Server, Storage Engine, Protocol, Client: Suspects in a MySQL Performance Mystery

FOSDEM Database DevRoom 2026

Vitor Oliveira
Huawei Cloud Database Advanced Technology Laboratory

2026/01/31

About me

I'm a Performance Architect with background in database and in HPC performance analysis and optimization.

Huawei

Since 2021 I have worked at Huawei's Cloud Database Advanced Technology Laboratory, Shannon Research Center, based in Trondheim, Norway.

My activity focuses on database optimization for OLTP and HTAP, profiling tools and novel architectures involving persistent memory, low-latency networks and memory fabrics.

Oracle

Prior to that I worked at Oracle's MySQL Replication team since 2014.

There I collaborated in projects like Group Replication and MySQL Database Service (HA) on OCI, and worked closely on features like flow control and WRITESET dependency tracking, among others.

linkedin.com/in/vitor-s-p-oliveira

The Mystery

Porting a new heap table storage engine to **MySQL**, we observed that **the throughput on TPC-C was less than half of GaussDB's,** a PostgreSQL-based database server where it was developed.

My goal today is to guide you through our process of understanding that gap and our efforts to improve MySQL to fill it.

HUAWEI

About TPC-C

TPC-C is a widely recognized benchmark standard for measuring the performance of online transaction processing (OLTP) systems.

It was defined in 1992 by the Transaction Processing Council in https://www.tpc.org/tpcc/.

The benchmark focuses on complex transactions, a mix of five concurrent transaction types:

- NewOrder (45%), Payment (43%), OrderStatus (4%), Delivery (4%) and StockLevel (4%).

tpmC represents the number of NewOrder transactions executed per minute

- not the total number of transactions executed!

HUAWEI

Main Suspects

1. **The Client**
   - Initial tests were performed using BenchmarkSQL on GaussDB and using mysql-tpcc on MySQL, which can create inconsistencies.

2. **The Handler**
   - The handler layer is storage engine specific and maybe is not exploring it effectively.

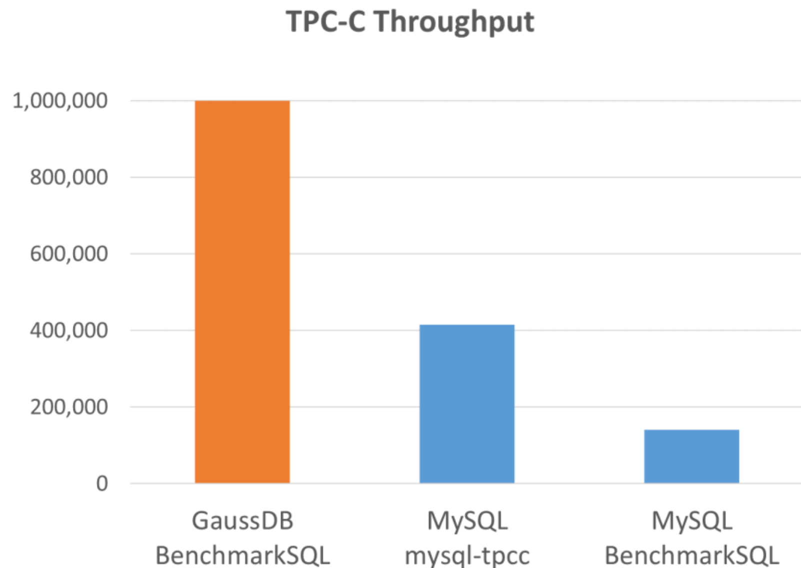
3. **The Server**
   - While both MySQL and PostgreSQL are highly optimized, complex system interactions with the storage engine could still cause unexpected slowdowns..

4. **The Client Protocol**
   - Different databases servers have specific client protocols, and differences in their design can also have impact on the performance.

HUAWEI

The Client

1. The results initially obtained were:
   > BenchmarkSQL on GaussDB: **1M tpmC**
   > mysql-tpcc on MySQL: **415k tpmC**



2. To eliminate the client as a suspect:
   > use the same tool on both database servers!
   > BenchmarkSQL supports both engines and provides the highest performance yet, so its the obvious choice.

3. **Unexpected Result:** BenchmarkSQL only delivered **140k tpmC on MySQL!**

HUAWEI

Optimizer

The query plans selected by the optimizer are one of the most impactful aspects of database performance on complex queries.

1. **Different plans**
   - We compared the plans between databases and MySQL was selecting less optimized plans;
   - After fixing the statistics calculations that went away and plans became mostly similar.

2. **Optimizer bug**
   - We found that the range scan incorrectly ignored the lower bound in some queries:

```
select * from t1, t2 where t1.a=100 and t2.a>=t1.b-20 and t2.a<t1.b;
"Covering index range scan on t2 using a over (NULL < a < 100)"
```

After these improvements, BenchmarkSQL was now at **750k tpmC…** but still 25% away.

HUAWEI

Client variations

We wanted to know how the storage engine was exercised at the server and handler levels.

1. Consistent single-threaded replay

- Extracted the queries generated by BenchmarkSQL to text files;
- Replayed that predictably through the command line on both database engines.
- Surprise: **MySQL now replayed the workload twice faster!**

2. Custom parallel TPC-C client

- Built our own client tool to replicate the TPC-C workload using BenchmarkSQL database;
- Test individual queries in the transactions, not just the full transactions.
- And JDBC was adding a layer of complexity we could do without;
- **And MySQL continues to be faster.**

HUAWEI

Prepared statements

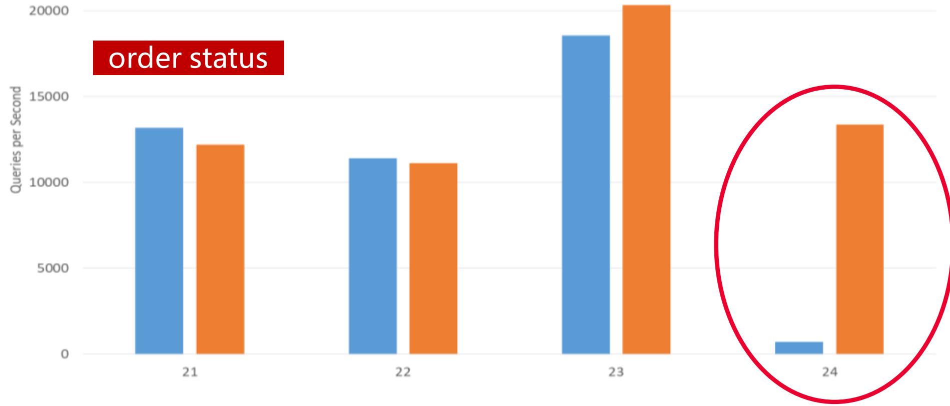
1. A critical insight emerged when we added prepared statements to the tool:
   - prepared statements improved MySQL performance **by about 25%**,
   - but they made GaussDB **140% faster.**

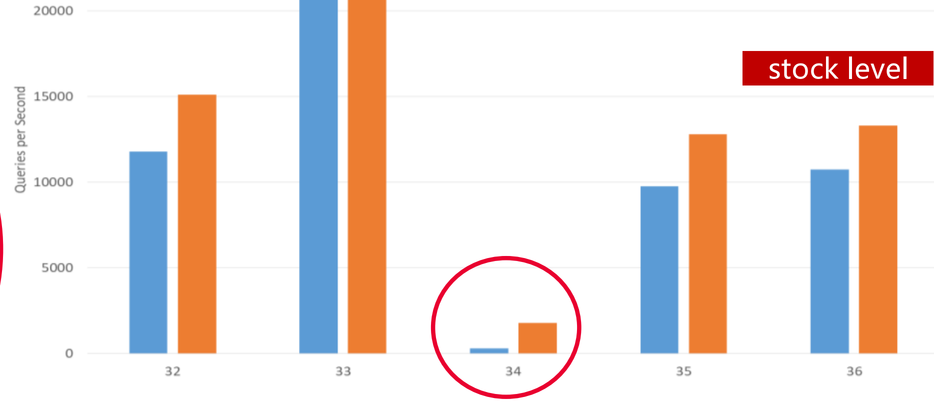
2. **Session plan cache**
   - we introduced the session plan cache in MySQL.
   - when a session uses a prepared statement, selecting from a single table, the plan chosen by the MySQL optimizer is kept in a cache, so that a next execution of the same statement avoids re-planning.

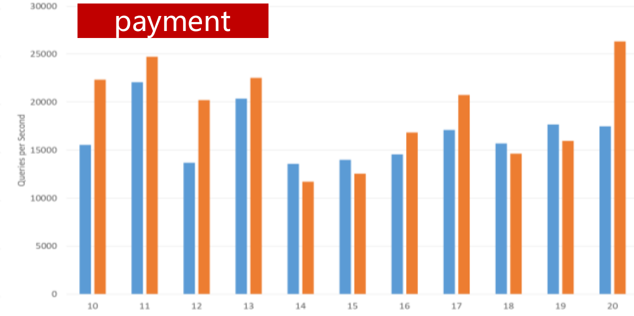
It provided around **5% improvement**

Heap table SE InnoDB

Query Comparison







Heap table differences

The problematic queries demonstrated a fundamental difference between clustered indexes and the heap table indexes. At its root are queries in the format:

```
SELECT count(id) FROM table1 WHERE c_w_id = 12 AND c_d_id= 2 AND c_last='X';
```

Assuming the following indexes:

```
PRIMARY KEY (`c_w_id`,`c_d_id`,`id`),
KEY `idx_customer1` (`c_w_id`,`c_d_id`,`c_last`,`c_first`)
```

1. Additional table lookup

- The id field is not part of the secondary index, but InnoDB includes it anyway.
- The heap table engine does not, so it needs to lookup the primary table for id.

HUAWEI

Heap table differences

1. Change the client query

- The client could avoid COUNT(id) and use COUNT(*) or COUNT(0), since id is known to be NOT NULL (primary key).

or

2. Query rewrite optimization

- "select count(a) from t1 where b>1;" can be rewritten as "select count(0) from t1 where b>1;"
- this allows an index on b to become a covering index, avoiding the lookup entirely.

We implemented this automatic rewrite in MySQL, gaining an additional **10% throughput** and closing part of the remaining gap.

HUAWEI

Partially-consumed resultset

Query 26 was also taking a lot longer to execute on MySQL:

```
SELECT no_o_id FROM new_order WHERE no_d_id=? AND no_w_id=? ORDER BY no_o_id ASC
```

> That was tracked down to the query returning unnecessary rows to BenchmarkSQL;
> This suggested a difference in how GaussDB and MySQL handle result-set streaming.

A quick fix

> Adding LIMIT 1 would prevent unnecessary row transmission, but…
> Testing showed the most effective replacement query to be:

```
SELECT MIN(no_o_id) FROM new_order  WHERE no_d_id = ? AND no_w_id = ?
```

> This removed the overhead seen on Q26, proving **6% improvement** in throughput.

The Client Protocol

A new suspect!

> The differing impact of Q26 on the two databases pointed to a new suspect: the client protocol itself.

> The root cause was traced to the fetchSize parameter in the client protocol.

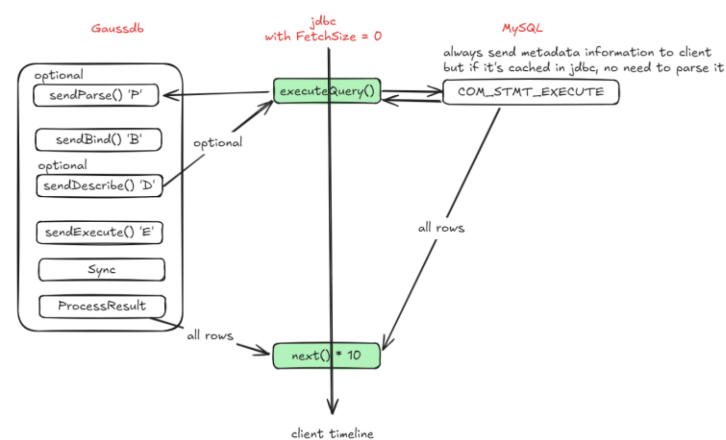
FetchSize

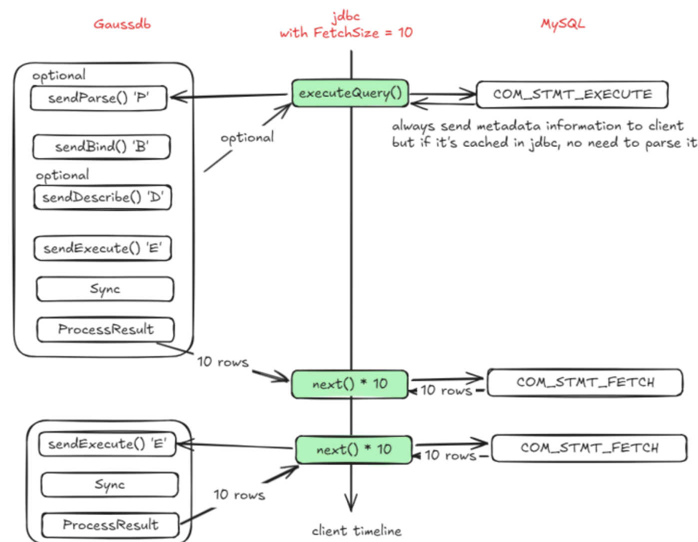
> Defines how many rows to return per request.

> GaussDB: optimal at `fetchSize=10`, slower at 0.

> MySQL: optimal at `fetchSize=0`, slower at 10.

> Finding: MySQL requires an additional server request if fetchSize is non-zero!





The Client Protocol

At this point the GaussDB's performance had also increased to **1.7M tpmC,** and MySQL was still behind it at **1.4M tpmC**.

1. Low-level messaging analysis revealed two issues:

- MySQL sends/receives more messages and overall larger messages than GaussDB;
- And MySQL uses two system calls per message (size + payload), adding some overhead.

2. Bulk prepared statements were handled differently

- BenchmarkSQL used bulk operations (updating/inserting up to 15 rows at once) in the NewOrder transaction,
- However, MySQL lacks native bulk prepared statement support, so the JDBC connector splits each batch into individual queries, increasing round-trips.

The Client Protocol

MySQL's JDBC rewriteBatchedStatements option batches inserts but not updates, requiring another approach to address this.

Change MySQL JDBC connector

- We modified MySQL JDBC connector to also rewrite the bulk update query in NewOrder.
- But generalizing update rewriting in the JDBC connect is complex, error-prone, and a maintenance burden.

or

Add bulk prepared statement support to MySQL

- We also introduced native bulk prepared statement support in MySQL, similar to MariaDB's protocol and JDBC connector.
- This approach showed a slight overhead (3%) compared to the query rewriting alternative.

HUAWEI

Conclusion

With all the improvements mentioned, **MySQL now reached 1.7M tpmC**, finally matching the GaussDB!

With additional optimizations like PGO/LTO, it was eventually able to reach over 2M tpmC.

Final thought

This study underscores a critical lesson: database performance, for OLTP workloads in particular, is determined not by any single component, but by the precise alignment of the entire database stack, from the client down to the storage engine.

HUAWEI

Thank you.

Bring digital to every person, home and organization for a fully connected, intelligent world.

Copyright©2018 Huawei Technologies Co., Ltd.
All Rights Reserved.

The information in this document may contain predictive statements including, without limitation, statements regarding the future financial and operating results, future product portfolio, new technology, etc. There are a number of factors that could cause actual results and developments to differ materially from those expressed or implied in the predictive statements. Therefore, such information is provided for reference purpose only and constitutes neither an offer nor an acceptance. Huawei may change the information at any time without notice.

Send (+) / Received (-)
Exec time
Sends
Receives

Sends: 28

Transaction latency 2.1 ms

Total bytes sent: 3172

Receives: 28

Total bytes received: 8266

Average wait 38 us

Median wait 31 us

Max wait 388 us

Sent/Received (bytes)

Execution time (us)

10,000
7,500
5,000
2,500
0
-2,500
-5,000
-7,500
-10,000
-12,500

2,250
2,000
1,750
1,500
1,250
1,000
750
500
250
0

GaussDB

MySQL

Send (+) / Received (-)
Exec time
Sends
Receives

Sends: 33

Transaction latency 3.8 ms

Total bytes sent: 6688

Receives: 84

Total bytes received: 22823

Average wait 32 us

Median wait 19 us

Max wait 178 us

Sent/Received (bytes)

Execution time (us)

10,000
7,500
5,000
2,500
0
-2,500
-5,000
-7,500
-10,000
-12,500

2,250
2,000
1,750
1,500
1,250
1,000
750
500
250
0