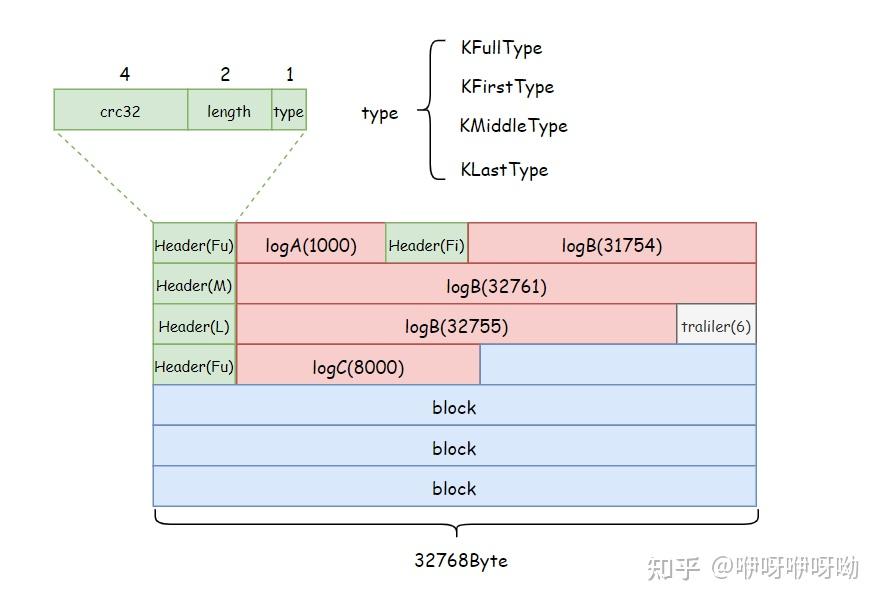
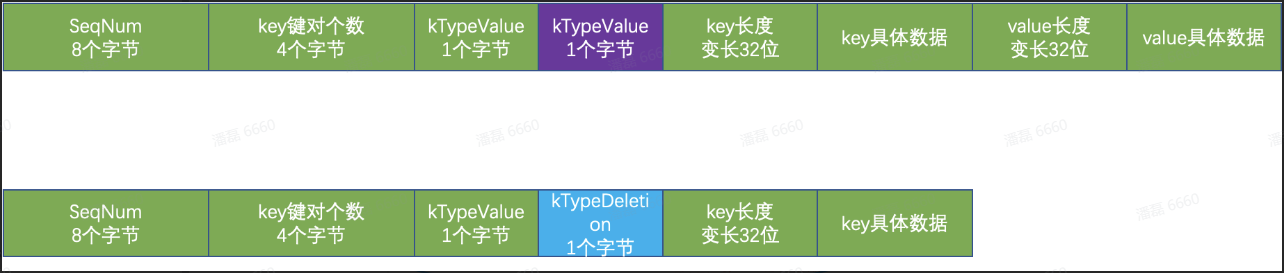
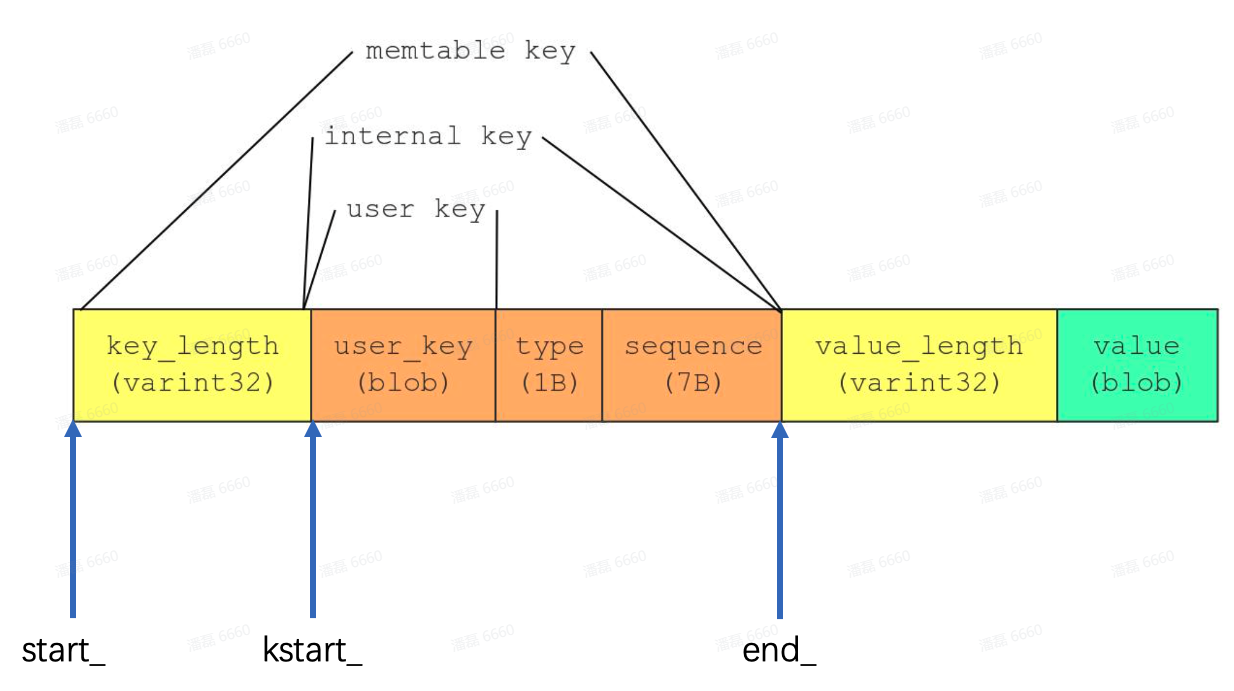
**1、WAL Log：**



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。

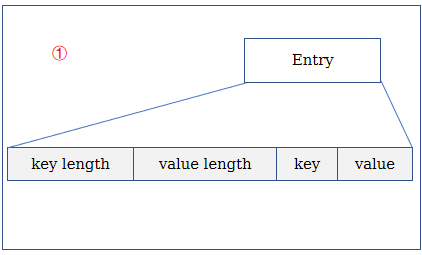
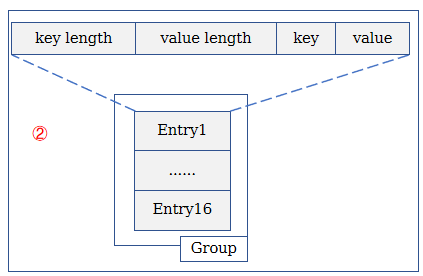
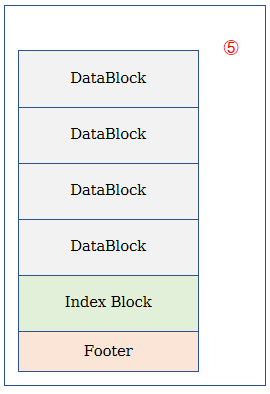


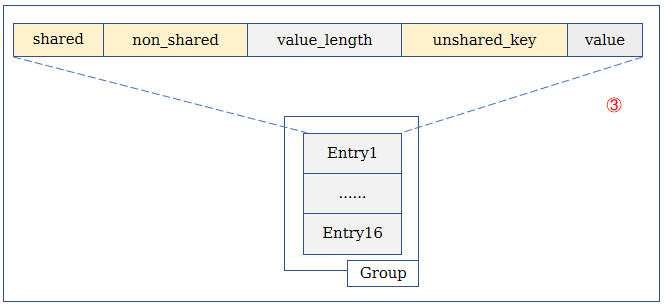
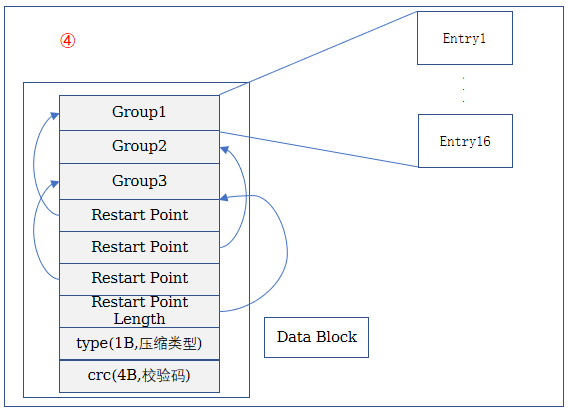
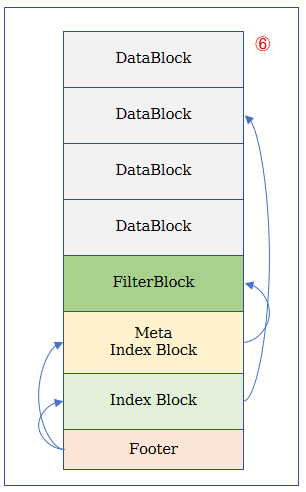
1. **Memtable（skiplist）：**

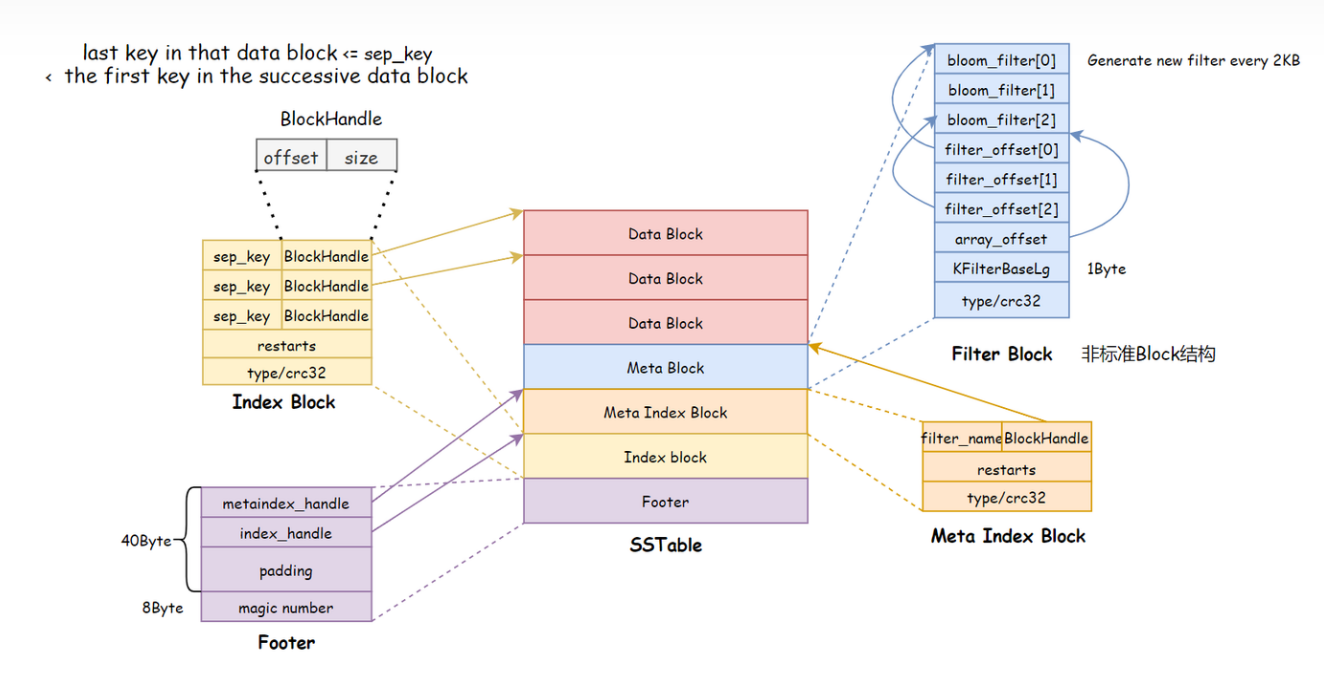


这里的memtable key就是lookup key，这一整个就是memtable entry。**3、SSTable:**

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。

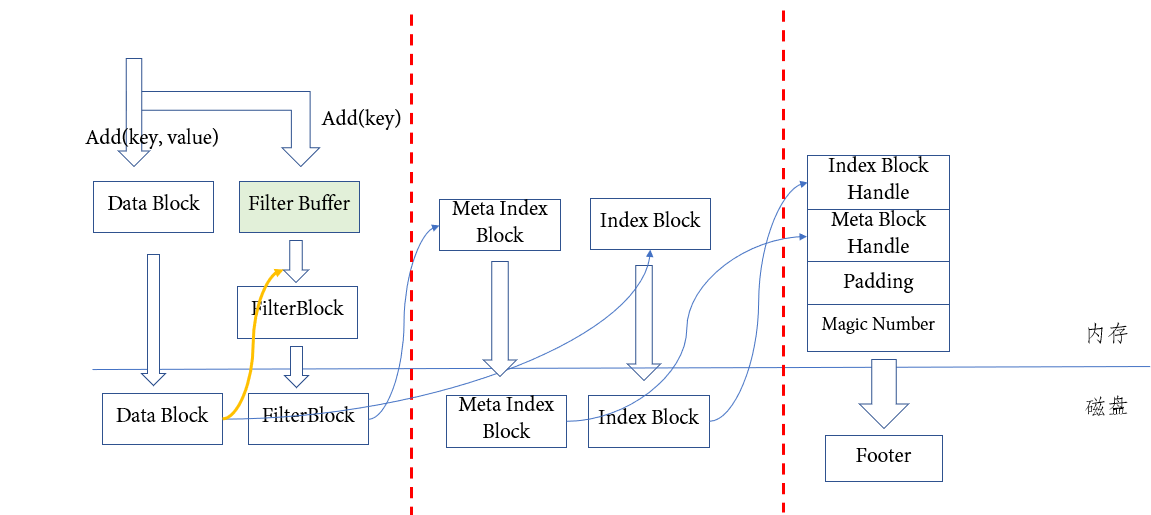
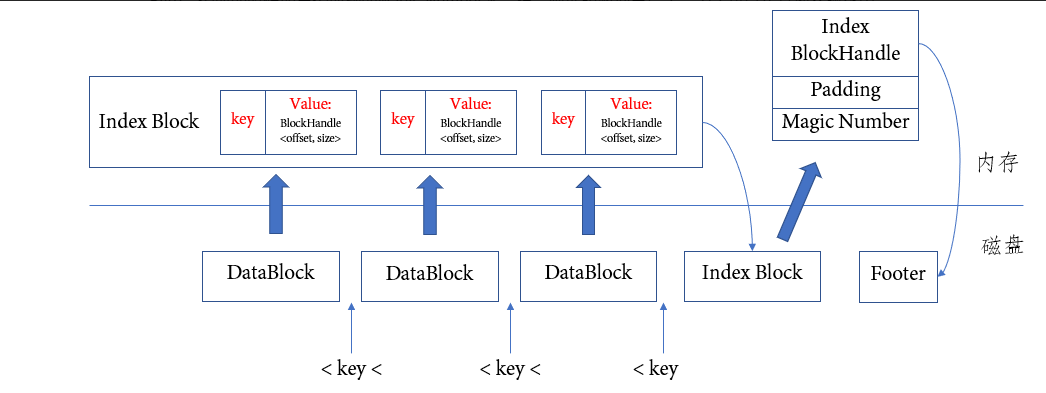
  



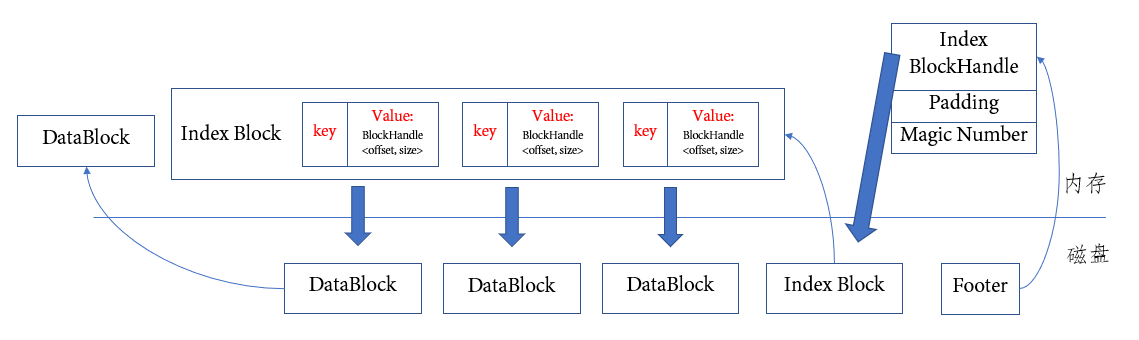
注：Filter Block是一个DataBlock对应一个bloom\_filter，但是filter\_offset是每2KB生成一个，所以这里的图应该是一个bloom\_filter有多个filter\_offset对应。

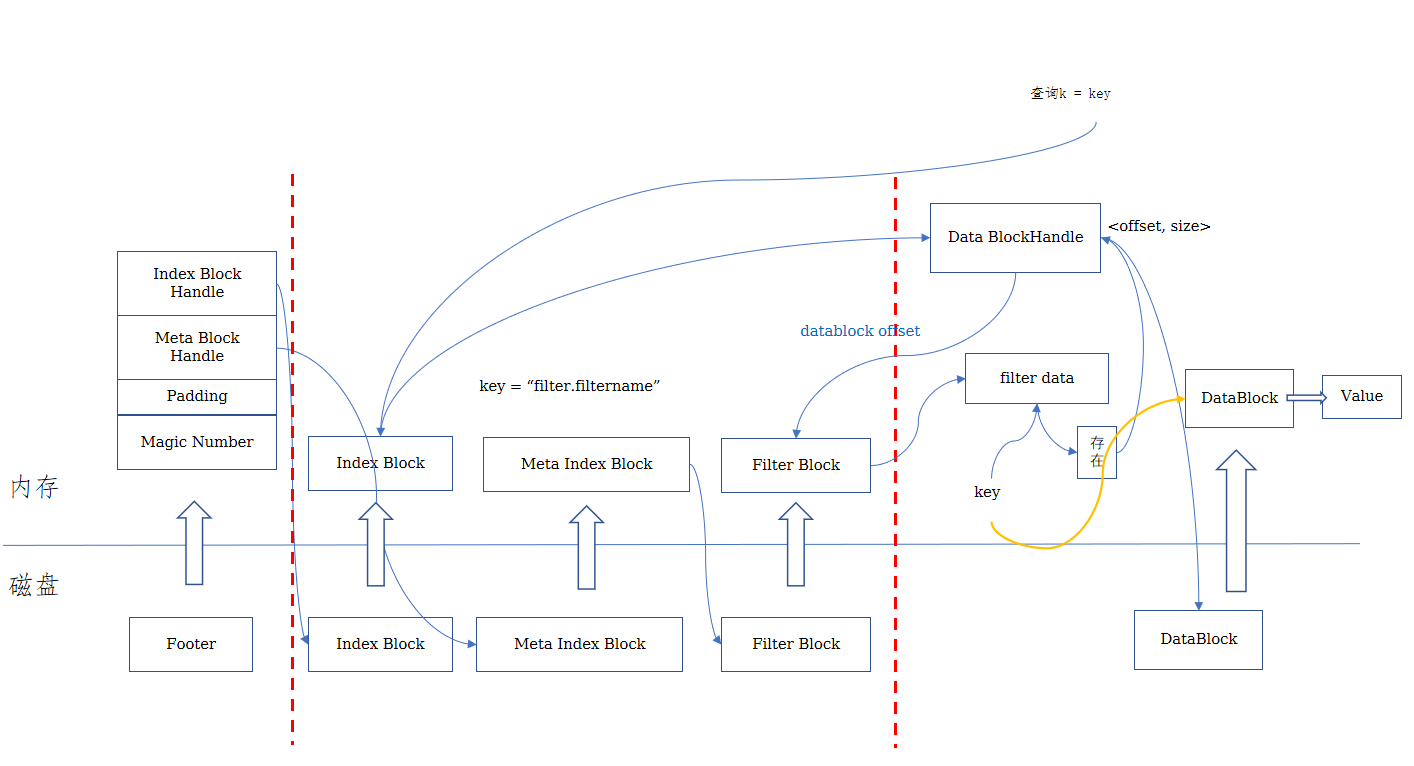
1. **构建过程**



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：**

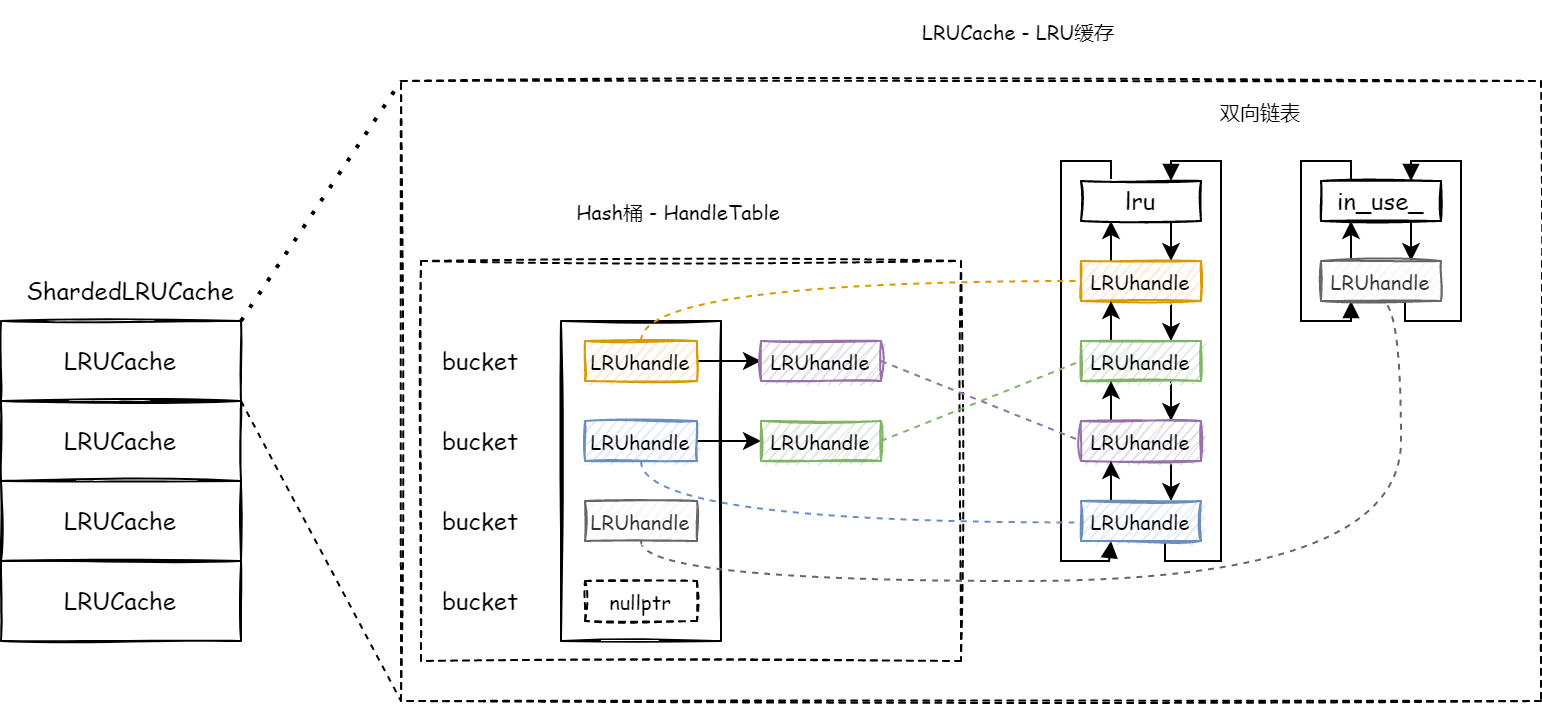


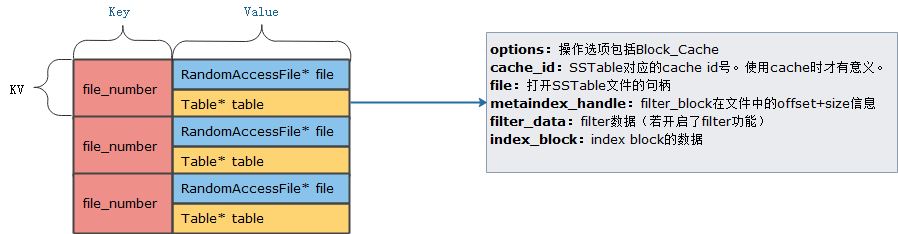


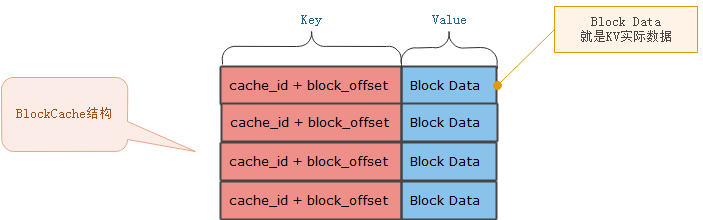
查询一个key之前，需先读取Footer，解析出Index Block Handle和Meta Block Handle，根据它们从磁盘中读取Index Block和Meta Index Block，然后用key = "filter.filtername"从Meta Index Block进行查询，得到Filter Block的BlockHandle之后从磁盘中读取到Filter Block（就是获取唯一的布隆过滤器）。查询一个key时，先在Index Block中二分查找得到Data BlockHandle，之后利用BlockHandle的offset，计算出该DataBlock对应第几个filter 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：**

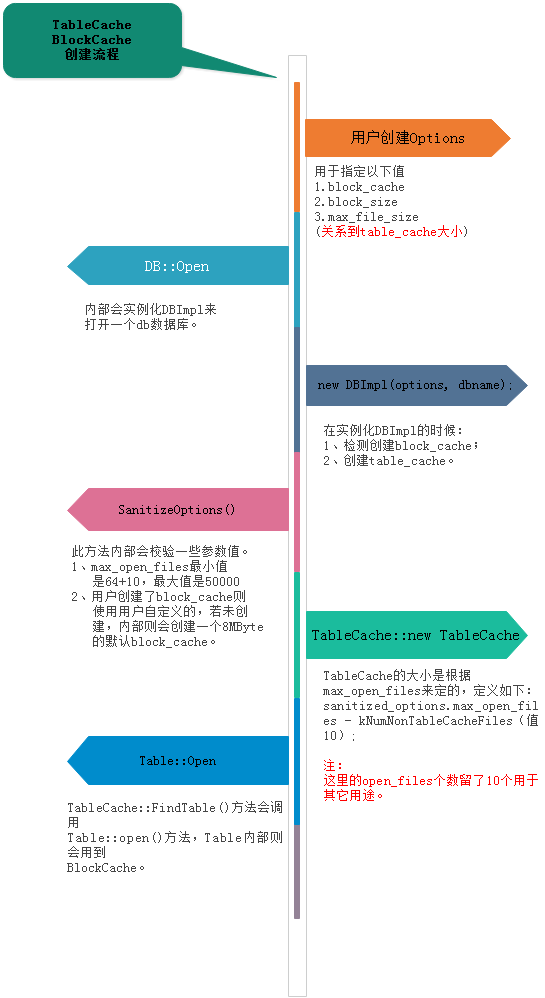
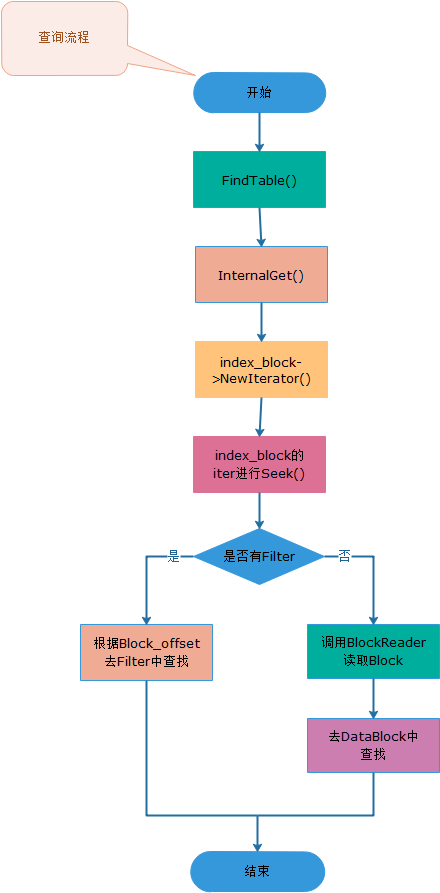
缓存的主要是SSTable，LevelDB将其封装成Table Cache进行使用，但实际上在TableCache只是将TableAndFile保存到缓存中，且TableCache是存放的SSTable文件句柄、Footer、IndexBlock等信息，DataBlock并没有直接加载进来，而是等后续需要使用DataBlock时才将其加载到内容。DataBlock加载到内存也分两种情况，如果采用的时mmap内存映射的方式，则不需要添加的Cache（Block Cache是配置可选的，即在配置文件中指定是否打开这个功能。根据cache\_id和DataBlock在SSTable中的偏移量作为key，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





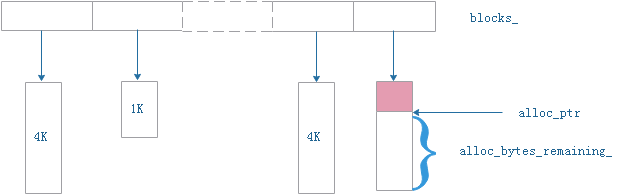


LevelDB 就是这样通过两个cache来加快读取速度的。从这里可以看出，如果读取的数据局部性比较好，也就是说要读的数据大部分在cache里面都能读到，那么读取效率应该还是很高的，而如果是对key进行顺序读取效率也应该不错，因为一次读入后可以多次被复用。但是如果是随机读取，效率会有所下降。

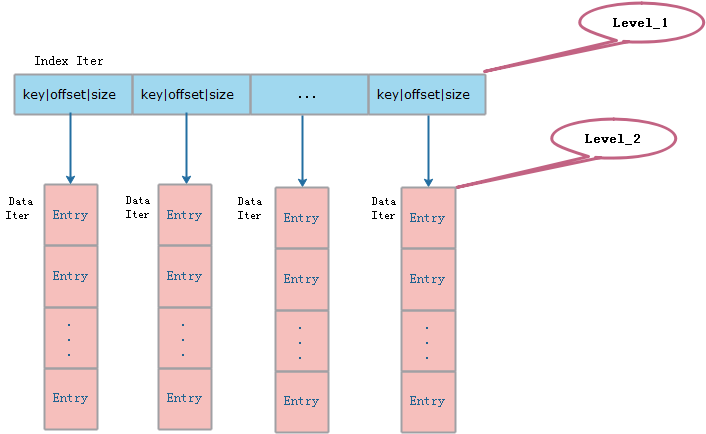
1. **Arena：**

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。



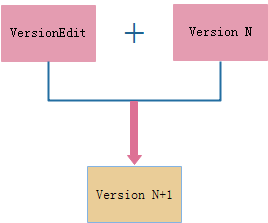
# **TwoLevelIterator:**

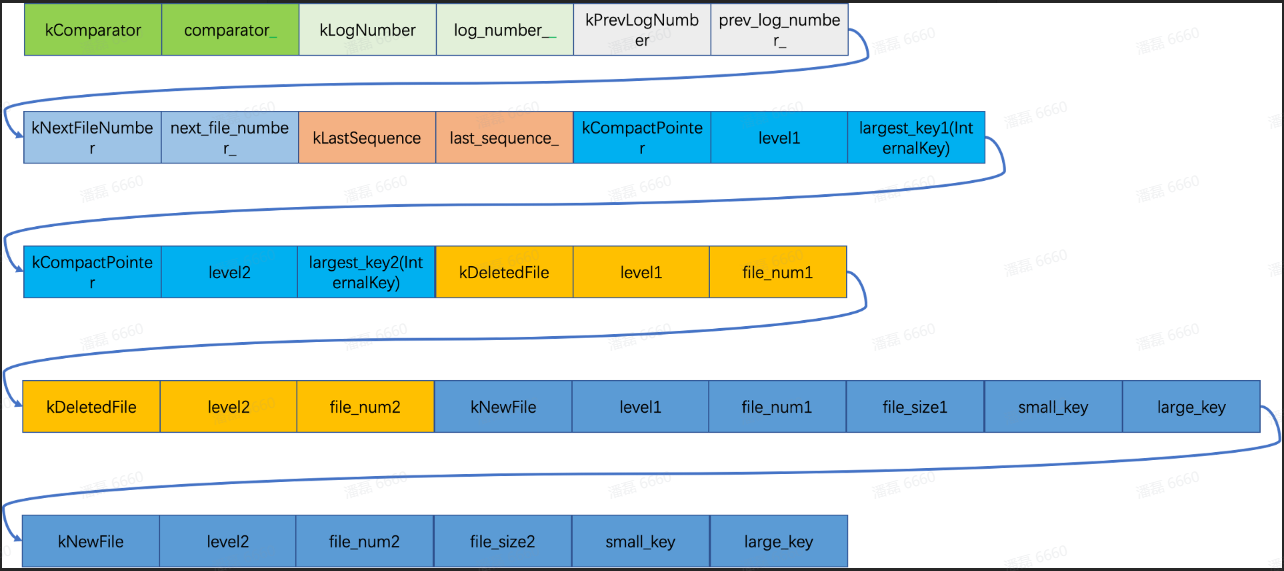
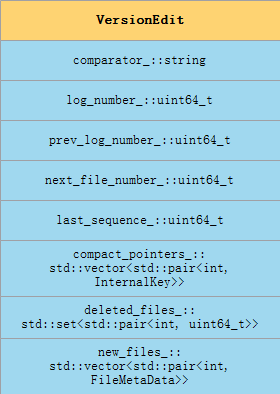
Level\_1是Index Block迭代器，Level\_2是指向DataBlock迭代器，通过这样的设计可以实现对SSTable的所有key进行向前、向后扫描的批量查询工作。



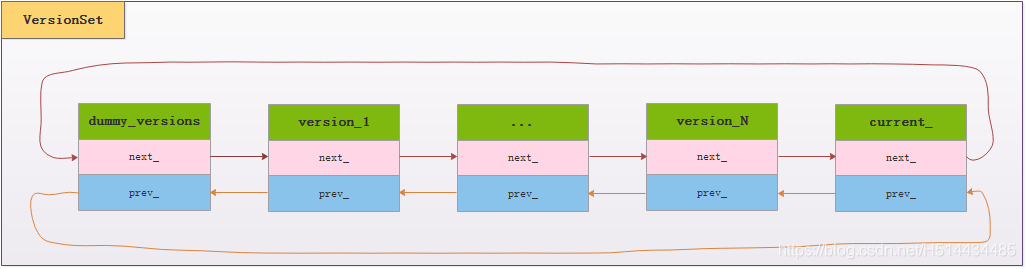
# **VersionEdit:**

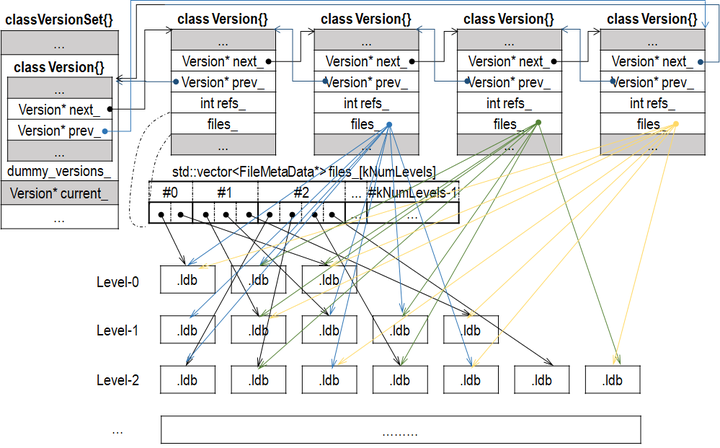
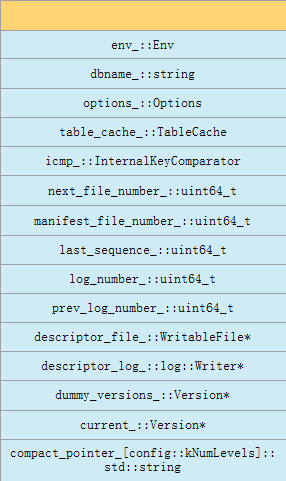
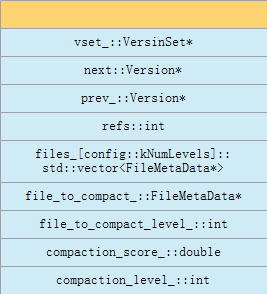
compact过程中会有一系列改变当前Version的操作（FileNumber增加，删除一些SSTable，增加一些新的SSTable），为了缩小Version切换的时间点，将这些操作封装成VersionEdit，待compact完成时，将VersionEdit一次应用到当前Version即可得到最新状态的Version。

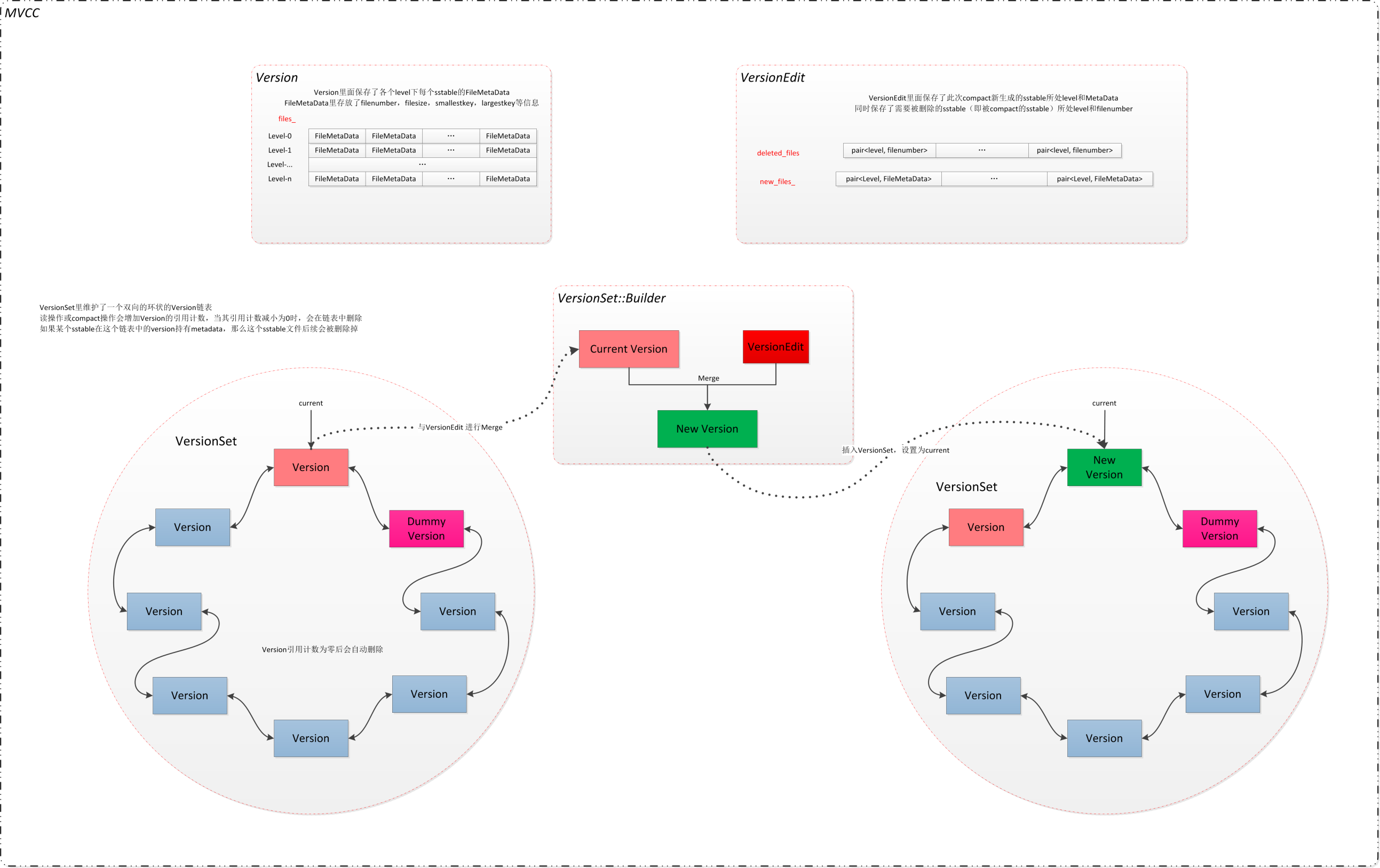


# **VersionSet:**

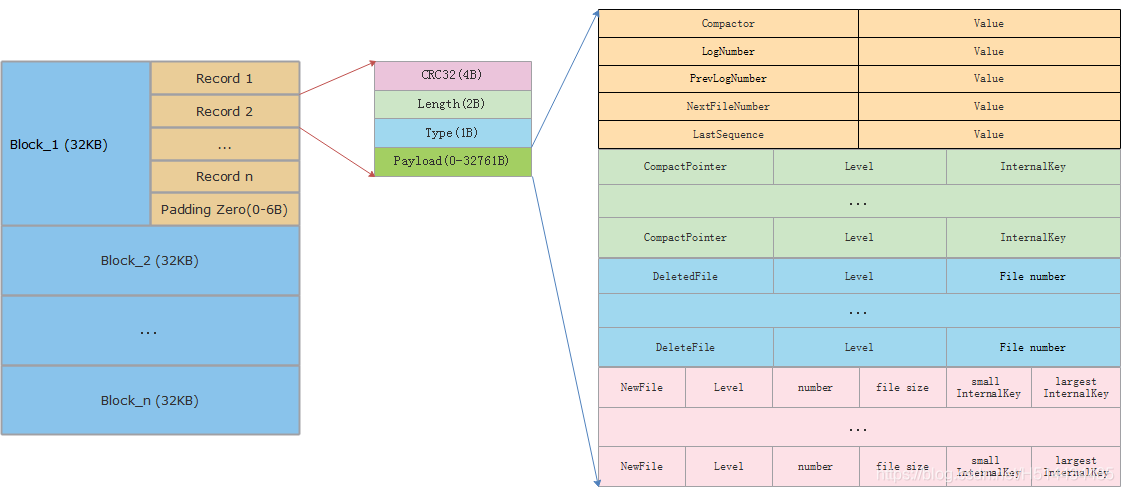




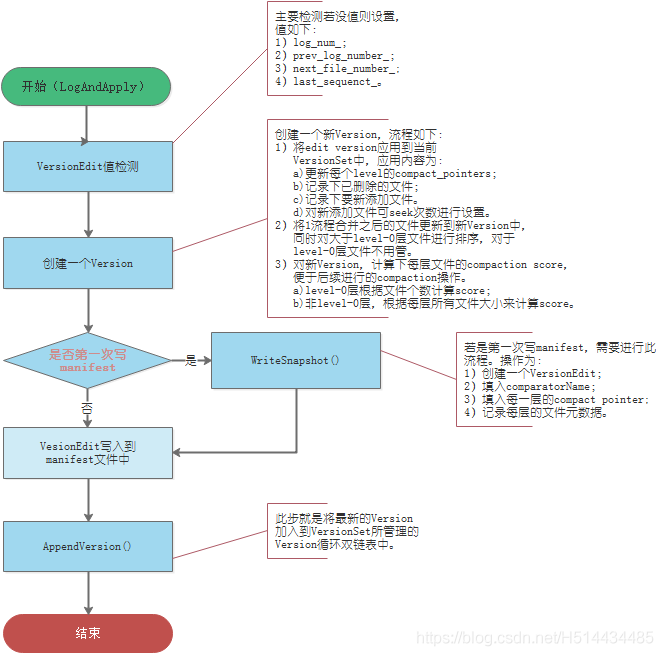


# **Manifest:**

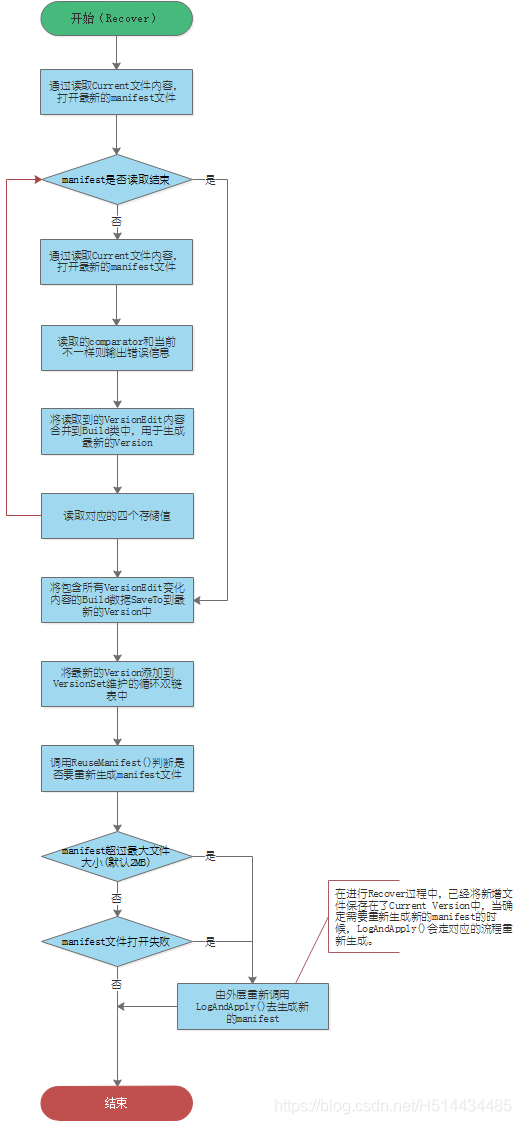
为了重启leveldb后可以恢复到退出前的状态，需要将db中的状态保存下来，这些信息就保存在Manifest中，而CURRENT文件保存了Manifest文件的名称。manifest中不仅保存当前的状态，也会将历史的状态也保存起来，考虑到每次状态的完全保存需要空间和耗费的时间比较多，当前采用的方式是，只在manifest开始保存完整的状态信息（通过VersionSet::WriteSnapShot()来实现），接下来只保存每次compact产生的VersionEdit。实际上，Manifest文件保存的元信息与Log记录格式相同，都是以块为单位，元信息的内容为VersionEdit结构编码之后生成的字符串，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：**

