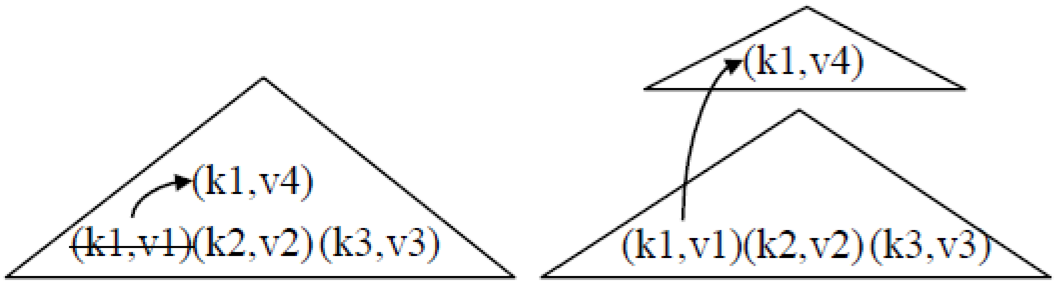
基于LSM的KV存储写放大优化 - tiering

2021.10.31

目录

- LSM基础与写放大
- 写放大的优化-tiering
    - LSM-trie
    - PebblesDB
    - WipDB
- 总结

LSM 基础



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

Fig. 1: Examples of In-Place and Out-of-Place Update Structures: each entry contains a key (denoted as “k”) and a value (denoted as “v”)

就地更新（例如B+树）直接覆盖旧记录以存储新的更新，如图a

1. 仅存储每个记录的最新版本
2. 更新会导致随机I/O（树形结构还涉及平衡性的维护）
3. 索引页会因为更新和删除而碎片化，从而降低了空间利用率

异地更新始终将更新存储到新位置，如图b

1. 利用顺序I/O来处理写操作
2. 不覆盖旧数据，也简化了数据恢复的过程
3. 记录可能存储在多个位置中的任何一个位置
4. 需要单独的数据合并过程来提高存储和查询效率

LSM 基础

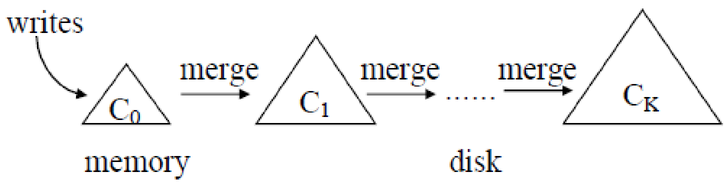


Fig. 2: Original LSM-tree Design

1996年LSM问世

1. 多组件组成
2. 组件采用B+Tree结构
3. 复杂的滚动合并

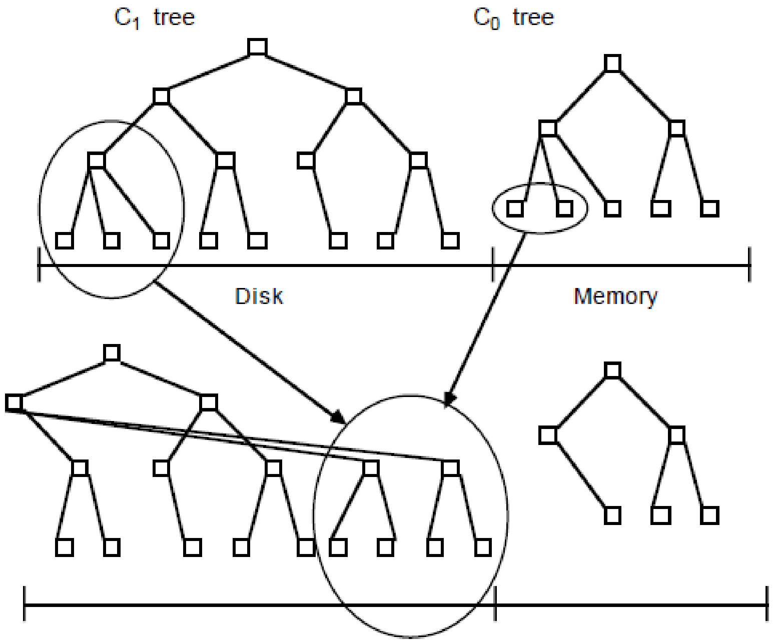
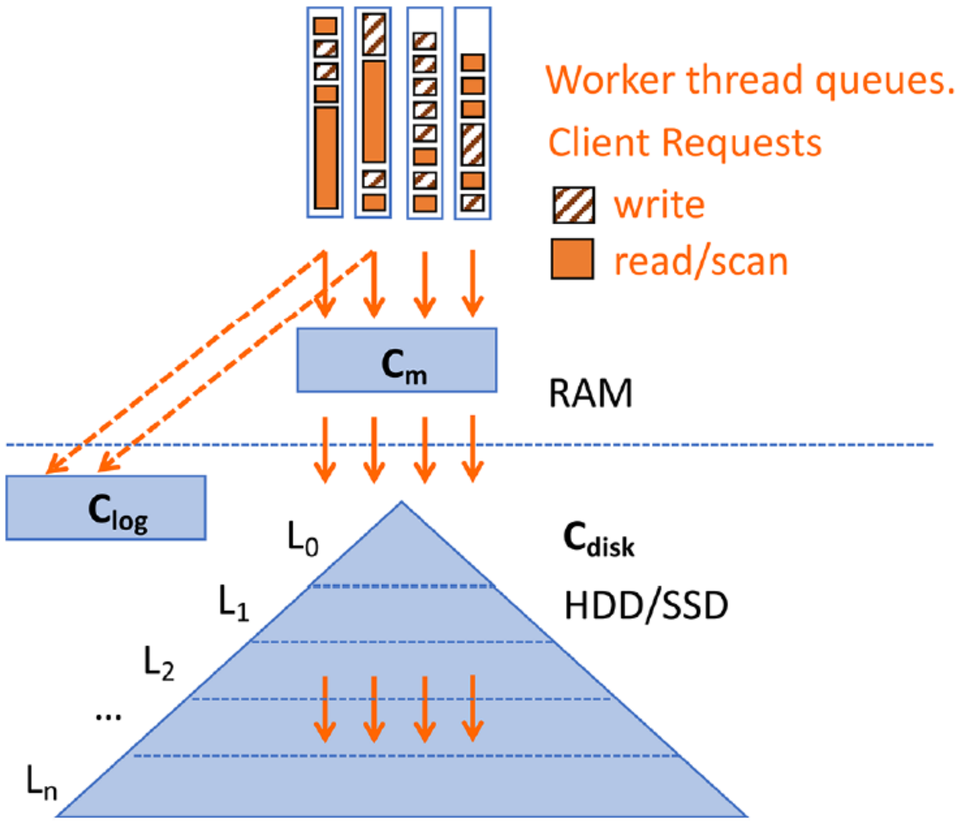


Figure 2.2. Conceptual picture of rolling merge steps, with result written back to disk

LSM 基础

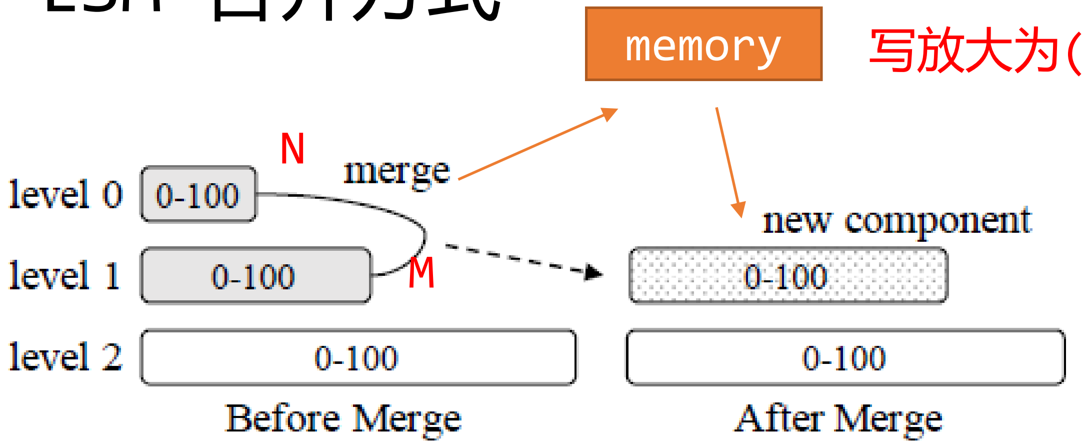


现在的LSM：

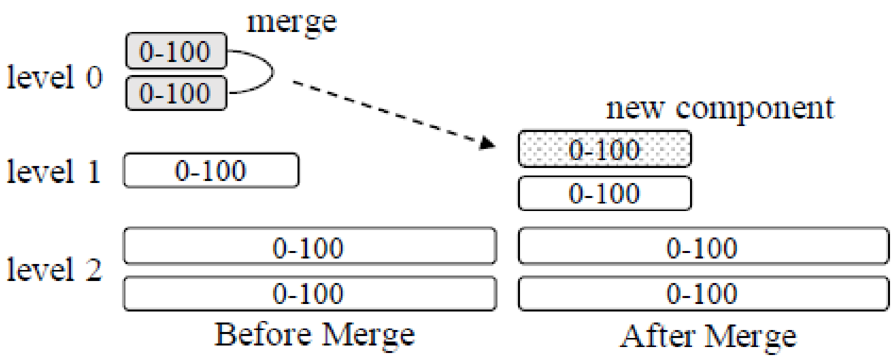
1. 预写日志提供数据可恢复性
2. 层数固定，且大小指数递增
3. 磁盘组件使用排序字符表（sstable）
4. 磁盘合并无需修改现有组件
5. 缺点：降低读效率

LSM 合并方式

写放大为(M+N)/N = 实际数据量/真实数据量



(a) Leveling Merge Policy: one component per level



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

leveling

1. 每个level只维护一个组件
2. 同一level中kv唯一
3. 每一层最大写放大为level容量比
4. 总写放大是几十倍
5. 写效率低

Tiering

1. 每个level维护T个组件
2. 同一level中kv重复
3. 每一层写放大为1
4. 总写放大为几倍
5. 读效率低

LSM Partition

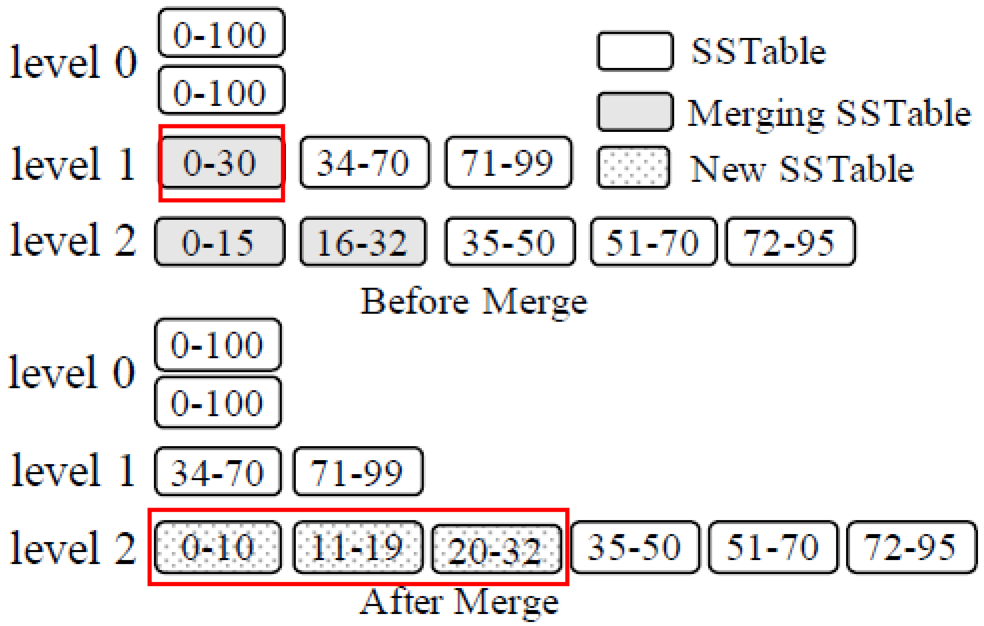


Fig. 4: Partitioned Leveling Merge Policy

Leveling Partition

1. 当前经典架构
2. 大组件依据范围划分为sstable
3. 降低compaction时间
4. 降低新组件临时空间
5. 降低冷数据compaction频率

LSM partition

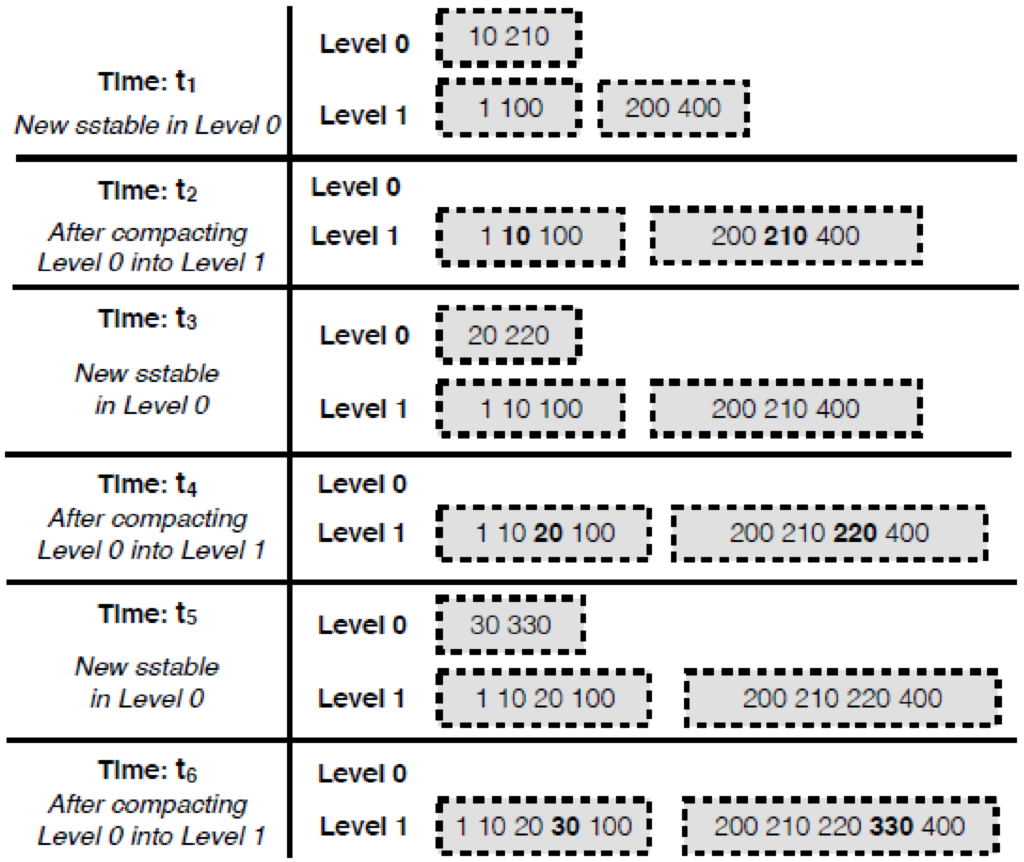


Figure 2: LSM Compaction. The figure shows sstables being inserted and compacted over time in a LSM.

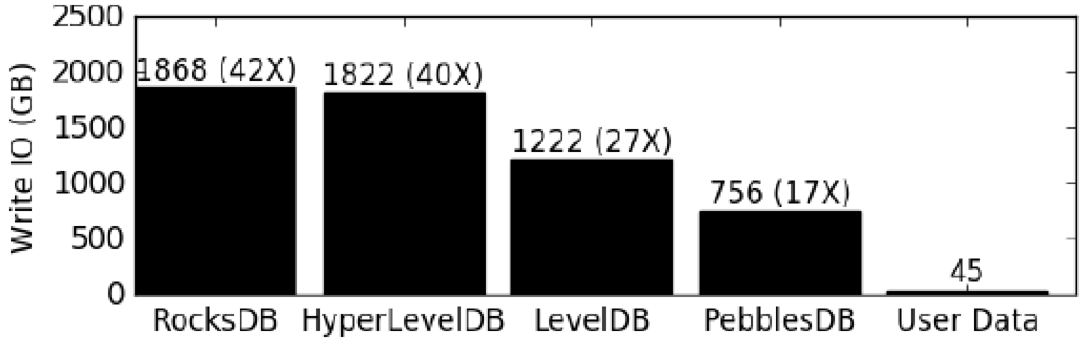
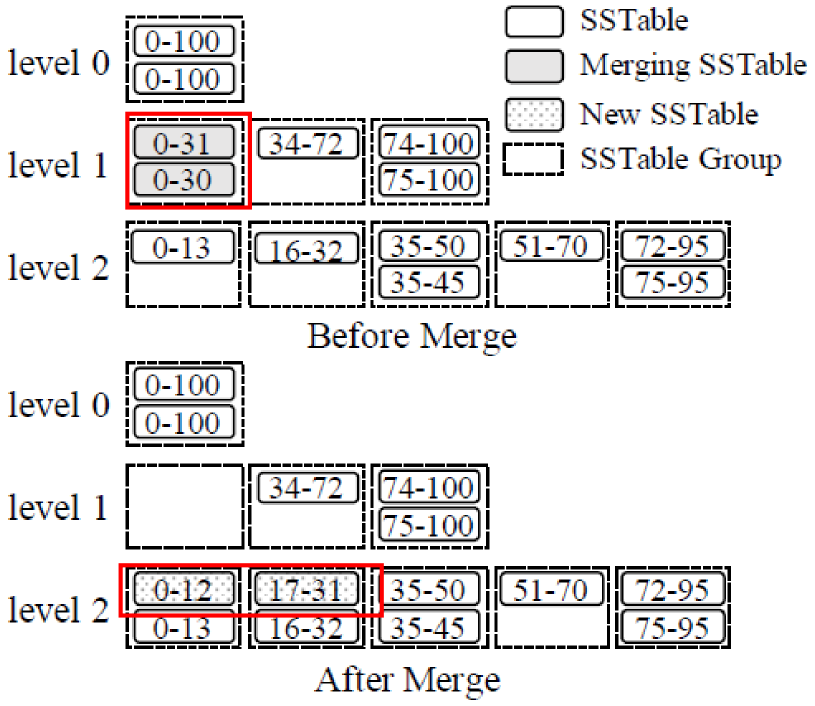


Figure 1: Write Amplification. The figure shows the total write IO (in GB) for different key-value stores when 500 million key-value pairs (totaling 45 GB) are inserted or updated. The write amplification is indicated in parenthesis.

Leveling Partition

1. 依然有太多垃圾回收
2. 较大写放大，消耗资源
3. 降低ssd寿命

LSM partition



(a) Partitioned Tiering with Vertical Grouping

Tireing Partiton

1. 写放大为1
2. 但压缩数据量大大减少
3. Sstable大小不固定
4. 组间有序且不重叠

LSM 基础

总结

为了提高写吞吐量
原地更新->异地更新

→

为了提高读效率
compaction

↓

Compaction造成
写放大

←

写放大降低吞吐量

Leveling->Tiering

问题：多少工业数据库使用Tiering？Tiering难点在哪？

目录

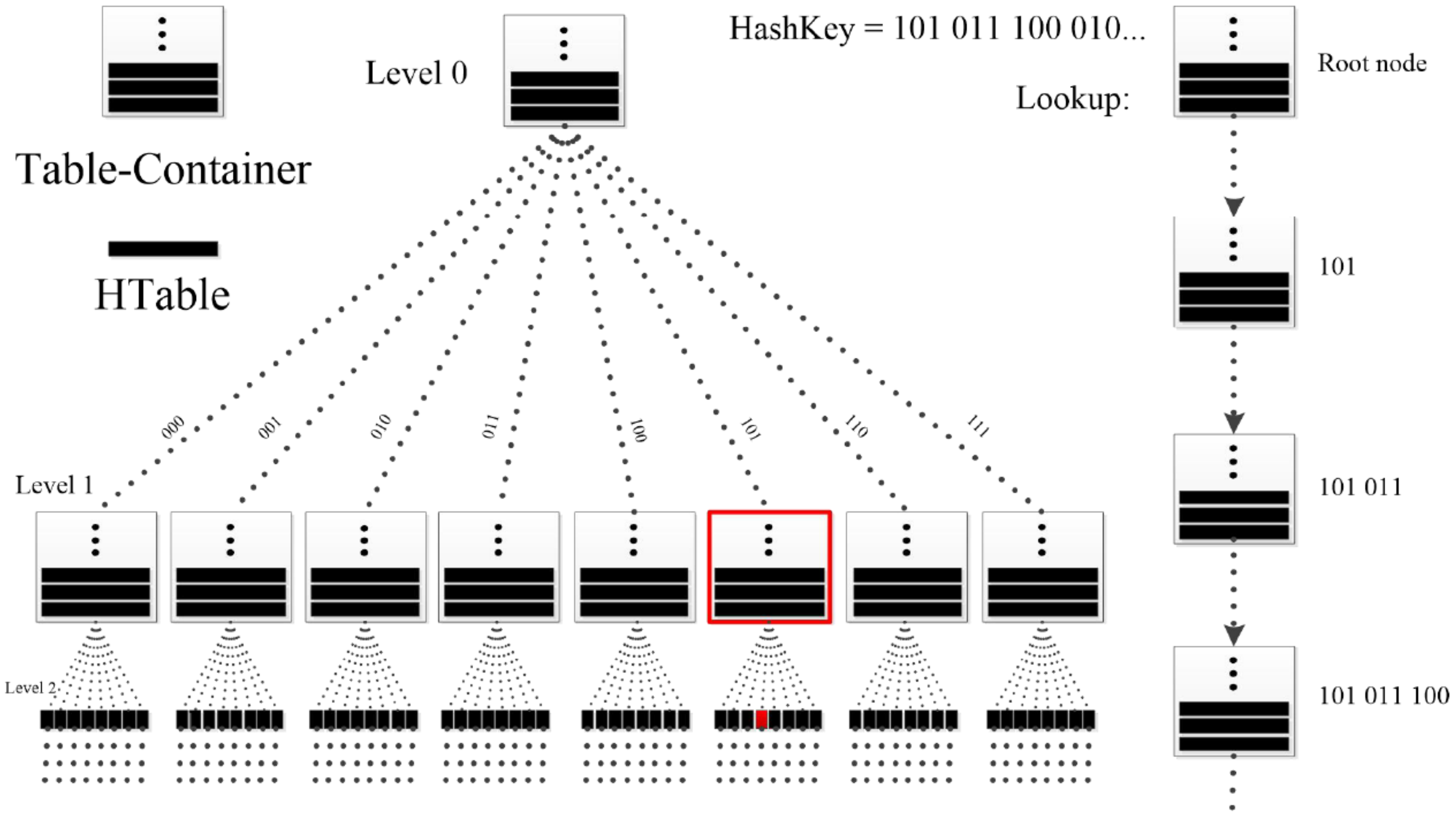
- LSM基础与写放大
- 写放大的优化-tiering
    - LSM-trie (2015)
    - PebblesDB (2017)
    - WipDB (2021)
- 总结

LSM-trie

LSM-trie=LSM+hash+前缀树

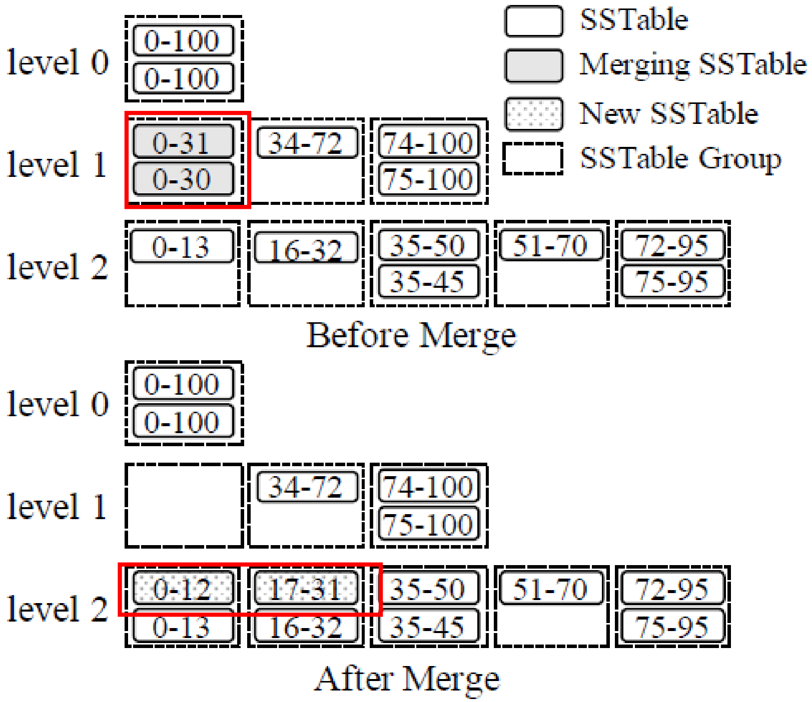
+tiering:

1. 使用SHA-1对key值进行hash
2. 使用hash值决定key属于哪一个节点
3. 多level->trie
4. trie同一深度视为level
5. 一个节点有多个sub_level
6. 节点间独立，且不重叠
7. 无索引



LSM-trie: An LSM-tree-based Ultra-Large Key-Value Store for Small Data

LSM-trie



(a) Partitioned Tiering with Vertical Grouping

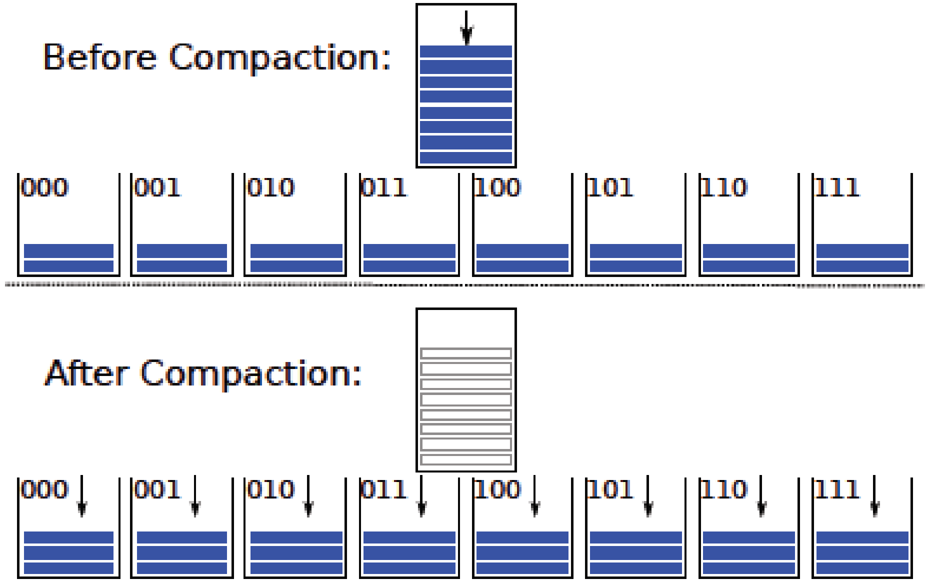
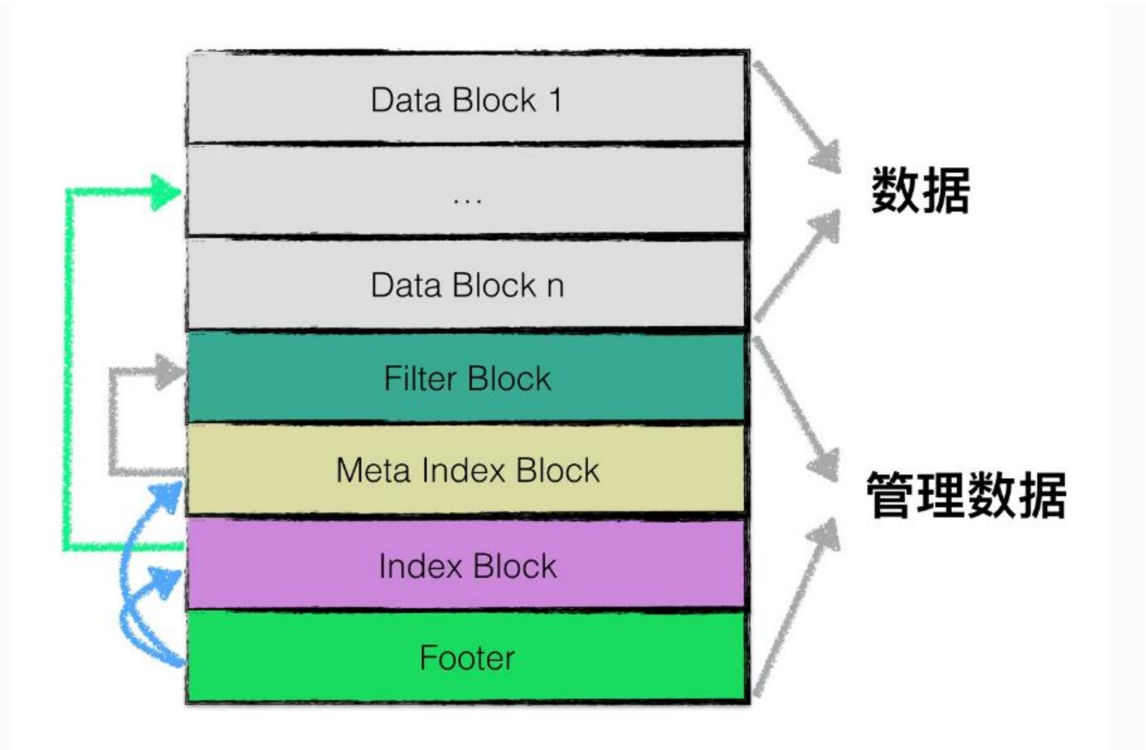
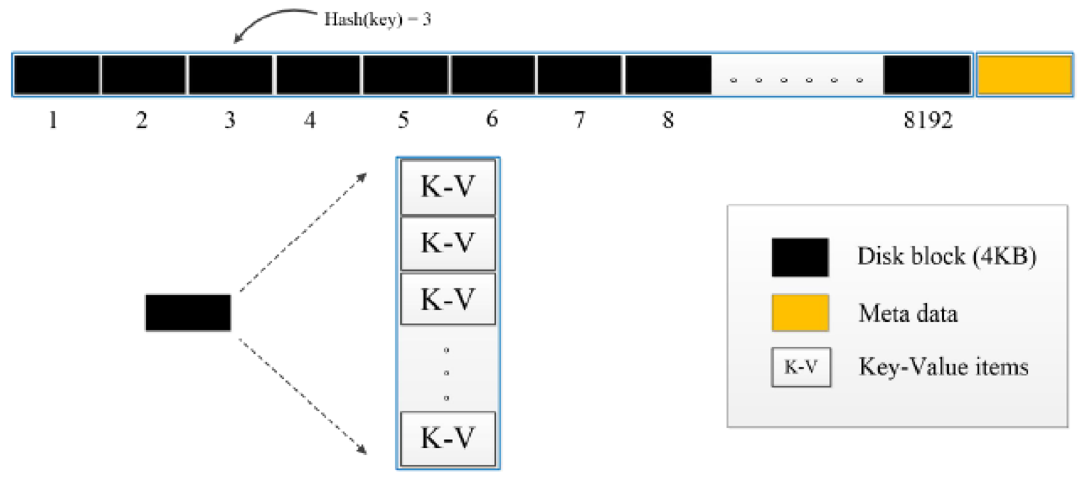


Figure 3: *A compaction operation in the trie.*

LSM-trie

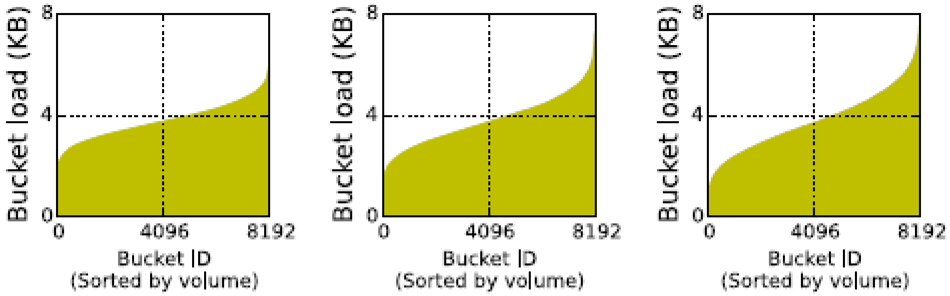




- 一个Htable有m个block;
- Kv的hashkey=h
- Kv所属block=h mod m

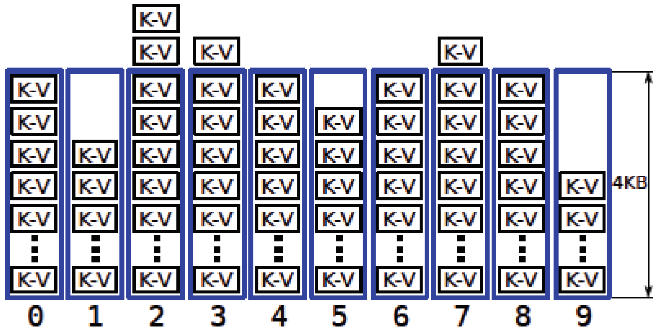
Block是固定大小，block负载是否均衡?

LSM-trie



(a) *100 B* (b) *200 B* (c) *300 B*

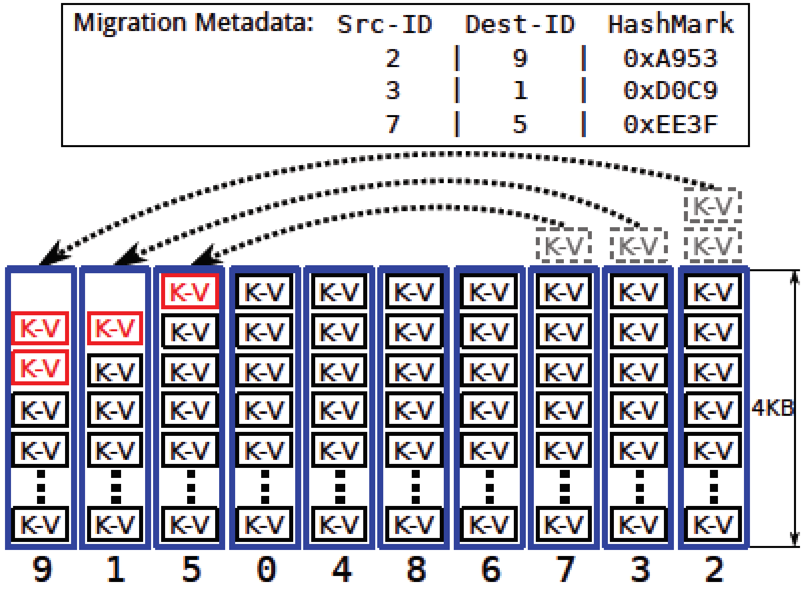
Figure 5: *Distribution of bucket load across buckets of an HTable with a uniform distribution of KV-item size and an average size of 100 B (a), 200 B (b), and 300 B (c). The keys follow the Zipfian distribution. For each plot, the buckets are sorted according to their loads in terms of aggregate size of KV items in a bucket.*



(a) *KV items are assigned to the buckets by the hash function, causing unbalanced load distribution.*

- 数据分布不稳定
- Kv平均值越大越分布越不均匀

LSM-trie



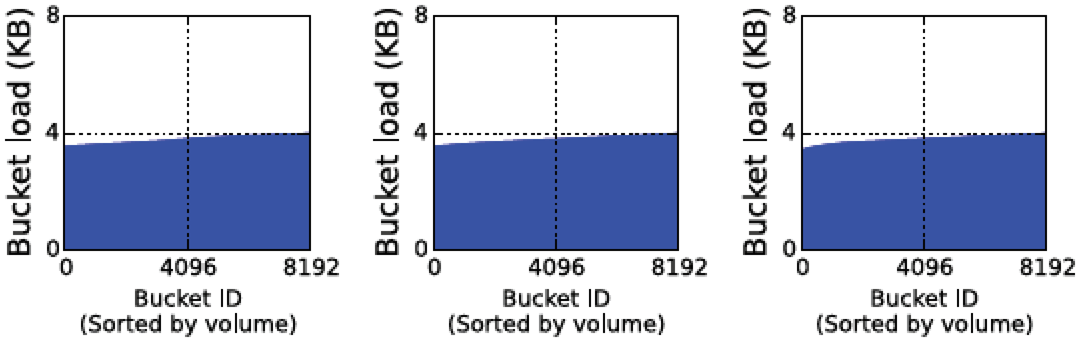
(b) *Buckets are sorted according to their loads and balanced by using a greedy algorithm.*

Figure 6: *Balancing the load across buckets in an HTable.*

解决block负载不均匀：

- 将高负载压力转移到低负载block
- 贪心思想
  1. 对初始负载量排序
  2. 总是将负载最大的压力给负载最低的
- 允许跨多block转移
- 大kv使用专用block

LSM-trie



(a) *100 B* (b) *200 B* (c) *300 B*

Figure 7: *Bucket load distribution after load balancing for HTables with different average item sizes.*

LSM-trie

优点

1. 简单粗暴，但有效
2. 减少写放大、提高写吞吐量
3. 较高的读效率
4. 低内存占用

缺点

1. 不适合大kv
2. 不支持范围查询

如何支持范围查询?

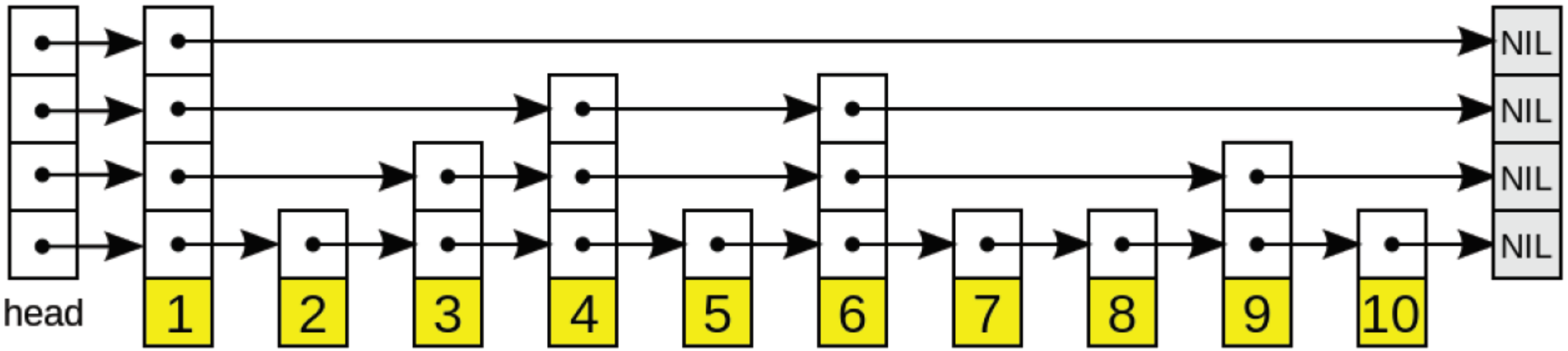
目录

- LSM基础与写放大
- 写放大的优化-tiering
  - LSM-trie
  - PebblesDB
  - WipDB
- 总结

PebblesDB

SkipList：随机化存储的多层线性链表结构

1. 按层构建，底层存储所有数据。
2. 上层充当底层数据“快速通道”。
3. 利用概率均衡技术，简化插入、删除操作，保证绝大多操作拥有O(logn)的良好效率。



PebblesDB: Building Key-Value Stores using Fragmented Log-Structured Merge Trees

PebblesDB

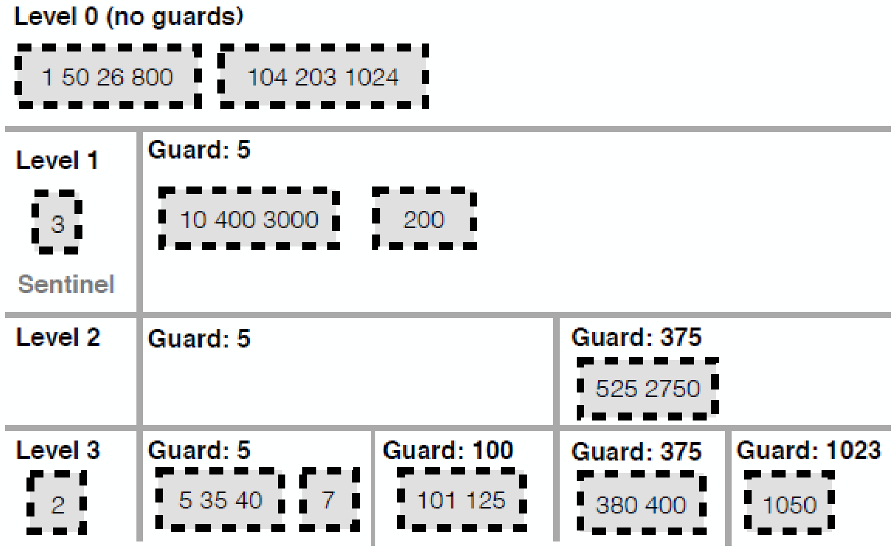


Figure 3: FLSM Layout on Storage. The figure illustrates FLSM's guards across different levels. Each box with dotted outline is an sstable, and the numbers represent keys.

FLSM (Fragmented LSM) : LSM + SkipList

1. 应用partition + tiering思想。
2. 核心：使用guard管理数据，实现partition
3. 一个Guard对应一个key，Guard将数据分为不相交部分
4. 每个Guard关联一个key范围，Gi->Ki、Gi+1->Ki+1，则Gi=>[Ki,Ki+1)
5. 每层多个Guard，越下层Guard越多，粒度越细
6. 上层Guard一定在下层出现

PebblesDB

Select guard:

1. 非静态
2. 使用概率函数防止数据分布不均匀
3. gp(key,i)概率选择key是否是level i 的guard
4. 层数越大概率越大
5. Lecel i+1的guard是level i的超集
6. Guard选择可以是不存在的key

问题：没有考虑IO大小

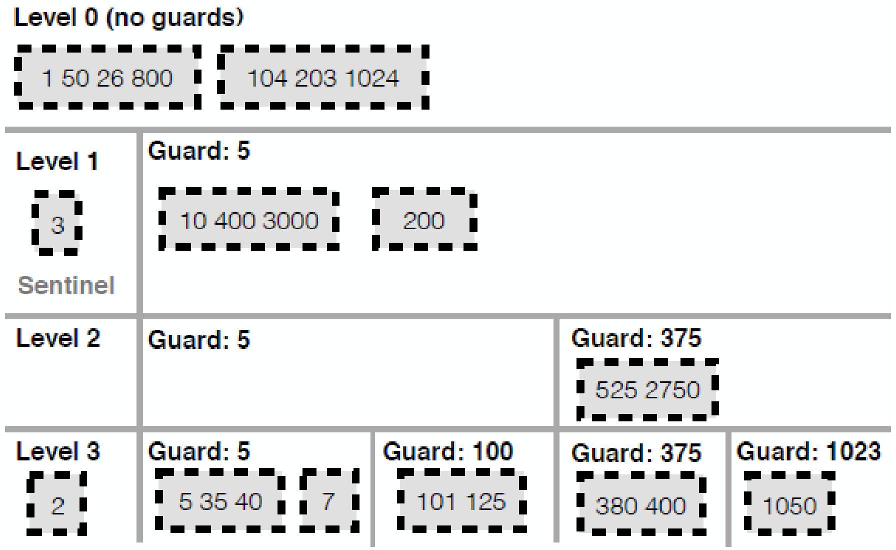


Figure 3: FLSM Layout on Storage. The figure illustrates FLSM's guards across different levels. Each box with dotted outline is an sstable, and the numbers represent keys.

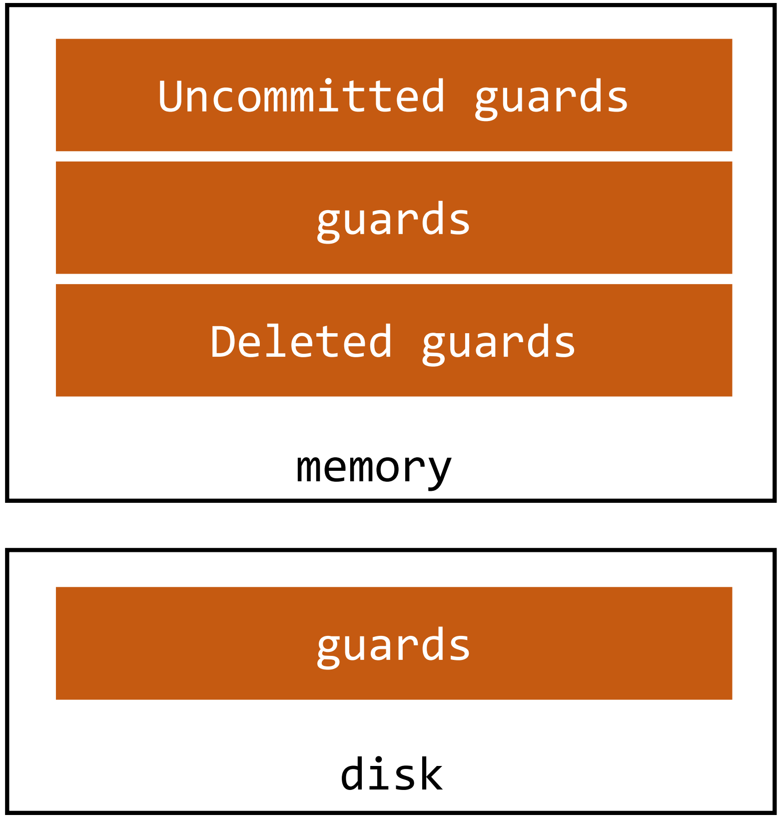
PebblesDB

Insert guard:

1. 异步insert
2. Uncommitted guard不影响FLSM的读写
3. Compation时新guard生效

Delete guard:

1. Key range为空、数据分布不均匀
2. 异步delete
3. 下层level不一定delete



PebblesDB

Compaction:

1. Level大小达到阈值
2. sstable数目达到阈值
3. Seek()次数达到阈值
4. 多线程compaction
5. 充分利用SSD多读写通道
6. 增加读吞吐量

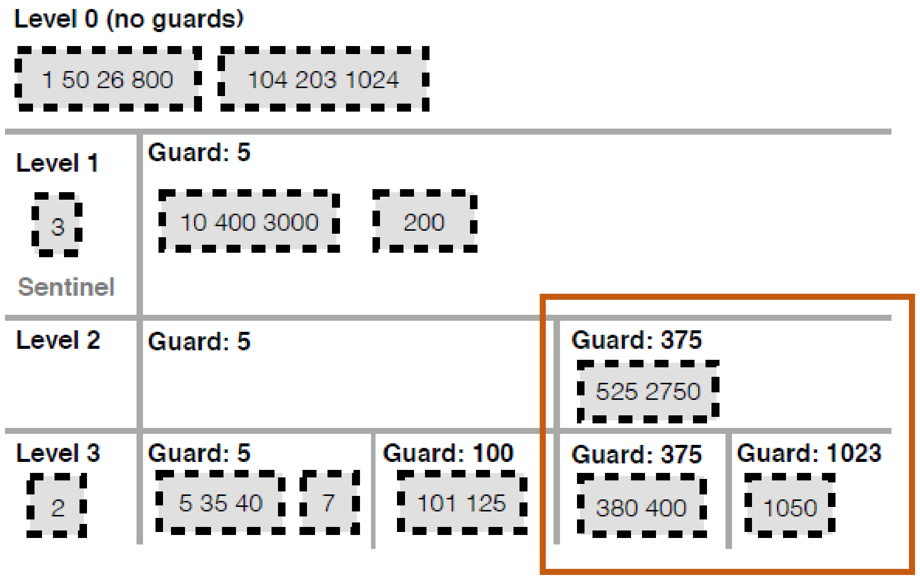


Figure 3: FLSM Layout on Storage. The figure illustrates FLSM's guards across different levels. Each box with dotted outline is an sstable, and the numbers represent keys.

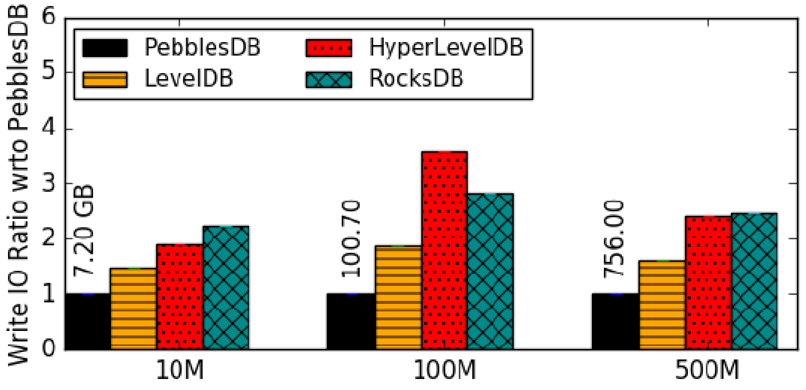
PebblesDB-评估

优点：

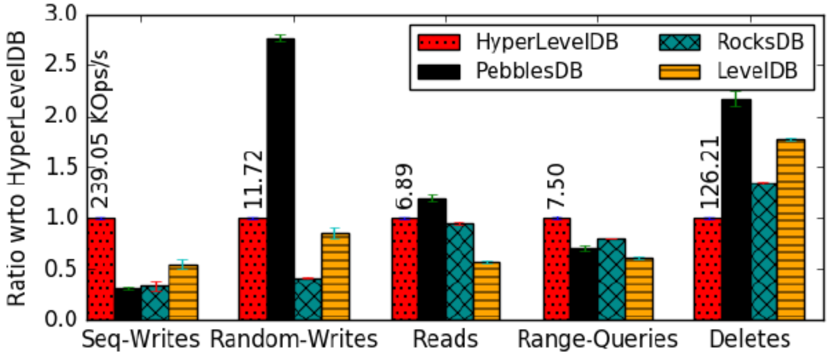
1. 降低写放大，提高写吞吐量
2. 提高并发度，并行compaction、并行查找
3. 细粒度compaction

缺点：

1. 范围读性能下降
2. Guard的频繁变动
3. 增加维护guard的内存消耗
4. 不适合顺序写入数据



(a) Write Amplification

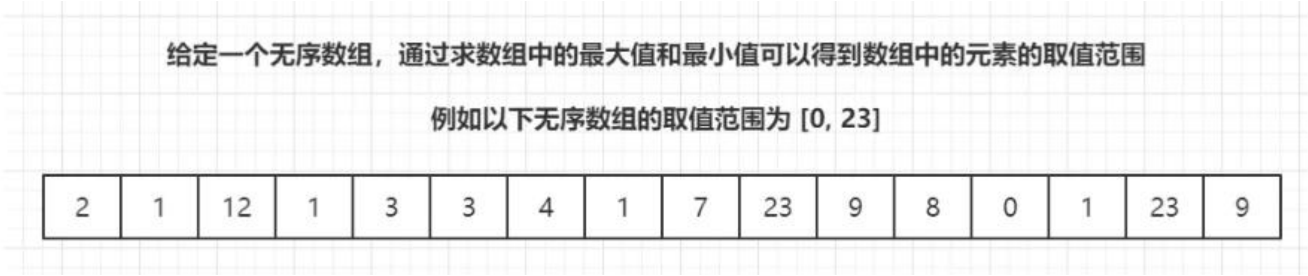


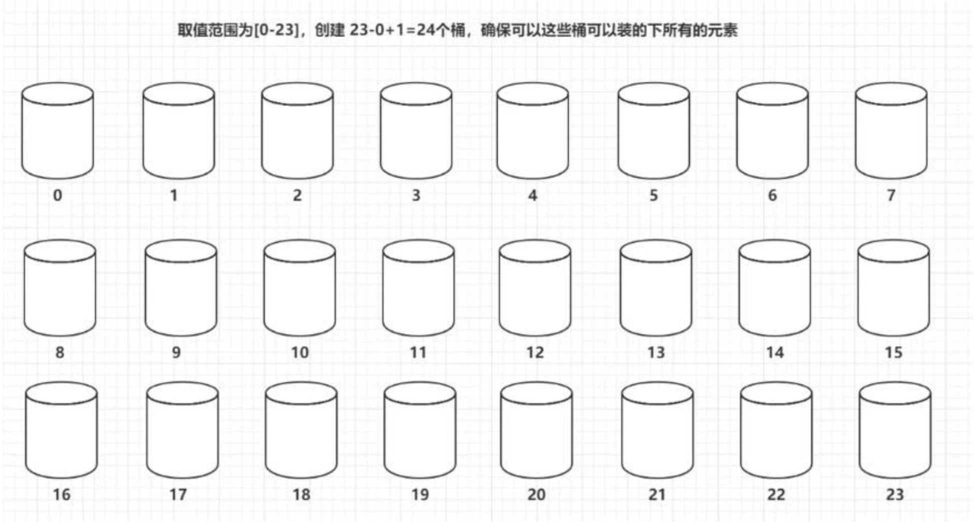
(b) Single-threaded operations

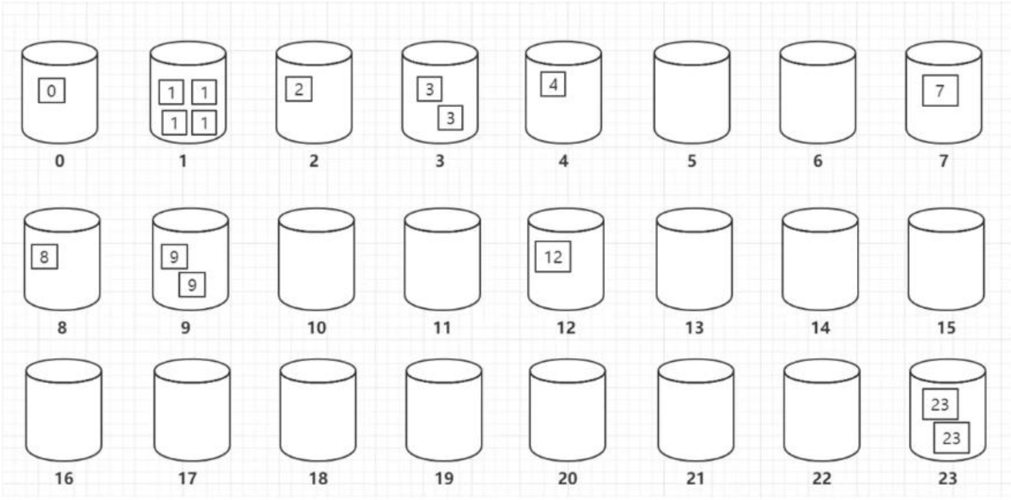
目录

- LSM基础与写放大
- 写放大的优化-tiering
    - LSM-trie
    - PebblesDB
    - WipDB
- 总结

WipDB-桶排序







WipDB: A Write-in-place Key-value Store that Mimics Bucket Sort

WipDB

WipDB(Write-in-place):

1. 模拟桶排序对key partition
2. 桶间有序，桶内无序
3. 多Memtable
4. 桶内使用小LSM维护，且不再partition
5. Tiering减少写放大
6. 并行调度

可能的问题：

1. 频繁重新分桶导致重写数据
2. 依赖桶空间的稳定性
3. 要求桶空间的负载均衡

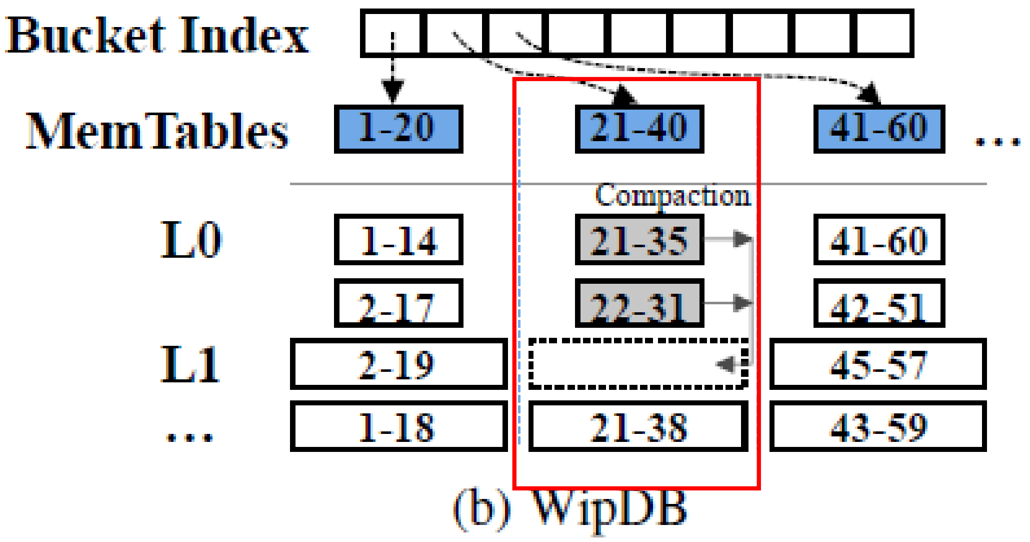


Fig. 1: *Architectures of LSM tree and WipDB*

WipDB

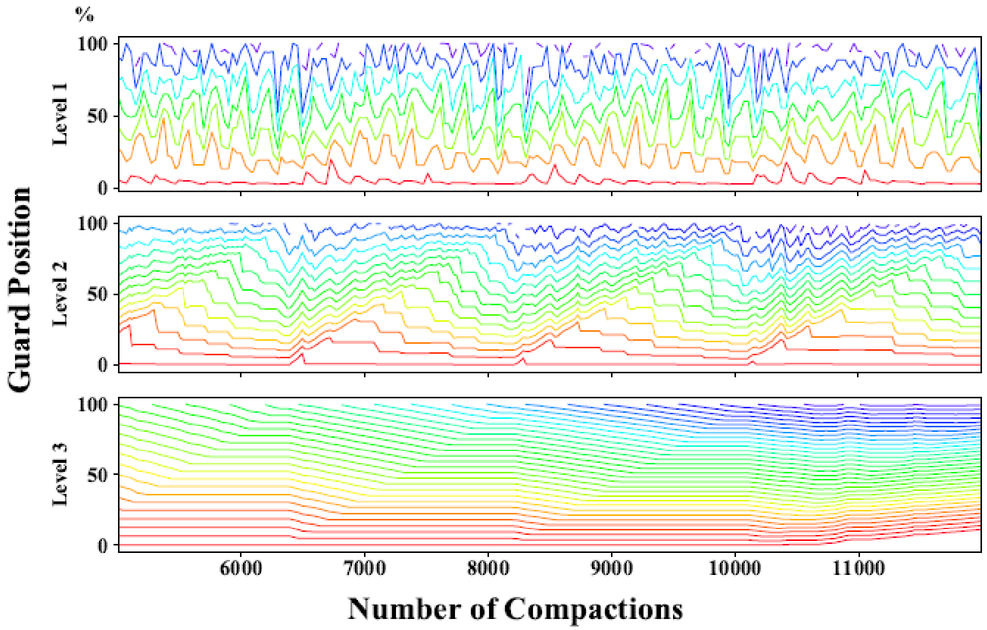


Fig. 2: *Guard positions in different levels in LevelDB (L1, L2, and L3) after certain number of compactions in the system. The position is expressed as a percentage of the guard key in the entire key space* ($0..10^9$)*. A workload with the uniform distribution is used here.*

平均kv 100b，持续插入10亿条数据

桶的稳定是关键：

1. Key分布越稳定桶范围越稳定
2. 现实世界，一定时间范围数据相对稳定
    1. 商品的分类、数量、受欢迎程度等
3. LSM上层空间小，生命周期短、key分布不稳定
4. LSM总是将数据不断合并至底层
5. 底层数据量大，抵消临时的数据激增，更稳定

仅依据最后一层作为桶空间划分依据

WipDB-bucket space split&merge

平衡bucket：

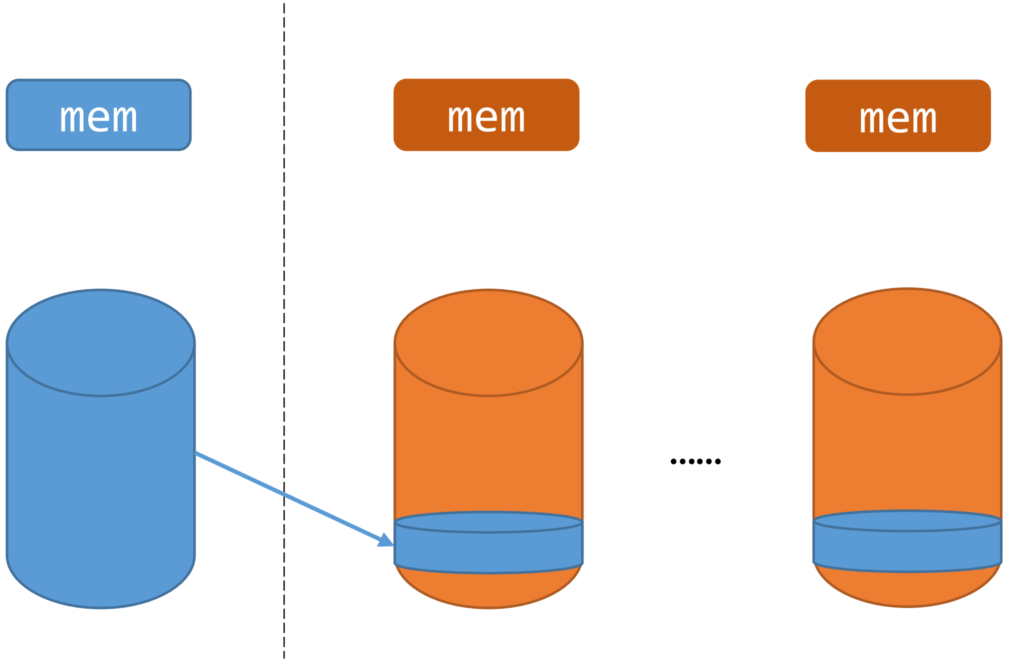
1. 初始化只有一个桶
2. 桶容量具有上限

Split：

1. Split的最小单位是桶
2. 抽样split
    1. 从L*T*(N-1)个样本中选择N-1个guard
    2. L：level数；T：sublevel数；N：每次桶划分为N个桶
3. 异步split，不影响旧数据请求
4. 最后为新桶分配新的memtable

Merge：

1. Memtable消耗内存
2. 小桶与邻桶合并



WipDB-compaction

Compaction触发:

1. Level大小达到阈值
2. Sublevel数量达到阈值，Sub_count
3. Subtable访问次数达到阈值，Read_count [min_count,max_count]
4. 优先级调度
    1. 动态更新优先级，把资源给最需要的数据
    2. P=read_weight*rela_read_count+rela_sub_count
    3. Read_weight可调节，更好的平衡桶资源

WipDB

优点：

- Tiering+桶（小LSM）
- 减少写放大、提高写吞吐量
- 动态优先级调节compaction
- 并发调度
- 更稳定的key分布策略

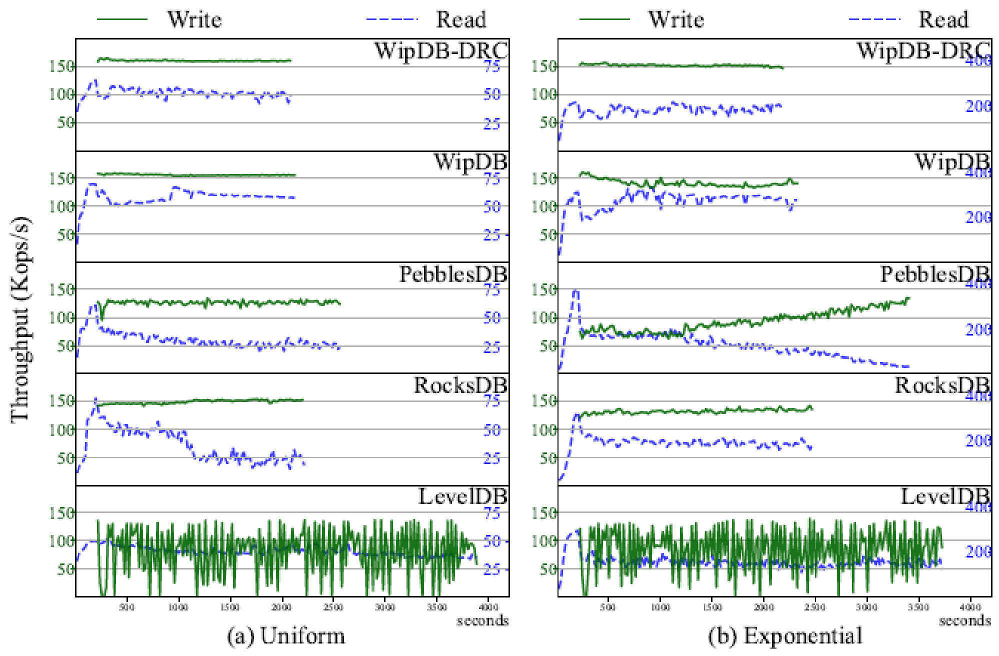


Fig. 8: *Throughput with mixed read/write requests. One thread sends 300 million random (uniform) write requests at the rate of 150 Kops/s(if possible). Eight threads send read requests until all the writes finish. WipDB that Disable Read-aware Compaction is marked as 'WipDB-DRC'.*

目录

- LSM基础与写放大
- 写放大的优化-tiering
    - LSM-trie
    - PebblesDB
    - WipDB
- 总结

总结

	Lsm-trie	PebblessDB	WipDB
思想	Tiering+Partition		
核心	hash	skiplist	Buffer sort
如何分区	前缀树	Guard分区	桶分区
优点	快速读写 减少内存占用	并发调度	并发调度 动态优先级
缺点	不支持范围查找 不适合大kv	消耗内存 分区变动频繁	消耗内存

总结

降低写放大

- Tiering策略
  - Partition
  - 混合leveling、Tiering
- 优化compaction
  - 跨level合并
  - 冷热数据区分
  - IO调度
- KV分离

谢谢