Reaching the Pinnacle of TPC-DS: Co-design of Architecture, Executor, and Storage in TDSQL

Yiteng Chu[1], Jie Jiang[2], Yuxing Chen[1,*], Feng Zhang[1], Yuxin Tang[1], Anqun Pan[2], Xin Wu[2], Xiang Hu[2], Hailin Lei[2], Qian Zhang[2], Ziyan Lin[2], Xiaoliang Yan[2], Xiaojian Fan[2], Liang Zhang[2], Bin Qu[2], Wen Zhang[2], Mengbang Xing[2], Lixiong Zheng[2], Wei Lu[1], Yunpeng Chai[1], Xiaoyong Du[1]

[1]Renmin University of China [2]Tencent Inc.

[1]{chuyiteng, axinggu, fengzhang, yuxintang, lu-wei, ypchai, duyong}@ruc.edu.cn

[2]{zeus, aaronpan, ericxwu, josehu, harlylei, qiannzhang, andrelin, anthonyyan, kaysonfan, constlzhang, ericbqu, sevenwzhang, rockyxing, paterzheng}@tencent.com

ABSTRACT

Enterprise data explosion and the urgent industry demand for real-time complex multidimensional analytics require OLAP databases to be highly scalable, efficient, and cost-effective. Though diverse solutions (shared-nothing MPP databases, cloud-native decoupled systems, in-process analytical engines) exist with respective strengths, they all have critical inherent flaws.

In response, this paper presents TDSQL, a distributed OLAP database system developed by Tencent. We leverage its native architectural advantages, analyze the merits and drawbacks of state-of-the-art systems, and elaborate on the rationale behind our technical solution selection and proprietary innovations tailored for TDSQL. Specifically, built upon the traditional MPP execution framework, TDSQL incorporates the Forward Node mechanism to enhance scalability. By optimizing parallel execution, runtime filter strategies, and designing and implementing a vectorized execution engine, TDSQL achieves rapid response to large-scale complex queries. Experimental results based on the TPC-DS benchmark demonstrate that TDSQL ranks first among publicly reported systems. In a cluster configuration for 10,000 GB data size, TDSQL achieves a score of 72.6 million QphDS, which is 1.81 times and 3.82 times the scores of the second- and third-highest-performing database systems in public TPC-DS results, respectively, while offering a 79% and 37% lower cost per 1000 QphDS.

PVLDB Reference Format:

Yiteng Chu, Jie Jiang, Yuxing Chen, Feng Zhang, Yuxin Tang, Anqun Pan, Xin Wu, Xiang Hu, Hailin Lei, Qian Zhang, Ziyan Lin, Xiaoliang Yan, Xiaojian Fan, Liang Zhang, Bin Qu, Wen Zhang, Mengbang Xing, Lixiong Zheng, Wei Lu, Yunpeng Chai, Xiaoyong Du. Reaching the Pinnacle of TPC-DS: Co-design of Architecture, Executor, and Storage in TDSQL. PVLDB, 19(12): 4009 - 4022, 2026.

doi:10.14778/3827998.3828012

1 INTRODUCTION

In recent years, the rapid growth of data volumes has made distributed OLAP databases, designed to process large-scale datasets and support complex analytical queries, essential for data analysis and decision-making [2, 27, 43, 81, 84]. TDSQL, an enterprise-grade distributed database developed by Tencent, is deployed on Tencent Cloud [18] and currently serves over 500,000 users. Although TDSQL has been extensively optimized for OLTP scenarios [15, 75], this paper focuses on its OLAP-oriented optimizations for analytical workloads. Its exceptional performance is particularly evident in the official TPC-DS benchmark [73], a globally recognized standard for evaluating OLAP system capabilities. TDSQL secures first place and sets a new global record, demonstrating its capabilities in large-scale data processing and high-concurrency OLAP scenarios.

Continuous growth of enterprise data and complex analytical demands pose OLAP database development challenges for the industry [5, 21, 32, 49] and also for Tencent. Our systems must support real-time ingestion and querying of over 300 TB of monthly data for payment and telecommunications services, provide financial-grade reliability for mission-critical workloads across more than 4,000 enterprise customers including seven of China's top ten banks, and deliver extreme scalability for complex multidimensional analytics [16, 28, 34, 50, 81] such as multi-terabyte e-commerce log processing and security analysis over billions of devices.

State-of-the-art OLAP systems have evolved along three major architectural paradigms. First, shared-nothing MPP databases and distributed query engines enable large-scale data processing. Early representatives including GreenPlum [46] and Vertica [38] pioneered columnar storage and batch ETL for accelerated analytics. Later engines such as PrestoDB [68] and Impala [36] adopted compute–storage decoupling and runtime optimizations. Recent real-time systems including Druid [79] and ClickHouse [63] further improve latency via optimized ingestion and multi-level indexing. Second, cloud-native decoupled platforms such as Databricks [80] and Snowflake [21] provide elastic, large-scale SQL analytics [54]. Third, in-process engines like DuckDB [61] offer lightweight, high-performance single-node execution for local analytical workloads.

While these diverse paradigms offer solutions for large-scale query analysis, they still face critical, architecture-specific challenges. First, within shared-nothing MPP databases, most systems [46, 63, 70] rely on traditional MPP architectures, requiring numerous point-to-point connections between nodes for query execution. This communication overhead limits system scalability. Second, both shared-nothing MPP databases and cloud-native decoupled systems [4, 36, 46, 72] frequently encounter low CPU utilization during distributed query runs, finding it challenging to fully leverage computational resources, thereby restricting query performance.

Proceedings of the VLDB Endowment, Vol. 19, No. 12 ISSN 2150-8097.
doi:10.14778/3827998.3828012

* Yuxing Chen is the corresponding author.

Third, across all these distributed architectures, systems suffer from brittle static optimizations. For instance, traditional runtime filters [11, 25, 51, 53, 85] trigger severe broadcast congestion and frequently become performance bottlenecks in low-selectivity queries.

We have designed and developed the distributed database system, TDSQL , which systematically addresses the OLAP challenges mentioned above. First, TDSQL introduces a centralized communication mechanism based on Forward Nodes within the traditional MPP framework, alleviating the "connection explosion" problem caused by data communication overhead. Second, TDSQL optimizes operator parallelism using a Virtual DN Parallelism strategy, achieving full pipeline parallelism at both the shard and operator levels. This significantly enhances intra-node parallel execution. Additionally, TDSQL incorporates a full-path vectorized execution engine that enables efficient batch processing of data [7, 8, 55–58, 67]. These strategies collectively improve utilization of computational resources and enhance query performance. Third, TDSQL applies a group-wise cascading communication strategy for runtime filters and an adaptive optimization strategy based on statistical information, effectively mitigating the negative impact of runtime filters.

We evaluate TDSQL based on the official TPC-DS benchmark [73] and achieve first place in the publicly reported results [74]. Under a 10,000 GB data size configuration, TDSQL achieves a score of 72,603,042 QphDS, outperforming the second- and third-place commercial systems by factors of 1.81 and 3.82, respectively. In the tests, the single-stream and multi-stream response times are 372.9 seconds and 731.9 seconds, respectively, showing a 56.0% and 74.0% reduction compared to the second-place system. The cost per 1000 QphDS is 37.52 CNY, which is 79% and 37% lower than the second- and third-place systems, respectively. Additionally, we evaluate TDSQL against open-source systems [46, 61, 68] such as Greenplum [46] and DuckDB [61] under the TPC-DS benchmark. Experimental results show that TDSQL achieves 4.34× higher query performance and superior scalability compared with Greenplum. In single-node evaluations, it further delivers 2.16× lower query latency than DuckDB, demonstrating its strong computational efficiency. Moreover, real-world performance tests also show that TDSQL consistently outperforms competitors by at least 40%.

The main contributions of this paper are summarized as follows:

- We introduce the design, implementation, and optimization of TDSQL, a distributed database system.
- We analyze current practices and introduce our solutions and corresponding implementations to address the challenges faced by current distributed OLAP databases.
- We conduct the official TPC-DS benchmark on TDSQL, achieving significant performance improvements in both score and cost efficiency. Furthermore, we demonstrate TDSQL's advantages through evaluations under real-world workloads.

2 BACKGROUND AND MOTIVATION

TPC-DS. This benchmark, defined by the Transaction Processing Performance Council, evaluates the performance of database systems through 99 complex SQL queries that model a retail decision-support workload. The queries cover multi-table joins, aggregations, sorting, and nested subqueries, stressing large-scale data processing and query-optimization capabilities. TPC-DS [73] is widely recognized as the standard for assessing modern analytical systems.

Table 1: Comparison to representative analytical open-source database systems. (Comm. = Communication, Op. = Operator, Vec. = Vectorized, Exec. = Execution, RT. = Runtime.)

Feature	GreenPlum	PrestoDB	ClickHouse	DuckDB	TDSQL
SQL Syntax Support	✓	✓		✓	✓
MPP Architecture	✓	✓	✓		✓
Centralized Comm.					✓
Full Op. Parallelism			✓	✓	✓
Vec. Exec. Engine			✓	✓	✓
Adaptive RT. Filter					✓
Columnar Storage	✓	✓	✓	✓	✓

Application Scenario. TDSQL, Tencent's enterprise-grade distributed database initiated in 2008 and open-sourced in 2019 [17], achieves the top rank in the official TPC-DS benchmark with a new global record. After over a decade of development, it has been widely deployed in finance, retail, telecom, and government sectors, serving more than 4,000 enterprise customers, including seven of China's top ten banks and PICC's core credit card platform. It supports a wide range of mission-critical analytical workloads, including operational analytics, regulatory reporting, and high-throughput log processing. For example, e-commerce platforms generate multi-terabyte logs within hours during major promotional events, requiring low-latency multidimensional analytics; telecommunications services require real-time analysis of call-detail records and network quality metrics. TDSQL has also been deployed for core workloads of a major domestic mobile phone manufacturer, covering billions of mobile devices and tens of millions of new security-related data entries daily. Details are in Section 8.6.

Revisiting Existing Distributed OLAP Databases. Distributed OLAP systems have evolved rapidly [1]. Early systems like Greenplum [46], Sybase IQ [47], Vertica [38], and Teradata DB [71] often used columnar storage but relied on heavy batch ETL. Later engines such as PrestoDB [68], Kylin [37], and Impala [36] pursued compute–storage decoupling and execution optimizations. Modern real-time OLAP systems, including Druid [79], Vedb-htap [12], ClickHouse [63], and Doris [23], support streaming ingestion and low-latency queries. Systems like HANA [26, 48, 62] extend analytics to data lakes via caching and elastic compute. Cloud platforms including Redshift [5], Snowflake [21], BigQuery [35], and Databricks [80] provide elastic large-scale analytics. DuckDB [61] offers high-performance embedded single-node analytics using vectorized execution.

Motivation. Despite remarkable progress, open-source and Cloud OLAP systems still face architectural and performance limitations that hinder scalability and execution efficiency. Table 1 compares four representative open-source systems (GreenPlum, PrestoDB, ClickHouse, and DuckDB) from key architectural dimensions. The comparison reveals several critical gaps that motivate TDSQL's design for large-scale complex analytical decision-making. First, most systems rely on traditional MPP architectures. Although such designs enable query parallelism, point-to-point connections cause communication overhead, scalability limits, and "connection

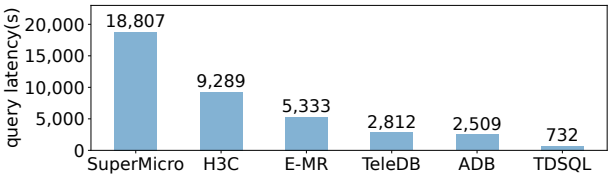


Figure 1: Query latency of DBMSs from TPC-DS public report

explosion" in large clusters. Cloud hyperscalers (e.g., Databricks) use decoupled architectures and external shuffles but incur network/I/O overhead from distributed storage, leaving massive scalability with native MPP's low-latency data movement unaddressed. Second, mature systems generally lack full operator-level parallelism and native vectorized execution. While some engines like DuckDB [61] have pioneered these paradigms, naively retrofitting them into mature architectures introduces severe intra-node coordination overheads and legacy compatibility barriers. Without a systematic co-design, CPUs are underutilized in complex workloads. Third, although runtime filters are prevalent in distributed OLAP systems, they suffer from two fundamental limitations: prohibitive communication overhead from flat full-mesh broadcasts and static evaluation brittleness, which causes performance regressions in low-selectivity scenarios as overhead outweighs pruning benefits. Finally, systems like ClickHouse are constrained by limited SQL support, while DuckDB is inherently bottlenecked by its single-node design in large-scale analytics.

Commercial OLAP databases exhibit similar architectural and performance limitations, leading to observable performance bottlenecks. Recent studies [14, 24, 70, 81] further show that under simulated real-world scenarios containing up to 10 TB of data, existing commercial databases still suffer from high query latency when executing complex workloads. Figure 1 illustrates the results of testing 99 SQL queries across multiple distributed OLAP databases from the TPC-DS open report [74]. Evidently, even the best-performing systems incur an average query latency of over 25 seconds under multi-stream workloads (2509 seconds over 99 queries). For high-load, real-time analytical environments, such latency is sometimes unacceptable. Therefore, designing a more efficient distributed OLAP system is urgently needed to meet demand for complex queries and large-scale analytical processing.

3 OVERVIEW

To tackle these challenges, TDSQL adopts a traditional MPP framework [46, 63, 70]. As Figure 2 illustrates, the GTM manages global transactions, the Coordinator handles query distribution, and Datanodes manage storage and execution. Furthermore, it integrates a custom columnar storage system, *Estore*, alongside execution-engine optimizations that significantly improve complex query efficiency.

Novelty. TDSQL introduces four key optimizations, summarized in Figure 2. First, we introduce Forward Nodes and establish a data forwarding intermediary layer, mitigating connection explosions from inter-node data communication in traditional MPP frameworks (Section 4). Second, we propose a Virtual DN Parallelism strategy for operator-level parallelism, which, when combined with the existing random parallelism, achieves efficient parallelization of

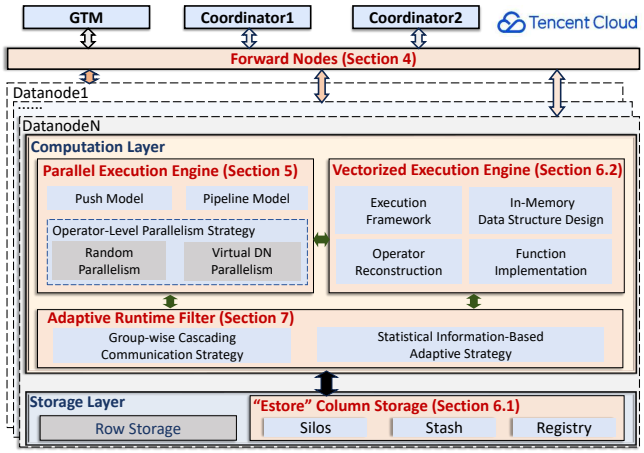


Figure 2: TDSQL system core architecture

all operators. Additionally, we replace the traditional pull execution with push and pipeline execution models, enabling fully asynchronous parallel execution of query plan partitions (Section 5). Third, we design and implement a native vectorized execution engine and its underlying columnar storage system, significantly improving TDSQL's query execution efficiency (Section 6). Fourth, we propose a group-wise cascading runtime filter communication strategy and an adaptive runtime filter application strategy based on historical data and statistical information, effectively reducing the communication overhead of runtime filters and mitigating the negative impact in low-selectivity scenarios (Section 7). Below, we organize these optimizations according to the bottlenecks they address: communication overhead, CPU efficiency, and runtime filter robustness.

(1) Topology Transformation via Forward Nodes. Traditional MPP architectures face the severe connection explosion problem during heavy data shuffling. To mitigate this without migrating to high-latency external storage services, we introduce the Data Forward Bus, composed of intermediary Forward Nodes. This architectural shift transforms the physical full-mesh topology into an efficient intra-server star topology. By multiplexing cross-machine connections and leveraging zero-copy shared memory for intra-machine communication, Forward Nodes alleviate connection bottlenecks while preserving strictly low-latency data movement.

(2) Co-designed Parallelization and Vectorization. To maximize CPU utilization, we systematically upgrade the execution engine across two dimensions. For parallelization, we introduce a two-tier strategy: (a) At the plan fragment level, we replace traditional pull execution with a push model [65], uniquely orchestrated by a Pipeline mechanism to ensure fully asynchronous execution while preventing resource overload. (b) At the operator level, we propose a Virtual DN Parallelism strategy. By enforcing strict intra-node data partitioning, it overcomes the inherent limitations of traditional random parallelism, enabling strictly lock-free parallel execution for complex operators that were previously unparallelizable. For vectorization, to address compatibility challenges within a mature ecosystem, we implement a non-intrusive execution framework built atop a hybrid columnar storage engine (Estore). By synergizing adaptive row-vector adapters with redesigned zero-copy memory

structures and SIMD-accelerated operators, it enables legacy scalar execution to seamlessly coexist with high-performance batch processing, drastically improving computational efficiency without requiring pervasive architectural refactoring.

(3) Hierarchical and Adaptive Runtime Filtering. To overcome the inherent scalability bottlenecks of runtime filters in shared-nothing architectures, we propose a novel group-wise cascading communication strategy. Unlike traditional flat broadcasting that triggers network congestion, this mechanism leverages Forward Nodes to perform local aggregation before inter-group transmission, effectively transforming the full-mesh "broadcast explosion" into a controlled hierarchical exchange. Complementing this, we introduce a multi-level adaptive framework that pioneers the integration of statistics-based construction checks with history-aware sliding-window bypassing. This dual-layer innovation ensures that TDSQL maintains maximum pruning efficacy while eliminating the performance regressions common in low-selectivity scenarios.

4 FORWARD NODE ARCHITECTURE

Efficient intermediate data movement, particularly the heavy shuffling inherent in distributed joins, remains a critical performance bottleneck in modern distributed OLAP systems [13]. Existing data exchange mechanisms face a latency–scalability trade-off and typically fall into three paradigms. First, traditional MPP architectures, such as Greenplum [46] and Vertica [38], rely on tightly coupled full-mesh point-to-point connections; although this design is low-latency, it suffers from connection explosion and resource contention as clusters scale. Second, cloud-native hyperscalers, including Databricks [80] and BigQuery [35], improve scalability through decoupled external shuffle services that write intermediate data to dedicated nodes or distributed storage, but the resulting disk I/O and multi-hop network overheads compromise low-latency targets. Third, engines such as PrestoDB [68] and ClickHouse [63] use TCP connection multiplexing or per-server connection pools to mitigate socket exhaustion, but these incremental optimizations do not address the root cause of topological inefficiency.

TDSQL employs an MPP architecture with a multi-Datanode, multi-process deployment per server for production-grade process-level isolation, confining resource contention and failures to individual Datanodes rather than impacting the entire server. To mitigate the associated connection overhead, we introduce Forward Node (FN) as an intermediate layer that replaces full-mesh P2P communication. FN unifies intra-server zero-copy shared memory and inter-server FN-to-FN connection multiplexing, cutting physical connection costs while retaining low-latency data transfer. Unlike Hyper Dimension Shuffle [59], which restructures the dataflow graph to bound fan-in and fan-out, FN optimizes only the physical communication path without altering the logical shuffle semantics.

4.1 Data Communication Layer

Analysis. Traditional MPP architectures face severe connection explosion during data redistribution. When the system is deployed across n servers with a total of N data nodes, during the data redistribution phase, each Datanode needs to establish point-to-point network connections with $(N-1)$ other Datanodes to perform data transfer, leading to a system connection scale of $O(N^2)$. When

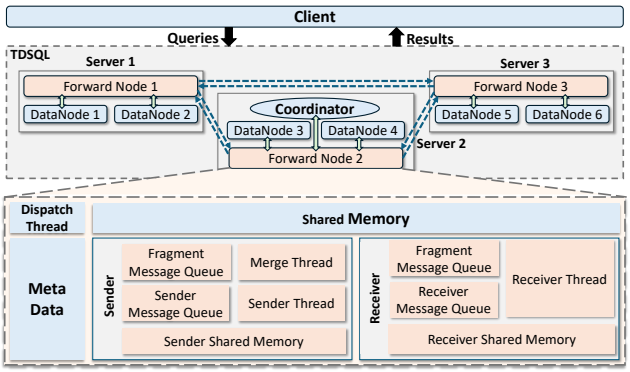


Figure 3: Forward Node-based transmission architecture

the system supports X concurrent queries, the number of connections will expand to $O(X \cdot M \cdot N^2)$ due to the complexity of queries often involving M levels of join operations. This connection growth consumes limited system resources such as memory and incurs overhead from socket creation, maintenance, and teardown, thereby limiting scalability.

Design. To address this, we propose a centralized data redistribution architecture based on the Forward Node as a data interaction bus, as shown in Figure 3. The core idea is a dedicated data forwarding intermediary layer, the Data Forward Bus, which is composed of Forward Nodes. A Forward Node is deployed on each server as the data communication proxy for that server. Within the server, the Forward Node exchanges data with the local Coordinators and Datanodes via a shared memory mechanism, avoiding the need for inter-node connections. Across servers, all Coordinator and Datanodes route data through the local Forward Node, with cross-server data transmission carried by the connections between Forward Nodes. Now, regardless of the query complexity or concurrency, the entire cluster's connection is limited to $O(n \cdot (n-1))$.

4.2 Modular Architecture

As shown in Figure 3, the Forward Node features a modular design with four core components. First, the metadata management module maintains communication metadata between the Forward Node and Coordinators/Datanodes, including process identifiers, etc. Second, the distributed shared memory module allocates dedicated memory spaces for each Datanode for data sharing, storing node statistics, receive queue states, and other related information. Third, at the data transmission layer, it establishes a bidirectional communication mechanism through the Sender and Receiver modules. The Sender module uses sender shared memory to buffer Fragment messages; a merge thread performs message merging operations, while the sender message queue and sender thread handle message scheduling and dispatching. Fourth, the Receiver module comprises receiver shared memory, receiver threads, and a receiver message queue, ensuring reliable data reception and processing.

4.3 Data Messages

Data messages serve as the core carriers for user data transmission and are fully routed and transmitted by Forward Nodes. They can

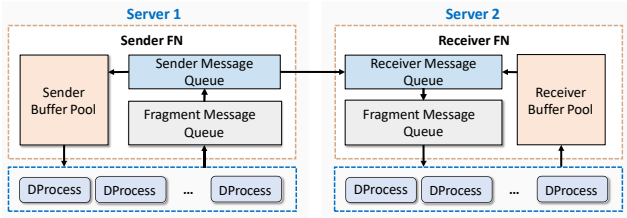


Figure 4: Data communication between Forward Nodes

be classified into two typical modes: intra-server communication and inter-server communication.

Intra-server Communication. TDSQL utilizes shared memory to implement zero-copy data transmission, enabling connectionless communication between nodes on the same server. When Coordinators and Datanodes on the same physical server need to exchange data, the source node directly writes the data into a pre-allocated memory area via the shared memory interface. The Forward Node, as the manager of the memory area, is responsible for routing and distributing the data. The target node reads the required data via the same shared memory interface.

Inter-server Communication. TDSQL adopts an indirect routing mechanism based on Forward Nodes. As shown in Figure 4, a source node requests shared memory from its local Forward Node, writes the data, and immediately returns, enabling asynchronous submission. The Forward Node's scheduling thread retrieves messages from the Fragment message queue, identifies their destinations using the fragment identifier (FID), and routes them to the corresponding sender message queues. Dedicated sending threads then perform network transmission, after which the memory block is returned to the sender buffer pool. At the receiving Forward Node, incoming data are stored in memory allocated from the receiver buffer pool and placed in the receiver message queue. The scheduling thread uses the FID to route each message to the corresponding Fragment message queue, where the target process reads the data and releases the memory block. By combining zero-copy shared memory, asynchronous message queues, and multi-threaded scheduling, TDSQL decouples data sending and receiving and improves inter-server transmission efficiency.

5 PARALLEL EXECUTION

Inefficient parallel execution is a primary bottleneck for complex analytical queries in distributed OLAP databases; TDSQL addresses this via two key architectural optimizations in its parallel execution framework. First, at the query plan fragment level, TDSQL shifts from the traditional pull-based model (used in Greenplum [46] and Impala [36]) to a push-based execution model, an established practice in modern engines like ClickHouse [63]. To avoid resource contention and system overload caused by naive push-based execution, TDSQL further introduces a Pipeline execution scheme with resource-aware orchestration, enabling safe asynchronous parallel execution. Second, at the operator level, intra-node multi-core execution suffers from two shortcomings: shared-memory engines have severe lock contention and cache thrashing, while morsel-driven engines (e.g., DuckDB [61]) use thread-local states to avoid locks but face expensive final merge bottlenecks. TDSQL addresses these limitations with a virtual Datanode (VDN) architecture. RemoteSubplan redistributes data across isolated VDNs to ensure data-range independence, enabling lock-free, zero-merge parallel execution for complex operators. Combined with a deterministic hybrid strategy that falls back to random parallelism based on upstream input distribution semantics, this forms TDSQL's core execution innovation to enhance node-level computation.

5.1 Fragment-Level Parallelism Optimization

Analysis. Many distributed database engines adopt a pull-based execution model [29], as exemplified by the Volcano iterator model [30], where consumers request data from producers. Although Volcano-style execution can support parallelism through exchange operators, the original demand-driven fragment framework may still suffer from synchronization waits and limited coordination among concurrently runnable fragments under a bounded resource budget.

Design. Unlike Volcano-style parallelism, which uses exchange operators within a pull-driven pipeline, TDSQL focuses on push-based cross-fragment orchestration. During query plan execution, lower-level plan fragments proactively push data to the higher-level plan fragments in real time, avoiding the synchronization waits in the original demand-driven framework. Meanwhile, TDSQL introduces a data thread mechanism, which performs asynchronous prefetching and data structure transformation of the pushed data, enabling the higher-level fragments to obtain the required data in a timely manner. This approach eliminates the dependency on the state and progress of other fragments, ensuring parallel processing of multiple plan fragments. This mechanism enables logical decoupling between plan fragments and fully asynchronous execution, significantly enhancing parallelism and query performance.

In resource-abundant environments, executing multiple plan fragments concurrently can significantly improve overall execution efficiency. However, in resource-constrained environments, excessive parallelism can lead to system overload. To address this, we have designed a Pipeline-based execution architecture. TDSQL organizes plan fragments into multiple Pipelines based on the producer-consumer relationships in the query plan tree and arranges their execution according to logical dependencies. Within each pipeline, different plan fragments are executed asynchronously in parallel. TDSQL allocates parallel threads to each plan partition as needed, ensuring that the total parallelism within a pipeline does not exceed TDSQL's maximum supported parallelism. This approach avoids the risk of system overload caused by simultaneously executing multiple plan fragments in parallel. Through this mechanism, TDSQL dynamically adjusts the parallelism of plan fragments within the Pipeline based on their complexity, thereby achieving better resource allocation and scheduling.

Example. As shown in Figure 5, the query plan can be divided into five distinct plan fragments. During execution, these fragments use the push model and can be executed in parallel without waiting for each other. If the query is run in a resource-constrained environment, parallel execution of multiple `HashJoin` operators across plan fragments may cause system overload. Thus, we utilize the pipeline execution mechanism. As shown in Figure 6, assuming TDSQL's maximum supported parallelism is 8, TDSQL organizes

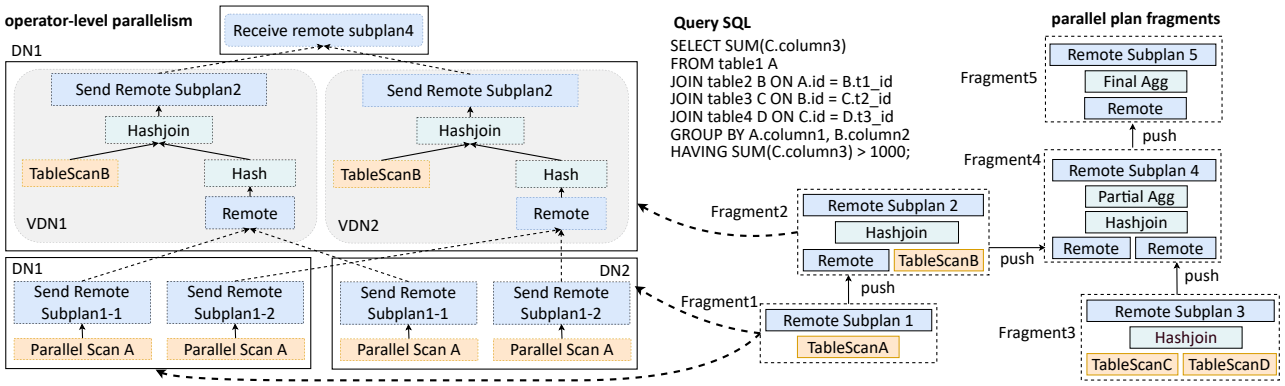


Figure 5: A parallel query plan

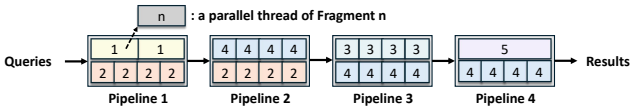


Figure 6: Pipeline-based execution plan

the five query plan fragments into four pipelines and establishes the corresponding execution flow based on their dependency relationships. While ensuring that the parallelism within each pipeline does not exceed 8, parallel threads are allocated to the two plan fragments in each pipeline as needed, thereby fully utilizing computing resources while avoiding system overload.

5.2 Operator-Level Parallelism Implementation

Analysis. Within the plan fragments of a node, traditional operator parallelism often relies on random parallelism, where input tuples are randomly assigned to multiple parallel worker threads without enforcing key-range independence [46]. While this random parallelism mechanism can improve query performance in certain scenarios, it is highly dependent on the coordination among workers, which presents significant limitations. Due to the random distribution of data across workers and the lack of a globally consistent partitioning strategy, hash buckets constructed by different workers may conflict, making it difficult to correctly aggregate local computation results. This issue renders parallelization of operators such as window aggregation and full join challenging. Additionally, the frequent coordination and data exchanges between workers introduce extra communication and synchronization overhead, significantly affecting the efficiency of parallel hash computations.

Design. To address these limitations, TDSQL introduces Virtual DN Parallelism, an intra-node partitioned execution strategy based on virtual Datanodes. This abstraction is related to exchange-style redistribution in parallel query engines such as PrestoDB [64, 68] and SQL Server [31, 39, 52], where tuples are locally routed among parallel workers according to partitioning requirements. In contrast, TDSQL adapts and integrates this idea into a virtual Datanode abstraction tailored to its MPP runtime. Within a node, TDSQL simulates multiple virtual Datanodes at the software level; each virtual Datanode owns an independent data range and executes operators autonomously, thereby reducing worker coordination overhead and enabling correct parallel execution for partition-sensitive operators. To ensure data independence across virtual Datanodes, the `RemoteSubplan` operator acts as the intra-node distribution boundary, redistributing tuples according to the downstream operator's partitioning requirements and maintaining partition consistency for the operator's results. With Virtual DN Parallelism, operators such as window aggregation and full join can be parallelized within a node, while the execution efficiency and parallelism of hash join operations are also significantly improved. The Virtual DN Parallelism strategy not only provides flexible adaptation to different operators' parallelism but also achieves near-linear performance scaling with the increase in the number of virtual Datanodes, thus offering dynamic tuning capabilities for diverse workloads.

In the Virtual DN Parallelism strategy, TDSQL relies on the `RemoteSubplan` operator for data redistribution, ensuring that data is partitioned across virtual Datanodes according to the parallelism degree and maintaining data range independence. However, its reliance on `RemoteSubplan` limits the applicability of Virtual DN Parallelism to certain operators. To address this limitation, we propose a hybrid operator-level parallelism strategy that chooses between random parallelism and Virtual DN Parallelism based on the distribution semantics of an operator's upstream inputs. An operator adopts Virtual DN Parallelism only when its upstream contains a `RemoteSubplan` operator that provides a key-partitioned input; otherwise, it falls back to random parallelism. Data skew is further mitigated because `RemoteSubplan` performs range-based repartitioning using data distribution statistics, ensuring balanced data volumes across virtual Datanodes. This deterministic decision rule enables fine-grained operator parallelism while preventing redundant data movement and improving execution efficiency.

Example. As shown in Figure 5, TDSQL applies differentiated operator execution strategies to the operators in fragment1 and fragment2 during query execution: (1) For the `TableScan` operator in fragment1, since there is no `RemoteSubplan` operator in the upstream, TDSQL uses traditional random parallelism for parallel execution of `TableScan` operators; (2) In fragment 2, for complex operators such as `Hashjoin` with upstream `RemoteSubplan` operators, TDSQL adopts the Virtual DN Parallelism strategy, ensuring

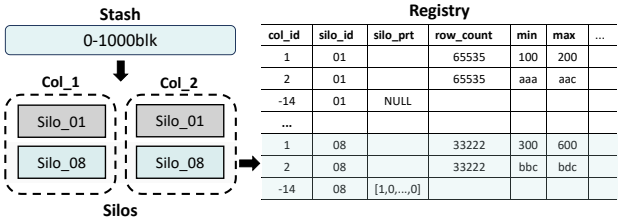


Figure 7: The storage design of *Estore* in TDSQL

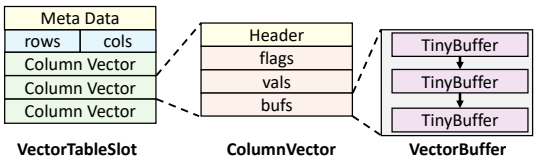


Figure 8: In-Memory data structures for vectorized execution

data isolation and correct result aggregation through pre-execution data redistribution.

6 SYSTEMATIC VECTORIZATION DESIGN

Vectorization and columnar storage have become de facto standards in modern OLAP systems, as exemplified by Databricks [80] and DuckDB [61]. However, integrating a native vectorized engine into TDSQL's mature ecosystem without degrading legacy workload performance remains a significant architectural challenge. Rather than reinventing vectorization fundamentals, our work focuses on a non-intrusive system co-design for seamless integration.

To bridge the gap between transactional ingests and analytical scans, TDSQL's Column Storage Engine (Estore) employs an LSM-like architecture, leveraging a row-oriented Stash to absorb mixed inserts without fragmenting the underlying columnar (Silos). Building on this storage foundation, we introduce four core non-intrusive execution upgrades: (1) an adaptive row-vector adapter that dynamically connects legacy operators with vectorized subtrees for hybrid execution; (2) a redesigned in-memory architecture featuring ColumnVector for efficient batched processing, backed by VectorBuffer linked lists to eliminate costly deep memory copies; (3) vectorized reconstruction of key operators (e.g., HashAgg, HashJoin) via Selection Vectors and bitmap-driven probing to enable branchless execution; and (4) a metadata-driven binary-search mapping framework that extends SIMD acceleration to thousands of existing scalar functions with minimal refactoring.

6.1 *Estore* Column Storage Engine

This part details how our design handles the impedance mismatch between high-throughput mixed inserts and columnar analytical scans [22, 82]. This paper implements a hybrid storage structure, Estore, tailored to our ecosystem to enhance both query performance and storage efficiency.

Our storage structure, as shown in Figure 7, organizes each column's data in a separate file, divided into *silos*, the fundamental storage and processing units. Each silo holds a fixed number of rows for one column with variable physical size. The *registry* table serves as the metadata catalog and forms the core organizational structure of the *Estore*. Each row in the registry corresponds to the metadata of a specific silo and facilitates system-level management of all silos. Additionally, the *stash* table is a row-oriented heap structure designed to accommodate fragmented data, streaming data, and OLTP-style mixed workloads. Functioning as a "staging area" for *Estore*, the stash table temporarily buffers small inserts when the number of inserted rows falls below a predefined threshold. These rows are stored in row format within the stash to avoid fragmentation in the registry/silo layout. TDSQL periodically merges the contents of the stash table into the appropriate silos.

6.2 Vectorized Execution Engine

Effective vectorized execution depends on both columnar-storage locality and the underlying execution model. To this end, TDSQL architects a non-intrusive vectorized execution engine framework, including dynamic vectorized transformation of query plans, design of in-memory data structures, vectorized reengineering of operators, and vectorized implementation of scalar functions.

6.2.1 Vectorized Execution Framework. The vectorized execution framework improves query performance by transforming tuple-at-a-time processing into column-vector-based batch processing, thereby exploiting the efficiency of vectorized computation.

To vectorize a query plan, the framework greedily traverses and rewrites the plan tree in a bottom-up manner. For each plan node, TDSQL checks whether the node, its expressions, and the entire subtree support vectorized execution. If so, the subtree runs in vectorized mode; otherwise, TDSQL inserts a row-vector conversion adapter between vectorized and non-vectorized operators. This preserves correctness and compatibility without rewriting the legacy execution engine.

6.2.2 Data Structures in Memory. In traditional row-based storage, data is organized in memory using `TupleTableSlot`, where data transfer between operators during query execution occurs at the granularity of a `TupleTableSlot`. However, this organization does not effectively support batched computation and column-oriented data transfer needs. To overcome this limitation, TDSQL designs a new memory organization for vectorized computation.

Memory Design. As shown in Figure 8, TDSQL introduces the `VectorTableSlot` data structure to represent tuple vectors, with the `ColumnVector` serving as the core data structure for column vectors. The `VectorTableSlot` acts as a container for multiple `ColumnVector` instances, which hold the vector data for each column, as well as maintains relevant metadata and system vector information. The `ColumnVector` stores the operands involved in the computation, thus supporting batched operations.

To enhance memory efficiency, `ColumnVector` classifies data by its length. For fixed-length data not exceeding 8 bytes, the data is directly stored in the `vals` array of `ColumnVector`; for fixed-length data exceeding 8 bytes and variable-length data, an indirect storage mechanism is employed, with data stored in memory blocks pointed to by the `buf` pointer. The `vals` array holds only the pointer to the start of the data. Additionally, `ColumnVector` tracks column data consistency features through parameters and uses a `flags` array

to maintain metadata such as null states and compression status, which supports optimization during subsequent computations.

To address memory management challenges in large-scale data storage, TDSQL employs `VectorBuffer` to manage memory blocks uniformly. This structure is implemented as a linked list of multiple `TinyBuffer` nodes, with each `TinyBuffer` representing a contiguous block of available memory. Through this self-maintaining buffer management mechanism, `TinyBuffer` ensures that variables are stored contiguously, thereby minimizing cache misses and improving memory access efficiency. Moreover, the separate storage of larger variables reduces memory copying between operators, facilitating faster memory reuse.

6.2.3 Vectorized Operator Reconstruction. TDSQL's execution engine vectorizes key operators, including `RemoteSubplan`, `Scan`, `HashAgg`, and `HashJoin`, achieving a full transition to batch-oriented processing. Since `HashAgg` and `HashJoin` dominate both query performance and CPU time, the following discussion focuses on their vectorized design and implementation.

HashAgg Operator Vectorization. TDSQL implements `HashAgg` using xxHash [20] and linear probing, with a two-tier Pointer Block–Payload Block hash-table structure [88]. When the hash table exceeds the memory threshold, intermediate data are spilled to disk and subsequently reloaded for merge processing. During vectorized execution, a SelectionVector partitions batched probing results into absent-key, existing-key, and collision groups, which are processed separately for insertion, update, and continued probing. This branch-reduced processing improves pipeline efficiency, while SIMD instructions accelerate hashing and key comparison.

HashJoin Operator Vectorization. TDSQL vectorizes both the build and probe phases of `HashJoin`. In the build phase, inner-table tuples are hashed in batches to construct the hash table. In the probe phase, outer-table tuples are processed in batches, and matched positions are recorded in a bitmap to support branchless batch output. SIMD instructions accelerate hashing and key comparison, while parallel hash-table construction and probing exploit task-level parallelism, thereby reducing per-tuple processing overhead.

6.2.4 Vectorized Function Implementation. In columnar storage systems, traditional scalar functions are typically designed to operate on individual data elements. When processing columnar vector data, these functions require row-by-row invocation, which limits the potential for batch processing and SIMD computation [8]. To address this limitation, this paper proposes a metadata-driven function mapping mechanism that, while preserving the existing data type architecture, only implements vectorized versions of the functions and establishes a mapping relationship between scalar and vectorized functions. During the vectorized execution phase, TDSQL efficiently locates and invokes the vectorized functions using a binary search algorithm. This design avoids rewriting thousands of functions, reducing implementation complexity and code intrusion while preserving scalability and maintainability.

7 RUNTIME FILTER OPTIMIZATION

TDSQL systematically implements three types of runtime filters, Bloom Filter [44, 69, 78], Range Filter [3, 45], and In Filter, to accommodate diverse data distributions and query patterns.

Analysis. While runtime filtering is an industry standard, scaling it in shared-nothing architectures exposes two critical bottlenecks demanding architectural co-design. First, severe broadcast overhead. In Shuffle Joins, traditional MPPs, such as Greenplum [46] and PrestoDB [68], suffer from severe network congestion due to full-mesh broadcasting. Meanwhile, cloud-native hyperscalers like Databricks [80] circumvent this via decoupled storage-layer pruning, leaving the compute-tier broadcast problem unresolved for native shared-nothing engines. Second, evaluation rigidity. In low-selectivity scenarios, static Cost-Based Optimizers (CBOs) often mandate the costly broadcast of useless filters because their cardinality estimations inevitably degrade after complex multi-table joins. Conversely, pure execution-time bypassing mechanisms, as seen in ClickHouse [63], still incur the initial sunk costs of filter construction and transmission.

Group-wise Cascading Communication Strategy. As a native shared-nothing engine, TDSQL cannot circumvent transmission bottlenecks via decoupled storage privileges. Instead, we mitigate the all-to-all broadcast explosion by introducing a hierarchical runtime filter transmission mechanism based on group aggregation and cascaded transmission. Specifically, multiple Datanodes within the same physical machine are organized into a group, where the Forward Node aggregates the locally generated runtime filters from all Datanodes to construct a group-level runtime filter. Next, each group's Forward Node performs a single inter-group broadcast to deliver the group-level runtime filter to the target machines. Upon receiving external group-level filters, the Forward Node of the target machine merges them with its locally generated filters to form the final global runtime filter, which is then distributed to all local Datanodes. This scheme significantly reduces redundant cross-node transmissions and effectively alleviates the communication overhead of large-scale runtime filters in Shuffle Joins.

Statistical Information-Based Adaptive Strategy. Rather than falling into the extremes of rigid static CBOs or pure reactive bypassing, we introduce a multi-level adaptive strategy to robustly mitigate performance degradation in low-selectivity scenarios.

During the construction phase, TDSQL adaptively determines whether to generate a runtime filter based on the estimated filtering effectiveness derived from statistical information. Consider a join between tables T and S on attribute Z, where NDV denotes the number of distinct values, and $NDV_T(Z)$ and $NDV_S(Z)$ denote the NDVs of attribute Z in T and S, respectively [41]. Assume that T is the probe-side base table and S is the build-side table. The filtering effectiveness of constructing a runtime filter is estimated as $\textit{filtering\_effectiveness} = 1 - \frac{NDV_S(Z)}{NDV_T(Z)}$. TDSQL sets a predefined threshold β; runtime filters are disabled if estimated filtering effectiveness drops below β.

When runtime filters are enabled, TDSQL incorporates a history-aware adaptive mechanism during the probe phase. As shown in Algorithm 1, during batch data processing, TDSQL employs a sliding-window approach to continuously monitor the filtering rate of previous batches and dynamically determine whether to apply the filter to the current batch. If the filtering rate of the previous batch is insufficient, TDSQL skips filtering for the current batch. Moreover, if two consecutive batches within the sliding window exhibit poor filtering rates, TDSQL disables runtime filters for both

Algorithm 1: Dynamic adaptive filtering algorithm

Input: $X = \{X_1, X_2, \ldots, X_n\}$, where X_i is the i-th batch with $|X_i|$ records, and $|x_i|$ denotes the number of records filtered out from the i-th batch, α is a system-defined threshold.
Output: The filtered results.

```
for i = 1 → n do
    F_i ⇐ |x_i| / |X_i|
    if i == 1 or F_{i-1} ≥ α then
        Enable Runtime Filter ;
    else
        if i == 2 or F_{i-2} ≥ α then
            Disable Runtime Filter ;
        else
            Disable Runtime Filter for current and next batches ;
            i ← i + 1 ;
return The filtered results;
```

the current and the next batch. This mechanism effectively mitigates the efficiency loss incurred when runtime filters are applied to specific batches of data with low selectivity.

These strategies enable TDSQL to adapt runtime filters dynamically based on historical performance and data traits, maximizing pruning efficacy while ensuring correctness and avoiding slowdowns in low-selectivity cases.

8 EVALUATION

8.1 Experimental Setup

Configuration. The experiments are conducted on a cluster comprising eight physical machines, each equipped with an Intel Xeon Platinum 8163 CPU @2.50GHz (112 vCPUs), 1024 GB memory, and a two-tier storage architecture consisting of 2×960 GB SATA SSDs for system storage and 16 × 7.68 TB NVMe SSDs for high-performance data storage. All machines are interconnected via a high-speed 100 Gbps Silver Fir Intelligent NIC network infrastructure.

Methodology. We compare TDSQL [19] with three open-source OLAP systems: GreenPlum [46], PrestoDB [68] and DuckDB [61]. GreenPlum (now no longer open-source) employs a traditional MPP framework and does not incorporate optimizations related to parallel and vectorized execution. PrestoDB adopts a decoupled storage and computation strategy, using the Apache ORC file format for data storage and integrating HDFS as the storage layer, with partial vectorization improvements implemented. DuckDB, as a single-node OLAP database, natively implements vectorization but does not support clustering. We also compare two commercial systems, AnalyticDB [81] and TeleDB [70], via official TPC reports [74]. In addition, we compare TDSQL with representative modern Cloud SQL warehouses, i.e., Databricks [80] and Snowflake [21]. We use a Databricks X-Large SQL warehouse for the 1 TB experiment in Figure 9, and Databricks 5X-Large and Snowflake 6X-Large warehouses for the 10 TB comparison in Table 3, all on Microsoft Azure. Since cloud warehouse sizes are vendor-managed and not hardware-identical, we report these results as practical cloud baselines rather than hardware-normalized comparisons.

Workload. To ensure evaluation accuracy and reliability, we adhere to standard TPC-DS specifications, comprehensive guidelines for implementing the TPC-DS decision support benchmark [73] that enable fair, reproducible performance assessments of complex analytical queries across database systems. We run benchmark tests on the TPC-DS workload with a default scale factor of 10000 (10 TB). Beyond the benchmark, we also evaluate TDSQL on real production OLAP workloads, including security analytics over billions of mobile devices generating tens of millions of new entries daily, and retail analytics over hundreds of millions of sales and inventory records, involving complex joins and aggregation queries.

Performance Metric. We use single-stream and multi-stream to describe benchmark-level query-stream concurrency. The single-stream setting runs one TPC-DS query stream, while the multi-stream setting runs multiple query streams concurrently. Pt_time and tt_time represent the total execution times for 99 queries in single-stream and multi-stream TPC-DS configurations [73]. Response time refers to either of these metrics, depending on the context. QphDS represents the database system's overall performance score according to the TPC-DS benchmark. Latency denotes the execution time of a single query, while CPU usage reflects the real-time average CPU utilization across all cores. All results are collected by averaging three executions of the respective tasks.

Table 2: TPC-DS publicly official reports

	Rank	Performance (QphDS)	pt_time	tt_time	Price/kQphDS (CNY)
TDSQL	1	72.60M	372.9s	731.9s	¥ 37.52
TeleDB	2	40.20M	847.8s	2811.6s	¥ 179.73
AnalyticDB	3	18.99M	849.9s	2508.5s	¥ 59.27

8.2 TPC-DS Results

Comparison with Commercial Systems. In the standard TPC-DS 10,000 GB benchmark test, TDSQL demonstrates exceptional performance, achieving the first-place results in the open report [74]. Table 2 summarizes the performance of TDSQL and compares it with the second-place system, TeleDB [70], and the third-place system, AnalyticDB [81, 83]. In the standard test suite consisting of 99 complex SQL queries, TDSQL achieves a processing capacity of 72,603,042 QphDS with a 128-node cluster configuration, outperforming TeleDB and AnalyticDB by 1.81× and 3.82×, respectively. In this test, TDSQL exhibits outstanding computational capabilities, completing the 99 complex analytical queries in 372.9 seconds under the single-stream setting and 731.9 seconds under the multi-stream setting. In both cases, TDSQL demonstrates clear performance advantages over TeleDB and AnalyticDB.

Also, TDSQL excels in data management and cost efficiency. TDSQL completes the loading of 10,000 GB of data in just 319.1 seconds, while the execution time for data manipulation operations is merely 25 seconds, further validating TDSQL's outstanding efficiency in large-scale data loading and maintenance tasks. As shown in Table 2, TDSQL demonstrates significant cost-effectiveness advantages, as reflected by the price/performance metric. Compared to TeleDB and AnalyticDB, TDSQL reduces the cost per 1000 QphDS by 79% and 37%, respectively. This achievement sets a new performance record in the TPC-DS benchmark [74].

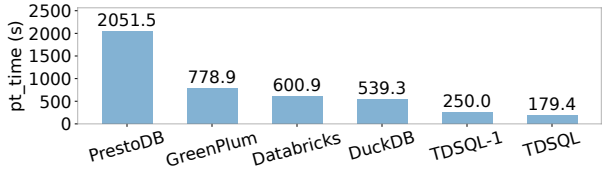


Figure 9: Comparison with Open-source and Cloud Systems

Comparison with Open-Source and Cloud Systems. To further assess, we conduct a 1000 GB TPC-DS benchmark test. Figure 9 presents a comparison of query latency across different systems. TDSQL's pt_time for 99 queries is 179.4s, outperforming traditional GreenPlum, validating its vectorized execution, parallel optimization, and runtime filter gains. PrestoDB has much higher latency due to its disaggregated storage-computation architecture. While flexible, it requires disk-to-memory data loading, adding heavy I/O overhead. Since DuckDB is designed for efficient single-node execution and lacks a distributed version, we evaluate it on a single-node setup matching our test environment. In this setup, DuckDB performs exceptionally well, surpassing distributed systems like GreenPlum and PrestoDB in query efficiency, reflecting the high efficiency of its execution engine. However, DuckDB's scalability in large-scale datasets is limited by the memory capacity and computational resources of a single machine. To further validate the computational capability of TDSQL in a single-node environment, we test TDSQL in such a setup. As shown in Figure 9, Single-node TDSQL (TDSQL-1) achieves a query latency of 250.0 seconds, compared to DuckDB's 539.3 seconds, demonstrating the superior query performance of TDSQL in single-node environments. Beyond open-source comparisons, we also compare our system against Databricks. Provisioned with an X-Large configuration to match TDSQL-1's compute capacity, this decoupled architecture prioritizes horizontal scalability and elastic storage. In contrast, TDSQL focuses on maximizing per-core efficiency through tight data integration. Consequently, TDSQL-1 maintains highly competitive response times, making it suitable for compute-dense scenarios. We further extend the cloud-warehouse comparison to the 10 TB workload in Table 3. TDSQL completes the 99-query single-stream run in 372.90 seconds, whereas Databricks and Snowflake take 782.36 seconds and 726.15 seconds, respectively. Under these practical cloud baseline configurations, TDSQL achieves 2.10× and 1.95× shorter response times, showing its efficiency on large-scale TPC-DS workloads.

8.3 Ablation Study

We conduct a series of ablation studies targeting the proposed key designs. By systematically removing specific optimization components, we construct multiple system variants for comparative analysis. Specifically, we design three single-ablation variants: the variant without parallel optimization strategies (w/o parallelism, w/o par), which relies on traditional parallel processing mechanisms; the variant without vectorized execution (w/o vectorization, w/o vec), which disables the vectorized execution engine and reverts to the traditional tuple-at-a-time execution model; and the variant without runtime filter (w/o runtime filter, w/o fil), which

Table 3: 10 TB TPC-DS comparison with cloud warehouses

System	TDSQL	Databricks	Snowflake
pt_time (s)	372.90	782.36	726.15

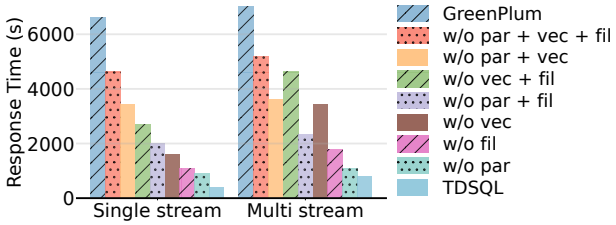


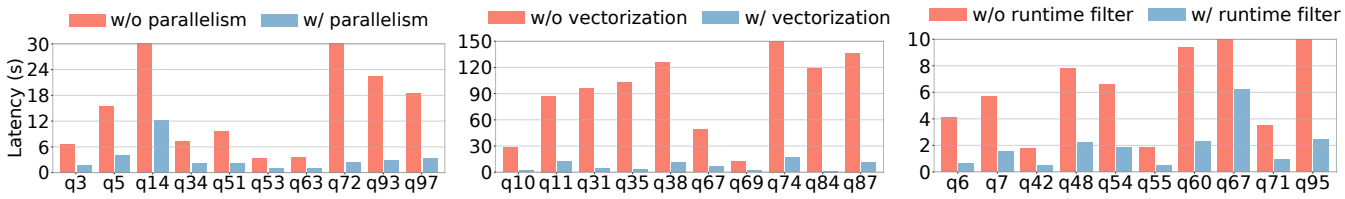
Figure 10: Comparison between TDSQL and all variants

eliminates the runtime filter mechanism. On top of these, we further construct combined ablation variants to assess the synergistic effects of multiple optimizations.

Overall Ablation Results. We evaluate TDSQL (full system and ablation variants) and GreenPlum using the TPC-DS benchmark's 99 standard queries, testing single-stream and multi-stream concurrent workloads as shown in Figure 10. The vectorized execution engine cuts overall query time by more than 70% in both workloads, highlighting the gains of batch-oriented processing. The runtime filter cuts total run time by 65.1% and 55.2% in single-stream and multi-stream settings, respectively, demonstrating its effectiveness in dynamically pruning irrelevant data and mitigating network transfer overhead. The adoption of the proposed parallel optimization strategies consistently improves query performance across workloads, thereby underscoring their effectiveness in query computation and resource management. Furthermore, our results reveal that the combination of multiple optimization techniques produces compounding effects, substantially outperforming any single optimization alone. Notably, all TDSQL variants outperform GreenPlum. Since GreenPlum follows a traditional MPP communication model, this comparison is consistent with the benefit of Forward Nodes in reducing inter-node communication overhead for complex distributed queries.

Detailed Analysis of Ablation Impact. Next, we perform a fine-grained analysis of the response times for 99 queries under the single-stream setting in TDSQL and its three individual ablation variants. Furthermore, Figure 11 quantitatively evaluates the performance characteristics of each optimization by reporting the ten queries with the largest latency reductions, where query IDs are sorted within each subfigure.

First, enabling parallel execution optimizations improves 83% of queries, with 53% achieving more than a 50% speedup. As shown in Figure 11a, the top 10 queries with the most pronounced performance gains typically involve large-scale joins between fact tables as well as complex aggregations such as multi-level `GROUP BY` and `Window Agg` operations. In these scenarios, the combined use of the `RemoteSubplan` operator and the Virtual DN Parallelism strategy enables highly parallel execution of complex operators such as



(a) Parallel optimization strategy **(b) Vectorization execution** **(c) Runtime filter**

Figure 11: Impact of ablation on TPC-DS query latency

HashAgg. Furthermore, by performing fully asynchronous scheduling of plan fragments in conjunction with resource-aware load distribution, TDSQL achieves efficient parallel execution of fragment tasks. Overall, the adoption of parallel execution strategies substantially improves the utilization of computational resources and significantly reduces the response time of complex queries.

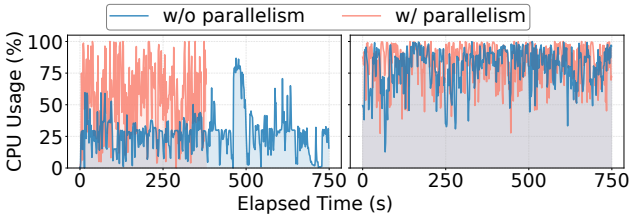
Second, utilizing the vectorized execution engine improves 78% of the 99 evaluated TPC-DS queries, with 46% achieving more than a 50% speedup. This shows optimization effects across the vast majority of queries. As shown in Figure 11b, we list the top 10 queries with the most significant response time improvements after enabling the vectorized execution engine. Analysis indicates that these queries typically involve computation-intensive operations, such as complex aggregations and large-scale sorting. The vectorized execution engine transforms the traditional row-by-row processing mode into a batch processing mode and leverages SIMD instruction sets, effectively reducing data transfer overhead and latency associated with upstream and downstream dependencies. This not only reduces I/O operations and improves CPU resource utilization, but also increases the throughput of computation-intensive tasks, thereby enhancing query response efficiency.

Third, runtime filter optimization improves 83% of the 99 evaluated TPC-DS queries, with 29% achieving over 50% speedup. Figure 11c shows the top 10 accelerated queries, which mostly involve joins between large fact tables and dimension tables. Early filtering during large-table scans removes substantial redundant data, cutting unnecessary transfer and processing. The grouped cascading transmission lowers communication overhead, and the statistics-driven adaptive scheme avoids slowdowns in low-selectivity cases, keeping runtime filters efficient for complex workloads.

8.4 CPU Monitoring in Ablation Experiment

To validate the mechanism of the relevant optimizations, we monitor the CPU utilization before and after the introduction of parallel optimization and the vectorized execution engine.

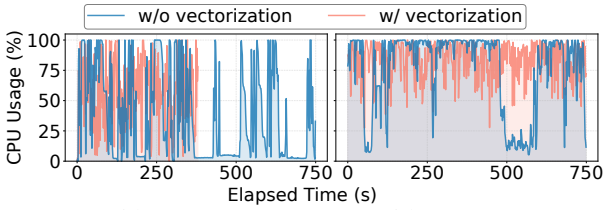
First, we evaluate the impact of parallel optimization by monitoring CPU utilization during TPC-DS benchmark tests. As shown in Figure 12, enabling parallel execution optimizations increases the average CPU utilization by 35% in the single-stream query scenario and by 6% in the multi-stream query scenario, validating the strategy's effectiveness in enhancing computational resource utilization and parallel processing capabilities. Moreover, by monitoring the distribution of CPU utilization across the nodes in the cluster, we find that the resource utilization is consistent across nodes, demonstrating that the strategy achieves good load balancing.



(a) Single stream **(b) Multi stream**

Figure 12: CPU utilization of the parallel optimization

Second, we conduct a detailed analysis of CPU utilization with and without the vectorized execution engine. As shown in Figure 13, enabling the vectorized execution engine consistently maintains high CPU utilization during queries. The batch processing mode significantly improves the utilization of CPU resources by operators. In contrast, without it, TDSQL suffers from low CPU utilization due to heavy I/O, with queries spending substantial time waiting for upstream data. After enabling the vectorized engine, the average CPU utilization on nodes increases by 24% and 12% in single-stream and multi-stream scenarios, respectively, further demonstrating the effectiveness in optimizing compute-intensive tasks.

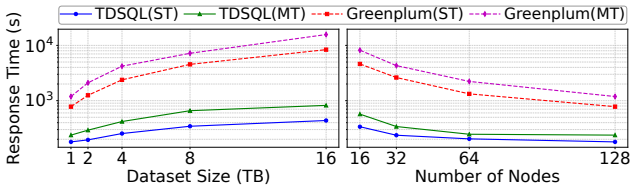


(a) Single stream **(b) Multi stream**

Figure 13: CPU utilization of the vectorized execution

8.5 Scalability Evaluation

Different Data Scales. To evaluate the scalability of TDSQL and the best competitor, Greenplum, under varying data volumes, we conduct experiments on datasets ranging from 1 TB to 16 TB. As shown in Figure 14a, at the 1 TB scale, TDSQL outperforms Greenplum by approximately 4.30× and 4.99× in single-stream and multi-stream response times, respectively. This gap widens with larger datasets, reaching nearly 19× faster query responses in both workloads at the 16 TB scale. As data size increases, Greenplum shows sharply deteriorating query latency. When the dataset grows from 1 TB to 16 TB, the single-stream execution time increases



(a) Different data sizes **(b) Different node sizes**

Figure 14: Query latency under single-stream (ST) and multi-stream (MT) experimental settings

from 778.99s to 8377.55s, and the multi-stream latency grows from 1187.99s to 15700.91s, highlighting its limitations for large-scale analytics. In contrast, TDSQL shows a far smoother latency growth. The single-stream latency increases from 179.38s to 437.19s, and the multi-stream latency grows from 237.88s to 821.82s, showing a moderate increase. This controlled growth stems from optimized parallel execution and vectorized batch processing, which maintain efficient cluster-wide data handling as data scales expand.

Different Machine Scales. To evaluate the scalability under different cluster sizes, we conduct experiments on the TPC-DS benchmark with a 1 TB dataset in environments comprising 16, 32, 64, and 128 nodes. The 1 TB scale is chosen because Greenplum encounters out-of-memory issues when processing a 10 TB dataset on small-scale clusters. As shown in Figure 14b, TDSQL consistently achieves significantly lower latency than Greenplum in all configurations, and its performance steadily improves as the node number increases. This end-to-end scalability result benefits from the Forward Node-based design described in Section 4, where intra-server shared memory and inter-server connection multiplexing help reduce connection-management overhead at larger cluster sizes, thereby contributing to TDSQL's horizontal scalability.

8.6 Case Study of Real Workloads

Currently, we have successfully deployed TDSQL across over 15,000 instances for OLAP scenarios. Among these deployments, migrations from systems such as PostgreSQL and similar systems have achieved an overall performance improvement of more than 50%. Next, we also conduct a systematic evaluation of TDSQL in real-world OLAP scenarios. As shown in Table 4, we select two real-world workload scenarios, PSI and Retail, and conduct tests using representative query tasks in each. The PSI scenario, which is security-focused, analyzes security data generated from billions of mobile devices, with tens of millions of new entries recorded every day. This results in an enormous and constantly growing dataset, where OLAP is used to perform in-depth analysis. The evaluation tasks for PSI cover three core categories: large-scale scans (W2, W5), complex joins (W3, W6), and aggregation-intensive analyses (W4). The Retail scenario, the other tested workload, uses OLAP for omni-channel operational analysis, and its test focus is primarily on evaluating complex join tasks (W1), a key query type for computing sales, inventory, and promotion metrics to support data-driven retail decision-making. The experimental results demonstrate that TDSQL consistently outperforms the anonymous competing system (OP) across all scenarios, exhibiting substantial performance advantages with at least a 40% improvement in each scenario. As TDSQL is increasingly adopted in practical deployments, we plan to further refine TDSQL based on real-world feedback to enhance its efficiency and robustness under diverse workloads.

Table 4: Comparison of TDSQL to opponent (OP)

Database	W1	W2	W3	W4	W5	W6
OP	1292ms	93ms	433ms	545ms	53ms	61ms
TDSQL	423ms	39ms	195ms	287ms	29ms	34ms

8.7 Discussion and Insight

Potential for Performance Improvement. Unlike the primary memory bottleneck observed in TPC-C testing [15], our TPC-DS benchmark reveals a CPU utilization bottleneck during parallel execution of large-scale complex queries, with CPU usage exceeding 90%, while memory remains low at 60%. This indicates that further CPU performance optimizations could unlock substantial performance gains. This issue parallels the challenges in retail and e-commerce analytics, where sufficient memory exists for parallel query processing, but CPU constraints force queries into wait queues. We hypothesize that leveraging more powerful computing hardware, such as upgrading to newer Intel/AMD processors, offloading tasks to GPUs [9, 40, 66, 76, 77], or adopting specialized analytical processors [6] could substantially improve computational capacity and system throughput.

Trade-off Between Query Efficiency and Parallelism. It is often assumed that higher parallelism leads to better query efficiency, but our observations show this is not always the case. Similar to OLTP hotspots [33, 75], when the level of parallelism becomes excessively high, system performance actually experiences an obvious degradation. Currently, we set the maximum parallelism based on empirical experience to achieve near-optimal performance. However, this approach may prove too complex for certain clients. Therefore, in the future, we plan to develop a strategy for real-time adjustment of the maximum parallelism, driven by real-time workload analysis and AI-assisted decision-making based on the current load [10, 42, 60, 86, 87]. This approach aims to mitigate the impact of parallelism settings on query efficiency as much as possible.

9 CONCLUSION

This paper presents the design and optimization of TDSQL, an OLAP database system designed for complex analytical scenarios. TDSQL employs a series of optimization techniques to enhance system scalability and significantly improve the execution efficiency of complex queries on large-scale datasets. Experimental results demonstrate that TDSQL achieved an outstanding performance of 72.6 million QphDS in the TPC-DS benchmark, significantly surpassing the second- and third-ranked systems in the publicly available TPC-DS results, highlighting its advantages in data ingestion, processing, and query execution.

ACKNOWLEDGMENTS

This work is supported by the National Natural Science Foundation of China (No. 62322213, U25B2018, 62461146205), BAAI, and Beijing Science and Technology Project (Z251100008125032).

REFERENCES

[1] Daniel Abadi, Peter Boncz, Stavros Harizopoulos, Stratos Idreos, Samuel Madden, et al. 2013. The design and implementation of modern column-oriented database systems. *Foundations and Trends® in Databases* 5, 3 (2013), 197–280.

[2] Michael O Akinde, Michael H Böhlen, Theodore Johnson, Laks VS Lakshmanan, and Divesh Srivastava. 2003. Efficient OLAP query processing in distributed data warehouses. *Information Systems* 28, 1-2 (2003), 111–135.

[3] Karolina Alexiou, Donald Kossmann, and Per-Åke Larson. 2013. Adaptive range filters for cold data: Avoiding trips to siberia. *Proceedings of the VLDB Endowment* 6, 14 (2013), 1714–1725.

[4] Michael Armbrust, Reynold S Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K Bradley, Xiangrui Meng, Tomer Kaftan, Michael J Franklin, Ali Ghodsi, et al. 2015. Spark sql: Relational data processing in spark. In *Proceedings of the 2015 ACM SIGMOD international conference on management of data*. 1383–1394.

[5] Nikos Armenatzoglou, Sanuj Basu, Naga Bhanoori, Mengchu Cai, Naresh Chainani, Kiran Chinta, Venkatraman Govindaraju, Todd J Green, Monish Gupta, Sebastian Hillig, et al. 2022. Amazon Redshift re-invented. In *Proceedings of the 2022 International Conference on Management of Data*. 2205–2217.

[6] Cagri Balkesen, Nitin Kunal, Georgios Giannikis, Pit Fender, Seema Sundara, Felix Schmidt, Jarod Wen, Sandeep R. Agrawal, Arun Raghavan, Venkatanathan Varadarajan, Anand Viswanathan, Balakrishnan Chandrasekaran, Sam Idicula, Nipun Agarwal, and Eric Sedlar. 2018. RAPID: In-Memory Analytical Query Processing Engine with Extreme Performance per Watt. In *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*, Gautam Das, Christopher M. Jermaine, and Philip A. Bernstein (Eds.). ACM, 1407–1419. https://doi.org/10.1145/3183713.3190655

[7] Alexander Behm, Shoumik Palkar, Utkarsh Agarwal, Timothy Armstrong, David Cashman, Ankur Dave, Todd Greenstein, Shant Hovsepian, Ryan Johnson, Arvind Sai Krishnan, et al. 2022. Photon: A fast query engine for lakehouse systems. In *Proceedings of the 2022 International Conference on Management of Data*. 2326–2339.

[8] Peter A Boncz, Marcin Zukowski, and Niels Nes. 2005. MonetDB/X100: Hyper-Pipelining Query Execution.. In *Cidr*, Vol. 5. 225–237.

[9] Jiashen Cao, Rathijit Sen, Matteo Interlandi, Joy Arulraj, and Hyesoon Kim. 2023. Gpu database systems characterization and optimization. *Proceedings of the VLDB Endowment* 17, 3 (2023), 441–454.

[10] Cheng Chen, Wenlong Ma, Congli Gao, Wenliang Zhang, Kai Zeng, Tao Ye, Yueguo Chen, and Xiaoyong Du. 2025. GaussDB-AISQL: a composable cloud-native SQL system with AI capabilities. *Frontiers of Computer Science* 19, 9 (2025), 199608.

[11] Guanduo Chen, Zhenying He, Meng Li, and Siqiang Luo. 2024. Oasis: An optimal disjoint segmented learned range filter. *Proceedings of the VLDB Endowment* 17, 8 (2024), 1911–1924.

[12] Jianjun Chen, Li Zhang, Yu Xie, Wei Ding, Lixun Cao, Ye Liu, Yonghua Ding, Fangshi Li, Ke Wu, Haibo Xiu, et al. 2025. Vedb-htap: a highly integrated, efficient and adaptive htap system. *Proceedings of the VLDB Endowment* 18, 12 (2025), 4896–4909.

[13] WenGuang Chen, JiDong Zhai, Jin Zhang, and WeiMin Zheng. 2009. LogGPO: An accurate communication model for performance prediction of MPI programs. *Science in China Series F: Information Sciences* 52, 10 (2009), 1785–1791.

[14] Yuxing Chen, Jie Jiang, Xiaolong He, Ziyi Lu, Shuo Han, Anqun Pan, Chong Zhu, Chang Liu, Xuan Zhao, Yu Li, Yan Tang, Dongzhi Zhao, Wen Zhang, Hailin Lei, Lixiong Zheng, Feng Zhang, Mingde Zhang, Yiteng Chu, Wei Lu, Yunpeng Chai, and Xiaoyong Du. 2026. TDSQL-Boundless: A Distributed Database System for Large-scale Heterogeneous Multi-Table Workloads. In *Companion of the International Conference on Management of Data* (India) *(SIGMOD Companion '26)*. Association for Computing Machinery, New York, NY, USA, 252–265. https://doi.org/10.1145/3788853.3803090

[15] 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. *Proceedings of the VLDB Endowment* 17, 12 (2024), 3869–3882.

[16] Dawei Cheng, Yao Zou, Sheng Xiang, and Changjun Jiang. 2025. Graph neural networks for financial fraud detection: a review. *Frontiers of Computer Science* 19, 9 (2025), 199609.

[17] Tencent Cloud. 2025. TBase. https://github.com/Tencent/TBase. Accessed: 2025-09-27.

[18] Tencent Cloud. 2025. Tencent Cloud official site. https://www.tencentcloud.com. Accessed: 2025-09-29.

[19] Tencent Cloud. 2025. TencentDB for PostgreSQL. https://www.tencentcloud.com/products/postgres?from_qcintl=topnav. Accessed: 2025-09-29.

[20] Yann Collet. 2025. XXHash: Extremely fast hash algorithm. https://xxhash.com/. Accessed: 2025-09-30.

[21] Benoit Dageville, Thierry Cruanes, Marcin Zukowski, Vadim Antonov, Artin Avanes, Jon Bock, Jonathan Claybaugh, Daniel Engovatov, Martin Hentschel, Jiansheng Huang, et al. 2016. The snowflake elastic data warehouse. In *Proceedings of the 2016 International Conference on Management of Data*. 215–226.

[22] Zhenghao Ding, Xinyi Zhang, Chao Zhang, Yishen Sun, Kai Xu, Wei Lu, and Xiaoyong Du. 2026. Breaking the Isolation-Freshness Trade-off: Joint Adaptive Storage Optimization for HTAP Systems. *Proceedings of the VLDB Endowment* 19, 6 (2026), 1142–1155.

[23] Doris. 2025. Apache Doris. https://doris.apache.org/. Accessed: 2025-09-27.

[24] E-MapReduce. 2025. E-MapReduce. https://www.alibabacloud.com/en/product/emapreduce?_p_lc=1. Accessed: 2025-09-27.

[25] Navid Eslami, Ioana O Bercea, and Niv Dayan. 2025. Diva: Dynamic Range Filter for Var-Length Keys and Queries. *Proceedings of the VLDB Endowment* 18, 11 (2025), 3923–3936.

[26] Franz Färber, Norman May, Wolfgang Lehner, Philipp Große, Ingo Müller, Hannes Rauhe, and Jonathan Dees. 2012. The SAP HANA Database – An Architecture Overview. *IEEE Data Eng. Bull.* 35, 1 (2012), 28–33. http://sites.computer.org/debull/A12mar/hana.pdf

[27] Ningxuan Feng, Feng Zhang, Jiazao Lin, Jidong Zhai, and Xiaoyong Du. 2019. Statistical analysis and prediction of parking behavior. In *IFIP international conference on network and parallel computing*. Springer, 93–104.

[28] Xiaoyue Feng, Dashan Wei, Chaopeng Guo, and Jie Song. 2025. L2SM: a query-optimized linked LSM-tree for HTAP workloads. *Frontiers of Computer Science* 19, 7 (2025), 197606.

[29] Goetz Graefe. 1989. An Extensible and Parallel Query Evaluation System. (1989).

[30] Goetz Graefe. 1990. Encapsulation of parallelism in the volcano query processing system. *ACM SIGMOD Record* 19, 2 (1990), 102–111.

[31] Goetz Graefe, Ross Bunker, and Shaun Cooper. 1998. Hash joins and hash teams in Microsoft SQL Server. In *VLDB*, Vol. 98. 86–97.

[32] Anurag Gupta, Deepak Agarwal, Derek Tan, Jakub Kulesza, Rahul Pathak, Stefano Stefani, and Vidhya Srinivasan. 2015. Amazon redshift and the case for simpler data warehouses. In *Proceedings of the 2015 ACM SIGMOD international conference on management of data*. 1917–1923.

[33] Yinhao Hong, Hongyao Zhao, Wei Lu, Xiaoyong Du, Yuxing Chen, Anqun Pan, and Lixiong Zheng. 2025. A Hybrid Approach to Integrating Deterministic and Non-deterministic Concurrency Control in Database Systems. *Proc. VLDB Endow.* 18, 5 (2025), 1376–1389.

[34] Jean-François Im, Kishore Gopalakrishna, Subbu Subramaniam, Mayank Shrivastava, Adwait Tumbde, Xiaotian Jiang, Jennifer Dai, Seunghyun Lee, Neha Pawar, Jialiang Li, et al. 2018. Pinot: Realtime olap for 530 million users. In *Proceedings of the 2018 International Conference on Management of Data*. 583–594.

[35] Google Inc. 2025. BigQuery Materialized Views. https://cloud.google.com/bigquery/docs/materialized-views-intro. Accessed: 2025-11-15.

[36] Marcel Kornacker, Alexander Behm, Victor Bittorf, Taras Bobrovytsky, Casey Ching, Alan Choi, Justin Erickson, Martin Grund, Daniel Hecht, Matthew Jacobs, et al. 2015. Impala: A Modern, Open-Source SQL Engine for Hadoop.. In *Cidr*, Vol. 1. Asilomar, CA, 9.

[37] Kylin. 2025. Apache Kylin. https://kylin.apache.org/. Accessed: 2025-09-27.

[38] Andrew Lamb, Matt Fuller, Ramakrishna Varadarajan, Nga Tran, Ben Vandier, Lyric Doshi, and Chuck Bear. 2012. The vertica analytic database: C-store 7 years later. *arXiv preprint arXiv:1208.4173* (2012).

[39] Per-Åke Larson, Adrian Birka, Eric N Hanson, Weiyun Huang, Michal Nowakiewicz, and Vassilis Papadimos. 2015. Real-time analytical processing with SQL server. *Proceedings of the VLDB Endowment* 8, 12 (2015), 1740–1751.

[40] Rubao Lee, Minghong Zhou, Chi Li, Shenggang Hu, Jianping Teng, Dongyang Li, and Xiaodong Zhang. 2021. The art of balance: a RateupDB™ experience of building a CPU/GPU hybrid database product. *Proceedings of the VLDB Endowment* 14, 12 (2021), 2999–3013.

[41] Jiajun Li, Runlin Lei, and Zhewei Wei. 2025. A survey of estimating number of distinct values. *Frontiers of Computer Science* 19, 9 (2025), 199611.

[42] Pengju Liu, Pan Cai, Kai Zhong, Cuiping Li, and Hong Chen. 2025. LRP: learned robust data partitioning for efficient processing of large dynamic queries. *Frontiers of Computer Science* 19, 9 (2025), 199607.

[43] 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.

[44] Lailong Luo, Deke Guo, Richard TB Ma, Ori Rottenstreich, and Xueshan Luo. 2018. Optimizing bloom filter: Challenges, solutions, and comparisons. *IEEE Communications Surveys & Tutorials* 21, 2 (2018), 1912–1949.

[45] Siqiang Luo, Subarna Chatterjee, Rafael Ketsetsidis, Niv Dayan, Wilson Qin, and Stratos Idreos. 2020. Rosetta: A robust space-time optimized range filter for key-value stores. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 2071–2086.

[46] Zhenghua Lyu, Huan Hubert Zhang, Gang Xiong, Gang Guo, Haozhou Wang, Jinbao Chen, Asim Praveen, Yu Yang, Xiaoming Gao, Alexandra Wang, et al. 2021. Greenplum: a hybrid database for transactional and analytical workloads. In *Proceedings of the 2021 International Conference on Management of Data*. 2530–2542.

[47] Roger MacNicol and Blaine French. 2004. Sybase IQ multiplex-designed for analytics. In *Proceedings of the Thirtieth international conference on Very large data bases-Volume 30*. 1227–1230.

[48] Norman May, Alexander Böhm, Daniel Ritter, Frank Renkes, Mihnea Andrei, and Wolfgang Lehner. 2025. SAP HANA Cloud: Data Management for Modern

Enterprise Applications. In *Companion of the 2025 International Conference on Management of Data, SIGMOD/PODS 2025, Berlin, Germany, June 22-27, 2025*, Volker Markl, Joseph M. Hellerstein, and Azza Abouzied (Eds.). ACM, 580–592. https://doi.org/10.1145/3722212.3724452

[49] Sergey Melnik, Andrey Gubarev, Jing Jing Long, Geoffrey Romer, Shiva Shivakumar, Matt Tolton, Theo Vassilakis, Hossein Ahmadi, Dan Delorey, Slava Min, et al. 2020. Dremel: A decade of interactive SQL analysis at web scale. *Proceedings of the VLDB Endowment* 13, 12 (2020), 3461–3472.

[50] Puya Memarzia, Huaxin Zhang, Kelvin Ho, Ronen Grosman, and Jiang Wang. 2024. GaussDB-Global: A Geographically Distributed Database System. In *2024 IEEE 40th International Conference on Data Engineering (ICDE)*. IEEE, 5111–5118.

[51] Gabriel Mersy, Zhuo Wang, Stavros Sintos, and Sanjay Krishnan. 2024. Optimizing Collections of Bloom Filters within a Space Budget. *Proceedings of the VLDB Endowment* 17, 11 (2024), 3551–3564.

[52] Microsoft. 2025. Query Processing Architecture Guide – SQL Server. https://learn.microsoft.com/en-us/sql/relational-databases/query-processing-architecture-guide.

[53] Bernhard Mößner, Christian Riegger, Arthur Bernhardt, and Ilia Petrov. 2022. bloomRF: On performing range-queries in Bloom-Filters with piecewise-monotone hash functions and prefix hashing. *arXiv preprint arXiv:2207.04789* (2022).

[54] Shuangcheng Niu, Jidong Zhai, Xiaosong Ma, Xiongchao Tang, Wenguang Chen, and Weimin Zheng. 2015. Building semi-elastic virtual clusters for cost-effective HPC cloud resource provisioning. *IEEE Transactions on Parallel and Distributed Systems* 27, 7 (2015), 1915–1928.

[55] Zaifeng Pan, Feng Zhang, Yanliang Zhou, Jidong Zhai, Xipeng Shen, Onur Mutlu, and Xiaoyong Du. 2021. Exploring data analytics without decompression on embedded GPU systems. *IEEE Transactions on Parallel and Distributed Systems* 33, 7 (2021), 1553–1568.

[56] Pedro Pedreira, Orri Erling, Masha Basmanova, Kevin Wilfong, Laith Sakka, Krishna Pai, Wei He, and Biswapesh Chattopadhyay. 2022. Velox: meta's unified execution engine. *Proceedings of the VLDB Endowment* 15, 12 (2022), 3372–3384.

[57] Orestis Polychroniou, Arun Raghavan, and Kenneth A Ross. 2015. Rethinking SIMD vectorization for in-memory databases. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. 1493–1508.

[58] Orestis Polychroniou and Kenneth A Ross. 2019. Towards practical vectorized analytical query engines. In *Proceedings of the 15th International Workshop on Data Management on New Hardware*. 1–7.

[59] Shi Qiao, Adrian Nicoara, Jin Sun, Marc Friedman, Hiren Patel, and Jaliya Ekanayake. 2019. Hyper dimension shuffle: Efficient data repartition at petabyte scale in scope. *Proceedings of the VLDB Endowment* 12, 10 (2019), 1113–1125.

[60] Shao-Jie Qiao, Han-Lin Fan, Nan Han, Lan Du, Yu-Han Peng, Rong-Min Tang, and Xiao Qin. 2025. Learning database optimization techniques: the state-of-the-art and prospects. *Frontiers of Computer Science* 19, 12 (2025), 1912612.

[61] Mark Raasveldt and Hannes Mühleisen. 2019. Duckdb: an embeddable analytical database. In *Proceedings of the 2019 international conference on management of data*. 1981–1984.

[62] Daniel Ritter, Mihnea Andrei, Sukhyeun Cho, Maik Goergens, Taehyung Lee, Norman May, Amit Pathak, and Paul R. Willems. 2025. The HANA Native Query Engine for Lakehouse Systems. *Proc. VLDB Endow.* 18, 12 (2025), 4831–4845. https://www.vldb.org/pvldb/vol18/p4831-ritter.pdf

[63] Robert Schulze, Tom Schreiber, Ilya Yatsishin, Ryadh Dahimene, and Alexey Milovidov. 2024. Clickhouse-lightning fast analytics for everyone. *Proceedings of the VLDB Endowment* 17, 12 (2024), 3731–3744.

[64] Raghav Sethi, Martin Traverso, Dain Sundstrom, David Phillips, Wenlei Xie, Yutian Sun, Nezih Yegitbasi, Haozhun Jin, Eric Hwang, Nileema Shingte, et al. 2019. Presto: SQL on everything. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*. IEEE, 1802–1813.

[65] A Shaikhha, M Dashti, and C Koch. 2018. Push versus pull-based loop fusion in query engines. *Journal of Functional Programming* 28 (2018).

[66] Anil Shanbhag, Samuel Madden, and Xiangyao Yu. 2020. A Study of the Fundamental Performance Characteristics of GPUs and CPUs for Database Analytics. In *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*, David Maier, Rachel Pottinger, AnHai Doan, Wang-Chiew Tan, Abdussalam Alawini, and Hung Q. Ngo (Eds.). ACM, 1617–1632. https://doi.org/10.1145/3318464.3380595

[67] Juliusz Sompolski, Marcin Zukowski, and Peter Boncz. 2011. Vectorization vs. compilation in query execution. In *Proceedings of the Seventh International Workshop on Data Management on New Hardware*. 33–40.

[68] Yutian Sun, Tim Meehan, Rebecca Schlussel, Wenlei Xie, Masha Basmanova, Orri Erling, Andrii Rosa, Shixuan Fan, Rongrong Zhong, Arun Thirupathi, et al. 2023. Presto: A decade of SQL analytics at Meta. *Proceedings of the ACM on Management of Data* 1, 2 (2023), 1–25.

[69] Sasu Tarkoma, Christian Esteve Rothenberg, and Eemil Lagerspetz. 2011. Theory and practice of bloom filters for distributed systems. *IEEE Communications Surveys & Tutorials* 14, 1 (2011), 131–155.

[70] TeleDB. 2025. TeleDB Database. https://www.ctyun.cn/products/topic/teledb. Accessed: 2025-09-27.

[71] Teradata. 2024. Teradata Database. https://www.teradata.com/resources/datasheets/teradata-database. Accessed: 2025-09-27.

[72] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Suresh Anthony, Hao Liu, Pete Wyckoff, and Raghotham Murthy. 2009. Hive: a warehousing solution over a map-reduce framework. *Proceedings of the VLDB Endowment* 2, 2 (2009), 1626–1629.

[73] TPC. 2025. TPC-DS: On-Line Transaction Processing Benchmark. https://www.tpc.org/tpcds/default5.asp. Accessed: 2025-09-27.

[74] TPC. 2025. TPC-DS: On-Line Transaction Processing Benchmark. https://www.tpc.org/tpcds/results/tpcds_results5.asp?version=3. Accessed: 2025-09-27.

[75] 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 *SIGMOD Conference Companion*. ACM, 675–688.

[76] Dalin Wang, Feng Zhang, Weitao Wan, Hourun Li, and Xiaoyong Du. 2021. FineQuery: Fine-Grained Query Processing on CPU-GPU Integrated Architectures. In *CLUSTER*. IEEE, 355–365.

[77] Bowen Wu, Dimitrios Koutsoukos, and Gustavo Alonso. 2025. Efficiently Processing Joins and Grouped Aggregations on GPUs. *Proc. ACM Manag. Data* 3, 1 (2025), 39:1–39:27. https://doi.org/10.1145/3709689

[78] Rongbiao Xie, Meng Li, Zheyu Miao, Rong Gu, He Huang, Haipeng Dai, and Guihai Chen. 2021. Hash adaptive bloom filter. In *2021 IEEE 37th International Conference on Data Engineering (ICDE)*. IEEE, 636–647.

[79] Fangjin Yang, Eric Tschetter, Xavier Léauté, Nelson Ray, Gian Merlino, and Deep Ganguli. 2014. Druid: A real-time analytical data store. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*. 157–168.

[80] Matei Zaharia, Ali Ghodsi, Reynold Xin, and Michael Armbrust. 2021. Lakehouse: A New Generation of Open Platforms that Unify Data Warehousing and Advanced Analytics. In *11th Conference on Innovative Data Systems Research, CIDR 2021, Virtual Event, January 11-15, 2021, Online Proceedings*. www.cidrdb.org. http://cidrdb.org/cidr2021/papers/cidr2021_paper17.pdf

[81] Chaoqun Zhan, Maomeng Su, Chuangxian Wei, Xiaoqiang Peng, Liang Lin, Sheng Wang, Zhe Chen, Feifei Li, Yue Pan, Fang Zheng, et al. 2019. AnalyticDB: real-time OLAP database system at Alibaba cloud. *Proceedings of the VLDB Endowment* 12, 12 (2019), 2059–2070.

[82] Chao Zhang, Guoliang Li, Jintao Zhang, Xinning Zhang, and Jianhua Feng. 2024. HTAP databases: A survey. *IEEE Transactions on Knowledge and Data Engineering* 36, 11 (2024), 6410–6429.

[83] Fangyuan Zhang, Caihua Yin, Hua Fan, Fenghua Fang, Yineng Chen, Xuqi Wang, Mengqi Wu, Bing Chen, Tianbo Jin, Sibo Wang, Wenchao Zhou, and Feifei Li. 2025. AnalyticDB-PG: A Cloud-native High-performance Data Warehouse in Alibaba Cloud. *Proc. VLDB Endow.* 18, 12 (2025), 5139–5152. https://www.vldb.org/pvldb/vol18/p5139-zhou.pdf

[84] Feng Zhang, Jidong Zhai, Xipeng Shen, Dalin Wang, Zheng Chen, Onur Mutlu, Wenguang Chen, and Xiaoyong Du. 2021. TADOC: Text analytics directly on compression. *The VLDB Journal* 30, 2 (2021), 163–188.

[85] Huanchen Zhang, Hyeontaek Lim, Viktor Leis, David G Andersen, Kimberly Keeton, and Andrew Pavlo. 2019. Succinct range filters. *ACM SIGMOD Record* 48, 1 (2019), 78–85.

[86] Xukang Zhang, Huanchen Zhang, and Xiaofeng Meng. 2025. Intra-Query Elasticity for Cloud-Native Data Analysis. *Proc. ACM Manag. Data* 3, 3 (2025), 178:1–178:28. https://doi.org/10.1145/3725315

[87] Qiushi 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.

[88] Tianqi Zheng, Zhibin Zhang, and Xueqi Cheng. 2020. Saha: a string adaptive hash table for analytical databases. *Applied Sciences* 10, 6 (2020), 1915.