# 概述

## 对比

VS MySQL + innodb:

数据量不需要预估，支持百节点的集群，按需要加节点。

数据压缩比高。压缩率是innodb的3- 20倍。

内部支持分布式事务，用户无感知。

结合存储引擎的WAL日志复制，回放延迟低。

Schema online change支持更多，比如减列，改列类型，修改字符集等。

相比MySQL 的MGR，多写性能更好，更加稳定。

VS TiDB：

MySQL兼容性做的更好，兼容MySQL handler层实现，原生支持MySQL所有的功能

对于水平分表的业务支持更好，相对于TiDB key value无规则分布，可以有效减少二阶段事务，远程回表等操作，时延更稳定。

Schema online change支持的 操作更多。

支持对称节点部署，不区分存储节点和计算节点，支持小型化部署，对于计算下推，分布式执行更友好。

## 核心能力

1、MySQL兼容

1）通过扩展 MySQL handler功能，支持mysql所有的功能

2）通用的schema online change 框架。支持schema变更不停业务。比MySQL支持更多的schema online change操作，与pt schema online change相比，解决了他存在的bug，DDL操作的原子性由数据库内核完成，不需要人为干预。

3）binlog导入导出。(全局binlog导出在2022年规划)

2、可扩展性

1）支持pb级数据，从物理资源来看，计算资源，内存，存储都支持扩展，设计上不存在扩展的瓶颈点。

2）所有节点都是对等的，都可以访问全局的数据，都支持读写的workload。

3）伸缩能力：可以按需增加，减少计算节点，存储节点。

4）完整的分布式事务能力，让用户像使用单机MySQL一样使用TDSQL3.0

5）支持百级节点的集群管理能力

节点管理

连接收敛，支持高并行，为性能服务。

时间戳服务，支持千万级/秒的时间戳申请服务。

3、性价比

存储的压缩率高，是innodb的3-20倍。

可以支持按需售卖，计算节点和存储节点都可以按需售卖。

4、性能

并行计算+下推，支持大查询的性能优化。

数据分片分裂不杀事务，避免数据变迁过程中影响性能。

数据物理位置感知：针对水平分表的业务避免做二阶段提交，索引回表，可以把更多的计算下推到存储节点。

5、高可靠

计算层无状态。

存储层实现multi\_raft协议实现多副本的高可靠。

数据迁移过程中，实现数据块分裂，迁移的原子性保证。

# 架构

图示

描述已自动生成

## 多主架构

## HTAP方案

### 一体化方案

### 松耦合方案

# 原理

## SQLEngine

### 概述

### 功能

#### Online-DDL

#### binlog

### 优化

#### bRPC网络框架改造

##### 改造方案

1、网络框架

1）在brpc中增加和mc 时间戳通道交互的协议TDGts protocol

2）将sqlengine和tdstore，mc的网络交互模式重构为brpc的channel模式

2、和前端mysql client的交互

1）增加bthread的启动模式，一个connection对应一个bthread

2）和mysql client的socket有网络交互的时候要使用非阻塞io，bthread能主动切走并在对应的fd上有event的时候自动切回来

3、锁，条件变量相关

将mysql mutex，condition variable相关的宏定义用bthread mutex，condition variable替换

4、thread local

1）将current\_thd，THR\_MALLOC这种用于原本的线程级参数传递的变量修改为bthread local

2）自增值cache在和tdstore交互获取自增值的方面使用了thread local并显示调用了libco的代码，整体重构为hash bucket+mutex的模式

3）open ssl中有thread local，mysql默认将open ssl的mutex注册为了mysql\_mutex xxx，而mysql mutex又被我们替换为了bthread mutex，这意味着open ssl内部使用mutex的时候可能产生切换，bthread被调度到其他pthread会导致thread local访问异常crash，需要取消mysql像open ssl注册修改mutex的行为

4）region cache使用了thread local，替换为hash bucket + mutex形式

5）storage/temptable下面的thread\_local uint8\_t \*shared\_block，thread\_local Tables tables按照bthread local改造

6）storage/perfschema/pfs.cc中的thread\_local PFS\_table\_context \*THR\_PFS\_contexts[THR\_PFS\_NUM\_KEYS]按照bthread local改造

7）sqlengine的百分位统计使用了thread local，考虑直接用bvar替换

5、对brpc channel的资源管理

mc域名对应的ip变更后需要重新初始化channel并更新到全局，这个过程中要保证和获取channel之间的线程安全并且不能有额外的性能损耗

tdstore ip变动后需要清理掉对应的无效channel

6、现有代码中的阻塞api以及显示的libco调用

1）目前代码中显示的poll调用会阻塞线程，需要替换为bthread可切换的模式，目前看这类调用并不多，多数为用于等待超市的功能，例如select sleep(5)，可以逐个手动修改

2）sqlengine互联使用了mysql client同步api，将server对fd和bthread的切换关联行为也应用在client lib中

3）并行框架中的libco调用需要改为bthread相关的语义

##### 参数

1、变量持久化在mc后，需要处理grpc通道的channel初始化的参数获取问题（这是一类问题，包括—version的时候目前也会访问到没初始化好的参数）

2、和channel管理相关参数可配，方便性能调优，tdstore通道的连接数需要增加新的配置并且默认为1，brpc连接数过多容易性能下降

3、brpc中的负责处理工作线程和fd唤醒的epoll线程数改为可配，方便性能调优

4、网络相关的参数命名需要修改，也要通知云平做同步修改（这里需要考虑在之前的版本通过mc持久化的变量如果到了新的版本变量不存在了会导致sqlengine无法启动，是跳过还是升级的时候通过mc的接口先做修改再升sqlengine）

##### 优化

1、不同类型的rpc设置独立的超时时间（比如get record超时应该比较短，count下推的rpc超时时间应该比较长）

2、通过sql查询bvar的统计信息/开发brpc的http端口查询bvar统计，方便性能分析

3、优化ebpf 获取rpc时耗的方式，降低ebpf trace的时候带来的性能损耗

##### 其他

1、sql执行过程中定期查看前端和client连接的socket是否正常，不正常需要设置对应thd的killed标志，防止sql在部分连接断开的情况下又”复活“修改数据的行为

2、清理和thread\_local mysql\_mutex\_lock\_num相关的调用

3、判断是否是后台线程的时候不能用thd->thread\_id,CurrentThdHelper中的set\_new\_thread\_id需要加回来，可以考虑用thd->is\_extra\_thread替代

4、调通unittest

5、确认iptable封网后brpc的恢复时间

6、升级brpc到1.0.0版本

7、移除libco相关的代码

#### 查询优化

#### 事务请求优化

原只读事务流程：

文本

低可信度描述已自动生成

原读写事务流程：

图形用户界面

描述已自动生成

##### 问题

原事务流程存在的问题：

1、对于oltp请求来说（比如sysbench的oltp\_read\_write）

1）读相关的请求会开启子事务，commit时参与者数量增多导致性能下降

2）StartParticipant和EndParticipant两个rpc没太大意义但是存在额外的网络开销

2、对于查询并行优化来说，并行执行的每个子任务需要关心互相之间动态开启的事务是否有重叠

##### 优化方案

1、去除StartParticipant和EndParticipant两个rpc

2、对于put/delete/batch put等写入请求的处理方式

1）SQLEngien请求的时候会先查看参与者列表，如果没有开启过事务则发起rpc的时候会携带“先开启事务的语义”，如果已经开启过事务会携带“已经开启过事务”的语义

2）TDStore根绝SQLEngine发送的是否开启事务语义决定是否要先开启事务再执行rpc请求

3）SQLEngine接收到TDStore写入成功的应答则将事务参与者加入到SQLEngine侧的事务参与者列表中

3、对于get，scan，batch get等读请求如果是带锁语义或者是在同一个事务中执行过写入操作之后才执行的则和写入请求处理保持一致（之所以要求有过写操作则读请求也需要携带“是否已经开启事务”的语义是为了防止TDStore被动切主以后子事务可能丢失需要对读请求报错给SQLEngine事务不存在，否则可能导致用户在同一个事务中读不到自己写入的数据）

4、对于get，scan，batch get等读请求不带锁语义的tdstore不需要判定参与者是否已经开启事务，直接用sqlengine在读请求中携带的时间戳读取即可，SQLEngine的scan请求自己会用last\_fetch\_key确定每次scan的起始点无需TDStore保存事务上下文

5、scan record的response可以携带scan的数据，不要等到scan next的时候才返回数据

6、由于读请求不会在TDStore上开子事务，单纯依靠TDStore上报的最小snapshot ts所保证的snapshot有效性就不可靠了，这里需要SQLEngine侧也上报给mc活跃事物最小的transid，保证TDStore不会在事务读期间将本能读到的snapshot过早的purge掉

**该优化带来的好处：**

1、对于oltp请求来说（比如sysbench的oltp\_read\_write）

1）读相关的请求不会开启子事务，可以大大减少commit时分布式事务参与者的数量

2）少了StartParticipant和EndParticipant两个无效rpc，减少网络交互

2、对于查询并行优化来说，无论是下推的并行还是单机内的多线程并行，并行执行的每个子任务不用关心互相之间动态开启的事务是否有重叠，专注在并行执行本身需要处理的问题上即可，只需要带着同样的时间戳读取，无需关心子事务开启关闭之类的问题（注意如果同一个事务中已经执行过写入操作，则后续的读请求不要开启并行）

## SQLEngineAgent

## TDStore

### 概述

### 原理

#### RocksDB主流程

#### MyRocks编解码

#### MyRocks Insert Update编码

#### MyRocks Select编解码调用栈

#### MyRocks的Record格式

#### Region创建与分裂

Region是TDSQL中TDStore向MC和sqlEngine暴露的最小数据存储单元，一个Region代表着一个三副本的raft group（副本数当然也是可配的，这里先用三副本作为例子），每个副本称为该region的一个peer，三副本的配置下，按照一主两备的方式提供服务。

一个Region中所存储的数据的key均落在该region的 [start\_key, end\_key) 区间内, 随着数据的不断插入，一个region的总size会不断变大，所包含的记录条数总量会变大，为了分散在同一个region上的并发读写，region需要分裂，即split。Region split的触发条件有两个（满足其中一个即可分裂)：

region 中的记录条数超过“记录条数阀值”

region 的总大小超过“体积阀值”

##### 集群的初始化以及Region的创建

##### Region分裂

#### RocksDB Compaction

### 功能

**1、数据存储与多副本一致性**

数据存储：持久化KV数据，需要满足高性能读写，直接采用了开源rocksdb （已实现）（存在需要持续优化项，如：大value的情况下kv分离、数据文件边界和表边界对齐等）

多副本一致性：分布式系统需要解决单点问题，一份数据会存储在多个节点上（通常是3副本），在一个节点故障的情况下，能快速找到其它节点继续提供无损服务，直接采用了开源braft （已实现）

数据存储与多副本一致性结合：Raft共识算法会把上层逻辑抽象成日志状态机，我们需要解决的就是如何把rocksdb变成Raft日志状态机，同时还要解决log compact时全量镜像如何产生与全量数据拷贝的问题 （已实现）

learn角色支持：可以优化Raft成员变更的一些流程以及实现一些部署形态上的功能（已规划，待实现）

支持列存（待规划）

**2、分布式事务**

原子性：一个事务涉及多个节点的数据要么全部成功，要么全部失败。采用了自研的协商式两阶段提交，在提交时协商式两阶段提交相比其他类型的两阶段提交可以减少等待的层级，在事务时耗上存在一定优势 （已实现）

隔离性：即事务并发控制，从最基本的事务不能出现脏读、脏写，到可重复读，再到可串行化。 在rocksdb提供的MVCC和乐观事务模型的基础上，引入了范围冲突的概念自研提供了可串行化隔离级别的支持，最终支持Read Commit/Repeated read/Serializability 三种隔离级别供上层选择 （已实现）（存在下一步功能，悲观事务模型）

全局一致性读：A/B两个事务并发执行时，如果事务A在节点1上读到了事务B提交的数据，那么在其他节点上也要读到事务B的数据。我们引入了全局时间戳服务（MetaCluster Service提供）提供全局单调递增的逻辑时间戳，并把这个逻辑时间戳替换掉了rocksdb原生的Sequence(单机本地计数)，并改造了rocksdb原生的事务可见性判断以及compation过程中删除过期数据的逻辑，因为全局一致性读意味着某个全局事务即使当前没有在本机上读取过，不代表未来不会读，因此需要有机制保留该事务可见的版本 （已实现）

分布式事务与Raft结合：两阶段提交把事务拆成了多个日志Redo/Prepare/Commit/Clear，不同事务之间的这次日志会交错在一起，这给TDStore停机后数据恢复和Raft log compact带来了难题。我们通过在rocksdb将内存中的数据刷入磁盘时，才触发Raft log compact，并使用计算出的活跃事务最小Raft log index做日志删除点位，来保证恢复的安全性，并在恢复时合理跳过已经刷到磁盘的被截断的事务日志，保证了各个模块的紧密且协调的配合。（已实现）

learn节点一致性读：可以提升集群在OLAP上或者大规模读的能力（待规划）

**3、数据分片管理与调度支持**

在TDStore中数据是分片管理的，每个数据分片是一段数据范围，我们称之为Region。每个Region默认保持在256M左右。数据分片的目的是为了方便数据调度，拆解热点。

Multi Raft支持：每个Region就是一个Raft组，这样会方便调度，依赖开源braft的支持，但是开源braft每个组会产生一个日志文件，在TDStore可能会有成千上万的Region的情况下，会影响写入性能，原因是文件太多，顺序写变成随机写了，因此TDStore将其进行了改造，把所以所有Raft组的日志统一提交到rocksdb里面，保持了顺序写的特点 （已实现）

Region分裂：如果业务集中的写一段范围，那么这段范围对应的Region的数据量就会持续膨胀，到达阈值之后就要分裂。Region分裂是数据范围的逻辑分段，一个Region会变成两个Region，这个过程中要尽可能小的影响事务，这里存在非常多的细节 （已实现）（目前已实现分裂不杀事务，但是对事务提交的时耗还存在影响，需要持续优化）

Region迁移：Region从一个TDStore节点迁移到另外一个TDStore节点，通过该功能可以实现TDStore的扩容和负载均衡。通过Raft 成员增减来实现的，难点在于回迁和各种并发处理。（已实现）

Region合并：如何业务会删除历史数据，那么有些Region的容量可能就会变小甚至为空，需要把这些Region合并回收，减少元数据的同时避免（已规划，待实现）

Region热点监控和上报：可能会存在某些Region或者某些TDStore比其他Region或者TDStore读写流量高很多的情况，TDStore必须具备将这些信息收集并上报到MetaCluster Service的能力，供其进一步调度用。（已初步实现，需要持续优化）

**4、数据形态与地理感知（实现中）**

地理感知：向用户提供副本位置，副本个数等控制的api，让用户通过数据分布策略来控制数据的位置，让数据满足 zone/Available Zone/region 级别的可靠性。

形态感知：在TDStore和Region之间引入了一层中间层，以中间层为一个Raft组。这样对于可以水平拆分的业务，结合分区表，把一起访问的数据放在TDStore与Region的中间层上，可以把二阶段提交算法优化至一阶段提交，结合算子下推技术技术，把回表join等操作下推至存储层完成，大幅提升数据库的性能。

**5、OLAP支持与资源管理**

计算下推支持：配合SQLEngine完成计算下推，极大的减少数据中间传输的消耗。（已初步实现，正在实现更多情况的下推，还要支持元数据缓存的能力）

资源管理：OLAP的查询会比较消耗cpu和IO，如果没有隔离管理能力，在现网上可能会出现OLAP类的查询对OLTP类的查询影响极大的情况（已规划，待实现）

**6、数据库生态支持**

数据备份和恢复：分布式集群下的备份和恢复，需要专门的工具来支持（全量数据的备份和恢复已实现，正在优化速率和故障容忍能力，增量数据备份和恢复待binlog实现后完成）

binlog：配合SQLEngine，在TDStore learn节点使用raft log产生binlog （已规划，待实现）

快速数据导入工具的支持：当前数据导入依赖DTS和mydumper等工具，而且这些工具还需要DBA手动设置一些参数，使用上不是很方面。同时这些工具的导入速率也不够快。（已规划，待实现）

#### Online-DDL支持

#### 事务

#### 存储

#### 分裂切主迁移

#### Region状态机

#### 内存控制

#### 备份恢复

### 工具

#### TDStore Client

TDStore的客户端工具，支持集群中单个TDStore的状态信息查询以及操作，包括tdstore状态查询、region状态查询、transaction操作以及状态查询等功能。

#### TDStore Dashboard

TDStore的全局客户端工具，支持集群中所有TDStore的状态信息查询以及操作，包括tdstore状态查询、region状态查询、transaction操作以及状态查询等功能。

#### TDStore Raft Log Dump、修改工具

TDStore的raft log解析工具，支持查询某个region中的所有raft log的详细内容。

#### TDStore BRPC Web Service

TDStore使用brpc进行TDStore间的远程通信，rpc\_view可以方便的访问TDStore中brpc服务提供的相关统计值的历史趋势和动态曲线

#### TDStore td\_option\_update

TDStore 使用td\_option\_update 动态更新TDOptions的配置项。

#### TDStore Client New

TDStore的客户端工具，支持Single以及Global两种模式，提供tdstore状态查询、region状态查询、transaction操作等功能。

#### TDStore数据一致性校验工具

用于校验TDStore多个副本之间数据是否一致

#### TDStore接口压测工具

memory\_benchmark\_main key value接口的压测工具

region上的读写请求压力监控工具：

perf\_monitor 监控region上的读写请求压力

perf\_region.py监测各个region的热度情况（qps）

#### Rocksdb工具

## TDStoreMonitor

## TDMetricsCluster

### 概述

TDMetaCluster（简称MC）是TDSQL3.0中的管控平面模块，主要功能特性划归为「对内」和「对外」两大类：

对内：MC 提供全局唯一事务时间戳，管理 SQLEngine 和 TDStore 节点元数据，基于管控策略调度分布式存储层的数据分布及形态变迁；

对外：MC 向集群层面的管控层提供实例级别的任务管理接口和能力，对接可视化运营模块提供实例层级的数据可视化 Dashboard 能力；

### 功能

#### 全局唯一事务时间戳

**能力现状：**

MC可提供千万级QPS的时间戳服务，在关键路径上做了深度优化，极限性能压测场景下，单请求延时（<1ms）大幅低于TiDB的时间戳服务延时（6-7ms）；

**未来规划：**

MC支持多节点跨AZ部署下的就近访问，消除单点时间戳服务的限制；

#### 数据资源管控与调度

**能力现状：**

MC作为实例管控平面，可以支撑单实例百万级别数据分片的元数据管控；

已实现对数据资源的创建、分裂、迁移、删除、切主等一系列的任务调度；

已具备对存储层的基于数据物理特征的数据量均衡以及 Leader 均衡；

已支持计算/存储节点的动态扩缩容，并实现数据自动迁移和动态均衡；

**未来规划：**

感知数据的逻辑层面特征，理解并维护每个数据分片的 物理↔逻辑两个层面的映射；

基于数据形态的感知，以及粒度更精细的workload 统计，建立更智能和多维的调度模型——最终用户侧效果表现为，能动态提升实例的整体性能；

实现通过工具接口或可视化运维界面，令高阶用户具备个性化管理数据调度规则的能力，实现“用户态”的资源隔离；

实现基于联合共识（Joint-Consensus）和 Learner 角色的副本成员变更，解决迁移过程中的可用性问题；

#### 实例级管控

**能力现状：**

MC 已具备基本实例任务管控能力，包括实例隔离、备份恢复、动态扩缩计算/存储节点等等；

**未来规划：**

跨 AZ 支持，单实例「同城双中心」部署下，N 副本全同步和 Majority 模式切换；

支持跨集群灾备实例关系的建立、切换、解除；

#### 自身高可用

**能力现状：**

MC为分布式组件，节点间数据同步以及Leader选举由内嵌的etcd所提供的能力保证；

生产环境中运行的实例来看，MC未发生过由于进程原因导致的切主；

**未来规划：**

支持定期导出、备份、重导入历史数据，结合etcd Compaction ；

MC 数据彻底丢失、损坏等极端场景下，支持从TDStore重建数据；

### ETCD路径

#### 全量备份任务

1、任务状态-put and get

/<cluster\_group>/<cluster\_id>/tdbr/backup\_task\_status/<task\_id>

2、向tdstore agent发布子任务-put

/<cluster\_group>/<cluster\_id>/tdbr/backup\_task\_meta/<task\_id>

3、tdstore agent监听子任务-watch with prefix

/<cluster\_group>/<cluster\_id>/tdbr/backup\_task\_meta/

4、子任务结果发布-put and get（一个region对应一个子任务）

/<cluster\_group>/<cluster\_id>/tdbr/<task\_id>/backup\_subtask\_status\_<region\_id>

#### 增量备份任务

1、向 tdstore agent 发布子任务 - put

/<cluster\_group>/<cluster\_id>/tdbr/incr\_backup\_task\_meta

2、查询 tdstore agent 增量备份进行状态 - put and get

/<cluster\_group>/<cluster\_id>/tdbr/incr\_backup\_status\_<ip:port>\_<region\_id>

#### 恢复任务

1、任务状态-put and get

/<cluster\_group>/<cluster\_id>/tdbr/recover\_task\_status/<task\_id>

2、向tdstore agent发布子任务-put

/<cluster\_group>/<cluster\_id>/tdbr/recover\_task\_meta/<task\_id>

3、tdstore agent监听子任务-watch with prefix

/<cluster\_group>/<cluster\_id>/tdbr/recover\_task\_meta/

4、子任务结果发布-put and get（一个region对应一个子任务）

/<cluster\_group>/<cluster\_id>/tdbr/<task\_id>/recover\_subtask\_status\_<ip:port>\_<region\_id>

#### MC备份任务

1、任务状态-put and get

/<cluster\_group>/<cluster\_id>/tdbr/mc\_backup\_task\_status/<task\_id>

2、向mc agent发布子任务-put

/<cluster\_group>/<cluster\_id>/tdbr/mc\_backup\_task\_meta/<task\_id>

3、mc agent监听子任务-watch with prefix

/<cluster\_group>/<cluster\_id>/tdbr/mc\_backup\_task\_meta/

4、子任务结果发布-put and get（实际上只有一个子任务）

/<cluster\_group>/<cluster\_id>/tdbr/<task\_id>/mc\_backup\_subtask\_status\_leader

#### MC恢复任务

1、任务状态-put and get

/<cluster\_group>/<cluster\_id>/tdbr/mc\_recover\_task\_status/<task\_id>

2、向mc agent发布子任务-put

/<cluster\_group>/<cluster\_id>/tdbr/mc\_recover\_task\_meta/<task\_id>

3、mc agent监听子任务-watch with prefix

/<cluster\_group>/<cluster\_id>/tdbr/mc\_recover\_task\_meta/

4、子任务结果发布-put and get（一个mc对应一个子任务）

/<cluster\_group>/<cluster\_id>/tdbr/<task\_id>/mc\_recover\_subtask\_status\_<ip:port>

## TDBR

# 事务

# 数据分布

# 备份恢复

# 兼容性

# 扩展性

# 高并发

## 计算引擎高并发

### bRPC网络框架

### 事务请求优化

### 计算下推

#### limit condition push down

#### limit offset condition pushdown

#### Join下推

#### ICP下推

#### 通用下推

#### 聚合下推

#### 回表下推

### 并行优化

#### Region并行

#### 单表scan并行

#### 并行查询

## 存储引擎高并发

### 独立线程处理braft心跳并打包

### raft log并行回放

### region follower直接回放WriteBatch格式的write\_set

### region leader 并发apply

### TDStore ebpf跟踪点规划与实现

### 自增值性能优化

## 元数据管理高并发

# 高可用

# 数据压缩

# 数据迁移

# 运维

# 应用场景