MyRocks: LSM-Tree Database Storage Engine Serving Facebook's Social Graph

http://www.vldb.org/pvldb/vol13/p3217-matsunobu.pdf

Abstract

基于B+Tree的InnoDB瓶颈：

1.叶子的合并分裂操作带来的碎片问题，造成空间浪费

2.随机写的时候，写放大问题严重，假定一个 BTree 结点保存了 100 条数据，如果我们修改了一条，那么当这个结点要写回磁盘/SSD 时，整个结点（对应的页）都要写回，写放大就是 100 倍！

而LSM Tree可以有效解决上述问题。MyRocks则是基于RocksDB开发的MySQL存储引擎。

Introduction

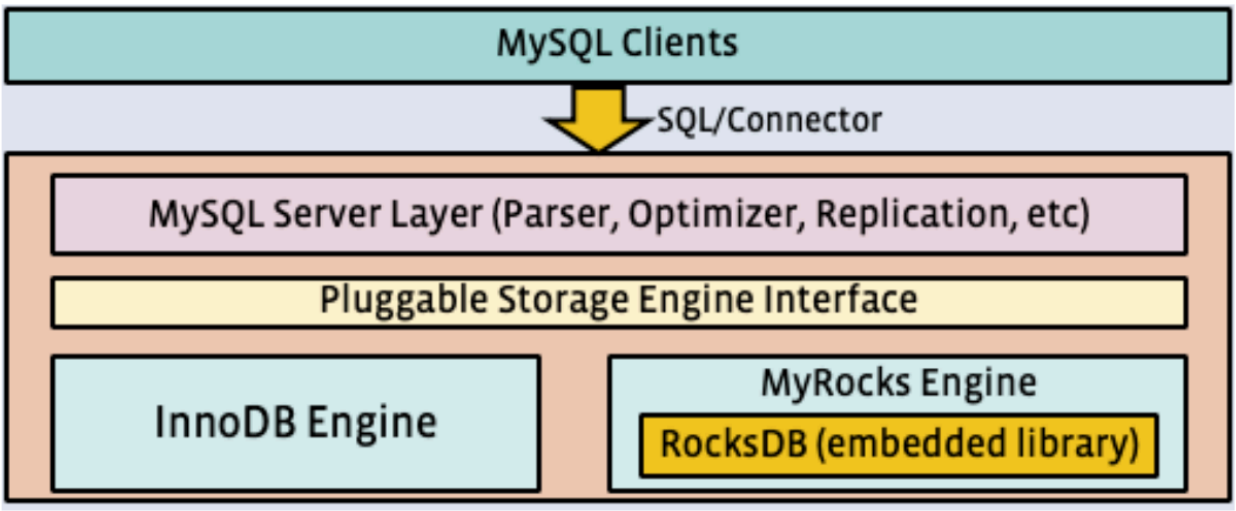
随着硬件的进步，存储设备的趋势是从速度慢但是便宜的磁盘存储到速度快但是昂贵的闪存。Facebook内部的DB系统的瓶颈因此也从IOPS变成存储容量。从空间角度来看，InnoDB主要有三大瓶颈:

1.B+tree的索引问题

2.压缩的低效率

3.处理事务的时候需要为每一行记录额外维护13bytes(MVCC)

因此fb的研发团队把RocksDB这么一个基于LSM-Tree的KV storage进行二次开发，以存储层插件形式开发出"MyRocks"作为MySQL的存储引擎，不需要对解析层，优化器层做任何改动。



技术挑战

1.CPU,Memory和IO压力

MyRocks相比于Innodb，能把数据规模减半。压缩是个CPU-Intensive的过程，并且数据量的减少意味着我们可以单位时间传输更多吞吐量的数据，导致IO压力也会变大。

2.Key Comparasions

LSM的key 比较的次数会比B+树更加的频繁

3.LSM往往需要in-memory的布隆过滤器来提高读性能，舒缓读放大。

不难想象这会对内存额外造成较大压力

4.Compaction带来的写放大

如果写入速率过高，Compaction很可能会给CPU和IO造成短时间的压力。

5.Tombstone的管理

LSM Tree的删除操作是通过增加marker来标记该记录是否被删除的。那么意味着更新和删除操作效率难免会受到影响。

创新之处

针对以上提到的问题，MyRocks提出以下方法来解决：

1.Prefix Bloom Filter

加快range scan的效率

2.更高效的字符比较方法

Compaction的时候需要对memtable的key进行多次比较，因此

一个高效的字符比较方法能够有效提高Compaction性能

3.新的tombstone/deletion类型，二级索引的维护更加高效

4.批量导入，避免数据导入过快时候的compaction会引起stall，并且compaction引入限流机制，避免stall产生

5.RocksDB层使用更快速的压缩算法，而底层使用压缩效率高（相应地压缩时间会变慢）的算法。

那么memtable flush和compaction的速度就可以赶得上写入。

此外，MyRocks相比于InnoDB还有的原生优势是：

LSM相比于b+Tree的写放大可以减少，意味着需要flush的数据量也会减少。flush会涉及非常heavy的系统调用fsync

正文

1.UDB 的架构，B-Tree与LSM-tree的针对于闪存存储的比较

2.如何优化MyRocks的读场景和compaction

3.如何在生产应用MyRocks

UDB Architecture&Stroage

整体架构部分暂时忽略，我们更关注存储层的实现优化细节。

FB团队关注的一个大问题就是数据的压缩，因为硬件层面他们已经彻底放弃使用HDD而改用闪存设备,随之带来的影响就是存储数据肚饿成本大大增加。

B+树之所以会造成较大的空间浪费，原因主要在于叶子节点的分裂融合带来的索引的碎片问题。

另外一个比较严重的问题就是写放大。简单的一行数据的修改都需要把包含这一行的节点（一个page）重新flush到disk中，假如一个Page有100条数据，那么意味着写放大有100倍。

因此FB研发团队转向关注LSM-Tree作为存储引擎来解决空间浪费和写放大的问题。

但LSM-Tree是通过牺牲一定的读性能来换取高效地写入性能和空间优化效率。因此如何优化LSM-Tree来提高读性能也是一个关键的问题。

RocksDB: 针对闪存的优化

RocksDB(LSM树)的架构

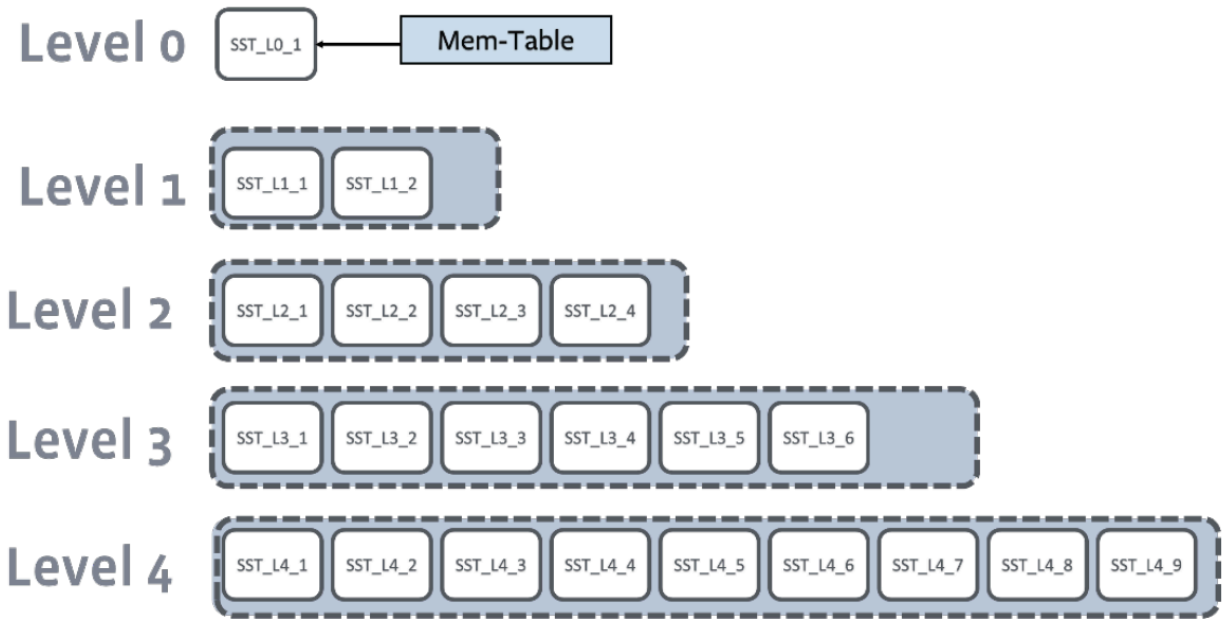
当数据写入RocksDB时候，首先会写入叫"MemTable"的内存buffer中，同时也会写入WAL中。当MemTable大小到达一定阈值，MemTable就会flush到disk上生成"SST"文件。SST文件内部按block划分，并且数据在SST内部是有序存储的。

每个SST文件还会有一个index block,用来二分搜索快速定位SST Block。

同时SST文件是分层存储的，每一层的文件大小随着指数级别上升。同时在合适的时候Level x 会和Level x+1 的SST文件做一个归并排序，生成一个新的SST文件。这个过程叫做Compaction.

读取过程：

首先会搜索所有的MemTable,紧接着Level-0 SST,如果还没有就往下一层寻找。每一层的寻找其实都是一个Binary Search. **同时每个SST文件都会维护一个Bloom Filter来消除SST文件内不必要的搜索**



使用RocksDB的理由

前面提到空间利用率和写放大是制约以B+Tree作为存储引擎的InnoDB的两大瓶颈。LSM树的更新是批量的，当flush page到disk的时候，page包含的数据都是更新的数据，不会像B+树一样同时把不需要更新的数据也flush到disk中，意味着写放大会显著减少。

而至于空间利用率，LSM树一方面不会有碎片问题，二是定期通过Compaction来清除已经删除的数据。并且压缩的效率也会大大提高：举个例子，如果16KB的数据需要压缩成5KB,InnoDB为了对齐会需要8KB来存储这5KB的数据，空间利用率只有62.5%. 而RocksDB使用5KB就可以了。

MyRocks的设计细节

https://zhuanlan.zhihu.com/p/45652076

- Clustered Index

	RocksDB Key		RocksDB Value	Rocks Metadata
Primary Key:	Internal Index ID	Primary Key	Remaining columns	SeqID, Flag

	RocksDB Key			Rocks Value	Rocks Metadata
Secondary Key:	Internal Index ID	Secondary Key	Primary Key	N/A	SeqID, Flag

一级索引：key包含Index ID 和 Primary Key, value 包含其他所有的列

二级索引： key包含Internal Index ID, Secondary Key和Primary Key

- 优化Key比较

lsm树相比于B+树会涉及到非常多次的key比较。

例如在范围查询中，寻找范围的初始key，在B+树中因为叶子节点已经是用链表有序排序了，因此需要一次binary search就够了。

然而lsm树对每个sst文件都要进行二分查找并且归并。rocksdb的迭代器本质上是对所有文件做堆排序，并且不断调整堆来找到最大的/最小的Key,涉及到非常多的Key比较。

因此MyRocks把key字段的数据编码为字节级别就可比较的方式，省去了反序列化的开销。

- 反向迭代Key

Rocksdb中正向迭代Key会比反向迭代快很多。

原因一是delta encoding，其次是因为老版本数据的存在，使得反向迭代需要在多读一次以获取最新版本的数据。并且memtable和SST文件中skipList都是由单向链表来连接的，因此也需要额外的查找请求。UDB团队根据他们的业务，自己实现了一个反向的rocksdb比较器，将反向迭代转为了内部的正向迭代。

- Prefix Bloom Filter

指定key的N个byte作为前缀, 过滤掉那些没有这些前缀的的key，加快Range Query的速度

- 定期compaction清除"TombStone"(删除标记)

Compaction是影响RocksDB性能的主要因素。RocksDB的二级索引包括许多列，包含时间戳和版本，并且还是个覆盖索引。目的是为了消除搜索primary key的随机读。然而每次更新数据的时候，都要更新时间戳和版本这两个字段，因此二级索引也需要频繁更新。

频繁更新二级索引列的值，在MyRocks的语境中意味着更新“key”。在RocksDB 中更新操作是由delete原来的key然后put新的key来完成的。tombstone作为一个标记，在memtable flush或者compaction的时候会忽略掉其标记的put操作。但是tombstone本身也会占据存储空间，在二级索引频繁更新的情景下，会产生大量的tombstone从而影响range query的性能。

FB解决方法是当在flush或者compaction的时候如果检测到大量的tombstone,就会额外触发一次compaction来减少tombstone的数量。

- 存储空间和压缩挑战

内存使用：

布隆过滤器对于加快RocksDB range query 性能至关重要。但是其会占用大量内存。一个工程trick是最后一层SST 文件不建bloom filter, 这样仍然能起作用，也极大舒缓了内存的压力。

总结和对毕设的启发之处：

介绍了Lsm Tree作为MySQL存储引擎相比于B+ Tree的一些优势，主要是解决了B+树的写放大和空间浪费问题。

然后论文主要也介绍了一些优化LSM的技巧，例如prefix bloom filter来加快range query查询性能，如改进压缩算法，如改善二级索引频繁更新带来的性能下降等问题。

论文其实也提到了FB内部把Hbase向MySQL for Rocks 迁移的工作，或许我的毕设也可以做做尝试。然后可以**考虑自己再实现一个布隆过滤器**，比较在范围查询的时候是否能提高性能。

Efficient Data Ingestion and Query Processing for LSM-Based Storage Systems

LSM 针对批量点查的优化

http://www.vldb.org/pvldb/vol12/p531-luo.pdf

本论文重点关注LSM-based 的存储系统的数据写入和query processing的性能。

主要提出了一系列关于优化批量点查的技巧，以及LSM-Tree作为二级索引的一些trick

Introduction

重点介绍加快即时查询效率的两种数据结构：二级索引和过滤器。

二级索引也是一棵LSM树，负责把secondary key映射成primary key.

Filter尤指Bloom Filter 或者 Range Filter,直接构建在LSM Tree内部，加快搜索效率。

但问题在于维护这些额外的数据结构会带来不小的开销。特别是涉及到更新操作。主流的基于LSM的存储系统如AsterixDB,MyRocks与Phoenix，每当进行写入操作的时候都会对这些数据结构进行一次点查，论文提到这里的点查是开销非常巨大的。

论文的主要贡献：

1.针对点查提出一系列的优化措施,如何做二级索引到primary index 的映射

2.提出创新的索引维护策略，如通过主键索引来清理过期的二级索引。以及提出mutable-bitmap

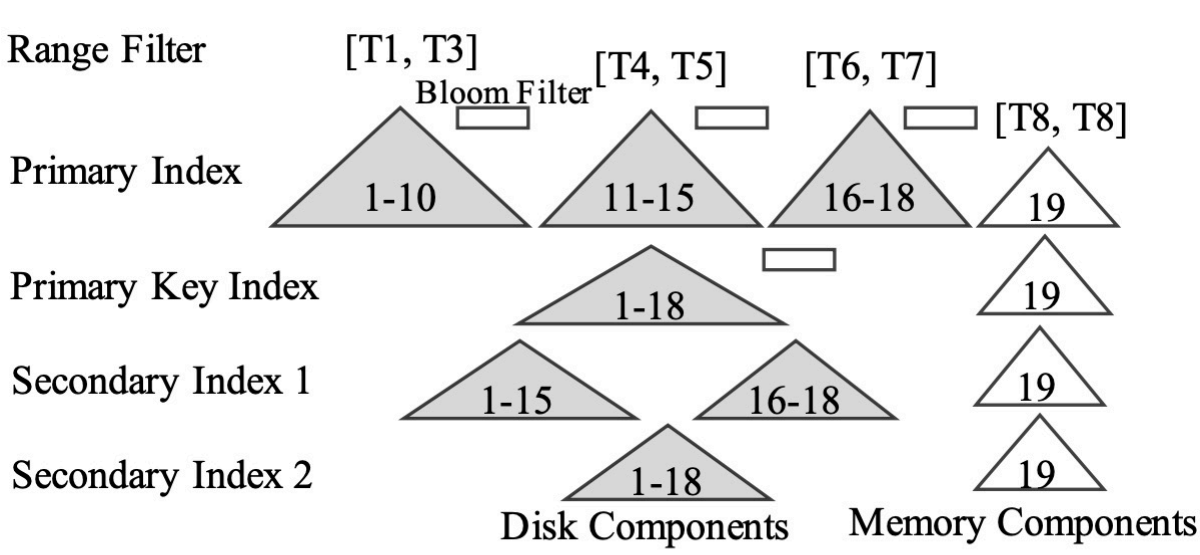


Figure 1: Storage Architecture

LSM STORAGE ARCHITECTURE

该部分介绍本论文提出的LSM存储架构

每一个dataset都会维护一个Primary Index, Primary Key Index, 一系列的二级索引。二级索引都是基于LSM树。

所有的索引都共享内存，所以他们通常都是同时flush到disk上的。

每一个索引都有一个component id，具体指的是这个索引存储记录的最小-最大Timestamp

例如Primary Index 1-10,意思是存储的记录的时间戳是1-10

通过Component ID可以推断出索引之间的新旧情况。例如二级索引1-15比16-18要老，并且与1-10的parimary index会有重叠。

那么primary index 和 primary key index 有什么区别呢?

primary key index 仅存储primary key本身，而primary key index存储的是被primary keys索引的记录数据。

primary index和primary key index内部通过B+树来维护。并且这两个组件都会有一个in-memory的布隆过滤器。只有当布隆过滤器表示key可能存在的时候，做点查。

二级索引为了解决重复的secondary key问题，会使用secondary key和primary key作为索引主键。当进行查询的时候，首先查询secondary index 来获得匹配的primary keys的列表，然后再根据primary index来进行点查。

除了布隆过滤器之外，每个Primary Index都会额外有一个 range filter作为二级过滤索引。

CRUD策略

插入记录，首先会执行一次点查。这一步为了效率可以直接查primary key index,如果primary key已经存在的话，直接忽略这个插入操作。否则所有内存中的索引都要更新这条新插入的记录。

删除记录，首先还是只行一次点查，如果不存在就忽略这个删除操作。如果存在的话，插入一条“tombstone”到所有的LSM 索引中，标记这个entry已经被删除。包括range filter和bloom filter都需要相应做更新。

更新记录，执行一次点查查询是否存在，如果存在的话，给所有的二级索引写入"tombstone"来标记旧记录被删除。同时内存中的filter也要做相应的更新。

LSM的点查优化技巧

批量点查：

虽然primary key整体已经是排好序的了，但是主键可能分散地由多个LSM二级索引来映射，所以需要同时查询多个LSM二级索引，这会涉及到随机IO.为了避免这种情况，论文提出了批量点查的算法来做优化，核心是把随机读取b+树叶子节点转化为顺序读取。

首先把排序好的主键划分为若干batch,对每一个batch的主键,顺序依次从新到旧访问若干个LSM二级索引。对一个batch中没有被二级索引映射到的primary key,就去查询bloom filter和primary key index。

blocked bloom filter--**缓存更加友好的bloom filters**

总结起来，blocked bloom filter就是把原来的bitmap划分为若干个block大小的更小的bloom filter

首先把bitmap按照CPU Cache Line的大小划分为若干个block.第一个hash函数负责把key映射到block中，然后剩下的k个hash函数负责把这个key映射到这个block中。因此可以把cache line的misses从k次降低到1次

总结

本篇论文的主要贡献是为了优化索引查询效率，给主键索引增加一系列的filter如bloom filter或者range filter。并且在更新频繁的场景提出了索引更新的优化策略。对毕设的启发是：

1.了解了blocked bloom filter相比于传统bloom filter的优点，在github也找到了开源实现，也许可以尝试使用一下来加快我们查询的性能。

2.为LSM提出的批量点查“batch point query”为读优化提供了思路,核心是消除读LSM的随机IO

Accordion: Better Memory Organization for LSM Key-Value Stores

改善lsm的内存碎片问题，在hbase上改

http://www.vldb.org/pvldb/vol11/p1863-bortnikov.pdf

Abstract

LSM两个主要问题：compaction会影响写入效率以及动态内存分配的碎片

论文主要贡献是提出了“Accordion”这种针对基于LSM存储引擎的内存组织规范，目的是为了解决上述问题。

正文

LSM的性能瓶颈

对Compaction是非常敏感的

如果Compaction速度过慢，会影响读性能。因为这就需要读取更多的文件，并且数据在sst分布的越分散，缓存的利用率就会越低。

如果Compaction速度过快，会影响写性能，因为会和写入现场争夺CPU和IO资源，同时过快的写入意味着过快的缓存更新失效。另外过快的话也会造成磁盘磨损，尤其是SSD

其次内存管理是个大问题。从内存flush到disk的数据量会影响到compaction的吞吐量，虽然加大内存容量可以减少flush的频率，但是lsm是不会从memory store清除被更新覆盖或者被删除的老数据。因此并没有减少memory flush到disk的数据量。目前的解决方法要不就是耗时耗力的垃圾回收，要么就是预分配的内存池，或者把数据移到堆外内存。

Accordion

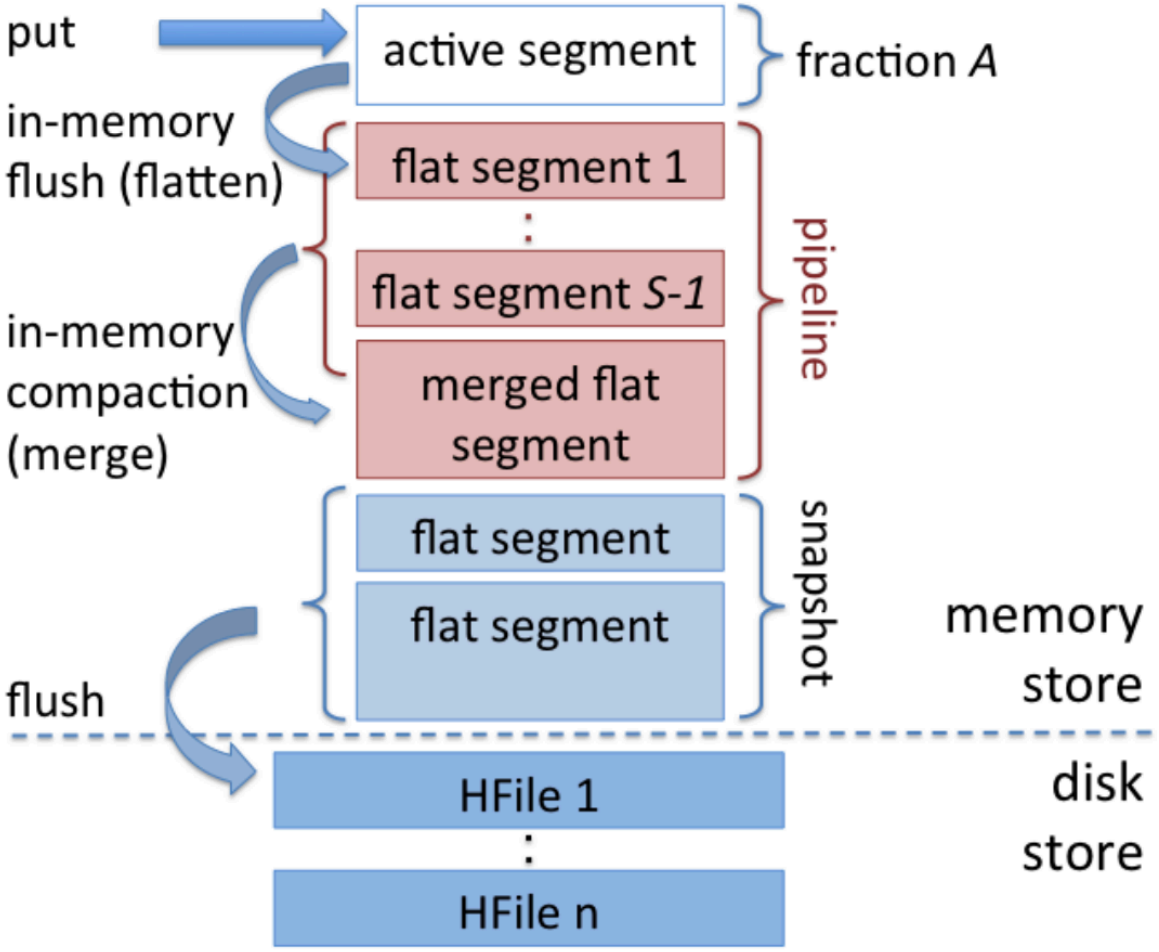
内存memTable分为两个部分：

active segment和flat segement

1.小的动态segment用于写入

2.由动态segment 直接在内存中进行flush而生成的序列化segment，那就不用直接在RAM存储java对象，只需要存储序列化后的bytes, 这个in-memory flush的频率会比disk的flush频率更高。并且compaction操作也是直接在内存中进行。

3.读文件时候，LSM是需要同时读取多个disk上的SST文件的，会带来随机IO. 而segment是内存存储的，读取搜索速度更快。



总结起来：

数据在memTable就直接消除冗余的脏数据，这样可以降低flush到disk的频率，从而减少写入放大和整个磁盘的占用空间。RAM中脏数据少了，能够直接在Memtable搜索的的key变多了，读效率提高。

而且由于刷新次数较少，memStore溢出时候写入操作停顿的频率会降低，改善写入性能。

而且磁盘上的数据越少，对块缓存压力也会越小，cache命中率越高。

总的来看，最后在磁盘上的数据少了，在disk中进行compaction的次数也会减少，会减少与生产中读写操作争夺CPU和IO资源.

而且内存的compaction就消除了脏数据，总的垃圾回收的开销也大大减少.

Hbase2.0已经引入了Accordion机制了,可以直接拿来使用。

Background部分

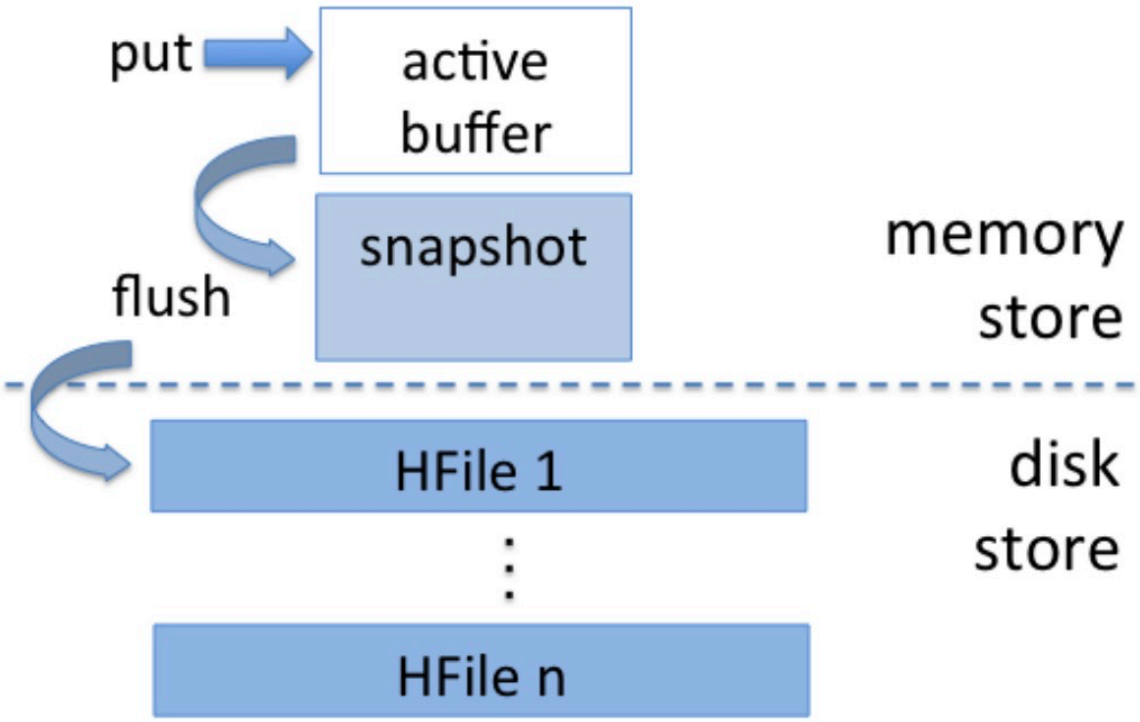
主要介绍的是Hbase的

数据模型

- 每一行数据由一个row key唯一标记
- 每一行可以包含多个列，多个列可以被划分成一个列族
- 数据可以有多个版本，每个版本由一个timestamp唯一指定
- 数据的最小单元叫做cell,是由row key, column key和timestamp来共同指定的。

架构

- 通过row key来做sharding, Hbase中的shard就是region.
- region是由一系列stores的集合，每个store关联一个列族。
- 每个store又是多个row key顺序排列的Hfiles所组成的



为了最大化put操作的效率，MemStore采取的是double buffering策略，一方面维护一个动态的buffer来处理put的记录，然后当动态的bufffer到达一定阈值的时候，又会在内存中flush成静态的buffer。然后后台会有flush线程把静态bufferflush到文件系统中，即生产HFile. HFile即Hbase中的SST文件。

Hbase中的WAL叫做Hlogs.不赘述。

内存组织

首先来讲一下Hbase原先的内存组织方式：

MemTable的buffer会为cell维护动态的索引，具体是用java的Concurrent skipList map

数据是多版本的，每次put操作会生成一个immutable的row。

这种实现方法有两个缺陷：

维护一个大的Concurrent SkipList Map还带来很多垃圾回收的性能消耗，特别是在object特别小,meta特别多的情况。

在flush之前不会消除冗余的过时版本数据。给读取带来不少IO的浪费。

Memory allocation

Hbase主要有两种策略在内存中分配data cell：

1.每个cell都是一个pojo

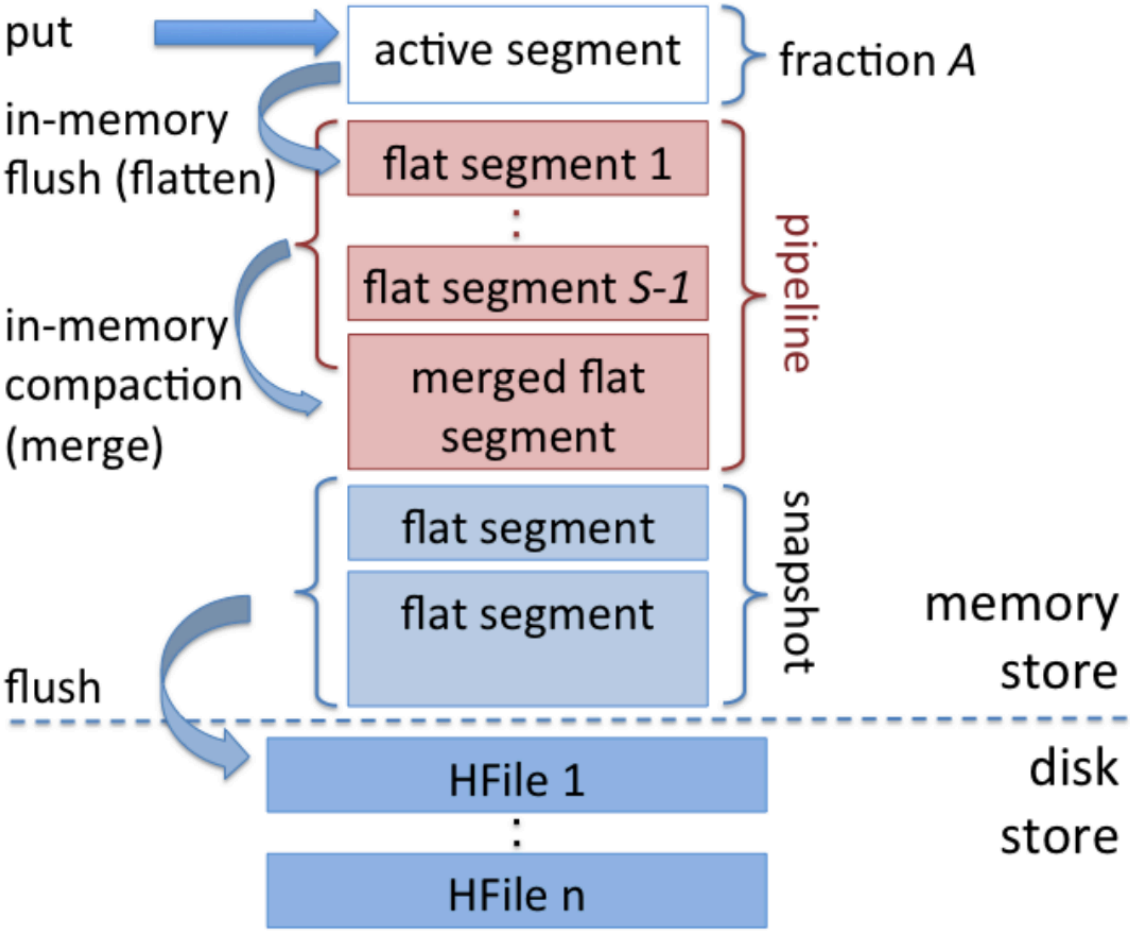
2.使用“memstore local allocation buffer——MSLAB”分配的字节数组来分配。 MSLAB可以和TLAB类比理解，意思应该是为每个MemStore单独分配一个其独占的，固定大小的chunk。每一个chunk负责分配维护这个MemStore所有cell。

详细介绍Accordion

MemTable里的数据组织方式是：若干个由创建时间排序的segments组成一条Pipeline。每个segment包含了一个flatten的index，用于映射若干个data cells。flatten的含义是把skip list替换成更加紧凑的有序排列的数组，用于减少内存中segment存储的开销,减少了memtable flush到disk的频率。并且数组形式更好支持binary search搜索。同时也可以分配在堆外内存。

最新的segment叫做‘active’ segment,是可更改的,负责处理put操作。剩余的所有segment都是不可更改的。其实就是在memTable内部应用LSM的思想再做细致的内存划分罢了（immutable memory segment类比sst 文件, mutable memory segment类比meltable）。读取文件的时候，和LSM需要读取多个文件类似，get和scan操作也需要遍历所有的memory segments。

虽然这些内存的segment不会像disk上的sst文件一样维护bloom filter来加速查询，但是也会维护若干个metadata例如key ranges和time ranges来加速。

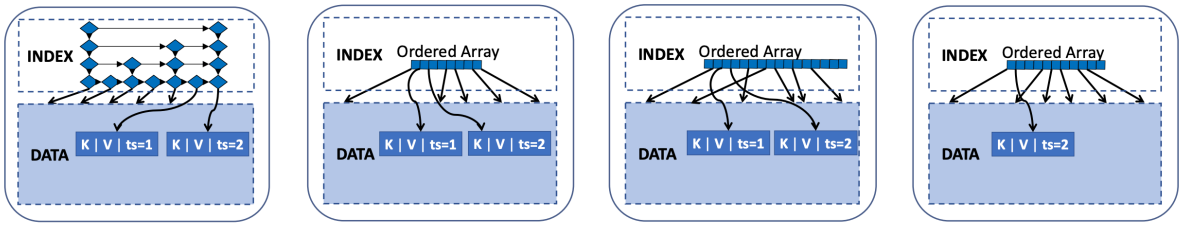


下面用四张图总结一下Accordion的核心思想：

1.活跃的segment的索引组织方式是跳表索引(Java中的ConcurrentSkipListMap)

2.当segment变成immutable后，把跳表索引flat成有序的数组，主要是为了减少内存的开销

3.但是有序数组并没有去重，只有内存中的compcation操作才能把相同数据的旧版本删掉



(a) Active segment (b) Flat segment (c) Merged segment (d) Compacted segment

总结

Accordion在MemTable中应用了LSM的思想，做了更加细致的划分，提出了segment这一内存中的存储单元。核心是通过直接在内存中清除冗余的脏数据，一方面减少内存使用，另一方面减少memTable flush 到 磁盘生成SST文件以及compaction的频率，从而提高系统整体性能。

对毕设的启发：

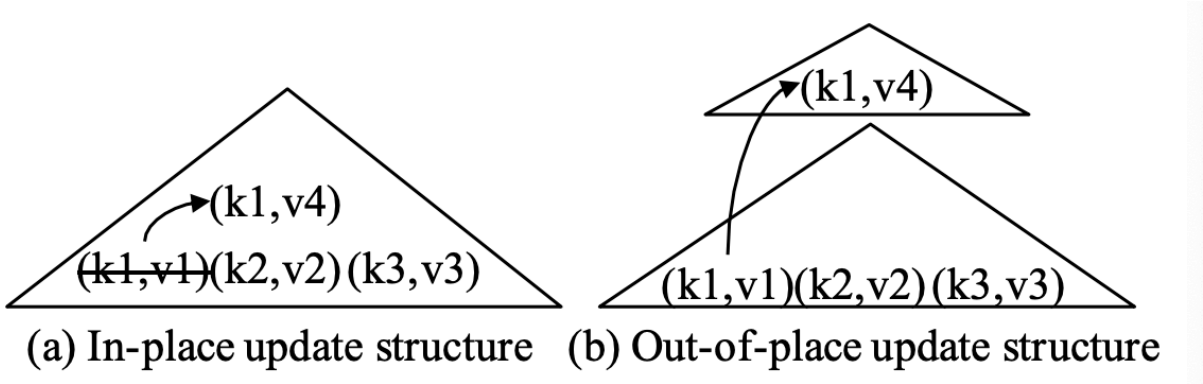
Accordion已经在HBase 2.0投入使用，可以尝试比较Hbase1.26以及Hbase2.0开启该功能前后的性能。

LSM-based Storage Techniques: A Survey

一篇较为完整的LSM存储引擎的综述，还介绍了其许多优化后的LSM变种和实际工业界的应用:LevelDB, RocksDB,HBase,Cassandra,AsterixDB

https://arxiv.org/pdf/1812.07527.pdf

LSM 介绍



b+ tree属于就地更新的数据结构，即新数据记录会覆盖原旧的记录。

优点：读性能好，因为读取的数据版本是最新的，不需要查询多个版本

缺点：牺牲了写性能，因为更新操作涉及到随机io,并且更新删除会造成page的碎片

而LSM属于out-of-place更新的数据结构，插入新的记录的时候，不会覆盖原来旧的记录，而是以追加的方式。

优点：写性能好，把随机io转化为顺序io

缺点： 读性能差，因为一条数据记录可能会有多个版本，需要读取多个位置。

数据结构

MemTable通常由skipList实现

SST文件通常包含一个index block和若干个data blocks

Data block存储着key有序排列的键值对

而Index Blocks存储着所有data blocks的key范围

点查&范围查

- 点查

必须从最新的key搜索到最老的key，直到找到合条件的记录

- 范围查

可以同时查询所有的memTable和SST文件，把在范围中的结果放到优先队列中

两种Compaction策略

Size-tiered compaction strategy (STCS)

按大小划分，当若干个SST文件到达阈值的时候就merge成更大的SST文件

Level-based compaction strategy (LBCS)

按照level来划分存储，当一个level的所有sst 文件大小到达阈值，就compact这个level的部分data到下一个level

常见优化

- Bloom Filter

用于SST文件的查询加速。尤其是点查，通过Bloom Filter过滤掉那些肯定不存在需要查询的key的SST文件。

核心思想就是用内存空间来换更少的disk IO,极大提高点查效率

- 分区

LevelDB 中把Disk Component划分为若干个大小固定的SSTables就是分区。减轻单个component的读取压力和compaction的压力

缺陷和改进

- 写放大

B+树和LSM的写放大代价分析具体比较可以在这里看到

https://tikv.org/deep-dive/key-value-engine/b-tree-vs-lsm/

LSM的写放大我的理解主要体现在数据插入后，随着compaction从上一层迁移到下一层产生的额外disk io。

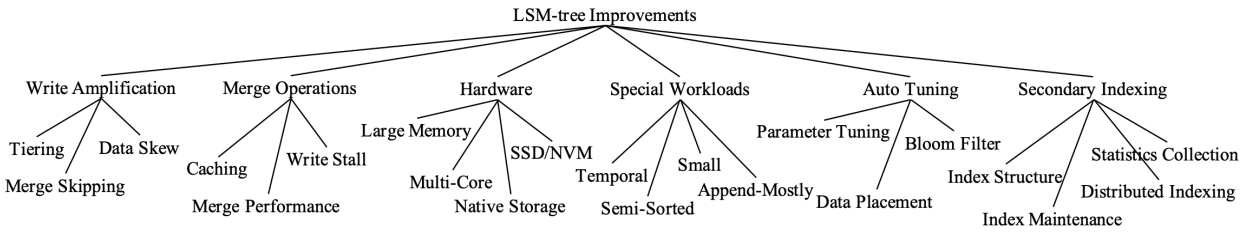
- Compaction

Compaction操作会和正常的读写操作争夺CPU和IO资源，影响性能同时也可能会造成写入的stall，同时过于频繁的compaction会导致buffer cache misses几率上升，进一步影响读的效率

- 二级索引优化

如何维护二级索引，从而尽可能减少对写性能的影响

下面是一张LSM优化方向的思维导图



代表性的LSM 系统

- LevelDB

基于sst的lsm系统先驱,一个单机KV引擎

- RocksDB

基于LevelDB开发,增添了许多优化点

1.动态调整compaction的阈值，尽可能减少空间放大

2.level 0的sst文件是不分区的，因此从level 0 到 level 1的flush通常会造成部分数据重复写入，造成写放大

3.相比于LevelDB用round-robin选择进行merge的sst文件,RocksDB提供更加好的策略如把不常用的sst文件或者是删除标记更多的的文件进行优先compaction，那么更频繁访问的sst文件就可以在更上层，减少读放大。

4.引入令牌桶限流机制，限制compaction的速率，减少对正常读写的性能影响

5.支持“read-modify-write”操作，现有系统在更新记录的时候，首先需要读取该记录，

而为了更高效实现更新写操作，现在RocksDB允许用户直接写数据到内存，然后只有在读的时候/merge的时候再进行多余数据的合并。

- Hbase

和LevelDB与RocksDB是单机版的KV引擎不同，Hbase是基于Hadoop生态的分布式系统。

Hbase中的数据分区叫做region,每一个region由一个LSM树来作为存储引擎。HBase还支持动态region划分

Hbase的merge策略比较灵活，会扫描所有可以merge并且写代价最小的HFile进行merge

不支持原生二级索引，需要额外用另外一张表来secondary key和primary key，通过*co-processors*来实现

并且Hbase还支持把大的region继续划分为更小的partition来独自进行merge

- Cassandra

和Hbase的主从架构不同，其为了避免单点故障采用的是去中心化架构。

支持原生二级索引

为了减少点查开销，二级索引是“lazy”地维护。更新的时候，如果发现老的记录在内存中已经存在，直接清除二级索引相应的记录。否则只有在进行merge操作的时候才会清除二级索引。

- AsterixDB

用于管理海量半结构化数据如json

是shared-nothing的架构，数据会根据主键hash分区到不同节点。每个分区就是一个LSM存储引擎。

并且每个分区还有primary index, primary key index

对毕设的启发

可以试用Cassandra 作为metadata存储的后端，而且因为其支持原生的二级索引，因此可以和Hbase做性能比较