# 概述

## F1

F1 是 Google 开发的分布式关系数据库，主要服务于 Google 的广告系统，它提供强一致性、高可用性，并支持传统 SQL 查询，近来也常常被称之为所谓的 NewSQL。

F1 是构建于 Spanner 之上的。Spanner 是 Google 开发的全球级数据存储引擎，它保证了数据存储的一致性和可用性，还通过 2PC（两阶段提交）提供了分布式事务读写。在分析 F1 时，我们可以简单地认为 Spanner 是一个全球分布的 kv 数据库。

F1 系统运行时由多台独立的 F1 服务器组成，为了保证整个系统的高可用性，F1 服务器被设计为无状态的，而且不存储数据——节点可以随时上线下线，客户端可以连接至任意节点发送请求。F1 服务器主要职能是将 RDBMS 中的结构化数据映射为可存储于 Spanner 的 kv 对，同时将客户端的 SQL 请求翻译成 get, set, del 等简单的 kv 操作。

Schema 也就是关系数据库中表、列、索引、约束等定义，对应于 SQL 中的 DDL。很显然，Schema 决定了 F1 服务器的具体工作方式，客户端请求的解析和验证由 Schema 决定，之后如何翻译成 kv 操作也由 Schema 决定，Schema 可以被认为是 F1 服务器运行时所依赖的元信息。实践中，F1 服务器运行时自身会缓存一份 Schema 并有一定的机制保持定时更新。

F1 中的 Schema 变更是在线的、异步的，Schema 变更的过程中所有数据保持可用，保持数据一致性，并最大限度的减小对性能的影响。最大的难点在于所有 F1 服务器的 Schema 变更是无法同步的，也就是说不同的 F1 服务器会在不同的时间点切换至新 Schema。

说明：在TDSQL2.0、TDSQL2.5中都采用了proxy/SQLEngine之间广播的方式同步，这种效率不是很高，并不是真正的Online-DDL，TDSQL3.0实现了真正的F1 Online-DDL。

由于所有的 F1 服务器共享同一个 kv 存储引擎，Schema 的异步更新可能造成严重的数据错乱。例如我们发起给一次添加索引的变更，更新后的节点会很负责地在添加一行数据的同时写入一条索引，随后另一个还没来得及更新的节点收到了删除同一行数据的请求，这个节点还完全不知道索引的存在，自然也不会去删除索引了，于是错误的索引就被遗留在数据库中。

# 现有解决方案

参考：

数据库并发控制原理：<https://zhuanlan.zhihu.com/p/464283526>

并发控制协议: 2PL / TS / OCC：<https://zhuanlan.zhihu.com/p/294657612>

## PostgreSQL Create Index Concurrently

PostgreSQL从8.2开始支持create index concurrently，可以在建索引时不阻塞DML。其原理是将create index分成4步做，每一步都是一个单独的事务：

1、先修改数据字典，在表上加SHARE UPDATE EXCLUSIVE锁，阻塞表上后续的Schema修改和vacuum操作；

2、拿到表的一个快照s1，等待表上当前的写事务结束，扫描表中基线数据、创建索引，完成第2步；

3、拿到表的一个快照s2，等待表上当前的写事务结束，再扫描一遍表中的数据，更新索引，等待所有持有比s2小的快照的事务结束，完成第3步。

4、让索引对DML可见。

在PG的方案中，使用快照隔离来创建索引的思想值得借鉴，但具体存在两个主要的问题：

1）不保证DDL进度。虽然建索引不阻塞用户事务，但是用户事务会阻塞建索引。建索引过程中需要等待其他并发事务结束，如果其他并发事务是长事务，就需要一直等下去，甚至没有访问待加索引的表的用户事务也可能导致建索引被阻塞；

2）故障不可恢复。PG中没有针对create index concurrently的故障恢复机制，加索引过程中因为认为或者故障导致中断的话，会导致索引不完整、不可用，并且不完整的索引会一直影响用户事务中的写操作，需要用户手动删除不完整的索引；

3）这是一个针对单机数据库设计的方案，在TDSQL 3.0架构中难以实现。在TDSQL 3.0架构中很难实现全局活跃事务列表，无法保证DDL事务等待系统中所有并发的DML事务结束，因此难以在TDSQL 3.0中实现类似PG这种方案。

此外，由于是针对单机数据库设计的方案，自然也没有考虑多个数据库实例之间如何同步schema的问题。

## MySQL (InnoDB) Instant Add Column

在MySQL5.6之后支持Inplace DDL，即将快照数据导入临时表、再追日志（需要禁写）、最终临时表转正的方式完成，需要复制表中所有数据，且存在禁写。但是这种方式相比更早版本MySQL中的DDL方式已经减少了对写操作的阻塞。

自MySQL8.0开始，在InnoDB中支持Instant Add Column，与PG Create Index Concurrently不同，这一功能的目的是支持顺序加列操作。在sh具体原理如下：

表1. InnoDB中 Instant Add Column行结构示意

|  |  |  |  |
| --- | --- | --- | --- |
| 写入时间点 | info\_bits | #column | row data |
| t0 | 0 | 3 | 1,1,’a’ |
| t1 | 1 | 4 | 2,2,’b’,2 |
| t2 | 1 | 5 | 3,3,’c’,3,3 |

如表1所示，在innodb行结构，利用info bits中原本保留的1个位来标记该行数据中是否包含instant列. 并且在行结构中也记录了该行中列的个数。有了这些信息，在读取数据时，就可以判断出该行数据的末尾部分是否有instant列、有几个这种column。这样就可以根据数据字典中的该表的schema信息、用默认值补齐缺失的instant 列。在数据字典中会保存instant column所有的默认值。

通过这种方式，可以做到在顺序加列时不用重写历史数据，只需要新写入的数据按照最新的schema写入数据即可。例如表1中，在t0时刻，表中没有任何instant 列，数据的行结构和原来的InnoDB没有区别。当在t1时刻写入数据时，表中已经有了一个instant 列（t0-t1之间发生了instant add column操作），那么新写入的数据会把info\_bits中对应位置位并记录当前表中列的个数。t2时刻也是类似。

但是这种方案也存在一些问题：

1、首先和PG Create Index Concurrently一样，这是个针对单机数据库的方案，没有考虑多个数据库实例之间同步schema的问题；

2、只能支持顺序在表的末尾加列，如果发生在指定位置（非末尾）加列、减列、修改列类型、加减索引其他等DDL操作，还是需要通过之前版本MySQL原有的方式执行、将数据拷贝并重写。

不过在MySQL Instant Add Column方案中，加列后不重写元数据的思想值得借鉴。

## XDB Fast DDL

XDB Fast DDL是对MySQL5.6中Inplace DDL的改进。XDB的存储引擎是类似RocksDB的KV Store，采用LSM-Tree存储结构。每个表在每个X-Server上都会存储为一棵单独的LSM-Tree。针对LSM的特性，XDB对Inplace DDL的实现进行了改进，具体的DDL执行步骤如下：

1、禁写、获取待执行DDL的数据表的一个Snapshot；

2、通过Snapshot遍历表中基线数据，进行数据转换，但是转换后的数据不是写入临时表，而是直接写入到LSM的L2中(不经过mem table)。

3、与第2步同时进行，放开写。由于转换后的数据是直接写入L2的，因此mem table、L0、L1中仍然可以接受新的写入，但是需要禁止LSM compaction机制向L2 merge数据。这一过程中mem table、L0、L1相当于是一个增量数据的write buffer；

4、待步骤2结束后，将L2与memtable,L0,L1合并，禁写，变更结束

5、放开L2 merge限制。

相比MySQL原生的Inplace DDL，XDB的执行流程中不需要去追log，避免了一直追不上log的问题。同时这一方案也减少了数据转换带来的写放大。

但是这种方案是依赖于底层存储的特殊设计的：每个表在每个节点上都会有一颗单独的LSM-Tree，表的一个分片所对应的数据记录和索引记录都在同一棵LSM-Tree中。只有这样，这种基于Compaction实现复杂数据转换（如补索引记录）才好实现。在TDSQL 3.0的架构中，数据记录及其对应的索引不保证在同一棵LSM-Tree中，甚至很多时候会在不同的节点上，要实现类似XDB的方案很困难。但是对于只涉及数据记录本身的数据转换（如增减列），XDB的做法具有借鉴意义，通过将数据转换下沉到TDSTore、并延迟到compaction时做，可以减少数据的写放大。

在schema同步方面，XDB同一个分片的主备之间通过paxos/raft log来同步schema，因此分片主备之间是schema强同步的。在第1步和第4步中，leader会分别写一条begin ddl和end ddl日志。而不同分片之间是没有schema强同步机制的。当DDL产生一个新版本的schema之后，新的schema只保存在系统表中，只有执行ddl的节点知道这个新的schema，其他各个X-Server并不能立即同步地获得这个新的schema，只有当新schema向某个分片写入数据时，所在的X-Server才能获得最新的schema。如果没有数据写入，则需要等待一段时间（如1-2s）,X-Server才会异步地获取最新的schema。这可能导致查询在一个较小的时间窗口内从不同的分片中读到schema不一致的数据。

## Google F1 Online Schema Change

Google F1的Online Schema Change方案有比较完整的论文介绍：

Online, Asynchronous Schema Change in F1(VLDB 2013)

在DDL执行，也就是数据转换这一块，F1将DDL分为两类：Logical DDL和Physical DDL。其中Logical DDL是指不需要立即进行全表数据转换的DDL，如修改表名、修改字段默认值等；Physical DDL是指需要立即扫描全表基线数据、进行数据转换的DDL，如加索引。

为了实现无状态SQL节点之间的schema和无阻塞的数据转换，F1采用了schema多状态（相当于多版本），允许集群中不同的SQL节点上同时存在多于1个schema状态，并采用了一套转换机制，保证不会发生数据不一致，并进行了证明。关于多状态（版本），F1论文中论证了要在无状态集群中实现schema的同步，必然要允许集群中同时存在至少2个状态。

为了保证集群中所有SQL节点在某个时间点后都能进入最新的状态，F1采用了租约机制。对于一个SQL节点，必须在一个租约内从一个全局的schema存储系统（相当于全局数据字典）读取最新的schema，否则就自动下线。这样可以保证整个集群在一个租期内都会获取到最新的状态。

由于不同的SQL节点可能处于不同的状态，为了保证数据一致性，F1在两个public的schema之间引入了一些中间状态，以加索引为例，当一个新的schema刚刚被创建后，集群中的节点都还处于absent状态，没有感知到新的schema。一个租期过后，所有的节点都进入了delete only状态。当一个节点进入delete only状态后，该节点上执行的删除操作就可以使用新的schema了。再过一个租期后，所有的节点都进入write only状态。当一个节点进入write only状态后，该节点上的delete/update/insert操作就可以使用新的schema了，但是读操作还使用老的schema。此时，集群中启动数据转换任务，以当前的时间戳t0获取一个基线数据的快照，扫描基线数据记录，生成索引记录，采用Thomas Write Rule、以t0为数据的版本号将索引记录写入KV存储引擎（Spanner）中。



图1. Google F1加索引过程中集群的状态转换

这一数据转换过程是可重入的，并且和其他用户事务没有互相阻塞。由于数据转换过程中，新的Schema已经在集群所有节点上处于write only状态，新写入的数据都会维护索引记录。并且由于新schema对读操作不可见，因此未完成的数据转换对读操作也不可见，不违反数据一致性。当数据转换完成后，集群中所有节点都在一个租期后进入public状态，新的索引对读写操作都可见，加索引过程完成。

之所以需要在write only之前有一个delete only状态，是因为无法保证所有节点同时进入write only状态。如果没有delete only用作过渡，可能出现这样的数据异常：一个节点N1已经进入write only状态，用新的Schema写入一条数据r1，并写入索引记录r1\_index，另外一个节点N2还处于absent状态，如果执行删除操作将r1删除，由于新的索引对N2不可见，它不会去删除r1\_index。之后当新的Schema进入public之后，就会出现可以读到r1\_index，却无法根据r1\_index找到对应的r1这种异常。

对于一些logical ddl，由于不用数据转换，中间状态会少一些。

F1的方案具有完备的理论证明，可以处理F1中各种不同类型的DDL，而不是像PG和MySQL中针对特定类型DDL设计的特定方案。但是F1这种方案也有一些缺点：

1、所有的DDL操作都至少需要两个租期才能执行完成。租期大小根据集群大小设定，对于较大的集群，租期可能是分钟级别的。这就导致DDL本身执行效率并不高，而且难易并发执行DDL。F1中DDL操作是先提交到一个类似git的版本控制系统中，然后定期由人工（如每星期2次）提取出来串行执行；

2、此外，正是由于schema同步是基于租约的实现的，和事务没有关系，因而做不到schema的可重复度，在一个事务中可能先后两次读到同一个表的不同schema结构，这对于业务来说有时会很奇怪、引发异常。

## TiDB Online DDL

TiDB中的Online DDL是对F1方案的开源实现。在实现上做了一些优化，尤其在数据转换任务的执行上，F1是采用MapReduce来执行数据转换，TiDB则设计了一套任务队列机制、利用TiDB的SQL节点来分布式执行数据转换任务。这样不用再另外维护一套MapReduce计算引擎。CockroachDB中借鉴F1方案进行了开源实现。

# 分类

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

# 原理

参考：

谷歌 F1 Online DDL的关键点：状态间兼容性：

<https://zhuanlan.zhihu.com/p/120719499>

理解 Google F1: Schema 变更算法：

<https://disksing.com/understanding-f1-schema-change/>

[《Spanner: Google’s Globally-Distributed Database》论文翻译](http://blog.mrcroxx.com/posts/paper-reading/spanner-osdi2012/)

[Google F1是如何做Schema变更的](https://juejin.cn/post/6844903623109902350)

[分布式 Schema 变更在 Google F1 的实践](https://zhuanlan.zhihu.com/p/367041904)

## 租约

F1中 Schema 以特殊的 kv 对存储于 Spanner 中，同时每个 F1 服务器在运行过程中自身也维护一份拷贝。为了保证同一时刻最多只有 2 份 Schema 生效，F1 约定了长度为数分钟的 Schema 租约，所有 F1 服务器在租约到期后都要重新加载 Schema。如果节点无法重新完成续租，它将会自动终止服务并等待被集群管理设施重启。

## 中间状态

前面已经提过，F1在Schema变更的过程中，会把一次Schema的变更拆解为多个逐步递进的中间状态。实际上我们并不需要针对每种Schema变更单独设计中间状态，总共只需要两种就够了：delete-only和write-only。

delete-only指的是Schema元素的存在性只对删除操作可见。

例如当某索引处于delete-only状态时，F1服务器中执行对应表的删除一行数据操作时能“看到”该索引，所以会同时删除该行对应的索引，与之相对的，如果是插入一行数据则“看不到”该索引，所以F1不会尝试新增该行对应的索引。

具体的，如果Schema元素是table或column，该Schema元素只对delete 语句生效；如果Schema元素是index ，则只对delete和update语句生效，其中update语句修改index的过程可以认为是先delete后再insert，在delete-only 状态时只处理其中的 delete 而忽略掉 insert 。

总之，只要某Schema元素处于delete-only状态，F1保证该Schema元素对应的kv对总是能够被正确地删除，并且不会为此Schema元素创建任何新的kv对。

write-only指的是Schema元素对写操作可见，对读操作不可见。

例如当某索引处于write-only状态时，不论是insert、delete，或是update，F1都保证正确的更新索引，只是对于查询来说该索引仍是不存在的。

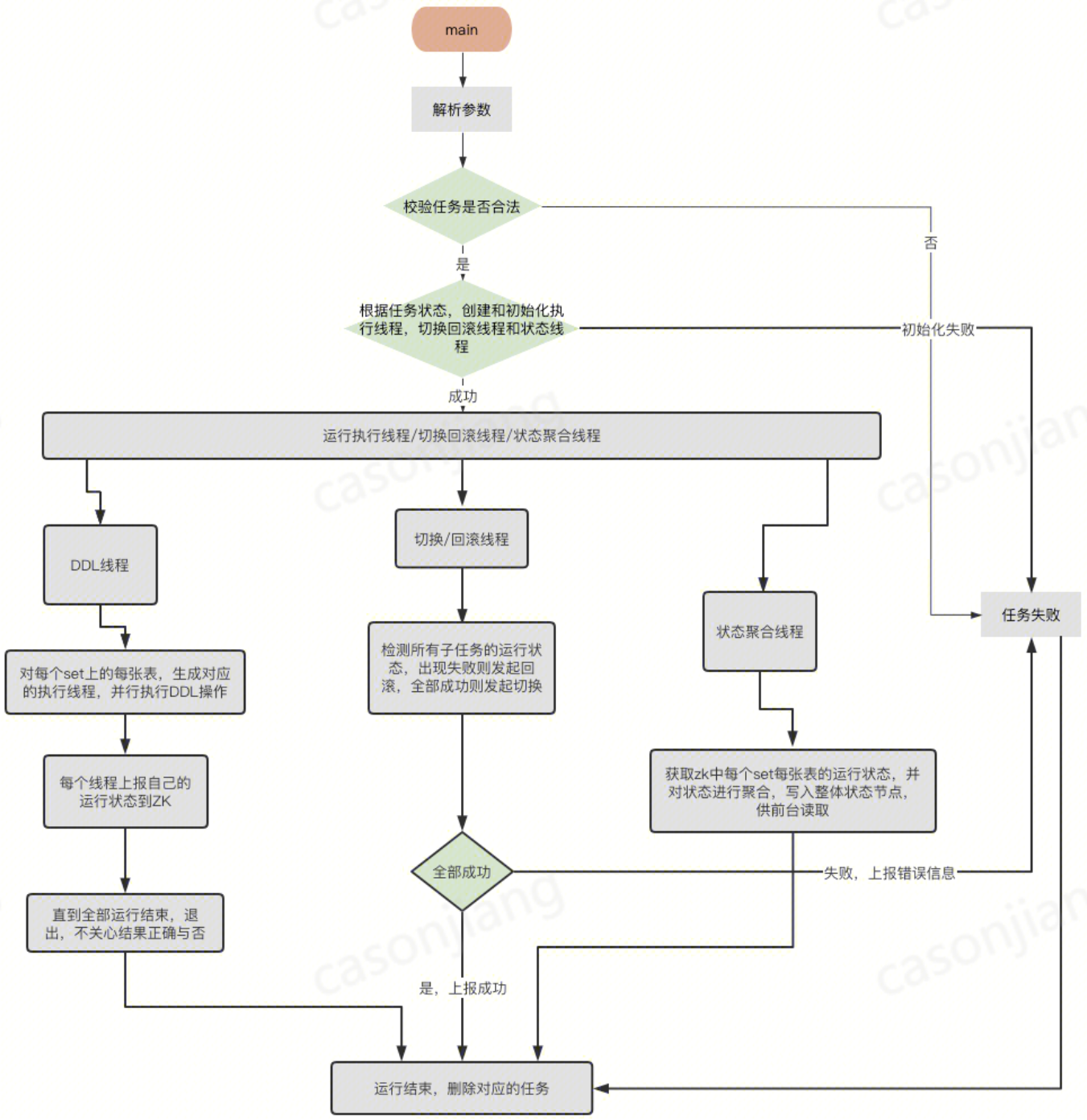
简单的归纳下就是write-only状态的Schema元素可写不可读。

# 分布式数据库实践

## TDSQL2.0

### 背景

DDL当前的调用流程如下：



存在的问题如下：

1. 任务的信息获取和解析分散在各处，校验混乱，反复和zk交互获取任务和解析。
2. 任务状态未得到统一的管理，本地的状态需要上传然后再读取，有大量和zk的无用交互，延迟大。
3. 任务状态的变化不清晰，状态的转换散落在各处，没有清晰的转换逻辑，难以维护和分析。
4. zk链接不统一，每个线程/方法各自使用自己新建的zk链接，异常情况下的重连，失败重试等难以得到较好处理。
5. 没有对 SQL 的语法解析，无法处理后续更复杂的各种 ddl sql。

针对网关对二级分区表DDL可能出现的多个分片结果不一致的情况，计划由ONLINE-DDL模块来完成对二级分区表的DDL操作，保证多个分片的DDL原子性。

针对二级分区表的DDL操作，包含下列类型：create table，drop table , rename table, alter table add/drop/moidfy column/index, truncate table.

* 与proxy的交互
* 任务下发
* 任务返回
* DDL执行方案：

用户权限处理

create table

drop table

rename table

alter table

truncate table

状态上报

错误容灾处理

### 方案设计

#### DDL两阶段

需要关注的点：

1、OnlineDDL与Proxy的链路打通（包括proxy任务的下发及结果返回）。

2、在现有的OnlineDDL模块中需要实现create table、rename table、truncate...等不走pt的逻辑，确保分布式场景中的原子性。

具体需要做的事情：

1、除了alter table外其他基本都不走pt逻辑，不走pt的逻辑需要重构一下

2、OnlineDDL模块中实现具体的逻辑（create/rename/truncate table...），需要公共抽象一下

1）create table a

创建临时表 a\_tmp 。

所有的set上临时表创建成功，进行切表 a\_tmp -> a，这步必须成功，否则进行重试，如果超过指定次数后进行回滚操作，并返回错误信息。

2）rename table

在所有set上先将a rename为一个临时表，a -> a\_tmp

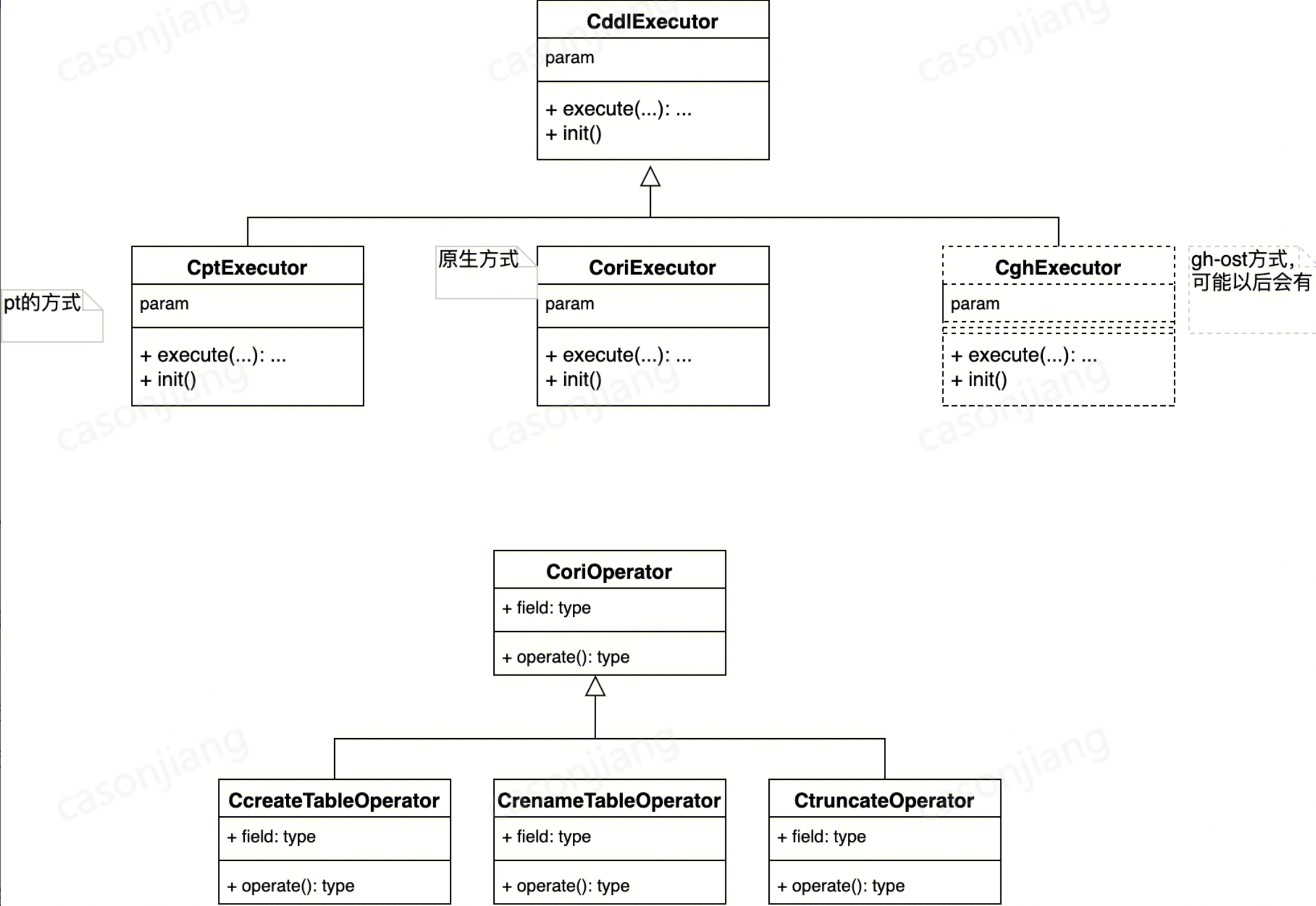
将表a\_tmp重命名为表b，a\_tmp -> b

3）truncate table

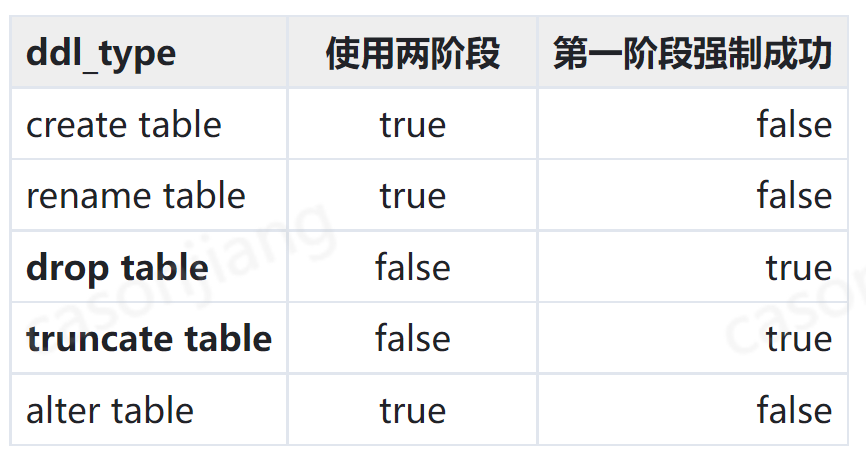
3、上报错误需要更加清晰，具体哪个set及什么错误，最好单独封装为类

4、任务执行的状态，整体的状态及每个set的状态缓存。

**非pt逻辑剥离方案：**



**DDL两阶段提交：**



#### 二级分区方案

实现DDL操作的最终一致性，即通过自动地重试或者回滚等手段，保证DDL操作最终完成或者恢复原来状态。不确保实时一致性，因此执行期间可能会看到中间状态（例如部分SET成功）。

##### 整体方案

###### proxy逻辑

简化proxy的处理逻辑，proxy直接将ddl交由onlineDDL组件执行

1、每个group的都有自己的独立的ddl任务表（默认存放在最后一个set上）

CREATE TABLE xa.ddl\_tasks\_t (

ask\_id int PRIMARY KEY AUTO\_INCREMENT, -- 任务ID

dbn varchar(64), -- 库名

tbn varchar(64), -- 表名

user varchar(64), -- 用户名%IP地址，用于权限检查

executor enum('onlineddl', 'proxy'), -- 执行者是网关还是online ddl组件

ddl text, -- DDL内容（格式为json，存放了每个set上需要执行的DDL语句）

meta\_modified text, -- DDL完成之后的元数据信息json格式

result enum('Init', 'Run', 'Failed', 'Success', 'canceled'), -- 任务状态

message varchar(128), -- 错误信息

INDEX idx\_dbn\_tbn(dbn, tbn));

其中meta\_modified描述了DDL完成之后需要修改的元数据信息，例如:

json[{ "type": "ZK", "operator": "add or replace""path": "/noshard1/group\_1576218409\_2013436/routes/rtns@shardkeys/rt@th.t1", "value": {"auto\_inc":"","auto\_inc\_step":0,"auto\_inc\_type":1,"dbtable":"th.t1","shardkey":"a"}}, {"type": "DB", "operator": "add or replace""name": "test.t1", "value": {"auto\_inc":"","auto\_inc\_step":0,"auto\_inc\_type":1,"dbtable":"th.t1","shardkey":"a"}}]

2、proxy接受到ddl请求之后，先向任务表中插入一条记录，并获取该任务到task\_id

3、如果ddl要交给onlineDDL执行，则proxy用任务的task\_id值来更新Zookeeper上的节点，例如${rootdir}/groupid/ddl\_task\_max\_id，告知Online DDL组件当前最新的DDL任务的task\_id。

4、online ddl组件每完成一个DDL任务，便通过更新${rootdir}/groupid/ddl\_task\_ans\_lastest\_id，向所有网关广播该任务的task\_id值。

5、proxy监听到任务完成事件之后：从DDL任务表中获取"正在执行"的任务的状态信息，如果状态为'Failed'或者'Success'，则向对应的客户端返回成功或者失败

###### online ddl组件

1、部署方式：以集群模式存在，多个group共享onlineDDL集群，便于升级和运维；和proxy共同部署，杜绝了流量单点问题，运维升级麻烦。

2、如果以集群方式部署，onlineDDL要确保group之间任务的公平性，避免因为个别group的任务阻塞其他group的任务

3、如果以集群方式部署，onlineDDL需要从group的任务表中读取任务，因此需要感知group的路由信息（如果并行执行ddl任务，还需确保互斥性质）

4、需要对DDL进行鉴权（可以创建和原始用户同权限的临时用户）

5、执行完DDL任务之后，还需要进行元数据的更新（zk、db上的路由信息）

6、onlineDDL任务如何取消？（proxy收到ctrl-z，or来自前台的取消）

7、在执行DDL任务之前，如果已经监测到了group上各个set的表结构不一致，该如何处理？

##### 网关执行DDL的流程

1、proxy收到用户的DDL请求之后，检查是否存在冲突的DDL任务，如果没有，则基于最新的元数据检查DDL是否合法；如果检查成功，则向DDL任务表中插入一条记录，具体流程如下：

* 获取与最后一个SET的连接(存放DDL任务表)；
* 获取DDL排他锁(名称为边名)：select get\_lock("表名", 1);
* 检查DDL任务表（即xa.ddl\_tasks\_t），目标库表上是否存在冲突的任务；如果有，则释放排他锁，并加入到等待队列，等到冲突的任务结束再进行重试，或者等待超时直接返回；
* 基于当前的元数据检查DDL任务是否合法（此时元数据必然是一致的）
* 向DDL任务表中插入一条记录，并获取记录对应的task\_id值。

CREATE TABLE xa.ddl\_tasks\_t (

task\_id int PRIMARY KEY AUTO\_INCREMENT, -- 任务ID

dbn varchar(64), -- 库名

tbn varchar(64), -- 表名

user varchar(64), -- 用户名%IP地址

executor enum('onlineddl', 'proxy'), -- 执行者是网关还是online ddl组件

ddl text, -- DDL内容（格式为json，存放了每个set上需要执行的DDL语句）

meta\_modified text, -- DDL完成之后的元数据信息json格式

result enum('Init', 'Run', 'Failed', 'Success'), -- 任务状态

message varchar(128), -- 错误信息

INDEX idx\_dbn\_tbn(dbn, tbn)

);

其中meta\_modified格式为json，例如：

[

{

"type": "ZK",

"path": "/noshard1/group\_1576218409\_2013436/routes/rtns@shardkeys/rt@th.t1",

"value": {"auto\_inc":"","auto\_inc\_step":0,"auto\_inc\_type":1,"dbtable":"th.t1","shardkey":"a"}

},

{

"type": "meta",

"name": "test.t1",

"value": {"auto\_inc":"","auto\_inc\_step":0,"auto\_inc\_type":1,"dbtable":"th.t1","shardkey":"a"}

}

]

2、如果DDL使用OnlineDDL组件来执行，则proxy用任务的task\_id值来更新Zookeeper上的节点：${rootdir}/groupid(setid)/ddl\_task\_max\_id，告知Online DDL组件新的DDL任务；如果DDL由Proxy自身来执行，则向后台DDL线程提交DDL任务。

3、proxy挂起当前会话，等待ddl完成（是否需要设定超时？）。

4、online ddl组件或者Proxy每完成一个DDL任务，便通过更新${rootdir}/groupid(setid)/ddl\_task\_ans\_lastest\_id，向所有网关广播该任务的task\_id值。

5、proxy监听到任务完成事件之后：

从DDL任务表中获取"正在执行"的任务的状态信息，如果状态为'Failed'或者'Success'，则向对应的客户端返回成功或者失败；

唤醒由于冲突而处于"等待状态"的DDL，进行重试。

##### 网关执行DDL具体步骤

将所有的DDL操作都抽象成DDLTask：

class DDLTask {

public:

virtual bool Rollbackable() = 0; // 是否能够回滚

virtual bool CheckDDL() = 0; // 检查DDL的合法性

virtual bool Recover() = 0; // 用于重试时，恢复到上次奔溃前的状态

virtual bool Rollback() = 0; // 回滚DDL

virtual bool Run() = 0; // 执行DDL

};

###### 网关DDL执行线程

网关DDL执行线程：

1、获取最新的表元数据信息（当前必然没有正在运行的其他Online DDL）

2、根据元数据信息，检查DDL是否合法：task->CheckDDL()

3、获取排他锁，避免重试线程执行：select get\_lock("test.t1\_DDL\_mutex", 0);

3、更新xa.ddl\_tasks\_t表，将task状态设置为'Run'

4、执行DDL（包括修改元数据）：task->Run();

5、如果成功则释放资源，并返回成功；

6、并向用户报错。如果任务可以回滚（即task->Rollbackable()）,则回滚任务（task->Rollback（））；否则，释放资源，交给重试线程重试；

###### 网关重试线程

网关重试线程：

1、从重试队列中获取需要重试的任务；如果队列为空，则从xa.ddl\_tasks\_t获取executor为'proxy'，状态为'Run'的任务；

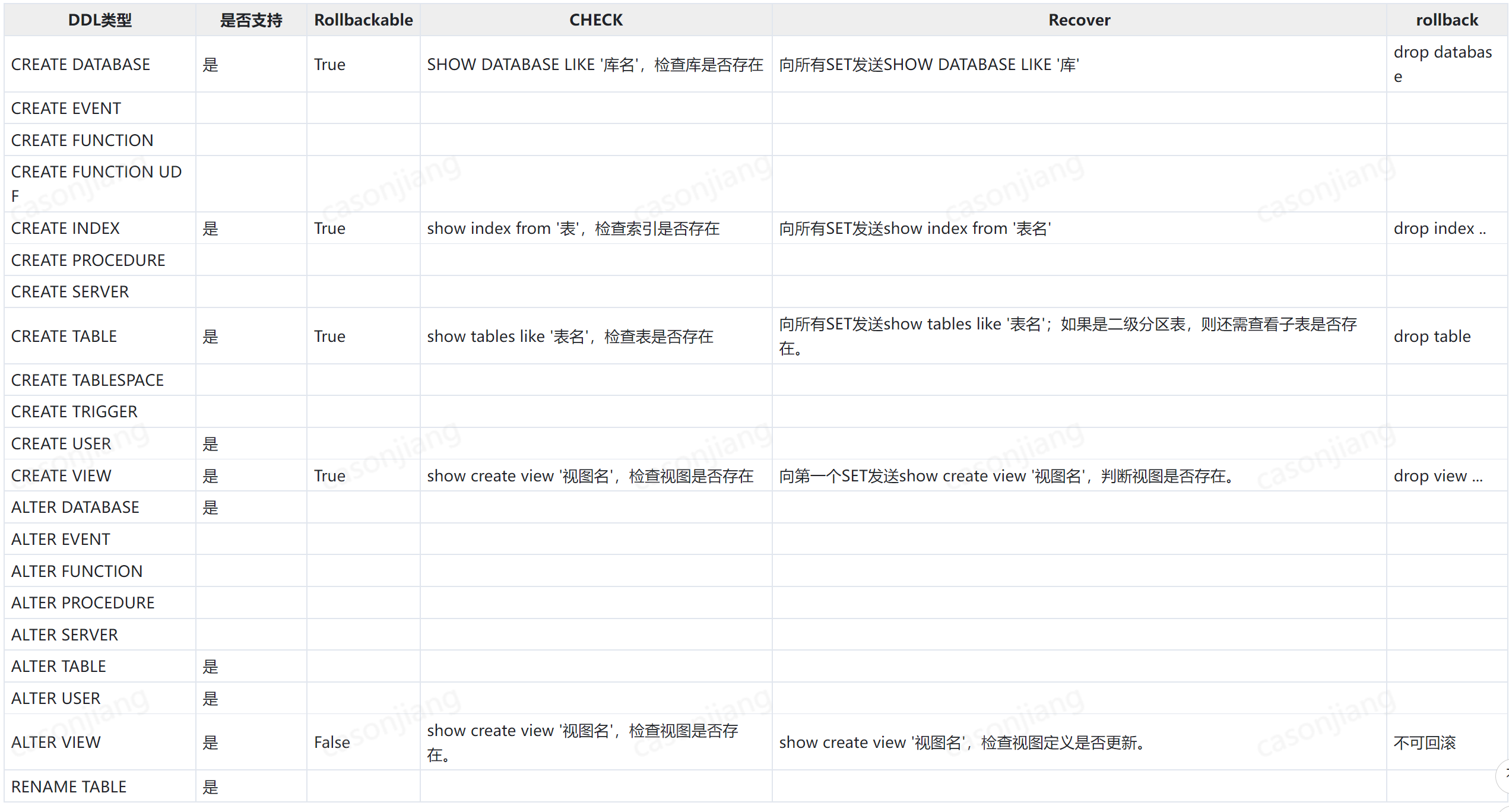
2、获取排他锁：select get\_lock("test.t1\_DDL\_mutex", 0);

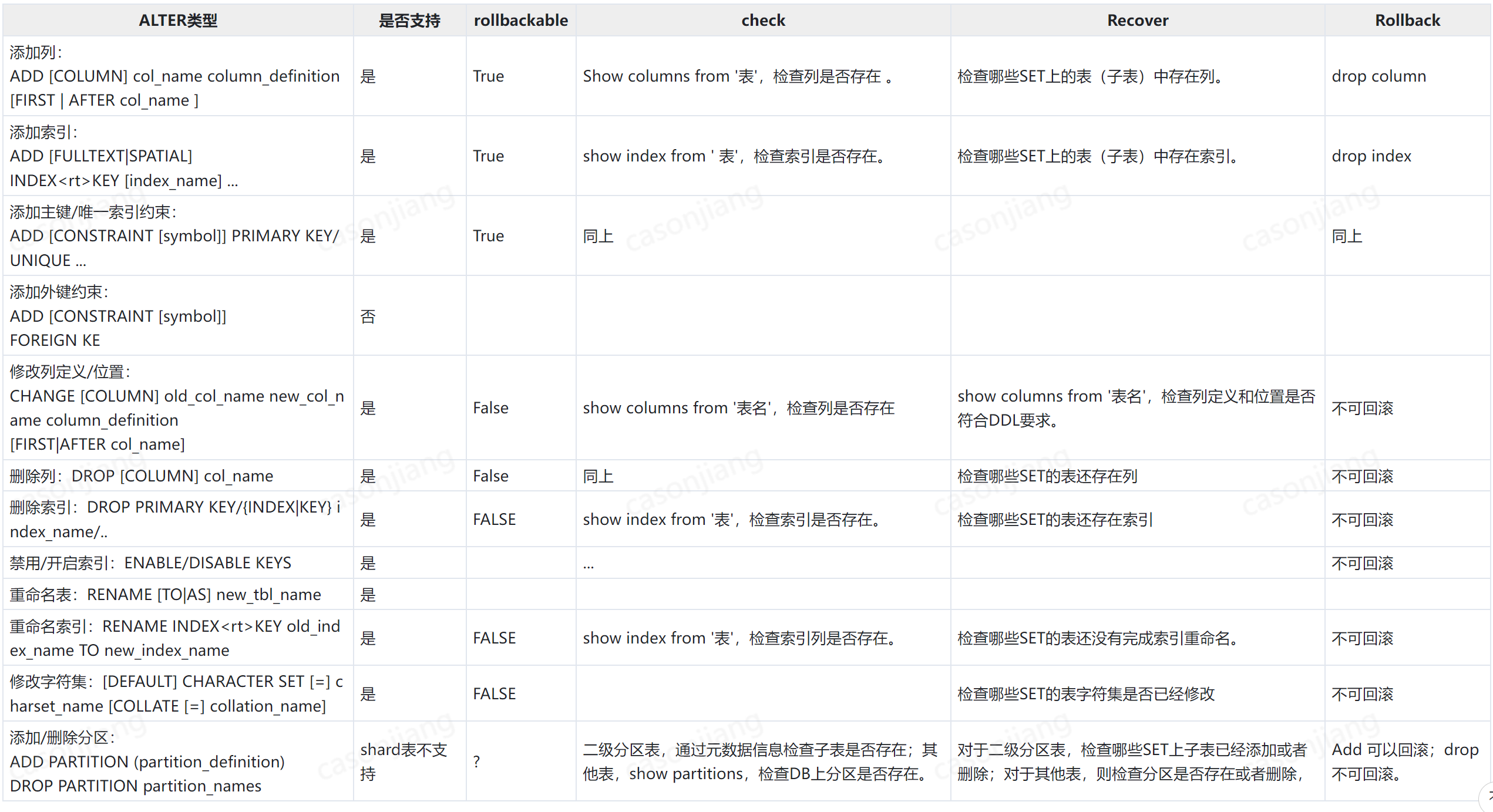
3、确保任务在xa.ddl\_tasks\_t的状态还是'Run'：如果状态不为'Run'，则重新获取下一个任务；

4、根据任务信息构建task对象，并调用task->Recover()得到当前DDL的执行情况（例如哪些set执行成功了，哪些还没有等）；

5、如果任务可以回滚，则回滚任务（task->Rollback())；否则进行推进任务(task->Run());

6、如果执行失败，则将任务在此放入到重试队列中，等待下一次重试（重试次数超过一定值，触发告警？）。





##### online ddl

1、监听到{rootdir}/groupid(setid)/ddl\_task\_max\_id发生变化，则从DDL任务表中取出需要执行的任务（已经完成的DDL任务的最小task id可以存放在 {rootdir}/groupid(setid)/ddl\_task\_min\_id），并加入到执行队列等待执行（可以按表或者按库？）。

2、执行DDL

3、如果DDL任务完成之后需要更新元数据(任务的meta\_modified不为空，或者当前DDL为删除库或者删除表)：

3.1、向最后一个SET（路由信息陈旧该怎么办？）的xa.table\_meta\_t写入表的元数据信息，并获取最大的版本号：

CREATE TABLE xa.table\_meta\_t (

epoch int AUTO\_INCREMENT, -- 版本号

dbn varchar(64), -- 库名

tbn varchar(64), -- 表名

delmark bool, -- 删除标记

meta text, -- json格式

create\_timestamp timestamp NOT NULL DEFAULT CURRENT\_TIMESTAMP,

update\_timestamp NOT NULL DEFAULT CURRENT\_TIMESTAMP ON UPDATE CURRENT\_TIMESTAMP,

PRIMARY KEY (dbn, tbn),

INDEX idx\_epoch(epoch)

);

说明：xa.table\_meta\_t中meta字段存放表的元数据信息，比如shardkey、二级分区键、分区函数等。内容为json格式：

{"auto\_inc":"","auto\_inc\_step":0,"auto\_inc\_type":1,"dbtable":"th.t1","shardkey":"a"}

和老版本proxy在zookeeper上的元数据信息保持一致：

[zk: localhost:2181(CONNECTED) 1] get /noshard1/group\_1576218409\_2013436/routes/rtns@shardkeys/rt@th.t1

{"auto\_inc":"","auto\_inc\_step":0,"auto\_inc\_type":1,"dbtable":"th.t1","shardkey":"a"}

**并发DDL元数据覆盖的问题：**

例如对二级分区添加子分区的操作，session1执行：alter table t1 add partition p1 values less than(100); 任务中meta\_modified为：

<test.t1 ...>

<p0 range\_begin="0000000000" range\_end="0000000050" />

<p1 range\_begin="0000000050" range\_end="0000000100" />

</test.t1>

session 1尚未执行完毕，且session 1的元数据尚未同步完成时，session2执行：alter table t1 add partition p1 values less than(200); 任务中meta\_modified为：

<test.t1 ...>

<p0 range\_begin="0000000000" range\_end="0000000050" />

<p2 range\_begin="0000000050" range\_end="0000000200" />

</test.t1>

此时session2执行之后的元数据会覆盖掉session 1产生的元数据，导致分区p1不可用。

proxy执行DDL任务时，如果DDL需要修改table\_meta\_t的元数据，则按如下方式进行：

1、获取名称为表名的session 级别的mutex，避免对同一个表的并发DDL：

select get\_lock("表名", 1);

2、查看xa.ddl\_tasks\_t表，查看当前在同一张表上是否存在状态为'run'的任务，如果存在直接报错（或者重试？）

3、读取xa.table\_meta\_t表，获取当前表最新的元数据；基于最新的元数据对DDL进行检查，并构建meta\_modified

-- 如果表已经存在，则epoch值会自动加1

replace into xa.table\_meta\_t(db, tbn, delmark, meta) values('test', 't1', false, '....');

-- 如果DDL是删除库/表。则直接将对应记录设置删除标记

replace into xa.table\_meta\_t(db, tbn, delmark) values('test', 't1', true);

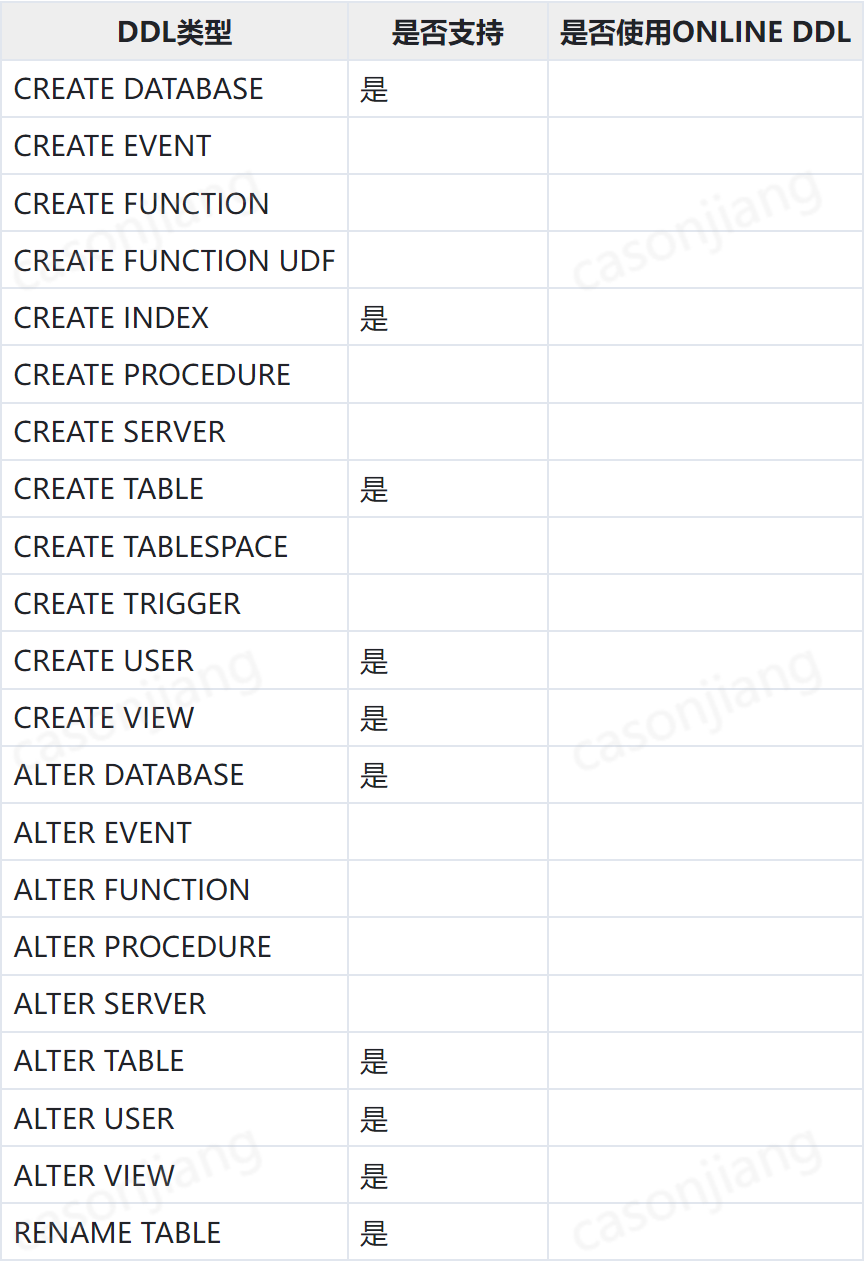
​3.2、更新Zookeeper的${rootdir}/current\_meta\_epoch，向所有网关广播元数据更新的消息；

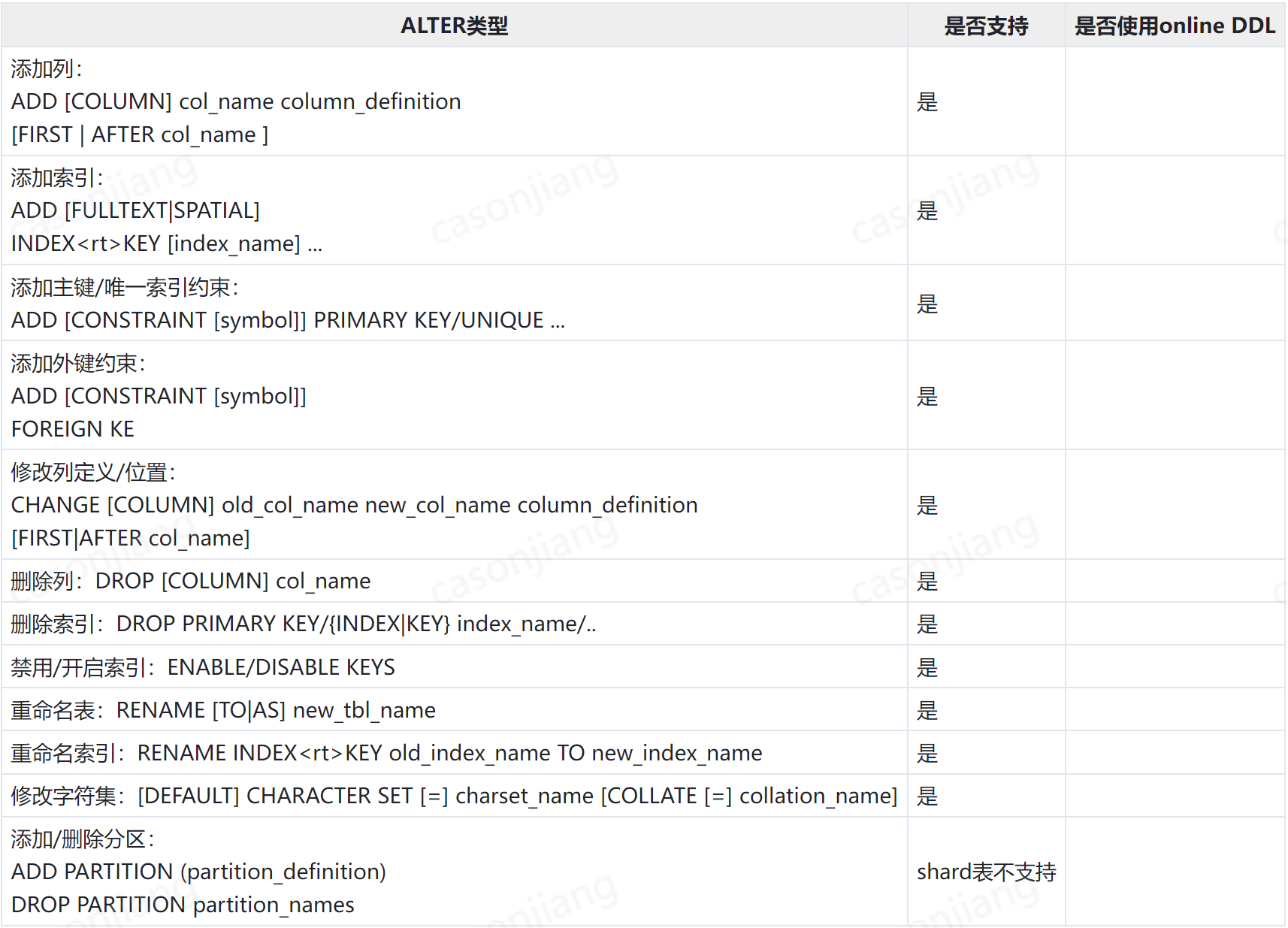
3.3、如果需要兼容老版本的proxy（${rootdir}/new\_meta\_store）,则还需要更新Zookeeper上的元数据；

4、任务完成之后，使用当前任务的task id更新Zookeeper的节点，${rootdir}/groupid(setid)/ddl\_task\_ans\_lastest\_id，告知proxy当前已经完成的任务。

5、router-update会监听Zookeeper上的"${rootdir}/groupid(setid)/current\_meta\_epoch"，并重新加载(last\_meta\_epoch, current\_meta\_epoch)变化区间内xa.table\_meta\_t中的元数据信息（last\_meta\_epoch为router本地的一个变量）。

6、proxy也会重新加载被更新的表的表结构信息到本地。





##### Online-DDL在ZK失效情况下方案

###### 目前的方案

1、在pt执行过程中，通过管道将plugin.pl的日志输出出来在步骤会在内存中进行记录缓存，等到pt执行完成后进行本地和zk的状态步骤stage比对,只有本地比zk更新的情况下才会更新本地到zk上。

在实现上使用一个枚举来确定步骤的编号：

enum STAGES\_NO {

before\_create\_new\_table,

after\_create\_new\_table,

before\_alter\_new\_table,

after\_alter\_new\_table,

before\_create\_triggers,

after\_create\_triggers,

before\_copy\_rows,

after\_copy\_rows,

before\_swap\_tables,

after\_swap\_tables,

before\_update\_foreign\_keys,

after\_update\_foreign\_keys,

before\_drop\_old\_table,

after\_drop\_old\_table,

before\_drop\_triggers,

before\_exit

};

2、如果pt执行结束后，zk出现极端情况依旧没有恢复，这时候会一直轮询上报zk。

###### 需要待确定方案

遇到的问题

如果所有的pt都执行完了，并且一直轮询上报zk知道超过阈值后的处理方案。

1、退出所有线程，performer进程结束，因为一直挂着也不行（就剩切表和删触发器），这时候回滚不了，因为连接不上zk获取不到正确的db连接信息，如果将db连接信息持久化到某个地方，在回滚的时候需要考虑中途发生主从切换的场景，但是持久化的db信息不是最新的。

2、因为performer在退出前zk失败连接不上，无法删除onlineddljob@job\_id节点。

3、由于zk连接不上，mng进程无法正确认为任务结束，虽然performer退出了，但是又会被mng进程拉起来。

考虑的解决方案

以避免人工介入为目的

1、performer在超时后正常退出，保留现场。

2、mng在重新拉起performer前判断下，如果zk没有恢复就不拉起，直到zk正常（这时候zk上的onlineddljob@job\_id、...dispatch@job\_id，这些节点还在，无非就是表的ansnode不是正确的）。

3、在performer被拉起后，可以考虑简单的续做问题，如果这个job的表就剩switch阶段了，那就进行switch操作；如果这个job的表还有没copy完成的，那么就回滚，后续可以拉个需求单考虑copy间断的续做。

4、唯一需要考虑的就是performer退出前，需要将表的当前状态持久化到某个地方，例如执行到那个阶段了，拷贝中还是已经拷贝完了等着切表。

5、这时候zk正常了，同步状态到zk上，job完成。

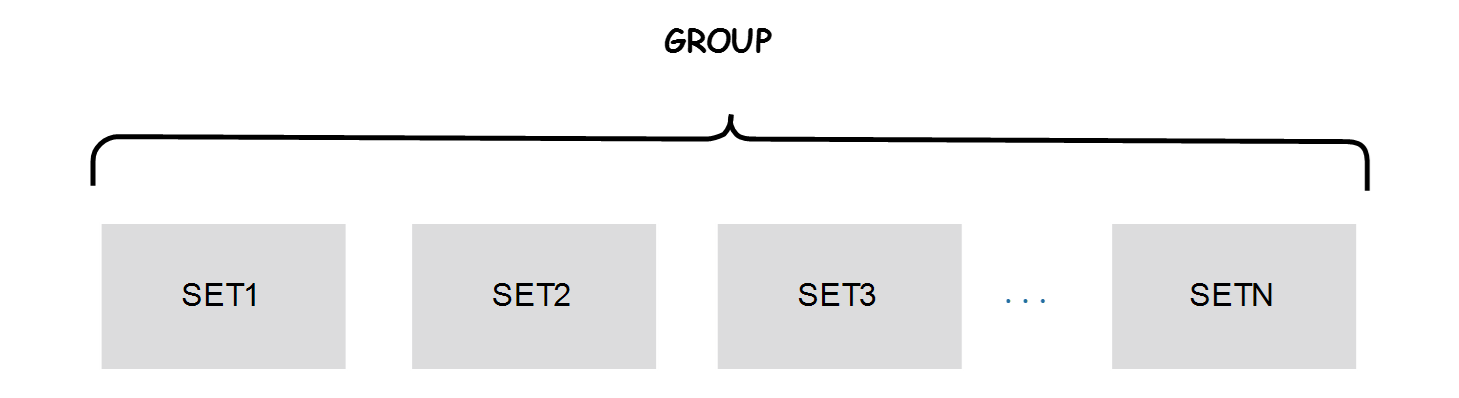
###### 确定的方案

1、performer在超时后正常退出，保留现场。

2、mng在重新拉起performer前判断下，如果zk没有恢复就不拉起，直到zk正常。

3、在performer被拉起后，从zk上获取job状态，整体job状态如果到达即将switch阶段，进行切表操作，否则进行回滚

### 原理



解决的关键问题：如何保证一个GROUP下多个SET执行PT-OSC的原子性，即要么同时成功，要么同时不成功。



1、以任务的方式来封装每个DDL操作。

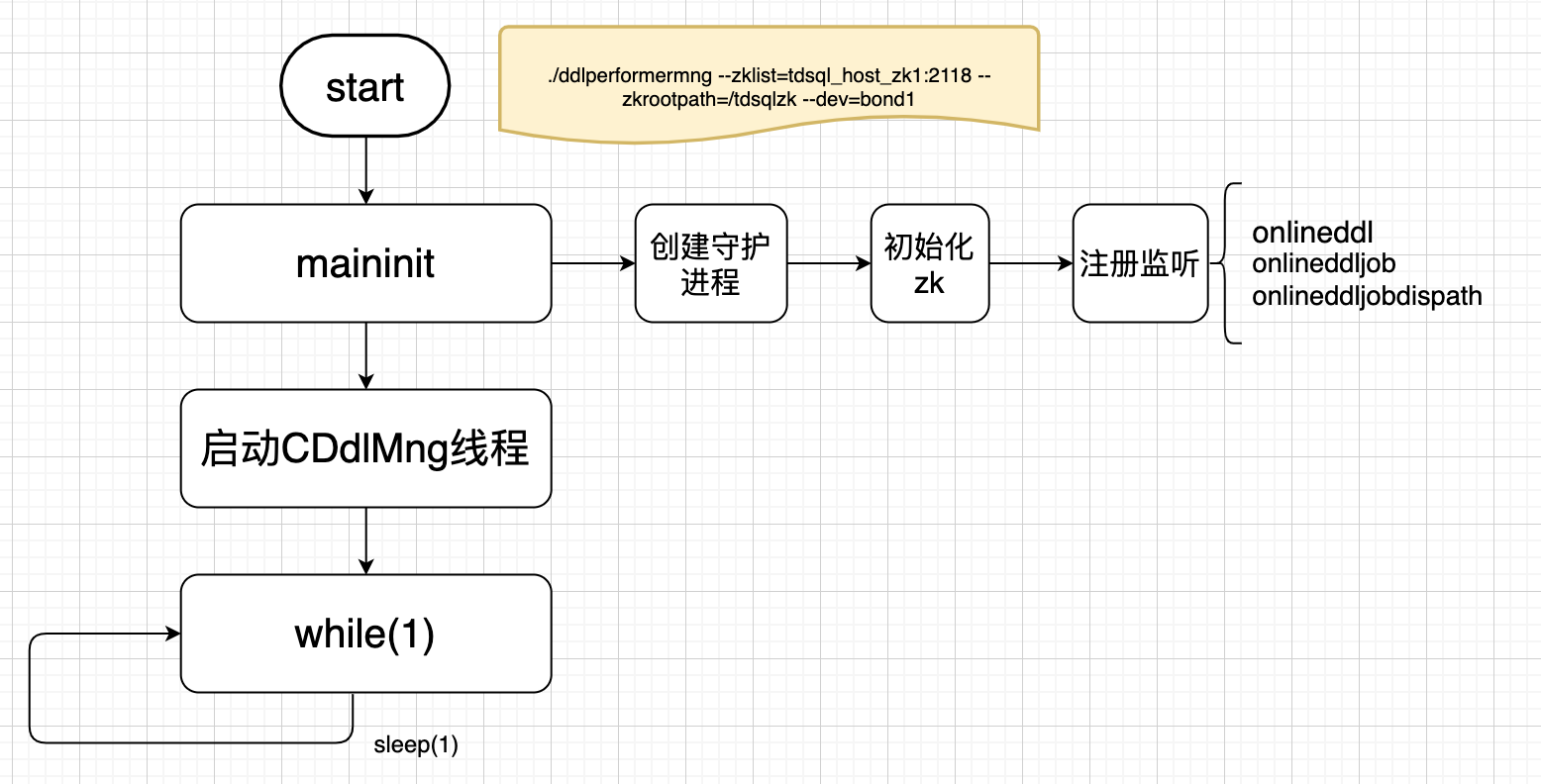
2、独立的进程去控制协调每个节点PT工具的执行情况。

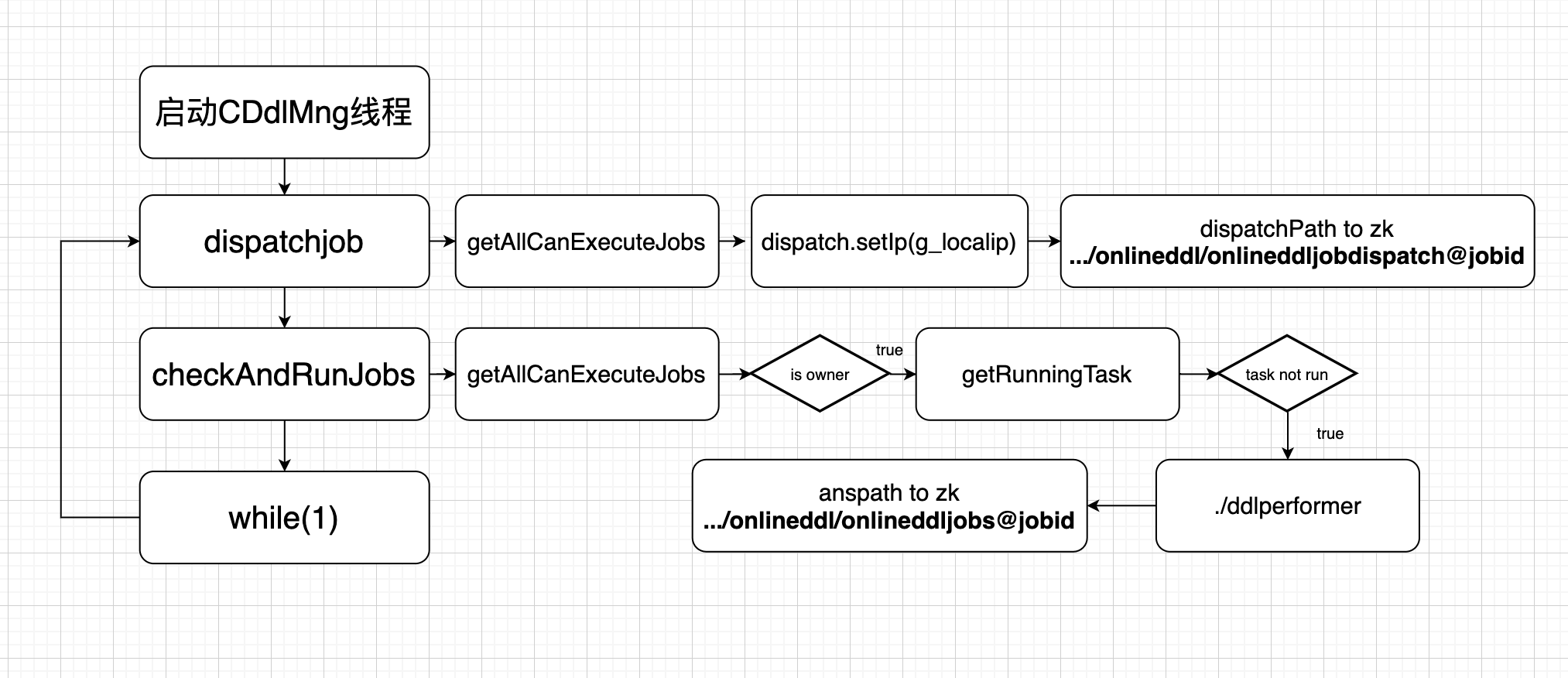
3、通过实现PT-OSC的HOOK接口，在执行的每个阶段将状态信息上报至ZOOKEEPER

4、在执行PT-OSC的过程中，不开启自动删除旧表的选项，便于回滚。

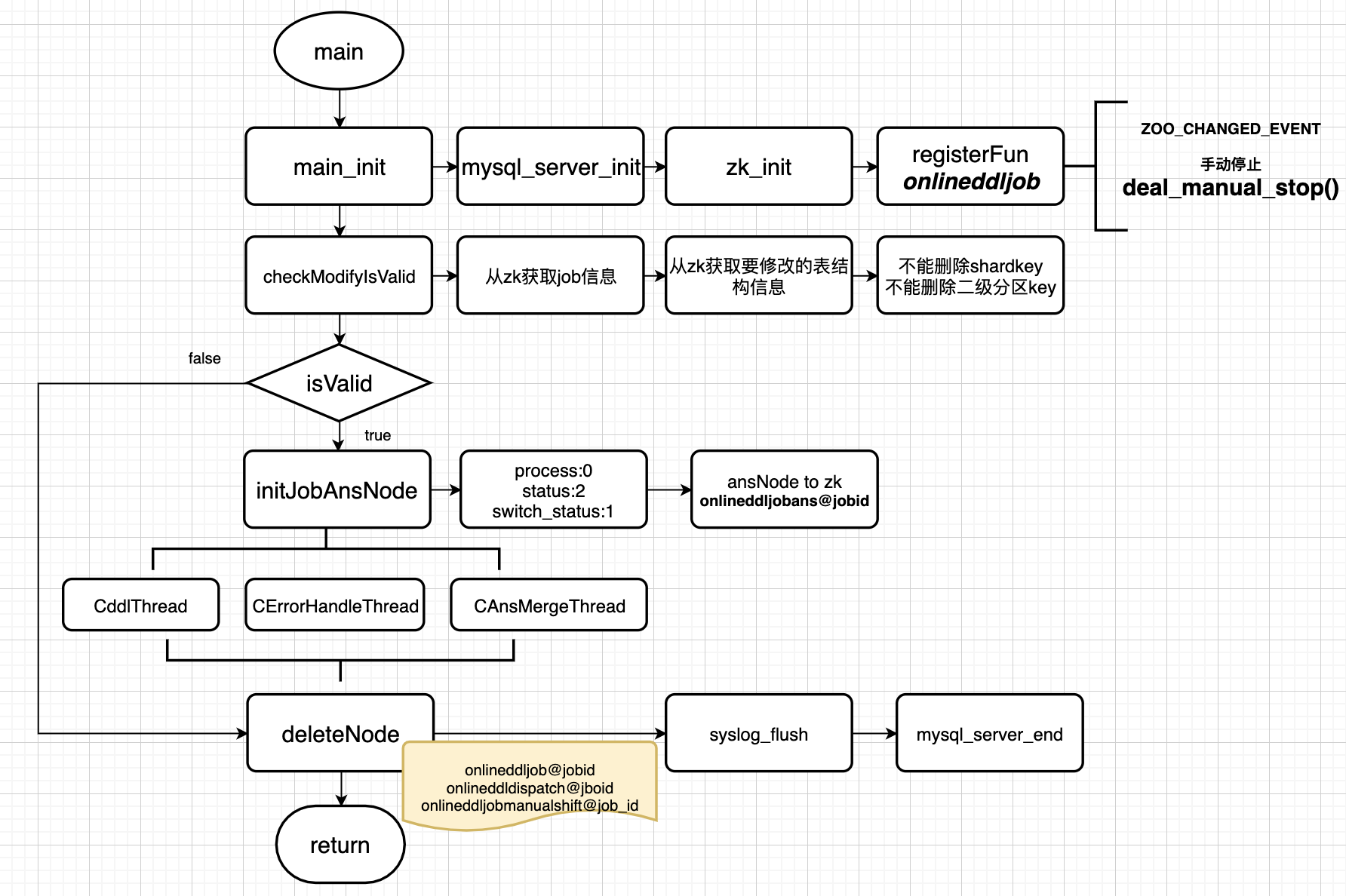
### 实现

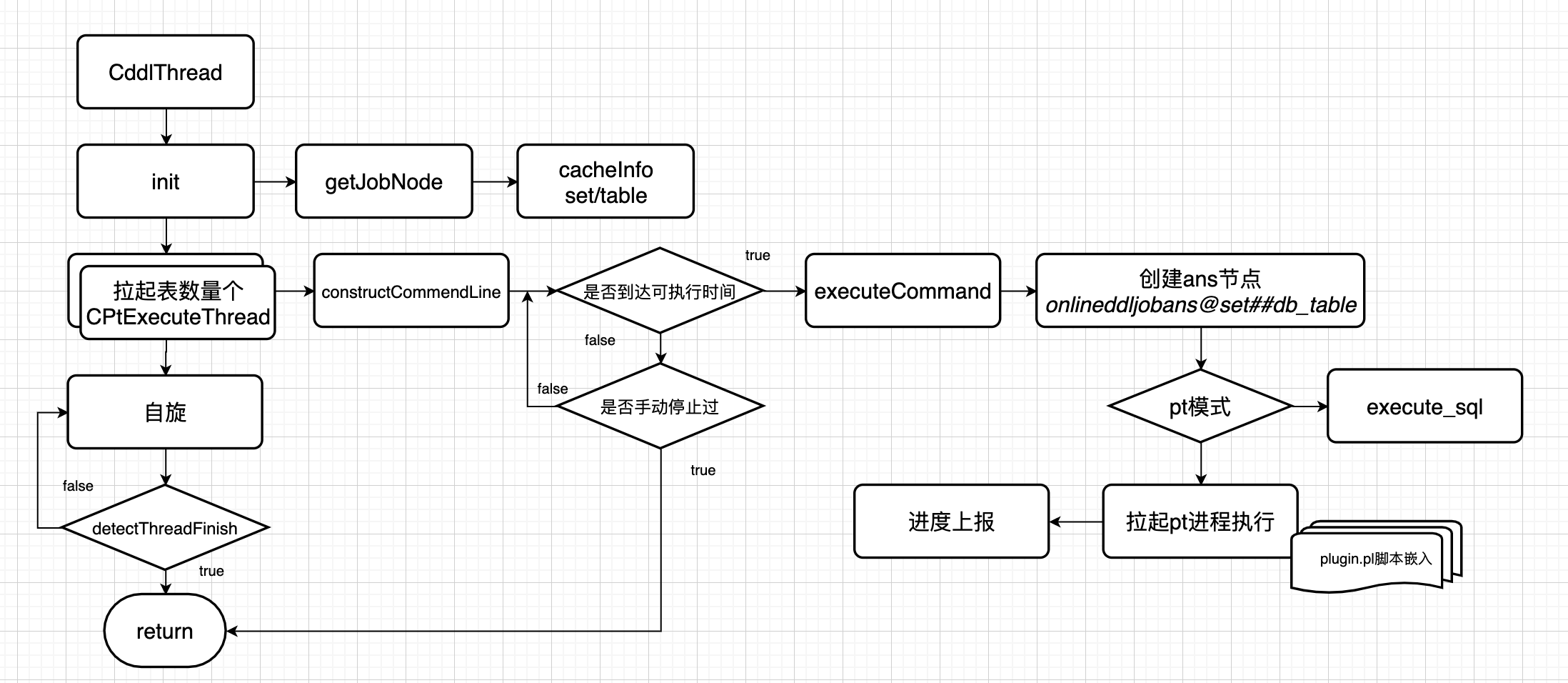
#### ddlperformermng

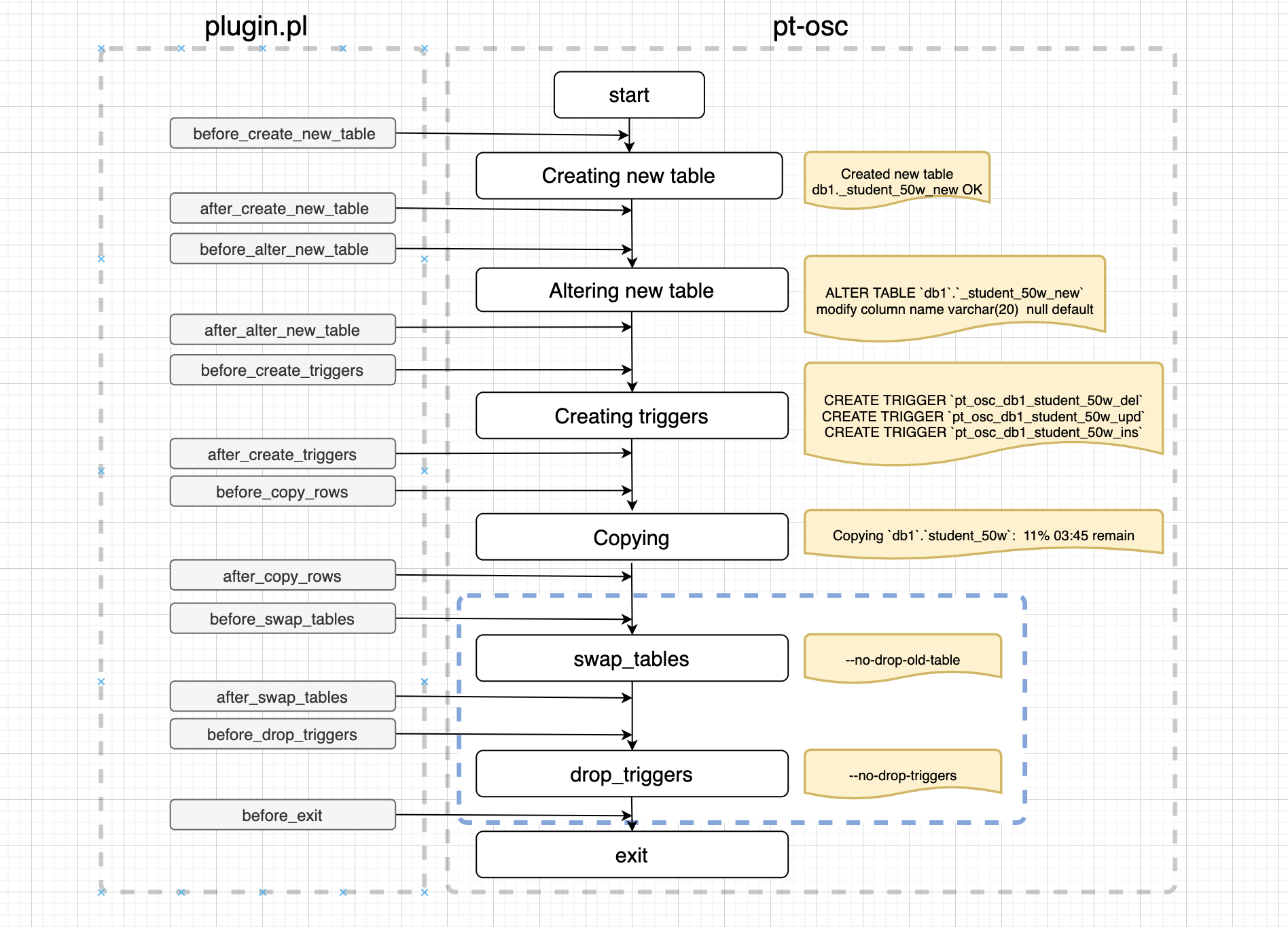


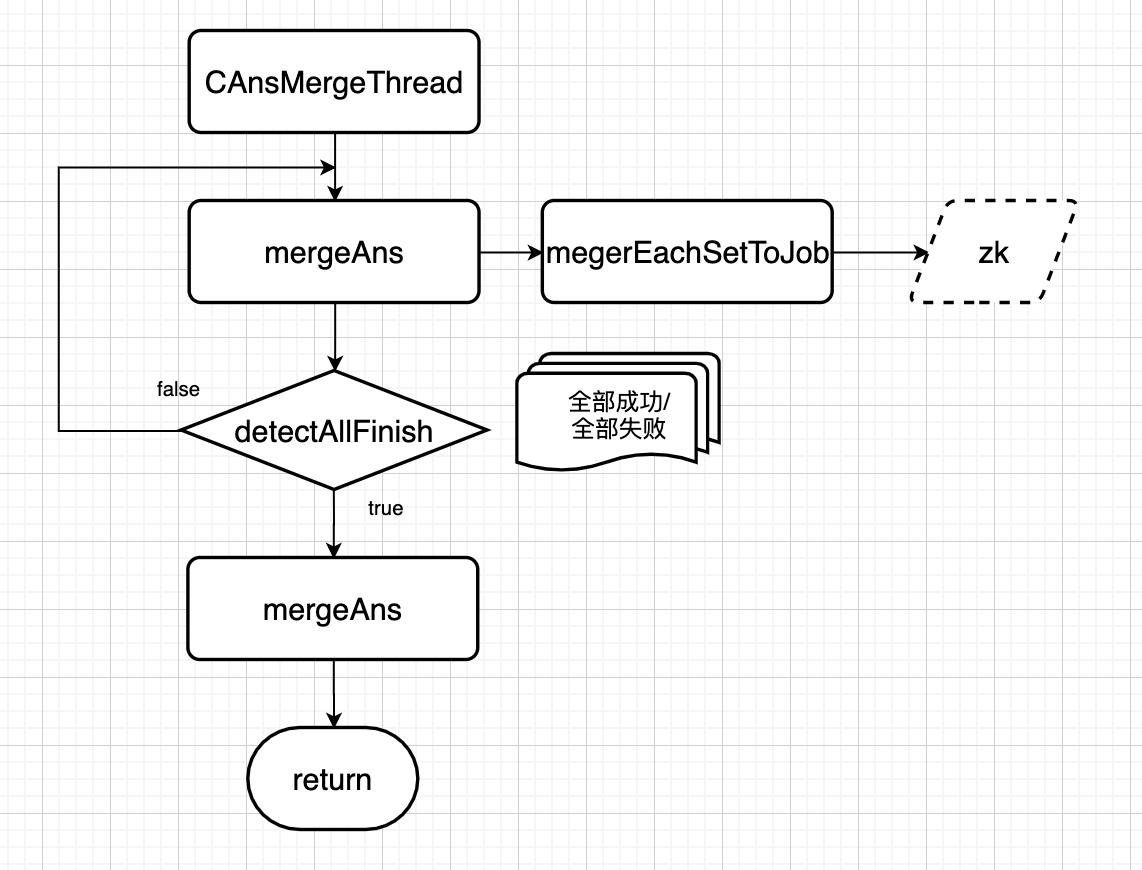


#### ddlperformer











## TDSQL3.0

### 背景

### 原理

Online DDL方案中用到的存储结构：

每个SQL Engine本地有一个schema cache，其中缓存着该SQL Engine节点访问过的表的schema。

在schema cache中，每个表的schema上带有两个时间戳：版本号version和最近确认版本号RVSV(recently verified schema version)，这两个时间戳和事务时间戳是同一套时间戳。

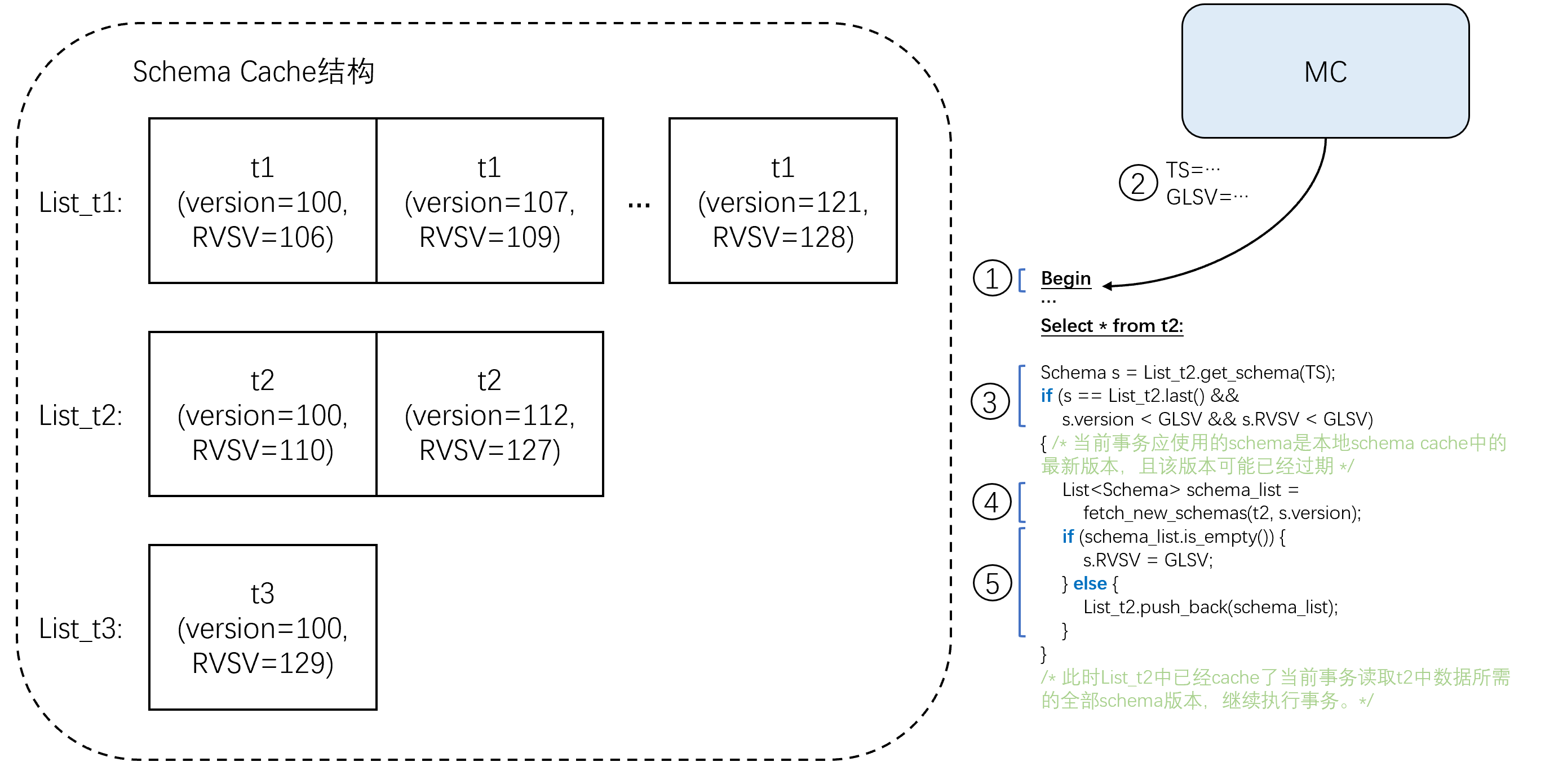
TDStore上以KV的形式存储着各个表的schema，每个表的schema存储为一条KV记录。key是schema\_[table id]\_[version]，value为protobuf序列化后的schema对象，其中包含了database id、table id、table name字段、索引等信息。

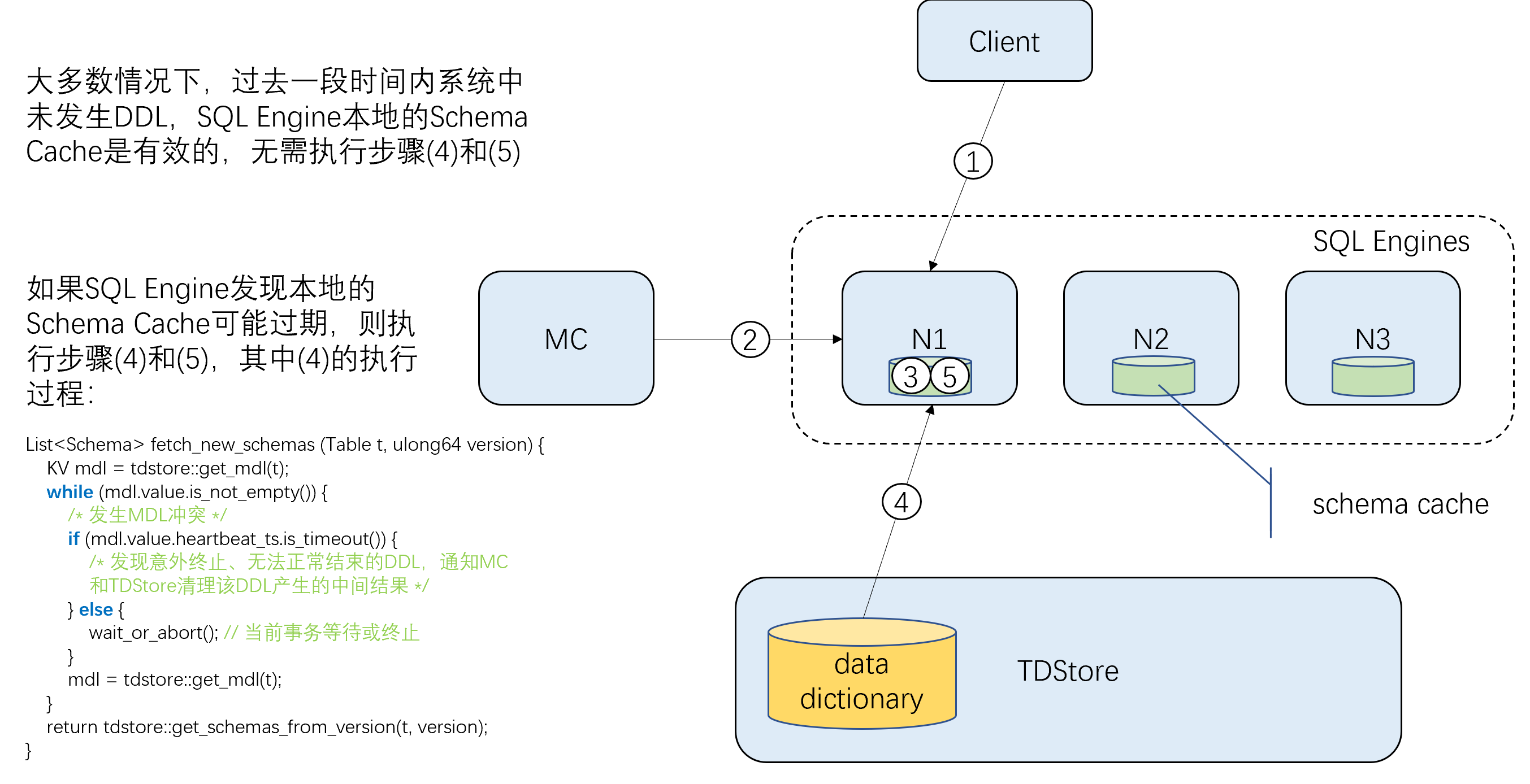
在TDStore上还为每张表存储一条metadata lock (MDL) 记录，key为mdl\_[table id]，value中包含当前持有该MDL的SQL Engine的engine id和心跳时间戳。Rocksdb可以保证对kv记录的原子读写。当一个SQL Engine开始执行DDL时，要获取对应表的MDL，即检查MDL记录的value是否为空，如果为空则将value改为自己的engine id和当前的系统时间（作为心跳时间戳，可以通过ntp或者MC同步，不要求很精确）写入value，加锁成功，否则加锁失败。DDL获取MDL后，会定时更新MDL上的心跳时间戳。

当一个SQL Engine正在执行普通事务、并且需要从TDStore拉取表的schema时，先检查对应MDL记录的value是否为空，为空才能继续拉取schema并继续执行事务，否则检查MDL上的心跳时间戳，如果心跳时间戳已经超时，则通知MC和TDStore清理发生故障的DDL，如果没有超时，则挂起或终止当前的事务。

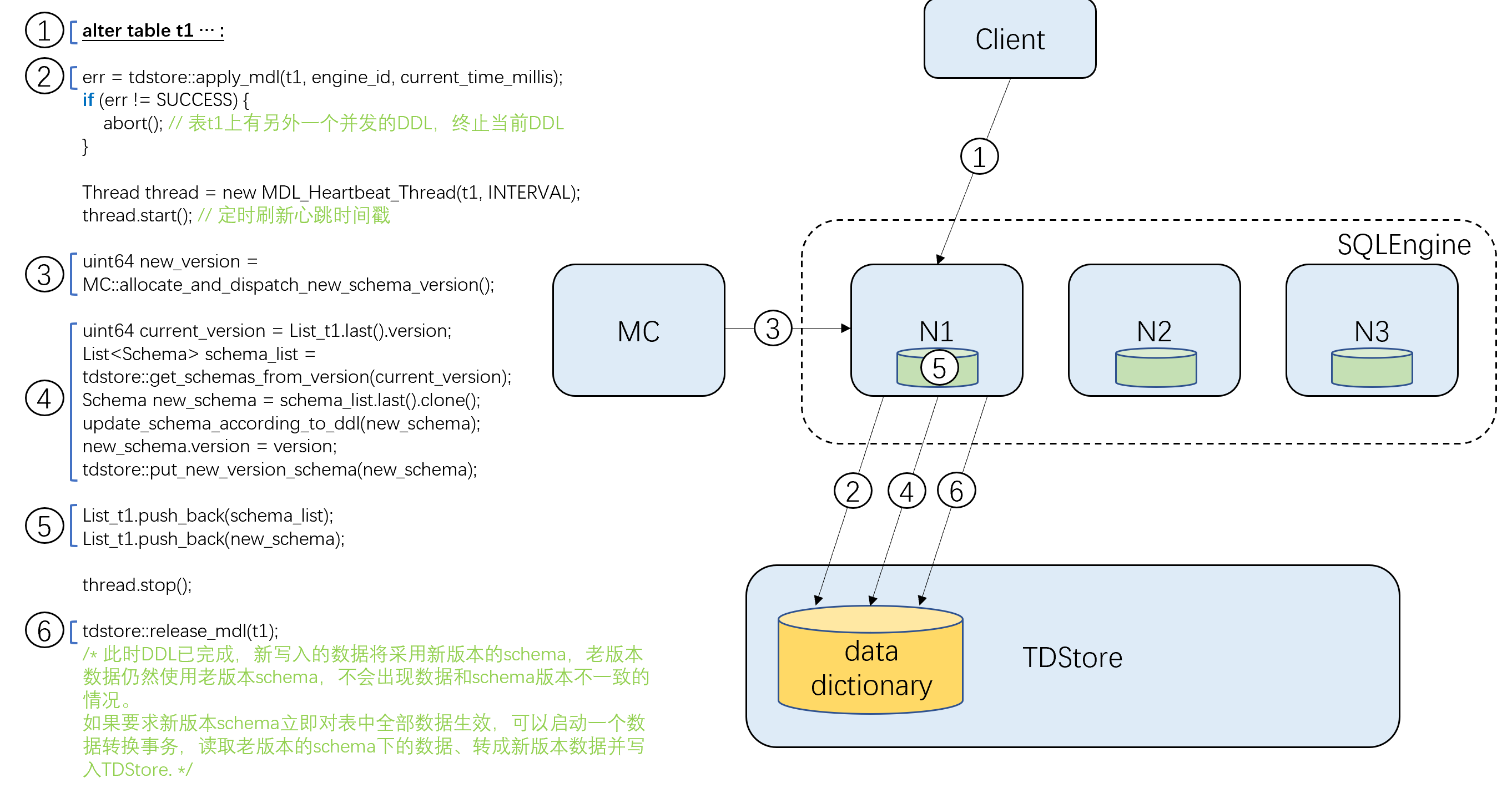
MC上维护一个状态：当前系统所有表上最近一次DDL产生的schema version: GLSV (global latest schema version)，不用记录到底是哪张表的，只起到提示SQL Engine可能需要更新Schema Cache的作用。当一个事务开始时，需要请求MC下发事务的开始时间戳TS，MV在下发TS的消息中顺带上GLSV。

**普通事务中获取schema的过程：**





**DDL执行过程：**



**数据转换：**

当DDL操作产生一个表的新版本Schema之后，如果需要新版本Schema尽快对表中全部数据生效，则需要启动一个数据转换事务，将老版本schema下的数据标记为删除，并转换成新版本数写入表中。

对数据转换事务的粒度和启动时机可以控制，例如选择一个低峰时段进行，或者将数据转换分成多个事务，每个事务只进行一小段数据的转换，避免造成表上后续事务被长时间阻塞。

也可以在range的分裂合并、LSM Tree Compaction过程中顺便做转换，但是会增加存储层的复杂度。

也可以考虑采用F1 Online DDL的思想，将数据转换分为多个中间状态，分步进行数据转换，但是具体实现可能比较复杂。

对用户事务的影响：

DDL不会阻塞其他表上的用户事务，但是会阻塞当前表上的用户事务。

不过由于数据转换可以在DDL之后异步进行，正常情况下，DDL本身执行时间应该是毫秒级的，不会造成当前表上用户事务的严重阻塞。

但是DDL可能会导致大量的SQL Engine在短时间内一起尝试从TDStore拉取新版本Schema，影响性能。但是由于每个SQL Engine只需要拉取一次Schema，影响时间不会太长。之前讨论认为这样可以接受。

此外，DDL之后的数据转换也会影响用户事务，数据转换本质上是批量的update，对用户事务影响较大。可以采用上一页中的方法进行优化。

**Schema历史版本回收：**

数据转换会将老版本schema下的数据转换成新版本schema下的事务，用户事务中的update、delete操作也会将老版本数据清除

当一个表在某个历史schema版本下不再有数据、也不再有活跃的用户事务时，就可以将这个版本的schema回收，以提高查询性能。

回收时，在TDStore上将要回收的schema版本标记为删除。SQL Engine定时查询TDStore上的数据字典，将带有删除标记的schema版本从自己的schema cache中删除。

**DDL回滚和故障恢复：**

**回滚：**

DDL的执行过程中只是修改TDStore上的数据字典，执行过程应该是毫秒级的，并且和正在执行的普通用户事务没有冲突，除了遇到同一个表上的并发DDL之外，不容易发生回滚。

如果遇到同一个表上的并发DDL，此时当前DDL处于第(2)步，其实什么也没做，直接返回失败即可。

如果遇到第(3)步及之后需要回滚的情况，那么回滚时需要通知MC回滚在第(3)步中申请的GLSV，并删除已经写入的新版本schema、释放MDL.

**故障恢复：**

如果执行DDL的节点发生宕机等故障，导致DDL无法正常完成或回滚，那么MDL上的心跳时间戳也会停止更新，其他事务在获取MDL时如果发现MDL上的心跳超时，则可以强制回滚未完成的DDL.

假如执行DDL的节点在一段时间之后又恢复过来、试图继续执行DDL时，如果发现自己的心跳已经超时，则主动停止、回滚DDL。

**DDL并发与死锁：**

不同表上的DDL可以并发，同一张表上的DDL在数据字典上有写写冲突、不能并发。

有些DDL可能会修改多张表的Schema，例如更换主键可能导致其他表上的外键也需要修改、删库需要删除库下所有的表。

那么并发的DDL就可能形成死锁，解决办法和普通事务死锁的解决办法一样，例如在执行DDL时按照table id从小到大的顺序获取MDL。

### 实现

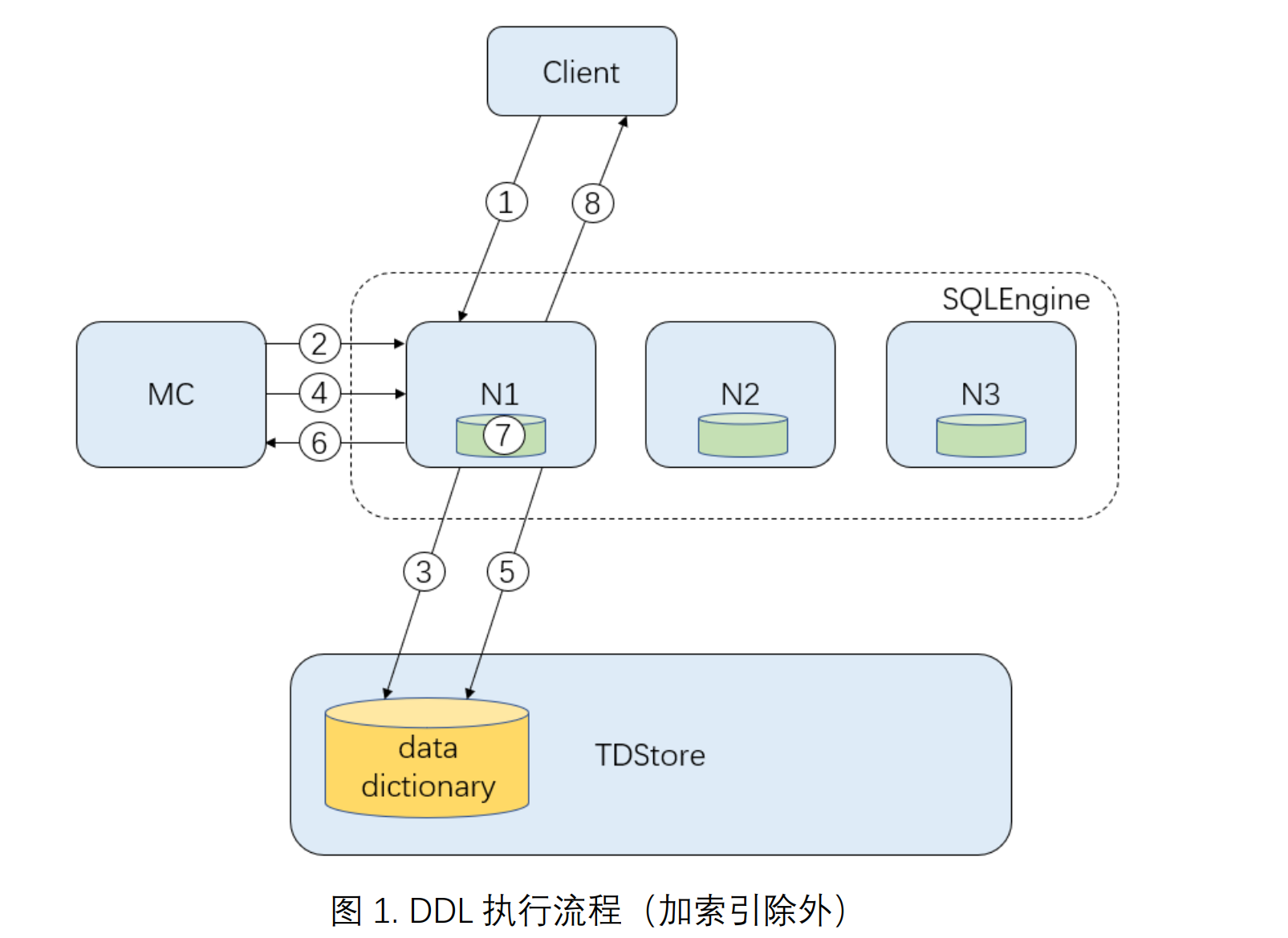
参考F1论文，DDL分为两大类：

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

Logical DDL流程如下：

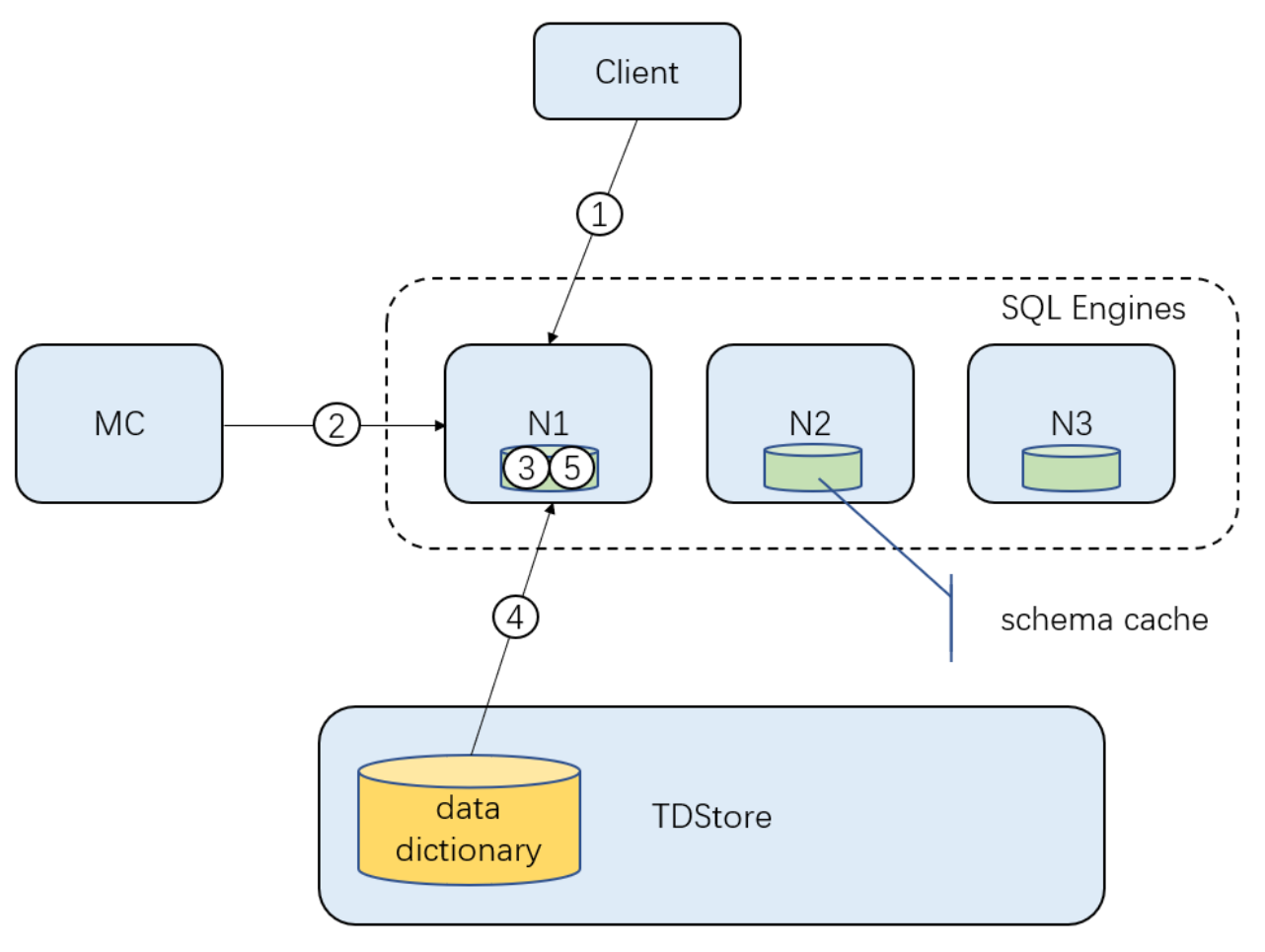


1. Client向SQL Engine发送DDL命令；
2. SQL Engine从MC取得schema version和DDL事务的begin timestamp，MC加锁阻塞之后的DDL；
3. DDL事务进入Prepare阶段，将状态为public的新版本schema写到全局数据字典中；
4. SQL Engine从MC取得DDL事务的commit timestamp；
5. DDL事务commit；
6. SQL Engine通知MC提交是否成功，MC释放锁，允许后续DDL事务开始。如果当前DDL事务提交成功，MC更新GLSV (Global Largest Schema Version)，使得新版本的schema全局可见；
7. 如果DDL事务提交成功，SQL Engine更新本地的数据字典缓存；
8. 通知Client DDL是否执行成功。

如果DDL提交成功，步骤(6)之后新版本的schema就对所有的SQL Engine 可见了，此后SQL Engine上开启的新事务都会看到新版本的schema。

Logical DDL中的数据转换由TDStore在compaction的过程中异步完成。

用户事务更新schema流程：



1. Client 向 SQL Engine 发送命令开始事务；
2. SQL Engine 从 MC 获取事务开始时间戳和当前的 GLSV；
3. SQL Engine 将 GLSV 与本地数据字典缓存中最大的 schema version 比较，如果相等，则本地数据字典缓存无需更新、继续执行事务，否则先执行(4)和(5)，完成本地数据字典缓存的更新之后再继续执行事务；
4. 从 TDStore 的全局数据字典中读取增量的 schema 更新信息；
5. 更新本地数据字典缓存。

用户事务中的读操作只能使用状态为public的最新版本的schema，写操作(delete update insert)可以使用write only或public状态下的最新版本schema。

用户事务put kv时，将所使用的schema version带在value上。

用户事务get kv时，在请求中带上所使用的schema version，TDStore将value转换成该version对应的格式返回给SQL Engine。

用户事务commit时还会再获取GLSV，如果事务中发生了写操作，则需要对比commit时的GLSV和begin时的GLSV是否相等，如果不相等，则执行上述的(4)和(5)更新本地数据字典缓存，如果事务中写操作涉及的表上产生了新版本schema，则回滚或重试当前事务。

#### Physical DDL

Physical DDL需要分为 3 步：

1. 创建状态为write only的新版本schema，用户事务中的update、delete、insert操作使用该新版本schema，读操作使用前一版本的schema；
2. 执行数据转换，对于加索引就是backfill索引记录，对于修改主键就是修改key；
3. 创建状态为public的新版本schema，用户事务中的所有操作都可以使用 public版本的schema。

其中第1和3步使用和Logical DDL一样的流程，只是第1步不给用户返回结果，在第3步执行完才返回结果。第2步是一个后台任务，参考F1/TiDB：

假设第1步的提交时间戳为t1，将表中的数据分成很多小的partition（划分方法参考TiDB），启动一个后台 job，多线程处理各个partition（以后可以做成分布式的）。 对于每个partition，用t1拿到一个snapshot，读取其中的静态数据，计算出转换后的结果、根据Thomas write rule写回TDStore中。由于kv的put具有幂等性，如果某个partition转换失败，只要重试即可，不需要额外的恢复。

## GoldenDB

### DDL流程设计

#### 离线DDL

OMM界面执行的DDL属于离线DDL，均不锁表，是因为运维人员能保证当前没有业务在执行。

#### Proxy DDL

##### 流程

Proxy发起的DDL处理流程：

Proxy->PM->MDS修改表状态->PM->通知所有Proxy禁表->Proxy向DB发起DDL操作->PM->MDS修改DDL并解禁->PM->将新的元数据推送到所有Proxy

##### 分类

Proxy的DDL分为在线DDL和非在线DDL。

###### 非在线DDL

对于一般的DDL，proxy会对表做禁表操作，在DDL完成之前，表一直处于禁表状态，不允许对表做任何操作，包括增删改查，返回客户端的错误为：ERROR 11204(HY000)：table ‘db.tb1’ si disabed!

###### 在线DDL

对于在线DDL，proxy不对表做禁表操作，允许并行DML。需要强调的是，proxy支持的在线DDL和mysql支持的在线DDL不一致，proxy只支持以下几种在线DDL：

|  |  |
| --- | --- |
| 类型 | 操作 |
| INDEX | add index |
| drop index |
| COLUMN | alter table add column |
| PARTITION | add partition |
| drop partition |
| truncate partition |
| reorganize partition |
| coalesce partition |
| remove partition |

Proxy不支持alter table t add column f1 int,add column t2语法。

###### 多分片串行/并行方式

修改proxy.ini文件ddl\_execute\_serial，指定DDL的执行顺序：

0：并行执行

1：串行执行

##### 在线DDL

在proxy上执行的DDL属于在线DDL，有些场景锁表有些场景不锁表：

###### 不锁表

1. 如下客户端下发DDL不锁表

alter table add column(NOT NULL default) --新增列必须在最后，不包含FIRST AFTER等指定位置的DDL

alter table add column 允许NULL --新增列必须在最后，不包含FIRST AFTER等指定列位置的DDL语句

alter table add index+drop index --不锁表

alter table add/drop partition --不锁表

alter table modify/change column类型兼容，详细如下：

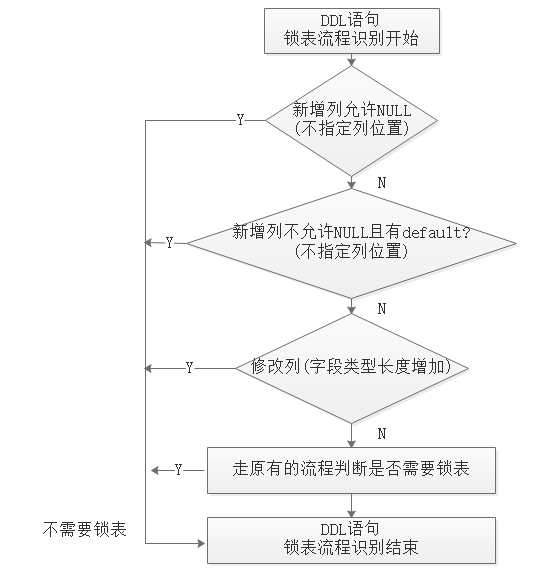
1. 扩长度char、varcha、decimals
2. int类型转换（tinyint->SMALLINT->int->BIGINT）
3. float转换为double，int转换为float double
4. 不停机DDL回滚问题

目前分析以上几种不禁表方案不影响回滚，所以不需要考虑回滚问题

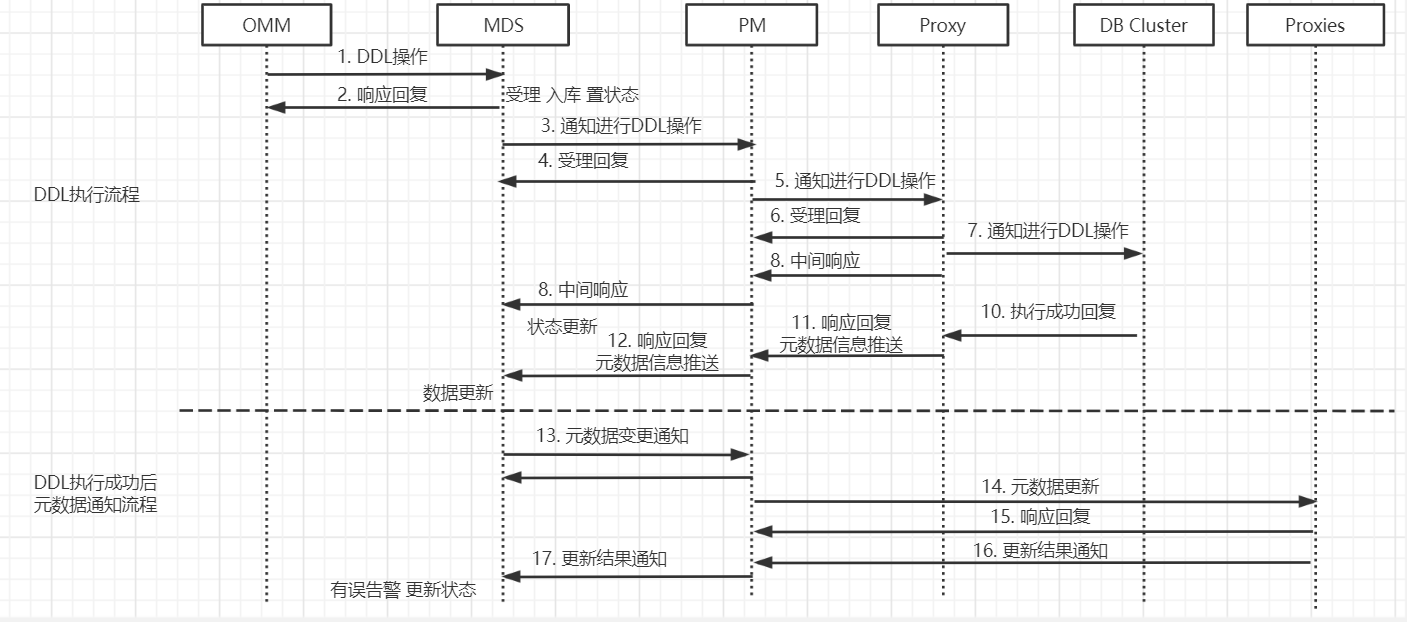
1. 不停机DDL执行失败不需要禁表

不停机DDL执行失败不需要禁表——需要告警

不锁表流程如下：



不锁表流程图：



###### 锁表

锁表流程图：



### Proxy和DB差异

|  |  |  |
| --- | --- | --- |
| 场景 | Proxy | DB |
| 新增字段 | 在线DDL，不禁表 | 在线DDL |
| 指定新增字段位置 | 非在线DDL，禁表 | 在线DDL |
| 修改字段注释 | 非在线DDL，禁表 | 在线DDL |
| 增加重复列 | 不报错 | 报错 |
| 批量修改语法 | 不支持 | 支持 |
| 新增字段时，部分分片成功，部分分片失败 | 在线DDL，不禁表  结果：proxy报错，没有禁表，仍然允许业务使用该表 | 分片1原表结构  分片2新表结构 |

### 使用建议

#### 批量处理

通过合并多个DDL，达到性能最大化。例如，如果新增多个字段，可以通过一条DDL完成，这样可以达到只复制一次ibd文件，减少IO消耗和执行时间。

alter table t add column t1,add column t2;

#### 定制DDL的执行方式

语法：alter table xxx,ALGORITHM=xx,Lock=xx;

ALGORITHM与LOCK参数描述：

|  |  |  |
| --- | --- | --- |
| 参数名 | 取值 | 作用 |
| ALGORITHM | COPY | 允许读操作，不允许写操作 |
| INPLACE | 允许读写操作 |
| DEFAULT | 默认值，根据DDL的类型，优选选择INPLACE |
| LOCK | EXCLUSIVE | 对整个表加排它锁，不允许DML并行 |
| SHARED | 对整个表加共享锁，允许读取，不允许写 |
| NONE | 尽可能少加锁，允许读写 |
| DEFAULT | 默认值，根据DDL的类型，尽可能的允许DML |

##### ALGORITH

ALGORITH参数解析：

1. ALGORITHM=COPY相当于offline DDL，ALGORITHM=INPLACE相当于online DDL
2. 支持INPLACE方式的DDL，也支持COPY方式；支持COPY方式的DDL，不一定支持INPLACE方式

例如：

1. 添加字段DDL，支持INPLACE同时支持COPY方式，默认是INPLACE方式，可以强制指定ALGORITHM=COPY，最终是通过offline DDL的方式执行的：

alter table t1 modify column f20 varchar(10),ALGORITHM=COPY;

1. 修改字段数据类型DDL，只支持COPY方式，不支持INPLACE方式，默认是COPY方式，如果强制指定ALGORITHM=INPLACE，报错：

alter table t1 modify column f14 varchar(10),ALGORITHM=INPLACE;

ERROR 1864:ALGORITHM=INPLACE is not supported.Reason:cannot change column type INPLACE,Try ALGORITHM=COPY

##### LOCK

LOCk参数解析：

1. LOCK=SHARED相当于Offline DDL级别的锁，LOCK=NONE相当于online DDL级别的锁
2. LOCK参数不能指定为低于对应DDL级别的锁（锁级别EXCLUSIVE > SHARED > NONE）

例如，修改字段的数据类型属于offline DDL，需要加SHARED锁，不允许写操作，如果LOCK指定为NONE就会报错：

alter table t1 modify column f14 int,lock=none;

ERROR 1864:LOCK=NONE is not supported.Reason:cannot change column type INPLACE,Try LOCK=SHARED

1. LOCK参数可以指定为高于对应DDL级别的锁

例如，添加字段属于online DDL，允许读写操作，如果LOCK指定为EXCLUSIVE，则不允许读写操作

alter table t1 add column f20 varcahr(10),lock=exclusive;

1. 建议不要显式制定LOCK，由DDL自行判断加合适的锁

#### 采用最小代价方式

1. 从对业务的影响、对资源消耗以及执行时间等角度考虑，不同类型的DDL代价差别比较大

Online DDL的no-rebuild方式代价最小，不仅允许并行DML，而且不消耗IO资源，执行通常很快；

Online DDL rebuild方式的DDL代价中等，允许并行DML，但是对空间和IO资源要求高；

Offline DDL，不仅影响业务，同时对空间和IO资源要求高，性能相对Online DDL rebuild低。

Online DDL no-rebuild < Online DDL rebuild < Offline DDL

特别是在多种DDL组合的需求时，计算最小代价的方式。

1. 对于online DDL rebuild方式，并行的DML对DDL执行时间和IO影响，DDL尽量在业务量小的时候进行

#### Proxy和DB分开执行DDL

通过在proxy和DB上分别执行DDL，来达到proxy上不支持的DDL语法，以及达到最佳性能。

样例场景：

Proxy不支持“alter table ta add column f1 int,add column t2,add column t3;”语法，如果分成3个DDL执行，代价将是上述语法的3倍。

最佳实践：

1. 直连每个分片主DB执行“alter table t add column f1 int,add column t2,add column t3;”，使DB上新增3个字段
2. 待第一步完成后，再通过proxy执行3个拆分后的DDL，是元数据和DB上的表结构保持一致

#### 新增字段

|  |  |
| --- | --- |
| 方案 | 从proxy发起，每次新增一个字段；增加多个字段时，分多个时间段，每个时间段只增加一个字段 |
| 语句 | alter table t\_name add column c\_name data\_type; |
| 是否允许在线读写操作 | 是 |
| 磁盘IO影响 | 产生大量IO，对数据库性能有影响 |
| 影响DDL执行时长因素 | 表数据量，DDL过程中的在线写操作数量 |
| 执行限制 | 1. 不允许在later语句中加first、after关键字； 2. 选择业务量少的时间段执行 |

#### 修改字段类型

|  |  |
| --- | --- |
| 方案 | 从proxy发起 |
| 语句 | alter table t\_name modify column c\_name newData\_type; |
| 是否允许在线读写操作 | 否 |
| 磁盘IO影响 | 产生大量IO，对数据库性能有影响 |
| 影响DDL执行时长因素 | 表数据量，DDL过程中的在线写操作数量 |
| 执行限制 | 1. 确认执行DDL期间没有在线业务对表进行读写，否则涉及在线业务会失败；   2、选择业务量少的时间段执行 |

#### 修改字段comment

|  |  |
| --- | --- |
| 方案 | 从proxy发起 |
| 语句 | alter table t\_name modify column c\_name data\_type comment ‘xxx’; |
| 是否允许在线读写操作 | 否，时间极短 |
| 磁盘IO影响 | 无 |
| 影响DDL执行时长因素 | 无 |
| 执行限制 | 1. 确认执行DDL期间没有在线业务对该表进行读写，否则涉及在线业务会失败； 2. 选择业务量少的时间段执行 |

#### 新增索引

|  |  |
| --- | --- |
| 方案 | 从proxy发起 |
| 语句 | alter table t\_name add index idx\_name(c\_name1,...); |
| 是否允许在线读写操作 | 是 |
| 磁盘IO影响 | 产生大量IO，对数据库性能有影响 |
| 影响DDL执行时长因素 | 表数据量 |
| 执行限制 | 选择业务量少的时间段执行 |

#### 新增分区

|  |  |
| --- | --- |
| 方案 | 从proxy发起 |
| 语句 | alter table t\_name add partition(partition p\_name values less than(xxx)); |
| 是否允许在线读写操作 | 是 |
| 磁盘IO影响 | 无 |
| 影响DDL执行时长因素 | 无 |
| 执行限制 | 选择业务量少的时间段执行 |