-5.2689553466917849e-17,-1.0945303210463697
```

窗口函数

最近更新时间：2026-05-06 16:28:13

> **说明：**
> 适用版本：TCHouse-X 内核版本 2.0.0及以上版本。

窗口函数在与当前行相关的行集合（即“窗口”）上执行计算。与聚合函数不同，窗口函数不会将多行压缩为一行，而是为每一行保留其原始身份并附带计算结果。

核心语法

所有窗口函数都必须配合 `OVER` 子句使用：

```
function(args) OVER (
      [PARTITION BY expression]
      [ORDER BY expression [ASC|DESC]]
      [frame]
)
```

关键概念

- `PARTITION BY`：将数据集划分为多个分区。计算在每个分区内独立进行。若省略，则整个结果集视为一个分区。
- `ORDER BY`：定义分区内行的排列顺序。这对于排名函数和涉及物理范围（如累加）的计算至关重要。
- `frame`：指定函数在处理特定输入行时所涵盖的滑动行窗口。
  - 范围可以是 `ROWS`（基于行数）或 `RANGE`（基于逻辑值范围），从 `frame_start` 延伸到 `frame_end`。
  - 如果未指定 `frame_end`，则默认使用 `CURRENT ROW`（当前行）。

常用窗口函数

排名与编号函数

函数	排序方式	特点	示例结果 (1, 1, 2)
`row_number()`	唯一编号	即使值相同，编号也不重复。	1, 2, 3
`rank()`	跳跃排名	值相同时排名并列，但会“跳过”后续编号。	1, 1, 3

`dense_rank()`	连续排名	值相同时排名并列，且编号保持连续。	1, 1, 2
`ntile(n)`	切片/分桶	将数据均匀分成 `n` 份，返回所属桶号。	1, 1, 2 (n=2)

```
WITH sales_data AS (
    SELECT 'Alice' AS name, 'Sales' AS dept, 5000 AS amount UNION ALL
    SELECT 'Bob'   AS name, 'Sales' AS dept, 5000 AS amount UNION ALL
    SELECT 'Chris' AS name, 'Sales' AS dept, 4000 AS amount UNION ALL
    SELECT 'David' AS name, 'Sales' AS dept, 3000 AS amount UNION ALL
    SELECT 'Eve'   AS name, 'Sales' AS dept, 2000 AS amount
)
SELECT
    name,
    amount,
    row_number() OVER (ORDER BY amount DESC) AS row_num,
    rank()       OVER (ORDER BY amount DESC) AS rk,
    dense_rank() OVER (ORDER BY amount DESC) AS d_rk,
    ntile(2)     OVER (ORDER BY amount DESC) AS bucket
FROM sales_data;

-- 结果:
-- name,amount,row_num,rk,d_rk,bucket
-- Alice,5000,1,1,1,1
-- Bob,5000,2,1,1,1
-- Chris,4000,3,3,2,1
-- David,3000,4,3,2
-- Eve,2000,5,5,4,2
```

偏移与取值函数

这类函数用于跨行获取数据，常用于计算同比、环比或寻找临界点。

函数	功能描述	核心参数
lag(x, o, d)	向下偏移。获取当前行之前的第 `o` 个值。	`o`：偏移量，`d`：默认值
lead(x, o, d)	向上偏移。获取当前行之后的第 `o` 个值。	`o`：偏移量，`d`：默认值

first_value(x)	返回窗口内的第一个值。	常配合 `ORDER BY` 使用
last_value(x)	返回窗口内的最后一个值。	常配合 `ORDER BY` 使用
nth_value(x, n)	返回窗口内第 `n` 个指定值。	常配合 `ORDER BY` 使用

```
-- 观察用户每日登录次数的变化（环比增长）。
WITH daily_logins AS (
    SELECT '2026-01-01' AS `date`, 100 AS cnt UNION ALL
    SELECT '2026-01-02' AS `date`, 120 AS cnt UNION ALL
    SELECT '2026-01-03' AS `date`, 110 AS cnt UNION ALL
    SELECT '2026-01-04' AS `date`, 150 AS cnt
)
SELECT
    `date`,
    cnt AS "当日登录数",
    -- lag(x, o, d): 获取前 1 天的次数，若无则显示 0
    lag(cnt, 1, 0) OVER (ORDER BY `date`) AS "前日登录数",
    -- lead(x, o, d): 获取后 1 天的次数，若无则显示 0
    lead(cnt, 1, 0) OVER (ORDER BY `date`) AS "次日登录数"
FROM daily_logins;

-- 结果:
-- date,当日登录数,前日登录数,次日登录数
-- 2026-01-01,100,0,120
-- 2026-01-02,120,100,110
-- 2026-01-03,110,120,150
-- 2026-01-04,150,110,0

-- 订单列表中，找出每个用户的“首次下单金额”和“最近一次下单金额”。
WITH orders AS (
    SELECT 'User_A' AS user_id, '2026-01-01' AS order_date, 100 AS price
UNION ALL
    SELECT 'User_A' AS user_id, '2026-01-05' AS order_date, 250 AS price
UNION ALL
    SELECT 'User_A' AS user_id, '2026-01-10' AS order_date, 300 AS price
)
SELECT
    user_id,
```

```
    order_date,
    price,
    -- 获取该用户的第一笔订单金额
    first_value(price) OVER (PARTITION BY user_id ORDER BY order_date)
AS "首单金额",
    -- 获取该用户的最新一笔订单金额
    last_value(price) OVER (
        PARTITION BY user_id
        ORDER BY order_date
        ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING
    ) AS "终单金额"
FROM orders;

-- 结果:
-- user_id,order_date,price,首单金额,终单金额
-- User_A,2026-01-01,100,100,300
-- User_A,2026-01-05,250,100,300
-- User_A,2026-01-10,300,100,300


-- 获取每个部门薪资排名第二的员工。
WITH salary_data AS (
    SELECT 'Tech' AS dept, 'Alice' AS name, 15000 AS salary UNION ALL
    SELECT 'Tech' AS dept, 'Bob' AS name, 12000 AS salary UNION ALL
    SELECT 'Tech' AS dept, 'Chris' AS name, 10000 AS salary
)
SELECT
    dept,
    name,
    salary,
    -- nth_value(x, n): 获取按薪资降序排列后的第 2 名
    nth_value(name, 2) OVER (
        PARTITION BY dept
        ORDER BY salary DESC
        ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING
    ) AS "部门榜眼"
FROM salary_data;

-- 结果:
-- dept,name,salary,部门榜眼
```

```
-- Tech,Alice,15000,Bob
-- Tech,Bob,12000,Bob
-- Tech,Chris,10000,Bob
```

分布与统计函数

这类函数用于分析数据在整体中的位置百分比。

函数	计算公式	典型用途
percent_rank()	(rank - 1) / (总行数 - 1)	计算当前行在全集中的相对百分比排名。
cume_dist()	小于等于当前值的行数 / 总行数	计算累积分布，常用于帕累托分析。

```
WITH score_data AS (
    SELECT 'Alice' AS name, 95 AS score UNION ALL
    SELECT 'Bob'   AS name, 95 AS score UNION ALL
    SELECT 'Chris' AS name, 80 AS score UNION ALL
    SELECT 'David' AS name, 60 AS score
)
SELECT
    name,
    score,
    -- percent_rank(): (rank - 1) / (total_rows - 1)
    percent_rank() OVER (ORDER BY score DESC) AS p_rank,
    -- cume_dist(): 小于等于当前值的行数 / 总行数
    cume_dist() OVER (ORDER BY score DESC) AS c_dist
FROM score_data;

-- 结果:
-- name,score,p_rank,c_dist
-- Alice,95,0,0.5
-- Bob,95,0,0.5
-- Chris,80,0.6666666666666663,0.75
-- David,60,1,1
```

URL 函数

最近更新时间：2026-05-06 16:28:13

> **说明：**
> 适用版本：TCHouse-X 内核版本 2.0.0及以上版本。

`url_encode`

语法

```
url_encode(value) -> varchar
```

功能说明

对字符串进行转义编码，确保其能够安全地作为 URL 查询参数（Query Parameter）的名称或值。

编码规则

- 不编码字符：字母数字字符（A-Z, a-z, 0-9）以及部分特殊字符（.、-、*、_）。
- 空格处理：ASCII 空格字符编码为 +。
- UTF-8 转换：所有其他字符转换为 UTF-8 编码，每个字节表示为 %XX（其中 XX 是该字节的大写十六进制值）。

示例

```
SELECT url_encode('https://spark.apache.org'); -- 输出:
https%3A%2F%2Fspark.apache.org
```

url_decode

语法

```
url_decode(value) -> varchar
```

功能说明

对经过 URL 编码的字符串进行解码，还原为原始字符串。此函数是 url_encode() 的逆运算。

示例

```
SELECT url_decode('https%3A%2F%2Fspark.apache.org');  -- 输出:
```

```
'https://spark.apache.org
```

关键字

最近更新时间：2026-05-06 16:28:13

关键字在 TCHouse-X 中具有特殊含义，不能直接用作库名、表名、列名或其他标识符。

标识符引用规则

如果您必须使用保留字作为标识符，必须使用反引号（ ` ）将其括起来。

示例

```
CREATE TABLE select (x INT);     -- 错误：select 是保留字
CREATE TABLE `select` (x INT);   -- 正确：使用了反引号
```

关键字列表

ADD	AGGREGATE	ALL
ALLOCATE	ALTER	ANALYTIC
AND	ANTI	ANY
API_VERSION	ARE	ARRAY
ARRAY_AGG	ARRAY_MAX_CARDINALITY	AS
ASC	ASENSITIVE	ASYMMETRIC
AT	ATOMIC	AUTHORIZATION
AUTO	AVRO	BEGIN_FRAME
BEGIN_PARTITION	BETWEEN	BIGINT
BINARY	BLOB	BLOCK_SIZE
BOOLEAN	BOTH	BUCKETS
BY	CACHED	CACHEMETRICS
CALLED	CAMUSER	CARDINALITY
CASCADE	CASCADED	CASE

CAST	CHANGE	CHAR
CHARACTER	CHARSET	CLASS
CLOB	CLOSE_FN	CLUSTERED
COLLATE	COLLATION	COLLECT
COLUMN	COLUMNS	COMMENT
COMMIT	COMPACT	COMPRESSION
COMPUTE	CONDITION	CONNECT
CONSTRAINT	CONTAINS	CONVERT
COPY	CORRESPONDING	CREATE
CROSS	CUBE	CURRENT
CURRENT_DEFAULT_TRANSFORM_GROUP	CURRENT_PATH	CURRENT_ROLE
CURRENT_ROW	CURRENT_SCHEMA	CURRENT_TIME
CURRENT_TRANSFORM_GROUP_FOR_TYPE	CURSOR	CYCLE
DATA	DATABASE	DATABASES
DATE	DATETIME	DBPROPERTIES
DEALLOCATE	DEC	DECFLOAT
DECIMAL	DECLARE	DEFAULT
DEFINE	DELETE	DELIMITED
DEREF	DESC	DESCRIBE
DETERMINISTIC	DISABLE	DISCONNECT
DISTINCT	DIV	DOUBLE
DROP	DYNAMIC	EACH

ELEMENT	ELSE	EMPTY
ENABLE	ENCODING	END
END_FRAME	END_PARTITION	ENFORCED
ENGINES	EQUALS	ESCAPE
ESCAPED	EVERY	EXCEPT
EXEC	EXECUTE	EXISTS
EXPLAIN	EXTENDED	EXTERNAL
FALSE	FAST	FETCH
FIELDS	FILEFORMAT	FILES
FILE_SIZE_THRESHOLD_MB	FILTER	FINALIZE_FN
FIRST	FLOAT	FN
FOLLOWING	FOR	FOREIGN
FORMAT	FORMATTED	FRAME_ROW
FREE	FROM	FULL
FUNCTION	FUNCTIONS	FUSION
GET	GLOBAL	GRANT
GROUP	GROUPING	GROUPS
HASH	HAVING	HOLD
HORN_CTE_MATERIALIZE	HORN_DPHYPER_ENABLE	HUDIPARQUET
ICEBERG	IDENTIFIED	IF
IGNORE	ILIKE	IN
INCREMENTAL	INDEX	INDEXES
INDICATOR	INITIAL	INIT_FN

INNER	INOUT	INPATH
INSENSITIVE	INSERT	INT
INTEGER	INTERMEDIATE	INTERSECT
INTERSECTION	INTERVAL	INTO
INVALIDATE	IREGEXP	IS
JOIN	JSONFILE	JSON_ARRAY
JSON_ARRAYAGG	JSON_EXISTS	JSON_OBJECT
JSON_OBJECTAGG	JSON_QUERY	JSON_TABLE
JSON_TABLE_PRIMITIVE	JSON_VALUE	KEYS
KILL	KUDU	LARGE
LAST	LATERAL	LEADING
LEFT	LEXICAL	LIKE
LIKE_REGEX	LIMIT	LINES
LISTAGG	LOAD	LOCAL
LOCALTIMESTAMP	LOCATION	MANAGEDLOCATION
MAP	MATCH	MATCHED
MATCHES	MATCH_NUMBER	MATCH_RECOGNIZE
MERGE	MERGE_FN	METADATA
METHOD	MINUS	MODEL
MODELS	MODEL_OPTIONS	MODIFIES
MULTISET	MYSQL	NAMES
NATIONAL	NATURAL	NCHAR
NCLOB	NO	NONE

NORELY	NORMALIZE	NOT
NOVALIDATE	NULL	NULLS
NUMERIC	OCCURRENCES_REGEX	OCTET_LENGTH
OF	OFFSET	OMIT
ON	ONE	ONLY
OPTIMIZE	OR	ORC
ORDER	OUT	OUTER
OVER	OVERLAPS	OVERLAY
OVERWRITE	PARQUET	PARQUETFILE
PARTITION	PARTITIONED	PARTITIONS
PATTERN	PER	PERCENT
PERCENTILE_CONT	PERCENTILE_DISC	PLUGINS
PORTION	POSITION	POSITION_REGEX
POSTGRESQL	PRECEDES	PRECEDING
PREPARE	PREPARE_FN	PRIMARY
PROCEDURE	PRODUCED	PROPERTIES
PTF	PURGE	QUERY
RANGE	RCFILE	READS
REAL	RECOVER	RECURSIVE
REF	REFERENCES	REFERENCING
REFRESH	REGEXP	REGR_AVGX
REGR_AVGY	REGR_COUNT	REGR_INTERCEPT
REGR_R2	REGR_SLOPE	REGR_SXX

REGR_SXY	REGR_SYY	RELEASE
RELY	RENAME	REPEATABLE
REPLACE	REPLICATION	RESOURCE
RESOURCES	RESTRICT	RETURNS
REVOKE	RIGHT	RLIKE
ROLE	ROLES	ROLLUP
ROW	ROWS	RUNNING
RWSTORAGE	SAVEPOINT	SCHEMA
SCHEMAS	SCOPE	SCROLL
SEARCH	SEEK	SELECT
SEMI	SENSITIVE	SEQUENCEFILE
SERDEPROPERTIES	SERIALIZE_FN	SESSION
SET	SETS	SHOW
SIMILAR	SKIP	SMALLINT
SOME	SORT	SPARK
SPEC	SPECIFIC	SPECIFICTYPE
SQLEXCEPTION	SQLSTATE	SQLWARNING
STATIC	STATS	STATUS
STORAGEHANDLER_URI	STORED	STRAIGHT_JOIN
STREAM	STRING	STRUCT
SUBMULTISET	SUBSET	SUBSTRING_REGEX
SUCCEEDS	SYMBOL	SYMMETRIC
SYNC	SYSTEM_TIME	SYSTEM_VERSION

TABLE	TABLES	TABLESAMPLE
TBLPROPERTIES	TCI	TERMINATED
TEXTFILE	THEN	TIMESTAMP
TIMESTAMPNTZ	TIMESTAMPTZ	TIMESTAMP_NTZ
TIMEZONE_HOUR	TIMEZONE_MINUTE	TINYINT
TO	TRAILING	TRANSLATE_REGEX
TRANSLATION	TREAT	TRIGGER
TRIM_ARRAY	TRUE	TRUNCATE
UESCAPE	UNBOUNDED	UNCACHED
UNION	UNIQUE	UNKNOWN
UNLOCK	UNNEST	UNSET
UPDATE	UPDATE_FN	UPSERT
USE	USERS	USING
VALIDATE	VALUES	VALUE_OF
VARBINARY	VARCHAR	VARIABLES
VARYING	VERSIONING	VIEW
VIEWS	VIRTUAL_WAREHOUSE	WARMUP
WARNINGS	WHEN	WHENEVER
WHERE	WINDOW	WITH
WITHIN	WITHOUT	ZORDER

内表和外表说明

最近更新时间：2026-05-06 16:28:13

TCHouse-X 的数据表分为 内表 (Internal Table) 和外表 (External Table)。两者主要区别在于数据存储位置、数据管理权限和支持的 DML 操作。
核心特性对比如下：

特性	内表 (Internal Table)	外表 (External Table)
数据存储	TCHouse-X 托管存储（数据与元数据统一管理）	用户自管的 COS 存储桶、HDFS（数据与元数据分离）
DDL 语法	`CREATE TABLE`	`CREATE EXTERNAL TABLE`
存储格式	TCHouse-X 自研结构，系统自动决定	建表时可指定（如 `STORED AS TEXTFILE`）
核心优势	更高的读写性能，支持行级修改	零 ETL，可直接读取外部数据资产
DML 支持	完整支持: `INSERT`，`LOAD`，`UPDATE`，`DELETE`，`TRUNCATE`	有限支持: 仅支持 `INSERT`，`LOAD`，`TRUNCATE`。不支持 `UPDATE` / `DELETE`。
文件优化	支持系统自动合并文件 (COMPACTION)	不支持 COMPACTION，写入易产生小文件

详细说明与示例

内表

内表是 TCHouse-X 自研的表结构，为追求极致的读写性能而设计。

- 性能: 具有更高的读写性能和查询效率。
- 存储: 数据仅存储在 TCHouse-X 的托管存储中，用户不需手动维护数据文件。
- DDL 约束: 建表时使用 CREATE TABLE，不支持指定 LOCATION 和底层存储格式。
- DML 支持: 支持完整的 DML 操作，包括行级的 UPDATE 和 DELETE。

示例：创建内表

```
CREATE TABLE default.dt1 (
    id INT,
    name VARCHAR(50)
```

```
);
```

外表

外表用于连接 TCHouse-X 外部已有的数据源，实现分析计算。

- 存储: 数据存储在用户自管的 COS 桶或 HDFS 中。
- DDL 约束: 必须使用 CREATE EXTERNAL TABLE 语法，且必须通过 LOCATION 参数指定数据文件的路径。
- 读取格式: 支持对 Parquet、Textfile、ORC、JSON 等主流文件进行读操作。
- DML 限制: 不支持行级的数据修改 (UPDATE, DELETE)。
- 文件问题: 不支持文件合并 (COMPACTION) 特性，写入操作会创建多个小文件。

示例：创建外表

```
CREATE EXTERNAL TABLE default.dt1 (
    id INT,
    name VARCHAR(50)
)
STORED AS TEXTFILE
LOCATION 'cosn://bucket-name/path';
```