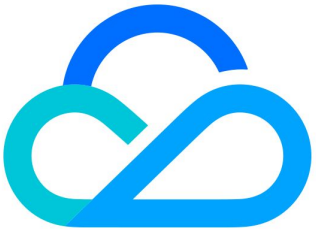
云数据库 PostgreSQL

实践教程



腾讯云

【联系我们】

我们致力于为您提供个性化的售前购买咨询服务，及相应的技术售后服务，任何问题请联系 4009100100或95716。

文档目录

实践教程
跨库访问

最近更新时间：2025-05-08 19:14:32

跨库访问是指对本实例中其他库中的数据，或其他实例中的数据进行数据读取、写入、联合操作的过程。跨库访问的目标对象统称为外部数据源。
云数据库 PostgreSQL 提供用于访问外部数据源的插件，用于实现、增强跨库访问能力。外部访问插件包括两种：

种类	插件
同构跨库访问插件	dblink、postgresql_fdw
异构跨库访问插件	mysql_fdw、cos_fdw

跨库访问操作步骤如下：
1. 使用 “CREATE EXTENSION 插件名;” 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。

另外，跨库访问插件强大的访问能力如果不加以约束，可能会带来一定的安全隐患。因此，云数据库 PostgreSQL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：**
> 大版本10及以上的云数据库 PostgreSQL 内核才支持 dblink 插件，请知悉。

插件辅助参数

- **host**
  进行跨实例访问的必填项。目标实例的 IP 地址。
- **port**
  进行跨实例访问的必填项。目标实例的端口号。
- **instanceid**
  实例 ID
  - 进行云数据库 PostgreSQL 间跨实例访问的必填项。格式类似 postgres-xxxxxx、pgro-xxxxxx，可在 控制台 的实例列表中查看。
  - 如果目标实例在腾讯云 CVM 上，则填写为 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 服务的数据库名称。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。目标实例所属类型如下：
  - 取值为1时，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。
  - 取值为2时，目标实例在腾讯云 CVM 机器上。
  - 取值为3时，目标实例为腾讯云外网自建。
  - 取值为4时，目标实例为云 VPN 接入的实例。
  - 取值为5时，目标实例为自建 VPN 接入的实例。
  - 取值为6时，目标实例为专线接入的实例。
- **uin**
  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库。

```
postgres=>create role user1 with LOGIN  CREATEDB
tencentdb_superuser  PASSWORD 'password1';
CREATE ROLE
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=# grant all on database testdb2 to user2;
GRANT
postgres=> \c testdb2 user2
You are now connected to database "testdb2" as user "user2".
Testdb2=> create schema test_schema;
CREATE SCHEMA
testdb2=> create table test_schema.test_table2(id integer);
CREATE TABLE
testdb2=> insert into test_schema.test_table2 values (1);
INSERT 0 12
```

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  | 1.1     | public     | foreign-data wrapper for remote
PostgreSQL servers
(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 'xxx.xxx.xxx.xxx',dbname 'testdb2', port '5432',
instanceid 'postgres-xxxxx');
CREATE SERVER
```

- 不跨实例，仅跨数据库访问，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 'xxx.xxx.xxx.xxx', dbname 'testdb2', port '5432',
instanceid 'ins-xxxxx', access_type '2', region 'ap-guangzhou', uin
'xxxxxx', own_uin 'xxxxxx');
  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 'xxx.xxx.xxx.xxx',dbname 'testdb2', port '5432',
instanceid 'ins-xxxxx', access_type '2', region 'ap-guangzhou', uin
'xxxxxx', own_uin 'xxxxxx', vpcid 'vpc-xxxxxx', subnetid 'subnet-
xxxxx');
  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 'xxx.xxx.xxx.xxx',dbname 'testdb2', port '5432',
access_type '3', region 'ap-guangzhou', uin 'xxxxxx', own_uin 'xxxxxx');
```

```
  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 'xxx.xxx.xxx.xxx',dbname 'testdb2', port '5432',
access_type '4', region 'ap-guangzhou', uin 'xxxxxx', own_uin 'xxxxxx',
vpngwid 'xxxxxx');
```

- 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 'xxx.xxx.xxx.xxx',dbname 'testdb2', port '5432',
access_type '5', region 'ap-guangzhou', uin 'xxxxxx', own_uin 'xxxxxx',
vpngwid 'xxxxxx');
```

- 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 'xxx.xxx.xxx.xxx',dbname 'testdb2', port '5432',
access_type '6', region 'ap-guangzhou', uin 'xxxxxx', own_uin 'xxxxxx',
dcgid 'xxxxxx');
  CREATE SERVER
```

步骤4：创建用户映射

```
testdb1=> create user mapping for user1 server srv_test1 options (user
'user2',password 'password2');
CREATE USER MAPPING
```

步骤5：创建外部表

```
testdb1=> create foreign table foreign_table1(id integer) server srv_test1
options(schema_name ‘test_schema’,table_name'test_table2');
CREATE FOREIGN TABLE
```

> **说明：**
> 同实例的跨 database 访问，只需填写目标表的名称 table_name，无需填写 schema_name 参数。

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
- 9.5 版本 SERVER 创建
- 10 版本 SERVER 创建
- 11 版本 SERVER 创建
- 12 版本 SERVER 创建
- 13 版本 SERVER 创建
- 14 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 from within a database
 pg_stat_log         | 1.0     | public     | track runtime execution
statistics of all SQL statements executed
 pg_stat_statements | 1.6     | public     | track execution statistics of
all SQL statements executed
 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-2123455r dbname=postgres access_type=1 user=dbadmin
```

```
password=P302!');
 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_owner from information_schema.schemata') as
t(a varchar(50),b varchar(50),c varchar(50));
    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_fdw 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   |
Description
---------------------+---------+------------+------------------------------
---------------------------------------
 dblink              | 1.2     | public     | connect to other PostgreSQL
databases from within a database
 mysql_fdw           | 1.1     | public     | Foreign data wrapper for
querying a MySQL server
 pg_stat_log         | 1.0     | public     | track runtime execution
statistics of all SQL statements executed
```

```
 pg_stat_statements | 1.9     | public     | track planning and execution
statistics of all SQL statements executed
 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 '171.16.10.13',port '3306',instanceid 'cdb-l1d95grp',uin
'100026380431');
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_user',password 'Secret!123');
CREATE USER MAPPING
```

步骤四：访问外部数据

> **说明：**
> 本实例所要连接的 mysql 数据库中必须至少有一张表，才能通过 IMPORT FOREIGN SCHEMA 导入表结构。

```
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：2024-04-30 16:00:12

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 ATTACH PARTITION tab_\$dateStr FOR VALUES IN ('\$dateStr')";
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
$$
DECLARE
    dateStr varchar;
BEGIN
    SELECT to_char(DATE 'tomorrow', 'YYYYMMDD') INTO dateStr;
    EXECUTE
        format('CREATE TABLE tab_%s (LIKE tab INCLUDING INDEXES)',
dateStr);
    EXECUTE
        format('ALTER TABLE tab ATTACH PARTITION tab_%s FOR VALUES IN
(%L)', dateStr, dateStr);
    RETURN 1;
END;
$$;

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_char(NEW.ts, 'YYYYMMDD'));

        /*
         * 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\')' % (table, table, date, date))


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：2024-05-11 17:57:21

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	李四	{民歌,中国风,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：
用户表：

用户ID	用户名
1	张三

2	李四
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,X,Y,
Z,a,b,c,d,e,f,g,h,i,j,k,l,m,n,o,p,q,r,s,t,u,v,w,x,y,z}';
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_string_array(5,10);
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=4.263..4.502 rows=
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=4.240..4.240 r
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 loops=1)
-> Bitmap Heap Scan on account (cost=109.08..21800.56 rows=6332 width=8)
(actual time=1.655..7.
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.
.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_int_array(100000,10);
```

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 rows=0 loops=1)
Workers Planned: 2
Workers Launched: 2
-> Parallel Seq Scan on account1 (cost=0.00..189982.68 rows=104
width=19) (actual time=962.640..962.640 rows=0 loops=3)
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 14.189 ms, Total 17.645 ms
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=0.126..0.127 rows=0 loops=1)
Recheck Cond: (tag @> '{100,5711}'::integer[])
-> Bitmap Index Scan on tag_inx_2 (cost=0.00..49.87 rows=250 width=0)
(actual time=0.124..0.124 rows=0 loops=1)
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=0.130..0.131 rows=1 loops=1)
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=0.125..0.125 rows=1 loops=1)
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[61568,97350];
QUERY PLAN

------------------------------------------------------------------------------
------------------------------------------------------------
------
Gather (cost=1961.06..173801.15 rows=99750 width=19) (actual
time=5.020..28.885 rows=2066 loops=1)
Workers Planned: 2
Workers Launched: 2
-> Parallel Bitmap Heap Scan on account1 (cost=961.06..162826.15
rows=41562 width=19) (actual time=1.623..3.305 rows=689 loo
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 time=0.685..0.685 rows=2066 loops=1)
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 3.489 ms, Total 5.836 ms
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 loops=1)
```

```
-> GroupAggregate (cost=32.42..32.46 rows=1 width=12) (actual
time=0.955..0.956 rows=1 loops=1)
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=1)
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 time=0.044..0.067 rows=4 loops=1
)
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 time=0.031..0.031 rows
=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 loops=1)
Group Key: uin_offset
-> Sort (cost=32.42..32.43 rows=4 width=22) (actual time=0.043..0.045
rows=4 loops=1)
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 time=0.029..0.036 rows=4 loops=1)
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 time=0.021..0.021 rows=4 loops=1)
```

```
  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_tables where schemaname='public' order by
pg_relation_size(relid) desc;
   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_user_indexes where schemaname='public' order by
pg_relation_size(relid) desc;
   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。

一条 SQL 实现查询附近的人

最近更新时间：2024-05-05 16:00:13

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 。

```
\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,X,Y,Z,a
```

```
,b,c,d,e,f,g,h,i,j,k,l,m,n,o,p,q,r,s,t,u,v,w,x,y,z}';
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(st_makepoint(150-random()*100, 90-
random()*100), 4326);
```

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(116.404177 39.909652)',4326),
location) from t_user order by location <-> 'SRID=4326;POINT(116.404177
39.909652)'::geometry limit 5;
```

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
```

```
39.909652)',4326), location) from t_user where
ST_DWithin(location::geography, ST_GeographyFromText('POINT(116.404177
39.909652)'), 1000.0);
```

如何配置云数据库 PostgreSQL 作为 GitLab 外部数据源

最近更新时间：2023-11-29 20:47:31

背景

Gitlab 是 种类似 github 的服务，企业可以使 它来提供 git 存储库的内部管理。它是 个 我托管的 Git-repository 管理系统，可以保持 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 户在他们的代码所在的服务器上进行定位。
- GitLab 免费提供 限数量的私人和公共存储库。
- 代码 段可以共享项 中的少量代码， 不是共享整个项 。

Gitlab 服务的元数据库在新版本（12.1）中目前仅 持 PostgreSQL，Gitlab 介绍不再支持 MySQL 的理由如下：

- 法 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 法添加 TEXT 类型字段的 度限制。
- MySQL 不 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 ，集成的数据库服务存在 定的安全风险，并且数据库服务的可靠性和可 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 稳定的外部数据库服务。而 Gitlab 在 Gitlab HA Repmgr 包中才 持基于 patroni 版本的 可 数据库。但是 维护集群是 件成本较 的事情，采 腾讯云数据库可极 的减少这些维护操作。本 介绍如何将 Gitlab 中的嵌 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 台，内存需4GB以上，磁盘需50GB以上。建议 /opt 独 挂载 个数据盘。
   - 腾讯云数据库 PostgreSQL 个，规格根据 身实际情况进 配置。可初始选择 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 需要安装 Gitlab 服务的服务器中。
3. **安装 GitLab**

   使 root 户执 下列语句安装 Gitlab，若最后 步操作提示存在依赖包未安装，可直接通过 yum 或其他安装 具提前安装完成：

```
curl
https://packages.gitlab.com/install/repositories/gitlab/gitlab-ce/script
```

```
.rpm.sh > gitlab-ee_install.sh
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　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　致，　法创建成功。

GitLab 版本	最　持的 PostgreSQL 版本
13.0	11
14.0	12

2. 通过客户端　具登录　腾讯云数据库 PostgreSQL 中。可使　psql　具测试是否可以直接访问数据库，若　法访问，请排查　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. 　先在数据库中创建　个 GitLab 所使　的账号，如 gitlab，请注意此账号必须拥有 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 <当前管理员账号>; grant pg_tencentdb_superuser to gitlab;
```

4. 然后创建　个 gitlab 所管理使　的 database：

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

> ⚠ **注意：**
> 在 GitLab 库中必须要能　持 `pg_trgm`、`btree_gist`、`plpgsql` 插件，　需提前创建，在初始化 GitLab 时候将　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 GitLab 安装服务器中，找到 gitlab 配置 件，默认为：/etc/gitlab/gitlab.rb。默认此 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 件的最后加 以下信息，将腾讯云数据库 PostgreSQL 数据源加 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
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 `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 域名的场景常伴有数据库后端是做了负载均衡或者 可 ，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 修改。

步骤4：初始化与登录使用 GitLab

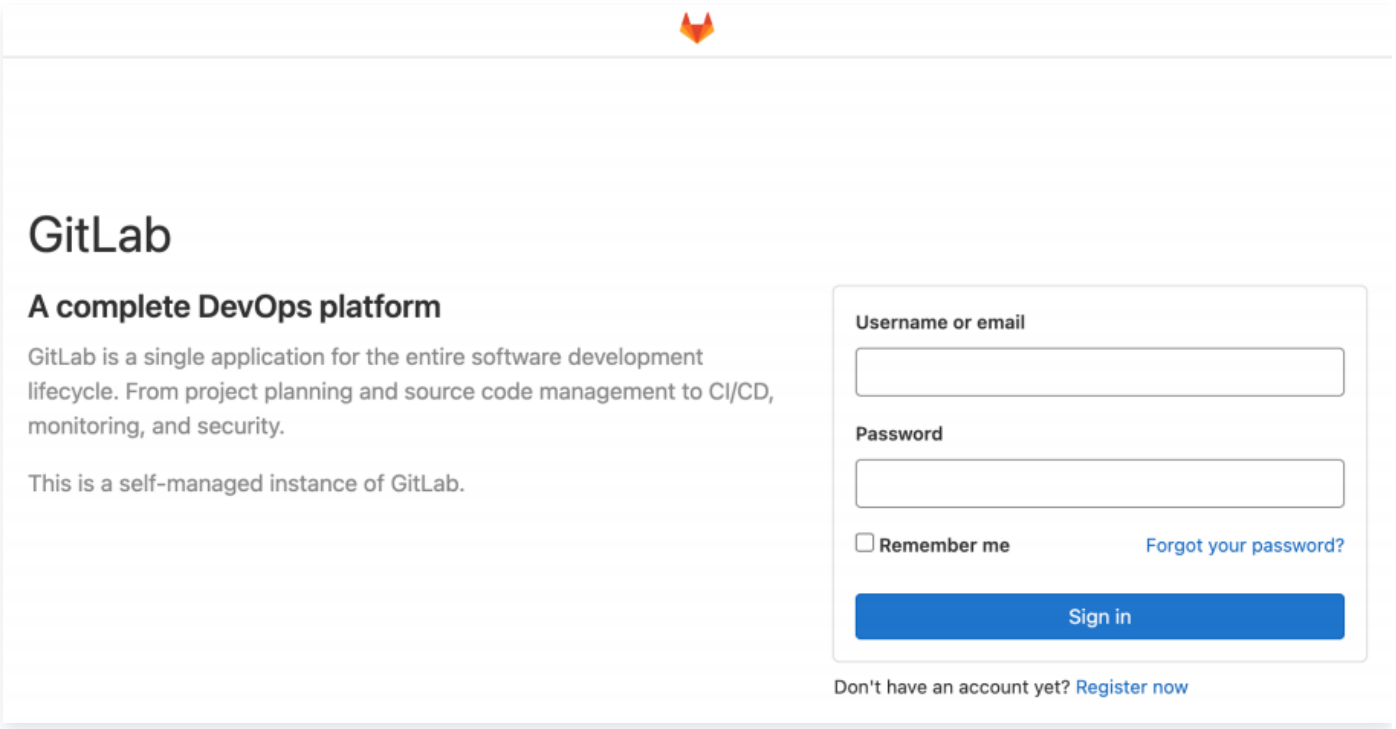
1. 执 以下命令初始化 GitLab，此命令执 会消耗 段时间，请耐心等待，当提示：`gitlab Reconfigured!` 时，说明已经初始化完成。

```
gitlab-ctl reconfigure
```

2. 执 以下命令启动 GitLab。

```
gitlab-ctl startok
```

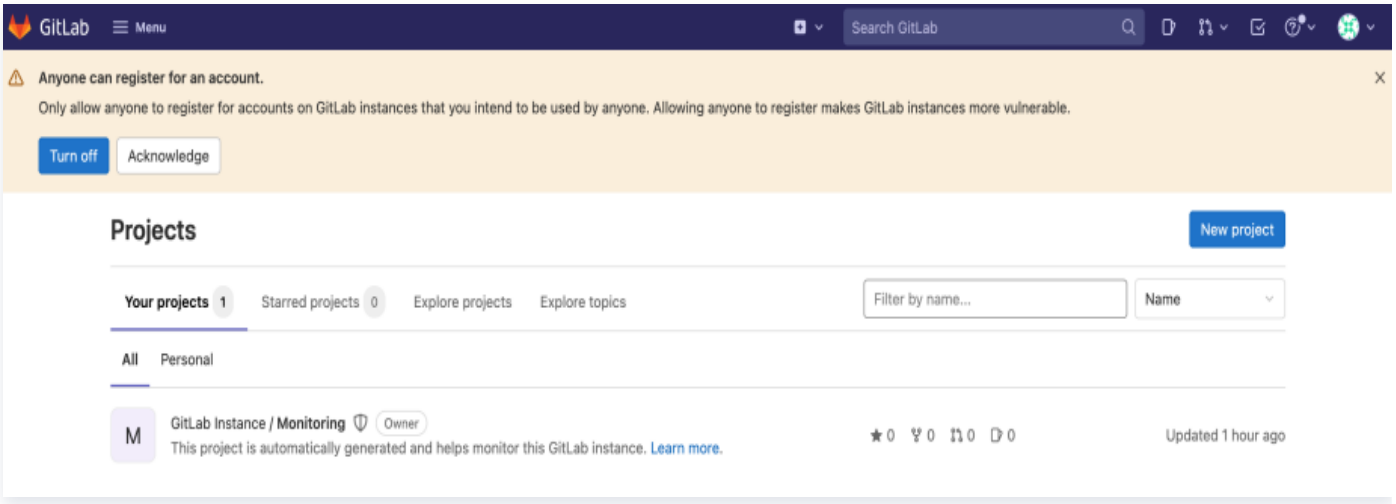
3. 可使　以下 URL 访问 GitLab，若　法访问通，可能是服务器防　墙的限制。
   地址示例：`http://{可访问的服务器 IP 地址}/users/sign_in`
   登录界面如下：



4. 初始登录账号为 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　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　GitLab。

通过 cos_fdw 插件支持分级存储能力

最近更新时间：2025-05-26 10:36:21

背景

数据库作为数据处理、存放等的核　模块，随着业务的发展，其数据量会越来越　；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　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　落冷处理。
对于数据库而言，如何最　化地存储数据以及更好的提供统　数据处理接　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　持多种成本的存储介质，供　户选择使　。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　，但性能更强的高性能 SSD 中。更好的服务　户，保证业务的正常运行，并且兼顾　户成本，是　种极具性价　的存储　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　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　样访问 COS 中的数据，实现冷热存储分离。　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　引擎**：多种存储介质，　需业务层改动代码，直接使用 PostgreSQL 数据协议均可实现统　访问。
- **成本更低**：相对　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　外表创建，即可像原表　样直接使用。
- **无限存储**：COS 存储容量不设上限，　户可以根据实际情况进行动态存储，不再担心容量问题。
- 　**持联合查询表**：多种存储的表　持联合查询，跨区 join 等，这在其他混合引擎上是　法直接实现的，均需要　个统一的数据融合节点才能　持。

支持版本

目前分级存储　持以下版本的云数据库 PostgreSQL：

- PostgreSQL 10
- PostgreSQL 11
- PostgreSQL 12
- PostgreSQL 13
- PostgreSQL 14
- PostgreSQL 15
- PostgreSQL 16

使用 cos_fdw 的方法

需要按照以下顺序使　cos_fdw：

1. 导出数据。
2. 上传　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

　先申请　个在数据库与 COS 同地域，同可　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　psql 命令访问　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. PostgreSQL 客户端　具安装完成后，进　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　容量的 CVM 进　转储上传。请参见 通过 COSFS 挂载 。
4. 针对当前环境，可执　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　此服务器中。再执　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　法安装成功 COSFS 的，可以在上　命令中加　--force 参数强制安装。

7. 安装完成 COSFS 后，执　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　个　件进　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
```

导出数据

挂载完成后，即可进　数据导出。
如果存在　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　完成后，就可以在 COS 桶对应　录中找到导出的　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　加密处理，加密算法依赖于 pgcrypto 插件，因此我们在使　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　加密存储。不同的实例将会使不同的密钥，最　限度保护用户信息。我们可以　`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　持将多个 COS　件可以映射到同　个 FOREIGN TABLE 中，在 filepath 参数中填写多个　件名，每个　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　个的文件大小打印出来，并计算出来所有文件的总大小。

```
-- 单□□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 `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□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)
-- □前不□持往外部表中写□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□
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， 前仅 持 csv
delimiter	指定数据的分隔符
quote	指定数据的引 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 于寻求 在线支持 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 api http status:403
• 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:
OGVmYzZiMmQzYjA2OWNhODk0NTRkMTBiOWVmMDAxODc0OWRkZjk0ZDM1NmI1M2E2MTRlY2MzZDh
mNmI5MWI1OTBjYzE2MjAxN2M1MzJiOTdkZjMxMDVlYTZjN2FiMmI0MWMyZGYxMDAyZmVmMjNkZD
Q5NGViMDhiZWJkOTE2YzI=
```

通过 pgpool 实现读写分离

最近更新时间：2025-05-15 17:02:02

背景

当所有请求都由主库进行处理时，可能造成主库压力过大，响应变慢，影响系统稳定性和扩展能力。如果能够将写请求发送到主库，并且将读请求发送到从库，就可以实现分摊主库压力，提高查询性能。这样的方式就称为读写分离。
pgpool 工具可以作为客户端和 PostgreSQL 集群之间的代理层，智能分发 SQL 请求。
本文介绍如何配置通过 pgpool 实现读写分离。

前提条件

两个数据库节点，分别作为主节点和只读节点。
已经完成 PostgreSQL 的安装的云服务器，用于部署 pgpool 。云服务器、两个数据库节点的 PostgreSQL 版本需一致。

步骤1: 安装 pgpool

点击下载 pgpool 下载地址 ，将 pgpool 安装包下载到本地后，再将包上传到服务器。具体上传方法请参见 Linux 系统通过 FTP 上传文件到云服务器 。
上传完成后，依次执行以下命令，完成 pgpool 的安装。 其中， pgpool 的版本号可根据需要进行修改。

```
[root@VM-10-6-tencentos ~]# tar -zxvf pgpool-II-4.4.5.tar.gz
[root@VM-10-6-tencentos ~]# cd pgpool-II-4.4.5
[root@VM-10-6-tencentos pgpool-II-4.4.5]# ./configure
[root@VM-10-6-tencentos pgpool-II-4.4.5]# make
[root@VM-10-6-tencentos pgpool-II-4.4.5]# make install
```

完成后，您可执行以下命令查询是否安装成功。若返回 pgpool 的版本信息，则安装成功。

```
[root@VM-10-6-tencentos pgpool-II-4.4.5]# pgpool --version
pgpool-II version 4.4.5 (nurikoboshi)
```

步骤2: 修改配置文件

> **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。

1. 配置 pgpool.conf 文件

安装 pgpool-II 将自动生成文件 pgpool.conf.sample ，执行以下命令，将其拷贝并重命名为 pgpool.conf ，从而进行配置文件的修改。

```
[root@VM-0-15-tencentos pgpool-II-4.4.5]# cp
/usr/local/etc/pgpool.conf.sample /usr/local/etc/pgpool.conf
[root@VM-10-6-tencentos pgpool-II-4.4.5]# vi /usr/local/etc/pgpool.conf
```

按i键进入编辑模式，进行以下项的修改。

```
#listen_addresses = '0.0.0.0'

#backend_hostname0 = '云数据库实例 IP 地址'
#backend_port0 = 5432

#enable_pool_hba = on
```

您可参考以下 pgpool.conf 文件的重要参数。

> **注意：**
> - 如下配置为重点参数示例配置，请您根据自身业务特点进行调整，并在上线前严格测试。
> - 云数据库 PostgreSQL 主实例已经具备 HA 切换能力，不需要 pgpool 进行切换，因此对于 backend_flag0 参数，主节点需要配置为ALWAYS_PRIMARY，备节点需配置为 DISALLOW_TO_FAILOVER。

```
#------------------------------------------------------------------------
-------
# 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 = 'localhost'
                                    # 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 = '主节点数据库 ip 地址'
                                    # 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 = '备节点数据库 ip 地址'
#backend_port1 = 5433
#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
#------------------------------------------------------------------------
-------
# - Concurrent session and pool size -

#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.
#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 = 0
                                    # 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.
```

2. 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的密码文件。请使用如下命令生成密码文件。其中 --username 参数使用连接云数据库的账号名称，示例为 dbadmin ，示例密码为 password ，请根据实际需要修改命令。返回的信息将自动写入 Pgpool-II 的 pool_passwd 文件中。

```
[root@VM-10-6-tencentos pgpool-II-4.4.5]# cd /usr/local/bin
[root@VM-10-6-tencentos bin]# pg_md5 --md5auth --username=dbadmin
password
[root@VM-10-6-tencentos bin]# more /usr/local/etc/pool_passwd
dbadmin:md50b0cdb5c1d1f30fe83e5a*******
```

步骤3: 配置 PCP 命令（可选）

PCP 命令是 pgpool-II 用于管理功能的接口，是完成启停后端数据库节点、查看节点状态、重载配置等操作的必要工具。若您仅将 pgpool 用于负载均衡等操作，则可跳过本步骤。
要使用 PCP 命令，必须进行用户认证。这种认证需要在 pcp.conf 文件中另外定义一个用户和密码。首先，执行以下命令，将 pcp.conf.sample 文件拷贝并重命名为 pcp.conf ，从而进行配置文件的修改。

```
[root@VM-10-6-tencentos bin]# cd /usr/local/etc
[root@VM-10-6-tencentos etc]# cp /usr/local/etc/pcp.conf.sample
/usr/local/etc/pcp.conf
```

执行以下命令，获取 pcp 用户和密码。

```
[root@VM-10-6-tencentos etc]# pg_md5 --md5auth --username=pcpuser password2
[root@VM-10-6-tencentos etc]# more /usr/local/etc/pool_passwd
dbadmin:md50b0cdb5c1d1f30fe83e5a*******
pcpuser:md5907cf835939adc1c736e2*******
```

复制返回的 pcpuser:******** ，执行以下命令，编辑 pcp.conf 文件。

```
[root@VM-10-6-tencentos etc]# vi /usr/local/etc/pcp.conf
```

按 i 进入编辑模式，将复制的内容粘贴到文件末尾，再按 esc 键退出编辑模式，直接输入 :wq 保存修改并退出文件。

步骤4: 配置数据库节点

Pgpool-II 通过配置多个后端节点，才能实现负载均衡、故障转移、高可用等功能。如果要实现读写分离，则使用两个后端数据库节点，分别为主节点和只读节点，两个节点要求如下：

节点	作用
主节点	负责写操作和读操作（如果开启读写分离）
备节点	只处理读操作

执行以下命令，编辑 pgpool.conf 配置文件。

```
[root@VM-10-6-tencentos etc]# vi /usr/local/etc/pgpool.confg
```

按 i 进入编辑模式，找到 Backend Connection Settings ，进行以下修改：

```
# - Backend Connection Settings -

#backend_hostname0 = '主节点 ip 地址'
                                    # 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 = '备节点 ip 地址'
#backend_port1 = 5433
#backend_weight1 = 1
#backend_data_directory1 = '/data1'
#backend_flag1 = 'DISALLOW_TO_FAILOVER'
#backend_application_name1 = 'server1'
```

再找到 LOAD BALANCING MODE ，当 load_balance_mode 被设置为 true，客户进行的 SELECT 查询将分发到所设置的多个数据库节点执行，实现读写分离和负载均衡。

再按 esc 键退出编辑模式，直接输入 :wq 保存修改并退出文件。

步骤5: 启动 pgpool-II 并验证读写分离

由于本文使用源码编译的方式安装 pgpool-II ，不会自动生成 systemd 服务文件，需要手动创建。执行以下命令，创建 pgpool 用户，再创建并进入 pgpool.service 服务文件：

```
[root@VM-10-6-tencentos etc]# sudo useradd -r -s /sbin/nologin pgpool
[root@VM-10-6-tencentos etc]# vi /etc/systemd/system/pgpool.service
```

按 i 进入编辑模式，将以下内容粘贴进文件：

```
[Unit]
Description=Pgpool-II connection pool server
After=network.target

[Service]
Type=simple
User=pgpool
Group=pgpool
ExecStart=/usr/local/bin/pgpool -n -f /usr/local/etc/pgpool.conf #此处需填写
pgpool.conf文件路径，可根据实际情况调整
Restart=on-failure

[Install]
WantedBy=multi-user.target
```

再依次执行以下命令，加载 systemd 配置并启动服务：

```
[root@VM-10-6-tencentos etc]# sudo mkdir -p /var/run/pgpool
[root@VM-10-6-tencentos etc]# sudo systemctl daemon-reload
[root@VM-10-6-tencentos etc]# sudo systemctl enable pgpool
[root@VM-10-6-tencentos etc]# sudo systemctl start pgpool
```

返回信息出现 process started 代表服务启动成功。

> **说明：**
> 在云服务器连接主节点并执行语句" SELECT 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。

```
[root@VM-0-15-tencentos ~]# /usr/local/pgsql/bin/psql -h127.0.0.1 -p9989 -U
dbadmin -d postgres
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 | select_cnt | load_balance_node | replication_delay |
replication_state | replication_sync_state | last_st
atus_change
---------+-------------+------+--------+-----------+-----------+---------+-
--------+------------+-------------------+-------------------+-------------
------+------------------------+--------
-------------
 0       | *.*.*.*   | 5432 | up     | unknown   | 0.500000  | primary |
unknown | 4          | false             | 0                 |
|                       | 2024-02
-27 20:04:13
 1       | *.*.*.* | 5432 | up     | unknown   | 0.500000  | standby |
unknown | 13         | true              | 0                 |
|                       | 2024-02
-27 20:04:13
(2 rows)

postgres=>
```

读写分离设置成功。

通过 auto_explain 插件实现慢 SQL 分析

最近更新时间：2025-05-15 17:02:02

> **注意：**
> - auto_explain 的开启需要重启数据库，请您提前规划运维时间窗。
> - auto_explain 开启后会有一定的性能损耗，与具体的业务有关，请先充分测试。
> - auto_explain 开启后可能会因为产生过多的日志而导致磁盘空间的上升，请知悉。
> - 如您需要开启 auto_explain 并下载日志，请 提交工单 联系我们。

重点参数说明

auto_explain 插件提供一种自动记录 SQL 执行计划的功能。当您在实例中开启该插件之后，可以通过在 控制台 的参数设置中，进行详细能力的配置。下面将针对重点的几个参数进行说明，详细说明请参考 官方文档 。

- auto_explain.log_min_duration

  该参数主要用于决定执行耗时超过多长时间的 SQL 语句会被记录执行计划。默认为-1，代表不记录。自定义范围500-60000 ，单位为毫秒。
- auto_explain.log_analyze

  加此参数可以打印执行计划的实际执行时间。默认 off ，关闭状态。
- auto_explain.log_timing

  加此参数可以打印语句执行时间。默认 off ，关闭状态。
- auto_explain.log_verbose

  加此参数可以增加 explain 中 verbose 信息输出，获得更详细的执行计划信息。默认 off ，关闭状态。

示例说明

假如实例中有 database a_all，该 database 的 public 模式下有10张表格，分别为：

```
student_info_b0、student_info_b1、student_info_b2、student_info_b3、student_info_b4、
student_info_b5、student_info_b6、student_info_b7、student_info_b8、student_info_b9
```

当前云数据库 PostgreSQL 实例已经开启了 auto_explain 插件。其参数值如下：

```
a_all=> show auto_explain.log_min_duration;
 auto_explain.log_min_duration
-------------------------------
 800ms
(1 row)

a_all=> show auto_explain.log_analyze;
 auto_explain.log_analyze
--------------------------
 on
```

```
(1 row)

a_all=> show auto_explain.log_verbose;
 auto_explain.log_verbose
--------------------------
 on
(1 row)

a_all=> show auto_explain.log_timing;
 auto_explain.log_timing
-------------------------
 on
(1 row)
```

执行如下语句：

```
SELECT user_id, COUNT(*) OVER (PARTITION BY user_id) as countFROM (
SELECT user_id FROM student_info_b0    UNION ALL    SELECT user_id FROM
student_info_b1    UNION ALL    SELECT user_id FROM student_info_b2
UNION ALL    SELECT user_id FROM student_info_b3    UNION ALL    SELECT
user_id FROM student_info_b4    UNION ALL    SELECT user_id FROM
student_info_b5    UNION ALL    SELECT user_id FROM student_info_b6
UNION ALL    SELECT user_id FROM student_info_b7    UNION ALL    SELECT
user_id FROM student_info_b8    UNION ALL    SELECT user_id FROM
student_info_b9) AS all_students;
```

在 云数据库 PostgreSQL 的控制台 看到的慢日志记录如图所示：

下载的 auto_explain 日志中执行计划如下：

```
duration: 147.603 ms  plan:
Query Text: SELECT user_id, COUNT(*) OVER (PARTITION BY user_id) as count
FROM (
    SELECT user_id FROM student_info_b0
    UNION ALL
    SELECT user_id FROM student_info_b1
    UNION ALL
    SELECT user_id FROM student_info_b2
    UNION ALL
    SELECT user_id FROM student_info_b3
    UNION ALL
    SELECT user_id FROM student_info_b4
    UNION ALL
    SELECT user_id FROM student_info_b5
    UNION ALL
    SELECT user_id FROM student_info_b6
    UNION ALL
    SELECT user_id FROM student_info_b7
    UNION ALL
    SELECT user_id FROM student_info_b8
    UNION ALL
    SELECT user_id FROM student_info_b9
) AS all_students;
```

```
WindowAgg  (cost=19181.71..21924.66 rows=156740 width=14) (actual
time=56.009..116.522 rows=157000 loops=1)
  Output: student_info_b0.user_id, count(*) OVER (?)
  ->  Sort  (cost=19181.71..19573.56 rows=156740 width=6) (actual
time=55.956..72.756 rows=157000 loops=1)
        Output: student_info_b0.user_id
        Sort Key: student_info_b0.user_id
        Sort Method: external merge  Disk: 2448kB
        ->  Append  (cost=0.00..3511.10 rows=156740 width=6) (actual
time=0.010..20.861 rows=157000 loops=1)
              ->  Seq Scan on public.student_info_b0  (cost=0.00..272.74
rows=15674 width=4) (actual time=0.009..1.367 rows=15700 loops=1)
                    Output: student_info_b0.user_id
              ->  Seq Scan on public.student_info_b1  (cost=0.00..272.74
rows=15674 width=6) (actual time=0.005..1.302 rows=15700 loops=1)
                    Output: student_info_b1.user_id
              ->  Seq Scan on public.student_info_b2  (cost=0.00..272.74
rows=15674 width=6) (actual time=0.004..1.316 rows=15700 loops=1)
                    Output: student_info_b2.user_id
              ->  Seq Scan on public.student_info_b3  (cost=0.00..272.74
rows=15674 width=6) (actual time=0.005..1.318 rows=15700 loops=1)
                    Output: student_info_b3.user_id
              ->  Seq Scan on public.student_info_b4  (cost=0.00..272.74
rows=15674 width=6) (actual time=0.006..1.320 rows=15700 loops=1)
                    Output: student_info_b4.user_id
              ->  Seq Scan on public.student_info_b5  (cost=0.00..272.74
rows=15674 width=6) (actual time=0.005..1.294 rows=15700 loops=1)
                    Output: student_info_b5.user_id
              ->  Seq Scan on public.student_info_b6  (cost=0.00..272.74
rows=15674 width=6) (actual time=0.004..1.377 rows=15700 loops=1)
                    Output: student_info_b6.user_id
              ->  Seq Scan on public.student_info_b7  (cost=0.00..272.74
rows=15674 width=6) (actual time=0.005..1.327 rows=15700 loops=1)
                    Output: student_info_b7.user_id
              ->  Seq Scan on public.student_info_b8  (cost=0.00..272.74
rows=15674 width=6) (actual time=0.006..1.285 rows=15700 loops=1)
                    Output: student_info_b8.user_id
              ->  Seq Scan on public.student_info_b9  (cost=0.00..272.74
rows=15674 width=6) (actual time=0.004..1.293 rows=15700 loops=1)
                    Output: student_info_b9.user_id
```

如此，我们能清晰的查看该慢 SQL 的详细执行计划，并进行后续的业务分析。

使用 pglogical 进行逻辑复制

最近更新时间：2025-05-08 19:14:32

pglogical 是 PostgreSQL 的逻辑复制扩展插件，使用“发布/订阅”的数据模型进行数据复制。该模型包含“发布端”和“订阅端”，发布端用于定义和发布指定的数据，订阅端可以选择订阅，从而接收并使用数据变更。pglogical 允许用户只复制特定的数据内容，减少了不必要的数据传输和处理，使选择性复制更加高效。
pglogical 的复制可应用于多种场景，包括：

- 数据库大版本升级。
- 完整的数据库复制。
- 利用复制集，选择性地筛选表、行、列。
- 可从多个上游服务器，用于数据的聚集和合并。

前提条件

> **说明：**
> - 如您需要使用 pglogical 插件，请您 提交工单 联系我们添加 shared_preload_libraries 参数。修改 shared_preload_libraries 参数会重启实例，请确保业务有重连机制。
> - 云数据库 PostgreSQL 支持同一地域下的同一实例或者不同实例间配置 pglogical 逻辑复制。
> - 当前只有内核版本在 v11.22_r1.21、v12.20_r1.24、v13.16_r1.19、v14.13_r1.26、v15.6_r1.13、v16.4_r1.7 及以上且大版本为11~16的云数据库 PostgreSQL 实例才支持 pglogical 插件。

请您确认 shared_preload_libraries 参数中包含 pglogical，如下图所示：

```
postgres=> show shared_preload_libraries;

shared_preload_libraries

--------------------------------------------------------------------------------
-------------------------------------------
-------------

pg_stat_statements,pg_stat_log,wal2json,decoderbufs,decoder_raw,pg_hint_pla
n,rds_server_handler,tencentdb_pwdcheck,pgaud
it,pglogical
(1 row)
```

参数 wal_level 的值为 logical，如下图所示。

```
postgres=> show wal_level;
```

```
 wal_level
-----------
 logical
(1 row)
```

如您需要修改 wal_level 参数，请您进入 云数据库 PostgreSQL 控制台 的参数设置页面，修改参数：

操作步骤

在发布端和订阅端实例确认可安装插件中是否有 pglogical：

```
postgres=> select * from pg_available_extensions where name='pglogical';
   name     | default_version | installed_version |             comment
------------+-----------------+-------------------+--------------------------
-------
 pglogical | 2.4.4           |                   | PostgreSQL Logical
Replication
(1 row)
```

安装插件

新建 database，名称为 am，并切换至 am：

```
postgres=> create database am;
CREATE DATABASE
postgres=> \c am
You are now connected to database "am" as user "dbadmin".
am=>
```

创建表格 t，并插入数据：

```
am=> create table t(a int primary key, b int);
CREATE TABLE
```

```
am=> insert into t(a,b)values(1,1),(2,2),(3,3);
INSERT 0 3
am=> select * from t;
 a | b
---+---
 1 | 1
 2 | 2
 3 | 3
(3 rows)
```

在发布端和订阅端实例都要执行以下命令，创建 pglogical 插件：

```
am=> create extension pglogical;
CREATE EXTENSION
```

可使用以下命令查询是否安装成功：

```
am=>  select * from pg_extension where extname='pglogical';
  oid  |  extname  | extowner | extnamespace | extrelocatable | extversion
| extconfig | extcondition
-------+-----------+----------+--------------+----------------+------------
+-----------+--------------
 16424 | pglogical |    16387 |        16423 | f              | 2.4.4
|           |
(1 row)
```

在发布端创建发布节点

创建发布节点, dns 中参数请参考 跨库访问 。这里的参数均填写发布端所在实例信息。

```
am=> SELECT pglogical.create_node(node_name := 'provider1',dsn :=
'host=10.*.*.* port=5432 dbname=am user=**** password=****
instanceid=postgres-****** uin=********');
 create_node
-------------
  2976894835
(1 row)
```

配置复制集

如下示例是配置复制集中的一种，使用 pglogical_republication_set_add_all_tables 函数，将指定 schema 下的所有表添加到指定的复制集中。’default’指复制集的名称，可自定义，此处使用默认的复制集名

称；’public’指要发布的表所在的 schema 名称，此处为 public。
更多参数请参考 pglogical 官方文档 。

```
am=> SELECT pglogical.replication_set_add_all_tables('default',
ARRAY['public']);
 replication_set_add_all_tables
--------------------------------
 t
(1 row)
```

在订阅端创建订阅节点

在订阅端对应的 database 中创建表格：

```
postgres=> \c am
psql (14.11, server 15.6)
WARNING: psql major version 14, server major version 15.
         Some psql features might not work.
You are now connected to database "am" as user "dbadmin".
am=> create table t(a int primary key, b int);
CREATE TABLE
```

在订阅端创建节点，dns 中参数请参考 跨库访问 。这里的参数均填写**订阅端**所在实例信息。

```
am=> SELECT pglogical.create_node(node_name := 'subscriber1',dsn :=
'host=10.*.*.* port=5432 dbname=am user=**** password=****
instanceid=postgres-***** uin=*******');
 create_node
-------------
   330520249
(1 row)
```

在订阅端创建订阅

在订阅端创建节点，dns 中参数请参考 跨库访问 。这里的参数均填写**发布端**所在实例信息。

```
am=> SELECT pglogical.create_subscription(subscription_name :=
'subscription1',provider_dsn := 'host=10.*.*.* port=5432 dbname=am
user=**** password=**** instanceid=postgres-**** uin=*******');
 create_subscription
---------------------
          1763399739
```

```
(1 row)
```

在订阅端查询表，校验目标端数据。若查询结果与发布端数据一致，则逻辑复制成功：

```
am=> select * from t;
 a | b
---+---
 1 | 1
 2 | 2
 3 | 3
(3 rows)
```

使用 tencentdb_ai 插件构建 AI 应用

最近更新时间：2025-02-19 18:48:12

本文需要添加 `tencentdb_ai` 插件，该插件的使用请参考 使用 tencentdb_ai 插件调用大模型 。

集成 DeepSeek-V3 实现聊天

1. 创建 tencentdb_ai 插件。

```
damoxing=> CREATE EXTENSION tencentdb_ai CASCADE;
NOTICE:  installing required extension "pgcrypto"
CREATE EXTENSION
damoxing=> SELECT * FROM pg_extension;
  oid  |   extname    | extowner | extnamespace | extrelocatable |
extversion | extconfig | extcondition
-------+--------------+----------+--------------+----------------+------
------+-----------+--------------
 14275 | plpgsql      |       10 |           11 | f              | 1.0
|           |
 16631 | pgcrypto     |    16615 |         2200 | t              | 1.3
|           |
 16668 | tencentdb_ai |    16615 |         2200 | t              | 1.0
| {16670}   | {""}
(3 rows)
```

2. 添加 `DeepSeek-V3` 模型。

```
damoxing=> SELECT tencentdb_ai.add_model('deepseek-v3', '2024-05-22',
'ap-guangzhou', '$.Response.Choices[*].Message.Content');
 add_model
-----------

(1 row)

damoxing=> SELECT tencentdb_ai.update_model_attr('deepseek-v3',
'SecretId', 'AKID**************');
 update_model_attr
-------------------

(1 row)

damoxing=> SELECT tencentdb_ai.update_model_attr('deepseek-v3',
'SecretKey', '********************');
```

```
 update_model_attr
-------------------

(1 row)
```

3. 准备云数据库 `PostgreSQL` 对象。

```
damoxing=> CREATE TABLE chat_logs (
    id SERIAL PRIMARY KEY,
    user_input TEXT,
    system_response TEXT,
    timestamp TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
CREATE TABLE
damoxing=>
```

4. 调用大模型聊天。

创建存储过程调用 `deepseek-v3` 的 API 接口聊天：

```
damoxing=> CREATE OR REPLACE FUNCTION chat_with_deepseek(user_input
TEXT)
RETURNS TEXT AS $$
DECLARE
    system_response TEXT;
BEGIN
        SELECT tencentdb_ai.chat_completions('deepseek-v3', user_input)
INTO system_response;
    INSERT INTO chat_logs (user_input, system_response)
    VALUES (user_input, system_response);

    RETURN system_response;
END;
$$ LANGUAGE plpgsql;
CREATE FUNCTION
damoxing=>
```

调用存储过程聊天：

```
damoxing=> SELECT chat_with_deepseek('hi, deepseek');

chat_with_deepseek
```

```
------------------------------------------------------------------------
------------------------------------------------------------
------------------------------------------------------------------------
---------
 "Hello! It seems like you're trying to reach DeepSeek, but I’m not sure
what you’re looking for. Could you clarify? Are you referring t
o a specific tool, service, or something else? Let me know how I can
assist!"
(1 row)

damoxing=>
```

5. 查询聊天结果。

```
damoxing=> SELECT * FROM chat_logs ORDER BY timestamp DESC;
 id |  user_input  |
system_response

|         timestamp
----+--------------+-----------------------------------------------------
------------------------------------------------------------
------------------------------------------------------------------------
----------------------------+----------------------------
  3 | hi, deepseek | "Hello! It seems like you're trying to reach
DeepSeek, but I’m not sure what you’re looking for. Could you clarify?
 Are you referring to a specific tool, service, or something else? Let
me know how I can assist!" | 2025-02-18 16:02:39.145322
(1 row)

damoxing=>
```

集成 lke-text-embedding-v1 实现文本检索

1. 创建 `tencentdb_ai` 插件。

```
damoxing_lke_embdding=> CREATE EXTENSION tencentdb_ai CASCADE;
NOTICE:  installing required extension "pgcrypto"
CREATE EXTENSION
damoxing_lke_embdding=> SELECT * FROM pg_extension;
  oid  |   extname    | extowner | extnamespace | extrelocatable |
extversion | extconfig | extcondition
```

```
-------+--------------+----------+--------------+----------------+------
------+-----------+--------------
 14275 | plpgsql      |       10 |           11 | f              | 1.0
|           |
 16631 | pgcrypto     |    16615 |         2200 | t              | 1.3
|           |
 16668 | tencentdb_ai |    16615 |         2200 | t              | 1.0
| {16670}   | {""}
(3 rows)
```

2. 添加 `lke-text-embedding-v1` 模型。

```
damoxing_lke_embdding=> SELECT tencentdb_ai.add_model('lke-text-
embedding-v1', '2024-05-22', 'ap-guangzhou', NULL);
-[ RECORD 1 ]
add_model |

damoxing_lke_embdding=> SELECT tencentdb_ai.update_model_attr('lke-text-
embedding-v1', 'SecretId', 'AKID**************');
-[ RECORD 1 ]-----+-
update_model_attr |

damoxing_lke_embdding=> SELECT tencentdb_ai.update_model_attr('lke-text-
embedding-v1', 'SecretKey', '********************');
-[ RECORD 1 ]-----+-
update_model_attr |

damoxing_lke_embdding=>
```

3. 准备云数据库 PostgreSQL 对象。

```
damoxing_lke_embdding=> CREATE TABLE documents (
    id SERIAL PRIMARY KEY,
    content TEXT,
    embedding FLOAT8[]
);
CREATE TABLE
damoxing_lke_embdding=>
```

4. 创建生成嵌入向量的存储过程。

```
damoxing_lke_embdding=> CREATE OR REPLACE FUNCTION
generate_embedding(doc_contents TEXT[])
```

```
RETURNS FLOAT8[][] AS $$
DECLARE
    embeddings FLOAT8[][] := '{}';
    embedding FLOAT8[];
    doc_content TEXT;
BEGIN
        SELECT * from tencentdb_ai.get_embedding('lke-text-embedding-
v1', doc_contents) INTO embedding;
    RETURN embedding;
END;
$$ LANGUAGE plpgsql;
CREATE FUNCTION
damoxing_lke_embdding=>
```

5. 创建插入文档及其嵌入向量的存储过程。

```
damoxing_lke_embdding=> CREATE OR REPLACE FUNCTION
insert_documents(doc_contents TEXT[])
RETURNS VOID AS $$
DECLARE
    embedding FLOAT8[];
    doc_content TEXT;
BEGIN
        doc_content := doc_contents[0];
        embedding :=  generate_embedding(doc_contents);
        INSERT INTO documents (content, embedding)
        VALUES (doc_contents, embedding);
END;
$$ LANGUAGE plpgsql;
CREATE FUNCTION
damoxing_lke_embdding=>
```

6. 创建余弦相似度计算的存储过程。

```
damoxing_lke_embdding=> CREATE OR REPLACE FUNCTION
cosine_similarity(vec1 FLOAT8[], vec2 FLOAT8[])
RETURNS FLOAT8 AS $$
DECLARE
    dot_product FLOAT8 := 0;
    norm1 FLOAT8 := 0;
    norm2 FLOAT8 := 0;
    similarity FLOAT8;
BEGIN
```

```
    FOR i IN 1..array_length(vec1, 1) LOOP
        dot_product := dot_product + vec1[i] * vec2[i];
        norm1 := norm1 + vec1[i] * vec1[i];
        norm2 := norm2 + vec2[i] * vec2[i];
    END LOOP;

    similarity := dot_product / (sqrt(norm1) * sqrt(norm2));
    RETURN similarity;
END;
$$ LANGUAGE plpgsql;
damoxing_lke_embdding=>
```

7. 创建文本检索的存储过程。

```
damoxing_lke_embdding=> CREATE OR REPLACE FUNCTION
search_similar_documents(query TEXT, limit_count INT)
RETURNS TABLE(id INT, content TEXT, similarity FLOAT8) AS $$
DECLARE
    query_embedding FLOAT8[];
BEGIN
    query_embedding := generate_embedding(ARRAY[query]);

    RETURN QUERY
    SELECT d.id, d.content, cosine_similarity(d.embedding,
query_embedding) AS similarity
    FROM documents d
    ORDER BY similarity DESC
    LIMIT limit_count;
END;
$$ LANGUAGE plpgsql;
CREATE FUNCTION
damoxing_lke_embdding=>
```

8. 插入文档及其嵌入向量。

```
damoxing_lke_embdding=> SELECT insert_documents(ARRAY['This is a sample
document.']);
-[ RECORD 1 ]----+-
insert_documents |

damoxing_lke_embdding=> SELECT insert_documents(ARRAY['Another example
of a document.']);
-[ RECORD 1 ]----+-
```

```
insert_documents |

damoxing_lke_embdding=> SELECT insert_documents(ARRAY['This document is
about PostgreSQL and text embeddings.']);
-[ RECORD 1 ]----+-
insert_documents |

damoxing_lke_embdding=> SELECT insert_documents(ARRAY['Deep learning
models can generate text embeddings.']);
-[ RECORD 1 ]----+-
insert_documents |

damoxing_lke_embdding=>
```

9. 进行文本检索。

```
damoxing_lke_embdding=> SELECT * FROM
search_similar_documents('PostgreSQL text embeddings', 5);
-[ RECORD 1 ]----------------------------------------------------------
id         | 3
content    | {"This document is about PostgreSQL and text embeddings."}
similarity | 0.8649945496222797
-[ RECORD 2 ]----------------------------------------------------------
id         | 4
content    | {"Deep learning models can generate text embeddings."}
similarity | 0.6824638514797066
-[ RECORD 3 ]----------------------------------------------------------
id         | 1
content    | {"This is a sample document."}
similarity | 0.66412244051794
-[ RECORD 4 ]----------------------------------------------------------
id         | 2
content    | {"Another example of a document."}
similarity | 0.6142928906219256

damoxing_lke_embdding=>
```

集成 lke-reranker-base 实现 RAG

1. 创建 tencentdb_ai 插件。

```
damoxing_rerank=> CREATE EXTENSION tencentdb_ai CASCADE;
NOTICE:  installing required extension "pgcrypto"
```

```
CREATE EXTENSION
damoxing=> SELECT * FROM pg_extension;
  oid  |   extname    | extowner | extnamespace | extrelocatable |
extversion | extconfig | extcondition
-------+--------------+----------+--------------+----------------+------
------+-----------+--------------
 14275 | plpgsql      |       10 |           11 | f              | 1.0
|           |
 16631 | pgcrypto     |    16615 |         2200 | t              | 1.3
|           |
 16668 | tencentdb_ai |    16615 |         2200 | t              | 1.0
| {16670}   | {""}
(3 rows)
```

2. 添加 lke-reranker-base 和 lke-text-embedding-v1 模型。

```
damoxing_rerank=> SELECT tencentdb_ai.add_model('lke-reranker-base',
'2024-05-22', 'ap-guangzhou', '$.Response.ScoreList');
 add_model
-----------

(1 row)

damoxing_rerank=> SELECT tencentdb_ai.update_model_attr('lke-reranker-
base', 'SecretId', 'AKID**************');
 update_model_attr
-------------------

(1 row)

damoxing_rerank=> SELECT tencentdb_ai.update_model_attr('lke-reranker-
base', 'SecretKey', '********************');
 update_model_attr
-------------------

(1 row)

damoxing_rerank=> SELECT tencentdb_ai.add_model('lke-text-embedding-v1',
'2024-05-22', 'ap-guangzhou', NULL);
 add_model
-----------

(1 row)
```

```
damoxing_rerank=> SELECT tencentdb_ai.update_model_attr('lke-text-
embedding-v1', 'SecretId', 'AKID**************');
 update_model_attr
-------------------

(1 row)

damoxing_rerank=> SELECT tencentdb_ai.update_model_attr('lke-text-
embedding-v1', 'SecretKey', '********************');
 update_model_attr
-------------------

(1 row)
```

3. 准备云数据库 PostgreSQL 对象。

```
damoxing_rerank=> CREATE TABLE documents (
    id SERIAL PRIMARY KEY,
    content TEXT,
    embedding FLOAT8[]
);
CREATE TABLE
damoxing_rerank=>
```

4. 创建生成嵌入向量的存储过程。

```
damoxing_rerank=> CREATE OR REPLACE FUNCTION
generate_embedding(doc_contents TEXT[])
RETURNS FLOAT8[][] AS $$
DECLARE
    embeddings FLOAT8[][] := '{}';
    embedding FLOAT8[];
    doc_content TEXT;
BEGIN
        SELECT * from tencentdb_ai.get_embedding('lke-text-embedding-
v1', doc_contents) INTO embedding;
    RETURN embedding;
END;
$$ LANGUAGE plpgsql;
CREATE FUNCTION
```

5. 创建插入文档及其嵌入向量的存储过程。

```
damoxing_rerank=> CREATE OR REPLACE FUNCTION
insert_documents(doc_contents TEXT[])
RETURNS VOID AS $$
DECLARE
    embedding FLOAT8[];
    doc_content TEXT;
BEGIN
        doc_content := doc_contents[0];
        embedding :=  generate_embedding(doc_contents);
        INSERT INTO documents (content, embedding)
        VALUES (doc_contents, embedding);
END;
$$ LANGUAGE plpgsql;
CREATE FUNCTION
```

6. 创建召回和大模型重排的存储过程。

```
damoxing_rerank=> CREATE OR REPLACE FUNCTION retrieve_and_rerank(query
TEXT)
RETURNS TABLE (id INT, content TEXT, score FLOAT8) AS $$
DECLARE
    keyword_results RECORD;
    vector_results RECORD;
    query_vector VECTOR;
    rerank_content TEXT[];
    rerank_ids INT[];
    rerank_scores my_record;
    i INT;
BEGIN
    -- 生成查询向量
    SELECT generate_embedding(ARRAY[query])::vector INTO query_vector;

    -- 初始化重排序内容数组
    rerank_content := ARRAY[]::TEXT[];
    rerank_ids := ARRAY[]::INT[];

    -- 关键词召回
    FOR keyword_results IN
        SELECT d.id, d.content, d.embedding
        FROM documents d
        WHERE to_tsvector('english', d.content) @@
plainto_tsquery('english', query)
    LOOP
```

```
        -- 向量召回
        FOR vector_results IN
            SELECT d.id, d.content, d.embedding
            FROM documents d
            WHERE d.id = keyword_results.id
            ORDER BY d.embedding <-> query_vector
            LIMIT 10
        LOOP
            rerank_content := array_append(rerank_content,
vector_results.content);
            rerank_ids := array_append(rerank_ids, vector_results.id);
        END LOOP;
    END LOOP;

    -- 大模型重排序

    SELECT *  FROM tencentdb_ai.run_rerank('lke-reranker-base', query,
rerank_content) INTO rerank_scores;
    RETURN QUERY
    select text_field, numeric_field from rerank_scores;
END;
$$ LANGUAGE plpgsql;
```

7. 调用存储过程。

```
damoxing_rerank=>SELECT insert_documents(ARRAY['This is the first
document.']);
-[ RECORD 1 ]----+-
insert_documents |
damoxing_rerank=> SELECT * FROM retrieve_and_rerank('document');
("{""This is the first document.""}",-1.5107422)
```

使用 Debezium 采集 PostgreSQL 数据

最近更新时间：2025-05-15 17:02:02

业务应用场景中，常常需要实时捕获数据库的数据变更并将其同步至其他系统。该过程可通过 Debezium 实现，Debezium 用于监控数据库变化和捕捉数据变动事件，并以事件流的形式导出。
本文将说明如何使用 Debezium 采集云数据库 PostgreSQL 中的数据。

前提条件

准备处于同一 VPC 下的云数据库 PostgreSQL 实例和云服务器实例。

步骤1:部署环境

1.云服务器配置 java 环境

Debezium 属于 java 应用，需要在云服务器配置 java 环境，为其正常运行提供基础。
依次执行下方命令，下载 jdk18 安装包并解压。

```
#下载jdk8
[root@VM-10-18-tencentos ~]# wget --no-check-certificate --header "Cookie:
oraclelicense=accept-securebackup-cookie" \
https://download.oracle.com/java/18/archive/jdk-18.0.2_linux-x64_bin.tar.gz
#解压安装包
[root@VM-10-18-tencentos ~]# tar -zxvf jdk-18.0.2_linux-x64_bin.tar.gz -C
/usr/local/

#重命名目录
[root@VM-10-18-tencentos ~]# sudo mv /usr/local/jdk-18.0.2 /usr/local/jdk18
```

执行下列命令，进入配置文件内容。

```
[root@VM-10-18-tencentos ~]# vim /etc/profile
```

按 i 键进入编辑模式，在文件末尾添加以下内容：

```
export JAVA_HOME=/usr/local/jdk18
export PATH=$JAVA_HOME/bin:$PATH
export CLASSPATH=.:$JAVA_HOME/lib
```

添加完毕后，按 esc 键退出编辑模式，再输入 :wq 保存修改并退出文件内容。
执行以下命令使配置立即生效。

```
[root@VM-10-18-tencentos ~]# source /etc/profile
```

可通过以下命令检查 java 是否配置成功。

```
[root@VM-10-18-tencentos ~]# java -version
java version "18.0.2" 2022-07-19
Java(TM) SE Runtime Environment (build 18.0.2+9-61)
Java HotSpot(TM) 64-Bit Server VM (build 18.0.2+9-61, mixed mode, sharing)
```

如果显示 java 版本信息，说明配置成功。

2.本地 Kafka 部署

您可选择手动在官网 Apache Kafka 下载自己需要的版本的二进制包（ Binary download ），然后上传到 CVM 上。具体请参见 Linux 系统通过 FTP 上传文件到云服务器 。
也可直接执行以下命令下载，版本号可根据实际需要自行替换。

```
[root@VM-10-18-tencentos ~]# wget
https://downloads.apache.org/kafka/3.7.2/kafka_2.13-3.7.2.tgz
```

将 kafka 安装包下载到 CVM 之后，依次执行以下命令完成安装。

```
#创建kafka的安装目录
[root@VM-10-18-tencentos ~]# mkdir -p /data/zookeeper
#解压kafka安装包
[root@VM-10-18-tencentos ~]# tar -zxvf kafka_2.13-3.7.2.tgz -C /data/

#重命名解压后的目录
[root@VM-10-18-tencentos ~]# cd /data/
[root@VM-10-18-tencentos data]# mv kafka_2.13-3.7.2 kafka_dev
```

执行下行命令，进入 Zookeeper 配置文件。

```
root@VM-10-18-tencentos data]# cd /data/kafka_dev/config
[root@VM-10-18-tencentos config]# vim
/data/kafka_dev/config/zookeeper.properties
```

进入文件内后，按i键进入编辑模式，找到 dataDir ，将其修改为 /data/zookeeper ，确保 dataDir 指向正确的存储路径。

```
dataDir =/data/zookeeper
```

修改完成后，按 esc 键退出编辑模式，再直接输入 :wq 保存修改并退出文件内容。

3.修改 Kafka 配置文件

执行以下命令，创建 kafka 日志目录。

```
[root@VM-10-18-tencentos config]# mkdir -p /data/kafka_dev/logs/
```

执行以下命令，进入 kafka 配置文件。

```
[root@VM-10-18-tencentos config]# vim connect-distributed.properties
```

进入文件后，按 i 键进入编辑模式，修改以下内容。若云服务器与云数据库处于同一 VPC 下，建议填写云服务器的内网 IP 地址。

```
listeners=PLAINTEXT://部署kafka所在机器的ip:9092     #若该项所在行首有#号，需将#删去并修改
log.dirs=/data/kafka_dev/logs/connect.log
zookeeper.connect=部署kafka所在机器的ip:2181
```

执行以下命令，进入 kafka connect 的配置文件 connect-distributed.properties 。

```
[root@VM-10-18-tencentos config]# vim connect-distributed.properties
```

进入文件后，按 i 键进入编辑模式，修改以下内容。若云服务器与云数据库处于同一 VPC 下，建议填写云服务器的内网 IP 地址。

```
group.id=connect-cluster
bootstrap.servers=部署kafka所在机器的ip:9092
# 定义插件路径
plugin.path=/data/kafka_connect/plugins
```

4.启动 Zookeeper 和 Kafka

使用以下命令启动 zookeeper 。

```
[root@VM-10-18-tencentos config]# cd /data/kafka_dev
```

```
[root@VM-10-18-tencentos kafka_dev]# nohup bin/zookeeper-server-start.sh
config/zookeeper.properties &> zookeeper.log &
```

可输入以下命令确认 zookeeper 任务是否正常在后台运行。

```
[root@VM-10-18-tencentos kafka_dev]# jobs
```

若返回信息包含 zookeeper 和 running ，则正常运行。
再执行以下命令启动 kafka 。

```
[root@VM-10-18-tencentos kafka_dev]# nohup bin/kafka-server-start.sh
config/server.properties &> kafka.log &
```

可输入以下命令确认 kafka 任务是否正常在后台运行。

```
[root@VM-10-18-tencentos kafka_dev]# jobs
```

若返回信息包含 kafka 和 running，则正常运行。

步骤2:在 PostgreSQL 中创建逻辑复制发布

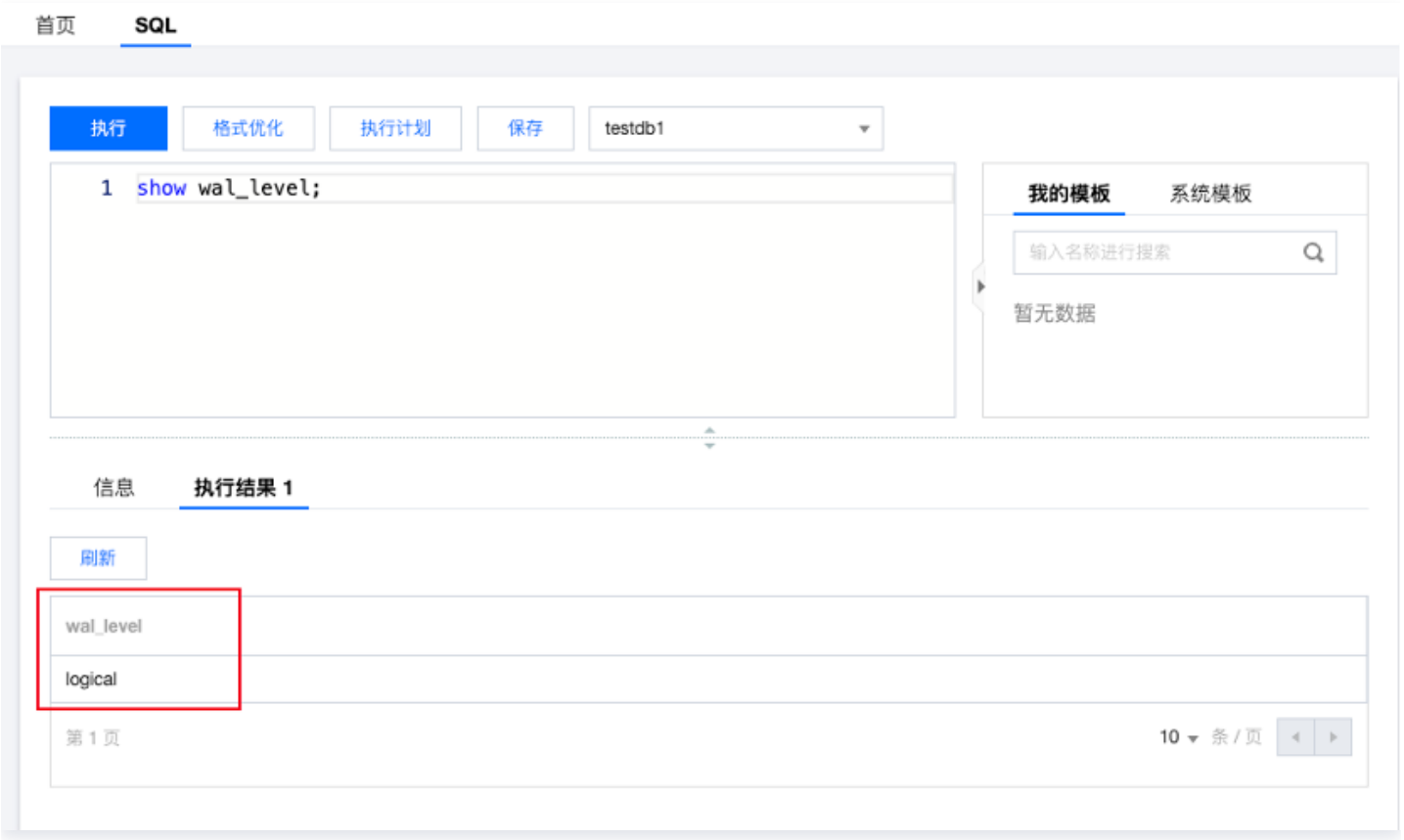
逻辑发布（ Publication ）定义了哪些表的数据变更会被发布，Debezium 通过绑定到逻辑发布上的逻辑复制槽（ Failover Slot ）来捕获变更数据。对于逻辑复制槽的具体说明请参见 逻辑复制槽故障转移（Failover Slot）。因此，您需要创建 publication 和 failover slot ，才能实现数据的捕获和同步。

1.开启逻辑复制

进入控制台，找到需要采集数据的实例，在实例详情页面点击参数设置，将 wal_level 参数默认值修改为为 logical ，参数值修改后需要重启实例才能生效。

修改后，可登录实例，使用以下查询语句查看 wal_level 是否修改成功。

```
show wal_level;
```



2.创建 Publication（逻辑发布）

使用类型为 pg_tencentdb_superuser 的账号，登录需要发布的数据库控制台。执行以下命令创建 publication 。

```
CREATE PUBLICATION pg_demo_publication FOR ALL TABLES;
```

其中， pg_demo_publication 指该 publication 的名称，您可自行定义， FOR ALL TABLES 指将当前数据库中的全部表都进行发布。若您需要指定发布哪几张表，则可选择执行以下命令：

```
CREATE PUBLICATION pg_demo_publication FOR table_name1, table_name2;
```

您可执行以下命令查看刚刚创建的 publication ，确认要发布的表。

```
SELECT * FROM pg_publication_tables WHERE pubname = 'pg_demo_publication';
```

若希望确认有哪些操作会进行发布，可执行以下命令查看。

```
SELECT * FROM pg_publication WHERE pubname = 'pg_demo_publication';
```

pg_publication 表用于存储所有已创建的 publication 信息。其中， puballtables 列为 true ，则表示发布数据库中的所有表； pubinsert 列为 true ，表示发布表的 insert 操作，其他列相同。
执行以下命令，创建 tencentdb_failover_slot 插件。

```
CREATE EXTENSION tencentdb_failover_slot;
```

执行以下命令，创建 logical_failover_slot 。

```
SELECT pg_create_logical_failover_slot('failover_alot_name','pgoutput');
```

创建完成后，可使用以下命令查看 Failover Slot 信息。

```
SELECT * FROM pg_failover_slots;
```

步骤3:开启 Debezium

1.安装 Debezium 插件

登录云服务器，依次执行以下命令下载 debezium-connector-postgresql 插件，并解压到指定路径。

```
[root@VM-10-18-tencentos ~]# mkdir -p /data/kafka_connect/plugins
[root@VM-10-18-tencentos ~]# wget
https://repo1.maven.org/maven2/io/debezium/debezium-connector-
postgres/2.7.3.Final/debezium-connector-postgres-2.7.3.Final-plugin.tar.gz
[root@VM-10-18-tencentos ~]# tar -zxvf debezium-connector-postgres-
2.7.3.Final-plugin.tar.gz -C /data/kafka_connect/plugins
```

2.启动 kafka connect

启动 kafka connect 前，请确保已经启动了 kafka 。启动及确认方法请参见 步骤1 。
执行以下命令启动 kafka connect 。

```
[root@VM-10-18-tencentos ~]# cd /data/kafka_dev
[root@VM-10-18-tencentos kafka_dev]# nohup bin/connect-distributed.sh
config/connect-distributed.properties &> connect.log &
```

3.创建 debezium connector

在云服务器执行下面的命令，进行 debezium connector 的创建。需要自行根据实际情况填写的项已为您标出。

说明：

若云服务器与云数据库 PostgreSQL 处于同一 VPC 下，建议填写云服务器机器、云数据库的内网 IP 。

```
curl -XPOST "http://部署kafka所在云服务器的ip:8083/connectors/" \
-H 'Content-Type: application/json' \
-d '{
    "name": "test_connector",
    "config": {
                "connector.class":
"io.debezium.connector.postgresql.PostgresConnector",
                "database.hostname": "PostgreSQL数据库所在机器的IP",
                "database.port": "5432",
                "database.user": "有发布权限的PostgreSQL用户名",
                "database.password": "发布用户的密码",
                "database.dbname": "创建publication的数据库",
                "database.server.name": "pg_demo",
                "slot.name": "pg_demo_failover_slot",
                "topic.prefix": "pg_demo",
                "publication.name": "pg_demo_publication",
                "publication.autocreate.mode": "all_tables",
                "plugin.name": "pgoutput"
        }
    }'
```

需要您自行填写及可自定义的参数说明如下：

参数	说明
name	连接器的名称，必须唯一。
database.hostname	云数据库的 IP 地址。建议您填写内网 IP。
database.user	用于连接云数据库的用户名。 该用户需要有足够权限完成发布。推荐使用类型为 pg_tecenten_superuser 的用户。
database.password	用户的密码。
database.dbname	创建 publication 的数据库名称。
slot.name	逻辑复制槽的名称。 请填写之前创建的逻辑复制槽名称。

publication.name	逻辑发布的名称。 请填写之前创建的逻辑发布名称。

您可登录控制台，使用以下命令查看 publication 。

```
SELECT * FROM pg_publication;
```

可使用以下命令查看 failover Slot 信息。

```
SELECT * FROM pg_failover_slots;
```

创建完毕后，您可通过在云服务器输入以下命令查看连接状态。

```
curl "http://部署kafka所在机器的ip:8083/connectors/pg_demo_connector/status"
```

返回的信息中包含 running 则为正常运行。

步骤4:测试数据变更

执行以下命令在 CVM 中登录云数据库。其中 -h 后填写为云数据库 IP。若云服务器与云数据库处于同一 VPC 下，建议填写为内网 IP。

```
[root@VM-10-18-tencentos kafka_dev]# su - postgres
[postgres@VM-10-18-tencentos ~]$ /usr/local/pgsql/bin/psql -h *.*.*.* -p
5432 -U dbadmin -d postgres
Password for user dbadmin:
psql (16.4, server 16.8)
Type "help" for help.

postgres=>
```

创建一张新表，插入数据进行测试。

```
postgres=> CREATE TABLE linktest (
    id SERIAL PRIMARY KEY
);
CREATE TABLE
postgres=> insert into linktest values(1);
INSERT 0 1
```

观察 kafka 日志，若 kafka 正常，则连接建立成功。

在 CVM 本地搭建 PostgreSQL 异地灾备环境

最近更新时间：2025-05-15 17:02:02

背景

为保证业务正常运行，一个高可用的数据库架构是不可或缺的。数据库一旦出现丢失、不可用等问题，将会造成重大影响和经济损失。通过主备架构，当主数据库因突发硬件故障而无法工作时，备用数据库可立即接管服务，保证数据库服务能够正常提供。而在 CVM 自建 PostgreSQL 灾备环境，则能够将云上数据同步至本地，提高对不可抗力的应对能力。
本文主要介绍如何搭建云数据库 PostgreSQL 为主数据库，云服务器 CVM 自建 PostgreSQL 为备数据库的主备环境。

前置条件

云服务器 CVM 中必须安装与云数据库**相同**版本的 PostgreSQL 数据库，详细安装教程请参考 PostgreSQL 官方文档 。Linux 系统的云服务器配置请参见 快速配置 Linux 云服务器 。示例使用 PostgreSQL 14 版本。

> **注意：**
> 本方法需要用到 pg_basebackup 工具，两个工具均仅支持13及以上的 PostgreSQL 版本。

步骤1: 将云数据库主库数据物理备份到 CVM 本地备库

1. 使用 root 账号登录云服务器 CVM ，执行以下命令安装中文字符集。

   1.1. 若您的 CVM 系统为 Debian/Ubuntu ,请使用 root 权限使用以下命令：

   ```
   [root@VM-10-18-tencentos ~]# sudo locale-gen zh_CN.UTF-8
   ```

   1.2. 若您的 CVM 系统为 CentOS/TencentOS ,请使用 root 权限执行以下命令进入配置文件内容：

   ```
   [root@VM-10-17-tencentos ~]# vi /etc/locale.conf
   ```

   按 i 键进入编辑模式，将文件内容修改为：

   ```
   LANG="zh_CN.UTF-8"
   ```

   按 esc 键退出编辑模式，再直接输入 :wq 保存修改并退出。退出后，执行以下命令使配置生效：

```
[root@VM-10-17-tencentos ~]# source /etc/locale.conf
```

1.3. 安装完毕，您可通过以下命令查看字符集是否安装成功：

```
[root@VM-10-17-tencentos ~]# locale -a | grep zh_CN
zh_CN
zh_CN.gb18030
zh_CN.gb2312
zh_CN.gbk
zh_CN.utf8
```

若返回信息前缀为 zh ，则安装成功。

2. 创建普通权限用户。若您的本地数据库中已经存在普通权限用户，可跳过本步骤。

> ⚠ **注意：**
> 创建备份数据目录时，必须使用**普通权限用户**。PostgreSQL 出于安全考虑，启动命令 pg_ctl 不允许以 root 或 postgres 超级用户身份运行。

使用以下命令完成创建普通用户 pgsql 。

```
[root@VM-10-6-tencentos pgsql]# useradd -r -s /bin/bash pgsql
```

3. 通过重命名目录，将当前云服务器中的数据目录进行备份。

系统默认数据目录为 /usr/local/pgsql/data ，您可根据实际情况修改命令。
执行以下命令，使用普通用户 pgsql ，停止当前 PostgreSQL 服务运行：

```
[root@VM-10-17-tencentos ~]# su - pgsql #使用普通用户 pgsql 登录本地数据库
[pgsql@VM-10-17-tencentos ~]$ cd /usr/local/pgsql/data
[pgsql@VM-10-17-tencentos data]$ /usr/local/pgsql/bin/pg_ctl stop  -D
/usr/local/pgsql/data
waiting for server to shut down.... done
server stopped
```

执行以下命令，重命名数据目录，将其作为备份。示例备份目录名为 data-bak-2025 ，可根据实际情况修改。

```
[pgsql@VM-10-6-tencentos postgresql-14.2]# mv /usr/local/pgsql/data
/usr/local/pgsql/data-bak-2025
```

重新新建一个数据目录 /usr/local/pgsql/data 作为数据库存放数据的目录。将新的数据库目录权限修改为700 ，即仅限目录所有者拥有读、写和执行权限，其他用户无操作权限：

```
[pgsql@VM-10-6-tencentos postgresql-14.2]# cd /usr/local/pgsql
[pgsql@VM-10-6-tencentos pgsql]# mkdir data
[pgsql@VM-10-6-tencentos pgsql]# chmod 700 data
```

4. 执行以下命令，使用 pg_basebackup 工具在线备份主库数据。

   其中，-h 参数后需填写云数据库的 IP 地址，-U 参数后需填写连接数据库的账号，此处为 dbadmin 。若云数据库与云服务器处于同一 VPC 下，则建议填写云数据库的内网 IP 地址。

```
[pgsql@VM-10-6-tencentos pgsql]# exit
[root@VM-10-6-tencentos pgsql]# date &&
/usr/local/pgsql/bin/pg_basebackup -R -Xs -D /usr/local/pgsql/data -h
*.*.*.* -p 5432 --verbose -P -U dbadmin && date
 Password:
```

   提示 Password: ，输入连接数据库账号的密码，按回车键即可。
   最终显示信息 pg_basebackup: base backup completed 则代表物理备份成功。

步骤2: 修改配置文件，添加主备链路

1. 执行以下命令，进入 postgresql.conf 配置文件中。

```
[root@VM-10-6-tencentos pgsql]# vi /usr/local/pgsql/data/postgresql.conf
```

   按 i 键进入编辑模式，找到以下内容，在每行的行首添加英文字符 # 将其注释掉。

```
synchronous_standby_names
extension_blacklist
tencentdb_enable_copy_to
tencentdb_relcache_evict_num
tencentdb_relcache_max_num
soft_limit_connections
shared_preload_libraries
tencentdb_syscache_max_num
basebackup_exclude_paths
tencentdb_syscache_evict_num
tencentdb_enable_trusted_extension
tencentdb_az_five
disable_dblink_connect_to_other
tencentdb_enable_superuser_unsafe_behaviour
```

> **说明：**

> 开源环境无法识别特殊参数，请您将配置文件中所有 tencentdb 为前缀的项全部注释掉。

并且，修改以下项：

```
port = 5432 #将原本的端口号修改为5432
log_directory = '/usr/local/pgsql/logs' #可根据实际需要调整
log_destination = 'csvlog' #将原本的'csvlog,auditlog'修改为log_destination
= 'csvlog'
```

最后，在文件末尾添加两行：

```
unix_socket_directories = '/usr/local/pgsql/data' #可根据实际需要调整
primary_conninfo = 'host=*.*.*.* port=5432 user=dbadmin password=对应密码'
#host填写云数据库IP地址，user填写连接数据库的账号名
```

修改完毕后，按 esc 键退出编辑模式，并输入 :wq 保存修改并退出文件。

2. 执行以下命令，进入 pg_hba.conf 配置文件中。

```
[root@VM-10-6-tencentos pgsql]# vi /usr/local/pgsql/data/pg_hba.conf
```

按 i 键进入编辑模式，找到以下内容，在行首添加英文字符 # 将其注释掉。

```
host    all     postgres  0.0.0.0/0     reject
```

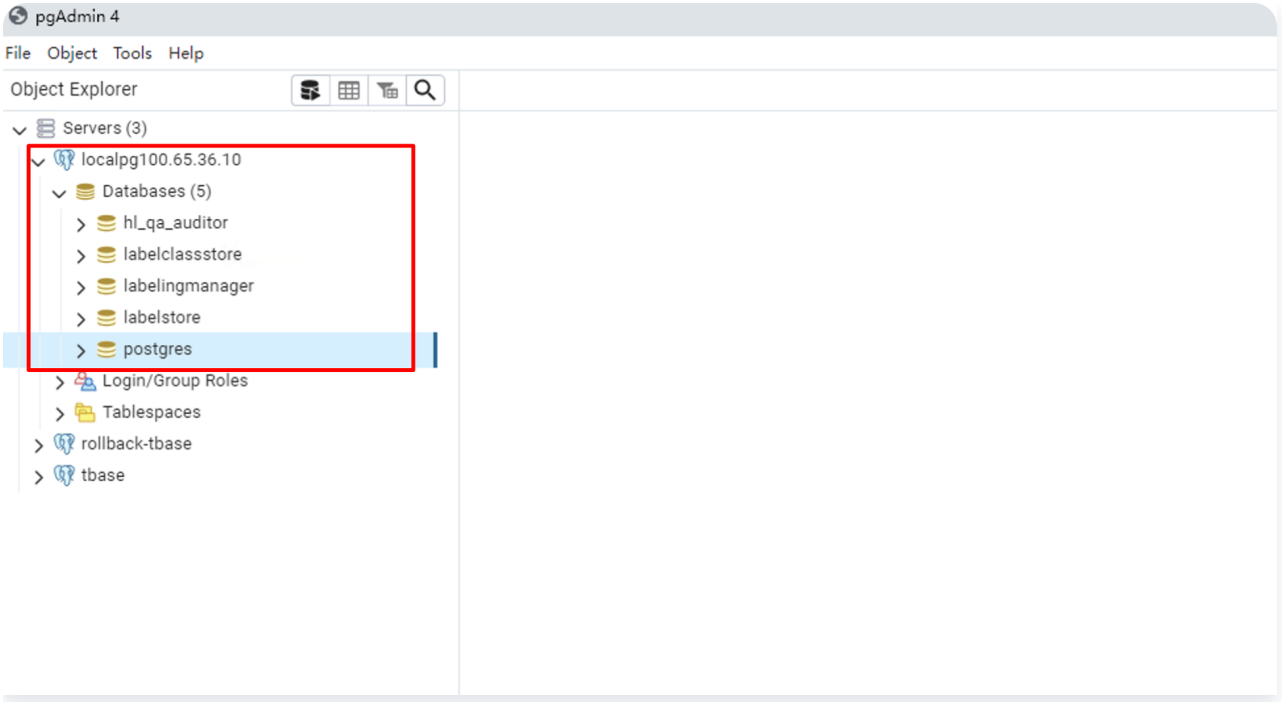
修改完毕后，按 esc 键退出编辑模式，并输入 :wq 保存修改并退出文件。

步骤3: 启动数据库，查看备份数据

使用普通用户 pgsql 账号登录，启动云服务器本地的数据库：

```
[root@VM-10-6-tencentos pgsql]# su - pgsql
[pgsql@VM-10-6-tencentos pgsql]$ /usr/local/pgsql/bin/pg_ctl -D
/usr/local/pgsql/data -l logfile start
waiting for server to start.... done
server started
```

在 pgAdmin 客户端连接到云服务器的数据库，可以查看到物理备份的数据。

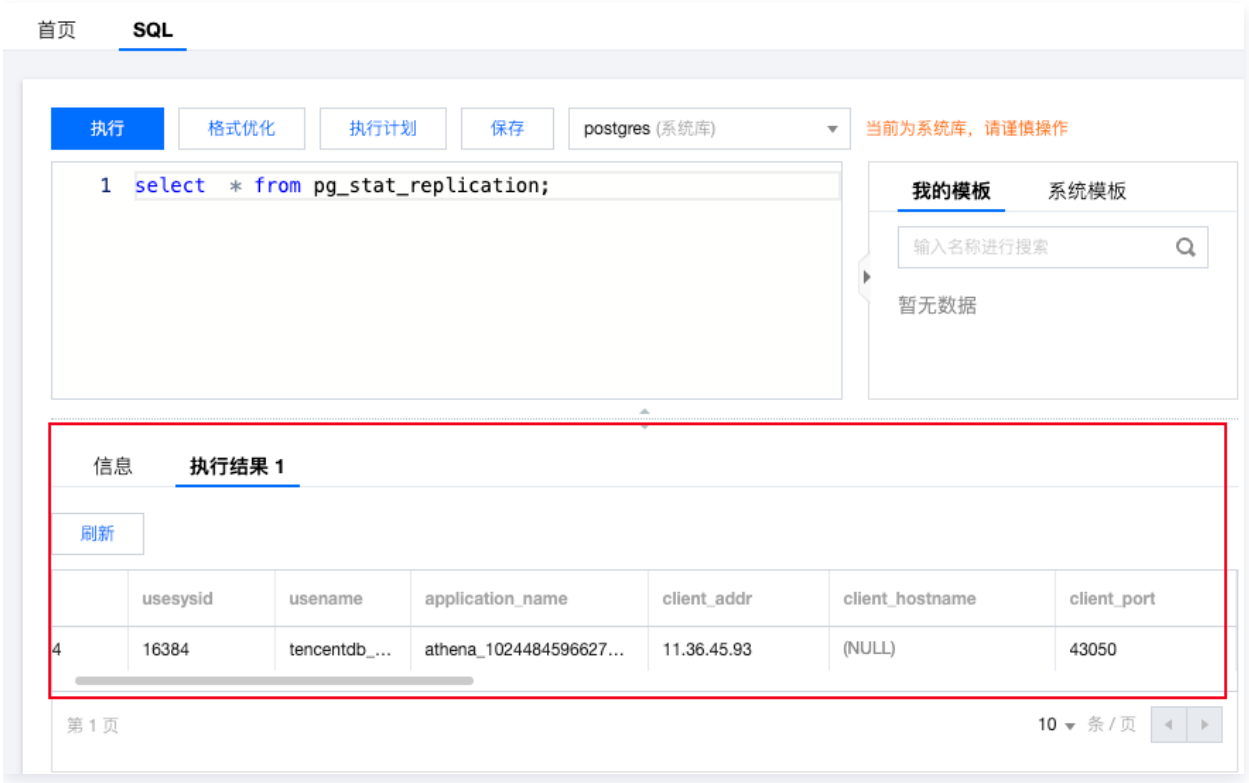


步骤4: 验证主备链路

登录 云数据库 PostgreSQL 控制台，登录数据库实例，执行以下 SQL 语句：

```
select  * from pg_stat_replication;
```

如下图所示，执行结果将显示刚才建立的主备链路。



主备环境搭建成功。