NoSQL Databases: II

The Log-Structured Merge Tree (LSM-Tree)

金培权（jpq@ustc.edu.cn）

主要内容

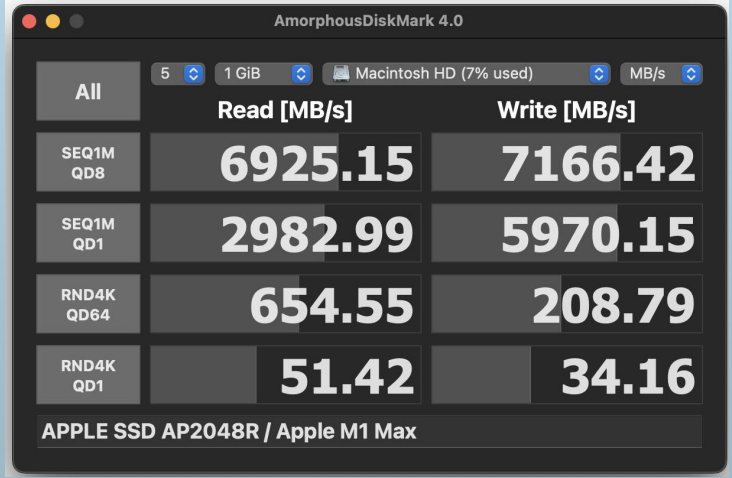
- B+-tree的问题
- LSM-tree的设计思想
- LSM-tree的实现
- LSM-tree的优缺点

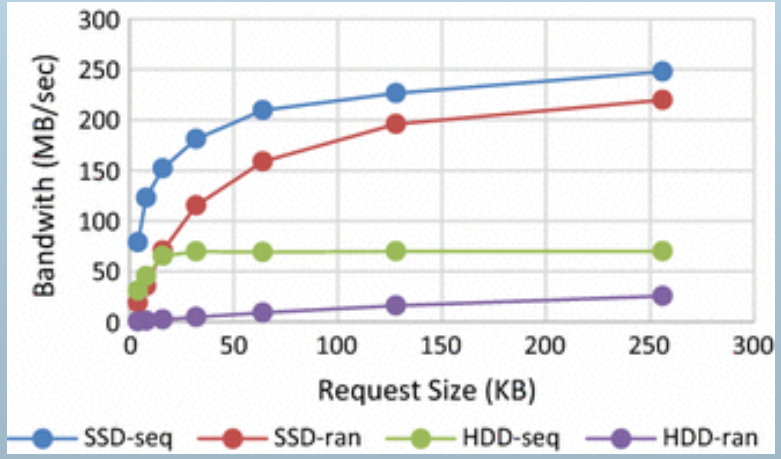
一、B+-tree的问题

- 原位更新（In-Place Update）
- 写代价高，写性能差
  - 对叶节点的写基本都是随机写
  - 级联分裂、合并等SMO操作带来大量的随机写
    - SMO：Structure Modifying Operation

B+-tree的特点：

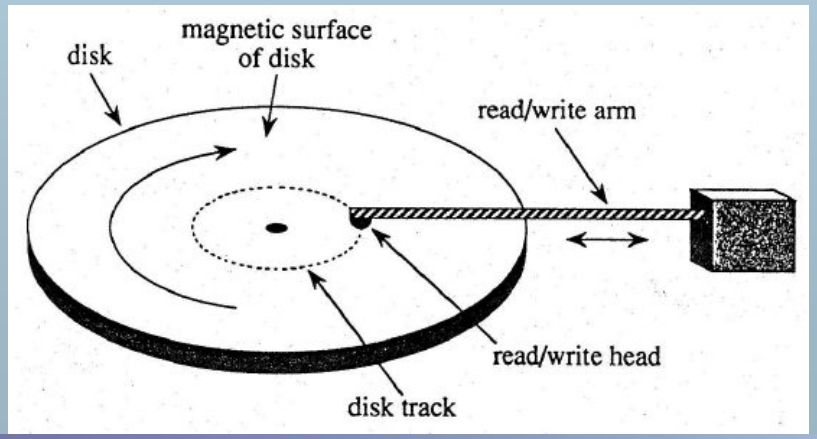
- 读性能好
- 写性能差

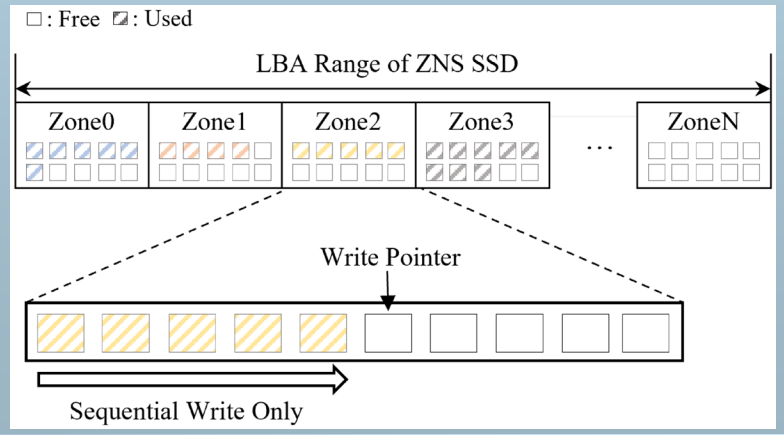




一、B+-tree的问题

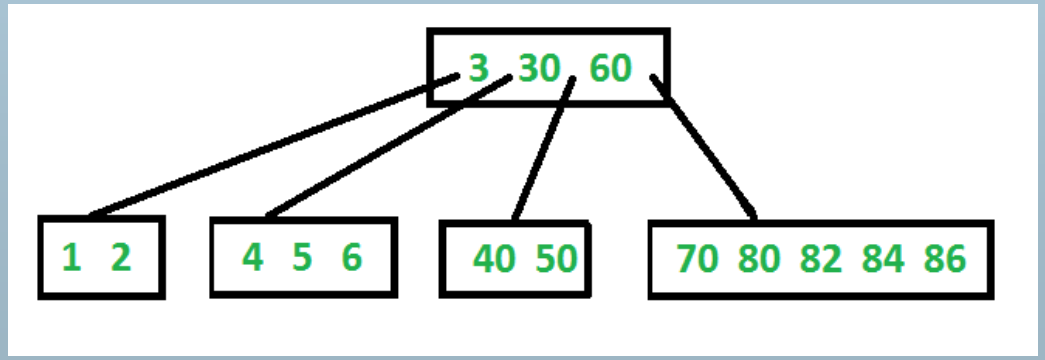
- 如何避免随机写？
  - 采用log-structured的Append-only写
  - 回顾：Undo日志、Redo日志
    - 日志项不允许修改，只能Append
  - Append方式写日志一般可视作顺序写，写性能高
    - 可以在支持随机读写的设备通过软件实现顺序写
    - 也有设备只支持顺序写，如ZNS SSD，需要软件适配





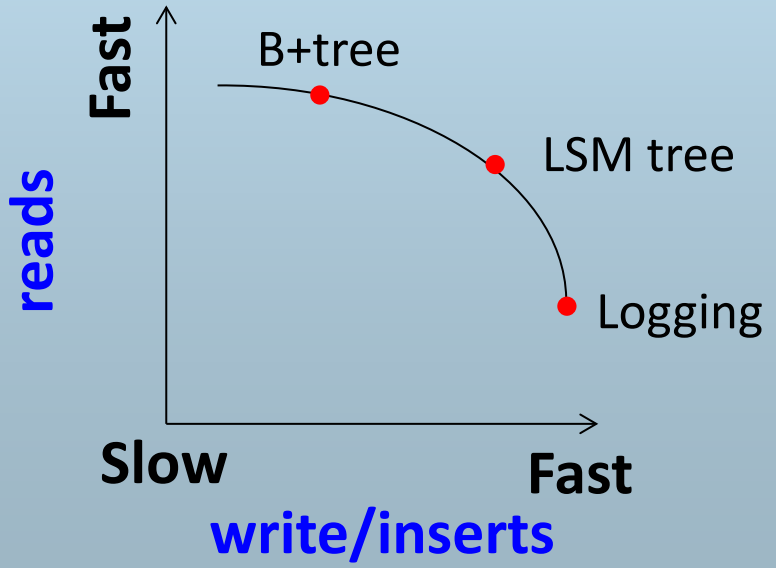
一、B+-tree的问题

- 但是直接把B+-tree的节点改成Log Structured的 Append-Only不能解决随机写的问题
  - 叶节点需要有序
    - 把新的键值Append到叶节点最后不行，会降低读性能
  - 不同节点的更新依然是随机写
    - 另一种思路：把所有叶节点的更新合并起来一起顺序写，并且可以推广到其它层，比如某一层的节点每次更新都批量一起写入
    - 大数据场景下，每一层如果是一个文件，每次写入都是顺序写



二、LSM-tree的设计思路

- B+-tree: 写慢读快
- Logging：写快读慢
- LSM-tree：先保证写快，同时读也较快

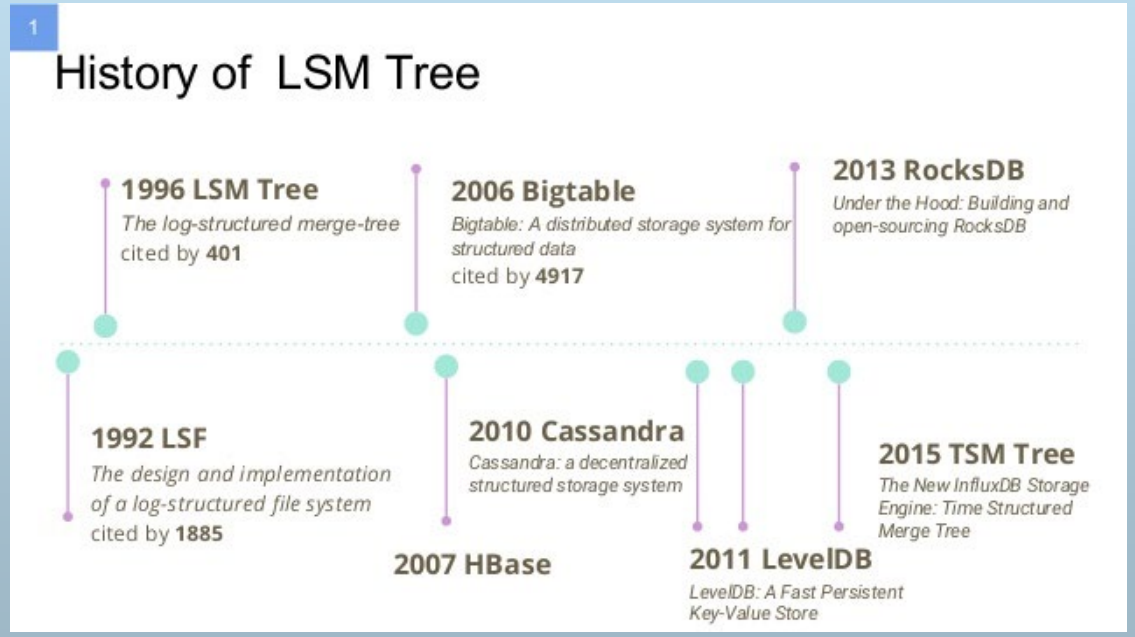


二、LSM-tree的设计思路

- **Log-Structured Merge Tree**
  - **同时结合内存结构和磁盘结构（page-based）**
    - 数据先写到内存结构，然后再写入磁盘
  - **Log-Structured：采用Append方式写磁盘数据**
  - **Merge Write：内存数据批量合并写入磁盘**
    - 将多个小的随机写转换为顺序写
  - **数据分层写入磁盘**
    - 避免一次批量写的数据量过大
      - 内存压力过大
      - 批量写时IO太多
  - **每一层的数据均有序**

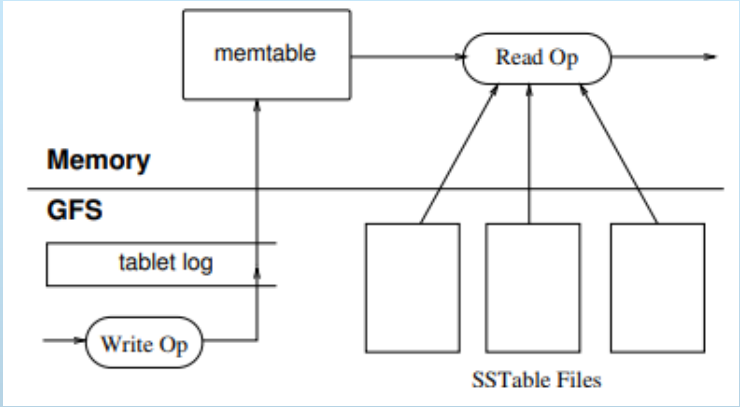
二、LSM-tree的设计思路

- 1996年由O'Neil等提出，借鉴了Log-Structured思想（1992）
- 2006年，Google的BigTable采用LSM-tree作为存储引擎
- 被很多NoSQL引擎采用：HBase（2007），Cassendra（2010），LevelDB（2011，Google），RocksDB（2013，Facebook）等

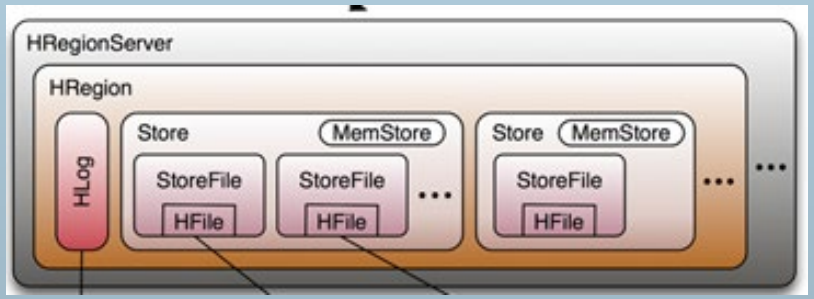


O'Neil P; et al. The log-structured merge-tree[J]. Acta Informatica, 1996, 33(04): 351-385.

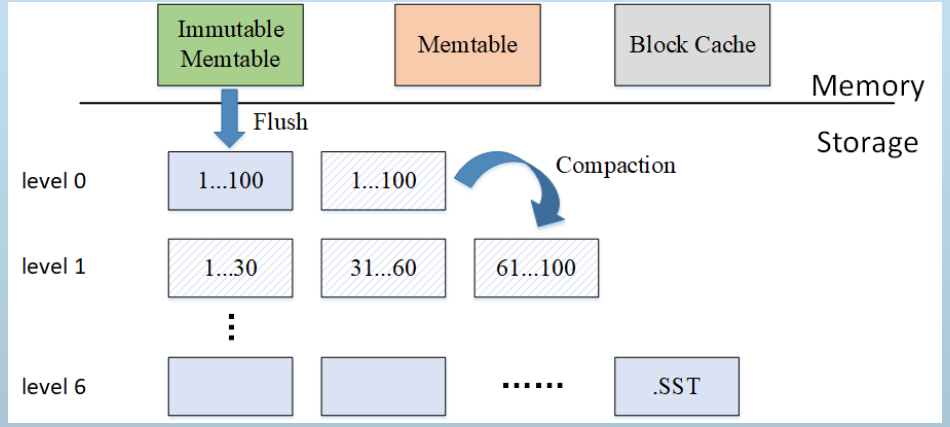
二、LSM-tree的设计思路



BigTable

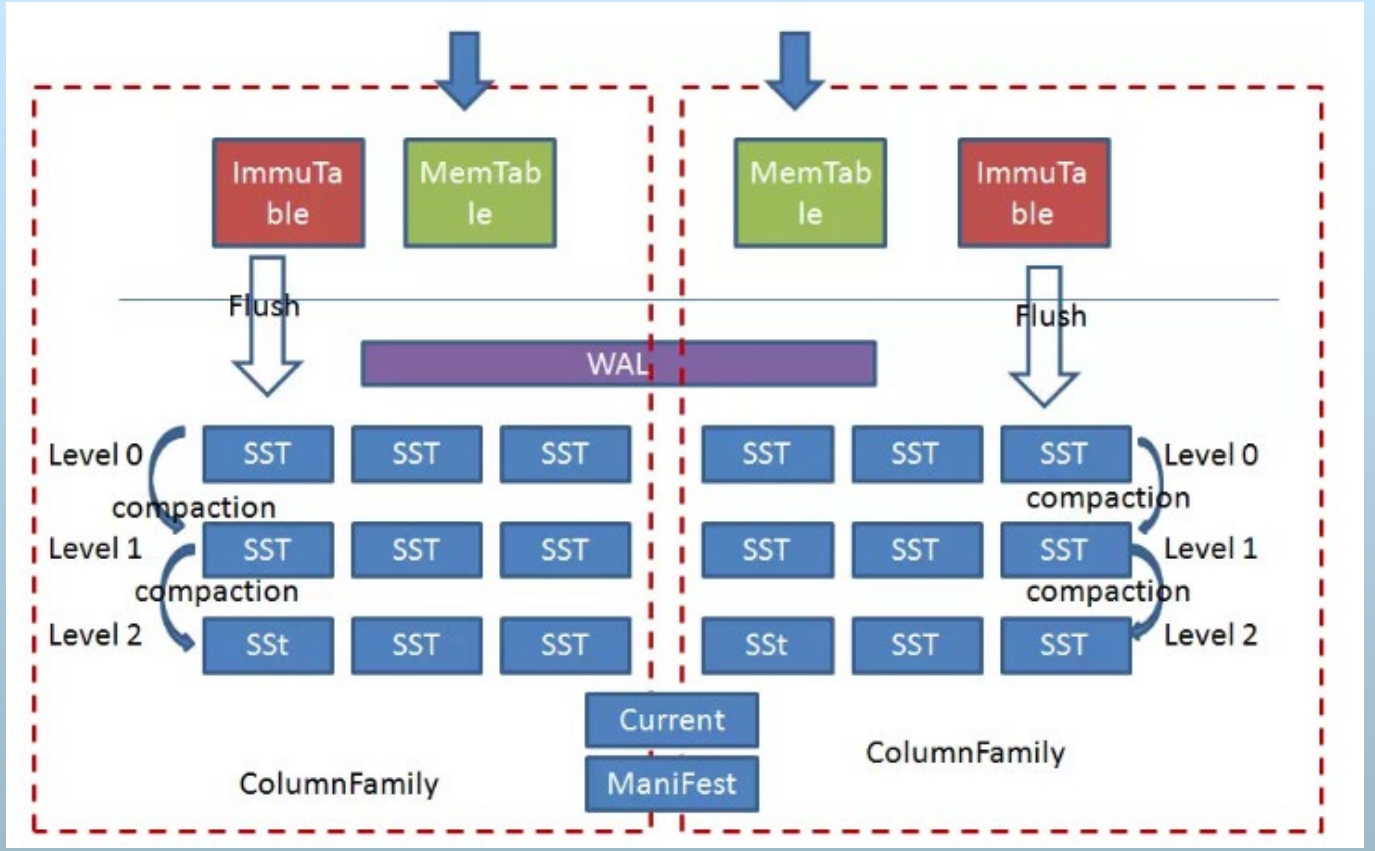


HBase



LevelDB

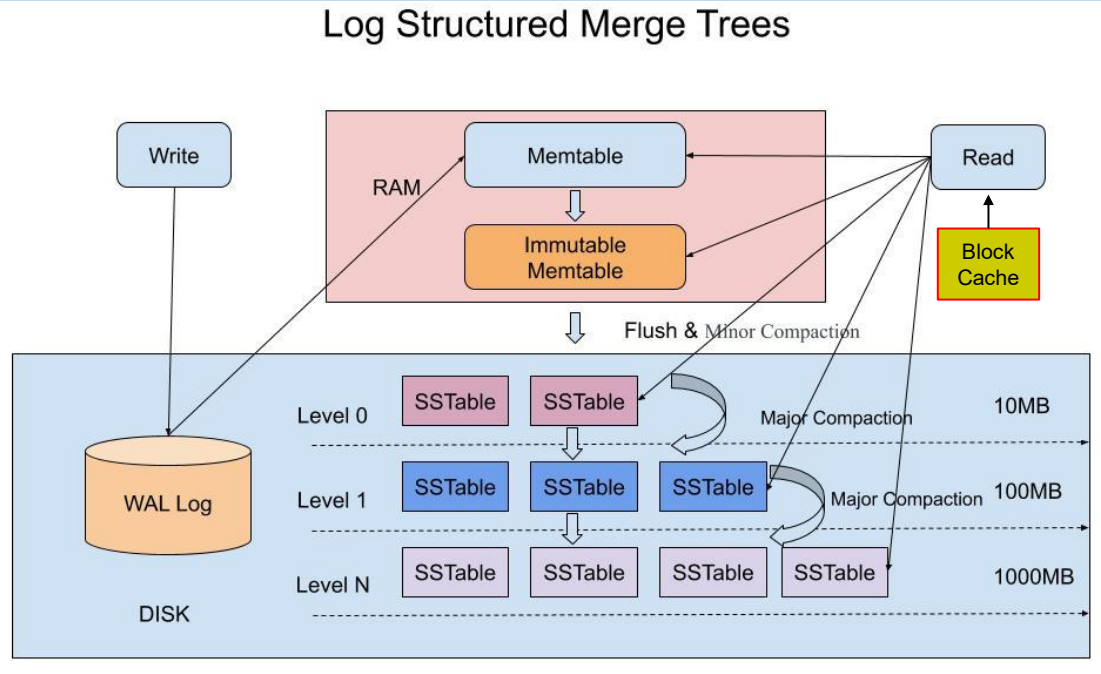
二、LSM-tree的设计思路



RocksDB

三、LSM-tree的实现：以LevelDB为例

- LevelDB是一款写性能十分优秀的可持久化的KV存储引擎，其实现原理是依据LSM-Tree（Log Structed-Merge Tree），由Google开源
  - 可视作BigTable的开源版本。数据总是先写入DRAM，则批量分层顺序写入Disk
  - 内存通过Memtable和Immutable Memtable两块区域轮转写，避免写阻塞

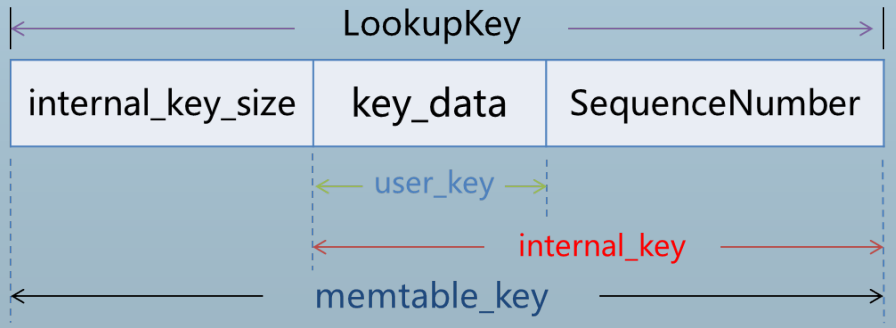


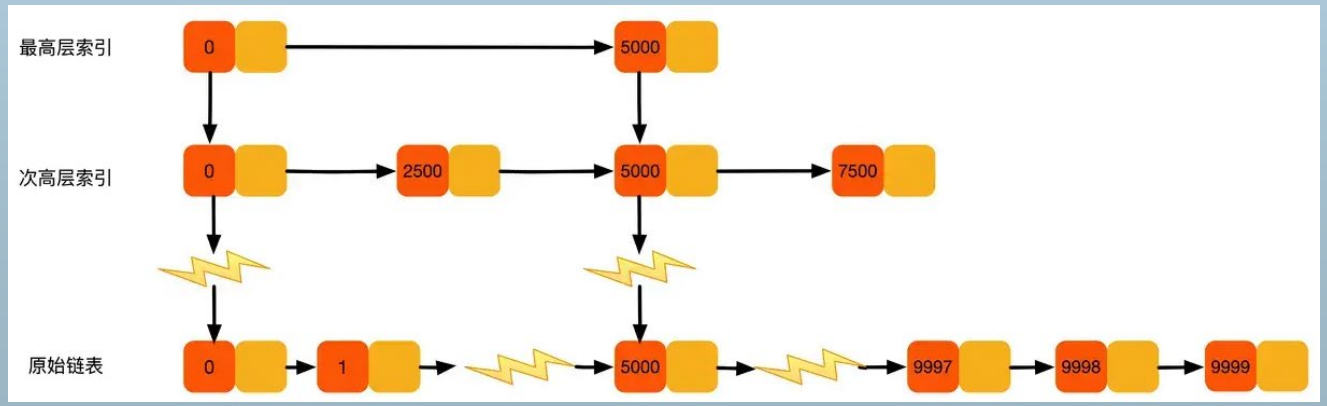
数据量小

数据量大

Memtable

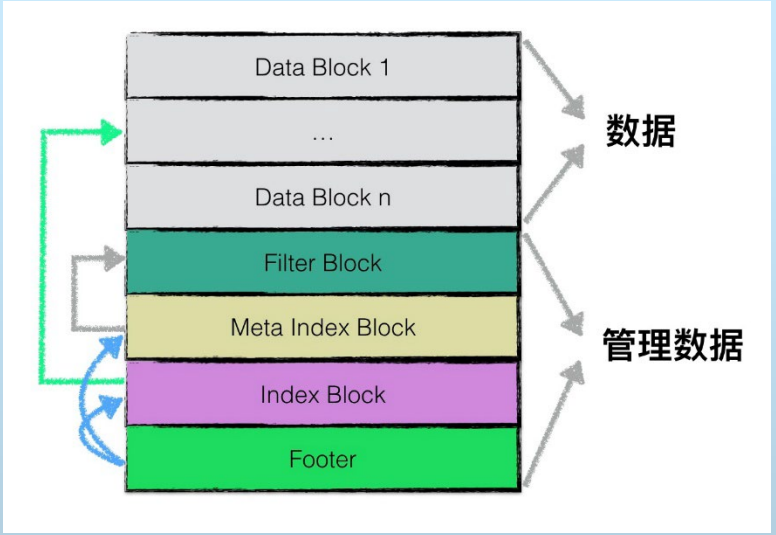
- KV记录组织成有序的skiplist结构。大小由option.write_buffer_size确定，默认4MB
  - Key和value均为变长字节流
    - SequenceNumber定义了version(56bits)和value_type(8bits)
    - 删除KV时通过插入value_type为删除标记的记录来表示（0表示删除，1表示有效）
  - 支持高效插入和二分查找的高效内存链表结构
  - 类似一种内存多级索引结构
  - 与内存B+树相比优势在于避免了插入操作时的平衡结构调整代价

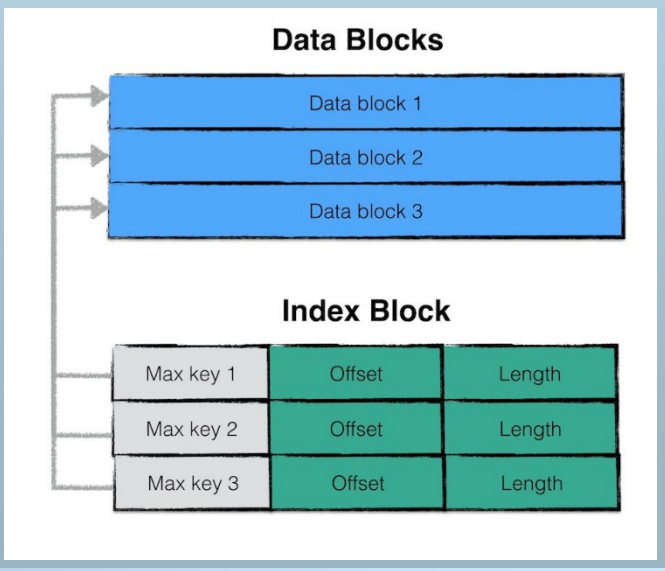




核心数据结构： SSTable（Sorted String Table）

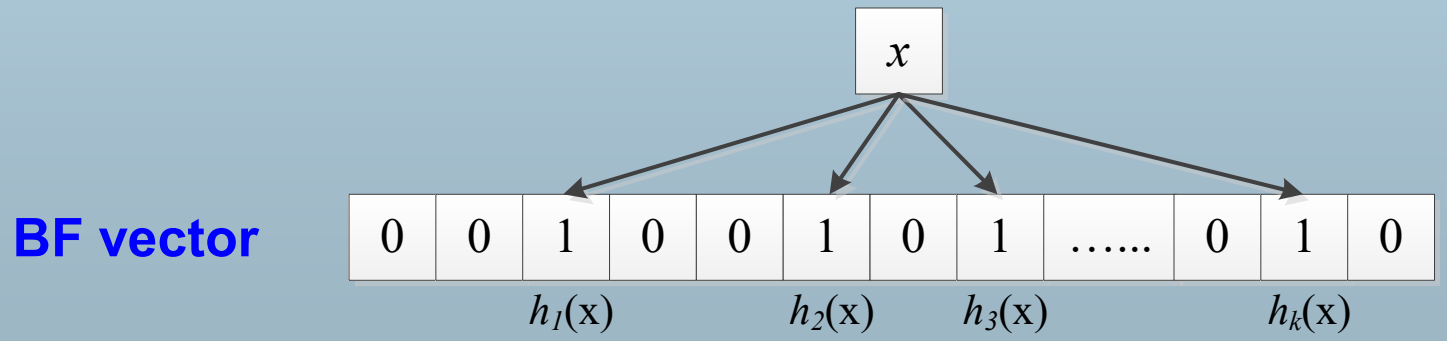
- Page-based：大小可配置，例如2MB
  - 所有KV在SST内都是有序存储
- Data Block: 用来存储key value数据对；
- Filter Block: 用来存储一些过滤器相关的数据（布隆过滤器；
- Meta Index Block: 用来存储filter block的索引信息（索引信息指在该sstable文件中的偏移量以及数据长度）；
- Index Block：用来存储每个data block的索引信息；
- Footer: 用来存储meta index block及index block的索引信息；





核心数据结构：SSTable（Sorted String Table）

- **Filter Block**
  - 使用Bloom Filter来加速存在性查询，减少无效IO
  - BF: A bit vector, each bit is calculated by a hash function returning 1 or 0
  - Insert a key $x \in S$: first calculate $h_i(x)$, then set BF[$h_i(x)$] = 1
  - Membership query "Is $y \in S$?": calculate $h_1(y), h_2(y), \ldots, h_k(y)$, compared with existing Bloom filters



核心数据结构：SSTable（Sorted String Table）

- **Filter Block**
  - **Bloom Filter的False Positive Rate (`fpr`)**
    - 当BF没命中时，key肯定不存在SST中
    - 但BF命中时，由于hash冲突特性可能出现假阳性 (实际SST中并不存在key)
      - 导致无效的**Index Block查询**

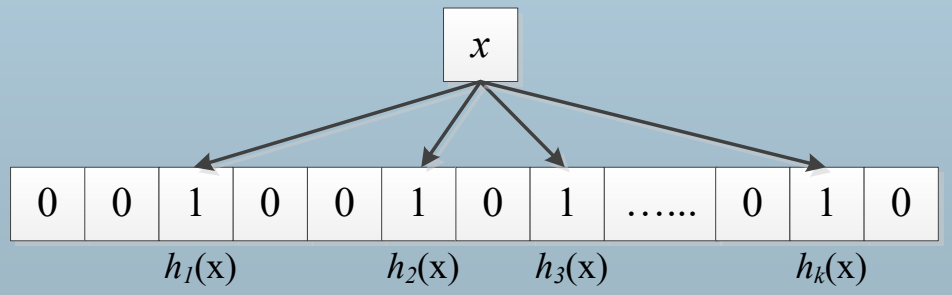
False-positive rate

$$f^{BF} = \left(1 - e^{-\frac{nk}{m}}\right)^k$$

- n: set size (number of keys in one node)
- m: bit-vector length
- k: hash-function count

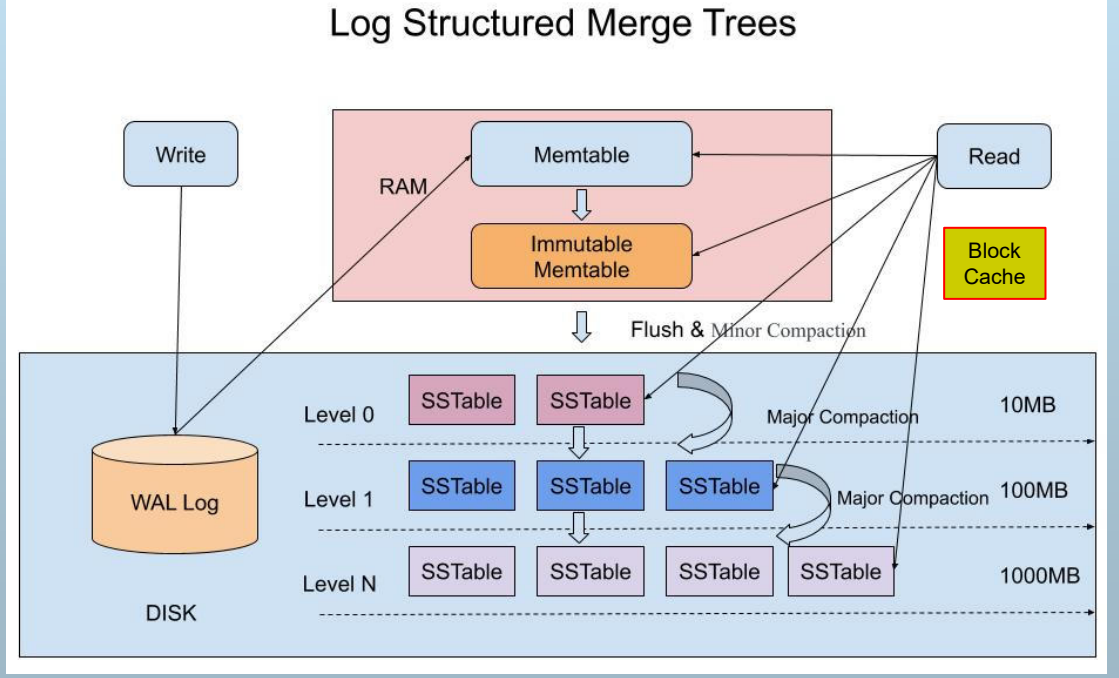
f^{BF} **is minimized when** $k = 0.7 \cdot \frac{m}{n}$

$$f^{BF} \approx 0.6185^{\frac{m}{n}},\ where\ k = 0.7 \cdot \frac{m}{n}$$



LSM-tree的Write过程

- 写数据的过程
  - 当收到一个写请求时，会先把该条数据记录在WAL Log里面，用作故障恢复
  - 当写完WAL Log后，会把该条数据写入Memtable
  - 当Memtable超过一定的大小后，会在内存里面冻结，变成不可更新的Immutable Memtable，同时新生成一个Memtable继续提供服务
  - 当Immutable Memtable数量超过阈值时Flush到磁盘上的L0层，此步骤也称为 Minor Compaction
    - Flush是顺序写
    - L0层的SSTable的key range可能会出现重叠，在层数大于L0层之后的SSTable，不存在重叠key
  - 当每层的SSTable的score超过阈值（score基于大小或者SST文件个数计算），触发Major Compaction，避免浪费空间
    - 注意由于SSTable都是有序的，所以采用merge sort进行高效合并。

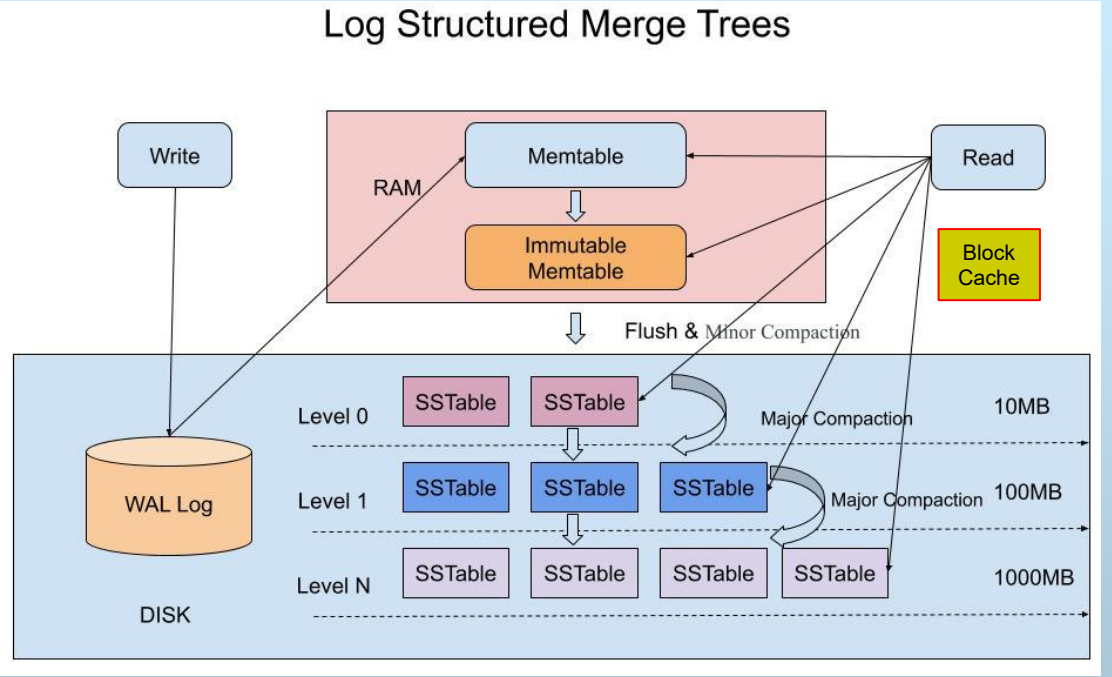


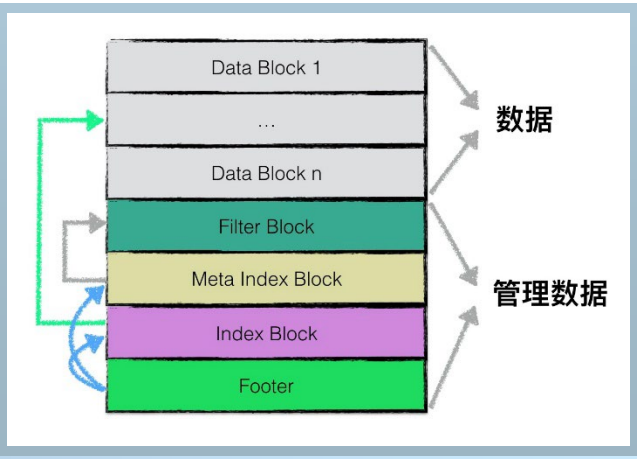
LSM-tree的Write过程

- 写数据的过程
  - 如果是随机的Insert操作，直接写入Memtable，很快
  - 如果是删除操作?
    - 因为SSTable不可更改，所以将删除操作转换为Insert操作，但是在key的SequenceNumber中使用一个1字节的特殊标记表示该key已经被删除
    - 如果有频繁的删除操作，将使得SSTable中存在很多无效的数据
    - 这些数据将在Compaction时回收

LSM-tree的Read过程

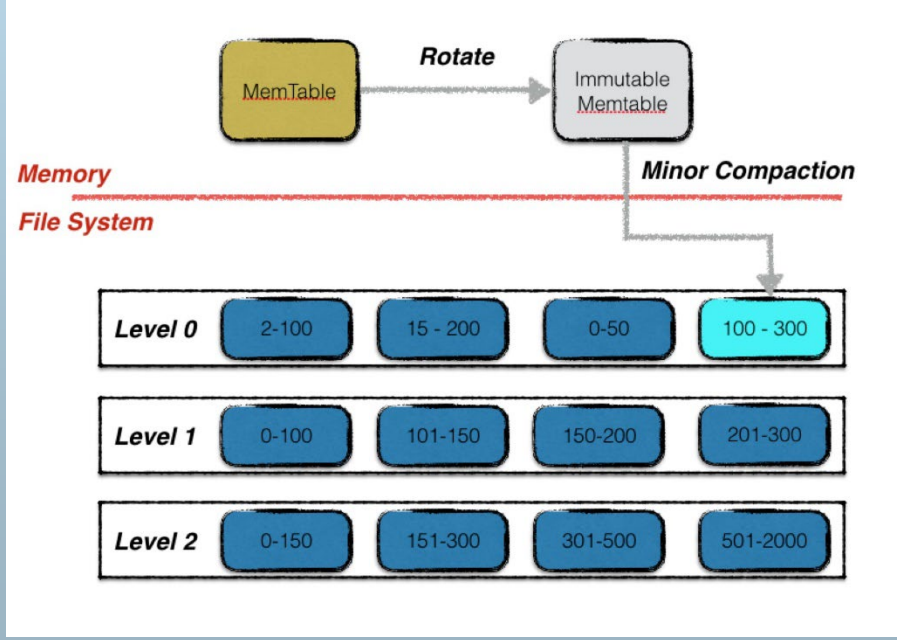
- 读数据的过程
  - 当收到一个读请求的时候，会直接先在Memtable和Immutable Memtable里查询，如果查询到就返回
  - 如果没有查询到就会依次下沉，直到把所有的Level查询一遍得到最终结果
  - 很显然，越往下层查询IO代价越高
  - LevelDB采用的读优化策略
    - 增加Block Cache
    - 增加Filter Block（使用Bloom Filter），如果Key在某个SST中不存在可以避免扫描SSTable
    - 增加Index Block，避免扫描整个SSTable的所有Block





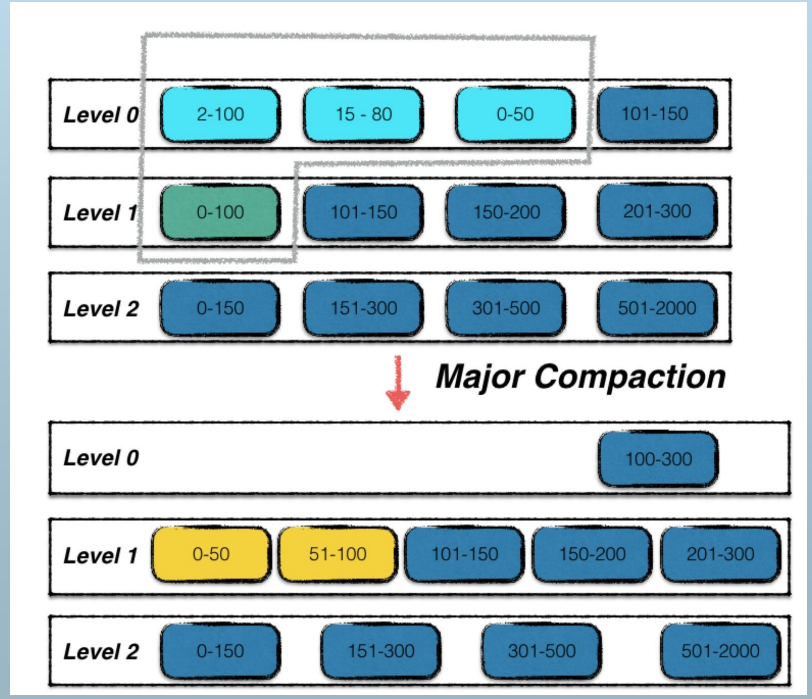
LSM-tree的Compaction操作

- **Compaction操作**
  - **Minor Compaction**
    - **Immutable MemTable** Flush -----> **SSTable (L0)**
    - **L0中的每个SSTable内部有序，但SSTable之间可能会存在重复的key（因为是整个Immutable Memtable dump到文件）**



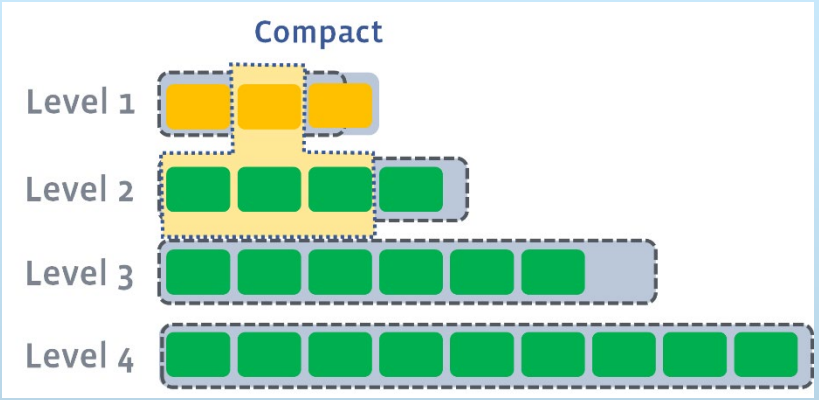
LSM-tree的Compaction实现

- **Compaction操作**
  - **Major Compaction**
    - L0 --> L1， L2 --> L3，…， SSTable (L_i) --> SSTable (L_{i+1})
    - 当Li层的score超过阈值时（基于文件数和大小计算）
      - L0的文件数不能太多，重复key range会影响读性能
      - L1开始每一层每次只会读一个SST，所以读性能不受文件数影响，但文件数过多会增加Compaction的IO代价
      - 因此，LevelDB中默认L0层10MB，L1开始按10倍数限制文件数，即L1为100MB，L2为1000MB
    - Compaction时执行垃圾回收，抛弃掉已经被删除的KV，减小SSTable数量和大小
  - **L0 --> L1的Major Compaction具体过程**
    - 选择L0层的第一个文件
    - 将L0与L1中所有与选中文件的key range重叠的文件都读入内存
    - 多路归并排序，并抛弃掉无效的KV
    - 按照SST大小重新写入L1层 （注意L1的SST大小可能与L0不同）
    - 如果L1层也触发了compaction条件，则会继续触发L1合并



LSM-tree的Compaction实现

- **Major Compaction**
  - L1 --> L2， L2 --> L3, ...
  - L1层开始每一层的SST文件key range不存在重叠

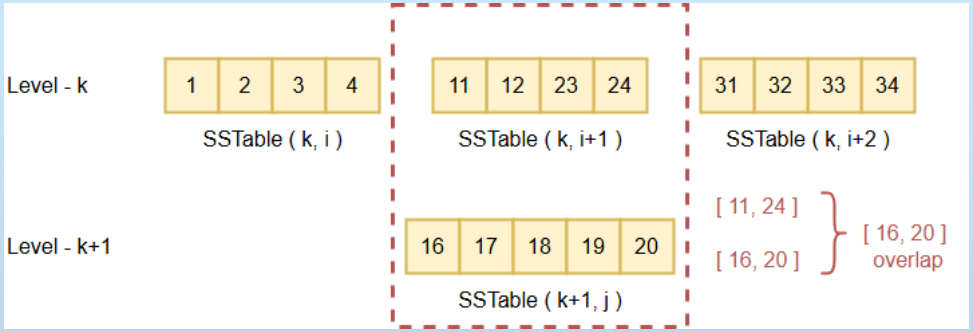


  - Size Compaction
    - 当Li层的score超过阈值时（基于文件数和大小计算），SST数目/Li层允许的大小
    - 选择Li层一个执行合并的SST文件（round-robin方式轮询）
    - 将Li+1层中所有与Li层选中的SST文件的key range都重叠的文件作为候选合并对象
    - 多路归并排序，并抛弃掉无效的KV
    - 按照SST大小重新写入Li+1层
    - 如果Li+1层也触发了compaction条件，则会继续触发合并

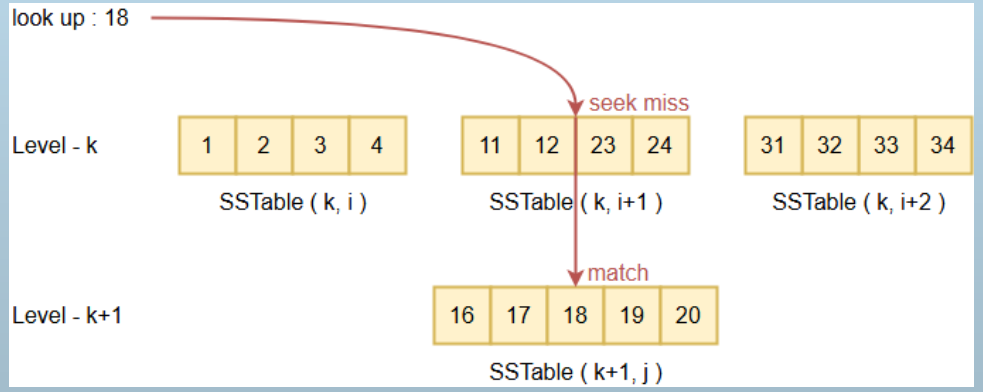
LSM-tree的Compaction实现

- **Major Compaction**
  - L1 --> L2， L2 --> L3, ...
  - L1层开始每一层的SST文件key range不存在重叠，但不同层之间可能存在重叠
  - 存在问题
    - 如果key查询总是在Level k没命中（seek miss），导致查询下推到Level k+1，则会增加查询IO代价，影响读性能
    - Index中只存储Max Key（非密集索引）
  - **Seek Compaction**
    - 为每个SST引入一个allowed_seeks参数
    - 每次SST触发seek miss， 并将allowed_seeks减1
    - 当allowed_seeks=0时，触发seek compaction
    - 将触发的SST以及下一层中重叠范围的SST读入内存
    - 排序后写入下一层

不同level之间的key overlap

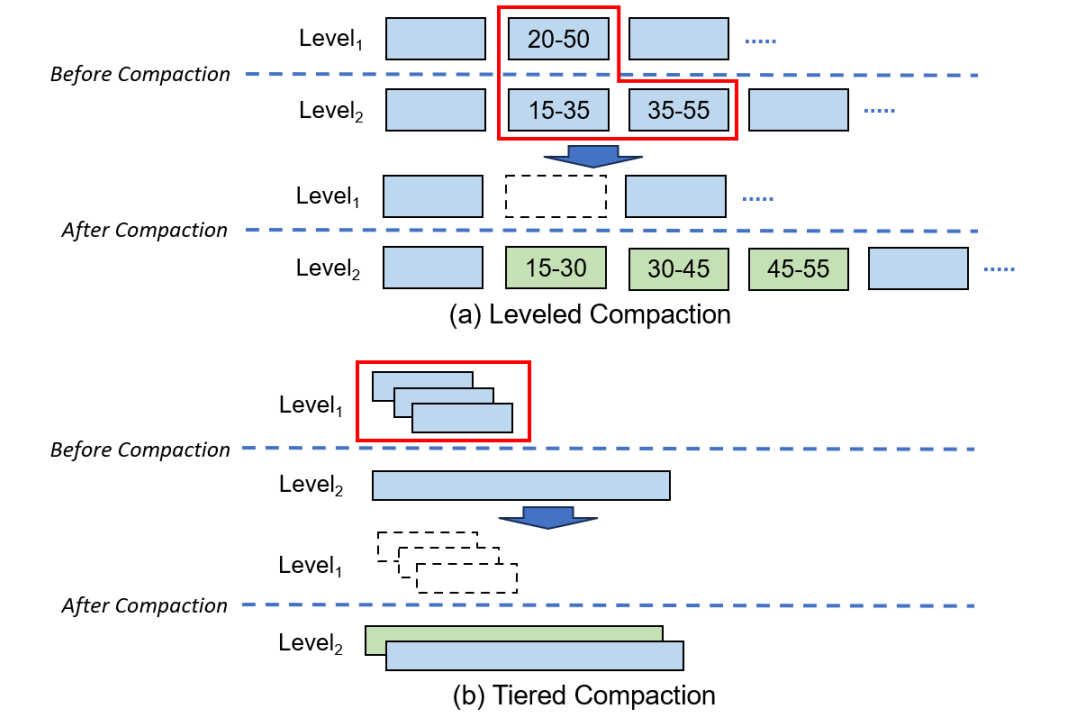


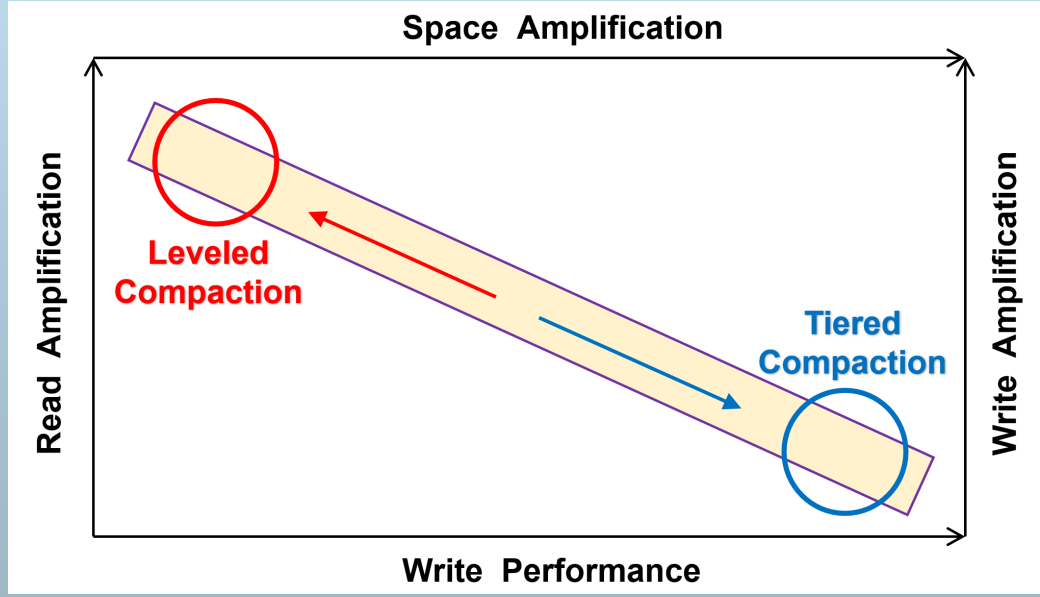
key overlap导致seek miss



LSM-tree的Compaction实现

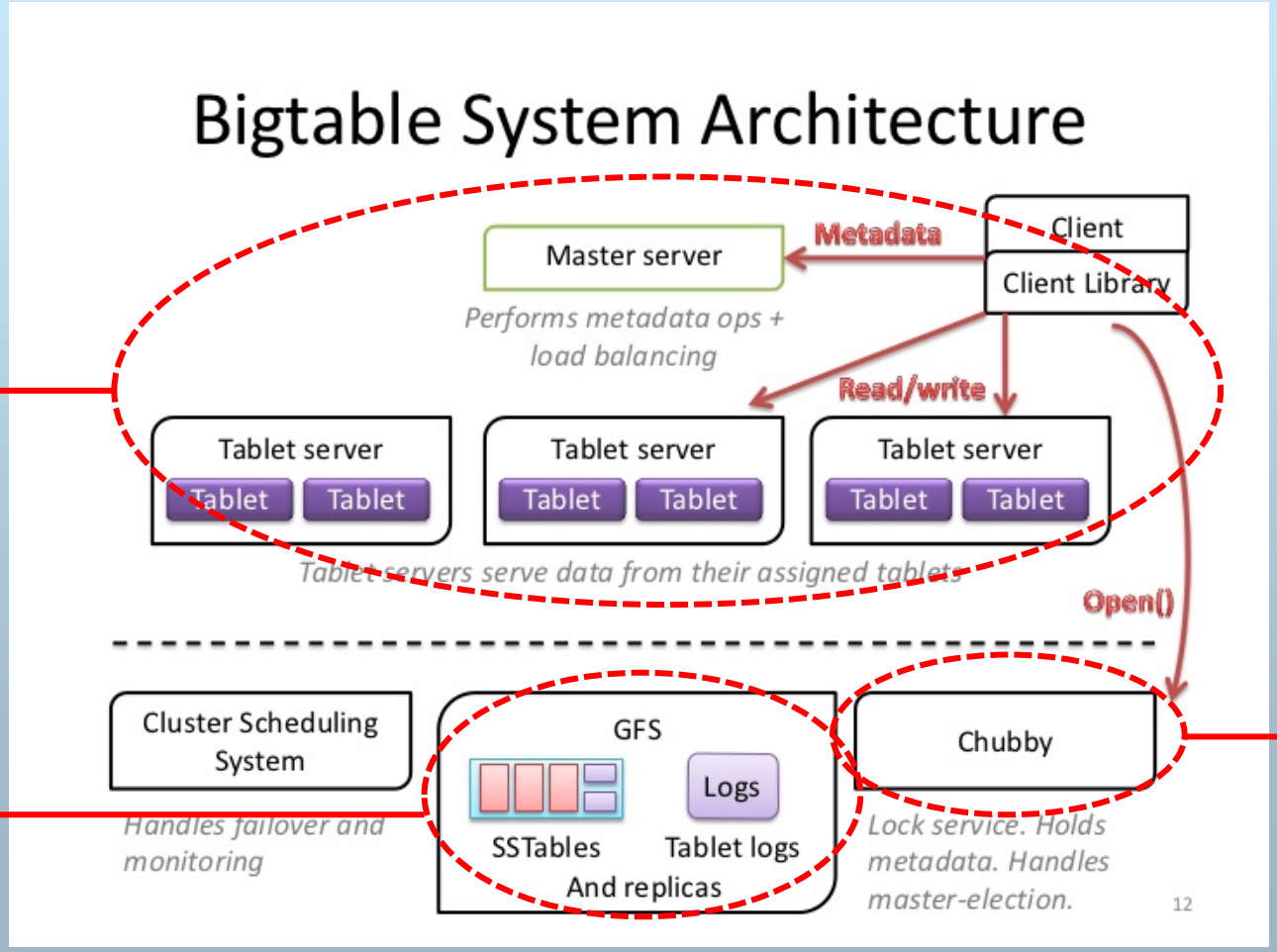
- **Size Compaction也叫Leveled Compaction**
  - **Compaction之后每一层形成了一个全局有序的run**
  - **如果每一层中compaction之后存在多个有序的run，则称为Tiered Compaction**
    - **Minor Compaction就是一种Tier Compaction**





四、LSM-tree在HBase上的实现

- **HBase：Hadoop dataBase**



HBase

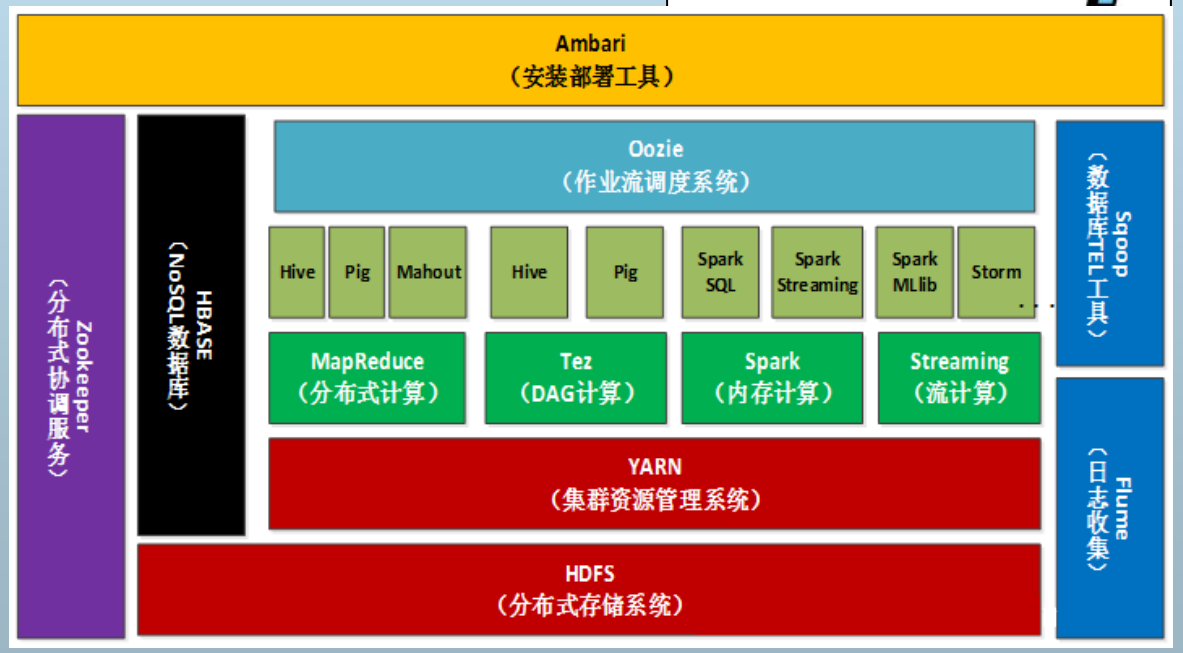
HDFS

Zookeeper

Google的GFS (SOSP'03)
和Bigtable (OSDI'06)

HBase简介

- **HBase：Hadoop dataBase**
  - 起源于Google Bigtable
  - Column Store，同时也使用了Key/Value存储
  - 分布式数据库，支持海量数据存储



HBase数据模型

- 从行存储到列存储

EmpId	Lastname	Firstname	Salary
1	Smith	Joe	40000
2	Jones	Mary	50000
3	Johnson	Cathy	44000

数据格式（行存储）：

1, Smith, Joe, 40000;

2, Jones, Mary, 50000;

3, Johnson, Cathy, 44000;

数据格式（列存储）：

1,2,3;

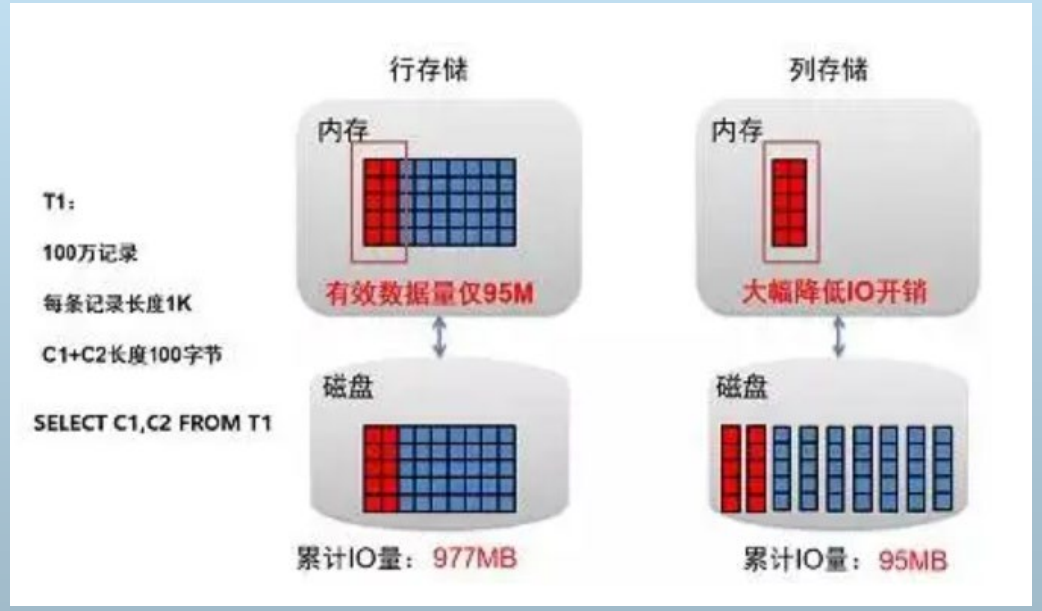
Smith, Jones, Johnson;

Joe, Mary, Cathy;

40000, 50000, 44000;

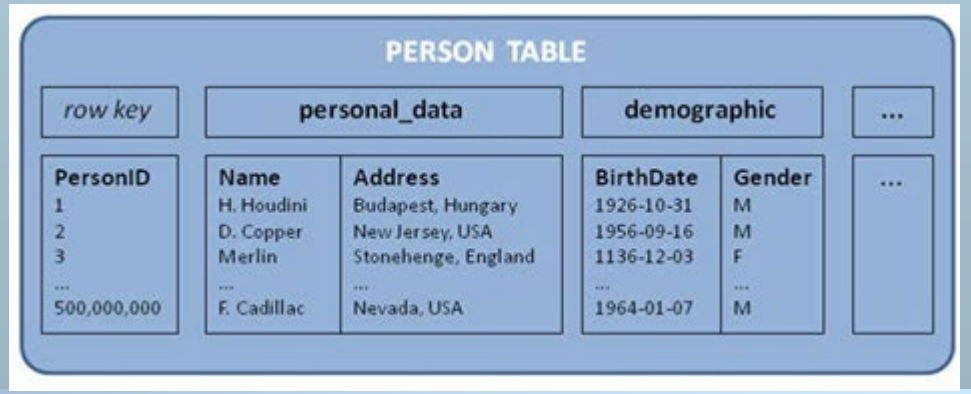
HBase数据模型

- 为什么用列存储？
  - 降低查询I/O代价



HBase相关概念

- **表（Table）：HBase采用表来组织数据，表由行和列组成，列划分为若干个列族**
- **行（Row）：每个HBase表都由若干行组成，每个行由行键（row key）来标识**
- **列族（Column Family）：一个HBase表被分组成许多“列族”（Column Family）的集合，每个列族有动态的列集合**
- **列限定符（Column Qualifier）：列族里的数据通过列限定符来定位**
- **单元格（Cell）：在HBase表中，通过行、列族和列限定符确定一个“单元格”（cell），单元格中存储的数据没有数据类型，都是字节流**
- **时间戳（Version）：每个单元格都保存着同一份数据的多个版本，这些版本采用时间戳进行索引**



Table的逻辑结构

行键	时间戳	列族contents	列族anchor
"com.cnn.www"	t5		anchor:cnnsi.com=”CNN”
	t4		anchor:my.look.ca="CNN.com"
	t3	contents:html="<html>..."	
	t2	contents:html="<html>..."	
	t1	contents:html="<html>..."	

Table的物理存储结构

- Table的每个列族存储在单独的文件中
- Row Key & Version numbers 在每个列族中复制
- 空cell不存储

列族contents

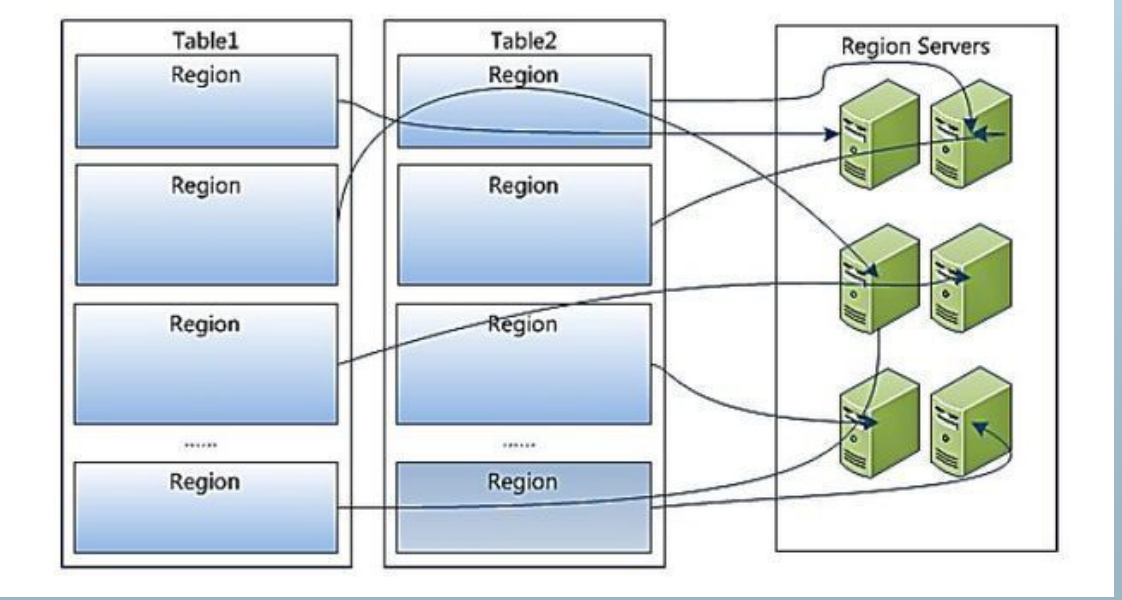
行键	时间戳	列族contents
"com.cnn.www"	t3	contents:html="<html>..."
"com.cnn.www"	t2	contents:html="<html>..."
"com.cnn.www"	t1	contents:html="<html>..."

列族anchor

行键	时间戳	列族anchor
"com.cnn.www"	t5	anchor:cnnsi.com=”CNN”
"com.cnn.www"	t4	anchor:my.look.ca="CNN.com"

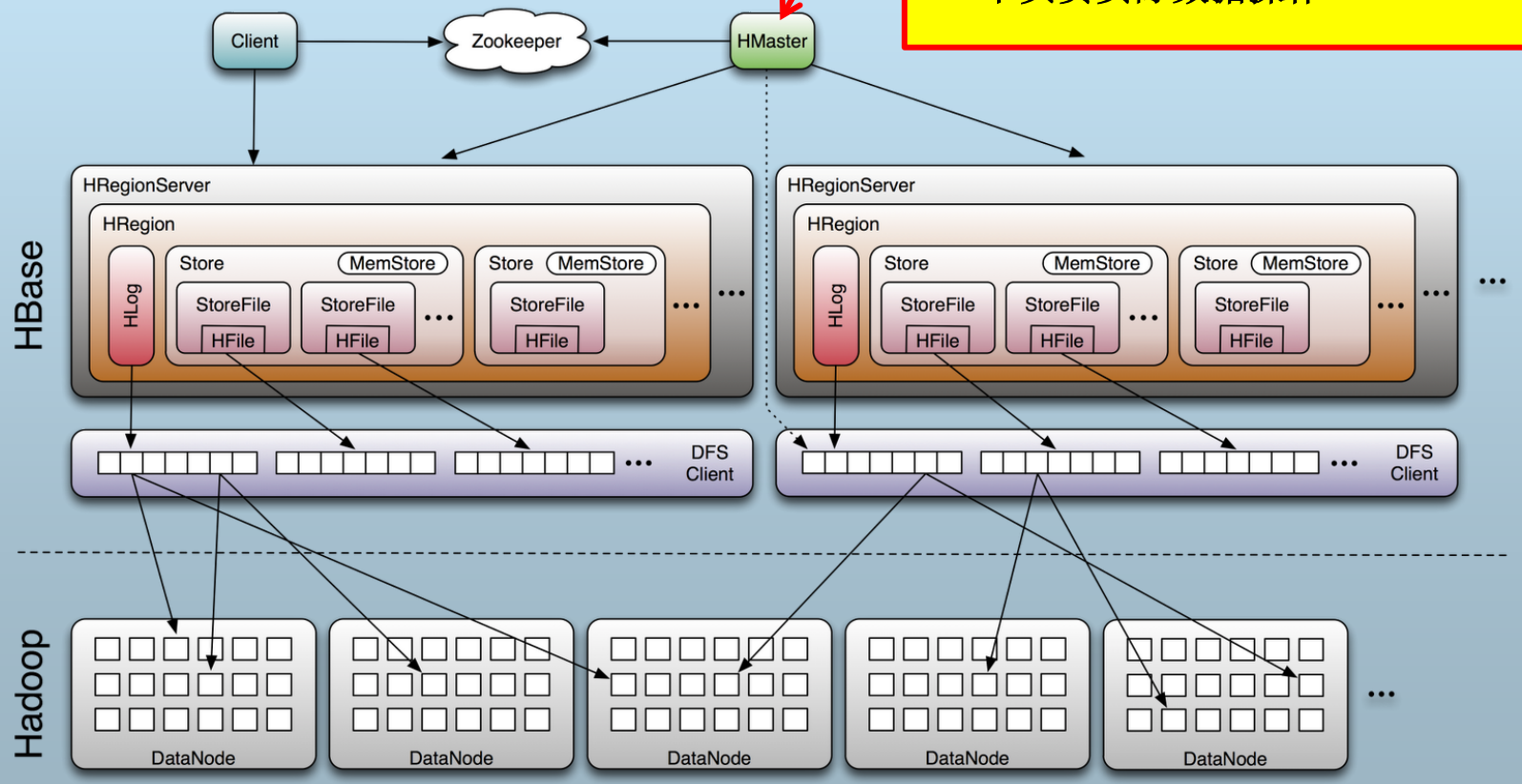
Table的物理存储结构

- Table和Region
  - Table中所有行按Row Key排序
  - 单个Table一开始只有一个Region
  - 随着记录越来越多，单个Region太大，达到阈值，分裂成2个Region
  - Region是HBase中分布式存储分配的最小单元
  - 不同Region分布在不同的Region Server上



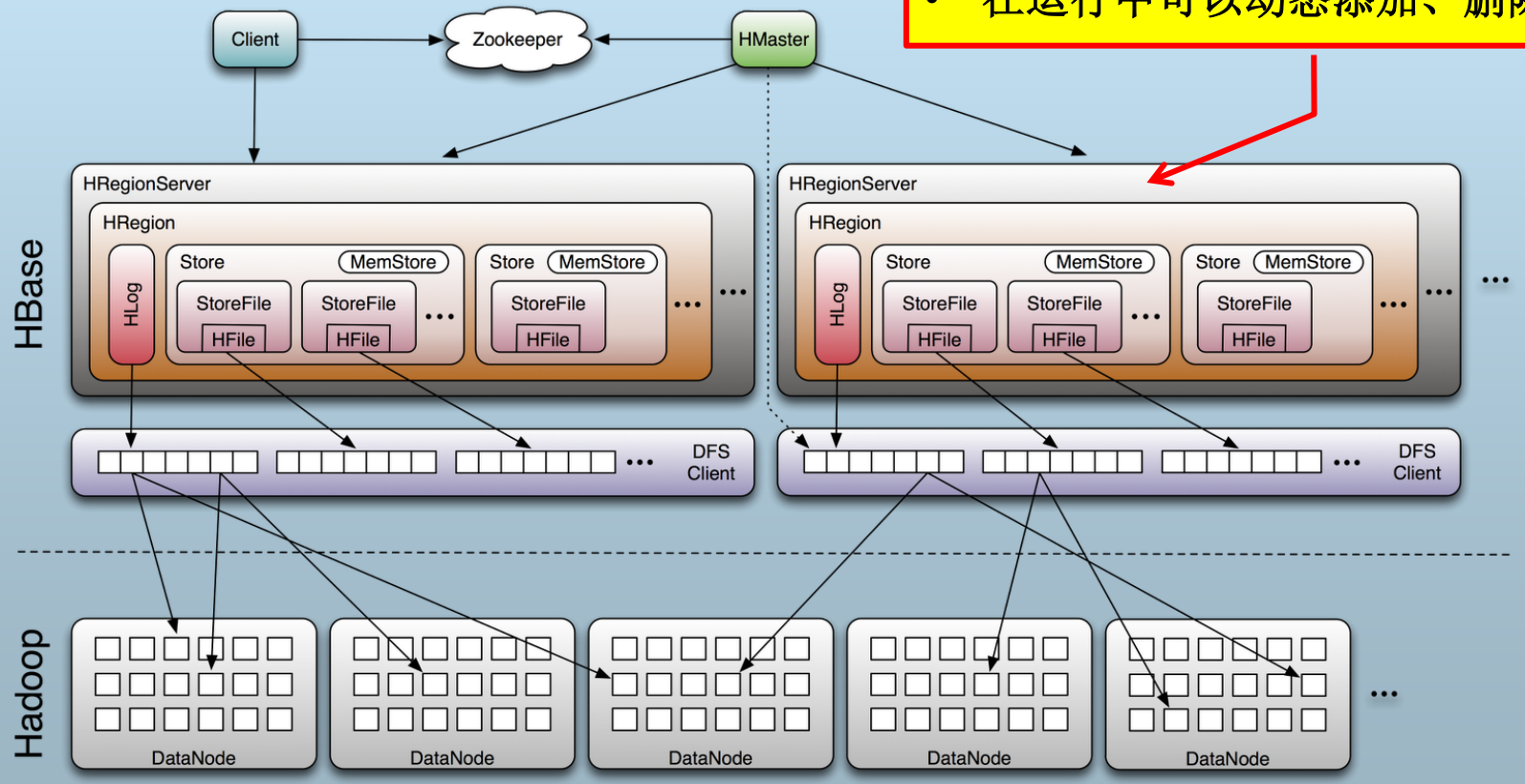
HBase架构

- 管理用户对table的创建、删除、修改操作
- 为Region server分配region
- 负责Region server的负载均衡
- 发现失效的Region server并重新分配其上的region（通过Zookeeper实现）
- 不负责实际数据操作



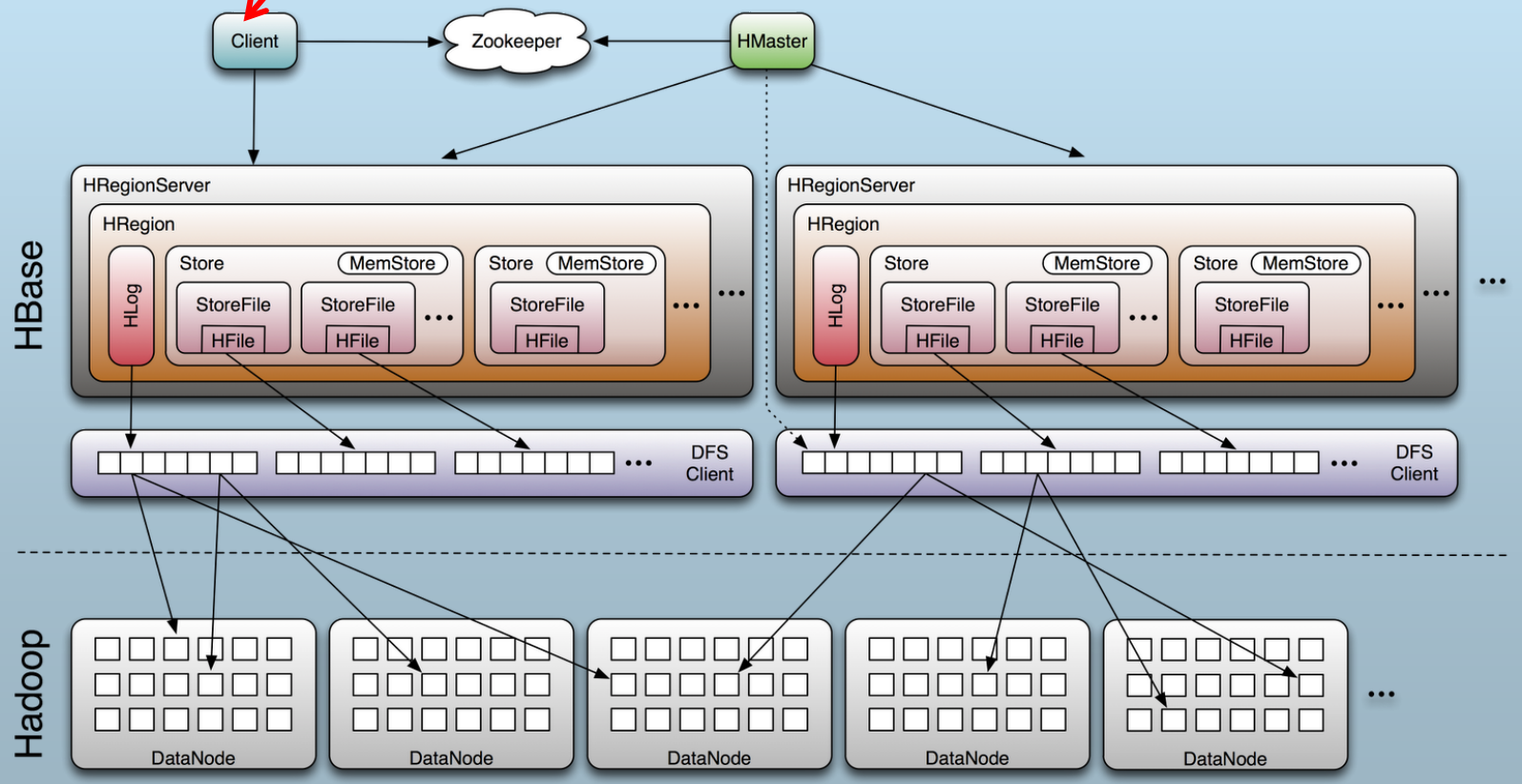
HBase架构

- 维护region，处理对这些region的IO请求，直接与client进行数据通信
- 负责切分（split）在运行过程中变得过大的region
- 对region进行compact操作
- 在运行中可以动态添加、删除



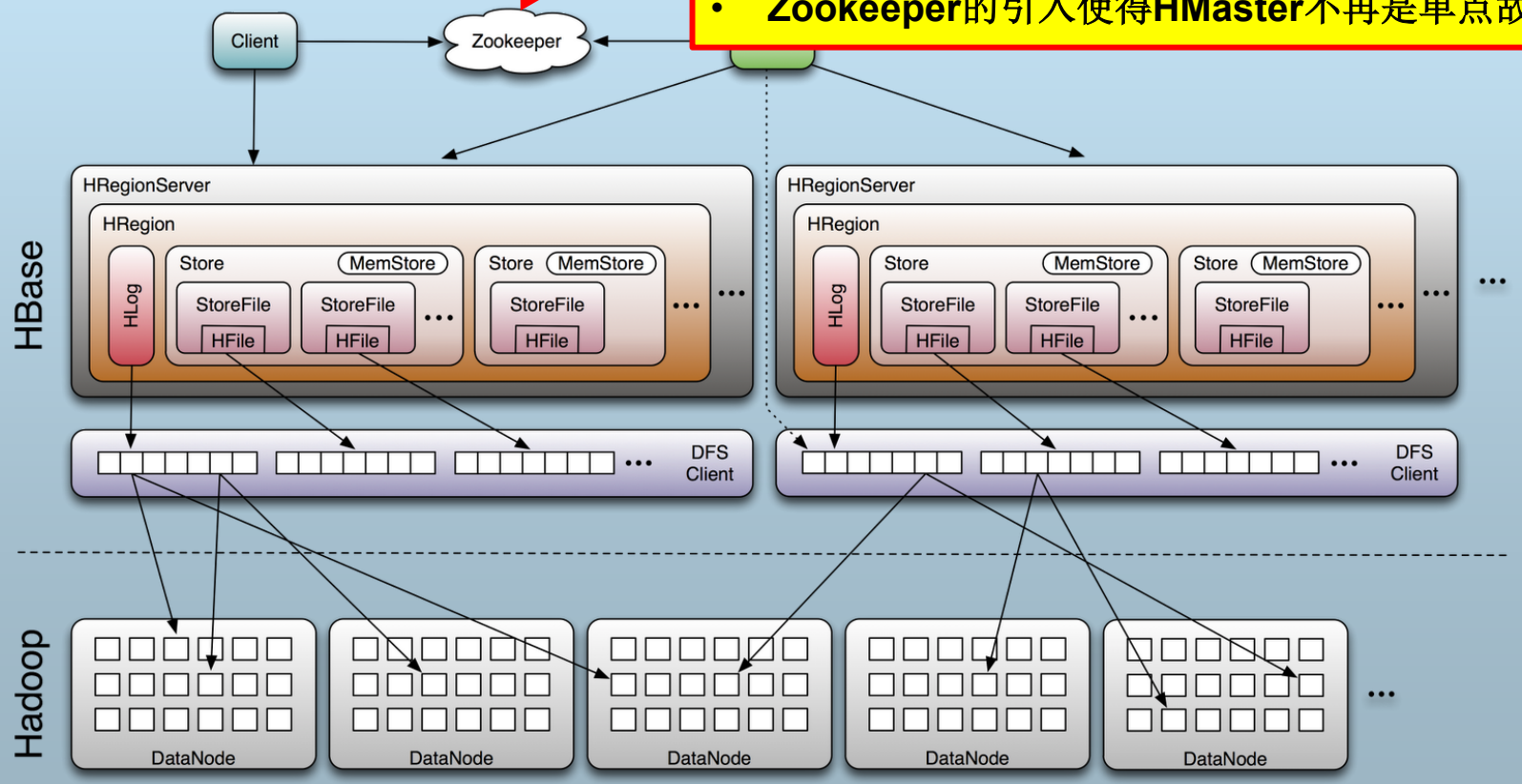
HBase架构

- 包含访问HBase的接口，并维护cache来加快对HBase的访问，比如region的位置信息



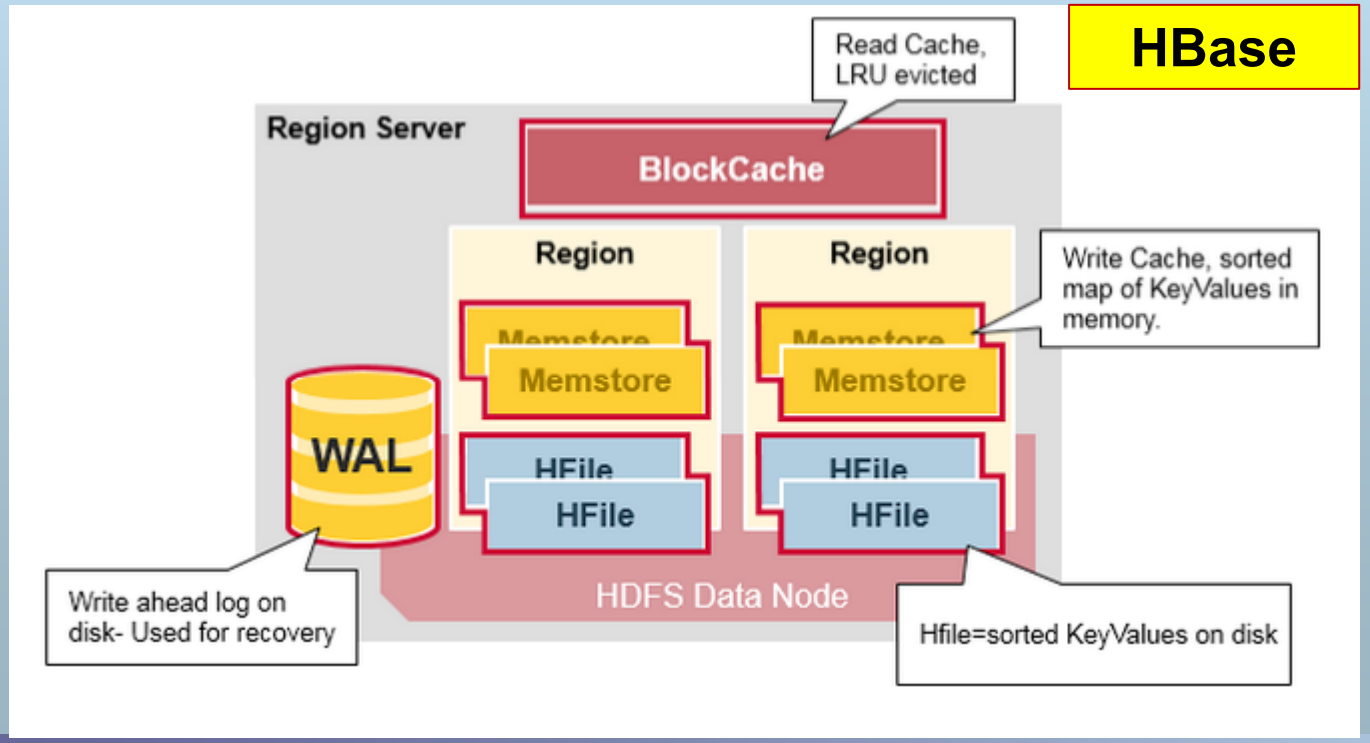
HBase架构

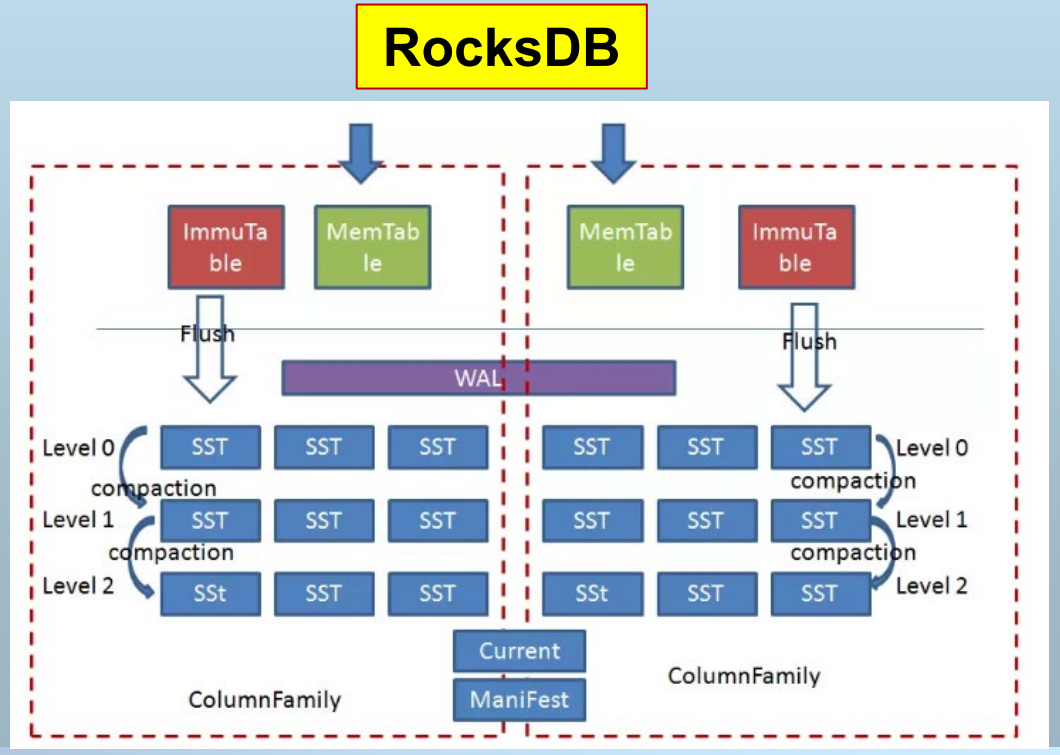
- 通过voting保证任何时候集群中只有一个HMaster，HMaster与Region Server启动时会向Zookeeper注册
- 存储所有Region的寻址入口
- 实时监控Region Server的上线和下线信息，并实时通知给HMaster
- 存储HBase的schema和table元数据
- Zookeeper的引入使得HMaster不再是单点故障



HBase vs RocksDB

- Memstore=Memtable+Immutable Memtable
- HFile=SST
- WAL和Block Cache相同
- Region=Column Family



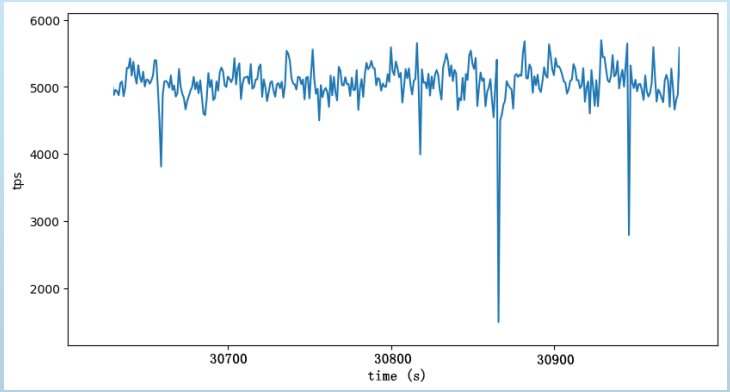


五、LSM-tree的总结

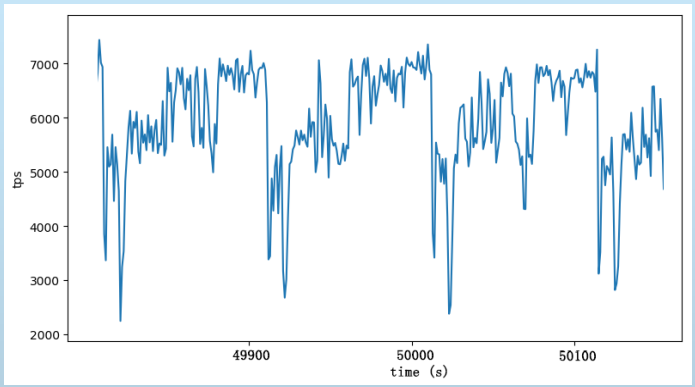
- 优点
  - 把随机写操作转化为顺序写，支持高吞吐的写（尤其适合分布式大数据应用场景）
  - 采用Append方式写数据，读写操作相互独立，可以支持高并发应用
  - 适合写多读少的应用
- 缺点
  - 读性能较差，尤其是range query
  - 空间放大严重，需要Compaction才能回收空间
  - Compaction操作导致系统性能抖动
    - 系统资源消耗高
      - I/O代价（写放大、I/O带宽消耗）
      - CPU和内存消耗
    - Block Cache失效

六、Compaction优化

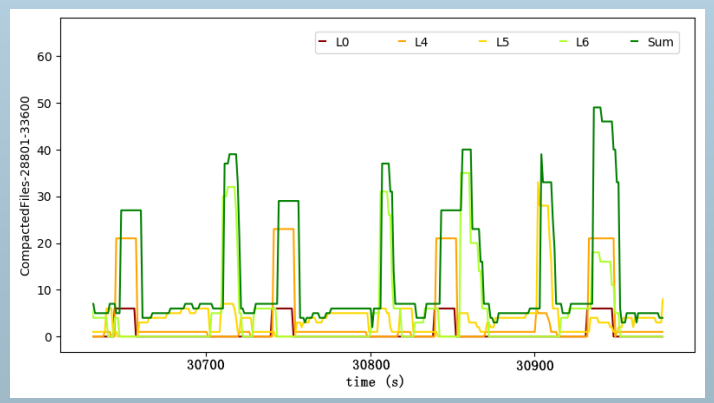
TPC-C@sysbench

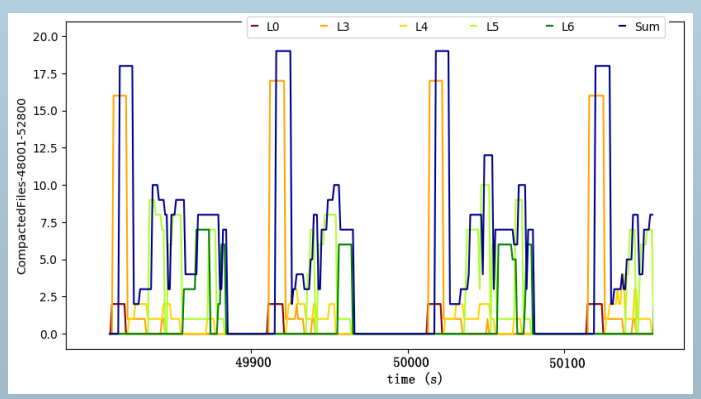


OLTP@sysbench



Throughput





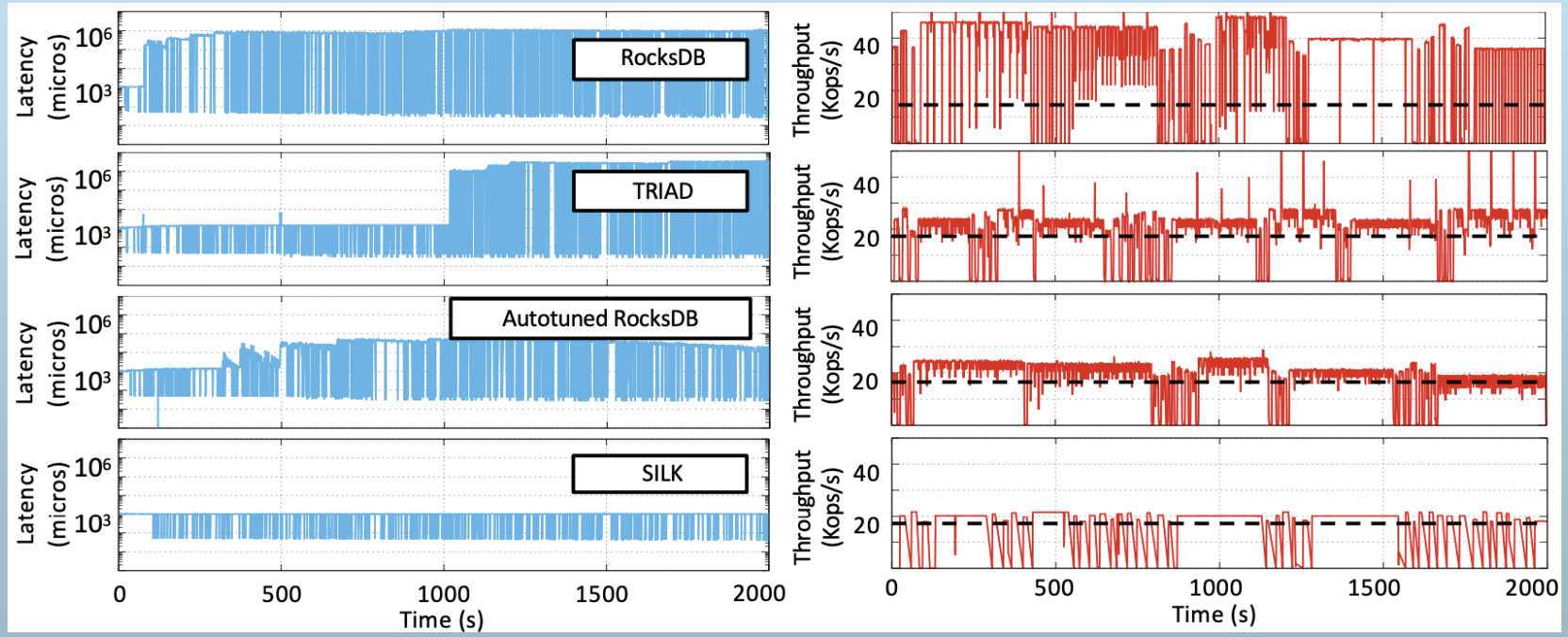
正在Compact的SSTable文件数

Compaction对LSM-tree性能的影响

六、Compaction优化

一些Compaction优化工作

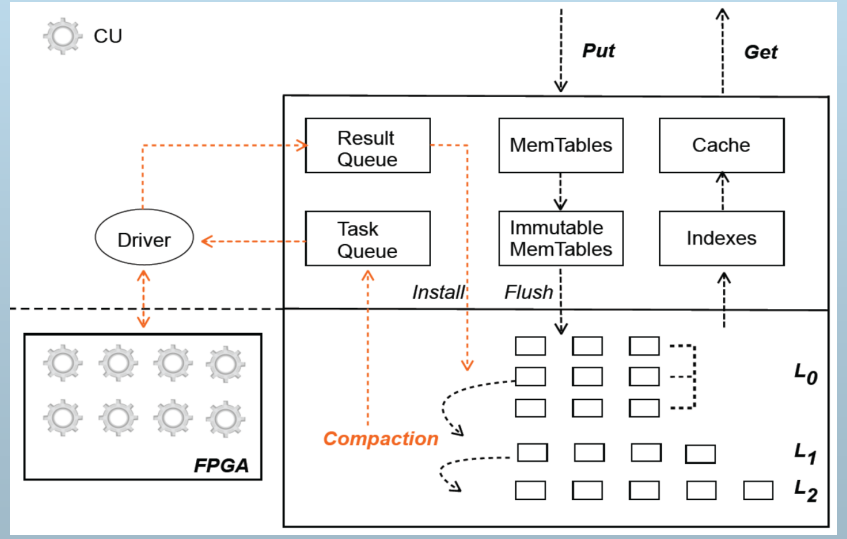
- **SILK （ATC 2019, Best Paper）**
  - 针对周期性有峰值的负载，高负载时延迟下层合并但继续上层合并，低负载时再执行下层合并

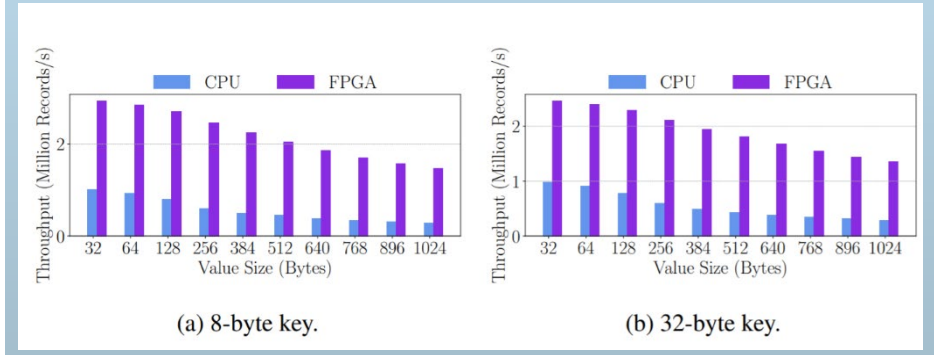


Oana Balmau et al. SILK: Preventing Latency Spikes in Log-Structured Merge Key-Value Stores. ATC 2019 (**Best Paper**)

一些Compaction优化工作

- **FPGA-Accelerated Compaction (FAST'20)**
  - 理论上有助于平滑LSM-tree性能的抖动
  - 需要专用的FGPA及驱动支持
  - Offloading Compaction to DPUs?

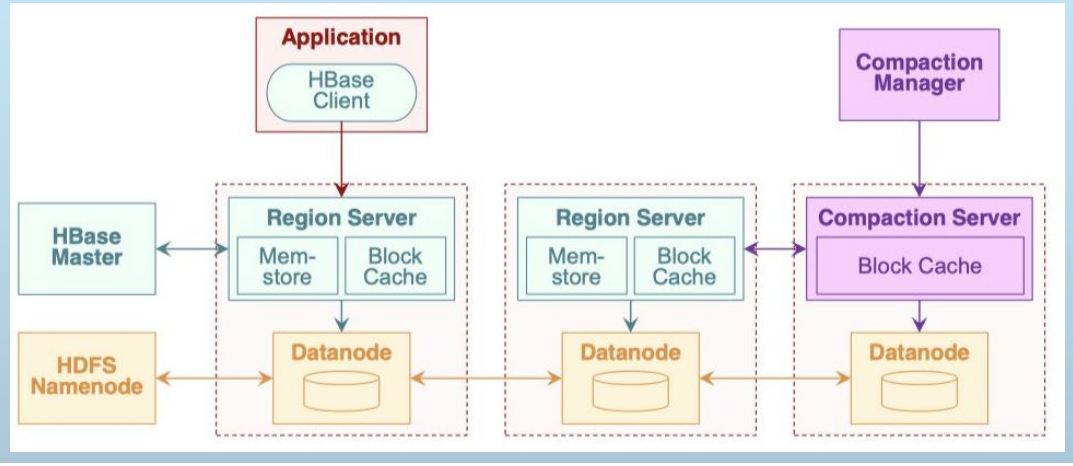


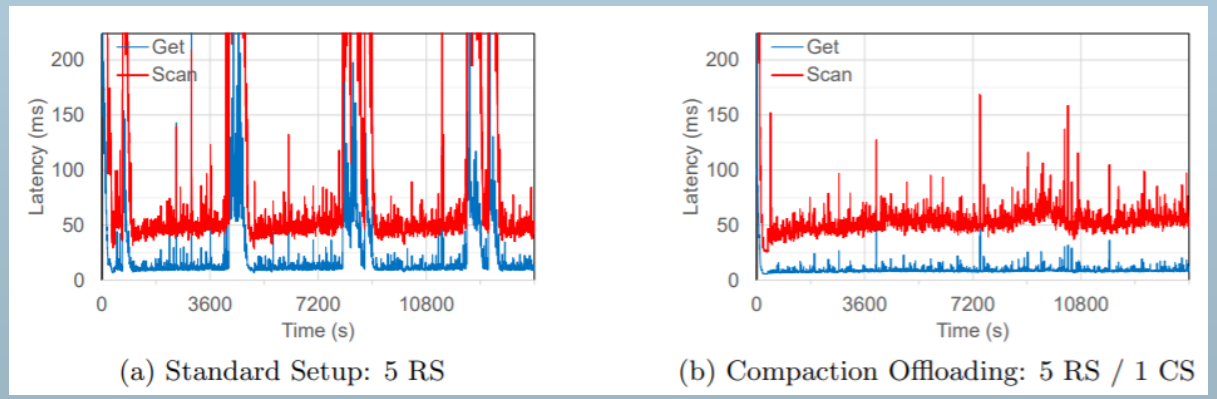


Teng Zhang et al. FPGA-Accelerated Compactions for LSM-based Key-Value Store. FAST 2020

一些Compaction优化工作

- **Offload Compaction (VLDB'15)**
  - 借助独立的Compaction Server来完成合并，从而不消耗本地Server的IO和CPU资源

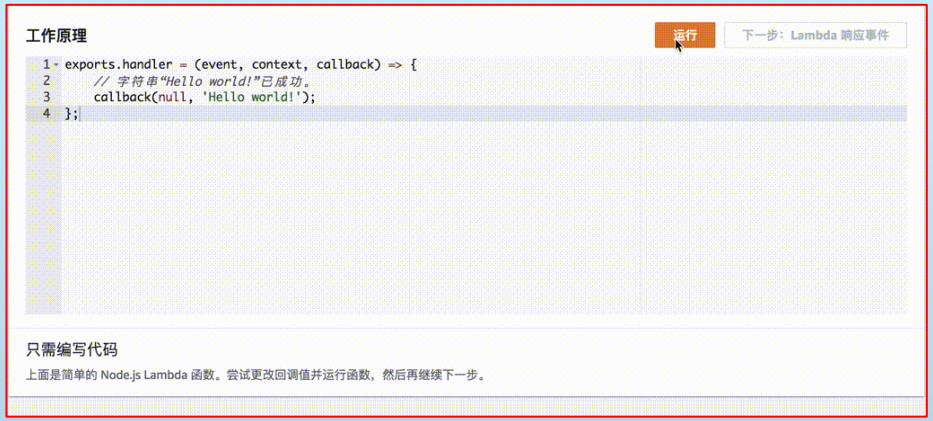


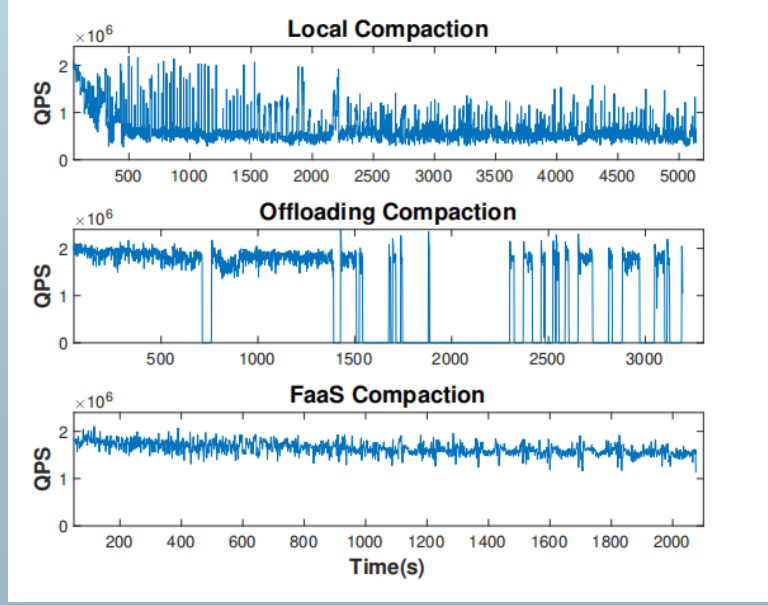


Muhammad Yousuf Ahmad et al. Compaction management in distributed key-value datastores. VLDB 2015

一些Compaction优化工作

- **FaaS Compaction (CIKM'21)**
  - **FaaS: Function as a Service**
    - 函数即服务，新型Elastic Computing
    - 根据需求自动扩/缩容，按需计算成本
    - 2014年提出，大厂迅速跟进
      - Amazon AWS Lamda (2014)
      - Google Cloud (2016)
      - 阿里、腾讯、字节 ...
  - **思路**
    - 当需要执行Compaction时，将任务发送给FaaS实例执行
    - 从而不占用本地服务器的资源，保证系统性能的稳定性

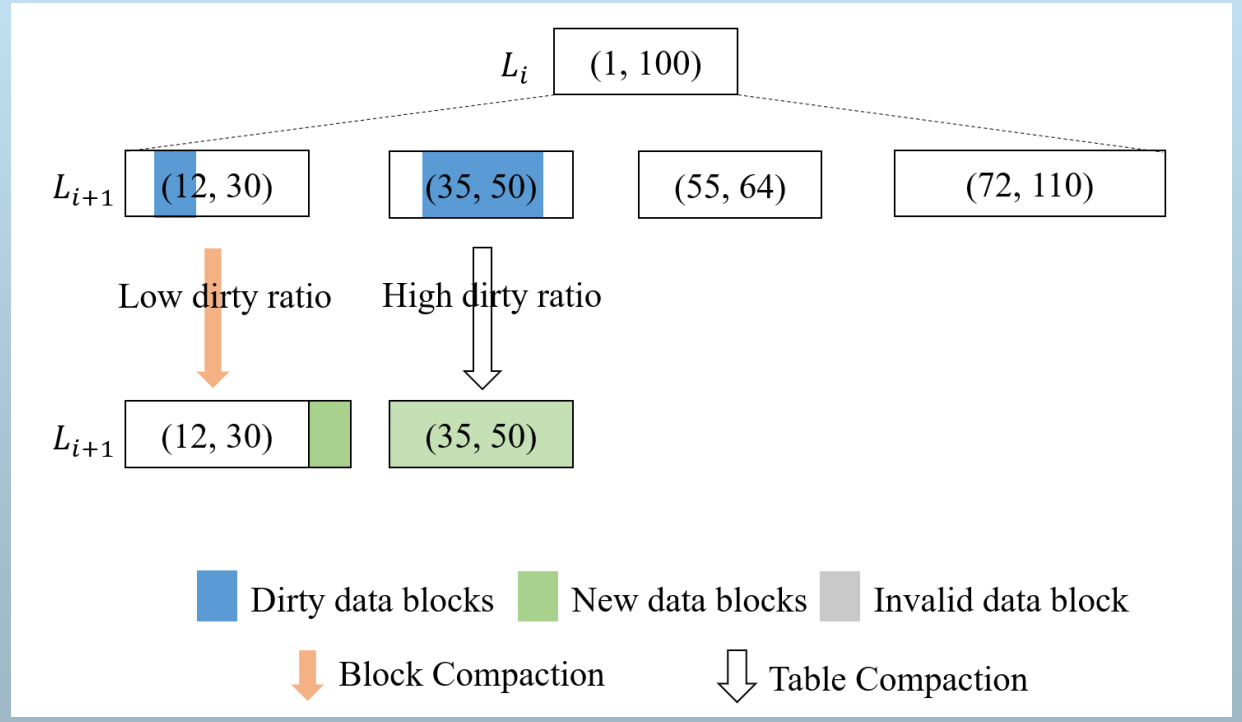


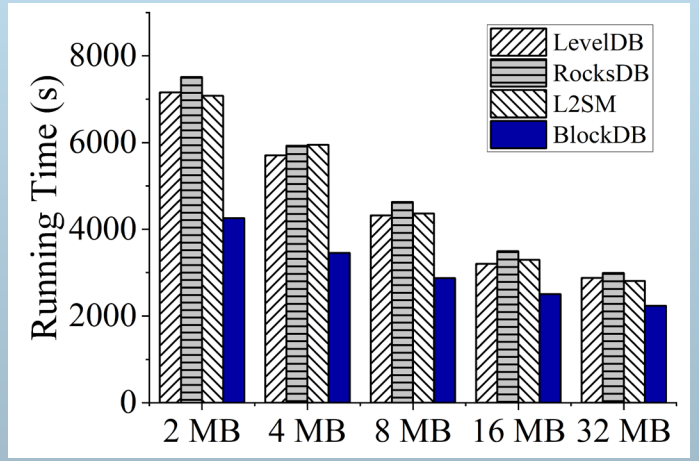


Jianchuan Li et al Elastic and Stable Compaction for LSM-tree: A FaaS-Based Approach on TerarkDB. CIKM 2021

一些Compaction优化工作

- **BlockDB：键值数据库引擎的自适应块合并 (ICDE'22)**
  - **自适应合并：动态选择合并粒度：SSTable或者Block，减少不必要的数据重写，降低缓存失效率**



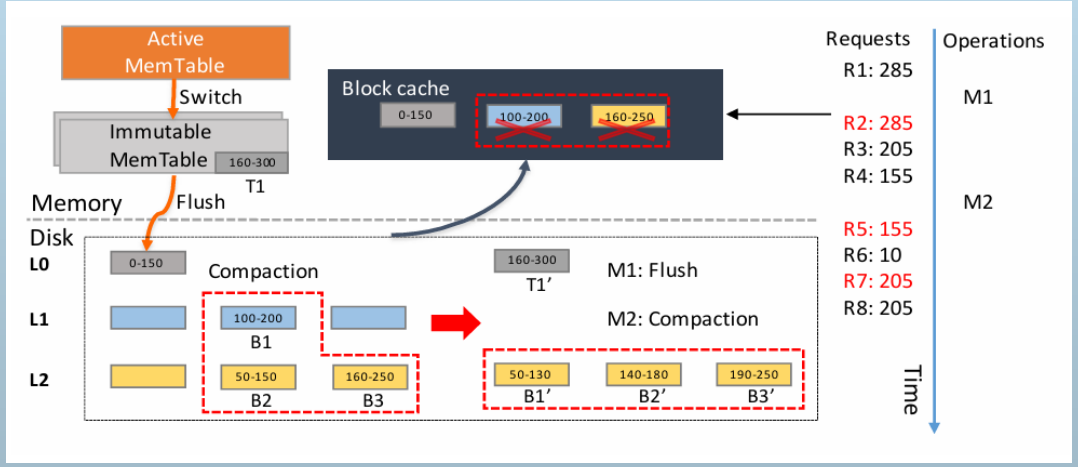


Xiaoliang Wang, et al.: **Reducing Write Amplification of LSM-Tree with Block-Grained Compaction.** ICDE 2022: 3119-3131

一些Compaction优化工作

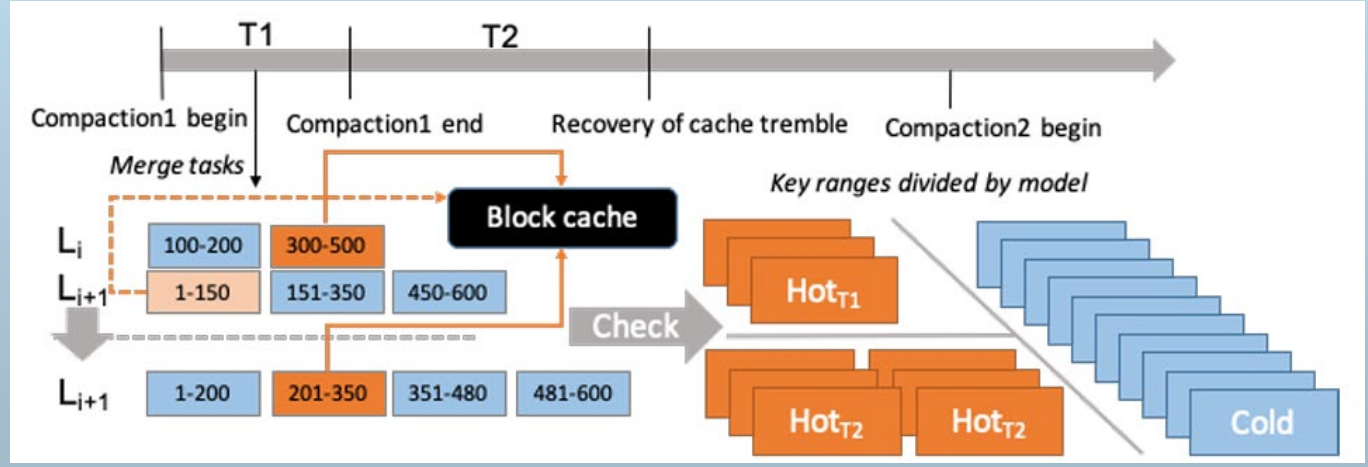
- **Leaper：通过预取解决Compaction导致的Block Cache失效问题 (VLDB'20)**
  - **学习型缓存预取：通过预测存储引擎中的热数据并且在访问之前将他们预取到Block Cache中来避免Compaction导致的Block Cache Invalidation问题**

Block Cache Invalidation



Leaper

将合并影响的块换出(T1)，将受影响的数据中的热数据预取入缓存(T2)



Lei Yang, et al.: **Leaper: A Learned Prefetcher for Cache Invalidation in LSM-tree based Storage Engines.** Proc. VLDB Endow. 13(11): 1976-1989 (2020)

本章小结

- B+-tree的问题
- LSM-tree的设计思想
- LSM-tree的实现
- LSM-tree的优缺点