



PDF Download
3722212.3724457.pdf
09 February 2026
Total Citations: 1
Total Downloads: 999

Latest updates: <https://dl.acm.org/doi/10.1145/3722212.3724457>

RESEARCH-ARTICLE

TXSQL: Lock Optimizations Towards High Contented Workloads

DONGHUI WANG, Tencent, Shenzhen, Guangdong, China

YUXING CHEN, Tencent, Shenzhen, Guangdong, China

CHENGYAO JIANG, Tencent, Shenzhen, Guangdong, China

ANQUN PAN, Tencent, Shenzhen, Guangdong, China

WEI JIANG, Tencent, Shenzhen, Guangdong, China

SONGLI WANG, Tencent, Shenzhen, Guangdong, China

[View all](#)

[Open Access Support](#) provided by:

[Tencent](#)

[Renmin University of China](#)

Published: 22 June 2025

[Citation in BibTeX format](#)

SIGMOD/PODS '25: International
Conference on Management of Data
June 22 - 27, 2025
Berlin, Germany

Conference Sponsors:
[SIGMOD](#)

TXSQL: Lock Optimizations Towards High Contented Workloads

Donghui Wang
Yuxing Chen*
Chengyao Jiang
Anqun Pan
Tencent Inc.
Shenzhen, China
{dorseywang,axingguchen,idavidjiang,
aaronpan}@tencent.com

Wei Jiang
Songli Wang
Hailin Lei
Chong Zhu
Lixiong Zheng
Tencent Inc.
Shenzhen, China
{vitalejiang,stanleewang,harlylei,
teddyzhu,paterzheng}@tencent.com

Wei Lu
Yunpeng Chai
Feng Zhang
Xiaoyong Du
Renmin University of China
Beijing, China
{lu-wei,ypchai,fengzhang,
duyong}@ruc.edu.cn

Abstract

Two-phase locking (2PL) is a fundamental and widely used concurrency control protocol. It regulates concurrent access to database data by following a specific sequence of acquiring and releasing locks during transaction execution, thereby ensuring transaction isolation. However, in strict 2PL, transactions must wait for conflicting transactions to commit and release their locks, which reduces concurrency and system throughput. We have observed this issue is exacerbated in high-contented workloads at Tencent, where lock contention can severely degrade system performance. While existing optimizations demonstrate some effectiveness in high-contention scenarios, their performance remains insufficient, as they suffer from lock contention and waiting in hotspot access.

This paper presents optimizations in lock management implemented in Tencent's database, TXSQL, with a particular focus on high-contention scenarios. First, we discuss our motivations and the journey toward general lock optimization, which includes lightweight lock management, a copy-free active transaction list, and queue locking mechanisms that effectively enhance concurrency. Second, we introduce a hotspot-aware approach that enables certain highly conflicting transactions to switch to a group locking method, which groups conflicting transactions at a specific hotspot, allowing them to execute serially in an uncommitted state within a conflict group without the need for locking, thereby reducing lock contention. Our evaluation shows that under high-contented workloads, TXSQL achieves performance improvements of up to 6.5x and up to 22.3x compared to state-of-the-art methods and systems, respectively.

CCS Concepts

• Information systems → Database transaction processing.

Keywords

Transaction; Two-phase locking; High-contented workloads

*Yuxing Chen is the corresponding author.



This work is licensed under a Creative Commons Attribution 4.0 International License.
SIGMOD-Companion '25, Berlin, Germany
© 2025 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-1564-8/2025/06
<https://doi.org/10.1145/3722212.3724457>

ACM Reference Format:

Donghui Wang, Yuxing Chen, Chengyao Jiang, Anqun Pan, Wei Jiang, Songli Wang, Hailin Lei, Chong Zhu, Lixiong Zheng, Wei Lu, Yunpeng Chai, Feng Zhang, and Xiaoyong Du. 2025. TXSQL: Lock Optimizations Towards High Contented Workloads. In *Companion of the 2025 International Conference on Management of Data (SIGMOD-Companion '25)*, June 22–27, 2025, Berlin, Germany. ACM, New York, NY, USA, 14 pages. <https://doi.org/10.1145/3722212.3724457>

1 Introduction

In Tencent Cloud [13]'s Online Transaction Processing (OLTP) scenarios, databases typically experience low request loads most of the time. During these periods, the volume of user requests is below the system's processing capacity. However, there are occasional brief intervals characterized by sudden spikes in request traffic. For instance, in e-commerce, best-selling products or major promotional events can lead to surges in user visits and transaction volumes. These sudden spikes are often regarded as highly contented workloads, where the most frequently accessed data typically becomes a *hotspot*. Although these workload transactions are typically short [20, 36, 40, 64, 76, 77], the system's efforts to maintain transaction isolation result in numerous lock requests and contention, which negatively impacts performance [4, 33, 38].

Pessimistic concurrency control protocols, such as two-phase locking (2PL) [6], are more effective for managing high-conflict transactions [45, 54]. However, traditional 2PL protocols often struggle to address scenarios involving hotspots (e.g., [5, 31]). Consequently, several enhancement strategies have been proposed to mitigate these limitations. An intuitive approach is to reduce the duration of lock holding. For instance, QURO [69] introduces a commit-time-update mechanism that allows transactions to acquire locks later, thereby minimizing lock holding time. Escrow [49] is a locking mechanism that temporarily stores the access rights to resources through a third-party intermediary to reduce lock contentions. Similarly, Bamboo [28] proposes a violation of the 2PL protocol by releasing locks earlier to achieve the same goal. While these methods demonstrate some effectiveness in conflicted workloads, they cannot eliminate lock acquisitions and contention during hotspot access. In contrast, deterministic protocols (e.g., [42, 62]) can effectively eliminate locking by scheduling transactions in a conflict-free manner. However, they face limitations due to scheduling overhead and the requirement for pre-acquisition of read-write sets, which are generally considered impractical [21].

Most academic implementations of these novel protocols are found in in-memory database prototypes (e.g., DBX1000 [72]), while industry practitioners tend to be cautious about modifying concurrency control protocols. For example, systems like MySQL and SQL Server rely on traditional 2PL and multi-version concurrency control (MVCC) [7], or a combination of both. Also, rather than altering the transaction layer to mitigate hotspot workloads, they often focus on modifying the application layer. For instance, most of our Tencent Cloud database instances implement request restrictions at the application layer to prevent sudden spikes in hotspot requests (for further details, see Section 4.6). Additionally, some databases employ proactive hotspot identification and SQL rewriting [29] at the application layer to address ad-hoc high-contented workloads (e.g., [50]). However, these implementations may not necessarily represent the most effective solutions for managing hotspot access in commercial databases, and in certain dynamic scenarios, they may not be applicable at all.

In certain high-contented workloads at Tencent, we have found that request restrictions can effectively mitigate excessive lock contention. However, this approach is less effective in dynamic hotspot situations (see Section 2.3), where transactions concentrate on updating one or a few data items. In these cases, executing transactions serially may yield better performance by eliminating lock contention and associated overhead. To offer a practical solution for managing hotspots, this paper provides insights into lock optimization within Tencent Database TXSQL [14]. First, we discuss our motivations and the overarching lock optimization process, including lightweight lock management, a lock-free active transaction list, and a queue lock mechanism. These optimizations enhance concurrency and improve performance in typical high-contention scenarios. Secondly, we place particular emphasis on its optimizations for hotspot situations. Specifically, TXSQL is designed to meet the following two criteria for concurrency control when addressing hotspots: (1) the protocol is workload-agnostic and does not interfere with existing application logic; and (2) the protocol maintains performance when concurrency increases.

Inspired by the benefits of deterministic protocols in scheduling high-conflicted transactions, we propose a group locking mechanism within the 2PL protocol framework for managing hotspots. Unlike traditional 2PL, which treats all data uniformly, our approach distinguishes between hotspot and non-hotspot data. We group transactions updating hotspot data, allowing them to proceed without locking, as long as they adhere to a total order. When there are no hotspot updates, TXSQL reverts to traditional 2PL, ensuring minimal impact on overall throughput. Our main contributions are as follows:

- We provide motivation, optimizations, and insights in TXSQL to address lock conflict issues.
- When involving hotspot data, we propose a group locking mechanism that groups hotspot data accesses and executes them serially without locking. Also, we describe its correctness in terms of deadlock, rollback, and failure recovery.
- Compared to state-of-the-art methods and systems, evaluation shows that our approach achieves performance improvements of up to 6.5x and 22.3x, respectively.

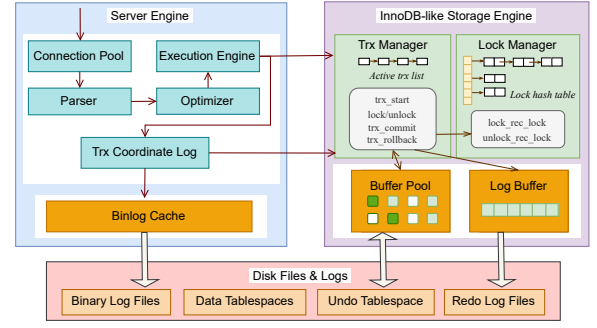


Figure 1: The core architecture of TXSQL.

2 Preliminary

This section introduces the basic 2PL design, as well as TXSQL's system architecture and its applications.

2.1 Two-Phase Locking (2PL)

The 2PL protocol, widely implemented in many database management systems, is designed to prevent data conflicts between concurrent transactions. Transaction execution is divided into two distinct phases: the growing phase and the shrinking phase. In the growing phase, a transaction can acquire locks and access data but is prohibited from releasing any locks. In the shrinking phase, a transaction may release locks but cannot acquire any new locks. The 2PL ensures transactions do not conflict with one another during execution, thereby maintaining transaction isolation.

Mutual Exclusion (Mutex) is a synchronization mechanism used in multithreaded programming, designed to prevent multiple threads from simultaneously accessing shared resources, thereby avoiding data races and inconsistencies. A mutex ensures that only one thread can access a specific resource or code segment at any given time. It is commonly utilized in 2PL lock management.

2.2 Transaction Execution Workflow

TXSQL [15], an open-source MySQL branch maintained by Tencent Cloud, is fully compatible with MySQL's syntax and APIs. Figure 1 depicts the core system architecture of TXSQL. The primary optimizations discussed in this paper focus on the transaction manager and the lock manager at the Storage layer. The transaction manager is responsible for the lifecycle of transactions, including their initiation, execution, commit, and rollback. It ensures the atomicity of transactions, meaning that they either complete successfully in their entirety or do not execute at all. In contrast, the lock manager implements a locking mechanism to regulate access to database resources during transaction execution, such as rows and tables.

When a transaction initiates a row update in the transaction manager, the specific row can be uniquely identified by locating the record's associated tablespace through the *space_id*, identifying the page containing the record via the *page_no*, and pinpointing the exact position of the record within a page using the *heap_no*. Therefore, the combination of $\langle space_id, page_no, heap_no \rangle$ serves as a unique identifier for a row. Once the corresponding record is located,

the system first checks the lock manager whether any active transactions conflict with the lock that the current transaction intends to acquire. Locks created by all active transactions are managed by a hash lock table, where the key is generated from the record's $\langle \text{space_id}, \text{page_no} \rangle$, and the value is the lock object (lock_t*). If no conflict exists, the lock is successfully acquired, its status is set to SUCCESS, and it is inserted into the hash table lock_sys . If a conflict is detected, the transaction is placed in a waiting state and added to a wait queue in lock_sys . Once the transaction holding the lock commits and releases the lock, it will awaken the waiting transactions in lock_sys , allowing the awakened transaction to acquire the lock. This process enables multiple transactions to operate simultaneously without interference.

2.3 Typical Applications at Tencent

High contented vs. Hotspot. High-contented workloads occur in concurrent environments where multiple transactions compete for access to the same resources, such as locks, data items, or database rows, leading to transaction delays. A hotspot refers to specific data items that are accessed frequently, causing them to become “hot”. Hotspots typically focus on a limited number of items, representing an extreme form of high-contented workloads.

WeChat Red Envelope is a popular application on WeChat, which has a monthly active user base exceeding 1.3 billion. TXSQL has facilitated WeChat’s red envelope payment system, allowing users to send red envelopes (monetary gifts) to others via WeChat. On traditional Chinese New Year’s Eve, we successfully handled a peak of Transaction per Second (TPS) up to 14 million red envelopes, showing the system’s capability to manage high volumes of concurrent transactions effectively.

Tencent Financial Technology (FiT) serves as a financial platform, offering a range of mobile payment and financial services. Its primary application is WeChat Pay, which facilitates an average of over one billion payment transactions daily. These transactions encompass a wide array of daily activities, including shopping, dining, transfers, and entertainment. During holidays or special events, such as Double Eleven Day, transaction volumes on WeChat Pay typically experience a significant surge.

3 Motivation and Design

This section begins by presenting some general lock optimization techniques in TXSQL, followed by a discussion of the design and motivation underlying both initial and current hotspot lock mechanisms within TXSQL.

3.1 General Lock Optimizations

3.1.1 Lightweight Locking. In the vanilla InnoDB Storage engine, table locks and row locks are the primary lock types, with all concurrent transaction lock information managed by a global lock system. This lock system has two significant shortcomings: (1) It created a large number of row locks. For instance, 100 concurrent transactions, each updating 10 rows, create up to 1,000 lock records for low- or non-contented workloads. (2) In high-contented workloads, even with partitioning optimizations [11, 34] applied, acquiring the sharded mutex for the corresponding page still incurs a substantial amount of lock wait time (see Figure 6c).

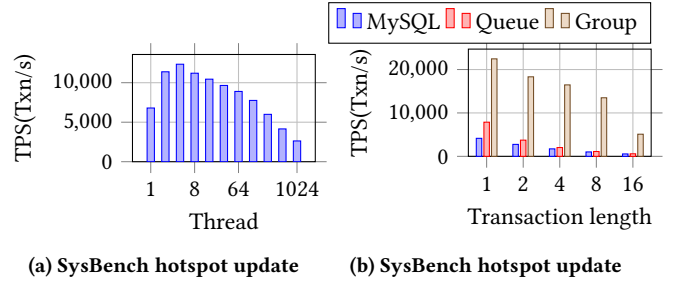


Figure 2: Motivation of hotspot optimization.

The goal of lightweight lock optimization is to streamline lock logic by minimizing the number of locks created and lowering the overhead associated with maintaining row locks. To achieve this, we introduced a variable named trx_lock_wait to manage transactions waiting for locks. This variable utilizes a lock-free hash design implemented with a map container, where the record ID serves as the key and the record value is a queue of waiting transactions. By recording transaction IDs, we can swiftly identify conflicts, creating locks only when necessary, thereby effectively reducing lock overhead.

3.1.2 Copy-Free Active Transaction List. During concurrent transaction execution, it checks the transaction IDs of all active transactions to determine which version of the data should be read in MVCC¹. Directly accessing the active transaction list can lead to performance issues, as it necessitates locking the list each time data is read. To mitigate this problem, one intuitive solution is to create a copy of the active transaction list during data reads, allowing checks on the copied list without locking the original. However, when read-write conflicts are substantial, the overhead associated with maintaining these copies can still be considerable.

To address this, we leverage snapshots in MVCC to avoid direct copying and locking of the active transaction list. Specifically, we aim to determine which data is visible by comparing the transaction ID with the version information of the data. We introduce a new attribute, del_ts , for each transaction ID, which indicates the deletion timestamp of each transaction. This attribute aids in assessing the transaction’s status, and now visibility can be determined by snapshot information and the del_ts attribute without acquiring the active transaction list, thus improving concurrency. Detailed implementation can be found in [15].

3.2 Queue Locking for Hotspot Access

We have identified performance issues associated with updates to hotspot rows, particularly when multiple transactions attempt to update the same row simultaneously. Notably, testing MySQL with a SysBench hotspot update workload (setup in Section 6.1) at a concurrency level of 1024 showed worse performance than serial execution at a concurrency level of 1, as illustrated in Figure 2a. Analysis revealed that this issue stems from the lock wait queue in the lock hash table lock_sys for each data row, where the cost of deadlock detection increases with the length of the queue.

¹ReadView structure is employed in InnoDB for MVCC.

Since deadlock detection is invoked during each transaction, and locks *lock_sys*, blocking other transactions. Consequently, as concurrency increases, the length of the lock wait queue also grows, leading to higher deadlock detection costs.

A common approach is to reduce the locks holding time of transactions (e.g., QURO [69]), which often (a) employs commit-time updates to defer write lock occupancy or (b) utilizes timestamp splitting to separate read and write timestamps. Another approach involves implementing a lightweight strategy to manually restrict access to hotspot data rows, similar to PolarDB's solution [50] by adding a hint to transaction hotspot query. However, these approaches necessitate modifications to the transaction syntax. For example, QURO requires reordering queries, while PolarDB adds hints for hot queries, meaning they cannot automatically detect hotspot updates during runtime for dynamic workloads.

We propose a queue locking mechanism specifically designed for hotspots. To enhance its generality and practicality, our design avoids syntax modifications and can automatically detect hotspot updates by monitoring row update operations and collecting update information in a lightweight manner (details in Section 4.1). Once a hotspot row is detected, its unique identifier is inserted as a key into a hotspot hash table, with the corresponding value being a queue that holds waiting transactions. Subsequently, when a transaction requests a row, if the unique identifier of that row is present in the hotspot hash table, the transaction is queued and will be awakened by preceding transactions that release their locks upon committing. Unlike traditional locking mechanisms, transactions updating hotspot rows do not directly acquire locks from the lock manager. Instead, they queue up before attempting to lock, thereby avoiding the overhead associated with contention for the lock manager.

To mitigate potential deadlocks during hotspot updates, we implemented a timeout mechanism instead of direct deadlock detection for two main reasons: (1) Performance: Deadlock detection requires traversing all related transaction information, which our tests (see Figure 2a) indicated is less efficient in hotspot scenarios. The timeout mechanism generally performs better and allows for configurable transaction wait times, making it more adaptable to dynamic real-world conditions. (2) Code complexity: Integrating deadlock detection would increase code complexity, complicating maintenance and raising the risk of new errors and instabilities.

3.3 Group Locking for Hotspot Access

We achieved improved results with the queue locking mechanism, however, its effectiveness was limited in customer workloads due to full synchronization between primary and secondary servers required for high availability, which increased transaction latency. Our analysis, using the SysBench workload with varying transaction lengths, confirmed that longer transaction latencies diminish queue locking effectiveness, as shown in Figure 2b.

We further survey current approaches and categorize them into three types of scheduling: (1) Thread-level scheduling (e.g., [54]): This approach schedules thread utilization through a thread pool to avoid thread-level contention. (2) Transaction-level scheduling (e.g., [10, 26, 35, 37, 69]): Similar to queue locking in Section 3.2, this method minimizes lock holding time to mitigate lock contention. (3) Query-level scheduling (e.g., [19, 25, 42, 67, 80]): This

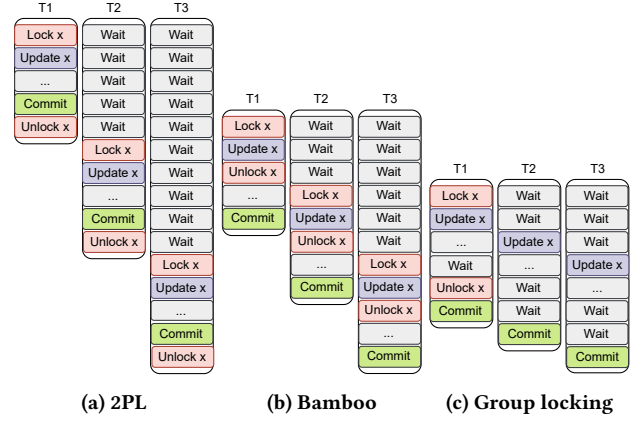


Figure 3: Locking schedules of update a hotspot row *x* under (a) 2PL, (b) Bamboo [28], and (c) group locking.

includes deterministic protocols that eliminate lock contention and transaction chopping that enables early reads of dirty data. For (1), they are orthogonal to our optimization focus in this paper. For (2), their benefits diminish significantly in high-latency transactions as investigated in Figure 2b, since locks can only be obtained after the conflicting transaction commits. For (3), they are generally impractical for conventional cloud service providers like us, as pre-obtaining user read-write sets is typically challenging. Moreover, in non-conflict situations, their performance often lags behind non-deterministic protocols due to scheduling overhead [31].

The state-of-the-art hotspot-oriented protocol that improves 2PL is Bamboo [28], which violates 2PL by releasing locks before commit as depicted in Figure 3b. In contrast, traditional 2PL requires locks to be released only after commit, as depicted in Figure 3a. While Bamboo is effective for certain high-contented workloads, it still incurs overhead from acquiring and releasing locks for each update. To address this limitation, we, inspired by deterministic protocols, propose a group locking mechanism. This mechanism logically groups a set of updates on a conflicting hotspot data row, allowing these updates to be executed sequentially within the group without locking. As depicted in Figure 3c, locking and unlocking operations occur only once per group, enabling conflicting transactions to proceed without locks. This approach eliminates the need for lock release and wake-up processes between transactions within the group, significantly reducing the time spent waiting for locks.

Note that all current solutions and our solution only update a single hotspot row per transaction and lack support for transactions with multiple hotspot rows. Such concurrent multi-row hotspot transactions often trigger deadlocks, resulting in cascading rollbacks and poor performance. In practice, transactions usually update one hotspot at a time.

4 Implementation

This section outlines the implementation of group locking in TXSQL and addresses various challenges and optimizations. Section 4.1 focuses on the detection and representation of hotspot rows, while Section 4.2 details the transaction processing mechanisms for group locking. Ensuring correctness in transaction processing between

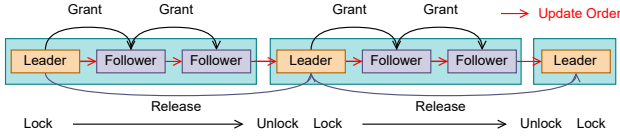


Figure 4: Design of group locking.

hotspot and non-hotspot data rows is crucial, and this is addressed in Sections 4.3 and 4.4, which detail how TXSQL maintains correct commit and rollback orders. Section 4.5 discusses the deadlock prevention mechanism, and Section 4.6 presents latency optimizations to tackle challenges arising from group locking.

4.1 Management of Hotspots

Recall the locking process: if a transaction attempts to acquire a lock that conflicts with an active transaction, it will be added to the corresponding lock wait queue. A longer wait queue for a specific row indicates higher contention, suggesting that the row is a hotspot. We define a row as a *hotspot* when the number of waiting transactions exceeds a threshold (e.g., 32, which is a rule of thumb [63]). Once identified, the hotspot is added to a hotspot hash table, denoted as *hot_row_hash*, and subsequent update transactions for this row must wait in the queue. To manage these entries effectively, a background thread periodically monitors the *hot_row_hash*. If an entry is no longer a hotspot (e.g., no waiting transactions), it is removed from the *hot_row_hash*, and future updates revert to the standard 2PL protocol. This simple yet effective mechanism enables us to identify and manage hotspot data with minimal overhead.

4.2 Transaction Processing

The concurrently arriving transactions on a hotspot row are automatically organized into groups, as depicted in Figure 4. The key design of the group lock is as follows:

- **Transaction role:** Within a group, transactions are either a leader or followers. The initial transaction serves as the leader, responsible for lock acquisition and release. The remaining transactions as followers do not participate in any locking operations.
- **Transaction execution:** Within a group, hotspot updates execute serially. Once a hotspot update is completed, it automatically grants execution to the next follower. Queries except those updating hotspot rows are executed in parallel.
- **Lock management:** When the leader transaction commits, it releases the row lock and awakens the next leader. The current leader releases the lock only after the last update within the group is completed. Subsequently, the new leader will acquire the row lock, thereby forming a new group.

During execution, a *dependency list* is formed and maintained based on the update order, which determines the order of commits and rollbacks. As queries are executed, each hotspot update transaction is assigned a globally incrementing identifier, referred to as *hot_update_order*, which is added to the dependency list, establishing a sequence of dependencies. A transaction can commit (or rollback) if it has no preceding (or subsequent) transaction in the dependency list. More details on commit and rollback order will be discussed in Sections 4.3 and 4.4, respectively.

Algorithm 1 Transaction Execution Process

```

1: function EXECUTE(T, item)
2:   if item is in hot_spots and item.dep_list ≠ 0 then
3:     item.waiting_updates.push_back(T)
4:     os_event_wait(T.event)           ▶ Wait to be awakened
5:   else
6:     lock_rec_add(item)               ▶ Traditional 2PL locking
7:   if T.hot_update_state ≠ NONE then
8:     trx.hot_update_order ← global_hot_update_order.fetch_add(1)
9:     item.dep_list.push_back(T)       ▶ Update dependency list
10:    update(T, item)                  ▶ Execute update
11:    if T.status = GRANTED then
12:      item.granting_new_trx ← false
13:    if T.hot_update_status = RUNNING and T.is_leader then
14:      item.switching_new_leader ← false
15:    if item.switching_new_leader then
16:      return                          ▶ To next leader
17:    next_trx ← item.waiting_updates.front()
18:    item.granting_new_trx ← true
19:    next_trx.hot_update_status ← GRANTED
20:    os_event_set(next_trx.event)

```

The transaction execution process, as shown in Algorithm 1, involves acquiring locks and queuing transactions, updating, and awakening transactions, as follows:

Step 1 - Locking and queuing (Lines 2-9): Check whether the item is a hotspot and whether it has a dependency list. If both conditions are met, the transaction is added to the waiting queue for updates and will remain in a waiting state until awakened. Otherwise, the item is locked by 2PL locking. If the transaction's hot update status is not empty, it is added to the item's dependency list, and the global hot update order is updated.

Step 2 - Updating (Lines 10-14): Update the item and the state of the transaction. If the transaction's state is GRANTED, the item will no longer grant access to a new follower transaction. If the transaction's hot update status is RUNNING and the transaction is the leader, the item will not switch to a new leader.

Step 3 - Awakening the next leader/follower (Lines 15-20): If the item is switching to a new leader, the process returns. Otherwise, the next transaction is retrieved from the waiting queue, the item's state for granting access to the new transaction is set to true, the hot update status of the next transaction is marked as GRANTED, and the next transaction is awakened.

Switching to hotspot. Upon initially transitioning to group locking, the older waiting transactions remain in the waiting queue of the *lock_sys*. Once a transaction is identified as a hotspot row, any new conflicting transactions are queued in the *waiting_update* list of *hot_lock_sys*. The leader of the group locking mechanism prioritizes awakening the older waiting transactions from *lock_sys*. Only after all these transactions have been processed does the leader begin to awaken the new waiting transactions from *waiting_update*.

4.3 Commit Order Guarantee

In group locking, we ensure that the commit order aligns with the update order through the use of the maintained dependency list. The transaction commit process, detailed in Algorithm 2, consists

Algorithm 2 Transaction Commit Process

```

1: function COMMIT(T, item)
2:   item.switching_new_leader  $\leftarrow$  true  $\triangleright$  No more granting new trx
3:   while item and item.granting_new_trx do
4:     ut_delay(10)  $\triangleright$  Wait updates complete for all granted trx
5:     for lock in T.locks do
6:       if lock_rec_release(lock) then continue
7:       next_trx  $\leftarrow$  item.waiting_updates.front()  $\triangleright$  Get next trx
8:       next_trx.hot_update_status  $\leftarrow$  RUNNING
9:       os_event_set(next_trx.event)
10:      next_trx.is_leader  $\leftarrow$  true
11:   commit_trx(T)
12:   item.dep_list.erase(T)  $\triangleright$  Remove from dependency list

```

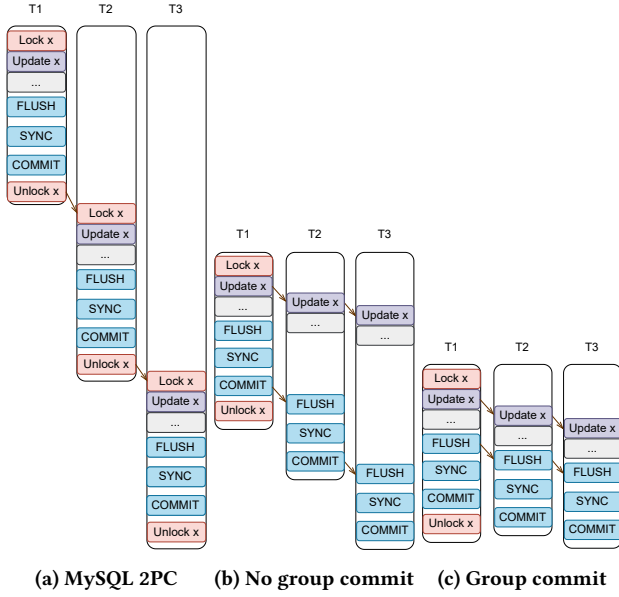


Figure 5: Locking schedules of commit phase in 2PC for updating a hotspot row *x* under (a) MySQL, (b) TXSQL W/O group commit, and (c) TXSQL with group commit.

of several key steps: releasing locks, awakening the next leader, and committing the transaction, as follows:

Step 1 (Lines 2-10) - Release locks and awaken the next leader (Lines 2-10): Set the switching state of the new leader for the item to *true*. Then, wait until the last granted transaction is completed. Next, iterate through all locks associated with the transaction and process each lock accordingly. If the lock can be released successfully, then continue. Next, retrieve the next transaction from the waiting updates queue, set its hot update status to RUNNING, and wake it up. Simultaneously, designate this next transaction as the leader.

Step 2 (Lines 11-12) - Commit transaction: Commit the transaction and remove it from the dependency list of the item.

Group commit optimization. In TXSQL, the XA/2-phase commit (2PC) mechanism is employed to ensure the consistency of storage-level logs (redo log and undo log) and server-level logs (binlog). This mechanism divides the transaction commit process into two phases: Prepare and Commit. The Commit phase is further subdivided into

Algorithm 3 Transaction Rollback Process

```

1: function ROLLBACK(T, item)
2:   if T.status == GRANTED then
3:     item.granting_new_trx  $\leftarrow$  false
4:     if T.hot_update_status == RUNNING  $\wedge$  T.is_leader then
5:       item.switching_new_leader  $\leftarrow$  false
6:       while (item.dep_list.back()  $\neq$  trx  $\vee$  item.granting_new_trx  $\vee$ 
7:         item.switching_new_leader) do
8:         ut_delay(10)  $\triangleright$  Wait all subsequent transactions rolled back
9:         rollback_trx(T)
10:        item.dep_list.erase(T)  $\triangleright$  Remove from dependency list

```

three stages: Flush, Sync, and Commit. In 2PL, a lock is required for each update, and these locks can be released after the commit phase in 2PC. Consequently, the order of the commit phase aligns with the order of updates, as depicted in Figure 5a. However, in TXSQL with group locking, only the leader is required to acquire a lock, which may result in discrepancies between the update order and the order in which updates arrive during the commit phase. Therefore, it is essential to enforce consistency between the update order and the commit phase order. Furthermore, since the commit phase consists of three serially executed stages (i.e., Flush, Sync, and Commit), executing them strictly in accordance with the update order can easily lead to a critical path and performance bottleneck as depicted in Figure 5b.

To address this, we implement a group commit optimization in 2PC. As depicted in Figure 5c, similar to group locking in 2PL, the first thread to enter the commit phase queue is designated as the leader, while subsequent threads function as followers. We ensure that transactions enter the Flush stage queue in the order of hotspot updates. When the leader of a commit phase group releases the lock, a group commit can occur. This approach not only guarantees the order of the commit phase but also leverages group commit to enhance overall efficiency.

4.4 Rollback Order Guarantee

TXSQL does not experience cascading abort in the absence of group locking. However, cascading aborts can occur in group locking, since the transaction can update the item without requiring a prior update to commit. To ensure correct aborts, the rollback order for hotspot updates must align with the reverse update order specified in the dependency list. The transaction rollback process, as shown in Algorithm 3, involves setting transaction and item states, performing safety checks, and executing the rollback, as follows:

Step 1 (Lines 2-5) - Set transaction and item states: If the transaction has been granted a hotspot update, the item will no longer grant new transactions. Additionally, if the hotspot update state of the transaction is RUNNING and the transaction is the leader, the item will not switch to a new leader.

Step 2 (Lines 6-7) - Check for safe rollback: Iteratively check whether the last transaction in the dependency list is not the current transaction and whether the item is granting new transactions or switching to a new leader. If either of these conditions is met, pause for a brief period before continuing the checks.

Step 3 (Lines 8-9) - Rollback the transaction: Execute the rollback of the transaction and remove it from the dependency list.

Rollback optimization. To achieve effective and efficient rollbacks, it is essential to implement corresponding safeguards at both the Server and Storage layers. When the Server layer initiates a transaction rollback, it sends a signal to the Storage layer. Upon receiving this signal, the Storage layer refrains from granting new followers and does not release locks to new leaders. Our tests have shown that, compared to allowing continued updates on hotspots that require a rollback, this approach significantly reduces the number of transactions that may abort during the rollback period. Once all transactions that need to be rolled back have been completed, another signal is sent to the Storage layer indicating that the rollback has finished, allowing it to resume granting and releasing operations. The Server layer is responsible for rolling back all uncommitted transactions but does not guarantee the order of the rollbacks; this order is ensured at the Storage layer. The dependency list allows us to determine whether a transaction is the most recent one to update the item. Only the transaction at the end of the list can be rolled back; other transactions must wait for the preceding transactions to complete their rollbacks.

4.5 Deadlock Handling

2PL can handle deadlocks through deadlock detection techniques or timeout mechanisms. When a deadlock is detected or a timeout occurs, the system can directly roll back one transaction to avoid deadlocks. However, with the introduction of our group locking mechanism in TXSQL, most updates occur without locks, making direct deadlock detection infeasible. An intuitive solution would be to integrate the dependency list into the existing deadlock detection mechanism of 2PL. However, this would inherently require more complex code logic. Additionally, as discussed in Section 3.2, the effectiveness of deadlock detection is limited in scenarios involving hotspot access. Our preliminary experiments indicate that a timeout mechanism outperforms deadlock detection in terms of performance. However, the timeout rollback may lead to situations where transactions cannot be rolled back immediately, potentially resulting in excessively long chains of cascading rollbacks, which can significantly impact overall performance [28].

Consequently, we have redesigned our deadlock prevention mechanism. Deadlocks can be avoided through a timeout mechanism when accessing only non-hotspot rows, as this scenario does not lead to cascading aborts. Therefore, our focus is primarily on deadlocks that arise from simultaneous access to both hotspot and non-hotspot rows. When a blocking situation is detected, and both the current transaction and the blocking transaction have updated the same hotspot row, the likelihood of a deadlock occurring is considerably high. In such cases, we proactively initiate a rollback to prevent deadlocks. We have observed that this rollback incurs significantly less overhead than a timeout rollback. We have provided an example in our extended paper in [65].

4.6 Latency Optimization

Unlike academic research, which typically imposes no limits on transaction requests per second, the industry often employs a fixed Transactions Per Second (TPS) rate model. This model sends a predetermined number of transaction requests to the database each

second, offering key advantages for customer workloads: (1) predictability in capacity planning and resource allocation, and (2) stability that ensures system availability under high load, minimizing the risk of crashes and performance degradation while maintaining consistent response times.

In group locking, each group is assigned a specific batch size, and hotspot updates are managed within the granting and releasing processes of that group. However, under a fixed TPS rate model, the level of concurrency can fluctuate significantly, resulting in abrupt and temporary decreases in request volume. This can frequently lead to empty wait queues, causing a group to enter periods where no subsequent transactions are available for granting. As a result, these groups may remain indefinitely stalled while awaiting the next hotspot transaction, thereby increasing overall latency.

One simple approach is to reduce the batch size, thereby decreasing the probability of waiting. However, this strategy may lead to an increased number of groups, resulting in more frequent lock acquisitions, and does not effectively address the underlying issue of transaction waiting. Another approach involves employing a background detection thread to periodically awaken the leader when possible. However, practical experience has shown that setting the period too short can lead to excessive checks of the *hot_row_hash* table, resulting in an over-acquisition of mutexes and a significant decline in performance. Conversely, setting the period too long can still result in a substantial number of high-latency transactions.

To address these challenges, we ultimately adopted a dynamic batch size strategy. When no waiting transactions are available, the leader directly releases the lock without assigning it to a new leader. In this scenario, any upcoming hotspot update transactions can continue the role of leader and initiate a new group.

5 Correctness

This section discusses the correctness of transaction dependency, serializability, and failure recovery in the context of group locking.

5.1 Transaction Dependency

The group locking mechanism enhances hotspot performance but compromises the original 2PL protocol, which requires a transaction to be committed before the next update begins. Group locking allows the next update to start immediately after completion, not after commit. To maintain the original execution and commit order of 2PL for hotspot data, three conditions must be met:

- **Transaction commit order:** When involving hotspot updates, a global queue by a dependency list is maintained to ensure that transactions are committed in the order of updates. Each transaction must verify that its predecessor has been committed before proceeding with its own commit.
- **Visibility of hotspot update:** To ensure that the next hotspot update transaction can access the results of the preceding hotspot update, it is essential to refresh the update results immediately after each hotspot update transaction is complete.
- **Transaction rollback order:** When a transaction needs to be rolled back, all subsequent transactions that depend on it should be rolled back in reverse order according to the dependency list.

5.2 Serializability

According to the theory of serializability, a schedule of transactions is considered serializable if and only if its serialization graph is acyclic [1, 8]. In 2PL, every schedule is serializable, as forming a cyclic graph violates the two-phase locking rule [28]. The core idea is that 2PL maintains the update order (i.e., conflict dependency in the conflict graph) consistent with the commit order. Therefore, to guarantee serializability in TXSQL, similar to Bamboo [28] violating 2PL but guaranteeing serializability, it is essential to maintain consistency between the update order and the commit order.

The commit order can be strictly enforced by adhering to the update order based on the dependency list. However, in the initial design, parallel updates may occur during the leader-switching phases in group locking. Consider a scenario in which transaction Trx1 is the last follower to be granted permission, while transaction Trx2 is the new leader that becomes awakened after the current leader releases the lock. Since Trx1 does not need to acquire any additional locks, it can directly update the hot row data. Simultaneously, Trx2 can also acquire the lock on the hotspot row and update the data. This situation lacks a mechanism to ensure the order of operations between these two transactions, potentially leading to an update loss anomaly, where both transactions read the same data and update it to different new values.

To prevent this, the old leader does not release the lock until all granted followers complete their updates, as discussed and implemented in Section 4.3. By doing so, we ensure that the update order is serial and unique. Since the commit order can align with the update order, we conclude that TXSQL can achieve serializability with group locking. Also, we describe the practical verification in the development process in Section 6.4.5.

5.3 Failure Recovery

In traditional 2PL failure recovery, unfinished transactions can roll back uncommitted updates individually, as the locking mechanism prevents concurrent transactions from simultaneously updating the same data row. However, when dealing with hotspot updates through group locking, it is essential to roll back multiple unfinished transactions in the correct order. To recap, the identifier *hot_update_order*, a globally incrementing identifier, has been added to the dependency list when hotspot updating. This identifier corresponds one-to-one with transactions, similar to the undo log header. We propose persisting the *hot_update_order* within the undo log header, however, there are currently no reserved fields for this purpose.

After a transaction is committed, the *hot_update_order* is removed from the dependency list and becomes ineffective, allowing us to repurpose the *TRX_UNDO_TRX_NO* field in the undo log header. This field originally records the transaction's *trx_no*, a sequence number generated upon commit to indicate the commit order of transactions. Since the effective periods of *hot_update_order* and *trx_no* do not overlap, we designate the first bit of the field as 1 to indicate a *hot_update_order*; otherwise, it indicates a *trx_no*.

Upon restart, we reconstruct the active transaction linked list from the undo log by reading the *TRX_UNDO_TRX_NO* field. If the transaction has not been committed, its globally incrementing value *hot_update_order* can be restored, allowing us to reorder the

active transaction linked list accordingly and proceed with sequential rollbacks in a single thread. If a failure occurs again during the rollback of hotspot transactions after a restart, those transactions that have already been rolled back will have completed their binlog entries. Consequently, upon the next restart, these transactions will not require rollback, allowing the remaining transactions to continue rolling back sequentially according to the *hot_update_order*, thereby maintaining correctness.

6 Evaluation

This section evaluates the performance of TXSQL empirically. Our goal is to validate three critical aspects of TXSQL: (1) its effectiveness in handling hotspot locks (Section 6.2); (2) its performance superiority over state-of-the-art solutions across a variety of high-contention scenarios (Section 6.3); (3) its performance under various workloads especially real-world ones in Tencent Cloud (Section 6.4).

6.1 Setup

We conducted our experiments on three servers, each equipped with an Intel(R) Xeon(R) Gold 6133 CPU @2.50GHz, featuring 80 cores, 753 GB of DRAM, and a 15 TB SSD. The operating system used was CentOS Linux release 7.2. The average network latency between servers was measured at 1.033 ms.

6.1.1 Workloads. Four benchmarks were conducted as follows: (1) **SysBench** [2] is a versatile open-source benchmarking tool that is frequently used in industrial database systems like MySQL and PostgreSQL. By default, the read/write (RW) ratio is set to 0.5, the transaction length (TL) is 14, and the default *skew factor* (SF) is set to 0.7 for Zipf distribution. The main workloads include hotspot update (RW=0, TL=1; All updates in one hotspot row), hotspot mix read/write (RW=0.5, SF=0.9), hotspot scan (RW=0, TL=10), uniform update (RW=0, uniform), and uniform read-only (RW=1, uniform). (2) **TPC-C** [11, 71] is a popular OLTP benchmark that simulates e-commerce scenarios. The standard TPC-C is implemented and each warehouse contains about 100 MB of data, and we vary the number of warehouses to simulate different contentions. (3) **FiT** simulates the transaction operations of the Financial Transaction system in Tencent. After data anonymization, the primary structure consists of two tables: a hot table that records account information with frequent updates to user balances, and a non-hot table that stores all transaction records. (4) **Hotspots** is a real-life workload combined with three applications that experience spikes of high-contented loads from time to time.

6.1.2 Baselines. **MySQL:** The base database MySQL v8.0.30. **O1:** The general lock optimization discussed in Section 3.1. **O2:** O1 and the queue locking optimization discussed in Section 3.2. **TXSQL:** O1 and the group locking optimization discussed in Section 3.3. By default, the group locking batch size is set to 10. **Aria** [42]: A state-of-the-art (SOTA) deterministic protocol representative which combines OCC and deterministic protocol without pre-knowing read-write sets. We tuned the best batch size for Aria. **Bamboo** [28]: SOTA solution that violates 2PL to boost performance. For an apples-to-apples comparison, we implemented all approaches, including Aria and Bamboo, within TXSQL.

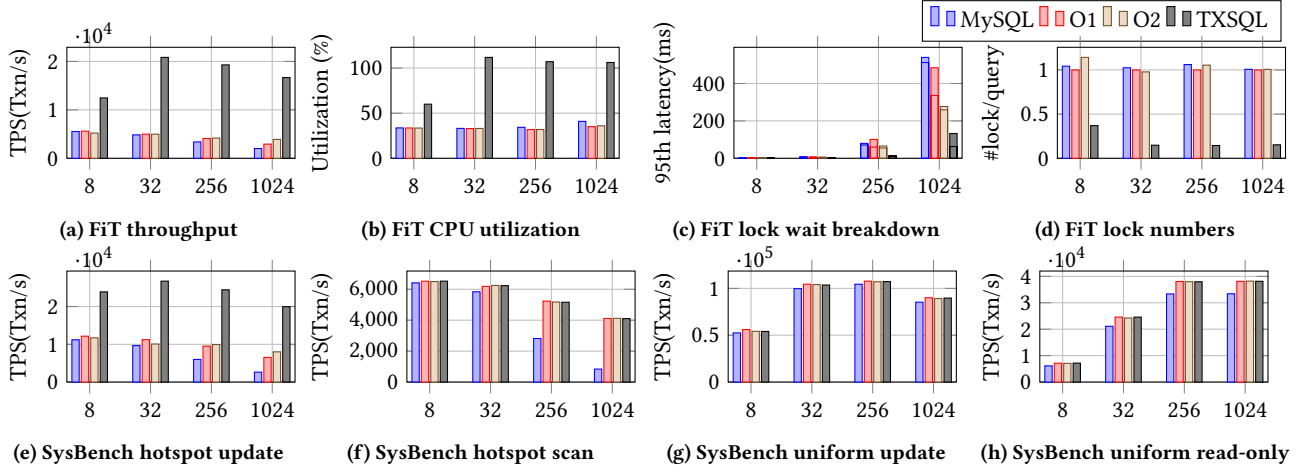


Figure 6: Effect of optimizations. The X-axis is the thread count, and the higher the more concurrent requests.

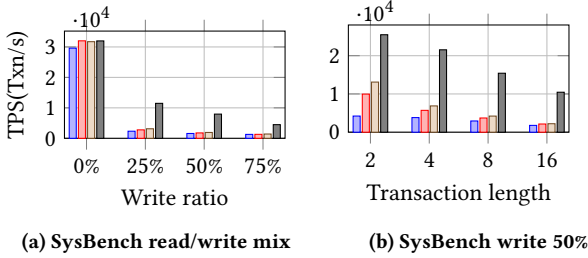


Figure 7: Effect of optimizations (Thread=1024).

6.2 Ablation Study

This part evaluates the effectiveness of the optimizations in TXSQL against MySQL, general lock optimization (O1), and queue locking optimization (O2) using FiT and SysBench workloads.

6.2.1 In-depth Performance Analysis. In the FiT workload, as the number of threads increases, both O1 and O2 show improvements in lock waiting times compared to MySQL, as shown in Figure 6c, as they reduce the lock-holding time, however, the 95th percentile latency is slightly higher in O1 due to the overhead required for actively detecting deadlocks (The entire bar indicates the transaction latency, while the line within the bar represents the lock waiting time). The benefits of O2 are still limited due to its long lock wait time for the hotspot. TXSQL, through the implementation of group locking, significantly reduces the number of locks, as shown in Figure 6d. As a result, TXSQL achieves better CPU utilization, as shown in Figure 6b, resulting in a clear throughput advantage over MySQL, O1, and O2, with improvements of up to 8.25x, as shown in Figure 6a.

6.2.2 Hotspot and Non-hotspot Scenarios. In SysBench workloads, TXSQL achieved up to a 7.5x improvement in hotspot update workload compared to those observed in the FiT workload, as shown in Figure 6e. In uniform scenarios (both updates and read-only workloads), as shown in Figures 6g and 6h, O2 and TXSQL show no performance improvement over O1, as neither triggers a hotspot

lock mechanism. Luckily, the overhead of hotspot detection for O2 and TXSQL remained below 2% in non-hotspot scenarios. In the hotspot scan workload, as shown in Figure 6f, O2 and TXSQL also showed no improvement due to the dispersion of updates across multiple hotspots. O2 and TXSQL implemented optimizations of O1, resulting in performance improvements over MySQL in general non-hotspot scenarios.

To conduct a more thorough analysis, we varied the write ratio and transaction length in SysBench hotspot update (TL=20) scenarios, as shown in Figure 7. TXSQL achieved the best performance across all scenarios, despite a decline in performance across all systems with increasing write ratios or transaction lengths. Notably, O2 is somewhat effective when the transaction length is small compared to MySQL; however, its effectiveness diminishes with larger transaction lengths. This is because the increased latency exacerbates the lock contention in hotspot scenarios.

6.3 Comparison to State-of-the-art Solutions

This part evaluates the efficiency of TXSQL against MySQL, a SOTA deterministic protocol (Aria), and a SOTA 2PL protocol (Bamboo) by various scenarios.

6.3.1 Scalability. We evaluated scalability using the SysBench hotspot update, as illustrated in Figure 8. With an increase in the number of threads, TXSQL achieved a performance improvement of up to 7x. This is attributed to the group locking mechanism, which significantly reduces lock contention for transactions at hotspots, thereby decreasing overall transaction latency, as evidenced by the 95th percentile latency statistics. Bamboo demonstrated superior performance under low concurrency compared to MySQL and Aria; however, its performance improvement under high concurrency was less effective. This is primarily because, although locks can be released before the transaction commits and the lock-holding time remains relatively short compared to Aria and MySQL, each update still requires acquiring locks, leading to substantial lock contention at hotspots. Aria, utilizing a deterministic algorithm,

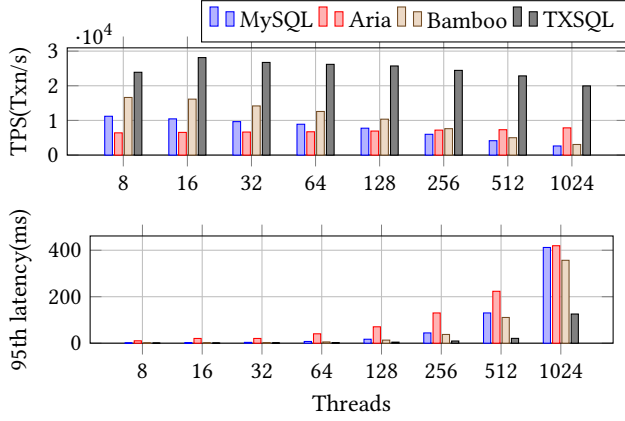


Figure 8: Effect of scalability by Sysbench hotspot update.

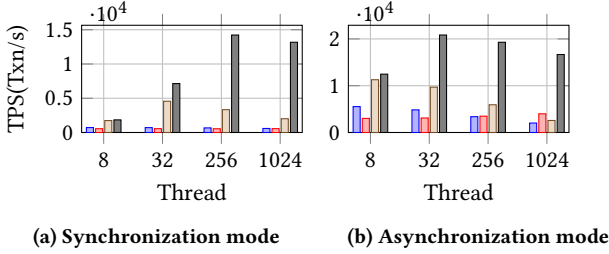


Figure 9: Effect of synchronization by FiT.

maintained stable TPS as the number of threads increased, thanks to its consistent scheduling of batch-size transactions.

6.3.2 Synchronization. In scenarios involving both synchronous and asynchronous replication, as illustrated in Figure 9, TXSQL achieved performance improvements of 22.3x and 8.2x, respectively. Although the overall performance declined with synchronous replication, TXSQL demonstrated a relatively greater enhancement. This is attributed to the longer lock-holding times of transactions, which the group locking mechanism effectively mitigates. This phenomenon is akin to the situation where an increase in transaction length leads to performance degradation, as shown in Figure 7b. Bamboo performs well under low concurrency due to its ability to release locks earlier, thereby minimizing conflicts. However, as contention increases, the intensification of lock competition results in sub-optimal performance. In the context of synchronous replication, Bamboo outperforms both Aria and MySQL, as the higher transaction latency in synchronous scenarios means that early lock release has a direct impact on reducing latency and enhancing performance.

6.3.3 Effect of Abort. This part tests the impact of cascading rollbacks on TXSQL and Bamboo by actively injecting rollback transactions during the SysBench hotspot update (RW=0.5, TL=16). As shown in Figure 10 (left), TXSQL outperformed Bamboo in terms of handling cascading rollbacks; however, both systems experienced significant effects, with rollback ratios increasing by more than tenfold, leading to substantial performance degradation. Cascading rollbacks were more likely to occur when transaction lengths

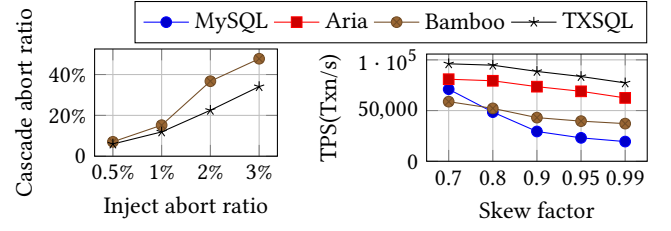


Figure 10: Effect of abort and skewness by SysBench.

were high and multiple hotspots were present. Luckily, for Tencent Cloud applications, we did not observe significant rollbacks when enabling group locking, as most transactions have relatively short lengths [36, 40, 64] and typically involve only one hotspot. Additionally, we can monitor the rollback rate, and if it becomes excessively high, we can disable group locking and revert to the original 2PL protocol on the fly.

6.3.4 Skewness. This part tests the impact of access skewness using SysBench update workload (TL=1), as shown in Figure 10 (right). As skewness increases, contention rises, leading to a significant enhancement in TXSQL’s performance, with improvements ranging from 1.6x to 3.9x. Notably, despite the high skewness, the contention level does not exceed that of a hotspot workload. In scenarios with a skewness level of 0.99, multiple hotspots may arise; however, the queuing situation for these hotspots is less severe than the contention observed during our previous tests with the SysBench hotspot update involving a single hotspot. The limited improvement observed in Bamboo can be attributed to the relatively short duration of the workload transactions, resulting in a shorter overall lock-holding time and minimal optimization potential. Conversely, Aria experiences a gradual increase in the impact of rollbacks on performance as skewness rises, with a rollback rate exceeding 20% at a skewness level of 0.99.

6.4 More Workloads and Scenarios

6.4.1 Real-world Applications. Since 2023, our internal FIT payment and financial services have upgraded over 20,000 instances from MySQL to the TXSQL kernel. This transition has resulted in an overall performance improvement of 30% and has collaboratively addressed long-standing performance jitter issues, where sudden spikes of load requests may significantly decrease the throughput. Additionally, the enhanced capability for handling hotspot update transactions has boosted performance by nearly 10 times.

We next present a real-world hotspot workload, a composite of three applications in Tencent Cloud, adhering to a fixed TPS rate, as shown in Figure 11. It is important to note that the machines employed in this context are distinct from those utilized in other experiments. In order to maintain consistency with the online instance deployment, the specific configuration adopted here consists of an Intel(R) Xeon(R) CPU E5-2620 24 cores, 128 GB DRAM, 1.8 TB SSD, with an average network latency of 2.516 ms between the source and 2 semi-sync replicas. The group locking optimization is an on-the-fly configurable parameter, activated after 23:55. We observe that, for the majority of the time, the TPS remains at a stable level. However, during the period when group locking is disabled,

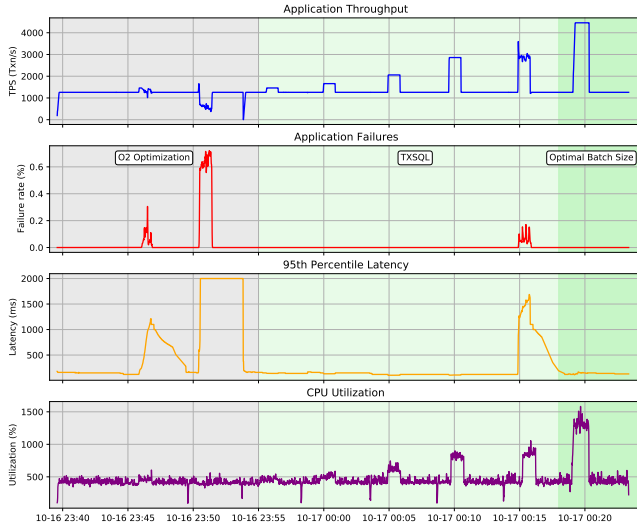


Figure 11: Online application workloads with hotspots.

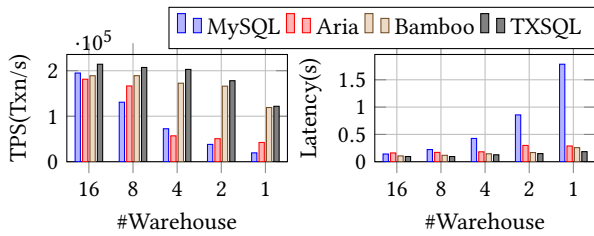


Figure 12: Effect of warehouses by TPC-C.

the system’s processing capacity is adversely affected by sudden surges in request volume, leading to increased failure rates and latency. For instance, at 23:52, a hotspot emerges, causing the TPS to drop below its original level, despite CPU utilization remaining low. Once group locking is enabled in TXSQL, the system effectively manages sudden requests with minimal transaction failures and latency increases. However, as TPS continues to rise under sustained request conditions, TXSQL can process higher throughput but may experience some degree of failure rates and increased latency. Nevertheless, by further increasing the batch size within the group at 00:18, TXSQL is still able to handle hotspots efficiently.

6.4.2 TPC-C. Under TPC-C workloads, a smaller number of warehouses leads to a greater number of conflicts. When the number of warehouses is set to one, TXSQL demonstrates a relatively better performance improvement compared to other opponents, as shown in Figure 12 (left). However, Bamboo also performs well under this workload, primarily because TPC-C workloads consist of long transactions, which benefit significantly from early lock releases. Nevertheless, Bamboo incurs the overhead of locking and unlocking operations for each SQL statement, which is more costly than group locking. In the case of Aria, although latency is low

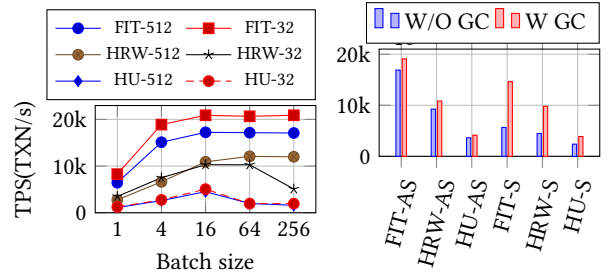


Figure 13: Effect of batch size and group commit (GC).

with fewer warehouses (as indicated by the average latency of Payment transactions in Figure 12 (right)), the high rate of transaction rollbacks negatively impacts its overall performance.

6.4.3 Batch Size. We investigated the impact of fixed batch size for group locking on performance with two thread settings (512 and 32), as shown in Figure 13. Initially, performance improved with larger batch sizes due to fewer required locks. In shorter transactions, like FIT, performance remained stable even at a batch size of 256. However, in longer update transactions, such as the SysBench hotspot update (HU) workload (RW=0,TL=16), performance declined beyond a certain batch size due to increased conflicts and rollbacks. Interestingly, in the SysBench hotspot read/write (HRW) workload (RW=0.5,TL=16), performance with large batch sizes was poorer under low concurrency (thread=32). Here, transaction arrival rates were lower than the rate at which group locking activated transactions, necessitating background threads to periodically check for waiting transactions. This indicates that maintaining a fixed batch size under a fixed TPS rate can increase latency and degrade performance. As discussed in Section 4.6, a dynamic batch size strategy can mitigate this issue by allowing the leader to release locks directly when no waiting transactions are present.

6.4.4 Group Commit. We continued the workloads from the previous subsection to test the performance comparison of group commit under both synchronous (S) and asynchronous (AS) modes with a setting of 512 threads. As shown in Figure 13, we observe a greater benefit from group commit in synchronous mode. This is attributed to the increased occurrence of grouped transactions, which enhances the likelihood of reducing network communication between the synchronization and commit phases.

6.4.5 Correctness Check. We employed various practical testing methods to ensure the correctness of results produced by the group locking mechanism. For specific workloads, we verify consistency by checking if the results align with the expected outcomes based on logical operations. For instance, in TPC-C, we confirm that total sales for a Warehouse match those of its District. In FiT, we log each transaction and compare the total transaction value with recorded values. We also use database and business logs in a reconciliation system for verification. Additionally, we utilize transaction correctness validation tools, including our DBChaos testing platform, which tests the execution results under numerous fault scenarios, including infrastructure failures (such as disk and network) and custom faults. Before going live, we conduct a *canary* release using

two parallel systems: one with group locking deployed and the one without, performing data verification every minute. We also conduct isolation checks using tools such as Jepsen [3] and IsoVista [27] to ensure the isolation correctness of our implementation.

6.4.6 Failure Recovery. This part evaluated the restart duration by terminating processes and reconnecting to the client. In the scenarios involving FiT-512, HRU-512, and HU-512, TXSQL achieved a TPS improvement of 5x to 13x compared to MySQL. The crash recovery durations were approximately 10 seconds for TXSQL, 7 seconds for MySQL, and 5 seconds for TXSQL without active transactions. The longer recovery time for TXSQL is due to both the length of active transactions and the volume of redo logs that need to be applied, which increases with higher TPS.

6.5 Insight and Discussion

Potential for performance improvement. This paper focuses on improving lock contention in concurrency control protocols for hotspot scenarios. In such scenarios, CPU and memory resources are not the bottlenecks; rather, the contention arises from single data access points. There are other avenues to address hotspot issues that complement group locking optimization. For instance, caching hotspot row locations in the index can mitigate overhead from repeated index lookups [30]. Additionally, we utilize a 2PC mechanism for both binlog and redolog to ensure the consistency of logical and physical logs, thus maintaining data correctness in extreme situations. However, this approach can significantly affect performance. Since the contents of these logs are consistent, we can optimize the mechanism by merging them or converting one into the other, allowing for a single log retention [11].

Trade-offs for simplicity and generality. The current solution is limited to updating a single hotspot row within a transaction and does not support transactions involving multiple hotspot rows. Concurrent transactions involving multiple hotspot rows can easily lead to deadlocks, causing cascading rollbacks and negatively impacting performance. In practice, transactions typically update only one hotspot at a time. If two rows within the same transaction are frequently updated simultaneously, it is advisable to merge these data elements into the same table or row according to database normalization principles (e.g., 3NF), to reduce data redundancy and enhance consistency. Our solution was compared only with one system MySQL for fairness. However, the group locking should be easily and effectively applicable to other 2PL systems.

Statements and isolation levels. UPDATE, INSERT, and DELETE statements, which are classified as Current Reads collectively referred to as update operations, require locks. Common SELECT statements under Repeatable Read and Read Committed isolation levels are considered Snapshot Reads and do not require locks. However, under the Serializable (SER) isolation level, SELECT is treated as a Current Read and requires locks. While the lock optimization in this paper is independent of isolation levels, this paper focuses on the strongest SER levels and supports SELECT FOR UPDATE. Since INSERT and DELETE operations do not create hotspots, *Phantom Reads* can be avoided within the group locking mechanism through next-key locking in indexes [44].

7 Related Work

OLTP optimizations. The recent researched concurrency control protocols, since 2012, can be classified into six categories: two-phase locking (2PL) [28, 54], timestamp ordering (TO), multi-version concurrency control (MVCC) [22, 24, 48, 68], optimistic concurrency control (OCC) [64, 73–75], deterministic concurrency control [9, 21, 23, 39, 42, 46, 52, 62], and adaptive concurrency control [56–58, 60, 61, 66, 67]. At present, deterministic and adaptive protocols are progressively gaining prominence in academia. However, deterministic protocols require prior knowledge of the read-write sets, while adaptive protocols pose significant challenges to correctness. As a result, these protocols are limited in adoption in the industry. This paper mainly focuses on optimizing the locking mechanism of the widely used 2PL protocol in TXSQL. Although many efforts to optimize OLTP systems focus on different aspects, such as consensus protocols (e.g., [12, 47, 70]), data partitioning or disaggregation (e.g., [53, 55, 59, 79]), data scheduling (e.g., [16–18, 43, 82]), new hardware (e.g., [9, 51, 81]), and performance parameter tuning (e.g., [32, 41, 78]), they fall outside the scope of concurrency control in this paper and are orthogonal to our objectives.

Hotspot optimizations. We have categorized hotspot optimizations into three types of scheduling. (1) Thread-level scheduling [54]: This approach can complement our work by utilizing a thread pool to manage thread utilization, aiming to minimize thread contention. (2) Transaction-level scheduling [10, 26, 35, 37, 69]: This method incorporates lock queuing or minimizes lock holding time to mitigate lock contention. However, its effectiveness is limited in high-latency scenarios. (3) Query-level scheduling (e.g., [19, 25, 42, 67, 80]): Deterministic protocols exemplify typical query-level scheduling methods. While these protocols can effectively eliminate lock contention, they necessitate prior knowledge of the read-write sets. This paper has employed query-level scheduling for hotspot updates by automatically detecting hot data access and implemented a group locking mechanism, which groups hotspot updates and executes them serially without locking to enhance performance.

8 Conclusion

In this paper, we presented the Tencent Database System, TXSQL, and its optimizations towards high-contented workloads. We provide motivation, optimizations, and insights into TXSQL to address lock conflict issues. When dealing with hotspot data, we propose a group locking mechanism that groups hotspot data accesses and executes them serially without locking. Additionally, we describe its correctness in terms of deadlock prevention, rollback, and failure recovery. Through extensive analysis and performance evaluation, we demonstrate that TXSQL achieves performance improvements of up to 6.5x and 22.3x compared to state-of-the-art methods and systems, respectively. Our results also indicate the effectiveness and efficiency of TXSQL in real-world workloads.

9 Acknowledgment

We would like to express our sincere gratitude to the reviewers for their invaluable comments and insightful suggestions. We acknowledge the dedicated efforts of the TXSQL team, whose development work has been instrumental in the realization of this project.

References

- [1] Atul Adya, Barbara Liskov, and Patrick E. O’Neil. 2000. Generalized Isolation Level Definitions. In *ICDE*. IEEE Computer Society, 67–78.
- [2] Akopytovd. 2024. *SysBench github repo*. <https://github.com/akopytov/sysbench>.
- [3] Peter Alvaro and Kyle Kingsbury. 2020. Elle: Inferring Isolation Anomalies from Experimental Observations. *Proc. VLDB Endow.* 14, 3 (2020), 268–280.
- [4] Raja Appuswamy, Angelos-Christos G. Anadiotis, Danica Porobic, Mustafa Iman, and Anastasia Ailamaki. 2017. Analyzing the Impact of System Architecture on the Scalability of OLTP Engines for High-Contention Workloads. *Proc. VLDB Endow.* 11, 2 (2017), 121–134.
- [5] Claude Barthels, Ingo Müller, Konstantin Taranov, Gustavo Alonso, and Torsten Hoefler. 2019. Strong consistency is not hard to get: Two-Phase Locking and Two-Phase Commit on Thousands of Cores. *Proc. VLDB Endow.* 12, 13 (2019), 2325–2338.
- [6] Philip A. Bernstein and Nathan Goodman. 1981. Concurrency Control in Distributed Database Systems. *ACM Comput. Surv.* 13, 2 (1981), 185–221.
- [7] Philip A. Bernstein and Nathan Goodman. 1983. Multiversion Concurrency Control - Theory and Algorithms. *ACM Trans. Database Syst.* 8, 4 (1983), 465–483.
- [8] Philip A. Bernstein, David W. Shipman, and Wing S. Wong. 1979. Formal Aspects of Serializability in Database Concurrency Control. *IEEE Trans. Software Eng.* 5, 3 (1979), 203–216.
- [9] Nils Boeschen and Carsten Binnig. 2022. GaccO - A GPU-accelerated OLTP DBMS. In *SIGMOD Conference*. ACM, 1003–1016.
- [10] Xusheng Chen, Haoze Song, Jianyu Jiang, Chaoyi Ruan, Cheng Li, Sen Wang, Gong Zhang, Reynold Cheng, and Heming Cui. 2021. Achieving low tail-latency and high scalability for serializable transactions in edge computing. In *EuroSys*. ACM, 210–227.
- [11] Yuxing Chen, Anqun Pan, Hailin Lei, Anda Ye, Shuo Han, Yan Tang, Wei Lu, Yunpeng Chai, Feng Zhang, and Xiaoyong Du. 2024. TDSQL: Tencent Distributed Database System. *Proc. VLDB Endow.* 17, 12 (2024), 3869–3882.
- [12] Austin T. Clements, M. Frans Kaashoek, Nickolai Zeldovich, Robert Tappan Morris, and Eddie Kohler. 2013. The scalable commutativity rule: designing scalable software for multicore processors. In *SOSP*. ACM, 1–17.
- [13] Tencent Cloud. 2024. *Tencent Cloud official site*. <https://www.tencentcloud.com/>.
- [14] Tencent Cloud. 2024. *TXSQL: database kernel maintained by the Tencent database team*. <https://www.tencentcloud.com/document/product/236/35988?lang=en&pg=>.
- [15] Tencent Cloud. 2024. *TXSQL gitee repo*. <https://gitee.com/X-SQL/TXSQL>.
- [16] Carlo Curino, Yang Zhang, Evan P. C. Jones, and Samuel Madden. 2010. Schism: a Workload-Driven Approach to Database Replication and Partitioning. *Proc. VLDB Endow.* 3, 1 (2010), 48–57.
- [17] Sudipto Das, Divyakant Agrawal, and Amr El Abbadi. 2010. G-Store: a scalable data store for transactional multi key access in the cloud. In *SoCC*. ACM, 163–174.
- [18] Sudipto Das, Shoji Nishimura, Divyakant Agrawal, and Amr El Abbadi. 2011. Albatross: Lightweight Elasticity in Shared Storage Databases for the Cloud using Live Data Migration. *Proc. VLDB Endow.* 4, 8 (2011), 494–505.
- [19] Bailu Ding, Lucja Kot, and Johannes Gehrke. 2018. Improving Optimistic Concurrency Control Through Transaction Batching and Operation Reordering. *Proc. VLDB Endow.* 12, 2 (2018), 169–182.
- [20] Haowen Dong, Chao Zhang, Guoliang Li, and Huanchen Zhang. 2024. Cloud-Native Databases: A Survey. *IEEE Trans. Knowl. Data Eng.* 36, 12 (2024), 7772–7791. <https://doi.org/10.1109/TKDE.2024.3397508>
- [21] Zhiyuan Dong, Chuzhe Tang, Jia-Chen Wang, Zhaoguo Wang, Haibo Chen, and Binyu Zang. 2020. Optimistic Transaction Processing in Deterministic Database. *J. Comput. Sci. Technol.* 35, 2 (2020), 382–394.
- [22] Dominik Durner and Thomas Neumann. 2019. No false negatives: Accepting all useful schedules in a fast serializable many-core system. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*. IEEE, 734–745.
- [23] Jose M. Faleiro, Daniel Abadi, and Joseph M. Hellerstein. 2017. High Performance Transactions via Early Write Visibility. *Proc. VLDB Endow.* 10, 5 (2017), 613–624.
- [24] Jose M. Faleiro and Daniel J. Abadi. 2015. Rethinking serializable multiversion concurrency control. *Proc. VLDB Endow.* 8, 11 (2015), 1190–1201.
- [25] Jose M. Faleiro, Alexander Thomson, and Daniel J. Abadi. 2014. Lazy evaluation of transactions in database systems. In *SIGMOD Conference*. ACM, 15–26.
- [26] Goetz Graefe, Mark Lillibridge, Harumi A. Kuno, Joseph A. Tucek, and Alistair C. Veitch. 2013. Controlled lock violation. In *SIGMOD Conference*. ACM, 85–96.
- [27] Long Gu, Si Liu, Tiancheng Xing, Hengfeng Wei, Yuxing Chen, and David A. Basin. 2024. IsoVista: Black-box Checking Database Isolation Guarantees. *Proc. VLDB Endow.* 17, 12 (2024), 4325–4328.
- [28] Zhihan Guo, Kan Wu, Cong Yan, and Xiangyao Yu. 2021. Releasing Locks As Early As You Can: Reducing Contention of Hotspots by Violating Two-Phase Locking. In *SIGMOD Conference*. ACM, 658–670.
- [29] Ravindra Guravannavar and S. Sudarshan. 2008. Rewriting procedures for batched bindings. *Proc. VLDB Endow.* 1, 1 (2008), 1107–1123.
- [30] Xiangpeng Hao and Badrish Chandramouli. 2024. Bf-Tree: A Modern Read-Write-Optimized Concurrent Larger-Than-Memory Range Index. *Proc. VLDB Endow.* 17, 11 (2024), 3442–3455.
- [31] Rachael Harding, Dana Van Aken, Andrew Pavlo, and Michael Stonebraker. 2017. An Evaluation of Distributed Concurrency Control. *Proc. VLDB Endow.* 10, 5 (2017), 553–564.
- [32] Herodotos Herodotou, Yuxing Chen, and Jiaheng Lu. 2021. A Survey on Automatic Parameter Tuning for Big Data Processing Systems. *ACM Comput. Surv.* 53, 2 (2021), 43:1–43:37.
- [33] Yihe Huang, William Qian, Eddie Kohler, Barbara Liskov, and Liuba Shrira. 2020. Opportunities for Optimism in Contended Main-Memory Multicore Transactions. *Proc. VLDB Endow.* 13, 5 (2020), 629–642.
- [34] Ryan Johnson, Ippokratis Pandis, Nikos Hardavellias, Anastasia Ailamaki, and Babak Falsafi. 2009. Shore-MT: a scalable storage manager for the multicore era. In *EDBT (ACM International Conference Proceeding Series, Vol. 360)*. ACM, 24–35.
- [35] Hyungsoo Jung, Hyuck Han, Alan D. Fekete, Gernot Heiser, and Heon Young Yeom. 2013. A scalable lock manager for multicores. In *SIGMOD Conference*. ACM, 73–84.
- [36] Robert Kallman, Hideaki Kimura, Jonathan Natkins, Andrew Pavlo, Alex Rasin, Stanley B. Zdonik, Evan P. C. Jones, Samuel Madden, Michael Stonebraker, Yang Zhang, John Hugg, and Daniel J. Abadi. 2008. H-store: a high-performance, distributed main memory transaction processing system. *Proc. VLDB Endow.* 1, 2 (2008), 1496–1499.
- [37] Hideaki Kimura, Goetz Graefe, and Harumi A. Kuno. 2012. Efficient Locking Techniques for Databases on Modern Hardware. In *ADMS@VLDB*. 1–12.
- [38] Donald Kossmann, Tim Kraska, and Simon Loesing. 2010. An evaluation of alternative architectures for transaction processing in the cloud. In *SIGMOD Conference*. ACM, 579–590.
- [39] Yu-Shan Lin, Ching Tsai, Tz-Yu Lin, Yun-Sheng Chang, and Shan-Hung Wu. 2021. Don’t Look Back, Look into the Future: Prescient Data Partitioning and Migration for Deterministic Database Systems. In *SIGMOD Conference*. ACM, 1156–1168.
- [40] Haonan Lu, Shuai Mu, Siddhartha Sen, and Wyatt Lloyd. 2023. NCC: Natural Concurrency Control for Strictly Serializable Datastores by Avoiding the Timestamp-Inversion Pitfall. In *OSDI*. USENIX Association, 305–323.
- [41] Jiaheng Lu, Yuxing Chen, Herodotos Herodotou, and Shvinnath Babu. 2019. Speedup Your Analytics: Automatic Parameter Tuning for Databases and Big Data Systems. *Proc. VLDB Endow.* 12, 12 (2019), 1970–1973.
- [42] Yi Lu, Xiangyao Yu, Lei Cao, and Samuel Madden. 2020. Aria: A Fast and Practical Deterministic OLTP Database. *Proc. VLDB Endow.* 13, 11 (2020), 2047–2060.
- [43] Yi Lu, Xiangyao Yu, and Samuel Madden. 2019. STAR: Scaling Transactions through Asymmetric Replication. *Proc. VLDB Endow.* 12, 11 (2019), 1316–1329.
- [44] C. Mohan. 1990. ARIES/KVL: A Key-Value Locking Method for Concurrency Control of Multiaction Transactions Operating on B-Tree Indexes. In *VLDB*. Morgan Kaufmann, 392–405.
- [45] Shuai Mu, Sebastian Angel, and Dennis E. Shasha. 2019. Deferred Runtime Pipelining for contentious multicore software transactions. In *EuroSys*. ACM, 40:1–40:16.
- [46] Shuai Mu, Yang Cui, Yang Zhang, Wyatt Lloyd, and Jinyang Li. 2014. Extracting More Concurrency from Distributed Transactions. In *OSDI*. USENIX Association, 479–494.
- [47] Shuai Mu, Lamont Nelson, Wyatt Lloyd, and Jinyang Li. 2016. Consolidating Concurrency Control and Consensus for Commits under Conflicts. In *OSDI*. USENIX Association, 517–532.
- [48] Thomas Neumann, Tobias Mühlbauer, and Alfons Kemper. 2015. Fast Serializable Multi-Version Concurrency Control for Main-Memory Database Systems. In *SIGMOD Conference*. ACM, 677–689.
- [49] Patrick E. O’Neil. 1986. The Escrow Transactional Method. *ACM Trans. Database Syst.* 11, 4 (1986), 405–430.
- [50] PolarDB. 2024. *PolarDB Hotspot Optimization*. <https://help.aliyun.com/zh/polardb/polardb-for-mysql/user-guide/hot-row-optimization>.
- [51] Shujian Qian and Ashvin Goel. 2024. Massively Parallel Multi-Versioned Transaction Processing. In *OSDI*. USENIX Association, 765–781.
- [52] Dai Qin, Angela Demke Brown, and Ashvin Goel. 2021. Caracal: Contention Management with Deterministic Concurrency Control. In *SOSP*. ACM, 180–194.
- [53] Abdul Quamar, K. Ashwin Kumar, and Amol Deshpande. 2013. SWORD: scalable workload-aware data placement for transactional workloads. In *EDBT*. ACM, 430–441.
- [54] Kun Ren, Alexander Thomson, and Daniel J. Abadi. 2012. Lightweight Locking for Main Memory Database Systems. *Proc. VLDB Endow.* 6, 2 (2012), 145–156.
- [55] Marco Serafini, Rebecca Taft, Aaron J. Elmore, Andrew Pavlo, Ashraf Aboulnaga, and Michael Stonebraker. 2016. Clay: Fine-Grained Adaptive Partitioning for General Database Schemas. *Proc. VLDB Endow.* 10, 4 (2016), 445–456.
- [56] Zechao Shang, Feifei Li, Jeffrey Xu Yu, Zhiwei Zhang, and Hong Cheng. 2016. Graph Analytics Through Fine-Grained Parallelism. In *SIGMOD Conference*. ACM, 463–478.
- [57] Chunzhi Su, Natacha Crooks, Cong Ding, Lorenzo Alvisi, and Chao Xie. 2017. Bringing Modular Concurrency Control to the Next Level. In *SIGMOD Conference*. ACM, 283–297.
- [58] Xuebin Su, Hongzhi Wang, and Yan Zhang. 2021. Concurrency Control Based on Transaction Clustering. In *ICDE*. IEEE, 2195–2200.

- [59] Rebecca Taft, Essam Mansour, Marco Serafini, Jennie Duggan, Aaron J Elmore, Ashraf Aboulmaga, Andrew Pavlo, and Michael Stonebraker. 2014. E-store: Fine-grained elastic partitioning for distributed transaction processing systems. *Proceedings of the VLDB Endowment* 8, 3 (2014), 245–256.
- [60] Dixon Tang and Aaron J. Elmore. 2018. Toward Coordination-free and Reconfigurable Mixed Concurrency Control. In *USENIX Annual Technical Conference*. USENIX Association, 809–822.
- [61] Dixon Tang, Hao Jiang, and Aaron J. Elmore. 2017. Adaptive Concurrency Control: Despite the Looking Glass, One Concurrency Control Does Not Fit All. In *CIDR*. www.cidrdb.org.
- [62] Alexander Thomson, Thaddeus Diamond, Shu-Chun Weng, Kun Ren, Philip Shao, and Daniel J. Abadi. 2012. Calvin: fast distributed transactions for partitioned database systems. In *SIGMOD Conference*. ACM, 1–12.
- [63] Boyu Tian, Jiamin Huang, Barzan Mozafari, and Grant Schoenebeck. 2018. Contention-Aware Lock Scheduling for Transactional Databases. *Proc. VLDB Endow.* 11, 5 (2018), 648–662.
- [64] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. 2013. Speedy transactions in multicore in-memory databases. In *SOSP*. ACM, 18–32.
- [65] Donghui Wang, Yuxing Chen, Chengyao Jiang, Anqun Pan, Wei Jiang, Songli Wang, Hailin Lei, Chong Zhu, Lixiong Zheng, Wei Lu, Yunpeng Chai, Feng Zhang, and Xiaoyong Du. 2025. TXSQL: Lock Optimizations Towards High Contented Workloads (Extended Version). <https://doi.org/10.48550/arXiv.2504.06854> (2025).
- [66] Jia-Chen Wang, Ding Ding, Huan Wang, Conrad Christensen, Zhaoguo Wang, Haibo Chen, and Jinyang Li. 2021. Polyjuice: High-Performance Transactions via Learned Concurrency Control. In *OSDI*. USENIX Association, 198–216.
- [67] Chao Xie, Chunzhi Su, Cody Little, Lorenzo Alvisi, Manos Kapritsos, and Yang Wang. 2015. High-performance ACID via modular concurrency control. In *SOSP*. ACM, 279–294.
- [68] Maysam Yabandeh and Daniel Gómez Ferro. 2012. A critique of snapshot isolation. In *EuroSys*. ACM, 155–168.
- [69] Cong Yan and Alvin Cheung. 2016. Leveraging Lock Contention to Improve OLTP Application Performance. *Proc. VLDB Endow.* 9, 5 (2016), 444–455.
- [70] Xinan Yan, Linguan Yang, Hongbo Zhang, Xiayue Charles Lin, Bernard Wong, Kenneth Salem, and Tim Brecht. 2018. Carousel: Low-Latency Transaction Processing for Globally-Distributed Data. In *SIGMOD Conference*. ACM, 231–243.
- [71] Zhenkun Yang, Chuanhui Yang, Fusheng Han, Mingqiang Zhuang, Bing Yang, Zhifeng Yang, Xiaojun Cheng, Yuzhong Zhao, Wenhui Shi, Huafeng Xi, Huang Yu, Bin Liu, Yi Pan, Boxue Yin, Junquan Chen, and Quanqing Xu. 2022. OceanBase: A 707 Million tpmC Distributed Relational Database System. *Proc. VLDB Endow.* 15, 12 (2022), 3385–3397.
- [72] Xiangyao Yu, George Bezerra, Andrew Pavlo, Srinivas Devadas, and Michael Stonebraker. 2014. Staring into the Abyss: An Evaluation of Concurrency Control with One Thousand Cores. *Proc. VLDB Endow.* 8, 3 (2014), 209–220.
- [73] Xiangyao Yu, Andrew Pavlo, Daniel Sánchez, and Srinivas Devadas. 2016. TicToc: Time Traveling Optimistic Concurrency Control. In *SIGMOD Conference*. ACM, 1629–1642.
- [74] Xiangyao Yu, Yu Xia, Andrew Pavlo, Daniel Sánchez, Larry Rudolph, and Srinivas Devadas. 2018. Sundial: Harmonizing Concurrency Control and Caching in a Distributed OLTP Database Management System. *Proc. VLDB Endow.* 11, 10 (2018), 1289–1302.
- [75] Yuan Yuan, Kaibo Wang, Rubao Lee, Xiaoning Ding, Jing Xing, Spyros Blanas, and Xiaodong Zhang. 2016. BCC: Reducing False Aborts in Optimistic Concurrency Control with Low Cost for In-Memory Databases. *Proc. VLDB Endow.* 9, 6 (2016), 504–515.
- [76] Chao Zhang, Guoliang Li, and Tao Lv. 2024. HyBench: A New Benchmark for HTAP Databases. *Proc. VLDB Endow.* 17, 5 (2024), 939–951. <https://doi.org/10.14778/3641204.3641206>
- [77] Chao Zhang, Guoliang Li, Jintao Zhang, Xinning Zhang, and Jianhua Feng. 2024. HTAP Databases: A Survey. *IEEE Trans. Knowl. Data Eng.* 36, 11 (2024), 6410–6429. <https://doi.org/10.1109/TKDE.2024.3389693>
- [78] Ji Zhang, Yu Liu, Ke Zhou, Guoliang Li, Zhili Xiao, Bin Cheng, Jia Shu Xing, Yangtao Wang, Tianheng Cheng, Li Liu, Minwei Ran, and Zekang Li. 2019. An End-to-End Automatic Cloud Database Tuning System Using Deep Reinforcement Learning. In *SIGMOD Conference*. ACM, 415–432.
- [79] Ming Zhang, Yu Hua, and Zhijun Yang. 2024. Motor: Enabling Multi-Versioning for Distributed Transactions on Disaggregated Memory. In *OSDI*. USENIX Association, 801–819.
- [80] Yang Zhang, Russell Power, Siyuan Zhou, Yair Sovran, Marcos K. Aguilera, and Jinyang Li. 2013. Transaction chains: achieving serializability with low latency in geo-distributed storage systems. In *SOSP*. ACM, 276–291.
- [81] Hongyao Zhao, Jingyao Li, Wei Lu, Qian Zhang, Wanqing Yang, Jiajia Zhong, Meihui Zhang, Haixiang Li, Xiaoyong Du, and Anqun Pan. 2024. RCBench: an RDMA-enabled transaction framework for analyzing concurrency control algorithms. *VLDB J.* 33, 2 (2024), 543–567.
- [82] Qishi Zheng, Zhanhao Zhao, Wei Lu, Chang Yao, Yuxing Chen, Anqun Pan, and Xiaoyong Du. 2024. Lion: Minimizing Distributed Transactions Through Adaptive Replica Provision. In *ICDE*. IEEE, 2012–2025.