

Architecture of Enterprise Applications 17

Log-Structured Database

Haopeng Chen

REliable, INtelligent and Scalable Systems Group (REINS)

Shanghai Jiao Tong University

Shanghai, China

<http://reins.se.sjtu.edu.cn/~chenhp>

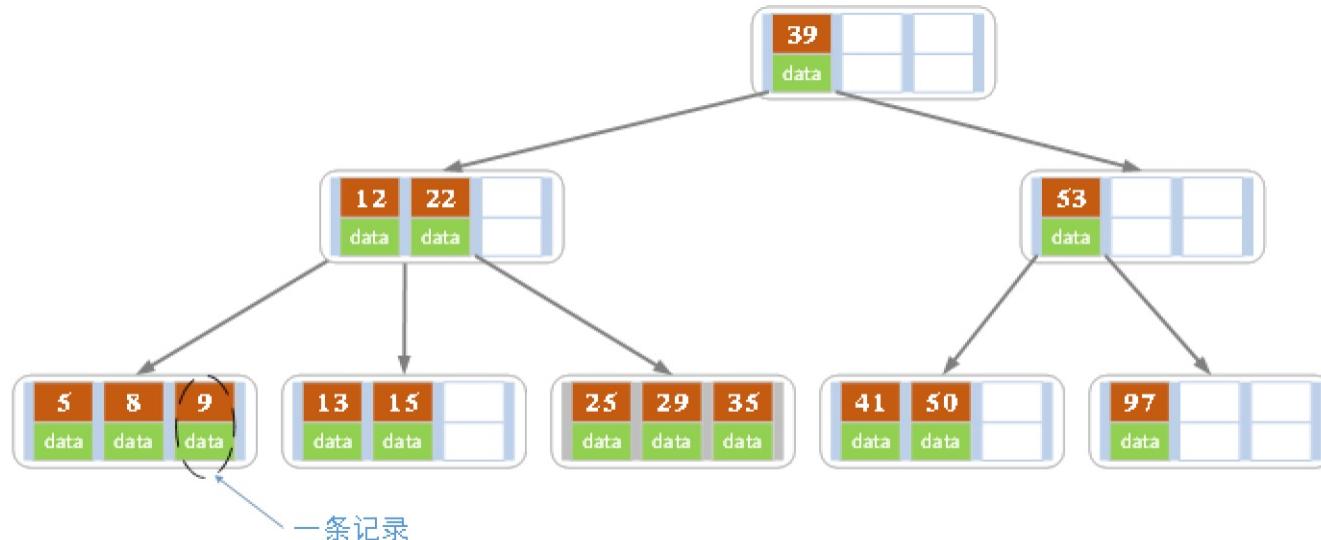
e-mail: chen-hp@sjtu.edu.cn

- **Contents**
 - KV数据结构
 - LSTM树与SSTable
 - 日志结构数据库
 - LevelDB & RocksDB
 - HTAP混合存储
- **Objectives**
 - 能够根据数据特性和数据访问模式，识别适合日志数据库存储的数据，设计并实现其在日志数据库中的存储和访问方案

- KV存储包含以下5个基本操作：
 - get(K) 查找key K对应的value
 - put(K,V) 插入键值对 (K, V)
 - update(K,V) 查找key K对应的value, 将其更新为V
 - delete(K) 删除key K对应的条目
 - scan(K1,K2) 得到从K1到K2范围内的所有key和value
- 数据结构
 - B树及其扩展
 - LSTM树
 - SSTable

- B树也称B-树,

- 它是一颗多路平衡查找树。我们描述一颗B树时需要指定它的阶数，阶数表示了一个结点最多有多少个孩子结点，一般用字母m表示阶数
- 当m取2时，就是我们常见的二叉搜索树

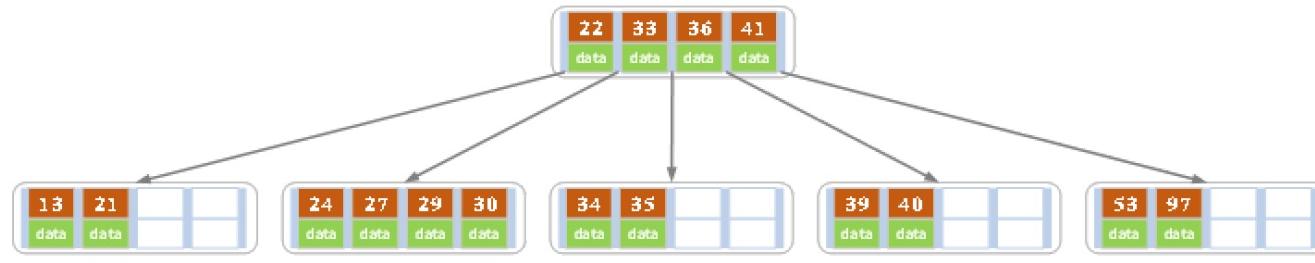


- 定义
 - 1. 每个结点最多有 $m-1$ 个key
 - 2. 根结点最少可以只有1个key
 - 3. 非根结点至少有 $\lceil m/2 \rceil - 1$ 个key (最差节点利用率为50%)
 - 4. 每个结点中的关键字都按照顺序排列, 每个关键字的左子树中的所有关键字都小于它, 而右子树中的所有关键字都大于它
 - 5. 所有叶子结点都位于同一层, 或者说根结点到每个叶子结点的长度都相同
 - 6. 设某个节点含有n个元素, 则
 - 如果是叶子节点, 该节点含有n个value
 - 如果是非叶子节点, 该节点不仅含有n个value, 还存有n+1个子节点的地址 (边)

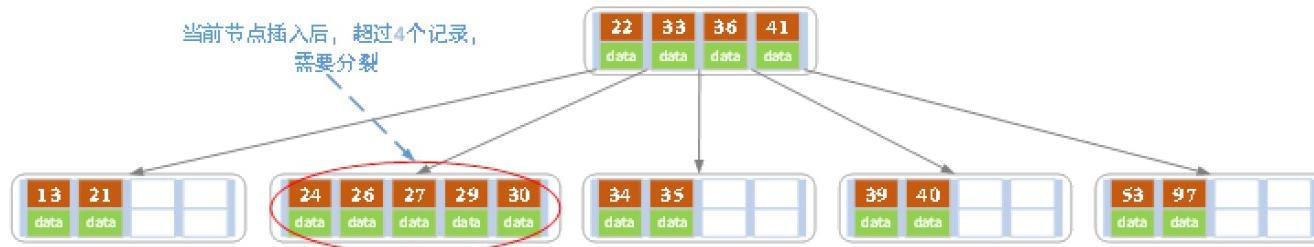
- **查找**
 - 即根据一个key, 得到对应的value(data), 也可能找不到
- **过程如下 :**
 - 1. 从根节点开始
 - 2. 在当前节点内通过二分查找定位到一个最小且 \geq key的key
 - 3. 如果key已经命中 (相等) , 则返回该key的值。如果没有命中, 则加载新的子节点, 返回第二步继续查找 (如果已经是叶子节点, 就可以返回不存在) 。
- **复杂度**
 - 整个过程时间复杂度是 $O(\log M \log N)$,
 - M是阶数, $\log M$ 对应在节点内做二分查找
 - N是元素个数, $\log N$ 对应加载节点的次数, 即树的深度

- **更新**
 - 更新就是先查找，然后修改value
 - 不会修改树的结构，但是可能会导致节点地址改动
 - 比如当将一个很小的value替换成一个很大的value，原始节点的物理空间不够，就需要重新将这个节点写入其他地方
- **插入**
 - 插入就是查找定位到节点后，在指定位置插入
 - 比较复杂的部分是当节点存在节点满了之后就需要分裂
 - 最开始插入会先插入到叶子节点，
 - 若满则以中值为划分分裂成两个，同时将中值加入到父节点中
 - 如果父节点又满，则进行相同操作，直到分裂向上的节点中没有满为止

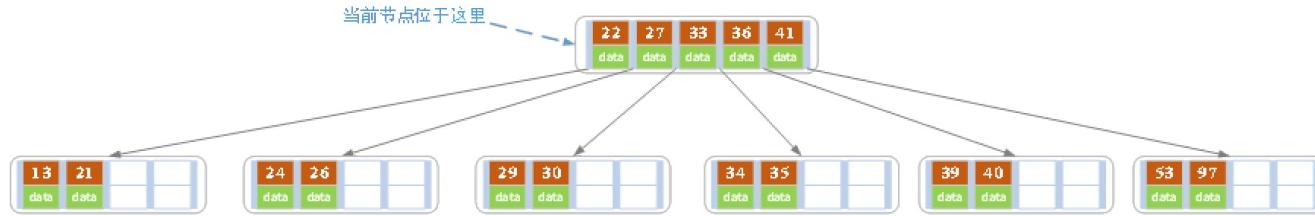
- 1. 初始：存在一棵M为5的树,即最大元素个数是4。



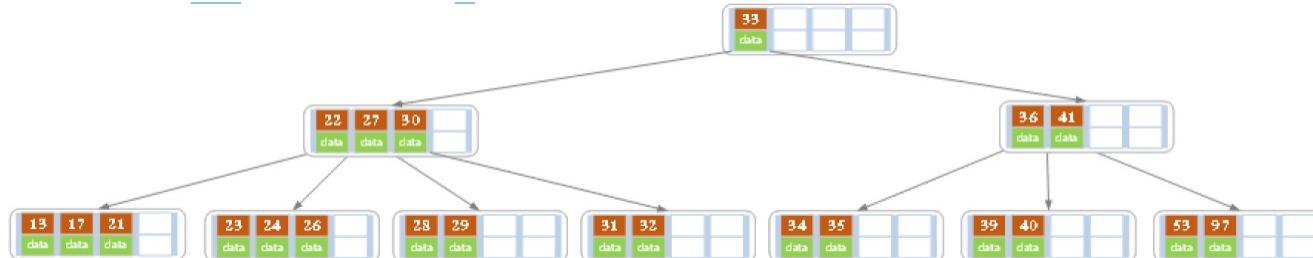
- 2. 插入成功，但是导致一个叶子节点满



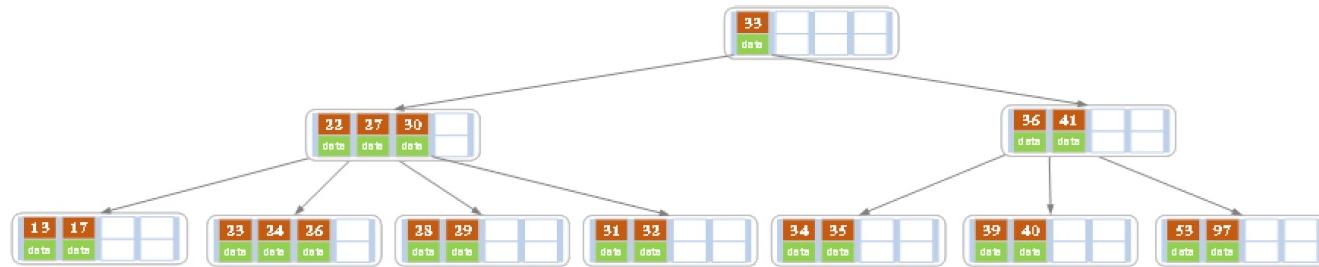
- 3. 按中值(27)分裂，并将中值插入到父亲节点



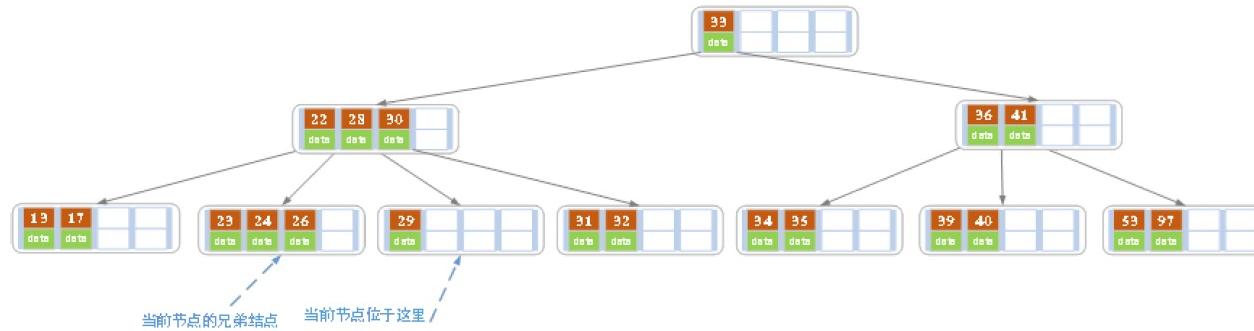
- 删除过程相对复杂
 - 但是其基本原则是，当节点删除一个元素时，优先从子节点找后继替代自己，如果做不到，就从兄弟节点找
 - 完成一个兄弟到父亲，父亲到自己的转移
 - 若兄弟也做不到，则和兄弟进行合并
 - 因为此时两个字节点个数都是 $[m/2] - 1$
- 1. 初始，阶数为5，即一个节点最少2个元素



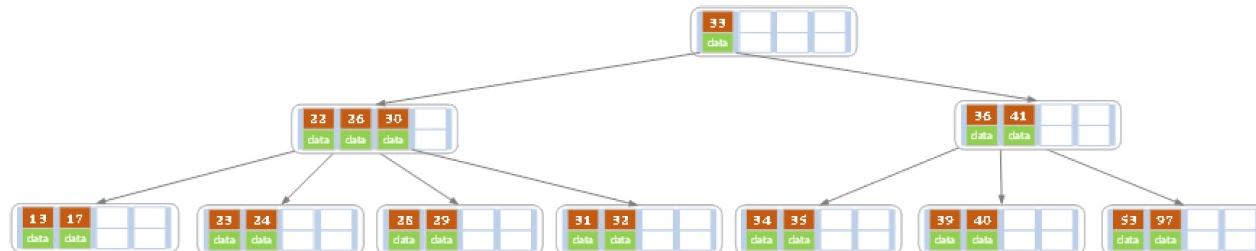
- 2. 删除21，正常删除，因为删除后节点个数仍然>2



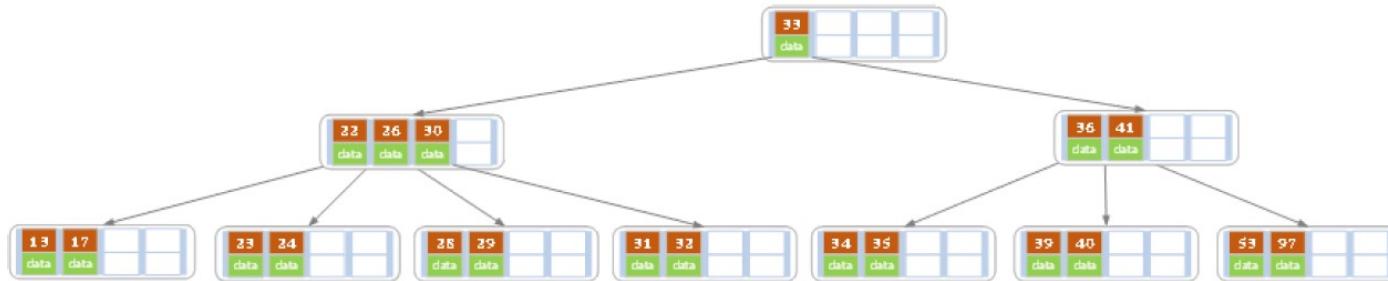
- 3. 删除27，该节点是非叶子节点，因此用子节点替换，
 - 1) 这里默认优先从右子节点替换



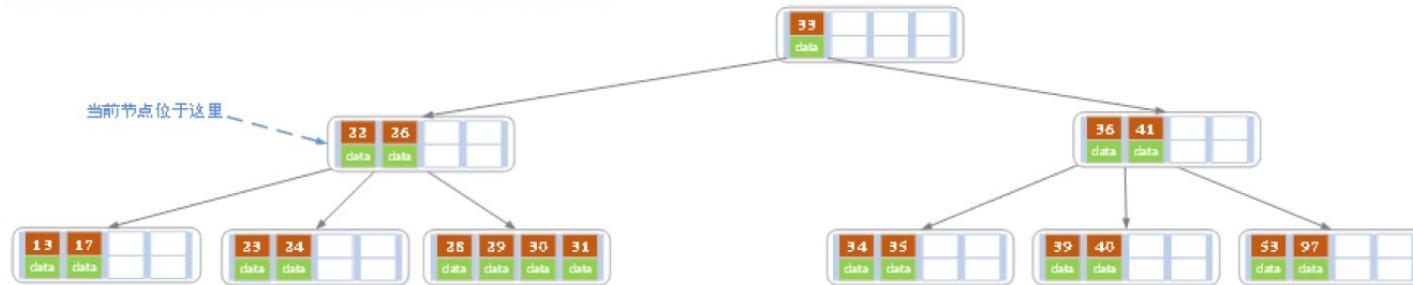
- 3. 删除27，该节点是非叶子节点，因此用子节点替换，
 - 2) 但是子节点不够，就需要从兄弟节点移动过来



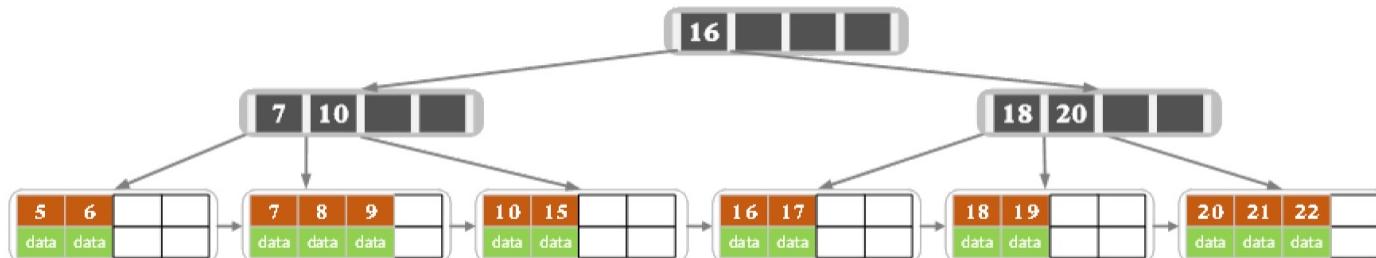
- 4. 删除32
 - 叶子节点个数不够，但是兄弟节点也无法转移



- 4. 删除32
 - 合并两个兄弟节点元素和兄弟之间的元素

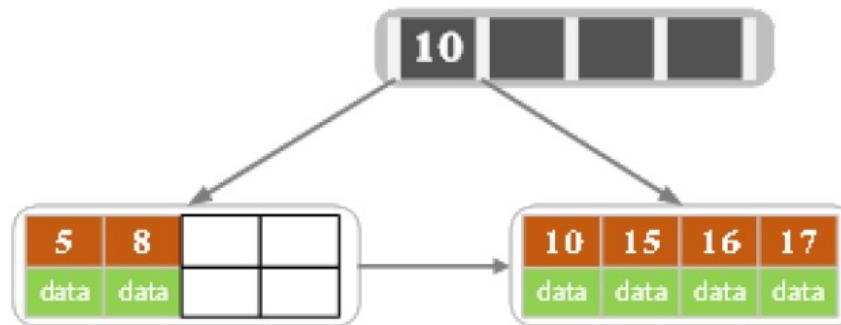


- 相比于b树，区别在于
 - 1. 非叶子节点是索引节点，只存储部分key和子节点地址，不存储value
 - 2. 所有key和value都会存储在叶子节点中
 - 3. 叶子节点之间存在顺序指向
- 查找和更新和B树类似
- 插入和删除思想一致，但是略有差别。

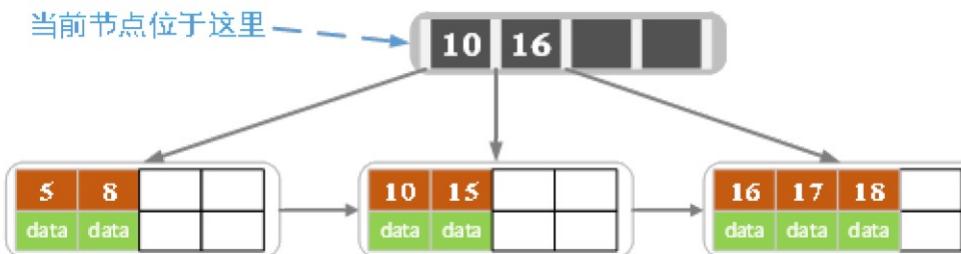
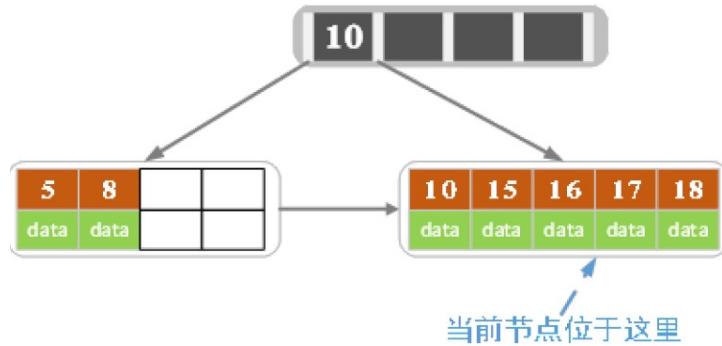


- 插入

- 插入思想不变。即数据插入到叶子节点，如果满，则分裂，将中值插入到父亲节点，直至某个节点不满。

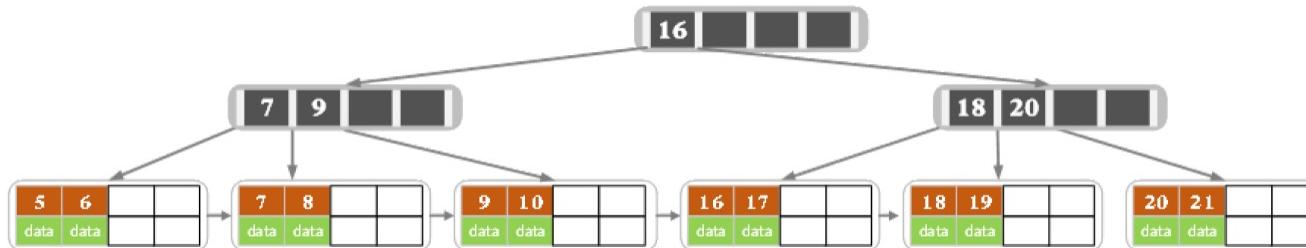


- 插入18，叶子节点满，所以分裂，中值插入到父亲节点

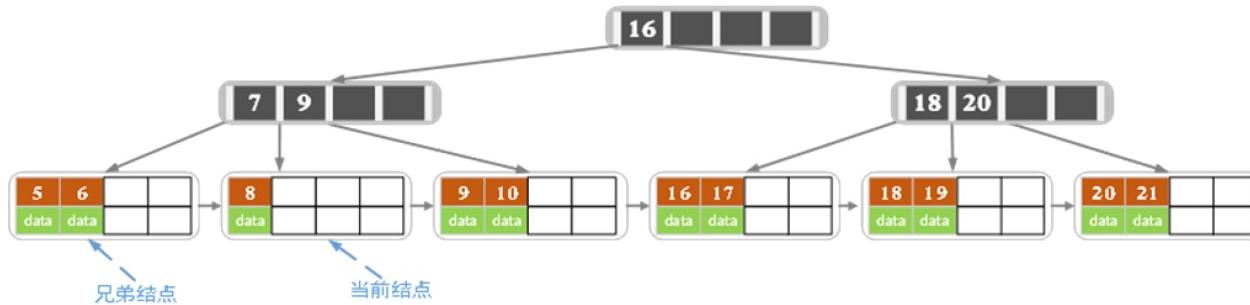


- 删除

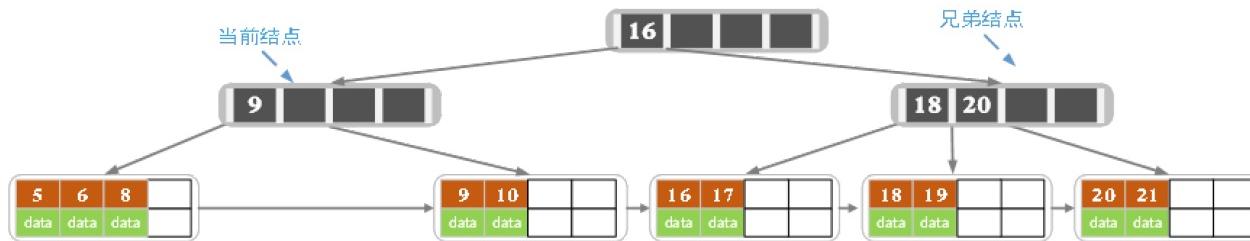
- 必须删除叶子节点上的元素，同时更新索引节点上的
- 由于父亲节点不存在记录，所以都是从兄弟之间转移
- 当无法转移的时候合并节点



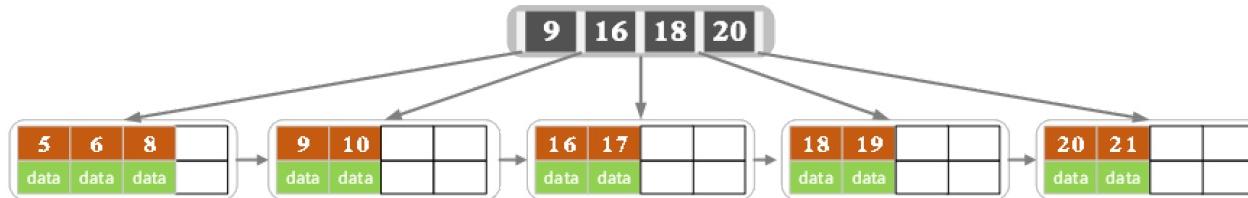
- 删除 7
 - 删除后发现数量不足，但是兄弟节点也无法转移



- 删除 7
 - 叶子合并



- 删除 7
 - 索引节点合并



- 优点
 - 快速查找
 - B+：
 - 对scan (range query, 范围扫描顺序访问) 更加友好
 - 索引节点内单个页存储元素个数可以更多，降低树的深度，尤其适合value较大的场景
 - B：查找更加快（不需要每次访问到子节点）
 - value热更新开销小
- 缺点
 - 插入操作慢
 - 空间放大率高（空间利用率不高），数据结构变更容易产生文件空洞
 - 不同节点随机存储在磁盘上，会产生大量的随机IO

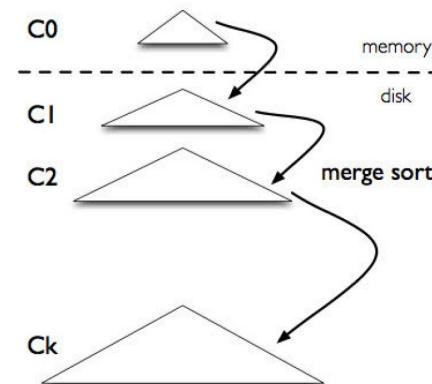
- **LSM-Tree : Log-Structured Merge Tree**

- 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批量顺序写性能远高于随机写性能的特点

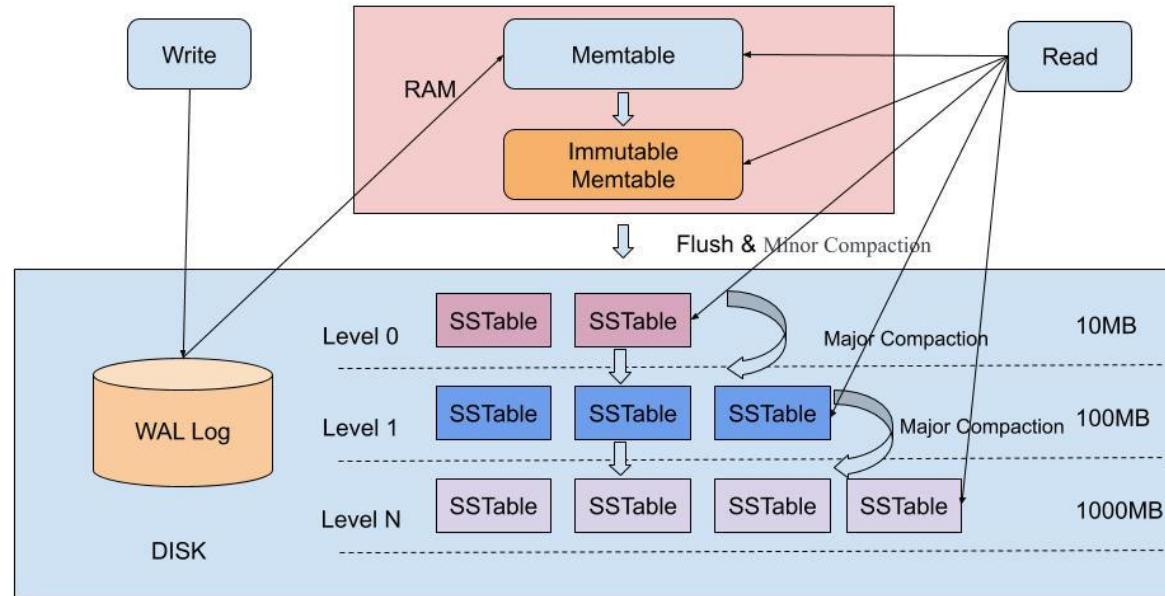
- **Log需要提高写入性能**

- Log以Append的模式追加写入，不存在删除和修改
 - 这种结构虽然大大提升了数据的写入能力，却是以牺牲部分读取性能为代价，故此这种结构通常适合于写多读少的场景

- C0 树（常驻内存）
 - C1-N 树(位于磁盘)



Log Structured Merge Trees



- 优点
 - 大幅度提高插入（修改、删除）性能
 - 空间放大率降低
 - 访问新数据更快，适合时序、实时存储
 - 数据热度分布和level相关
- 缺点
 - 牺牲了读性能（一次可能访问多个层）
 - 读、写放大率提升

- SSTable : Sorted String Table

- SSTable的特点

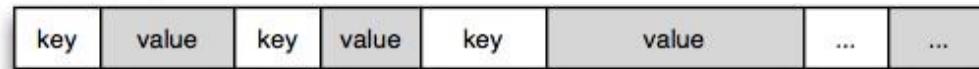
- 存储的是<键,值>格式的字节数据
 - 字节数据的长度随意，没有限制
 - 键可以重复，键值对不需要对齐
 - 随机读取操作非常高效

- SSTable的限制

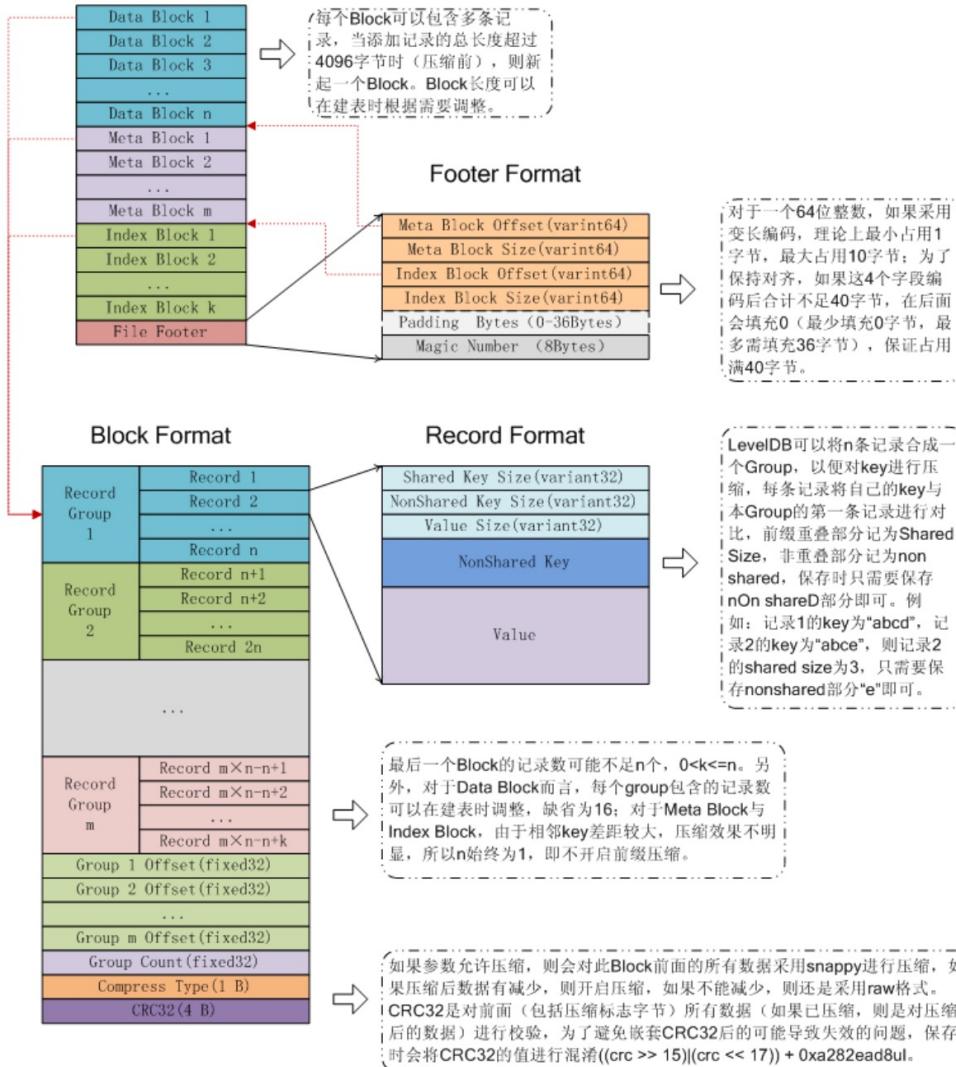
- 一旦SSTable写入硬盘后，就是不可变的，因为插入或者删除需要对SSTable文件进行大量的I/O操作
 - 不适合随机读取和写入，因为效率很低，原因同上一条

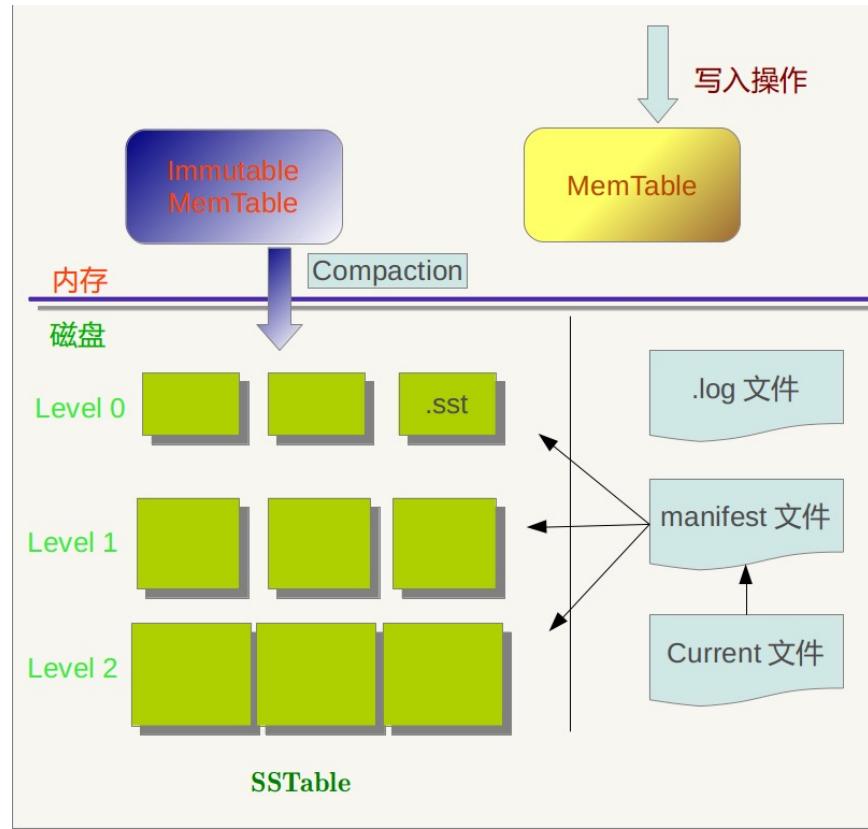
Index	
key	offset
key	offset
...	...

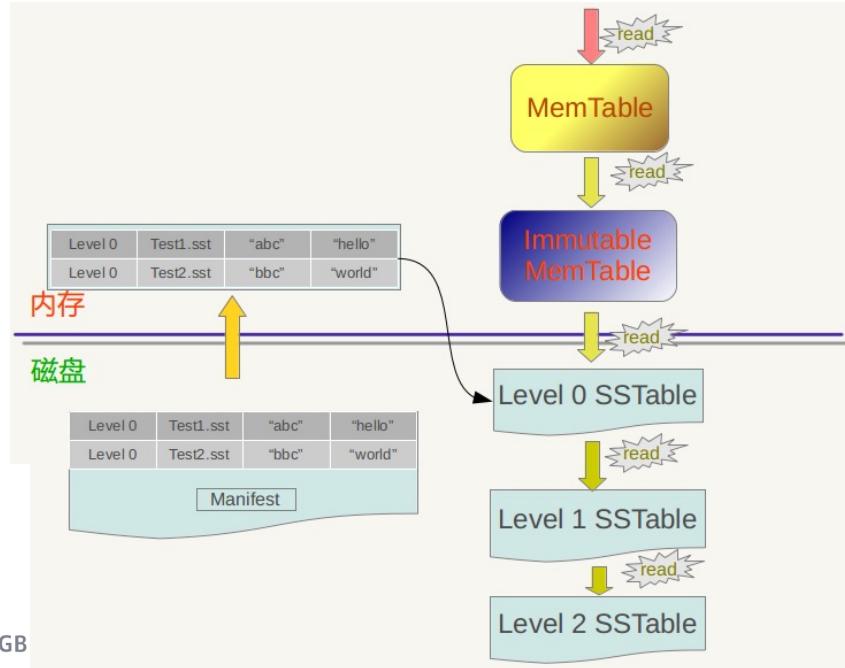
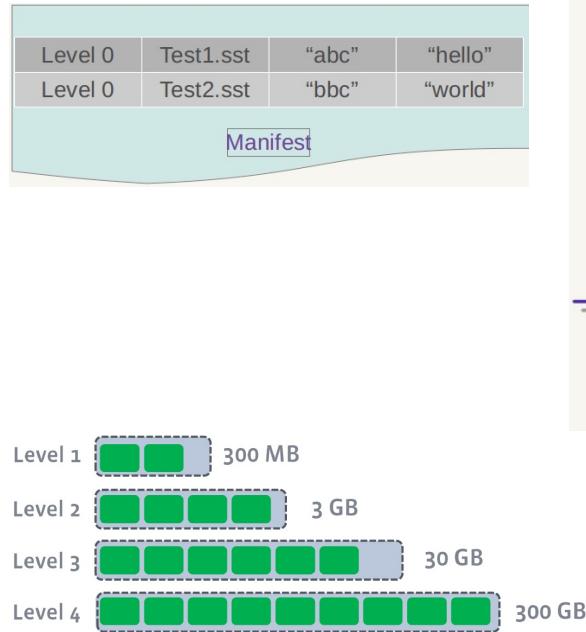
SSTable file



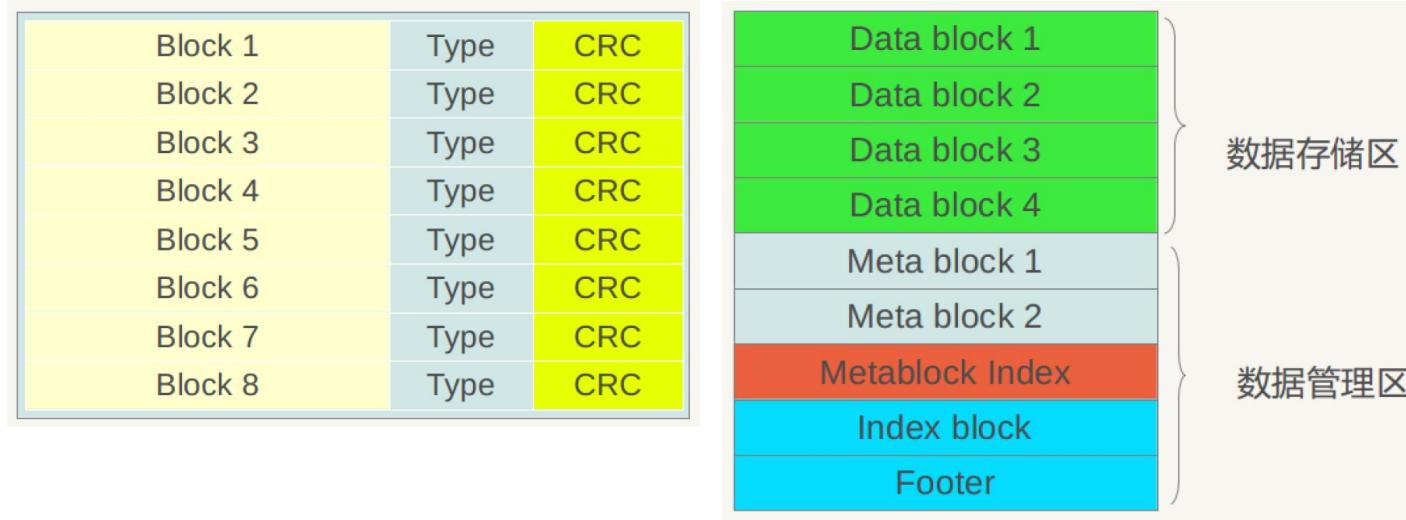
SSTable



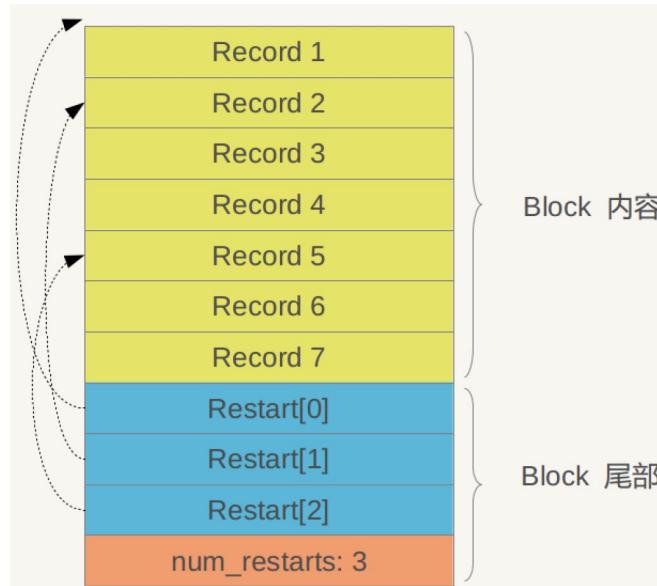




- SSTable

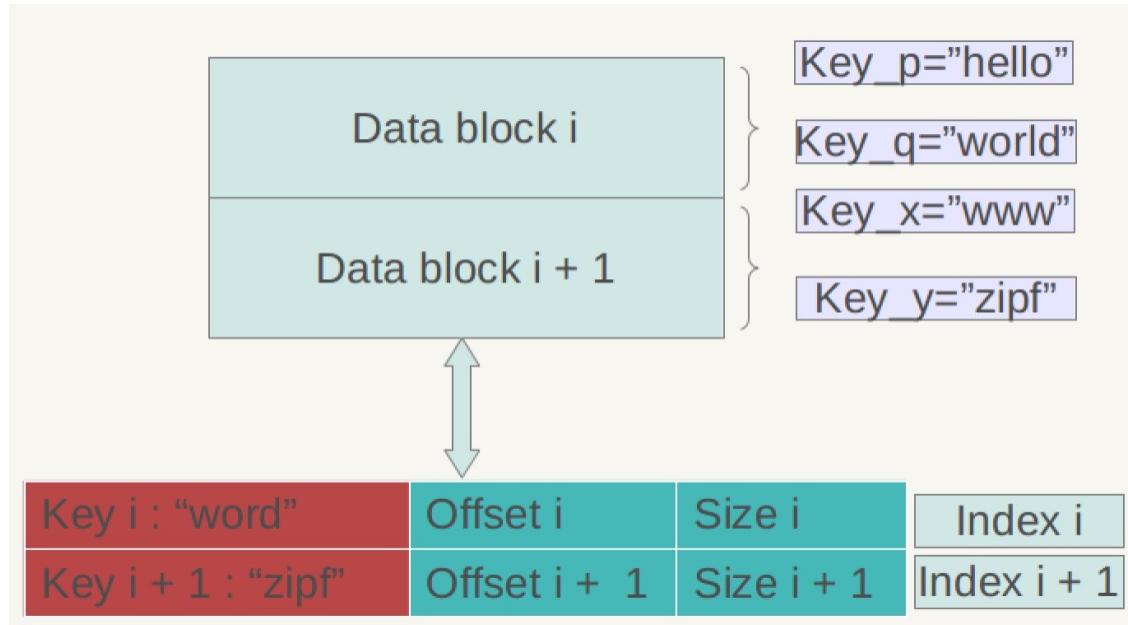


- SSTable 数据存储区

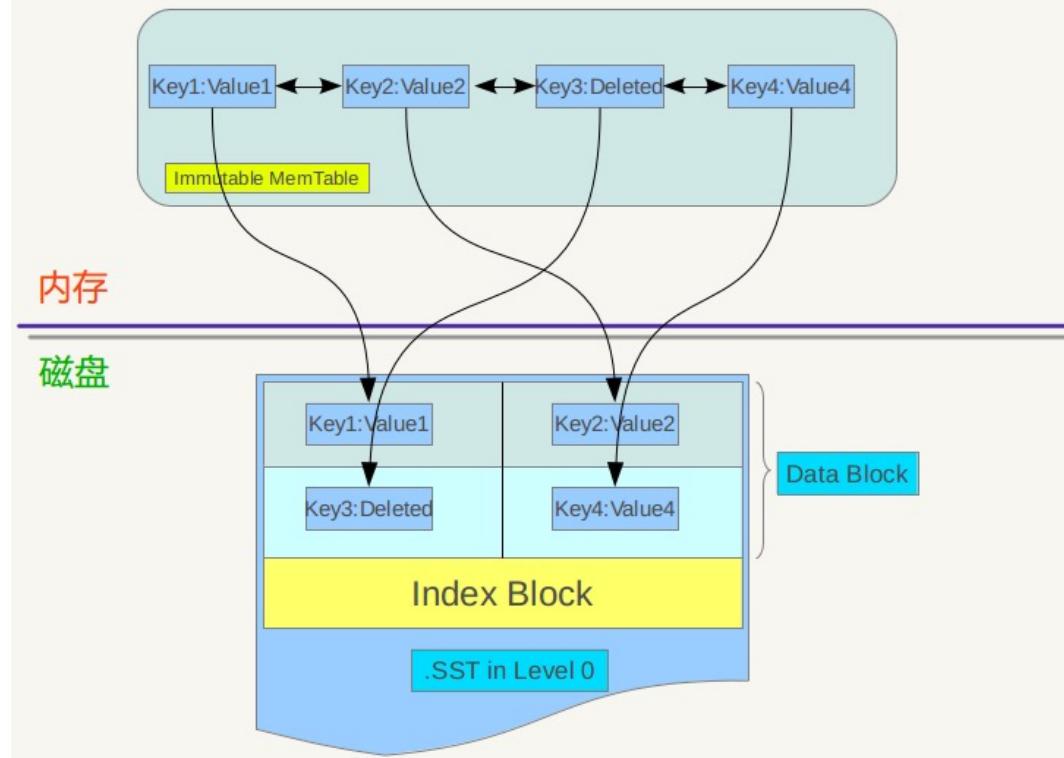


Record i	key共享长度	key非共享长度	value长度	key非共享内容	value内容
Record i+1	key共享长度	key非共享长度	value长度	key非共享内容	value内容

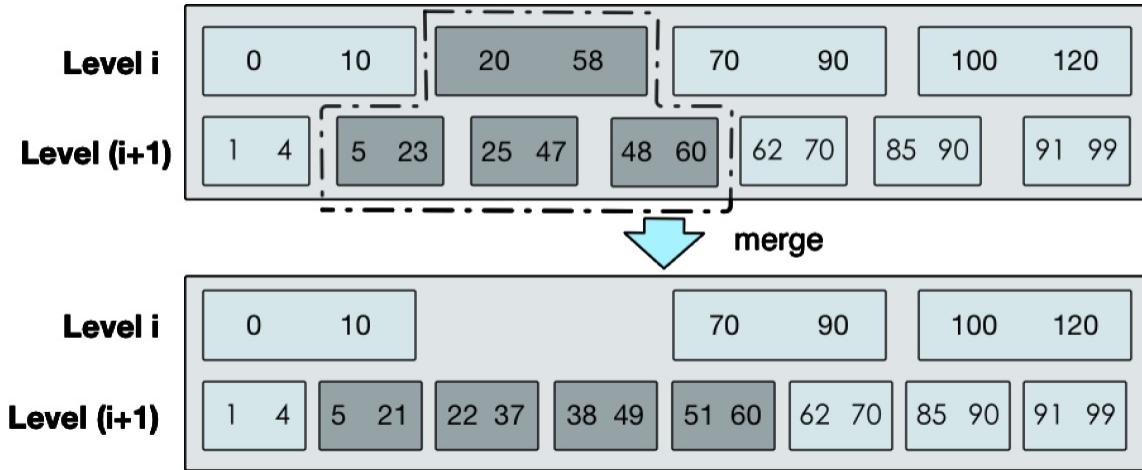
- SSTable 数据管理区



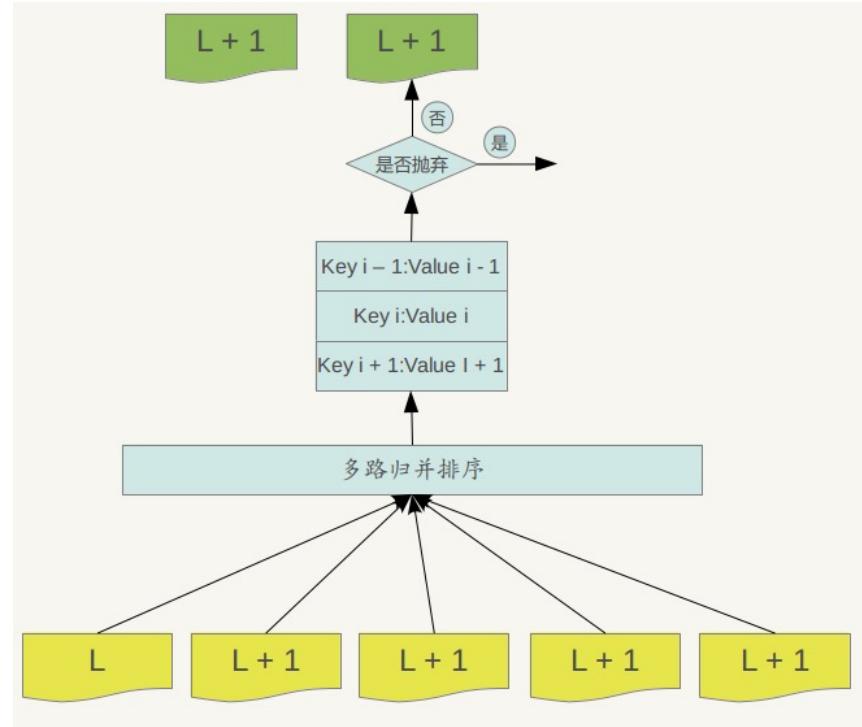
- Compaction



- Compaction



- Compaction



- RocksDB is an embeddable persistent key-value store for fast storage
 - RocksDB uses a **log structured database engine**, written entirely in C++, for maximum performance. Keys and values are just arbitrarily-sized byte streams.
 - RocksDB is developed and maintained by **Facebook** Database Engineering Team. It is built on earlier work on **LevelDB** by Sanjay Ghemawat (sanjay@google.com) and Jeff Dean (jeff@google.com)
 - <https://rocksdb.org/>
 - <https://rocksdb.org.cn/doc.html>
 - <https://github.com/facebook/rocksdb/>
 - <https://github.com/facebook/rocksdb/wiki/RocksDB-Basics>

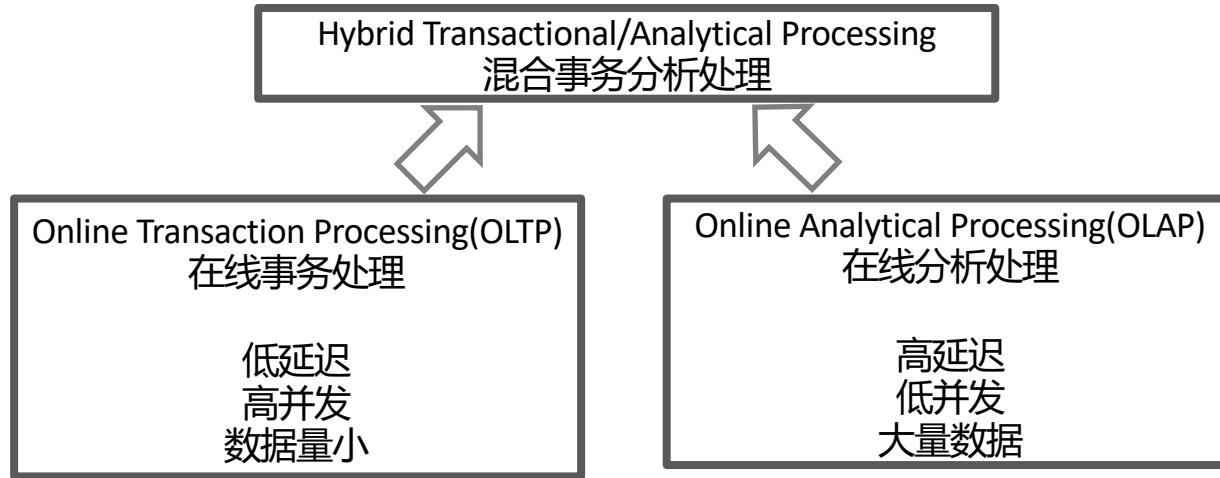


```
1 // Copyright (c) 2011-present, Facebook, Inc. All rights reserved.
2 // This source code is licensed under both the GPLv2 (found in the
3 // COPYING file in the root directory) and Apache 2.0 License
4 // (found in the LICENSE.Apache file in the root directory).
5
6 #include <cstdio>
7 #include <string>
8
9 #include "rocksdb/db.h"
10 #include "rocksdb/slice.h"
11 #include "rocksdb/options.h"
12
13 using namespace rocksdb;
14
15 std::string kDBPath = "/tmp/rocksdb_simple_example";
16
17 int main() {
18     DB* db;
19     Options options;
20     // Optimize RocksDB. This is the easiest way to get RocksDB to perform well
21     options.IncreaseParallelism();
22     options.OptimizeLevelStyleCompaction();
23     // create the DB if it's not already present
24     options.create_if_missing = true;
25
26     // open DB
27     Status s = DB::Open(options, kDBPath, &db);
28     assert(s.ok());
29 }
```

```
30    // Put key-value
31    s = db->Put(WriteOptions(), "key1", "value");
32    assert(s.ok());
33    std::string value;
34    // get value
35    s = db->Get(ReadOptions(), "key1", &value);
36    assert(s.ok());
37    assert(value == "value");
38
39    // atomically apply a set of updates
40    {
41        WriteBatch batch;
42        batch.Delete("key1");
43        batch.Put("key2", value);
44        s = db->Write(WriteOptions(), &batch);
45    }
46
47    s = db->Get(ReadOptions(), "key1", &value);
48    assert(s.IsNotFound());
49
50    db->Get(ReadOptions(), "key2", &value);
51    assert(value == "value");
```

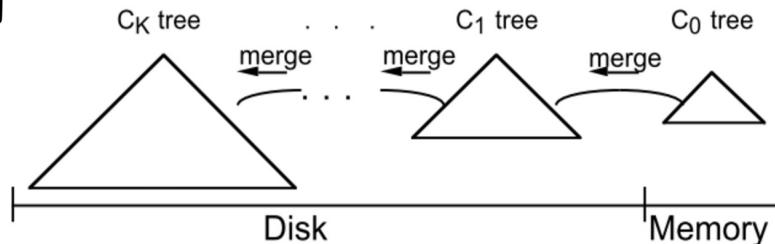
RocksDB Sample

```
53  {
54      PinnableSlice pinnable_val;
55      db->Get(ReadOptions(), db->DefaultColumnFamily(), "key2", &pinnable_val);
56      assert(pinnable_val == "value");
57  }
58
59  {
60      std::string string_val;
61      // If it cannot pin the value, it copies the value to its internal buffer.
62      // The internal buffer could be set during construction.
63      PinnableSlice pinnable_val(&string_val);
64      db->Get(ReadOptions(), db->DefaultColumnFamily(), "key2", &pinnable_val);
65      assert(pinnable_val == "value");
66      // If the value is not pinned, the internal buffer must have the value.
67      assert(pinnable_val.IsPinned() || string_val == "value");
68  }
69
70  PinnableSlice pinnable_val;
71  db->Get(ReadOptions(), db->DefaultColumnFamily(), "key1", &pinnable_val);
72  assert(s.IsNotNull());
73  // Reset PinnableSlice after each use and before each reuse
74  pinnable_val.Reset();
75  db->Get(ReadOptions(), db->DefaultColumnFamily(), "key2", &pinnable_val);
76  assert(pinnable_val == "value");
77  pinnable_val.Reset();
78  // The Slice pointed by pinnable_val is not valid after this point
79
80  delete db;
81
82  return 0;
83 }
```



LSM-Tree 抽象结构

- 分层结构
- 提供写优化
- TP事务友好

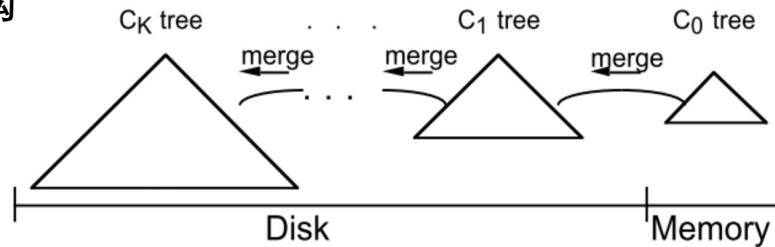


RocksDB由Facebook开发，是LSM-Tree数据结构的最成熟实现，越来越多地在主流数据库中作为新的底层存储引擎

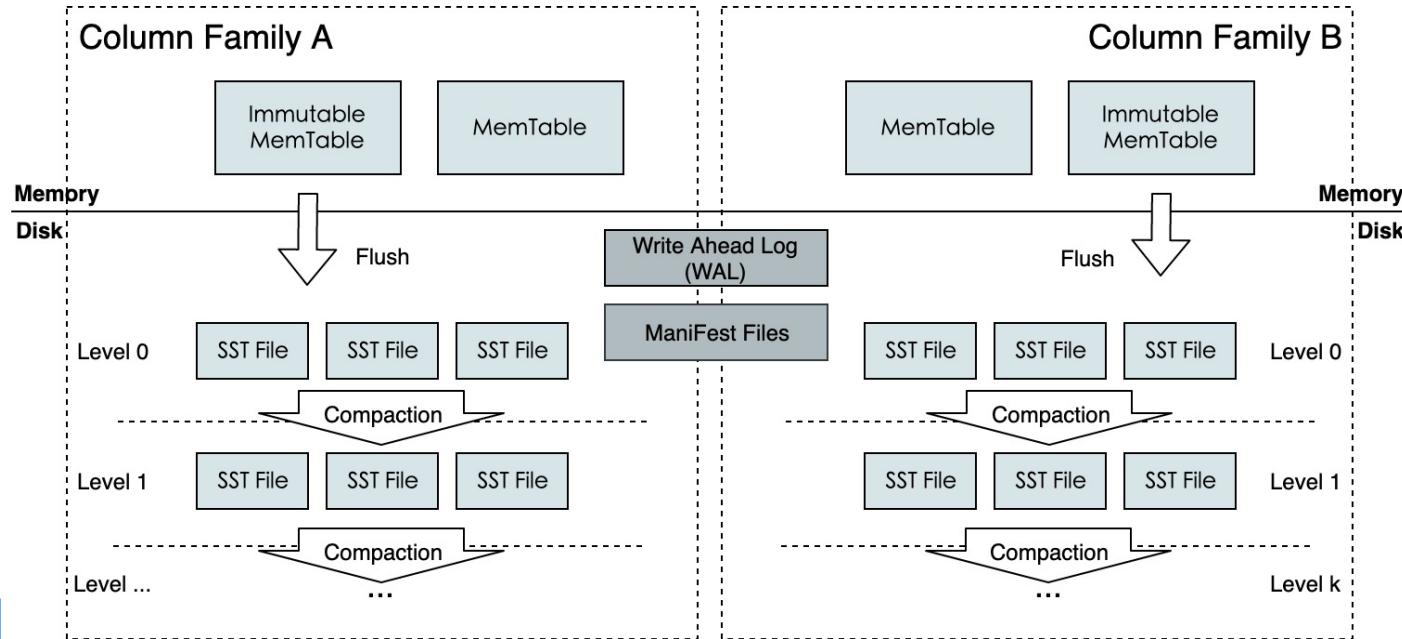


LSM-Tree 抽象结构

- 分层结构
- 提供写优化
- TP事务友好



RocksDB静态结构



LSM-TREE写阻塞问题

RocksDB写流程

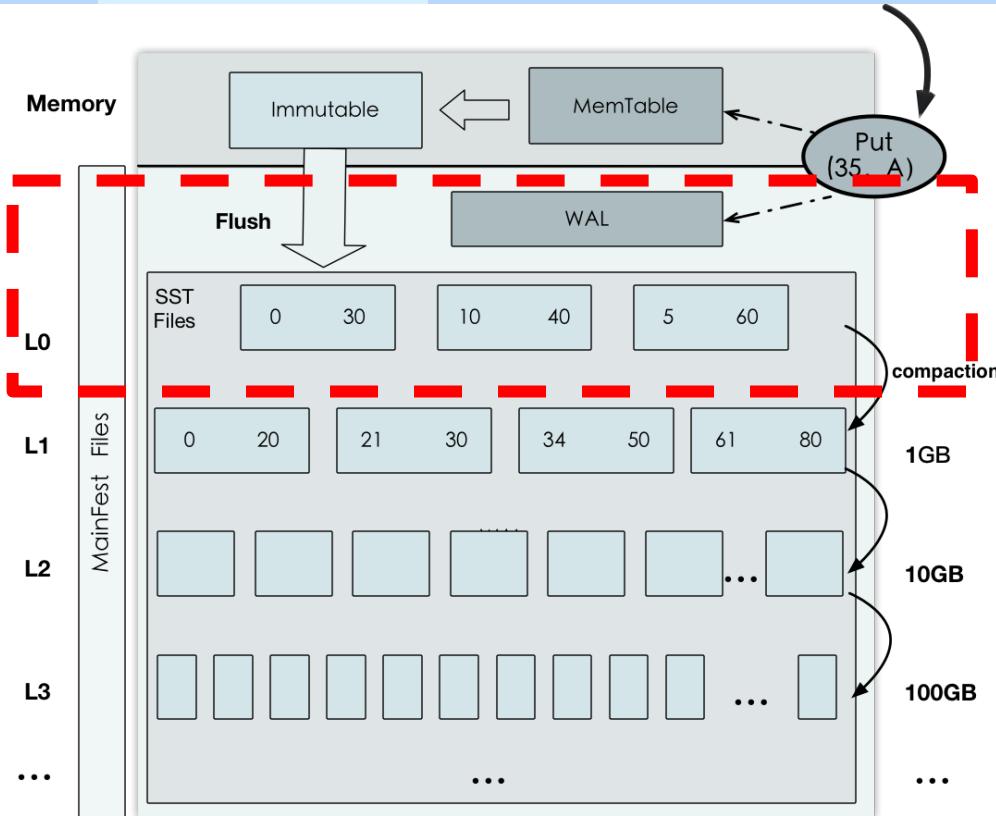
优 - 低延迟插入

- 写入内存后直接返回
- 后台异步写入磁盘

劣 - 写阻塞问题

- L0层满时将阻塞内存到磁盘的Flush过程
- L0层下沉Compaction过程无法多任务执行
- 异步写写放大严重，容易磁盘变成瓶颈

降低了TP事务的可用性



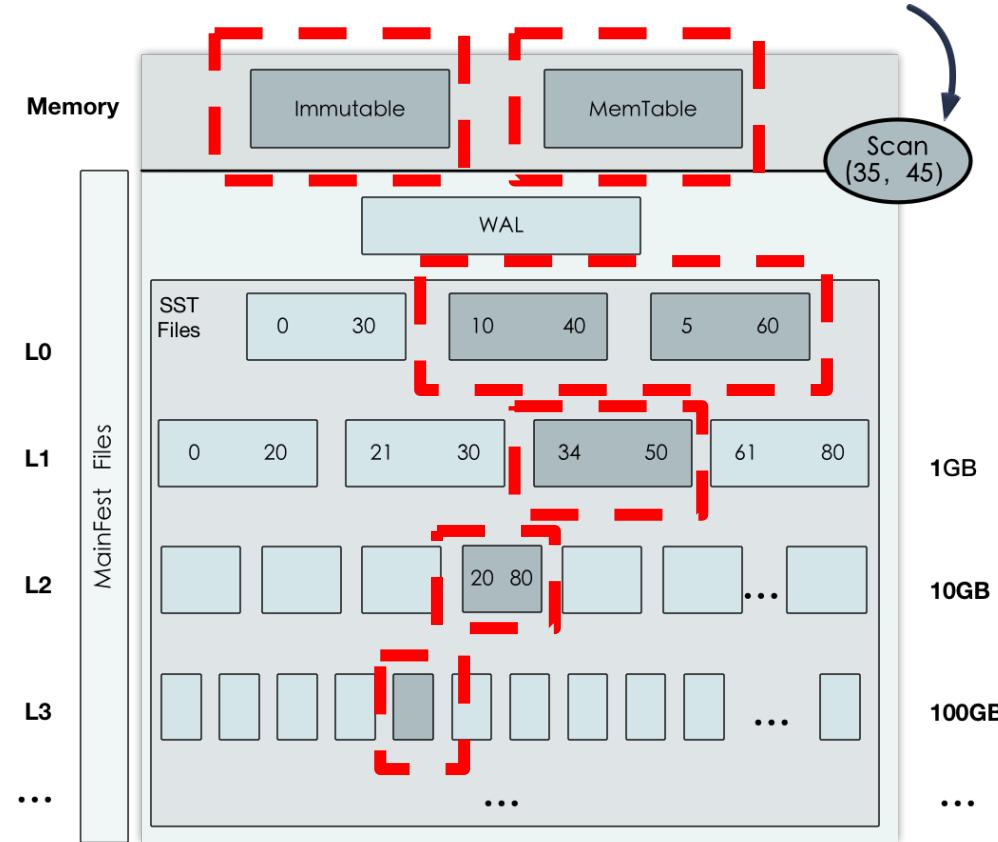
背景与问题 - LSM-TREE读放大问题

RocksDB读流程

劣 - 读放大问题

- 不同层级存储着不同版本的数据
- 需要访问所有可能的数据文件

限制了AP查询的性能



已有的行列混合存储的研究针对非LSM-Tree的数据库：

常见的列式存储格式：

- Row Group模式
- 行列折中方案

常见的行列混合存储决策算法：

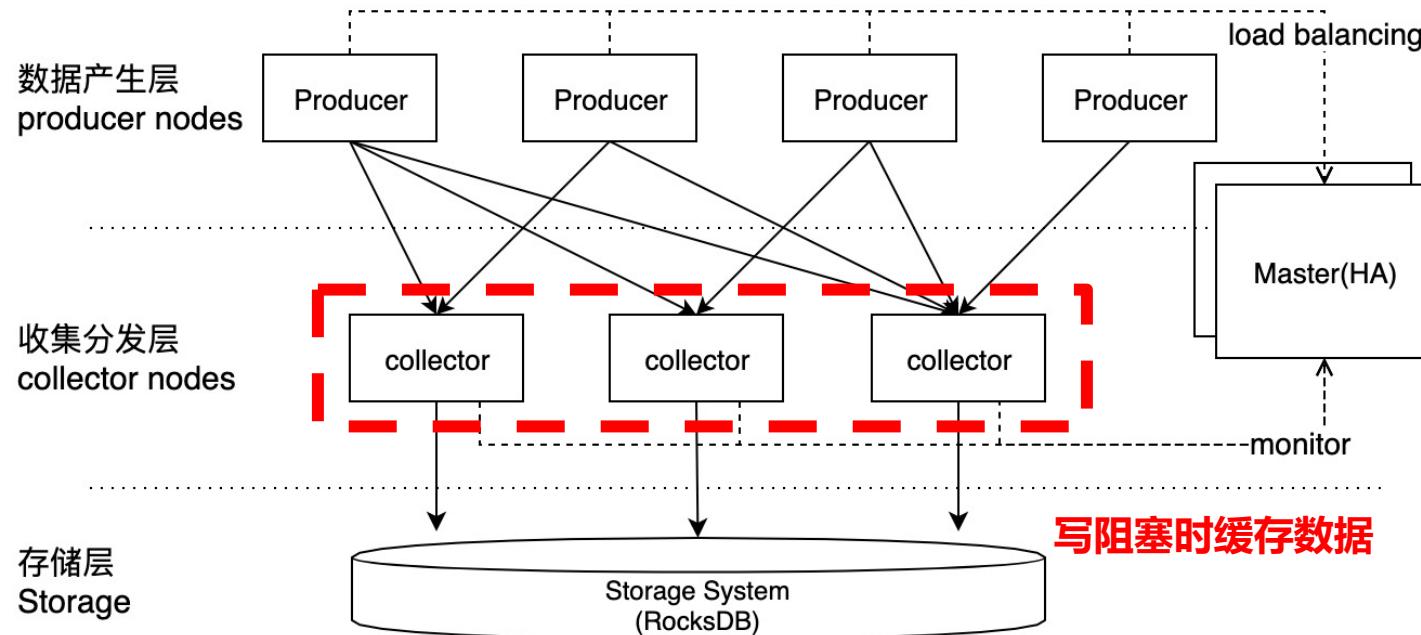
- 行列对等存储
- 行列差异存储
- 基于查询的分析

基于LSM-Tree的数据库进行行列混合存储的研究尚属热点

- 基于LSM-Tree的KV数据库优化主要针对写放大：
 - 同级内增加局部分区(PebblesDB)
 - 块大小优化
- 不同level存储策略优化
 - 混合存储结构可以参考的方面：
 - 根据冷热数据进行区分行列，如更新频率，读频率
 - 基于历史查询的分析

设计与实现 – 解决写阻塞问题

在RocksDB和数据源之间增加一个收集分发层，集群结构如下



解决写阻塞问题

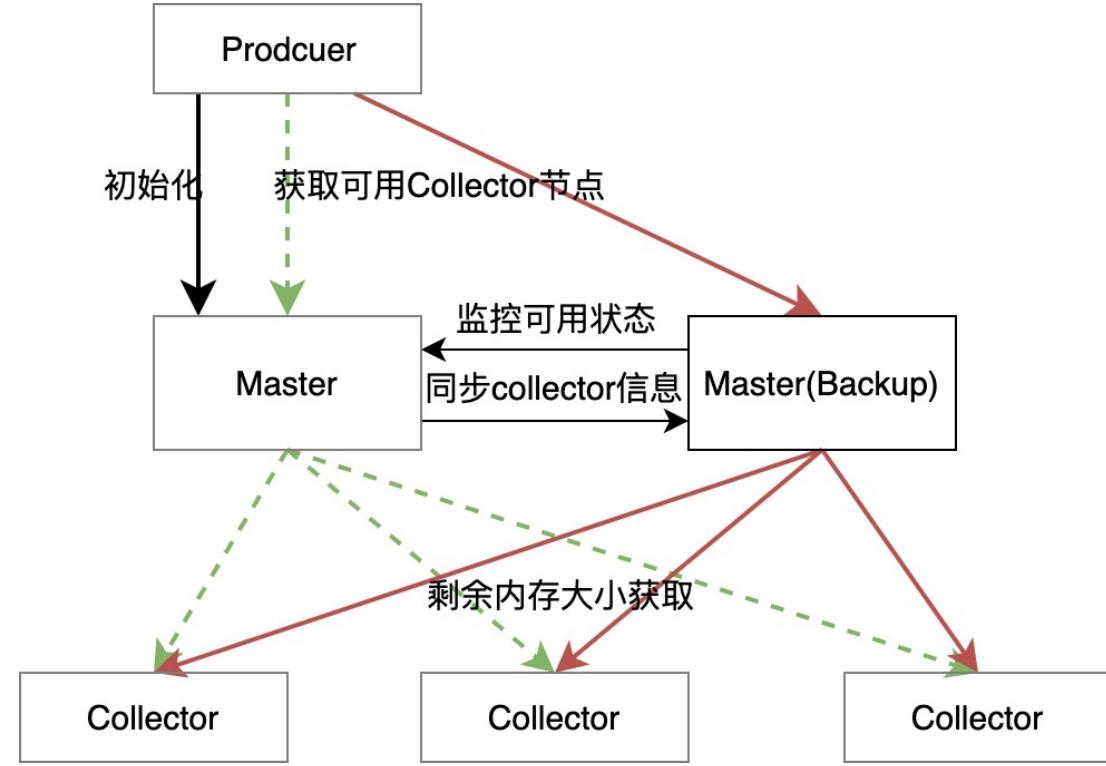
采用中心化设计：

Master采用主从备份
监控集群负载，
调控负载均衡。

负载均衡策略基于剩余
内存大小，
即分配到某个节点的概
率与剩余内存大小正比

$$P_i = \frac{FM_i}{\sum_{k=1}^n FM_k}$$

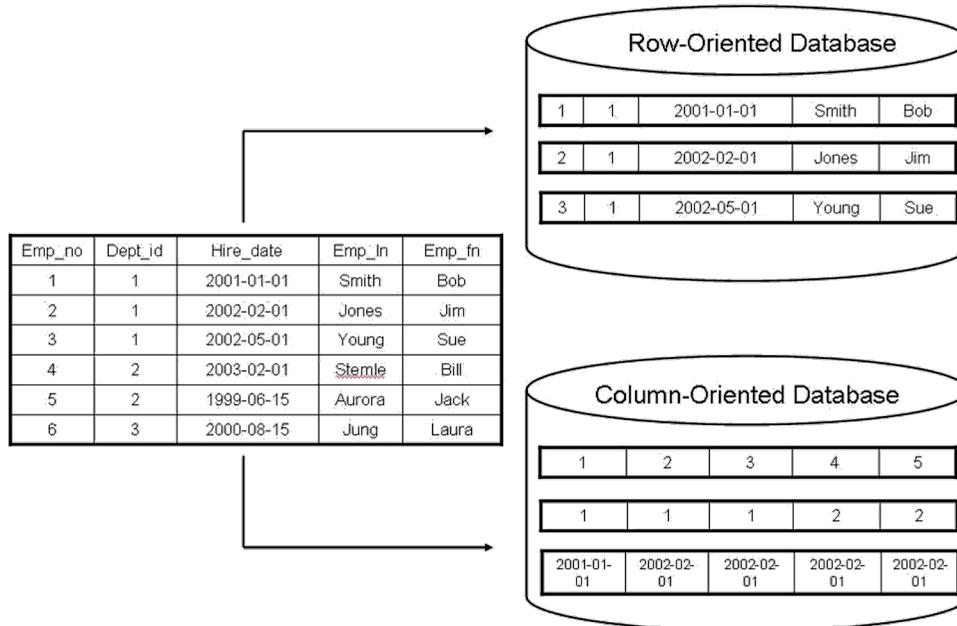
- P_i ：流量分配到节点*i*的概率
- FM_i ：节点剩余内存



解决读放大问题

在RocksDB中增加列式存储

列式存储在访问少量列时磁盘读取量更小，可以减少读放大的开销



行式存储(原)

- 适合事务处理(TP)
- 数据写入不需要拆分属性

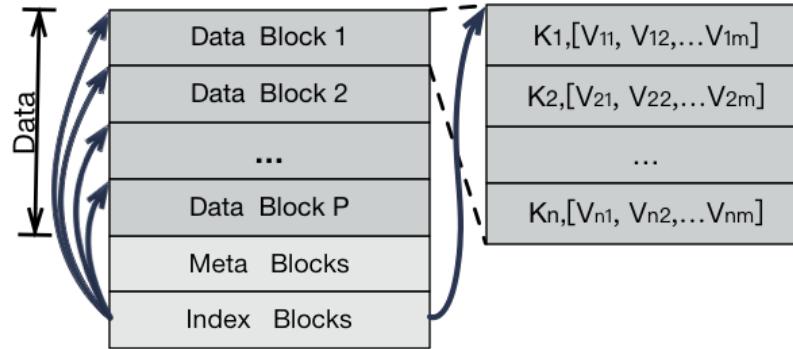
列式存储(新增)

- 适合分析处理(AP)
- 便于分类压缩
- 对内存友好

解决读放大问题

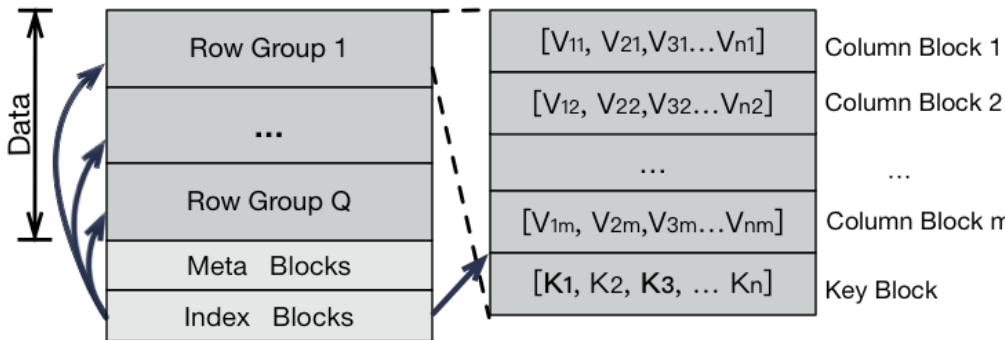
在RocksDB中增加列式存储

列式存储在访问少量列时磁盘读取量更小，可以减少读放大的开销



行式存储(原)

- 适合事务处理(TP)
- 数据写入不需要拆分属性



列式存储(新增)

- 适合分析处理(AP)
- 便于分类压缩
- 对内存友好

提出混合存储策略

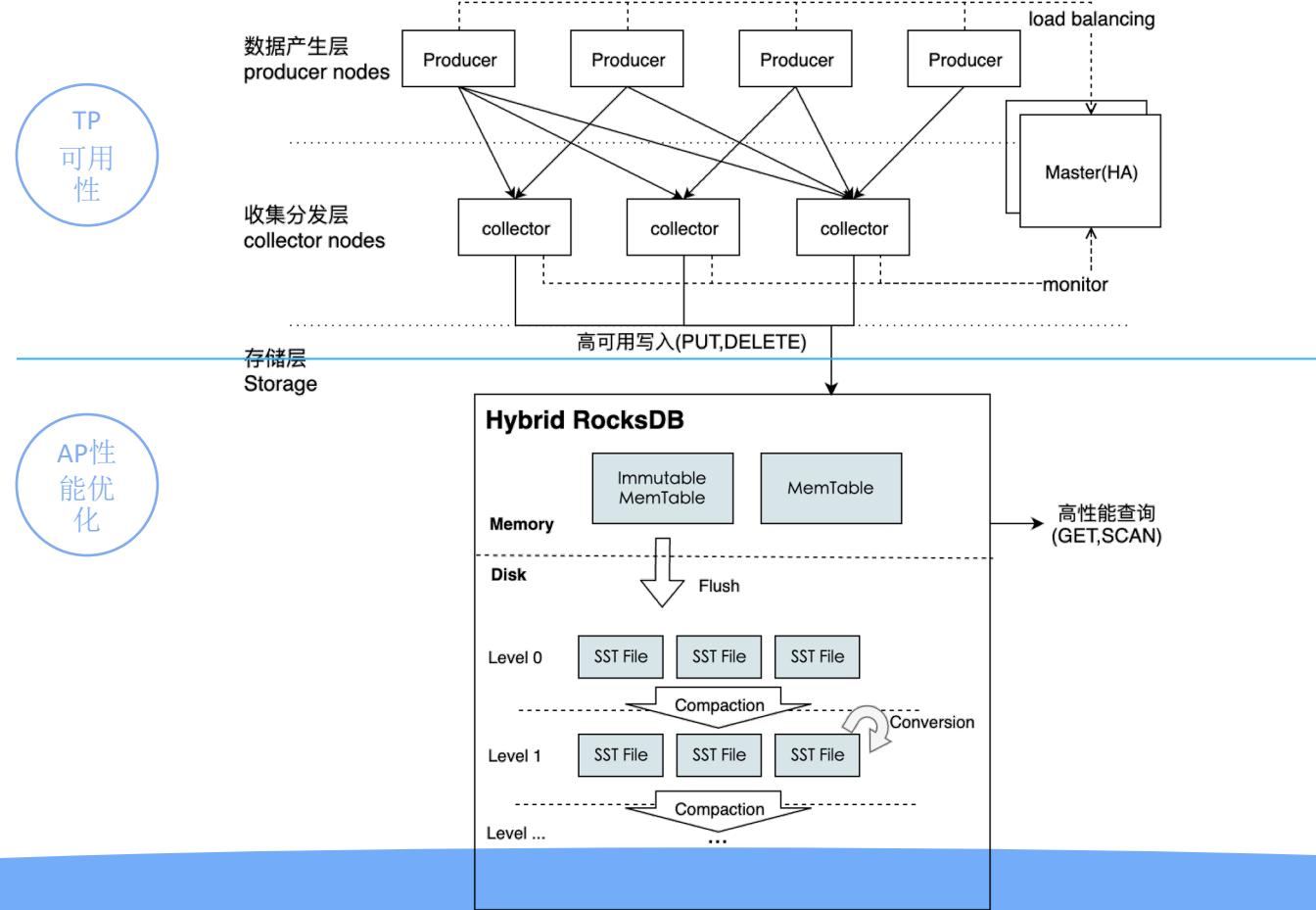
- =>常做事务的数据以行式存储,
- =>常做查询的数据以列式存储

以磁盘读写开销为格式转换的指标

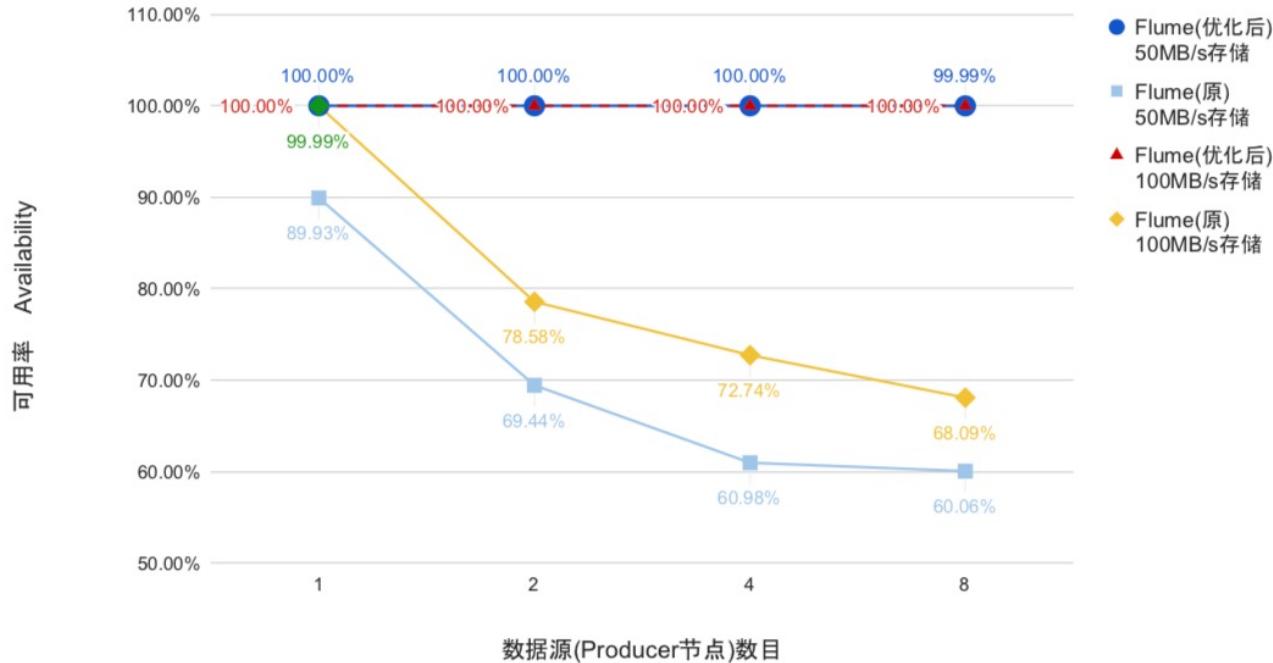
- 以文件为格式调整粒度
- 额外记录每个文件的历史操作
- 计算每个文件在行式存储和列式存储下重做历史操作时的磁盘读写次数

兼容原有的后台逻辑

- 新增主动的conversion后台过程
 - 监控所有文件，当满足【原格式读代价 > 新格式读代价 + 新格式写代价】条件，就触发原地格式转换
- 结合原有的compaction过程
 - 新文件产生时，根据祖先文件，选取读写代价总和最小的格式

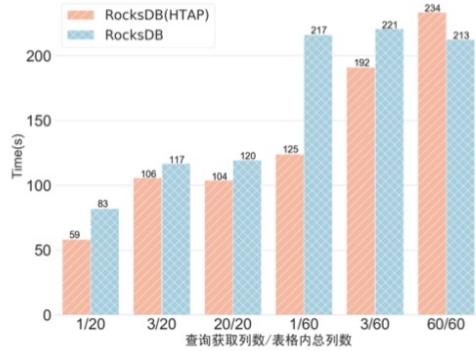


实验与验证 – 可用性提升

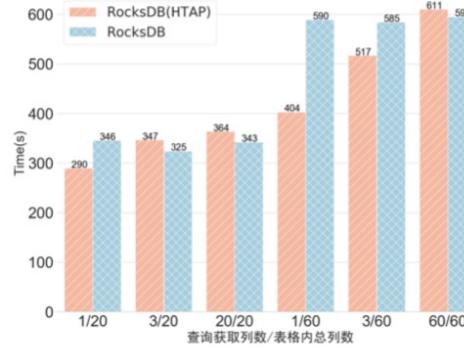


原 Flume 集群随着数据压力的增大，其可用性在不断下降。而经过优化后可用性维持在近 100%。

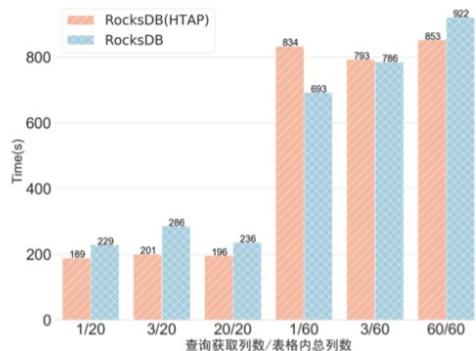
实验与验证 – HTAP负载测试



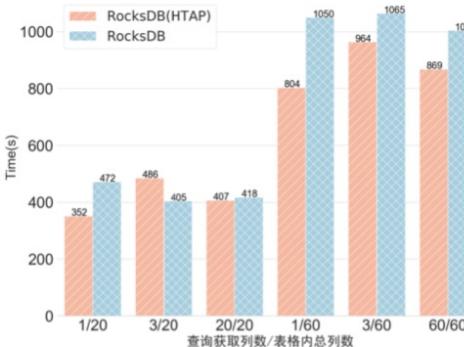
(a) 大量读, 读访问 20% 数据



(b) 大量读, 读访问 80% 数据



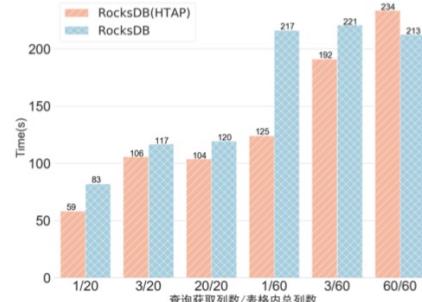
(c) 大量写, 读访问 20% 数据



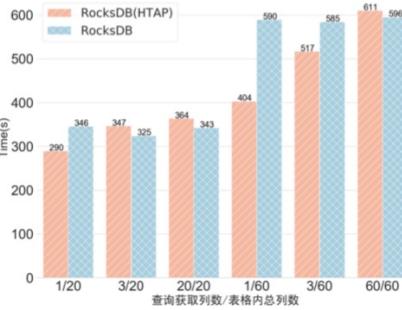
(d) 大量写, 读访问 80% 数据

实验与验证 – HTAP负载测试

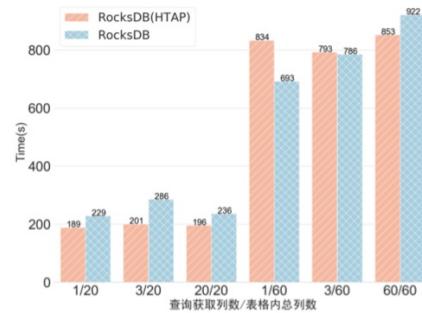
- 共24组测试中，20组混合存储RocksDB都得到了提升，列数越多提升越明显，最高可缩短50%时间



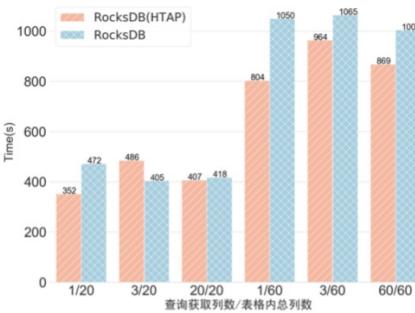
(a) 大量读，读访问 20% 数据



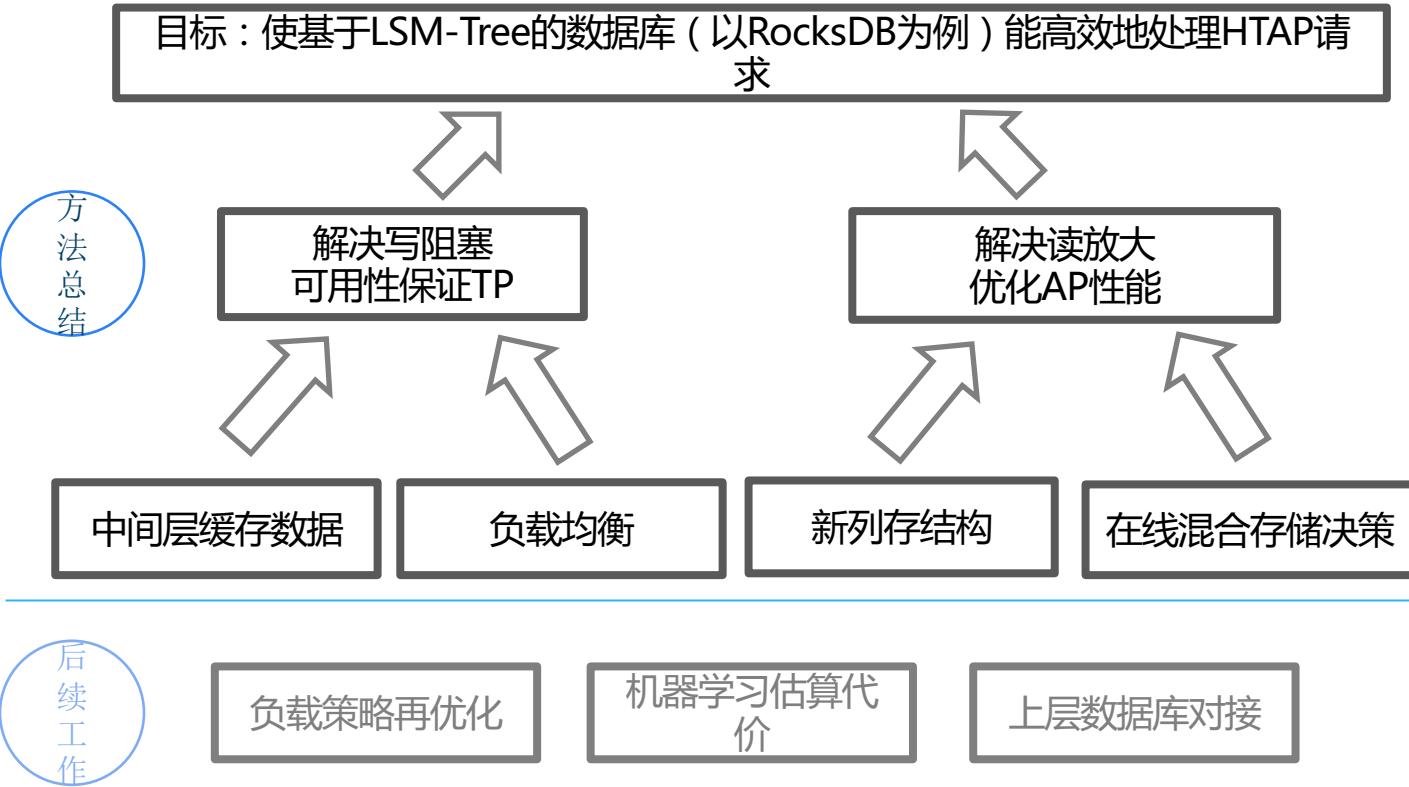
(b) 大量读，读访问 80% 数据



(c) 大量写，读访问 20% 数据



(d) 大量写，读访问 80% 数据



- 请你按照你的理解，用Word文档答复下列问题：
 1. 请阐述日志结构数据库适合什么样的应用场景？(1分)
 2. 请阐述日志结构数据库中的读放大和写放大分别是什么意思？(1分)
 3. 日志结构合并树中，WAL的作用是什么？(1分)
 - 请提交包含上述问题答案的文档
- 评分标准：
 - 上述问题答案不唯一，只要你的说理合理即可视为正确。

- B树和B+树的插入、删除图文详解
 - <https://www.cnblogs.com/nullzx/p/8729425.html>
- 日志结构的合并树 The Log-Structured Merge-Tree
 - <https://www.cnblogs.com/siegfang/archive/2013/01/12/lsm-tree.html>
- LSM-tree 基本原理及应用
 - <https://cloud.tencent.com/developer/news/340271>
- Storage engine design(Part2)
 - <https://akumuli.org/akumuli/2017/08/01/storage-engine-design2/>
- An Introduction to B ε trees and Write-Optimization
 - <https://www3.cs.stonybrook.edu/~rob/papers/login15.pdf>
- 深入理解什么是LSM-Tree
 - <https://blog.csdn.net/u010454030/article/details/90414063>

- B+ Tree、LSM、Fractal tree index 读写放大分析
 - http://kernelmaker.github.io/Btree_LSM_FTI
- Data structures for external memory
 - <http://blog.omega-prime.co.uk/2016/07/05/datastructures-for-external-memory/>
- IndexFS: Scaling File System Metadata Performance
 - <https://www.open-open.com/lib/view/open1420338434078.html>
- TokuDB的索引结构--分形树的实现
 - <http://openinx.github.io/2015/11/25/ft-index-implement/>
- TokuDB · 引擎特性 · HybridDB for MySQL高压缩引擎TokuDB 揭秘
 - <http://mysql.taobao.org/monthly/2017/07/04/>
- MySQL 高性能存储引擎 : TokuDB初探
 - <https://www.biaodianfu.com/tokudb.html>

- RocksDb中文网
 - <https://rocksdb.org.cn>
- LevelDB详解
 - <https://blog.csdn.net/linuxheik/article/details/52768223>
- Rocksdb源码剖析一----Rocksdb概述与基本组件
 - <https://blog.csdn.net/flyqwang/article/details/50096377>
- leveldb - sstable格式
 - <https://www.cnblogs.com/cobbliu/p/6194072.html>



Thank You!