TDSQL Boundless

Use Cases

Product Documentation

Contents

Use Cases
Technical Evolution and Usage Practices of Online DDL

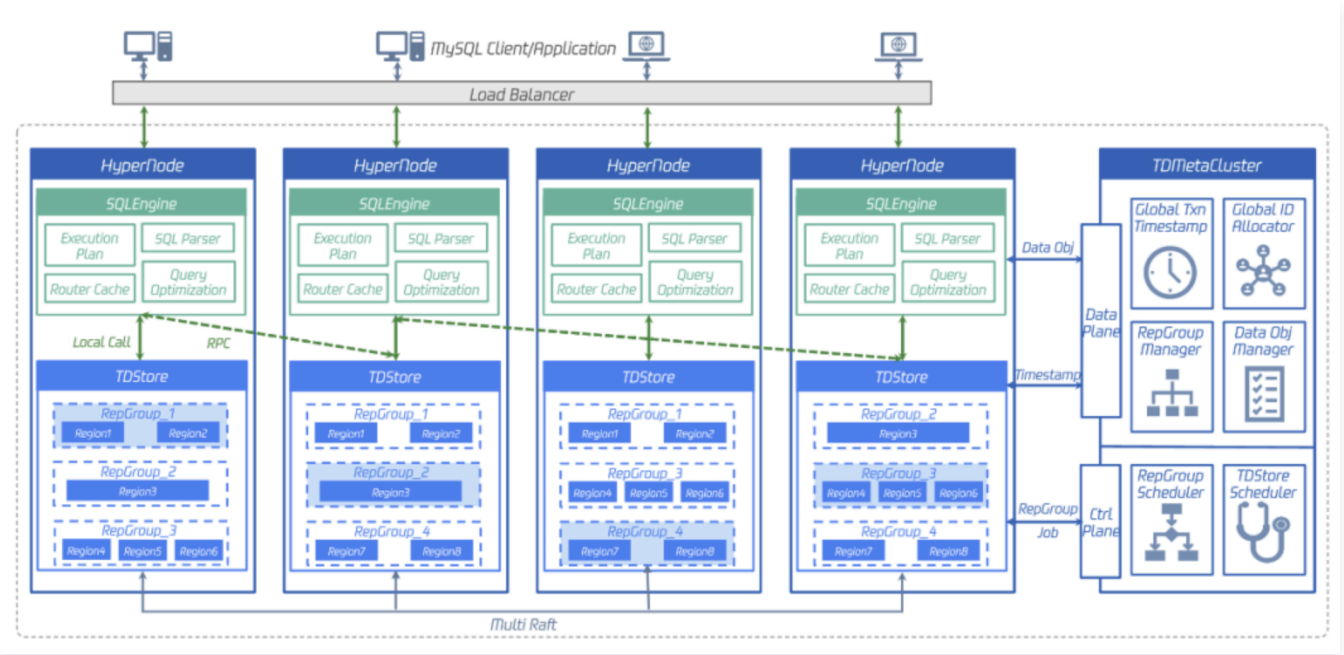
Last updated：2026-03-06 18:50:08

Introduction

In the history of database development, DDL (Data Definition Language) operations have always posed significant challenges. In early versions of mainstream commercial databases, traditional DDL operations often required placing an exclusive lock on data tables to prevent data inconsistencies during execution. This locking mechanism caused databases to become unavailable during DDL operations, which is unacceptable for large-scale modern applications requiring 7*24 operation.
To address this issue, ORACLE introduced Online Table Redefinition starting from 9i, and MySQL introduced Online DDL from version 5.6, both aiming to significantly reduce (or eliminate) application downtime required for modifying database objects (this feature is collectively referred to as Online DDL). Subsequently, MySQL 8.0 further introduced the Instant algorithm, marking a major breakthrough in DDL operations. Instant DDL allows certain operations—such as adding columns at the end, modifying column names, renaming tables, and so on—to be executed immediately without locking tables or copying data. This greatly accelerates DDL operations and minimizes business impact. Nevertheless, DBAs may still need to rely on external tools like pt-online-schema-change (pt-osc) in certain scenarios, as Online DDL does not support all types of DDL operations, and pt-osc offers additional flexibility through features like flow control.
With the rise of distributed databases, Online DDL presents new challenges. In distributed environments, DDL operations must be executed simultaneously across multiple nodes, requiring not only guaranteed safety for concurrent DDL & DML operations but also considerations for performance, execution efficiency, and crash-safe mechanisms.
This article shares the technical evolution and practical implementation of Online DDL in the latest product of the TencentDB TDSQL series: TDSQL Boundless.

Introduction to TDSQL Boundless Architecture

TDSQL Boundless is a high-performance, high-availability enterprise-grade distributed database solution developed by Tencent for financial-grade application scenarios. It adopts a containerized cloud-native architecture, delivering high-performance computing capabilities and cost-effective massive storage at the cluster level.
The figure below shows the architecture of TDSQL Boundless, where the entire instance is divided into two types of nodes:



- **The TDMetaCluster on the right serves as the management node**: responsible for management of cluster metadata and intelligent scheduling.
- **The HyperNode on the left is a peer node, also known as a hybrid node**: Each node comprises two components—the computing layer (SQLEngine) and the storage layer (TDStore). The SQLEngine layer is responsible for SQL parsing, query plan generation, query optimization, and so on, and sends transaction-related read/write requests to the underlying TDStore.

TDSQL Boundless architecture and features: fully distributed + storage-compute integration/storage-compute separation + data plane/control plane separation + high scalability + global consistency + high compression ratio.

Tackling the Challenges of TDSQL Boundless Online DDL

Current approach to executing DDL in traditional standalone MySQL:

1. instant DDL (ALGORITHM=INSTANT): Only modifies metadata in the data dictionary without copying data or rebuilding the table, leaving the original table data unaffected. Such DDL statements can be executed directly and completed immediately.
2. INPLACE DDL (ALGORITHM=INPLACE): Rebuilds the table "in-place" at the storage engine layer without blocking DML operations. However, for large table modifications, this may lead to prolonged primary-replica inconsistency. For such DDL statements, if greater control over the DDL process is required and replica lag is a concern, it is recommended to use third-party online schema change tools like pt-osc.
3. COPY DDL (ALGORITHM=COPY): Operations such as "modifying column types" or "changing table character sets" only support table copy, which blocks DML operations and does not qualify as Online DDL. For such DDL statements, it is recommended to use third-party online schema change tools like pt-osc.

Overall, beyond INSTANT DDL, traditional standalone MySQL primarily relies on third-party online schema change tools to execute DDL operations. However, this approach presents several notable drawbacks:

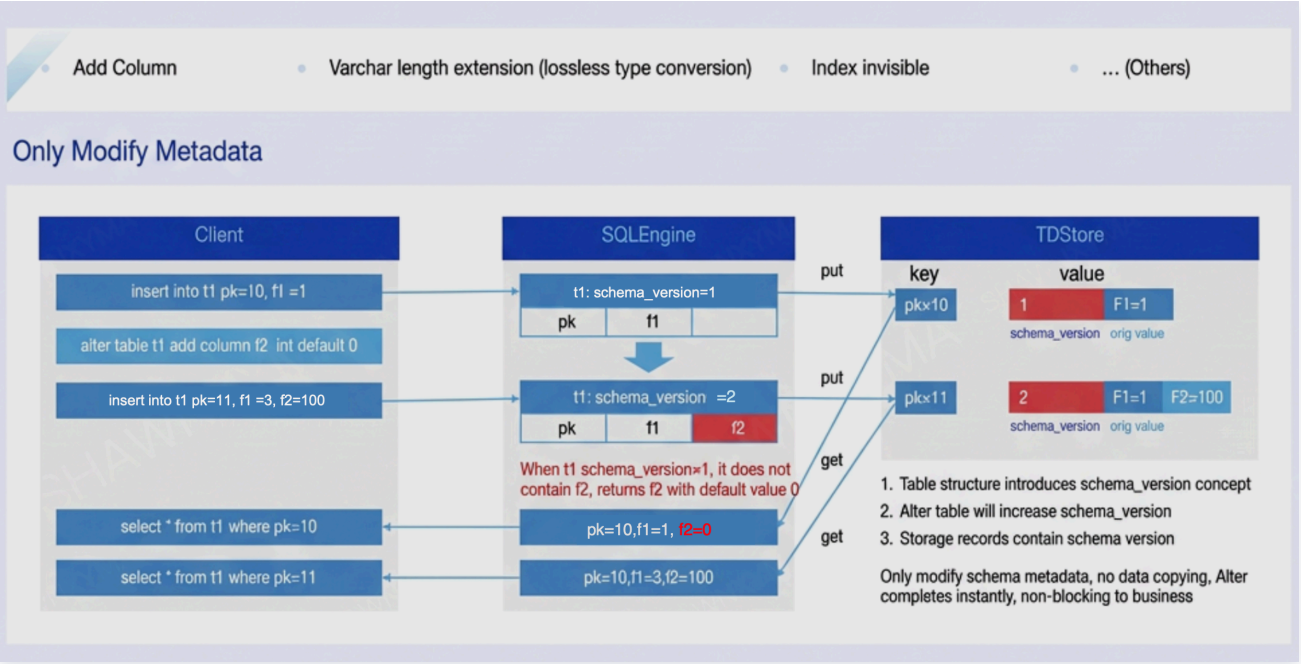
1. DDL may fail to acquire the lock due to large transactions or long-running queries, resulting in persistent waits and repeated failures.
2. All third-party tools require rebuilding the entire table, significantly sacrificing performance to ensure stability. Tests show that compared to native DDL execution modes (INSTANT/INPLACE/COPY), third-party tools exhibit a 10-fold or even orders-of-magnitude performance degradation in DDL execution. This is unacceptable given the current rapidly growing data volumes.

Compared to traditional standalone MySQL, executing DDL in distributed databases faces more and complex challenges:

1. Whether the native INSTANT DDL's ultra-fast execution feature is compatible and retained.
2. How to solve the concurrency control issue between DDL requiring data backfill and DML while maintaining execution efficiency?
3. In native distributed databases with a storage-compute separation architecture, compute nodes are loosely coupled (stateless). When DDL changes are executed on one compute node, how do other compute nodes within the same instance promptly detect these DDL modifications? Simultaneously, data read/write accuracy must be guaranteed throughout the DDL change process.
4. Whether in native distributed databases, where data is distributed across multiple nodes—with both node scale and storage capacity significantly larger than standalone MySQL—bypassing the storage layer directly + combining multi-machine parallelism can significantly accelerate DDL operations requiring table reconstruction.
5. Distributed Database DDL crash-safe issue.

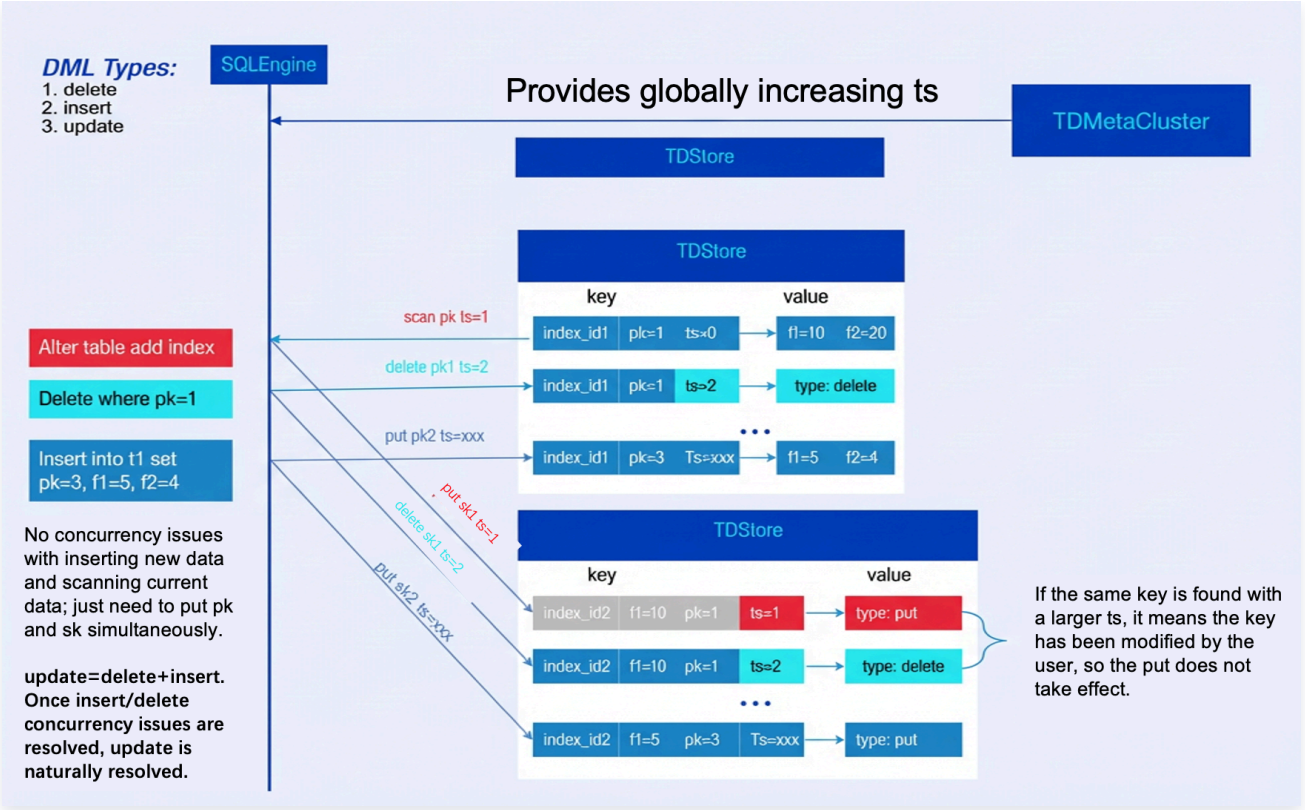
Below we will explore how TDSQL Boundless employs a series of innovative strategies to overcome various challenges faced by Online DDL.

1. By introducing multi-version schemas to solve instant DDL issues:



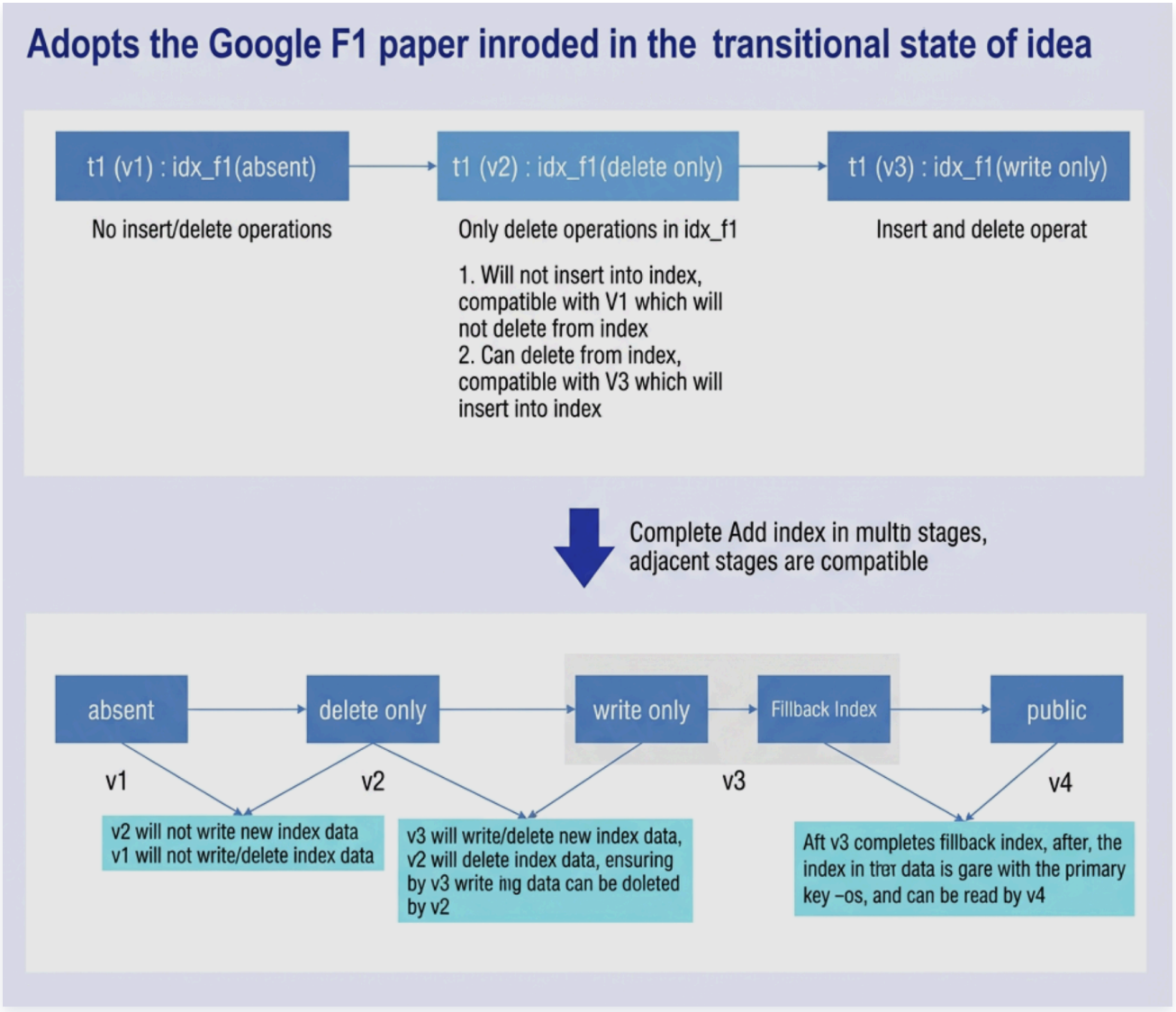
By introducing the concept of a version number (schema version) in the table structure. As shown in the figure above, table t1 initially has version 1; after a column is added, the version number becomes 2. New data rows carry the version number upon insertion. During data reading, the row version must be checked first: if the row version is 2, the current table structure is used for parsing; if the row version is 1 (lower than the current version), after it is determined that column F2 is absent in this version's schema, the default value is populated directly before returning to the client. This multi-version schema parsing mechanism enables non-destructive type changes—such as adding columns or extending varchar length—to complete instantly by modifying metadata only.

2. By introducing (Thomas Write Rule) to resolve concurrency control issues between DDL requiring data backfill and DML:



By introducing (Thomas Write Rule) to provide a concurrency control mechanism for DDL & DML operations, ensuring serialization order of database protocols. Compared to standard timestamp–based concurrency control methods, (Thomas Write Rule) ignores obsolete writes, reducing the likelihood of transactions being aborted.

3. Referencing Google F1's schema change methodology, transitional states are introduced to ensure smooth schema transitions during the change process:



Google F1's methodology for schema changes references Online, Asynchronous Schema Change in F1. The fundamental concept abstracted from this: when an action cannot immediately transition from one state (before adding an index) to another (after adding an index), transitional intermediate states can be introduced to ensure compatibility between adjacent states. Although entities cannot instantly enter a specific state, as long as they remain within two compatible states and maintain compatibility while progressively advancing through state transitions, they can gradually reach the final state and complete the entire transition.

The figure illustrates a typical add index operation in DDL. During the state transition of all SQLEngines from v1 –> v4, two transitional states (v2 and v3) are introduced. It can be observed that:

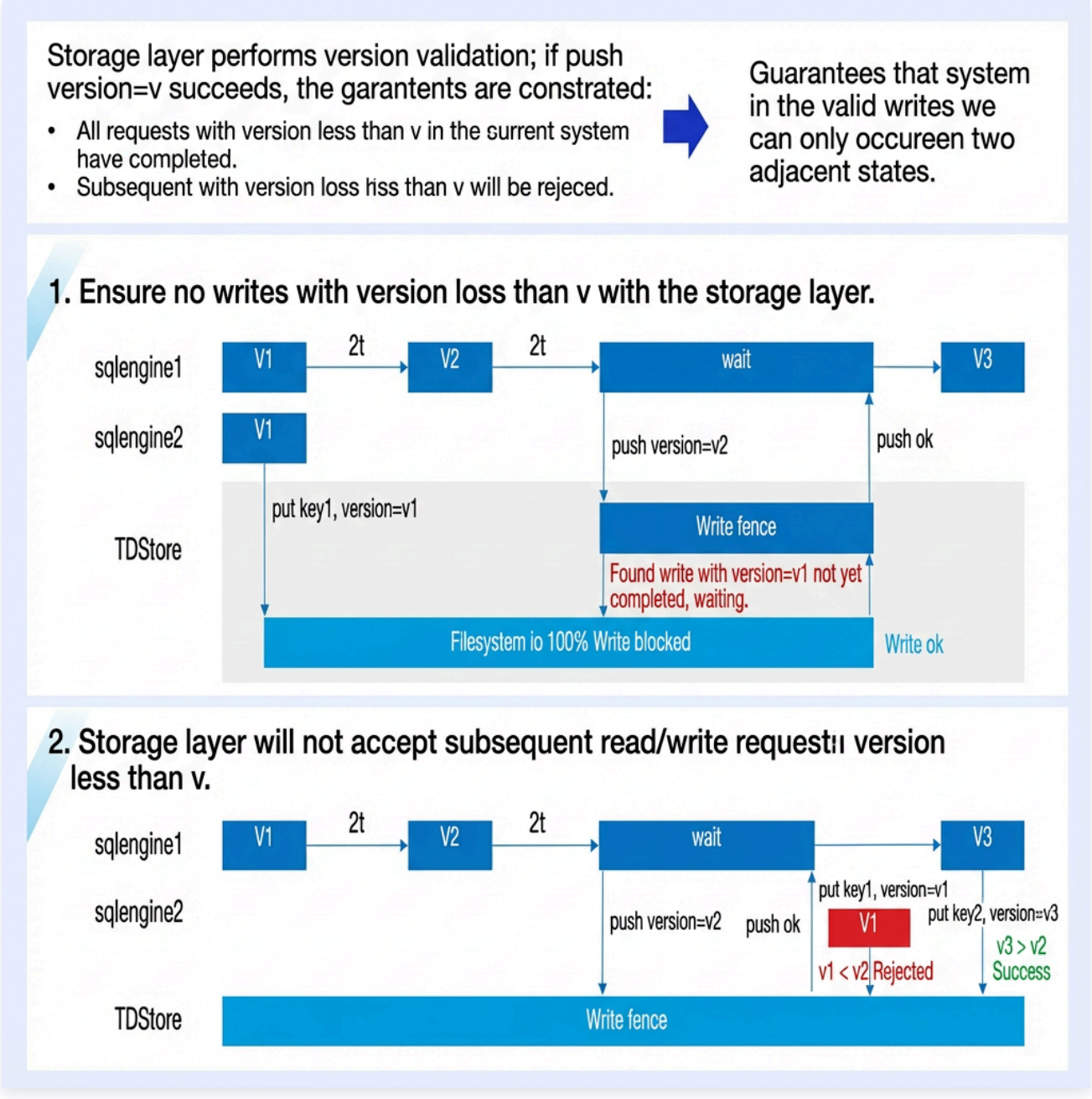
3.1 **Coexistence of v1 and v2**: v1 represents the old schema, while v2 is the new schema (delete-only). At this stage, all SQLEngines are either in the v1 or v2 state. Since the index is only manipulated during delete operations and no index has been created yet, no data inconsistency issues arise.

3.2 **Coexistence of v2 and v3**: v2 represents the new schema (delete-only), while v3 is the new schema (write-only). At this stage, all SQLEngines are either in the v2 or v3 state. SQLEngines in the v2 state can normally insert data, delete data, and manage indexes (both existing and new data lack indexes); SQLEngines in the v3 state can normally insert, delete data, and manage indexes. However,

since indexes remain invisible to users in both v2 and v3 states, and users only see data, data consistency is maintained from the user's perspective.

3.3 **Coexistence of v3 and v4**: v3 represents the new schema (write-only), while v4 is the new schema (index addition completed). At this stage, all SQLEngines are either in the v3 or v4 state. SQLEngines in the v3 state can normally insert and delete data and indexes (existing data still lacks indexes); SQLEngines in the v4 state have populated indexes for existing data, representing the final state after add index. It can be observed that in the v3 state, users see complete data; in the v4 state, both complete data and indexes are visible to users, thus maintaining data consistency.

4. By incorporating the concept of transitional states, a Write Fence mechanism is designed: Through version verification of requests at the storage layer, it ensures that successful writes from multiple SQLEngines can only occur between two adjacent states at any given moment, preventing data inconsistency.



Write Fence is an internal data structure of TDSQL Boundless, storing the mapping of schema_obj_id → schema_obj_version. It addresses the collaboration challenge between the SQLEngine at the computation layer and TDStore at the storage layer. We need to ensure TDStore is aware of relevant changes before the SQLEngine status is advanced.

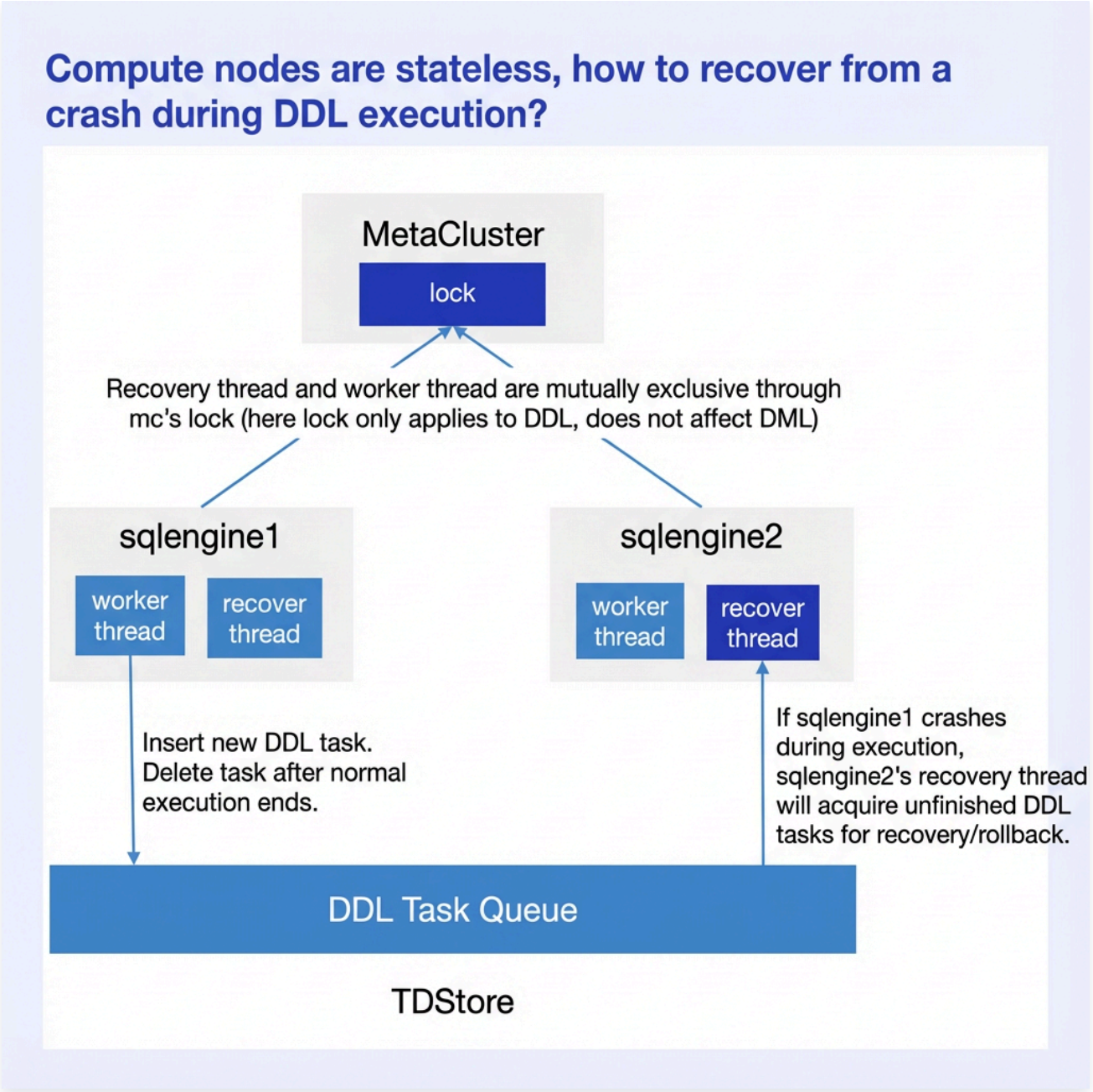
Before each SQLEngine enters the next state, it must push the current version to TDStore (push write fence). A successful push ensures two constraints:

- In the current system, all requests with versions lower than the pushed version have been completed.
- Subsequent requests with versions lower than the pushed version will be rejected.

Through the version verification mechanism at the storage layer, it ensures that successful writes from multiple SQLEngines can only occur between two adjacent states at any given moment, preventing data inconsistency.

The figure also demonstrates that even in extreme exception scenarios, if a node attempts to send a request with a version lower than v2 after successfully pushing the v2 version, the storage layer TDStore will detect that the request version is lower than the current v2 in the Write Fence and thus rejects the request. This ensures the correctness of the operation of the overall algorithm.

5. crash-safe Issue of DDL:



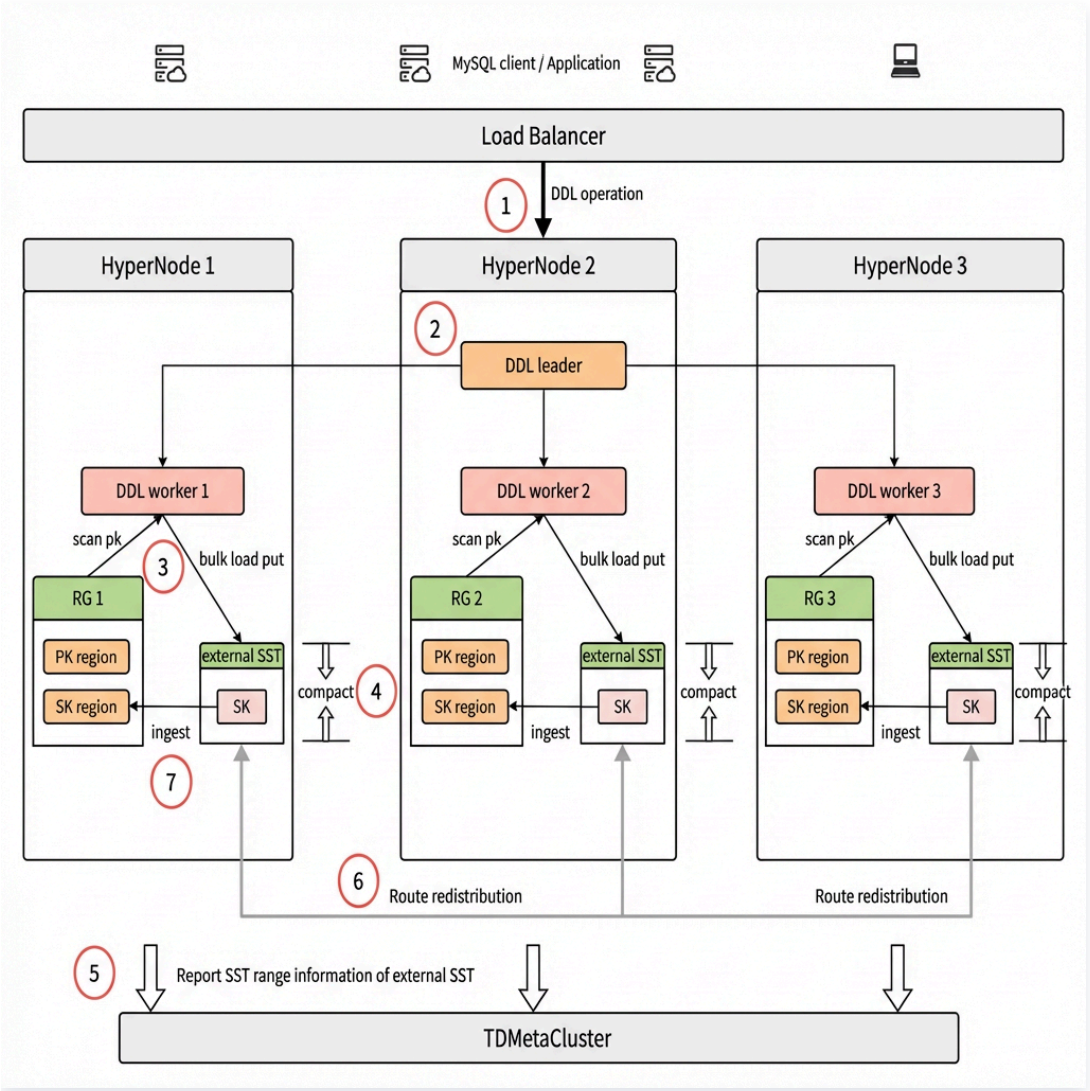
A DDL job queue + DDL recovery thread are introduced to ensure crash-safe for DDL. Note that the recovery thread operates independently from worker threads and does not wait for communication from each other. If execution fails due to network issues, the recovery thread takes over (the recovery thread may not reside on the same node as the original worker thread). Monitoring the DDL job queue is straightforward: query the `INFORMATION_SCHEMA.DDL_JOBS` system table and verify the final execution status via the `DDL_STATUS` field.

TDSQL Boundless currently supports Online DDL capabilities in most scenarios. For details, see OnlineDDL Description.

TDSQL Boundless Fast Online-DDL New Feature

From the above introduction, we can see that if data backfilling occurs during Online DDL in TDSQL Boundless, it adopts the (Thomas Write Rule) for batch backfilling and accelerates the process through

parallel execution. However, in practical usage, especially in scenarios with large data volumes, the execution time remains relatively long because Thomas Write requires comparing backfilled data with concurrent DML data using timestamps to determine which version to retain. Therefore, the latest version of TDSQL Boundless introduces the Fast Online-DDL feature: It uses a bulk load approach to organize backfilled data into external sst files and directly ingests them into the Lmax layer of TDStore. This eliminates the overhead of timestamp comparison and concurrency control, significantly improving execution efficiency.



The diagram above illustrates the architecture of Fast Online-DDL. Let's examine how it works:

1. The user initiates a DDL operation request, which is randomly routed through the load balancer to a peer node in the TDSQL Boundless instance. Here, it is assumed to be routed to HyperNode 2.
2. The HyperNode 2 node executes the DDL operation as the DDL leader, assigning workers to all relevant HyperNode peer nodes based on the PK data distribution of the target object.
3. Each worker scans local PKs to generate index data and writes it into external sst files using the bulk load method.
4. After all workers complete scanning the PKs, they perform a compaction operation (compact) on the external sst files to ensure no duplicate keys exist within sst files at the same level.

5. The compacted sst range information is reported to the TDMetaCluster (mc) management node. Based on this information, mc divides these ranges into regions (a logical concept referring to small data segments) and assigns them to corresponding RGs (Replication Group, a logical concept where one RG contains multiple regions and serves as the smallest multi-replica data synchronization unit), generating new routing information. Simultaneously, mc temporarily disables the split and merge of routing for regions involved in SK.
6. The HyperNode reorganizes the external sst files based on the new routing information and transfers them to the corresponding HyperNode nodes.
7. The external sst files that meet the new routing criteria are directly ingested into the Lmax layer of the user RG's sst files. Upon completion, mc removes the restrictions on split and merge operations for region routing involved in SK.

TDSQL Boundless Fast Online-DDL Practice

The following steps will create a large partitioned table and use the DDL statement `add index` to test the performance improvement of Fast Online-DDL.

Hardware Environment

Node Type	Node Specifications	Number of Nodes
HyperNode	16-Core CPU/32 GB of Memory/Enhanced SSD Cloud Disk 300 GB	3

Data Preparation

```
-- Create a partitioned table where the number of partitions is a
multiple of the nodes:
CREATE TABLE `lineitem` (
  `L_ORDERKEY` int NOT NULL,
  `L_PARTKEY` int NOT NULL,
  `L_SUPPKEY` int NOT NULL,
  `L_LINENUMBER` int NOT NULL,
  `L_QUANTITY` decimal(15,2) NOT NULL,
  `L_EXTENDEDPRICE` decimal(15,2) NOT NULL,
  `L_DISCOUNT` decimal(15,2) NOT NULL,
  `L_TAX` decimal(15,2) NOT NULL,
  `L_RETURNFLAG` char(1) NOT NULL,
  `L_LINESTATUS` char(1) NOT NULL,
  `L_SHIPDATE` date NOT NULL,
```

```
  `L_COMMITDATE` date NOT NULL,
  `L_RECEIPTDATE` date NOT NULL,
  `L_SHIPINSTRUCT` char(25) NOT NULL,
  `L_SHIPMODE` char(10) NOT NULL,
  `L_COMMENT` varchar(44) NOT NULL,
  PRIMARY KEY (`L_ORDERKEY`,`L_LINENUMBER`)
) ENGINE=ROCKSDB DEFAULT CHARSET=utf8mb3
PARTITION BY HASH (`L_ORDERKEY`) PARTITIONS 24;

-- Generate test data using the official TPC standard tool: TPC-H
v3.0.1. Download URL: tpc.org/tpch/.

-- Import, replacing the data file directory for LOAD DATA with your own
directory:
#!/bin/bash
for tbl in lineitem
do
  for i in {1..30}
  do
    echo "Importing table: $tbl"
    mysql $opts -e "set tdsql_bulk_load_allow_unsorted=1;set
tdsql_bulk_load = 1;LOAD DATA INFILE '/data/TPCH_test/dbgen/tpch-
100g/${tbl}.tbl.$i' INTO TABLE $tbl FIELDS TERMINATED BY '|';" &
  done
done
wait
date

-- Verify the number of records:
sql> select count(*) from tpchpart100g.lineitem;
+-----------+
| count(*)  |
+-----------+
| 600037902 |

-- Expected to be roughly evenly distributed across each node:
select sum(region_stats_approximate_size) as size, count(b.rep_group_id)
as region_nums, sql_addr, c.leader_node_name, b.rep_group_id from
information_schema.META_CLUSTER_DATA_OBJECTS a join
information_schema.META_CLUSTER_REGIONS b join
```

```
information_schema.META_CLUSTER_RGS c join
information_schema.META_CLUSTER_NODES d    on a.data_obj_id =
b.data_obj_id and b.rep_group_id = c.rep_group_id and c.leader_node_name
= d.node_name where a.table_name = 'lineitem' and a.data_obj_type =
'PARTITION_L1' group by rep_group_id order by leader_node_name;
+------------+-------------+-----------------+----------------------+--
------------+
| size       | region_nums | sql_addr        | leader_node_name     |
rep_group_id |
+------------+-------------+-----------------+----------------------+--
------------+
| 8879148586 |          76 | 9.30.0.133:15070 | node-three-001       |
64535 |
| 8878971995 |          81 | 9.30.2.175:15088 | node-three-002       |
513 |
| 8878089427 |          79 | 9.30.0.134:15070 | node-three-003       |
65082 |
+------------+-------------+-----------------+----------------------+--
------------+

-- If data is unevenly distributed, perform splitting and leader
switching.
ALTER INSTANCE SPLIT RG 64535 BY 'size';
ALTER INSTANCE TRANSFER LEADER RG 64535 TO 'node-three-003';
```

Fast Online DDL: Before and After It Is Enabled

Related parameters:

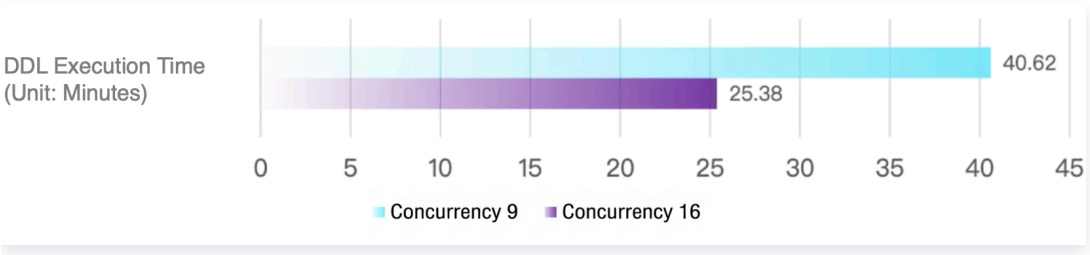
```
-- Number of worker threads for DDL operations (total across all nodes),
default value: 8
max_parallel_ddl_degree

-- Data backfill mode for DDL operations. Default value: 'ThomasWrite'.
To enable the Fast Online DDL feature, it must be set to 'IngestBehind'.
tdsql_ddl_fillback_mode
```

Using the default 'ThomasWrite' data backfill mode [in this mode, DDL threads are executed on a single machine], enable 9 and 16 threads respectively [without exceeding the number of CPUs per node] to test add

index:

```
-- ThomasWrite backfill mode:
set session max_parallel_ddl_degree=9;
set session tdsql_ddl_fillback_mode='ThomasWrite';
alter table tpchpart100g.lineitem add index
index_idx_q_part_key(l_partkey);
Query OK, 0 rows affected (40 min 37.62 sec)

set session max_parallel_ddl_degree=16;
set session tdsql_ddl_fillback_mode='ThomasWrite';
alter table tpchpart100g.lineitem add index
index_idx_w_part_key(l_partkey);
Query OK, 0 rows affected (25 min 22.95 sec)
```

With Fast Online DDL not enabled, execution time comparison:

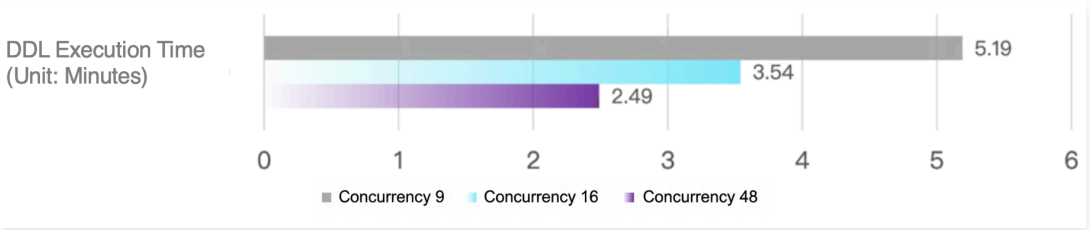


Enable Fast Online DDL using the 'IngestBehind' data backfill mode [in this mode, DDL threads are executed in a distributed manner]. Enable 9, 16, and 48 threads respectively [without exceeding the total number of CPUs across data distribution nodes] to test add index:

```
-- IngestBehind backfill mode:
set session max_parallel_ddl_degree=9;
set session tdsql_ddl_fillback_mode='IngestBehind';
alter table tpchpart100g.lineitem add index
index_idx_j_part_key(l_partkey);
Query OK, 0 rows affected (5 min 11.66 sec)

set session max_parallel_ddl_degree=16;
set session tdsql_ddl_fillback_mode='IngestBehind';
alter table tpchpart100g.lineitem add index
index_idx_k_part_key(l_partkey);
Query OK, 0 rows affected (3 min 32.15 sec)
```

```
set session max_parallel_ddl_degree=48;
set session tdsql_ddl_fillback_mode='IngestBehind';
alter table tpchpart100g.lineitem add index
index_idx_l_part_key(l_partkey);
Query OK, 0 rows affected (2 min 29.52 sec)
```

With Fast Online DDL enabled, execution time comparison:



Query for DDL execution results, DDL_STATUS field displays the final result:

```
--ddl_jobs: dictionary table that records the execution process of DDL
select * from INFORMATION_SCHEMA.DDL_JOBS where
date_format(START_TIMESTAMP,'%Y-%m-%d')='2024-11-22' and IS_HISTORY=1
order by START_TIMESTAMP desc limit 1\G
*************************** 1. row ***************************
             ID: 18
    SCHEMA_NAME: tpch100g
     TABLE_NAME: lineitem
        VERSION: 204
     DDL_STATUS: SUCCESS
START_TIMESTAMP: 2024-11-22 18:47:47
 LAST_TIMESTAMP: 2024-11-22 18:50:18
        DDL_SQL: alter table tpch100g.lineitem add index
index_idx_s_part_key(l_partkey)
      INFO_TYPE: ALTER TABLE
           INFO: {"tmp_tbl":{"db":"tpch100g","table":"#sql-
11746_212c8b_673ef509000030_9"},"alt_type":1,"alt_tid_upd":
{"tid_from":10013,"tid_to":10013},"cr_idx":
[{"id":10240,"ver":34,"stat":0,"tbl_type":2,"idx_type":2},
{"id":10241,"ver":34,"stat":0,"tbl_type":2,"idx_type":4},
{"id":10242,"ver":34,"stat":0,"tbl_type":2,"idx_type":4},
{"id":10243,"ver":34,"stat":0,"tbl_type":2,"idx_type":4},
{"id":10244,"ver":34,"stat":0,"tbl_type":2,"idx_type":4},
```

```
{"id":10245,"ver":34,"stat":0,"tbl_type":2,"idx_type":4},
{"id":10246,"ver":34,"stat":0,"tbl_type":2,"idx_type":4},
{"id":10247,"ver":34,"stat":0,"tbl_type":2,"idx_type":4},
{"id":10248,"ver":34,"stat":0,"tbl_type":2,"idx_type":4},
{"id":10249,"ver":34,"stat":0,"tbl_type":2,"idx_type":4},
{"id":10250,"ver":34,"stat":0,"tbl_type":2,"idx_type":4},
{"id":10251,"ver":34,"stat":0,"tbl_type":2,"idx_type":4},
{"id":10252,"ver":34,"stat":0,"tbl_type":2,"idx_type":4},
{"id":10253,"ver":34,"stat":0,"tbl_type":2,"idx_type":4},
{"id":10254,"ver":34,"stat":0,"tbl_type":2,"idx_type":4},
{"id":10255,"ver":34,"stat":0,"tbl_type":2,"idx_type":4},
{"id":10256,"ver":34,"stat":0,"tbl_type":2,"idx_type":4},
{"id":10257,"ver":34,"stat":0,"tbl_type":2,"idx_type":4},
{"id":10258,"ver":34,"stat":0,"tbl_type":2,"idx_type":4},
{"id":10259,"ver":34,"stat":0,"tbl_type":2,"idx_type":4},
{"id":10260,"ver":34,"stat":0,"tbl_type":2,"idx_type":4},
{"id":10261,"ver":34,"stat":0,"tbl_type":2,"idx_type":4},
{"id":10262,"ver":34,"stat":0,"tbl_type":2,"idx_type":4},
{"id":10263,"ver":34,"stat":0,"tbl_type":2,"idx_type":4},
{"id":10264,"ver":34,"stat":0,"tbl_type":2,"idx_type":4}],"rm_idx":
[],"init":false,"tmp_tab":false,"online_op":true,"wf_rmed":false,"online
_copy_stage":0,"idx_op":true,"row_applied":true,"row_apply_saved":true,"
current_schema_name":"tpch100g","crt_data_obj_task_id":29062709316158283
,"dstr_data_obj_task_id":0,"data_obj_to_be_dstr_arr":
[],"part_policy_ids":[1],"progress":"total: 200, scanned: 200
(100.00%)","fillback_mode":"IngestBehind","exec_addr":
{"ip":"10.0.20.144","port":6008},"recov_addr":
{"ip":"10.0.20.144","port":6008}}
     IS_HISTORY: 1
1 row in set (0.00 sec)
```


Detailed description of the ddl_jobs field:

Field	Description
ID	Each DDL JOB has a unique ID.
SCHEMA_NAME	Database name.
TABLE_NAME	Table name.

VERSION	INFO version number of the field parsing.
DDL_STATUS	Execution status of the DDL JOB, with three possible states: SUCCESS, FAIL, EXECUTING.
START_TIMESTAMP	Initiation time of the DDL JOB.
LAST_TIMESTAMP	End time of the DDL JOB.
DDL_SQL	Details of DDL statements.
INFO_TYPE	Type of DDL statement.
INFO	Metadata information during DDL execution (including execution progress of DDL statements such as add index and copy table).

Best Practices for TDSQL Boundless Online DDL

The Fast Online DDL capability of TDSQL Boundless combines parallel processing with bypass writes, making DDL operations more efficient and convenient.
However, if we fail to properly distinguish between large/small tables or do not partition data appropriately based on scale, the execution efficiency of Fast Online DDL may be significantly reduced. This is because, without proper partitioning, data in a large table is likely to reside on a single node. Consequently, DDL operations will be executed on that single node rather than being parallelized across multiple nodes, greatly diminishing execution efficiency.
Only by reasonably utilizing partitioned tables based on data scale can the distributed scalability of Fast Online DDL be fully leveraged.
Partitioning recommendations:

1. TDSQL Boundless is 100% compatible with native MySQL partitioned table syntax, supporting primary/secondary partitioning. It is primarily designed to address: (1) Capacity issues of large tables; (2) Performance issues under high-concurrency access.
2. Capacity issues of large tables: If a single table is expected to exceed the data disk capacity of a single node in the future, it is recommended to create primary hash or key partitioning to evenly distribute data across multiple nodes. Should data volume continue to grow, elastic scaling can be leveraged to progressively "reduce" disk usage per node.
3. Performance issues under high-concurrency access: For TP (Transaction Processing) services experiencing high-concurrency access, if a single node is expected to be unable to handle the excess read/write load, it is also recommended to create primary hash or key partitioning to evenly distribute read/write pressure across multiple nodes.
4. For partitioned tables created in points 2 and 3, it is recommended to select partition keys based on business characteristics that satisfy most core business queries. The number of partitions should be an

integer multiple of the number of instance nodes.

5. For data cleanup requirements, create RANGE–partitioned tables to perform rapid data cleanup using the truncate partition command. To also distribute data while meeting cleanup needs, further create partitioned tables with HASH subpartitions.

Development of TDSQL Boundless Online DDL

TDSQL Boundless is rapidly evolving, with features iterating and improving based on user needs. DDL parallel performance continues to be optimized. The current new version of TDSQL Boundless specifically enhances data backfilling performance for partitioned tables during add index operations. This prioritization stems from observing that many deployed business systems utilize large partitioned tables spanning terabytes or tens of terabytes, often requiring Online index additions. In the upcoming TDSQL Boundless release, we will fully implement the Fast Online DDL capability, delivering its benefits for partitioned tables during copy table operations, as well as for regular tables during add index and copy table operations.
As the core of Tencent Cloud Database's long–term strategy, TDSQL Boundless will continue to be driven by business needs, focusing on refining the product to deliver more efficient and stable services to users.

Lock Mechanism Analysis and Troubleshooting Practices

Last updated：2026-03-06 18:50:08

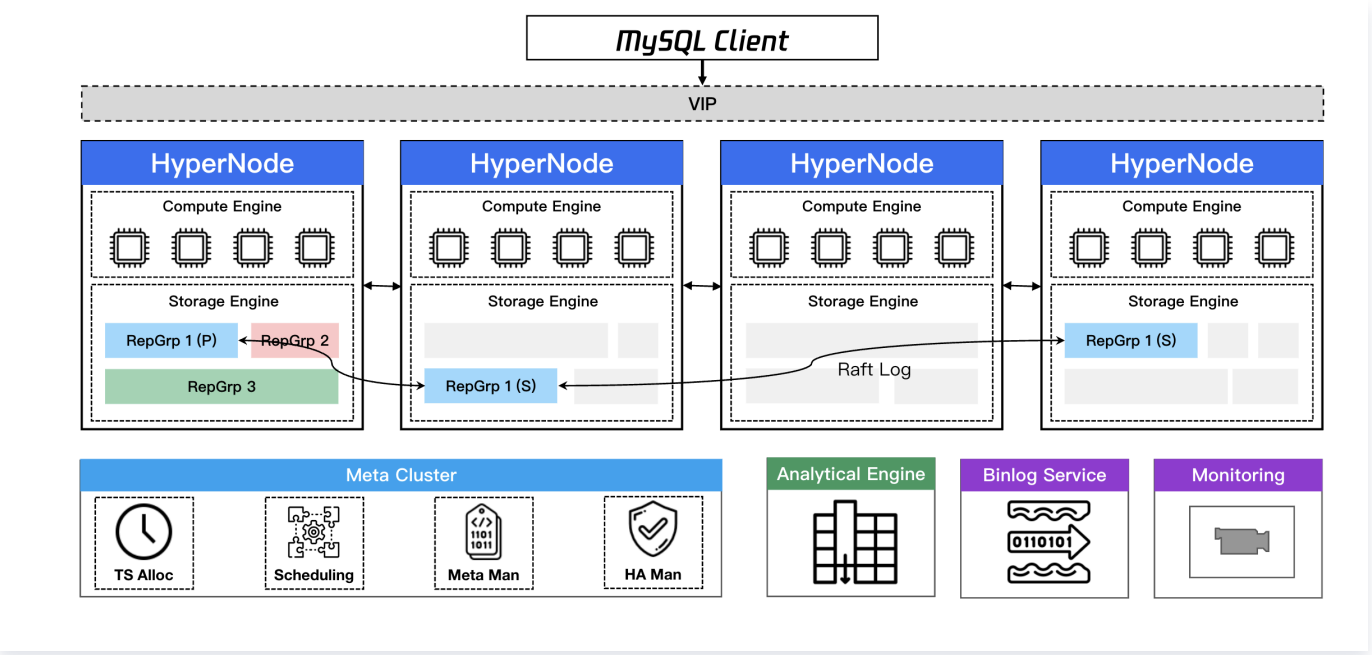
1. Introduction

Lock mechanisms are one of the core mechanisms for implementing concurrent access control in relational databases. Understanding their working principles is a key entry point for troubleshooting access conflicts. For example, common lock conflict errors in high-concurrency or complex transaction processing scenarios, (such as "**Lock wait timeout exceeded**"), essentially reflect that the data resources (such as rows and tables) requested by the current transaction have been locked by other transactions. When the current transaction continuously fails to obtain the lock and reaches the threshold for lock wait timeout, it is actively interrupted.

To resolve lock conflicts, the session holding the lock must release it. The best way to have the session release the lock is to identify the initiator of the long-term lock-holding session and contact the user to complete the transaction (commit or rollback). In emergency situations, the DBA may terminate the session holding the lock.

This article introduces **TencentDB TDSQL's latest product—TDSQL Boundless**. Using typical case studies, it analyzes the core role of lock mechanisms in transaction concurrency, revealing how they balance data consistency assurance with system throughput enhancement.

2. TDSQL Boundless Architecture Introduction



TDSQL Boundless is a high-performance, highly available enterprise-grade distributed database solution developed by Tencent for financial-grade application scenarios. It adopts a containerized cloud-native

architecture, providing high-performance computing capabilities and cost-effective massive storage at the cluster level.

TDSQL Boundless architecture and features: fully distributed + storage-compute integration/storage-compute separation + data plane/control plane separation + high scalability + global consistency + high compression ratio.

3. Lock in TDSQL Boundless

TDSQL Boundless, as a typical distributed system, requires not only mechanisms for concurrent access control within individual nodes but also ensures (Mutual Exclusion) across multiple nodes to prevent data inconsistencies caused by concurrent modifications of the same resource by multiple nodes.

Core Lock Types and Implementation Levels

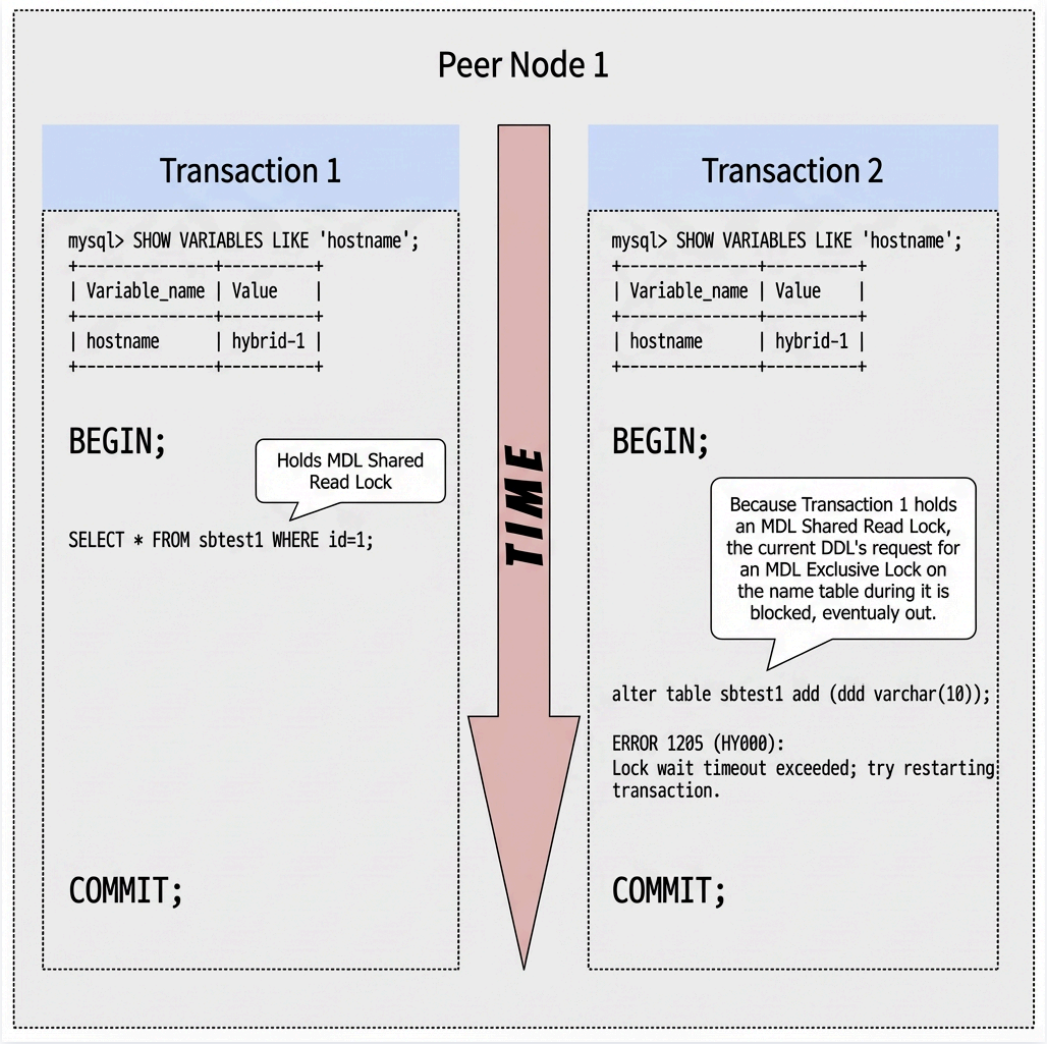
1. **Table-level locks (computing layer)** are coarse-grained locks that adopt MySQL's native Metadata Lock (MDL) to resolve concurrency conflicts between DDL/DML operations within a single node.
2. **Row-level locks & range locks (storage layer)** are fine-grained locks that prevent multiple sessions from concurrently modifying the same record, enabling precise concurrency control.
3. **Global object locks (applied by the computing layer and persisted in the TDMC layer)** are table-level locks. While table-level locks in the computing layer block concurrent DDL operations within a single node, global object locks coordinate DDL operations across multiple nodes.

Table-level locks act on the computing layer, and their conflict scenarios can be categorized into two types: intra-node and inter-node. The following sections will analyze the specific conflict situations in both scenarios respectively.

Table-Level Lock Conflict

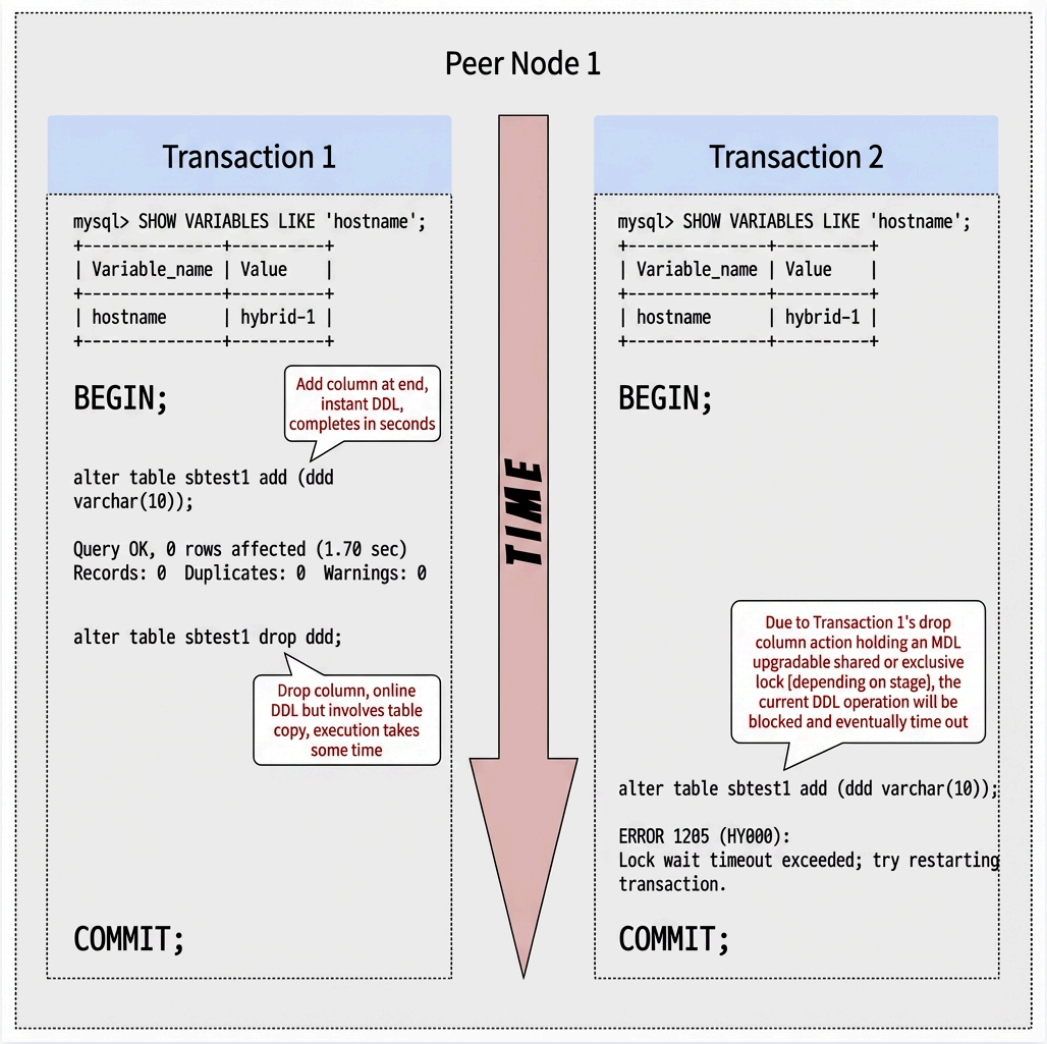
1. **Intra-node DDL-DML Conflict**

   In the following example, both Transaction 1 and Transaction 2 are connected to the same peer node hybrid-1. Transaction 1 explicitly starts a transaction to query table sbtest1 and holds an MDL shared read lock [table-level lock] on this table before the transaction ends. When Transaction 1 is still ongoing, Transaction 2 executing a DDL statement will be blocked, thereby protecting the metadata of table sbtest1.



2. **Intra-node DDL-DDL Conflict**

   Similarly, it is still assumed that both Transaction 1 and Transaction 2 are connected to the same peer node hybrid-1. Transaction 1 is performing a DDL operation on table sbtest1, holding the MDL shared lock and exclusive lock [table-level lock] on this table in phases, preventing the subsequent DDL operation of Transaction 2 from compromising the metadata of table sbtest1.

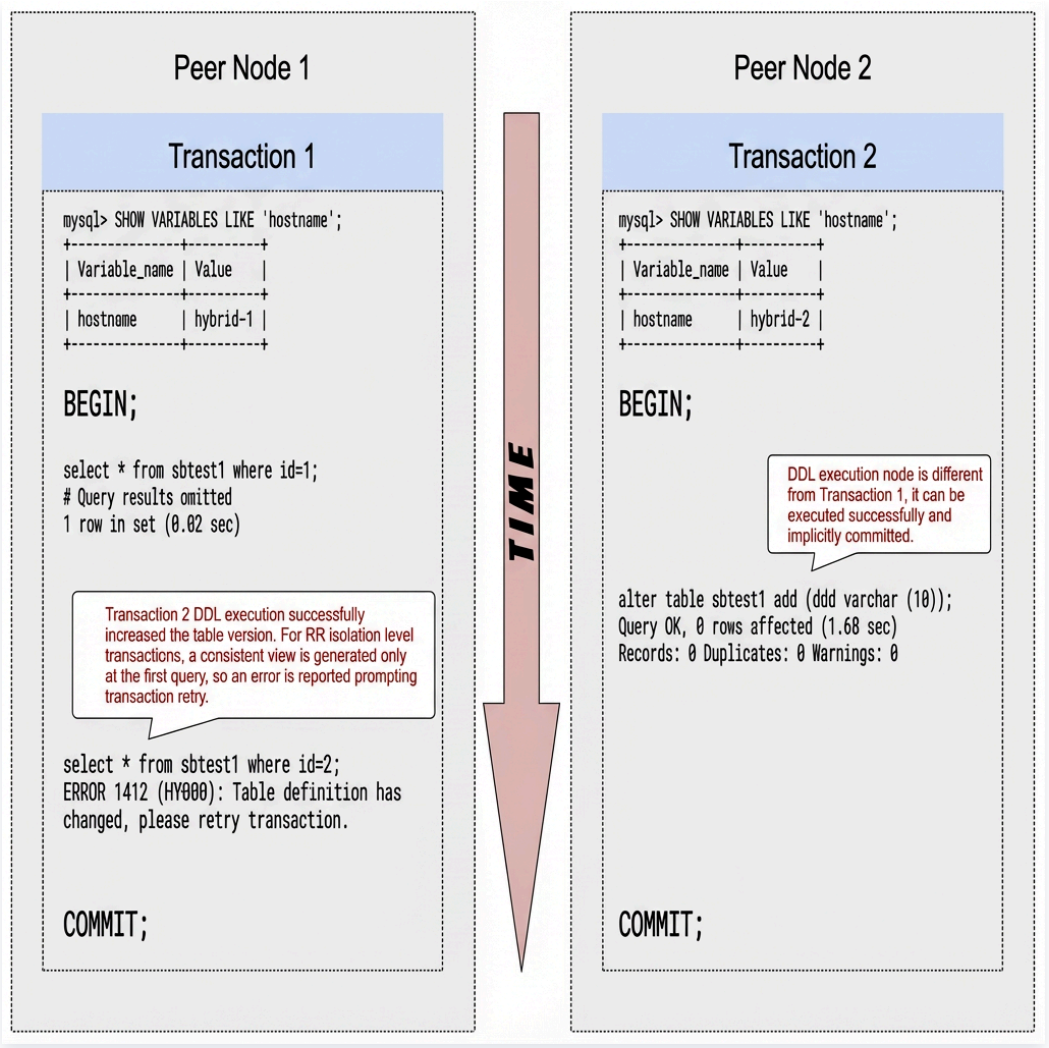


On the same peer node, DDL and DML operations are treated equally—whichever acquires the MDL lock first executes, with no inherent priority, as the MDL lock exists as an in-memory state within the process. However, TDSQL Boundless is a distributed database where sessions are distributed across all nodes. What would happen if sessions connected to two different peer nodes perform operations on the same table?

3. **Inter-node DDL-DML Conflict**

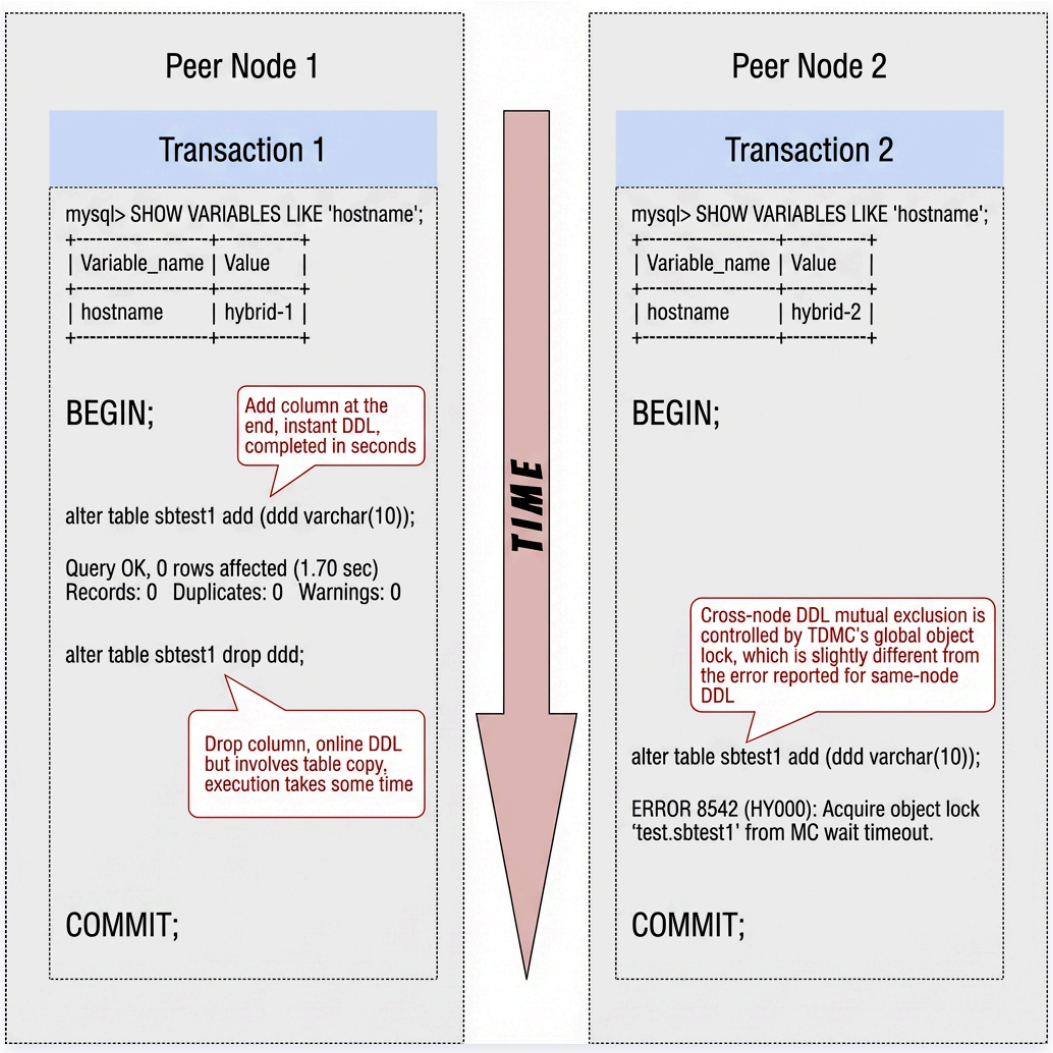
   Let's look at the following example: Transaction 1 and Transaction 2 are connected to peer nodes hybrid-1 and hybrid-2 respectively.

   Transaction 1 explicitly starts a transaction to query table sbtest1. Subsequently, Transaction 2 successfully executes a DDL statement on another node hybrid-2 (since no other session accesses table sbtest1 on this node, no lock conflict occurs), pushing up the schema version of table sbtest1. Subsequently, Transaction 1 continues to execute the query on table sbtest1 and reports an error. For the Repeatable Read isolation level, a Consistent Read View is generated only during the first query of a transaction. Returning records under a new version of the table structure would violate this principle, resulting in ERROR 1412 and prompting transaction retry (when catching this Exception, the business program should perform rollback and retry the transaction).



4. **Inter-node DDL-DDL Conflict**

   Cross-node DDL operations rely on the global object lock mechanism in the TDMC layer to achieve mutual exclusion, ensuring data consistency and operational orderliness in a distributed environment. The following example demonstrates mutual blocking of DDL operations on the same table. The difference from previous cases is that these operations are executed on different nodes. Here, mutual exclusion is not ensured by node-level MDL locks but by the global object lock in TDMC.

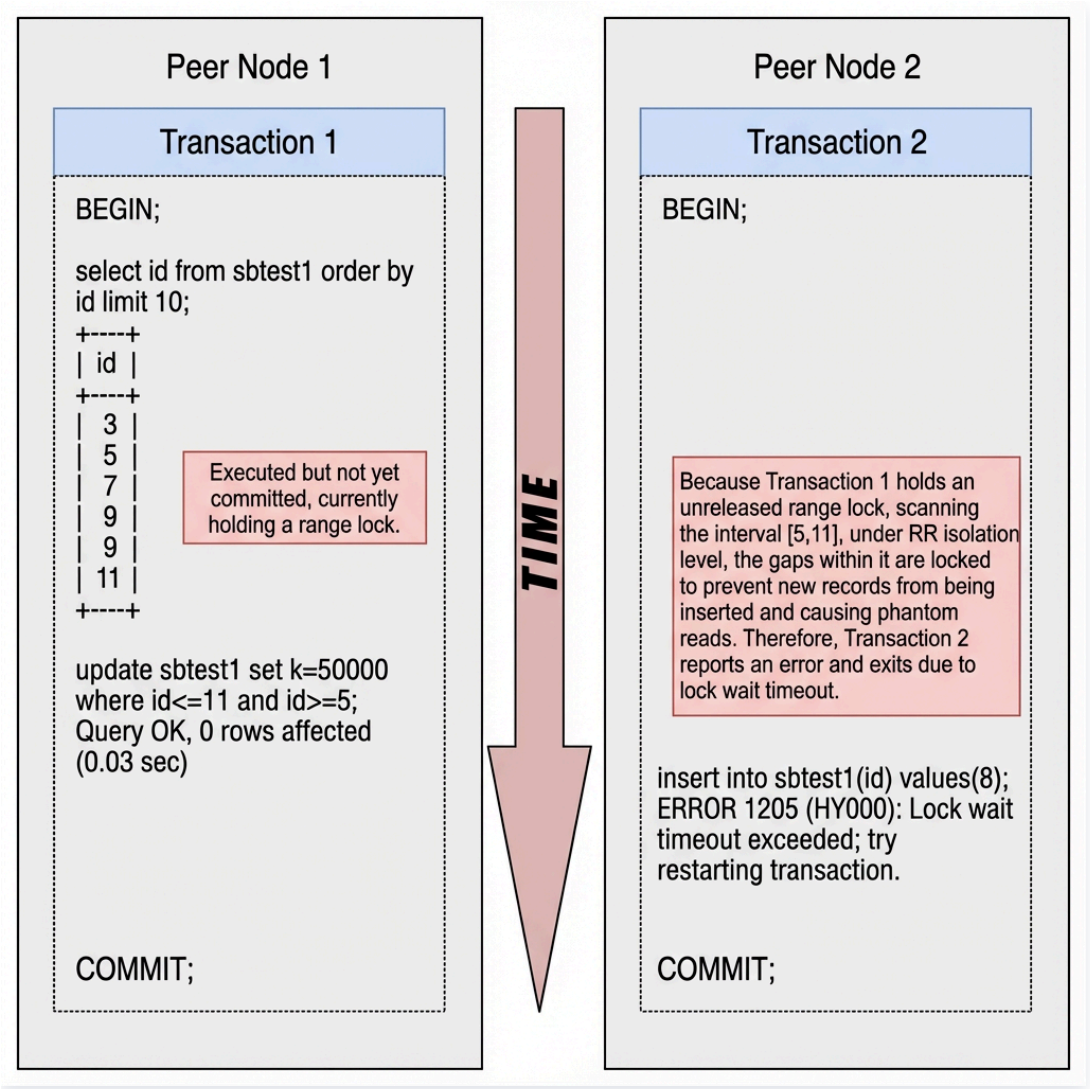


Row-level locks & range locks act on the storage layer. When two sessions encounter row lock or range conflicts, it indicates that both ultimately accessed the primary replica of the same storage node. Thus, regardless of whether the sessions are connected to the same SQLEngine node, the outcome remains identical. The following analysis will examine specific conflict scenarios involving range locks, row-level locks, and the particularly distinct case of row-level deadlocks.

Range Lock Conflict

When operations are performed on a range of a table, the TDSQL Boundless storage layer applies a range lock (range lock). The lock covers a left-closed, right-open key interval, behaving similarly to InnoDB's next-key lock under the Repeatable Read isolation level.

The following example demonstrates that under the Repeatable Read isolation level, updating the id range [5, 11] will lock the gaps in that range to prevent phantom reads caused by new record insertions.

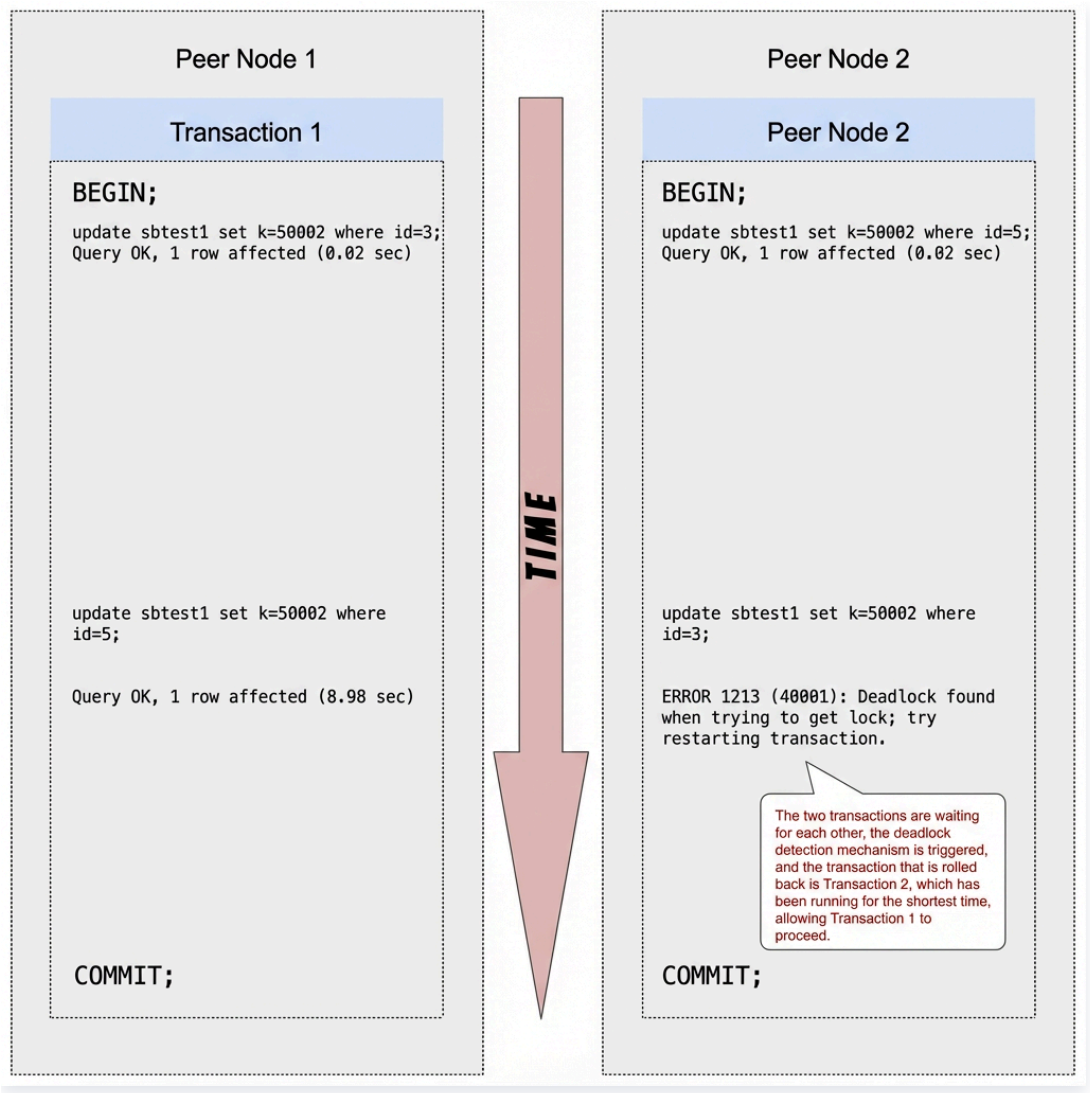


Row-Level Lock Conflict

When an operation is performed on a single key of a table, a row-level lock is applied. TDSQL Boundless maximizes concurrency capability of the database through its refined locking mechanism.
Specific examples will not be listed here. Compared to range locks, this approach only applies locks to individual rows, which can be analyzed independently.

Deadlock Conflict

Deadlock is a special case of row-level lock conflicts. It occurs when two or more sessions are waiting for data locked by the other party. Since both parties are waiting for the other, neither can complete the transaction to resolve the conflict.
TDSQL Boundless features automatic deadlock detection capability. By default, it rolls back the transaction with fewer data writes and returns an error, thereby releasing all other locks held by that session so that another session can proceed with its transaction.
In the following example, after Transaction 1 and Transaction 2 enter a deadlock, the deadlock detection mechanism takes effect. Transaction 2 is rolled back, allowing Transaction 1 to proceed.



Is There a Table-Level Lock at the Storage Layer?

InnoDB provides table-level S-locks and X-locks, but they offer "little practical value". These locks do not provide additional protection and only reduce concurrency capability. Therefore, the TDStore storage layer does not implement table-level locks. It only supports syntax parsing without actual functionality:

```
locktables sbtest1 read;
Query OK,0rows affected,1 warning (0.02 sec)

showwarnings;
+---------+------+-------------------------------------------------------------
-----------------------------+
|Level| Code | Message |
+---------+------+-------------------------------------------------------------
-----------------------------+
| Warning |8533|LOCK/UNLOCKoptionis used for compatibility only,and it
does not actually work.|
```

```
+---------+------+-------------------------------------------------------
---------------------------+
```

4. Best Practices

Lock timeouts and deadlocks are common issues in high-concurrency database systems. Effective lock management requires systematic optimization across three dimensions: business logic, database configuration, and troubleshooting.

Business Logic

Transaction design is the primary step in preventing lock issues. Adhering to the three principles of "**short transactions, lightweight operations, sequential access**" can effectively reduce the probability of lock conflicts.

1. Transaction execution time should be minimized. For instance, keep the number of rows operated per transaction under 2000. The more SQL statements included in a transaction and the larger the number of rows operated, the longer the lock holding time becomes, and the higher the probability of conflicts with other transactions.
2. Avoid performing user interactions during transactions; keep only essential data operations within the transaction to ensure rapid completion and resource release.
3. Ensure consistent order of resource access within transactions. When **multiple transactions need to access multiple resources, they must access these resources in a standardized sequence**, which is crucial for preventing deadlocks. For example, in transactions involving deducting inventory and creating orders, all transactions should either lock the inventory table first followed by the order table, or vice versa. However, it must be avoided that some transactions lock inventory first while others lock orders first, preventing circular wait scenarios.
4. TDSQL Boundless already supports Online DDL capabilities for most scenarios. However, it is still recommended to review the OnlineDDL documentation before DDL operations are initiated. Additionally, performing such operations during off-peak business hours is advised.

Database configuration

Secondly, when lock wait timeouts occur in production systems, priority should be given to verifying whether the configuration of relevant parameters is reasonable. For instances upgraded from previous versions, special attention must be paid to whether newly added lock control parameters in subsequent versions remain disabled for compatibility requirements. Enable them manually if necessary.
Database-related parameters:

1. **tdsql_lock_wait_timeout**: Controls the maximum lock wait time, with a default value of 50 seconds. In the scenario demonstrated in the diagram above, if a blocked session still cannot acquire the lock within 50 seconds, it will report the error "Lock wait timeout exceeded". Generally, there is no need to adjust

the default value. If severe lock conflicts occur in the business that cannot be quickly resolved, this value can be appropriately decreased as a temporary measure.

2. **tdstore_deadlock_detect (requires superuser privileges)**: Automatic deadlock detection feature. Default value is ON for newly purchased instances, and it is recommended to keep it enabled. For instances upgraded from older versions, this feature is disabled by default and can be manually enabled if required.
3. **tdstore_deadlock_victim (requires superuser privileges)**: Determines which transaction to roll back when a deadlock occurs [This parameter takes effect only when the tdstore_deadlock_detect deadlock detection feature is enabled]. The default value is "WRITE_LEAST", aligning with InnoDB behavior. Generally, no adjustment is needed. When set to "WRITE_LEAST", it prioritizes rolling back the transaction with fewer writes; when set to "START_LATEST", it prioritizes rolling back the transaction started later.

troubleshooting

Chapter 3 presents a comparative case analysis to intuitively demonstrate TDSQL Boundless' DDL/DML conflict handling mechanisms in both intra-node and cross-node scenarios. Errors encountered in actual business scenarios fall into two categories: DDL timeout failures or DML timeout failures. Follow the steps below for troubleshooting and resolution.

> **Note:**
> Refer to the definitions of dictionary tables in System Tables and System Views for the following content.

1. **DDL Timeout Failure:**

   If DDL execution reports the error "Lock wait timeout exceeded", it indicates that the session executing DDL is blocked by DML or DDL on the same node; if DDL execution reports the error ERROR 8542 (HY000): Acquire object lock 'test.sbtest1' from MC wait timeout, sql-node: node-tdsql3-xxxxxxxx-xxx, it indicates that the session executing DDL is blocked by DDL on another node.
   You can query performance_schema.metadata_locks to view the usage status of SQLEngine MDL locks on the current node [Note: In TDSQL Boundless, there's no need to set the performance_schema system variable to ON].

```
# Verify that session1 and session2 are connected to the same node.
# If not connected to the same node, the ddl can be executed
successfully. Refer to "Cross-node ddl-DML Conflict".

show variables like'hostname';

#session1
BEGIN;
```

```
UPDATE sbtest1 SET k =0WHERE id =999;

#session2
ALTERTABLE sbtest1 ADDCOLUMN new_column VARCHAR(255);

#View metadata_locks to see:
#The first row with LOCK_STATUS=GRANTED corresponds to session1,
indicating that the MDL lock has been acquired;
#The second row with LOCK_STATUS=PENDING corresponds to session2,
indicating that obtaining the MDL lock is pending.
#The broadcast HINT needs to be added at the beginning to specify
broadcasting queries on all nodes.

/*#broadcast*/select*from performance_schema.metadata_locks where
OBJECT_NAME='sbtest1' \G
***************************1.row***************************
          OBJECT_TYPE: TABLE
        OBJECT_SCHEMA: test
          OBJECT_NAME: sbtest1
          COLUMN_NAME: NULL
OBJECT_INSTANCE_BEGIN: 140384374661472
            LOCK_TYPE: SHARED_WRITE
        LOCK_DURATION: TRANSACTION
          LOCK_STATUS: GRANTED
               SOURCE: sql_parse.cc:6373
      OWNER_THREAD_ID: 4879164
       OWNER_EVENT_ID: 1
***************************2.row***************************
          OBJECT_TYPE: TABLE
        OBJECT_SCHEMA: test
          OBJECT_NAME: sbtest1
          COLUMN_NAME: NULL
OBJECT_INSTANCE_BEGIN: 140375267009376
            LOCK_TYPE: SHARED
        LOCK_DURATION: EXPLICIT
          LOCK_STATUS: PENDING
               SOURCE: ddl_executer.cc:245
      OWNER_THREAD_ID: 4879122
       OWNER_EVENT_ID: 1
2rowsinset(0.02 sec)
```

```
#When repeated queries are performed and show no change in the session
with LOCK_STATUS: GRANTED, locate the SESSION ID corresponding to the
thread holding the lock via OWNER_THREAD_ID. After confirming the
session can be safely terminated, first end it with the KILL command,
then reinitiate the DDL operation.
/*#broadcast*/select*from performance_schema.threads where
THREAD_ID=4879164\G
***************************1.row***************************
          THREAD_ID: 4879164
               NAME: thread/sql/one_connection
TYPE: FOREGROUND
     PROCESSLIST_ID: 2346946
   PROCESSLIST_USER: xxxxx
   PROCESSLIST_HOST: xxx.xxx.xxx.xxx
     PROCESSLIST_DB: test
PROCESSLIST_COMMAND: Sleep
   PROCESSLIST_TIME: 1330
  PROCESSLIST_STATE:
   PROCESSLIST_INFO:
   PARENT_THREAD_ID:
               ROLE:
       INSTRUMENTED: YES
            HISTORY: YES
    CONNECTION_TYPE: TCP/IP
       THREAD_OS_ID: 45448
     RESOURCE_GROUP:
       SQLEngine_id: node-tdsql3-db38679b-002
1rowinset(0.02 sec)

#KILL the lock holder (session with LOCK_STATUS: GRANTED).
#session1 was killed.

#session2
ALTERTABLE sbtest1 ADDCOLUMN new_column VARCHAR(255);
Query OK,0rows affected (1.67 sec)
Records: 0  Duplicates: 0  Warnings: 0
```

2. **DML Timeout Failure:**

If DML execution reports the error "Lock wait timeout exceeded", it generally indicates a failure to acquire a row-level lock or encountering a deadlock. For legacy instances upgraded from older versions, it is first recommended to check the automatic deadlock detection feature. If it is disabled, enabling it (without requiring a restart) is advised. Subsequently, if deadlocks occur again, the system will automatically resolve them.
If lock timeout issues persist, use the following method to identify the session ID holding the lock (Lock Holder), terminate the blocking session via KILL, release resources occupied by row locks, and allow the waiting session (Lock Waiter) to proceed.
You can query performance_schema.data_locks and performance_schema.data_lock_waits to view session information about locks held and locks waited for in TDStore [Note: In TDStore, there's no need to set the performance_schema system variable to ON].

```
#Regardless of whether session1 and session2 are connected to the same
node, the result remains the same; because row-level locks operate at
the storage layer, a conflict between the two sessions indicates they
both accessed the primary replica of the same storage node.

#session1
SELECT id FROM sbtest1 ORDERBY id limit10;
+----+
| id |
+----+
|1|
|3|
|5|
|7|
|9|
|11|
|12|
|13|
|14|
|15|
+----+
10rowsinset(9.23 sec)

BEGIN;
UPDATE sbtest1 SET k=50000WHERE id<=11AND id>=5;

#session2
INSERTINTO sbtest1(id)VALUES(8);
```

```
#Query the pessimistic lock information of the Lock Holder (Lock
Holder) on the current node.
#In TDStore, the range lock is left-closed and right-open, as seen in
the ENGINE_LOCK_ID column displaying the interval [5,12).

SELECT data_locks.*FROM performance_schema.data_locks,
performance_schema.data_lock_waits WHERE blocking_engine_lock_id =
engine_lock_id \G
***************************1.row***************************
ENGINE: RocksDB
                     ENGINE_LOCK_ID:
29374591168151726_[00002C7B80000005,00002C7B8000000C)
              ENGINE_TRANSACTION_ID: 29374591168151726
                           THREAD_ID: 1093359
                            EVENT_ID: NULL
                       OBJECT_SCHEMA:
                         OBJECT_NAME:
                      PARTITION_NAME: NULL
                   SUBPARTITION_NAME: NULL
                          INDEX_NAME: NULL
              OBJECT_INSTANCE_BEGIN: 140400912025696
                           LOCK_TYPE: PRE_RANGE
                           LOCK_MODE: Write
                         LOCK_STATUS: GRANTED
                           LOCK_DATA: NULL
                           START_KEY: 00002C7B80000005
                             END_KEY: 00002C7B8000000C
                  EXCLUDE_START_KEY: 0
            BLOCKING_TRANSACTION_NUM: 1
BLOCKING_CHECK_READ_TRANSACTION_NUM: 0
                      READ_LOCKED_NUM: 0
                           TINDEX_ID: 11387
                      DATA_SPACE_TYPE: DATA_SPACE_TYPE_USER
                REPLICATION_GROUP_ID: 257
                 KEY_RANGE_REGION_ID: 1302791
1rowinset(0.04 sec)

#View the session information of the Lock Holder (Lock Holder).
```

```
# !!! In TDStore's PERFORMANCE_SCHEMA.DATA_LOCKS table, the thread_id
column refers to processlist_id; whereas in official mysql, thread_id
refers to the thread_id in Performance_Schema.threads. This issue will
be fixed in future versions.

select*from information_schema.processlist where id=1093359\G
***************************1.row***************************
           ID: 1093359
USER: tdsql_admin
         HOST: xxx.xxx.xxx.xxx:35956
           DB: test
      COMMAND: Sleep
TIME: 946
        STATE: NULL
         INFO: NULL
      TIME_MS: 945727
    ROWS_SENT: 0
ROWS_EXAMINED: 4
1rowinset(0.12 sec)


#After the lock holder (Lock Holder) is KILLed, the blocked session
was executed successfully.
#session1 was killed.

#session2
INSERTINTO sbtest1(id)VALUES(8);
Query OK,1row affected (43.78 sec)

#Note: In high-concurrency scenarios, the queue of Lock Waiters may be
lengthy, potentially resulting in new Lock Holders emerging. Multiple
termination attempts may be required.
```

Data Intelligent Scheduling and Related Practices for Performance Optimization

Last updated：2026-03-06 18:50:08

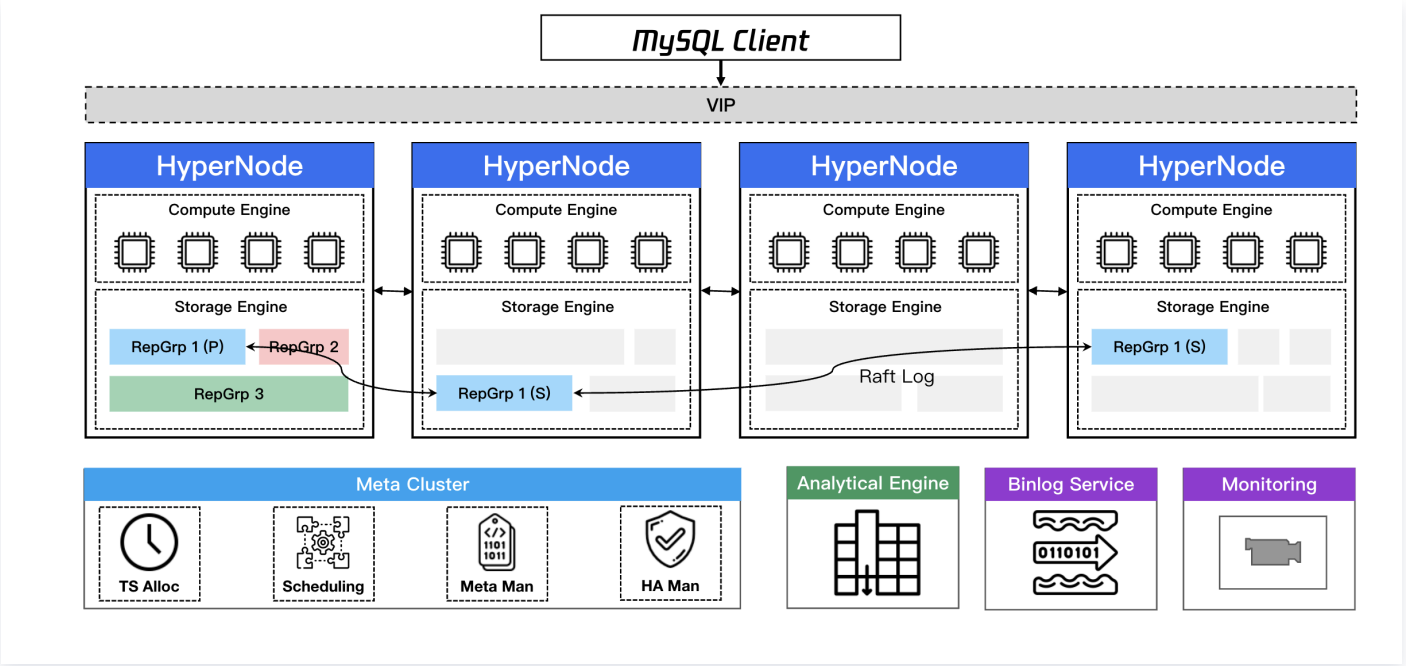
Introduction

After years of technological evolution, distributed databases have transcended the limitations of traditional "sharding" scenarios, evolving from a stopgap measure for massive data sets to a core architecture supporting elastic business expansion. TDSQL Boundless achieves dynamic adaptation between database operation modes and business scales through its (Integrated Centralized-Distributed Solution): During initial business phases, the system operates with full compatibility to centralized databases, preserving development practices and Ops frameworks. When businesses enter rapid growth stages, seamless transition to distributed mode requires only configuration-level adjustments. Online elastic scaling handles traffic spikes and data expansion. This process achieves two breakthroughs:

1. **Zero-modification Smooth Evolution**: The business logic layer does not need to adapt to distributed transaction logic, while the data layer maintains access consistency through intelligent routing.
2. **Seamless Architecture Transition**: The upgrade process from centralized to distributed architecture maintains business continuity, eliminating time-consuming data migration and risks of service disruption inherent in traditional solutions.

The integrated centralized-distributed solution of TDSQL Boundless primarily relies on intelligent data scheduling technology. Intelligent data scheduling technology not only realizes the integrated centralized-distributed solution but also innovatively introduces an optimization mechanism for data affinity in distributed scenarios. By combining predefined rules and customized policies, it intelligently schedules data units with strong business coupling to the same physical node, fundamentally avoiding the performance overhead of cross-node RPC access. This article will share the latest product in the TencentDB TDSQL series: TDSQL Boundless's intelligent data scheduling and related practices for performance optimization.

TDSQL Boundless Intelligent Data Scheduling Component TDMetaCluster



TDSQL Boundless is a high-performance, high-availability enterprise-grade distributed database solution developed by Tencent for financial-grade application scenarios. It adopts a containerized cloud-native architecture, delivering high-performance computing capabilities and cost-effective massive storage at the cluster level.

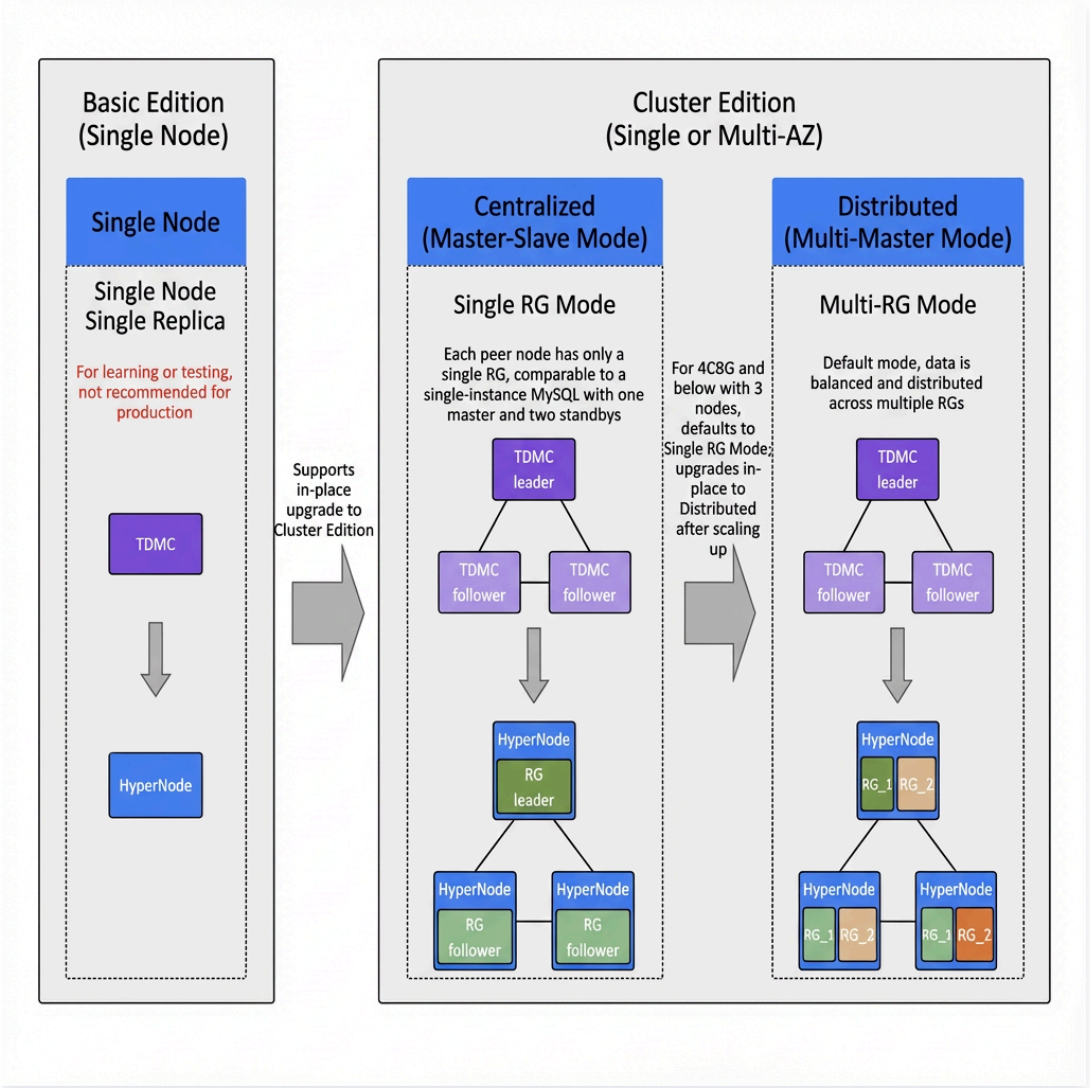
TDSQL Boundless architecture and features: fully distributed + storage-compute integration/storage-compute separation + data plane/control plane separation + high scalability + global consistency + high compression ratio.

In TDSQL Boundless, the component responsible for intelligent data scheduling is TDMetaCluster (TDMC for short, shown on the right side of the figure above), which serves as the central management and control module for TDSQL Boundless database instances.

1. **TDMC architecture:** A metadata management service based on the Raft protocol with one primary and two secondary nodes, where the Leader node handles requests.
2. **TDMC data plane**:
   - 2.1 Assign globally unique and monotonically increasing transaction IDs.
   - 2.2 Manage metadata for TDStore and SQLEngine.
   - 2.3 Manage the data routing information for Replication Groups.
   - 2.4 Manage global MDL locks.
3. **TDMC control plane**:
   - 3.1 Schedule the splitting, merging, migrating, and switching over of Replication Groups and Regions.
   - 3.2 Schedule the scaling of the storage layer.
   - 3.3 Schedule the load balancing of the storage layer.
   - 3.4 Issue alarms for abnormal events in various dimensions.

As the core scheduling engine for TDSQL Boundless instances, it not only manages the organization and distribution of data replicas through predefined rules but also supports flexible configuration of customized policies based on business needs. Next, we will explore how to select the appropriate deployment mode: centralized or distributed—based on business scenarios. While TDSQL Boundless, as a native distributed database, offers robust distributed processing capabilities, a centralized mode may better align with practical application requirements in specific scenarios.

TDSQL Boundless Centralized-Distributed Integration



First, based on assessment of business scale: including data volume, access frequency, response time, and scalability requirements, select the appropriate TDSQL Boundless specification. Even if initial estimates have deviations, there is no need to worry, as TDSQL Boundless, with its efficient AS capability, can flexibly adapt to various dynamically changing agile business scenarios.

As shown in the figure above, the deployment mode is selected as follows:

1. **Trial**: Select the "Basic Edition" (single node, single replica), without the feature of high availability.
2. **Production environment**: Select the "Cluster Edition" [triple-replica architecture based on the Raft majority protocol].

3. **Advanced Disaster Recovery**: The Cluster Edition supports both "single-AZ" and "multi-AZ" deployments, with the latter enabling disaster recovery across three AZs within the same region.
4. **Small-scale business**: When specifications are 4C/8G or below with 3 nodes, the system automatically enables centralized mode (single RG mode), where primary replicas are centrally stored, ensuring compatibility with the standalone MySQL experience.
5. **Medium-to-large-scale business**: When specifications exceed 4C/8G or the number of nodes exceeds 3, the system automatically switches to distributed mode (multi-RG mode), where data is evenly distributed and peer-to-peer read/write operations are performed across multiple nodes, ensuring high performance and high availability.

It is recommended to conduct functional and performance tests based on actual business scenarios to verify whether the configuration meets peak demand.
After understanding how to select TDSQL Boundless configurations, you may have no doubts about the centralized mode but still worry whether the distributed architecture affects performance. Such concerns are valid, and technology for intelligent database scheduling is key to solving such performance issues. Before it is delved deeper into, it's essential to grasp a critical design in distributed systems: data balancing—the cornerstone that ensures system performance and reliability. Next, we will detail the principles of data balancing and its practical implementation in TDSQL Boundless.

TDSQL Boundless Data Balancing

Data balancing is a key feature of distributed systems, encompassing two major dimensions: **Capacity Balancing** and **Hotspot Balancing**. Capacity balancing optimizes global resource utilization through uniform data distribution, while hotspot balancing focuses on resolving localized high-load issues.

Technical Highlights of TDSQL Boundless

1. **Capacity Balancing**

   1.1 **Three-tier Storage Model and Resource Pre-allocation**:

   Based on the three-tier storage model design (refer to Product Overview), each node pre-creates a primary RG, which can accommodate Regions for 100 tables by default. New RGs are automatically created only after exceeding this limit, reducing resource fragmentation, lowering management overhead, and improving performance.

   1.2 **Control of Two-dimensional Split**:

   - **Region-level**: Triggers a split at 256M/5 million keys and remains with the original RG.
   - **RG-level**: Triggers a split at 32G/160 million keys. A single-table RG splits only when exceeding 20% of the node capacity to avoid distributed transactions. After the split is performed, automatic migration occurs at the RG level to maintain capacity balancing.

   1.3 **Policy of Migration Priority**:

   Prioritize migrating follower replicas. Leader replicas are migrated only when they exist on a node, ensuring the process remains transparent to services.

1.4 **Linkage of Elastic Scaling**:

Data rebalancing is automatically triggered to adapt to changes in the number of nodes.

2. **Hotspot Balancing**

2.1 **Engine for Intelligent Decision**:

Employing a weighted moving average algorithm, it collects three-level traffic metrics (node/RG/data object) in real time, then identifies hotspot sources at each level and executes differentiated scheduling operations—such as cross-node leader switching or splitting + leader switching combinations.

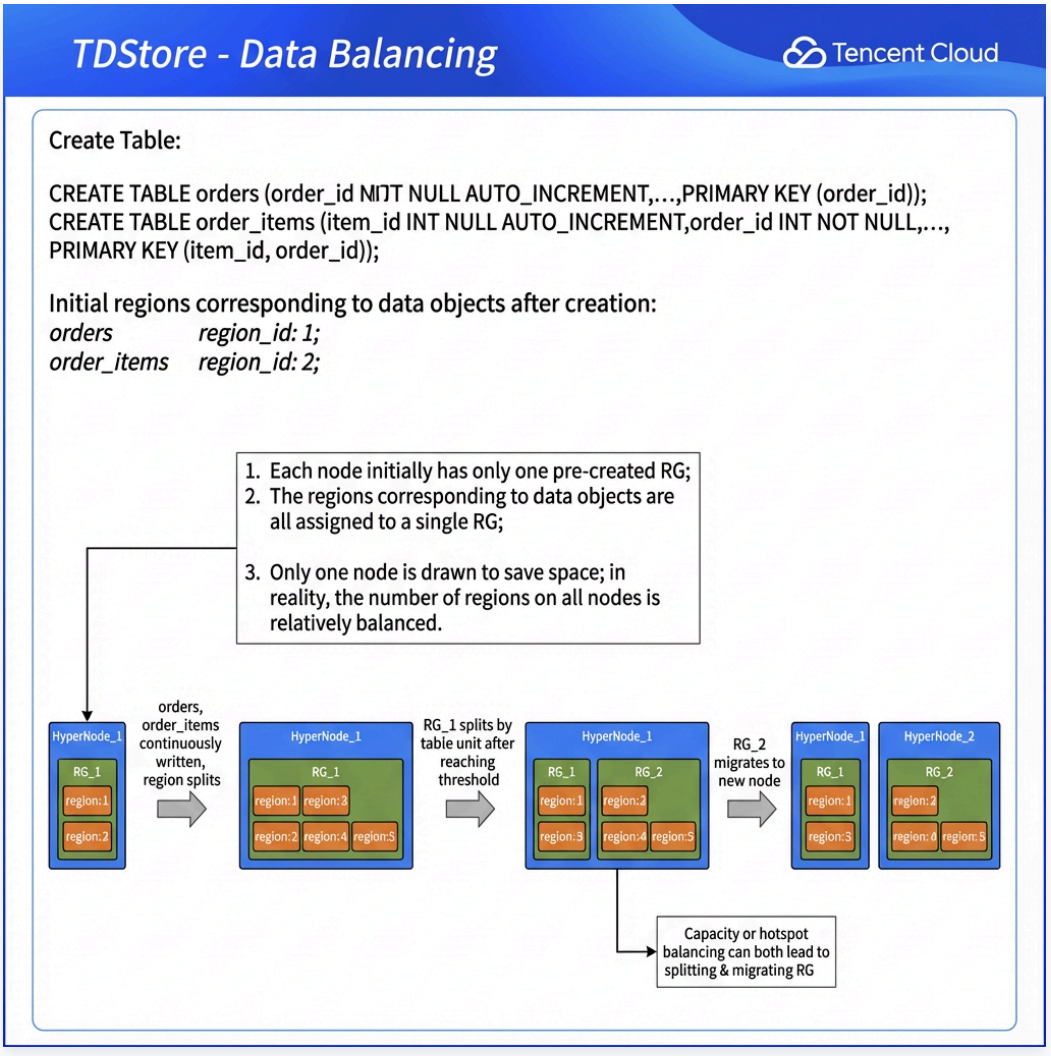
Mechanism for Stability Assurance:

Cooldown Period for Scheduling: After a hotspot RG is scheduled, it enters a cooldown state to prevent such RG from frequent leader switching.

Memory of Historical Scheduling: Records node migration trajectories to prevent hotspot ping-ponging.

2.2 **Mode for Scenario-based Scheduling**:

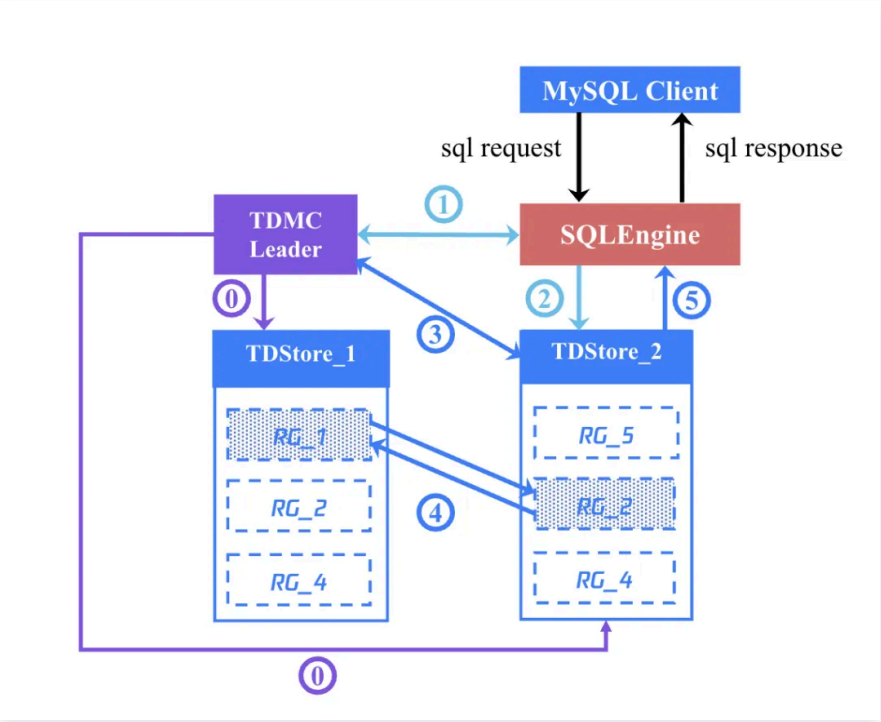
Manual configuration of read-write hotspot modes is supported, with read hotspot priority enabled by default to flexibly adapt to diverse business requirements.



However, for distributed database systems, mere data balancing does not equate to an "optimal" state. If the scheduling module fails to fully comprehend the logical meaning of database tables in the storage

layer, the system may still encounter performance bottlenecks.

To better illustrate this point, let's examine a typical distributed application scenario: when an application client submits an SQL statement to a TDSQL Boundless database instance, how TDMC (the management module) collaborates with SQLEngine (SQL engine) and TDStore (storage engine) to process the request.



Step 0: TDMC periodically issues global minimum snapshot points to all TDStores to ensure global consistency for read operations.

Step 1: The SQLEngine receives the SQL statement, initiates a transaction, and retrieves the timestamp of transaction start `begin_ts` from TDMC; it checks its local cache for RG routing information corresponding to the SQL statement. If none exists, it sends a request for key range query to TDMC to obtain the latest routing information.

Step 2: SQLEngine sends the request to the TDStore where the Leader replica of the corresponding RG resides.

Step 3: The Leader replica, acting as the coordinator, initiates a two-phase transaction and retrieves prepare_ts from TDMC.

Step 4: After all participants are prepared, retrieve `commit_ts` from TDMC and commit the transaction; if a leader switch occurs in other RGs involved during the process, query their latest Leader information from TDMC.

Step 5: Commit the transaction; upon successful completion, return the result to SQLEngine, which then returns the result to the client.

The execution process described above may face the following three performance challenges:

1. **Separation of Compute and Storage**: If compute nodes and storage nodes are strictly separated, network latency and bandwidth limitations can negatively impact performance. Data transfer between

different nodes introduces additional overhead, thereby reducing the response speed of the overall system.

2. **Scattered Storage of Data Objects**: If relevant data objects of the same table are excessively scattered—for example, table data stored in RG_1 while its secondary index resides in RG_2—even a simple write operation for a single-row transaction requires coordination through distributed transactions. This inevitably increases complexity and processing time.
3. **Cross-Machine Join Queries**: When multiple tables with relationships (such as Table A stored in RG_1 and Table B stored in RG_2) require JOIN operations or participate in distributed transactions, these operations will inevitably involve cross-machine communication, further exacerbating latency and resource consumption.

To address the first challenge, TDSQL Boundless adopts an integrated compute-storage architecture by physically binding SQLEngine with TDStore to form peer nodes, enhancing localized data access performance (using function calls instead of RPC). As for the remaining two challenges, they are resolved through TDSQL Boundless' intelligent scheduling mechanism. Next, we will delve into how TDSQL Boundless tackles these challenges.

TDSQL Boundless Intelligent Scheduling for Data

TDSQL Boundless implements scheduling for data affinity through a multi-level rule system, which includes three types of rules:

General rules: Such as the structural coupling of data objects in relational databases, this rule remains unchanged.

Predefined rules: Logical coupling that may enhance performance; if they do not align with business requirements, this rule can be modified.

Custom rules: Users can create them on their own based on business requirements.

Through this multi-level rule system, TDSQL Boundless achieves an optimal balance between system performance and business flexibility: ensuring efficient operation while significantly reducing transaction latency and improving speed of query response. Next, we will delve into the details of its technical implementation.

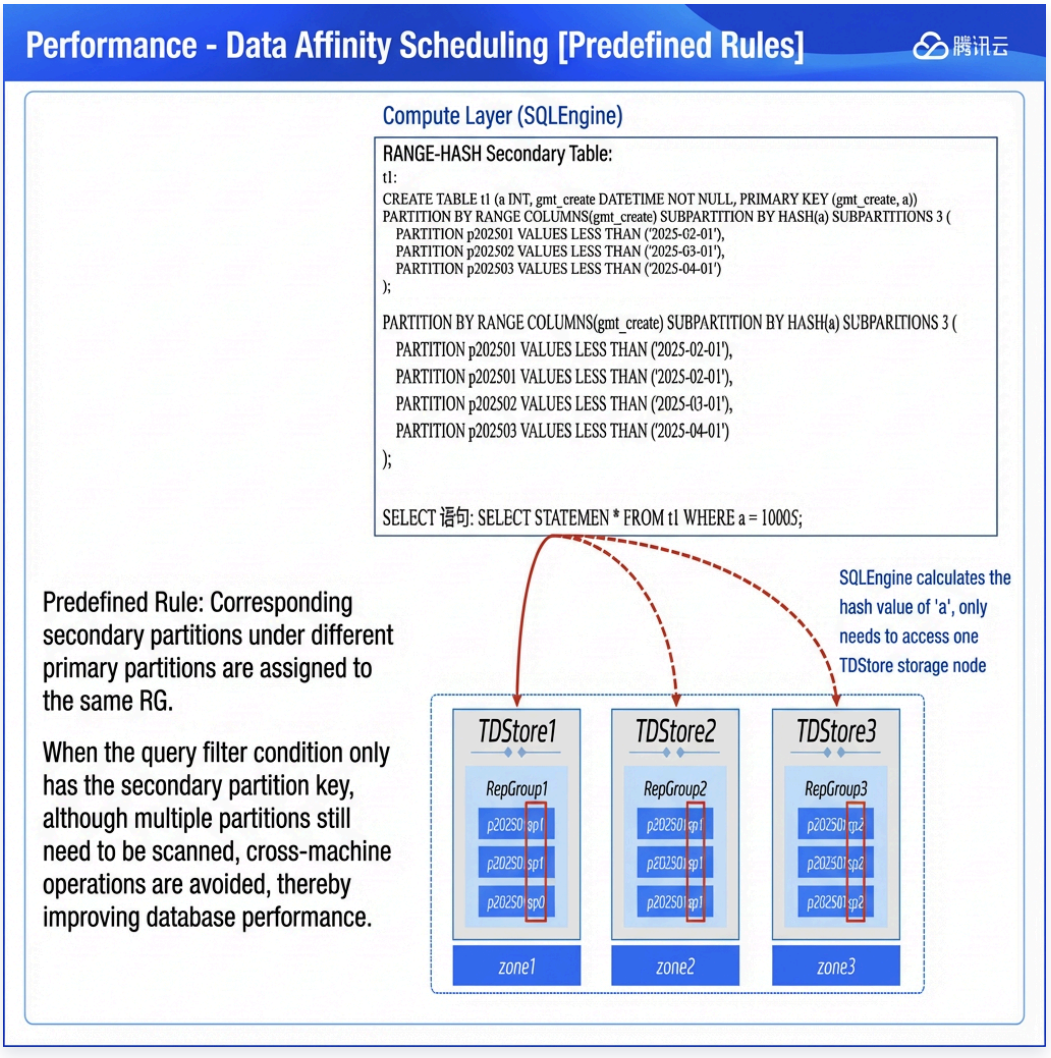
1. **General rules**

   The following diagram illustrates the three-layer data model of TDStore: In the system, a table does not directly correspond to a physical file but is first mapped to a range-based Region. The right side of the diagram shows that Table 1 is allocated two Regions, 01 and 03, where 01 corresponds to the primary key and 03 to its secondary index. When a table's data volume becomes very large, the Region can be split and then mapped to other data nodes. Between Regions and physical data, a Replication Group (RG) is introduced, allowing flexible placement of different range-based Regions into this RG. Each RG corresponds to a Raft log stream. If a transaction only modifies data within this replication group, it can be committed after completing a single-node transaction. However, if a transaction involves multiple replication groups, it must proceed through a two-phase commit as a distributed transaction.

Through built-in general rules, TDStore leverages data affinity to eliminate distributed transactions: by default storing a table's primary key and its corresponding secondary index within the same RG [as shown in the bottom-left of the figure, Table1's primary key is Region 1 and its secondary index is Region 3, both belonging to RG 1]. This ensures that updates and writes to the table are always processed as single-node transactions.

2. **Predefined rules**

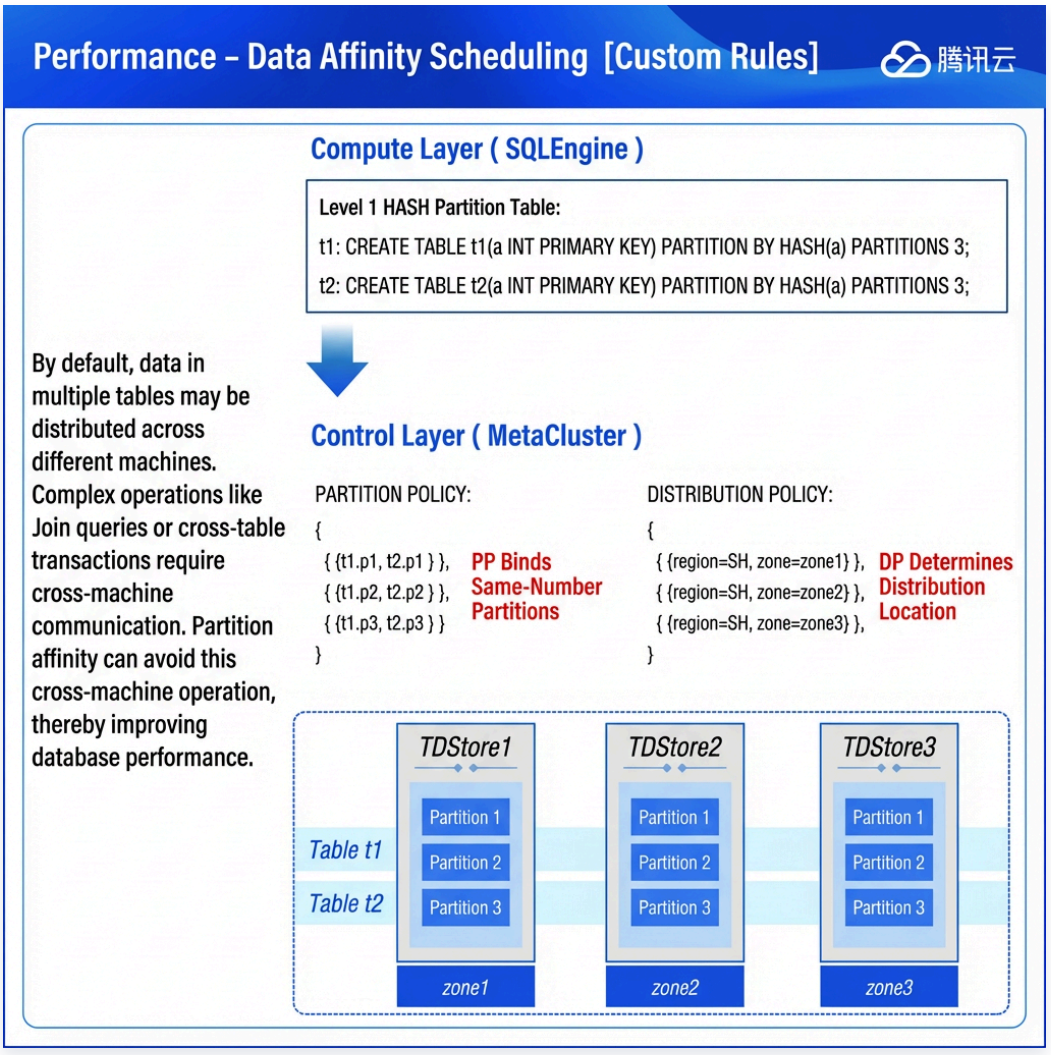
   TDSQL Boundless supports predefined data affinity rules, which are best practices summarized from daily operations adapting to business logic. For example: when secondary partitioned tables are created, corresponding secondary partitions under different primary partitions are assigned to the same RG, such as placing t1.p0.sp0 and t1.p0.sp0 in the same RG. If specific business requirements exist, there are methods to modify this default behavior.



3. **Custom rules**

   Additionally, TDSQL Boundless supports custom data affinity rules based on business logic. Users can control the physical storage proximity of a group of tables according to business data correlations, further optimizing transaction processing performance. Specifically, this is achieved by creating a binding policy for same-numbered partitions and node distribution policies for a table group through specific SQL

commands, then binding them to the table group (which requires identical partition types, partition counts, and partition values). The system will then collocate same-numbered partitions of these tables on a single machine, avoiding cross-node queries and distributed transactions to enhance performance.



TDSQL Boundless Practice of Partition Affinity Optimization

In the practical section, we provide an example of data affinity with "custom rules." Specifically, we will compare the performance of two tables during JOIN queries: one scenario where the tables satisfy data affinity, and another where they do not. Through this comparison, we can gain a clearer understanding of the practical impact of data affinity on query performance.

Hardware Environment

Node Type	Node Specifications	Number of Nodes
HyperNode	16-Core CPU/32 GB of Memory/Enhanced SSD 300 GB	3

Test Objective

TDSQL Boundless first supports implicit partition affinity for primary HASH-partitioned tables. As long as related tables are created as primary HASH partitions with the same number of partitions, same-numbered partitions will be scheduled to the same replication group by default. [Future releases will support manually creating affinity policies and richer types of partitioned tables.]
The data model for this test involves a JOIN query between orders and order details.

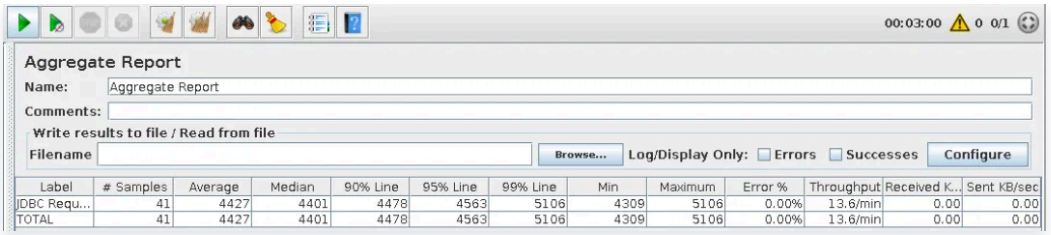
Data Preparation

```
-- Order table orders, primary hash-partitioned table:
CREATETABLE orders (
    order_id INTNOTNULLAUTO_INCREMENT,
    customer_id INTNOTNULL,
    order_date DATENOTNULL,
    total_amount DECIMAL(10,2)NOTNULL,
PRIMARYKEY(order_id)
)
PARTITIONBYHASH(order_id)
PARTITIONS 3;

-- Order details table order_items, with 1-3 items per order randomly,
primary hash-partitioned table:
CREATETABLE order_items (
    item_id INTNOTNULLAUTO_INCREMENT,
    order_id INTNOTNULL,
    product_id INTNOTNULL,
    quantity INTNOTNULL,
    price DECIMAL(10,2)NOTNULL,
PRIMARYKEY(item_id, order_id),
INDEX idx_order_id (order_id)
)
PARTITIONBYHASH(order_id)
PARTITIONS 3;

-- Data generation process omitted. Final record counts for both tables:
MySQL [klose]>selectcount(*)from orders;
+----------+
|count(*)|
+----------+
|1000000|
+----------+
1rowinset(0.10 sec)
```

```
MySQL [klose]>selectcount(*)from order_items;
+----------+
|count(*)|
+----------+
|1999742|
+----------+
1rowinset(0.39 sec)

-- Check data distribution to ensure same-numbered partitions are in the
same RG within peer nodes:
SELECT
  b.rep_group_id, a.data_obj_name, a.schema_name, a.data_obj_name,
c.leader_node_name,SUM(b.region_stats_approximate_size)AS
size,SUM(b.region_stats_approximate_keys)AS key_num
FROM
  INFORMATION_SCHEMA.META_CLUSTER_DATA_OBJECTS a,
  INFORMATION_SCHEMA.META_CLUSTER_REGIONS b,
  INFORMATION_SCHEMA.META_CLUSTER_RGS c
WHERE
  a.data_obj_id = b.data_obj_id
and b.rep_group_id = c.rep_group_id
and a.data_obj_type notlike'%index%'
and a.data_obj_type notlike'%AUTOINC%'
and a.schema_name ='klose'
and data_obj_name notlike'%bak%'
GROUPBY
  b.rep_group_id, a.schema_name, a.data_obj_name
ORDERBY1,2,3;
+--------------+----------------+-------------+----------------+--------
------------------+---------+---------+
| rep_group_id | data_obj_name  | schema_name | data_obj_name  |
leader_node_name          | size    | key_num |
+--------------+----------------+-------------+----------------+--------
------------------+---------+---------+
|868437| orders.p1       | klose       | orders.p1       | node-tdsql3-
c1528b96-002|3909181|333334|
|868437| order_items.p1 | klose       | order_items.p1 | node-tdsql3-
c1528b96-002|7708701|666508|
```

```
|869169| orders.p0      | klose       | orders.p0      | node-tdsql3-
c1528b96-001|3792790|333333|
|869169| order_items.p0 | klose       | order_items.p0 | node-tdsql3-
c1528b96-001|7575671|666757|
|869736| orders.p2      | klose       | orders.p2      | node-tdsql3-
c1528b96-003|3782701|333333|
|869736| order_items.p2 | klose       | order_items.p2 | node-tdsql3-
c1528b96-003|7550956|666477|
+--------------+---------------+-------------+----------------+--------
-----------------+---------+---------+

-- Test JOIN queries using Jmeter to observe the performance of local
scans:
SELECT COUNT(*) FROM orders a JOIN order_items b ON
a.order_id=b.order_id;

-- Execution plan: Each node starts a worker thread to complete parallel
subtasks on the local node. Using any sub-partition (p0-p2) of the
orders table on the current node as the driving table, each row record
is used to quickly find matching rows in the order_items table via index
lookup. Results from all worker threads are then aggregated and
returned. Execution efficiency will be significantly improved if all
operations are local scans:
|-> Aggregate: count(0)(cost=763055.75rows=2000000)
-> Gather (slice: 1, workers: 3)(cost=563055.75rows=2000000)
-> Aggregate: count(0)(cost=763055.75rows=2000000)
-> Nested loopinnerjoin(cost=563055.75rows=2000000)
->Index scan on a usingPRIMARY,with parallel scan ranges:
3(cost=112507.50rows=1000000)
->Index lookup on b using idx_order_id (order_id=a.order_id)
(cost=0.25rows=2)
```

- Single process, continuously executed for 3 minutes, completing 41 requests during this period with an average response time of 4.4s:



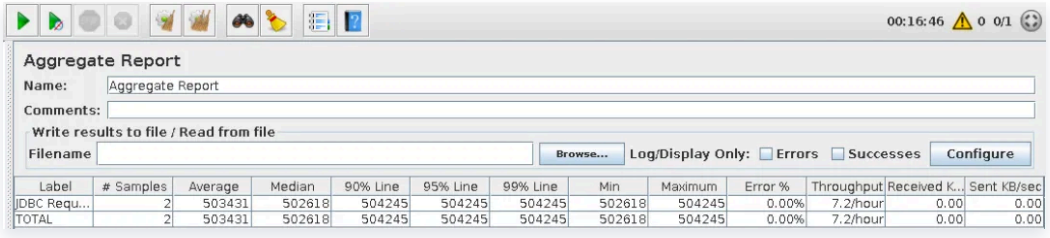
- Simulating scenarios where older versions of TDSQL Boundless did not automatically apply partition affinity:

```
-- Currently, same-numbered partitions are all located on the same
nodes:
+--------------+----------------+-------------+----------------+------
--------------------+---------+---------+
| rep_group_id | data_obj_name   | schema_name | data_obj_name   |
leader_node_name          | size     | key_num |
+--------------+----------------+-------------+----------------+------
--------------------+---------+---------+
|868437| orders.p1       | klose        | orders.p1       | node-tdsql3-
c1528b96-002|3909181|333334|
|868437| order_items.p1  | klose        | order_items.p1  | node-tdsql3-
c1528b96-002|7708701|666508|
|869169| orders.p0       | klose        | orders.p0       | node-tdsql3-
c1528b96-001|3792790|333333|
|869169| order_items.p0  | klose        | order_items.p0  | node-tdsql3-
c1528b96-001|7575671|666757|
|869736| orders.p2       | klose        | orders.p2       | node-tdsql3-
c1528b96-003|3782701|333333|
|869736| order_items.p2  | klose        | order_items.p2  | node-tdsql3-
c1528b96-003|7550956|666477|
+--------------+----------------+-------------+----------------+------
--------------------+---------+---------+

-- Simulating the state without default affinity of partitions in
previous versions of TDSQL Boundless, splitting RGs and scattering
them:

ALTER INSTANCE SPLIT RG 868437BY'table';
ALTER INSTANCE SPLIT RG 869169BY'table';
ALTER INSTANCE SPLIT RG 869736BY'table';
```

```
ALTER INSTANCE TRANSFER LEADER RG 869169TO'node-tdsql3-c1528b96-002';
ALTER INSTANCE TRANSFER LEADER RG 868437TO'node-tdsql3-c1528b96-003';
ALTER INSTANCE TRANSFER LEADER RG 869736TO'node-tdsql3-c1528b96-001';
ALTER INSTANCE TRANSFER LEADER RG 72133681TO'node-tdsql3-c1528b96-
001';

-- Adjust TDMC parameters to prevent switch-back of the pre-created RG
leader:
Set the control parameter of TDMC check-primary-rep-group-enabled to
0.

-- Check data distribution to confirm that same-numbered partitions
are no longer located on the same peer nodes:
SELECT
  b.rep_group_id, a.data_obj_name, a.schema_name, a.data_obj_name,
c.leader_node_name,SUM(b.region_stats_approximate_size)AS
size,SUM(b.region_stats_approximate_keys)AS key_num
FROM
  INFORMATION_SCHEMA.META_CLUSTER_DATA_OBJECTS a,
  INFORMATION_SCHEMA.META_CLUSTER_REGIONS b,
  INFORMATION_SCHEMA.META_CLUSTER_RGS c
WHERE
  a.data_obj_id = b.data_obj_id
and b.rep_group_id = c.rep_group_id
and a.data_obj_type notlike'%index%'
and a.data_obj_type notlike'%AUTOINC%'
and a.schema_name ='klose'
and data_obj_name notlike'%bak%'
GROUPBY
  b.rep_group_id, a.schema_name, a.data_obj_name
ORDERBY1,2,3;

+--------------+----------------+-------------+-----------------+------
--------------------+---------+---------+
| rep_group_id | data_obj_name  | schema_name | data_obj_name  |
leader_node_name         | size    | key_num |
+--------------+----------------+-------------+-----------------+------
--------------------+---------+---------+
|869736| orders.p2      | klose       | orders.p2      | node-tdsql3-
c1528b96-001|3782701|333333|
```

```
|72133681| order_items.p0 | klose       | order_items.p0 | node-
tdsql3-c1528b96-001|7575671|666757|
|869169| orders.p0       | klose       | orders.p0      | node-tdsql3-
c1528b96-002|3792790|333333|
|72132949| order_items.p1 | klose       | order_items.p1 | node-
tdsql3-c1528b96-002|7708701|666508|
|868437| orders.p1       | klose       | orders.p1      | node-tdsql3-
c1528b96-003|3909181|333334|
|72134504| order_items.p2 | klose       | order_items.p2 | node-
tdsql3-c1528b96-003|7550956|666477|
+--------------+---------------+-------------+----------------+------
--------------------+---------+---------+

-- Execute the JOIN query again. Since same-numbered partitions are
now all on different peer nodes, every row in the orders table
requires an RPC call for index matching in the order_items table.
Combined with approximately 1-3ms latency between three AZs, the
execution efficiency is expected to drop significantly:
SELECT COUNT(*) FROM orders a JOIN order_items b ON
a.order_id=b.order_id;
```

- Single process, continuously executed for 16 minutes and 46 seconds, during which 2 requests were completed, with an average response time of 8 minutes and 23 seconds, **the execution efficiency differs by a full 113 times.**



Future Development of Partition Affinity in TDSQL Boundless

We can see that the integrated centralized-distributed capability of distributed databases effectively bridges the gap in the evolution from traditional relational databases to distributed architectures. Simultaneously, when business scale grows to reach the trigger threshold for distributed architecture, the data intelligent scheduling capability based on a multi-level rule system significantly enhances business access efficiency and actual performance. The combination of smooth architectural evolution and intelligent dynamic scheduling provides a robust and intelligent evolution path for digital transformation.
TDSQL Boundless is in a period of rapid development, with all features being user-demand-oriented and undergoing continuous rapid iteration and refinement. We are still continuously optimizing the data intelligent

scheduling feature. The current version has pioneered adaptation of implicit partition affinity for commonly used primary HASH-partitioned tables. In future releases, we will further extend support for implicit partition affinity covering primary KEY-partitioned tables and range/list + hash secondary partitioned tables. Simultaneously, we will introduce flexibly customizable explicit partition affinity policies, with initial support for specification of explicit affinity for primary HASH-partitioned tables and regular tables.

TDSQL Boundless Selection Guide and Practical Tutorial

Last updated：2026-02-10 11:06:03

1. TDSQL Boundless Specifications Selection

- If migration is performed from InnoDB/B+ Tree storage, TDSQL uses LSM structure with compression enabled by default, the space can be estimated as 1/3 of the original space (single replica). For example: If there is 1T of data in InnoDB (single replica; if the system is configured as one primary and two secondaries, the total space is 3T), after migration is performed to TDSQL Boundless, the single replica space is approximately 333G, and with the default three replicas, it is about 1T. After data migration completes, disk utilization will be checked for scale in/out operations.
- The selection of high single node configuration and fewer nodes quantities is prioritized: for instance, if the total resource requirement is 12c/24g/600G, a configuration of 4c/8g/200G x 3 nodes is preferable to 2c/4g/100G x 6 nodes. This is because small nodes incur a higher proportion of distributed transactions and communication overhead between nodes.
- When scaling out is performed, vertical scaling should be prioritized; conversely, when scaling in is performed, reducing nodes should be prioritized.
- TDSQL Boundless enables dynamic scaling of CPU / memory / disk resources without impacting business operations.

Reference Data for Performance

The following table presents: Sysbench performance data for a 3-node configuration with 16Core CPU / 32GB Memory / Enhanced SSD CBS, testing 32 tables each containing 10 million records.
For details, see Performance Testing > Sysbench Test.

Scenario	Threads	TPS	QPS	P95 Latency (ms)
Point Query	32	42210.44	42210.44	1.06
	64	73809.92	73809.92	1.21
	128	143840.1	143840.08	1.27
	256	193770.5	193770.5	1.79
Read Only	32	3193.09	51089.45	12.75
	64	5970.22	95523.52	16.12
	128	7419.62	118713.85	31.37
	256	7892.26	126276.08	62.19
Index Update	32	7659.33	7659.33	5.57
	64	14070.75	14070.75	6.32
	128	23513.47	23513.47	7.89
	256	37637.48	37637.48	10.65
Non-index Update	32	8408.54	8408.54	5.09
	64	14303.25	14303.25	5.99
	128	25299.7	25299.7	7.56
	256	39764.65	39764.65	10.09
Write Only	32	4150.32	24901.94	10.27
	64	7997.37	47984.2	11.87
	128	13516.77	81100.61	15
	256	18488.22	110929.31	25.74
Mixed Read/Write	32	1863.3	37266.02	24.83
	64	3374.92	67498.4	29.19
	128	4356.41	87128.23	39.65
	256	4495.55	89911.04	82.96

2. Using TDSQL Boundless

Recommended Use Cases

- **Business Database and Table Sharding Scenario**: After migrating to TDSQL Boundless, you no longer need to shard databases or tables. This applies whether your business originally used TDSQL MySQL InnoDB distributed architecture or implemented database/table sharding independently.
- **Business Scenarios with High Storage Costs**: TDSQL Boundless features built-in compression capabilities that significantly reduce storage costs.
- **High Write Volume Scenarios**: such as log streaming scenarios.
- **HBase Replacement Scenario**: Provides support for secondary indexes, cross-row transactions, and so on.

SQL Limitations and Differences

TDSQL Boundless is syntactically compatible with MySQL 8.0, with minor limitations:

- Foreign keys are not supported.
- Virtual columns, GEOMETRY data type, descending indexes, and full-text indexes are not supported.
- The cache for auto-increment fields defaults to 100, ensuring global uniqueness but not global monotonic increment. Setting the auto-increment cache to 1 guarantees globally monotonically increasing values, but impacts write performance for batch operations that explicitly specify auto-increment values.

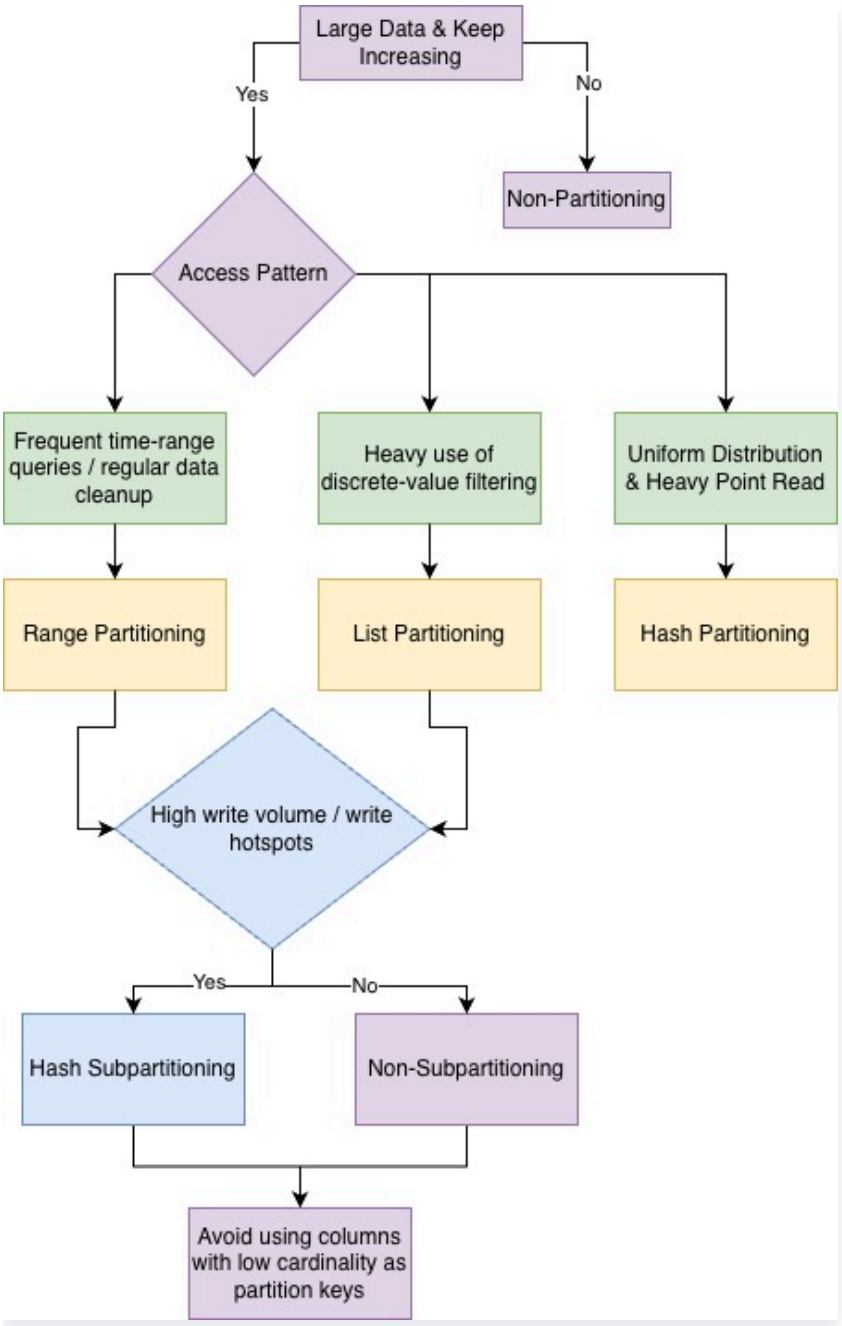
For more limitations and differences, see MySQL Compatibility.

Partition Table

- It is recommended to use HASH partitioning to distribute write hotspots. The number of partitions should be an integer multiple of the number of nodes. During queries, prioritize using HASH partition columns and avoid selecting columns with high duplicate values as partition keys.
- For scenarios requiring periodic purging of expired data, RANGE partitioning is recommended. Meanwhile, if there is high write volume and a need to distribute write hotspots, it is advisable to create tables using RANGE + HASH subpartitioning. Queries should optimally include predicate conditions with both the time column and HASH partition key.

Avoid excessive partitioning. For example, if data needs to be retained for 3 years, daily partitioning would create over 1000 partitions. Consider weekly or monthly partitioning instead to reduce the number of partitions.

For the decision-making process regarding partition type selection, see the following figure:



Online DDL Capability

TDSQL Boundless supports most Online DDL operations (including certain Inplace DDL operations that only require modifying metadata, collectively referred to as Online DDL). For specific capabilities, see OnlineDDL Description.

3. TDStore Storage

Unlike traditional MySQL master–slave replication, TDSQL Boundless employs smaller storage management units called RGs (replica groups). Each RG uses the Raft protocol to ensure replica consistency. The Leader of each RG can be distributed to any node in the instance. Therefore, any node in TDStore can handle read/write requests, enabling more efficient utilization of resources across all nodes.