# 分布式数据库 TDSQL

# 实践教程

腾讯云

【联系我们】

我们致力于为您提供个性化的售前购买咨询服务，及相应的技术售后服务，任何问题请联系 4009100100或95716。

# 文档目录

# 实践教程
# Online DDL 的技术演进与使用实践

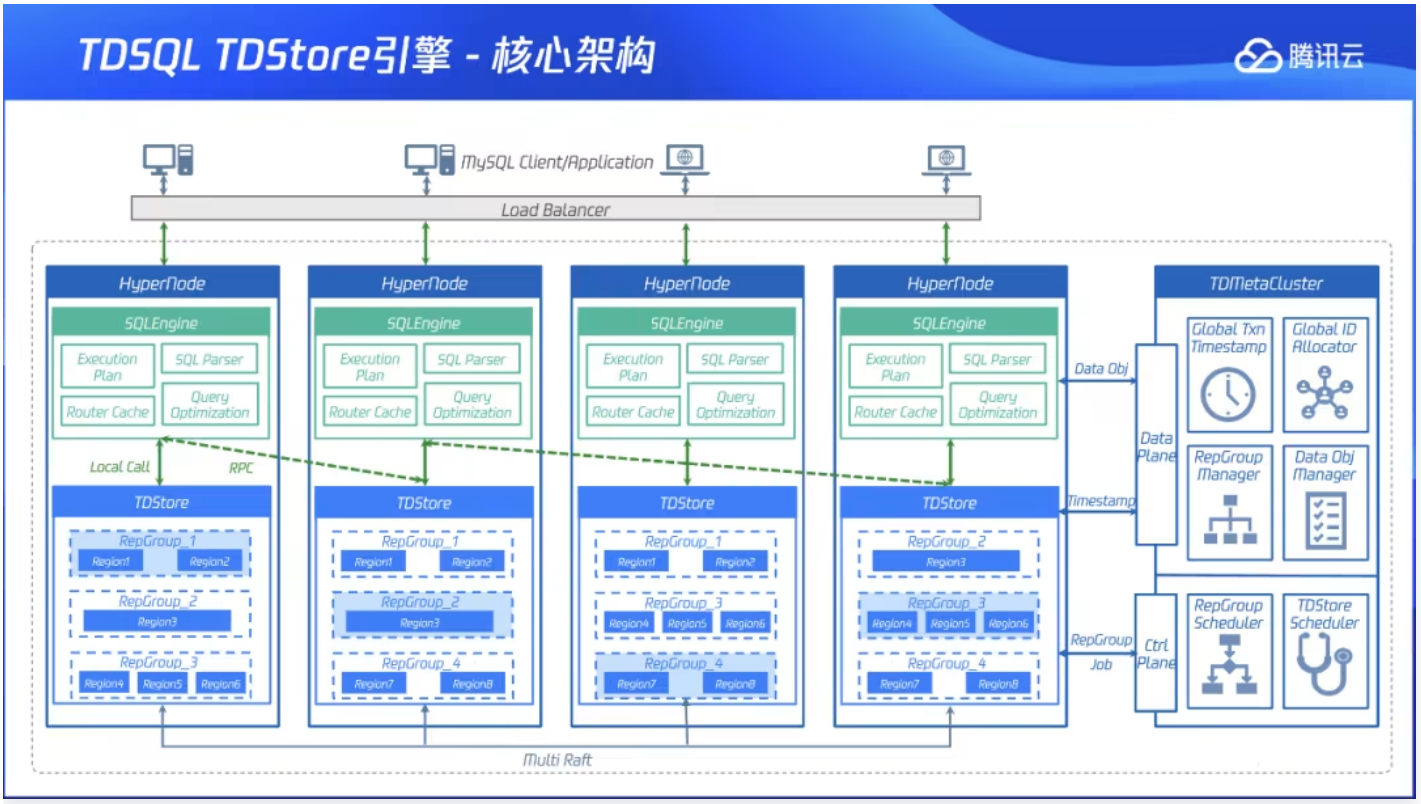
最近更新时间：2025-11-18 10:10:24

## 引言

在数据库的发展历史中，DDL (Data Definition Language) 操作一直是一个重要的挑战。在主流商业数据库的早期版本中，传统的 DDL 操作，往往都需要对数据表进行锁定，以防止在操作过程中发生数据不一致。这种锁定操作会导致数据库在 DDL 操作期间无法提供服务，对于需要7 * 24运行的大型现代应用来说，这是难以接受的。
为了解决这个问题，ORACLE 自 9i 引入 Online Table Redefinition，MySQL 自5.6引入 Online DDL，都旨在显著减少（或消除）更改数据库对象所需的应用程序停机时间（该特性以下统称 Online DDL）。随后，MySQL 8.0进一步引入了 Instant 算法，这是 DDL 操作的一个重大突破。Instant DDL 允许立即执行某些 DDL 操作，如尾部添加字段、修改字段名、修改表名等等，均无需锁定表或复制数据。这大大提高了 DDL 操作的速度，并减少了对业务的影响。尽管如此，在某些情况下 DBA 可能仍然需要依赖像 pt-online-schema-change(pt-osc) 这样的外部工具来执行 DDL 操作。这是因为 Online DDL 并不支持所有类型的 DDL 操作，同时 pt-osc 的流控等功能可以提供更多的灵活性。
随着分布式数据库的兴起，Online DDL 面临了新的挑战。在分布式环境中，DDL 操作需要在多个节点上同时进行，不仅需要保证 DDL & DML 操作并发的安全性，同时还要考虑性能、执行效率以及 crash-safe 问题。
**本文将分享腾讯云数据库 TDSQL 系列的最新产品：TDSQL Boundless 的 Online DDL 的技术演进与使用实践。**

## TDSQL Boundless 架构介绍

TDSQL Boundless 是腾讯面向金融级应用场景的，高性能、高可用的企业级分布式数据库解决方案，采用容器化云原生架构，提供集群高性能计算能力和低成本海量存储。
下图是 TDSQL Boundless 的架构图，可以看到整个实例分为两类节点：




- **右侧的 TDMetaCluster 是管控节点**：负责集群的元数据管理、智能化调度。
- **左侧的 HyperNode 是对等节点，或者叫做混合节点**：每个节点分为计算层 SQLEngine 和存储层 TDStore 两个部分，其中 SQLEngine 层负责 SQL 解析、查询计划生成、查询优化等，它向下层的 TDStore 发送事务相关的读写请求。

**TDSQL Boundless 架构和功能特性：全分布式 + 存算一体/存算分离 + 数据面/控制面分离 + 高可扩展 + 全局一致性 + 高压缩率。**

# TDSQL Boundless Online DDL 的难题攻克

**当下传统单机 MySQL 执行 DDL 的思路：**

1. instant DDL（ALGORITHM=INSTANT）：只需修改数据字典中的元数据，无需拷贝数据也无需重建表，原表数据不受影响。对此类 DDL 语句，可直接执行，瞬间完成。
2. inplace DDL（ALGORITHM=INPLACE）：存储引擎层"就地"重建表，虽然不阻塞 DML 操作，但对于大表变更可能导致长时间的主从不一致。对此类 DDL 语句，如果想使 DDL 过程更加可控，且对从库延迟比较敏感，建议使用第三方在线改表工具 pt-osc 完成。
3. copy DDL（ALGORITHM=COPY）：如"修改列类型"、"修改表字符集"操作，只支持 table copy，会阻塞 DML 操作，不属于 Online DDL。对此类 DDL 语句，建议使用第三方在线改表工具 pt-osc 完成。

**总体来看，传统单机 MySQL 除了 instant DDL 外，主流仍是采用第三方在线改表工具来执行 DDL 操作。然而，这种方法有一些明显的缺点：**

1. DDL 可能因大型事务或长查询而无法获得锁，导致持续等待和重复失败。
2. 所有第三方工具都需要重建整个表，为了确保稳定性而大幅牺牲性能。经测试表明，与原生 DDL 执行模式（INSTANT / INPLACE / COPY）相比，第三方工具执行 DDL 的性能下降了10倍甚至几个数量级，考虑到

当前不断快速增长的数据量，这是难以接受的。

**相比传统单机 MySQL，在分布式数据库中执行 DDL 将面临更多、更复杂的挑战：**

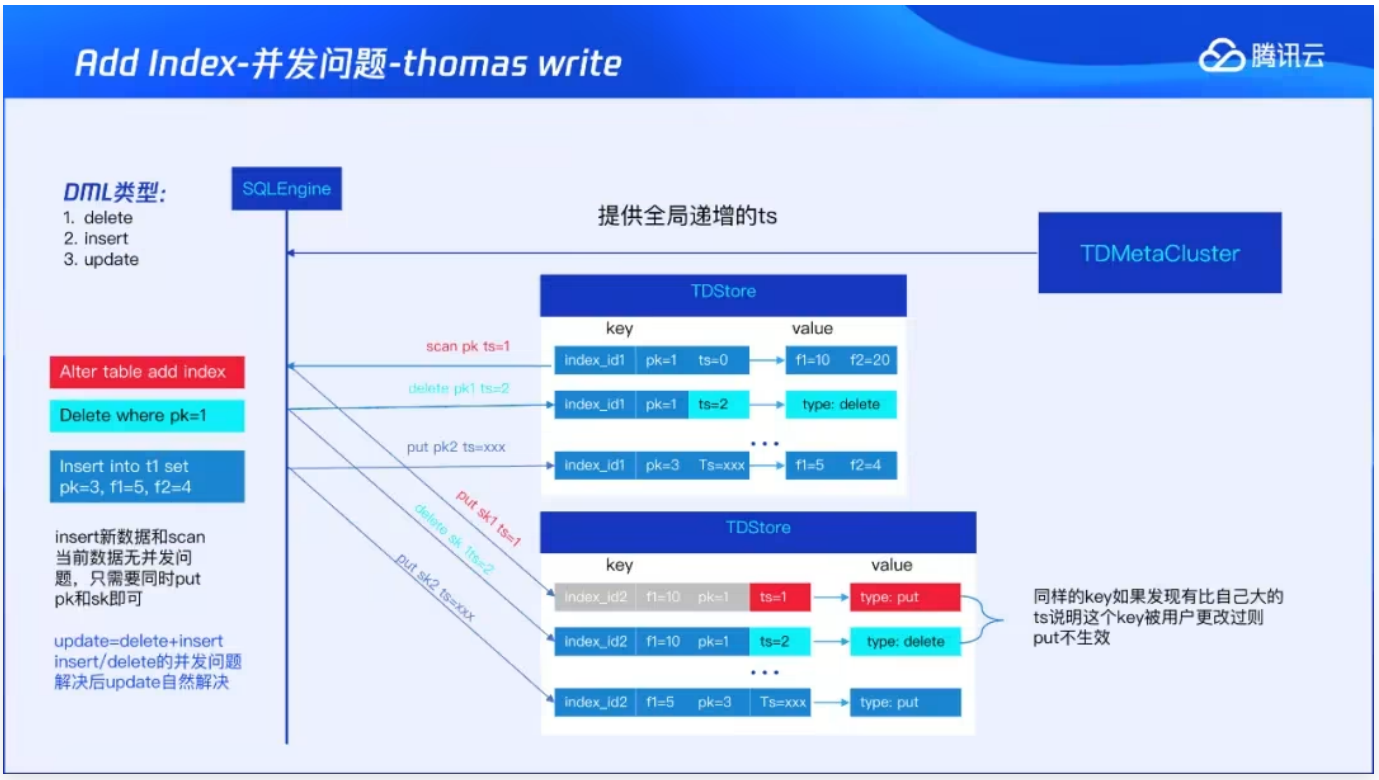
1. 原生的 instant DDL 极速执行特性是否兼容和保留？
2. 如何解决需要数据回填的 DDL 同 DML 之间的并发控制问题，且同时兼顾执行效率？
3. 原生分布式数据库大多是存算分离架构，每个计算节点之间是弱关联关系（无状态），当在某一个计算节点上执行 DDL 变更的时候，同实例中其他计算节点如何及时感知这个 DDL 的变更？同时，要在 DDL 变更过程中保证不会出现数据读写错误。
4. 原生分布式数据库中，数据被分散在多个节点上，无论是节点规模还是节点存储容量都要比单机 MySQL 大，是否可以通过存储层直接操作 + 多机并行结合的方式来大大加快需重建表的 DDL 操作？
5. 分布式数据库 DDL 的 crash-safe 问题。

**下面我们将探讨 TDSQL Boundless 如何运用一系列创新的策略来克服 Online DDL 所面临的各种挑战。**

1. 通过引入多版本 schema 解决 instant DDL 问题：

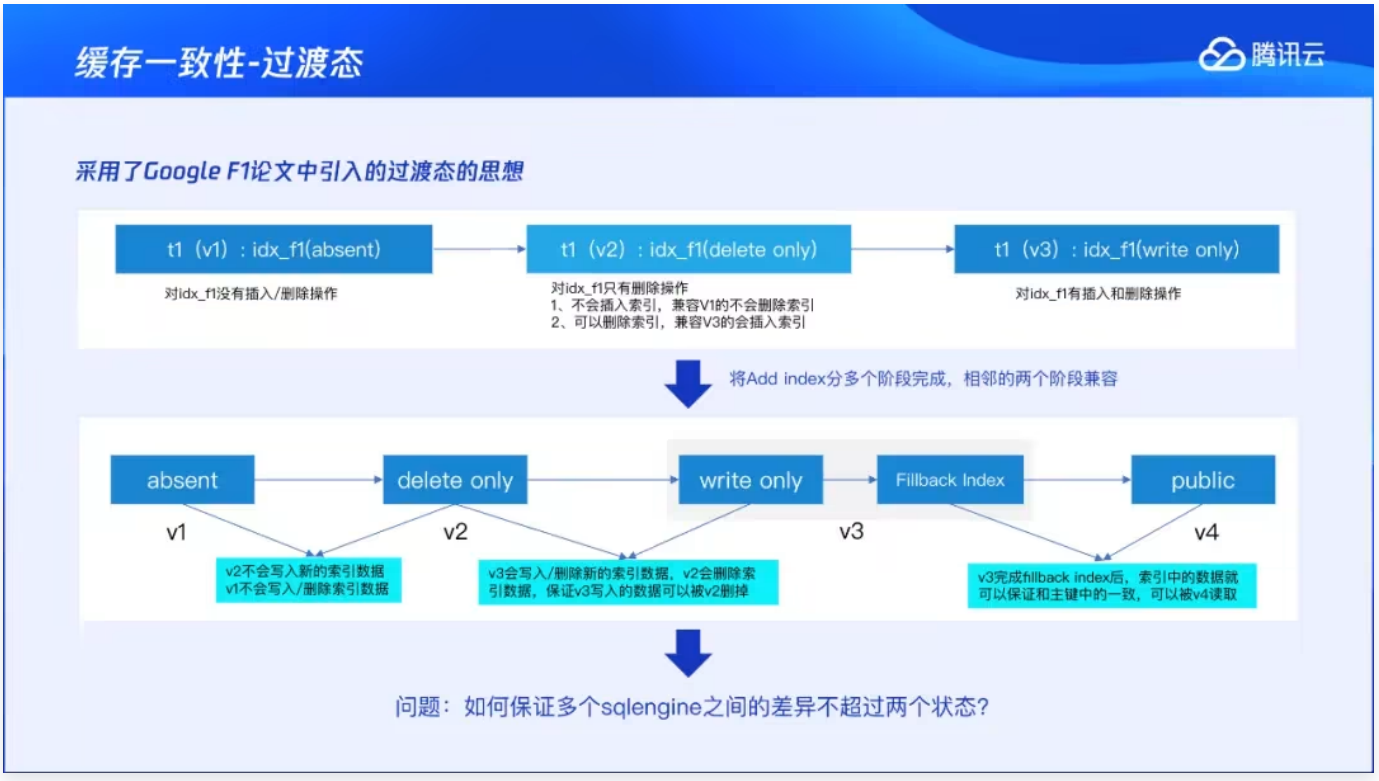

在表结构中引入版本号（schema version）概念。如上图中的 t1 表，初始版本为1，执行加列操作后，版本号变为2。新数据行在插入时会附着版本号。读取数据时需要先判断数据行的版本，如果数据行版本为2，就用当前的表结构进行解析；如果数据行版本为1，比当前版本小，确定 F2 列不在该版本的 schema 中后，直接填充默认值再返回到客户端。通过引入上述多版本 schema 解析规则，使得 Add column，varchar 扩展长度等无损类型变更只需修改元数据的 DDL 瞬间完成。

2. 通过引入托马斯写入法则（Thomas Write Rule）解决需数据回填的 DDL 同 DML 之间的并发控制问题：




通过引入托马斯写入法则（Thomas Write Rule）为 DDL & DML 操作提供并发控制机制，保证了数据库协议的序列化顺序。相比标准的时间戳并行控制方法，托马斯写入法则（Thomas Write Rule）忽略过期写，减少了交易被撤回的几率。

3. 参考 Google F1 的 schema 变更方法，引入过渡态，确保 schema 变更过程中是平滑的：




Google F1 的 schema 变更方法参考 Online, Asynchronous Schema Change in F1 。

从中可以抽象出的基本概念：当一件事情无法立即从一个状态（添加索引前）转变为另一个状态（添加索引后）时，可以尝试引入过渡性的中间状态，使相邻的两个状态兼容。这样尽管大家无法立刻进入某个状态，但只要身处兼容的两个状态下，并在不断往前转变状态的过程中依然保证所处状态的兼容性，就能按照这个模式逐步进入到最终状态，完成整个过渡。

上图以 DDL 中典型的 add index 举例，所有的 SQLEngine 从 v1 -> v4 状态变更过程中，引入了 v2、v3 两个过渡态，我们可以看到：

3.1 **v1 与 v2 共存**：v1 是 old schema，v2 是 new schema（delete-only）。此时所有的 SQLEngine 要么处于 v1 状态，要么处于 v2 状态。由于 index 只有在 delete 的时候才被操作，此时还没有 index 生成，不会有数据不一致的问题。

3.2 **v2 与 v3 共存**：v2 是 new schema（delete-only），v3 是 new schema（write-only）。此时所有的 SQLEngine 要么处于 v2 状态，要么处于 v3 状态。处于 v2 状态的 SQLEngine 可以正常插入数据、删除数据和索引（存量和新增数据都缺失索引）；处于 v3 状态的 SQLEngine 可以正常插入和删除数据和索引。但是由于 v2 和 v3 状态下，索引对于用户都是不可见的，用户能看到的只有数据，所以对用户而言还是满足数据一致性的。

3.3 **v3 与 v4 共存**：v3 是 new schema（write-only），v4 是 new schema（添加索引完成）。此时所有的 SQLEngine 要么处于 v3 状态，要么处于 v4 状态。处于 v3 状态的 SQLEngine 可以正常插入和删除数据和索引（存量数据仍缺失索引）；处于 v4 状态的 SQLEngine 已将存量数据的索引补全，是 add index 后的最终形态。 可以发现 v3 状态下，用户能看见完整的数据；v4 状态下，完整的数据和索引均能看见，因此数据也是一致的。

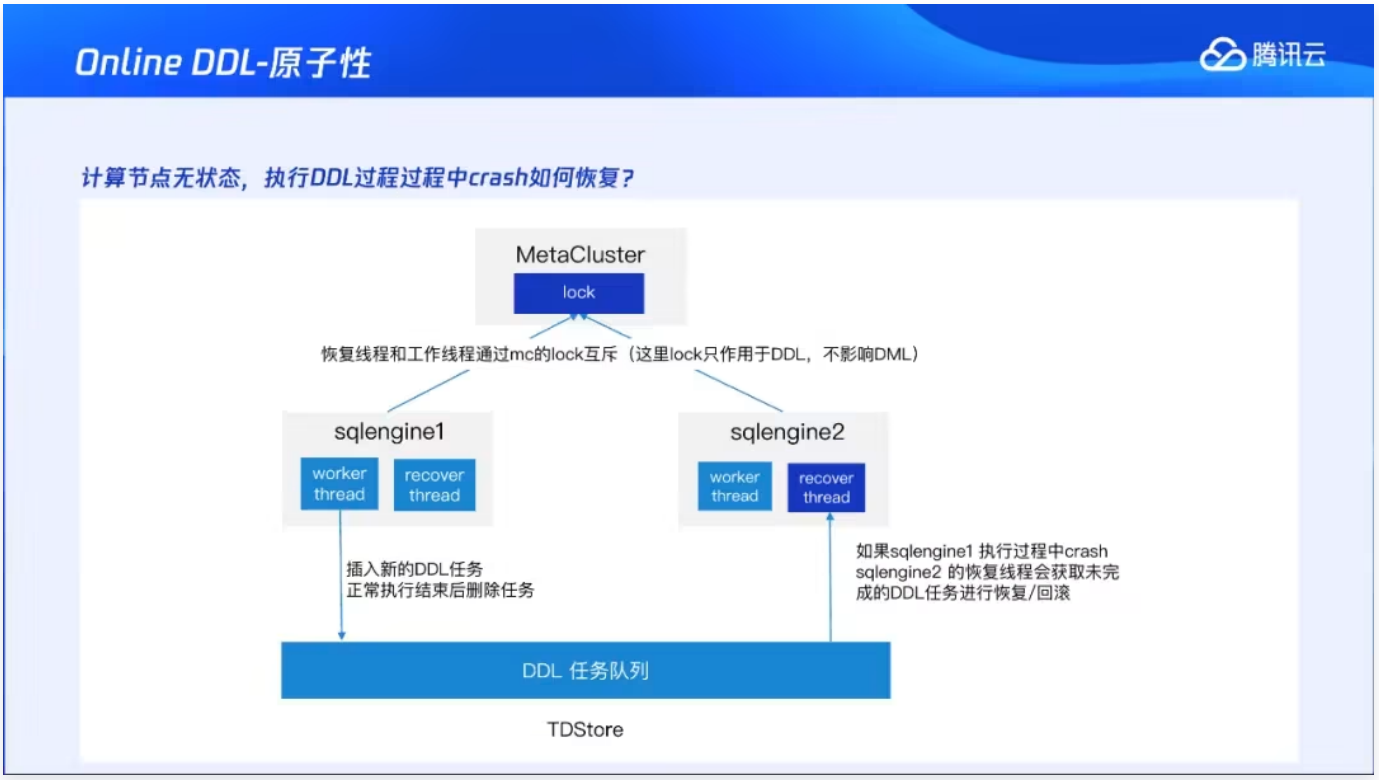
4. 结合过渡态思想，设计 Write Fence 机制：通过在存储层对请求做版本校验，保证了在任意时刻多个 SQLEngine 的有效写入只能在两个相邻的状态之间，不会产生数据不一致。

Write Fence 是 TDSQL Boundless 的内部数据结构，存储的是 schema_obj_id → schema_obj_version 的映射。Write Fence 解决的是计算层 SQLEngine 与存储层 TDStore 联动的问题，我们需要在推进 SQLEngine 状态前让 TDStore 感知到相关情况。
每个 SQLEngine 在进入下一状态前，需将当前版本推送至 TDStore 保存（push write fence）。只要 push 成功则保证两点约束：

- 当前系统中小于推送版本的请求已全部完成。
- 后续如果出现小于推送版本的请求会被拒绝。

通过存储层的版本校验机制，保证了在任意时刻多个 SQLEngine 的有效写入只能在两个相邻的状态之间，不会产生数据不一致。
上图中同时展示了即使在极端异常的场景中，假设某一节点在 push v2 版本已经成功的情况下，仍发送小于 v2 的请求，这时存储层 TDStore 就会发现该请求比当前 Write Fence 中的 v2 要小，从而拒绝执行，保证了整体算法运行的正确性。

5. DDL 的 crash-safe 问题：

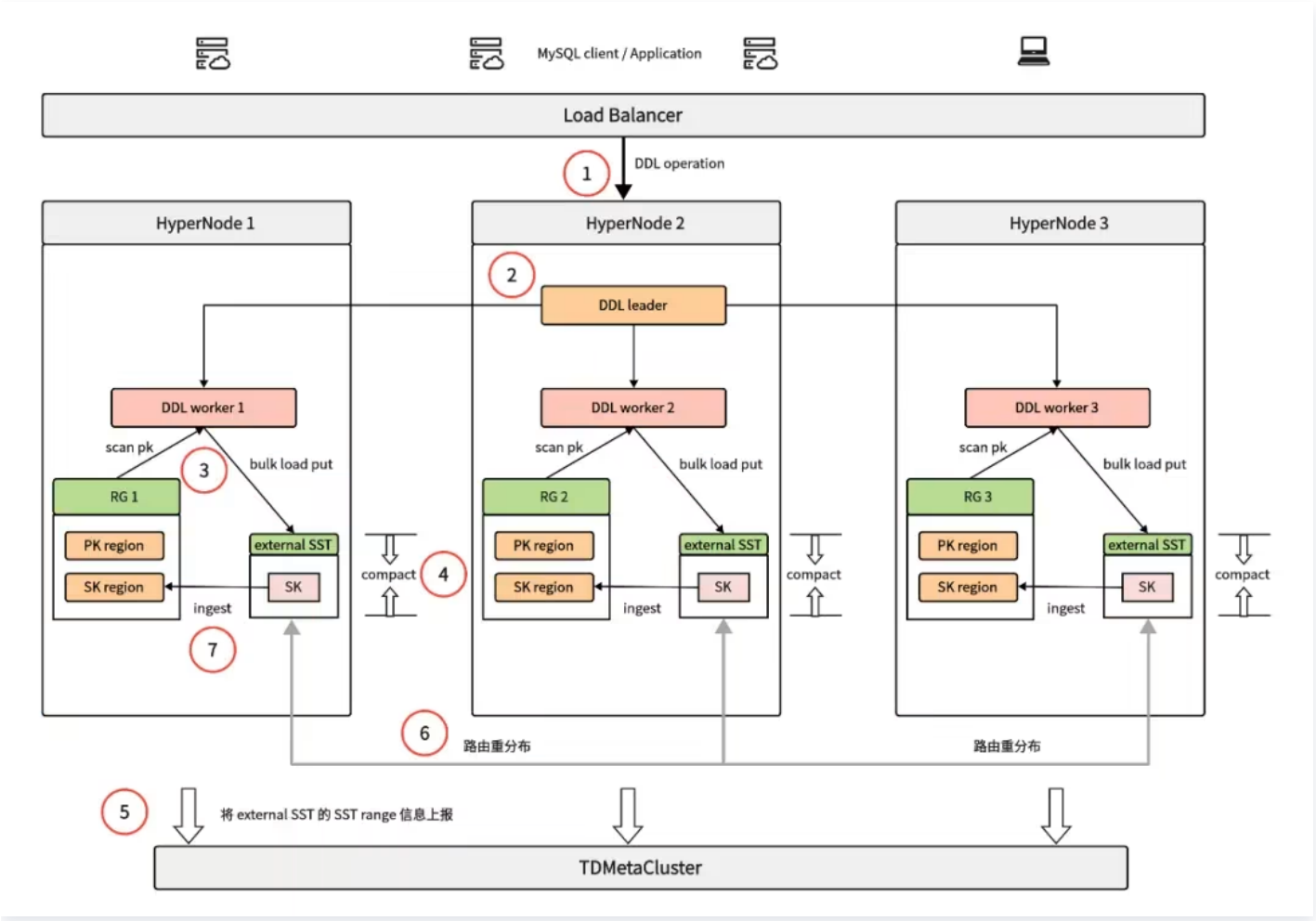



引入 DDL 任务队列 + DDL 恢复线程，保证 DDL 的 crash-safe。需要注意的是，恢复线程是独立于工作线程之外的，彼此之间不会形成等待通信。如果由于网络原因导致执行失败，恢复线程会执行接管操作（恢复线程不一定与原工作线程位于同一个节点）。监控 DDL 任务队列的方式也非常简单，可查询 `INFORMATION_SCHEMA.DDL_JOBS` 字典表，通过 `DDL_STATUS` 字段确认最终执行结果。
目前 TDSQL Boundless 已经支持大多数场景下的 Online DDL 能力，详细请参考 OnlineDDL 说明 。

# TDSQL Boundless Fast Online DDL 新特性

从上面介绍我们可以看到，TDSQL Boundless 如果在 Online DDL 中产生数据回填，采用的是托马斯写入法则（Thomas Write Rule）的方式进行批量回填，并通过并行来进行加速。但在实际使用中，特别是大数据量的场

景中，由于 Thomas Write 需要对回填数据和用户并发的 DML 数据使用时间戳方法对比以确定保留哪个版本，导致执行的时间依然比较长；因此最新版本的 TDSQL Boundless，引入了 Fast Online DDL 功能：采用 bulk load 的方式，将回填数据组成 external sst，通过 ingest 的方式直接导入 TDStore 的 Lmax 层，省去了时间戳对比并发控制的开销，可以极大提升执行效率。




上图是 Fast Online DDL 的架构图，我们来看一下它的工作原理：

1. 用户发起 DDL 操作请求，通过负载均衡器随机发往 TDSQL Boundless 实例中的对等节点，这里假设发往 HyperNode 2 节点。
2. 由 HyperNode 2 节点执行 DDL 操作，称为 DDL leader，它按照 DDL 所操作对象的 PK 数据分布将 worker 分配到所有相关的 HyperNode 对等节点。
3. 每个 worker scan 本地的 PK 生成索引数据，用 bulk load 方式写入 external sst 文件。
4. 所有 worker 完成 scan 所有的 PK 后，对 external sst 文件做一次压实操作（compact），保证同层 sst 文件之间 key 值不重复。
5. 将压实后的 sst range 信息上报给 TDMetaCluster（mc）管控节点，mc 根据这个信息将这些 range 划分 region（逻辑概念，一段段的小数据范围）并分配给对应的 RG（Replication Group，逻辑概念，一个 RG 包含多个 region，是最小的多副本数据同步单元），生成新的路由信息。同时 mc 暂时关闭 SK 涉及 region 路由的分裂合并。
6. HyperNode 根据新的路由信息重新组织 external sst 文件传输到对应的 HyperNode 节点。

7. 将满足新路由的 external sst 文件直接 ingest 到 user RG 的 sst 文件的 Lmax 层，完成后 mc 放开对 SK 涉及 region 路由的分裂合并限制。

# TDSQL Boundless Fast Online DDL 实践

以下步骤将创建一张大分区表，使用 add index 的 DDL 语句来测试 Fast Online DDL 在执行性能上的提升。

## 硬件环境

| 节点类型 | 节点规格 | 节点个数 |
| --- | --- | --- |
| HyperNode | 16Core CPU/32GB Memory/增强型 SSD 云硬盘300GB | 3 |

## 数据准备

```
-- 创建分区表，分区数为节点倍数：
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;

--造数可使用 TPC 官方标准工具：TPC-H v3.0.1，下载地址：tpc.org/tpch/

--导入，其中 LOAD DATA 的数据文件目录替换成自己的目录：
#!/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

--确认记录数:
sql> select count(*) from tpchpart100g.lineitem;
+-----------+
| count(*)  |
+-----------+
| 600037902 |

--预期数据大致均匀的分布在每个节点:
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 |
+------------+-------------+------------------+----------------------+--
------------+

--若数据分布不均匀，则进行分裂和切主
ALTER INSTANCE SPLIT RG 64535 BY 'size';
ALTER INSTANCE TRANSFER LEADER RG 64535 TO 'node-three-003';
```

## Fast Online DDL 开启前后对比

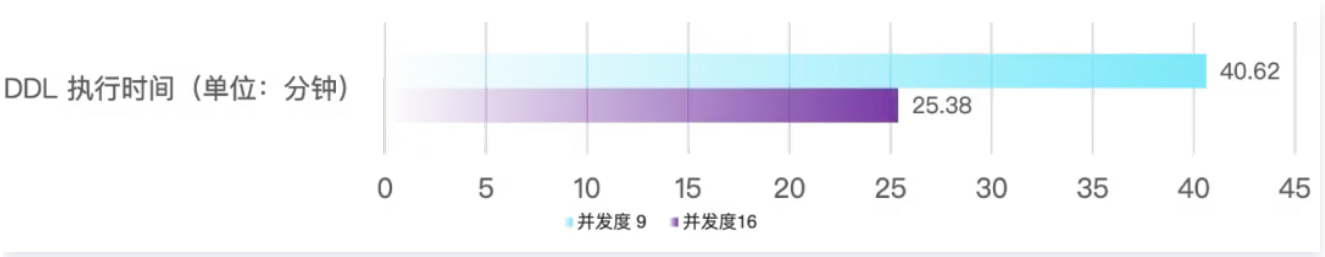
相关参数：

```
-- DDL 操作 worker 线程数（所有节点的 worker 总和），缺省值为 8：
max_parallel_ddl_degree

-- DDL 操作的数据回填模式，缺省值为'ThomasWrite'，要开启 Fast Online DDL 功能则
需要设置成'IngestBehind'：
tdsql_ddl_fillback_mode
```

使用默认的 'ThomasWrite' 数据回填模式【该模式下 DDL 线程为单机执行】，分别开启 9，16 线程数【不超单节点 CPU 数】测试 add index：

```
-- ThomasWrite 回填模式：
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)
```

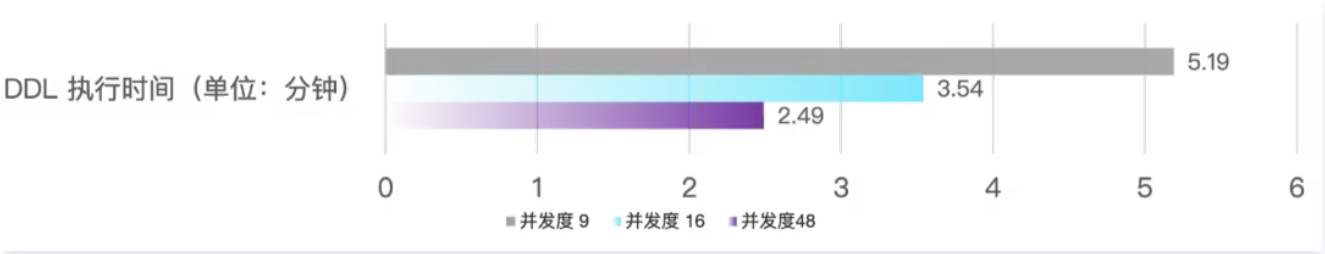
未开启 Fast Online DDL，执行时间对比：




**开启 Fast Online DDL，使用 'IngestBehind' 数据回填模式【该模式下 DDL 线程为多机执行】，分别开启 9，16，48 线程数【不超数据分布节点 CPU 总数】测试 add index：**

```
-- IngestBehind 回填模式：
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)
```

**开启 Fast Online DDL，执行时间对比：**




**DDL 执行结果查询，DDL_STATUS 字段显示最终结果：**

```
--ddl_jobs：记录ddl执行流程的字典表
```

```
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)
```

**ddl_jobs 字段详解：**

| 字段 | 说明 |
| --- | --- |
| ID | 每个 DDL JOB 都有唯一 ID。 |
| SCHEMA_NAME | 库名。 |
| TABLE_NAME | 表名。 |
| VERSION | INFO 字段解析版本号。 |
| DDL_STATUS | DDL JOB 执行状态，有 SUCCESS，FAIL，EXECUTING 三种状态。 |
| START_TIMESTAMP | DDL JOB 发起时间。 |
| LAST_TIMESTAMP | DDL JOB 结束时间。 |
| DDL_SQL | DDL 语句明细。 |
| INFO_TYPE | DDL 语句类型。 |
| INFO | 执行 DDL 的过程中的元数据信息（包含 add index, copy table 类型 DDL 语句的执行进度）。 |

# TDSQL Boundless Online DDL 的使用建议

TDSQL Boundless 的 Fast Online DDL 能力，通过并行处理和旁路写入相结合，使得 DDL 操作变得更加高效和便捷。

但如果我们没有正确地划分大/小表，或者没有根据数据规模进行适当的分区，那么 Fast Online DDL 的执行效率可能会大打折扣。这是因为一张大表，在没有进行适当分区的情况下，数据很可能都集中在单个节点上，因此 DDL 操作也会在单个节点上进行，而不是在多个节点上并行执行，这将大大降低执行效率。
只有根据数据规模合理的使用分区表，才能充分运用 Fast Online DDL 的分布式可扩展性能。
**创建分区建议：**

1. TDSQL Boundless 100% 兼容原生 MySQL 分区表语法，支持一/二级分区，主要用于解决：（1）大表的容量问题；（2）高并发访问的性能问题。
2. 大表的容量问题：如果单表大小预期未来将超过实例单节点数据盘大小，建议创建一级 hash 或 key 分区将数据均匀打散到多个节点上；如果未来数据量再增大，可通过弹性扩容的方式不断“打薄”磁盘水位。
3. 高并发访问的性能问题：高并发访问 TP 业务，如果预计单节点性能无法扛住超量读写压力的时候，也建议创建一级 hash 或 key 分区将读写压力均匀打散到多个节点上。
4. 第2、第3点中创建的分区表，建议结合业务特点选择能满足大部分核心业务查询的字段作为分区键，分区数建议为实例节点数量的倍数。
5. 如果有数据清理的需求，可创建 RANGE 分区表使用 truncate partition 命令进行快速数据清理；如果要兼顾数据打散，可进一步创建二级分区为 HASH 的分区表。

## TDSQL Boundless Online DDL 的发展

TDSQL Boundless 正在飞速发展，各项功能立足于用户的需求正在快速迭代与完善。DDL 并行性能仍在持续优化中，目前 TDSQL Boundless 的新版本着重优化了分区表在 add index 时的数据回填性能。因为我们观察到已上线的业务系统中，不少都是数 T、数十 T 的大分区表，且有需要在线添加索引的需求，因此这块能力我们进行了优先适配。在即将到来的下一个 TDSQL Boundless 版本中，我们会将 Fast Online DDL 能力进行完整呈现，带来分区表在 copy table，以及普通表在 add index、copy table 时的 Fast Online DDL 能力。
作为腾讯云数据库长期战略的核心，TDSQL Boundless 将持续以业务需求为导向，专注打磨产品，为用户提供更高效、更稳定的服务。

# 锁机制解析与问题排查实践

最近更新时间：2025-11-18 10:10:24

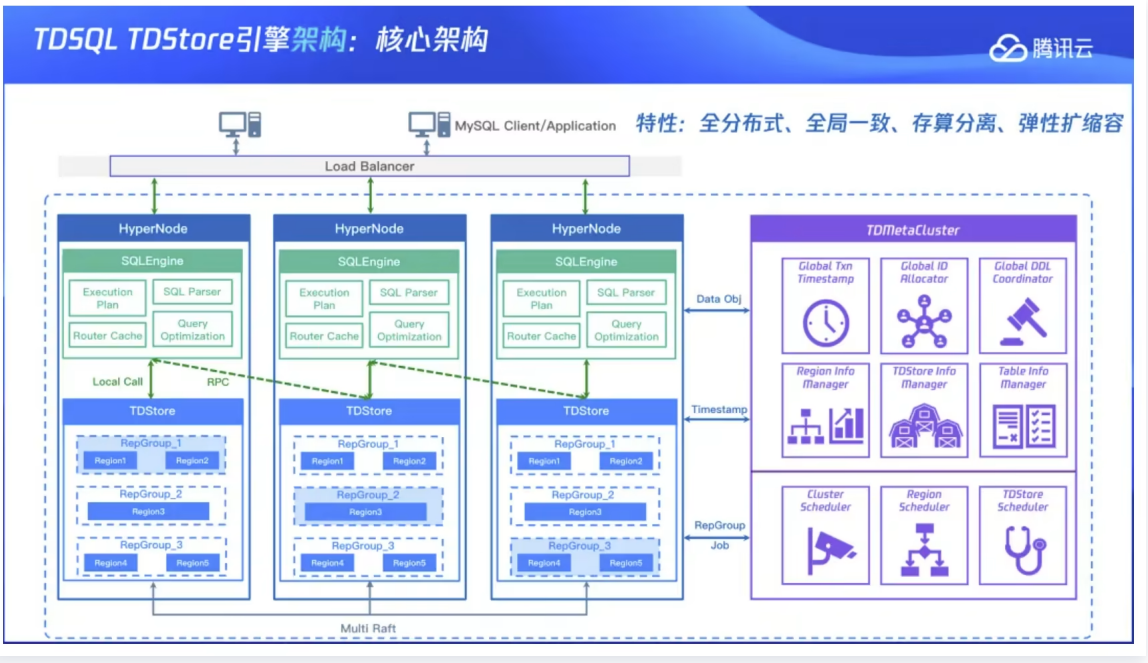
## 一、引言

锁机制是关系型数据库实现并发访问控制的核心机制之一，理解其工作原理是排查访问冲突问题的关键切入点。例如，在高并发或复杂事务处理场景中常见的锁冲突报错（如 “Lock wait timeout exceeded”），本质上反映了当前事务请求的数据资源（如行、表等）已被其他事务持有锁，当前事务在持续无法获取锁的情况下，达到锁等待超时阈值而被主动中断。

要解决锁冲突，持有锁的会话必须释放锁。让会话释放锁的最佳方式是找出长期持锁会话发起者并联系用户完成事务（提交或回滚）。紧急情况下，DBA 可以终止持有锁的会话。

本文介绍**腾讯云数据库 TDSQL 系列的最新产品——TDSQL Boundless**，结合典型问题案例，解析锁机制在事务并发中的核心作用，揭示其在保障数据一致性与提升系统吞吐量之间的平衡之道。

## 二、TDSQL Boundless 架构介绍




**TDSQL Boundless** 是腾讯面向金融级应用场景的，高性能、高可用的企业级分布式数据库解决方案，采用容器化云原生架构，提供集群高性能计算能力和低成本海量存储。

**TDSQL Boundless 架构和功能特性：全分布式 + 存算一体/存算分离 + 数据面/控制面分离 + 高可扩展 + 全局一致性 + 高压缩率。**

## 三、TDSQL Boundless 中的锁

TDSQL Boundless 作为典型的分布式系统，不仅单个节点内要有并发访问控制的机制，跨多节点仍要确保满足互斥性（Mutual Exclusion），以防止多节点同时修改同一资源导致数据不一致。

## 核心锁类型与实现层级

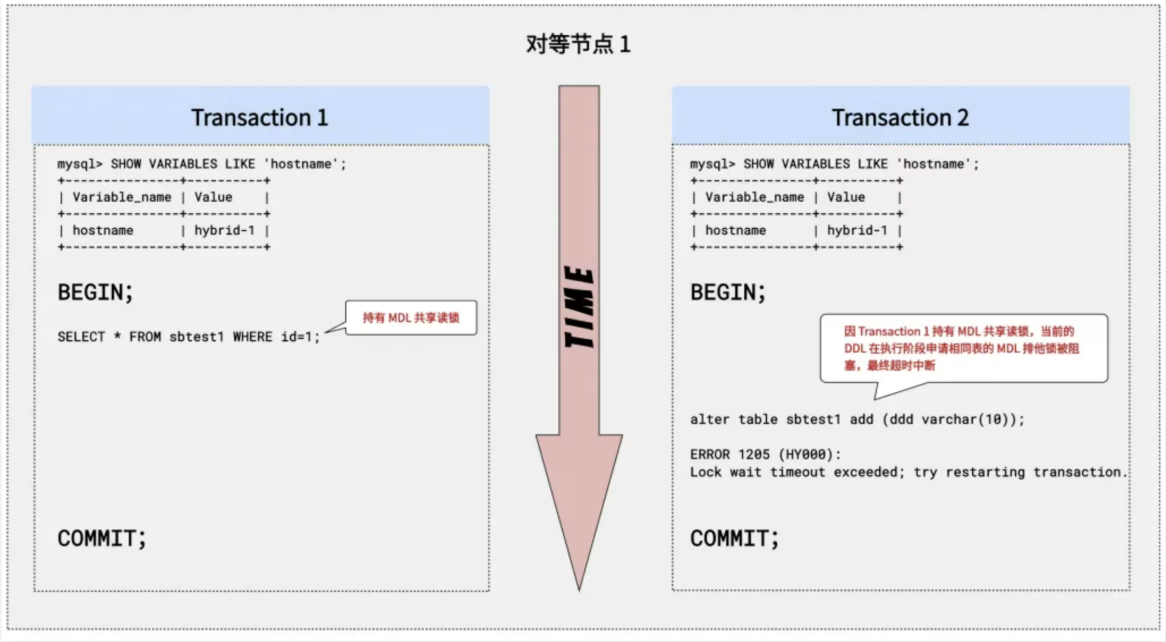
1. **表级锁（计算层）**粗粒度锁，采用 MySQL 原生的元数据锁（Metadata Lock, MDL），用于在单节点内解决 DDL/DML 之间的并发冲突问题。
2. **行级锁 & 范围锁（存储层）**细粒度锁，防止多会话并发修改同一记录，实现精准并发控制。
3. **全局对象锁（计算层申请，TDMC 层持久化）**计算层的表级锁用于在单节点内阻塞并发的 DDL 操作，而跨节点间 DDL 的协调则通过全局对象锁来发挥作用，也属于表级锁。

表级锁作用于计算层，其冲突场景可细分为同节点与跨节点两类，以下将逐一分析两种场景下的具体冲突情况。

## 表级锁冲突

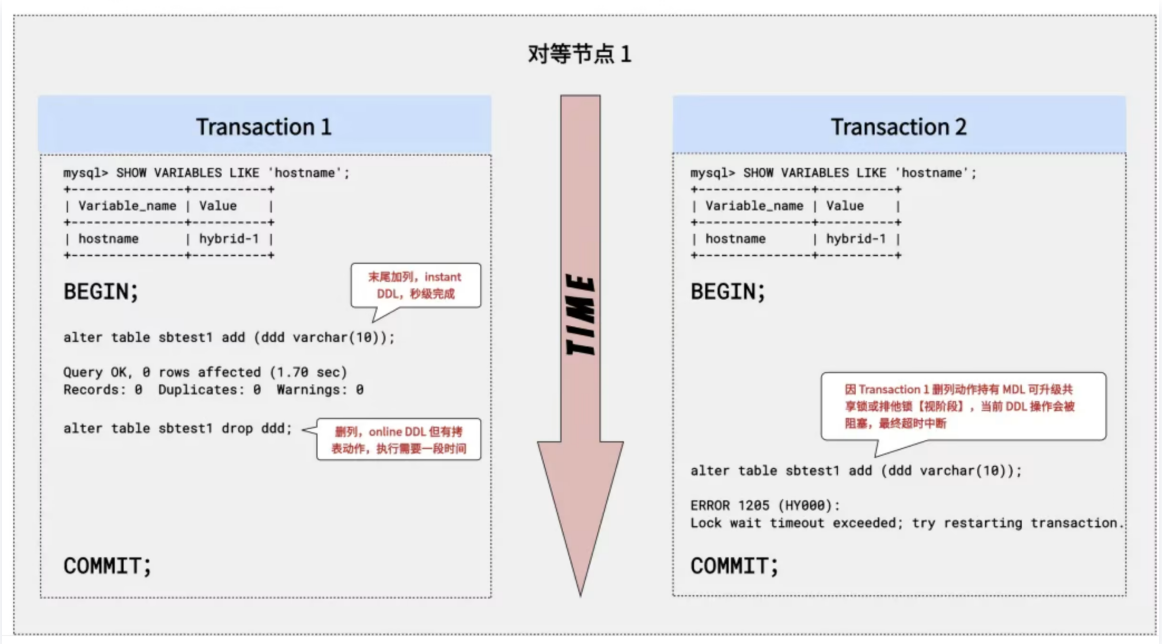
1. **同节点 DDL-DML 冲突**

   以下例子中，假设 Transaction 1 和 Transaction 2 都连到同一对等节点 hybrid-1 上。Transaction 1 显式开启事务查询表 sbtest1，在事务结束前持有该表的 MDL 共享读锁【表级锁】；在 Transaction 1 还未结束时，Transaction 2 执行 DDL 就会被阻塞，保护了表 sbtest1 的元数据。




2. **同节点 DDL-DDL 冲突**

   同样的，还是假设 Transaction 1 和 Transaction 2 都连到同一对等节点 hybrid-1 上。Transaction 1 正在表 sbtest1 进行 DDL 操作，分阶段持有该表的 MDL 共享锁和排他锁【表级锁】，避免后来的 Transaction 2 的 DDL 破坏表 sbtest1 的元数据。

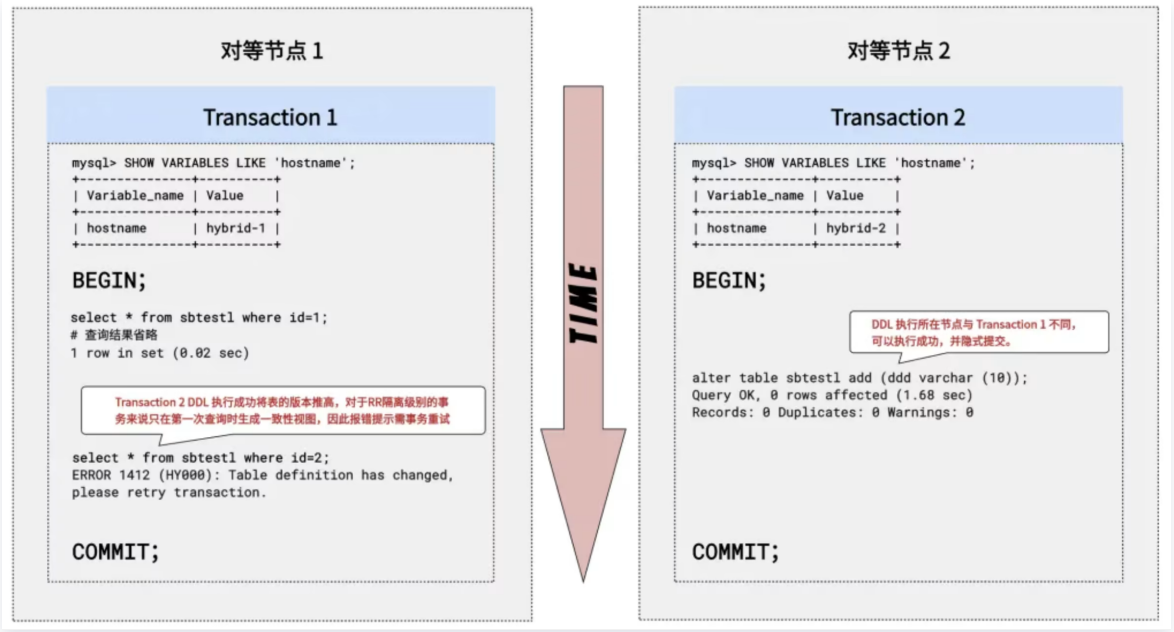



在同一对等节点上，DDL 和 DML 是平等的，谁先拿到 MDL 锁谁就执行，没有优先之分，MDL 锁是进程中的内存状态。而 TDSQL Boundless 是一个分布式数据库，会话连接进来会均匀打散在所有的节点上，如果连到两个不同对等节点的会话对同一张表执行操作，会是怎样呢?

3. **跨节点 DDL-DML 冲突**

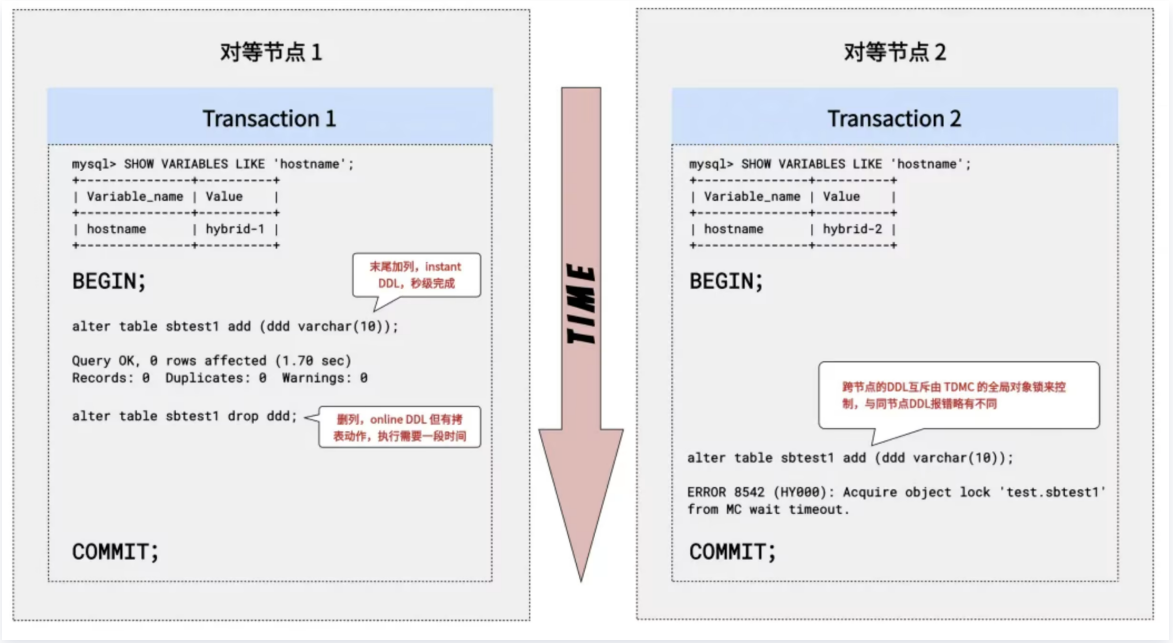
我们来看下面这个例子，Transaction 1 和 Transaction 2 分别连到对等节点 hybrid-1、hybrid-2。Transaction 1 显式开启事务查询表 sbtest1，Transaction 2 之后在另一个节点 hybrid-2 上执行 DDL 成功（因为在这个节点上没有其他会话访问表 sbtest1，不存在锁冲突），并将表 sbtest1 的 schema version 推高；

之后 Transaction 1 继续执行查询表 sbtest1 报错，因为对于Repeatable Read 隔离级别而言，只会在事务第一次查询时生成一致性视图（Consistent Read View），如果返回新版本表结构下的记录与这一点是相违背的，因此会报错 ERROR 1412 并提示事务重试（业务程序在捕获该 Exception 时应执行 rollback 并重试事务）。



### 4. 跨节点 DDL-DDL 冲突

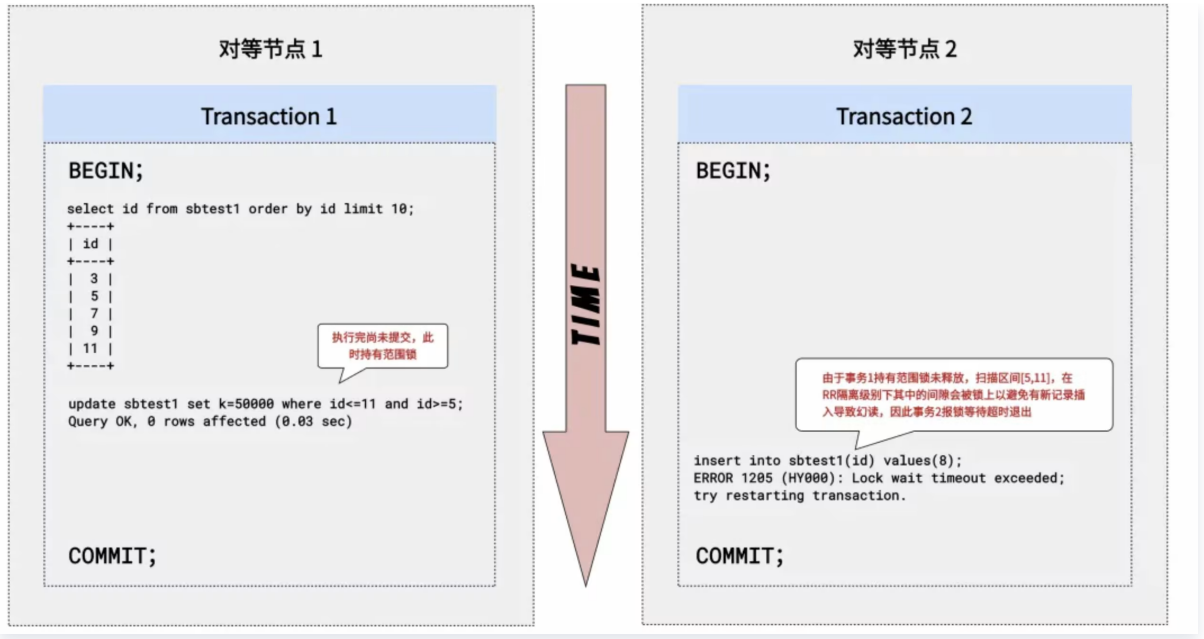
跨节点的 DDL 操作依赖于 TDMC 层的全局对象锁机制来实现操作的互斥性，确保分布式环境下的数据一致性与操作有序。
以下例子展示的还是同一张表的 DDL 相互阻塞，和之前区别的地方在于是在不同的节点上执行，这时互斥性不是由节点级的 MDL 锁来保护，而是通过 TDMC 的全局对象锁来保护。




行级锁 & 范围锁作用于存储层，当两个会话发生行锁或范围冲突，说明最终都访问了相同存储节点的主副本。因此无论其会话所连 SQLEngine 是不是同节点，都是相同的结果。以下将分析范围锁、行级锁，以及比较特殊的行级锁-死锁的具体冲突情况。

## 范围锁冲突

如果是对表的一段范围进行操作，TDSQL Boundless 存储层会加范围锁（range lock），锁的范围是左闭右开的一段 key 区间，在 Repeatable Read 隔离级别下同 Innodb 的 next-key lock 行为类似。
以下例子展示在可重复读(Repeatable Read) 隔离级别下，对 id 范围 [5, 11] 进行更新时会对 id 范围的间隙上锁，以避免有新纪录插入导致幻读。

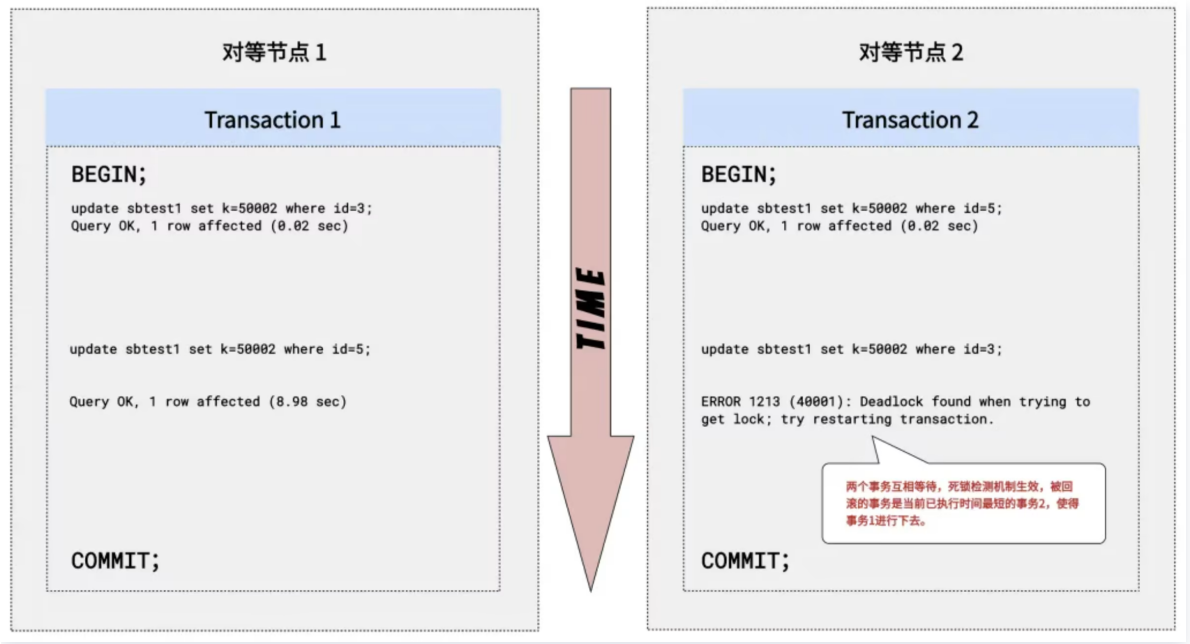



## 行级锁冲突

如果是对表的单个 key 进行操作，则加的是行级锁，TDSQL Boundless 通过精细化的锁机制将数据库的并发能力尽力最大化。
具体例子就不再列举了，相比范围锁而言只针对单行记录加锁，大家可以自行分析一下。

## 死锁冲突

死锁是行级锁冲突的一种特殊情况。当两个或多个会话等待已被对方锁定的数据时，就会发生死锁。由于双方都在等待对方，因此双方都无法完成事务来解决冲突。
TDSQL Boundless 具备自动死锁检测能力，默认回滚写数据量较少的事务并返回错误，这将释放该会话中的所有其他锁，以便另一个会话可以继续其事务。
以下例子中，Transaction 1 和 Transaction 2 互为死锁后，死锁检测机制生效，Transaction 2 被回滚，Transaction 1 得以继续执行。




### 存储层有没有表级锁?

InnoDB 提供表级别的 S 锁、X 锁，但其实非常“鸡肋”，它们并不会提供额外的保护，只会降低并发能力而已。因此 TDStore 存储层没有实现表级锁，仅支持语法解析，实际并没有起作用：

```
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.|
+---------+------+-------------------------------------------------------------
----------------------------+
```

## 四、最佳实践

锁超时和死锁是高并发数据库系统中的常见问题，有效的锁管理需要从业务逻辑、数据库配置和 troubleshooting 三个维度进行系统性优化。

### 业务逻辑

事务设计是预防锁问题的首要环节。坚持“**短事务、轻操作、顺序访问**”三大原则，可有效降低锁冲突概率。

1. 应尽量缩短事务执行时间。如单次事务操作的行数控制在2000以内。事务中包含的 SQL 语句越多、操作行数越大，持锁时间就越长，与其他事务的冲突概率就越高。
2. 避免在事务中进行用户交互，仅保留核心数据操作在事务中执行，确保事务快速完成并释放资源。
3. 确保事务中资源访问的顺序一致性。当**多个事务需要访问多个资源时，必须按统一顺序访问这些资源**，这是预防死锁的关键。例如，在库存扣减和订单创建的事务中，所有事务都应先锁定库存表再锁定订单表，或者相反，但绝不能有的事务先锁库存后锁订单，有的事务顺序相反，避免形成循环等待。
4. TDSQL Boundless 已经支持大多数场景下的 Online DDL 能力，但 DDL 操作发起前仍建议检查下，参考 OnlineDDL 说明，另建议在业务低峰期操作。

## 数据库配置

其次，针对生产系统在发生锁等待超时场景时，应优先核查相关参数的配置是否合理。针对由旧版本升级上来的实例，需特别关注后续版本新增的锁控制参数是否因兼容性需求保持关闭，必要时手动启用。
数据库相关参数：

1. **tdsql_lock_wait_timeout**：控制锁等待的最大时间，默认值为50秒。在上述图例中的场景演示中，被阻塞的会话如果在50秒内仍没法拿到锁，就会报错“Lock wait timeout exceeded”。一般无需调整默认值，如果业务有比较严重的锁冲突，无法迅速解决则可以适当调低该值来应急。
2. **tdstore_deadlock_detect（需超级管理员权限）**：死锁自动检测功能。新购实例默认值为ON，建议保持开启状态。从老版本升级上来的实例则默认关闭，如有需要可手工开启。
3. **tdstore_deadlock_victim（需超级管理员权限）**：当死锁发生时，决定选取哪个事务回滚【在 tdstore_deadlock_detect 死锁检测功能开启时，该参数才生效】。默认值为 “WRITE_LEAST” ，与 Innodb 行为保持一致。一般无需调整默认值，当设置为 “WRITE_LEAST” 时，优先选取写数据量较少的事务回滚；当设置为 “START_LATEST” 时，优先选取较晚开启的事务回滚。

## troubleshooting

第三章节通过案例对比直观呈现了 TDSQL Boundless 在节点内与跨节点场景下的 DDL/DML 冲突处理机制。实际业务场景中的报错就分为两类：DDL 超时失败或 DML 超时失败。通过以下步骤进行排查与处理。

> **说明：**
> 以下用到的字典表定义参见 系统表和系统视图 。

1. **DDL 超时失败：**

   如果 DDL 执行报错“Lock wait timeout exceeded”，则表明执行 DDL 的会话被同节点的 DML 或 DDL 阻塞；如果 DDL 执行报错ERROR 8542 (HY000): Acquire object lock 'test.sbtest1' from MC wait timeout，sql-node: node-tdsql3-xxxxxxxx-xxx，则表明执行 DDL 的会话被其他节点上的 DDL 阻塞。
   可以通过查询 performance_schema.metadata_locks 查看当前节点 SQLEngine MDL 锁的占用情况【TDSQL Boundless 中不需要将 performance_schema 系统变量设置为 ON】

```
# 确认 session1、session2 连在同一节点
# 如果不是连在同一节点上，ddl是可以执行成功的，参考“跨节点 DDL-DML 冲突”

show variables like'hostname';

#session1
BEGIN;
UPDATE sbtest1 SET k =0WHERE id =999;

#session2
ALTERTABLE sbtest1 ADDCOLUMN new_column VARCHAR(255);

#查看 metadata_locks，可以看到：
#第一行LOCK_STATUS=GRANTED的记录正是 session1，表示已获得 MDL 锁；
#第二行LOCK_STATUS=PENDING的记录正是 session2，表示获取 MDL 锁被挂起。
#需要在开头加上 broadcast HINT 指定在所有节点广播查询。

/*#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)

#当查询多次都发现 LOCK_STATUS: GRANTED 的会话一直没有变化时，可通过
OWNER_THREAD_ID 定位到持有锁的线程所对应的 SESSION ID，在确认该会话可安全终止
后，先通过 KILL 命令结束该会话，再重新发起 DDL 操作。
/*#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 持锁者（LOCK_STATUS: GRANTED 会话）
#session1 被杀

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

2. **DML 超时失败：**

如果 DML 执行报错“Lock wait timeout exceeded”，基本上是获取不到行级锁或是遇到了死锁。针对一路升级上来的老实例，首先建议检查死锁自动检测功能，如果是未打开状态，建议开启（无需重启），之后如再遇到死锁系统会自动解开。
如仍然有锁超时问题，使用如下方法找到识别持有锁（Lock Holder）的会话 ID，通过 KILL 终止阻塞会话，释放行锁资源，使等待会话（Lock Waiter）得以继续执行。
可以通过查询 performance_schema.data_locks、performance_schema.data_lock_waits 查看 TDStore 持有锁和等待锁的会话信息【TDStore 中不需要将 performance_schema 系统变量设置为 ON】

```
# 这里 session1、session2 无论是否连到同一节点，结果都是一样的；因为行锁是存储层的，两个会话发生行锁冲突，说明最终都访问了相同存储节点的主副本

#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);
```

```
#查询当前节点上，持锁者（Lock Holder）的悲观锁信息。
#TDStore 的 range lock 左闭右开，可以看到列 ENGINE_LOCK_ID 中显示为 [5,12)
区间

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)

#查看持锁者（Lock Holder）的会话信息。
# !!! TDStore 的表 PERFORMANCE_SCHEMA.DATA_LOCKS 中的 thread_id 列指的是
processlist_id；而官方 mysql 中，thread_id 指的是
Performance_Schema.threads 里的 thread_id。该问题会在之后的版本中修正。
```

```
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)


#KILL 持锁者（Lock Holder）后，阻塞会话执行成功。
#session1 被杀

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

#注意：高并发场景下，可能等待会话（Lock Waiter）队列比较长，这样可能又会再次出现
持锁者（Lock Holder），可能需要多杀几次。
```

# 数据智能调度及相关性能优化实践

最近更新时间：2025-11-18 10:10:24

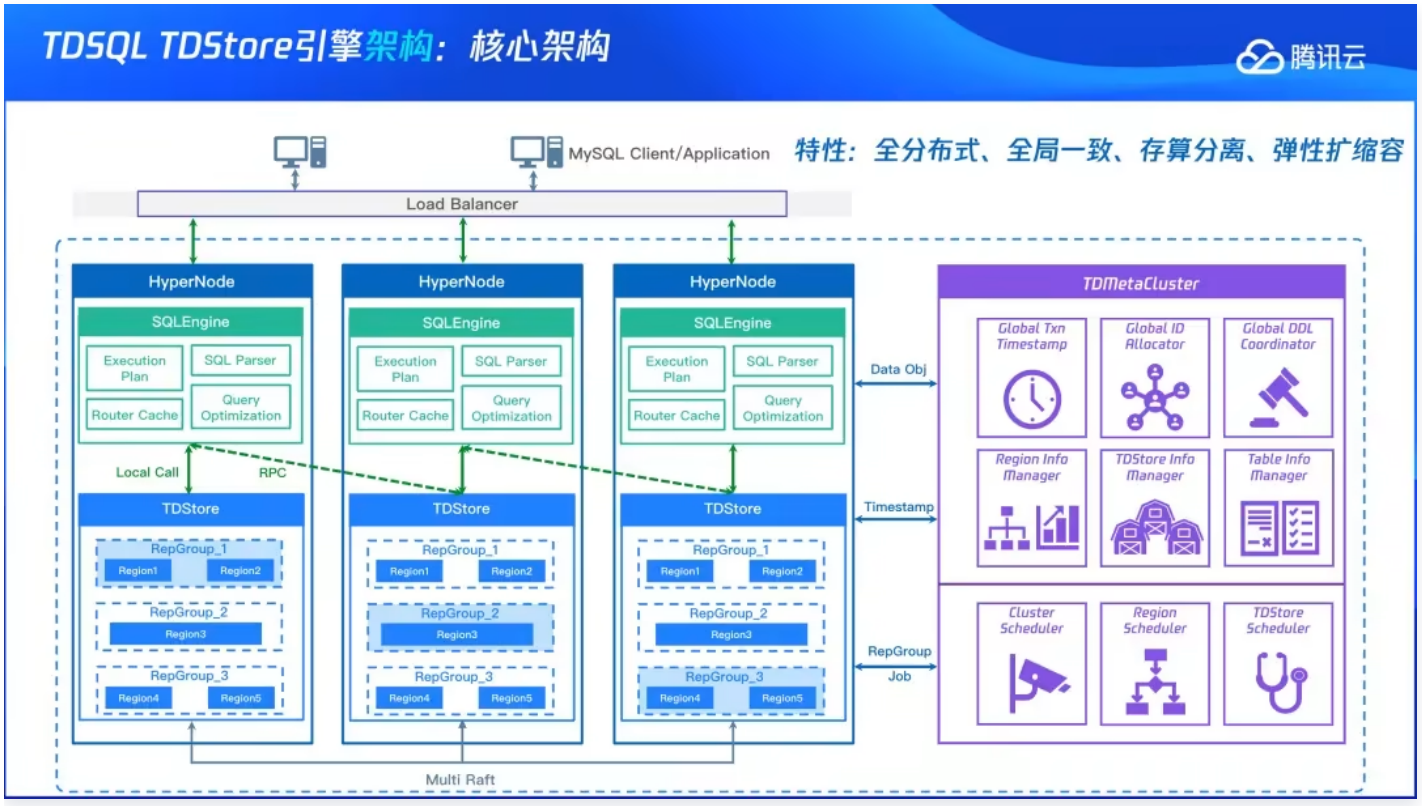
## 引言

历经多年技术演进，分布式数据库已突破传统"分库分表"的单一场景局限，从应对海量数据的权宜之策进化为支撑业务弹性扩展的核心架构。分布式数据库 TDSQL Boundless 通过构建集中分布式一体化能力（Integrated Centralized-Distributed Solution），实现了数据库运行模式与业务规模的动态适配：在业务初期，系统以完全兼容集中式数据库的形态运行，保障开发习惯与运维体系的延续性；当业务进入高速增长阶段，仅需配置级调整即可无缝切换至分布式模式，通过在线弹性扩缩容应对流量洪峰与数据膨胀。整个过程实现了两大突破：

1. **零改造平滑演进**：业务逻辑层无需适配分布式事务逻辑，数据层通过智能路由保持访问一致性。
2. **无感知架构切换**：从集中式到分布式的升级过程保持业务连续性，规避传统方案耗时的数据迁移与停服风险。

**分布式数据库 TDSQL Boundless 的集中分布式一体化能力主要依赖数据智能调度技术。**数据智能调度技术不仅实现了集中分布式一体化能力，更在分布式场景下创新性地引入数据亲和性优化机制，通过预定义规则与定制化策略结合，将具有强业务耦合的数据单元智能调度至同一物理节点，从根本上规避跨节点RPC访问的性能损耗。本文将分享腾讯云数据库 TDSQL 系列的最新产品：分布式数据库 TDSQL Boundless 的数据智能调度及相关性能优化实践。

## TDSQL Boundless 数据智能调度组件 TDMetaCluster




TDSQL Boundless 是腾讯面向金融级应用场景的，高性能、高可用的企业级分布式数据库解决方案，采用容器化云原生架构，提供集群高性能计算能力和低成本海量存储。

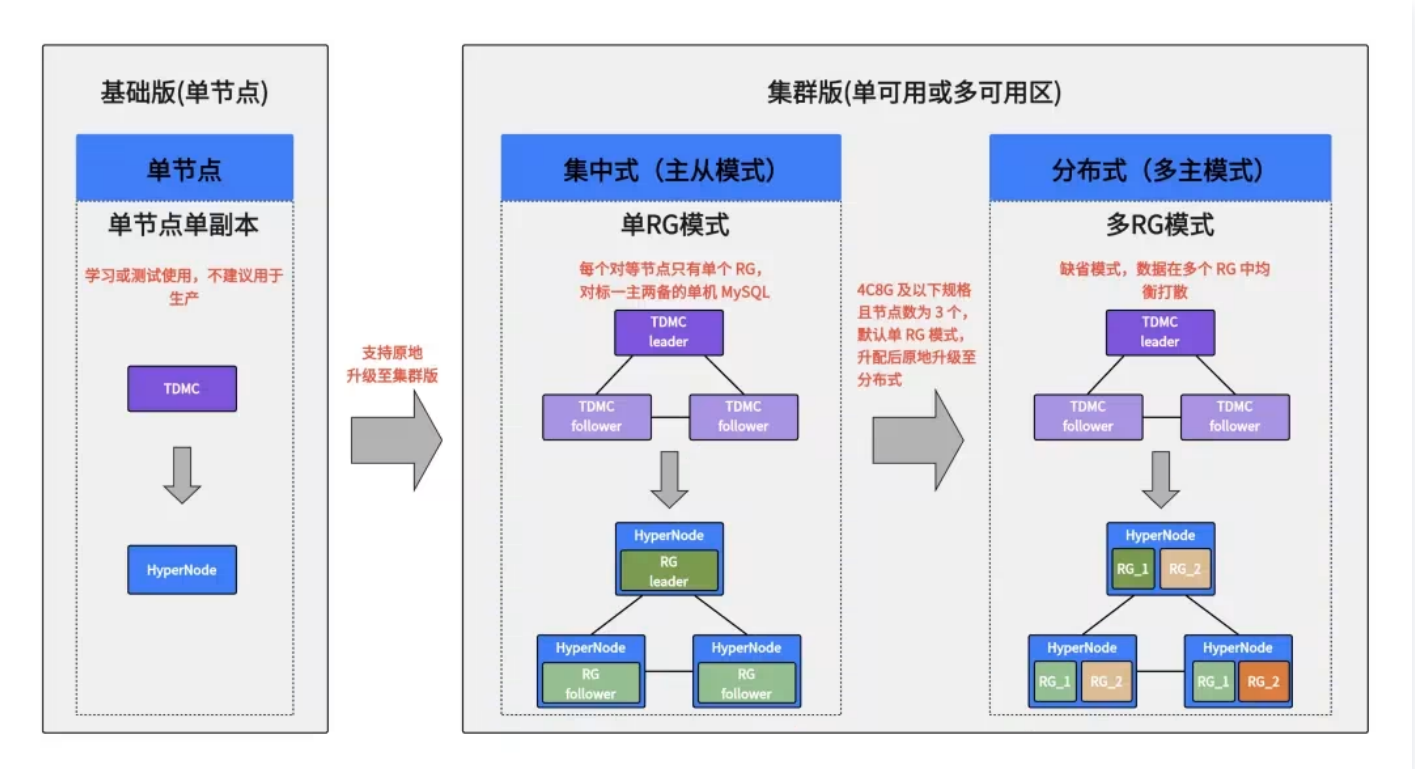
**TDSQL Boundless 架构和功能特性：全分布式 + 存算一体/存算分离 + 数据面/控制面分离 + 高可扩展 + 全局一致性 + 高压缩率。**

**在 TDSQL Boundless 中负责数据智能调度的正是组件 TDMetaCluster（上图右侧，以下简称 TDMC），它是 TDSQL Boundless 数据库实例的中心管控模块。**

1. **TDMC 的架构：**基于 Raft 协议的一主两备的元数据管理服务，其中 Leader 节点负责处理请求。
2. **TDMC 的数据面：**
   2.1 分配全局唯一且递增的事务ID。
   2.2 管理 TDStore 和 SQLEngine 的元数据。
   2.3 管理 Replication Group 数据路由信息。
   2.4 全局 MDL 锁管理。
3. **TDMC 的管控面：**
   3.1 调度 Replication Group 和 Region 的分裂、合并、迁移、切主。
   3.2 存储层的扩缩容调度。
   3.3 存储层的负载均衡调度。
   3.4 各维度的异常事件告警。

作为 TDSQL Boundless 实例的核心调度引擎，它不仅可以通过预定义规则管理数据副本的组织与分布，还支持用户根据业务需求灵活配置定制化策略。接下来，我们将探讨如何基于业务场景选择合适的部署模式：集中式或分布式模式。尽管 TDSQL Boundless 作为原生分布式数据库具备强大的分布式处理能力，但在特定场景下，集中式模式可能更契合实际应用需求。

## TDSQL Boundless 集中分布式一体化



首先，基于业务规模评估：包括数据量、访问频率、响应时间及扩展性需求，选择匹配的 TDSQL Boundless 规格。即使初期估算存在偏差也无需担忧，TDSQL Boundless 凭借其高效弹性伸缩能力，可灵活适配各类动态变化的敏态业务场景。
参考上图所示，部署模式选择如下：

1. **体验学习**：选择"基础版"【单节点单副本】，无高可用特性。
2. **生产环境**：选择"集群版"【基于 Raft 多数派协议的三副本架构】。
3. **高级容灾**：集群版支持"单可用区"和"多可用区"部署，后者可实现同地域三可用区容灾。
4. **小型业务**：4C/8G及以下规格且节点数为3时，系统自动启用集中式模式（单 RG 模式），主副本集中存储，兼容单机 MySQL 使用体验。
5. **中大型业务**：4C/8G以上规格或节点数超过3时，系统自动切换为分布式模式（多 RG 模式），数据均衡分布，多节点对等读写，确保高性能与高可用。

建议结合实际业务进行功能与性能测试，以验证配置是否满足峰值需求。
在了解如何选择 TDSQL Boundless 配置后，您可能对集中式模式已无疑问，但仍会担忧分布式架构是否影响性能。这种顾虑确实存在，而数据库智能调度技术正是解决此类性能问题的关键。在深入探讨前，需先理解分布式系统的一个关键设计：数据均衡——它是保障系统性能与可靠性的基石。接下来，我们将详细解析数据均衡的原理及其在 TDSQL Boundless 中的实践应用。

# TDSQL Boundless 数据均衡

数据均衡是分布式系统的一个关键特性，涵盖**容量均衡**与**热点均衡**两大维度：容量均衡通过均匀数据分布，优化全局资源利用率；而热点均衡则致力于解决局部高负载问题。

## TDSQL Boundless 的技术亮点

1. **容量均衡**

   1.1 **三层存储模型与资源预分配**：

   基于三层存储模型设计（参考 产品概述），每一个节点都预创建一个主 RG，默认可以容纳100张表的 Region，超限后才会自动创建新 RG，减少资源碎片化，降低管理开销并提升性能。

   1.2 **双维度分裂控制**：

   - **Region级**：256M/500万 Key 触发分裂，保持原 RG 归属。
   - **RG级**：32G/1.6亿 Key 触发分裂，单表 RG 超节点容量20%才分裂，避免分布式事务；分裂后以 RG 为单元自动迁移，维持容量均衡。

   1.3 **迁移优先级策略**：

   优先迁移备副本。仅在节点中存在主副本时，才进行主副本的迁移，确保业务无感知。

   1.4 **弹性扩缩容联动**：

   自动触发数据再均衡以适应节点数变化。

2. **热点均衡**

   2.1 **智能决策引擎**：

采用加权移动平均算法实时采集节点/ RG /数据对象三级流量指标，逐级判定热点源并执行差异化调度，如跨节点切主或分裂+切主组合操作。

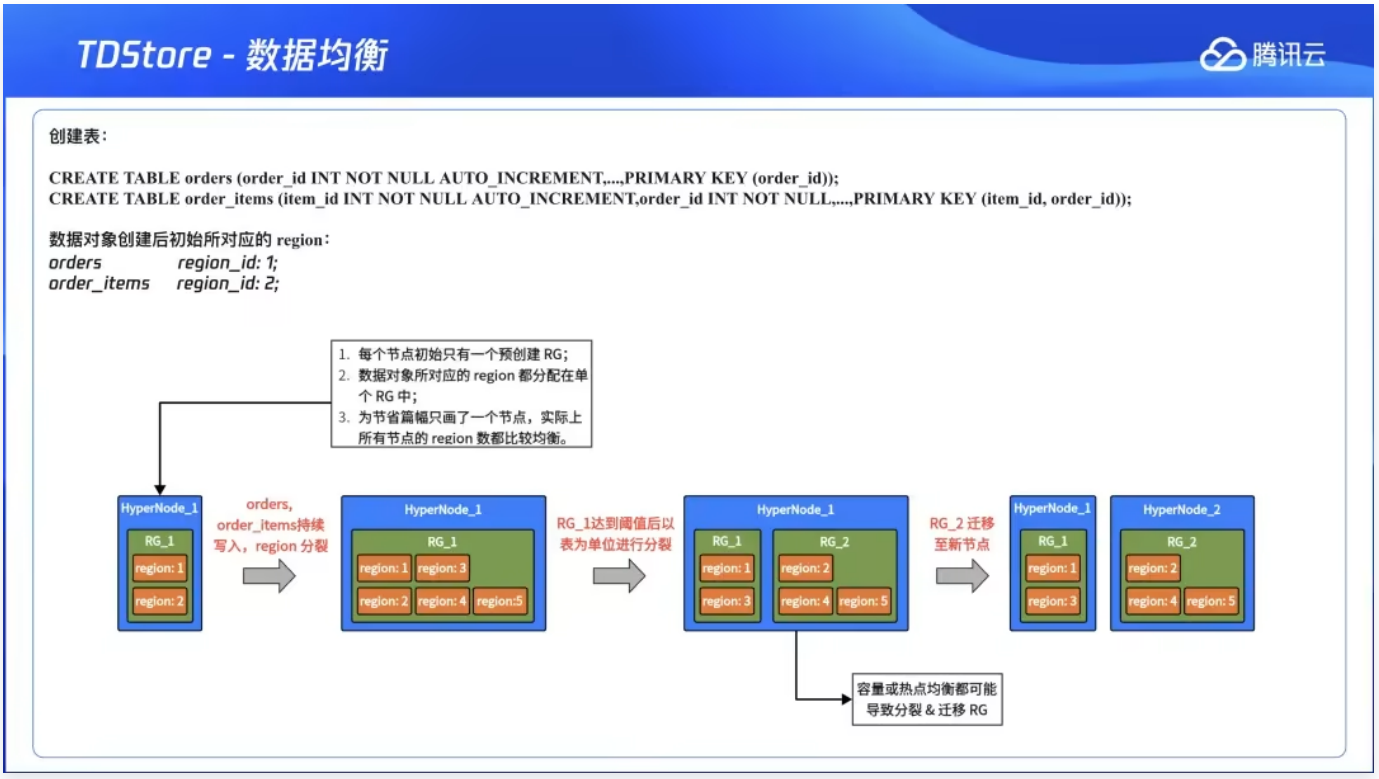
**稳定性保障机制：**

**调度冷却期**：热点 RG 调度后进入冷却状态，防止热点频繁切。

**历史调度记忆**：记录节点迁移轨迹，防止热点来回切。

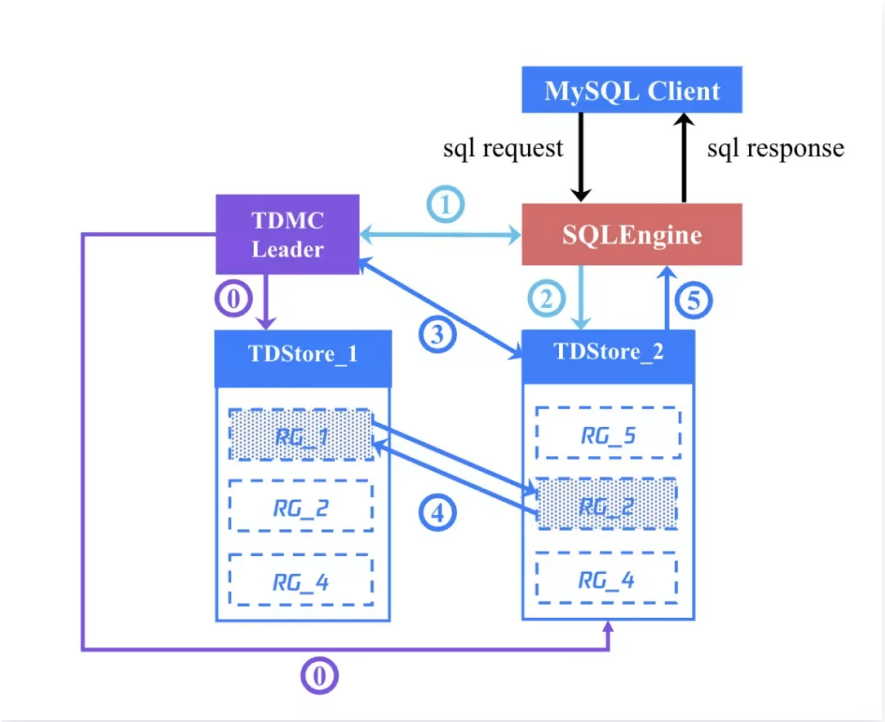
**2.2 场景化调度模式：**

支持手动配置读写热点模式，默认读热点优先，灵活适配不同业务需求。




**然而，对于分布式数据库系统而言，单纯的数据均衡并不等同于“最优”状态。如果调度模块未能充分理解存储层的库表逻辑含义，系统仍可能遭遇性能瓶颈。**

为了更好地说明这一点，我们考察一个典型的分布式应用场景：当应用程序客户端向 TDSQL Boundless 数据库实例提交一条 SQL 语句时，TDMC（管控模块）如何与 SQLEngine（SQL 引擎）以及 TDStore（存储引擎）协同工作以处理这一请求。




**Step 0**：TDMC 周期性向所有 TDStore 下发全局最小快照点，确保读操作的全局一致性。

**Step 1**：SQLEngine 接收到 SQL 语句，开启事务，从 TDMC 取回事务开启时间戳 `begin_ts`；在自身缓存中查询 SQL 语句对应的 RG 路由信息；若无，则将对应的 key 范围查询请求发送给 TDMC，由 TDMC 告知最新的路由信息。

**Step 2**：SQLEngine 将请求发送到对应的 RG 的 Leader 副本所在的 TDStore 上。

**Step 3**：Leader 副本作为协调者开启两阶段事务，从 TDMC 取回 prepare_ts。

**Step 4**：所有参与者 Prepare 就绪后，从 TDMC 取回 `commit_ts`，提交事务；若过程中，参与事务的其他 RG 发生切主，则从 TDMC 查询其最新的 Leader 信息。

**Step 5**：提交事务，成功后将结果返回至 SQLEngine，继而将结果返回到客户端。

**上述执行过程可能面临以下三个性能挑战：**

1. **计算与存储的分离**：如果计算节点和存储节点之间严格分离，网络延迟和带宽限制会对性能产生负面影响。数据在不同节点间的传输会增加额外的开销，从而降低整体系统的响应速度。
2. **数据对象的分散存储**：如果同一表的相关数据对象分布过于分散，例如表的数据存储在 RG_1 中，而其二级索引存储在 RG_2 中，那么即使是简单的单条事务写入操作也需要通过分布式事务来协调，这无疑增加了复杂性和处理时间。
3. **跨机器的关联查询**：当具有关联关系的多个表（如 A 表存储在 RG_1 中，B 表存储在 RG_2 中）需要进行 JOIN 操作或参与分布式事务时，这些操作将不可避免地涉及跨机器通信，进一步加剧了延迟和资源消耗。

针对第一个挑战，TDSQL Boundless 采用计算存储一体化架构，通过 SQLEngine 与 TDStore 物理绑定形成对等节点，提升本地化数据访问性能（函数调用替代 RPC）。至于其余两个挑战，则通过 TDSQL Boundless 的智能调度机制来解决。接下来，我们将具体探讨 TDSQL Boundless 是如何应对这些挑战的。

# TDSQL Boundless 数据智能调度

**TDSQL Boundless 通过多级规则体系实现数据亲和性调度。包括三类规则：**

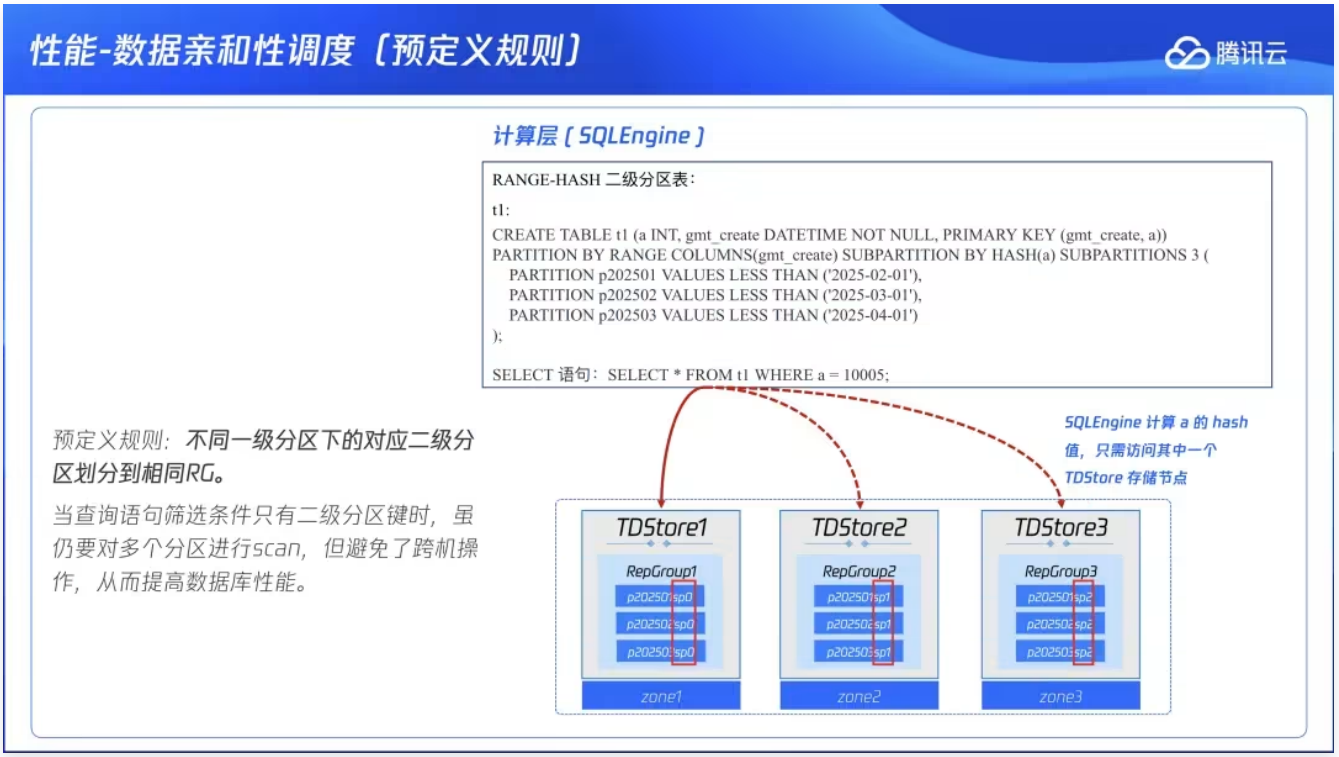
**通用规则**：如关系数据库中数据对象的结构耦合，此规则无需修改。
**预定义规则**：可能会给性能带来提升的逻辑耦合，如与业务需求不符，可以对此规则进行修改。
**自定义规则**：用户可根据业务需求自行创建。
通过这种分层规则体系，TDSQL Boundless 实现了系统性能与业务灵活性的最佳平衡：在确保高效运行的同时，显著降低事务延迟并提升查询响应速度。接下来，我们将深入解析其技术实现细节。

1. **通用规则**

   下图展示了 TDStore 的三层数据模型：在系统中一个表不会直接对应一个物理文件，而是会先映射到一个范围段 Region。图的右侧部分展示了 table 1 分配了01和03两个 Region，01对应主键，03对应其二级索引。当表的数据量非常大的时候，我们就可以把 Region 拆分，然后将其映射到其他数据节点上。在 Region 和物理数据之间引入了复制组（RG），可以灵活的把不同范围段的 Region 放入这个 RG 中，一个 RG 对应一个 Raft 日志流，如果一个事务只修改了这个复制组中的数据，那么其就可以做完单机事务之后提交，但是如果一个事务涉及到了多个复制组的话，就必须以分布式事务的形式走两阶段提交。
   通过内置的通用规则，TDStore 利用数据亲和性来消除分布式事务：默认将表的主键和其对应的二级索引存放在同一个 RG 中【如图的左下侧所示，table1 的主键是 Region 1，其二级索引是 Region 3，都归属于 RG 1】，这样就保证了对表的更新和写入都一定是单机事务。

2. **预定义规则**

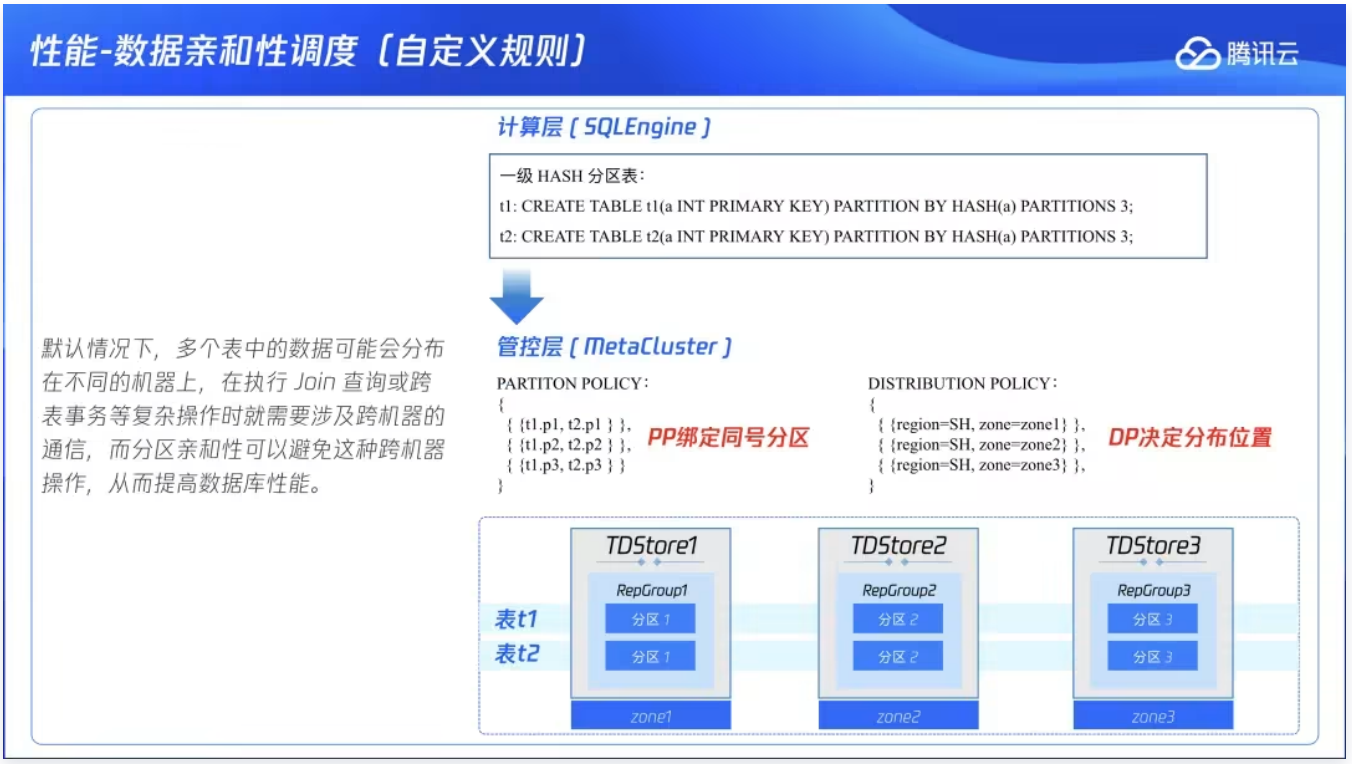
   TDSQL Boundless 支持预定义的数据亲和性规则，这些是在适配业务逻辑的日常工作中总结出的一些经验，如：创建二级分区表时，不同一级分区下的对应二级分区划分到相同 RG，如表 t1.p0.sp0 与 t1.p0.sp0 放到相同 RG。如果业务有特定需求，有方法修改其默认行为。




3. **自定义规则**

   此外，TDSQL Boundless 还支持根据业务逻辑自定义数据亲和性规则。用户可以根据业务数据的关联性控制一组表在物理存储上的邻近关系，进一步优化事务处理性能。具体的做法是通过特定的 SQL 命令创建一组表同

号分区的绑定策略，以及节点分布策略，并与一组表进行绑定（这组表要求分区类型、分区个数、分区值一致），系统就会将这组表的同号分区聚集在同一台机器上，避免跨机查询和分布式事务，提高性能。




# TDSQL Boundless 分区亲和性优化实践

在实践部分，我们举一个"自定义规则"数据亲和性的例子。具体来说，我们将对比两个表在执行 JOIN 查询时的表现：一个场景是这两个表满足数据亲和性，另一个场景则是它们不满足数据亲和性。通过对比，我们可以更直观地感受数据亲和性对查询性能的实际影响。

**硬件环境**

| 节点类型 | 节点规格 | 节点个数 |
| --- | --- | --- |
| HyperNode | 16Core CPU/32GB Memory/增强型 SSD 云硬盘300GB | 3 |

**测试目的**

TDSQL Boundless 率先支持的是一级 HASH 分区表的隐式分区亲和性，只要有关联关系的表都创建为一级 HASH 分区，且分区个数相同，同号分区默认就会被调度到同一个复制组【后续版本会支持手工创建亲和性策略，以及支持更丰富的分区表类型】。

本次测试的数据模型是订单和商品明细的关联查询。

**数据准备**

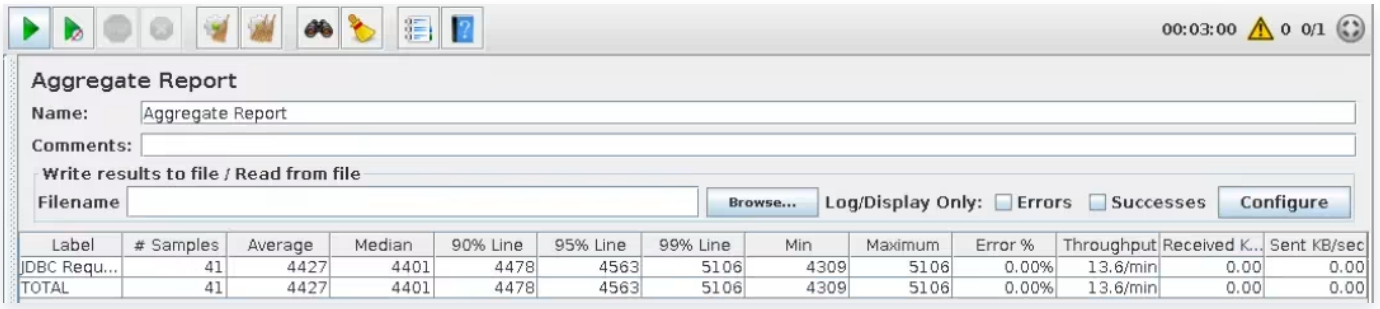
```
-- 订单表 orders，一级hash分区表：
CREATETABLE orders (
    order_id INTNOTNULLAUTO_INCREMENT,
    customer_id INTNOTNULL,
```

```
    order_date DATENOTNULL,
    total_amount DECIMAL(10,2)NOTNULL,
PRIMARYKEY(order_id)
)
PARTITIONBYHASH(order_id)
PARTITIONS 3;

-- 订单明细表 order_items，每个订单随机1-3个商品，一级hash分区表：
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;

--造数过程略，两表最终记录数：
MySQL [klose]>selectcount(*)from orders;
+----------+
|count(*)|
+----------+
|1000000|
+----------+
1rowinset(0.10 sec)

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

--查看数据分布，确保同号分区在相同对等节点的同一个 RG 中：
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|
+--------------+----------------+-------------+----------------+--------
------------------+---------+---------+

--使用 Jmeter 测试关联查询，观察 local scan 的性能:
SELECTCOUNT(*)FROM orders a JOIN order_items b ON a.order_id=b.order_id;
```

```
--执行计划，每个节点启一个 worker 线程，完成本节点的并行子任务：以 orders 表在当前节点的任一子分区（p0-p2）作为驱动表，其每一行记录到 order_items 表中利用索引快速找到满足条件的匹配行，最终汇聚所有 worker 线程的结果并返回；如果都是 local scan，执行效率将大幅提升：
|-> 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)
```

- 单进程，持续执行3分钟，在此期间完成请求41次，平均响应时间为4.4s：




- 模拟老版本 TDSQL Boundless 没有自动应用分区亲和性的情况：

```
--当前同号分区都在相同节点上：
+--------------+----------------+-------------+-----------------+------
--------------------+---------+---------+
| 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|
+--------------+----------------+-------------+----------------+------
-------------------+---------+---------+
```

--模拟先前版本 TDSQL Boundless 中默认没有分区亲和性的状态，将 RG 分裂，并进行打散：

```
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';
```

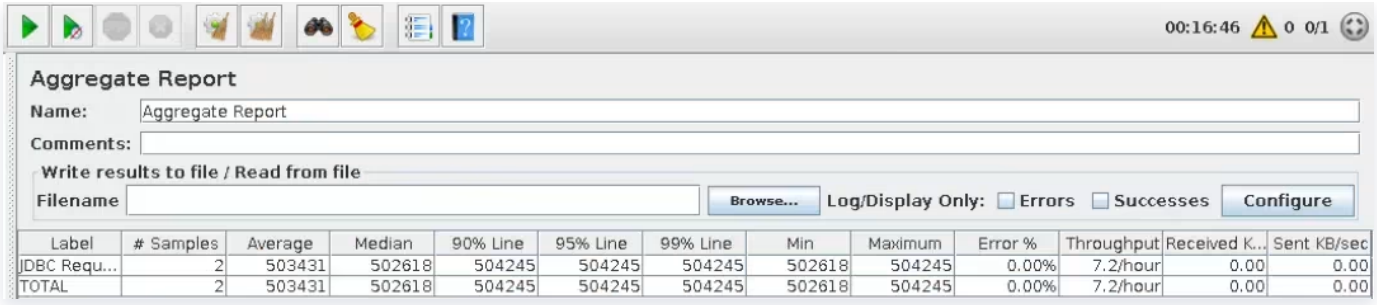
--调整 TDMC 参数，避免预创建 RG leader 回切：
将 TDMC 管控参数 check-primary-rep-group-enabled 设置为 0。

--查看数据分布，确认同号分区当前已不再相同对等节点上：

```
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|
+--------------+----------------+-------------+----------------+------
--------------------+---------+---------+

--再次执行关联查询，由于现在同号分区全都不在一个对等节点上，orders 表中每一行记录
到 order_items 表中进行索引匹配都要走 RPC 调用，叠加三可用区之间 1-3ms 左右的延
迟，预期执行效率会大幅下降：
SELECTCOUNT(*)FROM orders a JOIN order_items b ON
a.order_id=b.order_id;
```

- **单进程，持续执行16分46秒，在此期间完成请求数量2次，平均响应时间为8分23秒，执行效率相差了足足113倍。**




# TDSQL Boundless 分区亲和性的未来发展

我们可以看到，分布式数据库的集中分布式一体化能力有效地弥合了传统关系型数据库向分布式架构演进的鸿沟。同时当业务规模增长到达到分布式架构的触发阈值时，依托多级规则体系的数据智能调度能力，显著提升了业务的接入效率和实际性能。架构平滑演进与智能动态调度相结合，为数字化转型提供了稳健且智能化的演进路径。

TDSQL Boundless 正处于飞速发展期，各项功能以用户需求为导向，不断快速迭代和完善。我们仍在持续不断地优化数据智能调度功能，当前版本已率先为常用的一级 HASH 分区表适配了隐式分区亲和性。在未来版本中，我们将进一步扩展支持涵盖一级 KEY 分区表以及 range / list + hash 二级分区表的隐式分区亲和性；并同步引入灵活定制的显式分区亲和性策略，初期将支持一级 HASH 分区表和普通表的显式亲和性指定。

# TDSQL Boundless 选型指南与实践教程

最近更新时间：2025-11-18 10:10:24

## 一、TDSQL Boundless 规格选择

- 如果是从 InnoDB/B+ Tree 结构的存储迁移，由于 TDSQL 使用 LSM 结构默认会进行压缩，因此可以简单预估为原空间的1/3（单副本）。例如：InnoDB 中有1T数据（单副本，若为一主两备，则总空间为3T），迁移到 TDSQL Boundless 后，单副本空间约为333G，默认三副本则约为1T。数据迁移完成后，将检查磁盘使用率并进行扩缩容。
- 优先选择高节点配置，少节点数：例如如果总共为12c/24g/600G的资源需求，则4c/8g/200G x 3节点相较2c/4g/100G x 6节点更优。这是由于更多小节点会带来更多比例的分布式事务和节点之间通信开销。
- 扩容时优先进行垂直扩容；缩容时优先减少节点。
- TDSQL Boundless 实现了动态的 CPU / 内存 / 磁盘扩缩容，业务无感知。

### 性能参考数据

下表展示了：3节点，16Core CPU / 32GB Memory / 增强型 SSD 云硬盘。32张表，每张表一千万条记录的 Sysbench 性能数据。
详情请参考 性能测试 > Sysbench 测试 。

| 场景 | 线程数 | TPS | QPS | P95延迟（毫秒） |
|---|---|---|---|---|
| 点查 | 32 | 42,210.44 | 42,210.44 | 1.06 |
| | 64 | 73,809.92 | 73,809.92 | 1.21 |
| | 128 | 143,840.08 | 143,840.08 | 1.27 |
| | 256 | 193,770.50 | 193,770.50 | 1.79 |
| 只读 | 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 |
| 索引更新 | 32 | 7659.33 | 7659.33 | 5.57 |
| | 64 | 14070.75 | 14070.75 | 6.32 |
| | 128 | 23513.47 | 23513.47 | 7.98 |
| | 256 | 37637.48 | 37637.48 | 10.65 |
| 非索引更新 | 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 |
| 只写 | 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 |
| 读写混合 | 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 |

# 二、TDSQL Boundless 的使用

## 推荐使用场景

- **业务分库分表场景**：迁移至 TDSQL Boundless 后，无需再分库分表。如原使用 TDSQL MySQL InnoDB 分布式的业务，或是业务自己进行分库分表的业务。
- **存储成本高的业务场景**：TDSQL Boundless 默认具备压缩能力，可以显著降低存储成本。
- **写入量大的场景**：比如流水日志类场景。
- **HBase 替换场景**：提供二级索引支持、跨行事务支持等。

## SQL 限制和差异点

TDSQL Boundless SQL 语法上兼容 MySQL 8.0，有小部分限制：

- 不支持外键。
- 不支持虚拟列、GEOMETRY 类型、降序索引、全文索引。

- 自增字段 cache 默认100，可以保证全局唯一，不保证全局单调递增。如将自增 cache 设为1，可以保障全局单调递增，但会影响批量显式指定自增值的写入性能。

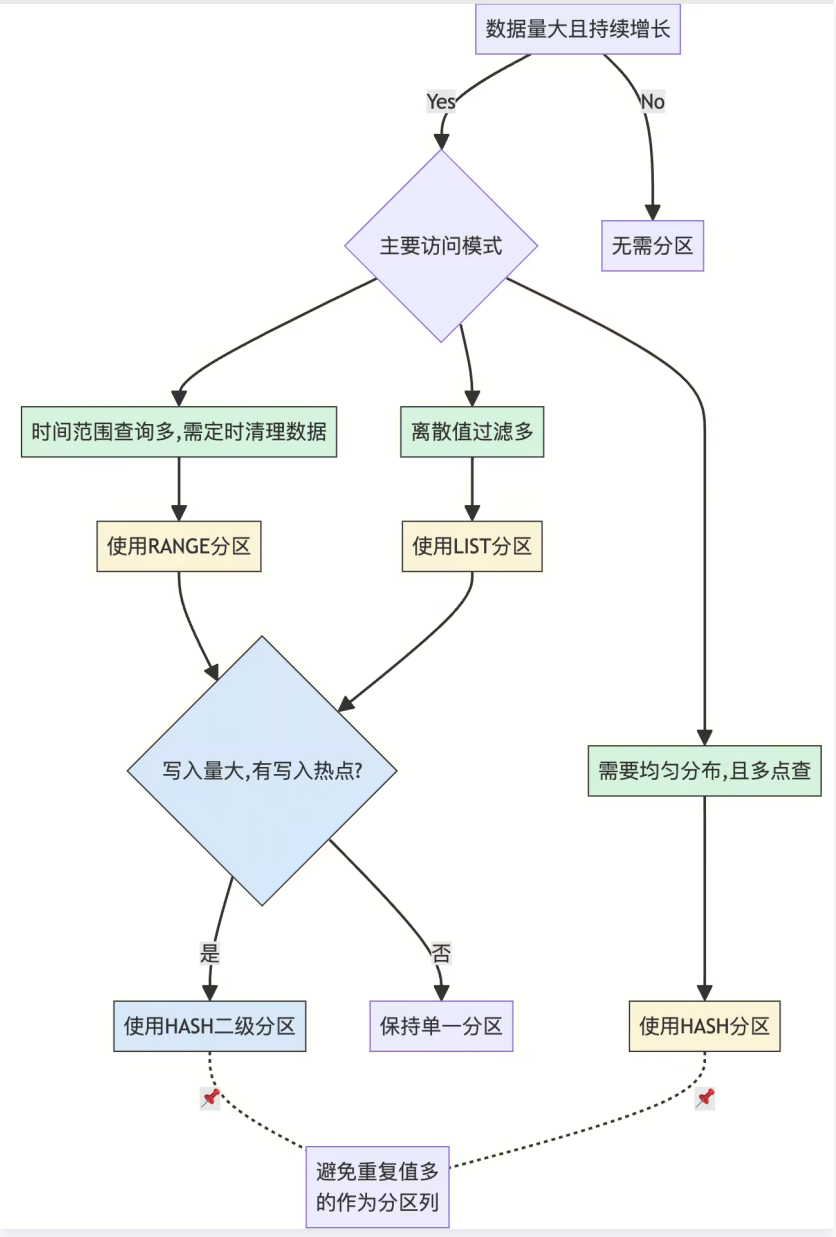
更多的限制和差异点请参考 使用说明 。

## 分区表

- 推荐使用 HASH 分区以打散写入热点，分区数量建议为节点数的整数倍；查询时优先使用 HASH 分区列，避免将重复值较多的列作为分区列。
- 对于需要定期清理过期数据的场景，推荐使用 RANGE 分区。与此同时，如果写入量大且需要打散写入热点，建议采用 RANGE + HASH 二级分区方式建表。查询的谓语条件带上时间列和 HASH 分区列最佳。

**分区数不宜过多。例如，若需要保留3年的数据，按天分区会产生1000个分区。建议选择按周或按月分区，以减少分区数量。**

分区类型的选择决策，请参考下图：



## Online DDL 能力

TDSQL Boundless 支持大部分的 Online DDL（包括部分只需要修改元数据的 Inplace DDL，在此也统称为 Online DDL），具体能力请参考 OnlineDDL 说明 。

# 三、TDStore 存储

和传统 MySQL 主从复制不一样，TDSQL Boundless 采用更小的存储管理单位 RG（replica group，复制组）。每个 RG 使用 Raft 协议保证副本的一致性，每个 RG 的 Leader 可以分布到实例的任意节点上，因此，TDStore 的任意节点都可以承载读写的请求，从而可以更充分地使用所有节点的资源。