

# Project 3: Benchmarking openGauss Against PostgreSQL: A Comparative Analysis

Li Yuxuan

## 1. Introduction

Aimed at comparing the efficiency of openGauss and PostgreSQL, we designed **three** parts of the experiments in the project. The first part contains the basic operations of databases including **large amount** of **insert**, **select**, **update** and **subquery** using the combination of these operations. The second part concentrates on the optimization of **index** on the two databases. The third part combining the introduction of Gaussdb in its official documents, tests its announced product strengths like **concurrent connection** and **SQL Bypass** for simple statements.

## 2. Environment Preparation

### Hardware:

Chip	Apple M4
Memory	16 GB
Docker Version	27.4.1
Colima Version	0.8.1
DBMS Version	openGauss 3.1.1 PostgreSQL 14-22.04_beta
Libpqxx Version	7.9.3
C++	C++ 20
Compiler	AppleClang 16.0.0.16000026

### Process:

1. Deploy containers of openGauss and PostgreSQL on **colima** using **docker**.

```
docker run --name project3-opengauss --privileged=true \  
-d -e GS_PASSWORD=Abc@1234 \  
-v /User/xxx/var/lib/opengauss -u root \  
-p 15432:5432 \  
docker.1panel.live/enmotech/opengauss:3.1.1
```

```
docker run -d \  
--name project3-postgres \  
-e POSTGRES_USER=postgres \  

```

```
-e POSTGRES_PASSWORD=123456 \
-e POSTGRES_DB=postgres \
-e PGDATA=/var/lib/postgresql/data/pgdata \
-p 25432:5432 \
-v /User/xxx/var/lib/postgresql/data \
docker.1panel.live/ubuntu/postgres:14-22.04_beta
```

The virtual machine has 4 CPUs and 16GiB of memory.

```
mio@MacBook-Pro-Mio ~ % colima list
```

PROFILE	STATUS	ARCH	CPU	MEMORY	DISK	RUNTIME	ADDRESS
default	Running	aarch64	4	16GiB	100GiB	docker	

2. Create a new database **pro3** in both openGauss and PostgreSQL as the experimental database.

3. Use the statement "show all" to get **system settings** of openGauss and PostgreSQL and export as csv files. Then use "**STL map**" to get the settings with the same names by a C++ program.

```
1 map<string,string>se;
2 string s;
3 ifstream fin,fni;
4 ofstream fout;
5 fin.open("Result_15.csv");//Setting of openGauss
6 fni.open("Result_10.csv");//Setting of PostgreSQL
7 fout.open("1.out");
8 while(getline(fin,s))
9 {
10     string ss;
11     for(int i=0;i<s.length();i++)
12     {
13         if(s[i]==' '){break;}
14         ss.push_back(s[i]);
15     }
16     assert(se.find(ss)==se.end());
17     se[ss]=s;
18 }
19 while(getline(fni,s))
20 {
21     string ss;
22     for(int i=0;i<s.length();i++)
23     {
24         if(s[i]==' '){break;}
25         ss.push_back(s[i]);
26     }
27     map<string,string>::iterator ite=se.find(ss);
```

---

```

28     if(ite!=se.end())
29     {
30         fout<<ite->second+'\n'+s+'\n'+'\n';
31     }
32 }

```

---

4. Find the different settings, make them a same value.

5. For the following concurrent connection experiments, set both the max\_connections to 256, set sysadmin\_reserved\_connections for openGauss and superuser\_reserved\_connections for PostgreSQL to 3.

## Primary Experiment

Cut down all connections to the databases. Use "docker stats" to query the CPU and Memory Usage of the databases when they are in idle state.

NAME	CPU %	MEM USAGE / LIMIT	MEM %
project3-opengauss	2.90%	428.2MiB / 15.59GiB	2.68%
project3-postgres	0.00%	54.97MiB / 15.59GiB	0.34%

Easy to find the CPU and Memory Usage of openGauss in idle state is much greater.

## 3. Basic operations

Create a new table **a** with one column id in both openGauss and PostgreSQL to complete the experiment in this part.

---

```

1 create table a(id int);

```

---

### (1) Insert

To avoid the influence of SQL Bypass, each time we have to insert two values at once.

a)

Use **uniform\_int\_distribution** function of C++ to generate **10<sup>6</sup>** distinct numbers. And then **shuffle** them to make them in random order.

---

```

1 mt19937 rd(random_device{}());
2 uniform_int_distribution<int>sc(0,2147483647);
3 vector<int>vec;
4 set<int>se;
5 freopen("1.out","w",stdout);

```

---

```
6 while(se.size()<1000000)
7 {
8     int nw=sc(rd);
9     if(se.find(nw)==se.end())se.insert(nw);
10 }
11 for(set<int>::iterator ite=se.begin();ite!=se.end();ite++)
12 {
13     vec.push_back(*ite);
14 }
15 shuffle(vec.begin(),vec.end(),rd);
16 for(vector<int>::iterator ite=vec.begin();ite!=vec.end();ite++)
17 {
18     printf("%d ",*ite);
19 }
```

---

b)

Use a C++ code to read the generated numbers (in.txt) and insert them into the databases two by two.

Use "explain analyse" statement to get the runtime and sum up the results. At the same time, use "docker stats" to get the CPU and Memory Usage.

Code:

Get runtime from "explain analyse" result (unit: ms).

---

```
1 inline double get_num(const char* s)
2 {
3     double res=0,base=1;
4     bool on_float=false;
5     while(*s)
6     {
7         if(*s=='.')on_float=true;
8         else if(*s>='0'&&*s<='9')
9         {
10             res=res*10+(*s-'0');
11             if(on_float)base*=10;
12         }
13         s++;
14     }
15     return res/base;
16 }
17 inline double get_time(const result &r)
18 {
19     return get_num(r.back().front().c_str());
20 }
```

---

Insert data into databases and sum up the runtime.

---

```

1 vector<int>vec;
2 int in;
3 freopen("in.txt","r",stdin);
4 for(int i=1;i<=1000000;i++)
5 {
6     scanf("%d",&in);
7     vec.push_back(in);
8 }
9 try
10 {
11     connection C1("dbname = pro3 user = gaussdb password = Abc@1234 host =
        localhost port = 15432");
12     double tot1=0,tot2=0;
13     if(C1.is_open())
14     {
15         for(int i=0;i<1000000;i+=2)
16         {
17             nontransaction W(C1);
18             tot1+=get_time(W.exec("explain analyse insert into a values ("+
                to_string(vec[i])+"),("+to_string(vec[i+1])+");"));
19         }
20     }
21     connection C2("dbname = pro3 user = postgres password = 123456 host =
        localhost port = 25432");
22     if(C2.is_open())
23     {
24         for(int i=0;i<1000000;i+=2)
25         {
26             nontransaction W(C2);
27             tot2+=get_time(W.exec("explain analyse insert into a values ("+
                to_string(vec[i])+"),("+to_string(vec[i+1])+");"));
28         }
29     }
30     printf("Runtime of openGauss : %f\nRuntime of PostgreSQL : %f\n",tot1,tot2);
31 }
32 catch(const std::exception& e)
33 {
34     std::cerr<<e.what()<<'\\n';
35 }

```

---

c)

Result:

For insert, the **runtime**, **CPU** and **Memory** usage of openGauss are all much more than PostgreSQL.

Result of code (unit: ms)

```

● mio@MacBook-Pro-Mio insert_test % ./gauss
Runtime of openGauss : 16500.300000
Runtime of PostgreSQL : 1574.237000

```

Result of docker stats

NAME	CPU %	MEM USAGE / LIMIT	MEM %	NET I/O	BLOCK I/O	PIDS
project3-opengauss	36.25%	564MiB / 15.59GiB	3.53%	36.4MB / 101MB	4.12MB / 2.39GB	41
project3-postgres	16.07%	80.84MiB / 15.59GiB	0.51%	39.4MB / 119MB	946kB / 2.58GB	9

## (2) Select

Following the experiment of insert, we test the efficiency of selecting data in a range.

a)

Generate  $10^6$  intervals.

```

1 mt19937 rd(random_device{}());
2 uniform_int_distribution<int>sc(0,2147483647);
3 for(int i=1;i<=1000000;i++)
4 {
5     int l=sc(rd),r=sc(rd);
6     if(l>r)swap(l,r);
7     printf("%d %d\n",l,r);
8 }

```

b)

Read the generated intervals (in2.txt) and select results with id in the range. (Because the running time is too long, we only use the first **2500** intervals.)

Use "explain analyse" statement to get the runtime and sum up the results. At the same time, use "docker stats" to get the CPU and Memory Usage.

```

1 for(int i=1;i<=5000;i++)
2 {
3     scanf("%d",&in);
4     vec.push_back(in);
5 }
6 try

```

---

```

7 {
8     connection C1("dbname = pro3 user = gaussdb password = Abc@1234 host =
        localhost port = 15432");
9     double tot1=0,tot2=0;
10    if(C1.is_open())
11    {
12        for(int i=0;i<5000;i+=2)
13        {
14            nontransaction W(C1);
15            tot1+=get_time(W.exec("explain analyse select * from a where id between
                "+to_string(vec[i])+" and "+to_string(vec[i+1])+";"));
16        }
17    }
18    connection C2("dbname = pro3 user = postgres password = 123456 host = localhost
        port = 25432");
19    if(C2.is_open())
20    {
21        for(int i=0;i<5000;i+=2)
22        {
23            nontransaction W(C2);
24            tot2+=get_time(W.exec("explain analyse select * from a where id between
                "+to_string(vec[i])+" and "+to_string(vec[i+1])+";"));
25        }
26    }
27    printf("Runtime of openGauss : %f ms\nRuntime of PostgreSQL : %f ms\n",tot1,
        tot2);
28 }
29 catch(const std::exception& e)
30 {
31     std::cerr<<e.what()<<'\\n';
32 }

```

---

c)

Result:

For select, the runtime of openGauss is nearly twice of PostgreSQL and the Memory usage is much more as well. But the CPU usage of openGauss is slightly less than PostgreSQL.

Result of code

```

● mio@MacBook-Pro-Mio select_test % ./gauss
Runtime of openGauss : 178340.756000 ms
Runtime of PostgreSQL : 84336.777000 ms

```

Result of docker stats

NAME	CPU %	MEM USAGE / LIMIT	MEM %	NET I/O	BLOCK I/O	PIDS
project3-opengauss	105.13%	677.3MiB / 15.59GiB	4.24%	727MB / 82.8GB	4.3MB / 4.55GB	40
project3-postgres	121.30%	94.77MiB / 15.59GiB	0.59%	67.2MB / 204MB	1.07MB / 4.5GB	10

### (3) Update

Similarly, we test the efficiency of updating data in a range.

a)

Use the intervals from experiment of select.

b)

Read the generated intervals (in2.txt), find rows with id in a range, and change the id to

-1. (Use the first 100 intervals.)

Use "explain analyse" statement to get the runtime and sum up the results. Then rollback.

At the same time, use "docker stats" to get the CPU and Memory Usage.

```

1 for(int i=1;i<=200;i++)
2 {
3     scanf("%d",&in);
4     vec.push_back(in);
5 }
6 try
7 {
8     connection C1("dbname = pro3 user = gaussdb password = Abc@1234 host =
        localhost port = 15432");
9     double tot1=0,tot2=0;
10    if(C1.is_open())
11    {
12        for(int i=0;i<200;i+=2)
13        {
14            transaction W(C1);
15            tot1+=get_time(W.exec("explain analyse update a set id = -1 where id
                between "+to_string(vec[i])+" and "+to_string(vec[i+1])+";"));
16            W.abort();

```



```

17     }
18 }
19 connection C2("dbname = pro3 user = postgres password = 123456 host = localhost
    port = 25432");
20 if(C2.is_open())
21 {
22     for(int i=0;i<200;i+=2)
23     {
24         transaction W(C2);
25         tot2+=get_time(W.exec("explain analyse update a set id = -1 where id
            between "+to_string(vec[i])+" and "+to_string(vec[i+1])+";"));
26         W.abort();
27     }
28 }
29 printf("Runtime of openGauss : %f ms\nRuntime of PostgreSQL : %f ms\n",tot1,
    tot2);
30 }
31 catch(const std::exception& e)
32 {
33     std::cerr<<e.what()<<'\n';
34 }

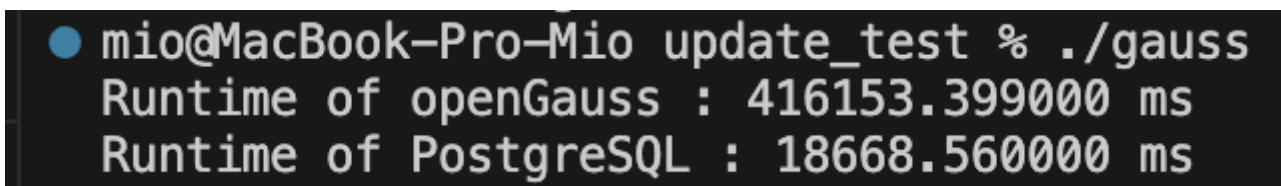
```

c)

Result:

For update, the **runtime**, the **CPU** and **Memory** usage of openGauss is much more than PostgreSQL.

Result of code



```

● mio@MacBook-Pro-Mio update_test % ./gauss
Runtime of openGauss : 416153.399000 ms
Runtime of PostgreSQL : 18668.560000 ms

```

Result of docker stats

NAME	CPU %	MEM USAGE / LIMIT	MEM %	NET I/O	BLOCK I/O	PIDS
project3-opengauss	119.83%	694.6MiB / 15.59GiB	4.35%	729MB / 82.8GB	4.45MB / 6.39GB	40
project3-postgres	98.78%	237.1MiB / 15.59GiB	1.49%	67.4MB / 204MB	64.8MB / 11.5GB	11

#### (4) Order By

In addition to selecting, we test the efficiency of selecting data in a range and outputting them in order then by "order by".

a)

Use the intervals from experiment of select.

b)

Read the generated intervals (in2.txt) and select results with id in the range of the intervals.(Use the first 2000 intervals.)

---

```
1 explain analyse select * from a where id between $1 and $2 order by id;
```

---

Get the runtime and sum up the results. And at the same time, use "docker stats" to get the CPU and Memory Usage.

---

```
1 for(int i=1;i<=4000;i++)
2 {
3     scanf("%d",&in);
4     vec.push_back(in);
5 }
6 try
7 {
8     connection C1("dbname = pro3 user = gaussdb password = Abc@1234 host =
        localhost port = 15432");
9     double tot1=0,tot2=0;
10    if(C1.is_open())
11    {
12        for(int i=0;i<4000;i+=2)
13        {
14            nontransaction W(C1);
15            tot1+=get_time(W.exec("explain analyse select * from a where id between
                "+to_string(vec[i])+" and "+to_string(vec[i+1])+" order by id;"));
16        }
17    }
18    connection C2("dbname = pro3 user = postgres password = 123456 host = localhost
        port = 25432");
19    if(C2.is_open())
20    {
21        for(int i=0;i<4000;i+=2)
22        {
23            nontransaction W(C2);
24            tot2+=get_time(W.exec("explain analyse select * from a where id between
                "+to_string(vec[i])+" and "+to_string(vec[i+1])+" order by id;"));
25        }
26    }
27    printf("Runtime of openGauss : %f ms\nRuntime of PostgreSQL : %f ms\n",tot1,
        tot2);
```

```

28 }
29 catch(const std::exception& e)
30 {
31     std::cerr<<e.what()<<'\n';
32 }

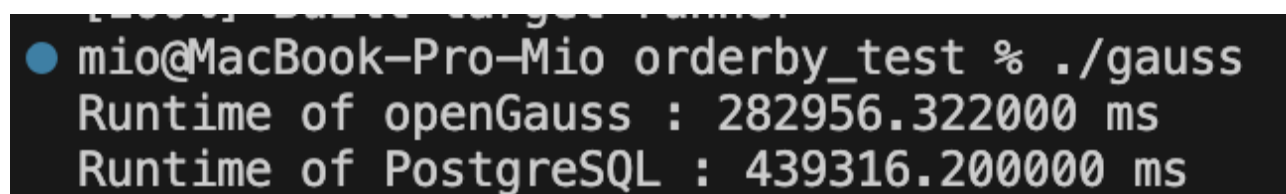
```

c)

Result:

For order by, the **runtime** of openGauss and **CPU** usage is less than PostgreSQL. But the **Memory** usage is more.

Result of code



```

● mio@MacBook-Pro-Mio orderby_test % ./gauss
Runtime of openGauss : 282956.322000 ms
Runtime of PostgreSQL : 439316.200000 ms

```

Result of docker stats

NAME	CPU %	MEM USAGE / LIMIT	MEM %	NET I/O	BLOCK I/O	PIDS
project3-opengauss	105.51%	791.7MiB / 15.59GiB	4.96%	729MB / 82.8GB	17.2MB / 20.8GB	41
project3-postgres	146.60%	317.5MiB / 15.59GiB	1.99%	67.8MB / 207MB	64.8MB / 14.4GB	9

## (5) Subquery

In the last of the part, we test the efficiency of **complex query** and take "updating data in a range to the minimum value in another range" as an example.

a)

Use the intervals from experiment of select.

b)

Read the generated intervals (in2.txt), change the rows with id in one range to the minimum id of rows with id in another range. (**500** queries.)(openGauss shows great weakness in update, so set the changed range to a small range  $[x, x + 10]$ ).

```

1 explain update a set id =
2 (
3     select min(id) from a where id between $1 and $2
4 )
5 where id between $3 and $4;

```

Get the runtime and sum up the results. And at the same time, use "docker stats" to get the CPU and Memory Usage. Then **rollback**.

---

```

1 for(int i=1;i<=100;i++)
2 {
3     scanf("%d",&in);
4     vec.push_back(in);
5 }
6 try
7 {
8     connection C1("dbname = pro3 user = gaussdb password = Abc@1234 host =
        localhost port = 15432");
9     double tot1=0,tot2=0;
10    if(C1.is_open())
11    {
12        for(int i=1;i<=100;i++)
13        {
14            transaction W(C1);
15            tot1+=get_time(W.exec("explain analyse update a set id = -1 where id
                between "+to_string(vec[i])+" and "+to_string(vec[i+1])+";"));
16            W.abort();
17        }
18    }
19    connection C2("dbname = pro3 user = postgres password = 123456 host = localhost
        port = 25432");
20    if(C2.is_open())
21    {
22        for(int i=1;i<=100;i++)
23        {
24            transaction W(C2);
25            tot2+=get_time(W.exec("explain analyse update a set id = -1 where id
                between "+to_string(vec[i])+" and "+to_string(vec[i+1])+";"));
26            W.abort();
27        }
28    }
29    printf("Runtime of openGauss : %f ms\nRuntime of PostgreSQL : %f ms\n",tot1,
        tot2);
30 }
31 catch(const std::exception& e)
32 {
33     std::cerr<<e.what()<<'\n';
34 }

```

---

c)

Result:

For subquery, the **runtime** of openGauss, **Memory** and **CPU** usage of openGauss is all more than PostgreSQL.

Result of code

```
mio@MacBook-Pro-Mio subquery_test % ./gauss
Runtime of openGauss : 147532.315000 ms
Runtime of PostgreSQL : 106114.500000 ms
```

Result of docker stats

NAME	CPU %	MEM USAGE / LIMIT	MEM %	NET I/O	BLOCK I/O	PIDS
project3-opengauss	111.08%	754.8MiB / 15.59GiB	4.73%	730MB / 82.8GB	17.3MB / 43.3GB	41
project3-postgres	100.24%	244.4MiB / 15.59GiB	1.53%	67.9MB / 207MB	64.8MB / 14.5GB	10

## 4. Operations on a table with index

Create a new table **b** with one column **id** in both openGauss and PostgreSQL to complete the experiment in this part. Then create a **unique index** **id\_idx** on column **id**.

```
1 create table b(id int);
2 create unique index on b(id);
```

Use **'/d'** to get the description of the tables:

```
[mio@MacBook-Pro-Mio ~ % psql pro3 -p 25432 -U postgres -h localhost
Password for user postgres:
psql (14.14 (Homebrew), server 14.12 (Ubuntu 14.12-0ubuntu0.22.04.1))
Type "help" for help.

[pro3=# \d b
               Table "public.b"
  Column | Type   | Collation | Nullable | Default
-----+-----+-----+-----+-----
   id    | integer |           |          |
Indexes:
    "b_id_idx" UNIQUE, btree (id)

[pro3=# quit
[mio@MacBook-Pro-Mio ~ % psql pro3 -p 15432 -U gaussdb -h localhost
Password for user gaussdb:
psql (14.14 (Homebrew), server 9.2.4)
Type "help" for help.

[pro3=> \d b
               Table "public.b"
  Column | Type   | Collation | Nullable | Default
-----+-----+-----+-----+-----
   id    | integer |           |          |
Indexes:
    "b_id_idx" UNIQUE, btree (id) TABLESPACE pg_default
```

## (1) Insert

Similarly, to avoid the influence of SQL Bypass, each time we have to insert two values at once.

a)

Using generated data in the last part. ( $10^6$  numbers)

b)

Write a C++ code that reads the generated numbers (in.txt) and insert them into the databases **two by two**.

Use "explain analyse" statement to get the runtime and sum up the results. At the same time, use "docker stats" to get the CPU and Memory Usage.

Code is the same as last part.

c)

Result:

For insert, the **runtime**, **CPU** and **Memory** usage of openGauss are all much more than PostgreSQL.

Result of code (unit: ms)

```

● mio@MacBook-Pro-Mio insert_test % ./gauss
Runtime of openGauss : 24004.434000
Runtime of PostgreSQL : 3949.625000

```

Result of docker stats

NAME	CPU %	MEM USAGE / LIMIT	MEM %	NET I/O	BLOCK I/O	PIDS
project3-opengauss	42.42%	787.5MiB / 15.59GiB	4.93%	740MB / 82.9GB	24.2MB / 46.3GB	41
project3-postgres	15.04%	239.3MiB / 15.59GiB	1.50%	84.8MB / 259MB	76.6MB / 15.6GB	10

## (2) Select

Following the experiment of insert, we test the efficiency of selecting data in a range.

a)

Use the intervals in the last part.

b)

Read the generated intervals (in2.txt) and select results with id in the range. (Use the first 2500 intervals.)

Use "explain analyse" statement to get the runtime and sum up the results. At the same time, use "docker stats" to get the CPU and Memory Usage.

To avoid the traverse, we use "count(\*)" instead of \*.

---

```

1 for(int i=1;i<=5000;i++)
2 {
3     scanf("%d",&in);
4     vec.push_back(in);
5 }
6 try
7 {
8     connection C1("dbname = pro3 user = gaussdb password = Abc@1234 host =
        localhost port = 15432");
9     double tot1=0,tot2=0;
10    if(C1.is_open())
11    {
12        for(int i=0;i<5000;i+=2)
13        {
14            nontransaction W(C1);
15            tot1+=get_time(W.exec("explain analyse select count(*) from b where id
                between "+to_string(vec[i])+" and "+to_string(vec[i+1])+";"));
16        }
17    }
18    connection C2("dbname = pro3 user = postgres password = 123456 host = localhost
        port = 25432");
19    if(C2.is_open())
20    {
21        for(int i=0;i<5000;i+=2)
22        {
23            nontransaction W(C2);
24            tot2+=get_time(W.exec("explain analyse select count(*) from b where id
                between "+to_string(vec[i])+" and "+to_string(vec[i+1])+";"));
25        }
26    }
27    printf("Runtime of openGauss : %f ms\nRuntime of PostgreSQL : %f ms\n",tot1,
        tot2);
28 }
29 catch(const std::exception& e)
30 {
31     std::cerr<<e.what()<<'\n';
32 }

```

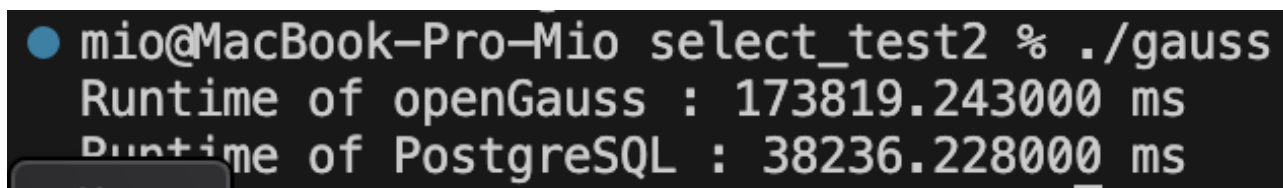
---

c)

Result:

For select, the **runtime** and the **Memory** usage of openGauss are much more than PostgreSQL. But the **CPU** usage of PostgreSQL is much more than openGauss.

Result of code



```

mio@MacBook-Pro-Mio select_test2 % ./gauss
Runtime of openGauss : 173819.243000 ms
Runtime of PostgreSQL : 38236.228000 ms

```

Result of docker stats

NAME	CPU %	MEM USAGE / LIMIT	MEM %	NET I/O	BLOCK I/O	PIDS
project3-opengauss	112.20%	940MiB / 15.59GiB	5.89%	797MB / 83GB	24.2MB / 52.8GB	40
project3-postgres	260.38%	241.1MiB / 15.59GiB	1.51%	136MB / 413MB	76.6MB / 18.9GB	12

### (3) Order By

In addition to selecting, we test the efficiency of selecting data in a range and outputting them in order then by "order by".

a)

Use the intervals from experiment of select in last part.

b)

Read the generated intervals (in2.txt) and select results with id in the range of the intervals.(Use the first **2000** intervals.)

---

```
1 explain analyse select * from a where id between $1 and $2 order by id;
```

---

Get the runtime and sum up the results. And at the same time, use "docker stats" to get the CPU and Memory Usage.

Code is the same as last part.

c)

Result:

For order by, the **runtime** of openGauss shows an strange increase compared to that without index. And PostgreSQL shows an better performance compared to that without index and is. Then they have a great difference in efficiency.

In other terms, the **CPU** and Memory usage of openGauss is greater than PostgreSQL.



Result of code

```

● mio@MacBook-Pro-Mio orderby_test % ./gauss
Runtime of openGauss : 449299.020000 ms
Runtime of PostgreSQL : 49334.694000 ms

```

Result of docker stats

NAME	CPU %	MEM USAGE / LIMIT	MEM %	NET I/O	BLOCK I/O	PIDS
project3-opengauss	120.42%	941MiB / 15.59GiB	5.90%	798MB / 83GB	24.2MB / 52.8GB	40
project3-postgres	99.03%	237MiB / 15.59GiB	1.48%	136MB / 416MB	77.7MB / 18.9GB	10

d)

Analysis:

By looking up the explanation of the process of the operation, we find openGauss:

	QUERY PLAN
1	Sort (cost=10118.65..10239.95 rows=48519 width=4)
2	Sort Key: id
3	-> Bitmap Heap Scan on b (cost=1189.57..6342.35 rows=48519 width=4)
4	Recheck Cond: ((id >= 1) AND (id <= 100000000))
5	-> Bitmap Index Scan on b_id_idx (cost=0.00..1177.44 rows=48519 width=0)
6	Index Cond: ((id >= 1) AND (id <= 100000000))

PostgreSQL:

	QUERY PLAN
1	Index Only Scan using b_id_idx on b (cost=0.42..4630.53 rows=46909 width=4)
2	Index Cond: ((id >= 1) AND (id <= 100000000))

In openGauss, it doesn't know if the data is already in order, so it need a extra sort on the data. But in PostgreSQL, the data are stored already in increase order, so it only needs to scan the index.

That's why there is a great difference in efficiency between them.

## 5. Special test of the strengths of openGauss

### (1) Concurrent Connection

openGauss uses thread pool model instead of progress model in PostgreSQL, so it is announced to have better concurrent connection performance. We use **sysbench** to test its

ability.

After tries, maybe due to the limitation of the device or the version of openGauss, even if changing the value of max\_connections, the number of connections at the same time could not exceed 200. So finally, we set max\_connections to 256 and test the condition that 100 and 200 users connect to the database at the same time.

- a. Create a new database sysbench\_test in both openGauss and PostgreSQL.
- b. Init sysbench Test Tables

openGauss

```
sysbench --db-driver=pgsql --pgsql-host=localhost --pgsql-port=15432 --pgsql-user=gaussdb --pgsql-password=Abc@1234 --pgsql-db=sysbench_test --tables=1 --threads=4 --time=60 --report-interval=10 oltp_read_write prepare
```

PostgreSQL

```
sysbench --db-driver=pgsql --pgsql-host=localhost --pgsql-port=25432 --pgsql-user=postgres --pgsql-password=123456 --pgsql-db=sysbench_test --tables=1 --threads=4 --time=60 --report-interval=10 oltp_read_write prepare
```

- c. Run the test (100 connections and 200 connections)

openGauss

```
sysbench --db-driver=pgsql --pgsql-host=localhost --pgsql-port=15432 --pgsql-user=gaussdb --pgsql-password=Abc@1234 --pgsql-db=sysbench_test --threads=100 --time=300 --report-interval=10 oltp_read_write run
```

PostgreSQL

```
sysbench --db-driver=pgsql --pgsql-host=localhost --pgsql-port=25432 --pgsql-user=postgres --pgsql-password=123456 --pgsql-db=sysbench_test --threads=100 --time=300 --report-interval=10 oltp_read_write run
```

- d. Result

openGauss(100 connections):

```
SQL statistics:
  queries performed:
    read:          59164
    write:         13339
    other:         9210
    total:         81713
  transactions:   3204 (10.22 per sec.)
  queries:        81713 (260.66 per sec.)
  ignored errors: 1022 (3.26 per sec.)
  reconnects:     0 (0.00 per sec.)

General statistics:
  total time:      313.4827s
  total number of events: 3204

Latency (ms):
  min:            6.45
  avg:            9607.33
  max:            89407.48
  95th percentile: 0.00
  sum:            30781883.33

Threads fairness:
  events (avg/stddev): 32.0400/5.29
  execution time (avg/stddev): 307.8188/4.53
```

PostgreSQL(100 connections):

```
SQL statistics:
  queries performed:
    read:          78358
    write:         17518
    other:         12302
    total:         108178
  transactions:    4222 (13.62 per sec.)
  queries:         108178 (349.03 per sec.)
  ignored errors:  1375 (4.44 per sec.)
  reconnects:      0 (0.00 per sec.)

General statistics:
  total time:      309.9383s
  total number of events: 4222

Latency (ms):
  min:             5.08
  avg:             7286.98
  max:             74011.74
  95th percentile: 0.00
  sum:             30765619.90

Threads fairness:
  events (avg/stddev): 42.2200/6.26
  execution time (avg/stddev): 307.6562/2.62
```

openGauss(200 connections):

```
SQL statistics:
  queries performed:
    read:          25508
    write:         4802
    other:         3995
    total:         34305
  transactions:    1102 (3.11 per sec.)
  queries:         34305 (96.87 per sec.)
  ignored errors:  720 (2.03 per sec.)
  reconnects:      0 (0.00 per sec.)

General statistics:
  total time:      354.1211s
  total number of events: 1102

Latency (ms):
  min:             8.26
  avg:             62227.72
  max:             354093.61
  95th percentile: 0.00
  sum:             68574945.23

Threads fairness:
  events (avg/stddev): 5.5100/1.99
  execution time (avg/stddev): 342.8747/12.32
```

PostgreSQL(200 connections):

```
SQL statistics:
  queries performed:
    read:          32046
    write:         5939
    other:         5077
    total:         43062
  transactions:    1359 (3.97 per sec.)
  queries:         43062 (125.92 per sec.)
  ignored errors:  930 (2.72 per sec.)
  reconnects:      0 (0.00 per sec.)

General statistics:
  total time:      341.9858s
  total number of events: 1359

Latency (ms):
  min:             8.38
  avg:             48571.85
  max:             262182.22
  95th percentile: 0.00
  sum:             66009138.96

Threads fairness:
  events (avg/stddev): 6.7950/2.11
  execution time (avg/stddev): 330.0457/10.74
```

openGauss does **not** show any strength on concurrent connection compared to PostgreSQL in this test. But the difference becomes **smaller** as the number of connections increases. So maybe when the number of connections is **extremely large**, openGauss will perform better, but the experiment cannot be carried out on a personal computer.

## (2) SQL Bypass

When the operation is simple enough, openGauss will use SQL Bypass to change the execution strategy of the SQL, in order to optimize the constant.

We take the statement of inserting one value at once as an example.

- Create a new table **d** with one column id of int type.
- Use the  $10^6$  numbers generated in part one.
- Insert them **one by one** to d. Use "**explain analyse**" statement to get the runtime and sum up the results. At the same time, use "**docker stats**" to get the CPU and Memory Usage.

---

```

1 for(int i=1;i<=1000000;i++)
2 {
3     scanf("%d",&in);
4     vec.push_back(in);
5 }
6 try
7 {
8     connection C1("dbname = pro3 user = gaussdb password = Abc@1234 host =
        localhost port = 15432");
9     double tot1=0,tot2=0;
10    if(C1.is_open())
11    {
12        for(int i=0;i<1000000;i++)
13        {
14            nontransaction W(C1);
15            tot1+=get_time(W.exec("explain analyse insert into d values (" +
                to_string(vec[i]) + ");"));
16        }
17    }
18    connection C2("dbname = pro3 user = postgres password = 123456 host = localhost
        port = 25432");
19    if(C2.is_open())
20    {
21        for(int i=0;i<1000000;i++)
22        {
23            nontransaction W(C2);
24            tot2+=get_time(W.exec("explain analyse insert into d values (" +
                to_string(vec[i]) + ");"));
25        }

```

```

26     }
27     printf("Runtime of openGauss : %f\nRuntime of PostgreSQL : %f\n",tot1,tot2);
28 }
29 catch(const std::exception& e)
30 {
31     std::cerr<<e.what()<<'\n';
32 }

```

d. Use "explain" to obtain the concrete process.

openGauss:

	QUERY PLAN
1	[Bypass]
2	Insert on d (cost=0.00..0.01 rows=1 width=0)
3	-> Result (cost=0.00..0.01 rows=1 width=0)

PostgreSQL:

	QUERY PLAN
1	Insert on d (cost=0.00..0.01 rows=0 width=0)
2	-> Result (cost=0.00..0.01 rows=1 width=4)

openGauss has used SQL Bypass but PostgreSQL has not.

e. Result:

For insert, the runtime, CPU and Memory usage of openGauss are all much more than PostgreSQL. Although SQL Bypass has been used, the time efficiency does not show significant change.

Result of code (unit: ms)

```

● mio@MacBook-Pro-Mio insert_test2 % ./gauss
Runtime of openGauss : 35868.847000
Runtime of PostgreSQL : 3375.017000

```

Result of docker stats

NAME	CPU %	MEM USAGE / LIMIT	MEM %	NET I/O	BLOCK I/O	PIDS
project3-opengauss	47.04%	1.004GiB / 15.59GiB	6.44%	827MB / 83.2GB	44MB / 53.9GB	41
project3-postgres	15.95%	97.04MiB / 15.59GiB	0.61%	272MB / 528MB	8.66MB / 925MB	10

## 6. Conclusion

From the experiment, openGauss does not show significant strengthes compared to PostgreSQL in basic functions. The only condition that it is obviously better than PostgreSQL in runtime is select statement with order by. But in table with index, PostgreSQL is also faster than openGauss on this query because of omitting the process of sort. And the announced strengths of openGauss does not work well in this test where the data size and connections are not extremely large.

To analyze the reason, on the one hand, openGauss is an RDBMS specially faced engineering use where the data size and the number of connections are extremely large. But optimizations of more conditions may lead to a large constant in the basic conditions. That may be why it uses more CPU and memory in most tests, and performs not that ideal.

On the other hand, this version of openGauss is developed based on an old version of PostgreSQL 10. The new version of PostgreSQL introduced many new optimizations in generating plan that may not be updated to openGauss. So, it may perform bad in some special conditions, such as select statement with order by on a table with index.

In conclusion, openGauss is not a very suitable RDBMS for basic personal use. And there is a long way to go for openGauss to keep track of the steps of PostgreSQL and show its own strengths.