云数据仓库 PostgreSQL

最佳实践

产品文档

腾讯云

版权所有：腾讯云计算（北京）有限责任公司

【版权声明】

©2013-2024 腾讯云版权所有

本文档著作权归腾讯云单独所有，未经腾讯云事先书面许可，任何主体不得以任何形式复制、修改、抄袭、传播全部或部分本文档内容。

【商标声明】

腾讯云

及其它腾讯云服务相关的商标均为腾讯云计算（北京）有限责任公司及其关联公司所有。本文档涉及的第三方主体的商标，依法由权利人所有。

【服务声明】

本文档意在向客户介绍腾讯云全部或部分产品、服务的当时的整体概况，部分产品、服务的内容可能有所调整。您所购买的腾讯云产品、服务的种类、服务标准等应由您与腾讯云之间的商业合同约定，除非双方另有约定，否则，腾讯云对本文档内容不做任何明示或默示的承诺或保证。

文档目录

最佳实践
数仓表开发

最近更新时间：2024-02-19 15:58:15

云数据仓库 Postgresql 数据库中的表与其它关系型数据库中的表类似，不同的是表中的行被分布在不同 Segment 上，表的分布策略决定了在不同 Segment 上面的分布情况。

创建普通表

CREATE TABLE 命令用于创建一个表，创建表时可以定义以下内容：

表的列以及 数据类型

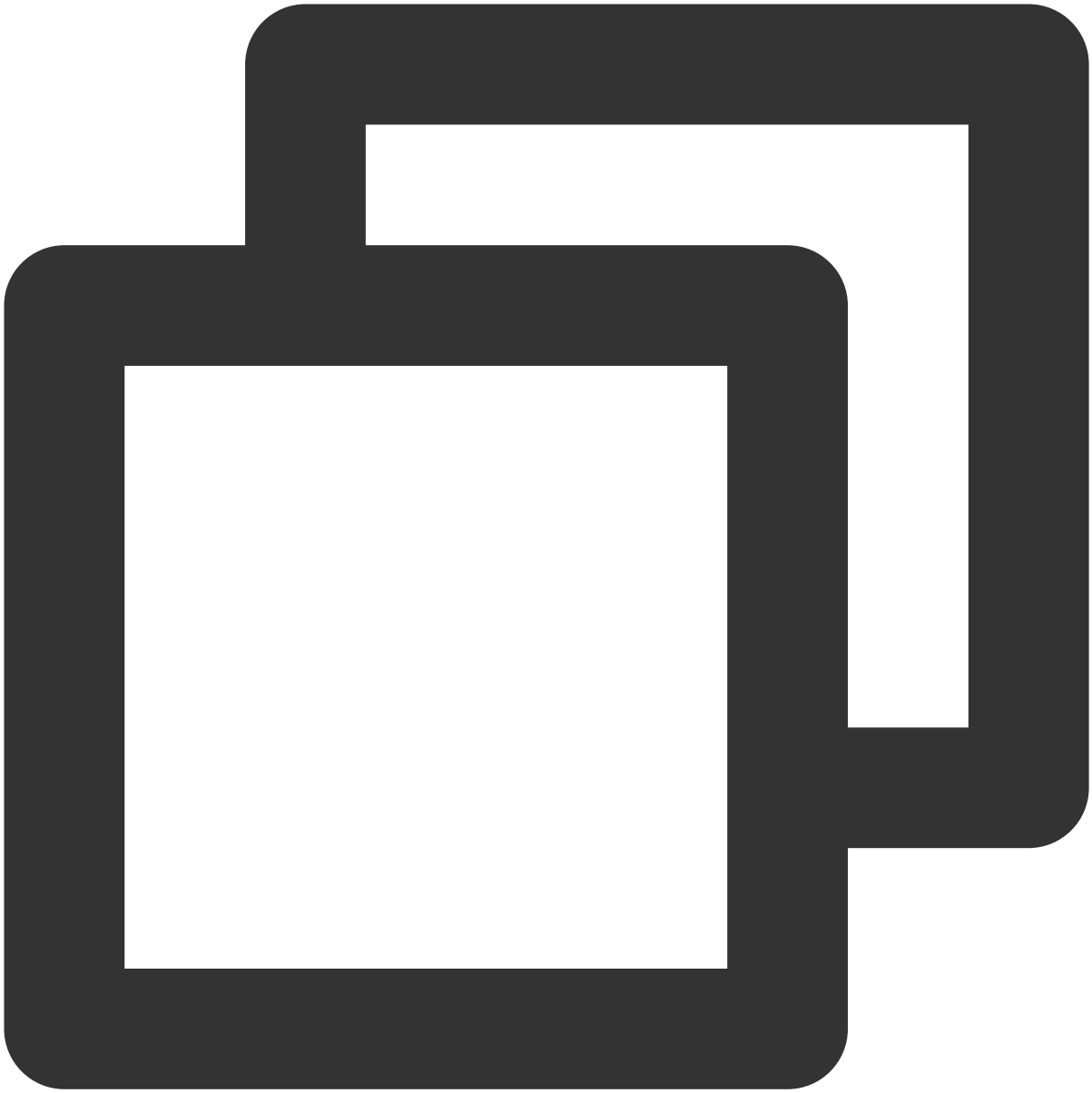
表约束的定义

表分布定义

表存储格式

分区表定义

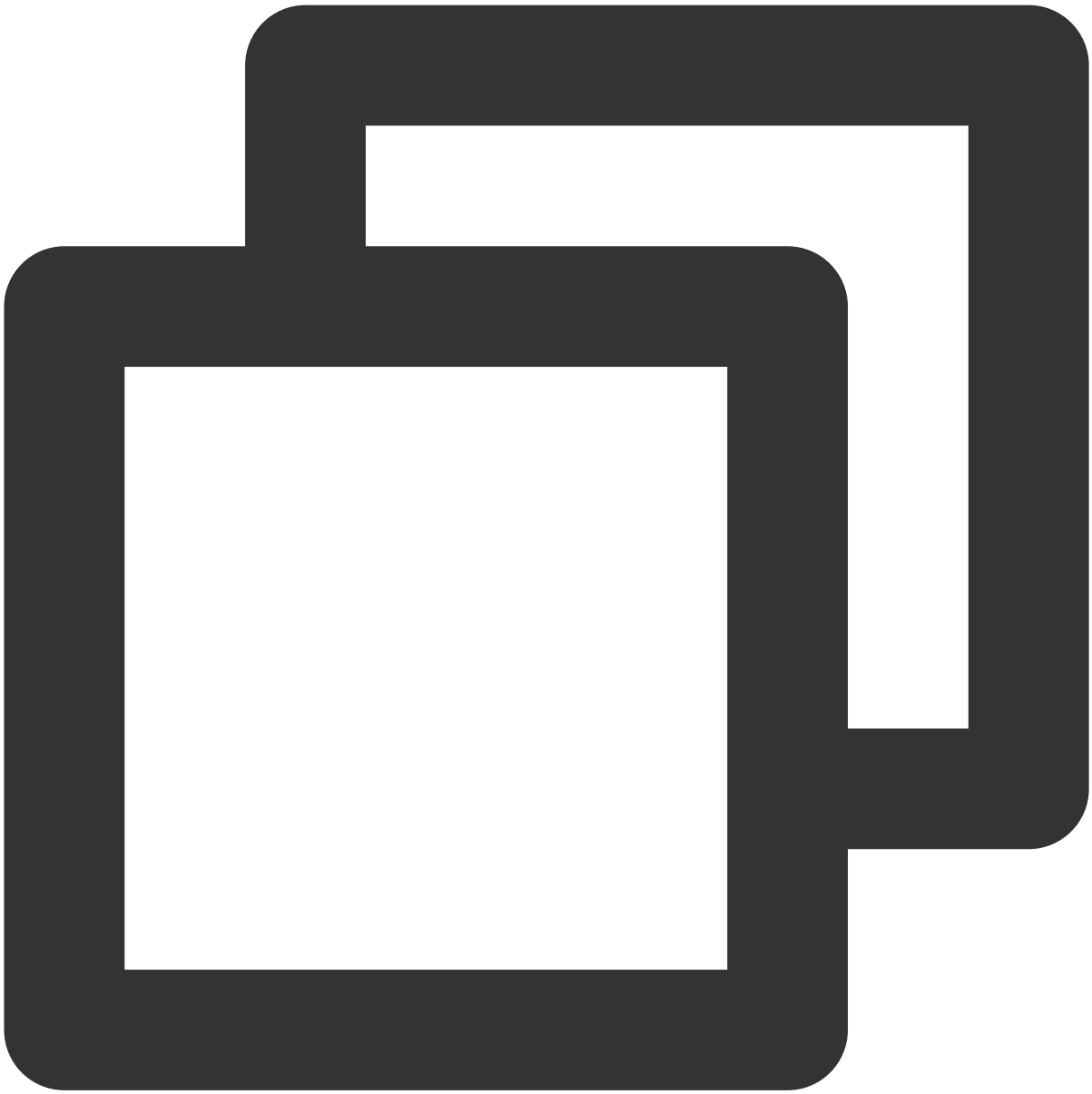
使用 `CREATE TABLE` 命令创建表，格式如下：



```
CREATE TABLE table_name (
[ { column_name data_type [ DEFAULT default_expr ]    -- 表的列定义
   [column_constraint [ ... ]                         -- 列的约束定义
]
   | table_constraint                                 -- 表级别的约束定义
   ])
   [ WITH ( storage_parameter=value [, ... ] )        -- 表存储格式定义
   [ DISTRIBUTED BY (column, [ ... ] ) | DISTRIBUTED RANDOMLY ]  -- 表的分布键定义
   [ partition clause]                                -- 表的分区定义
```

示例：

以下示例中的建表语句创建了一个表，使用 **trans_id** 作为分布键，并基于 **date** 设置了 RANGE 分区功能。

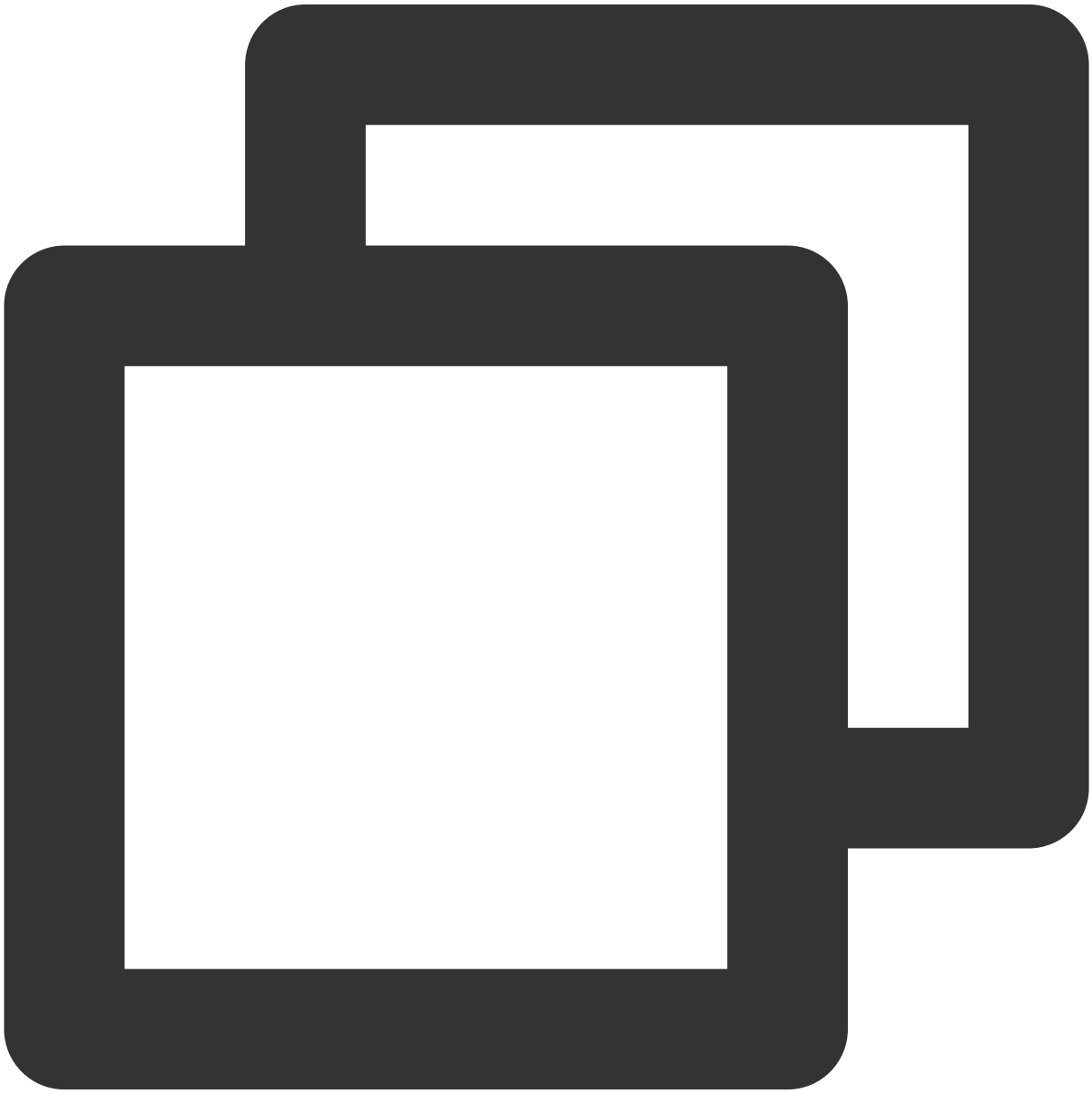


```
CREATE TABLE sales (
  trans_id int,
  date date,
  amount decimal(9,2),
  region text)
  DISTRIBUTED BY (trans_id)
  PARTITION BY RANGE(date)
  (start (date '2018-01-01') inclusive
```

```
    end (date '2019-01-01') exclusive every (interval '1 month'),
    default partition outlying_dates);
```

创建临时表

临时表（Temporary Table）会在会话结束时自动删除，或选择性地在当前事务结束的时候删除，用于存储临时中间结果。创建临时表的命令如下：



```
CREATE TEMPORARY TABLE table_name(…)
```

```
    [ON COMMIT {PRESERVE ROWS | DELETE ROWS | DROP}]
```

说明:临时表的行为在事务块结束时的行为可以通过上述语句中的ON COMMIT来控制。

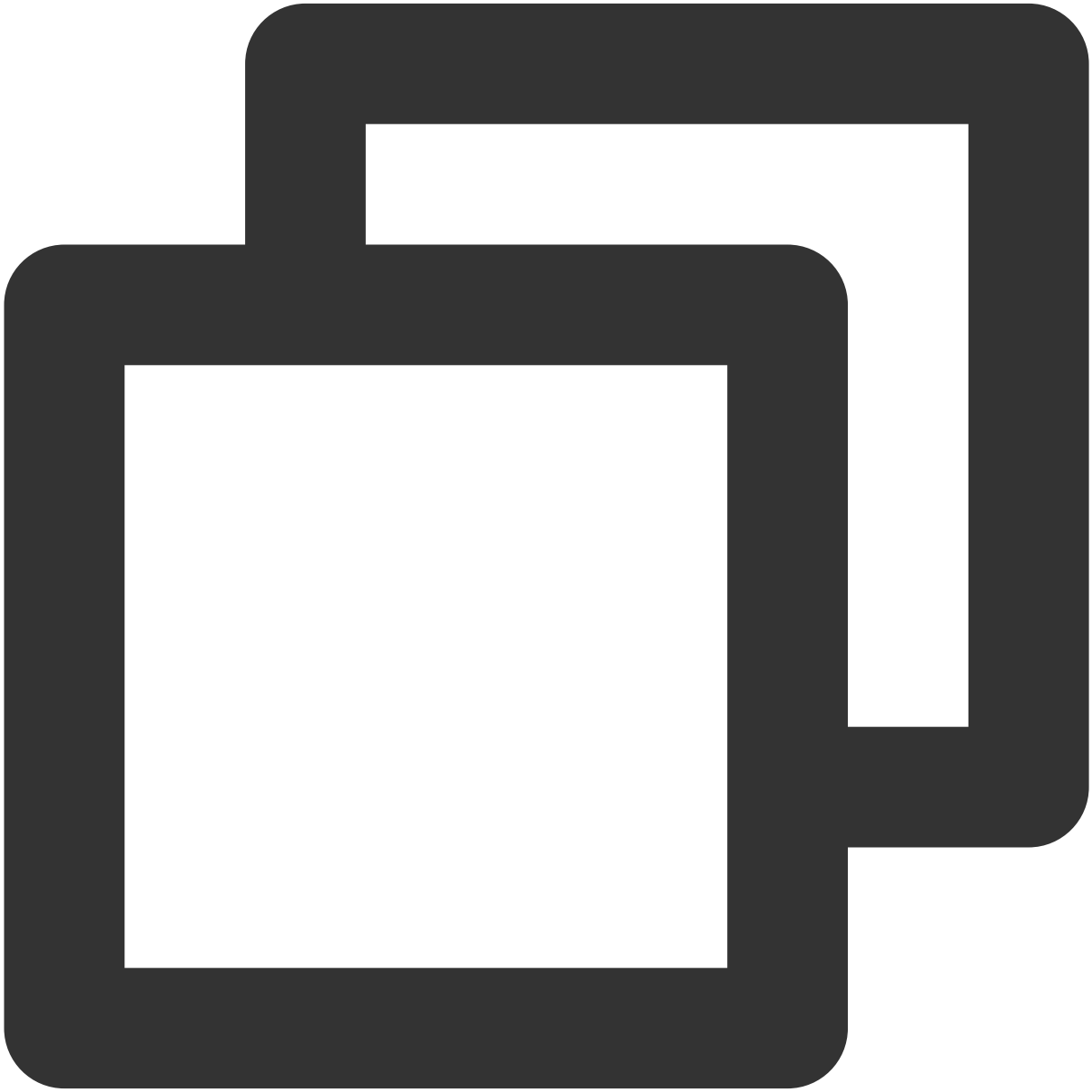
PRESERVE ROWS：在事务结束时候保留数据，这是默认的行为。

DELETE ROWS：在每个事务块结束时，临时表的所有行都将被删除。

DROP：在当前事务结束时，会删除临时表。

示例：

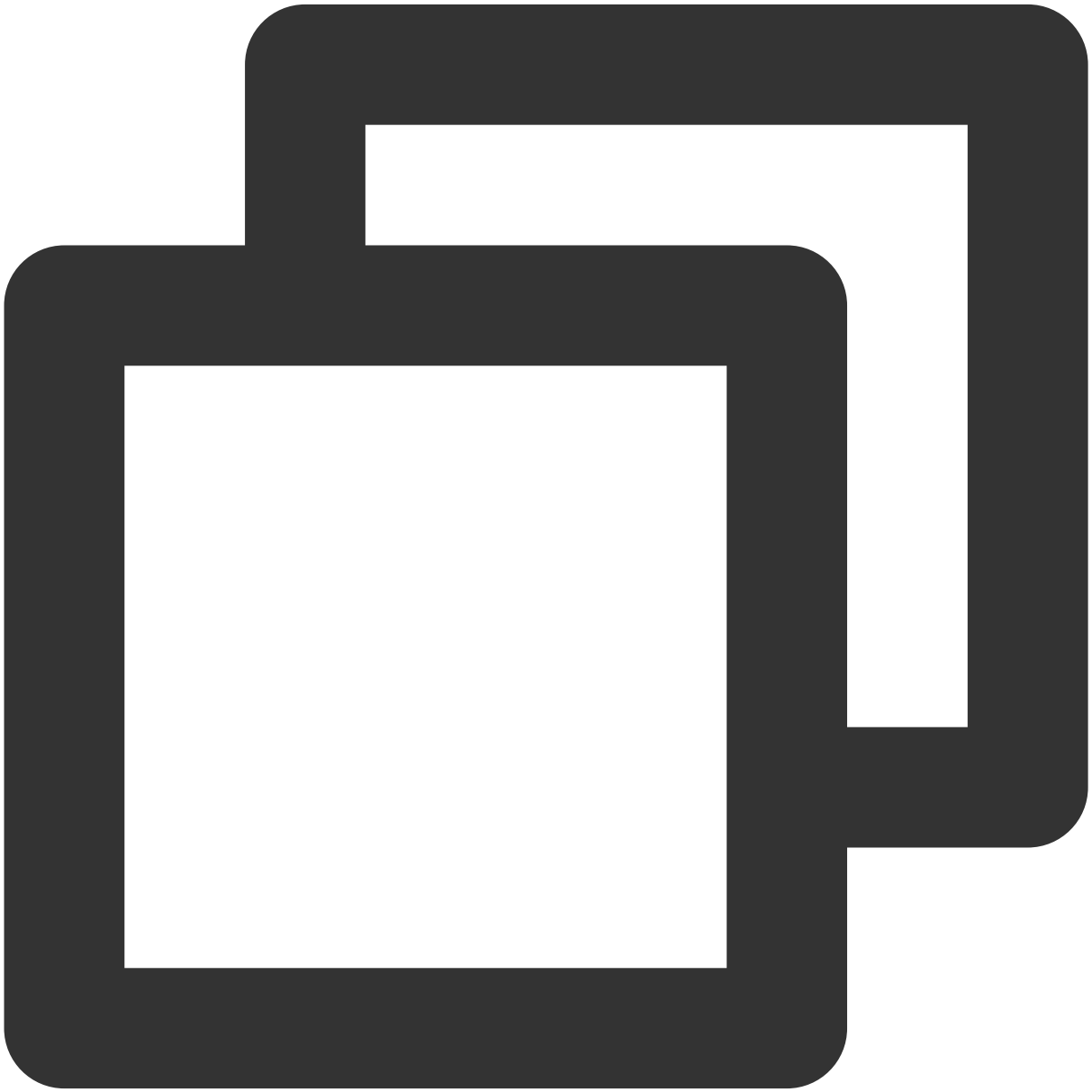
创建一个临时表，事务结束时候删除该临时表。



```
CREATE TEMPORARY TABLE temp_foo (a int, b text) ON COMMIT DROP;
```

表约束的定义

您可以在列和表上定义约束来限制表中的数据，但是有以下一些限制：
CHECK 约束引用的列只能在其所在的表中。
UNIQUE 和 PRIMARY KEY 约束必须包含分布键列，UNIQUE 和 PRIMARY KEY 约束不支持追加优化表和列存表。
允许 FOREIGN KEY 约束在云数仓 Postgresql 上无效。
实际使用约束命令如下:

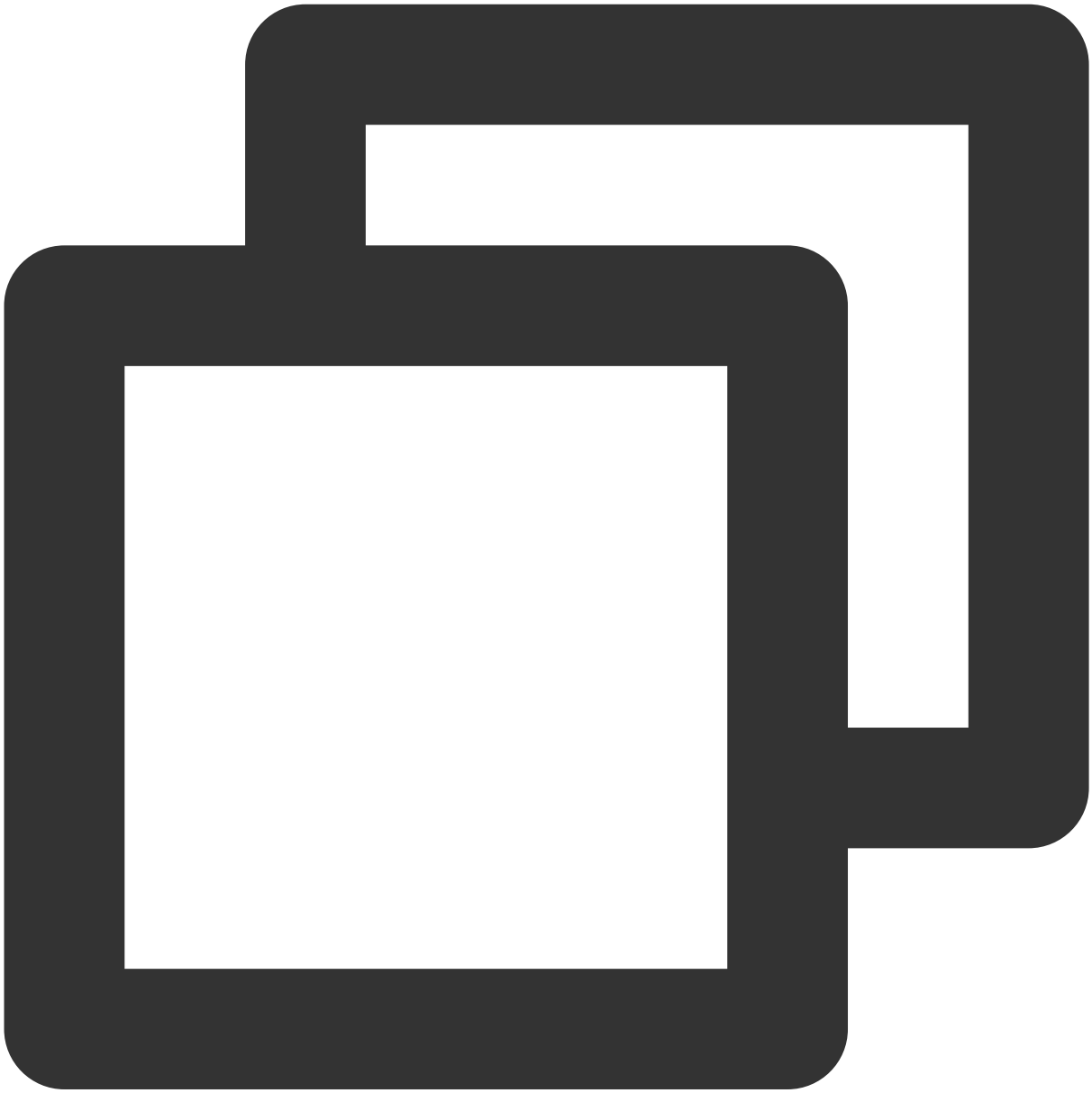


```
UNIQUE ( column_name [, ... ] )
    | PRIMARY KEY ( column_name [, ... ] )
```

```
    | CHECK ( expression )
```

检查约束

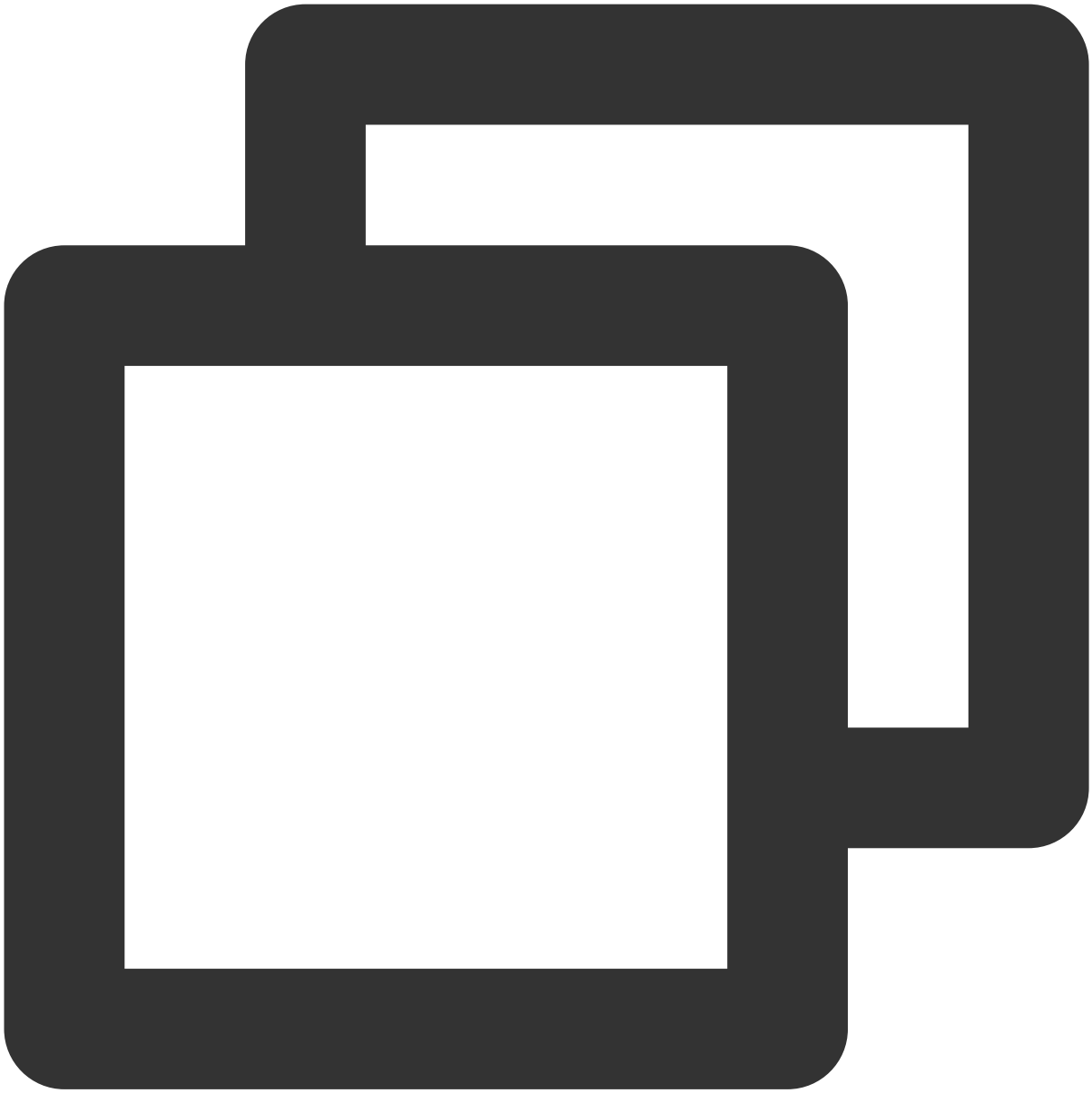
检查约束（Check Constraints）指定列中的值必须满足一个布尔表达式，例如：



```
CREATE TABLE products
            ( product_no integer,
              name text,
              price numeric CHECK (price > 0) );
```

非空约束（Not-Null Constraints）

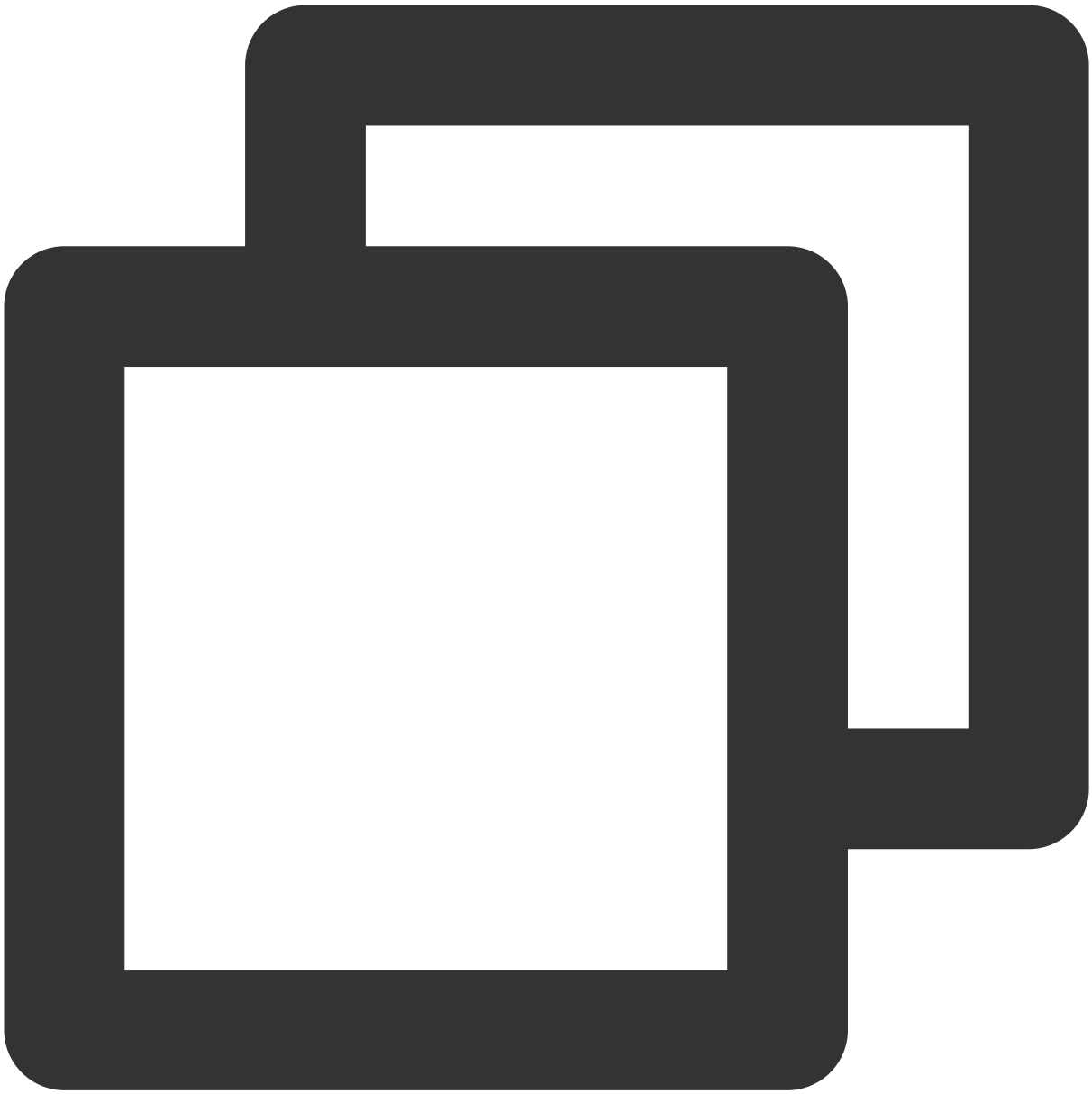
非空约束（Not-Null Constraints）指定列不能有空值，例如：



```
CREATE TABLE products
       ( product_no integer NOT NULL,
         name text NOT NULL,
         price numeric );
```

唯一约束（Unique Constraints）

唯一约束（Unique Constraints）确保一列或者一组列中包含的数据对于表中所有的行都是唯一的。包含唯一约束的表必须是哈希分布，并且约束列需要包含分布键列，例如：



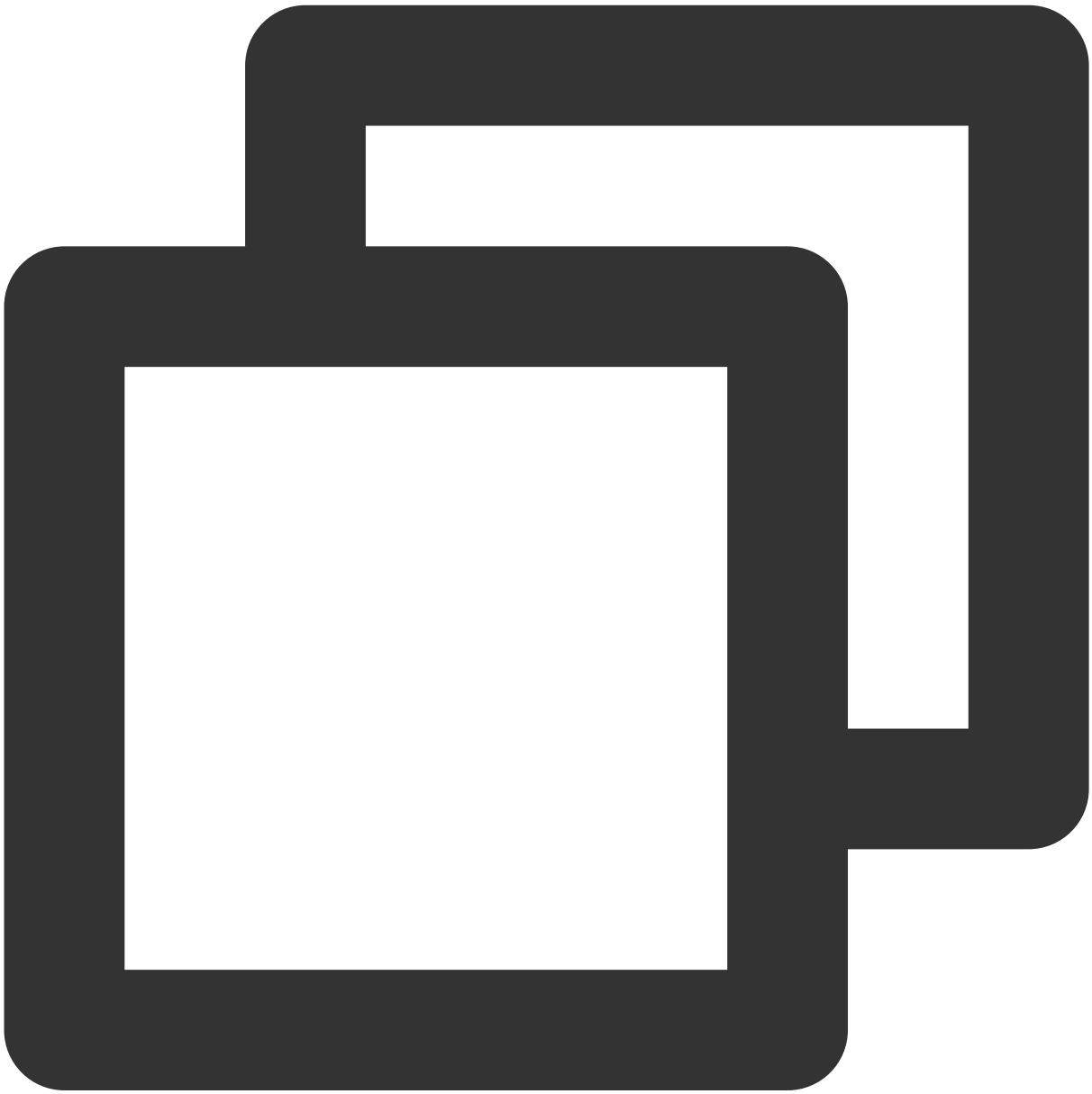
```
CREATE TABLE products
       ( product_no integer UNIQUE,
         name text,
         price numeric)
      DISTRIBUTED BY (product_no);
```

注意：

仅行存 HEAP 表支持主键约束，APPEDN ONLY 表均不支持主键约束。

主键约束（Primary Keys Constraints）

主键约束（Primary Keys Constraints）是一个 UNIQUE 约束和一个 NOT NULL 约束的组合。包含主键约束的表必须是哈希分布，并且约束列需要包含分布键列。如果一个表具有主键，这个列（或者这一组列）会被默认选中为该表的分布键，例如：



```
CREATE TABLE products
       ( product_no integer PRIMARY KEY,
         name text,
         price numeric)
      DISTRIBUTED BY (product_no);
```

注意：

仅行存 HEAP 表支持主键约束，APPEDN ONLY 表均不支持主键约束。

表分布键选择

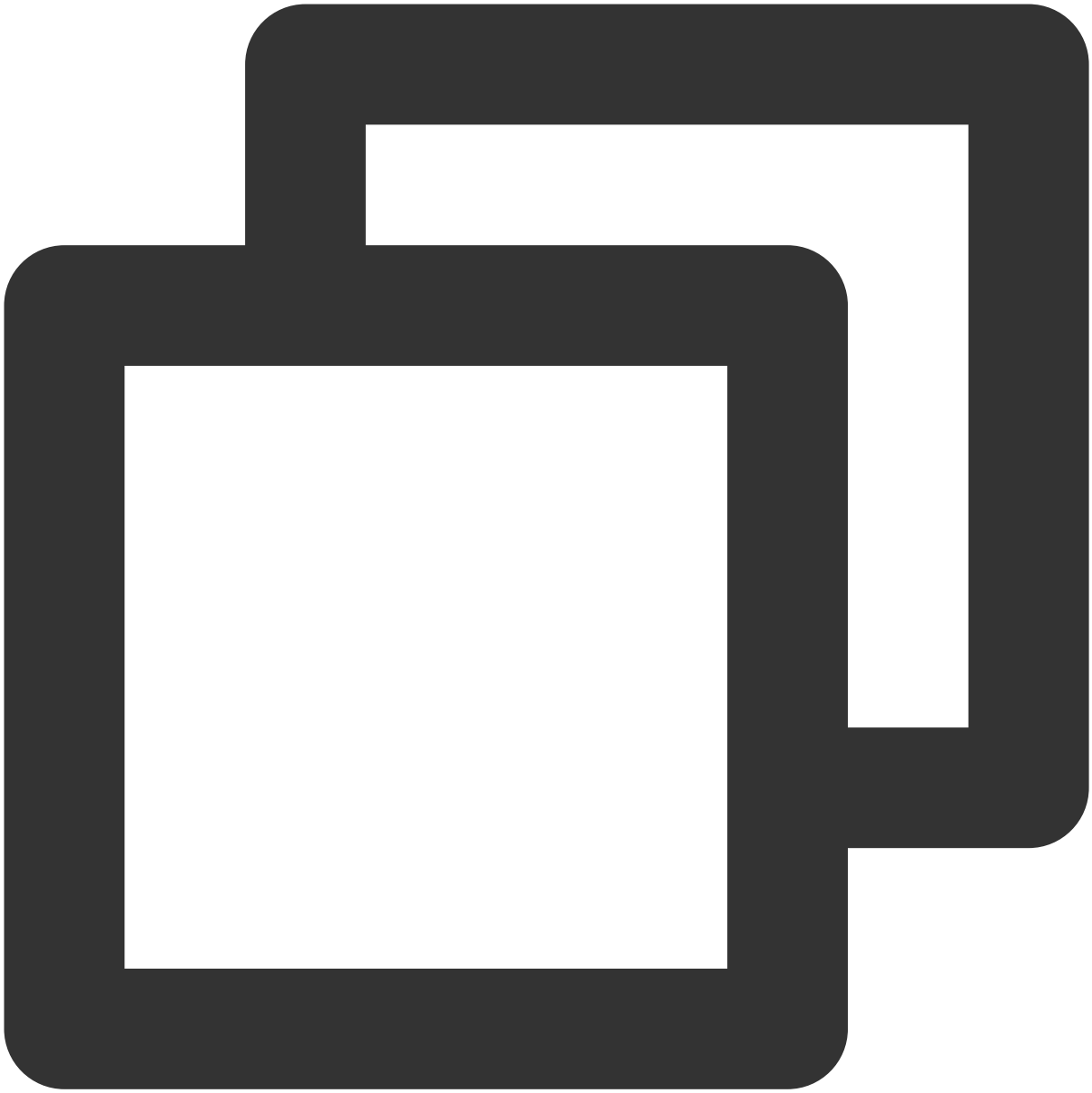
最近更新时间：2024-02-19 15:58:16

本文介绍云数据仓库 PostgreSQL 如何选择表的分布策略。

选择表分布策略

云数据仓库 PostgreSQL 支持三种数据在节点间的分布方式，按指定列的哈希（HASH）分布、随机（RANDOMLY）分布、复制（REPLICATED）分布。

版权所有：腾讯云计算（北京）有限责任公司



```
CREATE TABLE <table_name> (...) [ DISTRIBUTED BY (<column>  [,..] ) | DISTRIBUTED R
```

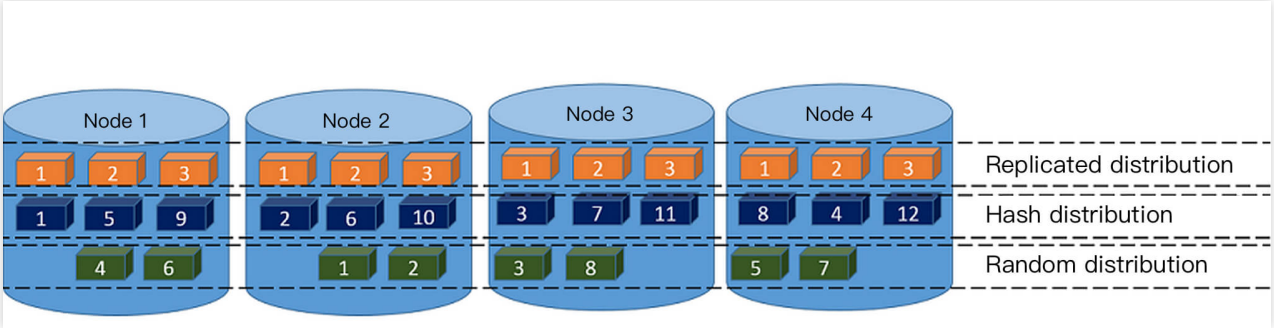
建表语句 `CREATE TABLE` 支持如下三个分布策略的子句：

`DISTRIBUTED BY (column, [ ... ])` 指定数据按分布列的哈希值在节点（Segment）间分布，根据分布列哈希值将每一行分配给特定节点（Segment）。相同的值将始终散列到同一个节点。选择唯一的分布键（例如 Primary Key）将确保较均匀的数据分布。哈希分布是表的默认分布策略， 如果创建表时未提供 DISTRIBUTED 子句，则将 PRIMARY KEY 或表的第一个合格列用作分布键。如果表中没有合格的列，则退化为随机分布策略。

`DISTRIBUTED RANDOMLY` 指定数据按循环的方式均匀分配在各节点 （Segment） 间，与哈希分布策略不同，具有相同值的数据行不一定位于同一个 segment 上。虽然随机分布确保了数据的平均分布，但只建议当表没有合适的

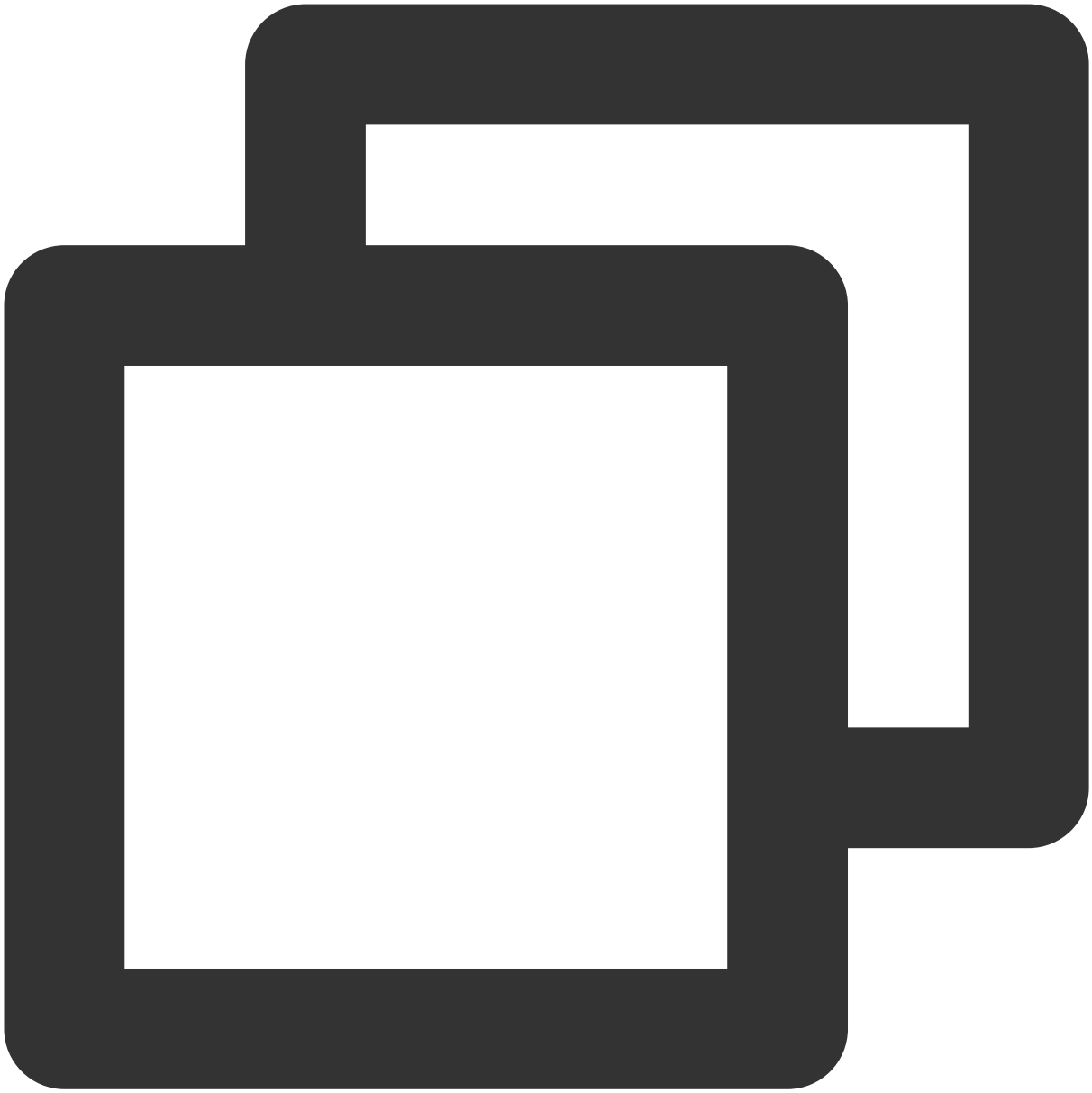
离散分布的数据列作为哈希分布列时采用随机分布策略。

`DISTRIBUTED REPLICATED` 指定数据为复制分布，即每个节点（Segment）上有该表的全量数据，这种分布策略下表数据将均匀分布，因为每个 segment 都存储着同样的数据行，当有大表与小表join，把足够小的表指定为 replicated 也可能提升性能。



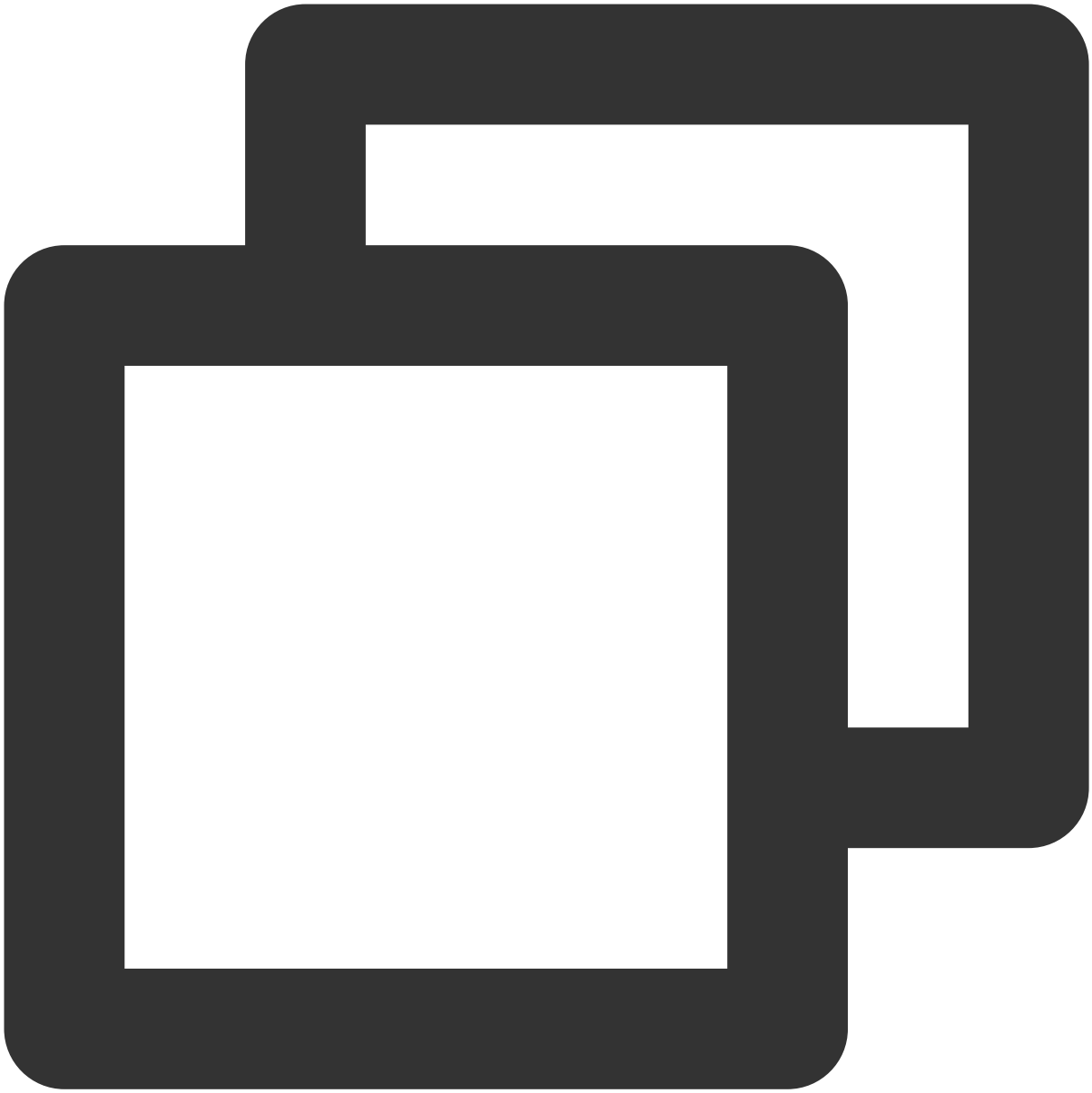
示例如下：

示例中的建表语句创建了一个哈希（Hash）分布的表，数据将按分布键的哈希值被分配到对应的节点 Segment 数据节点。



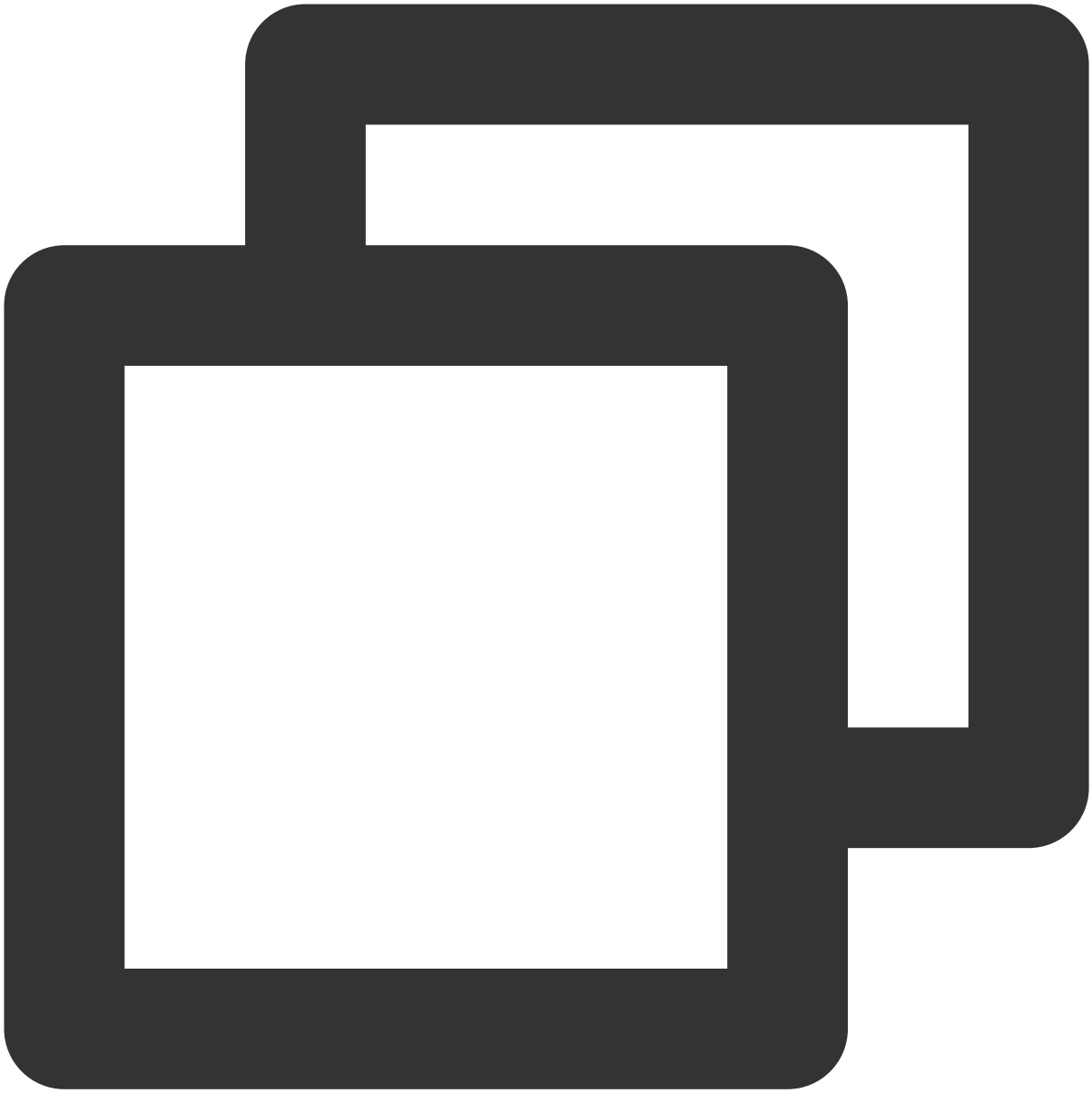
```
CREATE TABLE products (name varchar(40),
                       prod_id integer,
                       supplier_id integer)
                       DISTRIBUTED BY (prod_id);
```

示例中的建表语句创建了一个随机（Randomly）分布的表，数据被循环着放置到各个 Segment 数据节点。 当表没有合适的离散分布的数据列作为哈希分布列时，可以采用随机分布策略。



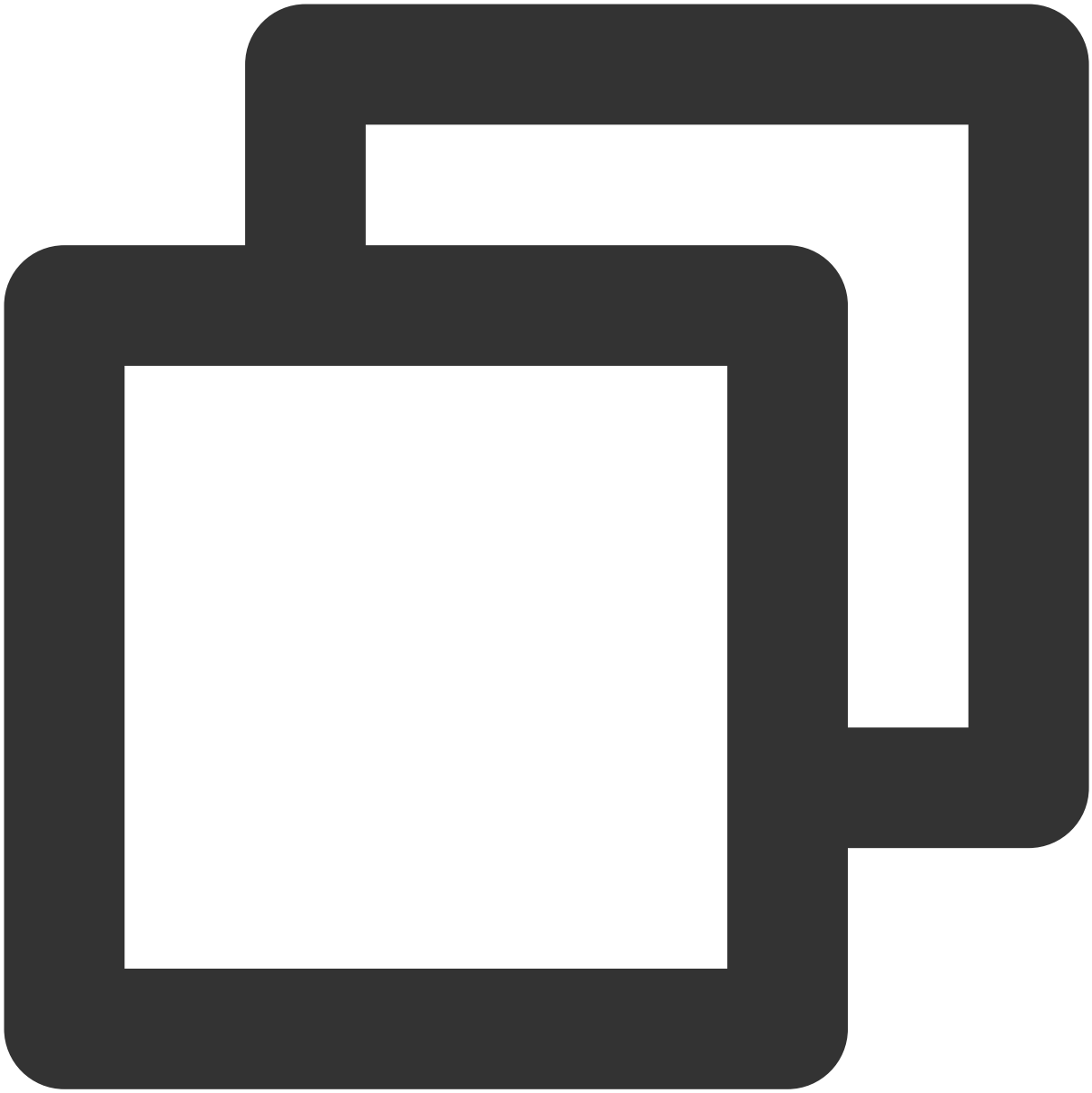
```
CREATE TABLE random_stuff (things text,
                           doodads text,
                           etc text)
                           DISTRIBUTED RANDOMLY;
```

示例中的建表语句创建了一个复制（Replicated）分布的表，每个Segment数据节点都存储有一个全量的表数据。



```
CREATE TABLE replicated_stuff (things text,
                               doodads text,
                               etc text)
                               DISTRIBUTED REPLICATED;
```

对于按分布键的简单查询，包括 UPDATE 和 DELETE 等语句，AnalyticDB PostgreSQL 具有按节点的分布键进行数据节点裁剪的功能，例如 products 表使用 prod_id 作为分布键，以下查询只会被发送到满足 prod_id=101的 segment 上执行，从而极大提升该 SQL 执行性能：



```
select * from products where prod_id = 101;
```

表分布键选择原则

合理规划分布键，对表查询的性能至关重要，有以下原则需要关注：
尽量不使用复制表，复制表容易导致查询退化，反而出现查询变慢情况。

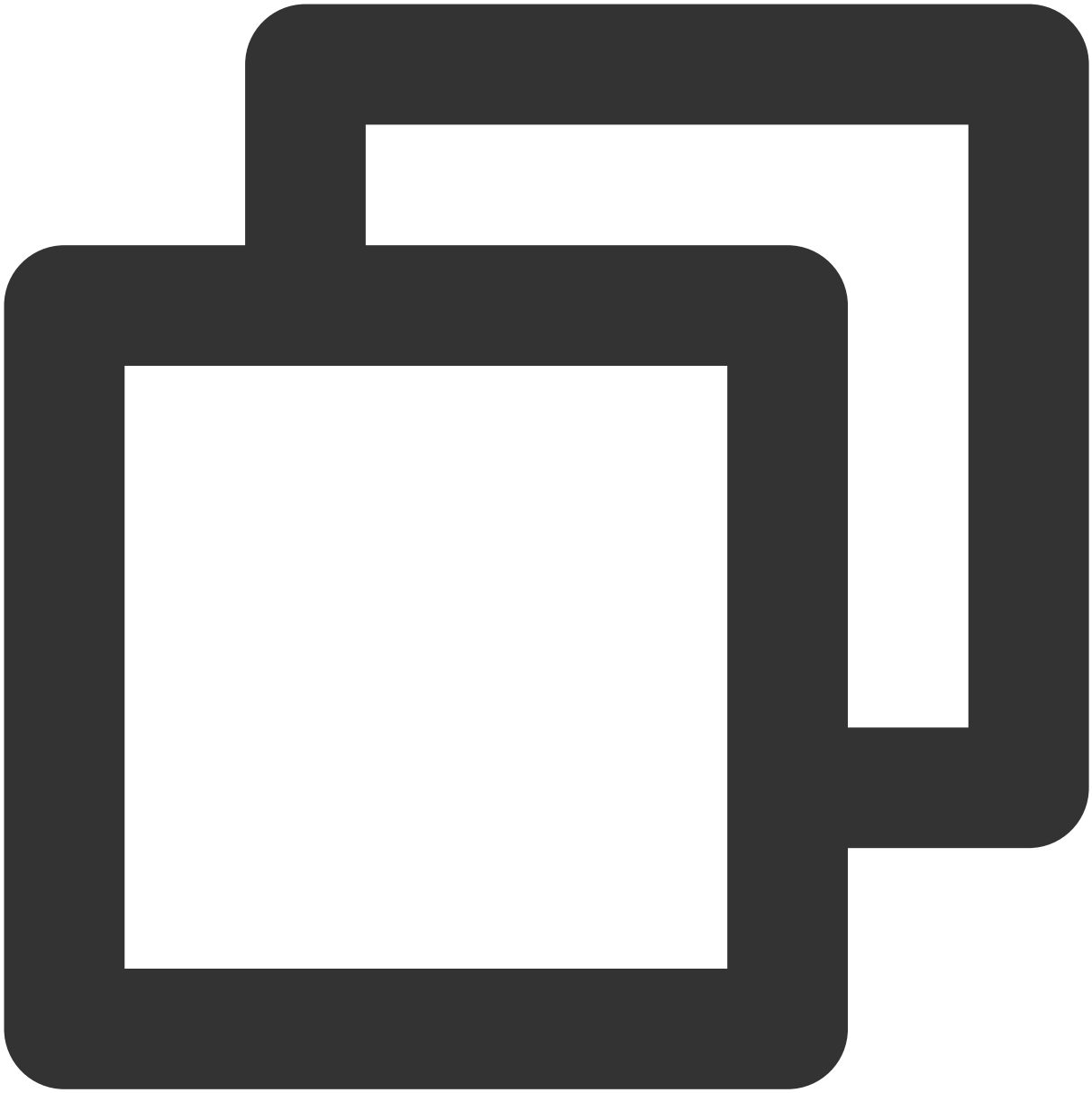
选择数据分布均匀的列或者多个列：若选择的分布列数值分布不均匀，则可能导致数据倾斜。某些 Segment 分区节点存储数据多（查询负载高）。根据木桶原理，时间消耗会卡在数据多的节点上。故不应选择 bool 类型，时间日期类型数据作为分布键。

选择经常需要 JOIN 的列作为分布键，可以实现图一所示**本地关联（Collocated JOIN）**计算，即当 JOIN 键和分布键一致时，可以在 Segment 分区节点内部完成 JOIN。否则需要将一个表进行重分布（**Redistribute motion**）来实现图二所示**重分布关联（Redistributed Join）**或者广播其中小表(**Broadcast motion**)来实现图三所示**广播关联（Broadcast Join），后两种方式都会有较大的网络开销**。

尽量选择高频率出现的查询条件列作为分布键，从而可能实现按分布键做节点 Segment 的裁剪。

若未指定分布键，默认表的主键为分布键，若表没有主键，则默认将第一列当做分布键。

分布键可以被定义为一个或多个列。例如：

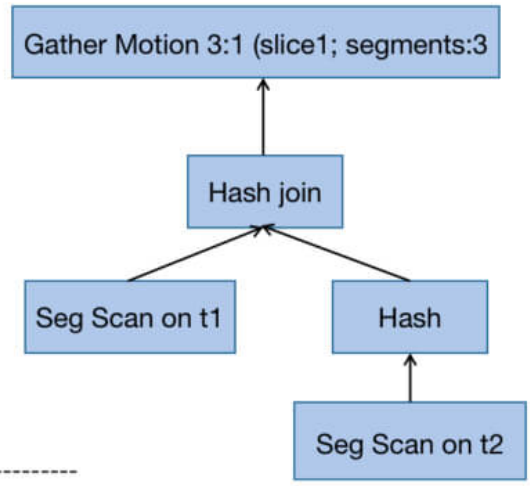


```
create table t1(c1 int, c2 int) distributed by (c1,c2);
```

Figure 1. Join

Both the t1 and t2 tables use the same distribution column.

Joins are completed within the respective segments, and there is no data network transfer (motion).

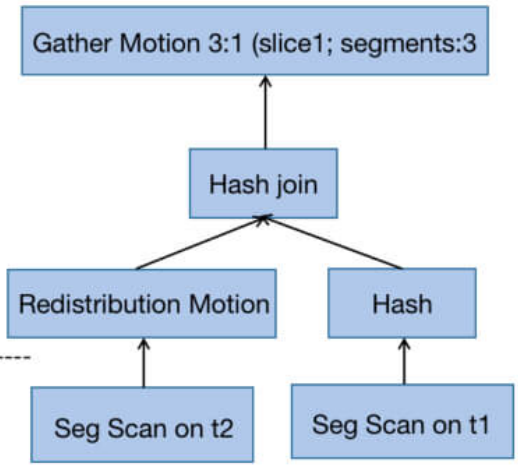



```
QUERY PLAN
------------------------------------------------------------------------------
Gather Motion 3:1  (slice1; segments: 3)  (cost=3.23..6.48 rows=10
width=16)
->  Hash Join  (cost=3.23..6.48 rows=4 width=16)
Hash Cond: t2.c1 = t1.c1
->  Seq Scan on t2  (cost=0.00..3.11 rows=4 width=8)
->  Hash  (cost=3.10..3.10 rows=4 width=8)
->  Seq Scan on t1  (cost=0.00..3.10 rows=4 width=8)
Optimizer: legacy query optimizer
```

Figure 2. Redistribution

The t1 and t2 tables have different distribution columns.

The t2 table is redistributed (redistribution motion) according to the distribution key of the t1 table and then joins with t1.




```
QUERY PLAN
------------------------------------------------------------------------------
-----------
Gather Motion 3:1  (slice2; segments: 3)  (cost=3.23..6.70 rows=10
width=16)
->  Hash Join  (cost=3.23..6.70 rows=4 width=16)
Hash Cond: t2.c2 = t1.c1
->  Redistribute Motion 3:3  (slice1; segments: 3)  (cost=0.00..3.33
rows=4 width=8)
Hash Key: t2.c2
->  Seq Scan on t2  (cost=0.00..3.11 rows=4 width=8)
->  Hash  (cost=3.10..3.10 rows=4 width=8)
->  Seq Scan on t1  (cost=0.00..3.10 rows=4 width=8)
Optimizer: legacy query optimizer
```

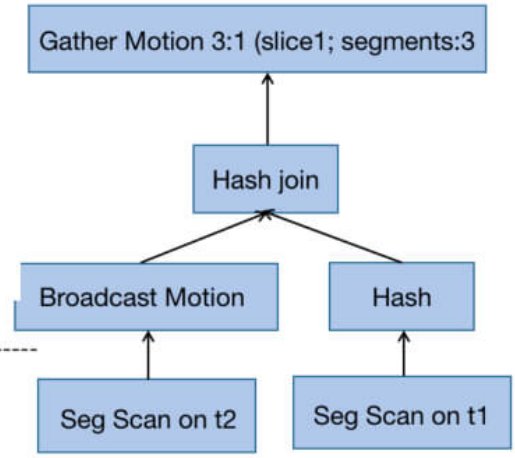
版权所有：腾讯云计算（北京）有限责任公司

Figure 3. Broadcast

The t1 and t2 tables have different distribution columns.

The t2 table is redistributed (broadcast motion) according to the distribution key of the t1 table and then joins with t1.

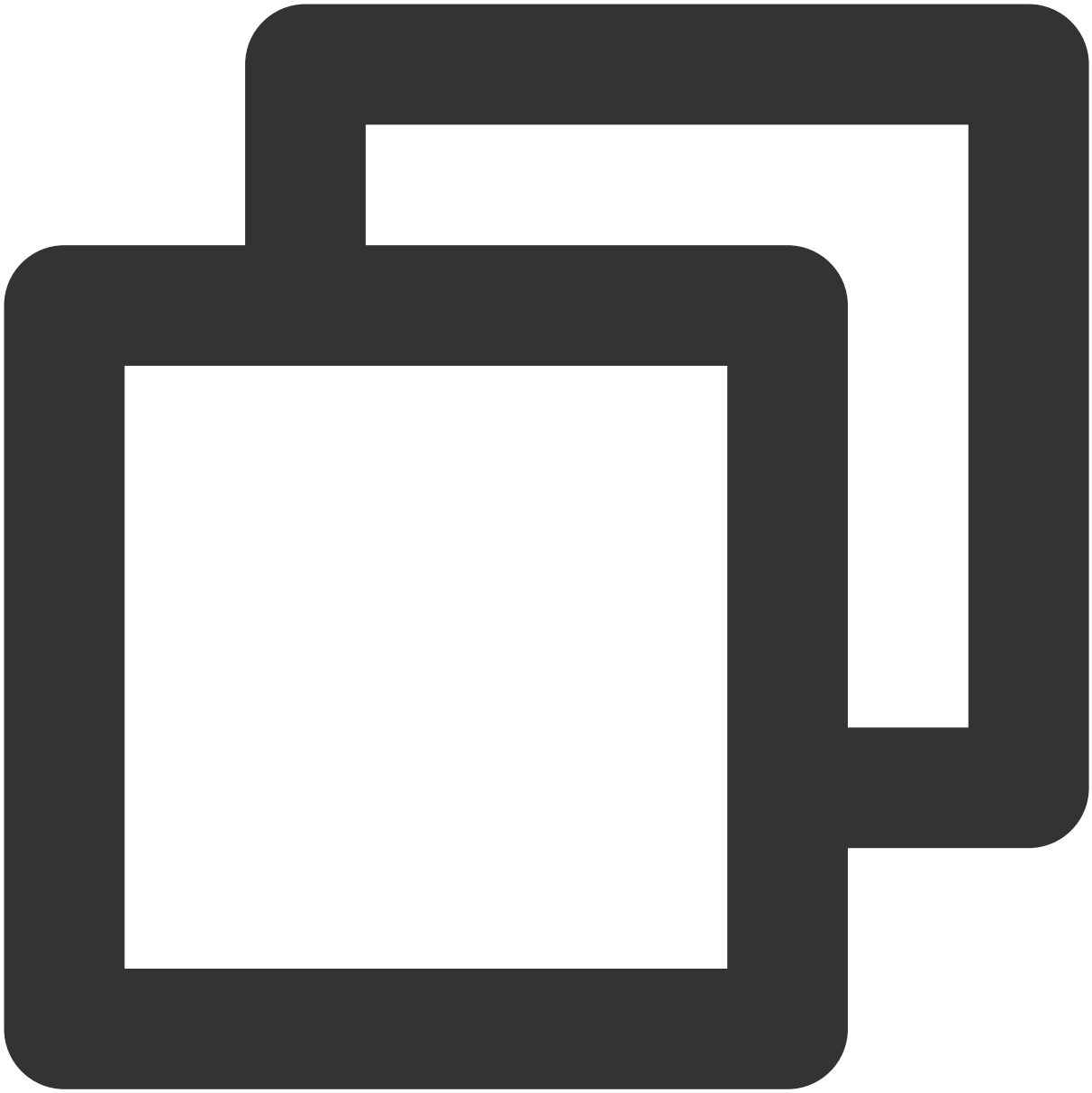
Broadcast motion is usually performed for small dimension tables.




```
QUERY PLAN
------------------------------------------------------------------------------------
---------
Gather Motion 3:1  (slice2; segments: 3)  (cost=3.25..6.96 rows=10
width=16)
->  Hash Join  (cost=3.25..6.96 rows=4 width=16)
Hash Cond: t1.c2 = t2.c2
->  Broadcast Motion 3:3  (slice1; segments: 3)  (cost=0.00..3.50
rows=10 width=8)
->  Seq Scan on t1  (cost=0.00..3.10 rows=4 width=8)
->  Hash  (cost=3.11..3.11 rows=4 width=8)
->  Seq Scan on t2  (cost=0.00..3.11 rows=4 width=8)
Optimizer: legacy query optimizer
```

表分布键的限制

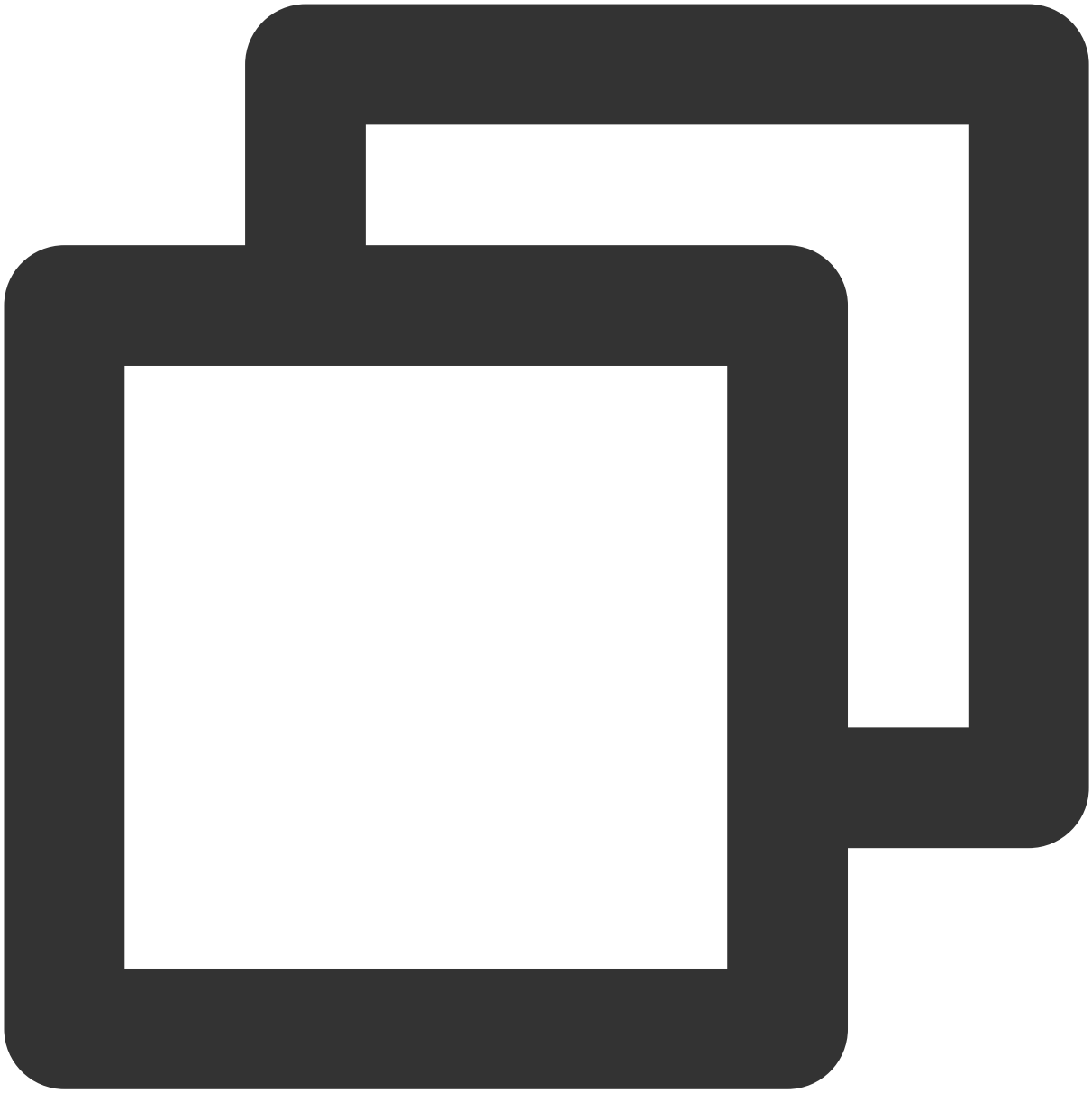
主键和唯一键必须包含分布键。例如：



```
create table t1(c1 int, c2 int, primary key (c1)) distributed by (c2);
会创建失败
```

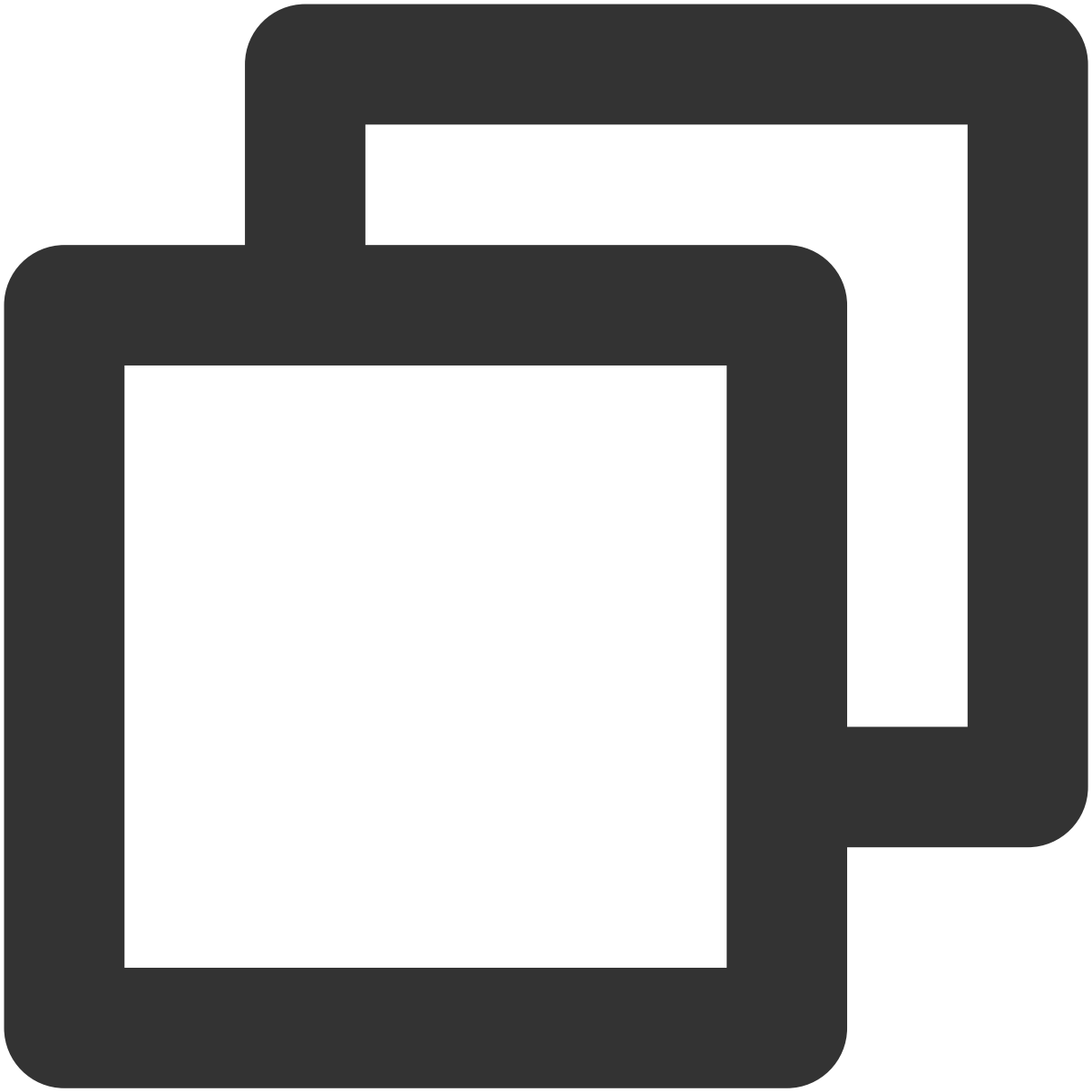
分布键创建合理性分析

当分布键创建不合理时，会导致表数据出现数据不一致的问题，可以用如下语句查看数据分布情况：



```
create table t1(c1 int, c2 int) distributed by (c1);
select gp_segment_id,count(1) from  t1 group by 1 order by 2 desc;
 gp_segment_id | count
---------------+--------
             0 |     1000
             1 |     68
(2 rows)
```

发现数据节点间差异过大时，可以修改分布键，以使数据更为均衡。



```
ALTER TABLE <table_name> SET WITH (REORGANIZE=true)
DISTRIBUTED BY (<distribution columns>);
```

表存储格式选择

最近更新时间：2024-02-19 15:58:15

本文介绍云数据仓库 PostgreSQL 如何选择存储格式。

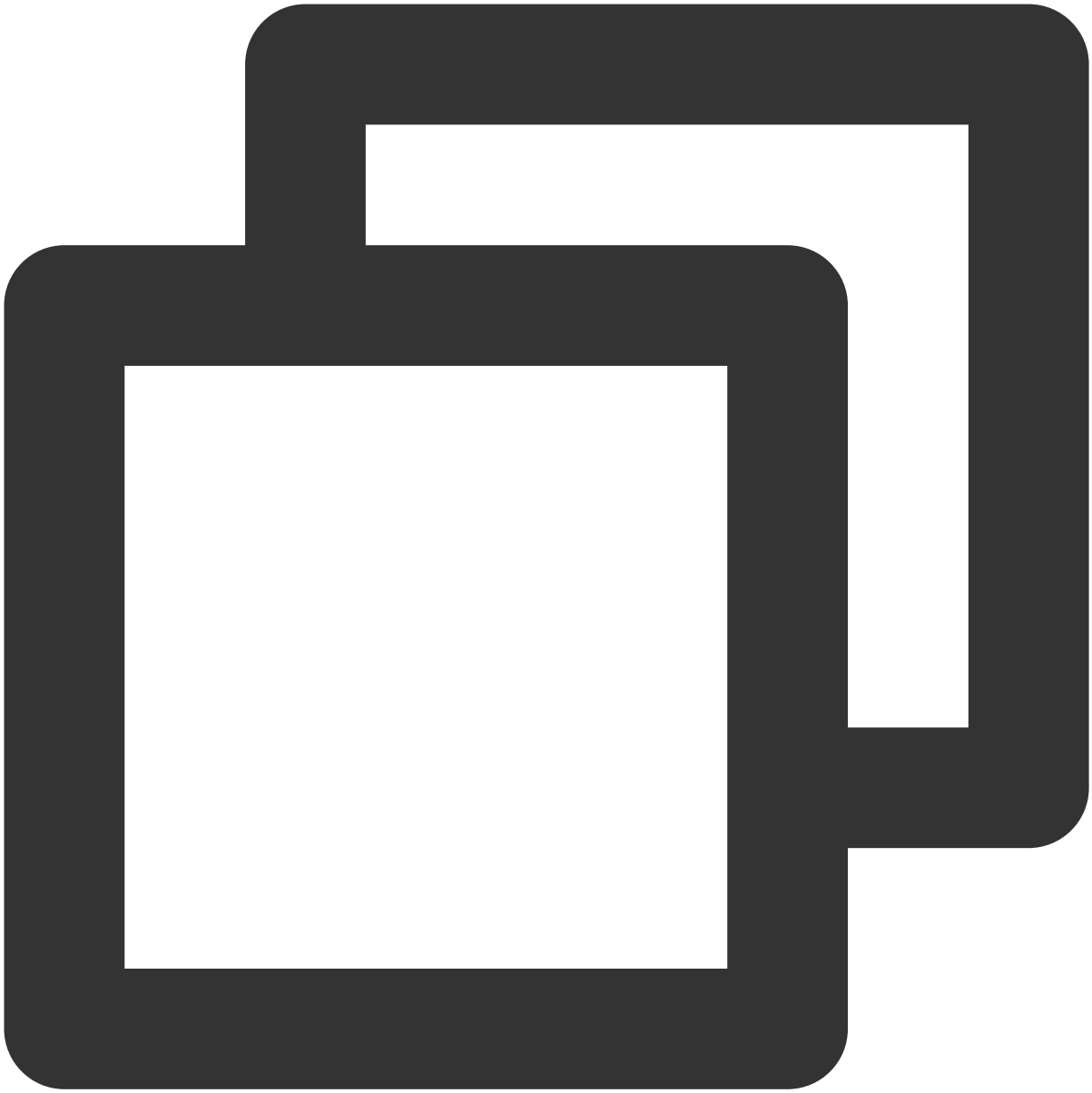
存储格式介绍

Greenplum（以下简称 GP）有2种存储格式，Heap 表和 AO 表（AORO 表，AOCO 表）。
Heap 表：这种存储格式是从 PostgreSQL 继承而来的，目前是 GP 默认的表存储格式，只支持行存储。
AO 表： AO 表最初设计是只支持 append 的（就是只能 insert），因此全称是 Append-Only，在4.3之后进行了优化，目前已经可以 update 和 delete 了，全称也改为 Append-Optimized。AO 支持行存储（AORO）和列存储（AOCO）。

HEAP 表

Heap 表是从 PostgreSQL 继承而来，使用 MVCC 来实现一致性。如果您在创建表的时候没有指定任何存储格式，那么 GP 就会使用 Heap 表。
Heap 表支持分区表，只支持行存，不支持列存和压缩。需要注意的是在处理 update 和 delete 的时候，Heap 表并没有真正删除数据，而只是依靠 version 信息屏蔽老的数据，因此如果您的表有大量的 update 或者 delete，表占用的物理空间会不断增大，这个时候需要依靠 vacuum 来清理老数据。
Heap 表不支持逻辑增量备份，因此如果要对Heap表做快照，每次都需要导出全量数据。
建表语句：

版权所有：腾讯云计算（北京）有限责任公司



```
CREATE TABLE heap(
    a int,
    b varchar(32)
) DISTRIBUTED BY (a);
```

最佳实践

如果该表是一张小表，例如数仓中的维度表，或者数据量在百万以下，推荐使用 Heap 表。
如果该表的使用场景是 OLTP 的，例如有较多的 update 和 delete，查询多是带索引的点查询等，推荐使用 Heap 表。

AO 表

AO 表设计的目的就是为了数仓中大型的事实表。AO 表支持行存(不推荐使用)和列存，并且也支持对数据进行压缩。

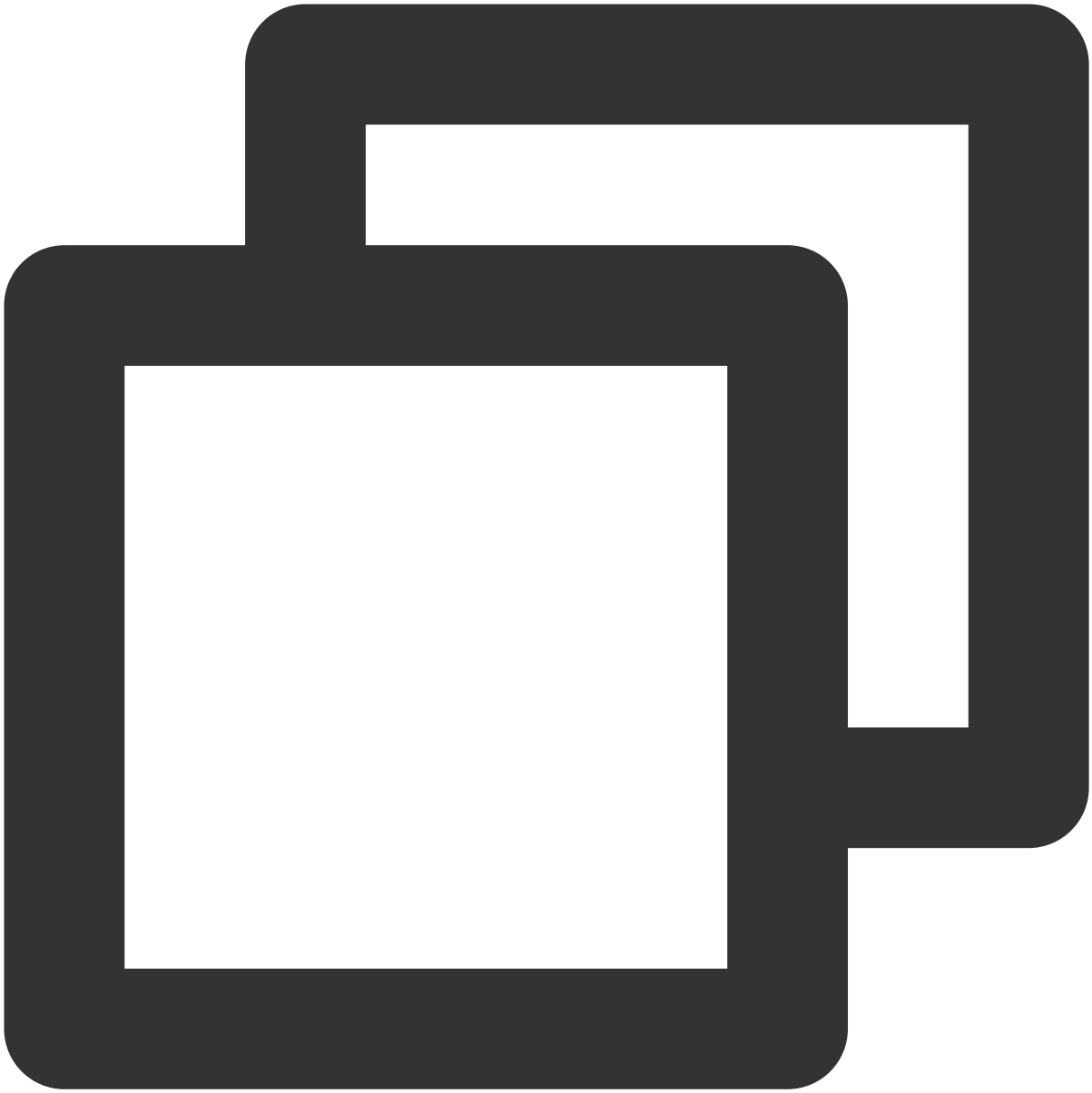
AO 表无论是在表的逻辑结构还是物理结构上，都与 Heap 表有很大的不同。例如上文所述 Heap 表使用 MVCC 控制 update 和 delete 之后数据的可见性，而 AO 表则使用一个附加的 bitmap 表来实现，这个表的内容就是表示 AO 表中哪些数据是可见的。

对于有大量 update 和 delete 的 AO 表，同样需要 vacuum 进行维护，不过在 AO 表中，vacuum 需要对 bitmap 进行重置并压缩物理文件，因此通常比 Heap 的 vacuum 要慢。

AO 列存

AOC 列存表按列方式组织数据，同时也支持列级别压缩。

建表语句如下，这里还加入了分区特性：



```
CREATE TABLE aoco(
    a int  ENCODING (compresstype=zlib, compresslevel=5),
    b int  ENCODING (compresstype=none),
    c varchar(32) ENCODING (compresstype=RLE_TYPE, blocksize=32768),
    d varchar(32),
    fdate date
)
WITH (appendonly=true, orientation=column, compresstype=zlib, compresslevel=6, bloc
DISTRIBUTED BY (a)
PARTITION BY RANGE(fdate)
(
```

```
    PARTITION pn START ('2018-11-01'::date) END ('2018-11-10'::date) EVERY ('1 day'
    DEFAULT PARTITION pdefault
);
```

压缩

压缩主要用于列存表或者追加写（"appendonly=true"）的行存表，有以下两种类型的压缩可用。

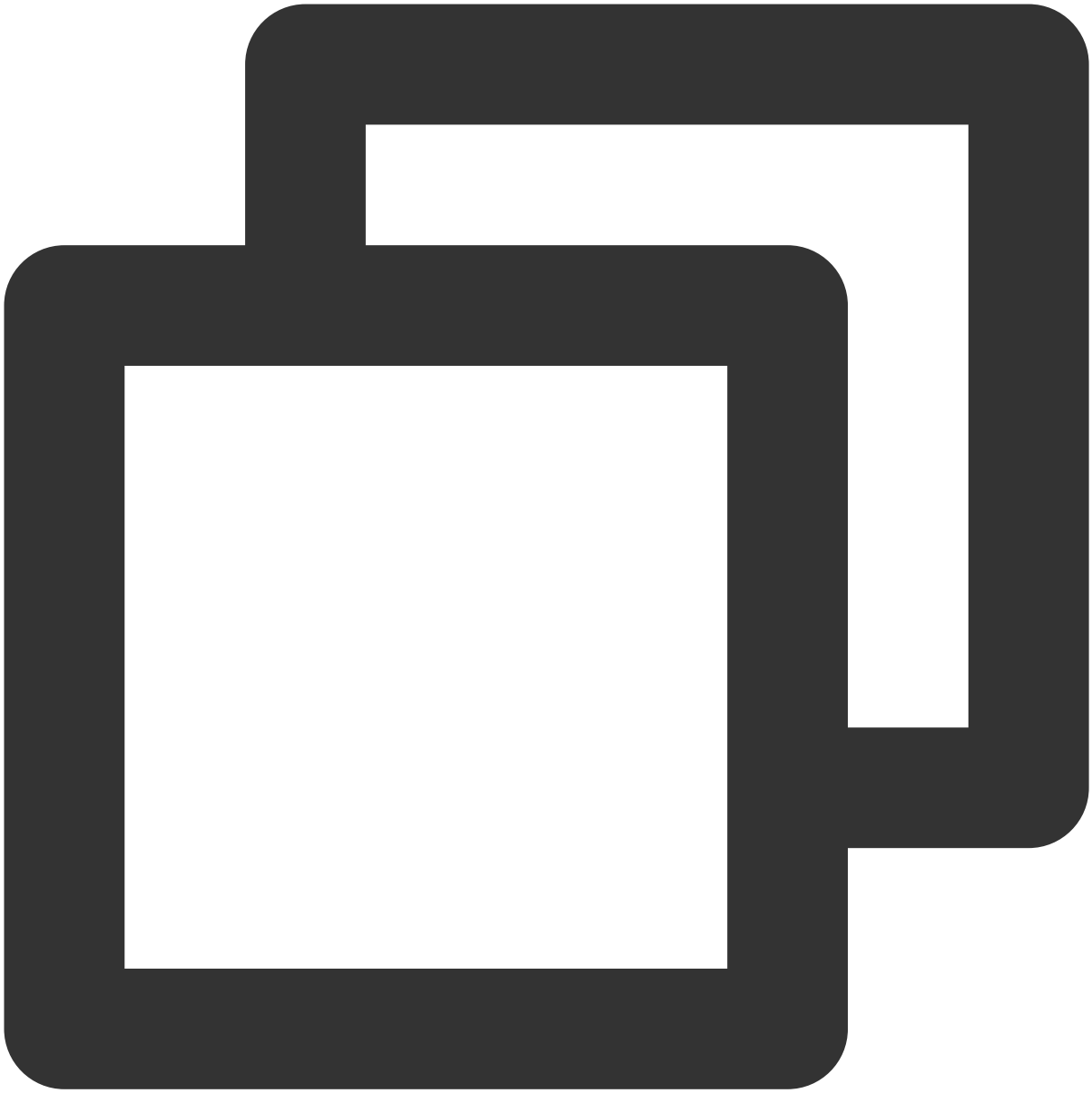
应用于整个表的表级压缩。

应用到指定列的列级压缩。用户可以为不同的列应用不同的列级压缩算法。

目前腾讯云数据仓库 PostgreSQL 支持 zstd、zlib、rle_type。

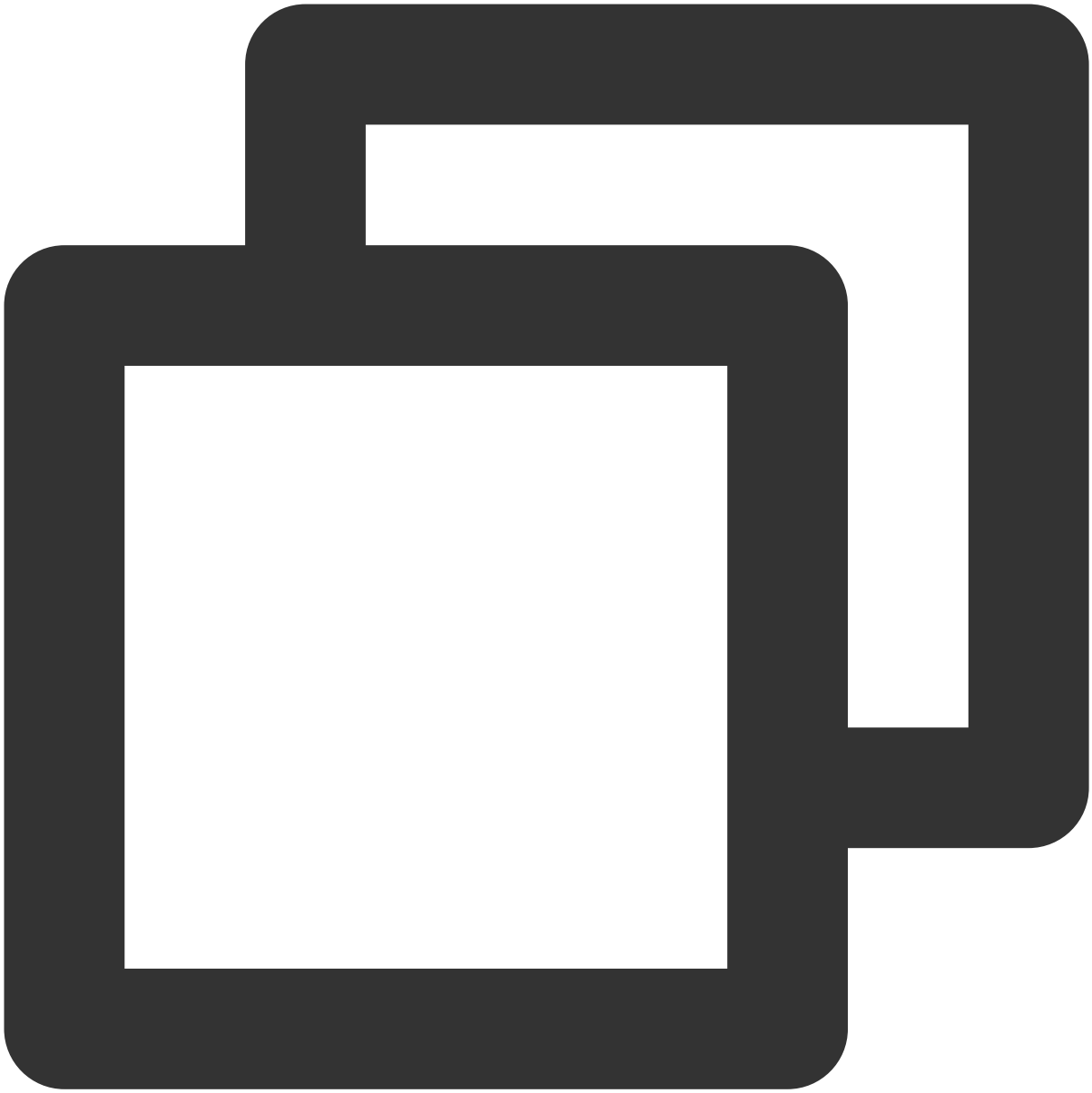
示例：

创建一个使用 zlib 压缩且压缩级别为5的列存表。



```
CREATE TABLE foo (a int, b text)
   WITH (appendonly=true, orientation=column, compresstype=zlib, compresslevel=5);
```

创建一个使用 zstd 压缩且压缩级别为5的列存表。



```
CREATE TABLE foo (a int, b text)
   WITH (appendonly=true, orientation=column, compresstype=zstd, compresslevel=5);
```

最佳实践

AOCO 表通常用于数仓中的核心事实表，这种表字段多，数据量大，主要是用于 OLAP 场景，也就是查询的过程不会S ELECT * FROM，而是对其中部分字段进行读取和聚合。
由于 AOCO 表一般用于大表，因此经常搭配压缩和分区，以减少表的实际存储量来提升性能。

版权所有：腾讯云计算（北京）有限责任公司

一般情况下，压缩格式选择 zlib，压缩级别可以采用折中的4或者5，但是对于有大量重复值的字段，记得要采用 RLE_TYPE 压缩格式。

blocksize 不要设置过大，特别是对于分区表，GP 对于每个分区的每个字段都会维护一个 buffer，blocksize 过大，会导致消耗的内存过大，通常就采用默认值32768即可。

版权所有：腾讯云计算（北京）有限责任公司

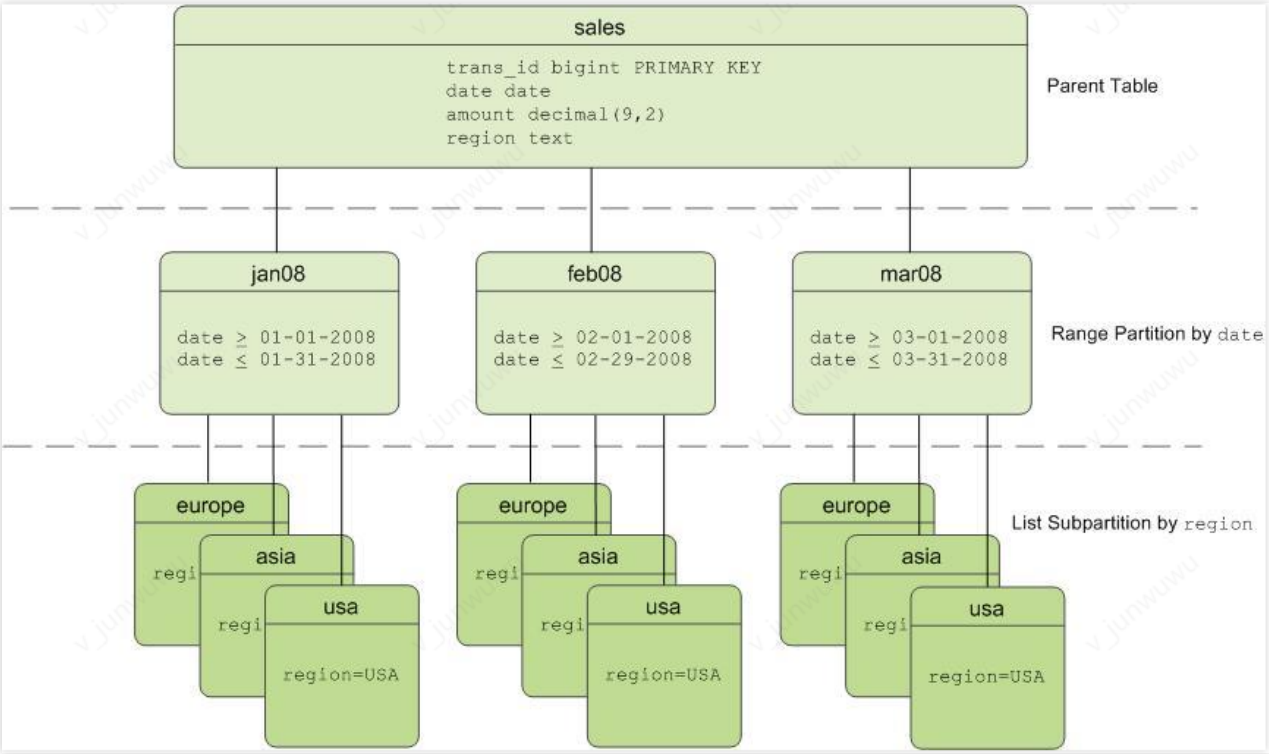
表分区使用

最近更新时间：2024-02-19 15:58:16

本文介绍云数据仓库 PostgreSQL 如何使用表的分区能力。

什么是分区表

分区表就是将一个大表在物理上分割成若干小表，并且整个过程对用户是透明的，也就是用户的所有操作仍然是作用在大表上，不需要关心数据实际上落在哪张小表里面。云数据仓库 PostgreSQL 中分区表的原理和 PostgreSQL 一样，都是通过表继承和约束实现的。
分区表示例如下：



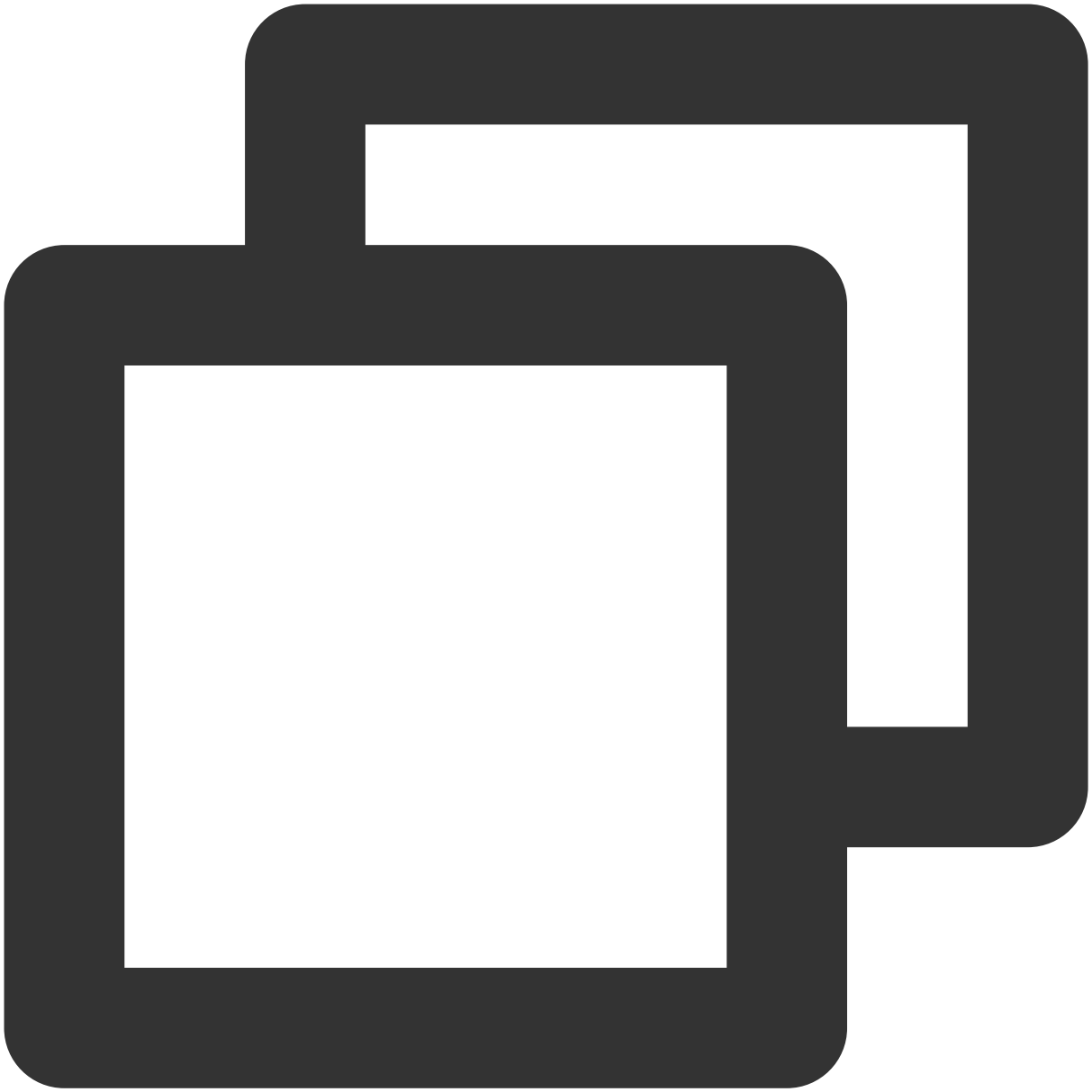
使用分区表场景

是否使用分区表，可以通过以下几个方面进行考虑：
表数据量是否足够大：通常对于大的事实表，例如数据量有几千万或者过亿，我们可以考虑使用分区表，但数据量大小并没有一个绝对的标准可以使用，一般是根据经验，以及对目前性能是否满意。

表是否有合适的分区字段：如果数据量足够大了，这个时候我们就需要看下是否有合适的字段能够用来分区，通常如果数据有时间维度，例如按天，按月等，是比较理想的分区字段。
表内数据是否具有生命周期：通常数仓中的数据不可能一直存放，一般都会有一定的生命周期，例如最近一年等，这里就涉及到对旧数据的管理，如果有分区表，就很容易删除旧的数据，或者将旧的数据归档到 对象存储 等更为廉价的存储介质上。
查询语句中是否含有分区字段：如果您对一个表做了分区，但是所有的查询都不带分区字段，这不仅无法提高性能反而会使性能下降，因为所有的查询都会扫描所有的分区表。

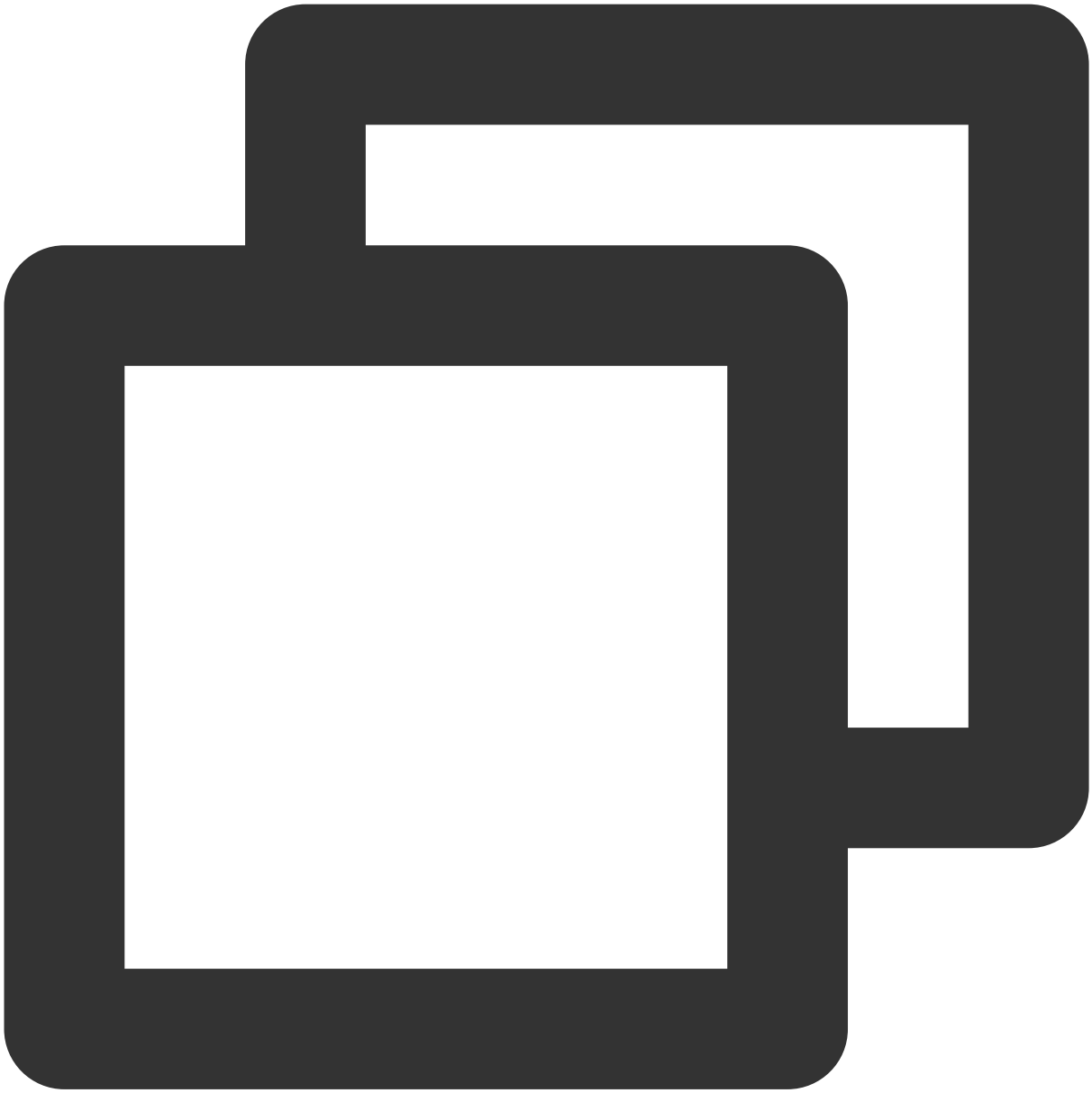
创建分区表

范围分区（Range Partition）
列表分区（List Partition）
组合分区（A combination of both types）
范围分区示例：



```
CREATE TABLE test_range_partition
(
    uid int,
    fdate character varying(32)
)
PARTITION BY RANGE(fdate)
(
        PARTITION p1 START ('2018-11-01') INCLUSIVE END ('2018-11-02') EXCLUSIVE,
        PARTITION p2 START ('2018-11-02') INCLUSIVE END ('2018-11-03') EXCLUSIVE,
        DEFAULT PARTITION pdefault
);
```

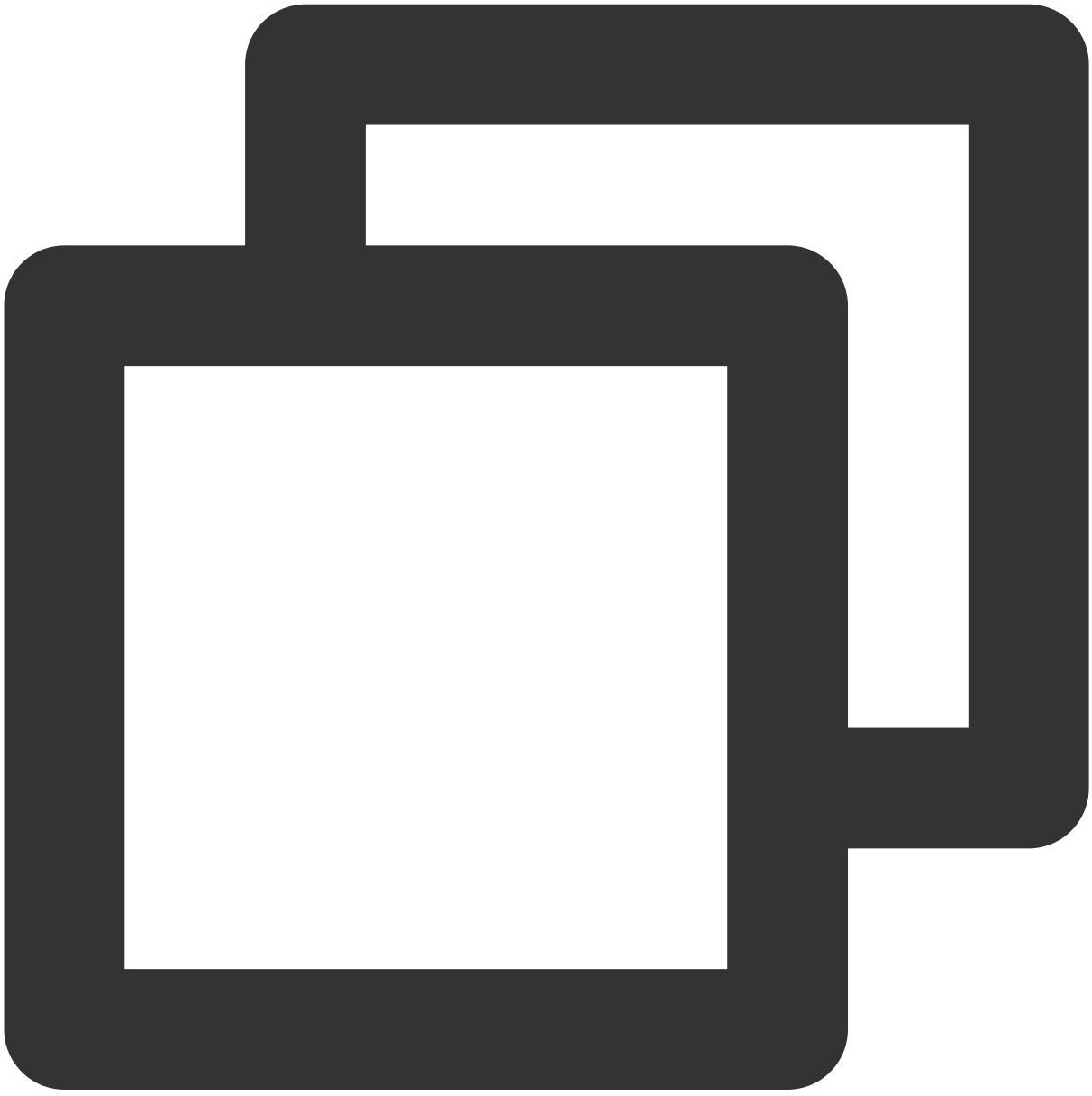
以上例子是按天建表，如果时间跨度比较大，会导致建表语句很长，书写起来也不方便，这时候可以使用以下语法：



```
CREATE TABLE test_range_partition_every_1
(
    uid int,
    fdate date
)
partition by range (fdate)
(
```

```
    PARTITION pn START ('2018-11-01'::date) END ('2018-12-01'::date) EVERY ('1 day'
    DEFAULT PARTITION pdefault
);
```

列表分区示例：



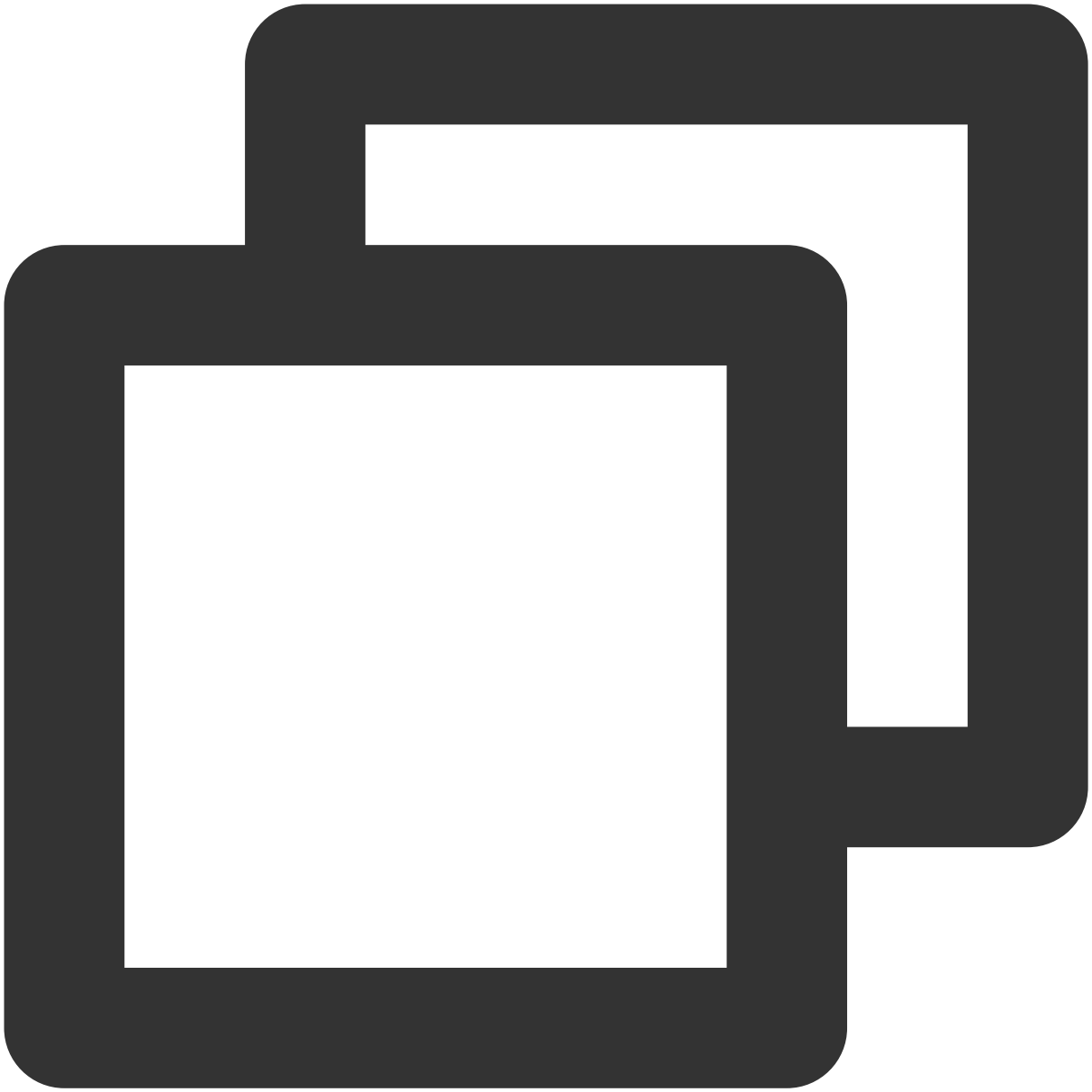
```
CREATE TABLE test_list_partition
(
    uid int,
    gender char(1)
)
PARTITION BY LIST (gender)
```

```
(
    PARTITION girls VALUES ('F'),
    PARTITION boys VALUES ('M'),
    DEFAULT PARTITION pdefault
);
```

管理分区表

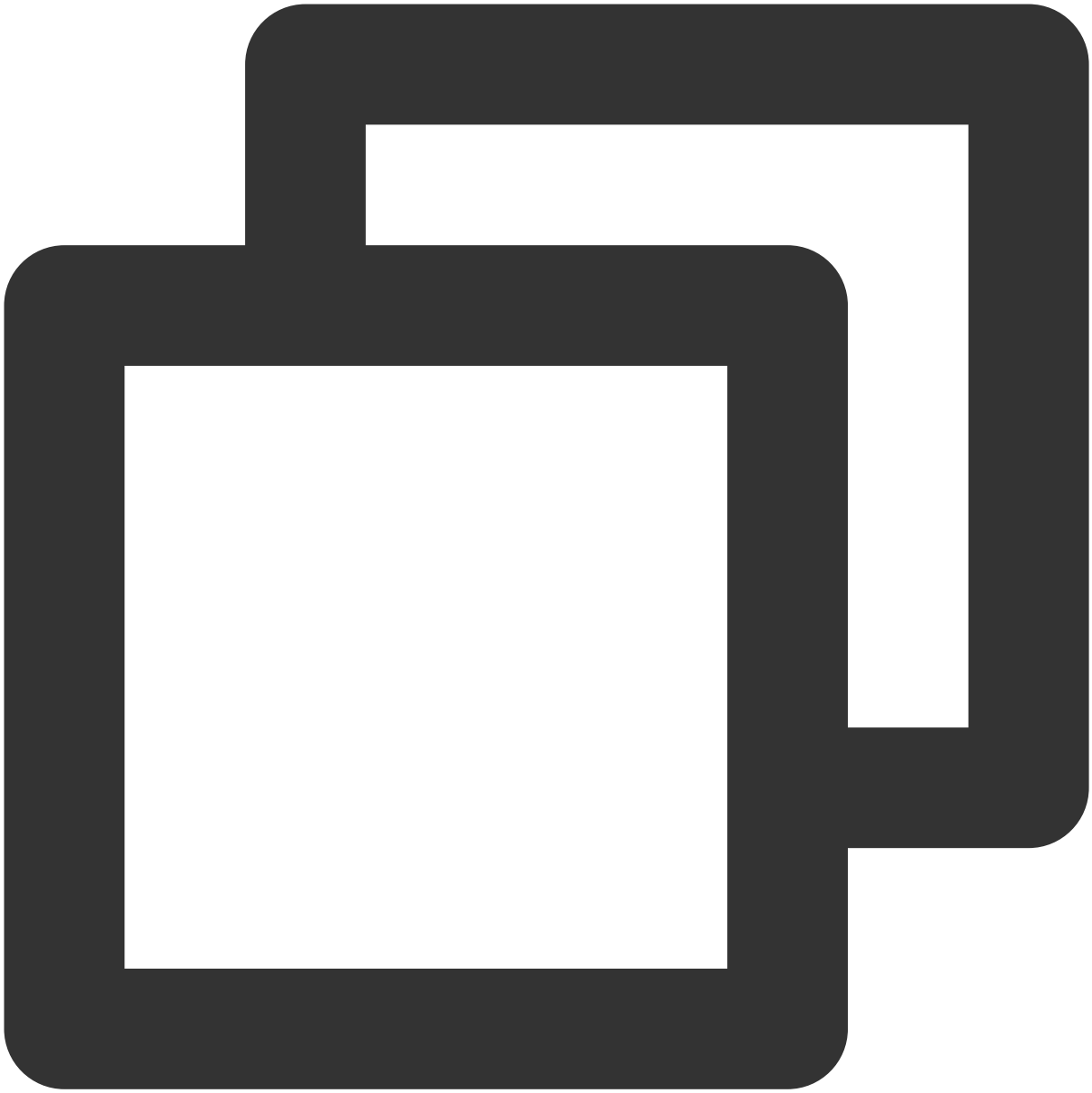
分区表也是一张表，所以对于表的很多操作也可以作用于分区表上，这里列举了常用的一些操作：

清空子分区



```
ALTER TABLE test_range_partition TRUNCATE PARTITION p1;
```

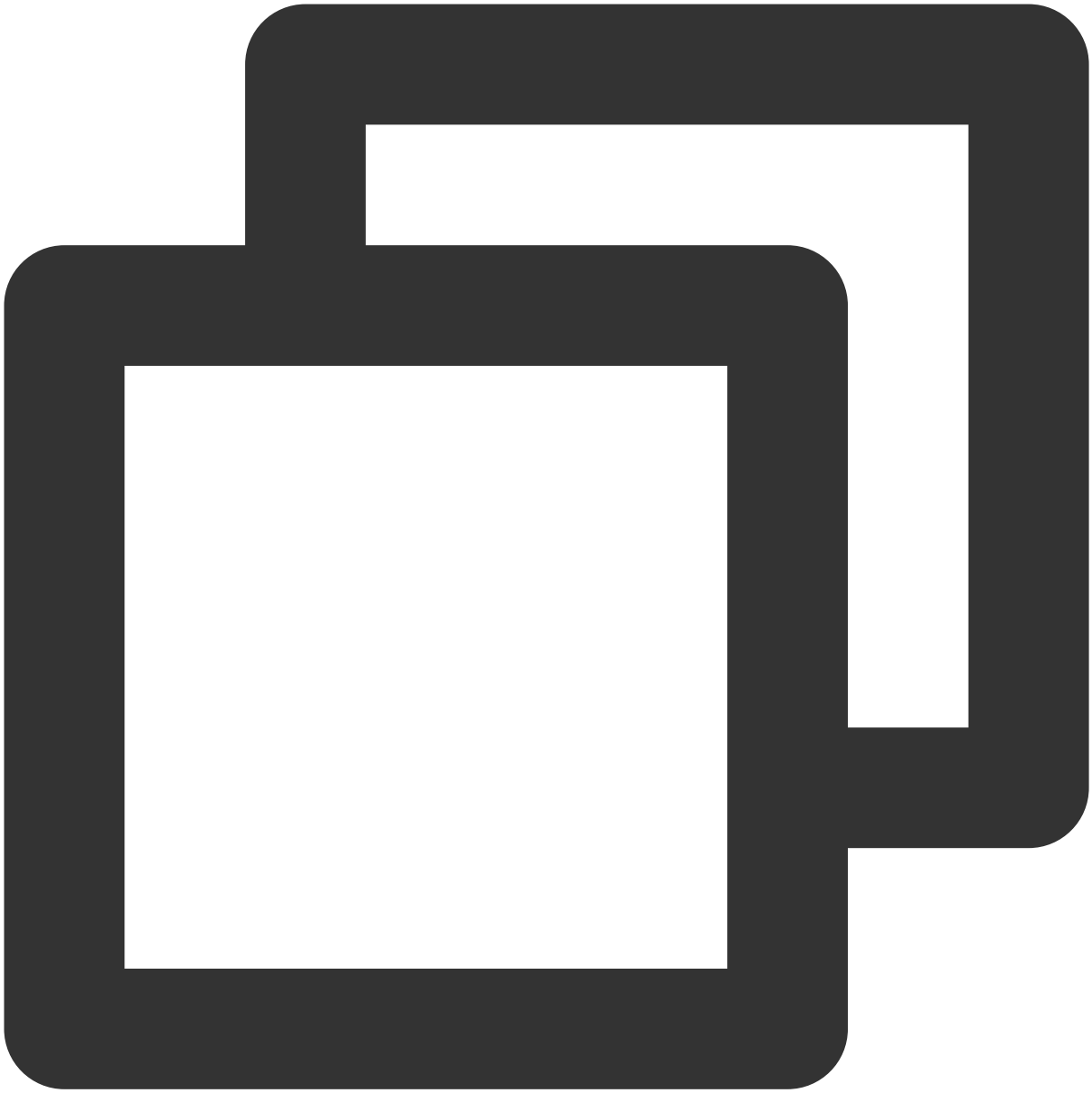
删除子分区



```
ALTER TABLE test_range_partition DROP PARTITION p1;
```

说明：

DROP PARTITION 之后跟的是 partition name，而不是 partition table name，这两者之间是有区别的，如果是使用 EVERY 语法创建的分区表，您需要通过 pg_partitions 表查询到对应分区的 partition name。



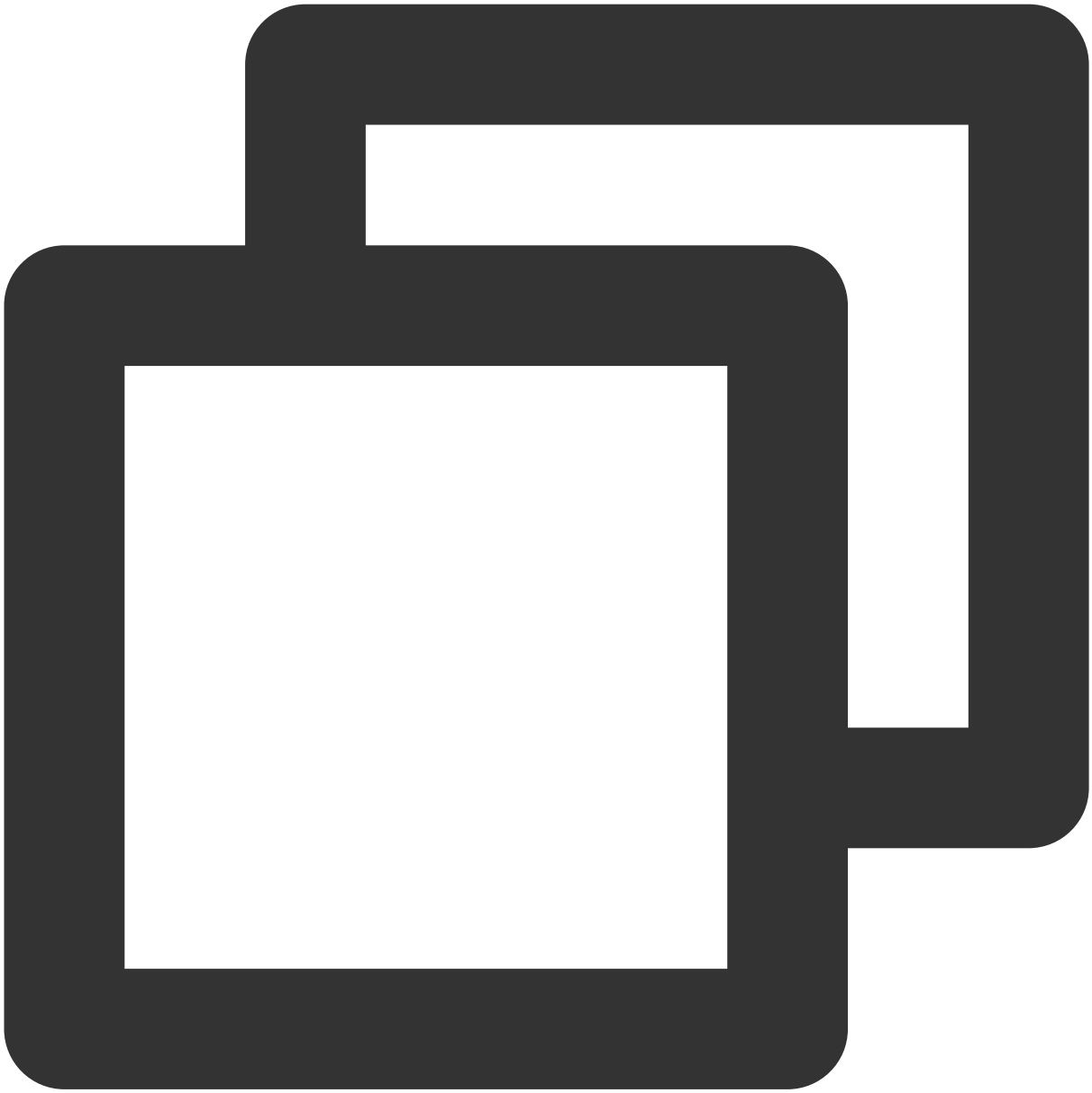
```
ALTER TABLE test_range_partition ADD PARTITION p3 START ('2018-11-03') INCLUSIVE EN
```

说明：

如果分区表中含有 DEFAULT 分区，会出现如下错误 `ERROR:  cannot add RANGE partition "p3" to relation "test_range_partition" with DEFAULT partition "pdefault"`，解决办法可以参见滚动分区。

滚动分区

通常按时间分区的表，都有一个特性，就是分区会不断往前滚动，例如一个按天分区，保存最近10天的分区表，每到新一天，就会要删除10天前的分表，并且创建一个新的分区表容纳最新的数据。
如果是含有默认分区的，可以使用分区 Split。

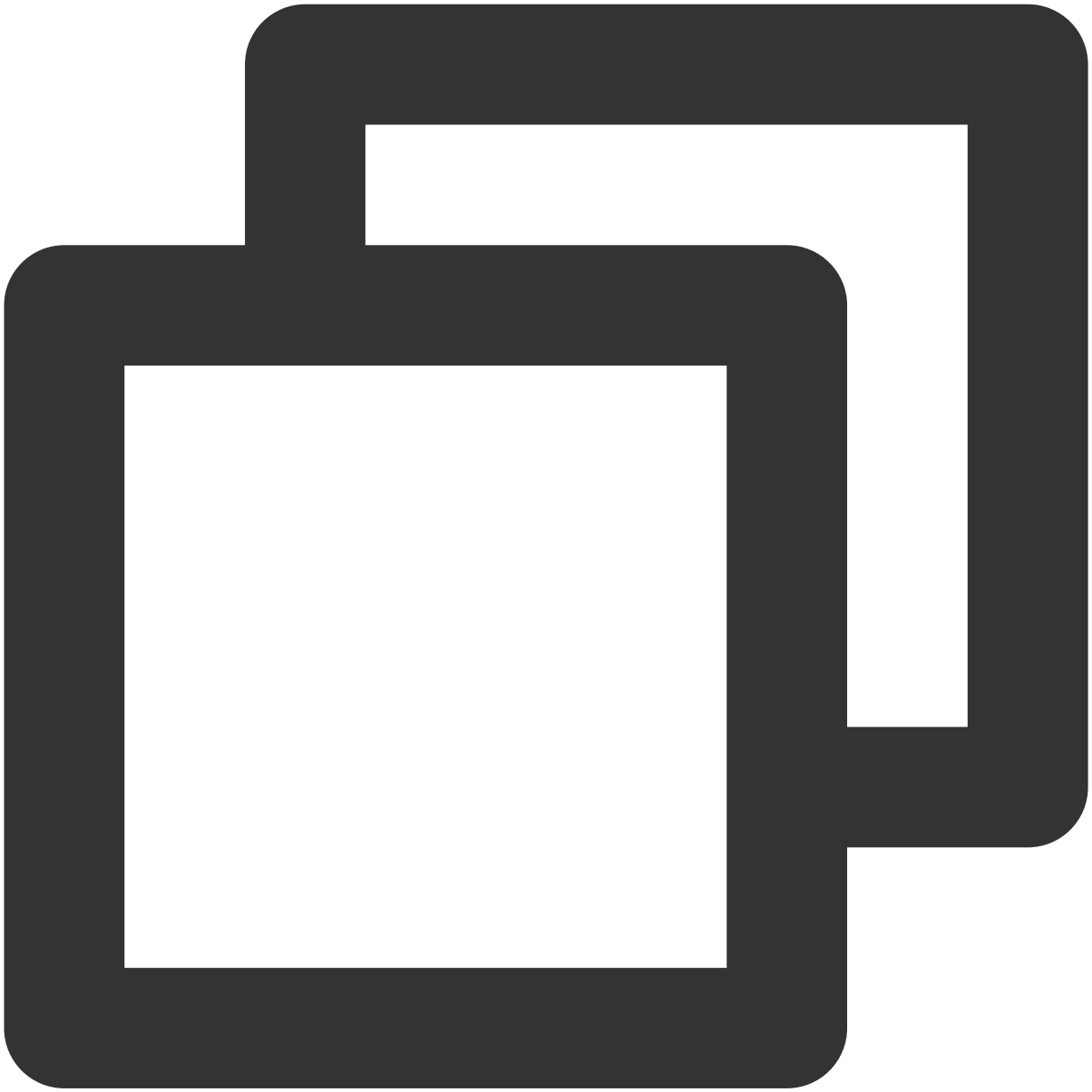


```
ALTER TABLE test_range_partition SPLIT DEFAULT PARTITION START ('2018-11-03') INCLU
```

这样新分区就被添加，同时保留了默认分区，然后在删除老的分区就完成新老分区的更替。

交换分区

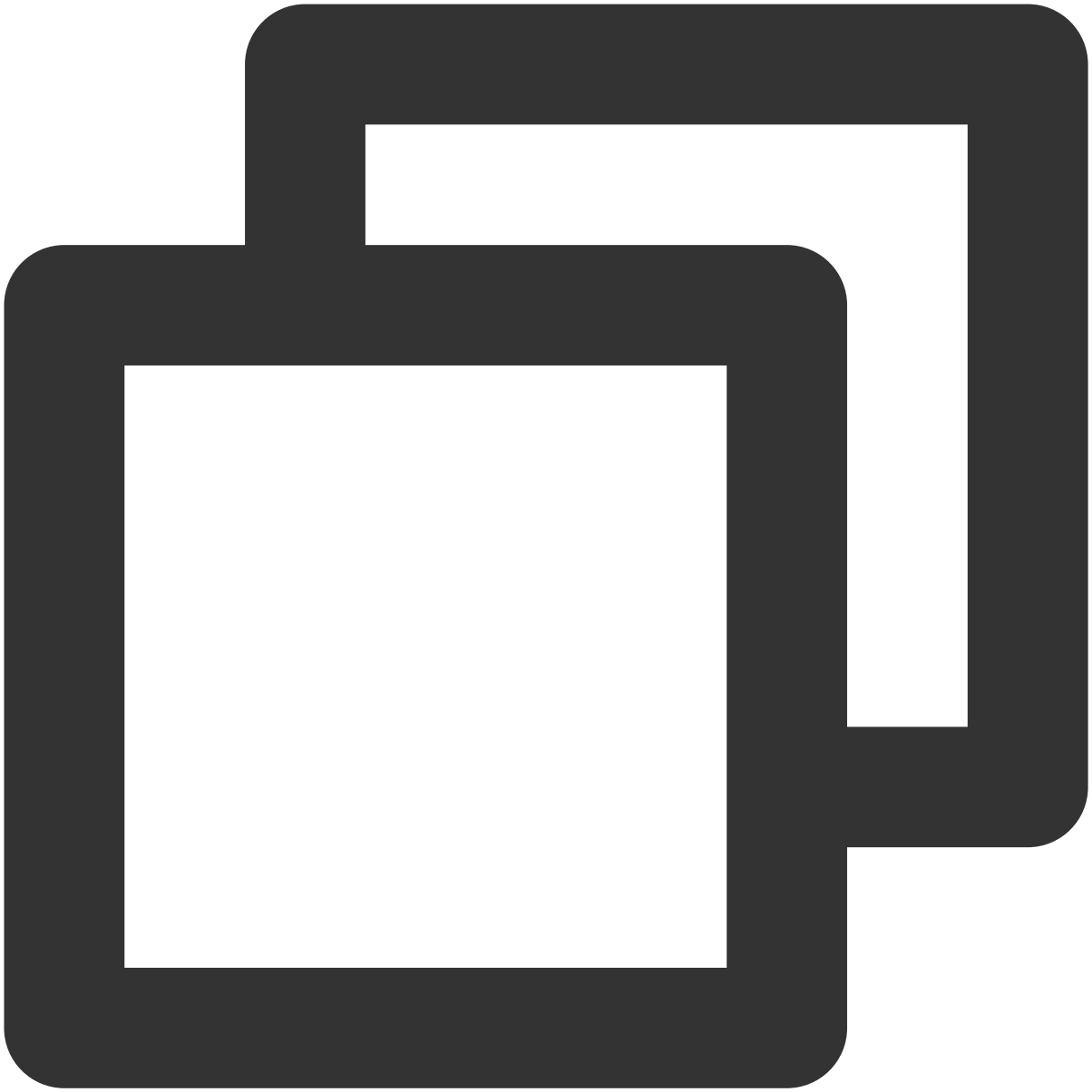
交换分区就是将一张普通的表和某张分区表进行交换，这个功能在数据分层存储十分有用。
例如我们会需要根据对象存储的不同目录设置分区，这个需求就可以使用交换分区完成，这样对于一张大表，他的较少查询的历史数据就可以放在对象存储上，语法如下：



```
ALTER TABLE {table_name} EXCHANGE PARTITION {partition_name|FOR (RANK(number))|FOR
```

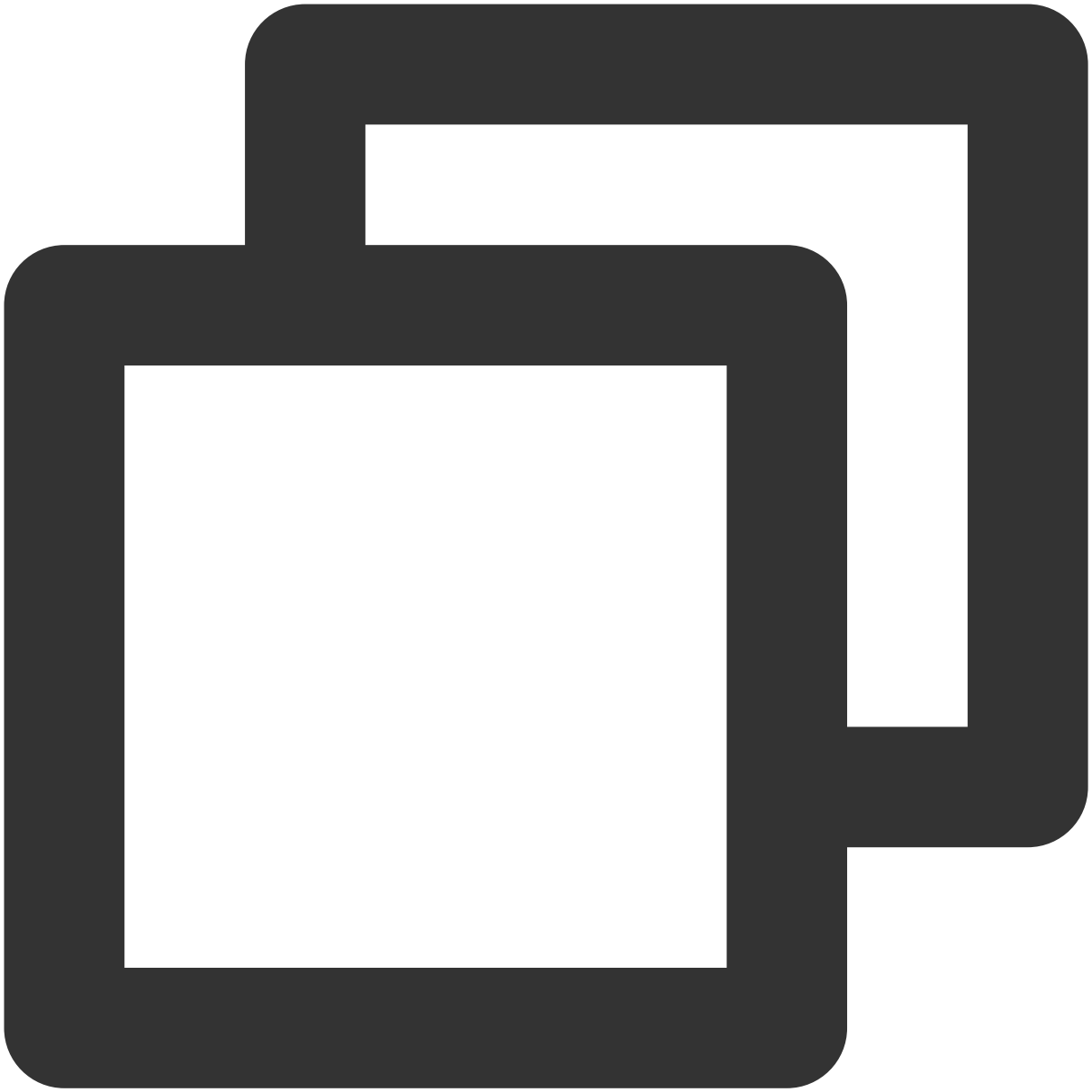
查询分区

与分区相关的系统表或者视图如下：



```
pg_partition
pg_partition_columns
pg_partition_encoding
pg_partition_rule
pg_partition_templates
pg_partitions
```

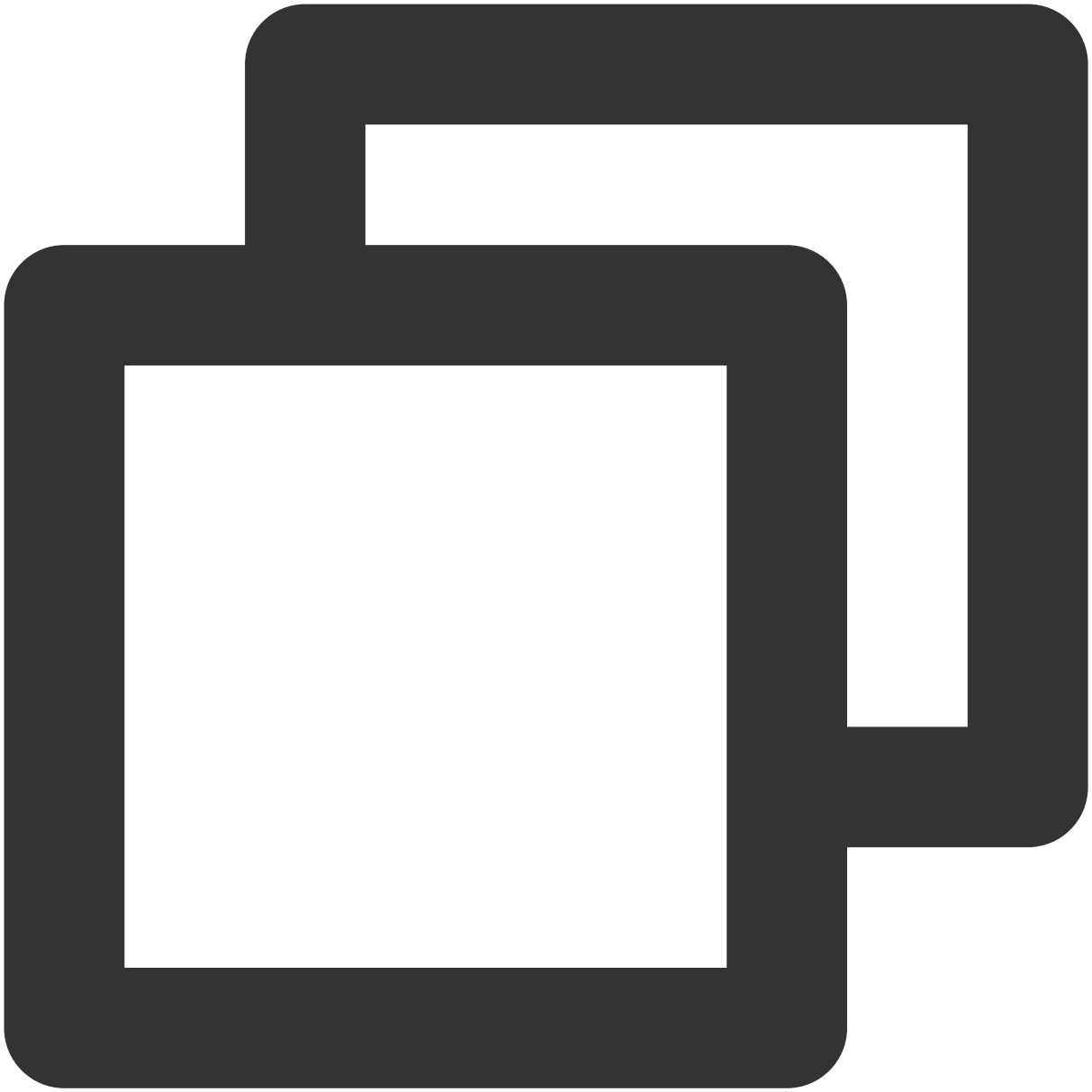
查看分区基本信息



```
t2=# select * from pg_partitions where partitiontablename = 'test_range_partition_1
-[ RECORD 1 ]------------+-----------------------------------------------------
schemaname               | public
tablename                | test_range_partition
partitionschemaname      | public
partitiontablename       | test_range_partition_1_prt_p1
partitionname            | p1
parentpartitiontablename |
parentpartitionname      |
partitiontype            | range
partitionlevel           | 0
```

```
partitionrank             | 1
partitionposition         | 2
partitionlistvalues       |
partitionrangestart       | '2018-11-01'::character varying(32)
partitionstartinclusive   | t
partitionrangeend         | '2018-11-02'::character varying(32)
partitionendinclusive     | f
partitioneveryclause      |
partitionisdefault        | f
partitionboundary         | PARTITION p1 START ('2018-11-01'::character varying(32))
parenttablespace          | pg_default
partitiontablespace       | pg_default
```

查看分区定义



```
t2=# select pg_get_partition_def('test_range_partition'::regclass,true);
-[ RECORD 1 ]--------+-------------------------------------------------------------
pg_get_partition_def | PARTITION BY RANGE(fdate)
                     |           (
                     |           PARTITION p1 START ('2018-11-01'::character varyin
                     |           PARTITION p2 START ('2018-11-03'::character varyin
                     |           DEFAULT PARTITION pdefault
                     |           )
```

分区表使用最佳实践

分区的粒度

通常像范围分区的表都涉及到粒度问题，例如按时间分表，究竟是按天，按周，按月等。粒度越细，每张表的数据就越少，但是分区表的数量就会越多，反之亦然。
关于分区表的数量，这里没有绝对的标准，一般来说分区表的数量在100左右已经算是比较多了。
分区表数目过多，会有多方面的影响，例如查询优化器生成执行计划较慢，同时很多维护工作也都会变慢，例如 vacuum，recovering segment，expanding the cluster, checking disk usage 等。

查询语句

为了充分利用分区表的优势，需要在查询语句中尽量带上分区条件。最终目的是扫描尽量少的分区表。

插件使用

最近更新时间：2024-02-19 15:58:15

背景说明

云数据仓库 PostgreSQL 是基于 PostgreSQL 的 MPP 集群架构，因此也兼容部分 PostgreSQL 生态的插件。本文列出了云数据仓库 PostgreSQL 目前支持的插件类型以及使用方法，若您有其它插件的支持需求，可 联系我们 咨询。

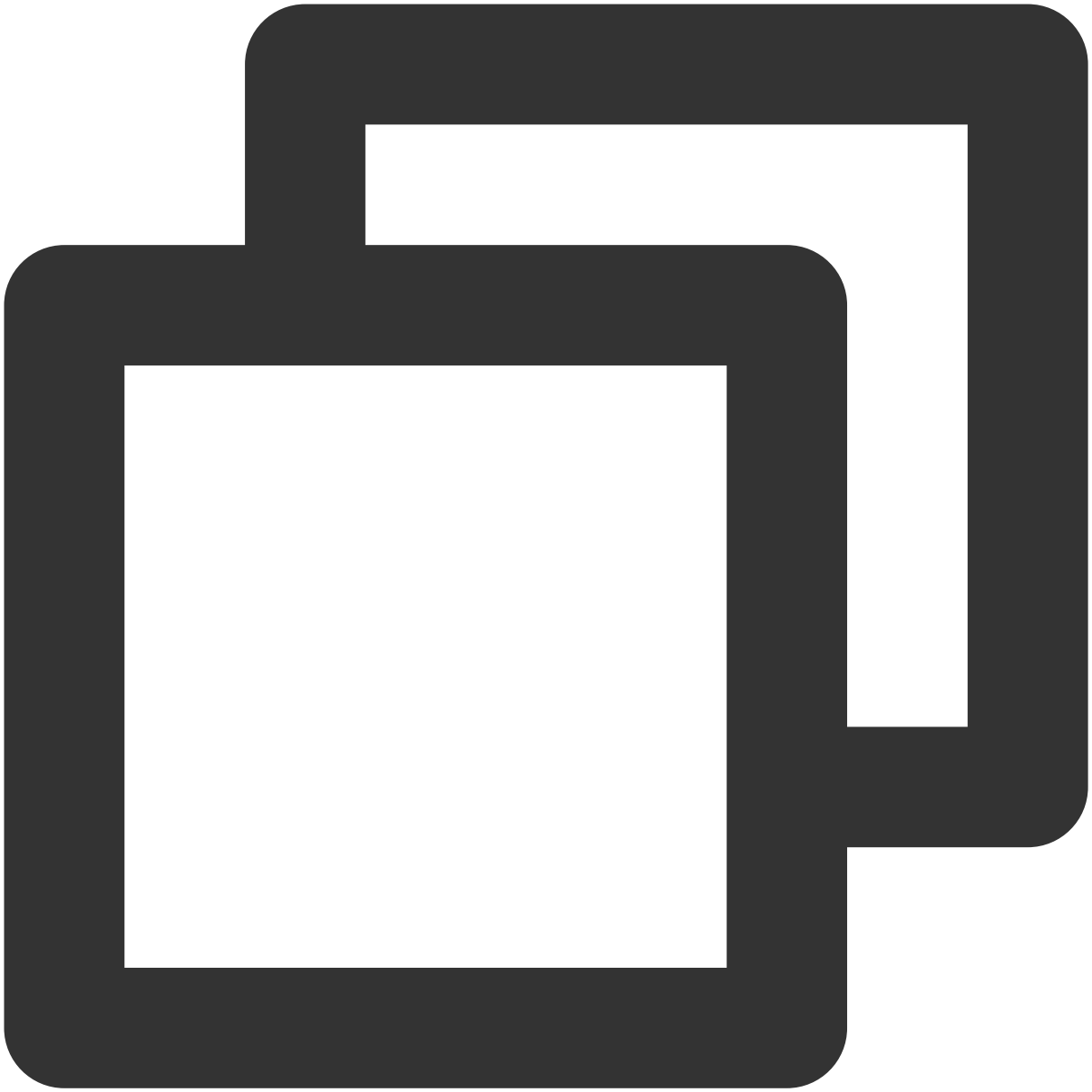
插件列表

postgis：版本2.5.2，空间数据库插件，具体可参考 geospatial git。
hll：版本2.14，HyperLogLog 算法插件，具体可参考 postgresql-hll git。
roaringbitmap：版本0.2.66，Bitmap 压缩算法插件，具体可参考 gpdb-roaringbitmap。
orafce：版本3.7，oracle 函数兼容插件，具体可参考 orafce git。
pgcrypto：版本1.1，加密插件，具体可参考 postgresql pgcrypto。
fuzzystrmatch：版本1.0，计算字符串相似点距离插件，具体可参考 postgresql fuzzystrmatch。

使用说明

云数据仓库 PostgreSQL 默认未创建以上插件，用户可以根据自己的需求自行创建或删除，语法如下：

版权所有：腾讯云计算（北京）有限责任公司

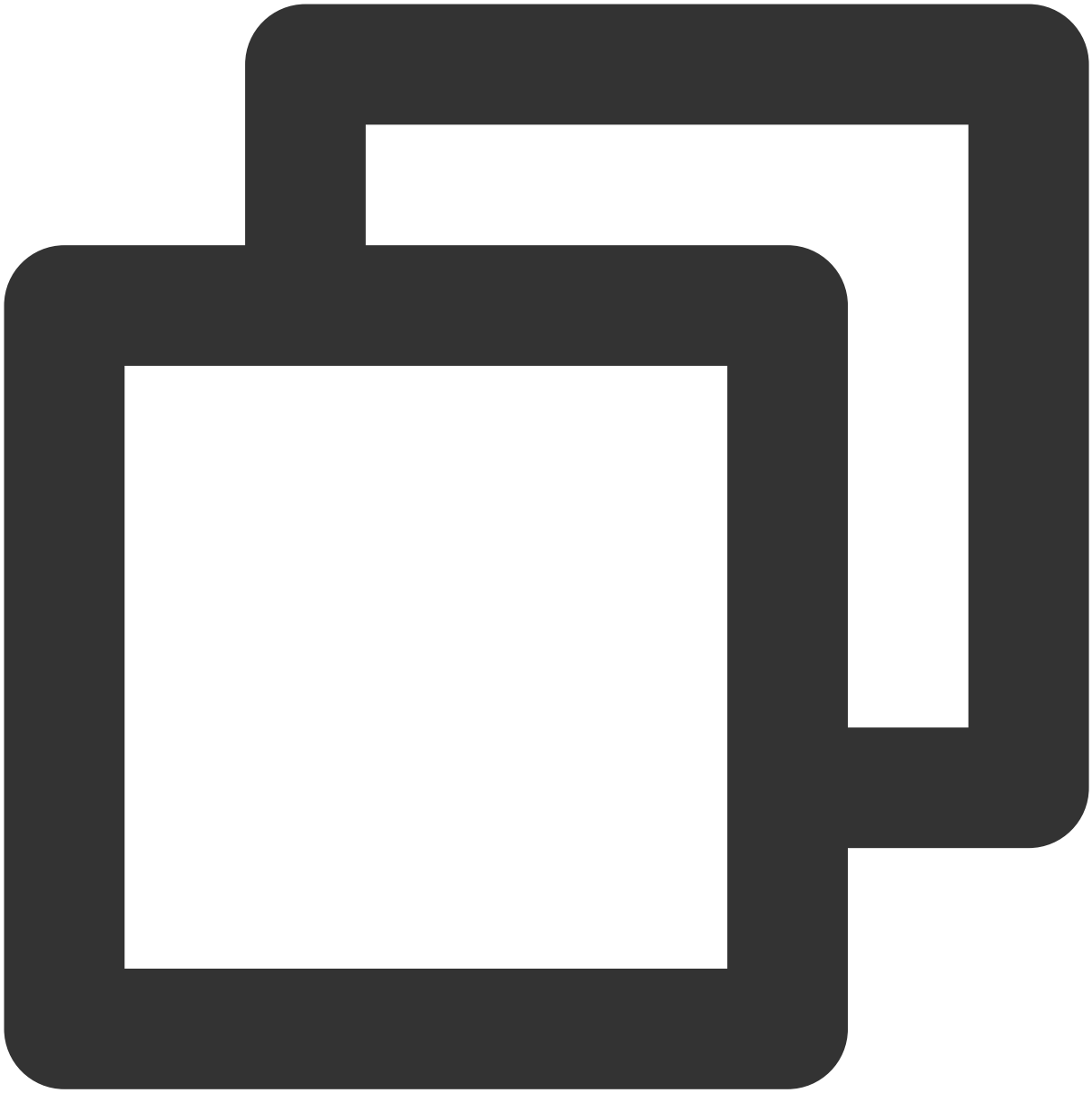


```
CREATE EXTENSION {extension name};
DROP EXTENSION {extension name}
```

注意：

插件的作用域是数据库，也就是在每一个需要使用插件的数据库内，都需要先运行创建语句。

查看当前数据库安装的插件以及插件版本，可使用如下语法：



```
test_db=> \\dx
                                                    List of installed extensions
  Name   | Version |   Schema   |                                                Descript
ion
---------+---------+------------+-----------------------------------------------------
-----------------------------------------------
 hll     | 2.14    | public     | type for storing hyperloglog data
 orafce  | 3.7     | public     | Functions and operators that emulate a subset of
functions and packages from the Oracle RDBMS
 plpgsql | 1.0     | pg_catalog | PL/pgSQL procedural language
(3 rows)
```

冷备数据

最近更新时间：2024-02-19 15:58:16

本文主要介绍如何在业务侧周期备份数据。

背景

尽管云数据仓库 PostgreSQL 对数据做了主备，但是在某些场景下，仍然需要对重要数据进行全量冷备，例如异常删除数据。由于目前云数据仓库 PostgreSQL 暂不支持自动冷备数据，因此需要业务侧手动完成相关工作。在云数据仓库 PostgreSQL 中，数据备份使用 COS 作为存储介质，对 COS 数据操作可参考 使用外表高速导入或导出 COS 数据。

影响

使用本文提及方法进行数据备份，会对集群造成以下影响，需要提前注意：
1. 脚本运行会提高集群负载，特别网络侧开销较大，建议评估好备份时间，在业务低峰期进行。
2. 脚本运行会在每个库创建一个 COS 插件。
3. 脚本运行会对每张需要备份的表创建一张 COS 外表，备份结束后会进行删除。

问题

使用本文提交方法进行数据备份，可能会遇到以下问题：

报错信息	处理办法
`ERROR: permission denied for external protocol cos`	`GRANT ALL ON PROTOCOL cos TO {backup_user}`
`ERROR: permission denied for schema {schame_name}`	`GRANT ALL ON SCHEMA {schame_name} to {backup_user}`
`ERROR: permission denied for relation {table_name}`	`GRANT SELECT ON {table_name} to {backup_user}`

步骤

版权所有：腾讯云计算（北京）有限责任公司

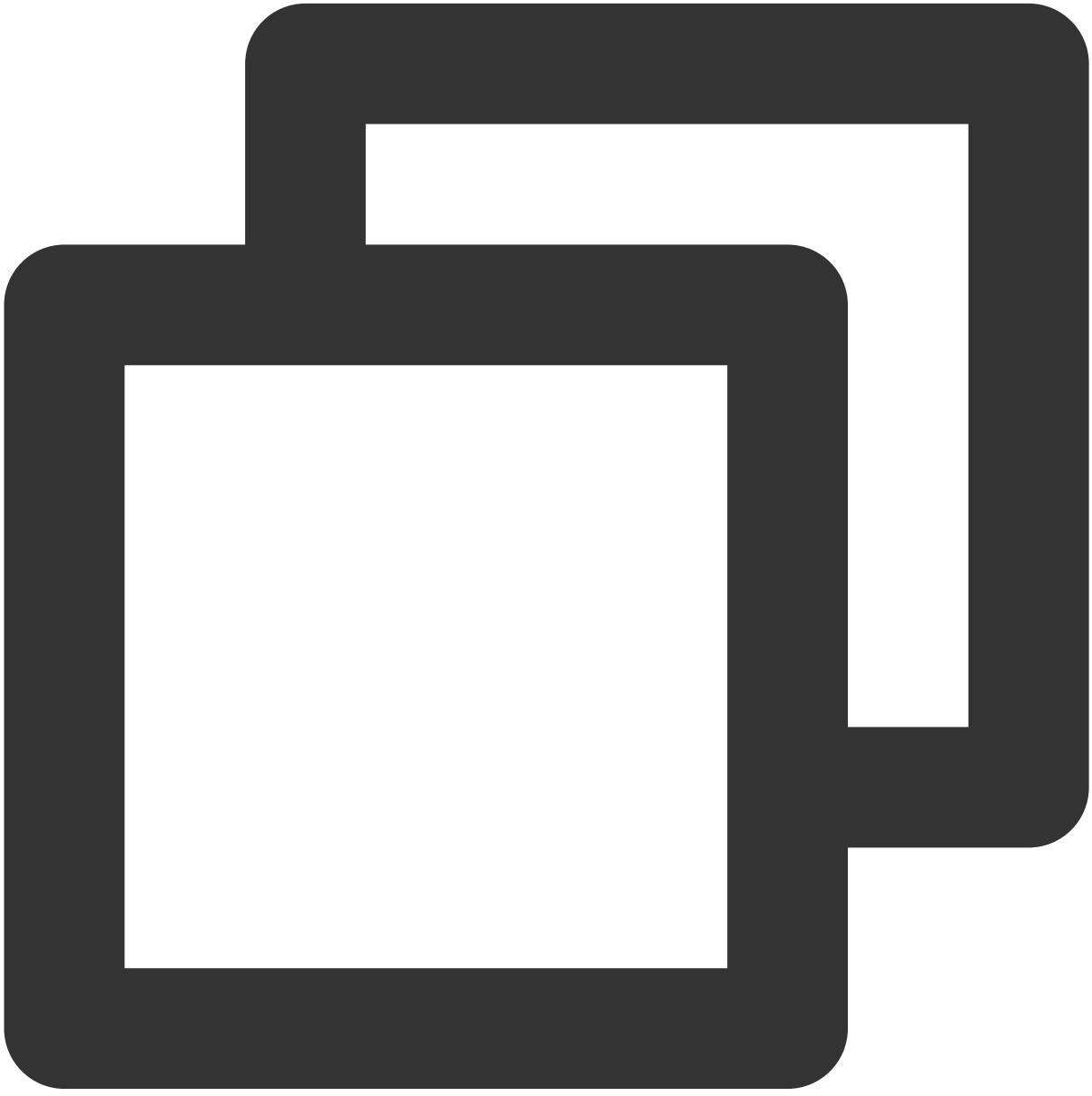
以下 shell 脚本提供了备份整个云数据仓库 PostgreSQL 集群数据的功能，用户可根据需要进行扩充，配合 crontab 完成周期冷备的任务，也可直接下载使用 backup_cdw_v101.sh。

注意：

删除可写外表，不会删除 COS 上对应数据。

备份数据，会导致系统负载升高，建议在系统空闲时运行。

备份时间取决于数据量以及集群规格，简单来说集群节点数越多，备份速度越快。



```
#!/bin/bash

set -e
```

```
# 云数据仓库 PostgreSQL 连接参数，需要填写
PWD='' # 必填
HOST='' # 必填
USER='' # 必填
DEFAULT_DB='postgres'

# 备份参数，需要填写
SECRET_ID='' # 必填
SECRET_KEY='' # 必填
COS_URL='' # 必填  类似 test-1301111111.cos.ap-guangzhou.myqcloud.com
COMPRESS_TYPE='gzip' # COS 上的文件是否采用压缩格式，支持 gzip|none

echo -e "\\n`date "+%Y%m%d %H:%M:%S"` backup task start\\n"

# step1 : 获取数据库列表
db_list=`PGPASSWORD=${PWD} psql -t -h ${HOST} -p 5436 -d ${DEFAULT_DB} -U ${USER} -

# step2 : 遍历需要备份的数据库
for db in $db_list
do
    # template0 template1 gpperfmon 3个db属于模板以及系统库，不需要备份
        if [ "$db" = "template0" -o $db = "template1" -o $db = "gpperfmon" ];then
                continue
        fi

        echo -e "\\n************************************************"
        echo -e "backup database:{$db} start"
        db_start=`date +%s`

    # step3 : 获取当前日期
    # 使用日期作为 COS 存储路径的一部分，以此区分不同日期备份的数据
    cur_date=`date +%Y%m%d`

    # step4 : 获取需要备份的列表
    # 这里去掉了外表，虚拟表，临时表，复制表（暂不支持），对于分区表，只备份子表
    table_list=`PGPASSWORD=${PWD} psql -t -h ${HOST} -p 5436 -d ${db} -U ${USER} -c

    # step5 : 创建cos插件
    PGPASSWORD=${PWD} psql -h ${HOST}  -p 5436 -d ${db} -U ${USER} -c "CREATE EXTEN

    # step6 : 遍历列表，依次备份
    for table in $table_list
    do
        sleep 1
        table_start=`date +%s`
        echo -e "backup ${table} start"
```

```
            # 这里命名必须加在后面，格式是{schema}.{table}
            backup_table="${table}_cdw_backup_cos"

        # step7 : 创建 COS 备份表
        PGPASSWORD=${PWD} psql -h ${HOST}  -p 5436 -d ${db} -U ${USER} -c "CREATE W


        # step8 : 导入原表数据到备份表
        PGPASSWORD=${PWD} psql -h ${HOST}  -p 5436 -d ${db} -U ${USER} -c "INSERT I

        # step9 : 删除备份外表
        # 注：删除外表不会删除COS上对应的数据
        PGPASSWORD=${PWD} psql -h ${HOST}  -p 5436 -d ${db} -U ${USER} -c "DROP EXT

        table_end=`date +%s`
        echo -e "backup ${table} done, cost $[table_end - table_start]s\\n"
    done

        db_end=`date +%s`
        echo -e "backup database:{$db} done, cost $[db_end - db_start]s"
        echo -e "**********************************************\\n"
done
```

统计信息和空间维护

最近更新时间：2024-02-19 15:58:16

背景信息

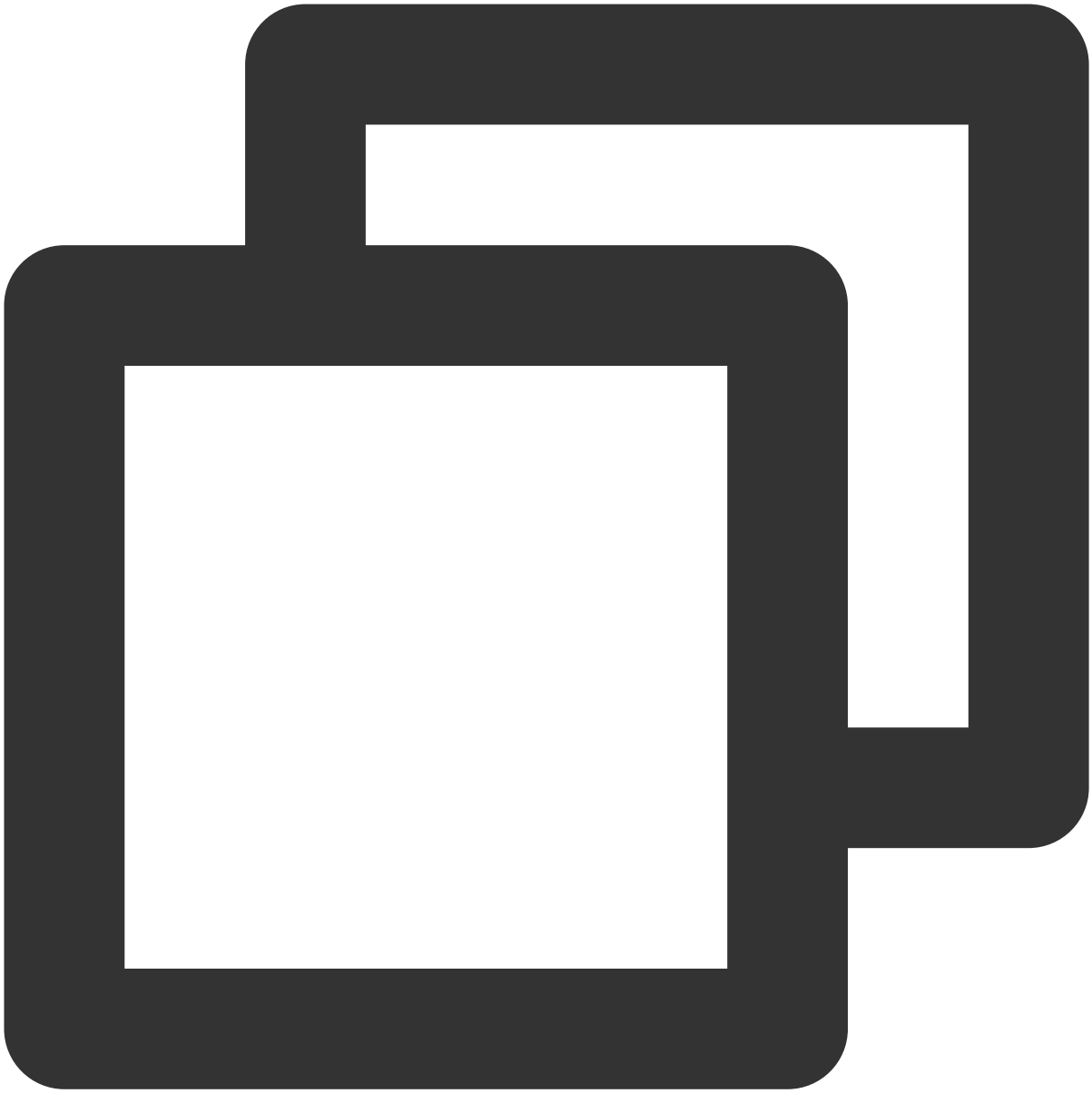
集群的统计信息对于集群的使用非常关键，云数据仓库 PostgreSQL 的查询优化器是根据动态计算出来的 cost（代价）来判断如何进行选择。那如何计算代价呢？这里一般是基于代价模型和统计信息，代价模型是否合理，统计信息是否准确都会影响查询优化的效果。
集群的表的空间利用也会影响查询代价，当表有更新操作（包括 INSERT VALUES、UPDATE、DELETE、ALTER TABLE ADD COLUMN 等）时，会在系统表和被更新的数据表中留存不再被使用的垃圾数据，造成系统性能下降，并占用大量磁盘空间，所以也需要定期监测表的数据膨胀情况。
下文就详细介绍统计信息和数据膨胀的定期监控以及维护。

统计信息收集

ANALYZE 收集一个数据库中表的内容的统计信息，并且将结果存储在 `pg_statistic` 系统目录中。查询计划器会使用这些统计信息来帮助确定查询最有效的执行计划，统计信息包含表的数据量、索引等信息，一个好的查询计划是基于准确的表统计信息。

ANALYZE 说明

ANALYZE 是 Greenplum 提供的收集统计信息的命令。ANALYZE 支持三种粒度：列、表、库，具体如下:



```
CREATE TABLE foo (id int NOT NULL, bar text NOT NULL) DISTRIBUTED BY (id); // 创建测
ANALYZE foo(bar);  // 只搜集 bar 列的统计信息
ANALYZE foo; // 搜集 foo 表的统计信息
ANALYZE; // 搜集当前库所有表的统计信息，需要有权限才行
```

ANALYZE 使用限制

ANALYZE 会给目标表加 SHARE UPDATE EXCLUSIVE 锁，即会与 DDL 语句冲突。

ANALYZE 速度

ANALYZE 是一种采样统计算法，通常不会扫描表中所有的数据，但是对于大表，也仍会消耗一定的时间和计算资源。

ANALYZE 使用时机

根据上文所述，ANALYZE 会加锁并且也会消耗系统资源，因此运行命令需要选择合适的时机尽可能少的运行。以下4种情况发生后建议运行 ANALYZE。
批量加载数据后，例如新表创建导入数据后。
创建索引后。
INSERT、UPDATE、DELETE 大量数据后。
VACUUM FULL 执行清理后。

ANALYZE 分区表

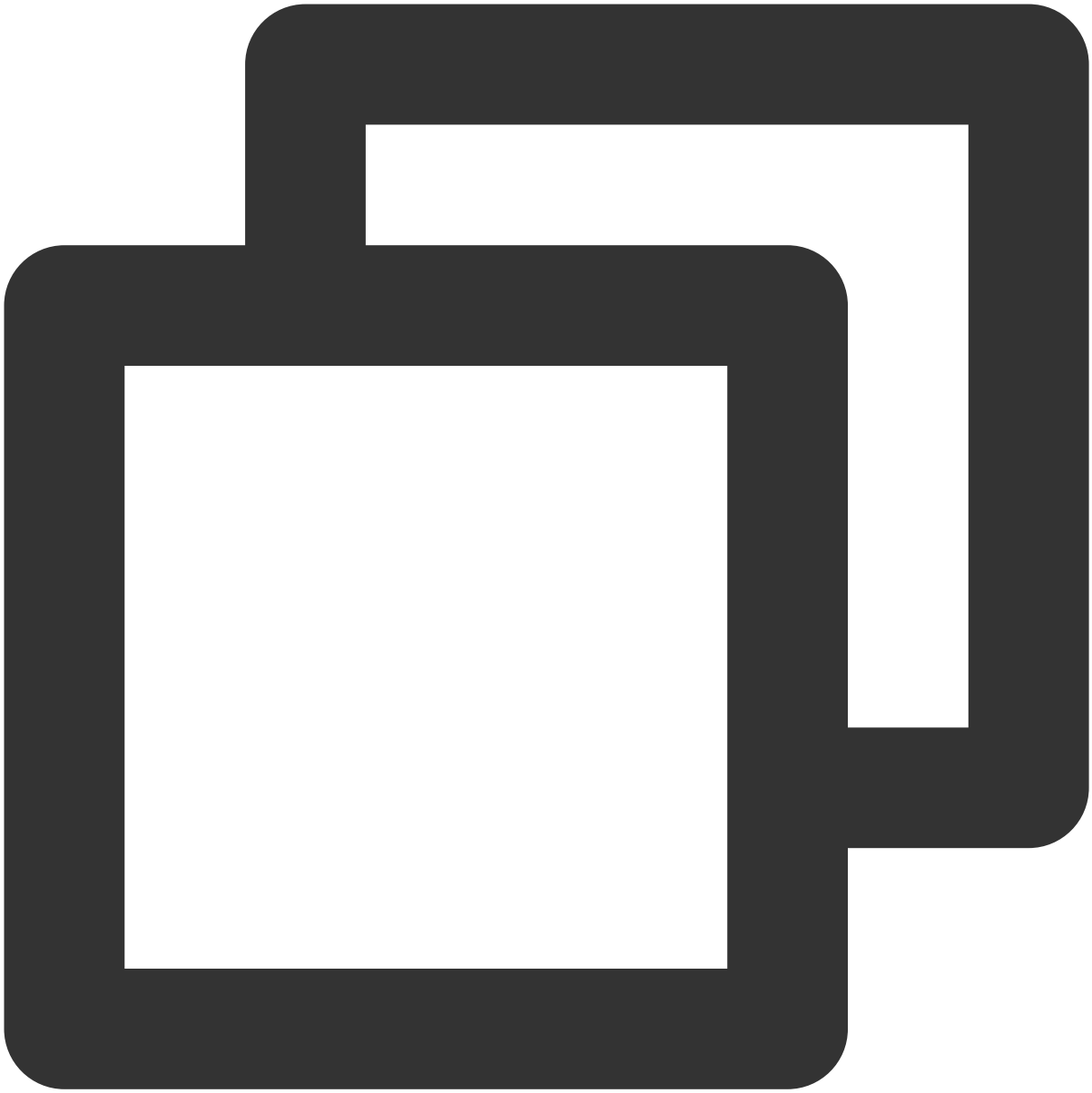
只要保持默认值，不去修改系统参数 `optimizer_analyze_root_partition`，那么对于分区表的操作并没有什么不同，直接在 root 表上进行 ANALYZE 即可，系统会自动把所有叶子节点的分区表的统计信息都收集起来。
如果分区表的数目很多，那在 root 表上进行 ANALYZE 可能会非常耗时。分区表通常是带有时间维度的，历史的分区表并不会修改，因此建议单独 ANALYZE 数据会发生变化的分区。

数据膨胀

Greenplum 数据库的堆表使用 PostgreSQL 的多版本并发控制（MVCC）存储实现。数据库会逻辑删除被删除或更新的行，但是该行的一个不可见映像将保留在表中，随着操作的进行，表的不可见数据会越来越多，在显著影响存储空间时会导致表操作性能严重下滑，并且膨胀的数据会占用大量的数据空间，因此需要定期对数据库的膨胀进行处理。

数据膨胀的监控

gp_toolkit 模式提供了一个 gp_bloat_diag 视图，它通过预计页数和实际页数的比率来确定表膨胀。要使用这个视图，需确定为数据库中所有的表都收集了最新的统计信息。然后运行下面的 SQL：



```
gpadmin=# SELECT * FROM gp_toolkit.gp_bloat_diag;
 bdirelid | bdinspname | bdirelname | bdirelpages | bdiexppages |                bd
----------+------------+------------+-------------+-------------+-------------------
    21488 | public     | t1         |          97 |           1 | significant amoun
(1 row)
```

其中 bdirelpage 是 t1 表目前实际页面，bdiexppages 为 t1 表期望页面，当膨胀率超过4倍时就会被统计在该表中，没有出现在表中也有可能会有轻度膨胀，也可以对比不同时间的表的空间大小判断是否存在数据膨胀。

数据表膨胀的清理

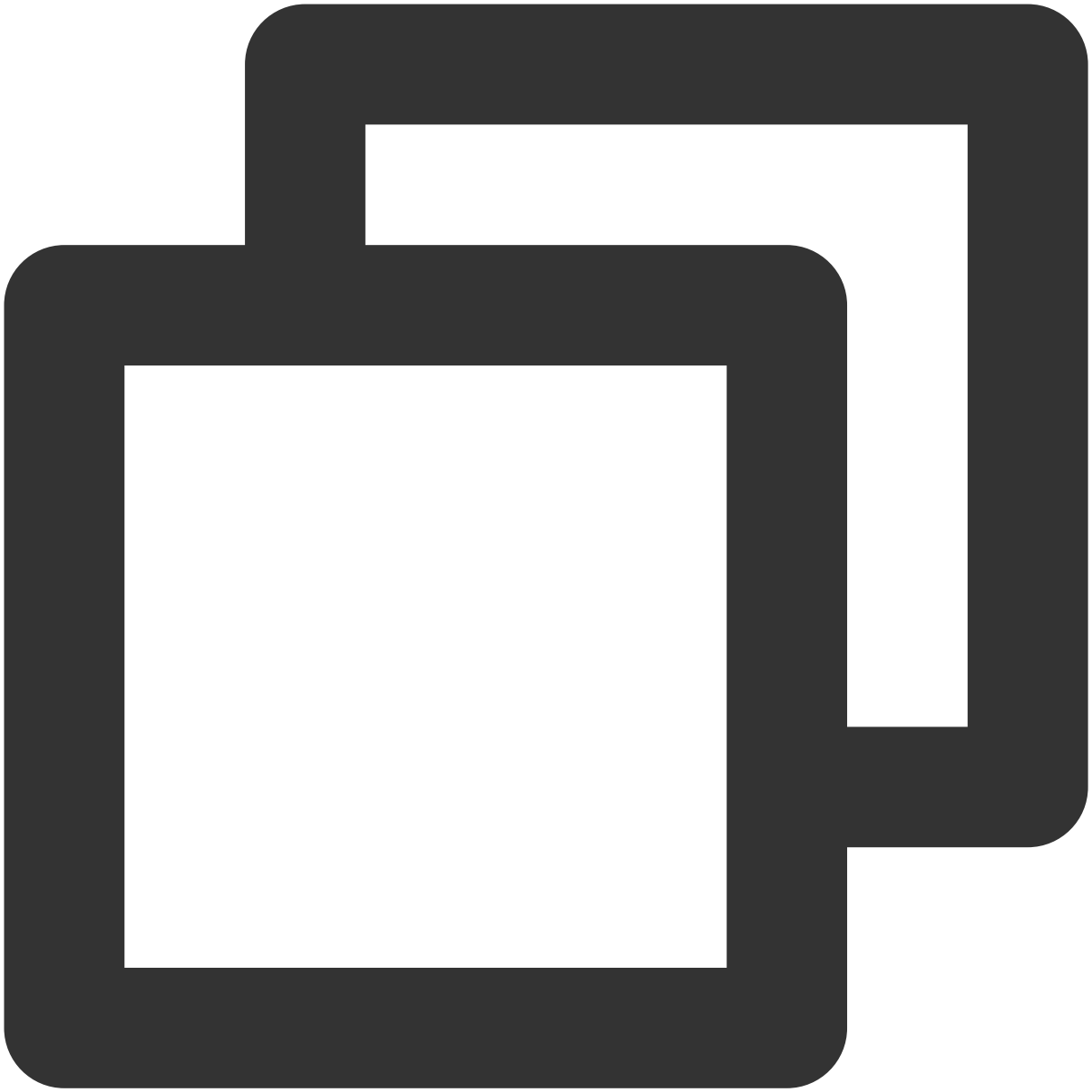
`VACUUM <tablename>` 命令会把过期行加入到共享的空闲空间映射中，这样这些空间能被重用。当在被频繁更新的表上定期运行 VACUUM 时，过期行所占用的空间可以被迅速地重用，从而缓解表膨胀。需根据表更新，删除速度来决定 VACUUM 执行的周期。

注意：

VACUUM 和 ANALYZE 一样会持有共享更新锁（SHARE UPDATE EXCLUSIVE），该命令可能和 DDL 操作互锁。当表已经出现明显膨胀时，VACUUM 只能起到延缓继续增长的作用，并不能够立即回收空间，这时需要使用 VACUUM FULL 命令，该命令能够立即回收所有膨胀空间，不过 VACUUM FULL 操作会对操作表加上访问独占（ACCESS EXCLUSIVE），期间该表上其余所有 DDL 和 DML 都将被锁住，并且针对大型表可能会需要很长时间执行，因此需要谨慎操作。

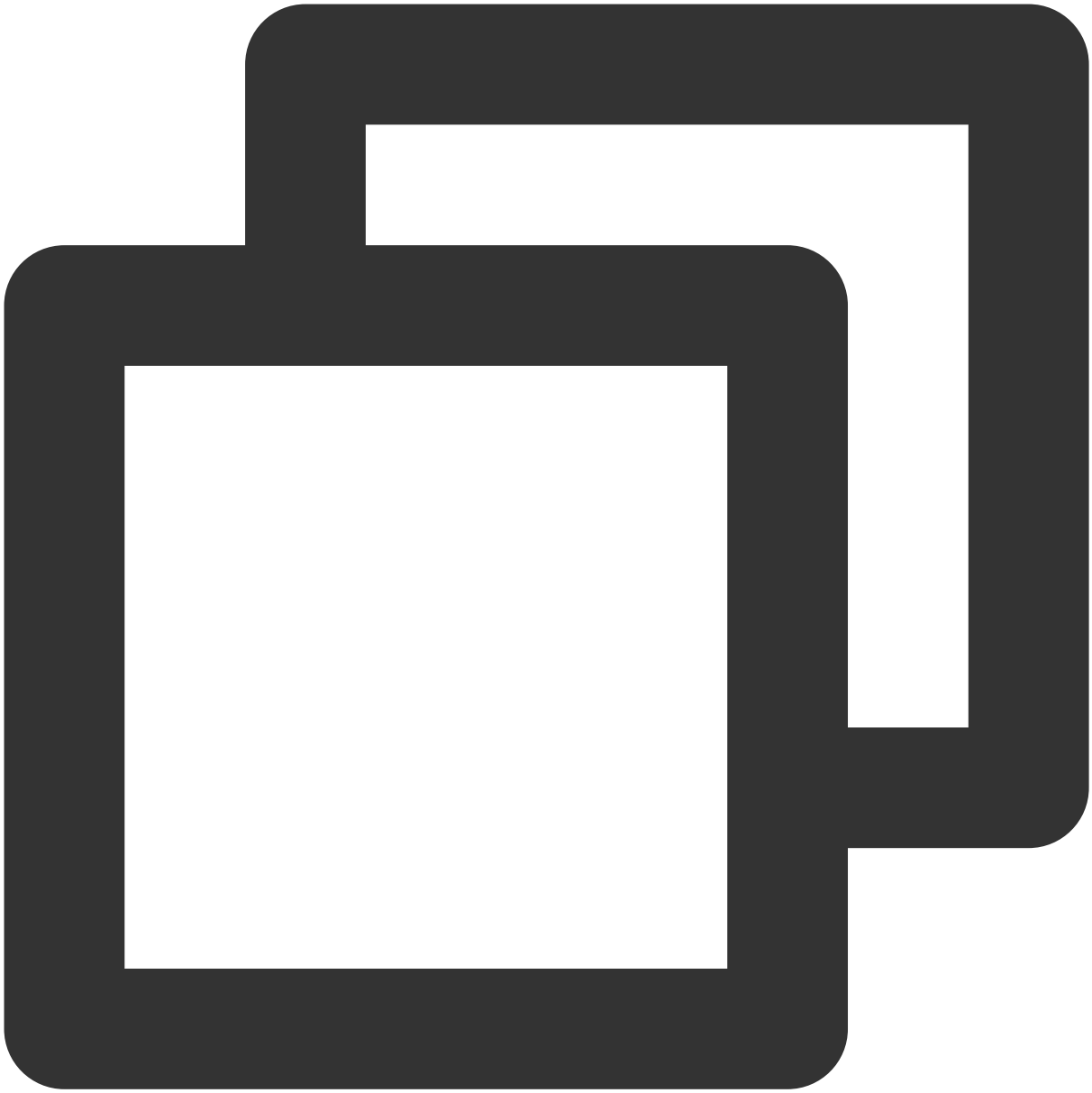
还有一种处理膨胀的方式就是进行表数据重分布：

1. 记录表的分布键。
2. 把该表的分布策略改为随机分布。



```
ALTER TABLE <tablename> SET WITH (REORGANIZE=false)
             DISTRIBUTED randomly;
```

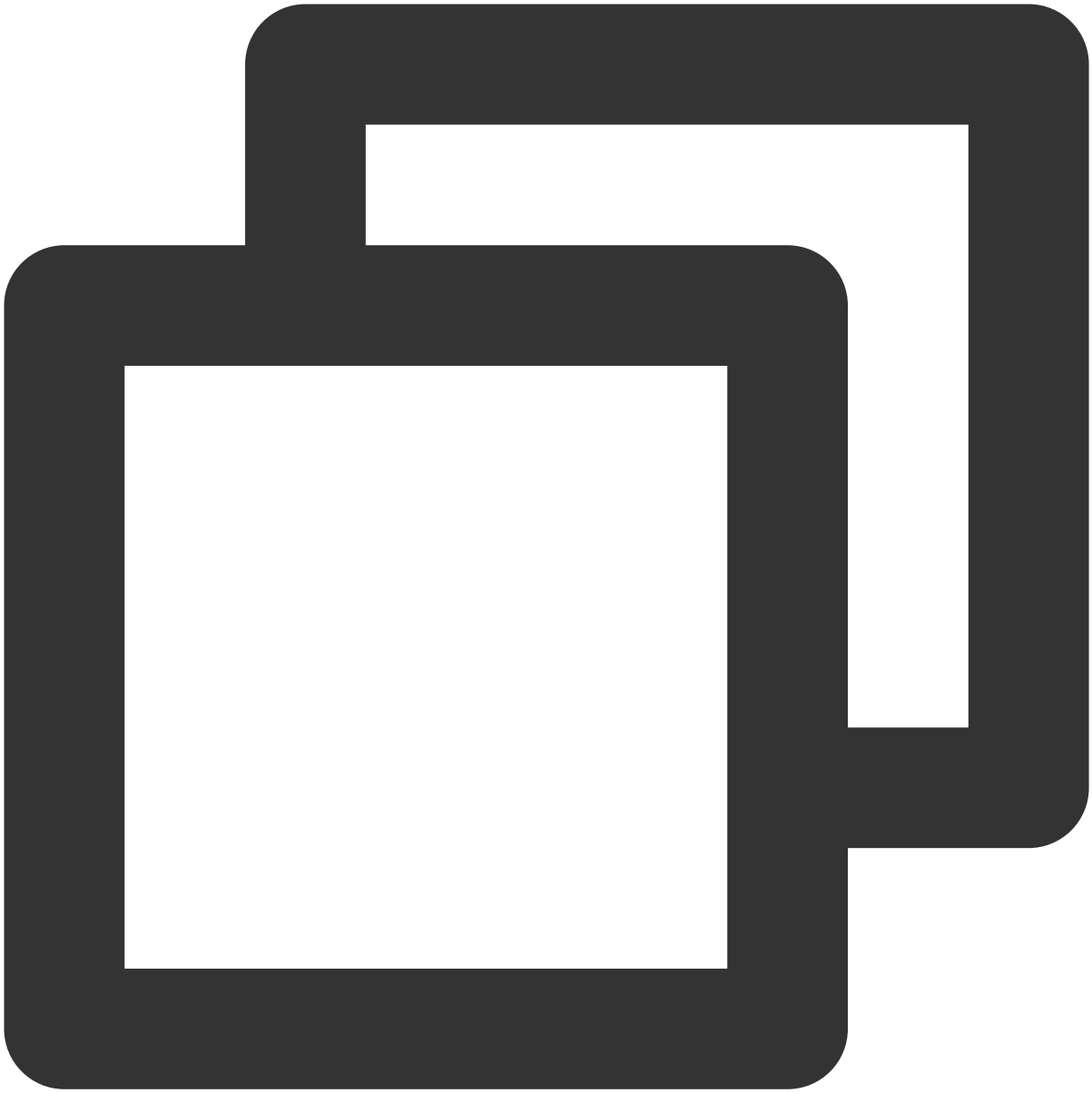
3. 把分布策略改回初始设置。



```
ALTER TABLE <table_name> SET WITH (REORGANIZE=true)
DISTRIBUTED BY (<original distribution columns>);
```

索引膨胀的处理

VACUUM FULL 命令只会从表中恢复空间。要从索引中恢复空间，需要使用 REINDEX 命令重建它们。



```
REINDEX TABLE <table_name>     --- 重建一个表上所有索引
REINDEX INDEX <index_name> --- 重建一个指定索引
```

注意：

REINDEX 和 VACUUM FULL 一样会加上访问独占（ACCESS EXCLUSIVE），因此需要谨慎操作。

定期进行集群维护

在使用集群的过程中，需要定期进行数据膨胀消除和统计信息维护，这样才能够让集群的使用性能达到最优。

不锁表清理

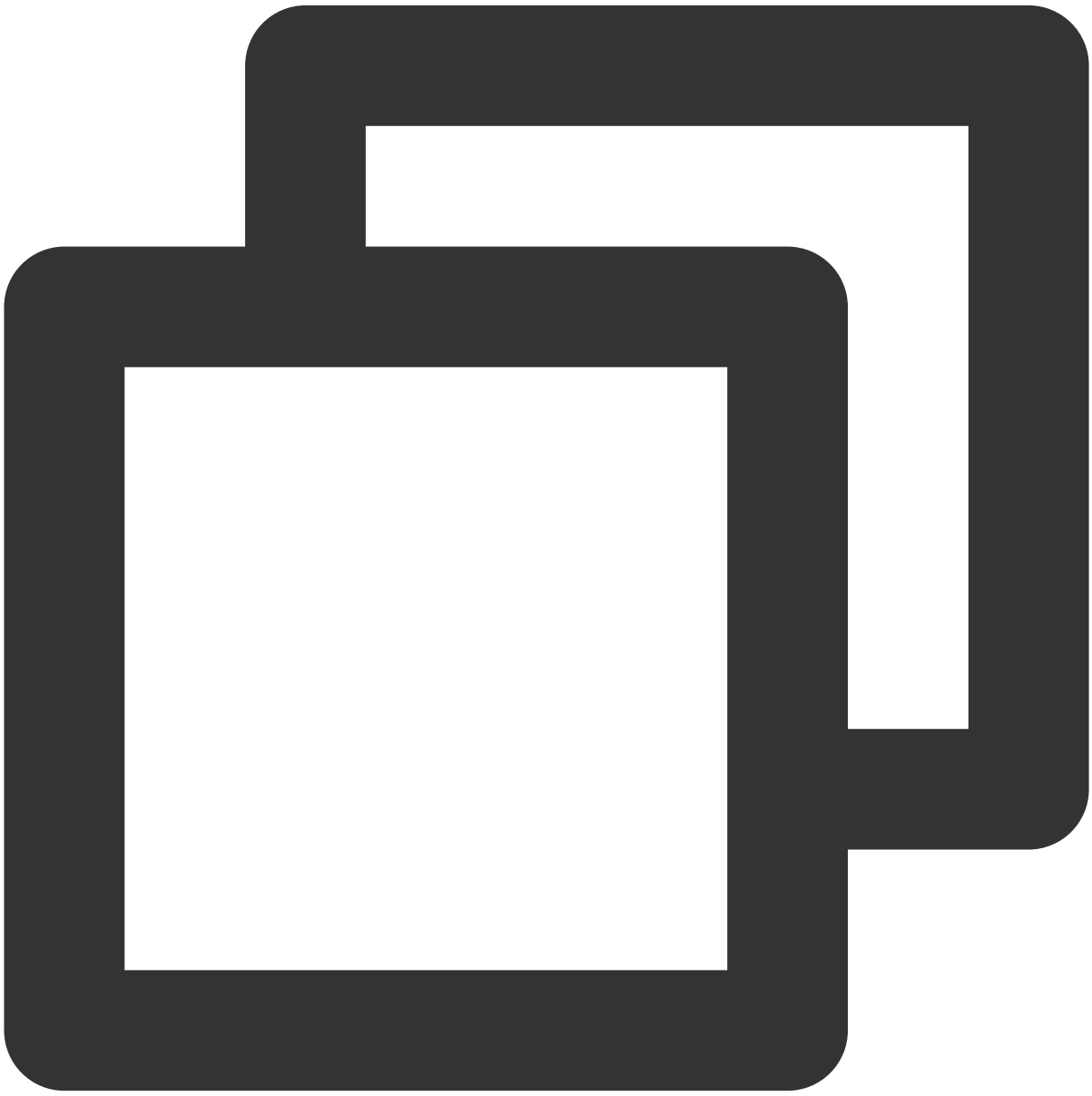
上文提到 `VACUUM <tablename>` 可以在不锁表的情况下标记可回收的空间，减缓数据膨胀，不锁表清理不会影响数据表读写，只是无法使用 DDL，并且会占用一定 CPU 内存资源。

建议频率：

大批量实时更新数据，每日多条数据发生变化，建议每天执行一次，每周至少两次。

正常情况一周执行一次，或者不频繁更新数据可以每月执行一次。

使用下述脚本可以通过 cron 定时任务清理一张表。



```
#!/bin/bash
export PGHOST=xxx.xxx.x.x
export PGPORT=5436
export PGUSER=test
export PGPASSWORD=test
db="test"
psql -d $db -e -c "VACUUM test_table;"
```

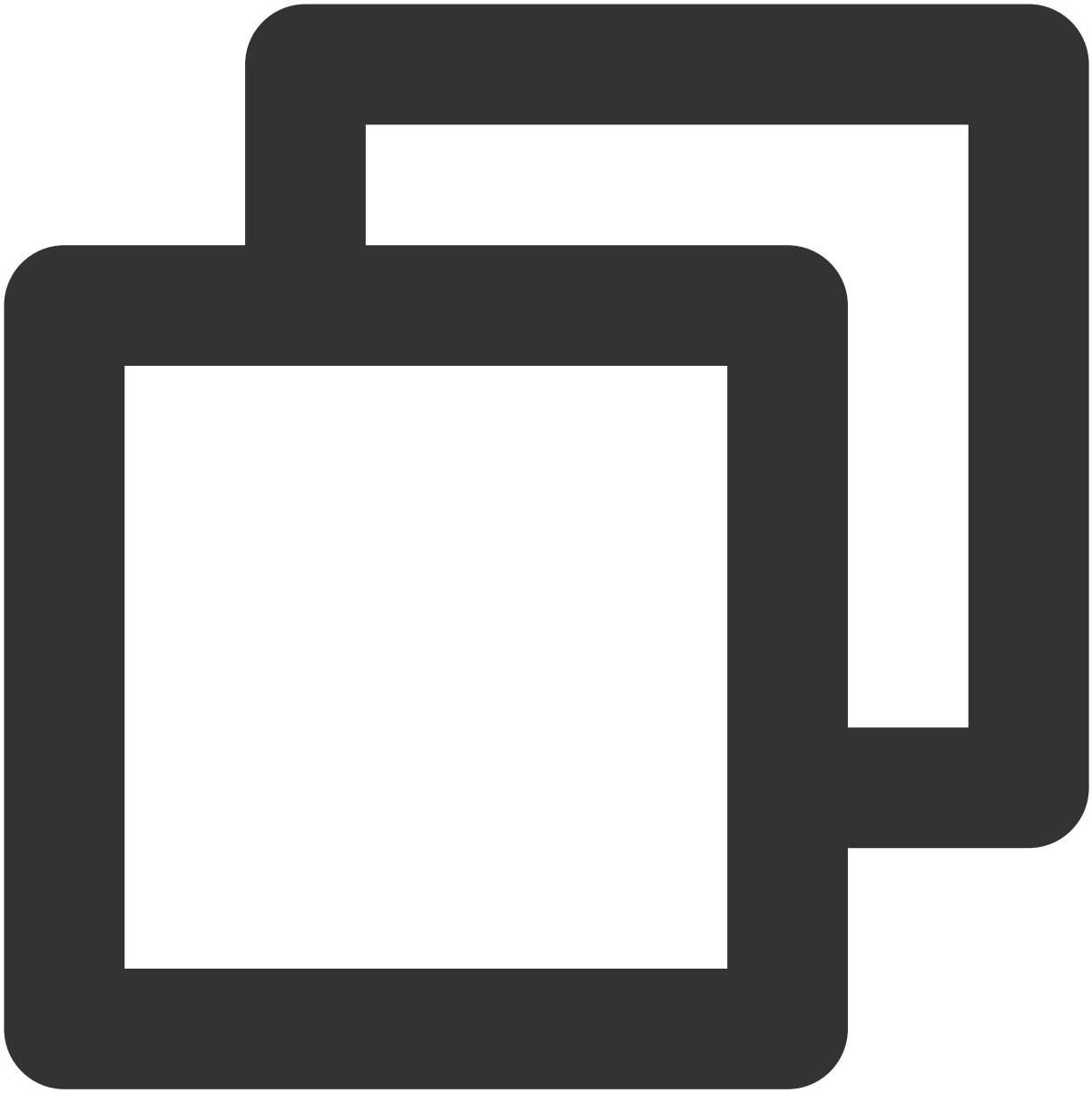
锁表全量清理

如果一张表会经常更新删除，那么建议规划一个业务暂停窗口，执行 VACUUM FULL 以及 REINDEX 来回收表的所有膨胀空间，锁表清理会对执行的表加上访问排它锁，期间被清理的表将无法进行任何操作。

1. 执行 `VACUUM FULL <tablename>`
2. 执行 `REINDEX TABLE <tablename>` （没有索引的表可以不做）
3. 执行 `ANALYZE <tablename>`

建议频率：建议每周执行一次。如果每天会更新几乎所有数据，需要每天做一次。

使用下述脚本可以实现集群表定期清理，建议在周末凌晨非业务期间进行。



```
#!/bin/bash
export PGHOST=xxx.xxx.x.x
export PGPORT=5436
export PGUSER=test
export PGPASSWORD=test
db="test"
psql -d $db -e -c "VACUUM FULL test_table;"
psql -d $db -e -c "REINDEX TABLE test_table;"
psql -d $db -e -c "ANALYZE test_table;"
```

版权所有：腾讯云计算（北京）有限责任公司