

**浅析开源项目之LevelDB**

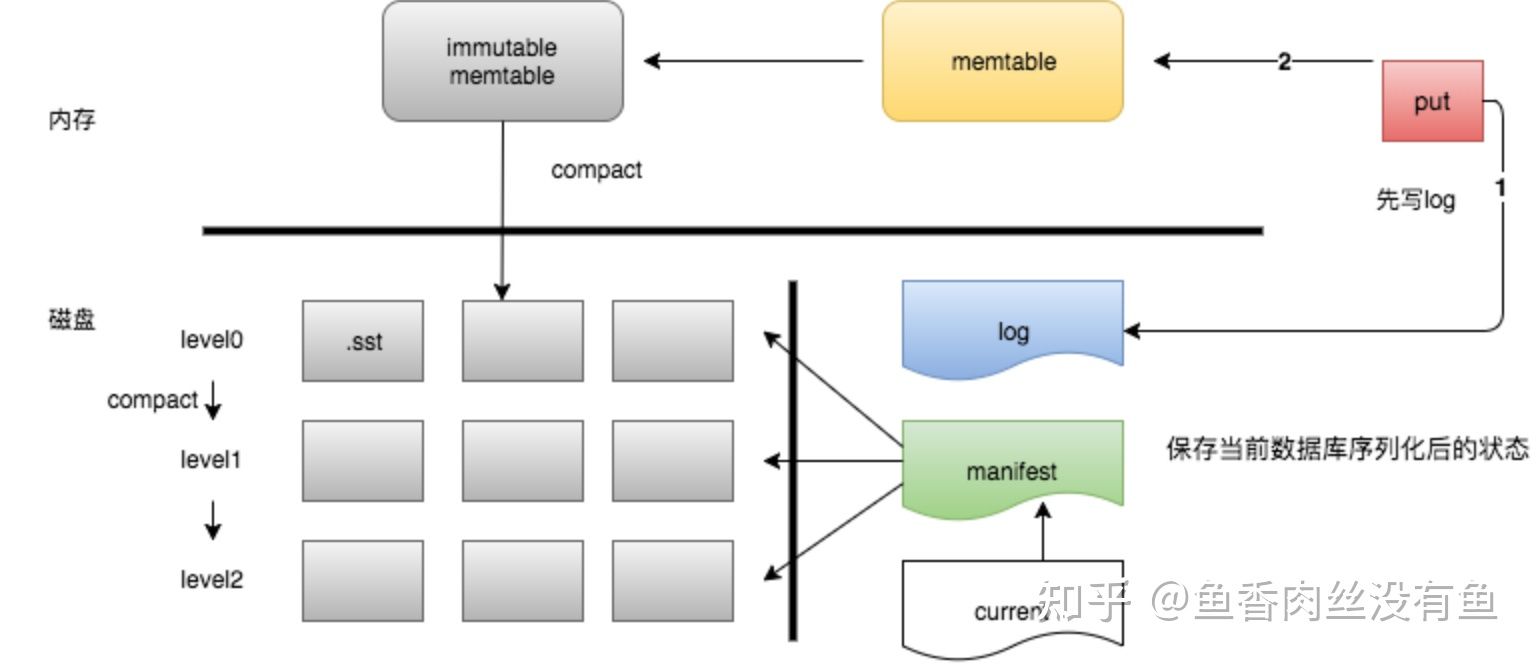
**前言**

LevelDB是一个单机持久化的KV存储引擎，通常用在分布式KV、分布式数据库等领域，本文简要介绍LevelDB的整体架构以及涉及的重要模块，使读者对LevelDB有一个大致清晰的认识。

**目录**

* 架构设计
* 重要类图
* 数据结构
  + MemTable
  + WAL
  + SSTable
  + Manifest
* IO流程
  + Open流程
  + Put流程
  + Get流程
* Compaction
  + Minor Compaction
    - 触发时机
    - 执行过程
    - 输出
  + Major Compaction
    - 触发时机
    - 执行过程
    - Pick SSTable
    - 输出
  + Manual Compaction
    - 触发时机
    - 执行过程
    - 输出
* 版本控制

**架构设计**



LevelDB整体由以下6个模块构成：

**MemTable：**KV数据在内存的存储格式，由SkipList组织，整体有序。

**Immutable MemTable：**MemTable达到一定阈值后变为不可写的MemTable，等待被Flush到磁盘上。

**Log：**有点类似于文件系统的Journal，用来保证Crash不丢数据、支持批量写的原子操作、转换随机写为顺序写。

**SSTable：**KV数据在磁盘的存储格式，文件里面的key整体有序，一旦生成便是只读的，L0可能会有重叠，其他层sstable之间都是有序的。

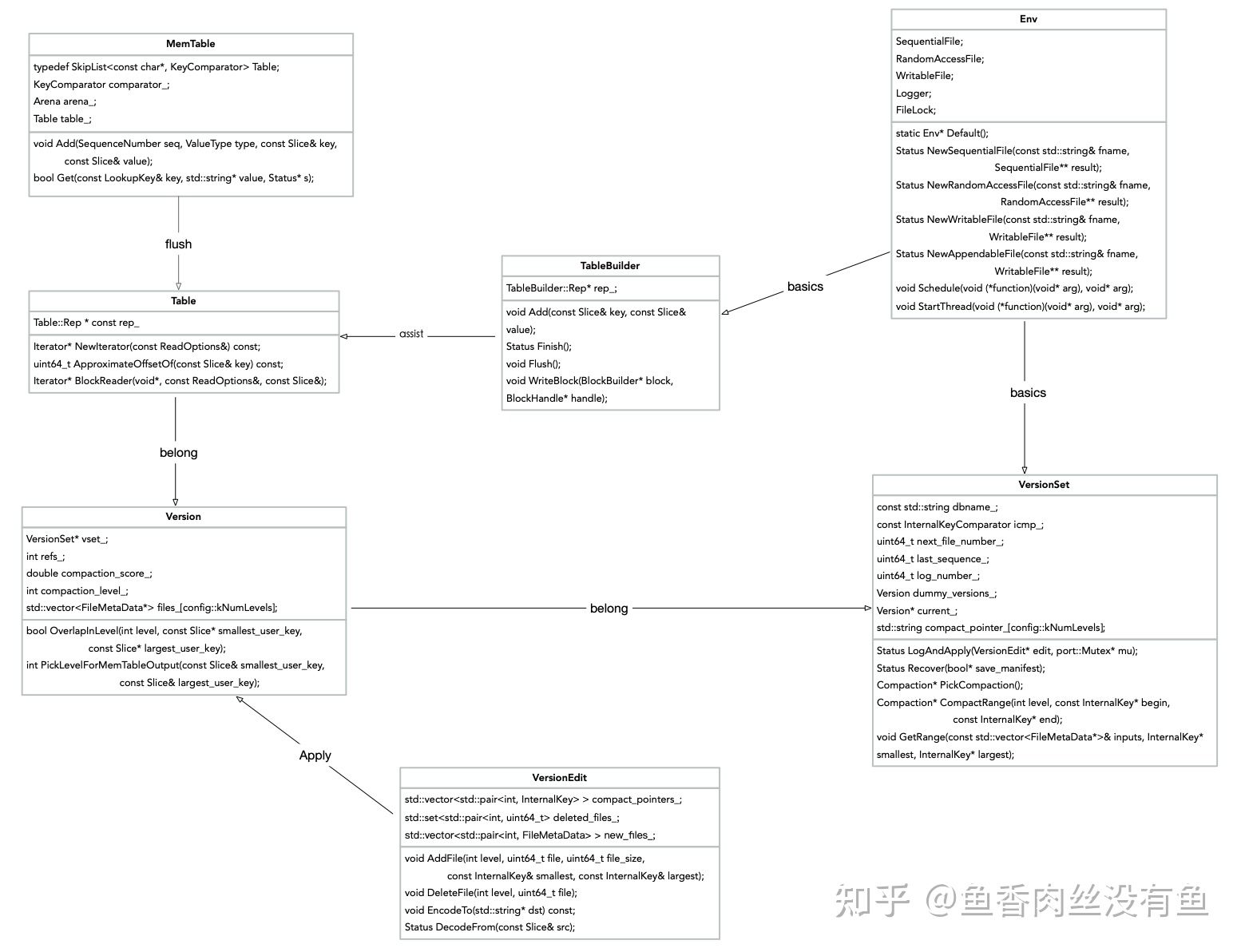
**Manifest：**增量的保存DB的状态信息，使得重启或者故障后可以恢复到退出前的状态。

**Current：**记录当前最新的Manifest文件。

**相关概念**

* **Env：**leveldb将操作系统相关的操作(文件、线程、时间)抽象成Env，用户可以实现自己的Env(BlueRocksEnv)，灵活性比较高。
* **SequenceNnumber：**leveldb的每次更新都会携带一个版本，由递增的DB唯一的SequenceNnumber标识。
* **Snapshot：**本质上一个SequenceNnumber，用来给整个DB打快照，使用双向循环链表保存多个snapshot。
* **Comparator：**key排序的比较方法，默认按照字节比较，用户可传入自己的比较方法。
* **Slice：**存放KV数据的容器，类似于Cellar的data\_entry和ceph的bufferlist。
* **FileMetaData：**sstable的元信息，包括文件大小、smallest\_key，largest\_key等。
* **WAL Block：**WAL文件和Manifest文件使用的block组织形式。
* **Block：**sstable的数据组织粒度，多个kv会聚合成block一次写入，读取时也是按照block粒度读取。
* **BlockHandle：**block的元信息，记录位于sstable的 offset/size。
* **BlockBuilder：**负责生成block。
* **TableBuilder：**负责生成sstable。

**重要类图**



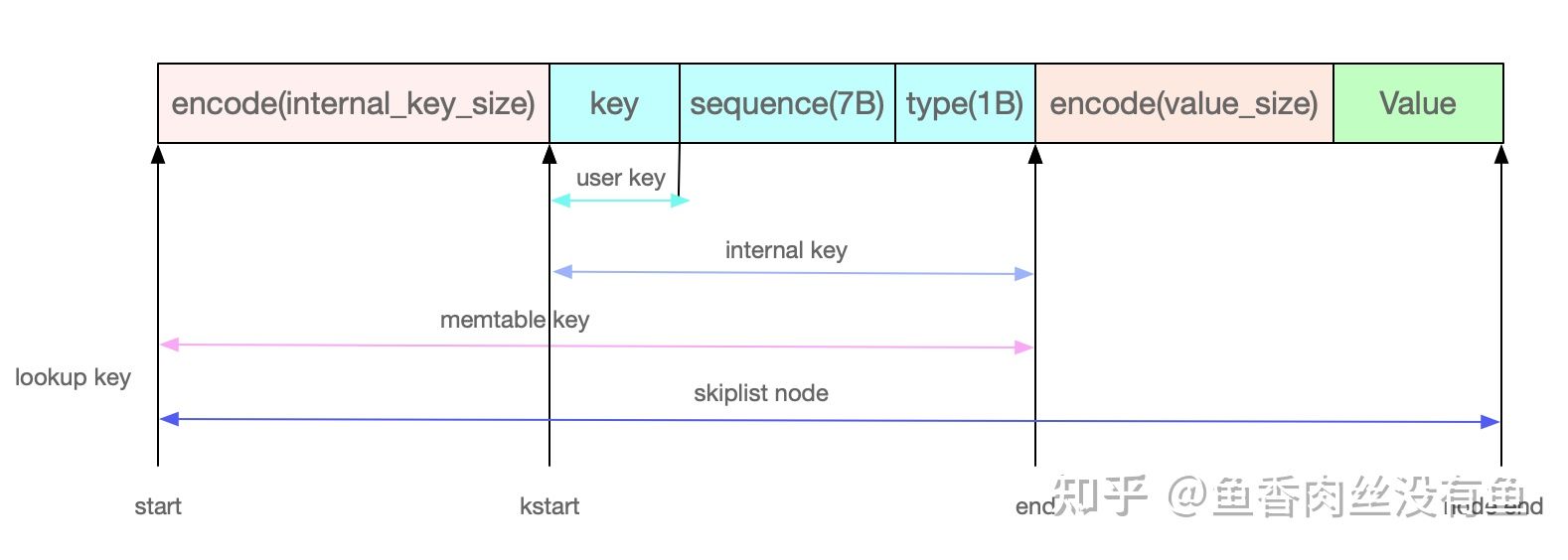
**数据结构**

**MemTable**

MemTable以及Immutable MemTable是KV数据在内存中的存储格式，底层数据结构都是SkipList，插入查找的时间复杂度都是Olog(n)。

MemTable的大小通过参数write\_buffer\_size控制，默认4MB，最多5MB dump(最大batch size为1MB)成SSTable。

当一个MemTable大小达到阈值后，将会变成Immutable MemTable，同时生成一个新的MemTable来支持新的写入，Compaction线程将Immutable MemTable Flush到L0上。所以在LevelDB中，同时最多只会存在两个MemTable，一个可写的，一个只读的。



**SkipList Node：**

由于SkipList是链表形式的，所以我们需要把KV数据的映射形式转换成该形式。如上图所示，[start, node\_end]区间就代表一个SkipList Node。

**Lookup Key：**

Loopup Key是LevelDB里面经常使用的查找Key，包含user\_key、internal\_key、memtable\_key。

1. UserKey：最直接简单的client传入的key。
2. InternalKey：UserKey+SequenceNumber(7B)+ValueType(1B)，SSTable用的key。
3. MemTableKey：InternalKeyLength+InternalKey，MemTable用的key。

**Key比较：**

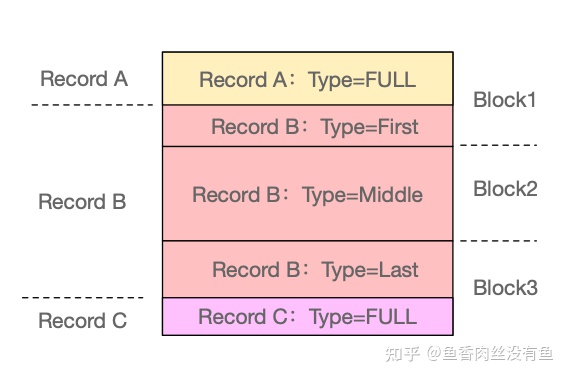
1. 从MemTableKey中解析出InternalKey。
2. 先比较UserKey，akey < bkey 则返回-1。若UserKey相等则比较SequenceNumber。
3. 解析出逆序排序的SequenceNumber，越大数据越新即：akey\_seq\_num > bkey\_seq\_num 则返回-1，更快的找到数据。
4. 递增排序示例： keya10--->keya8--->keya6--->keyb5--->keyb3--->keyc1。

**WAL**

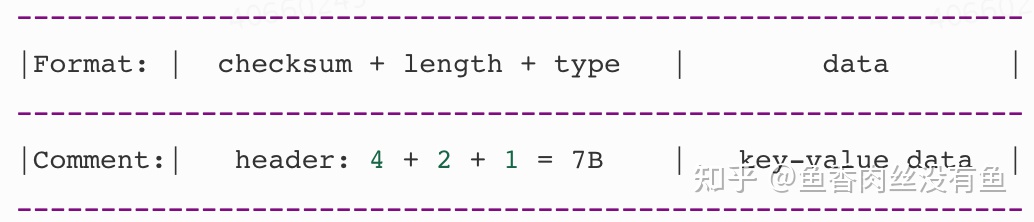
WAL即Log，每次数据都会先顺序写到Log中，然后再写入MemTable，可以起到转换随机写为顺序写以及保证Crash不丢数据的作用。

一个完整的Log由多个固定大小的block组成，block大小默认32KB；block由一个或者多个record组成。

**WAL Format：**



**Record Format:**



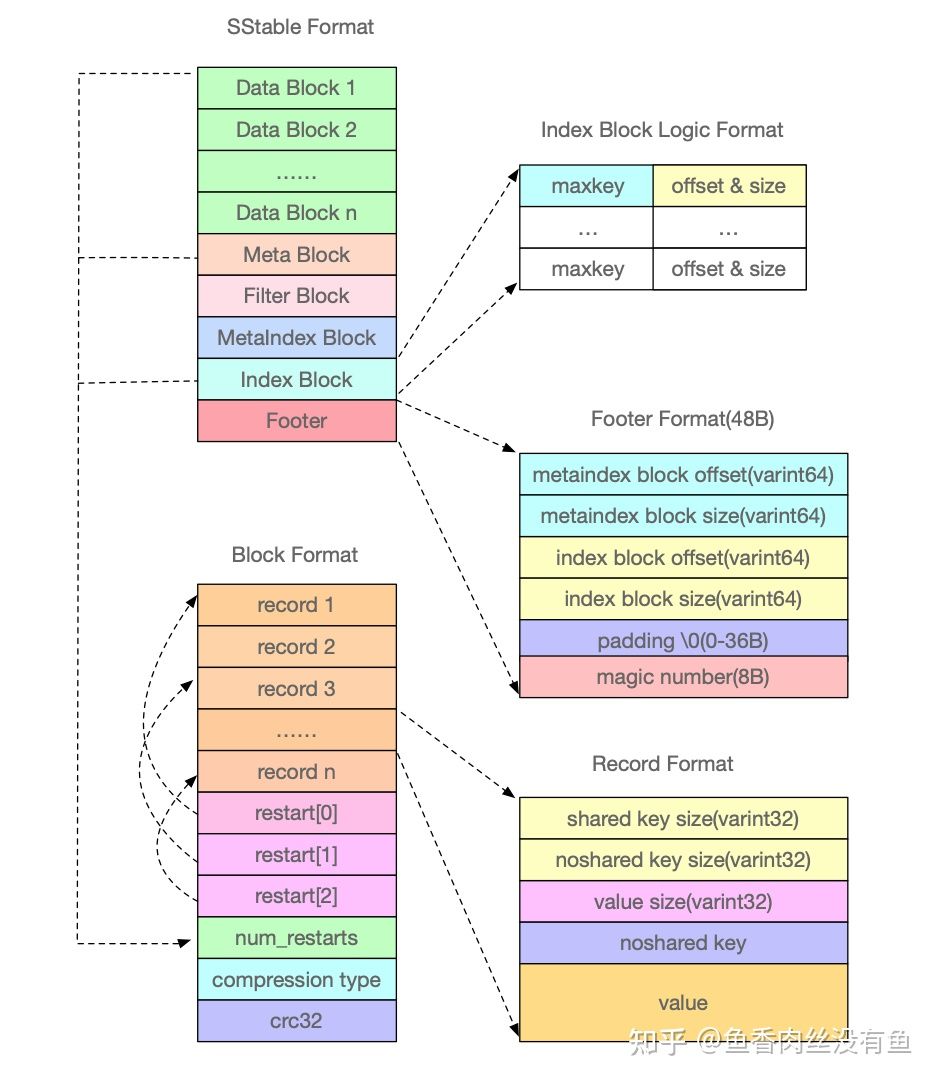
* checksum：计算type和data的crc。
* length：data的长度，2Byte可表示64KB，而block为32KB，刚好够用。
* type：一个record可以在一个或者跨越多个block，类型有5种：FULL、First、Middle、Last、Zero(预分配连续的磁盘空间用)。
* data：用户的kv数据。

**读写概要：**

1. 如果block剩余空间小于kHeaderSize(7B)，则填充0。
2. 如果block剩余空间等于kHeaderSize(7B)，则仅仅写入header，不写入数据。
3. header使用小端序存储。
4. PosixWritableFile类有一个64KB的WriteBuffer，先写入该buf，写满则Flush，然后剩余的字节数如果大于buf总大小直接全部写磁盘，否则写buf。
5. 每写一个record或者block写满就要Flush数据，注意Flush的语义仅仅是调用系统调用写入数据到PageCache。
6. 读取record时如果发现crc不一致，则report Corruption错误。
7. 只有在DB启动Replay时才会读取Log且每次读取kBlockSize(32KB)的数据。

**SSTable**

SSTable整体的格式如下图所示：



* DataBlock：存储实际的kv data、type、crc。
* MetaBlock：暂时没有使用，不过可将Filter Block当成一种特殊的MetaBlock。
* MetaIndexBlock：保存MetaBlock的索引信息，目前仅有一行KV数据，记录了FilterBlock的name以及offset/size。
* IndexBlock：保存每个DataBlock的LastKey和在SST文件中的offset/size。
* Footer：文件末尾固定长度的数据，保存MetaIndexBlock、IndexBlock的索引信息。

SSTable中的BlockSize大小默认为4K，MetaIndex、DataBlock、IndexBlock都是使用同样的BlockBuilder来构建Block，区别是里面的KV数据不同。

DataBlock中的KV是有序存储的，相邻的key之间很有可能重复，因此采用前缀压缩来存储key，后一个key只存储与前一个key不同的部分。

然后重启点指出的位置就表示该key不按前缀压缩，而是完整存储该key。对于MetaBlock和IndexBlock来说由于相邻key差距比较大，所以不开启前缀压缩，即block\_restart\_interval为1。

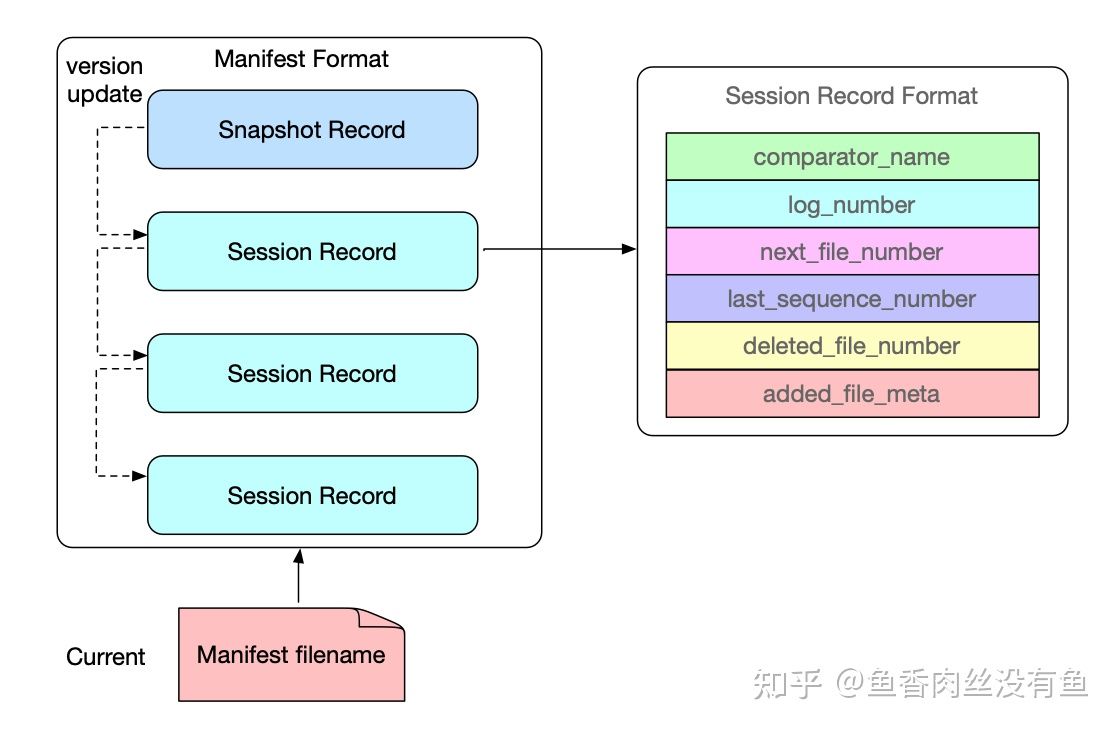
**TableCache：**缓存SST文件的元信息，包含文件fd、metaindex\_block、index\_block、filter\_block等，默认缓存1000个SSTable文件的元信息，可通过options.max\_open\_files指定。

**BlockCache：**缓存未压缩的data block数据，默认8MB，16个shard，可通过options.block\_cache指定大小，读取的时候也可以指定是否放到cache中。

**Manifest**

Manifest文件以增量的方式持久化版本信息，DB中可能包含多个Manifest文件，需要Current文件来指向最新的Manifest。

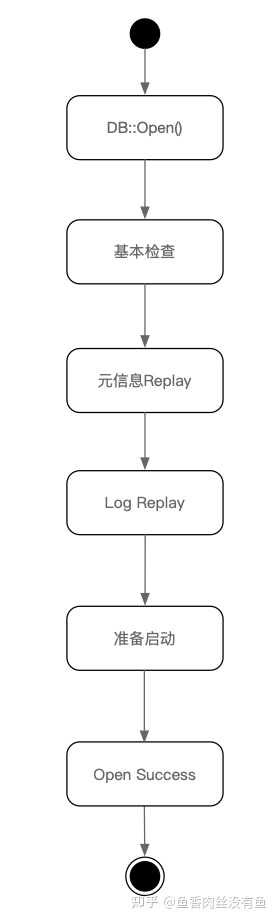
Manifest包含了多条Record，第一条是Snapshot Record，记录了DB初始的状态；之后的每条Record记录了从上一个版本到当前版本的变化，具体格式如下图：



每次做完Minor Compaction、Major Compaction或者重启Replay日志生成新的Level0文件，都会触发版本变更。

**IO流程**

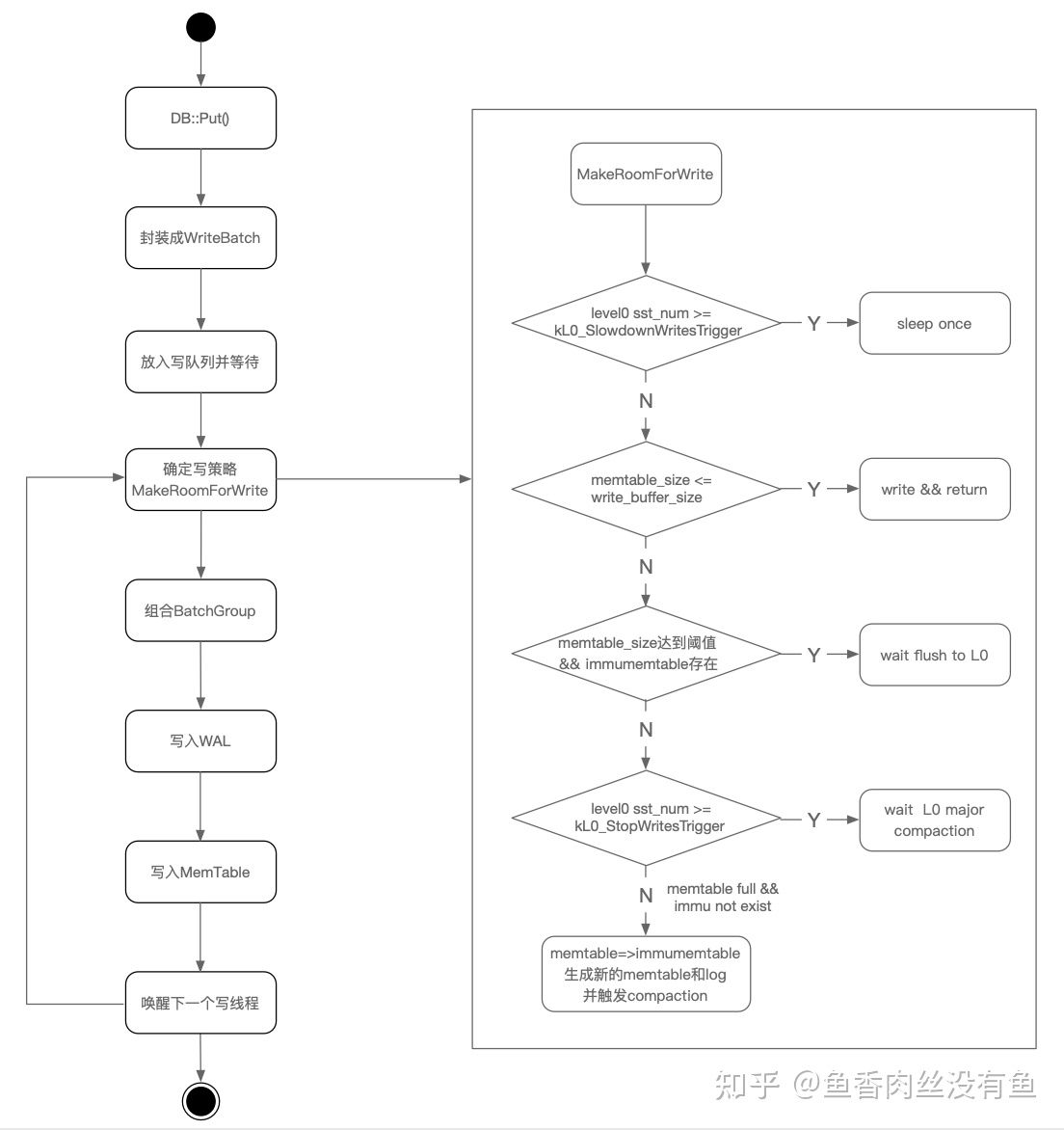
**Open流程**



1. DB::Open()
2. 基本检查
   1. 对DB Lock文件加锁确保一份数据只能启动一个DB实例。
   2. 根据option传入的create\_if\_missing/error\_if\_exists参数做不同的处理。
3. 元信息Replay
   1. 从Current文件获取当前的Manifest文件。
   2. 从Manifest文件依次读取并解析每个record Apply到builder。
   3. 创建当前唯一的Version并调用builder的SaveTo方法保存信息到当前Version。
   4. 调用Finalize()计算下次compaction要处理的level。
   5. 判断是否需要Reuse Manifest，减少DB的启动时间。
   6. 检查从Manifest解析的最终状态的基本信息是否完整并应用到当前DB状态。
   7. DB已恢复到上次退出的状态。
4. Log Replay
   1. 遍历DB中的Log文件，根据LogNumber找到需要replay的Log。
   2. 遍历Log中的record重建MemTable，并且MemTable达到阈值时就dump成SSTable。
   3. 将最后的MemTable dump成SSTable。
   4. 根据Log的FileNumber和遍历record的SequenceNumber修正从Manifest获取的值。
5. 准备启动
   1. 生成新的Log文件并更新DB元信息，调用LogAndApply方法。
   2. 删除无用文件，尝试触发compaction。
6. Open Success。

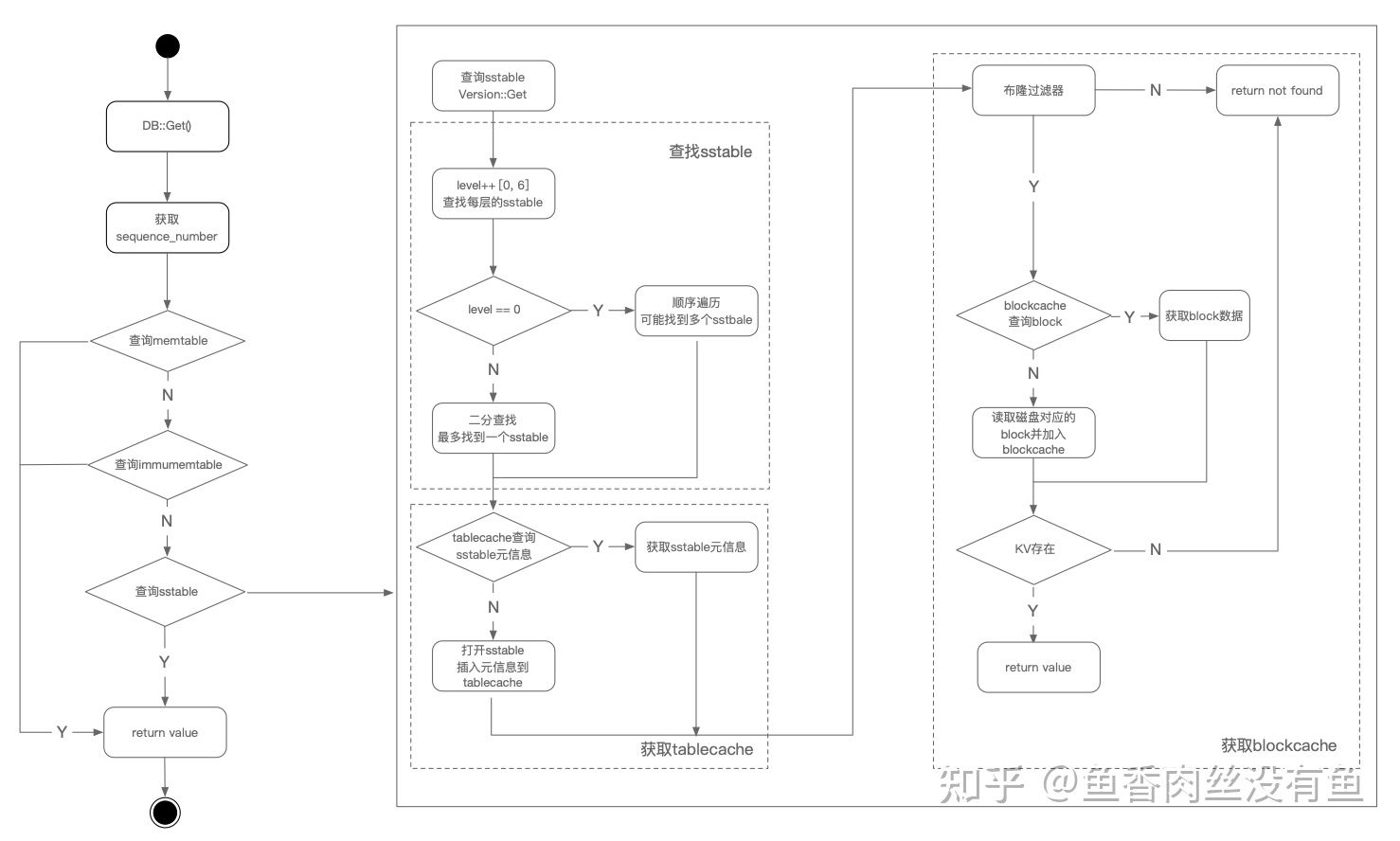
**Put流程**

1. 先写Log，再写MemTable，通过WriteOptions的sync参数控制是否Log落盘。
2. 无论是单个写入还是批量写入都会封装成WriteBatch接口，写入string类型的变量里面。
3. Put/Delete的每个key都会有一个唯一的全局递增的SequenceNumber(7 Bytes)。
4. 组合BatchGroup时阈值为1MB，如果size小于128K，则阈值为size+128K，减少小写延迟。



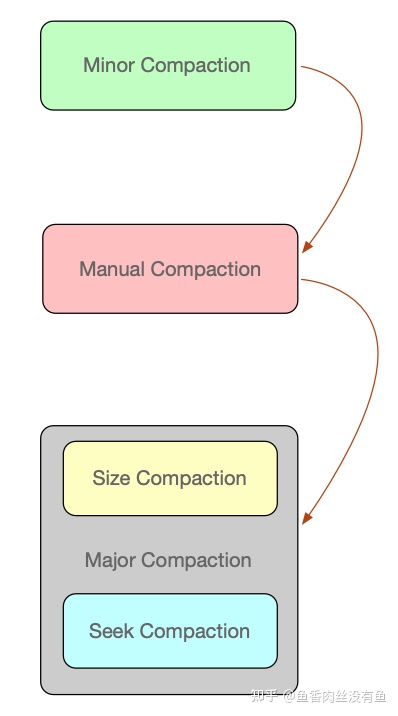
**Get流程**

1. 先从VersionSet中获取当前的SequnceNumber(如果指定snapshot则使用snapshot的sn)。
2. 读取MemTable再读取ImmutableMemTable，如果没有找到，则从磁盘上依次读取Level0~LevelN。
3. 查询Level时先查找符合的sstable，再获取sstable的tablecache、最后获取blockcache。
4. 由于Level0之间的SST文件可能会有Key重叠，Level1~N之间的SST文件不会有Key重叠，所以查找sstable时L0需要遍历，其他Level二分查找。



**Compaction**

LevelDB的写入和删除都是追加写WAL，所以需要Compaction来删除那些重复的、过期的、待删除的KV数据，同时也可以加速读的作用，其类型和优先级如下图所示：



**compaction类型：**

LevelDB中有三类Compaction：

1. minor compaction：immutable memtable持久化为sstable。
2. major compaction：sstable之间的compaction，多路归并排序。
3. manual compaction：外部调用CompactRange产生的Compaction。

其中major compaction有两类：

1. size compaction：根据level的大小来触发。
2. seek compaction：每个sstable都有一个seek miss阈值，超过了就会触发。

**compaction优先级：**

LevelDB在MaybeScheduleCompaction/BackgroundCompaction中完成对compaction优先级的调度。

具体优先级为：minor > manual > size > seek。

1. 如果immutable memtable不为空，则dump到L0的sstable。
2. 如果is\_manual为true即manual compaction，则调用CompactRange。
3. 最后调用PickCompaction函数，里面会优先进行size compaction，再进行seek compaction。

**Minor Compaction**

minor compaction将immutable memtable持久化为sstable，我们主要关注何时触发以及生成的sstable需要放在哪一层。

**触发时机**

Write(Put/Delete)、CompactRange、Recovery以及compaction之后都会触发minor compaction，最频繁触发的操作还是Write操作。

**执行过程**

当immutable memtable持久化为sstable的时候，大多数情况下都会放在L0，然后并不是所有的情况都会放在L0，具体放在哪一层由PickLevelForMemTableOutput函数计算。

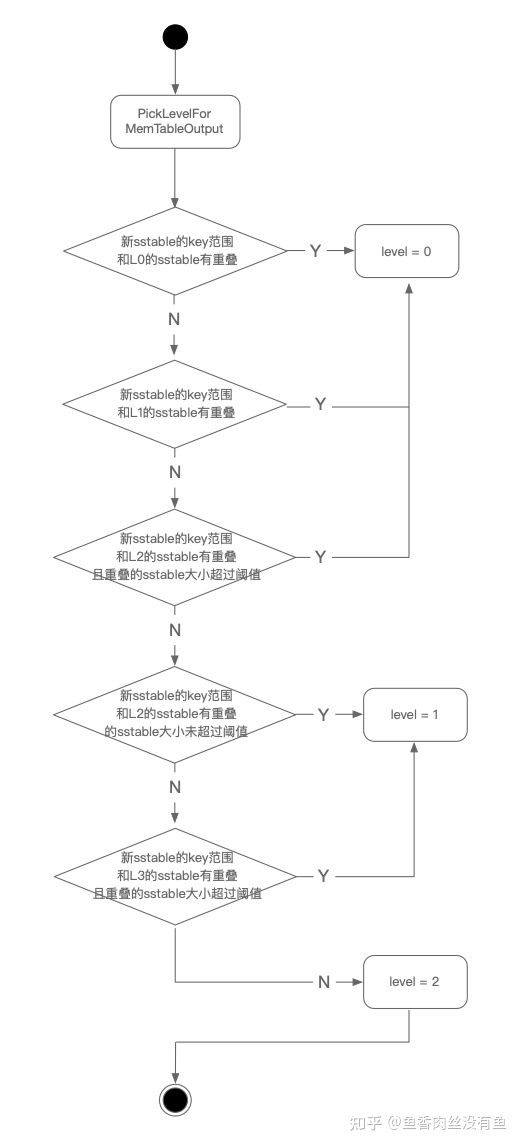
理论上应该需要将dump的sstable推至高level，因为L0文件过多会导致**查找耗时增加**以及**compaction时内部IO消耗严重；**

但是又不能推至太高的level，因为需要控制查找的次数，而且某些范围的key更新频繁时，往高level compaction**内部IO消耗严重**，而且也不易compaction到高level，导致**空间放大严重**。

所以PickLevelForMemTableOutput在选择输出到哪个level的时候，需要权衡查找效率、compaction IO消耗以及空间放大，大体策略如下：

1. 最高可推至哪层由kMaxMemCompactLevel控制，默认最高L2。
2. 如果dump成的sstable和L0/L1有重叠，则放到L0。
3. 如果dump成的sstable和L2有重叠且重叠sstable总大小超过 10 \* max\_file\_size，则放在L0。
   1. 因为此时如果放在L1会造成compaction IO消耗比较大。
   2. 所以放在L0，之后和L1的sstable进行compaction，减小sstable的key范围，从而减小下次compaction涉及的sstable总大小。
4. 如果dump成的sstable和L3有重叠且重叠sstable总大小超过 10 \* max\_file\_size，则放在L1。

具体PickLevelForMemTableOutput的策略如下图：



**输出**

minor compaction生成的sstable不受 max\_file\_size(default 2MB)的限制，通常为write\_buffer\_size的大小。

**Major Compaction**

major compaction是LevelDB compaction中最复杂的部分，主要包含size\_compaction和seek\_compaction，会进行重复数据、待删除的数据的清理，减少空间放大，提高读效率。

**Seek Compaction**

每个sstable都有一个allowed\_seek的初始化阈值，表示允许seek\_miss多少次；每当get miss 的时候都会减1，当减为0的时候标记为需要compaction的文件，参与compaction，从而避免不必要的seek miss消耗IO。但是引入了布隆过滤器之后，查找miss消耗的IO就会小很多，seek compaction的作用也大大减小。

**接下来主要介绍Size Compaction**

**触发时机**

1. DB Open时会触发compaction。
2. Write(Put、Delete)会检查是否需要触发size compaction。
3. Get时会检查是否需要触发seek compaction。
4. compaction之后会检查是否需要再次触发compaction。

**执行过程**

1. 调用versions\_->PickCompaction()函数获取需要参加compaction的sstable。
2. 如果不是manual且可以TrivialMove，则直接将sstable逻辑上移动到下一层。
   1. 当且仅当level\_n的sstable个数为1，level\_n+1的sstable个数为0，且该sstable与level\_n+2层重叠的总大小不超过10 \* max\_file\_size。
3. 获取smallest\_snapshot作为sequence\_number。如果有snapshot则使用所有snapshot中最小的sequence\_number，否则使用当前version的sequence\_number。
4. 生成MergingIterator对参与compaction的sstable进行多路归并排序。
5. 依次处理每对KV，把有效的KV数据通过TableBuilder写入到level+1层的sstable中。
   1. 期间如果有immu memtable，则优先执行minor compaction。
   2. 重复的数据直接跳过，具体细节处理如下：
      1. 如果有snapshot，则保留大于smallest\_snapshot的所有的record以及一个小于smallest\_snapshot的record。
      2. 如果没有snapshot，则仅保留sequence\_number最大的record。
   3. 有删除标记的数据则判断 level i+2 以上层有没有该数据，有则保留，否则丢弃。
6. InstallCompactionResults将本次compaction产生的VersionEdit调用LogAndApply写入到Manifest文件中，期间会创建新的Version成为Current Version。
7. CleanupCompaction以及调用DeleteObsoleteFiles删除不属于任何version的sstable文件以及WAL、Manifest文件。

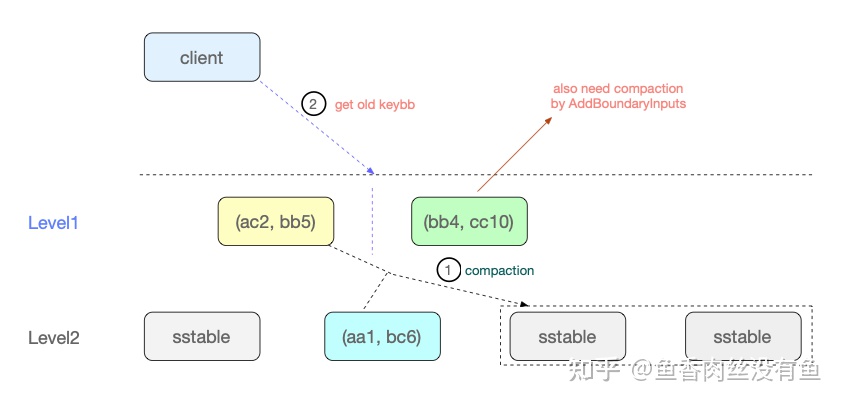
**Pick SSTable**

**1、选取出最急需进行compaction的level。**

* 1. 为了选择出最紧急compaction的level，每层都会有一个score值，在VersionSet::Finalize函数中计算。
  2. 然后选取score值最大且大于等于1的 level进行major compaction。
  3. level\_0 score = L0\_current\_sstable\_number / L0\_compaction\_trigger(4)
  4. level\_x score = Lx\_total\_file\_size / max\_bytes\_for\_Lx
  5. 每层阈值：L0 = 10MB，L1 = 10MB，Ln = 10^n MB

**2、选取 level\_i上的sstable。**

* 1. 每个level都有一个string类型的compact\_pointer来判断需要从该level的那个位置开始compaction。
  2. 选择一个或者多个大于compact\_pointer的sstable参与compaction，大部分情况下是一个。
     1. 会先选择一个大于compact\_pointer的sstable参与compaction。
     2. 在此修复了一个snapshot compaction下数据不一致的BUG：[https://github.com/google/leveldb/pull/339](https://link.zhihu.com/?target=https%3A//github.com/google/leveldb/pull/339" \t "_blank)



* + 1. BUG产生：随着compaction的不断进行，在有snapshot的情况下，可能会导致每一层中有许多按照sequence number排序的user\_key相同的record，如果这些record比较多或者对应的value比较大，那么这些record就会被分散保存到相邻的sstable，从而触发这个BUG。
    2. BUG修复：会调用AddBoundaryInputs函数添加同层的有和当前选取的sstable的largest\_key的user\_key相等的其他sstable参与compaction。

*// AddBoundaryInputs 主要包含以下两步*

InternalKey largest\_key **=** FindLargestKey(compaction\_files);

FileMetaData**\*** smallest\_boundary\_file **=** FindSmallestBoundaryFile(level\_files);

*// 选取重叠的sstable的判断条件*

**if** (icmp.Compare(f**->**smallest, largest\_key) **>** 0 **&&**

user\_cmp**->**Compare(f**->**smallest.user\_key(), largest\_key.user\_key()) **==** 0) {

}

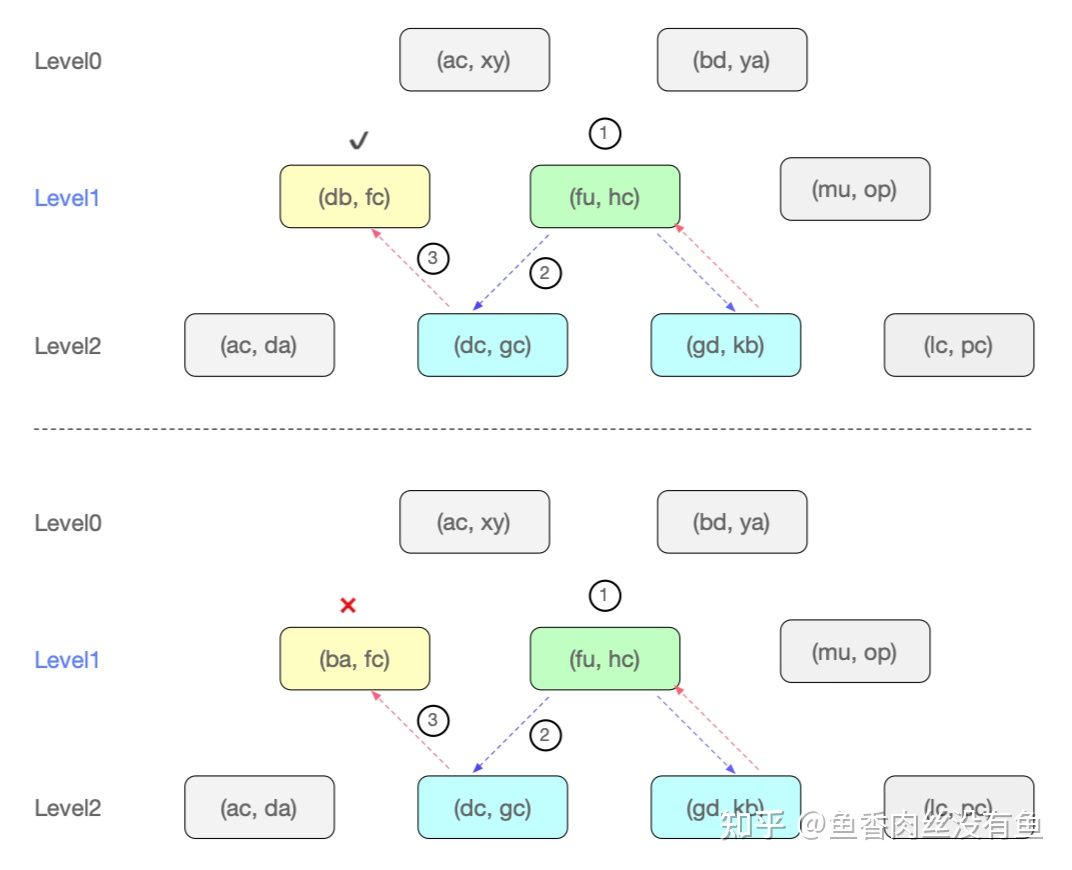
1. 如果是level0，还需要选取所有与当前sstable重合的sstable参与compaction，因为L0允许sstable之间重叠。

**3、选取 level i+1上的sstable。**

根据 level i上选取出的sstable，确定其[smallest, largest]，然后选出 level i+1 上与其有重叠的所有sstable。

**4、是否需要扩展 level i上的sstable。**

* 1. 在已经选取的 level i+1的sstable数量不变的情况下，尽可能的增加 level i 中参与compaction的sstable数量。
  2. 总的参与compaction的sstable的大小阈值为 25 \* max\_file\_size。
  3. 计算出 level i 和 level i+1 的[smallest, largest]，然后计算出和level i 上有哪些sstable重叠，如果 level i 上新增的sstable不会与 level i+1 上的非compaction的sstable重叠，则加入此次compaction。



**输出**

level i 和 level i+1 上的sstable compaction后生成的sstable放在 level i+1 上，同时生成新的version，删除之前参与compaction的sstable。

**Manual Compaction**

**触发时机**

外部调用DBImpl::CompactRange(const Slice*begin, const Slice* end)时触发manual compaction。

Slice **begin**("key\_start");

Slice **end**("key\_end");

db**->**CompactRange(**&**begin, **&**end);

*// CompactRange实现*

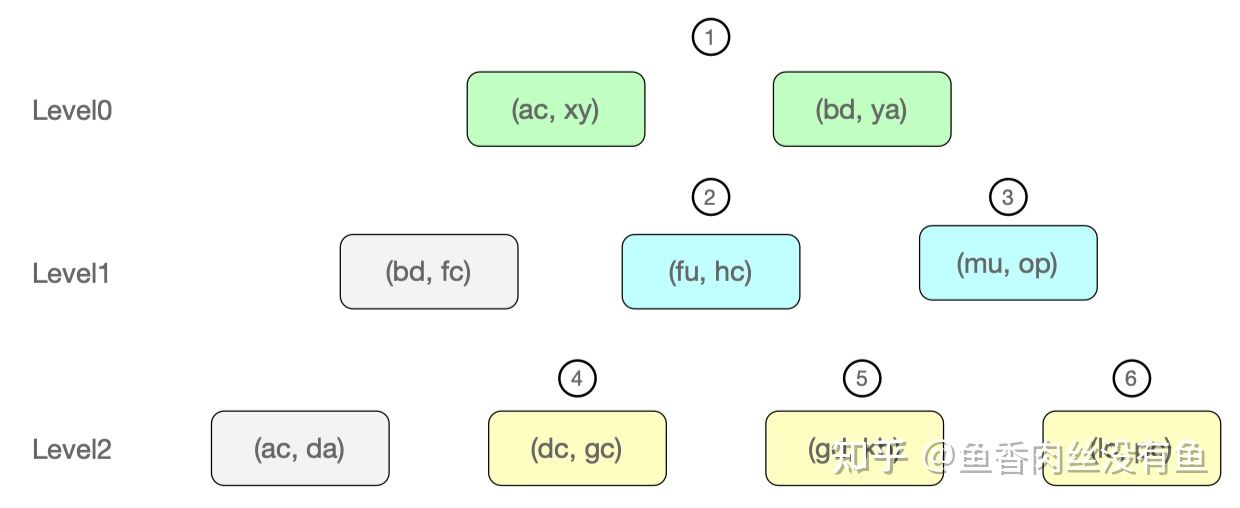
TEST\_CompactMemTable();

**for** (**int** level **=** 0; level **<** max\_level\_with\_files; level**++**) {

TEST\_CompactRange(level, **&**begin, **&**end);

}

**执行过程**



manual compaction会一个level一个level的compact所有与[begin, end]有重叠的sstable，具体流程如下：

1. 写入一条nullptr的batch等待前面的batch结束从而尝试将memtable的数据dump成sstable。
2. 从 level0 开始每层都依次执行manual compaction。
   1. 在当前level选取与range(begin, end)有重叠的所有sstable，如果重叠的sstable过多则选择的sstable总大小不超过MaxFileSizeForLevel(2MB)。
   2. 选取 level i+1 上的sstable，流程和 Major Compaction - Pick SSTable - 选取 level i+1上的sstable 一致。
   3. 具体的compaction过程和major compaction一样，执行完后更新manual compaction的begin为new\_begin，循环执行2.a，重新选择sstable做compaction。
3. 循环执行下一层的manual compaction，直到所有level执行完为止。

**输出**

同major compaction，会导致产生很多version以及sstable。

**版本控制**

LevelDB使用MVCC来避免读写冲突，相比于锁机制，提升了读写效率。版本控制的主要作用为：

1. 记录compaction之后，DB由哪些SSTable组成。
2. 记录哪些SSTable属于哪个Version。

版本控制整体由以下几部分组成：

**Version：**管理当前DB元信息以及每一层的SSTable文件集合，引用计数为0时自动从VersionSet删除。

**VersionEdit：**记录SSTable文件的变化，新增的SSTable以及删除的SSTable。

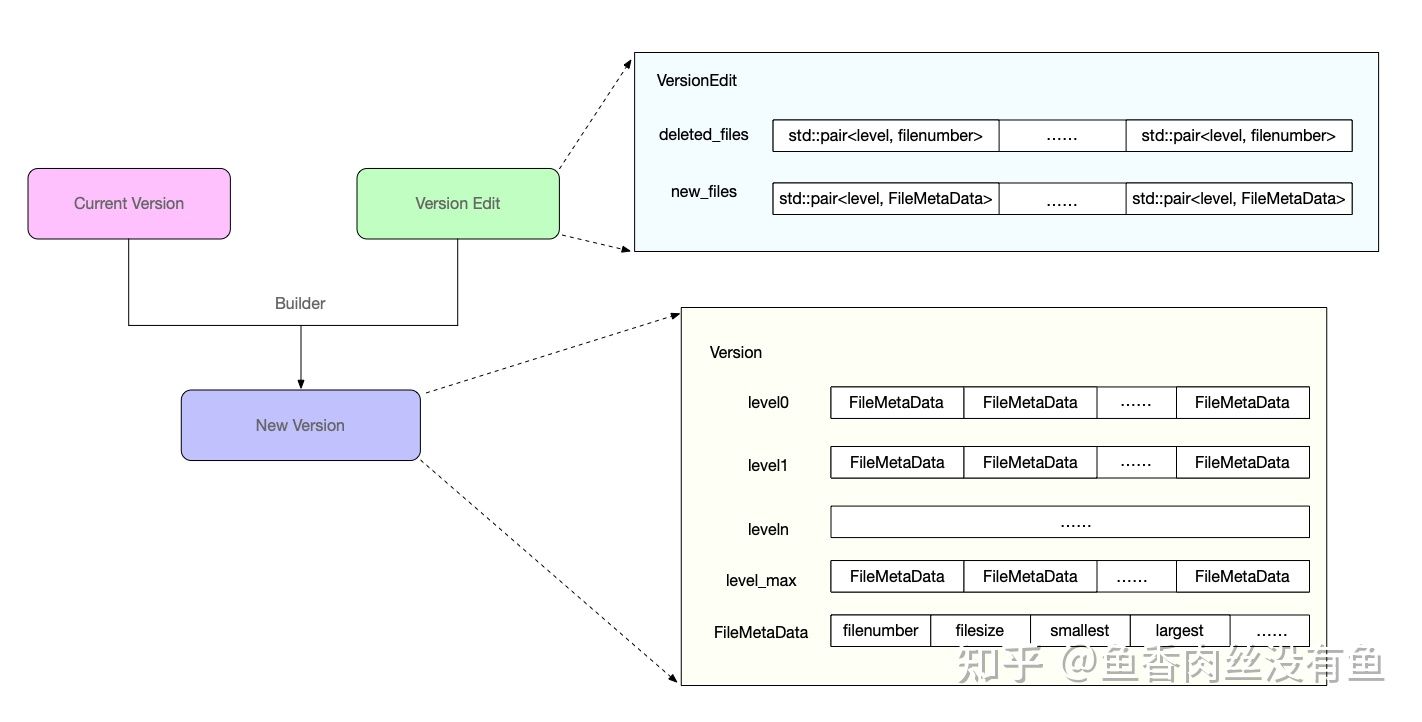
**VersionSet：**使用双向循环链表管理DB当前所有活跃的Version，最后一个节点为dummy\_version。

**Builder：**将VersionEdit应用到Version的过程封装成VersionSet::Builder。主要包含Apply(+)、SaveTo(=)两个方法。

**Manifest：**持久化当前DB的状态以及每次的VersionEdit。

**Current：**指向当前最新的Manifest文件。

**Version + VersionEdit = NewVersion**



VersionSet应用VersionEdit生成新Version主要都在**LogAndApply**函数中，流程如下：

Version**\*** v **=** **new** Version(**this**);

{

Builder **builder**(**this**, current\_);

builder.Apply(edit);

builder.SaveTo(v);

}

1. 生成新Version并应用VersionEdit。
2. 调用Finalize计算下次compaction要处理的Level。
3. 如果不复用老的Manifest文件则创建一个新的并且写入Snapshot Record。
   1. 重启时在**Recovery()/ReuseManifest()**中会判断。
   2. 如果Manifest文件大于TargetFileSize则不复用老的Manifest文件，等到LogAndApply时创建新的Manifest文件。
   3. 如果Manifest文件小于TargetFileSize则复用老的Manifest文件，在LogAndApply中就可以跳过创建Manifest文件的步骤。
4. 将VersionEdit的内容作为Session Record写入Manifest。
5. 如果新建Manifest文件则更新Current文件指向最新的Manifest文件。
6. 调用AppendVersion将最新的Version更新到VersionSet。