**MemTable事务优化方案**

[当前方案介绍](#h.mboez4bc4rcg)

[整体结构](#h.4pl9gyhfgz61)

[写事务设计](#h.ijvggrqw4d7z)

[批处理设计](#h.2zct515c7yl7)

[读事务隔离性](#h.5htcdl2zur33)

[优化方案设计](#h.runxww9sk603)

[Hash查询优化](#h.y7e2nfkbdwaw)

[内存使用优化](#h.5a2rmi9786lr)

[读写事务设计](#h.jn10pkiafg2g)

[批处理与事务](#h.eaw710z5eq7n)

[几种特殊情况的处理](#h.pbwfqhqlmkor)

[删除行操作](#h.d7348iy18tp)

[行内数据合并操作](#h.hyv10db7w2e3)

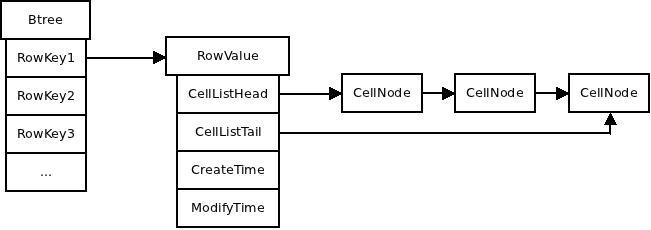
[写事务回滚操作](#h.jk5m1vglue9d)

[行不存在的情况](#h.ss9q3htm5esb)

# 当前方案介绍

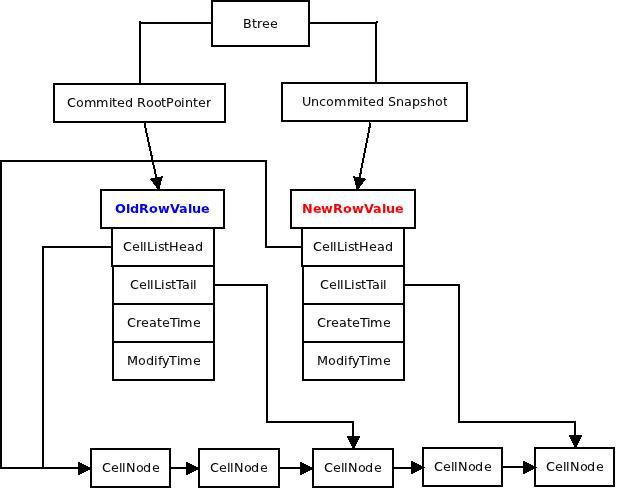
## 整体结构

如下图所示，在memtable现有的实现中，行数据通过Btree索引，对列数据的更新保存在CellNode结构中，使用链表串连起来，并在btree中保存链表的头尾指针，读取这一行的数据时，遍历链表读取所有cell的数据。



## 写事务设计

使用Btree的copy on write特性实现事务的隔离性和原子性，一个事务内所有的行的更新在btree的一个快照点上执行，btree提供原子的快照点提交或删除操作，用于实现事务的提交或回滚。如下图所示，一行内列数据的修改操作直接加到上述链表的尾部，然后修改RowValue中的链表尾指针等其他元数据，读取时遍历链表到尾指针指向的node为止，对于未提交的事务数据，虽然已经接到链表尾部，但还没修改尾指针，所以不会被读出。由于btree并不管理内存，因此btree中RowValue以指针形式保存，每次对行数据的修改都构造一个新的RowValue来替换原来的指针，而旧的RowValue由于没有引用计数机制所以不能删除重用。



## 批处理设计

使用批处理的方式提高TPS，一次批处理中包含的多次事务日志一次性提交，日志提交成功后再将整个批处理提交可见。因此要保证批处理整体的原子性和隔离性，而单个事务只要可以单独回滚即可。即实现在未提交的批处理中，已提交的事务可见，未提交的事务可回滚；而整个批处理可以原子提交或回滚。

在现有的实现中，使用btree的提供的多级copy on write特性实现，即在开启了一个快照点A的基础上，还可以再开启新的快照点B，在B快照点上的修改可以提交或回滚到快照点A，将快照点A提交后外部才可以见到过程中的所有已提交修改。

举例说明如下：

1、memtable保存有行A，B，C，读线程可读到A，B，C

2、开启一次批处理

3、开启一次事务

4、写入行D

5、提交事务

6、读线程可读到A，B，C，写线程可读到D

7、开启一次事务

8、写入行E

9、回滚事务

10，读线程可读到A，B，C，写线程可读到D

11、提交批处理

12，读线程可读到A，B，C，D

## 读事务隔离性

需要保证在处理一次读请求的过程中也满足读事务的隔离性，即在读事务执行的过程中只能读取到读事务开始那一刻的数据，之后的更新即使已经提交也要保证不能被读取到。在现有的实现中使用Btree的copy on write特性，在读事务开始时获取当前btree中已被提交的数据的快照，执行这个读事务时都从这个快照点中读取数据，而后续的数据更新不影响这个快照点。

# 

# 优化方案设计

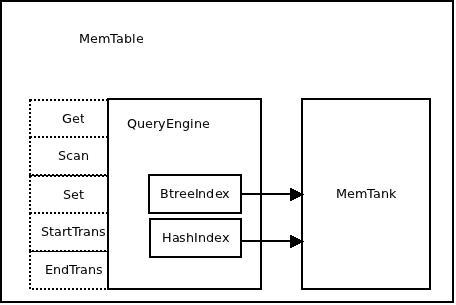
本次memtable优化方案主要对两个方面进行优化，包括使用Hash优化单行get查询，和优化memtable内存使用。

## Hash查询优化

对于单行get查询的需求，使用btree索引行数据，对1000w行数据进行二分查找，最坏要付出23次rowkey比较和内存随机寻址的代价，并且在执行写操作对行数据进行更新的时候也要先读出行数据后再执行更新。因此使用hash索引行数据，将单行数据查询的复杂读优化到O(1)，不仅可以提高multi-get的性能，也能提高写入速度。

使用Hash优化的实现方式比较简单，如下图所示的memtable结构，增加一个与Btree结构平级的HashMap，保存Rowkey到RowValue的映射，其中RowKey和RowValue都以指针形式保存，实际数据空间由memtable维护。将Btree和HashMap封装到一个名为QueryEngine的结构，这个engine结构为memtable提供set/scan/get接口，和开启/结束批处理和事务的接口。读写事务的原子性和隔离性，在QueryEngine层处理，不再依赖btree实现，hashmap和btree今后只作为最简单的随机查询和顺序查询的索引使用。QueryEngine实现读写事务的方案将在下一节说明。

HashMap使用简单的定长方式实现，在初始化时确定桶个数，之后不能修改，使用链表法解决hash冲突。根据已有的测试结果，选择murmurhash作为默认的hash函数。



## 内存使用优化

如上一章所述，在现有的实现方案中，行每更新一次就要构造一个新的RowValue结构写入btree，而在新的优化方案中，由于hash和btree只是用来做get和scan的索引，不再处理读写事务逻辑，因此可以考虑不重新构造RowValue结构的更新方式。

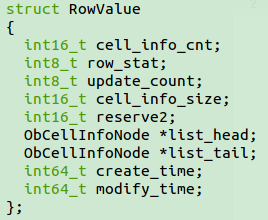
如下图所示，目前RowValue中存储的数据包括，列cell链表头尾指针，create\_time和 modify\_time，行状态标志row\_stat，cell个数等统计字段。其中：

1、create\_time在行构建的时候被写入，之后不会修改

2、除create\_time和list\_head之外的字段在行每次更新时会被修改

3、读事务中会先读取row\_stat，确认状态后从list\_head遍历读取到list\_tail，并根据需要构造create\_time/modify\_time的cell输出

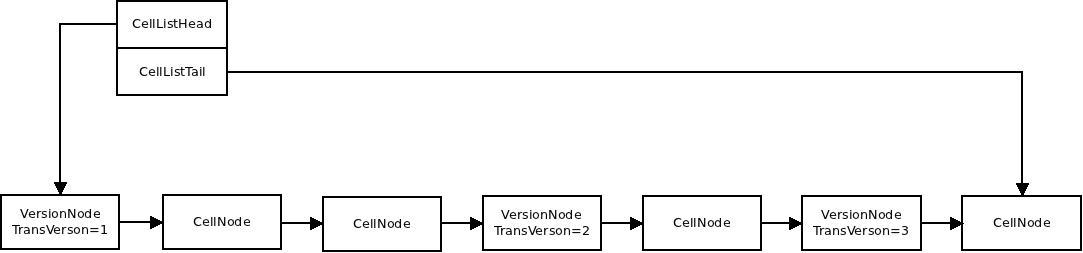
4、写事务中出现链表过长时会进行合并，合并后需要修改list\_head/list\_tail和统计字段



### 读写事务设计

使用事务版本号来实现读写事务的原子性和隔离性，每个写事务拥有一个全局递增的版本号(主备不需要同步这个版本号，各自维护)，在写入的数据中保存所在事务的版本号，QueryEngine中保存当前已提交的最大的版本号。开启读事务时，先获取这个当前已提交的最大版本号，读取数据时只读出小于或等于这个版本号的数据。因为cell数据是按时间顺序追加到链表中的，并且已接入链表的CellNode不会被更改或删除，因此我们可以利用这个特性实现在读事务中读取历史版本的需求。

如下图所示，每次修改行数据时，在链表中先写入一个特殊的CellNode，用于表示它后面的CellNode所在写事务的版本号(下面称为VersionNode)，然后将需要写入的CellNode追加到链表尾部，修改链表尾指针，最后修改行状态信息和统计字段。执行读事务时，先判断行状态信息是否表示为行存在(有数据或行被删除都算做行存在)，如果存在则从链表头遍历到链表尾，当遇到VersionNode且它保存的事务版本号大于当前写事务的版本号时停止遍历。



因此在这种设计中RowValue结构不需要被整体的原子性更新，更新行数据的时候不需要构造新的RowValue结构(省去了每次更新都浪费40个字节的内存占用)，而是找到已有的RowValue的指针，直接修改里面的字段即可，在与读写事务并发的情况下：

1、多数情况写事务不修改list\_head指针，cell\_node直接被追加到链表尾部，保证读事务从list\_head遍历的到链表结束或指定的VersionNode是安全的

2、CellNode先被追加到链表尾部，才修改row\_stat状态标志，因此读事务先判断row\_stat标志后保证能后安全的从list\_head开始遍历链表

3、只有在保证新的链表构造好后，才修改list\_head指针，通过新旧list\_head遍历得到的最后数据是一致的

为了RowValue结构中的空间，使用modify\_time当作事务id记录到VersionNode中，因此可以去掉RowValue中的modify\_time字段。为了在系统时钟调整时保证事务id自增，需要在产生新的事务id时判断是否大于上一次事务id，否则将上一次的事务id加1后作为本次事务id。备机回放日志时所需要的事务id从主机传来的事务modify\_time获取。

### 批处理与事务

如第一大部分所述，多个事务的批处理需要保证原子性和隔离性，而批处理内的单个事务可在批处理内实现提交或回滚。因此我们将每个批处理当作一次写事务，分配一个事务id，更新当前已提交事务id的值，即实现了批处理的原子提交。将批处理内的子事务当二级事务，通过记录子事务和批处理中涉及到的行的修改操作，可以实现单个事务或整个批处理的回滚。

### 几种特殊情况的处理

#### 删除行操作

在现有的实现中，由于ObMerger的局限，删除行操作需要先清空已有的链表，然后再加入表示删除行的CellNode，在优化方案中，我们不使用这种对删除行操作的特殊处理，而是当作普通更新一样接入到链表尾部，在读取的时候利用已有的RowCompaction工具类对迭代出的结果进行整理以满足ObMerger对输入数据的要求。

#### 行内数据合并操作

当链表中的CellNode个数过多时，需要对统一列的数据更新进行合并，构造新的CellNode进行合并操作，不对已有的CellNode进行修改。需要注意的是，合并后需要对list\_head和list\_tail进行修改，对这两个指针非原子的修改将导致一段时间内的list\_head和list\_tail分别指向新旧链表，但是由于读取过程中从list\_head遍历到链表结束或指定的VersionNode即可，不需要以list\_tail做结束标志，因此这里的非原子修改对读事务不造成影响，因此行数据的合并操作可以在写事务的过程中执行。

对行内数据的合并会将众多历史更新版本合并为一个版本，但是需要保证合并前已经开始的读事务能够读取到正确的历史版本，而一旦修改list\_head之后，历史版本的数据虽然存在但是已经不能被继续访问到了，因此执行合并的时候需要进行判断，只能将当前正在执行的读事务版本号之前的数据进行合并。举例如下时序操作：

1、当前已提交的事务版本为10，行A的链表中包含了版本1到版本10的更新数据

2、开启一次读事务，因此读事务id为10

3、对行A执行10次写事务，已提交的事务版本变为20，判断行A需要合并

4、对行A进行合并，由于当前版本为10的读事务未结束，因此合并时只能将版本1到版本10的数据进行合并，版本10及之后的数据不能合并

5、版本为10的读事务读取行A，能后读到版本1到版本10的修改

使用无锁方式，获取多个当前正在执行的读事务的最小版本号：使用一个全局数组，每个数组元素对应一个读线程，读事务开始时将事务版本写入当前线程对应的那个数组元素，读事务结束时将对应的元素值置为无效，要执行行数据合并时，先遍历这个数组找到最小的事务id，这个事务id之前的数据可以合并，大于这个事务id的修改不能合并。

需要注意的是，由于可能存在遍历链表时没有找到已开始的读事务id，但是遍历完成后有读事务开始的情况，由于行数据的合并操作在写事务的过程中执行，因此需要注意对正在执行的写事务中写入的数据不能进行合并。

#### 写事务回滚操作

事务和批处理操作结束后可能被回滚到事务开始前的状态，因此在写事务执行过程中保存被修改过的RowValue之前的内容和执行事务后RowValue结构的指针在一个容器中，在事务提交时清空这个容器；或回滚时，遍历这个容器获取这些RowValue，将指针指向的内容回滚到之前保存的值。需要注意的是需要先回滚行状态标志row\_stat，然后再依次回滚list\_tail和统计字段值。

对于在事务中插入新行的情况，回滚时没有必要从HashIndex和BtreeIndex中删除索引，只需要将row\_stat状态置为行不存在即可。对于事务内有行内数据合并的情况，由于合并的数据仅限当前事务之前，因此不需要特殊处理，直接回滚list\_tail即可。

此外，因为读事务读到更高版本的VersionNode就会终止读取，这一特性使得被回滚的事务所占内存空间可以在回滚后被重用，而没有对读事务的引用计数的要求。在支持条件更新功能之前，可以暂时不实现这部分内存重用逻辑。

#### 行不存在的情况

除了行在BtreeIndex或HashIndex中查不到的情况之外，还可能存在如下几种情况，读取时能够从BtreeIndex或HashIndex获取到RowValue指针，但是读取结果应该为行不存在：

1、新插入的行所在事务尚未提交，RowValue的版本号大于读事务版本号

2、新插入的行所在事务被回滚，RowValue内容为初始化的值