LevelDB实现解读

2017/10/26 张涛

大纲

- 简介
- 整体架构
- Memtable
- Log
- SSTable
- Cache
- 版本控制
- MANIFEST
- Compaction
- 接口流程
- 其他及优化细节
- 总结

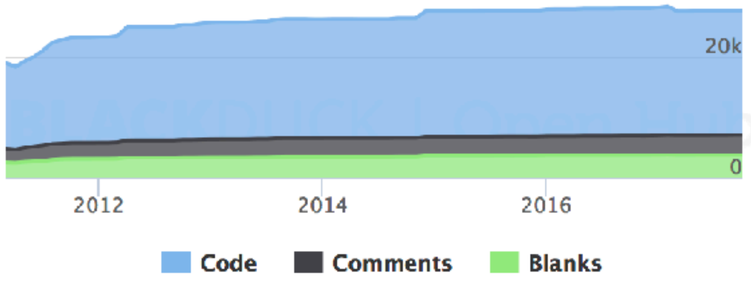
LevelDB——简介

什么是LevelDB?

- LevelDB是一个轻量级、单机平台下、高性能、持久化key/value存储数据库，由Google的Jeff Dean和Sanjay Ghemawat在2011年开发。
- LevelDB是借鉴了Google BigTable的设计理念，但没有使用一行BigTable中的代码，所以它可以开放源码发布。
- Dean和Ghemawat设计LevelDB的初衷是为了替换SQLite 来实现Chrome’s IndexedDB的后台备份。

Code

Lines of Code

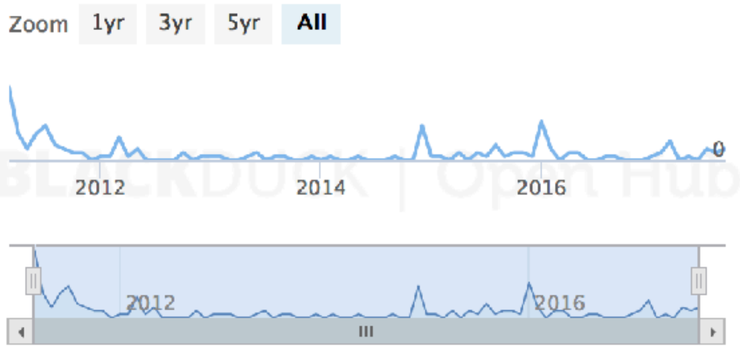


Languages

C++ 93% 4 Other 7%

Activity

Commits per Month



30 Day Summary

Sep 24 2017 — Oct 24 2017

13 Commits

3 Contributors

12 Month Summary

Oct 24 2016 — Oct 24 2017

29 Commits

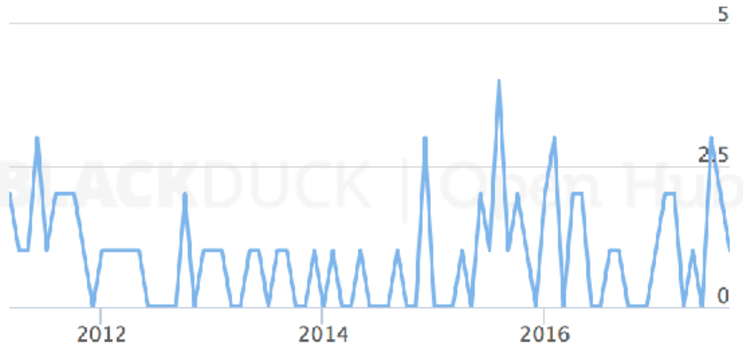
Up + 6 (26%) from previous 12 months

5 Contributors

Down -7 (58%) from previous 12 months

Community

Contributors per Month



Most Recent Contributors

- cmumford
- sanjay
- costan
- corrado
- scrubbed
- davidair

Ratings

2 users rate this project:

3.0/5.0

Click to add your rating

Review this Project!

LevelDB是什么与不是什么?

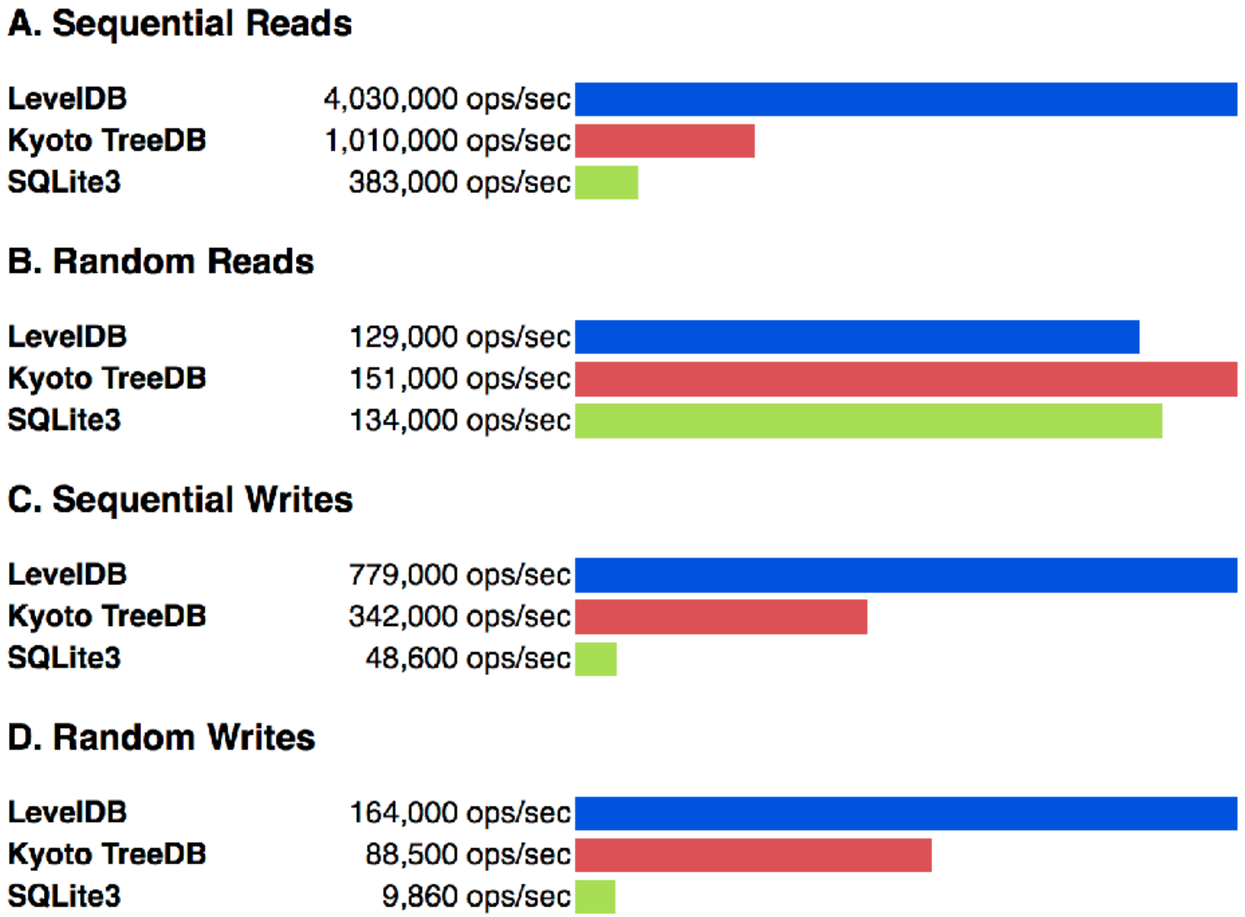
- LevelDB只是一个C++类库，可以在许多项目中直接引用
- LevelDB不是一个RDBMS，它不提供查询引擎，SQL语句，和事务处理等复杂功能。
- LevelDB有许多其他语言变种的实现，如LevelUP(Node.js)，和Plyvel(Python)

LevelDB特性

- key/value都可以是任意字节长度
- 按照键来排序(排序方法可用户自定义)，且保存索引以加速查找
- 压缩存储(提供可选的Snappy压缩)节省空间
- 非常高性能的写入速度，但读取速度相对一般。非常适合写多读少的应用
- 多线程安全
- 基本操作：Get() Put() Delete() Batch()。

LevelDB-Baseline Benchmark

- 每个DB设置4M的cache
- 每个DB以异步的写方式打开
- 每对key16字节，value100字节
- 顺序读取/写入以递增的顺序遍历key空间
- 随机读取/写入以随机顺序遍历key空间。

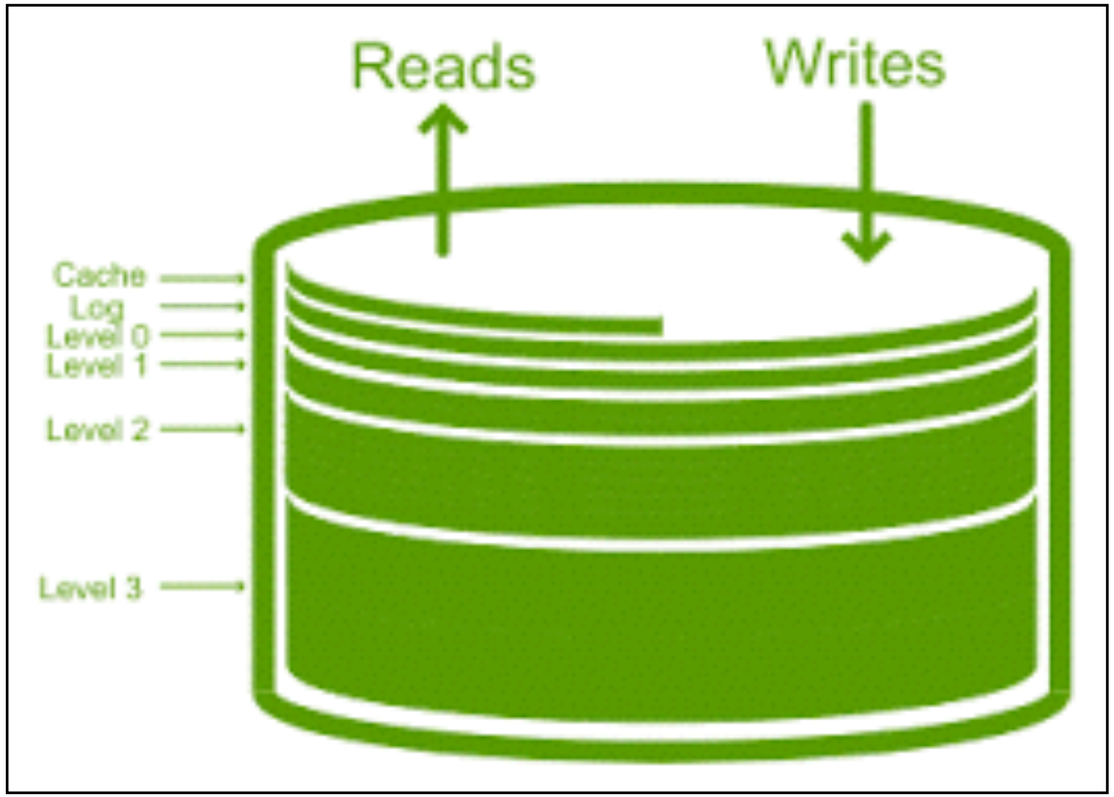


LevelDB的获取、编译和使用

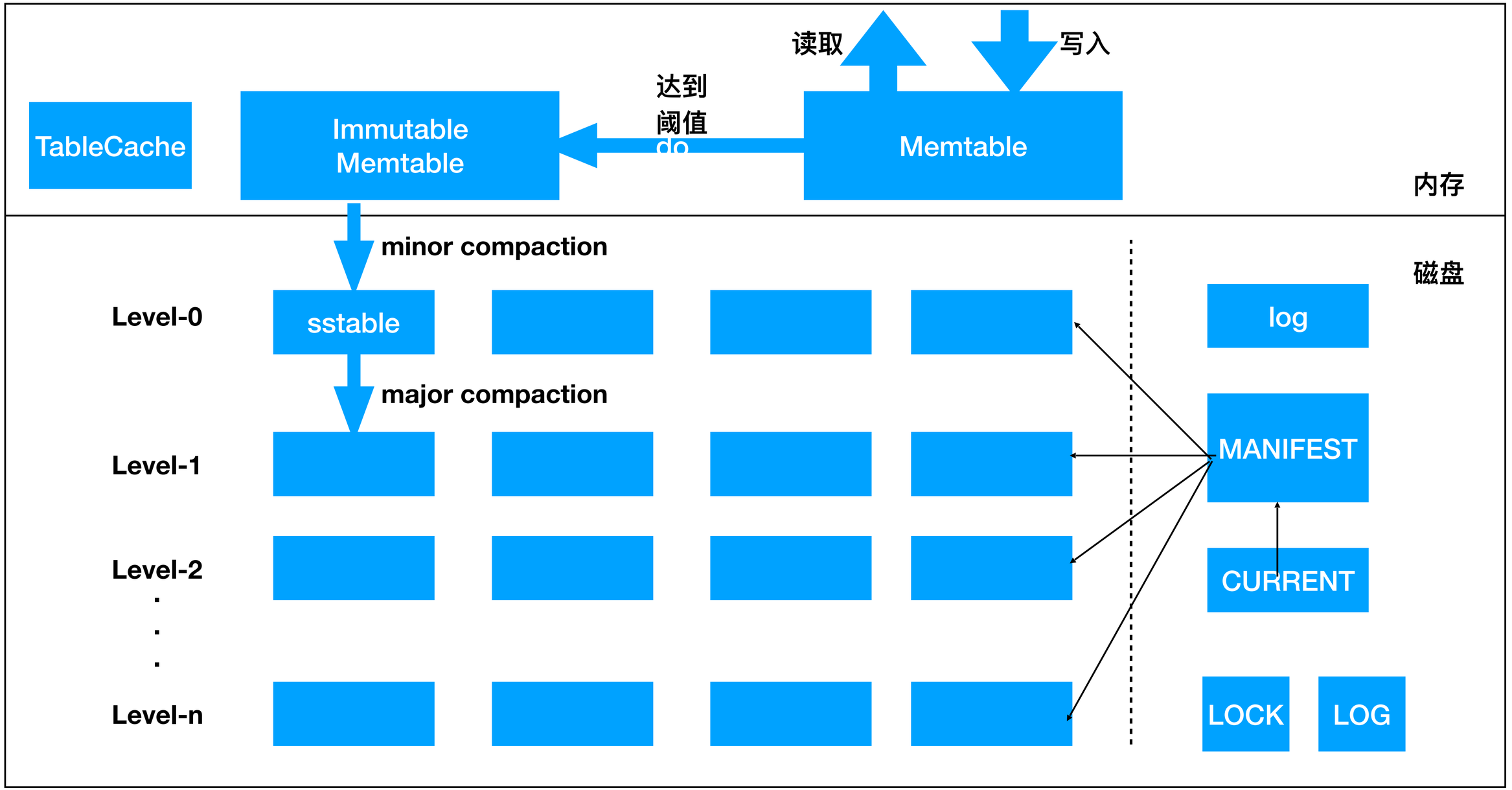
- Github地址:https://github.com/google/leveldb
- Linux、macOS下的编译与使用：
  - 在leveldb-master主目录执行make
  - 编写一个文件如test.cpp
  - g++ test.cpp -o test_result ./out-static/libleveldb.a -I ./include/ -lpthread

LevelDB——整体架构

LevelDB整体架构



LevelDB操作流程



LSM树（Log-Structured-Merge Tree）

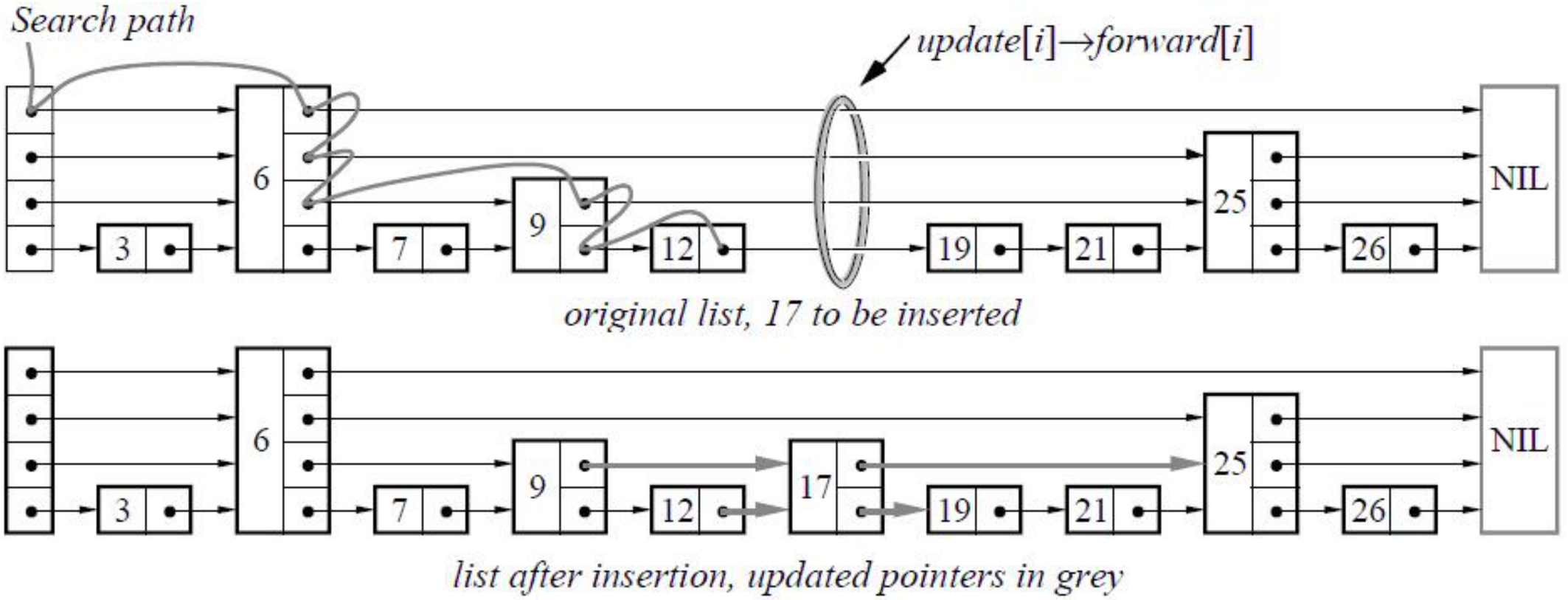
- LSM是levelDB的存储策略或者说文件结构策略。
- 基本思想:将修改的数据保存在内存，达到一定数量后在将修改的数据批量写入磁盘。读取时需要合并磁盘中的历史数据和内存中最近的修改操作。
- LSM存储模型同样支持增、删、读、改以及顺序扫描操作
- LSM模型利用批量写入有效的规避了随机写问题，虽然牺牲了部分读性能，但大大提升了写性能。

LevelDB——Memtable

LevelDB组件——Memtable

- **Memtable**：内存数据结构，跳表实现，新的数据会首先写入这里,，sstable中的数据均来源于此
- 两种Memtable
  - Activated memtable
  - Immutable memtable
- Memtable底层由SkipList实现,所有数据都是按照key有序存放
- 接口内容:
  - Get():从memtable中获取一条记录
  - Add():将一条记录插入到memtable中
  - Iterator():返回memtable的迭代器。底层是调用SkipList封装好的迭代器

Memtable——SkipList示意图

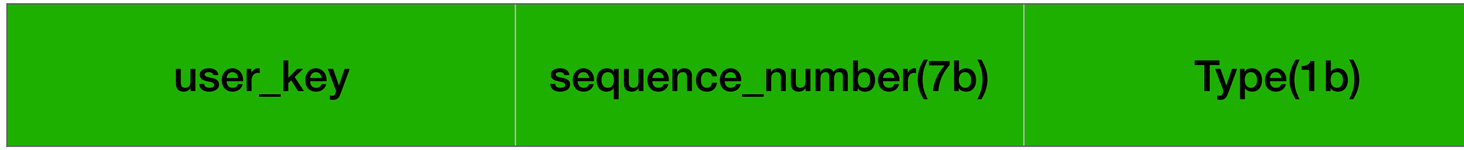


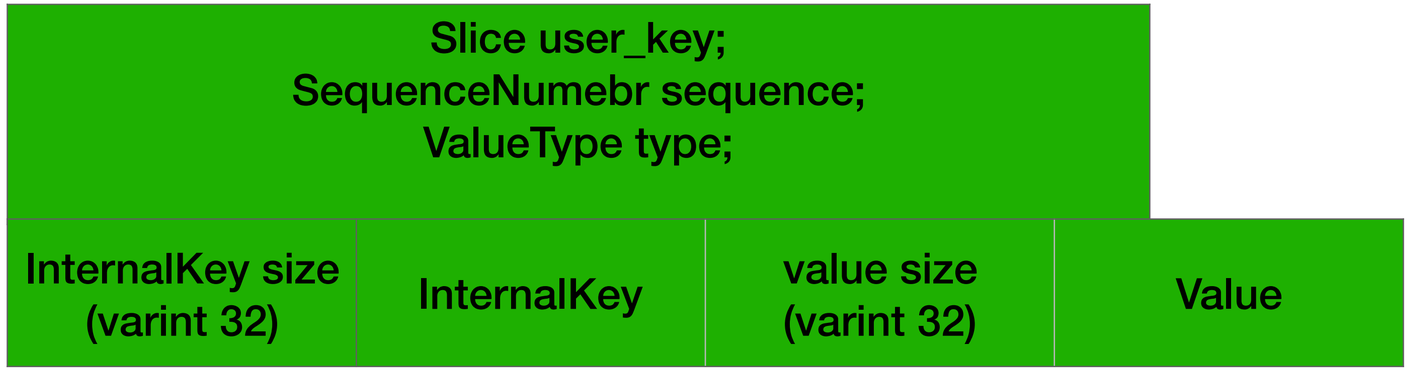
Memtable——SkipList特点

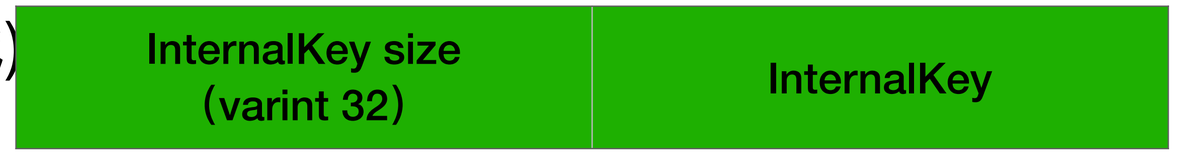
- 有序链表+二分查找
- 大约有O(log n)层
- 空间复杂度:O(log n)
- 期望的查找、插入、删除时间:O(log n)
- level+1的元素总数几乎是level层的1/2
- 从最顶层开始查找，依次向后遍历，若到达尾部则从下一层开始查找。

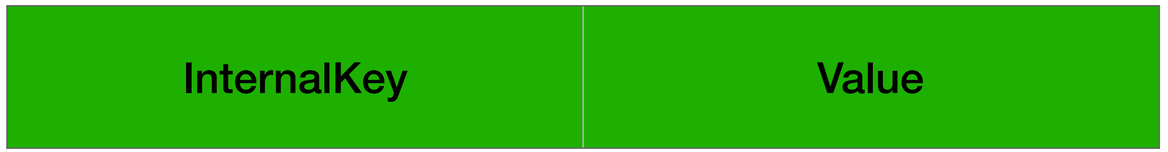
LevelDB中的各种键类型

- InternalKey
- ParsedInternalKey(是个struct)
- SkipList内部存储的key
- LookupKey(用于memtable中查找)
- SSTable中存储的键值格式





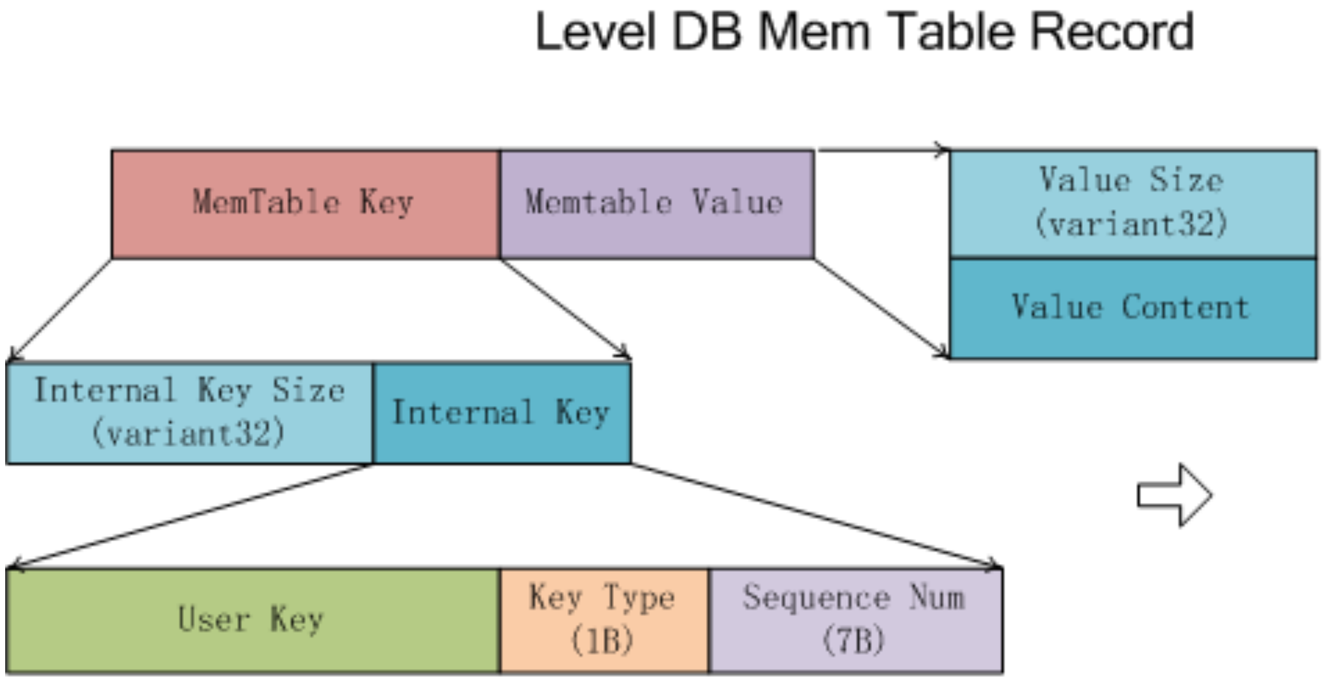




数据存储

- leveldb所有数据都是字符形式，即使是整型，也将被转换为字符型存储。
- leveldb有两种整型和字符型数据转换。一种是fixed，一种是varint。
  - fixed转换相对简单，就是将int的每一个字节存入字符数组中即可
  - varint这种转型将一个字节分成两部分，前7个字节存储数据，第8个字节表示高位是否还有数据

Memtable键值格式

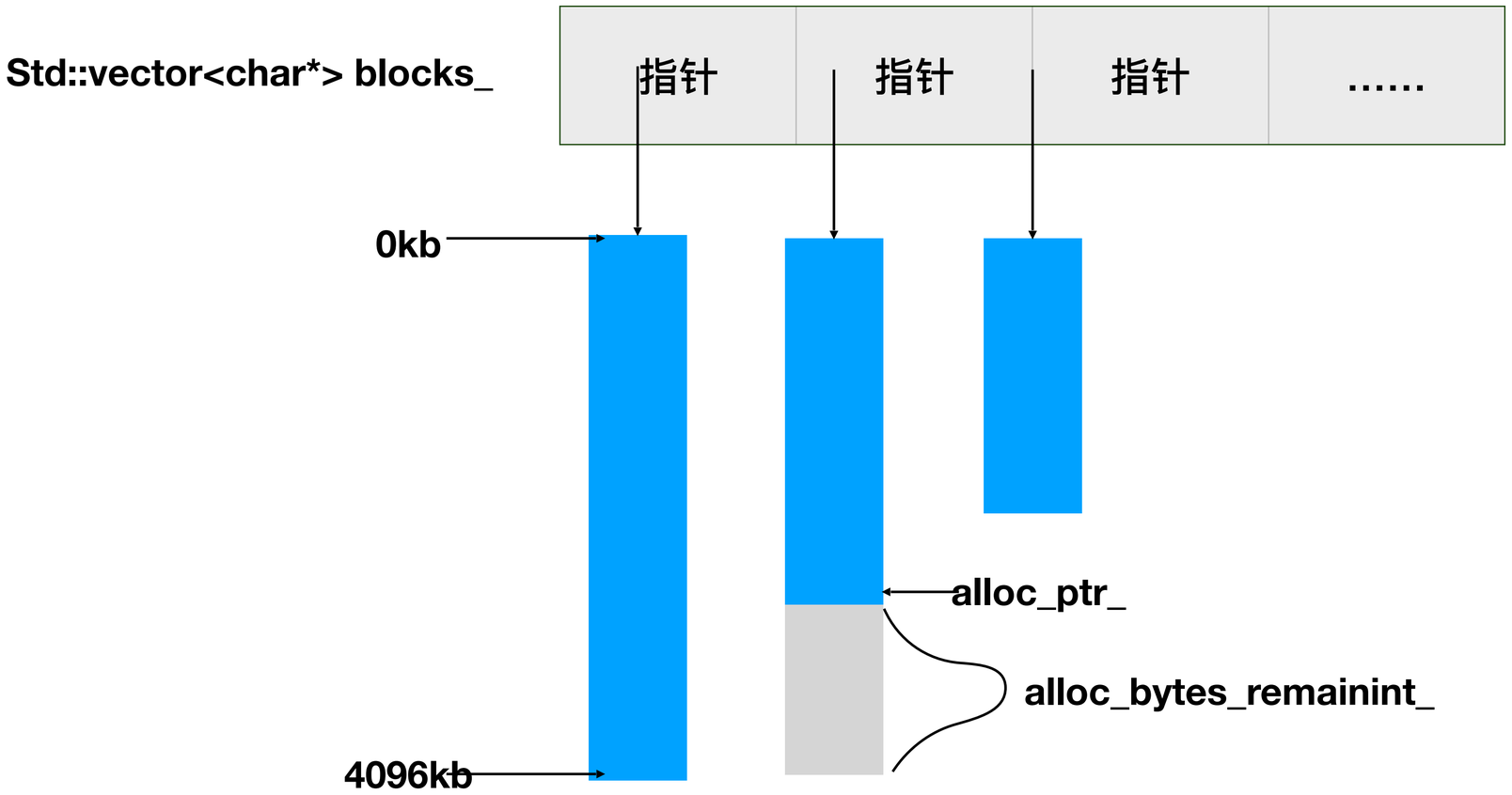


Sequence Num在保存时只保存低7字节，所以取值范围为[0,2^{56}-1)。
Key Type有两种取值：0表示删除，1表示更新。
Internal Key的比较方法：先比较User Key，如果User Key相等，再比较Sequence Num，Sequence Num大的反而比较小。这样可以使得最新的key在跳表中排在前面。
在获取两个拥有不同的user key的internal key之间的最短分割key时，先获取它们user key的分割串，然后Sequence Num取256-1，Key Type取1；寻找最小后继key的方法与此相同。

Memtable——Arena内存池

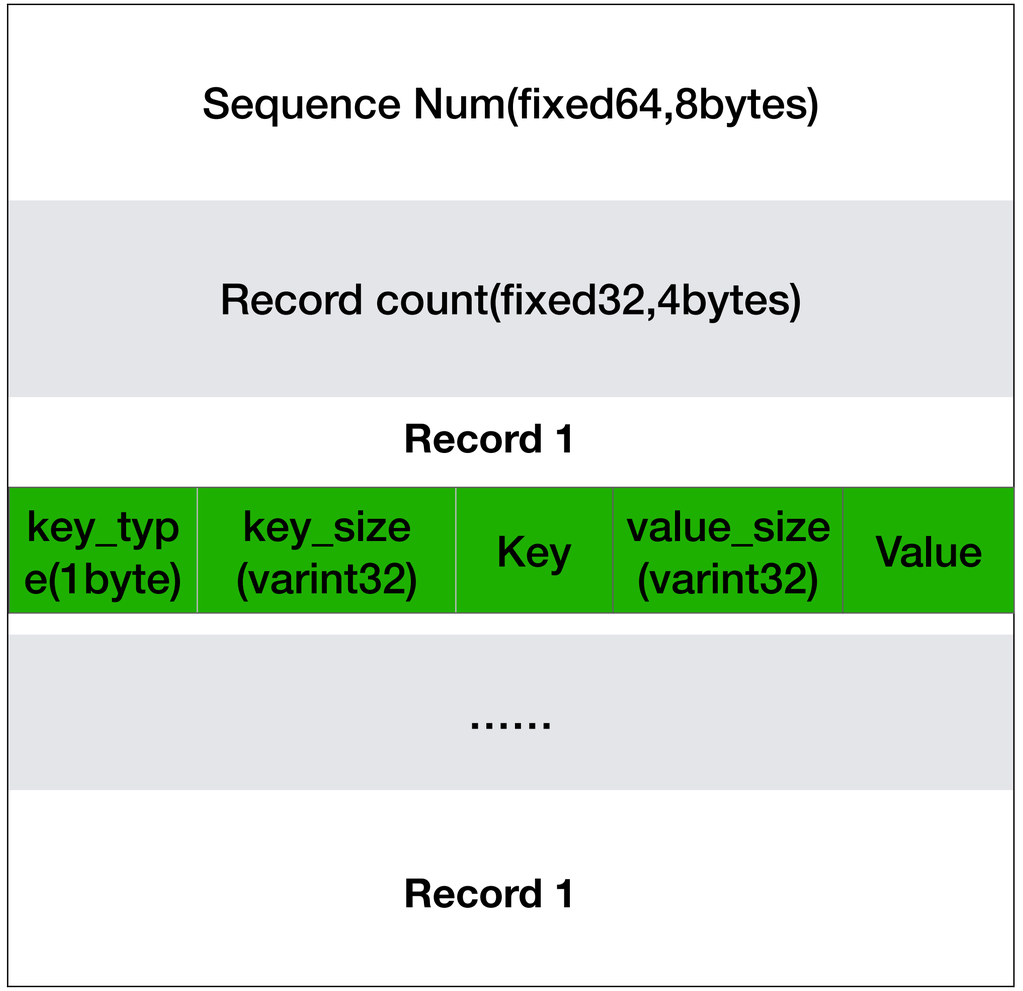
- 内存池的作用:完成内存管理。包括申请、分配、释放内存。
- 优点：减少malloc/free(new/delete)调用的次数，减少内存申请或分配所带来的系统开销。
- 申请内存和分配内存的区别:
  - 申请内存：使用new来向操作系统申请一块连续的内存区域。
  - 分配内存：将已经申请的内存分配给项目组件(如memtable)使用。

Memtable——Arena内存池示意图



WriteBatch操作

- WriteBatch类完成批量的读写操作.
- WriteBatch提供了Put和Delete接口
- 即使是单条记录的Put、Delete其实现时都需要先创建一个WriteBatch对象，然后调用WriteBatch的Put、Delete方法。



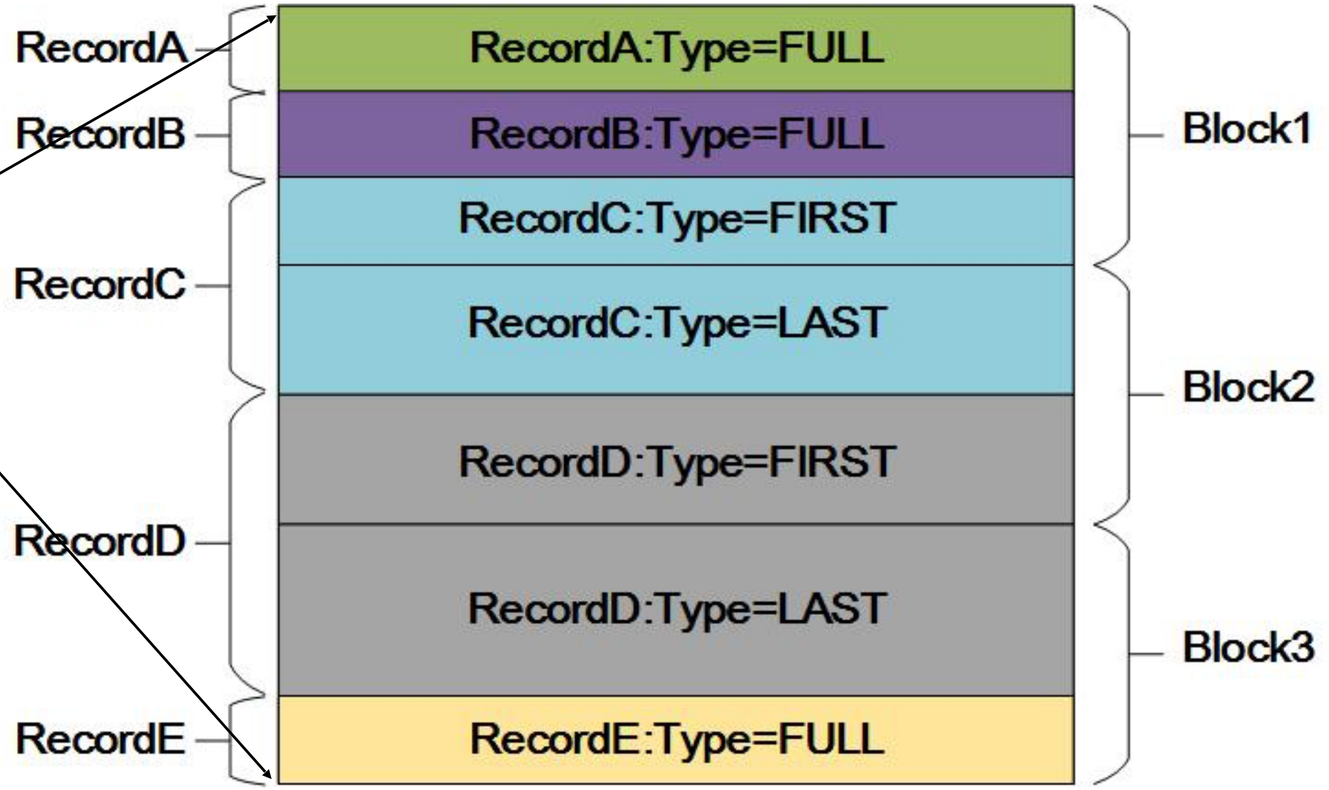
WriteBatch格式

LevelDB——log

LevelDB组件——log文件

- **Log文件：**写Memtable前会先写Log文件，Log通过append的方式顺序写入。Log的存在使得机器宕机导致的内存数据丢失得以恢复
- 作用与格式
  - log文件用来保存最近更新的数据
  - log文件由若干个连续的32k大小的逻辑块组成，块内有一条或多条记录。
  - 记录可能跨块，通过4中type来区分:FULL/FIRST/MIDDLE/LAST
  - binlog和MANIFEST均使用这种log格式
- 何时创建与何时删除?
  - 当一次Immutable memtable完成compaction时会删除旧的log,并生成新的log文件

log文件格式



类型：FULL/FIRST/MIDDLE/LAST

LevelDB组件——log文件的读与写

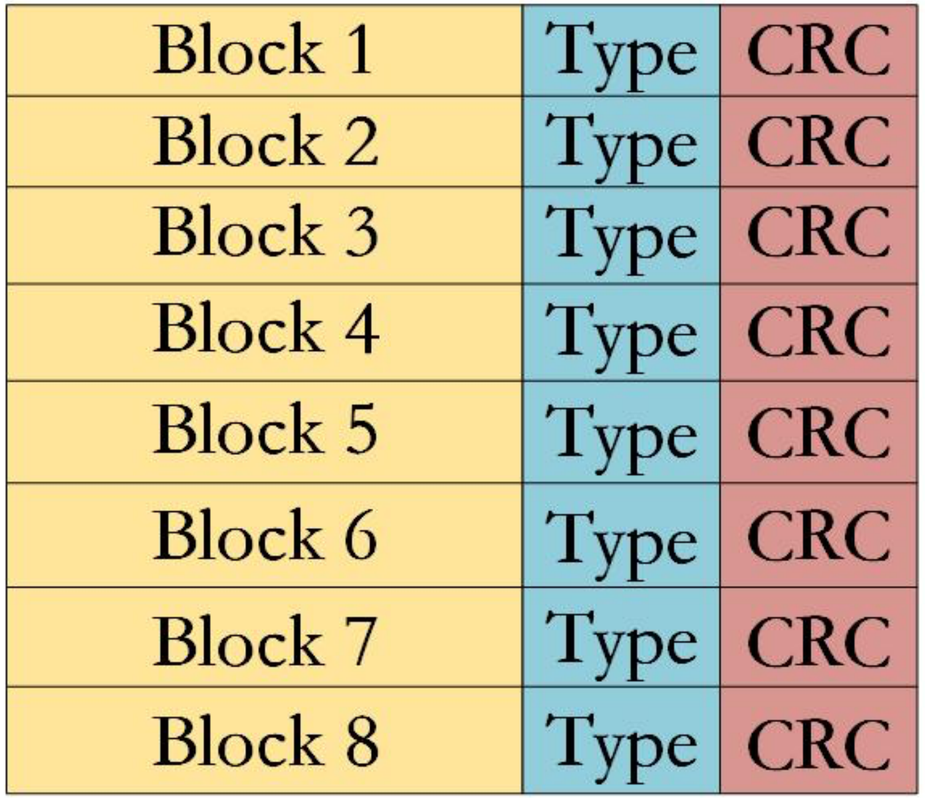
- 写操作
  - 考虑到一致性，没有按照block为单位来写。且写入时无序。
  - 在插入一条记录到memtable之前，会先写入log文件中。
  - 每条记录的格式会先经过WriteBatch将数据序列化。
- 读操作
  - 以block为单位进行读取。
  - 每重新打开DB时，都会将log回放到memtable中，使系统恢复到上次关闭的状态。
  - 按照当初写入的格式反序列化成相应的数据类型。

LevelDB——SSTable

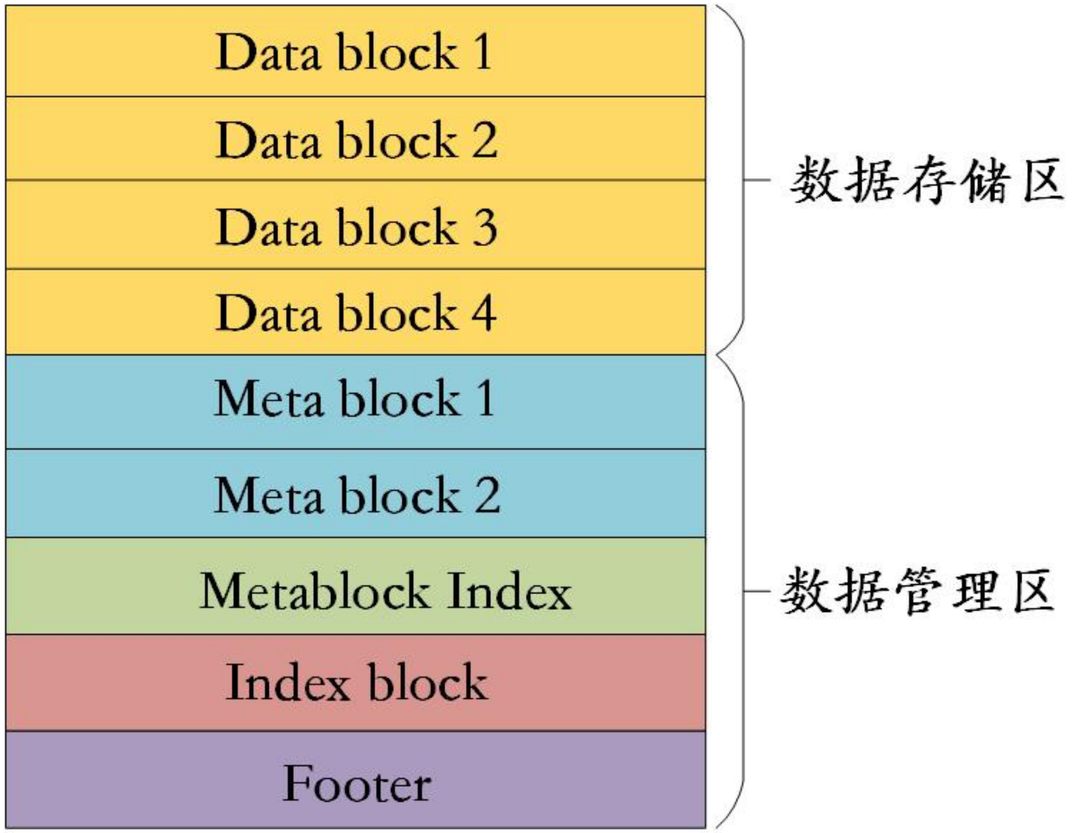
LevelDB组件——SSTable(Sorted String Table)

- sst是由Immutable memtable持久化到磁盘的文件
- sst默认限制2M，所以系统中会存在大量该文件
- sst文件由若干个4K大小的blocks组成
- Level-0：最多4个sst文件
- Level-1：总大小不超过10M
- Level3+:总大小不超过上一个Level*10的大小
- 比如:0->4 sst, 1->10M, 2->100M, 3-> 1G, 4->10G, 5->100G, 6->10T

LevelDB组件——SSTable的格式



sstable中的块格式

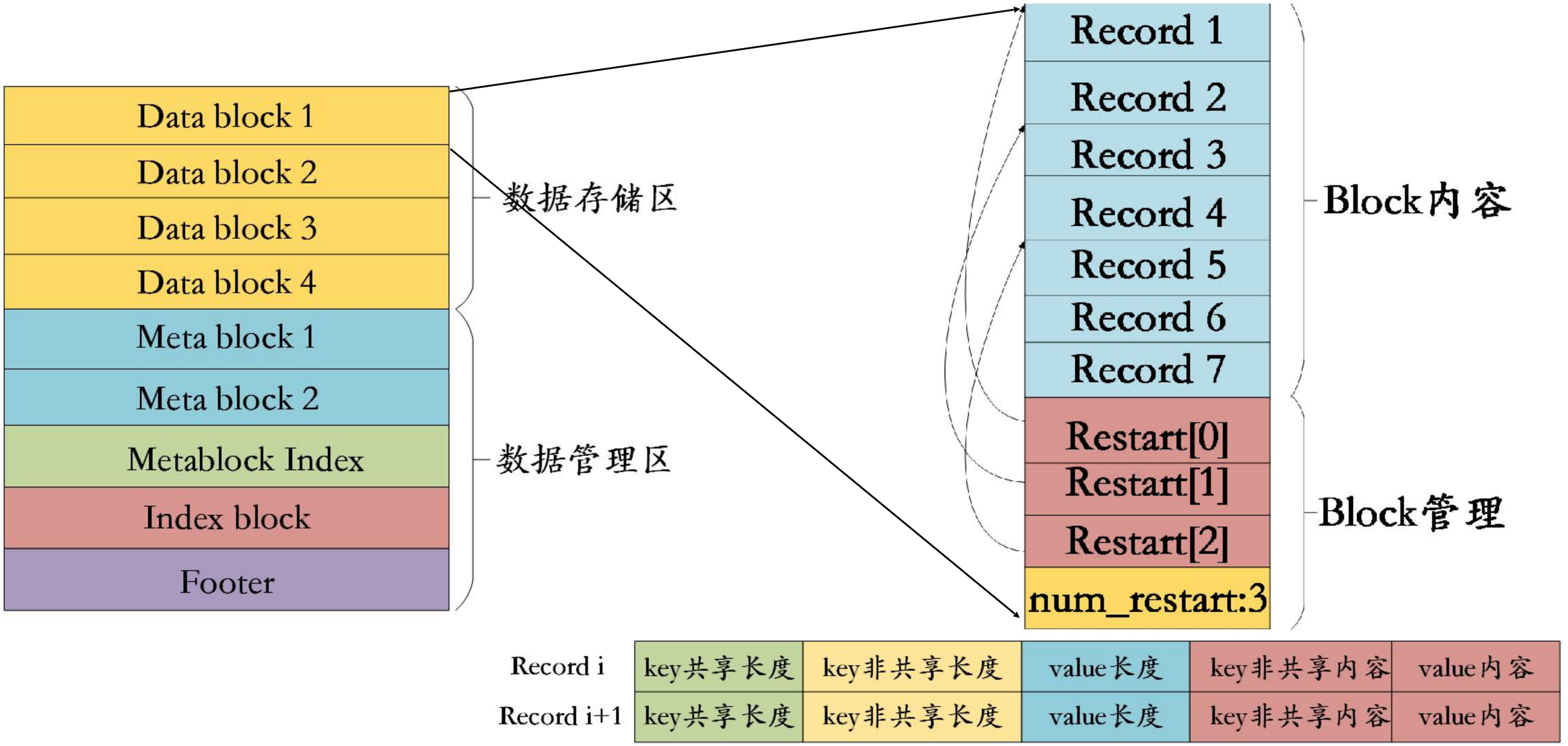


SSTable文件格式

LevelDB——SSTable块

- Data Block:存储实际的K/V数据(InternalKey/Value)
- Meta Block:每个data_block对应一个meta_block中的记录,保存data_block中的key_size/value_size/kv_counts之类的统计信息。目前只有Bloom Fliter过滤器，为了快速定位某个data block是否有该数据。
- Meta Index Block:保存meta_block的索引信息。
- Index Block:每条记录即使每个data block的last_key以及其在sstable中的索引信息。
- Footer：文件末尾固定长度的数据，保存着metaindex_block和index_block的索引信息，为达到固定长度，末尾会填充空白字节。

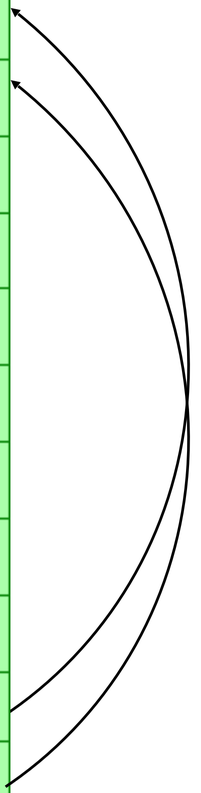
LevelDB组件——SSTable(Data Block格式)



LevelDB组件——Block Data示例

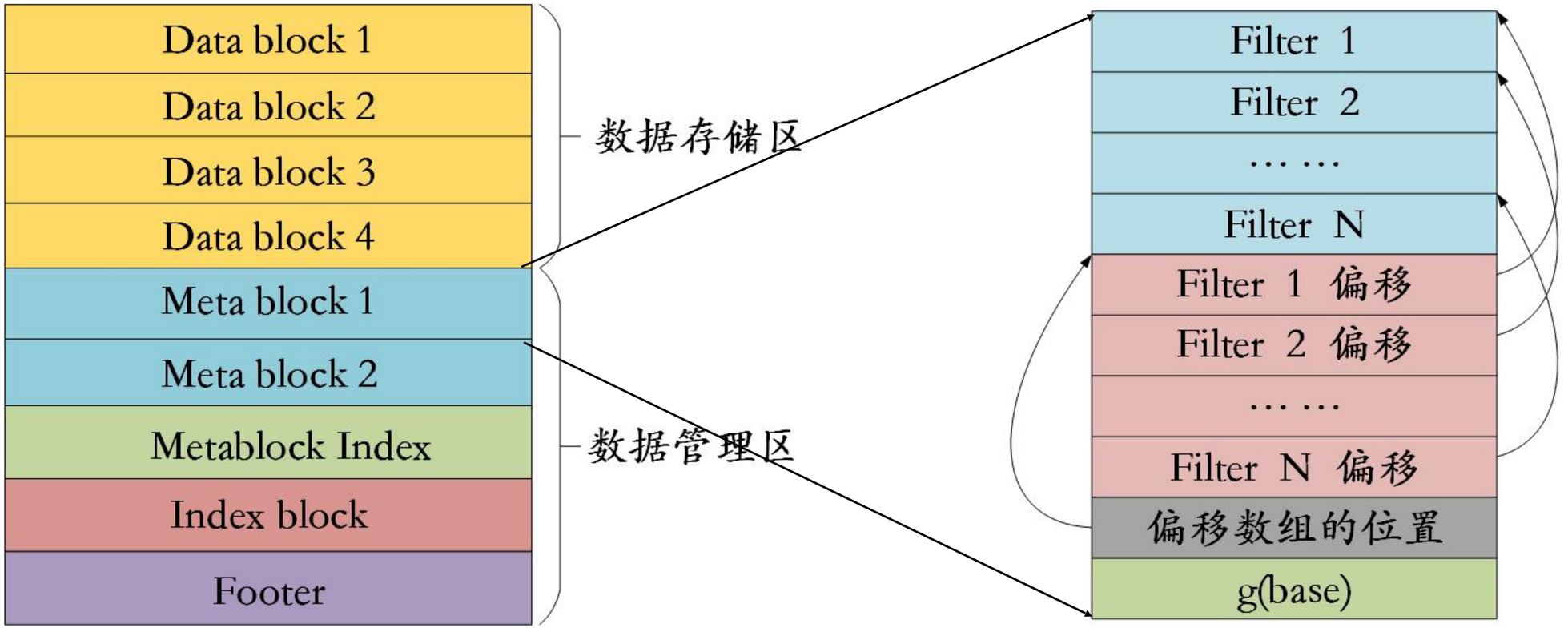
- Block Data利用前缀压缩来节省空间
- 重启点所指向的第一条记录不压缩
- 默认的block_restart_interval是16,即每隔16条记录后就记录一条完整记录

Shared_bytes	Unshared_bytes	Value_bytes	Unshared_key_data	Value_data
0	11	Sizeof(name1)	“51174500101”	“name1”
10	1	Sizeof(name2)	“2”	“name2”
10	1	Sizeof(name3)	“3”	“name3”
		...		
9	2	Sizeof(name10)	“10”	“name10”
		...		
9	2	Sizeof(name16)	“16”	“name16”
0	11	Sizeof(name17)	“51174500117”	“name17”
		...		
Restarts[0]				
Restarts[1]				
...				
Number_restarts				
Trailer(type(1b)+CRC(4b))				

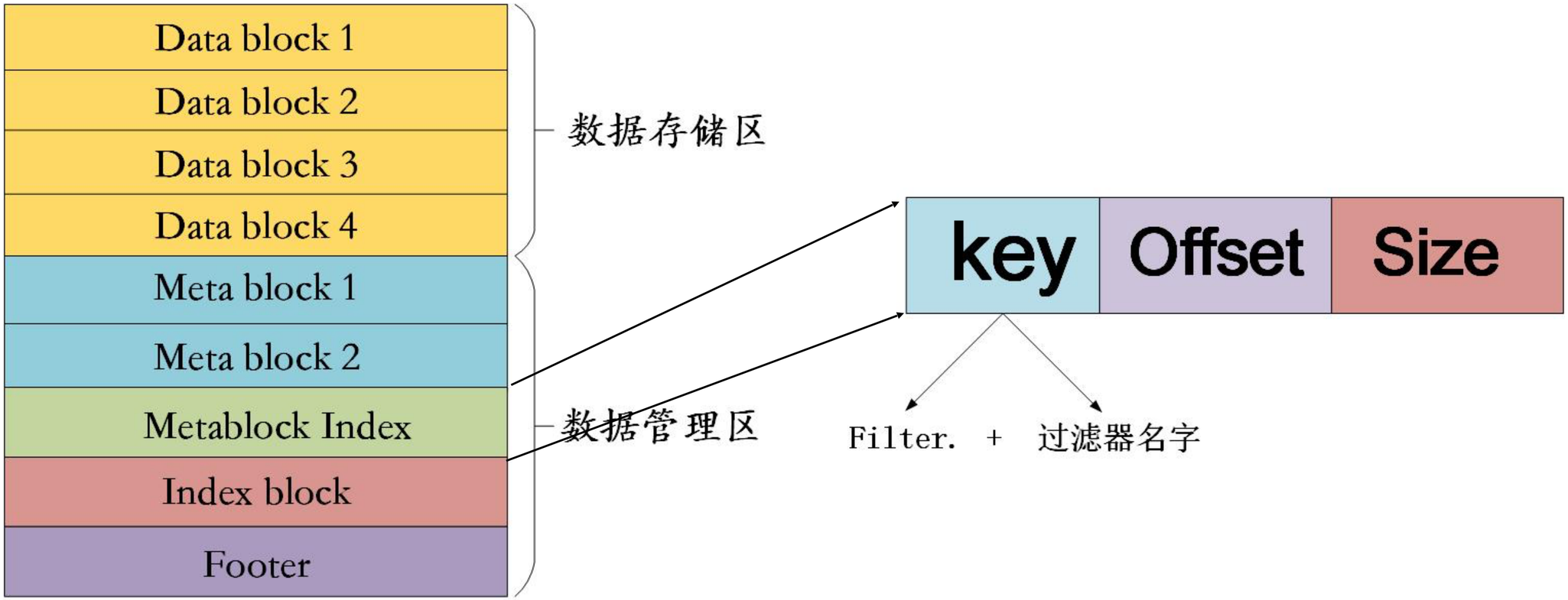


关于学号/姓名(key/value)在sstable中的存储示例

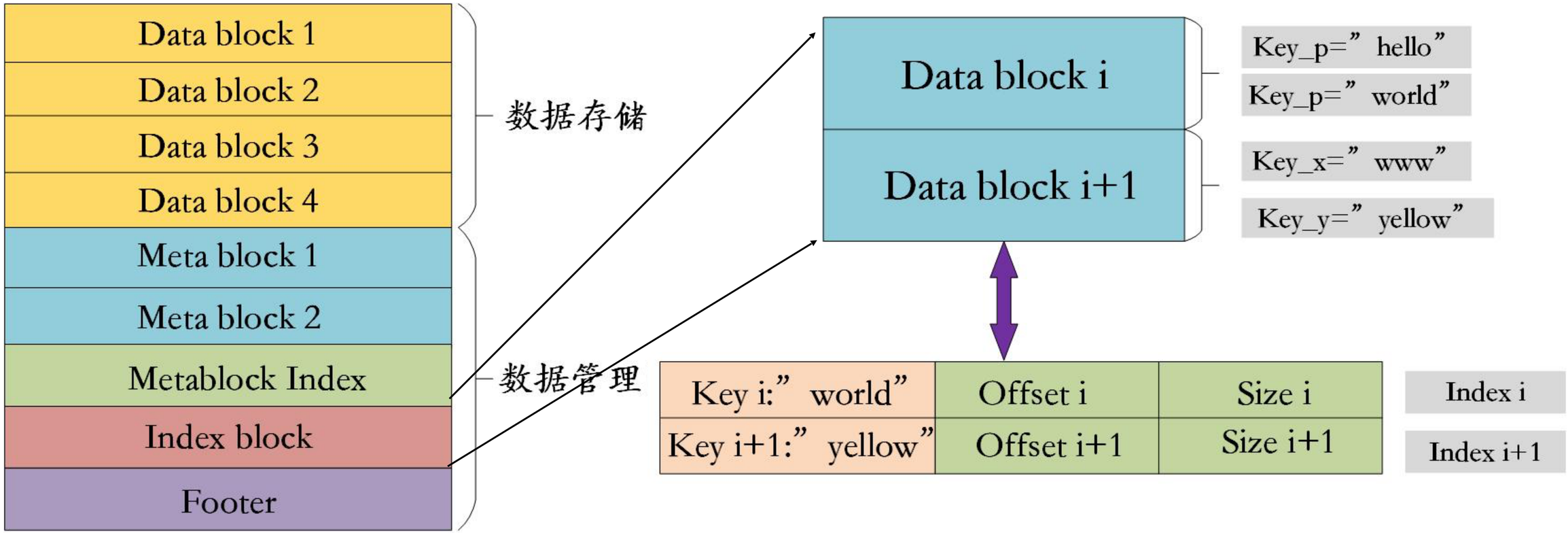
LevelDB组件——Meta block格式



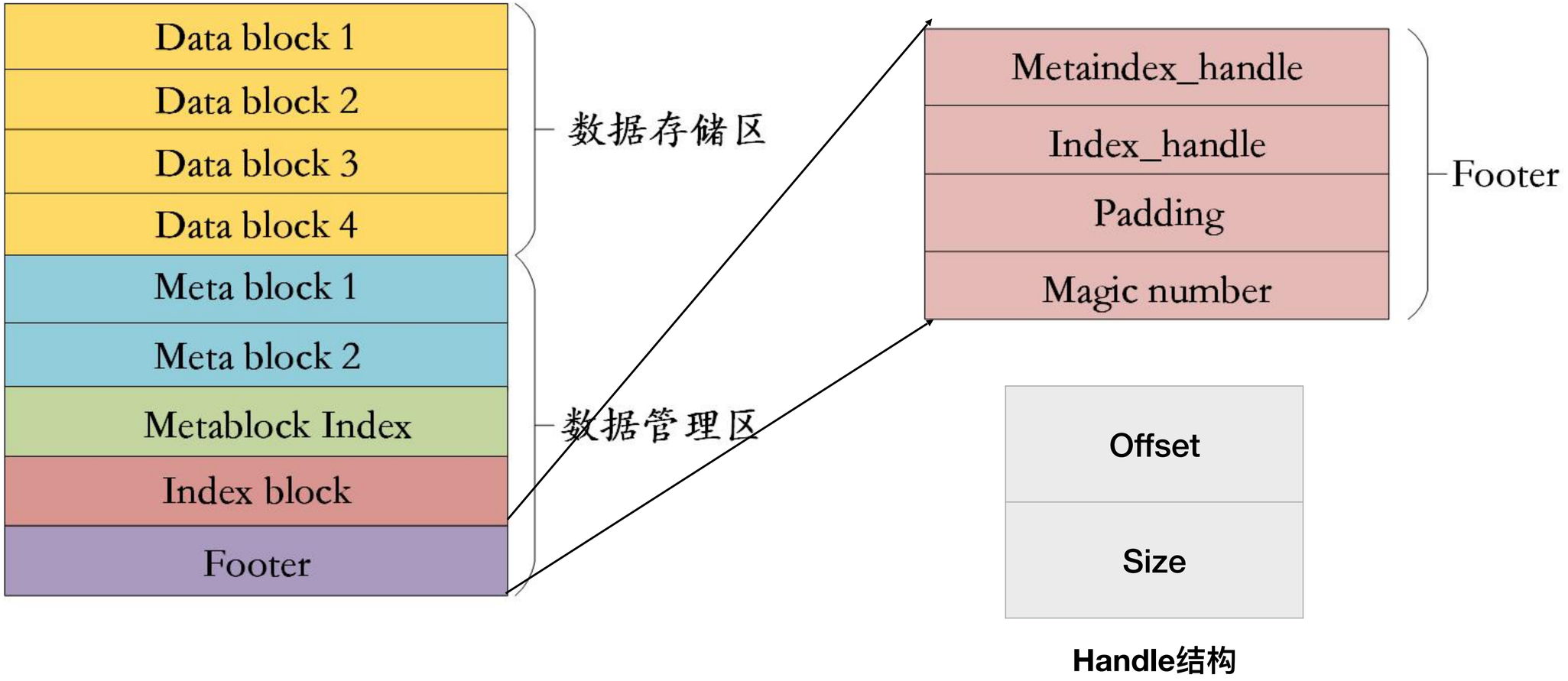
LevelDB组件——Metablock Index格式



SSTable——Index Block



LevelDB组件——Footer格式



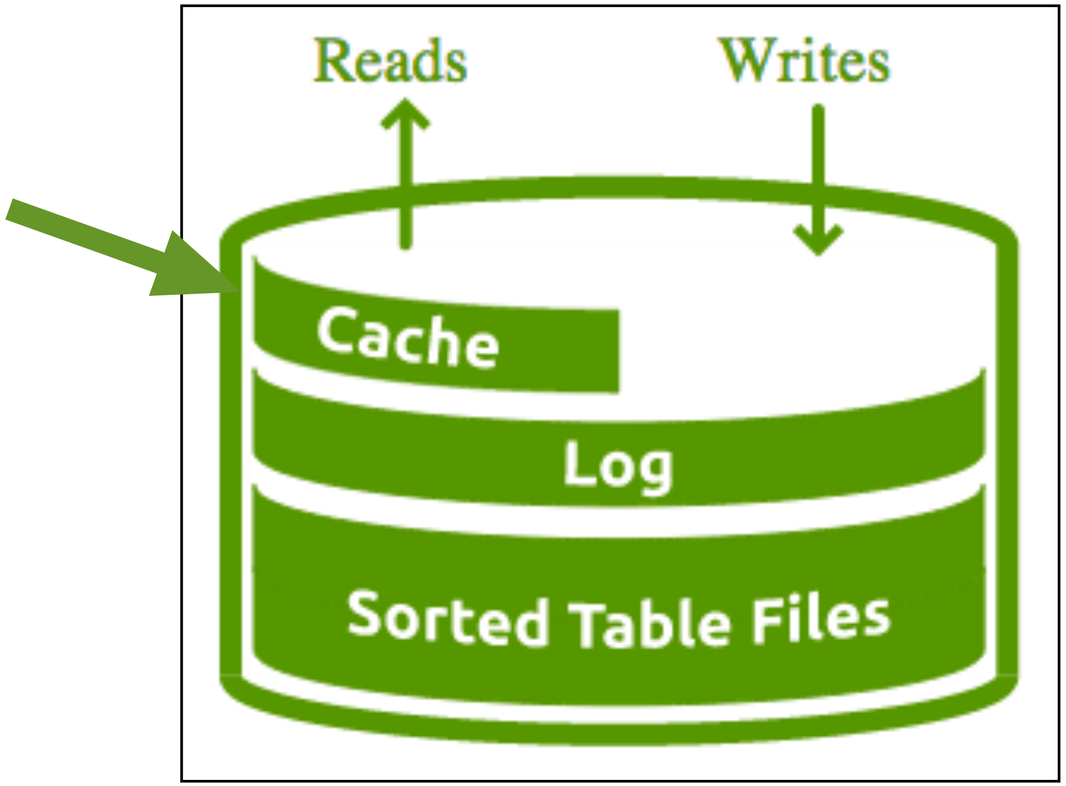
LevelDB组件——SSTable中块内容总结

	Data block	Data Index block	Meta index block
key	实际插入的key	block(i)的最大key<=key_<=block(i+1)最大key	Meta name
value	实际插入的value	Data block(i)的handle	Meta block(i)的handle

LevelDB——Cache

LevelDB组件——Cache

- 两种Cache
  - BlockCache，缓存Block中的数据。通过Option选项来设置。
  - TableCache，缓存sstable的元数据，当从sstable中查找记录时，加快查找速度。



LevelDB组件——TableCache

- TableCache使用双向循环链表和hash表和LRU思想，hash表为了加速查找
- 每创建/打开一个sstable文件就将它放到TableCache中
- 共有16个缓存池,由每个sstable编号的hash值的高四位，分配到缓存池的指定位置
- 对sstable的查找都是调用TableCache::NewIterator()来获得sstable本身的Itereator，sstable中的Iterator会调用具体每个Block的Iterator来查找相应记录。

TableCache中的Hash表

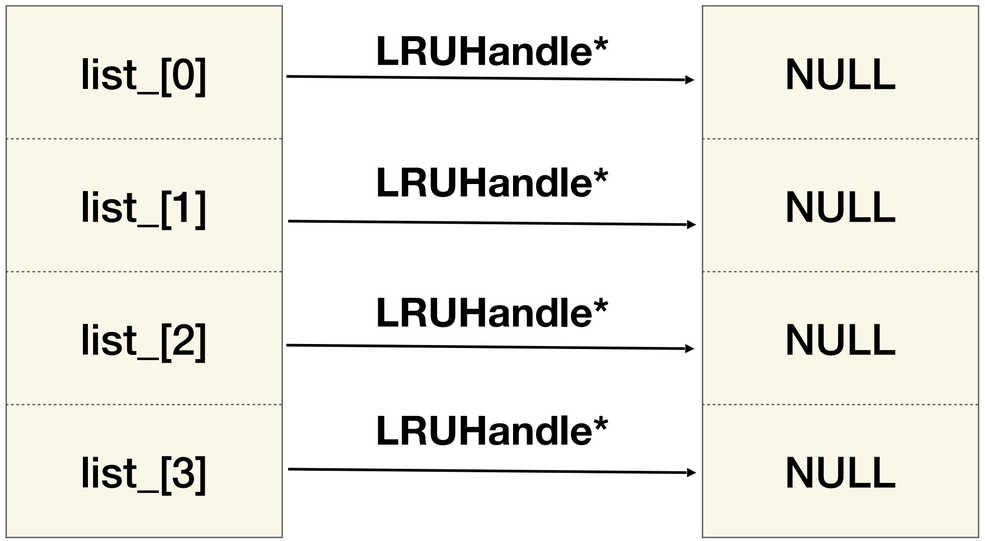
- HandleTable维护一个哈希表。它将hash值相同的所有元素串联成一个双向链表，通过指针next_hash来解决hash碰撞。

- 初始时，length=elems_=0,list=NULL,然后调用resize()，初始分配4个元素，空间不足时，成倍增加。

- Hash表中list_每个元素均指向一个双向循环列表。

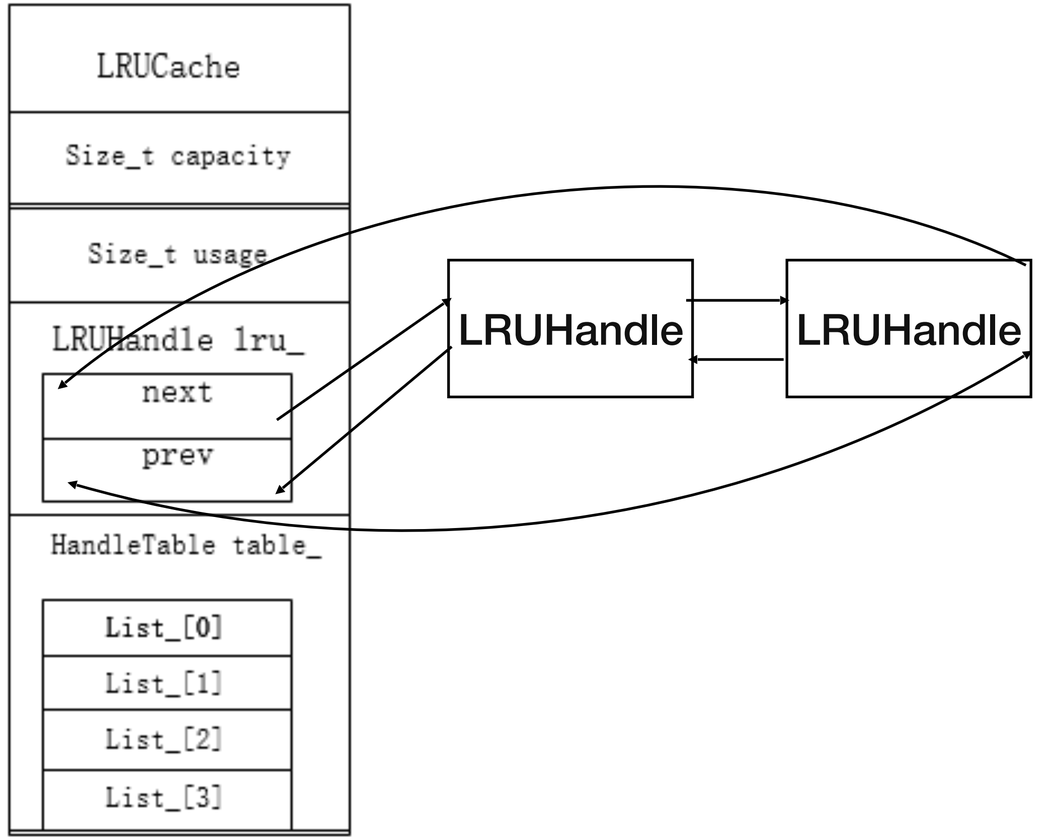
LRUHandle类
LRUHandle* next_hash
LRUHandle* next
LRUHandle* prev
uint32_t hash

HandleTable类
uint32_t length_ //hash表长
uint32_t elems_ //hash表中元素总个数
LRUHandle** list_ //hash表头



TableCache中的LRUCache

- LRUCache是指一个缓存，同时它用到了LRU思想。
- 插入元素时先将它加入到链表尾部，再根据哈希值将其插入到Hash表中。通过[hash&(length-1)]来确定在hash表中的下标。
- 若Hash表中已存在和要插入的hash值时，将原有元素从链表中移除。

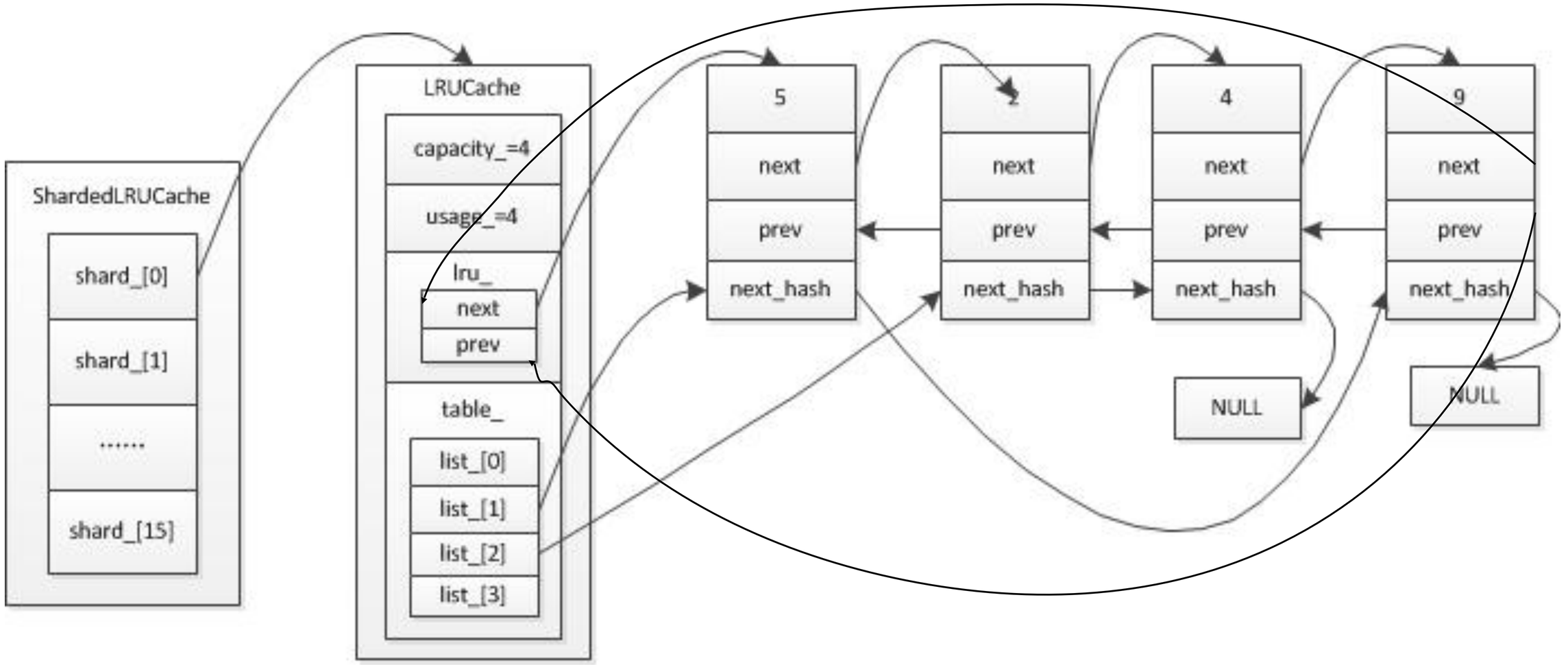


TableCache中的ShardedLRUCache

- 每一个LRUCache是线程安全的，故为了多线程访问，减少锁开销，ShardedLRUCache内部将所有Cache根据hash值的高4位分成16份，即有16个LRUCache分片。

- 查找key时先计算key属于哪一个分片(取32位hash值的高四位)，找到对应的LRUCache分配，然后再相应的LRUCache中查找。

LevelDB组件——TableCache示意图



LevelDB组件——TableCache总结

- 首先根据hash值的高4位，将所有的Cache分成16份，每一份一个LRUCache。
- LRUCache维护所有元素（LRUHandle）,通过锁来保证线程安全，同时当前Cache的最大容量和已用内存。
- 每一个LRUCache中有一个循环链表，和一个hash表，通过循环链表来实现LRU思想，hash表来加速查找。

LeveIDB——版本控制

LevelDB组件——Version的作用

- Version用来保存某一版本所有层的sstable的元信息
- Version还需要检测哪些层、哪些文件需要执行合并操作。
- Version记录下一个用来合并的文件指针，对于每一层，记录上一次compaction的ending_key。
- DB中会存在多个Version的情况（何时会出现？），它们通过双向循环链表连接起来，当某个Version的引用计数为0且不是最新Version时，会从链表中移除，且该Version内的sstable也可以被删除了。

LevelDB组件——Version类图

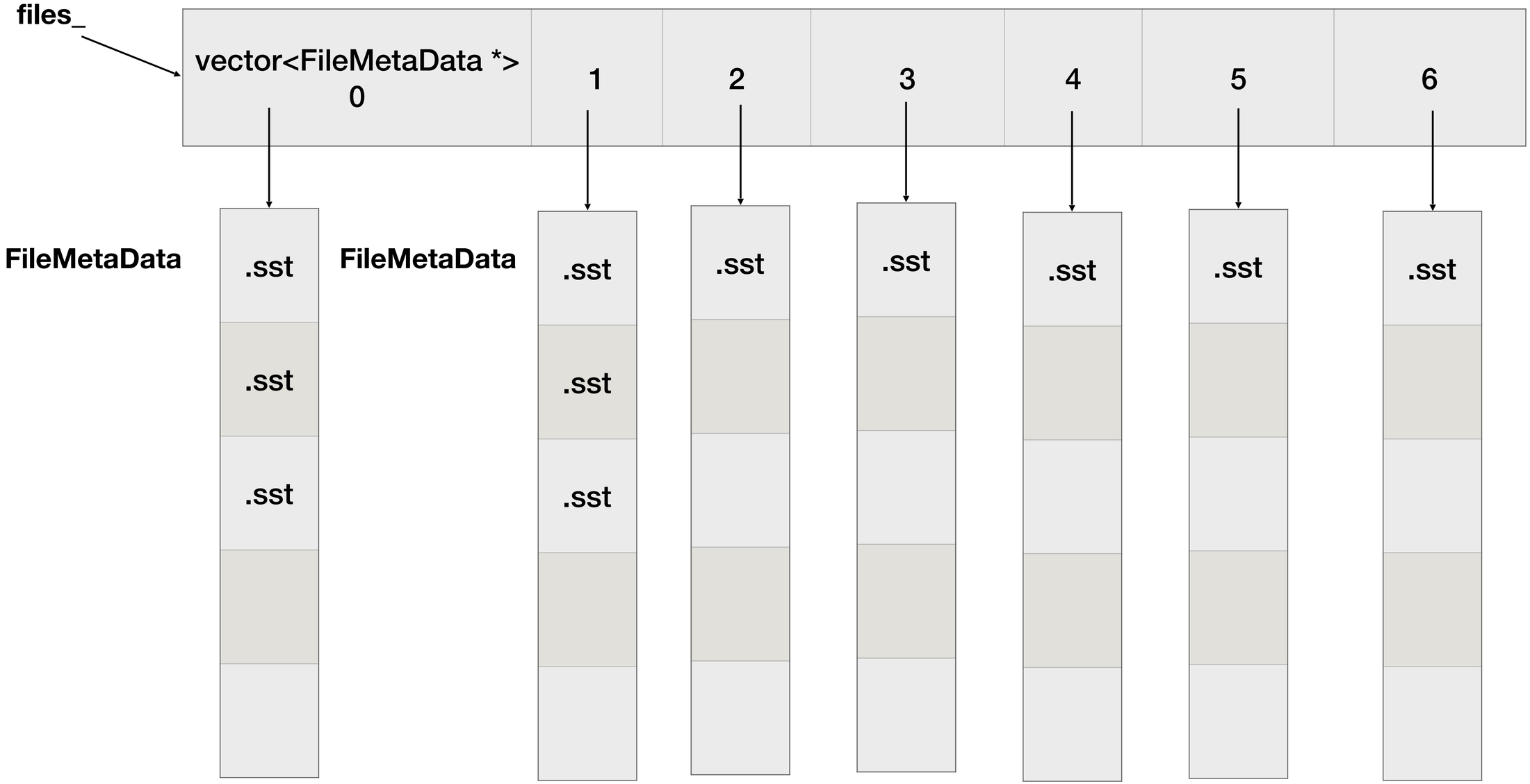
- LeveldDB写与sstable无关，而Get()操作则和各level种的sstable的数量，大小，以及range分布会直接影响读的效率。

- Level的均衡状态定义:level-0最多只有一个sstable,level-1以及以上的各level种key-range分布均匀，期望更多的查找可以遍历更少的level即可定位到。

Version类
VersionSet* vset_
Version* next_
Version* prev
int refs_
Std::vector<FileMetaData*> files_[config::kNumLevels]
FileMetaData* file_to_compact_; Int file_to_compact_level
Double compaction_score Int compaction_level_

Version类主要成员

LevelDB组件分析——Version示意图



Version量化Level是否均衡的策略（一）

- 量化level的不均衡比重，有两种量化方法
- compaction_score（平衡条件为compaction_score < 1，否则触发合并）
  - Level-0:每个sstable有overlap的可能，且由memtable直接dump而来，故不受kTargetFileSize的限制。所以sstable的count更有意义。计算方法为compaction_score=count(sstable)/KL0_CompactionTrigger
  - Level-1+:每个sstable大小固定，故限制该层的总大小,计算方法为：compaction_score=size(sstable)/(quota_size)
    quota_size=kBaseLevelSize*10^(level_num-1)

Version量化Level是否均衡的策略（二）

- file_to_compact_（对单个sstable文件的IO做了更细致的优化）
  - 一次查找如果对过于一个sstable进行了查找，说明处于低level上的sstable并没有提供高命中率。可以认为它处在不最优的状态，我们认为compaction后会倾向于均衡的状态，故在一个sstable的seek次数达到一定阈值后，主动对其进行compaction。
  - seek次数的阈值(FileMeatData中的allowed_seeks)由SAS磁盘读IO确定。保守设置allowed_seeks=sstable_size(2M)/16k=128
  - 每次Get操作时,若有超过一个sstable文件进行了查找，则将第一个进行查找的sstable的allowed_seeks减一，并检查是否用光。

LevelDB组件——VersionEdit

- VersionEdit用来记录每一次版本内所有操作的更改变化量（FileNumber增加，删除参与合并的两层level的sstable，增加的新的sstable.....）
- 记录上一个版本要在哪些层删除的文件，以及将在哪些层生成新的sstable文件。
- 每次compaction后，将version_edit应用到current_版本中，并生成新版本，且将version_edit encode写入MANIFEST文件中。

LevelDB组件——VersionEdit类图

- Compact_pointers_：要更新level的compact_pointer。<层数，start_key>
- deleted_files_:要删除的sstable<层数，文件编号>
- new_files_:新添加的sstable<层数，文件元信息>

VersionEdit类
Std::string comparator_
uint64_t log_number_
uint64_t log_number_
uint64_t prev_log_number_
uint64_t next_file_number_
SequenceNumber last_sequence
Bool has_comparator_
bool has_log_number_
Bool has_prev_log_number_
bool has_next_file_number_
Bool has_last_sequence_
Std::vector<std::pair<int,InternalKey>> compact_pointers_
Std::set<std::pair<int,uint64_t>> deleted_files_
Std::vector<std::pair<int,FileMetaData>> new_files_

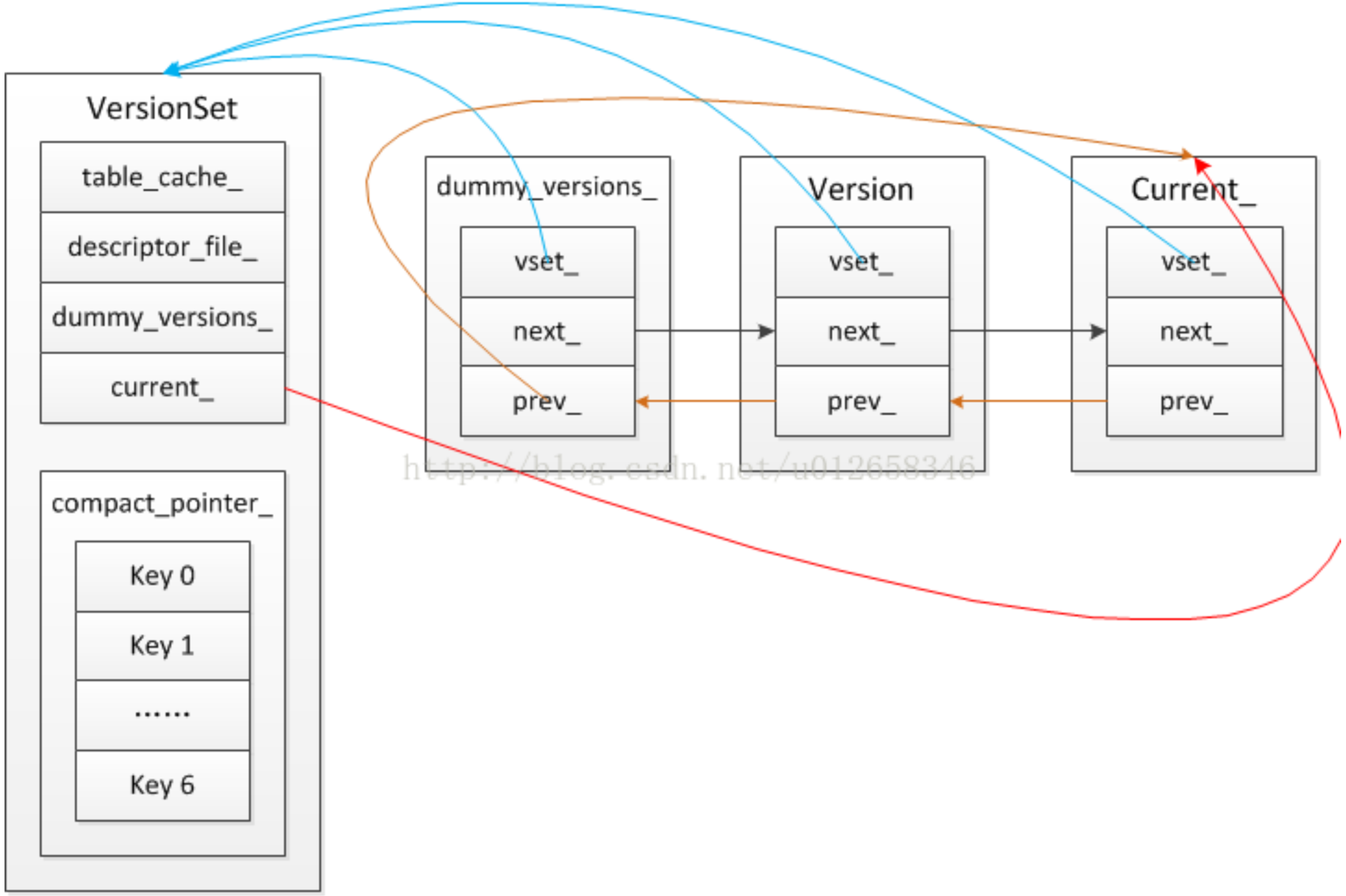
LevelDB组件——VersionSet

- VersionSet管理整个DB的当前状态
- compact_pointer_[config::kNumLevels]用来保存每个level下一次compact的start-key。(这个变量为什么不放在Version类中？)

VersionSet类
Const std::string dbname_
Const Options* const options_
TableCache* const table_cache_
Uint64_t next_file_number
uint64_t manifest_file_number
uint64_t last_sequence_
Unit64_t log_number_
Unit64_t prev_log_number_
WritableFile* descriptor_file_
Log::Writer* desciptor_log_
Version dummy_versions_
Version* current_
Std::string compact_pointer_[config::kNumLevels]

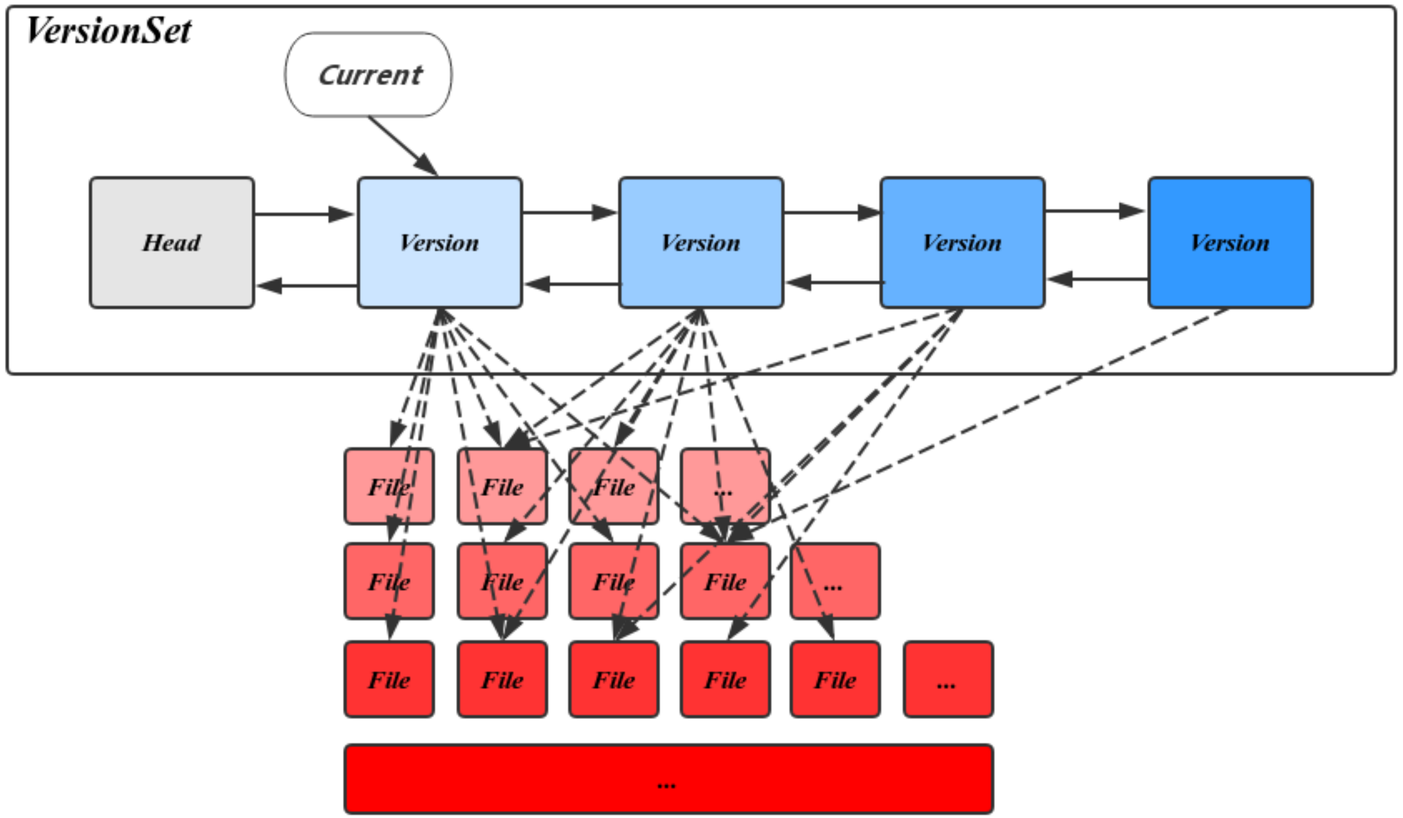
VersionSet类主要成员

LevelDB组件——VersionSet示意图



LevelDB组件——VersionSet与Version的联系

- 不同version之间仅仅是一些文件在一个level中被添加，而在另一个level中被删除。
- version之间的变化是通过不断的在current_版本中应用version_edit来生成更新的版本。



LevelDB——MANIFEST

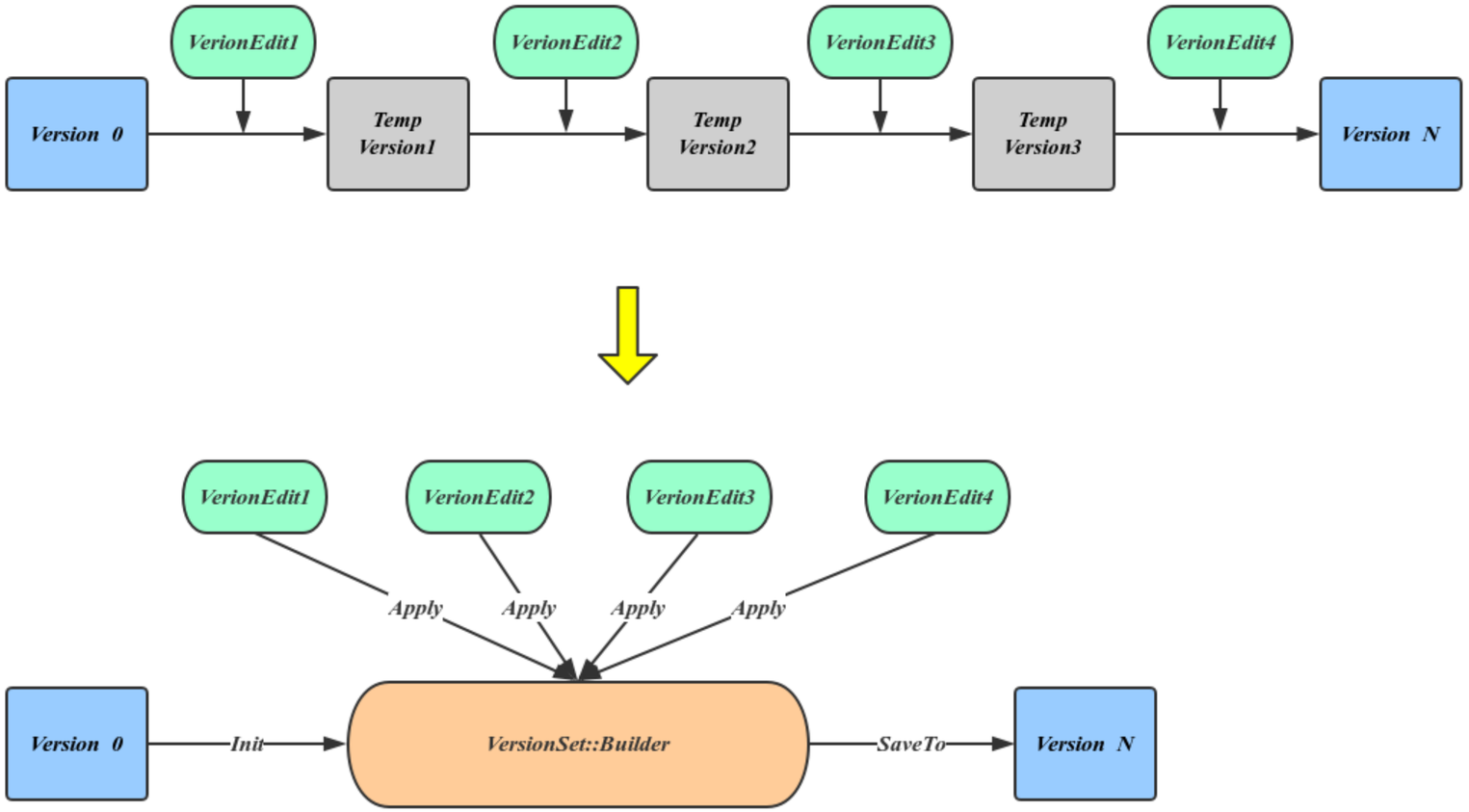
LevelDB组件——MANIFEST文件

- **Manifest文件：** Manifest文件记录上一个DB被打开之后的所有操作的集合，也即保存了系统所有的元信息。
- 作用:
  - MANIFEST不仅保存当前状态，还会保存所有的历史状态
  - 通过回放MANIFEST文件来使DB回到上一次关闭之前的状态(FileNumber, SequenceNumber, 各level的文件数目，size，range，compact_pointers_等信息)
- 每次状态的完全保存需要非常多的空间和时间，有什么改进方法?

MANIFEST文件的回放过程（一）

- 解决方案:
  - 只在MANIFEST开始(新建时)保存完整的状态(VersionSet::WriteSnapshot())，接下来只保存每次compaction的结果(VersionEdit),重启DB时，根据开头的起始状态，依次将后续的VersionEdit回放，即可恢复到退出前的状态(Version)
- 如何回放:
  - 恢复元信息即依次应用MANIFEST中的VersionEdit记录，这个过程中有大量的中间Version产生，。LevelDB引入VersionSet::Builder来避免这种中间变量，方法是先将所有的VersoinEdit内容整理到VersionBuilder中，然后一次应用产生最终的Version，这种实现上的优化如下图所示：

MANIFEST文件的回放过程（二）



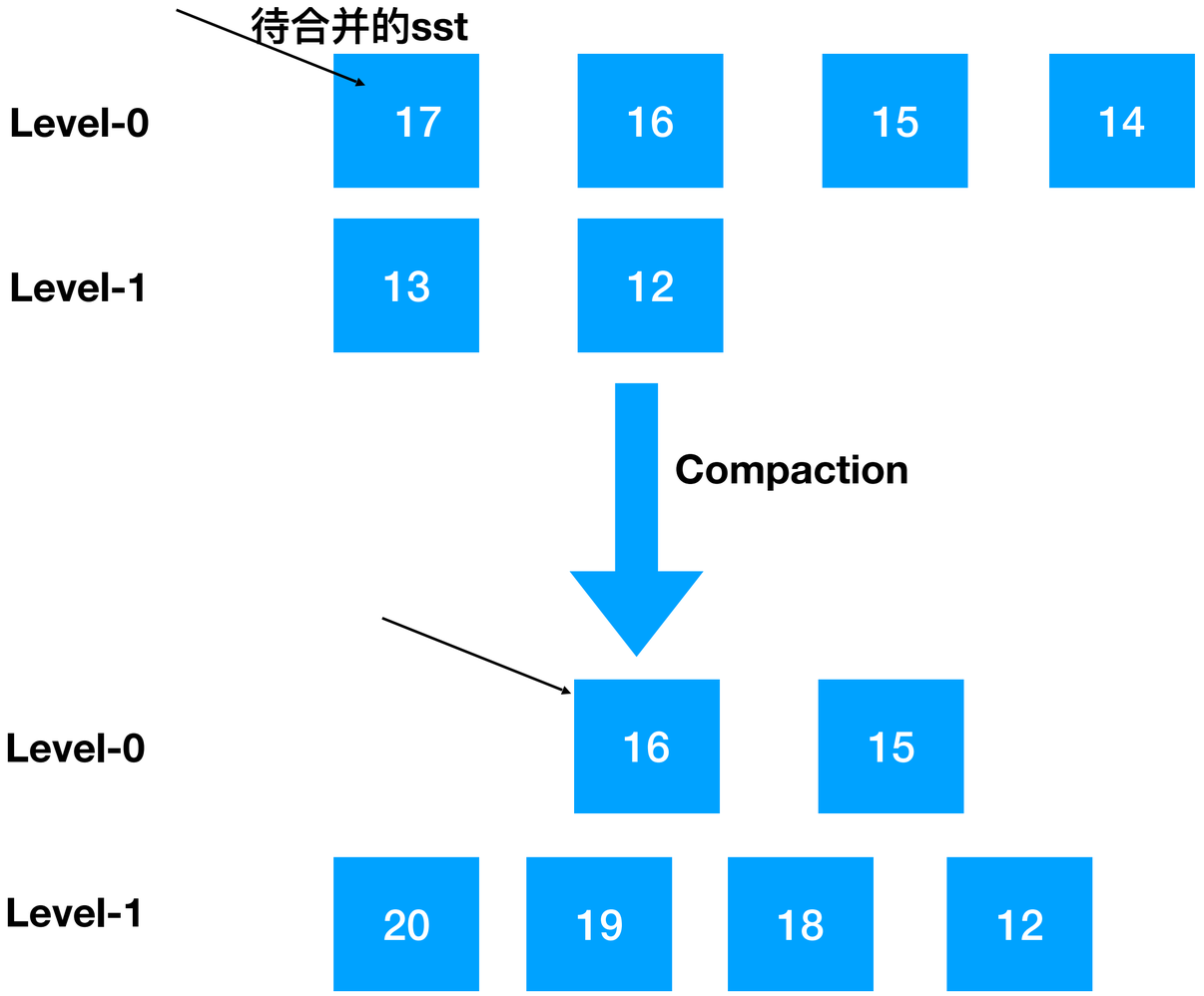
LevelDB——Compaction

LevelDB的Compaction操作

- LevelDB为什么需要Compaction?
- LevelDB中Compaction的分类：
  - Minor compaction(Memtable->sstable)
  - Major compaction(Level-n->level->n+1)
    - Trivial Compaction:直接将level-n的一个文件移入到level-n+1
    - DoCompactionWork:做真正的sstable合并操作
- 触发Compaction的原因：
  - 容量触发Compaction
  - Seek触发Compaction
  - 手动Compaction
- 合并规则:
  - 将Level-n的文件f和Level-n+1中所有和文件f有交叉的文件进行多路归并排序，新生成的文件添加到Level-n+1层

LevelDB组件——Compaction过程

- 若Level-0的17和14号sstable有重复，且和上层的13号sstable有文件，则将这三个文件的元信息加入到inputs_[2]的删除队列中。然后调用合并函数，将会生成新的的sstable。并将这些变化写入version_edit中，然后将version_edit应用到CURRENT版本，生成新版本并添加到version_set链表中。至此，合并操作完成。



LevelDB——接口流程

LevelDB主要流程——Open操作

1. 基本检查

   - 根据传入的DB路径，判断是否已经有DB实例启动。
   - 根据Option内的create_if_missing/error_if_exists,来确定数据目录已存在时要做的处理。

2. DB元信息检查

   - 从CURRENT文件中读取当前的MANIFEST文件并回放其中的每条记录
   - 检查解析MANIFEST的最终状态的基本信息是否完整。（log number, FileNumber, SequenceNumber）,将其生效成DB当前状态。此时，整个DB的各种元信息
   - 此时，DB恢复成上一次退出前的状态。

LevelDB主要流程——Open操作

3. 从log中恢复上一次可能丢失的数据

- 遍历DB中的文件，根据已经过得的DB元信息LogNumber和PrevLogNumber，找到上一次未处理的log文件。
- 遍历log文件中的record(Put时WriteBatch写入的)，重建memtable，达到memtable阈值，就dump到sstable，期间用record中的SequenceNumber修正从MANIFEST中读取的当前SequenceNumber。
- 将最后的memtable dump到sstable。（会触发Compaction）
- 根据log文件的FileNumber来修正当前的FileNumber。

4. 生成新的log文件。更新DB的元信息(VersionSet::LogAndApply().生成最新的MANIFEST文件)，删除无用文件，尝试compact

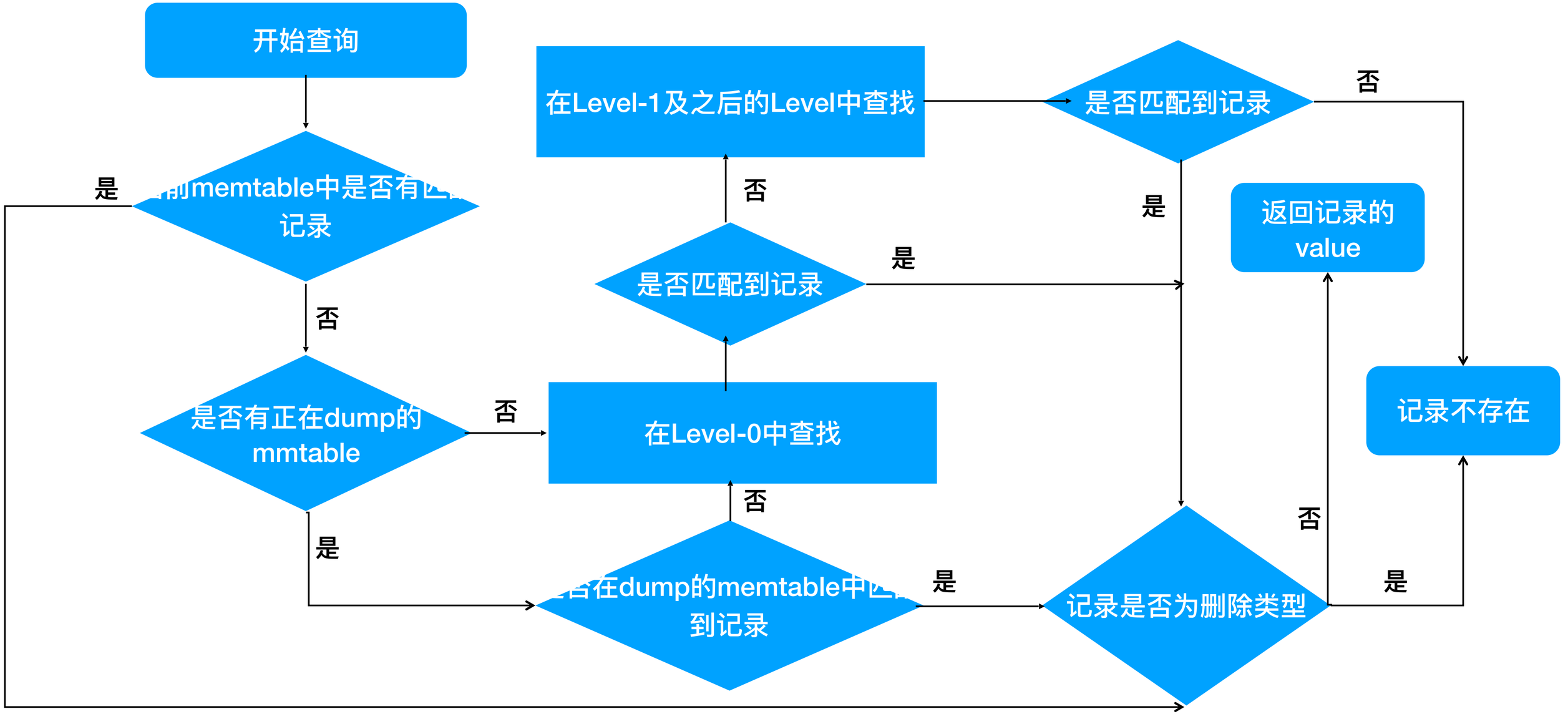
LevelDB主要流程——Put操作

- DBImpl::Put()调用DB::Put()方法，后者的方法中调用DBImpl::Write();
- 将key/value封装成WriteBatch
- DBImpl::Write()先确保memtable和log文件有空间用于数据的写入(DBImpl::MakeRoomForWrite())，对已满的memtable后台调度Compaction
- 将WriteBatch中的数据添加到日志(每条记录:7bytes(记录首部)+12bytes(batch首部)+1byte(数据类型)+key.size(varint32)+key+value.size(varint32)+value)
- 将WriteBatch添加到memtable(检查是否达到阈值，可能会触发compaction)

LevelDB主要流程——Get操作

- DBImpl::Get(options, Slice &key, Slice *value)操作用来从DB中找到指定的key,并且将值存储在value中。
- 查找顺序:
  - 首先在memtable中查找
  - 其次在immetable memtable中查找
  - 若还没找到，则到sstable中进行查找(有可能会出发compaction)

LevelDB——Get操作流程图



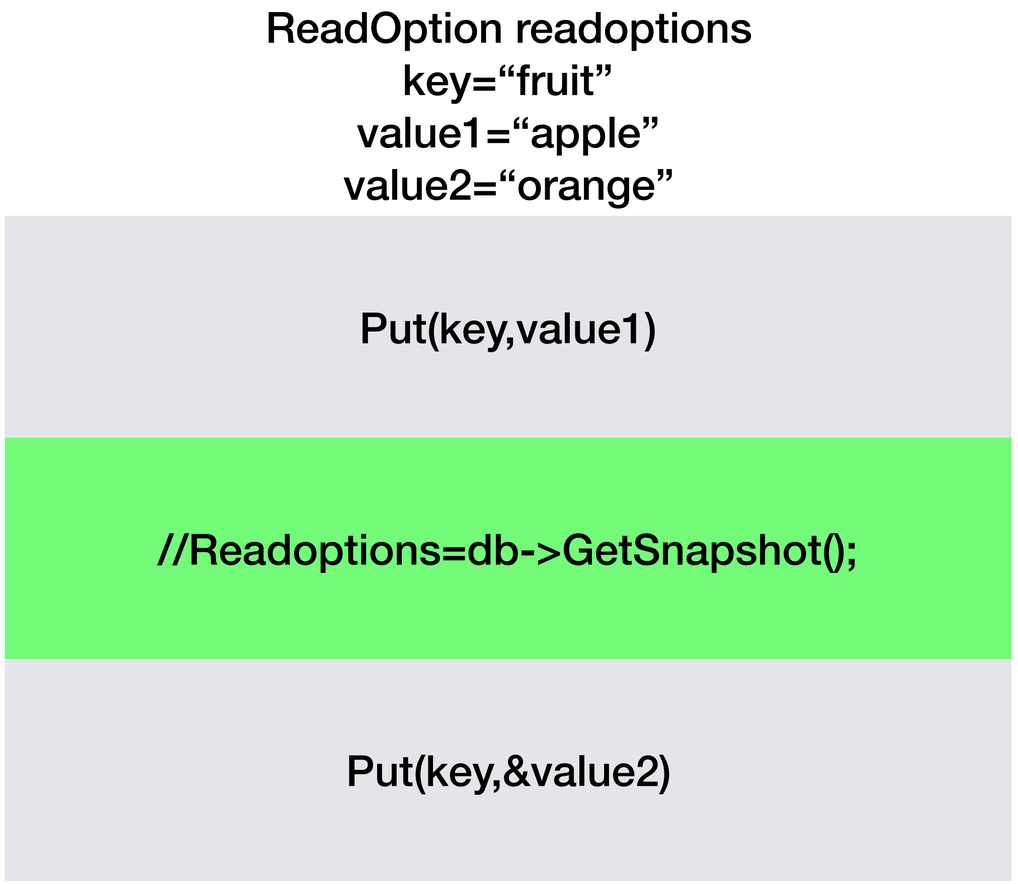
LevelDB主要流程——Delete操作

Delete操作相比于Put操作，只是在构造WriteBatch时，设置ValueType为kTypeDeletion,其他流程和Put完全相同

LevelDB——其他及优化细节

LevelDB组件——Snapshot

- leveldb的快照主要用来读取某个时间点之前的数据，DB中可能存在key/value相同的数据，查找时系统只返回最新数据。
- 快照的本质就是保存某一时刻最大的sequence_number。当查找时以该序列号为查找基准，只查找小于该序列号的记录。
- 快照会碰到什么问题?



LevelDB——其他

- 垃圾回收(GC)

每次compaction和recovery之后都会有文件被废弃，成为垃圾文件。GC就是删除这些文件的，它在每次compaction和recovery完成之后被调用。

- 迭代器(Iterator)的设计非常巧妙，从底层迭代开始，一层层往上封装。迭代器最大的好处是方便了底层数据的检索，屏蔽实现细节。
  - SkipList迭代器->Memtable迭代器
  - Block迭代器/文件迭代器->SSTable双层迭代器->TableCache迭代器->Version迭代器->MergeIterator->DBIterator

LevelDB——优化细节(一)

- 可以减少数据量以及IO的细节:
  - 对key进行前缀压缩
  - 内部存储的key只有user_key,sequence_number和type
  - sstable元信息以及block数据都有cache
  - log文件/MANIFEST文件采用相同的存储格式，都以log::block为单位。

LevelDB——优化细节(二)

- 可以加速key定位的细节:
  - Memtable使用SkipList，提供O(log n)的查找复杂度
  - 分level管理sstable,对于非level-0,最多查找一个sstable
  - sstable中分block管理，且block的key范围有序，一个key仅在一个block中，真正的IO只有一次。（若不在Cache中，则会调用Table::Open()先打开sstable,再将索引数据加入TableCache中。）
  - Block的元信息(Index Block)和Block中标志每个前缀压缩区间开始offset的retarts集合都可以用二分查找来加速定位。

LevelDB——优化细节(三)

- 均衡读写效率的细节
  - level-0上sstable数量的阈值来主动限制写速度。避免过多level-0 sstable文件影响读效率。
  - 避免生成过多和level-n+2 相覆盖的level-n+1的sstable
  - 控制每个level的平衡状态(通过compaction_score和file_to_compact)，后者可主动compaction，以避免坏情况的发生

LevelDB——优化细节(四)

- 其他的优雅封装
  - SequenceNumber和FileNumber解决了数据的时间点
  - ValueType将数据更新(Put/Delete)统一处理逻辑
  - sstable格式中数据区和索引区使用同样的Block格式,统一处理逻辑
  - 将对当前DB状态的修改封装成VersionEdit，一次Apply
  - log格式，以Block为单位，IO友好，block内分record，利于解析
  - MANIFEST文件中只保存一次全量状态，后续仅保存每次的修改，减少IO

LevelDB——总结

LevelDB——总结

- LevelDB是Jeff Dean大神之作，代码质量高，结构清晰
- 非常推荐去阅读优秀的开源项目源码。从源码中学习
- 某一个实现可能需要循环反复的看，每次看都会有所收获

谢谢！ Q&A