# Online-DDL

## 概述

Online DDL的主要目标是实现DDL操作不阻塞用户事务的执行，并尽可能降低对用户事务性能的影响。此外，对于TDSQL 3.0，也需要考虑在分布式环境下DDL本身的执行效率和故障恢复。

TDSQL 3.0采用了主流NewSQL系统架构，为实现计算层的灵活性和高可扩展，SQLEngine实例是无状态的。在无状态的SQLEngine节点之间没有强同步协议，无法保证所有SQLEngine节点在任何时刻都有完全一致的schema信息。而保证数据强一致性对于TDSQL这类事务型数据库来说是最起码的要求，因此数据一致性和schema弱同步对于TDSQL 3.0来说是两个硬性的要求，在TDSQL 3.0 DDL方案的设计中必须兼顾这两方面。

为了在schema弱同步的前提下保证数据一致性，我们采用了多版本schema的思想，在同一时刻允许不同SQLEngine上有不同版本的schema，但保证以下两点：

1、SQLEngine之间schema差异是收敛的，即当系统中出现新版本schema之后，任何一个SQLEngine都保证在未来一段有限长的时间内或者某个特定的时间节点同步到最新的schema，否则该SQLEngine不再是合法的实例、不能执行任何事务；

2、允许合法的SQLEngine使用不同版本的schema，并且都可以正常执行事务、不会破坏数据一致性。

此外，在DDL的执行方面，为了保证DDL不阻塞用户事务以及DDL本身的故障恢复和执行效率，我们参考了Google F1 Schema Change论文，对于需要做即时数据转换的DDL（如加索引），引入一个Write Only的中间版本schema，并采用Thomas Write Rule来写入数据。

## 现有解决方案

### 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

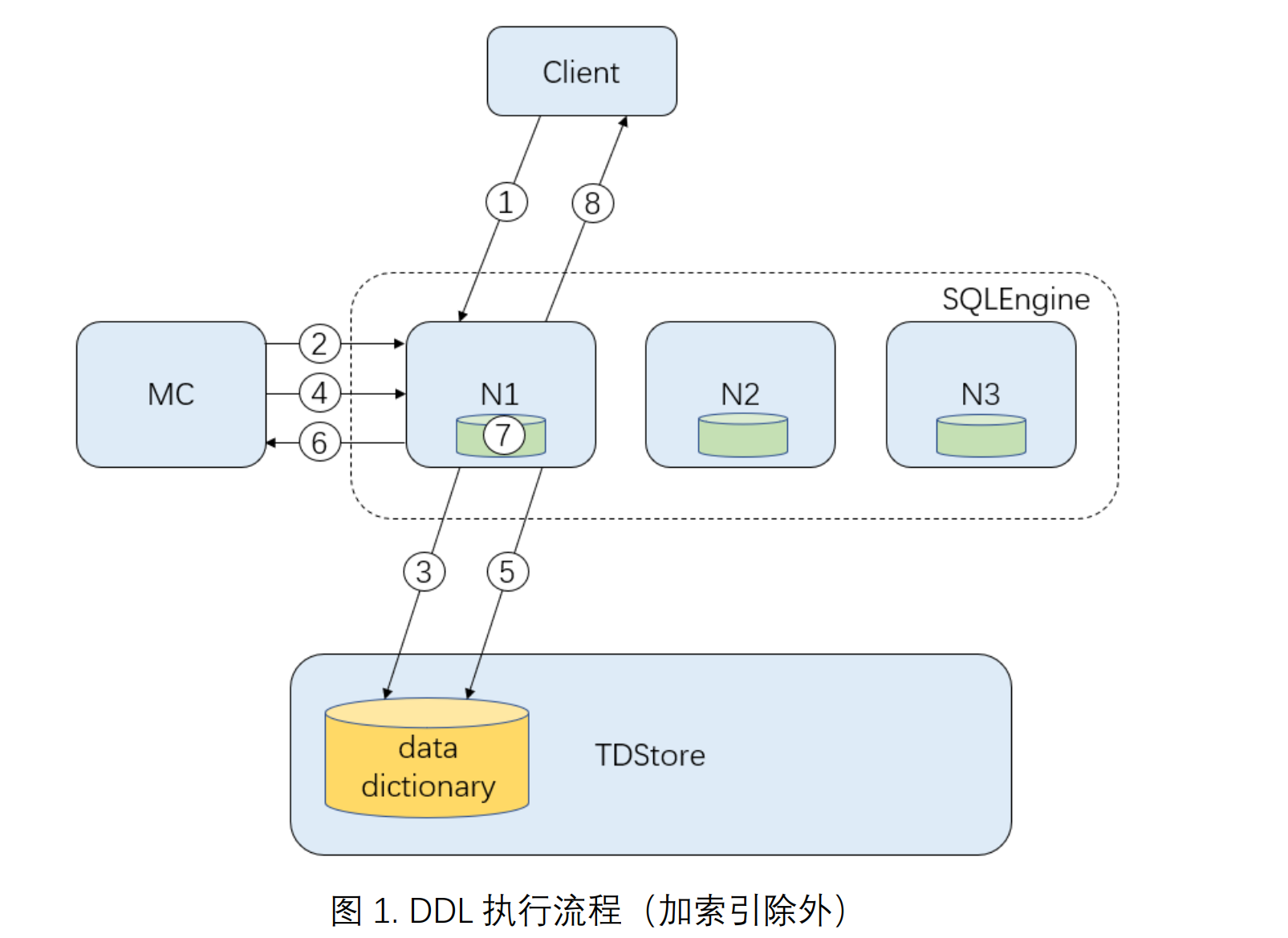
参考F1论文，DDL分为两大类：

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

Logical DDL流程如下：

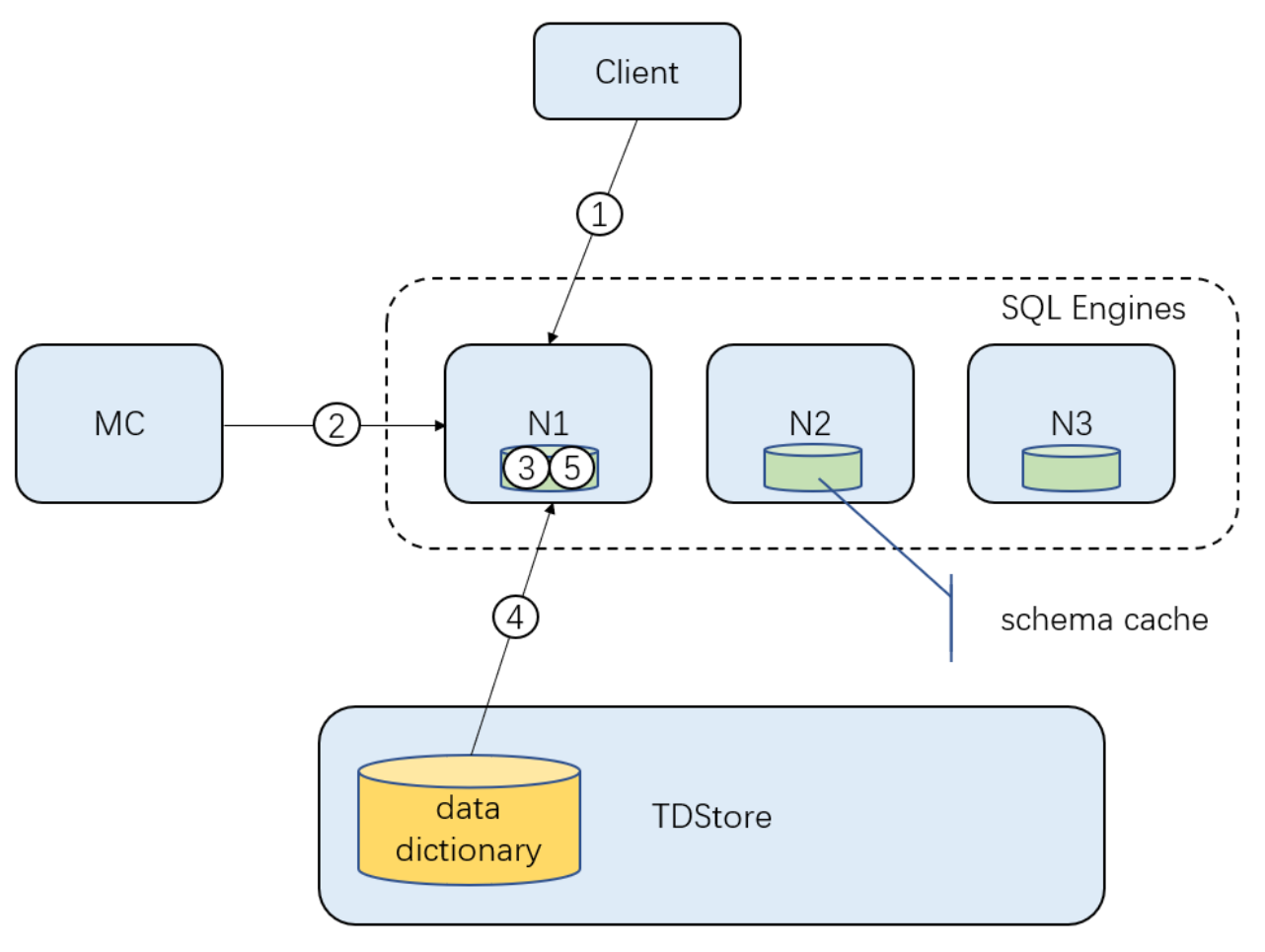


1. Client向SQL Engine发送DDL命令；
2. SQL Engine从MC取得schema version和DDL事务的begin timestamp，MC加锁阻塞之后的DDL；
3. DDL事务进入Prepare阶段，将状态为public的新版本schema写到全局数据字典中；
4. SQL Engine从MC取得DDL事务的commit timestamp；
5. DDL事务commit；
6. SQL Engine通知MC提交是否成功，MC释放锁，允许后续DDL事务开始。如果当前DDL事务提交成功，MC更新GLSV (Global Largest Schema Version)，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

## 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。

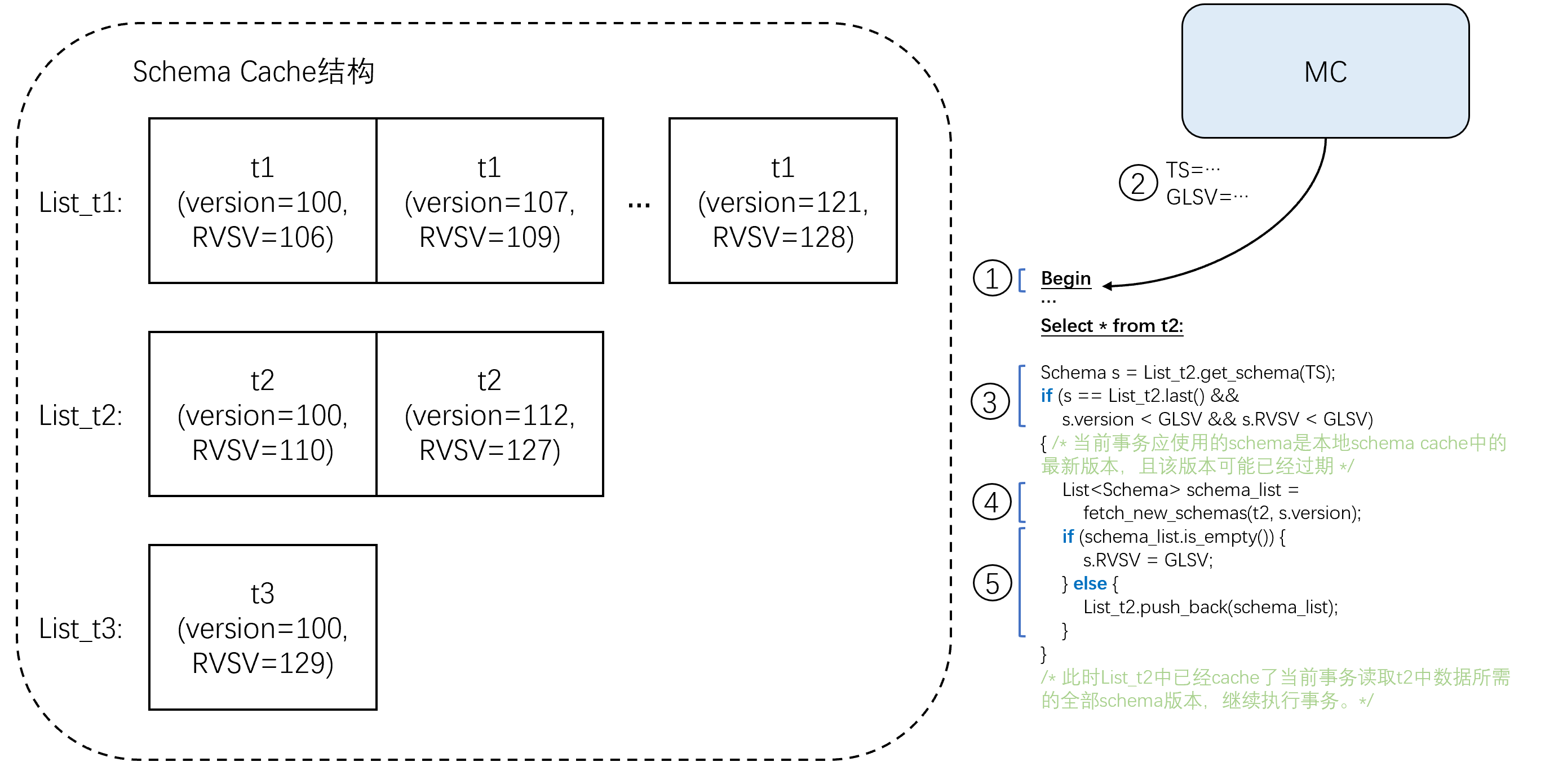
在TDStore上还为每张表存储一条metadata lock (MDL) 记录，key为mdl\_[table id]，value中包含当前持有该MDL的SQL Engine的engine id和心跳时间戳。Rocksdb可以保证对kv记录的原子读写。当一个SQL Engine开始执行DDL时，要获取对应表的MDL，即检查MDL记录的value是否为空，如果为空则将value改为自己的engine id和当前的系统时间（作为心跳时间戳，可以通过ntp或者MC同步，不要求很精确）写入value，加锁成功，否则加锁失败。DDL获取MDL后，会定时更新MDL上的心跳时间戳。

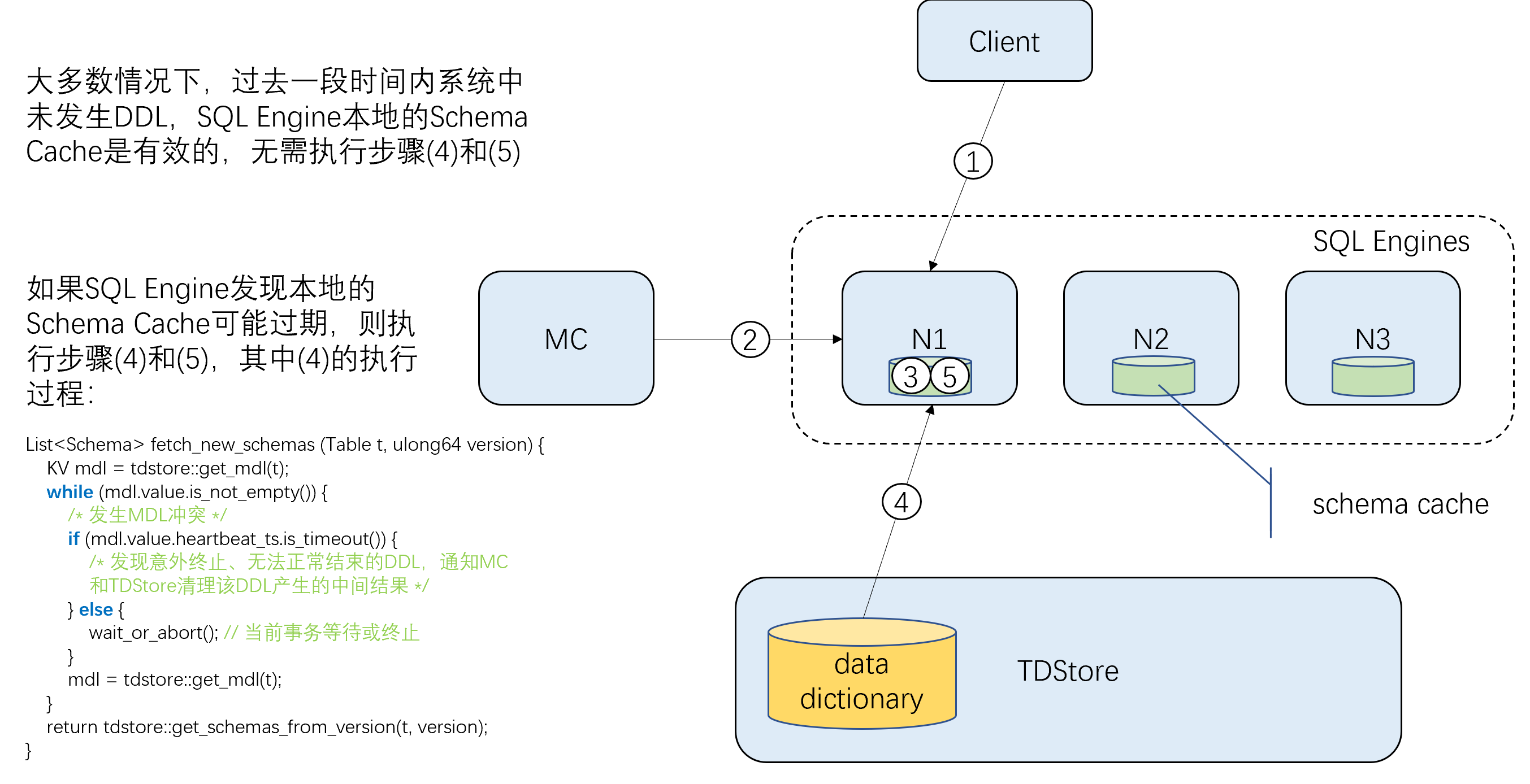
当一个SQL Engine正在执行普通事务、并且需要从TDStore拉取表的schema时，先检查对应MDL记录的value是否为空，为空才能继续拉取schema并继续执行事务，否则检查MDL上的心跳时间戳，如果心跳时间戳已经超时，则通知MC和TDStore清理发生故障的DDL，如果没有超时，则挂起或终止当前的事务。

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，MC在下发TS的消息中顺带上GLSV。

### 普通事务中获取Schema过程

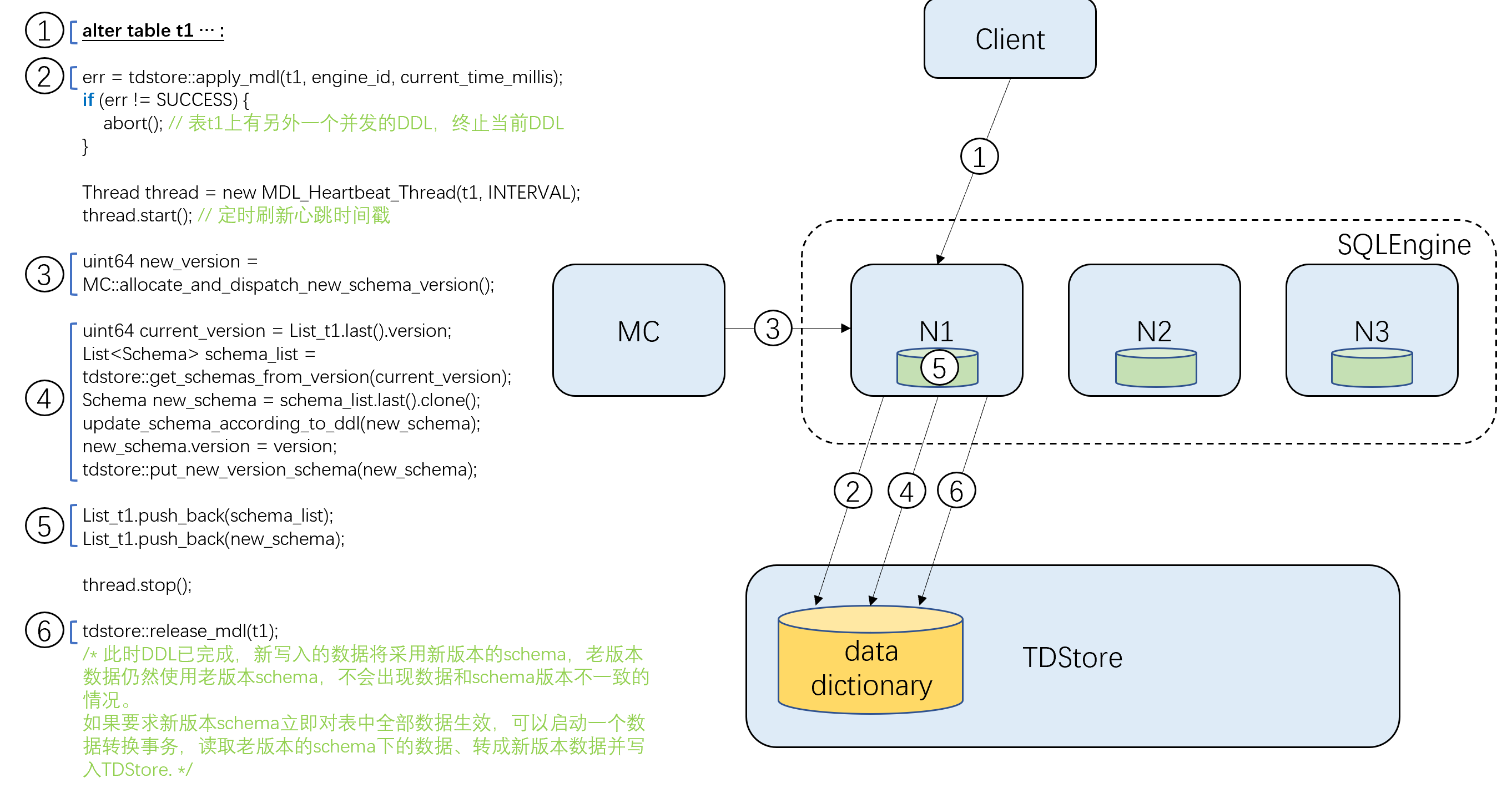
**普通事务中获取schema的过程：**





### DDL执行过程

**DDL执行过程：**



#### 数据转换

**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

**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。这样是可以接受。

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。

#### Schema历史版本回收

**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下的数据，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

**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

**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。

### DDL分段执行

DDL分段执行的目的在于尽可能地保证事务并发，不受DDL语句的强干扰。

#### 基本流程

最常用的DDL分为CRAETE/DROP/ALTER三类，最常用的DDL对象为database和table。所以组合一下有六类，其中最复杂的又属ALTER TABLE。

术语说明：

ddl\_jobs：一张用来记录ddl执行流程的系统表，每个ddl流程都会被记录，但并非每条记录都会被正常移除

恢复线程：后台持续扫描ddl\_jobs表并根据表中的信息进行故障恢复或接替执行的线程

##### 通用流程

###### start\_ddl\_job

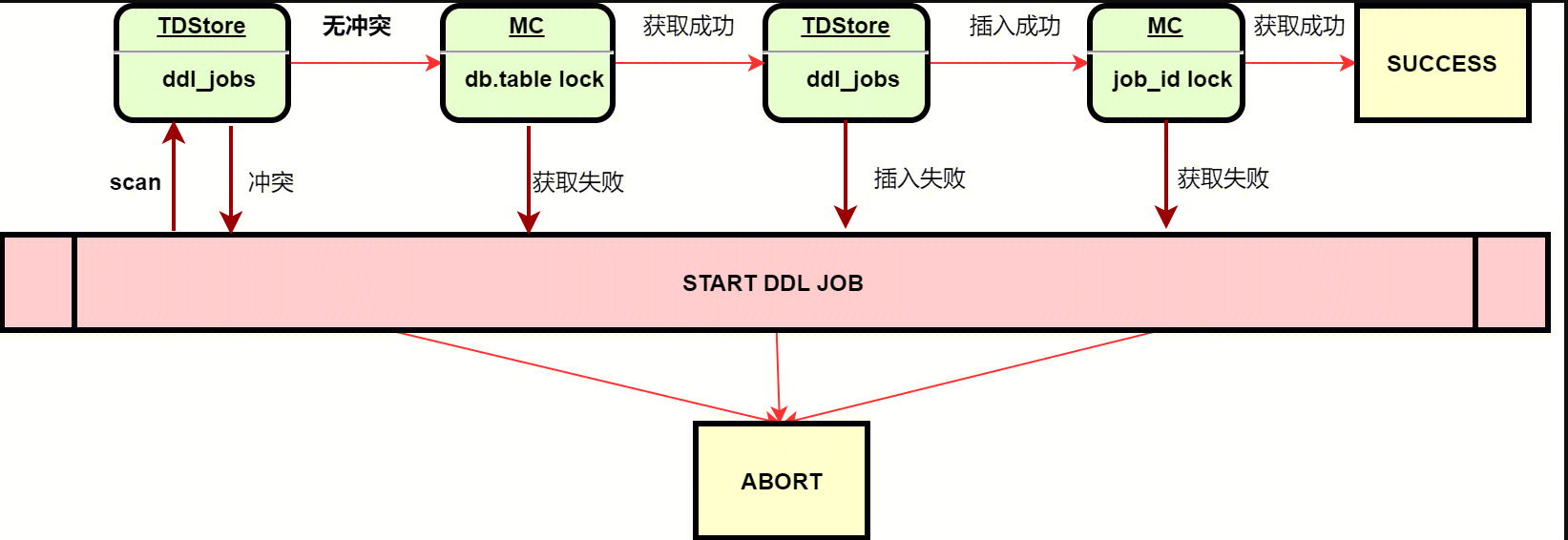
start\_ddl\_job是SQLEngine执行一个DDL语句的开端，其主要功能包括：

1、扫描ddl\_jobs表，确定集群全局没有在执行与当前ddl冲突的语句。如ALTER TABLE和DROP TABLE，这里是利用存储层来做第一次冲突检测

2、获取库表名形成的key，并在MC进行注册，这里是利用管控层来做的第二次冲突检测

3、将本次ddl的信息存入ddl\_jobs表中，以进行故障恢复

4、检查3中存入记录的存在性（与恢复线程行为一致）并获取job-id锁



###### push\_ddl\_job

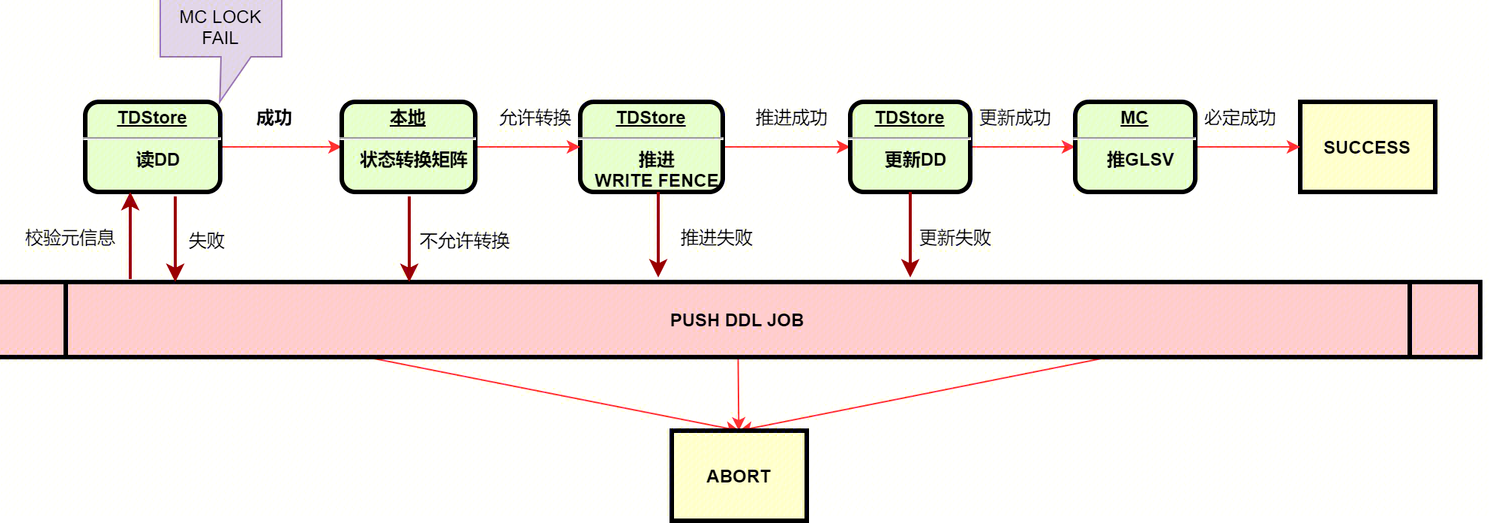
push\_ddl\_job用来推进DDL执行状态，是分段DDL执行的实施者，其主要功能包括：

1、基于最新的数据字典信息做出版本信息的修改，修改object的状态和版本

2、检测object状态转换是否符合逻辑（见状态转换矩阵）

3、添加/推进write fence（删除操作不需要额外修改数据字典信息）

4、保存推进信息，以备故障恢复



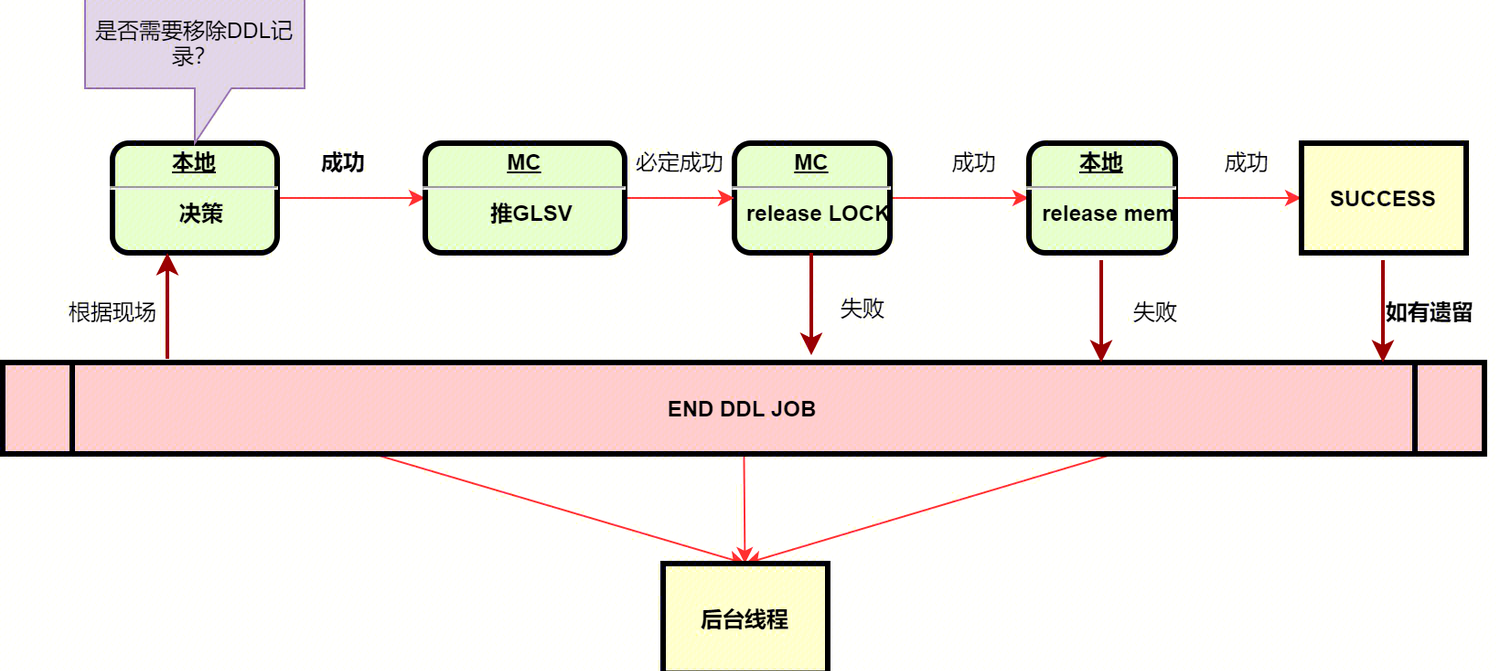
###### end\_ddl\_job

end\_ddl\_job用来结束DDL执行，进行一些善后工作：

1、判断当前DDL信息是否需要保存在ddl\_jobs表中，是否需要保存的依据是是否需要恢复线程来恢复或者继续执行

2、推进global schema version

3、释放本DDL执行流程中申请的各种全局锁，并释放相应的内存

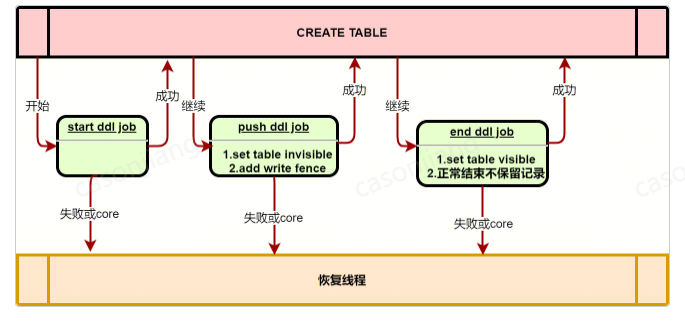


##### CREATE TABLE

CREATE TABLE是比较简单的DDL场景，但是其本身也有多种语法规则（CREATE/CREATE...LIKE.../CREATE...SELECT...）分别来自三个执行分支，

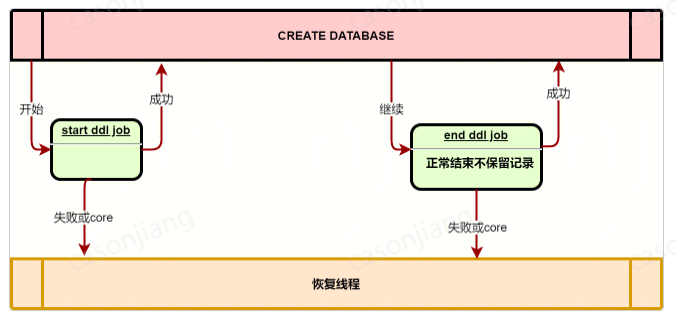
最终汇合于mysql\_create\_table\_no\_lock，统一地将create的起始位置定在这个函数中。

其执行流程为：



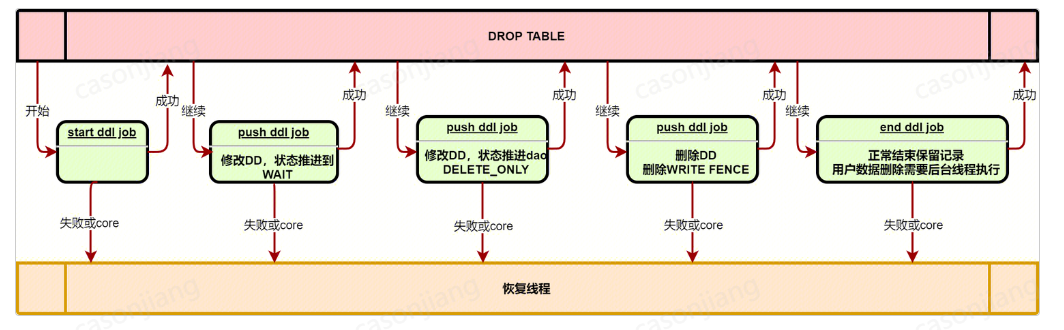
##### CREATE DB

CREATE DATABASE的逻辑只涉及元数据的增加，不涉及write fence等数据场景。



##### DROP TABLE

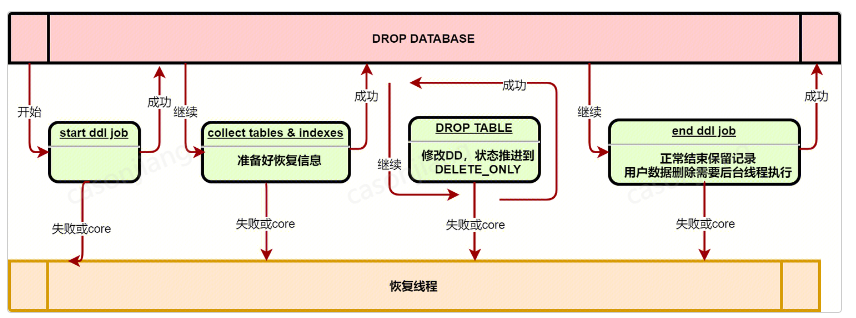
DROP TABLE的逻辑涉及元数据的更新/删除，用户数据的删除以及WRITE FENCE的删除。需要考虑DDL进行过程中并发的DML事务的处理。



一旦push ddl job进入到了DELETE\_ONLY的状态，意味着DDL事务已经不可回滚，不管成功失败都需要恢复线程接手处理。

##### DROP DB

DROP DATABASE的逻辑基本是对DROP TABLE的二次封装。在DDL执行过程中，需要收集被DROP的database中的所有表信息，记录到ddl\_jobs中。对于每张DB中的表 ，执行DROP TABLE的逻辑。

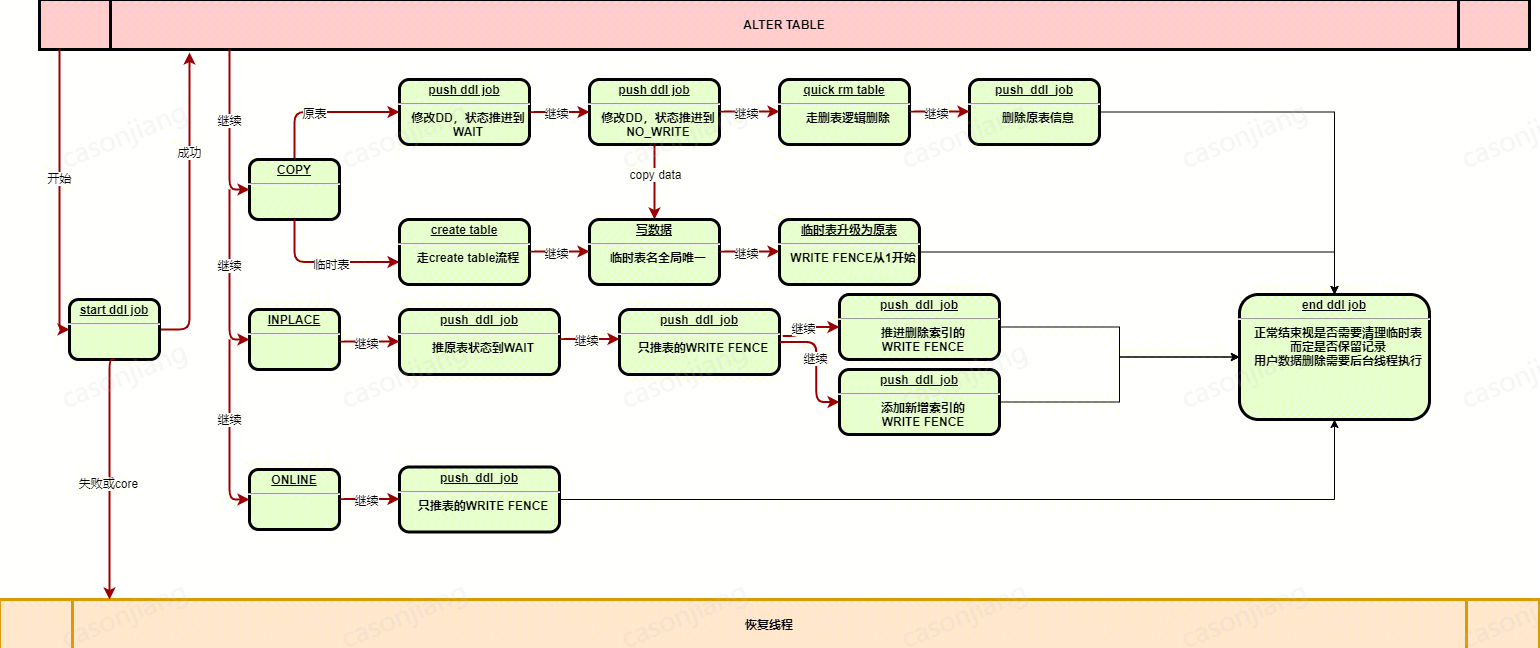


这里逐个删表的逻辑导致的问题在于：

1. 删库阶段各个表所在的状态都不一样；
2. 对于包含大量表的库，由于读写数据字典的操作不能在一个事务里完成，会造成TDStore读写阻塞，外界表象是SQLEngine删库极慢。

##### ALTER TABLE

ALTER TABLE是DDL中最复杂的部分，按照ALTER实现的算法包括：COPY/INPLACE/ONLINE



在COPY算法中，由于需要涉及两表拷数据，其执行流程为：

1、首先会创建一张临时表（记为tmp1），临时表的创建流程和普通的create table流程一致，在创建临时表的过程中，原表依旧可读可写；

2、临时表创建完毕，原表会被推到WAIT状态，该状态会等待已经在这张表上开启的读写事务完成；

3、待2中的事务结束后，原表被推到NO\_WRITE状态，该状态只允许在该表上开启只读事务；

4、开始将数据从原表拷贝到临时表；

5、拷表完成，原表明修改成临时表名（记为tmp2），修改tmp1为原表名；

6、tmp2被删除，删除流程和普通的drop table流程一致。

**COPY对应的恢复流程为：**

1、创建tmp1失败→无需恢复，因为没有实质性的动作发生

2、tmp1创建成功，但是push原表状态失败→删除tmp1，因为原表没有发生任何改变

3、拷数据失败→删除tmp1，因为原表没有发生任何改变

4、修改原表名为tmp2失败→删除tmp1，因为原表没有发生任何改变

5、tmp1改为原表名失败→删除tmp1,且将tmp2改名成原表名

6、删除tmp2失败->继续删除tmp2

**在INPLACE算法中，不涉及拷数据的问题：→删除临时表，因为原表没有发生任何改变**

1、创建内存的临时表结构，原表被推到WAIT状态，并推表的WRITE FENCE

2、推被删除索引（如有）的WRITE FENCE，防止事务使用老索引

3、创建新建索引（如有）的WRIET FENCE，保证正常读写

4、数据回填成功，更新表结构

**INPLACE对应的恢复流程为：**

1、临时表尚未创建→无需恢复

2、临时表已创建→删除临时表

3、临时表已改名成原表

4、清理垃圾数据，推一下WRITE FENCE

**在ONLINE算法中，也不涉及拷贝数据**

1、创建内存临时表结构

2、推进表的WRITE FENCE，防止事务使用老版本

3、更新表结构

**ONLINE对应的恢复流程为：**

1、临时表尚未创建→无需恢复

2、临时表已创建→删除临时表

3、临时改名成原表

4、推一下WRITE FENCE

##### ALTER DB

ALTER DB本身没有涉及用户数据，只能修改元数据，用来修改DB的字符集/可见性等信息。其修改是原子的，不存在中间状态。所以不需要恢复线程的干预。

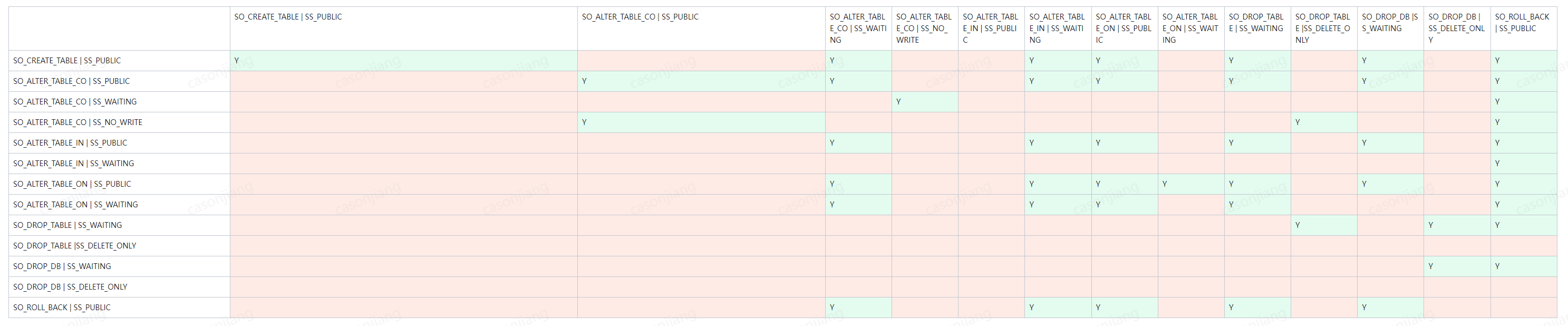
##### RENAME TABLE

RENAME TABLE与ALTER TABLE RENAME语法的作用一样，但是实现上有些差别。

RENAME TABLE允许用户输入类似于rename table t1 to t2, t2 to t3；RENAME本身是原子的，但是类似于多表在一个语句中RENAME，做不到原子，且难以记录，因为RENAME的操作语句中的表名可能前后依赖，不能一把拿全。原生逻辑的atomic\_ddl是可以实现原子操作的，但SQLEngine不行。

恢复线程也无法处理，所以接受RENAME TABLE存在部分成功部分失败。

#### 状态转移矩阵



状态转换矩阵是对DDL状态转换的再次验证，其中标Y的表示从状态1到状态2的转换是允许的。

在DDL恢复线程中不验证状态转换矩阵，因为DDL可能在任意阶段失败，而恢复线程可能从任意阶段恢复，其中的状态转换关系是不确定的。

### 多SQLEngine DDL修正/事务时间戳需要和建表时间戳比较

#### 问题描述

建表create table t1(id int primary key, f1 int, ff int)

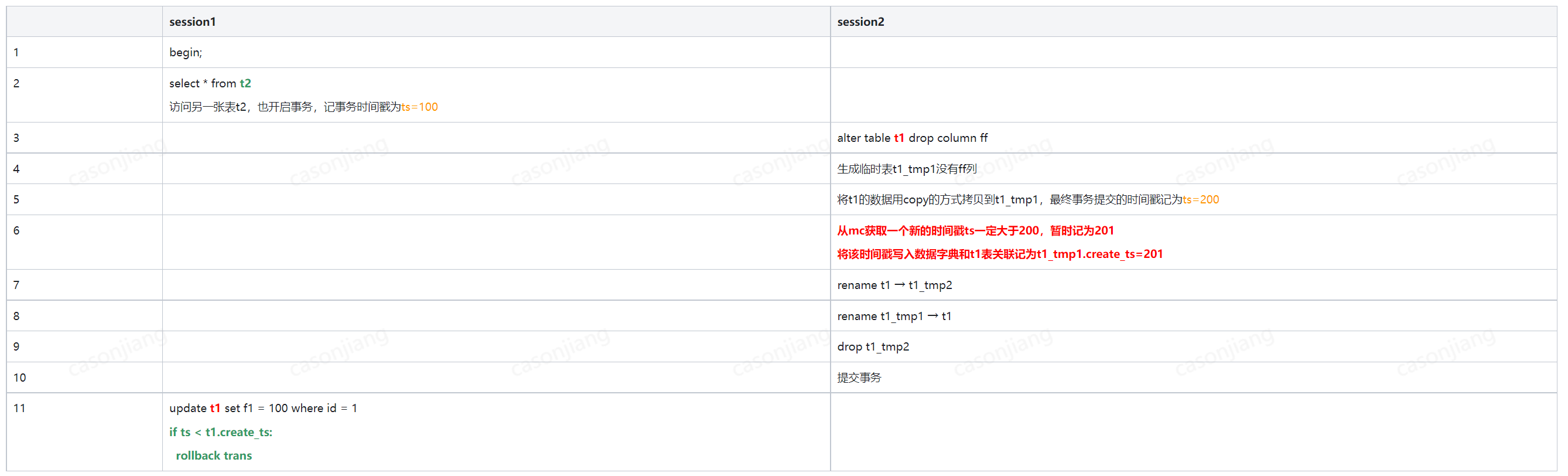
insert into t1 values(1,1,1)



产生这个问题的原因是session1访问t1用的时间戳ts=100对于t1表的数据来说是一个无效的时间戳，因为t1由于拷贝表将数据的时间戳全部推进到了ts=200

#### 解决方案推导

##### copy table ddl



引入步骤6的操作，让session1获取的t1表的时候可以判断出自己用的时间戳不能读取t1表的数据

##### inplace ddl

对于表create table t1(id int primary key, f1 int, f2 int, f3 int)

如果用inplace方式执行alter table t1 add index idx\_f1(f1)，不会拷贝原表，只会扫描pk生成索引idx\_f1

如果我们继续采用copy table中描述的方案改变表t1的时间戳，会导致在t1上用pk读的数据的操作也被错误的回滚掉，这不太友好，更为合理的方式是将这个时间戳记录在表和索引上。



##### online ddl

add column不用变更create\_ts，变更索引需要检查schema\_status，如下：



#### 解决方案总结

##### DDL流程变更

###### create table

1. create table写入数据字典
2. 从mc获取时间戳记为create\_ts
3. 将create\_ts保存在对应表的dd::Tables和dd::Indexs中

###### copy table ddl

1. copy数据到临时表后提交事务
2. 从mc获取时间戳记为create\_ts
3. 将create\_ts保存在对应临时表的dd::Tables和dd::Indexs中
4. rename 临时表

###### inplace alter ddl 变更索引

1. 扫描pk回填idx数据后提交事务
2. 从mc获取时间戳记为create\_ts
3. 将create\_ts保存在表的dd::Indexs中
4. commit 数据字典事务，使index对外可见（index对外可见一定要在第三步之后）

###### online alter ddl变更索引

1. idx\_f1.schema\_status=delete only，写入数据字典提交事务
2. idx\_f1.schema\_status=write only，写入数据字典提交事务
3. 回填数据
4. 从mc获取时间戳ts记为ts=150，idx\_f1.create\_ts = ts = 150更新到数据字典，提交事务
5. idx\_f1.schema\_status=public，写入数据字典提交事务

##### DML流程变更

###### open\_table

if trans\_ts > 0 and trans\_ts < table.create\_ts

rollback trans

###### 用索引去扫数据

if trans\_ts < idx.create\_ts:

# 第一次访问这个表，但是自己的事务时间戳更小，不能用这个索引

rollback trans

##### 数据字典结构变更

上述流程需要在数据字典dd::Tables, dd::Indexs表中添加一个create\_ts字段，考虑到未来多sqlengine架构可能还有其他需要通过数据字典同步的字段，这里未每一个数据字典表都添加一个extra\_info字段，类型为longtext

存储json数据

{

"version": 1, 用于识别后续版本升级

"create\_ts": xxx

}

#### 和write fence 方案的区别

两者解决的主要问题不一样

##### create\_ts方案

决定事务中第一次访问某张表时自己的时间戳是否有效，事务是否可继续（因为之前没访问过这个表，write fence也无法发挥作用）

##### write fence方案

1、通过create\_ts的验证下，借助schema\_status决定第一次访问表的行为（比如delete only不能对这个索引插入）

2、根据tindex\_id，schema\_version，schema\_status决定事务中第二次以及以上访问某张表的行为以及事务是否能推进

#### 扩展比较

innodb中使用的和上述方法类似的思路，只是innodb不是分布式的，所以create\_ts不用持久化在数据字典中，只需要保存在内存中即可。

struct dict\_index\_t {

trx\_id\_t trx\_id; /\*!< id of the transaction that created this

index, or 0 if the index existed

when InnoDB was started up \*/

}

/\*\* Check whether index can be used by transaction

@param[in] trx transaction\*/

bool dict\_index\_t::is\_usable(const trx\_t \*trx) const {

/\* Indexes that are being created are not usable. \*/

if (!is\_clustered() && dict\_index\_is\_online\_ddl(this)) {

return false;

}

/\* Cannot use a corrupted index. \*/

if (is\_corrupted()) {

return false;

}

/\* Check if the specified transaction can see this index. \*/

return (table->is\_temporary() || trx\_id == 0 ||

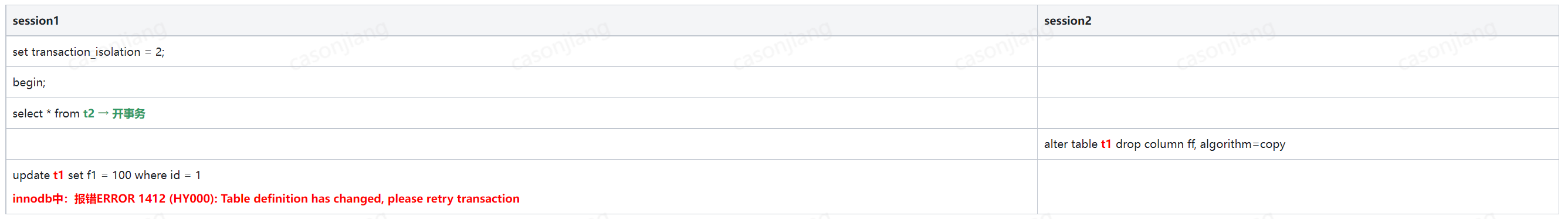
!MVCC::is\_view\_active(trx->read\_view) ||

trx->read\_view->changes\_visible(trx\_id, table->name));

}

建表create table t1(id int primary key, f1 int, ff int)

insert into t1 values(1,1,1)



### 并行数据回填

#### 背景

目前DDL虽然实现了Online，但实际执行的时间依旧取决于表的大小，对于大表，要执行完DDL需要花费大量的时间，主要就是花费在了回填数据的过程中。因此需要实现一种并行回填的机制，加速大表DDL执行的效率。

#### 实现规划

DDL中的数据回填大致分为两类，1.新增索引；2.基于拷表实现的DDL(online或非online)，所以并行数据回填的实现大致会分成两个大阶段。

##### 新增二级索引回填

二级索引回填相对而言更容易实现，其大致过程就是扫描原表全表数据并进行二级索引构造与写入的过程。实现并行回填的前提是对表数据进行划分，实现一个worker线程处理部分数据，所有worker线程同步处理的目标、

###### 不涉及unique key

###### 涉及unique key

涉及unique key的索引回填还需要额外关注unique key的冲突检测，该检测也要改造成并行。

##### 基于拷表的数据回填

#### 实现方案

##### 新增二级索引回填

###### 串行回填实现（当前逻辑）

当前串行回填实现的实现方式是扫描全表数据，构建创建的SK并通过thomas write的形式回填到TDStore中。

index fillback

|-mysql\_alter\_table

|--mysql\_online\_alter\_table

|---tdsql::ddl::OnlineAddIndexes

|----tdsql::ddl::FillBackIndexes

|-----handler::ha\_index\_fillback\_sk

|------myrocks::ha\_rocksdb::index\_fillback\_sk // 并行回填改造，全表扫描改为range扫描

|-------myrocks::ha\_rocksdb::index\_first\_intern // 构建扫描range，全表扫描就使用了index\_prefix(4Bytes),希望这里能进行切分

|-------tdsql::BatchPutCntl::AddKV

###### 并行回填实现

基于串行回填这一实现，一个朴素的并行回填的执行方式是将全表扫描的数据进行切分。（按region或者按index切分不同的[start\_key, end\_key)）。思考：对index\_prefix进行切分如何能保证相对均衡？参考PK的长度？参考region分布？

首先需要解决的问题是范围如何进行切分：

假设设置的并行度为P，主键范围为[10000,10001)

方案1：按照主键的范围进行均匀切分，也就是将区间[10000,10001)划分为P个区间，每个task执行1个回填任务,这里可以参考range scan的实现方式。

这里的问题是，这个区间比较难划分出来，即使划分出来，数据分布可能也并不均匀。比如某些区间涉及多个region，某些区间只涉及单个region。

|-IndexRangeScanIterator::Read

|--QUICK\_RANGE\_SELECT::get\_next

|---handler::ha\_multi\_range\_read\_next

|----handler::multi\_range\_read\_next

|-----myrocks::ha\_rocksdb::read\_range\_first

|------myrocks::ha\_rocksdb::index\_read\_map\_impl

|-------myrocks::ha\_rocksdb::init\_tdsql\_iterator

方案2：按照region进行划分，根据[start\_key, end\_key)来获取region信息的方式目前是成熟的。尽管每个region上的数据分布并不一定完全均匀，但总体而言这种方式是可以均衡子任务的负担的。

具体步骤：

1、根据index\_id求出待回填数据（PK）的start\_key,end\_key记作[pk\_skey, pk\_ekey)

新增一个handler接口tdsql\_get\_storage\_format\_pk用于转换SQL层的PK的index id与storage format的index id

假设storage format的PK index id范围是[B, C)

2、根据[pk\_skey, pk\_ekey)求出这个范围内的region信息，得到若干个region

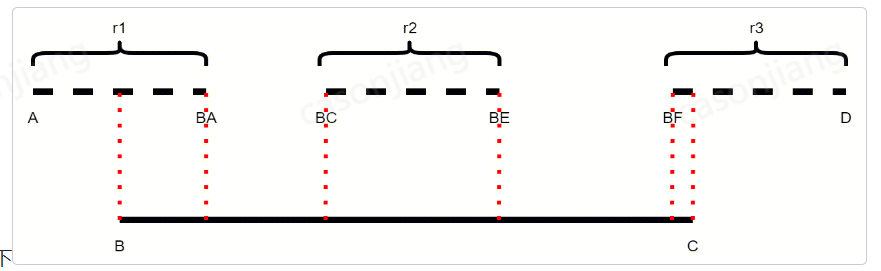
这里获取region信息的方案很成熟，所以调用已有接口即可

此处求出各个region的范围，每个region中的信息不一定都符合处于[B, C)之间

假设获取到了3个相关region r1[A, BA), r2[BC, BE), r3[BF, D)

3、求出每个region的数据范围与[pk\_skey, pk\_ekey)的交集，求出的值就是我们希望在这个region上扫描的数据，如此往复得到N个范围

1）作出示意图如下



2）[B, C)中实际有数据的范围就是[B, BA),[BC, BE), [BF, C)

3）求得以上范围后实际上已经掌握真实的数据信息了，后面即使r1,r2,r3发生分裂，我们也不需要关心

4、构建回填子任务

1）根据tdsql\_parallel\_thread\_number的值初始化子任务队列，这是最大并发数，由并发框架控制

2）第3步中求出了每个子任务的范围，那么每个子任务只需要回填这一个范围的数据就行了

3）新增handler接口ha\_index\_fillback\_sk\_by\_range，原理和全表扫描一致，只不过全表扫描的是[B,C)，而该接口可以扫指定范围

4）task1扫描回填[B, BA)，task2扫描回填[BC, BE)，task3扫描回填[BF, C),

5）尽管看上去每个子任务的工作量可能并不均衡，但相较于直接均分[B,C)，上述过程至少保证每个子任务都“有活干”

伪代码表示：

tdsql::Slice start, end;

tdsql\_get\_storage\_format\_pk(&start, &end)

foreach mem in mems:

KeyRangeIntersectMember(start, end, mem, key\_range) // 求出每个region上属于[start\_key, end\_key)的部分

GenerateTask(key\_range) // 求出每个子任务需要处理的真实范围

task = create\_task(key\_range) // 每个子任务只处理部分数据

fillback\_task.add\_task(task)

fillback\_task.run() // 每个工作线程调用ha\_index\_fillback\_sk\_by\_range，并行执行子任务

###### 并行回填涉及unique key

如果新增二级索引没有unique key约束，那么按照3.1.2中介绍已经可以完成并行化目标。但如果增加的index是唯一索引，那么上述实现尚不足够，还需要对唯一性检查进行并行化处理。

基本思路：

这里需要有一处可以记录被PUT的二级索引的min key和max key的位置，有了[min\_key, max\_key)也就可以确定单个回填任务需要确认的unique key信息。handler信息中，不管是Rdb\_key\_def还是Rdb\_table\_def都是全局共享的，所以不能将信息直接放在这两个共享结构中，而应该单独为记录[min\_key, max\_key)开辟空间。由于每个并行任务都单独open了TABLE，所以这个空间放在handler中应该可行。

这里需要考虑的问题有：

1、单次添加多个index，怎么区分？

使用vector来区分多个index？

2、何时重置[min\_key, max\_key)的值？

填充[min\_key, max\_key)之前

3、并行执行中某个子任务报错，主线程如何得知？比如唯一索引冲突？

4、重复的index值恰好被分到了两个任务中怎么办？

当前模型无解，子任务之间无法相互认证，只能是回填完之后完整检查一遍，能做的最大并行就是对单次新增多个UK的情况下，每个UK可以并行检查。

重复二级索引的前缀相同，这里能否在Thomas写的时候下推给TDStore辨别？

##### 基于拷表的数据回填

基于拷表的数据回填本质上也是扫描PK，比二级索引回填稍复杂的是，不能直接在rocksdb引擎层调用thomas put接口进行数据写入，而要通过handler层的ha\_write\_row自上而下地进行数据回填。

###### 串行回填实现

串行online copy

|-mysql\_alter\_table

|--tdsql::ddl::OnlineCopyAlter

|---tdsql::ddl::OnlineCopyDataToNewTable

|----tdsql::ddl::CorrelateOption::WriteRow

|-----handler::ha\_write\_row

|------myrocks::ha\_rocksdb::write\_row

|-------myrocks::ha\_rocksdb::update\_write\_row

|--------myrocks::ha\_rocksdb::flush\_cache\_key

|---------Transaction\_impl::flush\_batch\_record

###### 并行回填实现

基本同并行回填实现方案2一致，不同的是，基于拷表回填的batch KV缓存在事务中，而多个task共用一个事务对象缓存必然会造成问题。针对这一问题，想了两个方案：

1、在事务对象中开辟多个batch KV->a.影响事务在并行框架中的只读属性；b.维护事务到task的映射关系比较复杂

2、选一处每个task独有的位置存放batch KV

这里选择第2种方式，在handler中新增一个Thomas写对象来缓存batch KV，由于每个task都会新open一个表对象，所以task之间的batch KV是不会相互影响的。

**online copy伪码：**

class ha\_rocksdb {

ParallelTransaction\* m\_parallel\_txn; //用于并行RPC

ParallelThomasRulePutCntl\* m\_parallel\_thomas\_put; //为每个handler分配batch KV的存储空间

bool in\_parallel\_mode() { return m\_parallel\_txn != nullptr; } //指示当前handler是否是并行子任务

}

|-ha\_rocksdb::update\_pk

...

if (in\_parallel\_task())

flush\_cache\_key(K,V) //待写入数据暂存至handler

else

flush\_cache\_key(tx,K,V) //待写入数据暂存至事务

|-ha\_rocksdb::update\_sk

...

if (in\_parallel\_task())

flush\_cache\_key\_in\_parallel(K,V) //待写入数据暂存至handler

else

flush\_cache\_key(tx,K,V) //待写入数据暂存至事务

|-ha\_rocksdb::update\_write\_row

...

if (in\_parallel\_task())

flush\_cache\_key\_in\_parallel() //将handler的KV刷到tdstore

else

flush\_cache\_key(tx) //将事务中的KV刷到tdstore

|-ha\_rocksdb::end\_bulk\_insert

...

if (in\_parallel\_task())

flush\_batch\_record\_in\_parallel() //将handler剩余的KV刷到tdstore

else

tx->flush\_batch\_record() //将事务中剩余的KV刷到tdstore

###### 问题

实现过程中遇到的主要问题就是THD的使用，大致分两类：

1. current\_thd为空导致断言失败；
2. 针对THD变量做的一些backup/recover之类的工作可能导致变量无法恢复到最初值。