分布式数据库 TDSQL

自研内核

腾讯云

文档目录

自研内核
内核概述

最近更新时间：2025-12-24 12:03:43

TDSQL Boundless 计算引擎高度兼容 MySQL 8.0，以功能层面而言，用户可参考 MySQL 8.0 官方文档 。对于不兼容部分，用户可参考 兼容性说明 。
在存储层面，TDSQL Boundless 的存储引擎使用了 LSM Tree 架构而非原生 MySQL 的 B+Tree 引擎，且进行了分布式改造，因此其存储部分行为不完全等价于原生 MySQL。关于存储和计算引擎的基本介绍，用户可参考 系统原理 。
在高度兼容 MySQL 8.0 的基础上，TDSQL Boundless 加入了诸多扩展功能，这使得 TDSQL Boundless 能更好地协助用户构建海量数据下的应用。

模块	特性	说明
功能类	数据高压缩比	相比于传统的 MySQL InnoDB 引擎，TDSQL Boundless 提供压缩能力。本文为有计划进行数据库迁移或选型的用户提供磁盘的容量评估。
	闪回读取和备机读取	闪回读取和备机读取是数据库系统中一种读取历史数据版本的机制。用户可以指定特定历史时间点或距离当前时间的时间间隔来获取数据库在过去某个时刻的数据快照。
	同步表	同步表功能可以使一张表在多个物理节点存储一致的副本，这样针对该表的访问可以最大程度转化为本地访问，消除 RPC。同步表适合写少读多的情况。
	Hint	TDSQL Boundless 在兼容 MySQL 官方 Hint 标准的基础上，针对分布式并行执行特性扩展了专门的并行 Hint。这些 Hint 主要用于优化器指导，帮助用户更精细地控制查询的并行执行策略，提升复杂查询的性能。
性能类	Bulk Load 快速导入模式	TDSQL Boundless 支持以 bulk load 的方式向数据库快速导入数据。相比于执行传统 SQL 普通事务写入的方式，bulk load 快速导入模式在性能上通常提升5~10倍以上，适合在业务上线阶段用作迁移现有的大型数据库到全新的 TDSQL Boundless 集群。
	副本迁移性能优化	TDSQL Boundless 在数据压缩（Compaction）过程中主动对齐 SST 文件与 Region 边界，从而使得副本迁移时能直接识别和传输整个文件，极大提升了效率。
	自适应热点锁管理器	TDSQL Boundless 的自适应热点锁管理器在 LSM-tree 上高效支持悲观锁，锁管理器可以随 Region 一起分

		裂，有效打散热点瓶颈。

功能类特性
数据高压缩比

最近更新时间：2025-12-24 12:03:43

相比于传统的 MySQL InnoDB 引擎，TDSQL Boundless 提供最高3.81的压缩比。本文为有计划进行数据库迁移或选型的用户提供磁盘的容量评估。

核心优势解读：为何 TDSQL Boundless 能实现如此高的压缩比?

TDSQL Boundless 采用了 LSM 结构存储数据，由于 LSM 的 Append only 的特点，其避免了 B+ 树因原地更新导致频繁的随机写入产生的数据页内碎片。同时LSM 后台 compaction 的特点，避免了每次写入都要压缩数据的开销，极大地降低了压缩对性能的影响。因此 TDSQL Boundless 相比 MySQL InnoDB，在相近的性能表现下，数据压缩比可达3.81。
解读完内核，我们来看下实测数据，量化对比 TDSQL Boundless 数据库与 MySQL 在存储相同数据时的空间占用，并验证高压缩比在不同数据模型（OLTP、OLAP）下的有效性，为用户从 MySQL 迁移时，提供科学的磁盘容量规划指引。

测试概述

测试环境

选项	说明
云平台	腾讯云
实例规格	16Core CPU/32GB Memory/增强型 SSD 云硬盘 300GB

对比数据库

- 对照组：MySQL 8.0（使用 InnoDB 存储引擎，默认配置）
- 实验组：TDSQL Boundless（内置高效数据压缩引擎）

关键配置

MySQL 8.0 InnoDB 引擎默认配置：

```
# MySQL InnoDB 默认参数配置
innodb_file_per_table=ON
innodb_page_size=16K
# 未启用 innodb_page_compression （代表绝大多数生产环境的默认情况）
```

```
innodb_page_compression=OFF
```

测试数据集

- Sysbench 基准测试
- TPC-C 基准测试
- TPC-H 基准测试

测试计划

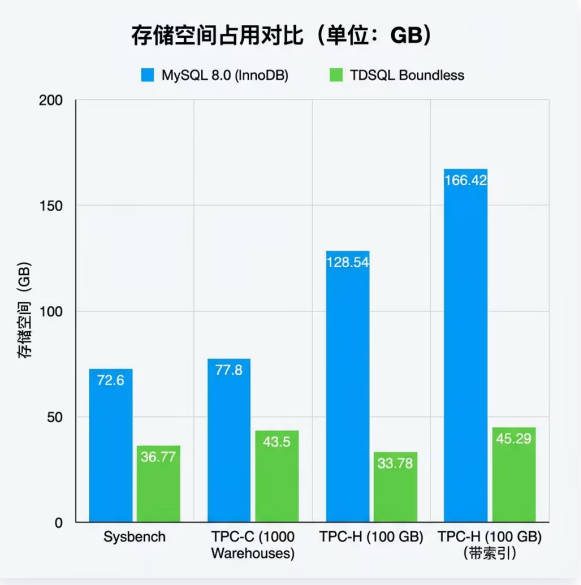
准备测试数据

- Sysbench 基准测试：参考 Sysbench 测试，初始化32张表，每张表1000万条记录。
- TPC-C 基准测试：参考 TPC-C 测试，初始化1000个 Warehouse。
- TPC-H 基准测试：参考 TPC-H 测试，初始化8张表(带索引)，生成100GB测试数据集。

测试结果与数据分析

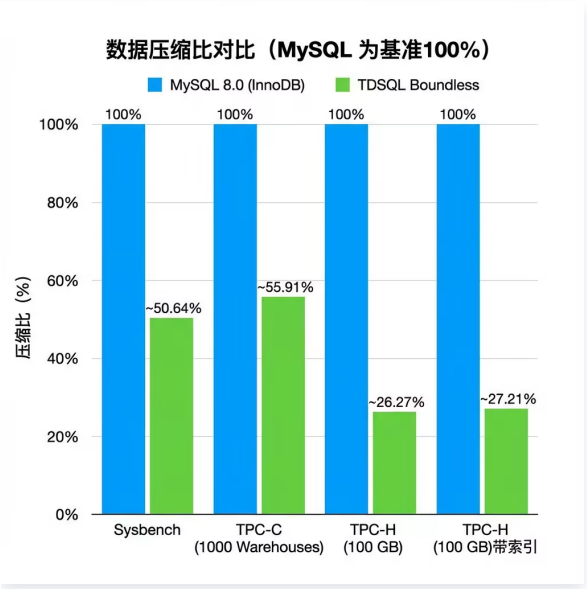
存储空间占用对比（单位：GB）

数据集	MySQL 8.0 (InnoDB)	TDSQL Boundless
Sysbench	72.6GB	36.77GB
TPC-C （1000 Warehouses）	77.8GB	43.5GB
TPC-H (100GB)	128.54GB	33.78GB
TPC-H (100GB)（带索引）	166.42GB	45.29GB



数据压缩比对比（MySQL 为基准100%）

数据集	MySQL 8.0 (InnoDB)	TDSQL Boundless
Sysbench	100%	~50.64%
TPC-C (1000 Warehouses)	100%	~55.91%
TPC-H (100GB)	100%	~26.27%
TPC-H (100GB)（带索引）	100%	~27.21%



用户迁移指南：磁盘容量规划建议

如果您正在从 MySQL 迁移到 TDSQL Boundless，并希望评估目标数据库的磁盘需求，请遵循以下简单指引：

第一步：评估 MySQL 的磁盘使用情况

登录您的 MySQL 实例，执行以下命令查看总数据大小：

```
SELECT table_schema AS 'Database',
       ROUND(SUM(data_length + index_length) / 1024 / 1024 / 1024, 2) AS
'Size(GB)'
FROM information_schema.TABLES
GROUP BY table_schema;
```

第二步：根据业务场景选择压缩系数

源端 MySQL 业务数据类型	建议规划系数

通用 OLTP 业务（订单、用户等）	按 MySQL 空间的 50% 规划
日志、监控、时序数据	按 MySQL 空间的 30% 规划
数据仓库、报表分析 (AP)	按 MySQL 空间的 30% 规划

第三步：计算目标磁盘容量

所需磁盘约等于源端 MySQL 容量 ✖ 规划系数。

闪回读取和备机读取

最近更新时间：2025-12-24 12:03:42

概述

闪回读取和备机读取是数据库系统中一种读取历史数据版本的机制。用户可以指定特定历史时间点或距离当前时间的时间间隔来获取数据库在过去某个时刻的数据快照。

- **闪回查询**：当发生误操作（如 `UPDATE` 或 `DELETE` 了错误的数据）时，可以通过查询误操作发生前的时间点，快速找回历史数据，极大地简化了数据恢复流程。TDSQL Boundless 提供了语句级别的闪回查询功能。
- **备机读取**（Stale Read）：在对数据实时性要求不高的读密集型业务场景中，可以将读请求转发到备机（Follower）节点。这在牺牲了秒级数据实时性的前提下，有效分担了主机（Leader）的读取压力，降低了读取延迟，提升了系统整体的资源利用率和吞吐能力。TDSQL Boundless 提供了语句级别、事务级别、会话级别三种级别的 Stale Read 功能，并且允许用户依据业务情况选择读取数据的副本角色。

支持版本

- 闪回查询支持17.0.0及以上版本。
- 备机读取支持20.0.0及以上版本。

使用限制

闪回查询功能用于在数据库运维过程中快速查询误操作前的历史数据。使用该功能时需注意以下几点：

1. **存储引擎限制**
   - 仅支持 RocksDB 引擎物理表
   - 不支持视图、其他引擎表
   - 不支持 `last_insert_id()` 等无实际列对应的函数
2. **时间点指定方式**
   - **混合逻辑时钟时间戳**（GTS）：精确定位到具体快照数据，如 SQL 指定29610460101738510的GTS。
   - **秒级闪回查询**：查询指定秒内的最新写入数据，相当于逻辑部分全为1的混合逻辑时间戳。如 SQL 指定2025-01-01 12:00:00时间点。
   - 不支持指定未来时间（返回错误 `Can't stale read from the future`）。
3. **操作限制**
   - 单条 SQL 仅支持一个闪回快照，即所有子查询共享同一闪回时间点，不可分别指定。
   - 仅支持自动提交的无锁只读查询。不支持 `SELECT FOR UPDATE`、`FOR SHARE`、`LOCK IN SHARE MODE` 等锁定查询；不支持 `INSERT...SELECT` 语句；不支持 `BEGIN; SELECT...` 语法。
   - 不支持 DDL 操作。若表结构发生变更，需通过回收站恢复表结构和数据。闪回查询的行为和数据取决于 DDL 的具体形式，结果不固定。

4. **数据范围限制**
   - 无法查询早于全局最早快照（GES）的数据。
   - GES 受 MC 参数 `advance-global-earliest-snapshot-delay` 控制。启用闪回功能会延迟 GES 推进，可能增加存储占用和查询耗时。

使用说明

准备工作

```
-- 启用闪回查询功能
SET persist tdsql_enable_stale_read = ON;
```

闪回查询使用说明

语法

闪回查询引入了全新的 `AS OF` 语法，查询指定时间点的数据。

```
SELECT ... FROM <table_name> [ AS OF {TIMESTAMP timestamp_option | GTS
gts_option} ]
```

timestamp_option 支持类型

凡是可以 `get_date` 的表达式，都可以作为 `timestamp_option`，包括但不限于：

- 时间字面值：`'2023-09-05 12:00:00'`
- 时间函数：`NOW()`，`SYSDATE()`，`UTC_TIMESTAMP()`
- 用户变量：`SELECT NOW() INTO @time1`
- 时间算术表达式：`NOW + INTERVAL 1 SECOND`，`@time1 - INTERVAL 1 YEAR`

gts_option 支持类型

凡是可以作为整数的表达式，都可以作为 `gts_option`，包括但不限于：

- 整数字面值：`29610460101738510`
- 函数和算术表达式：`((UNIX_TIMESTAMP(@time1) << 24) | 0xFFFFFF)`
- 用户变量：`@gts1`

示例

```
-- 创建测试表
CREATE TABLE t1(a INT PRIMARY KEY, b INT);

SELECT sleep(3);
```

```
-- 记录初始时间点
SELECT NOW() INTO @time0;
SELECT ((UNIX_TIMESTAMP(@time0) << 24) | 0xFFFFFF) INTO @gts0;

SELECT sleep(3);

-- 插入测试数据
INSERT INTO t1 VALUES(1,1),(2,2),(3,3),(4,4),(5,5),(6,1),(7,2),(8,3),
(9,4),(10,5);

SELECT sleep(3);

-- 记录数据插入后时间点
SELECT NOW() INTO @time1;
SELECT ((UNIX_TIMESTAMP(@time1) << 24) | 0xFFFFFF) INTO @gts1;

SELECT sleep(3);

-- 更新数据
UPDATE t1 SET b=b*10;

-- 当前数据查询
SELECT * FROM t1 ORDER BY a;
a»b
1»10
2»20
3»30
4»40
5»50
6»10
7»20
8»30
9»40
10»■50

-- 闪回查询到初始状态（空表）
SELECT * FROM t1 ORDER BY a AS OF TIMESTAMP @time0;
a»b
SELECT * FROM t1 ORDER BY a AS OF GTS @gts0;
```

```
a»b

-- 闪回查询到插入数据后的状态
SELECT * FROM t1 ORDER BY a AS OF TIMESTAMP @time1;
a»b
1»1
2»2
3»3
4»4
5»5
6»1
7»2
8»3
9»4
10»■5


SELECT * FROM t1 ORDER BY a AS OF GTS @gts1;
a»b
1»1
2»2
3»3
4»4
5»5
6»1
7»2
8»3
9»4
10»■5
```

备机读取使用说明

设置读取节点

```
-- 0：Leader节点，1：Follower节点
SET persist tdsql_stale_read_role=1;
```

语句级别备机读取

操作与语句级别的闪回查询相同。

会话级别/事务级别备机读取

提供 session 变量 `tdsql_read_staleness_t` ，支持会话级别和事务级别的查询。

```
--查询当前数据
+----+-----+
| id | num |
+----+-----+
|  1 |  10 |
|  2 |  20 |
|  3 |  30 |
|  4 |  40 |
+----+-----+

set tdsql_read_staleness_t='-10'; #读取10秒前的数据

--插入新的数据
insert into t1 values (5,50);

--查看当前时间
select now();
+---------------------+
| now()               |
+---------------------+
| 2025-02-16 17:57:50 |
+---------------------+

--查询当前数据
select * from t1; # 返回历史数据
+----+-----+
| id | num |
+----+-----+
|  1 |  10 |
|  2 |  20 |
|  3 |  30 |
|  4 |  40 |
+----+-----+

--等待一段时间
--查看当前时间
```

```
select now();
+---------------------+
| now()               |
+---------------------+
| 2025-02-16 17:58:01 |
+---------------------+

--再次查询当前数据
select * from t1; # 返回新数据
+----+-----+
| id | num |
+----+-----+
|  1 |  10 |
|  2 |  20 |
|  3 |  30 |
|  4 |  40 |
|  5 |  50 |
+----+-----+
```

设置 `tdsql_read_staleness_t` 后，还可以开始事务查询数据。

```
--查询当前数据
select * from t1;
+----+-----+
| id | num |
+----+-----+
|  1 |  10 |
|  2 |  20 |
|  3 |  30 |
|  4 |  40 |
|  5 |  50 |
+----+-----+

--插入新的数据
insert into t1 values (6,60);

--开启事务
begin;
select * from t1; # 返回历史数据
```

```
+----+-----+
| id | num |
+----+-----+
|  1 |  10 |
|  2 |  20 |
|  3 |  30 |
|  4 |  40 |
|  5 |  50 |
+----+-----+
select sum(num) as total_num from t1;
+-----------+
| total_num |
+-----------+
|       150 |
+-----------+
rollback;

--等待一段时间
--开启事务
begin;
select * from t1; # 返回新数据
+----+-----+
| id | num |
+----+-----+
|  1 |  10 |
|  2 |  20 |
|  3 |  30 |
|  4 |  40 |
|  5 |  50 |
|  6 |  60 |
+----+-----+
select sum(num) as total_num from t1;
+-----------+
| total_num |
+-----------+
|       210 |
+-----------+
rollback;
```

相关参数

参数名	参数范围	类型	默认值	取值范围	是否需重启	说明
tdsql_enable_stale_read	GLOBAL	Boolean	ON	ON/OFF	否	闪回查询功能的开关。
advance-global-earliest-snapshot-delay	GLOBAL	Integer	10	0~43200	否	MC 一侧用于调控闪回查询的可查询的时间范围，单位：秒。 此参数的值越大，闪回查询支持的历史数据查询时间越长，同时占用的存储空间和查询时长也可能增加。
tdsql_stale_read_role	GLOBAL	UINT	0	0/1	否	设置 Stale Read 路由访问的副本。 • 0：表示 Leader 节点。 • 1：表示 Follower 节点。
tdsql_read_staleness_t	SESSION	CHAR	''	-	否	设置 Stale Read 数据陈旧时间。示例：`set session tdsql_read_staleness_t='-3'` 开启 Stale Read staleness 模式，读取数据的时间为当前物理时间前3秒。 注意：`set session tdsql_read_staleness_t=''`（空字符）表示关闭 Stale Read staleness 模式。

同步表

最近更新时间：2025-12-24 12:03:42

概述

同步表是一种特殊的表。数据写入时，它的修改需要强同步到多个 Follower 副本后，才能返回。因此，它的多个 Follower 副本能提供强一致性读服务。它适合写入频率较低，读操作较多、读操作延迟要求高的负载。
目前同步表以广播表的形式对外提供服务，用户的所有同步表的 Region 共用一个特殊的广播同步 RG(Replication Group)。该 RG 在用户创建第一个同步表时创建，后续用户的其他同步表 Region 也会调度到该同步 RG。广播同步 RG 在系统内每个节点上均包含一个副本，每个 Follower 定期向 Leader 申请租约，在 Leader 上发生写入时，需要等待直到修改强同步到每个持有有效租约的 Follower 上后才能返回。在有读取请求时，同步表会先尝试本地读取，本地读取失败再读取 Leader 。

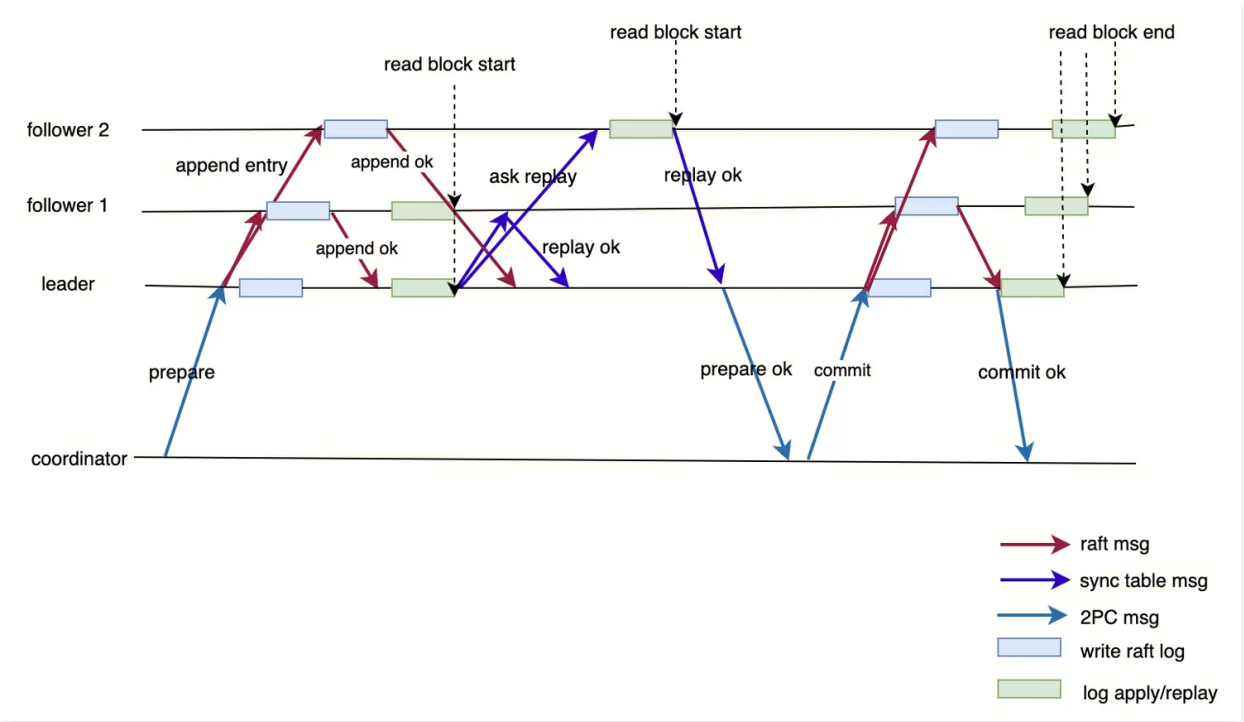
工作原理

普通事务写入流程

1. 数据在事务提交前会暂存在事务上下文的 write batch 中。
2. 进入提交阶段，Leader 会将 write batch 中的数据落盘形成事务日志，通过 Raft 协议同步到各副本所在节点。
3. 形成多数派后，Leader 再把数据先写入 RocksDB，然后返回客户端。Follower 副本则回放 Raft 日志并写入 RocksDB。

同步表事务写入流程

1. 在执行阶段，同步表事务和普通事务一样都把数据暂存在事务上下文的 write batch 中。
2. 进入提交阶段后，Leader 同样会以 Raft 日志的形式将数据同步给 Follower。
3. 和普通事务不同的是，同步表要求所有有效的 Follower 都返回了日志，然后才能返回客户端。



具体来说，对于分布式事务，Leader 向 Follower 做强同步的逻辑如下：

3.1 提交 redo 和 prepare 日志。

3.2 获取所有 lease valid 的副本。

3.3 询问每个副本是否已经回放 prepare 日志。

3.4 如果所有副本已经回放 prepare 日志，应答协调者。

3.5 否则，跳转到 步骤2 重试。

Follower 上收到读请求时不能再直接报错，而是需要检查可读性（包括 lease 检查等），可读性检查通过后读的逻辑和 Leader 上一致，即先检查锁信息看是否有处于 prepare 状态的事务需要阻塞自己，没有则直接根据读版本号读取 RocksDB。

Leader 需要维护一个有效 Follower 列表，提交阶段需要向所有有效 Follower 进行强同步。这个有效 Follower 列表是通过 lease 机制来维护的。Follower 定期向 Leader 申请 lease，Leader 收到 lease申请后，如果同意授予 lease，则将该节点加入有效 Follower 列表，并通过 Raft 日志同步 lease log。之后 Leader 上的所有修改都需要向该 Follower 同步。如果 Follower 长时间未续约 lease 导致 lease 过期，Leader 将该 Follower 踢出有效列表，后续不再向该 Follower 做强同步。Follower 在 lease 有效期内才可以对外提供强同步服务，lease 过期则禁止提供强同步服务。该机制保证 Follower 上的 lease 先过期，即 Follower 先停止提供强同步服务；Leader 上的 lease 后过期，即 Leader 后停止强同步。

使用限制

- 目前同步表以广播表的形式对外提供服务，用户必须创建广播同步表来使用同步表。
- 任意一个 Follower 副本故障，可能导致写请求卡顿一个租约的时间。
- 读请求路由策略本地优先，本地访问失败则访问 Leader 。
- 目前暂不支持同步表的属性变换，即不支持从同步表变为非同步表，也不支持从非同步表变为同步表。

- 同步表不能是分区表。
- 广播同步日志流一旦被创建出来后就不会再被销毁，即使用户删除所有的同步表。
- 副本个数过多会导致写请求性能下降。

使用说明

同步表适用于读多写少且对延迟敏感的业务表。如系统参数、全局配置表、数据仓库维度表（如 TPCC 中的 item 表）。

语法

```
CREATE TABLE table_name (
    column_definitions
) sync_level = node(all) distribution = node(all);
```

同步表的读写访问语法和普通表保持一致。但读取时，具体访问的节点和连接有关。具体来说，TDSQL Boundless 会优先访问连接所在的本地节点的同步表副本，只有当本地节点不可读时（例如 lease 过期），才会退化为访问 Leader 节点。

示例

```
tdsql [demo]> CREATE TABLE test_sync_table(
    c1 INT PRIMARY KEY,
    c2 INT,
    c3 INT,
    INDEX idx(c2)
) sync_level = node(all) distribution = node(all);
Query OK, 0 rows affected (2.90 sec)
```

相关参数

参数名	默认值	说明
tdstore_sync_table_max_allowed_log_lag	1024	Leader 在决定是否授予 Follower lease 时，需要考虑 Follower 的日志回放进度。如果 Follower 当前回放进度落后于 Leader 超过该参数值时，Leader 将不会授予 lease。
tdstore_sync_table_lease_interval_ms	2000	Follower 向 Leader 申请 lease 的周期。

tdstore_sync_table_log_interval_ms	1000	Leader 同步 Raft lease log 的周期。
tdstore_sync_table_lease_len_us	10000000	Leader 每次向 Follower 授予的 lease 有效期。

Hint

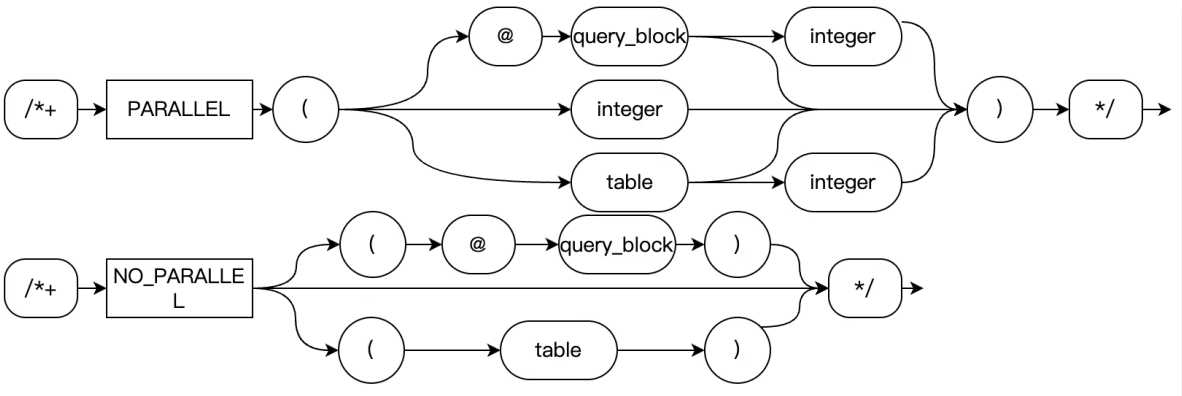
最近更新时间：2025-12-24 12:03:43

概述

TDSQL 在兼容 MySQL 官方 Hint 标准的基础上，针对分布式并行执行特性扩展了专门的并行 Hint。这些 Hint 主要用于优化器指导，帮助用户更精细地控制查询的并行执行策略，提升复杂查询的性能。

PARALLEL/NO_PARALLEL Hint 并行度控制

可以使用 PARALLEL Hint 指定进行 Parallel Scan 的表和它的并行度。语法如下：



PARALLEL Hint 是一个3个层级的 Hint Global、Query Block 和 Table 级别，举例说明：

```
PARALLEL(num) -- Global级、Query Block级
PARALLEL(@qbname num) -- Query Block级
PARALLEL(tablename) -- Table 级
PARALLEL(tablename num) -- Table 级
NO_PARALLEL -- Global级、Query Block级
NO_PARALLEL(@qbname) -- Query Block级
NO_PARALLEL(tablename) -- Table 级
```

其中，NO_PARALLEL 是禁止并行，tablename 和 qbname 的语法请参考 MySQL 官方文档。

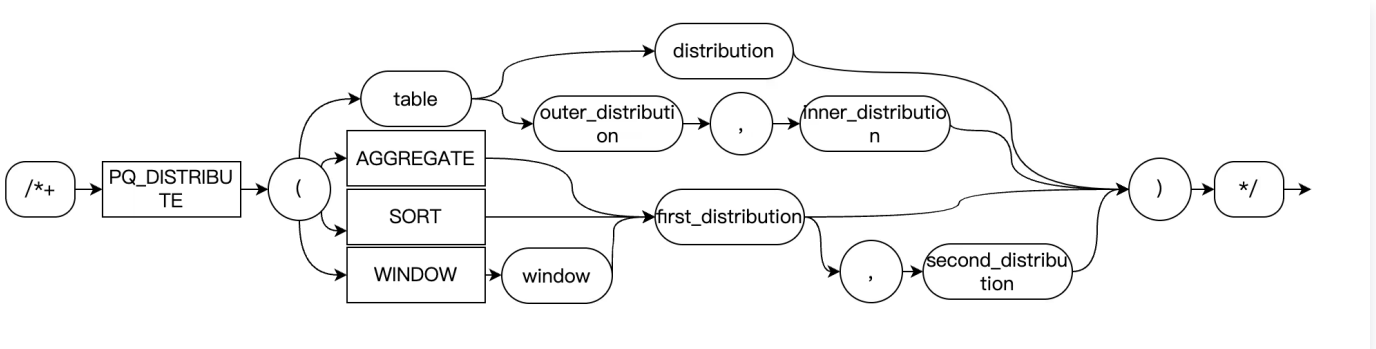
- **Global 级**：当在主查询中指定，没有给定 query block name 和 table name 时。这时 num 将作为整个查询的默认并行度，子查询还可以通过自己的PARALLEL Hint 来覆盖它。
- **Query Block 级**：当在非主查询中指定，或者给定了 query block name，但没有给定 table name 时。这时 num 将作为这个子查询的默认并行度，可以被表级的覆盖。
- **Table 级**：当在 Hint 中给定了表名时。如果只给表名没给 num，会向上（即：Query Block、Table 级别、max_parallel_degree 变量）确定并行度，如果最后没有拿到并行度，即到 max_parallel_degree 时该值为0，则不会并行。另外，我们可以支持指定多个表并行，将来的计划中会支持多个表并行。

这里 Query Block 级和 Global 级（当 Query Block 级未指定时）是可以当前决定当前 Query Block 是否使用并行计划的，如当它被指定了 NO_PARALLEL 时那么当时 Query Block 就不会进入并行优化阶段，而表级别只影响当前表是否进行并行扫描，也就是说如果 Query Block 和 Global 级别未指定 Hint，而当前的 max_parallel_degree 大于0，即使所有表都使用了 NO_PARALLEL，并行优化器还是会找出一个并行扫描表来并行执行。
针对 PARALLEL Hint 的语法，显示一些用法如下：

```
-- Global 级别
EXPLAIN select /*+ PARALLEL(4) */ * from t1 where a > 4;

-- Query block 级别，只在最顶层查询起作用
EXPLAIN select /*+ QB_NAME(q1) PARALLEL(@q1 4) */ * from t1 where a > 4;

-- Query block 级别，只在子查询中有效
explain select * from t3 where a = (select /*+ PARALLEL(2) */ count(a)
from t3);

-- Table 级别
EXPLAIN select /*+ PARALLEL(t1 4) */ from t1 where a > 4;

-- Table 级别
EXPLAIN select /*+ PARALLEL(t3@q2 4) */ * from t3 where a = (select /*+
QB_NAME(q2) */ count(a) from t3);

-- Table 级别
EXPLAIN select /*+ PARALLEL(@q2 t3 4) */ * from t3 where a = (select /*+
QB_NAME(q2) */ count(a) from t3);
```

PQ_DISTRIBUTE Hint 数据分布策略

用于指示优化器如何在查询计划中添加数据重分布操作。语法如下：



```
-- 基本语法
PQ_DISTRIBUTE(tablespec strategy1 strategy2)
PQ_DISTRIBUTE(tablespec strategy)

-- 针对特定操作类型
PQ_DISTRIBUTE(target strategy1 strategy2)
```

- `target`：操作目标，支持 AGGREGATE、SORT 和 WINDOW，分别指定 GROUP BY，ORDER BY， WINDOW 函数的分布策略，WINDOW 可以在后面指定 WINDOW 的名字。
- `strategy1`、`strategy2`：重分布的策略，支持 NONE、GATHER、HASH、BROADCAST。

JOIN 操作分布策略

- `PQ_DISTRIBUTE(t1 HASH, HASH)`、`PQ_DISTRIBUTE(t1@qb1 HASH, HASH)` 或 `PQ_DISTRIBUTE(@qb1 t1 HASH HASH)`：指定 t1 和它前一个表做 JOIN 时，先使用 HASH 重分布，再在各个节点上做 JOIN。
- `PQ_DISTRIBUTE(t1 BROADCAST, NONE)`：指定 t1 和它前一个表做 JOIN 时，前一个也就是外表做 Broadcast，再和 t1 做 JOIN。
- `PQ_DISTRIBUTE(t1 GATHER)`：指定 t1 表先做 Gather，再和其他表做 JOIN。

使用示例：

```
-- 两个表都进行 HASH 重分布后 JOIN
SELECT /*+ PQ_DISTRIBUTE(t1 HASH, HASH) */ *
FROM table1 t1 JOIN table2 t2 ON t1.id = t2.id;

-- 外表广播，内表不重分布
SELECT /*+ PQ_DISTRIBUTE(t1 BROADCAST, NONE) */ *
FROM small_table t1 JOIN large_table t2 ON t1.id = t2.id;

-- 表数据收集到 Leader 后执行 JOIN
SELECT /*+ PQ_DISTRIBUTE(t1 GATHER) */ *
FROM table1 t1 JOIN table2 t2 ON t1.id = t2.id;
```

GROUP BY 聚合分布策略

并行 GROUP BY 操作最多会执行二阶段，这里第1个策略表示是第一阶段 GROUP BY 之前的数据分发操作，第2个则表示第二阶段之前的数据分发操作，如果只有一个策略则表示是一阶段。

- `PQ_DISTRIBUTE(AGGREGATE NONE)`：完全下推一阶段。
- `PQ_DISTRIBUTE(AGGREGATE NONE, GATHER)`：Worker 和 Leader 二阶段。

- `PQ_DISTRIBUTE(AGGREGATE NONE HASH)`：Worker 和 Worker 二阶段。
- `PQ_DISTRIBUTE(AGGREGATE GATHER)`：只在 Leader 上的一阶段。

使用示例：

```
-- 完全下推的一阶段聚合
SELECT /*+ PQ_DISTRIBUTE(AGGREGATE NONE) */
       department, AVG(salary)
FROM employees
GROUP BY department;

-- Worker 和 Leader 二阶段聚合
SELECT /*+ PQ_DISTRIBUTE(AGGREGATE NONE, GATHER) */
       department, AVG(salary)
FROM employees
GROUP BY department;

-- Worker 和 Worker 二阶段聚合
SELECT /*+ PQ_DISTRIBUTE(AGGREGATE NONE HASH) */
       department, AVG(salary)
FROM employees
GROUP BY department;

-- 只在 Leader 上执行的一阶段聚合
SELECT /*+ PQ_DISTRIBUTE(AGGREGATE GATHER) */
       department, AVG(salary)
FROM employees
GROUP BY department;
```

ORDER BY 排序分布策略

- `PQ_DISTRIBUTE(SORT NONE, GATHER)`：Worker + Leader 的 Merge Sort。
- `PQ_DISTRIBUTE(SORT GATHER)`：只在 Leader 上做 Sort。

使用示例：

```
-- Worker 局部排序 + Leader 归并排序
SELECT /*+ PQ_DISTRIBUTE(SORT NONE, GATHER) */ *
FROM large_table
ORDER BY create_time DESC;
```

```
-- 只在 Leader 上全局排序
SELECT /*+ PQ_DISTRIBUTE(SORT GATHER) */ *
FROM large_table
ORDER BY create_time DESC;
```

WINDOW 函数分布策略

PQ_DISTRIBUTE(WINDOW GATHER)

策略说明： 将窗口函数计算完全集中在 Leader 节点执行。所有数据先收集到 Leader 节点，然后在单节点上执行窗口函数计算。

适用场景：

- 数据量相对较小，可以单节点处理
- 需要全局排序的窗口函数（如 RANK() OVER (ORDER BY ...)）
- 窗口函数分区数较少，无法有效并行化

示例：

```
-- 在 Leader 节点上计算全局排名
SELECT /*+ PQ_DISTRIBUTE(WINDOW GATHER) */
       student_id,
       score,
       RANK() OVER (ORDER BY score DESC) as global_rank
FROM exam_scores;

-- 计算每个学生的成绩变化趋势（数据量较小）
SELECT /*+ PQ_DISTRIBUTE(WINDOW GATHER) */
       student_id,
       exam_date,
       score,
       LAG(score) OVER (PARTITION BY student_id ORDER BY exam_date) as
prev_score
FROM student_scores;
```

PQ_DISTRIBUTE(WINDOW HASH)

策略说明： 按照窗口函数的 PARTITION BY 子句的列进行哈希分布，在各个 Worker 节点上并行执行窗口函数计算。

适用场景：

- 数据量较大，需要并行处理
- 窗口函数有明确的分区键，且分区数据分布均匀

- 每个分区内的数据量适中，适合并行计算

示例：

```
-- 按学科分区并行计算排名
SELECT /*+ PQ_DISTRIBUTE(WINDOW HASH) */
       subject,
       student_id,
       score,
       RANK() OVER (PARTITION BY subject ORDER BY score DESC) as
subject_rank
FROM exam_scores;

-- 按部门分区计算工资累计总和
SELECT /*+ PQ_DISTRIBUTE(WINDOW HASH) */
       department,
       employee_id,
       salary,
       SUM(salary) OVER (PARTITION BY department ORDER BY hire_date) as
cumulative_salary
FROM employees;
```

PQ_DISTRIBUTE(WINDOW win1 GATHER)

策略说明： 针对特定命名的窗口（使用 WINDOW 子句定义）采用 GATHER 策略执行。

适用场景：

- 查询中包含多个窗口函数，需要对特定窗口采用不同策略
- 命名的窗口函数需要特殊处理
- 混合使用不同分布策略优化复杂查询

示例：

```
-- 为特定命名的窗口使用GATHER策略
SELECT /*+ PQ_DISTRIBUTE(WINDOW win1 GATHER) */
       student_id,
       subject,
       score,
       AVG(score) OVER win1 as avg_score,
       RANK() OVER (PARTITION BY subject ORDER BY score DESC) as rank
FROM exam_scores
WINDOW win1 AS (PARTITION BY student_id);
```

```
-- 多个命名窗口的不同策略
SELECT /*+ PQ_DISTRIBUTE(WINDOW win1 GATHER) PQ_DISTRIBUTE(WINDOW win2
HASH) */
       department,
       employee_id,
       salary,
       AVG(salary) OVER win1 as dept_avg,
       SUM(salary) OVER win2 as running_total
FROM employees
WINDOW win1 AS (PARTITION BY department),
       win2 AS (PARTITION BY department ORDER BY hire_date);
```

PQ_DISTRIBUTE(WINDOW win1 HASH)

策略说明： 针对特定命名的窗口采用 HASH 分布策略并行执行。

适用场景：

- 特定命名的窗口函数数据量较大，需要并行优化
- 不同窗口函数需要采用不同的并行策略
- 精细化控制复杂查询的执行计划

示例：

```
-- 为特定窗口使用 HASH 分布
SELECT /*+ PQ_DISTRIBUTE(WINDOW win1 HASH) */
       product_category,
       product_id,
       sales_amount,
       SUM(sales_amount) OVER win1 as category_total,
       RANK() OVER (ORDER BY sales_amount DESC) as global_rank
FROM sales_data
WINDOW win1 AS (PARTITION BY product_category);

-- 混合策略：一个窗口 GATHER，另一个窗口 HASH
SELECT /*+ PQ_DISTRIBUTE(WINDOW win1 GATHER) PQ_DISTRIBUTE(WINDOW win2
HASH) */
       customer_id,
       order_date,
       amount,
       AVG(amount) OVER win1 as cust_avg,  -- 小数据量用GATHER
```

```
       SUM(amount) OVER win2 as running_sum -- 大数据量用HASH
FROM orders
WINDOW win1 AS (PARTITION BY customer_id),
       win2 AS (PARTITION BY customer_id ORDER BY order_date);
```

SUBQUERY Hint 子查询并行策略

并行查询 MySQL SUBQUERY Hint 增加了2个并行相关的策略（strategy），这些策略可以和 MySQL 原来的策略混合使用。

- PQ_PRE_EVALUATION：指定子查询在引用它的父查询执行之前先提前执行，并行 Worker 之间直接读取子查询结果。
- PQ_INLINE_EVALUATION：强制不提前执行子查询，也就是按 MySQL 的逻辑根据父查询的需要按需执行。

示例：

```
-- 指定这个 IN 子查询使用 MATERIALIZATION 策略并且在并行计划不使用预先执行策略
EXPLAIN FORMAT=TREE
SELECT * FROM t1
WHERE t1.a IN (
    SELECT /*+ SUBQUERY(MATERIALIZATION, PQ_INLINE_EVALUATION) */ a
    FROM t2
);

-- 指定 Derived Table 子查询 t 使用预先执行策略
EXPLAIN FORMAT=TREE
SELECT /*+ NO_MERGE(t) pq_distribute(t1 none) */ t1.a
FROM  t1, (
        SELECT /*+ subquery(pq_inline_evaluation) */  a
        FROM t2
) t
WHERE t1.a = t.a;
```

性能类特性
Bulk Load 快速导入模式

最近更新时间：2025-12-24 12:03:43

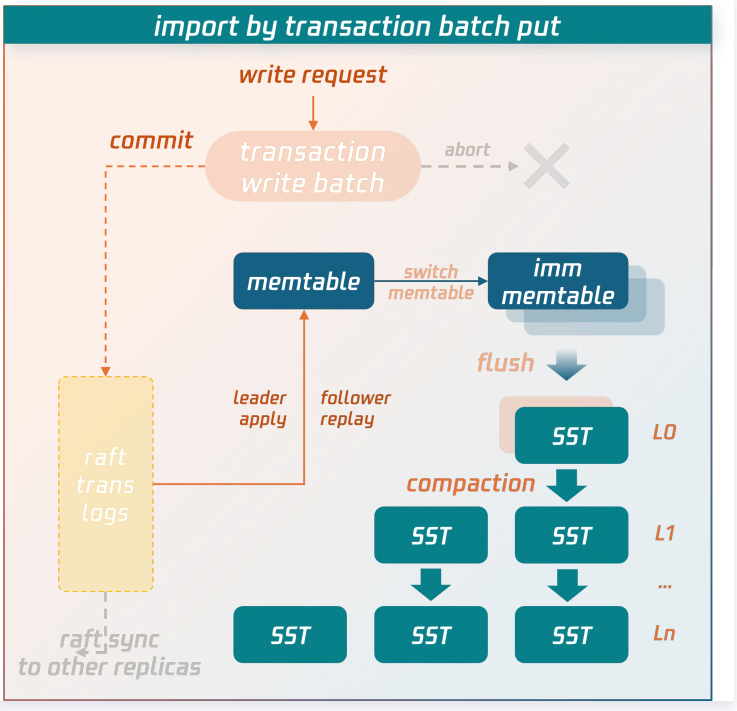
概述

TDSQL Boundless 支持以 bulk load 的方式向数据库快速导入数据。相比于执行传统 SQL 普通事务写入的方式，bulk load 快速导入模式在性能上通常提升5~10倍以上，适合在业务上线阶段用作迁移现有的大型数据库到全新的 TDSQL Boundless 集群。
目前支持用 `INSERT INTO / REPLACE INTO` 语句进行 bulk load 导入数据。

工作原理

普通事务写入流程

数据在事务提交前会暂存在事务上下文的 write batch 中。进入提交阶段，write batch 中的数据会先落盘形成事务日志，通过 raft 协议同步到各副本所在节点。形成多数派后，数据会先写入内存中的 memtable 结构中，memtable 攒满后会 flush 落盘形成 SST 数据文件。SST 文件以 LSM-tree 结构进行分层组织存放，通过后台异步线程执行 compaction 操作，将数据进行合并和清理。



Bulk Load 写入流程

Bulk load 导入的写入流程尽量绕过了普通事务较为冗长的写入链路，直接将数据写入到压缩的数据文件中。
具体流程：
1. 接收到 bulk load 事务写入请求的节点（源节点）会先将 SQL 语句解析和编码成 key-value（KV）。

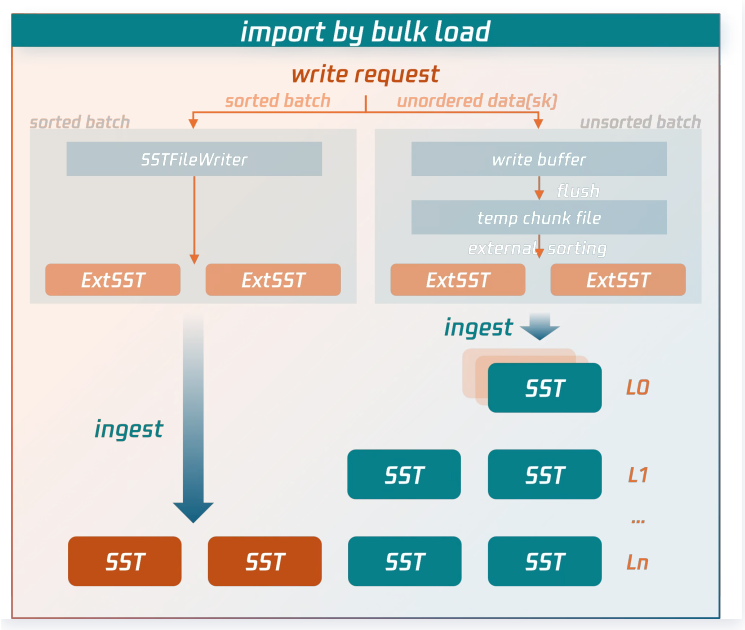
- 对于已有序的数据：直接写入到外部 SST 文件中。
- 对于无序的数据：先写到临时数据文件中，然后进行外部归并排序，生成外部 SST 文件。

2. bulk load 事务进入提交阶段后，源节点从 MC 控制节点拉取最新的 replication group（RG）路由信息，确定外部 SST 文件最终所属的 RG 及副本位置
3. 根据 RG 路由信息，源节点将外部 SST 文件发送到 RG 副本所在节点（数据节点）。
4. 确认外部 SST 文件被数据节点全部接收且校验成功后，源节点向 RG leader 数据节点发送 bulk load 事务提交请求，由 RG leader 数据节点同步一条 bulk load commit log（raft 日志），日志形成多数派后，通过执行或回放这条日志，RG 副本所在数据节点将外部 SST 文件直接插入到 LSM-tree 的合适位置。至此，数据写入到了数据库中。

与普通事务写入流程相比，bulk load 写入流程性能优化关键点：

- 不进行事务冲突检测。
- 无需将数据临时驻留在内存 write batch 中，而是直接落盘写入 SST 数据文件。
- 不需要先将数据落盘到事务日志并同步（注意：虽然 bulk load 事务也会写一条 commit log 用于维持主备数据写入一致，但并不需要将事务数据序列化到日志中，仅同步一些必要的元信息）。
- SST 数据文件无需通过 flush、compaction 操作由 LSM-tree 的最上层进入，而是直接将外部 SST 文件尽量插入到 LSM-tree 的尽量底层的位置。

这些 bulk load 所做的针对性优化，显著减少了 CPU、内存和 I/O 资源，是导入性能提升的关键点。



使用说明和限制

下面会介绍 bulk load 导入相关的参数、使用限制，以及性能相关的最佳实践。

相关参数

Session Variables

参数名	默认值	说明
tdsql_bulk_load	false	开启 bulk load 模式，符合条件的 SQL 语句会以 bulk load 的方式直接写入数据。
tdsql_bulk_load_allow_unsorted	false	bulk load 导入的主键数据是否支持乱序。当 tdsql_bulk_load_allow_unsorted 打开后，会对待导入数据进行额外的排序，使得导入速度降低。
tdsql_bulk_load_allow_sk	true	是否支持导入二级索引数据。 如果导入的表已经存在二级索引，那么该变量需要设置为 true，否则无法走 bulk load 优化（仍然走正常事务逻辑或者 batch put 优化）。当打开时，需要由数据源端来保证数据唯一性，即不存在主键冲突。如果存在 unique secondary index 时，也需要数据源端保证二级索引数据的唯一性。否则会导致主键和二级索引不一致，或者打破二级索引数据唯一性的约束。
tdsql_bulk_load_allow_insert_ignore	true	是否允许含有 ignore 语法的 SQL 语句（`INSERT IGNORE`）走 bulk load 优化。注意：当打开时，需要由数据源端来保证数据唯一性，即不存在主键冲突。因为 bulk load 模式不做主键冲突检查，即便允许 ignore 关键字，新数据仍会静默地覆盖掉旧数据，而不会遵循 ignore 语法（丢弃新数据保留旧数据）。
tdsql_bulk_load_allow_auto_organize_txn	false	是否开启 bulk load 事务自动攒批提交，由数据库自行控制 bulk load 事务数据到合适的规模后发起自动提交。 注意：该选项开启后，当客户端返回事务提交成功请求时，对应的数据可能还在攒批，未达到自动提交的数据量阈值，因此可能不可见。通常用于对性能要求更高的大规模数据迁移导入，且迁移阶段无读请求，对该阶段数据无实时可见性要求，导入完毕后达到最终一致状态即可的场景。

使用限制

bulk load 也并非数据导入 TDSQL Boundless 的“银弹”（No Silver Bullet）。总体而言，bulk load 导入模式是以一定程度上牺牲事务的 ACID 属性，来换取更高的导入性能。例如，bulk load 事务之间不做冲突检测，需要由使用者来处理好源数据中的数据冲突，或者用户需要知悉 bulk load 对于冲突数据的处理逻辑。以及，当 bulk load 事务提交失败时，如果涉及到多个 RG 的数据，可能出现半提交现象，即一部分 RG 的数据写入成功，一部分 RG 写入失败，这实际上违反了事务的原子性。因此遇到提交失败的情况时，重试成功后会达到数据最终一致。

此外，使用 bulk load 模式存在如下具体场景和限制。

语法层面：

- `SET SESSION tdsql_bulk_load = ON`，即 bulk load 模式需要打开。
- 仅优化 `INSERT INTO` / `REPLACE INTO` 语句，其他语句不支持 bulk load 模式。
- `INSERT INTO` / `REPLACE INTO` 形式的 SQL，必须有多个 VALUES。如果只有一行数据，不支持 bulk load 模式。
- `INSERT INTO SET` / `REPLACE INTO SET` 形式的 SQL 只能写入一行数据，不支持 bulk load 模式。
- 不优化 `INSERT INTO ... ON DUPLICATE KEY UPDATE`。因为 bulk load 模式下不会去检测已存在的 primary key，也就无法进行 `UPDATE` 操作。
- 对于 `INSERT INTO`，实际上执行的是 `REPLACE INTO` 语法。即当导入的新数据与旧数据存在主键冲突时，并不会报错，而是直接静默地用新数据把旧数据覆盖掉。
- 对于含有 `IGNORE` 语法的 SQL 语句（ `INSERT IGNORE INTO` ），需要提前 `SET SESSION tdsql_bulk_load_allow_insert_ignore = ON`，否则无法走 bulk load 导入。
- 对于含有 `IGNORE` 语法的 SQL 语句（ `INSERT IGNORE INTO` ），bulk load 导入时，需要由源端数据保证数据唯一性，即导入的新数据与旧数据不存在主键冲突，导入的新数据之间也不存在主键冲突。否则，由于 bulk load 模式下，主键数据会被静默地覆盖掉，只保留一行，这会违反 `IGNORE` 语法（即丢弃新数据保留旧数据）。

二级索引：

- 当表上有 secondary index 二级索引时，需要提前 `SET SESSION tdsql_bulk_load_allow_sk = ON`，否则无法走 bulk load 优化。
- 当表上有 secondary index 二级索引时，bulk load 模式下需要由源端数据保证数据唯一性，即导入的新数据与旧数据不存在主键冲突，导入的新数据之间也不存在主键冲突。否则，由于 bulk load 模式下，主键数据会被静默地覆盖掉，只保留一行，但是二级索引数据编码后的 key 是没有唯一性的，就会出现二级索引数据的新旧两条记录都被保存下来了，导致主键和二级索引不一致。
- 对于需要建立 secondary index 二级索引的表，在 21.x 或之后的版本，用 bulk load 导数据时，优先推荐建表时先不建立 secondary index。待导入完成后，再创建 secondary index，创建 secondary index 的过程可以开启 fast online DDL 优化（需要手动开启），性能上会更好，而且可以绕过“源端数据唯一性”的限制。
- 当表上有 unique secondary index 唯一二级索引时，需要由源端数据保证数据唯一性，即既不存在主键冲突，也不违反二级索引数据的唯一性。否则会导致主键和二级索引不一致，或者打破二级索引数据唯一性的约束。

其他限制：

- 系统表不支持 bulk load 模式。因为系统表不应该发生大规模写入，而且一旦系统表有问题，集群将无法启动。因此系统表采用更加稳健的写入路径。
- 存在触发器的表，不支持 bulk load 模式导入。
- bulk load 模式目前与 DDL 是互斥的。即表上若存在进行中的 DDL，不支持 bulk load 模式；若正处于 bulk load 模式下导数据，新的 DDL 请求会被拒绝。
- 通过 bulk load 模式导入的数据，不会生成 binlog，不会同步到灾备集群的备实例。

性能调优

为了使 bulk load 导入模式能够达到理想性能，最大化利用系统资源，我们将一些在实践过程中积累的调优经验总结罗列如下，供参考。

按序导入

按序导入是指数据按照主键从小到大的顺序去导入，数据库可以更高效地处理有序的数据。相对于乱序导入，按序导入性能提高20%~300%。
这里的按序，分为两个层面的有序。

- bulk load 事务内部有序：在一个 bulk load 事务内部，数据需要按照主键顺序排列。
- bulk load 事务之间可排序：将一个事务所写入数据的最小主键和最大主键作为这个事务的区间，如果事务的区间互相没有重叠，则认为事务之间是可排序的。换句话说，将两个事务所写入的数据，按照主键从小到大的顺序进行排序，不存在相互交叉的情况，则这两个事务是可排序的。

当以上两个有序的条件同时满足时，认为数据导入是完全有序的，性能最佳。
如果第一条不满足，即事务内部的一条条记录还没有按照主键顺序排好序，那么在导入之前，必须将参数 tdsql_bulk_load_allow_unsorted 打开，否则会失败报错。当 tdsql_bulk_load_allow_unsorted 打开后，每个事务中的数据会先暂存在临时文件中，在提交阶段对其进行外部归并排序，最后再按序输出到外部 SST 文件中。如果事务内部数据有序，则不需要该步骤，直接写到外部 SST 文件。因此事务内部无序要比有序消耗更多的 CPU 和 I/O 资源。
如果第二条不满足，那么事务和事务之间生成的外部 SST 文件会有区间重叠，当插入到 LSM-tree 时，会因为相互重叠导致在下层没有合适的插入位置，导致更多的 SST 文件被最终插入到最上层（level-0），会触发向下层的 compaction 操作，在 bulk load 数据导入期间，消耗一部分 I/O 资源，影响整体性能。

并发导入

当磁盘性能较好时（NVMe 磁盘），bulk load 性能瓶颈点很可能在 CPU 而不在 I/O，可以尝试适当增加导入线程的并发数（50~100）来提升导入效率。
系统支持通过 `tdstore_bulk_load_sorted_data_sst_compression`、`tdstore_bulk_load_unsorted_data_sst_compression`、`tdstore_bulk_load_temp_file_compression` 参数（需要超级管理员权限）调整 bulk load 导入过程中产生的外部 SST 文件和待排序临时数据文件所使用的压缩算法， 来平衡 CPU 和 I/O、磁盘空间。

```
SET GLOBAL
tdstore_bulk_load_sorted_data_sst_compression=zstd,snappy,lz4,auto,nocom
pression,...;
```

控制事务大小

与普通事务先把未提交数据暂存在内存的 write batch 不同，bulk load 事务会把未提交数据直接写到外部 SST 文件。因此大事务不会导致内存占用过多 OOM。
与之相反，bulk load 事务不宜过小，否则一方面会产生较小的 SST 文件，另一方面频繁的 bulk load 事务提交导致导入性能也不会好。

通常一个 bulk load 事务的数据量在200M以上，会达到比较好的性能。注意，外部 SST 文件默认是开启压缩算法的，以 ZSTD 为例，与原始数据相比，压缩率可能在10倍甚至更多，因此可以考虑让一个 bulk load 事务写入的原始数据量在2GB以上。

事务自动攒批提交

对于性能要求更高的大规模数据迁移导入，且迁移阶段无读请求的使用场景，也可以考虑开启 bulk load 事务自动攒批提交， `SET GLOBAL tdsql_bulk_load_allow_auto_organize_txn = ON;` 由数据库自行控制 bulk load 事务数据到合适的规模后发起自动提交，从而维持较高的导入性能。但需注意，该选项开启后，当客户端返回事务提交成功请求时，对应的数据可能还在攒批，未达到自动提交的数据量阈值，因此可能不可见。建议导入完毕后进行行数校验。

二级索引

对于表上存在二级索引的场景，表上的二级索引越多，bulk load 性能会变差。这是因为二级索引数据一定是无序的，需要进行额外的排序操作，从而增加了导入过程中的计算开销。针对存在二级索引的场景，优化建议适当增大 bulk load 事务的大小。
在21.x或之后的版本，更为推荐的方式是：用 bulk load 导数据时，考虑建表时先不建立二级索引。待导入完成后，再创建二级索引，创建二级索引时开启 fast online DDL 优化（需要手动开启），整体性能上会更好。

分区表

从计算层看，每个分区表会分配一个 unique t_index_id。实际处理逻辑跟处理一张普通表是类似的。需要考虑一个 bulk load 事务跨多个分区表的场景。

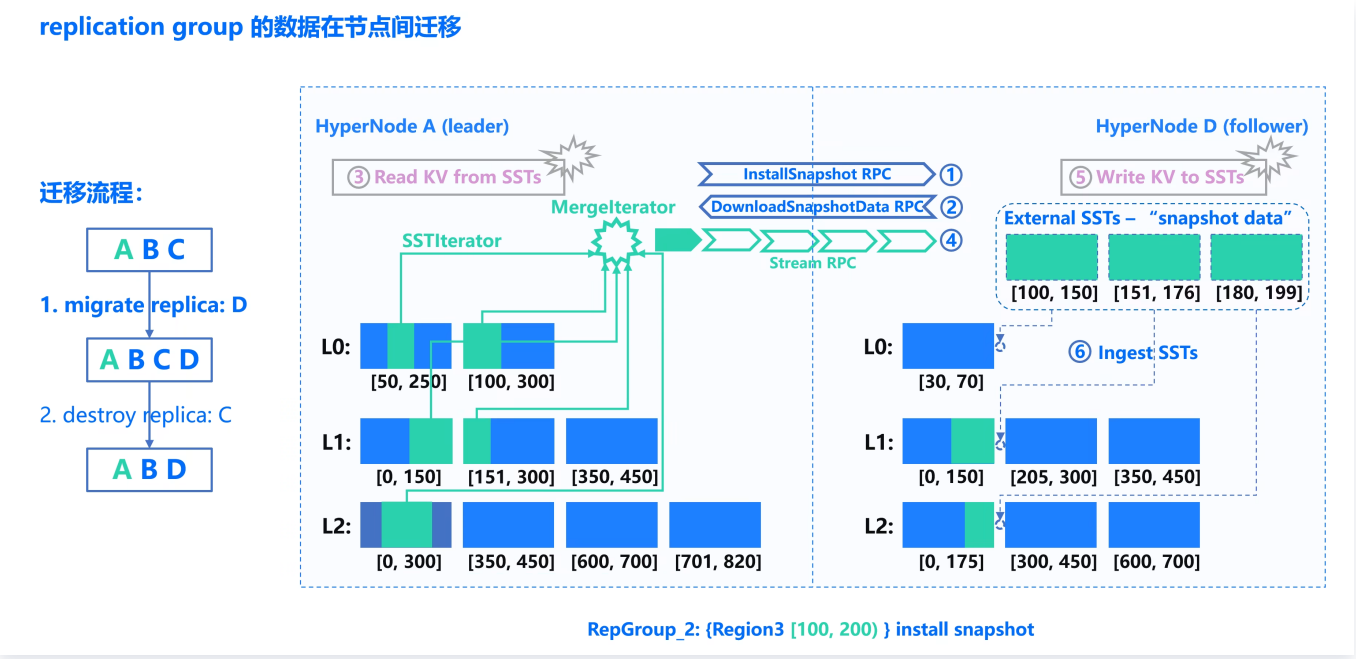
- 如果是 range 分区，且按序导入，那么一个 bulk load 事务中的数据通常会集中在一个 range 分区上，或者少量 range 相邻的分区上。此时一个 bulk load 事务通常不会涉及太多的分区表，跟普通表的场景类似，不需要做额外处理。
- 如果是 hash 分区，且按序导入，那么一个 bulk load 事务中的数据会被均匀地打散到各个 hash 分区上。此时一个 bulk load 事务会几乎涉及所有的分区表。此时需要把 bulk load 事务大小继续调大，从而保证一个 bulk load 事务中分配到每个 hash 分区上的数据量不要太少。假设一共 N 个 hash 分区，那么推荐 bulk load 事务大小比普通表场景扩大 N 倍。目前在分成16个 hash 分区的场景下，性能要比不进行 hash 分区（普通表）损失15%左右。
- 如果存在二级分区，需要考虑一个 bulk load 事务中的数据会涉及多少个二级分区，最终分配到每个二级分区上的数据量不要过少。

总而言之，如果一个 bulk load 事务涉及的分区数量越多，性能会逐渐退化。从存储层去看，其实就是一个 bulk load 事务涉及了过多的 RG 作为事务参与者。类似于普通事务也会有类似的问题，涉及的分区越多，RG 层面的参与者越多，性能越差。

副本迁移性能优化

最近更新时间：2025-12-24 12:03:43

传统迁移流程

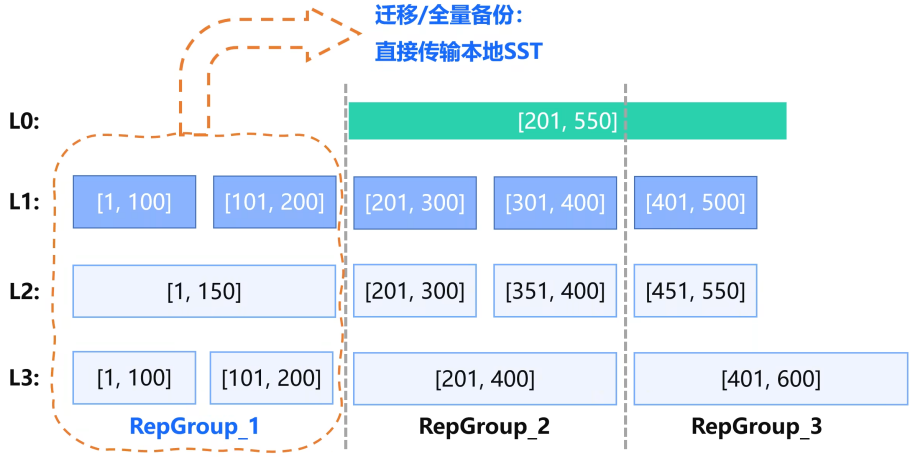


1. **数据提取阶段**：从 Leader 的 LSM-Tree 中遍历提取特定 key range 的数据。数据可能分散在不同层级、多个 SST 文件；需要逐条扫描和筛选，IO 开销大。
2. **数据传输阶段**：逐条发送 KV 数据到目标端，传输效率低，网络利用率差。
3. **数据写入阶段**：目标端先写入 external SST，再 ingest 到 LSM-Tree。需要与本地 SST 文件聚合，产生额外压缩开销。
4. **副本删除阶段**：从 LSM-Tree 中移除数据，需要遍历底层 SST 文件，IO 开销大。

优化方案：SST 文件边界对齐

存储层数据文件SST与replication group对齐 – 提升迁移性能

- **优化前：**从 src node 中一条条读取 kv 数据并发送
- **优化后：**直接传输压缩后的 SST 文件



在底层数据压缩（compaction）过程中，增加对 SST 文件和数据分片 Region 边界对齐的逻辑处理。

- **实现机制**
  - 在后台执行 compaction 过程时，系统会调用名为 SST partitioner 的模块。在 compaction 检测到文件较大时，按 Region 边界进行拆分，确保同一 SST 文件仅包含单一数据分片的数据，而不会将不同 region 的数据合并在一起。
  - 如下图所示，L1层：100-200, 200-300, 300-400等范围清晰分离；L2/L3层：无数据跨越 Region 边界。
- **优化后的迁移流程**

  1.1 **文件直接识别：**无需遍历数据，直接确定 Region 对应的 SST 文件。

  1.2 **整文件传输：**直接传输压缩后的 SST 文件到目标端。

  1.3 **直接 ingest：**目标端无需复杂压缩，直接插入数据。
- **技术细节处理：**迁移前先将 L0 层乱序数据 compaction 到 L1 层。由于 L0 层数据由 memtable 刷写而来，其数据量相对有限，压缩开销可接受。

性能提升

- compaction 相关 IO 开销下降。
- 迁移、全量备份性能提升。
- 区间物理删除性能提升。
- 为批量导入（Bulk Load）功能提供了技术基础，实现了 SST 文件的高效传输和加载，显著提升了 TDStore 的数据迁移效率。

方案	速率	备注

stream RPC 传输 KV (优化前)	22.6M/s	每个 RepGroup 数据量20G
直接发送本地 SST (优化后)	455.1M/s	

自适应热点锁管理器

最近更新时间：2025-12-24 12:03:43

事务优化：自研事务并发控制系统（适配高弹性的 RG 架构）

- **自适应数据热点的锁管理器**：在 LSM-tree 上高效支持悲观锁，锁管理器可以随 region 一起分裂，有效打散热点瓶颈。
- **分布式死锁检测算法**：采用了开销最低的集中式死锁检测，以确保性能最优。

传统方案及其局限性

- 业内常规手段：使用固定个数的 hash lock map 来管理事务锁等待。
- 存在两个问题：
  - 高弹性架构下，事务锁迁移效率低。
  - 不支持谓词锁，导致在 RR（可重复读）隔离级别下会出现幻读。

TDStore 数据分片联动方案

- TDStore 采用 sort lock map，支持范围段加锁。
- 同时绑定 region 分片，锁管理器可以随 region 一起分裂。