

Project

方酉城 12310519

摘要：opengauss是华为基于postgresql开发的关系型数据库，本次project目的旨在比较opengauss与postgresql的差异不同。主要从**并发性能，吞吐量对比，查询延迟与执行时间，资源内存占用，安全性，可拓展性**方面来进行综合的比较评测。

Github:<https://github.com/niko-shiybu/DBProject3.git>

1.性能测试

1.1吞吐量对比 (TPS)

事务吞吐量 (TPS, Transactions Per Second) 为测量数据库在单位时间内可以处理的事务数量，通过这个指标可以了解系统在不同负载下的处理能力。由于openGauss是基于PostgreSQL进行开发演化，对于PostgreSQL数据库，采用PostgreSQL自带的压力测试工具：**pgbench**，通过设置不同的用户数量及事务数量进行模拟并发用户进行事务处理，**记录每秒完成的事务数**。而对与openGuass,我们通过sql的脚本文件进行事务的模拟测试设置相同的用户数与事务数。通过以上测试，对比两个数据库的并发性能中吞吐量的对比。其中TPS值越高，表示数据库能够在单位时间内处理更多的事务，通常意味着数据库性能较好，能够处理更多并发的用户请求。

(1)在opengauss与postgresql中分别设置相同的事务数量，比较不同数量的用户数下的TPS数值（简单查询模式）

在PostgreSQL进行测试，设置用户数量为10, 30, 50, 70, 事务数量为10000.先进行初始化的操作，执行以下语句进行初始化,创建 pgbench 所需的表和一些测试数据

```
1 | pgbench -i -h localhost -U postgres -d project3
```

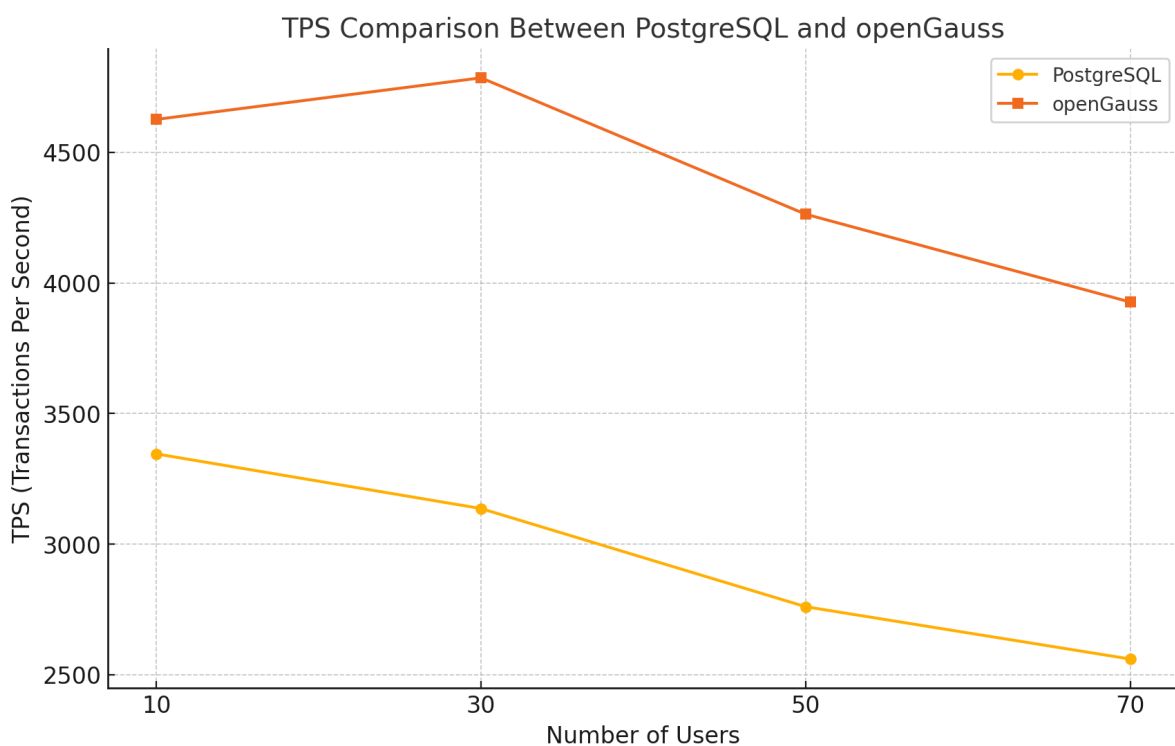
之后，进行pgbench测试(分别输入以下语句)

```
1 pgbench -h localhost -U ppostgres -d project3 -c 10 -t
  10000
2
3 pgbench -h localhost -U ppostgres -d project3 -c 30 -t
  10000
4
5 pgbench -h localhost -U ppostgres -d project3 -c 50 -t
  10000
6
7 pgbench -h localhost -U ppostgres -d project3 -c 70 -t
  10000
```

从终端获取postgresql的测试结果，在用户数为10，30，50，70下，tps的值分别为3345.2462，3136.076098，2760.234296，2559.934119。

接下来进行opengauss的相应测试，由于opengauss内部没有内置的pgbench，无法直接使用工具测试，所以我通过编写java文件连接至opengauss然后模拟多用户并行执行简单查询事务，并计算相应的tps。首先创造测试数据表 test_accounts、test_branches 或 test_history，然后通过sql语句向表中插入随机数据，编写java文件连接数据库，模拟多名用户在处理多种事务数的情况，分别测试用户数为10，30，50，70.对应tps的值分别为4626.09580，4785.07821，4263.05341，3926.75302

画出两个数据库不同用户数下的关系图



结论：分析数据可得，从图表可以看出，无论是在 10 个用户、30 个用户、50 个用户，还是 70 个用户的情况下，**openGauss 的 TPS（每秒事务数）始终显著高于 PostgreSQL**。这表明，在相同的测试条件下，openGauss 的吞吐能力优于 PostgreSQL。这种差异可能归因于 openGauss 在底层性能优化方面的改进，例如更高效的事务管理机制、更低的锁争用设计以及更好的多线程支持

(2)在opengauss与postgresql中分别设置相同的用户数量，比较不同数量的事务数下的TPS数值（简单查询模式）

在PostgreSQL进行测试，设置用户数量为10，30，50，70，事务数量为10000.先进行初始化的操作，执行以下语句进行初始化,创建 pgbench 所需的表和一些测试数据.

```
1 | pgbench -i -h localhost -U postgres -d project3
```

之后，进行pgbench测试(分别输入以下语句)

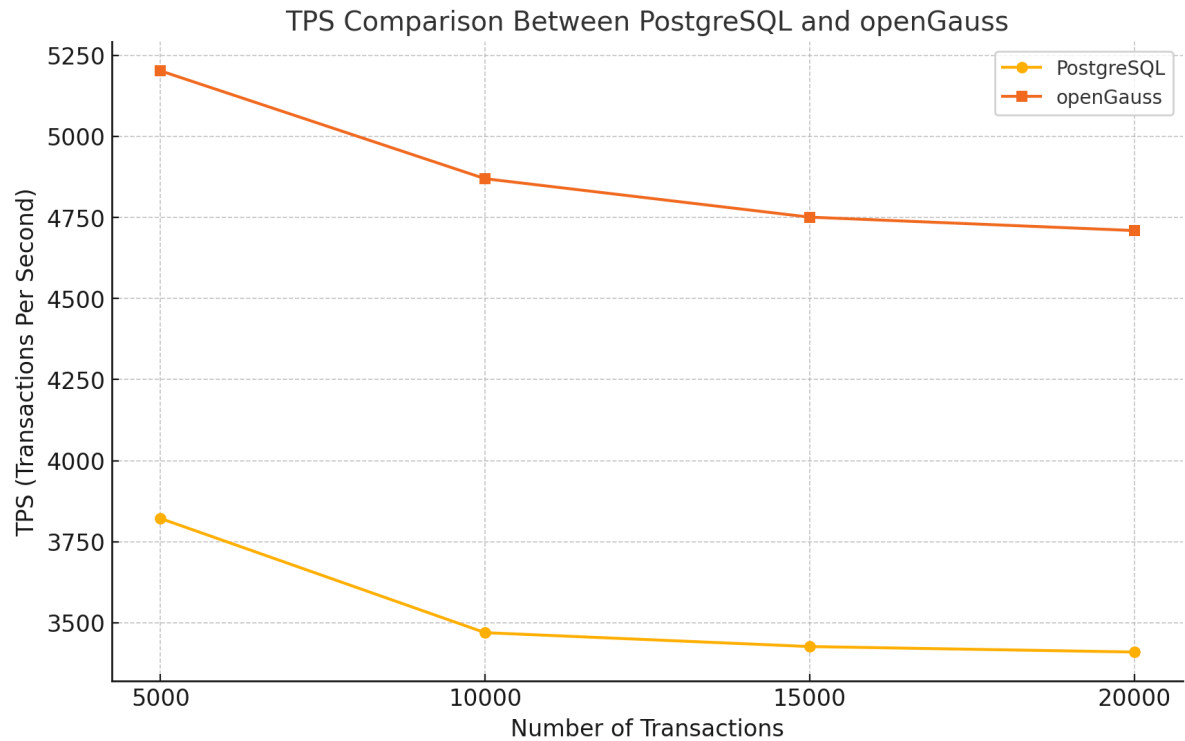
```
1 | pgbench -h localhost -U postgres -d project3 -c 10 -t 5000
2 |
3 | pgbench -h localhost -U postgres -d project3 -c 10 -t 10000
4 |
5 | pgbench -h localhost -U postgres -d project3 -c 10 -t
  | 150000
6 |
7 | pgbench -h localhost -U postgres -d project3 -c 10 -t
  | 200000
```

从终端获取得postgresql的测试结果，在事务数为5000，10000，15000，20000下，tps的值分别为3821.560342， 3469.230252， 3426.383571， 3409.507458.

接下来通过Java文件来对openGuass进行测试

在事务数为5000，10000，15000，20000下，tps的值分别为5202.361202, 4869.631342, 4751.043521, 4709.72308.

画出两个数据库不同事务数下的关系图



结论：分析数据可得，通过测试 PostgreSQL 和 openGauss在不同事务数量（5000、10000、15000、20000）下的 TPS（每秒事务数）性能，可以得出 **openGauss 的 TPS 在所有事务数量下均显著高于 PostgreSQL，表现出更高的吞吐能力**

(3) 总结结论

openGuass比postgresql具有更高的吞吐量（tps）,高并发场景下表现良好，能够有效地管理多个并发用户或线程的事务操作,适合需要支持大量并发用户访问的系统，例如社交媒体、在线视频流平台和实时分析系统.

1.2查询延迟与执行时间

1.2.1数据集的准备

为了更好地模拟数据库的使用场景，并支持性能测试和功能验证，编写了一个 Python 脚本，利用工具库 **psycopg2** 和 **Faker** 来随机生成和导入大规模的测试数据。该脚本主要面向电商系统的订单数据，模拟了包括订单号、商品类别、商品价格、订单时间等多个常见字段，总计生成了 50 万条数据。通过这些数据，我们可以更真实地还原电商系统中的高并发、高流量场景。

在数据生成过程中，**Faker** 库被用来生成随机且逼真的信息。例如，订单号作为唯一标识，商品类别随机选择（如电子产品、家居用品等），商品价格则在 10.00 元到 1000.00 元之间随机浮动，订单时间和描述均为模拟生成。整个数据表的结构包括了订单状态（是否有效）、购买数量等字段，适合用于测试插入和查询性能。

生成完成后，脚本通过 **psycopg2** 与数据库交互，批量插入这些数据以提高效率（通过 `executemany` 方法一次性插入多条记录）。针对 50 万条数据的插入操作，脚本充分利用了事务机制来保证数据的原子性和一致性。数据插入完成后，可以通过简单的 SQL 语句对表数据进行验证，例如统计订单总数或查询特定类别的订单。

通过这个脚本，我们成功模拟了电商系统中的订单数据，为数据库性能测试（如插入速度、查询延迟、并发处理能力）提供了高质量的数据环境。

1.2.2查询延迟与执行时间测试

- 简单查询

对于postgresql和opengauss,统一进行相同的两种查询测试，记录对于五十万条数据查询的延迟与执行时间

- 查询有效订单的总数

```
1 | SELECT COUNT(*) FROM test_data WHERE is_active = TRUE;
```

记录postgresql和opengauss的查询时间t1,t2

- 查询类别为“Sports”的订单总数

```
1 | SELECT COUNT(*) FROM test_data WHERE category = 'Sports';
```

记录postgresql和opengauss的查询时间t1,t2

结果如下

查询方式/时间	opengauss	postgresql
查询有效订单的总数/ t1	276ms	283 ms
查询类别为“Sports”的订单总数 / t2	98ms	101ms

结论：在简单查询的任务中，两个数据库对于查询的执行时间几乎没有差别。

- 聚合操作查询

由于聚合操作涉及到计算方面，对于postgresql和opengauss,统一进行查询类别为“Sports”的商品价格均值，记录对于五十万条数据查询的延迟与执行时间

```
1 | SELECT AVG(price) FROM test_data WHERE category =  
   | 'Sports';
```

记录postgresql和opengauss的查询时间t1,t2

结果如下

时间	opengauss/t2	postgresql/t1
	190ms	189ms

结论：在聚合操作查询的任务中，两个数据库对于查询的执行时间几乎没有差别

1.2.3结论

对于简单查询，openGuass与postgresql 的执行时间效率相同，无论是简单查找还是聚合函数进行计算查找，从这方面来说他们的性能并无差别

2.资源内存占用


2.1监控资源工具的准备

为了对比比较Postgresql与openGuass的两个数据库在不同情况的下，考虑对比他们的资源占用情况，这是衡量数据库性能好坏的一项重要指标。因此，我选择windows的一个小型轻量级的检测工具Process Explorer，它可以显示数据库的cpu使用率，内存占用指标，线程数量等资源占用的指标，通过使用检测工具Process Explorer以及任务资源管理器，以此来判断对比两个数据库在资源内存上的消耗情况。对于opengauss我们通过查看docker中的资源使用率来获取结果

2.2测试过程

(1) 使用 Process Explorer 监控空闲状态

通过断掉空闲的用户连接，来评估两个数据库在无活动时的基础内存占用，以下分别为postgresql和openGuass的资源消耗检测



















	postgres.exe	< 0.01	3,124 K	15,224 K	10244		
CONTAINER ID	NAME	CPU %	MEM USAGE / LIMIT	MEM %	NET I/O	BLOCK I/O	PIDS
f4c7f52ed42e	project3-opengauss	0.91%	857.6MiB / 7.402GiB	11.32%	124kB / 289kB	0B / 0B	35

排除docker可能占用的网络连接消耗，postgresql与openGuass在空闲状态下的cpu的占比消耗都很小。postgresql的内存占比 3124 K 约等于 3 MB，opengauss约为898MB，从内存占用来说，opengauss空闲状态下的内存占用会比postgresql高

(2) 高并发查询测试

对模拟高并发查询的测试，我们仍然通过决定用户数与事务数量来进行测试，由于不同数量的用户与事务数量影响的事实上是总事务数量，所以本次测试只是改变用户数量而不改变事务数量。对于Postgresql,我仍然使用pgbench来进行测试，对于openGuass,仍然通过使用java连接数据库的形式进行测试。以下测试结果

- 用户10，事务数10000 (Postgresql)

Process	CPU	Private Bytes	Working Set	PID	Description
 postgres.exe	0.20	4,960 K	31,792 K	24500	
 postgres.exe	0.26	4,992 K	31,708 K	24460	
 postgres.exe	0.20	4,988 K	31,628 K	24276	
 postgres.exe	0.20	4,984 K	31,692 K	24116	
 postgres.exe	0.07	4,984 K	31,768 K	24004	
 postgres.exe		5,876 K	15,564 K	23364	
 postgres.exe		3,704 K	11,636 K	23320	
 postgres.exe	0.26	4,952 K	31,548 K	23096	
 postgres.exe		3,604 K	10,376 K	23000	
 postgres.exe	0.33	5,016 K	31,548 K	22608	
 postgres.exe	0.33	4,960 K	31,592 K	20628	
 postgres.exe	< 0.01	3,188 K	10,836 K	15772	
 postgres.exe	0.13	4,980 K	31,696 K	14088	
 postgres.exe		3,788 K	22,404 K	11496	
 postgres.exe		3,656 K	10,160 K	11184	
 postgres.exe	< 0.01	3,124 K	13,932 K	10244	
 postgres.exe	0.13	4,984 K	31,604 K	9760	
 postgres.exe		2,700 K	9,392 K	8884	

PostgreSQL Server (19)

13.5%0.4%19.8 MB/...0 Mbps

> PostgreSQL Server (20)	17.7%	0.4%	20.4 MB/...	0 Mbps
> PostgreSQL Server (19)	12.7%	0.4%	18.3 MB/...	0 Mbps

测试结果显示用户10，事务数10000下postgresql的cpu占比最高达到17.7，其余时候稳定在13%左右，而内存消耗大约维持在0.4%左右

- 用户30，事务数10000 (Postgresql)

postgres.exe	0.06	5,052 K	39,740 K	10652
postgres.exe		4,984 K	39,664 K	10780
postgres.exe		4,916 K	39,616 K	1512
postgres.exe	0.06	5,020 K	39,700 K	23664
postgres.exe	< 0.01	5,036 K	39,640 K	25424
postgres.exe	0.18	4,992 K	39,624 K	22096
postgres.exe	0.06	5,024 K	39,632 K	21756
postgres.exe	0.06	4,984 K	39,700 K	10512
postgres.exe	0.12	4,988 K	39,720 K	14448
postgres.exe	0.06	5,016 K	39,696 K	24564
postgres.exe	0.12	5,016 K	39,608 K	1968
postgres.exe	0.18	5,016 K	39,672 K	12828
postgres.exe	0.37	4,992 K	39,748 K	21432
postgres.exe	< 0.01	4,988 K	39,644 K	25496
postgres.exe	0.18	4,980 K	39,628 K	344
postgres.exe	0.24	4,996 K	39,748 K	24880
postgres.exe	0.30	5,032 K	39,736 K	22556
postgres.exe	0.06	5,016 K	39,748 K	24372
postgres.exe	< 0.01	4,996 K	39,652 K	6700
postgres.exe	0.12	4,984 K	39,732 K	25152
postgres.exe	0.18	5,000 K	39,628 K	20952
postgres.exe	< 0.01	4,988 K	39,688 K	18348
postgres.exe	0.24	5,016 K	39,648 K	24188
postgres.exe	0.12	5,008 K	39,744 K	21352
postgres.exe	0.06	5,016 K	39,696 K	21300
postgres.exe	0.06	4,996 K	39,700 K	14788
postgres.exe	0.06	5,032 K	39,612 K	23800
postgres.exe	< 0.01	5,048 K	39,804 K	21132



> PostgreSQL Server (38)	39.8%	1.0%	20.3 MB/...	0 Mbps
> PostgreSQL Server (38)	26.7%	1.1%	10.1 MB/...	0 Mbps
> PostgreSQL Server (38)	30.1%	1.1%	19.1 MB/...	0 Mbps

测试结果显示用户30，事务数10000下postgresql的cpu占比最高达到39.8，其余时候稳定在30%左右，而内存消耗大约维持在1.1%左右

- 用户10，事务数10000 (openGuass)





在一开始测试的时候，发现总cpu占用率竟然高达了80%，这是比较令人费解的，后来发现80%中有相当一部分是由VmmemWSL占比，高达60%！，后来发现VmmemWSL是Windows 管理 **WSL 2** 虚拟机的关键进程。WSL 2 使用完整的虚拟化技术运行 Linux 内核，并在一个轻量级的虚拟机中提供 Linux 环境。vmmemWSL 是用于管理和分配该虚拟机的内存和 CPU 的服

务进程，与openGuass本身的使用并没有关系。去除掉这一使用，观测 datagrip的cpu的占比应为openGuass服务器的资源消耗.

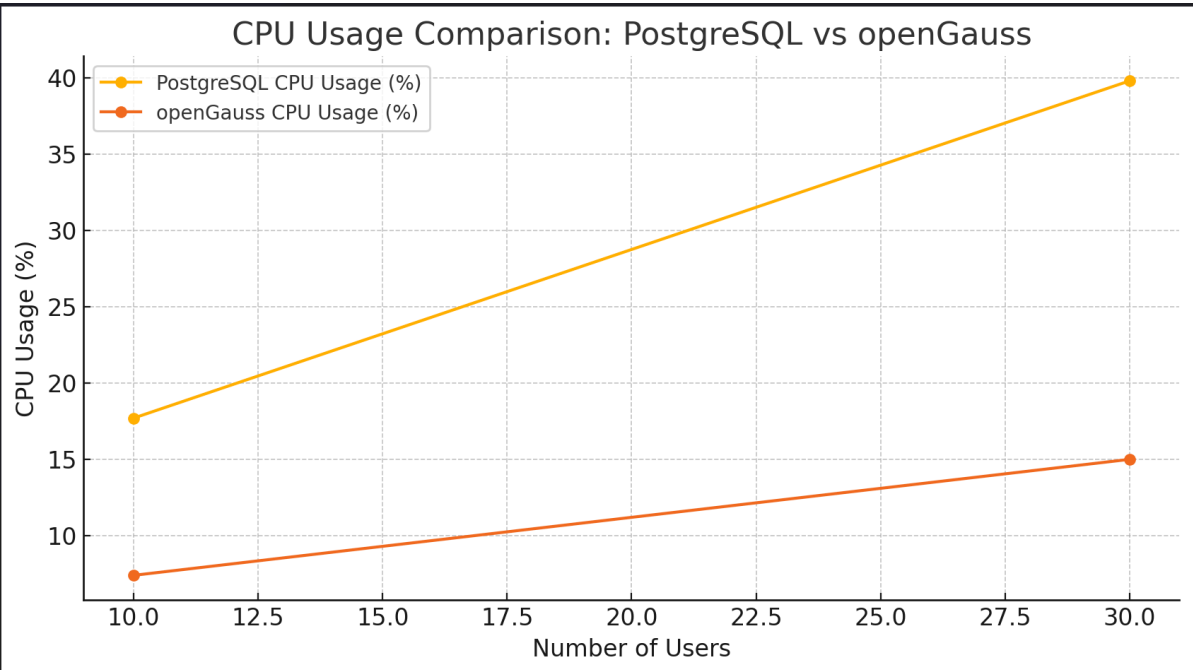
>  DataGrip	2.6%	16.5%	9.3 MB/秒	0 Mbps
>  DataGrip	7.4%	16.4%	9.9 MB/秒	0 Mbps

测试结果显示用户30，事务数10000下openGuass的cpu占比最高达到7.4，其余时候稳定在3%左右，而内存消耗大约维持在16.4%左右

- 用户30，事务数10000（openGuass）

Process	CPU	Private Bytes	Working Set	PID	Description	Company Name	
LocationNotifica...		1,968 K	1,140 K	13480	位置通知	Microsoft Corporation	
amdow.exe		2,196 K	1,780 K	17040	Radeon Settings: Deskt...	Advanced Micro Device...	
com.docker.backe...		37,872 K	20,140 K	6868	Docker Desktop Backend	Docker Inc.	
com.docker.back...	1.84	117,780 K	101,924 K	12792	Docker Desktop Backend	Docker Inc.	
com.docker.de...		15,856 K	1,124 K	3180			
com.docker.bu...		29,632 K	18,792 K	5896			
Docker Desкто...	0.35	132,032 K	83,572 K	5444	Docker Desktop	Docker Inc.	
Docker Desкто...	< 0.01	149,616 K	90,688 K	14840	Docker Desktop	Docker Inc.	
Docker Desкто...		16,628 K	7,328 K	14244	Docker Desktop	Docker Inc.	
Docker Desкто...	< 0.01	91,616 K	123,440 K	2724	Docker Desktop	Docker Inc.	
>  DataGrip				11.0%	16.1%	8.8 MB/秒	0 Mbps
>  DataGrip				8.5%	16.5%	10.7 MB/...	0 Mbps
>  DataGrip				15.0%	16.9%	10.5 MB/...	0 Mbps
>  DataGrip				9.8%	16.9%	7.1 MB/秒	0 Mbps

2.3结论



PostgreSQL 在内存使用方面更轻量，但随着用户增加，CPU 负载增长较快。openGauss 的内存使用较高，但在 CPU 使用上更为高效，特别是在高并发场景下具有明显优势。

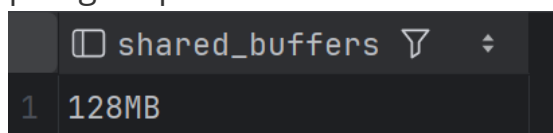
思考：为啥openGuass的内存占比会高于postgresql呢？我通过多次测试仍然是openGuass的内存占用高，按理来说openGuass基于postgresql开发，内存优化应该会更好。后来我发现了**可能**的原因。

当输入以下sql语句时

```
1 | show shared_buffers;
```

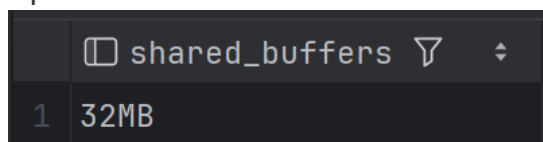
shared_buffers是数据库服务器为共享缓冲区分配的内存大小，更大的shared_buffers 会使 PostgreSQL 缓存更多的数据，减少磁盘访问，减少内存压力提高查询性能。于是我分别在postgresql和openGuass中输入语句查看他们的shared_buffers的值，结果如下

postgresql：



```
shared_buffers
1 | 128MB
```

openGuass：



```
shared_buffers
1 | 32MB
```

结果发现postgresql的服务器共享缓冲区分配的内存大小是openGuass的4倍，这意味这很有可能由于openGuass的服务器所分配的内存大小较小，使得查询过程中磁盘访问较多，增大内存压力，导致openGuass的内存占用结果比postgrsql高！

3.安全性

对数据库进行**安全性验证**是确保数据库在面对潜在安全威胁时能够正常工作的重要环节。因此我从这几个方面来进行验证测试

3.1验证访问控制（用户权限管理）

分别对两个数据库进行创建用户test_user,对用户赋予一定权限，随后测试是否能够执行权限之内和之外的事情

postgresql:

```
1 CREATE USER test_user WITH PASSWORD '12345';
2 CREATE ROLE read_only_role;
3 GRANT CONNECT ON DATABASE project3 TO read_only_role;
4 GRANT SELECT ON ALL TABLES IN SCHEMA public TO
  read_only_role;
5 GRANT read_only_role TO test_user;
```

创建用户test_user只有只读权限，然后切换用户，进行插入数据的操作

```
1 INSERT INTO test_data VALUES (1, 'test');
```

```
project3.public> INSERT INTO test_data VALUES (1, 'test')
[2024-12-20 21:01:44] [42501] 错误： 对表 test_data 权限不够
```

openGuass:

```
1 CREATE USER test_user WITH PASSWORD '12345678abc@';
2 CREATE ROLE read_only_role;
3 GRANT CONNECT ON DATABASE project3 TO read_only_role;
4 GRANT SELECT ON ALL TABLES IN SCHEMA public TO
  read_only_role;
5 GRANT read_only_role TO test_user;
```

创建用户test_user只有只读权限，然后切换用户，进行插入数据的操作

```
1 INSERT INTO test_data VALUES (1, 'test');
```

结果：

```
project3.public> INSERT INTO test_data VALUES (1, 'test')
[2024-12-20 21:17:01] [42501] ERROR: permission denied for relation test_data
[2024-12-20 21:17:01] 详细： N/A
```

结论：两种数据库用户权限管理和访问控制机制均为有效。

3.2 加密性能

对于postgresql的数据安全我们较为了解，这里主要关注openGuass是否能做到数据安全，好在openGuass是开源的，我们可以分析其源码是否做了一些关于数据安全性的保护。

根据源码分析，在**builtins.h**的这个文件下，openGuass提供了加密解密函数的入口函数，对用户导入导出的数据进行加密保护以及解密的功能

```
1 extern Datum aes_encrypt(PG_FUNCTION_ARGS);
2 extern Datum aes_decrypt(PG_FUNCTION_ARGS);
```

而**cipherfn.cpp**这个文件用于加密和解密功能的核心实现，主要用于对敏感数据（如密码、密钥、数据库配置选项等）进行加密保护。文件中包含 AES-128 和 SM4 两种加密算法的实现，以下是源码中的加密解密接口函数

```
1 gs_encrypt_aes128_function //加密函数
2 gs_encrypt_sm4_function
```

```
1 gs_decrypt_aes128_function//解密函数
2 gs_decrypt_sm4_function
```

在进入加密接口的函数后，加密过程会继续调用调用内部函数

gs_encrypt_aes_speed，开始加密的核心处理：明文数据会被分解，引入随机盐值以增强安全性，生成相应的密文数据结构。

了解了openGuass具有加密解密功能之后，现在我们来测试Postgresql和openGuass的加密性能所需时间部分，我们使用的数据集仍然是由python脚本生成的test_data

```
1 CREATE TABLE device_data (
2     device_id SERIAL PRIMARY KEY,      -- 自动增长的设备ID
3     device_name TEXT NOT NULL,         -- 设备名称
4     device_type TEXT NOT NULL,         -- 设备类型
5     storage_location TEXT NOT NULL,    -- 存储位置
6     manufacture_date DATE NOT NULL    -- 生产日期
7 );
8
9 -- 插入模拟数据
```

```

10 INSERT INTO device_data (device_name, device_type,
    storage_location, manufacture_date)
11 SELECT
12     'Device_' || i,
13     CASE WHEN i % 3 = 0 THEN 'Type_A'
14         WHEN i % 3 = 1 THEN 'Type_B'
15         ELSE 'Type_C' END,
16     'Location_' || (i % 5),
17     CURRENT_DATE - (i % 100) * INTERVAL '1 day'
18 FROM generate_series(1, 100) AS i;

```

```

1  -- PostgreSQL 加密
2  CREATE EXTENSION IF NOT EXISTS pgcrypto;
3  UPDATE device_data
4  SET
5      device_name = pgp_sym_encrypt(device_name,
    'Encrypt#100'),
6      device_type = pgp_sym_encrypt(device_type,
    'Encrypt#100'),
7      storage_location = pgp_sym_encrypt(storage_location,
    'Encrypt#100');
8
9  -- OpenGauss 加密
10 UPDATE device_data
11 SET
12     device_name =gs_encrypt_aes128(device_name,
    'Encrypt#100'),--使用了gs_encrypt_aes128
13     device_type = gs_encrypt_aes128(device_type,
    'Encrypt#100'),
14     storage_location = gs_encrypt_aes128(storage_location,
    'Encrypt#100');

```

结果: [2024-12-21 16:32:29] 100 rows affected in 274 ms

[2024-12-21 16:35:35] 100 rows affected in 30 ms

postgresql用时 **274ms**, openGuass 用时 **30ms**,OpenGauss加密的效率远高于PostgreSQL**

3.3加密数据的安全性测试

(1) 读取加密数据

openGuass:

	device_id ▾	device_name ▾	device_type ▾	st
1	1	LMIEzVFZvoUuwRzv/woodE2h87sD8E8toj63g...	LMIEzVFZvoUuwRzv/woodPcXQfh5ARVvYXEWN...	LMIEz
2	2	LMIEzVFZvoUuwRzv/woodD++sLLAvx8GIwdi4...	LMIEzVFZvoUuwRzv/wood6UrcJQ++Xza6LSxw...	LMIEz
3	3	LMIEzVFZvoUuwRzv/woodJcSRv2f0TGNY3t65...	LMIEzVFZvoUuwRzv/woodMfoyf6iqJ6HpK197...	LMIEz
4	4	LMIEzVFZvoUuwRzv/woodKxYUZ0+qjf1enWfV...	LMIEzVFZvoUuwRzv/woodL3kNmcTbf+70gS6J...	LMIEz
5	5	LMIEzVFZvoUuwRzv/woodP+Y/lhdbxpjzkQEb...	LMIEzVFZvoUuwRzv/wood6EqF23AXh5BHDH5F...	LMIEz
6	6	LMIEzVFZvoUuwRzv/woodHzzY5xVpdHWGouZD...	LMIEzVFZvoUuwRzv/woodIV8MJ7+g0aZdD3L7...	LMIEz
7	7	LMIEzVFZvoUuwRzv/wood6EIrSDFRpVqZxjFu...	LMIEzVFZvoUuwRzv/woodIgZ1iqjZY/MTBzkY...	LMIEz
8	8	LMIEzVFZvoUuwRzv/woodLWiQWLJvdayQAR6c...	LMIEzVFZvoUuwRzv/woodB0uZTJIUDm6Hw1JN...	LMIEz
9	9	LMIEzVFZvoUuwRzv/woodKDLN87X10oioukDH...	LMIEzVFZvoUuwRzv/woodCr94HdMxHHfWLGy0...	LMIEz
10	10	LMIEzVFZvoUuwRzv/woodDH04wUSnDnDdirCa...	LMIEzVFZvoUuwRzv/wood01/kih59vIwyfzC...	LMIEz
11	11	LMIEzVFZvoUuwRzv/woodH82gmX0fS+pbk0D6...	LMIEzVFZvoUuwRzv/woodIqw9aI5gCVUR5sT1...	LMIEz

postgresql:

	device_id ▾	device_name ▾	device_type ▾	st
1	1	LMIEzVFZvoUuwRzv/woodE2h87sD8E8toj63g...	LMIEzVFZvoUuwRzv/woodPcXQfh5ARVvYXEWN...	LMIEz
2	2	LMIEzVFZvoUuwRzv/woodD++sLLAvx8GIwdi4...	LMIEzVFZvoUuwRzv/wood6UrcJQ++Xza6LSxw...	LMIEz
3	3	LMIEzVFZvoUuwRzv/woodJcSRv2f0TGNY3t65...	LMIEzVFZvoUuwRzv/woodMfoyf6iqJ6HpK197...	LMIEz
4	4	LMIEzVFZvoUuwRzv/woodKxYUZ0+qjf1enWfV...	LMIEzVFZvoUuwRzv/woodL3kNmcTbf+70gS6J...	LMIEz
5	5	LMIEzVFZvoUuwRzv/woodP+Y/lhdbxpjzkQEb...	LMIEzVFZvoUuwRzv/wood6EqF23AXh5BHDH5F...	LMIEz
6	6	LMIEzVFZvoUuwRzv/woodHzzY5xVpdHWGouZD...	LMIEzVFZvoUuwRzv/woodIV8MJ7+g0aZdD3L7...	LMIEz
7	7	LMIEzVFZvoUuwRzv/wood6EIrSDFRpVqZxjFu...	LMIEzVFZvoUuwRzv/woodIgZ1iqjZY/MTBzkY...	LMIEz
8	8	LMIEzVFZvoUuwRzv/woodLWiQWLJvdayQAR6c...	LMIEzVFZvoUuwRzv/woodB0uZTJIUDm6Hw1JN...	LMIEz
9	9	LMIEzVFZvoUuwRzv/woodKDLN87X10oioukDH...	LMIEzVFZvoUuwRzv/woodCr94HdMxHHfWLGy0...	LMIEz
10	10	LMIEzVFZvoUuwRzv/woodDH04wUSnDnDdirCa...	LMIEzVFZvoUuwRzv/wood01/kih59vIwyfzC...	LMIEz
11	11	LMIEzVFZvoUuwRzv/woodH82gmX0fS+pbk0D6...	LMIEzVFZvoUuwRzv/woodIqw9aI5gCVUR5sT1...	LMIEz

两个数据库均已经成功加密

(2)破解加密数据

对于加密数据的破解攻击，有很多种方式，包括暴力破解，字典攻击，重复数据块攻击，侧信道攻击等等，不过鉴于方法的可实现性和可操作性，这里只选择暴力破解攻击来尝试破解加密数据（已知加密算法）

以postgresql为例，进行暴力破解

```
1 CREATE TEMP TABLE possible_keys (  
2     key TEXT  
3 );  
4  
5 INSERT INTO possible_keys (key) VALUES  
6     ('TestKey100'),  
7     ('Encrypt#123'),  
8     ('AnotherKey456'),
```

```

9  -- 遍历加密数据尝试解密
10 DO $$
11 DECLARE
12     encrypted_device_name TEXT;
13     decrypted_device_name TEXT;
14     current_key TEXT;
15 BEGIN
16 FOR encrypted_device_name IN SELECT device_name FROM
device_data LOOP
17     FOR current_key IN SELECT key FROM possible_keys LOOP
18         BEGIN
19             decrypted_device_name :=
pgp_sym_decrypt(encrypted_device_name, current_key);
20 RAISE NOTICE '尝试密钥: %, 解密结果: %', current_key,
COALESCE(decrypted_device_name, '解密失败');
21         EXCEPTION
22             WHEN others THEN
23
24                 RAISE NOTICE '密钥 % 解密失败, 错误: %',
current_key, SQLERRM;
25         END;
26     END LOOP;
27 END LOOP;
28 END $$;

```

结果如下:

postgresql:

```

尝试密钥: WrongKey789, 解密结果: \xc30d04070302989bb7e25fbd50e467d2c00b
[2024-12-21 16:59:36] completed in 750 ms

```

openGuass:

!

```

密钥 WrongKey789 解密失败, 错误: decrypt the cipher text failed!
[2024-12-21 16:57:59] completed in 2 s 442 ms

```

结论: openGuass的解密时间比postgresql的时间要长, 认为openGuass的加密数据比postgresql更加安全

4.拓展性

拓展性 (Scalability) 是衡量一个系统在面对不断增长的工作负载时，能否通过增加资源（例如硬件或节点）来维持或提升性能的能力。为此我们可以创建分区表，通过增加数据量对比两个数据库的查询时间效率。

首先创建分区表

```
1 CREATE TABLE orders (--分区表
2     order_id SERIAL,
3     customer_id INT NOT NULL,
4     order_date DATE NOT NULL,
5     order_amount NUMERIC NOT NULL,
6     region TEXT,
7     PRIMARY KEY (order_id, order_date)
8 ) PARTITION BY RANGE (order_date);
9
10 CREATE TABLE orders_2023 PARTITION OF orders--具体分区
11 FOR VALUES FROM ('2023-01-01') TO ('2024-01-01');
12
13 CREATE TABLE orders_2024 PARTITION OF orders
14 FOR VALUES FROM ('2024-01-01') TO ('2025-01-01');
15
16
```

```
1 INSERT INTO orders (customer_id, order_date, order_amount,
2     region)
3 SELECT i, '2023-06-01'::DATE + (i % 100), i * 100,
4     'Region_' || (i % 5)
5 FROM generate_series(1, 1000000) i;
6
7 CREATE TABLE orders_2025 PARTITION OF orders--创建新分区
8 FOR VALUES FROM ('2025-01-01') TO ('2026-01-01');
9
10 SELECT * FROM orders WHERE order_date BETWEEN '2023-06-01'
11 AND '2023-06-30';--执行查询
```

结果：比较两个数据库的查询时间的效率

postgresql:

```
lic> SELECT * FROM orders WHERE order_date BETWEEN '2023-06-01' AND '2023-06-30'
20:17:04] 500 rows retrieved starting from 1 in 160 ms (execution: 7 ms, fetching: 153 ms)
```

openGuass:

```
SELECT * FROM orders WHERE order_date BETWEEN '2023-06-01' AND '2023-06-30'  
[28:29] 500 rows retrieved starting from 1 in 86 ms (execution: 6 ms, fetching: 80 ms)
```

对比时间，openGuass在分区拓展上的查询性能比postgresql的要好

5.测试结果汇总

- openGuass比postgresql具有更高的吞吐量（tps）,高并发场景下表现良好，能够有效地管理多个并发用户或线程的事务操作,适合需要支持大量并发用户访问的系统，例如社交媒体、在线视频流平台和实时分析系统.
- PostgreSQL** 在内存使用方面更轻量，但随着用户增加，CPU 负载增长较快。**openGauss** 的内存使用较高，但在 CPU 使用上更为高效，特别是在高并发场景下具有明显优势
- postgresql的服务器共享缓冲区分配的内存大小是openGuass的4倍，很有可能是导致openGuass的内存占用结果比postgrsqli高的原因
- 对于安全性方面，OpenGuass所采用的AES-128 和 SM4 两种加密算法相较于postgresql有较高的数据加密性能，安全性更高
- 分区可拓展性方面，openGuass的可拓展能力更强