


# 解锁 LevelDB 的奥秘



鹅厂架构师   
已认证账号

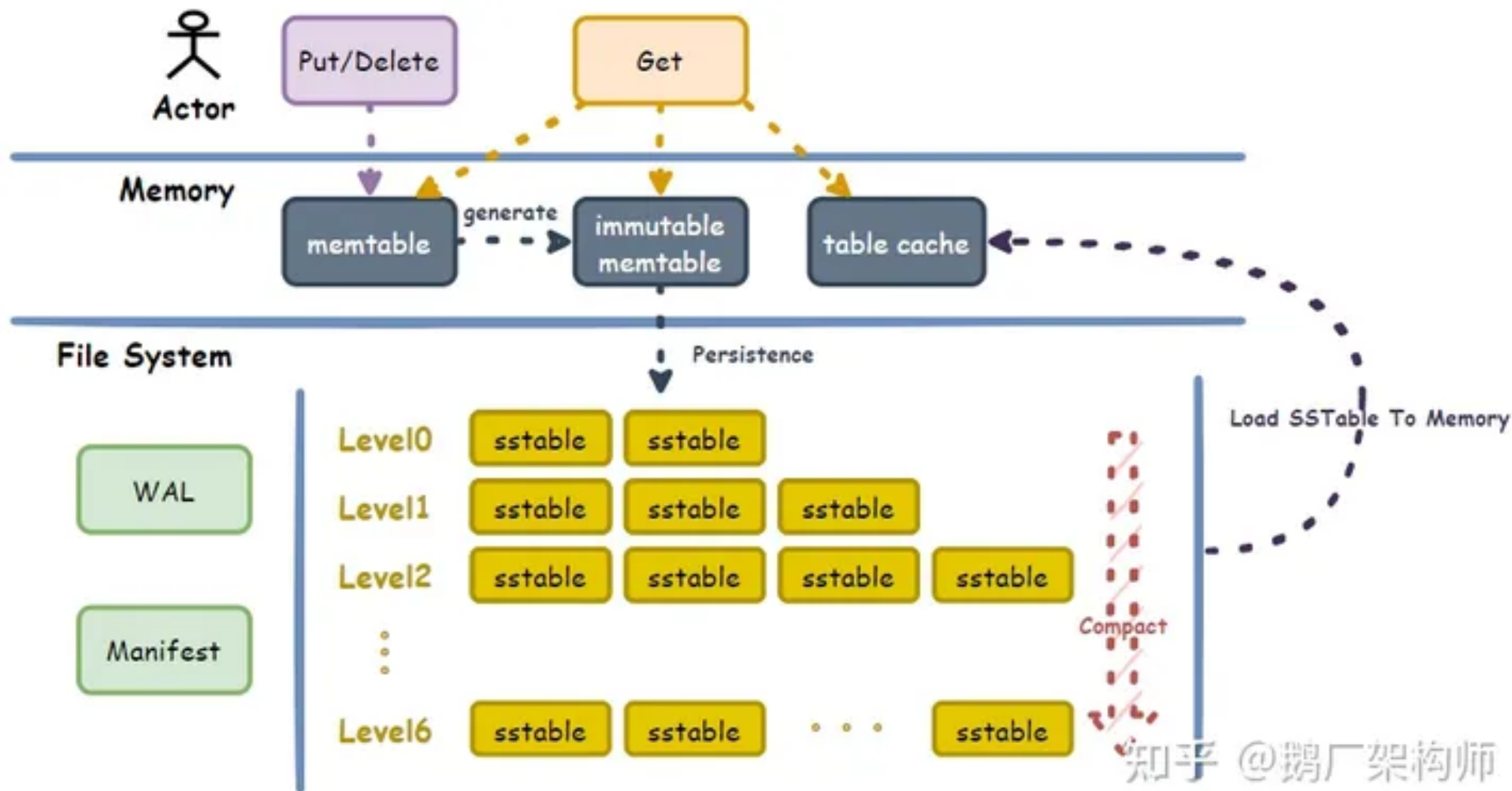
以下内容来自腾讯工程师 tmq

导语：源码不难，注释无敌。

## 一、基础架构

LevelDB 是 Google 传奇工程师 Jeff Dean 和 Sanjay Ghemawat 开源的 KV 库，具有轻量、高速、高可靠等特点，能做到 400,000/s 的写和 60,000/s 的读，相比于 MySQL 和 Redis 这些设计复杂的数据库系统，LevelDB 也许只能算是一个提供存储能力的库，它只提供了几个很基础的操作 api：Get、Put、Delete。一般情况下不会有业务直接使用 levelDB 来作为数据库存储，而是使用基于该库进行封装的数据库，比如 rocksDB、indexedDB 等。

其架构图如下：



它的架构主要分为两层：Memory 和 File System，通过对这两部分的极致设计来尽可能的提高 IO。

Memory 层有 memtable、immutable memtable 和 table cache 三大模块组成：

- memtable：用户的写操作并不会直接写入磁盘，而是记录在内存的 memtable 中，该结构由 SkipList 实现，这提供了  $O(\log n)$  的读写能力；
- immutable memtable：这是一个典型的无锁双 Buffer 理念设计，memtable 负责用户的写操作，而 immutable memtable 负责将内存的数据落地到磁盘，每次 memtable 存储的数据达到阈值时（默认是 4MB）就会生成一个新的 memtable 同时将旧的 memtable 转成 immutable memtable，接着启动异步线程开始将内存数据进行落地；
- table cache：实现上是 LRU Cache，负责从磁盘中读取数据缓存到内存中来提供高效读写能力。

Memory 层主要是给用户提供一个良好的读写能力，但是不管是 immutable memtable 还是 table cache 都依赖磁盘 IO 的能力，所以磁盘文件结构的设计也至关重要。File System 由 WAL、Manifest、SSTable 三种文件组成：

- WAL：全称 Write-Ahead Log 预写式日志，属于数据库中的常青树技术，通过先写日志再写数据来保证可靠性和一致性，几乎没有不使用该技术的数据库；
- Manifest：LevelDB 元数据日志，用于保存当前数据库版本、拥有哪些数据文件等信息；
- SSTable：存储了落地的数据，这是一个只读文件，只能做读取和删除两个动作，不能修改，其文件内容按照数据有序的存储，可以快速地进行定位和获取。

## 二、IO 设计

### LSMT

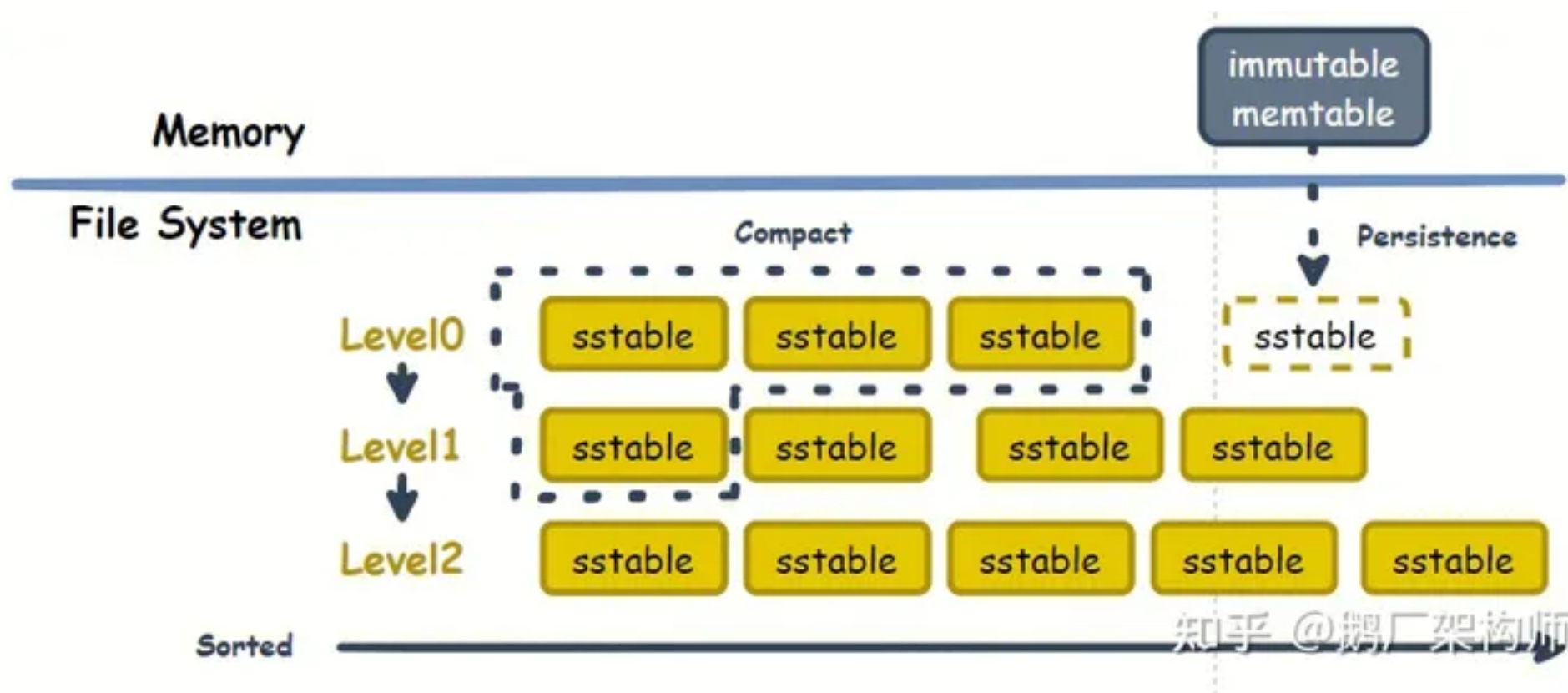
了解到磁盘性能顺序写和随机写的差异。不管是在传统的硬件磁盘上还是在高速SSD 上，顺序写比随机写一直有着很大的性能优势。

为了能够尽可能的提高数据库的写能力，LSMT 被设计而出，其全称叫 Log Structed-Merge Tree，最开始出现在 BigTable。

以往的数据库里面，我们更新一条记录时必然有一个读取然后再更新的过程，也就是记录是覆盖更新而非增量更新，比如 MySQL，虽然 MySQL 的 Change Buffer 的存在一定程度上缓解了这种情况，但是适用场景有限。这种覆盖更新会涉及两次 IO，读一次和写一次，写操作大概率会触发随机写。

而 LSMT 不走寻常路，采用增量更新，当用户更新一条记录时，只会增加一条新纪录，而不会直接删除旧记录或者更新旧记录，同时这条新纪录是以一种追加写的方式落地到磁盘文件的，这充分的利用了顺序写。

LevelDB 实现上由于 immutable memtable 会在内存中存储 4MB 的数据才会落地，也就是说一次性写了 4MB 的内容，这何止是追加写，这直接咔咔创建新文件。然后通过多路归并的方式不断的进行本地数据的合并，我们可以通过看下图来进一步了解 LevelDB 是如何去实现 LSMT 的：



### 落地

在上图中我们从上往下看，首先是 immutable memtable 持久化落地一个 sstable 到磁盘中，这里是直接落到了 Level\_0 层，这一层有一个特点，每个 sstable 文件虽然文件内 key 有序，但是文件之间并不保证有序，所以 key 会有重叠，如下图 Level\_0 三个文件 key 的范围都有重叠：



这样设计的好处是 immutable memtable 落地时，不需要考虑当前内存里面新写入的记录的关键字是否已经和存在于 Level<sub>0</sub> 的 sstable 有冲突。不过这会造成一些读性能的损失，由于 Level<sub>0</sub> 每个 sstable 可能都会存在某一个 key，在做读取操作时，需要遍历 Level<sub>0</sub> 层的所有文件，如果发现查询的 key 位于这个文件的最小 key 和最大 key 之间，就需要进一步读取文件内容去做判断。再加上快照读的操作，即使找到了相同 key，版本不合适还得继续往下找。

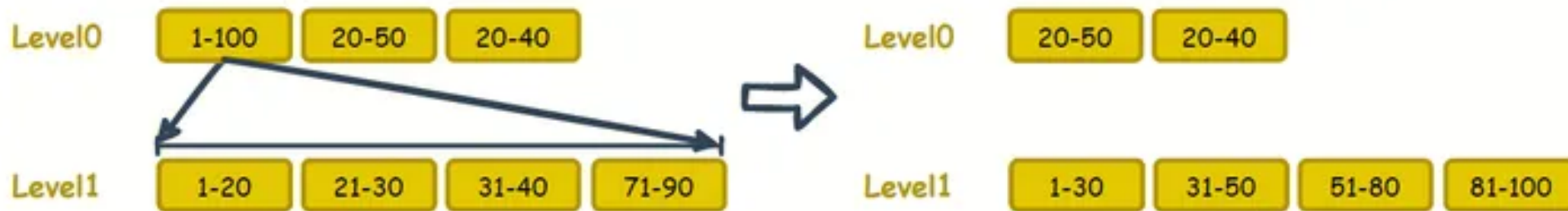
### 合并压缩

从落地逻辑可以看出来，如果 Level<sub>0</sub> 不断的增加文件，那么每次磁盘查询时会遍历所有文件，读性能会不断的降低，最终不堪一用。为了解决这种情况，就需要尽可能减少 Level<sub>0</sub> 的文件个数，LSMT 的方法是新增了一个 level1 层，该层的每个 sstable 文件内 key 单调递增，且每个文件之间也单调递增即 A 文件所有 key 一定小于 B 文件所有 key，这保证了该层的每个文件 key 不会重叠。

当需要查询某个 key 时，先会处理 Level<sub>0</sub> 每个文件，然后对于 level1 层则可以直接通过判断文件 key 的范围来快速定位，由于 Level<sub>1</sub> 层单调递增特性，最终只检索一个文件内容即可。这种情况下，只要将 Level<sub>0</sub> 维持在一定数量内，那么整体的读性能就不会太差，LevelDB 尽量会将 Level<sub>0</sub> 层文件个数维持在 4 个以内。

如何生成 Level<sub>1</sub> 层的文件呢？当 Level<sub>0</sub> 层的文件超过阈值时，就会将 Level<sub>0</sub> 层的文件与 Level<sub>1</sub> 层中有 key 重叠的文件进行合并，将其中无效的 key 给丢弃掉后，生成新的单调递增的文件放到 Level<sub>1</sub> 层中，这种过程被称为**合并压缩**，LevelDB 中规定除去 Level<sub>0</sub> 层外，其他层新生成的文件大小均为 2MB（并不是真的所有文件都是 2MB，只是阈值为 2MB）。

但是仅新增一个 Level<sub>1</sub> 层会无上限的**放大合并压缩时的写操作**，比如下图：



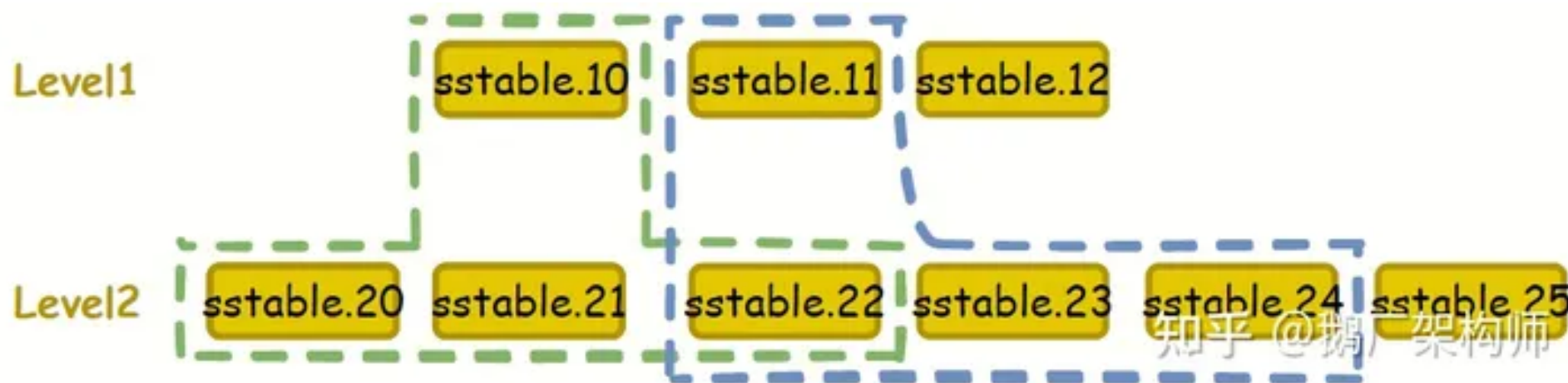
上图中，Level<sub>0</sub> 层在压缩第一个 sstable 文件时，发现 Level<sub>1</sub> 层所有的文件都与它内部的 key 有重叠，为了保证 sstable 文件合并压缩后，Level<sub>1</sub> 层保证单调有序且每个文件之间也有序，必须读取所有 Level<sub>1</sub> 层的文件，然后通过归并排序的方式，重新生成 Level<sub>1</sub> 层的文件。

很明显，一次合并压缩相当于重写整个 Level<sub>1</sub> 层，这个量级就是指数级的，随着存储大小增长，这种情况会越来越糟糕。

LevelDB 会像处理 Level<sub>0</sub> 层一样，处理 Level<sub>1</sub> 层，控制其存储大小阈值为 10MB，通过限制 Level<sub>1</sub> 的总文件大小，从 Level<sub>0</sub> 向 Level<sub>1</sub> 合并压缩时总体的读写消耗是可控的。因为 Level<sub>1</sub> 层撑死也就 10MB 文件，如果在 100M/s 顺序读写能力的磁盘上操作的话，合并 Level<sub>1</sub> 层所有文件，读 10MB 加上写 10MB 也就是花费 20ms 的时间，如果再平摊到每次用户请求花费 1ms 去压缩一下，20 个请求差不多就能很好的分摊这次合并压缩的代价。

当我们限制 Level\_1 层的大小时，就不得不将多余的存储往更高层进行压缩合并，然后我们就需要新增一个 Level\_2 层，同样的 Level\_2 也会面临像 Level\_1 层的问题，不同的是，Level\_0 往 Level\_1 压缩时，由于用户行为不可控，Level\_0 某一个文件与 Level\_1 层所有的文件存在 key 重叠情况就不可控，同理读写消耗也是不可控的，所以才不得不限制 Level\_1 层的文件大小。

但是对于新增的 Level\_2 层却没有这个问题，由于 Level\_1 保证了每个 sstable 文件内有序，同时文件之间有序，所以每次将 Level\_1 层一个文件（文件大小为 2MB）合并压缩到 Level\_2 层时，因为 Level\_2 也同样遵循 Level\_1 的 sstable 特性，那么 Level\_2 层的一个文件有且最多与 Level\_1 层中的两个文件重叠。如下图：



上图中的 sstable.22 左半部分与 sstable.10 重叠，右半部分与 sstable.11 重叠，而这种情况有且仅发生在边界，除了边界外，其他情况 Level\_2 的文件只会与 Level\_1 重叠一个文件，正如图中的 sstable.21 就只会与 sstable.10 重叠，sstable.23 只会与 sstable.11 重叠。

所以 Level\_2 层因为 key 重叠需要处理的文件数平均最坏情况为  $\text{size}(\text{Level}_2) / \text{size}(\text{Level}_1) + 2$  个文件，其中的 2 说的就是 sstable.22 这种正好处于边界的情况。而  $\text{size}(\text{Level}_2) / \text{size}(\text{Level}_1)$  则说的是 sstable.21 这种情况。

Level\_2 的出现，让压缩的量级从指数级，降到了  $\text{size}(\text{Level}_2) / \text{size}(\text{Level}_1)$  的线性级，接着就会遇到另外一个问题，怎么去遏制这种线性级读写消耗的增长呢？

LevelDB 中通过划分更多层来缓解这种线性增长，LevelDB 中除去 Level\_0 外，还会划分出从 Level\_1 到 Level\_6 层，Level\_1 层阈值为 10MB，Level\_2 为 100MB，依此类推，每层的总大小阈值为  $10^n$  MB，也就是说  $\text{size}(\text{Level}_{N+1}) / \text{size}(\text{Level}_N) = 10$ ，当合并压缩 Level\_N 层的一个文件时，Level\_{N+1} 层会取 12 个文件，所以总共 26MB 读 + 26MB 写，在 100M/s 的磁盘中就是 0.5s 左右的消耗，这是每一层最差的合并压缩消耗。

LevelDB 的最高 Level\_6 层存储大概 1TB 左右的内容，超过 1TB 后，由于 Level\_6 不能再往更高层合并压缩数据，合并压缩消耗就会不限控制的线性增长。得益于现在硬件的发展，实际上很多硬件的 IO 能力已经有了很高的提升，比如 SSD 顺序读写可达 3000 MB/s，按照每次 0.5s 的压缩损耗，Level\_{N+1} 可以是 Level\_N 的 375 倍，Level\_6 就能存储 64 EB，也就是说即便到了未来，LevelDB 依旧顶得住。

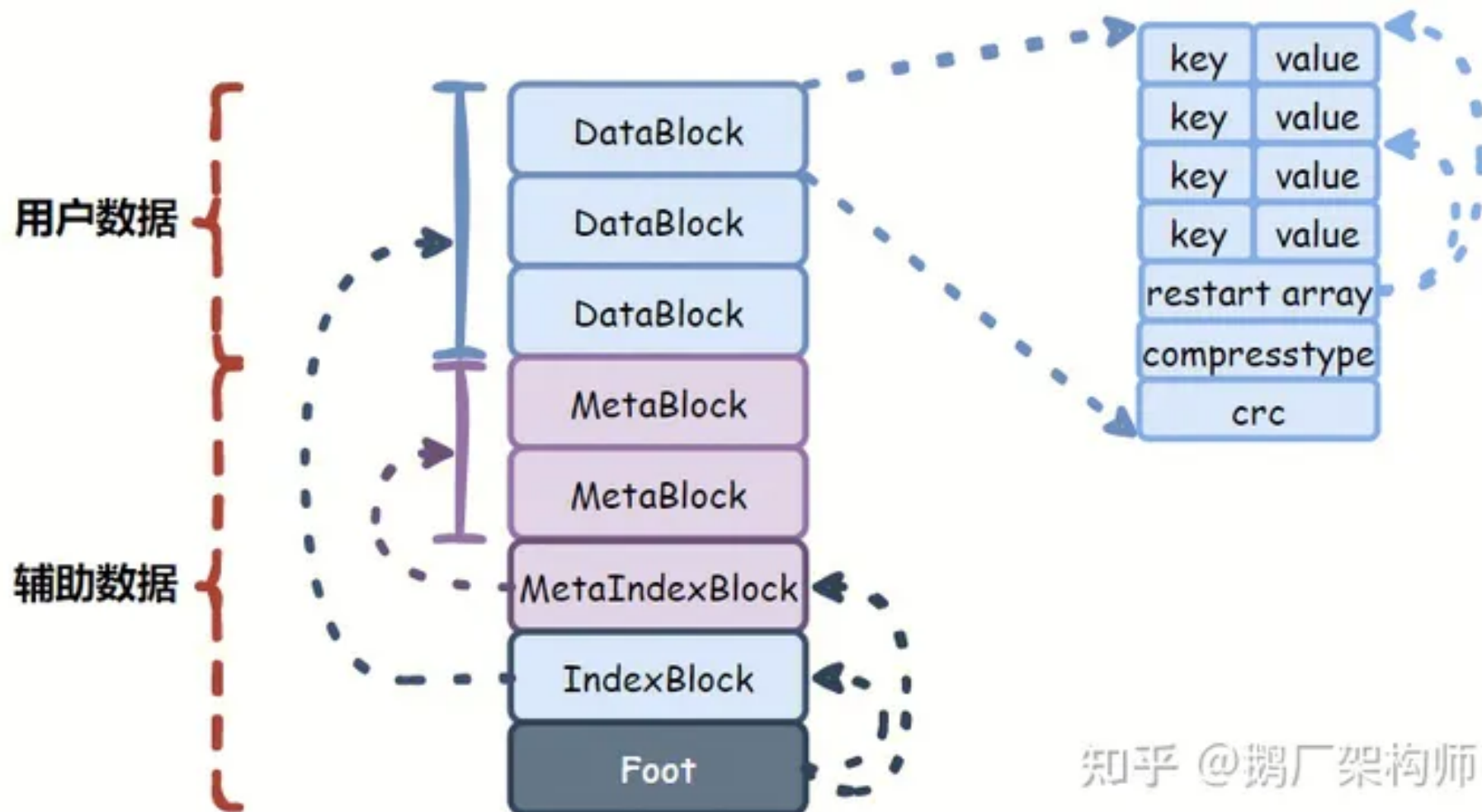
## SSTable

SSTable 全称又叫 Sorted String Table，从字面上就能看出来该文件存储的数据按照字符串有序排列，在 LevelDB 中一个 SSTable 文件里面的内容大致可以分为两类：

1. **用户数据**：真正存储着用户的 key-value pair 信息；
2. **辅助数据**：存储着数据的偏移量或者其他元数据，可以帮助 LevelDB 更好的处理数据。

其文件结构图如下：





知乎 @鹅厂架构师

上图中 SSTable 从上往下，结构依次为：

- **DataBlock**：存储着用户的 key-value pair 数据；
- **MetaBlock**：存储着一些元数据，比如布隆过滤信息，SSTable 的存储记录数等；
- **MetaIndexBlock**：存储着 MetaBlock 的偏移位置，用于快速定位读取 MetaBlock；
- **IndexBlock**：存储着 DataBlock 的偏移位置，用于快速定位读取 DataBlock；
- **Foot**：存储着 MetaIndexBlock、IndexBlock 的偏移位置，该类型块的大小固定为 48 个字节。

我们细说上面几个块的结构。

#### DataBlock & MetaBlock

一个 SSTable 文件有若干个 DataBlock，每个 DataBlock 包含着：

- 1、**key-value**：其中 key-value 中 key 并非每次都存储着全部的字符串，而是存储相比于上一个 key 的除去公共最长前缀外的差异字符串，结构如下：

	4 B	4 B	4 B		
abc	shared_size:0	noshared_size:3	value_size	abc	value
abdef	shared_size:2	noshared_size:3	value_size	def	value
abdex	shared_size:4	noshared_size:1	value_size		

2、**restart array**：重启数组，作用是做二分查找，每隔几条记录就会生成一个 restart point，LevelDB 默认约定隔 16 条记录生成一个新的 restart point，可以通过 `option.block_restart_interval` 来调整，该结构的使用方式：先是二分 restart array，然后在相邻的 restart point 之间顺序遍历；

需要注意的是，由于每一条记录并没有存储 key 的全部字符，而是增量的差异字符串，导致我们获取一个 key 所有字符必须从数据最开始遍历才可以，但是该特性与 restart array 的功能有冲突，所以**每一个作为 restart point 点的 key，必须存储的是一个 key 的全部字符**；

3、**compresstype**：压缩格式；

4、**crc**，校验码，用于校验 Block 是否有异常。

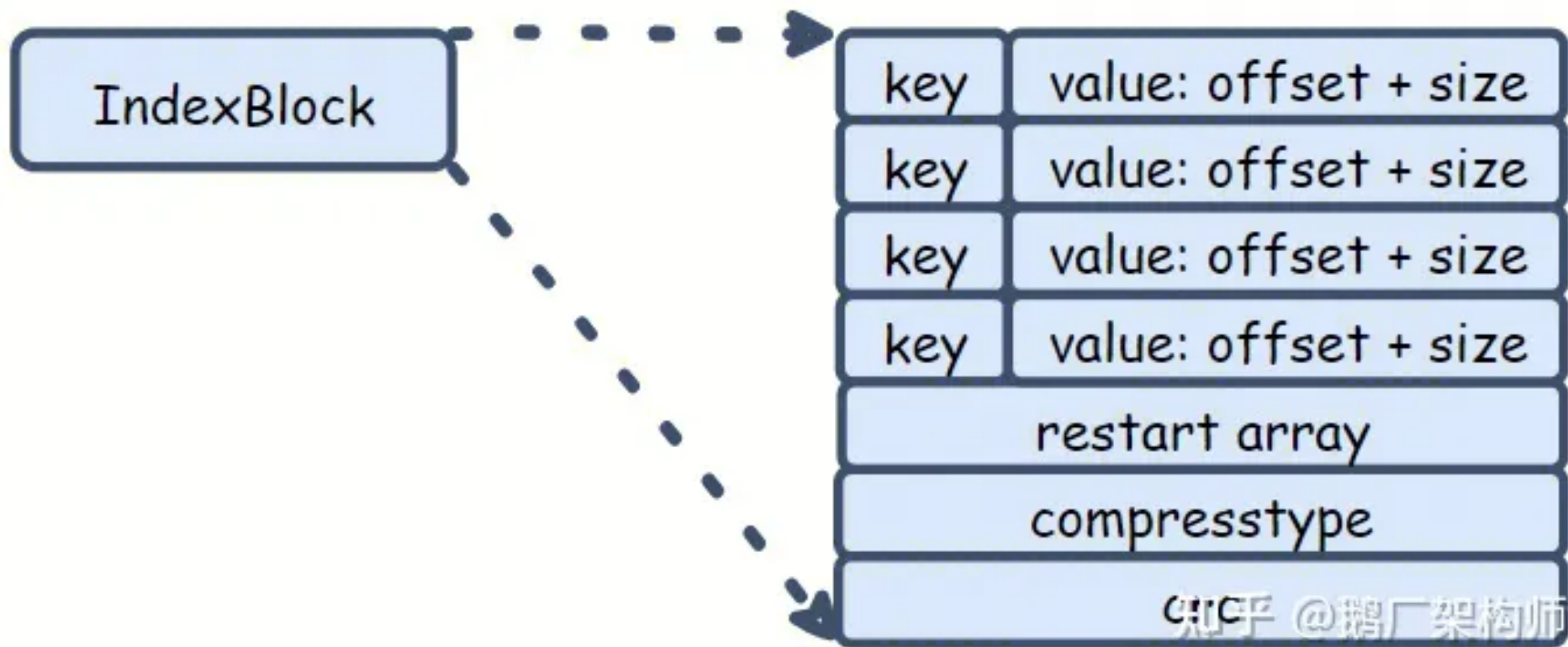
每个 DataBlock 的大小不是固定的，但是有一个最大阈值，默认是 4 KB，当追加一条记录到 DataBlock 后 Block 大小超过了该阈值，后续追加记录就会生成一个新的 DataBlock 来写，开发者可以通过 `option.block_size` 来改变这个默认值，这里设置阈值的目的是有两个：

1. 配合 crc 做数据校验，可以保证即便某一个 DataBlock 损坏了，其他的 Block 也可正常读取，控制住了数据坏点影响的范围；
2. 在低内存设备，比如嵌入式 STM32，车机等设备中运行，每次读取一个 DataBlock 到内存中仅占用 4KB，是一种时间换空间的平衡。

除了 DataBlock 外，还有一种 MetaBlock 存储的是元数据，在 LevelDB 1.5 版本中，MetaBlock 存储着一种叫做 Filter MetaBlock 的过滤元信息，可以防止不必要的 IO 消耗，提高查找效率。Filter MetaBlock 的具体实现可以参考后续的过滤器章节。

### MetaIndexBlock & IndexBlock

MetaIndexBlock 存储着指向 MetaBlock 在 SSTable 的偏移量和其大小，当前由于只有一个 Filter MetaBlock，所以 MetaIndexBlock 仅存储了一个偏移量和大小。同理 IndexBlock 也是如此，只是 IndexBlock 存储的是所有 DataBlock 的偏移量和大小，其结构图示如下：



通过上图可以知道，IndexBlock 本质上还是一个 DataBlock，其实现也是和 DataBlock 共用的一个 BlockBuilder 对象，只是 value 值存储的是 DataBlock 块的偏移数据，查找流程：

1. 读取 IndexBlock 的数据到内存中；
2. 然后通过 restart array 二分查找，找到数据在哪个 DataBlock；
3. 然后通过 offset 和 size 读取相应的 DataBlock；
4. 接着在 DataBlock 进行查找，其步骤也是重复上述过程，直到找到相应的 key。

值得一提的是，IndexBlock 的 restart array 结构，生成 restart point 间隔是 2 个记录（代码中写死了），相邻两个 DataBlock 的起始 key 的公共最长前缀可以理解为一个 DataBlock 内所有记录的公共最长前缀。这种情况下，间隔越大，节省存储的收益越低，公共最长前缀反而成为了阻碍，所以需要尽可能减少 block\_restart\_interval 的值，这里设置为 2 个记录，个人感觉应该是经验使然，具体到业务上，可能还是要基于场景个性化设置才能更好的发挥其作用。

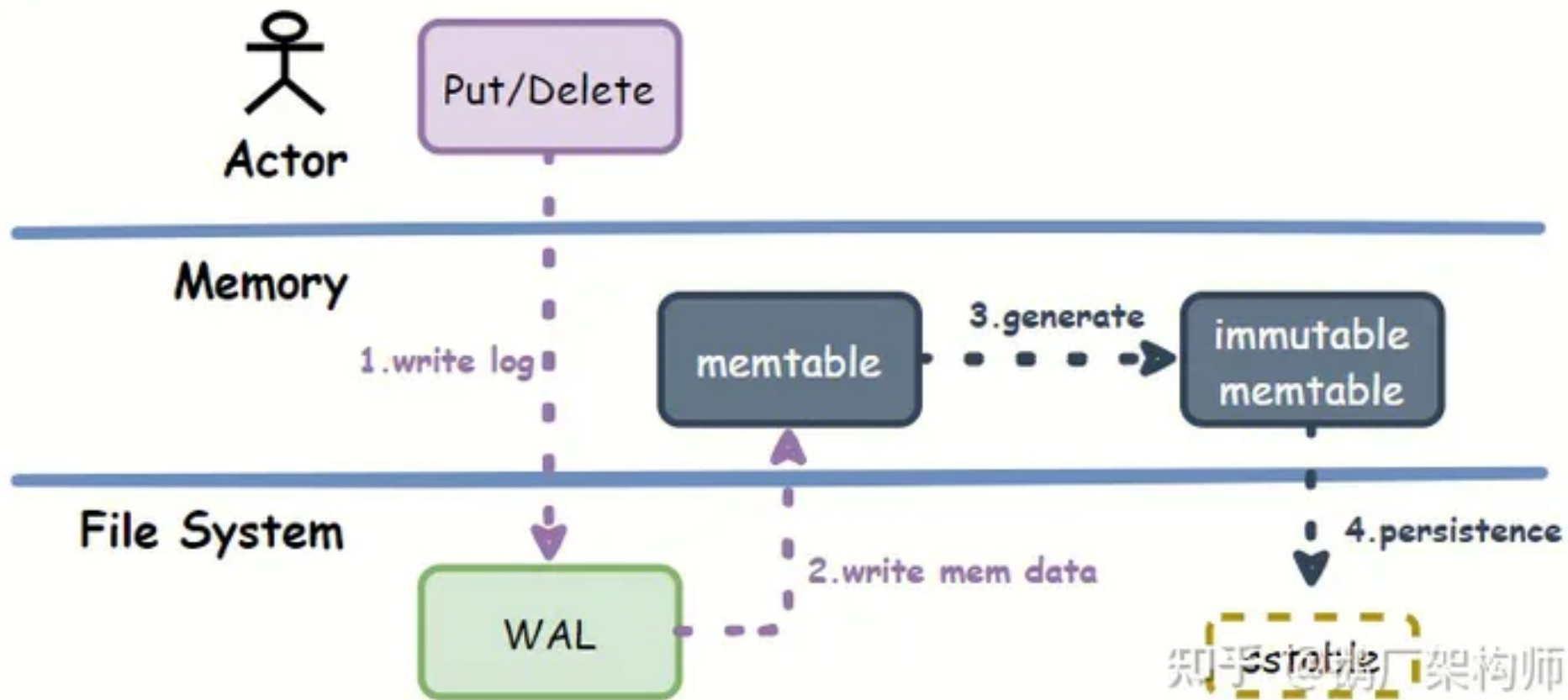
## Foot

Foot 是一个固定大小为 48 字节的结构，其存储了 IndexBlock 和 MetaIndexBlock 的偏移位置。由于 Foot 位于文件末尾，同时大小固定，所以我们打开文件后，直接读取末尾的 48 字节就可以得到 Foot 结构，从而进一步解析 IndexBlock，达到解析整个文件的目的。

## WAL 日志

WAL 是数据库系统中常见的一种手段，用于保证数据操作的原子性和持久性。在使用 WAL 的系统中，所有的修改在提交之前都要先写入 log 文件中。在 LevelDB 中也不例外，看下图：

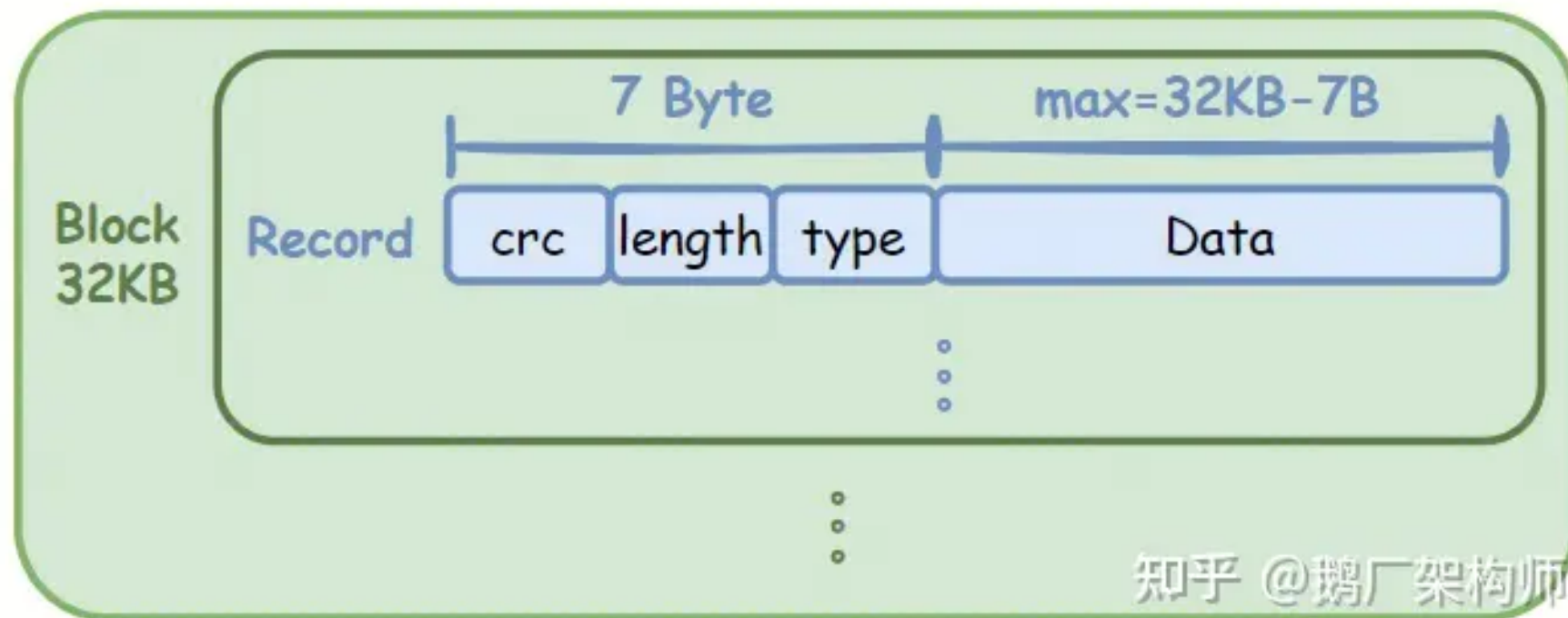




上图中，用户做任何写操作时，都会先将操作写入 WAL 中，落地到磁盘，然后才会将记录写入 memtable 中，这样在进程异常、系统掉电等异常情况发生时，也可以从 WAL 中恢复数据。

每一个 memtable 都对应一个 WAL，直到这个 memtable 落地到 sstable 后，该 WAL 才会被删除。WAL 使用的 log 文件格式如下图：

WAL



一个 WAL 文件包含若干个 32KB 大小的 Block，Block 中包含若干条 Record，如果 Record 导致 Block 超过了 32KB，就会被分为两条记录，如果超过了 64KB，就会被分成三条记录，以此类推，保证 Block 只会是 32KB。WAL 只需要进行顺序读取，采用固定大小的好处显而易见：

1. 不像 SSTable 需要防止读取无用数据，WAL 没有这个顾虑，读进内存的数据都是有用数据，采用 32KB 一次性能读更多的数据，有效的提高读取效率，同时减少磁盘碎片的产生；
2. 一个 Block 一定只有 32KB，在写入 WAL 时不需要在内存中申请一个很大的 Buffer 来存储；
3. 在发生数据异常时，能快速跳过异常 Block，而 SSTable 还要反查 IndexBlock 才知道下一个 Block 的偏移位置和大小。

LevelDB 的线程模型是单线程写，多线程读。写操作只会会有一个线程在处理，所以每次处理时会处理多条 Write 操作，因此在 Record 中的 Data 并非只存储一条 Write 操作，而是某时刻滞留在写队列中的多条 Write 操作合并成一个 Batch 结构，序列化后存到了 Data 中。同时**上述文件格式是一个通用的结构**，在后续的 Manifest 文件中也会用到。

## 压缩策略

通过 LSMT 的合并压缩内容，我们了解到 LevelDB 需要通过合并压缩来提高读性能，那采用什么样的压缩策略可以尽可能减少对用户的影响，同时充分的利用磁盘的 IO 性能呢。LevelDB 中的压缩按照规模可以分为两类：

- **Minor Compaction**：将 immutable memtable 落地到一个 sstable 的操作；
- **Major Compaction**，将每一层的 SSTable 向更高层做合并压缩的操作。

上述的划分只是两种不同的压缩方式，我们更关心什么时机去触发压缩操作，这是一个值得思考的问题，在 LevelDB 中如果按照压缩时机也可以分为两类：

- **被动时机**：
  - 是否存在 immutable memtable，存在则将 immutable table 落地到 SSTable；
  - 记分牌算法根据 level total size 为每一层打分，分数大于 0 且分数越大，该层越先被触发合并触发，其算法具体为

```
if level == 0:
    score = (Level_0 file num) / 4 // 控制 Level 0 层的文件数量
else :
    score = size(Level_N) / 10^N // 此时该层的总大小已经超过了 10^N 的上限，必须做合并压缩

if score > 0:
    compact
```

**主动时机**：如果一个 SStable 文件被反复的访问了 n 次，则该文件会被触发一次压缩，这个 n 的计算规则为

```
n = (file size) / 16KB
```

在被动时机的记分牌算法中，对于 Level\_0 层做了特殊处理，**并非用该层的总大小来打分的，而是文件个数**，这是出于以下情况的考虑：

1. Level\_0 层的文件是直接从 immutable memtable 生成的，所以会受 options\_.write\_buffer\_size 配置的影响，使用总大小来打分并不合适；
2. 由于 Level\_0 文件数据重叠的特殊性，每次 Level\_0 层向 Level\_1 合并压缩时都有可能需要读取 Level\_0 所有的文件，所以非常有必要控制 Level\_0 层的文件数量来减少不必要的磁盘性能消耗。

除此之外，**主动时机中为什么是除以 16KB 呢**，原因如下：

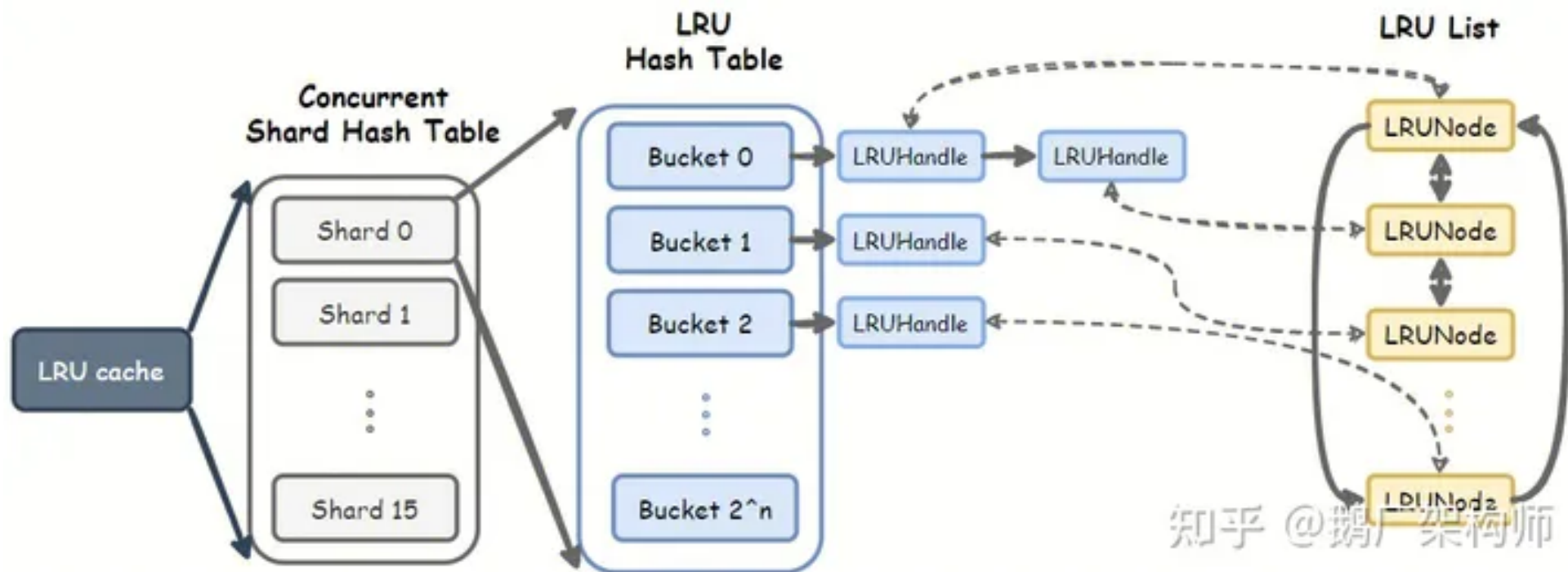
1. 假设当前是在顺序读写 IO 能力 100M/s 的磁盘上，一次查询操作耗时 10ms；
2. 按照合并压缩的规则，一个文件是 1MB，压缩 1MB 文件，需要读写 25 MB 即需要花费 250 ms，相当于 25 次查询操作的消耗；
3. 所以当 一个文件被查询了 25 次时已经相当于将这个文件进行了一次压缩，以此为作为一个文件可以被触发压缩的主动时机，反推一下，一次查询操作耗时就相当于 40KB（1MB / 25）的数据进行合并压缩，理论上应该取 40KB，但为了适应更多低磁盘设备，同时降低主动时机对性能的影响，LevelDB 采用了比较保守的方式，取了 16 KB。

在压缩过程中，LevelDB 还会通过**降低用户体验，来保证数据库的基础体验**：

- 如果 Level\_0 的数量不少于 8 个，则当前写操作会让出 1ms CPU 给到压缩线程去压缩；
- 如果 memtable 已经超过 options.write\_buffer\_size，但是 immutable memtable 还没有落地到 SStable，则当前写操作直接休眠，必须等到 immutable memtable 写入 SStable 后才能接着执行；
- 如果 Level\_0 的文件数量不少于 12 个，则当前写操作直接休眠，必须等到 Level\_0 的数量小于 12 个才往下执行。

## 缓存机制

先看 LevelDB 用于实现缓存类：



整个缓存类分为三个模块：

- **Concurrent Shared Hash Table**：将 key 通过 hash 划分成多个桶，固定为 16 个桶，每个桶都对应着一个 LRU List，每个桶都可以并发读写操作；
- **LRU Hash Table**：使用 hash 表指向 LRU List 节点，同时该表会动态调整大小；
- **LRU List**：根据 LRU 算法维护着双向链表。

该类在我看来，是纯粹给到 LevelDB 中缓存 Table 文件信息使用，虽然 DataBlock 的 Cache 默认使用了该结构，但是从 LevelDB 的开发者暴露 `option.block_cache` 让用户自定义 DataBlock 的 Cache 结构就可以看出来，他更希望使用 LevelDB 的开发人员可以根据自身情况设计 DataBlock 的 Cache。

抛开 LevelDB 使用该缓存类的场景，直接分析其优劣：

#### • 优点：

1. 实现简单，跟大部分 LRU List 实现差不多，甚至直接让 GPT 生成也可以；
2. Concurrent Shared 的存在降低锁粒度，有效的提高了并发读取缓存的能力，这种 hash 分表来提高并发也是一种常见的手段；
3. 动态 LRU Hash Table 能够兼顾 LRU List 的增长，防止性能降低。

#### • 缺点：

1. Concurrent Shared 某些情况下会存在**部分 Shared 高热**，从而导致真正的**热节点被频繁淘汰**；
2. LRU Hash Table 的增长操作是在一个锁里面完成的，这导致只要触发了动态增长大小，读写 LRU List 的操作就**必须得等待**。

在 LevelDB 内部**仅在**缓存 table 辅助信息时使用 LRU Cache，上面提到的缺点反而没有对性能造成影响：

1. 对于第一个缺点：使用 LRU Cache 存储 table 辅助信息时，是以 key 为文件的序号，value 为一个 `Table::Rep` 结构，里面存储着 `index_block`、`meta_index_block` 等数据，由于文件序号是连续增长的，这能保证 key 生成的 hash 值会是均匀的，从而分配的 Shared 均匀，就不会存在部分高热的情况；
2. 对于第二个缺点：由于文件的总数量是有预期的，一个文件是 2MB，6 层文件存储 1T 的数据，也就是 524288 个 2MB 文件，而且一台 linux 服务器同时打开的文件句柄是有限的，所以实际上真正缓存起来的 table 信息会更少，在一个锁里面处理完动态增长的逻辑对 LevelDB 影响有限。

对于文件中的 DataBlock 的缓存，LevelDB 是直接使用 DataBlock 的起始 key 为 缓存 key，value 则是对应的 Block 数据，这里缓存 DataBlock 的结构，如果用户没有主动设置 `option.block_cache`，就会默认使用 LRU Cache，其利弊上述已经讲清。

### 三、文件元数据管理

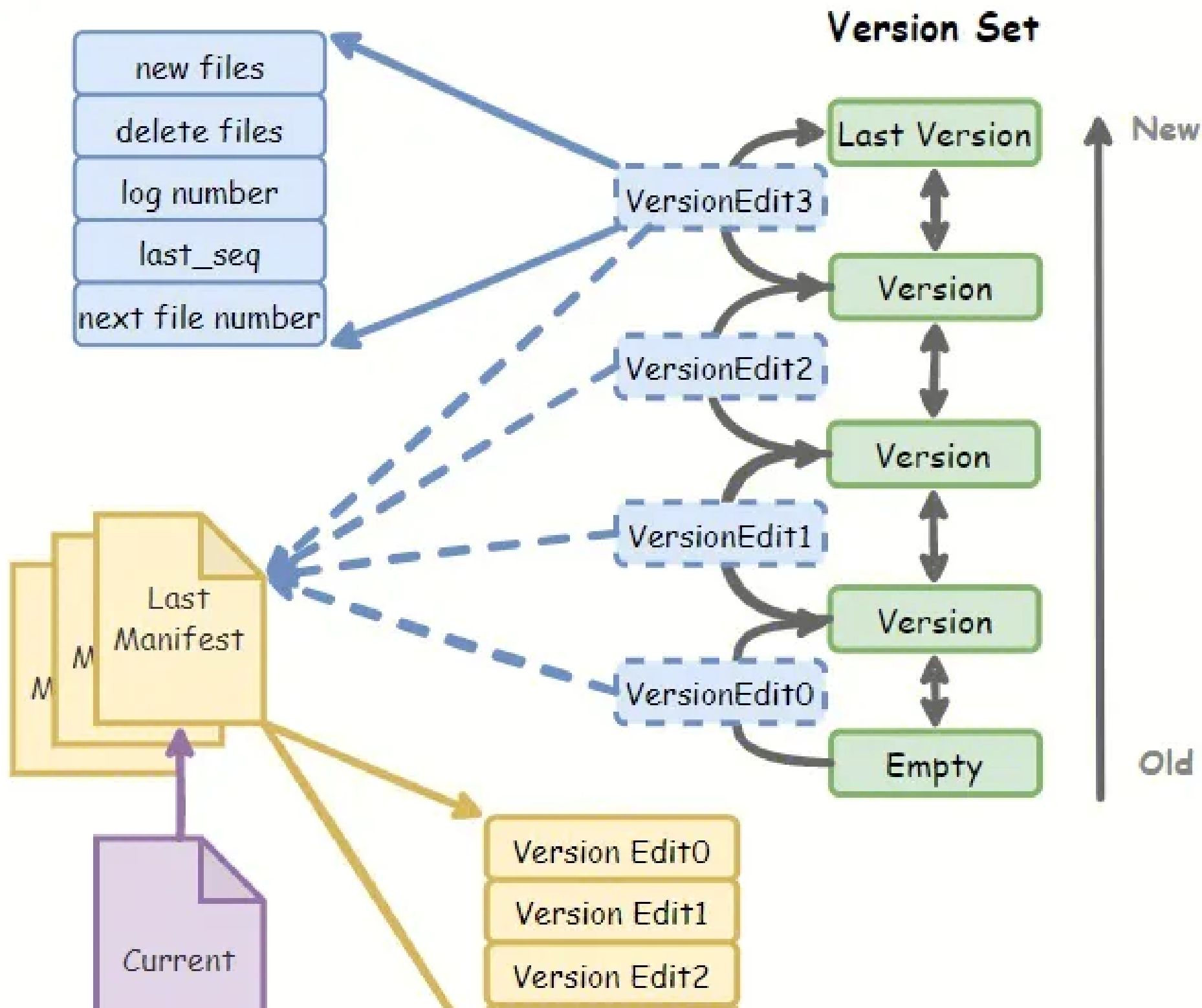
LevelDB 的用户数据由 SSTable 落地存储，又由 WAL 保证在系统断电等异常的情况保证可以恢复数据。同理 SSTable 和 WAL 这些文件的元数据也需要得到保证，否则当系统重启之后无法判断某个 SSTable 属于哪些层级，是否存在 SSTable 文件缺失等信息。

LevelDB 解决上面的问题的方式是通过 Version 和 Manifest 的协同，构建起了文件元数据管理模型。

#### Version & Manifest

首先看 Version 和 Manifest 的协同图：





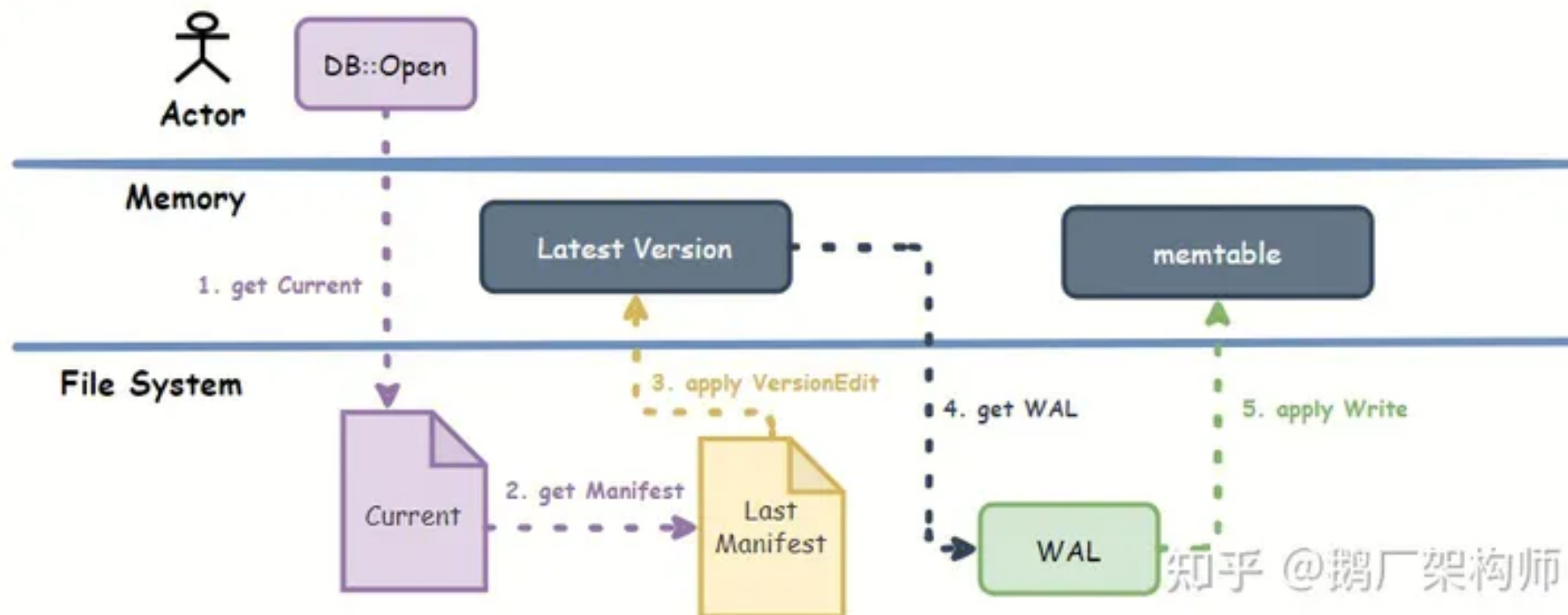
上图中的各个部件：

- **VersionEdit**：存储着增量变化的内容，每次文件变动时会生成一个当前文件变动的 VersionEdit：
  - new files：当前新增了哪些文件，比如 immutable memtable 落地到 sstable，就会导致 Level\_0 层新增一个文件；
  - delete files：当前废弃了哪些文件，比如 Level\_N 向 Level\_{N+1} 合并压缩，Level\_N 就会废弃掉被合并压缩的文件；
  - log number：当前 WAL 的日志文件 ID，用于找到具体的 WAL 文件；
  - last seq：当前最新的操作序号，每次写操作都会导致 seq + 1；
  - next file number：下一个新文件的文件 ID。
- **Version**：记录文件元数据的类，每一次文件变动都会通过上一次的 Version 对象和当前的 VersionEdit 生成新的 Version 对象；
- **Manifest**：每次 DB::Open 打开一个新的 LevelDB 实例时，都会生成一个全新的 Manifest 文件，该文件存储着 VersionEdit 的序列化数据，当系统崩溃或者断电后，可以通过一步一步的回溯，得到数据库异常前的状态；
- **Current**：指向最新的 Manifest。

其中 Manifest 文件存储数据的方式与 WAL 是一样的，WAL 中 Record Data 部分存储的是写操作记录，而 Manifest 存储的则是每次的 VersionEdit。

## Recover

系统崩溃了，如何通过文件元数据和 WAL 来恢复数据，LevelDB Recover 的流程如下：



具体步骤如下：

1. 当系统奔溃后，不需要额外操作，我们直接通过 `DB::Open` 打开当前的 LevelDB 实例；
2. 获取 Current 文件，拿到最新的 Manifest 是哪个文件；
3. 通过加载 Manifest 文件，依次应用所有的 VersionEdit 将 Version 恢复到最新；
4. 接着通过 Version 中的 log number 读取相应的 WAL 文件；
5. 应用 WAL 中的写操作，通过相同的写流程写入到 memtable 中，这个过程可能会触发生成 immutable memtable 以及写 SStable 动作。

在一些极端情况下，恢复操作允许 Manifest 丢失，Manifest 的作用是帮我们维护文件层级、最新操作序号、最新文件号等信息，当 Manifest 丢失后，我们也可以通过将所有的 SStable 当做 0 层，重新进行合并压缩来构建新的层级关系以及最新文件号和操作序号等信息。

## MVCC & 快照

MVCC 的英文全称是 Multiversion Concurrency Control，中文含义是「多版本并发控制技术」。

讲解 LevelDB 中的 MVCC 之前，首先需要深入了解 LevelDB 中 key 的构成，LevelDB 的有序性是通过 key 来维持的，相邻的两个 key 是有序的，非 `Level_0` 层的 SStable 文件之间是有序的，大家直观的感觉，维护这个有序性的 key 就是用户执行 Write 操作时写入的 key，其实不然，LevelDB 中 memtable 以及 SStable 中存储的 key 的结构如下：

```
|      InternalKey      |
+-----+-----+-----+
| user key | seq | type |
+-----+-----+-----+
```

LevelDB 将这个 key 叫做 **InternalKey**，其分为三部分：

- user key：用户传进来的 key；
- seq：当前写操作的序号；
- type：当前是一次删除操作，还是普通的写操作。

InternalKey 的有序性可以通过 `option.comparator` 来控制的，如果用户不传递自定义的排序函数，默认情况下采取如下排序策略：

1. 先按照 user key 升序；
2. 然后按照 seq 降序；
3. 最后按照 type 降序。

LevelDB 每次写数据都是增量记录，然后落地到存储中，相同的 user key 不会覆写，也就是说会同时存在多个相同 user key 的记录，所以 SStable 中的数据排列是这样的：

user key	seq	type
aba	2	1
abb	6	1
abb	4	0
abc	7	1
abc	5	0
abc	1	1

从上图中可以看出，LevelDB 存储结构就保留着同一个 user key 的多个数据版本，天然实现了 MVCC，所以问题就只剩下如何解决我们的可见性问题。

在 MySQL 中有一个 ReadView 结构来构建用户的可见数据范围，在 LevelDB 中则是一个叫做 **Snapshot** 的快照结构体，该结构很简单，仅有一个 number 字段，存储着当前的操作序号。

当我们进行查询时，LevelDB 会生成一个 **LookUpKey**，该 Key 由 user key、snapshot.seq 和 type 组成，然后去查找，查找的方案是找到一个小于等于当前 LookUpKey 的第一条记录。

比如我们拿到 snapshot.seq = 4，然后我们去查找 abc，生成的 LookUpKey 为 user key=abc, seq=4, type=1，根据 comparator 判断，我们就会查找到 user key=abc, seq=1, type=1 这条记录。

对于没有传 Snapshot 的查找操作，会直接取当前最大的 seq 当做 LookUpKey 中的 seq，然后进行查找，这样我们就能查找到最新的 user key 更新记录。

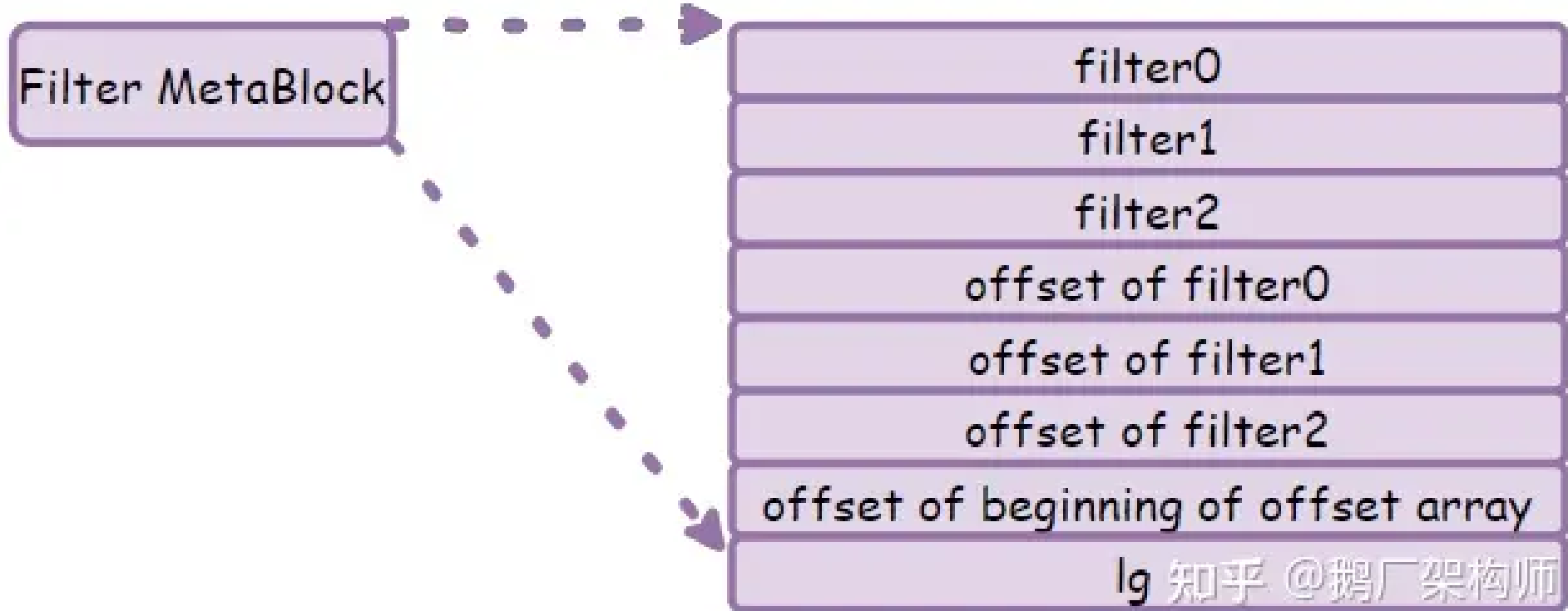
LevelDB 存储多个数据版本会造成大量的数据冗余，需要适当进行丢弃，丢弃 user key 低于最新版本的其他数据版本的前提是：该数据版本不再被引用。判断时机是在合并压缩(包含 minor和 major 两个时机)的时候。

## 四、奇思妙招

### 过滤器

为了提高查询效率，LevelDB 内置了一个 Filter MetaBlock 的数据结构，跟 DataBlock 存储到一起，这是一个可选项，默认情况下 LevelDB 并不会写该结构，如果你设置了 options.filter\_policy 选项，才会起作用。

LevelDB 在存储 Filter MetaBlock 时结构如下：



上图中的各个部件：

- **filter 块**：按照 `lg`（该值为2KB）划分 DataBlock 中的数据来生成 filter，比如 DataBlock 为 4KB，所以此时会生成 2 个 filter 块，第一个 filter 块对应 0-2KB，第二个 filter 块对应 2KB-4KB；
- **offset of filter**：存储着 filter 块的偏移量；
- **offset of offset array**：存储着第一个 offset of filter 的偏移量；
- **lg**：将 DataBlock 按照 lg 大小划分多个 filter。

LevelDB 默认提供了布隆过滤器，用户可以直接将其填写到 `options.filter_policy` 来使用它。filter 块存储的内容就是布隆过滤器的结果。通过布隆过滤器能够做到不在 filter 块中的数据就一定不存在于 DataBlock，以此减少查询次数，提高查询效率。

接下来需要打破刚刚建立的认知，假设一个 DataBlock 为 4KB，filter 块会生成 2 个，真实情况并不是第一个 filter 块指向 0-2KB，第二个 filter 块指向 2KB-4KB。而是**第二个 filter 块指向 0-4KB**。第一个 filter 块不存储任何过滤信息，它的存在只是为了方便 DataBlock 使用 offset array 下标进行快捷查找：

1. 通过 `DataBlock offset / 2KB` 得到下标；
2. 然后直接通过 `offset array[下标]` 就可以直接拿到当前 DataBlock 对应的过滤信息。

## 五、总结

LevelDB 功能强悍的同时在代码实现上也很简洁且出众，各模块的功能单一集中，代码注释方面更是可以称之为典范，其整体架构能够很清晰的表达出它的设计理念。麻雀虽小五脏俱全，LevelDB 将实现一个 KV 库的基础技能都用上了。非常建议可以去看看 LevelDB 的源码。



## 资料

1. [leveldb github](#)
2. [leveldb-handbook](#)