# 腾讯云数据仓库 TCHouse-P

# 开发指南

腾讯云

# 文档目录

# 开发指南
# 常量与宏

最近更新时间：2021-03-02 17:48:35

| 参数 | 功能 | 相关 SQL 语句 |
| --- | --- | --- |
| CURRENT_CATALOG | 当前数据库 | SELECT CURRENT_CATALOG; |
| CURRENT_ROLE | 当前用户 | SELECT CURRENT_ROLE; |
| CURRENT_SCHEMA | 当前数据库模式 | SELECT CURRENT_SCHEMA; |
| CURRENT_USER | 当前用户 | SELECT CURRENT_USER; |
| LOCALTIMESTAMP | 当前时间 | SELECT LOCALTIMESTAMP; |
| SESSION_USER | 当前系统用户 | SELECT SESSION_USER; |
| USER | 当前用户，与 CURRENT_USER 一致 | SELECT USER; |

# 函数和操作符
# 逻辑操作符

最近更新时间：2021-03-02 17:49:09

常用的逻辑操作符有 AND、OR、NOT，有三个结果值 TRUE、FALSE、NULL。其中 NULL 代表未知值。

| a | b | a AND b | a OR b | NOT a |
|---|---|---|---|---|
| TRUE | TRUE | TRUE | TRUE | FALSE |
| TRUE | FALSE | FALSE | TRUE | FALSE |
| TRUE | NULL | NULL | TRUE | FALSE |
| FALSE | FALSE | FALSE | FALSE | TRUE |
| FALSE | NULL | FALSE | NULL | TRUE |
| NULL | NULL | NULL | NULL | NULL |

# 比较操作符

最近更新时间：2021-03-02 17:49:18

**常用的比较操作符和功能见下表：**

| 操作符 | 功能 |
| --- | --- |
| < | 小于 |
| > | 大于 |
| <= | 小于或等于 |
| >= | 大于或等于 |
| = | 等于 |
| <> 或 != | 不等于 |

# 字符串函数和操作符

最近更新时间：2024-08-15 16:19:42

**这一部分主要是讲字符串的一些操作函数，具体见下表表格。**

| 函数 | 返回值 | 描述 | 示例 | 结果 |
| --- | --- | --- | --- | --- |
| string \|\| string | text | 字符串连接 | 'Post' \|\| 'greSQL' | PostgreSQL |
| string \|\| non-string or non-string \|\| string | text | 字符串和非字符串之间的连接 | 'Value: ' \|\| 42 | Value: 42 |
| bit_length(string) | int | 字符串的 bit 数 | bit_length('jose') | 32 |
| char_length(string) or character_length(string) | int | 字符串的字符数 | char_length('jose') | 4 |
| lower(string) | text | 转为小写字符 | lower('TOM') | tom |
| octet_length(string) | int | 字符串的字节数 | octet_length('jose') | 4 |
| overlay(string placing string from int [for int]) | text | 替换子字符串 | overlay('Txxxxas' placing 'hom' from 2 for 4) | Thomas |
| position(substring in string) | int | 子字符串在字符串中的位置 | position('om' in 'Thomas') | 3 |
| substring(string [from int] [for int]) | text | 截取子字符串 | substring('Thomas' from 2 for 3) | hom |
| substring(string from pattern) | text | 正则匹配子字符串 | substring('Thomas' from '...$') | mas |
| trim([leading \| trailing \| both] [characters] fromstring) | text | 移除子字符串 | trim(both 'x' from 'xTomxx') | Tom |

| | | | | |
|---|---|---|---|---|
| upper(string) | text | 转为大写字母 | upper('tom') | TOM |

数据库还提供一些其他的字符串函数，用于支持对字符串的处理，见下面的表格。

| 函数 | 返回值 | 功能 | 示例 | 结果 |
|---|---|---|---|---|
| ascii(string) | int | 获取第一个字符的 ASCII 码 | ascii('x') | 120 |
| btrim(string text [, characters text]) | text | 从 string 开头和结尾（而非中间）删除只包含在 characters 里（缺省是空白）的字符的最长字符串 | btrim('xyxtrimyyx', 'xy') | trim |
| chr(int) | text | ASCII 码转字符 | chr(65) | A |
| convert(string bytea, src_encodingname, dest_encodingname) | bytea | 使用指定的转换名字改变编码 | convert('text_in_utf8', 'UTF8', 'LATIN1') | – |
| convert_from(string bytea,src_encoding name) | text | 转换为数据库默认的编码格式 | convert_from('text_in_utf8', 'UTF8') | text_in_utf8represented in the current database encoding |
| convert_to(string text, dest_encodingname) | bytea | 转换为编码为某种格式的数据 | convert_to('some text', 'UTF8') | – |
| decode(string text, type text) | bytea | 解码函数 | decode('MTIzAAE=', 'base64') | 123\000\001 |
| encode(data bytea, type text) | text | 编码函数 | encode(E'123\000\001', 'base64') | MTIzAAE= |

| | | | | |
|---|---|---|---|---|
| initcap(string) | text | 每个单词的第一个字母大小，其余小写 | initcap('hi THOMAS') | Hi Thomas |
| length(string) | int | 字符串长度 | length('jose') | 4 |
| length(stringbytea, encoding name ) | int | 指定编码格式的字符串长度 | length('jose', 'UTF8') | 4 |
| ltrim(string text [, characters text]) | text | 从 string 开头（而非中间）删除只包含在 characters 里（缺省是空白）的字符的最长字符串 | ltrim('zzzytrim', 'xyz') | trim |
| md5(string) | text | 计算字符串的 MD5 值 | md5('abc') | 900150983cd24fb0d6963f7d28e17f72 |
| pg_client_encoding() | name | 获取客户端的编码格式 | pg_client_encoding() | SQL_ASCII |
| regexp_replace(string text, patterntext, replacement text [, flags text]) | text | – | regexp_replace('Thomas', '.[mN]a.', 'M') | ThM |
| repeat(string text, n int) | text | 重复 n 次字符串 | repeat('Pg', 4) | PgPgPgPg |
| replace(string text, from text, totext) | text | 替换字符串 | replace('abcdefabcdef', 'cd', 'XX') | abXXefabXXef |
| rtrim(string text [, characters text]) | text | 从 string 结尾（而非中间）删除只包含在 characters 里（缺省是空白）的字符的最长字符串 | rtrim('trimxxxx', 'x') | trim |
| strpos(string, substring) | int | 获取子串的位置 | strpos('high', 'ig') | 2 |

| | | | | |
|---|---|---|---|---|
| substr(string, from [, count]) | text | 获取子串 | substr('alphabet', 3, 2) | ph |
| to_ascii(string text [, encodingtext]) | text | 转为 ASCII 码格式 | to_ascii('Karel') | Karel |
| to_hex(number int or bigint) | text | 将数字转为16进制 | to_hex(2147483647) | 7fffffff |

# 模式匹配

最近更新时间：2022-07-04 15:04:11

## LIKE

数据库支持多种模式匹配方式。第一种就是 LIKE 或者 NOT LIKE。

| 示例 | 结果 |
| --- | --- |
| ‘abc’ LIKE ‘abc’ | true |
| ‘abc’ LIKE ‘abcz’ | false |
| ‘abc’ LIKE ‘a%’ | true |
| ‘abc’ LIKE ‘_a_’ | false |
| ‘abc’ LIKE ‘a_’ | false |
| ‘abc’ LIKE ‘a’ | false |

## SIMILAR TO

第二种模式匹配是数据库支持的 SIMILAR TO，使用方式和 LIKE 相似，不过支持更多的匹配语法，具体见下表格。

| 语法 | 描述 | 示例 | 结果 |
| --- | --- | --- | --- |
| \| | 支持可选的匹配 | 'abc' SIMILAR TO '%(b\|d\|c\|e)%' | True |
| * | 前一个元素0或者多次重复 | 'abc' SIMILAR TO 'abcd*' | True |
| 'abc' SIMILAR TO 'abc*' | true | – | – |

| | | | |
|---|---|---|---|
| 'abc' SIMILAR TO 'ab*' | false | - | - |
| + | 前一个元素1次或者多次重复 | 'abc' SIMILAR TO 'abc+' | True |
| 'abc' SIMILAR TO 'abcc+' | false | - | - |

## POSIX 正则表达式

数据库还支持 POSIX 正则表达式，它提供了比 LIKE 和 SIMILAR TO 操作符更强大的函数。POSIX 正则表达式支持的函数如下表格。

| 函数 | 功能 | 示例 | 结果 |
|---|---|---|---|
| substring(string from pattern) | 从 string 中按照正则 pattern 截取字符串 | substring('foobar' from 'o.b') | oob |
| regexp_replace(source, pattern, replacement [, flags ]) | 替换 source 中匹配的字符串 | regexp_replace('foobarbaz', 'b..', 'X') | fooXX |
| regexp_matches(string, pattern [, flags ]) | 函数返回一个文本数组，该数组由匹配一个 POSIX 正则表达式模式得到的所有被捕获子串构成 | regexp_matches('foobarbequebaz', '(bar)(beque)'); | {bar,beque} |
| regexp_split_to_table(string, pattern [, flags ]) | 函数把一个 POSIX 正则表达式模式当作一个定界符来分离一个串 | SELECT regexp_split_to_table('the quick brown fox jumped', E'\\s+') ; | the<br>quick<br>brown<br>fox<br>jumped |
| regexp_split_to_array(string, pattern [, flags ]) | 和 regexp_split_to_table 类似，是一个正则表达式分离函数，不过它的结果以一个 text 数组的形式返回 | SELECT regexp_split_to_array('the quick brown fox jumped', E'\\s+'); | {the,quick,brown,fox,jumped} |

# 日期时间函数和操作符

最近更新时间：2024-08-15 16:19:42

**下表展示了数据库支持的常用日期和时间函数以及操作符。**

| 操作符 | 示例 | 结果 |
| --- | --- | --- |
| + | date '2001-09-28' + integer '7' | date '2001-10-05' |
| + | date '2001-09-28' + interval '1 hour' | timestamp '2001-09-28 01:00:00' |
| + | date '2001-09-28' + time '03:00' | timestamp '2001-09-28 03:00:00' |
| + | interval '1 day' + interval '1 hour' | interval '1 day 01:00:00' |
| + | timestamp '2001-09-28 01:00' + interval '23 hours' | timestamp '2001-09-29 00:00:00' |
| + | time '01:00' + interval '3 hours' | time '04:00:00' |
| - | - interval '23 hours' | interval '-23:00:00' |
| - | date '2001-10-01' - date '2001-09-28' | integer '3' |
| - | date '2001-10-01' - integer '7' | date '2001-09-24' |
| - | date '2001-09-28' - interval '1 hour' | timestamp '2001-09-27 23:00:00' |
| - | time '05:00' - time '03:00' | interval '02:00:00' |
| - | time '05:00' - interval '2 hours' | time '03:00:00' |
| - | timestamp '2001-09-28 23:00' - interval '23 hours' | timestamp '2001-09-28 00:00:00' |
| - | interval '1 day' - interval '1 hour' | interval '1 day -01:00:00' |
| - | timestamp '2001-09-29 03:00' - timestamp '2001-09-27 12:00' | interval '1 day 15:00:00' |
| * | 900 * interval '1 second' | interval '00:15:00' |

| | | |
|---|---|---|
| * | 21 * interval '1 day' | interval '21 days' |
| * | double precision '3.5' * interval '1 hour' | interval '03:30:00' |
| / | interval '1 hour' / double precision '1.5' | interval '00:40:00' |

**下面的表格是常用的时间/日期函数。**

| 函数 | 返回值 | 描述 | 示例 | 结果 |
|---|---|---|---|---|
| age(timestamp, timestamp) | interval | 计算时间差 | age(timestamp '2001-04-10', timestamp '1957-06-13') | 43 years 9 mons 27 days |
| age(timestamp) | interval | 计算时间差 | age(timestamp '1957-06-13') | 43 years 8 mons 3 days |
| current_date | date | 现在的日期 | select current_date; | 2019-02-18 |
| current_time | time with time zone | 现在的一天中的时间 | select current_time; | 16:42:59.991189 +08 |
| current_timestamp | timestamp with time zone | 现在的时间戳 | select current_timestamp; | 2019-02-18 16:43:20.167284 +08 |
| date_part(text, timestamp) | double precision | 获取时间日期的一部分值 | date_part('hour', timestamp '2001-02-16 20:38:40') | 20 |
| date_part(text, interval) | double precision | 获取时间日期的一部分值 | date_part('month', interval '2 years 3 months') | 3 |
| date_trunc(text, timestamp) | timestamp | 对日期时间的指定部分清零 | date_trunc('hour', timestamp '2001-02-16 20:38:40') | 2001-02-16 20:00:00 |
| extract(field from timestamp) | double precision | 与 date_part 类似 | extract(hour from timestamp '2001-02-16 20:38:40') | 20 |

| | | | | |
|---|---|---|---|---|
| extract(field from interval) | double precision | 与 date_part 类似 | extract(month from interval '2 years 3 months') | 3 |
| localtime | time | 获取本地时间 | select localtime; | 16:56:09.339026 |
| localtimestamp | timestamp | 本地日期和时间 | select localtimestamp; | 2019-02-18 16:56:42.331012 |
| now() | timestamp with time zone | 获取现在时间 | select now(); | 2019-02-18 16:56:58.843212 +08 |
| timeofday() | text | 现在日期和时间 | select timeofday(); | Mon Feb 18 16:57:27.677262 2019 CST |

# 几何函数和操作符

最近更新时间：2022-07-04 10:26:13

数据库支持的几何类型有 point、box、lseg、line、path、polygon、circle，因此数据库也提供了一系列的几何函数，具体见下列表格。

## 几何操作符

| 操作符 | 功能描述 | 示例 | 结果 |
| --- | --- | --- | --- |
| + | 转换平移 | box '((0,0),(1,1))' + point '(2.0,0)' | (3,1),(2,0) |
| - | 转换平移 | box '((0,0),(1,1))' - point '(2.0,0)' | (-1,1),(-2,0) |
| * | 伸展/旋转 | box '((0,0),(1,1))' * point '(2.0,0)' | (2,2),(0,0) |
| / | 伸展/旋转 | box '((0,0),(2,2))' / point '(2.0,0)' | (1,1),(0,0) |
| # | 两个图形交接面 | box '((1,-1),(-1,1))' # box '((1,1),(-1,-1))' | (1,1),(-1,-1) |
| @-@ | 图形的长度或者周长 | @-@ path '((0,0),(1,0))' | 2 |
| @@ | 中心 | @@ circle '((0,0),10)' | (0,0) |
| ## | 第一个图形与第二个图形最近的点 | point '(0,0)' ## lseg '((2,0),(0,2))' | (1,1) |
| <-> | 图形之间的距离 | circle '((0,0),2)' <-> circle '((4,0),1)' | 1 |
| && | 两个图形是否相交 | box '((0,0),(1,2))' && box '((0,0),(2,3))' | t |
| << | 图形1是否严格在图形2的左边 | circle '((0,0),1)' << circle '((5,0),1)' | t |
| >> | 图形1是否严格在图形2的右边 | circle '((5,0),1)' >> circle '((0,0),1)' | t |
| &< | 图形1的最右边是否不超过图形2的最右边 | box '((0,0),(1,1))' &< box '((0,0),(2,2))' | t |

| | | | |
|---|---|---|---|
| &> | 图形1的最右边是否不超过图形2的最左边 | box '((0,0),(3,3))' &> box '((0,0),(2,2))' | t |
| <<\| | 图形1是否严格在图形2下边 | box '((0,0),(2,2))' <<\| box '((4,5),(6,6))' | t |
| \|>> | 图形1是否严格在图形2上边 | box '((5,6),(7,7))' \|>> box '((0,0),(4,4))' | t |
| <^ | 图形1是否低于图形2（允许接触） | box '((5,6),(7,7))' <^ box '((0,0),(4,4))' | f |
| >^ | 图形1是否高于图形2（允许接触） | box '((5,6),(7,7))' >^ box '((0,0),(4,4))' | t |
| ?# | 相交 | lseg '((-1,0),(1,0))' ?# box '((-2,-2),(2,2))' | t |
| ?- | 是否水平 | ?- lseg '((-1,0),(1,0))' | t |
| ?- | 两个图形水平对齐 | point '(1,0)' ?- point '(0,0)' | t |
| ?\| | 是否垂直 | ?\| lseg '((-1,0),(1,0))' | f |
| ?\| | 两个图形是否垂直对齐 | point '(1,2)' ?\| point '(1,1)' | t |
| ?-\| | 两条线是否垂直 | lseg '((0,0),(0,1))' ?-\| lseg '((0,0),(1,0))' | t |
| ?\|\| | 两条线是否平行 | lseg '((-1,0),(1,0))' ?\|\| lseg '((-1,2),(1,2))' | t |
| @> | 图形1是否包含图形2 | circle '((0,0),2)' @> circle '((0,0),1)" | t |
| <@ | 图形1是否被图形2包含 | point '(1,1)' <@ circle '((0,0),2)' | t |
| ~= | 是否相同 | polygon '((0,0),(1,1))' ~= polygon '((1,1),(0,0))' | t |

## 几何函数

| 函数 | 返回值类型 | 描述 | 示例 | 结果 |
|---|---|---|---|---|
| area(object) | double precision | 面积 | area(box '((0,0),(1,2))') | 2 |
| center(object | point | 中心点 | center(box '((0,0),(1,2))') | (0.5, |

| | | | | |
|---|---|---|---|---|
| ) | | | | 1) |
| diameter(circle) | double precision | 圆的直径 | diameter(circle '((0,0),2.0)') | 4 |
| height(box) | double precision | 高 | height(box '((0,0),(2,3))') | 3 |
| isclosed(path) | boolean | 图形是否闭合 | isclosed(path '((0,0),(1,1),(2,0))') | t |
| isopen(path) | boolean | 图形是否开放 | isopen(path '[(0,0),(1,1),(2,0)]') | t |
| length(object) | double precision | 图形的长度 | length(path '((-1,0),(1,0))') | 2 |
| npoints(path) | int | 图形的顶点数 | npoints(path '[(0,0),(1,1),(2,0)]') | 3 |
| npoints(polygon) | int | 图形的顶点数 | npoints(polygon '((1,1),(0,0))') | 2 |
| radius(circle) | double precision | 圆的半径 | radius(circle '((0,0),2.0)') | 2 |
| width(box) | double precision | 水平长度 | width(box '((0,0),(3,2))') | 3 |

## 几何类型转换函数

| 函数 | 返回值类型 | 描述 | 示例 | 返回值 |
|---|---|---|---|---|
| box(circle) | box | 圆转为矩形 | box(circle '((0,0),3.0)') | (2.12132034355964,2.12132034355964),(-2.12132034355964,-2.12132034355964) (1 row) |
| box(point, point) | box | 点转为矩形 | box(point '(0,0)', point '(2,2)') | (2,2),(0,0) |
| box(polygon) | box | 多边形转为矩 | box(polygon '((0,0),(1,1),(2,0))') | (2,1),(0,0) |

| | | | | |
|---|---|---|---|---|
| | | 形 | | |
| circle(box) | circle | 矩形转为圆 | circle(box '((0,0),(2,2))') | <(1,1),1.4142135623731> |
| circle(point, double precision) | circle | 中心和半径转为圆 | circle(point '(0,0)', 3.0) | <(0,0),3> |
| circle(polygon) | circle | 多边形转为圆 | circle(polygon '((0,0),(1,1),(2,0))') | <(1,0.333333333333333),0.924950591148529> |
| lseg(box) | lseg | 矩形转为线段 | lseg(box'((-1,0),(1,0))') | [(1,0),(-1,0)] |
| lseg(point, point) | lseg | 多点转为线段 | lseg(point '(-1,0)', point '(1,0)') | [(-1,0),(1,0)] |
| path(polygon) | path | polygon to path | path(polygon '((0,0),(1,1),(2,0))') | ((0,0),(1,1),(2,0)) |
| point(double precision, double precision) | point | construct point | point(23.4, -44.5) | (23.4,-44.5) |
| point(box) | point | 图形中心 | point(box '((-1,0),(1,0))') | (0,0) |
| point(circle) | point | 圆的中心 | point(circle '((0,0),2.0)') | (0,0) |
| point(lseg) | point | 线段的中心 | point(lseg '((-2,0),(2,0))') | (0,0) |
| point(polygon) | point | 多边形的中心 | point(polygon '((0,0),(1,1),(2,0))') | (1,0.333333333333333) |
| polygon(box) | polygon | 矩形转为四个点的多边形 | polygon(box '((0,0),(1,1))') | ((0,0),(0,1),(1,1),(1,0)) |

| | | | | |
|---|---|---|---|---|
| polygon(circle) | polygon | 圆形转为12个点的多边形 | polygon(circle '((0,0),2.0)') | – |
| polygon(npts, circle) | polygon | 圆形转为 n 个点的多边形 | polygon(12, circle '((0,0),2.0)') | ((-2,0),(-0.618033988749895,1.90211303259031),(1.61803398874989,1.17557050458495),(1.6180339887499,-1.17557050458495),(-0.618033988749894,-1.9021130325903 1)) |
| polygon(path) | polygon | 路径转为多边形 | polygon(path '((0,0),(1,1),(2,0))') | ((0,0),(1,1),(2,0)) |

# 序列操作函数

最近更新时间：2021-03-02 17:57:47

数据库支持序列。序列对象是特殊的单行表，通过 CREATE SEQUENCE 创建，一个序列通常被用于生成唯一的表数据。序列的函数如下表格。例如，创建一个序列：`CREATE SEQUENCE myseq START 101;` 。

| 函数 | 返回值 | 描述 | 结果 |
| --- | --- | --- | --- |
| nextval('myseq') | bigint | 序列自增并返回最新的值 | 101 |
| setval('myseq',102) | bigint | 重置序列现在的值 | 102 |

# 条件表达式

最近更新时间：2021-03-10 09:47:56

## CASE

数据库支持 CASE 表达式，和其他语言支持 IF/ELSE 功能一致。
示例：

```
SELECT a,
       CASE WHEN a=1 THEN 'one'
            WHEN a=2 THEN 'two'
            ELSE 'other'
       END
    FROM test;
```

## COALESCE

COALESCE 返回参数中第一个为非 NULL 的值，如果所有参数都为 NULL，则返回 NULL。

```
SELECT COALESCE(null,1,2,null) ;

 coalesce

----------

        1
```

## NULLIF

NULLIF(value1,value2) 返回值：如果 value1 和 value2 相等，返回 NULL。否则，返回 value1。

## GREATEST 和 LEAST

这两个函数分别获取一列值中的最大值或者最小值。当所有参数都没有（为 NULL）时，则返回 NULL。

# 聚合函数

最近更新时间：2021-03-02 17:58:07

**数据库的聚合函数对一些列的值进行计算得到一个返回值，下列表格中是一些内置聚合函数。**

| 函数 | 参数类型 | 返回值类型 | 描述 |
| --- | --- | --- | --- |
| avg(expression) | 所有数字类型 | 数值类型 | 平均值 |
| bit_and(expression) | smallint, int, bigint, bit | 与参数一致 | 所有非空值的位与 |
| bit_or(expression) | smallint, int, bigint, bit | 与参数一致 | 所有非空值的位或 |
| bool_and(expression) | bool | bool | 所有值为 true，则返回 true，否则返回 false |
| bool_or(expression) | bool | bool | 存在至少一个值为 true，则返回 true，否则返回 false |
| count(*) | - | bigint | 返回行数 |
| count(expression) | any | bigint | 返回表达式值为非 null 的所有总和 |
| every(expression) | bool | bool | 与 bool_and 一致 |
| max(expression) | array,numeric,string, date/time type | 与参数一致 | 输入参数中最大的值 |
| min(expression) | array, numeric, string, date/time type | 与参数一致 | 输入参数中最小的值 |
| sum(expression) | smallint, int, bigint, real, double precision,numeric, interval | - | 所有输入参数中的和 |

# 子查询表达式

最近更新时间：2021-03-03 09:38:33

数据库还支持子查询表达式，下面列举的子查询返回值为 TRUE/FALSE。
原始表 t1 中数据如下：

```
select * from t1;
 a1
----
  1
  3
  2
```

## EXISTS/NOT EXISTS

EXISTS 的参数是一个 SELECT 语句，或者说子查询。系统对子查询进行运算以判断它是否返回行。如果它至少返回一行，则 EXISTS 结果就为 TRUE；如果子查询没有返回任何行，EXISTS 的结果是 FALSE。
这个子查询通常只是运行到能判断它是否可以生成至少一行为止，而不是等到全部结束。因此，通常子查询返回结果内容不是我们正在需要关注的，因此通常优化写法为：`...WHERE EXISTS(SELECT 1 FROM ...)` 。

| 示例 | 结果 |
| --- | --- |
| select a1 from t1 where EXISTS(select a1 from t1 where a1>2); | a1 ---- 2 1 3 |
| select a1 from t1 where EXISTS(select a1 from t1 where a1>3); | a1 ---- |
| select a1 from t1 where EXISTS(select a1 from t1 where a1>1) and a1>2; | a1 ---- 3 |

## IN/NOT IN

expression IN (subquery)，右边是一个圆括弧括起来的子查询，它必须只返回一个字段。左边表达式对子查询结果的每一行进行一次计算和比较。如果找到任何相等的子查询行，则 IN 结果为 TRUE。如果没有找到任何相等行，则结果为 FALSE（包括子查询没有返回任何行的情况）。
表达式或子查询行里的 NULL 遵照 SQL 处理布尔值和 NULL 组合时的规则。如果两个行对应的字段都相等且非空，则这两行相等；如果任意对应字段不等且非空，则这两行不等；否则结果是未知（NULL）。如果每一行的结果都是不等或 NULL，并且至少有一个 NULL，则 IN 的结果是 NULL。
特别注意 NOT IN 的子查询如果存在 NULL 值，则直接返回 FALSE。

| 示例 | 结果 |
| --- | --- |
| select * from t1 where a1 in (select a1 from t1 where a1 > 2); | a1 ---- 3 (1 row) |
| select * from t1 where a1 in (4,5,6); | a1 ---- (0 rows) |
| select * from t1 where a1 in (2,5,6); | a1 ---- 2 (1 row) |

# ANY/SOME

expression operator SOME/ANY (subquery)，右边是一个子查询，它必须只返回一个字段值。左边表达式使用 operator 对子查询结果的每一行进行一次计算和比较，其结果必须是布尔值。如果至少获得一个真值，则 ANY 结果为 TRUE。如果全部获得假值，则结果是 FALSE（包括子查询没有返回任何行的情况）。SOME 是 ANY 的同义词。IN 与 ANY 可以等效替换。
同 EXISTS 类似，这个与子查询结果的比较通常只是运行到能判断它是 TRUE 为止，而不是等到全部结束。

| 示例 | 结果 |
| --- | --- |
| `select * from t1 where a1 < any(select a1 from t1 where a1 <3);` | a1 ---- 1 (1 row) |
| `select * from t1 where a1 < some(select a1 from t1 where a1 <4);` | a1 ---- 1 2 (2 rows) |
| `select * from t1 where a1 < any(select a1 from t1 where a1 <1);` | a1 ---- (0 rows) |

# ALL

expression operator ALL (subquery)，右边是一个圆括弧括起来的子查询，它必须只返回一个字段。左边表达式使用 operator 对子查询结果的每一行进行一次计算和比较，其结果必须是布尔值。如果全部获得真值，ALL 结果为 TRUE（包括子查询没有返回任何行的情况）。如果至少获得一个假值，则结果是 FALSE。
NOT IN 等价于 ALL。
同 EXISTS 类似，这个与子查询结果的比较通常只是运行到能判断它是 FALSE 为止，而不是等到全部结束。
如果子查询为 NULL，则直接返回 TRUE。

| 示例 | 结果 |
| --- | --- |
| `select * from t1 where a1 < all(select a1 from t1 where a1 <1);` | a1 ---- 1 3 2 (3 rows) |

| | |
|---|---|
| `select * from t1 where a1 < all(select a1 from t1 where a1 >2);` | a1 ---- 2 1 (2 rows) |

# 行值与数组的比较

最近更新时间：2021-03-03 09:39:39

| 操作符 | 功能描述 | 示例 | 结果 |
| --- | --- | --- | --- |
| IN | 与子查询的 IN 一致 | select * from t1 where a1 in (2,5,6); | a1<br>----<br>2<br>(1 row) |
| NOT IN | 与子查询的 NOT IN 一致 | select * from t1 where a1 not in (2,5,6); | a1<br>----<br>1<br>3<br>(2 rows) |
| ANY/SOME | 对比结果至少有一个真值，则返回 TRUE，否则返回 FALSE | select * from t1 where a1 < any(array[1,3]); | a1<br>----<br>2<br>1<br>(2 rows) |
| ALL | 对比结果都为 TRUE，则返回 TRUE，否则返回 FALSE | select * from t1 where a1 < ALL(array[3]); | a1<br>----<br>2<br>1<br>(2 rows) |

# 序列函数

最近更新时间：2021-03-03 09:56:02

**数据库提供的序列函数，返回多个值，具体见下面表格。**

| 函数 | 参数类型 | 返回类型 | 描述 | 示例 |
| --- | --- | --- | --- | --- |
| generate_series(start, stop) | 整数 | 返回一系列值 | 从 start 开始，自增到 stop 的一系列值 | select * from generate_series(2,4);<br>generate_series<br>----------<br>2<br>3<br>4<br>(3 rows) |
| generate_series(start, stop, step) | 整数 | 返回一系列值 | 从 start 开始，步长为 step 自增到 stop 的一系列值 | select * from generate_series(5,1,-2);<br>generate_series<br>----------<br>5<br>3<br>1<br>(3 rows) |

# 系统信息函数

最近更新时间：2024-08-15 16:19:42

**数据库提供的系统信息函数可以获取会话和系统信息，具体见下表。**

| 函数名 | 返回值 | 说明 |
| --- | --- | --- |
| current_database() | name | 现在的数据库名称 |
| current_schema() | name | 现在的 schema 名称 |
| current_schemas(boolean) | name[] | search_path 设置的所有 schema |
| current_user | name | 用户 |
| inet_client_addr() | inet | 当前客户端的地址（远程连接模式下有效） |
| inet_client_port() | int | 当前客户端的端口（远程连接模式下有效） |
| inet_server_addr() | inet | 接受连接的服务端 IP 地址（远程连接模式下有效） |
| inet_server_port() | int | 接受连接的服务端端口（远程连接模式下有效） |
| pg_postmaster_start_time() | timestamp with time zone | 服务器启动时间 |
| session_user | name | 会话用户名称 |
| user | name | 等价于 current_user |
| version() | text | PostgreSQL 版本 |

# 系统管理函数

最近更新时间：2021-03-03 09:56:20

## 配置设置函数

下面是查询和修改运行时配置参数的函数，见下面表格。

| 函数 | 返回值 | 描述 | 示例 | 结果 |
| --- | --- | --- | --- | --- |
| current_setting(setting_name) | text | 现在的参数值 | SELECT current_setting('datestyle'); | current_setting ---------- ISO, MDY |
| set_config(setting_name, new_value, is_local) | text | 设置参数值，并返回最新的值 | SELECT set_config('log_statement_stats', 'off', false); | set_config ---------- off |

## 服务信号函数

| 函数 | 返回值 | 描述 |
| --- | --- | --- |
| pg_cancel_backend(pid int) | boolean | 取消一个后端的当前查询 |
| pg_reload_conf() | boolean | 触发重新加载配置文件 |
| pg_rotate_logfile() | boolean | 滚动服务器的日志文件，只有内置日志收集器的有用，其他的则没用，因为没有管理日志的子进程 |

### 数据库对象大小函数

| 函数 | 返回值 | 描述 | 示例 | 结果 |
| --- | --- | --- | --- | --- |

| | | | | |
|---|---|---|---|---|
| pg_column_size(any) | int | 获取数据的存储空间字节大小 | select pg_column_size('ddewewe'); | 8 |
| pg_database_size(oid) | bigint | 指定数据库的大小 | select oid,* from pg_database; select pg_database_size(16384); | pg_database_size<br>---<br>127632410 |
| pg_database_size(name) | bigint | 指定数据库的大小 | select pg_database_size('gpperfmon'); | 127632410 |
| pg_relation_size(oid) | bigint | 获取指定表或者索引的大小 | select pg_relation_size(17787); | pg_relation_size<br>---<br>65536 |
| pg_relation_size(text) | bigint | 获取表或者索引的大小 | select pg_relation_size('t1'); | pg_relation_size<br>---<br>65536 |
| pg_size_pretty(bigint) | text | 字节数转为格式化的大小 | select pg_size_pretty(122212121); | pg_size_pretty<br>---<br>117 MB |
| pg_tablespace_size(oid) | bigint | 获取指定表空间的大小 | select oid,* from pg_tablespace ; select pg_tablespace_size(1663); | pg_tablespace_size<br>---<br>262275170 |
| pg_tablespace_size(name) | bigint | 获取指定表空间的大小 | select pg_tablespace_size('pg_default'); | pg_tablespace_size<br>---<br>262275170 |
| pg_total_relation_size(oid) | bigint | 指定表所占的包括索引和数据的磁盘空间 | select oid,relname from pg_class where relname='t1'; select pg_total_relation_size(17787); | pg_total_relation_size<br>---<br>65536 (1 row) |
| pg_total_relation_size(text) | bigint | 指定表所占的包括索引和数据的磁盘空间 | select pg_total_relation_size('t1'); | pg_total_relation_size<br>---<br>65536 |

# DDL 语法一览表

最近更新时间：2025-03-05 15:08:32

## 定义数据库

数据库是组织、存储和管理数据的仓库，而数据库定义主要包括：创建数据库、修改数据库属性，以及删除数据库。所涉及的 SQL 语句，请参考下表。

### 数据库定义相关 SQL

| 功能 | 相关 SQL |
| --- | --- |
| 创建数据库 | CREATE DATABASE |
| 修改数据库属性 | ALTER DATABASE |
| 删除数据库 | DROP DATABASE |

## 定义模式

模式是一组数据库对象的集合，主要用于控制对数据库对象的访问。所涉及的 SQL 语句，请参考下表。

### 模式定义相关 SQL

| 功能 | 相关 SQL |
| --- | --- |
| 创建模式 | CREATE SCHEMA |
| 修改模式属性 | ALTER SCHEMA |
| 删除模式 | DROP SCHEMA |

## 定义表

表是数据库中的一种特殊数据结构，用于存储数据对象以及对象之间的关系。所涉及的 SQL 语句，请参考下表。

### 表定义相关 SQL

| 功能 | 相关 SQL |
| --- | --- |
| 创建表 | • SHARD 表：CREATE TABLE<br>• 复制表：CREATE TABLE ... DISTRIBUTE BY REPLICATION<br>• 列存表：CREATE TABLE ... WITH(orientation=column)<br>• 外表：外表读写 COS 使用说明 |

| | |
|---|---|
| 修改表属性 | ALTER TABLE |
| 删除表 | DROP TABLE |

> ⚠ 注意
> 同一个 alter 语句仅支持执行一个 DDL 操作，同时执行多个 DDL 操作可能会导致索引元数据损坏。

## 定义分区表

分区表是一种逻辑表，数据是由普通表存储的，主要用于提升查询性能。所涉及的 SQL 语句，请参考下表。

### 分区表定义相关 SQL

| 功能 | 相关 SQL |
|---|---|
| 创建分区表 | • range 分区表：CREATE TABLE ….PARTITION BY range…<br>• list 分区：CREATE TABLE ….PARTITION BY list…<br>• hash 分区：CREATE TABLE ….PARTITION BY hash…<br>• 多级分区表：CREATE TABLE ... PARTITION OF ... PARTITION BY ... |
| 删除分区 | DROP TABLE ... |

> ⚠ 注意
> 同一个 alter 语句仅支持执行一个 DDL 操作，同时执行多个 DDL 操作可能会导致索引元数据损坏。

## 定义索引

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

### 索引定义相关 SQL

| 功能 | 相关 SQL |
|---|---|
| 创建索引 | CREATE INDEX |
| 删除索引 | DROP INDEX |

> ⚠ 注意
> 同一个 alter 语句仅支持执行一个 DDL 操作，同时执行多个 DDL 操作可能会导致索引元数据损坏。

# 定义视图

视图是从一个或几个基本表中导出的虚表，可用于控制用户对数据访问，请参考下表。

| 功能 | 相关 SQL |
| --- | --- |
| 创建视图 | CREATE VIEW |
| 删除视图 | DROP VIEW |

# DML 语法一览表

最近更新时间：2021-03-03 10:18:54

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

## 插入数据

向表中插入一条或者多条数据。

| 功能 | 相关 SQL |
| --- | --- |
| 插入数据 | INSERT |

## 修改数据

修改表中的一条或者多条数据。

| 功能 | 相关 SQL |
| --- | --- |
| 修改数据 | UPDATE |

## 查询数据

在数据表中查询符合指定条件的数据。

| 功能 | 相关 SQL |
| --- | --- |
| 查询数据 | SELECT |

## 删除数据

可以删除指定的一条或者多条数据，也可以清空一张表。

| 功能 | 相关 SQL |
| --- | --- |
| 删除数据 | DELECT |
| 清空数据 | TRUNCATE |

## 拷贝数据

在一个文件和一个表之间复制数据。

| 功能 | 相关 SQL |
| --- | --- |
| 拷贝数据 | COPY |

# DCL 语法一览表

最近更新时间：2021-03-03 10:19:32

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

## 授权

定义访问特权。

| 功能 | 相关 SQL |
| --- | --- |
| 授权 | GRANT |

## 收回权限

回收指定的权限。

| 功能 | 相关 SQL |
| --- | --- |
| 回收授权 | REVOKE |

# 数据库开发基础
# 数据类型

最近更新时间：2024-07-24 17:02:21

## 数字类型

| 名字 | 存储尺寸 | 描述 | 范围 |
| --- | --- | --- | --- |
| smallint | 2字节 | 小范围整数 | -32768 至 +32767 |
| integer | 4字节 | 整数的典型选择 | -2147483648 至 +2147483647 |
| bigint | 8字节 | 大范围整数 | -9223372036854775808 至 +9223372036854775807 |
| decimal | 可变 | 用户指定精度，精确 | 最高小数点前131072位，以及小数点后16383位 |
| numeric | 可变 | 用户指定精度，精确 | 最高小数点前131072位，以及小数点后16383位 |
| real | 4字节 | 可变精度，不精确 | 6位十进制精度 |
| double precision | 8字节 | 可变精度，不精确 | 15位十进制精度 |
| smallserial | 2字节 | 自动增加的小整数 | 1到32767 |
| serial | 4字节 | 自动增加的整数 | 1到2147483647 |
| bigserial | 8字节 | 自动增长的大整数 | 1到9223372036854775807 |

## 字符类型

| 名字 | 描述 |
| --- | --- |
| character varying(n), varchar(n) | 有限制的变长 |
| character(n), char(n) | 定长，空格填充 |

| | |
|---|---|
| text | 1G |

## 二进制数据类型

| 名字 | 存储尺寸 | 描述 |
|---|---|---|
| bytea | 1或4字节外加真正的二进制串 | 变长二进制串 |

## 日期类型

| 名字 | 存储尺寸 | 描述 | 最小值 | 最大值 | 解析度 |
|---|---|---|---|---|---|
| timestamp [ (p) ] [ without time zone ] | 8字节 | 包括日期和时间（无时区） | 4713 BC | 294276 AD | 1微秒 / 14 位 |
| timestamp [ (p) ] with time zone | 8字节 | 包括日期和时间，有时区 | 4713 BC | 294276 AD | 1微秒 / 14 位 |
| date | 4字节 | 日期（没有一天中的时间） | 4713 BC | 5874897 AD | 1日 |
| time [ (p) ] [ without time zone ] | 8字节 | 一天中的时间（无日期） | 0:00:00 | 24:00:00 | 1微秒 / 14 位 |
| time [ (p) ] with time zone | 12字节 | 仅仅是一天中的时间，带有时区 | 00:00:00+1459 | 24:00:00−1459 | 1微秒 / 14 位 |
| interval [ fields ] [ (p) ] | 16字节 | 时间间隔 | −178000000年 | 178000000年 | 1微秒 / 14 位 |

## 布尔类型

| 名字 | 存储字节 | 描述 |
|---|---|---|
| boolean | 1字节 | 状态为真或假 |

## 更多的数据类型介绍

请点击连接 https://www.postgresql.org/docs/10/static/datatype.html 查看更多的数据类型说明

# 自增列与序列的用法

最近更新时间：2024-07-24 17:02:21

## 序列的创建和访问

```
[tbase@VM_0_29_centos tbase_mgr]$ psql -p 15432 -U tbase -d postgres
psql (PostgreSQL 10 (TBase 2.01))
Type "help" for help.

-- 建立序列
postgres=# create sequence tbase_seq;
CREATE SEQUENCE

-- 建立序列，不存在时才创建
postgres=# create sequence IF NOT EXISTS tbase_seq;
NOTICE: relation "tbase_seq" already exists, skipping
CREATE SEQUENCE

-- 查看序列当前的使用状况
postgres=# \\x
Expanded display is on.
postgres=# select * from tbase_seq ;
-[ RECORD 1 ]-
last_value | 1
log_cnt | 0
is_called | f

-- 获取序列的下一个值
postgres=# select nextval('tbase_seq');
 nextval
---------
 1

-- 获取序列的当前值，这个需要在访问 nextval()后才能使用
postgres=# select currval('tbase_seq');
 currval
---------
 1

-- 可以后下面的方式来获取序列当前使用到那一个值
postgres=# select last_value from tbase_seq ;
```

```
 last_value
------------
 3

-- 设置序列当前值
postgres=# select setval('tbase_seq',1);
 setval
--------
 1
```

## 序列在 DML 中使用

```
-- 插入数据，使用序列
postgres=# INSERT INTO t (id, nickname) VALUES (nextval('tbase_seq'),
'TBase 好');
INSERT 0 1

postgres=# select * from t;
 id | nickname
----+-----------
 1 | 腾讯 TBase
 2 | TBase 好
(2 rows)
```

## 序列作为字段的默认值使用

```
-- 将序列设置为字段的默认值
postgres=# alter table t alter column id set default
nextval('tbase_seq');

-- 插入数据，字段使用序列默认值
postgres=# INSERT INTO t (nickname) VALUES ('hello TBase');

postgres=# select * from t;
 id | nickname
----+-------------
 3 | hello TBase
 1 | 腾讯 TBase
 2 | TBase 好
(3 rows)
```

## 序列作为字段类型使用

```
-- 删除表
postgres=# drop table t;
DROP TABLE

-- 创建表，字段使用序列类型
postgres=# create table t (id serial not null, nickname text);
CREATE TABLE

-- 插入数据，字段自动使用序列
postgres=# INSERT INTO t (nickname) VALUES ('hello TBase');
INSERT 0 1

postgres=# select * from t;
 id | nickname
----+-------------
  1 | hello TBase
(1 row)
```

## 删除序列

```
-- 删除序列
postgres=# drop sequence tbase_seq;
DROP SEQUENCE

-- 删除序列，不存在时跳过
postgres=# drop sequence IF EXISTS tbase_seq;
NOTICE: sequence "tbase_seq" does not exist, skipping
DROP SEQUENCE
```

# 库/模式/表/索引/视图等 DDL 操作

最近更新时间：2025-03-05 15:08:32

## 数据库管理

### 创建数据库

要创建一个数据库，您必须是一个超级用户或者具有特殊的 CREATEDB 特权，默认情况下，新数据库将通过克隆标准系统数据库 template1 被创建。可以通过写 TEMPLATE name 指定一个不同的模板。特别地，通过写 TEMPLATE template0 您可以创建一个干净的数据库，它将只包含您的 TBase 版本所预定义的标准对象。

### 默认参数创建数据库

```
postgres=# create database tbase_db;
CREATE DATABASE
```

### 指定克隆库

```
postgres=# create database tbase_db_template TEMPLATE template0;
CREATE DATABASE
```

### 指定所有者

```
postgres=# create role pgxz with login;
CREATE ROLE
postgres=# create database tbase_db_owner owner pgxz;
CREATE DATABASE
```

列出数据库及其详细信息：

```
List of databases
| Name              | Owner | Encoding | Collate   | Ctype   | Access
privileges | Size   | Tablespace | Description       |
|-----------------|-------|----------|----------|--------|--------------
----|-------|------------|-------------------|
| tbase_db_owner    | pgxz   | UTF8      | en_US.utf8 | en_US.utf8 |
| 18 MB | pg_default |
```

### 指定编码

```
postgres=# create database tbase_db_encoding ENCODING UTF8;
CREATE DATABASE
```

**列出数据库及其详细信息：**

```
List of databases
| Name                    | Owner    | Encoding | Collate   | Ctype   |
Access privileges | Size   | Tablespace | Description       |
|-----------------------|--------|----------|----------|--------|--------
----------|-------|------------|--------------------|
| tbase_db_encoding     | tbase   | UTF8      | en_US.utf8 | en_US.utf8 |
| 18 MB | pg_default |
```

## 创建 GBK 编码数据库

```
postgres=# CREATE DATABASE db_gbk template template0 encoding = gbk
LC_COLLATE = 'zh_CN.gbk' LC_CTYPE = 'zh_CN.gbk';
CREATE DATABASE
```

**列出数据库及其详细信息：**

```
List of databases
| Name   | Owner | Encoding | Collate   | Ctype   | Access privileges |
Size   | Tablespace | Description       |
|-------|-------|----------|----------|--------|------------------|-----
--|------------|--------------------|
| db_gbk | tbase | GBK       | zh_CN.gbk | zh_CN.gbk |                   |
19 MB | pg_default |
```

## 创建 GB18030 编码数据库

```
postgres=# create database db_gb18030 template=template0
encoding=gb18030 LC_COLLATE = 'zh_CN.gb18030' LC_CTYPE =
'zh_CN.gb18030';
CREATE DATABASE
```

**列出数据库及其详细信息：**

```
List of databases
```

```
| Name         | Owner | Encoding | Collate         | Ctype           |
Access privileges |
|--------------|-------|----------|---------------|---------------|------
------------|
| db_gb18030   | tbase | GB18030  | zh_CN.gb18030   | zh_CN.gb18030   |
...
```

## 指定排序规则

```
postgres=# create database tbase_db_lc_collate lc_collate 'C';
CREATE DATABASE
```

**列出数据库及其详细信息：**

```
List of databases
| Name                    | Owner | Encoding | Collate | Ctype | Access
privileges | Size   | Tablespace | Description       |
|----------------------|-------|----------|---------|-------|-----------
--------|-------|------------|--------------------|
| tbase_db_lc_collate   | tbase | UTF8     | C       | en_US.utf8 |
| 18 MB | pg_default |
```

## 指定分组规则

```
postgres=# create database tbase_db_lc_ctype LC_CTYPE 'C' ;
CREATE DATABASE
```

**列出数据库及其详细信息：**

```
List of databases
| Name                 | Owner | Encoding | Collate   | Ctype | Access
privileges | Size   | Tablespace | Description       |
|---------------------|-------|----------|----------|-------|------------
------|-------|------------|--------------------|
| tbase_db_lc_ctype | tbase | UTF8     | en_US.utf8 | C     |
| 18 MB | pg_default |
```

## 配置数据可连接

```
postgres=# create database tbase_db_allow_connections ALLOW_CONNECTIONS
true;
CREATE DATABASE
```

查看数据库连接配置：

```
postgres=# select datallowconn from pg_database where
datname='tbase_db_allow_connections';
datallowconn
--------------
t
```

## 配置连接数

```
postgres=# create database tbase_db_connlimit CONNECTION LIMIT 100;
CREATE DATABASE
```

查看数据库连接数配置：

```
postgres=# select datconnlimit from pg_database where
datname='tbase_db_connlimit';
datconnlimit
--------------
100
```

## 配置数据库可以被复制

```
postgres=# create database tbase_db_istemplate is_template true;
CREATE DATABASE
```

查看数据库模板配置：

```
postgres=# select datistemplate from pg_database where
datname='tbase_db_istemplate';
datistemplate
---------------
t
```

## 多个参数一起配置

```
postgres=# create database tbase_db_mul owner pgxz CONNECTION LIMIT 50
template template0 encoding 'utf8' lc_collate 'C';
CREATE DATABASE
```

## 修改数据库配置

### 修改数据库名称

```
postgres=# alter database tbase_db rename to tbase_db_new;
ALTER DATABASE
```

### 修改连接数

```
postgres=# alter database tbase_db_new connection limit 50;
ALTER DATABASE
```

### 修改数据库所有者

```
postgres=# alter database tbase_db_new owner to tbase;
ALTER DATABASE
```

### 配置数据默认运行参数

```
postgres=# alter database tbase_db_new set search_path to
public,pg_catalog,pg_oracle;
ALTER DATABASE
```

更多 `set` 的用法可以参考运维文档。

## Alter database 不支持的项目

| 项目 | 备注 |
| --- | --- |
| encoding | 编码 |
| lc_collate | 排序规则 |
| lc_ctype | 分组规则 |

## 删除数据库

### 删除数据库

```
postgres=# drop database tbase_db_new;
DROP DATABASE
```

删除数据库时，如果该数据库已经有 session 连接上来，则会提示如下错误：

```
postgres=# drop database tbase_db_template;
ERROR:  database "tbase_db_template" is being accessed by other users
DETAIL:  There is 1 other session using the database.
```

可以使用 `with (force)` 参数强制删除数据库：

```
postgres=# drop database tbase_db_template with(force);
DROP DATABASE
```

## 模式管理

模式本质上是一个名字空间，在不同的数据库系统中有不同的叫法。在 Oracle 中通常称为用户，在 SQL Server 中称为框架，在 MySQL 中称为数据库。模式中可以包含表、数据类型、函数以及操作符。对象名称可以在不同模式中重名，访问特定模式中的对象时，可以使用“模式名.对象名”的格式。

### 创建模式

- 标准语句：

```
postgres=# create schema tbase;
  CREATE SCHEMA
```

- 扩展语法，当模式不存在时才创建：

```
postgres=# create schema if not exists tbase;
NOTICE: schema "tbase" already exists, skipping
CREATE SCHEMA
```

- 指定模式的所属用户：

```
postgres=# create schema tbase_pgxz AUTHORIZATION pgxz;
CREATE SCHEMA
```

使用 `\dn` 命令列出模式及其所有者：

```
List of schemas
Name | Owner
------------+-------
tbase_pgxz | pgxz
(1 row)
```

## 修改模式属性

- 修改模式名：

```
postgres=# alter schema tbase rename to tbase_new;
ALTER SCHEMA
```

- 修改所有者：

```
postgres=# alter schema tbase_pgxz owner to tbase;
ALTER SCHEMA
```

## 删除模式

- 删除模式命令：

```
postgres=# drop schema tbase_new;
DROP SCHEMA
```

- 当模式中存在对象时，删除会失败，并提示错误信息。可以使用 `CASCADE` 选项强制删除及其依赖对象：

```
postgres=# drop schema tbase_pgxz CASCADE;
NOTICE: drop cascades to table tbase_pgxz.t
DROP SCHEMA
```

## 配置用户访问模式权限

普通用户访问某个模式下的对象需要两个授权步骤：访问对象本身和访问模式。

- 授权用户访问特定表：

```
postgres=# grant select on tbase.t to pgxz;
```

```
GRANT
```

- 授权用户使用特定模式：

```
postgres=# grant USAGE on SCHEMA tbase to pgxz;
GRANT
```

## 配置访问模式的顺序

TCHouse-P 数据库使用 `search_path` 运行变量来配置访问数据对象的顺序。

- 当前连接用户的搜索路径：

```
postgres=# show search_path ;
search_path
----------------
"$user", public
(1 row)
```

- 创建数据表时，如果不指定模式，则表存放于第一个搜索模式下：

```
postgres=# create table t(id int,mc text);
NOTICE: Replica identity is needed for shard table, please add to this
table through "alter table" command.
CREATE TABLE
```

- 访问不在搜索路径中的对象时，需要写全路径：

```
postgres=# select * from t1;
ERROR: relation "t1" does not exist
postgres=# select * from tbase_schema.t1;
id | mc
----+----
(0 rows)
```

# 创建和删除数据表

## 不指定 shard key 建表方式

不指定 shard key 建表时，系统默认使用第一个字段作为表的 shard key。

```
postgres=# create table t_first_col_share(id serial not null, nickname
text);
CREATE TABLE
postgres=# \d+ t_first_col_share
```

| Column | Type | Modifiers | Storage | Stats target | Description |
|---|---|---|---|---|---|
| id | integer | not null default nextval('t_first_col_share_id_seq'::regclass) | plain | | |
| nickname | text | | extended | | |

Has OIDs: no
Distribute By SHARD(id)
Location Nodes: ALL DATANODES

**分布键选择原则：**

- **分布键只能选择一个字段。**
- **如果有主键，则选择主键做分布键。**
- **如果主键是复合字段组合，则选择字段值选择性多的字段做分布键。**
- **也可以把复合字段拼接成一个新的字段来做分布键。**
- **没有主键的可以使用 UUID 来做分布键。**
- **总之一定要让数据尽可能的分布得足够散。**

## 指定 shard key 建表方式

**指定 shard key 建表时，可以通过 `distribute by shard(column_name)` 语句指定。**

```
postgres=# create table t_appoint_col(id serial not null, nickname text)
distribute by shard(nickname);
CREATE TABLE
postgres=# \d+ t_appoint_col
```

Table "public.t_appoint_col"

| Column | Type | Modifiers | Storage | Stats target | Description |
|---|---|---|---|---|---|

| | | | | | |
|---|---|---|---|---|---|
| id | integer | not null default nextval('t_appoint_col_id_seq'::regclass) | plain | | |
| nickname | text | | extended | | |

Has OIDs: no
Distribute By SHARD(nickname)
Location Nodes: ALL DATANODES

## 指定 group 建表方式

指定 group 建表方式通过 `to group group_name` 语句指定。

```
postgres=# create table t (id integer, nc text) distribute by shard(id)
to group default_group;
CREATE TABLE
postgres=# \d+ t
```

Table "public.t"

| Column | Type | Modifiers | Storage | Stats target | Description |
|---|---|---|---|---|---|
| id | integer | | plain | | |
| nc | text | | extended | | |

Has OIDs: no
Distribute By SHARD(id)
Location Nodes: ALL DATANODES

## 逻辑分区表

## range 分区表

创建主分区和子分区。

```
postgres=# create table t_native_range (f1 bigint, f2 timestamp default
now(), f3 integer) partition by range(f2) distribute by shard(f1) to
group default_group;
NOTICE: Replica identity is needed for shard table, please add to this
table through "alter table" command.
```

```
CREATE TABLE
```

建立两个子表：

```
postgres=# create table t_native_range_201709 partition of
t_native_range (f1, f2, f3) for values from ('2017-09-01') to ('2017-10-
01');
postgres=# create table t_native_range_201710 partition of
t_native_range (f1, f2, f3) for values from ('2017-10-01') to ('2017-11-
01');
```

注意添加子分区时会影响主表的数据 DML 操作。

### 默认分区表

没有默认分区表时插入会出错。

```
postgres=# insert into t_native_range values(2,'2017-08-01',2);
ERROR: no partition of relation "t_native_range" found for row
```

添加默认分区表：

```
postgres=# CREATE TABLE t_native_range_default PARTITION OF
t_native_range DEFAULT;
CREATE TABLE
postgres=# insert into t_native_range values(2,'2017-08-01',2);
INSERT 0 1
```

## MAXVALUE 分区

创建MAXVALUE分区。

```
postgres=# CREATE TABLE t_native_range_maxvalue PARTITION OF
t_native_range for values from ('2017-11-01') to (maxvalue);
CREATE TABLE
postgres=# insert into t_native_range values(1,'2017-11-01',1);
INSERT 0 1
```

所有比 `2017-11-01` 大的数据都存储到子表 `t_native_range_maxvalue` 。

## MINVALUE 分区

创建 MINVALUE 分区。

```
postgres=# CREATE TABLE t_native_range_minvalue PARTITION OF
t_native_range for values from (minvalue) to ('2017-09-01');
CREATE TABLE
postgres=# insert into t_native_range values(1,'2017-08-01',1);
INSERT 0 1
```

### 查看表结构

查看 `t_native_range` 表结构。

```
postgres=# \d+ t_native_range
```

Table "tbase_pg_proc.t_native_range"

| Column | Type | Collation | Nullable | Default | Storage | Stats target | Description |
|---|---|---|---|---|---|---|---|
| f1 | bigint | | | | plain | | |
| f2 | timestamp without time zone | | | now() | plain | | |
| f3 | integer | | | | plain | | |

Partition key: RANGE (f2)
Partitions: ...
Distribute By: SHARD(f1)
Location Nodes: ALL DATANODES

## list 分区表

创建主分区和子分区。

```
postgres=# create table t_native_list(f1 bigserial not null, f2 text, f3
integer, f4 date) partition by list(f2) distribute by shard(f1) to group
default_group;
```

建立两个子表，分别存入“广东”和“北京”。

```
postgres=# create table t_list_gd partition of t_native_list(f1, f2, f3,
f4) for values in ('广东');
postgres=# create table t_list_bj partition of t_native_list(f1, f2, f3,
f4) for values in ('北京');
```

查看表结构：

```
postgres=# \d+ t_native_list
```

Table "tbase.t_native_list"

| Column | Type | Collation | Nullable | Default | Storage | Stats target | Description |
|---|---|---|---|---|---|---|---|
| f1 | bigint | | not null | nextval('t_native_list_f1_seq'::regclass) | plain | | |
| f2 | text | | | | extended | | |
| f3 | integer | | | | plain | | |
| f4 | date | | | | plain | | |

Partition key: LIST (f2)
Partitions: ...
Distribute By: SHARD(f1)
Location Nodes: ALL DATANODES

## 创建 default 分区

没有 default 分区情况下会出错，插入会出错。

```
postgres=# insert into t_native_list values(1,'上海',1,current_date);
ERROR: no partition of relation "t_native_list" found for row
```

创建后就能正常插入。

```
postgres=# CREATE TABLE t_native_list_default PARTITION OF t_native_list
DEFAULT;
CREATE TABLE
postgres=# insert into t_native_list values(1,'上海',1,current_date);
INSERT 0 1
```

## Hash 分区表

**创建4个子分区的 hash 表时，需要指定分区的个数。使用分区数作为算子来计算每条数据所在的分区。目前，hash 分区不支持添加和删除操作。**

```
postgres=# create table t_hash_partition(f1 int, f2 int) partition by
hash(f2);
create table t_hash_partition_1 partition of t_hash_partition FOR VALUES
WITH(MODULUS 4, REMAINDER 0);
create table t_hash_partition_2 partition of t_hash_partition FOR VALUES
WITH(MODULUS 4, REMAINDER 1);
create table t_hash_partition_3 partition of t_hash_partition FOR VALUES
WITH(MODULUS 4, REMAINDER 2);
create table t_hash_partition_4 partition of t_hash_partition FOR VALUES
WITH(MODULUS 4, REMAINDER 3);
```

**插入数据：**

```
postgres=# insert into t_hash_partition values(1,1),(2,2),(3,3);
COPY 3
```

**查询结果：**

```
postgres=# select * from t_hash_partition;
+----+----+
| f1 | f2 |
+----+----+
|  1 |  1 |
|  3 |  3 |
|  2 |  2 |
+----+----+
(3 rows)
```

**TBase 自动根据分区值进行剪枝：**

```
postgres=# explain select * from t_hash_partition where f2=2;
                                    QUERY PLAN
-------------------------------------------------------------------------------
-------
 Remote Fast Query Execution  (cost=0.00..0.00 rows=0 width=0)
   Node/s: dn001, dn002
   ->  Append  (cost=0.00..26.88 rows=7 width=8)
         ->  Seq Scan on t_hash_partition_3  (cost=0.00..26.88 rows=7
width=8)
               Filter: (f2 = 2)
(5 rows)
```

## 多级分区表

### 创建主表

```
postgres=# create table t_native_mul_list(f1 bigserial not null, f2
integer, f3 text, f4 text, f5 date)
partition by list (f3) distribute by shard(f1) to group default_group;
NOTICE:  Replica identity is needed for shard table, please add to this
table through "alter table" command.
CREATE TABLE
```

### 创建二级表

```
postgres=# create table t_native_mul_list_gd partition of
t_native_mul_list for values in ('广东')
partition by range(f5);
NOTICE:  Replica identity is needed for shard table, please add to this
table through "alter table" command.
CREATE TABLE

postgres=# create table t_native_mul_list_bj partition of
t_native_mul_list for values in ('北京')
partition by range(f5);
NOTICE:  Replica identity is needed for shard table, please add to this
table through "alter table" command.
CREATE TABLE

postgres=# create table t_native_mul_list_sh partition of
t_native_mul_list for values in ('上海');
```

```
NOTICE:  Replica identity is needed for shard table, please add to this
table through "alter table" command.
CREATE TABLE
```

### 创建三级表

```
postgres=# create table t_native_mul_list_gd_201701 partition of
t_native_mul_list_gd(f1,f2,f3,f4,f5)
for values from ('2017-01-01') to ('2017-02-01');
NOTICE:  Replica identity is needed for shard table, please add to this
table through "alter table" command.
CREATE TABLE

postgres=# create table t_native_mul_list_gd_201702 partition of
t_native_mul_list_gd(f1,f2,f3,f4,f5)
for values from ('2017-02-01') to ('2017-03-01');
NOTICE:  Replica identity is needed for shard table, please add to this
table through "alter table" command.
CREATE TABLE

postgres=# create table t_native_mul_list_bj_201701 partition of
t_native_mul_list_bj(f1,f2,f3,f4,f5)
for values from ('2017-01-01') to ('2017-02-01');
NOTICE:  Replica identity is needed for shard table, please add to this
table through "alter table" command.
CREATE TABLE

postgres=# create table t_native_mul_list_bj_201702 partition of
t_native_mul_list_bj(f1,f2,f3,f4,f5)
for values from ('2017-02-01') to ('2017-03-01');
NOTICE:  Replica identity is needed for shard table, please add to this
table through "alter table" command.
CREATE TABLE
```

TBase 支持1级和2级分区混用，分区不需要都平级。

## 数据库复制表

复制表在所有 dn 节点都存储一份相同的数据。创建复制表后，可以插入数据，并在不同节点上查询到相同的结果。复制表不支持触发器和外键。

### 创建复制表

```
CREATE TABLE t_rep (id INT, mc TEXT) DISTRIBUTE BY REPLICATION TO GROUP
default_group;
```

### 插入数据

```
INSERT INTO t_rep VALUES(1, 'TBase'), (2, 'pgxz');
```

### 查询数据

```
EXECUTE DIRECT ON (dn001) 'SELECT * FROM t_rep';
EXECUTE DIRECT ON (dn002) 'SELECT * FROM t_rep';
```

## 列存表管理

列存表只在 TBase-v3 中支持。

### 创建列存表

```
CREATE TABLE t_col_test (f1 INT, f2 VARCHAR(32), f3 DATE) WITH
(orientation='column');
```

### 查看列存表结构

```
\d+ t_col_test
```

### 指定压缩类型

- 所有列使用统一的压缩级别：

```
CREATE TABLE t_col1 (f1 INT, f2 INT, f3 DATE) WITH
(orientation=column, compression=HIGH);
```

- 配置默认压缩级别：

```
SHOW default_rel_compression;
SET default_rel_compression TO 'middle';
```

- 设置不同列不同压缩级别：

```
CREATE TABLE t_col3 (
  f1 INT ENCODING(compression=no),
  f2 INT ENCODING(compression=low),
  f3 INT ENCODING(compression=middle),
  f4 INT ENCODING(compression=high)
) WITH (orientation=column);
```

## 列存分区表

### 创建主表

```
CREATE TABLE t_native_range (
  f1 BIGINT,
  f2 TIMESTAMP DEFAULT NOW(),
  f3 INTEGER
) PARTITION BY RANGE (f2) WITH (orientation=column, compression=low);
```

### 创建子表

```
CREATE TABLE t_native_range_201709 PARTITION OF t_native_range FOR
VALUES FROM ('2017-09-01') TO ('2017-10-01') WITH (orientation=column,
compression=low);
CREATE TABLE t_native_range_201710 PARTITION OF t_native_range FOR
VALUES FROM ('2017-10-01') TO ('2017-11-01') WITH (orientation=column,
compression=low);
```

### 查看表结构

```
\d+ t_native_range
```

## 列存 Stash 表

列存表的 Stash 功能允许将单条插入或更新操作的数据首先存储在 Stash 中，然后根据设置将数据持久化存储至列存储结构。

### 创建 Stash 表

```
CREATE TABLE t_stash (f1 INT, f2 INT) WITH (orientation=column,
stash_enabled=on);
```

### 修改表为 Stash 表

```
ALTER TABLE t_stash SET (stash_enabled=off);
```

## 外表管理

## 创建 hdfs_fdw 插件

```
CREATE EXTENSION hdfs_fdw;
```

## 创建 Server

### 创建 COS Server

```
CREATE SERVER $cos_server_name
 FOREIGN DATA WRAPPER hdfs_fdw
 OPTIONS (
  address 'cos://$bucketname',
  appid '$appid',
  access_keyid '$ak',
  secret_accesskey '$sk',
  region '$region',
  client_type 'cos'
);
```

### 创建 HDFS Server

```
CREATE SERVER $hdfs_server_name
 FOREIGN DATA WRAPPER hdfs_fdw
 OPTIONS (
  address 'ofs://xxxxxx.chdfs.ap-guangzhou.myqcloud.com',
  appid '$appid',
  client_type 'hdfs'
);
```

### 创建融合桶 Server

```
CREATE SERVER $cosn_server_name
 FOREIGN DATA WRAPPER hdfs_fdw
 OPTIONS (
```

```
  address 'cosn://$bucketname',
  appid '$appid',
  access_keyid '$ak',
  secret_accesskey '$sk',
  region '$region',
  client_type 'cosn'
);
```

## 创建 Postgres Server

```
CREATE SERVER postgres_fdw_postgres_db
 FOREIGN DATA WRAPPER postgres_fdw
 OPTIONS (
  host '172.16.64.14',
  port '11345',
  dbname 'postgres'
);
```

## 配置 User Mapping

```
CREATE USER MAPPING FOR $TDSQL-A_USER SERVER $cos_server_name;
CREATE USER MAPPING FOR $TDSQL-A_USER SERVER $hdfs_server_name;
CREATE USER MAPPING FOR $TDSQL-A_USER SERVER $cosn_server_name;
```

## 创建外表

### 创建非分区表

```
CREATE FOREIGN TABLE test_csv(
  id INT,
  name TEXT
)
SERVER $ServerName
OPTIONS (
  FORMAT 'csv',
  FOLDERNAME '$数据目录/',
  distribute 'shard'
);
```

## 创建 TEXT 格式的表

```
CREATE FOREIGN TABLE test_text(
  id INT,
  name TEXT
)
SERVER $ServerName
OPTIONS (
  FORMAT 'text',
  DELIMITER '$列分隔符',
  FOLDERNAME '$数据目录/',
  distribute 'shard'
);
```

### 创建 ORC 格式的表

```
CREATE FOREIGN TABLE orc_table(
  a BIGINT,
  b TEXT,
  c FLOAT
)
SERVER $ServerName
OPTIONS (
  FORMAT 'orc',
  FOLDERNAME '$数据目录/',
  distribute 'shard'
);
```

### 创建分区表

```
CREATE FOREIGN TABLE login_logs_parquet(
  l_id TEXT,
  l_loginName TEXT,
  l_date TEXT,
  year TEXT,
  month TEXT
)
SERVER cosn_server
OPTIONS (
  FORMAT 'parquet',
  FOLDERNAME '$数据目录/',
  distribute 'shard',
  PARTITION 'year,month'
```

```
);
```

### 查询外表并将数据导入内部表

```
INSERT INTO 内部表 SELECT * FROM 外部表;
```

## dblink 外表

### 创建 dblink

```
CREATE DATABASE LINK abc
CONNECT TO "TBASE"
IDENTIFIED BY 'tbase2018'
USING '(DESCRIPTION =
  (ADDRESS = (PROTOCOL = TCP)(HOST = 172.16.0.17)(PORT = 1521))
  (CONNECT_DATA =
    (SERVER = DEDICATED)
    (SERVICE_NAME = ora12c)
  )
)';
```

### 查询 dblink 系统表

```
SELECT * FROM pg_dblink;
```

### 访问远程表

```
SELECT * FROM t1@abc;
```

### 删除 dblink

```
DROP DATABASE LINK abc;
```

## tbase_dblink

### 创建 postgres_fdw

```
CREATE EXTENSION postgres_fdw;
```

### 创建 dblink

```
SET dblink_types = 'postgresql';
CREATE DATABASE LINK tbase_dblink
CONNECT TO "tbase"
IDENTIFIED BY 'tbase@2017'
USING '(DESCRIPTION =
  (ADDRESS = (PROTOCOL = TCP)(HOST = 172.16.0.23)(PORT = 11379))
  (CONNECT_DATA =(SERVER = DEDICATED)(SERVICE_NAME = postgres)
(SCHEMA_NAME = public))
  (COLLATE = YES)(DEFAULT = YES)(NOT_NULL = YES)
)';
```

### 查询 dblink 系统表

```
SELECT * FROM pg_dblink WHERE dblinkname='tbase_dblink';
```

### 访问远程表

```
SELECT * FROM t2@tbase_dblink;
```

### 删除 dblink

```
DROP DATABASE LINK tbase_dblink;
```

## 使用 IF NOT EXISTS 创建表

```
CREATE TABLE IF NOT EXISTS t(id INT, mc TEXT);
```

## 指定模式创建表

```
CREATE TABLE public.t(id INT, mc TEXT);
```

## 使用查询结果创建数据表

```
CREATE TABLE t(id INT, mc TEXT) DISTRIBUTE BY SHARD(mc);
INSERT INTO t VALUES(1, 'tbase');
```

```
CREATE TABLE t_as AS SELECT * FROM t;
```

## 删除数据表

```
DROP TABLE t;
DROP TABLE public.t;
DROP TABLE IF EXISTS t;
DROP TABLE tbase_schema.t1 CASCADE;
```

## 删除分区子表

找到分区子表名，直接删除子表。

```
DROP TABLE t_time_range_part_1;
```

# 创建和删除索引

## 普通索引

```
postgres=# create index t_appoint_id_idx on t_appoint_col using
btree(id);
CREATE INDEX
```

## 唯一索引

创建唯一索引

```
postgres=# create unique index t_first_col_share_id_uidx on
t_first_col_share using btree(id);
CREATE INDEX
```

非 shard key 字段不能建立唯一索引

```
postgres=# create unique index t_first_col_share_nickname_uidx on
t_first_col_share using btree(nickname);
```

错误信息：

```
ERROR: Unique index of partitioned table must contain the hash/modulo
distribution column.
```

## 表达式索引

### 创建表t_upper

```
postgres=# create table t_upper(id int,mc text);
```

### 创建索引t_upper_mc

```
postgres=# create index t_upper_mc on t_upper(mc);
```

### 插入数据并分析

```
postgres=# insert into t_upper select t,md5(t::text) from
generate_series(1,10000) as t;
postgres=# analyze t_upper;
```

### 查询计划，使用表达式索引。

```
postgres=# explain select * from t_upper where upper(mc)=md5('1');
```

## 条件索引

### 创建表t_sex

```
postgres=# create table t_sex(id int,sex text);
```

### 创建索引t_sex_sex_idx

```
postgres=# create index t_sex_sex_idx on t_sex (sex);
```

### 插入数据并分析

```
postgres=# insert into t_sex select t,'男' from
generate_series(1,1000000) as t;
```

```
postgres=# insert into t_sex select t,'女' from generate_series(1,100) as
t;
postgres=# analyze t_sex;
```

查询计划，使用条件索引。

```
postgres=# explain select * from t_sex where sex ='女';
```

# gist 索引

创建表 t_trgm 并创建 gist 索引

```
postgres=# create table t_trgm (id int,trgm text,no_trgm text);
postgres=# create index t_trgm_trgm_idx on t_trgm using gist(trgm
gist_trgm_ops);
```

注意：仅行存表支持

# gin 索引

# pg_trgm 索引

删除索引 t_trgm_trgm_idx

```
postgres=# drop index t_trgm_trgm_idx;
```

创建 gin 索引 t_trgm_trgm_idx

```
postgres=# create index t_trgm_trgm_idx on t_trgm using gin(trgm
gin_trgm_ops);
```

注意：仅行存表支持

# jsonb 索引

创建表 t_jsonb 并创建 jsonb 索引

```
postgres=# create table t_jsonb(id int,f_jsonb jsonb);
postgres=# create index t_jsonb_f_jsonb_idx on t_jsonb using
gin(f_jsonb);
```

### 数组索引

创建表 t_array 并插入数据

```
postgres=# create table t_array(id int, mc text[]);
postgres=# insert into t_array select t,('{'||md5(t::text)||'}')::text[]
from generate_series(1,1000000) as t;
```

创建数组索引 t_array_mc_idx

```
postgres=# create index t_array_mc_idx on t_array using gin(mc);
```

## Btree_gin 任意字段索引

创建表 gin_mul 并插入数据

```
postgres=# create table gin_mul(f1 int, f2 int, f3 timestamp, f4 text,
f5 numeric, f6 text);
postgres=# insert into gin_mul select random()*5000, random()*6000,
now()+((3000060000*random())||' sec')::interval , md5(random()::text),
round((random()*100000)::numeric,2), md5(random()::text) from
generate_series(1,1000000);
```

创建 btree_gin 扩展并创建索引 gin_mul_gin_idx

```
postgres=# create extension btree_gin;
postgres=# create index gin_mul_gin_idx on gin_mul using
gin(f1,f2,f3,f4,f5,f6);
```

## 多字段索引

创建一个包含多个字段的索引，可以提高查询性能，尤其是在使用`OR`查询条件时。但需要注意，Bitmap扫描最多只支持两个不同字段的条件。

```
CREATE TABLE t_mul_idx (f1 int, f2 int, f3 int, f4 int);
CREATE INDEX t_mul_idx_idx ON t_mul_idx(f1, f2, f3);
```

### 多字段索引使用注意事项

- 当使用 `OR` 查询条件时，Bitmap 扫描最多支持两个不同字段的条件。

### 查询计划示例

- 查询使用多个字段的索引：

```
EXPLAIN SELECT * FROM t_mul_idx WHERE f1=1 OR f2=2;
```

- 当查询条件超过两个字段时，将使用Seq Scan而非Bitmap Heap Scan：

```
EXPLAIN SELECT * FROM t_mul_idx WHERE f1=1 OR f2=2 OR f3=3;
```

### 索引与性能

- 如果查询的返回字段全部在索引中，可以实现 Index Only Scan，减少 I/O 开销。
- 在插入操作中，多字段索引的性能通常优于单字段索引。

### 删除索引

```
DROP INDEX t_appoint_id_idx;
```

## 修改表结构

### 修改表名

```
postgres=# ALTER TABLE t RENAME TO tbase;
```

### 给表或字段添加注释

```
postgres=# COMMENT ON TABLE tbase IS 'TBase 分布式关系型数据库系统';
postgres=# COMMENT ON COLUMN tbase.nickname IS 'TBase 昵称是大象';
```

### 给表增加字段

```
postgres=# ALTER TABLE tbase ADD COLUMN age INTEGER;
```

### 修改字段类型

```
postgres=# ALTER TABLE tbase ALTER COLUMN age TYPE FLOAT8;
```

## 修改字段默认值

```
postgres=# ALTER TABLE tbase ALTER COLUMN age SET DEFAULT 0.0;
```

## 删除字段

```
postgres=# ALTER TABLE tbase DROP COLUMN age;
```

## 添加主键

```
postgres=# ALTER TABLE t ADD CONSTRAINT t_id_pkey PRIMARY KEY (id);
```

## 删除主键

```
postgres=# ALTER TABLE t DROP CONSTRAINT t_id_pkey;
```

## 重建主键

```
postgres=# CREATE UNIQUE INDEX CONCURRENTLY t_id_temp_idx ON t (id);
postgres=# ALTER TABLE t DROP CONSTRAINT t_pkey, ADD CONSTRAINT t_pkey
PRIMARY KEY USING INDEX t_id_temp_idx;
```

## 添加外键

```
CREATE TABLE t_p (f1 INT NOT NULL, f2 INT, PRIMARY KEY (f1));
CREATE TABLE t_f (f1 INT NOT NULL, f2 INT);
postgres=# ALTER TABLE t_f ADD CONSTRAINT t_f_f1_fkey FOREIGN KEY (f1)
REFERENCES t_p (f1);
```

## 删除外键

```
postgres=# ALTER TABLE t_f DROP CONSTRAINT t_f_f1_fkey;
```

## 修改表所属模式

```
postgres=# ALTER TABLE t SET SCHEMA public;
```

## 修改表所属用户

```
postgres=# ALTER TABLE tbase OWNER TO pgxz;
```

## 修改字段名

```
postgres=# ALTER TABLE tbase RENAME COLUMN city TO cityname;
```

## 修改表的填充率

```
postgres=# ALTER TABLE t1 SET (fillfactor=80);
```

## 添加触发器

## INSERT 触发器

```
postgres=# CREATE OR REPLACE FUNCTION t_trigger_insert_trigger_func()
RETURNS TRIGGER AS $body$
BEGIN
  IF NEW.f2 < 0 THEN
    NEW.f2 := 0;
  END IF;
  RETURN NEW;
END;
$body$
LANGUAGE plpgsql;

postgres=# CREATE TRIGGER t_trigger_insert_trigger BEFORE INSERT ON
t_trigger FOR EACH ROW EXECUTE PROCEDURE
t_trigger_insert_trigger_func();
```

## UPDATE 触发器

```
postgres=# CREATE OR REPLACE FUNCTION t_trigger_update_trigger_func()
RETURNS TRIGGER AS $body$
BEGIN
  IF NEW.f2 < 0 THEN
    NEW.f2 := OLD.f2;
  END IF;
```

```
  RETURN NEW;
END;
$body$
LANGUAGE plpgsql;

postgres=# CREATE TRIGGER t_trigger_update_trigger BEFORE UPDATE ON
t_trigger FOR EACH ROW EXECUTE PROCEDURE
t_trigger_update_trigger_func();
```

## DELETE 触发器

```
postgres=# CREATE OR REPLACE FUNCTION t_trigger_delete_trigger_func()
RETURNS TRIGGER AS $body$
BEGIN
  IF OLD.f2 = 0 THEN
    RETURN NULL;
  END IF;
  RETURN OLD;
END;
$body$
LANGUAGE plpgsql;

postgres=# CREATE TRIGGER t_trigger_delete_trigger BEFORE DELETE ON
t_trigger FOR EACH ROW EXECUTE PROCEDURE
t_trigger_delete_trigger_func();
```

## 多个事件触发器

```
postgres=# CREATE TABLE t_trigger_mulevent(f1 INT, f2 INT);
CREATE OR REPLACE FUNCTION t_trigger_mulevent_func() RETURNS TRIGGER AS
$body$
BEGIN
  IF NEW.f2 < 0 THEN
    NEW.f2 := 0;
  END IF;
  RETURN NEW;
END;
$body$
LANGUAGE plpgsql;

postgres=# CREATE TRIGGER t_trigger_insert_update_trigger BEFORE INSERT
OR UPDATE ON t_trigger_mulevent FOR EACH ROW EXECUTE PROCEDURE
```

```
t_trigger_mulevent_func();
```

### 删除触发器

```
postgres=# DROP TRIGGER t_trigger_insert_update_trigger ON
t_trigger_mulevent;
postgres=# DROP FUNCTION t_trigger_mulevent_func();
```

## 视图创建和删除

### 创建视图

```
postgres=# create view t_range_view as select * from t_range;
CREATE VIEW
postgres=# select * from t_range_view;
```

| f1 | f2 | f3 | f4 |
| --- | --- | --- | --- |
| 1 | 2017-09-27 23:17:39.674318 | 1 | |
| 2 | 2017-09-27 23:17:39.674318 | 50 | |
| 2 | 2017-09-27 23:17:39.674318 | 110 | |
| 1 | 2017-09-27 23:39:45.841093 | 151 | |
| 3 | 2017-09-27 23:17:39.674318 | 100 | |

(5 rows)

#### 数据类型重定义

```
postgres=# create view t_range_view as select f1,f2::date from t_range;
CREATE VIEW
postgres=# select * from t_range_view;
```

| f1 | f2 |
| --- | --- |
| 1 | 2017-09-27 |
| 2 | 2017-09-27 |

| | |
|---|---|
| 2 | 2017-09-27 |
| 1 | 2017-09-27 |
| 3 | 2017-09-27 |

(5 rows)

**数据类型重定义,以及取别名。**

```
postgres=# create view t_range_view as select f1,f2::date as mydate from
t_range;
CREATE VIEW
postgres=# select * from t_range_view;
```

| f1 | mydate |
|---|---|
| 1 | 2017-09-27 |
| 2 | 2017-09-27 |
| 2 | 2017-09-27 |
| 1 | 2017-09-27 |
| 3 | 2017-09-27 |

(5 rows)

**tbase 支持视图引用表或字段改名联动，不受影响。**

```
postgres=# \d+ t_view
View "tbase.t_view"
 Column | Type | Collation | Nullable | Default | Storage | Description
--------+---------+-----------+----------+---------+----------+---------
----
 id     | integer |           |          |         | plain    |
 mc     | text    |           |          |         | extended |
View definition:
 SELECT t.id,
 t.mc
 FROM t;
```

```
postgres=# alter table t rename to t_new;
ALTER TABLE
Time: 62.875 ms

postgres=# alter table t_new rename mc to mc_new;
ALTER TABLE
Time: 22.081 ms

postgres=# \d+ t_view
View "tbase.t_view"
 Column | Type | Collation | Nullable | Default | Storage | Description
--------+---------+-----------+----------+---------+----------+---------
----
 id     | integer |           |          |         | plain    |
 mc     | text    |           |          |         | extended |
View definition:
 SELECT t_new.id,
 t_new.mc_new AS mc
 FROM t_new;
```

## 删除视图

```
postgres=# drop table t;
DROP TABLE

postgres=# create table t (id int,mc text);
CREATE TABLE
postgres=# create view t_view as select * from t;
CREATE VIEW
postgres=# create view t_view_1 as select * from t_view;
CREATE VIEW
postgres=# create view t_view_2 as select * from t_view;
CREATE VIEW

postgres=# drop view t_view_2;
DROP VIEW

#使用cascade强制删除依赖对象

postgres=# drop view t_view;
ERROR:  cannot drop view t_view because other objects depend on it
DETAIL:  view t_view_1 depends on view t_view
```

```
HINT:  Use DROP ... CASCADE to drop the dependent objects too.
postgres=# drop view t_view cascade;
NOTICE:  drop cascades to view t_view_1
DROP VIEW
```

# select语句

最近更新时间：2024-07-24 17:02:21

## 访问函数

```
postgres=# select md5(random()::text);
```

结果：

```
 md5
----------------------------------
 3eb6c0c8f8355f0b0f0cad7a8f0f7491
```

## 数据排序

### 按某一列排序

```
postgres=# INSERT into tbase (nickname) VALUES('TBase 好');
postgres=# INSERT into tbase (id,nickname) VALUES(1,'TBase 分布式数据库的时
代来了');
postgres=# select * from tbase order by id;
```

结果：

```
 id | nickname
----+------------------------------
  1 | hello TBase
  1 | TBase 分布式数据库的时代来了
  2 | TBase 好
(3 rows)
```

### 按第一列排序

```
postgres=# select * from tbase order by 1;
```

### 按 id 升序排序，再按 nickname 降序排序

```
postgres=# select * from tbase order by id, nickname desc;
```

### 随机排序

```
postgres=# select * from tbase order by random();
```

### 计算排序

```
postgres=# select * from tbase order by md5(nickname);
```

### 排序使用子查询

```
postgres=# select * from tbase order by (select id from tbase order by
random() limit 1);
```

### null 值排序结果处理

```
postgres=# insert into tbase values(4,null);
postgres=# select * from tbase order by nickname nulls first;
postgres=# select * from tbase order by nickname nulls last;
```

### 按拼音排序

```
postgres=# select * from (values ('张三'), ('李四'),('陈五')) t(myname)
order by myname;
postgres=# select * from (values ('张三'), ('李四'),('陈五')) t(myname)
order by convert(myname::bytea,'UTF-8','GBK');
postgres=# select * from (values ('张三'), ('李四'),('陈五')) t(myname)
order by convert_to(myname,'GBK');
postgres=# select * from (values ('张三'), ('李四'),('陈五')) t(myname)
order by myname collate "zh_CN.utf8";
```

## where 条件使用

### 单条件查询

```
postgres=# select * from tbase where id=1;
```

## 多条件 and

```
postgres=# select * from tbase where id=1 and nickname like '%h%';
```

## 多条件 or

```
postgres=# select * from tbase where id=2 or nickname like '%h%';
```

## ilike 不区分大小写匹配

```
postgres=# create table t_ilike(id int,mc text);
postgres=# insert into t_ilike values(1,'tbase'),(2,'TBase');
postgres=# select * from t_ilike where mc ilike '%tb%';
```

## where 条件支持子查询

```
postgres=# select * from tbase where id=(select (random()*2)::integer
from tbase order by random() limit 1);
```

## null 值查询方法

```
postgres=# select * from tbase where nickname is null;
postgres=# select * from tbase where nickname is not null;
```

## exists，只要有记录返回就为真

```
postgres=# create table t_exists1(id int,mc text);
postgres=# insert into t_exists1 values(1,'tbase'),(2,'TBase');
postgres=# create table t_exists2(id int,mc text);
postgres=# insert into t_exists2 values(1,'tbase'),(1,'TBase');
postgres=# select * from t_exists1 where exists(select 1 from t_exists2
where t_exists1.id=t_exists2.id);
```

## exists 等价写法

```
postgres=# select t_exists1.* from t_exists1, (select distinct id from
t_exists2) as t where t_exists1.id=t.id;
```

# 分页查询

默认从第一条开始，返回一条记录：

```
postgres=# select * from tbase limit 1;
```

结果：

```
 id | nickname
----+-------------
 1 | hello TBase
(1 row)
```

使用 `offset` 指定从第几条开始，0 表示第一条开始，返回 1 条记录：

```
postgres=# select * from tbase limit 1 offset 0;
```

结果：

```
 id | nickname
----+-------------
 1 | hello TBase
(1 row)
```

从第 3 条开始，返回两条记录：

```
postgres=# select * from tbase limit 1 offset 2;
```

结果：

```
 id | nickname
----+-----------------
 1 | TBase 分布式数据库的时代来了
(1 row)
```

使用 `order by` 可以获得一个有序的结果：

```
postgres=# select * from tbase order by id limit 1 offset 2;
```

结果：

```
 id | nickname
----+-----------
  2 | TBase 好
(1 row)
```

## 合并多个查询结果

不过滤重复的记录：

```
postgres=# select * from tbase union all select * from t_appoint_col;
```

结果：

```
 id | nickname
----+-----------------
  1 | hello TBase
  2 | TBase 好
  1 | TBase 分布式数据库的时代来了
  1 | hello TBase
(4 rows)
```

过滤重复的记录：

```
postgres=# select * from tbase union select * from t_appoint_col;
```

结果：

```
 id | nickname
----+-----------------
  1 | TBase 分布式数据库的时代来了
  1 | hello TBase
  2 | TBase 好
(3 rows)
```

## 返回两个结果的交集

```
postgres=# select * from t_intersect1 INTERSECT select * from
```

```
t_intersect2;
```

结果：

```
 id | mc
----+-------
 1 | tbase
(1 row)
```

## 返回两个结果的差集

```
postgres=# select * from t_except1 except select * from t_except2;
```

结果：

```
 id | mc
----+-------
 2 | tbase
(1 row)
```

## any 用法

只需要大于其中一个值即为真：

```
postgres=# select * from t_any where id > any (select 1 union select 3);
```

结果：

```
 id | mc
----+-------
 2 | TBase
(1 row)
```

## all 用法

需要大于所有值才为真：

```
postgres=# select * from t_all where id > all (select 1 union select 2);
```

结果：

```
 id | mc
----+-------
  3 | TBase
(1 row)
```

# 聚集查询

## 统计记录数

```
postgres=# select count(1) from tbase;
```

结果：

```
 count
-------
     3
(1 row)
```

## 统计不重复值的记录表

```
postgres=# select count(distinct id) from tbase;
```

结果：

```
 count
-------
     2
(1 row)
```

## 求和

```
postgres=# select sum(id) from tbase;
```

结果：

```
 sum
-----
```

```
 4
(1 row)
```

## 求最大值

```
postgres=# select max(id) from tbase;
```

结果：

```
 max
-----
 2
(1 row)
```

## 求最小值

```
postgres=# select min(id) from tbase;
```

结果：

```
 min
-----
 1
(1 row)
```

## 求平均值

```
postgres=# select avg(id) from tbase;
```

结果：

```
        avg
--------------------
 1.3333333333333333
(1 row)
```

## 分组字段合并成一个字符串

```
postgres=# create table t1(f1 int, f2 text, f3 text);
postgres=# insert into t1 values(1,'a','a');
postgres=# insert into t1 values(1,'b','b');
postgres=# insert into t1 values(2,'a','a');
postgres=# select f1, string_agg(f2, ',') from t1 group by f1;
```

**结果：**

```
 f1 | string_agg
----+------------
 1  | a,b
 2  | a
(2 rows)
```

## 去重和自定义聚合函数

**在聚合函数中使用 `DISTINCT` 和 `ORDER BY` 需要满足特定条件，可以自定义函数来实现更灵活的聚合。**

# 多表关联

## 内连接

```
postgres=# select * from tbase inner join t_appoint_col on
tbase.id=t_appoint_col.id;
```

**结果：**

```
 id | nickname | id | nickname
----+----------+----+----------
 1  | hello TBase | 1  | hello TBase
 1  | TBase 分布式数据库的时代来了 | 1  | hello TBase
(2 rows)
```

## 左外连接

```
postgres=# select * from tbase left join t_appoint_col on
tbase.id=t_appoint_col.id;
```

**结果：**

```
 id | nickname    | id | nickname
----+-------------+----+-------------
 1  | hello TBase | 1  | hello TBase
 2  | TBase 好    |    |
 1  | TBase 分布式数据库的时代来了 | 1  | hello TBase
(3 rows)
```

## 右外连接

```
postgres=# select * from tbase right join t_appoint_col on
tbase.id=t_appoint_col.id;
```

结果：

```
 id | nickname                      | id | nickname
----+-------------------------------+----+-------------
 1  | TBase 分布式数据库的时代来了 | 1  | hello TBase
 1  | hello TBase                   | 1  | hello TBase
    |                               | 5  | Power TBase
(3 rows)
```

## 全连接

```
postgres=# select * from tbase full join t_appoint_col on
tbase.id=t_appoint_col.id;
```

结果：

```
 id | nickname    | id | nickname
----+-------------+----+-------------
 1  | hello TBase | 1  | hello TBase
 2  | TBase 好    |    |
 1  | TBase 分布式数据库的时代来了 | 1  | hello TBase
    |             | 5  | Power TBase
(4 rows)
```

# 聚合函数并发计算

## 单核计算

```
postgres=# set max_parallel_workers_per_gather to 0;
postgres=# select count(1) from t_count;
```

### 二核并行

```
postgres=# set max_parallel_workers_per_gather to 2;
postgres=# select count(1) from t_count;
```

### 四核并行

```
postgres=# set max_parallel_workers_per_gather to 4;
postgres=# select count(1) from t_count;
```

## not in 子句包含 null 的情况

```
postgres=# create table t_not_in(id int, mc text);
NOTICE: Replica identity is needed for shard table, please add to this
table through "alter table" command.
CREATE TABLE
postgres=# insert into t_not_in values(1, 'tbase'), (2, 'pgxz');
INSERT 0 2
postgres=# select * from t_not_in where id not in (3, 5);
```

**结果：**

```
 id | mc
----+-------
 1 | tbase
 2 | pgxz
(2 rows)
```

```
postgres=# select * from t_not_in where id not in (3, 5, null);
```

**结果：无输出行**

## 只查询特定数据节点（dn）的数据

```
postgres=# create table t_direct(id int, mc text);
NOTICE: Replica identity is needed for shard table, please add to this
table through "alter table" command.
CREATE TABLE
postgres=# insert into t_direct values(1, 'tbase'), (3, 'pgxz');
INSERT 0 2
postgres=# EXECUTE DIRECT ON (dn001) 'select * from t_direct';
```

**结果：**

```
 id | mc
----+-------
 1 | tbase
(1 row)
```

```
postgres=# EXECUTE DIRECT ON (dn002) 'select * from t_direct';
```

**结果：**

```
 id | mc
----+------
 3 | pgxz
(1 row)
```

**查询所有节点的数据：**

```
postgres=# select * from t_direct;
```

**结果：**

```
 id | mc
----+-------
 1 | tbase
 3 | pgxz
(2 rows)
```

# 特殊应用

## 多行变成单行

```
postgres=# create table t_mulcol_tosimplecol(id int, mc text);
NOTICE: Replica identity is needed for shard table, please add to this
table through "alter table" command.
CREATE TABLE
postgres=# insert into t_mulcol_tosimplecol values(1, 'tbase'), (2,
'TBase');
INSERT 0 2
postgres=# select array_to_string(array(select mc from
t_mulcol_tosimplecol), ',');
```

**结果：**

```
 array_to_string
-----------------
 tbase,TBase
(1 row)
```

## 一列变成多行

```
postgres=# create table t_col_to_mulrow(id int, mc text);
NOTICE: Replica identity is needed for shard table, please add to this
table through "alter table" command.
CREATE TABLE
postgres=# insert into t_col_to_mulrow values(1, 'tbase,TBase');
INSERT 0 1
postgres=# select regexp_split_to_table((select mc from t_col_to_mulrow
where id=1 limit 1), ',');
```

**结果：**

```
 regexp_split_to_table
-----------------------
 tbase
 TBase
(2 rows)
```

# 查询记录所在数据节点

```
postgres=# select xc_node_id, * from t1;
```

结果：

```
 xc_node_id | f1 | f2
------------+----+----
 2142761564 | 1  | 3
 2142761564 | 1  | 3
(2 rows)
```

查询并映射节点名称：

```
postgres=# select t1.xc_node_id, pgxc_node.node_name, t1.* from t1,
pgxc_node where t1.xc_node_id=pgxc_node.node_id;
```

结果：

```
 xc_node_id | node_name | f1 | f2
------------+-----------+----+----
 2142761564 | dn001     | 1  | 3
 2142761564 | dn001     | 1  | 3
(2 rows)
```

# grouping sets/rollup/cube 用法

## group by 用法

创建销售明细表并插入数据：

```
create table t_grouping(id int, dep varchar(20), product varchar(20),
num int);
insert into t_grouping values(1, '业务 1 部', '手机', 90);
-- 更多插入数据 ...
```

按部门和产品进行分组汇总：

```
postgres=# select dep, product, sum(num) from t_grouping group by dep,
product order by dep, product;
```

结果：

```
 dep        | product    | sum
------------+------------+-----
```

```
 业务 1 部 | 电脑        | 80
 业务 1 部 | 手机        | 160
 业务 2 部 | 电脑        | 120
 业务 2 部 | 手机        | 50
 业务 3 部 | 电脑        | 80
 业务 3 部 | 手机        | 160
```

使用 grouping sets 进行分组：

```
postgres=# select dep, product, sum(num) from t_grouping group by
grouping sets((dep), (product), ());
```

结果：

```
 dep        | product    | sum
------------+------------+-----
 业务 1 部 | （空）       | 240
 业务 2 部 | （空）       | 170
 业务 3 部 | （空）       | 240
 电脑       | （空）       | 280
 手机       | （空）       | 370
（空）       | （空）       | 650
```

使用 rollup 和 cube：

```
postgres=# select dep, product, sum(num) from t_grouping group by
rollup((dep), (product));
postgres=# select dep, product, sum(num) from t_grouping group by
cube((dep), (product));
```

结果与 grouping sets 相同。

# PREPARE 预备使用

## 创建一个预备语句

```
postgres=# create table t1(f1 int, f2 int);
CREATE TABLE
postgres=# insert into t1 values(1, 1), (2, 2);
COPY 2
```

```
postgres=# PREPARE usrrptplan (int) AS SELECT * FROM t1 WHERE f1=$1;
PREPARE
postgres=# EXECUTE usrrptplan(1);
```

**结果：**

```
 f1 | f2
----+----
 1  | 1
(1 row)
```

## 释放一个预备语句

```
postgres=# DEALLOCATE usrrptplan;
DEALLOCATE
postgres=# EXECUTE usrrptplan(1);
ERROR: prepared statement "usrrptplan" does not exist
```

# insert 语句

最近更新时间：2024-07-24 17:02:21

## SQL 插入语句

### 插入单条记录

- 指定所有字段：

```
postgres=# insert into tbase(id, nickname) values(1, 'hello TBase');
```

结果：`INSERT 0 1`

- 指定某些字段，未指定字段将使用默认值：

```
postgres=# insert into tbase(nickname) values('TBase 好');
```

结果：`INSERT 0 1`

- 不指定字段，则默认所有字段与建表时一致：

```
postgres=# insert into tbase values(nextval('t_id_seq'::regclass),
'TBase 好');
```

结果：`INSERT 0 1`

- 字段顺序可以任意排列：

```
postgres=# insert into tbase(nickname, id) values('TBase swap', 5);
```

结果：`INSERT 0 1`

- 使用 `default` 关键字，即值为建表时指定的默认值方式：

```
postgres=# insert into tbase(id, nickname) values(default, 'TBase
default');
```

结果：`INSERT 0 1`

- 插入记录后的数据：

```
postgres=# select * from tbase;
```

| id | nickname |
| --- | --- |
| 1 | hello TBase |
| 2 | TBase 好 |
| 5 | TBase swap |
| 3 | TBase 好 |
| 4 | TBase default |

## 插入多数记录

- 插入多条记录：

```
postgres=# insert into tbase(id, nickname) values(1, 'hello TBase'),
(2, 'TBase 好');
```

结果： `INSERT 0 2`

- 插入后查询结果：

```
postgres=# select * from tbase;
```

| id | nickname |
| --- | --- |
| 1 | hello TBase |
| 2 | TBase 好 |

## 使用子查询插入数据

- 使用子查询插入数据：

```
postgres=# insert into tbase(id, nickname) values(1, (select relname
from pg_class limit 1));
```

结果： `INSERT 0 1`

- 插入后查询结果：

```
postgres=# select * from tbase;
```

| id | nickname |
| --- | --- |
| 1 | pg_statistic |

## 从另外一个表取数据进行批量插入

- 从另一个表批量插入数据：

```
postgres=# insert into tbase(nickname) select relname from pg_class
limit 3;
```

结果：`INSERT 0 3`

- 插入后查询结果：

```
postgres=# select * from tbase;
```

| id | nickname |
| --- | --- |
| 5 | pg_type |
| 6 | pg_toast_2619 |
| 4 | pg_statistic |

## 大批量的生成数据

- 使用 `generate_series` 函数大批量生成数据：

```
postgres=# insert into tbase select t, md5(random()::text) from
generate_series(1,10000) as t;
```

结果：`INSERT 0 10000`

- 插入后计数：

```
postgres=# select count(1) from tbase;
```

| count |
| --- |
| 10000 |

## 返回插入数据，轻松获取插入记录的 serial 值

- 插入记录并返回所有字段：

```
postgres=# insert into tbase(nickname) values('TBase 好') returning *;
```

| id | nickname |
| --- | --- |
| 7 | TBase 好 |

- 指定返回字段：

```
postgres=# insert into tbase(nickname) values('hello TBase') returning
id;
```

| id |
| --- |
| 8 |

# insert..update 更新

- 使用 `ON CONFLICT` 子句进行更新：

```
postgres=# insert into t values(1,'pgxz') ON CONFLICT (id) DO UPDATE
SET nc = 'tbase';
```

结果：`INSERT 0 1`

- 更新后查询结果：

```
postgres=# select * from t;
```

| id | nc |
| --- | --- |
| 1 | tbase |

# insert all

- 使用 `insert all` 语句：

```
postgres=# create table t5(f1 int, f2 int);
postgres=# create table t6(f1 int, f2 int);
```

```
postgres=# insert all into t5 values(1, 1) into t6 values(1, 1) select
1 as f1, 1 as f2;
```

结果：`INSERT 0 2`

- 查询 t5 表：

```
postgres=# select * from t5;
```

| f1 | f2 |
| --- | --- |
| 1 | 1 |

- 查询 t6 表：

```
postgres=# select * from t6;
```

| f1 | f2 |
| --- | --- |
| 1 | 1 |

# update 语句

最近更新时间：2024-07-24 17:02:21

## 单表更新

```sql
update tbase set nickname ='Hello TBase' where id=1;
```

**条件表达方法**

```sql
update tbase set nickname = 'Good TBase' where nickname is null;
```

**查询结果：**

```sql
select * from tbase;
```

| id | nickname |
| --- | --- |
| 2 | TBase 好 |
| 1 | Hello TBase |
| 3 | Good TBase |

**分布键不能更新：**

```sql
update t2 set f1=1 where f2=1;
```

**错误信息：**

```
ERROR: Distributed column "f1" can't be updated in current version
```

## 多表关联更新

```sql
update tbase set nickname ='Good TBase' from t_appoint_col where
t_appoint_col.id=tbase.id;
```

**查询结果：**

```
select * from tbase;
```

| id | nickname |
| --- | --- |
| 2 | TBase 好 |

## 返回更新的数据

```
update tbase set nickname = nickname where id = (random()*2)::integer
returning id, nickname;
```

返回结果：

| id | nickname |
| --- | --- |
| 2 | TBase 好 |

## 多列匹配更新

```
update tbase set ( nickname , age ) = ( 'TBase 好' ,
(random()*2)::integer );
```

查询结果：

```
select * from tbase;
```

| id | nickname | age |
| --- | --- | --- |
| 1 | TBase 好 | 2 |
| 2 | TBase 好 | 0 |

## Shard Key 不允许更新

```
update tbase set id=8 where id=1;
```

错误信息：

```
ERROR: Distribute column or partition column can't be updated in current
version
```

# delete 语句

最近更新时间：2024-07-24 17:02:21

## 带条件删除

```sql
select * from tbase;
```

| id | nickname |
| --- | --- |
| 2 | TBase 好 |
| 1 | Hello TBase |
| 3 | |
| 4 | TBase good |

```sql
delete from tbase where id=4;
```

### null 条件的表达方式

```sql
delete from tbase where nickname is null;
```

```sql
select * from tbase;
```

| id | nickname |
| --- | --- |
| 2 | TBase 好 |
| 1 | Hello TBase |

## 多表关联删除数据

```sql
set prefer_olap to on;
```

```sql
delete from tbase using t_appoint_col where tbase.id=t_appoint_col.id;
```

```
select * from tbase;
```

| id | nickname |
| --- | --- |
| 2 | TBase 好 |

## 返回删除数据

```
delete from tbase returning *;
```

| id | nickname |
| --- | --- |
| 2 | TBase 好 |

## 删除所有数据

```
insert into tbase select t,random()::text from generate_series(1,100000)
as t;
```

```
truncate table tbase;
```

# 游标的使用

最近更新时间：2024-07-24 17:02:21

## 定义一个游标

```
begin;
```

```
DECLARE tbase_cur SCROLL CURSOR FOR SELECT * from tbase ORDER BY id;
```

> ⚠ **注意：**
> 游标需要放在一个事务中使用

## 提取下一行数据

```
FETCH NEXT from tbase_cur ;
```

| id | nickname |
| --- | --- |
| 1 | hello TBase |

```
FETCH NEXT from tbase_cur ;
```

| id | nickname |
| --- | --- |
| 2 | TBase 好 |

## 提取前一行数据

```
FETCH PRIOR from tbase_cur ;
```

| id | nickname |
| --- | --- |
| 1 | hello TBase |

```
FETCH PRIOR from tbase_cur ;
```

## 提取最后一行

```
FETCH LAST from tbase_cur ;
```

| id | nickname |
| --- | --- |
| 5 | TBase swap |

## 提取第一行

```
FETCH FIRST from tbase_cur ;
```

| id | nickname |
| --- | --- |
| 1 | hello TBase |

## 提取该查询的第 x 行

```
FETCH ABSOLUTE 2 from tbase_cur ;
```

| id | nickname |
| --- | --- |
| 2 | TBase 好 |

```
FETCH ABSOLUTE -1 from tbase_cur ;
```

| id | nickname |
| --- | --- |
| 5 | TBase swap |

## 提取当前位置后的第 x 行

```
FETCH ABSOLUTE 1 from tbase_cur ;
```

| id | nickname |
| --- | --- |
| 1 | hello TBase |

```
FETCH RELATIVE 2 from tbase_cur ;
```

| id | nickname |
| --- | --- |
| 3 | TBase 好 |

## 向前提取 x 行数据

```
FETCH FORWARD 2 from tbase_cur ;
```

| id | nickname |
| --- | --- |
| 1 | hello TBase |
| 2 | TBase 好 |

```
FETCH FORWARD 2 from tbase_cur ;
```

| id | nickname |
| --- | --- |
| 3 | TBase 好 |
| 4 | TBase default |

## 向前提取剩下的所有数据

```
FETCH FORWARD all from tbase_cur ;
```

| id | nickname |
| --- | --- |
| 3 | TBase 好 |
| 4 | TBase default |
| 5 | TBase swap |

## 向后提取 x 行数据

```
FETCH BACKWARD 2 from tbase_cur ;
```

| id | nickname |
| --- | --- |
| 5 | TBase swap |
| 4 | TBase default |

## 向后提取剩下的所有数据

```
FETCH BACKWARD all from tbase_cur ;
```

| id | nickname |
| --- | --- |
| 3 | TBase 好 |
| 2 | TBase 好 |
| 1 | hello TBase |

# copy 的使用

最近更新时间：2024-07-26 11:44:21

COPY 用于 TBase 表和标准文件系统文件之间数据互相复制。COPY TO 可以把一个表的内容复制到一个文件，COPY FROM 可以从一个文件复制数据到一个表（数据以追加形式入库），COPY TO 也能复制一个 SELECT 查询的结果到一个文件。如果指定了一个列表，COPY 将只把指定列的数据复制到文件或者从文件复制数据到指定列。如果表中有列不在列表中，COPY FROM 将会为那些列插入默认值。 使用 COPY 时 TBase 服务器直接从“本地”一个文件读取或者写入到一个文件。该文件必须是 TBase 用户（运行服务器的用户 ID）可访问的并且应该以服务器的视角来指定其名称。

## 实验表结构及数据

```sql
\d+ t
```

| Column | Type | Modifiers | Storage | Stats target | Description |
| --- | --- | --- | --- | --- | --- |
| f1 | integer | not null | plain | | |
| f2 | character | not null | extended | | |
| f3 | timestamp | default now() | plain | | |
| f4 | integer | | plain | | |

数据测试过程可以行再录入修改

```sql
select * from t;
```

| f1 | f2 | f3 | f4 |
| --- | --- | --- | --- |
| 3 | pgxz | 2017-10-28 18:24:05.645691 | |
| 1 | Tbase | | 7 |
| 2 | | 2017-10-28 18:24:05.643102 | 3 |

# COPY TO 用法详解--复制数据到文件中

## 导出所有列

```
copy public.t to '/data/pgxz/t.txt';
```

```
cat /data/pgxz/t.txt
```

默认生成的文件内容为表的所有列，列与列之间使用 tab 分隔开来。NULL 值生成的值为\N。

## 导出部分列

```
copy public.t(f1,f2) to '/data/pgxz/t.txt';
```

```
cat /data/pgxz/t.txt
```

只导出 f1 和 f2 列

## 导出查询结果

```
copy (select f2,f3 from public.t order by f3) to '/data/pgxz/t.txt';
```

```
cat /data/pgxz/t.txt
```

查询可以是任何复杂查询

## 指定生成文件格式

生成 csv 格式

```
```sql
copy public.t to '/data/pgxz/t.txt' with csv;
```

```
cat /data/pgxz/t.txt
```

生成二进制格式

```
copy public.t to '/data/pgxz/t.txt' with binary;
```

默认为 TEXT 格式
```

## 使用 delimiter 指定列与列之间的分隔符

```
copy public.t to '/data/pgxz/t.txt' with delimiter '@';
```

```
cat /data/pgxz/t.txt
```

指定分隔文件各列的字符。文本格式中默认是一个制表符，而 CSV 格式中默认是一个逗号。分隔符必须是一个单一的单字节字符，即汉字是不支持的。使用 binary 格式时不允许这个选项。

## NULL 值的处理

```
copy public.t to '/data/pgxz/t.txt' with NULL 'NULL';
```

```
cat /data/pgxz/t.txt
```

记录值为 NULL 时导出为 NULL 字符。使用 binary 格式时不允许这个选项。

## 生成列标题名

```
使用 CSV 格式和 HEADER 选项导出表：
```sql
copy public.t to '/data/pgxz/t.txt' with csv HEADER;
```

导出结果查看：

```
cat /data/pgxz/t.txt
```

```
f1,f2,f3,f4
1,Tbase,,7
2,pgxc,2017-10-28 18:24:05.643102,3
3,pgxz,2017-10-28 18:24:05.645691,
```

尝试在非 CSV 模式下使用 HEADER 选项会导致错误：

```
copy public.t to '/data/pgxz/t.txt' with HEADER;
```
```

```
ERROR: COPY HEADER available only in CSV mode
```

## 导出 oids 系统列

**创建一个带有 oids 的新表：**

```
drop table t;
CREATE TABLE t (
    f1 integer NOT NULL,
    f2 text NOT NULL,
    f3 timestamp without time zone,
    f4 integer
) with oids DISTRIBUTE BY SHARD (f1);
```

**导入数据并包含 oids：**

```
copy t from '/data/pgxz/t.txt' with csv;
```

**查询确认导入结果：**

```
select * from t;
```

| f1 | f2 | f3 | f4 |
| --- | --- | --- | --- |
| 1 | Tbase | | 7 |
| 2 | pg'", xc | 2017-10-28 18:24:05.643102 | 3 |
| 3 | pgxz | 2017-10-28 18:24:05.645691 | |

**使用 oids 选项导出数据：**

```
copy t to '/data/pgxz/t.txt' with oids;
```

**导出结果查看：**

```
cat /data/pgxz/t.txt
```

```
35055 1 Tbase \N 7
```

```
35056 2 pg'", xc 2017-10-28 18:24:05.643102 3
35177 3 pgxz 2017-10-28 18:24:05.645691 \N
```

## 使用 quote 自定义引用字符

默认 CSV 导出引用字符为双引号：

```
copy t to '/data/pgxz/t.txt' with csv;
```

导出结果查看：

```
cat /data/pgxz/t.txt
```

```
1,Tbase,,7
2,"pg'"", xc%",2017-10-28 18:24:05.643102,3
3,pgxz,2017-10-28 18:24:05.645691,
```

自定义引用字符为百分号：

```
copy t to '/data/pgxz/t.txt' with quote '%' csv;
```

导出结果查看：

```
cat /data/pgxz/t.txt
```

```
1,Tbase,,7
2,%pg'", xc%%%,2017-10-28 18:24:05.643102,3
3,pgxz,2017-10-28 18:24:05.645691,
```

## 使用 escape 自定义逃逸符

不指定 escape 时，默认与 QUOTE 值相同：

```
copy t to '/data/pgxz/t.txt' with quote '%' csv;
```

自定义 escape 为 '@'：

```
copy t to '/data/pgxz/t.txt' with quote '%' escape '@' csv;
```

**导出结果查看：**

```
cat /data/pgxz/t.txt
```

```
1,Tbase,,7
2,%pg'", xc@%%,2017-10-28 18:24:05.643102,3
3,pgxz,2017-10-28 18:24:05.645691,
```

**尝试使用多字节字符作为 escape 会导致错误：**

```
copy t to '/data/pgxz/t.txt' with quote '%' escape '@@' csv;
```

```
ERROR: COPY escape must be a single one-byte character
```

## 强制给某个列添加引用字符

**使用 CSV 格式和 `force_quote` 选项导出指定列：**

```sql
copy t to '/data/pgxz/t.txt' (format 'csv', force_quote (f1,f2));
```

**导出结果查看：**

```
cat /data/pgxz/t.txt
```

```
"1","Tbase",,7
"2","pg'"", xc%",2017-10-28 18:24:05.643102,3
"3","pgxz",2017-10-28 18:24:05.645691,
```

**指定多列强制添加引用字符，列顺序可以任意排列：**

```
copy t to '/data/pgxz/t.txt' (format 'csv', force_quote (f1,f4,f3,f2));
```

**导出结果查看：**

```
cat /data/pgxz/t.txt
```

```
"1","Tbase",,"7"
"2","pg'"", xc%","2017-10-28 18:24:05.643102","3"
"3","pgxz","2017-10-28 18:24:05.645691",
```

尝试在非 CSV 模式下使用 `force_quote` 选项会导致错误：

```
copy t to '/data/pgxz/t.txt' (format 'text', force_quote (f1,f2,f3,f4));
```

```
ERROR: COPY force quote available only in CSV mode
```

## 使用 encoding 指定导出文件内容编码

使用不同编码导出文件：

```
copy t to '/data/pgxz/t.csv' (encoding utf8);
copy t to '/data/pgxz/t.csv' (encoding gbk);
```

使用 `set_client_encoding` 设置客户端编码并导出 CSV 文件：

```
set client_encoding to gbk;
copy t to '/data/pgxz/t.csv' with csv;
```

## 使用多字节做分隔符

腾讯云数据仓库 TCHouse-P 5.06.2.1 以上版本支持多字节分隔符。
创建新表并插入数据：

```
create table t_position_copy(f1 int, f2 varchar(10), f3 int);
insert into t_position_copy values(123, '阿弟', 2021);
```

使用多字节分隔符导出数据：

```
copy t_position_copy to '/data/tbase/t_position_copy_mul_delimiter.text'
with delimiter '@\u0026';
```

导出结果查看：

```
cat /data/tbase/t_position_copy_mul_delimiter.text
```

```
123@\u0026阿弟@\u00262021
```

# COPY FROM 用法详解--复制文件内容到数据表中

## 导入所有列

```
cat /data/pgxz/t.txt
```

```
truncate table t;
copy t from '/data/pgxz/t.txt';
```

```
select * from t;
```

## 导入部分指定列

将指定列的数据复制到文件：
```sql
copy t(f1,f2) to '/data/pgxz/t.txt';
```

查看文件内容：

```
cat /data/pgxz/t.txt
```

```
1 Tbase
2 pg'", xc%
3 pgxz
```

清空表 `t` 并从文件导入指定列：

```
truncate table t;
copy t(f1,f2) from '/data/pgxz/t.txt';
```

查询确认导入结果：

```
select * from t;
```

| f1 | f2 | f3 | f4 |
| --- | --- | --- | --- |
| 1 | Tbase | 2017-10-30 11:54:16.559579 | |
| 2 | pg' | 2017-10-30 11:54:16.559579 | |
| 3 | pgxz | 2017-10-30 11:54:16.560283 | |

## 指定导入文件格式

导入文本文件：

```
copy t from '/data/pgxz/t.txt' (format 'text');
```

导入 CSV 文件：

```
copy t from '/data/pgxz/t.csv' (format 'csv');
```

导入二进制文件：

```
copy t from '/data/pgxz/t.bin' (format 'binary');
```

查询确认导入结果：

```
select * from t;
```

| f1 | f2 | f3 | f4 |
| --- | --- | --- | --- |
| Tbase | | | 7 |
| pg'", xc% | 2017-10-28 18:24:05.643102 | 3 | |
| pgxz | 2017-10-28 18:24:05.645691 | | |

## 使用 delimiter 指定列与列之间的分隔符

使用自定义分隔符导入文本文件：

```
copy t from '/data/pgxz/t.txt' (format 'text', delimiter '%');
```

使用自定义分隔符导入 CSV 文件：

```
copy t from '/data/pgxz/t.csv' (format 'csv', delimiter '%');
```

查询确认导入结果：

```
select * from t;
```

| f1 | f2 | f3 | f4 |
| --- | --- | --- | --- |
| 1 | Tbase | | 7 |
| 2 | pg'", xc% | 2017-10-28 18:24:05.643102 | 3 |
| 3 | pgxz | 2017-10-28 18:24:05.645691 | |

## NULL 值处理

导入文件中包含 NULL 字符串：

```
copy t from '/data/pgxz/t.txt' (null 'NULL');
```

查询确认 NULL 值处理结果：

```
select * from t;
```

| f1 | f2 | f3 | f4 |
| --- | --- | --- | --- |
| 1 | Tbase | | 7 |
| 2 | pg'", xc% | 2017-10-28 18:24:05.643102 | 3 |
| 3 | pgxz | 2017-10-28 18:24:05.645691 | |

> ⚠ **注意：**
> 字符串 NULL 在不同情境下可能不被识别，导致导入错误。

```
copy t from '/data/pgxz/t.txt' (null 'null');
```

```
ERROR: invalid input syntax for type timestamp: "NULL"
CONTEXT: COPY t, line 1, column f3: "NULL"
```

## 自定义 quote 字符

**查看 CSV 文件内容：**

```shell
cat /data/pgxz/t.csv
```

```
1,Tbase,,7
2,%pg'", xc%%%,2017-10-28 18:24:05.643102,3
3,pgxz,2017-10-28 18:24:05.645691,
```

**导入 CSV 文件时配置 quote 字符：**

```
copy t from '/data/pgxz/t.csv' (format 'csv', quote '%');
```

## 自定义 escape 字符

**查看 CSV 文件内容：**

```
cat /data/pgxz/t.csv
```

```
1,Tbase,,7
2,"pg'@", xc%",2017-10-28 18:24:05.643102,3
3,pgxz,2017-10-28 18:24:05.645691,
```

**导入 CSV 文件时配置 escape 字符：**

```
copy t from '/data/pgxz/t.csv' (format 'csv', escape '@');
```

**查询确认导入结果：**

```
select * from t;
```

| f1 | f2 | f3 | f4 |
| --- | --- | --- | --- |
| 1 | Tbase | | 7 |
| 2 | pg'", xc% | 2017-10-28 18:24:05.643102 | 3 |
| 3 | pgxz | 2017-10-28 18:24:05.645691 | |

## CSV Header 忽略首行

查看 CSV 文件内容（含头部）：

```
cat /data/pgxz/t.csv
```

```
f1,f2,f3,f4
1,Tbase,,7
2,"pg'""", xc%",2017-10-28 18:24:05.643102,3
3,pgxz,2017-10-28 18:24:05.645691,
```

导入 CSV 文件时忽略 header：

```
copy t from '/data/pgxz/t.csv' (format 'csv', header true);
```

查询确认导入结果：

```
select * from t;
```

| f1 | f2 | f3 | f4 |
| --- | --- | --- | --- |
| Tbase | | | 7 |
| pg'", xc% | 2017-10-28 18:24:05.643102 | 3 | |
| pgxz | 2017-10-28 18:24:05.645691 | | |

## 导入 oid 列值

查看文本文件内容（含 oid）：

```
cat /data/pgxz/t.txt
```

```
35242 1 Tbase \N 7
35243 2 pg'", xc% 2017-10-28 18:24:05.643102 3
35340 3 pgxz 2017-10-28 18:24:05.645691 \N
```

**导入文本文件并保留 oid：**

```
truncate table t;
copy t from '/data/pgxz/t.txt' (oids true);
```

**查询确认导入结果：**

```
select oid, * from t;
```

| oid | f1 | f2 | f3 | f4 |
| --- | --- | --- | --- | --- |
| 35242 | 1 | Tbase | | 7 |
| 35243 | 2 | pg'", xc% | 2017-10-28 18:24:05.643102 | 3 |
| 35340 | 3 | pgxz | 2017-10-28 18:24:05.645691 | |

## 使用 FORCE_NOT_NULL 把某列中空值变成长度为 0 的字符串

**查看 CSV 文件内容（含空值）：**

```
cat /data/pgxz/t.csv
```

```
1,Tbase,,7
2,"pg'""", xc%",2017-10-28 18:24:05.643102,3
3,pgxz,2017-10-28 18:24:05.645691,
4,,2017-10-30 16:14:14.954213,4
```

**导入 CSV 文件使用 FORCE_NOT_NULL：**

```
truncate table t;
copy t from '/data/pgxz/t.csv' (format 'csv', FORCE_NOT_NULL (f2));
```

**查询确认某列空值处理结果：**

```
select * from t where f2 = '';
```

| f1 | f2 | f3 | f4 |
| --- | --- | --- | --- |
| 4 | | 2017-10-30 16:14:14.954213 | 4 |

## Encoding 指定导入文件的编码

**使用 `enca` 命令确定文件编码：**

```shell
enca -L zh_CN /data/pgxz/t.txt
```

```
Simplified Chinese National Standard; GB2312
```

**导入文件到数据库：**

```
copy t from '/data/pgxz/t.txt';
```

**查询确认导入结果：**

```
select * from t;
```

| f1 | f2 | f3 | f4 |
| --- | --- | --- | --- |
| 1 | Tbase | | 7 |
| 2 | pg'", xc% | 2017-10-28 18:24:05.643102 | 3 |
| 3 | pgxz | 2017-10-28 18:24:05.645691 | |

**不指定导入文件的编码格式，则无法正确导入中文字符。使用 GBK 编码重新导入：**

```
truncate table t;
copy t from '/data/pgxz/t.txt' (encoding gbk);
```

**再次查询确认导入结果：**

```
select * from t;
```

| f1 | f2 | f3 | f4 |
| --- | --- | --- | --- |
| 1 | Tbase | | 7 |
| 2 | pg'", xc% | 2017-10-28 18:24:05.643102 | 3 |
| 3 | pgxz | 2017-10-28 18:24:05.645691 | |
| 4 | 腾讯 | 2017-10-30 16:41:09.157612 | 4 |

**可以使用 `enconv` 命令转换文件编码后再导入数据：**

```
truncate table t;
enconv -L zh_CN -x UTF-8 /data/pgxz/t.txt
copy t from '/data/pgxz/t.txt';
```

**查询确认转换编码后的导入结果：**

```
select * from t;
```

## Position 指定字段长度导入

**查看待导入的文本文件内容：**

```
cat /data/tbase/t_position_copy.txt
```

```
123 阿弟 2021
```

**使用 position 指定字段长度进行导入：**

```
copy t_position_copy (f1 position(1:3), f2 position(4:9), f3
position(10:13)) from '/data/tbase/t_position_copy.txt';
```

## position 指定字段长度和函数处理导入

```
```shell
cat /data/tbase/t_position_copy.txt
```
```

```
123 阿弟 202001012020010112121212
```

```
copy t_position_copy (f1 position(1:3), f2 position(4:9), f3
position(10:17) to_date($3,'yyyymmdd'), f4 position(18:31)
to_timestamp($4,'yyyymmddhh24miss')) from
'/data/tbase/t_position_copy.txt';
```

## position 指定使用的编码

**腾讯云数据仓库 TCHouse-P 5.06.2.1 以上版本支持，这个功能是用于要导入的数据文件的字符编码与数据库的编码不一致时能正常导入数据。**

```
file /data/tbase/t_position_copy.txt
```

```
/data/tbase/t_position_copy.txt: ISO-8859 text
```

**不指定编码导入数据时会出错：**

```
copy t_position_copy (f1 position(1:3), f2 position(4:7), f3
position(8:11)) from '/data/tbase/t_position_copy.txt';
```

```
ERROR: multibyte character "Ü2" is truncated for column f2
CONTEXT: COPY t_position_copy, line 1: "123°¢µÜ2021",
nodetype:1(1:cn,0:dn)
```

**指定要导入的文件编码为 gbk：**

```
copy t_position_copy (f1 position(1:3), f2 position(4:7), f3
position(8:11)) from '/data/tbase/t_position_copy.txt' encoding 'gbk';
```

```
select * from t_position_copy;
```

| f1 | f2 | f3 |
| --- | --- | --- |
| 123 | 阿弟 | 2021 |

注意，gbk 编码中一个汉字的长度是 2。
如果您的 gbk 文件是从 oracle 导出来的，则导入到 tbase 的 utf 数据库这样处理才能正常导入：

```
copy t4(f1 position(1:1), f2 position(2:3)) from '/data/tbase/t4.txt'
with csv encoding 'gbk';
```

如果不指定，则导入遇到如“ ”，“ ”字会出错：

```
copy t4(f1 position(1:1), f2 position(2:3)) from '/data/tbase/t4.txt'
encoding 'gbk';
```

```
ERROR: character with byte sequence 0xab 0x0a in encoding "GBK" has no
equivalent in encoding "UTF8"
CONTEXT: COPY t4, line 1: "1«1ÌÚ", nodetype:1(1:cn,0:dn)
```

## 使用多字节分隔符

腾讯云数据仓库 TCHouse-P5.06.2.1 以上版本支持。

```
create table t_position_copy(f1 int, f2 varchar(10), f3 int);
```

```
cat /data/tbase/t_position_copy_mul_delimiter.text
```

```
123@\u0026阿弟@\u00262021
```

```
copy t_position_copy from
'/data/tbase/t_position_copy_mul_delimiter.text' with delimiter
'@\u0026';
```

```
select * from t_position_copy;
```

| f1 | f2 | f3 |
| --- | --- | --- |
| 123 | 阿弟 | 2021 |

# json/jsonb 的使用

最近更新时间：2024-07-24 17:02:21

TBase 不只是一个分布式关系型数据库系统，同时它还支持非关系数据类型 json。JSON 数据类型是用来存储 JSON（JavaScript Object Notation）数据的。这种数据也可以被存储为 text，但是 JSON 数据类型的 优势在于能强制要求每个被存储的值符合 JSON 规则。也有很多 JSON 相关的函数和操作符可以用于存储在这些数据类型中的数据。JSON 数据类型有 json 和 jsonb。它们接受完全相同的值集合作为输入。主要的实际区别是效率。

## json 应用

### 创建 json 类型字段表

```sql
```sql
create table t_json(id int,f_json json);
```

### 插入数据

```sql
insert into t_json values(1,'{"col1":1,"col2":"tbase"}');
insert into t_json values(2,'{"col1":1,"col2":"tbase","col3":"pgxz"}');
```

### 通过键获得 JSON 对象域

```sql
select f_json ->'col2' as col2 ,f_json -> 'col3' as col3 from t_json;
```

| col2 | col3 |
| --- | --- |
| "tbase" | |
| "tbase" | "pgxz" |

### 以文本形式获取对象值

```sql
select f_json ->>'col2' as col2 ,f_json ->> 'col3' as col3 from t_json;
```

| col2 | col3 |
| --- | --- |
```

| | |
|---|---|
| tbase | |
| tbase | pgxz |

# jsonb 应用

## 创建 jsonb 类型字段表

```
create table t_jsonb(id int,f_jsonb jsonb);
```

## 插入数据

```
insert into t_jsonb values(1,'{"col1":1,"col2":"tbase"}');
insert into t_jsonb values(2,'{"col1":1,"col2":"tbase","col3":"pgxz"}');
```

- jsonb 插入时会移除重复的键。

## 更新数据

- 增加元素

```
update t_jsonb set f_jsonb = f_jsonb || '{"col3":"pgxz"}'::jsonb where
id=1;
```

- 更新原来的元素

```
update t_jsonb set f_jsonb = f_jsonb || '{"col2":"tbase"}'::jsonb where
id=3;
```

- 删除某个键

```
update t_jsonb set f_jsonb = f_jsonb - 'col3';
```

## jsonb_set()函数更新数据

```
update t_jsonb set f_jsonb = jsonb_set( f_jsonb , '{col}', '"pgxz"' ,
true ) where id=1;
```

```
update t_jsonb set f_jsonb = jsonb_set( f_jsonb , '{col}', '"pgxz"' ,
false ) where id=2;
update t_jsonb set f_jsonb = jsonb_set( f_jsonb , '{col2}', '"pgxz"' ,
false ) where id=3;
```

# jsonb 函数应用

## jsonb_each() 将 json 对象转变键和值

```
select * from jsonb_each((select f_jsonb from t_jsonb where id=1));
```

| key | value |
| --- | --- |
| col | "pgxz" |
| col1 | 1 |
| col2 | "tbase" |

## jsonb_each_text() 将 json 对象转变文本类型的键和值

```
select * from jsonb_each_text((select f_jsonb from t_jsonb where id=1));
```

| key | value |
| --- | --- |
| col | pgxz |
| col1 | 1 |
| col2 | tbase |

## row_to_json() 将一行记录变成一个 json 对象

```
select row_to_json(tbase) from tbase;
```

```
{"id":1,"nickname":"tbase"}
{"id":2,"nickname":"pgxz"}
```

## json_object_keys()返回一个对象中所有的键

```
select * from json_object_keys((select f_jsonb from t_jsonb where
id=1)::json);
```

| json_object_keys |
| --- |
| col |
| col1 |
| col2 |

# jsonb 索引使用

TBase 为文档 jsonb 提供了 GIN 索引，GIN 索引可以被用来有效地搜索在大量 jsonb 文档（数据）中出现的键或者键值对。

## 创建立 jsonb 索引

```
create index t_jsonb_f_jsonb_idx on t_jsonb using gin(f_jsonb);
```

## 测试查询的性能

- 没有索引开销

```
select * from t_jsonb where f_jsonb @> '{"col1":9999}';
```

- 有索引开销

```
select * from t_jsonb where f_jsonb @> '{"col1":9999}';
```

# with 子查询及递归的使用

最近更新时间：2024-07-24 17:02:21

## with 子查询及递归的使用

```sql
create table t_area(id int,pid int,area varchar);
insert into t_area values(1,null,'广东省');
insert into t_area values(2,1,'深圳市');
insert into t_area values(3,1,'东莞市');
insert into t_area values(4,2,'南山区');
insert into t_area values(5,2,'福田区');
insert into t_area values(6,2,'罗湖区');
insert into t_area values(7,3,'南城区');
insert into t_area values(8,3,'东城区');
```

## with 子查询

使用 `with` 子查询来选择 `pid=1` 的所有行：

```sql
with t_gd_city AS (
  select * from t_area where pid=1
)
select * FROM t_gd_city;
```

查询结果：

| id | pid | area |
| --- | --- | --- |
| 2 | 1 | 深圳市 |
| 3 | 1 | 东莞市 |

# 外表
# COS 外表

最近更新时间：2025-06-09 15:48:13

腾讯云数据仓库 TCHouse-P 支持通过 hdfs_fdw 插件以外表形式读写 COS 。

## 创建 SERVER

根据 COS 类型，选择创建以下 SERVER。`$BucketName` 表示已经创建的存储桶的名称。

### 创建 COS SERVER

```
CREATE SERVER $ServerName
FOREIGN DATA WRAPPER hdfs_fdw
OPTIONS (address 'cos://$BucketName', appid '$appid', access_keyid
'$ak', secret_accesskey '$sk', region '$region', client_type 'cos');
```

### 创建 COSN SERVER

```
CREATE SERVER $ServerName
FOREIGN DATA WRAPPER hdfs_fdw
OPTIONS (address 'cosn://$BucketName', appid '$appid', access_keyid
'$ak', secret_accesskey '$sk', region '$region', client_type 'cosn',
ranger '$ip:$port', is_s3 'true/false');
```

⚠ **注意：**

1. COSN 如果开启了 Ranger 鉴权，则将参数 `ranger` 设置为 Ranger 的 IP:PORT；
2. COSN 默认支持 S3 协议；若需 POSIX 协议，则将参数 `is_s3` 设置为 false。

## 创建 USER MAPPING

`$ServerName` 表示已经创建 Server 的名称。

```
CREATE USER MAPPING FOR $用户名 SERVER $ServerName;
```

## 创建只读外表

腾讯云数据仓库 TCHouse-P 支持通过外表向 COS 端读取 CSV、TEXT、Parquet、ORC 这 4 种格式文件。
外部表所指定文件格式需要和 COS 端所读取的文件格式保持一致。而且，所读取的数据文件必须位于该 COS 的 `$FOLDERNAME` 目录中。

## 创建 CSV 格式的的只读外表

```
CREATE FOREIGN TABLE test_csv(id int, name TEXT)
SERVER $ServerName
OPTIONS (FORMAT 'csv', DELIMITER '$列分隔符', FOLDERNAME '/$数据目录/',
distribute 'shard');
```

**说明：**

1. 如果读取单个文件，则参数 `FOLDERNAME` 结尾不用加“/”，例如：FOLDERNAME '/test.txt'。
2. 如果 `DELIMITER` 以特殊字符作为分隔符，则需添加“E”，例如：以TAB键为分隔符，则 DELIMITER E'\t'。

## 创建 TEXT 格式的只读外表

```
CREATE FOREIGN TABLE test_text(id int, name TEXT)
SERVER $ServerName
OPTIONS (FORMAT 'text', DELIMITER '$列分隔符', FOLDERNAME '/$数据目录/',
distribute 'shard');
```

## 创建 Parquet 格式的只读外表

```
CREATE FOREIGN TABLE test_parquet (col1 text, col2 bigint, col3 double
precision)
SERVER $ServerName
OPTIONS (FORMAT 'parquet', FOLDERNAME '/$数据目录/', distribute 'shard');
```

## 创建 ORC 格式的只读外表

```
CREATE FOREIGN TABLE test_orc(a bigint, b text, c float)
SERVER $ServerName
OPTIONS (FORMAT 'orc', FOLDERNAME '/$数据目录/', distribute 'shard');
```

## 创建分区表的只读外表

以 Hive 建表语句为例。

```
CREATE TABLE login_logs_parquet(l_id string, l_loginName string, l_date
string)
partitioned by(year string, month string) stored as parquet;
```

**说明：**

1. Hive 分区字段提升为腾讯云数据仓库 TCHouse-P 的普通字段。
2. 参数 `PARTITION` 中填写 Hive 上的分区字段，多级分区需用逗号隔开，例如：PARTITION 'year,month'。

```
CREATE FOREIGN TABLE test_table(l_id text,l_loginName text,l_date
text,year text,month text)
SERVER $ServerName
OPTIONS (FORMAT '$文件格式', FOLDERNAME '/$数据目录/', distribute 'shard',
PARTITION 'year,month');
```

## 将外表数据导入内表

```
INSERT INTO $内部表 SELECT * FROM $读外表;
```

# 创建只写外表

腾讯云数据仓库 TCHouse-P 支持通过外表往 COS 端写入 CSV、TEXT、ORC 这 3 种格式文件。

## 创建 CSV 格式的只写外表

```
CREATE FOREIGN TABLE w_test_csv(id int,name text)
SERVER $ServerName
OPTIONS (FORMAT 'csv', DELIMITER '|', FOLDERNAME '/$数据目录/', distribute
'shard') WRITE ONLY;
```

## 创建 TEXT 格式的只写外表

```
CREATE FOREIGN TABLE w_test_txt(id int, name text)
SERVER $ServerName
OPTIONS (FORMAT 'text', DELIMITER '|', FOLDERNAME '/$数据目录/',
distribute 'shard') WRITE ONLY;
```

## 创建 ORC 格式的只写外表

```
CREATE FOREIGN TABLE w_test_orc(id int,name text)
SERVER $ServerName
OPTIONS (FORMAT 'orc', FOLDERNAME '/$数据目录/', distribute 'shard') WRITE
ONLY;
```

## 将内表数据导出外表

```
INSERT INTO $写外表 SELECT * FROM $内部表;
```