**Palo2存储层设计文档**

修改记录

|  |  |  |  |
| --- | --- | --- | --- |
| No | 修改内容 | 日期 | 修改人 |
| 1 |  |  |  |
| 2 |  |  |  |
| 3 |  |  |  |

## 背景

Palo 1.x版本中存在一些问题，如下：

1. 元数据问题

Palo 1.x中所有的元数据持久化在MySQL中，并且为了更新方便，元数据表进行了normalization设计。随着集群中用户及表增加，元数据量也不断增大。很多操作都涉及到sql join，性能问题也逐渐开始明显。FE需要通过DM来获取元数据，代价也比较大。

1. 进程太多

每个模块涉及的进程太多，造成部署及运维比较麻烦。同一台机器上的不同模块之间也需要通过网络进行交互，效率低。

1. 两个客户端问题

用户需要使用两个客户端来完成管理和查询功能，很容易给用户造成误解，使用上也带来不便。

1. Master和Slave交互方式

Master与Slave之间的交互都是通过不断轮询Meta来实现，没有任务存在的情况下仍然需要去执行轮询操作，造成一定的资源浪费。而且这种方式任务下发到Slave会有延迟。

1. 依赖组件过于重量

依赖比较重量的MySQL，不利于维护，不利于云化部署，并且也没有较好的HA方案。

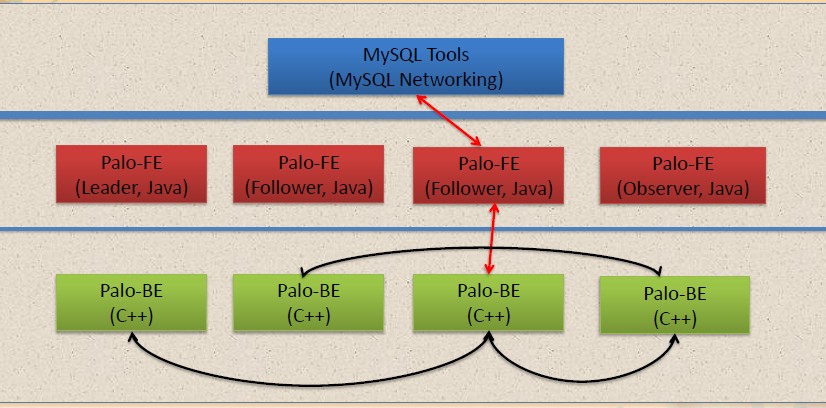
因此存储层管理的框架需要重新设计实现。

## 总体概述

本期要达到的目标如下：

1. DM和FE合为一个进程FE，Agent、Olapengine和BE合为一个进程BE。
2. 元数据采用内存和操作日志的方式进行管理，该技术也被称作Replicated State Machines。这是一种常用的、高效的、高可用的管理元数据的的技术，GFS、HDFS、RAMCloud都采用了此技术。
3. 用户统一通过MySQL客户端来管理和查询表。

总体的实现架构如下：



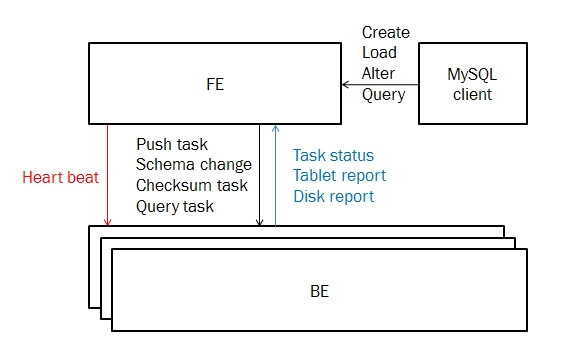
FE来接收用户管理和查询的sql，管理元数据及各类任务，管理BE的状态信息，执行查询规划。

BE接收并执行FE下发的各类任务，包括建表、删表、导数据、schema change、查询等。

## 重要模块功能设计要点

## FE和BE之间的交互

### 框架



FE需要维护各类任务的队列，按照BE来组织，主动将任务发送给BE。BE异步处理收到的任务，并将任务的状态定期向FE汇报，完成后向FE汇报结果。

FE通过心跳来查询BE的状态。（FE定期向BE发送，好处是FE挂了切换备FE方便，另外心跳可以通过专用的thrift server来管理）

FE发送给BE的命令都是异步执行的，并且都是幂等的。

BE周期性地汇报其管理的tablet信息、磁盘使用情况和各类任务的状态信息。

### FE BE重启

1. FE重启

流程如下：

1. FE加载Image，并回放edits，恢复元数据内存结构，并生成还未完成的任务放入各自的队列。
2. FE加载BE列表，向BE握手，告诉其汇报的FE地址。并将对应BE的任务重新发送给BE。
3. BE向FE汇报tablet信息。
4. BE重启

其本身是无状态的，BE启动后加载完所管理的副本信息后就能开始服务，之后会收到FE发送的心跳信息及还未完成的任务，然后BE可以将tablet信息和磁盘使用情况汇报给FE。

## 内存主要的数据结构

元数据存放在内存中，并通过image和edits持久化到磁盘上。主要包括表相关信息（Database、TableFamily、Table、Tablet、Replica、Column等）、BE相关信息、各类Job信息（导入、roll up、schema change、分裂等）、Task信息（Job中需要下发给BE的任务）。

### 表相关

**Map<Long, Database> idToDb;** // id to db

**Map<String, Database> nameToDb;** // name to db

**Database**

Long id;

   String name;

         Map<Long, TableFamily> idToTableFamily;

         Map<String, TableFamily> nameToTableFamily;

         Map<Long, View> idToView;

         Map<String, View> nameToView;

Map<String, List<Function>> nameToFunction;

         ReadWriteLock rwLock;

**TableFamily**

Table baseTable;

         Map<Long, Table> idToRollupTable;

         Map<String, Table> nameToRollupTable;

         TableFamilyType type; (OLAP/MYSQL)

TableFamilyState state; (NORMAL/ROLL\_UP/SCHEMA\_CHANGE/SPLIT)

Long committedVersion;

Long committedVersionHash;

**Table**

         Long id;

         String name;

         List<Column> columns;

         Map<String, Column> nameToColumn;

         Map<String, String> properties;

**OlapTable** extends **Table**

Partition partition;

List<OlapTablet> tablets;

Map<Long, OlapTablet> idToTablet;

StorageType storageType; (Columnar/RowColumnar)

Short shortKeyColumnCount;

Short replicationNum;

Int schemaVersion;

Long schemaHash;

List<Column> schemaChangeTo; // for schema change

TableState state; (NORMAL/ROLL\_UP/SCHEMA\_CHANGE/SPLIT)

Long rowCount;

**Partition**

         Int bucketNum;

PartitionType type; (RANDOM/HASH/KEY\_RANGE)

**RandomPartition** extends **Partition**

**HashPartition** extends **Partition**

List<Column> partitionColumns;

**KeyRangePartition** extends **Partition**

List<Column> partitionColumns;

RangeMap<OlapKey, OlapTablet> keyToTablet;

**OlapTablet**

         Long id;

         List<OlapReplica> replicas;

**OlapReplica**

         Long id;

         Long backendId;

         ReplicaState state; (INIT/NORMAL/ROLL\_UP/ROLL\_UP\_FINISH/…)

         Long version;

         Long versionHash;

         Long rowCount;

Long dataSize;

Disk disk;

**Column**

        String name;

         PrimitiveType dataType;

AggregationType aggType;

String defaultValue;

Int strLen;

Int precision;

Int scale;

**Map<Long(BackendId), Map<String, CreateReplicaTask>> createReplicaTasks;** // BE tabletId to create replica task

**CreateReplicaTask**

Long dbId;

Long tableFamilyId;

Long tableId;

Long tabletId;

Short shortKeyColumnCount;

Long schemaHash;

StorageType storageType;

List<Column> columns;

### BE

**Map<Long, Backend> idToBackend;** //BE映射

**Backend**

Long id;

Long host;

Int heartPort; //心跳用

Int agentPort; // 控制用

Int queryPort; // 查询用

Long lastUpdateMs;

DiskStats diskStats;

### 导入和按条件删除

**Map<Long, LoadJob> idToLoadJob;** // loadJobId to loadJob

**Map<Long, List<LoadJob>> dbToLoadJobs;** // db to loadJob list

**Map<Long, Map<String, List<LoadJob>>> dbLabelToLoadJobs;** // db label to loadJob list

**Map<Long, Map<Long, Long>> tableFamilyVersionHistories;** // tableFamilyId version to loadJobId

**LinkedHashMap<Long, LoadJob> idToPendingLoadJob;** // id to pending job

**LinkedHashMap<Long, LoadJob> idToEtlLoadJob;** // id to etl job

**LinkedHashMap<Long, LoadJob> idToLoadingLoadJob;** // id to loading job

**LinkedHashMap<Long, LoadJob> idToQuorumLoadJob;** // id to quorum finished job

**Set<Long> loadingTableFamilyIds;** // loading table family id set

**LoadJob**

Long id;

Long dbId;

String label;

Int timeout;

Double maxFilterRatio;

String cluster;

JobState state;

Int progress;

Long creationTime;

Long etlStartTime;

Long etlDoneTime;

Long loadingStartTime;

Long loadingDoneTime;

String failMsg;

String etlJobId;

EtlJobStatus etlJobStatus;

Boolean etlDataDeleted;

JobType type;

Map<Long, TableFamilyLoadInfo> idToTableFamilyLoadInfo;

Map<Long, TableFamilyDeleteInfo> idToTableFamilyDeleteInfo;

Map<Long, TabletLoadInfo> idToTabletLoadInfo;

Set<Long> quorumTablets;

Set<Long> fullTablets;

Boolean allReplicaFinished;

**TableFamilyLoadInfo**

Long version;

Long versionHash;

List<Source> sources;

**Source**

List<String> fileUrls;

List<String> columnNames;

String columnSeprator;

Boolean isNegative;

**TabletLoadInfo**

String filePath;

Set<Long> sentReplicas;

Set<Long> finishedReplicas;

Long tableFamilyId;

Long tableId;

**TableFamilyDeleteInfo**

Long version;

Long versionHash;

List<DeleteCondition> conditions;

**DeleteCondition**

String columnName;

String conditionOp;

String conditionValue;

**Map<Long(BackendId), Map<String, PushTask>> pushTasks;** // BE pushIdent to push task

// pushIdent: tabletId + version

**PushTask**

Long dbId;

Long tableFamilyId;

Long tableId;

Long tabletId;

Long version;

Long versionHash;

String filePath;

Long loadJobId;

### 副本均衡

**Map<Long(BackendId), Map<String, CloneTask>> cloneTasks;** // 不同的优先级队列

**CloneTask**

Long tabletId;

Long backendId;

CloneTaskState state;

CloneTaskType type;

Long startTime;

Long doingTime;

Long finishTime;

String failMsg;

### Roll up

**Map<long, RollupJob> rollupJobs** // need Persistence, new table id to rollup job

**RollupJob**

long databaseId;

long tableFamilyId;

long tableId; // new table id

long baseTableId; // can’t be deleted before rollup finish

Map<long, long> // new tablet id map to base tablet id

List<**Column**> columns

**Partition** partiton

String tableName

List<long> unfinishedReplicaIds

**Map<Long(BackendId), Map<String, RollupTask>> rollupTasks;** // string is a signature

**RollupTask**

long backendId;

long databaseId;

long tableFamilyId;

long tableId;

long tabletId;

long baseTabletId;

List<**Column**> columns;

Long schemaHash;

short shortKeyColumnCount;

TableDataFileType dataFileType;

### Schema change

**Map<long, SchemaChangeJob> schemaChangeJobs //** need Persistence, key is table family id

**SchemaChangeJob**

long databaseId;

long tableFamilyId;

List<**Column**> columns // new column defs

Map<long, List<String>> table\_defs // tableId to column names

List<long> tabletIds // all relate tablet id

List<long> unfinsihReplicaIds // unfinish replica id

**Map<Long(BackendId), Map<String, SchemaChangeTask>> schemaChangeTasks;** // tabletId should be signature

**SchemaChangeTask**

long backendId;

long databaseId;

long tableFamilyId;

long tableId;

long tabletId;

List<**Column**> columns;

Long schemaHash;

short shortKeyColumnCount;

TableDataFileType dataFileType;

### 分裂

**Map<long, SplitJob> splitJobs** // key is tablet id

**SplitJob**

long databaseId;

long tableFamilyId;

long tableId;

long tabletId;

list<long> subTabletIds;

list<long> unfinishReplicaIds;

String splitKey;

**SplitJobStatus status; //** (INIT / QUERY / RUNNING / FINISHED/DELAY\_DELETE)

Long modifyTime;

**Map<Long(BackendId), Map<String, GetSplitKeyTask>> getSplitKeyTasks; /**/ tabletId should be signature

**Map<long, long> splitKeys; // story replicaId and splitKey**

**Map<Long(BackendId), Map<String, SplitTask>> splitTasks;** // tabletId should be signature

**GetSplitKeyTask**

long backendId;

long databaseId;

long tableFamilyId;

long tableId;

long tabletId;

List<String> partitionColumnNames

**SplitTask**

long backendId;

long databaseId;

long tableFamilyId;

long tableId;

long baseTabletId;

long child1TabletId;

long child2TabletId;

List<**Column**> columns;

Long schemaHash;

short shortKeyColumnCount;

TableDataFileType dataFileType;

## 主要流程

### 建表

1. 用户通过MySQL客户端连接FE，执行建表命令。
2. FE生成表相关的信息（随机选取online的BE），多线程发送创建replica命令给BE。
3. BE同步创建replica，并返回结果。
4. FE收到BE返回的结果，如果所有replica都创建成功，则将表相关的信息添加到元数据，返回成功给用户。如果有replica创建失败，返回失败给用户。已创建的replica通过BE汇报来删除。

### 导入和按条件删除

导入流程如下：

1. 用户通过MySQL客户端连接FE，执行导入命令。
2. FE将导入信息（包括导入的表及对应的导入文件等）添加到元数据中，并将导入任务状态置为pending。
3. FE轮询pending状态的任务，提交给hadoop进行处理。成功提交后，记录hadoop etl任务的job id，并更新导入任务的状态为etl。
4. FE轮询etl状态的任务，根据hadoop etl任务的状态更新导入任务进度，当hadoop etl完成后获取生成文件的列表，并将文件与tablet的对应关系添加到导入任务中，更新导入任务对应的版本号信息，更新导入任务的状态为loading。
5. FE轮询loading状态的任务，将push任务加入push队列，同时给相应的BE发送push命令。BE收到push任务后，将任务加入push队列并执行push操作。执行过程中和完成时都会向FE汇报当前进度。FE不断检查导入任务是否满足quorum，如果满足，则更新任务状态为finished。
6. FE轮询quorum完成的任务，最终确保每个副本导入完成。

按条件删除的实现按照push空版本加删除条件的方法，由于在每个副本上的操作很快能完成，所以采用同步来做。

1. 用户通过MySQL客户端连接FE，执行删除命令。
2. FE先做一些严格检查（比如副本online、版本情况），如果不满足检查条件，则返回失败给用户。FE生成删除任务信息，并多线程发送按条件删除命令给BE。
3. BE同步增加空版本并记录删除条件，并返回结果。
4. FE收到BE返回的结果，如果所有副本都执行成功，则更新任务状态为成功，并返回成功给用户。如果有些副本没执行成功，则回滚已成功的副本，并返回失败给用户。

导入的过程中可能出现版本未追齐的情况，处理如下：

1. 正常导入任务

导入副本满足quorum完成后，导入任务即可成功。导入线程需要轮询检查quorum完成的任务，确保任务中的每个副本都追上版本。

对于含有正在roll up表的table family的导入，导入完成时可以不检查roll up表，roll up表的数据由base来转化。

Push task失败时，BE应该返回当前副本的version及version hash，方便及时纠错。BE在做tablet汇报时，也需要更新元数据。

1. 其他情况

对于clone等操作后，副本版本是落后的，则BE做tablet汇报的时候，通过table family导入版本历史信息找到对应生效版本对应的导入任务，来触发追版本。

### 副本均衡

包括副本增加、副本删除、副本迁移三个流程。

1. 副本增加

当某个BE不能连接（可能网络问题或宕机）、BE上的一块盘坏掉或者BE发现某个副本损坏时，需要将对应的副本删除，FE来调度完成副本的补足。流程如下：

1. FE轮询元数据中的tablet，获取副本不足的tablet（BE汇报副本损坏时可以主动触发）。按照一定规则选择BE，将任务加入clone队列，并执行clone操作。规则如下：
2. 优先选择空间使用率低的BE
3. 表tablet的分布尽量均匀
4. BE收到clone任务后，将任务加入clone队列并执行clone。执行过程中和完成时都会向FE汇报当前进度。Clone完成后向FE汇报完成。
5. FE收到任务完成的汇报后，在元数据中生效副本，删除队列中的任务。
6. 副本删除
7. FE轮询元数据中的tablet，获取副本过多的tablet。按照一定的规则选择要删除的副本，将任务加入删除队列，并执行删除操作。规则如下：
8. 优先删除offline、版本低的副本
9. 优先选择空间使用率高的BE
10. 删除后表tablet的分布尽量均匀
11. BE收到删除任务后，将任务加入删除队列并执行删除。执行过程中和完成时都会向FE汇报当前进度。删除完成后向FE汇报完成。
12. FE收到任务完成的汇报后，删除队列中的任务。
13. 副本迁移

该流程主要是将空间使用率高的BE上的副本迁移到空间使用率低的BE上并且将每个table的tablet平均分配到各BE上。FE轮询元数据中的tablet，获取空间使用率高或者table中tablet分配较多的BE上的tablet。后续的操作按照副本增加和副本删除的流程来完成均衡。

1. 副本均衡策略
2. 策略

副本均衡的目标如下：

1. 补足副本，保持系统可靠。
2. 每个BE的空间使用率接近。
3. 每个table的tablet尽可能均匀分布，查询分布更均匀。

增加副本和减少副本时应该尽可能实现以上的目标。对于减少副本的操作，还应该优先删除offline、版本低的副本。

建表时会随机选择BE来分配副本。

1. 触发时间

当某个BE不能连接、BE上的一块盘坏掉或者BE发现某个副本损坏时，均会触发副本增加。后两种情况可以立即触发，第一种情况可以先暂时将BE置为不可查（可能由于网络问题，一段时间后会恢复），延迟触发副本增加。

根据tablet副本数和online的情况，来计算clone任务的优先级。优先级高的clone优先进行。优先级最高的情况（比如tablet只有1个副本）也应该立即触发。

1. 限制

需要考虑整个集群和每个BE clone任务的数目。BE clone时需要限速。

### Roll up

1. 用户通过MySQL客户端连接FE，执行建立Rollup命令
2. 对提交的请求进行基本的参数检查，根据table family是否提交过导入数据，检查rollup的列、分区信息和base rollup是否合理。
3. 根据table family是否有提交过数据导入：
4. 没有提交过导入：后面和create table一样的流程
5. 提交过导入：进入create rollup流程。以下是create rollup流程。
6. 构建table(rollup)对象,添加到table family对象下。table family、table和新table的replica对象的状态都为roll up。
7. 生成create rollup到任务队列，并分发到各BE。
8. 当元数据修改后，再提交的导入作业，dpp不会生成新的rollup对应的导入数据和push task。
9. 从BE接收到完成作业的消息：
10. 当接收到finish rollup的消息后，将对应task对应的replica的状态改为rollup finish。此时新版本的数据还是由base来转化。
11. 当unfinishedReplicaIds的列表为空时，将table的状态改为rollup finish。（此处必须是所有replica，防止有replica在没有完成的情况下，再次进行schema change类操作）。
12. FE中有一个线程轮询rollup作业，检查是否新建table的tablet的replica版本追上TableFamily并且没有正在执行的导入任务，追上就要将rollup状态改为normal, TableFamily状态改为normal。此后提交的导入，都会生成新建table的导入数据和push task。判断一个批次是否完成时，也要检查新建table的导入是否完成。

具体的轮询逻辑：检查rollupJobs中Table的状态，如果table状态为rollup finish，检查LoadJob列表中该DB的最新一个作业的fullTablets中是否包含所有新建table的tablet。如果是，说明版本已追赶成功，可以用于查询服务，将table的状态改为normal。将Job信息删除。

1. Roll up表完成后的新导入，不能保证base和roll up导入的先后顺序，可能造成roll up数据依旧由base来转化，但是roll up表也会重新push，olapengine会用roll up的数据替换base转化的版本，从而保持数据一致。
2. 如果rollup整体超时或失败，将rollup的元信息从TableFamily中删除，BE上产生的垃圾文件会在之后的汇报中由FE判断为无用文件，进而进行删除。将Job和task信息删除。

状态变化：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Rollup开始 | Replica 完成rollup | 所有replica完成rollup | 所有tablet追上版本 |
| TableFamily | Rollup(不可schema change) | Rollup | Rollup | Normal（可进行其他schema change） |
| Table(Rollup) | Rollup（不可查询，push完成不检查） | Rollup | Rollup finish（不可查询，push完成不检查） | Normal（可以查询，push完成需检查） |
| Replica | Rollup（不可导入） | Rollup finish（不导入） | Rollup finish（不导入） | Normal（可导入） |

### Schema change

1. 用户通过MySQL客户端连接FE，执行schema change命令
2. 针对提交的参数进行检查。TableFamily的状态要是normal，不能包含未完成的数据删除作业，不能有一个以上的schema版本。
3. 将新的schema版本信息添加到各个table，新版本schema不生效。
4. 遍历所有涉及到的table的replica，将replica的状态改为schema change，table和table family的状态也改为schema change。
5. 给每个replica生成SchemaChangeTask。并下发给各自的backend。
6. 从BE接收到完成作业的消息：
7. 当接收到finish schema change的消息后，将对应task对应的replica的状态改为schema change finish。从对应SchemaChangeJob中的unfinsihReplicaIds中删除。将task信息删除。
8. 当unfinsihReplicaIds为空时，将TableFamily的状态改为normal，Table状态改为normal。生效新版本的schema，旧版本schema无效。（因为导入的task中是包含schema 的版本的，即便有旧schema的导入数据还有没完全导入，只要olap上旧的schema版本还在，还是可以导入数据的。）
9. 如果某些schema finish返回失败，在整体没有超时的情况下进行重试。
10. 如果schema change整体超时或失败，将schema的版本信息从各Table中删除，BE上产生的垃圾文件会在之后的汇报中由FE判断为无用文件，进而进行删除。
11. old schema删除：DM轮询没有删除且unfinsihReplicaIds为空的schema change job，尝试进行old schema的删除。

由于新版本的schema生效前，可能已经产生了很多旧schema的导入数据，在old schema删除前必须保证所有这些批次已经导入。所以在删除前要保证所有TableFamily相关的导入作业要全部完成（全部副本导入完成，不能仅仅是满足quorum）。检查方法为：检查LoadJob列表中该DB的最新一个作业的fullTablets中是否包含所有schema change的table的tablet。

将old schema从table的元数据中删除，将schema change job信息删除。生成drop table（针对旧schema版本）的task，并下发给各BE。（不用生成job，因为drop的task即便丢失了，以后BE和FE对照olap表时，也会重新生成drop task的）

状态变化

|  |  |  |  |
| --- | --- | --- | --- |
|  | schema change  开始 | Replica 完成Schema change | 所有replica完成Schema change |
| TableFamily | schema change(不可schema change) | schema change | Normal(可进行其他schema change，schema版本生效) |
| Table | schema change | schema change | Normal |
| Replica | schema change（可导入） | schema change finish（可导入） | Normal（可导入） |

### 分裂

1. 通过配置决定集群是否进行分裂操作。在配置允许进行split的情况下，进行split。
2. 当BE汇报给FE每个olap的tablet的信息时，对tablet的信息进行检查，当发现tablet超过一些限制时（比如大小等），且tablet的table family和tablet的状态是normal或split且只包含一个schema版本的、且数据方式为key range的tablet进行分裂。检查tablet所属的table family是否有未完成的delete作业。检查tablet的所有replica副本online的数>=quorum。
3. 内存中有一个splitJobs保存进行分裂的tablet及状态。将需要进行分裂的tablet按照权重顺序遍历，如果需要进行split的tablet的id不存在于splitJobs中且splitJobs的长度未达到配置中最大进行分裂的个数，则放入splitJobs，标记状态为INIT。
4. 遍历splitJobs，对init状态的tablet，选取版本最新的一个replica进行分裂点的查询。生成GetSplitKeyTask，并下发给BE。记录下发的时间modifyTime，修改状态为QUERY。
5. 通过modifyTime检查QUERY状态的是否有超时，超时的重新进行下发GetSplitKeyTask。
6. 当从BE收到返回的query\_split\_key的结果时，设置到对应的tablet信息中。
7. 将已获取query\_split\_key的tablet，按照query\_split\_key构建两个新的tablet，设置新tablet的状态为split，replica的状态为split，table family的状态为split。~~将新的tablet信息添加至table下~~（考虑到split是FE内部触发的，如果中途宕机，采取整体重新做的策略）。按照query\_split\_key构建每个replica的SplitTask，并下发给BE。在splitJobs中记录modifyTime，修改tablet信息为RUNNING。
8. 之后的轮询中，检查RUNNING状态的tablet是否被删除，如果被删除，将对应的tablet信息从splitJobs中删除。根据modifyTime检查是否超时，如果超时重新发送SplitTask。
9. 接收到BE发送的split的结果：
10. 如果是成功的，将replica的状态从split改为split\_finish，从unfinishReplicaIds中去除。删除split task信息。
11. 如果失败，重新发送SplitTask，更新modifyTime。
12. 如果unfinishReplicaIds为，将splitJobs的状态改为FINISH。
13. 轮询FINISH状态的所有split job，检查LoadJob列表中该DB的最新一个作业的fullTablets中是否包含所有新建的tablet。如果是，说明版本已追赶成功，不会再有dpp只生成了base tablet的导入，防止之后延迟删除base tablet导致导入无法成功。（就是因为会异步删除base tablet的数据，所以必须保证子tablet生效、父tablet删除的时候，所有以父tablet生成导入数据的批次必须所有replica导入完成，否则就有可能父tablet删除后，导入只有父tablet导入数据的批次无法导入数据）将两个子tablet和replica的状态改为normal，base tablet的状态改为delete。将table family的状态改为normal。删除splitJobs中的tablet相关信息。更新job的状态为DELAY\_DELETE，更新modifyTime。
14. FE轮询状态为DELAY\_DELETE的split job，如果当前时间与modifyTime的差值大于设置时间（FE查询的查询超时时间，主要是防止有大查询未完成，还需要BE上的olpa表进行查询），删除base tablet的元数据，删除split job，生成删除对应replica的drop table的task。
15. 关于取消操作：由于split和rollup、schema change操作是互斥的。可能用户需要提交这次操作的时候，split正在进行，所以需要可以手动进行取消。取消操作记录提交的table family的列表，在选取分裂tablet时，跳过属于这些table family的tablet，同时将splitJobs中属于这些talble family的talbet的记录删除，相关子tablet的元信息删除。相关未完成分裂的replica生成drop task进行删除。

状态变化

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | split开始 | Replica 完成split | 所有replica完成split | 子tablet完全追上版本 |
| TableFamily | split (不可schema change、rollup可以split) | split | split | Normal（可进行其他schema change） |
| Base Table | normal | normal | normal | Delete |
| Table | split（不可查询，不生成push task，不可split） | split | split finish（不可查询，不生成push task） | Normal（可以查询，生成push task） |
| Replica | split（不可导入） | Split finish（不可导入） | Split finish（不可导入） | Normal（可导入） |

### 各流程关系

当前的流程包括导入、按条件删除、roll up、schema change、分裂、clone。之间的关系如下：

1. 导入与导入

同一个table family的loading阶段是串行的，其他阶段是可以并行的。

1. 导入与按条件删除

按条件删除是通过导入空版本的方式来实现，所以删除和导入共用了loading过程。因此删除任务过程中，可以提交导入任务，但是loading阶段是串行的。

为了让用户更加明确是删除哪个批次之前的数据，所以删除时不能有正在导入（未结束状态）的任务。

1. 导入与roll up

Roll up的过程中可以提交导入。Roll up表的数据由base表来追齐到当前生效版本，roll up表完成后可以自己导入。处于roll\_up\_finished状态的副本不能导入。完成导入任务不需要检查正在roll up表的导入情况。

提交roll up操作时允许有正在导入的任务。Roll up表生效时需要检查所有副本都追齐到当前生效版本并且没有正在进行的导入任务。

1. 导入与schema change

处于schema\_change和schema\_change\_finish的副本都可以导入。导入的过程中base表可以给新表转数据。ETL始终以元数据中生效的schema来进行处理。

提交schema change操作时允许有正在导入的任务。Schema change完成时不需要检查有没有正在导入的任务，副本的导入以生成文件中的schema hash为准，导入任何一个都会给另一个转数据。删除老schema的时候需要检查所有副本都追齐到当前生效版本并且没有正在导入的任务。

1. 导入与分裂

提交导入任务的时候不需要关心是否有分裂正在进行，dpp只生成父tablet的数据，子tablet的数据由BE在push父tablet的时候同时生成，完成导入任务时不需要检查分裂状态的tablet对应副本是否导入完成（子tablet数据是否追上父tablet由分裂过程来控制）。分裂完成后，导入任务只生成子tablet的数据，父tablet延迟删除。

分裂完成时需要检查子tablet数据是否追上，还需要检查没有正在导入的任务（如果有导入任务，那么可能只生成了父tablet数据，则需要等待本次导入完成）。

1. 导入与clone

Clone的副本如果缺版本，可以通过追数据流程来导入新数据。

1. 按条件删除与按条件删除

Loading阶段是串行的，所以同一个table family的按条件删除不能并行执行。

1. 按条件删除与roll up(schema change、分裂)

删除过程中不能做roll up、schema change、分裂，同样roll up、schema change、分裂过程中也不允许删除，否则可能会造成数据不一致。

1. 按条件删除与clone

删除的过程中允许clone，会将删除条件一起clone。如果缺版本，可以通过追数据流程来导入新数据。

1. Roll up、schema change与分裂

Roll up、schema change、分裂在BE上都是一种schema\_change的操作，相互之间是互斥的，可以通过table\_family的状态来实现互斥。完成时都是需要检查所有副本完成。

schema change完成后会存在两个schema，在删除老的schema之前不允许进行roll up或分裂。

1. 分裂与分裂

分裂完成但是base还未删除的子tablet不允许进行分裂。同一个table family下的不同tablet可以同时进行分裂。

1. Clone与roll up(schema change、分裂)

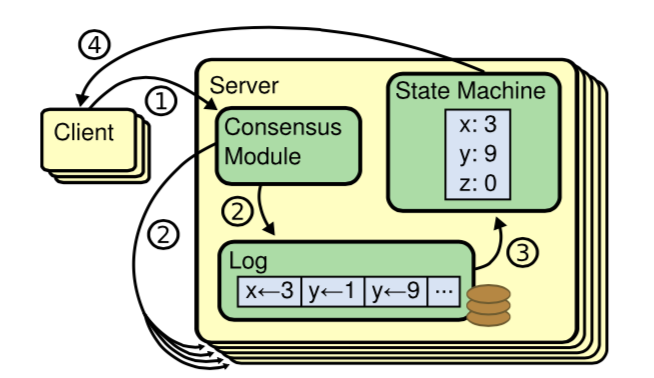
Clone时只能选择normal状态的副本进行clone。删除时不能删除roll up和分裂的base tablet。Clone完成时需要检查schema hash跟meta中当前table的是否一致（如果不一致，可能是刚做过schema change）。

Roll up、schema change、分裂时不能有clone状态的副本，而且副本必须都为online，online的副本数>=quorum的时候才能触发。

## 元数据持久化

### Replicated State Machines方案介绍

Replicated State Machines技术如下图所示：



一个cluster包含多台机器，其中有一台是leader，其余的为follower，当leader挂掉或者网络不通时，follower可以选举新的leader。每个server包含一个state machine、log和consensus module。State Machine是需要保证高可靠高可用的元数据信息；log用来存储一系列的元数据修改命令，server回放log中的元数据修改命令，将最新的元数据修改反映到state machine中。Consensus module通过一个一致性算法（比如raft或者paxos）来保证replicated log的一致性。

### 分期方案和目标

我们将该方案分2步完成：

1. Palo 2.0工作

两台frontend server，一个为主，另外一个为checkpoint节点。主节点将edit log写本地，备节点从主节点异步的拖走edit log并进行checkpoint后将最新的checkpoint回传到主节点。这种方案适合元数据较大的情况，HDFS采用的就是这种方案。

1. Palo 2.1工作

多台frontend server，一个主，多个从，使用raft协议来保证多个节点间的一致性。

### Image文件

Image存储了持久化的元数据镜像，主要包含6部分内容：header，DB信息，各种流程Job信息，和导入的标签历史，UserInfo，QueryInfo信息。 详细信息如下：

**Header**

dbCount (long)

tableFamilyCount(long)

tabletCount(long)

DBIdXOR

seralizedId (long)

**DBInfo**

DB1, DB2, …… , DBn (DataBase)

**JobInfo**

loadJobCount (int)

loadJob1, …… , loadJobm (LoadJob)

rollupJobCount (int)

rollupJob1, …… , rollupJobn (RollupJob)

schemaChangeCount (int)

scJob1, scJob2, …… , scJobk (SchemaChangeJob)

**UserInfo**

**User 权限列表**

**QueryInfo**

**QueryVersion**

tableId, version列表

**Functions**

Function列表

… 等小红整理后补充

**LabelInfo**

labelCount (long)

labelInfo1, labelInfo2, …… , labelInfom (LabelInfo)

其中DBInfo中存储了一系列的持久化的database。LoadJob / SchemaChangeJob / RollupJob是对应内存结构的持久化，在持久化过程中去除了没必要的信息，比如导入的哪些tablet已经达到了quorum，这些信息在replica report的过程中可以重新构建出来。LabelInfo存储了最近一段时间的导入标签，便于导入的时候去重。

Label

Long timeInMS

DBName

Label

State

Msg

Datasize

// 其他关注信息

在LabelInfo中我们只存储完成的（成功或者失败）的导入的label信息，正在导入的label信息在LoadJob中包含，在加载日志或者Image的时候会创建临时的label信息索引到LoadJob。

QueryInfo用来存储查询引擎相关的信息，其中queryVersion用来存储Table正在查询的版本信息，BE获取该信息后会根据这个版本信息来调整BE策略，tablet删除等操作。Functions用来存储udf/udaf等信息。

考虑到元数据的量比较少，Palo 2.0我们暂时不使用并行加载的技术，未来元数据量变大后，为了加快冷启动的速度，可以在头部记录每个部分的起始位置，然后通过多个线程并行加载每个部分。

### Editlog

每种对元数据的修改对应一条edit，每个edit包含一个OP\_CODE来表示操作类型和Writable的对象来表示该操作需要的参数信息。

Frontend修改元数据的操作流程大致如下：接受用户或者内部的请求，进行相关的处理，修改内存的元数据结构；将修改edit持久化到磁盘，fsync操作；请求返回。对于修改完内存，持久化到磁盘失败的情况，进程将停止服务退出。

Editlog顺序递增进行编号，当某个editlog的大小超过设定的阈值（比如50MB）的时候，Frontend通过rollEditlog的操作，将正在写入的editlog关闭，生成一个新的editlog。

### Checkpoint过程

checkpoint通过备节点完成，其过程如下：

1）备节点通过HTTP协议，获取主节点上最新的image和edits编号，分别为iversion, eversion。

2）备节点通过HTTP协议，获取最新的上述的image.iversion

3）备节点加载image.iversion

4）备节点依次从主节点获取edits.iversion+1 ~ edits.eversion 并依次回放到内存

5）备节点将自身内存镜像写成 image.eversion

6）备节点清空内存镜像

7）备节点将新生成image.eversion通过HTTP协议传输给主节点

### 各种流程的持久化方案

1. CreateDB

通过logCreateDB记录edits，OP\_CODE=CreateDB, DBId(long), DBName (String)

1. DropDB

通过logDropDB来记录edits，OP\_CODE=DropDB，DBName (String)。DropDB操作实际上除了修改内存外，还需要异步发送删除Task给Backend，但是这些Task不需要持久化，master切换后，可以通过replica report来获得要删除的replica。

1. CreateTable

通过logCreateTable来记录edits，OP\_CODE=CreateTable，TableCreated (Table)。创建replica的操作不需要持久化。

1. DropTable

通过logDropTable来记录edits，OP\_CODE=DropTable，TableId (long) 。删除Replica的过程和DropDB类似。

1. Load

a) 通过logLoadJobStart来记录edits，OP\_CODE=LoadJobStart, job (LoadJob)

b) 数据导入的过程中不记录状态变化，如果版本导生效前master挂掉，所有未完成的导入Job重头开始

c) 导入完成，版本生效，通过logLoadJobDone来记录edits，OP\_CODE=LoadJobDone, job (LoadJob 填充了tablet对应的导入的HDFS文件列表)，我们不记录已经导成功的replica的信息，因为这些信息可以通过backend的replica report得到。

d) 所有replica版本生效，删除hdfs上ETL中间文件，记录logLoadJobRemove来记录edits， OP\_CODE=LoadJobRemove, loadJobId (long) ，在回放该edits的时候要将对应的DB和label加入到loadLabel映射中。

e) 我们保留一段时间的loadLabel，在每次做checkpoint的时候，会将老的LoadLabel清理掉，不写到新的image中。

1. 副本管理

考虑到副本的均衡，复制和删除都可以通过replica report来获得，所以副本管理相关的元数据只保留在内存，不做持久化。

1. Rollup

通过logRollupStart记录edits，OP\_CODE=OP\_ROLLUP, rollupJob (RollupJob)。中间过程不持久化信息，frontend出问题后重新进行rollup。

当所有replica都完成rollup的时候，通过logRollupDone记录edits删除rollupJob，

OP\_CODE=RollupDone，rollupJobSeganature(String)

同时回放的时候要更改元数据状态信息。

1. SchemaChange

和Rollup类似，参考rollup。

1. 分裂

分裂不需要持久化，在分裂的过程中如果失败，本次分裂取消（分裂产生的中间数据在做replica report的时候会被要求删除）。等到下次检查分裂条件满足后，重新进行分裂操作。

### 启动流程

1. 进入safemode状态，不接受用户的请求
2. 在元数据目录找到编号最新的image
3. 加载最新的image，按编号大小顺序加载编号在其之后的edits.xx 和 edits
4. 从配置中获取所有的backend地址信息，向backend发送register RPC，将master的report地址发送给backend
5. Backend向frontend发送replica report (replicaId, tabletId, tableId, version, versionHash, schemaVersion, schemaHash等信息)
6. 对接受到的replica report信息，填充master catalog。

对于backend上存在，frontend没有元数据的加入到待删除队列

1. 当所有backend都blockreport完（或者强制命令退出），退出safemode，打开RPC Server可以接受用户请求
2. 对于各种未完成的LoadJob / RollupJob / SchemaChangeJob，重头开始进行

## Olapengine改进

### 向量化查询

2.0 impalad BE和olap engine代码合并到一个进程，不再使用impalad 和olap engine之间原有的RPC接口，impalad BE通过调用 olap engine提供的API函数。

2.0 demo版本中，impalad BE调用next\_batch函数，olap engine直接从单个delta文件中读取block信息并解压，然后传递给上层，没有多版本的merge和聚合。当时这么设计，主要是考虑到olap engine行列式的存储格式，符合向量化查询的列式存储需求，可以提高代码执行效率。

但这样的方式有严重问题，olap engine不对多个版本之间的数据做merge和聚合，会导致查询的结果不正确，比如对于聚合方式为repalce的value，olap engine就会返回多条数据给上层。

现在的大致方案改为：

1）olap engine从文件中读取block信息后，对block内的列式存储数据的相关列进行向量化操作。

2）olap engine对各版本向量化过滤后的数据进行merge和聚合。当上层下发的预聚合属性为false时，olap engine要根据full key对replace属性的value聚合。

3）olap engine每次返回一条tuple给上层。

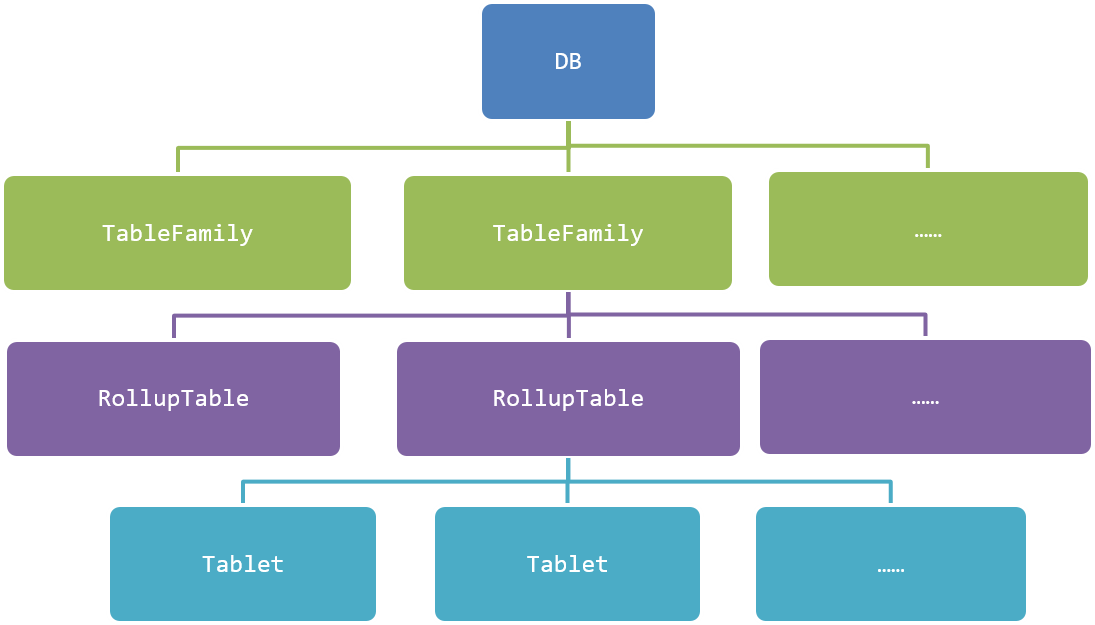
### 表目录

Palo 1.x复用OlapEngine作为单机存储引擎，并对数据组织做调整以适配OlapEngine的使用；但实际使用过程中发现OlapEngine的数据逻辑层级关系（DB/Table/Index）与Palo的数据逻辑层级关系（DB/TableFamily/RollupTable/Tablet）不完全相同，出现了部分数据层级的冗余，也不方便支持用户更改表名等新需求，故需要调整OLAPEngine的数据逻辑层级关系及使用方式；数据的逻辑层级关系主要体现在物理存储时目录结构的组织，本节会主要通过设计新的目录结构对数据的逻辑层级关系作调整。

1. 已知需求

Palo数据的逻辑层级关系如下图，新的目录结构会和该层级关系对应，且满足如下已知需求：

1. 去除冗余信息：Palo 1.x中目录结构存在废弃、多余的信息；
2. 支持表名变更：由于直接采用名称组成目录名、文件名，Palo 1.x支持表名变更困难；
3. 代码使用方便：方便BE、master等模块的使用；
4. 考虑运维：暴露一定信息，方便运维、跟进Bug。



1. 目录结构设计

Palo 1.x中目录结构组织及实际例子如下，后面逐步对该目录进行调整：

Path：【RootPath】/【DBName】/【TableName】\_【IndexName】\_【TabletId】\_【ReplicaId】/【IndexName】/【SchemaHash】/【FileName】

Example：/home/disk2/palo/data/columbus/ad\_PRIMARY\_31323\_128022/PRIMARY/693399955

File：【TableName】\_【IndexName】\_【TabletId】\_【ReplicaId】\_【IndexName】\_【SchemaHash】\_【BeginVersion】\_【EndVersion】\_【VersionHash】\_【SegId】

Example：ad\_PRIMARY\_31323\_128022\_PRIMARY\_693399955\_0\_30045\_7403827168488827452\_0

1. 去除冗余信息
2. Palo 1.x的目录结构中包含已经废弃的IndexName，而且在目录名和文件名中共出现4次，直接去除；
3. ReplicaId是冗余的：目前DM在做副本均衡时，可以保证相同副本落在不同机器上，单机不存在表名的冲突问题；如果不存储ReplicaId信息，副本Clone时不需要做额外的重命名工作；
4. 文件命名的标黑部分为Header文件名，其中TableName、TabletId、SchemaHash也是冗余的，在代码中并无特殊意义，这里仅保留TabletId，作为Header文件命名（TabletId.hdr）。
5. 文件名中的Version、Segment用于区分不同版本、不同segment的文件，需要保留；VersionHash用于导入相同版本不同VersionHash数据时的数据回滚，需要保留；

按上述描述更新后的目录结构如下：

Path：【RootPath】/【DBName】/【TableName】\_【TabletId】/【SchemaHash】

Example：/home/disk2/palo/data/columbus/ad\_31323/693399955

File：【TabletId】\_【BeginVersion】\_【EndVersion】\_【VersionHash】\_【SegId】

Example：31323\_0\_30045\_7403827168488827452\_0

1. 支持表名变更

由于直接使用Name作为目录名、文件名，导致当用户有变更表名等需求时，需要修改涉及到的元信息及所有Tablet的存储信息，实现难度大，且操作元数据时需要频繁转换Name和Id；因此做如下替换：

1. 替换DBName为DBId；
2. 替换Table\_name为TableId；

按上述描述更新后的目录结构如下：

Path：【RootPath】/【DBId】/【TableId】\_【TabletId】/【SchemaHash】

Example：/home/disk2/palo/data/8/561\_31323/693399955

File：【TabletId】\_【BeginVersion】\_【EndVersion】\_【VersionHash】\_【SegId】

Example：31323\_0\_30045\_7403827168488827452\_0

1. 代码使用方便及考虑运维

Palo 2中新版元数据采用树状结构组织元数据，为方便从树根节点DB到叶子节点Tablet的查询，需要把DBId、TableFamilyId、TableId、TabletId存放在Tablet信息中，同时把这些信息放在目录结构中也方便运维。

按上述描述更新后的目录结构如下：

Path：【RootPath】/【DBId】/【TableFamilyId】\_【TableId】\_【TabletId】/【SchemaHash】

Example：/home/disk2/palo/data/8/338\_561\_31323/693399955

File：【TabletId】\_【BeginVersion】\_【EndVersion】\_【VersionHash】\_【SegId】

Example：31323\_0\_30045\_7403827168488827452\_0

这里把TableFamilyId、TableId、TabletId打平，并不设计为层级结构，原因如下：以目前最大的产品线Holmes为例，该DB下包含近200个TableFamily，每个TableFamily下平均有2个Table，每个Table平均分配50个Tablet；如果部署在一个100台大硬盘机器的集群上，在3副本的情况下，每台机器每个DB平均包含600个Tablet，每个TableFamily平均包含3个Tablet，每个Table平均包含1.5个Tablet；可以看出DB下Tablet并不多，设计为层级结构反而显得冗余。

1. 其他考虑

存在一种考虑把OlapEngine设计成更纯粹的单机存储系统，按照这个思路设计的目录结构如下：

【RootPath】/【HashId】/【TabletId】/【SchemaHash】/【FileName】

该方案的优缺点如下：

**优点：**易于扩容，如果发现某个HashId目录下Tablet过多，可以通过新建更多HashId目录并均衡其他目录下部分Tablet到新目录，重新加载Tablet即可；

**缺点：**丢失DB/TableFamily/Table/Tablet信息，不利于Bug、统计表的Tablet分布等问题跟进，降低了系统的可维护性。

但Palo 2不考虑上述设计，原因如下：

1. 以步骤3中介绍最大产品线Holmes为例，DB下平均只有600 Tablet；如果是直连盘机器或者其他产品线，Tablet更少；
2. Palo 1.x使用类似的目录层级，目前单DB下最多400Tablet，使用正常。
3. 主要受影响流程
4. 建表

建表参数需要包含DBId、TableFamilyId、TableId、TabletId，建表时需要把这些信息添加到Header文件中。

1. 启动加载

OlapEngine在启动时遍历数据目录加载所有Tablet，放入TabletId到Tablet的map中，由于可能正在进行SchemaChange，同一Tablet下可能包含多个Tablet，故数据结构如下：

std::map<int, std::list<OLAPTable>> table\_map; // TabletId to Tablets

1. 获取表

通过TabletId获得相应Tablet，包括查询、导入等基本所有代码。

### 代码清理

由于下面原因导致Palo 2 BE中存在冗余代码，需要清理代码，保证BE代码清晰易读、健壮高效：

1. Palo2计划去除多余包依赖，需要梳理依赖包的使用情况。
2. Palo 1.x为加快开发进度，复用OlapEngine作为单机存储引擎，但由于两者数据层级关系不完全相同，导致BE内部存在冗余代码；
3. Impala-Be与OlapEngine合并后，通过内部接口直接通信，网络通信及请求解析相关代码被废弃；
4. 清理依赖包

Palo 2 Demo中可清理的依赖包如下：

1. bmock: 影响主要模块的测试代码；
2. libxml：影响CREATE\_TABLE、CREATE\_ROLLUP\_TABLE、ALTER\_TABLE、SPLIT\_TABLE接口；
3. configure：已经替换使用gflag，可以直接清理；
4. bsl、ullib：使用很少，需要编译定位依赖处；
5. 废弃接口

Palo 1.x中可废弃的代码如下：

1. command\_executor请求参数解析接口；
2. fetch\_more接口；
3. cumulative version hash校验接口；

### 数据正确性校验

Palo 2设计可扩展至百台以上的集群规模，并为百度众多重要产品线提供报表及分析服务；这使得原本单机小概率的硬件或软件问题都变得有可能发生，且不可忽视；然而分布式系统在运行中，不可避免有异常发生，就会导致在读取或存储时出现脏数据。

脏数据通常比较容易出现在系统对数据的处理过程中，比如Palo 2中的数据导入过程、数据合并过程（BE & CE）、SchemaChange过程等；当然，磁盘问题等非系统处理过程也可能导致脏数据的产生；Palo 2会考虑Palo 1.x使用经验，从online和offline两方面对数据正确性做校验，尽早、主动发现脏数据，减少对用户的影响：

**Online校验：**系统在导入、查询、数据合并、SchemaChange等全部数据处理过程，统计数据行数变化做校验；这种方案轻量，对线上服务基本无影响，且可以尽早发现程序Bug等问题，但不能覆盖全部数据不一致问题。

**Offline校验：**利用多副本在数据归并后的一致性，进行副本之间数据正确性的校验；这种方案会扫描全部数据，开销很大，故系统会在低峰期扫描副本数据做副本间校验，并控制并发度和速度，降低扫描过程对线上服务的影响。

结合上述两种方式，基本可以做到尽早发现脏数据、覆盖全部问题，并降低对线上服务的影响；Palo 2中的数据正确性校验策略总体如下图：



1. Online校验
2. 导入校验

导入过程需要从HDFS下载数据到本地，并写入对应Tablet；该过程容易引入脏数据，校验过程如下（Palo 1.x中已实现）：

* + 1. DPP在写入本地文件时，累积生成Tablet数据的Checksum并写入文件中；
    2. BE在导入过程中同样生成一份Checksum，并与DPP生成的Checksum进行对比；如果不同，重新进行ETL过程生成数据导入即可。

1. 查询校验

查询时校验过程如下（Palo 1.x中已实现）：

1. 写入数据时，针对每个Block记录Checksum信息；
2. 查询读取数据Block时，重新计算Block的Checksum值，并与记录的历史Checksum进行对比；如果不同，则报告存在脏数据，且本次查询失败。
3. 数据合并校验

由于数据合并过程中可能存在数据删除，数据合并的校验过程如下：

1. 获取数据合并过程依赖的全部Delta，并读取数据文件Header中记录的行数S1…Sn;
2. 在数据合并（BE、CE）过程中，统计新生成Delta的行数N、被合并的行数M、并过滤的行数F；
   * + 1. 如果N+M+F等于S1+…+Sn，则初步认为合并过程无问题，并把N记录于新Delta数据文件Header中；
       2. 否则，报告存在脏数据，且本次数据合并失败。
3. SchemaChange校验

对于Link型SchemaChange，不做校验；对于非Link型SchemaChange，统计新生成Delta的行数N、被合并的行数M，校验逻辑类似数据合并过程。

1. Rollup校验

统计新生成Delta的数据行数N、被合并的行数M，校验逻辑类似数据合并过程。

1. 分裂校验

统计新生成Delta的数据行数、被过滤的行数F，校验逻辑类似数据合并过程。

1. Offline校验

在指定版本的条件下，同一Tablet的数据归并后是一致，故利用归并结果的Checksum做多版本间的检验；具体流程如下：

1. FE在集群低峰期，遍历所有Tablet并生成计算Checksum的任务，分发给BE处理；
2. 低峰期检验主要考虑扫描Tablet全部数据会给系统带来较大压力，低峰期可配置，默认为22点至8点；
3. FE分发Checksum任务时，需要控制并发度，并发度可配置，默认为1个；
4. BE接收到Checksum任务后，调用接口计算Checksum值，完成后汇报给FE；
5. BE在计算Checksum值时进行限速，限速可配置，默认为5M/s;
6. 当Checksum计算失败时，BE负责进行重试，多次重试仍然失败，则返回失败；
7. FE接收BE的汇报，并更新任务状态；当Tablet所有副本均完成Checksum任务时，校验多副本的Checksum值：
   1. 如果副本之间完全一致，则认为数据正常；
   2. 如果排除计算失败的任务后副本间一致，也认为数据正常，但在日志中记录下这种情况；这种现象一般为机器宕机、副本均衡导致，校验过程中出现概率较大但通常无问题；
   3. 否则副本可能已损坏，由于副本不一致通常由Bug导致，且不能简单认为某一副本不一致即该副本出现问题（可能副本均衡复制了问题副本），故前期策略为打印日志触发报警，并跟进发现潜在问题。

### VARCHAR索引优化

2.0针对索引存的优化主要在以下几方面：

1）在short key中记录varchar所有字段。在当前short key中，varchr索引不是全部记录，是截断的，因此当short key中包含的varchar处在中间位置时，就会导致使用short key查询时出问题。

2）增加delta pruning。根据mesa的论文，此种方式对于以时间为第一列的存储，效果提升很明显，时间越新的数据，访问越频繁。

3）对short key使用LRU，以delta为单位进行缓存。目前所有的index都是常驻内存的，随着表的增大，会占用过多内存，影响OS的cache，进而影响查询。

1. 索引结构

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Entry1（k1 k2(OFFSET) k3） | | | | |
| Entry2（k1 k2(OFFSET) k3） | | | | |
| ………………………… | | | | |
| EntryN（k1 k2(OFFSET) k3） | | | | |
| LEN+DATA | LEN+DATA | LEN+DATA | LEN+DATA | ………… |

1. 索引创建

1）每生成一条short key entry，写入文件，对于varchar字段，记录的是offset，varchar内容在内存中缓存。（为了避免缓存的varchar过大，可以调节segment的大小）

2）每个segment Index文件最后序列化时，将所有缓存的varchar内容，最加到文件末尾。

3）版本对应的data文件和index文件生成后，在头文件中记录版本文件对应的delta pruning信息。

1. 索引读取

1）将delta的Index加载到内存，并放入LRU中

2）读取每条entry时，对于varchar字段，要根据offset获取varchar内容。

1. 索引查找流程

索引主要是为了准确定位start/end key的位置，读取start/end key范围内的信息。定位 start/end key位置，首先通过索引信息定位到相关block位置信息，再读取block信息，比较full key，准确定位位置。对索引查找流程的优化，有助于提高查询的效率。

**行列存**



**列存**



1. 其他说明

使用列存的index stream，可以达到short key的效果，能取代short key。

首先阐明下short key的用途。Short key作为稀疏索引，主要作用是定位包含start/end key的block位置范围。short key的长度一般都start/end key短，因此无法精准start/end key所在的block，还需要通过读取文件获得full key进行比较才能定位到具体的block位置。如果short key的长度>=start/end key，则可以准确定位到某个block。也就是说，short key越长，则定位block越准确。

使用列的min max信息，是能够找到start/end key所在block的。流程如下：

1）key1是严格有序的，通过二分查找start/end key1，可以定位到一个block范围[>=key1的第一个block，>key1的第一个block]；

2）在获得的key1 block范围内，再使用二分查找start/end key2，再次获得一个block范围；

3）一次重复，直到找到start/end key中最后一个key范围。在此范围中，根据range/end range值，读取block的每行内容并进行比较，获取准确的start/end key行。

如下图所示，start key=3,4,5 查找流程如下：



### Be和cumulative优化

BE和cumulative目前遇到的主要问题有：

1）BE时，旧的base文件和要合并进来的cumulative或delta文件，要重新读写一遍进行聚合等操作，造成大量IO，影响查询。以useid为第一列的表，无法避免大文件合并。

2）在BE或cumulative时，可以对不同版本文件的value进行聚合，但对于那些只存在于单个版本的key数据（比如base中较老的数据），没有数据可以聚合，重读写这些数据只会造成IO的浪费。

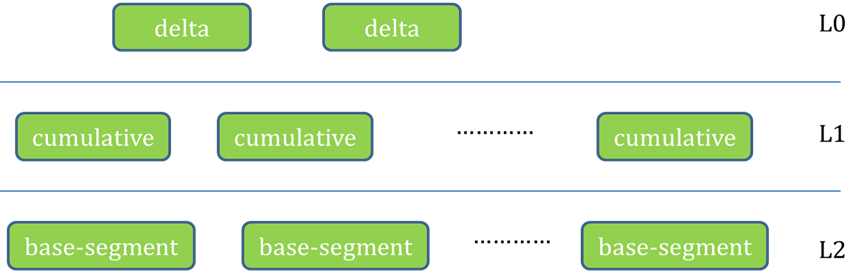
设计如下两种方案。

1. 分层

此方案在逻辑上采用类似levelDB分层的结构，将delta、cumulative和base当做不同的层次来处理。

**文件结构**

采用类似levelDB的结构，将文件分为L0 L1 L2三个层次：



1） L0存放delta版本，向下合并到L1的cumulative版本中。

2） L1存放cumulative版本，向下合并到L2的base版本中。

3）L2存放base版本。

**合并流程**

本方案分为两个层次的合并，两个层次的合并分别由两个不同的线程控制，分别进行合并。对于以时间为第一列的表，各层的数据向上合并时，可直接追加在上层的末尾；而以useid为第一列的表，则必须进行多版本文件的归并操作。

* L0->L1

触发策略：

周期轮训各个表，可以被合并的delta（通过master获取）个数达到阈值（默认4）时，启动L0向L1合并的操作。

合并策略：

1）delta的总大小大于阈值（默认50M）时，生成一个新的cumulative

1）L1层中版本最新的cumulative size小于某阈值（默认50M）时，将 delta与此cumulative版本进行合并，否则这批delta生成一个新的cumulatvie版本。

2）delete delta和size大于阈值（默认100M）的delta，不能进行cumulative合并，直接添加到L1层

策略参数：

|  |  |
| --- | --- |
| trigger\_to\_create\_cum | 可以被合并的Delta数目超过这个值时，触发Cumulative |
| Max\_file\_num | 一次cumulative可以被合并的最大delta数目 |
| max\_file\_size | 当delta大于此值时，不允许将此delta进行cumulative。 |
| max\_cumulative\_size | 当要合并的delta总大小大于此值时，生成新的cumulative；当cumulative大小大于此值时，不能和delta合并 |

* L1->L2

触发策略：

1）delete版本必须满足超时时间才可以被合并

2）L1层中，某个未超时delete版本之前的cumulative版本总size与L2层的size比值大于阈值（默认10%）时，启动L1向L2合并的操作。

3）如果cumulative版本size不满足要求，但是包含超时的delete版本，此时启动L1向L2的合并操作。

4）提供手动功能，下发命令，合并当前可以合并的cumulative版本。

合并策略：

将L1中的所有cumulative版本合并到L2的base中。

策略参数：

|  |  |
| --- | --- |
| ratio\_trigger\_to\_create\_be | 控制产生BE的触法条件，当cumulative版本size与base size比值大于这个值时，触发BE |
| delete\_delta\_expire\_time | Delete版本被合入BE的缓冲时间 |

**方案总结**

优点：

1）将原本的delta、cumulative和base，映射到了类似levelDB的多个层次，在方案上没有做较大调整，复用原来的代码，修改量较小。

缺点：

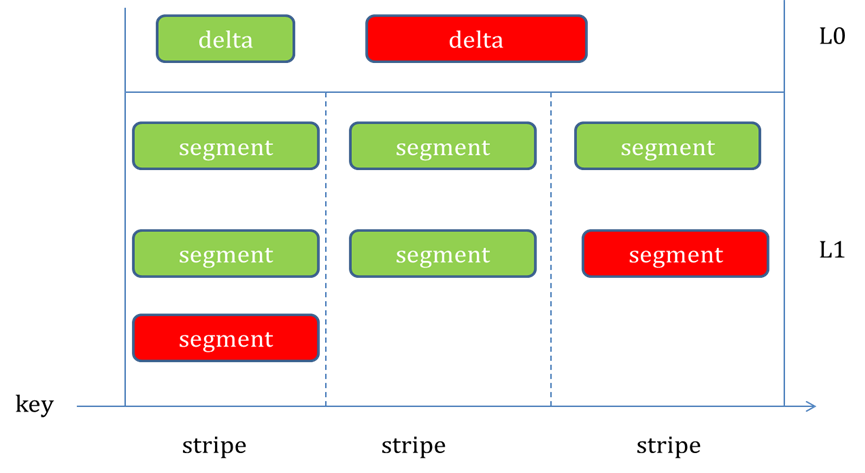
对于不以时间为第一列的表，key的重复较高，无法根本解决文件合并时IO量大的情况。

1. Stripe+分层

该方案参考HBASE-7667，主要是根据表的key range，划分了多个sub range，每个sub range即为一个stripe。表内部文件的分布结构，类似于levelDB，分为L0层和L1层。所有的delta版本位于L0层，L1层存储base版本，不同key range的segment属于不同的stripe。

**文件结构**

文件结构如下图所示。



当delta满足条件可以被合并，将delta根据sub range划分，写入不同的stripe（如上图红色部分），再在各个stripe内部进行文件的归并和聚合。

**合并流程**

触发策略：

1）delete版本，只有超时后才能被合入L1层

2）小于查询指定版本号的delta，并且不存在小于该delta版本号的delete版本，可以被合入L1

合并策略：

1）被合并的delta根据sub range的划分，写入不同的stripe，生成segment文件。

2）在stripe内部，判断segment之间是否有key重叠，有重叠则进行segment的合并。

3） stripe之间的平衡操作。当某个stripe的文件过大，可以和相邻的stripe进行合并，或者分裂成两个，调整stripe的key range。

**方案总结**

优点：

1）查询时，只要读取对应key range的stripes和L0文件即可，

2）在各个stripe内部，根据需要分别进行文件的归并和聚合，减少了IO操作。

3）delete版本，可只在单个stripe进行合并

缺点：

1）相对于方案一，修改了代码的结构，工作量较多。增加stripe信息，每个segment要增加min max信息等。

2）对于以useid为第一列的情况，合并一个delta，会在每个stripe中产生segment，此segment在单个stripe可能和所有segment都有key重叠，则需要stripe所有文件重新归并，依然需要大IO操作。

1. 方案选择

使用方案一。方案二并不能彻底解决一useid为第一列所造成的大IO操作，且代码修改量较大。

### Schema change优化

由于涉及到的修改模块较多，一期版本暂缓实现，本次涉及只给出大致思路。

当用户需要修改表结构而进行schema change时，当前的代码实现，是通过读取表的内容，按行增加、删除字段或调整字段顺序。当表较大时，这样对CPU、IO和内存的消耗很大，影响查询的性能。

但是对于某些情况，比如正对某个table family只增加某个value列时，可以对原表的文件创建硬链接，查询时提供新列的默认值，在BE时再对表文件进行修改，这样可以节省schema change的大量资源，这种方式称为linked schema change。

满足下列情况下使用linked schema change：

1）只修改value列时，可使用，包括增加或删除value列

2）不修改key列顺序时，可使用，包括增加或删除尾部key列。

3）在key列中间插入新列时，可使用。

4）改变key列顺序时，禁止使用，比如删除非尾部的key列，调换key列顺序。

5）此表由linked schema change创建，且还有版本没有经过BE转为新的schema，则禁止使用。

## Agent改进

### 目的

Agent模块原本是palo1.x项目中的一个独立部署模块，部署在数据节点上。主要作用为轮询DM模块对指定数据节点相关的元数据的修改，将对元数据的修改转化为对应的作业同步到数据节点上。其他功能主要是包括心跳、汇报单机资源状况、回收单机无效存储资源等功能。

palo1.x单独部署模块较多，对管理和维护都带来了成本。

在palo2.0版本中，元数据将由在mysql中进行存储改为在DM模块的内存中进行管理，如果采用之前的轮询方式，随着集群节点的增加，元数据的访问压力将随着增长。

所以此次palo2.0版本的Agent模块在功能上与之前相同的情况下，总体变化如下：

1. 与数据节点上的其他进程合并，不再进行单独的部署。与本地存储引擎不再使用socket通信，直接调用内部接口。
2. 不再轮询元数据修改，改为使用thrift与DM进行通信。DM模块主动通知Agent要进行的作业，Agent也会直接向DM汇报作业结果以及数据节点状态等信息。

### 目标

本期需要达到的目标如下：

1、Agent模块与olap engine合并入一个进程。采用内部接口进行调用。

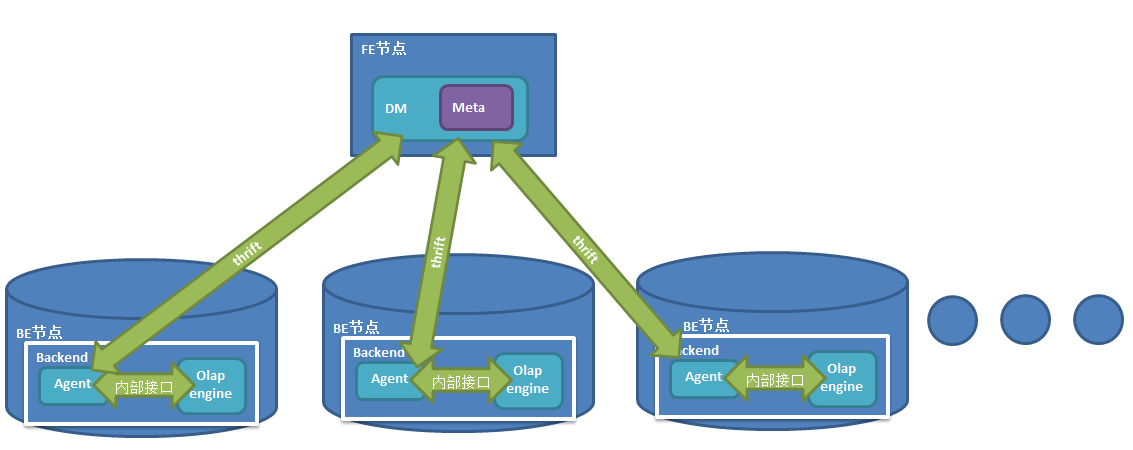
2、DM模块与Agent模块之间采用thrift进行主动通信。

3、实现palo1.0中的create table、drop table、push、create rollup、schema change、split、clone等针对olap engine的作业接口。由于DM与Agent之间都采用主动通信的方式，为了避免消息重复发送，要求所有作业接口的实现必须是幂等，均可以重复调用。

4、DM会主动与Agent的heart beat thrift server通信，agent返回表示自己存活。同时也要定期向DM汇报本机磁盘存储状态以及olap engine表的存储状态等。

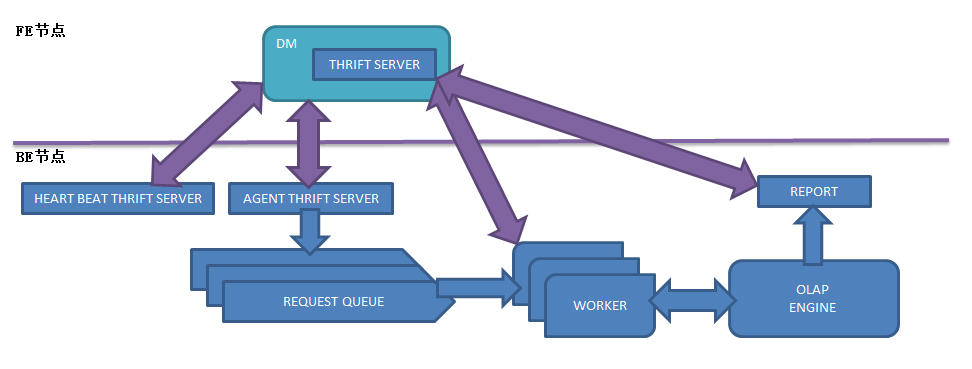
5、Agent模块均不持久化作业相关的消息。为了防止DM或Agent宕机或其他原因造成的消息丢失，在Agent启动及运行期间，需要和DM同步作业信息。

### 架构图



如上图所示，Agent模块与olap engine运行于同一进程内，使用内部接口进行交互，一同被部署在数据节点上。Agent模块与DM使用thrift进行通信。数据节点可以进行横向扩展。

### 主要逻辑



DM通过访问各Agent节点的agent thrift server将作业下发。

Agent接收到作业后，根据作业类型放入不同的队列中。队列根据作业性质进行分组，比如时效性要求高的create table、push等作业有单独队列。create rollup、schema change、clone等时效性要求不高且要限制单机同时进行个数的作业可以使用同一个队列。放入队列后就向DM返回消息，表示已接收到作业。

之后每个队列会有数个对应的worker负责处理，按照作业类型和参数与olap engine交互。作业完成后，worker会尝试自己与DM的thrift server通信，将结果进行汇报。如果失败，进行几次重试，重试失败后放弃汇报。之后BE汇报task列表时，FE会发现task缺失并重新发送task给BE，由于操作是幂等的，BE之后会再次尝试汇报结果给FE。

一个独立线程会汇报当前接收到的作业列表情况给DM，同时汇报线程也会获取olap engine的表情况和磁盘存储情况，对DM进行汇报。

Agent有一个独立的heart beat thrift server，由DM主动发送请求，agent接收后返回表明自己存活。

## 其他流程

### Dpp数据删除

FE检查quorum完成的导入任务所有副本都完成时，即可触发dpp数据的删除。

对于磁盘坏掉的情况，修复后重新加入时，需要检查副本的版本与table family生效版本之间的数据是否已经被删除。如果已删除，会造成数据无法追上。如果重新加入时，副本数已经是满足的，则直接删掉。

### 同步磁盘状态

BE定期向FE同步tablet信息、磁盘状态和各类任务的状态。

### BE已删除数据的清理

BE删除表时将schema hash目录改为.delete.1260304014.20141028092509，BE需要定期清理这类目录及上层相关的空目录。

## 测试重点

回归1.x的所有case

FE和BE之间信息交互有无异常

各流程执行有无异常

## 实现排期

元数据持久化部分大致排期如下：

1. 2014.12.1 ~ 2015.1.1 开发完成
2. 2015.3.1 ~ 2015.5.1 开发完成

FE和BE agent部分：

2014.11.10~2014.11.23 整理demo代码，完善框架（2周）

2014.11.24~2014.12.14完善建表、导入、roll up流程，开发按条件删除（3周）

2014.12.15~2015.1.18 副本迁移、schema change、分裂开发（5周）

2015.1.19~年底 其他流程开发、修bug、优化（4周）

BE olapengine部分：

2014.11.10~2014.11.23 整理代码部分，先不包含单测（2周）

2014.11.24~2014.12.14 目录结构、向量化（3周）

2014.12.15~2015.01.04 向量化、checksum online(3周)

2015.01.05~2015.01.25 单测移植、checksum offline、BE优化（3周）

剩余时间主要用来配合测试修Bug、varchar优化设计、SchemaChange优化设计等