levelDB原理剖析

郑罗海

LSM Tree

定义

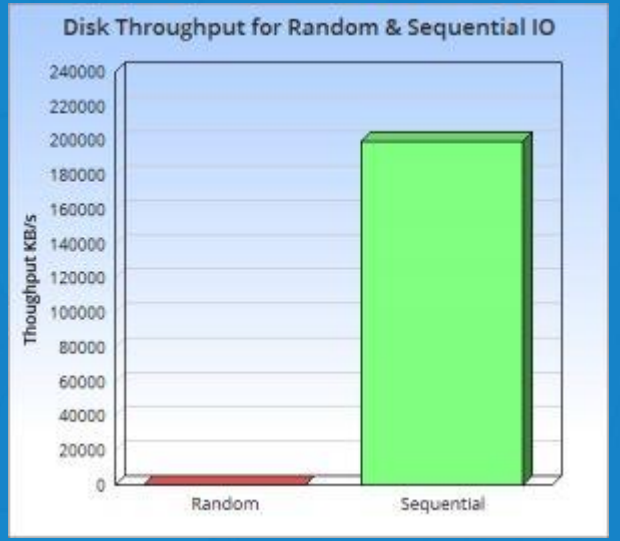
- LSM (Log Structured-Merge) Tree是google的" BigTable"论文提到的一种文件组织方式。

应用

- Hbase， Cassandra，LevelDB

核心思想

- 通过分层，有序，面向磁盘的数据结构，把磁盘的随机写合并成顺序写，从而达到极高的写性能
- 在读放大，空间放大，写放大三者中寻求最优平衡点



levelDB简介

levelDB基本用法

- Get， Put，Delete
- Put和Delete可以组合成批操作
- Snapshot快照查询

RocksDB

- Facebook在levelDB某一个分支基础上改造而来，主要对一些细节进行调整和对外暴露更多配置参数

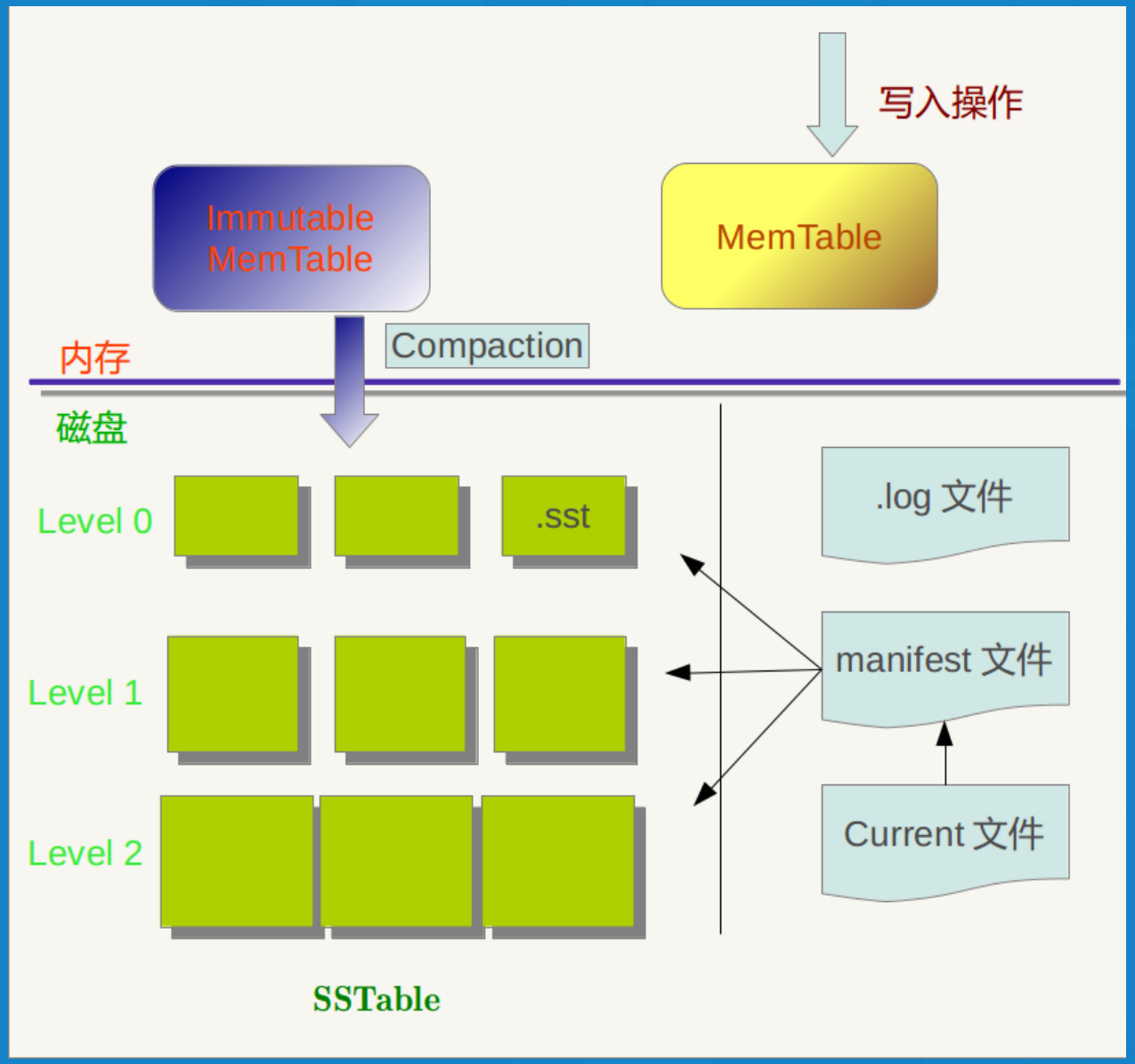
levelDB的局限

- 非关系型数据库
- 只提供sdk，需要集成到调用方进程里
- 单库限制单进程访问，数据无法分布式存储

levelDB的应用场景

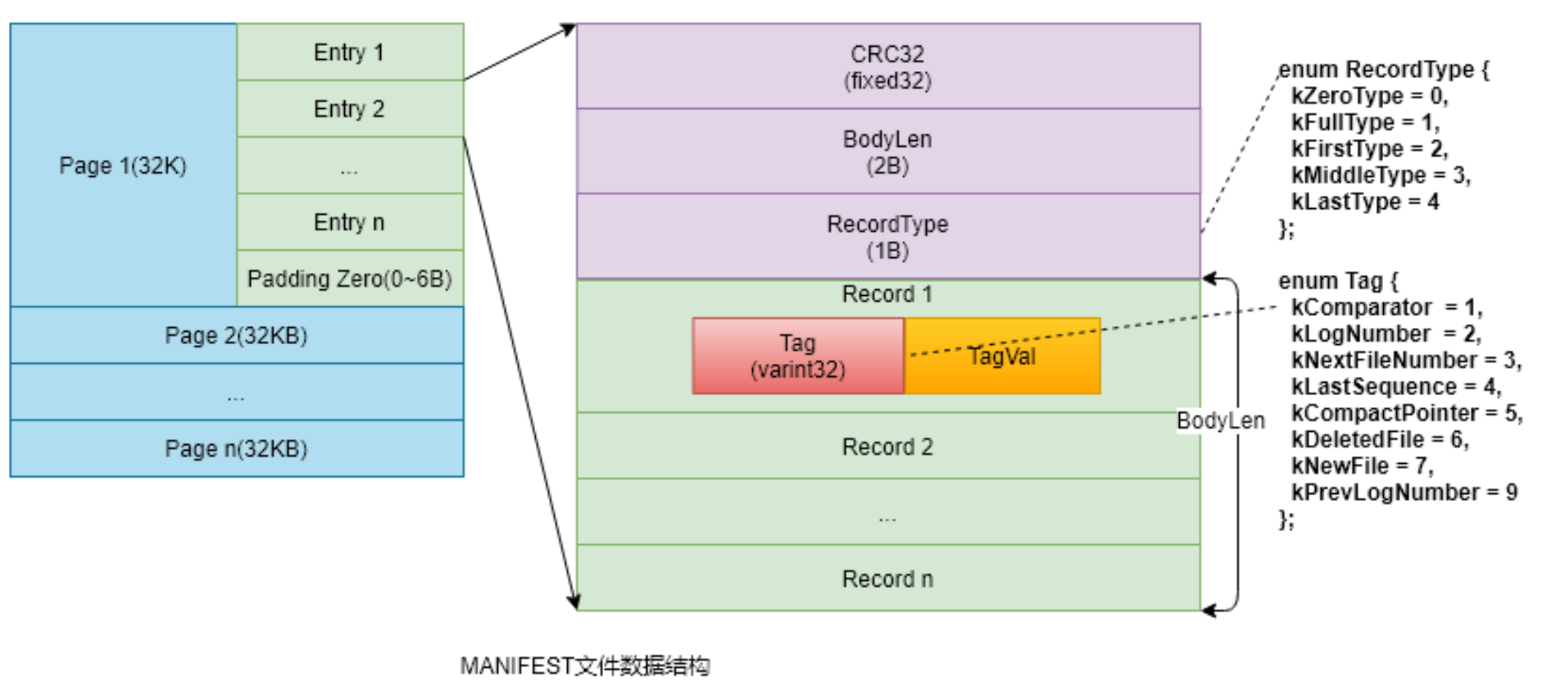
- 适用于写多读少的场景
- 短小精悍的存储引擎：mysql存储引擎myrocks，TiDB存储引擎，SSDB
- 轻量，部署简单：区块链，Chrome浏览器，Google earth

系统架构



文件类型	说明
CURRENT	记录当前使用的清单文件名
MANIFEST-[0-9]	清单文件
[0-9] .log	写备份文件
[0-9] .sst(ldb)	数据文件
LOCK	Db锁文件
LOG	日志文件

MANIFEST清单文件



MANIFEST文件数据结构

- 引入清单文件的目的主要是：1，提供sst文件的快速索引；2，协助levelDB实现版本管理(清单文件映射到内存后会变成一个version列表，列表头元素就是当前最新版本)；3，协助服务宕机后的恢复

MANIFEST清单文件

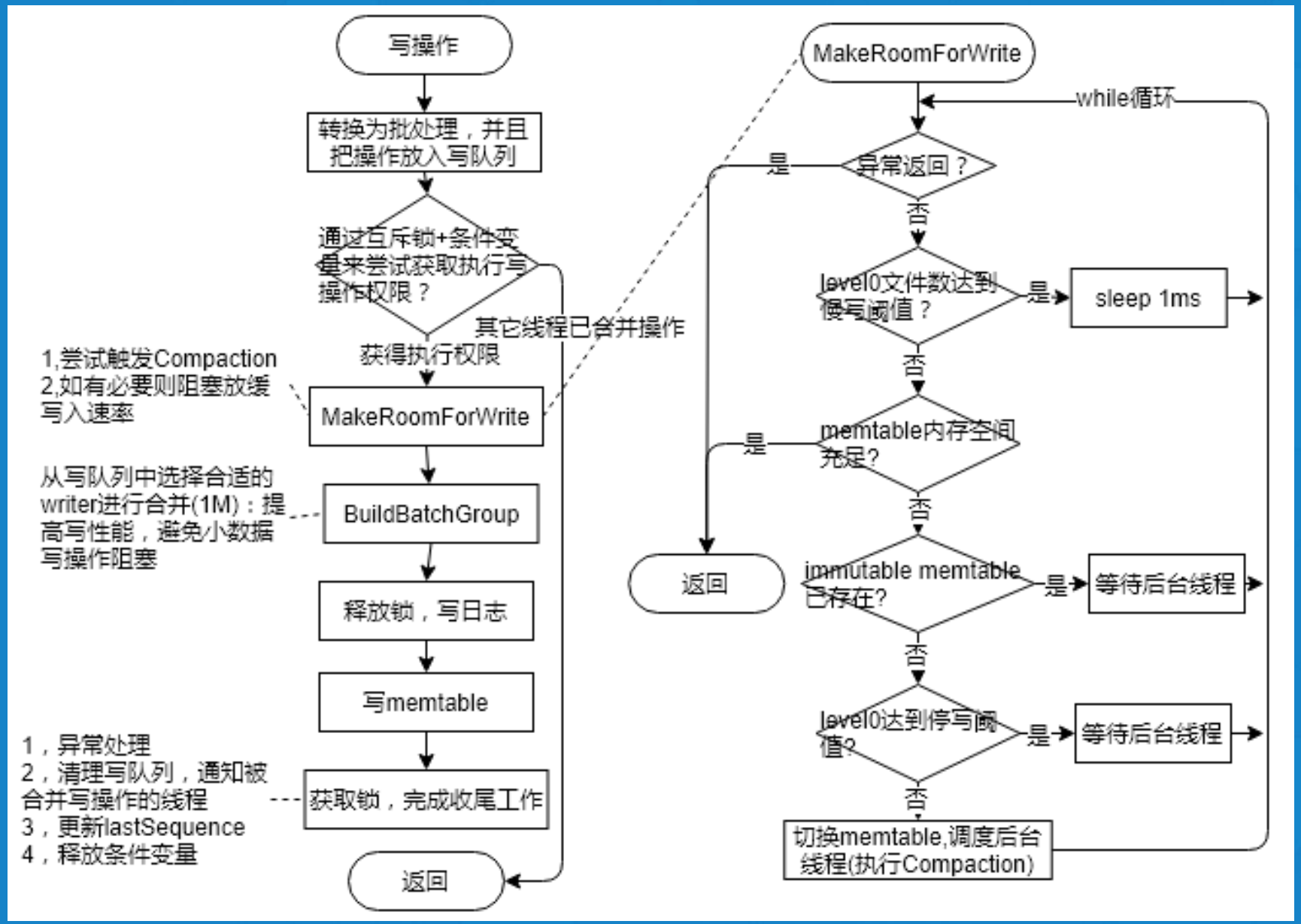
Level DB Manifest Format

Type=kComdparator (Varint32)	Comparator Name Length (Varint32)	Comparator Name (String)	
Type=kLogNumber (Varint32)	Log Number (Varint64)		
Type=kPrevLogNumber (Varint32)	Prev Log Number (Varint64)		
Type=kNextFileNumber (Varint32)	Next File Number (Varint64)		
Type=kLastSequence (Varint32)	Last Sequence Number (Varint64)		
Type=kCompactPointer (Varint32)	Level (Varint32)	Internal Key Length (Varint64)	Internal Key (String)
...			
Type=kCompactPointer (Varint32)	Level (Varint32)	Internal Key Length (Varint64)	Internal Key (String)
Type=kDeletedFile (Varint32)	Level (Varint32)	File Num (Varint64)	
...			
Type=kDeletedFile (Varint32)	Level (Varint32)	File Num (Varint64)	
Type=kNewFile (Varint32)	Level (Varint32)	File Num (Varint64)	File Size (Varint64)
Smallest Key Length (Varint32)	Smallest Key (String)	Largest Key Length (Varint32)	Largest Key (String)
...			
Type=kNewFile (Varint32)	Level (Varint32)	File Num (Varint64)	File Size (Varint64)
Smallest Key Length (Varint32)	Smallest Key (String)	Largest Key Length (Varint32)	Largest Key (String)

- 清单文件的详细存储信息

例如新增一个sst文件时，就会在清单文件中增加一个KNewFile类型的记录

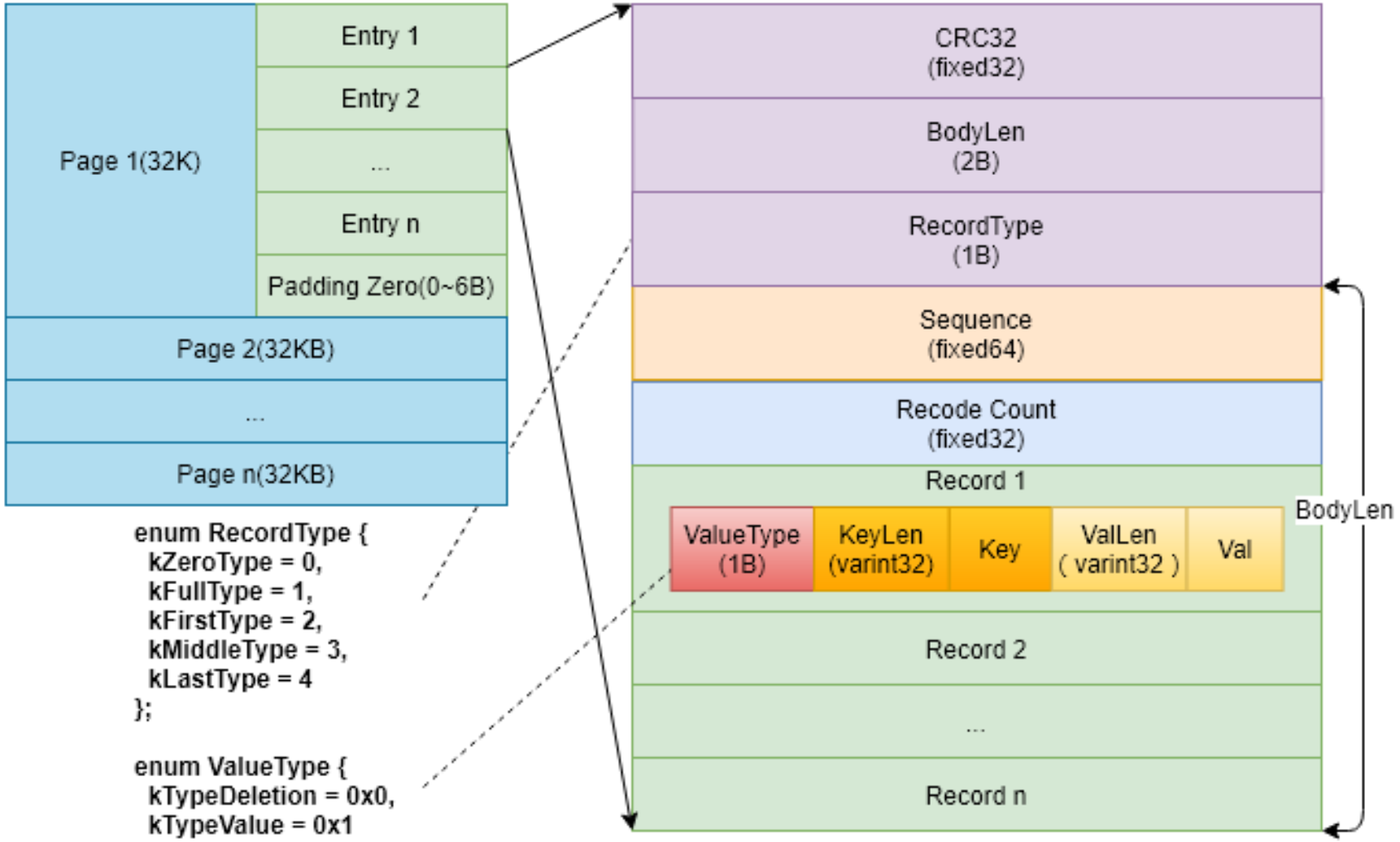
写流程图



亮点

- 构造了一个写队列(合并写)，提高写性能。同时也避免小数据量的写请求被长时间阻塞。
- 由于写队列存在，写日志和memtable这种耗时操作不需要加锁，释放锁去做其他资源同步。

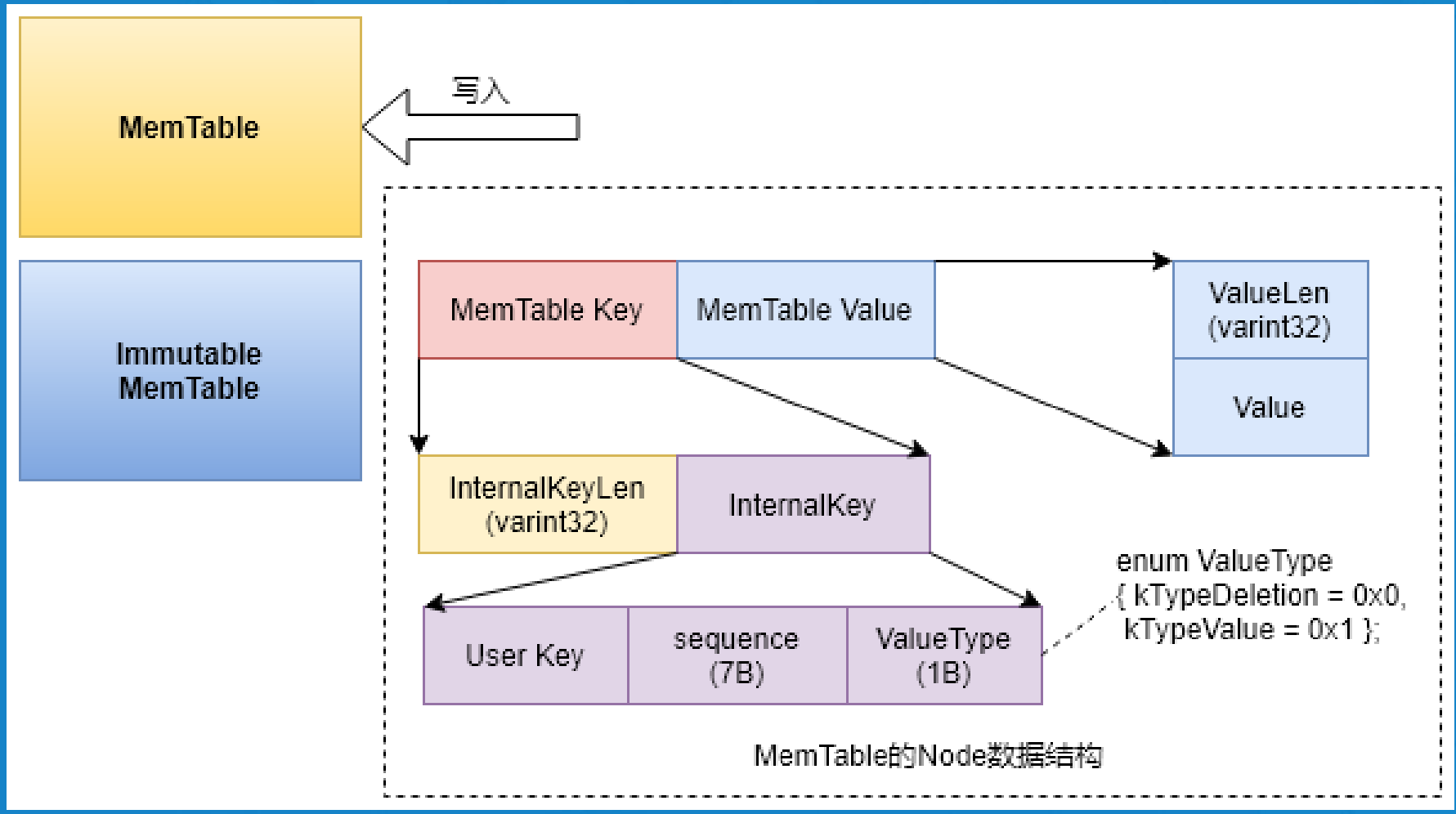
写入log文件



log文件数据结构

- levelDB写入时都是先写log文件，服务宕机内存丢失，重启时可以从log文件恢复
- 左图是log文件存储的数据结构
- levelDB通过mmap方式来访问log文件。

写入内存

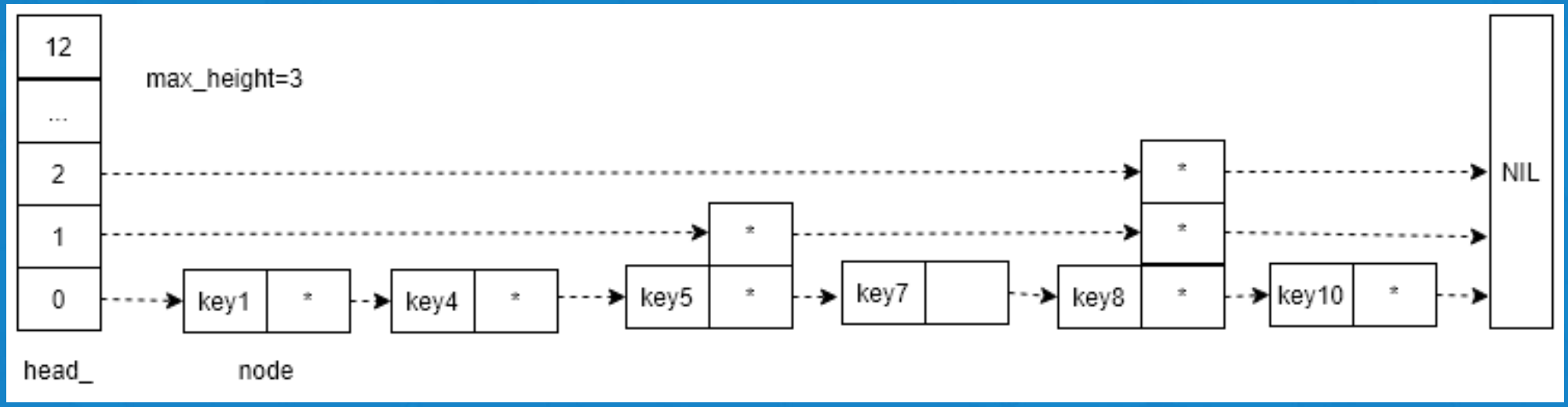


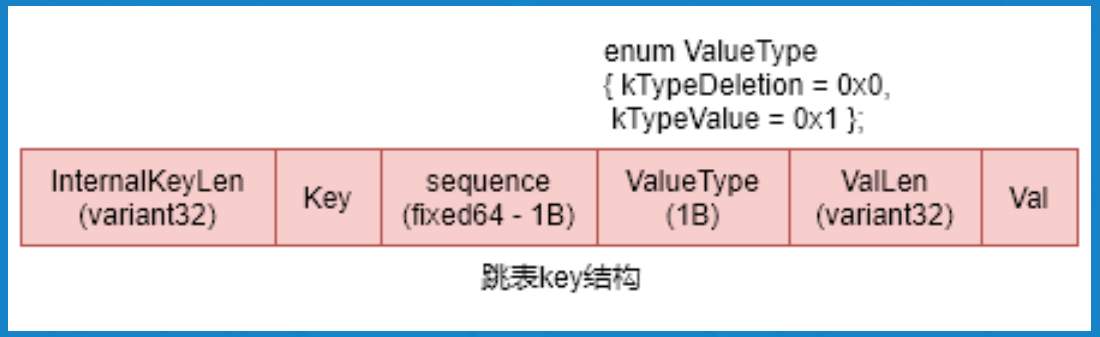
- 双Buffer机制

  levelDB维护两个Memtable，当一个Memtable写满时，自动转换为只读的Immutable Memtable。后台线程会在适当时机将Immutable Memtable刷入磁盘。

- 左图虚线框内是写入Memtable的Key-Value数据结构

MemTable的算法-跳表





亮点

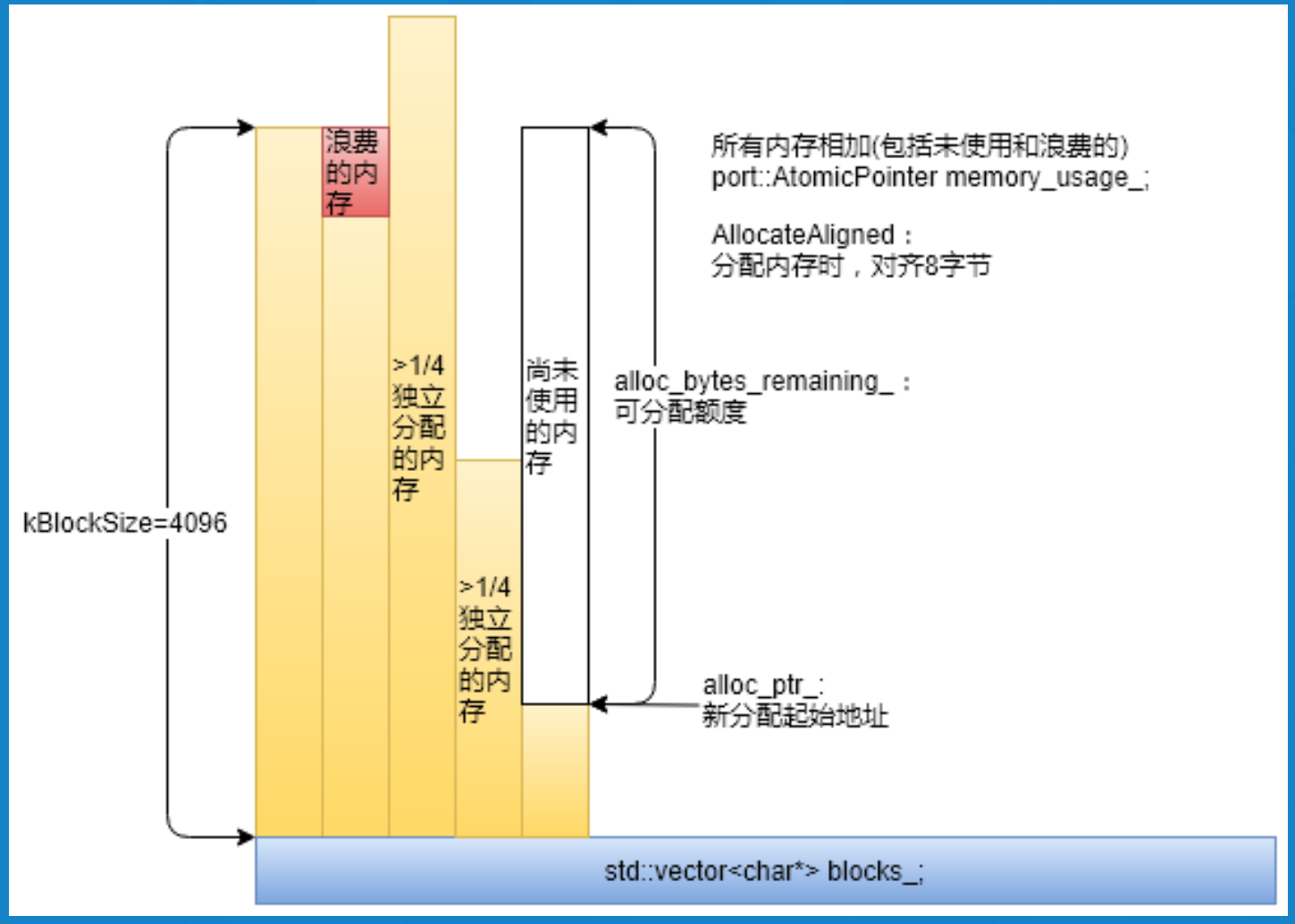
- 查询和插入性能类似二分查找
- 支持多线程无锁读写
- 支持snapshot快照查询
- 相同user_key，最新版本的数据排在前面

Key比较逻辑:

1，优先比较key，按照字典序排序，key越小越排在前面

2，key相等则比较sequence+ValueType，值越大反而排在前面

内存分配管理Arena



- Arena管理内存

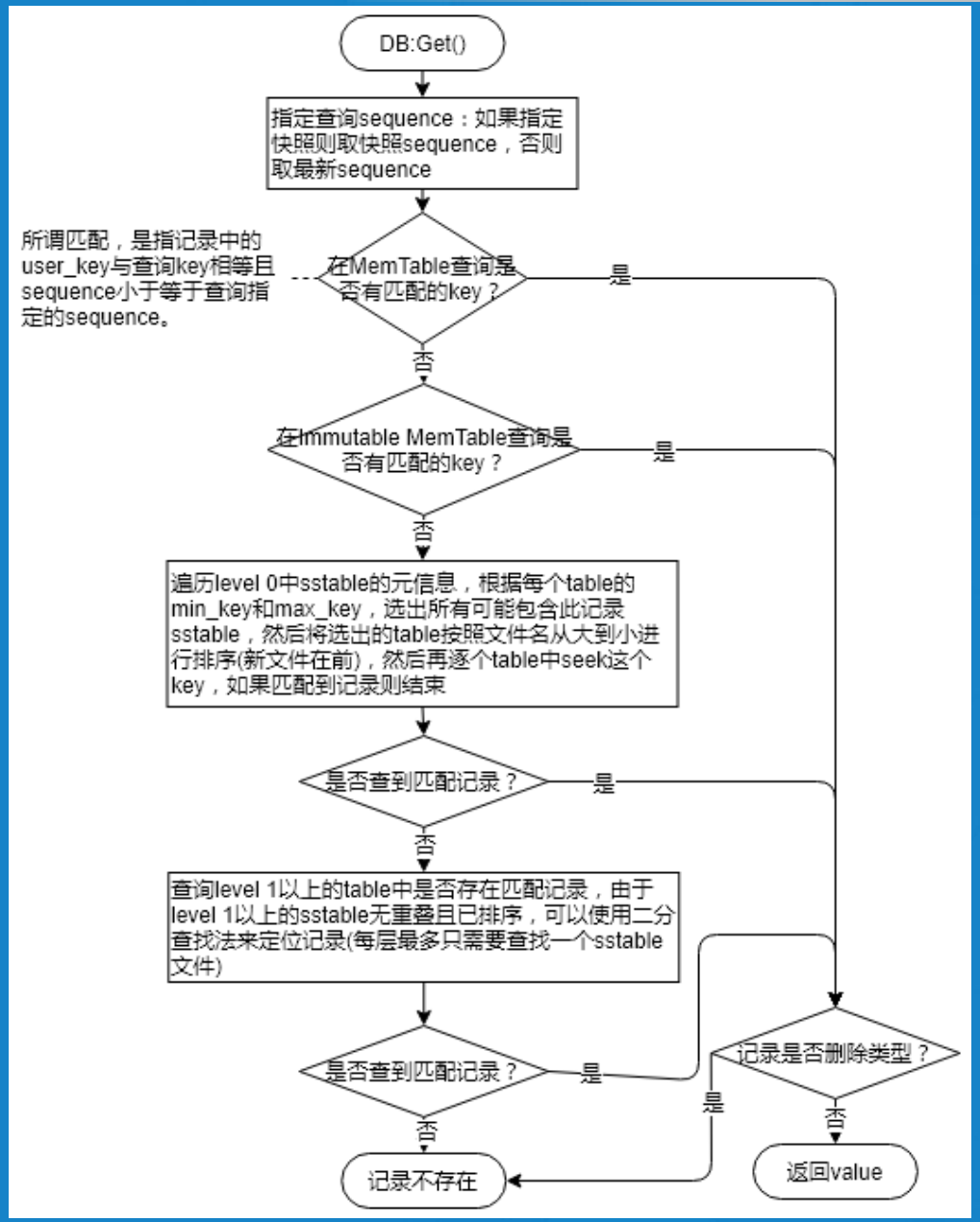
  以4k或者大于1k为单位申请内存，避免小内存频繁申请。

- Slice字符串类

  level实现了Slice类用来管理(Arana分配的)内存指针和位移，避免了内存拷贝。

解决小内存频繁申请的问题

查询流程图



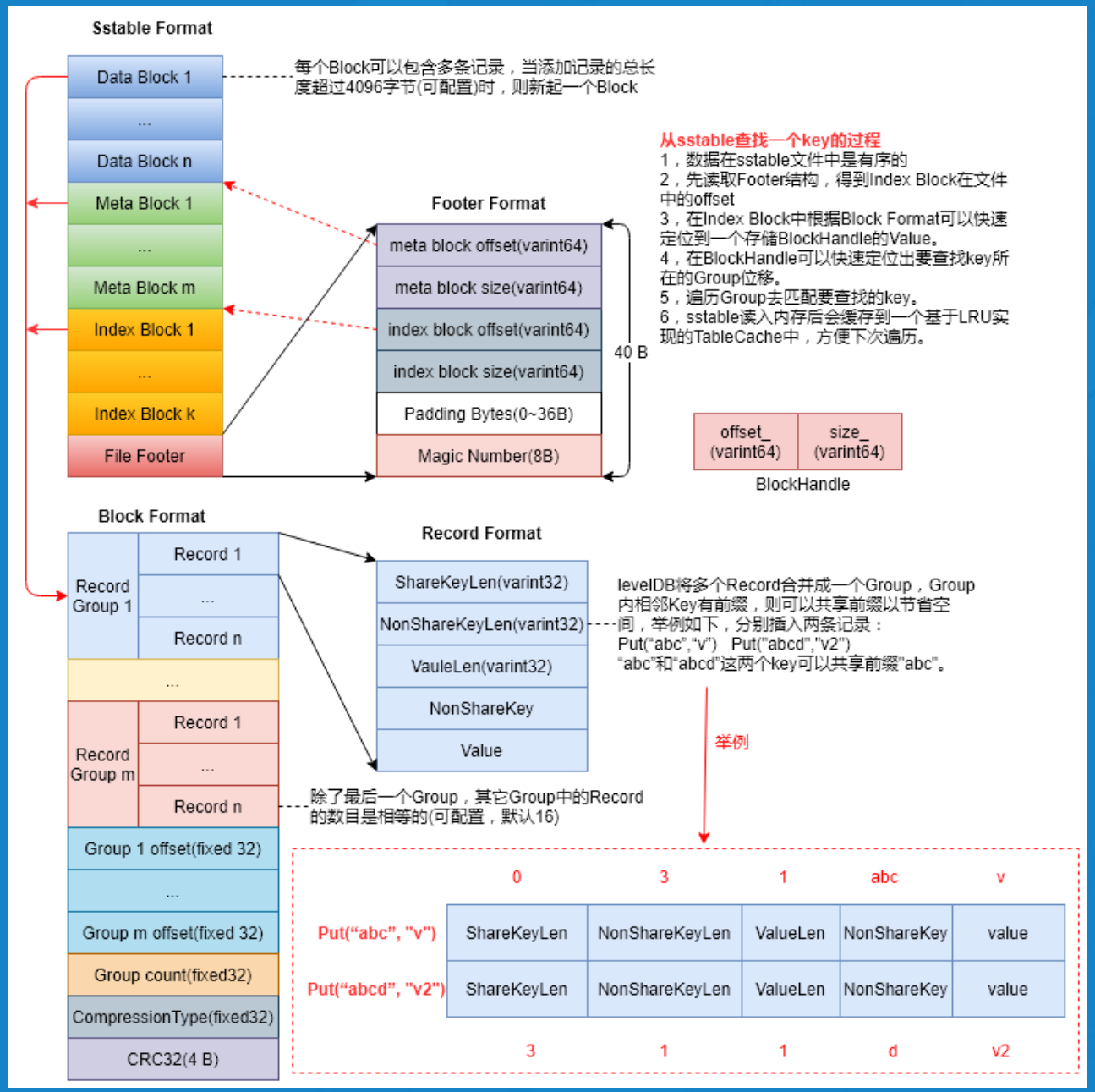
- 读MemTable

  从MemTable读取就是跳表读取，由于跳表中相同的usr_key，版本号最大的放在前面，所以第一个读到总是最新的user_key内容。

- 从sstable文件读取

  sstable文件有特殊的数据结构(见下页)，levelDB可以快速seek到一个文件中的某个key。

sst文件数据结构



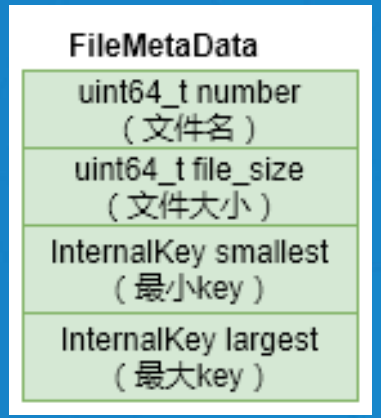
➢Sstable文件

1，sstable文件是levelDB用来存储数据的库文件。

2，sstable存储的数据是按照key排序，相邻key有公共前缀则会共享前缀以压缩空间。

3，sstable的数据结构提供快速索引一个key的方法，性能类似二分查找。

4，每新增一个sst文件，都会在MANIFEST清单文件中增加一条记录，记录内容如下图



levelDB的compaction

Compaction的目的

- 数据持久化

  把memTable内存中的数据刷到磁盘

- 平衡读写差异，提高读写性能

  通过compaction控制第i层的数据大小接近(10^i)MB(0<i<6)，同时也会尝试将冷门数据往高层合并。

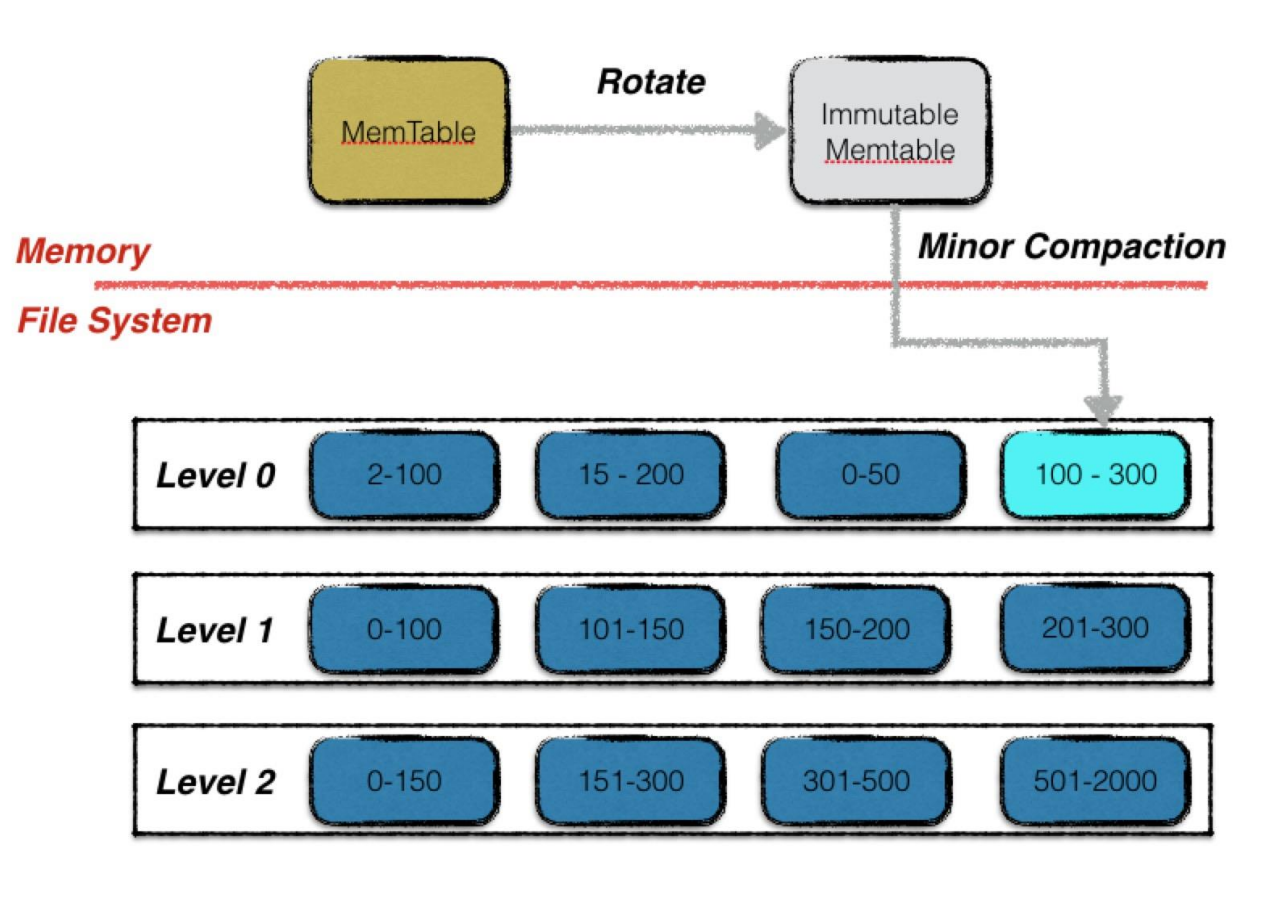
- 整理数据

  levelDB数据的更新和删除都是延后执行，只有在compaction的时候才会真正合并和释放空间。

Compaction的分类

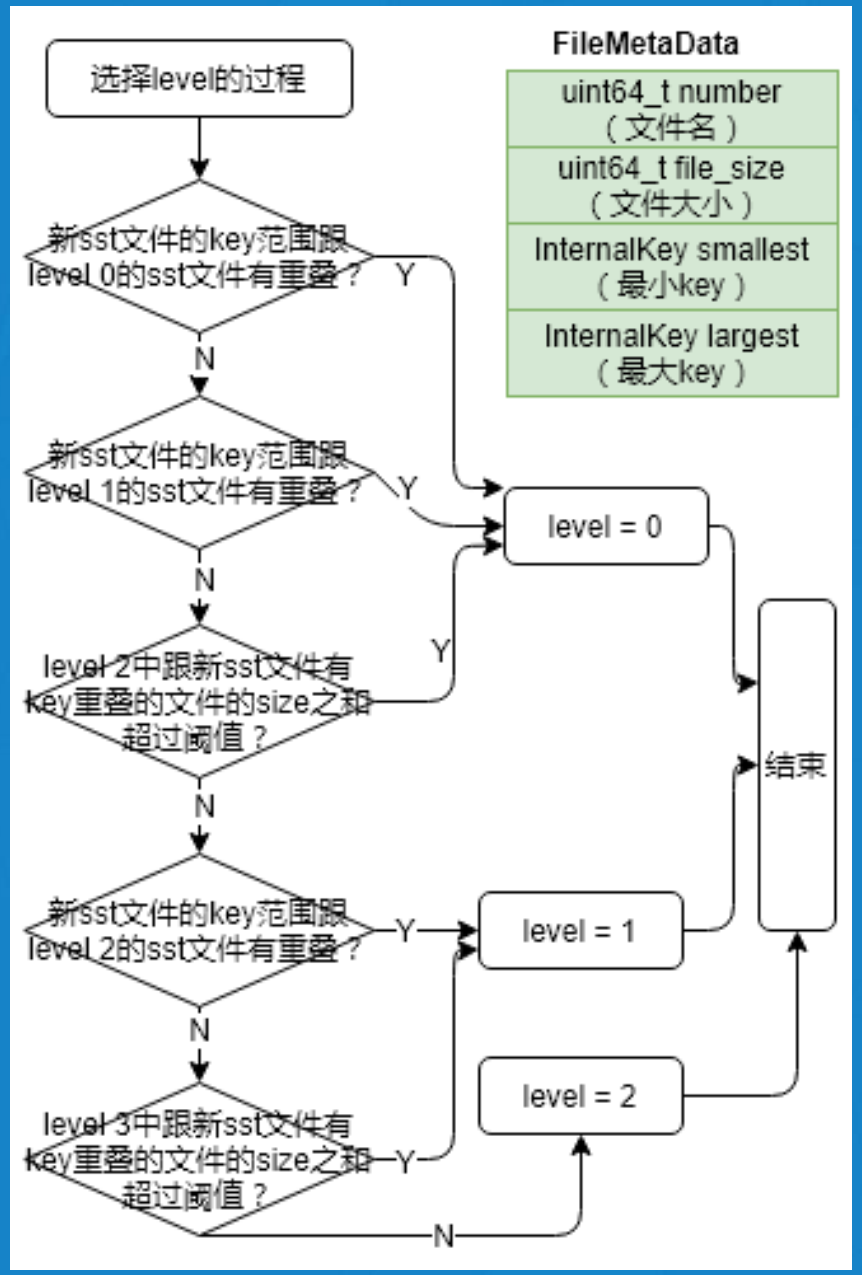
- minor compaction：将MemTable内存刷入磁盘
- major compaction：调整7层sstable的数据

minor compaction



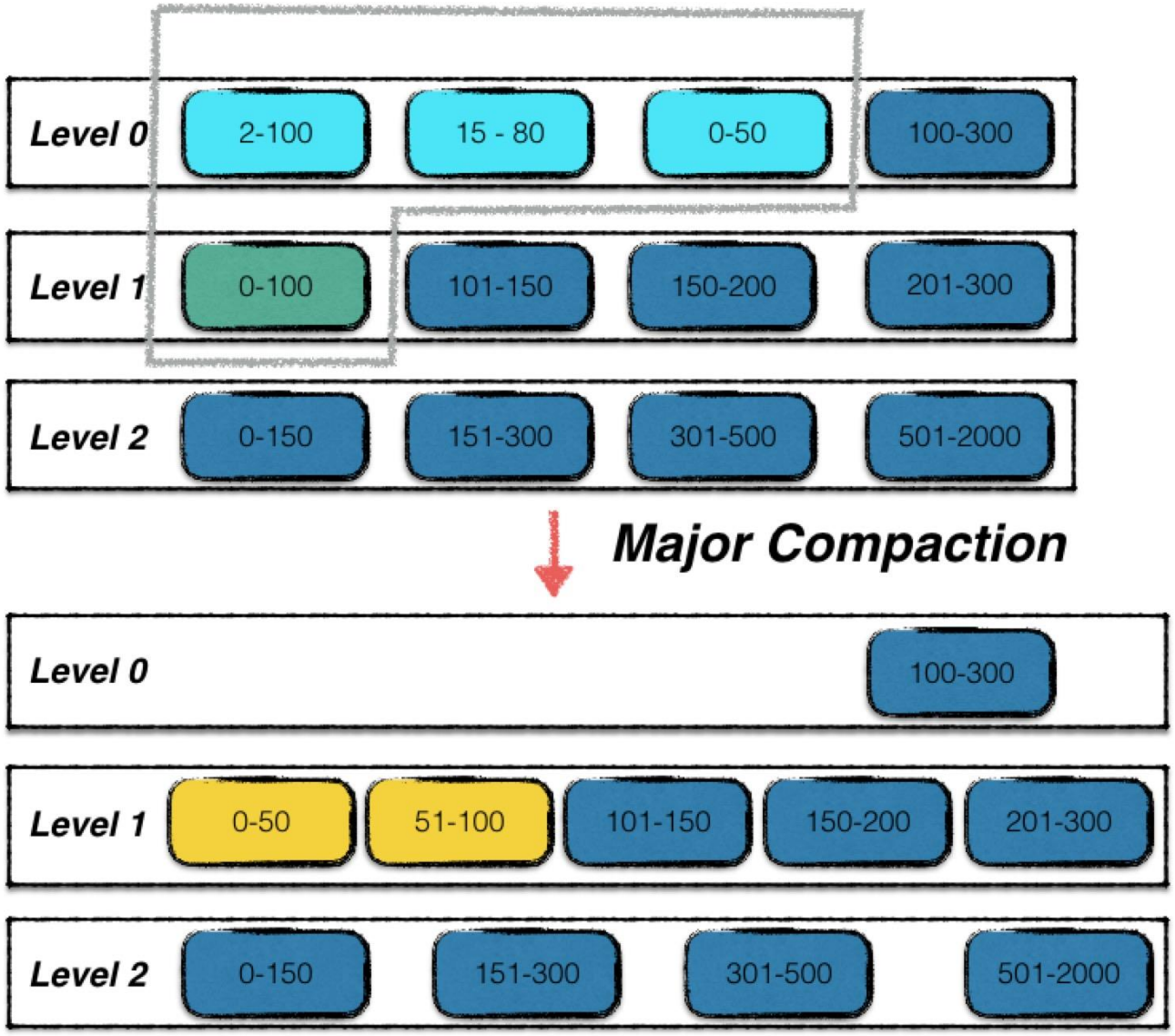
- 写DB时触发，默认MemTable大小超过4MB就会触发。
- 过程：
  1，将内存数据刷盘为一个sst文件。
  2，选择新sst文件要放置的level。

选择level的过程



- Minor compaction为新sst文件选择level的过程是一个权衡折中的过程。
- 限制选择的最高level不超过level 2。
- 做了预判，尽量避免后续频繁compaction(节省IO)。

major compaction



- Major compaction是将不同层的sst文件进行合并。
- 触发条件：

  1，Manual:手工调用触发。

  2，Size：当level i层文件总大小超过(10^i)MB触发。

  3, Seek：某个文件无效读取次数过多也会触发。
- 合并优先级：

  Minor > Manual > Size > Seek

  |_____major_____|

Size compaction

- Level会在当前版本为每一层都维护一个score，其中

  第0层的score=0层文件总数/4

  第i层的score=i层文件总大小/(10^i)MB，其中1<i<6

- 核心过程

  1，选出上面score的最大值，如果score>1则确定在该层执行compaction(假设是第n层)。

  2，选出第n层中参与合并的sst文件列表：

  2.1，如果n==0：遍历所有sst文件，存在重叠的sst文件加入到合并列表中。

  2.2，n!=0：获取当层上次操作compaction的largest key赋值为begin_key，然后顺序遍历第n层的sst文件，获取第一个largest key>begin_key的sst文件加入到合并列表。

  3，选出第n+1层参与合并的sst文件列表：

  3.1，计算出从第n层得到的文件列表的key的范围：begin_key和end_key。

  3.2，根据begin_key和end_key去遍历n+1层的sst文件，选出有重叠的文件加入合并列表。

  4，合并2跟3得到的文件列表，计算列表文件总size，如果总size小于阈值50M，则会继续尝试在第n层选择合适的sst文件。

  5，对最终列表中所有sst文件，通过归并排序算法计算出若干个新的sst文件放入n+1层。这个过程中会合并update数据，释放delete数据。

Seek compaction

- Level每生成一个sst文件时，都会设置一个allowed_seek阈值：
  allowed_seek=max(新sst文件的大小/16K, 100)
- 每次查询key，如果命中了sst文件，但是在文件中找不到key，则将allowed_seek减一
- 当第n层某个sst文件的allowed_seek<0，那么该文件会放入合并列表，后续流程跟size compaction类似，把文件合并到n+1层。

横向比较

数据库	丢数据	读性能	写性能	支持数据类型	内存依赖	事务
levelDB |rocksDB	不会	*	高	Kev-value	小	不支持
redis	可能	高	高	各种类型	大	*
mysql (innoDB)	不会	*	低	*	小	支持

谢谢聆听