Leveldb 原理解析

吴德妙

2018/01/05

demiaowu@163.com

Outline

- Introduction
- Design and Implementation
- Problems
- Improvement and Optimization
- Summary
- References

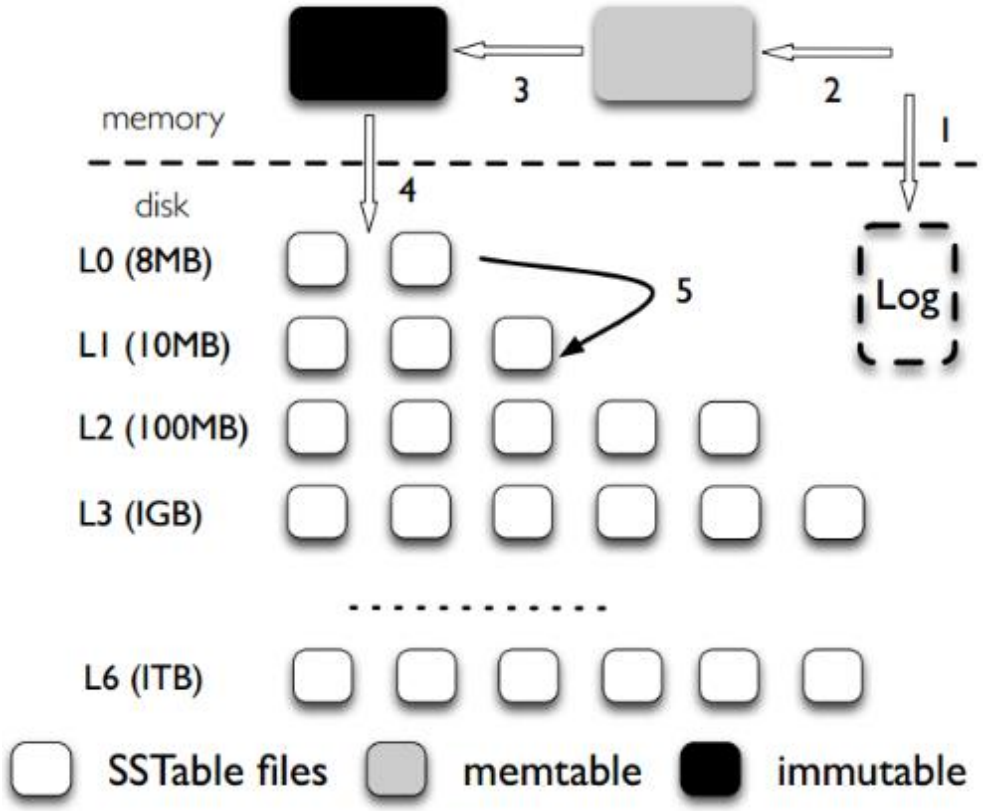
Introduction

Introduce

- **Key-value library**
  - 高效的轻量级的key-value存储库
- **Open source**
  - https://github.com/google/leveldb
- **Made In Google**
  - Authors，also develop GFS\MapReduce\Bigtable\Protocol Buffers

Architecture

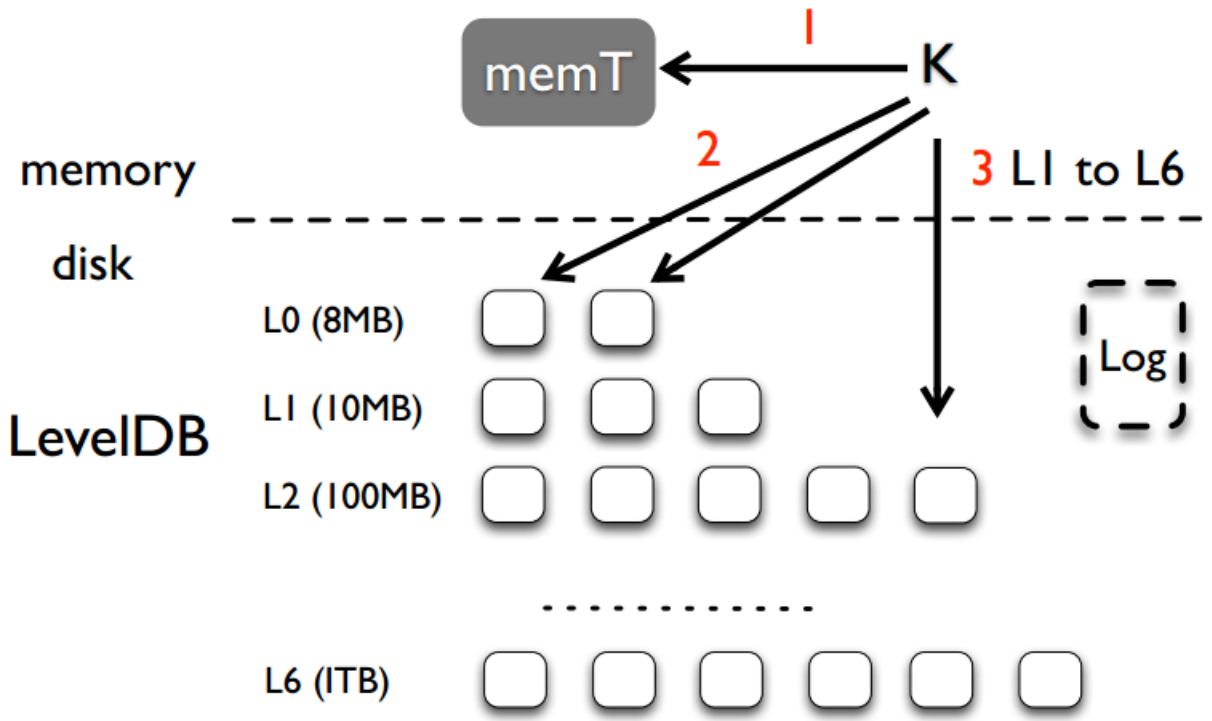
- Log（Write Ahead Log, WAL）：写Memtable前会先写log文件，log通过append的方式顺序写入
- Memtable：内存数据结构，跳跃表实现
- Immutable Memtable：达到Memtable设置的容量上限后，memtable会变为immutable，不再接受用户写入，同时会有新的Memtable生成；
- SSTable files（sst）：磁盘数据有序存储文件。分为Level 0到Level n多层，每一层包含多个SST文件；单个SST文件容量随层次增加成倍增长；其中level0的SST文件由immutable直接dump产生，其他level的SST文件由其上一层的文件和本层文件归并产生；



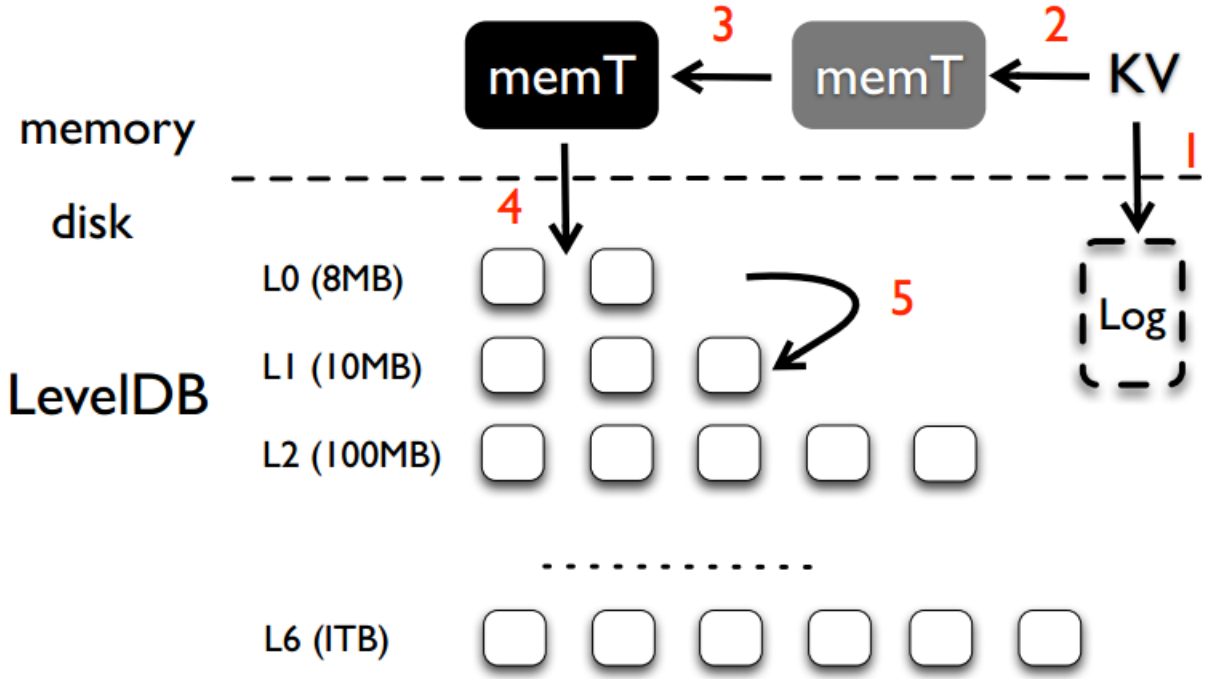
Basic operation

- 读
  - `db->Get(leveldb::WriteOptions(), key1, value);`
- 写
  - `db->Put(leveldb::WriteOptions(), key2, value);`
- 删
  - `db->Delete(leveldb::WriteOptions(), key1);`

Read


memT
K
1
2
3 L1 to L6
memory
disk
LevelDB
L0 (8MB)
L1 (10MB)
L2 (100MB)
L6 (1TB)
Log

Write


memT
3
memT
2
KV
memory
disk
1
4
L0 (8MB)
5
Log
LevelDB
L1 (10MB)
L2 (100MB)
L6 (1TB)

Delete

- 延迟删除
  - 标记删除，插入删除标记记录
  - Compaction的时候，删除标记和真正数据相遇，才会执行删除

Design and Implementation

Random vs Sequential（I/O）

- **HDD**
  - 顺序读写 100x 随机读写
- **读优化**
  - Cache（局部性）
- **写**
  - ???

Log Struct Merge Tree

- C0树：内存
- C1树：磁盘

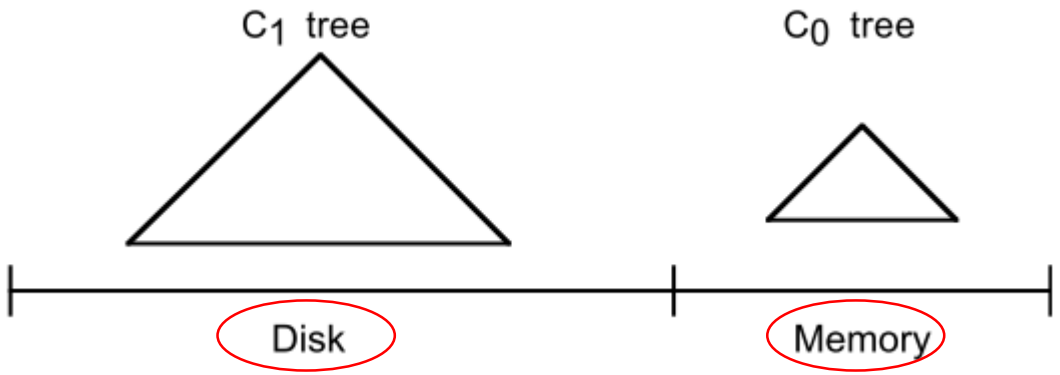
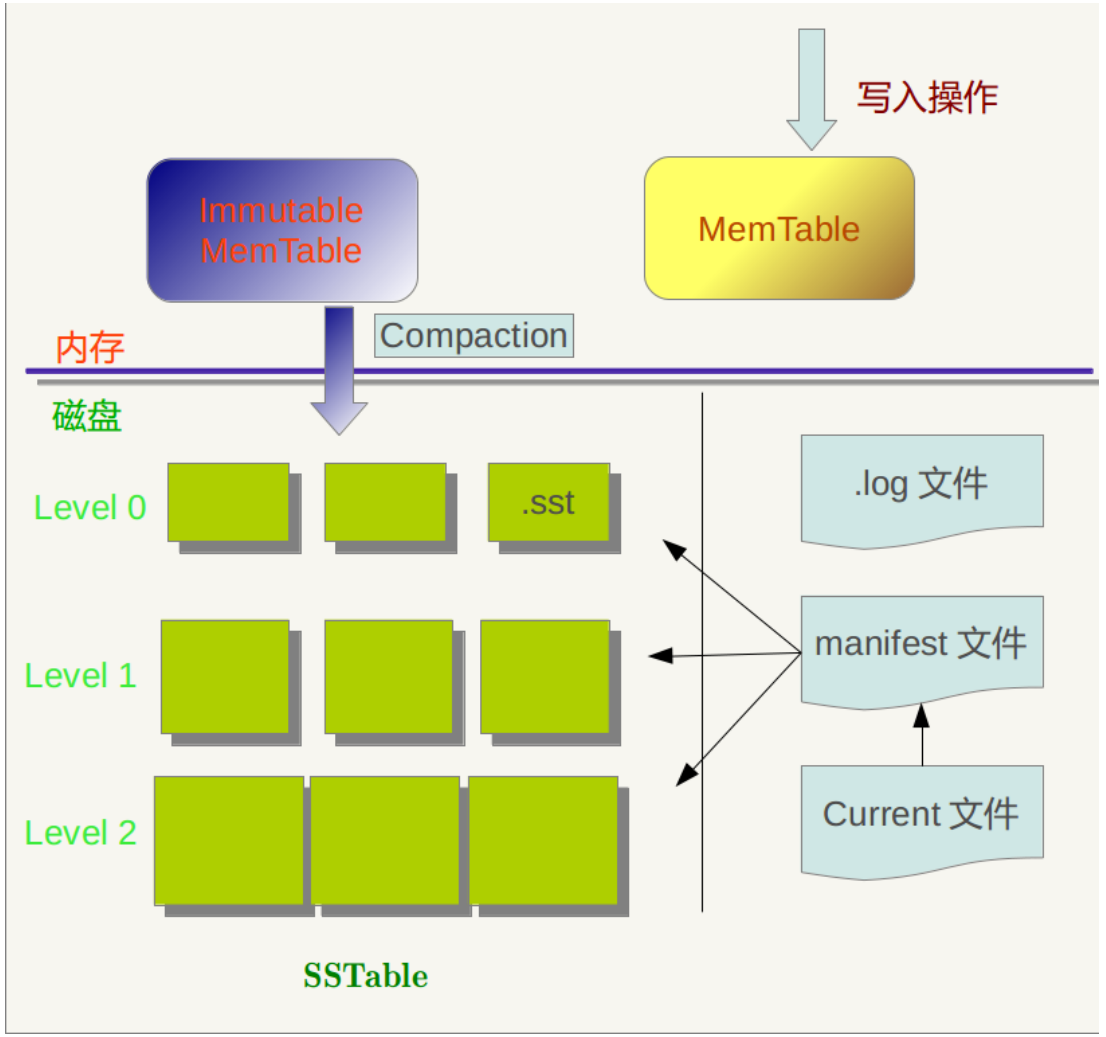


图1：两部件的LSM-tree

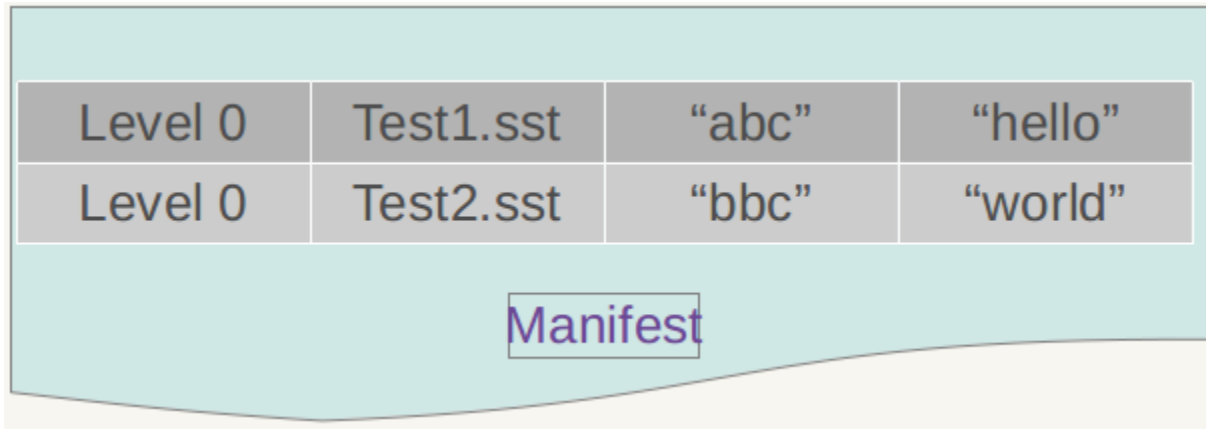
优化写入：随机写入 -> **批量顺序写入**（C0->C1）

Overview


写入操作
Immutable
MemTable
MemTable
Compaction
内存
磁盘
Level 0
.sst
Level 1
Level 2
.log 文件
manifest 文件
Current 文件
SSTable

Manifest

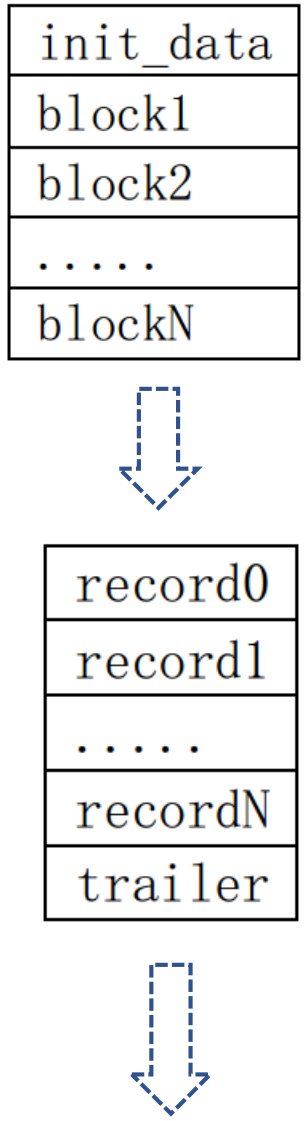
- **Manifest**
  - SSTable各个文件的meta数据，比如属于哪个Level，文件名称叫啥，最小key和最大key各自是多少



Manifest逻辑结构

Log

- **Log**：由32k的Block组成
  - Init data暂时为空
- **Block**：由Records组成
  - Trailer：小于 record 头长度(checksum/length/type)， 填0
- **Record**： 每次更新写入作为一个 record
  - Checksum: crc32校验
  - Length：data部分数据的长度
  - Type： 有四种类型FULL、FIRST、MIDDLE、LAST
    全部、开始、 中间 、 最后
  - Data：| key size | key | value size | value |



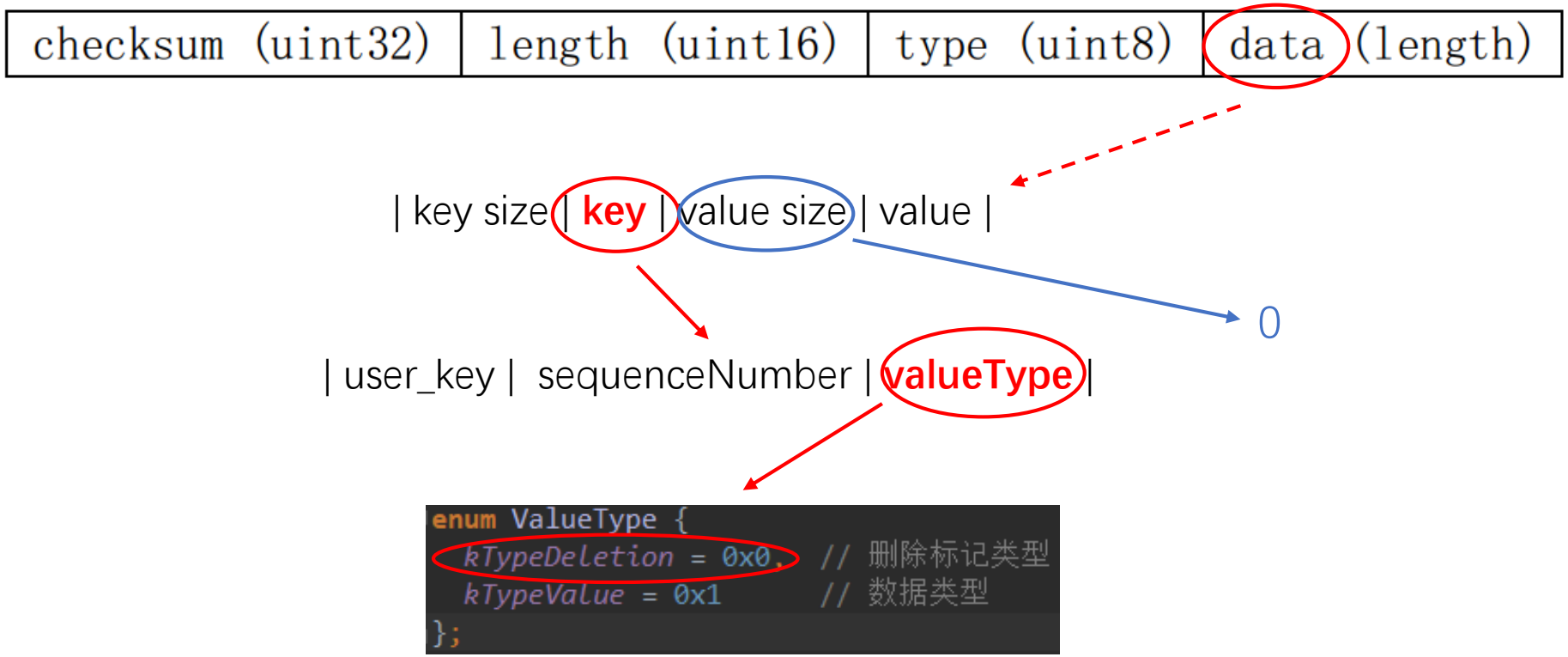
checksum (uint32)	length (uint16)	type (uint8)	data (length)

Log-huge record

- **Record**： 每次更新写入作为一个 record
  - Type： 有四种类型FULL、FIRST、MIDDLE、LAST
    全部、开始、 中间 、 最后
- **Record跨Block**

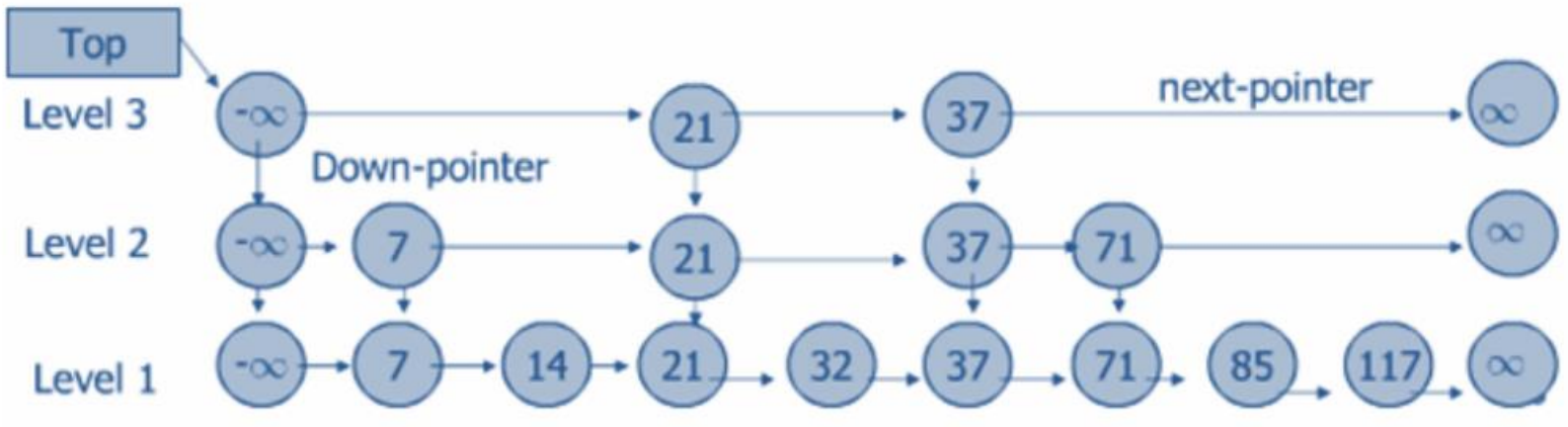
Log-delete record

- **Delete**：
  - 标记删除



Memtable

- **跳跃表**
  - 有序
  - 插入、查找的时间复杂度都是O(logn)（**平衡二叉搜索树**）



Immutable memtable

- **跳跃表**
  - 不可变，不接受写入（会在Minor Compaction中持久化成sst文件）

SStable

- Data block
  - 存储实际的 kv 数据
- Index block
  - 保存每个 data block 的 last key 及其在 sstable 文件中的block handle
- Meta block
  - 在配置filter policy的情况下，会有Bloom Filter Block
- Metaindex block
  - meta index block会有一条key位为filename，value为meta data block handle的偏移量和size。

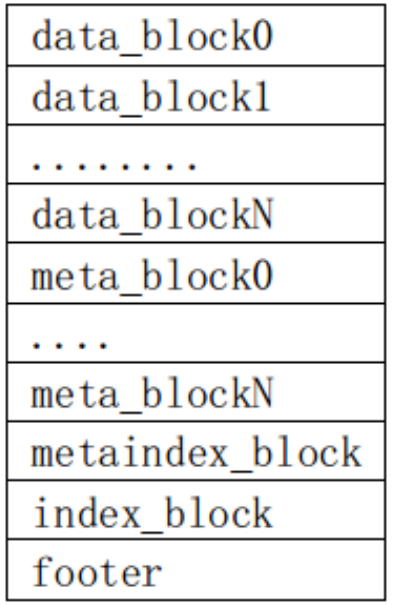
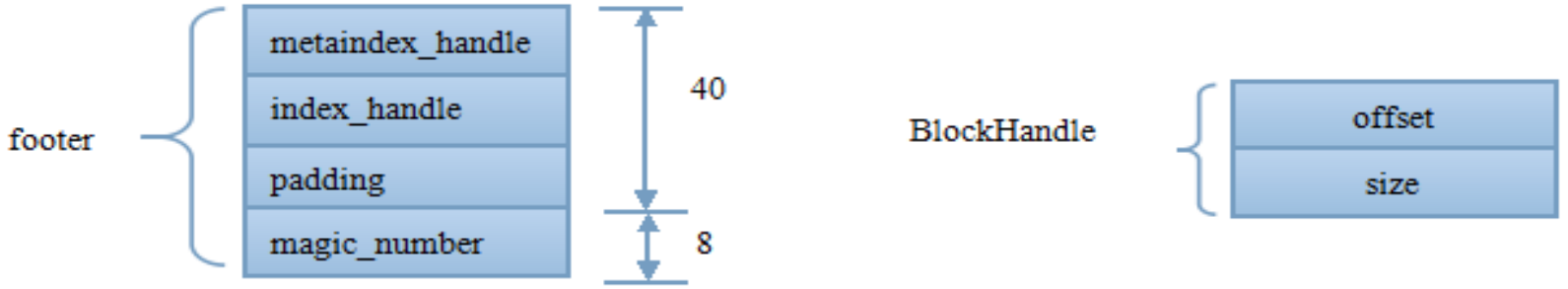


图1. sstable

Sstable-Footer

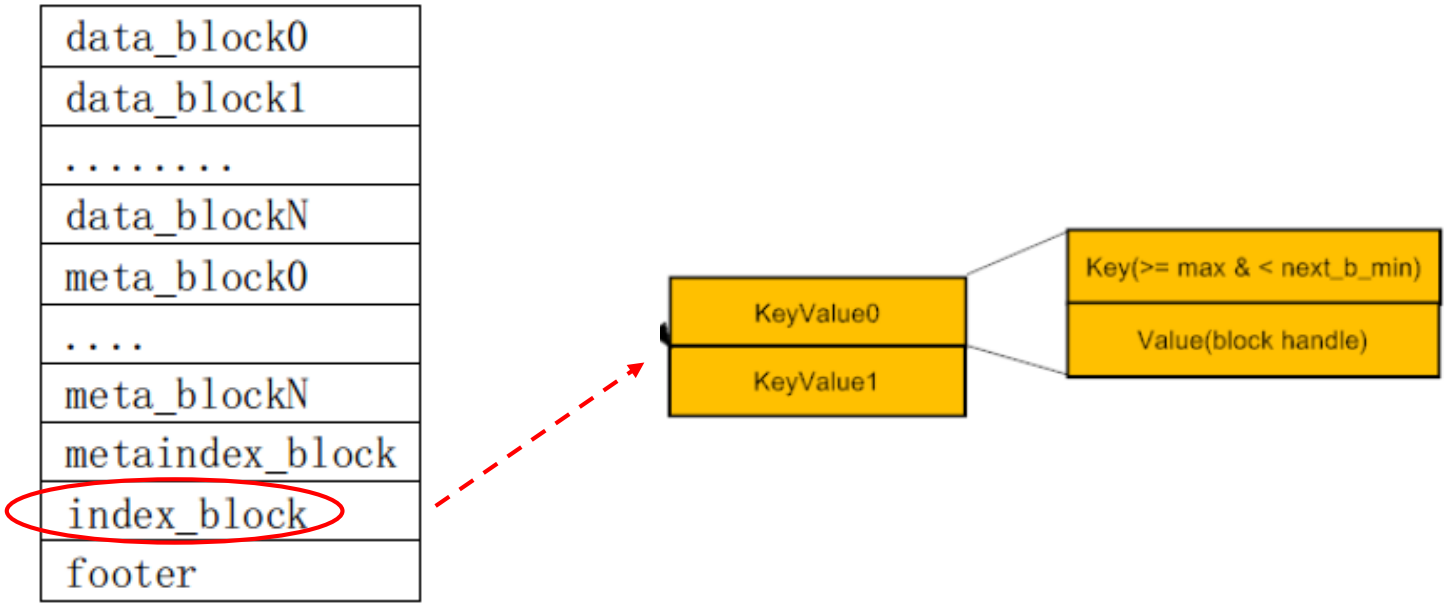
- Footer
  - Block handle是采用的 varint

图1. sstable



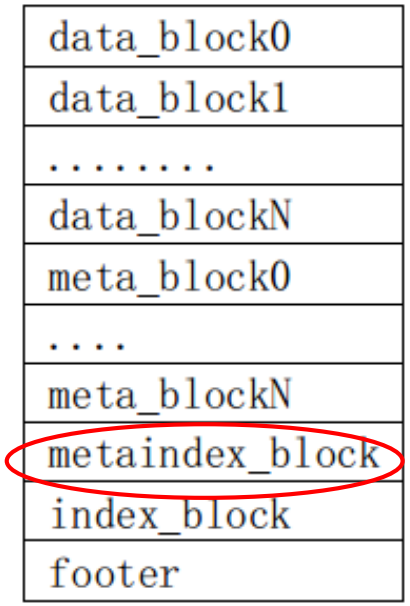
SStable-index block

- **Index block**
  - 每个 data block 的 last key 及其 sstable 文件中的block handle



SStable-meta index block

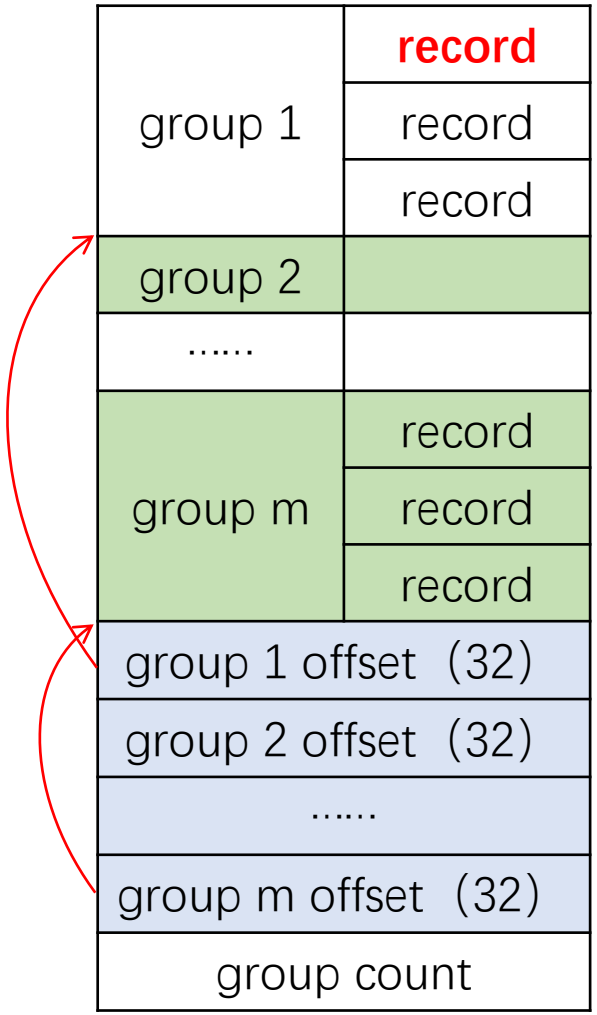
- **Index block**
  - 每个 meta block 的 key 及其在sstable 文件中的block handle



Sstable-Data Block

data block
data block
data block

data
Compress type（char）
crc32（uint32）

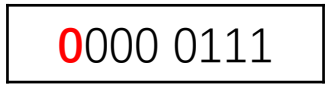


与group的第一条record的key对比

shared key size (varint32)
non-shared key size (varint32)
value size (varint32)
Non-shared key
value

SStable-varint

- **变长整型**
  - 一个字节的8个位中
  - 最高位为特殊位：1，表示后续一个字节还是该数字的一部分，0表示结束
  - 其他7位表示数字
  - < 128的数字可以用一个字节表示
  - 268435455 < x < 2^32-1 需要5个字节表示

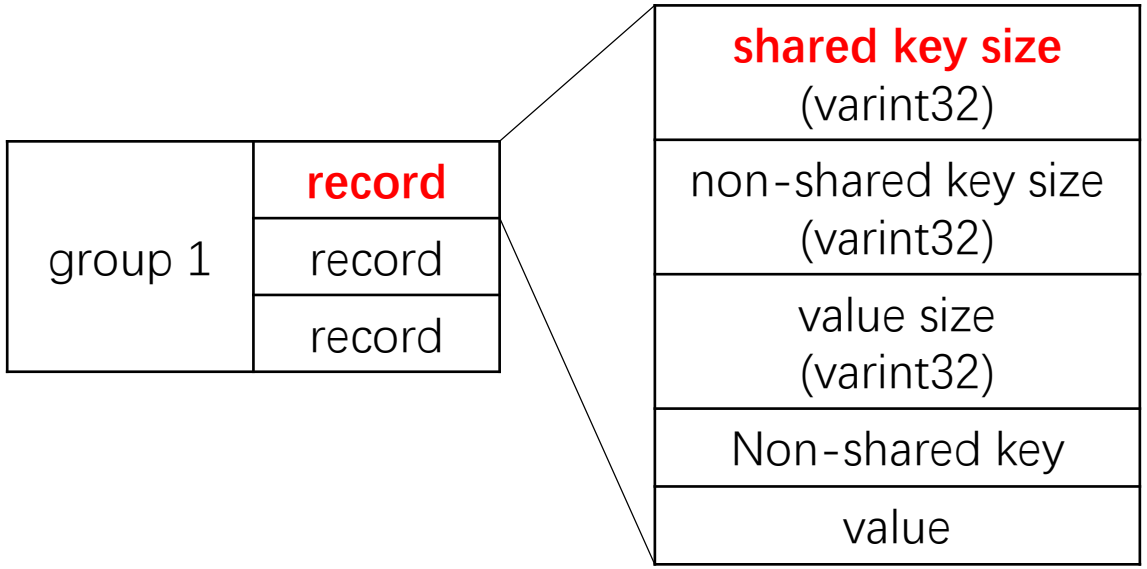


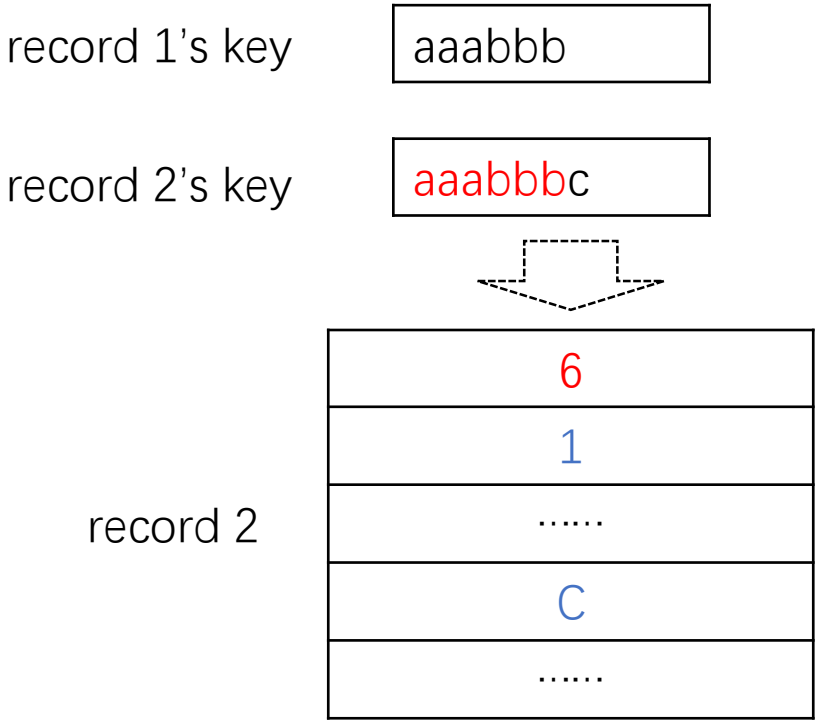
varint32示例：7

varint32示例：300

SStable-prefix compression

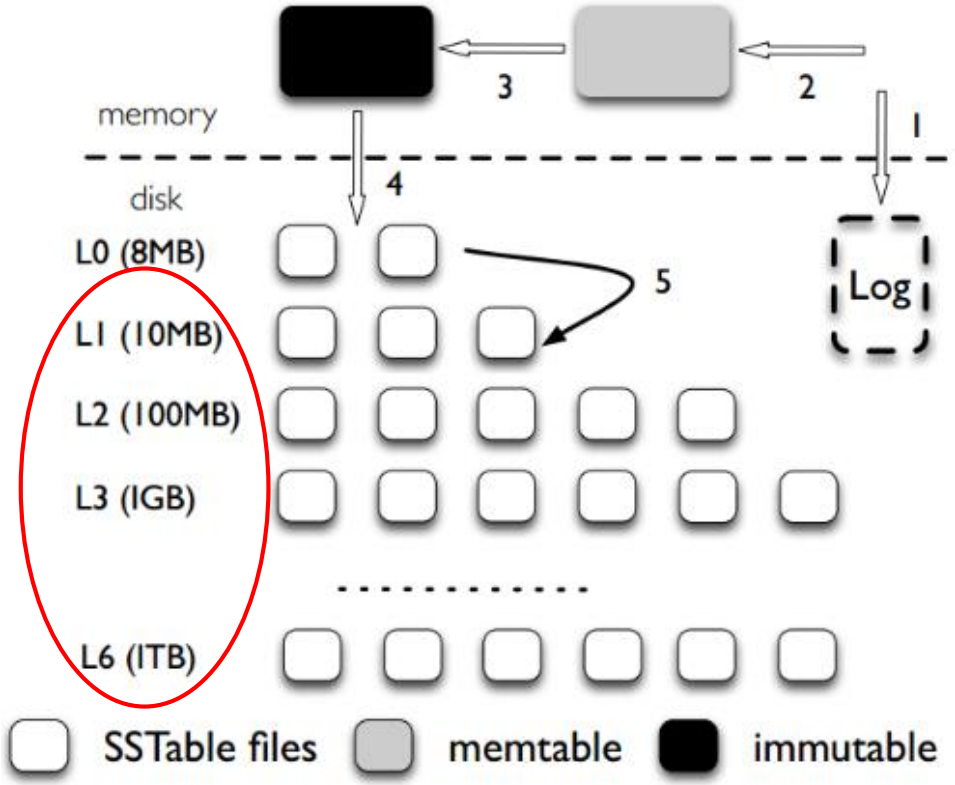
- **前缀压缩**
  - 一个group 里面的key做前缀压缩
  - shared key是与group的第一条record的key对比





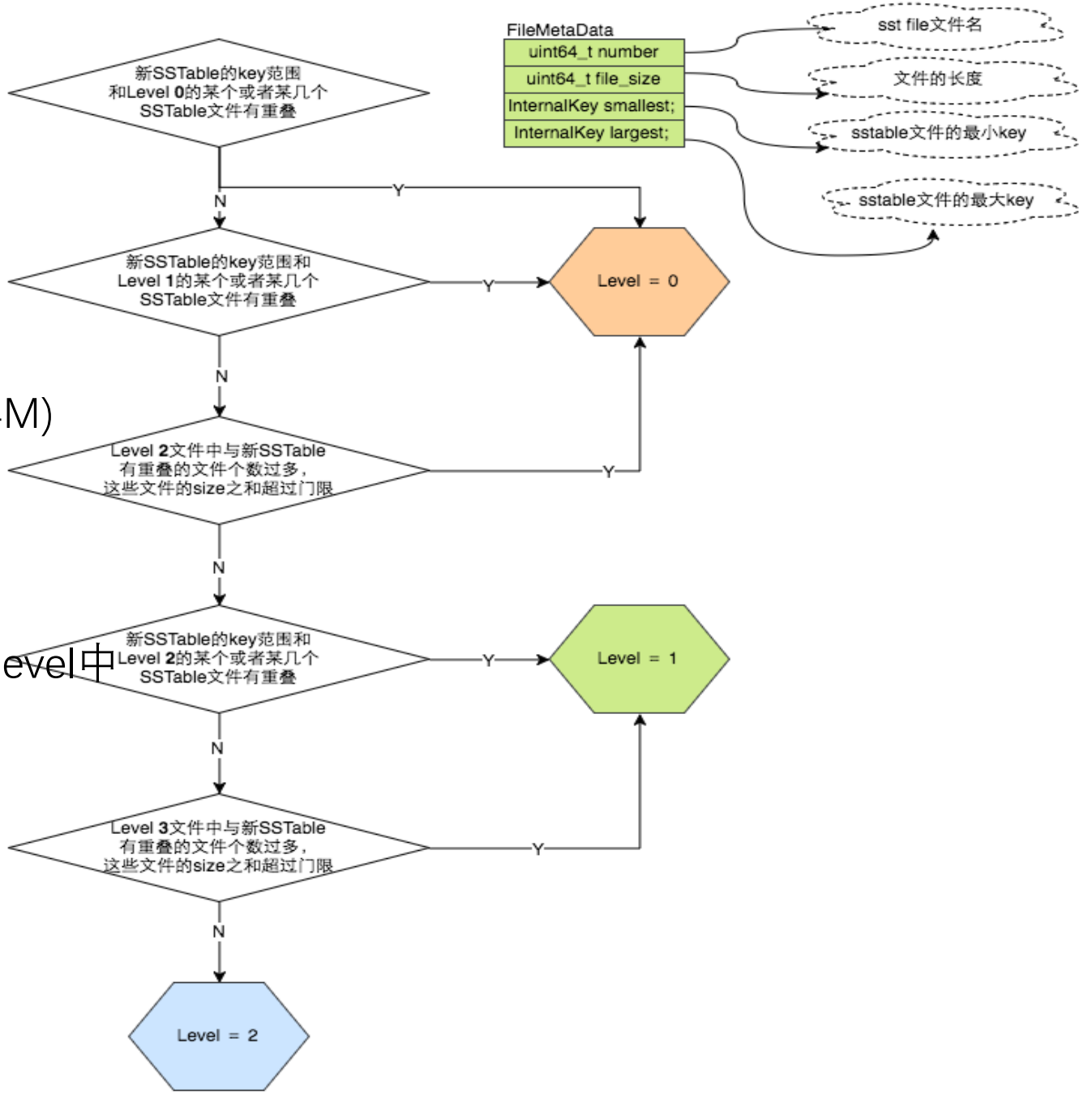
Compaction

- **Minor Compaction**
  - Immutable memtable持久化成sst文件
- **Major Compaction**
  - sst文件之间的compaction



Minor Compaction

- **时机**
  - 写入判断：Memtable 占用内存
    - > options_.write_buffer_size (default 4M)
- **核心过程**
  - 第1步：memsstable格式化成sst文件
  - 第2步：选择放置的level，规则如图
  - 第3步：将新sst文件放到第2步选出的level中
- **动机**
  - 将新sst推到更高Level，控制Level 0的文件个数，影响Read和Compaction
  - 又不能推得过高，控制Read的路径



Major Compaction

- **Manual Compaction**
  - 人工触发的Compaction，有外部接口调用触发，LevelDB内部不会调用
  - void DBImpl::CompactRange(const Slice* begin, const Slice* end)
- **Size Compaction**
  - 根据每个Level的总文件大小来进行，保证各个level文件总量的均衡，保证读的性能
- **Seek Compaction**
  - 每个文件的seek次数都有一个阈值，如果超过了这个阈值，则认为需要Compaction

Priority: **Minor** > Manual > **Size** > Seek

Size Compaction

- 根据Score决定Compection的Level
- Level 0

  score = 文件数 / 4

  – Level 0 不同sst文件key存在重叠，过多会影响读性能
- Level 1~6

  score = 整个level所有的file size总和 / 此level的阈值

~~第 0 层：~~ ~~10M~~（level 0 可以忽略，其采用的是文件个数计算 score）
第 1 层： 10M
第 2 层： 100M
第 3 层： 1000M （ 1G）
第 4 层： 100000M （ 10G）
第 5 层： 1000000M （ 100G）
当然了 Level 6 就不用算了，它已经是最高的层级了，不会存在 Compact 了。

Seek Compaction

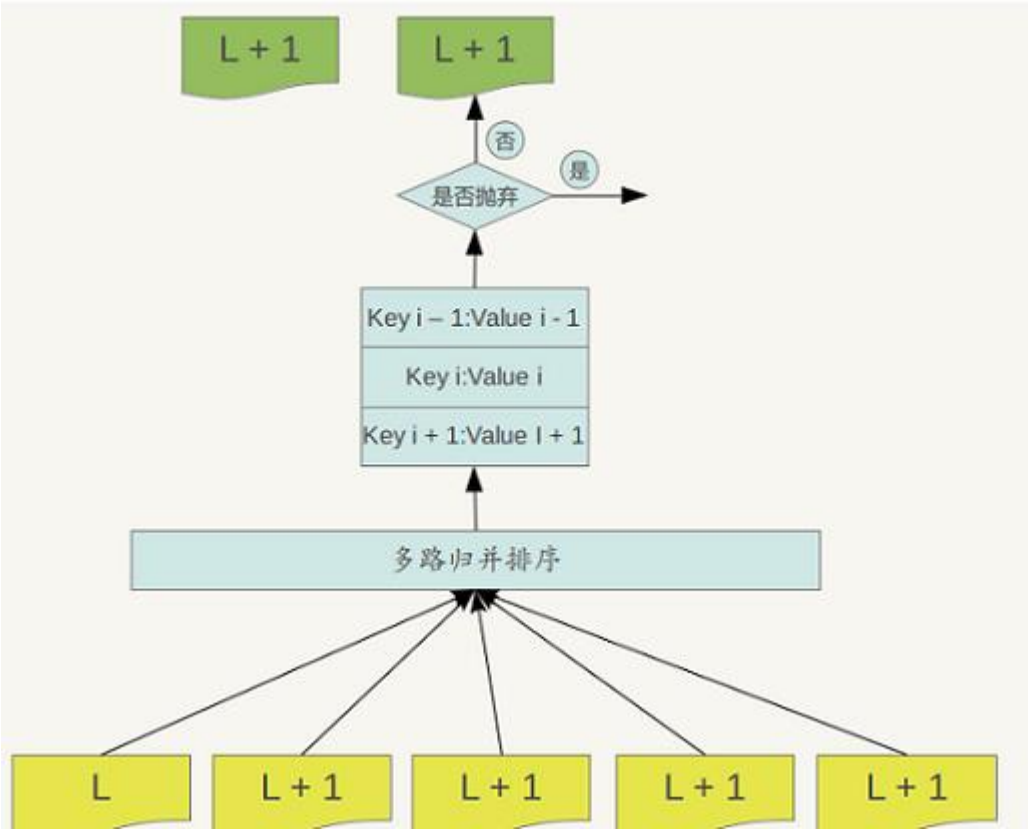
- **Seek Miss**
  - Level n 和Level n+1存在key range重叠，且Level n中不存在这个key
- **Allowed Seek Miss**
  - `allowed_seeks = (sst文件的file size / 16384);  // 16348—16kb`

```
if ( allowed_seeks < 100 )
    allowed_seeks = 100;
```

- **动机**
  - Seek Miss查找同样会消耗IO，这个消耗在达到一定数量可以抵消一次Compaction操作消耗的IO，所以对Seek较多的文件应该主动触发一次Compaction。

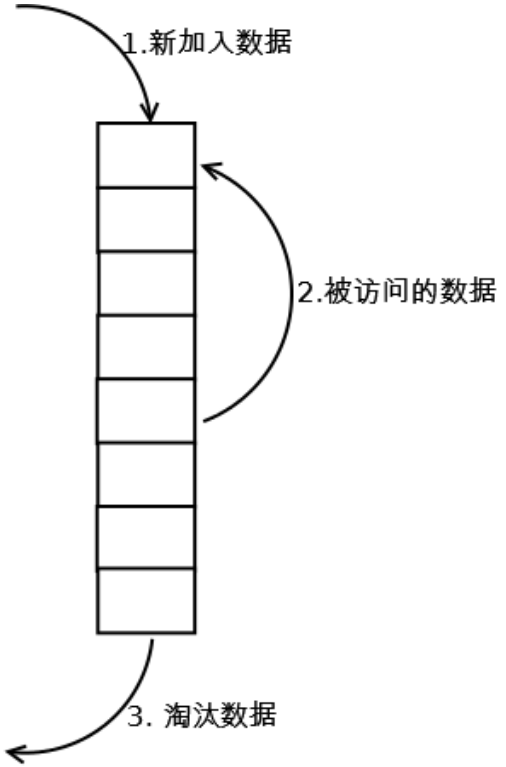
Done Compaction

- **归并排序**
  - 1个Level n 的sst文件和
    多个Level n+1 的sst文件
    归并排序生成新的sst文件
- **sst 文件大小**
  - 默认 kTargetFileSize = 2 * 1048576（2M）



Read-Cache

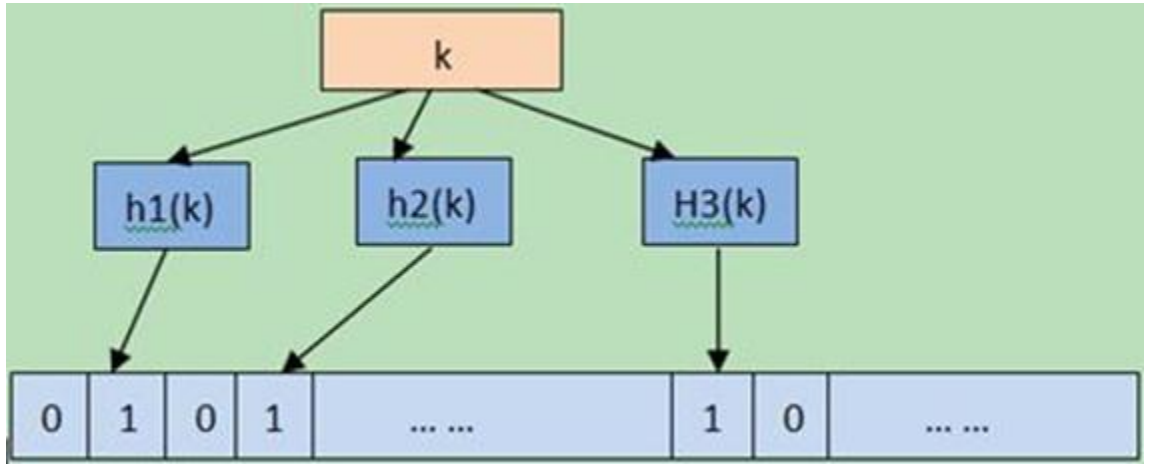
- **Cache**
  - Cache分片：内部默认有16个LRUCache，查找key的时候，先计算属于哪一个LRUCache（**分段锁**、类似Memcached的设计）
- **LRUCache**
  - Hash + LRU（Least recently used)
  - LRU：双向链表
  - Hash ：定位 key 在LRU的位置
  - 查找、插入、删除时间复杂度 O(1)



LRU逻辑原理

Read-Bloom Filter

- **原理**
  - 位图 + 多个Hash function
- **查询特性**
  - 不在：一定不在
  - 在：可能不在（误判，但是概率很小）



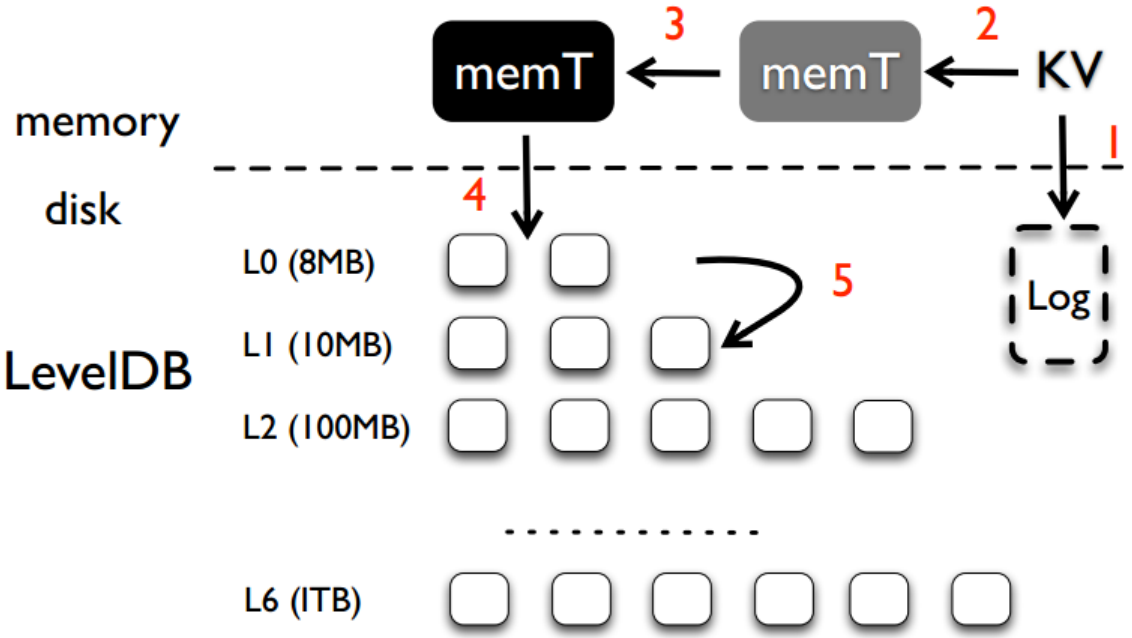
- **实现**
  - 多个Hash function：一个hash函数做一定的处理得到多个hash函数

    已知hash函数H1，则可以计算出hash函数H2=(H1(x)>>17) | (H1(x)<<15)

    Gi(x)=H1(x)+iH2(x)， Gi(x)就是第i次循环 x 得到的hash值

Write-limit

- 写入限制
  - config::kL0_SlowdownWritesTrigger = 8
    当level 0的文件数>=8时，就sleep 1ms
  - kL0_StopWritesTrigger = 12;
    进入条件变量wait，等待下次SignalAll
  - 每次执行Done Compaction在Signal
- 动机
  - 避免写入过快，造成Level 0 文件过多，影响读
  - 避免写入过快，造成大量写带宽（Write和Compaction），影响读
- 问题
  - 会卡前端的I/O



Problems

I/O amplification

- **Write amplification**
  - Compaction
- **Read amplification**
  - Seek Miss（如图2.中 第2、3步）
- **卡前端I/O**
  - I/O放大造成后端Compaction来不及
- **HDD：**顺序读写 100x 随机读写
- **SSD：**顺序读写 <10x 随机读写
  - **是否还划算 ???**
- **大value**
  - 写放大异常明显

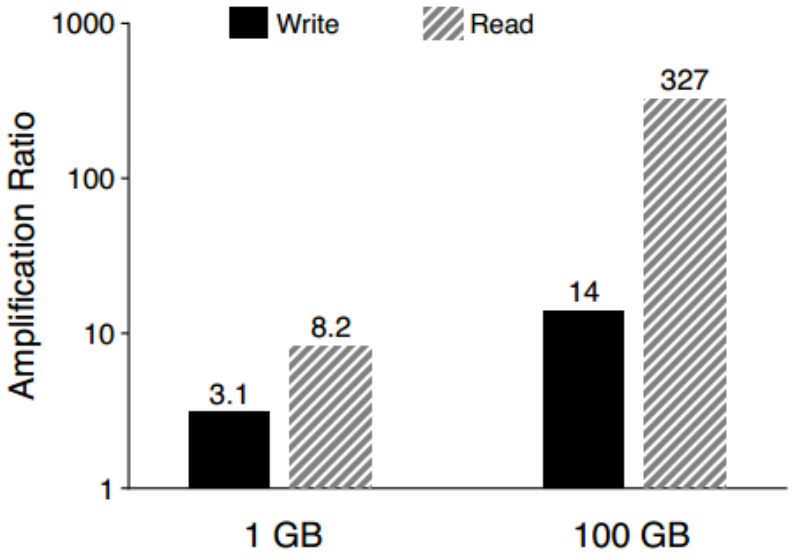


图1. Write and Read Amplification

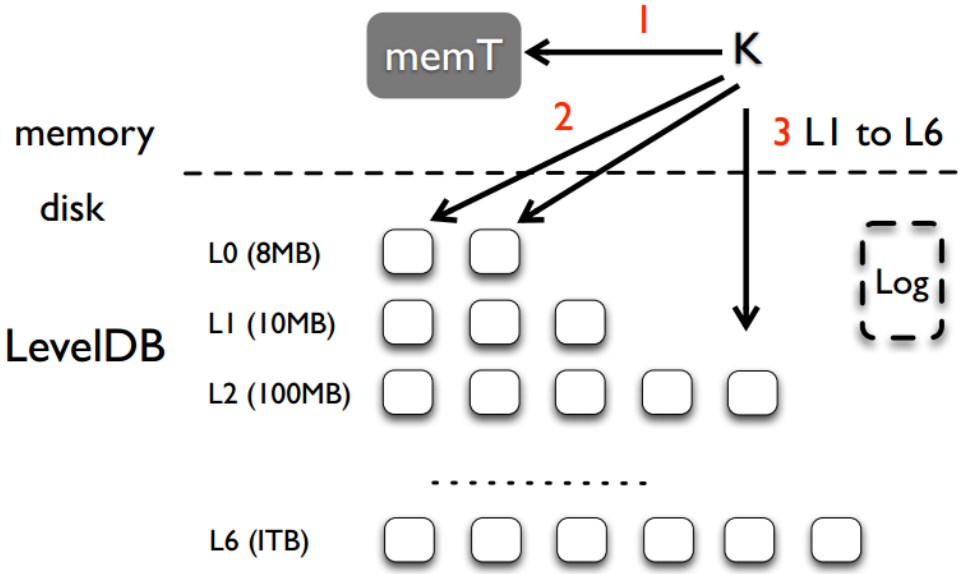
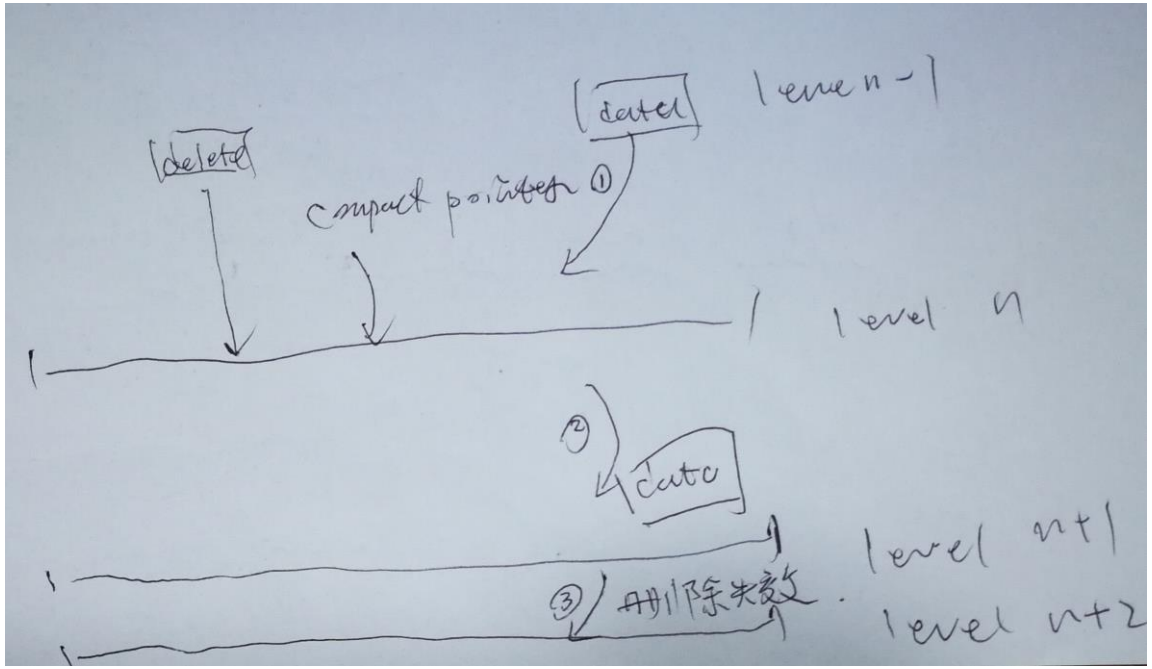


图2. Read

Delete lose efficacy

- **数据膨胀**
  - 删除标记和实际数据在Compaction中如果不能相遇，那么将导致删除的数据不能被删除
  - **示例**：后来的key越来越大；写入一段数据在删除一段数据；数据写入速率快



Improvement and Optimization

Separating Keys from Values in SSD-conscious Storage

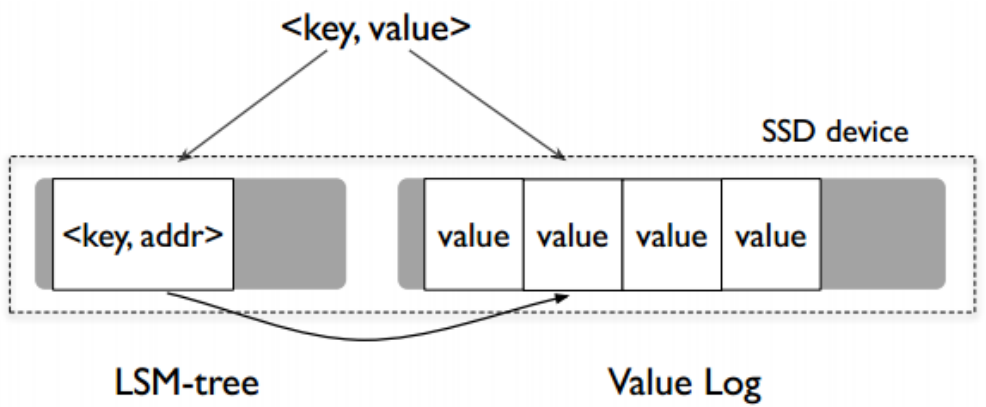
I/O amplification

- **大value**
  - 数据长度越大，越容易触发Compaction，从而造成写放大；
  - 如果把上层文件看做下层文件的cache，大数据长度会造成这个cache能cache的数据个数变少，从而读请求更大概率的需要访问下层数据，从而造成读放大；

- **SSD：**顺序读写 <10x 随机读写
  - **是否还划算 ???**

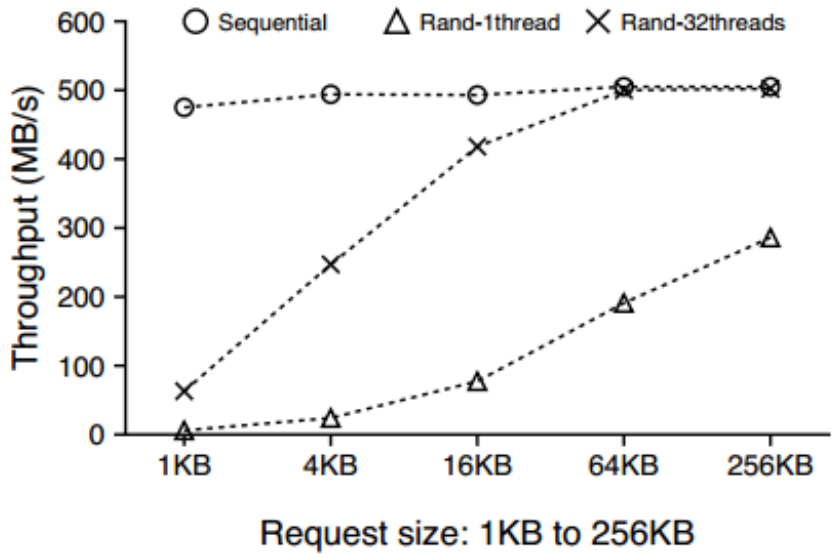
Separating Keys from Values

- **Write**
  - Append log,
  - 将key写入LSM-tree,
  - Append value
- **Read**
  - LSM-tree中获取地址，然后读取Value
- **Delete**
  - 标记删除key和value，无效Value交给之后的垃圾回收
- **好处**
  - 避免了归并时无效的value而移动，从而极大的降低了读写放大
  - 显著减少了LSM的大小，以获得更好的cache效果



Challenges

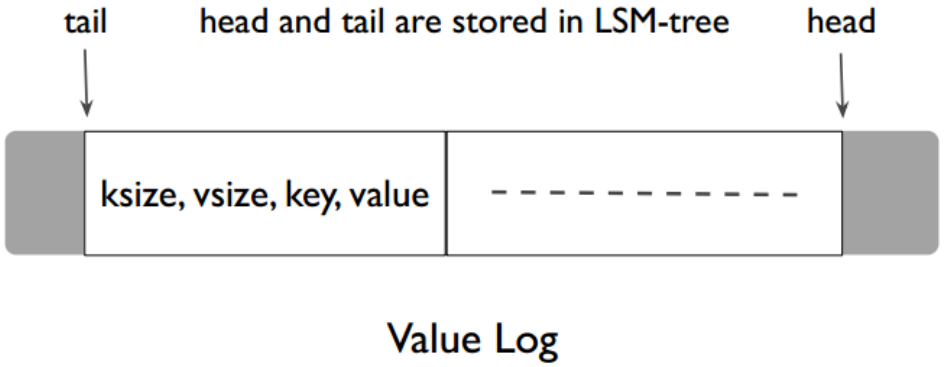
- **Range Read**
  - Key Value的分离，Range操作从顺序读变成了顺序度加多次随机读，从而变得低效。
  - 利用SSD并行IO的能力，可以将这种损失尽量抵消



Sequential and Random Reads on SSD

Challenges

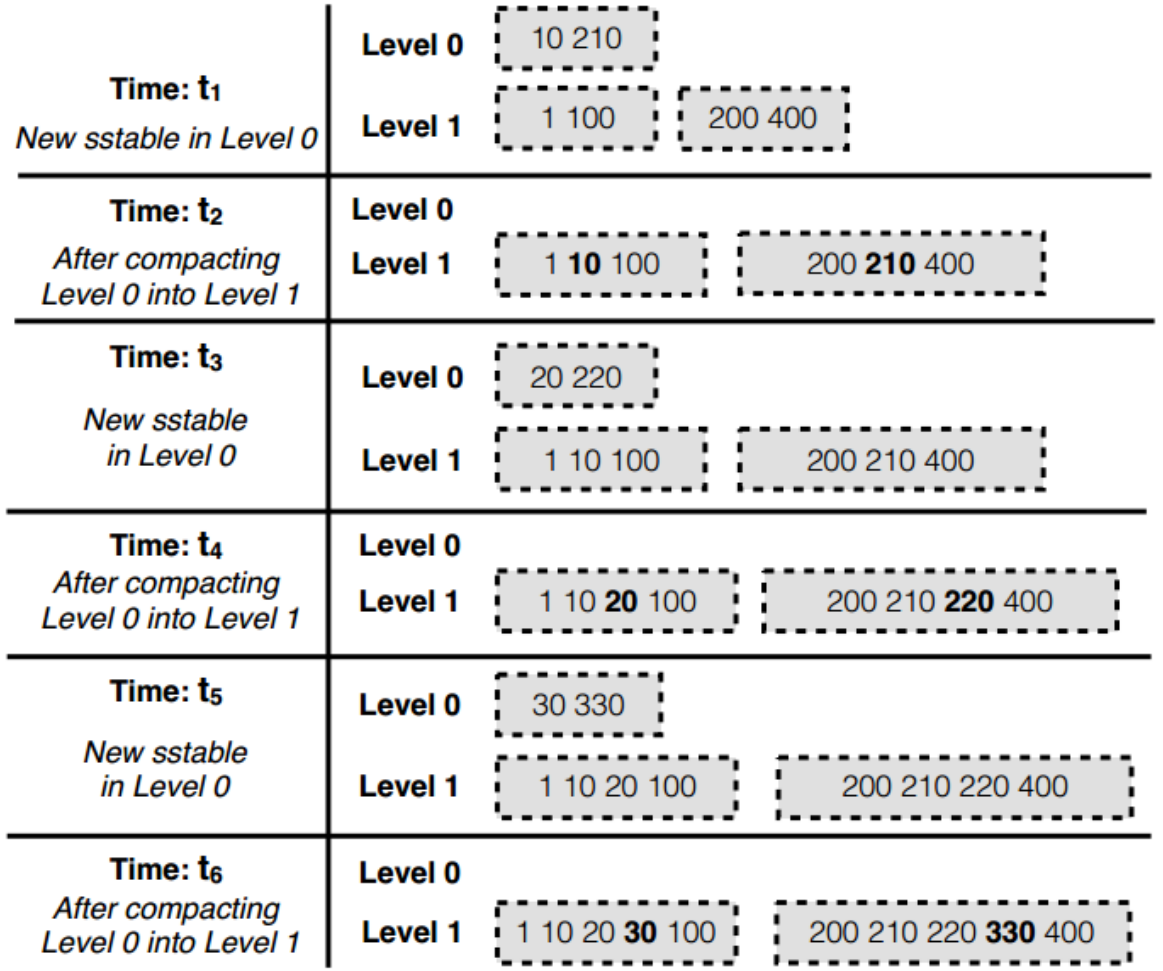
- **Garbage Collection**
  - Compaction过程需要被删除的数据只是删除了Key，Value还保留在分开的Log中
  - 其中head的位置是新的Block插入的位置，tail是Value回收操作的开始位置，垃圾回收过程被触发后，顺序从Tail开始读取Block，将有效的Block插入到Head。删除空间并后移Tail。
  - 有效的数据需要重新Append，也也是写放大，需要空间放大和写放大的权衡
- **More**
  - 多value log文件 + 统计每个value log文件的有效数据比，决定垃圾回收的时机



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

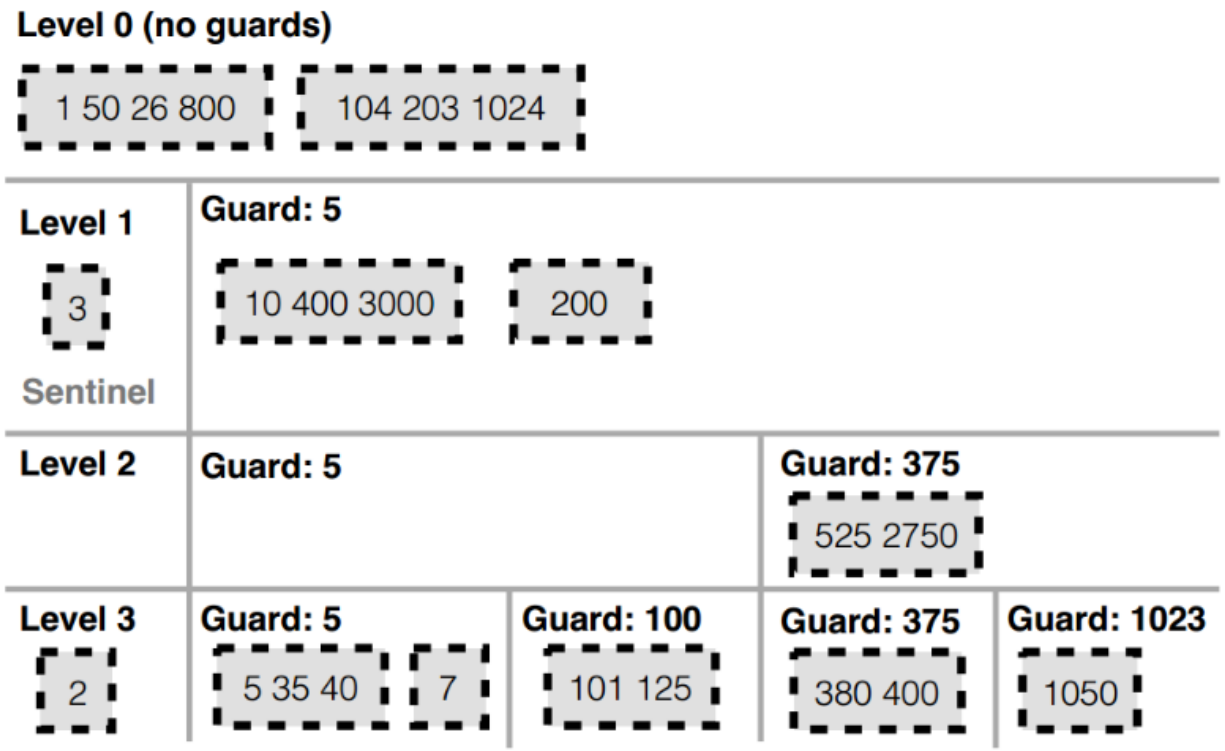
LSM Compaction

- **存在问题**
  - 同一level反复的rewrite
    例如1、10等
- **存在问题**
  - 反复的rewrite,例如1、10等



Guards

- **Motivation**
  - 消除同一level反复的rewrite
- **Idea**
  - Skip List增加Guards，保证搜索的平衡性
- **Add guards**
  - 增加Guards，保证搜索的平衡性



Summary

Summary

- 通用的 key-value 存储框架
- 根据自己的场景调优

References

- Standing on the shoulders of giants

[1]. http://www.cnblogs.com/haippy/archive/2011/12/04/2276064.html

[2]. Tags of leveldb. http://bean-li.github.io/tags/

[3]. 那岩. Leveldb实现解析.pdf

[4]. Leveldb docs. https://github.com/google/leveldb/tree/master/doc

[5]. https://dirtysalt.github.io/html/leveldb.html#orgheadline186

[6]. LSM upon SSD. http://catkang.github.io/2017/04/30/lsm-upon-ssd.html

[6]. O'Neil P, Cheng E, Gawlick D, et al. The log-structured merge-tree (LSM-tree)[J]. Acta Informatica, 1996, 33(4): 351-385.

[7]. Raju P, Kadekodi R, Chidambaram V, et al. PebblesDB: Building Key-Value Stores using Fragmented Log-Structured Merge Trees[C]//Proceedings of the 26th Symposium on Operating Systems Principles. ACM, 2017: 497-514.

[8]. Lu L, Pillai T S, Gopalakrishnan H, et al. WiscKey: Separating keys from values in SSD-conscious storage[J]. ACM Transactions on Storage (TOS), 2017, 13(1): 5.

End

Thank you
Q&A