# 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

#### DDL

##### CREATE TABLE

表元数据以key-value的形式存储在tdstore上，key为库表名，value为pb格式的表元数据

1. 根据create table sql，库表名拼接成Key（db\_name+tbl\_name），尝试从本地Region cache查询key所在的region
2. 本地region cache获取到key所在的region->5，本地region cache查询不到key所在的region->3
3. 通过协议GetRegionByKey从meta-cluster处获取此key应该处于哪一个region
4. 可能异常：GetRegionByKey返回错误码（非OK）/校验GetRegionByKey回包的cluster id和SQLEngine.tdsql\_cluster\_id不一致，报错，结束；网络问题收不到GetRegionByKey回包，重试3次（硬编码，有需要可做成参数配置）仍收不到回报，断开客户连接，结束；没有发生异常->5
5. SQLEngine通过协议GetTableSchema，从该region上尝试get该库表的元信息记录
6. 可能异常：GetTableSchema返回错误码（非OK）/校验GetTableSchema回包的cluster id和SQLEngine.tdsql\_cluster\_id不一致/校验GetTableSchema回包的key值和请求的key值不一致，报错，结束；网络问题收不到GetTableSchema回包，重试3次（硬编码，有需要可做成参数配置）仍收不到回包，断开客户端连接，结束；没有发生异常->7
7. GetTableSchema返回keyNotFound，get不到此库表的记录，则集群中不存在此库表->8，否则集群中存在此库表->12
8. 集群中第一次创建此库表，通过协议GenerateIndexID从meta-cluster获取表的index id
9. 可能异常：GenerateIndexID返回错误吗（非OK）/校验GenerateIndexID回包的cluster id和SQLEngine.tdsql\_cluster\_id不一致，报错，结束；网络问题收不到GenerateIndexID的回包，重试3次（硬编码，有需要可做成参数配置）仍收不到回包，断开客户端连接，结束；没有发生异常，index id用于在sqlengine内部创建表对象
10. 将此库表的元数据编为key-value格式，通过协议PutTableSchema写到region
11. 可能异常：PutTableSchema返回错误码（非OK）/校验PutTableSchema回包的cluster id和SQLEngine.tdsql\_cluster\_id不一致，报错，结束；网络问题收不到PutTableSchema回包，重试3次（硬编码，有需要可做成参数配置）仍收不到回包，断开客户端连接，结束；没有异常->结束，返回客户建表成功
12. GetTableSchema协议返回元数据，key-value为“db\_name+tbl\_name，pb形式表元数据”
13. 解析pb形式的表元数据，根据这个元数据，此sqlengine内存建表对象，返回客户建表成功

##### ALTER TABLE/RENAME TABLE

基本流程是和原生流程一致的，在alter table过程中会将临时表结构也存入TDStore，在修改完成后再把这个临时结构删除。

##### ANALYSE TABLE

同原生MySQL8.0。

##### CHECKSUM TABLE

同原生MySQL8.0。

##### TRUNCATE TABLE

删除表中所有数据，但保留表结构。

#### DML

##### INSERT

基本流程：

1. 根据insert sql拼接出key-value格式的数据行，尝试从从本地的region cache获取key落在哪一个region上
2. region cache获取到key所在的region->5，region cache没有缓存key所在的region->3
3. 通过协议GetRegionByKey从meta-cluster获取key所在的region
4. 可能异常：GetRegionByKey返回错误码（非OK）/校验GetRegionByKey回包的cluster id和SQLEngine.tdsql\_cluster\_id不一致，报错，结束；网络问题收不到GetRegionByKey回包，重试3次仍收不到回包，断开客户连接，结束；没有发生异常->5
5. SQLEngine通过协议GetRecord从region上尝试get即将插入的key值
6. 可能异常：GetRecord返回错误码（非OK）/校验GetRecord回包的cluster id和SQLEngine.tdsql\_cluster\_id不一致/校验GetRecord回包的key值和请求的key值不一致，报错，结束；网络问题收不到GetRecord回包，重试3次仍收不到回包，断开客户连接，结束；没有发生异常->7
7. GetRecord返回keyNotFound，get不到key值的记录，表中不存在重复键值—>8，否则表中已有该key值，返回客户Duplicate key错误，结束
8. 通过协议PutRecord把key-value写到region上
9. 可能异常：PutRecord返回错误码（非OK）/校验PutRecord回包的cluster id和SQLEngine.tdql\_cluster\_id不一致，报错，结束；网络问题收不到PutRecord回包，重试3次仍收不到回包，断开客户连接，回滚事务，结束；没有异常，返回客户端插入成功，结束

##### REPLACE INTO

如果发现表中已经有此行数据（根据主键或者唯一索引判断）则先删除此行数据，然后插入新的数据。否则，直接插入新数据。

##### UPDATE

1. 根据update sql拼接出key-value格式的数据行，尝试从本地的region cache获取key落在哪一个region上
2. region cache获取到key所在的region->5，region cache没有缓存key所在的region->3
3. 通过协议GetRegionByKey从meta-cluster获取key所在的region
4. 可能异常：GetRegionByKey返回错误码（非OK）/校验GetRegionByKey回包的cluster id和SQLEngine.tdsql\_cluster\_id不一致，报错，结束；网络问题收不到GetRegionByKey回包，重试3次仍收不到回包，断开客户连接，结束；没有发生异常->5
5. SQLEngine通过协议GetRecord从region上尝试get要更新的key值
6. 可能异常：GetRecord返回错误码（非OK）/校验GetRecord回包的cluster id和SQLEngine.tdsql\_cluster\_id不一致/校验GetRecord回包的key值和请求的key值不一致，报错，结束；网络问题收不到GetRecord回包，重试3次仍收不到回包，断开客户连接，结束；没有发生异常->7
7. GetRecord返回keyNotFound，get不到key值记录，表中不存在要被更新的数据行，返回客户“Query OK，0 rows affected”，结束；GetRecord返回key值所在的数据行->8
8. update主键字段->9，update其他字段->11
9. 通过DeleteRecord删除旧key数据行
10. 可能异常：DeleteRecord返回错误码（非OK）/校验DeleteRecord回包的cluster id和SQLEngine.tdsql\_cluster\_id不一致，报错，结束；网络问题收不到DeleteRecord回包，重试3次收不到回包，断开客户连接，回滚事务，结束；没有异常->11
11. 通过PutRecord把新数据行key-value写到region（tdstore会添加版本信息）
12. 可能异常：PutRecord返回错误码（非OK）/校验PutRecord回包的cluster id和SQLEngine.tdql\_cluster\_id不一致，报错，结束；网络问题收不到PutRecord回包，重试3次仍收不到回包，断开客户连接，回滚事务，结束；没有异常，返回客户端更新成功，结束

##### DELETE

1. 根据delete sql拼出要删除哪一个key值，尝试从本地的region cache获取key落在哪一个region上
2. region cache获取到key所在的region->5，region cache没有缓存key所在的region->3
3. 通过协议GetRegionByKey从meta-cluster获取key所在的region
4. 可能异常：GetRegionByKey返回错误码（非OK）/校验GetRegionByKey回包的cluster id和SQLEngine.tdsql\_cluster\_id不一致，报错，结束；网络问题收不到GetRegionByKey回包，重试3次仍收不到回包，断开客户连接，结束；没有发生异常->5
5. SQLEngine通过协议DeleteRecord把key值数据行从region上删掉

可能异常：DeleteRecord返回错误码（非OK）/校验DeleteRecord回包的cluster id和SQLEngine.tdsql\_cluster\_id不一致，报错，结束；网络问题收不到DeleteRecord回包，重试3次收不到回包，断开客户连接，回滚事务，结束；没有异常，返回客户删除成功，结束

#### Online-DDL

参考F1论文，DDL分为两大类：

1. Logical DDL：不需要同步数据转换(data reorganization)的，如修改表名、字段名、增加删除列等；
2. Physical DDL: 需要同步进行数据转换的，如增加二级索引、修改主键等。

##### Logical DDL

##### Physical DDL

#### LOAD DATA

同原生MySQL8.0。

#### 存储过程

目前在创建存储过程的过程中没有进行KV操作，处理流程和原生的MySQL8完全一致。

#### 创建函数

目前在创建函数的过程中没有进行KV操作，处理流程和原生的MySQL8完全一致。

#### 触发器

目前在创建触发器的过程中没有进行KV操作，处理流程和原生的MySQL8完全一致。

#### 视图

目前在创建视图的过程中没有进行KV操作，处理流程和原生的MySQL8完全一致。

#### 创建事件

目前在创建事件的过程中没有进行KV操作，处理流程和原生的MySQL8完全一致。

#### 分布式事务

**前置条件：**

集群已建立，各组件通信正常

输入

分三种情况：

a.显式开启、显式提交的事务，用户以begin/start transaction开启事务，commit/rollback结束事务

b.autocommit = on，单语句自动提交的事务

c.隐式开启、显式提交的事务，autocommit = false，隐式开启事务，用户以commit/rollback结束事务

**处理过程：**

1、显式开启和提交的事务

1）用户通过sql START TRANSACTION或BEGIN显式开启事务，初始化sqlengine的事务上下文

2）sqlengine事务上下文的初始化包括：通过GetRawTxnTimestamp（裸socket服务）获取事务时间戳和事务ID，记下事务的超时时间

3）用户执行每条dml/dql，sqlengine初始化当前语句上下文

4）sqlengine当前语句上下文的初始化包括：stmt id++，记下语句的超时时间

5）根据SQL拼接Key值，从region cache/meta-cluster处得到key值落在哪一个region，put/get/del等rpc和region交互（参考前文CRUD有详细流程）

6）stmt上下文有数据结构stmt\_participants维护了当前语句操作了哪些region，和region一次数据操作成功后，把该region添加到stmt\_participants中

7）可能异常：stmt操作多个region，其中某个region失败，导致需要回滚整个stmt->8；stmt操作没有发生异常，语句正常结束->9

8）发送AbortStmt到stmt\_participants中的各个region以回滚stmt

9）事务上下文又数据结构txn\_participants维护了当前事务操作了哪些region，一条语句正常结束后，stmt\_participants move追加到txn\_participants

10）事务提交，选择txn\_participants中某一个region（暂时选取列表中的第一个region）作为2PC的协调者，发送Commit RPC到协调者region上，提交请求附带参与者列表

11）可能异常：无法收到Commit回包，重发3次也无法收到回包，事务提交状态未决，断开客户连接，结束；没有异常，返回提交成功/提交失败（取决于Commit回包），结束

12）事务回滚，SQLEngine分别发送EndParticipant RPC到txn\_participants的每一个region上

13）可能异常：无法收到EndParticipant回包，重发3次也无法收到回包，忽略，tdstore会本地超时而回滚事务，返回客户回滚成功，结束

14）超时处理：事务流程代码的关键节点增加事务超时、语句超时的判断。执行过程中事务超时，则回滚事务->12，并报错；如果执行过程中语句超时，则回滚语句->8，并报错，事务不回滚（可继续执行）

2、单语句自动提交的事务

1）autocommit=ON，用户执行dml/dql，隐式开启事务，初始化sqlengine事务上下文

2）sqlengine初始化事务上下文包括：通过GetRawTxnTimestamp（裸socket服务）获取事务时间戳和事务ID，记下事务的超时时间

3）根据SQL拼接Key值，从region cache/meta-cluster处得到key值落在哪一个region，put/get/del等rpc和region交互（参考前人CRUD有详细流程）

4）和region数据操作成功后，region追加到sqlengine事务上下文的txn\_participants

5）sql完成，事务自动提交，选择txn\_participants中某一个region（暂时选择列表中的第一个region）作为2PC的协调者，发送Commit RPC到协调者region上，提交请求附带参与者列表

6）可能异常：无法收到Commit回包，重试3次仍无法收到回包，事务提交状态未决，断开客户连接

7）事务超时：执行过程中，事务超时，发送EndParticipant到txn\_participants的每一个region上，报错“事务超时”

单语句自动提交的事务，没有语句超时

3、隐式开启、显式提交的事务

1）autocommit=OFF，用户执行dml/dql，隐式开启事务，初始化sqlengine事务上下文

2）后续流程同a. 显式开启、显式提交的流程

如果隐式开启的事务迟迟没有提交，事务会超时被回滚并报错；如果执行过程中语句超时，回滚语句并报错，事务不回滚（可继续执行）

**处理结果：**

事务超时，报错“事务超时”

事务提交状态未决，断开客户连接

事务未超时且提交结果明确，返回提交OK

#### 协程池

基本流程：

1. SQLEngine启动时参数thread\_handling=co-threads，系统初始化协程环境
2. SQLEngine以协程模式处理用户请求

#### Region Cache

**前置条件：**

3.0集群已创建，SQLEngine正确链接到meta-cluster，用户输入ddl/dml/dql访问tdstore存储的数据。

**Region cache的添加：**

单点数据操作（输入Key）：

1. SQLEngine启动后，region cache为空
2. 根据key值在region cache中检索其所在的region，无法找到
3. 通过协议GetRegionByKey从meta-cluster获取到key值所在的region信息
4. 将meta-cluster返回的region信息缓存在region cache中

范围数据操作（输入Key range）：

1. SQLEngine启动后，region cache为空
2. 在region cache中检索key range所在的region，无法找到
3. 通过协议GetRegionsByKeyRange从meta-cluster获取key range所在的regions信息
4. 将返回的regions信息缓存到region cache

Region cache的更新：

单点数据操作（输入Key）

1、region cache包含某个region，使得key落在其中

2、根据key值在region cache中找到这个region

通过StartParticipant开启事务，GetRecord/PutRecord/DeleteRecord数据操作时，tdstore回复错误码EC\_TDS\_TRANS\_REGION\_RANGE\_MISMATCH或EC\_TDS\_TRANS\_REGION\_VERSION\_ERROR或EC\_TDS\_REGION\_KEY\_NOT\_EXIST，表示该region上发生过分裂等操作，事务或数据请求该发往新的region上

1. 通过协议GetRegionByKey从meta-cluster获取到key值所在的region信息
2. 删除原region，添加从meta-cluster返回的region信息到region cache中

范围数据操作与单点数据操作类似

**Region cache的查询：**

1. region cache中存在某些region信息
2. 根据key值在region cache中找到某个region，满足key值落在其中

Region cache的释放：SQLEngine shutdown时释放

**处理结果：**

给出一个key值，总能落在一个region上，不存在失败的情况（网络异常导致无法从meta-cluster获取到信息另当别论）。

Region cache的添加、更新的出发点都是根据key值获取所在的region信息，所以处理结果为：返回key值所在的region。

#### 连接收敛

**前置条件：**

rocksdb\_tdstore\_per\_thread=2

rocksdb\_mc\_per\_thread=2

**输入：**

1000个并发连接

**处理过程：**

连接收敛

**处理结果：**

和tdstore以及mc的连接不会随着客户端的增加而增加

连接数最大=rocksdb\_net\_threads\_num\*(1+1+N)

#### 超时处理

协程模型下处理客户连接（read/write）的超时逻辑和原生mysql（pipe/shm/send/recv）一致

connect\_timeout/net\_read\_timeout/net\_write\_timeout/wait\_timeout

#### binlog

### 优化

#### 多SQLEngine

#### 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

##### 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

#### 存储

##### TDStore Recover方案

###### 概述

在TDStore内部，事务提交时，数据是先写到内存（memtable）中，当到达一定条件时，例如memtable的大小超过限制等，会触发TDStore将memtable中的数据刷到磁盘上。这样当TDStore异常退出或者机器重启时，memtable中的数据就会丢失。需要recover方案来进行补齐。

TDStore依赖raft log(raft log中包含了事务提交的数据)来补齐重启后丢失的memtable中的数据。只要raft log一直存在，理论上讲数据就一定安全。但是现实中我们不可能将历史的raft log一直保存。因为有两个因素限制了我们，第一是磁盘的容量，第二个是故障恢复的速度。所以我们必须对raft log进行purge。但是purge raft log必须十分小心，因为purge的多了，recover时数据可能就不能完整恢复了。所以什么时候purge，purge到那个位置，就是我们的recover方案核心部分了。

###### 现有方案

1、TDStore目前的recover方案大体流程

前半段：在事务写commit日志成功后，把数据写入memtable之前，会先检测是否需要flush memtable（如果决定flush，那么TDStore会创建一个新的memtable，并触发后台线程将旧的memtable刷到磁盘上）。如果要执行flush的话，会获取当前region正在applying raft log index，将其保存到versionset中，并记录一个raft log 的镜像点位（snapshot meta，主要是保存last\_applied\_index以及\_last\_applied\_term）。

后半段：等待TDStore的后台线程刷memtable到磁盘后，将versionset中的applying raft log index保存到manifest文件中，并以记录的snapshot meta的点位对raft log进行purge。

2、TDStore目前的recover方案存在的问题

1）目前的recover方案本来还有一步非常关键的freeze步骤，这一步在流程的前半段执行，将所有活跃事务杀死，这样就可以保证不会有事务日志横跨两个memtable，且raft log purge的时候不会把一组完整的事务日志切分了。但是这样就会频繁杀事务，在原则上就和我们整体的设计目标有冲突。其次这个方案实现的也不好，频繁导致死锁，所以不得不去掉freeze这一步。

2）snapshot meta和applying raft log index作用重复

3）applying raft log index 不安全，因为applying表示的是正在执行的日志，这条日志对应的数据还没有写到TDStore内部

4）备机上获取的点位存在问题的，因为备机有异步回放，很多事务日志在队列里面并没有执行回放，从而导致recever时，数据丢失。

5）去掉freeze步骤后，事务日志被部分截取，导致重启进行recever时会存在事务出错的问题，导致数据丢失。

6）存在bug，例如后台线程执行flush时获取保存的braft meta有时却会失效，获取不到。

###### 改进方案

鉴于旧的recover方案的问题，新recover做如下改造：

1、彻底摒弃freeze步骤（目前还有一些残留代码）

2、去掉写入versionset中的applying raft log index。由braft的机制保证开启回放点。

3、去掉备机多线程异步回放。（这个多线程异步回放还会导致备机数据一致性问题，例如后提交的事务，先回放；先提交的事务，后回放。这样旧数据可能会位于sst中的更高层次，导致读取到旧版本数据）(修正，暂时不去掉，去掉的话，代码改动较大，改为按region id 做异步线程分配的标记，保证一个region内的事务有序)

4、在发起flush memtable保存raft log 的snapshot meta时，不以当前最新的commited log index为位点。而是从当前活跃事务中获取一个最小的commited log index为位点（事务进入2pc阶段后，就会开始写raft log，只要第一个raft log写成功，就会获取到一个commited log index）。这样在raft log purge时，就可以保证当前未完成提交的事务，在purge后一定能被成功回放。从而保证了数据安全。

5、基于第4点优化之后，在TDStore重新进行recover回放时仍然会存在部分已提交的事务不完整，这些不完整不会导致数据安全性问题，因为在发起save snapshot meta时，这些事务必然已经提交了（否则最小的commited log index就是它们了，那么它们就不会被截断了）。需要一定进制保证这种不完整的事务回放能跳过，不影响正常的回放。这里的做法是，回放时发现日志不完整，就标记上一个标记。然后在执行clear log后记录这个clear log index，与整个TDStore第一条回放事务(first replay transaction)的 clear log index进行比较，如果比first replay transaction的 clear log index小则说明是正常的，那么就正常释放回放事务上下文，如果比first replay transaction的clear log index大，那说明是异常的，说明TDStore recover时存在问题，直接panic。（first replay transaction的 clear log index比不完整的事务回放的晚，这里在实现时需要先把不完整的事务的clear log index，等待mini clear log index回放出来时，统一比较）

说明：TDStore recover时，第一个被回放的日志，必定是属于一个完整事务的第一条日志。因为raft log purge截取的时候，是以当前活跃事务的最小log index为镜像点进行的。

举例:

Ri：事务Ti的redo log

Pi：事务Ti的prepare log

Ci：事务Ti的commit log

Li：事务Ti的clear log

理想的事务log排列：

R1 P1 C1 L1 R2 P2 C2 L2

事务log可能的排列：

R1 R2 P1 C1 L1 P2 C2 L2

这样，在后面C2这个点触发log purge时，我们会定位在R2这个点，然后recover的时候，事务1的第一条就是P1，这样是不完整的，目前我们的代码会报错。通过优化5可以解决这种报错。

6、定位并解决后台线程执行flush时获取保存的braft meta有时却会失效的问题。

##### region数据的物理删除

##### multi raft log存储

##### pipeline write

##### TDStore压缩算法

##### 事务日志提交中的各种规则和约束以及对于各类情况的处理

##### 故障注入（libfiu）

##### compaction限速方案

##### 数据位置感知设计文档

### 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）

#### REGION & REPLICATION

##### CREATE REGION

处理过程：

1. 创建region：创建包含region的状态、副本等元信息
2. 构建raft group：多副本进行选举，选出新的leader
3. 启动 replay server
4. 回复MC

处理结果：

正确：TDStore中创建出一个新的多副本region，上报给MC，在MC中能看到此region的元数据信息

失败：返回给MC具体的错误信息

##### Remove Region

处理过程

1. 获取要删除region的元信息
2. 删除此region元信息和raft信息
3. 删除数据文件
4. 回复MC

处理结果：

正确：在TDStore中删除此region的元信息和raft信息，在MC中无法查询到此region，清除数据文件

失败：返回给MC具体的错误信息

##### Split Region

处理过程

1. 计算region的分裂点
2. 上报region的分裂点到MC，获取新的region的元信息
3. 创建新的region，更新region元信息，创建新region的raft group（与create region类似）
4. 回复MC

处理结果

正确：旧的region完成分裂，创建出新的region，新旧region元信息正确，在MC中可以查询新旧region信息

失败：返回给MC具体的错误信息

##### Create Replica

处理过程

1. 创建region的元信息
2. 启动单节点raft，副本处于init状态，不对外提供服务
3. 回复MC

处理结果

正确：创建出一个不包含数据的region 副本，MC中可以查询到此副本的元信息。

失败：返回给MC具体的错误信息

##### Add Replica

处理过程

1. 添加新副本的raft节点到目标region的raft group中
2. 更新region元信息，包含region的member\_verison和member的信息等
3. 回复MC

处理结果

正确：region的member\_version和member信息更新完成，在MC中可以查询到region最新的信息。新的副本可以回放leader节点同步来的日志

失败：回复给MC具体的错误信息

##### Online Replica

前置条件

存在处于init、offline 状态的region的副本

处理过程

1. 修改此副本状态为F\_ONLINE状态
2. 修改此副本元信息
3. 回复MC

处理结果

正确：此副本的状态变更为F\_ONLINE，MC中可以查询此副本的最新状态

失败：返回给MC具体的错误信息

##### Offline Replica

前置条件

有处于online或者working状态的follower副本节点

输入

要offline的副本的tdstore\_id、region\_id、状态等信息

处理过程

1. 修改此副本的状态为F\_OFFLINE
2. 修改此副本的元信息
3. 回复MC

处理结果

正确：副本状态变为F\_OFFLINE，MC中可以查询到此副本变化后的状态

失败：返回给MC具体的错误信息

##### Remove Replica

前置条件

有处于offline状态的副本

输入

要删除副本的region的region\_id、状态、member\_version、quorum、要删除的member的元信息等

处理过程

1. 从raft group中删除此副本的raft节点
2. 修改region的元信息
3. 回复MC

处理结果

正确：此副本处于F\_REMOVE状态，不删除副本的数据文件。MC中可以查询到此副本新的状态。

失败：返回给MC具体的错误信息

##### Destroy Replica

前置条件

有处于remove状态的副本

输入

要销毁的副本的region\_id和tdstore\_id

处理过程

1. 删除此副本的region元信息
2. 删除此副本的数据文件
3. 回复MC

处理结果

正确：从MC查询不到此副本信息，更新region中的副本信息，删除掉此副本的数据文件

失败：返回给MC具体的错误信息

##### Transfer Leader

前置条件

此region中至少包含一个leader节点和一个follower节点

输入

要接切换leader的region\_id、新leader所在的tdstore\_id、旧的leader所在的tdstore\_id等信息

处理过程

1. 切换raft group中的leader到新的TDStore上
2. 更新region的member信息
3. 回复MC

处理结果

正确：完成主备切换，在MC中可以查询到region中各个member的最新状态

失败：返回给MC具体的错误

#### Online-DDL支持

#### 分布式事务

##### 分布式事务两阶段提交

**前置条件：**

tdstore创建多个region（region数目大于1）

**输入：**

按照协议进行分布式事务，分别调用RPC：StartParticipant，Commit，EndParticipant发起事务并执行提交。

可使用Sqlengine作为前端，开始事务后，进行跨Region的增删改查后提交事务。

**处理过程：**

Sqlengine调用StartParticipant开启目标Region上的参与者上下文，并进行读写访问，最后调用Commit进行事务提交，收到Commit的TDStore会构造出协调者上下文，通过两阶段提交协议原子的提交事务，当提交失败时原子的回滚整个事务。

Sqlengine调用StartParticipant开启目标Region上的参与者上下文后，也可以调用EndParticipant放弃本次事务，EndParticipant不会进入两阶段提交需要Sqlengine发给所有参与者。

当参与者在开始事务上下文后，会启动定时器，如果直到定时器超时都没有收到Prepare/Commit请求或者EndParticipant请求，会直接放弃事务上下文。

**处理结果：**

1. TDStore收到事务Commit构造出协调者后，整个事务要么Abort要么Commit成功。
2. Sqlengine发送EndParticipant到所有的参与者放弃事务。
3. 参与者超时放弃事务，整个事务要么由协调者原子回滚，要么所有参与者相继超时（或者Sqlengine的推进下）放弃事务。

##### 分布式事务写写冲突检测

**前置条件：**

TDStore创建至少一个Region

**输入：**

同时开启两个分布式事务，分别对同一个Key进行Put操作，然后分别提交这两个事务。

**处理过程：**

事务在读写阶段，将所有Put操作涉及的key都缓存起来，在事务进入两阶段提交时，参与者于Prepare阶段进行写写冲突检测，检测在本事务生命周期内是否与其它事务有冲突，如有冲突则返回协调者Prepare失败，协调者原子的回滚整个事务后返回Sqlengine(或者TDStore Client) 事务Abort；如果Prepare阶段没有写写冲突，则参与者返回协调者Prepare成功，协调者将进一步收集其它参与者Prepare的结果。

**处理结果：**

上述两个事务，先提交者成功，后提交者事务回滚。

##### 分布式事务SI隔离级别

**前置条件：**

TDStore创建至少一个Region

**输入：**

事务一读取某个Key

事务二修改这个Key的值，并提交

事务一再次读取这个Key

**处理过程：**

数据在TDStore中是多版本保存，读取时根据Timestamp进行可见性判断。

**处理结果：**

事务一再次读取Key的值与第一次相同，不受事务二的影响。

##### 分布式事务prepare阶段阻塞新事务读

**前置条件：**

TDStore创建至少一个Region

**输入：**

事务一进入提交阶段的同时，事务二开启并读取事务一中刚刚有写入操作的Key。

**处理过程：**

事务在读写阶段，将所有Put操作涉及的Key都缓存起来，在事务进入两阶段提交时，参与者于Prepare阶段将缓存的Key与Prepare\_ts联合插入内存锁中，内存锁结构如：<Key,Prepare\_ts>。

同时事务在读取数据时，都要判断要读取的Key是否在内存锁中，如果存在且本事务的Start\_ts 大于内存锁中Key对应的 Prepare\_ts时，该读取要被阻塞。

**处理结果：**

如果事务二读取的时候，事务一恰好在Prepare阶段，则读取被阻塞，直到事务一Commit成功或者Abort后继续读取，或者阻塞超时收到读取错误的提示.

##### 分布式事务的异常处理

#### 存储

##### find key range overlap files

**输入：**

start\_key, end\_key

**处理过程：**

根据key range从LSM-tree中找到跟range overlap的files

1. 获取column family data的super version（加锁）
2. 获取该cf的LSM-tree的每层sstable files的meta信息
3. 遍历每层file的meta信息，判断是否跟key range overlap
4. 将overlap的sstable文件Push到输出的vector中

**处理结果：**

将所有与key range的sstable文件push到输出参数的vector中

##### get key with user\_key(not skip)

**输入：**

start\_key, end\_key

**处理过程：**

1. 创建sstable file reader的iterator, lower\_bound是start\_key, upper\_bound是end\_key
2. seek to first
3. 遍历的时候获取顺序获取每个entry，不跳过相同user key的kv entry

**处理结果：**

得到user\_key的所有sequence number版本数据

##### 流式rpc的功能

**前置条件：**

随着region数据迁移开启，迁移完成结束。

**处理过程：**

TDStore收到region数据迁移的指令之后，先判断data\_version是否为0，

如果为0，则表示没有静态数据版本， 则不需要开启流式rpc

如果不为0， 则需要迁移静态数据版本，开启流式RPC

**处理结果：**

在回调on\_received\_message()表示中收到了数据

在回调on\_idle\_timeout()中表示rpc超时

在回调on\_closed()表示rpc连接关闭

##### SSTable文件写入

**输入：**

IOBuf数据

**处理过程：**

1. 如果没有打开sstable文件，则打开一个新的sstable文件
2. 如果已经打开了sstable文件，将数据写入到sstable文件中，如果超过了配置文件中的target\_file\_size\_base的大小，则close该sstable文件，并打开一个新的sstable文件。

**处理结果：**

将IOBuf中的数据写入到sstable文件中

##### Ingetst sstable文件

**输入：**

Sstable文件

**处理过程：**

调用rocksdb的IngestExternalFiles接口将sstable文件插入到LSM-tree中

**处理结果：**

将一个新的sstable插入到lsm-tree中，并在rocksdb的storage info中能看到该文件的Meta信息

##### Region迁移数据校验工具

**输入：**

start\_key, end\_key

**处理过程：**

1. 通过storage层的iterator接口遍历数据源端[start\_key, end\_key)的数据，计算出checksum值
2. 同样，遍历出目的段[start\_key, end\_key)的数据，计算出checksum
3. 对比两边的checksum是否一样
4. 当出现checusm不一样的，提供逐条kv对比的功能，找第一条不匹配的kv数据

**处理结果：**

如果两边数据一样，则返回true

如果两边数据不一样，逐条对比每条kv，返回第一条不匹配的kv数据

##### 存储层数据校验工具

**处理过程：**

1. 保存用户数据插入的数据
2. 将用户数据插入到tdstore中
3. TDStore将数据flush到sstable文件中(做一次flush/compaction，或者做一次重启)
4. 从sstable中通过kv接口遍历数据生成checksum，与用户数据的checksum做对比
5. 当出现checusm不一样的，提供逐条kv对比的功能，找第一条不匹配的kv数据

**处理结果：**

如果两边数据一样，则返回true

如果两边数据不一样，则返回第一条数据不匹配的kv数据

#### 分裂切主迁移

#### Region状态机

#### 内存控制

#### 备份恢复

### 优化

### 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。

说明：在TDSQL2.0中也应用MC的时间戳，提供全局一致性读。

### 功能

#### 全局唯一事务时间戳

**输入：**

SQLEngine 通过时间戳专用通道向MC发动请求，请求的结构体为定长为 33B的经 pb 编码后的 GetTxnTimestampRequest 字节数组

**处理过程：**

MC 接受到请求后，获取当前时间戳，以及全局 schema version，赋值到 GetTxnTimestampResponse 结构体后经 pb 编码后返回。

##### 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。

02164320402

#### 数据资源管控与调度

##### 生成新Index ID

前置条件：

cluster 已启动

输入：

SQLEngine 调用 GenerateIndexID RPC 或 GenerateSysIndexID RPC

处理过程：

MC 为请求分配新的 index id

处理结果：

成功则返回ok，以及分配的 index 的起始标号和数量；否则为失败，具体信息会包含在 response header 中。

##### 分配表自增字段值

前置条件：

对应的 index id 已经向 MC 注册过  
处理过程：

MC 根据请求分配并返回自增字段的区间

处理结果：

成功则返回ok，并包含所分配的自增值的区间；否则为失败，具体信息会包含在 response header 中

##### 提供 Region 路由信息的功能

前置条件：

cluster 已启动

处理过程：

MC 根据请求中包含的 key 或者 key range，查找这些 key 所处的 region，然后返回对应的 region 或 region 元数据列表，以及当前 region 的 leader。

处理结果：

成功则返回ok，以及对应的 region元数据；否则为失败，具体信息会包含在 response header 中。

##### 提供下发 Region 分裂任务的功能

前置条件：

Cluster 已启动

处理过程：

MC 向指定或随机选取的 region 下发 split region 的任务，发送给该 region 当前的leader，region leader 收到该任务后，计算出要使用的 split key，调用 AskSplitRegion RPC，向MC汇报，收到MC的包含ok的回包后，tdstore 进行分裂，分裂完成后，调用 ReportRegionJob 向 MC 汇报。

处理结果：

分裂完成后，外部通过调用 GetAllRegions RPC，可以获取到新分裂出来的region的信息。如 Region 1 原本的 key 区间为 [a, z), split key 为 k，将 region 1 分裂为 region 1 和 region 2 之后，region 1 的区间变为 [a, k),region 2 的区间为 [k,z)。

##### 提供下发增加和删除 Replica 任务的功能

前置条件：

Cluster 已启动，删除副本时当前region 副本数大于1，增加副本时当前副本数少于集群中 tdstore 的总数

处理过程：

通过改变指定的 region 的quorum 值，MC 向 region 下发增加或删除replica的任务。完成后，tdstore 调用 ReportRegionReplicaJob RPC 向 MC 汇报。

处理结果：

完成后，外部通过调用 GetAllRegions RPC，可以获取到增加或删除副本后的region的当前成员列表状态。

##### 提供下发迁移Replica任务的功能

前置条件：

Cluster 已启动，下发任务时当前 region 副本数少于集群中 tdstore 的总数

处理过程：

MC 向 region 下发迁移 replica（可在请求中指定，若不指定，则随机选择）的任务，迁移任务的过程分为五个步骤，以当前3副本状态的 region 为例：

1. 选择一个没有该region 的 tdstore，创建一个新的 replica，并让该 replica 同步数据；
2. 将该 replica 添加进 region 成员里，quorum 变为 4；

3、选择一个leader以外的旧的replica，将其 offline；

4、将该 replica 移除出 region 的成员列表，quorum 变为 3；

5、将该 replica 的数据销毁，即 destroy

处理结果：

完成后，外部通过调用 GetAllRegions RPC，可以获取到迁移replica后region 的新的状态展示。

##### 下发Region切主任务的功能

前置条件：

Cluster 已启动，下发任务时当前 region 副本数大于1

处理过程：

MC 向指定的region下发切主的任务，发送给该 region 当前的leader，region leader 收到该任务后，根据任务中指定的 new leader，在内部进行角色转换，将 leader 角色切给新主后，调用 ReportRegionJob 向 MC 汇报。

处理结果：

完成后，外部通过调用 GetAllRegions RPC，可以获取到切主后region 的新的状态展示，leader 已经变为新的 tdstore。

##### 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

##### 创建新的cluster

MC 接收到该请求后，先判断所指定的 cluster ID 是否已存在，然后判断所指定的默认副本数是或合法，接口相关参数说明详见接口文档，检验通过后，创建cluster 相关运行环境。

##### TDStore注册

前置条件：

指定的 cluster 已创建

输入：

TDStore 模块调用 RegisterTDStore RPC

处理过程：

MC 接收到该请求后，根据 store ID 判断该 store 是否为新注册的 store，或是已经注册过，目前处于重启阶段的 store。若为新注册的 store，则需要判断是否与现有的已经注册的store 的 ID 重合。

处理结果：

若为新注册的 store，返回 ok 表示成功，否则为失败，具体信息会包含在 response header 中。

若为重启的 store，则返回该 store 上所存储的所有 region 的元数据的列表。

##### 启动（bootstrap）新cluster

前置条件：

cluster 已创建，有符合 cluster 默认副本数量 TDStore 已完成注册

处理过程：

MC在检测到有足够数量的已注册的 store 后，会依次向store下发创建第一个 system region 和 创建第一个 user region 的任务，当两个region 的创建成功后，该 cluster 的状态则变更为 bootstrapped.

处理结果：

通过 GetClusterInfo 可在返回结果中获取到集群状态为 bootstrapped，通过 GetAllRegions 可在返回中获取到当前cluster中有一个system region 和 一个 user region。

##### 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

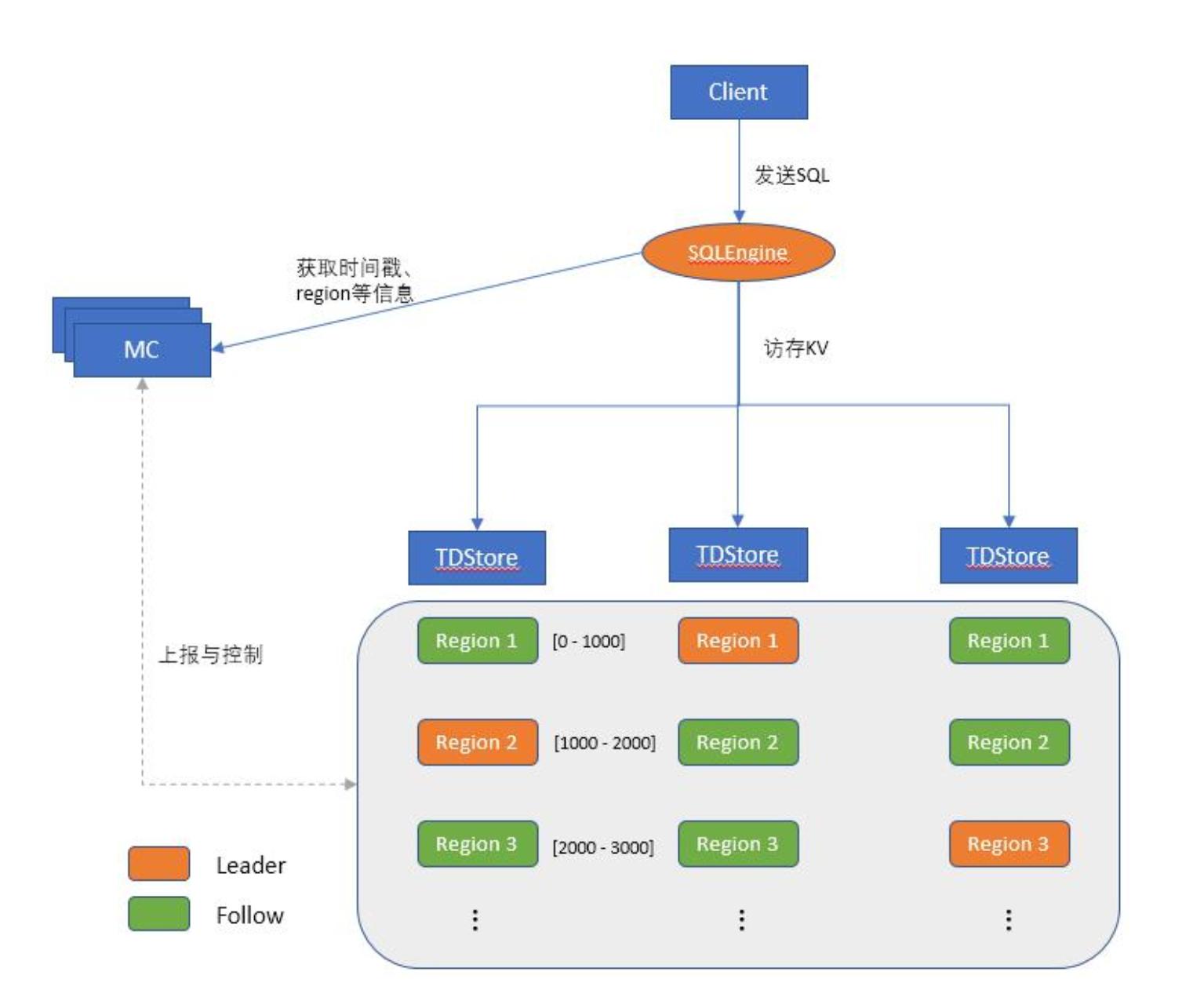
/<cluster\_group>/<cluster\_id>/tdbr/mc\_recover\_task\_meta/

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

# 分布式事务



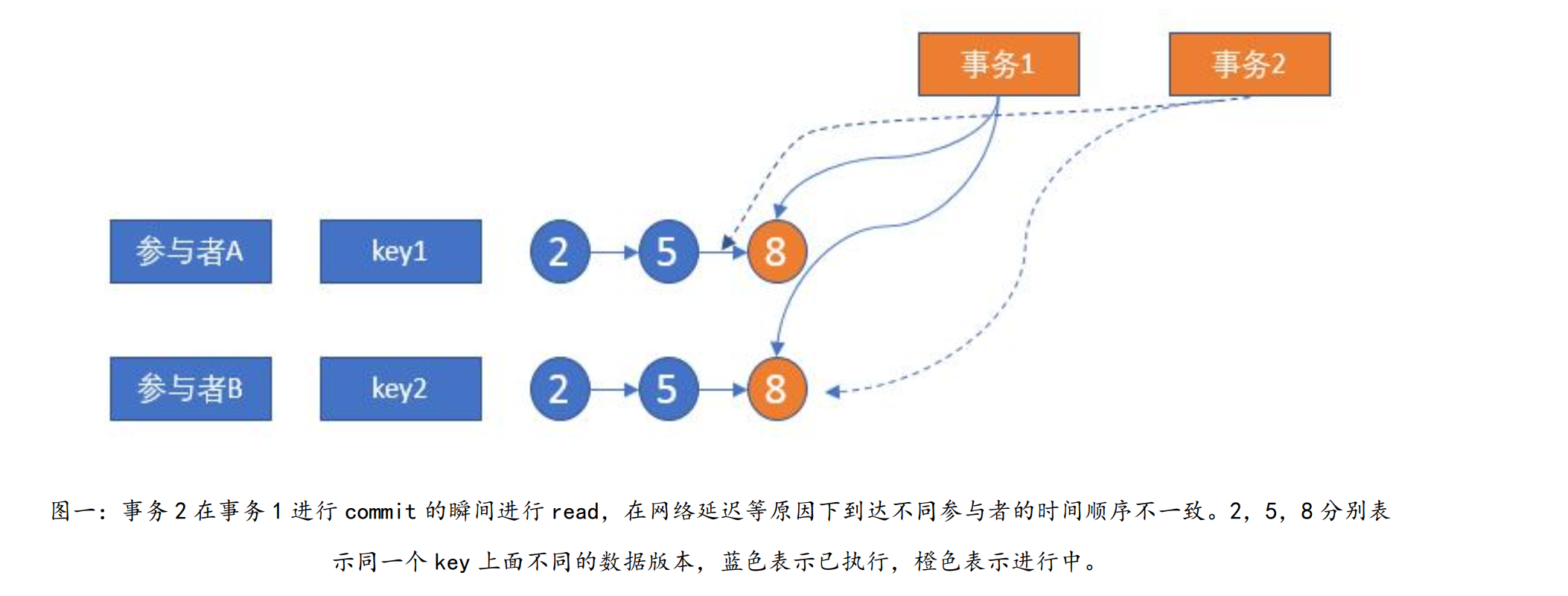
TDSQL3.0主要由SQLEngine、MC、TDStore三个模块构成。其中TDStore是存储节点，对外提供KV操作，TDStore上数据拆分成一个个的逻辑单元region，不同TDStore的region之间组成一个raft组，形成主备关系，只有主能提供写服务；SQLEngine解析Client发送过来的SQL，转换成KV操作访问TDStore存储数据或者读取数据。MC提供全局时间戳服务，同时管理与控制 TDStore上region的分布，向其它模块提供路由服务。

TDSQL3.0采用**乐观并发控制**，提供**SI隔离级别**。使用两阶段提交保证分布式事务的原子性，由于SQLEngine无状态的特点（不能记录协调者日志且允许发送永久性故障），所以我们将协调者身份下放到TDStore上，由SQLEngine 提交事务时指定参与者中的哪个TDStore作为协调者，具有协调者身份的TDStore会将协调者日志合并自己的参与者日志中，故障恢复时会将协调者身份一并恢复出来。

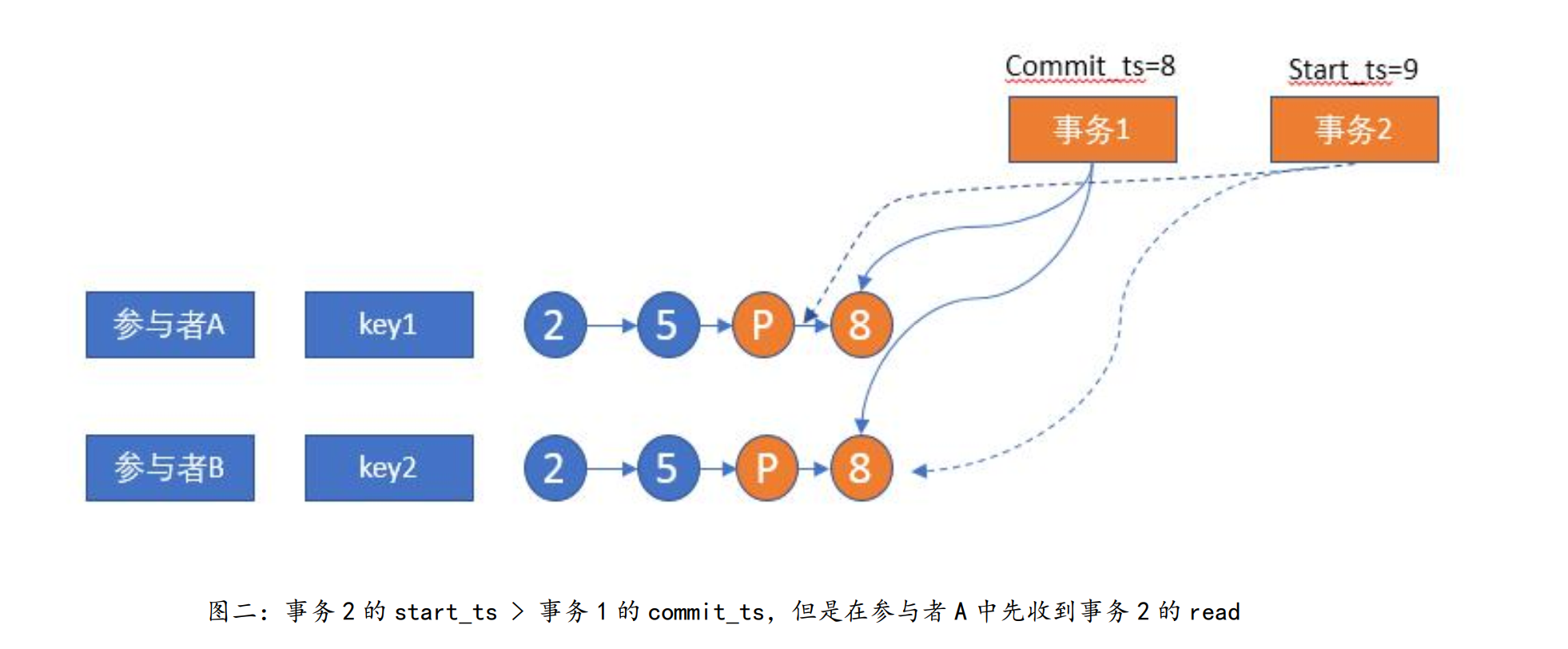
## 两阶段提交事务模型

### 事务模型

如图一所示，对于分布式事务1、2分别有两个参与者A、B。分布式事务1是写事务，分别在参与者A的key1和参与者B的key2上有写入。分布式事务2是读事务，需要读取key1和key2。现在在分布式事务1进行commit的瞬间，事务2进行 read（业务视角）。其中事务2的start\_ts可能大于或者小于事务1的 commit\_ts（内部视角）。由于网络延迟以及操作顺序等原因，事务1、2到达参与者A、B的时间顺序不一致，我们如何保证事务2不会读到事务1的部分写入呢，即事务2要么读到事务1在所有参与者上的全部写入，要么全部读不到。

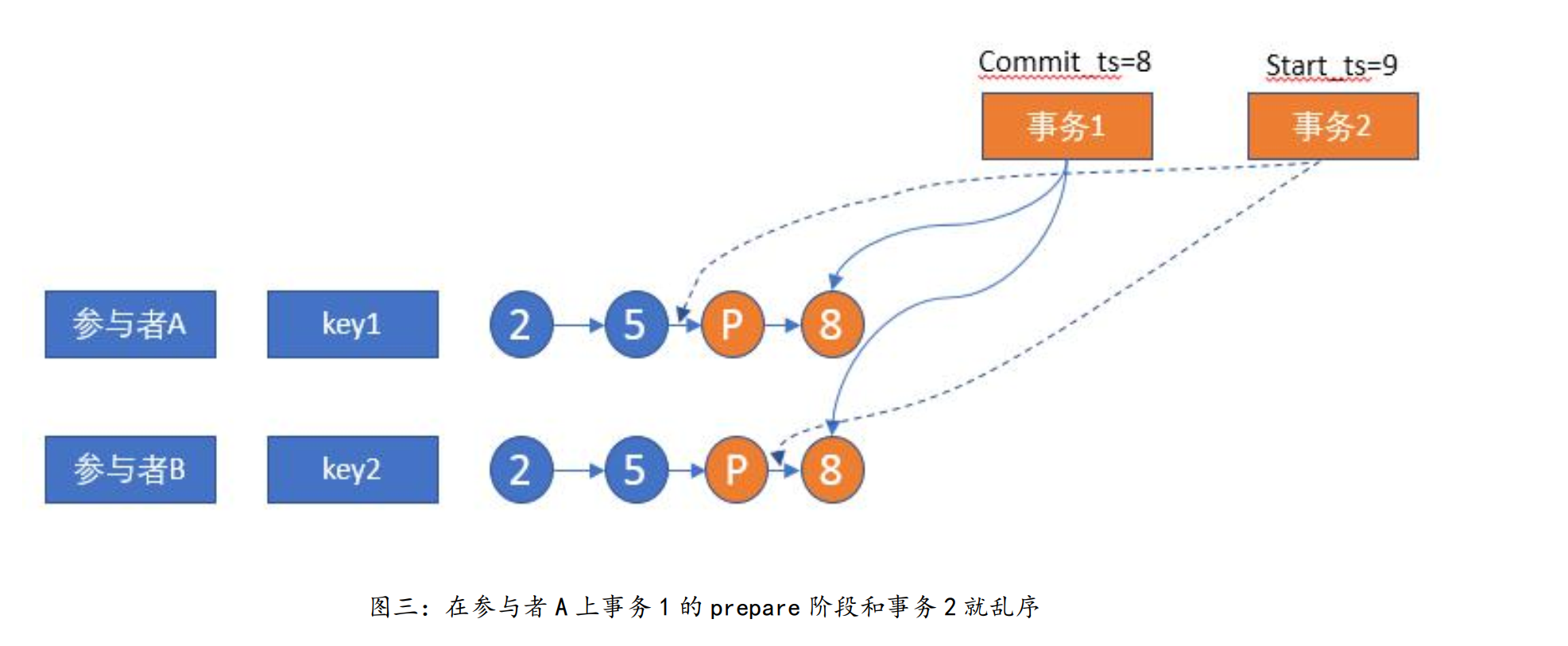


事务模型 1：2PC，事务处于prepare 阶段时，其它事务无感知，Commit 后感知。对于这个模型，在如图二所示的场景下会出现问题。在图二中事务 1 的commit\_ts小于事务2的 start\_ts，且在事务1中参与者A、B都已经进入了prepare阶段；参与者A先收到事务2的read，后收到事务1的commit；参与者B先收到事务1的commit，后收到事务2的 read。此时，事务2在参与者A上没有读到事务1的写入，在参与者B上却读到了。



事务模型2：2PC，事务获取到commit\_ts，然后再进行prepare，且当参与者进入prepare后会阻塞其它事务read相应的key，直到该事务commit或者rollback。

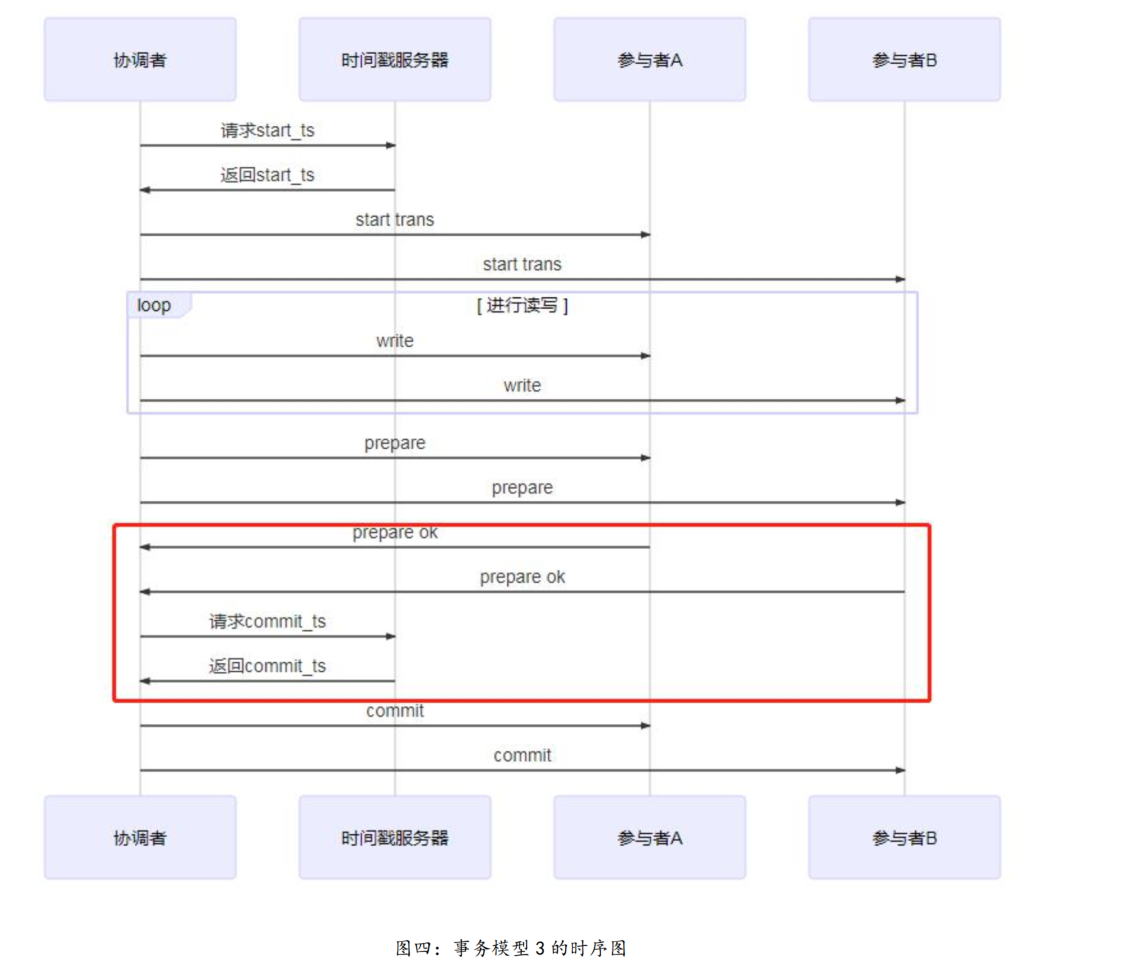
事务模型2可以解决图二场景的问题， 因为在参与者A上事务2的read会被事务1的prepare阻塞，直到事务1 commit成功，这样事务2在参与者A、B上都read到了事务1 的写入。但是在图三场景下，事务模型2仍然会存在问题。图三场景在prepare阶段就发生了顺序上的不一致，首先事务2的start\_ts大于事务1的commit\_ts；参与者A先收到事务2的read，后收到事务 1 的 prepare；参与者B先收到事务1的 prepare，后收到事务2的read；。此时，在参与者B上，事务2会被事务1阻塞，直到事务1commit成功，由于事务2的start\_ts大于事务1的commit\_ts，最终事务2在参与者B上还是会read到事务1的写入，但是在参与者A上事务2由于感知不到事务1的存在，所以read不到事务1的写入，这就得到了不一致读。



从上述两个事务模型中，我们可以看出干扰分布式事务最大的问题是事务时序 跟 参与者时序不一致的问题，即事务时序大的 在部分参与者上却可能会先执行。对于写写冲突我们可以有事务并发控制模型解决，OCC可以在prepare时检测冲突，PCC可以在主键写入时进行加锁阻塞。但是对于读一致性，由于参与者时序的问题，仅仅阻塞是不够的。我们需要分布式事务模型可以保证如果读事务R的T(R)大于写事务W的 T(W) (即读的 start\_ts 大于写的commit\_ts)，那么事务R在所有参与者上都能读取到事务W的写入。

事务模型3： 2PC，协调者在事务的所有参与者prepare成功后，再向时间戳服务器（MC）获取commit\_ts，同时所有事务参与者在进入prepare阶段后需要阻塞其它事务读取相应的 key。

事务模型3的时序图如图四所示，最关键的一步就是commit\_ts的获取时机是在所有参与者prepare成功之后。



这个模型不会出现图三的场景，因为当参与者A先收到事务2的read， 后收到事务1的prepare时，一定不会出现事务2的start\_ts大于事务1的commit\_ts的情况，我们可以使用反证法来证明这一点，这里就不详细展开了。我们主要推论一下，事务模型3是否可以保证如果读事务R的T(R)大于写事务W的 T(W)，那么事务R在所有参与者上都能读取到事务W的写入。

由于T(W)<T(R)，时间戳服务器肯定是在T(R)之前给出T(W)；因此，事务W是在事务R收到T(R)之前请求了T(W)作为提交时间戳。我们知道事务R在收到T(R)之前不能执行读取操作，而事务W在它的提交时间戳T(W)之前必定在所有参与者上完成了prepare且prepare执行成功；因此，上面的推理保证了事务W在事务R做任何读之前就在所有参与者上成功进行了prepare；事务R的读要么看到已经完全提交的写记录，要么被事务W的Prepare阶段阻塞着直到事务W commit或者rollback。所以在T(W)<T(R)的前提下，只要事务 W 进行了commit，那么它的写对事务R的读取都是可见的。由此可知事务模型 3 可以做到分布式一致性读。

综上所述，我认为commit\_ts获取时机才是percolator作为分布式事务的核心。而引入lock和write列反而是percolator建立在bigtable上的妥协之举，因为bigtable只能提供单行的原子事务，而不能做更多的保证。

如果我们完全采用Percolator所有模型，有以下优点：

1、 sqlengine层可以对事务写进行缓存，减少写事务 RPC。

2、 Store层可以作为一个分布式 keymap，只向上层提供单行的原子事务即可，不需要作为参与者， 参与到分布式事务中，这样store实现的复杂度会较低。

缺点：

1、 除了data列之外需要额外引入lock以及write列， 造成了数据冗余。 且使得在事务commit阶段（业务视角），要多次写入RPC（分为prewrite以及commit，在prewrite阶段必须把所有key的lock列都写入数据之后才能进入commit）。 其中每次写入rpc对于store来说都是数据写入，都需要同步到备机上才能给 sqlengine 层应答。这样commit的延迟会很大。

2、这种架构的并发控制模型不能轻易的从 OCC 转变为 PCC。

如果我们采用上述事务模型3，有以下优点：

1、store层在分布式事务有更多参与， 降低了sqlengine 层的复杂性。（事务写写冲突，以及prepare阶段的阻塞都有store层进行）

2、这种架构下的并发控制模型，从OCC切换到PCC，只需要store层切换并发控制模型即可，这非常容易做到。

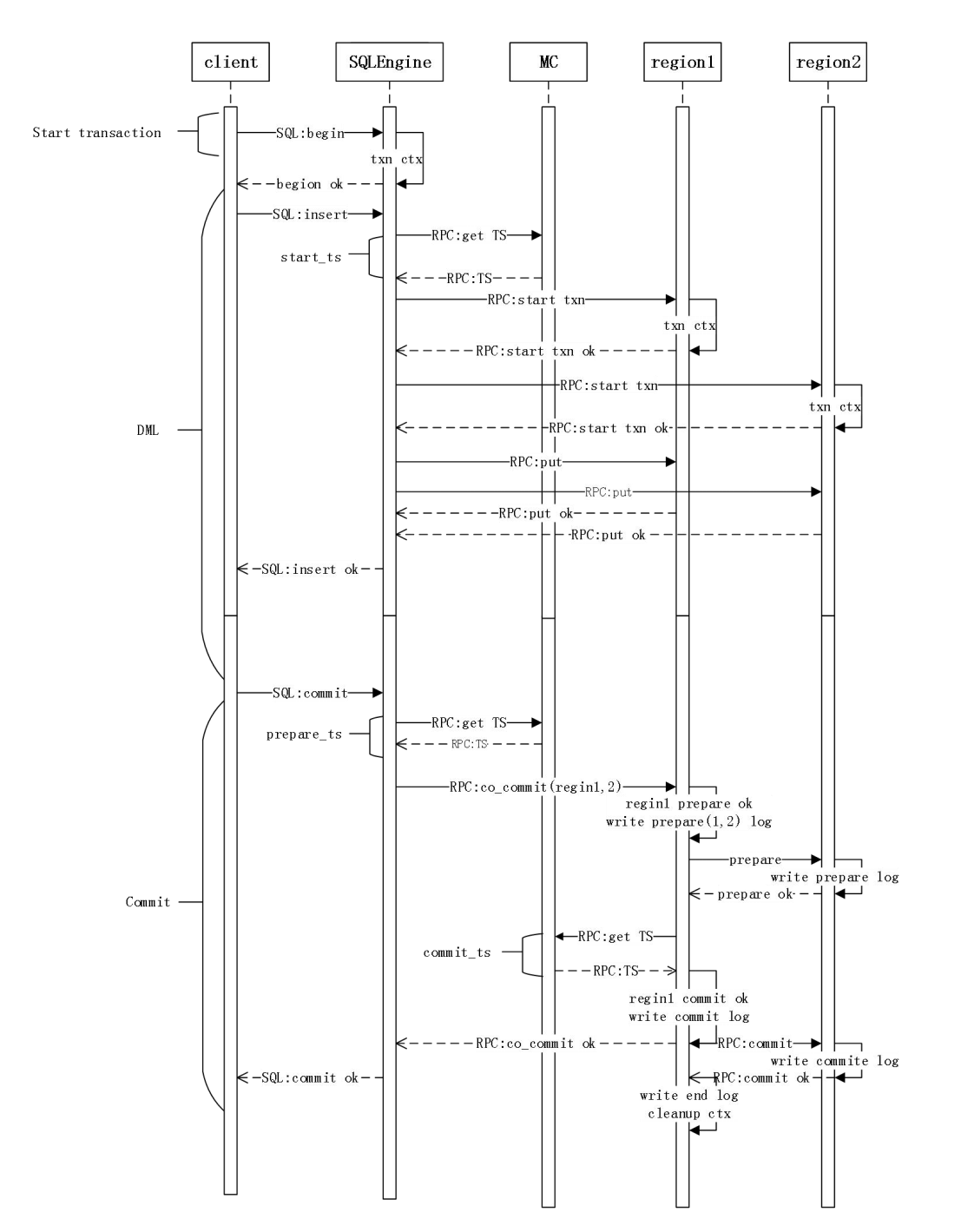
3、Commit阶段，rpc较少，延迟会比较低。

缺点：

1、store层的实现会较为复杂，需要保证store主备切换后，事务继续进行。

### 分布式事务概况

分布式事务正常流程时序图：



原子性，由二阶段提交来保证，要么在所有参与者上成功，要么在所有参与者上失败。由于作为协调者的SQLEngine是无状态的，不存储协调者日志， 所以我们将协调者身份下放到TDStore上。

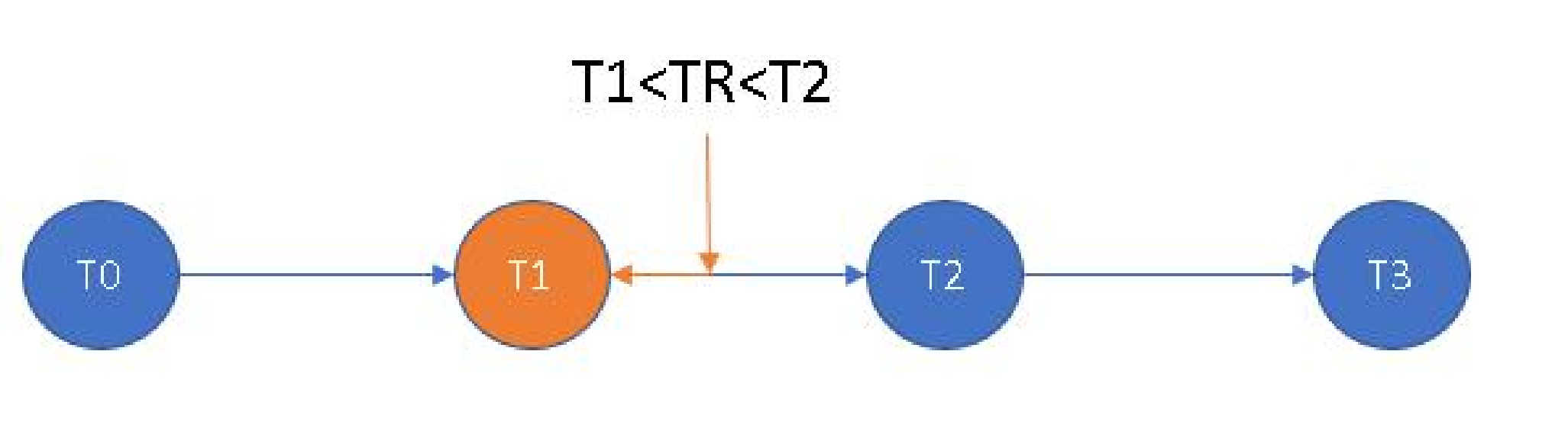
持久性，由作为参与者的TDStore来保证，TDStore实现了RAFT协议保证了单节点故障下的持久性。

一致性，ACID中的一致性为数据约束一致性，由SQLEngine执行SQL的时候保证，不由事务模型决定。

隔离性，主要取决于数据库的并发控制模型。基于TDStore乐观并发控制的实现方式，TDSQL3.0 事务模型算得上是一种独特的2PC-MVOCC，对外提供SI隔离级别。

#### 数据多版本实现方式

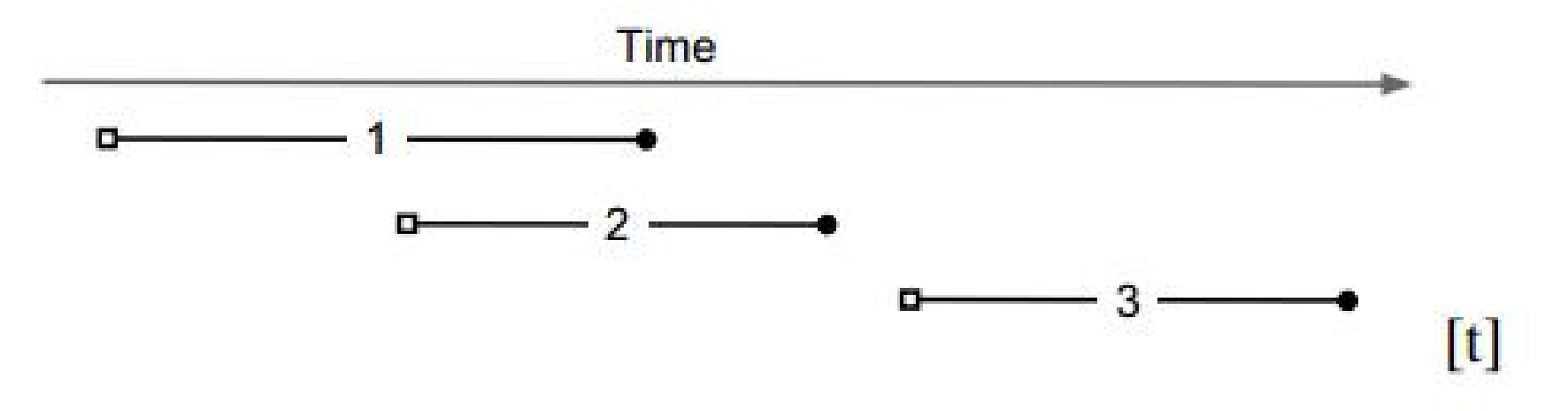
TDStore对每个数据项存储多个时间戳版本，形如（key,timestamp）->value，所以通过key和一个时间戳，我们可以读到小于这个时间戳的最新数据，如图所示，读取时间点TR大于T1，但是小于T2，所以读到的是这个KEY在T1时间戳上的数据。基于这个实现我们可以看出一个时间戳就是数据库的一个“快照镜像”。



TDStore读取数据时间戳规则

#### 事务的生命周期

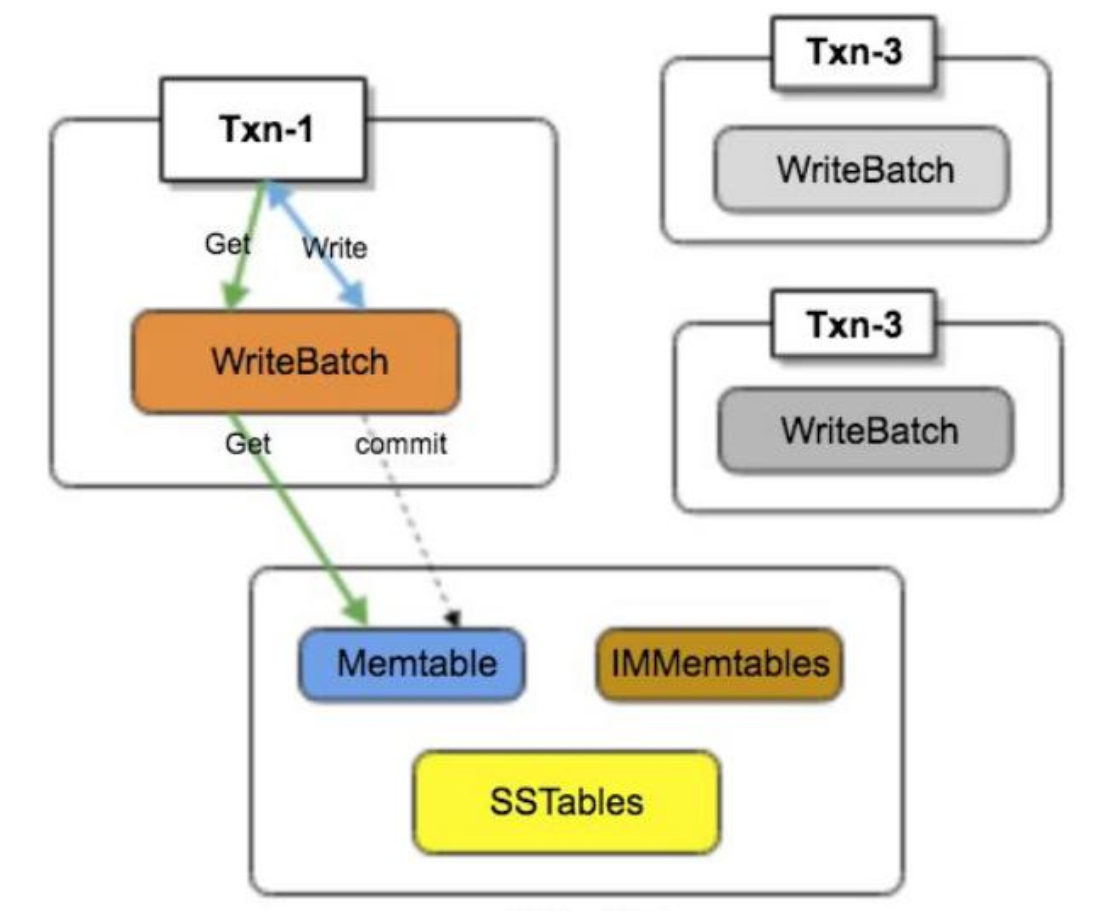
同时得益于全局时间戳服务，我们可以在分布式环境中定义一个事务的生命周期，形如[start\_ts,commit\_ts]表示一个事务的开启时间和提交时间，事务用start\_ts访问数据库的“快照”，在commit的时候将数据与commit\_ts一起写入数据库产生一个新的快照。通过这个生命周期我们可以定义事务的先后顺序以及是否并发，如图所示事务3的start\_ts大于事务1和事务2的commit\_ts，即事务1、2早于事务3，所以事务1、2提交的数据对事务3来说是可见的，但是对于事务1和事务2之间它们的生命周期有重叠，所以我们认为它们是并发的。



事务的生命周期

#### 乐观并发控制

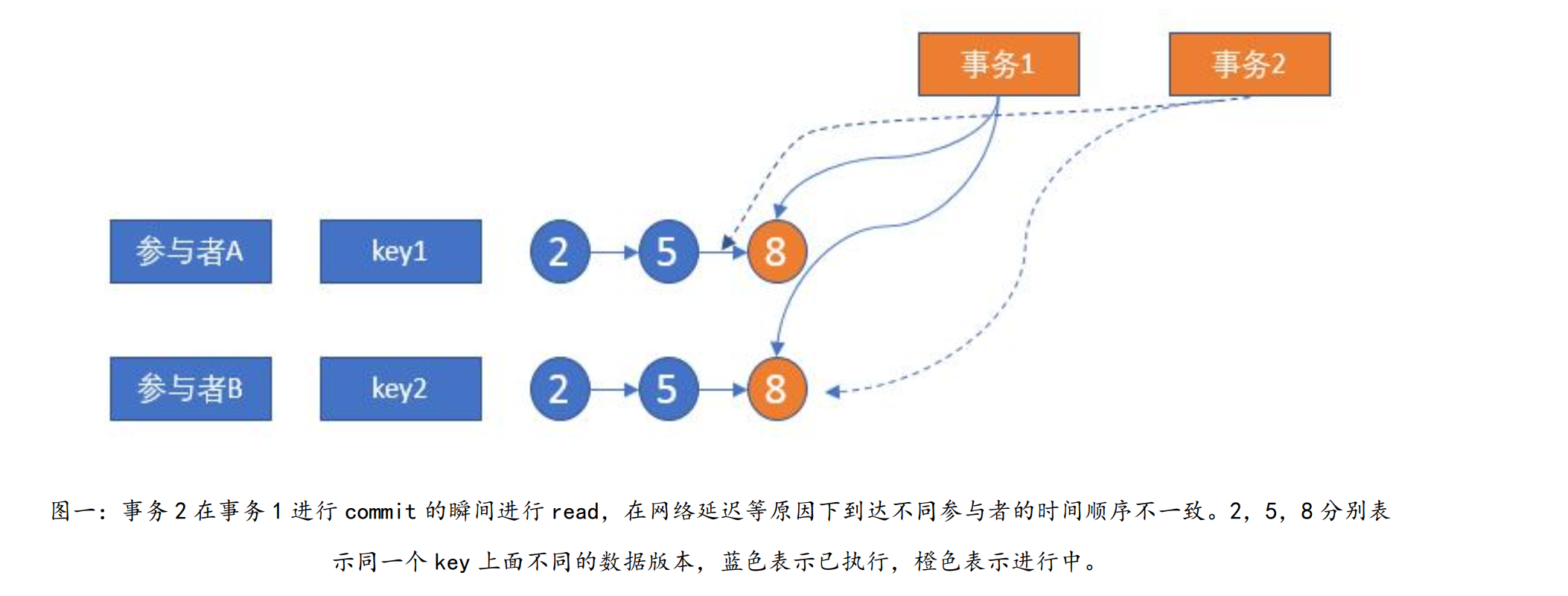
上图中的事务1和事务2在时间线上有重叠，如果它们的写数据也有重叠（注意我们目前提供的是SI隔离级别，读取是快照读，并不涉及读写与写读冲突），则事务1与事务2中必有一个事务需要回滚，否则就会出现丢失更新的现象。在TDSQL3.0中采用OCC，后提交的事务会被回滚掉，即上图中的事务2会被回滚。这里的实现是基于rocksdb的乐观事务，如下图所示，本事务中所有的 Write 都缓存在事务的private空间（rocksdb的writebatch），只有在提交时才进行有效性检测，判断与已提交的事务是否存在写写冲突（比较DB中 key最新的timestamp与写事务start\_ts的大小），检测通过才写入真正的DB中，对其它事务可见（注意这里与传统的OCC不一样，并不是用read set去做有效性检测，原因还是SI隔离级别）。不过结合2PC和RAFT之后，rocksdb的乐观事务并不能完全满足我们的要求。



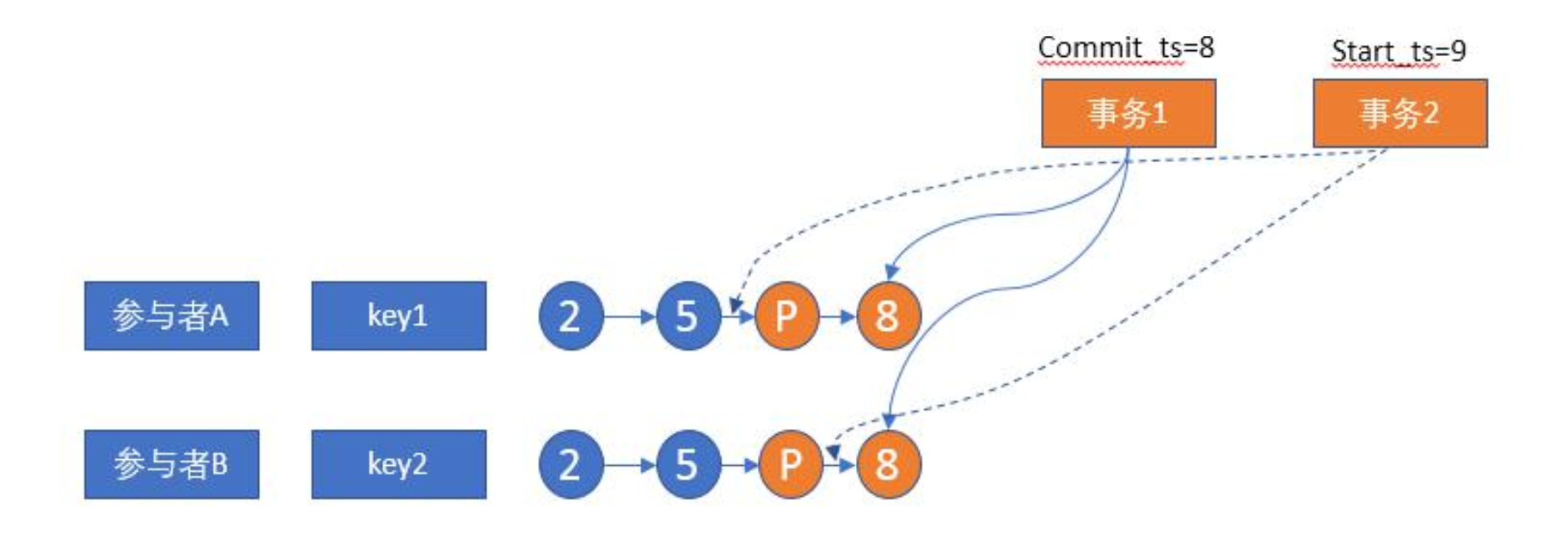
rocksdb乐观事务实现，Write写入事务private空间，不会出现脏读且方便回滚

#### commit\_ts的获取时机

由于网络延迟等原因，在分布式事务中全局单调递增的时间戳到达不同的参与者的顺序可能会不一致。如下图所示，对于分布式事务1、2分别有两个参与者A、B。分布式事务1是写事务，分别在参与者A的key1和参与者B的key2上有写入。分布式事务2是读事务，需要读取key1和key2。现在在分布式事务1进行commit的瞬间，事务2进行read（业务视角）。其中事务2的start\_ts大于事务1的 commit\_ts（内部视角）。如果下图中的参与者A上处于Prepare状态的事务1，对事务2来说完全无感知，那么事务2在参与者A上就读不到事务1的写入，但是在参与者B上能读到事务1的写入，这就出现了部分读的情况，不符合事务的隔离性。所以在分布式事务的2PC中，处于Prepare状态的事务要阻塞其它事务的读取。



仅仅只是阻塞还不够，如下图所示场景在prepare阶段就发生了顺序上的不一致。事务2的start\_ts大于事务1的commit\_ts；参与者A先收到事务2的read，后收到事务1的prepare；参与者B先收到事务1的prepare，后收到事务2的read；。此时，在参与者B上，事务2会被事务1阻塞，直到事务1commit 成功，由于事务2的start\_ts大于事务1的commit\_ts，最终事务2在参与者B上还是会read到事务1的写入，但是在参与者A上事务2由于感知不到事务1的存在，所以read不到事务1的写入，同样破坏了事务的隔离性。



在参与者A上事务1在prepare阶段就和事务2读乱序

所以在TDSQL3.0的事务模型中我们规定，只有在所有的参与者返回prepare成功后，才可以获取commit\_ts，这样就保证如果读事务R的T(R)大于写事务W的T(W)，那么事务R在所有参与者上都能读取到事务W的写入。

#### prepare\_ts的作用

prepare的时候会阻塞读（哪怕start\_ts相对较早），于是我们引入了prepare\_ts目的是做精细化的阻塞控制，这样当读事务的start\_ts大于写事务的start\_ts但是小于写事务的prepare\_ts时不会被阻塞，因为这种读是读老版本的数据没必要阻塞。

### TDStore乐观事务并发控制

### 异常处理与日志恢复

在TDSQL3.0的分布式事务中我们采用了TDStore作为协调者的日志式两阶段提交的方案，由于TDStore实现了RAFT多副本，出现副本中多数派同时故障且无法恢复的可能性极低，而且这种情况一旦发生当前实例的数据已然无法保证完整性，所以我们的方案并不考虑这种异常情况（这种情况需要用备份来进行处理）。因此在下面讨论中都基于一个前提，即副本故障可以恢复。其中参与者的事务状态迁移如下图所示，作为参与者只能在prepare状态才有可能 failed，commit和abort都必须执行成功。同时在进入prepare状态之前，参与者的状态都不需要写日志，因为故障恢复时不需要恢复这些状态。协调者的事务状态迁移如下图所示，由于协调者的日志合并在第一个参与者上，所以在状态迁移时不必记录日志，故障恢复的时候从参与者日志中恢复出状态即可，但是在状态迁移的流程上，关键步骤中都必须先进行本地参与者的状态变更，再通知其它参与者。

## 优化

### Drop database/table大事务消除方案

### raft多副本一致性读

### Select for update（可串行化隔离级别）实现方案

### TDStore事务心跳-租约机制

# 数据分布

# 复制/一致性

## TDSQL复制

## 一致性校验

tdstore\_data\_validation用于检验tdstore各个region的所有副本之间的数据一致性。

其基本原理是：

首先从mc获得所有region副本信息；然后等待所有的region replica jobs执行完毕，以及follower的raft log index追上leader；

对于每个region，在各个tdstore节点上开启一个scan事务，读取tdstore节点上该region副本的kv数据，并写入到一个文件中；

最后调用linux diff命令来比对个副本生成的数据文件，来判断主备数据是否是一致的。

使用方法：

--mc\_server 为必须指定的参数，需要指定mc leader节点的地址。

其余则为可选参数，其中：

--result\_dir region kv数据文件和检验结果文件的存放路径，默认为./kvdiff。

--region\_id 若指定具体的region\_id，则只会检验该region；否则会检验所有的region。

--only\_failed\_region 若设置为true，则只会检验被记录在${result\_dir}/failed\_region\_list.txt （每次检验过程中遇到错误失败的region会被记录在该文件中）中的region；否则会检验所有的region。默认为false。

--wait\_raft\_log\_ready 是否需要等待region的所有副本回放raft log都已经追上leader之后，再开始执行检验。默认为true。

--wait\_jobs\_finished 是否需要等待所有的region/replica jobs都执行完毕之后，再开始执行检验。默认为true。

--timeout\_ms 为与tdstore mc之间rpc的超时时间。当region中数据量较大，且有较多的delete record未来得及compaction清理时，scan事务相关的rpc执行时间可能较长，因此建议设置长一些。目前默认为600s。

--wait\_region\_ready\_retry\_count 等待所有的region replica jobs执行完毕，以及follower的raft log index追上leader的重试次数。

--wait\_region\_ready\_interval\_time\_s 等待所有的region replica jobs执行完毕，以及follower的raft log index追上leader的重试时间间隔。

--print\_check\_region\_stats\_num 检验完毕后，打印region各个副本的统计信息，包括每个副本包含的kv数量（num\_records）、数据量（num\_bytes）、scan事务的执行时间（scan\_time）、输出数据文件的时间（output\_time），以及比对各个副本数据文件的时间（diff\_time）。

以region的总时间开销进行从大到小排序，打印显示前print\_check\_region\_stats\_num个region的信息。

--max\_region\_data\_size\_use\_diff 当region副本的数据文件较大时，直接使用linux diff命令可能会导致oom，因此我们以max\_region\_data\_size\_use\_diff为阈值，当不超过max\_region\_data\_size\_use\_diff时直接调用linux diff；否则，对于大文件采用逐行顺序比对的方法来避免oom。

# 备份恢复

## 备份

## 恢复

# 兼容性

## MySQL兼容性

## Oracle兼容性

# 扩展性

## 弹性扩容

## 异构数据库/多源同步

# 高并发

## SQLEngine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

## TDStore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

## MC高并发

# 高可用

## SQLEngine高可用

## TDStore高可用

## MC高可用

# 数据压缩

# 数据迁移

# 数据校对

# 运维/监控告警

## perf\_monitor

监控region上的读写请求压力。

特性：

1. 每隔3s查询并展示一个TDStore节点中所有region上的读写请求压力
2. 支持按照读压力和写压力排序
3. 支持指定展示的region数量

使用方法：

./perf\_monitor --server=[tdstore\_ip:port] --topn\_type=read|write --region\_num=[num]

## raft\_dump

## raft\_log\_tool

# 应用场景