Table of Contents

Preface i

Who Should Read This Book? i

Outline of This Book ii

References and Further Reading iii

Special Terminology Explanation iii

Deployment Supplementary Explanation iii

How to Contact Us iv

Acknowledgments iv

Part1 Issues 1

Chapter 1: Traditional Methods for Resolving MySQL Issues 1

1.1 How to Solve MySQL Problems? 1

1.2 How Are MySQL Problems Solved? 1

1.2.1 Addressing Scalability Issues in MySQL 5.7 1

1.2.2 MySQL Joins Struggle to Meet User Demands 3

1.2.3 Challenges of MySQL Semisynchronous Replication in Large-Scale Deployments 3

1.2.4 Majority-Based Mechanism Performance and Its Accuracy 4

1.2.5 Group Replication: Localhost Deployment Reports Unreachable 6

1.3 Summary 8

Chapter 2: Mysterious MySQL Issues 9

2.1 SysBench Read-Write Test Exhibits Super-Linear Throughput Growth 9

2.2 Post-PGO, TPC-C Throughput Declines Instead 11

2.3 Unexpected Negative Effects of Thread Pool After MySQL Scalability Improvements 12

2.4 In MySQL 8.0, TPC-C Throughput Drops Too Quickly 13

2.5 Repeatable Read Surprisingly Outperforms Read Committed 15

2.6 Group Replication Throughput Lower Than Semisynchronous Replication 17

2.7 Modified Group Replication Outperforms Semisynchronous Replication 18

2.8 SysBench Shows No Effect, TPC-C Performs Well 20

2.9 Is Disabling NUMA Really Beneficial for MySQL? 23

2.10 Summary 24

Part2 Basics 25

Chapter 3: How to Solve Software Problems Effectively 25

3.1 Analysis Strategy 25

3.1.1 Psychological Strategies 25

3.1.2 Simplification Strategies 25

3.1.3 Pattern-finding Strategies 26

3.1.4 Strategies to Increase Reproducibility 26

3.1.5 Strategies to Find Key Evidence 27

3.2 Logical Thinking 28

3.2.1 Deductive Reasoning 29

3.2.2 Inductive Reasoning 29

3.2.3 Abductive Reasoning 30

3.2.4 Reductio ad Absurdum 30

3.2.5 Utilizing Various Methods Comprehensively 30

3.3 Tactics for Solving Problems 30

3.3.1 Balancing Different Options 31

3.3.2 Decompose the Problem 31

3.3.3 Seeking Theoretical Support 32

3.3.4 Logic-based Testing 33

3.4 Principles of Logical Reasoning 34

3.5 Logical Reasoning: Key to Solving Complex Problems 35

3.6 The Difference Between Solving Difficult Problems and Doing Exercises 35

3.7 Common Problem-Solving Frameworks 36

Chapter 4: Fundamentals of Computer Science 38

4.1 System Architecture 38

4.1.1 SMP 38

4.1.2 NUMA 39

4.1.3 X86 Architecture 42

4.1.4 ARM Architecture 43

4.2 Data Structure 45

4.2.1 Array 45

4.2.2 Linked List 48

4.2.3 Queue 49

4.2.4 Heap 50

4.2.5 Hash Table 50

4.2.6 Red-Black Tree 52

4.2.7 B+ Tree 54

4.2.8 Hybrid Data Structure 54

4.3 Algorithm 56

4.3.1 Search Algorithm 56

4.3.2 Sorting Algorithm 58

4.3.3 Greedy Algorithm 61

4.3.4 Dynamic Programming 62

4.3.5 Amortized Analysis 62

4.3.6 Paxos 64

4.4 Operating system 65

4.4.1 CPU Scheduling 66

4.4.2 Process Model 70

4.4.3 Thread Model 70

4.4.4 Staged Model 71

4.4.5 Memory Allocation 73

4.4.6 System Call 75

4.5 Compiler Theory 75

4.5.1 Finite State Machine 75

4.5.2 MySQL Parser 76

4.5.3 PGO 76

4.6 Queueing Theory 77

4.6.1 Single Queue Bottleneck 77

4.6.2 Multiple Queue Bottlenecks 79

4.6.3 The Mutual Influence Between Different Queue Bottlenecks 80

4.6.4 The Relationship Between Resource Utilization and Response Time 84

4.6.5 Transaction Throttling Mechanism 85

4.7 Computer Networking 86

4.7.1 FLP Impossibility Theory 86

4.7.2 TCP/IP Protocol Stack 87

4.7.3 TCP State Machine 88

4.7.4 Will Data Content Be Corrupted During Network Transmission? 90

4.7.5 Batching and Pipelining 90

4.7.6 Impact of Network Latency on Performance 94

4.7.7 Network Partition 95

4.7.8 RDMA 96

4.7.9 The Importance of Packet Capture Information 96

4.7.10 Logical Reasoning in Network Issues 97

4.8 Performance Optimization 100

4.8.1 Performance Optimization Flowchart 101

4.8.2 Throughput vs. Response Time 102

4.8.3 Amdahl's Law 104

4.8.4 Performance Modeling 107

4.8.5 Challenges in the Limitations of Performance Analysis Tools 108

4.8.6 Mitigating Scalability Issues 109

4.8.7 Optimize Response Time 112

4.8.8 Tail Behavior of Response Time: Challenges for SLAs 116

4.8.9 Performance Issues in NUMA Systems 116

4.8.10 Performance Improvement is Not Purely Additive 116

4.9 Distributed Theory 118

4.9.1 CAP Theorem 118

4.9.2 Read Consistency 120

4.9.3 Consensus 121

4.9.4 Distributed Transaction 122

4.10 Database Fundamentals 122

4.10.1 Relational Model and SQL Language 122

4.10.2 Database Security 123

4.10.3 Database Integrity 123

4.10.4 SQL Rewrite 123

4.10.5 SQL Injection 124

4.10.6 The Significant Impact of Indexes on Performance 124

4.10.7 State-Machine Replication 125

4.11 Software Architecture Design 125

4.11.1 Layered Architecture 125

4.11.2 Primary-Secondary/Multi-Primary Architecture 127

4.11.3 Event-Driven Architecture 130

4.12 Software Testing 131

4.12.1 The Importance of Testing 131

4.12.2 Balancing Test Cases and Development Efficiency 131

4.12.3 Ensuring Consistency in the Testing Environment 131

4.12.4 How to Test Efficiently? 132

4.12.5 Is Testing About Discovering Issues or Verifying Them? 132

Chapter 5: MySQL Internals 133

5.1 The "Storage Stack" of InnoDB 133

5.2 Transactions 134

5.3 Concurrency Control 135

5.4 Transaction Isolation Level 136

5.5 MVCC 137

5.6 InnoDB Architecture 137

5.7 Log Manager 140

5.8 Lock Scheduling Algorithms 140

5.9 Binlog File 141

5.10 Group Commit Mechanism 141

5.11 Execution Plan 142

5.12 Partitioning 143

5.13 Coordination Avoidance 144

5.14 Disaster Recovery 144

5.15 Idempotence 144

5.16 Thread Pool 145

5.17 Traditional Cluster 146

5.17.1 Asynchronous Replication 146

5.17.2 Semisynchronous Replication 147

5.17.3 How Scalable is Semisynchronous Replication? 148

5.18 Group Replication 149

5.18.1 Why Implement Group Replication? 150

5.18.2 Why Was Mencius Initially Adopted? 150

5.18.3 Why Introduce the Single-Leader Multi-Paxos Algorithm? 150

5.18.4 Does Group Replication Lose Data? 151

5.18.5 Will Group Replication Outperform Semisynchronous Replication? 151

5.18.6 Is a Single Thread Sufficient for Underlying Paxos Communication? 151

5.18.7 Paxos Single Leader vs. Group Replication Single-Primary Mode 151

5.18.8 Is Single Leader Multi-Paxos Universally Applicable in Single-Primary Mode? 152

5.19 MySQL Secondary Replay 154

5.19.1 Introduction to MySQL Secondary Replay 154

5.19.2 The Difference Between Replay and Transaction Execution 154

5.19.3 The Role of Speeding Up MySQL Secondary Replay 154

5.19.4 The Architecture of MySQL Secondary Replay 155

5.20 The Integration of AI with Databases 157

5.20.1 Learning-based Database Configuration 157

5.20.2 Learning-based Database Optimization 158

5.20.3 Learning-based Database Design 158

5.20.4 Learning-based Database Monitoring 159

5.20.5 Learning-based Database Security 159

5.20.6 Performance Prediction 159

5.20.7 AI Challenges 159

5.20.8 AI Summary 160

5.21 How MySQL Internals Work in a Pipeline Fashion? 160

5.22 Why MySQL Needs to Support High Concurrency? 161

5.23 Scalability of MySQL Clusters 163

5.24 MySQL Problem Diagnosis 166

5.25 The Significant Differences Between BenchmarkSQL and SysBench 167

Chapter 6: How to Scientifically Test MySQL Performance? 170

6.1 Common Pitfalls 170

6.1.1 Non-Reproducibility 170

6.1.2 Failure To Optimize 171

6.1.3 Overly-specific Tuning 172

6.1.4 Cold vs. Hot Runs 173

6.1.5 Cold vs. Warm Runs 173

6.1.6 Human Error 174

6.2 Comprehensive Testing 175

6.2.1 Testing with Multiple Tools 175

6.2.2 Identifying Contention-Intensive Tests 175

6.2.3 Understanding TPC-C Testing Characteristics 175

6.2.4 Incorporating Thinking Time in Testing 177

6.2.5 Impact of I/O on Testing 178

6.2.6 Long-Term Stability Testing 179

6.2.7 Online Traffic Testing 181

Part3 Practice 182

Chapter 7: Key Improvements of MySQL 8.0 Over MySQL 5.7 182

7.1 Scaling Up: InnoDB Improvements 182

7.1.1 Redo Log Optimization 182

7.1.2 Optimizing Lock-Sys Through Latch Sharding 188

7.1.3 Latch Splitting in trx-sys 190

7.1.4 Latch Sharding for trx-sys 191

7.1.5 Summary 192

7.2 Evaluating Performance Gains in MySQL Lock Scheduling Algorithms 192

7.3 Enhancements in MySQL Execution Plans 202

7.3.1 Hash Join Implementation in MySQL 202

7.3.2 Introduction of Hypergraph Algorithm in MySQL 203

7.4 Cost Savings with Binlog Compression 206

References 210

Appendix 219

Glossary 219

1 NUMA 219

2 PGO 219

3 Fault Tolerance 219

4 High Availability 219

5 Transaction 220

6 Dual One 220

7 Replication 220

8 Paxos Algorithm 220

9 State Machine Replication 220

10 Row-based Binlog 220

11 Read Committed Isolation Level 221

12 Thread Pool 221

13 OLTP 221

14 Network Latency 221

15 Response Time 221

16 AI 221

17 TPC-C 222

18 Throughput 222

19 Idempotence 222

20 Crash Recovery 222

21 GTID 223

22 Latch vs. Lock: Key Differences 223

23 MVCC 225

24 Doublewrite 225

25 Causality 226

26 Maintaining Transaction Order with replica_preserve_commit_order 226

27 Pipelining 226

28 Achieving Strong Consistency in Read/Write Operations 226

29 View Change 227

30 Network Partition 227

31 Transaction Throttling 227

32 Data Structure 227

33 Algorithms 227

34 Thundering Herd 227

35 Balanced Replay Speed 228

Testing Tool 228

1 SysBench 228

2 TPC-C Testing Tool 228

How MySQL Processes SQL? 229

MySQL Architecture 234

1 Client 234

2 Server 235

MySQL Cluster 235

Testing Related Materials 237

1 Hardware Configurations 237

2 Operating System 238

3 BenchmarkSQL Script 238

4 SysBench Script 238

5 Using the tpcc-mysql Script for Benchmarking 239

6 Configuration Parameters 239

7 Source Code Repository 242

About the Author 242

Preface

Who Should Read This Book?

When software professionals encounter complex problems, they often face contradictions and struggle to find solutions. To overcome these challenges, it's crucial to view tough problems as opportunities for significant rewards. Embracing difficult challenges can lead to valuable insights and breakthroughs.

With 20 years of problem-solving experience, most challenging issues in server-side software can be resolved through logical analysis. Despite its global prominence, MySQL faces a variety of problems. The journey in addressing these MySQL issues has enriched understanding and led to the development of unique insights, which inspired the writing of this book.

This book uses MySQL challenges as case studies to explore problem analysis and resolution strategies. Readers will gain a deeper appreciation for logical reasoning, data structures, algorithms, and more through practical examples and insightful discussions.

This book covers the following topics:

1. **Logical Reasoning for MySQL**: Techniques for analyzing complex MySQL issues using logical reasoning.
2. **Computer Science Fundamentals**: Key principles of computer science relevant to MySQL.
3. **MySQL Internals**: An introduction to the basics of MySQL kernels.
4. **Performance Testing**: Methods for scientifically testing MySQL performance.
5. **MySQL 8.0 Enhancements**: Significant improvements in MySQL 8.0 compared to MySQL 5.7.
6. **MySQL Improvements**: Further optimizations in a standalone MySQL instance.
7. **Group Replication Advancements**: Notable enhancements in MySQL Group Replication.
8. **Secondary Replay Enhancements**: Optimizations in MySQL's secondary replay.
9. **Performance Optimization Techniques**: Strategies for optimizing MySQL applications.

Focus of the Book:

Building on the goal of enhancing MySQL scalability, this book explores various optimization strategies to increase throughput, reduce response times, and ensure that MySQL secondaries closely match the performance of the primary server, thereby achieving high availability failover with very little delay in most scenarios.

Target Audience:

1. **Quality-Focused Developers**: Individuals committed to high-quality software development.
2. **Problem Solvers**: Those interested in tackling complex software problems.
3. **MySQL Enthusiasts**: Readers looking to understand MySQL from the ground up.
4. **Testing Practitioners**: Users keen on mastering effective testing techniques.
5. **Software Architects**: Those designing large-scale software systems.
6. **Performance Engineers**: Engineers struggling with performance optimization.
7. **Computer Science Students**: Learners aiming to strengthen their foundational knowledge.
8. **MySQL Researchers**: Researchers conducting studies based on MySQL.

Outline of This Book

Chapter 1 explores how users approach and solve challenging MySQL problems. Chapter 2 focuses on mysterious issues in MySQL 8.0.

Chapter 3 emphasizes the importance of logical reasoning skills in addressing MySQL challenges effectively. Chapter 4 provides practical computer fundamentals essential for understanding MySQL issues. Chapter 5 introduces MySQL kernel basics, laying the groundwork for subsequent problem-solving discussions. Chapter 6 covers scientific methods for testing MySQL.

Chapter 7 highlights notable optimizations in MySQL 8.0. Chapter 8 examines improvements for resolving issues specific to MySQL 8.0. Chapter 9 discusses advancements in MySQL Group Replication, while Chapter 10 addresses enhancements in MySQL secondary replay. Together, these chapters not only boost MySQL performance but also lay the foundation for achieving high availability clusters.

Chapter 11 presents techniques to enhance MySQL 8.0 application performance without code changes.

Finally, Chapter 12 outlines future directions for MySQL improvements.

References and Further Reading

This book focuses on analyzing and solving MySQL problems, so a certain level of computer science background is recommended. To support understanding and maintain continuity, key terminology is included in the "Glossary" section of the appendix. For those lacking a foundation in MySQL, please refer to the related content in the appendix or consult dedicated MySQL books.

Special Terminology Explanation

1. **MySQL Secondary Replay**

   This term represents the common replay process for asynchronous replication, semisynchronous replication, and Group Replication. In this book, 'MySQL primary' is consistently used to represent the MySQL source, and 'MySQL secondary' to represent the MySQL replica.

2. **Dual One**

   Specifically refers to two major configurations in MySQL: *sync_binlog=1* and *innodb_flush_log_at_trx_commit=1*.

3. **Balanced Replay Speed**

   For MySQL secondary replay, this defines balanced replay speed where the MySQL secondary matches the primary under normal circumstances. When the speed is at or below the balanced replay speed, there is no significant lag in transaction replay progress on the secondary. However, if the speed exceeds this threshold, the secondary begins to lag behind the primary in transaction replay progress.

Deployment Supplementary Explanation

When deploying MySQL for testing, it is preferable to match the test environment as closely as possible to the production environment, unless certain configurations significantly interfere with performance analysis. Below are specific declarations related to MySQL deployment during the testing process:

1. Mainstream servers used are all NUMA architecture servers.
2. Servers are generally x86 architecture.

3. SSD hardware disks are used.
4. All tests are conducted on the Linux operating system.
5. MySQL standalone tests typically use version 8.0.27, while MySQL cluster versions generally use version 8.0.32.
6. Improvements to MySQL are referred to as improved MySQL or modified MySQL.
7. The transaction isolation level in TPC-C tests is *Read Committed*.
8. The storage engine used for transactions is InnoDB.
9. binlog_format is set to *row-based* format.
10. MySQL, whether primary or secondary, uses GTID (Global Transaction ID).
11. Cluster settings include *replica_preserve_commit_order=on*.
12. Unless stated otherwise, TPC-C tests are generally based on partitioned large tables.

How to Contact Us

For configurations, patches, and additional information related to this book, please visit our webpage: https://github.com/advancedmysql/.

Acknowledgments

This book meticulously organizes a wealth of ideas and knowledge contributed by numerous individuals, encompassing insights from both academic research and industrial practice. In computing, there is a natural inclination towards the latest innovations, yet our historical foundations offer valuable lessons. With over 100 references to articles, blog posts, documentation, and other resources, this book has been an indispensable source of learning for me. I am profoundly grateful to the authors for their generous contributions of knowledge.

Several individuals have been instrumental in the writing of this book by reviewing drafts and offering feedback. I am especially grateful for the contributions of Hongshen Wang, Jinrong Ye, Riyao Gao, and Haitao Gao. Naturally, I take full responsibility for any remaining errors or contentious opinions in this book.

Finally, my deepest gratitude to my family, whose unwavering support has been indispensable throughout this nearly six-month writing journey. You're the best.

Part1 Issues

Chapter 1: Traditional Methods for Resolving MySQL Issues

1.1 How to Solve MySQL Problems?

Resolving issues related to MySQL usage is generally straightforward and involves gathering information and applying logical reasoning, making these problems typically resolvable. However, tackling inherent issues within MySQL itself proves significantly more complex. Fortunately, MySQL users are resourceful and have devised various solutions, such as implementing thread pools to mitigate scalability challenges.

To effectively resolve the myriad issues of MySQL, relying solely on peripheral fixes and patches is inadequate. It's essential to delve into the core nature of these problems and comprehensively address them to achieve meaningful progress. For example, MySQL Group Replication has persistently struggled with instability. Despite robust development efforts, the ongoing challenge lies in adopting the correct problem-solving approach, resulting in difficulties in reducing bugs.

1.2 How Are MySQL Problems Solved?

Here are some classic issues listed, along with typical resolutions in real-world MySQL usage scenarios.

1.2.1 Addressing Scalability Issues in MySQL 5.7

The following figure illustrates the relationship between TPC-C throughput and concurrency in MySQL 5.7.39 under a specific configuration. This includes setting the transaction isolation level to Read Committed and adjusting the ***innodb_spin_wait_delay*** parameter to mitigate throughput degradation.

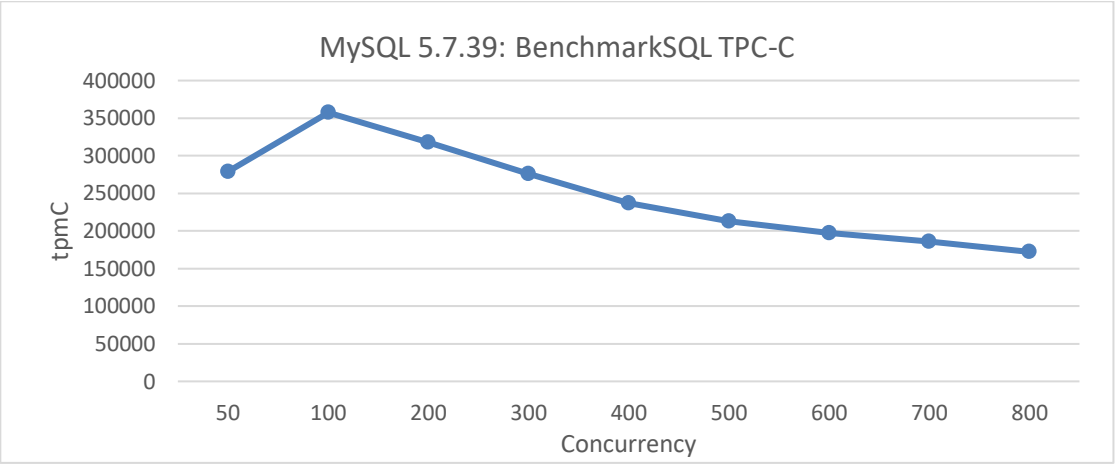


Figure 1-1. Scalability issues in MySQL 5.7.39 during BenchmarkSQL testing.

From the figure, it is evident that scalability issues significantly limit the increase in MySQL throughput. For example, after 100 concurrency, the throughput begins to decline. Due to MySQL's historical scalability challenges, Percona even open-sourced a thread pool to address these issues. The following figure illustrates the relationship between TPC-C throughput and concurrency after configuring the Percona thread pool.

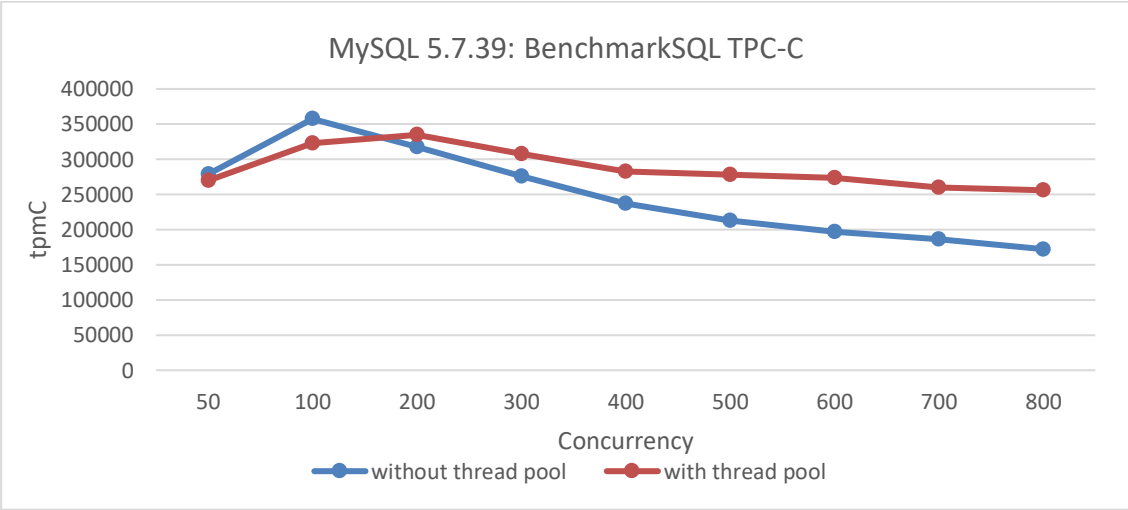


Figure 1-2. Percona thread pool mitigates scalability issues in MySQL 5.7.39.

For MySQL 5.7.39, the thread pool effectively mitigates the decline in throughput degradation, although it also slightly reduces peak throughput due to inherent overhead costs. This overhead becomes evident when comparing performance at lower concurrency levels.

As MySQL 8.0 versions improve scalability, the effectiveness of the thread pool in mitigating throughput degradation diminishes. Subsequent chapters will provide real-world cases to substantiate this observation.

1.2.2 MySQL Joins Struggle to Meet User Demands

Due to the absence of hash join support in MySQL 5.7, the following statistical SQL query took a prolonged 3.82 seconds to execute.

```
mysql> explain SELECT count(*) FROM bmsql_district JOIN bmsql_order_line ON ol_d_id = d_id;
+----+-------------+----------------+------------+-------+-----------------------+-----------------------+---------+-------------------------------------+--------+----------+-------------+
| id | select_type | table          | partitions | type  | possible_keys         | key                   | key_len | ref                                 | rows   | filtered | Extra       |
+----+-------------+----------------+------------+-------+-----------------------+-----------------------+---------+-------------------------------------+--------+----------+-------------+
|  1 | SIMPLE      | bmsql_district | NULL       | index | NULL                  | PRIMARY               | 8       | NULL                                |    100 |   100.00 | Using index |
|  1 | SIMPLE      | bmsql_order_line | NULL     | ref   | bmsql_order_line_index | bmsql_order_line_index | 4      | benchmarksql.bmsql_district.d_id    | 331589 |   100.00 | Using index |
+----+-------------+----------------+------------+-------+-----------------------+-----------------------+---------+-------------------------------------+--------+----------+-------------+
2 rows in set, 1 warning (0.00 sec)

mysql> SELECT count(*) FROM bmsql_district JOIN bmsql_order_line ON ol_d_id = d_id;
+----------+
| count(*) |
+----------+
| 29971910 |
+----------+
1 row in set (3.82 sec)
```

Figure 1-3. Non-hash join performance in MySQL 5.7.

In MySQL 8.0, the introduction of hash joins in the execution plan has enhanced join performance. For the same SQL query mentioned earlier, specifying hash join through comments reduced the execution time to just 1.22 seconds, a significant improvement from the previously observed 3.82 seconds.

```
mysql> explain SELECT /*+ JOIN_ORDER(bmsql_order_line, bmsql_district) */  count(*) FROM bmsql_district JOIN bmsql_order_line ON ol_d_id = d_id;
+----+-------------+----------------+------------+-------+-----------------------+-----------------------+---------+------+---------+----------+-------------------------------------------------------------------+
| id | select_type | table          | partitions | type  | possible_keys         | key                   | key_len | ref  | rows    | filtered | Extra                                                             |
+----+-------------+----------------+------------+-------+-----------------------+-----------------------+---------+------+---------+----------+-------------------------------------------------------------------+
|  1 | SIMPLE      | bmsql_order_line | NULL     | index | bmsql_order_line_index | bmsql_order_line_index | 4      | NULL | 2984302 |   100.00 | Using index                                                       |
|  1 | SIMPLE      | bmsql_district | NULL       | index | NULL                  | PRIMARY               | 8       | NULL |     100 |    10.00 | Using where; Using index; Using join buffer (hash join)           |
+----+-------------+----------------+------------+-------+-----------------------+-----------------------+---------+------+---------+----------+-------------------------------------------------------------------+
2 rows in set, 1 warning (0.00 sec)

mysql> SELECT /*+ JOIN_ORDER(bmsql_order_line, bmsql_district) */  count(*) FROM bmsql_district JOIN bmsql_order_line ON ol_d_id = d_id;
+----------+
| count(*) |
+----------+
| 29971910 |
+----------+
1 row in set (1.22 sec)
```

Figure 1-4. Hash join performance in MySQL 8.0.

This illustrates the enhancements in join execution introduced in MySQL 8.0, highlighting one of its key improvements.

1.2.3 Challenges of MySQL Semisynchronous Replication in Large-Scale Deployments

Since the introduction of the Paxos protocol, an increasing number of databases have adopted state machine replication to construct highly available clusters.

The Paxos protocol relies on a majority-based mechanism rooted in set theory to achieve consensus. For instance, in a cluster with 3 MySQL nodes, 2 nodes can form a majority. If these 2 nodes reach

consensus, the system can operate continuously. Theoretically, adherence to this majority-based mechanism should prevent divergence or split-brain issues under any anomaly.

Raft, another widely used protocol, is essentially a simplified version of the Paxos protocol. The following section describes Meta's implementation of MySQL high availability using the Raft protocol [71], which simplifies traditional high availability processes.

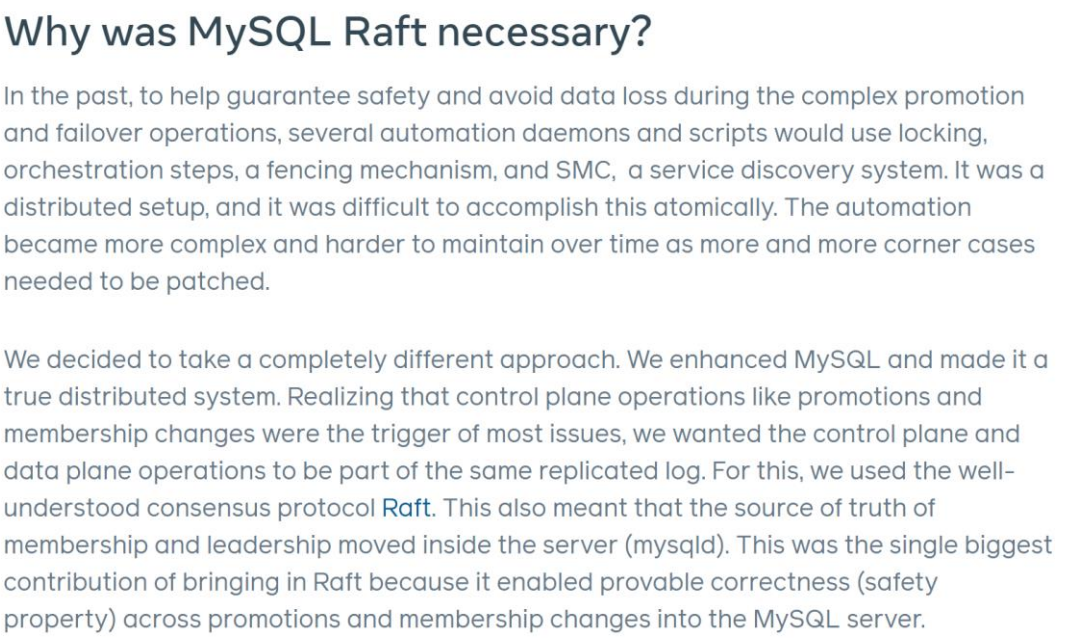


Figure 1-5. Reasons Meta uses raft.

Meta's case study demonstrates that the new solution based on the Raft protocol simplifies high availability challenges. Raft is often viewed as a streamlined version of the Paxos protocol. Similarly, a Group Replication solution based on the Paxos protocol, if implemented correctly, could also offer an elegant solution.

1.2.4 Majority-Based Mechanism Performance and Its Accuracy

With the transaction isolation level set to Read Committed, simulations based on Group Replication were conducted under various network latency conditions.

The deployment setup of Group Replication is illustrated as follows: On machine A, two MySQL instances are deployed—one serving as the primary and the other as the secondary. These two

instances form the majority and communicate via localhost. Machine B hosts a third instance deployed as a member of the cluster, with a network latency of X milliseconds.

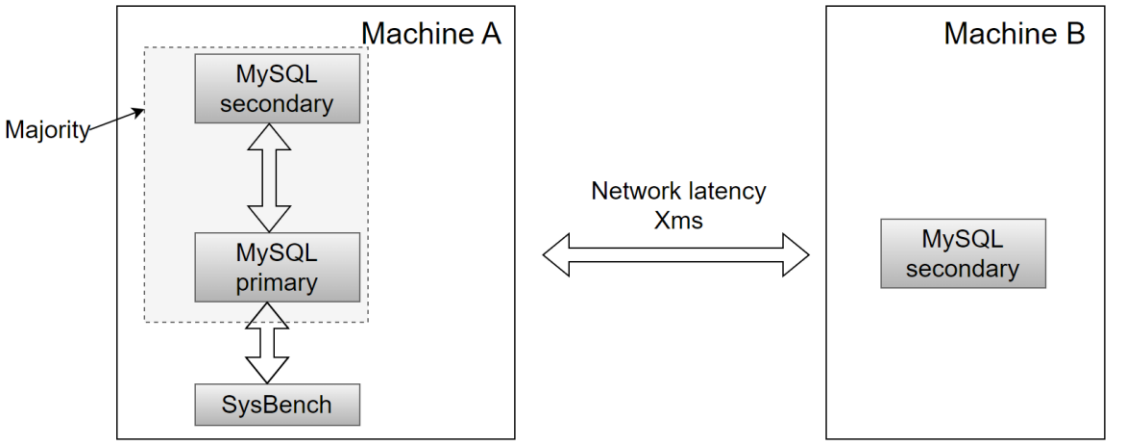


Figure 1-6. Deployment diagram for majority-based mechanism performance testing.

In theory, with a majority-based mechanism, a cluster of 3 nodes only needs responses from 2 nodes to provide results to the client. By this logic, local SysBench tests should demonstrate very high efficiency.

Throughput comparisons over time have been conducted for machine B in scenarios within the same data center and across data centers with latencies of 10ms, 100ms, and 1000ms. Specific results are illustrated in the following figure.

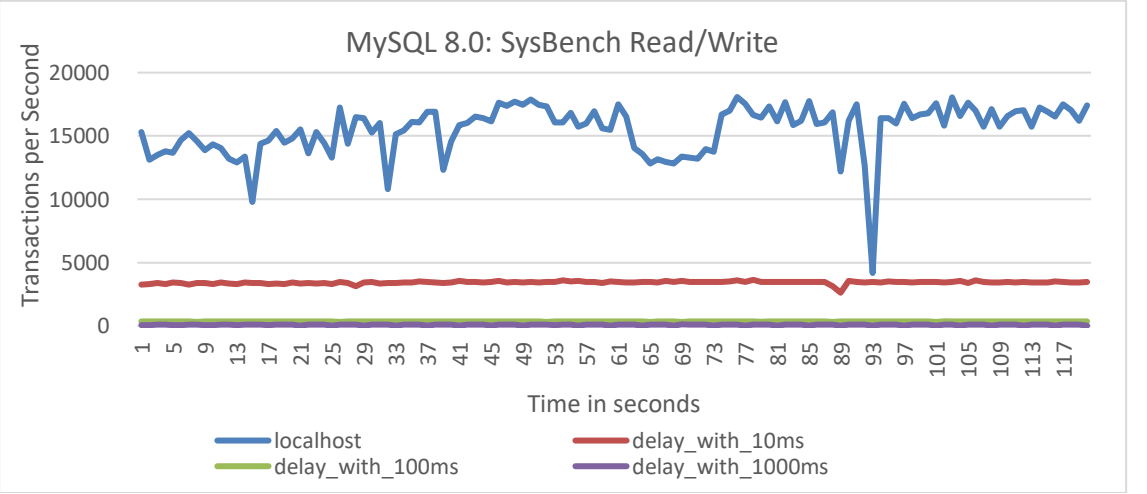


Figure 1-7. Performance testing results of the default multi-leader Paxos algorithm.

From the figure, it is evident that under the default mode of Group Replication, throughput across data centers deviates significantly from theoretical expectations. For example, in scenarios with 10ms network latency, the cluster's throughput decreases to one-fifth of its original level.

To address this discrepancy, starting from MySQL 8.0.27, the ***group_replication_paxos_single_leader*** option was introduced. Enabling this option utilizes the single leader Paxos algorithm instead of the default multi-leader Paxos algorithm.

After configuring the system to use the single leader Paxos algorithm, tests were conducted under the same conditions using the SysBench testing tool. The test results are as follows.

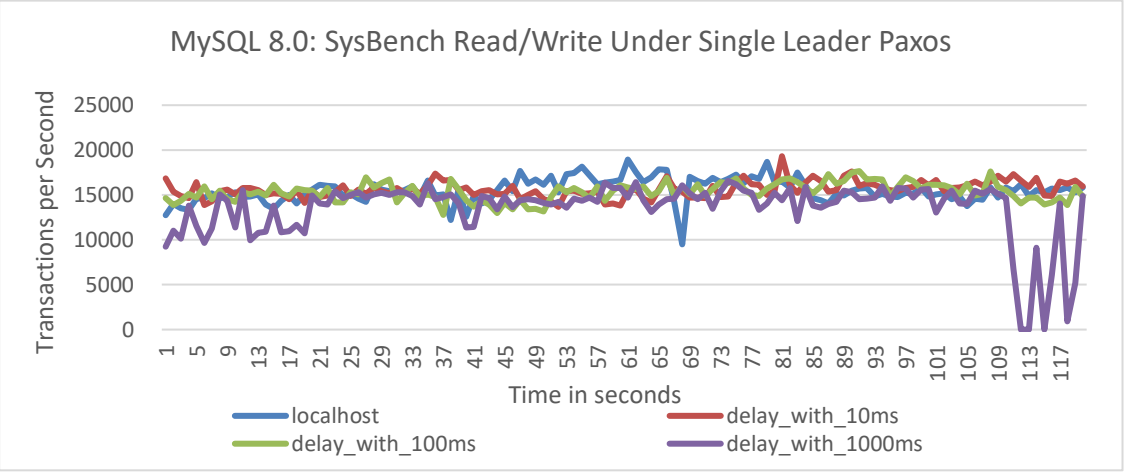


Figure 1-8. Performance testing results of the single-leader Paxos algorithm.

From the figure, it is evident that the results of the single leader mode are significantly better than previous results. However, the drawback is that the single leader mode can only function in MySQL's single-primary mode, and its application scope is limited, unable to be applied in scenarios requiring strong consistency for read and write operations (before/after).

Is this modification to the underlying Paxos algorithm truly the optimal solution? This will be explored in depth in the following chapters.

1.2.5 Group Replication: Localhost Deployment Reports Unreachable

Deploying Group Replication with two nodes on the same machine, where communication between these nodes occurs via localhost, theoretically should not encounter network-related jitter, packet loss, or issues with unreachable peers under normal conditions.

```
mysql> select * from performance_schema.replication_group_members;
+---------------------------+--------------------------------------+-------------+-------------+--------------+-------------+----------------+----------------------------+
| CHANNEL_NAME              | MEMBER_ID                            | MEMBER_HOST | MEMBER_PORT | MEMBER_STATE | MEMBER_ROLE | MEMBER_VERSION | MEMBER_COMMUNICATION_STACK |
+---------------------------+--------------------------------------+-------------+-------------+--------------+-------------+----------------+----------------------------+
| group_replication_applier | 5bde79ee-10f5-11ef-8984-5c6f691467c0 | 127.0.0.1   |       63306 | ONLINE       | PRIMARY     | 8.0.27         | XCom                       |
| group_replication_applier | 7af985d4-10f5-11ef-a33a-5c6f691467c0 | 127.0.0.1   |       53306 | ONLINE       | SECONDARY   | 8.0.27         | XCom                       |
+---------------------------+--------------------------------------+-------------+-------------+--------------+-------------+----------------+----------------------------+
```

Figure 1-9. Localhost deployment relationships displayed by Performance Schema.

During TPC-C data loading tests with BenchmarkSQL under normal pressure, the MySQL error logs revealed multiple instances of the primary and secondary nodes reporting each other as unreachable.

Here is a partial screenshot of the error log from the MySQL primary:

```
2024-05-13T18:37:50.430183+08:00 0 [Note] [MY-011953] [InnoDB] Page cleaner took 7571ms to flush 2000 pages
2024-05-13T18:41:56.301804+08:00 0 [Warning] [MY-011493] [Repl] Plugin group_replication reported: 'Member with address 127.0.0.1:53306 has become unreachable.'
2024-05-13T18:41:56.302314+08:00 0 [ERROR] [MY-011495] [Repl] Plugin group_replication reported: 'This server is not able to reach a majority of members in the group. This server
 will now block all updates. The server will remain blocked until contact with the majority is restored. It is possible to use group_replication_force_members to force a new grou
p membership.'
2024-05-13T18:41:57.827165+08:00 0 [Warning] [MY-011735] [Repl] Plugin group_replication reported: '[GCS] Shutting down an outgoing connection. This happens because something mig
ht be wrong on a bi-directional connection to node 127.0.0.1:53318. Please check the connection status to this member'
2024-05-13T18:41:57.866106+08:00 0 [Note] [MY-011735] [Repl] Plugin group_replication reported: '[GCS] Connecting to 127.0.0.1:53318'
2024-05-13T18:41:57.868061+08:00 0 [Note] [MY-011735] [Repl] Plugin group_replication reported: '[GCS] Connected to 127.0.0.1:53318'
2024-05-13T18:41:57.868812+08:00 0 [Warning] [MY-011494] [Repl] Plugin group_replication reported: 'Member with address 127.0.0.1:53306 is reachable again.'
2024-05-13T18:41:57.869080+08:00 0 [Warning] [MY-011498] [Repl] Plugin group_replication reported: 'The member has resumed contact with a majority of the members in the group. Re
gular operation is restored and transactions are unblocked.'
2024-05-13T18:42:58.137951+08:00 0 [Warning] [MY-011735] [Repl] Plugin group_replication reported: '[GCS] Shutting down an outgoing connection. This happens because something mig
ht be wrong on a bi-directional connection to node 127.0.0.1:53318. Please check the connection status to this member'
2024-05-13T18:42:58.138174+08:00 0 [Note] [MY-011735] [Repl] Plugin group_replication reported: '[GCS] Failure reading from fd=-1 n=18446744073709551615 from 127.0.0.1:53318'
2024-05-13T18:42:58.143932+08:00 0 [Note] [MY-011735] [Repl] Plugin group_replication reported: '[GCS] Connecting to 127.0.0.1:53318'
2024-05-13T18:42:58.144612+08:00 0 [Note] [MY-011735] [Repl] Plugin group_replication reported: '[GCS] Connected to 127.0.0.1:53318'
2024-05-13T18:43:06.612340+08:00 0 [Warning] [MY-011493] [Repl] Plugin group_replication reported: 'Member with address 127.0.0.1:53306 has become unreachable.'
2024-05-13T18:43:06.612772+08:00 0 [ERROR] [MY-011495] [Repl] Plugin group_replication reported: 'This server is not able to reach a majority of members in the group. This server
 will now block all updates. The server will remain blocked until contact with the majority is restored. It is possible to use group_replication_force_members to force a new grou
p membership.'
2024-05-13T18:43:06.613065+08:00 0 [Warning] [MY-011494] [Repl] Plugin group_replication reported: 'Member with address 127.0.0.1:53306 is reachable again.'
2024-05-13T18:43:06.613102+08:00 0 [Warning] [MY-011498] [Repl] Plugin group_replication reported: 'The member has resumed contact with a majority of the members in the group. Re
gular operation is restored and transactions are unblocked.'
2024-05-13T18:49:11.569388+08:00 0 [Note] [MY-011735] [Repl] Plugin group_replication reported: '[GCS] Connecting to 127.0.0.1:53318'
2024-05-13T18:49:11.569596+08:00 0 [Note] [MY-011735] [Repl] Plugin group_replication reported: '[GCS] Connected to 127.0.0.1:53318'
2024-05-13T18:49:22.635913+08:00 0 [Note] [MY-011735] [Repl] Plugin group_replication reported: '[GCS] Connecting to 127.0.0.1:53318'
2024-05-13T18:49:22.636319+08:00 0 [Note] [MY-011735] [Repl] Plugin group_replication reported: '[GCS] Connected to 127.0.0.1:53318'
2024-05-13T18:49:45.470441+08:00 0 [Note] [MY-011735] [Repl] Plugin group_replication reported: '[GCS] Connecting to 127.0.0.1:53318'
2024-05-13T18:49:45.470607+08:00 0 [Note] [MY-011735] [Repl] Plugin group_replication reported: '[GCS] Connected to 127.0.0.1:53318'
2024-05-13T18:49:46.433614+08:00 0 [Warning] [MY-011493] [Repl] Plugin group_replication reported: 'Member with address 127.0.0.1:53306 has become unreachable.'
```

Figure 1-10. A partial screenshot of the error log from the MySQL primary.

Here is a partial screenshot of the error log from the MySQL secondary:

```
2024-05-13T15:33:35.644491+08:00 0 [Warning] [MY-011493] [Repl] Plugin group_replication reported: 'Member with address 127.0.0.1:63306 has become unreachable.'
2024-05-13T15:33:35.644542+08:00 0 [ERROR] [MY-011495] [Repl] Plugin group_replication reported: 'This server is not able to reach a majority of members in the group. This server
 will now block all updates. The server will remain blocked until contact with the majority is restored. It is possible to use group_replication_force_members to force a new grou
p membership.'
2024-05-13T15:33:37.643586+08:00 0 [Warning] [MY-011494] [Repl] Plugin group_replication reported: 'Member with address 127.0.0.1:63306 is reachable again.'
2024-05-13T15:33:37.643641+08:00 0 [Warning] [MY-011498] [Repl] Plugin group_replication reported: 'The member has resumed contact with a majority of the members in the group. Re
gular operation is restored and transactions are unblocked.'
2024-05-13T15:33:55.493662+08:00 0 [Warning] [MY-011735] [Repl] Plugin group_replication reported: '[GCS] Shutting down an outgoing connection. This happens because something mig
ht be wrong on a bi-directional connection to node 127.0.0.1:63318. Please check the connection status to this member'
2024-05-13T15:33:56.372732+08:00 0 [Note] [MY-011735] [Repl] Plugin group_replication reported: '[GCS] Connecting to 127.0.0.1:63318'
2024-05-13T15:33:56.372892+08:00 0 [Note] [MY-011735] [Repl] Plugin group_replication reported: '[GCS] Connected to 127.0.0.1:63318'
2024-05-13T18:41:56.607615+08:00 0 [Warning] [MY-011493] [Repl] Plugin group_replication reported: 'Member with address 127.0.0.1:63306 has become unreachable.'
2024-05-13T18:41:56.607691+08:00 0 [ERROR] [MY-011495] [Repl] Plugin group_replication reported: 'This server is not able to reach a majority of members in the group. This server
 will now block all updates. The server will remain blocked until contact with the majority is restored. It is possible to use group_replication_force_members to force a new grou
p membership.'
2024-05-13T18:41:57.827228+08:00 0 [Warning] [MY-011494] [Repl] Plugin group_replication reported: 'Member with address 127.0.0.1:63306 is reachable again.'
2024-05-13T18:41:57.827605+08:00 0 [Warning] [MY-011498] [Repl] Plugin group_replication reported: 'The member has resumed contact with a majority of the members in the group. Re
gular operation is restored and transactions are unblocked.'
2024-05-13T18:42:00.927461+08:00 0 [Warning] [MY-011735] [Repl] Plugin group_replication reported: '[GCS] Shutting down an outgoing connection. This happens because something mig
ht be wrong on a bi-directional connection to node 127.0.0.1:63318. Please check the connection status to this member'
```

Figure 1-11. A partial screenshot of the error log from the MySQL secondary.

From these logs, warnings indicating 'has become unreachable' are evident. Under normal conditions, frequent 'unreachable' reports in a localhost scenario like this are unexpected. Blaming the network for Group Replication's delayed issue handling isn't optimal. Future chapters will explore enhancements to the network probing mechanism to address these false reporting issues.

1.3 Summary

This chapter analyzed how users approach solving MySQL issues. MySQL 8.0 has made strides in improvement, notably in mitigating scalability issues and introducing support for hash joins, which is a positive development. However, there remain numerous unresolved issues, both longstanding and newly emerging. Effectively addressing MySQL issues demands a thorough understanding of the problems and relevant theories; otherwise, the understanding may be incomplete.

The next chapter will demonstrate the considerable challenge of resolving obscure MySQL issues through various case studies. It necessitates a broad knowledge base and extensive logical reasoning to pinpoint the root causes of these issues.

Chapter 2: Mysterious MySQL Issues

MySQL, the most popular open-source database software with a history spanning several decades, is renowned for its simplicity and user-friendly nature, making it a cornerstone choice among internet companies. Despite its widespread adoption, MySQL faces a variety of challenges.

This chapter identifies nine puzzling MySQL peculiarities that are illustrative and pave the way for deeper exploration in upcoming topics.

2.1 SysBench Read-Write Test Exhibits Super-Linear Throughput Growth

In the MySQL 8.0.27 release version, for example, in a NUMA environment on x86 architecture, using SysBench to remotely test MySQL's read-write capabilities. The MySQL transaction isolation level is set to Read Committed. MySQL instances 1 and 2 are deployed on the same machine, with a testing duration of 60 seconds. The results of separate SysBench tests for MySQL instance 1 and instance 2 are shown in the following figure.

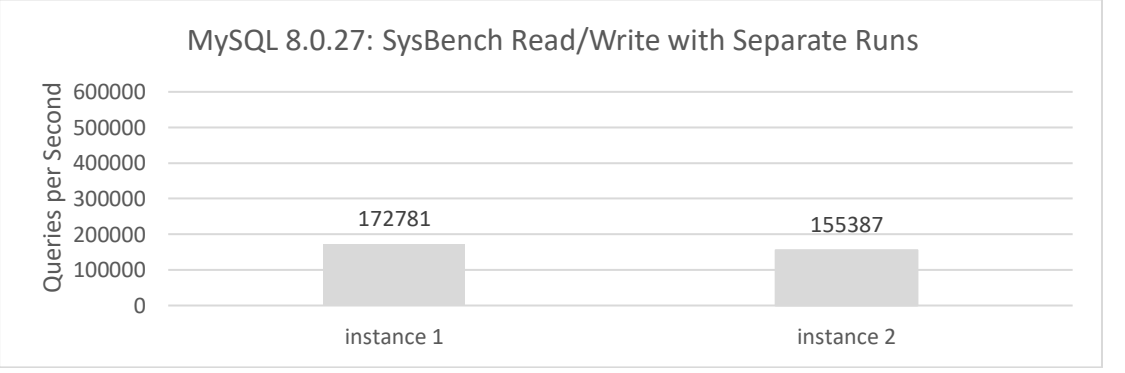


Figure 2-1. Throughput of MySQL running separately.

The throughput of each instance is modest, with figures of 172,781 QPS and 155,387 QPS respectively. Combined, the two instances achieve a total throughput of 328,168 QPS. When using SysBench to simultaneously test the read and write capabilities of these two instances, the obtained throughputs are 271,232 QPS and 275,197 QPS respectively.

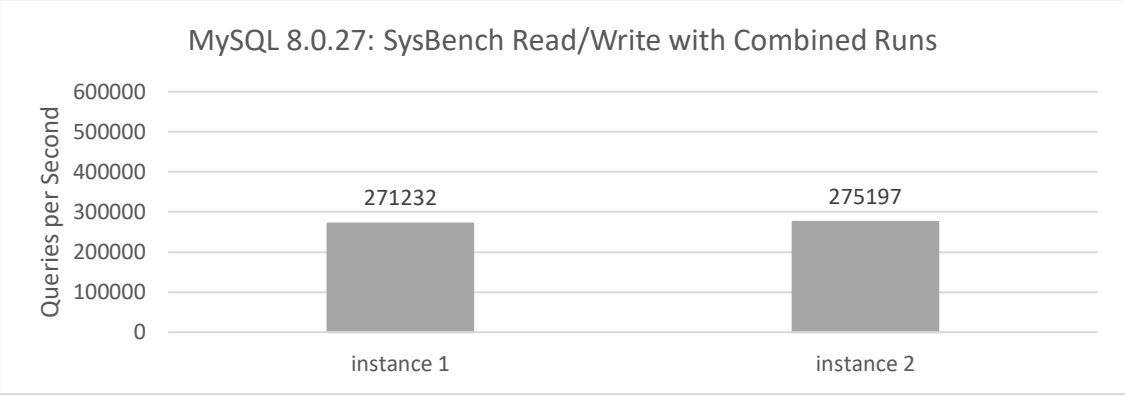


Figure 2-2. Throughput of MySQL running together.

The combined throughput of the two MySQL instances is 546,429 QPS. This data demonstrates that when these two MySQL instances share the same machine, the aggregated throughput is significantly higher than the sum of their individual throughputs when run separately. For detailed statistical comparisons, please refer to the following figure.

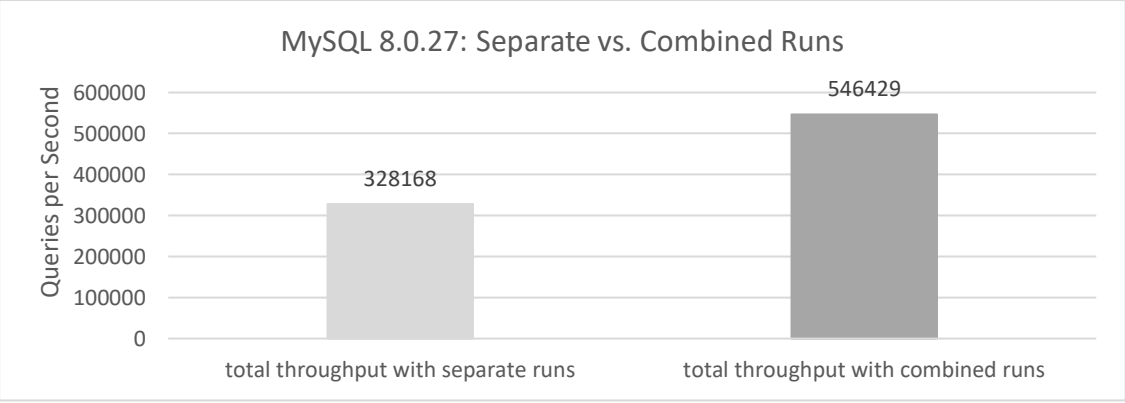


Figure 2-3. Total throughput of running separately vs. running together.

In terms of mathematical logic, if the total throughput when running two instances together is roughly equal to the sum of the throughputs when running them separately, it represents a linear relationship. If the combined throughput exceeds this sum, it suggests a superlinear relationship.

What causes superlinear relationships? Does MySQL truly exhibit superlinear capabilities? Only by delving into practical computer fundamentals and advanced MySQL concepts can one fully unravel this mystery.

2.2 Post-PGO, TPC-C Throughput Declines Instead

Profile-guided optimization (PGO) is a well-established technique for improving compile-time optimization decisions. Profile information is collected through instrumentation or sampling of the executable, and this data is used to optimize the executables it was gathered from [81]. Despite its effectiveness, PGO has not been widely adopted by software projects due to its cumbersome dual-compilation model. Nevertheless, PGO remains a highly valuable optimization technique to consider for enhancing MySQL performance, as it theoretically has the potential to significantly improve MySQL's efficiency.

The following diagram illustrates the application of PGO to higher versions of MySQL 8.0.

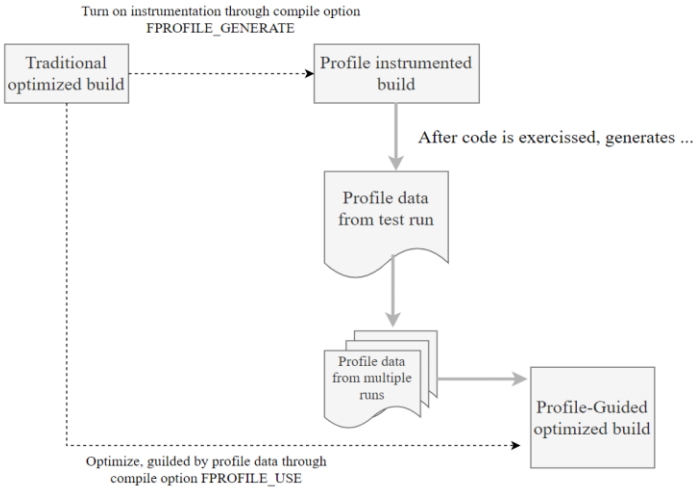


Figure 2-4. Using PGO in higher versions of MySQL 8.0: a step-by-step guide.

From the diagram, the Profile-Guided Optimization (PGO) mechanism involves several steps:

1. Initially, compile a specific version of MySQL with the compilation option "-DFPROFILE_GENERATE=ON".

2. Start this MySQL version and capture training data by running performance tests such as TPC-C, which helps collect performance metrics.

3. After completing the training phase, perform a second compilation with the option "-DFPROFILE_USE=ON". During this compilation, the compiler automatically utilizes the gathered statistical data to optimize conditional branches and related aspects, significantly enhancing the performance of the resulting MySQL executable.

The following figure illustrates the relationship between throughput and concurrency before and after applying PGO to MySQL 8.0.27.

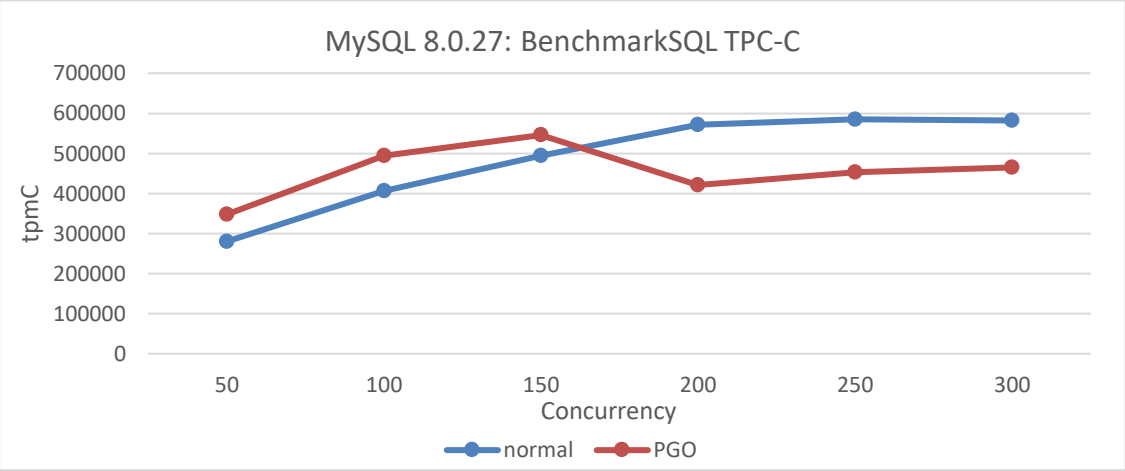


Figure 2-5. Performance comparison tests before and after using PGO in MySQL 8.0.27.

Based on the figure, it's evident that PGO leads to notable enhancements in MySQL throughput at lower concurrency levels. However, beyond 150 concurrency, both the overall throughput and peak performance show a decline.

Does PGO primarily benefit low-concurrency scenarios, or are there additional factors limiting its effectiveness? This question delves into queueing theory and system architecture. Further exploration of practical computer fundamentals will provide deeper insights into this matter.

2.3 Unexpected Negative Effects of Thread Pool After MySQL Scalability Improvements

After applying various scalability patches to MySQL 8.0.27, it's crucial to evaluate whether the Percona thread pool still effectively addresses scalability issues. The following figure depicts the results of TPC-C testing on a standalone MySQL instance using BenchmarkSQL. The deep blue line indicates the configuration with the Percona thread pool enabled (thread pool size = 128), while the deep red line represents the configuration with the thread pool disabled. The test covered concurrency levels ranging from 50 to 2000, utilizing a database with 1000 warehouses.

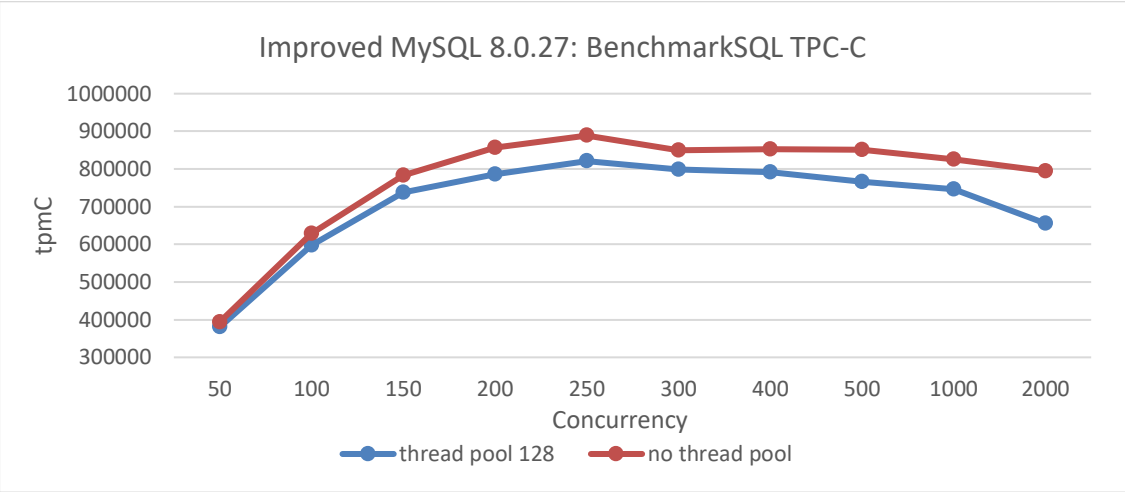


Figure 2-6. **Percona Thread Pool** caused a significant decrease in throughput compared to when the thread pool was disabled.

From the figure, it is clear that enabling the Percona thread pool led to a significant decrease in throughput compared to when the thread pool was disabled. It's notable that even without the thread pool enabled, the scalability of MySQL 8.0 improved markedly compared to MySQL 5.7 versions. This suggests that the additional benefit of using the Percona thread pool for enhancing TPC-C testing was limited. As MySQL's inherent scalability issues are largely addressed in version 8.0, the utility of the Percona thread pool becomes less pronounced. Moreover, the Percona thread pool mechanism itself introduces overhead, which is reflected in the figure's results.

It's important to acknowledge that the Percona thread pool remains valuable in scenarios involving frequent connection creation and severe contention. However, the key question remains: what exactly contributed to such significant improvements in MySQL scalability? Future chapters will explore these mysteries further.

2.4 In MySQL 8.0, TPC-C Throughput Drops Too Quickly

The standards for long-term TPC-C testing are as follows: the TPC-C benchmark requires that the database run for at least eight hours with jitters less than 2% in two hours [25].

Based on MySQL 8.0.27, long-term TPC-C testing was conducted using the BenchmarkSQL tool. Below are the BenchmarkSQL testing parameters:

```
warehouses=1000
loadWorkers=100
```

```
terminals=200
warehouses-begin=1
warehouses-end=1000
//To run specified transactions per terminal- runMins must equal zero
runTxnsPerTerminal=0
//To run for specified minutes- runTxnsPerTerminal must equal zero
runMins=480
//Number of total transactions per minute
limitTxnsPerMin=0
//Set to true to run in 4.x compatible mode. Set to false to use the
//entire configured database evenly.
terminalWarehouseFixed=false
//The following five values must add up to 100
//The default percentages of 45, 43, 4, 4 & 4 match the TPC-C spec
newOrderWeight=45
paymentWeight=43
orderStatusWeight=4
deliveryWeight=4
stockLevelWeight=4
```

From the above, it can be seen that there are 1000 warehouses, with a concurrency of 200, and terminalWarehouseFixed is set to false. This setting enables each transaction to utilize a different warehouse ID every time, thereby accessing data across all warehouses.

The following figure illustrates the throughput over time during long-term testing. The TPC-C throughput shows a decline rate that significantly surpasses expectations, nearing a 50% decrease.

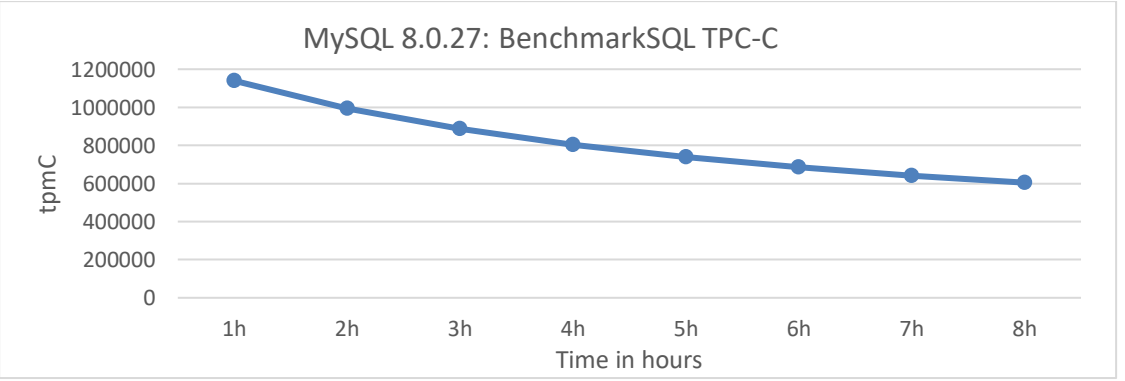


Figure 2-7. Performance degradation exposed during BenchmarkSQL testing of MySQL 8.0.27.

This problem was identified during testing using BenchmarkSQL and may not necessarily occur with other TPC-C testing tools. As of the current version, MySQL 8.0.38, the issue of rapid throughput decline has not been fully resolved. Subsequent chapters will delve into detailed explanations of the

underlying causes of this issue.

2.5 Repeatable Read Surprisingly Outperforms Read Committed

Transaction isolation is fundamental to database processing, represented by the 'I' in the ACID acronym. The isolation level determines the balance between performance and the reliability, consistency, and predictability of results when multiple transactions concurrently make changes and queries. Commonly used isolation levels are Read Committed, Repeatable Read, and Serializable. By default, InnoDB uses Repeatable Read.

InnoDB employs distinct locking strategies for each isolation level, impacting query locking behavior under concurrent conditions. Depending on the isolation level, queries may need to wait for locks currently held by other sessions before execution begins [24]. There's a common perception that stricter isolation levels can degrade performance. How does MySQL perform in practical scenarios?

Tests were conducted across Serializable (SER), Repeatable Read (RR), and Read Committed (RC) isolation levels using two benchmark types: SysBench uniform and pareto tests. The SysBench uniform test simulates low-conflict scenarios, while the SysBench pareto test models high-conflict situations. Due to excessive deadlock logs generated during the SysBench pareto test, which significantly interfered with performance analysis, these logs were suppressed by modifying the source code to ensure fair testing conditions. Moreover, the MySQL testing program utilized a modified version for accuracy, rather than the original version.

The figure below presents results from the SysBench uniform test, where concurrency increases from 50 to 800 in doubling increments. Given the few conflicts in this test type, there is little variation in throughput among the three transaction isolation levels at low concurrency levels. However, beyond 400 concurrency, the throughput of the Serializable isolation level exhibits a notable decline.

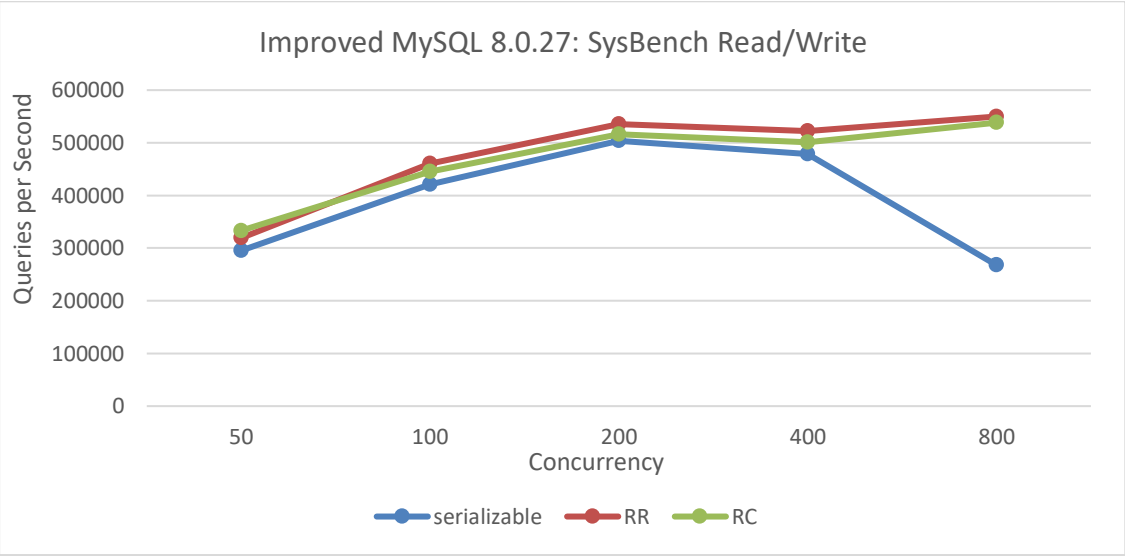


Figure 2-8. SysBench read-write performance comparison with low conflicts under different isolation levels.

Below 400 concurrency, the differences are minor because of fewer conflicts in the uniform test. With fewer conflicts, the impact of lock strategies under different transaction isolation levels is reduced. However, Read Committed is mainly constrained by frequent acquisition of MVCC ReadView, resulting in performance inferior to Repeatable Read.

Continuing with the SysBench test under pareto distribution conditions, specific comparative test results can be seen in the following figure.

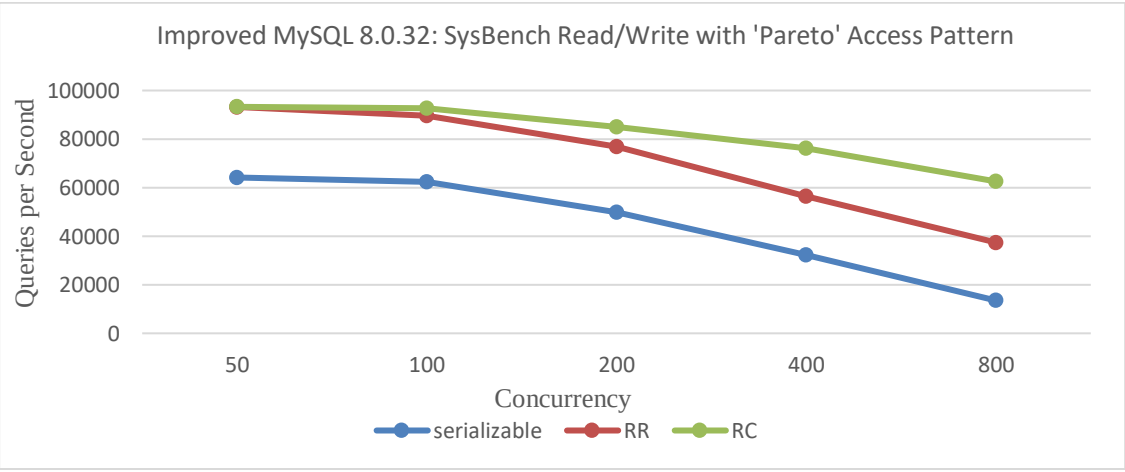


Figure 2-9. SysBench read-write performance comparison with high conflicts under different

isolation levels.

The figure clearly illustrates that in scenarios with significant conflicts, performance differences due to lock strategies under different transaction isolation levels are pronounced. As anticipated, higher transaction isolation levels generally exhibit lower throughput, particularly under severe conflict conditions.

In scenarios with few conflicts, performance is primarily constrained by the overhead of acquiring ReadView in MVCC. This is because, under the Read Committed isolation level, MySQL must copy the entire active transaction list each time it reads from the global active transaction list, whereas under Repeatable Read, it only needs to obtain a copy of the active transaction list at the start of the transaction.

To summarize, in low-conflict tests like SysBench uniform, the overhead of MVCC ReadView is the predominant bottleneck, outweighing lock overhead. Consequently, Repeatable Read performs better than Read Committed. Conversely, in high-conflict tests like SysBench pareto, lock overhead becomes the primary bottleneck, resulting in Read Committed outperforming Repeatable Read.

2.6 Group Replication Throughput Lower Than Semisynchronous Replication

During Group Replication operation, a certification database is maintained. Regular cleanup of outdated certification information is crucial to manage memory usage efficiently. However, this cleanup process involves acquiring a global latch, temporarily pausing MySQL primary execution until the certification information is cleared.

In contrast, traditional semisynchronous replication requires the MySQL secondary to process a substantial amount of relay log event information. Only after these relay log events are written to disk can the secondary send acknowledgment (ack) information back to the MySQL primary. This process includes network interactions, handling numerous relay log events, and disk flushes, resulting in relatively longer response times for semisynchronous replication.

Overall, with semisynchronous replication, the MySQL primary must wait for acknowledgment from the MySQL secondary after relay log events are written to disk before it can proceed. In contrast, Group Replication continues processing once consensus is achieved at the Paxos layer, without waiting for log writes at that layer. Theoretically, Group Replication can achieve higher throughput.

In the scenario of a two-node cluster, TPC-C throughput comparisons based on concurrency were conducted between Group Replication and semisynchronous replication. Please refer to the following

figure for details.

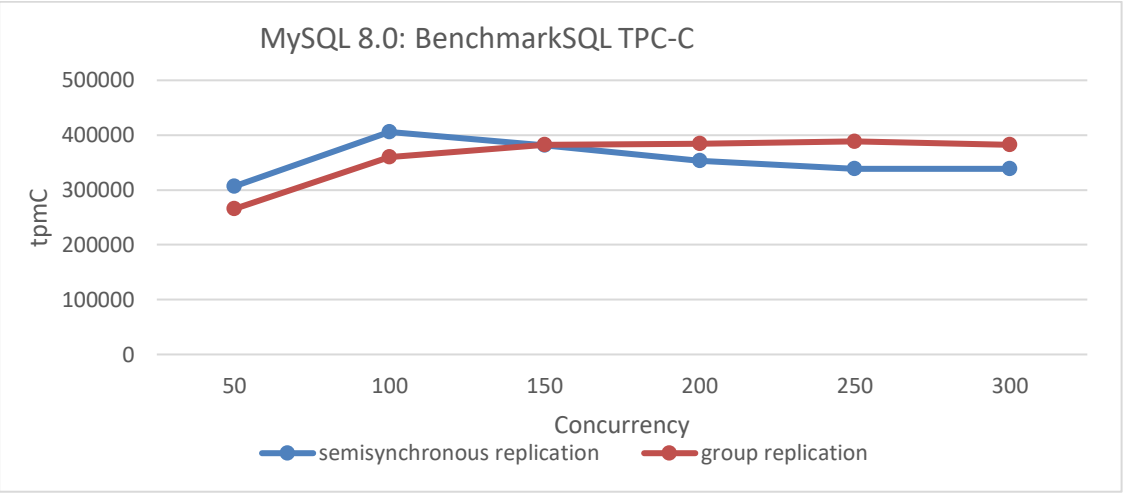


Figure 2-10. Performance comparison between Group Replication and semisynchronous replication.

The figure indicates that under low concurrency, semisynchronous replication outperforms Group Replication, whereas Group Replication shows superior performance under high concurrency. Semisynchronous replication reaches its peak performance at 100 concurrency, whereas Group Replication peaks at 250 concurrency, albeit with less impressive peak performance compared to semisynchronous replication. These test results are unexpected. What could be the issue?

The root problem lies in the certification database mechanism used by Group Replication, which is absent in semisynchronous replication. This mechanism involves substantial memory allocation and deallocation, significantly limiting throughput improvement. Despite not requiring Paxos log persistence, this bottleneck negates the advantages of Group Replication. It is clear that the implementation of the certification database mechanism poses the primary performance challenge for Group Replication.

2.7 Modified Group Replication Outperforms Semisynchronous Replication

Group Replication has been extensively enhanced while addressing scalability issues in MySQL 8.0.32. To validate these improvements, simultaneous testing of semisynchronous replication and Group Replication with Paxos log persistence was conducted. The deployment setup included two-node configurations for both semisynchronous and Group Replication, hosted on the same machine with independent SSDs and NUMA binding to isolate each node. Specifically, the MySQL primary

node utilized NUMA nodes 0 to 2, while the MySQL secondary node utilized NUMA node 3. All settings, except those directly related to semisynchronous or Group Replication configurations, remained identical.

The following figure shows the throughput comparison of semisynchronous replication and Group Replication with Paxos log persistence under different concurrency levels.

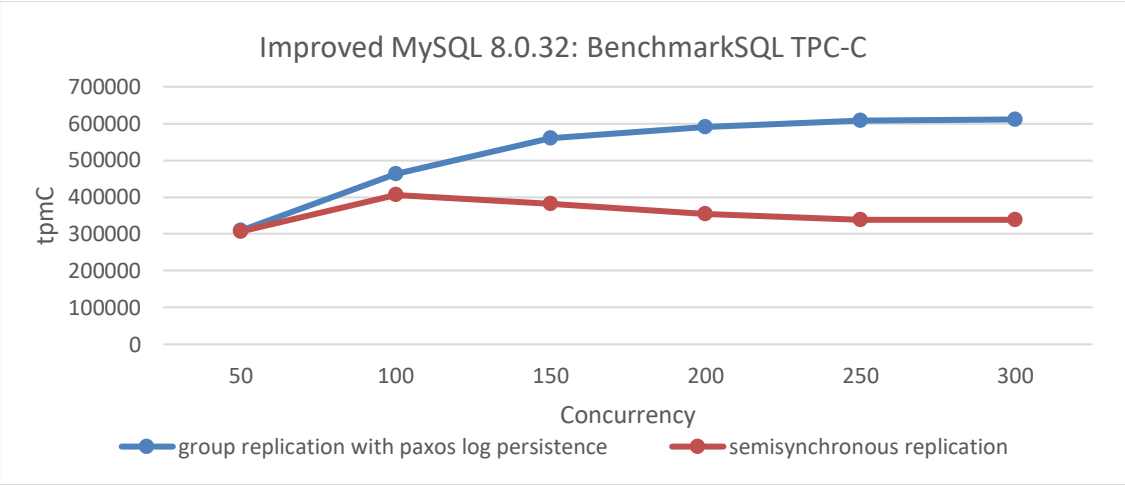


Figure 2-11. Performance comparison between Group Replication with Paxos log persistence and semisynchronous replication.

Both employ persistence mechanisms, with Group Replication utilizing Paxos log persistence and semisynchronous replication utilizing relay log persistence. Due to these distinct mechanisms, Group Replication with Paxos log persistence demonstrates significantly superior performance compared to semisynchronous replication.

Meta Company has implemented a MySQL high availability solution based on Raft. According to tests conducted by Meta Company developers, the performance of the Raft-based improved version is comparable to that of semisynchronous replication. For specific details, refer to the figure below [76].

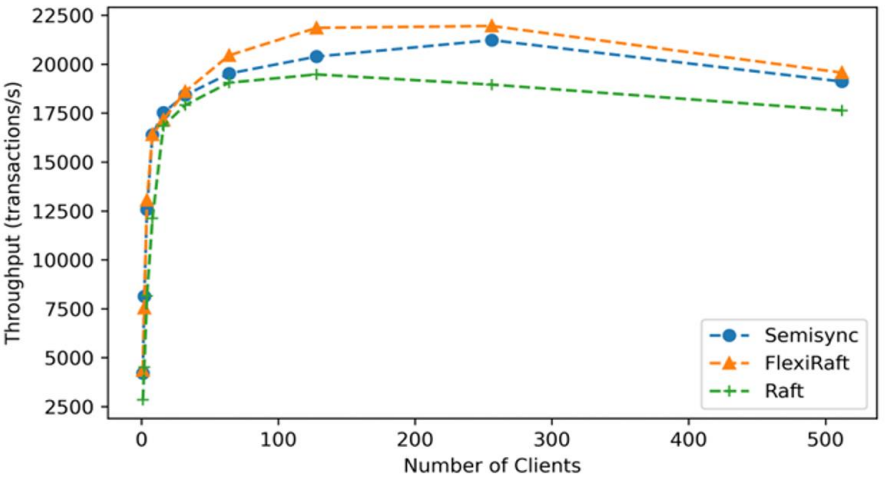


Figure 2-12. Throughput comparison borrowed from Meta paper.

In theory, Group Replication can leverage a batching-based disk persistence mechanism, eliminating the need to process binlog events during disk writes, thereby achieving higher expected throughput. It is noteworthy that when comparing performance with semisynchronous replication, the Group Replication setup includes Paxos log persistence. This comparison ensures fairness and mitigates any influence from the native Group Replication version lacking Paxos log persistence on test outcomes. Future chapters will delve into detailed discussions on modifying Group Replication and examining specific factors contributing to the scalability challenges of native semisynchronous replication.

2.8 SysBench Shows No Effect, TPC-C Performs Well

The specifics of the lock-sys optimization in MySQL 8.0 are detailed below:

```
commit 1d259b87a63defa814e19a7534380cb43ee23c48
Author: Jakub Łopuszański <jakub.lopuszanski@oracle.com>
Date:   Wed Feb 5 14:12:22 2020 +0100
    WL#10314 - InnoDB: Lock-sys optimization: sharded lock_sys mutex

    The Lock-sys orchestrates access to tables and rows. Each table, and each row,
    can be thought of as a resource, and a transaction may request access right for
    a resource. As two transactions operating on a single resource can lead to
    problems if the two operations conflict with each other, Lock-sys remembers
    lists of already GRANTED lock requests and checks new requests for conflicts in
    which case they have to start WAITING for their turn.

    Lock-sys stores both GRANTED and WAITING lock requests in lists known as queues.
    To allow concurrent operations on these queues, we need a mechanism to latch
```

```
these queues in safe and quick fashion.

In the past a single latch protected access to all of these queues.
This scaled poorly, and the management of queues become a bottleneck.
In this WL, we introduce a more granular approach to latching.

Reviewed-by: Pawel Olchawa <pawel.olchawa@oracle.com>
Reviewed-by: Debarun Banerjee <debarun.banerjee@oracle.com>
  RB:23836
```

This marks a scalability improvement in MySQL 8.0. This enhancement resolves the bottleneck of the global latch, thereby freeing lock scheduling in the InnoDB storage engine. To validate the effectiveness of this lock-sys optimization, refer to the specific test results illustrated in the figure below using SysBench read-write tests.

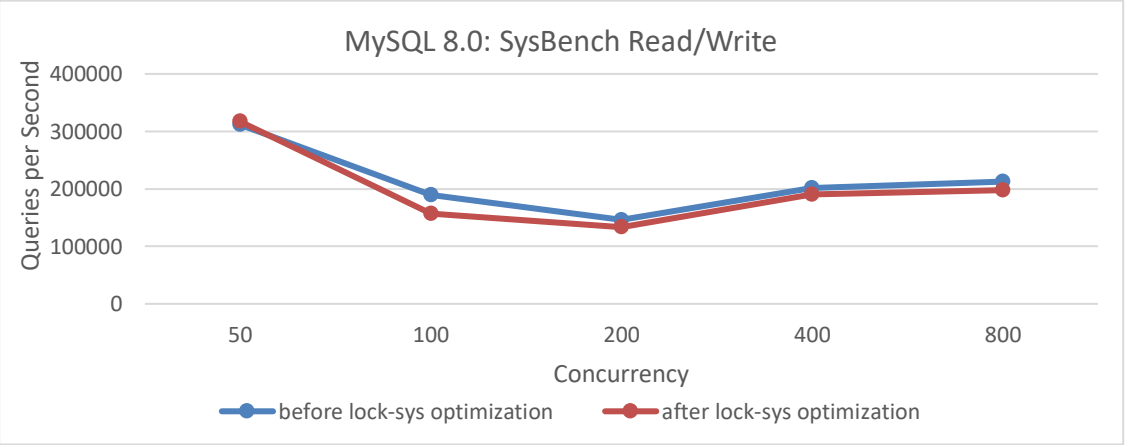


Figure 2-13. Comparison of SysBench read-write tests before and after lock-sys optimization.

It's surprising that after implementing the lock-sys optimization, the throughput decreased, which was unexpected. To mitigate interference from NUMA compatibility issues in MySQL code, the MySQL running instance was bound to NUMA node 0 (similar to an SMP environment). The test results are as follows.

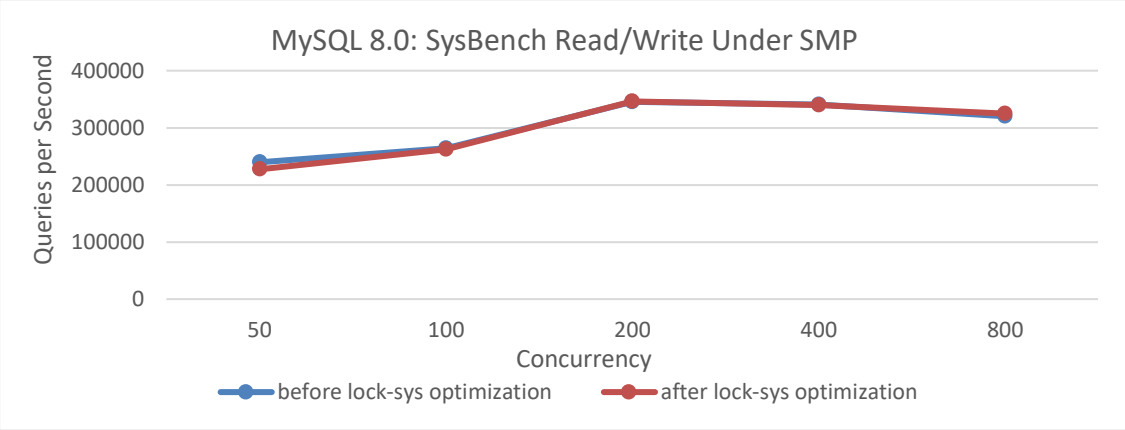


Figure 2-14. Comparison of SysBench read-write tests before and after lock-sys optimization under SMP.

The figure shows that the difference before and after optimization is minimal, almost negligible. This suggests that during the testing process, the effectiveness of the lock-sys optimization is overshadowed by other factors, resulting in distorted test results. However, binding to NUMA node 0 reduced interference from other bottlenecks, narrowing the performance gap. It also indicates that the lock-sys optimization has limited impact on SysBench standard read-write tests.

Using BenchmarkSQL for TPC-C testing, the results are as follows:

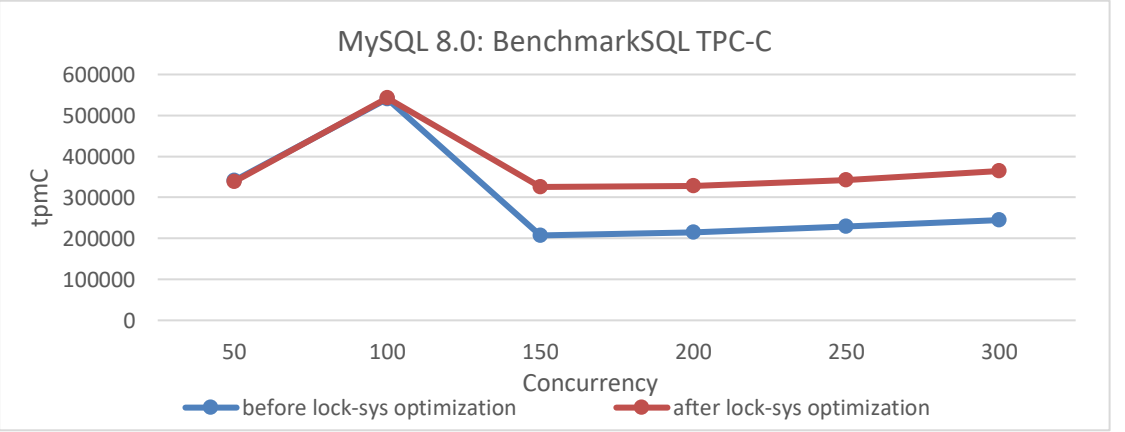


Figure 2-15. Comparison of BenchmarkSQL tests before and after lock-sys optimization.

The figure demonstrates a noticeable improvement from the lock-sys optimization. However, it raises questions as to why SysBench testing shows no effect while BenchmarkSQL testing does. Understanding the differences between these two tools and important considerations during testing

will be thoroughly discussed in upcoming chapters.

2.9 Is Disabling NUMA Really Beneficial for MySQL?

The impact of disabling NUMA on the MySQL primary node was initially tested. The deployment setup was as follows: BenchmarkSQL high-pressure stress tests were conducted on two x86 machines with identical hardware configurations. One machine had NUMA disabled in the BIOS, while the other had NUMA enabled. The comparison of TPC-C throughput versus concurrency is illustrated in the figure below.

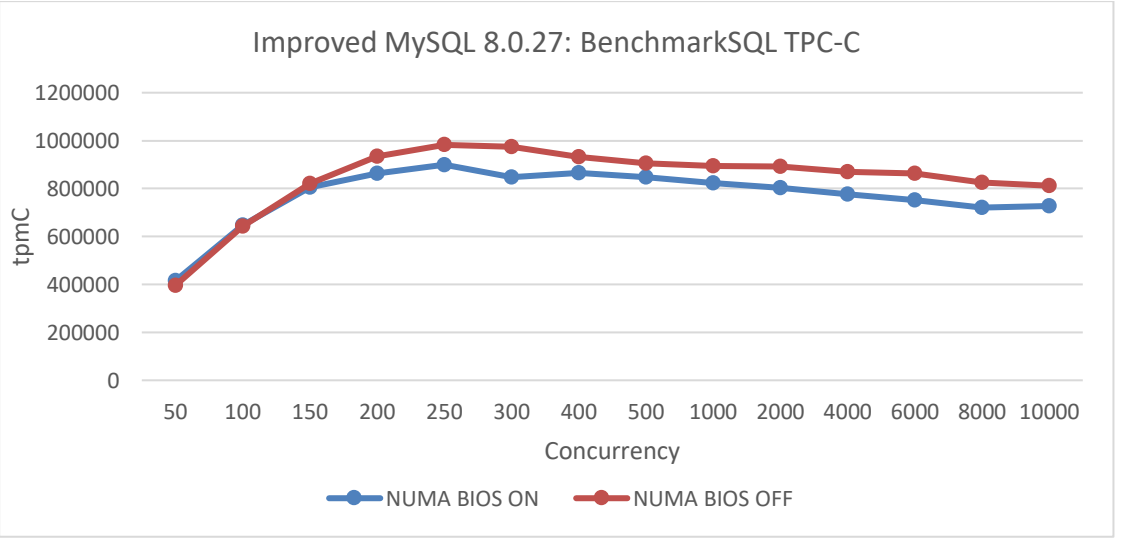


Figure 2-16. Significantly improved TPC-C throughput by disabling NUMA in BIOS.

The figure demonstrates that disabling NUMA on x86 machines significantly improves TPC-C throughput. This improvement stems from the favorable memory allocation mechanism after disabling NUMA in the BIOS, particularly beneficial for applications like MySQL primary servers.

Now, does disabling NUMA also benefit MySQL secondary replay? Using the same machines mentioned earlier for testing, the setup details are as follows: in the environment where NUMA is disabled in the BIOS, NUMA binding cannot be utilized, allowing all memory to be utilized. Conversely, in the environment where NUMA is enabled in the BIOS, MySQL secondaries are bound to NUMA node 0. The following figure illustrates the balanced replay speeds tested in these different environments.

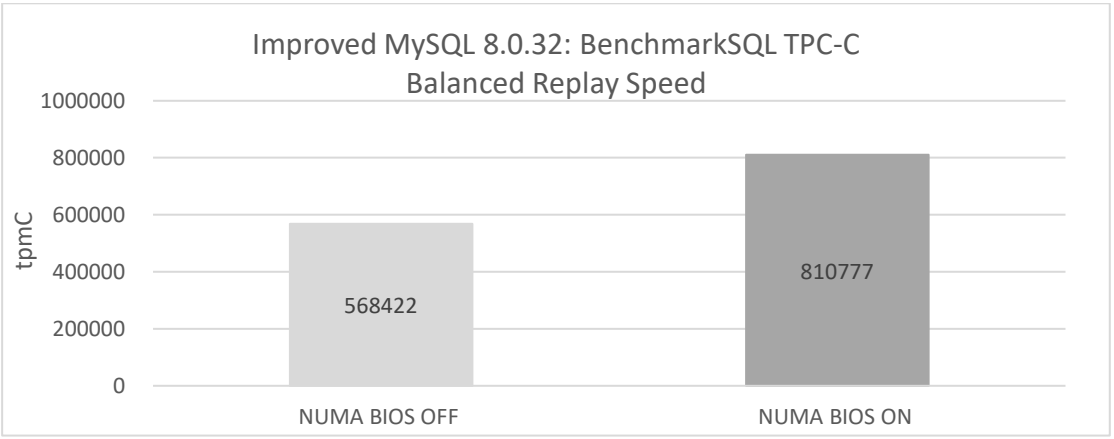


Figure 2-17. Comparison of balanced replay speed before and after disabling NUMA in BIOS.

The figure reveals that disabling NUMA at the BIOS level results in a balanced replay speed of only around 570,000 tpmC, primarily due to the unresolved NUMA unfriendliness issues with MySQL secondaries. In contrast, enabling NUMA and binding MySQL secondaries to a single NUMA node can achieve a balanced replay speed exceeding 810,000 tpmC. This test highlights the disadvantage of disabling NUMA at the BIOS level for MySQL secondary replay efficiency. Effective MySQL secondary replay with NUMA disabled necessitates addressing these NUMA unfriendliness issues, as failure to do so significantly reduces efficiency. The upcoming chapter 10 on enhancing MySQL secondary replay will provide a detailed examination of these NUMA unfriendliness issues.

2.10 Summary

This chapter delves into classic and intricate MySQL issues, which pose significant challenges for analysis. The resolution of these issues begins with a detailed logical analysis, as outlined in the following chapter. Addressing and solving these problems necessitates a profound understanding of computer fundamentals and MySQL internals. Computer fundamentals encompass a broad range of topics including computer architecture, data structures, algorithms, operating systems, computer networks, compilers, queueing theory, and distributed systems theory, among others. These topics will be thoroughly explored in Chapter 4. Chapter 5 will focus specifically on MySQL internals.

Part2 Basics

Chapter 3: How to Solve Software Problems Effectively

When encountering challenging problems, software professionals often rely on experience to find solutions. However, especially with the numerous MySQL issues discussed in this book, it is challenging to identify the root cause of problems based solely on experience. Therefore, scientific methods are needed to analyze and solve these problems.

This chapter discusses how to analyze and solve problems from two different levels: strategy and tactics [1].

3.1 Analysis Strategy

There are many analysis strategies, including psychological strategies, simplification strategies, pattern-finding strategies, strategies to increase reproducibility, and strategies to find key evidence.

3.1.1 Psychological Strategies

Many people often feel intimidated by problems, especially when they appear mysterious or difficult to solve. When they encounter such challenges, their first instinct is to seek help from others if they cannot resolve them on their own. However, relying solely on others can hinder the opportunity to develop and strengthen one's own logical thinking skills through problem-solving.

When faced with difficulties, particularly those that seem complex, it is crucial to approach them with optimism. These challenges can serve as opportunities to unlock one's potential and enhance problem-solving abilities. MySQL, as a complex database software, presents numerous such challenges.

Firstly, encountering these challenges leads to a deeper understanding of MySQL over time. Secondly, they offer invaluable opportunities to tackle complex issues, thereby enhancing developers' problem-solving capabilities. Lastly, the historical issues associated with MySQL are invaluable; without them, books like this one would not exist. Each challenge contributes to expanding our knowledge base and refining our approach to solving MySQL-related problems.

3.1.2 Simplification Strategies

When a problem occurs and reproducing the original conditions is very complex, it's essential to

simplify the reproduction environment step by step. This approach allows you to check if the problem still occurs. If it does, you continue simplifying until the problem no longer manifests. This strategy significantly simplifies the difficulty of problem analysis. For instance, when troubleshooting issues with a MySQL cluster, begin by checking if the problem occurs on a single MySQL instance. If it does, there's no need to set up the entire cluster; you can troubleshoot using the single instance.

Simplifying the reproduction environment step-by-step helps isolate core factors contributing to an issue, making diagnosis and resolution easier. For instance, when dealing with Group Replication issues, while some problems only occur under strict user conditions, most can be replicated in a local development environment. Local replication eliminates the need for extensive on-site troubleshooting, facilitating focused problem analysis and resolution.

3.1.3 Pattern-finding Strategies

When a problem recurs, it often reveals underlying characteristics, facilitating the statistical aggregation of patterns and regularities associated with it. This data supports issue reproduction and inference-making. For instance, after PGO, instead of increasing, peak performance decreased. Testing at various concurrency levels showed a significant decline under high concurrency, with improvement observed at lower levels. This insight lays the groundwork for addressing subsequent issues and reinforces confidence in PGO's potential to enhance throughput in high concurrency scenarios.

3.1.4 Strategies to Increase Reproducibility

Many issues are environment-specific, often manifesting sporadically, especially under high concurrency. The challenge lies in addressing these infrequent occurrences, which may happen just once every few months. Increasing the frequency of issue reproduction—from every few months to every few hours or minutes—significantly simplifies their resolution.

How can this be achieved? Capturing patterns in issue occurrence is crucial. For example, when addressing simultaneous failures in Group Replication that sporadically freeze views, analyzing statistical patterns reveals critical insights. These issues often cluster around specific thresholds. Adjusting lower-level communication timeout settings to align with network interruption durations enables more frequent issue reproduction. Once these critical factors are understood, the likelihood of reproducing issues increases significantly, laying a solid foundation for effective problem resolution.

Therefore, the ability to reproduce issues often proves crucial in solving challenging problems. Enhancing the reproducibility by capturing the characteristics of issue recurrence is key to expediting issue resolution.

3.1.5 Strategies to Find Key Evidence

Sometimes, revealing the characteristics of a problem directly is challenging; it necessitates creating conditions that fully manifest the issue. Let's revisit the figure below, using PGO as an example.

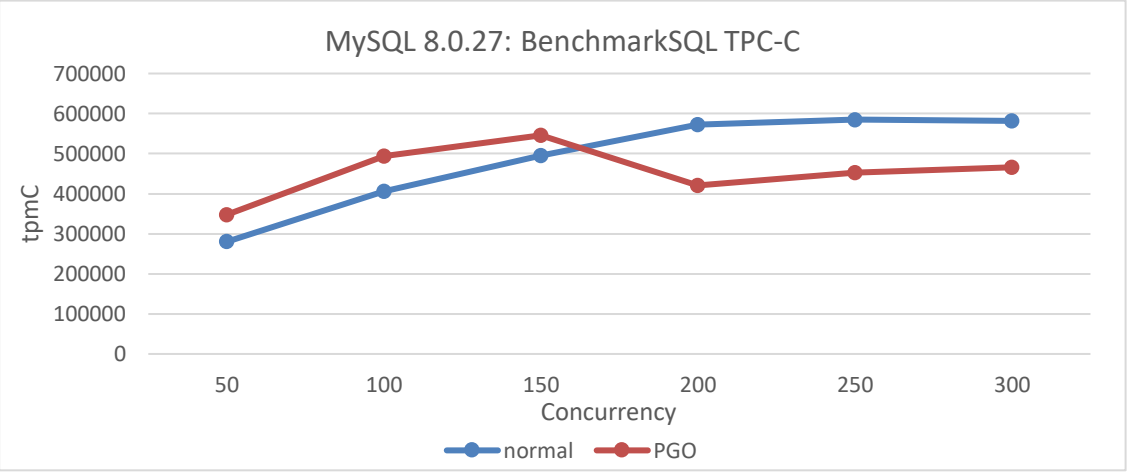


Figure 3-1. Performance comparison tests before and after using PGO in MySQL 8.0.27.

PGO (Profile-Guided Optimization) is effective in enhancing throughput under low concurrency. However, there is a strong belief that interference from other factors in high concurrency scenarios limits PGO's ability to achieve its intended effectiveness.

Based on the above analysis, binding the MySQL instance to NUMA node 0 allowed us to explore whether PGO can achieve its performance potential in an SMP-like environment. For specific comparative test results, please refer to the following figure:

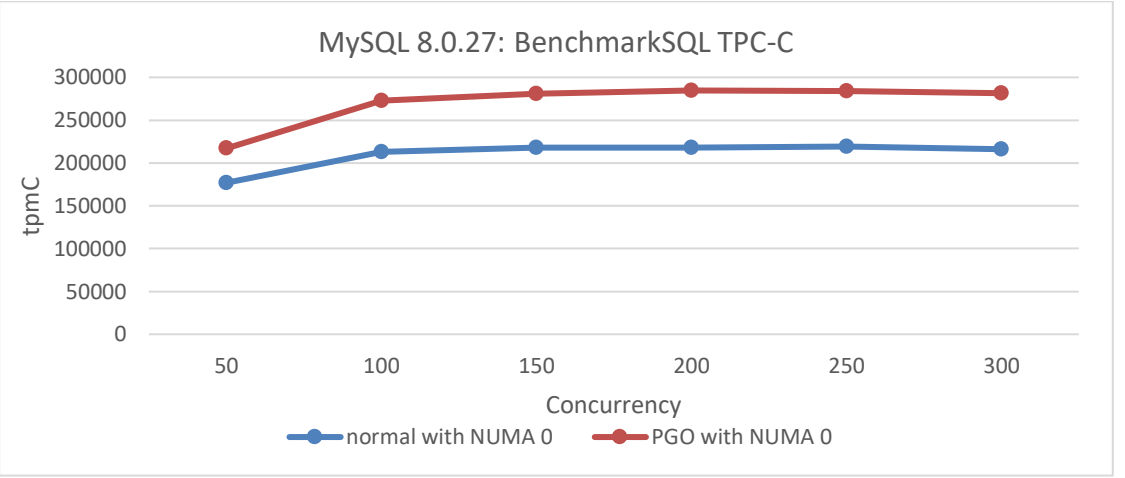


Figure 3-2. Performance comparison tests before and after using PGO in MySQL 8.0.27 under SMP.

From the figure, binding to a single NUMA node demonstrates that PGO significantly enhances performance across all concurrency scenarios shown. This finding underscores PGO's ability to boost MySQL throughput, establishing a strong basis for improving performance in NUMA environments.

Regarding the sharp decline in TPC-C throughput discussed in section 2.4, users encountered this issue during regression testing after upgrading to MySQL version 8.0.29. Notably, this performance drop was absent in MySQL version 8.0.25. User feedback on this discrepancy is pivotal, greatly facilitating the resolution process.

3.2 Logical Thinking

At the core of programming lies problem-solving. Logical reasoning empowers programmers to break down complex problems into manageable components and devise practical solutions by analyzing requirements, understanding relationships between elements, and planning logically and efficiently. This structured approach enables programmers to think critically and innovate creatively [101]. Logical reasoning is crucial in computer engineering for creating justifiable inferences. Enhanced use of logical thinking in workplaces boosts productivity by refining decision-making and minimizing errors. Rooted in sequential thought, logic underpins all systematic analysis, solution design, error correction, and performance optimization in programming. It utilizes deductive, inductive, abductive reasoning, and reductio ad absurdum to gather evidence, identify contradictions, and ensure software reliability

3.2.1 Deductive Reasoning

Deductive reasoning involves deriving conclusions from premises, often structured as syllogisms. A syllogism typically comprises a major premise, a minor premise, and a conclusion. For instance:

- Major premise: All cluster databases cannot simultaneously meet the three requirements of CAP (consistency, availability, and partition tolerance).
- Minor premise: Group Replication is a type of cluster database.
- Conclusion: Therefore, Group Replication also cannot meet the three requirements of CAP.

Given Group Replication's inability to meet these requirements, it should be chosen based on user-specific needs rather than attempting to satisfy all criteria.

Here's another example:

- Major premise: Asynchronous network systems struggle to differentiate between slow response and system outage.
- Minor premise: TCP communication relies on asynchronous networks.
- Conclusion: Thus, TCP communication systems also encounter challenges in distinguishing between slow response and actual system failure.

Due to these challenges, TCP design must incorporate specific features such as robust timeout mechanisms and support for idempotency.

3.2.2 Inductive Reasoning

Inductive reasoning extrapolates from specific instances to general conclusions. For instance:

- On x86 NUMA platforms, native MySQL 8.0.32 shows poor scalability under the Read Committed isolation level.
- On ARM NUMA platforms, native MySQL 8.0.32 similarly demonstrates poor scalability under the Read Committed isolation level.

Therefore, on various NUMA platforms, native MySQL 8.0.32 consistently exhibits poor scalability under the Read Committed isolation level. Inductive reasoning is crucial for problem analysis and holds practical significance across diverse applications.

3.2.3 Abductive Reasoning

Abductive reasoning, also known as backward reasoning, involves deriving the best explanation from observed facts. It is a widely employed method in solving programming problems, especially when symptoms are evident but the underlying cause is unclear. For example, after optimizing MySQL with PGO (Profile-Guided Optimization), if throughput peaks decrease rather than increase, abductive reasoning may reveal that poor MySQL handling in NUMA environments is causing this outcome.

3.2.4 Reductio ad Absurdum

"Reductio ad absurdum" is a method of disproving a proposition by demonstrating that its logical conclusion leads to an absurd or contradictory outcome. This approach aims to show that a statement or hypothesis must be false because accepting it would result in an illogical or unacceptable result. It is a common method used to refute arguments, characterized by the strategy of "retreating to advance", where introducing absurdity exposes flaws in reasoning.

For example, Group Replication often reports network unreachable errors. To verify the reliability of these error messages, consider them accurate. However, deploying Group Replication in a controlled localhost environment, where external network access isn't necessary, should not result in the same network errors. Yet, during the normal TPC-C data loading process, frequent network unreachable errors persist. This inconsistency indicates that the error messages cannot be relied upon.

3.2.5 Utilizing Various Methods Comprehensively

The challenges in software problem-solving differ significantly from those in mathematical problems, often influenced by human misjudgments and numerous external factors, leading to frequent errors in judgment. The theoretical foundation in this area is relatively underdeveloped, heightening the complexity of problem-solving.

Utilizing logical thinking methods is crucial for solving complex programming issues. These methods often necessitate synthesizing diverse approaches to address challenges, occasionally incorporating probability information.

3.3 Tactics for Solving Problems

Strategies themselves rarely directly solve problems; more powerful methods are needed to resolve them.

3.3.1 Balancing Different Options

Once the root cause of the problem is identified, there are often multiple options to choose from, requiring a careful balance of their pros and cons. For example, when addressing the poor compatibility of InnoDB with NUMA, there are two solutions: the first is latch-free modification, which is complex, error-prone, and costly to maintain; the second is improving the speed of accessing critical latch resources, a simpler solution but not a definitive fix as it still exhibits issues under extremely high concurrency. Given the need to maintain compatibility with MySQL's official releases during modification, the second option was chosen, which was a prudent decision. As for scalability issues under high concurrency, implementing mechanisms such as transaction throttling can improve performance comparable to latch-free solutions.

3.3.2 Decompose the Problem

Complex problems often bring multiple associated issues that can significantly interfere with analysis and judgment. To effectively solve such problems, it's crucial to eliminate these disturbances, simplify the issue, and minimize the risk of misjudgment. For instance, when optimizing the Paxos algorithm in Group Replication, addressing other performance issues beforehand is essential to accurately gauge the impact of Paxos algorithm optimization.

Once all disturbances are eliminated, if the core problem remains overly complex, decomposing it further becomes necessary. For example, implementing dynamic programming to find the optimal join order involves breaking down this complex functionality into manageable steps. Here is another example of redo log optimization, as detailed in the figure below:

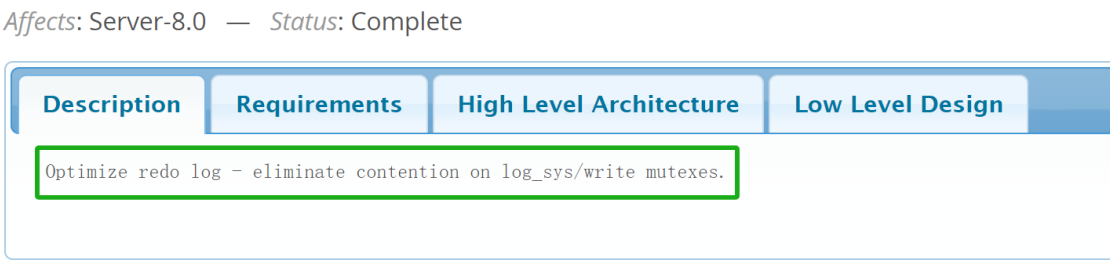


Figure 3-3. Description of the official worklog for redo log optimization.

To address the scalability issues of redo log, developers have subdivided this problem into the following sub-problems:

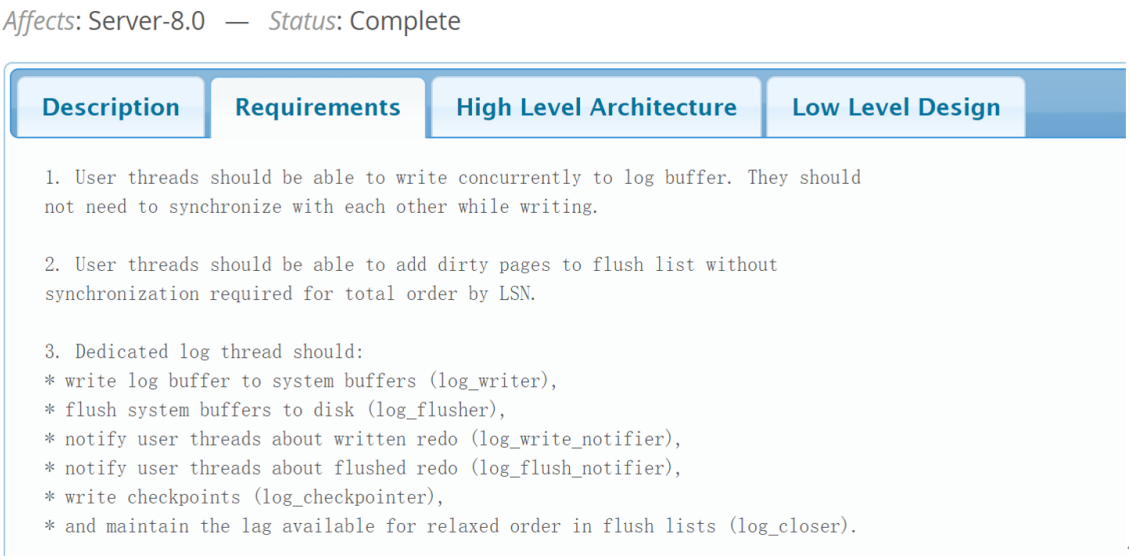


Figure 3-4. Requirements of the official worklog for redo log optimization.

This involves latch-free concurrent writing to the log buffer, dedicated threads for handling write and flush processes, and so on.

3.3.3 Seeking Theoretical Support

In the process of enhancing Group Replication, rooted in State Machine Replication (SMR) theory, it's crucial to begin by acquiring a substantial amount of theoretical knowledge through thorough paper reviews. This approach fosters a deeper understanding and maintains clarity throughout the improvement process, ensuring a comprehensive grasp of the concepts and preventing deviation from the intended path.

The book "Designing Data-Intensive Applications" discusses theory very well, specifically as follows [49]:

Although the theoretical papers and proofs are not always easy to understand, and sometimes make unrealistic assumptions, they are incredibly valuable for informing practical work in this field: they help us reason about what can and cannot be done, and help us find the counterintuitive ways in which distributed systems are often flawed. If you have the time, the references are well worth exploring.

It is worth mentioning that this book includes a large amount of scholarly material as the basis for

deductive reasoning and theoretical support.

3.3.4 Logic-based Testing

To assess the performance enhancement potential of a new feature, rigorous iterative testing and validation across various perspectives and environments are essential. Reliable conclusions ensure predictable outcomes in subsequent tests. Any deviations observed during validation must be thoroughly investigated to identify potential interfering factors.

For instance, consider the thread pool mechanism's impact on performance, particularly under high-concurrency testing conditions. Previous validations on MySQL version 5.7 demonstrated significant throughput improvements with the thread pool. However, subsequent modifications in MySQL 8.0 have revealed a different scenario. Case studies in Chapter 2.3 indicate a negative scalability impact of the Percona thread pool. MySQL's official technical blog confirms these findings, highlighting a decline in the thread pool's effectiveness over time [56].

> The MySQL Thread Pool serves as a solution to address the limitations of the default connection handling mechanism. The original design of the thread pool addressed the needs of MySQL 5.6, but as the MySQL server locking structures and algorithms have been improved over time the additional benefits provided by the original thread pool design have been reduced. The Max Transaction Limit feature in the MySQL 8.0 thread pool revitalizes the thread pool as a customer tool.

When conducting performance comparison tests, it's crucial to eliminate human errors such as changes in the environment or configurations. To mitigate this issue, repeating the initial test and comparing its results with those of the first test can help identify significant changes. If there are no noticeable differences, it suggests that the intermediate tests are relatively reliable.

For example, incorporating an additional test, depicted in the following figure, to compare its performance variance from the initial test.

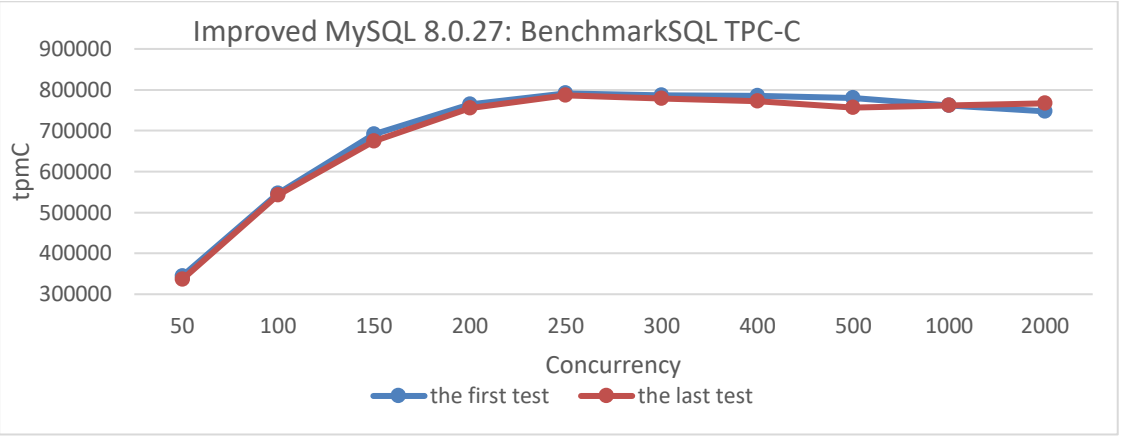


Figure 3-5. Additional test for comparing performance variance from the initial test.

Repeating the initial test can significantly mitigate issues of testing uncertainty and environmental contamination. However, this alone is not sufficient; it's also necessary to explain any abnormal results from each test. If an explanation cannot be provided and the issue is reproducible, it often presents a new opportunity for optimization.

For instance, during TPC-C testing, an anomaly in throughput was observed, which was later found to be due to omitting the use of jemalloc. It was discovered that performance was actually better without jemalloc. Extensive testing uncovered that MySQL's jemalloc 3.6 did not deliver optimal performance, prompting exploration into alternative jemalloc versions. These tests underscored the significant influence of memory allocation tools on overall performance.

3.4 Principles of Logical Reasoning

Here are principles of logical reasoning [27], which can provide some assistance in how to conduct logical inference:

(1) ask for reasons before accepting a conclusion,
(2) give an argument to support your conclusion,
(3) design your reasons to imply the conclusion,
(4) recognize the value of having more relevant information,
(5) weigh the pros and cons,
(6) consider the possible courses of action,
(7) look at the consequences of these various courses of action,
(8) evaluate the consequences,
(9) consider the probabilities that those various consequences will actually occur,
(10) delay making important decisions when practical,
(11) assess what is said in light of the situation,

(12) use your background knowledge and common sense in drawing conclusions,
(13) remember that extraordinary statements require extraordinarily good evidence,
(14) defer to the expert,
(15) remember that firmer conclusions require better reasons,
(16) be consistent in your own reasoning,
(17) be on the lookout for inconsistency in the reasoning of yourself and others,
(18) check to see whether explanations fit all the relevant facts,
(19) you can make your opponent's explanation less believable by showing that there are alternative explanations that haven't been ruled out,
(20) stick to the subject,
(21) don't draw a conclusion until you've gotten enough evidence.

3.5 Logical Reasoning: Key to Solving Complex Problems

At its core, the problem lies in a logical confusion. Clarifying these complex logical relationships requires precise data and continuously following clues through logical reasoning to uncover the root cause of the issue. Leveraging the power of this logical thinking can greatly enhance a programmer's problem-solving abilities.

While data structures and algorithms are central to programming, solving programmatic issues goes beyond merely being familiar or proficient in them. It requires logical reasoning skills. Therefore, the content of this chapter forms the foundation for subsequent topics.

3.6 The Difference Between Solving Difficult Problems and Doing Exercises

During exercises, essential information is typically provided, allowing students to apply textbook knowledge and logical reasoning to find solutions. This process helps develop problem-solving skills. However, upon entering the workforce and facing real programming challenges, individuals often feel unprepared. Schools rarely teach how to handle complex issues in programs like MySQL, leaving beginners bewildered and unsure where to start or how to find necessary information.

Complex problems can make programmers feel as if they are encountering supernatural phenomena, especially when the information they obtain is conflicting or misleading. In practice, solving challenging programming issues is markedly different from solving exercise problems. Even skilled teams may spend months sifting through distracting information to identify the root cause, which can lead to increasing frustration.

The main similarity between solving exercises and addressing real-world problems is logical reasoning. The difference lies in information accessibility: exercises provide essential details upfront,

while real-world problems require reasoning to uncover crucial, often obscured information. This process involves distinguishing truth from falsehood, filtering out distractions, and identifying genuinely relevant conditions for effective problem-solving.

In many cases, the difficulty of obtaining critical problem-solving information far surpasses the challenge of solving the problem itself.

3.7 Common Problem-Solving Frameworks

In the software industry, when faced with problems, the initial instinct is often to check if others have encountered similar issues. However, during the problem-solving process, individuals frequently rely on personal experience to propose solutions, sometimes without a solid foundation. Suspicions may point to network faults or configuration issues, prompting various actions that may only temporarily alleviate the problem without addressing its root cause. Both questioners and solvers often prioritize quick fixes, potentially neglecting the crucial role of logical reasoning.

Consider a real-life scenario: a DBA conducting MySQL performance tests uncovers unstable and low throughput. They provide details on CPU, memory, and configurations, noting thread-level IO with significant dirty page activity but omitting specific IO device information. Speculation about network issues is dismissed through DBA-conducted tests. Suggestions regarding spin delay configuration are similarly disproven. Such speculative approaches prove inefficient, failing to uncover underlying issues.

Professional problem-solving necessitates logical deduction based on observed phenomena to systematically narrow down the problem's scope. This involves analyzing usage patterns across varying levels of concurrency, attempting to replicate issues on different machine types, and reviewing past software versions for similar problems. These insights contribute to precise logical reasoning.

Returning to the DBA's case study, binding NUMA nodes on a high-spec machine adjusted CPU and memory resources to match user hardware specifications, while maintaining a high-spec disk. Tests subsequently demonstrated significantly improved and stable throughput. Ultimately, the critical issue was pinpointed to disparities in IO devices. This method effectively identifies root causes instead of relying solely on experiential or speculative methods.

Effective software problem-solving relies on logical reasoning. Below is a figure illustrating a common problem-solving framework.

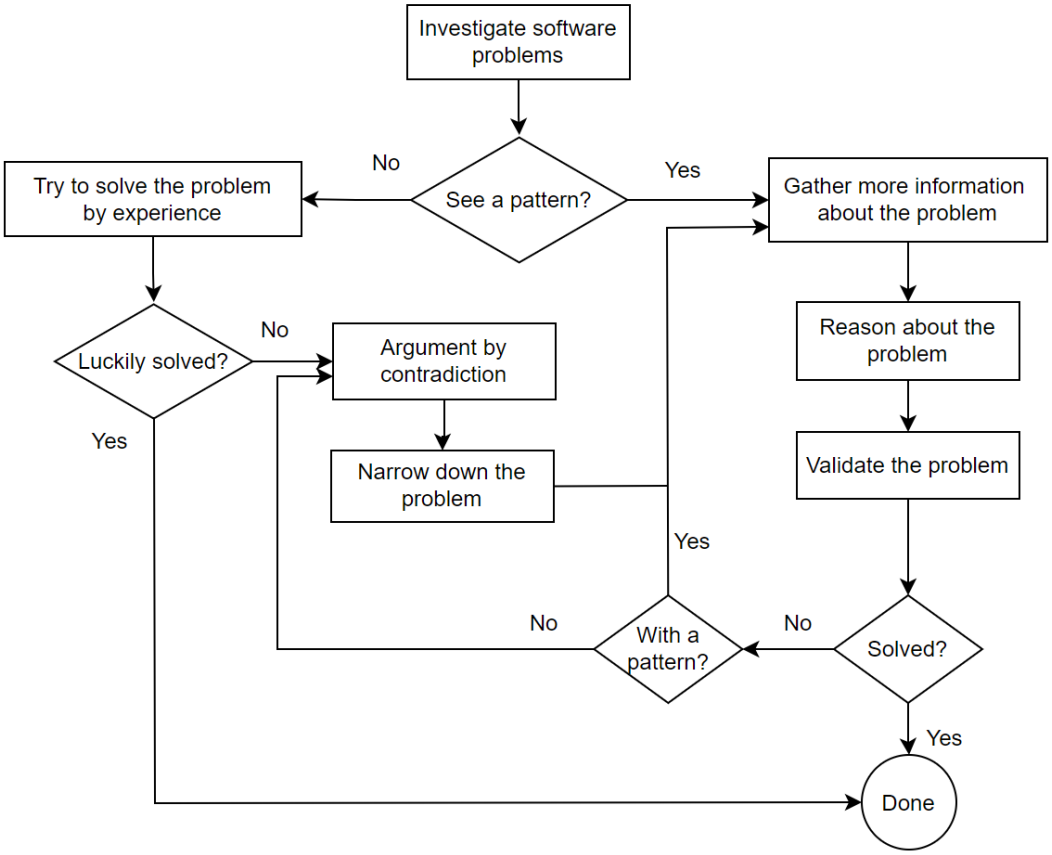


Figure 3-6. A general framework for solving program issues.

The figure integrates logical reasoning and information retrieval. Maintaining logical thinking ensures each step's reliability. Without it, one risks falling into traps that hinder effective software problem-solving.

Chapter 4: Fundamentals of Computer Science

To effectively address MySQL's myriad issues, a solid foundation in computer science is indispensable. MySQL not only encapsulates a wealth of computer science principles but also presents learners with diverse challenges, offering valuable practical experience. This chapter centers on essential computer science fundamentals necessary for resolving MySQL issues. The emphasis is on applying this foundational knowledge in practical scenarios, rather than theoretical teaching.

4.1 System Architecture

System architecture defines the high-level structure of a software or hardware system, detailing its components, their interrelationships, and how they collaborate to fulfill system objectives.

4.1.1 SMP

In an SMP (Symmetric Multiprocessing) system, multiple tightly-coupled processors share all resources such as bus, memory, and I/O system. This architecture facilitates equal access to memory, peripherals, and the operating system across all CPUs without distinction, emphasizing shared resource utilization among processors.

The following is a typical SMP architecture diagram.

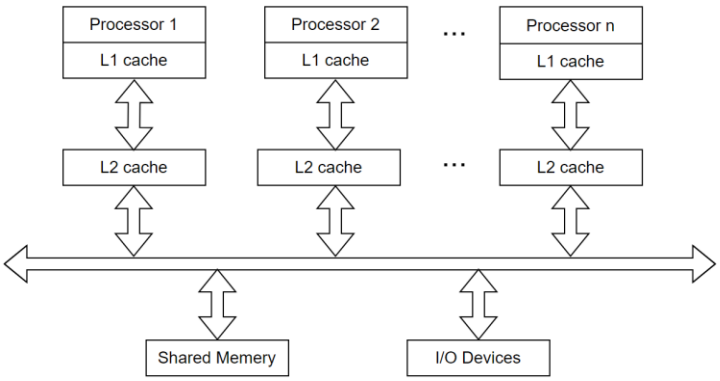


Figure 4-1. A typical SMP architecture.

In this setup, access to local L1 and L2 caches is extremely fast, and the overhead of CPU switching is negligible compared to NUMA architecture. SMP architecture typically supports moderate

throughput under normal conditions. However, it is the inability of SMP to scale effectively for higher throughput demands that led to the development of NUMA architecture.

4.1.2 NUMA

In the era of increasingly multicore systems, memory hierarchy is evolving towards non-uniform distributed architectures. Non-uniform memory architecture (NUMA) systems, which offer superior scalability compared to SMP counterparts, feature multiple memory nodes distributed throughout the system. Each node is physically adjacent to a subset of cores, but the entire physical address space of all nodes is globally visible, allowing cores to access memory locally or remotely. Consequently, data access times are non-uniform and vary based on data location. Accessing data from a remote node can lead to performance degradation due to latency and interconnect contention if many cores access large amounts of data remotely. These issues can be mitigated by co-locating threads with their data whenever possible, facilitated by NUMA-aware scheduling algorithms.

The throughput of the cross-chip interconnect is typically lower than that of on-chip memory controllers. Remote memory accesses that traverse this interconnect also experience higher latencies compared to local memory accesses. Due to the diversity in their memory interfaces, these multiprocessors are categorized as NUMA systems. The performance impact of remote memory accesses can be significant; in current implementations, the NUMA factor can result in up to a 2X slowdown for certain applications.

NUMA nodes are interconnected within the same physical server. When a CPU needs to access remote memory, it incurs a wait time. This limitation is fundamental to why NUMA servers struggle to achieve linear performance scalability as CPUs increase [95].

Here is the classic architecture figure of NUMA.

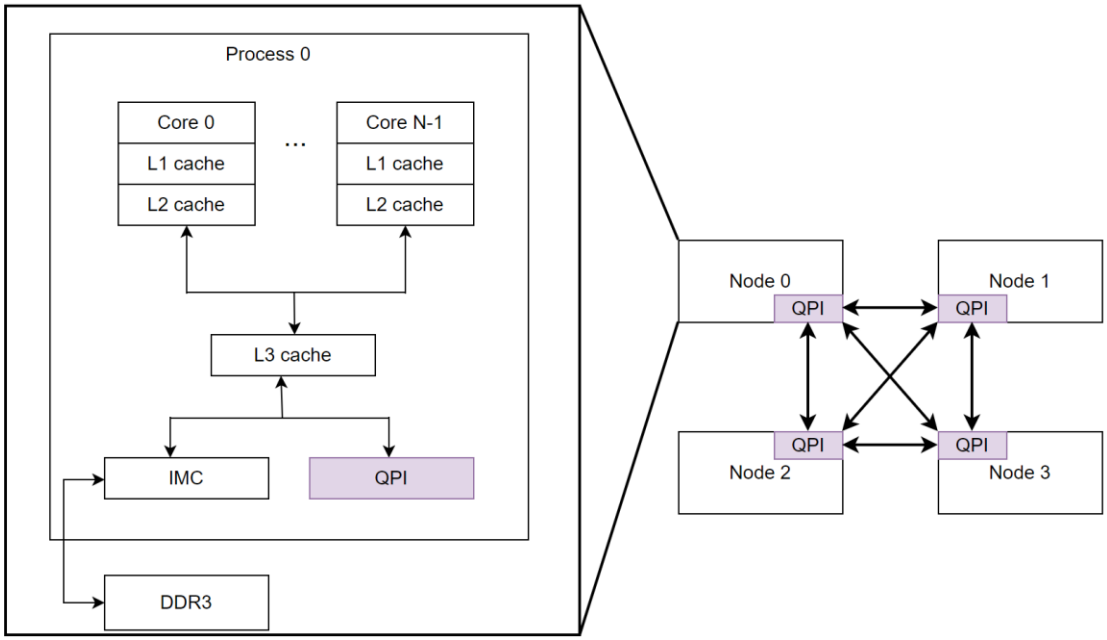


Figure 4-2. A typical NUMA architecture.

Access within the same NUMA node can be likened to a specialized form of SMP access, characterized by high efficiency primarily because of reduced CPU switching costs within the NUMA node. However, a drawback is that memory bandwidth within the same NUMA node can quickly become a bottleneck, constraining scalability. Accessing memory between different NUMA nodes involves higher costs, but it offers greater overall memory bandwidth. Effectively managing memory access in a NUMA environment poses a significant challenge.

The figure below illustrates the comparison results of TPC-C tests across different concurrency levels. The dark blue curve shows tests conducted with NUMA node 0 fixed, while the deep red curve represents tests utilizing all 4 NUMA nodes of the machine. The testing employed an enhanced version of MySQL, operating with 1000 warehouses and using the widely adopted Read Committed transaction isolation level.

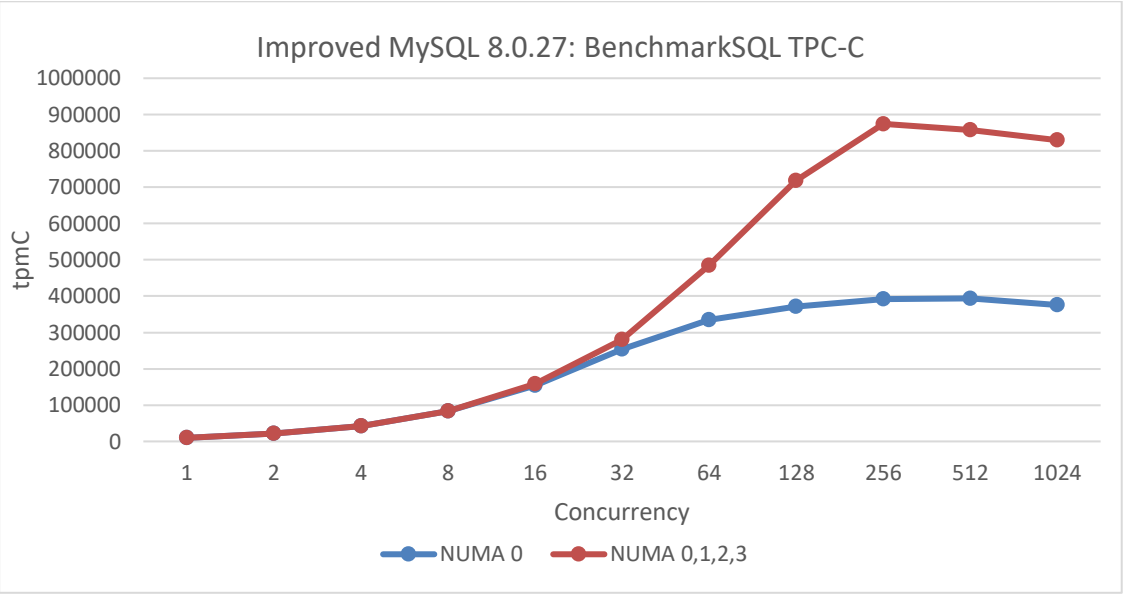


Figure 4-3. Performance Comparison in SMP vs. NUMA.

In the scenario where NUMA node 0 is bound, the throughput versus concurrency curve is notably smooth. Even under high concurrency, there is only a slight decline in throughput, indicating low thread context switching costs. However, throughput consistently remains below 400,000 tpmC due to significant limitations in memory bandwidth, characteristic of traditional SMP architecture.

In contrast, when utilizing all NUMA nodes, the throughput curve is relatively worse. This is attributed to reduced memory efficiency and increased context switching costs when accessing across NUMA nodes, resulting in less stable throughput. Nevertheless, scalability is greatly improved, with peak throughput increasing by 123% compared to using a single NUMA node.

Utilizing 4 NUMA nodes does not result in a fourfold increase in performance but rather approximately a 2.x times increase. This non-linear scaling is inherent to NUMA architecture and is compounded by MySQL's challenges in linear scalability.

MySQL's difficulties in achieving linear scalability arise from several factors, including the Read Committed transaction isolation mechanism based on MVCC ReadView. This involves copying a global ReadView to a local one, leading to contention among threads for global ReadView updates. Moreover, in NUMA environments, frequent cross-NUMA node accesses are necessary, further complicating scalability.

Many pieces of code are not suitable for NUMA environments. For example, frequent latch contention

in critical sections can lead to frequent cross-NUMA node context switches. Slow operations within critical sections can cause inefficient program execution due to frequent cache migrations across NUMA nodes, necessitating data movement between caches.

To achieve optimal performance on NUMA systems [7], the following strategies are crucial:

1. Maximize the proportion of memory accesses routed to local nodes.
2. Balance traffic across nodes and interconnect links.

An unbalanced distribution of memory requests can significantly increase memory access latency on overloaded controllers, sometimes reaching up to 1000 cycles compared to approximately 200 cycles on non-overloaded controllers.

While Microsoft SQL Server and Oracle DBMS are NUMA-aware, MySQL is not [11]. Therefore, large-scale applications like MySQL, which lack NUMA awareness, offer significant optimization potential in NUMA environments.

Interestingly, for MySQL, disabling NUMA can potentially improve performance. According to a case study in Section 2.9, disabling NUMA configuration in the BIOS enhanced performance for the MySQL primary. This adjustment impacts memory allocation strategies, leading to more consistent memory access between NUMA nodes. This aligns with optimization strategy 2 mentioned above, benefiting MySQL's throughput and stability.

4.1.3 X86 Architecture

X86 uses a complex system called CISC (Complex Instruction Set Computing). This can do a lot of tasks at once but makes the processor more complicated and expensive to create. X86 architectures allow more direct interaction with memory, facilitating a depth of computational tasks at the expense of higher power consumption [81]. Overall, x86 has higher performance capabilities, ideal for demanding computational tasks.

For mainstream NUMA machines utilizing the x86 architecture, please refer to the figure below for an analysis of memory access overhead between different NUMA nodes on a specific machine:

```
node distances:
node   0   1   2   3
  0:  10  21  21  21
  1:  21  10  21  21
  2:  21  21  10  21
  3:  21  21  21  10
```

Figure 4-4. Node distances in a typical x86 architecture.

From the figure, it is evident that under the x86 architecture, the access overhead is 10 units for accessing the local node and 21 units for accessing remote NUMA nodes. Below are the TPC-C test results of improved MySQL 8.0.27 with BenchmarkSQL:

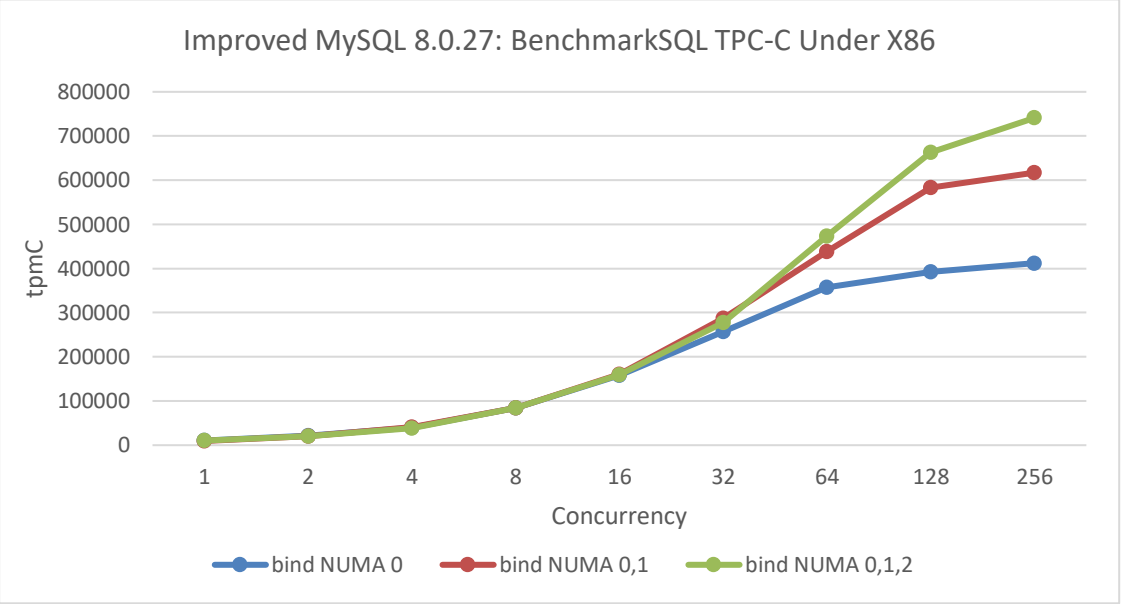


Figure 4-5. Performance comparison under different NUMA bindings in a typical x86 architecture.

Under the x86 architecture, the addition of each NUMA node results in a noticeable increase in high-concurrency throughput. This highlights the robust performance capabilities of x86 architecture in effectively managing memory access across NUMA nodes, thereby leveraging its full potential for high-performance computing.

4.1.4 ARM Architecture

Historically, ARM processors have prioritized power efficiency, dominating the mobile systems market, whereas x86 processors have led in high-performance computing. The primary distinction

between ARM and x86 processors lies in their instruction sets: ARM utilizes the RISC (Reduced Instruction Set Computing) architecture, which simplifies instructions to enhance speed and energy efficiency. This simplicity makes ARM ideal for battery-powered devices like smartphones [87]. In contrast, x86 employs the CISC (Complex Instruction Set Computing) architecture, which supports a wider range of complex operations but typically consumes more power.

Regarding memory handling, ARM processors focus on register-based processing to minimize direct memory access, thereby improving energy efficiency. In contrast, x86 architectures allow for more direct interaction with memory, enabling a broader range of computational tasks at the cost of higher power consumption. Servers based on ARM architecture are renowned for their low power consumption and cost-effectiveness. However, they generally exhibit lower overall performance compared to x86 architecture and may have limitations in memory access scalability.

For mainstream NUMA machines utilizing the ARM architecture, please refer to the figure below to observe the memory access overhead between different NUMA nodes on a specific machine:

```
node distances:
node   0   1   2   3
  0:  10  16  32  33
  1:  16  10  25  32
  2:  32  25  10  16
  3:  33  32  16  10
```

Figure 4-6. Node distances in a typical ARM architecture.

The memory access overhead between different NUMA nodes under the ARM architecture is notably more complex and varies significantly compared to the x86 architecture.

Here are the test results on an ARM machine, using MySQL code and configuration similar to those on an x86 machine, with approximately the same settings, including 1000 warehouses.

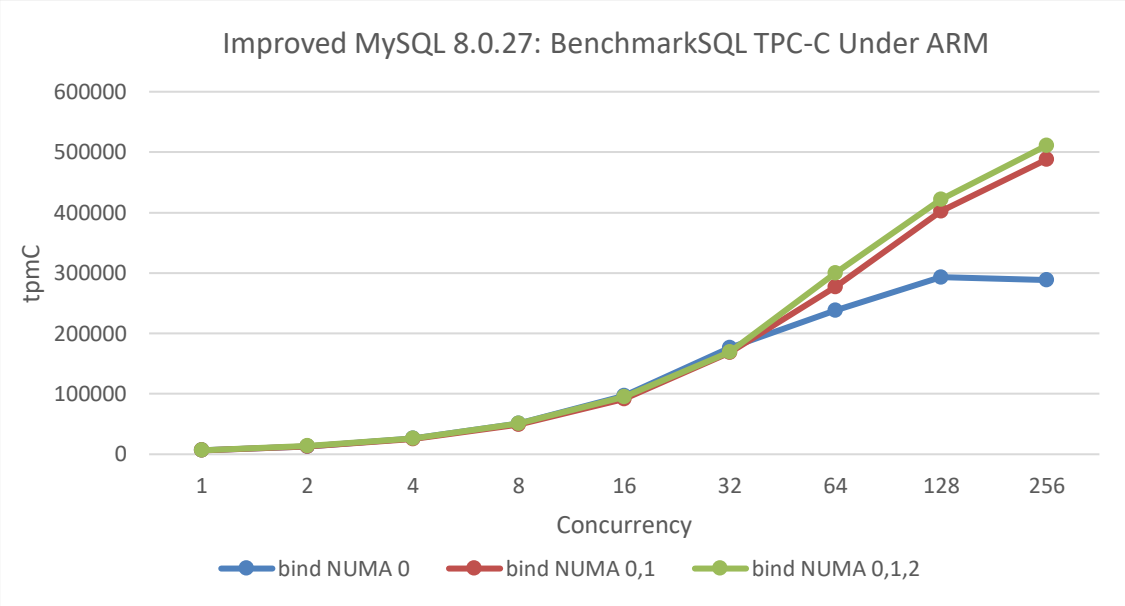


Figure 4-7. Performance comparison under different NUMA bindings in a typical ARM architecture.

From the figure, it is evident that binding NUMA node 0 and 1 significantly improves throughput. However, adding NUMA node 2 does not noticeably enhance throughput, primarily due to NUMA node distances inherent in the ARM architecture. Extensive testing has revealed that MySQL demonstrates better scalability on x86 architecture compared to ARM.

4.2 Data Structure

This section explores the fundamental data structures in MySQL, encompassing arrays, linked lists, queues, heaps, hash tables, red-black trees, B+ trees, and hybrid data structures. These data structures do not inherently possess advantages or disadvantages; their effectiveness depends on their application tailored to specific system architectures and the characteristics of practical data.

4.2.1 Array

An array consists of elements arranged in a specific order, typically of the same type. Elements are accessed via an integer index to specify the required item. Arrays are usually implemented with contiguous memory allocation and can be either fixed-length or resizable [81]. In MySQL, arrays commonly used include dynamic vectors and fixed-length arrays, with the choice depending on specific needs. Vectors can dynamically resize, while fixed-length arrays have a predetermined size.

In the MySQL InnoDB storage engine, MVCC ReadView employs a data structure akin to a vector to store transaction IDs of all active transactions. This dynamic array supports varying lengths, adapting to changes in the active transaction list despite size fluctuations. For the Read Committed transaction isolation level, each read operation utilizes its own ReadView.

Here are some details about the ReadView object.

```
private:
 // Disable copying
 ReadView(const ReadView &);
 ReadView &operator=(const ReadView &);

private:
 /** The read should not see any transaction with trx id >= this
 value. In other words, this is the "high water mark". */
 trx_id_t m_low_limit_id;

 /** The read should see all trx ids which are strictly
 smaller (<) than this value.  In other words, this is the
 low water mark". */
 trx_id_t m_up_limit_id;

 /** trx id of creating transaction, set to TRX_ID_MAX for free
 views. */
 trx_id_t m_creator_trx_id;

 /** Set of RW transactions that was active when this snapshot
 was taken */
 ids_t m_ids;

 /** The view does not need to see the undo logs for transactions
 whose transaction number is strictly smaller (<) than this value:
 they can be removed in purge if not needed by other views */
 trx_id_t m_low_limit_no;
```

The variable m_ids is a data structure of type ids_t, which closely resembles ***std::vector***. For more details, see below:

```
/** This is similar to a std::vector but it is not a drop
in replacement. It is specific to ReadView. */
class ids_t {
  typedef trx_ids_t::value_type;

  /**
```

```
  Constructor */
  ids_t() : m_ptr(), m_size(), m_reserved() {}
  /**

  Destructor */
  ~ids_t() { ut::delete_arr(m_ptr); }

  /** Try and increase the size of the array. Old elements are copied across.
  It is a no-op if n is < current size.
  @param n          Make space for n elements */
     void reserve(ulint n);
```

Do fixed-length arrays have practical value? In MySQL, buffer pool chunks are organized using fixed-length arrays. Details are provided below:

```
/** @brief The buffer pool structure.

NOTE! The definition appears here only for other modules of this
directory (buf) to see it. Do not use from outside! */

struct buf_pool_t {
  ...
  /** Number of buffer pool chunks */
  volatile ulint n_chunks;

  /** New number of buffer pool chunks */
  volatile ulint n_chunks_new;

  /** buffer pool chunks */
  buf_chunk_t *chunks;

  /** old buffer pool chunks to be freed after resizing buffer pool */
  buf_chunk_t *chunks_old;

  /** Current pool size in pages */
  ulint curr_size;

  /** Previous pool size in pages */
  ulint old_size;

  /** Size in pages of the area which the read-ahead algorithms read
  if invoked */
  page_no_t read_ahead_area;

  /** Hash table of buf_page_t or buf_block_t file pages, buf_page_in_file() ==
```

```
true, indexed by (space_id, offset).  page_hash is protected by an array of
mutexes. */
hash_table_t *page_hash;
```

Above, the array name and type are defined, while below, dynamic memory allocation is carried out based on the array's member type.

```
buf_pool->chunks = reinterpret_cast<buf_chunk_t *>(ut::zalloc_withkey(
    UT_NEW_THIS_FILE_PSI_KEY, buf_pool->n_chunks * sizeof(*chunk)));
```

From a practical standpoint, leveraging fixed-length arrays can offer substantial performance benefits. Their stability prevents performance fluctuations due to memory reallocation, and their cache-friendliness further enhances efficiency. Subsequent chapters will include several examples where the use of fixed-length arrays significantly improves performance or alleviates performance bottlenecks.

4.2.2 Linked List

A linked list (or simply "list") is a linear collection of data elements, called nodes, where each node contains a value and a reference to the next node in the sequence. The primary advantage of linked lists over arrays is their efficiency in inserting and removing elements without relocating the entire list. However, operations like random access to a specific element are generally slower with linked lists compared to arrays [81].

MySQL commonly uses the list from the standard library, which typically implements a doubly linked list to facilitate easy insertion and deletion, though it often suffers from poor query performance. In mainstream NUMA architectures, linked lists are generally inefficient for querying due to non-contiguous memory access patterns. Consequently, linked lists are best suited as auxiliary data structures or for scenarios involving smaller data volumes.

Below is the list data structure used by ***undo***. As the ***undo*** list grows longer, MVCC efficiency is significantly reduced.

```
using Recs = std::list<rec_t, mem_heap_allocator<rec_t>>;
...
/** Undo recs to purge */
Recs *recs;
```

To address this issue, some databases adopt centralized storage for undo history versions, which significantly reduces the cost of garbage collection.

4.2.3 Queue

In computer science, a queue is a collection of entities organized in a sequence, where entities can be added at one end and removed from the other. The operation of adding an element to the rear is called enqueue, while removing an element from the front is called dequeue. This makes a queue a first-in-first-out (FIFO) data structure, meaning the first element added will be the first one removed. In other words, elements are processed in the order they are added.

Queues are linear data structures, or sequential collections, and are commonly used in computer programs. They can be implemented using circular buffers or linked lists. In MySQL, queues are often encapsulated with additional functionalities, such as synchronized queues and double-ended queues, for FIFO processing needs. For instance, the incoming member shown below uses a synchronized queue to store Group Replication's applier packets, serving as a cache for data related to Paxos network interactions and relay log disk writes. This buffering helps manage data when relay log writing lags behind.

```
/* The incoming event queue */
Synchronized_queue<Packet *> *incoming;
```

Double-ended queues are commonly used in various applications. For instance, MySQL utilizes std::deque to implement a general-purpose mem_root_deque. Details are provided below:

```
/**
  A (partial) implementation of std::deque allocating its blocks on a MEM_ROOT.

  This class works pretty much like an std::deque with a Mem_root_allocator,
  and used to be a forwarder to it. However, libstdc++ has a very complicated
  implementation of std::deque, leading to code blowup (e.g., operator[] is
  23 instructions on x86-64, including two branches), and we cannot easily use
  libc++ on all platforms. This version is instead:

   - Optimized for small, straight-through machine code (few and simple
     instructions, few branches).
   - Optimized for few elements; in particular, zero elements is an important
     special case, much more so than 10,000.

  ...
 */
template <class Element_type>
class mem_root_deque {
```

4.2.4 Heap

In computer science, a heap is a tree-based data structure that maintains the heap property and is typically implemented using an array [81]. It serves as an efficient implementation of the abstract data type known as a priority queue. Priority queues are often referred to as "heaps" regardless of their underlying implementation. In a heap, the element with the highest (or lowest) priority is always at the root. However, unlike a sorted structure, a heap is partially ordered.

Heaps are particularly useful when there is a need to repeatedly access and remove the element with the highest or lowest priority, or when insertions and removals of the root node occur frequently. Priority queues, which are frequently implemented with heaps, are also used in MySQL. For instance, in MySQL version 8.0.34, the data structure purge_pg_t (detailed below) utilizes the priority_queue from the standard library to efficiently find the oldest transaction ID.

```
typedef std::priority_queue<
   TrxUndoRsegs, std::vector<TrxUndoRsegs, ut::allocator<TrxUndoRsegs>>,
   TrxUndoRsegs>
      purge_pq_t;
```

From a mathematical perspective, heap data structures have a balanced tree structure with minimal theoretical tree levels. However, in modern architectures, they present notable drawbacks. Heaps have a non-sequential access pattern, moving from the root to the leaves, which is not cache-friendly. This makes heaps suitable for relatively small datasets but less efficient as data scales up. The inefficiency in cache access could explain why heap-based algorithms, such as heap sort, often do not consistently outperform quicksort in average performance metrics despite their theoretical advantages.

4.2.5 Hash Table

A hash table, also known as a hash map, is a data structure designed for fast value retrieval based on keys. It uses a hashing function to map keys to indices in an array, allowing for average constant-time access. Hash tables are commonly used in dictionaries, caches, and database indexing. Despite their efficiency, hash collisions can degrade performance, and techniques such as chaining and open addressing are used to manage them.

The primary advantage of hash tables is their rapid query speed, but they can be less cache-friendly due to the dispersed memory pointers stored in hash slots. This dispersion can lead to inefficiencies during frequent access operations.

In MySQL, hash tables are widely used, leveraging both STL types like unordered_set and

unordered_map, as well as custom-designed hash tables tailored to specific use cases. For instance, the hash_table_t data type, used in the buffer pool for page management, exemplifies such specialized implementations.

```
/** Hash table of buf_page_t or buf_block_t file pages, buf_page_in_file() ==
true, indexed by (space_id, offset).  page_hash is protected by an array of
mutexes. */
hash_table_t *page_hash;
```

The data members of hash_table_t are as follows:

```
/* The hash table structure */
class hash_table_t {
 public:
  hash_table_t(size_t n) {
    const auto prime = ut::find_prime(n);
    cells = ut::make_unique<hash_cell_t[]>(prime);
    set_n_cells(prime);

    /* Initialize the cell array */
    hash_table_clear(this);
  }
  ~hash_table_t() { ut_ad(magic_n == HASH_TABLE_MAGIC_N); }

  /** Returns number of cells in cells[] array.
   If type==HASH_TABLE_SYNC_RW_LOCK it can be used:
  - without any latches to peek a value, before hash_lock_[sx]_confirm
  - when holding S-latch for at least one n_sync_obj to get the "real" value
  @return value of n_cells
  */
  size_t get_n_cells() { return n_cells.load(std::memory_order_relaxed); }

  /** Returns a helper class for calculating fast modulo n_cells.
   If type==HASH_TABLE_SYNC_RW_LOCK it can be used:
  - without any latches to peek a value, before hash_lock_[sx]_confirm
  - when holding S-latch for at least one n_sync_obj to get the "real" value */
  const ut::fast_modulo_t get_n_cells_fast_modulo() {
    return n_cells_fast_modulo.load();
  }
  ...
```

The core certification database of Group Replication uses the std::unordered_map hash table to handle a large volume of certification information.

```
typedef std::unordered_map<
    std::string, Gtid_set_ref *, std::hash<std::string>,
    std::equal_to<std::string>,
    Malloc_allocator<std::pair<const std::string, Gtid_set_ref *>>>
    Certification_info;
...
/**
  Certification database.
*/
Certification_info certification_info;
```

Despite utilizing efficient data structures like hash tables, the system faces performance challenges due to frequent access to elements. This issue arises because the memory access pattern in the certification database is non-contiguous, leading to inefficient memory access. Consequently, while hash tables offer advantages, they are not always the optimal choice for performance in this context.

4.2.6 Red-Black Tree

In computer science, a red-black tree is a self-balancing binary search tree known for efficient storage and retrieval of ordered data. Each node in a red-black tree has an additional "color" bit, typically red or black, which helps maintain the tree's balanced structure. MySQL frequently uses the map from the STL (Standard Template Library), which implements a red-black tree to preserve order. In contrast, unordered_map in the STL is a hash table and does not maintain order, which is why it is called an "unordered map."

The Pages below illustrates the use of a map data structure for efficient lookup, modification, and sequential traversal.

```
/* Assuming a page size, read the space_id from each page and store it
in a map. Find out which space_id is agreed on by majority of the
pages.  Choose that space_id. */
for (uint32_t page_size = UNIV_ZIP_SIZE_MIN; page_size <= UNIV_PAGE_SIZE_MAX;
     page_size <<= 1) {
  /* map[space_id] = count of pages */
  typedef std::map<space_id_t, ulint, std::less<space_id_t>,
                   ut::allocator<std::pair<const space_id_t, ulint>>>
      Pages;

  Pages verify;
  ulint page_count = 64;
  ulint valid_pages = 0;
```

```
  /* Adjust the number of pages to analyze based on file size */
  while ((page_count * page_size) > file_size) {
    --page_count;
  }
```

MySQL has also implemented a red-black tree tailored to its specific needs. For instance, the code snippet below shows SEL_ROOT, a red-black tree utilized to store key ranges.

```
/**
  A graph of (possible multiple) key ranges, represented as a red-black
  binary tree. There are three types (see the Type enum); if KEY_RANGE,
  we have zero or more SEL_ARGs, described in the documentation on SEL_ARG.

  As a special case, a nullptr SEL_ROOT means a range that is always true.
  This is true both for keys[] and next_key_part.
*/
class SEL_ROOT {
 ...
  /**
    Insert the given node into the tree, and update the root.
    @param key The node to insert.
  */
  void insert(SEL_ARG *key);

  /**
    Delete the given node from the tree, and update the root.
    @param key The node to delete. Must exist in the tree.
  */
  void tree_delete(SEL_ARG *key);

  /**
    Find best key with min <= given key.
    Because of the call context, this should never return nullptr to get_range.
    @param key The key to search for.
  */
  SEL_ARG *find_range(const SEL_ARG *key) const;
    ...
```

Generally, red-black trees offer advantages such as low insertion and update costs and support for sequential traversal. However, their non-sequential memory access can reduce cache efficiency, making them less ideal for high-performance, compute-intensive tasks.

4.2.7 B+ Tree

A B+ tree is an m-ary tree characterized by a large number of children per node, including a root, internal nodes, and leaves. The root may either be a leaf or a node with two or more children [81].

B+ trees excel in block-oriented storage contexts, such as filesystems, due to their high fanout (typically around 100 or more pointers per node). This high fanout reduces the number of I/O operations needed to locate an element, making B+ trees especially efficient when data cannot fit into memory and must be read from disk.

InnoDB employs B+ trees for its indexing, leveraging their ability to ensure a fixed maximum number of reads based on the tree's depth, which scales efficiently. For specific details on B+ tree implementation in MySQL, refer to the file btr/btr0btr.cc.

4.2.8 Hybrid Data Structure

In various application scenarios, relying on a single data structure may not always yield optimal performance. Combining different data structures can often lead to significant improvements. For instance, in MySQL, the MVCC ReadView initially uses a dynamic array (vector) to maintain the active transaction list, utilizing binary search for querying. However, in high-concurrency environments, this list can grow excessively, making it less efficient in NUMA environments. To mitigate this issue, a hybrid approach is employed: recent transactions are stored in a static array for quick access, while long-running transactions are placed in a dynamic array. This dual-array strategy, managed by multiple variables, enhances access speed and efficiency. For further details, see below:

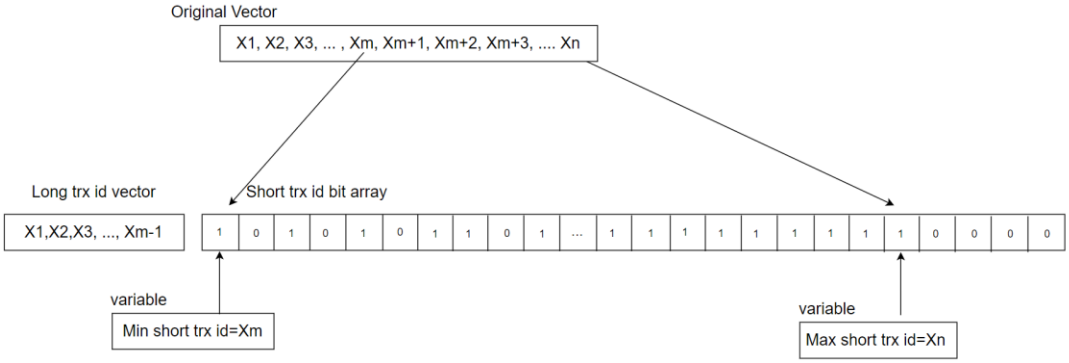


Figure 4-8. A new hybrid data structure suitable for active transaction list in MVCC ReadView.

To better illustrate the concept of hybrid data structures, consider the following example:

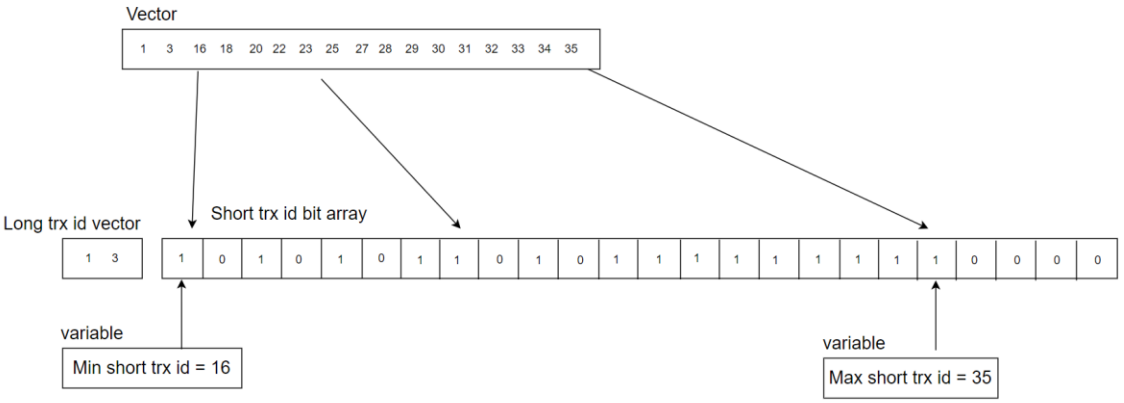


Figure 4-9. A detailed example for the new hybrid data structure for active transaction list.

The active transaction list length is 17, with each transaction ID requiring 8 bytes. Storing this using a dynamic array (vector) would necessitate at least 17 * 8 = 136 bytes. By converting to a hybrid data structure, transaction IDs are stored in a static array with bit representation, requiring only 3 bytes. The dynamic array then holds two transaction IDs (1 and 3), taking up 16 bytes. Additionally, two auxiliary variables consume 16 bytes. Consequently, the hybrid data structure totals 3 + 16 + 16 = 35 bytes, which is 101 bytes less than the original approach.

Regarding query efficiency, the hybrid data structure offers substantial improvements. For instance, to check if transaction ID=24 is in the active transaction list:

- In the original approach, a binary search is needed, with a time complexity of O(log n).
- With the hybrid structure, using the minimum short transaction ID as a baseline allows direct querying through the static data, achieving a time complexity of O(1).

In NUMA environments, as shown in the figure below, it can be seen that simply changing the data structure can significantly increase the throughput of TPC-C under high-concurrency conditions, greatly alleviating scalability issues related to MVCC ReadView.

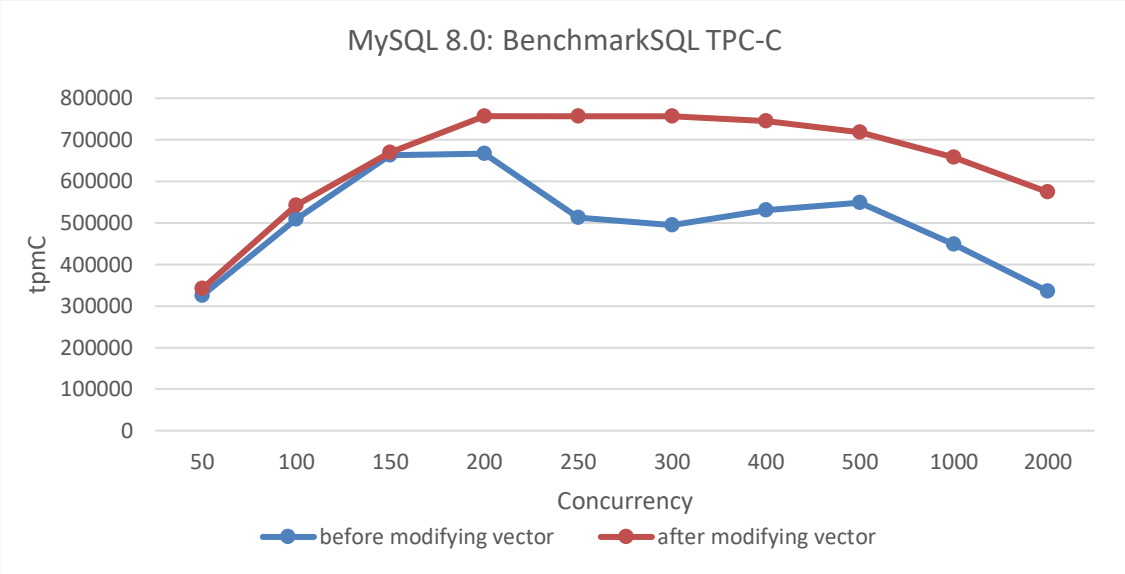


Figure 4-10. Performance improvement with new hybrid data structure in NUMA.

4.3 Algorithm

This section covers various problem-solving algorithms in MySQL, including search algorithms, sorting algorithms, greedy algorithms, dynamic programming, amortized analysis, and the Paxos series of algorithms.

4.3.1 Search Algorithm

In computer science, a search algorithm is designed to locate information within a specific data structure or search space [81].

MySQL commonly employs several search algorithms, including binary search, red-black tree search, B+ tree search, and hash search. Binary search is particularly effective for searching datasets without special characteristics or when dealing with smaller datasets. For instance, binary search is used to determine if a record is visible in the active transaction list, as detailed below:

```
/** Check whether the changes by id are visible.
@param[in]    id      transaction id to check against the view
@param[in]    name    table name
@return whether the view sees the modifications of id. */
[[nodiscard]] bool changes_visible(trx_id_t id,
                                   const table_name_t &name) const {
  ut_ad(id > 0);
```

```
  if (id < m_up_limit_id || id == m_creator_trx_id) {
    return (true);
  }
  check_trx_id_sanity(id, name);
  if (id >= m_low_limit_id) {
    return (false);
  } else if (m_ids.empty()) {
    return (true);
  }
  const ids_t::value_type *p = m_ids.data();
  return (!std::binary_search(p, p + m_ids.size(), id));
}
```

The B+ tree search, implemented in btr/btr0btr.cc, is a well-established method ideal for indexing scenarios. Hash lookup, another prevalent search method in MySQL, is frequently used, such as in Group Replication for certification databases, as detailed below:

```
bool Certifier::add_item(const char *item, Gtid_set_ref *snapshot_version,
                         int64 *item_previous_sequence_number) {
  DBUG_TRACE;
  mysql_mutex_assert_owner(&LOCK_certification_info);
  bool error = true;
  std::string key(item);
  Certification_info::iterator it = certification_info.find(key);
  snapshot_version->link();
  if (it == certification_info.end()) {
    std::pair<Certification_info::iterator, bool> ret =
        certification_info.insert(
            std::pair<std::string, Gtid_set_ref *>(key, snapshot_version));
    error = !ret.second;
  } else {
    *item_previous_sequence_number =
        it->second->get_parallel_applier_sequence_number();
    if (it->second->unlink() == 0) delete it->second;
    it->second = snapshot_version;
    error = false;
  }
  ...
  return error;
}
```

The choice of search algorithm is flexible and usually adjusted based on performance bottlenecks. For instance, the hash-based search algorithm highlighted earlier emerged as the primary bottleneck during MySQL secondary operations, as illustrated by the specific performance flame graph below:

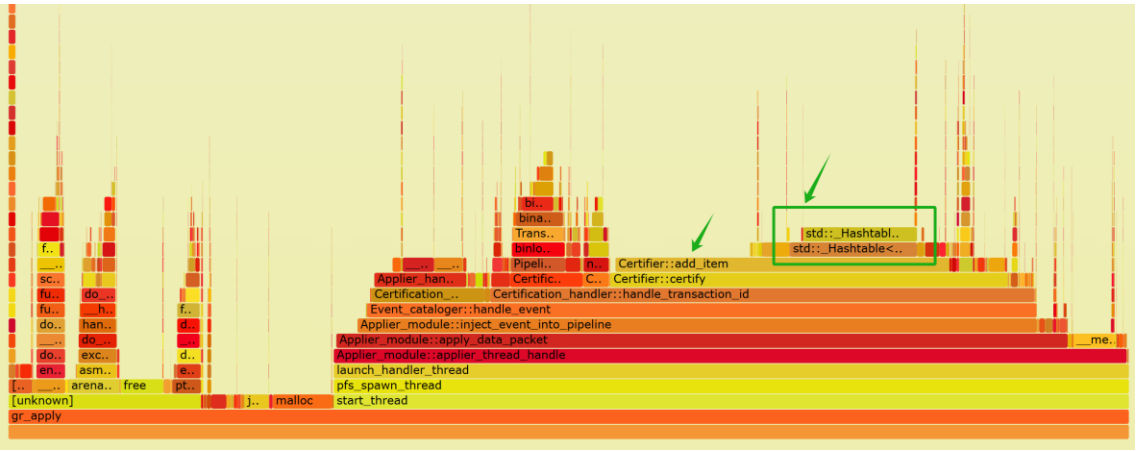


Figure 4-11. An example of bottlenecks in hash-based search algorithm.

The figure shows that hash search in Certify::add_item accounts for half of the total time, highlighting a significant bottleneck. This suggests the need to explore alternative search algorithms. Further details on potential solutions are discussed in the following chapters.

4.3.2 Sorting Algorithm

In computer science, a sorting algorithm arranges elements of a list into a specific order, with the most frequently used orders being numerical and lexicographical, either ascending or descending. Efficient sorting is crucial for optimizing the performance of other algorithms, such as search and merge algorithms, which require sorted input data [81].

Commonly used sorting algorithms do not inherently have a distinction of superiority or inferiority; each serves its own purpose. For example, consider *std::sort* from the C++ Standard Library. It employs different sorting algorithms based on the data's characteristics. When the data is mostly ordered, it uses insertion sort. See details below:

```
// sort
template <typename _RandomAccessIterator, typename _Compare>
_GLIBCXX20_CONSTEXPR inline void __sort(_RandomAccessIterator __first,
                                        _RandomAccessIterator __last,
                                        _Compare __comp) {
  if (__first != __last) {
    std::__introsort_loop(__first, __last, std::__lg(__last - __first) * 2,
                          __comp);
    std::__final_insertion_sort(__first, __last, __comp);
  }
```

```
}
```

Insertion sort is used in cases where the data is mostly ordered because it has a time complexity close to linear in such scenarios. More importantly, insertion sort accesses data sequentially, which significantly improves cache efficiency. This cache-friendly nature makes insertion sort highly suitable for sorting small amounts of data.

For instance, in the ***std::sort*** function, when calling the ***__introsort_loop*** function, if the number of elements is less than or equal to 16, sorting is skipped, and control returns to the sort function. The sort function then utilizes insertion sort for sorting.

```
/**
  *  @doctodo
  *  This controls some aspect of the sort routines.
 */
 enum { _S_threshold = 16 };

/// This is a helper function for the sort routine.
template <typename _RandomAccessIterator, typename _Size, typename _Compare>
_GLIBCXX20_CONSTEXPR void __introsort_loop(_RandomAccessIterator __first,
                                           _RandomAccessIterator __last,
                                           _Size __depth_limit,
                                           _Compare __comp) {
  while (__last - __first > int(_S_threshold)) {
    if (__depth_limit == 0) {
      std::__partial_sort(__first, __last, __last, __comp);
      return;
    }
    --__depth_limit;
    _RandomAccessIterator __cut =
        std::__unguarded_partition_pivot(__first, __last, __comp);
    std::__introsort_loop(__cut, __last, __depth_limit, __comp);
    __last = __cut;
  }
}
```

Within the ***__introsort_loop*** function, if the recursion depth exceeds a threshold, the ***partial_sort*** function, which is based on heap data structures and offers stable performance, is utilized. Overall, the main part of ***__introsort_loop*** employs an improved version of quicksort, eliminating left recursion and reducing the overhead of function calls.

Discussing sorting algorithms is essential not only because MySQL extensively uses these standard

library sort algorithms but also to draw insights on optimization strategies from their implementations. This involves selecting different algorithms based on specific circumstances to leverage their respective strengths, thereby enhancing the overall performance of the algorithm.

In MySQL code, similar optimization principles are applied. For instance, in the ***sort_buffer*** function, when the ***key_len*** value is small, it uses the ***Mem_compare*** function, which is suitable for short keys. When ***prefilter_nth_element*** > 0, it employs ***nth_element*** (similar to the partitioning idea of ***std::sort***'s ***quicksort***), selecting the required elements for subsequent sorting.

```
size_t Filesort_buffer::sort_buffer(Sort_param *param, size_t num_input_rows,
                                    size_t max_output_rows) {
 ...
 if (num_input_rows <= 100) {
  if (key_len < 10) {
   param->m_sort_algorithm = Sort_param::FILESORT_ALG_STD_SORT;
   if (prefilter_nth_element) {
    nth_element(it_begin, it_begin + max_output_rows - 1, it_end,
                Mem_compare(key_len));
    it_end = it_begin + max_output_rows;
   }
   sort(it_begin, it_end, Mem_compare(key_len));
   ...
   return std::min(num_input_rows, max_output_rows);
  }
  param->m_sort_algorithm = Sort_param::FILESORT_ALG_STD_SORT;
  if (prefilter_nth_element) {
   nth_element(it_begin, it_begin + max_output_rows - 1, it_end,
               Mem_compare_longkey(key_len));
   it_end = it_begin + max_output_rows;
  }
  sort(it_begin, it_end, Mem_compare_longkey(key_len));
  ...
  return std::min(num_input_rows, max_output_rows);
 }

 param->m_sort_algorithm = Sort_param::FILESORT_ALG_STD_STABLE;
 // Heuristics here: avoid function overhead call for short keys.
 if (key_len < 10) {
  if (prefilter_nth_element) {
   nth_element(it_begin, it_begin + max_output_rows - 1, it_end,
               Mem_compare(key_len));
   it_end = it_begin + max_output_rows;
  }
  stable_sort(it_begin, it_end, Mem_compare(key_len));
```

```
  ...
 } else {
  ...
 }
 return std::min(num_input_rows, max_output_rows);
}
```

In general, when using sorting algorithms, the key principles are flexibility in application and cache-friendliness. By considering the algorithm's complexity, one can find the most suitable sorting algorithm.

4.3.3 Greedy Algorithm

A greedy algorithm follows the heuristic of making the locally optimal choice at each stage. Although it often does not produce an optimal solution for many problems, it can yield locally optimal solutions that approximate a globally optimal solution within a reasonable amount of time.

One of the most challenging issues in generating an execution plan based on SQL is selecting the join order. Currently, MySQL's execution plan uses a straightforward greedy algorithm for join order, which performs well in certain scenarios. See the details below for more information:

```
/**
 Find a good, possibly optimal, query execution plan (QEP) by a greedy search.
 ...
 @note
  The following pseudocode describes the algorithm of 'greedy_search':

  @code
  procedure greedy_search
  input: remaining_tables
  output: pplan;
  {
   pplan = <>;
   do {
    (t, a) = best_extension(pplan, remaining_tables);
    pplan = concat(pplan, (t, a));
    remaining_tables = remaining_tables - t;
   } while (remaining_tables != {})
   return pplan;
  }
 ...
*/
```

```
bool Optimize_table_order::greedy_search(table_map remaining_tables) {
```

Determining the optimal join order is an NP-hard problem, making it prohibitively costly in terms of computational resources for complex joins. Consequently, MySQL uses a greedy algorithm. Although this approach may not always yield the absolute best join order, it balances computational efficiency with decent overall performance.

In some join operations, suboptimal join order selection can degrade performance, which might explain why users sometimes criticize MySQL's performance.

4.3.4 Dynamic Programming

Dynamic programming simplifies a complex problem by breaking it down into simpler sub-problems recursively. A problem exhibits optimal substructure if it can be solved optimally by solving its sub-problems and combining their solutions. Additionally, if sub-problems are nested within larger problems and dynamic programming methods are applicable, there is a relationship between the larger problem's value and the sub-problems' values. Two key attributes for applying dynamic programming are optimal substructure and overlapping sub-problems [81].

In the context of execution plan optimization, MySQL 8.0 has explored using dynamic programming algorithms to determine the optimal join order. This approach can greatly improve the performance of complex joins, though it remains experimental in its current implementation.

It is important to note that, due to potentially inaccurate cost estimation, the join order determined by dynamic programming algorithms may not always be the true optimal solution. Dynamic programming algorithms often provide the best plan but can have high computational overhead and may suffer from large costs due to incorrect cost estimation [100]. For a deeper understanding of the complex mechanisms involved, readers can refer to the paper "Dynamic Programming Strikes Back".

4.3.5 Amortized Analysis

In computer science, amortized analysis is a method for analyzing an algorithm's complexity, specifically how much of a resource, such as time or memory, it takes to execute. The motivation for amortized analysis is that considering the worst-case runtime can be overly pessimistic. Instead, amortized analysis averages the running times of operations over a sequence [81].

This section discusses applying amortized analysis to address MySQL problems. While it differs from traditional amortized analysis, the underlying principles are similar and find many practical

applications in addressing MySQL performance issues. For example, during the refactoring of Group Replication, significant jitter was observed when cleaning up outdated certification database information in multi-primary scenarios. The figure below shows read-write tests in MySQL's multi-primary mode with Group Replication, using SysBench at 100 concurrency levels over 300 seconds.

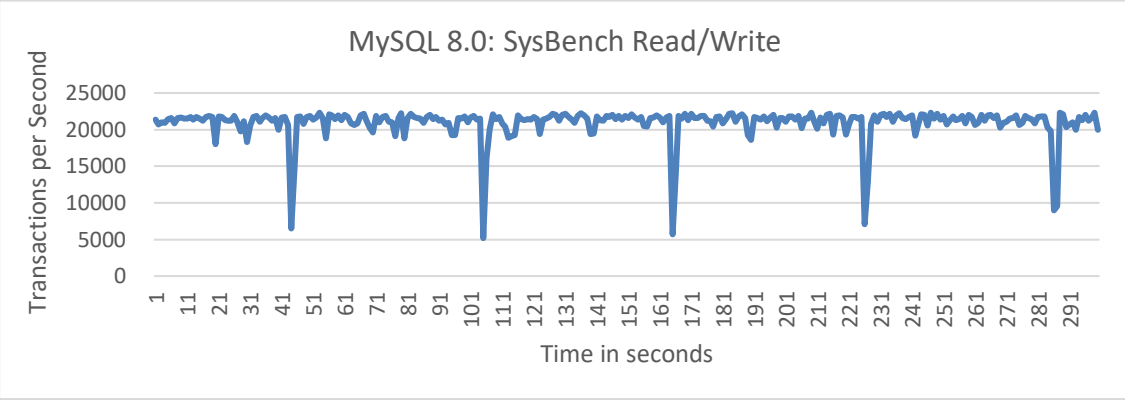


Figure 4-12. Performance fluctuation in MySQL Group Replication.

To address this issue, an amortization strategy was adopted for cleaning up outdated certification database information. See the specific details in the following figure:

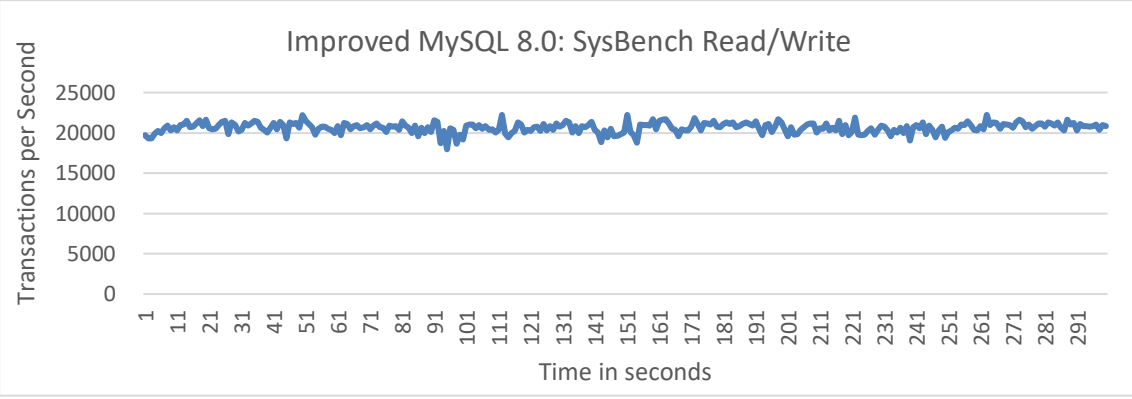


Figure 4-13. Eliminated performance fluctuations in enhanced MySQL Group Replication.

MySQL experienced severe performance fluctuations primarily due to cleaning outdated certification database information every 60 seconds. After redesigning the strategy to clean every 0.2 seconds, resulting in performance fluctuations on the order of milliseconds (ms), these fluctuations became imperceptible during testing. The modified version of MySQL has eliminated sudden performance drops, mainly by applying the amortized approach to reduce significant fluctuations.

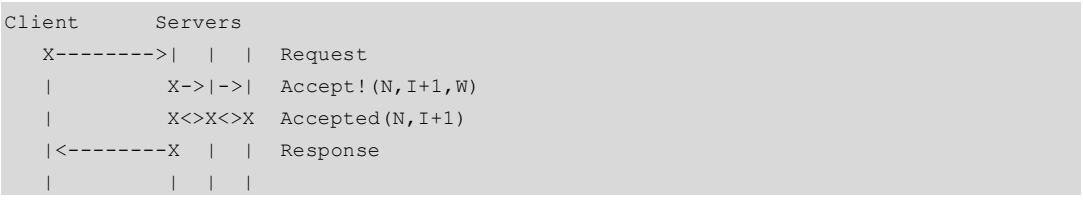
It is important to note that cleaning every 0.2 seconds requires each MySQL node to promptly send its GTID information at intervals of approximately 0.2 seconds. This high-frequency sending is challenging to meet using traditional Multi-Paxos algorithms because these algorithms typically require leader stability over a period of time. Therefore, a single-leader based Multi-Paxos algorithm struggles to handle sudden performance drops effectively, as the underlying algorithm lacks support for such frequent operations.

4.3.6 Paxos

Maintaining consistency among replicas in the face of arbitrary failures has been a major focus in distributed systems for several decades. While naive solutions may work for simple cases, they often fail to provide a general solution. Paxos is a family of protocols designed to achieve consensus among unreliable or fallible processors. Consensus involves agreeing on a single result among multiple participants, a task that becomes challenging when failures or communication issues arise. The Paxos family is widely regarded as the only proven solution for achieving consensus with three or more replicas, as it addresses the general problem of reaching agreement among 2F + 1 replicas while tolerating up to F failures. This makes Paxos a fundamental component of State Machine Replication (SMR) and one of the simplest algorithms for ensuring high availability in clustered environments [43].

The mathematical foundation of Paxos is rooted in set theory, specifically the principle that the intersection of a majority of sets must be non-empty. This principle is key to solving high availability issues in complex scenarios.

However, the pure Paxos protocol often falls short in meeting practical business needs, particularly in environments with significant network latency. To address this, various strategies and enhancements have been developed, leading to several Paxos variants, such as Multi-Paxos, Raft, and Mencius in Group Replication. The following figure illustrates an idealized sequence of Multi-Paxos, where a stable leader allows a proposal message to achieve consensus and return a result to the user within a single Round-trip Time (RTT)[81].



Starting with MySQL 8.0.27, Group Replication incorporates two Paxos variant algorithms: Mencius and the traditional Multi-Paxos. Mencius was designed to address the challenge of multiple leaders in Paxos and was initially developed for wide-area network applications. It allows each node to directly send propose messages, enabling any MySQL secondary to communicate with the cluster at any time without burdening a single Paxos leader. This approach supports consistency in reads and writes and enhances the multi-primary functionality of Group Replication. In contrast, the Multi-Paxos algorithm employs a single-leader approach. Non-leader nodes must request a sequence number from the leader before sending messages to the cluster. Frequent requests to the leader can create a bottleneck, potentially limiting performance.

To achieve high throughput with either Mencius or Multi-Paxos, leveraging batching and pipelining technologies is essential. These optimizations, commonly used in state machine replication [89], significantly enhance performance. For instance, the figure below illustrates how batching improves Group Replication throughput in local area network scenarios.

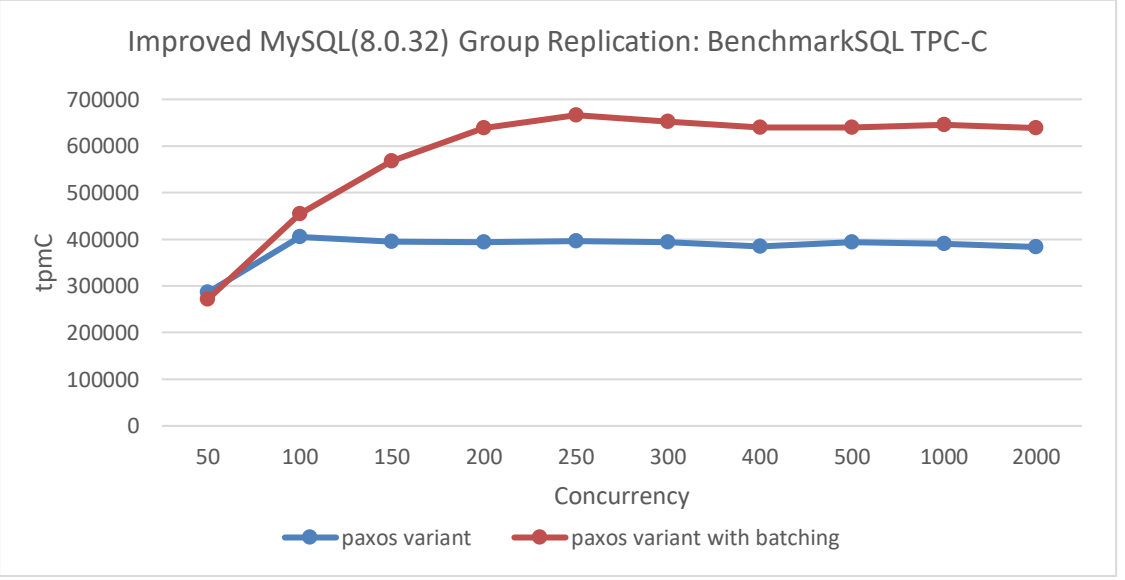


Figure 4-14. Effect of batching on Paxos algorithm performance in LAN environments.

4.4 Operating system

An operating system (OS) is system software that manages computer hardware and software resources and provides common services for computer programs [81].

MySQL is primarily deployed on the Linux operating system. To maintain focus, the discussions here

are based on scenarios within the Linux operating system environment. This includes CPU scheduling, process and thread models, staged models, memory allocation, and system calls, all of which are closely related to MySQL performance.

4.4.1 CPU Scheduling

The job of allocating CPU time to different tasks within an operating system is known as CPU scheduling. The organizational structure of Linux CPUs is illustrated in the figure below [98].

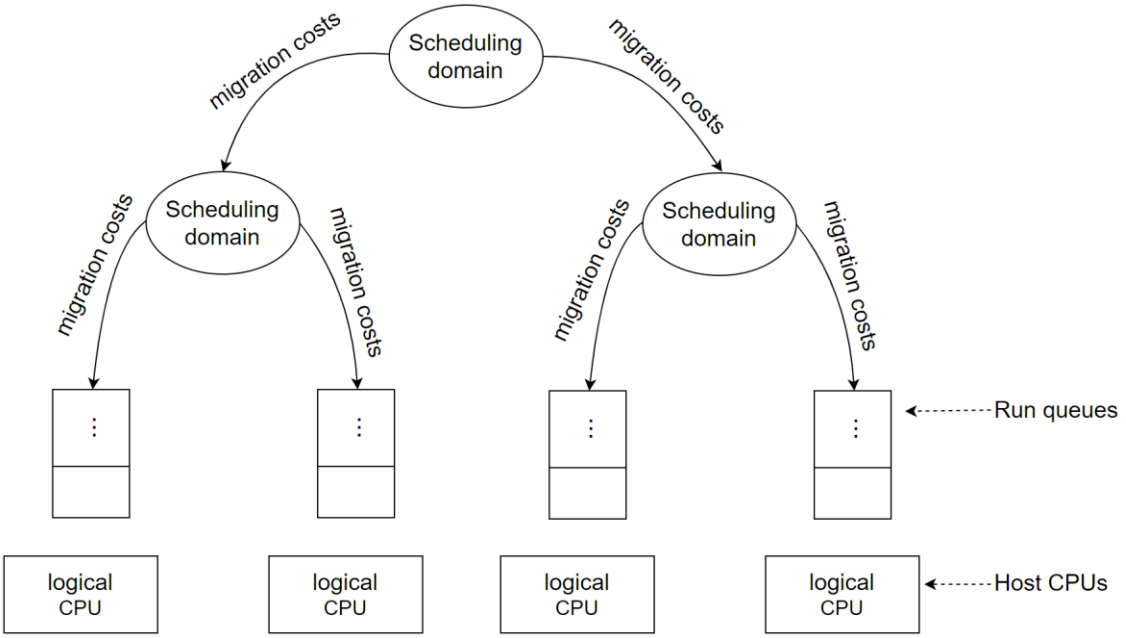


Figure 4-15. Linux CPU scheduling.

Based on the hardware layout of physical cores, the Linux scheduler maintains hierarchically ordered scheduling domains. Basic scheduling domains consist of processes running on physically adjacent cores, such as those on the same chip. Higher-level scheduling domains group these basic domains, such as chips on the same motherboard.

The Linux scheduler operates as a multi-queue scheduler, meaning that each logical host CPU has its own run queue of processes waiting to be executed. Each virtual CPU is queued in one of these run queues. Moving a virtual CPU from one run queue to another is referred to as CPU migration. The scheduler may decide to migrate a virtual CPU when the estimated wait time is too long, the current run queue is full, or another run queue needs to be filled.

Migrating a virtual CPU within the same scheduling domain incurs less cost compared to moving it to a different domain due to the proximity of caches between cores. The Linux scheduler uses detailed information about migration costs between different scheduling domains or CPUs to determine if a migration is beneficial.

Understanding CPU scheduling details is crucial for diagnosing MySQL performance issues. A key question is whether Linux's scheduling mechanisms can effectively manage thousands of concurrent threads in MySQL. Since MySQL operates on a thread-based model, it's important to assess how the Linux scheduler handles such a high volume of threads. Does it simply allocate CPU time evenly among them?

Consider a scenario where there are N user threads and C CPU cores, with each core supporting dual hyper-threading. Ideally, without considering context switch overhead, each user thread should receive $\frac{2C}{N}$ of the CPU execution time per second. As N increases, the average CPU allocation per thread decreases. For example, if $N=100000$ and $C=3$, each thread would only receive about 60 microseconds of CPU time per second. Given that context switches typically incur costs in the tens of microseconds range, a significant portion of CPU time would be lost to context switching, thereby reducing overall CPU efficiency.

As the number of user threads increases, the Linux scheduler struggles to manage CPU time effectively, resulting in inefficiencies and performance degradation due to frequent context switches. To address this, the system enforces a minimum execution granularity, ensuring that each process runs for at least 100 microseconds before being preempted. This approach minimizes the inefficiencies of short scheduling intervals. The Completely Fair Scheduler (CFS) uses this minimum granularity to prevent excessive switching costs as the number of runnable processes grows.

At the same time, increasing the number of CPU cores ensures that each user thread receives sufficient execution time. Coupled with maintaining a minimum execution granularity, the cost of context switching can be significantly mitigated.

Next, let's examine the cost of thread blocking. The disadvantage of blocking is the high cost of context switching—typically 12-20 microseconds—which must be performed twice per lock handoff (once to sleep, and again to wake)[6]. In general, under the condition of maintaining minimum execution times, the cost of context switching compared to current mainstream hardware environments is already quite minimal. Therefore, employing blocking methods has practical value.

Further examining how Linux CPU schedules threads for running programs, specifics can be seen in

the figure below:

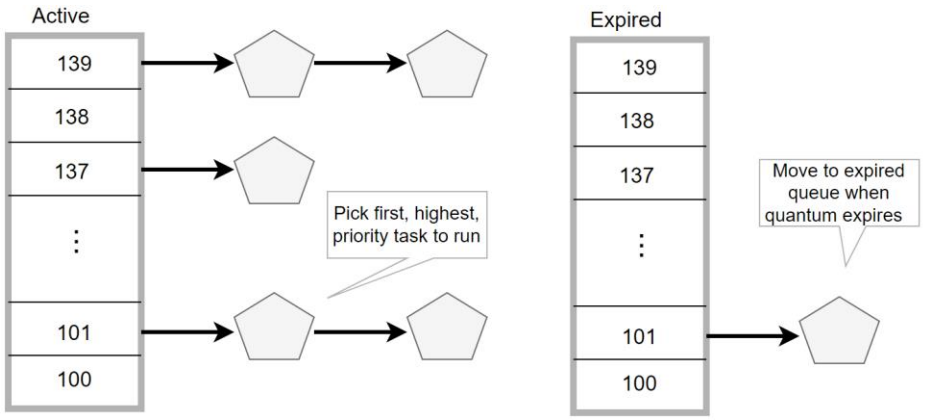


Figure 4-16. How Linux CPU schedules threads for running programs.

The smaller the number, the higher the priority level. The scheduler prioritizes threads with higher priority levels, and after a thread's time slice expires, it is placed in the "expired" queue.

MySQL internal threads can be in different states such as running, waiting for I/O, or waiting for locks. Threads waiting for disk I/O are placed into the corresponding disk wait queue and are not scheduled by the Linux scheduler into the active or expired queues.

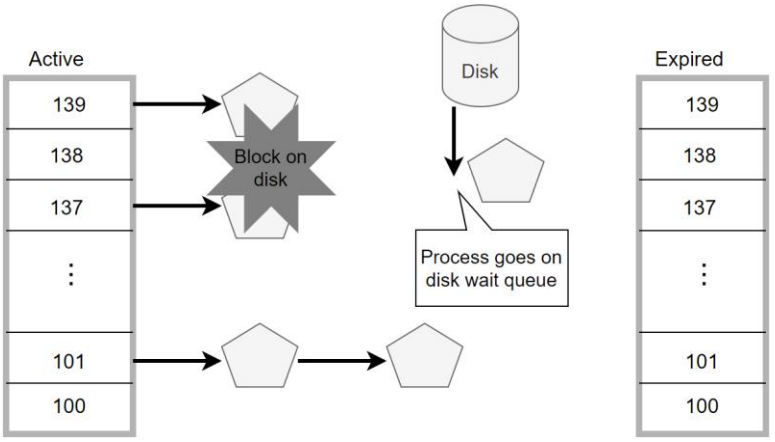


Figure 4-17. Linux CPU scheduling of threads waiting for I/O.

This means that these threads waiting for I/O have little impact on other active threads. In MySQL, ransaction lock waits function similarly to I/O waits—threads voluntarily block themselves and wait to be activated, which is generally manageable in cases where conflicts are not severe.

It is worth mentioning that it is advisable to avoid having a large number of threads waiting for the same global latch or lock, as this can lead to frequent context switches. In NUMA environments, it can also cause frequent cache migrations, thereby affecting MySQL's scalability.

With the increasing number of CPU cores and larger memory sizes available today, the impact of thread creation costs on MySQL has become smaller. Except for special scenarios such as short connection applications, MySQL can handle a large number of threads given sufficient memory. The key is to limit the number of active threads running concurrently. In theory, MySQL can support thousands of concurrent threads.

Using principles of CPU scheduling, MySQL implements transaction throttling mechanisms, such as limiting the number of threads entering the InnoDB transaction system. This ensures that the concurrency of transactions remains manageable. Too many threads entering InnoDB can lead to latch contention, significantly reducing efficiency.

The following figure illustrates the use of transaction throttling mechanisms to limit a maximum of 512 threads entering the InnoDB storage engine, depicting MySQL's single-instance throughput from 50 to 10,000 concurrency.

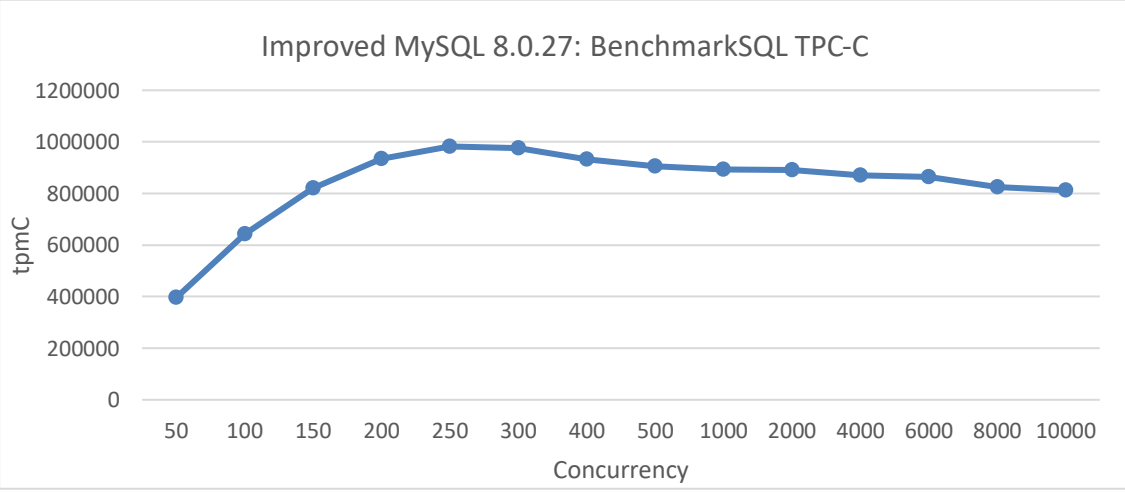


Figure 4-18. Maximum TPC-C throughput in BenchmarkSQL with transaction throttling mechanisms.

As illustrated in the figure, TPC-C throughput exceeds 800,000 tpmC even at 10,000 concurrency, showing only a slight decrease from the peak value.

4.4.2 Process Model

In Linux, processes and threads are fundamental to multitasking and parallel execution. A process is an independent execution unit with its own memory space, while a thread is a smaller execution unit within a process, sharing its memory space.

Key differences include:

- **Memory Consumption:** Processes require separate memory space, making them more memory-intensive compared to threads, which share the memory of their parent process. A process typically consumes around 10 megabytes, whereas a thread uses about 1 megabyte.
- **Concurrency Handling:** Given the same amount of memory, systems can support significantly more threads than processes. This makes threads more suitable for applications requiring high concurrency.

When building a concurrent database system, memory efficiency is critical. MySQL's use of a thread-based model offers an advantage over traditional PostgreSQL's process-based model, particularly in high concurrency scenarios. While PostgreSQL's model can lead to higher memory consumption, MySQL's threading model is more efficient in handling large numbers of concurrent connections.

Additionally, although Nginx uses a multi-process model, it achieves scalability through asynchronous programming techniques.

4.4.3 Thread Model

For MySQL, the thread-based model is advantageous over the process model due to its theoretical support for tens of thousands of concurrent threads. However, the paper "A Case for Staged Database Systems" highlights some shortcomings of this model [34].

Challenges of the Thread-Based Model:

1. **Cache Performance:** The thread-based execution model often results in poor cache performance with multiple clients.
2. **Complexity:** The monolithic design of modern DBMS software leads to complex, hard-to-maintain systems.

Pitfalls of Thread-Based Concurrency:

1. **Thread Management:** There is no optimal number of preallocated worker threads for

varying workloads. Too many threads can waste resources, while too few restrict concurrency.

2. **Context Switching:** Context switches during operations can evict a large working set from the cache, causing delays when the thread resumes.

3. **Cache Utilization:** Round-robin scheduling does not consider the benefit of shared cache contents, leading to inefficiencies.

Despite ongoing improvements in operating systems, the thread model continues to face significant challenges in optimization.

In MySQL, a significant issue occurs when a large number of threads enter the InnoDB transaction system. This increases latch contention and extends the MVCC ReadView global active transaction list, causing frequent cache migrations. This problem is especially pronounced in ultra-high concurrency scenarios. For example, the figure below shows how Group Replication throughput varies with concurrency in a network environment with 10ms latency.

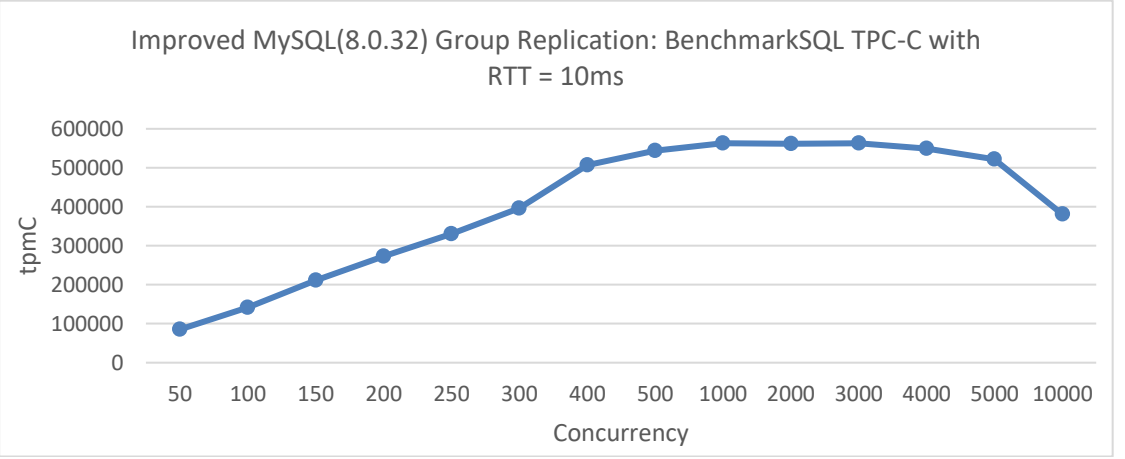


Figure 4-19. Maximum thread scalability of Group Replication under 10ms network latency.

At 5000 concurrency, throughput remains relatively stable. However, beyond this point, throughput declines sharply. In extremely high concurrency scenarios, severe latch conflicts and frequent cache migrations significantly reduce MySQL's efficiency.

4.4.4 Staged Model

The staged model is a specialized type of thread model that minimizes some of the drawbacks associated with handling numerous concurrent threads. This model involves breaking the database

system into distinct modules, each encapsulated into self-contained stages interconnected through queues. By addressing both hardware and software challenges, the staged model provides effective solutions to the limitations of traditional DBMS designs [34].

Benefits of the Staged Model

1. **Targeted Thread Allocation**: Each stage allocates worker threads based on its specific functionality and I/O frequency, rather than the number of concurrent clients. This approach allows for more precise thread management tailored to the needs of different database tasks, compared to a generic thread pool size.

2. **Voluntary Thread Yielding**: Instead of preempting a thread arbitrarily, a stage thread voluntarily yields the CPU at the end of its stage code execution. This reduces cache eviction during the shrinking phase of the working set, minimizing the time needed to restore it. This technique can also be adapted to existing database architectures.

3. **Exploiting Stage Affinity**: The thread scheduler focuses on tasks within the same stage, which helps to exploit processor cache affinity. The initial task brings common data structures and code into higher cache levels, reducing cache misses for subsequent tasks.

4. **CPU Binding Efficiency**: The singular nature of thread operations in the staged model allows for improved efficiency through CPU binding, which is especially effective in NUMA environments.

The staged model is extensively used in MySQL for tasks such as secondary replay, Group Replication, and improvements to the Redo log in MySQL 8.0. However, it is not well-suited for handling user requests due to increased response times caused by various queues. MySQL primary servers prioritize low latency and high throughput for user-facing operations, while tasks like secondary replay, which do not interact directly with users, can afford higher latency in exchange for high throughput.

The figure below illustrates the processing flow of Group Replication. In this design, Group Replication is divided into multiple subprocesses connected through queues. This staged approach offers several benefits, including:

- **High Efficiency**: By breaking down tasks into discrete stages, the system can process tasks more effectively.

- **Cache-Friendly Access**: The design minimizes cache misses by ensuring that related tasks are executed in sequence.

- **Pipelined Processing**: Tasks are handled in a pipelined manner, allowing for improved throughput

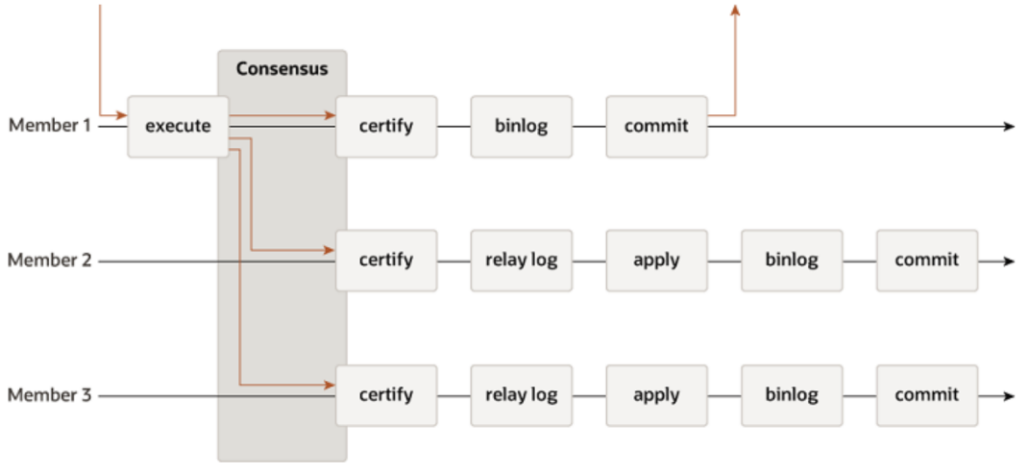


Figure 4-20. MySQL Group Replication protocol.

In the staged model, using a single thread can become a bottleneck when processing large amounts of data. For instance, in the apply replay process shown in the figure, the thread is responsible for both CPU-intensive tasks, such as reading and decoding the relay log, and for scheduling and coordination. This can significantly hinder the performance of MySQL secondary replay, limiting its ability to process data efficiently and quickly.

4.4.5 Memory Allocation

The Linux memory management subsystem oversees virtual memory, demand paging, and memory allocation for both kernel structures and user programs. Memory hot-spots often arise from improper data initialization [7], especially under the default first-touch policy, which can lead to uneven memory distribution across NUMA nodes. To mitigate this, initialize data in the computation-partition where it will be used.

Modern CPUs generate high memory request rates that can overwhelm the interconnect or memory controller. NUMA systems, with their numerous hardware threads, face scalability issues due to concurrent memory allocation requests. Solutions include overriding the memory allocator, defining thread placement and affinity schemes, and adjusting OS configurations. Efficient memory allocators are particularly beneficial in NUMA systems where performance penalties from inefficient memory or cache use can be significant.

Linux prioritizes local node allocation and minimizes thread migrations across nodes using scheduling domains. This reduces inter-domain migrations but may affect load balance and performance. To optimize memory usage:

1. Identify memory-intensive threads.
2. Distribute them across memory domains.
3. Migrate memory with threads.

Understanding these memory management principles is crucial for diagnosing and resolving MySQL performance issues. Linux aims to reduce process interference by minimizing CPU switches and cache migrations across NUMA nodes.

The figure below illustrates a comparative experiment. The deep blue curve shows the total throughput of running four MySQL instances on a NUMA system without node binding. In contrast, the deep red curve represents the total throughput when each MySQL instance is bound to different NUMA nodes. This latter setup reflects the ideal throughput, as it minimizes CPU scheduling interference between MySQL instances.

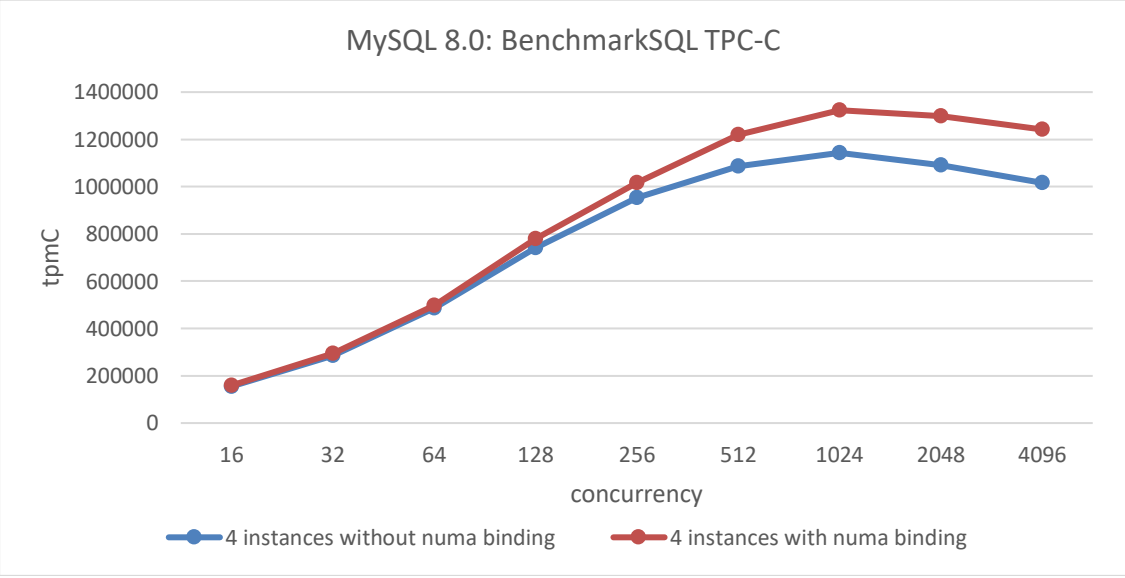


Figure 4-21. Comparing ideal NUMA scheduling with actual Linux scheduling of multiple instances.

Based on the figure, it's evident that while the operating system strives to approach the ideal throughput under low concurrency, the gap widens as concurrency increases. This widening gap

occurs because memory scheduling becomes more complex with a higher number of threads.

For MySQL applications, effectively utilizing NUMA-based memory allocation is crucial. Test results indicate that for TPC-C type workloads, disabling NUMA at the BIOS level can enhance throughput for MySQL primary servers. This improvement is due to the more favorable memory allocation configuration provided by this setup for MySQL primary and similar applications.

4.4.6 System Call

Traditionally, the performance cost of system calls is primarily due to mode switch time. This time includes executing the system call instruction in user mode, transitioning to kernel mode, and eventually returning to user mode. Modern processors handle this mode switch as a processor exception, which involves flushing the user-mode pipeline, saving registers to the kernel stack, changing the protection domain, and directing execution to the exception handler. After this, a return from exception is needed to resume execution in user mode [29].

Frequent system calls can significantly impact MySQL's performance. For example, Percona's thread pool relies on system calls that introduce significant overhead, potentially reducing some of the performance gains from the thread pool.

4.5 Compiler Theory

Compiler Theory is a crucial computer science course that covers the fundamental principles and methods of compiler construction. It includes topics such as language and grammar, lexical analysis, syntax analysis, syntax-directed translation, intermediate code generation, memory management, code optimization, and target code generation. Central to the course is the finite state machine.

4.5.1 Finite State Machine

A finite state machine (FSM) is a mathematical model used to represent a finite number of states, transitions, and actions within a system. FSMs are essential in various fields such as compilers, network protocols, and game engines, where they describe complex systems, algorithms, and event responses. The core concept of FSMs is state transition, which defines the relationships and actions between different states.

In MySQL, FSMs play a crucial role in lexical and syntax analysis, as well as in Group Replication. For example, in Group Replication, FSMs help manage decisions during Paxos handling, such as whether to continue processing or to exit with an error if memory allocation fails. Typically, if a state

cannot continue processing, it exits with an error to prevent Byzantine faults.

FSMs are also used in TCP/IP protocols to detect abnormal states. For instance, a high number of CLOSE WAIT states on server-side TCP connections can indicate that many connections have not been closed properly.

4.5.2 MySQL Parser

The MySQL Parser, implemented in C/C++, uses GNU Bison and Flex for parsing SQL queries. Flex generates tokens, while Bison handles syntax parsing. SQL parsing is crucial for operations like read/write evaluations by MySQL proxy and translating SQL into syntax trees.

With the increase in CPU core count, the CPU overhead for MySQL parsing has become a smaller proportion of overall usage. This reduction in overhead might explain why MySQL 8.0 removed the query cache.

4.5.3 PGO

Profile-guided optimization (PGO) is a compiler technique that enhances program performance by using profiling data from test runs of the instrumented program. Rather than relying on programmer-supplied frequency information, PGO leverages profile data to optimize the final generated code, focusing on frequently executed areas of the program. This method reduces reliance on heuristics and can improve performance, provided the profiling data accurately represents typical usage scenarios.

MySQL's large codebase, which adheres to the Pareto principle (80/20 rule), is particularly suited for PGO. However, the effectiveness of PGO can be influenced by I/O storage devices and network latency. On systems with slower I/O devices, like hard drives, I/O becomes the primary bottleneck, limiting PGO's performance gains due to Amdahl's Law. In contrast, on systems with faster I/O devices such as SSDs, PGO can lead to substantial performance improvements. Network latency also affects PGO effectiveness, with higher latency generally reducing the benefits.

In summary, while MySQL 8.0's PGO capabilities can greatly enhance computational performance, the actual improvement depends on the balance between computational and I/O bottlenecks in the server setup. The following figure demonstrates that with SSD hardware configuration and NUMA binding, PGO can significantly improve the performance of MySQL.

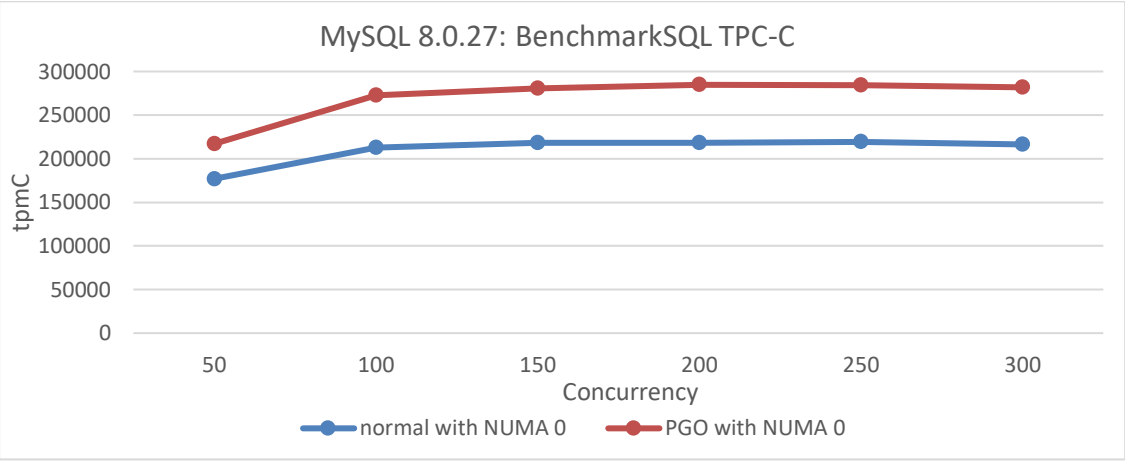


Figure 4-22. Performance comparison tests before and after using PGO in MySQL 8.0.27 under SMP.

4.6 Queueing Theory

Queueing theory is the mathematical study of waiting lines or queues, used to predict queue lengths and waiting times [81]. It is widely applied in computer science and information technology. For example, in networking, routers and switches rely on queues to manage packet transmission. By applying queueing theory, designers can optimize these systems for responsive performance and efficient resource utilization. Although queueing theory is not specifically designed to improve software performance, it serves as a model for evaluating system efficiency by examining how resources, both physical and logical, are utilized. According to queueing theory, nearly any resource can be viewed as a queue, with bottleneck resources analyzed as such to understand their characteristics.

4.6.1 Single Queue Bottleneck

The following figure depicts a scenario where MySQL primary and MySQL secondary form a two-node cluster under a network latency of 10ms.

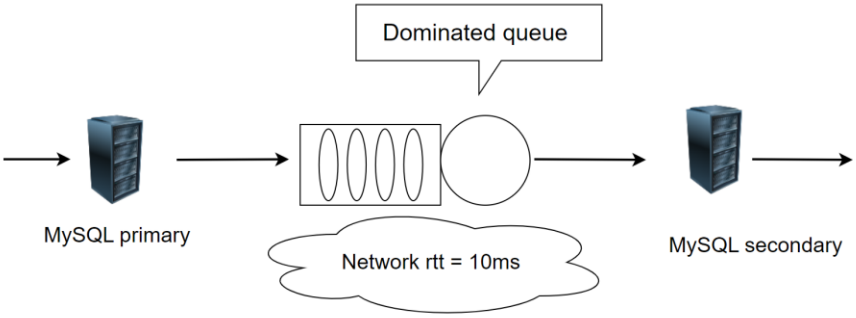


Figure 4-23. Testing architecture for Group Replication with pure Paxos protocol

The cluster's Paxos algorithm employs a modified Mencius approach, removing batching and pipelining, making it similar to pure Paxos. Tests were conducted at various concurrency levels under a network latency of 10ms, as illustrated in the following figure:

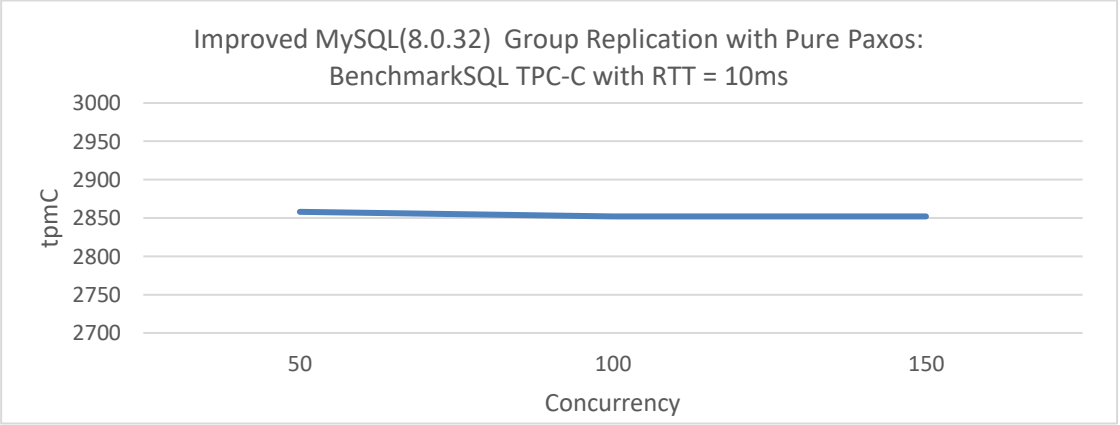


Figure 4-24. Results of testing Group Replication with pure Paxos protocol

In a WAN testing scenario, the throughput remains nearly constant across different concurrency levels—50, 100, or 150—because the time MySQL takes to process TPC-C transactions is negligible compared to the network latency of 10ms. This network latency dominates the overall transaction time, making the impact of concurrency changes relatively insignificant.

The throughput calculation formula in such scenarios simplifies to:

$$\text{tpmTOTAL} \approx 60 \times \frac{1}{Network\ Latency} = 60 \times \frac{1}{0.01} = 6000 \text{ transactions per second}$$

$$\text{tpmC} \approx \text{tpmTOTAL} \times 0.45 = 6000 \times 0.45 = 2700 \text{ tpmC}$$

This aligns with the test results above, where 0.45 is an empirical factor derived from extensive testing, representing the ratio of tpmC to tpmTOTAL. The tests indicate that, under a 10ms network latency with no additional bottlenecks, throughput remains consistent across different concurrency levels. This consistency is due to the serial nature of Paxos communication, as batching and pipelining are not employed. Confirmation of these findings is supported by packet capture analysis.

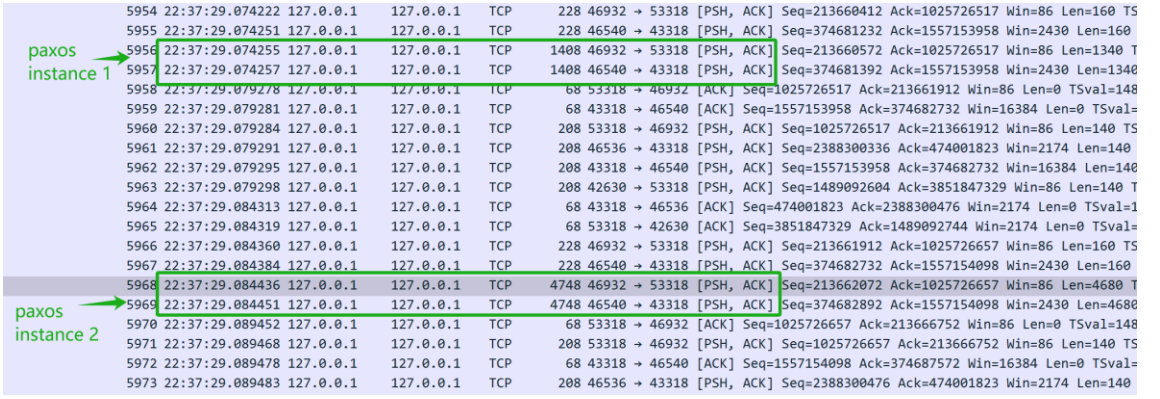


Figure 4-25. Insights into the pure Paxos protocol from packet capture data.

In the figure, the network latency between the two Paxos instances is approximately 10ms, matching the exact network delay. Numerous examples suggest that pure Paxos communication is inherently serial. In scenarios where network latency is the predominant factor, it acts as a single queue bottleneck. Consequently, regardless of concurrency levels, the throughput of pure Paxos is limited by this network latency.

4.6.2 Multiple Queue Bottlenecks

A complex program often involves multiple queues, and unless one queue dominates, several bottlenecks can exist simultaneously. The figure below illustrates a basic server queue model with a single CPU and two disks. When the CPU is busy, requests are queued, and the same happens when accessing the disks. In this model, queueing theory can be applied to calculate the server's throughput capacity.

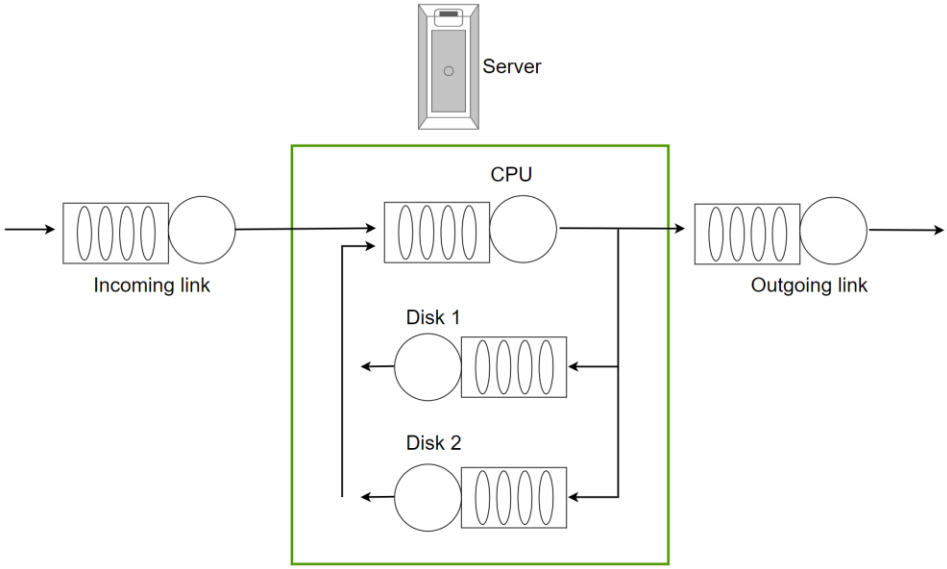


Figure 4-26. A basic server queue model with single CPU and dual Disks.

In modern NUMA environments with abundant CPUs and cross-NUMA access interference, directly calculating system throughput using queueing theory can be challenging. For instance, the following figure illustrates the latch queue model of MySQL 5.7.

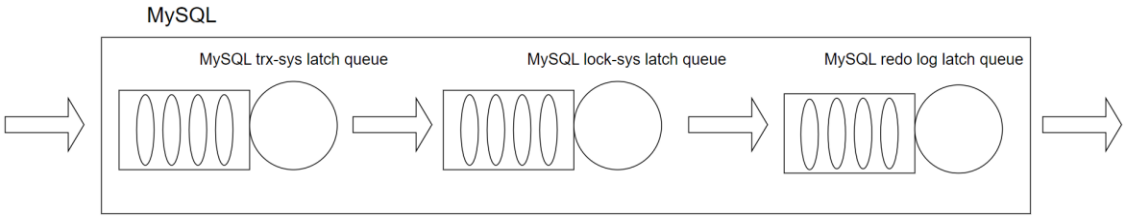


Figure 4-27. The latch queue model in MySQL 5.7.

This involves the latch queue bottlenecks within the InnoDB storage engine, a significant scalability issue in MySQL 5.7. In this queue model, addressing individual bottlenecks alone is insufficient to resolve scalability problems, and accurately assessing the impact of these queue bottlenecks on throughput remains challenging.

4.6.3 The Mutual Influence Between Different Queue Bottlenecks

The following figure is a simplified example of a multi-queue model in a NUMA environment.

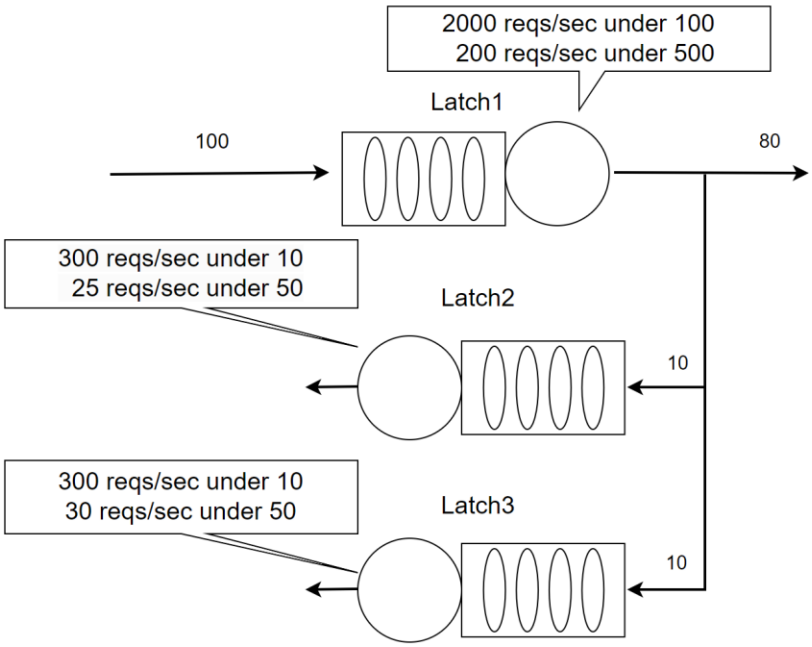


Figure 4-28. A multi-queue model in a NUMA Environment.

In the case of 100 concurrent threads, the latch1 queue can handle 2,000 requests per second. However, with 500 concurrent threads, its throughput drops to only 200 requests per second. This discrepancy arises because increased conflicts lead to significant context switching, causing cache content to migrate across NUMA nodes, which sharply reduces memory access efficiency. As concurrency rises, this inefficiency grows, leading to a notable decline in throughput.

Under normal conditions, with 100 concurrent threads passing through latch1, 10 threads typically proceed to latch2 and another 10 to latch3. Both latch2 and latch3 can handle 300 requests per second with 10 concurrent threads, meaning that no bottlenecks are encountered.

When 500 concurrent threads interact with latch1, the increased conflicts result in fewer requests successfully passing through, leading to reduced throughput. After identifying and optimizing latch1 (as shown in the figure below), its throughput capacity increased significantly: it can now handle 1,000 requests per second with 500 concurrent threads. This optimization has effectively mitigated the previous performance degradation caused by latch1 conflicts.

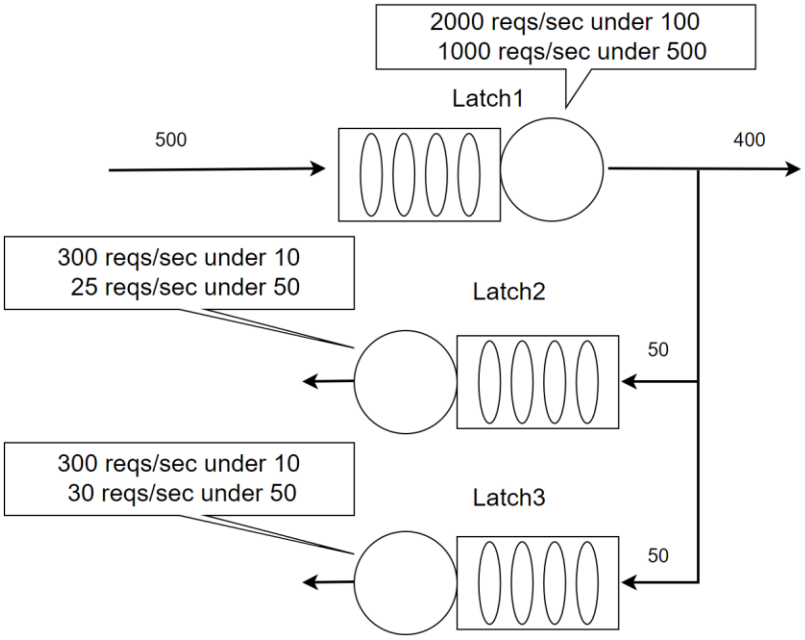


Figure 4-29. A multi-queue model mitigating latch1 bottleneck in NUMA environments.

With 500 concurrent threads reaching the latch1 queue, the improved processing efficiency allows 50 threads to progress to latch2 and another 50 to latch3. However, severe conflicts at both latch2 and latch3 reduce their processing capacities to just 25 and 30 requests per second, respectively. This disrupts the balance of latch2 and latch3 queues, which were stable before optimizing latch1.

In the scenario of multi-queue bottlenecks, optimizing one bottleneck may not necessarily lead to an increase in overall throughput; it could potentially even decrease throughput. Real-life examples of this include instances where Profile-Guided Optimization (PGO) led to decreased throughput under high concurrency, illustrating this phenomenon.

Let's review again the throughput situation under high concurrency after applying PGO:

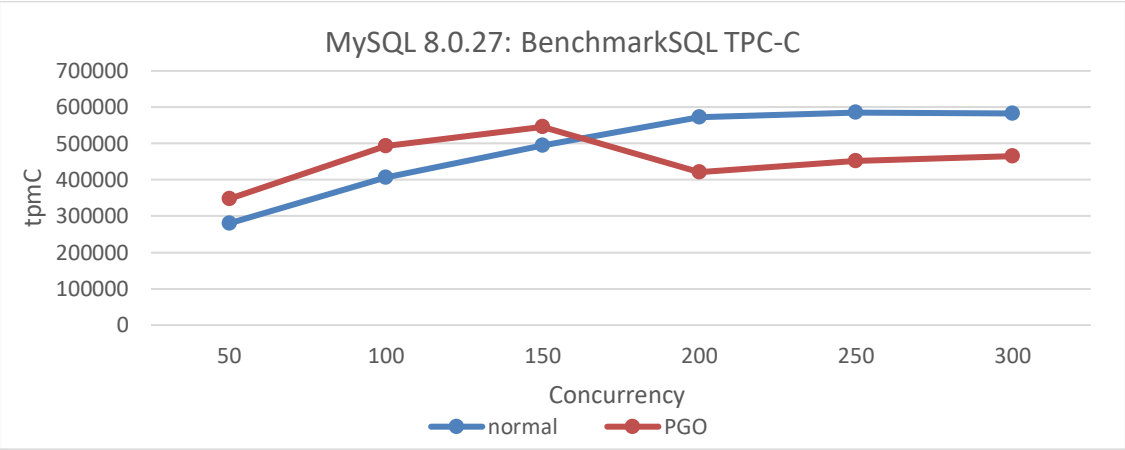


Figure 4-30. Performance comparison tests before and after using PGO in MySQL 8.0.27.

Profile-Guided Optimization (PGO) accelerates CPU computation efficiency, enabling a larger number of threads to enter the latch queue earlier under high concurrency. This exacerbates latch conflicts compared to the pre-optimization state, leading to decreased throughput. Interestingly, when PGO is disabled, throughput decreases under low concurrency but increases under high concurrency. This counterintuitive phenomenon is common when multi-queue bottlenecks coexist.

Let's examine the performance comparison before and after redo log optimization:

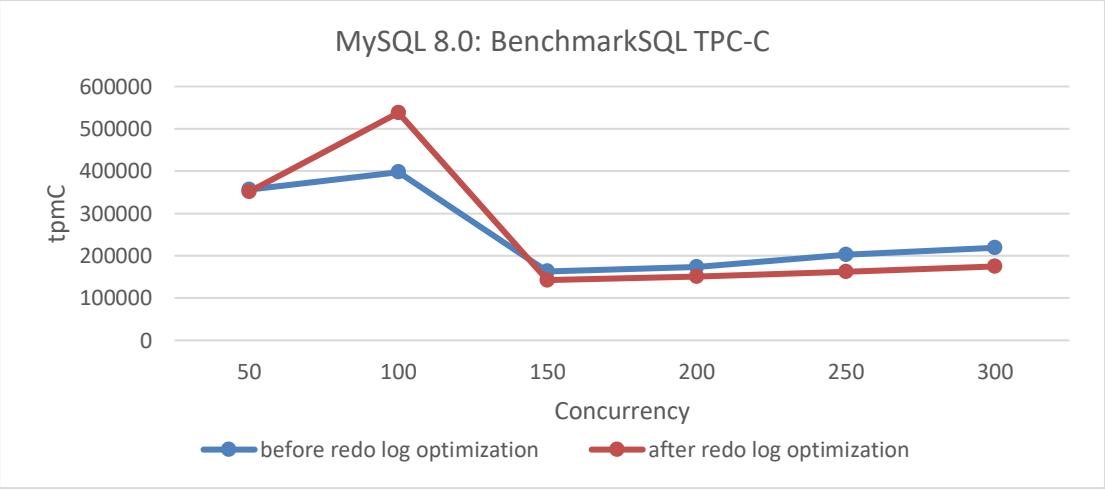


Figure 4-31. Performance comparison tests before and after redo log optimization in MySQL 8.0.

The figure shows that performance improves under low concurrency but decreases under high concurrency. This repeatedly tested phenomenon indicates that not all optimizations show immediate

results; eliminating disturbances is crucial to observe true effectiveness. These issues often occur in NUMA environments, so mitigating NUMA compatibility problems is essential to avoid misjudgments in performance optimization. Developers' reliance on data can overlook the real reasons for performance declines, leading to missed optimization opportunities.

4.6.4 The Relationship Between Resource Utilization and Response Time

According to queueing theory, as resource utilization increases, average response time also increases, and as utilization approaches 100%, average response time deteriorates sharply. The following figure shows concurrent tests on CPU resources using SysBench, displaying the relationship between average response time and CPU utilization. Initially, the average response time increases slowly. However, around 60% CPU utilization, the curve sharply rises, reaching over 13 milliseconds at 96% utilization.

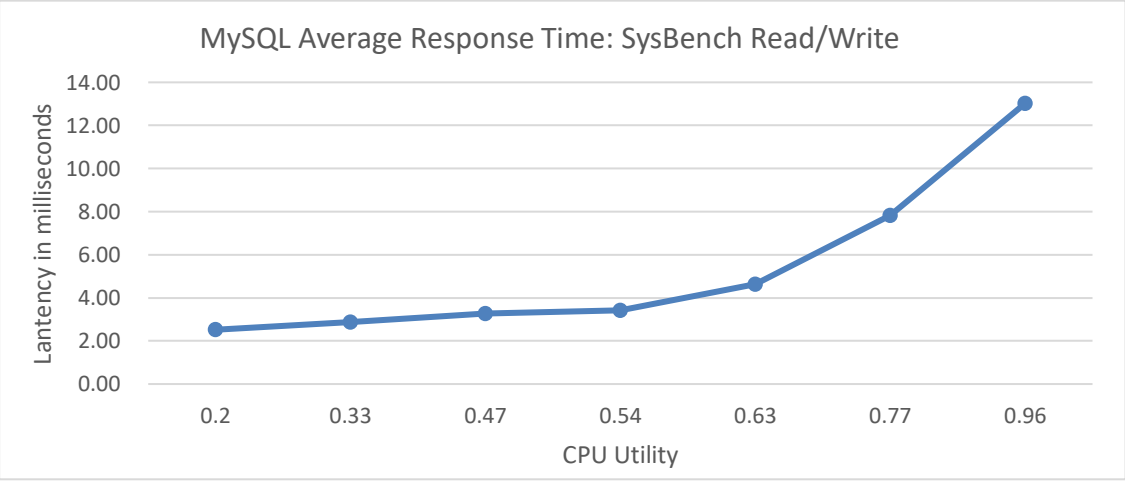


Figure 4-32. Relationship between average response time and CPU utilization.

If resources are depleted, such as running out of memory, the machine's response time can become very slow. Therefore, controlling resource utilization is essential. According to queueing theory, for MySQL OLTP systems, maintaining response times requires keeping utilization within certain limits. Finding the optimal utilization point is crucial to maintain efficiency, which is the basis of transaction throttling theory. While queueing theory doesn't directly solve problems, it provides guidance for performance optimization and deepens the understanding of system performance.

4.6.5 Transaction Throttling Mechanism

To prevent performance degradation, controlling resource usage is crucial. For MySQL OLTP applications, managing concurrency entering the transaction system is key due to multiple latch queue bottlenecks and the replication of global active transaction lists.

A practical transaction throttling mechanism for MySQL is as follows:

1. Before entering the transaction system, check if the number of concurrent processing threads exceeds the limit.
2. If the limit is exceeded, block the user thread until other threads activate this thread.
3. If the limit is not exceeded, allow the thread to proceed with processing within the transaction system.
4. Upon transaction completion, activate the first transaction in the waiting queue.

This approach helps maintain performance by controlling concurrency and managing resource usage effectively. The following figure illustrates the relationship between TPC-C throughput and concurrency under transaction throttling conditions, with 1000 warehouses.

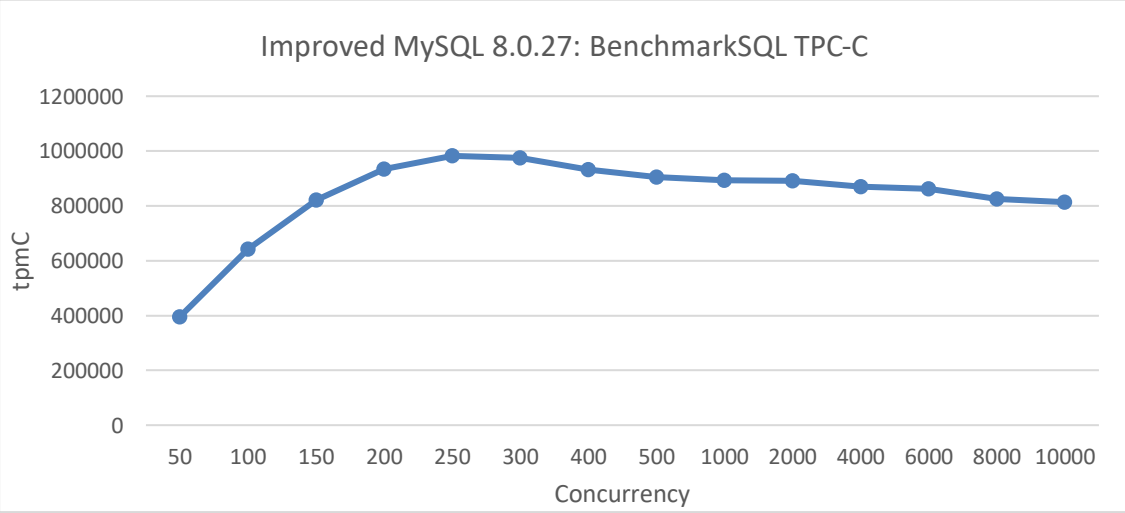


Figure 4-33. Maximum TPC-C throughput in BenchmarkSQL with transaction throttling mechanisms.

From the figure, it is evident that implementing transaction throttling mechanisms significantly improves MySQL's scalability.

4.7 Computer Networking

Computer networking can be considered a branch of computer science, computer engineering, and telecommunications, as it relies on the theoretical and practical application of these disciplines. The field has been shaped by numerous technological developments and historical milestones [81].

In schools, networking courses often teach the theory, but they don't usually show students how to actually analyze and solve network problems. Consequently, when issues arise, many feel lost and lack the logical thinking skills needed for effective problem-solving. While network knowledge can be acquired at any time, identifying the correct approach to problem-solving requires extensive practice. A correct approach ensures smooth problem resolution, whereas a wrong direction can significantly increase the difficulty of solving issues.

This section discusses common theories, techniques, network partitioning, and logical reasoning used to address MySQL-related issues in the context of networking.

4.7.1 FLP Impossibility Theory

In a fully asynchronous message-passing distributed system with potential process crash failures, the 1985 FLP impossibility result by Fischer, Lynch, and Paterson proves that achieving consensus with a deterministic algorithm is impossible. However, practitioners often disregard this result because real systems usually exhibit some level of synchrony. The worst-case scheduling scenarios posited by the FLP result are unlikely to occur in practice, except in adversarial situations like an intelligent denial-of-service attack [83].

The FLP impossibility theorem is valuable in problem-solving as it highlights the challenge of distinguishing between lost and delayed messages in a network. In practice, while message delays are typically bounded, delays in systems like MySQL can be prolonged, making it difficult to tell if a node's information is lost or simply delayed. Misjudgments here can lead to errors, such as Group Replication incorrectly reporting nodes as unreachable.

The Mencius algorithm used in Group Replication addresses the FLP impossibility by using a failure detector oracle to bypass the result. Like Paxos, it relies on the failure detector only for liveness. Mencius requires that eventually, all and only faulty servers are suspected by the failure detector. This can be achieved by implementing failure detectors with exponentially increasing timeouts [57].

To avoid the problems posed by the FLP impossibility, careful design is needed. TCP, for example, addresses this with timeout retransmission and idempotent design, ensuring that even if duplicate

messages are received due to transmission errors, they can be safely discarded.

4.7.2 TCP/IP Protocol Stack

The Internet protocol suite, commonly known as TCP/IP, organizes the set of communication protocols used in the Internet and similar computer networks [81]. It provides end-to-end data communication, specifying how data should be packetized, addressed, transmitted, routed, and received. The suite is divided into four abstraction layers, each classifying related protocols based on their networking scope:

1. **Link Layer**: Handles communication within a single network segment (link).
2. **Internet Layer**: Manages internetworking between independent networks.
3. **Transport Layer**: Facilitates host-to-host communication.
4. **Application Layer**: Enables process-to-process data exchange for applications.

An implementation of these layers for a specific application forms a protocol stack. The TCP/IP protocol stack is one of the most widely used globally, having operated successfully for many years since its design. The following figure illustrates how a client program interacts with a MySQL Server using the TCP/IP protocol stack.

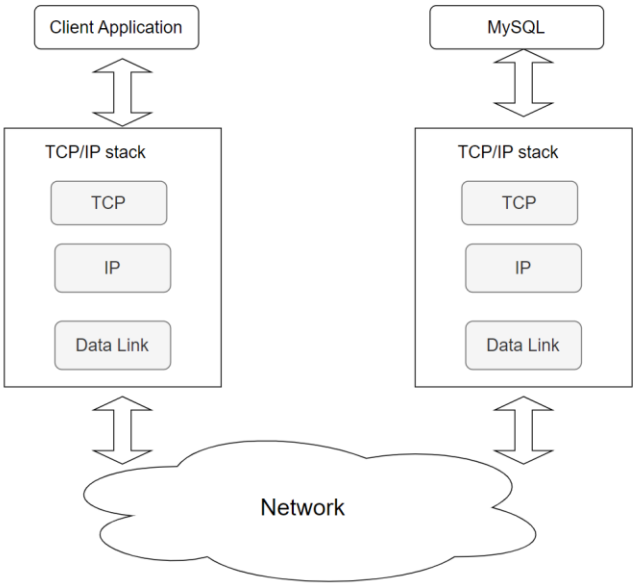


Figure 4-34. A client program interacts with a MySQL Server using the TCP/IP protocol stack.

Due to the layered design of the TCP/IP protocol stack, a client program typically interacts only with the local TCP to access a remote MySQL server. This design is elegant in its simplicity:

1. **Client-Side TCP**: Handles sending SQL queries end-to-end to the remote MySQL server. It manages retransmission if packets are lost.
2. **Server-Side TCP**: Receives the SQL queries from the client-side TCP and forwards them to the MySQL server application. After processing, it sends the response back through its TCP stack.
3. **Routing and Forwarding**: TCP uses the IP layer for routing and forwarding, while the IP layer relies on the data link layer for physical transmission within the same network segment.

Although TCP ensures reliable transmission, it cannot guarantee that messages will always reach their destination due to potential network anomalies. For example, SQL requests might be blocked by a network firewall, preventing them from reaching the MySQL server. In such cases, the client application might not receive a response, leading to uncertainty about whether the request was processed or still in transit.

To address these issues, additional tools like packet capture can help determine if SQL requests have reached the MySQL server, thereby clarifying the situation.

4.7.3 TCP State Machine

The following figure depicts the classic TCP state machine used for transitions between TCP states.

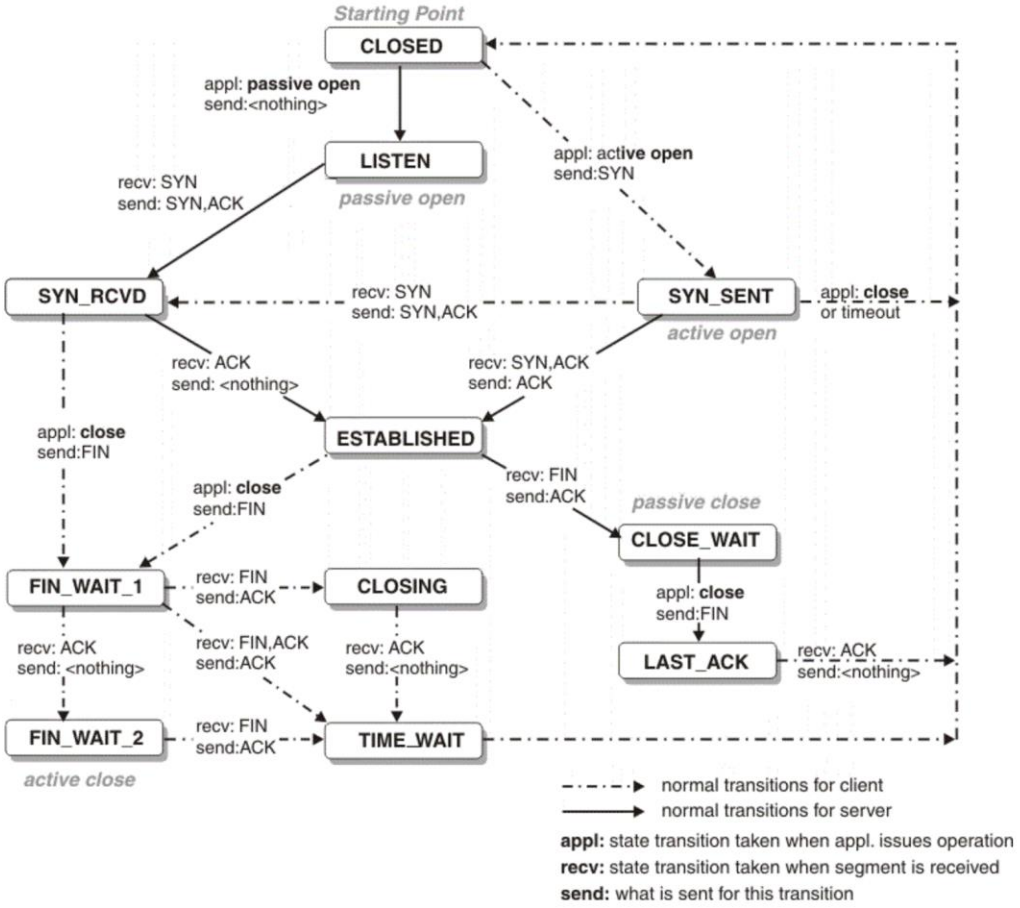


Figure 4-35. Classic TCP state machine overview.

A flexible understanding of state transitions is crucial for troubleshooting MySQL network issues. For example:

- **CLOSE_WAIT State**: A large number of CLOSE_WAIT states on the server indicates that the application did not close connections promptly or failed to initiate the close process, causing connections to linger in this state.
- **SYN_RCVD State**: Numerous SYN_RCVD states may suggest a SYN flood attack, where an excessive number of SYN requests overwhelm the server's capacity to handle them effectively.

Understanding these state transitions helps in diagnosing and addressing network-related problems more effectively.

4.7.4 Will Data Content Be Corrupted During Network Transmission?

Many software professionals analyzing MySQL network issues often suspect data corruption during transmission, leading to questions about TCP reliability.

In the TCP/IP protocol stack, IP networks are unreliable, while TCP ensures reliability. How does TCP ensure reliable transmission to the user program? There are two main scenarios: packet loss and data corruption. When facing packet loss in an IP network, TCP employs a mechanism called timeout and retransmission. According to the FLP impossibility theorem, even if intermediate devices introduce delays, retransmitted data packets, including those from before, are likely to reach the destination. TCP achieves idempotency in receiving information based on sequence numbers and payload lengths, effectively resolving issues caused by packet loss and delay. For data corruption, TCP uses checksum verification. If a discrepancy is detected, corrupted data is discarded, and the sender retransmits the data.

However, TCP transmission is not immune to security risks. Checksum verification is provided, but it can't ensure security against intentional tampering by intermediate devices that alter both payload and checksum simultaneously. This is why TLS/SSL protocols were developed to enhance data security, though they can complicate problem analysis in typical situations.

4.7.5 Batching and Pipelining

Paxos is a widely used state machine replication protocol, and its performance can be significantly enhanced through batching and pipelining optimizations. However, tuning these optimizations for optimal performance can be complex, as their effectiveness is influenced by various factors including network latency and bandwidth, node speeds, and application properties.

Below is a comparison test in LAN environments using BenchmarkSQL for TPC-C. There are 1000 warehouses, and concurrency ranges from 50 to 2000. Deep blue represents throughput without batching, while deep red represents throughput with batching.

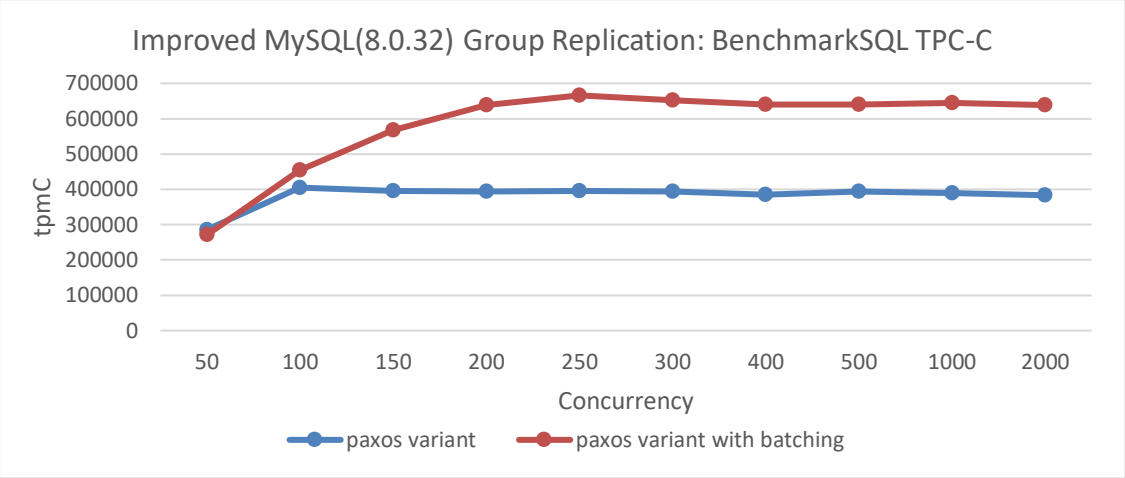


Figure 4-36. Effect of batching on Paxos algorithm performance in LAN environments.

From the figure, it is clear that batching significantly increases throughput under high concurrency. In a LAN environment, does pipelining also have a similar effect? The following figure presents comparative tests conducted without batching to evaluate whether pipelining alone can also substantially boost throughput.

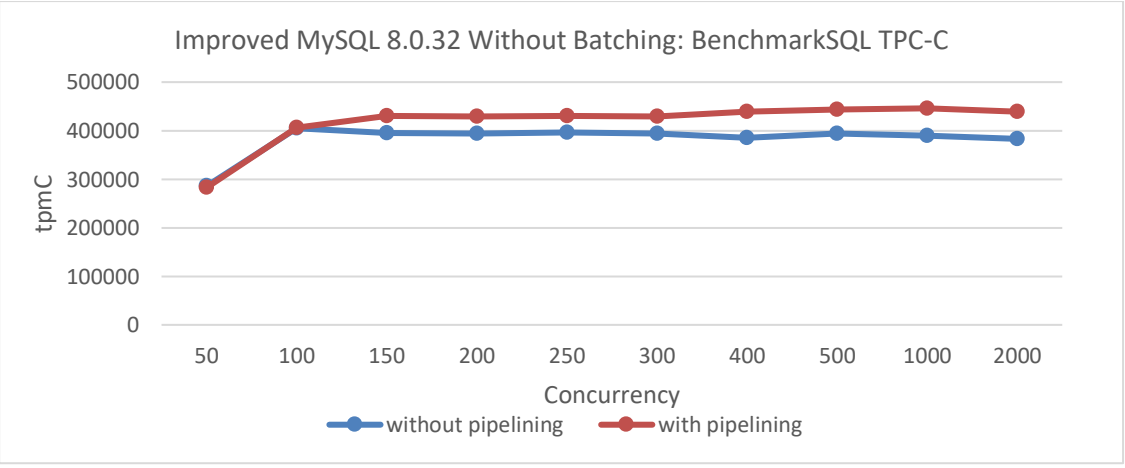


Figure 4-37. Effect of pipelining on Paxos algorithm performance in LAN environments.

From the figure, it is clear that while pipelining does enhance TPC-C throughput in LAN environments, the improvement is relatively modest. Amazingly, in LAN environments with batching already enabled, adding pipelining offers little additional benefit, as shown in the following figure.

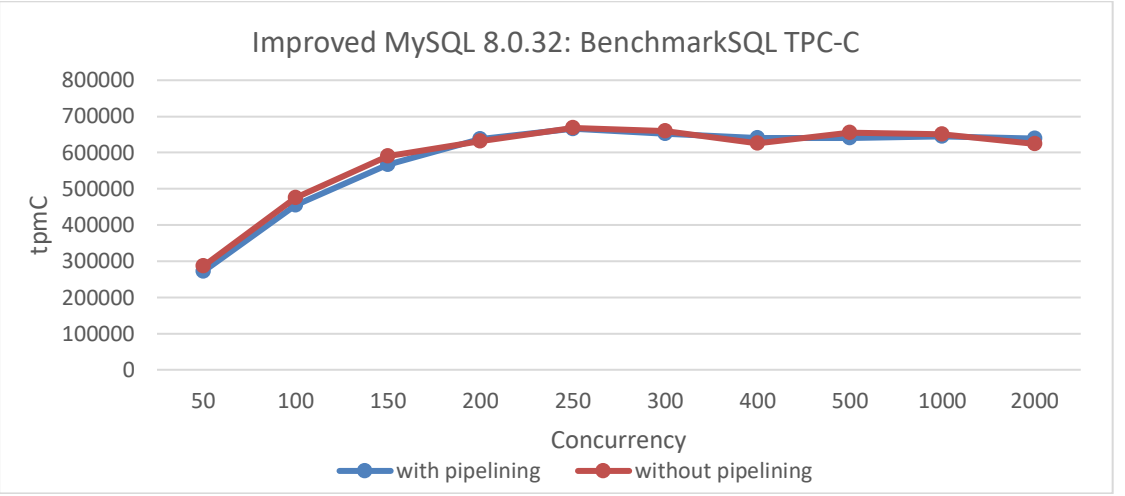


Figure 4-38. Effect of pipelining on batched Paxos algorithm performance in LAN environments.

How does pipelining perform in WAN environments? Let's explore its impact. The following figure shows the throughput of 1000 concurrency under different numbers of pipelining, with a network latency of 10ms and Paxos algorithm enabling batching.

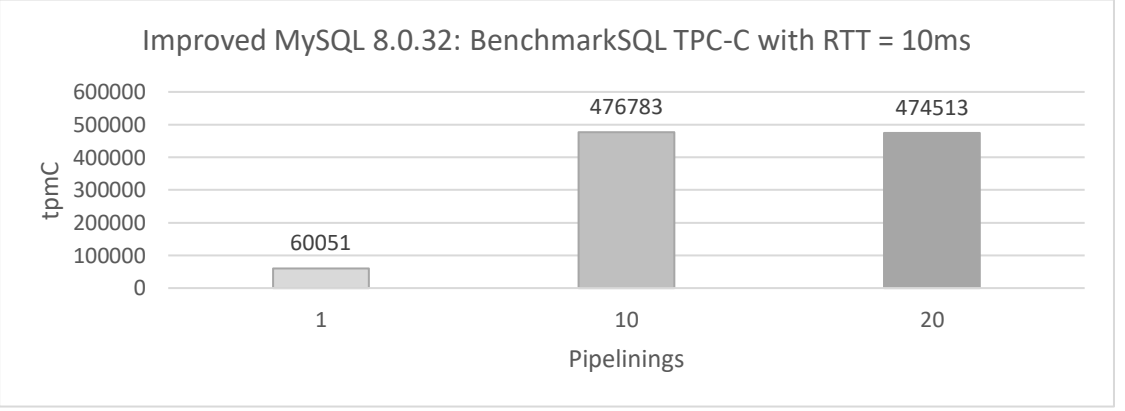


Figure 4-39. Effect of pipelining on batched Paxos algorithm performance in WAN environments.

From the figure, it is clear that with just one pipelining, throughput is relatively low at 60,051 tpmC. With ten pipelinings, throughput increases significantly to 476,783 tpmC. However, increasing the pipelining to twenty shows little difference compared to having 10 pipelinings. In summary, batching is highly effective in LAN environments, while pipelining provides significant benefits in WAN environments. Both techniques are effective tools for boosting throughput.

Let's continue delving into batching in WAN environments. The following figure shows the

comparison of throughput for 1000 concurrency before and after enabling batching, with pipelining disabled and a network latency of 10ms.

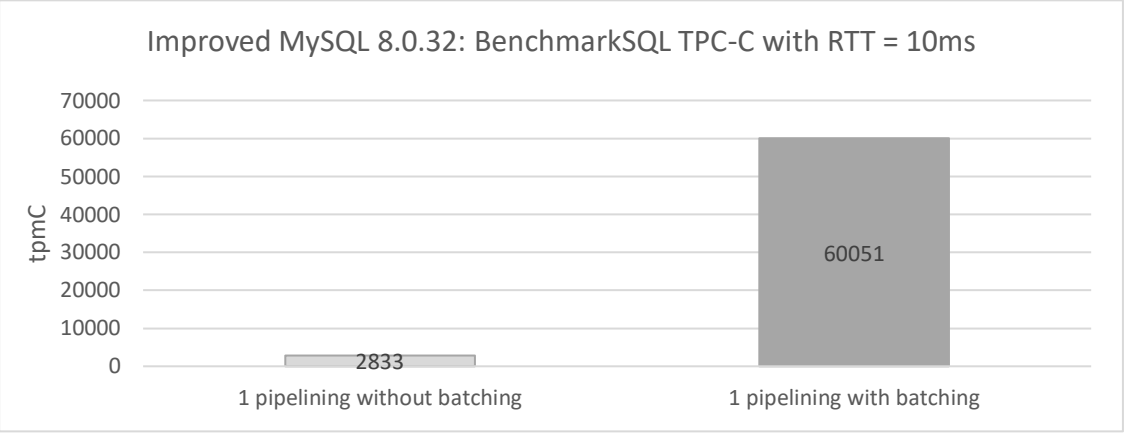


Figure 4-40. Effect of batching on Paxos algorithm performance in WAN environments.

From the figure, it is evident that with both pipelining and batching disabled, throughput drops to just 2833 tpmC. This figure reflects the performance of the pure Paxos algorithm in WAN environments, where throughput is notably low.

Why does pure Paxos perform poorly in WAN environments? Refer to the packet capture details in the following figure for insights.

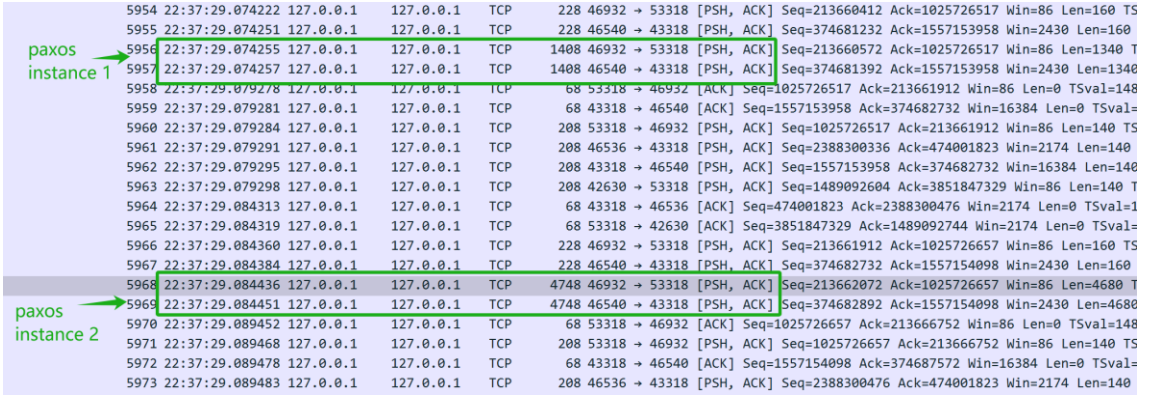


Figure 4-41. Insights into the pure Paxos protocol from packet capture data.

From the figure, it is evident that the delay between two Paxos instances is around 10ms, matching the network latency. The low throughput of pure Paxos stems from its serial interaction nature, where network latency primarily determines throughput.

In general, the test conclusions of pipelining and batching are consistent with the conclusions in the following paper [88]:

Tuning Paxos for high-throughput with batching and pipelining

Nuno Santos, André Schiper
Ecole Polytechnique Fédérale de Lausanne (EPFL)
Email: firstname.lastname@epfl.ch

***Abstract*—Paxos is probably the most popular state machine replication protocol. Two optimizations that can greatly improve its performance are batching and pipelining. Nevertheless, tuning these two optimizations to achieve optimal performance can be challenging, as their effectiveness depends on many parameters like the network latency and bandwidth, the speed of the nodes, and the properties of the application. We address this question, by first presenting an analytical model of the performance of Paxos that can be used to obtain values for tuning batching and pipelining. We then present experiments validating the model and investigating how these two optimizations interact in both a LAN and a WAN setting. The results show that although batching by itself is usually sufficient to maximize the throughput in a LAN environment, in a WAN it must be complemented with pipelining.**

and potentially in a higher throughput. Batching can easily be implemented on top of Paxos, as it does not require any changes to the ordering protocol. *Pipelining* [1] is an extension of the basic Paxos protocol where the leader initiates new instances of the ordering protocol before the previous ones have completed. This optimization is particularly effective when the network latency is high, as it allows the leader to pipeline several instances on the slow link.

Batching and pipelining are used by most replicated state machine implementations, as they usually provide performance gains between one and two orders of magnitude. Nevertheless, in order to achieve the best throughput, they must be carefully tuned. For batching, it is necessary to

Figure 4-42. Impact of pipelining and batching on Paxos performance.

4.7.6 Impact of Network Latency on Performance

Network latency critically affects MySQL performance. In a localhost test case with 1000 warehouses, the MySQL instance bound to NUMA nodes 0, 1, and 2, and BenchmarkSQL on NUMA node 3, latency's impact on throughput is examined across concurrency levels of 50, 100, and 150.

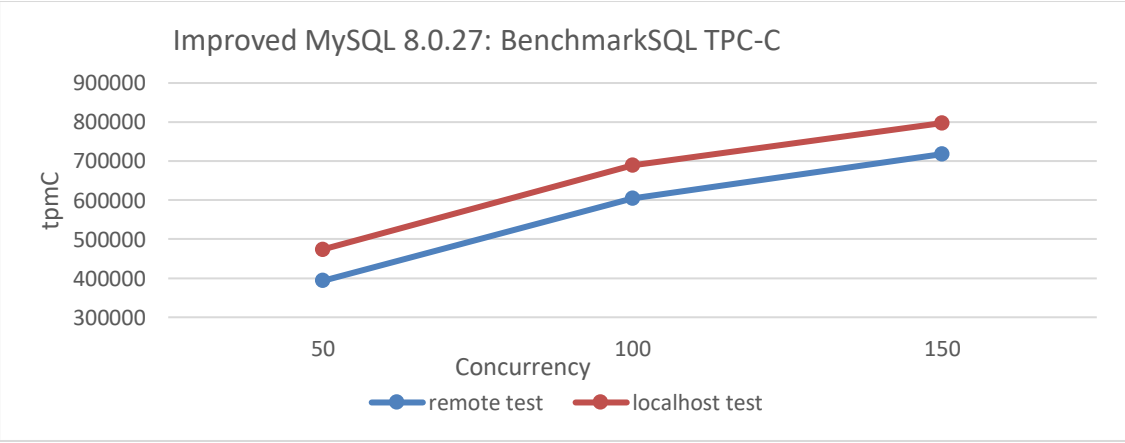


Figure 4-43. Impact of network latency on performance.

Testing in a localhost scenario can significantly reduce network latency. From the figure, it can be

seen that reducing network latency has a noticeable impact on MySQL throughput, reaching 800,000 tpmC at 150 concurrency. Higher concurrency was not tested because MySQL throughput was nearing its bottleneck with the utilization of three NUMA nodes. This bottleneck would interfere with assessing the impact of network latency on throughput.

4.7.7 Network Partition

Network partitioning is a network failure that splits members into groups, preventing direct communication between groups [81].

Network partitioning can be categorized into three main types [12], as shown in the figure below.

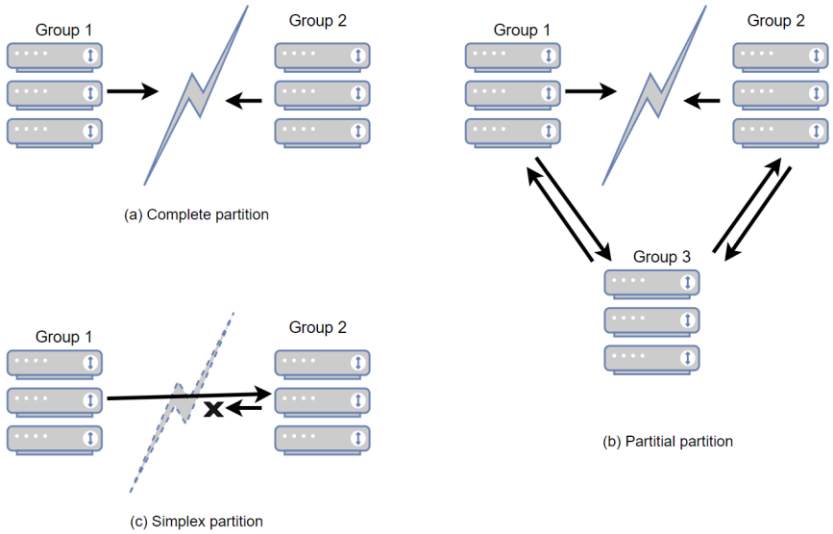


Figure 4-44. Network partitioning types.

The figure categorizes network partitions into three types:

1. **Complete Network Partition (a)**: Two partitions are completely disconnected from each other, widely recognized as a network partition.
2. **Partial Network Partition (b)**: Group 1 and Group 2 are disconnected from each other, but Group 3 can still communicate with both. This is termed a partial network partition.
3. **Unidirectional Network Partition (c)**: Communication is possible in one direction but not the other, known as a unidirectional network partition.

The most complex type is the partial network partition. Partial partitions isolate a set of nodes from

some, but not all, nodes in the cluster, leading to a confusing system state where nodes disagree on whether a server is up or down. These disagreements are poorly understood and tested, even by expert developers [12].

For high-availability components like Group Replication, testing is extremely challenging. Without theoretical support, one might not even consider that network partitions come in three types. The second and third types of scenarios are generally the most problematic because they are often either untested or not thoroughly tested. Networks are complex, and diagnosing issues, especially with network partitions, is particularly challenging. However, there is logic behind it, and solving network partition issues typically requires gathering information and logical reasoning.

4.7.8 RDMA

Remote Direct Memory Access (RDMA) enables direct memory access between two computers without involving the operating system, cache, or storage. Unlike traditional IP communication, RDMA bypasses kernel intervention, reducing CPU overhead. The RDMA protocol allows the host adapter to place network packets directly into the application's memory space, bypassing kernel processing and copying. This requires applications to implement the InfiniBand Verbs API.

While RDMA significantly reduces TCP processing latency and increases MySQL throughput in LAN scenarios, its effectiveness is limited in cross-datacenter environments due to network latency. Additionally, the high cost of modifying MySQL limits RDMA's adoption in practical use cases.

4.7.9 The Importance of Packet Capture Information

Due to the complexity of network issues and the lack of effective information support for software professionals, there is a continuous risk of misjudgment and falling into traps of obscure problems. To effectively resolve MySQL-related network issues, reliable and accurate information is essential for confidently moving towards a successful resolution. Mastering packet capture analysis is crucial for tackling these obscure network problems and can often play a decisive role in the resolution process.

To capture packets specifically for MySQL interactions, the following command can be used, assuming 3306 is MySQL's listening port:

```
tcpdump -i any tcp and port 3306 -s 250 -w mysql.pcap -v
```

For more advanced features, refer to the tcpdump documentation for detailed commands. While

solving complex network issues can be challenging, packet capture itself remains relatively straightforward.

4.7.10 Logical Reasoning in Network Issues

Many software professionals lack in-depth knowledge of TCP/IP logic reasoning, which often leads to misidentifying issues as mysterious problems. Some are deterred by the complexity of TCP/IP networking literature, while others are misled by confusing details in Wireshark. For instance, a DBA facing performance issues might misinterpret packet capture data in Wireshark, erroneously concluding that TCP retransmissions are the cause.

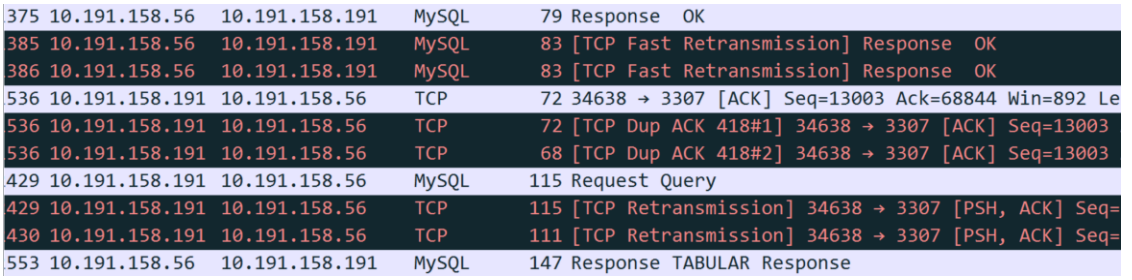


Figure 4-45. Packet capture screenshot provided by DBA suspecting retransmission issues.

Since retransmission is suspected, it's essential to understand its nature. Retransmission fundamentally involves timeout retransmission. To confirm if retransmission is indeed the cause, time-related information is necessary, which is not provided in the screenshot above. After requesting a new screenshot from the DBA, the timestamp information was included.

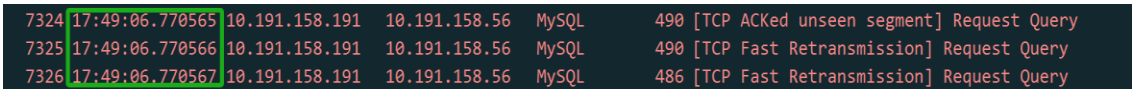


Figure 4-46. Packet capture screenshots with time information added by the DBA.

When analyzing network packets, timestamp information is crucial for accurate logical reasoning. A time difference in the microsecond range between two duplicate packets suggests either a timeout retransmission or duplicate packet capture. In a typical LAN environment with a Round-trip Time (RTT) of around 100 microseconds, where TCP retransmissions require at least one RTT, a retransmission occurring at just 1/100th of the RTT likely indicates duplicate packet capture rather than an actual timeout retransmission.

Another classic case illustrates the importance of logical reasoning in network problem analysis.

One day, one business developer came rushing over, saying that a scheduled script using the MySQL

database middleware had failed in the early morning hours with no response. Upon hearing about the issue, I checked the error logs of the MySQL database middleware but found no valuable clues. So, I asked the developers if they could reproduce the problem, knowing that once reproducible, a problem becomes easier to solve.

The developers tried multiple times to reproduce the issue but were unsuccessful. However, they made a new discovery: they found that executing the same SQL queries during the day resulted in different response times compared to the early morning. They suspected that when the SQL response was slow, the MySQL database middleware was blocking the session and not returning results to the client.

Based on this insight, the database operations team were asked to modify the script's SQL to simulate a slow SQL response. As a result, the MySQL database middleware returned the results without encountering the hang issue seen in the early morning hours.

For a while, the root cause couldn't be identified, and developers discovered a functional issue with the MySQL database middleware. Therefore, developers and DBA operations became more convinced that the MySQL database middleware was delaying responses. In reality, these issues were not related to the response times of the MySQL database middleware.

From the events of the first day, the problem did indeed occur. Everyone involved tried to pinpoint the cause, making various guesses, but the true reason remained elusive.

The next day, developers reported that the script issue reoccurred in the early morning, yet they couldn't reproduce it during the day. Developers, feeling pressured as the script was soon to be used online, complained about the situation. My only suggestion was for them to use the script during the day to avoid problems in the early morning. With all suspicions focused on the MySQL database middleware, it was challenging to analyze the issue from other perspectives.

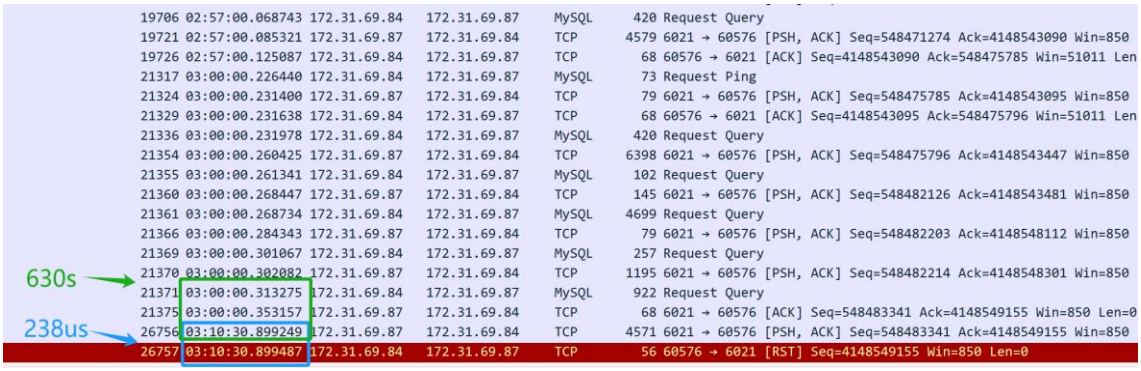
As a developer responsible for the MySQL database middleware, such mysterious issues cannot be easily overlooked. Ignoring them could impact subsequent use of the MySQL database middleware, and there is also pressure from leadership to resolve the issue promptly. Finally, it was decided to implement a low-cost packet capture analysis solution: during the execution of the script in the early morning, packet captures would be performed on the server to analyze what was happening at that time. The goal was to determine if the MySQL database middleware either failed to send a response at all or if it did send a response that the client script did not receive. Once it could be confirmed that the MySQL database middleware did send a response, the issue would not be attributed to the MySQL database middleware developers.

On the third day, developers reported that the early morning issue did not recur, and packet capture analysis confirmed that the problem did not occur. After careful consideration, it seemed unlikely that the issue was solely with the MySQL database middleware: frequent occurrences in the early morning and rare occurrences during the day were puzzling. The only course of action was to wait for the problem to occur again and analyze it based on the packet captures.

On the fourth day, the issue did not surface again.

However, on the fifth day, the problem finally reappeared, bringing hope for resolution.

The packet capture files are numerous. First, ask the developers to provide the timestamp when the issue occurred, then search through the extensive packet capture data to identify the SQL queries that caused the problem. The final result found is as follows:



19706	02:57:00.068743	172.31.69.84	172.31.69.87	MySQL	420	Request Query
19721	02:57:00.085321	172.31.69.87	172.31.69.84	TCP	4579	6021 → 60576 [PSH, ACK] Seq=548471274 Ack=4148543090 Win=850
19726	02:57:00.125087	172.31.69.84	172.31.69.87	TCP	68	60576 → 6021 [ACK] Seq=4148543090 Ack=548475785 Win=51011 Len
21317	03:00:00.226440	172.31.69.84	172.31.69.87	MySQL	73	Request Ping
21324	03:00:00.231400	172.31.69.87	172.31.69.84	TCP	79	6021 → 60576 [PSH, ACK] Seq=548475785 Ack=4148543095 Win=850
21329	03:00:00.231638	172.31.69.84	172.31.69.87	TCP	68	60576 → 6021 [ACK] Seq=4148543095 Ack=548475796 Win=51011 Len
21336	03:00:00.231978	172.31.69.84	172.31.69.87	MySQL	420	Request Query
21354	03:00:00.260425	172.31.69.87	172.31.69.84	TCP	6398	6021 → 60576 [PSH, ACK] Seq=548475796 Ack=4148543447 Win=850
21355	03:00:00.261341	172.31.69.84	172.31.69.87	MySQL	102	Request Query
21360	03:00:00.268447	172.31.69.87	172.31.69.84	TCP	145	6021 → 60576 [PSH, ACK] Seq=548482126 Ack=4148543481 Win=850
21361	03:00:00.268734	172.31.69.84	172.31.69.87	MySQL	4699	Request Query
21366	03:00:00.284343	172.31.69.87	172.31.69.84	TCP	79	6021 → 60576 [PSH, ACK] Seq=548482203 Ack=4148548112 Win=850
21369	03:00:00.301067	172.31.69.84	172.31.69.87	MySQL	257	Request Query
21370	03:00:00.302082	172.31.69.87	172.31.69.84	TCP	1195	6021 → 60576 [PSH, ACK] Seq=548482214 Ack=4148548301 Win=850
21371	03:00:00.313275	172.31.69.84	172.31.69.87	MySQL	922	Request Query
21375	03:00:00.353157	172.31.69.87	172.31.69.84	TCP	68	6021 → 60576 [ACK] Seq=548483341 Ack=4148549155 Win=850 Len=0
26756	03:10:30.899249	172.31.69.87	172.31.69.84	TCP	4571	6021 → 60576 [PSH, ACK] Seq=548483341 Ack=4148549155 Win=850
26757	03:10:30.899487	172.31.69.84	172.31.69.87	TCP	56	60576 → 6021 [RST] Seq=4148549155 Win=850 Len=0

Figure 4-47. Key packet information captured for problem resolution.

From the packet capture content above (captured from the server), it appears that the SQL query was sent at 3 AM. The MySQL database middleware took 630 seconds (03:10:30.899249-03:00:00.353157) to return the SQL response to the client, indicating that the MySQL database middleware did indeed respond to the SQL query. However, just 238 microseconds later (03:10:30.899487-03:10:30.899249), the server's TCP layer received a reset packet, which is suspiciously quick. It's important to note that this reset packet cannot be immediately assumed to be from the client.

Firstly, it is necessary to confirm who sent the reset packet—either it was sent by the client or by an intermediate device along the way. Since packet capture was performed only on the server side, information about the client's packet situation is not available. By analyzing the packet capture files from the server side and applying logical reasoning, the aim is to identify the root cause of the problem.

If the assumption is made that the client sent a reset, it would imply that the client's TCP layer no

longer recognizes the TCP state of this connection—transitioning from an established state to a nonexistent one. This change in TCP state would notify the client application of a connection issue, causing the client script to immediately error out. However, in reality, the client script is still waiting for the response to come back. Therefore, the assumption that the client sent a reset does not hold true—the client did not send a reset. The client's connection is still active, but on the server side, the corresponding connection has been terminated by the reset.

Who sent the reset, then? The primary suspect is Amazon's cloud environment. Based on this packet capture analysis, the DBA operations queried Amazon customer service and received the following information:

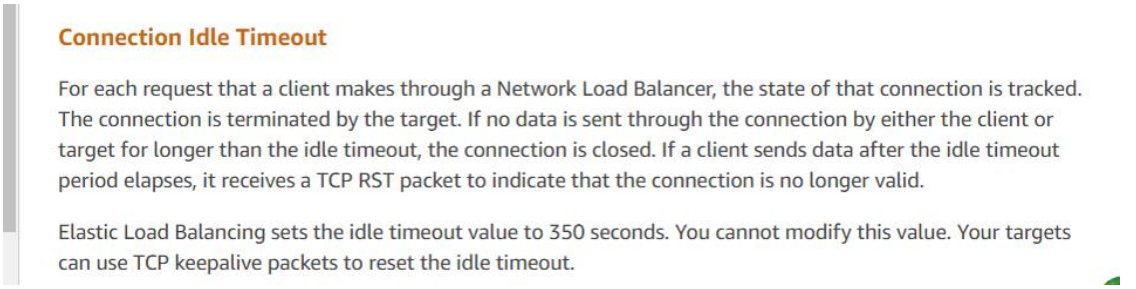


Figure 4-48. Final response from Amazon customer service.

Customer service's response aligns with the analysis results, indicating that Amazon's ELB (Elastic Load Balancer, similar to LVS) forcibly terminated the TCP session. According to their feedback, if a response exceeds the 350-second threshold (as observed in the packet capture as 630 seconds), Amazon's ELB device sends a reset to the responding party (in this case, the server). The client scripts deployed by the developers did not receive the reset and mistakenly assumed the server connection was still active. Official recommendations for such issues include using TCP keepalive mechanisms to mitigate these problems.

With the official response obtained, the issue was considered fully resolved.

This specific case illustrates how online issues can be highly complex, requiring the capture of critical information—in this instance, packet capture data—to understand the situation as it occurred. Through logical reasoning and the application of reductio ad absurdum, the root cause was identified.

4.8 Performance Optimization

Optimizing code performance requires logical reasoning. Programmers must analyze efficiency, evaluate trade-offs, and make informed decisions to enhance performance. This helps identify

bottlenecks, optimize algorithms, and efficiently use resources [101]. Inefficient code sequences, or performance bugs, cause significant degradation and resource waste, reducing throughput, increasing latency, and frustrating users, leading to financial losses. Diagnosing these bugs is challenging due to non-fail-stop symptoms, often requiring months of expert effort.

4.8.1 Performance Optimization Flowchart

For performance optimization, the process can be simplified into the following flowchart [73]:

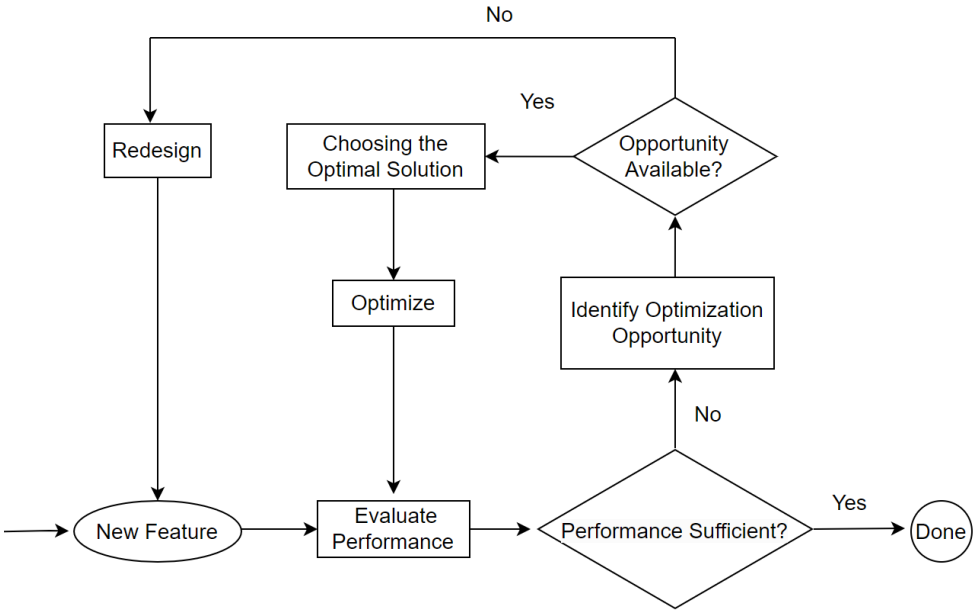


Figure 4-49. Performance optimization flowchart.

When assessing a new performance optimization feature, first test its impact within a single NUMA node. If there is no improvement within a single node but overall improvement in a NUMA environment, the feature may be scalability-related. Conduct theoretical analysis to understand the improvement mechanism. If it can be theoretically explained, the optimization is valid; if not, it is likely ineffective. If the feature doesn't improve performance, look for new optimization opportunities through extensive testing and comparisons. Both performance gains and losses can reveal potential optimization areas. For instance, if a new memory allocation tool causes a performance drop, this indicates its significant impact, necessitating finding the optimal tool. Upon identifying substantial optimization opportunities, evaluate the trade-offs to determine the best method. For example, if PGO decreases throughput under high concurrency, this presents an optimization opportunity. Solutions might include reducing processing time in critical sections through better data structures and

algorithms or completely eliminating latches, which involves extensive code modifications and complex logic, making maintenance challenging.

If optimization opportunities are still not captured, redesign may be necessary. For example, Group Replication adopts an architecture based on the "bucket principle", as illustrated in the figure below:

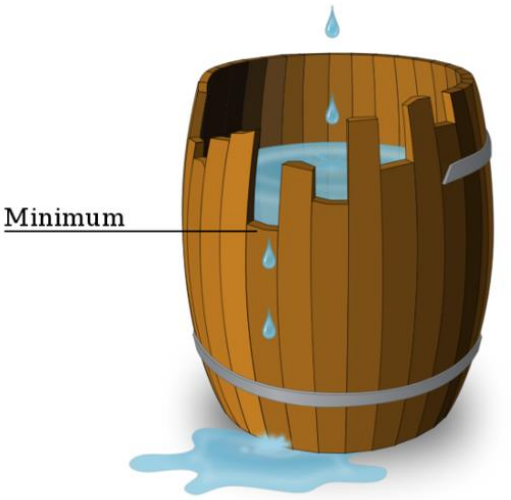


Figure 4-50. bucket principle.

The "bucket principle" refers to a barrel made up of multiple staves of wood, where the amount of water it can hold is determined by its shortest stave, not the longest. For Group Replication, according to the bucket principle, performance is determined by the slowest node in the cluster, often leading to performance that falls short of semisynchronous replication. Without altering the design, finding optimization points becomes difficult. Therefore, a redesign of Group Replication is necessary to fully address the performance deficiencies inherent in the certification database based on the bucket principle.

4.8.2 Throughput vs. Response Time

Throughput and response time have a generally reciprocal but subtly complex relationship [59]. Throughput focuses on resource utilization, examining how effectively server resources process tasks. Response time emphasizes user request responsiveness, impacting user experience.

In performance optimization, two main goals are:

1. **Optimal response time:** Minimize waiting for task completion.

2. **Maximal throughput:** Handle as many simultaneous tasks as possible.

These goals are contradictory: optimizing for response time requires minimizing system load, while optimizing for throughput requires maximizing it. Balancing these conflicting objectives is key to effective performance optimization.

Analyzing the relationship between MySQL semisynchronous throughput and concurrency using the SysBench tool, as illustrated in the figure below, provides insight into how these metrics interact in testing environments.

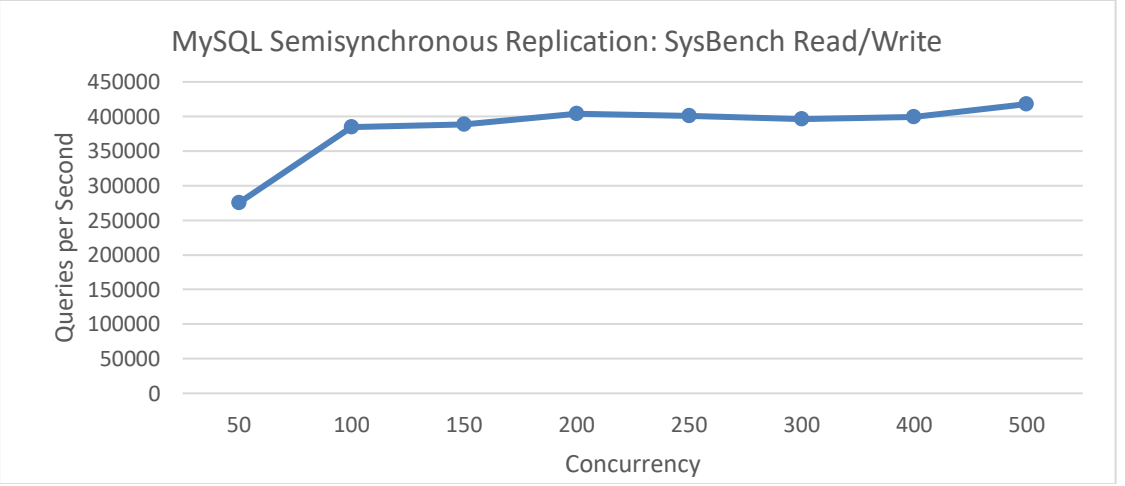


Figure 4-51. Semisynchronous throughput vs. concurrency relationship using SysBench.

The figure shows that throughput peaks at 500 concurrent connections. Similarly, the subsequent figure indicates that response time also peaks at 500 concurrency.

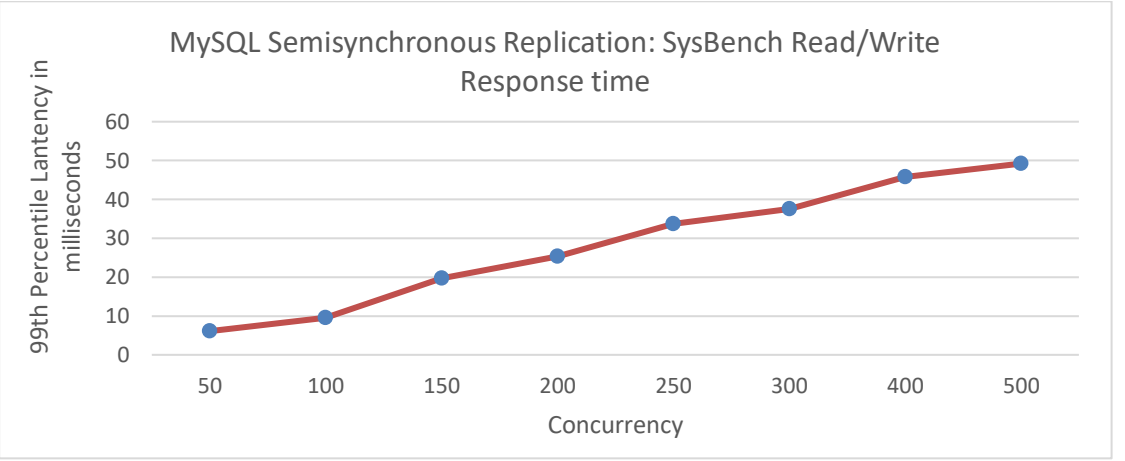


Figure 4-52. Semisynchronous response time vs. concurrency relationship using SysBench.

It is common to observe that high throughput is accompanied by high response times. With increasing computational resources, resource utilization efficiency may decrease and throughput may still only increase gradually. Pursuing higher throughput without considering quality and efficiency is shortsighted. Instead, finding the optimal balance between response time and throughput is crucial.

This balance point, known as the "knee", represents the optimal load level where throughput is maximized with minimal negative impact on response times. For systems handling randomly timed service requests, exceeding the knee value can lead to severe fluctuations in response times and throughput due to minor changes in load. Thus, managing load to remain below the knee value is essential [59].

For the semisynchronous test mentioned above, both throughput and response time are favorable at 100 concurrency. Implementing a transaction throttling mechanism can effectively limit MySQL's maximum concurrent transactions to 100.

4.8.3 Amdahl's Law

In computer architecture, Amdahl's Law provides a formula to predict the theoretical speedup in latency for a task with a fixed workload when system resources are improved [81].

Although Amdahl's Law theoretically holds, it often struggles to explain certain phenomena in the practical performance improvement process of MySQL. For instance, the same program shows a 10% improvement in SMP environments but a 50% improvement in NUMA environments. Measurements were conducted in SMP environments where the optimized portion, accounting for 20% of execution

time, was improved by a factor of 2 through algorithm enhancements. According to Amdahl's Law, the theoretical improvement should be calculated as $\frac{1}{(0.8+\frac{0.2}{2})} = \frac{1}{(0.8+0.1)} = \frac{1}{0.9} = 1.11$ times, with a maximum potential improvement of $\frac{1}{0.8} = 1.25$ times.

In practice, the 10% improvement in SMP environments aligns with theoretical expectations. However, the 50% improvement in NUMA environments significantly exceeds these predictions. This discrepancy is not due to a flaw in the theory or an error but rather because performance improvements in NUMA environments cannot be directly compared with those in SMP environments. Amdahl's Law is applicable strictly within the same environment.

Accurate measurement data is also challenging to obtain [22]. Developers typically use tools like perf to identify bottlenecks. The larger the bottleneck displayed by perf, the greater the potential for improvement. However, some bottlenecks are distributed or spread out, making it difficult to pinpoint them using perf and, consequently, challenging to identify optimization opportunities. For example, Profile-Guided Optimization (PGO) may not highlight specific bottlenecks causing poor performance in perf, yet PGO can still significantly enhance performance.

Taking the optimization of the MVCC ReadView data structure as an example illustrates the challenges in statistical measurements by tools. As depicted in the following figure, this optimization demonstrates substantial improvements in throughput under high concurrency scenarios.

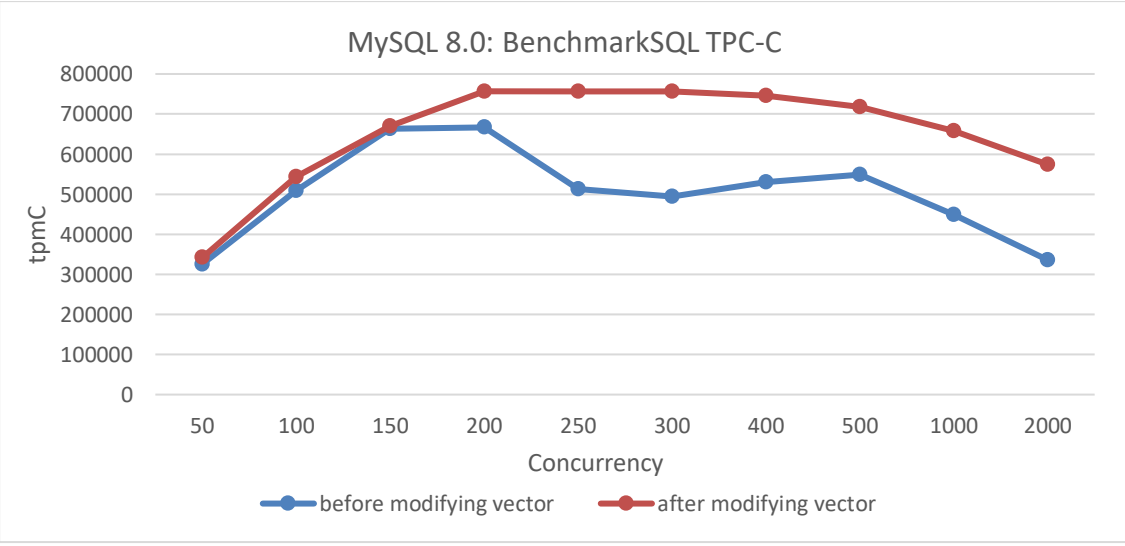


Figure 4-53. Performance comparison before and after adopting the new hybrid data structure.

Let's continue to analyze the perf statistics before optimizing the MVCC ReadView data structure with 300 concurrency, as shown in the specific figure below

```
Samples: 2M of event 'cycles', 4000 Hz, Event count (approx.): 1162569346558 lost: 186383/217301 drop: 62662/93614
Overhead  Shared Object             Symbol
  16.75%  mysqld                    [.] PolicyMutex<TTASEventMutex<GenericPolicy> >::enter
  16.22%  [kernel]                  [k] native_queued_spin_lock_slowpath.part.0
   2.30%  mysqld                    [.] MYSQLparse
   0.98%  mysqld                    [.] cmp_data
   0.89%  mysqld                    [.] buf::Block_hint::buffer_fix_block_if_still_valid
   0.82%  mysqld                    [.] row_search_mvcc
   0.78%  libc-2.28.so              [.] __memmove_avx_unaligned_erms
   0.73%  mysqld                    [.] rec_init_offsets_comp_ordinary
   0.69%  mysqld                    [.] Buf_fetch_normal::get
   0.67%  mysqld                    [.] page_cur_search_with_match
   0.66%  mysqld                    [.] buf_page_get_gen
   0.65%  libpthread-2.28.so        [.] __pthread_mutex_lock
   0.61%  mysqld                    [.] lex_one_token
   0.58%  mysqld                    [.] cmp_dtuple_rec_with_match_low
   0.54%  [kernel]                  [k] _raw_spin_lock
   0.53%  mysqld                    [.] build_template_field
   0.47%  libpthread-2.28.so        [.] __pthread_mutex_cond_lock
   0.46%  libjemalloc.so.2          [.] malloc

11:41:19,624 [Thread-251] INFO   jTPCC : Term-00,
11:41:19,624 [Thread-251] INFO   jTPCC : Term-00,
11:41:19,624 [Thread-251] INFO   jTPCC : Term-00, Measured tpmC (NewOrders) = 494718.72
11:41:19,624 [Thread-251] INFO   jTPCC : Term-00, Measured tpmTOTAL = 1099302.37
11:41:19,624 [Thread-251] INFO   jTPCC : Term-00, Session Start     = 2024-08-04 11:36:19
11:41:19,624 [Thread-251] INFO   jTPCC : Term-00, Session End       = 2024-08-04 11:41:19
11:41:19,624 [Thread-251] INFO   jTPCC : Term-00, Transaction Count = 5497885
```

Figure 4-54. The perf statistics before optimizing the MVCC ReadView data structure.

Perf analysis reveals that the first and second bottlenecks together account for about 33% of the total. After optimizing the MVCC ReadView, this percentage drops to approximately 5.7%, reflecting a reduction of about 28%, or up to 30% considering measurement fluctuations. According to Amdahl's Law, theoretical performance improvement could be up to around 43%. However, actual throughput has increased by 53%.

```
Samples: 1M of event 'cycles', 4000 Hz, Event count (approx.): 791460617510 lost: 536867/539707 drop: 64055/96803
Overhead  Shared Object             Symbol
   3.46%  mysqld                    [.] MYSQLparse
   3.30%  mysqld                    [.] PolicyMutex<TTASEventMutex<GenericPolicy> >::enter
   2.41%  [kernel]                  [k] native_queued_spin_lock_slowpath.part.0
   1.87%  mysqld                    [.] buf::Block_hint::buffer_fix_block_if_still_valid
   1.59%  mysqld                    [.] cmp_data
   1.32%  mysqld                    [.] row_search_mvcc
   1.18%  mysqld                    [.] rec_init_offsets_comp_ordinary
   1.16%  libc-2.28.so              [.] __memmove_avx_unaligned_erms
   1.16%  mysqld                    [.] Buf_fetch_normal::get
   1.03%  mysqld                    [.] buf_page_get_gen
   1.00%  libpthread-2.28.so        [.] __pthread_mutex_cond_lock
   1.00%  mysqld                    [.] page_cur_search_with_match
   1.00%  mysqld                    [.] lex_one_token
   0.96%  mysqld                    [.] cmp_dtuple_rec_with_match_low
   0.79%  mysqld                    [.] build_template_field
   0.77%  mysqld                    [.] rw_lock_s_unlock_func
   0.76%  libpthread-2.28.so        [.] __pthread_mutex_lock
   0.68%  mysqld                    [.] buf_page_optimistic_get
   0.67%  mysqld                    [.] btr_cur_search_to_nth_level
```

```
16:02:14,764 [Thread-67] INFO   jTPCC : Term-00,
16:02:14,764 [Thread-67] INFO   jTPCC : Term-00,
16:02:14,765 [Thread-67] INFO   jTPCC : Term-00, Measured tpmC (NewOrders) = 756757.48
16:02:14,765 [Thread-67] INFO   jTPCC : Term-00, Measured tpmTOTAL = 1681578.8
16:02:14,765 [Thread-67] INFO   jTPCC : Term-00, Session Start     = 2024-08-04 15:57:14
16:02:14,765 [Thread-67] INFO   jTPCC : Term-00, Session End       = 2024-08-04 16:02:14
16:02:14,765 [Thread-67] INFO   jTPCC : Term-00, Transaction Count = 8410023
```

Figure 4-55. The perf statistics after optimizing the MVCC ReadView data structure.

Based on extensive testing, it is concluded that the root cause of discrepancies lies in the inherent inaccuracies of perf statistics. Amdahl's Law itself does not cause misunderstandings under ideal conditions. However, due to measurement errors and human mistakes, applying this law to directly assess performance improvements requires caution. Additionally, Amdahl's Law may vary with changes in the environment.

4.8.4 Performance Modeling

MySQL's complexity makes performance modeling challenging, but focusing on specific subsystems can offer valuable insights into performance issues. For instance, when modeling the performance of major latches in MySQL 5.7, it's found that executing a transaction (with a transaction isolation level of Read Committed) involves certain operations:

- **Read Operations:** Pass through the trx-sys subsystem, potentially involving global latch queueing.
- **Write Operations:** Go through the lock-sys subsystem, which involves global latch queueing for lock scheduling.
- **Redo Log Operations:** Write operations require updates to the redo log subsystem, which also involves global latch queueing.

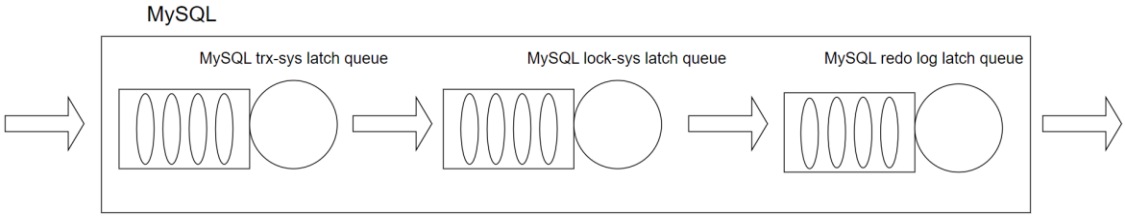


Figure 4-56. The latch queue model in MySQL 5.7.

In MySQL 5.7, poor scalability is mainly due to intense global latch contention among the ***trx-sys***, ***lock-sys***, and ***redo log*** subsystems. For instance, TPC-C performance tests, illustrated in the figure below, reveal poor scalability.

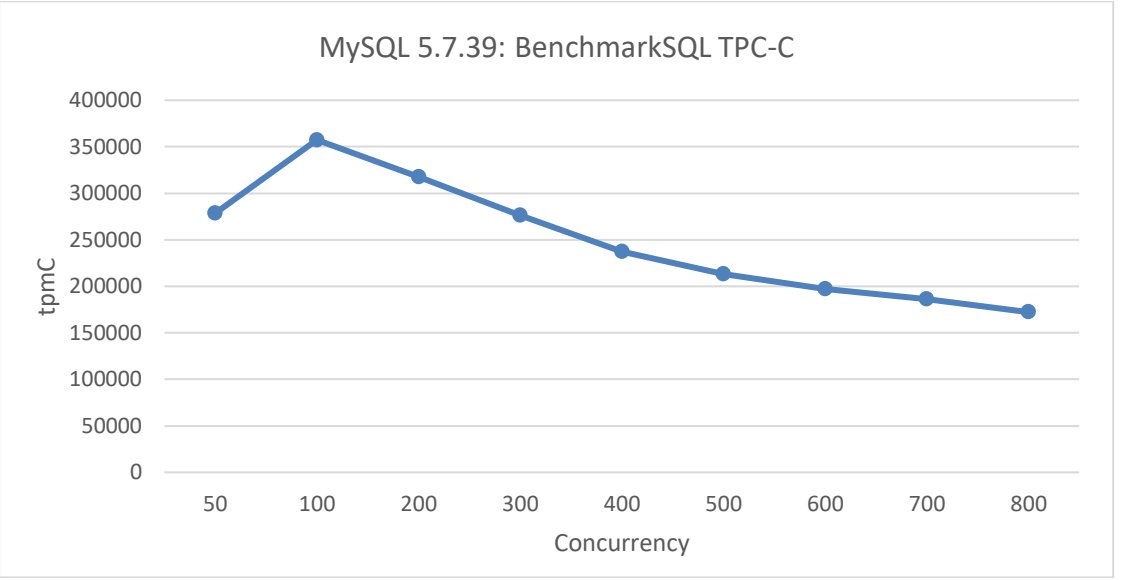


Figure 4-57. Scalability issues in MySQL 5.7.39 during BenchmarkSQL testing.

4.8.5 Challenges in the Limitations of Performance Analysis Tools

When facing performance issues in a program, the typical approach is to use a profiler to identify hot methods and optimize them to enhance performance. If the expected improvements are not realized, the usual suspects are poor memory system interactions or hardware misunderstandings, but the profiler is rarely questioned [23].

Conventional wisdom suggests that PMU sampling provides more reliable results for hot procedures (those with more samples) while colder procedures (with fewer samples) tend to be noisier. This discrepancy arises due to the inherent delays, or "skid", between the PMU overflow interrupt and signal delivery to the performance tool. This issue is prevalent across various architectures, including x86 and ARM, and is a significant source of measurement inaccuracies. In out-of-order processors, the delay in PMU counter overflow interrupts can be particularly pronounced [22].

Profiling alone may not suffice to pinpoint the root cause of performance issues, especially those with complex propagation or computation time wastage at function boundaries. Therefore, performance tools should be used strategically: when a clear bottleneck is identified, a deep analysis should be conducted to address it. However, the absence of an obvious bottleneck does not rule out the existence of underlying issues; alternative methods may be necessary to uncover them.

For instance, Profile-Guided Optimization (PGO) might not highlight optimization opportunities in

perf tools, yet it can still result in substantial performance gains by comprehensively optimizing computational code. Similarly, the trx-sys subsystem may exhibit severe latch bottlenecks due to poorly designed data structures that extend critical section durations. This issue, initially rooted in data structure design, can escalate into intense latch contention, creating a cascading effect.

4.8.6 Mitigating Scalability Issues

Saturated latches degrade multithreaded application performance, causing scalability collapse, particularly on oversubscribed systems (more threads than hardware cores). As threads circulate through a saturated latch, overall performance fades or drops abruptly due to competition over shared system resources like computing cores and last level cache (LLC). Increased threads lead to cache pressure, cache misses, and resource consumption by waiting threads, further exacerbating contention.

To address these scalability issues, consider the following measures:

- Improve critical resource access speed.
- Use latch sharding to reduce conflicts.
- Minimize unnecessary wake-up processes.
- Implement latch-free mechanisms.
- Design the architecture thoughtfully.
- Implement transaction throttling Mechanism.

4.8.6.1 Improve Critical Resource Access Speed

Using a hybrid data structure to improve MVCC ReadView reduces time spent in critical sections, significantly enhancing MySQL's scalability in NUMA environments. The figure 4.10 demonstrates that speeding up access to critical sections within the trx-sys substantially increases high-concurrency throughput in NUMA environments.

4.8.6.2 Latch Sharding to Reduce Latch Conflicts

In MySQL 8.0, latch sharding was implemented for the lock-sys to reduce latch overhead in the lock scheduling subsystem. The figure below compares performance before and after this improvement

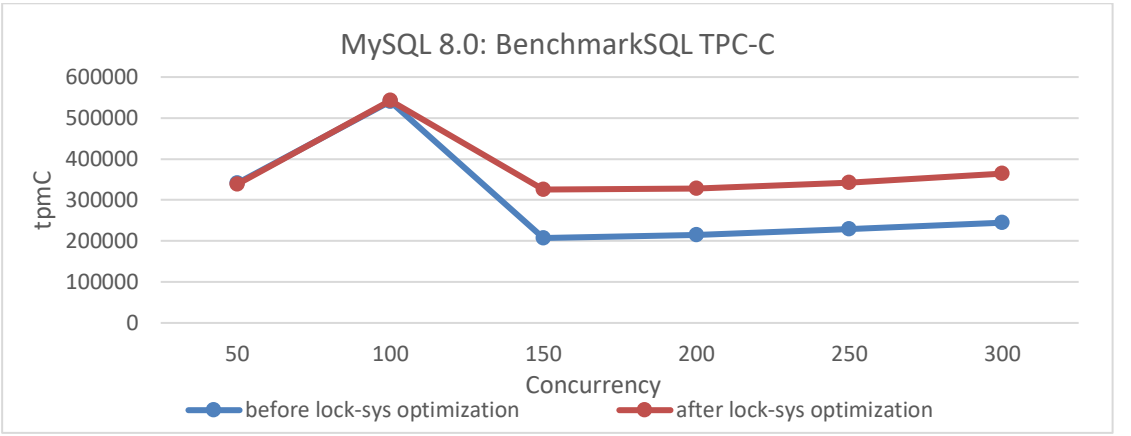


Figure 4-58. Comparison of BenchmarkSQL tests before and after lock-sys optimization.

4.8.6.3 Minimize Unnecessary Wake-up Processes

Binlog group commit adopts an inefficient activation mechanism, resulting in an issue similar to the thundering herd problem. When all waiting threads are activated, only some continue processing while others continue to wait. This leads to the activation of many unnecessary threads, causing significant CPU resource wastage.

The following figure shows the throughput comparison before and after optimizing binlog group commit:

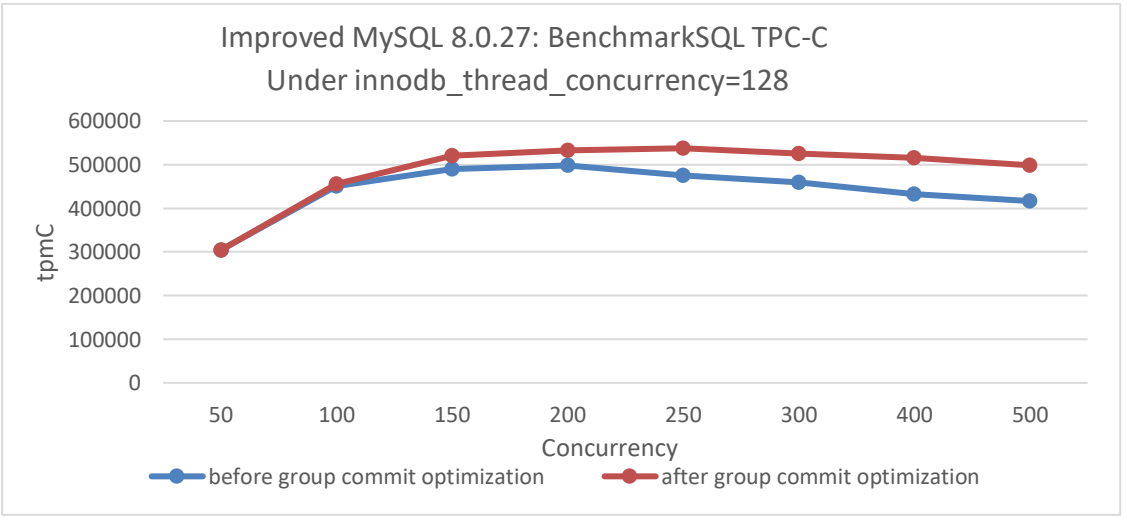


Figure 4-59. Impact of group commit optimization with innodb_thread_concurrency=128.

From the figure, it can be seen that optimizing this activation mechanism has noticeably improved throughput under high-concurrency conditions. For more detailed information, please refer to Section 8.1.2.

4.8.6.4 Latch-Free Processing

MySQL redo log optimization uses latch-free processing to significantly enhance the scalability of the redo log and greatly improve the performance of concurrent writes. The following figure shows the TPC-C throughput comparison before and after redo log optimization. There is a noticeable improvement in low-concurrency scenarios. However, in high-concurrency situations, throughput decreases instead of increasing, mainly due to mutual interference among multi-queue bottlenecks.

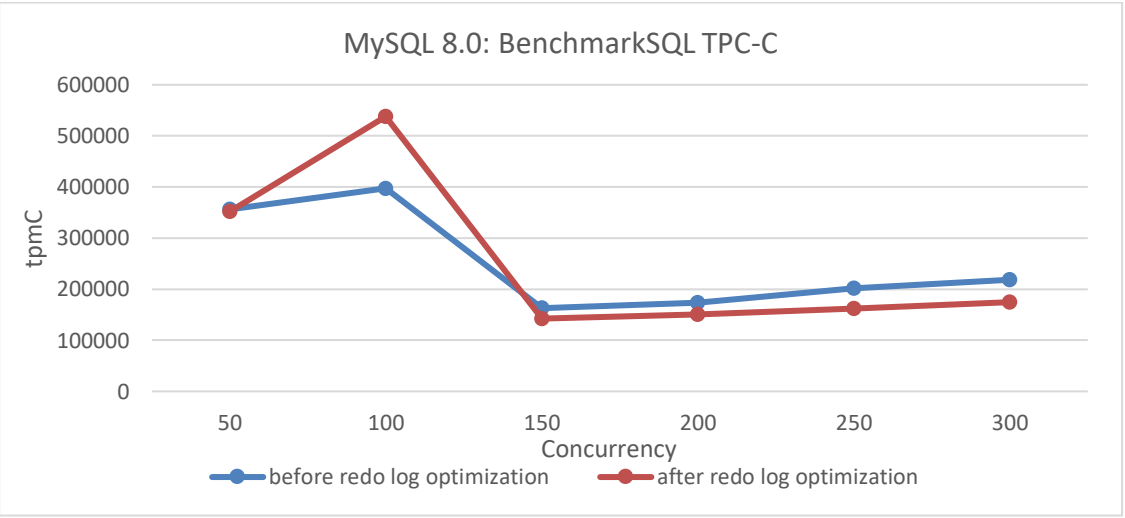


Figure 4-60. Comparison of BenchmarkSQL tests before and after redo log optimization.

For more detailed information, please refer to Section 7.1.1.

4.8.6.5 Implement Transaction Throttling Mechanism

This part is detailed in Chapter 8.3.

4.8.6.6 Design the Architecture Thoughtfully

Architecture design needs to consider long-term needs to avoid difficult future redesigns. For example, MySQL secondaries face numerous design issues, performing poorly in NUMA environments and suffering from architectural shortcomings that delay relay log file handling. These issues, compounded by historical problems, make improving MySQL secondary replay quite difficult.

4.8.7 Optimize Response Time

Only after significantly alleviating scalability issues can response times for user requests be effectively reduced, especially under high-concurrency conditions. Here are the methods to achieve reduced response times.

4.8.7.1 Optimize Data Structures and Algorithms

When addressing performance bottlenecks, leveraging the power of data structures and algorithms is often necessary. In performance analysis, if issues stemming from data structures are identified, significant gains can be achieved by optimizing based on the data's characteristics. Such optimizations tend to be relatively straightforward; for example, optimizing the MVCC ReadView data structure is a typical case.

Regarding algorithms, optimization opportunities are generally hard to find in mature modules. However, in less mature modules, numerous opportunities for optimization often exist. For instance, within Group Replication, there are many opportunities for algorithmic improvements. Two classic examples include optimizing the Paxos algorithm and enhancing the search algorithm in the last committed replay calculation, which will be detailed in subsequent chapters.

4.8.7.2 Emphasize Cache Friendliness

Cache has a significant impact on performance, and maintaining cache-friendliness primarily involves the following principles:

1. **Sequential Memory Access:** Access memory data sequentially whenever possible. Sequential access benefits cache efficiency. For example, algorithms like direct insertion sort, which operate on small data sets, are highly cache-friendly.
2. **Avoid False Sharing:** False sharing occurs when different threads modify parts of the same cache line simultaneously, leading to frequent cache invalidations and performance degradation. This often happens when different members of the same struct are modified by different threads concurrently.

False sharing (FS) is a well-known issue in multiprocessor systems, causing performance degradation in multi-threaded programs running in such environments. The figure below shows an example of false sharing.

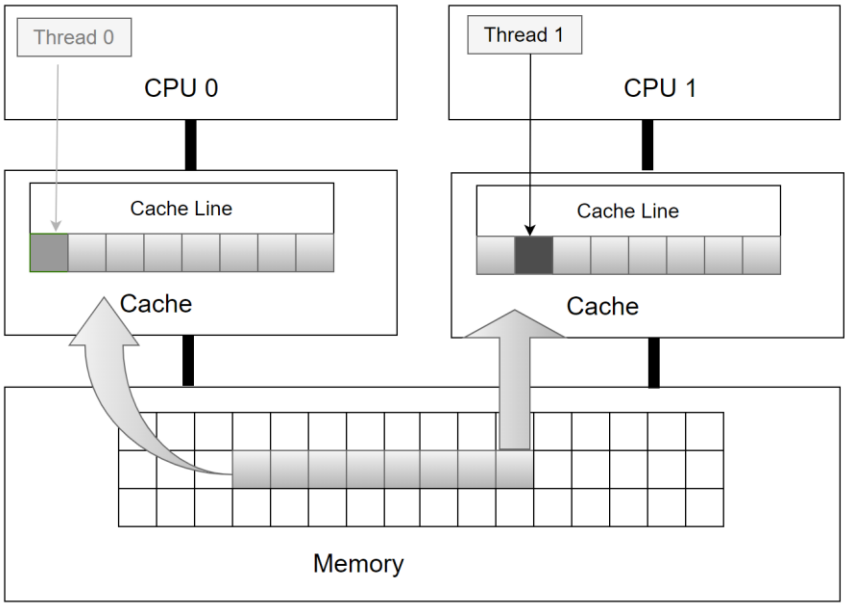


Figure 4-61. Illustrative example of false sharing.

Threads 0 and 1 update variables adjacent to each other on the same cache line. Although each thread modifies different variables, the cache line is invalidated with each iteration. Specifically, when CPU 1 writes a new value, it invalidates CPU 0's cache, causing a write-back to main memory. Similarly, when CPU 0 updates its variable, it invalidates CPU 1's cache by writing back CPU 1's cache line to main memory. If both CPUs repeatedly write new values to their variables, constant invalidations will occur between their caches and main memory. This significantly increases main memory access and causes substantial delays due to the high latency in data transfers between memory hierarchy levels [66].

In MySQL code, specific preventive measures have been implemented to address cache false sharing issues. For example, cache padding improvements for the Performance Schema are detailed in the following git log description.

```
commit 4d46b7560a4d91c85d10ef68ee349e4b1b4a7e17
Author: Marc Alff <marc.alff@oracle.com>
Date:   Fri Nov 8 20:58:48 2013 +0100
      Bug#17766582 PERFORMANCE SCHEMA OVERHEAD IN PFS_LOCK

      This fix is a general cleanup for code involving atomic operations in the
      performance schema, to reduce overhead and improve code clarity.
```

Changes implemented:
...
Added missing PFS_cacheline_uint32 to atomic counters,
to enforce no false sharing happens.

This is a performance improvement.

Removing these cache padding optimizations, as shown in the figure below, serves as the version before cache optimization.

```
diff --git a/storage/perfschema/pfs_global.h b/storage/perfschema/pfs_global.h
index f5b7e65c..a681e3ab 100644
--- a/storage/perfschema/pfs_global.h
+++ b/storage/perfschema/pfs_global.h
@@ -73,7 +73,7 @@ extern bool pfs_initialized;
 */
 struct PFS_cacheline_atomic_uint32 {
   std::atomic<uint32> m_u32;
-  char m_full_cache_line[PFS_CACHE_LINE_SIZE - sizeof(std::atomic<uint32>)];
+  //char m_full_cache_line[PFS_CACHE_LINE_SIZE - sizeof(std::atomic<uint32>)];

   PFS_cacheline_atomic_uint32() : m_u32(0) {}
 };
@@ -84,7 +84,7 @@ struct PFS_cacheline_atomic_uint32 {
 */
 struct PFS_cacheline_atomic_uint64 {
   std::atomic<uint64> m_u64;
-  char m_full_cache_line[PFS_CACHE_LINE_SIZE - sizeof(std::atomic<uint64>)];
+  //char m_full_cache_line[PFS_CACHE_LINE_SIZE - sizeof(std::atomic<uint64>)];

   PFS_cacheline_atomic_uint64() : m_u64(0) {}
 };
@@ -95,7 +95,7 @@ struct PFS_cacheline_atomic_uint64 {
 */
 struct PFS_cacheline_atomic_size_t {
   std::atomic<size_t> m_size_t;
-  char m_full_cache_line[PFS_CACHE_LINE_SIZE - sizeof(std::atomic<size_t>)];
+  //char m_full_cache_line[PFS_CACHE_LINE_SIZE - sizeof(std::atomic<size_t>)];

   PFS_cacheline_atomic_size_t() : m_size_t(0) {}
 };
```

Figure 4-62. Partial reversion of cache padding optimizations.

The performance before and after cache optimization is compared to determine if there is a noticeable difference. For accurate results, MySQL should be started with the Performance Schema enabled during the tests.

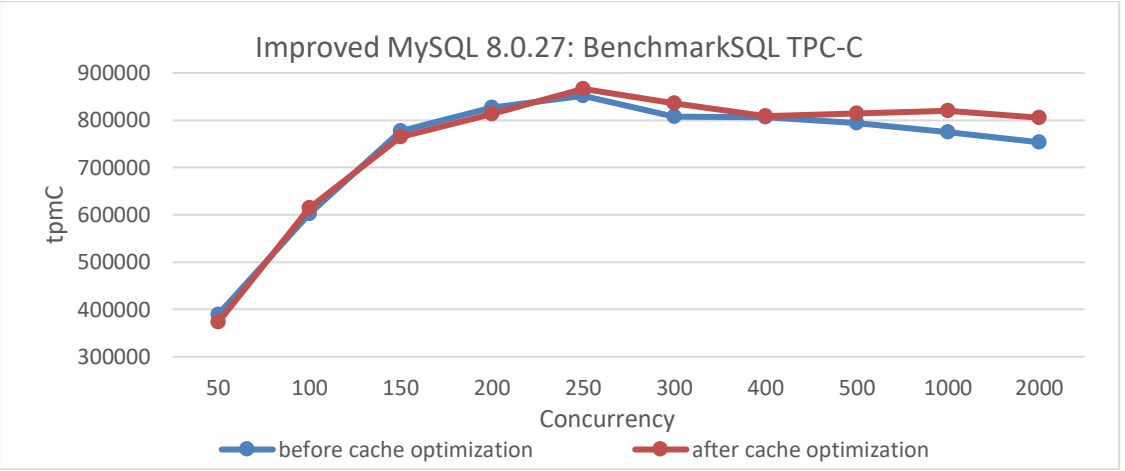


Figure 4-63. Comparison of BenchmarkSQL tests before and after cache optimization.

From the figure, it is evident that the cache padding optimizations show minimal impact under low concurrency conditions but do have an effect under high concurrency. It's worth noting that MySQL has implemented cache padding optimizations in multiple places, and the cumulative performance improvement can be significant.

4.8.7.3 PGO

This part is detailed in Chapter 11.1.

4.8.7.4 Using better memory allocation tools

In NUMA environments, effective memory allocation tools are crucial for performance, both for MySQL primary and secondaries. For more detailed information, please refer to Section 11.3.

4.8.7.5 Reduce Network Latency

For more detailed information, please refer to Section 4.7.6.

4.8.7.6 Summary

To optimize response times, not only can the general techniques discussed above be employed, but also business-specific optimizations can be made. For example, for MySQL, optimizing indexes can reduce response times. Detailed information on this can be found in the next chapter.

4.8.8 Tail Behavior of Response Time: Challenges for SLAs

Another important metric is the tail behavior of response time, defined as the probability that the response time exceeds a certain level x, or $P\{T > x\}$. Understanding this behavior is crucial for setting Service Level Agreements (SLAs), where a company might ensure that response times stay below x with 95% probability. Unfortunately, deriving tail behavior is often difficult [17].

Reducing response times at very high percentiles is challenging because they are easily affected by random events outside of your control, and the benefits are diminishing. Queueing delays contribute significantly to high-percentile response times. A server's limited parallel processing capacity (e.g., limited by CPU cores) means a few slow requests can hold up subsequent ones, causing head-of-line blocking. Even fast subsequent requests will appear slow to the client due to the wait. Therefore, it's essential to measure response times on the client side.

4.8.9 Performance Issues in NUMA Systems

NUMA architectures are commonly used in multi-socket systems to scale memory bandwidth. However, without a NUMA-aware design, programs can experience significant performance degradation due to inter-socket bandwidth contention [23].

MySQL performs poorly in NUMA environments: transactions with the Read Committed isolation level suffer, and MySQL secondary replay is severely impacted by intense latch contention. These issues will be addressed in subsequent chapters.

4.8.10 Performance Improvement is Not Purely Additive

Performance optimization is complex; the improvements from various optimizations do not simply add up. Apart from the non-additive performance case discussed in section 4.7.5, here is an additional example. The following figure shows the performance improvement of MySQL before optimizing MVCC ReadView, with the MySQL spin delay set to 20. There is a significant increase in throughput under high concurrency conditions.

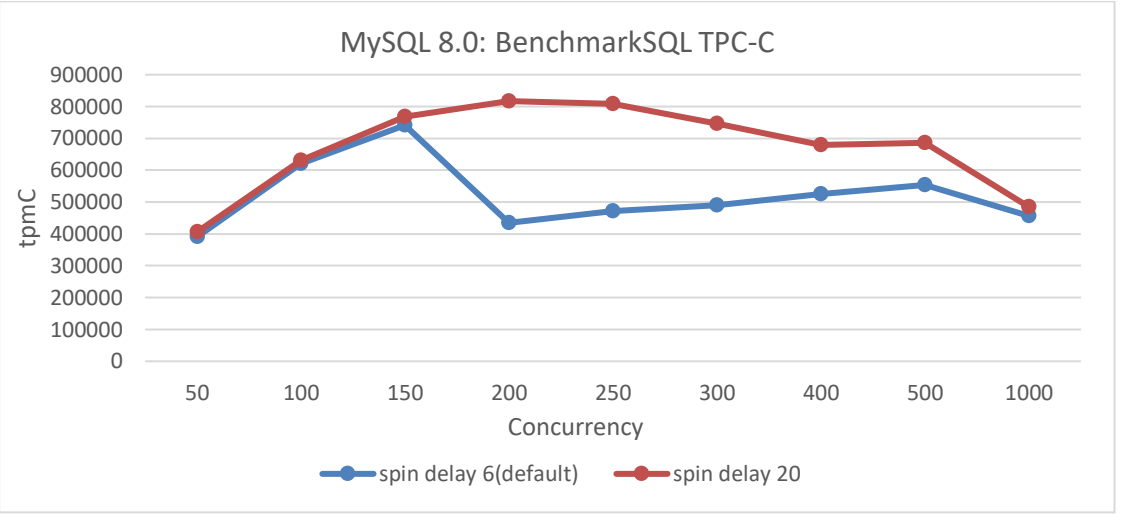


Figure 4-64. Comparison of BenchmarkSQL tests before and after spin delay optimization.

Here is the improvement in MVCC ReadView optimization① itself, as shown in the figure below. It can be seen that there is a more pronounced increase in throughput under high concurrency conditions.

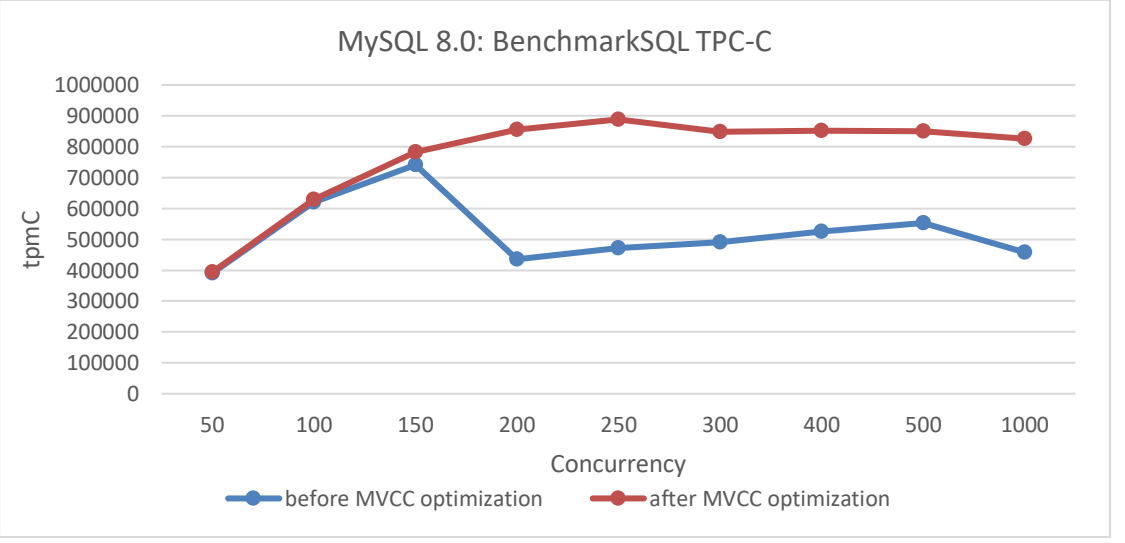


Figure 4-65. Comparison of BenchmarkSQL tests before and after MVCC optimization.

The following figure illustrates the combined effect of these two optimizations on the TPC-C

① This includes all optimizations from section 8.2.

throughput as concurrency increases.

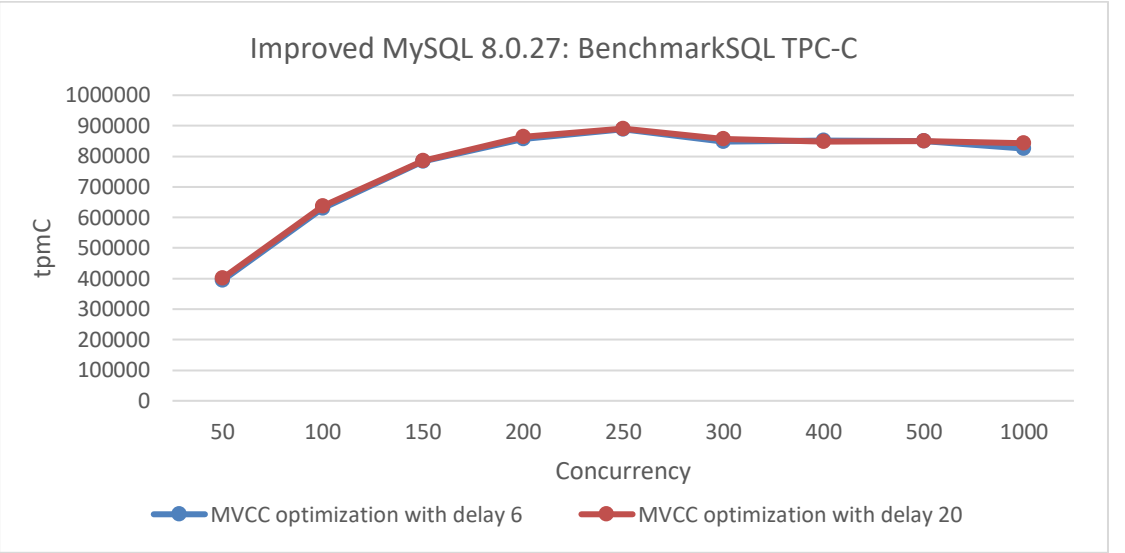


Figure 4-66. Comparison of BenchmarkSQL tests before and after spin delay optimization with MVCC improvements.

From the figure, it can be observed that after setting the MySQL spin delay parameter to 20, the throughput is similar to that with the default MySQL spin delay of 6. The MySQL spin delay parameter uses busy-waiting to reduce thread NUMA cross-node switches in critical sections. With fewer opportunities for NUMA cross-node switches in critical sections due to MVCC ReadView optimization, the effect of the MySQL spin delay parameter naturally diminishes.

4.9 Distributed Theory

MySQL clusters can be viewed as distributed databases and are similarly governed by principles of distributed theory, such as the CAP theorem and distributed consistency.

4.9.1 CAP Theorem

A distributed database has three very desirable properties:

1. Tolerance towards Network Partition
2. Consistency
3. Availability

The CAP theorem states: You can have at most two of these properties for any shared-data system

Theoretically there are three options:

1. Forfeit Partition Tolerance

   The system does not have a defined behavior in case of a network partition.

2. Forfeit Consistency

   In case of partition data can still be used, but since the nodes cannot communicate with each other there is no guarantee that the data is consistent.

3. Forfeit Availability

   Data can only be used if its consistency is guaranteed, which implies the need for pessimistic locking. This requires locking any updated object until the update has been propagated to all nodes. In the event of a network partition, it may take a considerable amount of time for the database to return to a consistent state, thereby compromising the guarantee of high availability.

Forfeiting Partition Tolerance is not feasible in realistic environments, as network partitions are inevitable. Therefore, it becomes necessary to make a choice between Availability and Consistency [32].

The proof process of the CAP theorem is a typical example of logical reasoning. The specific proof process is described below [32]:

> First Gilbert and Lynch defined the three properties:
>
> 1. Consistency (atomic data objects)
> A total order must exist on all operations such that each operation looks as if it were completed at a single instance. For distributed shared memory this means (among other things) that all read operations that occur after a write operation completes must return the value of this (or a later) write operation.
> 2. Availability
> Every request received by a non-failing node must result in a response. This means, any algorithm used by the service must eventually terminate.
>
> 3. Partition Tolerance
> The network is allowed to lose arbitrarily many messages sent from one node to another.
>
> With this definition, the theorem was proven by contradiction:

Assume all three criteria (atomicity, availability and partition tolerance) are fulfilled. Since any network with at least two nodes can be divided into two disjoint, non-empty sets $\{G_1,G_2\}$, we define our network as such. An atomic object o has the initial value v_0. We define a_1 as part of an execution consisting of a single write on the atomic object to a value $v_1 \neq v_0$ in G_1. Assume a_1 is the only client request during that time. Further, assume that no messages from G_1 are received in G_2, and vice versa.
Because of the availability requirement we know that a_1 will complete, meaning that the object o now has value v_1 in G_1.

Similarly a_2 is part of an execution consisting of a single read of o in G_2. During a_2 again no messages from G_2 are received in G_1 and vice versa. Due to the availability requirement we know that a_2 will complete.

If we start an execution consisting of a_1 and a_2, G_2 only sees a_2 since it does not receive any messages or requests concerning a_1. Therefore the read request from a_2 still must return the value v0. But since the read request starts only after the write request ended, the atomicity requirement is violated, which proves that we cannot guarantee all three requirements at the same time. q.e.d.

Understanding the CAP theorem lays a foundation for comprehending issues in distributed systems. For instance, in a Group Replication cluster, achieving strong consistency on every node may require sacrificing availability. The "after" mechanism of Group Replication can meet this requirement. Conversely, prioritizing availability means that during network partitions, MySQL secondaries might read stale data since distributed consistency in reads cannot be guaranteed.

4.9.2 Read Consistency

MySQL secondaries can handle read tasks, but they may not keep up with the pace of the MySQL primary. This can lead to issues with read consistency.

In a distributed environment, the paper "Replicated Data Consistency Explained Through Baseball" [67] describes various types of read operation consistency. Below is a detailed outline of common types of read operation consistency.

Guarantee	Description	Consistency	Performance	Availability
Strong Consistency	See all previous writes.	excellent	poor	poor
Read My Writes	See all writes performed by reader.	okay	okay	okay
Eventual Consistency	See subset of previous writes.	poor	excellent	excellent

Figure 4-67. Common types of read operation consistency.

The figure describes the three most common types of consistency: strong consistency, read-your-writes consistency, and eventual consistency. The pattern observed is that the stronger the consistency, the poorer the performance, and the lower the availability.

When reading data from multiple MySQL secondaries, various consistency read issues can easily arise. These can be mitigated using MySQL proxies. For instance, for the same user session, read operations can be directed to a single MySQL secondary. If the data on that MySQL secondary is not up-to-date, the operation waits, thus avoiding typical consistency read problems.

MySQL has made significant efforts to ensure consistent reads in clustered environments. During MySQL secondary replay, it is crucial to maintain the transaction commit order consistent with the relay log entry order. This means that a transaction can only commit once all preceding transactions have been committed. The ***replica_preserve_commit_order*** parameter in MySQL helps enforce this constraint.

To effectively mitigate consistency read issues, MySQL secondaries should ideally keep up with the pace of the MySQL primary. If the MySQL primary and its secondaries operate at similar speeds, users can quickly access the most recent data from the secondaries. Therefore, combining fast replay on MySQL secondaries with addressing consistency read issues can significantly ease these challenges.

4.9.3 Consensus

For decades, the Paxos algorithm has been synonymous with distributed consensus. Despite its widespread deployment in production systems, Paxos is often misunderstood and proves to be heavyweight, unscalable, and unreliable in practice. Extensive research has been conducted to better understand the algorithm, optimize its performance, and mitigate its limitations [50]. Over time, numerous variants of Paxos have emerged, each tailored to different application scenarios based on majority-based consensus. It is important to keep these variants distinct.

For example, Group Replication in MySQL supports both single-primary and multi-primary modes, requiring consistent read and write operations in single-primary mode. This necessitates a multi-leader Paxos algorithm for consensus. The MySQL's original Mencius algorithm faced performance issues in certain scenarios, leading to the introduction of a single-leader Multi-Paxos algorithm to address these concerns. Maintaining two different variants of the Paxos algorithm simultaneously poses challenges in code maintenance, and numerous regression issues discovered later have validated this viewpoint.

4.9.4 Distributed Transaction

In MySQL, the XA protocol can be used to implement the two-phase commit protocol, ensuring atomicity in distributed environments. Many database products use XA to implement distributed transactions with various transaction isolation levels. However, response times in this context are often not ideal.

As highlighted in "Designing Data-Intensive Applications" [49]: "XA has poor fault tolerance and performance characteristics, which severely limit its usefulness." MySQL XA transactions not only negatively impact response times but also pose challenges for MySQL secondary replay. The second phase of an XA transaction depends on the first phase, affecting concurrent replay on MySQL secondaries, as different phases must proceed serially.

Therefore, it is advisable for MySQL users to avoid XA transactions when possible. It's important to note that while the XA two-phase commit protocol bears some similarities to Paxos algorithms, the XA protocol cannot utilize batching techniques and requires all participants to agree before proceeding with subsequent operations.

4.10 Database Fundamentals

Database management platforms are expected to provide application programmers with an abstraction of ACID transactions, freeing them from concerns about anomalies that might arise from concurrency or failure [31]. This section provides a brief introduction to the fundamentals of general relational databases. Advanced fundamentals of MySQL will be covered in the next chapter.

4.10.1 Relational Model and SQL Language

The relational model, based on predicate logic and set theory, is a fundamental data model in databases. SQL (Structured Query Language) is a highly successful programming language, simplifying database interaction by allowing users to specify their requirements without needing to understand the underlying implementation.

The relational model and SQL are elegantly designed. Their invention required significant imagination to use a relational model and SQL for solving data storage problems, as is common today. Their success is largely due to their scalability and ease of use.

SQL is relatively easy to use and widely accepted in database systems. However, it lacks some complex processing patterns (e.g., iterative training) compared to other high-level machine learning

languages. Fortunately, SQL can be extended to support AI models, and user-friendly tools can be designed to support AI models in SQL statements [100].

Using AI to generate SQL queries is also a promising direction for the continued development of SQL.

4.10.2 Database Security

Security and ease of use often present a trade-off for users, requiring corresponding decisions based on their needs. To enhance security, MySQL utilizes TLS/SSL encrypted communication, ensuring that client-server interactions are encrypted. This increases security but adds complexity to problem analysis. For instance, analyzing encrypted SQL content captured using packet sniffing tools can be challenging.

4.10.3 Database Integrity

Database integrity refers to the logical consistency, correctness, validity, and compatibility of data within a database. It is ensured through various integrity constraints, making database integrity design synonymous with the design of these constraints.

For a standalone MySQL instance, maintaining database integrity is relatively straightforward compared to implementing it in Group Replication multi-primary mode. For instance, as noted in [24], Group Replication does not support the following features.

- **Foreign Keys with Cascading Constraints.** Multi-primary mode groups (members all configured with `group_replication_single_primary_mode=OFF`) do not support tables with multi-level foreign key dependencies, specifically tables that have defined `CASCADING` foreign key constraints. This is because foreign key constraints that result in cascading operations executed by a multi-primary mode group can result in undetected conflicts and lead to inconsistent data across the members of the group. Therefore we recommend setting `group_replication_enforce_update_everywhere_checks=ON` on server instances used in multi-primary mode groups to avoid undetected conflicts.

4.10.4 SQL Rewrite

Many database users, especially those in the cloud, may not write high-quality SQL queries. SQL rewriters aim to transform these queries into more efficient forms, such as pushing down filters or transforming nested queries into join queries [100]. Transforming SQL queries into equivalent high-quality versions is thus a very meaningful endeavor.

4.10.5 SQL Injection

SQL injection is a common and harmful vulnerability to databases. Attackers can exploit this vulnerability to modify or view data beyond their privileges by bypassing authentication or interfering with SQL statements, resulting in actions such as retrieving hidden data, subverting application logic, and performing union attacks [100].

For MySQL, preventing SQL injection involves several strategies. Besides using encrypted communication, users could employ prepared statements to avoid SQL injection. Prepared statements ensure that SQL code is separated from data inputs, significantly reducing the risk of injection attacks.

4.10.6 The Significant Impact of Indexes on Performance

In DBMS, indexes are vital for speeding up query execution, and selecting appropriate indexes is crucial for achieving high performance [100]. Below are the additional indexes created for BenchmarkSQL TPC-C testing, on top of the default indexes:

```
create index bmsql_oorder_index on bmsql_oorder (o_c_id);
```

The additional indexes are created to improve TPC-C throughput. The figure below shows the relationship between TPC-C throughput and concurrency before and afte index addition.

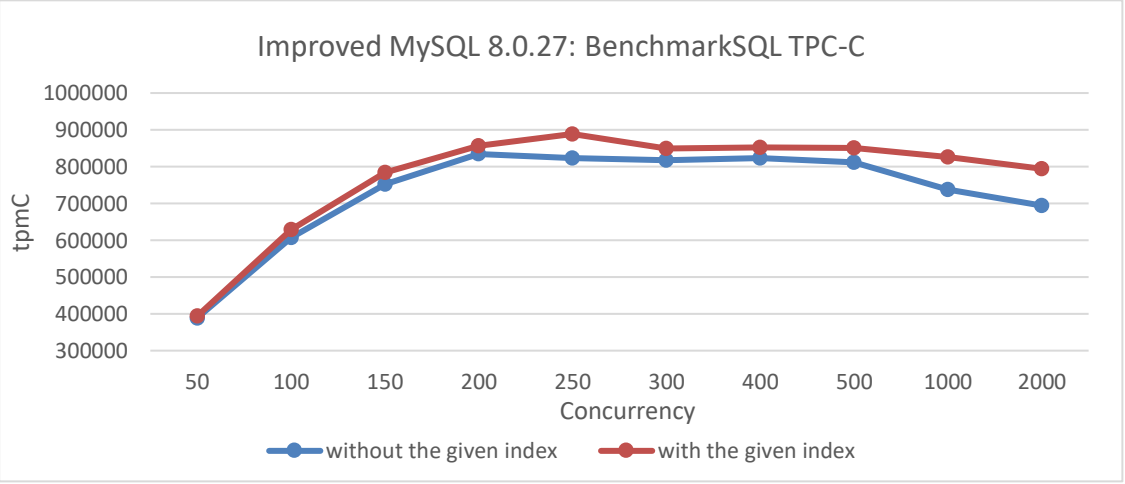


Figure 4-68. Comparison of BenchmarkSQL tests before and after index addition.

From the figure, it can be seen that establishing an effective index can significantly enhance MySQL TPC-C throughput.

4.10.7 State-Machine Replication

The most general approach to providing a highly available service is to use a replicated state machine architecture. With a deterministic service, the state and function are replicated across servers, and an unbounded sequence of consensus instances agrees upon the commands executed. This approach offers strong consistency guarantees and is broadly applicable [57].

State-machine replication is a well-established method for fault tolerance. It replicates a service on multiple servers, maintaining availability despite server failures. However, it has two performance limitations: it introduces overhead in response time due to the need to totally order commands, and service throughput cannot be increased by adding more replicas.

Group Replication is an implementation of state-machine replication. Extensive fixes and improvements have been made based on native MySQL, significantly enhancing stability and performance. Subsequent chapters will detail these improvements to Group Replication.

4.11 Software Architecture Design

Common software architectures include layered architecture, primary-secondary/multi-primary architecture, and event-driven architecture. A standalone MySQL instance processing employs a layered architecture, while clusters utilize either primary-secondary or multi-primary architecture. At the lower level of communication, Paxos employs an event-driven architecture. Therefore, MySQL can be described as a hybrid of multiple architectural styles.

4.11.1 Layered Architecture

Layered architecture is a widely adopted architectural style that separates different concerns into layers to address varying requirements independently. For example, the TCP/IP protocol stack is one of the most successful layered architectures, widely used in the Internet domain. MySQL itself is based on a layered architecture, enabling it to support various types of storage engines. The processing model of Group Replication similarly follows a layered architecture, as depicted in the diagram [24].

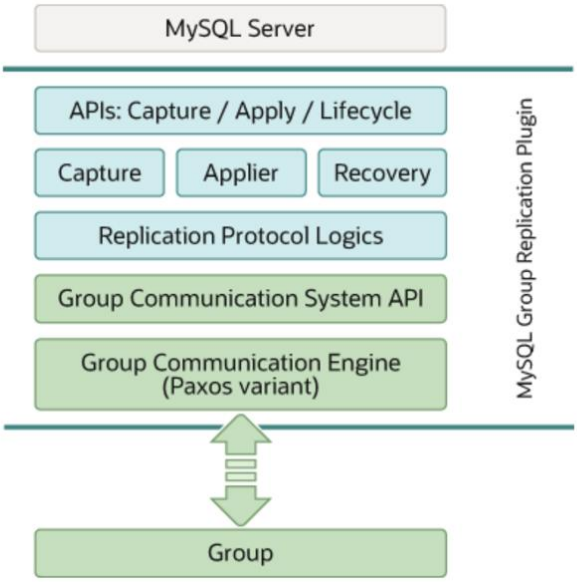


Figure 4-69. Group Replication plugin block diagram.

MySQL Server interacts with the Group Replication plugin through API calls. When a transaction needs to go through the Group Replication process, MySQL Server sends the transaction information to a designated queue via the Group Communication System API. Subsequently, the user thread enters a wait state, waiting for activation by threads responsible for Group Replication. The underlying Paxos layer (referred to as the Group Communication Engine in the diagram) is responsible for broadcasting the queue contents to group members. Once consensus is reached through the Paxos protocol, the upper layers are notified to proceed with further processing.

How is the scalability of this architecture? The following figure illustrates the relationship between Group Replication throughput and concurrency. Additionally, transaction throttling mechanisms are utilized to improve the scalability of InnoDB under scenarios with 2000+ concurrency, ensuring that too many user threads do not enter the InnoDB transaction system.

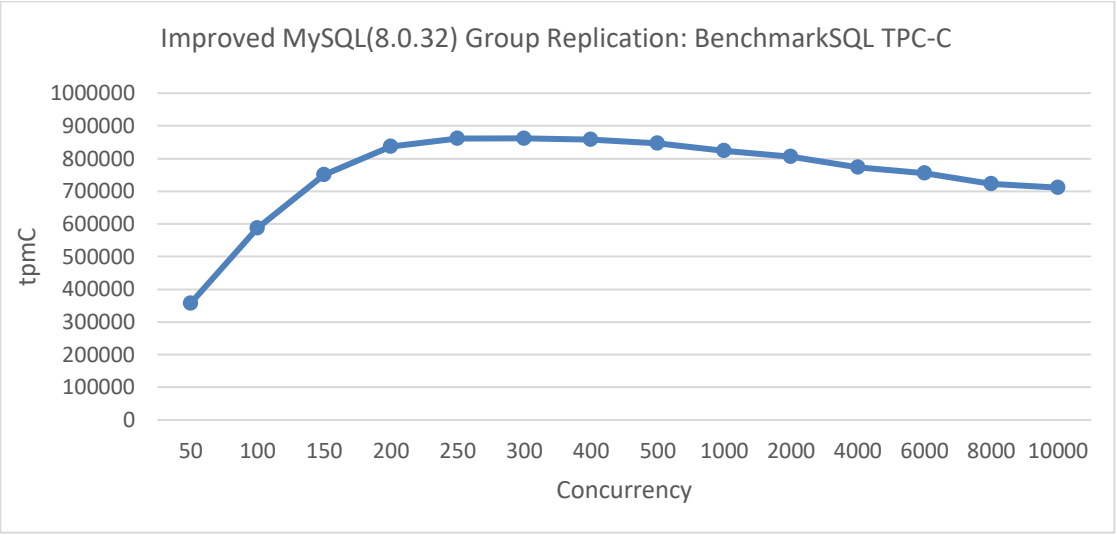


Figure 4-70. Group Replication TPC-C throughput in BenchmarkSQL with transaction throttling mechanisms.

From the figure, it can be seen that the architecture of Group Replication exhibits good inherent scalability, as the throughput does not sharply decrease with an increase in the number of threads.

4.11.2 Primary-Secondary/Multi-Primary Architecture

MySQL asynchronous replication, semisynchronous replication, and Group Replication single-primary all employ primary-secondary architectures. In MySQL, the primary executes transactions while the secondary replays them, and there is no need for synchronous coordination of writes between the primary and the secondary.

In a Group Replication multi-primary architecture, although transactions can be executed on any node, there are several known shortcomings:

1. Lack of a global transaction manager.
2. Limited transaction isolation levels.
3. Potential for traffic skew under heavy write pressure.

According to user feedback, users often utilize Group Replication multi-primary in the following ways:

1. They require that transactions between nodes do not conflict with each other.

2. Despite being multi-primary, transactions are executed on only one node to avoid the overhead of switching primaries.

The following figure shows SysBench's read-write performance over time, where each node accesses the same database and handles both read and write tasks.

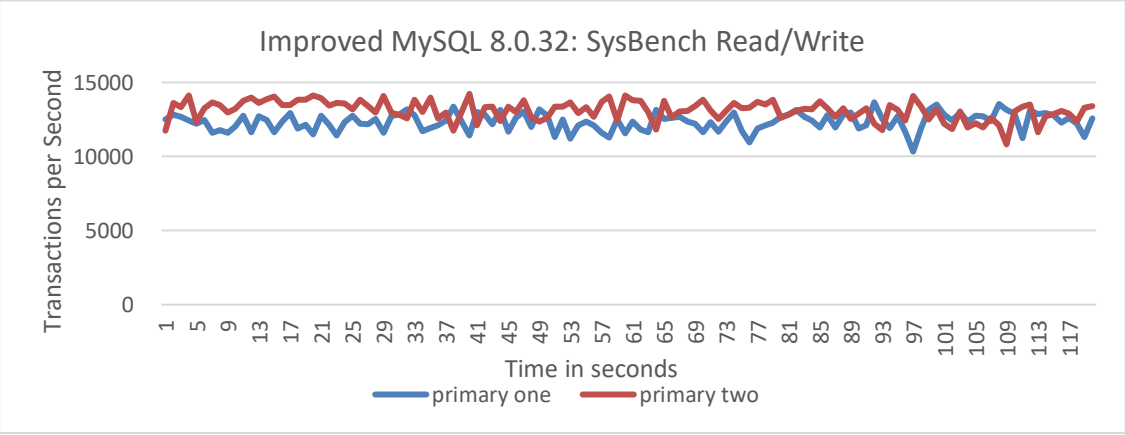


Figure 4-71. Group Replication throughput in SysBench read-write tests over time.

From the figure, it can be seen that the read-write tests between nodes coexist relatively harmoniously. The following figure shows SysBench write-only tests over time, where the Group Replication multi-primary architecture exhibits unstable throughput.

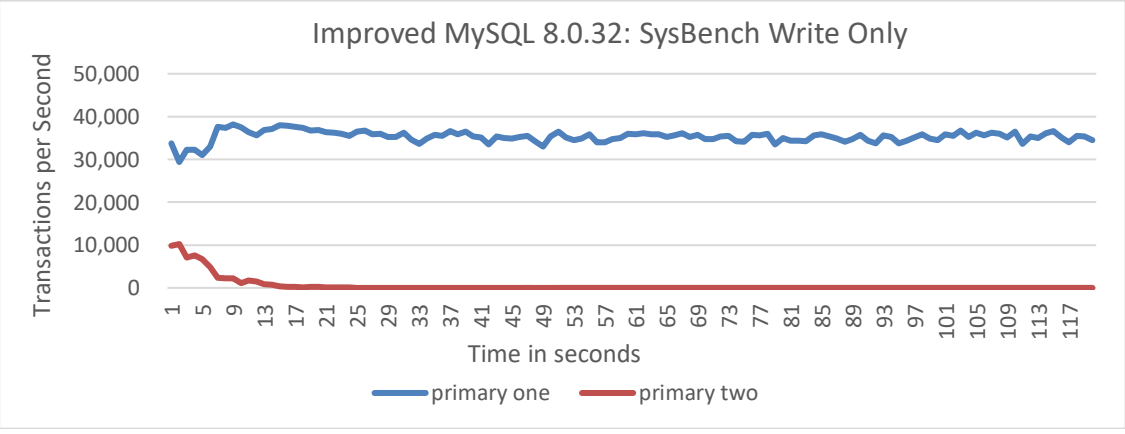


Figure 4-72. Group Replication throughput in SysBench write only tests over time.

Next, let's continue to examine the testing scenario in the Group Replication multi-primary architecture, where transactions between nodes do not conflict. Testing is conducted on primary one

for Database one and on primary two for Database two. This setup ensures that transactions executed by primary one and primary two have no possibility of conflict. The specific results are shown in the following figure:

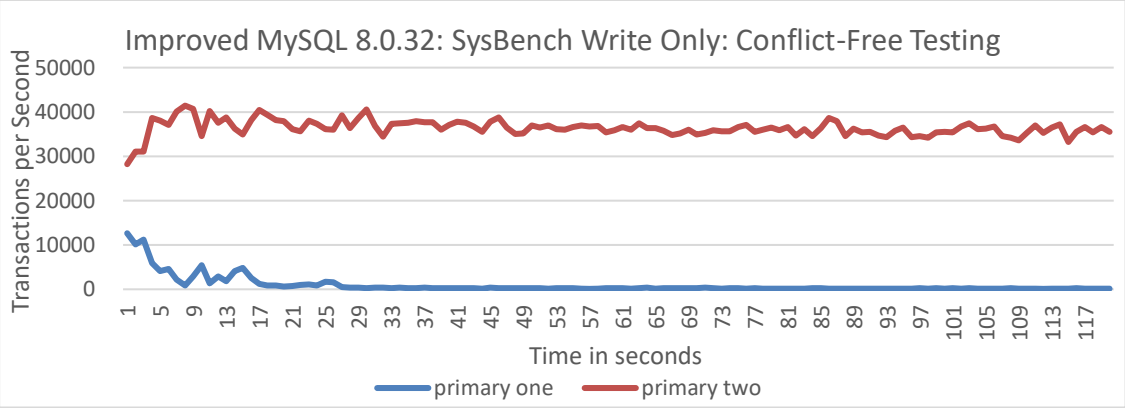


Figure 4-73. Group Replication throughput in SysBench write only tests: high concurrency and no conflicts over time.

From the figure, it can be seen that even without conflicts between transactions on different nodes, there is still throughput skew. The "primary one" node is primarily replaying transactions, severely limiting its capacity to accept new transactions.

Notably, these tests were conducted with 100 concurrency. Reducing the pressure to 10 concurrency alleviates the uneven traffic skew, as shown in the figure below.

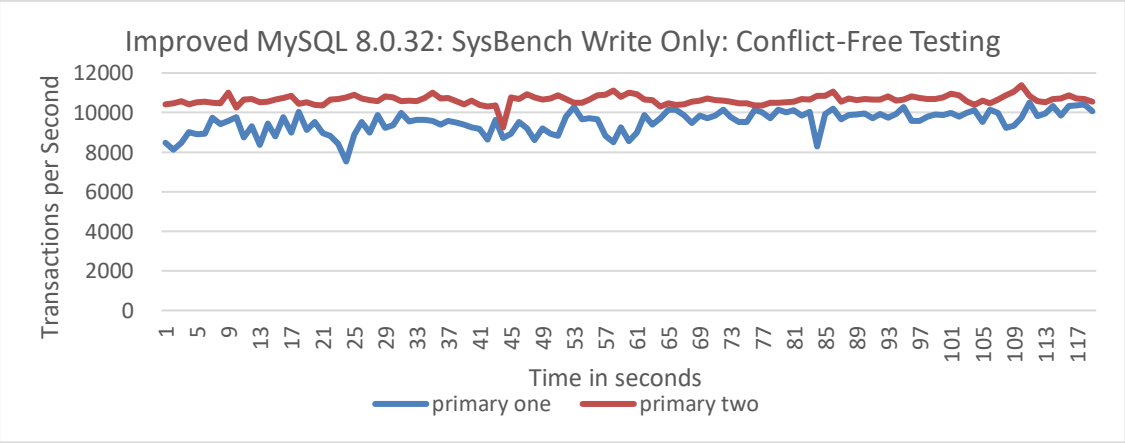


Figure 4-74. Group Replication throughput in SysBench write only tests: low concurrency and no conflicts over time.

The tests indicate that the scalability of the Group Replication multi-primary architecture is problematic and can only support small-scale traffic.

4.11.3 Event-Driven Architecture

Based on event-driven architecture, common in web servers like Nginx, Percona's thread pool can be seen as a rough approximation of event-driven systems. However, event-driven architecture isn't free—it incurs overhead from system calls, and this additional overhead constitutes the cost of using a thread pool.

MySQL's underlying Paxos communication can also be viewed as asynchronous event-driven. This communication operates in a single-threaded mode, requiring the avoidance of any synchronous communication processes. Unfortunately, MySQL has gradually violated these principles in its ongoing feature expansion, leading to throughput issues in certain scenarios due to prolonged synchronous processes dropping to zero. For issues related to Group Replication synchronization, refer to the highlighted code snippet below.

```
/* Try to connect to another node */
static int dial(server *s) {
  DECL_ENV
  int dummy;
  ENV_INIT
  END_ENV_INIT
  END_ENV;

  TASK_BEGIN
  IFDBG(D_BUG, FN; STRLIT(" dial "); NPUT(get_nodeno(get_site_def()), u);
        STRLIT(s->srv); NDBG(s->port, u));

  // Delete old connection
  reset_connection(s->con);
  X_FREE(s->con);
  s->con = nullptr;

  s->con = open_new_connection(s->srv, s->port, 1000);
  if (!s->con) {
    s->con = new_connection(-1, nullptr);
  }
```

4.12 Software Testing

4.12.1 The Importance of Testing

Testing and development are inseparable for MySQL developers. Due to the complex internal mechanisms of MySQL, external testers often struggle to grasp its logic and typically can only conduct black-box testing. Given MySQL's complexity, as many test cases as possible are generally required to reduce the workload of regression testing. Therefore, MySQL developers need to understand how to effectively utilize test cases to test their own programs.

4.12.2 Balancing Test Cases and Development Efficiency

Not every issue necessitates designing test cases; sometimes, the complexity of designing test cases exceeds that of fixing the code. For example, with Paxos-based communication within MySQL, there are no specific test cases, primarily relying on extensive manual testing by developers.

The principle of test case design should be to provide test cases where the cost is reasonable, avoiding them where the cost is excessive. Balancing this relationship effectively enhances development efficiency.

It's worth noting that the test cases for Group Replication have been found to be too lax, leading to significant challenges in regression testing after refactoring. Excessive and lax test cases can consume substantial development time, making it difficult to determine whether the issue lies with the test cases themselves or with the refactored code.

4.12.3 Ensuring Consistency in the Testing Environment

Testing results can fluctuate due to various environmental factors, necessitating consistent environments to ensure fairness. The introduction of SSDs has significantly reduced MySQL's I/O wait times; however, if SSD performance degradation is not monitored during testing, results may diverge from expectations. Using commands such as ***fstrim -a*** can mitigate SSD degradation effects, ensuring tests are less affected.

Linux operating systems utilize I/O caching, which can lead to variability in test results, especially over long intervals between tests. Therefore, it's advisable to minimize the time gap between two tests and repeat the initial test to assess fluctuations. If significant, the test results may become invalid.

In this book, testing tools like BenchmarkSQL or SysBench are typically employed. BenchmarkSQL

increases data volume over time, so for fairness and comparability of performance, tests commence after database refactoring to ensure reproducibility of results.

4.12.4 How to Test Efficiently?

The most efficient testing method is to utilize real online traffic for evaluation directly. However, this can be too risky for databases. A safer approach is to replicate online traffic into a dedicated testing environment.

In Oracle, Database Replay enables testing a system with real production workloads, helping identify potential issues before implementing changes on the production system. Any workload period can be captured with little overhead and used to drive a test system, maintaining the concurrency and load characteristics of the real workload. Maintaining these characteristics is crucial, as current testing solutions often lack synchronization based on data dependencies. Without proper synchronization, the workload does not perform as required, leading to poor coverage and inadequate load, leaving many issues undetected. Database Replay's data-based synchronization makes testing realistic and helps discover potential problems [62].

In MySQL, a common strategy involves taking a MySQL secondary instance offline for testing, configuring the necessary cluster, and replicating online MySQL requests to this new testing primary. The closer the testing primary resembles the production environment, the more accurate the test results. There are various methods to replicate online MySQL requests. This book recommends the open-source tool TCPCopy [105]. By using TCPCopy, many online issues have been successfully mitigated, laying a solid foundation for MySQL proxy enhancements [106]. For testing MySQL clusters, replicating online requests to the testing system using TCPCopy allows us to evaluate whether the modifications achieve the expected outcomes, such as performance improvements, and robustness.

4.12.5 Is Testing About Discovering Issues or Verifying Them?

For excellent developers, testing serves not only to verify known issues but also to uncover new ones. Verification of issues is the initial step to ensure that the software behaves as expected. Subsequently, the focus shifts to actively discovering potential issues within the program, preempting their discovery by testers. Adopting this strategy during the process of improving MySQL fundamentally ensures the quality of MySQL modifications. Practical experience has validated this approach as highly effective. Where possible, replicating online traffic for testing provides a robust means to identify potential issues within the software, further enhancing its overall quality.

Chapter 5: MySQL Internals

To address the numerous issues inherent in MySQL, it is essential to have a solid foundation of knowledge related to MySQL. This chapter provides detailed explanations of MySQL core fundamentals.

5.1 The "Storage Stack" of InnoDB

The following figure depicts the InnoDB storage stack from a developer's perspective. The upper layer primarily consists of the SQL layer, while the lower layer comprises the InnoDB storage engine layer with transaction capabilities. Interaction between the SQL layer and the InnoDB storage engine layer occurs through interfaces. The InnoDB storage engine primarily includes the transaction layer and the mini-transaction layer. InnoDB interacts with the operating system through system functions, and the operating system interacts with the hardware.

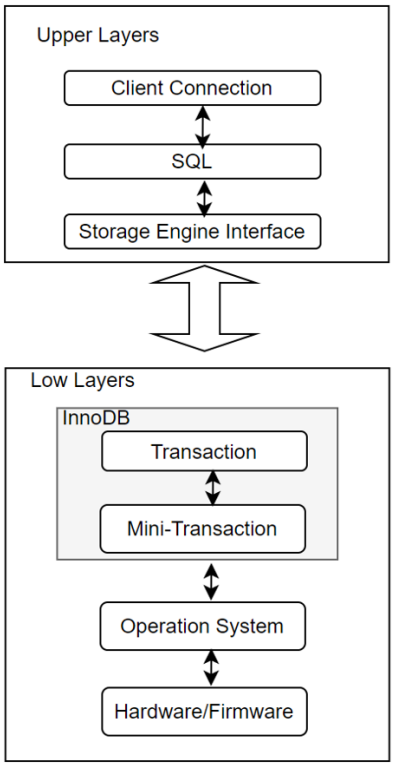


Figure 5-1. The InnoDB Storage Stack.

In the InnoDB storage engine, changes are applied through mini-transactions (mtr), which enable atomic modifications across multiple pages. This approach maintains data consistency during concurrent transactions and database anomalies. Since a single transaction often involves changes to multiple pages, mini-transactions ensure page-level consistency, meaning a single transaction typically comprises multiple mini-transactions.

The figure below illustrates the function call stack relationship between transactions and mini-transactions, showing how transactions use mini-transactions to execute low-level operations.

```
mysqld 3424449 24930974.012558:    1743550 cycles:
            55d7b7fa211d ut_delay+0x1d (/home/wangbin/mysql_old/bin/mysqld)
            55d7b7e8441c mtr_t::Command::prepare_write+0x53c (/home/wangbin/mysql_old/bin/mysqld)
            55d7b7e85e7b mtr_t::Command::execute+0x2b (/home/wangbin/mysql_old/bin/mysqld)
            55d7b7e863e9 mtr_t::commit+0x89 (/home/wangbin/mysql_old/bin/mysqld)
            55d7b7f2c191 row_upd_clust_rec+0xc1 (/home/wangbin/mysql_old/bin/mysqld)
            55d7b7f3156d row_upd_clust_step+0x84d (/home/wangbin/mysql_old/bin/mysqld)
            55d7b7f32f3b row_upd+0x6b (/home/wangbin/mysql_old/bin/mysqld)
            55d7b7f3320b row_upd_step+0x8b (/home/wangbin/mysql_old/bin/mysqld)
            55d7b7ef6685 row_update_for_mysql_using_upd_graph+0x1c5 (/home/wangbin/mysql_old/bin/mysqld)
            55d7b7efb35f row_update_for_mysql+0x3f (/home/wangbin/mysql_old/bin/mysqld)
            55d7b7e0d9c7 ha_innobase::update_row+0x127 (/home/wangbin/mysql_old/bin/mysqld)
            55d7b7e2ae31 non-virtual thunk to ha_innopart::update_row_in_part(unsigned int, unsigned char const*, unsigned char*)+0x31
            55d7b7add857 Partition_helper::ph_update_row+0x147 (/home/wangbin/mysql_old/bin/mysqld)
            55d7b76b730b handler::ha_update_row+0x1ab (/home/wangbin/mysql_old/bin/mysqld)
            55d7b7c0abc0 mysql_update+0x1810 (/home/wangbin/mysql_old/bin/mysqld)
            55d7b7c0cdcd Sql_cmd_update::try_single_table_update+0x1ed (/home/wangbin/mysql_old/bin/mysqld)
            55d7b7c0d106 Sql_cmd_update::execute+0x36 (/home/wangbin/mysql_old/bin/mysqld)
            55d7b7b7dc4a mysql_execute_command+0x8ca (/home/wangbin/mysql_old/bin/mysqld)
            55d7b7b83b95 mysql_parse+0x3e5 (/home/wangbin/mysql_old/bin/mysqld)
            55d7b7b84ec4 dispatch_command+0x1274 (/home/wangbin/mysql_old/bin/mysqld)
            55d7b7b861a0 do_command+0x220 (/home/wangbin/mysql_old/bin/mysqld)
            55d7b7c4cd18 handle_connection+0x298 (/home/wangbin/mysql_old/bin/mysqld)
            55d7b814b7e4 pfs_spawn_thread+0x154 (/home/wangbin/mysql_old/bin/mysqld)
            7ff596e0817a start_thread+0xea (/usr/lib64/libpthread-2.28.so)
```

Figure 5-2. Call stack relationship between transactions and mini-transactions.

It should be emphasized that the transaction layer and the mini-transaction layer together implement the functionality of a complete transaction.

5.2 Transactions

The ACID model outlines key database design principles essential for business data and mission-critical applications. MySQL, with components like the InnoDB storage engine, adheres closely to the ACID model to ensure data integrity and prevent corruption during exceptional conditions such as software crashes and hardware failures. Relying on ACID-compliant features eliminates the need for custom consistency checking and crash recovery mechanisms. However, in cases where additional safeguards exist, ultra-reliable hardware is used, or minor data loss or inconsistency is acceptable, MySQL settings can be adjusted to trade some ACID reliability for increased performance or throughput [24].

To implement a transaction, the following ACID properties must be satisfied:

1. **Atomicity**: Ensures "all or nothing" semantics, meaning either all operations of a transaction

are completed, or none are. This aspect mainly involves InnoDB transactions.

2. **Consistency**: Requires every transaction to maintain the predetermined integrity rules of the database, transforming it from one consistent state to another. Consistency is ensured by the DBMS and involves internal InnoDB processing to protect data from crashes.

3. **Isolation**: Prevents transactions from interfering with each other, ensuring incomplete transactions are not visible to others. Isolation is primarily managed through InnoDB transactions and the isolation level applied to each transaction.

4. **Durability**: Guarantees that once a transaction is committed, it remains so, even in the event of a crash. This aspect involves MySQL software features and the hardware configuration, and it is the most complex to provide specific guidelines for.

In the InnoDB storage engine:

- **Transaction Layer**:
    - **Atomicity, Consistency, and Isolation**: Achieved through locks and ReadView.
    - **Cross-Engine Atomic Commits**: Implemented using XA 2PC, ensuring atomicity between SQL layer binlogs and InnoDB redo logs, forming the basis for crash recovery.
- **Mini-Transaction Layer**:
    - **Atomic, Consistent, and Durable Modifications**: Managed through interactions with redo/undo logs across multiple pages, supporting crash recovery.

Overall, atomicity, consistency, and durability are jointly achieved through both the mini-transaction and transaction layers, while isolation is mainly managed at the transaction layer.

In InnoDB, each transaction is assigned a transaction ID that strictly increases in chronological order. Transaction IDs are generated not only by external transactions but also by various internal operations within MySQL, such as GTID updates triggering internal transactions for persistence.

5.3 Concurrency Control

High-performance transactional systems require concurrent transactions to meet performance demands. Without concurrency control, these systems cannot provide correct results or maintain consistent databases [81].

Concurrency control allows end-users to access a database simultaneously while maintaining the illusion that each transaction runs independently on a dedicated system, ensuring atomicity and isolation.

Two-phase locking (2PL) was the first proven method for ensuring the correct execution of concurrent transactions in a database system. Under 2PL, transactions must acquire locks on database elements before reading or writing them. A transaction needs a read lock to read an element and a write lock to modify it.

Online Transaction Processing (OLTP) systems rely on concurrency control protocols to ensure the serializability of concurrently executed transactions. When two parallel transactions attempt to access the same data item, the concurrency control protocol coordinates their accesses to maintain serializability. Different protocols achieve this in various ways. Locking-based protocols, such as two-phase locking (2PL), associate a lock with each data item. A transaction must acquire all necessary locks (shared or exclusive) before releasing any. Validation-based protocols, such as optimistic concurrency control (OCC), execute a transaction with potentially stale or uncommitted data and validate for serializability before committing [92].

It is worth noting that concurrency control is only one of the several aspects of a DBMS that affects scalability [92].

5.4 Transaction Isolation Level

Transaction isolation is fundamental to database processing. Isolation, the "I" in ACID, balances performance, reliability, consistency, and reproducibility when multiple transactions occur simultaneously [81]. In InnoDB, the traditional four transaction isolation levels are implemented, focusing here on Repeatable Read (RR), Read Committed (RC), and Serializable levels.

MySQL's default isolation level is Repeatable Read. In this level, a ReadView is obtained at the start of the transaction, ensuring consistent data reads throughout. The mechanism uses transaction ID information from ReadView to fetch the specified data version from the undo log, maintaining data consistency.

The most commonly used isolation level in MySQL is Read Committed, which is also Oracle's default. All TPC-C tests in this book use the Read Committed level. In Read Committed, each read operation in a transaction acquires a corresponding ReadView, potentially resulting in different data for identical reads if concurrent modifications occur.

Serializable isolation provides the strongest form of isolation, similar to serial execution. However,

Serializable isolation does not mandate serial execution; transactions can execute in parallel if they do not conflict.

5.5 MVCC

Due to performance and other considerations, databases rarely implement isolation levels based solely on locks. The MVCC + lock method is the most popular implementation, as it allows read requests without locking.

Under MVCC (Multi-Version Concurrency Control), each write operation creates a new version of a tuple, tagged with the transaction's timestamp. The DBMS maintains a list of versions for each element, determining which version a transaction will access during read operations. This ensures a serializable ordering of operations and prevents the rejection of read operations due to overwritten data.

InnoDB, a multi-version storage engine, keeps old versions of changed rows to support concurrency and rollback. This information is stored in undo tablespaces within rollback segments, which contain insert and update undo logs. Insert undo logs are needed only for transaction rollbacks and can be discarded upon commit. Update undo logs are used for consistent reads and can be discarded only when no transaction requires them for building an earlier version of a row.

Regularly committing transactions, including those with consistent reads, is recommended to prevent the rollback segment from growing excessively and filling up the undo tablespace in which it resides [24].

MySQL's InnoDB layer achieves MVCC through hidden fields (transaction ID, rollback pointer, delete flag), undo log files, and a visibility algorithm for records.

5.6 InnoDB Architecture

The following diagram illustrates the in-memory and on-disk structures that comprise the InnoDB storage engine architecture.

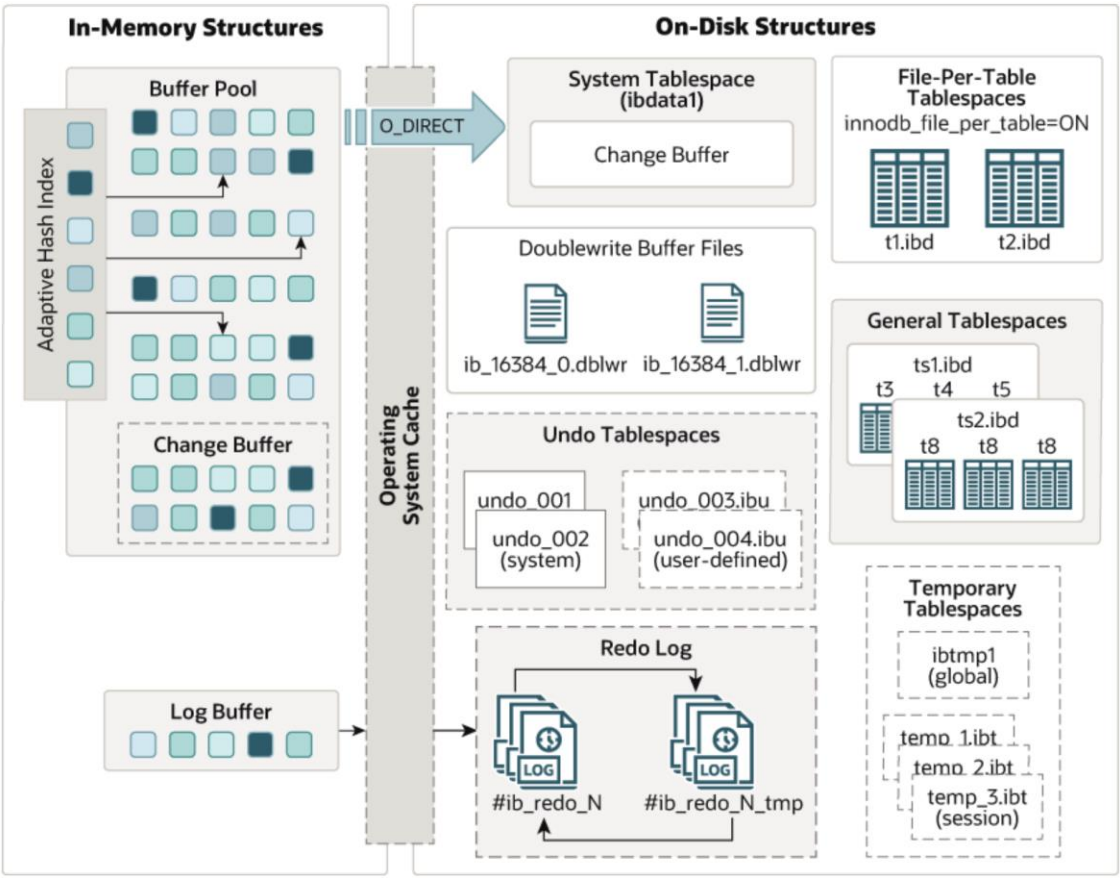


Figure 5-3. InnoDB Architecture.

The InnoDB architecture is divided into two main parts: In-Memory Structures and On-Disk Structures.

In-Memory Structures

1. **Buffer Pool**:
    - Caches table and index data in main memory, allowing frequently accessed data to be read directly from memory, speeding up processing.
    - Divided into pages to hold multiple rows, managed using a linked list and a variation of the least recently used (LRU) algorithm.
    - Key aspect of MySQL tuning for efficient high-volume read operations.

2. **Log Buffer**:
    - Holds data to be written to the log files on disk, periodically flushed to disk.
    - A larger log buffer allows large transactions to run without writing redo log data to disk before committing, reducing disk I/O.
    - Controlled by the ***innodb_flush_log_at_trx_commit*** variable.

On-Disk Structures

1. **Doublewrite Buffer**:
    - An intermediate storage area where pages from the buffer pool are written before their final position in InnoDB data files.
    - Ensures recovery from partial writes due to system crashes or unexpected shutdowns.
    - Efficient as it doesn't double the I/O overhead despite data being written twice.
2. **Redo Log**:
    - Disk-based structure used for crash recovery, correcting data from incomplete transactions.
    - Encodes changes from SQL statements or low-level API calls; replayed automatically during initialization after a crash.
    - Optimizes random writes into sequential log writes (ARIES algorithm) [2], enhancing performance.
    - Redo log files are crucial for acknowledging transaction completion.
3. **Undo Log**:
    - Contains records for undoing changes made by transactions, supporting consistent read operations.
    - Part of undo log segments within rollback segments, residing in undo tablespaces and the global temporary tablespace.
    - Essential for transaction rollbacks and MVCC (Multi-Version Concurrency Control) reads.

By effectively managing these structures, InnoDB achieves a balance of high reliability and performance.

5.7 Log Manager

The log manager is a critical component of modern DBMSs, often prone to bottlenecks due to its centralized design and dependence on I/O. Long flush times, log-induced latch contention, and contention for log buffers in main memory all impact scalability, with no single bottleneck solely responsible for suboptimal performance. Modern systems can achieve transaction rates of 100 ktps or higher, exacerbating the log bottleneck. Existing research offers partial solutions to these bottlenecks, but none provide a fully scalable log manager for today's multicore hardware [6].

The log manager was a major scalability bottleneck in MySQL 5.7. However, MySQL 8.0 underwent significant restructuring in this area, substantially improving MySQL's scalability.

5.8 Lock Scheduling Algorithms

In computing, scheduling is the action of assigning resources to perform tasks [81]. Scheduling algorithms are resource allocation strategies determined by the system's needs, such as FIFO (First In, First Out), Round Robin, and Shortest Job First (SJF). These algorithms are used in operating systems, databases, and networks.

MySQL 5.7 utilized the classic FIFO lock scheduling algorithm. Later versions, starting from MySQL 8.0.20, adopted the CATS (Contention-Aware Transaction Scheduling) lock scheduling algorithm. The purpose of this change was to improve the overall efficiency of MySQL operations and enhance throughput.

Let's analyze the CATS algorithm used in MySQL 8.0. The core idea of the CATS algorithm is to prioritize locks for transactions with higher weighted costs when locks are released. The figure below illustrates the principle mechanism of CATS [102]. Despite transaction t1 having a deeper subgraph, CATS allocates the lock to t2 because completing t2 allows triggering more concurrent transactions to execute.

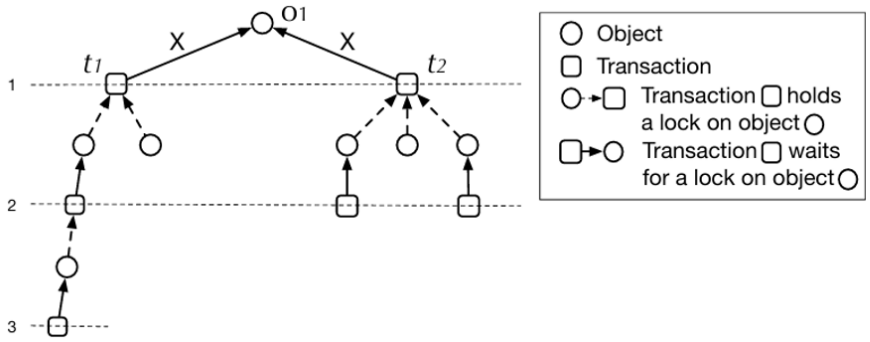


Figure 5-4. Lock scheduling example borrowed from the Paper 'Contention-Aware Lock Scheduling for Transactional Databases'.

The CATS (Contention-Aware Transaction Scheduling) algorithm theoretically has significant effectiveness in scenarios with severe lock contention. It has shown some impact in SysBench Pareto distribution test scenarios, but the exact extent of its effectiveness depends on specific circumstances.

Chapter 7 will subsequently provide a detailed discussion of the CATS scheduling algorithm.

5.9 Binlog File

MySQL enhances its versatility by introducing binlog files at the SQL layer to record transaction modifications, facilitating data replication and disaster recovery. For MySQL transactions, changes are first written to the binlog files and then to the redo log files, with atomicity ensured through the XA 2PC (Two-Phase Commit) mechanism.

Binlog files are crucial for data replication and high availability, supporting asynchronous replication, semisynchronous replication, and Group Replication.

This book focuses on row-based binlog, where transactions are stored as events.

With the advent of fast solid-state drives and techniques like group commit, the impact of log flush I/O times has lessened.

5.10 Group Commit Mechanism

MySQL introduced the binlog group commit mechanism to reduce the number of disk I/O operations by combining multiple binlog flush operations when multiple transactions are committed simultaneously. This approach reduces disk I/O by postponing log access to stable storage, gathering multiple commits in memory, and issuing a single write and flush for a set of transactions.

Log group commit strategies enhance disk performance by aggregating multiple log flush requests into a single I/O operation, reducing the frequency of disk accesses. However, group commit does not eliminate unwanted context switches, as transactions block pending notification from the log rather than blocking directly on I/O requests [6]. Efficient activation mechanisms are needed to reduce context switches, but the current MySQL implementation is inefficient. This issue will be thoroughly explored in Chapter 8.

5.11 Execution Plan

An execution plan details how a SQL statement is executed after optimization by the MySQL query optimizer. Depending on the table structure, indexes, and WHERE clause conditions, the optimizer considers various techniques to perform efficient lookups. Queries on large tables can be executed without reading all rows, and joins can be performed without comparing every row combination.

The MySQL query optimizer is designed for simple, OLTP-type queries and has limitations with complex queries. For instance, join order optimization in practical applications uses only left-deep plans and a greedy algorithm. The figure below illustrates MySQL query optimization and execution architecture [80]. The Parser and Resolver layers handle syntax checking, name resolution, access control, data types, and string collations. During the Prepare phase, logical transformations occur, such as merging derived tables, predicate pushdown, and converting subqueries.

Figure 5-5. MySQL query optimization and execution architecture borrowed from the Paper 'Integrating the Orca Optimizer into MySQL'.

Cost-based Optimization, which is limited to one SELECT block at a time, determines the best join

order, join method, and table access method. The optimizer generally considers only left-deep plans and performs aggregation after all tables are joined. Plan Refinement involves pushing selection conditions into tables and indexes, avoiding sorts if index scans deliver sorted rows, and adding aggregations, group-level filtering, and row limit enforcement.

Heuristics might miss the optimal plan, leading to higher execution times. Join Order Optimization has been extensively studied, with parallel approaches developed for multicore architectures. Due to the NP-hard nature of join order optimization, heuristic solutions and limited search spaces are used.

In general, there is still a lot of optimization potential in MySQL's execution plans. MySQL continues to explore new approaches, which have been reflected in MySQL 8.0.

5.12 Partitioning

Partitioning allows you to distribute table data across a file system based on rules you define, effectively storing different parts of a table as separate tables in various locations. This division, governed by a partitioning function, can use modulus, range or list matching, internal hashing, or linear hashing. The function, specified by the user, takes a user-supplied expression as its parameter, which can be a column value, a function acting on one or more column values, or a set of column values [24].

The benefits of using partitioning are as follows:

1. Enabling storage of more data than a single disk or file system partition can hold.
2. Simplifying data management by allowing easy removal of obsolete data through dropping partitions, and facilitating the addition of new data by adding partitions.
3. Optimizing queries by limiting searches to specific partitions that contain relevant data.

MySQL partitioning not only offers these benefits but also reduces latch contention for large tables under high concurrency. The following figure shows the impact on TPC-C throughput after partitioning a large table in BenchmarkSQL.

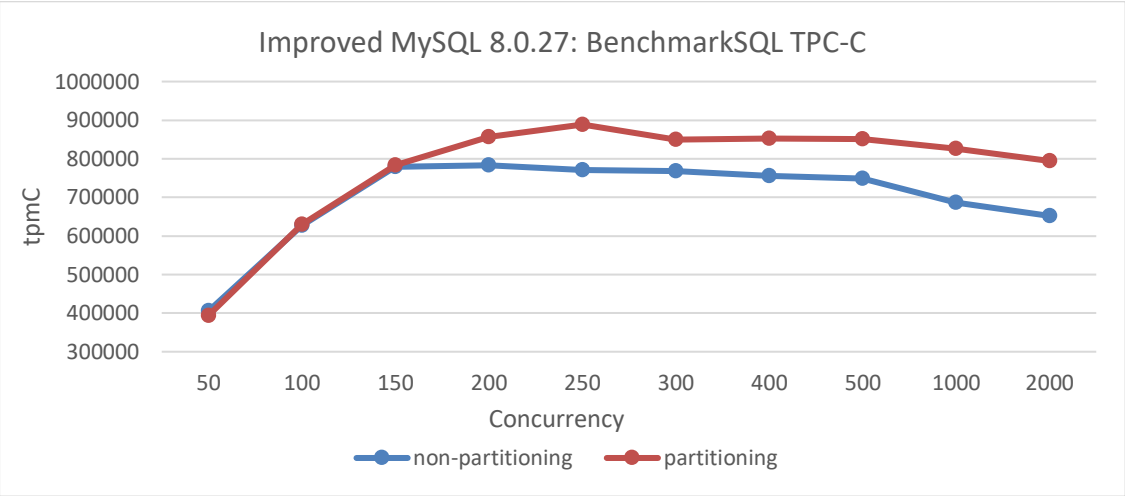


Figure 5-6. Comparison of BenchmarkSQL tests before and after partitioning.

The figure shows that partitioning has minimal impact under low concurrency. However, when concurrency exceeds 150, partitioning significantly improves throughput by alleviating latch conflicts in large tables.

Unless stated otherwise, all TPC-C tests in this book use partitioned large tables. Each table has its own latch, and partitioning employs latch sharding to reduce latch conflicts under high concurrency, preventing latch contention from affecting performance tests on large tables.

5.13 Coordination Avoidance

Minimizing coordination between concurrently executing operations is crucial for maximizing scalability, availability, and performance in database systems. However, coordination-free execution can compromise application correctness and consistency. While serializable transactions maintain correctness, they are not necessary for all applications and can limit scalability [39].

5.14 Disaster Recovery

Disaster recovery ensures a database can be brought back online after an outage. For MySQL, this involves timely flushing of binlog and redo logs, as well as writing to the doublewrite buffer to prevent recovery issues caused by damaged data pages.

5.15 Idempotence

Database code that creates or alters tables and routines should be idempotent to avoid issues if applied

multiple times. Idempotence prevents duplicate data creation during sync failures by recording progress by batch, rather than by individual record. When a sync is interrupted, the process must often restart at the beginning of the last batch, leading to reprocessing some data.

Here's an example of MySQL secondary replay. In the following code snippet, MySQL achieves idempotence during the replay process on the secondary.

```
const bool skip_transaction = is_already_logged_transaction(thd);
if (gtid_next_list == nullptr) {
  if (skip_transaction) {
    skip_statement(thd);
    return GTID_STATEMENT_SKIP;
  }
  return GTID_STATEMENT_EXECUTE;
} else {
```

The ***is_already_logged_transaction*** function is called to determine if a transaction has already been executed. If it has, ***skip_transaction*** is set to true. Consequently, the subsequent process immediately returns ***GTID_STATEMENT_SKIP***, halting further replay of the transaction.

5.16 Thread Pool

MySQL executes statements using one thread per client connection. As the number of connections increases beyond a certain threshold, performance degrades. The following figure shows the TPC-C throughput versus concurrency testing for MySQL 5.7.36.

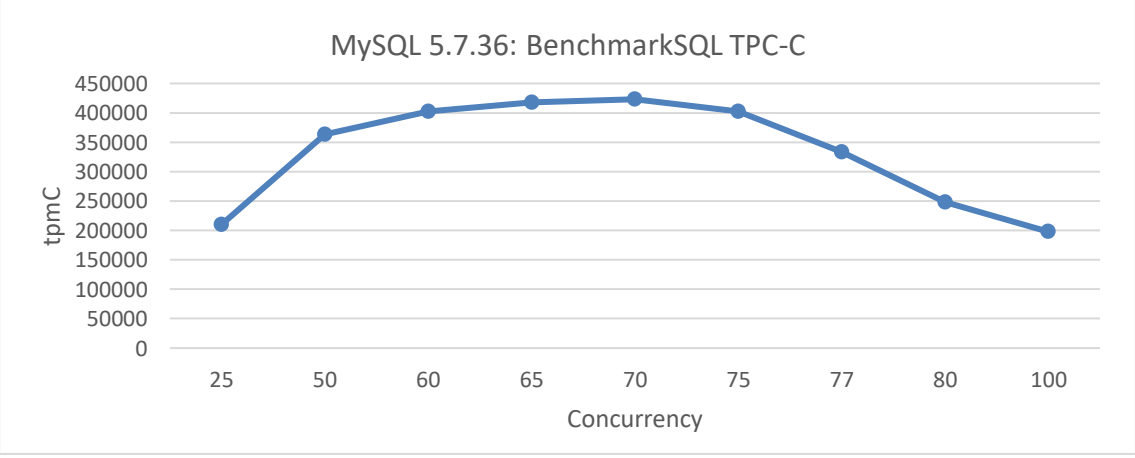


Figure 5-7. TPC-C throughput vs. concurrency in MySQL 5.7.36.

After a concurrency level of 75, the throughput sharply declines, confirming the above conclusion.

Thread pooling reuses a fixed number of threads to handle multiple client connections, reducing overhead and avoiding contention and context switching [56]. The MySQL Thread Pool separates user connections from threads. Each user connection no longer has a dedicated OS thread. Instead, the Thread Pool consists of Thread Groups, with a default of n groups. User connections are assigned to a Thread Group in a round-robin fashion. Each Thread Group manages a subset of connections, with one or more threads executing queries from those connections.

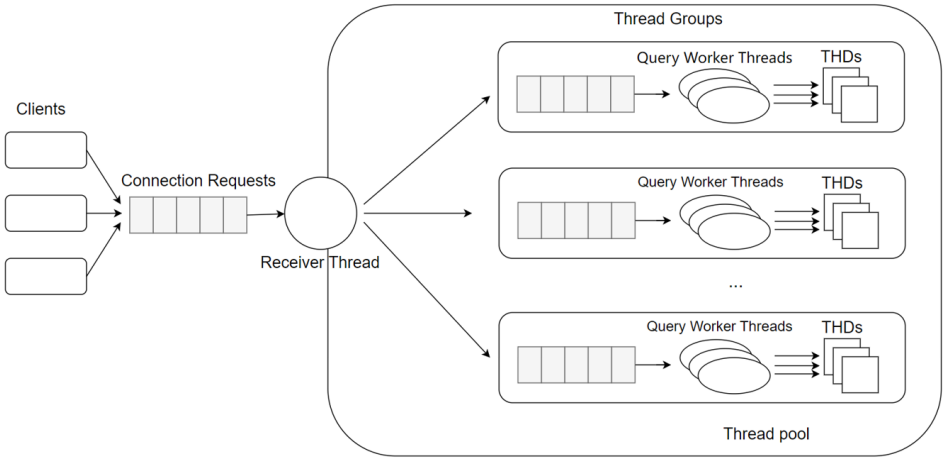


Figure 5-8. Thread pool model.

The Percona thread pool was widely used in MySQL 5.7, but with MySQL 8.0's improved scalability, its role has diminished. MySQL 8.0 introduced new thread pool modes designed to prevent performance degradation as user connections increase. The "Max Transaction Limit" feature limits the number of concurrently executing transactions, which improves overall throughput by reducing data locks and deadlocks on heavily loaded systems [56]. Thus, controlling user thread entry into the InnoDB storage engine is key to alleviating MySQL scalability issues.

5.17 Traditional Cluster

Traditional MySQL clusters rely on asynchronous and semisynchronous replication, which are straightforward and easy for maintenance personnel to manage.

5.17.1 Asynchronous Replication

Traditional MySQL replication uses a simple source to replica approach, with the primary applying transactions and then asynchronously sending them to the secondaries to be applied. This shared-nothing system ensures all servers have a full copy of the data by default [24].

Asynchronous replication offers better write scalability but at the cost of lower data coherence. The following figure is the flowchart of asynchronous replication [24]. The primary continues executing without waiting for acknowledgment from the secondary server, resulting in user SQL query response times comparable to a single server. However, this can lead to data loss if the primary fails before the secondary has received the latest data.

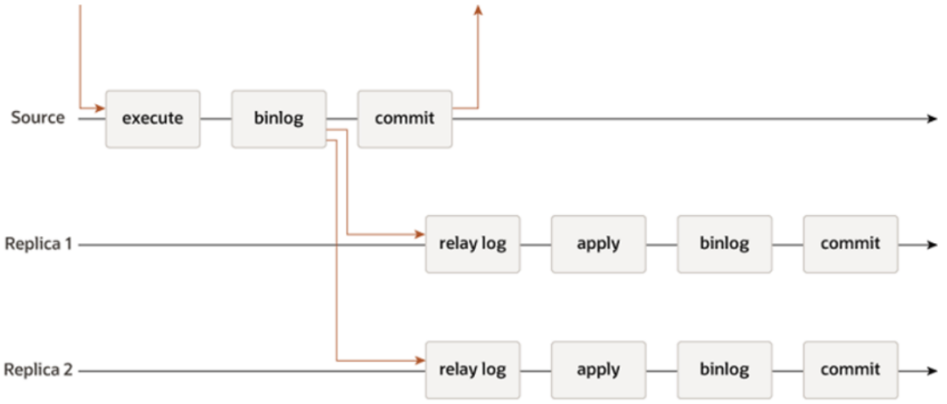


Figure 5-9. MySQL asynchronous replication.

When asynchronous replication is used, if the primary fails and a new leader is chosen, unreplicated writes from the old leader may be lost. This can cause conflicts and durability issues. Often, the solution is to discard the old leader's unreplicated writes, which may not meet clients' durability expectations. If both leaders accept writes without conflict resolution, there's a risk of both nodes shutting down if not properly managed [49].

5.17.2 Semisynchronous Replication

To address data loss in asynchronous replication, MySQL introduced semisynchronous replication. With semisynchronous replication, a transaction commit requires the corresponding binlog to be delivered to at least one MySQL secondary before proceeding. This ensures that at least one secondary has the most recent data.

In MySQL semisynchronous replication, the secondary sends an ACK reply to the primary only after the relay log is written to disk. The primary waits for at least one ACK reply before continuing the transaction. This introduces extra latency from network time, as well as the secondary processing binlog events and writing them to disk.

In traditional high availability setups, semisynchronous replication can be cumbersome and complex.

Meta, for instance, has highlighted these issues in their implementation of high availability based on the Raft protocol [71].

Since semisynchronous replication alone doesn't fully address high availability issues, many third-party tools have emerged, and MySQL has introduced Group Replication.

5.17.3 How Scalable is Semisynchronous Replication?

Here is the relevant code showing the process semisynchronous replication goes through before sending an ACK response:

```
while (!io_slave_killed(thd, mi)) {
  ulong event_len;
  ...
  THD_STAGE_INFO(thd, stage_waiting_for_source_to_send_event);
  event_len = read_event(mysql, &rpl, mi, &suppress_warnings);
  ...
  THD_STAGE_INFO(thd, stage_queueing_source_event_to_the_relay_log);
  event_buf = (const char *)mysql->net.read_pos + 1;
  ...
  if (RUN_HOOK(binlog_relay_io, after_read_event,
               (thd, mi, (const char *)mysql->net.read_pos + 1, event_len,
                &event_buf, &event_len))) {
    mi->report(ERROR_LEVEL, ER_REPLICA_FATAL_ERROR,
               ER_THD(thd, ER_REPLICA_FATAL_ERROR),
               "Failed to run 'after_read_event' hook");
    goto err;
  }

  ...
  QUEUE_EVENT_RESULT queue_res = queue_event(mi, event_buf, event_len);
  if (queue_res == QUEUE_EVENT_ERROR_QUEUING) {
    mi->report(ERROR_LEVEL, ER_REPLICA_RELAY_LOG_WRITE_FAILURE,
               ER_THD(thd, ER_REPLICA_RELAY_LOG_WRITE_FAILURE),
               "could not queue event from source");
    goto err;
  }
  ...
  if (RUN_HOOK(binlog_relay_io, after_queue_event,
               (thd, mi, event_buf, event_len, synced))) {
    mi->report(ERROR_LEVEL, ER_REPLICA_FATAL_ERROR,
               ER_THD(thd, ER_REPLICA_FATAL_ERROR),
               "Failed to run 'after_queue_event' hook");
    goto err;
```

```
  }

  ...
  thd->mem_root->ClearForReuse();
}
```

In the binlog file, a transaction consists of multiple events. For a TPC-C transaction, it is normal to have dozens of events. Each event goes through processes like ***read event***, ***after_read_event***, ***queue event***, and ***after_queue_event***. The more events a transaction contains, the longer the processing time, and all these events are processed by a single thread. This limitation in single-threaded processing, coupled with event-based handling, means that semisynchronous replication has limited computational capacity and poor scalability.

The following figure shows the TPC-C throughput versus concurrency testing for semisynchronous replication. It can be observed that the scalability of semisynchronous replication is very weak, far inferior to that of refactored Group Replication.

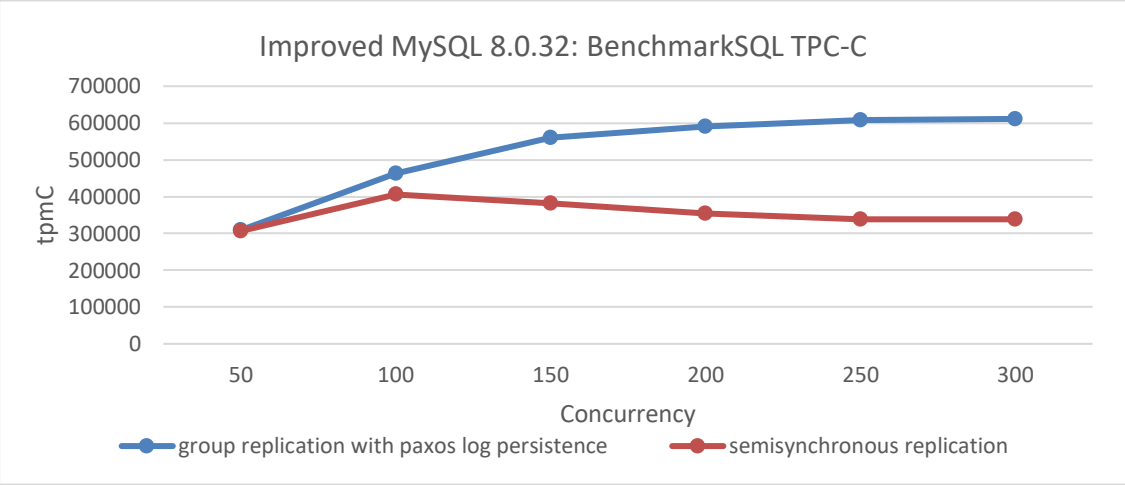


Figure 5-10. Performance comparison between Group Replication with Paxos log persistence and semisynchronous replication.

5.18 Group Replication

For continuous operation, a business requires high availability of its databases. To ensure continuous availability, a database must be fault-tolerant and robust to withstand failures. These qualities are achieved by Group Replication [24].

5.18.1 Why Implement Group Replication?

Asynchronous and semisynchronous replication cannot fully address high availability complexities. To achieve high availability, MySQL uses state machine replication based on the Paxos algorithm, known as Group Replication. This method theoretically solves the high availability issues that other replication methods cannot. Despite its potential, Group Replication faces numerous challenges, which is why it hasn't gained widespread popularity.

5.18.2 Why Was Mencius Initially Adopted?

Mencius is a multi-leader state machine replication protocol derived from Paxos [57]. It is designed to achieve high throughput under high client load and low latency under low client load, adapting to changing network and client environments. Mencius partitions the sequence of consensus protocol instances among servers, amortizing the leader load and increasing throughput when CPU-bound. It also fully utilizes available bandwidth and reduces latency by allowing clients to use a local server as the leader. Due to these advantages, Mencius aligns with the multi-primary mode design of Group Replication, where each MySQL node can perform write operations at any time, and was initially adopted by MySQL.

5.18.3 Why Introduce the Single-Leader Multi-Paxos Algorithm?

The single leader Multi-Paxos algorithm has the following characteristics [24]:

- It relies on a single leader to choose the request sequence.
- This simplicity results in high throughput and low latency for clients near the leader but higher latency for clients further away.
- The leader becomes a bottleneck, limiting throughput and creating an unbalanced communication pattern that underutilizes available network bandwidth.

MySQL introduced the single leader Multi-Paxos algorithm to improve performance and resilience in single-primary mode, especially when some secondary members are unreachable [24].

Tests in cross-datacenter scenarios show that using the single leader Multi-Paxos algorithm significantly improves Group Replication performance in single-primary mode. However, it has the drawback of high latency for other nodes needing to send data, as they must obtain the request sequence from the leader.

5.18.4 Does Group Replication Lose Data?

Group Replication implements state machine replication but does not inherently include durable state machine replication, meaning Paxos messages are not persisted. This design choice implies that while consensus can be reached within the cluster, data loss is possible in extreme cases. For instance, if all Group Replication nodes crash simultaneously and the MySQL primary cannot be restarted, the cluster formed by the remaining nodes may lose transaction data that had not been written to disk.

5.18.5 Will Group Replication Outperform Semisynchronous Replication?

The Mencius algorithm theoretically enables Group Replication to reach in-memory consensus, bypassing the need to parse transaction events at the Paxos layer. Additionally, batching mechanisms can merge several transactions into a single message for Paxos communication. Based on this, the throughput of Group Replication is expected to surpass that of semisynchronous replication.

5.18.6 Is a Single Thread Sufficient for Underlying Paxos Communication?

Paxos operates serially, but with pipelining and batching, it significantly improves throughput. Therefore, even with multi-threaded Paxos communication, Group Replication's overall system throughput is generally unaffected. However, large transactions can overwhelm a single thread, making multiple threads necessary to reduce wait times and speed up processing. In summary, while a single thread is sufficient for typical transactions, handling a large volume of large transactions may require multiple threads.

5.18.7 Paxos Single Leader vs. Group Replication Single-Primary Mode

Group Replication uses two Paxos variants: single leader Multi-Paxos and multiple leader Mencius. In single-primary mode, only one node handles transactional updates, while in multi-primary mode, all nodes can perform updates. The figure below illustrates the relationship between these algorithms and Group Replication's application modes:

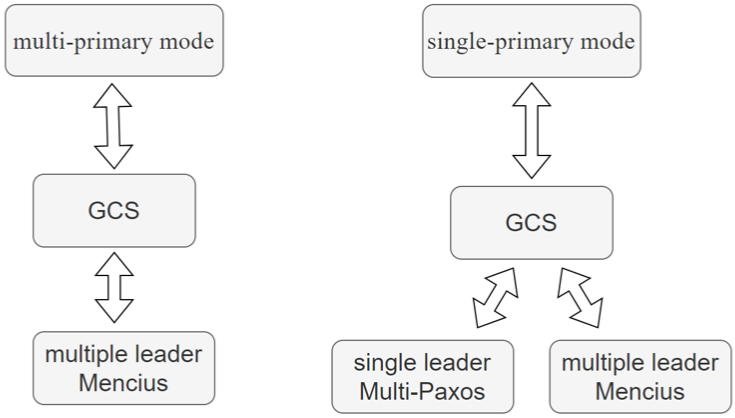


Figure 5-11. Relationship between Paxos variant algorithms and Group Replication modes.

In single-primary mode, both the Mencius algorithm and the single-leader Multi-Paxos algorithm can be used. However, in multi-primary mode, only the Mencius algorithm is applicable. This is because using the single-leader Multi-Paxos algorithm in multi-primary mode would severely degrade performance, as every non-leader node would need to request sequences from the leader node.

Group Replication in single-primary mode adopts the single-leader Multi-Paxos algorithm, assuming the leader node primarily sends messages while non-leader nodes have minimal roles. This allows efficient use of the single-leader Multi-Paxos algorithm, ensuring effective coordination and consensus among nodes.

5.18.8 Is Single Leader Multi-Paxos Universally Applicable in Single-Primary Mode?

In Group Replication's single-primary mode, consistent read operations require each MySQL node to send 'before' messages, while consistent write operations require 'after' messages. Using the single-leader Multi-Paxos algorithm in this context would significantly degrade performance.

The following figure illustrates SysBench's throughput for 100 concurrent read/write operations over time, with MySQL configured for strong consistency writes using the "after" mechanism.

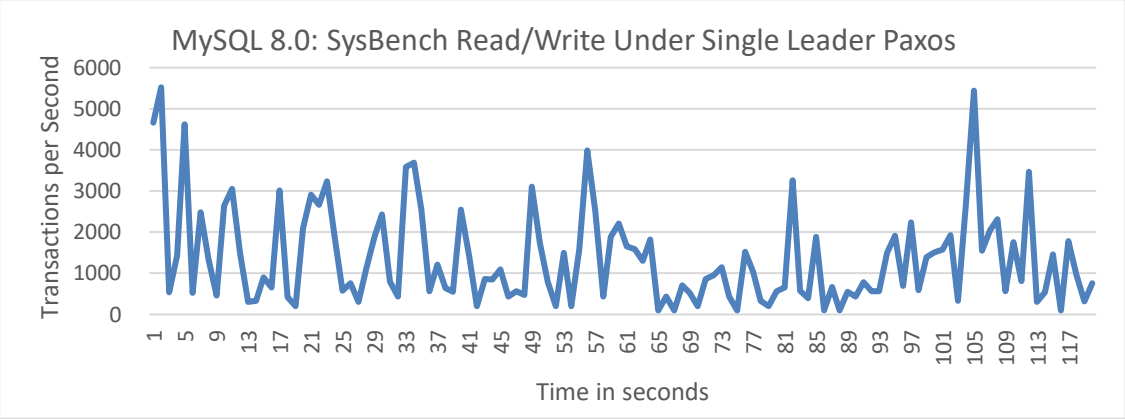


Figure 5-12. Performance of strong consistency writes using the "after" mechanism under single leader Paxos variant.

From the figure, it can be observed that after adopting the single leader Multi-Paxos, the throughput of MySQL's primary with strong consistency writes is significantly low. The following figure shows a partial screenshot of SysBench test results, indicating an average response time of 73ms.

```
[ 115s ] thds: 100 tps: 1459.97 qps: 29199.33 (r/w/o: 20439.53/5839.87/2919.93) lat (ms,99%): 292.60 err/s: 0.00 reconn/s: 0.00
[ 116s ] thds: 100 tps: 100.01 qps: 2000.13 (r/w/o: 1400.09/400.03/200.01) lat (ms,99%): 539.71 err/s: 0.00 reconn/s: 0.00
[ 117s ] thds: 100 tps: 1781.97 qps: 35639.42 (r/w/o: 24947.60/7127.88/3563.94) lat (ms,99%): 909.80 err/s: 0.00 reconn/s: 0.00
[ 118s ] thds: 100 tps: 962.02 qps: 19240.41 (r/w/o: 13468.28/3848.08/1924.04) lat (ms,99%): 450.77 err/s: 0.00 reconn/s: 0.00
[ 119s ] thds: 100 tps: 316.00 qps: 6320.09 (r/w/o: 4424.07/1264.02/632.01) lat (ms,99%): 450.77 err/s: 0.00 reconn/s: 0.00
[ 120s ] thds: 100 tps: 759.79 qps: 15085.74 (r/w/o: 10613.00/2953.17/1519.57) lat (ms,99%): 450.77 err/s: 0.00 reconn/s: 0.00
SQL statistics:
    queries performed:
        read:                            2294908
        write:                           655688
        other:                           327844
        total:                           3278440
    transactions:                        163922 (1365.81 per sec.)
    queries:                             3278440 (27316.11 per sec.)
    ignored errors:                      0      (0.00 per sec.)
    reconnects:                          0      (0.00 per sec.)

General statistics:
    total time:                          120.0173s
    total number of events:              163922

Latency (ms):
         min:                                    3.17
         avg:                                   73.21
         max:                                 1142.81
         99th percentile:                      549.52
         sum:                              12000787.04

Threads fairness:
    events (avg/stddev):           1639.2200/29.57
    execution time (avg/stddev):   120.0079/0.00
```

Figure 5-13. Partial screenshot of SysBench test results using the "after" mechanism under single leader Paxos variant.

Users reading official documentation might assume that the single leader Multi-Paxos algorithm can accelerate access. However, in reality, this algorithm is not suitable for consistent read/write operations.

5.19 MySQL Secondary Replay

5.19.1 Introduction to MySQL Secondary Replay

MySQL supports primary/secondary replication, typically using log shipping. In this setup, a primary instance generates logs for transactions, which are then shipped to one or more secondary instances. The secondaries replay these logs to mirror the primary's state, but may lag behind, affecting read-only queries' accuracy [28].

As server failures become common, replication for high availability is crucial. However, the need for high concurrency in transaction processing on multicore systems conflicts with traditional replication methods. This tension can lead to secondaries falling behind under heavy primary loads, increasing the risk of data loss or requiring throttling of the primary, thus impacting performance.

5.19.2 The Difference Between Replay and Transaction Execution

In MySQL's row-based logging, each operation is a log event. Row changes are recorded as insert, update, or delete events. These row events, along with transaction start and end events, define transaction boundaries. Insert events log new row images, update events log both before and after row images, and delete events log deleted row images.

During transactions, the primary writes updates to the log, which the secondary then fetches and replays. Secondaries can handle more read requests than the primary because replaying updates incurs only about half the workload of executing the original query. Additionally, read queries on the primary can conflict with update transactions, causing slowdowns, which supports dispatching read requests to secondaries [28].

5.19.3 The Role of Speeding Up MySQL Secondary Replay

The faster the MySQL secondary replay speed, the lower the likelihood of reading stale data from the MySQL secondary. For Group Replication, the speed of MySQL secondary replay is closely tied to high availability. If the MySQL secondary replay speed is fast enough, during a high availability switch, the new primary can immediately start serving new requests. Otherwise, it typically needs to wait until MySQL secondary replay is completed before it can serve new requests[①].

[①]Here, only 'BEFORE_ON_PRIMARY_FAILOVER' is considered.

5.19.4 The Architecture of MySQL Secondary Replay

The architecture of MySQL secondary replay is shown in the figure below:

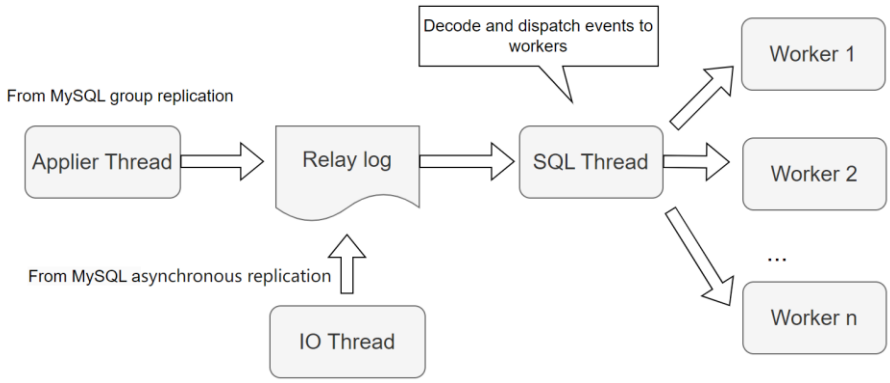


Figure 5-14. MySQL secondary replay architecture.

Asynchronous and semisynchronous replication both utilize an IO thread to read and store transaction events into the relay log. When Group Replication operates normally, it uses the applier thread to store applier events into the relay log. The SQL thread is crucial for MySQL secondary replay, responsible for parsing events and scheduling them. During MySQL secondary replay, multiple workers handle the replay process, each with its own worker queue where the SQL thread places pending events.

For MySQL secondary replay, the SQL thread acts not only as the scheduler but also reads and parses transaction events from the relay log files. When the relay log volume is small, the SQL thread can manage, but as the relay log grows, the SQL thread becomes the primary bottleneck. It struggles to keep up with the workload of parsing events and managing scheduling tasks. Moreover, the SQL thread encounters waiting situations under the following conditions:

1. Each worker queue has a fixed size with no adjustable parameters. If a transaction contains numerous events (e.g., large transactions), the worker queue quickly fills up, causing the SQL thread to wait.

2. If there aren't enough workers available, the SQL thread waits.

3. If the SQL thread finds a new transaction with a last committed value greater than the minimum logical timestamp (low-water-mark) of committed transactions (lwm value), it also needs to wait.

For example, the following code snippet illustrates how the SQL thread enters a waiting state when

the worker queue is full.

```
// possible WQ overfill
while (worker->running_status == Slave_worker::RUNNING && !thd->killed &&
      (ret = worker->jobs.en_queue(job_item)) ==
          Slave_jobs_queue::error_result) {
  thd->ENTER_COND(&worker->jobs_cond, &worker->jobs_lock,
                  &stage_replica_waiting_worker_queue, &old_stage);
  worker->jobs.overfill = true;
  worker->jobs.waited_overfill++;
  rli->mts_wq_overfill_cnt++;
  // wait if worker queue is full
  mysql_cond_wait(&worker->jobs_cond, &worker->jobs_lock);
  mysql_mutex_unlock(&worker->jobs_lock);
  thd->EXIT_COND(&old_stage);

  mysql_mutex_lock(&worker->jobs_lock);
}
```

Here is another example of code where the SQL thread needs to wait if it detects a newly started transaction with a last committed value greater than the currently calculated lwm (low-water mark) value:

```
bool Mts_submode_logical_clock::wait_for_last_committed_trx(
  Relay_log_info *rli, longlong last_committed_arg) {
 THD *thd = rli->info_thd;
 ...
 if ((!rli->info_thd->killed && !is_error) &&
    !clock_leq(last_committed_arg, get_lwm_timestamp(rli, true))) {
  PSI_stage_info old_stage;
  struct timespec ts[2];
  set_timespec_nsec(&ts[0], 0);

  assert(rli->gaq->get_length() >= 2);  // there's someone to wait

  thd->ENTER_COND(&rli->logical_clock_cond, &rli->mts_gaq_LOCK,
                  &stage_worker_waiting_for_commit_parent, &old_stage);
  do {
    // wait if lwm is less than last committed
    mysql_cond_wait(&rli->logical_clock_cond, &rli->mts_gaq_LOCK);
  } while ((!rli->info_thd->killed && !is_error) &&
           !clock_leq(last_committed_arg, estimate_lwm_timestamp()));
  min_waited_timestamp.store(SEQ_UNINIT);  // reset waiting flag
  mysql_mutex_unlock(&rli->mts_gaq_LOCK);
  thd->EXIT_COND(&old_stage);
```

```
    set_timespec_nsec(&ts[1], 0);
    rli->mts_total_wait_overlap += diff_timespec(&ts[1], &ts[0]);
  } else {
    min_waited_timestamp.store(SEQ_UNINIT);
    mysql_mutex_unlock(&rli->mts_gaq_LOCK);
  }

  return rli->info_thd->killed || is_error;
}
```

Therefore, the SQL thread is often busy and encounters many waiting situations, which is actually one of the main reasons for slow MySQL secondary replay.

5.20 The Integration of AI with Databases

Traditional database design relies on empirical methods and specifications, requiring human involvement (e.g., DBAs) for tuning and maintenance. AI techniques alleviate these limitations by exploring more design space than humans and replacing heuristics to solve complex problems. The existing AI techniques for optimizing databases can be categorized as follows [100].

5.20.1 Learning-based Database Configuration

1. **Knob Tuning**

   Databases have numerous knobs that need to be tuned by DBAs for different scenarios. This approach is not scalable for millions of cloud database instances. Recently, learning-based techniques have been used to automatically tune these knobs, exploring more combinations and recommending high-quality settings, often outperforming DBAs.

2. **Index/View Advisor**

   Indexes and views are essential for high performance, traditionally managed by DBAs. Given the vast number of column/table combinations, recommending and building appropriate indexes/views is costly. Recently, learning-based approaches have emerged to automate the recommendation and maintenance of indexes and views.

3. **SQL Rewriter**

   Many SQL programmers struggle to write high-quality queries, necessitating rewrites for performance improvement. For example, nested queries may be rewritten as joins for optimization. Existing methods use rule-based strategies, relying on predefined rules, which are

limited by the quality and scalability of the rules. Deep reinforcement learning can be used to select and apply rules effectively.

5.20.2 Learning-based Database Optimization

1. **Cardinality/Cost Estimation**

   Traditional database optimizers struggle to capture correlations between different columns/tables, leading to suboptimal cost and cardinality estimations. Recently, deep learning techniques have been proposed to improve these estimations by using neural networks to better capture correlations.

2. **Join Order Selection**

   SQL queries can have millions or even billions of possible execution plans. Efficiently finding a good plan is crucial, but traditional optimizers struggle with large tables due to the high cost of exploring vast plan spaces. Deep reinforcement learning methods have been developed to automatically select efficient plans.

3. **End-to-End Optimizer**

   A comprehensive optimizer must consider cost/cardinality estimation, join order, indexes, and views. Learning-based optimizers use deep neural networks to optimize SQL queries holistically, improving overall query performance.

5.20.3 Learning-based Database Design

Traditional databases are designed by architects based on experience, which limits the exploration of design spaces. Recently, learning-based self-design techniques have emerged [100]:

1. **Learned indexes**: These reduce index size and improve performance.
2. **Learned data structure design**: Different data structures suit different environments (e.g., hardware, read/write applications). Data structure alchemy creates an inference engine to recommend and design suitable structures.
3. **Learning-based Transaction Management**: Traditional techniques focus on protocols like OCC, PCC, MVCC, 2PC. New studies use AI to predict and schedule transactions, balancing conflict rates and concurrency by learning from data patterns and predicting future workload trends.

5.20.4 Learning-based Database Monitoring

Database monitoring captures runtime metrics such as read/write latency and CPU/memory usage, alerting administrators to anomalies like performance slowdowns and attacks. Traditional methods rely on administrators to monitor activities and report problems, which is inefficient. Machine learning techniques optimize this process by determining when and how to monitor specific metrics.

5.20.5 Learning-based Database Security

Traditional database security techniques, such as data masking and auditing, rely on user-defined rules and cannot automatically detect unknown vulnerabilities. AI-based algorithms address this by:

1. **Sensitive Data Discovery**: Automatically identifying sensitive data using machine learning.
2. **Anomaly Detection**: Monitoring database activities to detect vulnerabilities.
3. **Access Control**: Automatically estimating data access actions to prevent data leaks.
4. **SQL Injection Prevention**: Using deep learning to analyze user behavior and identify SQL injection attacks.

5.20.6 Performance Prediction

Query performance prediction is crucial for meeting service level agreements (SLAs), especially for concurrent queries. Traditional methods focus only on logical I/O metrics, neglecting many resource-related features, leading to inaccurate results.

Marcus et al. used deep learning to predict query latency under concurrency, accounting for interactions between child/parent operators and parallel plans. However, their pipeline structure caused information loss and failed to capture operator-to-operator relations like data sharing and conflict features.

To address this, Zhou et al. proposed a method using graph embedding. They modeled concurrent queries with a graph where vertices represent operators and edges capture operator correlations (e.g., data passing, access conflicts, resource competition). A graph convolution network was used to embed the workload graph, extract performance-related features, and predict performance based on these features [100].

5.20.7 AI Challenges

AI models require large-scale, high-quality, diversified training data for optimal performance, but obtaining such data in AI4DB is challenging due to security concerns and reliance on DBAs. For instance,

in database knob tuning, training samples depend on DBA experience, making it difficult to gather sufficient samples. Effective models also need data covering various scenarios, hardware environments, and workloads, necessitating methods that perform well with small training datasets.

Adaptability is a major challenge, including adapting to dynamic data updates, different datasets, new hardware environments, and other database systems [100]. Key questions include:

- How to adapt a trained model (e.g., optimizer, cost estimation) to other datasets?
- How to adapt a model to different hardware environments?
- How to adapt a model across different databases?
- How to support dynamic data updates?

Model convergence is crucial. If a model doesn't converge, alternative solutions are needed to avoid delays and inaccuracies, such as in knob tuning where non-converged models can't provide reliable online suggestions.

Traditional OLAP focuses on relational data analytics, but big data introduces new types like graph, time-series, and spatial data. New techniques are required to analyze these multi-model data types and integrate AI with DB techniques for enhanced analytics, such as image analysis.

Transaction modeling and scheduling are critical for OLTP systems due to potential conflicts between transactions. Learning techniques can optimize OLTP queries, like consistent snapshots. However, efficient models are needed to instantly model and schedule transactions across multiple cores and machines.

5.20.8 AI Summary

Integrating AI into MySQL offers many impactful opportunities and is a primary focus for our future development.

5.21 How MySQL Internals Work in a Pipeline Fashion?

Database engines like MySQL excel at medium concurrency, interleaving the execution of many transactions, most of which are idle at any given moment [6]. However, as the number of cores per chip increases with Moore's law, MySQL must exploit high parallelism to benefit from new hardware. Despite high concurrency in workloads, internal bottlenecks often prevent MySQL from achieving the needed parallelism.

MySQL's internal operations follow a pipelining approach, where each component functions methodically. To ensure correctness, MySQL uses latches and locks to prevent interference between concurrent operations. For crash-safe recovery, MySQL employs mechanisms like redo log, undo log, idempotence, and double write.

To support concurrent read and write operations, MySQL implements transaction isolation levels and MVCC (Multi-Version Concurrency Control). These mechanisms enable efficient handling of concurrent operations.

To mitigate single-server failure risks, MySQL uses clustering and high availability solutions. Some operations, such as redo log flushing and MVCC ReadView replication, are serialized. Fast execution of these processes supports high throughput, but slow serialization can hinder scalability.

MySQL manages various processes, including SQL parsing, execution plan generation, redo log writing, transaction subsystem operations, lock subsystem operations, binlog group commit, and network interactions. Effective throughput is achieved when these processes are handled by different threads without overlap. However, thread aggregation in a serialized process can lead to conflicts and performance issues.

Optimizing MySQL's efficiency involves faster response times, reduced CPU consumption, and enhanced scalability.

5.22 Why MySQL Needs to Support High Concurrency?

The inability to achieve high throughput is primarily due to the constraints of guaranteeing ACID. More cores lead to more concurrent transactions accessing the same contended data, requiring serialized access to ensure isolation. The maximum throughput of a database system depends on factors like hardware and software implementation. While better hardware and software can improve performance, throughput is fundamentally limited by contended operations. Contended operations vary with transaction isolation levels, multiversioning capabilities, and operation commutativity. Regardless, some operations will always require serialization, limiting throughput. Adding more processors only increases throughput if additional transactions do not conflict with existing ones [36].

In high-conflict scenarios, linear scalability of throughput is unachievable. However, scalability improves in low conflict scenarios, as evidenced by TPC-C tests with numerous warehouses. In practical MySQL operations, short bursts of high concurrency are common. Supporting high concurrency helps maintain MySQL stability during runtime.

Subsequently, the significance of supporting high concurrency specifically for TPC-C testing is

analyzed. The figure below illustrates the relationship between throughput and concurrency for MySQL version 5.7.39 under a 1ms thinking time scenario. In this context, 1ms thinking time means the user waits 1ms before sending the next request after receiving the response from the previous one.

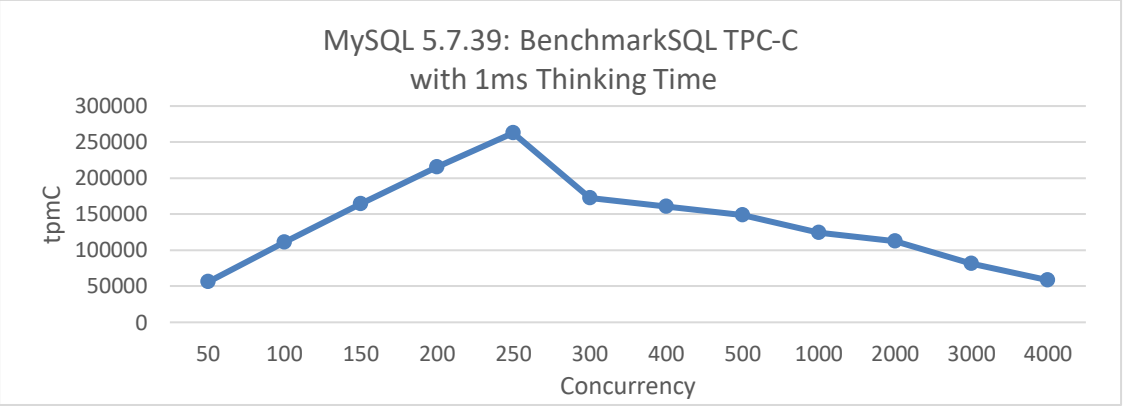


Figure 5-15. MySQL 5.7.39 pool scalability with 1ms thinking time.

From the figure, it can be observed that under a 1ms thinking time scenario, the throughput of MySQL 5.7.39 increases linearly at low concurrency levels. However, once it reaches 250 concurrency, the throughput sharply declines.

The following figure shows the relationship between throughput and concurrency for improved MySQL 8.0.27 under the same 1ms thinking time scenario:

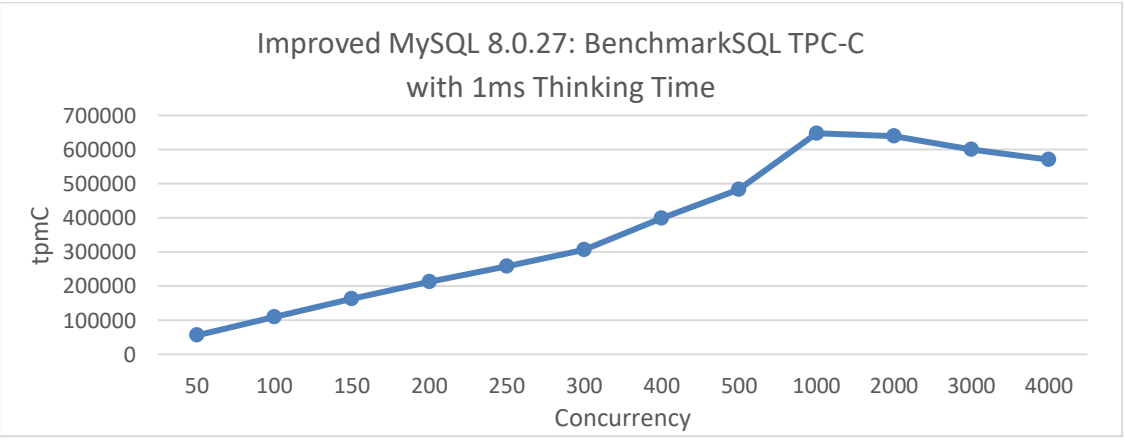


Figure 5-16. Excellent scalability of improved MySQL 8.0.27 with 1ms thinking time.

From the figure, it can be seen that the peak throughput is reached at 1000 concurrency, and this peak significantly exceeds the peak of MySQL 5.7.39 version.

High concurrency support in MySQL is not only crucial for scenarios with thinking time but also essential for deployments across multiple cities (geographically distributed deployments).

The following figure illustrates the relationship between throughput and concurrency for improved MySQL 8.0.27 under different network latency scenarios.

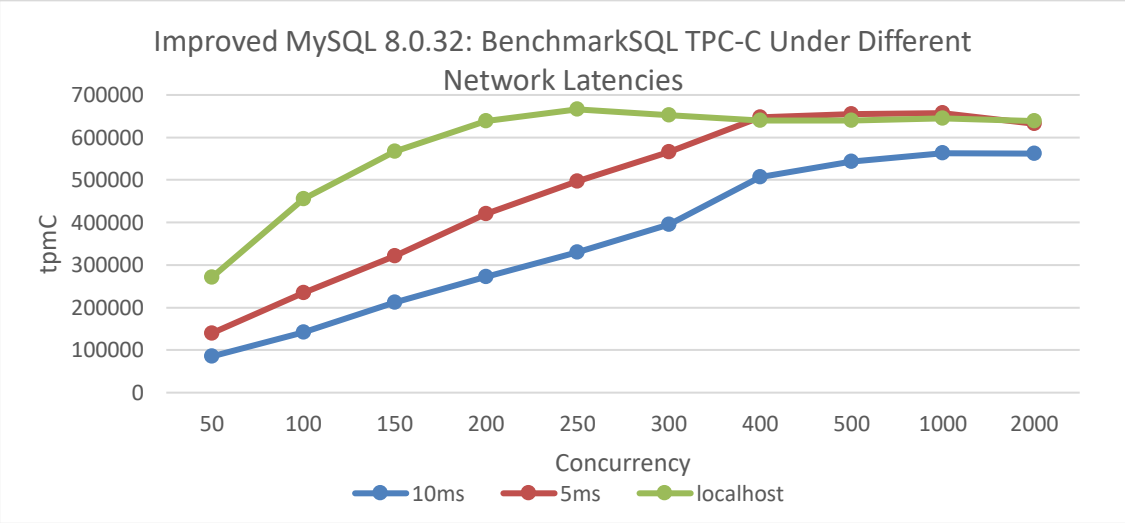


Figure 5-17. Excellent scalability of enhanced MySQL 8.0.27 under different network latencies.

In scenarios with network latency in the tens of microseconds range for localhost access, peak throughput is achieved at 250 concurrency. With a network latency of 5ms, peak throughput is reached at 500 concurrency, and with a 10ms latency, it requires 1000 concurrency.

Therefore, improving MySQL's scalability is highly meaningful. In low-conflict scenarios, enhancing scalability significantly improves throughput and has profound implications across various application contexts. However, in high-conflict situations, transaction throttling strategies are necessary to mitigate scalability issues.

5.23 Scalability of MySQL Clusters

First, let's examine the scalability of asynchronous replication. The following figure illustrates the relationship between throughput and concurrency for both a standalone MySQL instance and asynchronous replication.

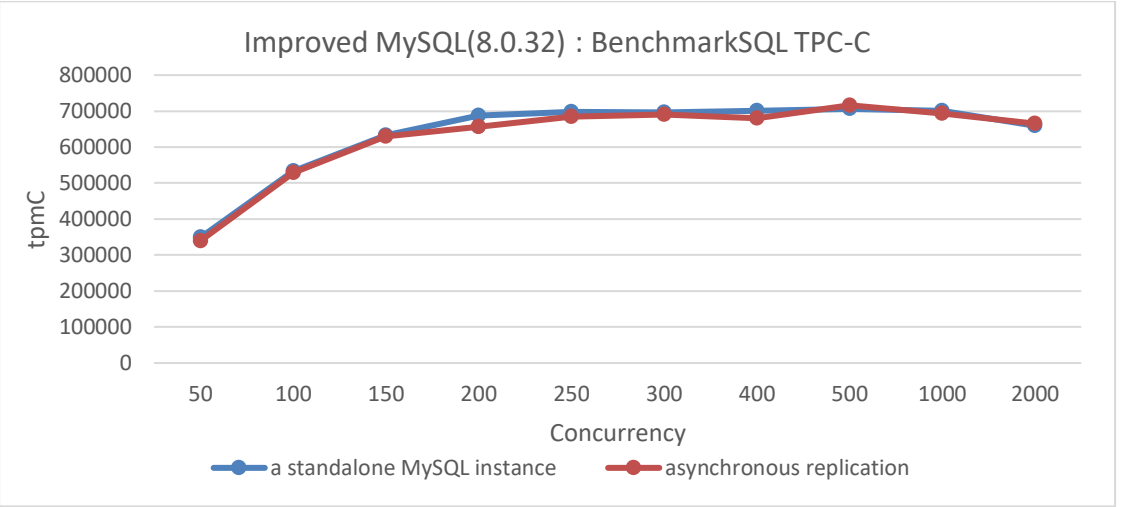


Figure 5-18. Excellent scalability of MySQL asynchronous replication.

From the figure, it can be seen that with asynchronous replication, the MySQL secondary does not significantly impact the throughput of the MySQL primary.

Let's now examine the scalability of semisynchronous replication. For example, the following figure illustrates the relationship between throughput and concurrency for semisynchronous replication.

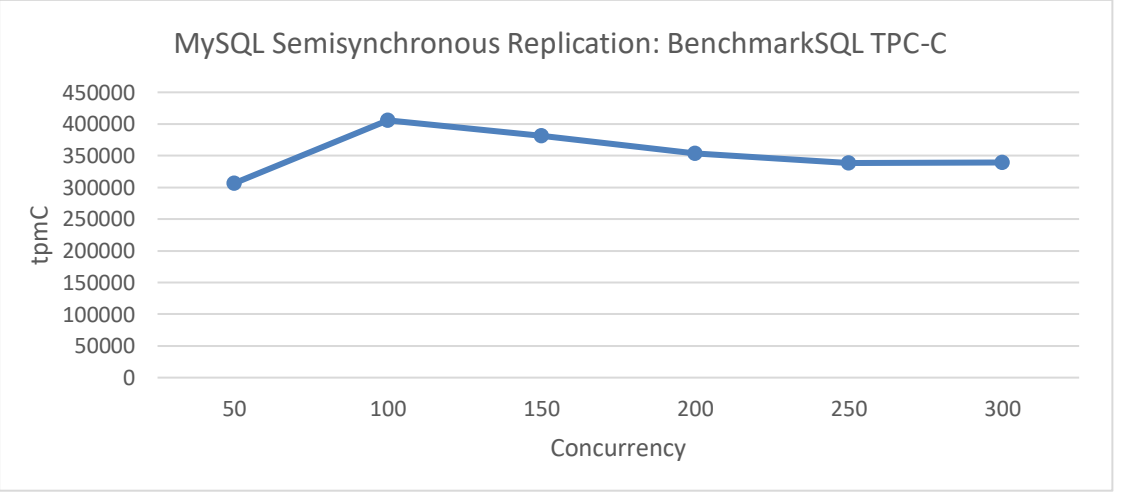


Figure 5-19. Poor scalability of MySQL semisynchronous replication.

From the figure, it is clear that the throughput of semisynchronous replication struggles to increase further. After reaching 100 concurrency, the throughput gradually declines, suggesting a barrier that limits additional scaling. For more details, please refer to section 5.17.3.

Finally, let's examine the scalability of Group Replication. Before delving into that, it's important to understand the performance drawbacks of state machine replication [75], as explained below:

However, the communication and synchronization cost of agreement entails that state-machine replication adds a considerable overhead to the system's performance, i.e., typical state-machine replication request rates are lower than the request rates of non-replicated systems.

Subsequently, the maximum scalability of Group Replication is tested, including the implementation of transaction throttling mechanisms to ensure that only a specified number of threads can access the transaction system.

The following figure shows the relationship between Group Replication throughput and concurrency:

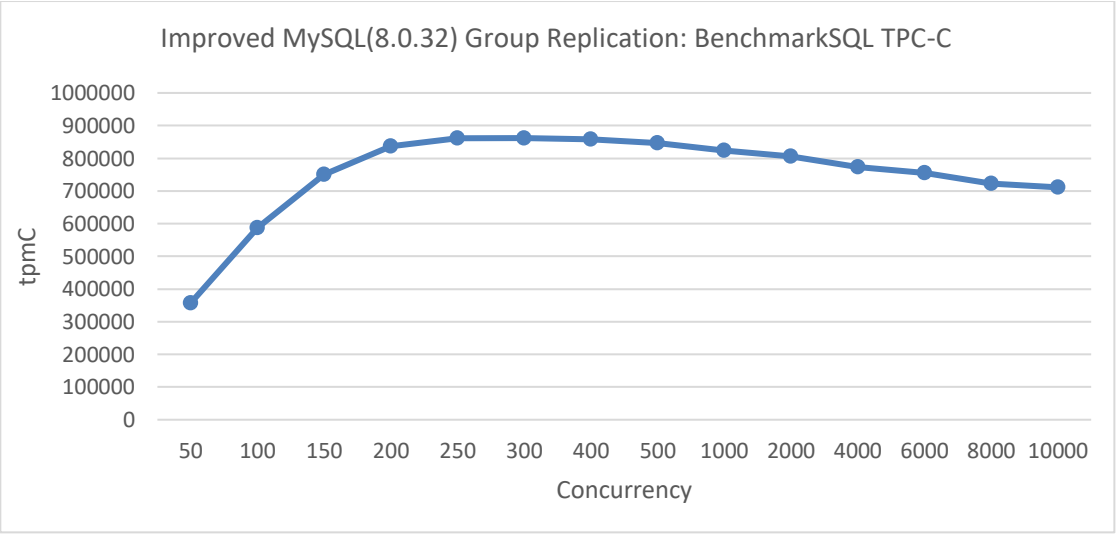


Figure 5-20. Excellent scalability of improved Group Replication with transaction throttling mechanisms.

When throughput peaks at 250 concurrency and then slowly declines, it clearly demonstrates that Group Replication exhibits excellent scalability under the transaction throttling mechanism. Compared to semisynchronous replication, Group Replication's design is significantly superior.

It is important to note that transaction throttling is implemented to manage the broad range of concurrency. An excessive number of transactions entering the InnoDB storage engine can degrade performance and interfere with the evaluation of Group Replication's scalability.

5.24 MySQL Problem Diagnosis

Bugs are an inevitable part of programming. Logical reasoning is crucial for identifying and fixing these bugs. It involves systematically analyzing the code, diagnosing the causes of issues, and devising a plan to address them based on logical deductions and inferences. This approach helps programmers debug and troubleshoot in a structured and effective manner [101].

In addition to the commonly used GDB debugging, another approach is to use logging to assist in troubleshooting issues. There will be case studies introducing this method in the subsequent chapters.

Besides the methods mentioned above, MySQL itself also provides several debugging solutions, which are described as follows:

1. MySQL is compiled with debug mode enabled, and trace output can be added in the command terminal (refer to the example below).

```
set global debug= 'd:T:t:i:o,/tmp/mysql.trace';
```

Executing specific SQL statements will output MySQL's internal function call relationships to the trace file, as shown in the example below:

```
T@8: 21:37:06.362636 | >dispatch_command
T@8: 21:37:06.362639 | | info: command: 3
...
T@8: 21:37:06.362667 | | >alloc_query
T@8: 21:37:06.362672 | | | thd_query: thd->thread_id():8 thd:0x7f8d6be18a00 query:select count(*) from sbtest1 limit 10000
T@8: 21:37:06.362675 | | <alloc_query
T@8: 21:37:06.362678 | | query: select count(*) from sbtest1 limit 10000
...
T@8: 21:37:06.362690 | | >dispatch_sql_command
T@8: 21:37:06.362692 | | | dispatch_sql_command: query: 'select count(*) from sbtest1 limit 10000'
T@8: 21:37:06.362696 | | | >THD::reset_for_next_command
T@8: 21:37:06.362771 | | | | <Security_context::checkout_access_maps
...
T@8: 21:37:06.362774 | | | <THD::reset_for_next_command
T@8: 21:37:06.362778 | | | >lex_start
T@8: 21:37:06.362785 | | | | >LEX::start
T@8: 21:37:06.362788 | | | | | >LEX::new_top_level_query
T@8: 21:37:06.362792 | | | | | | >LEX::new_query
T@8: 21:37:06.362799 | | | | | | | outer_field: creating ctx 0x7f8d6be55358
T@8: 21:37:06.362804 | | | | | | | outer_field: ctx 0x7f8d6be55358 <-> 5L# 1
T@8: 21:37:06.362807 | | | | | | <LEX::new_query
T@8: 21:37:06.362810 | | | | | <LEX::new_top_level_query
T@8: 21:37:06.362814 | | | | <LEX::start
T@8: 21:37:06.362816 | | | <lex_start
...
T@8: 21:37:06.362843 | | | >parse_sql
T@8: 21:37:06.362940 | | | | >Query_block::add_table_to_list
T@8: 21:37:06.362952 | | | | <Query_block::add_table_to_list
T@8: 21:37:06.362956 | | | | >Query_block::add_joined_table
T@8: 21:37:06.362959 | | | | | >MEM_ROOT::AllocSlow
T@8: 21:37:06.362962 | | | | | | enter: root: 0x7f8d6be1b908
T@8: 21:37:06.362965 | | | | | | >MEM_ROOT::AllocBlock
T@8: 21:37:06.362969 | | | | | | | <MEM_ROOT::AllocBlock
T@8: 21:37:06.362972 | | | | | | <MEM_ROOT::AllocSlow
T@8: 21:37:06.362975 | | | | <Query_block::add_joined_table
T@8: 21:37:06.362983 | | | | >Query_tables_list::set_stmt_unsafe
T@8: 21:37:06.362985 | | | | <Query_tables_list::set_stmt_unsafe
T@8: 21:37:06.362992 | | | <parse_sql
```

Trace information is very helpful for understanding how MySQL operates, but this method is only suitable for scenarios with very low traffic volume.

2. The Performance Schema collects statistics on various types of information, including memory usage information, locks, and statistics on condition variables, among others.

```
mysql> update performance_schema.setup_instruments set ENABLED='YES',TIMED='YES' where name like '%LOCK_done%';
Query OK, 0 rows affected (0.00 sec)
Rows matched: 1  Changed: 0  Warnings: 0

mysql> update performance_schema.setup_instruments set ENABLED='YES',TIMED='YES' where name like '%COND_done%';
Query OK, 0 rows affected (0.01 sec)
Rows matched: 1  Changed: 0  Warnings: 0

mysql> select * from  performance_schema.events_waits_summary_global_by_event_name order by SUM_TIMER_WAIT desc limit 20;
+---------------------------------------------------------+-----------+----------------+----------------+----------------+----------------+
| EVENT_NAME                                              | COUNT_STAR | SUM_TIMER_WAIT | MIN_TIMER_WAIT | AVG_TIMER_WAIT | MAX_TIMER_WAIT |
+---------------------------------------------------------+-----------+----------------+----------------+----------------+----------------+
| wait/io/table/sql/handler                               |  143806269 | 22665092486210082 |        1946160 |      157608432 |    961564912812 |
| idle                                                    |   12945752 |  4941341781897000 |        1004000 |      381696000 | 444803044333000 |
| wait/io/file/innodb/innodb_data_file                    |    1472208 |  3724543889168328 |              0 |     2529903060 |     22219365006 |
| wait/io/file/innodb/innodb_log_file                     |    5655986 |    41211995029110 |         383508 |        7286175 |     90061766118 |
| wait/io/file/sql/binlog                                 |     223164 |    21364097639004 |              0 |       95732469 |     66196176804 |
| wait/lock/table/sql/handler                             |    7721153 |     7220185422390 |          57240 |         934920 |       610594344 |
| wait/synch/mutex/sql/MYSQL_BIN_LOG::LOCK_done           |     305442 |     6099451446 0 |          21942 |        1996722 |      9598017510 |
| wait/io/file/innodb/innodb_dblwr_file                   |        651 |     2647167289 92 |        2951676 |      406630575 |      9124839234 |
| wait/io/file/sql/ERRMSG                                 |          5 |        1791425970 |        3850344 |      358285194 |      1574331822 |
| wait/io/file/innodb/innodb_temp_file                    |         60 |        1237089960 |        3273174 |       20617848 |        45052650 |
| wait/io/file/sql/binlog_index                           |         60 |         323929746 |              0 |        5398686 |        77797746 |
| wait/io/file/sql/relaylog                               |         19 |         318149460 |              0 |       16744608 |       105832944 |
| wait/io/file/sql/casetest                               |          5 |         100218654 |              0 |       20043540 |        71127378 |
| wait/io/file/sql/relaylog_index                         |         26 |          56883204 |              0 |        2187522 |         9951174 |
| wait/io/file/sql/pid                                    |          3 |          32825232 |        5633370 |       10941426 |        20728512 |
| wait/io/file/mysys/charset                              |          3 |          28280376 |        2123604 |        9426474 |        14808942 |
| wait/io/file/mysys/cnf                                  |          5 |          15435720 |         150732 |        3087144 |        10823130 |
| wait/io/file/sql/misc                                   |          1 |           3463020 |              0 |        3463020 |         3463020 |
| wait/synch/mutex/sql/MYSQL_BIN_LOG::LOCK_commit_queue   |          0 |                 0 |              0 |              0 |               0 |
| wait/synch/mutex/sql/MYSQL_BIN_LOG::LOCK_xids           |          0 |                 0 |              0 |              0 |               0 |
+---------------------------------------------------------+-----------+----------------+----------------+----------------+----------------+
20 rows in set (0.03 sec)
```

The figure above utilizes Performance Schema to perform statistical analysis on mutexes, making it convenient to identify which mutexes incur higher costs.

5.25 The Significant Differences Between BenchmarkSQL and SysBench

Using the case of optimizing lock-sys as an example, this section evaluates the significant differences between the SysBench tool and BenchmarkSQL in MySQL performance testing.

First, use SysBench's standard read/write tests to evaluate the optimization of lock-sys.

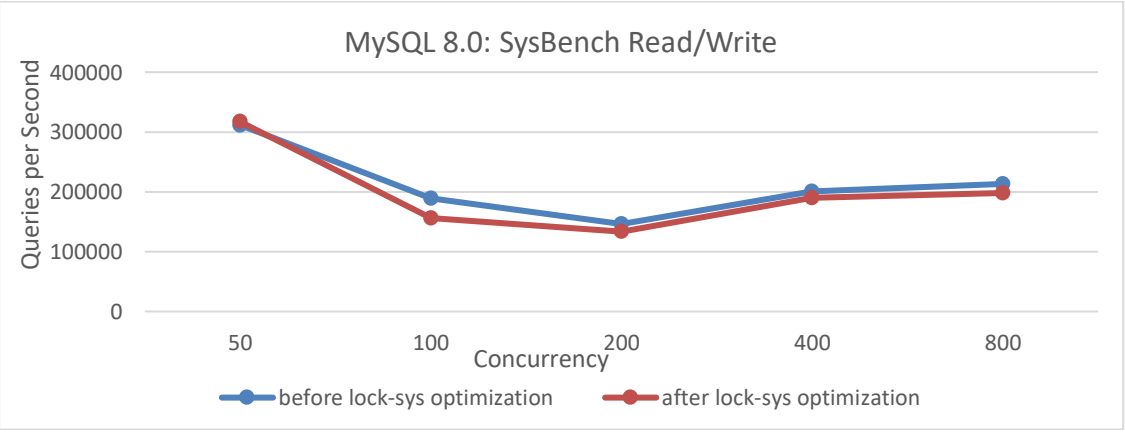


Figure 5-21. Comparison of SysBench read-write tests before and after lock-sys optimization.

From the figure, it can be observed that after optimization, the overall performance of the SysBench tests has actually decreased.

Next, using BenchmarkSQL to test this optimization, the results are shown in the following figure.

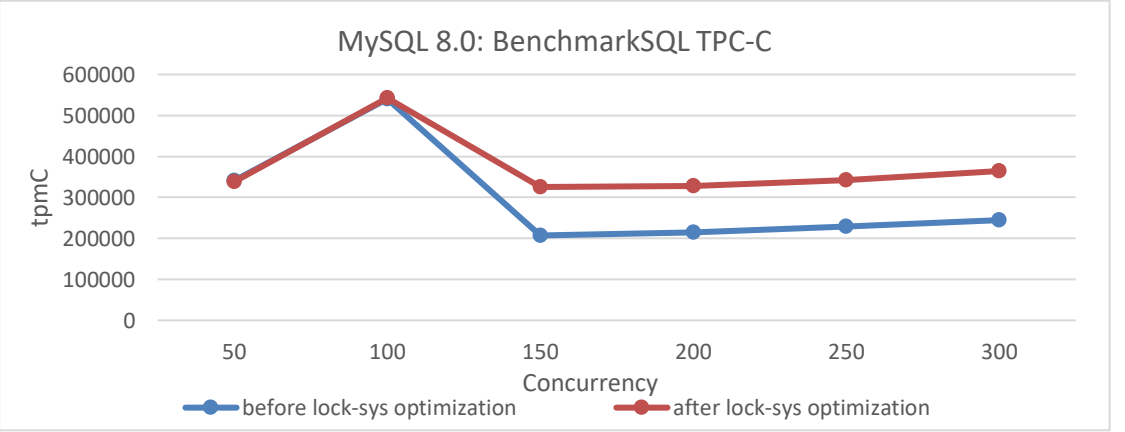


Figure 5-22. Comparison of BenchmarkSQL tests before and after lock-sys optimization.

From the figure, it can be seen that the results of BenchmarkSQL's TPC-C test indicate that the lock-sys optimization is effective. Why does such a significant difference occur? Let's analyze the differences in characteristics between these testing tools to understand why their tests differ.

SysBench RW testing is characterized by its speed and simplicity with SQL queries. Under the same concurrency conditions, SysBench typically handles fewer concurrent transactions compared to BenchmarkSQL. Therefore, in the face of latch queue bottlenecks like lock-sys, high concurrency in SysBench may equate to low concurrency in BenchmarkSQL. Consequently, lock-sys optimizations may not have a significant impact in scenarios where BenchmarkSQL operates at lower concurrency levels.

BenchmarkSQL, a widely used TPC-C testing tool, distributes user threads more evenly across various modules, reducing susceptibility to aggregation effects. In high-concurrency situations, optimizing lock-sys can significantly reduce latch conflicts and minimize impact on other queues, thereby improving throughput. BenchmarkSQL's TPC-C testing is better suited for uncovering deeper concurrency issues in MySQL compared to SysBench.

This analysis uses deductive reasoning to explore the differences between SysBench and BenchmarkSQL. It demonstrates that poor performance in SysBench tests does not necessarily indicate poor performance in production environments, and vice versa. This discrepancy arises

because SysBench test environments often differ significantly from real-world production environments. Consequently, SysBench test results should be used for scenario-specific performance comparisons rather than as comprehensive indicators of production capabilities.

It is worth noting that the main basis for performance testing and comparison in this book, mainly based on TPC-C, is as follows [91]:

TPC benchmark C also known as TPC-C which is the leading online transaction processing (OLTP) benchmark has been used to perform the comparison.

In this book, BenchmarkSQL is predominantly used for TPC-C testing. This choice is based not only on BenchmarkSQL's representative TPC-C testing capabilities but also on its higher alignment with real-world online environments.

Chapter 6: How to Scientifically Test MySQL Performance?

Performance benchmarking is commonly used to compare different systems or algorithms in both scientific literature and industrial publications. Although performance measurements might seem objective, various factors can influence benchmark results, intentionally or accidentally, favoring one system over another. There is a fundamental conflict of interest in performance benchmarking, especially when evaluations are made against previous versions or competitors' systems. Some results, often referred to as "benchmarketing", misrepresent performance data. Fair performance benchmarking is challenging, and it's easy to misrepresent data, either by accident or on purpose [16].

This book explores common pitfalls in MySQL performance benchmarking found in scientific works and describes how to avoid them to ensure fair performance comparisons.

6.1 Common Pitfalls

This section discusses common pitfalls encountered when performing performance comparisons between different MySQL versions.

6.1.1 Non-Reproducibility

Reproducibility is a cornerstone of scientific research, allowing others to verify results and identify errors. Without reproducibility, claims and numbers in experiments cannot be validated. Conducting reproducible experiments is relatively straightforward and cost-effective compared to larger scientific studies. However, many database research papers lack reproducibility due to closed-source code, undisclosed data, or proprietary systems.

To enable reproducibility, all configuration parameters must be provided, including minor details like the operating system, server installation, version, setup, and configuration flags. Additionally, source code for any algorithms or implementations should be made available [16].

Most tests in this book will include source code, MySQL configuration files, testing tool details, hardware specifications, and OS versions to ensure reproducibility and verification of the test results[1].

[1] All materials are expected to be released by August 1, 2025.

6.1.2 Failure To Optimize

Benchmarks are commonly used to compare systems or assess the effectiveness of new algorithms. Typically, experiments involve comparing a newly proposed system against an existing one to demonstrate superior performance.

This setup, however, can disincentivize proper optimization of the existing system, as poor performance of the current system can make the new system appear better. This issue is significant for systems that depend on proper configuration, as an improperly configured system can perform much worse than a well-configured one.

Optimizing a DBMS for a specific workload is complex and often requires expertise. Even minor optimizations can improve fairness in performance comparisons. Following optimization guidelines for benchmarks or involving representatives from the compared systems can also help achieve more accurate results [16].

For performance testing in this book, the following strategies are applied:

1. **Conduct Pre- and Post-Optimization Comparisons:** Where feasible, perform performance comparisons before and after optimization, aiming for minimal variation.
2. **Match Configuration Parameters:** Align configuration parameters as closely as possible with the production environment.
3. **Conduct Performance Comparisons on Identical Hardware**: Perform tests on the same x86 machine in a NUMA environment with identical configurations. Reinitialize MySQL data directories and clean SSDs (via TRIM) to avoid interference. Test across a range of concurrency levels to assess throughput and determine whether optimizations improve performance or scalability.
4. **Repeat Testing with NUMA Node Binding**: After binding to NUMA node 0, repeat the tests to compare performance in an SMP-like environment.
5. **Test on x86 Machines with NUMA Disabled**: Conduct comparative performance testing on x86 machines with identical hardware but with NUMA disabled (in BIOS).
6. **Evaluate Performance on ARM Machines**: Test comparative performance on ARM machines in a NUMA environment with similar MySQL configurations.
7. **Verify Consistency with Different Tools**: Use various testing tools to compare results and

ensure consistency. For example, employ BenchmarkSQL and modified versions of tpcc-mysql for TPC-C testing.

8. **Assess Performance Under Varying Network Latency**: Examine performance effects under different network latency conditions.

9. **Test Performance with Different "Thinking Time" Scenarios**: Evaluate how performance varies with different "thinking time" scenarios to gauge consistency.

10. **Perform Closed-Loop Testing**: Ensure no interference during testing by repeating the initial tests and comparing results with the first round. Small differences in test results indicate that the environment is relatively stable.

11. **Verify Bottleneck Interference**: Confirm whether interference from other bottlenecks under high concurrency has distorted the performance comparison.

12. **Analyze Theoretical Basis and Anomalies**: Evaluate whether the performance optimization has a theoretical basis and if any anomalies can be explained. Analyze the type of optimization, its general applicability, and which environments benefit most. Investigate anomalies to determine their causes.

6.1.3 Overly-specific Tuning

These issues can be mitigated by conducting a range of experiments beyond standardized benchmarks. While standardized benchmarks provide a useful baseline, some systems may be heavily optimized for them, reducing their effectiveness for comparison. Thus, additional queries should be tested and measured [16].

To address these issues, MySQL configuration should meet the following criteria:

1. **Minimize Impact of Configuration Parameters**: Ensure parameters, like buffer pool size, do not hinder other optimizations.

2. **Use Default Configurations**: Apply default settings for uncertain parameters, such as spin delay.

3. **Match Production Configurations**: Align test settings with production configurations, e.g., sync_binlog=1 and innodb_flush_log_at_trx_commit=1.

To overcome the limitations of single-type testing, employ a variety of test scenarios. For TPC-C,

include tests with varying conflict severity, thinking time, and network latency. For SysBench, use tests with Pareto distributions, read/write, and write-only operations.

These strategies ensure a comprehensive evaluation of MySQL performance optimizations under various conditions, resulting in more robust and reliable results.

6.1.4 Cold vs. Hot Runs

It's crucial to differentiate between 'hot' and 'cold' runs. Cold runs, which occur when relevant data is being loaded from persistent storage and queries are being parsed, are typically slower than hot runs where data is already cached. Performance measurements for cold and hot runs should be recorded and reported separately due to these differences [16].

For TPC-C testing, all test operations follow a standardized process to ensure consistency, and tests are conducted in an environment free from human interference. Additionally, to mitigate performance disturbances caused by different environments, closed-loop testing is employed to maximize the reliability of the test results as much as possible.

6.1.5 Cold vs. Warm Runs

Measuring cold runs requires caution to avoid accidentally recording warm runs. Proper measurement involves more than just restarting the database server and running a query, as operating systems may cache data in memory. The correct method is to stop the database server, clear all OS caches, restart the server, and then run the queries

An evaluation test was conducted to assess these impacts. For instance, the following figure compares TPC-C throughput and concurrency before and after clearing the cache.

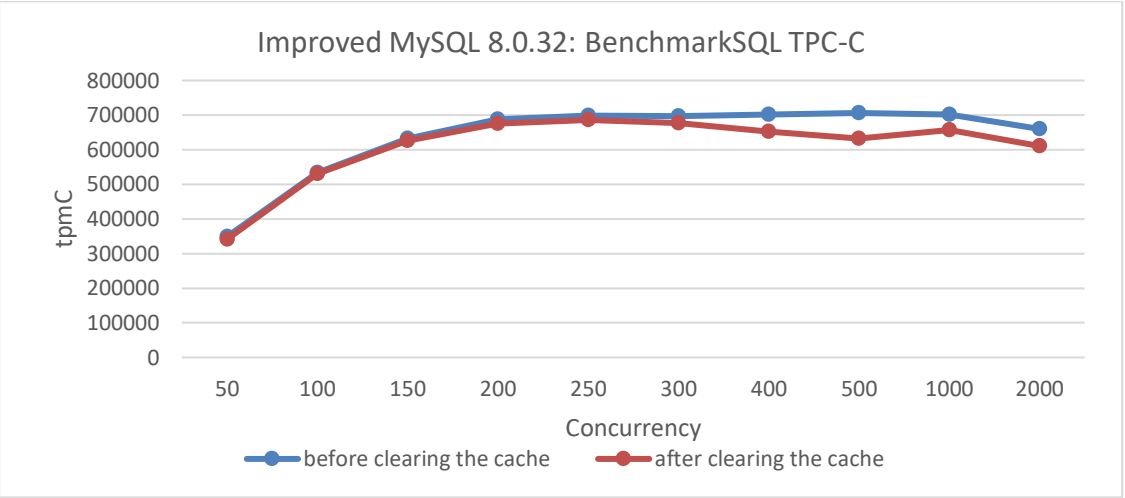


Figure 6-1. Comparison of BenchmarkSQL tests before and after clearing the cache.

From the figure, it can be observed that the throughput after clearing the cache is indeed lower compared to the previous throughput.

To mitigate such issues, closed-loop testing was employed, which effectively detects anomalies in the environment.

6.1.6 Human Error

When evaluating the performance of a newly designed algorithm, it is crucial to verify the correctness of the implementation to avoid overlooking bugs that may produce incorrect results. A bug might lead to performance metrics being based on faulty outcomes, potentially making the algorithm appear more efficient due to, for example, less data being processed. This issue is compounded if the benchmark is not reproducible, as incorrect implementations may be mistakenly accepted as valid.

Additionally, a program might work correctly for specific datasets but fail in general cases due to issues like inadequate overflow handling or hardcoding dataset properties. Always verify that the program produces correct results by comparing outputs against reference answers from benchmark specifications and checking results with varying data [16].

For performance testing, it's essential to consider whether optimization results are influenced by other factors. For example, in MySQL 8.0 with the CATS lock scheduling algorithm, the performance evaluation was affected by extensive deadlock logs output by MySQL.

6.2 Comprehensive Testing

Performance testing is an extremely complex process. Only through comprehensive testing and analysis can errors during the testing process be minimized, allowing for a scientific assessment of the value of an optimization.

6.2.1 Testing with Multiple Tools

SysBench, BenchmarkSQL, and a modified tpcc-mysql tool are primarily used for testing. SysBench tests read/write, write-only, and Pareto distribution scenarios. BenchmarkSQL is used for TPC-C testing across various concurrency levels. The modified tpcc-mysql tool tests MySQL and Group Replication under extreme conditions.

6.2.2 Identifying Contention-Intensive Tests

The more concurrent threads accessing the same data, the greater the contention. SysBench Pareto distribution tests are considered high-contention, while SysBench uniform distribution tests are low-contention. In modified BenchmarkSQL, testing with 1000 warehouses represents low contention, while testing with 10 warehouses under the same concurrency level indicates high contention.

6.2.3 Understanding TPC-C Testing Characteristics

The TPC-C benchmark is the gold standard for database concurrency control in both research and industry [39].

Experiment settings can significantly impact evaluation results. In TPC-C:

- Introducing wait time makes experiments I/O intensive.
- Removing wait time makes experiments CPU/memory intensive.
- Reducing the number of warehouses makes experiments contention intensive.

TPC-C can stress test almost every key component of a computer system, but this versatility poses challenges for fair comparisons between different systems [15].

At a high level, the following factors reduce contention:

- More warehouses

- Fewer cross-warehouse transactions
- Fewer workers/users per warehouse
- Adding wait time
- Short or no I/Os within a critical section

In low-contention settings, throughput is limited by the system's slowest component:

- Disk I/O if data exceeds DRAM size
- Network I/O if using traditional TCP stack and data fits in DRAM
- Centralized sequencers or global dependency graphs may also cause scalability bottlenecks

Conversely, the following factors increase contention:

- Fewer warehouses
- More cross-warehouse transactions
- More workers/users per warehouse
- No wait time
- Long I/Os within a critical section

In high-contention settings, throughput is determined by the concurrency control mechanism. Systems that can release locks earlier or reduce aborts will have advantages [15].

The following figure shows the relationship between TPC-C throughput and concurrency for different numbers of warehouses. The dark blue curve represents 100 warehouses, while the deep red curve represents 1000 warehouses. The figure illustrates that throughput for 100 warehouses is significantly lower than for 1000 warehouses due to more severe contention in the former scenario.

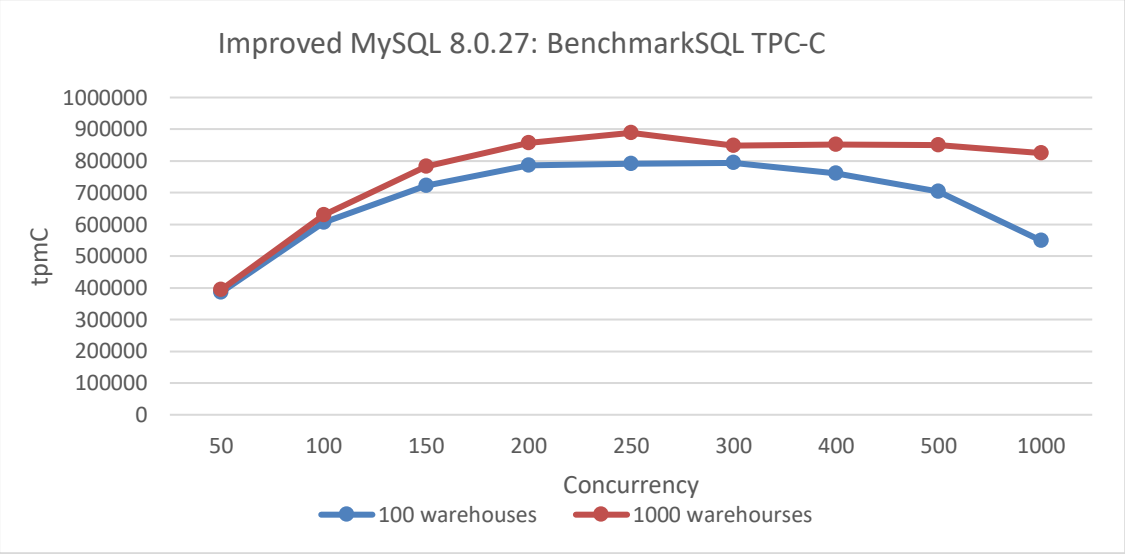


Figure 6-2. More warehouses indicate less severe contention.

6.2.4 Incorporating Thinking Time in Testing

TPC-C includes a wait time, simulating user behavior with keying and think times before each transaction. Each warehouse has ten terminals, one per district, allowing a maximum of ten concurrent transactions per warehouse. The vanilla TPC-C does not allow tuning these parameters, except for the number of warehouses.

Due to the wait time and the ten concurrent user limit per warehouse, the maximal throughput per warehouse is limited. To achieve higher throughput, many warehouses must be used, which is why systems like OceanBase and Oracle use millions of warehouses. For workloads storing large data with low throughput per GB, SSDs can meet throughput requirements more cost-effectively than DRAM, aligning with the hardware configurations of OceanBase and Oracle. Thus, vanilla TPC-C is an I/O intensive benchmark with low contention, as long wait times and limited concurrent users per warehouse mean low simultaneous access probability.

Most research addresses contentions, necessitating the removal of wait time for higher throughput and contention levels per warehouse. Fewer warehouses are used to maintain contention. TPC-C numbers with and without wait time are not comparable due to stress testing different system components [15].

TPC-C testing with a 1ms thinking time (pause before requests) is shown in the following figure,

illustrating the throughput-concurrency relationship.

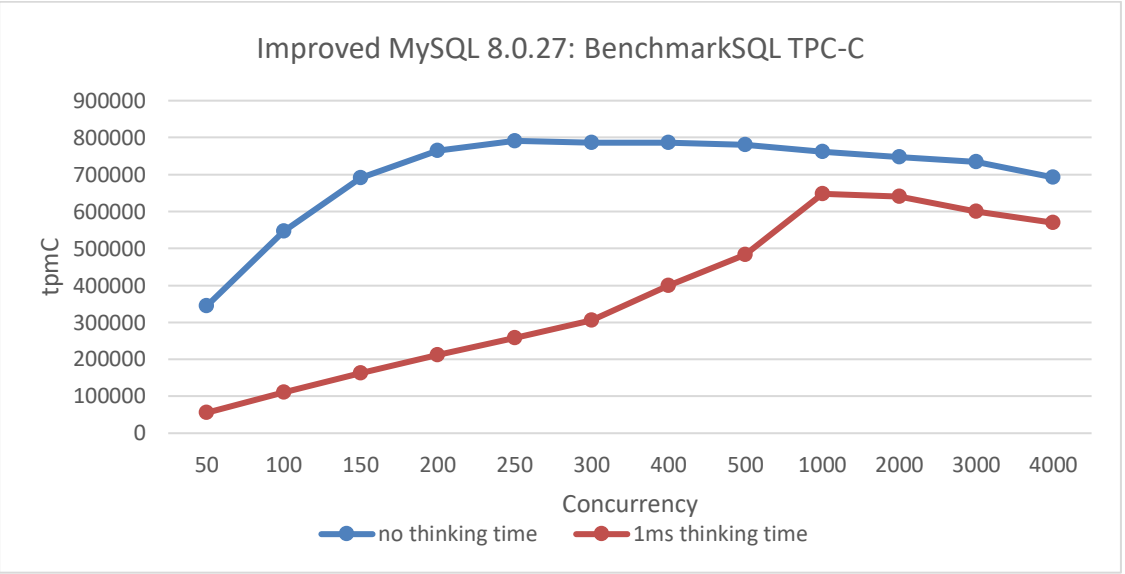


Figure 6-3. Impact of 1ms thinking time on throughput.

From the figure, it can be observed that with a 1ms thinking time, the peak throughput is achieved at 1000 concurrency, whereas in normal TPC-C testing, the peak is reached at 250 concurrency. Testing with thinking time introduces significant differences compared to regular BenchmarkSQL testing and also requires MySQL to have better scalability.

6.2.5 Impact of I/O on Testing

A fully-featured database system must handle network I/Os for protocols like 2PC, data replication, or Paxos, and disk I/Os for data persistence. Many studies omit these I/Os, raising questions about their impact.

Results may vary with hardware and implementation, but generally, TCP alone can support millions of transactions per second if only one packet is needed per TPC-C transaction. However, adding protocols like 2PC and replication increases packet requirements significantly. For instance, standard Paxos replication to three replicas requires at least four packets per transaction, which can overwhelm the TCP stack and become a bottleneck for high transaction rates.

Network and disk latency also impact throughput in contended workloads. Longer latencies in critical sections will reduce maximum throughput, with I/O latency having a more significant effect.

However, no solution is perfect: RDMA, for example, involves costly hardware and complex software and does not significantly aid in geo-distributed environments [15].

6.2.6 Long-Term Stability Testing

According to the TPC-C benchmark, the database must operate in a steady state for eight hours and exceed two hours in the performance collection phase. Additionally, the benchmark requires the database to maintain less than 2% jitter over two hours of testing [25].

To meet these stability requirements in MySQL testing, the following measures were implemented:

1. Regularly cleaning the binlog to prevent SSD performance degradation due to I/O space constraints.
2. Utilizing a larger number of warehouses.
3. Adding indexes.
4. Deploying multiple SSDs.

Following these measures, TPC-C testing was performed using BenchmarkSQL. The figure below illustrates the stability test comparison between MySQL 8.0.27 and the improved MySQL 8.0.27.

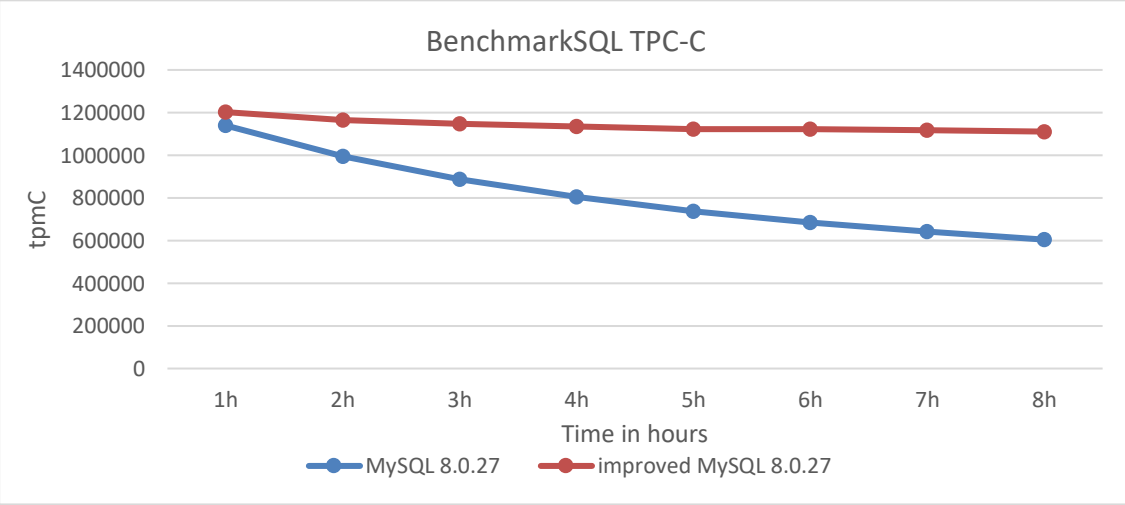


Figure 6-4. Comparison of stability tests: MySQL 8.0.27 vs. improved MySQL 8.0.27.

From the figure, it is evident that although MySQL and the improved MySQL start with similar throughput, the throughput of MySQL decreases more rapidly over time than expected, while the

improved MySQL remains significantly more stable.

Additionally, comparisons were made for the improved MySQL at different concurrency levels. The figure below shows the throughput over time: the deep blue curve represents 100 concurrency, while the deep red curve represents 200 concurrency.

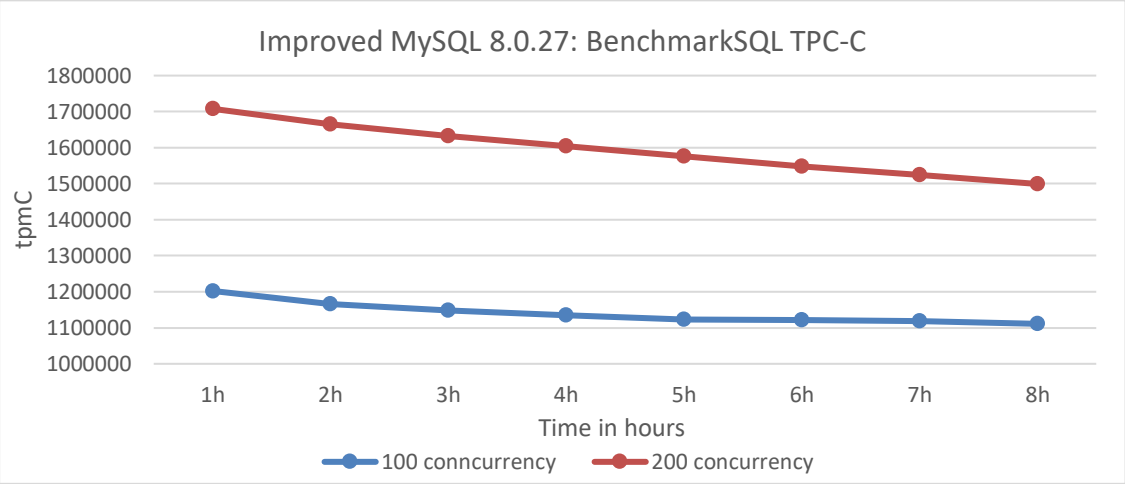


Figure 6-5. Stability test comparison: 100 vs. 200 concurrency.

From the figure, it can be observed that throughput is more stable at 100 concurrency. At 200 concurrency, the increased data processing leads to a faster decline in throughput, as the larger database scale on a single server results in slower access times.

The following figure compares the size of MySQL database files after completing an 8-hour test at 100 and 200 concurrency levels.

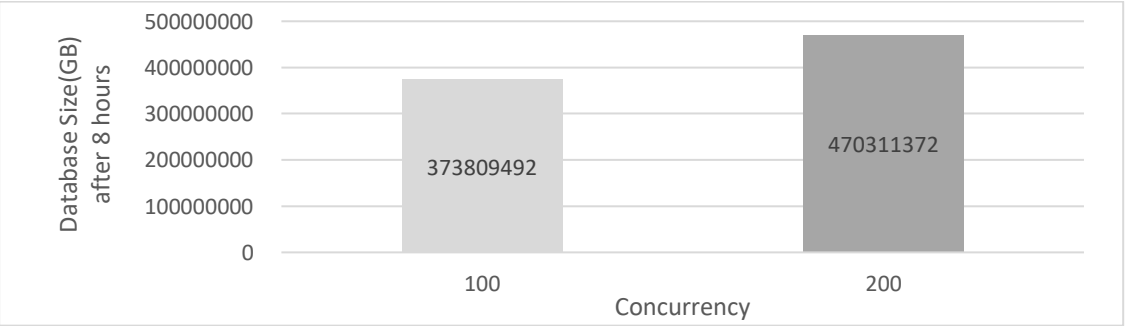


Figure 6-6. Database size comparison: 100 vs. 200 concurrency after 8 hours.

From the figure, it is evident that at 200 concurrency, the database size is significantly larger than at

100 concurrency. Generally, higher throughput tends to reduce stability, and various factors influence stability testing, making broad generalizations challenging.

6.2.7 Online Traffic Testing

Performance testing often fails to reflect real-world use cases and is typically reported with insufficient detail for replication or drawing accurate conclusions [72].

TCPCopy can capture a production workload and replay it on a test system, preserving the exact timing, concurrency, and transaction characteristics of the original workload. This allows for testing the impact of system changes without affecting the production environment.

The following figure illustrates the mechanism of replicating MySQL traffic from the online system to the MySQL testing system.

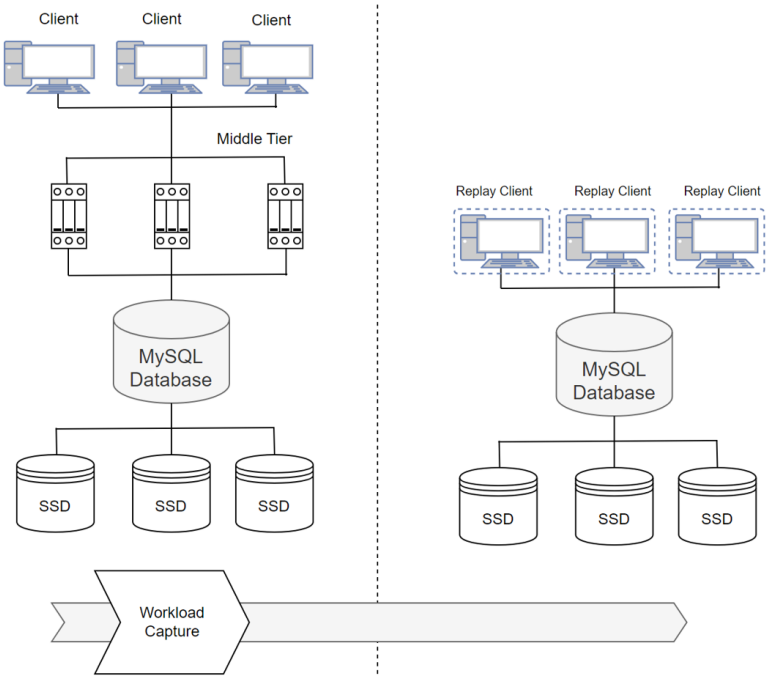


Figure 6-7. Mechanism for replicating MySQL traffic from production to testing systems.

Testing real-world online traffic is crucial for MySQL, as it effectively reveals potential issues such as performance stability, memory leaks, and the robustness of new MySQL versions.

Part3 Practice

Chapter 7: Key Improvements of MySQL 8.0 Over MySQL 5.7

MySQL 8.0 has introduced substantial improvements over MySQL 5.7. It not only enhances functionality and adds support for hash joins in execution plans but, more importantly, greatly improves scalability. These advancements lay a solid foundation for future enhancements.

7.1 Scaling Up: InnoDB Improvements

Early open-source DBMS code often used a coarse-grained latch for the entire kernel. In contrast, InnoDB has adopted a more refined approach, employing separate latches for different kernel components, such as the lock manager and buffer pool [31].

MySQL continues to improve scalability, and with MySQL 8.0, further enhancements have been made to the InnoDB storage engine's scalability. Here are the related improvements:

1. **Redo Log Optimization:** Enhancements to the redo log have facilitated subsequent performance improvements.
2. **Lock-sys Latch Sharding:** The lock-sys latch has been sharded, akin to read-write locks, to improve transactional locking.
3. **Trx-sys Latch Splitting and Sharding:** While contention for latches persists, optimizations in trx-sys lay a strong foundation for future improvements in MVCC ReadView.

These significant scalability improvements will be discussed in detail below.

7.1.1 Redo Log Optimization

Write-ahead logging is a fundamental, omnipresent component in ARIES-style concurrency and recovery, and it represents a significant potential bottleneck, especially in OLTP workloads making frequent small changes to data. Two logging-related impediments to database system scalability are identified, each challenging different levels of the software architecture [6]:

1. The high volume of small-sized I/O requests may saturate the disk.
2. Contention arises as transactions serialize access to in-memory log data structures.

The above potential bottlenecks are reflected in MySQL 5.7. Detailed information on redo log optimization can be found in "MySQL 8.0: New Lock-Free, Scalable WAL Design", where the complexity lies in how the sequential order of Log Sequence Numbers (LSN) is ensured in the new design. The article also highlights the following improvements [46]:

We have introduced dedicated threads for particular tasks related to the redo log writes. User threads no longer do writes to the redo files themselves. They simply wait when they need redo flushed to disk and it is not flushed yet.

This improvement completely changed the previous mechanism and laid a solid foundation for scalability. The following git log details the specific optimizations made to the redo log.

```
commit 6be2fa0bdbbadc52cc8478b52b69db02b0eaff40
Author: Paweł Olchawa <pawel.olchawa@oracle.com>
Date:   Wed Feb 14 09:33:42 2018 +0100

    WL#10310 Redo log optimization: dedicated threads and concurrent log buffer.

    0. Log buffer became a ring buffer, data inside is no longer shifted.
    1. User threads are able to write concurrently to log buffer.
    2. Relaxed order of dirty pages in flush lists - no need to synchronize
       the order in which dirty pages are added to flush lists.
    3. Concurrent MTR commits can interleave on different stages of commits.
    4. Introduced dedicated log threads which keep writing log buffer:
        * log_writer: writes log buffer to system buffers,
        * log_flusher: flushes system buffers to disk.
       As soon as they finished writing (flushing) and there is new data to
       write (flush), they start next write (flush).
    5. User threads no longer write / flush log buffer to disk, they only
       wait by spinning or on event for notification. They do not have to
       compete for the responsibility of writing / flushing.
    6. Introduced a ring buffer of events (one per log-block) which are used
       by user threads to wait for written/flushed redo log to avoid:
        * contention on single event
        * false wake-ups of all waiting threads whenever some write/flush
          has finished (we can wake-up only those waiting in related blocks)
    7. Introduced dedicated notifier threads not to delay next writes/fsyncs:
        * log_write_notifier: notifies user threads about written redo,
        * log_flush_notifier: notifies user threads about flushed redo.
    8. Master thread no longer has to flush log buffer.
    ...
    30. Mysql test runner received a new feature (thanks to Marcin):
        --exec_in_background.
```

```
Review: RB#15134
Reviewers:
   - Marcin Babij <marcin.babij@oracle.com>,
   - Debarun Banerjee <debarun.banerjee@oracle.com>.
Performance tests:
   - Dimitri Kravtchuk <dimitri.kravtchuk@oracle.com>,
   - Daniel Blanchard <daniel.blanchard@oracle.com>,
   - Amrendra Kumar <amrendra.x.kumar@oracle.com>.
QA and MTR tests:
   - Vinay Fisrekar <vinay.fisrekar@oracle.com>.
```

The new mechanism employs dedicated threads to flush redo log files, supports concurrent writes to the log buffer, removes global latches in the code, and introduces latch-free processing, significantly enhancing scalability.

A test comparing TPC-C throughput with different levels of concurrency before and after optimization was conducted. Specific details are shown in the following figure:

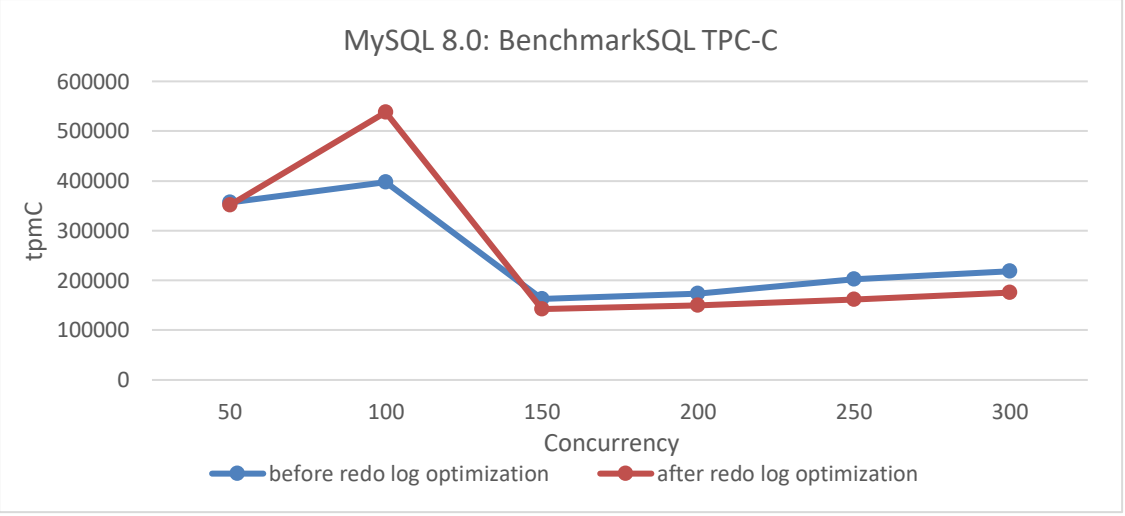


Figure 7-1. Impact of redo log optimization under different concurrency levels.

The results in the figure show a significant improvement in throughput at low concurrency levels but a decrease at high concurrency levels. This decrease can be attributed to two potential reasons:

1. **Unresolved Foundational Flaws:** During the transformation process, foundational issues may not have been fully addressed.

2. **Interference from Multiple Queue Bottlenecks:** Issues similar to multi-queue bottlenecks

interfering with each other may arise. Although performance in some areas has improved, other bottlenecks have worsened under high concurrency.

Extensive research suggests that the optimization should theoretically enhance throughput. The redo log optimization uses a group commit-like mechanism to reduce I/O overhead. Instead of immediately flushing redo log contents, user threads write to a log buffer and wait, while a dedicated thread batches and flushes the log to disk, notifying user threads when the process is complete. This approach is expected to significantly decrease I/O operations under high concurrency. Therefore, the most likely cause of performance issues is exacerbated bottlenecks in other queues.

Implementing redo log optimization is highly challenging, and without it, achieving throughput levels in the millions of tpmC would be nearly impossible.

Extensive testing revealed that the optimizations performed well under low concurrency conditions and significantly accelerated the TPC-C data loading process. Specific details are shown in the following figure:

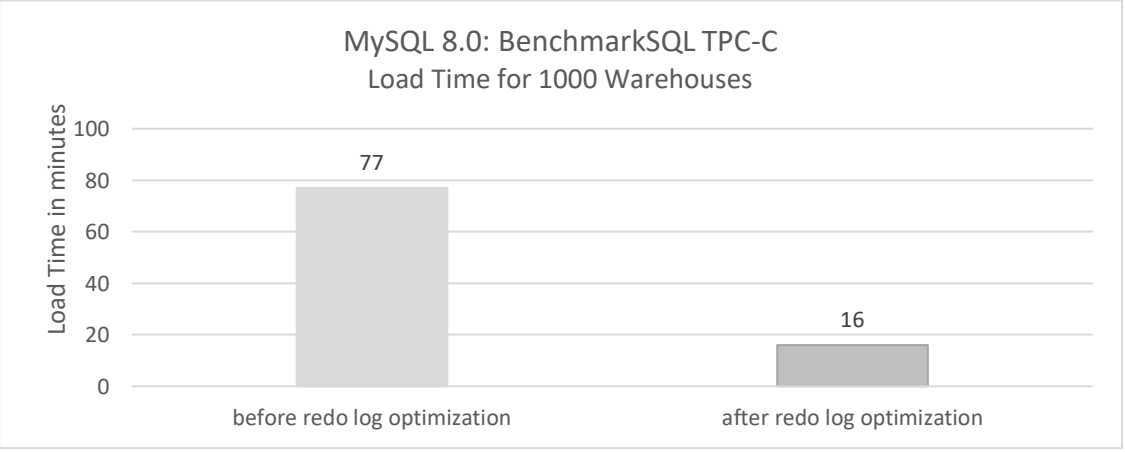


Figure 7-2. Impact of redo log optimization for TPC-C data loading time.

The TPC-C data loading process involves large transactions of up to 100MB. Previously, loading 1000 warehouses took 77 minutes, but with the optimization, it now takes only 16 minutes. This demonstrates that redo log optimization is highly effective for handling large transactions.

To assess the true value of this optimization, scalability enhancements were applied to MySQL 5.7.36. This process involved first applying the trx-sys patch, followed by the lock-sys patch, to evaluate the extent of throughput improvement. Specific details are shown in the following figure:

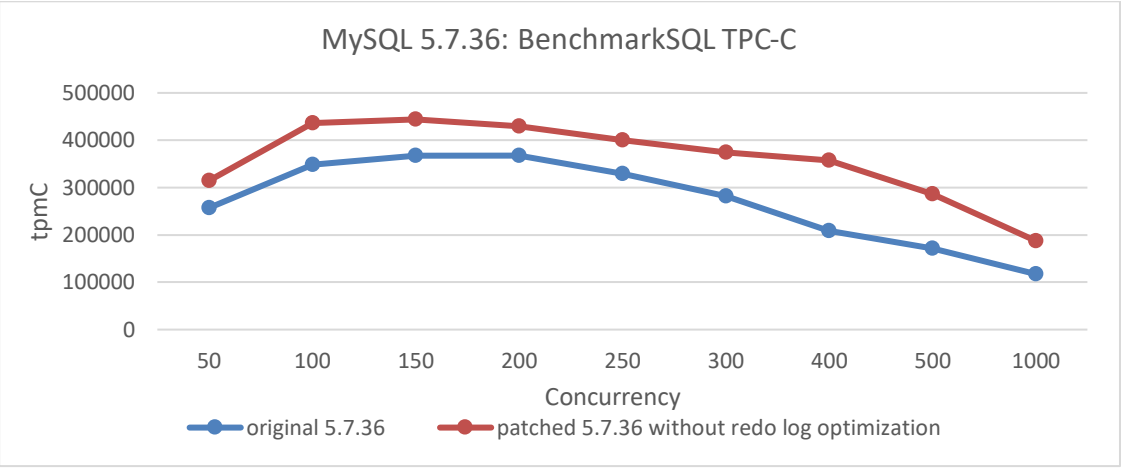


Figure 7-3. Indirect impact of redo log optimization.

From the figure, it can be seen that after applying the trx-sys and lock-sys scalability patches, MySQL 5.7.36 experienced an improvement in throughput. However, it did not fundamentally resolve scalability issues, especially when compared to the improved MySQL 8.0.27 version. The gap remains significant. To identify the bottleneck at 250 concurrency, the screenshot from the perf tool can be examined. Specific details are shown in the following figure:

```
Samples: 3M of event 'cycles', 4000 Hz, Event count (approx.): 1472987494835 lost: 0/0 drop: 0/0
Overhead  Shared Object             Symbol
   6.91%  mysqld                   [.] ut_delay
   5.15%  mysqld                   [.] mtr_t::Command::prepare_write
   4.64%  [kernel]                 [k] native_queued_spin_lock_slowpath.part.0
   2.63%  libpthread-2.28.so       [.] __pthread_mutex_cond_lock
   2.55%  mysqld                   [.] MYSQLparse
   2.33%  mysqld                   [.] btr_cur_search_to_nth_level
   1.80%  [kernel]                 [k] update_sg_lb_stats
   1.71%  mysqld                   [.] rec_get_offsets_func
   1.40%  mysqld                   [.] buf_page_get_gen
   1.38%  mysqld                   [.] row_search_mvcc
   0.97%  libc-2.28.so             [.] __memmove_avx_unaligned_erms
   0.89%  mysqld                   [.] buf::Block_hint::buffer_fix_block_if_still_valid
   0.86%  mysqld                   [.] locksys::Global_shared_latch_guard::Global_shared_latch_guard
   0.82%  mysqld                   [.] cmp_dtuple_rec_with_match_low
   0.78%  mysqld                   [.] lex_one_token
   0.72%  mysqld                   [.] my_lfind
   0.62%  mysqld                   [.] page_cur_search_with_match
   0.59%  mysqld                   [.] buf_page_optimistic_get
   0.58%  [kernel]                 [k] _raw_spin_lock
```

Figure 7-4. Screenshot from the perf tool at 250 concurrency.

From the figure, it is evident that the bottleneck is ***prepare_write***, which precisely corresponds to the bottleneck of writing redo log buffer in MySQL 5.7.36 version.

```
/** Prepare to write the mini-transaction log to the redo log buffer.
@return number of bytes to write in finish_write() */
ulint mtr_t::Command::prepare_write() {
```

```
switch (m_impl->m_log_mode) {
  case MTR_LOG_SHORT_INSERTS:
    ut_d(ut_error);
    /* fall through (write no redo log) */
    [[fallthrough]];
  case MTR_LOG_NO_REDO:
  case MTR_LOG_NONE:
    ut_ad(m_impl->m_log.size() == 0);
    return 0;
  case MTR_LOG_ALL:
    break;
  default:
    ut_d(ut_error);
    ut_o(return 0);
}
...
```

Let's analyze this function by examining its call stack relationship.

```
mysqld 3424449 24930974.012558:    1743550 cycles:
            55d7b7fa211d ut_delay+0x1d (/home/wangbin/mysql_old/bin/mysqld)
            55d7b7e8441c mtr_t::Command::prepare_write+0x53c (/home/wangbin/mysql_old/bin/mysqld)
            55d7b7e85e7b mtr_t::Command::execute+0x2b (/home/wangbin/mysql_old/bin/mysqld)
            55d7b7e863e9 mtr_t::commit+0x89 (/home/wangbin/mysql_old/bin/mysqld)
            55d7b7f2c191 row_upd_clust_rec+0xc1 (/home/wangbin/mysql_old/bin/mysqld)
            55d7b7f3156d row_upd_clust_step+0x84d (/home/wangbin/mysql_old/bin/mysqld)
            55d7b7f32f3b row_upd+0x6b (/home/wangbin/mysql_old/bin/mysqld)
            55d7b7f3320b row_upd_step+0x8b (/home/wangbin/mysql_old/bin/mysqld)
            55d7b7ef6685 row_update_for_mysql_using_upd_graph+0x1c5 (/home/wangbin/mysql_old/bin/mysqld)
            55d7b7efb35f row_update_for_mysql+0x3f (/home/wangbin/mysql_old/bin/mysqld)
            55d7b7e0d9c7 ha_innobase::update_row+0x127 (/home/wangbin/mysql_old/bin/mysqld)
            55d7b7e2ae31 non-virtual thunk to ha_innopart::update_row_in_part(unsigned int, unsigned char const*, unsigned char*)+0x31
            55d7b7add857 Partition_helper::ph_update_row+0x147 (/home/wangbin/mysql_old/bin/mysqld)
            55d7b76b730b handler::ha_update_row+0x1ab (/home/wangbin/mysql_old/bin/mysqld)
            55d7b7c0abc0 mysql_update+0x1810 (/home/wangbin/mysql_old/bin/mysqld)
            55d7b7c0cdcd Sql_cmd_update::try_single_table_update+0x1ed (/home/wangbin/mysql_old/bin/mysqld)
            55d7b7c0d106 Sql_cmd_update::execute+0x36 (/home/wangbin/mysql_old/bin/mysqld)
            55d7b7b7dc4a mysql_execute_command+0x8ca (/home/wangbin/mysql_old/bin/mysqld)
            55d7b7b83b95 mysql_parse+0x3e5 (/home/wangbin/mysql_old/bin/mysqld)
            55d7b7b84ec4 dispatch_command+0x1274 (/home/wangbin/mysql_old/bin/mysqld)
            55d7b7b861a0 do_command+0x220 (/home/wangbin/mysql_old/bin/mysqld)
            55d7b7c4cd18 handle_connection+0x298 (/home/wangbin/mysql_old/bin/mysqld)
            55d7b814b7e4 pfs_spawn_thread+0x154 (/home/wangbin/mysql_old/bin/mysqld)
            7ff596e0817a start_thread+0xea (/usr/lib64/libpthread-2.28.so)
```

Figure 7-5. Call stack relationship revealing bottleneck in redo log writing.

The figure clearly shows that the bottleneck lies in redo log writing. Without the redo log optimization patch, the scalability issues in MySQL 5.7.36 cannot be fundamentally resolved, underscoring the significant impact of redo log optimization.

Does redo log optimization currently have any side effects? Test data indicates that under low concurrency conditions, the number of flush operations increases significantly. Using SysBench read-write tests, the relationship between the average number of I/O flushes per transaction and concurrency was statistically analyzed. Specific details are shown in the following figure:

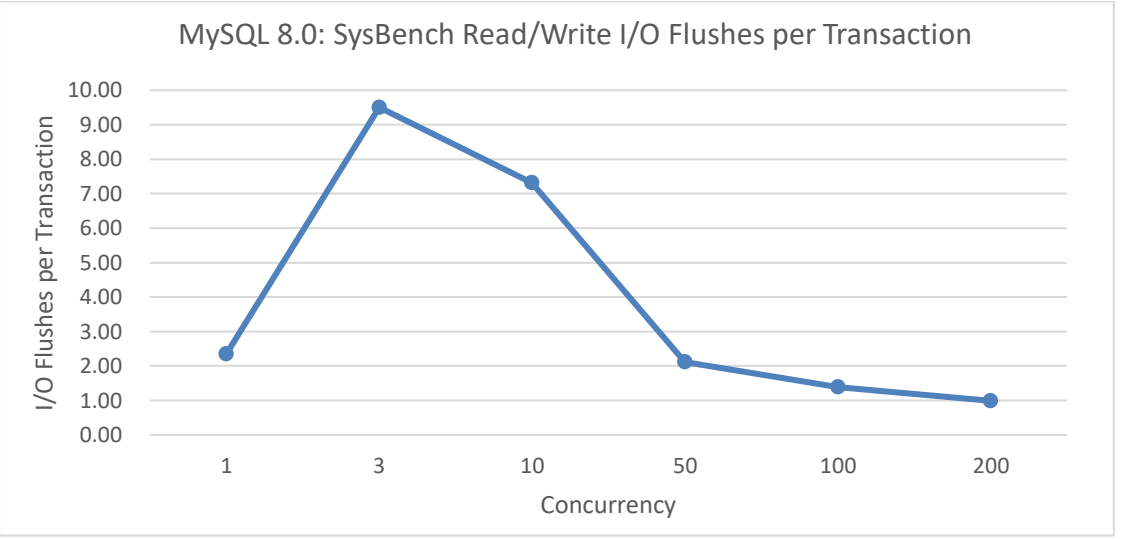


Figure 7-6. Side effects of redo log optimization at low concurrency: more I/O flushes.

From the figure, it can be observed that with 3 concurrent read-write operations, each transaction averages over 9 flushes, while at 200 concurrency, it decreases to around 1 flush per transaction. These average flush counts can be further optimized, but it requires finding a balance: timely flushing activates user threads more quickly but incurs higher I/O overhead, whereas delaying flushing reduces I/O costs but may increase user response times.

7.1.2 Optimizing Lock-Sys Through Latch Sharding

In MySQL 5.7, the lock system experienced significant latch contention issues, which severely impacted throughput under high concurrency. During transaction execution, frequent locking and unlocking operations require acquiring a global latch. When many user threads compete for this global latch, MySQL's scalability becomes a major concern.

Lock-sys optimization is the second major improvement made in MySQL 8.0. The following git log describes the specific details of the lock-sys optimization.

```
commit 1d259b87a63defa814e19a7534380cb43ee23c48
Author: Jakub Łopuszański <jakub.lopuszanski@oracle.com>
Date:   Wed Feb 5 14:12:22 2020 +0100

    WL#10314 - InnoDB: Lock-sys optimization: sharded lock_sys mutex

    The Lock-sys orchestrates access to tables and rows. Each table, and each row,
    can be thought of as a resource, and a transaction may request access right for
```

```
a resource. As two transactions operating on a single resource can lead to
problems if the two operations conflict with each other, Lock-sys remembers
lists of already GRANTED lock requests and checks new requests for conflicts in
which case they have to start WAITING for their turn.

Lock-sys stores both GRANTED and WAITING lock requests in lists known as queues.
To allow concurrent operations on these queues, we need a mechanism to latch
these queues in safe and quick fashion.

In the past a single latch protected access to all of these queues.
This scaled poorly, and the managment of queues become a bottleneck.
In this WL, we introduce a more granular approach to latching.

Reviewed-by: Pawel Olchawa <pawel.olchawa@oracle.com>
Reviewed-by: Debarun Banerjee <debarun.banerjee@oracle.com>
  RB:23836
```

Sharding the global latch theoretically can significantly improve scalability under high concurrency situations. Based on the program before and after optimizing with lock-sys, using BenchmarkSQL to compare TPC-C throughput with concurrency, the specific results are as shown in the following figure:

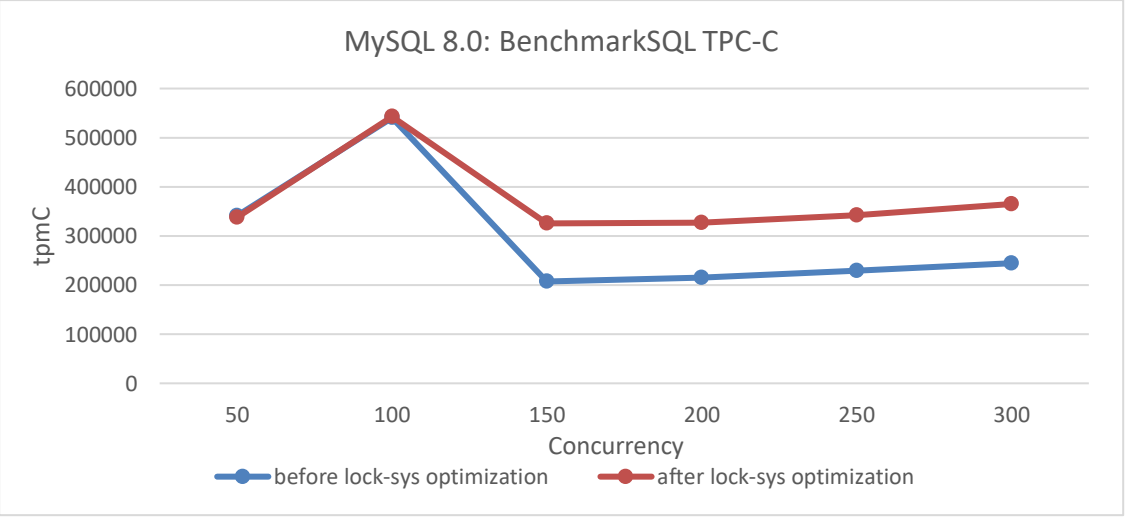


Figure 7-7. Impact of lock-sys optimization under different concurrency levels.

From the figure, it can be seen that optimizing lock-sys significantly improves throughput under high concurrency conditions, while the effect is less pronounced under low concurrency due to fewer conflicts.

7.1.3 Latch Splitting in trx-sys

The trx-sys subsystem in MySQL, closely related to MVCC, primarily involves read operations. Improvements to redo log and lock-sys are mainly associated with write operations.

MySQL 5.7 utilized a global latch to synchronize various operations within trx-sys. To enhance read capabilities, it was crucial to address this latch bottleneck. However, the intertwined logic made modification challenging.

In MySQL 8.0, the global latch was initially split. A new latch was introduced for the ***serialization_list***, allowing bypass of the global latch and reducing the contention pressure on it. The following git log describes the specific details of these optimizations.

```
commit e66d48b0c73d5fec278f81784bd5697502990263
Author: Paweł Olchawa <pawel.olchawa@oracle.com>
Date:   Mon Mar 1 15:52:30 2021 +0100

    BUG#27933068 USE DIFFERENT MUTEX TO PROTECT TRX_SYS->SERIALISATION_LIST

    This is an optimization patch, which reduces contention on the trx_sys_t::mutex
    by introducing a new mutex - the trx_sys_t::serialisation_mutex.

    The new mutex protects the trx_sys_t::serialisation_list and replaces the
    trx_sys_t::mutex when trx->no is being assigned.

    This is a modified version of the contribution patch which was created by Zhai Weixiang.
    Modifications:
    1. Periodical write of max_trx_id to the transaction system header page is modified.
    2. The trx_get_serial_no() is called when we do not hold trx_sys_t::mutex.
    3. Members in trx_sys_t are rearranged, so they are grouped by mutex that protects them.
    4. The new mutex received its own latch_id.
    5. InnoDB relies on rw_trx_max_id instead of max_trx_id in few places.
    6. The min_active_id is updated only when it really changes.

    RB: 19712
    Reviewed-by: Debarun Banerjee debarun.banerjee@oracle.com
```

Based on this optimization before and after, using BenchmarkSQL to compare TPC-C throughput with concurrency, the specific results are shown in the following figure:

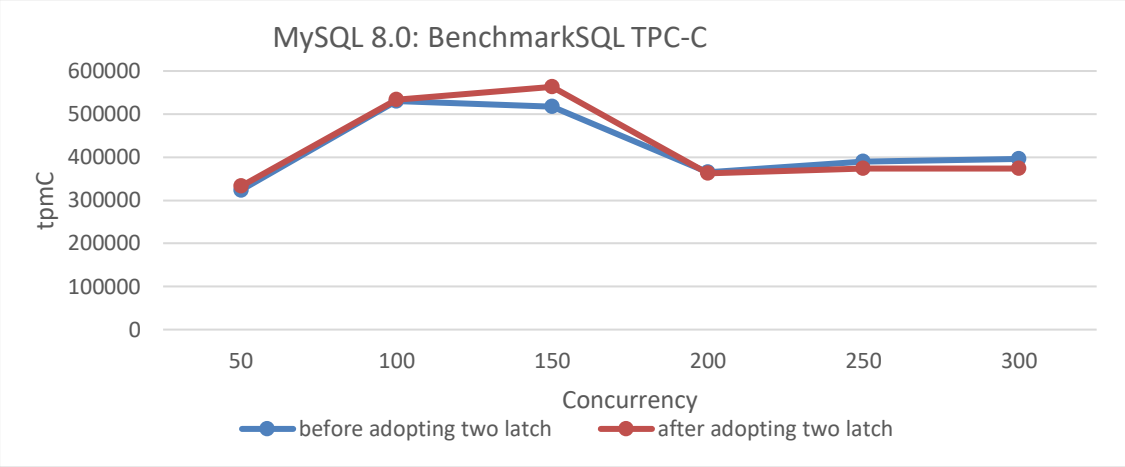


Figure 7-8. Impact of latch splitting in trx-sys under different concurrency levels.

From the figure, it can be seen that the optimization is effective at 150 concurrency. However, beyond 200 concurrency, throughput not only fails to increase but actually decreases. This decline at high concurrency levels is primarily due to interference from other queue bottlenecks.

7.1.4 Latch Sharding for trx-sys

In MySQL 8.0, further scalability improvements are made to the trx-sys subsystem. The ***rw_trx_set*** has been divided into shards, each with its own latch. This significantly reduces global latch contention for read operations. The following git log describes the specific details of these optimizations.

```
commit bc95476c0156070fd5cedcfd354fa68ce3c95bdb
Author: Paweł Olchawa <pawel.olchawa@oracle.com>
Date:   Tue May 25 18:12:20 2021 +0200

    BUG#32832196 SINGLE RW_TRX_SET LEADS TO CONTENTION ON TRX_SYS MUTEX

    1. Introduced shards, each with rw_trx_set and dedicated mutex.
    2. Extracted modifications to rw_trx_set outside its original critical sections
       (removal had to be extracted outside trx_erase_lists).
    3. Eliminated allocation on heap inside TrxUndoRsegs.
    4. [BUG-FIX] The trx->state and trx->start_time became converted to std::atomic<>
       fields to avoid risk of torn reads on egzotic platforms.
    5. Added assertions which ensure that thread operating on transaction has rights
       to do so (to show there is no possible race condition).

    RB: 26314
```

Reviewed-by: Jakub Łopuszański jakub.lopuszanski@oracle.com

Based on these optimizations before and after, using BenchmarkSQL to compare TPC-C throughput with concurrency, the specific results are as shown in the following figure:

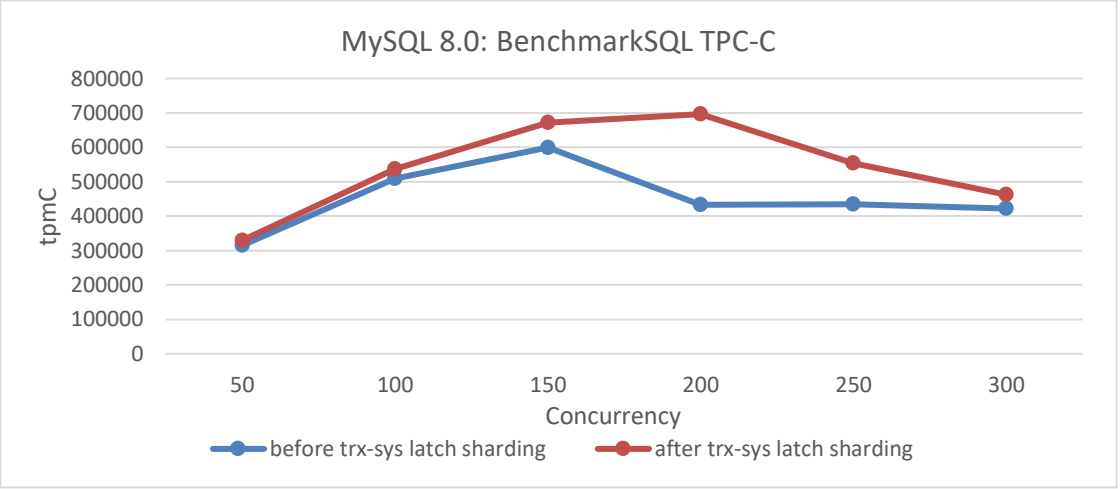


Figure 7-9. Impact of latch sharding in trx-sys under different concurrency levels.

From the figure, it can be seen that this improvement significantly enhances TPC-C throughput, reaching its peak at 200 concurrency. It is worth noting that the impact diminishes at 300 concurrency, primarily due to ongoing scalability issues in the trx-sys subsystem related to MVCC ReadView. This issue will be discussed further in the next chapter.

7.1.5 Summary

The series of scalability improvements mentioned above have laid a solid foundation for achieving high throughput in MySQL. Without these changes, subsequent improvements would lose their significance. Therefore, MySQL 8.0 has made significant advancements in scalability.

7.2 Evaluating Performance Gains in MySQL Lock Scheduling Algorithms

Scheduling is crucial in computer system design. The right policy can significantly reduce mean response time without needing faster machines, effectively improving performance for free. Scheduling also optimizes other metrics, such as user fairness and differentiated service levels, ensuring some job classes have lower mean delays than others [37].

MySQL 8.0 uses the Contention-Aware Transaction Scheduling (CATS) algorithm to prioritize transactions waiting for locks. When multiple transactions compete for the same lock, CATS determines the priority based on scheduling weight, calculated by the number of transactions a given transaction blocks. The transaction blocking the most others gets higher priority; if weights are equal, the longest waiting transaction goes first.

A deadlock occurs when multiple transactions cannot proceed because each holds a lock needed by another, causing all involved to wait indefinitely without releasing their locks.

After understanding the MySQL lock scheduling algorithm, let's examine how this algorithm affects throughput. Before testing, it is necessary to understand the previous FIFO algorithm and how to restore it. For relevant details, refer to the git log explanations provided below.

This WL improves the implementation of CATS to the point where the FCFS will be redundant (as often slower, and easy to "emulate" by setting equal schedule weights in CATS), so it removes FCFS from the code, further simplifying the lock_sys's logic.

Based on the red prompt, restoring the FIFO lock scheduling algorithm in MySQL is straightforward. Subsequently, throughput was tested using SysBench Pareto distribution scenarios with varying concurrency levels in the improved MySQL 8.0.32. Details are provided in the following figure.

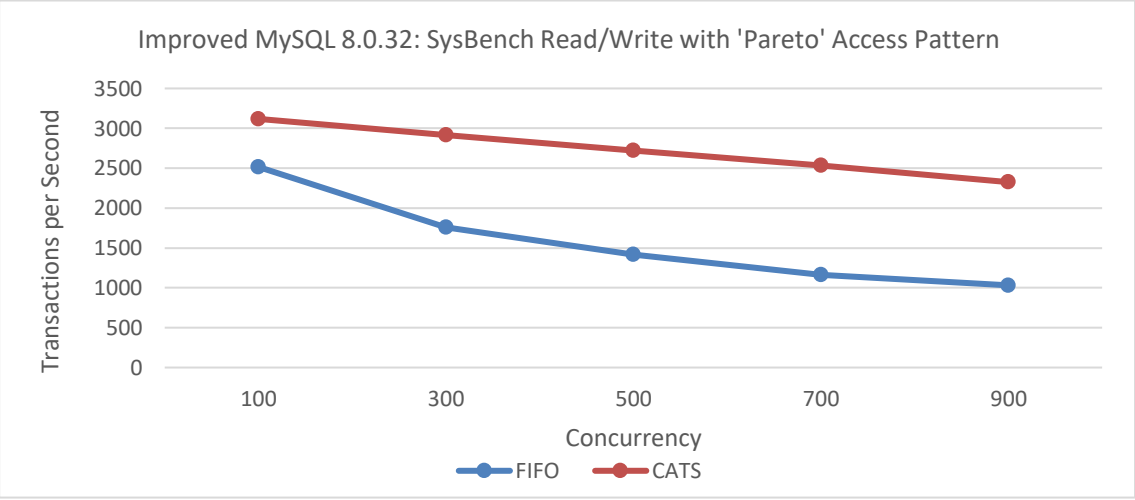


Figure 7-10. Impact of CATS on thoughput at various concurrency levels.

From the figure, it can be seen that the throughput of the CATS algorithm significantly exceeds that of the FIFO algorithm. To compare these two algorithms in terms of user response time, refer to the following figure.

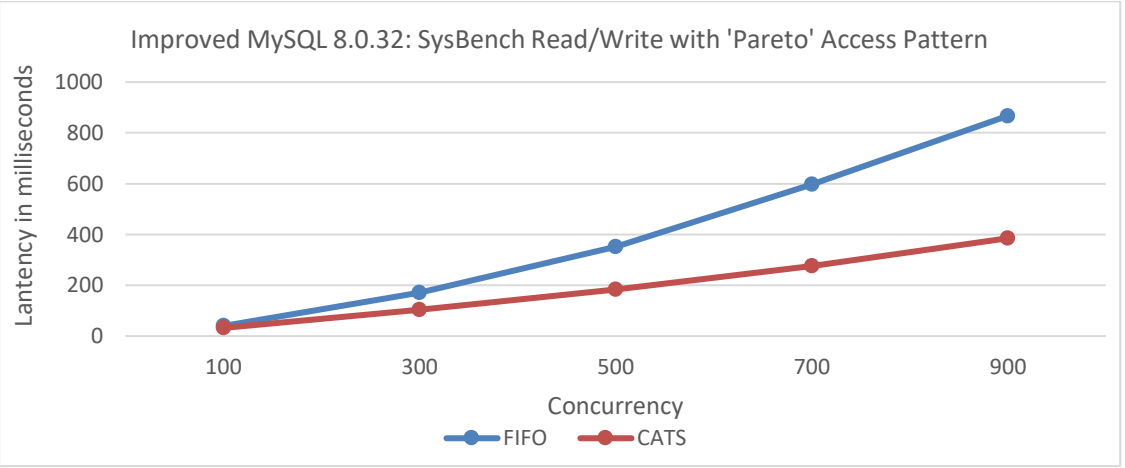


Figure 7-11. Impact of CATS on response time at various concurrency levels.

From the figure, it can be seen that the CATS algorithm provides significantly better user response times.

Furthermore, comparing deadlock error statistics during the Pareto distribution test process, details can be found in the following figure.

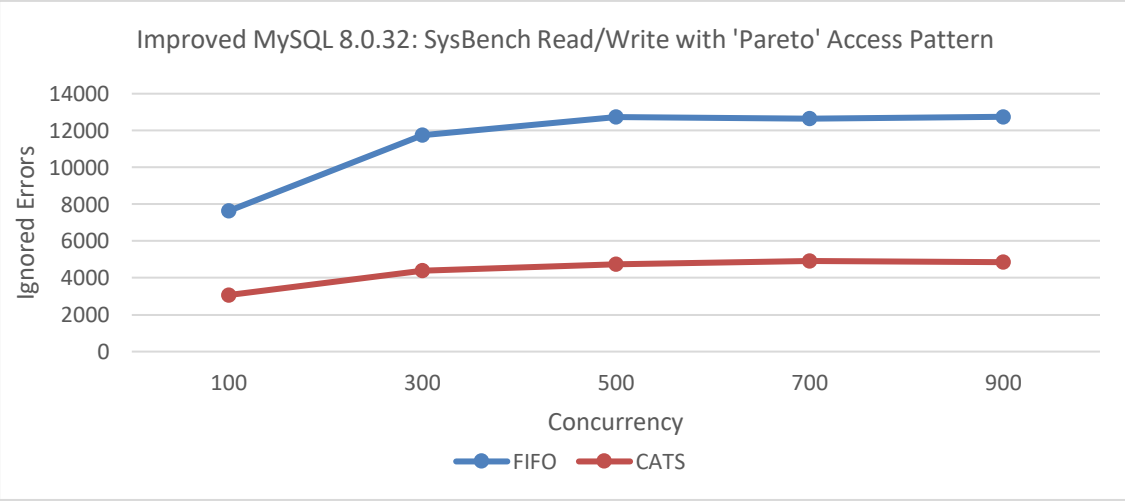


Figure 7-12. Impact of CATS on ignored errors at various concurrency levels.

Comparative analysis shows that the CATS algorithm significantly reduces deadlocks. This reduction in deadlocks likely plays a key role in enhancing performance. The theoretical basis for this correlation is as follows [15]:

Under a high-contention setting, the throughput of the target system will be determined by the concurrency control mechanism of the target system: systems which can release locks earlier or reduce the number of aborts will have advantages in such a setting.

The above test results align closely with MySQL's official findings. The following two figures, based on official tests [102], demonstrate the significant effectiveness of the CATS algorithm.

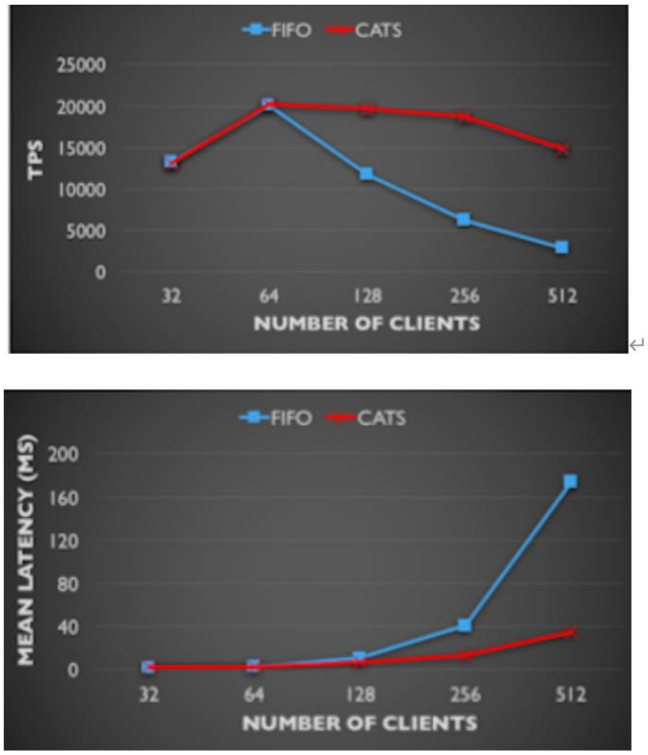


Figure 7-13. Comparison of CATS and FIFO in TPS and mean latency: insights from MySQL blog.

Additionally, MySQL's official requirements for implementing the CATS algorithm are stringent. Specific details are provided in the following figure:

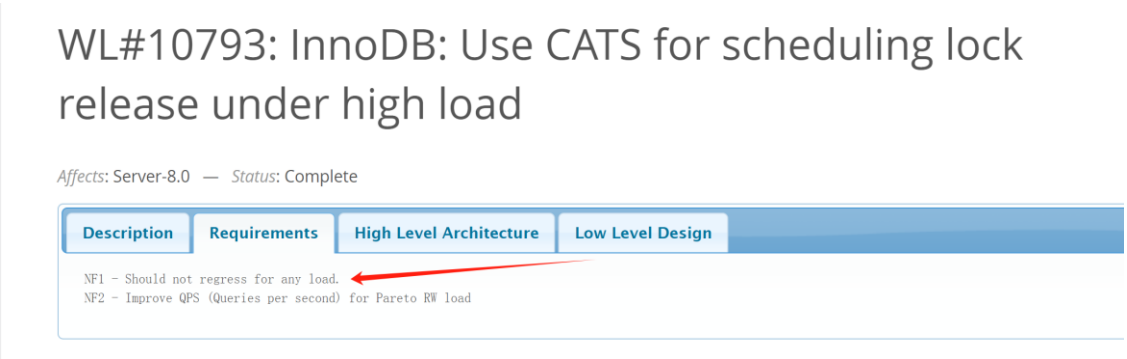
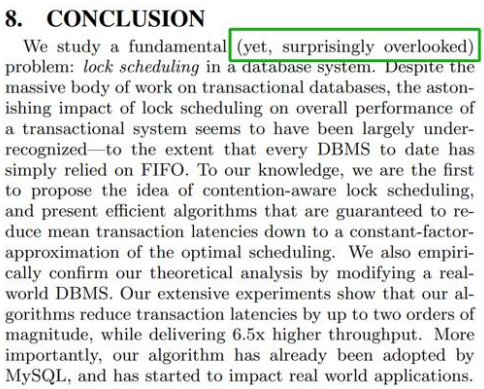


Figure 7-14. Requirements of the official worklog for CATS.

Therefore, with the adoption of the CATS algorithm, performance degradation should be absent in all scenarios. It seems like things end here, but the summary in the CATS algorithm's paper [37] raises some doubts. Details are provided in the following figure:



8. CONCLUSION

We study a fundamental (yet, surprisingly overlooked) problem: *lock scheduling* in a database system. Despite the massive body of work on transactional databases, the astonishing impact of lock scheduling on overall performance of a transactional system seems to have been largely underrecognized—to the extent that every DBMS to date has simply relied on FIFO. To our knowledge, we are the first to propose the idea of contention-aware lock scheduling, and present efficient algorithms that are guaranteed to reduce mean transaction latencies down to a constant-factor-approximation of the optimal scheduling. We also empirically confirm our theoretical analysis by modifying a real-world DBMS. Our extensive experiments show that our algorithms reduce transaction latencies by up to two orders of magnitude, while delivering 6.5x higher throughput. More importantly, our algorithm has already been adopted by MySQL, and has started to impact real world applications.

Figure 7-15. Doubts about the CATS paper.

Since the early days of databases, FIFO has been the mainstream lock scheduling algorithm, even under severe conflict. Over these years, the industry has neither replaced the FIFO lock scheduling algorithm nor found it causing throughput collapse. However, the paper questions the effectiveness of the FIFO algorithm. This suggests that either the industry has overlooked potential flaws in FIFO, or the paper's assessment is flawed, and FIFO does not have the serious issues suggested. This contradiction highlights a critical issue: one of these conclusions must be flawed; both cannot be correct.

Contradictions often present valuable opportunities for in-depth problem analysis and resolution. They highlight areas where existing understanding may be challenged or where new insights can be

gained.

This time, testing on the improved MySQL 8.0.27 revealed significant bottlenecks under severe conflicts. The ***perf*** screenshot is shown in the following figure:

```
Samples: 69K of event 'cycles', 4000 Hz, Event count (approx.): 23106178332 lost: 0/0 drop: 0/0
Overhead  Shared Object            Symbol
  14.44%  mysqld                   [.] lock_rec_grant
   8.80%  mysqld                   [.] DeadlockChecker::get_first_lock
   6.82%  mysqld                   [.] ut_delay
   6.61%  mysqld                   [.] DeadlockChecker::get_next_lock
   1.84%  mysqld                   [.] lock_wait_timeout_thread
   1.52%  mysqld                   [.] thd_killed
   1.11%  [kernel]                 [k] update_sg_lb_stats
   1.04%  mysqld                   [.] MYSQLparse
   0.94%  [kernel]                 [k] _raw_spin_lock
   0.79%  mysqld                   [.] row_search_mvcc
   0.77%  mysqld                   [.] buf_page_get_gen
   0.74%  mysqld                   [.] rw_lock_s_lock_func
   0.66%  libc-2.28.so             [.] __memmove_avx_unaligned_erms
   0.65%  mysqld                   [.] my_hash_sort_simple
   0.57%  mysqld                   [.] rec_get_offsets_func
   0.57%  mysqld                   [.] PolicyMutex<TTASEventMutex<GenericPolicy> >::enter
   0.48%  [kernel]                 [k] check_preemption_disabled
   0.47%  libc-2.28.so             [.] __memmove_sse2_unaligned_erms
   0.43%  mysqld                   [.] lock_table
```

Figure 7-16. Perf tool screenshot highlighting deadlock issues.

Based on the figure, the bottleneck seems to be related to deadlock issues. The MySQL error log file shows numerous error logs, with a partial screenshot provided below:

```
TRANSACTION 1065391, ACTIVE 0 sec starting index read
mysql tables in use 1, locked 1
4 lock struct(s), heap size 1136, 3 row lock(s), undo log entries 2
MySQL thread id 140, OS thread handle 140155190490880, query id 10625750 172.17.130.53 wb updating
DELETE FROM sbtest1 WHERE id=32
2024-08-08T11:06:37.313990+08:00 140 [Note] InnoDB: *** (2) HOLDS THE LOCK(S):

RECORD LOCKS space id 23 page no 4 n bits 144 index PRIMARY of table `sbtest`.`sbtest1` trx id 1065391 lock_mode X locks rec but not gap
Record lock, heap no 2 PHYSICAL RECORD: n_fields 6; compact format; info bits 0
 0: len 4; hex 80000001; asc     ;;
 1: len 6; hex 0000001041af; asc     A ;;
 2: len 7; hex 68000001e92efa; asc h    . ;;
 3: len 4; hex 800303c0; asc     ;;
 4: len 30; hex 38363830333335303736392d323036363038393531313132d343836393838; asc 86803350769-20660895111-486988; (total 120 bytes);
 5: len 30; hex 393630343733393383933392d313936323630323837362d333039333134; asc 96047398939-19626028276-309314; (total 60 bytes);

2024-08-08T11:06:37.314173+08:00 140 [Note] InnoDB: *** (2) WAITING FOR THIS LOCK TO BE GRANTED:

RECORD LOCKS space id 23 page no 4 n bits 144 index PRIMARY of table `sbtest`.`sbtest1` trx id 1065391 lock_mode X locks rec but not gap waiting
Record lock, heap no 33 PHYSICAL RECORD: n_fields 6; compact format; info bits 0
 0: len 4; hex 80000020; asc     ;;
 1: len 6; hex 0000001041be; asc     A ;;
 2: len 7; hex 6e0000021d182e; asc n     .;;
 3: len 4; hex 80028822; asc    ";;
 4: len 30; hex 3035353134353333030303332d363837383436353637383332d333433313233; asc 05514530003-68784656783-343123; (total 120 bytes);
 5: len 30; hex 3033353331303739303036 2d303131343636333938333 02d363333303430; asc 03531079006-01146639830-633040; (total 60 bytes);
```

Figure 7-17. Partial screenshot of numerous error logs.

Continuing the analysis of the corresponding code, the specifics are as follows:

```
void Deadlock_notifier::notify(const ut::vector<const trx_t *> &trxs_on_cycle,
                               const trx_t *victim_trx) {
  ut_ad(locksys::owns_exclusive_global_latch());

  start_print();
  const auto n = trxs_on_cycle.size();
```

```
 for (size_t i = 0; i < n; ++i) {
  const trx_t *trx = trxs_on_cycle[i];
  const trx_t *blocked_trx = trxs_on_cycle[0 < i ? i - 1 : n - 1];
  const lock_t *blocking_lock =
     lock_has_to_wait_in_queue(blocked_trx->lock.wait_lock, trx);
  ut_a(blocking_lock);

  print_title(i, "TRANSACTION");
  print(trx, 3000);

  print_title(i, "HOLDS THE LOCK(S)");
  print(blocking_lock);

  print_title(i, "WAITING FOR THIS LOCK TO BE GRANTED");
  print(trx->lock.wait_lock);
 }
 const auto victim_it =
    std::find(trxs_on_cycle.begin(), trxs_on_cycle.end(), victim_trx);
 ut_ad(victim_it != trxs_on_cycle.end());
 const auto victim_pos = std::distance(trxs_on_cycle.begin(), victim_it);
 ut::ostringstream buff;
 buff << "*** WE ROLL BACK TRANSACTION (" << (victim_pos + 1) << ")\n";
 print(buff.str().c_str());
 DBUG_PRINT("ib_lock", ("deadlock detected"));
 ...
 lock_deadlock_found = true;
}
```

From the code analysis, it's clear that deadlocks lead to a substantial amount of log output. The ignored errors observed during testing are connected to these deadlocks. The CATS algorithm helps reduce the number of ignored errors, resulting in fewer log outputs. This issue can be consistently reproduced.

Given this context, several considerations emerge:

1. **Impact on Performance Testing:** The extensive error logs and the resulting disruptions could potentially skew the performance evaluation, leading to inaccurate assessments of the system's capabilities.

2. **Effectiveness of the CATS Algorithm:** The performance improvement of the CATS algorithm may need re-evaluation. If the extensive output of error logs significantly impacts performance, its actual effectiveness may not be as high as initially believed.

Remove all logs from the ***Deadlock_notifier::notify*** function, recompile MySQL, and perform SysBench read-write tests under Pareto distribution. Details are provided in the following figure:

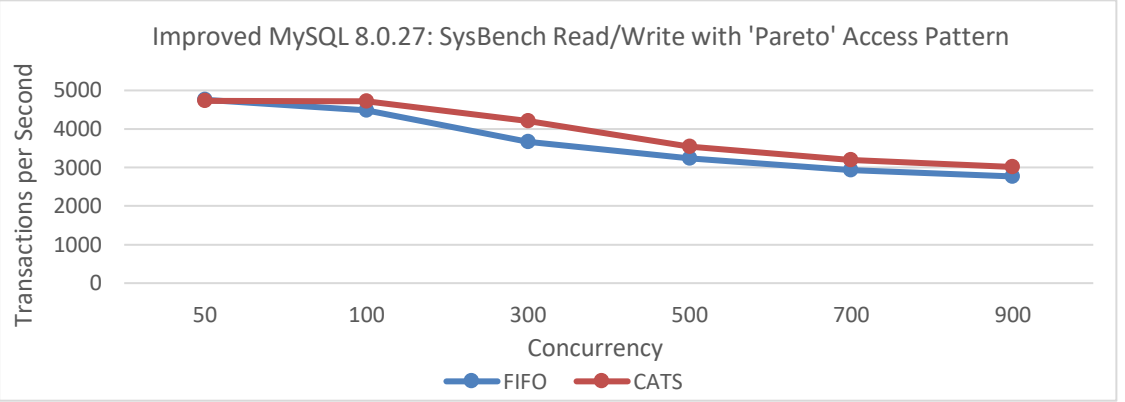


Figure 7-18. Impact of CATS on thoughput at various concurrency levels for improved MySQL 8.0.27 after eliminating interference.

From the figure, it is evident that there has been a significant change in throughput comparison. In scenarios with severe conflicts, the CATS algorithm slightly outperforms the FIFO algorithm, but the difference is minimal and much less pronounced than in previous tests. Note that these tests were conducted on the improved MySQL 8.0.27.

Let's conduct performance comparison tests on the improved MySQL 8.0.32, with deadlock log interference removed, using Pareto distribution.

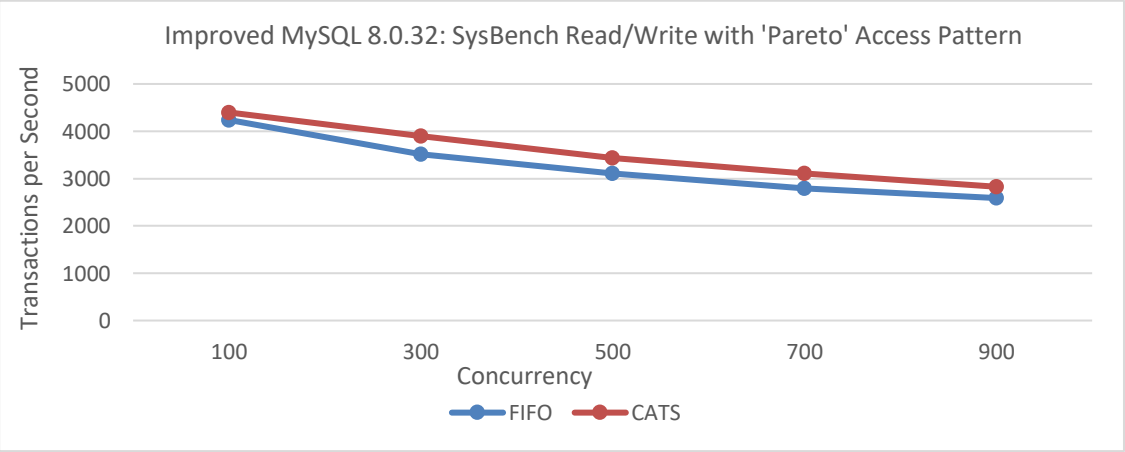


Figure 7-19. Impact of CATS on thoughput at various concurrency levels for improved MySQL 8.0.32 after eliminating interference.

From the figure, it is evident that removing the interference results in only a slight performance difference. This small variation makes it understandable why the severity of FIFO scheduling issues may be difficult to notice. The perceived bias from CATS authors and MySQL officials likely stems from the extensive log output interference caused by deadlocks.

Using the same 32 warehouses as in the CATS algorithm paper, TPC-C tests were conducted at various concurrency levels. MySQL was based on the improved MySQL 8.0.27, and BenchmarkSQL was modified to support 100 concurrent transactions per warehouse.

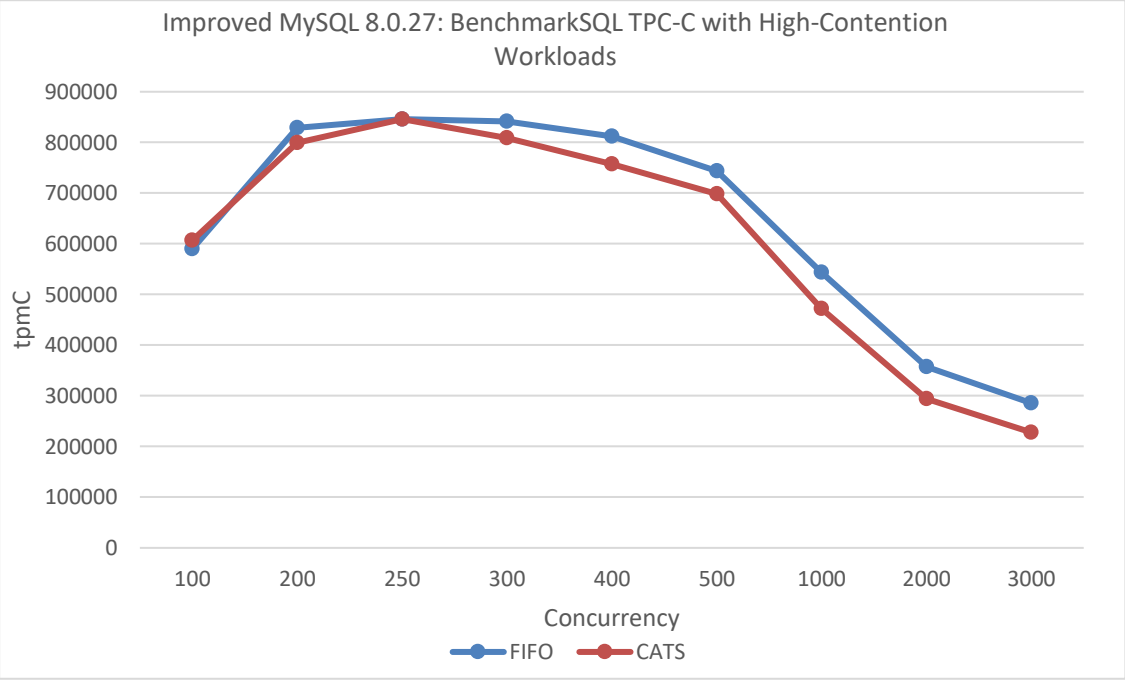


Figure 7-20. Impact of CATS on throughput at different concurrency levels under NUMA after eliminating interference, according to the CATS paper.

From the figure, it's evident that the CATS algorithm performs worse than the FIFO algorithm. To avoid NUMA-related interference, MySQL was bound to NUMA node 0 for a new round of throughput versus concurrency tests.

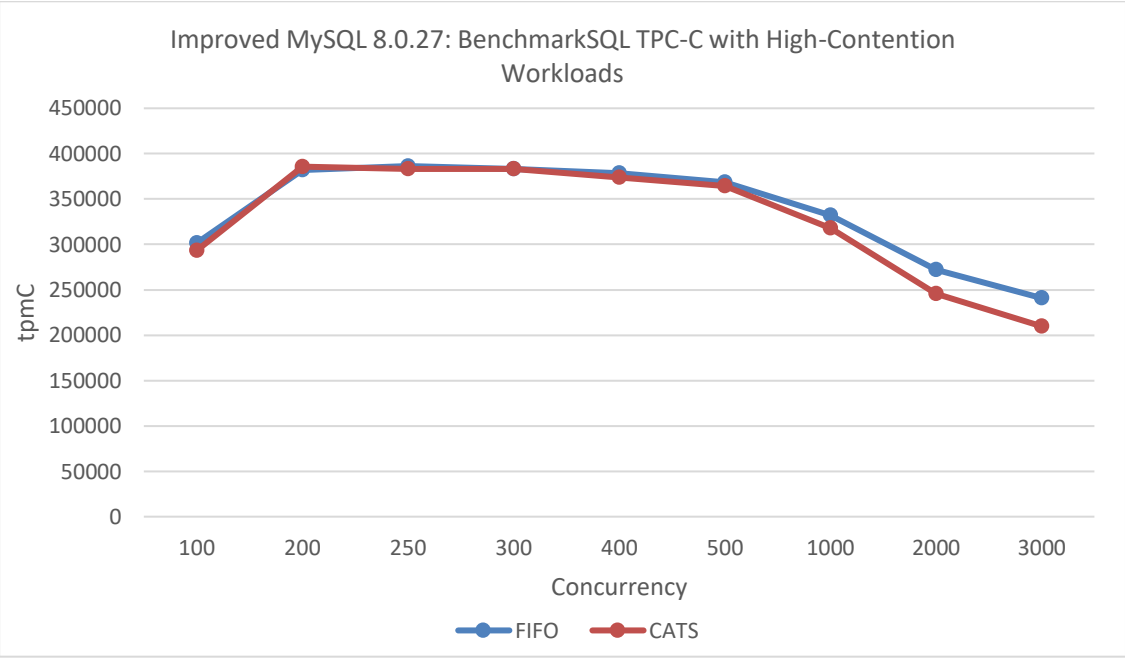


Figure 7-21. Impact of CATS on throughput at different concurrency levels under SMP after eliminating interference, according to the CATS paper.

In this round of testing, the FIFO algorithm continued to outperform the CATS algorithm. The decline in performance of the CATS algorithm in BenchmarkSQL TPC-C testing compared to improvements in SysBench Pareto testing can be attributed to the following reasons:

1. **Additional Overhead**: The CATS algorithm inherently introduces some extra overhead.
2. **NUMA Environment Issues**: The CATS algorithm may not perform optimally in NUMA environments.
3. **Conflict Severity**: The conflict severity in TPC-C testing is less pronounced than in SysBench Pareto testing.
4. **Different Concurrency Scenarios**: SysBench creates concurrency scenarios that differ significantly from those in BenchmarkSQL.

Finally, standard TPC-C testing was performed again with 1000 warehouses at varying concurrency levels. Specific details are shown in the following figure:

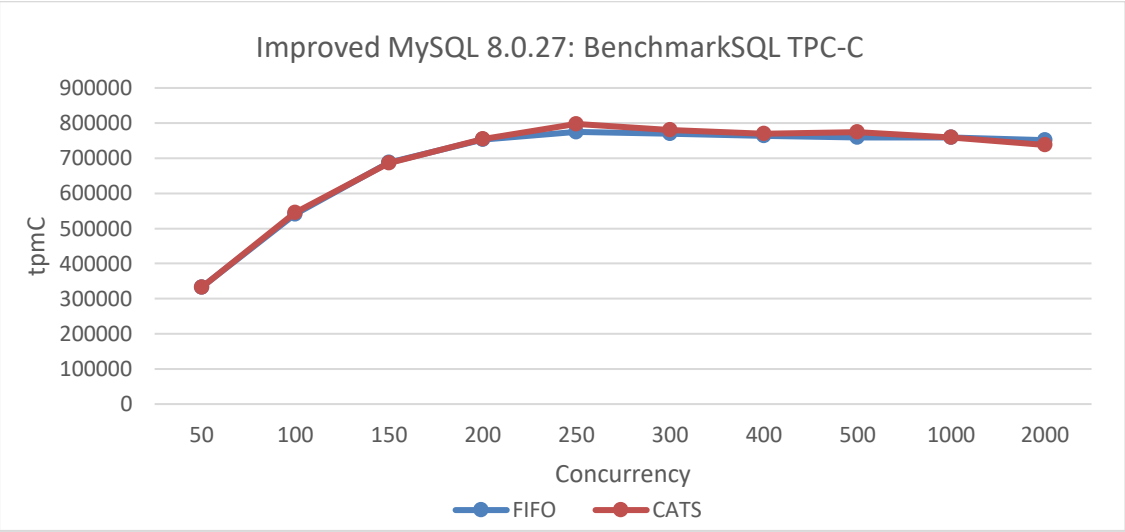


Figure 7-22. Impact of CATS on BenchmarkSQL throughput after eliminating interference.

From the figure, it is evident that there is little difference between the two algorithms in low-conflict scenarios. In other words, the CATS algorithm does not offer significant benefits in situations with fewer conflicts.

Overall, while CATS shows some improvement in Pareto testing, it is less pronounced than expected. The interference from deadlock log outputs during performance testing impacted the results. The CATS algorithm significantly reduces transaction deadlocks, leading to fewer log outputs and less performance degradation compared to the FIFO algorithm. When deadlock logs are suppressed, the difference between these algorithms is minimal, clarifying the confusion surrounding the CATS algorithm's performance.

Database performance testing is inherently complex and error-prone [16]. It cannot be judged by data alone and requires thorough investigation to ensure logical consistency.

7.3 Enhancements in MySQL Execution Plans

7.3.1 Hash Join Implementation in MySQL

As the name suggests, hashing is central to the hash join algorithm. It builds a hash table from one input table and then processes the other table row by row, using the hash table for lookups.

Hash joins are typically faster and are preferred over the block nested loop algorithm used in earlier MySQL versions. The benefits are substantial, as demonstrated by the practical case below.

In MySQL 5.7, which lacks hash join support, the SQL query relies on traditional join methods, resulting in a longer execution time of 3.82 seconds.

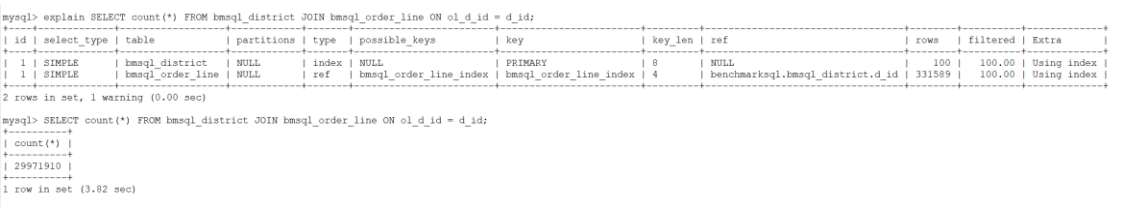


Figure 7-23. Non-hash join performance in MySQL 5.7.

MySQL 8.0 introduced hash join. For the same SQL query, using hash join with hints reduced the execution time to 1.22 seconds, a significant improvement over the 3.82 seconds with traditional methods.

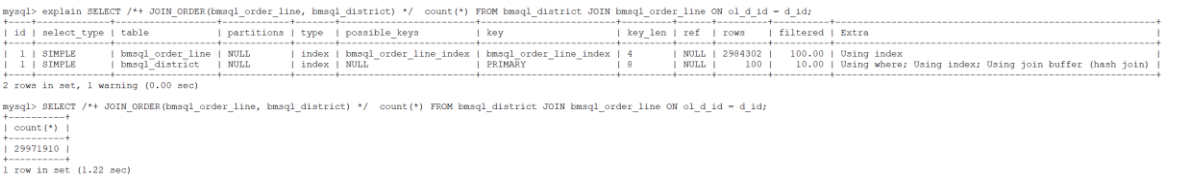


Figure 7-24. Hash join performance in MySQL 8.0.

Notably, hash join in MySQL 8.0 enhances join performance under the following conditions [24]:

1. No index is available
2. The query is I/O-bound
3. A large portion of a table is accessed
4. There are selective conditions across multiple tables
5. Increasing join_buffer_size can further improve performance

The introduction of hash join is a significant feature in MySQL 8.0, offering a promising solution for reducing response times.

7.3.2 Introduction of Hypergraph Algorithm in MySQL

The hypergraph algorithm was introduced in MySQL 8.0 but is currently only available in debug mode. The following git log provides specific implementation details of the hypergraph algorithm.

```
commit b9be77784bf690173522d8db015acf0e72f28f84
Author: Steinar H. Gunderson <steinar.gunderson@oracle.com>
Date:   Wed May 6 16:32:13 2020 +0200

    WL #14070: Hypergraph partitioning algorithm

    Implement DPhyp for hypergraph partitioning, a central component of the join
    optimizer. The algorithm enumerates all possible connected sub-hypergraphs
    of the larger join graph, in a bottom-up fashion. (That is, for a given graph G
    with a subgraphs A and B than can further be partitioned respectively into A1/A2
    and B1/B2, A1 and A2 will both be seen before A, which will in turn be seen
    before S. However, there is no guarantee that B1 or B2 is seen before A.)

    The algorithm is described in the paper "Dynamic Programming Strikes Back" by
    Neumann and Moerkotte. There is a somewhat extended version of the paper
    (that also contains a few corrections) in Moerkotte's treatise "Building Query
    Compilers". Some critical details are still missing, which we've had to fill in
    ourselves. We don't currently implement the extension to generalized
    hypergraphs, but it should be fairly straightforward to do later.

    Since our graphs can never have more than 61 tables, node sets and edge lists
    are implemented using 64-bit bit sets. This allows for a compact representation
    and very fast set manipulation; the algorithm does a fair amount of
    intersections and unions. If we should need extensions to larger graphs later
    (this will require additional heuristics for reducing the search space), we can
    use dynamic bit sets, although at a performance cost.

    This is implemented entirely independently of the server; there are no MySQL
    dependencies, short of some shared header files for bit manipulations. It is
    tested using unit tests and microbenchmarks.

      Change-Id: I7912c09ab69a17e607ee3b8fb2af2bd7602e54ec
```

From the above, it can be seen that the theoretical foundation for the hypergraph algorithm's implementation is detailed in the paper "Dynamic Programming Strikes Back" [64]. This highlights the high level of complexity involved in its implementation.

A cost-based query optimizer is crucial for the overall performance of a database management system, particularly in finding the optimal join order. Building on the efficient ***DPccp*** algorithm, which uses dynamic programming, a new algorithm, ***DPhyp***, is introduced to handle complex join predicates effectively. By modeling the query graph as a hypergraph and analyzing its connected subgraphs, DPhyp improves the optimization of non-inner joins, offering substantial performance gains over previous methods.

With advances in hardware, high-complexity algorithms are becoming practical. Even though some algorithms may not run in polynomial time, modern computers can handle large NP-complete problems efficiently. Dynamic programming techniques, while still exponential, are increasingly viable for moderate instance sizes, often achieving time complexities of O(2^n).

Nevertheless, when using the hypergraph algorithm, the number of tables involved in joins should be kept within reasonable limits to avoid potential performance issues. Performance comparisons were conducted for complex join operations in TPC-C, with and without hypergraph optimization enabled. Detailed results are shown in the following figure:

```
mysql> set optimizer_switch="hypergraph_optimizer=on";
Query OK, 0 rows affected, 1 warning (0.00 sec)

mysql> SELECT count(*) AS low_stock FROM ( SELECT s_w_id, s_i_id, s_quantity FROM bmsql_stock WHERE s_w_id = 5 AND s_quantity < 12
+-----------+
| low_stock |
+-----------+
|         6 |
+-----------+
1 row in set (0.88 sec)


mysql> set optimizer_switch="hypergraph_optimizer=off";
Query OK, 0 rows affected (0.00 sec)

mysql> SELECT count(*) AS low_stock FROM ( SELECT s_w_id, s_i_id, s_quantity FROM bmsql_stock WHERE s_w_id = 5 AND s_quantity < 12
+-----------+
| low_stock |
+-----------+
|         6 |
+-----------+
1 row in set (0.03 sec)
```

Figure 7-25. Effects of hypergraph algorithms on typical TPC-C SQL workloads.

From the figure, it is evident that enabling the hypergraph algorithm results in an execution time of 0.88 seconds, whereas disabling it reduces the time to 0.03 seconds. This demonstrates the significant performance impact of using the hypergraph algorithm. In many cases, the overhead of the hypergraph can be substantial. If MySQL's default execution plan leads to slow performance, the hypergraph algorithm might offer valuable improvements.

Let's further analyze the performance of the hypergraph algorithm by examining the perf flame graph in the following figure.

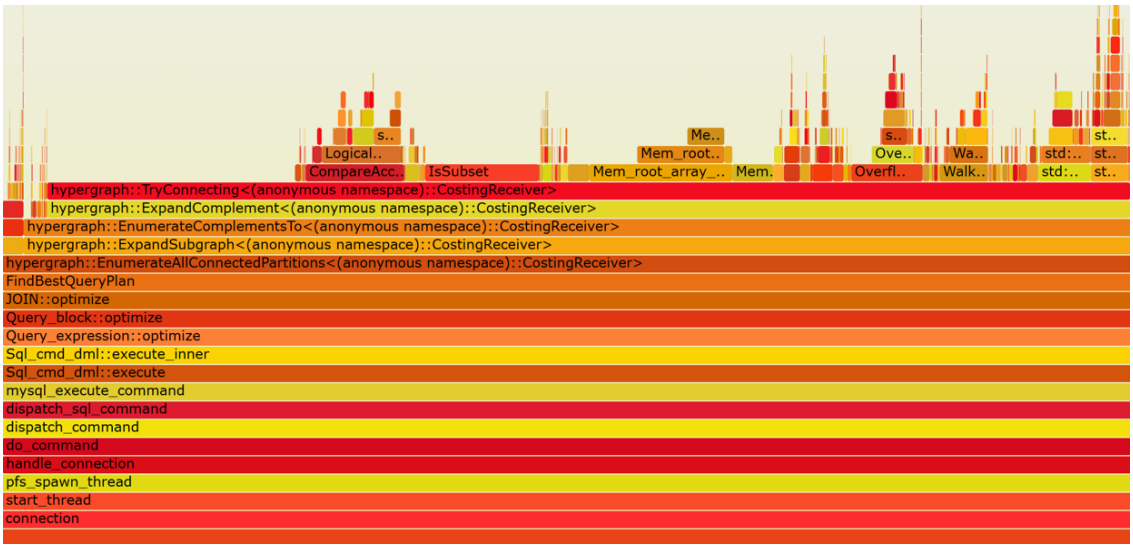


Figure 7-26. A typical flame graph of hypergraph algorithm.

From the figure, it is evident that the hypergraph algorithm (hypergraph*) consumes a significant amount of computation. Currently operating in single-threaded mode, there is substantial potential for optimizing the hypergraph algorithm.

Due to MySQL's absence of query plan caching, constructing optimal execution plans with the hypergraph algorithm is time-consuming, posing challenges for its effective use in production environments.

Notably, AI can also be utilized for optimizing execution plans, as discussed in section 5.20.2.

7.4 Cost Savings with Binlog Compression

Starting from MySQL 8.0.20, binlog compression is supported but disabled by default. It can be enabled with the ***binlog_transaction_compression*** parameter, and the zstd compression level can be adjusted using the ***binlog_transaction_compression_level_zstd*** parameter, with a default level of 3.

Using a Group Replication cluster within the same data center, the impact of binlog compression on TPC-C throughput and concurrency was examined using BenchmarkSQL. Both primary and secondary nodes were configured with the ***binlog_transaction_compression*** parameter. Specific test results are shown in the following figure:

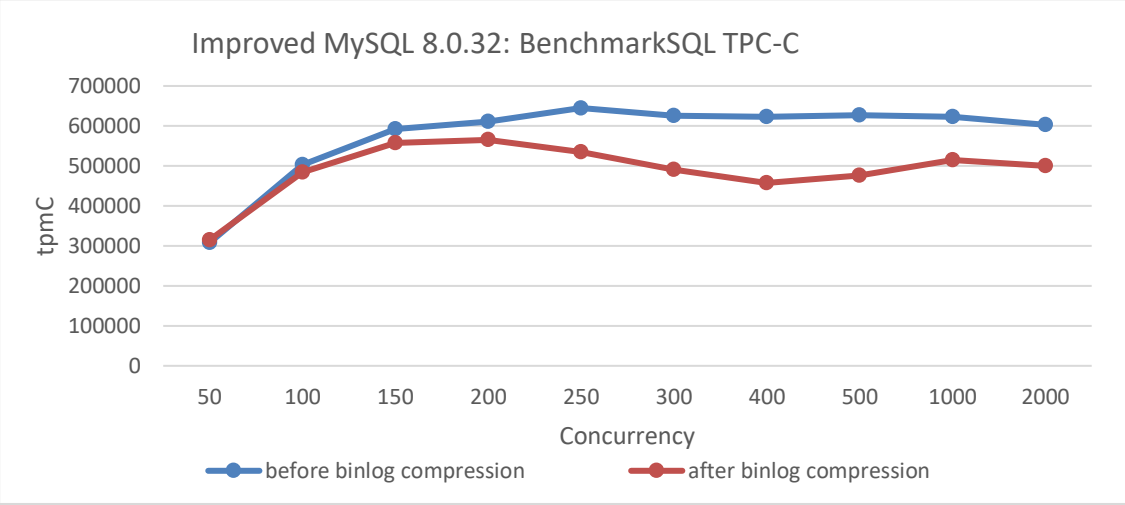


Figure 7-27. Impact of binlog compression on BenchmarkSQL performance.

From the figure, it is evident that enabling binlog compression significantly affects throughput, with noticeable fluctuations.

The next step is to compare binlog sizes before and after compression. Specific details are shown in the following figure:

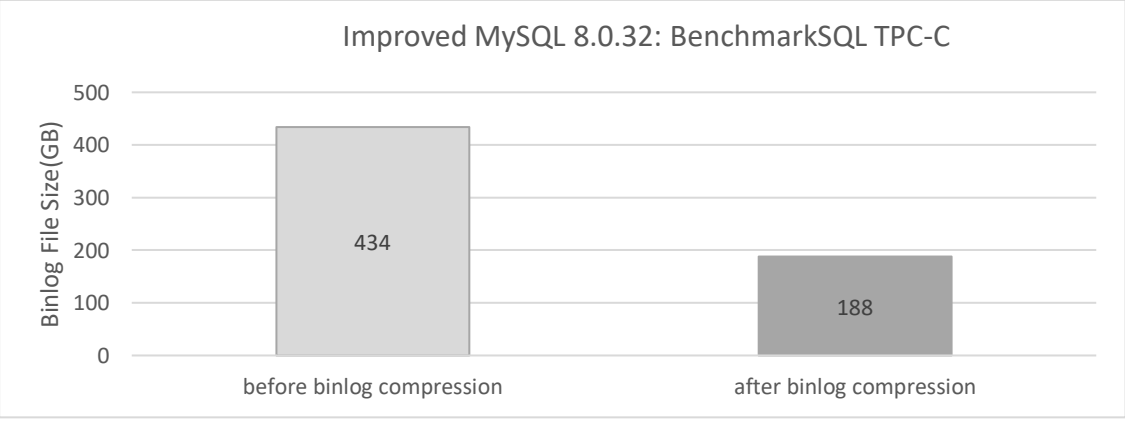


Figure 7-28. Effects of binlog compression after BenchmarkSQL testing.

From the figure, it is evident that binlog compression has a notable positive effect on TPC-C testing. It's worth noting that setting ***binlog_row_image=minimal*** can significantly reduce binlog size, but it has less impact on performance. Specific details are shown in the following figure:

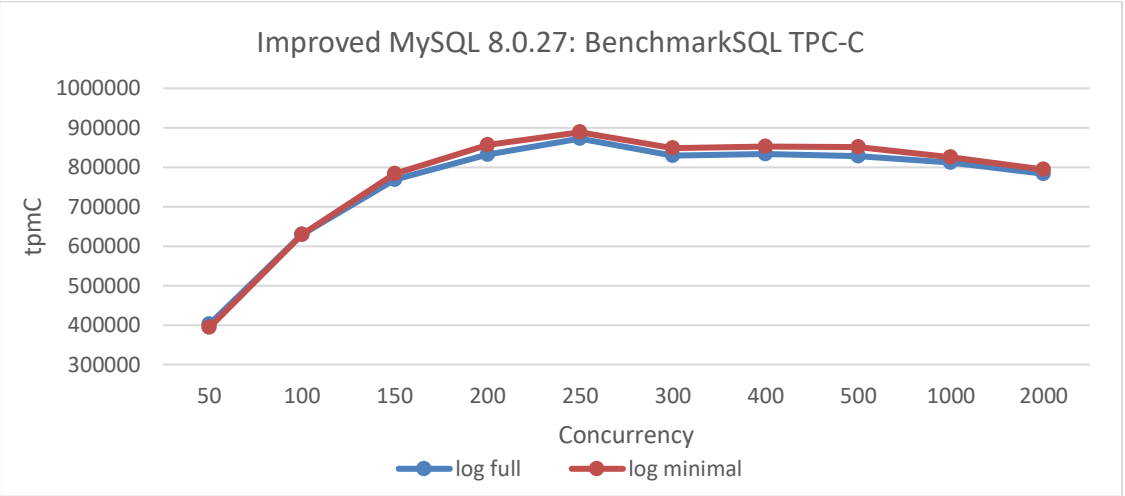


Figure 7-29. Impact of ***binlog_row_image=minimal*** on BenchmarkSQL performance.

Finally, let's examine the comparison of binlog sizes between ***binlog_row_image=minimal*** and ***binlog_row_image=full***. Specific details are shown in the following figure:

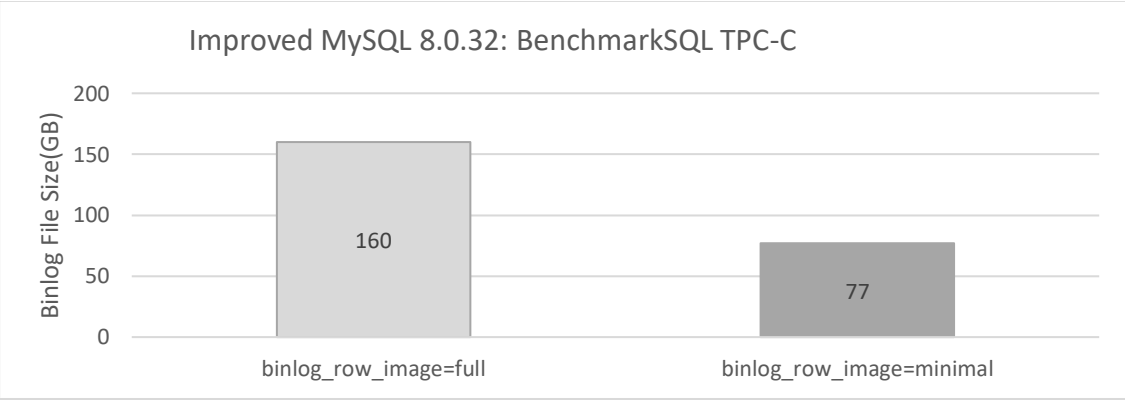


Figure 7-30. Effects of ***binlog_row_image=minimal*** after BenchmarkSQL testing.

From the figure, it can be seen that setting ***binlog_row_image=minimal*** can also significantly reduce the size of binlogs.

Overall, MySQL 8.0 offers effective solutions to address the issue of binlogs consuming substantial I/O space. Users can leverage binlog compression and, where feasible, further reduce binlog size by using ***binlog_row_image=minimal*** to save on storage costs. It’s important to note that the compression ratio can vary across different applications.

References

[1] Zeitz, P. (1999). The art and craft of problem solving. New York: John Wiley.

[2] C. Mohan, D.L. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz. ARIES: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. ACM TODS, 17 (1): 94--162, 1992.

[3] B. Schwartz, P. Zaitsev, V. Tkachenko, J. D. Zawodny, A. Lentz, and D. J. Balling. 2008. High Performance MySQL: Optimization, Backups, Replication and More. O'Reilly Media, USA.

[4] Miguel Correia, Daniel Gómez Ferro, Flavio P Junqueira, and Marco Serafini. 2012. Practical Hardening of {Crash-Tolerant} Systems. In 2012 USENIX Annual Technical Conference (USENIX ATC 12). 453--466.

[5] Tushar Deepak Chandra and Sam Toueg. 1996. Unreliable failure detectors for reliable distributed systems. Journal of the ACM (JACM) 43, 2 (1996), 225--267.

[6] Johnson, R., et al. Scalability of write-ahead logging on multicore and multisocket hardware. VLDBJ, pp. 239--263, 2012.

[7] Collin McCurdy and Jeffrey Vetter. 2010. Memphis: Finding and fixing NUMA-related performance problems on multi-core platforms. In IEEE International Symposium on Performance Analysis of Systems 8 Software (ISPASS'10). 87--96.

[8] Fabien Gaud, Baptiste Lepers, Jeremie Decouchant, Justin Funston, Alexandra Fedorova, and Vivien Quema. 2014. Large Pages May Be Harmful on NUMA Systems. In 2014 USENIX Annual Technical Conference (USENIX ATC 14). USENIX Association, Philadelphia, PA, 231-- 242. https://www.usenix.org/conference/atc14/technical-sessions/ presentation/gaud

[9] Vinson Young, Aamer Jaleel, Evgeny Bolotin, Eiman Ebrahimi, David Nellans, and Villa Oreste. 2018. Combining HW/SW Mechanisms to Improve NUMA Performance of Multi-GPU Systems. In Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture. ACM.

[10] Puya Memarzia, Suprio Ray, and Virendra C. Bhavsar. 2020. The art of efficient in-memory query processing on NUMA systems: A systematic approach. In Proceedings of the IEEE 36th International Conference on Data Engineering (ICDE'20). 781–792.

https://doi.org/10.1109/ICDE48307.2020.00073

[11] T. Kiefer, B. Schlegel, and W. Lehner. Experimental evaluation of NUMA effects on database management systems. In BTW, 2013.

[12] A. Alquraan, H. Takruri, M. Alfatafta, and S. Al-Kiswany. An analysis of network-partitioning failures in cloud systems. In Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation, OSDI'18, pages 51--68, Carlsbad, CA, USA, 2018. USENIX Association.

[13] Adnan Alhomssi and Viktor Leis. 2023. Scalable and Robust Snapshot Isolation for High-Performance Storage Engines. Proc. VLDB Endow. 16, 6 (2023), 1426–1438.

[14] Yi Lu, Xiangyao Yu, Lei Cao, and Samuel Madden. 2020. Aria: a fast and practical deterministic OLTP database. Proceedings of the VLDB Endowment 13, 12 (2020), 2047--2060.

[15] Y. Wang, M. Yu, Y. Hui, F. Zhou, Y. Huang, R. Zhu, et al., "A study of database performance sensitivity to experiment settings", Proceedings of the VLDB Endowment, vol. 15, no. 7, 2022.

[16] M. Raasveldt, P. Holanda, T. Gubner, and H. Muhleisen. Fair Benchmarking Considered Difficult: Common Pitfalls In Database Performance Testing. In 7th International Workshop on Testing Database Systems, DBTest, 2:1--2:6, 2018.

[17] M. Harchol-Balter. Performance Modeling and Design of Computer Systems: Queueing Theory in Action. Cambridge University Press, New York, NY, USA, 1st edition, 2013.

[18] R. Nair and T. Field, "Gapp: A fast profiler for detecting serialization bottlenecks in parallel linux applications", Proceedings of the ACM/SPEC International Conference on Performance Engineering, pp. 257-264, 2020.

[19] Alex Ramirez, Luiz Andre Barroso, Kourosh Gharachorloo, Robert Cohn, Josep Larriba-Pey, P. Geoffrey Lowney, and Mateo Valero. Code layout optimizations for transaction processing workloads. In Proceedings of the 28th Annual International Symposium on Computer Architecture, June 2001.

[20] R. Lavaee Mashhadi, Profile-guided memory layout: Theory and practice, 2018.

[21] Z. Majo and T. R. Gross. Memory System Performance in a NUMA Multicore Multiprocessor. In SYSTOR, 2011.

[22] Hao Xu, Qingsen Wang, Shuang Song, Lizy Kurian John, and Xu Liu. 2019. Can we trust profiling results? Understanding and fixing the inaccuracy in modern profilers. In Proceedings of the ACM International Conference on Supercomputing. 284–295.

[23] T. Mytkowicz, A. Diwan, M. Hauswirth, and P. F. Sweeney. Evaluating the accuracy of Java profilers. In PLDI, pages 187--197. ACM, 2010.

[24] https://dev.mysql.com/doc/refman/8.0/en/

[25] Alibaba Cloud. 2019. OceanBase Did Better than Any Other Database in the TPC-C Benchmark. alibaba-cloud.medium.com.

[26] A. Africa, T. Dolores, M. Lim, L. Miguel and V. Sayoc, "Understanding logical reasoning through computer systems", International Journal of EmergingTrends in Engineering Research, vol. 8, no. 4, pp. 1187-1191, 2020.

[27] B. H. Dowden, Logical reasoning (California State University, Sacramento, CA, 2019).

[28] C. Hong, D. Zhou, M. Yang, C. Kuo, L. Zhang, and L. Zhou. KuaFu: Closing the parallelism gap in database replication. In Proc. ICDE 2013, pages 1186--1195, 2013.

[29] I. Psaroudakis, T. Scheuer, N. May, and A. Ailamaki. Task scheduling for highly concurrent analytical and transactional main-memory workloads. In ADMS Workshop, 2013.

[30] Dai Qin, Angela Demke Brown, and Ashvin Goel. 2017. Scalable replay-based replication for fast databases. Proceedings of the VLDB Endowment (2017).

[31] H. Jung, H. Han, A. D. Fekete, G. Heiser, and H. Y. Yeom. A scalable lock manager for multicores. In SIGMOD, 2013.

[32] Simon, S.: Brewer's CAP Theorem. CS341 Distributed Information Systems, University of Basel (HS2012).

[33] Dice, D., Kogan, A.: Avoiding Scalability Collapse by Restricting Concurrency. Lecture Notes in Computer Science, pp. 363–376. Springer International Publishing (2019). https://doi.org/10.1007/978-3-030-29400-7_26

[34] Harizopoulos, S. and Ailamaki, A. 2003. A case for staged database systems. In Proceedings of the Conference on Innovative Data Systems Research (CIDR). Asilomar, CA. Harizopoulos, S. and Ailamaki, A. 2003. A case for staged database systems. In Proceedings of the Conference on Innovative Data Systems Research (CIDR). Asilomar, CA.

[35] Xiangyao Yu. An evaluation of concurrency control with one thousand cores. PhD thesis, Massachusetts Institute of Technology, 2015.

[36] K. Ren, J. M. Faleiro, and D. J. Abadi. Design principles for scaling multi-core OLTP under high contention. In Proceedings of the 2016 ACM SIGMOD International Conference on Management of Data, 2016.

[37] B. Tian, J. Huang, B. Mozafari, and G. Schoenebeck. Contention-aware lock scheduling for transactional databases. PVLDB, 11(5), 2018.

[38] Jiamin Huang, Barzan Mozafari, Grant Schoenebeck, and Thomas F Wenisch. 2017b. A top-down approach to achieving performance predictability in database systems. In Proceedings of the 2017 ACM International Conference on Management of Data. ACM, 745--758.

[39] Peter Bailis, Alan Fekete, Michael J. Franklin, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. 2014. Coordination avoidance in database systems. Proc. VLDB Endow. 3 (Nov. 2014), 185--196.

[40] Jelena Antic, Georgios Chatzopoulos, Rachid Guerraoui, and Vasileios Trigonakis. 2016. Locking made easy. In Proceedings of the International Middleware Conference (Middleware). 1--14.

[41] Baris Kasikci, Weidong Cui, Xinyang Ge, and Ben Niu. 2017. Lazy Diagnosis of In-Production Concurrency Bugs. In Proceedings of the 26th Symposium on Operating Systems Principles (SOSP).

[42] Guoliang Jin, Linhai Song, Xiaoming Shi, Joel Scherpelz, and Shan Lu. 2012. Understanding and detecting real-world performance bugs. In ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12, Beijing, China - June 11 - 16, 2012. 77–88..

[43] J. Rao, E. J. Shekita, and S. Tata. Using paxos to build a scalable, consistent, and highly available datastore. VLDB, 2011.

[44] Ailidani Ailijiang, Aleksey Charapko, Murat Demirbas, and Tevfik Kosar. 2020. WPaxos: Wide Area Network Flexible Consensus. IEEE Transactions on Parallel and Distributed Systems 31, 1 (2020), 211--223. https://doi.org/10.1109/TPDS.2019.2929793

[45] Alexandre Verbitski, Anurag Gupta, Debanjan Saha, Murali Brahmadesam, Kamal Gupta, Raman Mittal, Sailesh Krishnamurthy, Sandor Maurice, Tengiz Kharatishvili, and Xiaofeng Bao. 2017. Amazon Aurora: Design Considerations for High Throughput Cloud-Native Relational Databases. In ACM Conference on Management of Data (SIGMOD). 1041--1052.

[46] Paweł Olchawa. 2018. MySQL 8.0: New Lock free, scalable WAL design. dev.mysql.com/blog-archive.

[47] Jim Gray, Pat Helland, Patrick O'Neil, and Dennis Shasha. 1996. The dangers of replication and

a solution. In ACM SIGMOD Record, Vol. 25. ACM, 173--182.

[48] Jose M. Faleiro, Daniel Abadi, and Joseph M. Hellerstein. 2017. High Performance Transactions via Early Write Visibility. PVLDB 10, 5 (2017), 613--624.

[49] Martin Kleppmann. Designing Data-Intensive Applications. English. 1 edition. O'Reilly Media, Jan. 2017. ISBN: 978-1-4493-7332-0.

[50] Heidi Howard. Distributed consensus revised. PhD thesis, University of Cambridge, 2019. URL: https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-935.pdf.

[51] Michael Whittaker, Neil Giridharan, Adriana Szekeres, Joseph M. Hellerstein, and Ion Stoica. 2020. Bipartisan Paxos: A Modular State Machine Replication Protocol. arXiv CoRR abs/2003.00331 (2020). https://arxiv.org/abs/2003.0033

[52] Chandra, T.D., Griesemer, R., Redstone, J.: Paxos Made Live: An Engineering Perspective. In: Proceedings of the Twenty-sixth Annual ACM Symposium on Principles of Distributed Computing, New York, USA: ACM, pp. 398–407, (2007).

[53] J. Kirsch and Y. Amir. Paxos for system builders: An overview. In Proceedings of the 2nd Workshop on Large-Scale Distributed Systems and Middleware, page 3. ACM, 2008.

[54] Leslie Lamport. 2016. Paxos made simple. ACM Sigact News 32, 4 (2016).

[55] Dan R. K. Ports, Jialin Li, Vincent Liu, Naveen Kr. Sharma, and Arvind Krishnamurthy. Designing Distributed Systems Using Approximate Synchrony in Data Center Networks. In Proc. USENIX NSDI, 2015.

[56] https://dev.mysql.com/blog-archive/the-new-mysql-thread-pool/

[57] Y. Mao, F. P. Junqueira, and K. Marzullo. Mencius: building efficient replicated state machines for WANs. In Proc. 8th USENIX OSDI, pages 369--384, San Diego, CA, Dec. 2008.

[58] https://docs.percona.com/percona-server/8.0/threadpool.html

[59] C. Millsap, "Thinking Clearly About Performance," Queue, vol. 8, no. 9, pp. 10-20, 2010.

[60] H. Liu. Applying Queuing Theory to Optimizing the Performance of Enterprise Software Applications. In CMG-CONFERENCE-, volume 1, page 457. Computer Measurement Group; 1997, 2006.

[61] H. H. Liu, Software performance and scalability: a quantitative approach, John Wiley Sons, vol.

7, 2011.

[62] Leonidas Galanis, Supiti Buranawatanachoke, Romain Colle, Beno?t Dageville, Karl Dias, Jonathan Klein, Stratos Papadomanolakis, Leng Leng Tan, Venkateshwaran Venkataramani, Yujun Wang, and Graham Wood. 2008. Oracle Database Replay. In Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data. 1159--1170.

[63] Riccardo Mancini, Srinivas Karthik, Bikash Chandra, Vasilis Mageirakos, and Anastasia Ailamaki. 2022. Efficient Massively Parallel Join Optimization for Large Queries. In SIGMOD '22: International Conference on Management of Data, Philadelphia, PA, USA, June 12 - 17, 2022, Zachary Ives, Angela Bonifati, and Amr El Abbadi (Eds.). ACM, 122--135. https://doi.org/10.1145/3514221.3517871

[64] G. Moerkotte and T. Neumann. Dynamic programming strikes back. In Proceedings of the 2008 ACM SIGMOD international conference on Management of data, pages 539--552. ACM, 2008.

[65] G. Woeginger, "Exact algorithms for NP-hard problems: A survey" in Combinatorial Optimization Eureka You Shrink, New York, NY, USA:Springer, vol. 2570, pp. 185-207, 2003.

[66] Suntorn Sae-eung, "Analysis of false cache line sharing effects on multicore cpus", Master's Projects, vol. 01, 2010.

[67] D. Terry. Replicated data consistency explained through baseball, Microsoft Technical Report MSR-TR-2011-137, October 2011. To appear in Communications of the ACM, December 2013.

[68] James Aspnes. 2020. Notes on theory of distributed systems. arXiv preprint arXiv:2001.04235.

[69] Dominik Durner, Viktor Leis, and Thomas Neumann. 2019. On the Impact of Memory Allocation on High-Performance Query Processing. In Proceedings of the 15th International Workshop on Data Management on New Hardware (DaMoN).

[70] https://courses.engr.illinois.edu/cs423/sp2018/slides/13-linux-schedulers.pdf

[71] Anirban Rahut, Abhinav Sharma, Yichen Shen, Ahsanul Haque. 2023. Building and deploying MySQL Raft at Meta. engineering.fb.com

[72] Taipalus T. Database management system performance comparisons: A systematic survey. Published online January 3, 2023. Accessed July 31, 2023. http://arxiv.org/abs/2301.01095

[73] Holger Pirk. 2022. https://co339.pages.doc.ic.ac.uk/decks/Profiling.pdf

[74] ZHAI J D, ZHANG F, LI Q W, et al. Characterizing and optimizing TPC-C workloads on large-scale systems using SSD arrays[J]. Science China Information Sciences, 2016, 59(9): 92104.

[75] M Poke. 2019. Algorithms for High-Performance State-Machine Replication. DOCTORAL DISSERTATION. HELMUT SCHMIDT UNIVERSITY

[76] Ritwik Yadav and Anirban Rahut. 2023. FlexiRaft: Flexible Quorums with Raft. The Conference on Innovative Data Systems Research (CIDR) (2023).

[77] Jung-Sang Ahn, Woon-Hak Kang, Kun Ren, Guogen Zhang, and Sami Ben-Romdhane. 2019. Designing an efficient replicated log store with consensus protocol. In 11th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 19).

[78] Martin Davis. 2000. Engines of Logic: Mathematicians and the Origin of the Computer. W W Norton & Company.

[79] McInerny, D. Q. (2004). Being logical: A guide to good thinking. New York: Random House.

[80] Arunprasad P. Marathe, Shu Lin, Weidong Yu, Kareem El Gebaly, Per-Åke Larson, and Calvin Sun. 2022. Integrating the Orca Optimizer into MySQL. In Proceedings of the 25th International Conference on Extending Database Technology, EDBT 2022, Edinburgh, UK, March 29 - April 1, 2022. OpenProceedings.org, 2:511--2:523.

[81] https://en.wikipedia.org/wiki/

[82] https://www.mydistributed.systems/2021/04/paxos.html

[83] Urbán, P., Défago, X., Schiper, A.: Chasing the FLP impossibility result in a lan or how robust can a fault tolerant server be? In: 20th IEEE Symp. on Reliable Distributed Systems (SRDS), pp. 190–193. New Orleans, USA (2001)

[84] Jean-Pierre Lozi, Baptiste Lepers, Justin Funston, Fabien Gaud, Vivien Quéma, and Alexandra Fedorova. The linux scheduler: A decade of wasted cores. In Proceedings of the Eleventh European Conference on Computer Systems, EuroSys '16, New York, NY, USA, 2016. Association for Computing Machinery.

[85] Qizhe Cai, Shubham Chaudhary, Midhul Vuppalapati, Jaehyun Hwang, and Rachit Agarwal. 2021. Understanding Host Network Stack Overheads. In Proceedings of the 2021 ACM SIGCOMM 2021 Conference (Virtual Event, USA) (SIGCOMM '21). Association for Computing Machinery, New York, NY, USA, 65--77.

[86] Alysson Bessani, Marcel Santos, João Felix, Nuno Neves, and Miguel Correia. 2013. On the efficiency of durable state machine replication. In Proceedings of the 2013 USENIX Annual Technical Conference (USENIX ATC'13). 169--180.

[87] https://www.candtsolution.com/news_events-detail/what-is-the-difference-between-arm-and-x86/

[88] N. Santos and A. Schiper, "Tuning Paxos for high-throughput with batching and pipelining," in 13th International Conference on Distributed Computing and Networking (ICDCN 2012), Jan. 2012.

[89] J. Kończak, N. Santos, T. Zurkowski, P. T. Wojciechowski, and A. Schiper. JPaxos: state machine replication based on the Paxos protocol. Technical report, EPFL, 2011.

[90] TPC-C, Available: http://www.tpc.org/tpcc/

[91] R. N. Avula and C. Zou, "Performance evaluation of TPC-C benchmark on various cloud providers", Proc. 11th IEEE Annu. Ubiquitous Comput. Electron. Mobile Commun. Conf. (UEMCON), pp. 226-233, Oct. 2020.

[92] Guna Prasaad, Alvin Cheung, and Dan Suciu. 2018. Improving High Contention OLTP Performance via Transaction Scheduling. https://arxiv.org/abs/1810.01997v1

[93] Parisa Jalili Marandi, Marco Primi, and Fernando Pedone. 2011. High performance state-machine replication. In DSN. IEEE, 454–465.

[94] Karolos Antoniadis, Rachid Guerraoui, Dahlia Malkhi, and Dragos-Adrian Seredinschi. State machine replication is more expensive than consensus. Technical report, 2018.

[95] Sergey Blagodurov and Alexandra Fedorova. 2011. User-level Scheduling on NUMA Multicore Systems under Linux. In in Proc. of Linux Symposium.

[96] https://queue.acm.org/detail.cfm?ref=rss&id=2513149

[97] Nadav Rotem and Chris Cummins. 2021. Profile Guided Optimization without Profiles: A Machine Learning Approach. arXiv preprint arXiv:2112.14679 (2021).

[98] https://www.ibm.com/docs/en/linux-on-systems?topic=management-linux-scheduling

[99] Yanhua Mao. 2010. State Machine Replication for Wide Area Networks. Doctor of Philosophy in Computer Science. UNIVERSITY OF CALIFORNIA, SAN DIEGO

[100] Zhou, X., Chai, C., Li, G., Sun, J.: Database meets artificial intelligence: A survey. IEEE

Transactions on Knowledge and Data Engineering (2020).

[101] Prince Samuel. 2023. The Role of Logical Reasoning in Programming. medium.com.

[102] Sunny Bains. 2017. Contention-Aware Transaction Scheduling Arriving in InnoDB to Boost Performance. https://dev.mysql.com/blog-archive/

[103] Andres Freund. 2020. Improving Postgres Connection Scalability: Snapshots. techcommunity.microsoft.com.

[104] J. M. Hellerstein, M. Stonebraker, and J. R. Hamilton. Architecture of a database system. Foundations and Trends in Databases. 1(2) pp. 141--259, 2007.

[105] https://github.com/session-replay-tools/tcpcopy

[106] https://github.com/session-replay-tools/cetus

Appendix

Glossary

This book includes numerous terms. Familiarizing yourself with their meanings in advance will help you better understand the issues discussed.

1 NUMA

Non-Uniform Memory Access (NUMA) is a memory design for multiprocessing where access time depends on the memory's location relative to the processor. NUMA extends the scaling of Symmetric Multiprocessing (SMP) architectures. SMP struggles with scalability because it allows only one processor to access memory at a time, leading to bottlenecks. NUMA, now the mainstream server architecture, mitigates these issues. However, binding a MySQL instance to a single NUMA node effectively operates it as a quasi-SMP system, similar to a pure SMP architecture.

2 PGO

Profile-Guided Optimization (PGO) is a compiler technique that uses profiling data to enhance program runtime performance. As a dynamic optimization method, PGO improves code based on runtime information. MySQL code, in theory, benefits significantly from PGO due to its suitability for this optimization approach.

3 Fault Tolerance

Fault tolerance is a system's ability to continue operating correctly despite failures or faults in its components.

4 High Availability

High availability is a network resilience property that ensures an acceptable level of service despite faults and operational challenges.

5 Transaction

In a database management system, a transaction is a logical unit of work, often comprising multiple operations. Transactions not only simplify the complexity of application coding but are also one of the core functionalities of database management systems.

6 Dual One

Binlog is a logical log that records transaction modifications for data replication and disaster recovery in MySQL. In contrast, the redo log is a physical log that captures modifications to data pages, essential for recovering committed data after a crash. Both logs are crucial for MySQL crash recovery. To prevent data loss during unexpected failures, 'dual one' protection is essential, which involves ensuring timely disk flushing for both the binlog and redo log.

7 Replication

Replication relies on the primary server tracking all database changes (updates, deletes, etc.) in its binary log. This log records all events that alter the database's structure or content from server startup. SELECT statements are not recorded, as they do not modify the database. Replication functions by reading events from the binary log on the primary server and processing them on the secondary server. Events are recorded in various formats within the binary log, depending on the type of event.

8 Paxos Algorithm

Paxos is a family of protocols for solving consensus in a network of unreliable or fallible processors.

9 State Machine Replication

In computer science, state machine replication (SMR) is a method for implementing fault-tolerant services by replicating servers and coordinating client interactions with these replicas. In MySQL Group Replication clusters, SMR is employed using the Paxos algorithm.

10 Row-based Binlog

When using row-based logging, the primary writes events to the binary log that detail changes to individual table rows. Replication to the secondary involves copying these row change events to the secondary, a process known as row-based replication.

11 Read Committed Isolation Level

Put simply, the Read Committed isolation level ensures that any data read during a transaction is committed at the time of reading. It prevents the reader from seeing uncommitted or 'dirty' data. However, it doesn’t guarantee that if the transaction reads the same data again, it will be the same; the data can change after being read.

12 Thread Pool

In computer programming, a thread pool is a design pattern used to achieve concurrent execution. It maintains a pool of threads that are ready to execute tasks as they become available. This approach improves performance by reducing the overhead associated with frequently creating and destroying threads. The number of threads in the pool is adjusted based on the program's computing resources, optimizing task execution and resource utilization.

13 OLTP

Online Transaction Processing (OLTP) refers to database systems used in transaction-oriented applications, such as operational systems. These systems are designed to process and respond to user requests in real-time. This is in contrast to Online Analytical Processing (OLAP), which focuses on data analysis rather than transaction processing.

14 Network Latency

Network latency in packet-switched networks is typically measured as round-trip delay time, which includes the latency from source to destination and back. This latency significantly affects the performance of MySQL clusters.

15 Response Time

In computing, response time measures how long a system takes to respond to a service request, indicating the service's responsiveness.

16 AI

Artificial intelligence (AI) enables machines to exhibit intelligent behavior by developing methods and software for perception, learning, and goal-oriented actions. Machine learning (ML), a subset of AI, creates algorithms that learn from data to perform tasks without explicit instructions. Advances

in neural networks have significantly improved performance in many areas.

Deep learning, a prominent ML direction, has greatly enhanced fields like computer vision, speech recognition, and natural language processing, thanks to increased computational power (especially GPUs) and extensive training datasets like ImageNet.

In databases, AI can automate operations, tune performance parameters, and construct SQL statements, enhancing efficiency and functionality. MySQL HeatWave combines transactions, analytics, and machine learning into a single managed service

17 TPC-C

TPC-C is an OLTP benchmark that measures performance based on the rate of New-Order transactions executed per minute. This rate is reported as the tpmC (transactions per minute C), which serves as the benchmark's primary performance metric.

18 Throughput

Throughput measures the number of requests a system processes within a unit of time. Common statistical indicators include:

1. **Transactions Per Second (TPS):** The number of database transactions performed per second.
2. **Queries Per Second (QPS):** The number of database queries performed per second.
3. **tpmC for TPC-C:** The rate of New-Order transactions executed per minute in TPC-C benchmarks.

19 Idempotence

Idempotence is a property of certain operations in computer science where applying the operation multiple times has the same effect as applying it once.

20 Crash Recovery

Transactions in a database can be interrupted unexpectedly. If a failure occurs before all changes in a transaction are completed and committed, the database may become inconsistent and unusable. **Crash recovery** is the process of restoring the database to a consistent and usable state.

21 GTID

Global Transaction Identifiers (GTIDs) simplify replication by uniquely identifying each transaction, eliminating the need to refer to log files or positions when setting up new secondaries or handling failovers. With GTID-based replication, tracking transactions directly ensures easy verification of consistency: if all transactions committed on the primary are present on the secondary, consistency is guaranteed.

GTIDs are maintained between primary and secondary, enabling you to trace any transaction's origin through the binary log. Once a transaction with a specific GTID is committed on a server, duplicate transactions with the same GTID are ignored, ensuring that each transaction is applied only once on the secondary, thus preserving consistency.

In this book, MySQL Server is configured with the following settings before running:

```
gtid_mode=ON
enforce_gtid_consistency=ON
```

This configuration ensures that each MySQL node uses GTID (Global Transaction Identifier), simplifying the process of tracing and troubleshooting issues.

22 Latch vs. Lock: Key Differences

In computer science, a latch or mutex (short for mutual exclusion) is a synchronization primitive that prevents simultaneous access to state by multiple threads. Locking is a technique used to prevent concurrent access to data in a database, ensuring consistent results.

Latch is similar to mutex, while the lock structure in MySQL is as follows:

```
/** Lock struct; protected by lock_sys latches */
struct lock_t {
 /** transaction owning the lock */
 trx_t *trx;

 /** list of the locks of the transaction */
 UT_LIST_NODE_T(lock_t) trx_locks;

 /** Index for a record lock */
 dict_index_t *index;

 /** Hash chain node for a record lock. The link node in a singly
 linked list, used by the hash table. */
```

```
 lock_t *hash;

 union {
  /** Table lock */
  lock_table_t tab_lock;

  /** Record lock */
  lock_rec_t rec_lock;
 };
 ...
```

So, what distinguishes these two concepts? Consider this metaphor [31]:

- A **latch** secures a door, gate, or window in place but does not offer protection against unauthorized access.
- A **lock**, however, restricts entry to those without the key, ensuring security and control.

In MySQL, a global latch is employed to serialize specific processing procedures. For instance, the following is MySQL's description of the role of a global latch.

```
All of the steps above (except 2, as we usually know the page already) are
accomplished with the help of single line:

   locksys::Shard_latches_guard guard{*block_a, *block_b};

And to "stop the world" one can simply x-latch the global latch by using:

   locksys::Global_exclusive_latch_guard guard{};

This class does not expose too many public functions, as the intention is to
rather use friend guard classes, like the Shard_latches_guard demonstrated.
*/
class Latches {
 private:
  using Lock_mutex = ib_mutex_t;
 ...
```

In MySQL, locks are integral to the transaction model, with common types including row locks and table locks. Deadlock detection in MySQL is related to locks, not latches.

Understanding locks is crucial for:

- Implementing large-scale, busy, or highly reliable database applications

- Porting substantial code from other database systems
- Tuning MySQL performance

Familiarity with InnoDB locking and the InnoDB transaction model is essential for these tasks.

It is worth noting that lock objects in MySQL require latch protection to ensure correctness, as seen in the following code.

```
/** Grants a lock to a waiting lock request and releases the waiting
transaction. The caller must hold lock_sys latch for the shard containing the
lock, but not the lock->trx->mutex.
@param[in,out]    lock    waiting lock request
 */
static void lock_grant(lock_t *lock) {
  ut_ad(locksys::owns_lock_shard(lock));
  ut_ad(!trx_mutex_own(lock->trx));

  trx_mutex_enter(lock->trx);

  if (lock_get_mode(lock) == LOCK_AUTO_INC) {
    dict_table_t *table = lock->tab_lock.table;

    if (table->autoinc_trx == lock->trx) {
      ib::error(ER_IB_MSG_637) << "Transaction already had an"
                               << " AUTO-INC lock!";
    } else {
      ut_ad(table->autoinc_trx == nullptr);
      table->autoinc_trx = lock->trx;

      ib_vector_push(lock->trx->lock.autoinc_locks, &lock);
    }
  }
  ...
```

23 MVCC

Multiversion Concurrency Control (MVCC) is a non-locking concurrency control method used by database management systems to enable concurrent access to the database.

24 Doublewrite

The doublewrite buffer is a storage area where InnoDB writes pages from the buffer pool before

writing them to their final positions in the data files. If an unexpected failure occurs during a page write, InnoDB can use the copy in the doublewrite buffer to recover the page during crash recovery.

25 Causality

Causality is the relationship where one event or state (the cause) leads to the production of another event or state (the effect). The cause is partly responsible for the effect, and the effect depends on the cause.

26 Maintaining Transaction Order with replica_preserve_commit_order

In MySQL, the ***replica_preserve_commit_order*** configuration ensure that transactions on secondary databases are committed in the same order as they appear in the relay log. This setting lays the foundation for maintaining the causal relationship between transactions: if transaction A commits before transaction B on the primary, transaction A will also commit before transaction B on the secondary. This prevents inconsistencies where transactions could be read in the reverse order on the secondary.

27 Pipelining

In computing, pipelining, or pipeline processing, is akin to a manufacturing assembly line where different stages of a process are executed concurrently, even if some stages depend on the completion of others. This approach allows multiple operations to proceed simultaneously, improving overall efficiency and reducing processing time.

28 Achieving Strong Consistency in Read/Write Operations

MySQL Group Replication ensures strong consistency with two mechanisms:

Strong Consistency Read (Before Mechanism):

A read-write (RW) transaction waits for all prior transactions to complete before being applied. A read-only (RO) transaction waits for all prior transactions before execution. This guarantees that the transaction reads the latest value, affecting only the read latency.

Strong Consistency Write (After Mechanism):

A RW transaction waits until its changes are applied to all group members. This ensures that once committed on the local member, any subsequent reads on any member will see the committed value

or a more recent one. RO transactions are unaffected.

29 View Change

A view represents the active members of a MySQL Group Replication configuration at a given time. A view change occurs when the group configuration changes, such as when a member joins or leaves, and is communicated to all members simultaneously. Each view is uniquely identified by a view identifier, which is generated with each view change.

30 Network Partition

A network partition divides a computer network into separate subnets, either intentionally for optimization or due to device failures. Distributed software must be partition-tolerant, meaning it should continue to function correctly even when the network is partitioned

31 Transaction Throttling

In MySQL, transaction throttling manages high concurrency by limiting the number of user threads entering the transaction system at once. This approach helps reduce system pressure and enhances stability by controlling the load on the system.

32 Data Structure

In computer science, a data structure is a data organization, and storage format that is usually chosen for efficient access to data. It consists of data values, their relationships, and the operations that can be performed on them, forming an algebraic structure.

33 Algorithms

Design algorithms are instructions for solving specific problems. Logical reasoning is essential, as it involves creating a logically organized sequence of steps to achieve the desired outcome. Programmers must consider control flow, data flow, and interactions to ensure the algorithm is efficient, correct, and meets requirements.

34 Thundering Herd

In computer science, the thundering herd problem occurs when many processes or threads are awakened by an event, but only one can handle it. This leads to excessive competition for resources,

potentially freezing the system.

The pseudo-thundering herd effect is similar but allows some threads or processes to continue processing after activation, reducing the competition.

35 Balanced Replay Speed

For MySQL secondary replay, balanced replay speed is the point at which the secondary matches the primary's speed under normal conditions. If the primary's speed is at or below this threshold, there is minimal lag. However, if the primary's speed exceeds this threshold, the secondary begins to lag in transaction replay progress.

Testing Tool

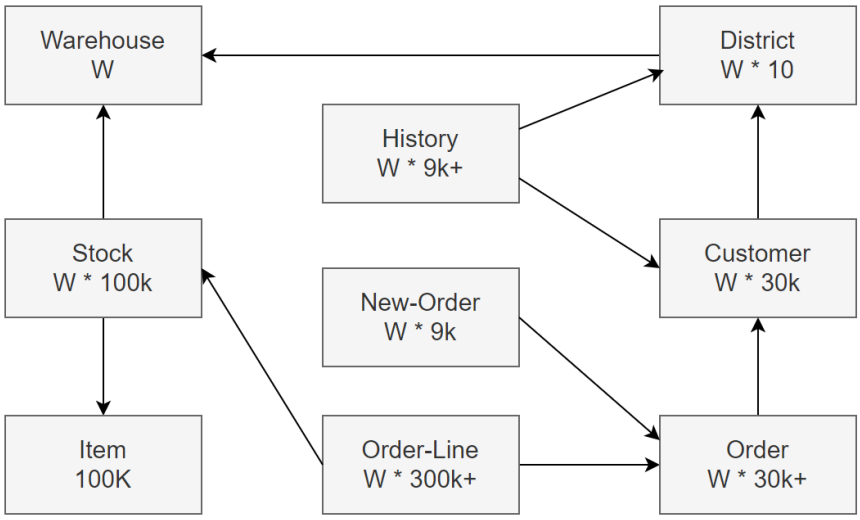
1 SysBench

SysBench is a widely used open-source benchmark tool for testing open-source database management systems (DBMSs). It provides a quick assessment of system performance without the need for elaborate benchmark setups. The tool supports various tests, including conventional read-write and write-only tests, as well as conflict-type tests based on Pareto distribution.

SysBench's main advantages are its simplicity, ease of use, and user-friendliness. However, this simplicity can also be a significant drawback. The oversimplification of the testing process may lead to distortions, potentially missing critical issues and failing to accurately represent online processing capabilities. Therefore, while SysBench offers a straightforward approach to benchmarking, it may not fully capture the complexities of database performance.

2 TPC-C Testing Tool

The TPC-C benchmark, defined by the Transaction Processing Council, is an OLTP test involving 9 tables with 10 foreign key relationships. All tables, except the Item table, scale in cardinality based on the number of warehouses (W) specified during the initial database load.



This schema is used by five different transactions, each creating varied access patterns:

1. **Item:** Read-only.
2. **Warehouse, District, Customer, Stock:** Read/write.
3. **New-Order:** Insert, read, and delete.
4. **Order and Order-Line:** Inserts with time-delayed updates, causing rows to become stale and infrequently read.
5. **History:** Insert-only.

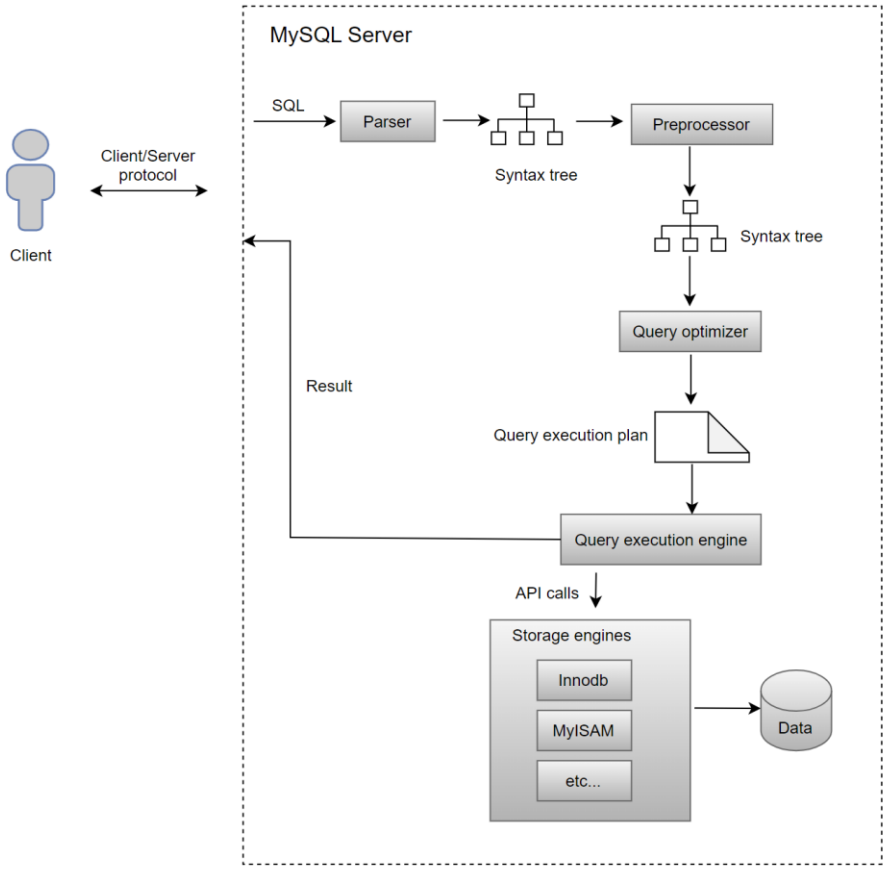
The diverse access patterns of this small schema with a limited number of transactions contribute to TPC-C's ongoing significance as a major database benchmark. In this book, BenchmarkSQL is primarily employed to evaluate TPC-C performance in MySQL.

How MySQL Processes SQL?

In computer science, the request-response or request-reply model is a fundamental communication method in networks. It involves a computer sending a request for data, and another computer responding to that request. Specifically, this pattern involves a request or sending a message to a replier system, which processes the request and returns a response.

MySQL uses the classic request-response model: clients send SQL queries to the MySQL Server,

which processes these queries and sends the responses back to the clients. The following figure illustrates the standard SQL query processing flow in MySQL Server 8.0.



Here's how MySQL Server processes a SQL request with a detailed example. Suppose a user sends the following SQL statement from a MySQL client to MySQL Server:

```
select name from student where id=1;
```

Before executing the SQL query, MySQL Server first parses the SQL statement using the "parser", which performs two essential tasks:

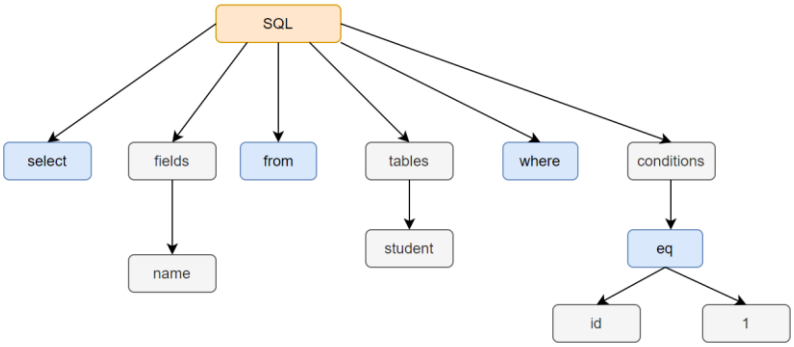
1. **Lexical analysis**

MySQL Server scans the SQL string you input and converts it into tokens, identifying keywords and other elements.

keyword	non-keyword	keyword	non-keyword	keyword	non-keyword	keyword	non-keyword
select	name	from	student	where	id	=	1

2. **Syntax analysis**

Using the tokens from lexical analysis, the syntax parser checks whether the SQL statement adheres to MySQL syntax rules. Upon successful validation, it constructs an SQL syntax tree. This tree structure helps subsequent modules extract key components like SQL types (e.g., SELECT, INSERT), table names, field names, and conditions from the WHERE clause. For example, the SQL statement provided will generate a syntax tree representing these components:



This captures the core of the parsing process, where MySQL Server uses a Bison parser to build the syntax tree.

After parsing, MySQL Server undertakes several steps to optimize query performance before actual execution begins:

1. **Preprocessor**

The preprocessor performs preliminary tasks such as verifying the existence of tables or fields and expanding wildcard characters like `*` in `select *` to include all table columns.

2. **Query Optimizer**

The query optimizer determines the execution plan for the SQL query. This phase includes:

- **Logical Query Rewrites:** Transforming queries into logically equivalent forms.
- **Cost-Based Join Optimization:** Evaluating different join methods to minimize execution cost.

- **Rule-Based Access Path Selection:** Choosing the best data access paths based on predefined rules.

The query optimizer generates the execution plan, which is then used by the query executor engine.

With the execution plan finalized, the query executor engine starts executing the SQL statement, interacting with the storage engine on a record-by-record basis.

Here are two types of execution processes: full table scan and index query.

1. Full Table Scan

Suppose a query is made to retrieve all information about students older than 20:

```
select * from student where age > 20;
```

Since this query condition does not use an index, the optimizer chooses a full table scan by setting the access type to ALL.

```
mysql> explain select * from student where age > 20;
+----+-------------+---------+------------+------+---------------+------+---------+------+------+----------+-------------+
| id | select_type | table   | partitions | type | possible_keys | key  | key_len | ref  | rows | filtered | Extra       |
+----+-------------+---------+------------+------+---------------+------+---------+------+------+----------+-------------+
|  1 | SIMPLE      | student | NULL       | ALL  | NULL          | NULL | NULL    | NULL |    3 |    33.33 | Using where |
+----+-------------+---------+------------+------+---------------+------+---------+------+------+----------+-------------+
```

The execution process for the executor and storage engine is as follows:

1. The Server layer calls the storage engine's full scan interface to start reading records from the table.
2. The executor checks if the age of the retrieved record exceeds 20. Records that meet this condition are dispatched to the network write buffer if there is available space.
3. The executor requests the next record from the storage engine in a loop. Each record is evaluated against the query conditions, and those that meet the criteria are sent to the network write buffer, provided the buffer is not full.
4. Once the storage engine has read all records from the table, it notifies the executor that reading is complete.
5. Upon receiving the completion signal, the executor exits the loop and flushes the query results to the client.

To optimize performance, MySQL minimizes frequent write system calls by checking if the network

buffer is full before sending records to the client. Records are sent only when the buffer is full or when the completion signal is received.

2. Index Query

Consider the following SQL query:

```
select name from student where id < 3 and name like 'wang%';
```

Before executing this query, add a secondary index on the name field in the student table:

```
alter table student add index index_name(name);
```

The execution plan for this SQL query can be viewed using the ***'explain'*** statement, which shows that the query now utilizes the newly created index.

```
mysql>  explain select name from student where id < 3 and name like 'wang%';
+----+-------------+---------+------------+-------+-------------------+------------+---------+------+------+----------+--------------------------+
| id | select_type | table   | partitions | type  | possible_keys     | key        | key_len | ref  | rows | filtered | Extra                    |
+----+-------------+---------+------------+-------+-------------------+------------+---------+------+------+----------+--------------------------+
|  1 | SIMPLE      | student | NULL       | range | PRIMARY,index_name | index_name | 73      | NULL |    1 |    66.67 | Using where; Using index |
+----+-------------+---------+------------+-------+-------------------+------------+---------+------+------+----------+--------------------------+
1 row in set, 1 warning (0.01 sec)
```

The execution process with an index is as follows:

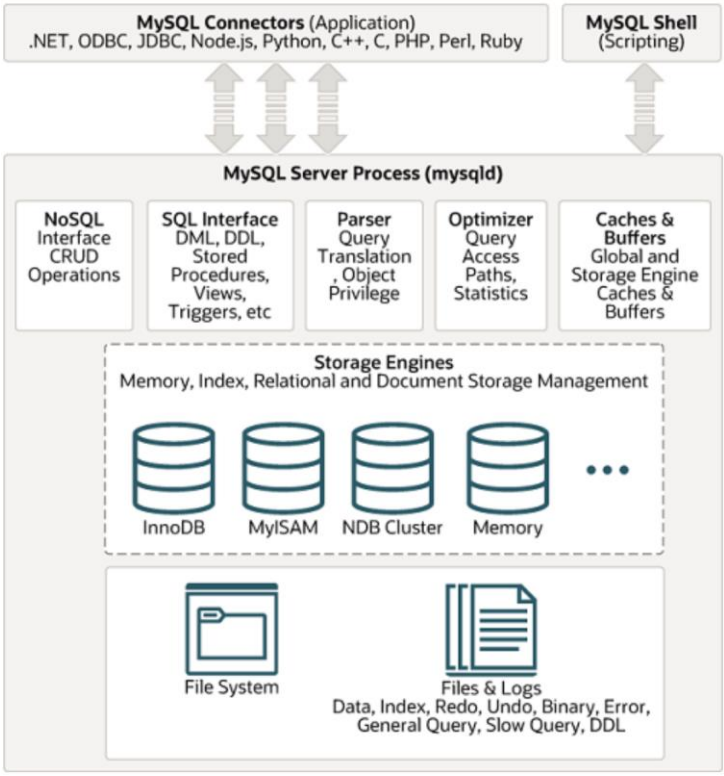
1. The executor requests the storage engine to locate the first index record matching the query condition (e.g., name LIKE 'wang%').
2. The storage engine retrieves and returns the matching index record to the Server layer.
3. The executor checks if the record meets the additional query conditions (e.g., id < 3).

   If conditions are met, the corresponding name is added to the network buffer, unless it is full. If conditions are not met, the executor skips the record and requests the next one from the storage engine.
4. This cycle continues as the executor repeatedly requests and evaluates the next index record that matches the query condition until all relevant index records are processed.
5. Once the storage engine indicates that all relevant index records have been processed, the executor exits the loop and sends the collected results to the client.

Using an index allows the storage engine to quickly locate necessary records, bypassing the need to scan the entire table. In general, this significantly enhances execution efficiency and speeds up the query.

MySQL Architecture

The following picture illustrates the client-server architecture:



MySQL follows the client-server architecture, which divides the system into two main components: Client and Server.

1 Client

1. The client is an application that interacts with the MySQL database server.
2. It can be a standalone application, a web application, or any program requiring a database.
3. The client sends SQL queries to the MySQL server for processing.

2 Server

1. The server is the MySQL database management system responsible for storing, managing, and processing data.
2. It receives SQL queries, processes them, and returns the result sets.
3. It manages data storage, security, and concurrent access for multiple clients.

The client communicates with the server over the network using the MySQL protocol, enabling multiple clients to interact concurrently. Applications use MySQL connectors to connect to the database server. MySQL also provides client tools, such as the terminal-based MySQL client, for direct interaction with the server.

The MySQL database server includes several daemon processes:

1. **SQL Interface**: Provides a standardized interface for applications to interact with the database using SQL queries.
2. **Query Parser**: Analyzes SQL queries to understand their structure and syntax, breaking them down into components for further processing.
3. **Query Optimizer**: Evaluates various execution plans for a given query and selects the most efficient one to enhance performance.

In MySQL, a storage engine is responsible for storage, retrieval, and management of data. MySQL's pluggable storage engine architecture allows selecting different storage engines, such as InnoDB and MyISAM, to meet specific performance and scalability requirements while maintaining a consistent SQL interface.

The file system organizes and stores various file types, including data and index files. MySQL uses log files, such as binary logs and redo logs, to maintain transactional consistency and support recovery mechanisms.

Overall, MySQL follows a client-server architecture where clients send SQL queries to the server for processing. MySQL supports a pluggable storage engine architecture, allowing for the management of data storage and retrieval with various features and performance characteristics.

MySQL Cluster

The most common way to create a fault-tolerant system is to use redundant components, allowing the

system to continue operating if one component fails.

Replication in MySQL copies data from one server (primary) to one or more servers (secondaries), offering several advantages:

1. **Scale-out solutions**: Spreads the load among multiple secondaries to improve performance. All writes and updates occur on the primary server, while reads can occur on secondaries, enhancing read speed.
2. **Data security**: Allows backups on secondaries without affecting primary data by pausing replication.
3. **Analytics**: Permits analysis on secondaries without impacting primary performance.
4. **Long-distance data distribution**: Creates local data copies for remote sites without needing constant access to the primary.

The original synchronization type is one-way asynchronous replication. The advantage of asynchronous replication is that user response time is unaffected by secondaries. However, there is a significant risk of data loss if the primary server fails and secondaries are not fully synchronized.

Semisynchronous replication, in addition to asynchronous replication, requires a commit on the primary server to wait until at least one secondary acknowledges and logs the transaction events. This ensures up-to-date data on secondaries but impacts user response time and introduces high-availability complexities.

Replication introduces significant complexity, as it requires managing multiple servers instead of one. This involves addressing classic distributed systems issues like network partitioning and split-brain scenarios. The main challenge is to coordinate these servers consistently, ensuring they agree on system and data states with each change. Essentially, servers must function as a distributed state machine, either progressing as a single entity or eventually converging to the same state.

MySQL Group Replication provides distributed state machine replication with strong server coordination. Servers within a group automatically maintain a consistent view through a built-in membership service, updating when servers join or leave. If a server fails, the failure detection mechanism alerts the group.

Transaction commits require a majority consensus on the global transaction sequence, ensuring uniform commit or abort decisions. In split-brain scenarios, network partitions halt progress until resolved. The Group Communication System (GCS) protocols ensure consistent data replication

through failure detection, membership management, and ordered message delivery, all powered by the Paxos algorithm as the core communication engine.

MySQL Group Replication can run in single-primary mode with automatic primary election, allowing only one server to accept updates at a time. Alternatively, advanced users can deploy the group in multi-primary mode, where all servers can accept concurrent updates. However, this requires applications to manage the limitations of such deployments.

Generally, simpler clusters are easier to set up but may become more challenging when issues arise. In contrast, more complex clusters are harder to configure initially but offer more elegant solutions for handling problems. The choice of replication mechanism should be based on the specific use case.

Testing Related Materials

Here are the detailed hardware configurations, operating systems, and various test scripts provided to help readers replicate some of the test results presented in this book.

1 Hardware Configurations

Intel(R) Xeon(R) Gold 6238 CPU @ 2.10GHz

cache_alignment : 64

cpu MHz : 3700.000

cache size : 30976 KB

Node details in a NUMA environment on x86 architecture:

```
available: 4 nodes (0-3)
node 0 cpus: 0 4 8 12 16 20 24 28 32 36 40 44 48 52 56 60 64 68 72 76 80 84 88 92 96 100 104 108 112 116 120 124 128 132 136 140 144 148 152 156 160 164 168 172
node 0 size: 96076 MB
node 0 free: 45308 MB
node 1 cpus: 1 5 9 13 17 21 25 29 33 37 41 45 49 53 57 61 65 69 73 77 81 85 89 93 97 101 105 109 113 117 121 125 129 133 137 141 145 149 153 157 161 165 169 173
node 1 size: 96720 MB
node 1 free: 53376 MB
node 2 cpus: 2 6 10 14 18 22 26 30 34 38 42 46 50 54 58 62 66 70 74 78 82 86 90 94 98 102 106 110 114 118 122 126 130 134 138 142 146 150 154 158 162 166 170 174
node 2 size: 96757 MB
node 2 free: 44511 MB
node 3 cpus: 3 7 11 15 19 23 27 31 35 39 43 47 51 55 59 63 67 71 75 79 83 87 91 95 99 103 107 111 115 119 123 127 131 135 139 143 147 151 155 159 163 167 171 175
node 3 size: 96747 MB
node 3 free: 86375 MB
node distances:
node   0   1   2   3
  0:  10  21  21  21
  1:  21  10  21  21
  2:  21  21  10  21
  3:  21  21  21  10
```

The disks are all NVMe SSDs.

Unless otherwise specified, most tests are conducted under these hardware conditions.

2 Operating System

The operating system kernel version is Linux 5.19.8.

3 BenchmarkSQL Script

The script for creating TPC-C tables can be found at the following address:

https://github.com/advancedmysql/mysql_8.0.27/blob/main/tableCreates.sql_base_for_test

The script for creating relevant indexes can be found at the following address:

https://github.com/advancedmysql/mysql_8.0.27/blob/main/indexCreates.sql_base_for_test

The main configuration script for BenchmarkSQL testing is as follows:

```
warehouses=1000
loadWorkers=100
terminals=300
warehouses-begin=1
warehouses-end=1000
//To run specified transactions per terminal- runMins must equal zero
runTxnsPerTerminal=0
//To run for specified minutes- runTxnsPerTerminal must equal zero
runMins=5
//Number of total transactions per minute
limitTxnsPerMin=0
//Set to true to run in 4.x compatible mode. Set to false to use the
//entire configured database evenly.
terminalWarehouseFixed=true
//The following five values must add up to 100
//The default percentages of 45, 43, 4, 4 & 4 match the TPC-C spec
newOrderWeight=45
paymentWeight=43
orderStatusWeight=4
deliveryWeight=4
stockLevelWeight=4
```

From the figure, it can be seen that there are typically 1000 warehouses, and each test lasts for 5 minutes.

4 SysBench Script

SysBench testing is quite simple; here are just the key parameters.

SysBench test parameters: -table_size=1000000 --tables=1

For tests with low contention, use the parameter --rand-type=uniform; for tests with high contention, use the parameter --rand-type=pareto.

5 Using the tpcc-mysql Script for Benchmarking

Download link for the tpcc-mysql improvement tool: http://www.anysql.net/

```
tpccmysql -- Percona TPCC MySQL Utility for Large Warehouse and Long Running

************************************
*** ###easy### TPC-C Load Generator ***
************************************
option h with value '10.233.165.235'
option N (New Order Target) with value '600000'
option P with value '8001'
......
  12, trx: 56744, 95%: 18.376, 99%: 22.755, max_rt: 41.029, 56748|39.021, 5675|9.035, 
  18, trx: 56062, 95%: 19.020, 99%: 25.842, max_rt: 58.534, 56059|60.499, 5605|22.664,
  24, trx: 55846, 95%: 19.728, 99%: 37.671, max_rt: 69.946, 55812|59.985, 5586|45.156,
  30, trx: 55208, 95%: 19.977, 99%: 40.671, max_rt: 68.655, 55199|57.416, 5509|44.571,
  36, trx: 54229, 95%: 24.181, 99%: 51.862, max_rt: 77.793, 54204|71.890, 5434|55.109,
  42, trx: 56534, 95%: 18.454, 99%: 28.947, max_rt: 69.159, 56535|70.094, 5654|46.345,
  48, trx: 56360, 95%: 18.476, 99%: 32.425, max_rt: 68.919, 56332|72.779, 5636|45.188,
```

How to use the tpcc-mysql tool?

Assuming the username is xxx and the password is yyy, the steps are as follows:

The data loading command is as follows:

```
./tpcc_load -h127.0.0.1 -d tpcc200 -u xxx -p "yyy" -P 3306 -w 200
```

The testing command is as follows:

```
./tpcc_start -h127.0.0.1 -P 3306 -d tpcc200 -u xxx -p "yyy" -w 200 -c 100 -r 0 -l 60 -F 1
```

6 Configuration Parameters

Due to numerous tests, only typical configurations are listed here. Special configurations require corresponding parameter modifications.

For specific details on the typical configuration parameters for a standalone MySQL instance, please refer to the following address:

```
https://github.com/advancedmysql/mysql_8.0.27/blob/main/my.cnf_base_for_test
```

It should be noted that by default, the tests are conducted under the ***Read Committed*** transaction

isolation level, with ***binary logging*** enabled, 'dual one' configuration, and ***doublewrite*** enabled.

For Group Replication, The primary server configuration parameters in the native MySQL version are as follows:

```
# for mgr
plugin_load_add='group_replication.so'
enforce-gtid-consistency
gtid-mode=on
loose-group_replication_member_expel_timeout=3
loose-group_replication_start_on_boot= OFF
loose-group_replication_group_name="aaaaaaaa-aaaa-aaaa-aaaa-baaaaaaaaaab"
loose-group_replication_local_address=127.0.0.1:63318
loose-group_replication_group_seeds= "127.0.0.1:63318,127.0.0.1:53318,127.0.0.1:43318"
loose-group_replication_member_weight=50
loose-group_replication_flow_control_mode=disabled

slave_parallel_workers=256
slave_parallel_type=LOGICAL_CLOCK
slave_preserve_commit_order=on
```

It should be noted that readers should modify the IP addresses to match their specific environment.

For Group Replication, the configuration parameters for the secondary server in the native MySQL version are as follows:

```
# for mgr
plugin_load_add='group_replication.so'
enforce-gtid-consistency
gtid-mode=on
loose-group_replication_member_expel_timeout=3
loose-group_replication_start_on_boot= OFF
loose-group_replication_group_name="aaaaaaaa-aaaa-aaaa-aaaa-baaaaaaaaaab"
loose-group_replication_local_address=127.0.0.1:53318
loose-group_replication_group_seeds= "127.0.0.1:63318,127.0.0.1:53318,127.0.0.1:43318"
loose-group_replication_member_weight=50
loose-group_replication_flow_control_mode=disabled

slave_parallel_workers=256
slave_parallel_type=LOGICAL_CLOCK
slave_preserve_commit_order=on
```

Regarding the improved Group Replication, the configuration parameters for the primary server are as follows:

```
# for mgr
plugin_load_add='group_replication.so'
enforce-gtid-consistency
gtid-mode=on
loose-group_replication_member_expel_timeout=3
loose-group_replication_start_on_boot= OFF
loose-group_replication_group_name="aaaaaaaa-aaaa-aaaa-aaaa-baaaaaaaaaab"
loose-group_replication_local_address=127.0.0.1:63318
loose-group_replication_group_seeds= "127.0.0.1:63318,127.0.0.1:53318,127.0.0.1:43318"
loose-group_replication_member_weight=50
loose-group_replication_applier_batch_size_threshold=10000
loose-group_replication_single_primary_fast_mode=1
loose-group_replication_flow_control_mode=disabled
loose-group_replication_broadcast_gtid_executed_period=1000

slave_parallel_workers=256
slave_parallel_type=LOGICAL_CLOCK
slave_preserve_commit_order=on
```

The parameter ***group_replication_single_primary_fast_mode***=1 disables the traditional database certification mode.

For the improved Group Replication, the configuration parameters for the secondary server are as follows:

```
# for mgr
plugin_load_add='group_replication.so'
enforce-gtid-consistency
gtid-mode=on
loose-group_replication_member_expel_timeout=3
loose-group_replication_start_on_boot= OFF
loose-group_replication_group_name="aaaaaaaa-aaaa-aaaa-aaaa-baaaaaaaaaab"
loose-group_replication_local_address=127.0.0.1:53318
loose-group_replication_group_seeds= "127.0.0.1:63318,127.0.0.1:53318,127.0.0.1:43318"
loose-group_replication_member_weight=50
loose-group_replication_applier_batch_size_threshold=10000
loose-group_replication_single_primary_fast_mode=1
loose-group_replication_flow_control_mode=disabled
loose-group_replication_broadcast_gtid_executed_period=1000

slave_parallel_workers=256
slave_parallel_type=LOGICAL_CLOCK
slave_preserve_commit_order=on
```

The details related to semisynchronous replication can be found at the following address:

https://github.com/advancedmysql/mysql_8.0.27/blob/main/semisynchronous.txt

7 Source Code Repository

The patch address for "Percona Server for MySQL 8.0.27-18" is as follows:

https://github.com/advancedmysql/mysql_8.0.27/blob/main/book_8.0.27_single.patch

Please note that this patch is focused on optimizing a MySQL standalone instance. The patch for clusters will be open-sourced in subsequent phases.

For a MySQL standalone instance, the patch includes optimizations such as MVCC ReadView improvements, binlog group commit enhancements, and query execution plan optimizations. For cluster versions, the patch adds optimizations for Group Replication, MySQL secondary replay, along with MVCC ReadView, binlog group commit, and query execution plan improvements, among other enhancements.

About the Author

In earlier years, Bin Wang worked at an internet company focused on developing high-performance computing and high-concurrency systems. He also contributed to open-source projects like TCPCopy [105] and MySQL Proxy [106], gaining valuable experience in problem-solving, particularly in logical thinking.

After leaving the internet company, he concentrated on MySQL-related development, successfully contributing to projects such as Group Replication, secondary replay, InnoDB storage engines, and query optimization. He has accumulated extensive experience in problem-solving within the MySQL domain.