云数据库 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】

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模式的承诺或保证。

文档目录

版权所有：腾讯云计算（北京）有限责任公司

最佳实践
跨库访问

最近更新时间：2024-03-20 11:01:39

云数据库 PostgreSQL 提供用于访问外部数据源的一类插件，外部数据源包括本实例其他库中数据或者其他实例的数据。跨库访问插件包含同构的跨库访问插件 dblink、postgresql_fdw，异构的跨库访问插件 mysql_fdw、cos_fdw。
跨库访问使用步骤如下：
1. 使用 “CREATE EXTENSION” 语句安装插件。
2. 为每个需要连接的远程数据库创建一个外部服务器对象并创建链接映射。
3. 使用对应的命令访问外部表以获取数据。
由于跨库访问插件可以直接跨实例访问或在同实例中进行跨 database 访问。云数据库 PostgreSQL 对创建外部服务器对象时进行了权限控制优化，根据目标实例所在环境进行分类管理。在开源版本基础上增加了额外辅助参数，来验证用户身份和调整网络策略。具体请参考下文 插件辅助参数。
说明：
dblink 插件当前只有大版本为10的云数据库 PostgreSQL 内核支持，请知悉。

插件辅助参数

host
跨实例访问时候为必须项。目标实例的 IP 地址。
port
跨实例访问时候为必须项。目标实例的 port。
instanceid
实例 ID
在云数据库 PostgreSQL 间跨实例访问时使用，当跨实例访问时为必选项。格式类似 postgres-xxxxxx、pgro-xxxxxx，可在 控制台 查看。
如果目标实例在腾讯云 CVM 上，则为 CVM 机器的实例 ID，格式类似 ins-xxxxx。
dbname
database 名，填写需要访问的远端 PostgreSQL 服务的 database 名字。若不跨实例访问，仅在同实例中进行跨库访问，则只需要配置此参数即可，其他参数都可为空。
access_type
非必须项。目标实例所属类型：
目标实例为 TencentDB 实例，包括云数据库 PostgreSQL、云数据库 MySQL 等，如果不显示指定，则默认该项。
目标实例在腾讯云 CVM 机器上。
目标实例为腾讯云外网自建。

版权所有：腾讯云计算（北京）有限责任公司

目标实例为云 VPN 接入的实例。
目标实例为自建 VPN 接入的实例。
目标实例为专线接入的实例。

uin

非必须项。实例所属的账号 ID，通过该信息鉴定用户权限，可参见 查询 uin。

own_uin

非必须项。实例所属的主账号 ID，同样需要该信息鉴定用户权限。

vpcid

非必须项。私有网络 ID，目标实例如果在腾讯云 CVM 的 VPC 网络中，则需要提供该参数，可在 VPC 控制台 中查看。

subnetid

非必须项。私有网络子网 ID，目标实例如果在腾讯云 CVM 的 VPC 网络中，则需要提供该参数，可在 VPC 控制台 的子网中查看。

dcgid

非必须项。专线 ID，目标实例如果需要通过专线网络连接，则需要提供该参数值。

vpngwid

非必须项。VPN 网关 ID，目标实例如果需要通过 VPN 进行网络连接，则需要提供该参数值。

region

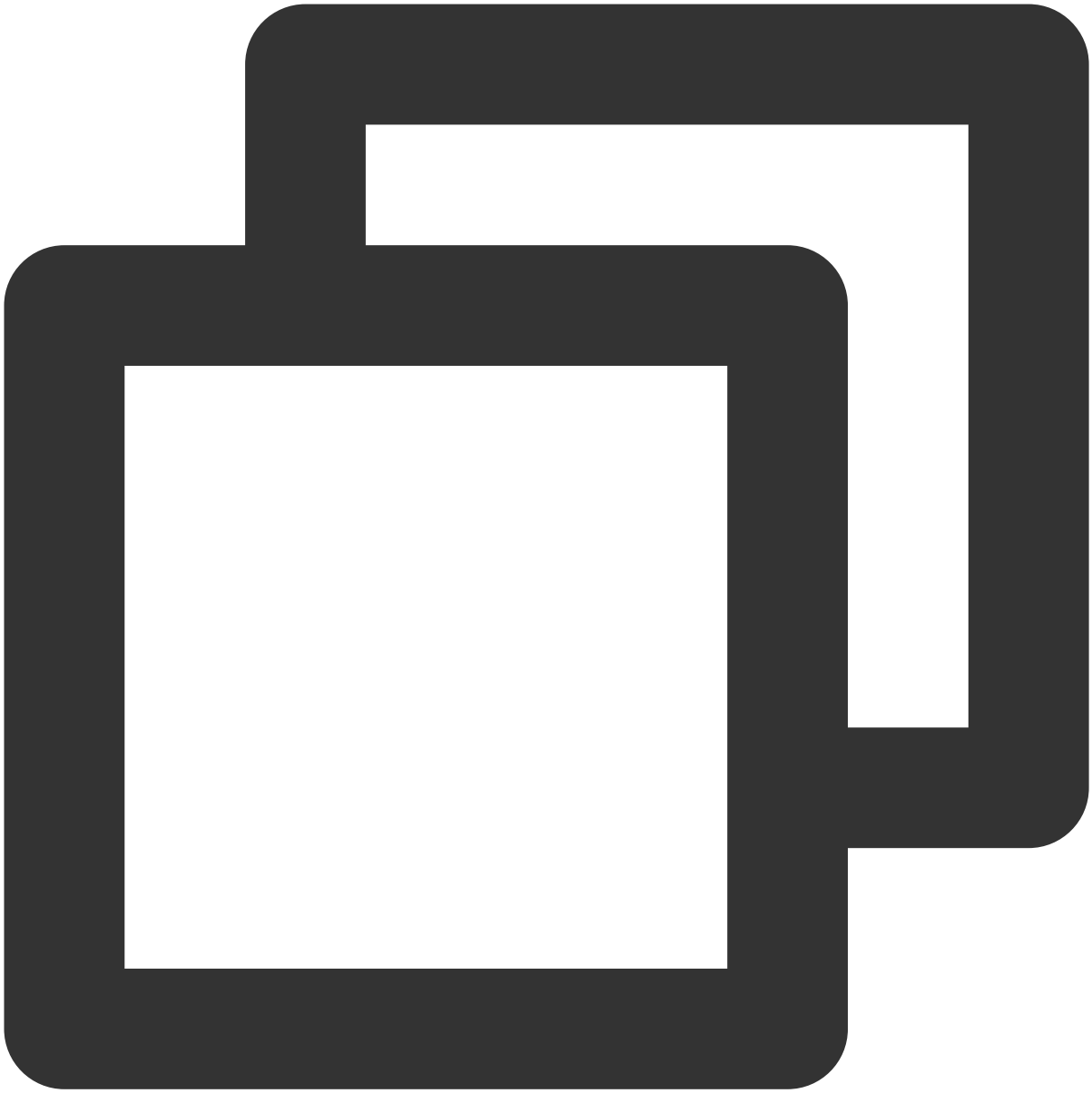
非必须项。目标实例所在地域，如 “ap-guangzhou” 表示广州。如果需要跨地域访问数据，则需要提供该参数值。

使用 postgres_fdw 示例

使用 postgres_fdw 插件可以访问本实例其他库或者其他 postgres 实例的数据。

步骤1：前置条件

1. 在本实例中创建测试数据。

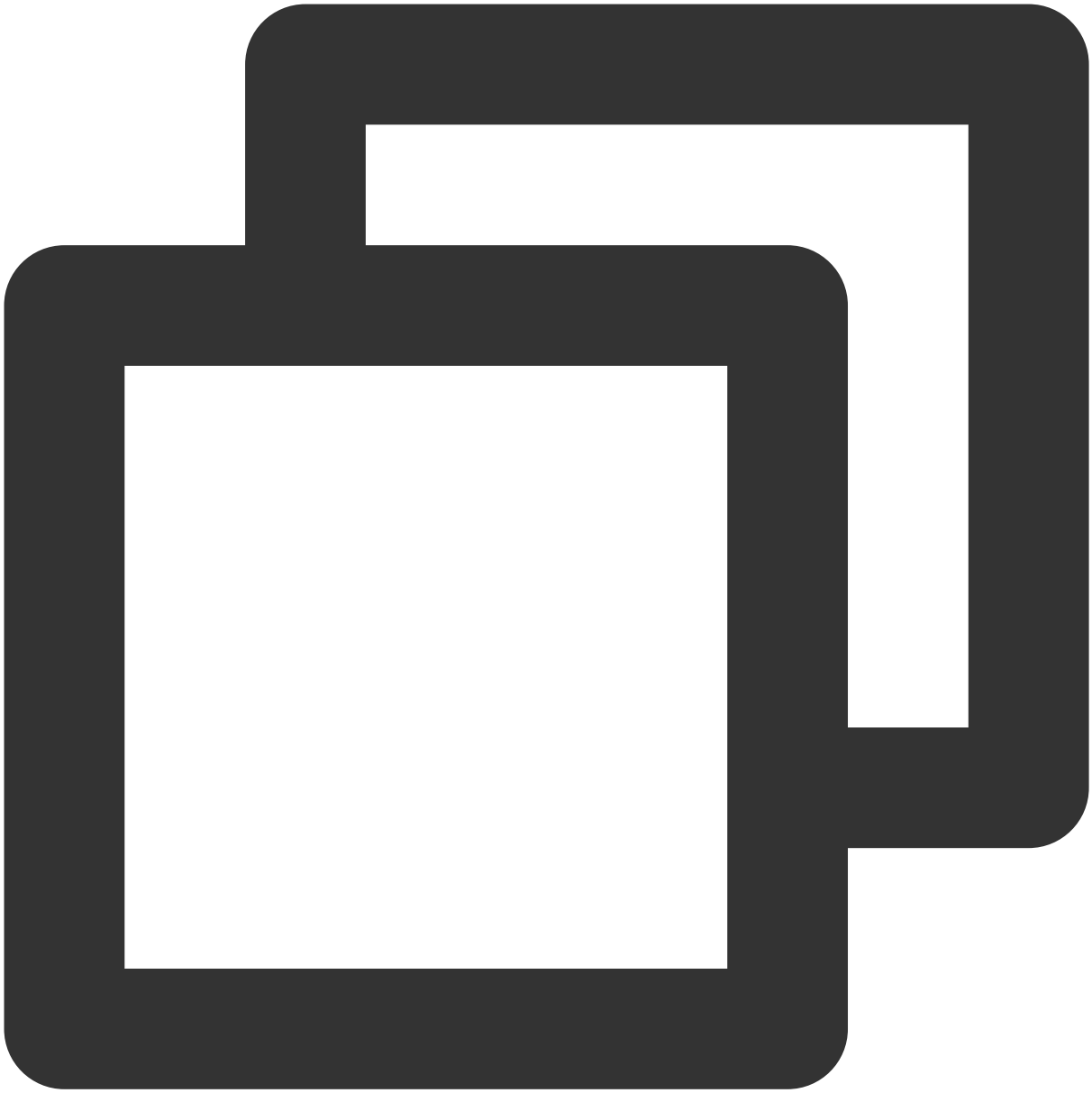


```
postgres=>create role user1 with LOGIN  CREATEDB PASSWORD 'password1';
postgres=>create database testdb1;
CREATE DATABASE
```

注意：

若创建插件报错，请 提交工单 联系腾讯云售后协助处理。

2. 在目标实例中创建测试数据。

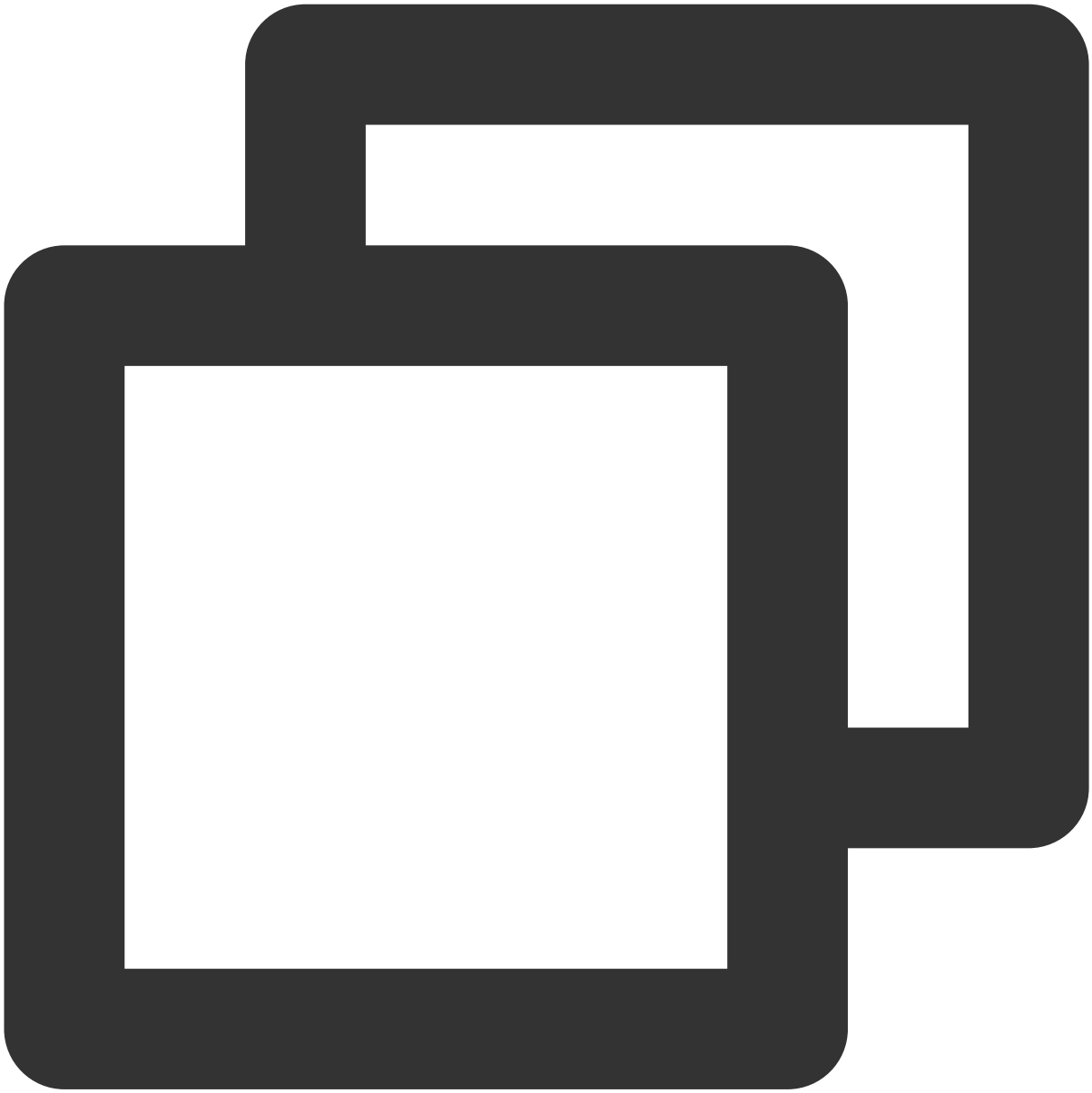


```
postgres=>create role user2 with LOGIN  CREATEDB PASSWORD 'password2';
postgres=> create database testdb2;
CREATE DATABASE
postgres=> \\c testdb2 user2
You are now connected to database "testdb2" as user "user2".
testdb2=> create table test_table2(id integer);
CREATE TABLE
testdb2=> insert into test_table2 values (1);
INSERT 0 1
```

步骤2：创建 postgres_fdw 插件

说明：

若创建插件时，提示插件不存在或权限不足，请 提交工单 处理。



```
#创建
postgres=> \\c testdb1
You are now connected to database "testdb1" as user "user1".
testdb1=> create extension postgres_fdw;
CREATE EXTENSION
#查看
testdb1=> \\dx
```

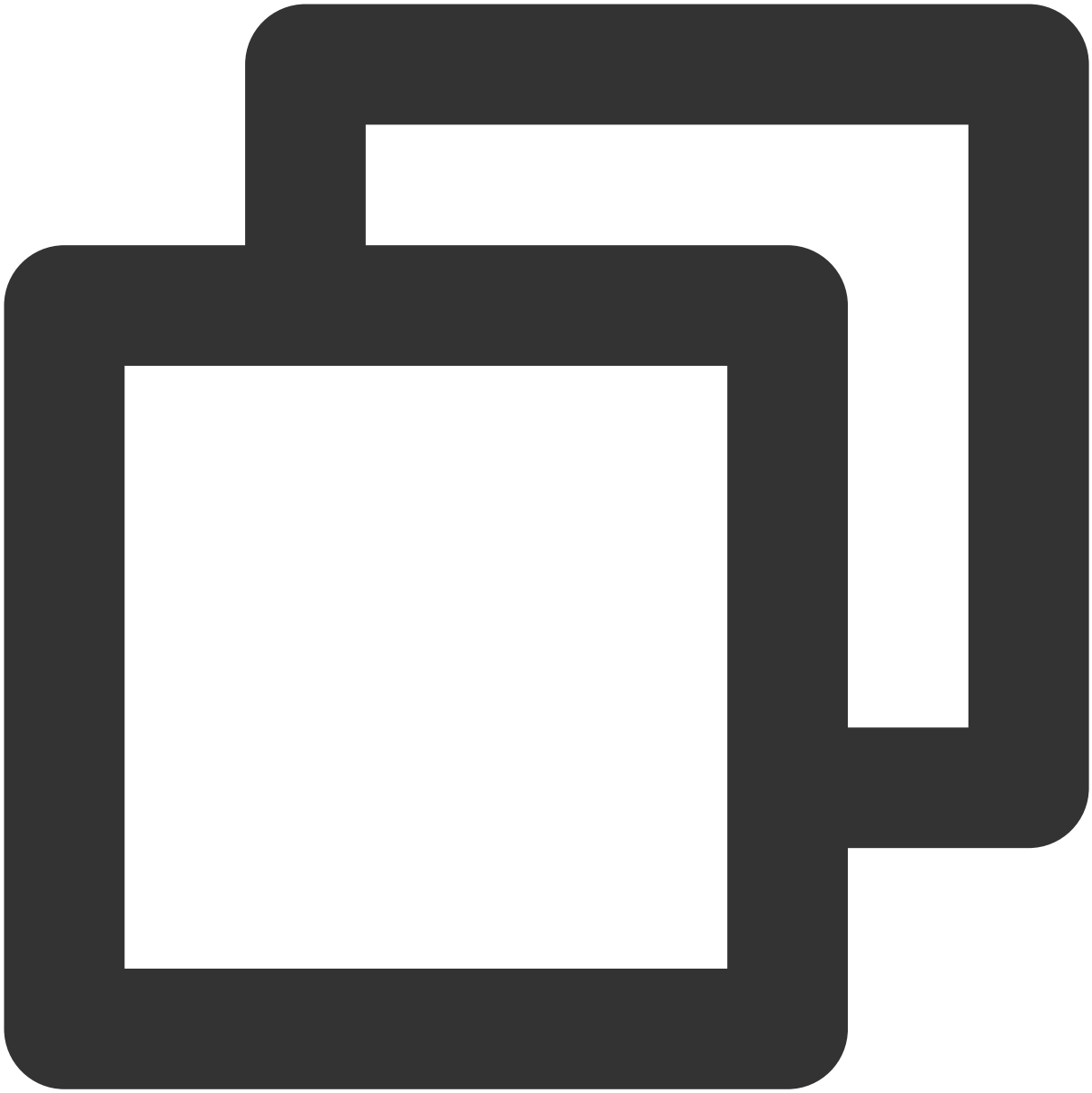
```
                              List of installed extensions
      Name      | Version |   Schema   |                     Description
----------------+---------+------------+------------------------------------------------------
  plpgsql       | 1.0     | pg_catalog | PL/pgSQL procedural language
  postgres_fdw | 1.0     | public     | foreign-data wrapper for remote PostgreSQL
(2 rows)
```

步骤3：创建 SERVER

注意：

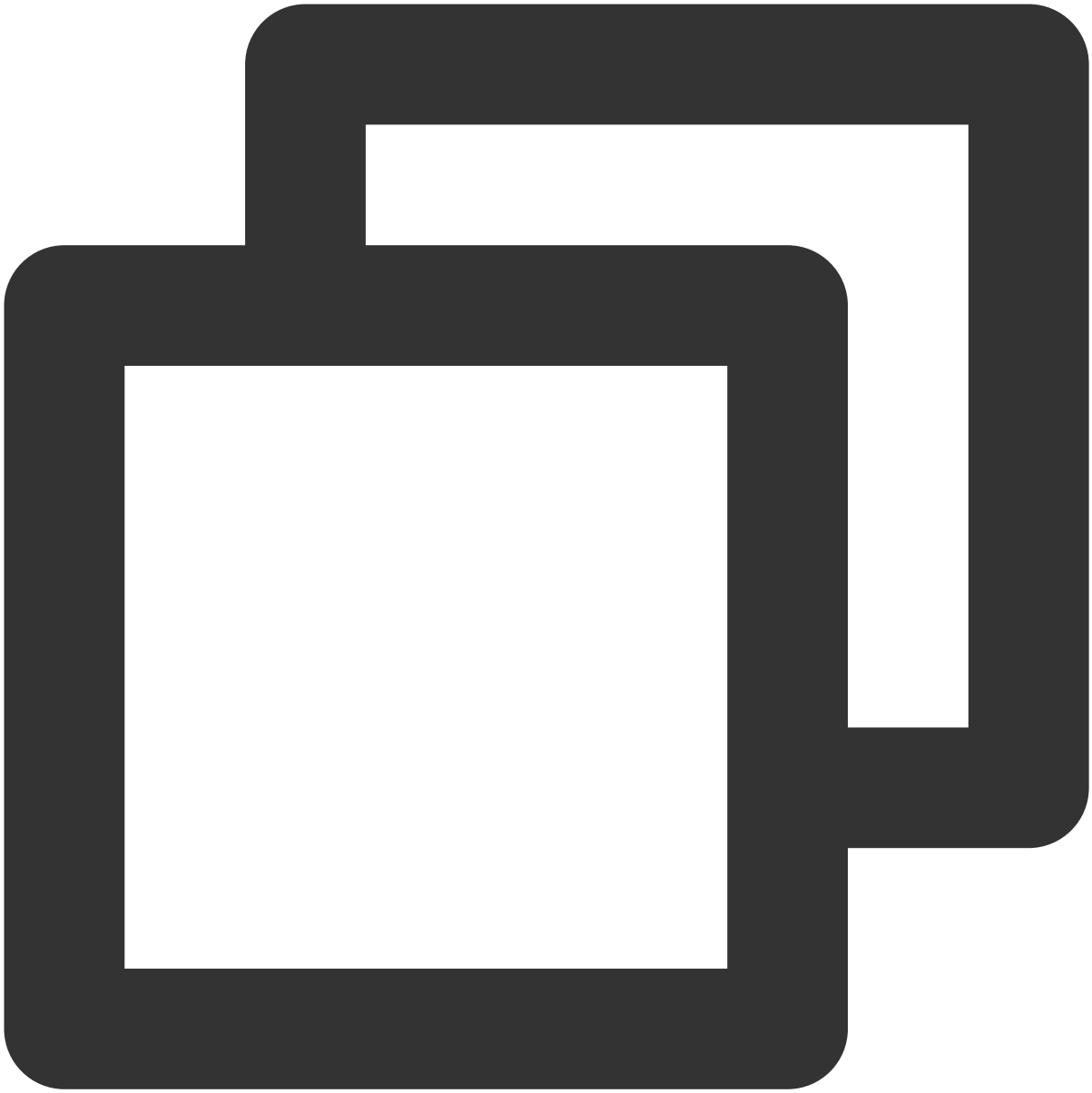
仅 v10.17_r1.2、v11.12_r1.2、v12.7_r1.2、v13.3_r1.2、v14.2_r1.0 及之后的内核版本支持跨实例访问。

跨实例访问。



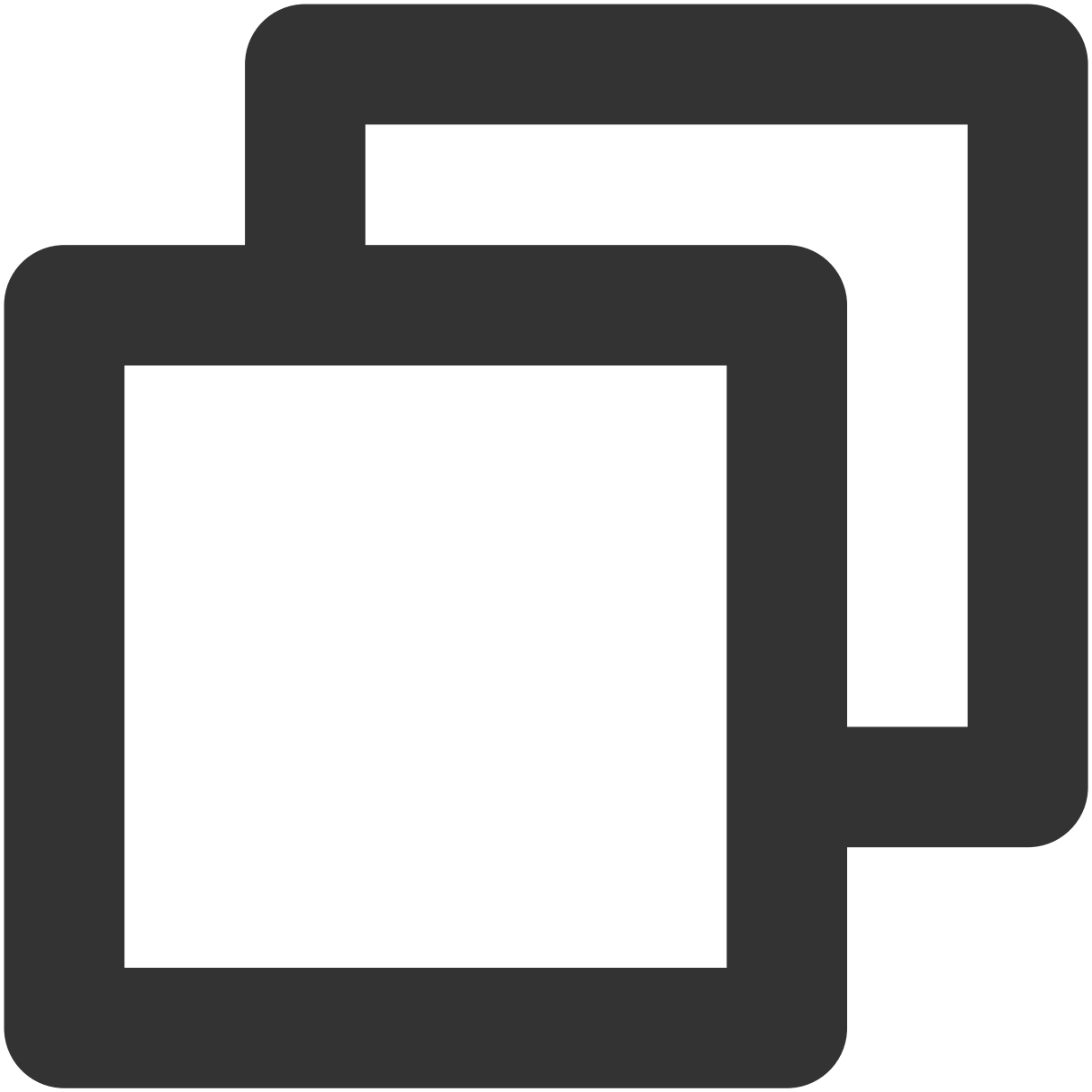
```
#从本实例的 testdb1 访问目标实例 testdb2 的数据
testdb1=>create server srv_test1 foreign data wrapper postgres_fdw options (host 'x
CREATE SERVER
```

不跨实例，仅跨 database 访问，仅需要填写 dbname 参数即可。



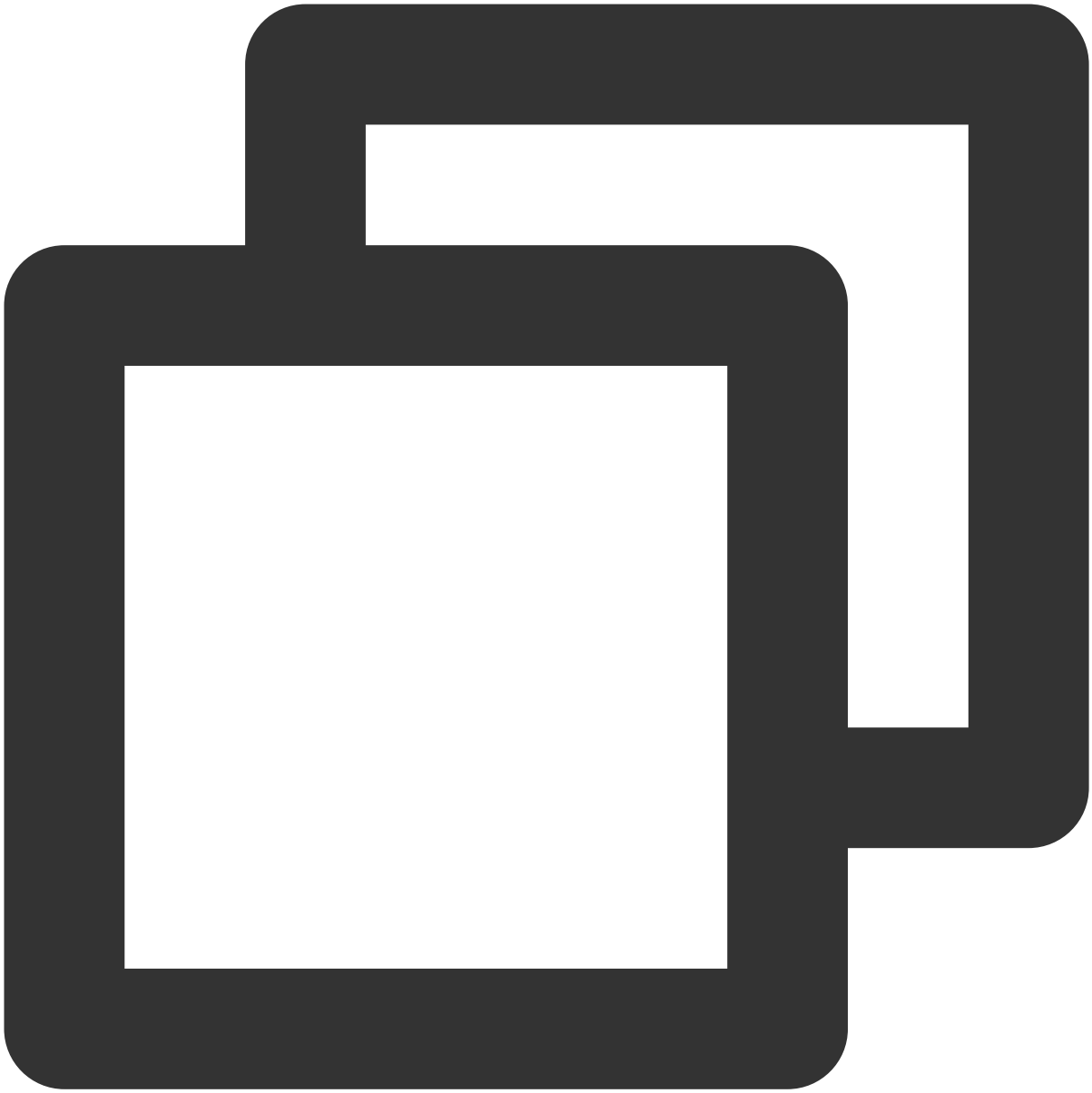
```
#从本实例的 testdb1 访问本实例 testdb2 的数据
create server srv_test1 foreign data wrapper postgres_fdw options (dbname 'testdb2'
```

目标实例在腾讯云 CVM 上，且网络类型为基础网络。



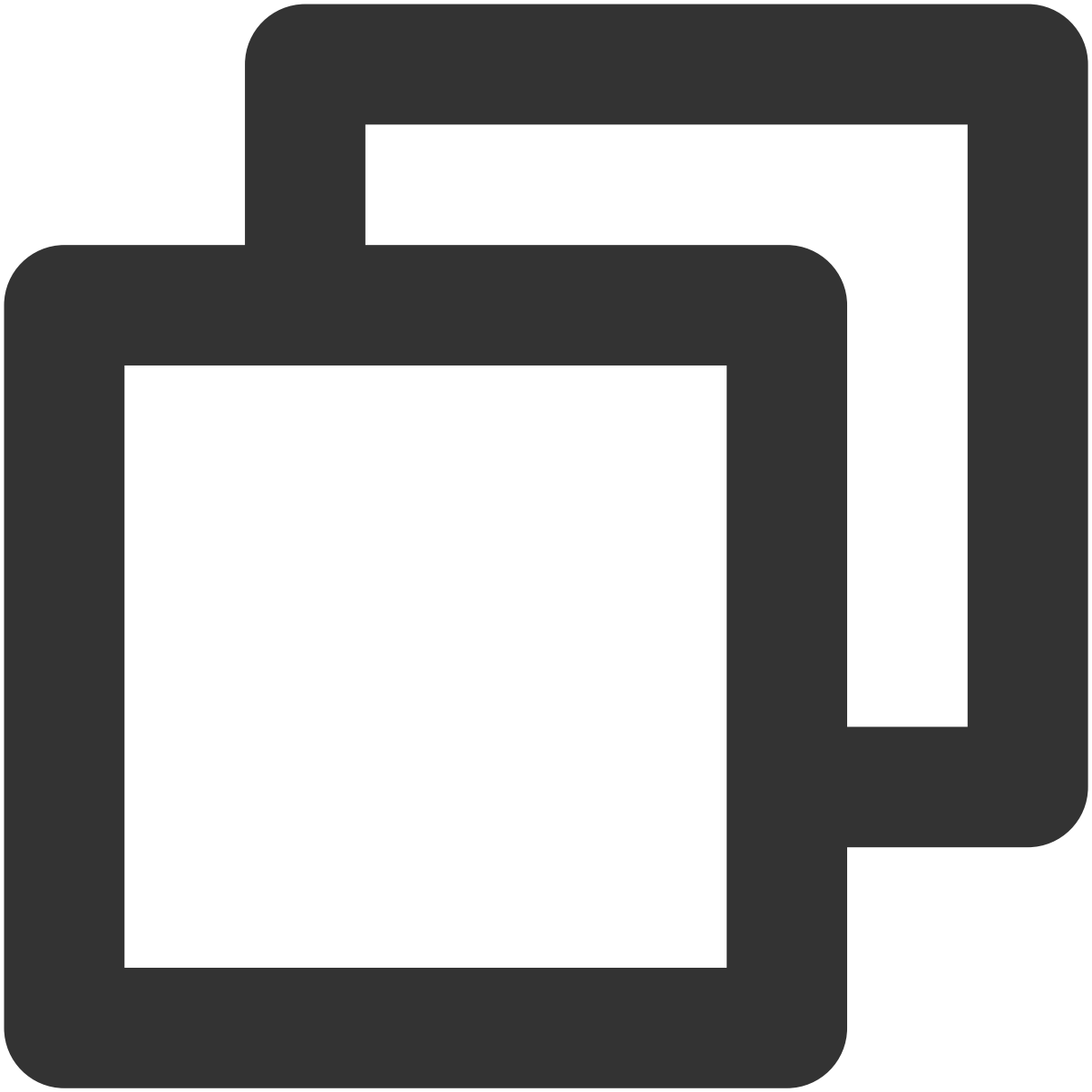
```
testdb1=>create server srv_test foreign data wrapper postgres_fdw options (host '
CREATE SERVER
```

目标实例在腾讯云 CVM 上，且网络类型为私有网络。



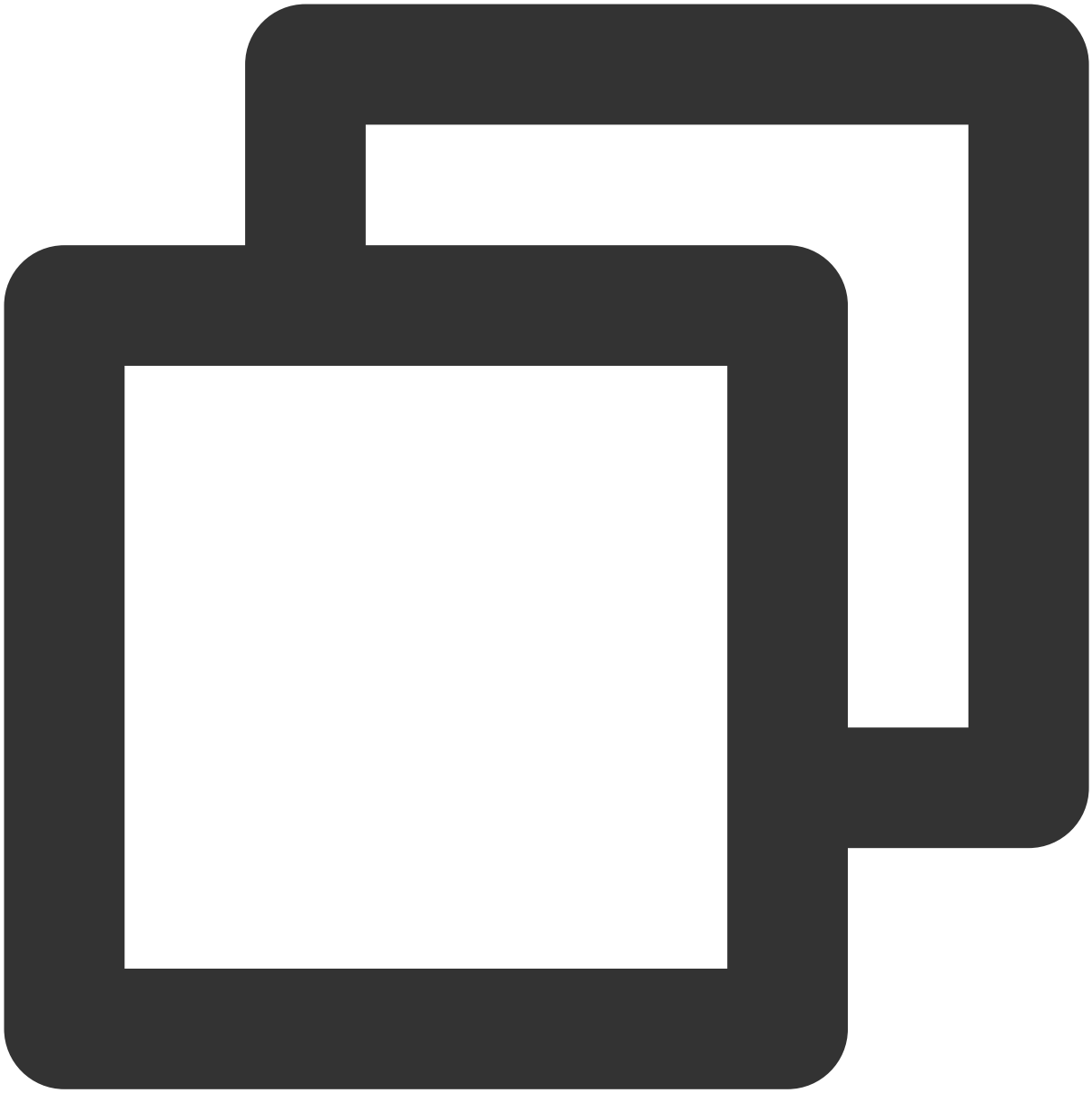
```
testdb1=>create server srv_test1 foreign data wrapper postgres_fdw options (host
CREATE SERVER
```

目标实例在腾讯云外网自建。



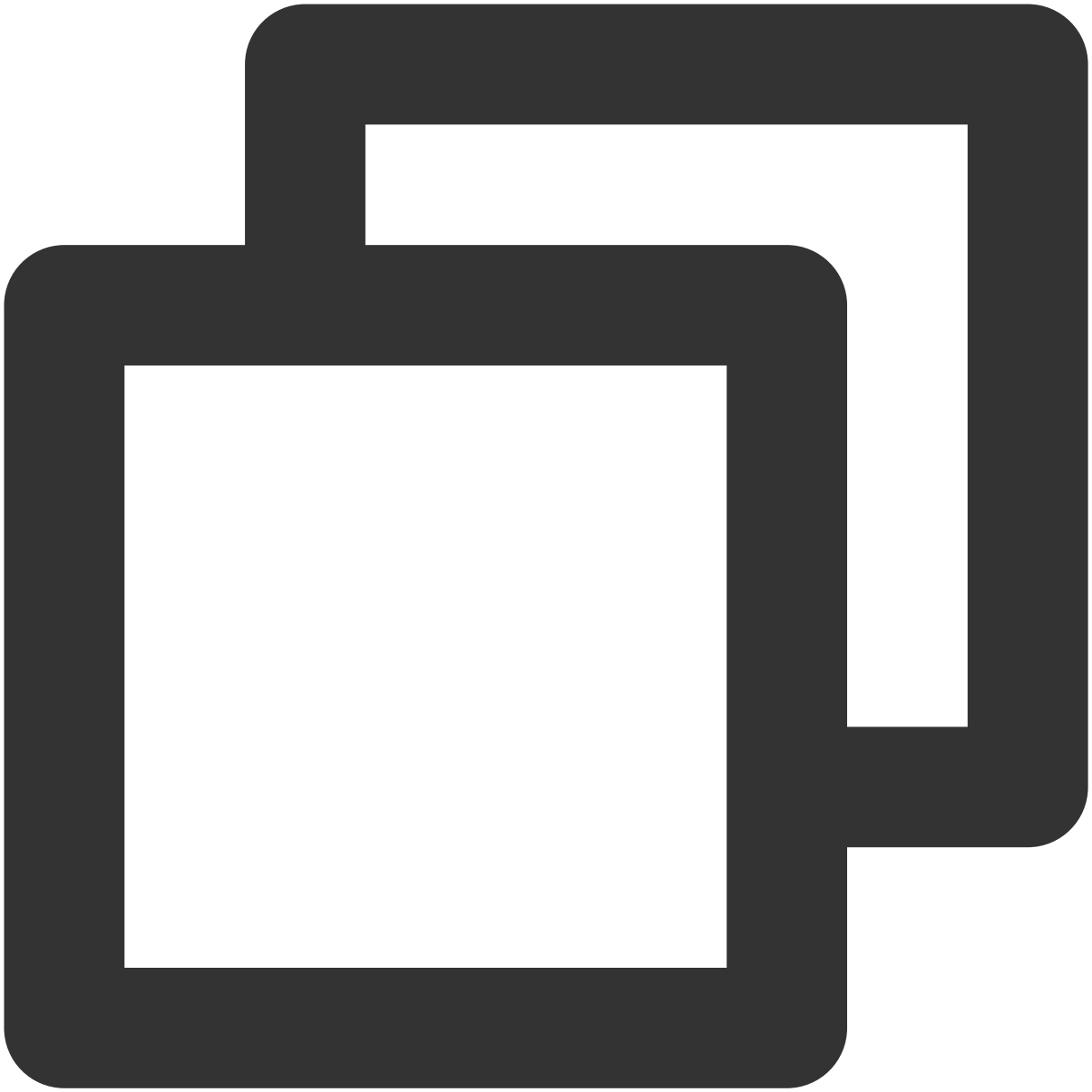
```
testdb1=>create server srv_test1 foreign data wrapper postgres_fdw options (host
CREATE SERVER
```

目标实例在腾讯云 VPN 接入的实例。



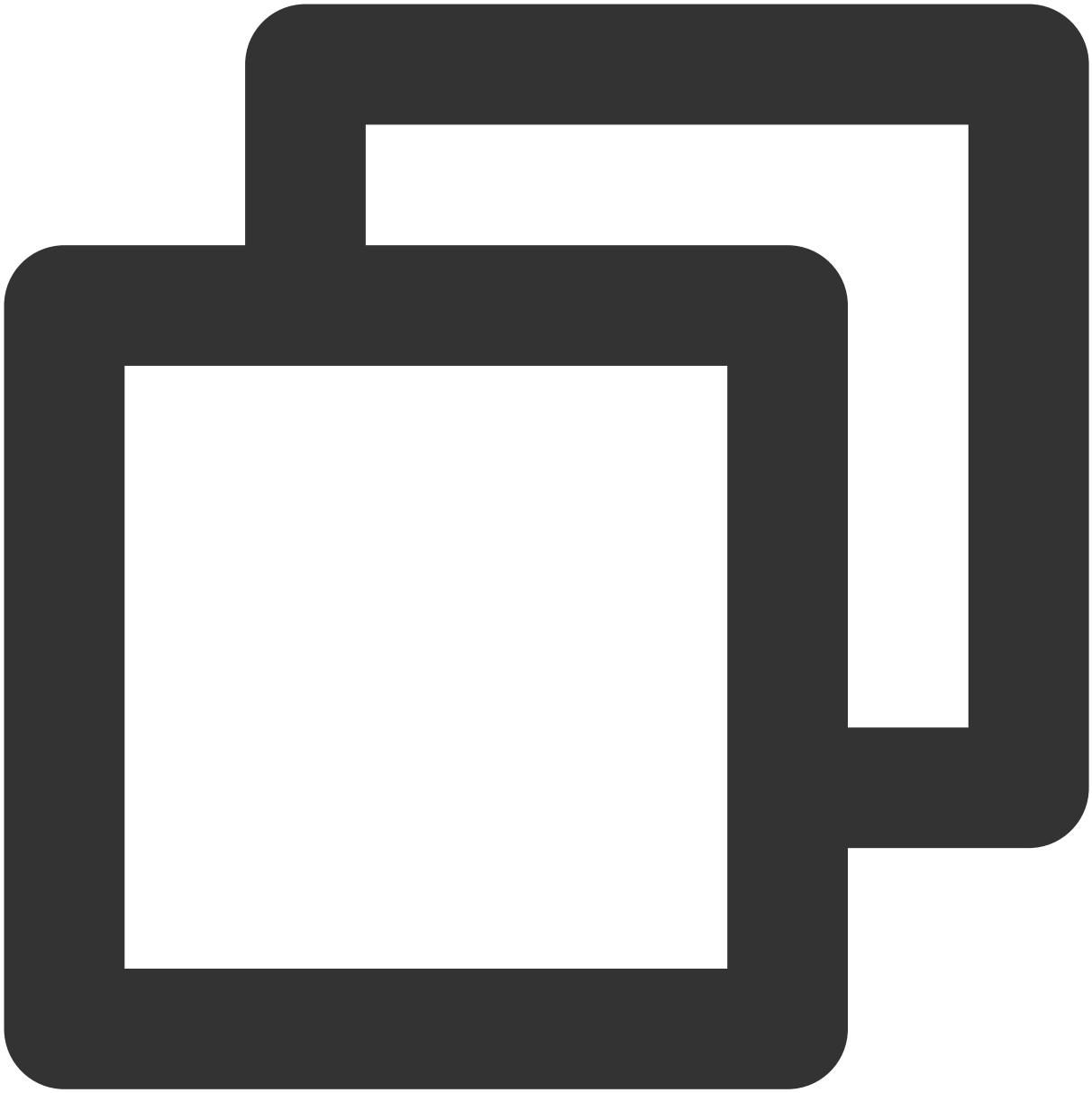
```
testdb1=>create server srv_test1 foreign data wrapper postgres_fdw options (host
```

目标实例在自建 VPN 接入的实例。



```
testdb1=>create server srv_test1 foreign data wrapper postgres_fdw options (host
```

目标实例在腾讯云专线接入的实例。

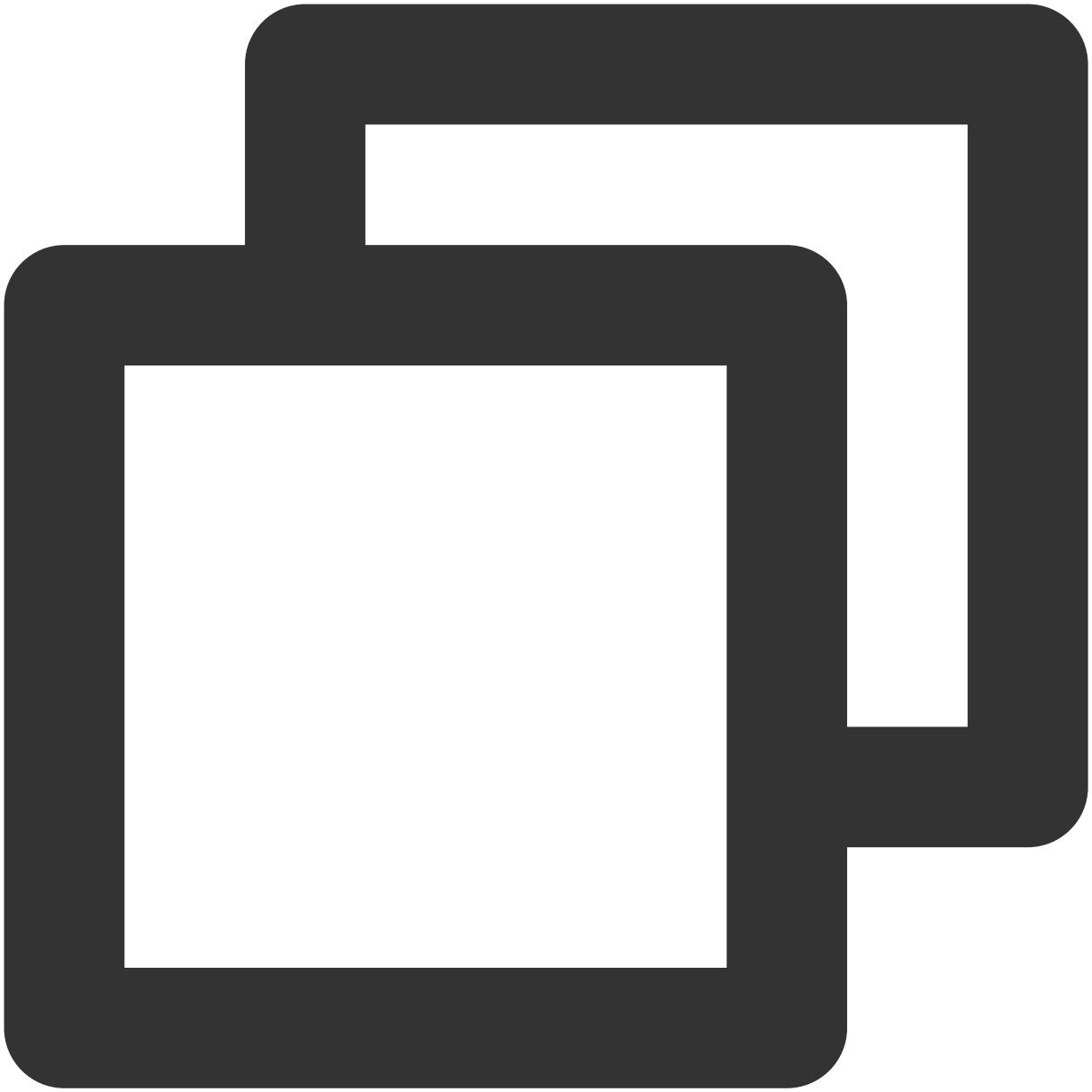


```
testdb1=>create server srv_test1 foreign data wrapper postgres_fdw options (host
CREATE SERVER
```

步骤4：创建用户映射

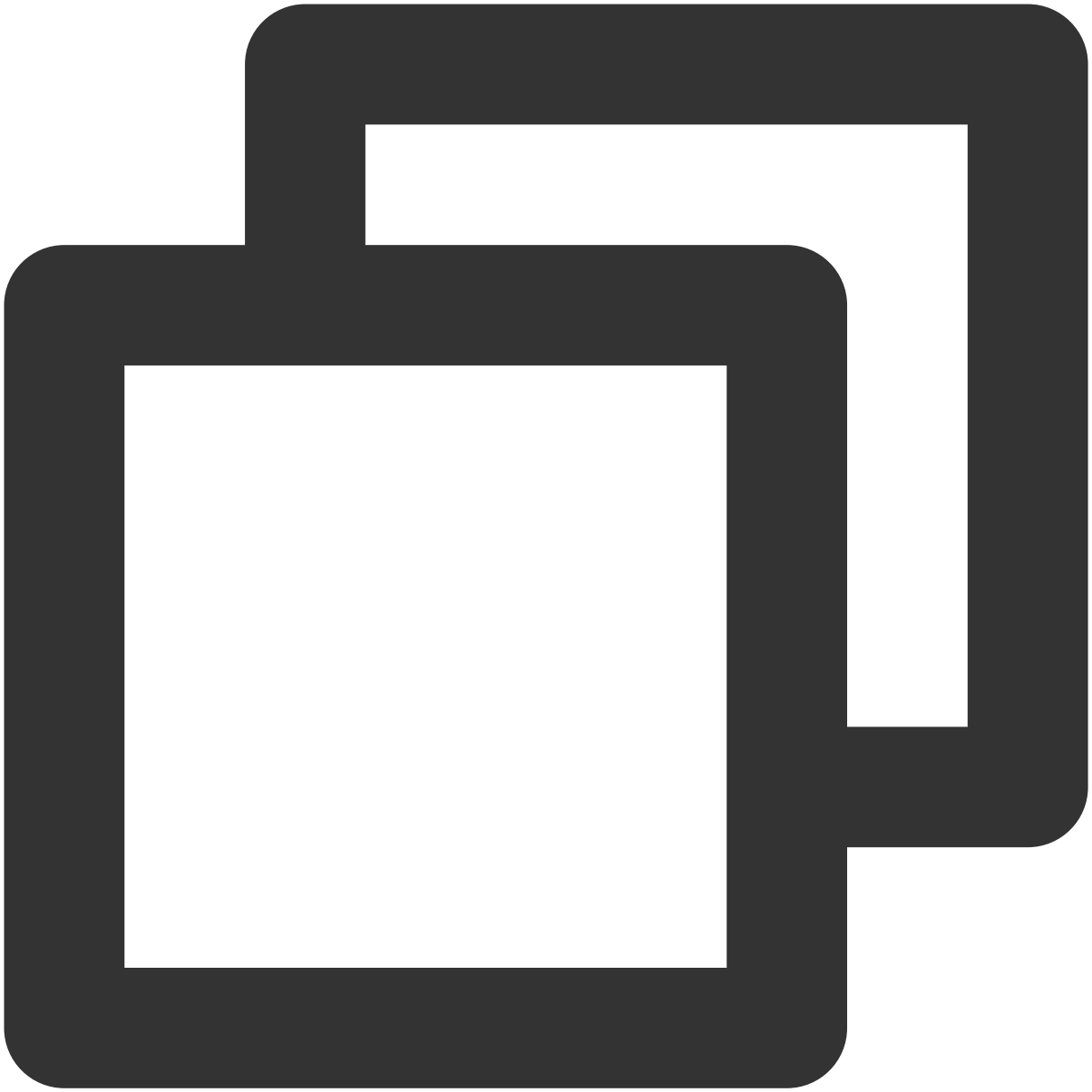
说明：

同实例的跨 database 访问则可跳过此步骤。



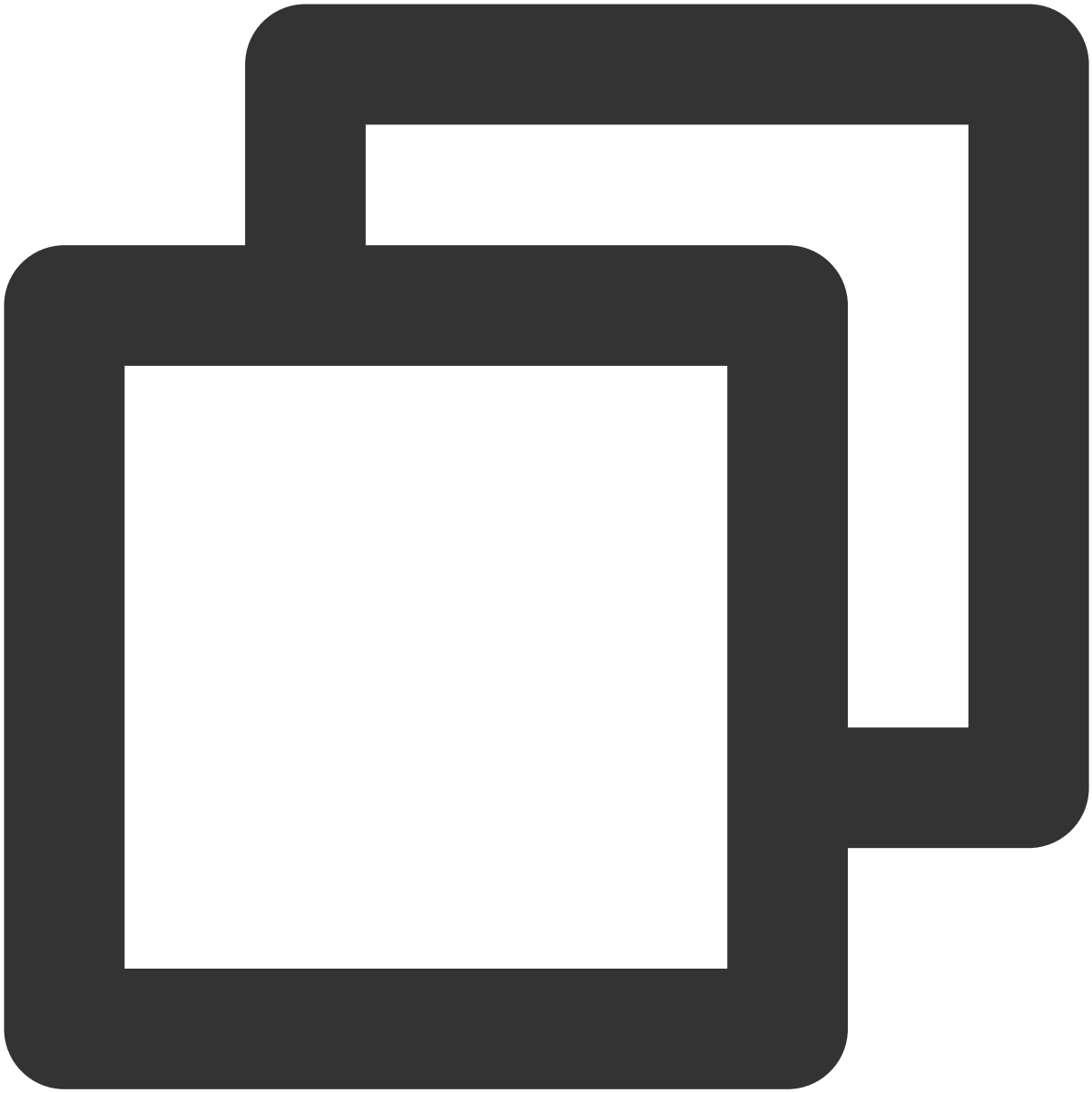
```
testdb1=> create user mapping for user1 server srv_test1 options (user 'user2',pass
CREATE USER MAPPING
```

步骤5：创建外部表



```
testdb1=> create foreign table foreign_table1(id integer) server srv_test1 options(
CREATE FOREIGN TABLE
```

步骤6：访问外部数据



```
testdb1=> select * from foreign_table1;
  id
----
   1
(1 row)
```

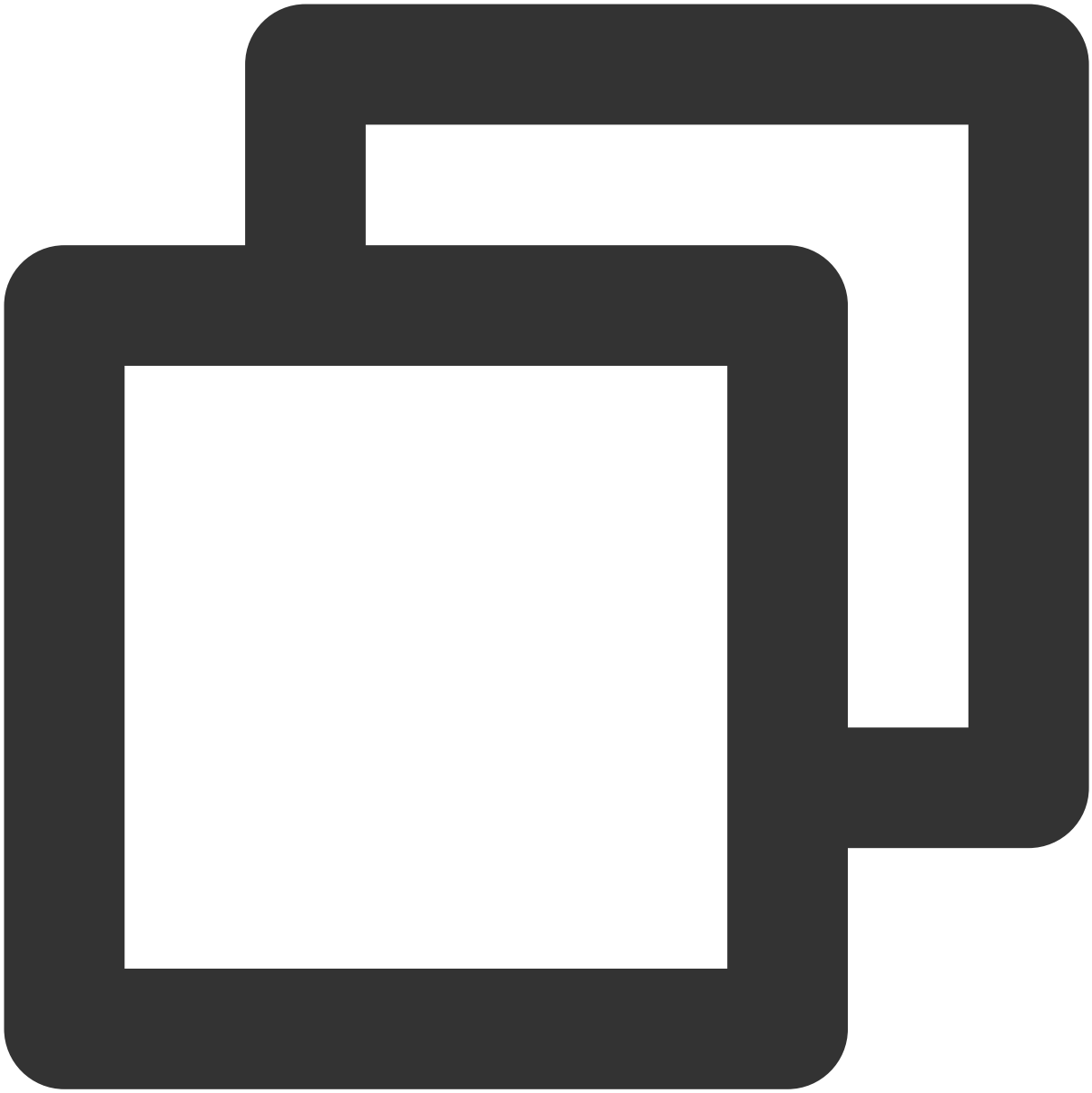
参考链接

postgres_fdw 官方介绍

9.3 版本 SERVER 创建

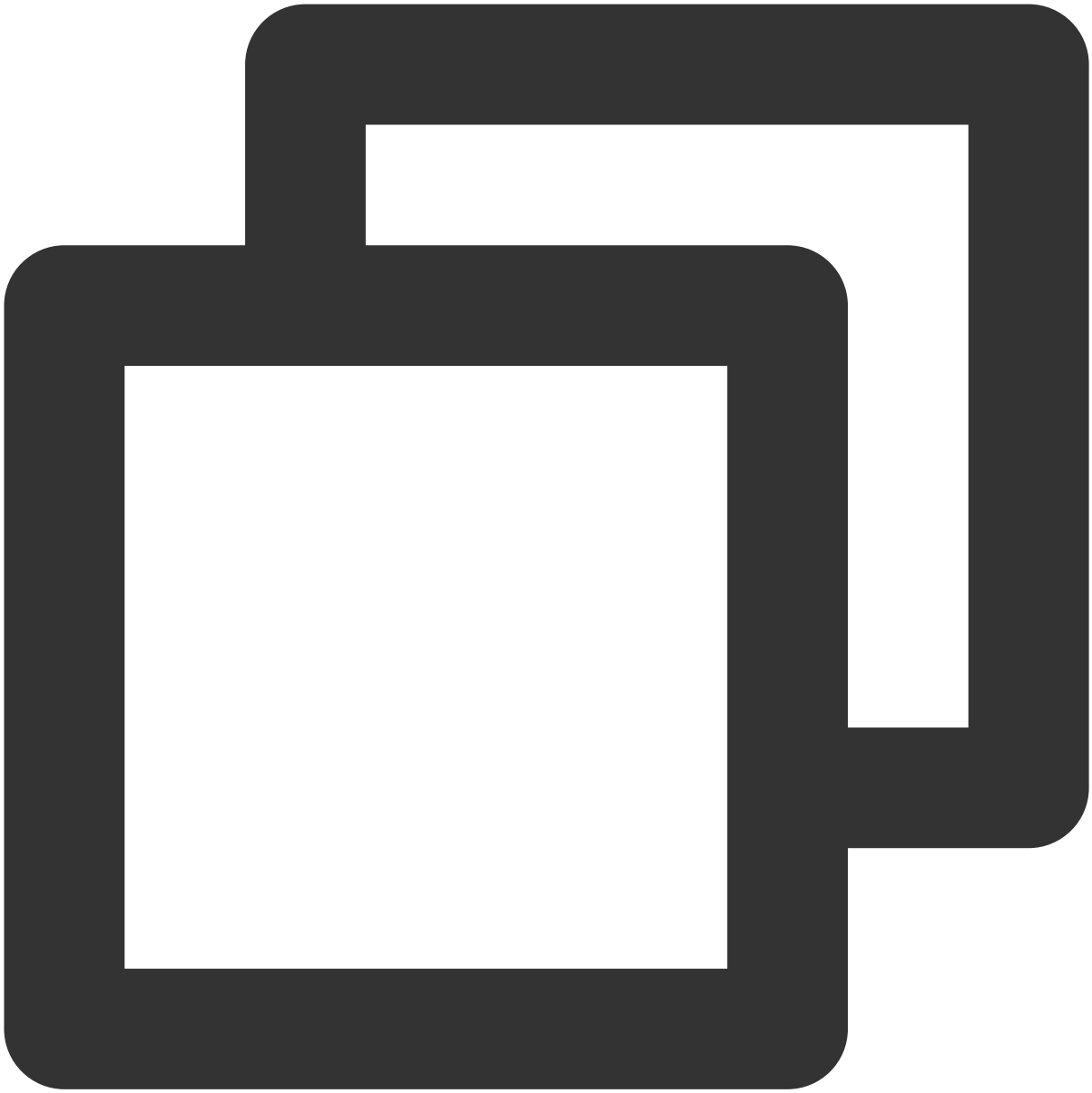
使用 dblink 示例

步骤一：创建 dblink 插件



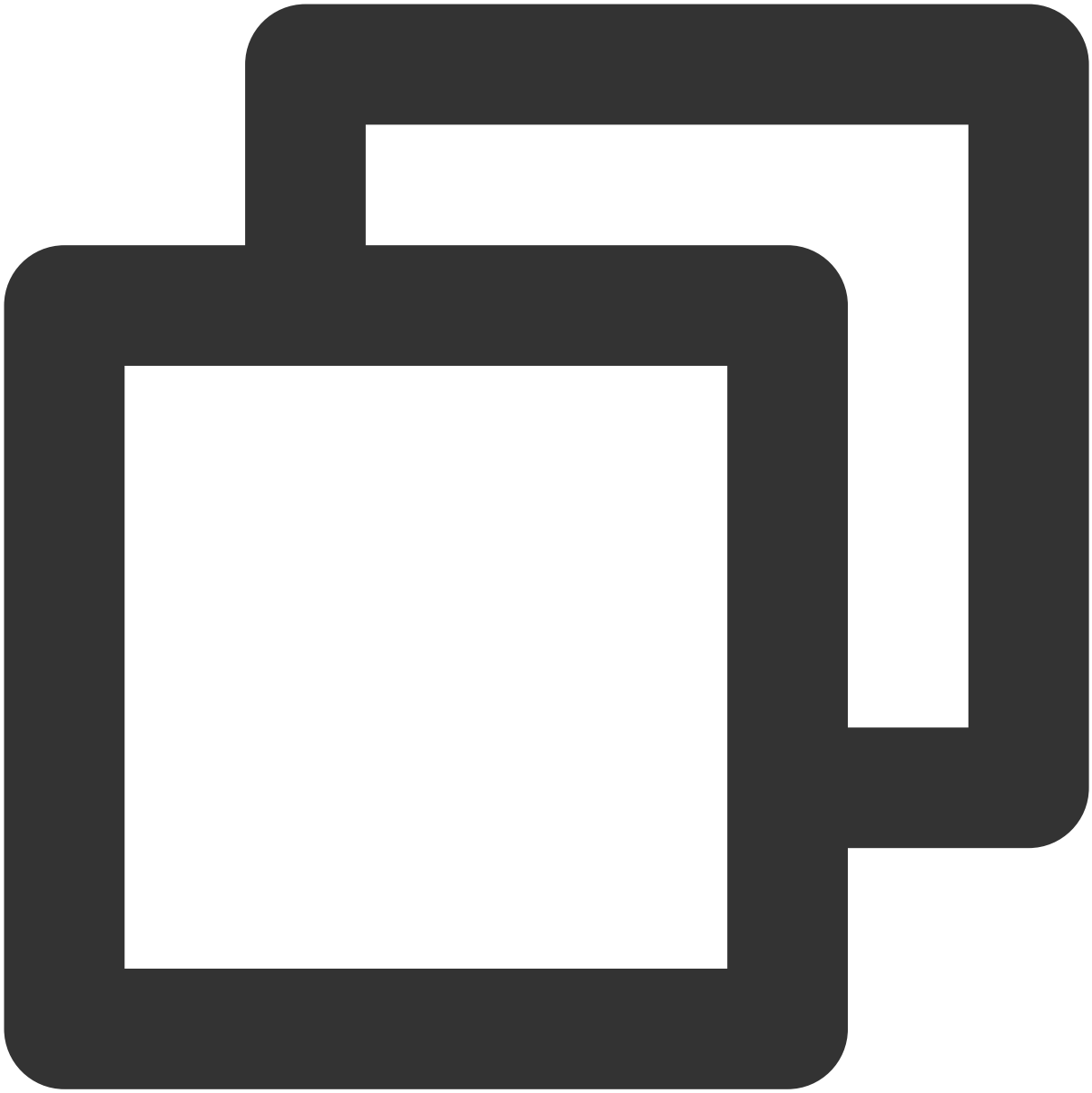
```
postgres=> create extension dblink;
postgres=> \\dx
                                       List of installed extensions
        Name         | Version |   Schema   |                              Description
---------------------+---------+------------+---------------------------------------
 dblink              | 1.2     | public     | connect to other PostgreSQL databases
 pg_stat_log         | 1.0     | public     | track runtime execution statistics of
 pg_stat_statements  | 1.6     | public     | track execution statistics of all SQL
 plpgsql             | 1.0     | pg_catalog | PL/pgSQL procedural language
(4 rows)
```

步骤二：建立 dblink 链接



```
select dblink_connect('yunpg1','host=10.10.10.11 port=5432 instanceid=postgres-2123
 dblink_connect
----------------
 OK
(1 row)
```

步骤三：访问外部数据



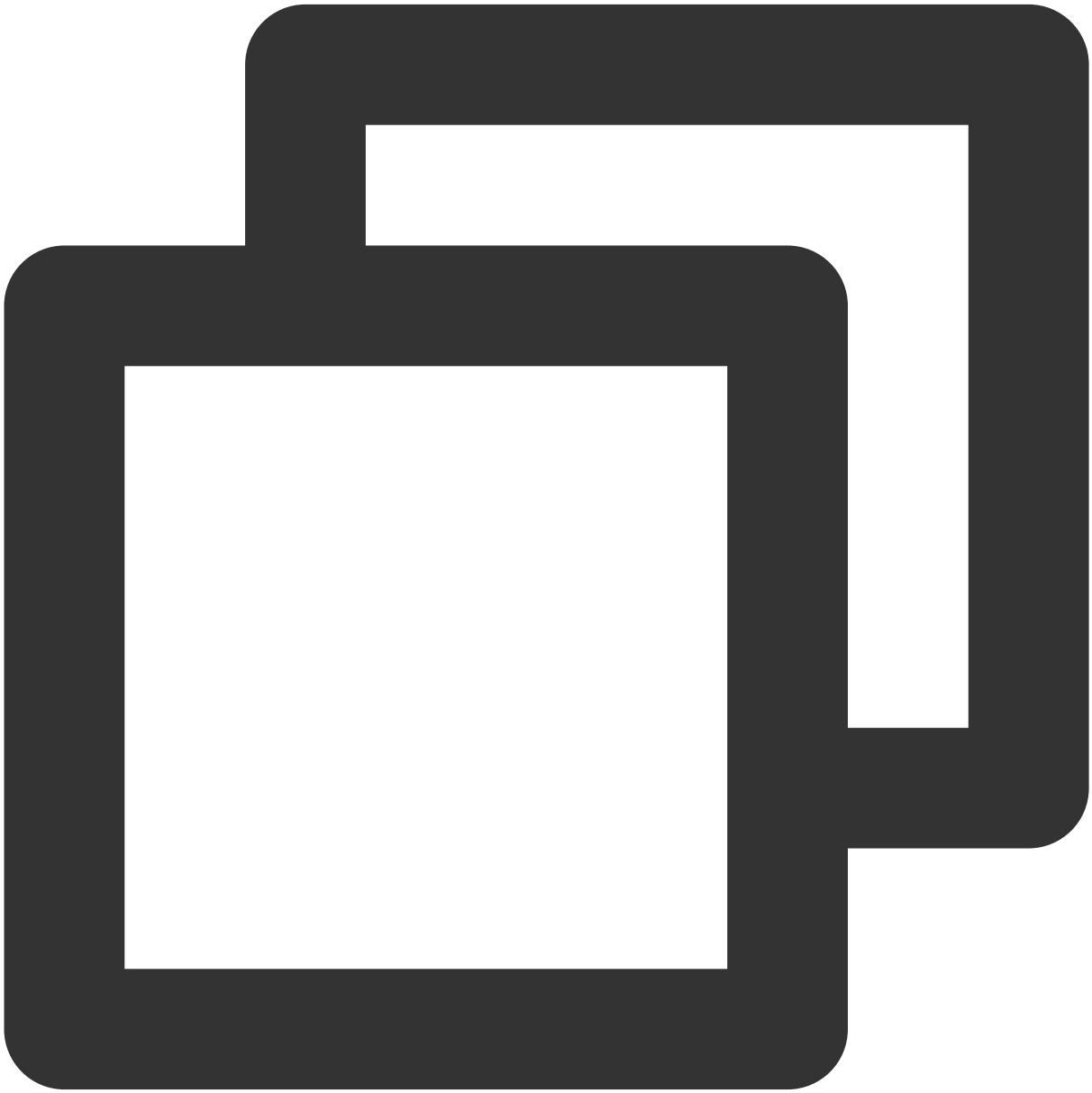
```
postgres=> select * from dblink('yunpg1','select catalog_name,schema_name,schema_ow
    a     |         b          |    c
----------+--------------------+---------
 postgres | pg_toast           | user_00
 postgres | pg_temp_1          | user_00
 postgres | pg_toast_temp_1    | user_00
 postgres | pg_catalog         | user_00
 postgres | public             | user_00
 postgres | information_schema | user_00
(6 rows)
```

参考链接

dblink 官方介绍

使用 mysql_fw 示例

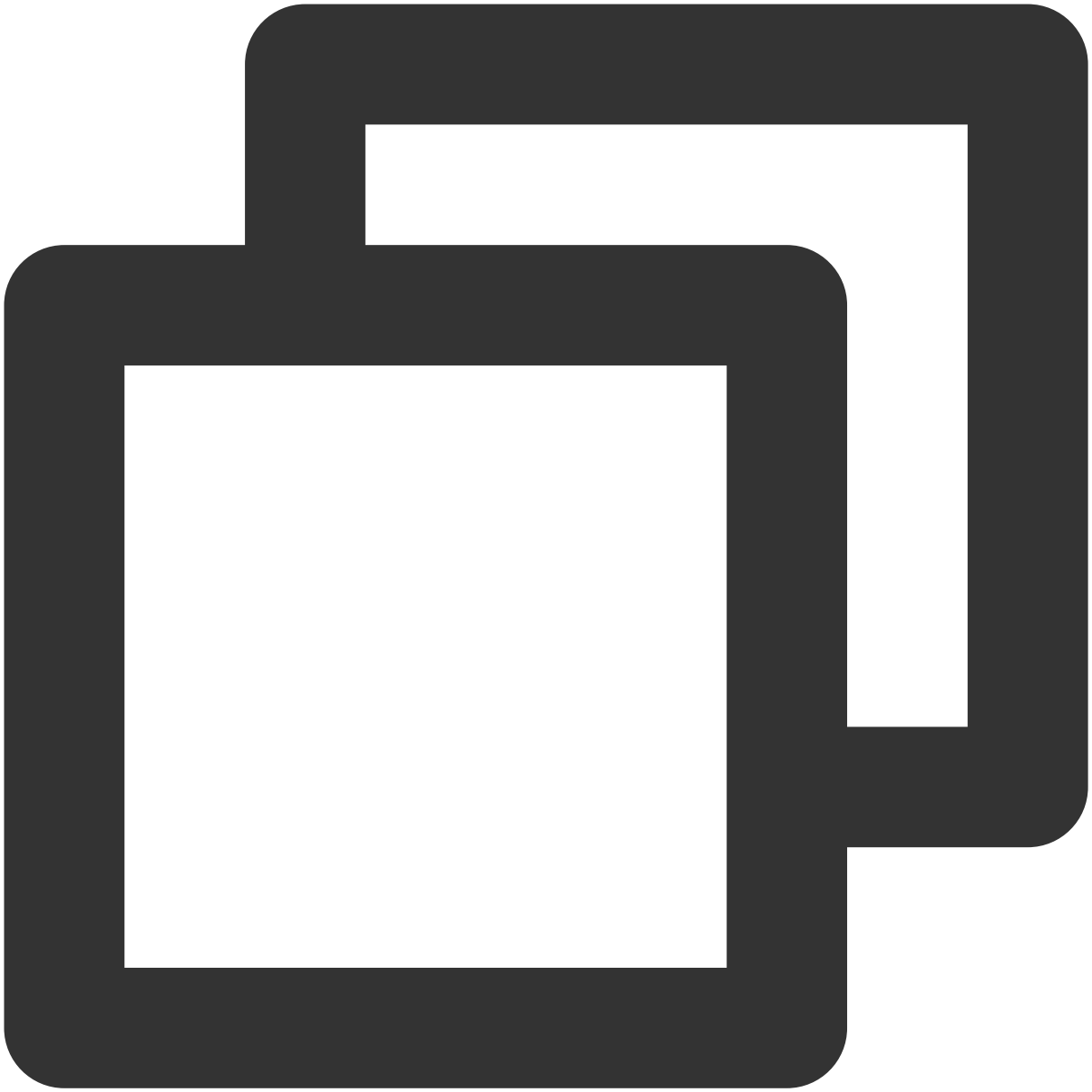
步骤一：创建 mysql_fdw 插件



```
postgres=> create extension mysql_fdw;
CREATE EXTENSION
```

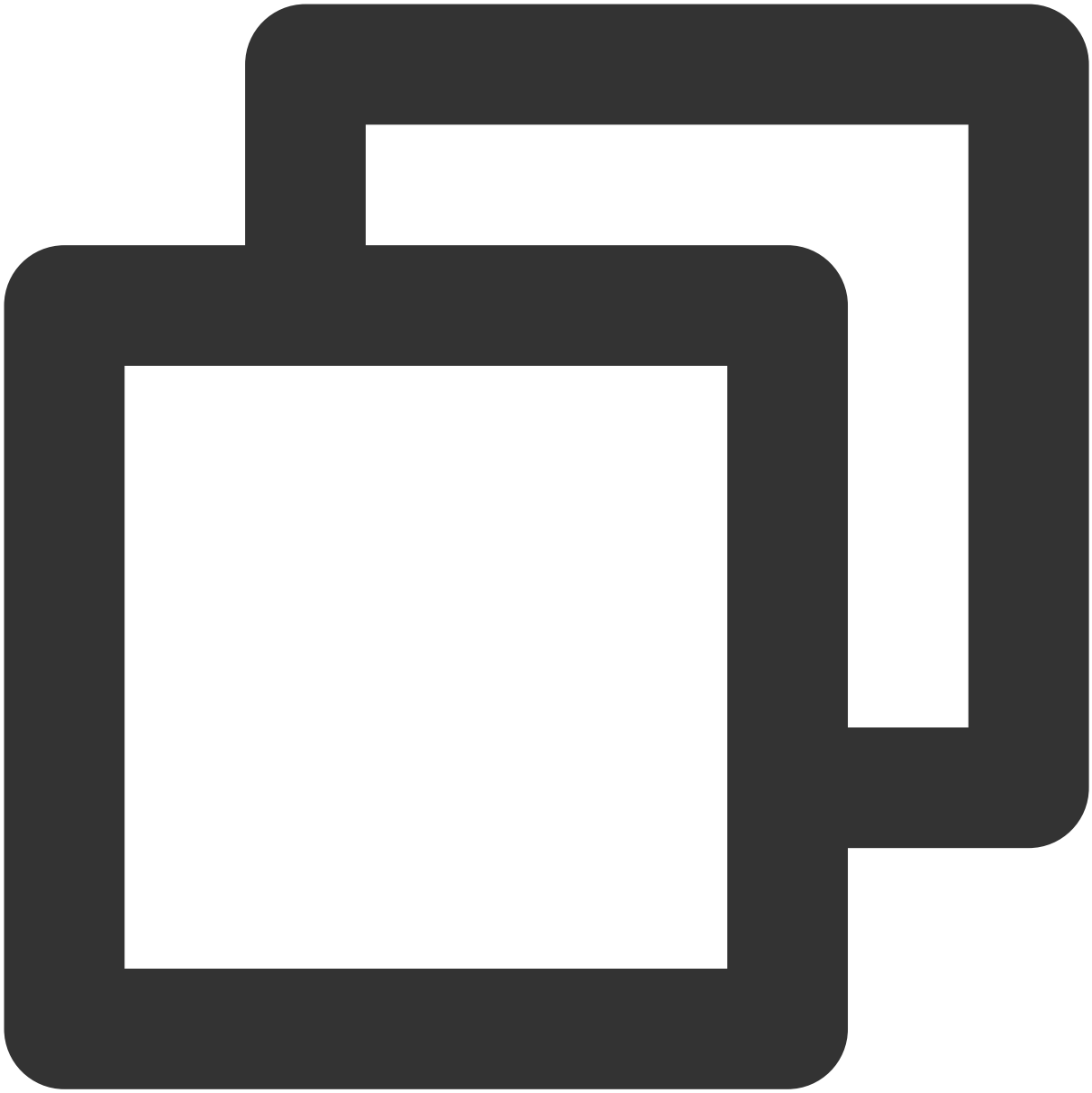
```
postgres=> \\dx;
                                          List of installed extensions
        Name         | Version |   Schema   |                                Descripti
---------------------+---------+------------+---------------------------------------
 dblink              | 1.2     | public     | connect to other PostgreSQL databases
 mysql_fdw           | 1.1     | public     | Foreign data wrapper for querying a My
 pg_stat_log         | 1.0     | public     | track runtime execution statistics of
 pg_stat_statements | 1.9     | public     | track planning and execution statistic
 plpgsql             | 1.0     | pg_catalog | PL/pgSQL procedural language
(5 rows)
```

步骤二：创建 SERVER



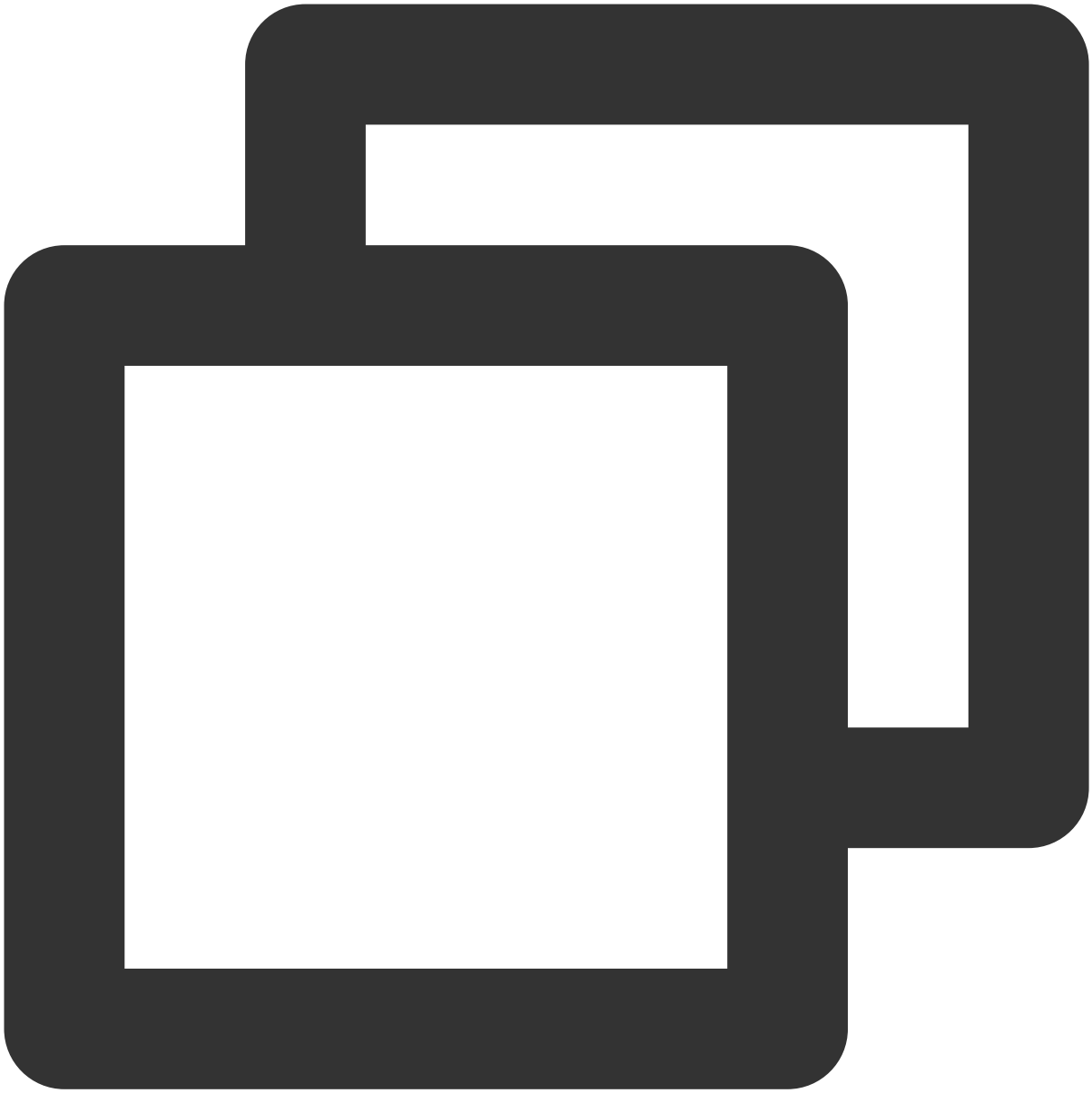
```
postgres=> CREATE SERVER mysql_svr  FOREIGN DATA WRAPPER mysql_fdw OPTIONS (host '1
CREATE SERVER
```

步骤三：创建外部用户映射



```
postgres=> CREATE USER MAPPING FOR PUBLIC SERVER mysql_svr OPTIONS (username 'fdw_u
CREATE USER MAPPING
```

步骤四：访问外部数据



```
postgres=> IMPORT FOREIGN SCHEMA hrdb FROM SERVER mysql_svr INTO public;
```

参考链接

mysql_fdw 官方介绍

使用 cos_fdw 示例

cos_fdw 使用示例请参考文档 通过 cos_fdw 插件支持分级存储能力。

使用注意

目标实例，需要注意以下几点：

1. 需要放开 PostgreSQL 的 hba 限制，允许创建的映射用户（如：user2）以 MD5 方式访问。hba 的修改可参考 PostgreSQL 官方说明。

2. 如果目标实例非 TencentDB 实例，且搭建有热备模式，当主备切换后，需要自行更新 server 连接地址或者重新创建 server。

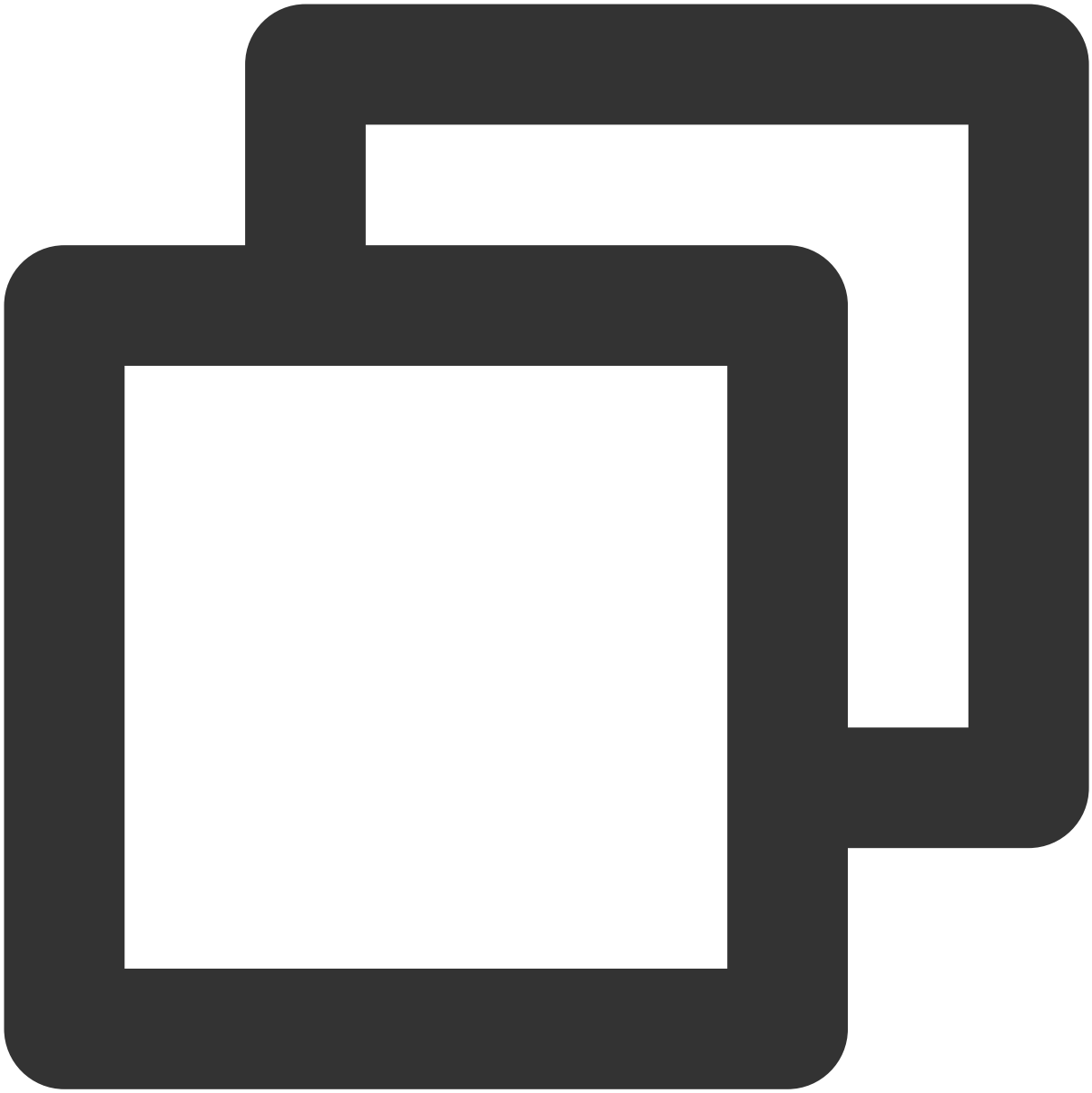
如何在 PostgreSQL 中自动创建分区

最近更新时间：2024-01-22 16:24:46

在 PostgreSQL 老版本中可通过继承支持表分区功能，如按时间，每月创建一个表分区，数据记录到对应分区中。在 PostgreSQL 10版本后，同样也支持了声明式分区了。本文将介绍如何提前创建分区或者根据写入数据实时创建分区。
下面将是常见的几种 PostgreSQL 自动创建分区表的方案。

场景

分区表在实际使用中，一般以时间字段作为分区键。如分区字段类型为 timestamp，分区方式为 List of values。
表结构如下：



```
CREATE TABLE tab
(
    id   bigint GENERATED ALWAYS AS IDENTITY,
    ts   timestamp NOT NULL,
    data text
) PARTITION BY LIST ((ts::date));
CREATE TABLE tab_def PARTITION OF tab DEFAULT;
```

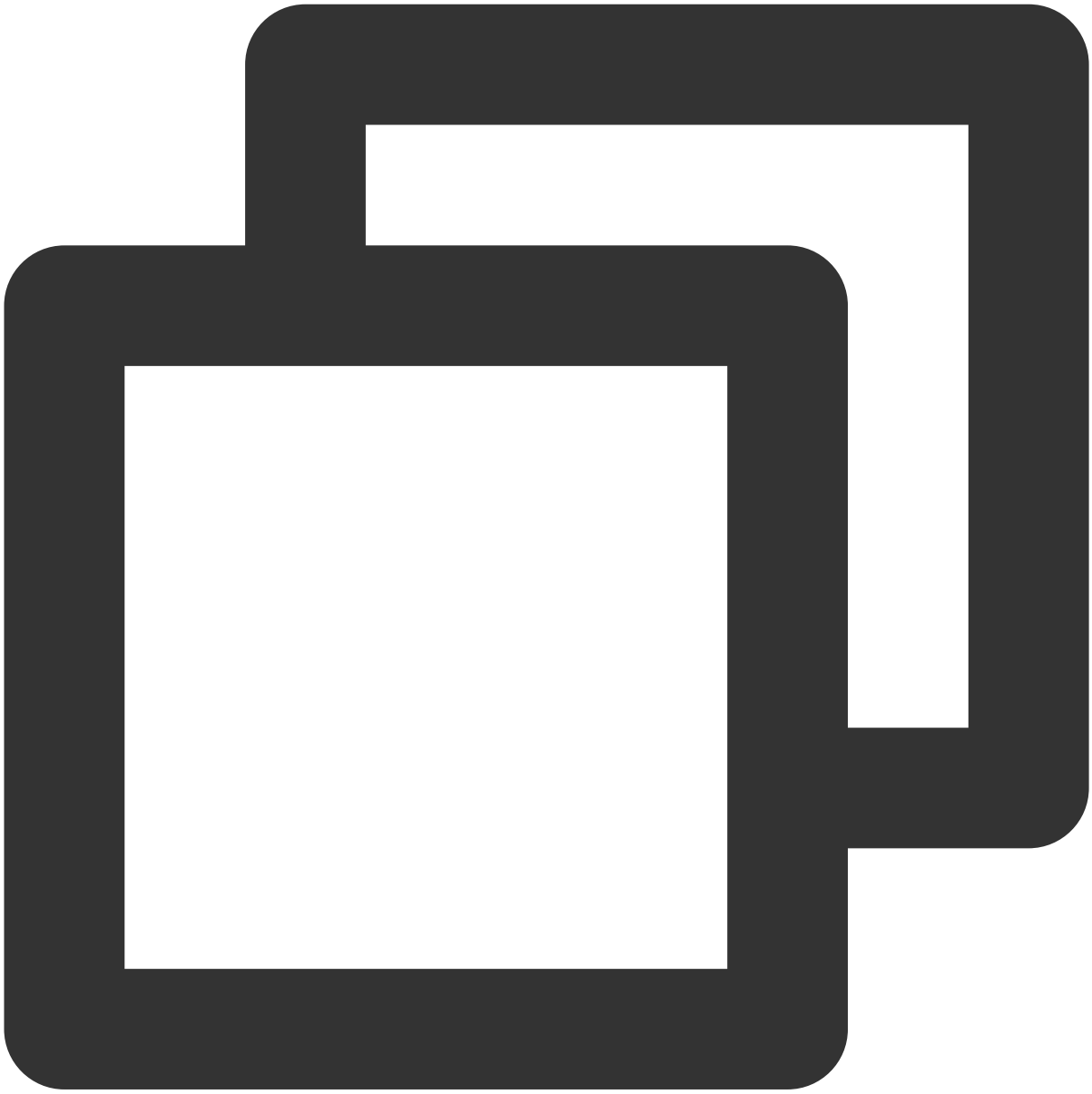
分区的创建一般分以下两种场景：

一、定时提前创建分区

定时提前创建分区只需一个定时任务调度工具即可实现，常见的定时任务调度工具和创建分区方法如下：

使用系统调度器，如 Crontab (Linux, Unix, etc.) 和 Task Scheduler (Windows)

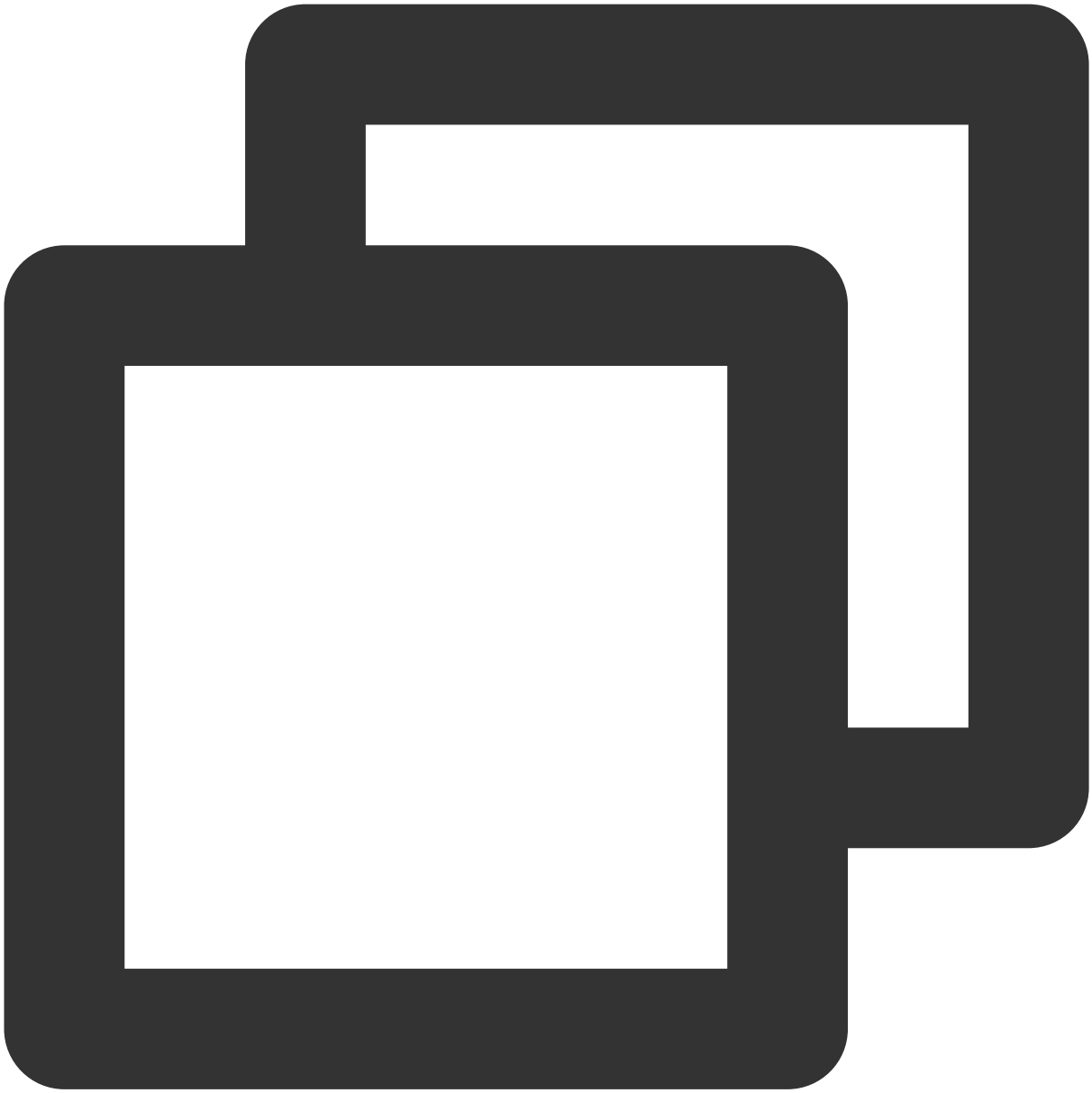
以 Linux 操作系统为例，每天下午14点创建次日的分区表：



```
cat > /tmp/create_part.sh <<EOF
dateStr=\\$(date -d '+1 days' +%Y%m%d);
psql -c "CREATE TABLE tab_\\$dateStr (LIKE tab INCLUDING INDEXES); ALTER TABLE tab
EOF
(crontab -l 2>/dev/null; echo "0 14 * * * bash /tmp/create_part.sh ") | crontab -
```

使用数据库内置调度器，如 pg_cron、pg_timetable

以 pg_cron 为例，每天下午14点创建次日的分区表：

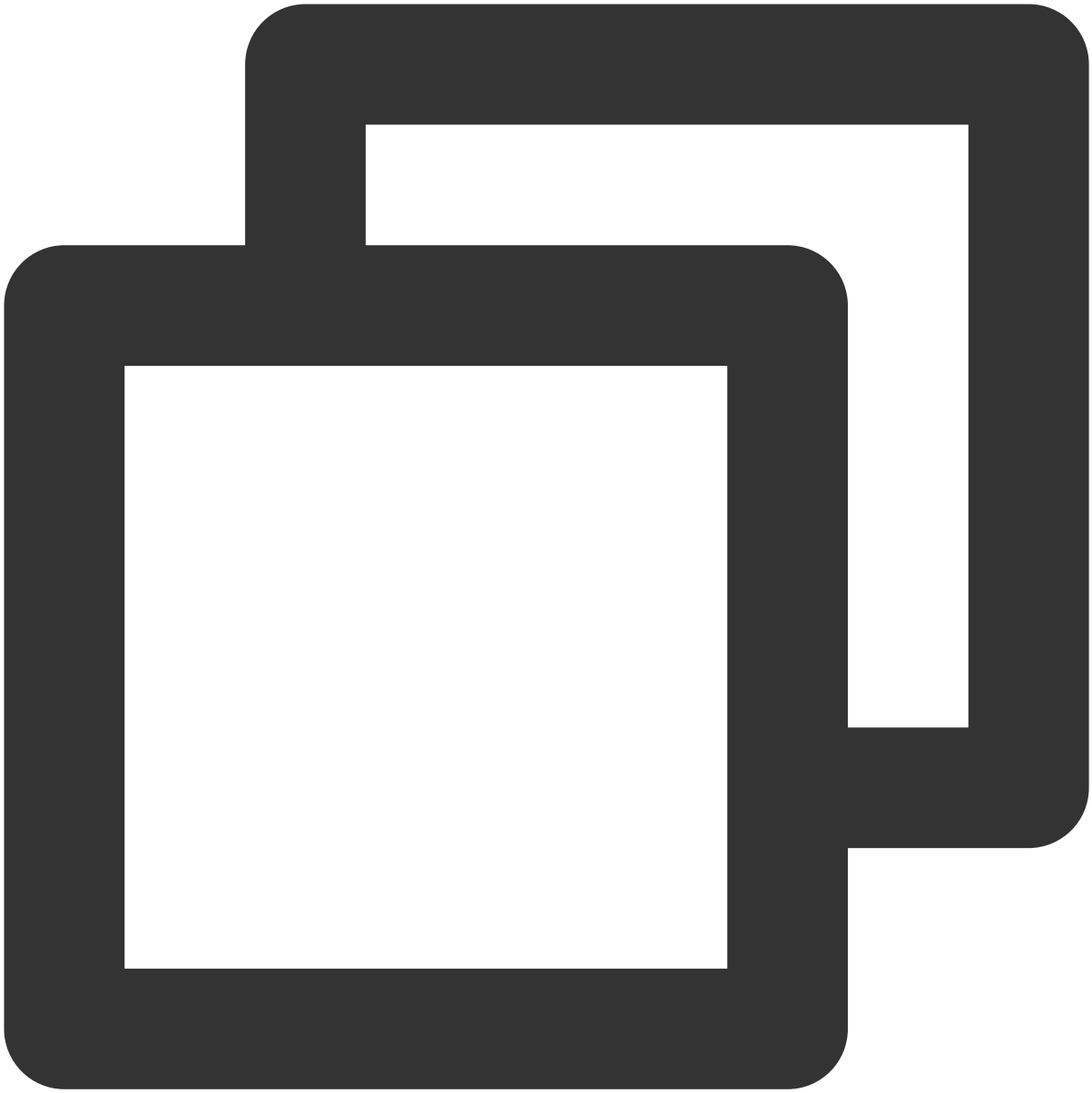


```
CREATE OR REPLACE FUNCTION create_tab_part() RETURNS integer
    LANGUAGE plpgsql AS
$$
DECLARE
    dateStr varchar;
BEGIN
    SELECT to_char(DATE 'tomorrow', 'YYYYMMDD') INTO dateStr;
    EXECUTE
```

```
        format('CREATE TABLE tab_%s (LIKE tab INCLUDING INDEXES)', dateStr);
    EXECUTE
        format('ALTER TABLE tab ATTACH PARTITION tab_%s FOR VALUES IN (%L)', dateSt
    RETURN 1;
END;
$$;

CREATE EXTENSION pg_cron;

SELECT cron.schedule('0 14 * * *', $$SELECT create_tab_part();$$);
```

使用专门的分区管理插件，如 pg_partman

以 pg_partman 为例，每天提前创建次日的分区表；



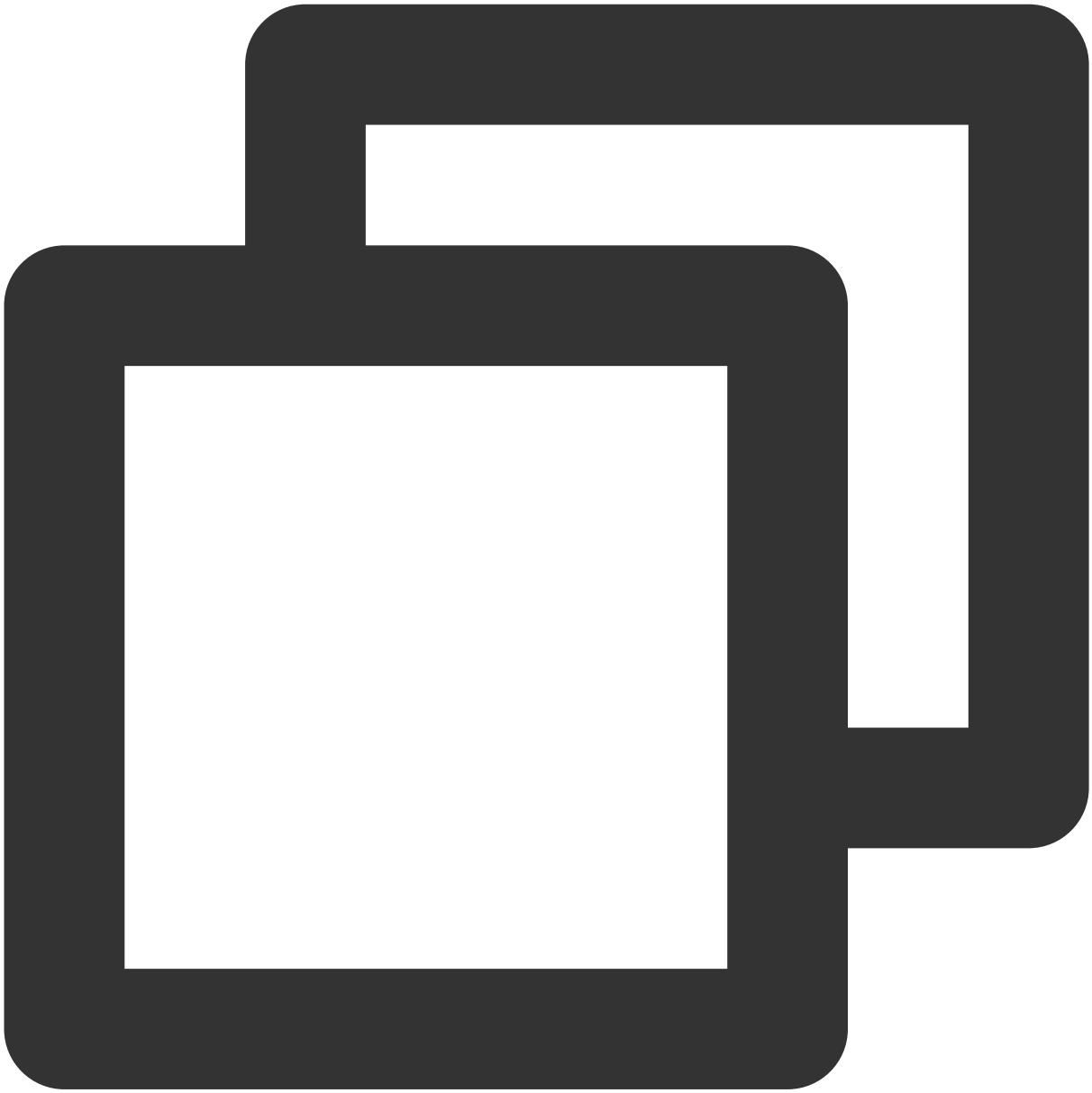
```
CREATE EXTENSION pg_partman;

SELECT partman.create_parent(p_parent_table => 'public.tab',
                             p_control => 'ts',
                             p_type => 'native',
                             p_interval=> 'daily',
                             p_premake => 1);
```

二、按需实时创建分区

如需按数据插入的需要来创建分区，可根据分区是否存在来判断该时间区间内有无数据的存在，一般采用触发器来实现。

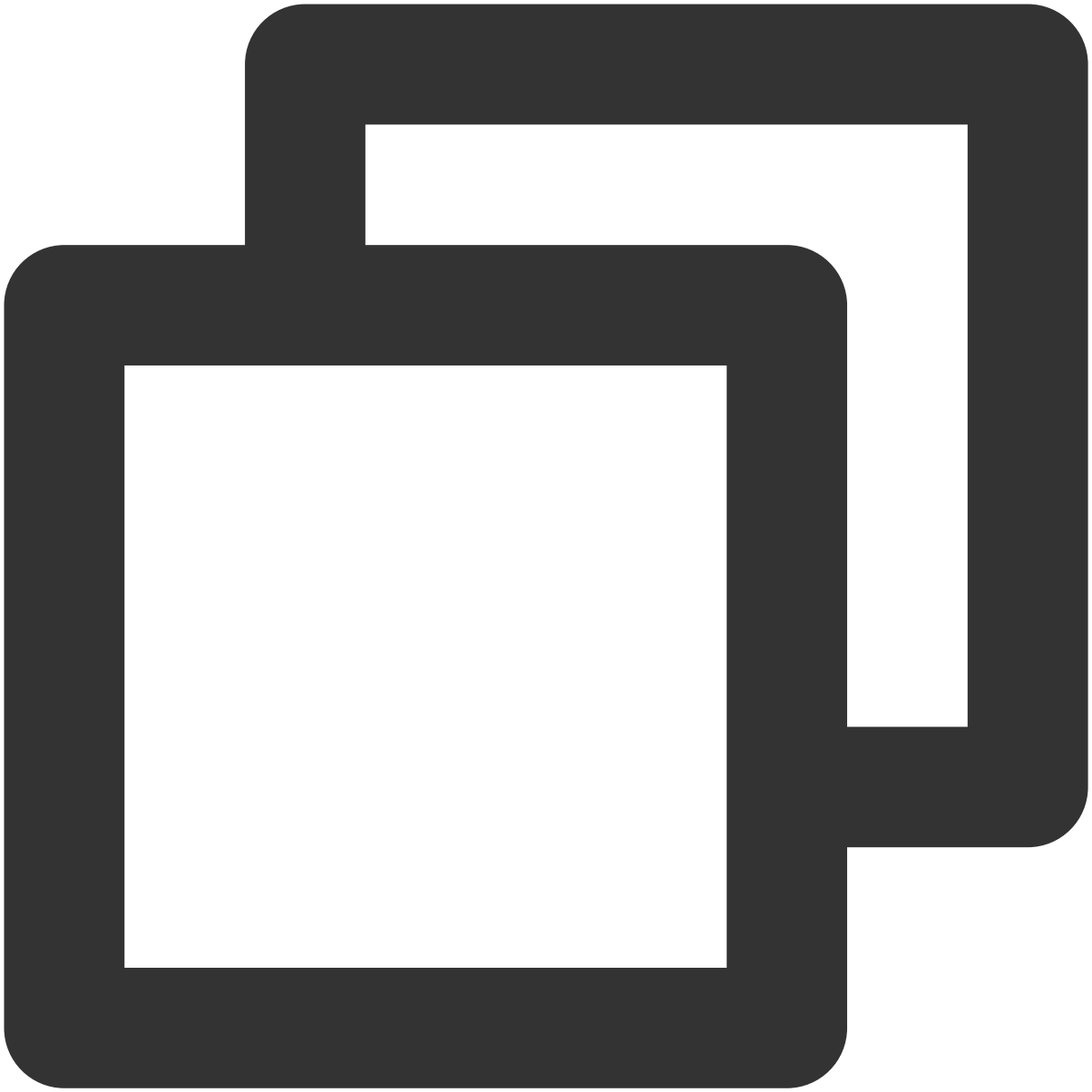
需注意此方法存在以下两个问题：

PostgreSQL 13及以上的版本才提供针对分区表的 BEFORE/FOR EACH ROW 触发器。



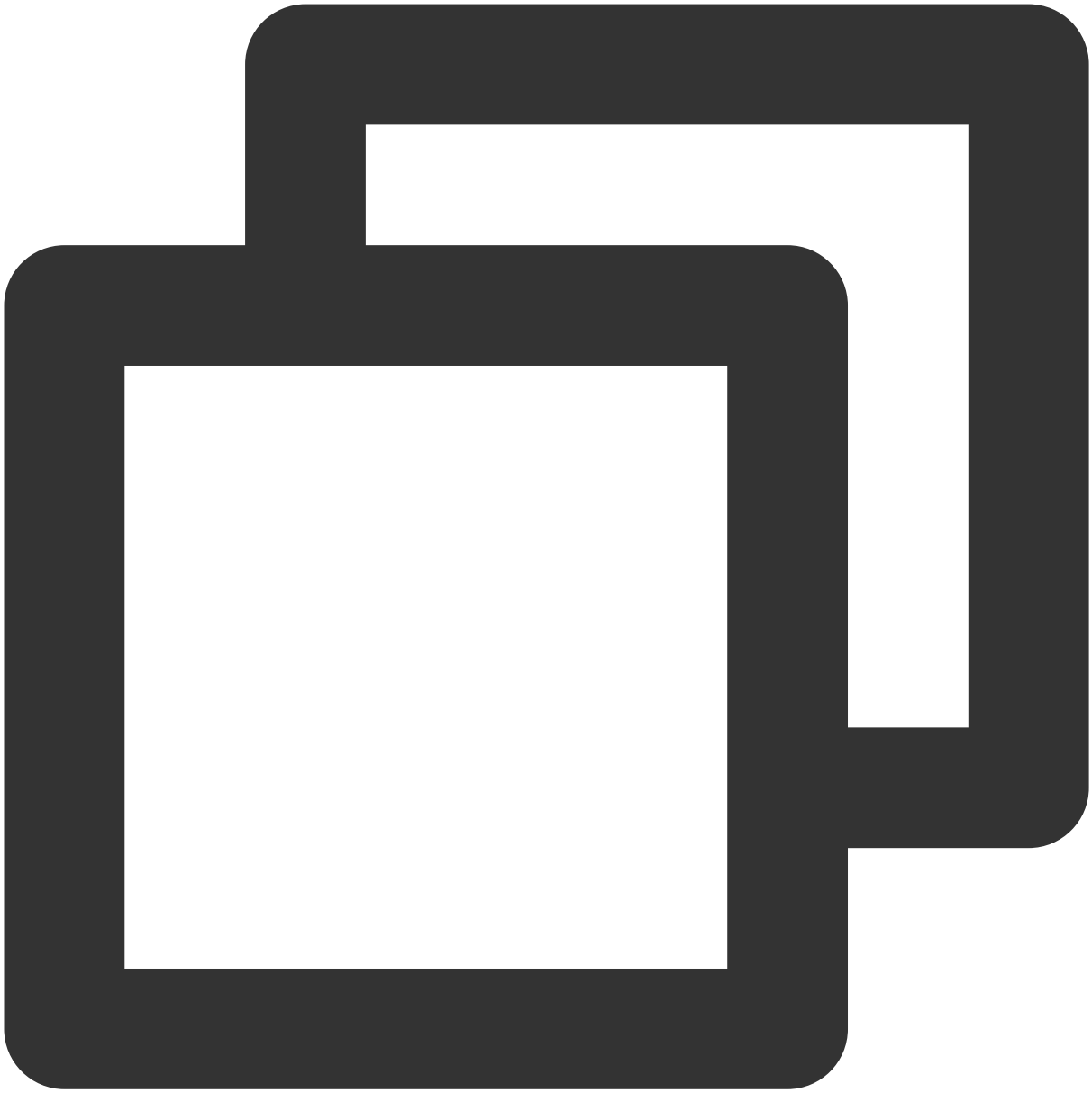
```
ERROR:  "tab" is a partitioned table
DETAIL:  Partitioned tables cannot have BEFORE / FOR EACH ROW triggers.
```

插入数据时，因为锁表的原因，无法修改分区表定义，即无法 ATTACH 子表， 因此必须有另一个连接来做 ATTACH 的操作，此处可以用 LISTEN/NOTIFY 机制来通知另一个连接进行分区定义的修改。



```
ERROR:  cannot CREATE TABLE .. PARTITION OF "tab"
      because it is being used by active queries in this session
或
ERROR:  cannot ALTER TABLE "tab"
      because it is being used by active queries in this session
```

触发器（实施子表创建和 NOTIFY）



```
CREATE FUNCTION part_trig() RETURNS trigger
    LANGUAGE plpgsql AS
$$
BEGIN
    BEGIN
        /* try to create a table for the new partition */
        EXECUTE
            format('CREATE TABLE %I (LIKE tab INCLUDING INDEXES)', 'tab_' || to_cha

        /*
         * tell listener to attach the partition
```

```
         * (only if a new table was created)
         */
        EXECUTE
            format('NOTIFY tab, %L', to_char(NEW.ts, 'YYYYMMDD'));
    EXCEPTION
        WHEN duplicate_table THEN
            NULL; -- ignore
    END;

    /* insert into the new partition */
    EXECUTE
        format('INSERT INTO %I VALUES ($1.*)', 'tab_' || to_char(NEW.ts, 'YYYYMMDD'
        USING NEW;

    /* skip insert into the partitioned table */
    RETURN NULL;
END;
$$;

CREATE TRIGGER part_trig
    BEFORE INSERT
    ON TAB
    FOR EACH ROW
    WHEN (pg_trigger_depth() < 1)
EXECUTE FUNCTION part_trig();
```

代码(实施 LISTEN 和子表 ATTACH)

```
#!/usr/bin/env python3.9
# encoding:utf8
import asyncio

import psycopg2
from psycopg2.extensions import ISOLATION_LEVEL_AUTOCOMMIT

conn = psycopg2.connect('application_name=listener')
conn.set_isolation_level(ISOLATION_LEVEL_AUTOCOMMIT)
cursor = conn.cursor()
cursor.execute(f'LISTEN tab;')


def attach_partition(table, date):
    with conn.cursor() as cs:
        cs.execute('ALTER TABLE "%s" ATTACH PARTITION "%s_%s" FOR VALUES IN (\\'%s\


def handle_notify():
    conn.poll()
    for notify in conn.notifies:
```

```
            print(notify.payload)
            attach_partition(notify.channel, notify.payload)
        conn.notifies.clear()

loop = asyncio.get_event_loop()
loop.add_reader(conn, handle_notify)
loop.run_forever()
```

总结

本文向您介绍了两种自动创建分区的方案，以下为每种方案的总结分析：
定时提前创建分区场景下的几种解决方案简单易懂，但会依赖系统或插件的定时管理机制，在运维、迁移时具有额外管理成本。
按需实时创建分区场景下，能按实际数据规律减少不必要的分区数量，但也需要较高版本(≥13)及额外连接来完成，复杂度比较高。
您可根据业务情况，选择合适的自动创建分区方式。

场景	版本	实现难易度	是否需要系统调度器或插件工具	是否需要额外连接机制	成本
定时提前创建分区	PostgreSQL 10	较简单	是	否	相对较高
按需实时创建分区	大于等于 PostgreSQL 13	较复杂	否	是	相对较低

基于 pg_roaringbitmap 实现超大规模标签查找

最近更新时间：2024-01-22 16:24:46

业务应用场景中常有通过标签进行筛选查询的功能，在数据量较大且标签值较多的场景下，数据容量会占用很多，性能也会较差，如何高效快速且不占用太多数据容量的情况下进行目标资源筛查，成为业务管理优化的不变话题。本文为您介绍如何基于 pg_roaringbitmap 插件轻松实现超大规模标签的查找。

pg_roaringbitmap 概述

pg_roaringbitmap 是一个基于 roaringbitmap 而实现的压缩位图存储数据插件，支持 roaring bitmap 的存取、集合操作，聚合等运算。

roaringbitmap 用途

roaringbitmap 在业务中常用来存储用户的属性标签，可增删改查这些属性标签以及根据这些存储的用户的标签，通过并集、交集等方法来筛选出特定的用户，以达到超大规模属性数据的精准快速查找，既提升了性能，同时也能降低存储空间，是大数据分析场景下极佳的应用实践。
如在传统模式下如有一张音乐类应用的用户标签表，如下表：

用户 ID	用户名	兴趣标签
1	张三	{古典,爵士,R&B,乡村}
2	黎四	{民歌,纯音乐}
3	王五	{HipHop,爵士,R&B,嘻哈,雷鬼}
...	...	...
1000000000	文本2	{摇滚,}

如想要找到喜欢纯音乐的所有用户，就需要根据兴趣标签列进行搜索，找到标签中包含纯音乐的行，然后将此数据返回给应用。
一般最简单的做法是：首先按照上表的结构在数据库中建立一张用户兴趣表，然后执行数组查询语句，找到兴趣标签进行包含查找。但这么做会有一个问题，在数据量较大且标签值较多的场景下，不仅数据容量占用得更多，而且性能会极差。所以我们需要更换一种实现方案，将此表拆分为三张表，兴趣标签作为主键，包含此兴趣标签的用户作为 bitmap 存储。如下表所示：

版权所有：腾讯云计算（北京）有限责任公司

用户表：

用户 ID	用户名
1	张三
2	黎四
N	...

标签表：

标签 ID	用户名
1	古典
2	民歌
N	...

用户标签表：

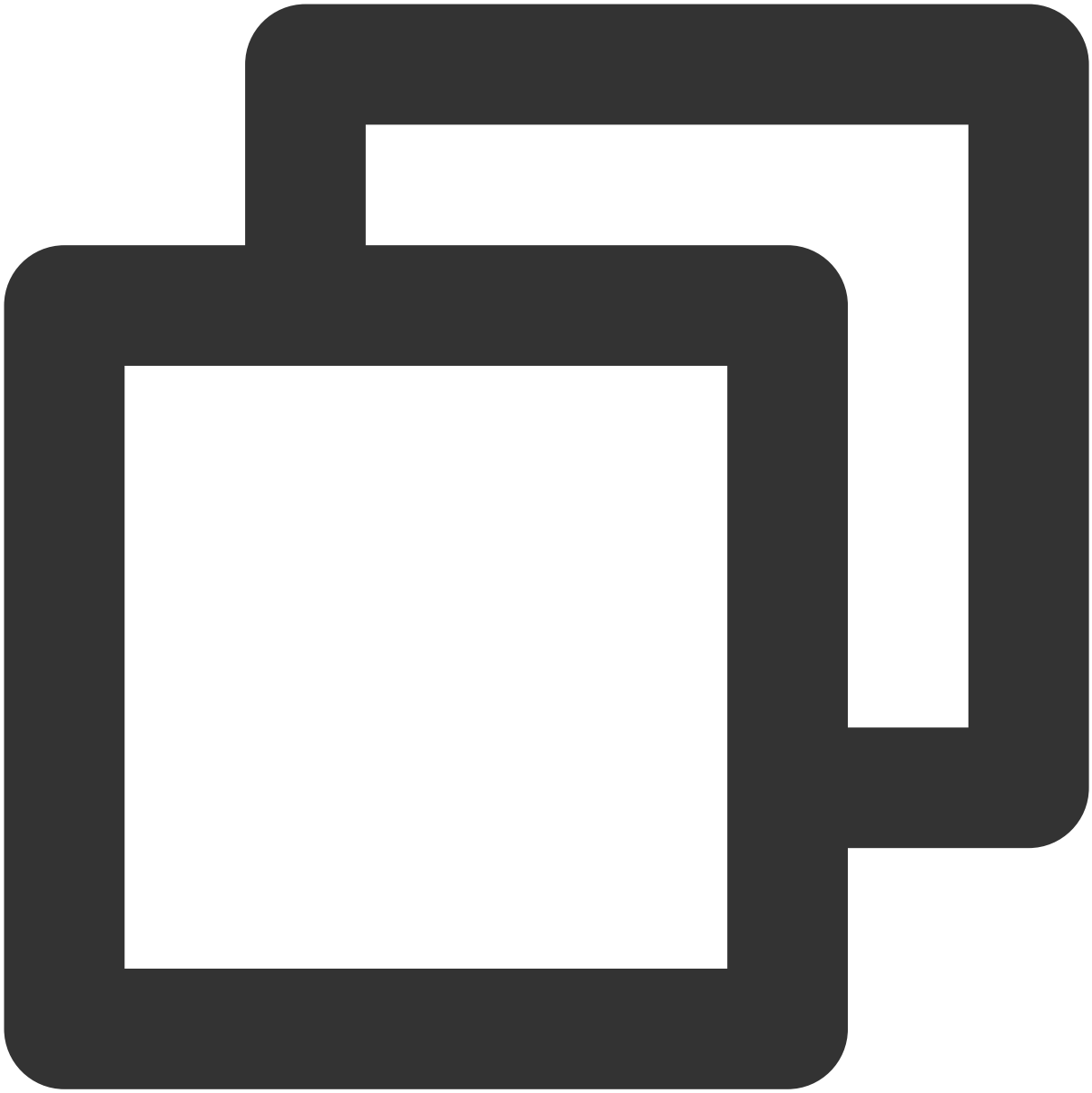
标签 ID	用户名
1	[1,3,7,123,423]
2	[5,31]
N	...

当需要查找同时喜欢听古典和民歌的用户时，直接在用户标签表对用户 ID 做 bitmap 查询即可，能够极大的提升性能并且容量占用可大幅降低。

传统方法与使用 roaringbitmap 方法查询性能对比

准备测试场景

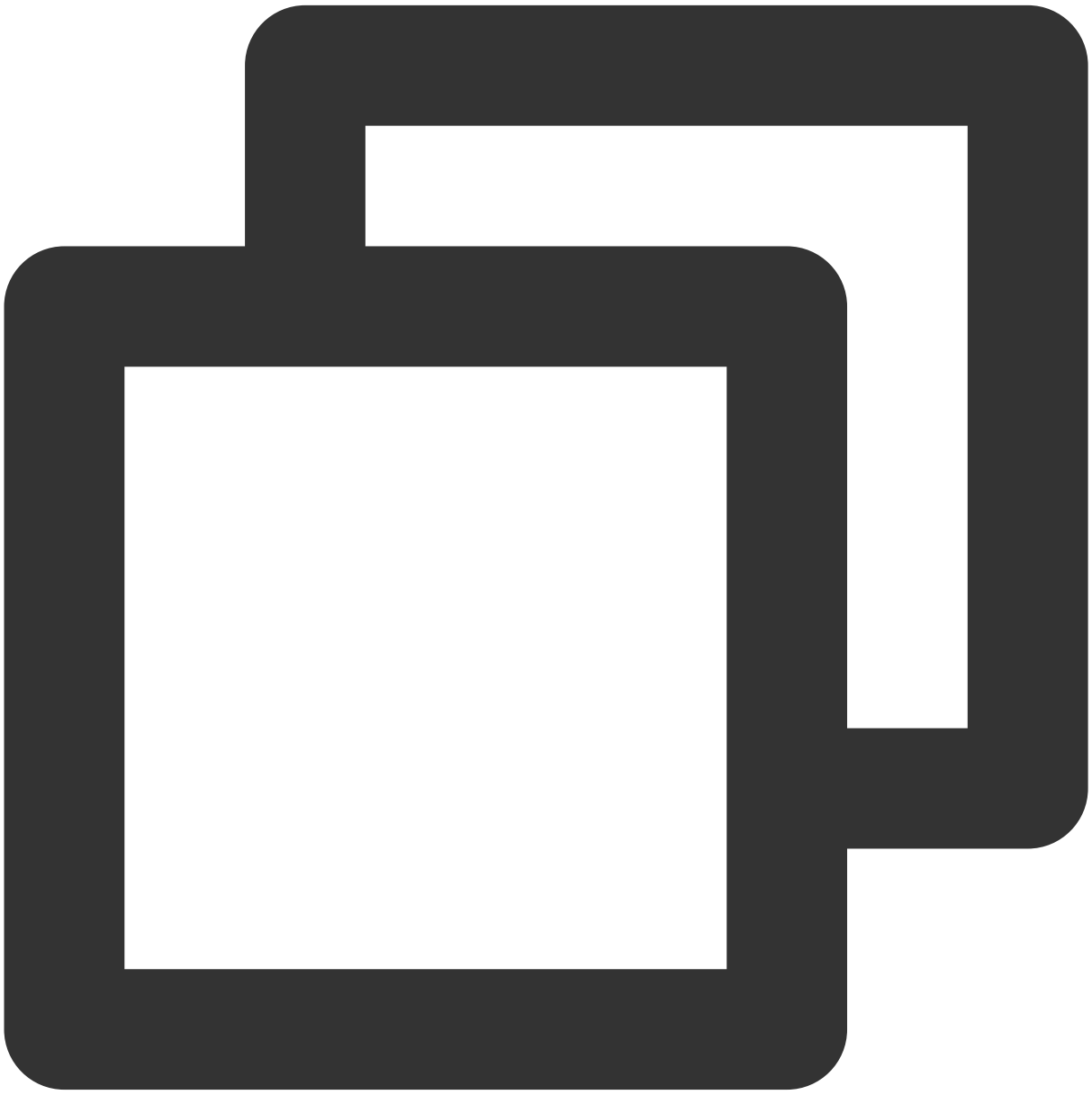
1. 创建一个随机字符的函数。



```
create or replace function random_string(length integer) returns text as
$$
declare
chars text[] := '{0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F,G,H,I,J,K,L,M,N,O,P,Q,R,S,T,U,V,W
result text := '';
i integer := 0;
length2 integer := (select trunc(random() * length + 1));
begin
if length2 < 0 then
raise exception 'Given length cannot be less than 0';
end if;
```

```
for i in 1..length2 loop
result := result || chars[1+random()*(array_length(chars, 1)-1)];
end loop;
return result;
end;
$$ language plpgsql;
```

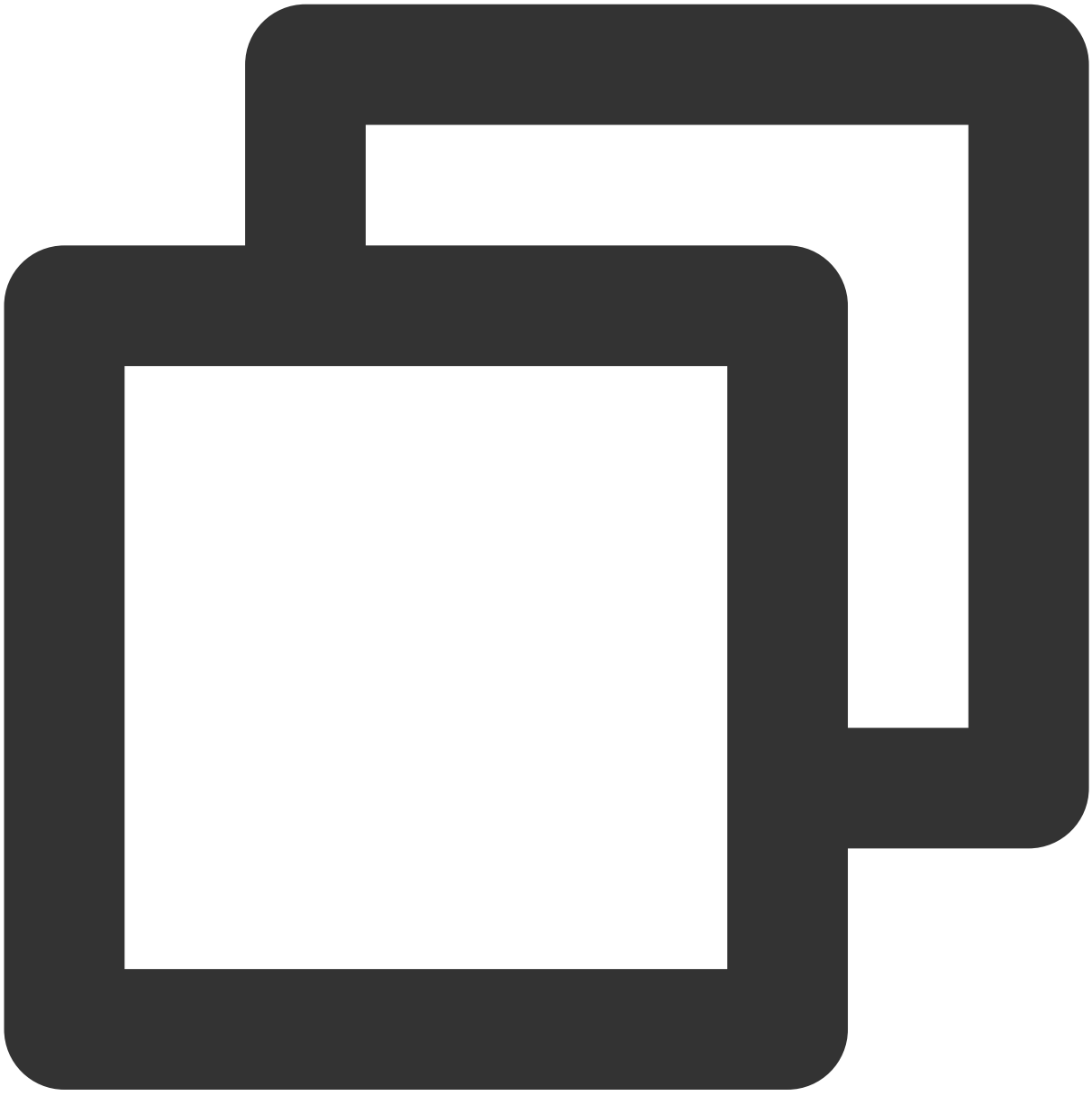
2. 创建一个生成随机整形数组的函数。



```
create or replace function random_int_array(int, int)
returns int[] language sql as
$$
```

```
select array_agg(round(random()* $1)::int)
from generate_series(1, $2)
$$;
```

3. 创建一个生成随机字符数组的函数。

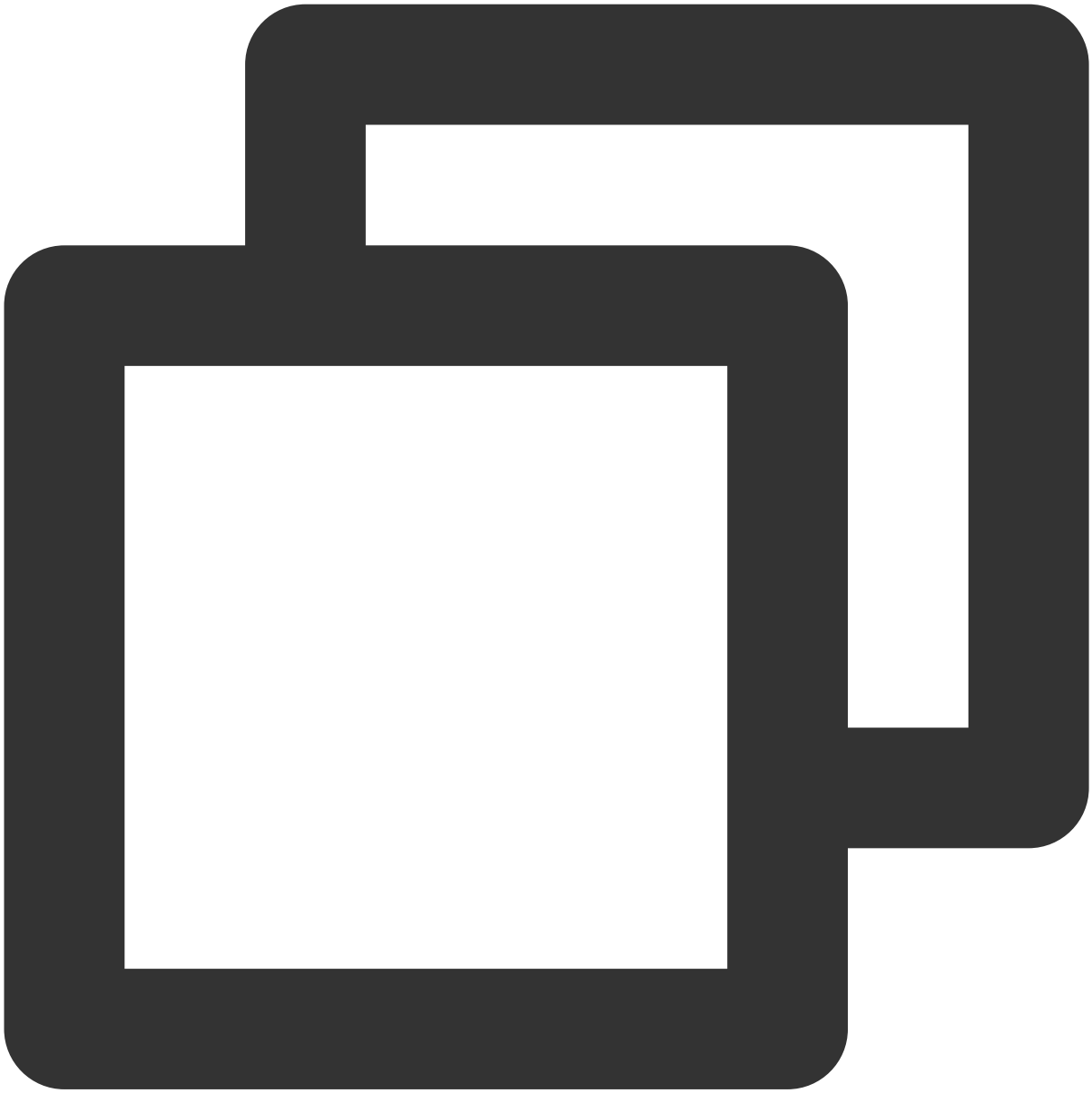


```
create or replace function random_string_array(int, int)
returns TEXT[] language sql as
$$
select array_agg(random_string($1)) from generate_series(1, $2);
$$;
```

方案1：传统方法

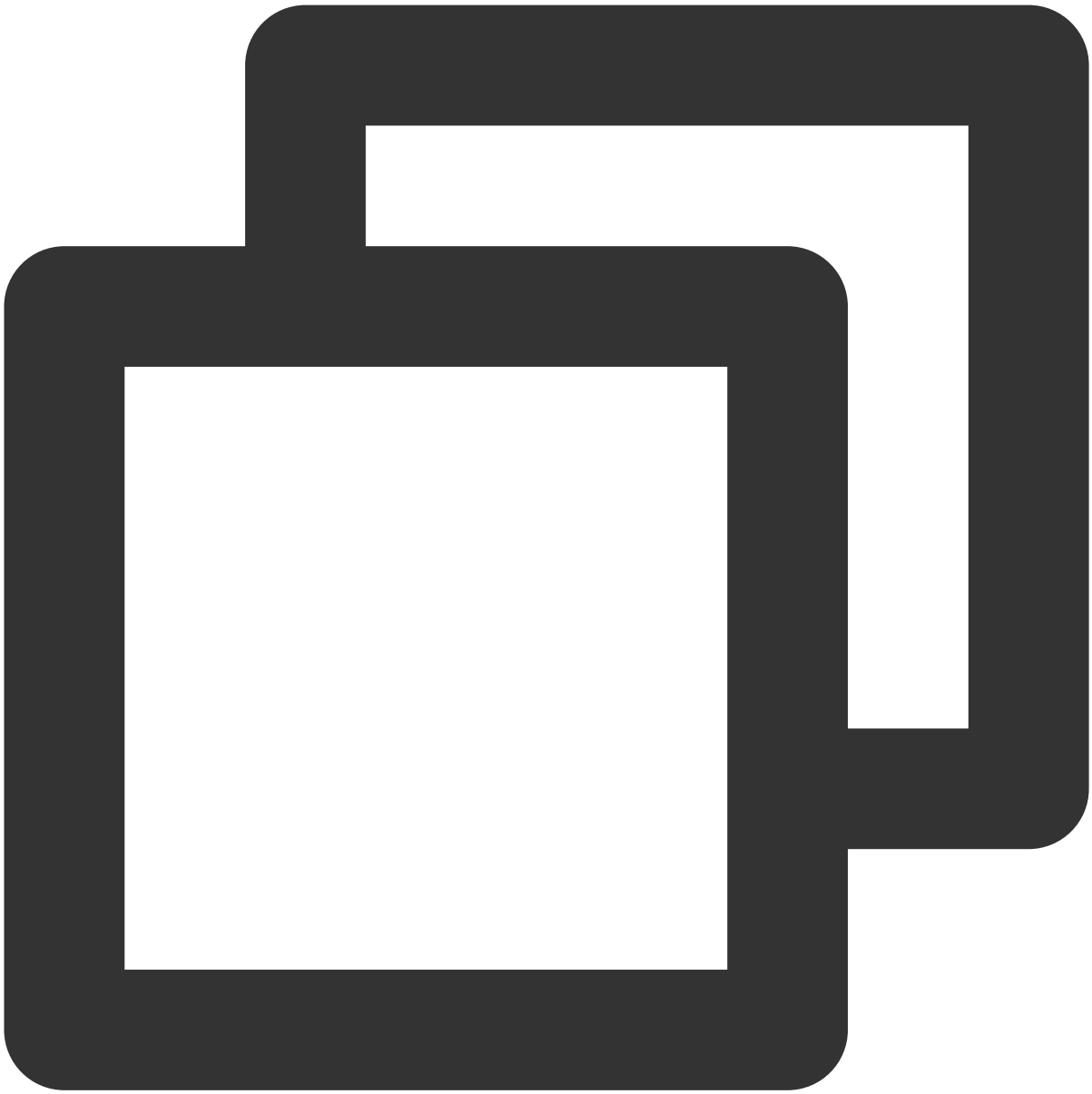
一张表解决一切。

1. 创建一个表包含所有数据。



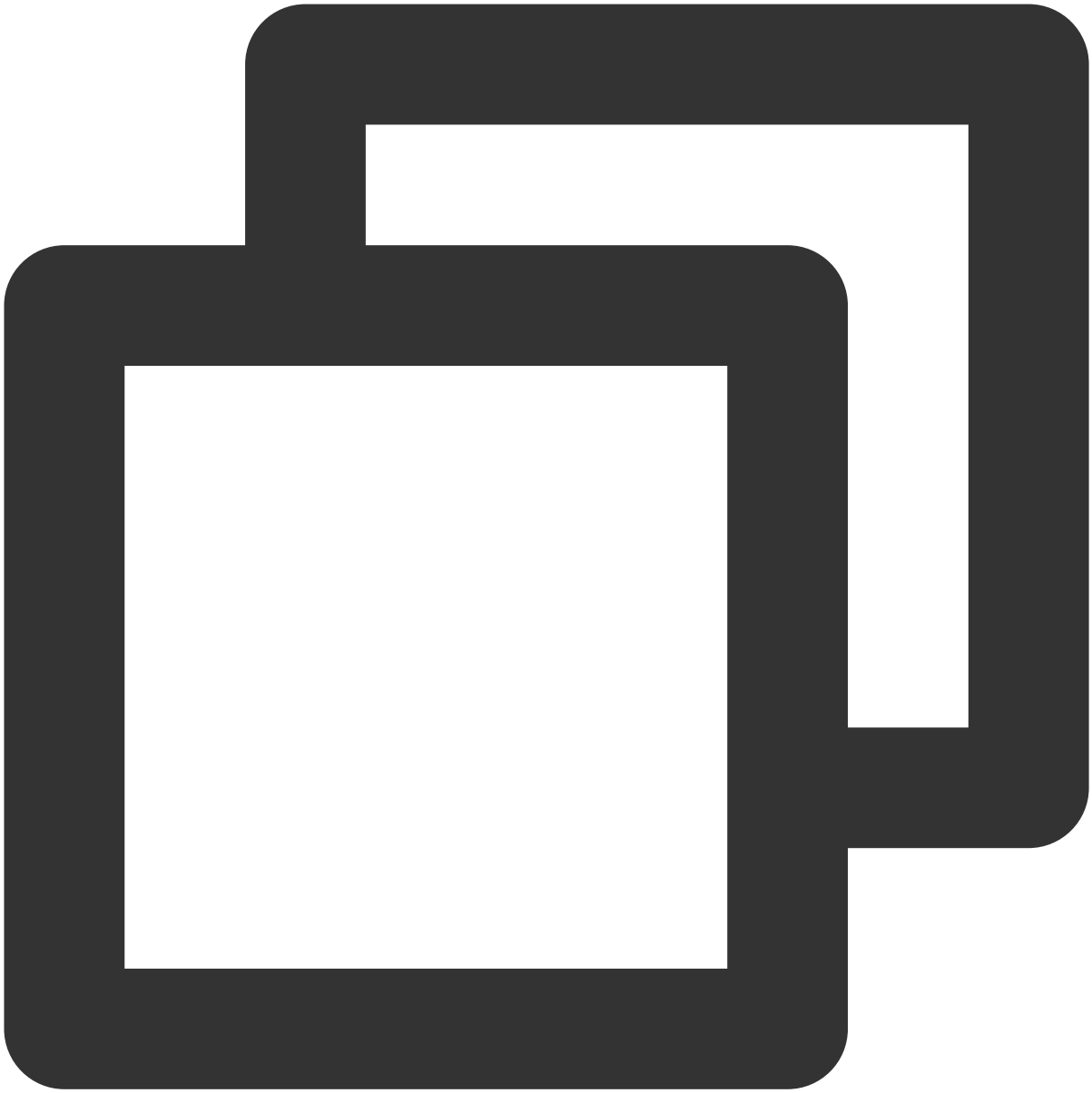
```
create table account(
uin bigint primary KEY,
name varchar,
tag TEXT []
);
```

2. 模拟插入1000W 个账号数据（需要使用到场景准备工作中的函数），并且创建 Gin 索引。



```
insert into account select generate_series(1,10000000), random_string(20),random_st
create index tag_inx on account USING GIN(tag);
```

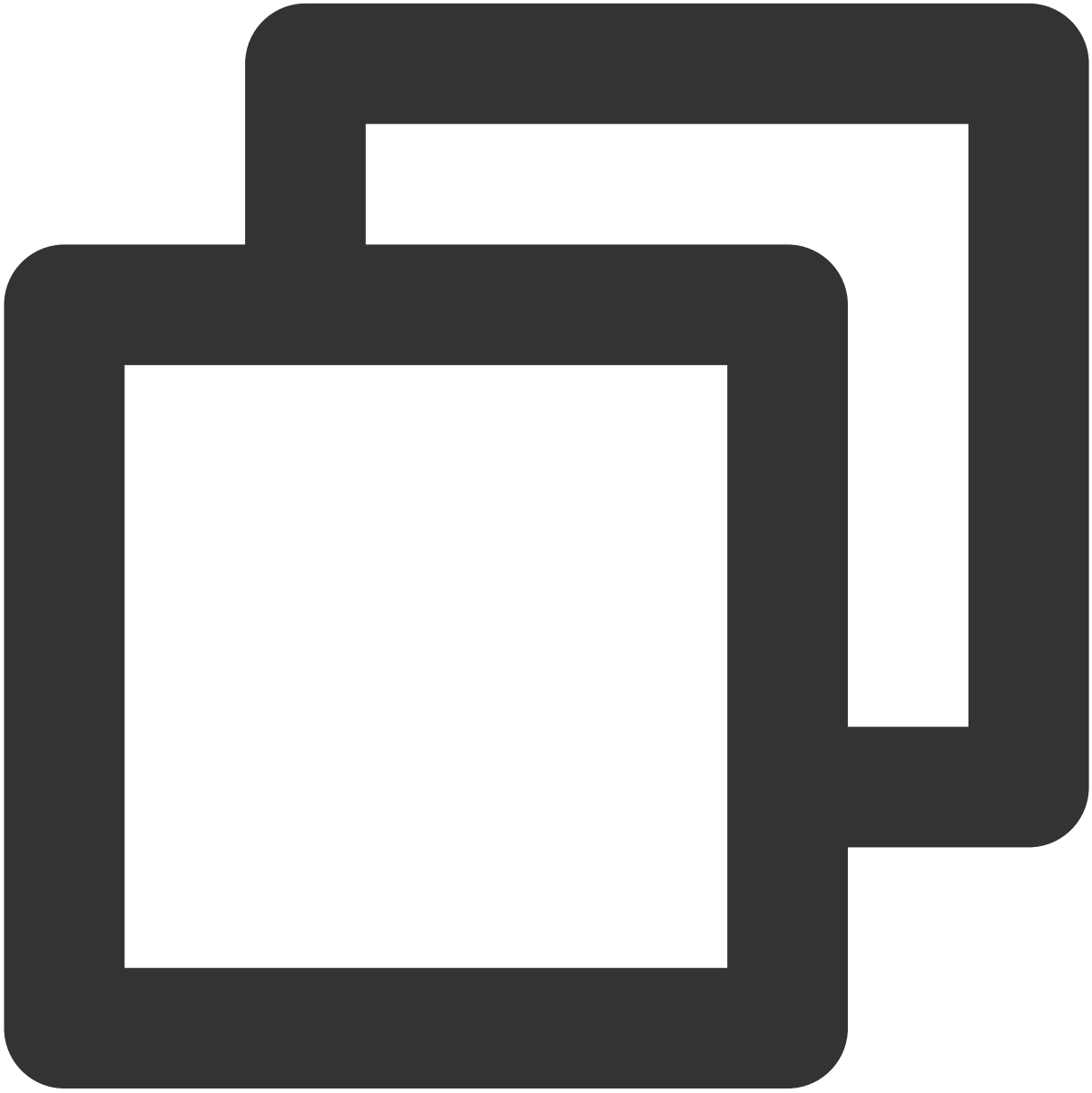
3. 执行查询，查找标签带 GN 和 o 的用户列表。



```
explain analyze select uin,name from account where tag @>ARRAY['GN','o'];

QUERY PLAN

------------------------------------------------------------------------------------------
-----------------
Bitmap Heap Scan on account (cost=52.81..466.86 rows=105 width=19) (actual time=4.2
184 loops=1)
Recheck Cond: (tag @> '{GN,o}'::text[])
Heap Blocks: exact=184
-> Bitmap Index Scan on tag_inx (cost=0.00..52.78 rows=105 width=0) (actual time=4.
```

```
ows=184 loops=1)
Index Cond: (tag @> '{GN,o}'::text[])
Planning Time: 0.108 ms
Execution Time: 4.528 ms
```

4. 执行查询，查找标签 lvXe 和 Zt 的人有 xx 个（第一次查询会较慢）。



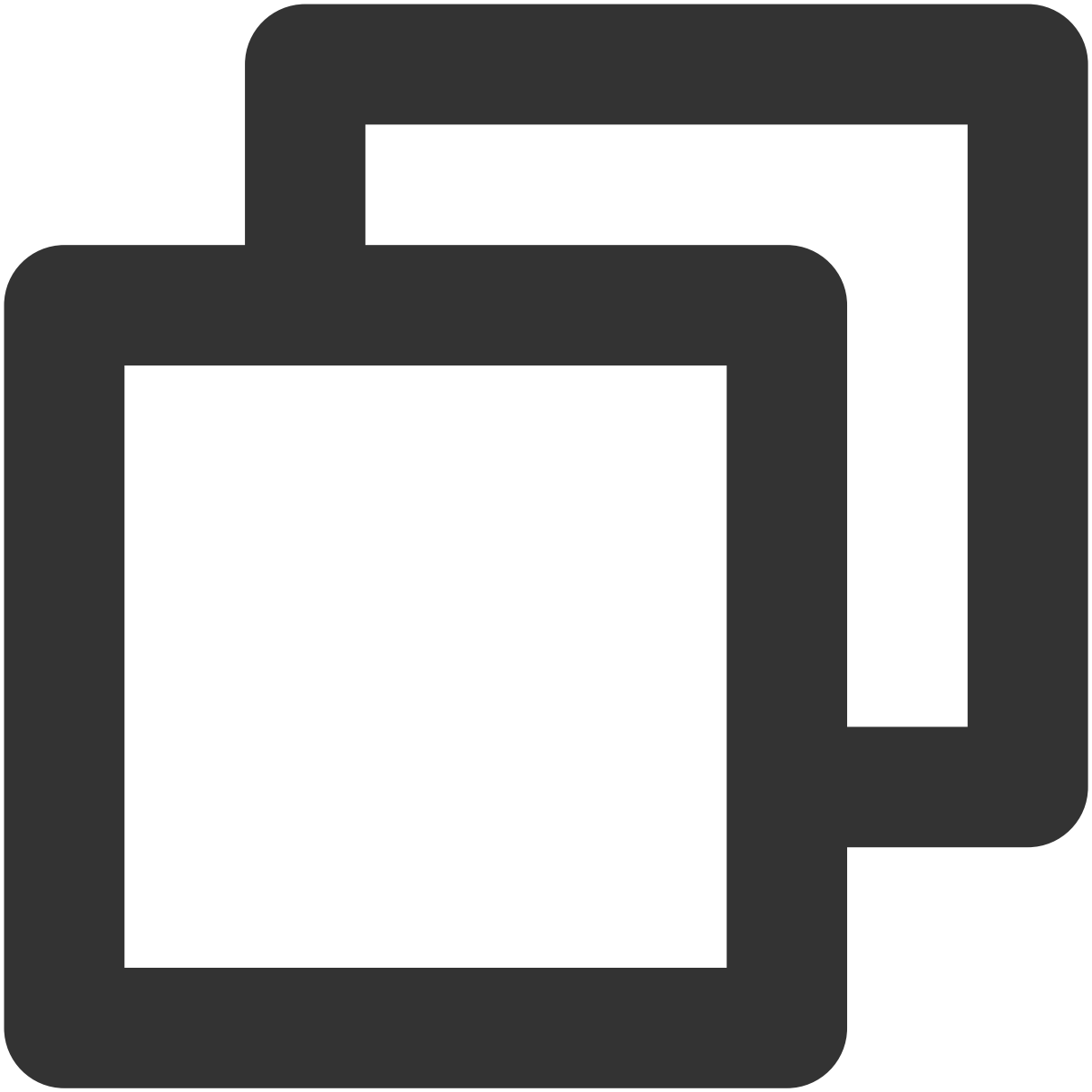
```
explain analyze select count(uin) from account where tag && ARRAY['lvXe','Zt'];

------------------------------------------------------------------------------------
--------------------------
Aggregate (cost=21816.39..21816.40 rows=1 width=8) (actual time=8.236..8.238 rows=1
```

```
-> Bitmap Heap Scan on account (cost=109.08..21800.56 rows=6332 width=8) (actual ti
901 rows=5390 loops=1)
Recheck Cond: (tag && '{lvXe,Zt}'::text[])
Heap Blocks: exact=5327
-> Bitmap Index Scan on tag_inx (cost=0.00..107.49 rows=6332 width=0) (actual time=
.0.962 rows=5390 loops=1)
Index Cond: (tag && '{lvXe,Zt}'::text[])
Planning Time: 0.110 ms
Execution Time: 8.270 ms
```

方案2：优化方案

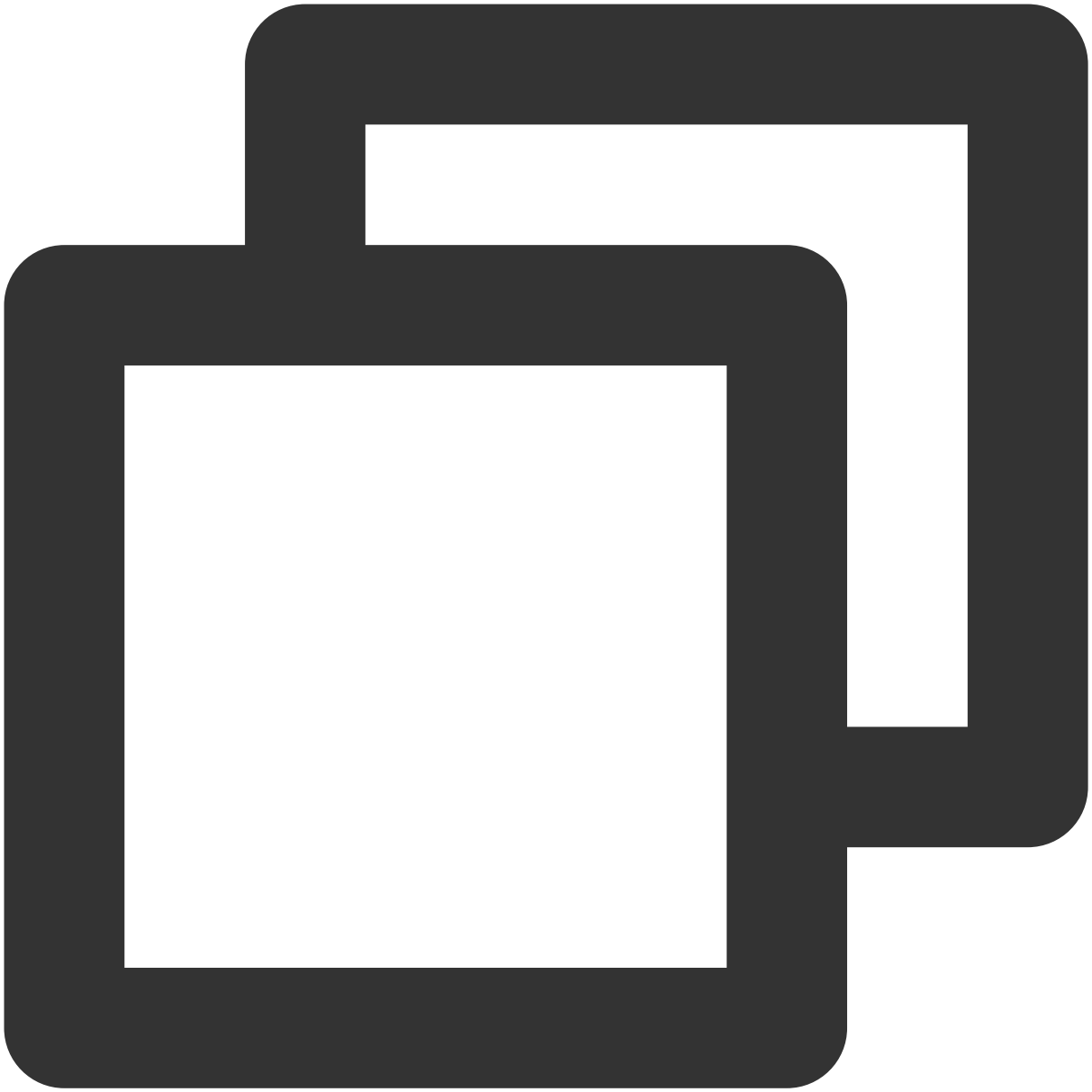
为了降低查询中标签字段的类型导致的性能减低，所以将上面表中的真实 tag 修改为 tagid。

1. 引入一个新的标签字典表。



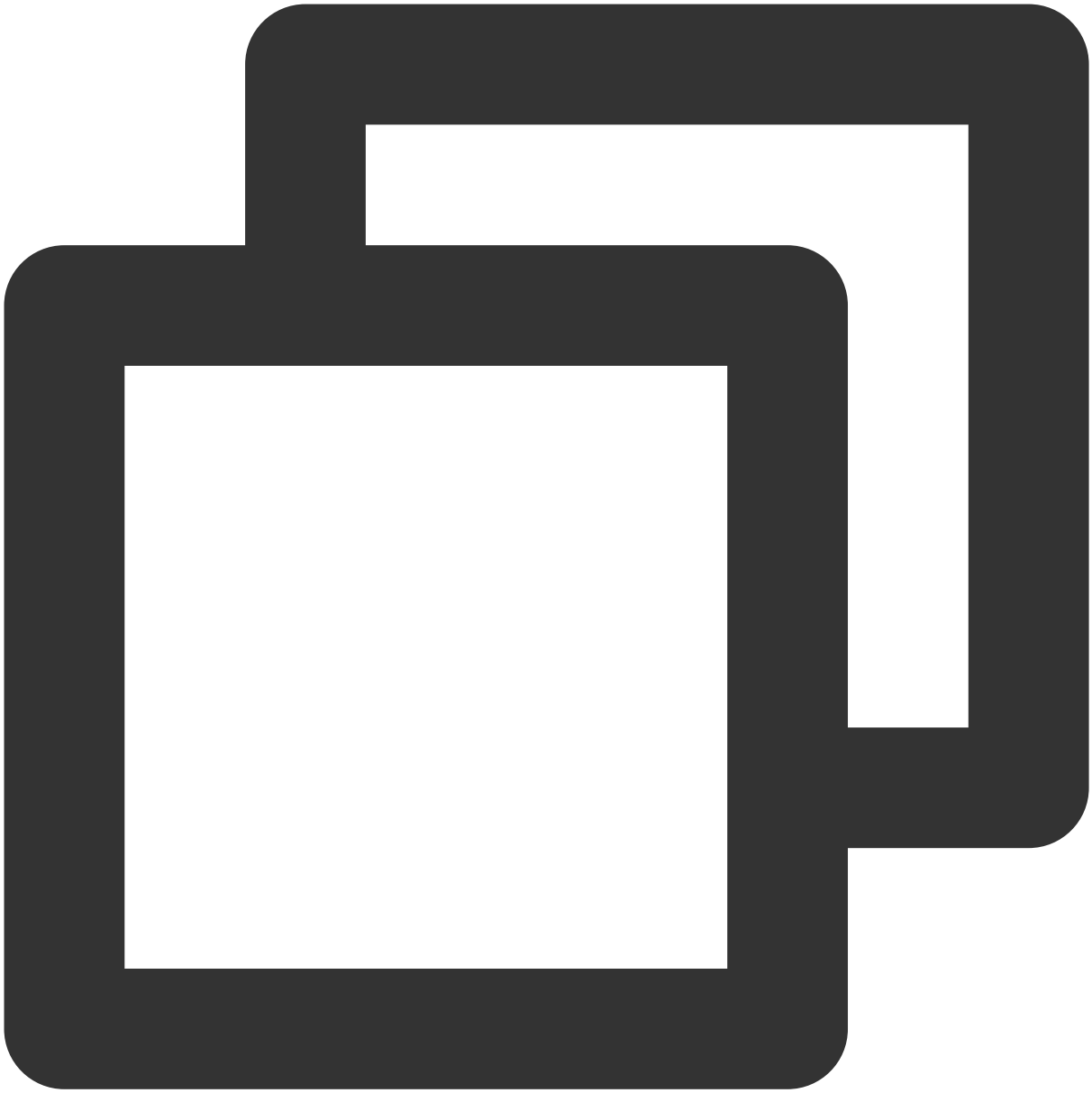
```
create table tag_dict (
tagid int primary key,
taginfo text
);
```

2. 假设一共有10W 种字典类型。



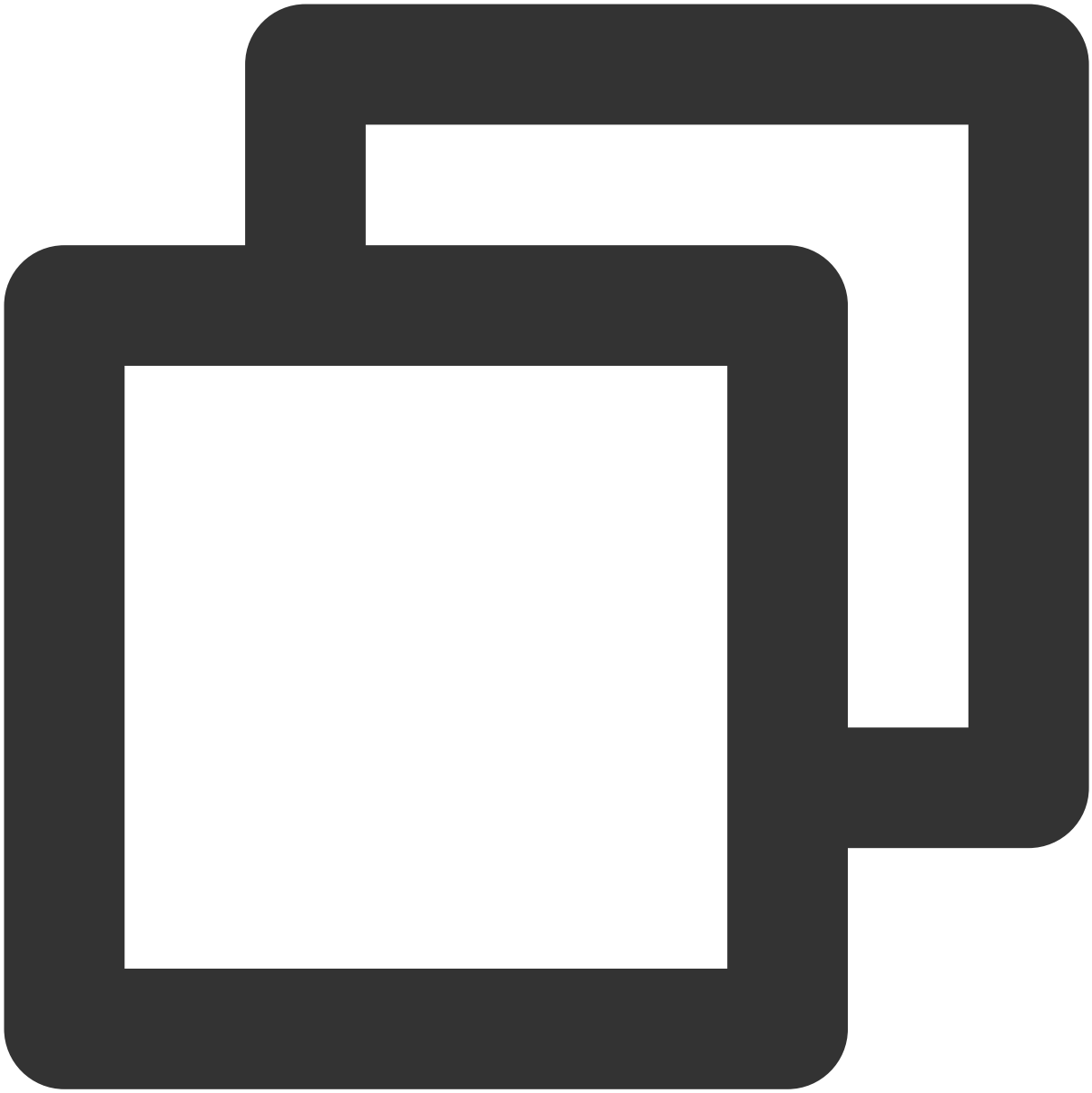
```
insert into tag_dict select generate_series(1,100000), md5(random()::text);
```

3. 创建一个新表用来存储用户和标签信息。



```
create table account1(
uin bigint primary KEY,
name varchar,
tag INT []
);
```

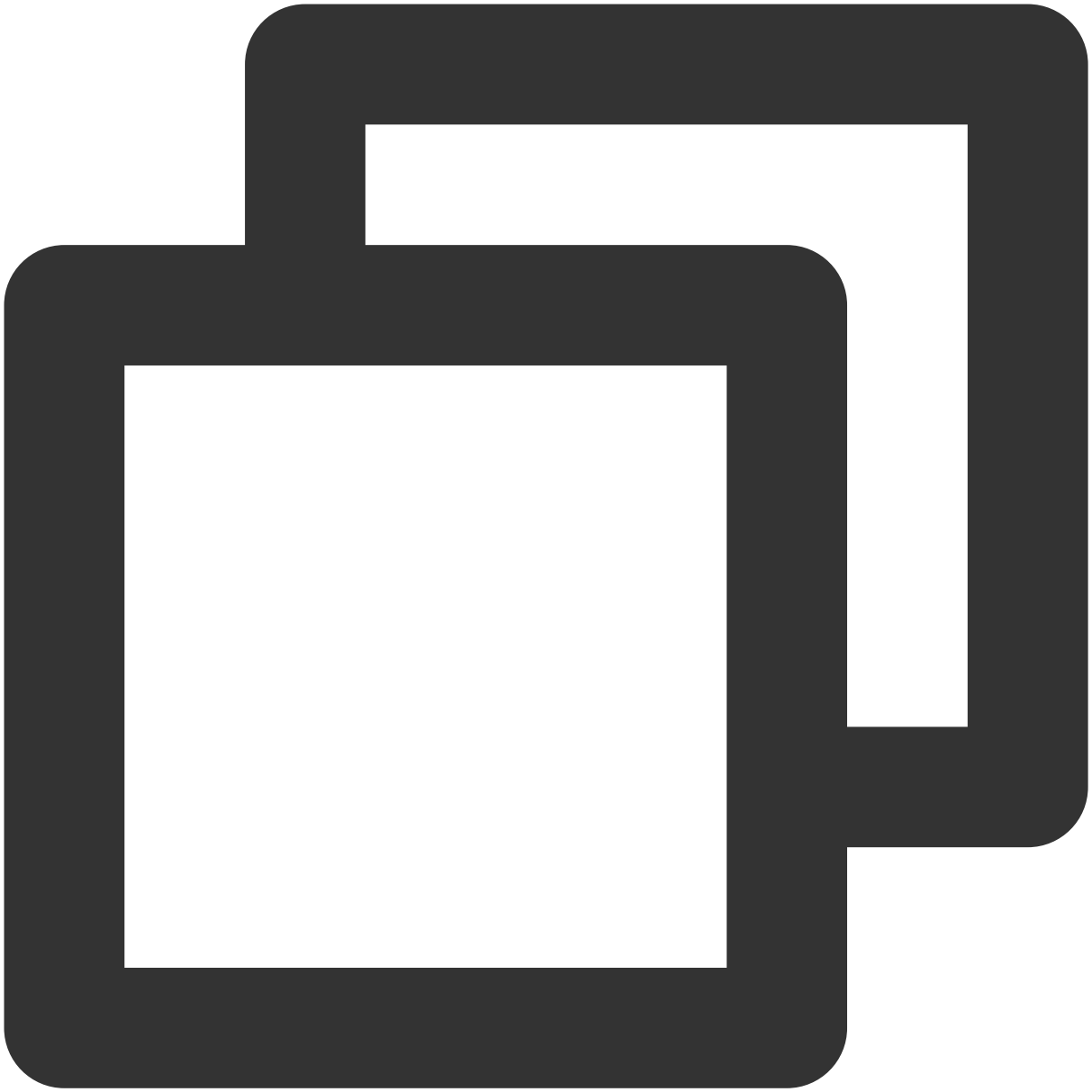
4. 插入1000W 个账号数据。



```
insert into account1 select generate_series(1,10000000), random_string(20),random_i
```

5. 查找同时有标签 ID 为100和5711的用户列表。

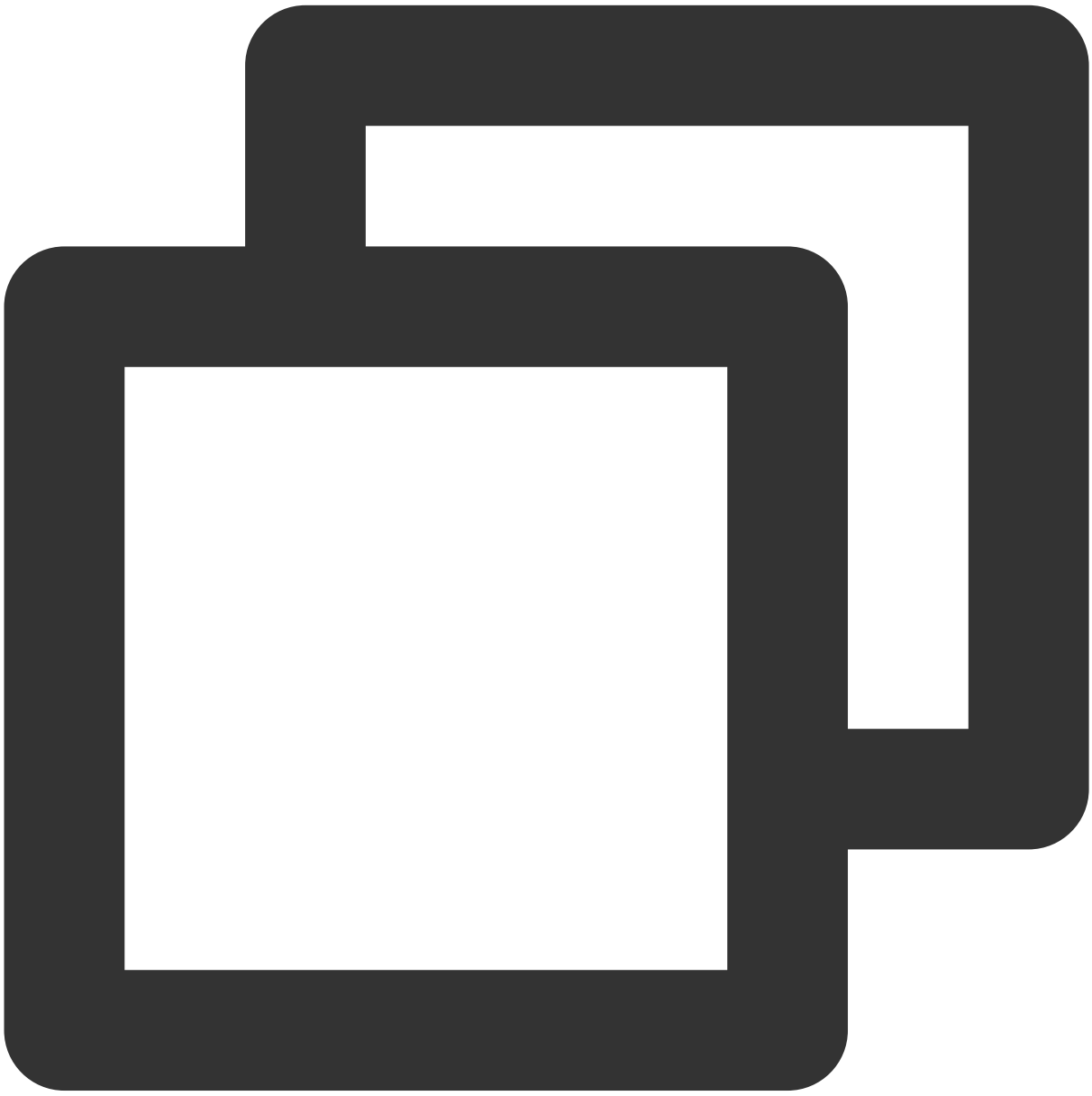
索引前：



```
test=> explain analyze select uin,name from account1 where tag @> ARRAY[100,5711];
QUERY PLAN
------------------------------------------------------------------------------------
Gather (cost=1000.00..191007.68 rows=250 width=19) (actual time=982.585..1000.806 r
Workers Planned: 2
Workers Launched: 2
-> Parallel Seq Scan on account1 (cost=0.00..189982.68 rows=104 width=19) (actual t
Filter: (tag @> '{100,5711}'::integer[])
Rows Removed by Filter: 3333333
Planning Time: 0.205 ms
JIT:
```

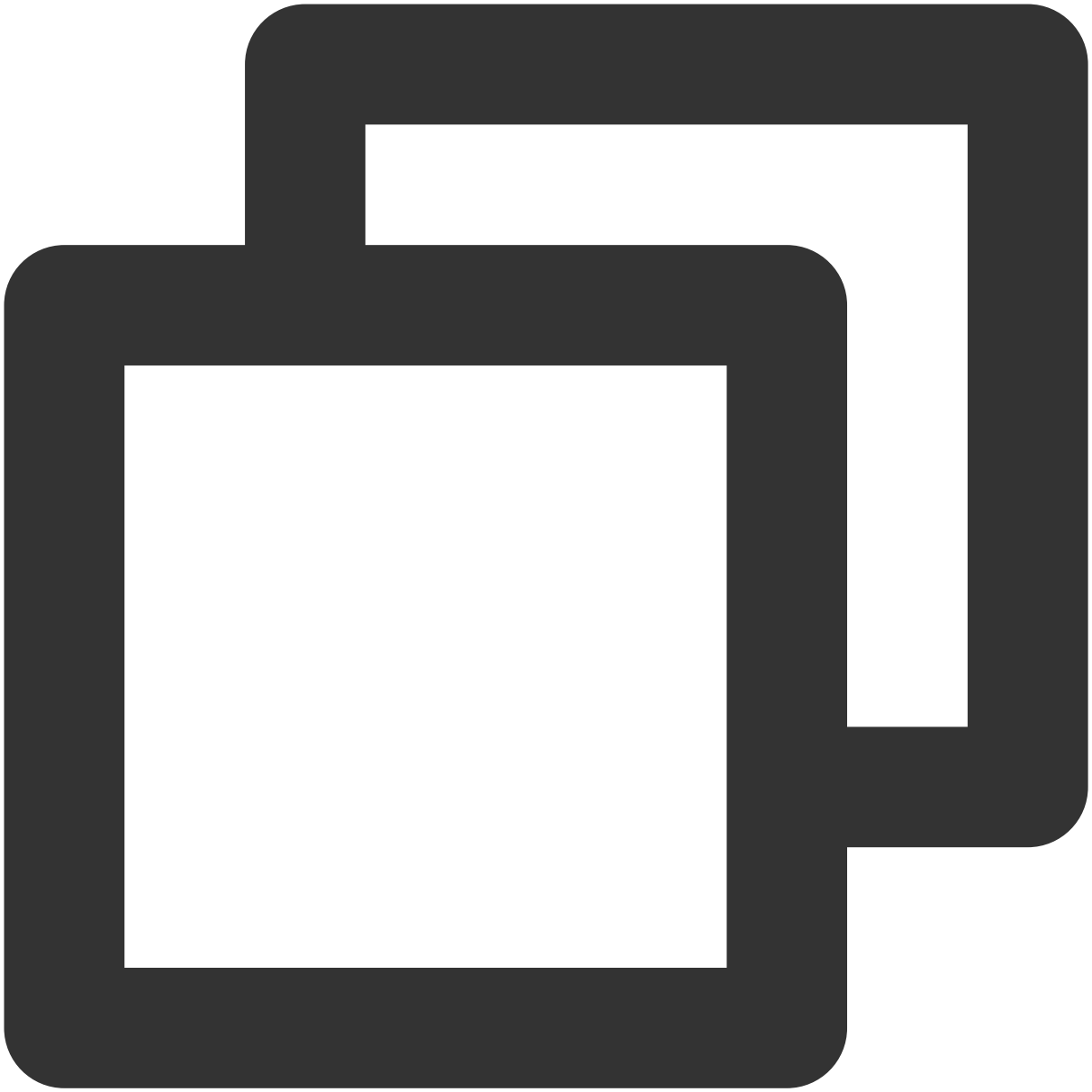
```
Functions: 12
Options: Inlining false, Optimization false, Expressions true, Deforming true
Timing: Generation 2.280 ms, Inlining 0.000 ms, Optimization 1.176 ms, Emission 14.
Execution Time: 1001.574 ms
(12 rows)
```

加索引：



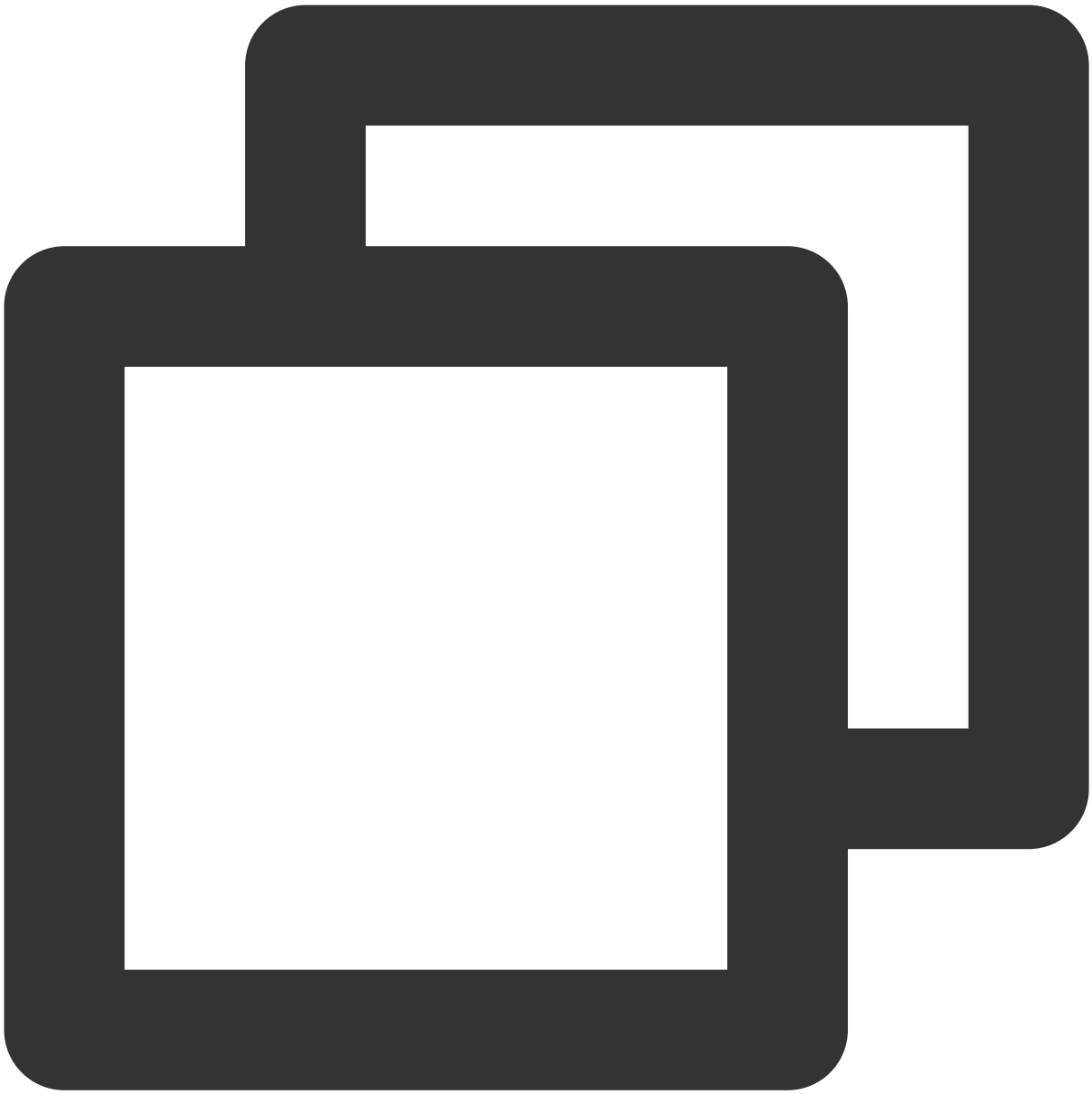
```
create index tag_inx_2 on account1 USING GIN(tag);
```

索引后：



```
test=> explain analyze select uin,name from account1 where tag @> ARRAY[100,5711];
QUERY PLAN
------------------------------------------------------------------------------------
Bitmap Heap Scan on account1 (cost=49.94..1021.13 rows=250 width=19) (actual time=0
Recheck Cond: (tag @> '{100,5711}'::integer[])
-> Bitmap Index Scan on tag_inx_2 (cost=0.00..49.87 rows=250 width=0) (actual time=
Index Cond: (tag @> '{100,5711}'::integer[])
Planning Time: 0.410 ms
Execution Time: 0.171 ms
(6 rows)
```

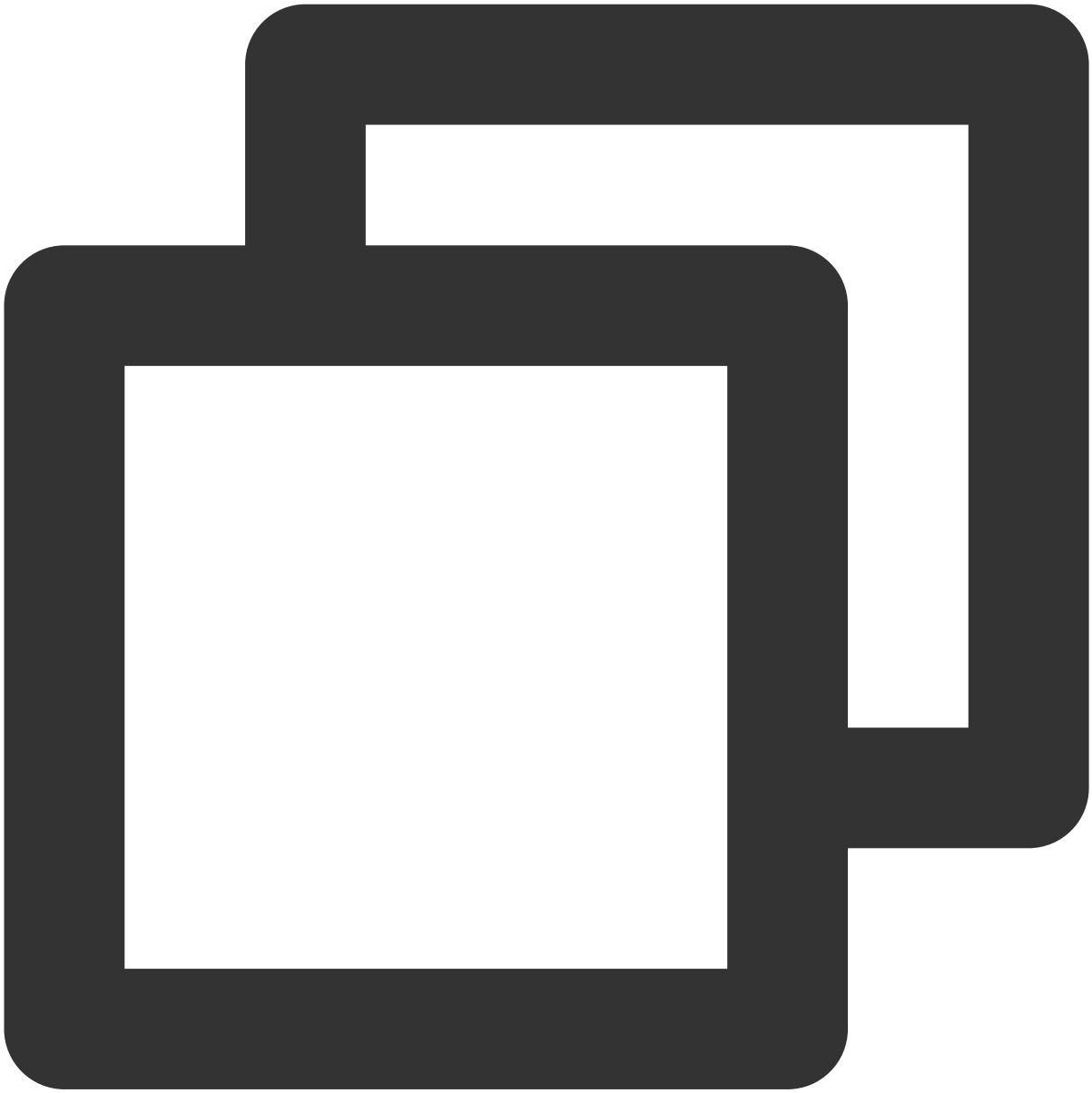
6. 查找同时有标签 ID 为61568，97350的用户列表。



```
test=> explain analyze select uin,name from account1 where tag @> ARRAY[61568,97350
QUERY PLAN
------------------------------------------------------------------------------------
Bitmap Heap Scan on account1 (cost=49.94..1021.13 rows=250 width=19) (actual time=0
Recheck Cond: (tag @> '{61568,97350}'::integer[])
Heap Blocks: exact=1
-> Bitmap Index Scan on tag_inx_2 (cost=0.00..49.87 rows=250 width=0) (actual time=
Index Cond: (tag @> '{61568,97350}'::integer[])
Planning Time: 0.071 ms
```

```
Execution Time: 0.151 ms
(7 rows)
```

7. 查找与 xx 有共同爱好（标签100和5711）的人有 xx 个。

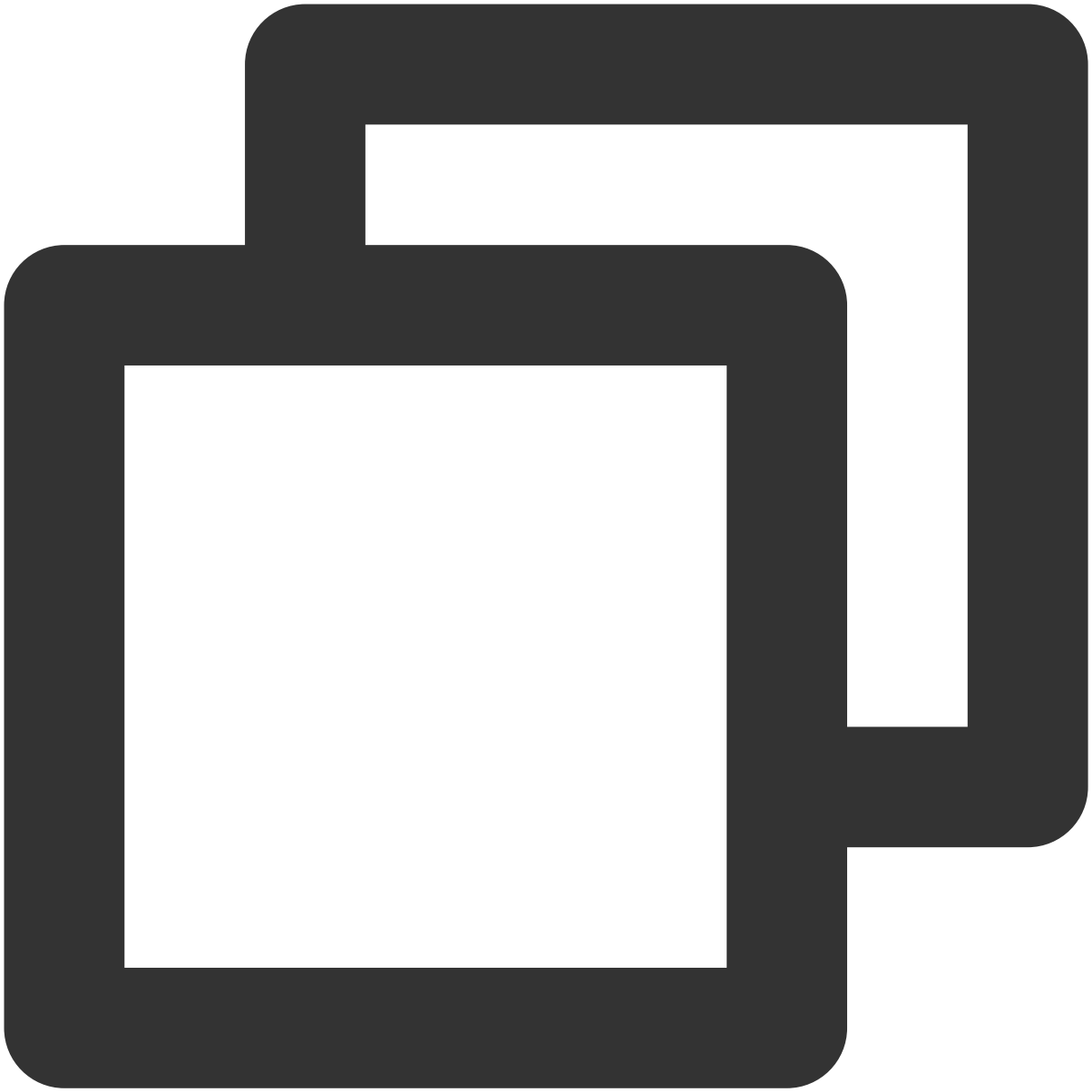


```
test=> explain analyze select count(uin) from account1 where tag && ARRAY[61568,973
QUERY PLAN

------------------------------------------------------------------------------------
------
Gather (cost=1961.06..173801.15 rows=99750 width=19) (actual time=5.020..28.885 row
Workers Planned: 2
```

```
Workers Launched: 2
-> Parallel Bitmap Heap Scan on account1 (cost=961.06..162826.15 rows=41562 width=1
ps=3)
Recheck Cond: (tag && '{61568,97350}'::integer[])
Heap Blocks: exact=2053
-> Bitmap Index Scan on tag_inx_2 (cost=0.00..936.12 rows=99750 width=0) (actual ti
Index Cond: (tag && '{61568,97350}'::integer[])
Planning Time: 0.082 ms
JIT:
Functions: 12
Options: Inlining false, Optimization false, Expressions true, Deforming true
Timing: Generation 2.078 ms, Inlining 0.000 ms, Optimization 0.270 ms, Emission 3.4
Execution Time: 29.725 ms
(14 rows)
```

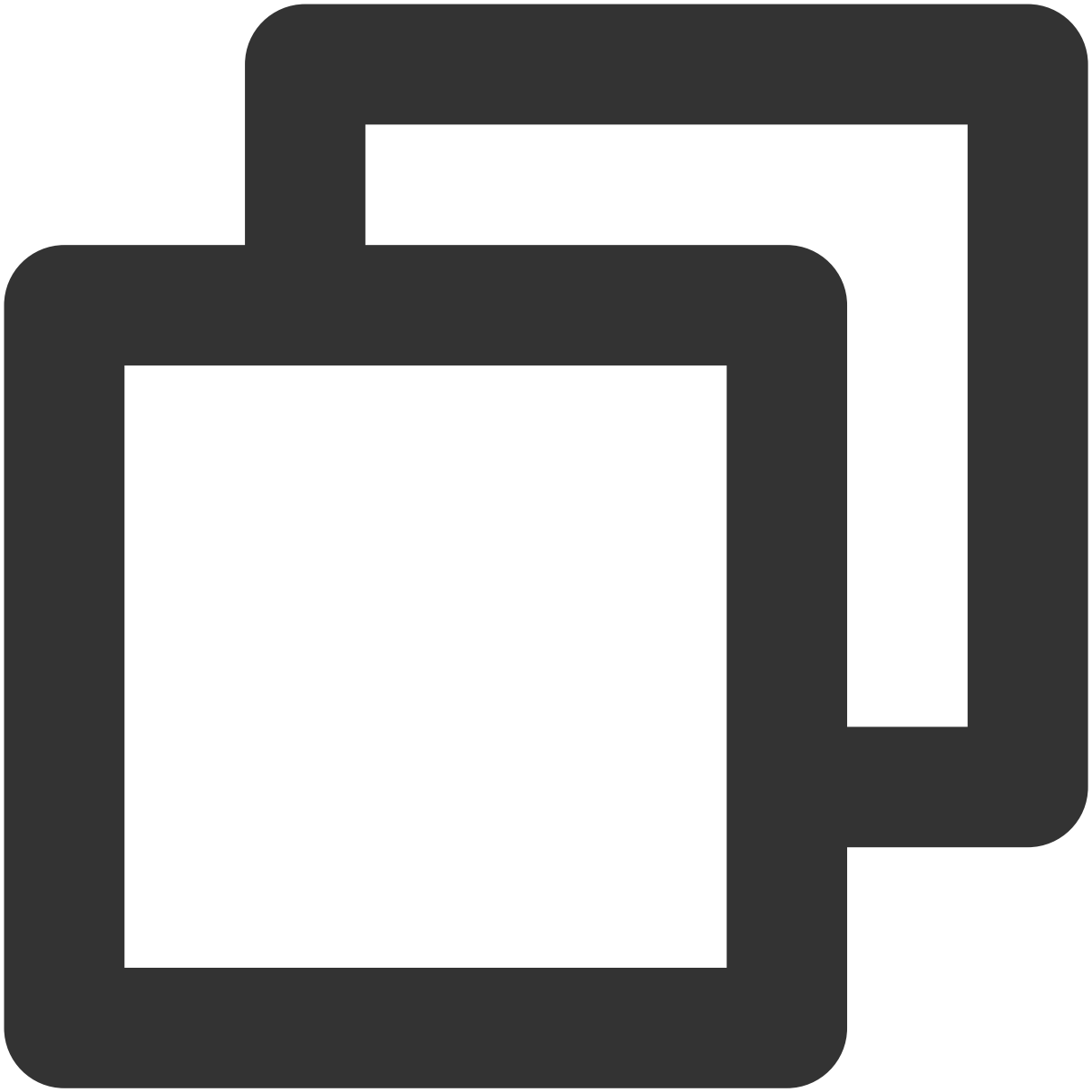
方案3：roaringbitmap

1. 首先需要创建插件，云数据库 PostgreSQL 天然集成了此插件，无需关注编译等操作，直接进入数据库中创建即可。



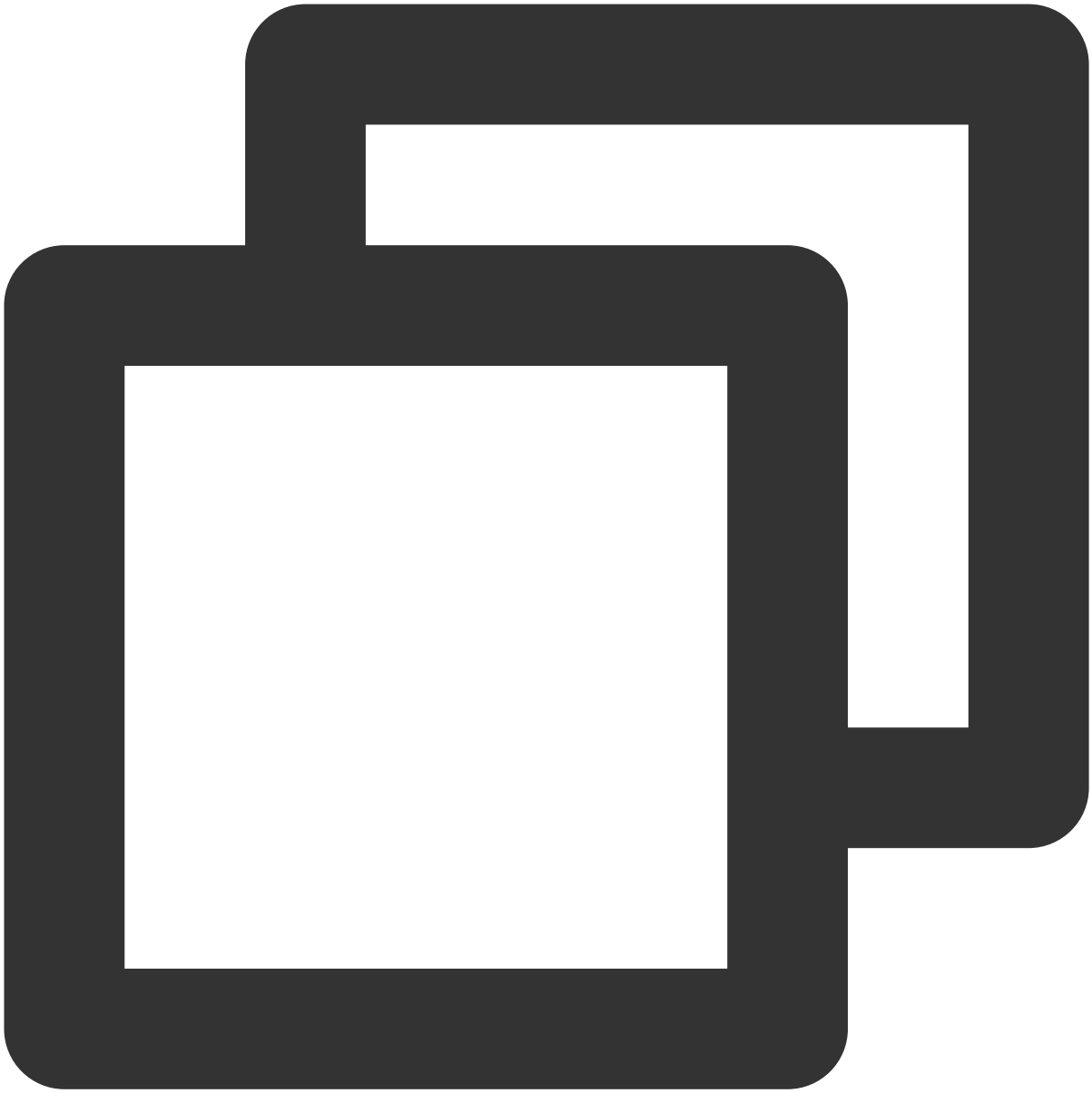
```
create extension roaringbitmap;
```

2. 创建标签用户对应表。



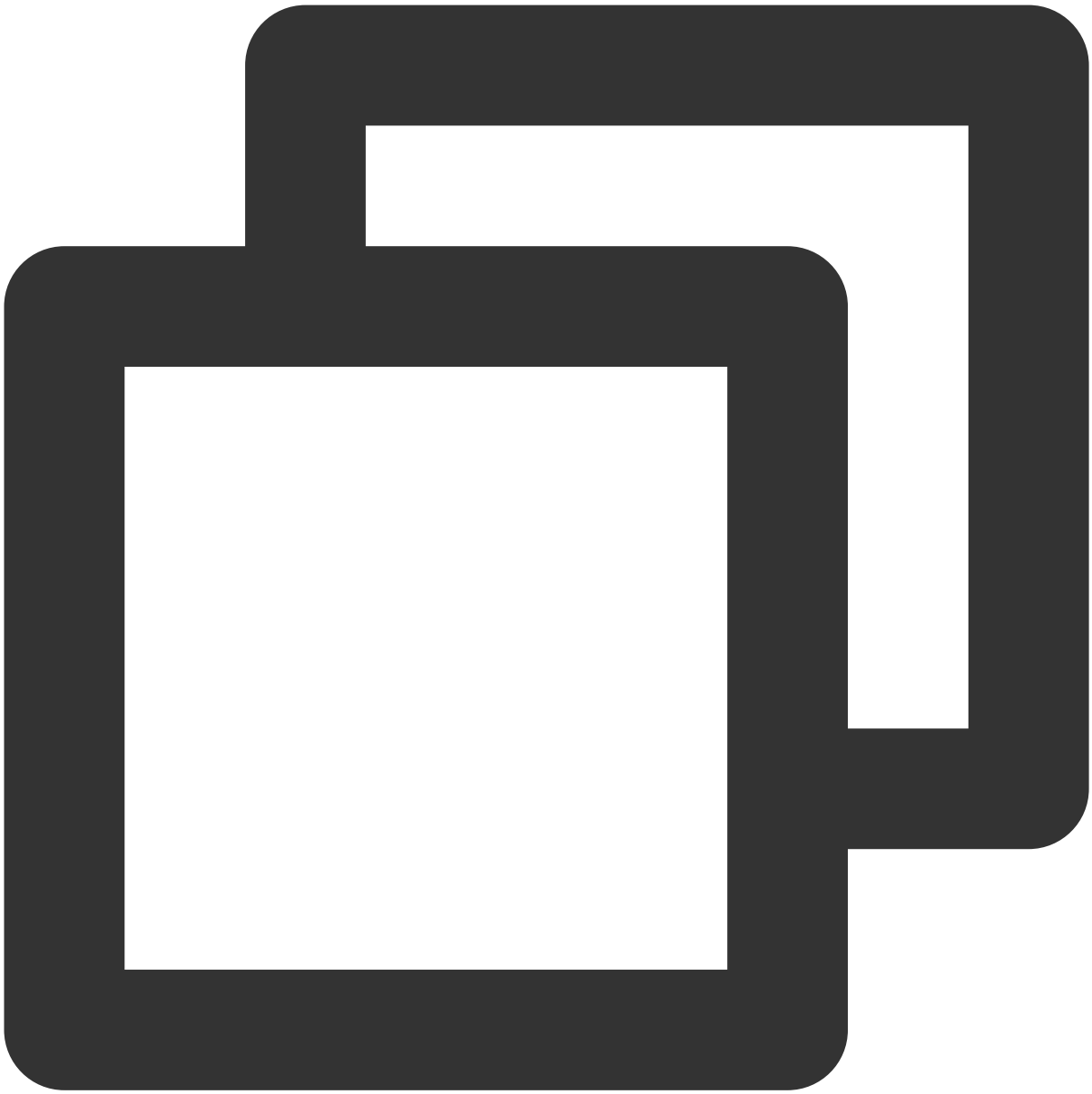
```
create table tag_uin_list(
tagid int primary key,
uin_offset int,
uinbits roaringbitmap
);
```

3. 根据之前的标签表插入10W条标签以及标签对应的用户数据。



```
insert into tag_uin_list
select tagid, uin_offset, rb_build_agg(uin::int) as uinbits from
(
select
unnest(tag) as tagid,
(uin / (2^31)::int8) as uin_offset,
mod(uin, (2^31)::int8) as uin
from account1
) t
group by tagid, uin_offset;
```

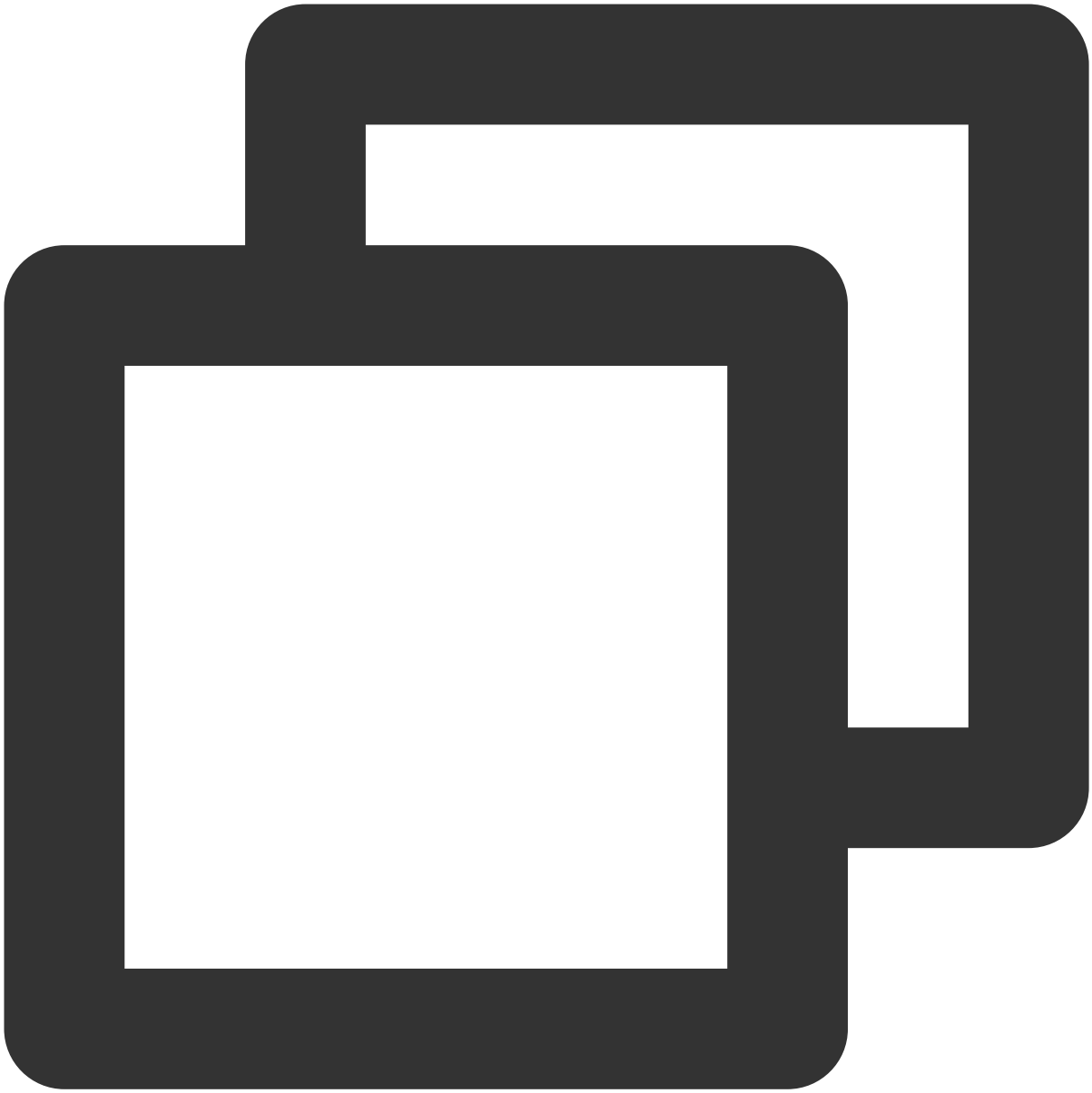
4. 查询标签有1，3，10，200的用户个数。



```
explain analyze select sum(ub) from
(
select uin_offset,rb_or_cardinality_agg(uinbits) as ub
from tag_uin_list
where tagid in (1,3,10,200)
group by uin_offset
) t;
```

```
QUERY PLAN

------------------------------------------------------------------------------------------
------------
Aggregate (cost=32.47..32.48 rows=1 width=32) (actual time=0.964..0.966 rows=1 loop
-> GroupAggregate (cost=32.42..32.46 rows=1 width=12) (actual time=0.955..0.956 row
Group Key: tag_uin_list.uin_offset
-> Sort (cost=32.42..32.43 rows=4 width=22) (actual time=0.107..0.109 rows=4 loops=
Sort Key: tag_uin_list.uin_offset
Sort Method: quicksort Memory: 25kB
-> Bitmap Heap Scan on tag_uin_list (cost=17.20..32.38 rows=4 width=22) (actual tim
)
Recheck Cond: (tagid = ANY ('{1,3,10,200}'::integer[]))
Heap Blocks: exact=4
-> Bitmap Index Scan on tag_uin_list_pkey (cost=0.00..17.20 rows=4 width=0) (actual
=4 loops=1)
Index Cond: (tagid = ANY ('{1,3,10,200}'::integer[]))
Planning Time: 0.289 ms
Execution Time: 1.083 ms
(13 rows)
```

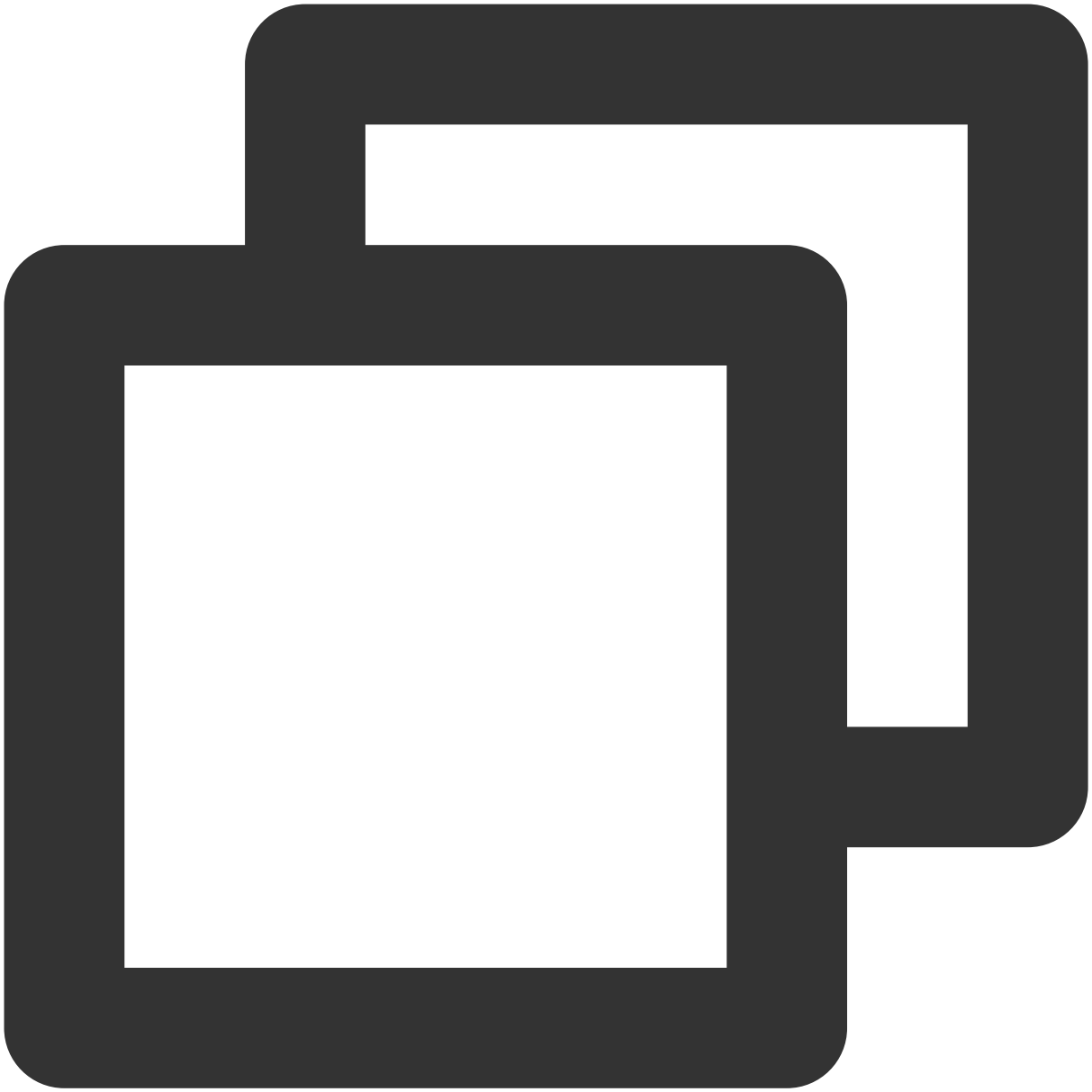
5. 查看标签有1，3，10，200的用户列表。



```
explain analyze select uin_offset,rb_or_agg(uinbits) as ub
from tag_uin_list
where tagid in (1,3,10,200)
group by uin_offset;
QUERY PLAN

------------------------------------------------------------------------------------
------
GroupAggregate (cost=32.42..32.46 rows=1 width=36) (actual time=0.246..0.246 rows=1
Group Key: uin_offset
-> Sort (cost=32.42..32.43 rows=4 width=22) (actual time=0.043..0.045 rows=4 loops=
```

```
Sort Key: uin_offset
Sort Method: quicksort Memory: 25kB
-> Bitmap Heap Scan on tag_uin_list (cost=17.20..32.38 rows=4 width=22) (actual tim
Recheck Cond: (tagid = ANY ('{1,3,10,200}'::integer[]))
Heap Blocks: exact=4
-> Bitmap Index Scan on tag_uin_list_pkey (cost=0.00..17.20 rows=4 width=0) (actual
Index Cond: (tagid = ANY ('{1,3,10,200}'::integer[]))
Planning Time: 0.119 ms
Execution Time: 0.310 ms
(12 rows)
```

查看索引以及表占用大小



```
test=> select relname, pg_size_pretty(pg_relation_size(relid)) from pg_stat_user_ta
   relname    | pg_size_pretty
--------------+----------------
 account      | 1545 MB
 account1     | 1077 MB
 t_user       | 651 MB
 tag_dict     | 6672 kB
 tag_uin_list | 5888 kB
(5 rows)

test=> select indexrelname, pg_size_pretty(pg_relation_size(relid)) from pg_stat_us
```

```
    indexrelname     | pg_size_pretty
--------------------+----------------
 tag_inx             | 1545 MB
 account_pkey        | 1545 MB
 tag_inx_2           | 1077 MB
 account1_pkey       | 1077 MB
 t_user_pkey         | 651 MB
 tag_dict_pkey       | 6672 kB
 tag_uin_list_pkey | 5888 kB
(7 rows)
```

结论

不同方案的查询性能对比如下表：

查询项	方案1	方案2	roaringbitmap 方案
查询包含指定标签的用户列表	4.528ms	0.151ms	0.310ms
查询具备共同标签的用户个数	8.27ms	29.725ms	1.083ms
数据容量统计	4635MB	3244.344MB	1237.12MB

基于上述三种方案可以明显看到，优化后效果非常明显，无论是容量还是性能都强于传统方案，roaringbitmap 方案整体上来看，无论是查询耗时还是数据容量占用都有很好的性能和效果。

版权所有：腾讯云计算（北京）有限责任公司

一条 SQL 实现查询附近的人

最近更新时间：2024-01-22 16:24:46

PostGIS 是关系型数据库 PostgreSQL 的一个扩展，PostGIS 提供如下空间信息服务功能：空间对象、空间索引、空间操作函数和空间操作符。同时，PostGIS 遵循 OpenGIS 的规范。
PostGIS 支持所有的空间数据类型，这些类型包括：点（POINT）、线（LINESTRING）、多边形（POLYGON）、多点（MULTIPOINT）、多线（MULTILINESTRING）、多多边形（MULTIPOLYGON）和集合对象集（GEOMETRYCOLLECTION）等。
PostGIS 也是业界功能较全面，能力强大的空间地理数据库引擎。现如今很多业务开发中，我们经常会遇到诸如“附近的某某”的需求，如何能快速实现，通过 PostGIS+ 关系型数据库 PostgreSQL 可以帮到您。
本文为您介绍，如何通过 PostGIS 实现“附近的对象”功能。

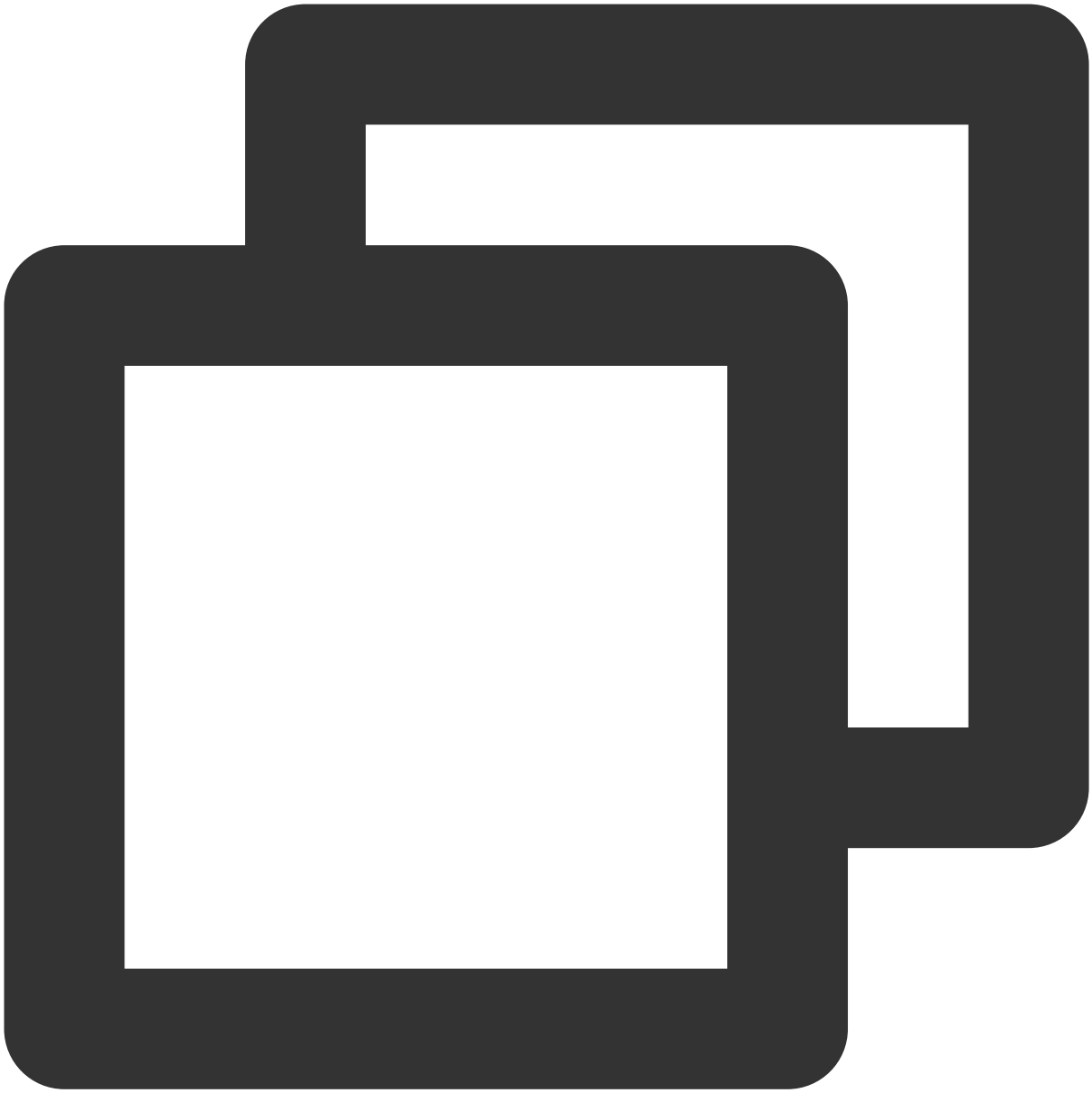
前提条件

已有一个 PostgreSQL 实例。
该实例支持 PostGIS 插件。

步骤1：创建插件

登录到数据库实例中，在业务数据库执行如下命令，登录方法您可参考 连接 PostgreSQL 实例。

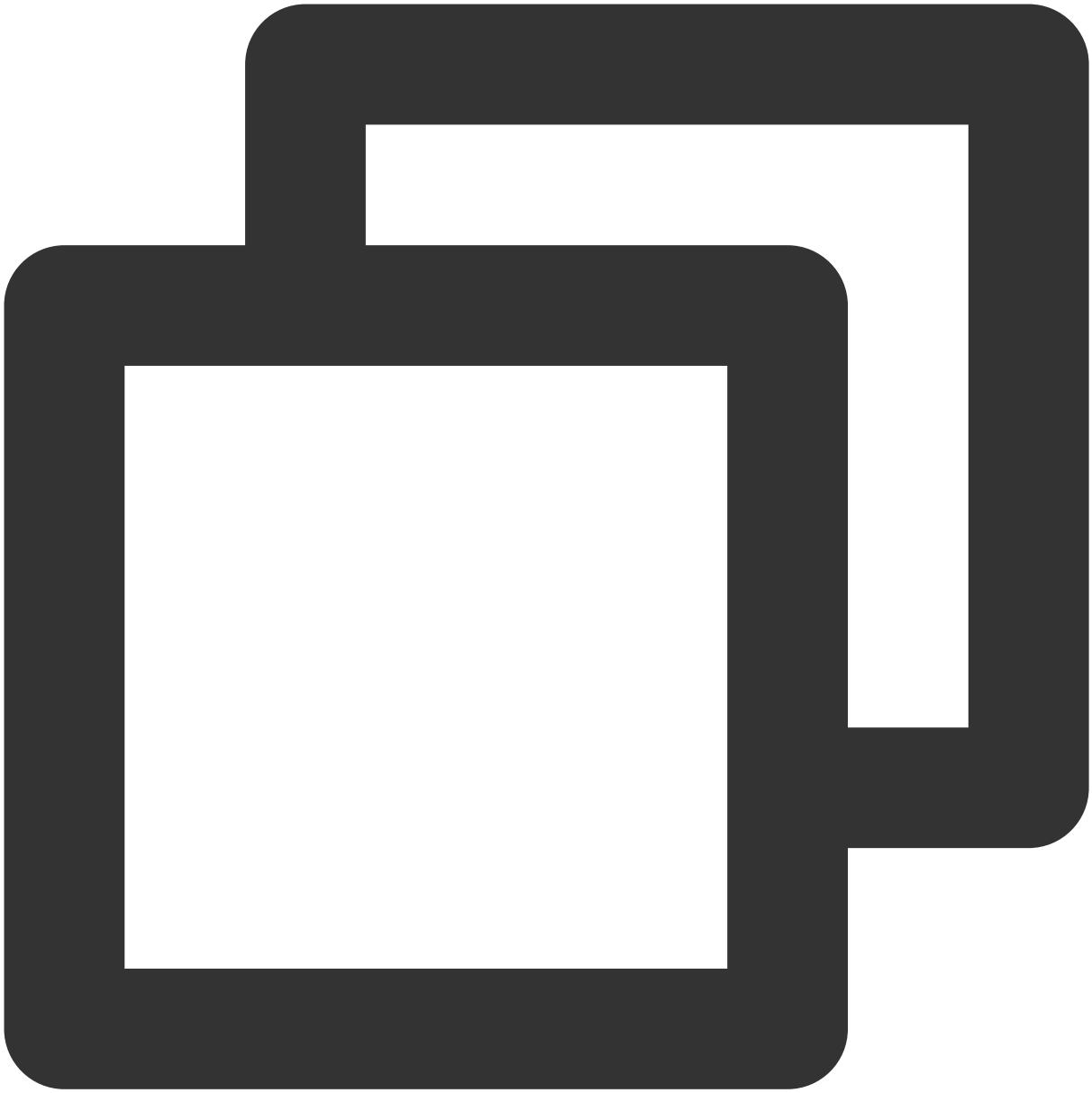
版权所有：腾讯云计算（北京）有限责任公司



```
\\c test
CREATE EXTENSION postgis;
CREATE EXTENSION postgis_topology;
```

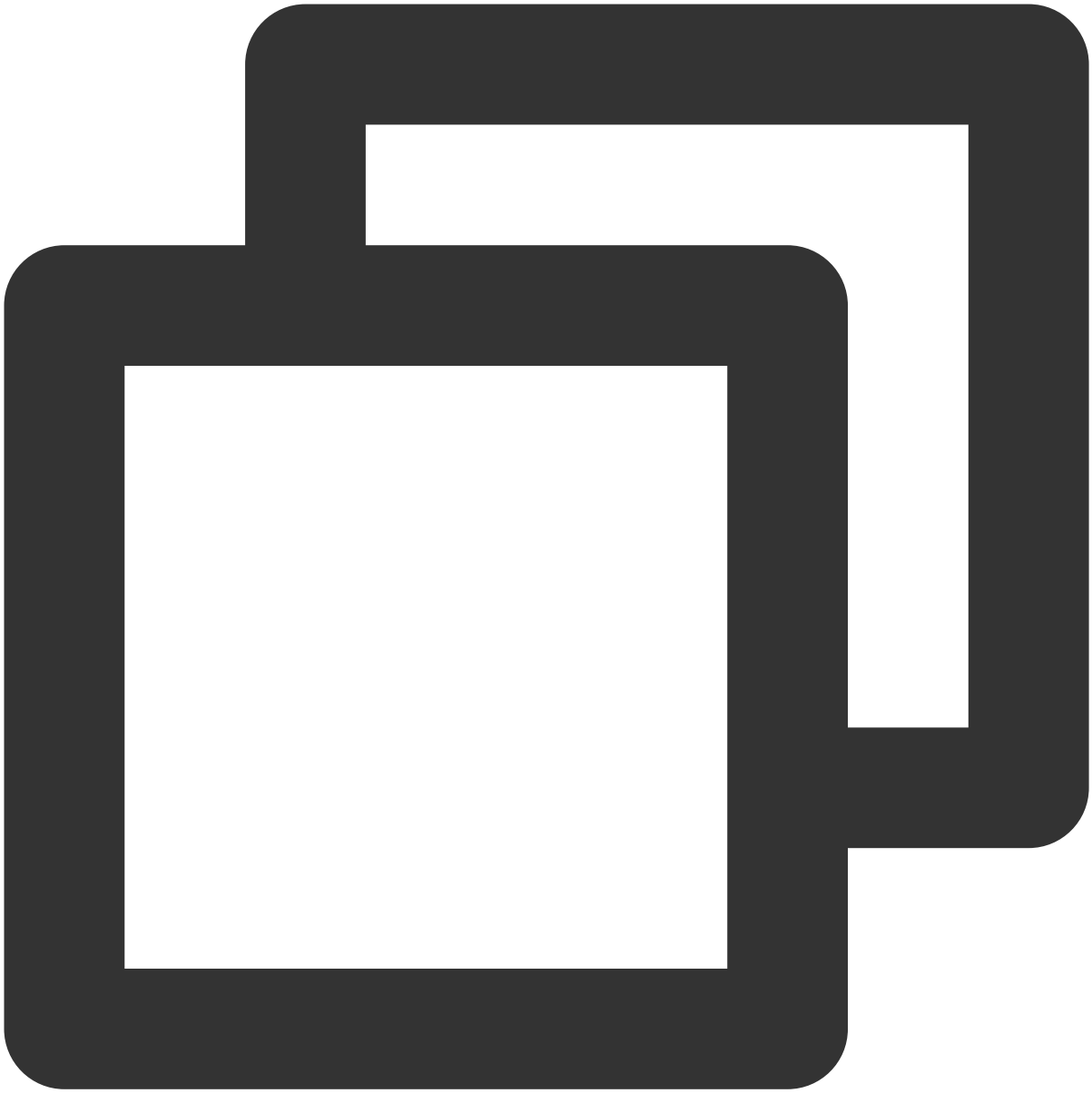
步骤2：创建测试表与索引

在业务数据库执行如下命令，TABLE 后的表名可自定义设置。



```
CREATE TABLE t_user(uid int PRIMARY KEY,name varchar(20),location geometry);
CREATE INDEX t_user_location on t_user USING GIST(location);
```

步骤3：插入测试数据



```
## 创建一个自动名字生成函数：
create or replace function random_string(length integer) returns text as
$$
declare
chars text[] := '{0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F,G,H,I,J,K,L,M,N,O,P,Q,R,S,T,U,V,W
result text := '';
i integer := 0;
length2 integer := (select trunc(random() * length + 1));
begin
if length2 < 0 then
raise exception 'Given length cannot be less than 0';
```

```
end if;
for i in 1..length2 loop
result := result || chars[1+random()*(array_length(chars, 1)-1)];
end loop;
return result;
end;
$$ language plpgsql;

## 插入一千万条测试数据
insert into t_user select generate_series(1,10000000), random_string(20),st_setsrid
```

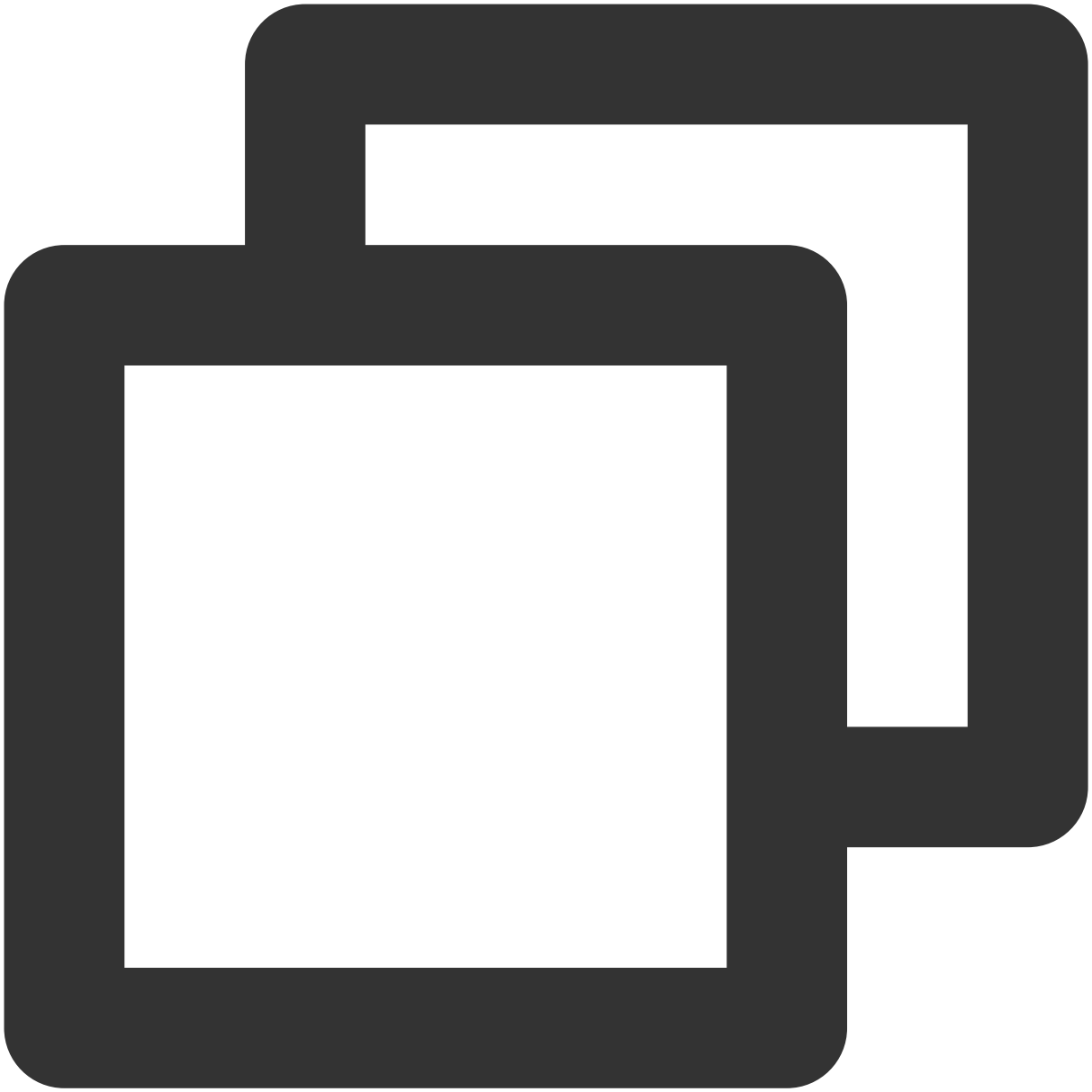
步骤4：查询附近的人

1. 首先在 拾取坐标系统 中随便找一个坐标。 此处用天安门广场的坐标作为示例：116.404177,39.909652。
2. 确定好后，以此作为要查询的坐标，然后在数据库中找到距离这个坐标最近的5个对象，并输出这五个对象离此地的距离，此处单位是：百公里。

说明：

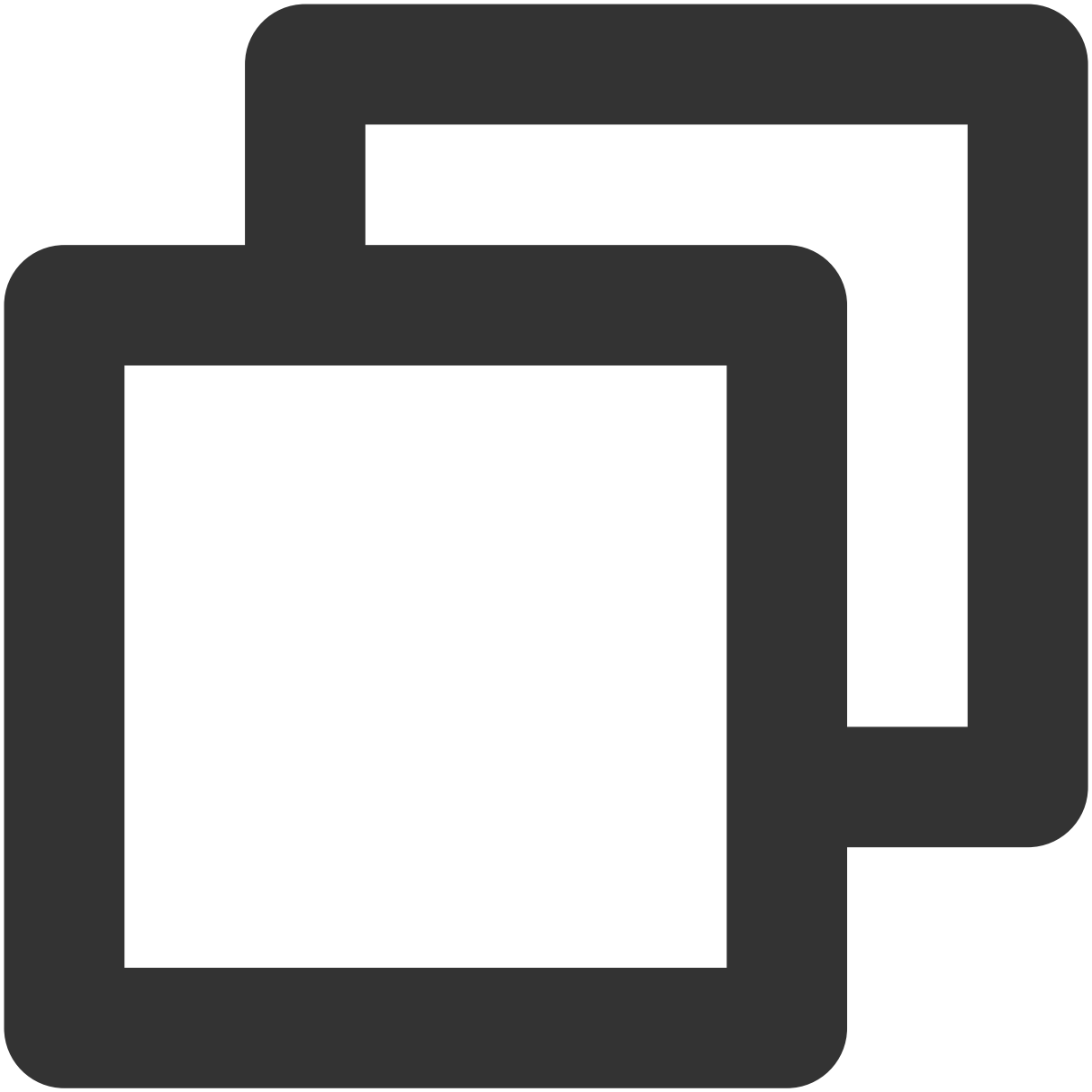
WGS84 是目前最流行的地理坐标系统。在国际上，每个坐标系统都会被分配一个 EPSG 代码，EPSG:4326 就是 WGS84 的代码。GPS 是基于 WGS84 的，所以通常我们得到的坐标数据都是 WGS84 的。一般我们在存储数据时，仍然按 WGS84 存储。

执行命令：



```
select uid, name, ST_AsText(location), ST_Distance(ST_GeomFromText('POINT(116.40417
```

3. 查看距离此坐标对象1000米以内的所有对象与距离。



```
select uid, name, ST_AsText(location),ST_Distance(ST_GeomFromText('POINT(116.404177
```

如何配置云数据库 PostgreSQL 作为 GitLab 外部数据源

最近更新时间：2024-01-22 16:24:46

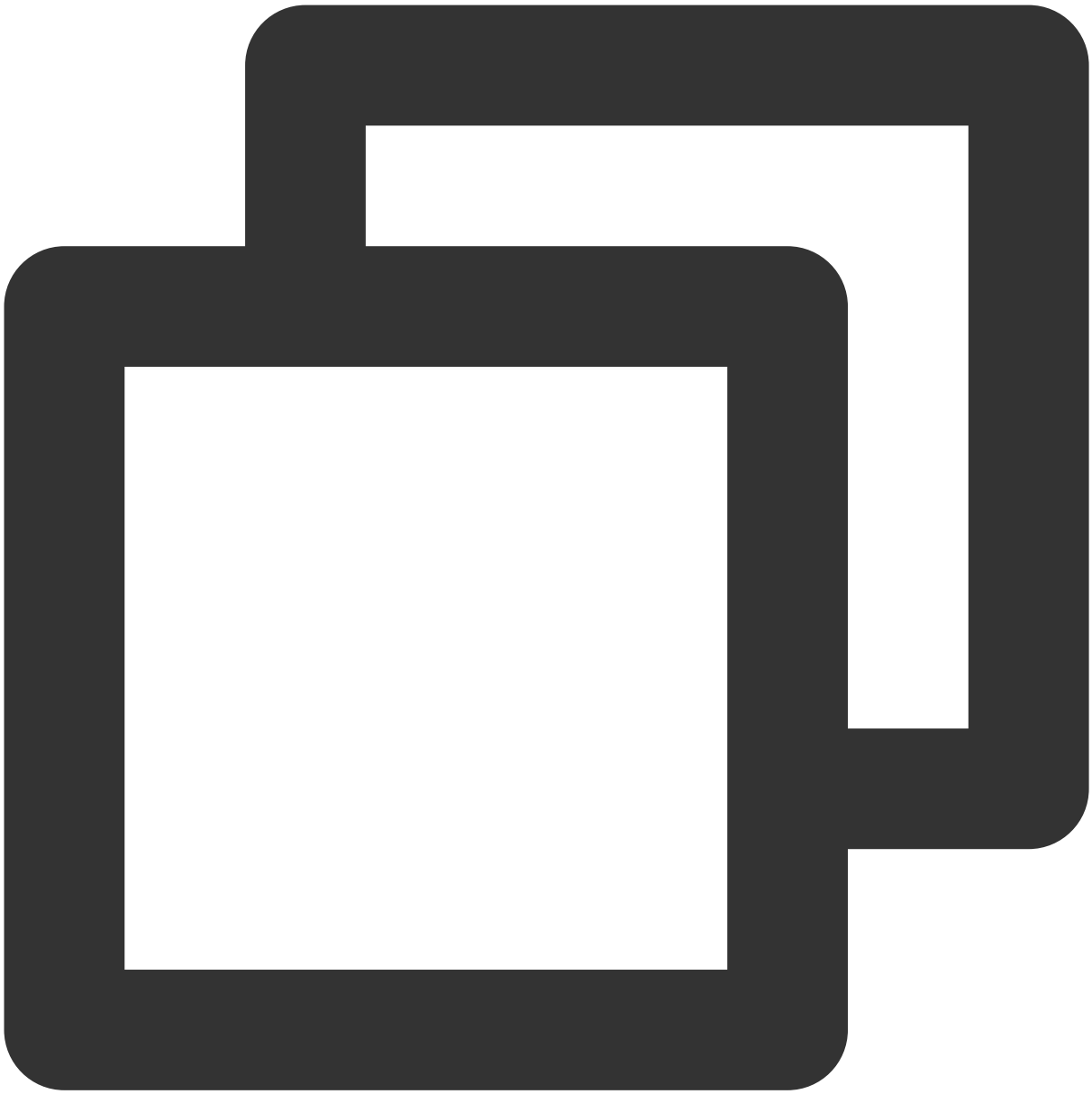
背景

Gitlab 是一种类似 github 的服务，企业可以使用它来提供 git 存储库的内部管理。它是一个自我托管的 Git-repository 管理系统，可以保持用户代码的私密性，并且可以轻松地部署代码的更改。其优势在于：
GitLab 提供了 GitLab Community Edition 版本，供用户在他们的代码所在的服务器上进行定位。
GitLab 免费提供无限数量的私人和公共存储库。
代码片段可以共享项目中的少量代码，而不是共享整个项目。
Gitlab 服务的元数据库在新版本（12.1）中目前仅支持 PostgreSQL，Gitlab 介绍不再支持 MySQL 的理由如下：
无法支持嵌套分组查询。
必须使用黑科技来提升 MySQL 对列的限制，这将导致 MySQL 拒绝存储数据。
MySQL 无法添加 TEXT 类型字段的长度限制。
MySQL 不支持分区索引。
而 PostgreSQL 都能支持到以上场景。所以 Gitlab 在安装包中集成了 PostgreSQL，但对于某些企业而言，集成的数据库服务存在一定的安全风险，并且数据库服务的可靠性和可用性都不能得到保证。为了确保代码托管服务的稳定，部分业务和企业会选择采用稳定的外部数据库服务。而 Gitlab 在 Gitlab HA Repmgr 包中才支持基于 patroni 版本的高可用数据库。但是自行维护集群是一件成本较高的事情，采用腾讯云数据库可极大的减少这些维护操作。
本文介绍如何将 Gitlab 中的嵌入式数据库服务更换为腾讯云数据库 PostgreSQL 服务。

步骤1：安装 GitLab

1. 准备资源
CentOS Linux release 7.6.1810 (Core)。
gitlab-ce 14.9.3 版本。
云服务器一台，内存需4GB以上，磁盘需50GB以上。建议 /opt 独立挂载一个数据盘。
腾讯云数据库 PostgreSQL 一个，规格根据自身实际情况进行配置。可初始选择一个规格较小的实例。再根据具体使用进行扩容。版本根据 Gitlab 的版本进行调整选择。
2. 下载 GitLab单击此处 在跳转页面下找到想要安装的 Gitlab 安装包，下载到本地后再上传至需要安装 Gitlab 服务的服务器中。
3. 安装 GitLab
使用 root 用户执行下列语句安装 Gitlab，若最后一步操作提示存在依赖包未安装，可直接通过 yum 或其他安装工具

提前安装完成：



```
curl https://packages.gitlab.com/install/repositories/gitlab/gitlab ce/script.rpm.s
sh gitlab-ee_install.sh
export EXTERNAL_URL=https://gitlab.example.com
yum install -y curl policycoreutils-python openssh-server cronie
rpm -ivh gitlab-ce-13.10.2-ce.0.el7.x86_64.rpm
```

步骤2：初始化数据源 PostgreSQL

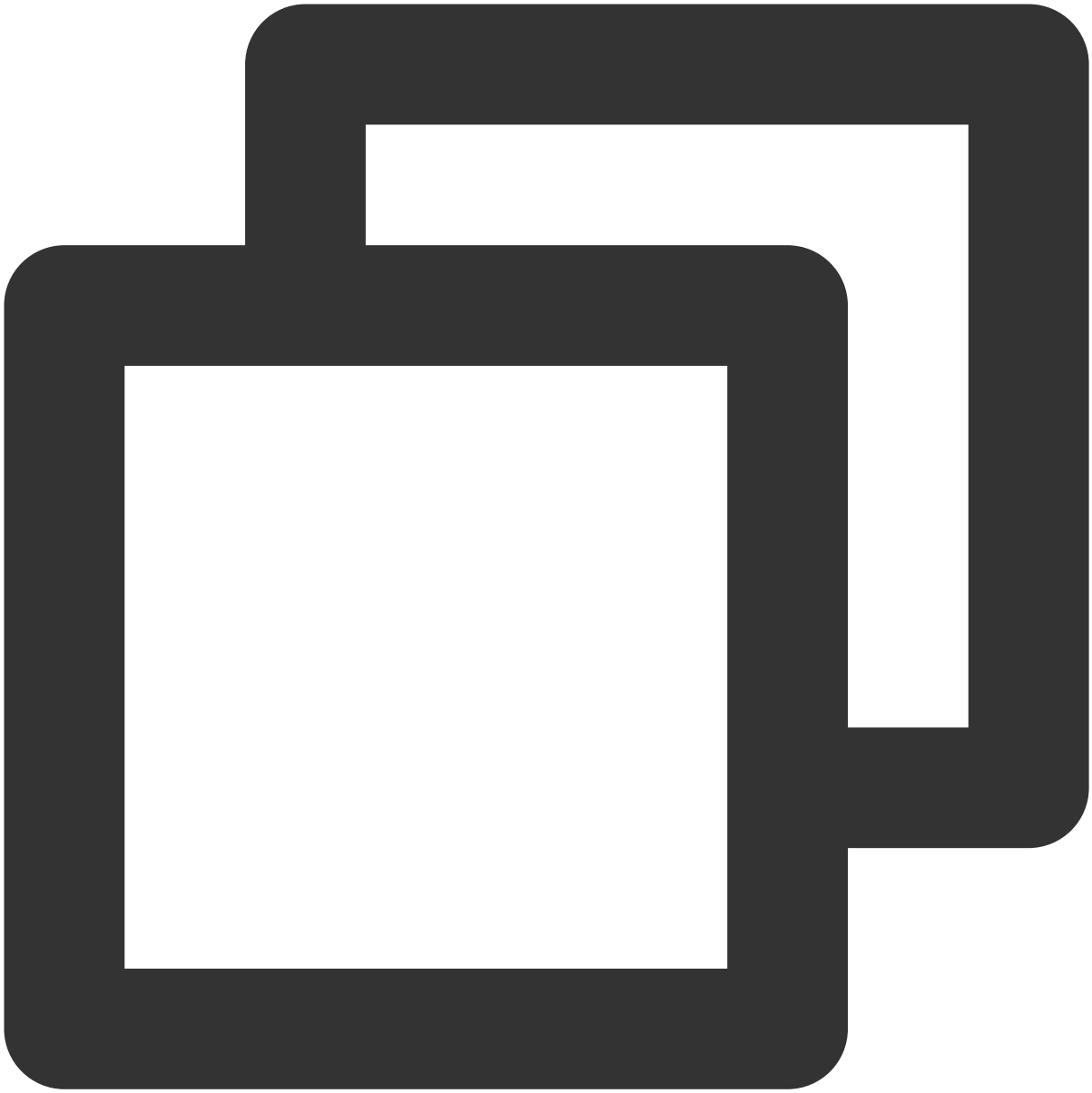
1. 可使用云上数据库服务，如腾讯云数据库 PostgreSQL，要创建腾讯云数据库 PostgreSQL 请参见 创建 PostgreSQL 实例。

注意：

创建或安装数据库时需确保数据库版本与 GitLab 版本要对应。否则在初始化 GitLab 时候将提示版本不一致，无法创建成功。

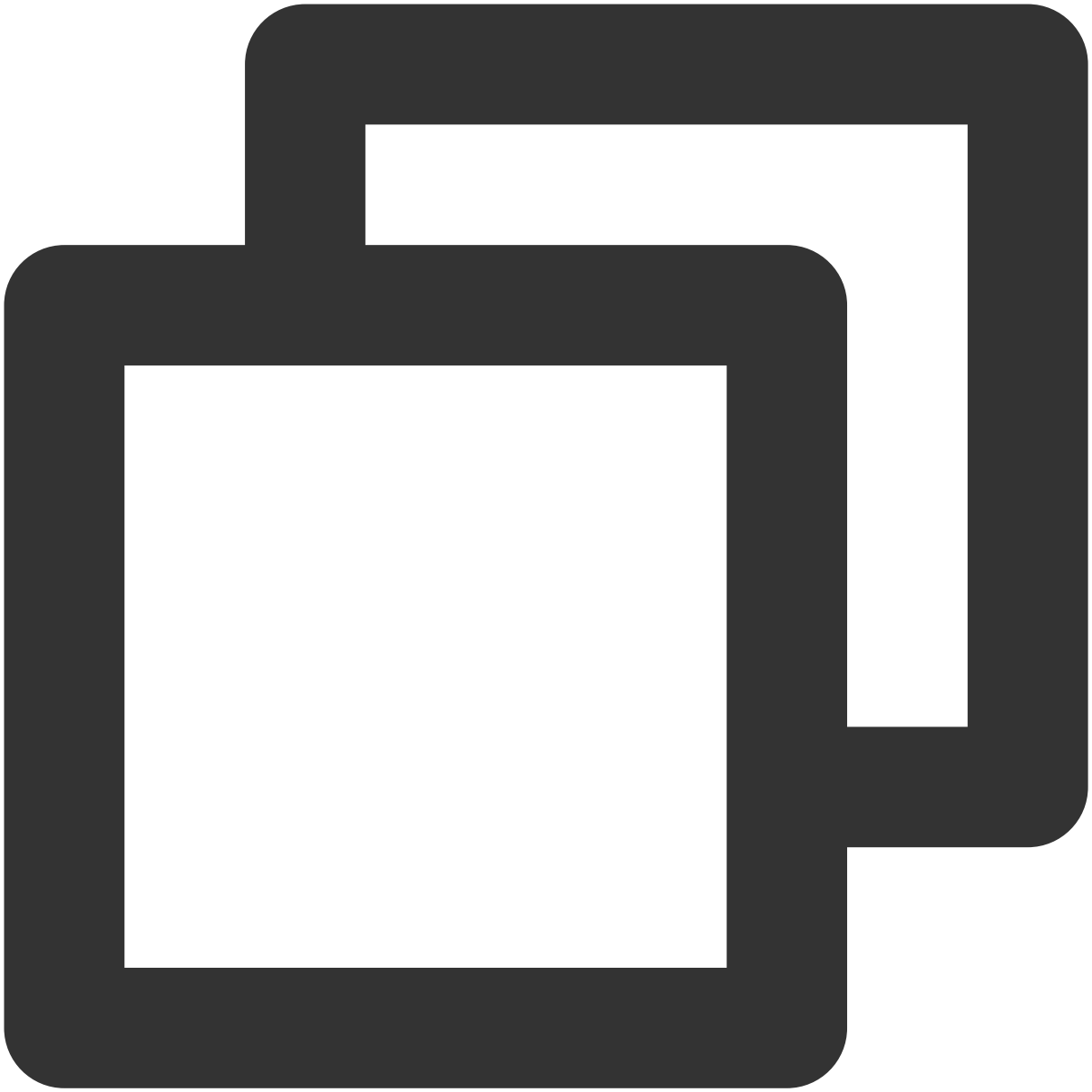
GitLab 版本	最小支持的 PostgreSQL 版本
13.0	11
14.0	12

2. 通过客户端工具登录至腾讯云数据库 PostgreSQL 中。可使用 psql 工具测试是否可以直接访问数据库，若无法访问，请排查网络链路和安全组配置。



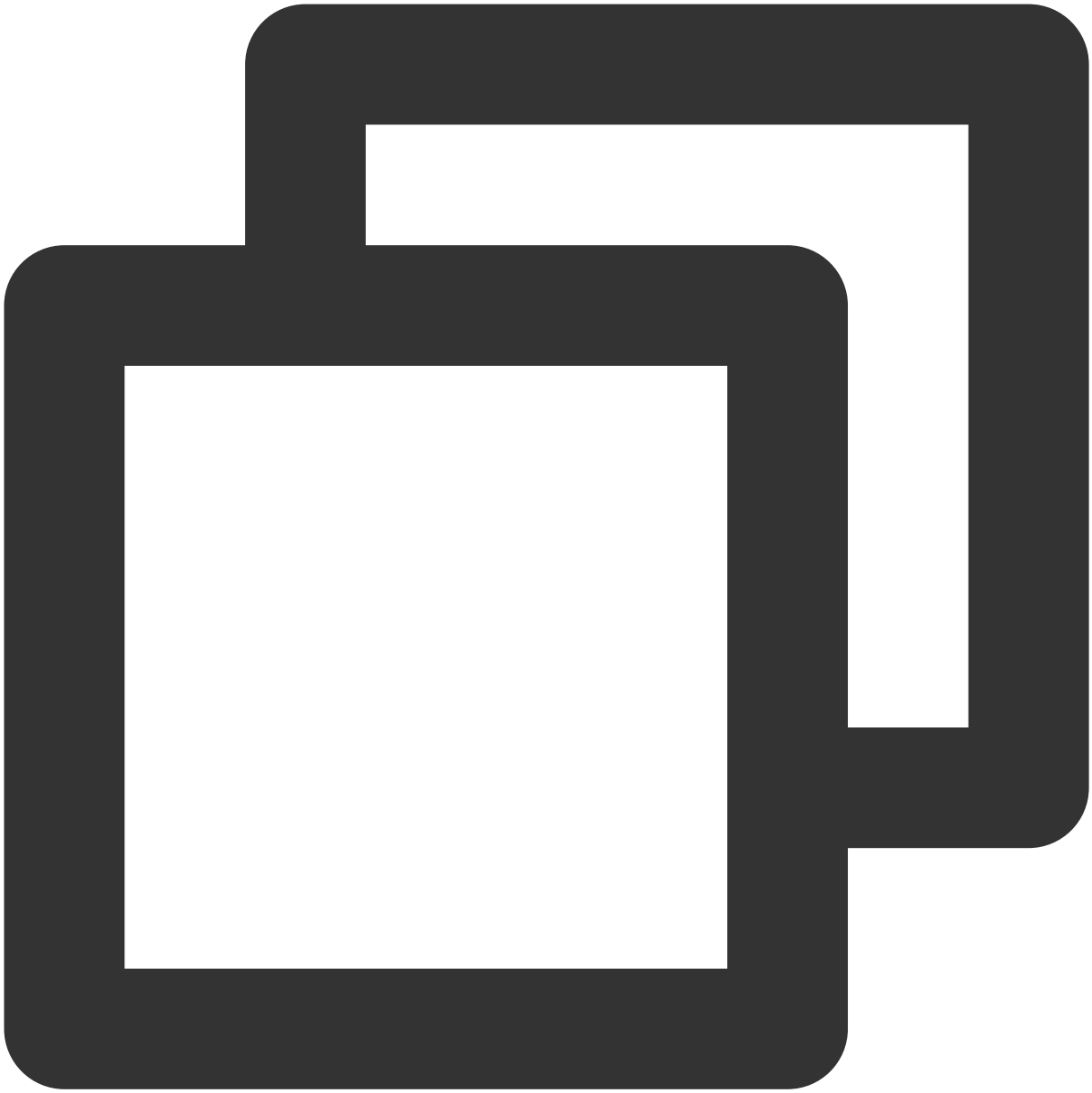
```
psql -U <数据库管理员> -p <端口> -d postgres -h <访问地址>
```

3. 首先在数据库中创建一个 GitLab 所使用的帐号，如 gitlab，请注意此帐号必须拥有 superuser 权限或者云数据库所给与的管理员权限，如腾讯云的 `pg_tencentdb_superuser` 。



```
create user gitlab login password 'gitlab_****_password#123';
grant gitlab to <当前管理员帐号>; grant pg_tencentdb_superuser to gitlab;
```

4. 然后创建一个 gitlab 所管理使用的 database：



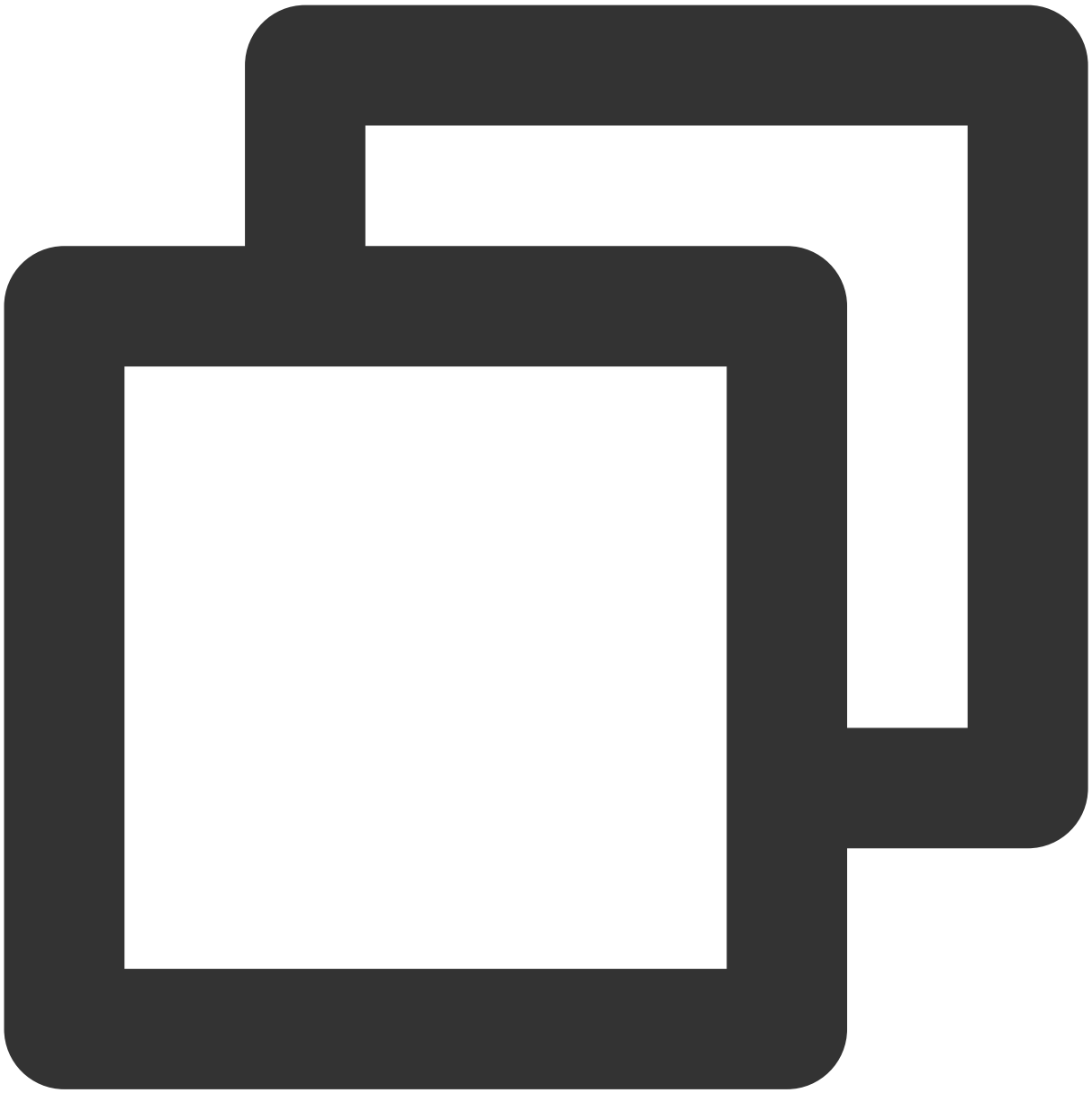
```
create database gitlab owner=gitlab ENCODING = 'UTF8';
```

注意：

在 GitLab 库中必须要能支持 `pg_trgm、btree_gist、plpgsql` 插件，无需提前创建，在初始化 GitLab 时候将自动创建。但是需要保证能创建成功。

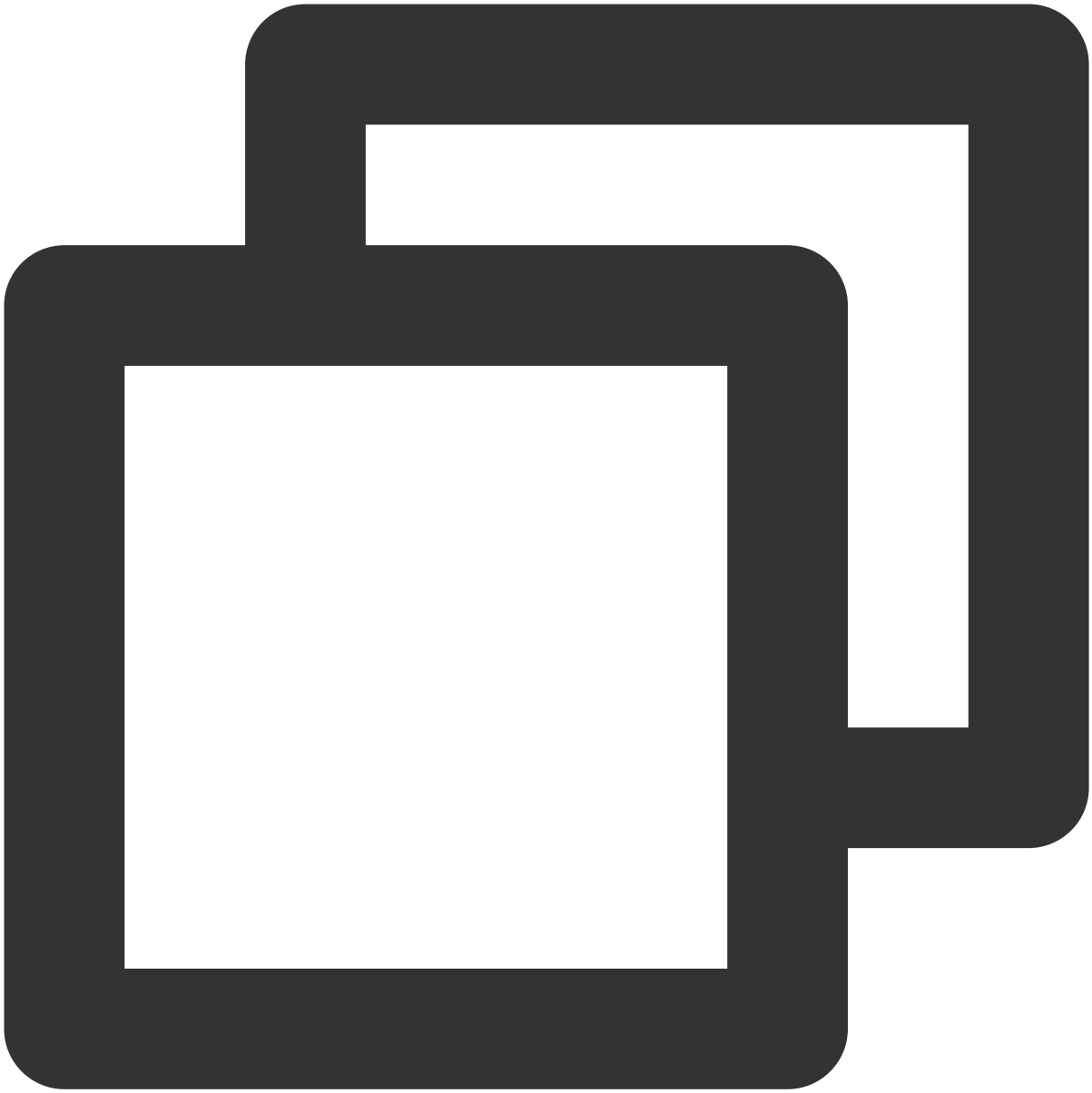
步骤3：修改 GitLab 元数据库为腾讯云数据库 PostgreSQL

1. 登录至 GitLab 安装服务器中，找到 gitlab 配置文件，默认为：/etc/gitlab/gitlab.rb。默认此文件中未配置任何信息，可通过下列命令查看到相关信息：



```
# cat /etc/gitlab/gitlab.rb |grep -v ^# | grep -v ^$
external_url 'http://gitlab.example.com'
```

2. 在此文件的最后加入以下信息，将腾讯云数据库 PostgreSQL 数据源加入到 gitlab 中：



```
## postgresql connect
## 此参数设置为 false 指禁用内置的 postgresql，而使用外部 postgresql 数据源
postgresql['enable'] = false
gitlab_rails['db_adapter'] = "postgresql"
gitlab_rails['db_encoding'] = "utf8"
## 数据库名
gitlab_rails['db_database'] = "gitlab"
gitlab_rails['db_pool'] = 100 ## 数据库用户
gitlab_rails['db_username'] = "gitlab"
## 密码，请根据自身配置修改
gitlab_rails['db_password'] = "gitlab_Test_password#123" ## 访问地址
```

版权所有：腾讯云计算（北京）有限责任公司

```
gitlab_rails['db_host'] = "gz-tdcpg-ep-6kvx6p19.sql.tencentcdb.com" ## 访问端口
gitlab_rails['db_port'] = "25870"
```

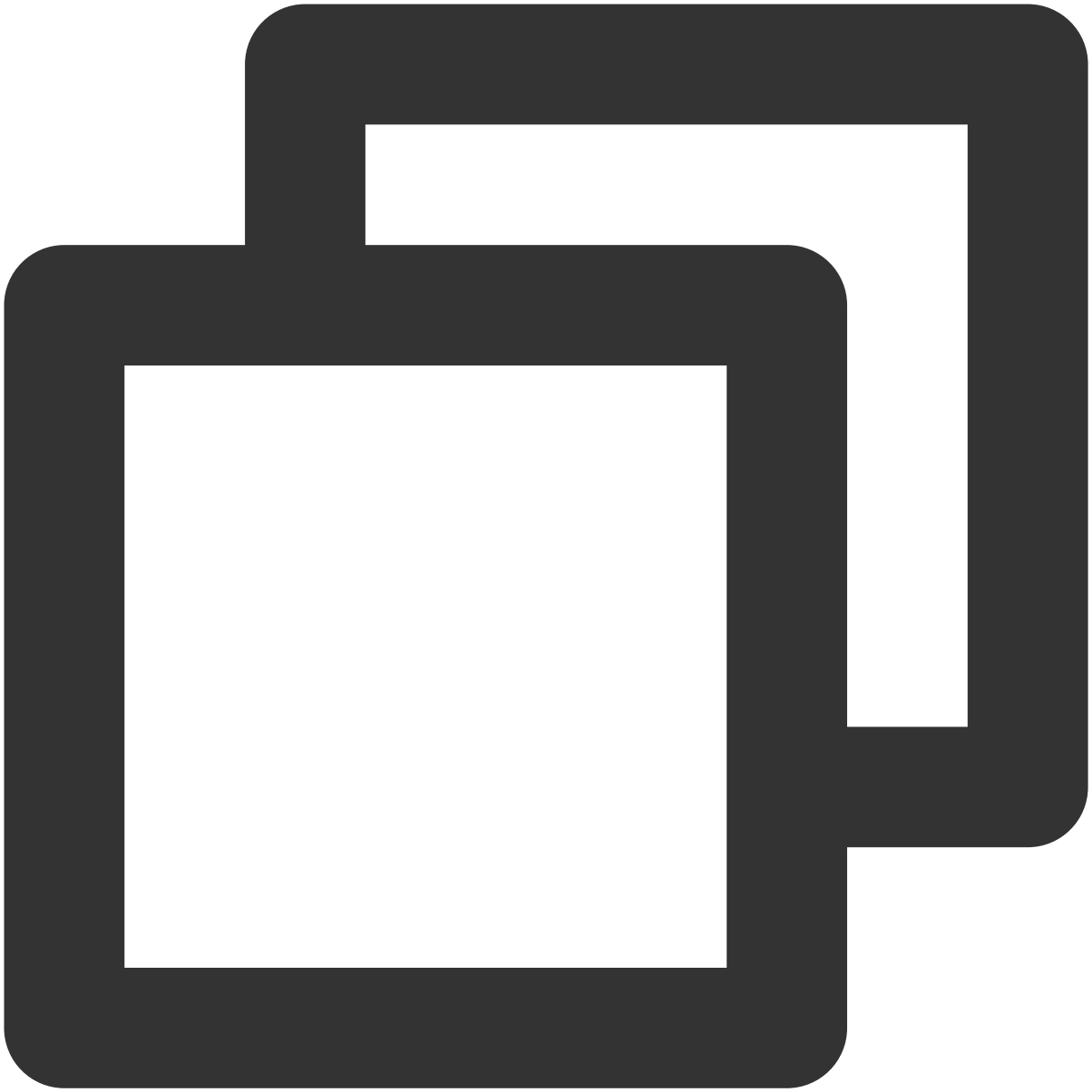
请注意，此处如果配置访问地址为域名时，在初始化时候，将提示：

`ActiveRecord::ConnectionNotEstablished: could not translate host name "gz-tdcpg ep-6kvx6p19.sql.tencentcdb.com " to address: Name or service not known`

所以若数据库访问为域名时，请使用 `ping` 命令找到此域名的 IP 地址或者找到能解析此域名的 DNS 服务器，不建议将访问地址的域名直接修改为 IP 地址，因为使用域名的场景常伴有数据库后端是做了负载均衡或者高可用，可直接在服务器中配置 DNS 服务器或者 host。若数据库服务有变化，则可直接修改 host 或者 DNS 服务，避免对 GitLab 服务进行修改。

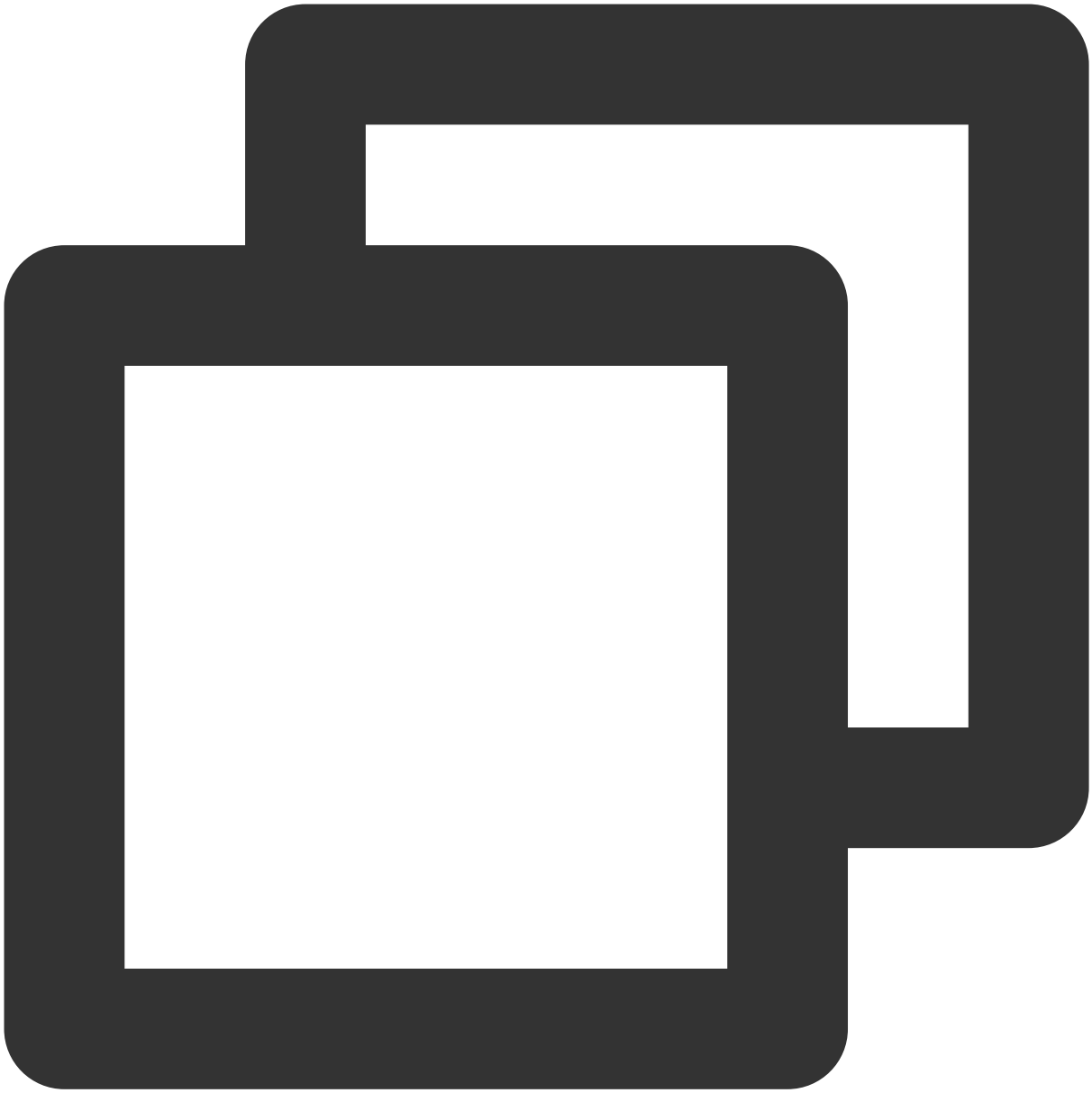
步骤4：初始化与登录使用 GitLab

1. 执行以下命令初始化 GitLab，此命令执行会消耗一段时间，请耐心等待，当提示：`gitlab Reconfigured!` 时，说明已经初始化完成。



```
gitlab-ctl reconfigure
```

2. 执行以下命令启动 GitLab。

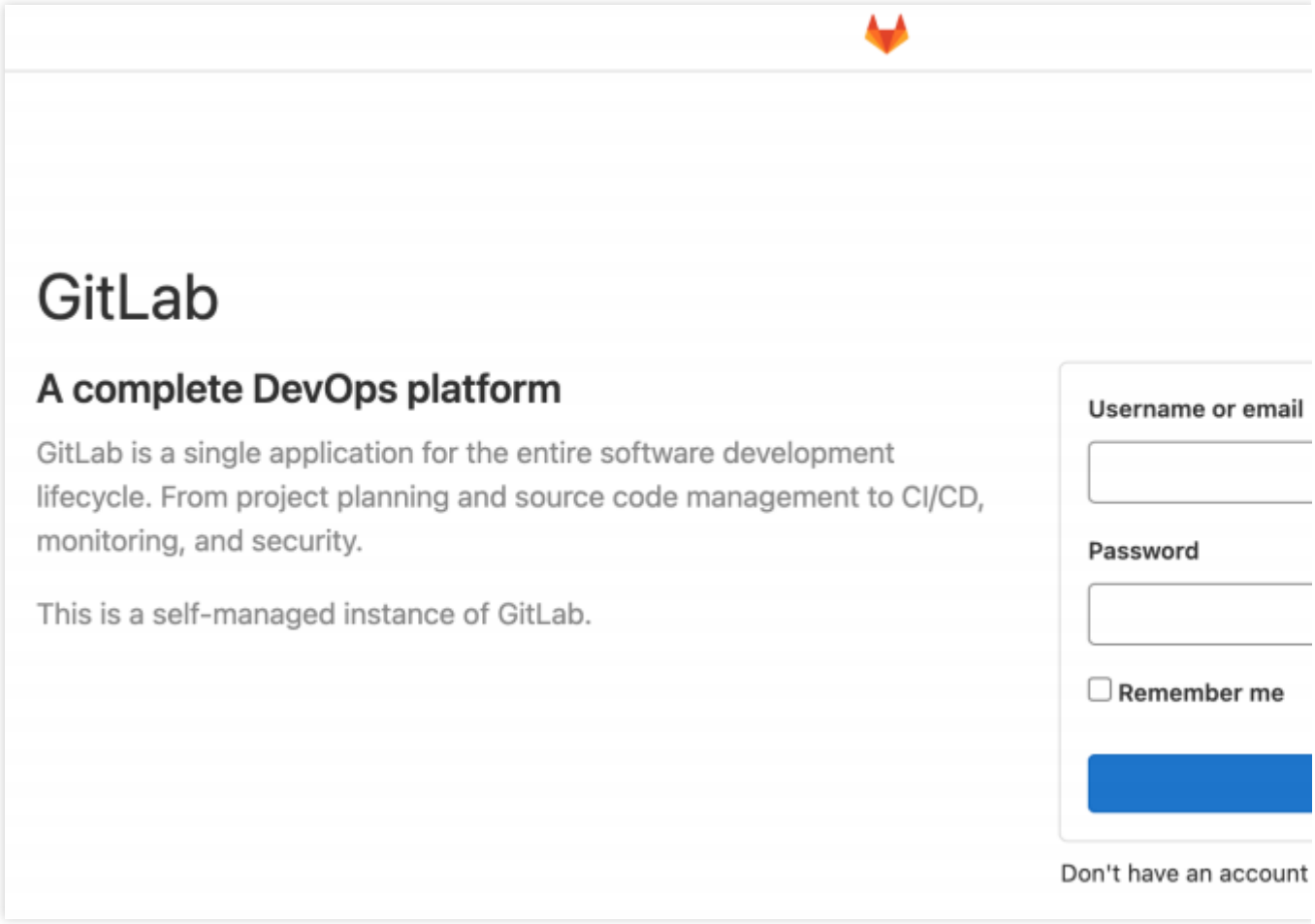


```
gitlab-ctl startok
```

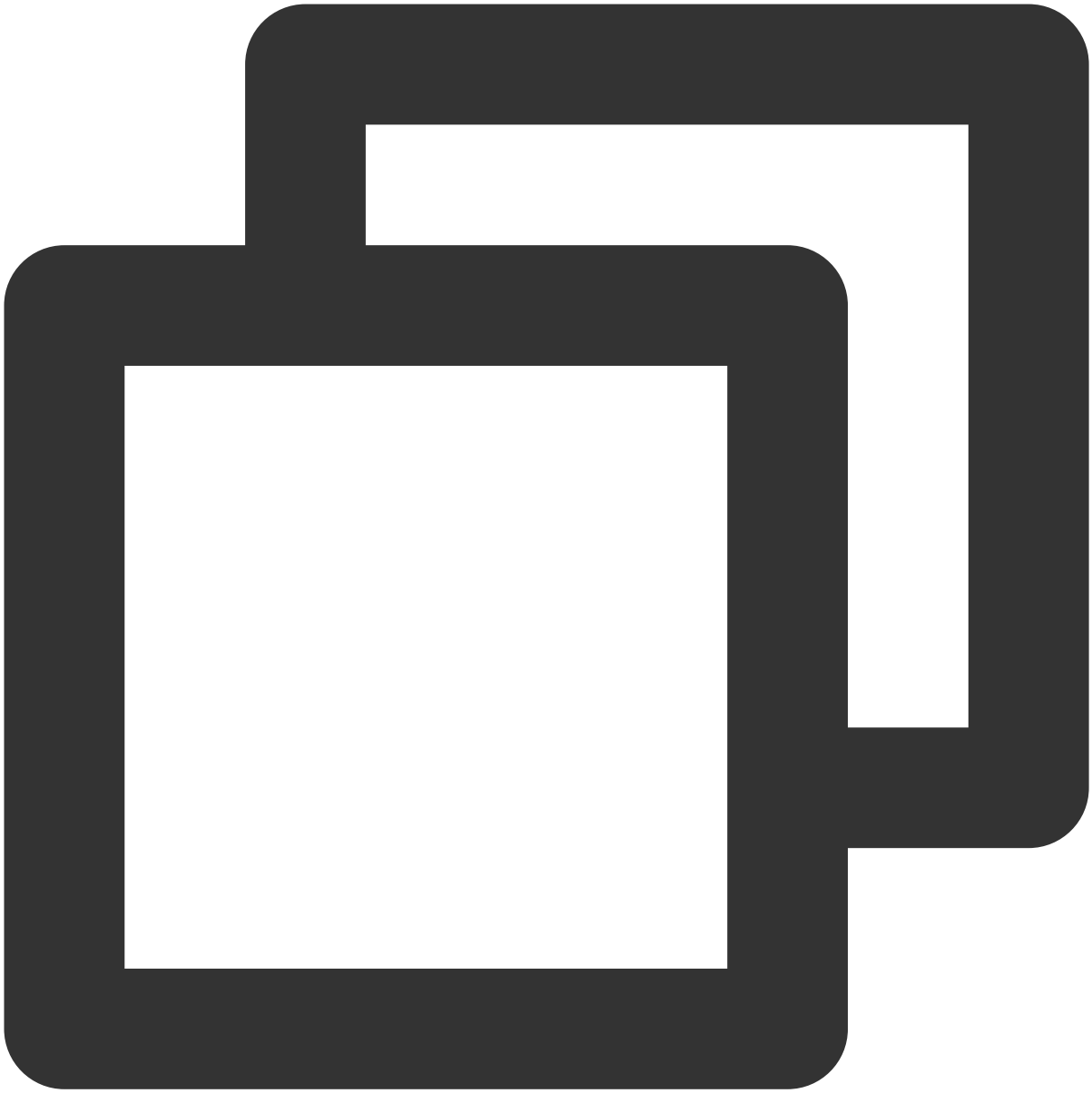
3. 可使用以下 URL 访问 GitLab，若无法访问通，可能是服务器防火墙的限制。

地址示例：`http://{可访问的服务器 IP 地址}/users/sign_in`

登录界面如下：



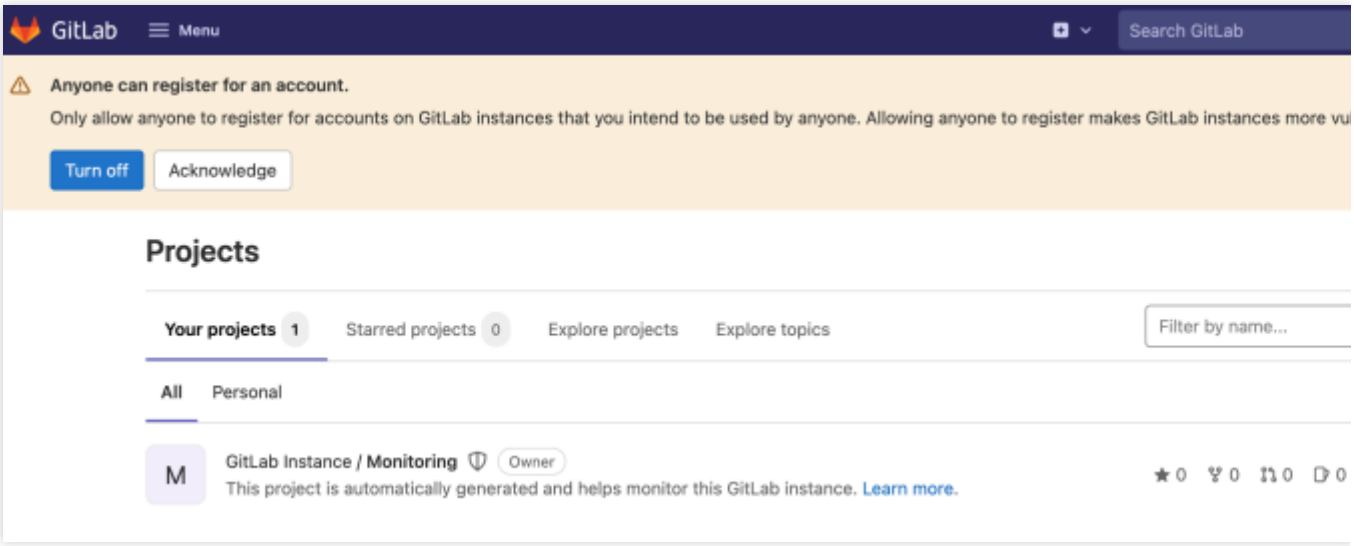
4. 初始登录帐号为 root，初始密码在初始完成后，会有如下的提示：



```
Password stored to /etc/gitlab/initial_root_password. This file will be
cleaned up in first reconfigure run after 24 hours.
```

说明：

可在服务器中的此文件中找到初始化密码。完成登录后，请记得修改密码。



此时，GitLab 就安装完成，后续将正式使用 GitLab。

通过 cos_fdw 插件支持分级存储能力

最近更新时间：2024-01-22 16:24:46

背景

数据库作为数据存放、处理、加工的核心组件，随着业务的发展，其数据量会越来越大；由于时间或业务设计逻辑的原因，会存在部分历史数据、归档数据。而业务对此类数据的访问并不频繁，但又不能删除，因为在某些场景下会使用到这些数据。为了提升数据库的处理性能，需要将此类数据进行落冷处理。
对于数据库而言，如何最大化地存储数据以及更好的提供统一数据处理接口尤为重要，腾讯云数据库 PostgreSQL 针对此类用户需求，提供数据分级存储方案。其核心原理是支持多种成本的存储介质，供用户选择使用。如，可使冷数据存放于性能略低，但成本低的存储中，将热数据存放于成本较高，但性能更强的高性能 SSD 中。更好的服务用户，保证业务的正常运行，并且兼顾用户成本，是一种极具性价比的存储方案。

方案简介

腾讯云 COS 是腾讯云提供的对象存储服务。分级存储当前实现的能力主要是基于 cos_fdw 插件连接和解析 COS 上的文件数据。
通过 cos_fdw 插件可以将 COS 中的数据加载到 PostgreSQL 数据库表中，像访问普通表一样访问 COS 中的数据，实现冷热存储分离。用户无需关心不同存储介质的访问形式，仅需要将 COS 存储中的数据文件配置到 PostgreSQL 数据库中即可。

方案优势

统一引擎：多种存储介质，无需业务层改动代码，直接使用 PostgreSQL 数据协议均可实现统一访问。
成本更低：相对高性能 SSD 存储，整体成本降低 86.25%。
使用简单：用户仅需要将源端数据导出 CSV 格式存放于 COS 中，在云数据库 PostgreSQL 基于插件进行外表创建，即可像原表一样直接使用。
无限存储：COS 存储容量不设上限，用户可以根据实际情况进行动态存储，不再担心容量问题。
支持联合查询表：多种存储的表支持联合查询，跨区 join 等，这在其他混合引擎上是无法直接实现的，均需要一个统一的数据融合节点才能支持。

支持版本

目前分级存储支持以下版本的云数据库 PostgreSQL：

版权所有：腾讯云计算（北京）有限责任公司

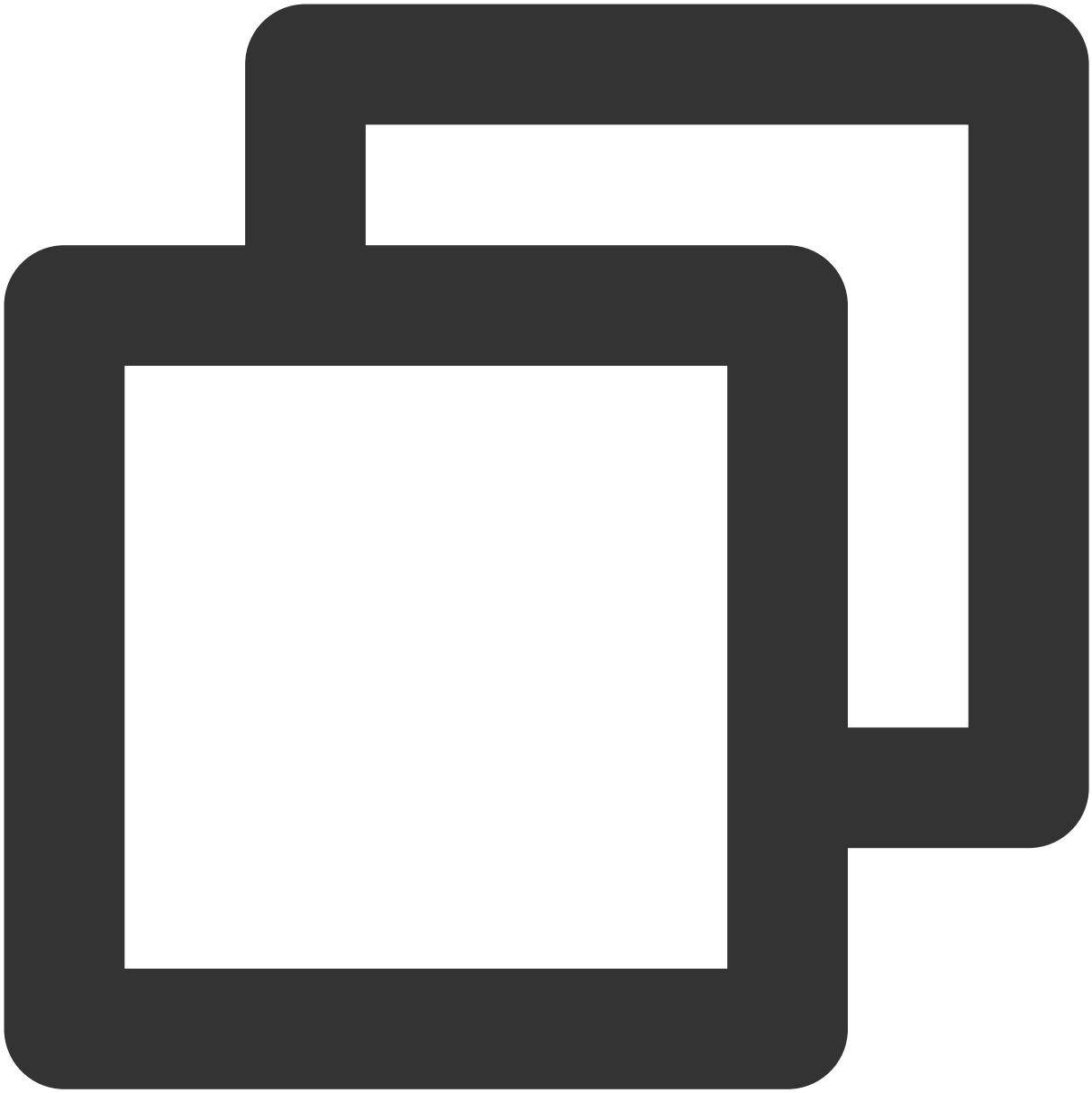
PostgreSQL 10
PostgreSQL 11
PostgreSQL 12
PostgreSQL 13
PostgreSQL 14

使用 cos_fdw 的方法

需要按照以下顺序使用 cos_fdw ：
1. 导出数据。
2. 上传至 COS。
3. 创建 cos_fdw 插件。
4. 创建 Foreign Server。
5. 创建 Foreign Table。
6. 查询外部表。

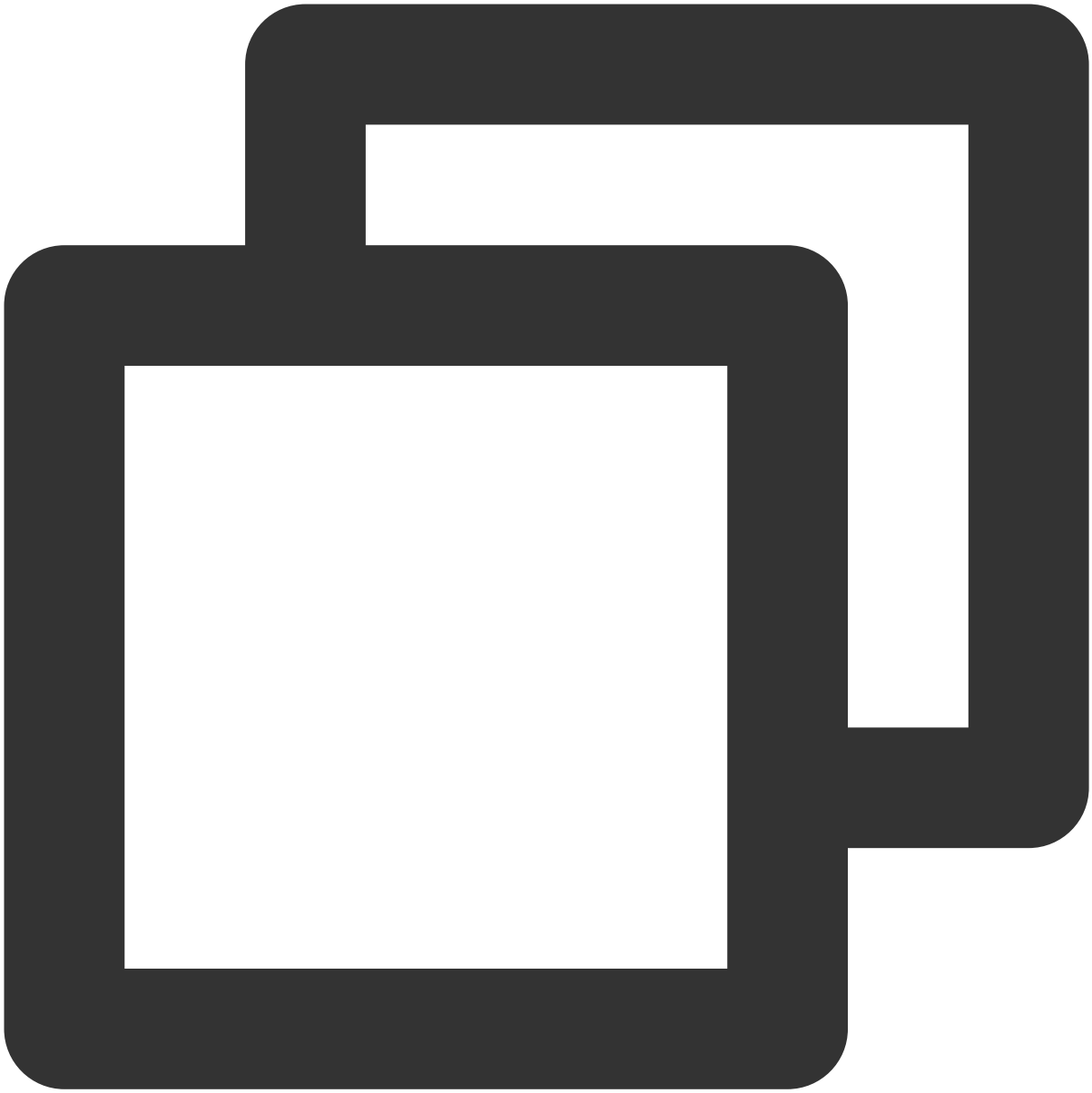
初始化环境

首先申请一个在数据库与 COS 同地域，同可用区的规格较小的中转服务器如 CVM。
操作系统建议为：Centos 7。
1. 安装 PostgreSQL 客户端，可参考 PostgreSQL 官网下载安装方案。



```
sudo yum install -y
https://download.postgresql.org/pub/repos/yum/reporpms/EL-7-
x86_64/pgdg-redhat-repo-latest.noarch.rpm
sudo yum install -y postgresql13
```

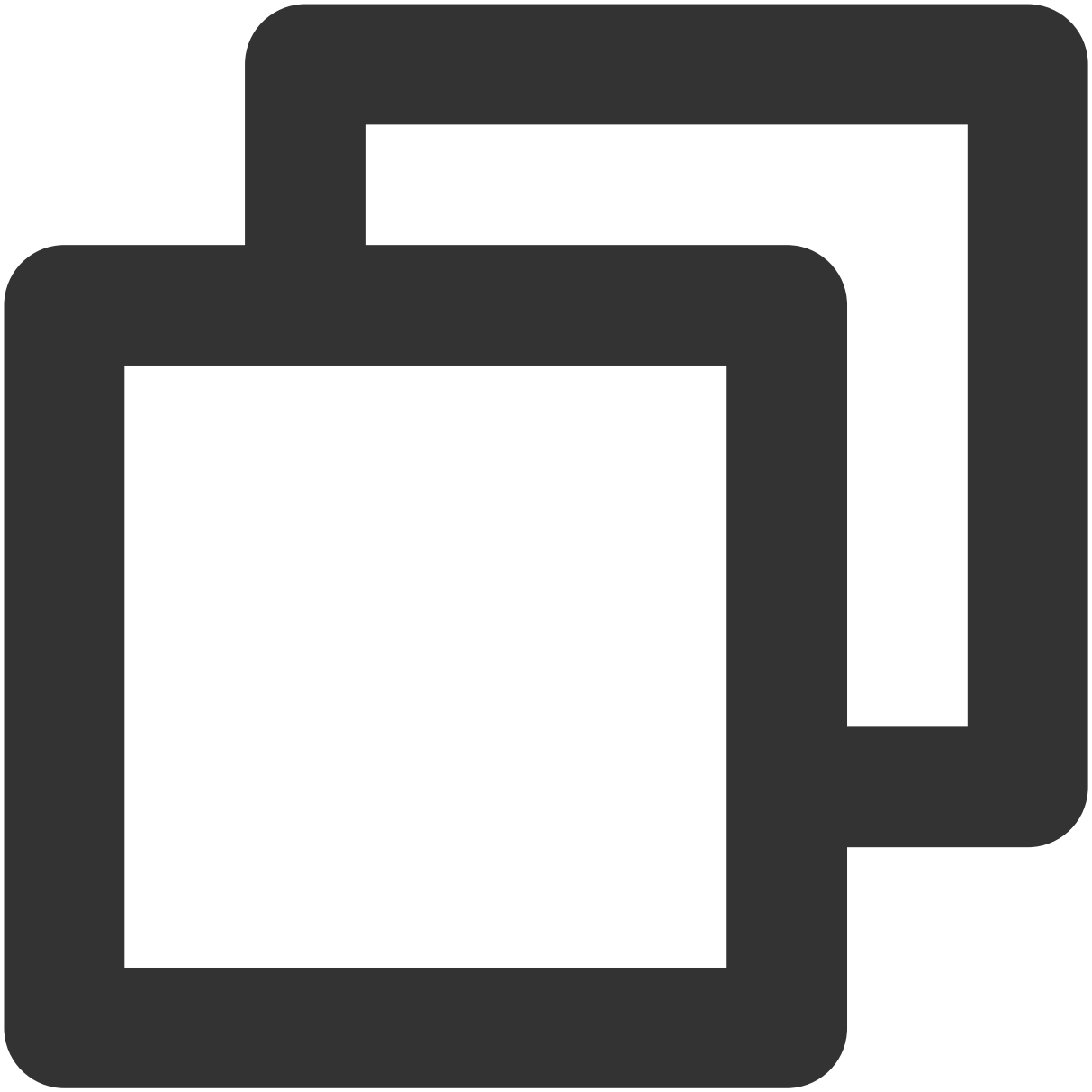
2. 安装完成后，可使用 psql 命令访问一下数据库，查看是否安装完成，命令如下：



```
psql -Uroot -p 5432 -h 10.x.x.8 -d postgres
Password for user root:
psql (13.6, server 13.3)
Type "help" for help.

postgres=>
```

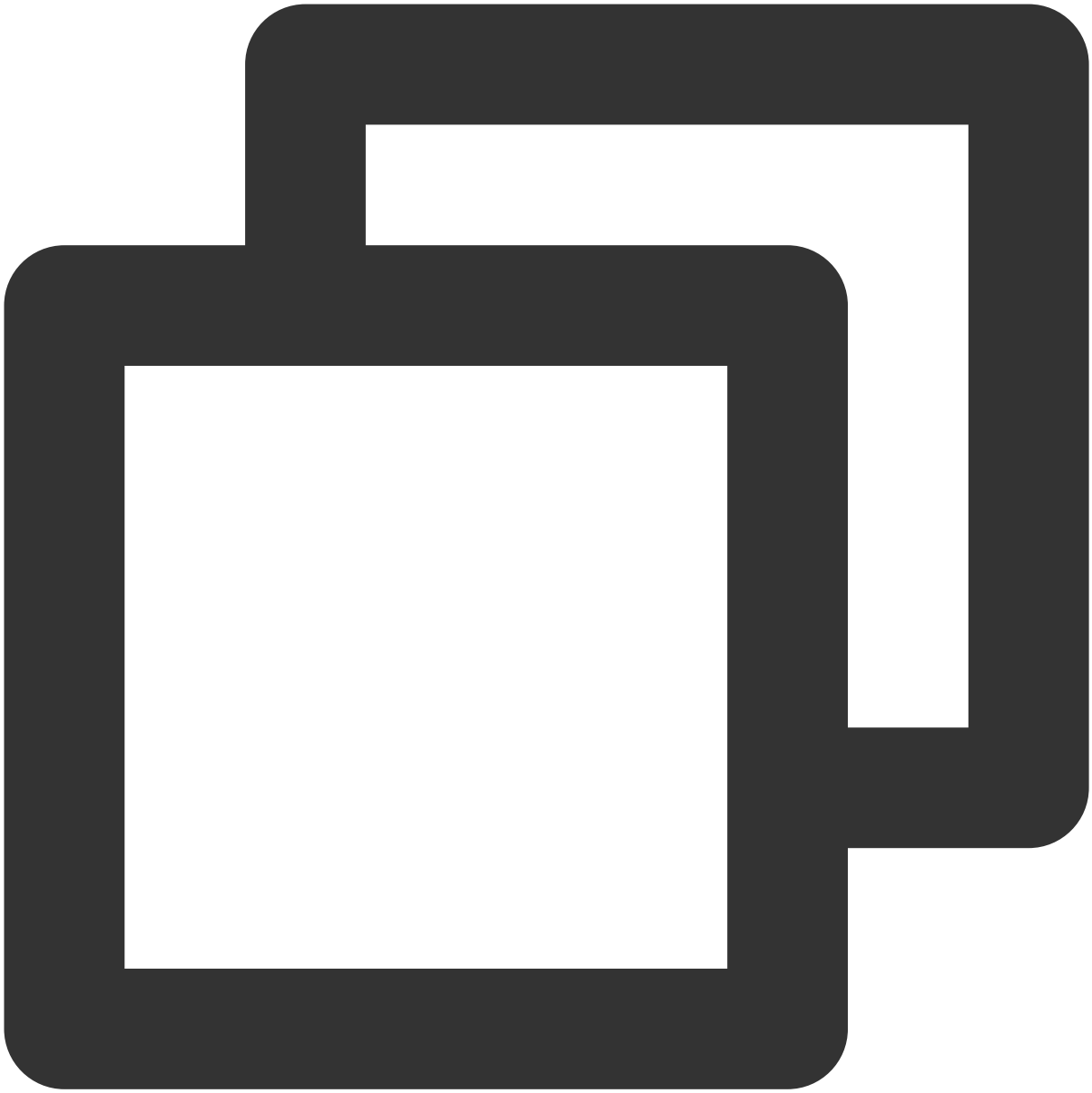
3. PostgerSQL 客户端工具安装完成后，进行 COS 挂载。这里我们通过 COSFS 挂载到服务器上的形式来进行，可避免需要更大容量的 CVM 进行转储上传。请参见 通过 COSFS 挂载。

4. 针对当前环境，可执行以下命令安装依赖包。



```
sudo yum install libxml2-devel libcurl-devel -y
```

5. 访问 COSFS 的 github 下载地址，下载 COSFS 的安装包。
6. 下载完成后将此安装包上传至此服务器中。再执行下列命令将 COSFS 安装成功。

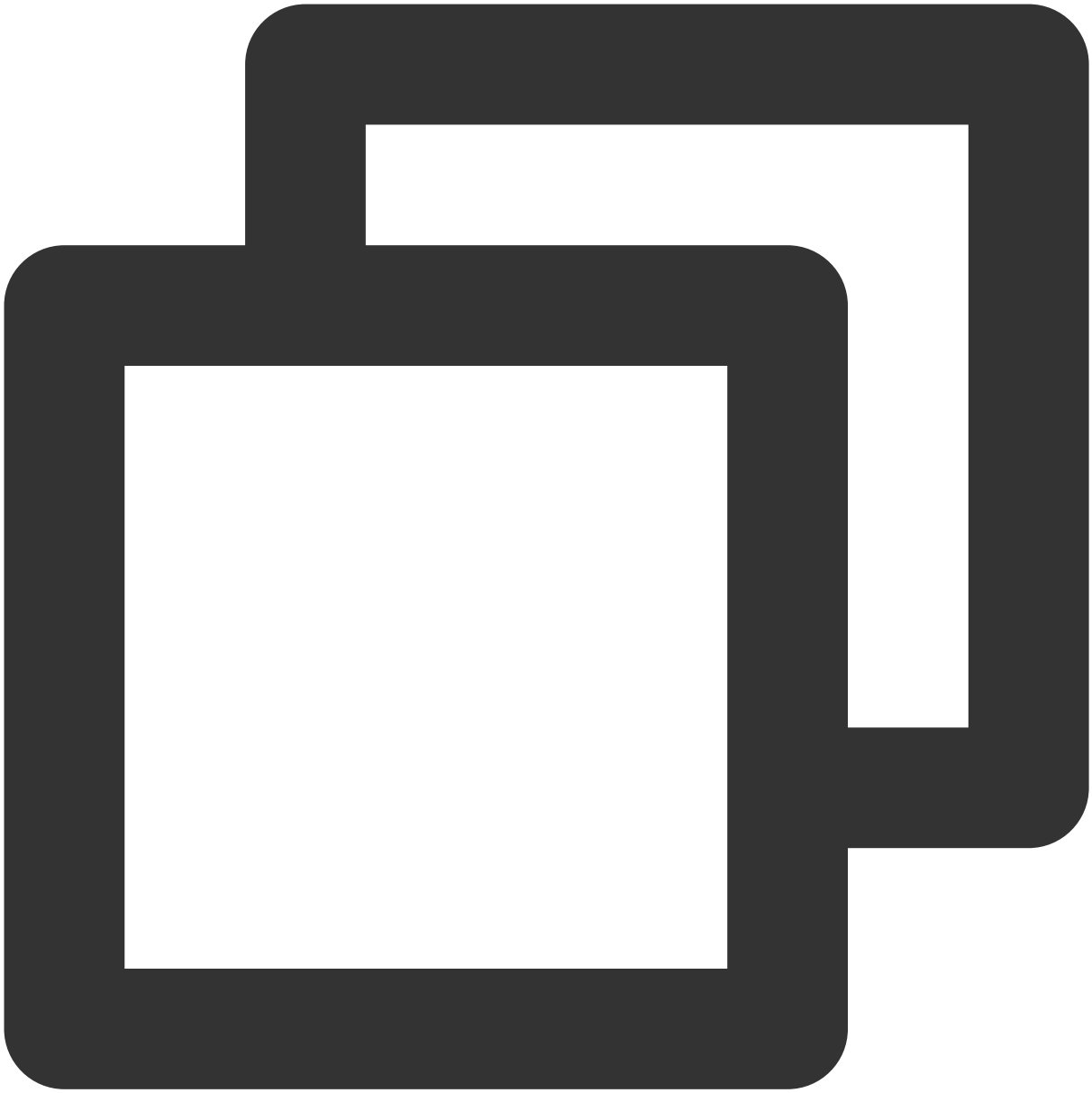


```
rpm -ivh cosfs-1.0.19-centos7.0.x86_64.rpm
```

注意：

如确定依赖包安装完成，但是依然无法安装成功 COSFS 的，可以在上面命令中加入 --force 参数强制安装。

7. 安装完成 COSFS 后，执行以下命令，将 COS 桶挂载到中转服务器中。

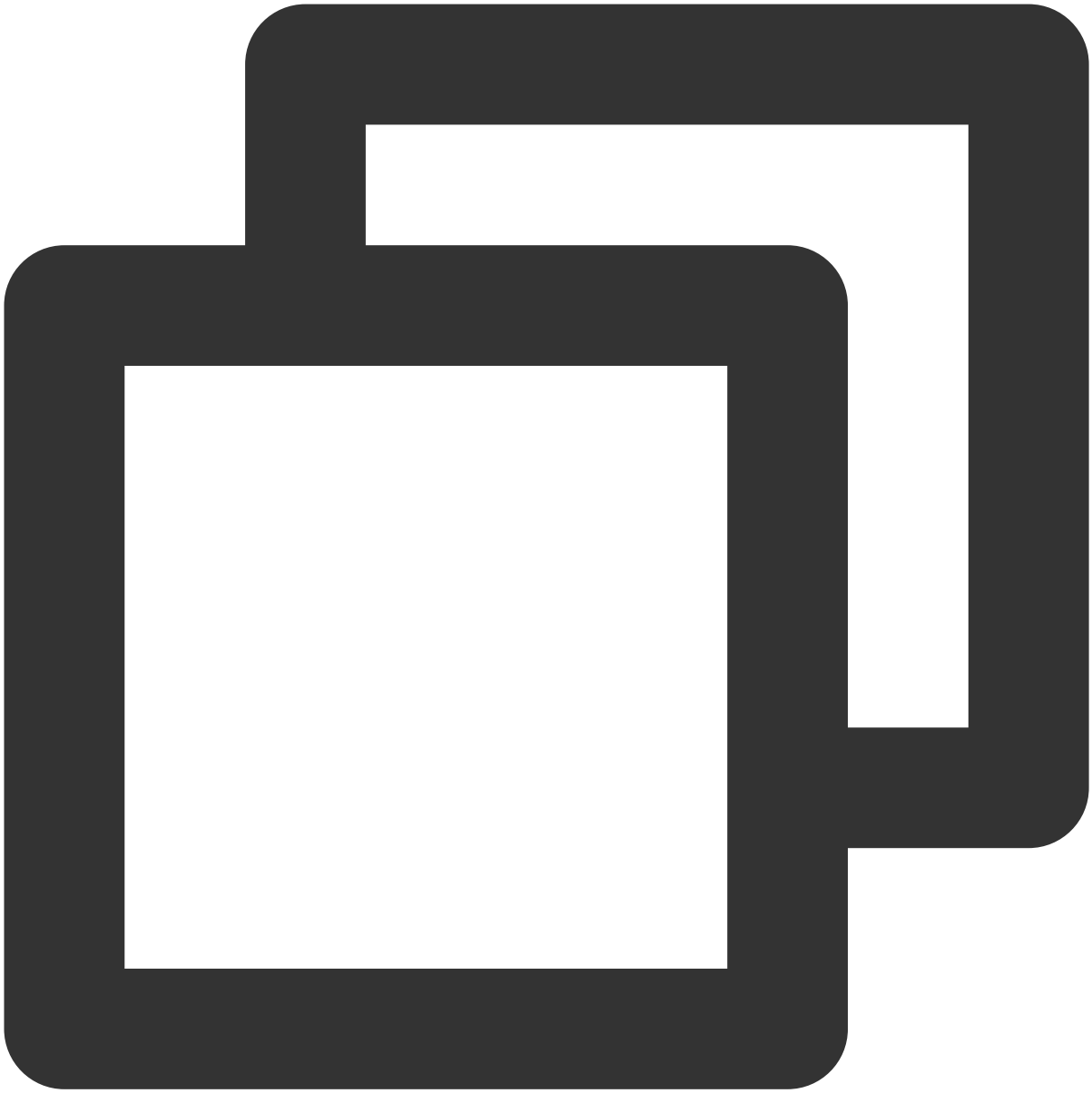


```
echo <BucketName-APPID>:<SecretId>:<SecretKey> > /etc/passwd-cosfs
chmod 640 /etc/passwd-cosfs
cosfs <BucketName-APPID> <MountPoint> -ourl=http://cos.<Region>.myqcloud.c
om -odbglevel=info -oallow_other
```

BucketName-APPID 为存储桶名称格式。

SecretId 和 SecretKey 为密钥信息。

8. 挂载完成后，可进入到挂载目录中，拷贝一个文件进行测试。查看是否挂载成功。亦可执行 df -h 查看挂载情况：

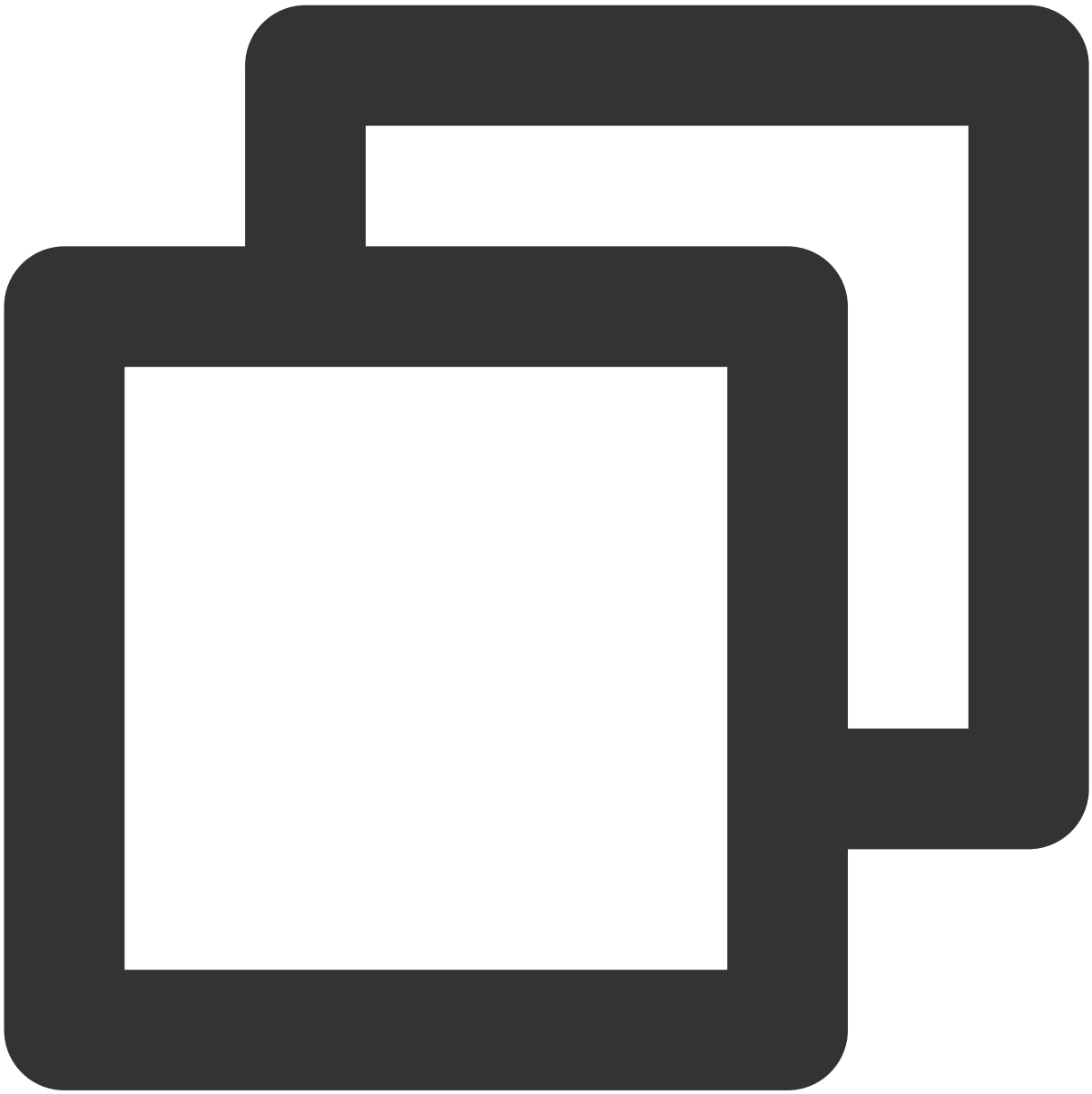


```
[root@VM-4-17-centos ~]# df -h
Filesystem Size Used Avail Use% Mounted on
devtmpfs 1.9G 0 1.9G 0% /dev
tmpfs 1.9G 0 1.9G 0% /dev/shm
tmpfs 1.9G 472K 1.9G 1% /run
tmpfs 1.9G 0 1.9G 0% /sys/fs/cgroup
/dev/vda1 50G 3.0G 44G 7% /
tmpfs 379M 0 379M 0% /run/user/0
cosfs 256T 0 256T 0% /mnt/pgstorage
```

版权所有：腾讯云计算（北京）有限责任公司

导出数据

挂载完成后，即可进行数据导出。
如果存在一张表 sensor_log，表结构如下：

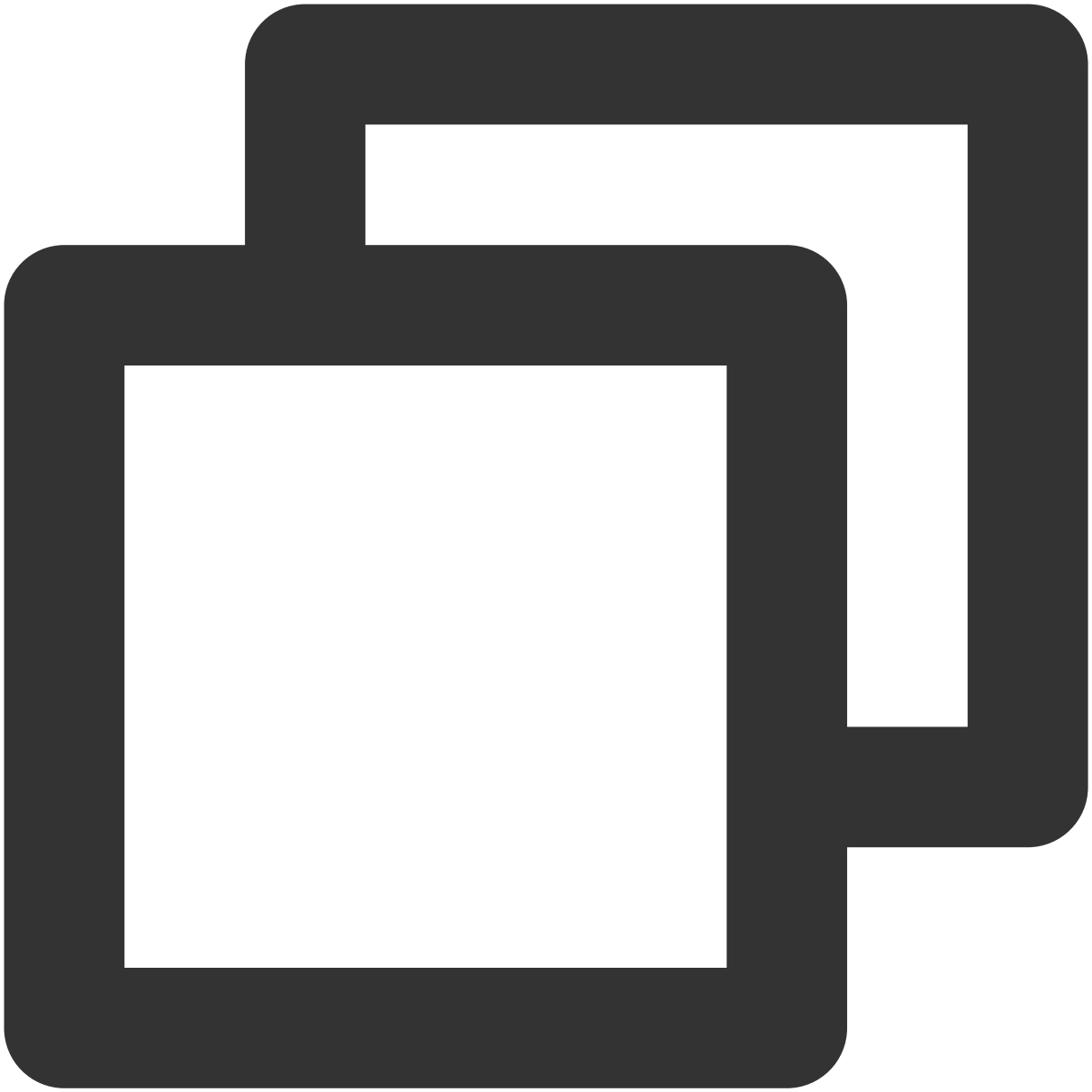


```
CREATE TABLE sensor_log (
 sensor_log_id SERIAL PRIMARY KEY,
 location VARCHAR NOT NULL,
 reading BIGINT NOT NULL,
 reading_date TIMESTAMP NOT NULL
```

```
);
CREATE INDEX idx_sensor_log_location ON sensor_log (location);
CREATE INDEX idx_sensor_log_date ON sensor_log (reading_date);
insert into sensor_log(location,reading,reading_date) values('38c-
1401',293857,current_timestamp);
insert into sensor_log(location,reading,reading_date) values('38c-
1402',293858,current_timestamp);
insert into sensor_log(location,reading,reading_date) values('34c-
1401',293859,current_timestamp);
insert into sensor_log(location,reading,reading_date) values('18c-
1401',2938510,current_timestamp);
```

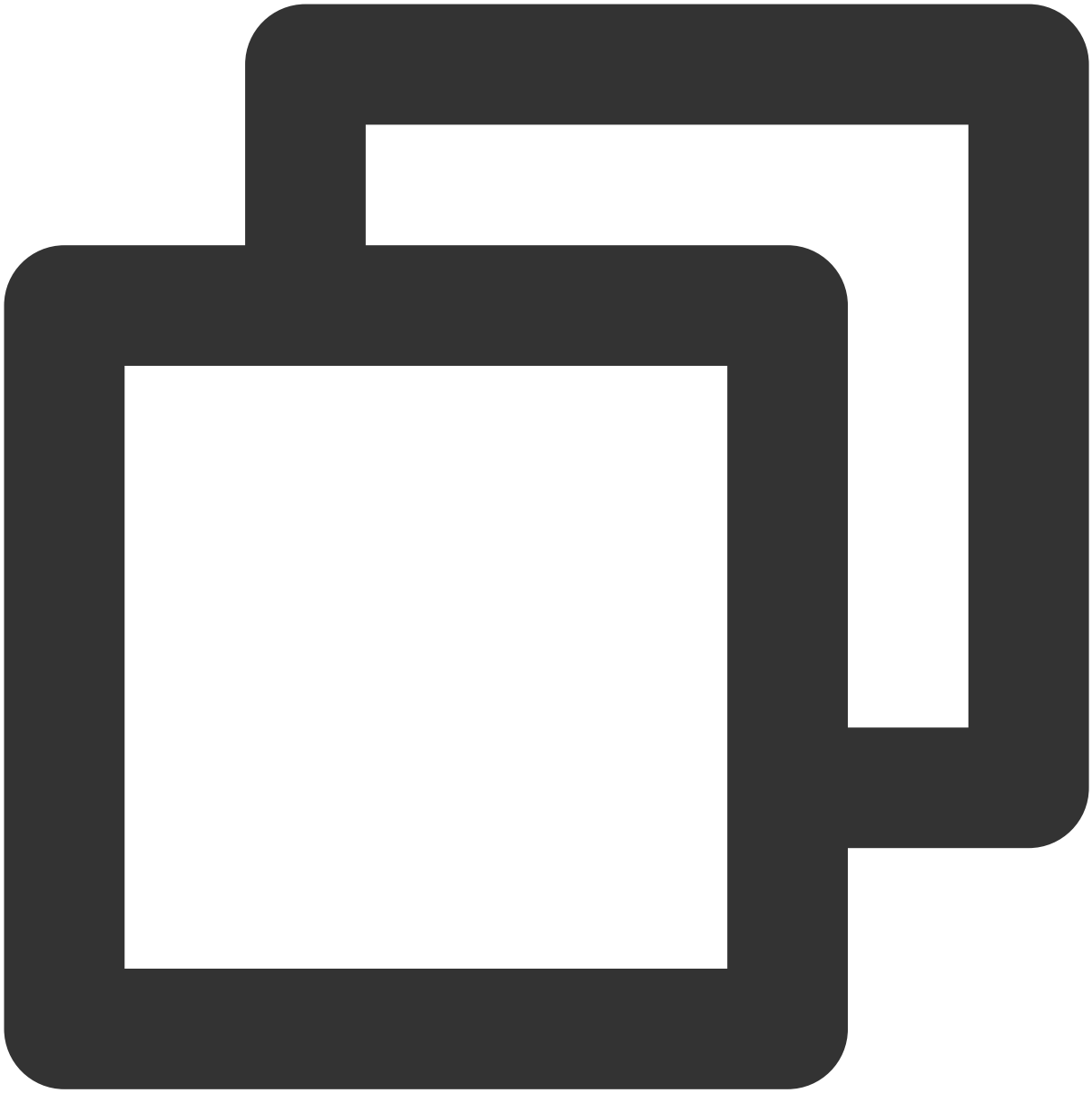
如果使用 psql 客户端进行数据导出，可按照以下流程进行操作，注意导出不要带 header。

导出整张表：



```
psql -U root -p 5432 -h 10.0.4.8 -d hehe -c \\COPY sensor_log
(sensor_log_id,location, reading,reading_date) TO '/mnt/xxx/sensor_log.csv' WITH
csv;
```

指定数据导出（支持数据筛选，过滤，多表联合，视图等场景）：



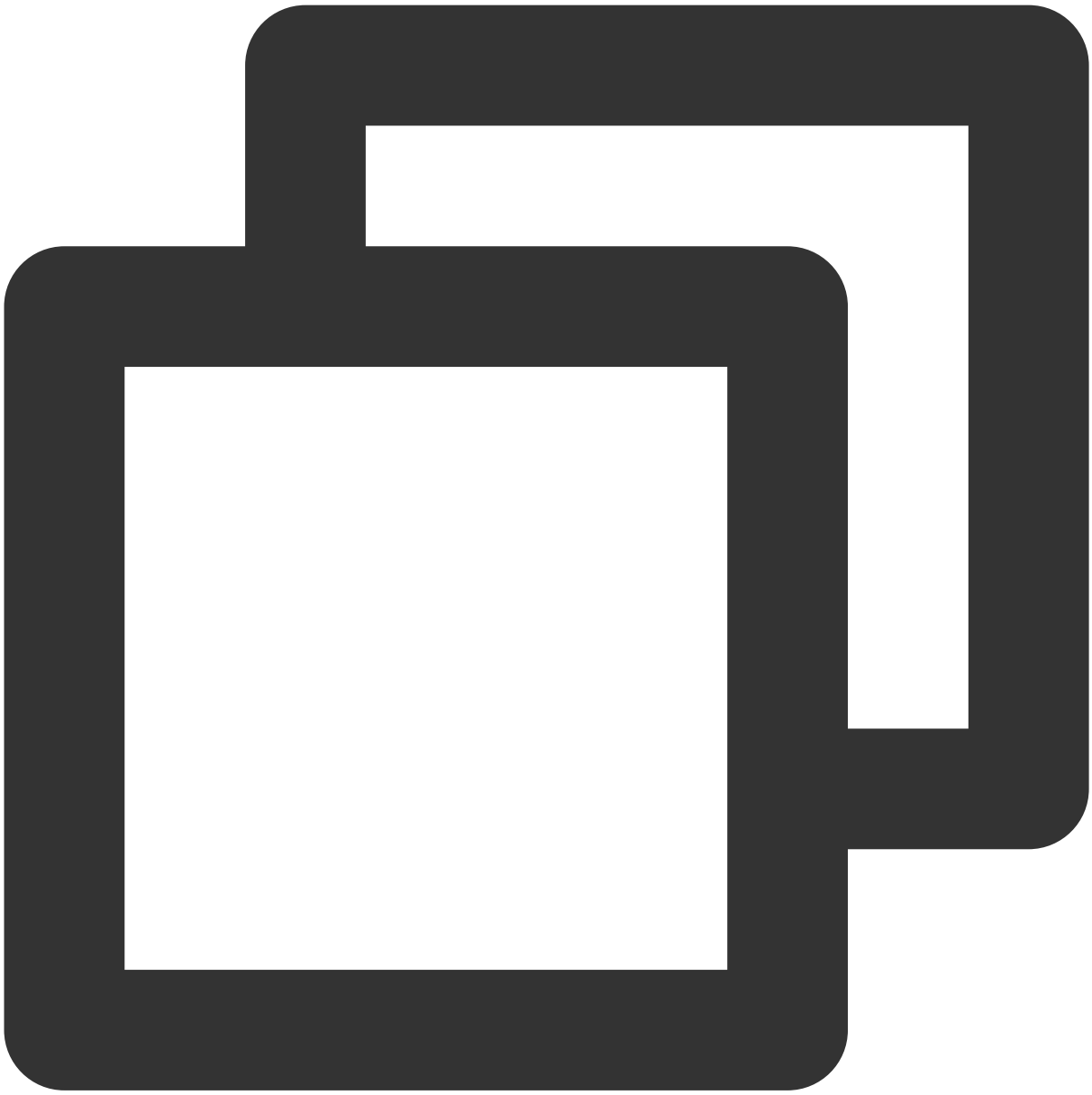
```
psql -U root -p 5432 -h 10.0.4.8 -d hehe -c '\\COPY (select * from sensor_log
where location='18c-1401') TO '/mnt/pgstorage/sensor_log.csv' WITH csv;'
```

上面的语句执行完成后，就可以在 COS 桶对应目录中找到导出的文件。

导入到 COS 的 csv 文件不需要带列名。

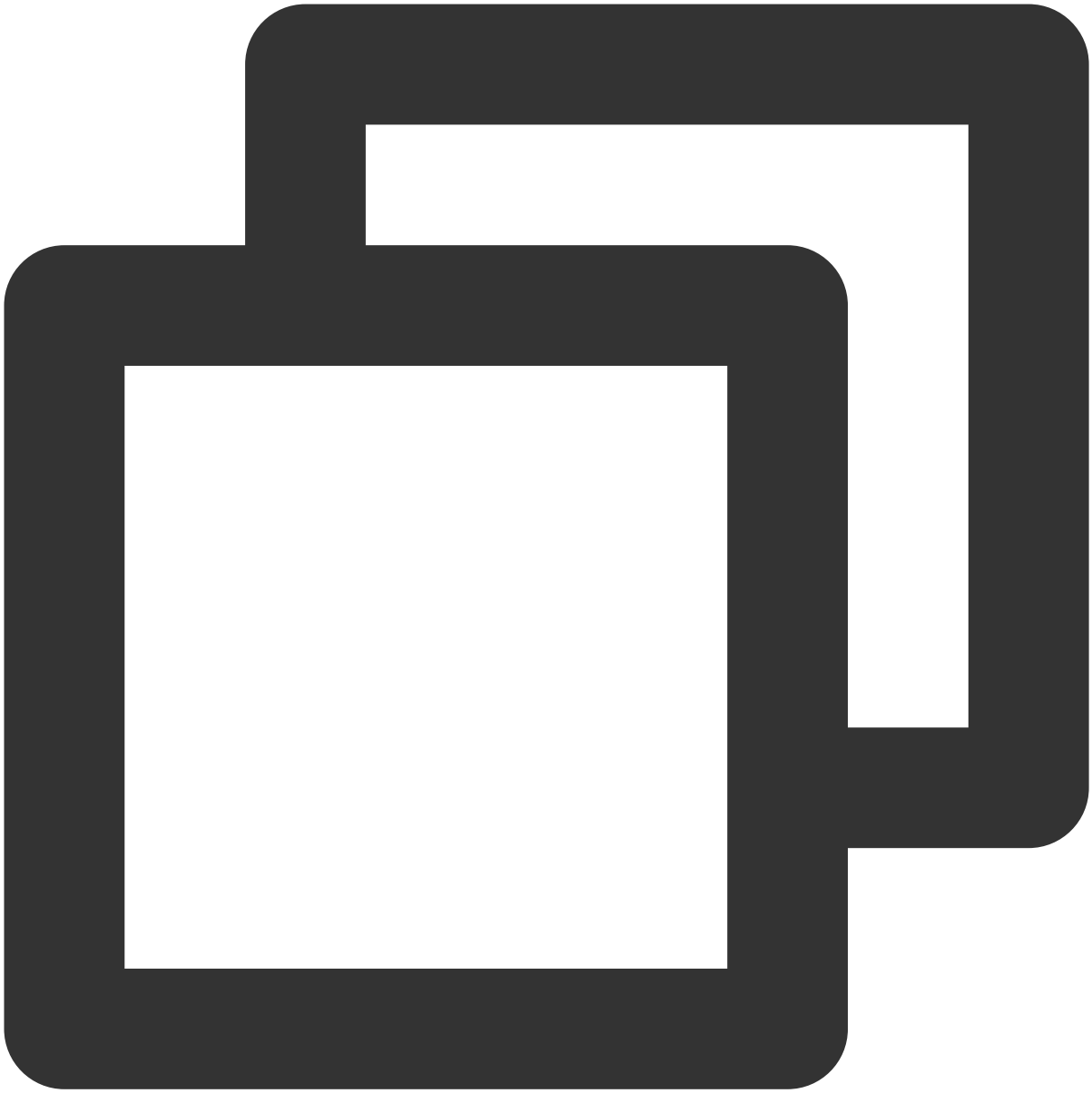
创建插件

cos_fdw 插件会对 COS 的 secret id 和 secret key 进行加密处理，加密算法依赖于 pgcrypto 插件，因此我们在使用时需要先安装 pgcrypto 插件。



```
CREATE EXTENSION pgcrypto;
CREATE EXTENSION cos_fdw;
```

创建 Foreign Server



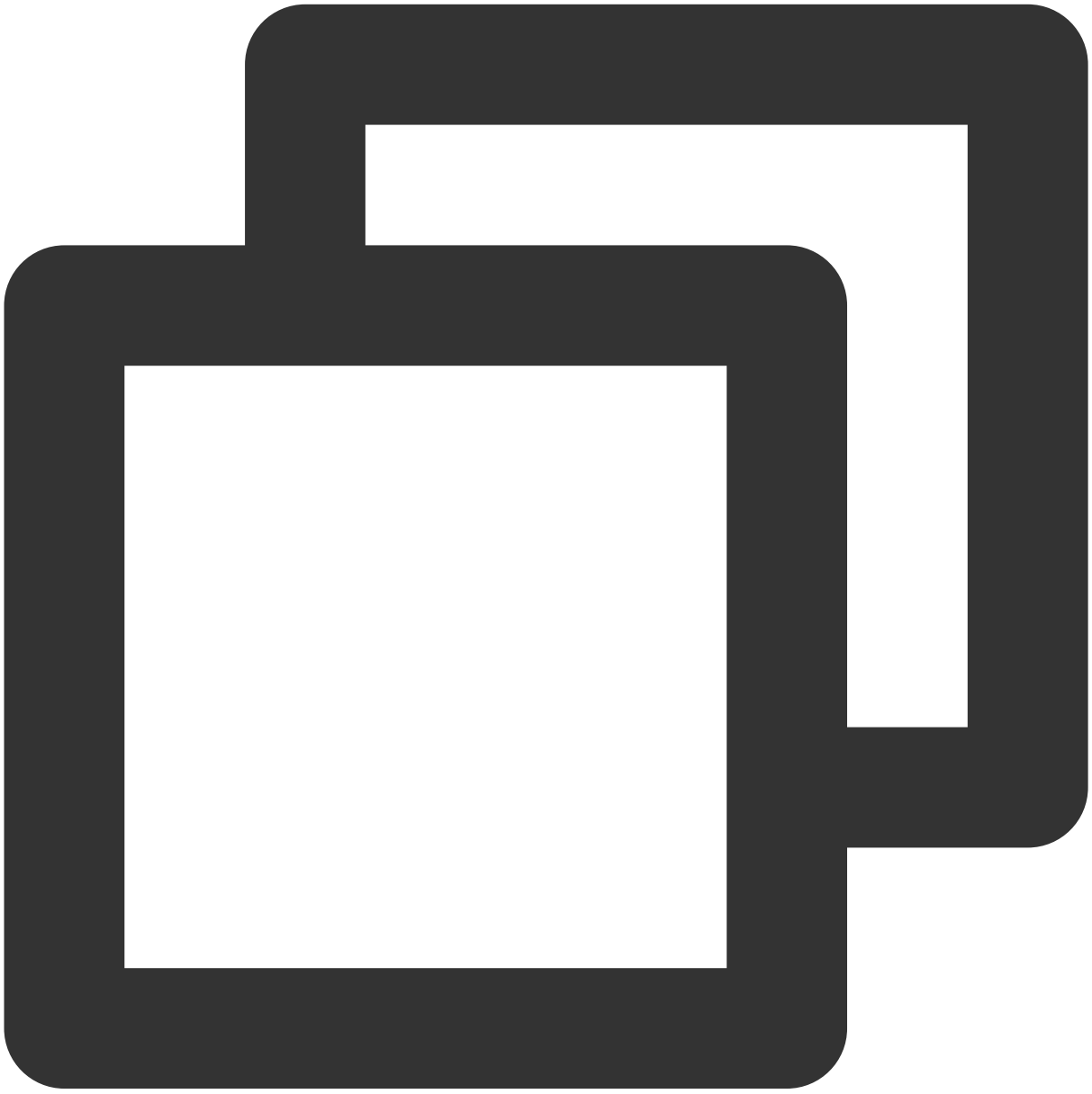
```
CREATE SERVER cos_server FOREIGN DATA WRAPPER cos_fdw OPTIONS(
 host 'xxxxxx.cos.ap-nanjing.myqcloud.com',
 bucket 'xxxxxxxx',
 id 'xxxxxxxx',
 key 'xxxxxxxxxx'
);
```

注意：

host 中配置的域名为 COS 桶的访问地址，地址前缀协议不需要带 http 或 https。

Foreign Server 中的 id 和 key 属于敏感信息，cos_fdw 会对其进行加密存储。不同的实例将会使用不同的密钥，最大限度保护用户信息。我们可以用 `SELECT * FROM pg_foreign_server;` 看到。

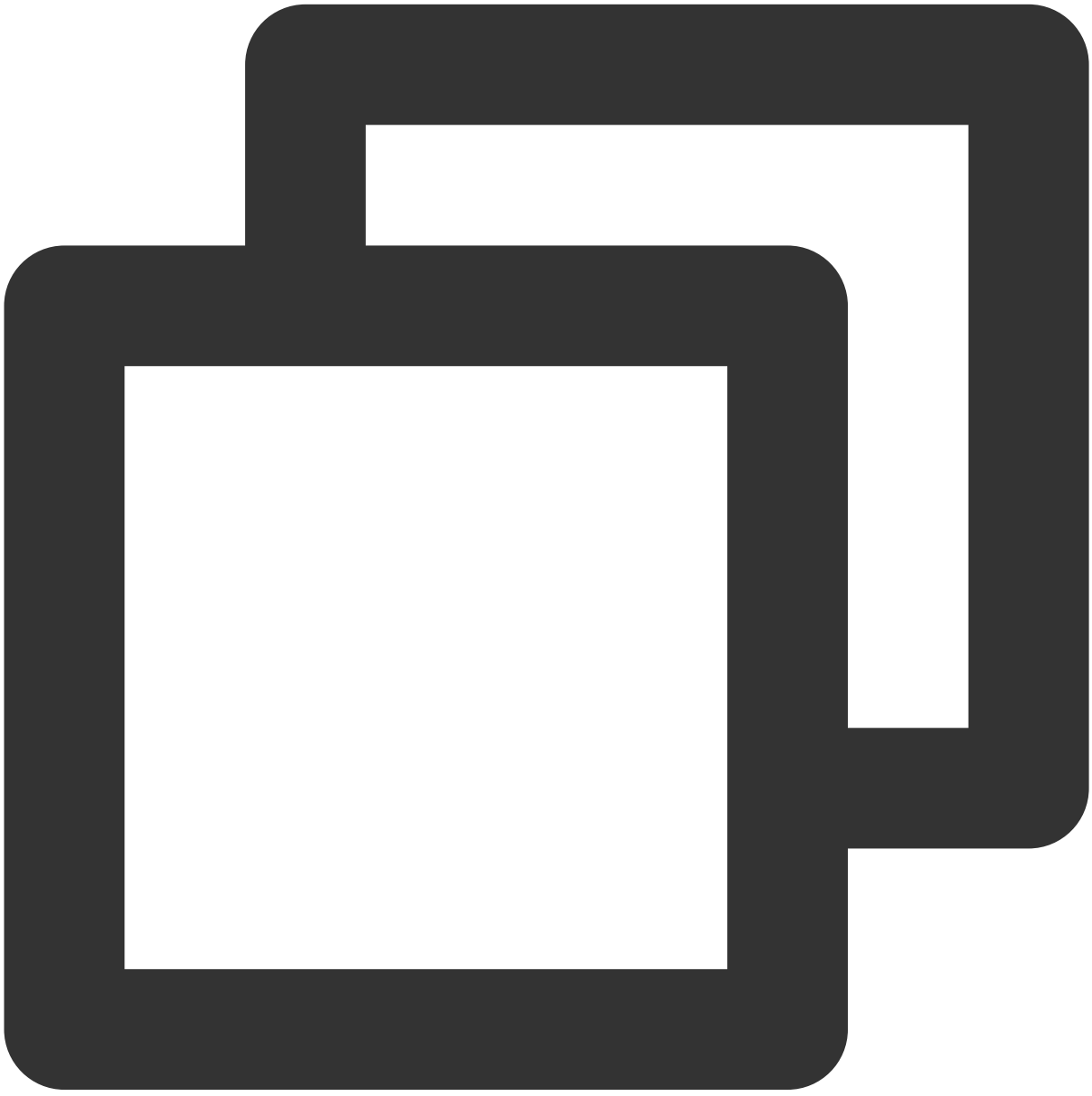
创建 COS 外部表



```
CREATE FOREIGN TABLE test_csv (
 word1 text OPTIONS (force_not_null 'true'),
 word2 text OPTIONS (force_not_null 'off') ) SERVER cos_server OPTIONS (
```

```
 filepath '/test.csv',
 format 'csv',
 null 'NULL'
);
```

cos_fdw 支持将多个 COS 文件可以映射到同一个 FOREIGN TABLE 中，在 filepath 参数中填写多个文件名，每个文件用 `,` 分隔即可（不允许出现多余空格）。



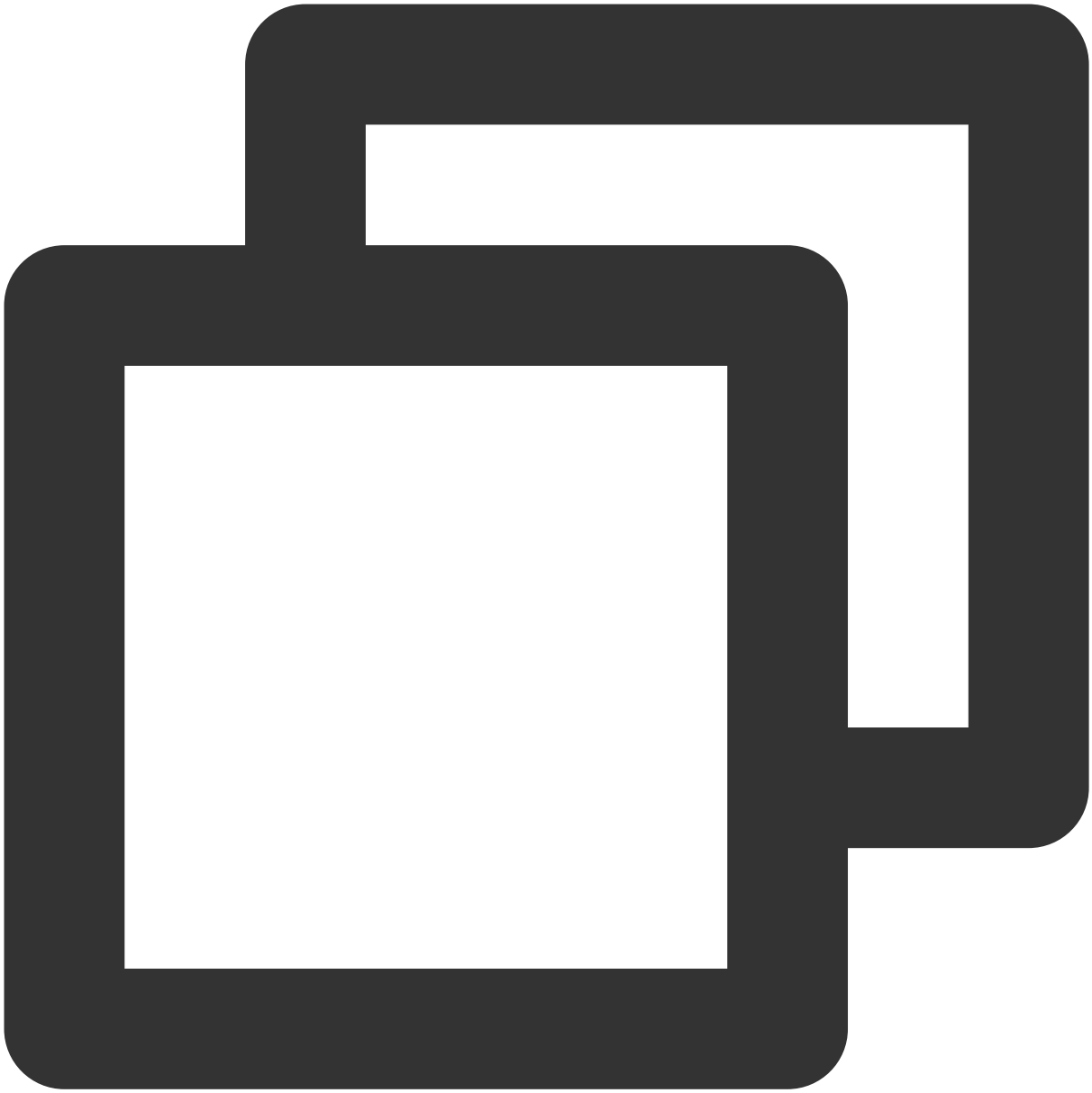
```
CREATE FOREIGN TABLE multi_csv (
 word1 text OPTIONS (force_not_null 'true'),
 word2 text OPTIONS (force_not_null 'off') ) SERVER cos_server OPTIONS (
```

```
  filepath '/a.csv,/b.csv,/c.csv.2',
  format 'csv',
  null 'NULL'
);
```

查询外部表

规划查询计划

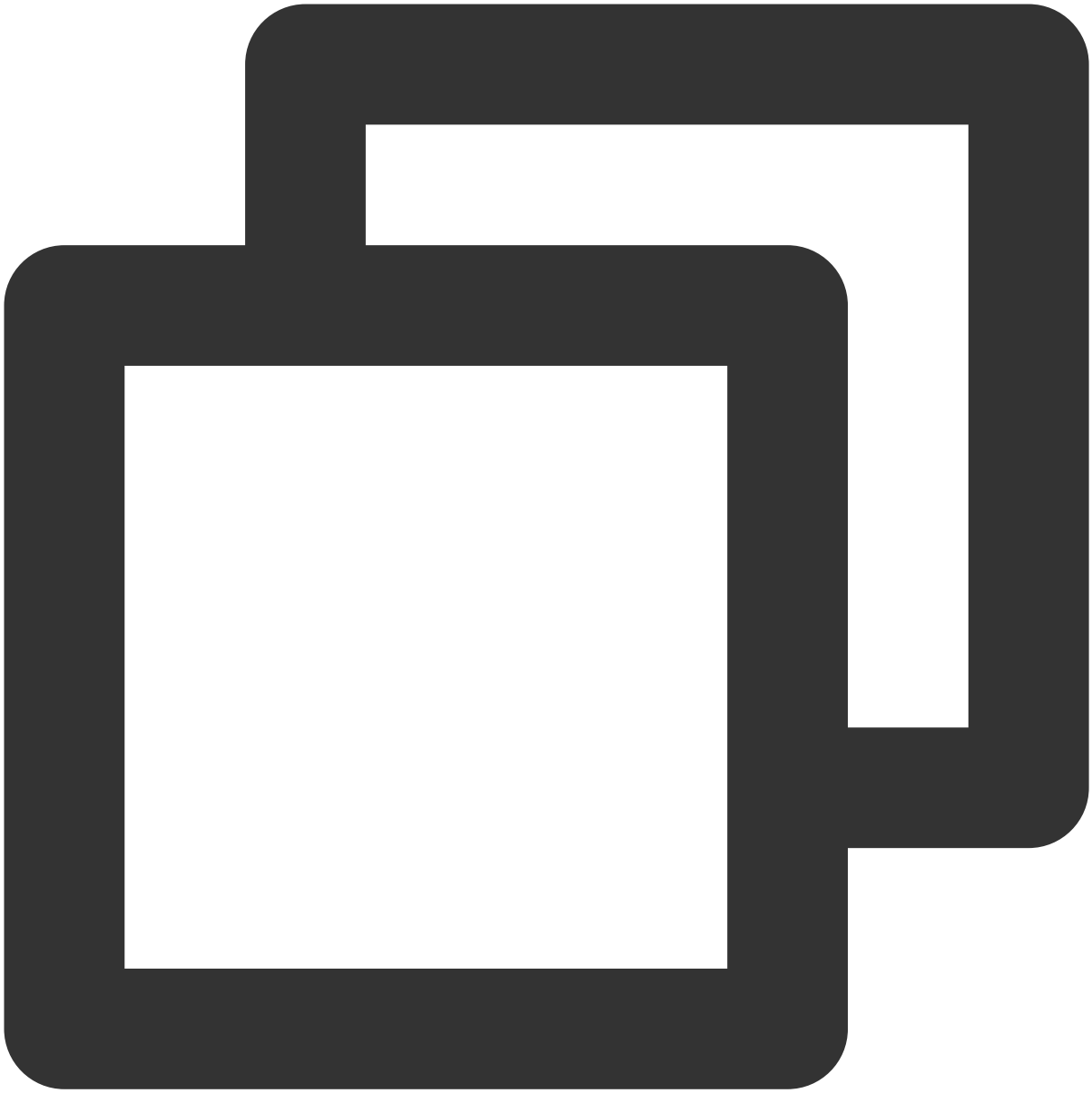
cos_fdw 能够预估外部文件的大小，为查询计划做规划。对于映射了多个 COS 文件的外部表，将会把它们每一个的文件大小打印出来，并计算出来所有文件的总大小。



```
-- 单一文件
postgres=# EXPLAIN SELECT * FROM test_csv;
 QUERY PLAN
-------------------------------------------------------------------
--------------
Foreign Scan on test_csv (cost=0.00..1.10 rows=1 width=128)
 Foreign COS Url: https://xxxxxxx.cos.ap-nanjing.myqcloud.com
 Foreign COS File Path: /test_csv.csv
 Foreign each COS File Size(Bytes): 86
 Foreign total COS File Size(Bytes): 86
(5 rows)
```

```
-- 多个文件
postgres=# EXPLAIN SELECT * FROM multi_csv;
 QUERY PLAN
------------------------------------------------------------------------
---------------
Foreign Scan on multi_csv (cost=0.00..1.20 rows=2 width=128)
 Foreign COS Url: https://xxxxxxxxxx.cos.ap-nanjing.myqcloud.com
 Foreign COS File Path: /a.csv,/b.csv,/c.csv.2
 Foreign each COS File Size(Bytes): 15,172,86
 Foreign total COS File Size(Bytes): 273
(5 rows)
```

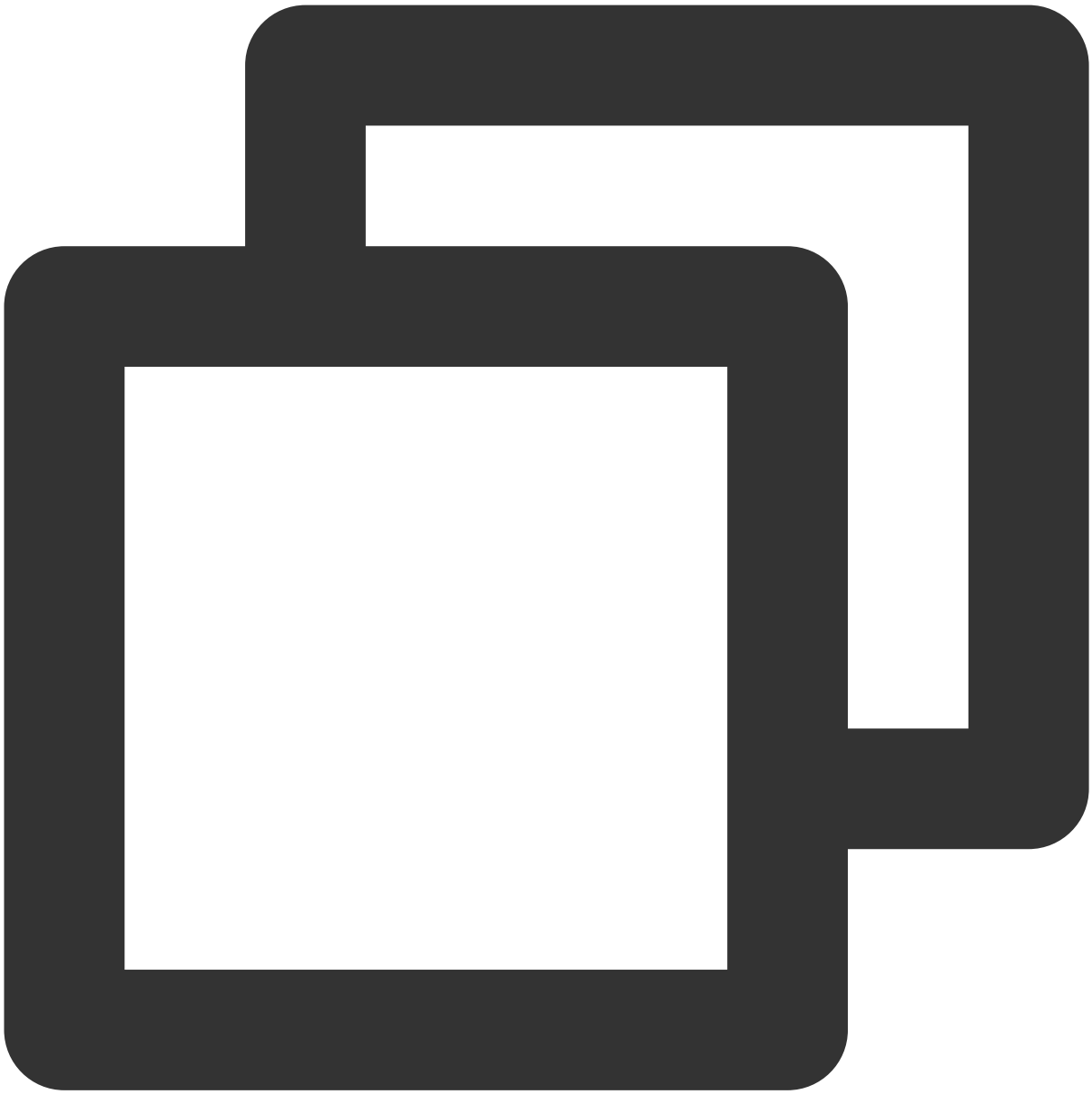
查询数据



```
postgres=# SELECT * FROM test_csv;
word1 | word2 | word3 | word4
-------+-------+-------+-------
AAA | aaa | 123 |
XYZ | xyz | | 321
NULL | | |
NULL | | |
ABC | abc | | (5 rows)
```

将外部表数据导入本地表

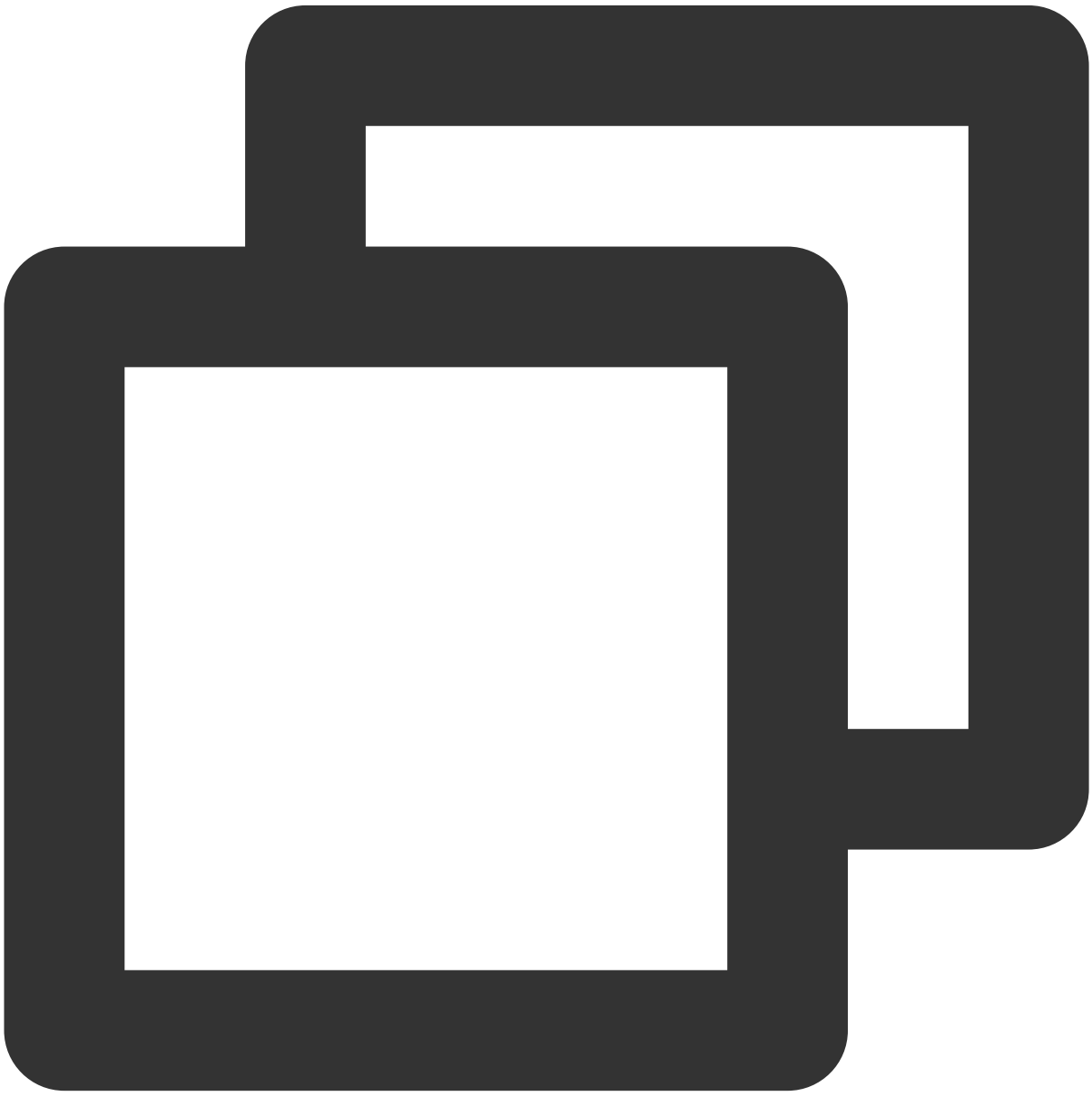
可以使用 `insert into ... select * from ...;` 类似的语句将外部表的数据导入本地表中。



```
postgres=# CREATE TABLE local_test_csv (
postgres(# a text,
postgres(# b text,
postgres(# c text,
postgres(# d text
postgres(# );
CREATE TABLE
postgres=# INSERT INTO local_test_csv SELECT * FROM test_csv;
INSERT 0 5
```

```
postgres=# SELECT * FROM local_test_csv;
 a | b | c | d
---+---+---+---
AAA | aaa | 123 |
XYZ | xyz | | 321
NULL | | |
NULL | | |
ABC | abc | | (5 rows)
```

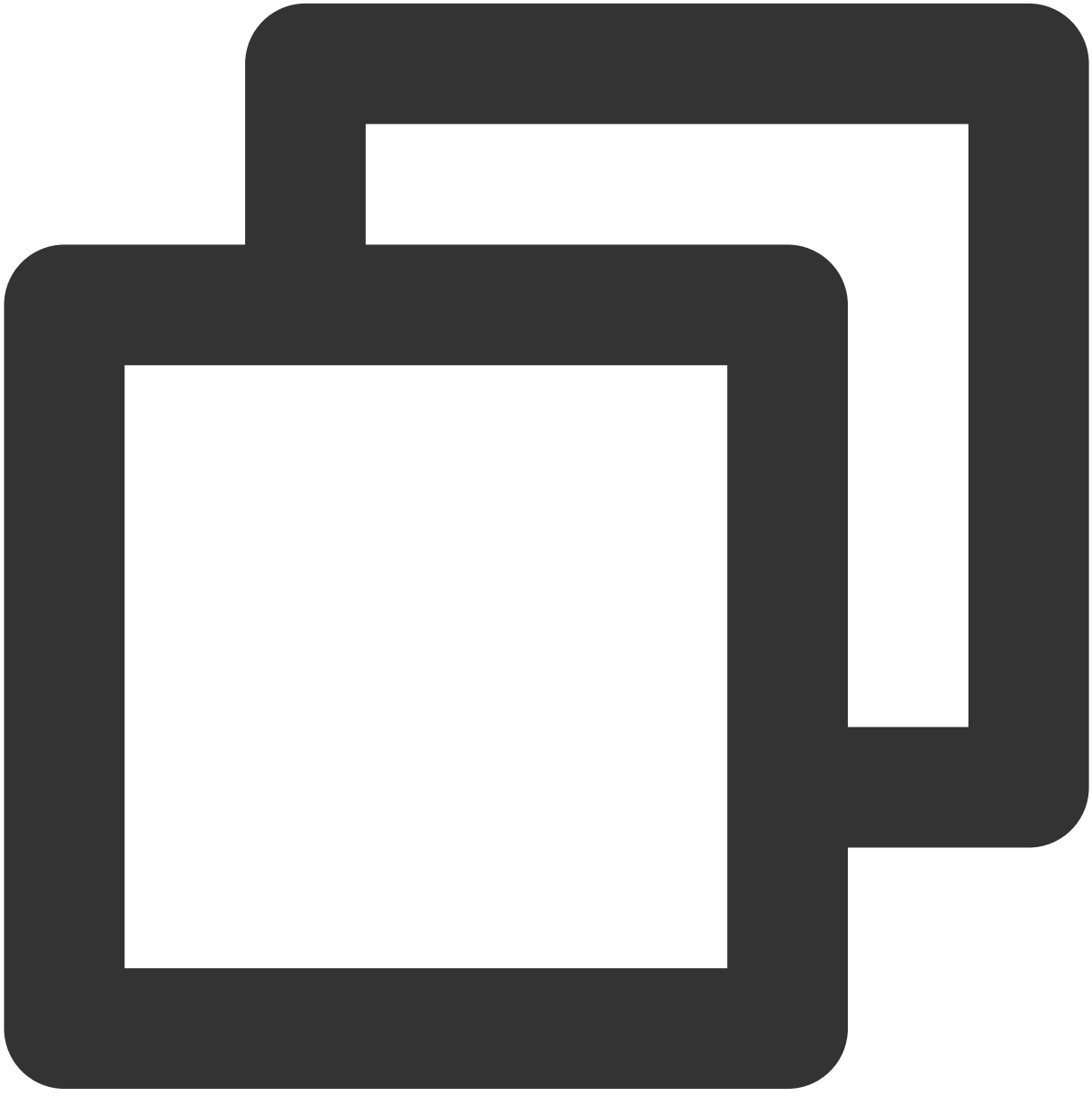
分区表查询



```
postgres=# CREATE TABLE pt (a int, b text) partition by list (a);
CREATE TABLE
postgres=# CREATE FOREIGN TABLE p1 partition of pt for values in (1) SERVER
cos_server
postgres-# OPTIONS (format 'csv', filepath '/list1.csv', delimiter ',');
CREATE FOREIGN TABLE
postgres=# CREATE TABLE p2 partition of pt for values in (2);
CREATE TABLE
-- 分区表支持查询
postgres=# SELECT tableoid::regclass, * FROM pt;
tableoid | a | b
----------+---+-----
p1 | 1 | foo
p1 | 1 | bar
(2 rows)
postgres=# SELECT tableoid::regclass, * FROM p1;
tableoid | a | b
----------+---+-----
p1 | 1 | foo
p1 | 1 | bar
(2 rows)
postgres=# SELECT tableoid::regclass, * FROM p2;
tableoid | a | b
----------+---+---
(0 rows)
-- 目前不支持往外部表中写入数据
postgres=# INSERT INTO pt VALUES (1, 'xyzzy'); -- ERROR
ERROR: cannot route inserted tuples to a foreign table
-- 本地表不受影响，可以正常往分区表中写入
postgres=# INSERT INTO pt VALUES (2, 'xyzzy');
INSERT 0 1
postgres=# SELECT tableoid::regclass, * FROM pt;
tableoid | a | b
----------+---+-------
p1 | 1 | foo
p1 | 1 | bar
p2 | 2 | xyzzy
(3 rows)
postgres=# SELECT tableoid::regclass, * FROM p1;
tableoid | a | b
----------+---+-----
p1 | 1 | foo
p1 | 1 | bar
(2 rows)
postgres=# SELECT tableoid::regclass, * FROM p2;
tableoid | a | b
----------+---+-------
```

```
p2 | 2 | xyzzy
(1 row)
```

删除插件



```
DROP EXTENSION cos_fdw;
```

参数说明

CERATE SERVER 参数

参数	说明
host	内网访问 COS 的地址，注意 host 不包含 http/https 前缀
bucket	存储桶名称，存储桶的命名格式为 BucketName-APPID，此处填写的存储桶名称必须为此格式
id	账号的 secret id
key	账号的 secret key

CREATE FOREIGN TABLE 参数

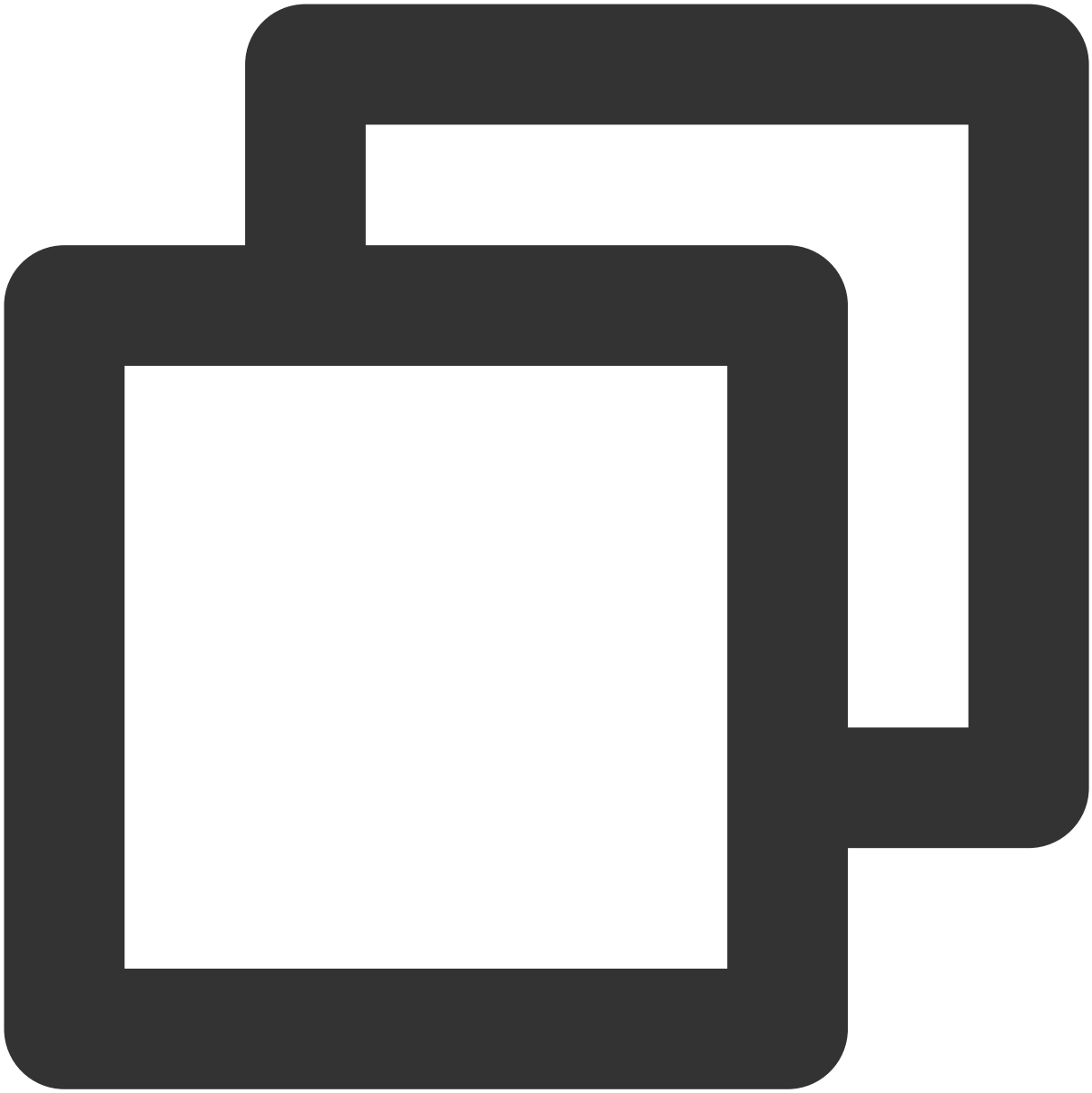
参数	说明
filepath	Sample
format	指定数据的格式，目前仅支持 csv
delimiter	指定数据的分隔符
quote	指定数据的引用字符
escape	指定数据的转义字符
encoding	指定数据的编码
null	指定匹配对应字符串的列为 null，例如 null 'NULL'，即列值为 'NULL' 的字符串为 null
force_not_null	指定该列的值不应该与空字符串匹配。例如，force_not_null 'id' 表示：如果 id 列的值为空，则该值为空字符串，而不是 null
force_null	指定该列的值与空字符串匹配。例如，force_null 'id' 表示：如果 id 列的值为空，则该值为 null

错误处理

当使用 cos_fdw 向 COS 请求数据超时，会显示以下内容：

code：出错请求的 HTTP 状态码。

错误请求的 HTTP header：显示错误的信息。其格式参见 公共响应头部。其中 `x-cos-request-id` 可以用于寻求 提交工单 排查问题。如果该项为空，表示未成功向 COS 发送请求。



• postgres=# SELECT * FROM test_csv; • ERROR: COS api return error. • DETAIL: COS a
• HTTP/1.1 403 Forbidden
• Content-Type: application/xml
• Content-Length: 0 • Connection: keep-alive
• Date: Thu, 07 Apr 2022 09:00:22 GMT
• Server: tencent-cos
• x-cos-request-id: NjI0ZWE4MjZfNDc1NGU0MDlfMjI3ZTJfMTI3YTJjMWM=
• x-cos-trace-id:
OGVmYzZiMmQzYjA2OWNhODk0NTRkMTBiOWVmMDAxODc0OWRkZjk0ZDM1NmI1M2E2MTRlY2MzZDhmNmI5MWI

通过 pgpool 实现读写分离

最近更新时间：2024-03-20 15:49:10

1、安装 pgpool

下载 pgpool 并进行安装，下载 地址。

```
$ ./configure
$ make
$ make install
```

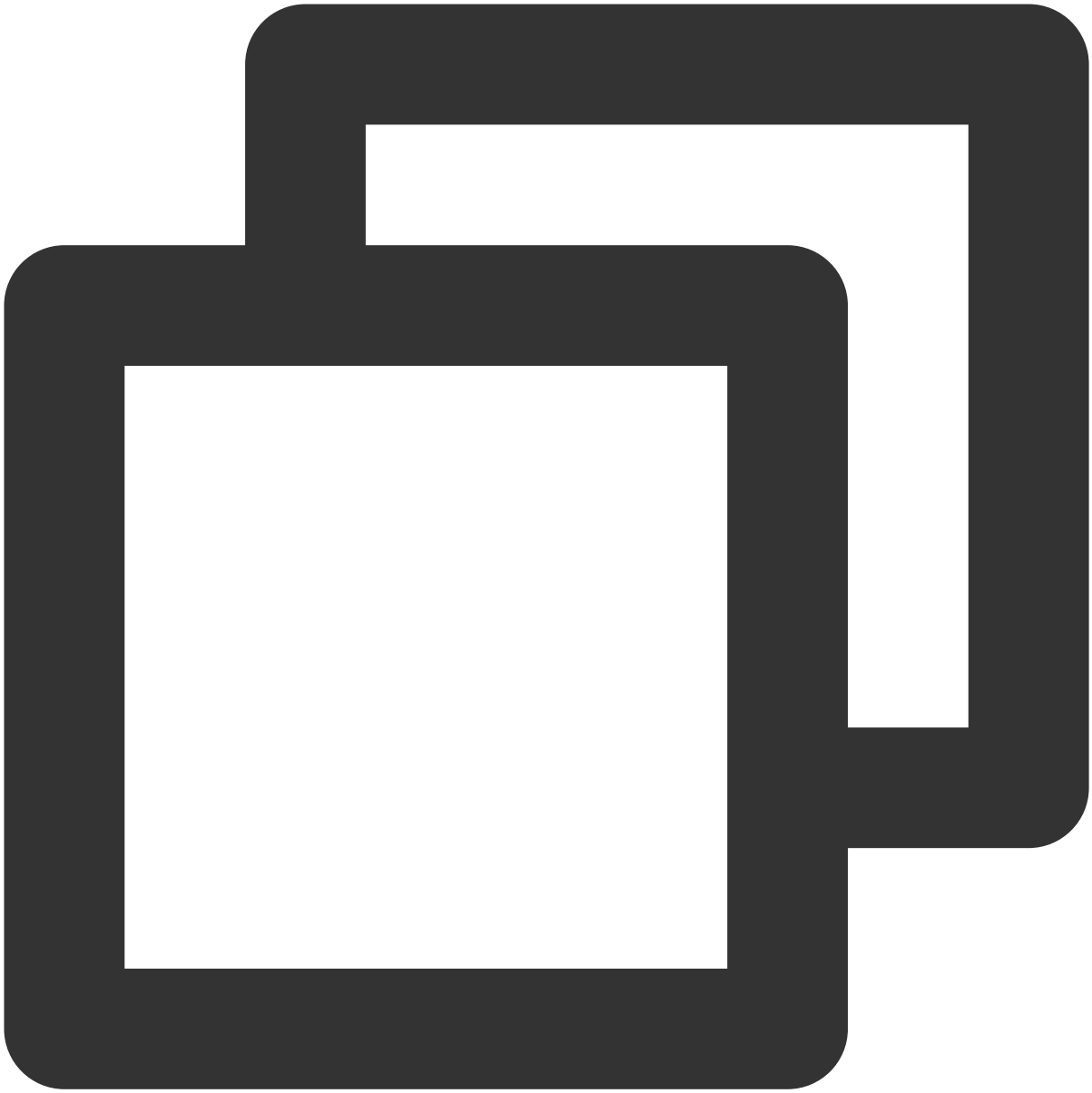
2、配置文件

说明：

使用 pgpool 实现负载均衡访问，所有认证发生在客户端和 pgpool 之间，同时客户端仍然需要继续通过 PostgreSQL 的认证过程。

配置 pool_passwd 密码文件

pool_passwd 密码文件是通过 pgpool 连接数据库时需要使用密码文件。可以使用如下命令生成密码文件：



```
[root@VM-0-15-tencentos ~]# cd /usr/local/bin
[root@VM-0-15-tencentos bin]# pg_md5 --md5auth --username=dbadmin password
[root@VM-0-15-tencentos bin]# more /usr/local/etc/pool_passwd
dbadmin:md50b0cdb5c1d1f30fe83e5a72061749681
```

配置 pgpool.conf 文件

当你安装 pgpool-II 后， pgpool.conf.sample 被自动建立。我们建议拷贝或者重命名它为 pgpool.conf ，然后你可以随意编辑它。

```
$ cp /usr/local/etc/pgpool.conf.sample /usr/local/etc/pgpool.conf
```

pgpool-II 默认只接受到 9999 端口的本地连接。如果你希望从其他主机接受连接，请设置

```
listen_addresses = 'localhost'
port = 9999
```

重要的 pgpool 配置如下，请参考：

```
#------------------------------------------------------------------------------
# BACKEND CLUSTERING MODE
# Choose one of: 'streaming_replication', 'native_replication',
#      'logical_replication', 'slony', 'raw' or 'snapshot_isolation'
# (change requires restart)
#------------------------------------------------------------------------------
backend_clustering_mode = 'streaming_replication'
#------------------------------------------------------------------------------
# CONNECTIONS
#------------------------------------------------------------------------------
# - pgpool Connection Settings -
listen_addresses = '0.0.0.0'
                                    # what host name(s) or IP address(es) to listen
on;
                                    # comma-separated list of addresses;
                                    # defaults to 'localhost'; use '*' for all
                                    # (change requires restart)
port = 9989
                                    # Port number
                                    # (change requires restart)
unix_socket_directories = '/tmp'
                                    # Unix domain socket path(s)
                                    # The Debian package defaults to
                                    # /var/run/postgresql
                                    # (change requires restart)
#unix_socket_group = ''
                                    # The Owner group of Unix domain socket(s)
                                    # (change requires restart)
reserved_connections = 0
                                    # Number of reserved connections.
                                    # Pgpool-II does not accept connections if over
                                    # num_init_chidlren - reserved_connections.
# - pgpool Communication Manager Connection Settings -
```

```
pcp_listen_addresses = ''
                                   # what host name(s) or IP address(es) for pcp
process to listen on;
                                   # comma-separated list of addresses;
                                   # defaults to 'localhost'; use '*' for all
                                   # (change requires restart)
pcp_port = 9898
                                   # Port number for pcp
                                   # (change requires restart)
pcp_socket_dir = '/tmp'
                                   # Unix domain socket path for pcp
                                   # The Debian package defaults to
                                   # /var/run/postgresql
                                   # (change requires restart)
listen_backlog_multiplier = 2
                                   # Set the backlog parameter of listen(2) to
                                   # num_init_children * listen_backlog_multiplier.
                                   # (change requires restart)
serialize_accept = off
                                   # whether to serialize accept() call to avoid
thundering herd problem
                                   # (change requires restart)
# - Backend Connection Settings -
backend_hostname0 = '172.16.0.3'
                                   # Host name or IP address to connect to for
backend 0
backend_port0 = 5432
                                   # Port number for backend 0
backend_weight0 = 1
                                   # Weight for backend 0 (only in load balancing
mode)
#backend_data_directory0 = '/data'
                                   # Data directory for backend 0
backend_flag0 = 'ALWAYS_PRIMARY'
                                   # Controls various backend behavior
                                   # ALLOW_TO_FAILOVER, DISALLOW_TO_FAILOVER
```

```
                                    # or ALWAYS_PRIMARY
backend_application_name0 = 'server0'
                                    # walsender's application_name, used for "show
pool_nodes" command
backend_hostname1 = '172.16.0.12'
backend_port1 = 5432
backend_weight1 = 1
#backend_data_directory1 = '/data1'
backend_flag1 = 'DISALLOW_TO_FAILOVER'
backend_application_name1 = 'server1'
# - Authentication -
enable_pool_hba = on
                                    # Use pool_hba.conf for client authentication
pool_passwd = 'pool_passwd'
                                    # File name of pool_passwd for md5
authentication.
                                    # "" disables pool_passwd.
                                    # (change requires restart)
allow_clear_text_frontend_auth = off
                                    # Allow Pgpool-II to use clear text password
authentication
                                    # with clients, when pool_passwd does not
                                    # contain the user password
# - SSL Connections -
ssl =off
                                    # Enable SSL support
                                    # (change requires restart)
#------------------------------------------------------------------------------
# POOLS
#------------------------------------------------------------------------------
num_init_children = 32
                                    # Maximum Number of concurrent sessions allowed
                                    # (change requires restart)
max_pool = 4
                                    # Number of connection pool caches per
connection
```

```
                                        # (change requires restart)
# - Life time -
child_life_time = 5min
                                        # Pool exits after being idle for this many
seconds
child_max_connections = 0
                                        # Pool exits after receiving that many
connections
                                        # 0 means no exit
connection_life_time = 0
                                        # Connection to backend closes after being idle
for this many seconds
                                        # 0 means no close
client_idle_limit = 0
                                        # Client is disconnected after being idle for
that many seconds
                                        # (even inside an explicit transactions!)
                                        # 0 means no disconnection
#------------------------------------------------------------------------------
# FILE LOCATIONS
#------------------------------------------------------------------------------
pid_file_name = '/var/run/pgpool/pgpool.pid'
                                        # PID file name
                                        # Can be specified as relative to the"
                                        # location of pgpool.conf file or
                                        # as an absolute path
                                        # (change requires restart)
logdir = '/tmp'
                                        # Directory of pgPool status file
                                        # (change requires restart)
#------------------------------------------------------------------------------
# CONNECTION POOLING
#------------------------------------------------------------------------------
connection_cache = on
                                        # Activate connection pools
                                        # (change requires restart)
```

```
                                        # Semicolon separated list of queries
                                        # to be issued at the end of a session
                                        # The default is for 8.3 and later
reset_query_list = 'ABORT; DISCARD ALL'
                                        # The following one is for 8.2 and before
#------------------------------------------------------------------------------
# LOAD BALANCING MODE
#------------------------------------------------------------------------------
load_balance_mode = on
                                        # Activate load balancing mode
                                        # (change requires restart)
ignore_leading_white_space = on
                                        # Ignore leading white spaces of each query
write_function_list = ''
                                        # Comma separated list of function names
                                        # that write to database
                                        # Regexp are accepted
                                        # If both read_only_function_list and
write_function_list
                                        # is empty, function's volatile property is
checked.
                                        # If it's volatile, the function is regarded as
a
                                        # writing function.
allow_sql_comments = off
                                        # if on, ignore SQL comments when judging if
load balance or
                                        # query cache is possible.
                                        # If off, SQL comments effectively prevent the
judgment
                                        # (pre 3.4 behavior).
disable_load_balance_on_write = 'transaction'
                                        # Load balance behavior when write query is
issued
                                        # in an explicit transaction.
                                        #
```

```
                                        # Valid values:
                                        #
                                        # 'transaction' (default):
                                        #     if a write query is issued, subsequent
                                        #     read queries will not be load balanced
                                        #     until the transaction ends.
                                        #
                                        # 'trans_transaction':
                                        #     if a write query is issued, subsequent
                                        #     read queries in an explicit transaction
                                        #     will not be load balanced until the
session ends.
                                        #
                                        # 'dml_adaptive':
                                        #     Queries on the tables that have already
been
                                        #     modified within the current explicit
transaction will
                                        #     not be load balanced until the end of the
transaction.
                                        #
                                        # 'always':
                                        #     if a write query is issued, read queries
will
                                        #     not be load balanced until the session
ends.
                                        #
                                        # Note that any query not in an explicit
transaction
                                        # is not affected by the parameter except
'always'.
 statement_level_load_balance = off
                                        # Enables statement level load balancing
 #------------------------------------------------------------------------------
 # HEALTH CHECK GLOBAL PARAMETERS
 #------------------------------------------------------------------------------
```

```
health_check_period = 0
                                        # Health check period
                                        # Disabled (0) by default
health_check_timeout = 20
                                        # Health check timeout
                                        # 0 means no timeout
health_check_user = 'nobody'
                                        # Health check user
health_check_password = ''
                                        # Password for health check user
                                        # Leaving it empty will make Pgpool-II to first
look for the
                                        # Password in pool_passwd file before using the
empty password
health_check_database = ''
                                        # Database name for health check. If '', tries
'postgres' frist,
health_check_max_retries = 60
                                        # Maximum number of times to retry a failed
health check before giving up.
health_check_retry_delay = 1
                                        # Amount of time to wait (in seconds) between
retries.
connect_timeout = 10000
                                        # Timeout value in milliseconds before giving up
to connect to backend.
                                        # Default is 10000 ms (10 second). Flaky network
user may want to increase
                                        # the value. 0 means no timeout.
                                        # Note that this value is not only used for
health check,
                                        # but also for ordinary conection to backend.
```

3、配置 PCP 命令

pgpool-II 有一个用于管理功能的接口，用于通过网络获取数据库节点信息、关闭 pgpool-II 等。要使用 PCP 命令，必须进行用户认证。这种认证和 PostgreSQL 的用户认证不同。这需要在 pcp.conf 文件中定义一个用户和密码。在这个文件中，一个用户名和密码成对地出现在每一行中，它们用冒号（:）隔开。密码为用 md5 哈希加密的格式。

4、配置数据库节点

```
# - Backend Connection Settings -
backend_hostname0 = '172.16.0.30'
                                    # Host name or IP address to connect to for
backend 0
backend_port0 = 5432
                                    # Port number for backend 0
backend_weight0 = 1
                                    # Weight for backend 0 (only in load balancing
mode)
#backend_data_directory0 = '/data'
                                    # Data directory for backend 0
backend_flag0 = 'ALWAYS_PRIMARY'
                                    # Controls various backend behavior
                                    # ALLOW_TO_FAILOVER, DISALLOW_TO_FAILOVER
                                    # or ALWAYS_PRIMARY
backend_application_name0 = 'server0'
                                    # walsender's application_name, used for "show
pool_nodes" command
backend_hostname1 = '172.16.0.16'
backend_port1 = 5432
backend_weight1 = 1
#backend_data_directory1 = '/data1'
backend_flag1 = 'DISALLOW_TO_FAILOVER'
backend_application_name1 = 'server1'
```

当 load_balance_mode 被设置为 true，pgpool-II 将在数据库节点之间分发 SELECT 查询。

```
load_balance_mode = on
                                    # Activate load balancing mode
                                    # (change requires restart)
ignore_leading_white_space = on
```

```
                                        # Ignore leading white spaces of each query
 write_function_list = ''
                                        # Comma separated list of function names
                                        # that write to database
                                        # Regexp are accepted
                                        # If both read_only_function_list and
write_function_list
                                        # is empty, function's volatile property is
checked.
                                        # If it's volatile, the function is regarded as
a
                                        # writing function.
 allow_sql_comments = off
                                        # if on, ignore SQL comments when judging if
load balance or
                                        # query cache is possible.
                                        # If off, SQL comments effectively prevent the
judgment
                                        # (pre 3.4 behavior).
 disable_load_balance_on_write = 'transaction'
                                        # Load balance behavior when write query is
issued
                                        # in an explicit transaction.
                                        #
                                        # Valid values:
                                        #
                                        # 'transaction' (default):
                                        #     if a write query is issued, subsequent
                                        #     read queries will not be load balanced
                                        #     until the transaction ends.
                                        #
                                        # 'trans_transaction':
                                        #     if a write query is issued, subsequent
                                        #     read queries in an explicit transaction
                                        #     will not be load balanced until the
session ends.
```

```
                                        #
                                        # 'dml_adaptive':
                                        #     Queries on the tables that have already
been
                                        #     modified within the current explicit
transaction will
                                        #     not be load balanced until the end of the
transaction.
                                        #
                                        # 'always':
                                        #     if a write query is issued, read queries
will
                                        #     not be load balanced until the session
ends.
                                        #
                                        # Note that any query not in an explicit
transaction
                                        # is not affected by the parameter except
'always'.
 statement_level_load_balance = off
                                        # Enables statement level load balancing
```

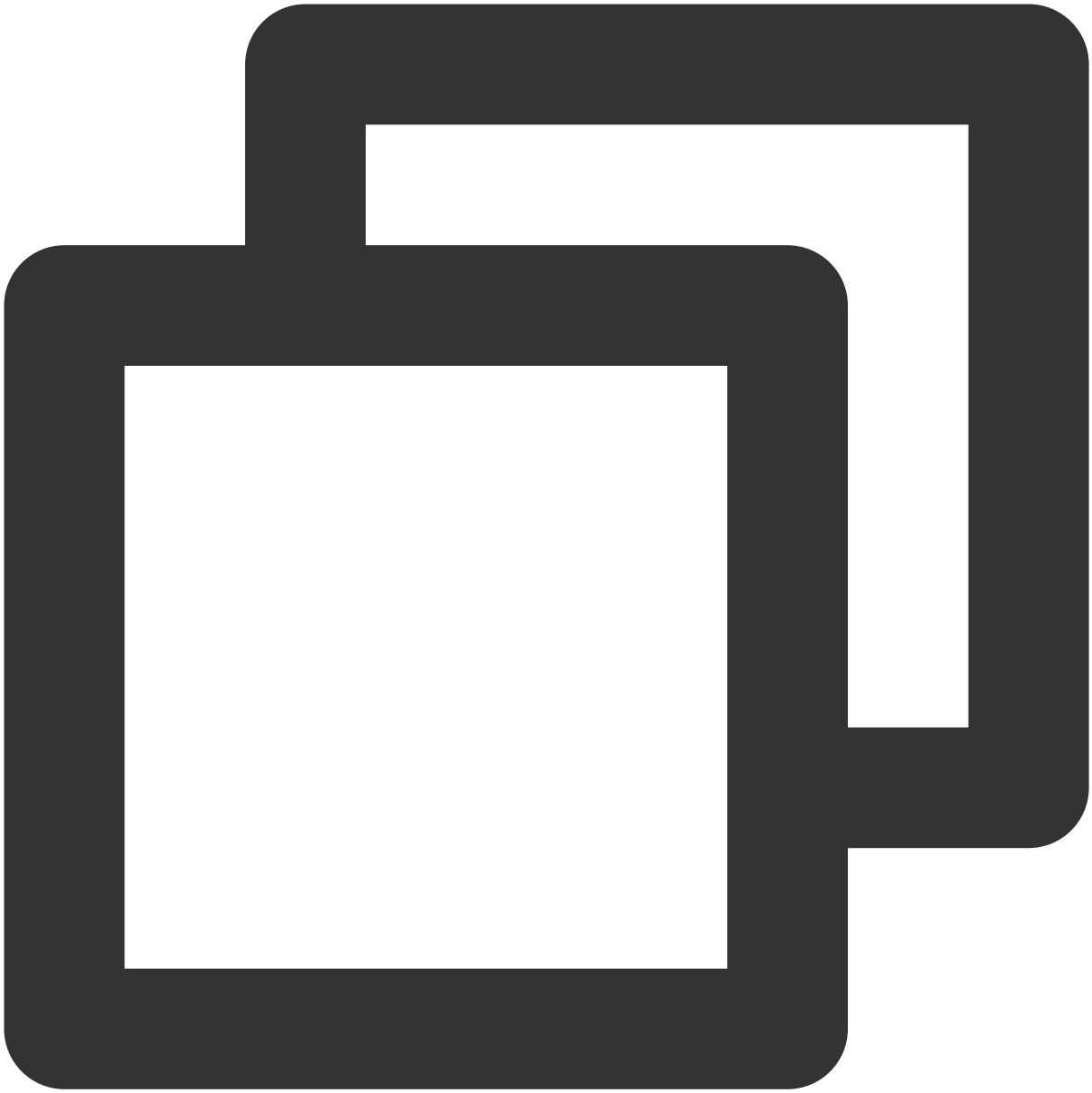
5、启动 pgpool-II 并验证读写分离

```
$ pgpool -n -d > /tmp/pgpool.log 2>&1 &
```

说明：

连接并查询 pg_is_in_recovery()，然后断开重连再查询 pg_is_in_recovery()，如果交替返回 false 和 true，说明是交替将请求发送给了主库和从库，即读写分离成功。

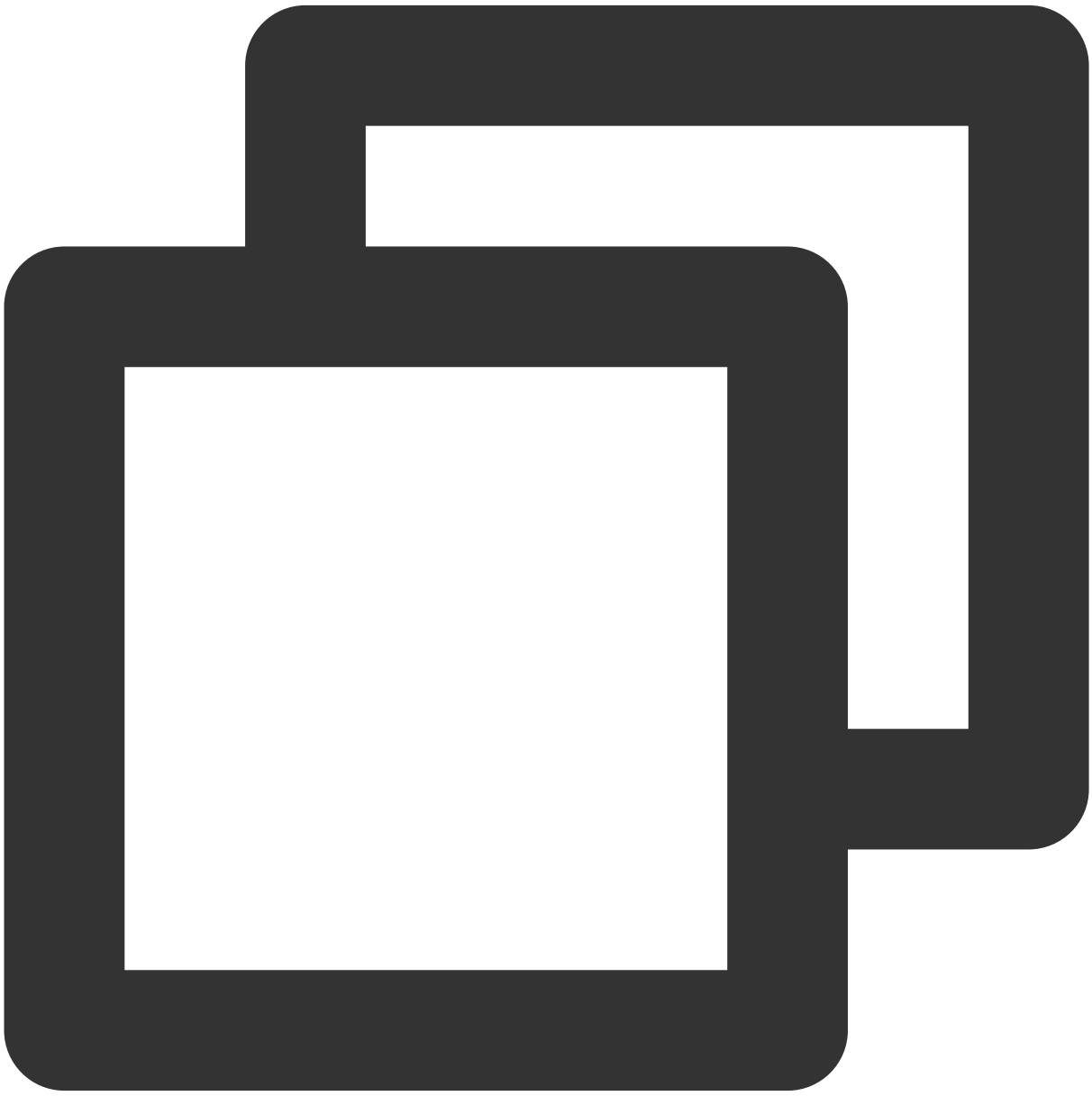
使用客户端 psql 连接 pgpool，展示 status 为正常。



```
[root@VM-0-15-tencentos ~]# /usr/local/pgsql/bin/psql -h127.0.0.1 -p9989 -Udbadmin
Password for user dbadmin:
psql (15.1)
Type "help" for help.

postgres=> show pool_nodes;
 node_id |  hostname   | port | status | pg_status | lb_weight |  role   | pg_role
atus_change
---------+-------------+------+--------+-----------+-----------+---------+---------
-------------
 0       | 172.16.0.30  | 5432 | up     | unknown   | 0.500000  | primary | unknown
```

```
-27 20:04:13
 1       | 172.16.0.16 | 5432 | up     | unknown   | 0.500000  | standby | unknown
-27 20:04:13
(2 rows)

postgres=>
```

在客户端使用读写 SQL，由于提前区分了读写实例和只读实例，发现读写分离成功。



```
postgres=> insert into pgpool1(id,name)values(3,'b');
INSERT 0 1
postgres=> select * from pgpool1;
```

```
 id | name
----+------
  1 | a
  2 | b
  3 | a
  4 | b
  3 | a
(5 rows)

postgres=>
```