1.Introduction

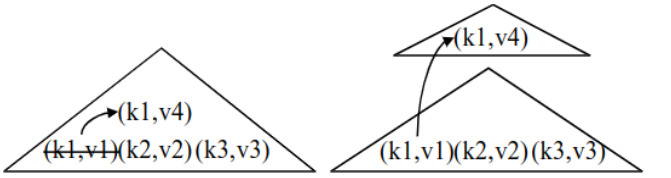
LSM tree具有的优势：高写性能，高空间利用率，结构可调整，并发和恢复机制相对简单；同时对于SSD来讲，存储空间比B-tree要小。

merge（合并）操作能改善空间利用率、读速度，并都有相关理论证明。

2.Background

2.1历史

不同于B+ tree的in-place 更新，lsm-tree采用out-in-place方式，即批量写入，下图是这两种写入方式的区别：



(a) In-Place Update Structure (b) Out-Of-Place Update Structure

这种基于日志的方式起源于1970年代，1980年代postgres 开创了log-structured思想，利用数据重组的后台进程合并日志。这里log-structured和lsm tree是有差异的。

缺点：

相关的记录在日志中比较分散，查询性能低下；同样的数据重复率高，空间占用大；没有一个衡量读写、空间的标准，导致很难在上面做优化；重组过程成为了系统的瓶颈。

lsm tree在1996年被提出，主要贡献：reorganization processes-->merge process，采用类似树的层次结构：

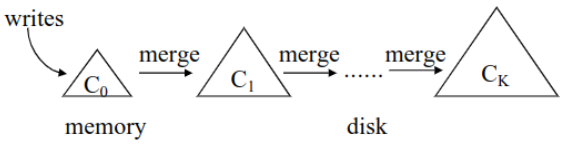


Fig. 2: Original LSM-tree Design

leveling merge & tried merge:

leveling 实现方式较为复杂，其提出的Ci+1/Ci = Ti，即下层数据量和上层数据量有着固定比率，影响深远。

tried 作为leveling 的并行化改进，每层都拥有T个component（或称run）。

2.2现在的lsm tree

2.2.1基础架构

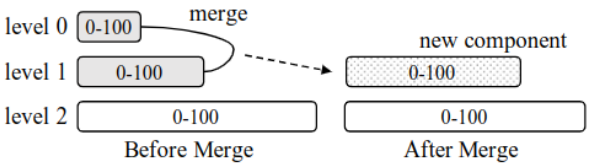
仍然使用**缓冲区消除随机写入的代价**。删除时使插入一个标记代表该数据项已经删除，后续合并（merge，也称compact）时会删掉该数据项。不同于以往的设计，磁盘上的immutability components除了合并是不可更改的，这样能够**更简单地实现并发和恢复**，同时意味着合并时的component是不可读的。

通常内存使用跳表或B+树实现并发写入的component，**磁盘上使用B+树或者SSTable**(sorted-string tables)。SSTable由数据和索引（数据项的范围）组成。

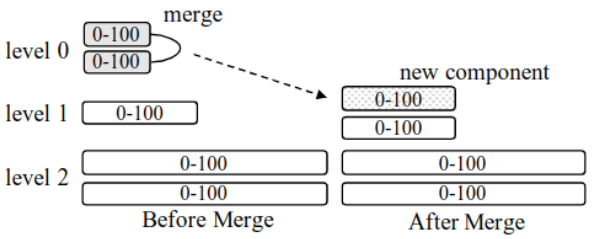
点查询通常要搜索多个run，当搜到一个最新的run时立即返回。

范围查询可以并行搜索多个run，把结果放入优先队列进行去重。

两种合并方式：leveling merge & tried merge



(a) Leveling Merge Policy: one component per level



(b) Tiering Merge Policy: up to T components per level

第一种是每层一个run，缓冲去数据满时，把缓冲区和第一层的数据合并放到第一层，若第一层放不下，递归上面操作放到第二层。

第二种是每层T（通常为T）个run，每次合并，把该层所有的run合并放到下层当中。

2.2.2经典的优化

- **布隆过滤器：使用类似位图的方式标记某个值是否放入数据库，由于该结构不支持删除，所以不能记录对应的删除操作（即具有那伪性质）。当布隆过滤器返回某个值存在时，并不以为着改值一定存在；但是返回某个记录不存在时，该记录一定不存在。理论错误率为：**

$$\left(1-e^{-kn/m}\right)^k,$$

k是hash函数的个数，n是键值对数目，m是过滤器大小（位数）。通常取k=10，错误率为1%左右。

- **分区：将一个较大的run划分成较小的多个run，优势：1.可以较少合并时间，同时减少合并时创建新文件的暂存空间;2.连续性的写操作通常避免合并，因为这些数据的key范围不会有重叠；同时对于倾斜的更新也不会造成冷数据的多次合并，只会经常合并热数据所在的run。**

对于以前使用的rolling merges？实现较为复杂，现在系统通常采用物理分区。

> **本文后续采用component作为未分区的run，而SSTable作为已经分区的run。**

分区是对于合并策略来讲的，适用于leveling和tiering。

目前分区优化仅仅在LevelDB和RocksDB工工业系统中实现，都是基于leveling合并方式，tiering合并的方式的分区提升写性能也在几篇文章中提出。

几种分区合并策略详见文章图解。

2.2.3并发控制和恢复

- **并发：**

并发控制关注的几点：读、写、缓冲区刷盘、合并。

读写：对于lsm tree使用的两种控制方式：locking scheme、muti-version scheme。通常采用后面一种多版本的实现方案，因为lsm tree本生就是一个多版本结构。

缓冲区刷盘、合并：

元数据。缓冲区刷盘和合并通常会修改元数据，所以要精心设计并发。

对于磁盘上的run，避免在使用时被合并操作删除，需要对该run加入引用计数器，针对range查询，创建一个快照，并讲引用计数器++。

- **恢复：**

日志预写(WAL) 保证持久化。

使用no-steal buffer管理策略，只有所有当前活动事务终止(包括提交？)时，才刷buffer到磁盘，恢复时只回放redo日志，不回放undo日志。

对于磁盘run，为保证故障后能够正确恢复，分两种策略：

第一种是对于采用未分区的，采用**时间戳对**。（这部分文章中没有讲具体实现方式，需要另行调研。）

第二种是对于采用分区的，像LevelDB和DocksDB，维护一个和SSTable分离的元数据日志保存对所有元数据的修改，恢复时回放该日志。

2.3各种操作开销

Merge Policy	Write	Point Lookup (Zero-Result/ Non-Zero-Result)	Short Range Query	Long Range Query	Space Amplification
Leveling	$O(T \cdot \frac{L}{B})$	$O(L \cdot e^{-\frac{M}{N}})$ / $O(1)$	$O(L)$	$O(\frac{s}{B})$	$O(\frac{T}{T-1})$
Tiering	$O(\frac{L}{B})$	$O(T \cdot L \cdot e^{-\frac{M}{N}})$ / $O(1)$	$O(T \cdot L)$	$O(T \cdot \frac{s}{B})$	$O(T)$

Table 1: Summary of Cost Complexity of LSM

以下都是讲最差情况下的开销。

需要的层数L估计：定义*B*为每个data page可以存放的entries数目，*P*为buffer可以放入的data page的个数，则每个buffer能放*B*P*个entry。假设系统插入和删除是相等的，则系统能存放的entry总数*N*是个定值。最大一层大约能放N*((T-1)/T)个enrtry，所以需要L：

$$L = \lceil \log_T \frac{N}{B \cdot P} \cdot \frac{T-1}{T} \rceil$$

写代价：

点查询代价：

短range查询：

长range查询：

3.已有改进方向分类

- **Write Amplification 写放大**
- **Merge Operations**
- **Hardware**
- **Special Workloads 特殊负载**
- **Auto-Tuning 自动调优**
- **Secondary Indexing 二级索引**

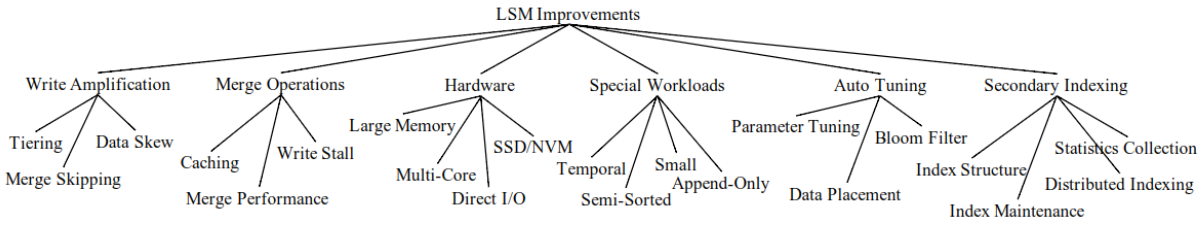


Fig. 6: Taxonomy of LSM Improvements

3.1 Write Amplification 写放大

优化目的：相对于B+树in-place更新，lsmtree的合并策略更容易导致写放大；同时也会减少SSD寿命。

- **Tiering**

WriteBuffer (WB) Tree 1.利用哈希分区进行负载平衡，每个SSTable组存储总量相当的数据。2.利用类似B+树的结构，当节点数量饱和时，拆分节点至多个子节点（或同级叶子节点）。

light-weight compaction tree (LWC-tree) 同样利用垂直分区平衡负载。在垂直负载下，SSTable大小不是固定的，而是一个大致相等的值，合并之后，如果run太大，则进行拆分，即缩小range范围，把部分range范围的数据给其他run。

PebblesDB 没看明白这段英文，必要时可看其相关论文。

dCompaction

- **Merge Skipping**

主要思想，直接把buffer数据刷到下面几层，跳过刷到第一层的步骤。

- **Exploiting Data Skew（利用数据倾斜）**

这就是一个缓存热数据的方案，对于已知的倾斜数据，放到内存中，避免多次合并产生不必要的开销。

3.2 Merge Operations

- **Improving Merge Performance**

VT-tree 假设有些run没有重叠Key的情况，直接用指针链接各个run，省去合并操作。但是这引入一个问题，会造成扫描（scan）随机I/O。所以设置了一个阈值K，只有run数目超过时，才合并run。这里不使用bloomfilter，而是使用quotient filters。因为构建quotient不需要访问所有的源数据。

对于合并操作，分读、排序、写三个部分，这三个部分对系统开销要求不一样，读写占用I/O高，合并占用CPU高，所以有人在并行读写、合并方面做出了一些贡献。

- **Reducing Buffer Cache Misses**

对于缓存的方式不是太了解，这里看的云里雾里。我的理解是合并之后会造成缓冲区命中率降低。

一个方案是把造成CPU、I/O高负载的合并操作发送到远程服务器上，然后通过智能预热算法预取数据。通过将大的未命中分成多个小的缓冲区未命中来减少查询影响。

The Log-Structured buffered Merge tree(LSbM-tree)。对于合并到下层的run中的数据，不是在合并后立马删除，而是先放到缓冲区里，由数据的访问率，后期决定是否清理掉。

- **Minimizing Write Stalls**

bLSM。目的：减少buffer刷盘和合并造成的write stalls。在每层加入一个额外的组件，尽可能并行去合并。

3.3 Hardware

- **Large Memory**

直接使用VM-managed data structures，会造成GC开销；但是使用对外内存B+树结构时，又会造成查询的定位开销。

FloDB使用两层设计，上层用hash来并发写入，然后下层用跳表支持大范围查询。但是，只有跳表的数据刷进跳表中时，才可以进行范围查询。

- **SSD/NVM**

FD-tree。使用fractional cascading 代替bloomfilter，提高查询性能，类似B+索引，每层都会有一个fence pointers指向下层。但是，实现复杂，合并时也复杂。但是，现在的系统倾向bloomfilter。

WiscKey。key-value分离，合并只对key有效，一定程度上减少开销，但是，value只是append，没有排序，查询不友好。对于value的垃圾回收也会造成性能瓶颈。

LOCS。利用Open-channel SSD来实现并行化。部分SSD时对外开放channel接口的。

Kreon。采用和WiscKey类似的思想，对于范围查询，会将其键值对存到一块新空间，这里面的键值都是排过序的。同时利用内存映射提升性能。

NovelLSM。

- **Multi-Core**

cLSM。把lsm run用并发链表组织起来，减少同步延时。buffer刷盘和合并都是链表的原子修改。

- **DirectI/O**

LSM-tree-based Direct Storage system (LDS) 。磁盘上主要分三个部分：数据块，版本日志和恢复日志。三部分具体不做过多笔记。

NoFTL-KV。从字面上很明显，去除SSD的STL层，获取更大性能。毕竟STL的磨损均衡和地址映射造成一定开销。LOCS的并行有点这方面的意思。

3.4 Special Workloads 特殊负载

3.5 Auto-Tuning 自动调优

- **Parameter Tuning**

Lim等人提了一个分析模型，分析有些删除的key应在最上的合并中删除，而不是一直传递下去。这是它的写入成本公式：

$$Unique(p) = N - \sum_{k \in K} (1 - f_X(k))^p$$

- **Tuning Merge Policies**

Dostoevsky。即leveling和tiering的合体版本。因为点查询、长范围查询、写放大都是在最接近下面run才有的，所以在上面几层，用tiering，下面几层用leveling。有两个参数可调，leveling的最大个数Z，和每层较小run个数最大值K。这个论文提供了大量的证明。

- **Dynamic BloomFilter Memory Allocation**
- **Optimizing Data Placement**

Mutant 。针对云存储优化的。它能监测SSTable的访问频率，然后把访问最高的子集放入到内存中。这可以在给定预算下，提供更好的性能。

3.6 Secondary Indexing 二级索引

概要：索引技术：如图：

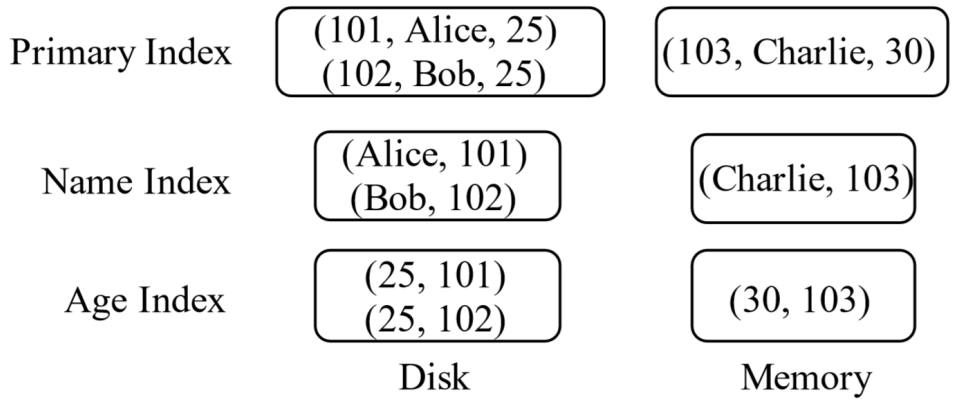


Fig. 14: Example of LSM-based Secondary Indexes

- **Index Structures**

The Log-Structured Inverted Index (LSII) 日志结构倒排索引。

4.Representative LSM-based Systems

- **LevelDB**

采用leveling merge的鼻祖。

- **RocksDB**

T为10。可以控制绝大部分数据在最下层，约90%，提供了较高的空间利用率。远远好于B树67%的利用率。

虽然基于leveling，第0层的SSTable没有分区，合并到第一层时，需要写入所有的SSTable，会造成性能瓶颈；所以它提供了可选的第0层tiering方案。

另外可以动态调整，确保写放大控制在O(T/T-1)。

除了round-robin合并策略，还提供了另外两种策略-cold first 和 delete-first。

- **HBase**

参考的google的bigtable，将数据划分为区域集合，每个区域有一个存储引擎管理。采用tiering策略。本生不支持二级索引，支持可插拔索引结构。

- **Cassandra**
- **AsterixDB**

5. Future Research Directions

Thorough Performance Evaluation

Partitioned Tiering Structure

Hybrid Merge Policy

Minimizing Performance Variance

Towards Database Storage Engines