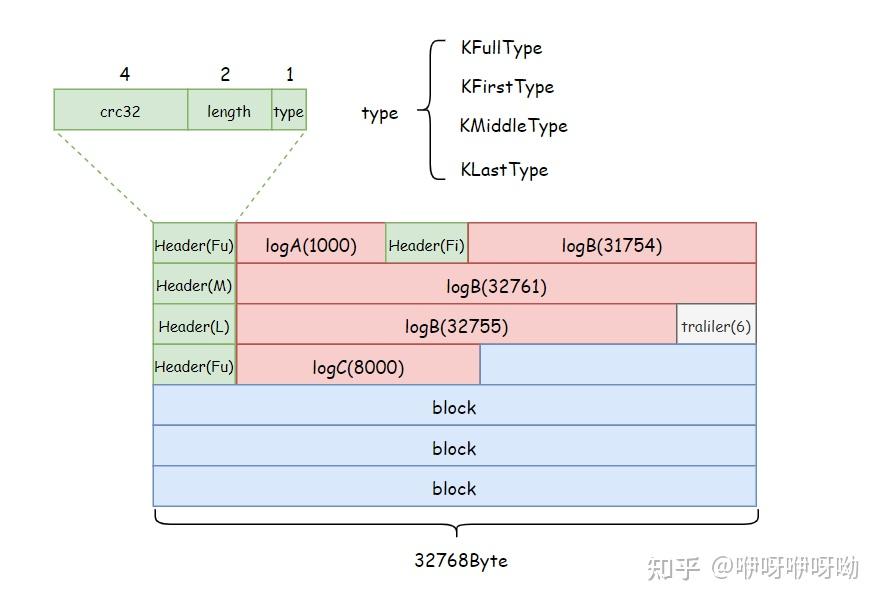
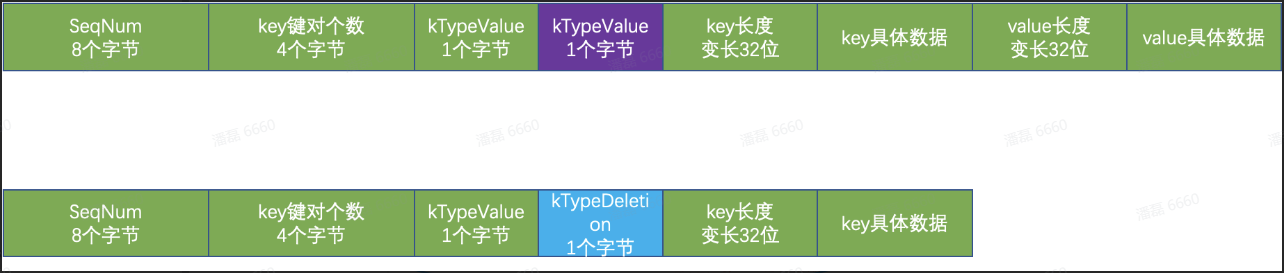
**1、WAL Log：**

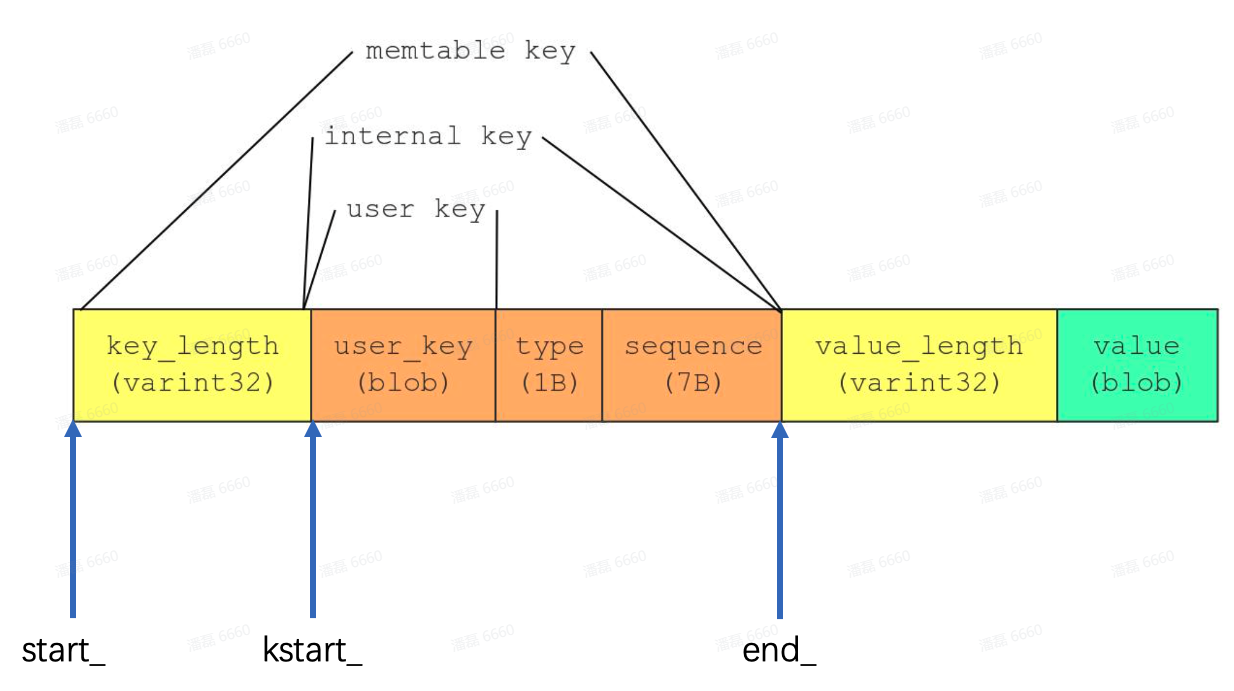
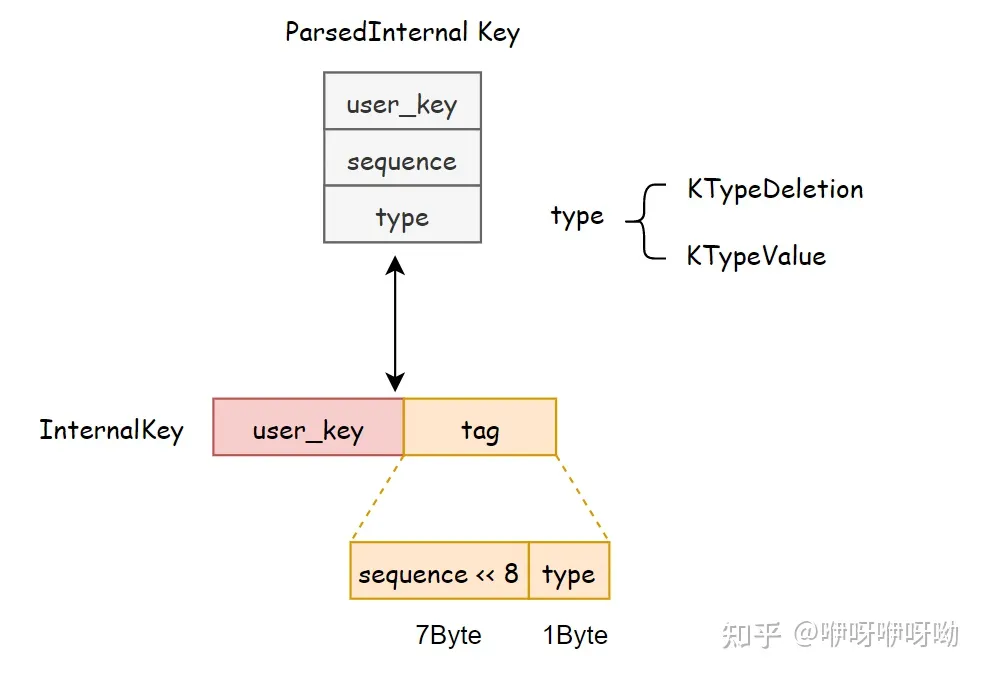


一条log的具体内容，即Header对应的Record，下图中的kTypeValue有两种类型，分别是kTypeValue和kTypeDelete。



VersionSet中记录的当前日志序列号和上一个日志序列号分别指的是mem\_和imm\_的，一个是正在写入的Memtable，另一个是已经写入完成的Memtable，是不可修改的，内存中mem\_和imm\_都只有一个，详细内容查看Write()函数解析。

1. **Memtable（skiplist）：**

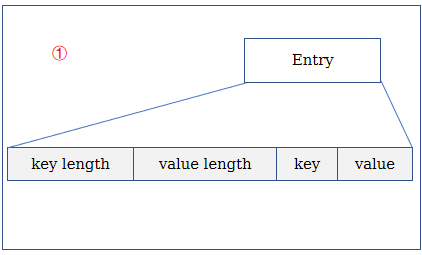
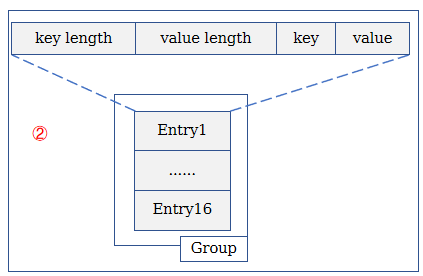
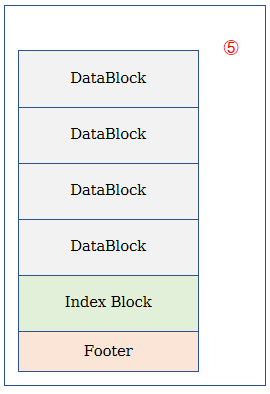


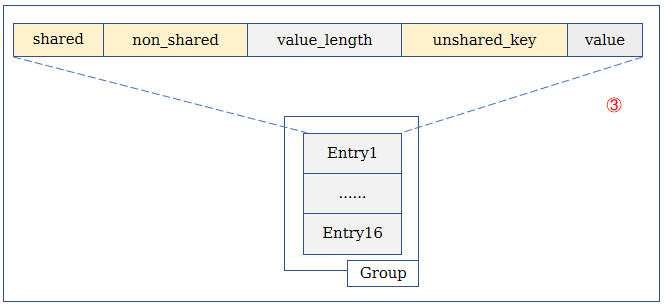
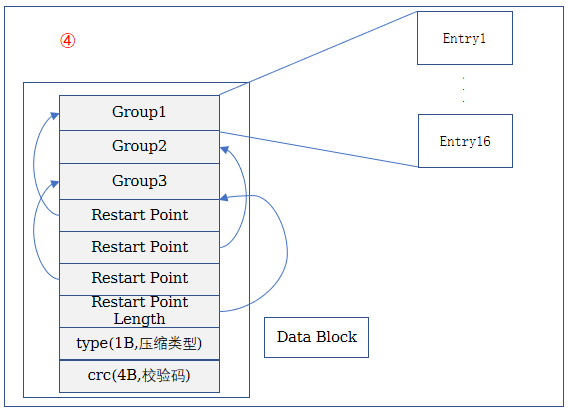
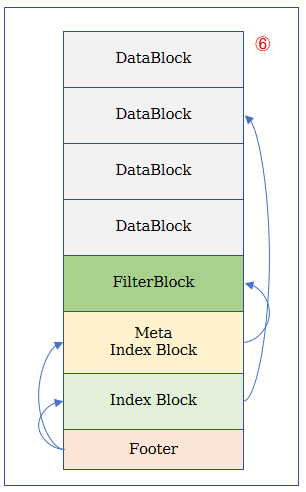
这里的memtable key就是lookup key，这一整个就是memtable entry，且sequence应该是头7字节，type是尾部1字节。

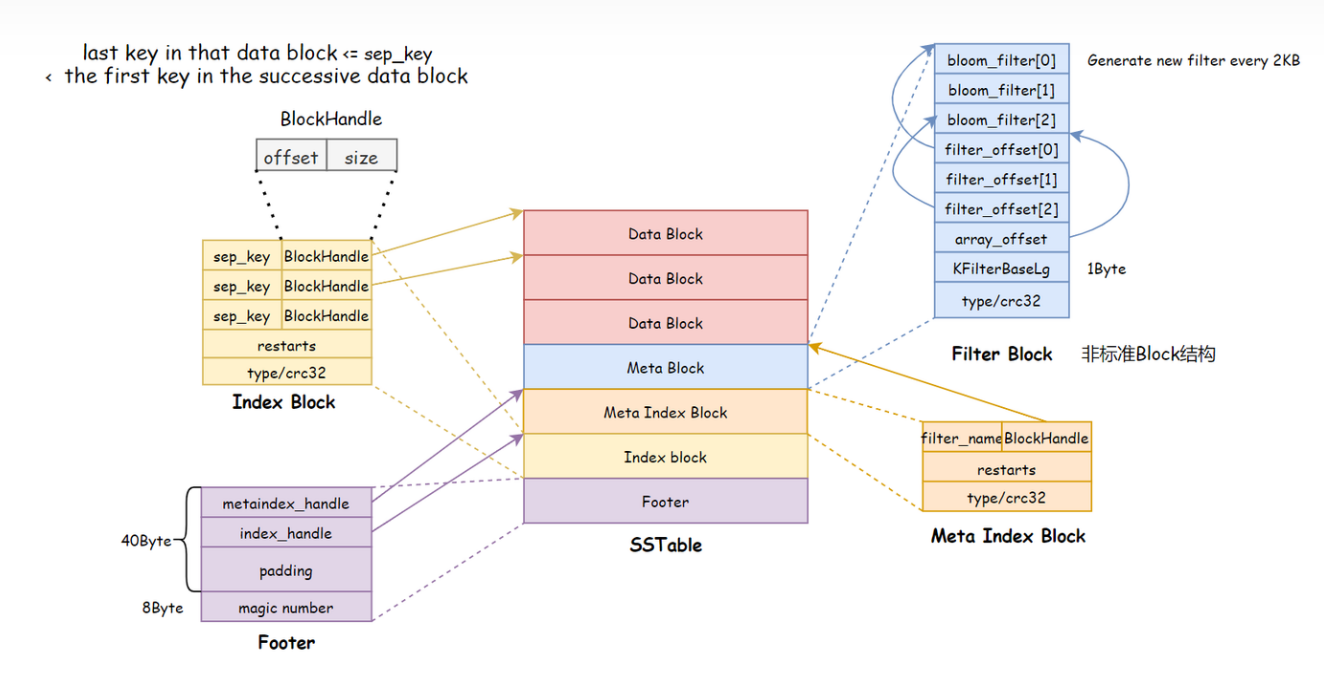
注意：SSTable是按照internal key从小到大排序的，同一层的SSTable间也是如此（level 0层例外），但需要注意的一点是当internal key的user\_key相等时，sequence越小，对应的internal key反而是越大的，这样做的目的是，对相同的 key，数据越新，sequence 越大，排在前面，更容易被找到。

**3、SSTable:**

DataBlock只有Group内部是共享key的，Group之间不共享key，一般16个Entry为一组;IndexBlock也是标准的Block，所以结构上与DataBlock是一致的，拥有共享的key，但是FilterBlock是非标准的，有自定义的格式，由于目前只有布隆过滤器，所以Meta Index Block原则上只有一条键值对。

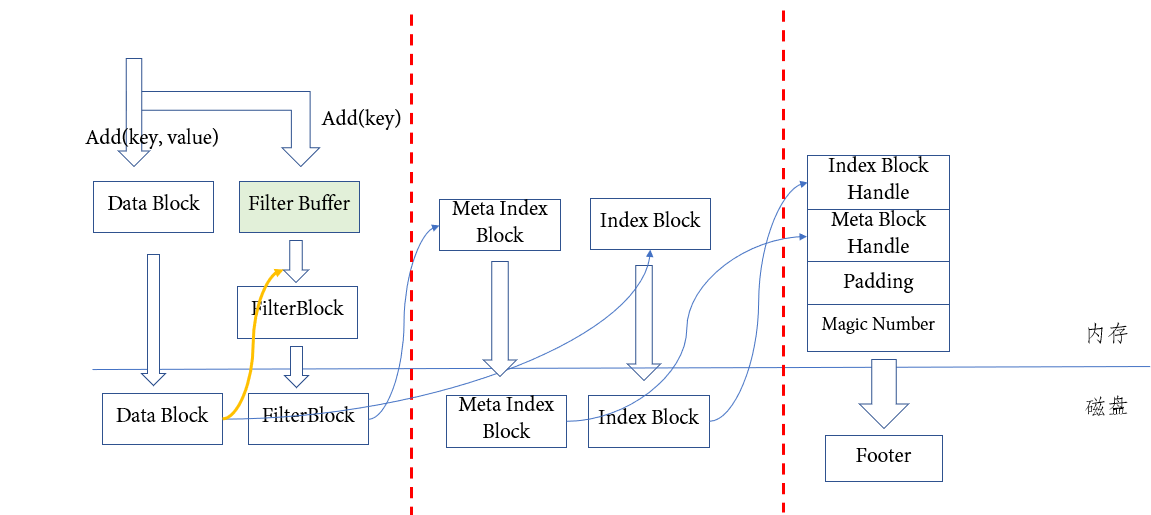
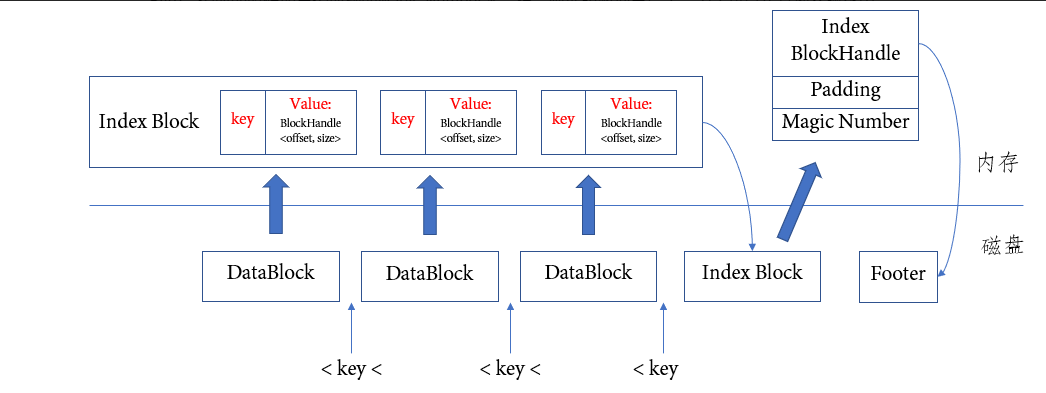
  



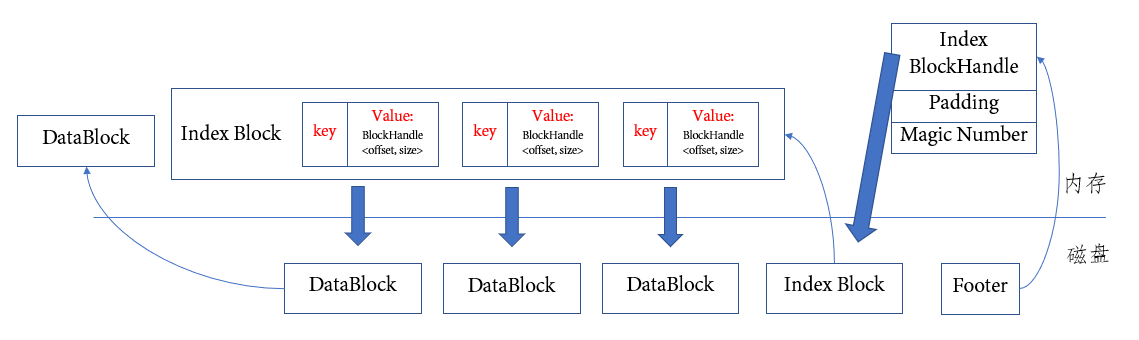
注：Filter Block是一个DataBlock对应一个bloom\_filter，但是filter\_offset是每2KB生成一个，所以这里的图应该是一个bloom\_filter有多个filter\_offset对应。

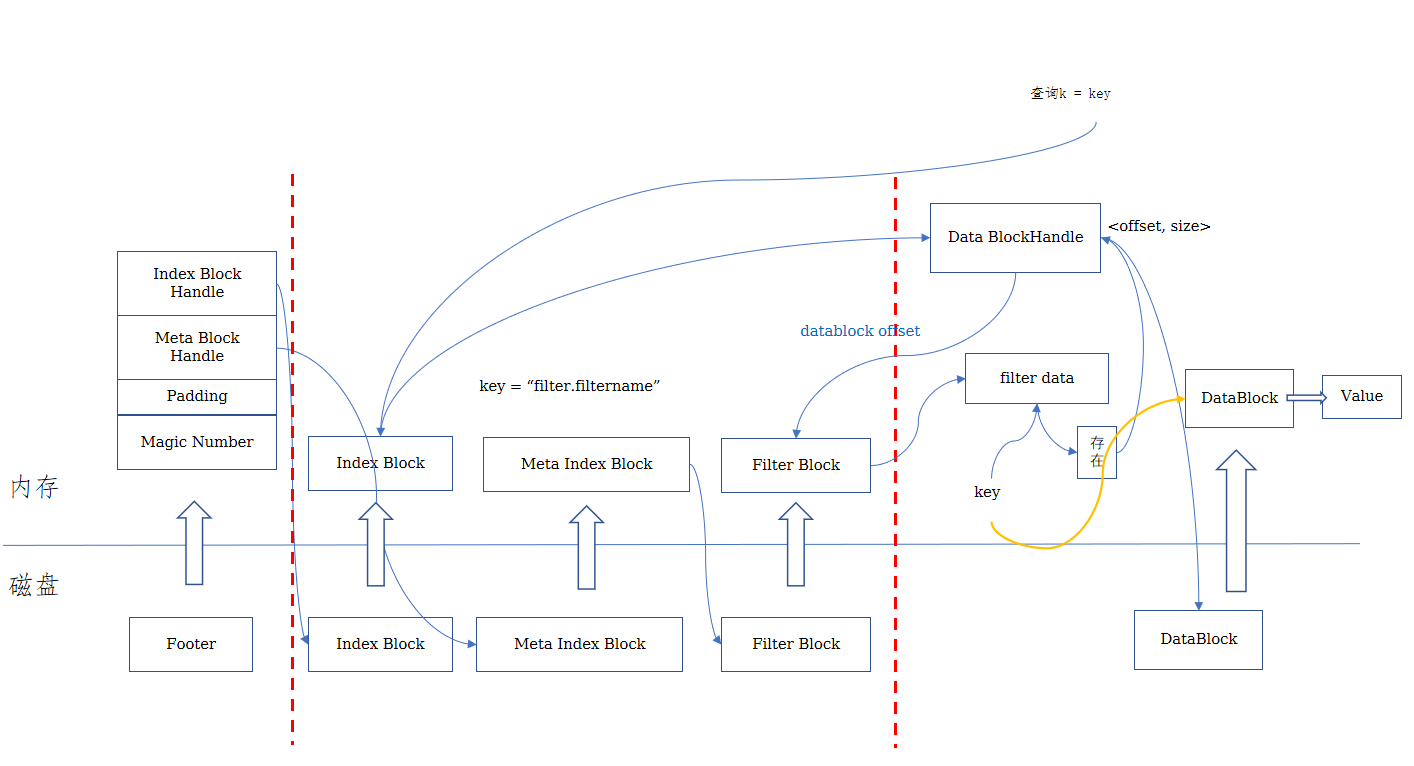
1. **构建过程**



首先会向DataBlock添加key/value数据，当DataBlock容量超过阈值4KB（**不包含压缩类型与CRC检验的5字节内容，因此压缩和解压缩都是针对实际存储数据的内容，并不包含这两个**）时，会把这个DataBlock持久化到磁盘。在向DataBlock添加key/value的同时会向FilterBuffer中添加key，待一个DataBlock持久化完成之后，会把FilterBuffer的数据构建成FilterBlock，等所有的DataBlock都持久化完成之后，FilterBlock也就构建完成，那么此时就可以把FilterBlock也进行持久化。FilterBlock持久化后会产生一个BlockHandle，把<"filter.filtername", BlockHandle>添加到Meta Index Block中，由于现在LevelDB只有一种过滤器：布隆过滤器，所以此时Meta Index Block的构建就完成了，再将其持久化到磁盘，得到一个Meta Block Handle。每持久化一个Data Block，我们都会把<key, BlockHandle>插入Index Block，等所有DataBlock持久化完成之后，Index Block就构建完成了，然后就可以把Index Block也持久化，同样也得到一个Index Block Handle。 再得到Index Block Handle和Meta Block Handle之后，便可以把它们同MagicNumber一起加入Footer，如果有空余的空间，用Padding数据填满，使得Footer的数据量固定（48字节），然后把Footer持久化到磁盘。

1. **查询流程：**

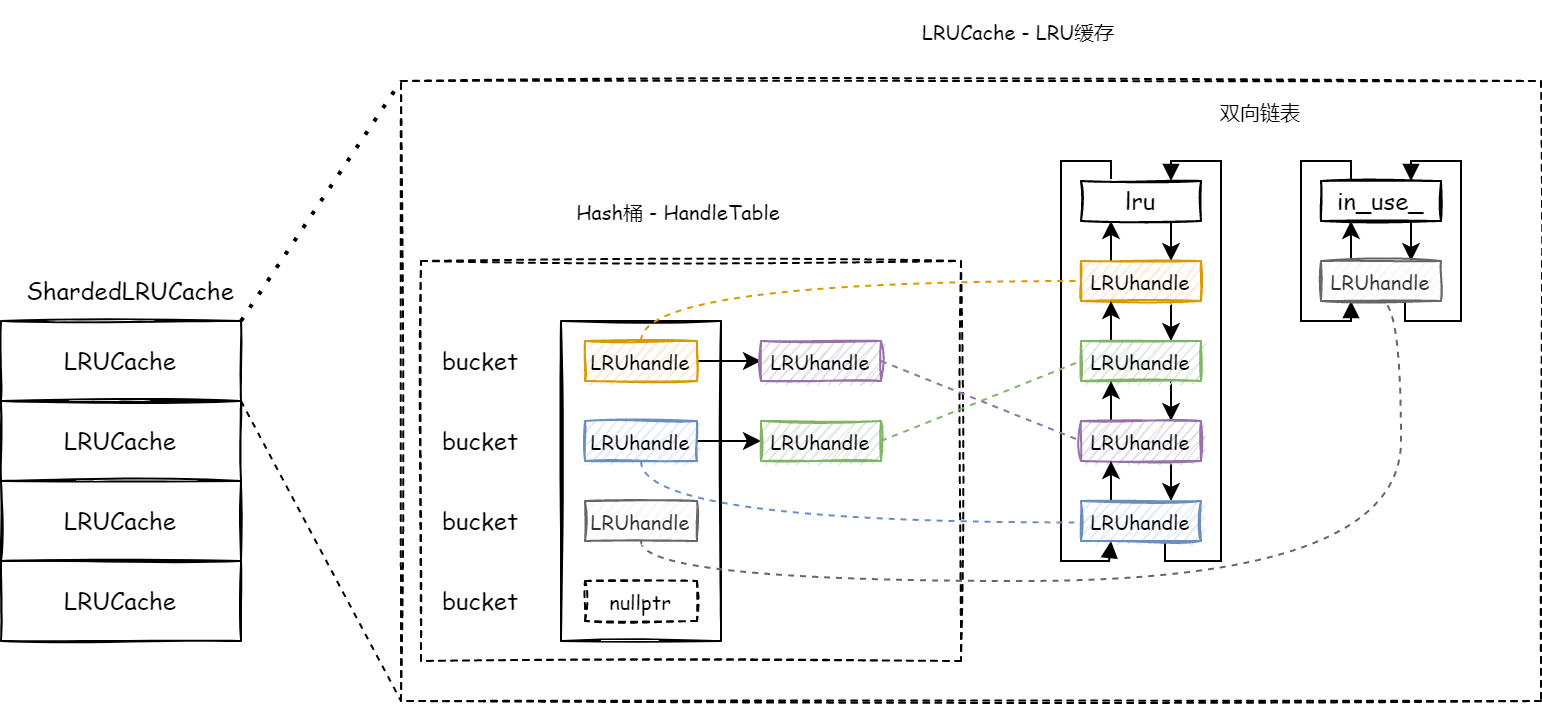




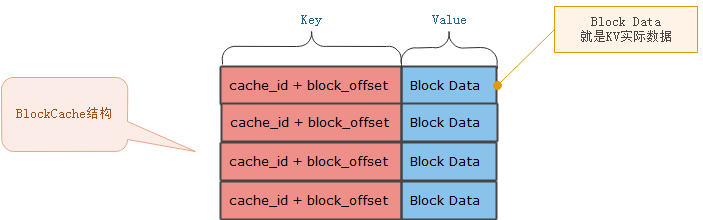
查询一个key之前，需先读取Footer，解析出Index Block Handle和Meta Block Handle，根据它们从磁盘中读取Index Block和Meta Index Block，然后用key = "filter.filtername"从Meta Index Block进行查询，得到Filter Block的BlockHandle之后从磁盘中读取到Filter Block（就是获取唯一的布隆过滤器）。查询一个key时，先在Index Block中二分查找得到Data BlockHandle，之后利用BlockHandle的offset，计算出该DataBlock对应第几个filter offset，然后在Filter Block获取filter data，再把filter data加载到布隆过滤器中查询key是否存在，如果存在，则根据Data BlockHandle从磁盘中读取DataBlock，再在DataBlock中进行查询最终得到value（二分查找，Restart Point指向Group的第一个键值对，key是完整的）。

1. **LRU Cache：**

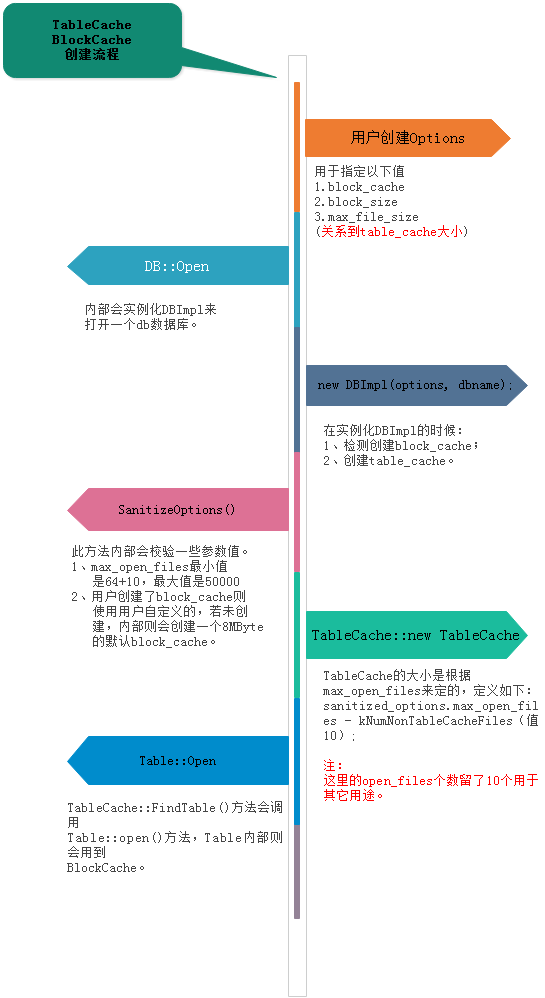
缓存的主要是SSTable，LevelDB将其封装成Table Cache进行使用，但实际上在TableCache只是将TableAndFile保存到缓存中，且TableCache是存放的SSTable文件句柄、Footer、IndexBlock等信息，DataBlock并没有直接加载进来，而是等后续需要使用DataBlock时才将其加载到内容。DataBlock加载到内存也分两种情况，如果采用的时mmap内存映射的方式，则不需要添加的Cache（Block Cache是配置可选的，即在配置文件中指定是否打开这个功能。根据cache\_id和DataBlock在SSTable中的偏移量作为key，注意它是共享的，即不同SSTable的DataBlock都可以放到这里），如果是随机读取，则需要将会DataBlock添加到Cache中，统一管理。





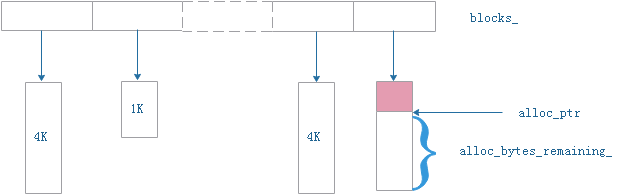


LevelDB 就是这样通过两个cache来加快读取速度的。从这里可以看出，如果读取的数据局部性比较好，也就是说要读的数据大部分在cache里面都能读到，那么读取效率应该还是很高的，而如果是对key进行顺序读取效率也应该不错，因为一次读入后可以多次被复用。但是如果是随机读取，效率会有所下降。

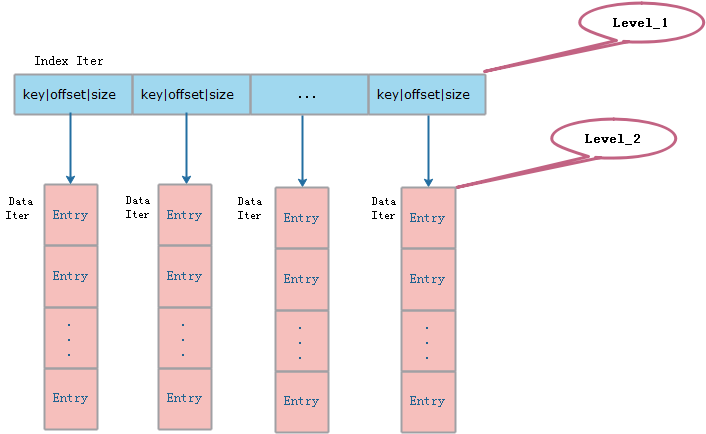
1. **Arena：**

频繁向系统申请内存容易造成碎片，申请内存很小时则更严重，粗粒度内存管理主要是给MemTable和Skiplist使用。申请的内存大于1024B时，直接从内存申请返回，避免对现有剩下的内存的浪费。申请内存小于等于1024B时，直接向系统申请4096内存大小并重新赋值alloc\_ptr、alloc\_bytes\_remaining\_，这样的话如果之前的block中还剩余内存，则直接浪费掉。



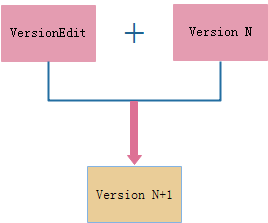
# **TwoLevelIterator:**

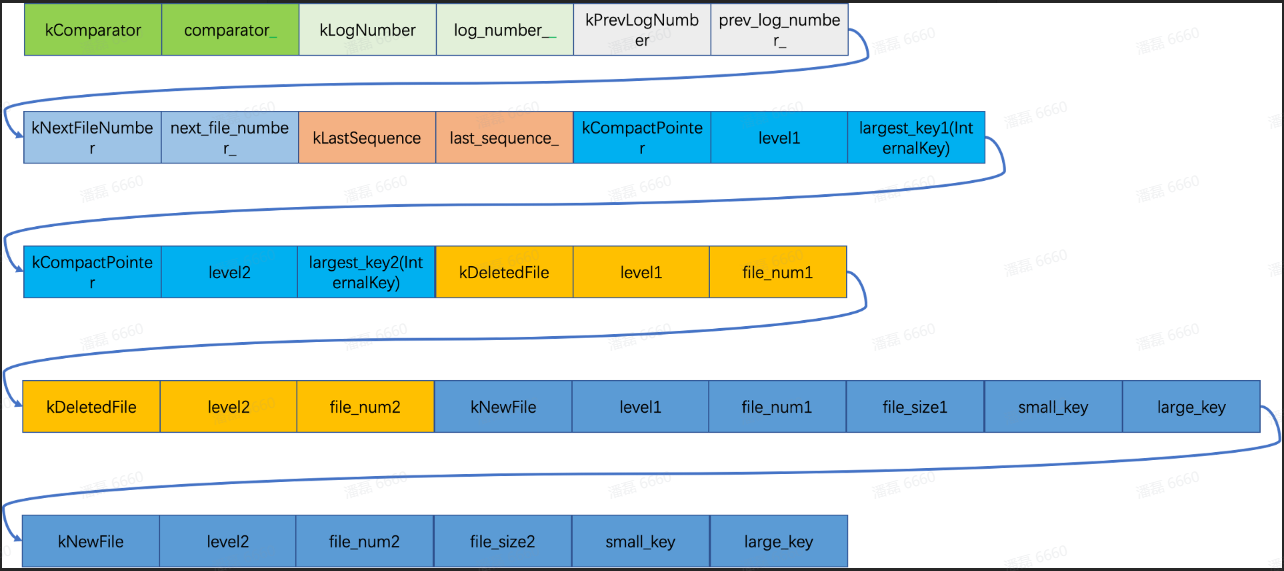
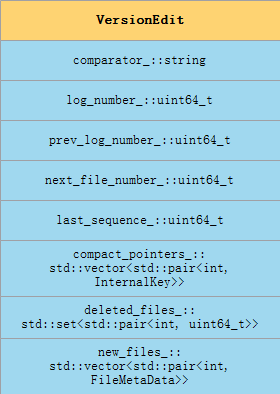
Level\_1是Index Block迭代器，Level\_2是指向DataBlock迭代器，通过这样的设计可以实现对SSTable的所有key进行向前、向后扫描的批量查询工作。



# **VersionEdit:**

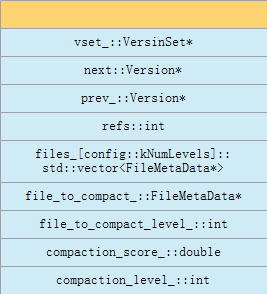
compact过程中会有一系列改变当前Version的操作（FileNumber增加，删除一些SSTable，增加一些新的SSTable），为了缩小Version切换的时间点，将这些操作封装成VersionEdit，待compact完成时，将VersionEdit一次应用到当前Version即可得到最新状态的Version。

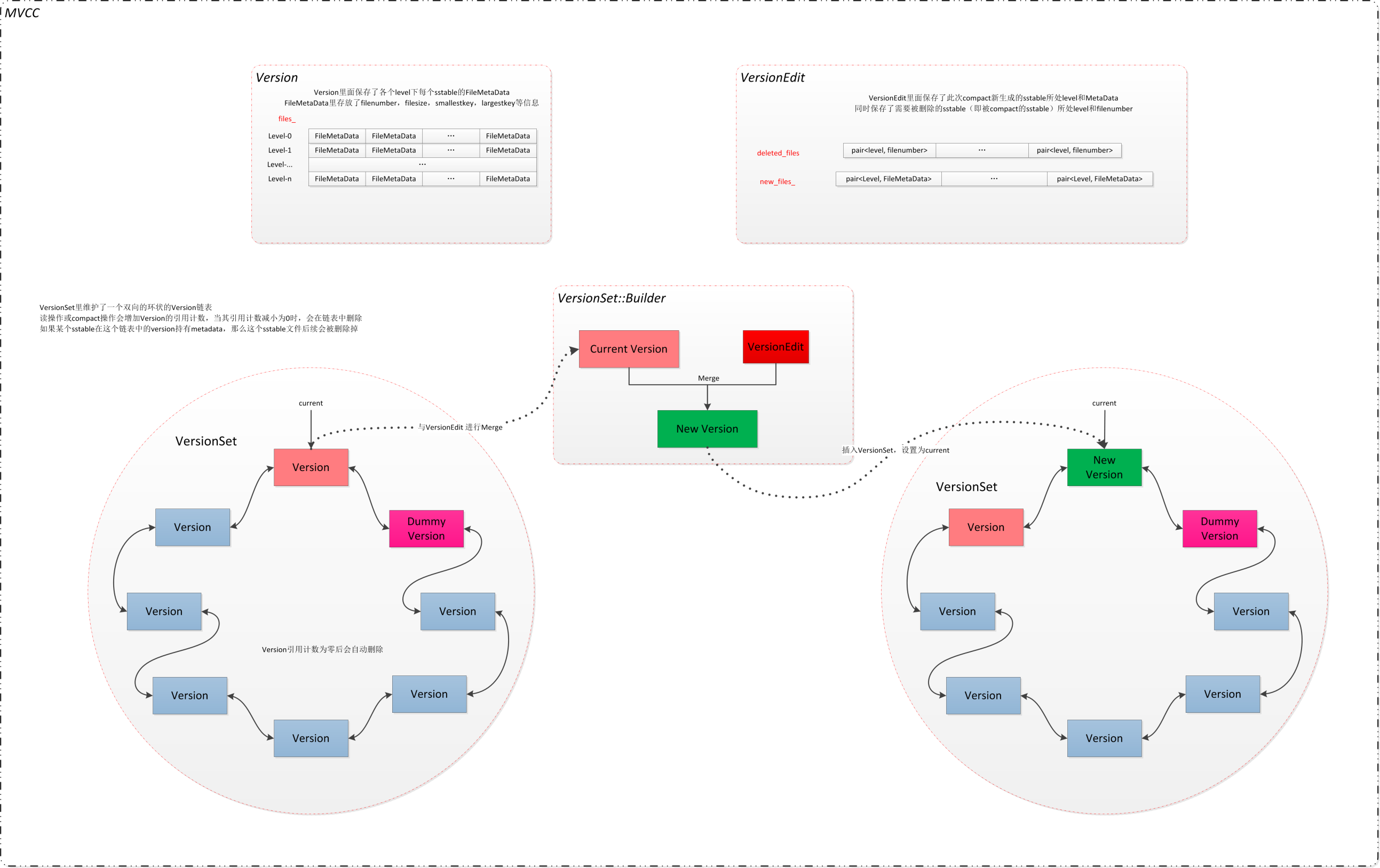


# **VersionSet:**

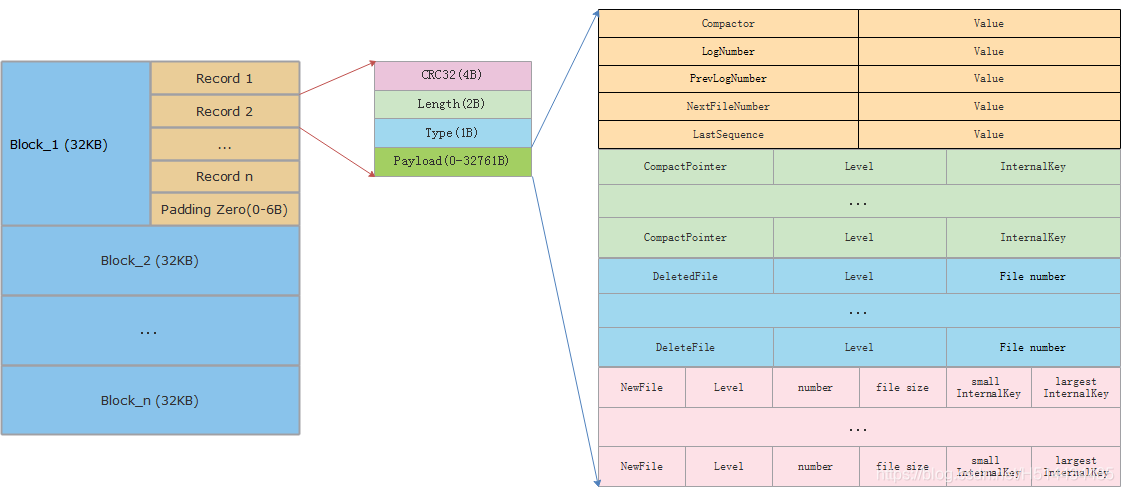




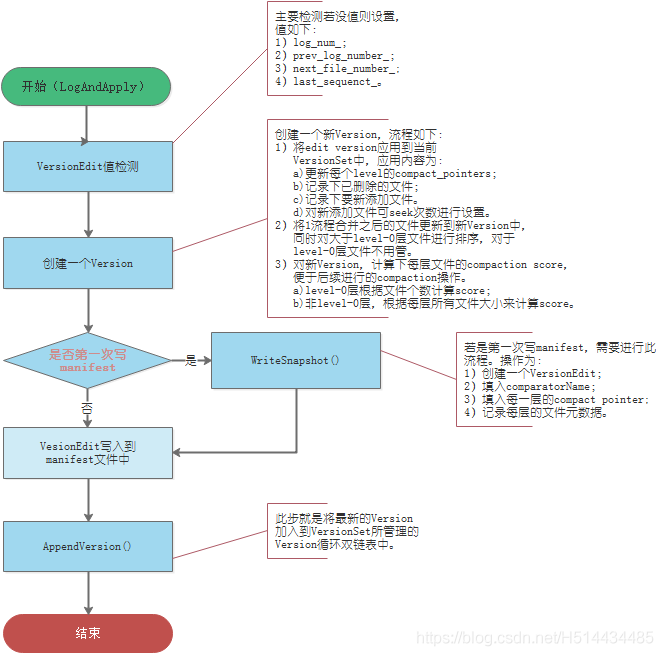


# **Manifest:**

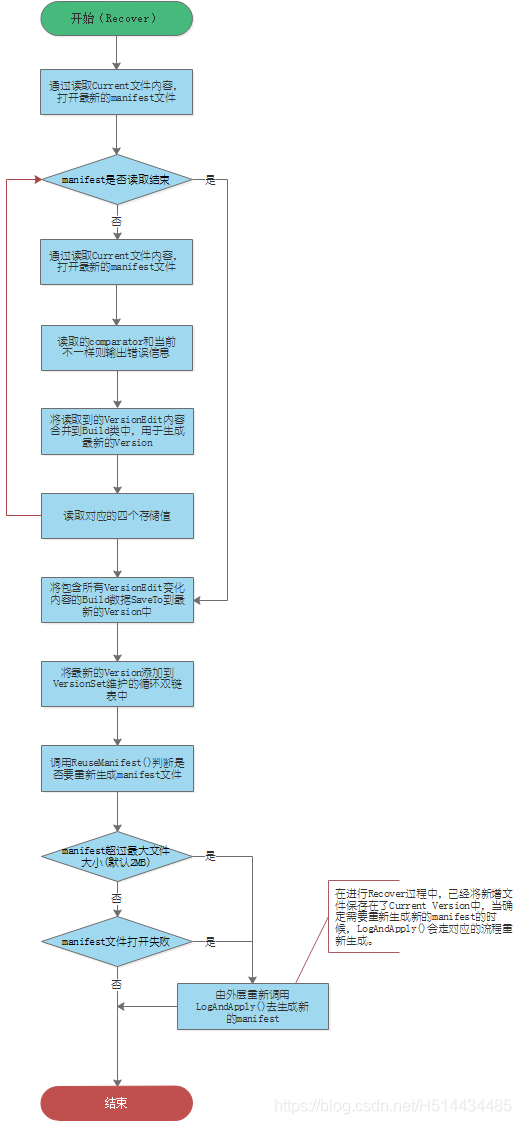
为了重启leveldb后可以恢复到退出前的状态，需要将db中的状态保存下来，这些信息就保存在Manifest中，而CURRENT文件保存了Manifest文件的名称。manifest中不仅保存当前的状态，也会将历史的状态也保存起来，考虑到每次状态的完全保存需要空间和耗费的时间比较多，当前采用的方式是，只在manifest开始保存完整的状态信息（通过VersionSet::WriteSnapShot()来实现），接下来只保存每次compact产生的VersionEdit。实际上，Manifest文件保存的元信息与Log记录格式相同，都是以块为单位，元信息的内容为VersionEdit结构编码之后生成的字符串，因为每个版本的生成都是由当前版本与一个VersionEdit合并之后生成的，因此读取元信息内容之后首先解码一个个的VersionEdit结构，将其依次应用到最新的版本中，即可恢复出当前版本。

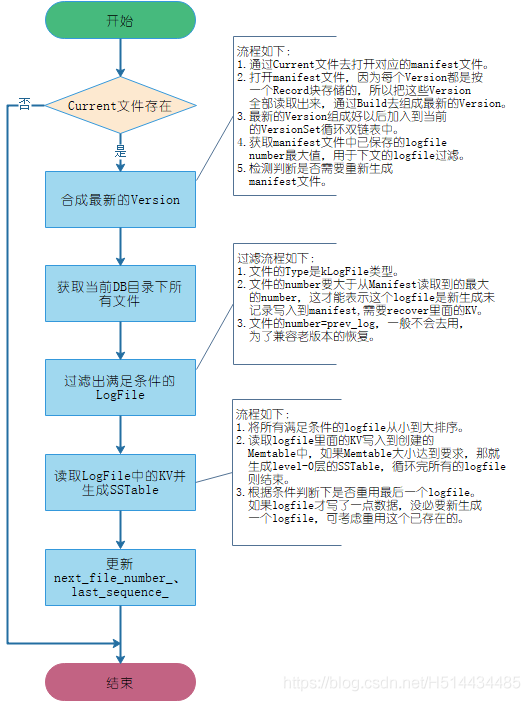


# **SSTable文件变化：**

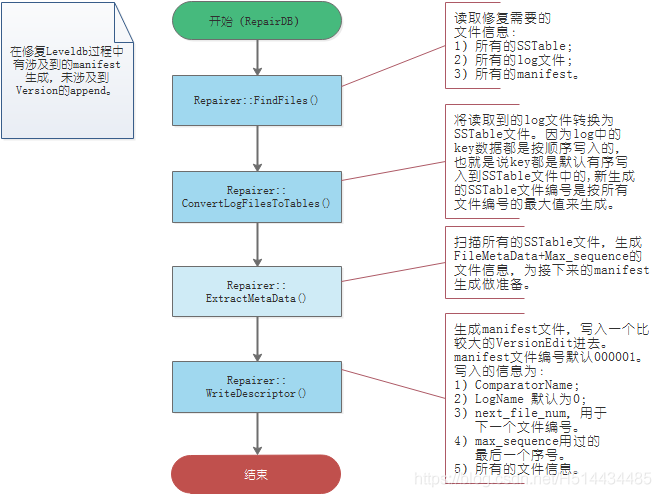


# **上电还原：**

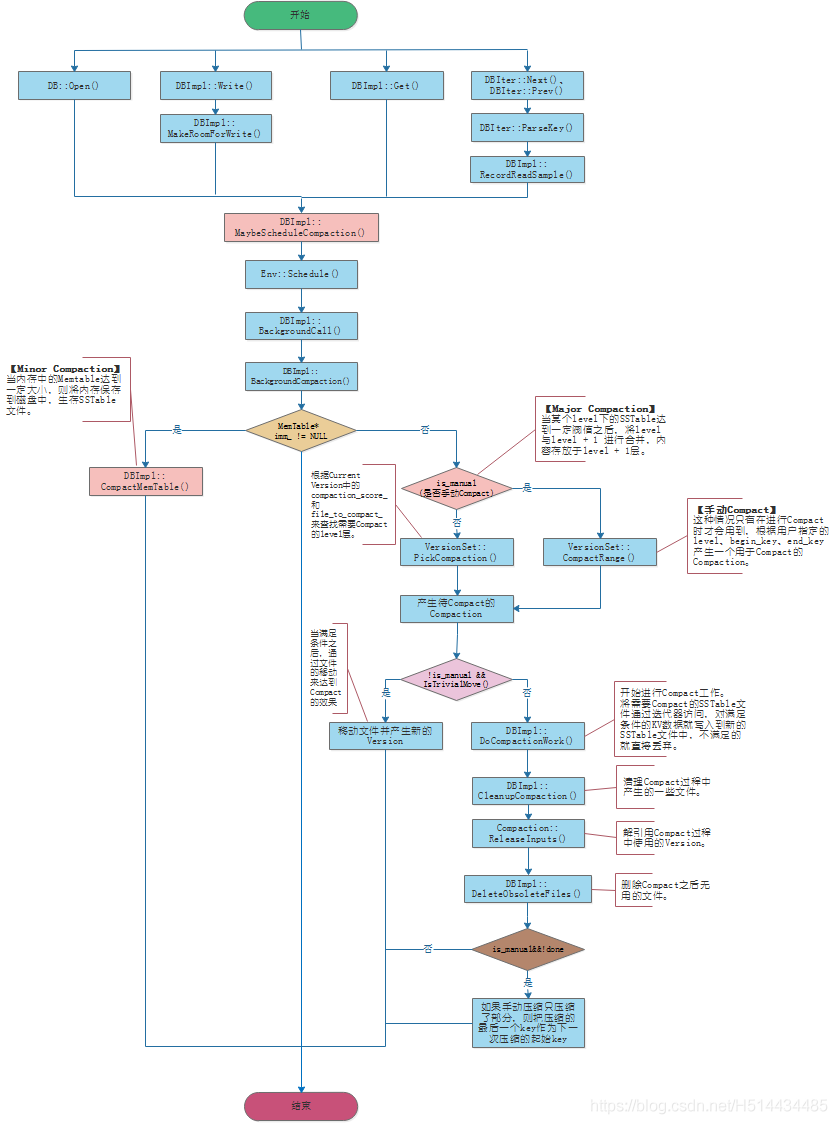




# **修复LevelDB：**



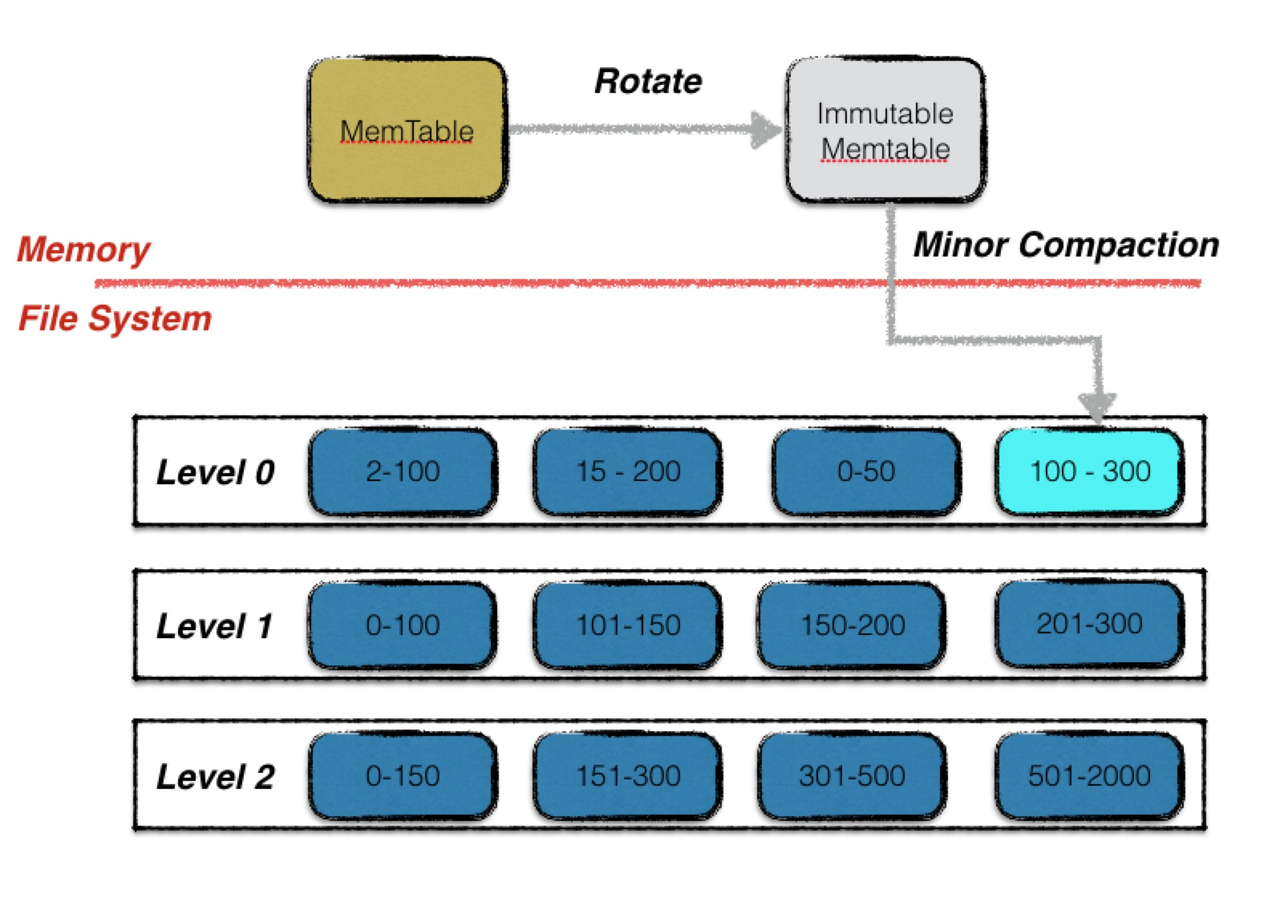
# **Compact触发流程：**



1. minor Compaction：

minor Compaction作用就是要快速的将内存中的MemTable落地到磁盘生成SSTable文件，整个过程中并未考虑到不同SSTable文件直接的key的顺序问题。新SSTable文件，不一定都放在level-0层，有可能level-1或level-2层，但最多是level-2层。

①将Immutable落地生成SSTable文件、同时将文件信息放入Table\_Cache中；②生成新的Version文件；③删除无用文件。

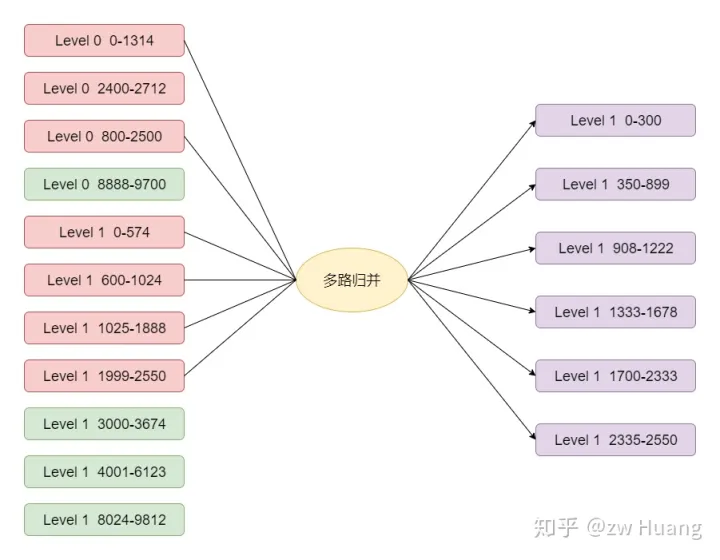
 

1. major Compation：

Major compaction 是将不同层级的 sst 的文件进行合并，注意图中level n+1只有一个SSTable，但实际上可能有多个SSTable的键范围与level n层有重叠。

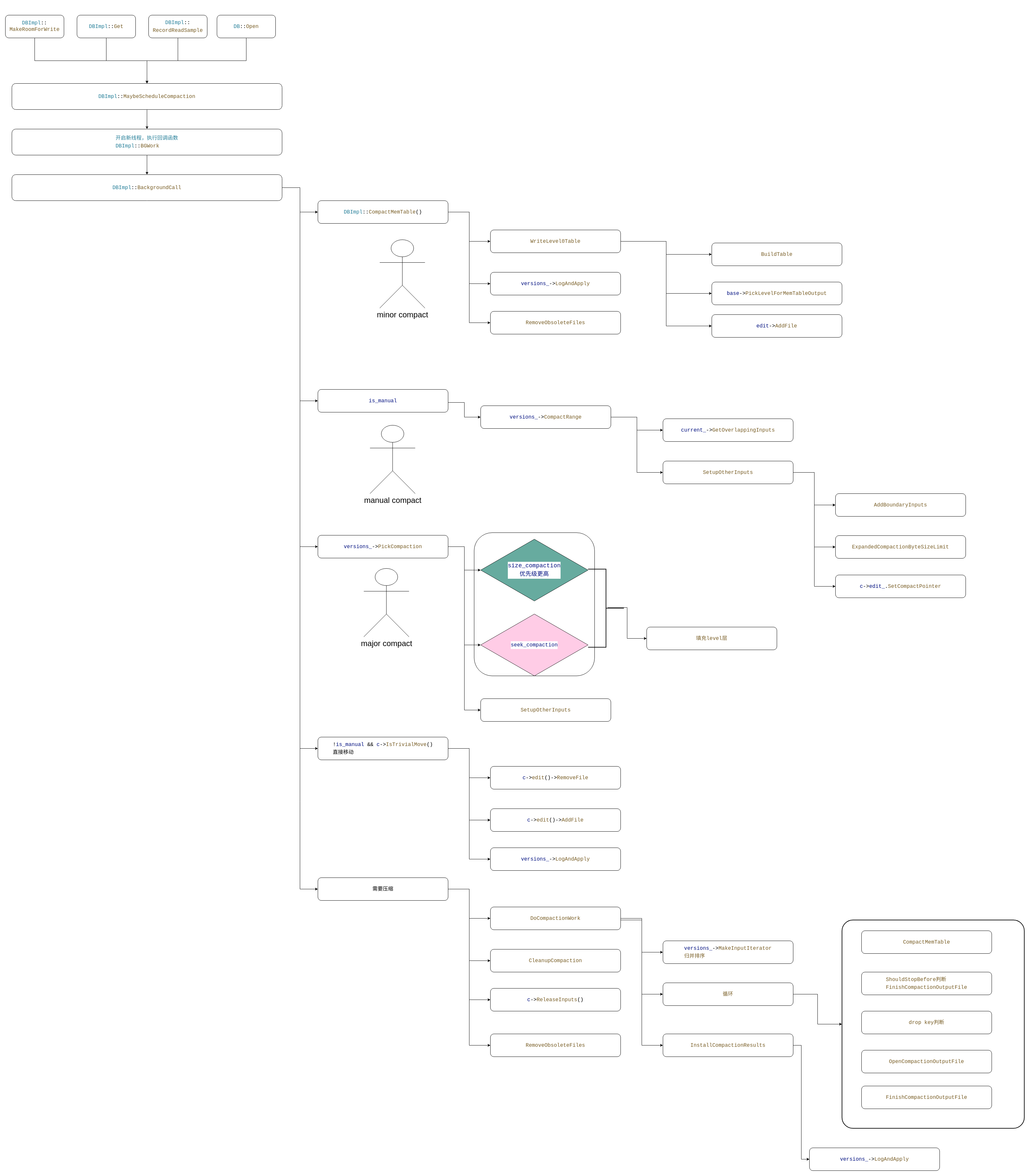
1）压缩基本思想：所有与level n层要压缩的SSTable文件有重叠的level n+1层文件都要参与compact，得到这些文件后反过来看下，如果在不增加level + 1 层文件的前提下，看能否增加level层的文件。也就是在不增加 level + 1 层文件，同时不会导致 compact 的文件过大的前提下，尽量增加 level 层的文件数。

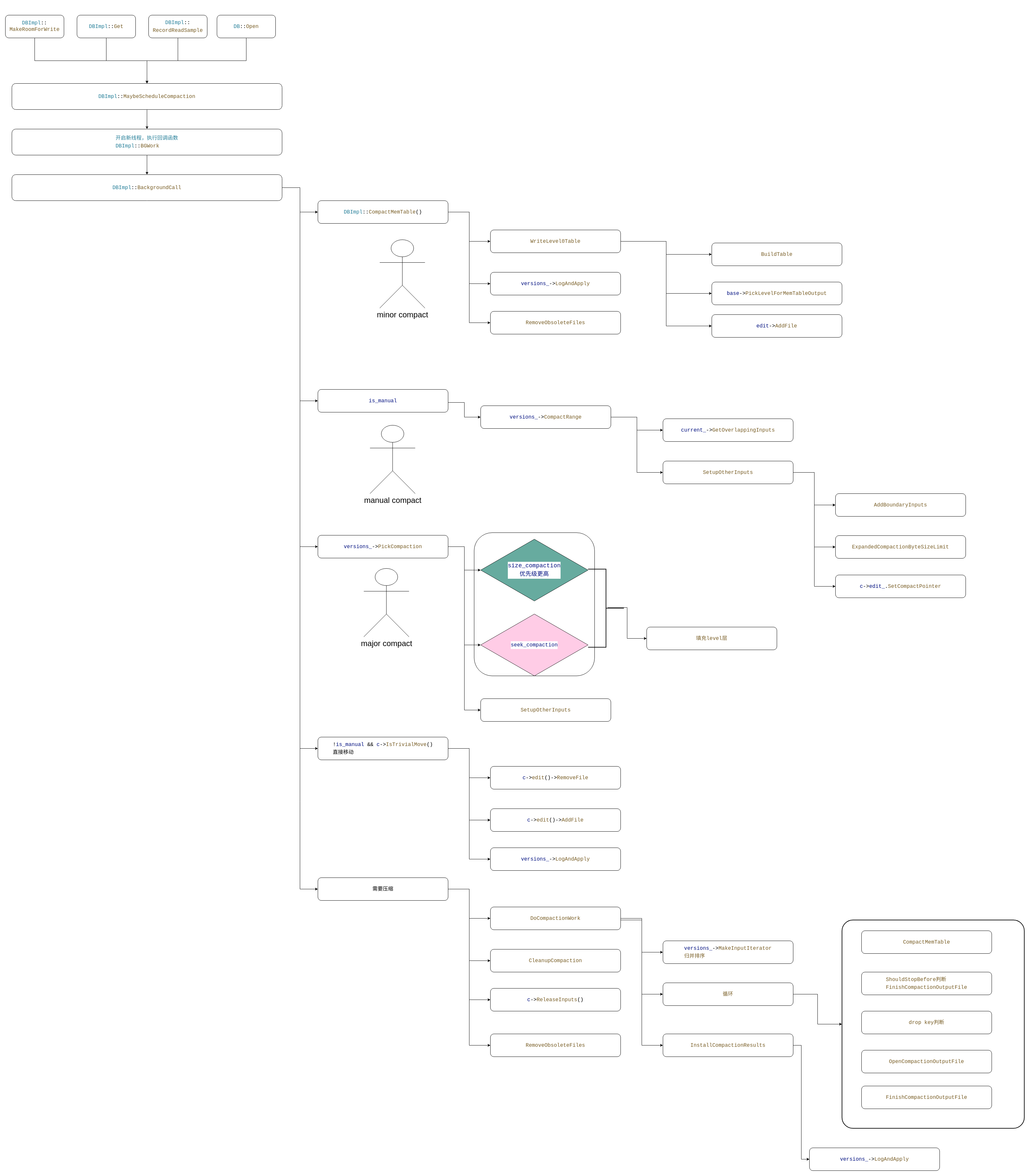
2）压缩性能考量：level与level+1层归并压缩之后，最后的文件是要放到level+1层的，在方法SetupOtherInputs(c)中获取到压缩之后的key范围[all\_start，all\_limit]，并查询出level+2层与level+1层overlap的SSTable，存放到grandparents\_（level n+2）中，主要目的是降低level层与level+1层压缩之后的SSTable文件将来与level+2层的SSTable文件进行归并压缩因overlap太多造成的Compact耗时太久问题。

1. manul Compation：

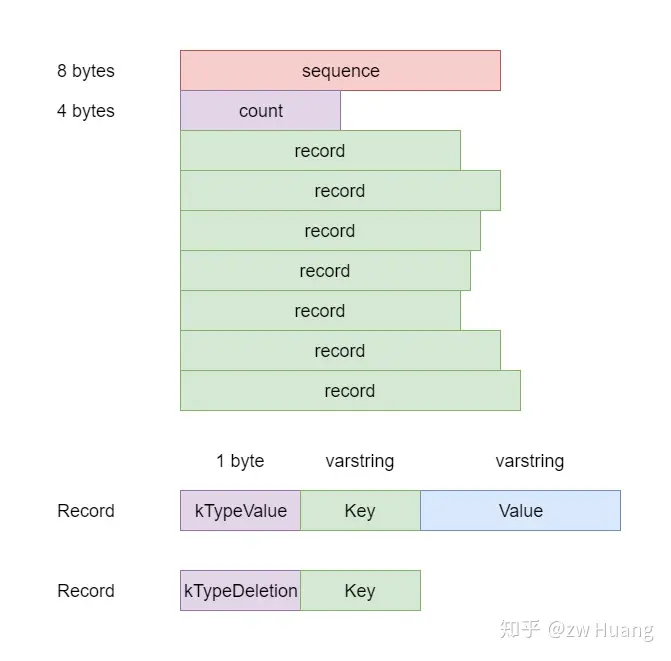
手动指定Compaction的生成一般用在性能测试中，正常系统运行过程中是不会调用的。





# **WriteBatch and Write：**

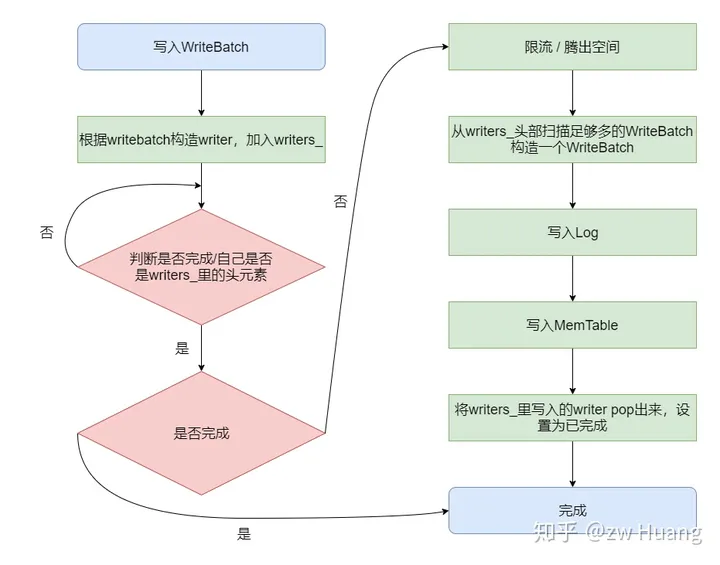
WriteBatch只是一个辅助结构，可以将多个Kv的写入按顺序累积起来，然后一次性写入提高效率。



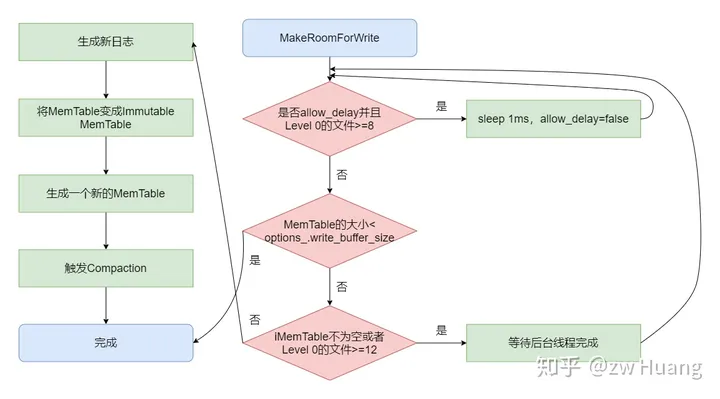
Write就是真实写入的函数了，将一个WriteBatch的内容写入到数据库，LevelDB的写入分为两步：

（1）写入数据到日志；（2）写入数据到MemTable。写入过程中，如果MemTable太大会触发MemTable写入到SSTable。

Write()函数为了实现线程安全，每个线程在写入时，都需要获取锁，但是这样会阻塞其它的线程，降低并发度。针对这个问题LevelDB做了一个优化，写入时当获取锁后，会将WriteBatch放入到一个std::deque<Writer\*> DBImpl::writers\_里，然后会检查writers\_里的第一个元素是不是自己，如果不是的话，就会释放锁。当一个线程检查到writers\_头元素是自己时，会再次获取锁，然后将writers\_里的数据尽可能多的写入。一次写入会涉及到写日志，占时间比较长，一个线程的数据可能被其它线程批量写入进去了，减少了等待。总结来说，一个线程的写入有两种情况：一种是恰好自己是头结点，自己写入，另外一种是别的线程帮助自己写入了，自己会检查到写入，然后就可以返回了。



Write()函数中会使用到的MakeRoomForWrite()，该函数主要是限流和触发后台线程等工作。首先判断Level 0的文件是否>=8，是的话就sleep 1ms，这里是限流的作用，Level 0的文件太多，说明写入太快，Compaction跟不上写入的速度，而在读取的时候Level 0的文件之间可能有重叠，所以太多的话，影响读取的效率，这算是比较轻微的限流，最多sleep一次；接下来判断MemTable里是否有空间，有空间的话就可以返回了，写入就可以继续；如果MemTable没有空间，判断Immutable MemTable是否存在，存在的话，说明上一次写满的MemTable还没有完成写入到SSTable中，说明写入太快了，需要等待Immutable MemTable写入完成；再判断Level 0的文件数是否>=12，如果太大，说明写入太快了，需要等待Compaction的完成；到这一步说明可以写入，但是MemTable已经写满了，需要将MemTable变成Immutable MemTable，生成一个新的MemTable，触发后台线程写入到SSTable中。

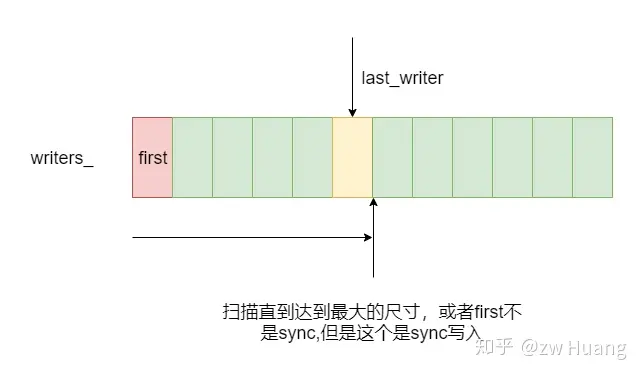


BuildBatchGroup()函数是批量写入的关键，它会从writers\_头部开始扫描，将尽可能多的writer生成一个新的WriterBatch，将这些writer的内容批量写入。

(1)首先计算出一个max\_size，表示构造的WriterBatch的最大尺寸；

(2)然后开始扫描writers\_数组，直到满足WriterBatch超过max\_size，或者；

(3)第一个writer的sync决定了这整个write是不是sync的。如果第一个writer不是sync的，碰到一个sync的writer，表明这个writer无法加入到这个批量写入中，所以扫描就结束了;如果第一个writer是sync的，后面的writer不是sync并不影响加入到BatchGroup中，因为writers\_队列永远被处理是队头，如果可以的话，就会稍待后面的writer。



# **Get：**

数据存在于三个地方：

（1）最新的写入在MemTable里；

（2）如果上一次MemTable写满后，转换为Immutable MemTable，如果还没有完成写入到SSTable里，那么就有可能存在于Immutable MemTable里；

（3）存在于SSTable里，SSTable里的数据从Level 0开始从新变旧，Level越低，数据越新。Level 0里的SSTable比较特殊，因为是MemTable转换过来的，虽然每个SSTable都是有序的，但是每个SSTable的键的范围可能有重叠，也就是一个键可能存在于多个SSTable里，文件名下标越大的SSTable里的键越新。而非level 0的SSTable，都是后台生成的，保证了SSTable之间的数据无重叠，一个键只有可能存在于一个SSTable里。

