参考资料

1.The Log-Structured Merge-Tree

2.一文带你看透基于LSM-tree的NoSQL系统优化方向

3.数据库存储与索引技术:分布式数据库基石——LSM树

4.WiscKey: Separating Keys from Values in SSD-conscious

5.dgraph-io/badger

6.badger源码分析

7.B树与LSM树读写和空间放大分析

8.大白话彻底搞懂 HBase Rowkey 设计和实现方式

9.Kvrocks: 一款开源的企业级磁盘KV存储服务

10.Kvrocks data structures design

11.布隆过滤器的原理及完整公式推导

1. LSM-Tree简介

1.1 LSM-Tree的概念

LSM-Tree 全称是Log Structured Merge Tree，是**一种分层、有序、面向磁盘的数据结构**，其核心思想是充分利用磁盘的顺序写性能要远高于随机写性能这一特性，将批量的随机写转化为一次性的顺序写。

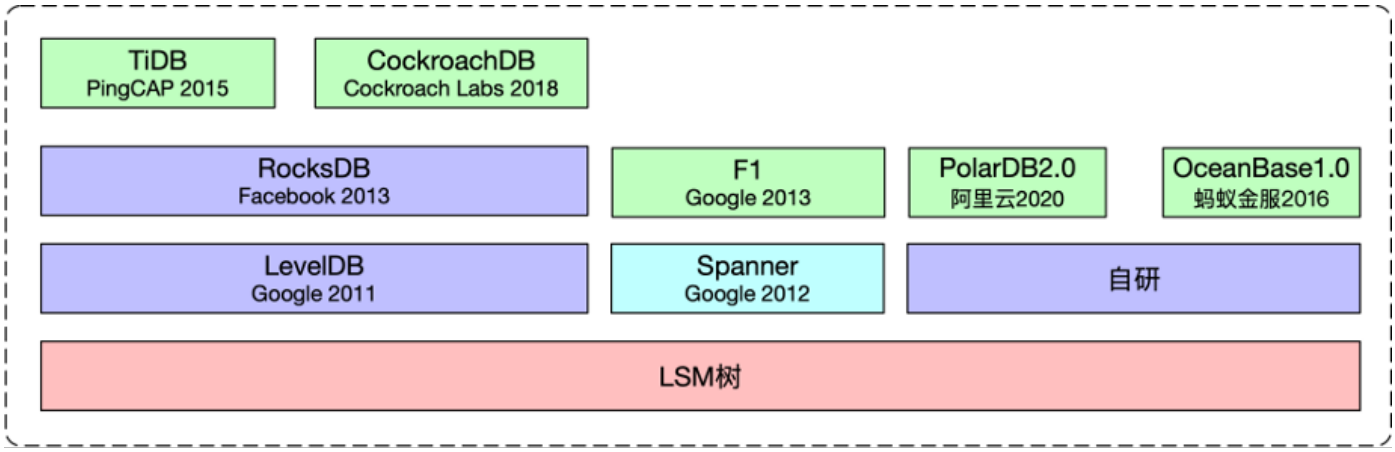
1.2 应用场景

以NoSQL为代表的分布式数据库多采用LSM树用于构建底层的存储系统。

1. Apache Cassandra：Cassandra 是一个高可用性、高可扩展性的分布式 NoSQL 数据库，它使用了 LSM 树来存储数据。

2. LevelDB：由 Google 开发的高性能键值对数据库，使用 LSM 树来存储数据。
3. RocksDB：由 Facebook 开发的高性能嵌入式键值对数据库，它是 LevelDB 的改进版本，同样使用 LSM 树来存储数据。
4. HBase：HBase 是一个分布式列存储数据库，它是基于 Hadoop 的一个开源项目。HBase 的数据存储是基于 LSM 树实现的。
5. ScyllaDB：ScyllaDB 是一个高性能的分布式 NoSQL 数据库，使用 LSM 树来存储数据，并且兼容 Cassandra API。
6. Apache Lucene：一个开源的信息检索库，广泛用于各种搜索引擎和全文检索系统。虽然Lucene主要关注于索引结构，但其底层的数据存储也使用了LSM Tree的设计思想。

另外TiDB的本地存储使用的是RocketsDB TiDB本地存储

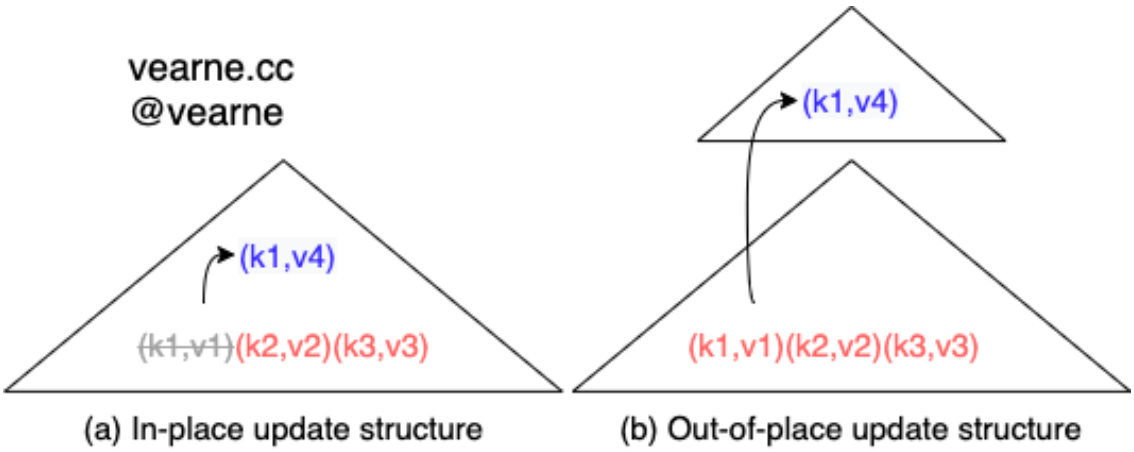


1.2 LSM-Tree相较于B+树的优势

1) insert

2) update & delete

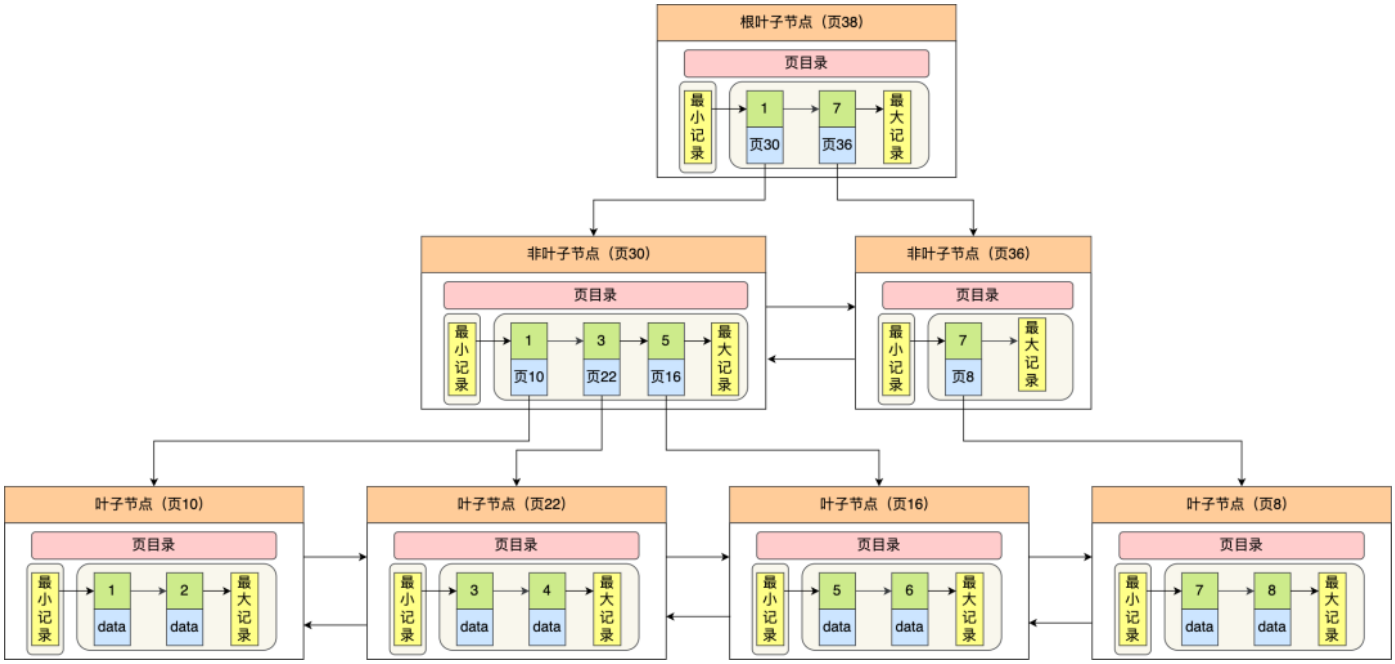
in-place update VS out-of-place update



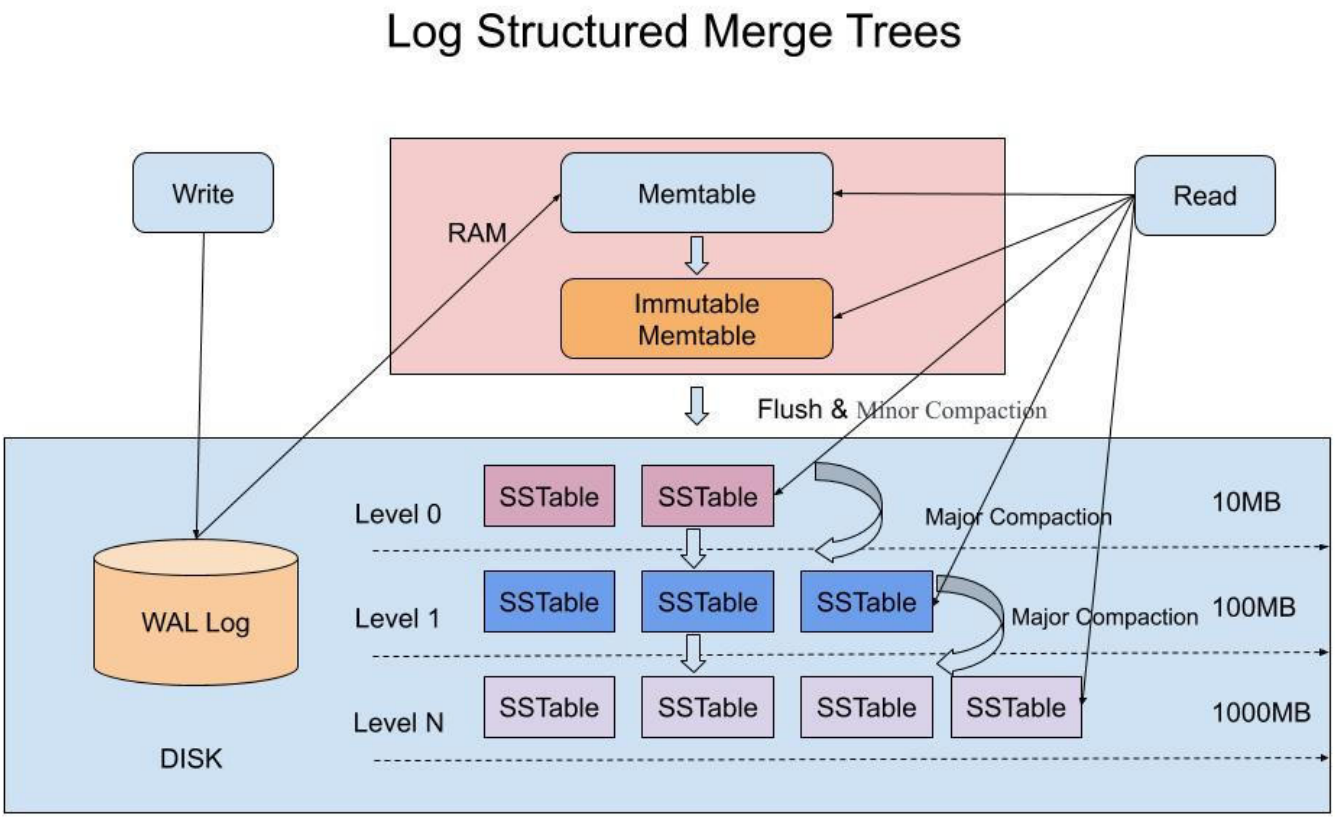
对于out-of-place update，insert、update 和 delete 都是Append，能充分利用硬盘的顺序I/O的能力。

> In HDDs, random I/Os are over 100× slower than sequential ones.

B+树结构



2. LSM-Tree的结构



2.1 结构说明

- `Memtable` 和 `Immutable Memtable` 在内存中
- `WAL Log` 和 `SSTable` (Sorted String Table)位于硬盘中(机械硬盘或SSD)
- `Immutable Memtable` 由后台线程异步flush到硬盘(Minor Compaction)
- 第i层的 `SSTable` 会定期Merge成新的SSTable存储到第i+j层，一般情况下j=1。(Major Compaction)
- 第i+1层和第i层的容量比值是固定值(fixed multiplier) **Why?**

> 原型中抽象出来一个模型，其中提出了在一个稳定的workload下，保持LSM-tree的层数不变，当每一层的大小比例 Ti `Ti = |Ci+1|/|Ci|` 在各个层之间维持一个稳定的值时， 写性能能够优化

2.2 优点和缺点

2.2.1 优点

- 把随机写转化为顺序写，支持高吞吐的写（尤其适合分布式大数据应用场景）
- 采用append方式写数据，读写操作互相独立，可支持高并发应用
- 适合写多读少的应用

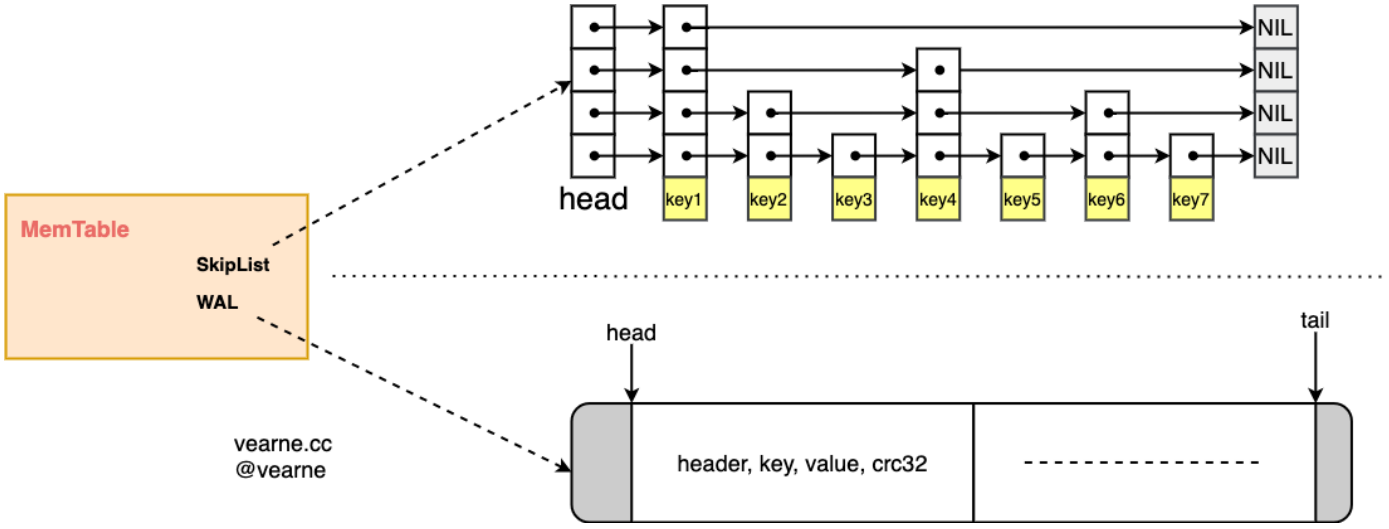
2.2.1 缺点

- 读性能较差
- 空间放大(Space Amplification)，需要compaction才能回收空间
- 写放大(Write Amplification)
- compaction操作导致系统性能抖动：

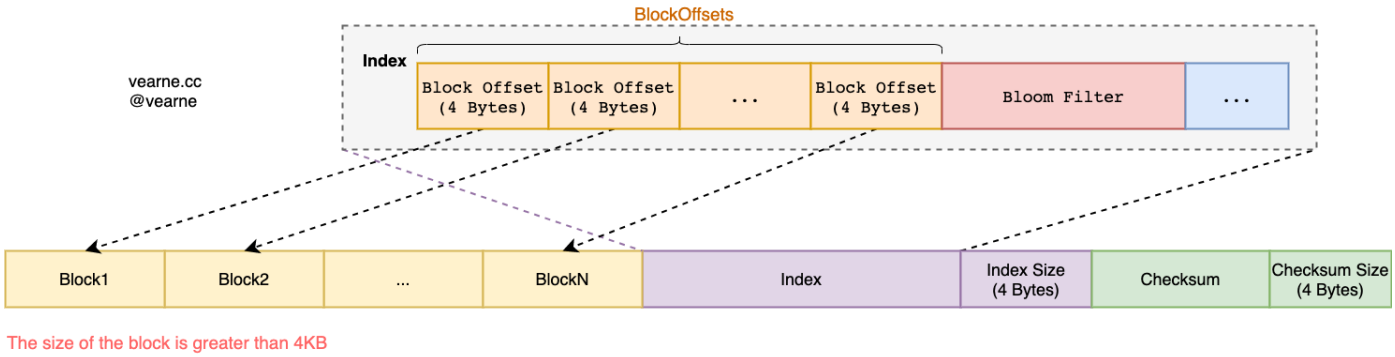
3.写入操作

1. 键值对写入 `Memtable`
2. `Memtable` 达到一定大小（默认为64MB）后，转换为 `Immutable Memtable`
3. 后台线程异步将 `Immutable Memtable` 转换为 `SSTable` 并flush到硬盘上(Minor Compaction)
4. 第i层的 `SSTable` 会定期Merge成新的 `SSTable` 存储到第i+j层，一般情况下j=1。(Major Compaction)

3.1 Memtable的结构

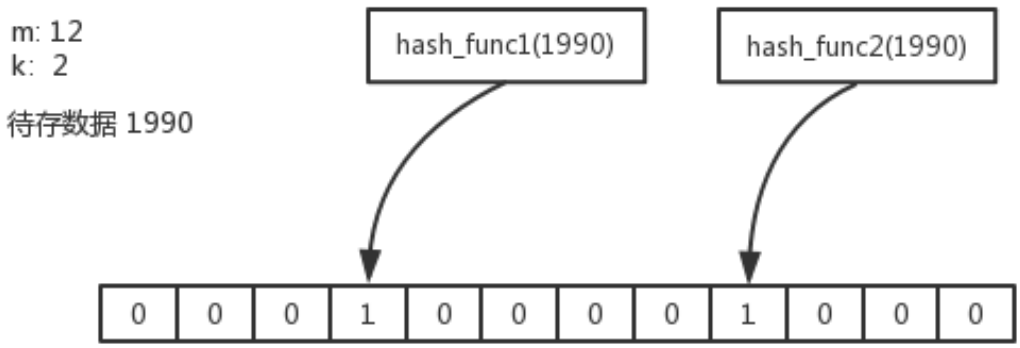


3.2 SSTable的结构



The size of the block is greater than 4KB

3.3 Bloom Filter

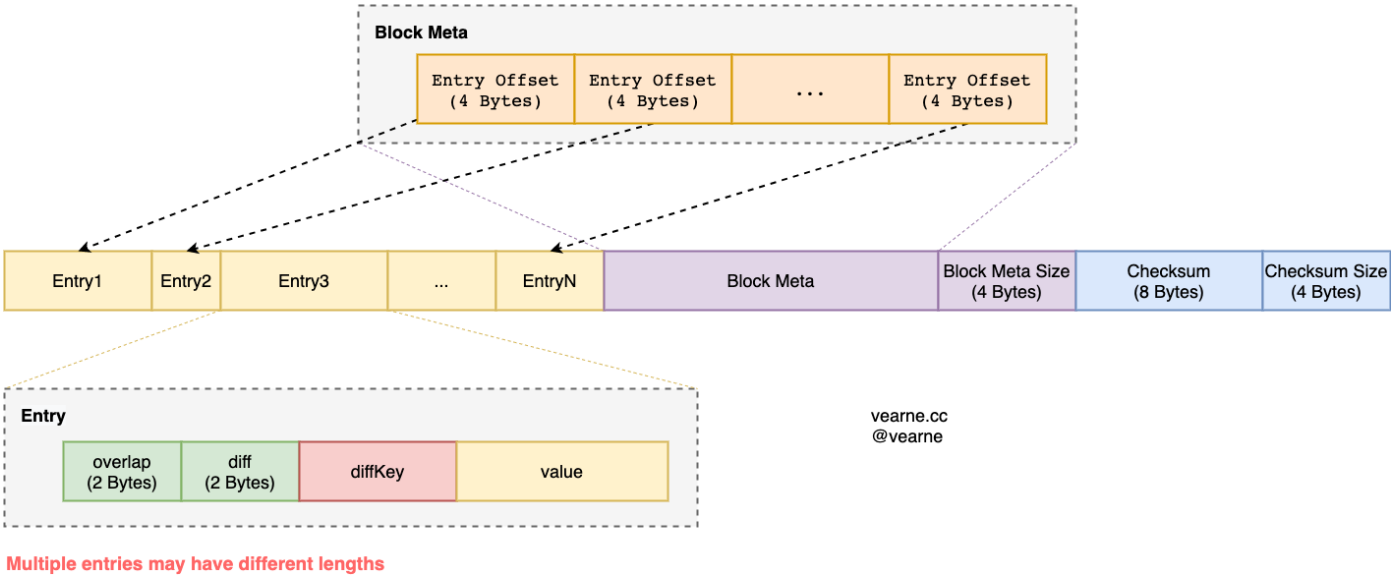


假定 `SSTtable` 的大小为64MB，Entry平均大小100B

n(预期存储的数据量)	p(假阳性概率)	k(哈希函数的数量)	m
640,000	0.01	7	约为500KB

- 相比HashMap节约大概80%空间开销
- BloomFilter种存储的数据量超过预设值，假阳性的概率会急剧上升

3.4 Block的结构



3.5 Key的前缀压缩技术

假设数据库用来存储用户的所有文章: RowKey被设计为 {uid.hashCode() % 10}:{date}:{uid}:{articleID}，value为文章的属性和详细内容

- `uid` 是用户ID， 用户数量不超过10亿
- `date` 为日历上的日期，格式为yyyyMMdd
- `articleID` 是文章ID

Key的设计说明

- {uid.hashCode() % 10} 是为了RowKey 需要能够均匀的分布到各个RegionServer
- `date` 固定长度8个字节
- `uid` 固定长度10位，长度不够的在左侧补0

某个Block中的数据形如

```
08:20230504:1285252117:4719447279666471
08:20230504:2801321721:4897678780010300
08:20230504:5873331601:4897678780010320
08:20230504:5803341781:4897678780010322
```

overlap	diffKey	备注
0	"08:20230504:1285252117:4719447279666471"	**baseKey**
12	"2801321721:4897678780010300"	
12	"5873331601:4897678780010320"	
12	"5803341781:4897678780010322"	

上面的例子使用前缀压缩技术大约节约了20%的空间

Q: 如果RowKey 被设计成 {uid.hashCode() % 10}:{uid}:{date}:{articleID} 有何不同?

4.合并操作(Compaction)

4.1 原因

- 解决空间放大
- 解决读放大

4.2 处理过程

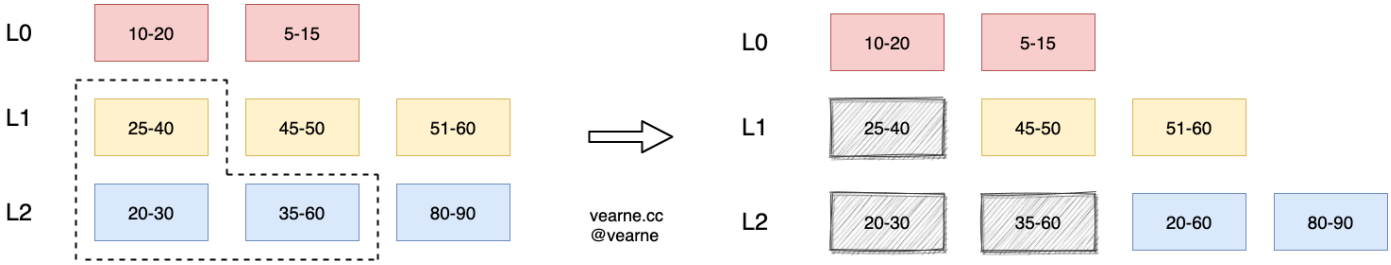
在badgerDB中, compaction 分为 `minor compaction` 和 `major compaction` 合并.

当 memtable 写满后 flush 到 L0 层的磁盘上, 被称为 minor compaction.
从 L0 层开始往下层合并数据, 被称为 `major compaction`, LSM Tree 常说的合并其实就是这个 `major compaction` 。

Compaction 合并本质是一个**归并排序**的过程, 把两层的 `SSTable` 进行合并, 并构建新的 `SSTable` 写到 **Level** n+1 层,

- 1.选择需要合并的**Level**，并确认key range范围
  - 对各个 **Level** 进行 score, 分值大的**Level**，优先选择
  - 对 **Level** 0 层，socre = **Level** 0 层 `SSTable` 的数量 / NumLevelZeroTables
  - 对 **Level** > 0 层, score = 当前有效使用空间 / 动态计算出的预期空间阈值.
- 2.通过迭代器把相关的 sstable 的数据读取出来, 进行归并排序.
  - Key相同的条目只会保留最新NumVersionsToKeep(默认值为1)条数据
  - delete或者过期的条目会被清除

    备注:需要设置discardTs，一般的LSM Tree实现只在最后一次清除
- 3.把合并后的数据写到 level N+1 层的 `SSTable` 里（**Level** 0层可以跳过empty **Level**）
- 4.更新manifest配置文件
- 5.删除**Level** n和**Level** n+1 层中旧的 `SSTable`
- 6. `Compaction` 操作是可以并发执行的



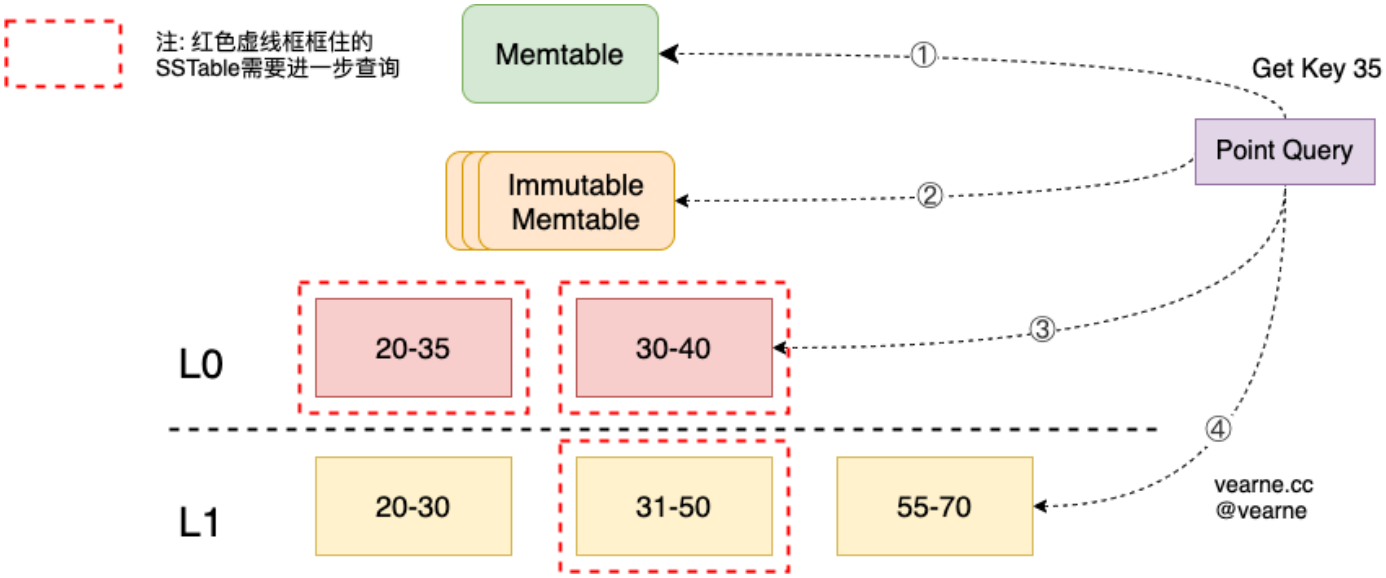
小技巧

badger中实际存储的key = realKey + uin32(math.MaxUint64 - timestamp)

这样做的目的是相同的key的情况，最新版本的数据能够优先被检索到

5.读取操作

5.1 Point Query



查询过程

- Step 0 如果是update事务，尝试从pendingWrites(写缓冲区)中读取
- Step 1 尝试从memtable读取
- Step 2 尝试从immutable memtable集合中读取
- Step3 尝试从 `Level` 0层的 `SSTable` 中获取
- Step4 尝试从 `Level` 1层的 `SSTable` 中获取, 逐层获取，直到最后一层

确定需要查询的 `SSTable`

- 对于 `Level` 0层，由于多个 `SSTable` 的key range可能有重叠，所以每个 `SSTable` 都需要查询一遍
- 对于 `Level` i层(i > 0), 由于 `SSTable` 的key range没有重叠，且它们是有序的，所以直接使用二分查找就可以定位到需要查询的 `SSTable`

`SSTable` 内部的查询逻辑

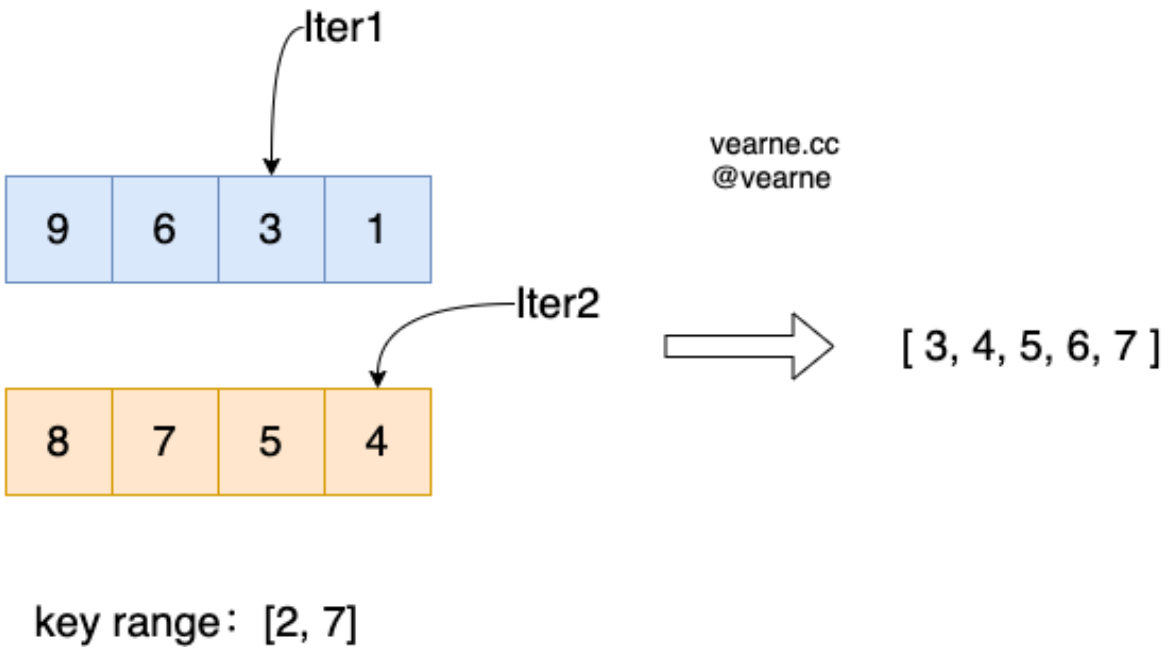
- 1）使用bloomfilter粗略的判断(会有一定的假阳性概率) `SSTable` 是否包含对应的key
- 2）使用BlockOffsets进行二分查找定位到对应的 `Block`
- 3）使用EntryOffsets进行二分查找定位到对应的 `Entry`

5.2 Range Query

使用 `Iterator` 来遍历range范围内的所有数据，该 `Iterator` 包含一组子 `Iterator` 。

查询的过程类似于归并排序中，合并多个有序数组，需要同时查询 `MemTable` , `Immutable MemTable` ，以及各个Level层的 `SSTable`

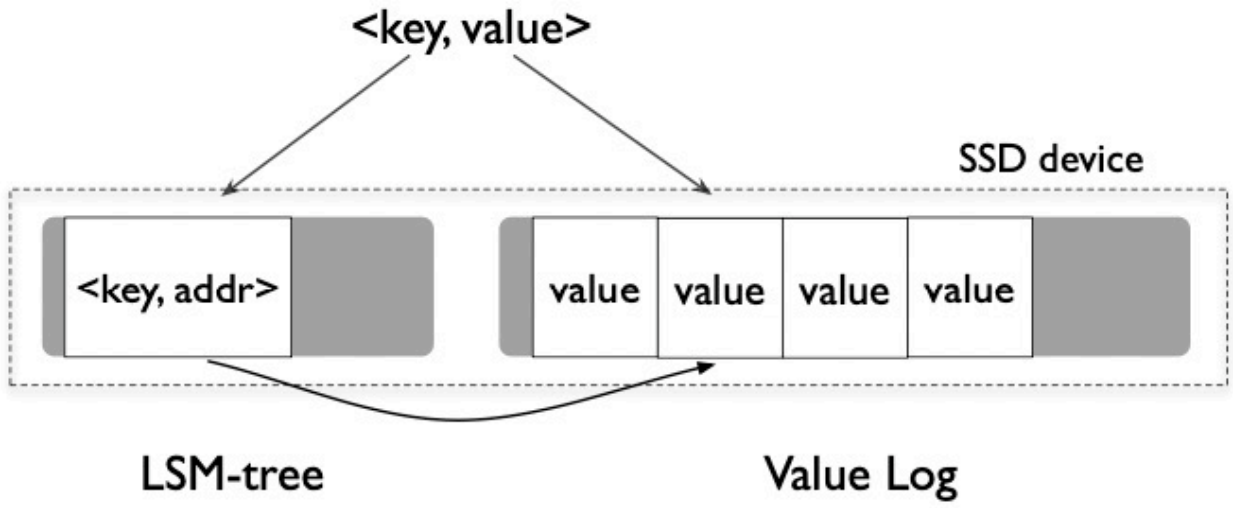
（注意: `Level` 0层每个 `SSTable` 都需要查询）



- Step 1 通过seek()定位到子迭代器的移动到指定位置(比startKey大的下一个最小的key)
- Step 2 比较多个子迭代器的当前元素，返回最小的元素，修改指针的位置
- Step 3 重复Step 2，直到返回的元素大于endkey

```
itr := txn.NewIterator(badger.DefaultIteratorOptions)
for itr.Seek("startKey"); itr.Valid(); itr.Next() {
    item := itr.Item()
    key := item.Key()
    if bytes.Compare(key, "endKey") > 0 {
      break
    }
    // rest of the logic.
}
```

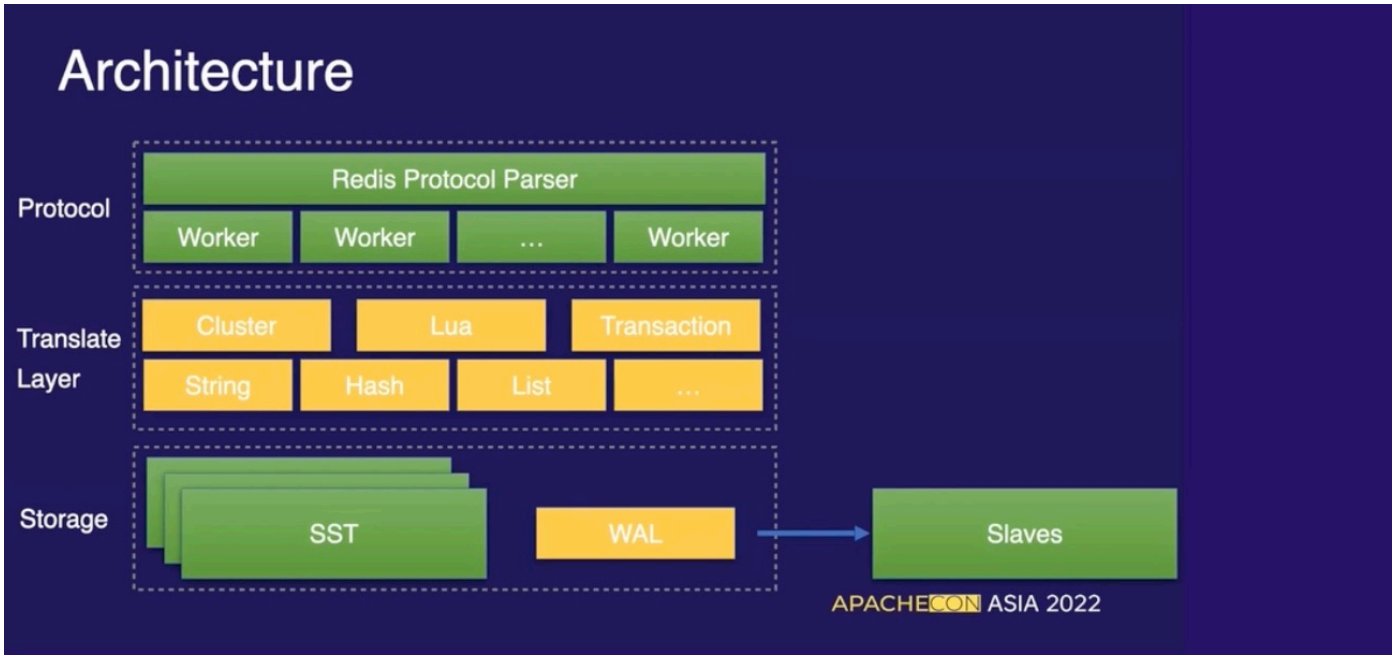
6.LSM-Tree的优化



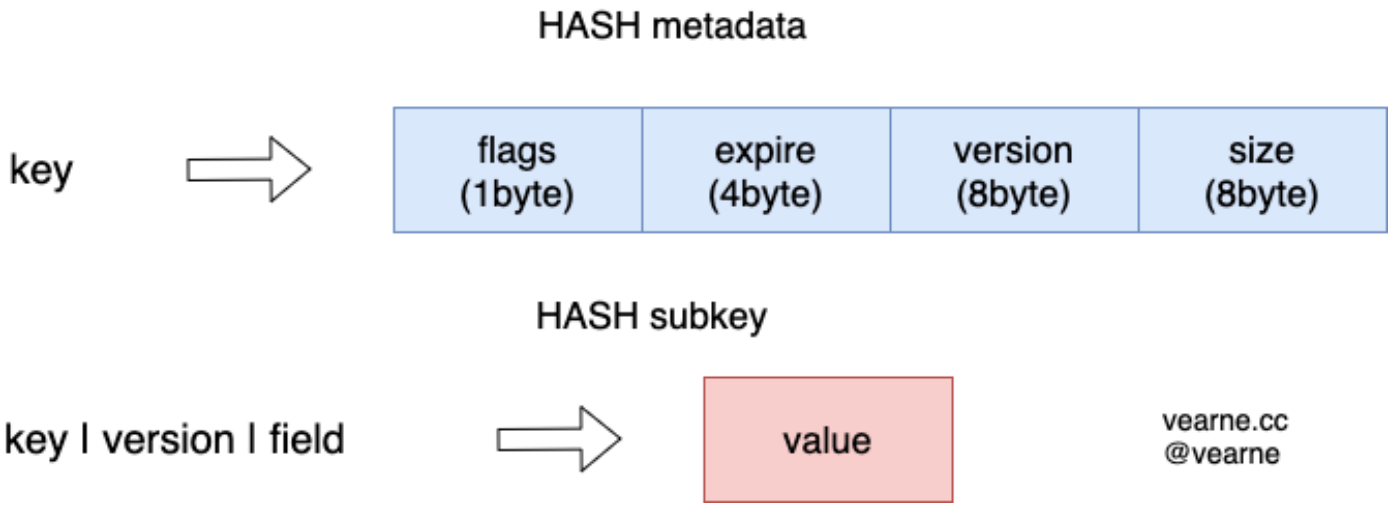
由于Compaction的存在，同一个key和value会被写入多次，因此 WiscKey: Separating Keys from Values in SSD-conscious 提出对于value较大(默认1MB)的情况，直接将value写到独立文件Value Log，在LSM-tree只存储地址，可以有效的缓解写放大问题。

7. 基于KV数据库，构建更为复杂的NoSQL数据库

Kvrocks 是基于 RocksDB 之上兼容 Redis 协议的 NoSQL 存储服务，最早是美图的内部项目，主要目标是提高 Redis 的容量以及降低内存使用成本。



以Hash结构举例



metadata和subkey位于2个不同的column families

- `flags` 目前是来标识当前 Value 的类型，比如是Hash/Set/List 等
- `expire` 是 Key 的过期时间
- `version` 是 Key 创建时自动生成的单调递增的 id，用于实现快速删除
- `size` 表示subkey的数量

以 HSET 命令为例，伪代码如下:

```
HSET key, field, value:
```

```
// 先根据 hash key 找到对应的 metadata 并判断是否过期
// 如果不存在或者过期则创建一个新的 metadata
metadata = rocksdb.Get(key)
if metadata == nil || metadata.Expired() {
   metadata = createNewMetadata();
}
// 根据 metadata 里面的版本组成 subkey
subkey = key + metadata.version+field
  if rocksdb.Get(subkey) == nil {
   metadata.size += 1
}
// 写入 subkey 以及更新 metadata
  rocksdb.Set(subkey, value)
rocksdb.Set(key, metadata)
```