云数据库 PostgreSQL

最佳实践

Contents

最佳实践
如何在 PostgreSQL 中自动创建分区

Last updated：2023-05-17 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 ATTA(
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
$$
DECLARE
    dateStr varchar;
BEGIN
    SELECT to_char(DATE 'tomorrow', 'YYYYMMDD') INTO dateStr;
    EXECUTE
        format('CREATE TABLE tab_%s (LIKE tab INCLUDING INDEXES)', dateStr);
    EXECUTE
        format('ALTER TABLE tab ATTACH PARTITION tab_%s FOR VALUES IN (%L)', dateStr, date
    RETURN 1;
END;
$$;

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, 'Y

      /*
       * 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\')' % (t


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

Last updated：2023-06-07 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,
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_strin
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.
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=4.2
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=
-> Bitmap Heap Scan on account (cost=109.08..21800.56 rows=6332 width=8) (actual ti
901 rows=5390 loops=1)
Recheck Cond: (tag && '{lvXe,Zt}'::text[])
Heap Blocks: exact=5327
-> Bitmap Index Scan on tag_inx (cost=0.00..107.49 rows=6332 width=0) (actual time=(
.0.962 rows=5390 loops=1)
Index Cond: (tag && '{lvXe,Zt}'::text[])
Planning Time: 0.110 ms
Execution Time: 8.270 ms
```

方案2：优化方案

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

1. 引入一个新的标签字典表。

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

2. 假设一共有10W种字典类型。

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

3. 创建一个新表用来存储用户和标签信息。

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

4. 插入1000W个账号数据。

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

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
Workers Planned: 2
Workers Launched: 2
-> Parallel Seq Scan on account1 (cost=0.00..189982.68 rows=104 width=19) (actual tin
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
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=
Recheck Cond: (tag @> '{100,5711}'::integer[])
-> Bitmap Index Scan on tag_inx_2 (cost=0.00..49.87 rows=250 width=0) (actual time=(
Index Cond: (tag @> '{100,5711}'::integer[])
```

```
Planning Time: 0.410 ms
Execution Time: 0.171 ms
(6 rows)
```

6. 查找同时有标签 ID 为61568，97350的用户列表。

```
test=> explain analyze select uin,name from account1 where tag @> ARRAY[61568,973
QUERY PLAN
------------------------------------------------------------------------------------------------------------
Bitmap Heap Scan on account1 (cost=49.94..1021.13 rows=250 width=19) (actual time=
Recheck Cond: (tag @> '{61568,97350}'::integer[])
Heap Blocks: exact=1
-> Bitmap Index Scan on tag_inx_2 (cost=0.00..49.87 rows=250 width=0) (actual time=(
Index Cond: (tag @> '{61568,97350}'::integer[])
Planning Time: 0.071 ms
Execution Time: 0.151 ms
(7 rows)
```

7. 查找与 xx 有共同爱好（标签100和5711）的人有 xx 个。

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

------------------------------------------------------------------------------------------------------------
------
Gather (cost=1961.06..173801.15 rows=99750 width=19) (actual time=5.020..28.885 rc
Workers Planned: 2
Workers Launched: 2
-> Parallel Bitmap Heap Scan on account1 (cost=961.06..162826.15 rows=41562 width=
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 tim
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 r
Execution Time: 29.725 ms
(14 rows)
```

方案3：roaringbitmap

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

可。

```
create extension roaringbitmap;
```

2. 创建标签用户对应表。

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

3. 根据之前的标签表插入10W条标签以及标签对应的用户数据。

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

4. 查询标签有1，3，10，200的用户个数。

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


QUERY PLAN

------------------------------------------------------------------------------------------------------------------------
------------
Aggregate (cost=32.47..32.48 rows=1 width=32) (actual time=0.964..0.966 rows=1 loop
-> GroupAggregate (cost=32.42..32.46 rows=1 width=12) (actual time=0.955..0.956 rov
Group Key: tag_uin_list.uin_offset
-> Sort (cost=32.42..32.43 rows=4 width=22) (actual time=0.107..0.109 rows=4 loops=
```

```
Sort Key: tag_uin_list.uin_offset
Sort Method: quicksort Memory: 25kB
-> Bitmap Heap Scan on tag_uin_list (cost=17.20..32.38 rows=4 width=22) (actual time=
)
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 tir
=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=
Group Key: uin_offset
-> Sort (cost=32.42..32.43 rows=4 width=22) (actual time=0.043..0.045 rows=4 loops=
Sort Key: uin_offset
Sort Method: quicksort Memory: 25kB
-> Bitmap Heap Scan on tag_uin_list (cost=17.20..32.38 rows=4 width=22) (actual time=
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 tir
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 whe
  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_index
   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

Last updated：2023-05-17 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
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_make
```

步骤4：查询附近的人

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

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

执行命令：

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

3. 查看距离此坐标对象1000米以内的所有对象与距离。

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

如何配置云数据库 PostgreSQL 作为 GitLab 外部数据源

Last updated：2023-05-17 16:00:13

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
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 的帐号，如 gitlab，请注意此帐号必须拥有 superuser 权限或者云数据库所给与的管理员权限，如腾讯云的 `pg_tencentdb_superuser` 。

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

4. 然后创建 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。

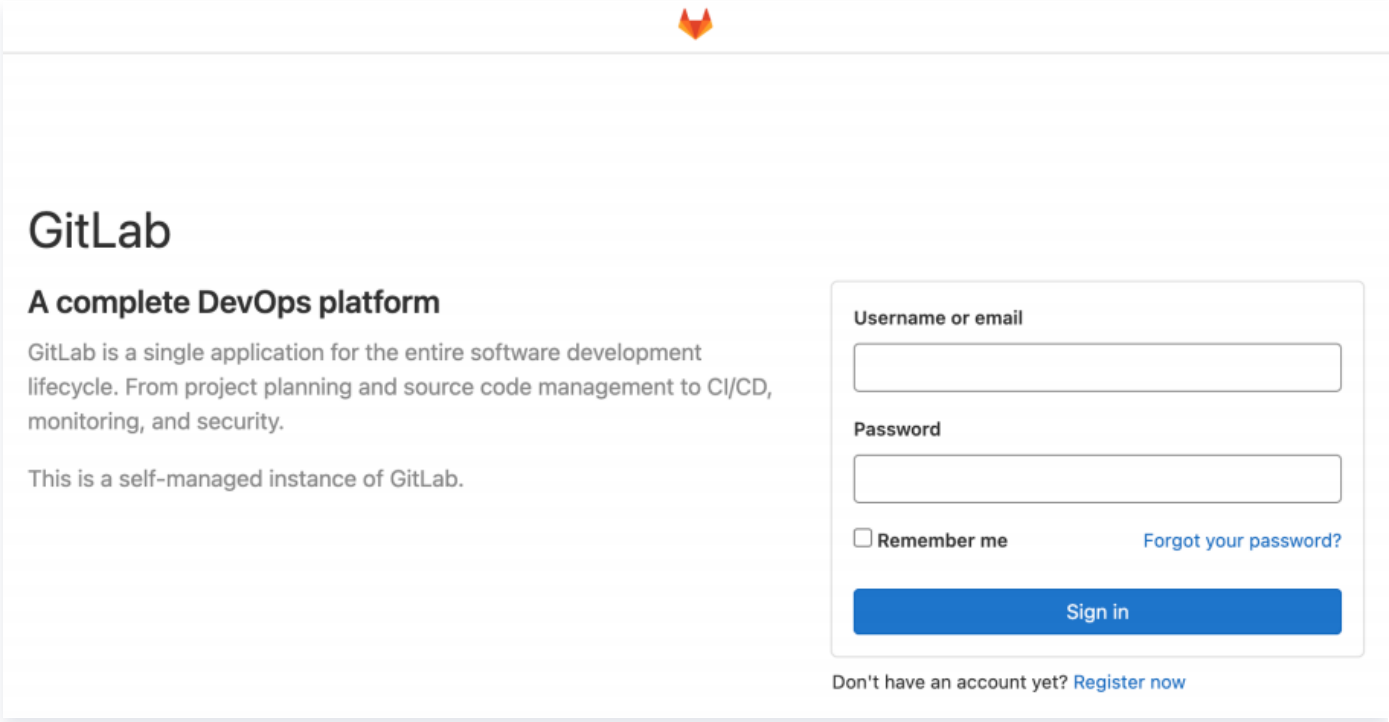
```
gitlab-ctl reconfigure
```

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. 初始登录帐号为 root，初始密码在初始完成后，会有如下的提示：

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

> **说明**
> 可在服务器中的此　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

Last updated：2023-05-17 16:00:13

背景

数据库作为数据存放、处理、加 的核 组件，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

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. PostgerSQL 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 htt
• 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:
OGVmYzZiMmQzYjA2OWNhODk0NTRkMTBiOWVmMDAxODc0OWRkZjk0ZDM1NmI1M2E2MTF
```