TDSQL: Tencent Distributed Database System

Yuxing Chen
Anqun Pan*
Hailin Lei
Tencent Inc.
{axingguchen,aaronpan, harlylei}@tencent.com

Anda Ye
Shuo Han
Yan Tang
Tencent Inc.
{andaye,shuohan, allenytang}@tencent.com

Wei Lu
Yunpeng Chai
Renmin University of China
lu-wei@ruc.edu.cn
ypchai@ruc.edu.cn

Feng Zhang
Xiaoyong Du
Renmin University of China
fengzhang@ruc.edu.cn
duyong@ruc.edu.cn

ABSTRACT

Distributed databases have become indispensable in contemporary computing and data processing, owing to their pivotal role in ensuring high availability and scalability. They effectively cater to the requirements of data management and high-concurrency access. However, developing a distributed database system that is well-suited for diverse application scenarios, particularly for large-scale applications, presents several challenges. These challenges include ensuring data consistency and achieving high levels of performance.

This paper presents TDSQL, a distributed database system that prioritizes core design principles of distributed systems, including high availability, strong consistency, and scalability. In particular, TDSQL has achieved high performance through over a decade of practical experience and optimization in various modules, such as the kernel, synchronous replication, and transaction processing, in large-scale application scenarios. By conducting the TPC-C benchmark test, TDSQL demonstrated outstanding performance, achieving a throughput of 814 million tpmC across 1650 database nodes, with a jitter rate of less than 0.2%. This jitter rate is an order of magnitude lower than the standard required, showcasing the system's stability and reliability. During the 8-hour TPC-C standard stress test, TDSQL successfully completed over 860 billion transactions and processed 40 trillion order details, with zero forced rollbacks and zero data inconsistency.

PVLDB Reference Format:
Yuxing Chen, Anqun Pan, Hailin Lei, Anda Ye, Shuo Han, Yan Tang, Wei Lu, Yunpeng Chai, Feng Zhang, and Xiaoyong Du. TDSQL: Tencent Distributed Database System. PVLDB, 17(12): 3869 - 3882, 2024.
doi:10.14778/3685800.3685812

1 INTRODUCTION

In recent years, the exponential growth in data volume and complexity has led to performance challenges for traditional centralized databases. Distributed database systems have emerged as a promising solution, offering high scalability [12, 15, 26, 55], availability [9, 60, 61], and performance [18, 32, 43, 50, 71, 80]. However, in distributed scenarios, new challenges arise, such as the trade-off between performance [46, 53] and consistency [8, 17, 54, 64, 70]. Tencent Distributed SQL (TDSQL), developed by Tencent Cloud [13], is a database system specifically designed to address the performance requirements of large-scale applications, e.g., e-commerce and banking scenarios, while also ensuring consistency, including strong synchronization [22, 44, 53, 59].

TDSQL is specifically designed to deliver high-performance and reliable databases for enterprises of all sizes. TDSQL has undergone iterative developments and meticulous optimizations to enhance its distributed functionalities, particularly focusing on improving distributed transaction processing capabilities. A noteworthy characteristic of TDSQL is its share-nothing architecture, which facilitates horizontal scaling across multiple nodes. This architecture empowers TDSQL to effectively handle substantial data volumes and manage high concurrency, with performance scaling that approaches linearity. Our TPC-C benchmark test has demonstrated the capability of TDSQL in efficiently processing data exceeding the 10 PB threshold while maintaining scalability on a single cluster equipped with over 100,000 physical cores. As an increasing number of financial industry enterprises, such as banks and securities firms, adopt TDSQL, it also offers a range of advanced features to ensure high availability. These features encompass auto-failover, data replication, primary-secondary switching, and recovery [79].

TDSQL was officially launched on Tencent Cloud [13] and has gained widespread adoption across 30,000 enterprises in various industries, including e-commerce, finance, government, and telecommunications. As a result, it has emerged as the market leader in China's distributed relational database market [30]. Significantly, TDSQL holds the distinction of being the first domestically developed database in China to be utilized in both internet-based distributed banking core systems and traditional banking core systems. It has also played a pioneering role in assisting domestic banks with migrating their core systems from centralized to distributed architectures. Currently, 7 out of the top 10 banks in China have already adopted TDSQL for services such as deposits, loans, payments, general ledger, and common operations.

This paper shares our experiences in designing, developing, and optimizing the TDSQL, a large-scale distributed database system. We conducted the official TPC-C benchmark test [58] on TDSQL. The results were impressive compared to open reports [57], as TDSQL achieved a remarkable performance of 814 million tpmC across 1650 database nodes (surpassing the second place by 15% overall and 8% per node) with a jitter rate less than 0.2%, which is an order of magnitude lower than the standard required. Throughout the 8-hour stress test, TDSQL demonstrated exceptional performance by handling a staggering volume of over 860 billion transactions and processing an astonishing 40 trillion order details, all without

This work is licensed under the Creative Commons BY-NC-ND 4.0 International License. Visit https://creativecommons.org/licenses/by-nc-nd/4.0/ to view a copy of this license. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org. Copyright is held by the owner/author(s). Publication rights licensed to the VLDB Endowment.
Proceedings of the VLDB Endowment, Vol. 17, No. 12 ISSN 2150-8097.
doi:10.14778/3685800.3685812

*Anqun Pan is the corresponding author.

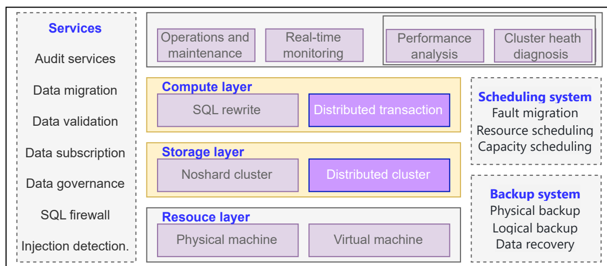


Figure 1: System design overview.

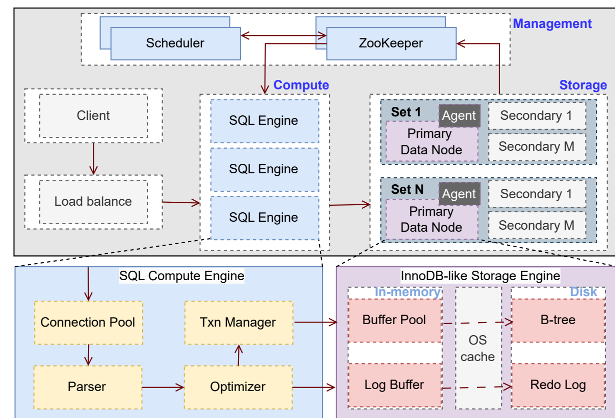


Figure 2: System core architecture.

a single transaction of forced rollback or inconsistency. In Tencent Cloud deployments, TDSQL proved to be highly cost-effective during the TPC-C test, with a remarkably low cost of 1.27 CNY/tpmC, which is only one-third of what comparable vendors offer.

The main contributions of this paper are summarized as follows:

- We introduce the core design and architecture of the TDSQL distributed database system. (Section 2)
- We detail implementations and optimizations of various modules in TDSQL. We focus on practical experiences gained from large-scale application scenarios. (Section 3)
- We conducted the official TPC-C benchmark test on TDSQL, achieving remarkable performance in terms of throughput (tmpC) and cost (price per tpmC). We also demonstrate our advantages via the real workload of the banks. (Section 4)

2 DESIGN OVERVIEW

Our primary focus is on designing TDSQL to facilitate rapid scale-out on commodity hardware, ensuring high performance for large-scale concurrent transactions and complex queries. Also, we aim to maintain data consistency and high availability even in the presence of hardware failures or other extreme scenarios. This section introduces our system design, core architecture, and applications.

2.1 System Design

Figure 1 shows the design overview of TDSQL, as follows:

Resource Layer. Starting from the bottom, the resource layer is the IaaS layer service, which can be physical machines or virtual machines, enabling TDSQL to manage the database instances.

Storage Layer. The storage layer, on top of the resource layer, emphasizes two storage forms in TDSQL: Noshard and distributed. Noshard is a centralized database, which supports high availability, data consistency, and 24/7 automatic failover. The distributed one additionally provides horizontal scalability.

Compute Layer. The compute layer, on top of the storage layer, serves as the computation engine. The compute layer primarily handles SQL-related processing, such as lexical analysis, syntax parsing, and SQL rewriting. This layer does not store data but focuses on real-time SQL computation, making it more CPU-intensive.

Management Layer. With management layer, DBAs can operate TDSQL via a web interface without the need to log in to the back-end. The management platform allows for the management of the distribution, scaling, and migration of compute and storage nodes.

Intelligent DBA. When faults occur, Intelligent DBA (e.g., [3]) nodes can try to analyze the causes of the faults, and identify reasons for slow SQL queries, sudden IO abnormalities, or machine failures. For example, SSDs experience an aging process, resulting in slower response times. It can also tune performance parameters [25, 40, 76].

Others. There are several other supporting and managing modules. For example, the scheduling system is responsible for overall resource scheduling, including adding and deleting database instances There is also a backup system, which serves as a cold backup center, supporting distributed storage systems such as HDFS and mountable distributed storage like Ceph [68]. We also provide auxiliary service modules, such as auditing, database migration services [19] between TDSQL and other databases, data validation, SQL firewall, injection detection, and other security-related modules.

2.2 System Core Architecture

TDSQL adopts a core architecture known as storage-compute separation [63]. As shown in Figure 2, the core architecture consists of three key parts: storage, compute, and management. TDSQL is built upon TXSQL [14], an open-source MySQL branch maintained by Tencent, which is fully compatible with MySQL's syntax and APIs. Figure 2 shows the key functionalities related to our optimizations in the SQL engine and data node. It includes numerous optimizations and fixes (e.g., consistency issues in § 3.3.1), extensive development (~3 million lines of code) of distributed features (e.g., physical replication in § 3.1 and lock optimizations in § 3.3.2), and performance enhancements specific to distributed properties (e.g., memory model optimization in § 3.3.4). A description of the core architecture is provided below.

Management Module includes the Scheduler cluster, which helps users automatically schedule and run various types of jobs [81], such as primary-secondary switches, managing resource additions in replication instances, or collecting monitoring data. TDSQL combines Scheduler and ZooKeeper [28] to activate specified resource plans within a time window, fulfilling various complex resource and job management requirements.

Storage Module consists of Set units, including Data nodes and Agents. Data nodes store replicas, and in high availability scenarios, a Set often contains one primary replica and two secondary replicas, across three physical nodes. Agents are auxiliary modules that

primarily monitor the health of Data nodes, report heartbeats to ZooKeeper, report resource usage, table size, access frequency, etc., monitor primary-secondary replication and data synchronization, and perform tasks such as migration, table consistency checks, and mirror backups. Data node refers to an InnoDB-like storage engine, which primarily includes a buffer pool for caching data and indexes, a redo log for primary-secondary synchronization and recovery.
Compute Module includes account authentication, connection management, SQL parsing, and route allocation. It is designed with a distributed architecture and parallel computation ability. It encapsulates functionalities such as permission verification, lexical analysis, syntax parsing, and distributed transaction control. In distributed scenarios, the TDSQL's SQL engine is also responsible for handling distributed transactions and maintaining globally unique auto-increment fields.

2.3 Application Scenarios

This paper primarily focuses on OLTP scenarios, including:
1) E-commerce: A significant application is in e-commerce transactions, akin to billing services, such as serving Tencent's "Honor of Kings" [56], a game with daily active users exceeding 100 million and daily transactions surpassing 10 million. These operations, including in-game item purchases, require real-time processing to ensure smooth gameplay and optimal user experience.
2) Finance: One of the key external financial clients is the bank. According to public data from the China Banking Association in 2019, Chinese banks process approximately 6.9 billion transactions daily, including deposits, withdrawals, transfers, and payments. These transactions require real-time processing and recording in the database. Banks also manage a vast amount of customer information, including personal details, account information, and credit data.
3) Online payment: TDSQL facilitates internal WeChat red packet payments within Tencent. Users can send red packets (monetary blessings) to others via WeChat Pay [65]. During the traditional Chinese New Year's Eve, we successfully handled a peak TPS of 14 million for red packets, with a total number of red packet exchanges reaching billions on that particular day. Moreover, with a monthly active user base exceeding 1.3 billion, WeChat Pay's daily transaction volume has already surpassed one billion.

TDSQL has been designed to meet the demands of these critical applications, featuring over 99.999% availability, strong data consistency [8, 22], high scalability through a share-nothing architecture, and high performance at a low cost.

3 IMPLEMENTATION AND OPTIMIZATION

In this section, we will provide a detailed explanation of the implementation and optimization of our key features. An overview of these implementations and optimizations is depicted in Figure 3.

3.1 High Availability

3.1.1 Physical Replication. Traditional logical replication, such as the key feature found in MySQL [46], offers a comprehensive ecosystem and supports various storage engines. However, its design, which relies on independent server-level logs, necessitates XA transaction coordination with the engine layer. In this design, a transaction requires two *fsync* function calls for persistence, and the sequential writing of binlog files can easily become a bottleneck, limiting system throughput. Despite attempts to reduce latency through methods like parallel replication [34, 79], MySQL and similar solutions still struggle to effectively reduce latency for large transactions or DDL operations. This limitation arises from the fact that binlog is written to the file only upon transaction commit. Moreover, despite ongoing efforts by the MySQL community to enhance logical replication, numerous bugs [42, 78] (including those introduced by new features) often result in data inconsistencies between primary and secondary replicas. Binlog replication fails to detect data inconsistencies caused by unexpected events like primary-secondary switches. When replication is interrupted, the general approach is to skip the problematic part or redo the secondary replica, both of which incur significant costs.

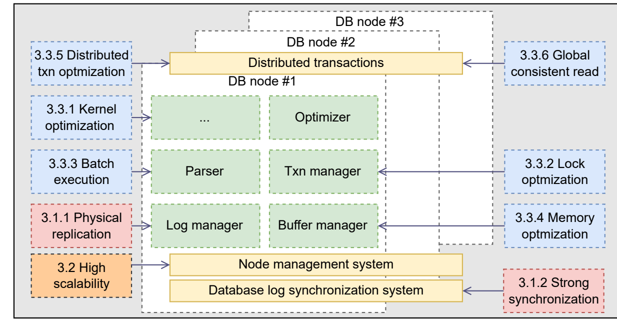


Figure 3: Overview of system optimizations.

Solution. To address this, TDSQL has developed a solution for physical replication, which utilizes redo logs for synchronization among multiple nodes, offering three key benefits: (i) Transaction commits now require only one commit log persistence (one fsync). Without the binlog bottleneck, we can significantly enhance the DML throughput of both individual nodes and entire clusters. (ii) Data updates can be synchronized while execution is in progress. Typically, when a statement is executed on the primary database, the corresponding log is simultaneously transmitted to the secondary replica and begins execution [7, 10]. (iii) The solution ensures the correctness of each applied log on the physical storage and guarantees consistency between the primary and secondary replicas via strong synchronization (will be discussed later). Figure 4 shows the architecture of physical replication with a three-replica setup:
1) Primary Replica: It contains the newest data and handles all read and write requests. During the execution of a specific transaction, the User session generates transaction logs, which are then copied to the global Log buffer. A dedicated Log writer thread writes these logs to the Log file and copies them to a Copy buffer circular queue. Additionally, a background Log flusher thread fsyncs the logs to the disk for durability. When a User session commits a transaction, it does not immediately return an OK response. Instead, it releases its user thread and joins a pending Commit queue. It waits for the logs to be sent to the secondary replica or log replica and receives an ACK response before actually committing the transaction and sending an OK response. To facilitate this process, a Listener thread continuously monitors the UDP package from the secondary replica. Upon receiving the corresponding ACK response, it parses the

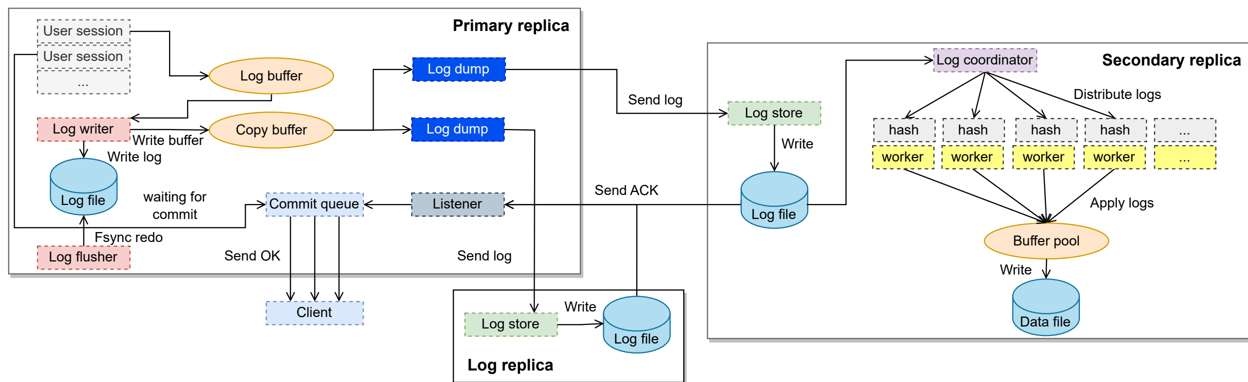


Figure 4: Architecture of physical replication.

ACK Log Sequence Number (LSN) and checks the Commit queue. If the ACK LSN is greater than or equal to the commit LSN, it retrieves the corresponding session and sends an OK response to the corresponding socket, indicating that the transaction has been successfully synchronized.

2) Secondary Replica: It contains the complete dataset. Upon establishing a connection with the primary replica, the secondary replica initiates a Log dump thread on the primary replica. This thread is responsible for sending logs to the secondary replica. The Log dump thread reads logs from the Copy buffer based on the last read LSN. If the Copy Buffer is overwritten, the logs are read from the Log file instead. The secondary replica stores these logs in its local disk and returns an ACK response to the primary replica. Upon receiving the logs from the primary replica, the Log store thread performs checksum verification and stores the logs in the local Log file. It then uses the UDP protocol to send the currently stored log LSN back to the primary replica. Additionally, it wakes up the Log apply thread, which consists of a Log coordinator thread and multiple worker threads [51]. The Log coordinator thread reads a batch of logs (default size is 32MB), parses them, and distributes them to different worker threads based on the hash of space_id (tablespace identifier) and page_no (page number). This ensures that logs belonging to the same data page are executed by the same worker thread. The Log coordinator thread wakes up the worker threads and asynchronously reads the next batch of logs for parsing, storing them in a backup worker hash. Ideally, this setup forms a pipeline for reading, parsing, and applying logs, thereby improving overall synchronization efficiency.

3) Log Replica: Similar to the secondary replica in log processing, the log replica serves primarily for high availability purposes and does not need to apply the logs. Its main function is to store the latest logs for backup and recovery. As the log replica only focuses on log synchronization, it consumes minimal CPU resources, utilizing around 80% of a single core. Also, it often requires less than 8GB of memory (e.g., in our TPC-C test), making it highly efficient in terms of resource consumption.

Optimization & Novelty. To address the issue during planned switches, we implement a buffer pool warm-up strategy in advance [36]. Specifically, the primary replica asynchronously dumps snapshot information of the buffer pool, identifies the hot range of B+Tree data, and shares this snapshot information with the secondary replica. The secondary replica then loads the snapshot information and performs the warm-up process asynchronously by directly scanning the B+Tree, without executing SQL statements. By adopting this approach, the impact on QPS is almost negligible, with less than a 5% decrease within the first minute. This proactive warm-up strategy significantly mitigates the performance impact during primary-secondary switching.

When comparing physical replication to binlog-based logical replication, it is generally observed that adopting physical replication results in enhanced performance, especially in terms of write performance and scenarios involving small transaction updates. For example, through testing under the SysBench [33] update scenario and TPC-C [58] benchmark, we observed significant performance improvements of at least 200% and 100%, respectively.

3.1.2 Strong Synchronization. Requesting stronger consistency in databases can potentially impact performance, and many databases prioritize performance over strict correctness [22]. For instance, MySQL supports asynchronous replication, which can introduce data consistency issues. This lack of consistency is unacceptable in scenarios that demand zero inconsistency [5], such as the financial industry. Semi-synchronous replication [15] offers a better solution, as it ensures data replication through strong synchronization. However, if the replica fails to respond within a specified time, it degrades to an asynchronous mode, which still presents consistency concerns. Moreover, semi-synchronous replication has its own correctness issues and performance deficiencies. For example, in scenarios involving inter-data center communication delays and high network jitter, there can be significant spikes in request processing. Consequently, these native solutions fail to meet the requirements of financial scenarios.

Solution. TDSQL introduces a robust synchronous replication mechanism based on the Raft protocol [48]. In this approach, when the primary replica receives a query request, it waits for successful responses from a majority of the replicas before acknowledging success to the client. For instance, in a configuration with one primary and two secondary replicas, once a commit request reaches the primary replica, it must await a successful response from at

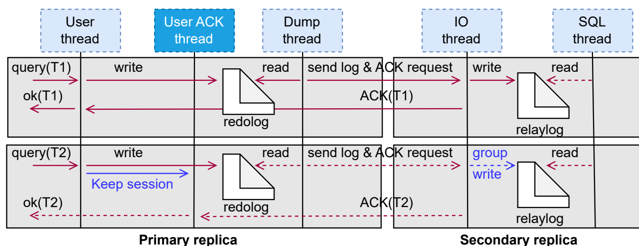


Figure 5: Synchronous vs. asynchronous user threads.

least one of the secondary replicas before confirming success to the client. This strong synchronization is a fundamental feature and plays a critical role in ensuring data consistency.

Optimization & Novelty. However, the adoption of the waiting mechanism for strong synchronization leads to a notable decrease in performance compared to the asynchronous mode, often resulting in only half of the original throughput. To mitigate this impact, we have implemented a thread pool that enables the transition from the previous serial execution of waiting for replica responses to a dedicated thread called the User ACK thread. This approach allows the user threads to be freed up to handle other user connections, thereby improving overall system efficiency and performance.

In Figure 5, we can observe the previous behavior of T1, where it had to wait for multiple serial synchronous interactions to complete before releasing the User thread. However, with the introduction of the User thread asynchronous method, the write operation in T2 only requires the User ACK thread to hold the session. This allows the User thread to be released, thereby increasing concurrency, i.e., the user thread no longer needs to wait for the dashed-line interactions. To further reduce latency, we have implemented group writing to replay logs in secondary replicas. This optimization contributes to improved performance and reduced latency. Compared to semi-synchronous replication, which is significantly affected by network jitter in inter-data center communication, the optimized strong synchronous replication in TDSQL demonstrates comparable performance and latency to the asynchronous mode.

Failover. After the primary replica fails, the secondary replica and log replica are checked. If the secondary replica has fewer logs than the log replica, it connects to the log replica and reads the missing logs. After applying the logs [45], the secondary replica forcefully promotes itself to become the new primary using an SQL statement. Any unfinished transactions are rolled back, and various background threads are started to initialize the transaction system. After the secondary replica is promoted to the new primary replica, the log replica connects to the new primary.

3.2 High Scalability in TDSQL

In situations where the performance or capacity of an instance is inadequate to meet workload requirements, vertical scaling can be employed to expand the configuration of the instance. Conversely, if the instance's capacity significantly exceeds the workload demands, scaling down can be implemented to reduce the configuration of the instance. In the case of a distributed instance, when the computing power or capacity falls short of the workload requirements, scaling out can be achieved by adding nodes to increase storage or computing capacity. Both vertical and horizontal scaling operations can be performed without interrupting services.

Both time-based and range-based sharding approaches may result in data skew, leading to imbalanced load and data capacity among shards. This imbalance occurs because data often exhibits distinct hot and cold characteristics, where the likelihood of accessing recent orders is much higher than older ones, for instance. TDSQL typically adopts a hash-based sharding scheme. In this approach, a specific field is selected for modulo calculation, ensuring that data is evenly distributed across different physical devices. By evenly distributing the data, load balancing is achieved, enhancing system availability and performance.

In practical deployments, the initial setup may involve deploying a single Manager node as a cost-effective solution due to factors such as a small cluster size, limited number of instances, or low workload requirements. However, as the need for system availability increases, it may become necessary to scale the management components. It is worth noting that even if all Manager nodes go offline, the Data nodes are not heavily reliant on them. This means that read and write requests to the Data nodes remain unaffected when the Manager nodes are not functioning, except for operations related to primary-secondary switching, scaling, and scheduling. Scaling can be performed manually or automatically. Manual scaling involves pre-scaling based on periodic workload scenarios, where adjustments are made in anticipation of expected changes in workload. On the other hand, automatic scaling relies on predefined rules, such as high CPU and disk utilization, to trigger scaling operations automatically.

The scaling process does not interrupt service continuity. Specifically, the scaling process involves several essential steps akin to those involved in the addition of replication [1, 20], including data synchronization, data verification, route updating, and deletion of redundant data. (1) Data synchronization synchronizes data to the new Set. If the user does not specify which shards to migrate from which source set to the new set, the system uses an average splitting method, such as consistent hash to achieve load balancing. (2) Data verification continuously catches up with data while continuously verifying data. This process may last for a while. When the delay difference between two synchronizations is very close, for example, we set a 5-second threshold. When we find that it has caught up within 5 seconds, we will enter the route update stage. (3) For route updating, it is necessary to freeze write requests. At this time, if there are writes from the workload, the system freezes this request and retries automatically after the route switch is successful. But this period will be very short. (4) The redundant data can be deleted after the route has been updated. Throughout these processes, it is crucial to consider factors such as load balancing and maintaining data consistency during updates and deletions.

Optimization & Novelty. To address these considerations, we have implemented storage layer partition blocking at the storage layer. This ensures that data will not be written from the compute layer during route updates, preventing any potential inconsistencies. During the deletion of redundant data, we perform SQL rewriting at the compute layer. SQL at the computing layer will access data according to the routing table. SQL rewriting refers to accessing data according to the new routing table when finding the partition

of the data. For data that is blocked and migrated to other nodes, the access plan will be rewritten according to the new routing table, so the blocked data will not be accessed. This ensures that even if redundant data exists, queries will not return unexpected or incorrect results. To manage the deletion effectively, we employ delayed deletion, which allows for a gradual deletion process, minimizing any significant IO jitters that could impact live workloads. Also, with issues during data synchronization, debugging and repairing can be performed using the redundant data.

3.3 Optimizaiton towards High Performance

In addition to the previously discussed aspects of high availability, strong consistency, and scalability, we have made significant efforts to enhance stability and performance through various fixes, implementations, and optimizations. In the following parts, we will provide a comprehensive description of these improvements.

3.3.1 Kernel Optimizations for Consistency. TDSQL has undergone a series of bug fixes to ensure its consistency and stability, as the performance becomes meaningless without the assurance of consistency. To verify the correctness of TDSQL, we have implemented a widely used standard banking transfer function test, similar to the durability test in the TPC-C ACID test [58] or the Jepsen test [6, 27, 31]. The test program begins by loading a substantial amount of fictitious account information into the system. It then initiates numerous connections and concurrently executes a multitude of money transfer transactions between users. The objective of this test is to ensure that the total balance of all accounts remains consistent and that each row on both the primary and secondary replicas is identical. To comprehensively assess the system's resilience, the test program intermittently and randomly terminates various components during the test runtime, including database processes, gateway processes, agents, and even itself on the primary replica. This testing has enabled us to identify multiple bugs in TDSQL. A number of these bugs were likewise detected in MySQL and reported to the official team, subsequently confirmed, and fixed [42, 78], including bugs found in the combination of distributed transaction processing (e.g., #109831, #87130, #84499), replication (e.g., #87560, #87389), and thread safety (e.g., #99643).

3.3.2 Lock Optimization. To effectively manage the parallel execution of a large number of transactions on each node, it is essential to address the bottlenecks caused by locking [24]. To achieve maximum concurrency, TDSQL has implemented extensive optimizations. These optimizations target various levels of locks, including table, row, transaction ID, and Purge. The following will delve into each optimization in detail, aiming to simplify the locking mechanism, reduce conflicts, and improve overall system efficiency.

Removal of Table-level Locks. InnoDB's table-level locks can cause unnecessary conflicts and complexity due to the existing Metadata Locks (MDL) at the server level. These locks, except for AUTO_INC, have corresponding MDL locks, allowing for their safe removal. However, in the case of unfinished transactions recovered from crashes, MDL locks are necessary for transaction rollback. During crash recovery, a Thread Handler (THD) object is generated for each recovered transaction, and an MDL lock is added. This process is delayed until the MDL lock system is fully established, and the MDL lock must be released once the transaction rollback is completed to ensure ongoing transactions.

Partitioning of Row-level Locks. InnoDB uses locking mechanisms during update operations, storing lock objects in a hash table and requiring a global lock for integrity. These lock objects are stored in a hash table, where the key value is derived from the space_id and page_no of the updated record. This causes contention on the associated mutex. A solution is to divide the global lock into multiple mutexes, each protecting a specific hash cell segment, reducing contention. This division is based on the natural distribution of lock objects across different hash cells, determined by the space_id and page_no. Deadlock detection uses a separate thread, acquiring mutexes involved in the deadlock cycle and performing duplicate verification due to the absence of a global lock. This optimization improves the performance and scalability of the locking mechanism, especially in DML operations.

Lock-free Transaction ID Management. InnoDB uses a global transaction ID array for the MVCC mechanism [69], tracking transaction IDs throughout their lifecycle. However, adding and removing IDs from the array requires a global lock, introducing performance overhead, especially in short transactions where conflicts are more likely. To address this, we have implemented a lock-free hash table [52], focusing on three fundamental operations: insertion, deletion, and traversal of the hash table. Instead of relying on linked list operations, we have replaced not only the global array but also the read-write transaction list and transaction object trx_t::no list with hash table operations. When a transaction is initiated and assigned a transaction ID, it is inserted into the lock-free hash table. The transaction ID serves as the key value in the hash table, while the stored object includes the transaction number (distinct from the transaction ID allocated during the prepare phase), the transaction object itself, and the mutex responsible for protecting the object. During the transaction prepare phase, the transaction object trx_t::no is directly updated. Upon transaction commit, the corresponding object is removed from the lock-free hash table. To determine the global minimum transaction ID, traversal of the lock-free hash table becomes necessary. However, when the hash table is considerably large, the traversal process can consume a significant amount of CPU resources. To address this, the hash structure has been optimized for CPU performance, such as aligning cache lines and relaxing unnecessary memory barriers.

Lock-free Scheduling of Parallel Purge Threads. InnoDB employs a single Purge thread to handle the cleanup of undo logs and discarded index records. However, this approach often proves inefficient under high-load conditions, during the Purge, two specific areas become hotspots for lock contention: (i) acquiring a global lock when waking up worker threads and (ii) acquiring a global lock when worker threads retrieve tasks from the task queue.

The contention for these locks introduces additional CPU overhead, which hampers the overall efficiency of the Purge thread. To address this, we replace the large lock with smaller locks and transform the lock queue into a lock-free queue, enabling multiple Purge threads. Specifically, during Purge execution, the main thread first parses the undo logs and generates a collection of tasks. Then, the worker threads are awakened. Since the number of worker threads remains constant throughout the runtime of the instance, it becomes feasible to convert the global lock into smaller locks held

by the worker threads for the wake-up operation. Regarding the tasks in the task queue, their number does not exceed the number of worker threads. Therefore, it becomes possible to assign specific partitions in the task queue to each worker thread, thereby avoiding conflicts with other worker threads.

3.3.3 Batch Execution with tdsql_returning Syntax. Through analysis of workload scenarios (e.g., TPC-C), we have observed that certain transactions frequently involve a sequence of operations that combine update+ or delete+ with select on the same data item. This execution often necessitates two separate SQL statements. To optimize this, we have introduced an approach known as the tdsql_returning syntax. This method allows us to directly return the results of updates and deletions, thereby reducing the overhead of an additional select statement.

The optimization has been implemented by: (i) Support for the tdsql_returning syntax is extended in update and delete statements. This syntax is then followed by the select query field syntax, which is semantically equivalent to concatenating the select fields of the update or delete table after tdsql_returning. (ii) During the syntax analysis phase, preprocessing is performed on the newly added syntax nodes. Before execution, the query fields in the syntax tree are checked for the presence of a tdsql_returning node. If such a node exists, preprocessing is conducted on the query fields, which may involve replacing asterisks with the appropriate fields. (iii) During the execution phase, the decision to send the result set to the client is determined based on the syntax tree. If necessary, the metadata of the result set fields is sent first, followed by transmitting the processed data to the client.

3.3.4 Memory Optimization. In our upcoming TPC-C benchmark tests, we have a demanding requirement for the database to support concurrent access from 640 million clients. To achieve this, each database node must be capable of efficiently handling hundreds of thousands of database connections simultaneously.

Optimizing Network Model for User Connections. Traditional methods like one thread per connection or using a thread pool present challenges: (i) Limited machine ports due to each connection needing access to all Data nodes. (ii) High network latency and CPU strain due to numerous small network packets. (iii) Performance decline after 20,000 concurrent connections due to thread-switching overhead. (iv) Significant memory overhead, with each user connection in the TDSQL kernel consuming about 3MB of memory, leading to low cache hit rates and poor performance. (v) Limited horizontal scalability as network and thread resource overhead grows exponentially with more nodes.

To address this, optimization of the network model is proposed as follows: (i) Transformation of the database transmission protocol: We introduce an asynchronous multiplexing approach by adding a *connid* field in the protocol header. This enables the maintenance of multiple client transaction sessions within a single TCP connection. Consequently, connection reuse at the session layer is achieved, reducing the required TCP connections per primary Data node from 1,200 to 24. As a result in our TPC-C test, the number of connections per Data node is significantly reduced to only 11,880. (ii) Refactoring of the network module: We refactor the network module of the database from blocking I/O to non-blocking I/O based on the *epoll* mechanism. This involves implementing dedicated read and write threads to handle data packet read, aggregation, unpacking, and other operations. By decoupling network I/O from workload SQL operations and utilizing lock-free memory queues for efficient data communication, we significantly improve packet aggregation capabilities. This ensures that the overhead of network I/O remains within 5% of the total CPU overhead. As the testing cluster scales, the number of connections grows linearly without exponential growth, providing a solid foundation for horizontal scalability. (iii) Additional optimizations for connection reuse: We have implemented various optimizations related to connection reuse, such as determining the maximum number of clients that can reuse a single connection, defining mechanisms for terminating specific sessions, and addressing packet interleaving issues.

Optimizing Memory Utilization and Stability. Memory-related challenges arise in two main areas within the large-scale system: (i) Each THD independently caches resources like stored procedures, leading to substantial memory overhead under high concurrency scenarios. (ii) Dynamic memory allocation during SQL execution can occasionally cause spikes in memory usage, particularly when resources are limited. To tackle these challenges, TDSQL implements the following optimizations.

Resource sharing has been transitioned from the session level to a global level. Previously, if parsing a stored procedure required 500K of memory and there were 50,000 connections, it would consume a staggering 23 GB of memory. However, by implementing global sharing and limiting the actual concurrency to 256, the required cache amount is reduced to within 1 GB, resulting in approximately 0.5 GB of actual memory consumption. To resolve resource conflicts in a multi-threaded environment after achieving global optimization, lock-free hash mechanisms are employed, as discussed previously. These mechanisms ensure performance comparable to that of a local cache while effectively managing resource conflicts. In our TPC-C test, the overhead of shared resources, which initially approached 300GB, has been successfully reduced to approximately 30 GB. This substantial reduction greatly enhances memory utilization within the system.

Another noteworthy aspect is the periodic fluctuation in memory usage observed during the stress tests, occasionally resulting in spikes. This phenomenon also affects the jitter rate requirements of the TPC-C test. To address this, we employ the use of *ebpf* to capture the stack traces of all memory allocation points throughout these stress tests. Then, we implement memory pooling techniques to consolidate all dynamic allocations, effectively eliminating the spikes caused by memory management issues. Different from MySQL's memory pooling, we, based on latency profiling, found parts that impact performance during memory allocation. Then, allocating these more impactful modules through the memory pool can effectively prevent memory stalls. This approach reduces transaction latency (by 5% by TPC-C test) as well as is successful in resolving occasional stalls that occur during the memory allocation phase, which effectively reduces throughput jitter.

3.3.5 Distributed Transaction Optimization. Addressing distributed transactions introduces a set of challenges that are not encountered in a single-node environment. It necessitates careful consideration of various complex scenarios and the ability to effectively handle them. For instance, it requires managing unreliable networks

across multiple nodes, handling primary-secondary switching, and addressing node failures while ensuring data consistency even in diverse fault recovery environments.

TDSQL incorporates a widely adopted two-phase commit (2PC) protocol [4] for distributed transactions. The native XA solution, while widely used, can result in data inconsistency and loss during primary-secondary failover [78]. To improve, we leverage the 2PC protocol for cross-node transactions and optimize the commit process using group commit techniques. For transactions that do not span multiple nodes, we can utilize the one-phase commit protocol, eliminating the need for prepare and coordination steps. This further streamlines the transaction process. In comparison to the native XA solution, which achieves only about half the per-node tpmC, TDSQL has undergone a series of optimizations and bug fixes. In a 1650-node TPC-C test, TDSQL maintains over 75% of the per-node tpmC observed in a single-node test and maintains over 93% of the per-node tpmC observed in a three-node test.

3.3.6 Global Consistent Read. In a single-node environment, established solutions exist for attaining read consistency. For instance, InnoDB employs the MVCC mechanism, which relies on the visibility view (Read view) of the active transaction chain [46]. Similarly, RocksDB determines visibility based on LSN (Log Sequence Number) [18]. However, in a distributed setting, addressing the global consistent read becomes more intricate due to factors such as partition locations and network conditions. Achieving global consistent read in such scenarios necessitates the consideration of multiple complex factors. Failure to achieve global consistent read is often regarded as a critical concern, particularly in the financial domain where accurate accounting is paramount. TDSQL specifically addresses the challenges of partition locations and network conditions in ensuring globally consistent reads by incorporating a high-performance global transaction component within its kernel. This component is responsible for node management and Global Transaction Sequence (GTS). In this system, each transaction is assigned a GTS by a centralized manager node during both the initiation and commit stages. When a query request scans rows in the prepare phase, it waits for the corresponding GTS to commit before conducting visibility checks. During these checks, the system ensures that the GTS of the current row is either equal to or less than the GTS of the query, thereby guaranteeing visibility. If the GTS of the row is greater than the GTS of the query, the row is considered invisible. However, it's important to note that a transaction can read its own write, regardless of the GTS.

The frequency of visibility checks is notably high, and it involves obtaining transaction status based on transaction IDs. And, this approach can potentially result in CPU resource wastage due to uncertain states. Therefore, optimizations are crucial for efficient implementation. One optimization involves utilizing the GTS of the prepare state instead of the commit state to enable early visibility judgments and reduce waiting time [21]. Additionally, an efficient mapping mechanism is implemented to unify transaction IDs and global unique timestamps, minimizing interference with TDSQL's log and file system while ensuring high performance. Similar to MySQL's MVCC, periodic cleanup of logs and data is performed based on the oldest active transaction [35]. This optimization enhances query efficiency and conciseness. Furthermore, we have

Table 1: Machine/instance types.

	Type I	Type II
CPU model	Intel(R) Xeon(R) Platinum 8631HC	Intel(R) Xeon(R) Platinum 8255C
Processor/core/thread	2/48/96	2/48/96
Memory	288 GB	768 GB
OS data disk (HDD)	100 GB	447 GB
Data storage (SSD)	500 GB	12 x 3.5 TB NVMe
Node number	990 instances	1653 machines
Tencent Cloud	S5.24XLARGE288	HYI12A

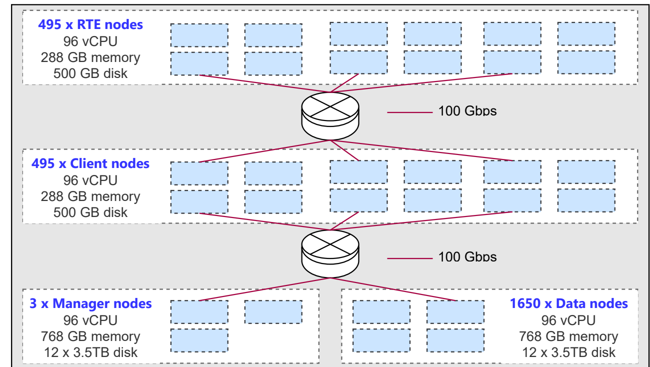


Figure 6: System configuration.

optimized the communication frequency in the two-phase commit and global transaction processes, aiming to minimize the overhead associated with introducing global timestamps. The performance impact of enabling global consistent read is less than 5%.

4 EVALUATION

4.1 Experimental setup

Methodology. In our TPC-C experiment, the testing cluster consists of four roles: (i) Remote Terminal Emulator (RTE) nodes, which simulate the clients, (ii) Client nodes, responsible for receiving client requests and communicating with the database, (iii) Manager nodes, which handle metadata such as routing tables, and (iv) Data nodes, deployed for storing and computing data. The Client node receives all requests from RTE and distributes them to the Data node based on the shard key. Upon receiving a request, the corresponding Data node executes the SQL statements and sends the results back to the Client node. The Client node then aggregates all the results and returns them to the RTE through the Web Server.

Configuration. Two types of machines/instances are selected and their hardware parameters are listed in Table 1. An overview of the system architecture is provided in Figure 6. In our deployment, we utilized a specific configuration consisting of 495 Type I instances for RTE, 495 Type I instances for Client nodes, 3 Type II machines for Manager nodes, and 1650 Type II machines for Data nodes. The Client, Manager, and Data nodes are integral components of the system under test (SUT), and their costs were calculated for the benchmark evaluation [57]. Each Data node is equipped with an Intel(R) Xeon(R) Platinum 8255C CPU @ 2.50GHz x 2 (96 vCPU), 768

Table 2: Initial statistic and 8-hour consumption (Size in GB).

Table	Rows	Data	Index	8H Space
Warehouse	64,003,500	15.00		
District	640,035,000	138.82		
Customer	1,920,105M	2,378,710	255,114	
Item	100,000	30.74		
Stock	6,400,350M	4,159,728		
New-orders	576,031.5M	35,739		
Orders	1,920,105M	172,919	115,766	35,224
Order-line	~19,201,054M	2,646,499		539,098
History	1,920,105M	273,400		55,692

Table 3: Prediction of 60-day disk consumption (Size in GB).

Free space	9,875	Storage per node	42,000
Dynamic space	3,092,819	Total storage	69,300,000
Static space	6,945,244	60-Day space	44,746,160
Daily growth	630,015	Remaining space	24,553,839

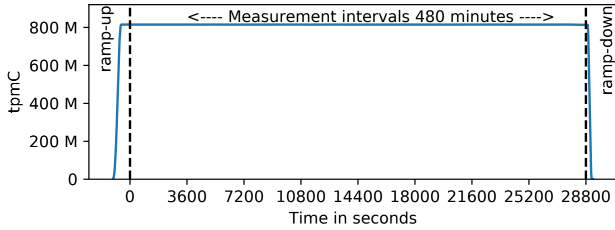


Figure 7: 8-hour run new-order throughput versus elapsed time with 814,854,791 tpmC and less than 0.2% jitter rate.

GB Memory, and 12 SSD disks (3.5 TB NVMe each, RAID0). These disks are formatted and managed by Tencent tlinux 2.2. The database used is TDSQL v10.3 Enterprise Pro Edition with Partitioning and Physical Replication. All nodes are deployed on Tencent Cloud. Overall, the entire SUT comprises more than 100,000 physical cores (200,000 threads), 1.4 PB memory, and 70 PB disk storage.

Benchmark. To ensure the accuracy and reliability of our evaluation, we have followed the standard TPC-C specifications [58]. These specifications provide a comprehensive guideline for implementing the TPC-C benchmark, which allows for fair and reproducible performance evaluations.

4.2 Initial Data and Disk Usage Analysis

The initial database is composed of 64,003,500 Warehouses, and the scaling of other tables follows a specific proportion as outlined in the TPC-C specification [58]. Table 2 presents the statistics of the tables, including cardinalities, data size, and index size.

Upon importing the data, we also assessed the disk usage overhead. The Orders, Order-line, and History tables are dynamic tables, meaning that their number of rows and disk usage will increase as the test progresses. Following the guidelines outlined in the TPC-C standard [58], we estimated the disk usage for an 8-hour duration based on the tpmC (transactions per minute) we obtained. The disk capacity is designed to accommodate data growth for runs lasting over 60 days, as indicated in Table 3. Specifically, our system has a disk capacity of nearly 70 PB, with an initial data size of 10 PB. Based on our estimations, we anticipate that the disk usage will reach approximately 45 PB within the 60-day timeframe.

Table 4: Key optimization breakdown by TPC-C.

OPT1: § 3.1.1	OPT2: § 3.3.2	OPT3: § 3.3.4
Physical replication	Lock optimization	Memory Optimization
100% tpmC ↑	15% tpmC ↑	90% memory ↓

4.3 8-Hour Stability Test

Figure 7 depicts the relationship between the tpmC of New-order transactions and the elapsed time. The benchmark test initiates with a ramp-up period, during which the system gradually increases its throughput. After approximately 20 minutes of ramp-up, the throughput reaches a stable state and remains consistent. The stable run is measured effectively over a duration of 8 hours (480 minutes), starting from 20 minutes after the benchmark test's initiation and continuing until the 500th minute. Following a ramp-down period of approximately 20 minutes, the throughput begins to decline until it eventually reaches 0 tpmC. This pattern illustrates the complete cycle of the benchmark test, encompassing the ramp-up, stable run, and ramp-down phases

Comparison with other TPC-C report results. During the 8-hour stability period, the system demonstrated an impressive throughput of 814,854,791 tpmC for New-order transactions across 1650 Data nodes. This remarkable achievement currently holds the first place in publicly reported TPC-C benchmarks [57], surpassing the second-best related work by over 15% in overall performance and 8% per node. In addition to the 0.1-second delay added to each transaction as required by the TPC-C, the average latency and 90th percentile latency for each transaction are both below 0.01 seconds and 0.02 seconds, respectively, demonstrating the system's exceptional responsiveness. When compared to public reports [57], our average and 90th percentile latency outperforms related works. To provide a visual representation of the response time distribution for New-order transactions, Figure 9a illustrates the New-order frequency distribution of response times.

TDSQL exhibited remarkable consistency and stability in this large-scale cluster, with a jitter rate of less than 0.2%, which is an order of magnitude lower than the standard required. This indicates the system's ability to maintain a steady performance over the entire 8-hour duration. Additionally, the test run ensured zero data inconsistency across more than 860 billion transactions and nearly 40 trillion shopping orders. Remarkably, there were no unexpected rollbacks observed for any transactions within the 8-hour period. Furthermore, we observed that each node in the system could handle up to 1.8 million queries per second (QPS), highlighting the impressive processing capabilities of the individual nodes.

Performance Optimization Breakdown. The worth-mentioning optimizations are (1) OPT1: Implementation and optimization of physical replication, (2) OPT2: Performance savings brought by various lock optimizations, and (3) OPT3: Significant high scalability performance guarantee brought by memory optimization.

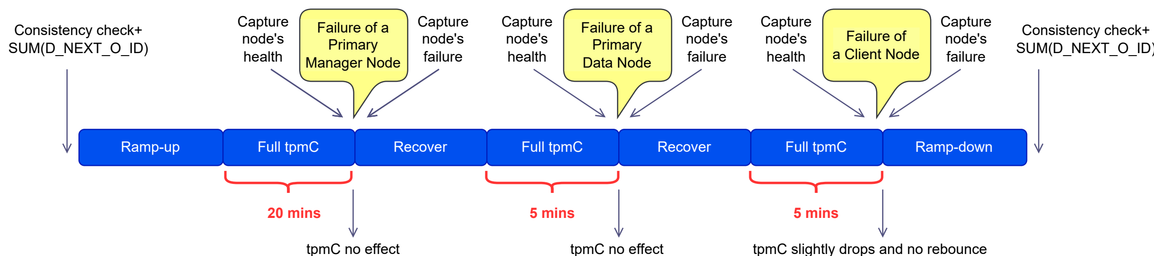
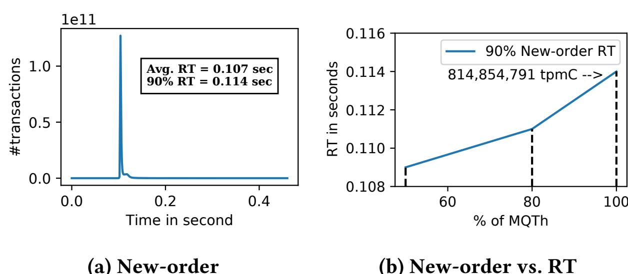
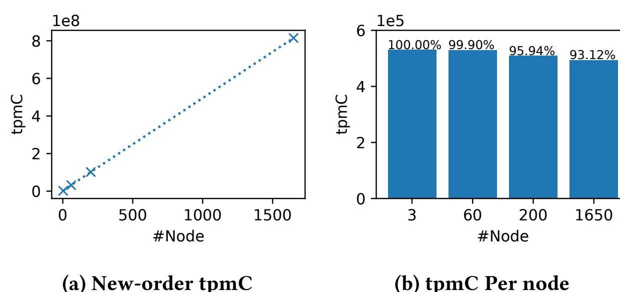


Figure 8: The procedures of durability test.



(a) New-order **(b) New-order vs. RT**

Figure 9: Frequency distribution of response times (RT).



(a) New-order tpmC **(b) tpmC Per node**

Figure 10: Scalability.

Table 4 summarizes the key optimization breakdown for performance improvement by TPC-C. Compared to the original physical replication, our implementation of OPT1 significantly improves performance, by over 100%, and even more than 2 times in SysBench write-intensive scenarios. Various lock optimizations contribute to an approximately 15% improvement (30% in high-contented scenarios). The memory savings of OPT3 are significant under the InnoDB B+tree. The larger the buffer pool reserved for InnoDB, the better the performance. We optimize the network model to save more than 90% of memory usage for connections, and as the number of nodes increases, memory usage only increases linearly.

4.4 ACID Test

TDSQL has passed the Atomicity, Consistency, and Isolation tests as mandated by TPC-C. These tests serve to validate the fundamental capabilities of the database. Furthermore, the Consistency test is not only conducted independently but also integrated into other tests, including the 8-hour stability test and durability test. By verifying that the total number of orders in the database aligns with the number of orders recorded by the RTE, the Consistency test ensures data consistency. The Durability test focuses on the system's ability to prevent data inconsistency in the event of machine failures. This test entails simulating the instantaneous loss of power to one node in the Client, Manager, and Data nodes.

As shown in Figure 8, the Durability test initiates with a consistency check, where the total number of orders (N1) is recorded by examining the next order ID from the District table (D_Next_O_ID). Once the RTE is launched, the test enters the ramp-up phase, and within 2-3 minutes, an observable increase in tpmC can be observed. After 10 minutes, the performance stabilizes and remains consistent for the subsequent 20 minutes. Then, the power to the primary Manager node is intentionally disconnected to simulate a failure. Remarkably, this failure does not impact performance, as another Secondary Manager seamlessly assumes the role of the new primary Manager. Following a smooth operation for five minutes, the power to a randomly selected primary Data node is disconnected to simulate its failure. Although performance experiences a slight decline, the primary-secondary switch is completed within a mere 18 seconds. Note that the recovery in the Recover phase is completed in seconds, but we will wait a longer period for the Full tmpC observation as required by TPC-C audit. After another five minutes of stable run, the power to a random Client node is disconnected to simulate its failure. This leads to a slight decrease in performance without any subsequent rebound. Continuing with a smooth run for an additional five minutes, the test then enters the ramp-down phase until tpmC reaches 0. Upon completion, another consistency check is conducted, and the current total number of orders (N2) is recorded. The difference between N2 and N1 is compared to the total number of successful orders recorded by the RTE, ensuring consistency and confirming the absence of any errors.

4.5 Other (50%, 80%) Stress Tests and Scalability

In addition, we conducted tests to assess the system's performance under various stresses imposed by the RTE (Real-Time Environment). This was accomplished by manipulating the number of query requests, such as reducing the number of warehouses or decreasing the count of RTEs. Figure 9b illustrates the 90th percentile latency of the New-order transaction under different stress levels by controlling the percent of Maximum Qualified Throughput (MQTh). As the stress intensifies, the 90th percentile latency also exhibits an

Table 5: Comparison of TDSQL to a centralized database (OP3).

Database	System TPS	Server	W1	W2	W3	W4	W5	Cost
TDSQL	6200	16*x86 machine	300 ms	100 ms	20 s	14 min	16 min	$
OP3	8000	1*Mainframe	70 ms	30 ms	60 s	60 min	180 min	$$$$

Table 6: Comparison of POC against two opponents (OP1 and OP2) by TPS (txn/S). The statistics (response time (RT) (S), and CPU and memory utilization) are collected.

Parameter		TPS comparison			TDSQL-statistics		
No.	THD	TDSQL	OP1	OP2	RT	CPU	Memory
W1	50	31.2	19.9	14.2	1.6	9.6 %	21.3%
	100	50.6	34.1	23.9	2.0	13.0%	21.9%
	200	71.0	52.6	32.7	2.8	17.0%	23.1%
W2	50	65.8	45.4	28.7	0.8	13.2%	81.7%
	100	105.0	73.1	46.8	0.9	21.5%	82.0%
	200	152.3	102.3	65.4	1.3	33.2%	82.6%
W3	50	281.5	81.5	186.4	0.2	15.7%	82.0%
	100	471.1	156.9	299.8	0.2	26.8%	82.3%
	200	691.4	265.3	437.8	0.3	41.6%	82.9%
W4	50	292.6	220.1	105.9	0.2	7.0 %	82.1%
	100	475.7	369.9	181.2	0.2	10.9%	82.2%
	200	665.1	468.3	276.4	0.3	15.5%	82.5%
W5	50	82.6	64.4	42.4	0.6	14.4%	82.1%
	100	133.6	108.0	74.6	0.7	23.9%	82.4%
	200	201.8	149.7	113.9	1.0	38.5%	83.0%

upward trend, indicating that the system's performance remains stable and aligns with our anticipated expectations.

We conducted tests using different node configurations, specifically with 3, 60, 200, and 1650 nodes, as illustrated in Figure 10a. The results indicate that the system exhibits nearly linear scalability, with its performance showing no signs of reaching a bottleneck. Furthermore, Figure 10b demonstrates that as the number of nodes increases, there is a slight decrease in tpmC per node. For instance, with 3 nodes, the tpmC per node is 0.53 million, while with 1650 nodes, it remains at a level of 0.49 million. This result surpasses other comparable publicly reported TPC-C results [57] by more than 8%. Even with 1650 nodes, the single-node QPS (Queries Per Second) level remains remarkably high, reaching 1.8 million.

4.6 Case Study of Bank Workloads

Performance Advantage – POC of Bank Workloads. To win the bid in banks, database vendors are required to conduct Proof of Concept (POC) tests using real workloads to evaluate the functionality and performance of their databases. TDSQL often competes with at least two other vendors, and generally, to secure the order needs to demonstrate superiority in the majority of workload scenarios. Below is a successful case from our POC tests in a bank using real workloads. The testing environment consisted of 8 x86 servers, each with 2*48 CPUs, 2*32 GB memory, and 3*2TB SSDs. Tested workloads are (W1) Loan Disbursement (write-intensive workload), (W2) Current Account Transfer (a mix of read and write operations), (W3) Transaction History Inquiry (read-intensive), (W4) Account Information Inquiry (read-intensive), and (W5) Second-Generation Payment Single Entry Accounting (write-intensive). W1 and W2 belong to the intensive write workload (e.g., on payday, when most companies pay employee salaries on a certain day at the beginning of the month), W3 and W4 belong to the read consistency workload (banks typically need to verify daily income and expenditure balances and reconciliation data to ensure consistency). W5 belongs to the highest throughput without abort workload (the maximum performance without abort, which is different from the regular pursuit of maximum performance).

In this POC, we conducted tests under various thread counts (THD=50,100,200) as per the bank's requirements. As illustrated in Table 6, TDSQL outperformed the anonymous competitors OP1 and OP2 by demonstrating a higher TPS under transactional (W2, W5), write-intensive (W1), and read-intensive (W3, W4) workloads. As the number of concurrent threads increased, the performance progressively improved. The CPU utilization rate suggests that there is potential for further performance enhancement by increasing the thread count under this configuration. Interestingly, one of the bank's requirements was to avoid transaction timeout rollbacks, hence the user count was not maximized during testing. For instance, we tested the maximum number of concurrent users without timeout rollbacks using W5. Under a single thread, TDSQL could support up to 6 concurrent users without any timeout rollbacks. However, a few timeout rollbacks began to occur when the user count increased to 7. In contrast, OP1 and OP2 could only support a maximum of 2 concurrent users under single-threaded operation.

Cost Advantage – Upgrading Centralized Database in a Bank. Traditional banks often deploy databases on mainframes or minicomputers, which incur high hardware costs. Some transactional and analytics workloads are slow and difficult to scale. The following introduces a comparison of the advantages of TDSQL upgrading a centralized database (OP3). The test environment for TDSQL uses traditional x86 servers of 16 DB nodes (each with 96 CPU, 265 GB, 2T SSD). Tested workloads are (W1) Core High-Frequency Transactions (a mix of read and write operations), (W2) Core Transaction Inquiry (read-intensive), (W3) Batch Disbursement and Deduction of 10,000 Transactions (write-intensive), (W4) End-of-Day Batch Analysis (read-intensive), and (W5) Interest Settlement for Deposits and Loans (a mix of read and write operations).

As shown in Table 5, when compared to OP3, TDSQL's average TPS is slightly lower, but the overall cost is only a quarter of OP3's. Although latency increases under core transactions and queries (W1 and W2) due to network latency, the significantly higher concurrency of the distributed database cluster compared to the mainframe results in a TPS that is not far off. The latency for W1 and W2 is within the acceptable range for banking transactions. In non-core transaction and analysis scenarios (W3, W4, W5), TDSQL, compared to OP3, overcomes certain disk usage bottlenecks thanks to distributed computing and effective disk expansion utilization, demonstrating a clear advantage in parallel querying and analysis.

After the upgrade to TDSQL, the bank's real peak daily throughput during online periods is 1.2 million transactions. The system resource usage rate is less than 5% during online periods and remains less than 10% even during peak periods of batch processing. Despite significantly reducing costs, the cluster still clearly possesses the capacity for future business expansion.

4.7 Insight and Discussion

1) Potential for Performance Improvement. During the test, we found that memory usage remains a significant bottleneck (with a usage rate exceeding 95%), while our vCPU usage is only around 60%, and IO write utilization is about 75%. If memory capacity increases, there is room for performance improvement. This issue is similar to cloud vendors' challenges when selling cloud services in reality. In some cases, even if the CPU resources are oversold, the overall CPU utilization rate is still not high. This is because we often need to physically partition memory capacity, which limits the number of instances we can sell on a machine due to memory capacity constraints. We envision enhancing TDSQL by incorporating larger memory machines, such as those utilizing CXL technology [2], or adopting a flexible architecture, such as shared storage [82], to facilitate more efficient resource allocation and utilization.

2) Trade-off Between Throughput and Variation. Various modules of server hardware have certain failure rates (0.2% from our TPC-C test experience), especially storage devices such as disks and memory. In our TPC-C test, we observed that higher stress levels tend to induce more failures. In our initial tests of the 8-hour stability test, we aimed for optimal tpmC. However, we consistently encountered disk or memory failures during the test, resulting in many forced rollback transactions. As a result, we were sometimes unable to meet the TPC-C auditing requirements of maintaining a consistent jitter rate below 2%. This is similar to the requirement of no timeout rollback transactions in our banking POC testing. To address this issue, we have intentionally reduced the stress level slightly (e.g., by 5%) to ensure a lower jitter rate (0.2%) during long-duration high-stress testing. We have also applied similar strategies in real-world scenarios, where we scale up our resources when the utilization of certain resources reaches 90% or even 80%. This approach minimizes the impact on hardware and business workloads.

5 RELATED WORK

Centralized Databases. A centralized database is a database system that stores data at a central node. Classic solutions come from Oracle [49], DB2 [29], and MySQL [46], where data consistency and integrity are relatively easy to ensure, and the management and maintenance of a centralized database are relatively simple. However, there are problems with poor fault tolerance, as there is no redundant data scattered in different places. There is also a problem with poor scalability, as the amount of data grows, the performance of the centralized database may not keep up as the physical configuration and capacity of a single machine are limited. Data backup and recovery may be difficult and time-consuming. TDSQL can be deployed in both centralized and distributed modes, supporting high availability, scalability, and automatic failover.

Distributed Databases. A distributed database is a database system that stores data across multiple physical locations. Data can be distributed across different nodes on the same network or across different networks. Distributed databases have high availability and fault tolerance, and can effectively adjust data distribution, use parallelism to improve query speed and load balancing. Classic solutions are Google Spanner [15], Oceanbase [71]. Unlike the B-tree used in TDSQL, Spanner and Oceanbase employ LSM-tree [47], which offers some space-saving advantages. However, they experience significant performance jitters due to the LSM compaction. CockroachDB [55] and YugabyteDB [72] are excellent distributed databases inspired by Google Spanner, and they use HLC (hybrid logical clock) instead of the TrueTime API to replace real-time timestamps. Therefore, their commit wait time must consider the logical time difference. TDSQL, however, uses the TrueTime API for timestamp allocation.

Large-scale Application Databases. Large-scale applications can process and analyze massive amounts of data to extract valuable insights and information. Snowflake [16, 62] is a widely used distributed OLAP database system that is based on a shared-storage architecture, separating storage from computation to achieve independent scalability and elasticity. Similar to Snowflake's architecture, AnalyticDB [66, 73] uses a hybrid storage format [37, 74, 75] (rather than pure columnar storage) to efficiently support OLAP and point query queries. These are scenarios that mostly lean toward OLAP applications [23]. Aurora [60, 61] is a popular OLTP database that leverages decentralized storage. It separates the storage engine from the compute engine. To reduce expensive network I/O costs, Aurora only sends logs instead of actual data pages to the storage engine. To improve performance, PolarDB[11] utilizes emerging hardware such as RDMA and 3D Xpoint SSD to reduce network overhead and transaction commit time, but using new hardware may result in higher costs [38, 39, 41, 53, 67, 77]. TDSQL deployed a share-nothing storage-compute separation architecture. This paper mainly focuses on TDSQL optimizing OLTP applications.

6 CONCLUSION

This paper shares over a decade of experience in the design, development, and optimization of TDSQL in large-scale testing and application scenarios. It covers various aspects such as kernel, synchronization, transaction, query, and memory allocation, detailing their implementation and optimization. Through the TPC-C benchmark test, TDSQL achieved a performance of 814 million tpmC with a jitter rate of less than 0.2%, an order of magnitude lower than the standard requirement. During the 8-hour stress test, TDSQL completed over 860 billion transactions and processed 40 trillion order details, with zero forced rollbacks and zero data inconsistency.

7 ACKNOWLEDGMENT

We would like to express our sincere gratitude to the anonymous reviewers for their invaluable comments and insightful suggestions. We also thank Wen Zhang and Weixiang Zhai for generously providing essential materials. Furthermore, we acknowledge the dedicated efforts of the TDSQL team, whose development work has been instrumental in the realization of this project.

REFERENCES

[1] Michael Abebe, Brad Glasbergen, and Khuzaima Daudjee. 2020. MorphoSys: automatic physical design metamorphosis for distributed database systems. *Proceedings of the VLDB Endowment* 13, 13 (2020), 3573–3587.

[2] Minseon Ahn, Andrew Chang, Donghun Lee, Jongmin Gim, Jungmin Kim, Jaemin Jung, Oliver Rebholz, Vincent Pham, Krishna Malladi, and Yang Seok Ki. 2022. Enabling CXL memory expansion for in-memory database management systems. In *Proceedings of the 18th International Workshop on Data Management on New Hardware*. 1–5.

[3] Amirhossein Aleyasen, Mark Morcos, Lyublena Antova, Marc Sugiyama, Dmitri Korablev, Jozsef Patvarczki, Rima Mutreja, Michael Duller, Florian M. Waas, and Marianne Winslett. 2022. Intelligent Automated Workload Analysis for Database Replatforming. In *SIGMOD Conference*. ACM, 2273–2285.

[4] Ghazi I. Alkhatib and Ronny S. Labban. 2002. Transaction Management in Distributed Database Systems: The Case of Oracle's Two-Phase Commit. *J. Inf. Syst. Educ.* 13, 2 (2002), 95–104.

[5] Ahmed Alquraan, Alex Kogan, Virendra J. Marathe, and Samer Al-Kiswany. 2020. Scalable, NearZero Loss Disaster Recovery for Distributed Data Stores. *Proc. VLDB Endow.* 13, 9 (2020), 1429–1442.

[6] Peter Alvaro and Kyle Kingsbury. 2020. Elle: Inferring Isolation Anomalies from Experimental Observations. *Proc. VLDB Endow.* 14, 3 (2020), 268–280.

[7] Panagiotis Antonopoulos, Peter Byrne, Wayne Chen, Cristian Diaconu, Raghavendra Thallam Kodandaramaih, Hanuma Kodavalla, Prashanth Purnananda, Adrian-Leonard Radu, Chaitanya Sreenivas Ravella, and Girish Mittur Venkataramanappa. 2019. Constant Time Recovery in Azure SQL Database. *Proc. VLDB Endow.* 12, 12 (2019), 2143–2154.

[8] 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.

[9] Edward Bortnikov, Eshcar Hillel, Idit Keidar, Ivan Kelly, Matthieu Morel, Sameer Paranjpye, Francisco Perez-Sorrosal, and Ohad Shacham. 2017. Omid, Reloaded: Scalable and Highly-Available Transaction Processing. In *FAST*. USENIX Association, 167–180.

[10] Dennis Butterstein, Daniel Martin, Knut Stolze, Felix Beier, Jia Zhong, and Lingyun Wang. 2020. Replication at the Speed of Change - a Fast, Scalable Replication Solution for Near Real-Time HTAP Processing. *Proc. VLDB Endow.* 13, 12 (2020), 3245–3257.

[11] Wei Cao, Zhenjun Liu, Peng Wang, Sen Chen, Caifeng Zhu, Song Zheng, Yuhui Wang, and Guoqing Ma. 2018. PolarFS: An Ultra-low Latency and Failure Resilient Distributed File System for Shared Storage Cloud Database. *Proc. VLDB Endow.* 11, 12 (2018), 1849–1862.

[12] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Michael Burrows, Tushar Chandra, Andrew Fikes, and Robert Gruber. 2006. Bigtable: A Distributed Storage System for Structured Data (Awarded Best Paper!). In *OSDI*. USENIX Association, 205–218.

[13] Tencent Cloud. 2024. *Tencent Cloud official site.* https://www.tencentcloud.com/.

[14] Tencent Cloud. 2024. *TXSQL gitee repo.* https://gitee.com/X-SQL/TXSQL.

[15] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson C. Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. 2012. Spanner: Google's Globally-Distributed Database. In *OSDI*. USENIX Association, 251–264.

[16] Benoît Dageville, Thierry Cruanes, Marcin Zukowski, Vadim Antonov, Artin Avanes, Jon Bock, Jonathan Claybaugh, Daniel Engovatov, Martin Hentschel, Jiansheng Huang, Allison W. Lee, Ashish Motivala, Abdul Q. Munir, Steven Pelley, Peter Povinec, Greg Rahn, Spyridon Triantafyllis, and Philipp Unterbrunner. 2016. The Snowflake Elastic Data Warehouse. In *SIGMOD Conference*. ACM, 215–226.

[17] Miguel Diogo, Bruno Cabral, and Jorge Bernardino. 2019. Consistency Models of NoSQL Databases. *Future Internet* 11, 2 (2019), 43.

[18] Siying Dong, Shiva Shankar P., Satadru Pan, Anand Ananthabhotla, Dhanabal Ekambaram, Abhinav Sharma, Shobhit Dayal, Nishant Vinaybhai Parikh, Yanqin Jin, Albert Kim, Sushil Patil, Jay Zhuang, Sam Dunster, Akanksha Mahajan, Anirudh Chelluri, Chaitanya Datye, Lucas Vasconcelos Santana, Nitin Garg, and Omkar Gawde. 2023. Disaggregating RocksDB: A Production Experience. *Proc. ACM Manag. Data* 1, 2 (2023), 192:1–192:24.

[19] Martyn Ellison, Radu Calinescu, and Richard F Paige. 2018. Evaluating cloud database migration options using workload models. *Journal of Cloud Computing* 7 (2018), 1–18.

[20] Aaron J. Elmore, Sudipto Das, Divyakant Agrawal, and Amr El Abbadi. 2011. Zephyr: live migration in shared nothing databases for elastic cloud platforms. In *SIGMOD Conference*. ACM, 301–312.

[21] 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. https://doi.org/10.14778/3055540.3055553

[22] Alexey Gotsman, Hongseok Yang, Carla Ferreira, Mahsa Najafzadeh, and Marc Shapiro. 2016. 'Cause I'm strong enough: reasoning about consistency choices in distributed systems. In *POPL*. ACM, 371–384.

[23] Jiawei Guan, Feng Zhang, Siqi Ma, Kuangyu Chen, Yihua Hu, Yuxing Chen, Anqun Pan, and Xiaoyong Du. 2023. Homomorphic Compression: Making Text Processing on Compression Unlimited. *Proc. ACM Manag. Data* 1, 4 (2023), 271:1–271:28.

[24] 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.

[25] Herodotos Herodotou, Yuxing Chen, and Jiaheng Lu. 2020. A Survey on Automatic Parameter Tuning for Big Data Processing Systems. *ACM Comput. Surv.* 53, 2 (2020), 43:1–43:37.

[26] Dongxu Huang, Qi Liu, Qiu Cui, Zhuhe Fang, Xiaoyu Ma, Fei Xu, Li Shen, Liu Tang, Yuxing Zhou, Menglong Huang, Wan Wei, Cong Liu, Jian Zhang, Jianjun Li, Xuelian Wu, Lingyu Song, Ruoxi Sun, Shuaipeng Yu, Lei Zhao, Nicholas Cameron, Liquan Pei, and Xin Tang. 2020. TiDB: A Raft-based HTAP Database. *Proc. VLDB Endow.* 13, 12 (2020), 3072–3084.

[27] Kaile Huang, Si Liu, Zhenge Chen, Hengfeng Wei, David A. Basin, Haixiang Li, and Anqun Pan. 2023. Efficient Black-box Checking of Snapshot Isolation in Databases. *Proc. VLDB Endow.* 16, 6 (2023), 1264–1276.

[28] Patrick Hunt, Mahadev Konar, Flavio Paiva Junqueira, and Benjamin C. Reed. 2010. ZooKeeper: Wait-free Coordination for Internet-scale Systems. In *USENIX Annual Technical Conference*. USENIX Association.

[29] IBM. 2024. *DB2*. https://www.ibm.com/products/db2.

[30] IDC. 2024. *IDC MarketScape*. https://www.idc.com/getdoc.jsp?containerId=CHC50734323.

[31] Jepsen. 2024. *Jepsen test*. https://jepsen.io/.

[32] 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.

[33] Alexey Kopytov. 2004. Sysbench: a system performance benchmark. *http://sysbench. sourceforge. net/* (2004).

[34] Juchang Lee, SeungHyun Moon, Kyu Hwan Kim, Deok Hoe Kim, Sang Kyun Cha, Wook-Shin Han, Chang Gyoo Park, Hyoung Jun Na, and Joo-Yeon Lee. 2017. Parallel Replication across Formats in SAP HANA for Scaling Out Mixed OLTP/OLAP Workloads. *Proc. VLDB Endow.* 10, 12 (2017), 1598–1609.

[35] Juchang Lee, Hyungyu Shin, Chang Gyoo Park, Seongyun Ko, Jaeyun Noh, Yongjae Chuh, Wolfgang Stephan, and Wook-Shin Han. 2016. Hybrid Garbage Collection for Multi-Version Concurrency Control in SAP HANA. In *SIGMOD Conference*. ACM, 1307–1318.

[36] Leon Lee, Siphrey Xie, Yunus Ma, and Shimin Chen. 2022. Index Checkpoints for Instant Recovery in In-Memory Database Systems. *Proc. VLDB Endow.* 15, 8 (2022), 1671–1683.

[37] Guoliang Li and Chao Zhang. 2022. HTAP Databases: What is New and What is Next. In *SIGMOD Conference*. ACM, 2483–2488.

[38] Gang Liu, Leying Chen, and Shimin Chen. 2023. Zen+: a robust NUMA-aware OLTP engine optimized for non-volatile main memory. *VLDB J.* 32, 1 (2023), 123–148.

[39] Jiesong Liu, Feng Zhang, Lv Lu, Chang Qi, Xiaoguang Guo, Dong Deng, Guoliang Li, Huanchen Zhang, Jidong Zhai, Hechen Zhang, Yuxing Chen, Anqun Pan, and Xiaoyong Du. 2024. G-Learned Index: Enabling Efficient Learned Index on GPU. *IEEE Trans. Parallel Distributed Syst.* 35, 6 (2024), 795–812.

[40] Jiaheng Lu, Yuxing Chen, Herodotos Herodotou, and Shivnath Babu. 2019. Speedup Your Analytics: Automatic Parameter Tuning for Databases and Big Data Systems. *Proc. VLDB Endow.* 12, 12 (2019), 1970–1973.

[41] Ziyi Lu, Qiang Cao, Hong Jiang, Yuxing Chen, Jie Yao, and Anqun Pan. 2024. FluidKV: Seamlessly Bridging the Gap between Indexing Performance and Memory-Footprint on Ultra-Fast Storage. *Proc. VLDB Endow.* 17, 6 (2024), 1377–1390.

[42] lucaszhai. 2024. *Bug lists*. https://bugs.mysql.com/search.php?cmd=display&status=All&severity=all&reporter=5698040.

[43] Zhenghua Lyu, Huan Hubert Zhang, Gang Xiong, Gang Guo, Haozhou Wang, Jinbao Chen, Asim Praveen, Yu Yang, Xiaoming Gao, Alexandra Wang, Wen Lin, Ashwin Agrawal, Junfeng Yang, Hao Wu, Xiaoliang Li, Feng Guo, Jiang Wu, Jesse Zhang, and Venkatesh Raghavan. 2021. Greenplum: A Hybrid Database for Transactional and Analytical Workloads. In *SIGMOD Conference*. ACM, 2530–2542.

[44] Yaser Mansouri, Adel Nadjaran Toosi, and Rajkumar Buyya. 2018. Data Storage Management in Cloud Environments: Taxonomy, Survey, and Future Directions. *ACM Comput. Surv.* 50, 6 (2018), 91:1–91:51.

[45] C. Mohan, Don Haderle, Bruce G. Lindsay, Hamid Pirahesh, and Peter M. Schwarz. 1992. ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging. *ACM Trans. Database Syst.* 17, 1 (1992), 94–162.

[46] MySQL. 2024. *MySQL*. https://www.mysql.com/.

[47] Patrick E. O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth J. O'Neil. 1996. The Log-Structured Merge-Tree (LSM-Tree). *Acta Informatica* 33, 4 (1996), 351–385.

[48] Diego Ongaro and John K. Ousterhout. 2014. In Search of an Understandable Consensus Algorithm. In *USENIX Annual Technical Conference*. USENIX Association, 305–319.

[49] Oracle 2024. *Oracle*. Oracle. https://www.oracle.com/.

[50] Daniel Peng and Frank Dabek. 2010. Large-scale Incremental Processing Using Distributed Transactions and Notifications. In *OSDI*. USENIX Association, 251–264.

[51] Dai Qin, Ashvin Goel, and Angela Demke Brown. 2017. Scalable Replay-Based Replication For Fast Databases. *Proc. VLDB Endow.* 10, 13 (2017), 2025–2036.

[52] Ori Shalev and Nir Shavit. 2003. Split-ordered lists: lock-free extensible hash tables. In *PODC*. ACM, 102–111.

[53] Alex Shamis, Matthew Renzelmann, Stanko Novakovic, Georgios Chatzopoulos, Aleksandar Dragojevic, Dushyanth Narayanan, and Miguel Castro. 2019. Fast General Distributed Transactions with Opacity. In *SIGMOD Conference*. ACM, 433–448.

[54] Jeff Shute, Radek Vingralek, Bart Samwel, Ben Handy, Chad Whipkey, Eric Rollins, Mircea Oancea, Kyle Littlefield, David Menestrina, Stephan Ellner, John Cieslewicz, Ian Rae, Traian Stancescu, and Himani Apte. 2013. F1: A Distributed SQL Database That Scales. *Proc. VLDB Endow.* 6, 11 (2013), 1068–1079.

[55] Rebecca Taft, Irfan Sharif, Andrei Matei, Nathan VanBenschoten, Jordan Lewis, Tobias Grieger, Kai Niemi, Andy Woods, Anne Birzin, Raphael Poss, Paul Bardea, Amruta Ranade, Ben Darnell, Bram Gruneir, Justin Jaffray, Lucy Zhang, and Peter Mattis. 2020. CockroachDB: The Resilient Geo-Distributed SQL Database. In *SIGMOD Conference*. ACM, 1493–1509.

[56] Tencent. 2024. *King of Honor*. https://www.honorofkings.com/.

[57] TPC 2023. *TPC-C All Results - Sorted by Performance*. TPC. https://www.tpc.org/tpcc/results/tpcc_results5.asp?print=false&orderby=tpm&sortby=desc.

[58] TPC 2023. *TPC-C: On-Line Transaction Processing Benchmark*. TPC. https://www.tpc.org/tpcc/default5.asp.

[59] Nathan VanBenschoten, Arul Ajmani, Marcus Gartner, Andrei Matei, Aayush Shah, Irfan Sharif, Alexander Shraer, Adam Storm, Rebecca Taft, Oliver Tan, Andy Woods, and Peyton Walters. 2022. Enabling the Next Generation of Multi-Region Applications with CockroachDB. In *SIGMOD Conference*. ACM, 2312–2325.

[60] 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 *SIGMOD Conference*. ACM, 1041–1052.

[61] Alexandre Verbitski, Anurag Gupta, Debanjan Saha, James Corey, Kamal Gupta, Murali Brahmadesam, Raman Mittal, Sailesh Krishnamurthy, Sandor Maurice, Tengiz Kharatishvili, and Xiaofeng Bao. 2018. Amazon Aurora: On Avoiding Distributed Consensus for I/Os, Commits, and Membership Changes. In *SIGMOD Conference*. ACM, 789–796.

[62] Midhul Vuppalapati, Justin Miron, Rachit Agarwal, Dan Truong, Ashish Motivala, and Thierry Cruanes. 2020. Building An Elastic Query Engine on Disaggregated Storage. In *NSDI*. USENIX Association, 449–462.

[63] Jianguo Wang and Qizhen Zhang. 2023. Disaggregated Database Systems. In *SIGMOD Conference Companion*. ACM, 37–44.

[64] Tianzheng Wang, Ryan Johnson, Alan D. Fekete, and Ippokratis Pandis. 2017. Efficiently making (almost) any concurrency control mechanism serializable. *VLDB J.* 26, 4 (2017), 537–562.

[65] WeChat. 2024. *WeChat Pay*. https://pay.weixin.qq.com/index.php/public/wechatpay_en.

[66] Chuangxian Wei, Bin Wu, Sheng Wang, Renjie Lou, Chaoqun Zhan, Feifei Li, and Yuanzhe Cai. 2020. AnalyticDB-V: A Hybrid Analytical Engine Towards Query Fusion for Structured and Unstructured Data. *Proc. VLDB Endow.* 13, 12 (2020), 3152–3165.

[67] Xingda Wei, Zhiyuan Dong, Rong Chen, and Haibo Chen. 2018. Deconstructing RDMA-enabled Distributed Transactions: Hybrid is Better!. In *OSDI*. USENIX Association, 233–251.

[68] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, Darrell D. E. Long, and Carlos Maltzahn. 2006. Ceph: A Scalable, High-Performance Distributed File System. In *OSDI*. USENIX Association, 307–320.

[69] Yingjun Wu, Joy Arulraj, Jiexi Lin, Ran Xian, and Andrew Pavlo. 2017. An Empirical Evaluation of In-Memory Multi-Version Concurrency Control. *Proc. VLDB Endow.* 10, 7 (2017), 781–792.

[70] Zhichen Xu, Ying Gao, and Andrew Davidson. 2023. Keep Your Distributed Data Warehouse Consistent at a Minimal Cost. *Proc. ACM Manag. Data* 1, 2 (2023), 190:1–190:25.

[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] Yugabyte. 2024. *YugabyteDB*. https://www.yugabyte.com/.

[73] Chaoqun Zhan, Maomeng Su, Chuangxian Wei, Xiaoqiang Peng, Liang Lin, Sheng Wang, Zhe Chen, Feifei Li, Yue Pan, Fang Zheng, and Chengliang Chai. 2019. AnalyticDB: Real-time OLAP Database System at Alibaba Cloud. *Proc. VLDB Endow.* 12, 12 (2019), 2059–2070.

[74] Chao Zhang, Guoliang Li, and Tao Lv. 2024. HyBench: A New Benchmark for HTAP Databases. *Proc. VLDB Endow.* 17, 5 (2024), 939–951.

[75] Chao Zhang, Guoliang Li, Jintao Zhang, Xinning Zhang, and Jianhua Feng. 2024. HTAP Databases: A Survey. *IEEE Transactions on Knowledge and Data Engineering* (2024), 939–951.

[76] Ji Zhang, Yu Liu, Ke Zhou, Guoliang Li, Zhili Xiao, Bin Cheng, Jiashu 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.

[77] Qian Zhang, Jingyao Li, Hongyao Zhao, Quanqing Xu, Wei Lu, Jinliang Xiao, Fusheng Han, Chuanhui Yang, and Xiaoyong Du. 2023. Efficient Distributed Transaction Processing in Heterogeneous Networks. *Proc. VLDB Endow.* 16, 6 (2023), 1372–1385.

[78] Wei Zhao. 2024. *Bug lists*. https://bugs.mysql.com/search.php?cmd=display&status=All&severity=all&reporter=9354215.

[79] Wenting Zheng, Stephen Tu, Eddie Kohler, and Barbara Liskov. 2014. Fast Databases with Fast Durability and Recovery Through Multicore Parallelism. In *OSDI*. USENIX Association, 465–477.

[80] Jingyu Zhou, Meng Xu, Alexander Shraer, Bala Namasivayam, Alex Miller, Evan Tschannen, Steve Atherton, Andrew J. Beamon, Rusty Sears, John Leach, Dave Rosenthal, Xin Dong, Will Wilson, Ben Collins, David Scherer, Alec Grieser, Young Liu, Alvin Moore, Bhaskar Muppana, Xiaoge Su, and Vishesh Yadav. 2021. FoundationDB: A Distributed Unbundled Transactional Key Value Store. In *SIGMOD Conference*. ACM, 2653–2666.

[81] Yiwen Zhu, Yuanyuan Tian, Joyce Cahoon, Subru Krishnan, Ankita Agarwal, Rana Alotaibi, Jesús Camacho-Rodríguez, Bibin Chundatt, Andrew Chung, Niharika Dutta, Andrew Fogarty, Anja Gruenheid, Brandon Haynes, Matteo Interlandi, Minu Iyer, Nick Jurgens, Sumeet Khushalani, Brian Kroth, Manoj Kumar, Jyoti Leeka, Sergiy Matusevych, Minni Mittal, Andreas Müller, Kartheek Muthyala, Harsha Nagulapalli, Yoonjae Park, Hiren Patel, Anna Pavlenko, Olga Poppe, Santhosh Ravindran, Karla Saur, Rathijit Sen, Steve Suh, Arijit Tarafdar, Kunal Waghray, Demin Wang, Carlo Curino, and Raghu Ramakrishnan. 2023. Towards Building Autonomous Data Services on Azure. In *SIGMOD Conference Companion*. ACM, 217–224.

[82] Tobias Ziegler, Philip A Bernstein, Viktor Leis, and Carsten Binnig. 2023. Is Scalable OLTP in the Cloud a Solved Problem. In *13th Annual Conference on Innovative Data Systems Research, CIDR 2023, Amsterdam, The Netherlands, January 8-11, 2023, Online Proceedings*.