DBMS Performance Evaluation Project Report

Zhenyu Wang

December 22, 2024

Abstract

This paper presents a comparative evaluation of the performance efficiency between openGauss and PostgreSQL. To assess the databases' capabilities, we utilized the pgbench [Doc24] benchmarking tool to conduct a series of tests. The evaluation focused on key performance metrics, including query response time, throughput, horizontal scaling, and vertical scaling. The results reveal notable differences in performance, with PostgreSQL consistently outperforming openGauss in several critical areas, particularly in query response time and throughput. However, openGauss shows potential in distributed and scalable environments, offering a promising solution for applications prioritizing scalability over raw transaction throughput or query latency. The findings provide insights into the strengths and weaknesses of both databases, helping to guide decision-making for different workload requirements.

Contents

Project Reports

Test Environment Overview

Hardware Specifications

Component	Details
CPU Model	12th Gen Intel(R) Core(TM) i9-12900H
Base Speed	2.50 GHz
Cores	14
Logical Processors	20
Virtualization	Enabled
L1 Cache	1.2 MB
L2 Cache	11.5 MB
L3 Cache	24.0 MB

CPU Specifications

Component	Details
Memory Capacity	32.0 GB
Memory Speed	4800 MT/s
Slots Used	2/2
Form Factor	SODIMM
Available Memory	10.9 GB
Used Memory	26.2/36.4 GB

Memory Specifications

Component	Details
Disk 0 (C:)	NVMe PM9A1 1024GB (Samsung)
Disk Type	SSD
Disk Capacity	954 GB
Disk 1 (D:)	NVMe Samsung SSD 980 PRO 1TB
Disk Type	SSD
Disk Capacity	932 GB

Disk Specifications

Component	Details
Wi-Fi Adapter	Killer(R) Wi-Fi 6E AX1675i
Network	SUSTech-wifi
IPv4 Address	10.26.77.52
IPv6 Address	2001:da8:201d:1109::4169

Wi-Fi Specifications

Component	Details
GPU 0 Model	Intel(R) Iris(R) Xe Graphics
GPU 1 Model	NVIDIA GeForce RTX 3070 Ti Laptop GPU

GPU Specifications

Software Specifications

Software Component	Version
openGauss	3.0.0 (Build 02c14696)
PostgreSQL	14.12
Docker Desktop	4.34.3.170107
pgbench[Doc24]	17rc1 (Server 9.2.4)

Software Specifications

`PostgreSQL` is a widely-used open-source relational database management system (RDBMS) known for its stability, extensibility, and standards compliance. It supports a variety of data types and advanced features like ACID compliance and complex queries, making it ideal for enterprise applications and data warehousing. Its strong community and extensive documentation enhance its reputation as a reliable solution.

`openGauss`, developed by Huawei, is an open-source RDBMS optimized for high performance and scalability in distributed environments. Based on PostgreSQL, it incorporates enhancements for reliability and security, making it well-suited for large-scale, data-intensive applications such as financial systems and e-commerce. Its architecture is designed for cloud-native and big data use cases, emphasizing fault tolerance and horizontal scalability.

However, does `openGauss` truly live up to the claims made about it? This project seeks to investigate and assess this question.

Task 1

When evaluating the quality of a database, we focus on four key metrics that are both important and easy to test. These metrics are crucial in most database evaluations and can be reliably quantified through standardized testing.

Query Response Time

Importance: Query response time directly impacts the user experience, especially under high concurrency. The speed of query execution is a core metric for evaluating database performance. Whether for **Online Transaction Processing** or **Online Analytical Processing** applications, fast query response is essential for us.

Testing Method: We can use benchmark tools (such as `pgbench`, `sysbench`, `HammerDB`, etc.) to execute queries of varying complexity, measure the average response time for each query type, and observe variations under different loads.

Throughput

Importance: Throughput measures the number of transactions (or queries) a database can process per second. It is an important metric for evaluating the processing capability of the database. In high-load scenarios, throughput reflects the efficiency and reliability of the database.

Testing Method: By simulating concurrent user requests, we can use tools like `pgbench` for stress testing. We will measure the number of transactions or queries the database handles per second, and assess throughput variations under different load conditions.

Scalability

Importance: As data volume and the number of users increase, database scalability becomes critical. The ability of a database to scale effectively to handle larger data volumes or higher concurrency is key to evaluating its long-term usability.

Testing Method: We can test scalability through both horizontal and vertical scaling. We will evaluate how the database's performance changes when adding nodes (horizontal scaling) or increasing hardware resources (vertical scaling). For instance, we will test how the database handles increased nodes in a distributed environment by measuring throughput, response time, and more.

Task 2

To test the efficiency differences between the two databases (`openGauss` and `PostgreSQL`), I used the `pgbench` benchmarking tool. However, the default test script for `pgbench` is not compatible with `openGauss`, so I had to create a custom script for efficiency testing. Next, I will test both databases on the four different aspects of efficiency mentioned in **Task 1**, in order to compare the differences between the two databases.

Query Response Time

To test the **Query Response Time** of two databases, I designed the `SQL` script `query_pgbench_init.sql`[Dia24] and captured the test performance through `pgbench`.

This `SQL` script performs a series of tasks on both `openGauss` and `PostgreSQL` databases to evaluate their query response capabilities. It does so by executing a variety of queries that test different aspects of the database's performance, such as simple queries, aggregations, data modifications, and more complex operations. Below is a breakdown of each task and how it contributes to evaluating the databases' query response time:

Creating a Table and Inserting Data: The script starts by creating a table called `test_table`, which includes columns for `id`, `name`, `amount`, and `date_created`. The table is populated with 100000 records where each record's name is generated as `"Test User"` followed by a number, and the amount is set as `i*10`. This large dataset simulates real-world data and sets the stage for testing how both `openGauss` and `PostgreSQL` handle larger tables.

Simple Query (Count Query): The first query in the transaction is a simple `SELECT COUNT(*)` that counts the number of rows where the `amount` is greater than 500. This tests the database's ability to perform basic filtering and aggregation operations. It's a relatively simple query that is likely to execute quickly, but it still stresses the database's ability to scan the table and perform the filtering operation efficiently.

Aggregation Query (Average Calculation): The second query calculates the average value of `amount` for records where `amount` is between 100 and 1000. This is an aggregation query that requires the database to filter the rows and compute an average. It tests the database's ability to perform aggregation efficiently, especially when dealing with a large dataset.

Complex Query with Sorting and Limiting: This query retrieves records where the `amount` is greater than 5000, orders them by `amount` in descending order, and limits the result to 100 records. This query tests the database's ability to handle sorting operations, which can be computationally expensive, especially when sorting large sets of data. The `LIMIT` clause also forces the database to stop processing once the desired number of rows is found, which tests the database's efficiency in handling large result sets.

Update Operation: The script then performs an `UPDATE` operation, which increases the `amount` by 100 for all records where `amount` is less than 500. Update queries modify the database, and this task evaluates the database's ability to handle write operations, particularly those that affect a significant portion of the data. Write operations, especially on large datasets, can be a performance bottleneck, so this test helps assess both `openGauss` and `PostgreSQL`'s response to such load.

Simple Insert Operation: A new row is inserted into the `test_table`. This simple insert operation is included to assess how the database handles single record inserts. While this may not be as intensive as bulk insert operations, it helps to evaluate basic insert performance in isolation.

Delete Operation with Subquery: This task deletes the first 100 records from the table where `amount` is greater than 5000. The query uses a subquery to identify the records to delete. The inclusion of a subquery in the `DELETE` operation tests how efficiently the database can handle more complex delete operations that require additional processing (e.g., selecting records through a subquery before performing the delete).

Range Query: This query retrieves all records where the `id` is between 10000 and 20000. This simulates a query with a specified range condition, which is a typical scenario in real-world applications (e.g., paginated results or filtered data). It tests how the database handles filtering based on a range condition and how well it can handle intermediate-sized result sets.

Complex Aggregation with Grouping: The next query performs a complex aggregation, grouping records by a calculated range of `amount` values (`Low`, `Medium`, `High`) and calculating the average amount for each group. This query tests the database's ability to handle grouping and complex aggregation, which is often necessary for reports or statistical analysis in real-world applications.

Query with Multiple Conditions: The script then executes a query that counts records where `amount` is greater than 500 and less than 1000. This query tests the database's ability to efficiently handle queries with multiple filtering conditions. It provides insight into how well the database can handle more complex filtering operations, especially when combined with ranges.

Committing the Transaction: Finally, the script commits the transaction to save all changes made by the `INSERT`, `UPDATE`, and `DELETE` operations. While not directly related to querying, this step evaluates the database's overhead in committing the changes.

By executing the following statement on the terminal, the above script can be used to perform query response time tests on two databases.

```
pgbench -h localhost -U gaussdb -d postgres -p 15432 -f
    query_pgbench_init.sql -T 60
pgbench -h localhost -U postgres -d postgres -p 5432 -f
    query_pgbench_init.sql -T 60
```

The test results captured by `pgbench` are as follows:

Metric	**openGauss**	**PostgreSQL**
Number of Transactions Processed	111	253
Number of Failed Transactions	0 (0.000%)	0 (0.000%)
Latency Average (ms)	543.290	237.654
Initial Connection Time (ms)	28.834	21.370
TPS (without initial connection time)	1.840636	4.207801

Performance Comparison between `openGauss` and `PostgreSQL`

The performance comparison between `openGauss` and `PostgreSQL` based on the `pgbench` results shows the following key differences:

1. Transactions Processed:

`openGauss` : 111 transactions in 60 seconds

`PostgreSQL` : 253 transactions in 60 seconds

`PostgreSQL` processed significantly more transactions (253) than `openGauss` (111) in the same time frame, indicating that `PostgreSQL` has better transaction throughput in this scenario.

2. Latency:

`openGauss` : Average latency of 543.290 ms

`PostgreSQL` : Average latency of 237.654 ms

`PostgreSQL`'s average latency is much lower (237.654 ms) compared to `openGauss` (543.290 ms), suggesting that `PostgreSQL` is more responsive and handles each transaction more efficiently, which may contribute to higher throughput.

3. Initial Connection Time:

`openGauss` : 28.834 ms

`PostgreSQL` : 21.370 ms

`PostgreSQL` has a slightly faster initial connection time compared to `openGauss`, though the difference (7.464 ms) is relatively small. This could indicate a slightly more optimized connection handling process in `PostgreSQL`.

4. Throughput (Transactions Per Second):

`openGauss` : 1.840636 TPS

`PostgreSQL` : 4.207801 TPS

`PostgreSQL` achieves a significantly higher transaction rate, nearly **2.3 times** higher than `openGauss`. This is in line with the higher number of transactions processed and lower latency. Higher TPS indicates that `PostgreSQL` can process more transactions within the same time period, which reflects better performance in handling the workload provided by `pgbench`.

`PostgreSQL` outperforms `openGauss` in terms of throughput, latency, and the number of transactions processed in the given test. These results suggest that for this specific benchmarking setup, `PostgreSQL` has better performance, particularly in handling simple transactions with lower latency and higher transaction rates. The difference in performance could be attributed to several factors such as:

- `PostgreSQL` is more mature and optimized for general workloads compared to `openGauss`.
- Potential differences in internal configurations, indexing, or query optimizations between `openGauss` and `PostgreSQL`.

Throughput

Next, we will test the throughput differences between two different databases, `openGauss` and `PostgreSQL`. Similar to the previous test, we will use `pgbench` for custom script testing. The script files are `pgbench_init.sql`[Dia24] and `tps_qps_pgbench_init.sql`[Dia24].

The `tps_qps_pgbench_init.sql`[Dia24] script is designed to evaluate the throughput performance (TPS and QPS) of the database under high-concurrency load. Unlike the previous script that primarily focused on query response time, this script emphasizes throughput by simulating multiple concurrent user requests using `pgbench`. Below is a detailed explanation of how this script tests throughput and its specific optimizations for multi-client concurrency testing.

Differences from Previous Script:

- Broader Range of Operations: Unlike the previous script, which focused mainly on query response time, this script includes a wider range of operations such as inserts, updates, and deletes. These operations simulate a more realistic workload and allow for a more comprehensive evaluation of database performance.
- Combination of Different Operations: By mixing various operations (read, write, update, and delete), this script provides a more holistic view of how the database handles different types of transactions under high load.

The script optimizes multi-client concurrency testing by simulating multiple clients and threads with the `-c` and `-j` parameters, reflecting real-world user activity. It wraps all operations in a transaction to test transaction handling, and includes a mix of read and write operations (`SELECT`, `INSERT`, `UPDATE`, `DELETE`) to simulate varied workloads. These optimizations allow the script to evaluate the database's throughput and performance under high concurrency and diverse transaction types.

By executing the following statement on the terminal, the above script can be used to perform query response time tests on two databases.

```
1 pgbench -h localhost -p 5432 -U postgres -d postgres -f
     tps_qps_pgbench_init.sql -T 60 -c 100 -j 32
2 pgbench -h localhost -p 15432 -U gaussdb -d postgres -f
     tps_qps_pgbench_init.sql -T 60 -c 100 -j 32
```

The test results captured by `pgbench` are as follows:

Metric	**openGauss**	**PostgreSQL**
Number of Transactions Processed	5503	7231
Number of Failed Transactions	5 (0.091%)	79 (1.081%)
Latency Average (ms)	1102.238	830.816
Initial Connection Time (ms)	99.029	441.713
TPS (without initial connection time)	90.642143	119.062854

Performance Comparison between `openGauss` and `PostgreSQL` (100 Clients, 32 Threads)

The performance comparison between `openGauss` and `PostgreSQL` based on the `pgbench` results shows the following key differences:

1. **Transactions Processed:**

`openGauss` (100 clients) : 5503 transactions processed

`PostgreSQL` (100 clients) : 7231 transactions processed

In the 100-client scenario, `PostgreSQL` processes more transactions (7231) compared to `openGauss` (5503), showing that `PostgreSQL` handles higher concurrency more efficiently, processing a larger number of transactions in the same amount of time.

2. **Failed Transactions:**

`openGauss` (100 clients) : 5 failed transactions (0.091%)

`PostgreSQL` (100 clients) : 79 failed transactions (1.081%)

`openGauss` has a significantly lower failure rate (0.091%) compared to `PostgreSQL` (1.081%). This suggests that `openGauss` is more stable and resilient under high concurrency, while `PostgreSQL` is more prone to failures as the number of clients increases.

3. Average Latency:

openGauss (100 clients) : 1102.238 ms (including failures)

PostgreSQL (100 clients) : 830.816 ms (including failures)

PostgreSQL has lower average latency (830.816 ms) compared to openGauss (1102.238 ms). Despite openGauss's better stability, it suffers from higher latency, indicating potential inefficiencies in handling concurrent requests or resource contention under load.

4. Initial Connection Time:

openGauss (100 clients) : 99.029 ms

PostgreSQL (100 clients) : 441.713 ms

openGauss has a significantly lower initial connection time compared to PostgreSQL, which is more than four times higher. This indicates that openGauss handles initial client connections much more efficiently, which could benefit applications that require a high number of rapid connections.

5. Transactions Per Second (TPS):

openGauss (100 clients) : 90.642143 TPS (without initial connection time)

PostgreSQL (100 clients) : 119.062854 TPS (without initial connection time)

PostgreSQL achieves higher TPS (119.06) compared to openGauss (90.64), demonstrating that PostgreSQL is better at scaling with high concurrency and can process more transactions per second under the 100-client, 32-thread scenario.

From the comparison between openGauss and PostgreSQL under the 100-client and 32-thread load, we observe that PostgreSQL generally performs better in terms of transactions processed and TPS (transactions per second), while openGauss excels in stability and connection efficiency. Specifically, PostgreSQL processed 7231 transactions compared to 5503 by openGauss, and its TPS was higher (119.06 vs. 90.64 for openGauss). Additionally, PostgreSQL achieved lower average latency (830.816 ms vs. 1102.238 ms) despite handling a larger number of transactions.

However, openGauss demonstrated a significantly lower rate of failed transactions (0.091% vs. 1.081% for PostgreSQL), indicating better stability under high concurrency. Furthermore, openGauss had a faster initial connection time (99.03 ms vs. 441.71 ms for PostgreSQL), making it more efficient in scenarios where rapid client connections are essential.

In summary, openGauss offers superior stability and connection efficiency under high concurrency, while PostgreSQL outperforms in throughput and latency, processing more transactions per second with lower response times.

Scalability

When evaluating the performance of a database, scalability is a critical factor to consider. As data volume and number of users grow, the database must be able to scale efficiently without service interruptions to handle the increased load. The scalability of the database is typically achieved through two approaches: **horizontal scaling** and **vertical scaling**. Horizontal scaling involves adding more server nodes to distribute the load, making it suitable for handling large amounts of data and high concurrency. In contrast, vertical scaling enhances the hardware resources (such as CPU, memory, and storage) of a single server to improve performance, often used for workloads that require significant computational power. This article will explore how to test the scalability of a database using these two methods and assess its ability to adapt to the growing demands of data and users.

Horizontal Scaling

To test horizontal scaling of databases, I focused on distributing the workload across multiple nodes to evaluate how well the system handles increasing traffic. For this, I used `pgpool2`[Pro24b][Pro24a] as the load balancer, which helps distribute queries across the master and replica nodes. The general process involves:

- **Setting up `pgpool2`:** I installed and configured `pgpool2` to manage multiple `PostgreSQL` or `openGauss` nodes, with the master handling write operations and replicas handling read operations. This setup ensures load balancing and efficient resource use.
- **Configuring Nodes:** I set up at least one master node and multiple replica nodes, enabling streaming replication on `PostgreSQL` and standby replication on `openGauss`. `pgpool2` is configured to direct read queries to replicas and write queries to the master.
- **Running the Test:** Using the script `horizontal_pgbench_init.sql`[Dia24], I performed a series of queries that simulate real-world database interactions, such as aggregation, inserts, updates, and deletes. This allowed me to test how the system handles different types of operations and how well the load is balanced across nodes.

In order to set up the horizontal scaling environment effectively, I made the following key configuration changes in both `postgresql.conf` and `pgpool.conf`:

In `postgresql.conf`:

- `wal_level = replica`: Enables replication.
- `max_wal_senders = 10`: Allows multiple replication connections.
- `hot_standby = on`: Enables read-only queries on replicas.

- `primary_conninfo`: Configures standby nodes to connect to the master for replication.

In `pgpool.conf`:

- `backend_hostname0`: Points to the master node.
- `backend_hostnamex`: Points to a replica node.
- `load_balance_mode = on`: Distributes read queries across replicas.
- `master_slave_mode = on`: Routes writes to the master and reads to replicas.

These changes optimize the system for horizontal scaling by balancing read traffic and managing replication efficiently.

The `SQL` script `horizontal_pgbench_init.sql`[Dia24] performs various operations on the `test_table` and can be used to evaluate how effectively the system handles distributed queries in a horizontally scaled environment. The script includes `SELECT`, `UPDATE`, `INSERT`, and `DELETE` queries that will test both read and write operations, allowing you to observe the system's ability to handle the load across multiple nodes.

- **Read-heavy Queries:** Operations like `SELECT COUNT(*)`, `AVG()`, and `ORDER BY` in `horizontal_pgbench_init.sql`[Dia24] benefit from horizontal scaling, as they can be distributed across replica nodes.
- **Write Operations:** The script also includes `UPDATE`, `INSERT`, and `DELETE` queries, which will be handled by the master node, allowing me to measure its capacity to handle write-intensive workloads.

This test helped me observe how adding more nodes impacts performance for both `PostgreSQL` and `openGauss`. Ideally, with more nodes, I should see improved read throughput and reduced latency, although write performance will still depend on the master node's capabilities.

By executing the following statement on the terminal, the above script can be used to perform query response time tests on two databases.

```
1 pgbench -h localhost -p 15432 -U gaussdb -d postgres -f
     horizontal_pgbench_init.sql -T 60 -c 100 -j 32
2 pgbench -h localhost -p 5432 -U postgres -d postgres -f
     horizontal_pgbench_init.sql -T 60 -c 100 -j 32
```

The test results captured by `pgbench` are as follows:

Metric	**openGauss (0 Nodes)**	**PostgreSQL (0 Nodes)**
Number of Transactions Processed	4921	8406
Latency Average (ms)	1227.744	713.484
Initial Connection Time (ms)	103.245	427.833
TPS (without initial connection time)	81.45	140.16
Metric	**openGauss (1 Node)**	**PostgreSQL (1 Node)**
Number of Transactions Processed	6505	8711
Latency Average (ms)	928.909	687.545
Initial Connection Time (ms)	81.497	465.266
TPS (without initial connection time)	107.65	145.45
Metric	**openGauss (2 Nodes)**	**PostgreSQL (2 Nodes)**
Number of Transactions Processed	6769	9640
Latency Average (ms)	893.800	622.160
Initial Connection Time (ms)	78.073	414.345
TPS (without initial connection time)	111.88	160.73
Metric	**openGauss (4 Nodes)**	**PostgreSQL (4 Nodes)**
Number of Transactions Processed	7266	10692
Latency Average (ms)	830.829	560.618
Initial Connection Time (ms)	94.264	425.048
TPS (without initial connection time)	120.36	178.37
Metric	**openGauss (8 Nodes)**	**PostgreSQL (8 Nodes)**
Number of Transactions Processed	7912	11860
Latency Average (ms)	760.361	505.144
Initial Connection Time (ms)	281.306	398.165
TPS (without initial connection time)	131.52	197.96

Performance Comparison between `openGauss` and `PostgreSQL` (100 Clients, 32 Threads) with Varying Number of Nodes

1. Transactions Processed:

Metric	openGauss (Nodes)	PostgreSQL (Nodes)
Number of Transactions Processed	4921 (0)	8406 (0)
Number of Transactions Processed	6505 (1)	8711 (1)
Number of Transactions Processed	6769 (2)	9640 (2)
Number of Transactions Processed	7266 (4)	10692 (4)
Number of Transactions Processed	7912 (8)	11860 (8)

Number of Transactions Processed (Scaling with Nodes)

`openGauss` shows a steady increase in the number of transactions processed as the number of nodes increases, but the growth rate is more gradual. The increase from 0 to 1 node is 1584 transactions, from 1 to 2 nodes is 264 transactions, and further increases are smaller: 4 nodes to 8 nodes shows a gain of 646 transactions.

`PostgreSQL`, on the other hand, demonstrates a more consistent and linear increase. From 0 to 1 node, the increase is 305 transactions, from 1 to 2 nodes is 929 transactions, and from 2 to 4 nodes is 1052 transactions. The gains become progressively larger as nodes increase, with the largest increase from 4 to 8 nodes (1168 transactions).

`PostgreSQL` exhibits a more robust improvement in transactions processed as more nodes are added, suggesting better horizontal scalability. It scales more efficiently with each additional node compared to `openGauss`.

2. Latency Average:

Metric	openGauss (Nodes)	PostgreSQL (Nodes)
Latency Average (ms)	1227.744 (0)	713.484 (0)
Latency Average (ms)	928.909 (1)	687.545 (1)
Latency Average (ms)	893.800 (2)	622.160 (2)
Latency Average (ms)	830.829 (4)	560.618 (4)
Latency Average (ms)	760.361 (8)	505.144 (8)

Latency Average (ms) (Scaling with Nodes)

`openGauss` shows a clear reduction in latency as the number of nodes increases, with a noticeable improvement from 0 to 1 node (298.835 ms). However, the rate of improvement slows as more nodes are added. The difference between 4 and 8 nodes is only 70.468 ms.

`PostgreSQL`, meanwhile, has much lower latency across all node configurations and also experiences steady improvements as the number of nodes increases. From 0 to 1 node, the latency drops by 25.939 ms, and the improvement continues to be significant up to 8 nodes, where latency is reduced by 208.34 ms from 0 nodes.

PostgreSQL consistently outperforms openGauss in terms of latency, even as the number of nodes increases. The improvement in openGauss is less pronounced, and its latency is significantly higher than PostgreSQL across all node counts.

3. Initial Connection Time:

Metric	openGauss (Nodes)	PostgreSQL (Nodes)
Initial Connection Time (ms)	103.245 (0)	427.833 (0)
Initial Connection Time (ms)	81.497 (1)	465.266 (1)
Initial Connection Time (ms)	78.073 (2)	414.345 (2)
Initial Connection Time (ms)	94.264 (4)	425.048 (4)
Initial Connection Time (ms)	281.306 (8)	398.165 (8)

Initial Connection Time (ms) (Scaling with Nodes)

openGauss consistently performs better than PostgreSQL in terms of initial connection time. The connection time remains under 100 ms for 0 to 2 nodes, which is significantly better than PostgreSQL, whose connection time is much higher and increases with additional nodes.

From 0 to 1 node, openGauss improves from 103.245 ms to 81.497 ms, but then connection time increases significantly from 4 nodes (94.264 ms) to 8 nodes (281.306 ms). This sharp increase suggests that openGauss may face some scalability issues when handling a large number of nodes, resulting in higher overhead for connections.

openGauss has a clear advantage in initial connection times, but the connection time grows disproportionately with the increase in nodes, which could be a concern in highly dynamic or short-duration workloads.

4. Transactions Per Second (TPS):

Metric	openGauss (Nodes)	PostgreSQL (Nodes)
TPS (without Initial Connection Time)	81.450 (0)	140.157 (0)
TPS (without Initial Connection Time)	107.653 (1)	145.445 (1)
TPS (without Initial Connection Time)	111.881 (2)	160.730 (2)
TPS (without Initial Connection Time)	120.361 (4)	178.374 (4)
TPS (without Initial Connection Time)	131.516 (8)	197.963 (8)

Transactions Per Second (TPS) (Scaling with Nodes, Without Initial Connection Time)

openGauss sees a consistent increase in TPS as nodes are added, with the rate of improvement slowing down as the number of nodes grows. The increase from 0 to 1 node is 26.2 TPS, and from 4 to 8 nodes, the increase is only 11.16 TPS.

PostgreSQL exhibits a more significant improvement in TPS. From 0 to 1 node, TPS increases by 5.29, and the increase from 4 to 8 nodes is 19.59 TPS. The overall TPS growth

in PostgreSQL is more consistent and pronounced across all node levels, indicating better horizontal scaling performance.

PostgreSQL outperforms openGauss in TPS, with a more consistent increase in throughput as the number of nodes increases. openGauss improves with each added node, but its throughput still lags behind PostgreSQL at all stages.

In conclusion, PostgreSQL demonstrates superior horizontal scalability compared to openGauss. It consistently outperforms openGauss in terms of transactions processed, transactions per second (TPS), and latency reduction as the number of nodes increases. PostgreSQL's performance improvements are more consistent and linear, indicating its ability to handle high-concurrency and large-scale workloads effectively. While openGauss excels in initial connection times, it shows slower and less pronounced improvements in transaction throughput and latency as nodes are added. The significant increase in connection time at 8 nodes in openGauss also suggests potential scalability challenges. Therefore, PostgreSQL is more suitable for deployments requiring robust horizontal scaling and optimized handling of large-scale, high-performance operations, while openGauss might be more efficient in environments with lighter loads or fewer nodes.

Vertical Scaling

To evaluate the vertical scalability of the two databases, openGauss and PostgreSQL, we focus on testing how the databases perform as we allocate more CPU cores and memory resources. Vertical scaling involves increasing the resources (such as CPU and memory) assigned to a single machine, which allows us to observe how the databases respond to more processing power and memory.

For this test, we used specific commands to control hardware resource allocation through the use of Docker, ensuring that each test scenario had different resource configurations. The commands used for this purpose were as follows:

```
1 --cpuset-cpus="0,1" --cpus="2.0" --memory="1g"
2 --cpuset-cpus="0,1,2,3" --cpus="4.0" --memory="2g"
3 --cpuset-cpus="0,1,2,3,4,5,6,7" --cpus="8.0" --memory="4g"
```

By testing the databases with varying amounts of CPU and memory resources, we aim to understand how each database scales when more resources are made available, and compare their performance under different resource configurations.

The vertical_pgbench_init.sql[Dia24] script is designed to evaluate database performance by executing a series of queries that simulate typical operations, including counting, averaging, updating, inserting, and deleting records. By testing these operations, the script highlights how different hardware resource allocations impact the database's performance, demonstrating the performance optimizations achieved through vertical scalability.

By executing the following statement on the terminal, the above script can be used to perform query response time tests on two databases.

```
pgbench -h localhost -p 5432 -U postgres -d postgres -f
    vertical_pgbench_init.sql -T 60 -c 20 -j 8
pgbench -h localhost -p 15432 -U gaussdb -d postgres -f
    vertical_pgbench_init.sql -T 60 -c 20 -j 8
```

The test results captured by `pgbench` are as follows:

Metric	**openGauss (2 Cores 1GB)**	**PostgreSQL (2 Cores 1GB)**
Transactions Processed	698	1810
Latency Average (ms)	1744.035	667.031
Initial Connection Time (ms)	47.323	91.418
TPS	11.47	29.98
Metric	**openGauss (4 Cores 2GB)**	**PostgreSQL (4 Cores 2GB)**
Transactions Processed	2032	3563
Latency Average (ms)	592.783	337.194
Initial Connection Time (ms)	68.248	79.918
TPS	33.74	59.31
Metric	**openGauss (8 Cores 4GB)**	**PostgreSQL (8 Cores 4GB)**
Transactions Processed	3396	7157
Latency Average (ms)	353.948	167.647
Initial Connection Time (ms)	67.609	95.967
TPS	56.51	119.30

Performance Comparison between `openGauss` and `PostgreSQL` (20 Clients, 8 Threads)

1. Transactions Processed:

Metric	**openGauss (Nodes)**	**PostgreSQL (Nodes)**
Number of Transactions Processed	698 (2 Cores 1GB)	1810 (2 Cores 1GB)
Number of Transactions Processed	2032 (4 Cores 2GB)	3563 (4 Cores 2GB)
Number of Transactions Processed	3396 (8 Cores 4GB)	7157 (8 Cores 4GB)

Number of Transactions Processed (Scaling with Nodes)

`PostgreSQL` consistently processes more transactions than `openGauss` across all configurations. As the number of cores and memory increases, `PostgreSQL` shows a more significant increase in transaction processing capabilities, whereas `openGauss` scales less effectively in

comparison. This suggests that PostgreSQL handles increased resources better in terms of transaction throughput.

2. Latency Average:

Metric	openGauss (Nodes)	PostgreSQL (Nodes)
Latency Average (ms)	1744.035 (2 Cores 1GB)	667.031 (2 Cores 1GB)
Latency Average (ms)	592.783 (4 Cores 2GB)	337.194 (4 Cores 2GB)
Latency Average (ms)	353.948 (8 Cores 4GB)	167.647 (8 Cores 4GB)

Latency Average (ms) (Scaling with Nodes)

PostgreSQL exhibits consistently lower latency compared to openGauss across all configurations. In the 2-core, 1GB configuration, PostgreSQL has a latency of 667.031 ms, which is significantly lower than openGauss's 1744.035 ms. As the configurations scale up, PostgreSQL maintains a clear advantage in latency reduction, suggesting that PostgreSQL is more optimized for reducing response times at higher resource configurations.

3. Initial Connection Time:

Metric	openGauss (Nodes)	PostgreSQL (Nodes)
Initial Connection Time (ms)	47.323 (2 Cores 1GB)	91.418 (2 Cores 1GB)
Initial Connection Time (ms)	68.248 (4 Cores 2GB)	79.918 (4 Cores 2GB)
Initial Connection Time (ms)	67.609 (8 Cores 4GB)	95.967 (8 Cores 4GB)

Initial Connection Time (ms) (Scaling with Nodes)

openGauss consistently has faster initial connection times compared to PostgreSQL across all configurations. In the 2-core, 1GB configuration, openGauss connects in 47.323 ms, while PostgreSQL takes 91.418 ms. This trend continues as the configuration scales up. Faster initial connection times can be advantageous in environments requiring rapid client connection and disconnection.

4. Transactions Per Second (TPS):

Metric	openGauss (Nodes)	PostgreSQL (Nodes)
TPS (without Initial Connection Time)	11.467 (2 Cores 1GB)	29.983 (2 Cores 1GB)
TPS (without Initial Connection Time)	33.739 (4 Cores 2GB)	59.313 (4 Cores 2GB)
TPS (without Initial Connection Time)	56.505 (8 Cores 4GB)	119.298 (8 Cores 4GB)

Transactions Per Second (TPS) (Scaling with Nodes, Without Initial Connection Time)

PostgreSQL outperforms openGauss in terms of TPS at all configurations. openGauss shows a noticeable improvement as the configurations scale up, reaching 56.505 TPS at 8 cores and 4GB, but PostgreSQL still maintains a substantial lead, with 119.298 TPS in the same

configuration. This indicates that PostgreSQL scales better in terms of throughput as the resources increase.

In conclusion, openGauss performs more efficiently under lower resource configurations due to its optimized handling of fewer clients and lower concurrency, which results in higher stability and fewer transaction failures. However, as resources scale (more cores and memory), PostgreSQL shows superior efficiency, processing more transactions with lower latency. This difference in scalability can be attributed to PostgreSQL's better optimization for parallel processing and higher concurrency, making it more suitable for larger workloads. openGauss, while stable, seems to hit a performance ceiling as the system scales, leading to a less efficient handling of increased load compared to PostgreSQL.

Task 3

Given the detailed data analysis provided in the earlier sections, we now turn to a comprehensive evaluation of the overall performance of openGauss and PostgreSQL. The data highlights key differences and trends across multiple metrics, including **Query Response Time**, **Throughput**, **Horizontal Scaling**, and **Vertical Scaling**. In this section, we summarize the performance characteristics of both databases, outline their respective strengths, and identify areas where each system excels or faces challenges. This analysis aims to offer a clear understanding of how each database performs in various operational scenarios and to help guide decisions based on specific workload requirements.

Key Findings

In terms of query response time, openGauss exhibits significantly higher latency compared to PostgreSQL. On average, openGauss has a query response time of 543.290 ms, while PostgreSQL achieves a more efficient 237.654 ms. This disparity suggests that PostgreSQL is more optimized for handling low-latency queries, making it more suitable for applications where quick query responses are critical.

When we look at Transactions Per Second (TPS), PostgreSQL also performs better, achieving a TPS of 4.21 compared to openGauss's 1.84. This trend indicates that openGauss may not be the best choice for environments that require processing a high number of individual transactions with minimal delay.

When testing for throughput, openGauss still shows some weaknesses compared to PostgreSQL. While openGauss processes a reasonable number of transactions, its throughput remains lower than that of PostgreSQL under heavy workloads (e.g., 100 clients, 32 threads). PostgreSQL achieves a throughput of 119.06 TPS, while openGauss manages only 90.64 TPS under similar conditions. Furthermore, the latency for openGauss is much higher (1102.238 ms compared to 830.816 ms for PostgreSQL), indicating that openGauss might struggle with handling high-throughput transactional workloads efficiently.

In terms of horizontal scalability, openGauss shows the ability to increase throughput as additional nodes are added. However, PostgreSQL still outperforms openGauss as the number of nodes grows. For instance, with 8 nodes, openGauss achieves a throughput of 131.52 TPS, while PostgreSQL reaches 197.96 TPS. This shows that while openGauss scales reasonably well, PostgreSQL is more efficient in utilizing additional nodes for better throughput.

Additionally, openGauss shows a linear increase in transactions processed as more nodes are added, but its performance still lags behind PostgreSQL at each node count. This suggests that PostgreSQL is better optimized for distributed environments and large-scale deployment scenarios.

When it comes to vertical scaling (i.e., increasing CPU cores and memory), both databases show improvement in performance as resources increase. However, openGauss continues to fall short in comparison to PostgreSQL. For instance, with 8 cores and 4GB of memory, openGauss processes 56.51 TPS, while PostgreSQL manages 119.30 TPS. This indicates that while openGauss does benefit from more resources, it does not scale as effectively as PostgreSQL in resource-constrained environments.

This difference may stem from openGauss's architecture, which, while capable of scaling, may not yet be as finely tuned as PostgreSQL in terms of CPU and memory utilization.

Strengths of openGauss

Scalability: One of the key strengths of openGauss is its scalability. Both horizontal and vertical scaling tests show that openGauss can handle increased workloads effectively as more resources or nodes are added. This makes it an appealing choice for distributed systems where scalability is a primary concern.

Reliability: Across all tests, both databases showed zero failed transactions, demonstrating their reliability and ability to handle workloads without crashing or encountering errors. This ensures that openGauss can be trusted in mission-critical applications, similar to PostgreSQL.

Distributed and Cloud-ready: Given its design around distributed systems, openGauss is better suited for applications that require cloud-native or distributed environments. Its ability to handle large-scale data across multiple nodes could be a decisive factor in such use cases.

Weaknesses of openGauss

Higher Latency: openGauss consistently exhibits higher latency compared to PostgreSQL, particularly in high-load scenarios. This makes openGauss less suitable for applications where fast response times are a priority, such as real-time data processing or interactive applications.

Lower Throughput: In throughput tests, openGauss falls behind PostgreSQL in both single-node and distributed configurations. This indicates that PostgreSQL is better at han-

dling large transaction volumes and can provide more efficient processing in high-demand environments.

Resource Utilization: While `openGauss` benefits from additional CPU cores and memory, it does not take full advantage of these resources compared to `PostgreSQL`, especially in high-performance setups. This suggests that `PostgreSQL` is more effective at utilizing system resources, delivering higher throughput with the same configuration.

Conclusion

`PostgreSQL` is the superior choice in terms of performance and scalability. It consistently outperforms `openGauss` in most key areas, particularly in query response time, throughput, and horizontal scaling. Its long-standing presence in the industry, combined with a mature and well-optimized architecture, makes it the preferred option for high-performance, low-latency environments. Whether it is handling complex queries or managing large datasets, `PostgreSQL` excels in providing reliable, efficient, and fast data processing capabilities, which have been tried and tested over the years across various applications.

On the other hand, `openGauss`, while still trailing behind `PostgreSQL` in terms of raw performance, holds significant promise for scalable, distributed applications. Its design focuses on leveraging the advantages of distributed systems, making it particularly appealing for environments that require horizontal scalability. While `openGauss` may not yet match the transaction throughput or query response times of `PostgreSQL`, its strengths in scalability and reliability allow it to stand out for systems where the ability to handle large-scale, distributed workloads is a higher priority than raw performance.

As the saying goes, "Even a person with a disability, even if sitting in a wheelchair, cannot run faster than a normal person." [Xio24] This perfectly captures the performance gap between `openGauss` and `PostgreSQL`. Despite `openGauss`'s promising features and growing capabilities, it still has a considerable distance to cover before it can match the raw performance and efficiency of `PostgreSQL`. While `openGauss` is not yet a top contender in terms of speed and throughput, it remains a viable option for organizations focused on scalability and distributed architectures.

Ultimately, the choice between `PostgreSQL` and `openGauss` depends largely on the specific needs of the system. If performance and low-latency are paramount, `PostgreSQL` is the clear winner. However, for environments where scalability, distributed workloads, and fault tolerance are more critical, `openGauss` could become a strong contender as it continues to evolve.

Recommendation

`PostgreSQL` is highly recommended for high-performance, low-latency applications, or systems that handle heavy transaction processing. Its mature architecture and optimized performance make it the ideal choice for environments that require fast query responses and

high throughput, such as financial systems, real-time analytics, and large-scale transactional databases.

`openGauss`, on the other hand, is better suited for distributed systems or environments where scalability and reliability are more critical than raw performance. It is a strong candidate for cloud-based applications, big data processing, and enterprise systems where the ability to scale horizontally and handle large datasets is more important than achieving the lowest latency or the highest throughput. While it currently lags behind `PostgreSQL` in these areas, `openGauss` has significant potential to close the performance gap as it continues to evolve.

Task 4

All the testing scripts mentioned in this paper are hosted on the `GitHub` platform. The code repository[Dia24] will be made open-source following the publication of this paper, providing a resource for further discussion and learning. For any questions or inquiries, please feel free to contact the author at 12312420@mail.sustech.edu.cn.

References

[Dia24] Diaosi1317092. Database project, 2024. Accessed: 2024-12-22.

[Doc24] PostgreSQL Documentation. pgbench - postgresql benchmarking tool, 2024. Accessed: 2024-12-22.

[Pro24a] Pgpool-II Project. Pgpool-ii 3.5.4 documentation (chinese version), 2024. Accessed: 2024-12-22.

[Pro24b] Pgpool-II Project. Pgpool-ii 4.5 documentation (english version), 2024. Accessed: 2024-12-22.

[Xio24] Zihao Xiong. Evaluation of opengauss. Personal communication, 2024. Delivered at the College of Engineering.