

# LevelDB 日知录系列

## LevelDB 日知录之一：LevelDB

说起 LevelDB 也许您不清楚，但是如果作为 IT 工程师，不知道下面两位大神级别的工程师，那您的领导估计会 Hold 不住了：Jeff Dean 和 Sanjay Ghemawat。这两位是 Google 公司重量级的工程师，为数甚少的 Google Fellow 之二。

Jeff Dean 其人：[research.google.com/people/jeff/](https://research.google.com/people/jeff/)，Google 大规模分布式平台 Bigtable 和 MapReduce 主要设计和实现者。

Sanjay Ghemawat 其人：[research.google.com/people/sanjay/](https://research.google.com/people/sanjay/)，Google 大规模分布式平台 GFS，Bigtable 和 MapReduce 主要设计和实现工程师。

LevelDB 就是这两位大神级别的工程师发起的开源项目，简而言之，LevelDB 是能够处理十亿级别规模 Key-Value 型数据持久性存储的 C++ 程序库。正像上面介绍的，这二位是 Bigtable 的设计和实现者，如果了解 Bigtable 的话，应该知道在这个影响深远的分布式存储系统中有两个核心的部分：Master Server 和 Tablet Server。其中 Master Server 做一些管理数据的存储以及分布式调度工作，实际

的分布式数据存储以及读写操作是由 Tablet Server 完成的，而 LevelDB 则可以理解为一个简化版的 Tablet Server。

LevelDB 有如下一些特点：

首先，LevelDB 是一个持久化存储的 KV 系统，和 Redis 这种内存型的 KV 系统不同，LevelDB 不会像 Redis 一样狂吃内存，而是将大部分数据存储到磁盘上。

其次，LevelDB 在存储数据时，是根据记录的 key 值有序存储的，就是说相邻的 key 值在存储文件中是依次顺序存储的，而应用可以自定义 key 大小比较函数，LevelDB 会按照用户定义的比较函数依序存储这些记录。

再次，像大多数 KV 系统一样，LevelDB 的操作接口很简单，基本操作包括写记录，读记录以及删除记录。也支持针对多条操作的原子批量操作。

另外，LevelDB 支持数据快照 (snapshot) 功能，使得读取操作不受写操作影响，可以在读操作过程中始终看到一致的数据。

除此外，LevelDB 还支持数据压缩等操作，这对于减小存储空间以及增快 IO 效率都有直接的帮助。

LevelDB 性能非常突出，官方网站报道其随机写性能达到 40 万条记录每秒，而随机读性能达到 6 万条记录每秒。**总体来说，LevelDB 的写操作要大大快于读操作，而顺序读写操作则大大快于随机读写操作。**至于为何是这样，看了我们后续推出的 LevelDB 日知录，估计您会了解其内在原因。

## LevelDB 日知录之二：整体架构

LevelDB 本质上是一套存储系统以及在这套存储系统上提供的一些操作接口。为了便于理解整个系统及其处理流程，我们可以从两个不同的角度来看待 LevelDB：静态角度和动态角度。从静态角度，可以假想整个系统正在运行过程中（不断插入删除读取数据），此时我们给 LevelDB 照相，从照片可以看到之前系统的数据在内存和磁盘中是如何分布的，处于什么状态等；从动态的角度，主要是了解系统是如何写入一条记录，读出一条记录，删除一条记录的，同时也包括除了这些接口操作外的内部操作比如 compaction，系统运行时崩溃后如何恢复系统等方面。

本节所讲的整体架构主要从静态角度来描述，之后接下来的几节内容会详述静态结构涉及到的文件或者内存数据结构，LevelDB 日知录后半部分主要介绍动态视角下的 LevelDB，就是说整个系统是怎么运转起来的。

LevelDB 作为存储系统，数据记录的存储介质包括内存以及磁盘文件，如果像上面说的，当 LevelDB 运行了一段时间，此时我们给 LevelDB 进行透视拍照，那么您会看到如下一番景象：

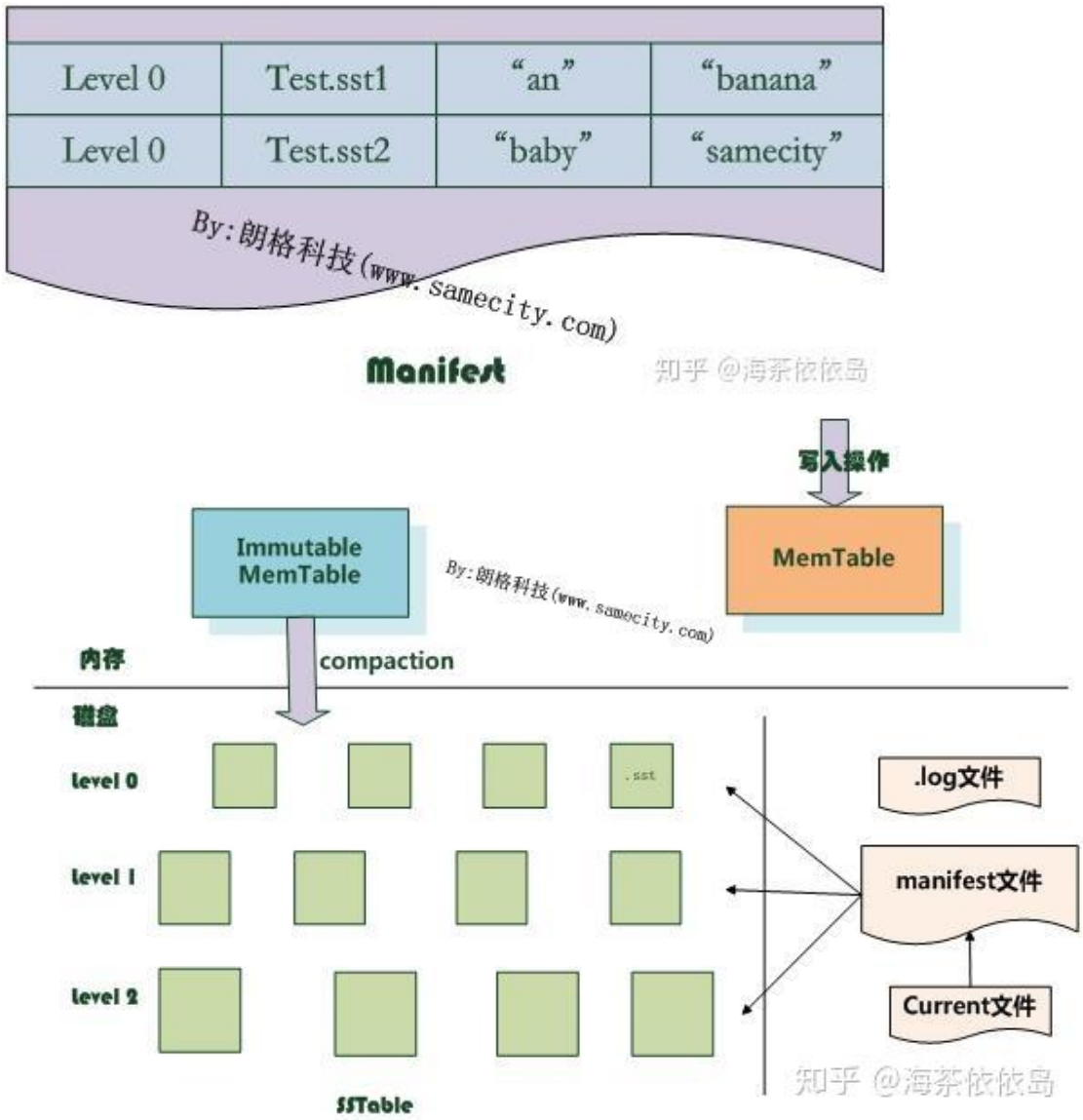


图 1.1: LevelDB 结构

从图中可以看出，构成 LevelDB 静态结构的包括六个主要部分：内存中的 MemTable 和 Immutable MemTable 以及磁盘上的几种主要文件：Current 文件，Manifest 文件，Log 文件以及 SSTable 文件。当然，LevelDB 除了这六个主要部分还有一些辅助的文件，但是以上六个文件和数据结构是 LevelDB 的主体构成元素。

LevelDB 的 Log 文件和 Memtable 与 Bigtable 论文中介绍的是一致的，当应用写入一条 Key:Value 记录的时候，LevelDB 会先往 Log 文件里写入，成功后将记录插进 Memtable 中，这样基本就算完成了写入操作，因为一次写入操作只涉及一次磁盘顺序写和一次内存写入，所以这是为何说 LevelDB 写入速度极快的主要原因。

Log 文件在系统中的作用主要是用于系统崩溃恢复而不丢失数据，假如没有 Log 文件，因为写入的记录刚开始是保存在内存中的，此时如果系统崩溃，内存中的数据还没有来得及 Dump 到磁盘，所以会丢失数据（Redis 就存在这个问题）。为了避免这种情况，LevelDB 在写入内存前先将操作记录到 Log 文件中，然后再记入内存中，这样即使系统崩溃，也可以从 Log 文件中恢复内存中的 Memtable，不会造成数据的丢失。

当 Memtable 插入的数据占用内存到了一个界限后，需要将内存的记录导出到外存文件中，LevelDB 会生成新的 Log 文件和

Memtable, 原先的 Memtable 就成为 Immutable Memtable , 顾名思义, 就是说这个 Memtable 的内容是不可更改的, 只能读不能写入或者删除。新到来的数据被记入新的 Log 文件和 Memtable, LevelDB 后台调度会将 Immutable Memtable 的数据导出到磁盘, 形成一个新的 SSTable 文件。SSTable 就是由内存中的数据不断导出并进行 Compaction 操作后形成的, 而且 SSTable 的所有文件是一种层级结构, 第一层为 Level 0, 第二层为 Level 1, 依次类推, 层级逐渐增高, 这也是为何称之为 LevelDB 的原因。

SSTable 中的文件是 key 有序的, 就是说在文件中小 key 记录排在大 key 记录之前, 各个 Level 的 SSTable 都是如此, 但是这里需要注意的一点是: Level 0 的 SSTable 文件 (后缀为 .sst) 和其它 Level 的文件相比有特殊性: 这个层级内的 .sst 文件, 两个文件可能存在 key 重叠, 比如有两个 Level 0 的 sst 文件, 文件 A 和文件 B, 文件 A 的 key 范围是: {bar, car}, 文件 B 的 key 范围是: {blue, samecity}, 那么很可能两个文件都存在 key= "blood" 的记录。对于其它 Level 的 SSTable 文件来说, 则不会出现同一层级内 .sst 文件的 key 重叠现象, 就是说 Level L 中任意两个 .sst 文件, 那么可以保证它们的 key 值是不会重叠的。这点需要特别注意, 后面您会看到很多操作的差异都是由于这个原因造成的。

SSTable 中的某个文件属于特定层级, 而且其存储的记录是 key 有

序的，那么必然有文件中的最小 key 和最大 key，这是非常重要的信息，LevelDB 应该记下这些信息。Manifest 就是干这个的，它记载了 SSTable 各个文件的管理信息，比如属于哪个 Level，文件名称叫啥，最小 key 和最大 key 各自是多少。下图是 Manifest 所存储内容的示意：

Level 0	Test.sst1	"an"	"banana"
Level 0	Test.sst2	"baby"	"samecity"

By: 朗格科技 (www.samecity.com)

**Manifest**

知乎 @海系依依岛

图 2.1: Manifest 存储示意图

图中只显示了两个文件（manifest 会记载所有 SSTable 文件的这些信息），即 Level 0 的 Test1.sst 和 Test2.sst 文件，同时记载了这些文件各自对应的 key 范围，比如 Test1.sst 的 key 范围是“an”到“banana”，而文件 Test2.sst 的 key 范围是“baby”到“samecity”，可以看出两者的 key 范围是有重叠的。

Current 文件是干什么的呢？这个文件的内容只有一个信息，就是记载当前的 manifest 文件名。因为在 LevelDB 的运行过程中，随着

Compaction 的进行, SSTable 文件会发生变化, 会有新的文件产生, 老的文件被废弃, Manifest 也会跟着反映这种变化, 此时往往会新生成 Manifest 文件来记载这种变化, 而 Current 则用来指出哪个 Manifest 文件才是我们关心的那个 Manifest 文件。

以上介绍的内容就构成了 LevelDB 的整体静态结构, 在 LevelDB 日知录接下来的内容中, 我们会首先介绍重要文件或者内存数据的具体数据布局与结构。

### **LevelDB 日知录之三: log 文件**

上节内容讲到 log 文件在 LevelDB 中的主要作用是系统故障恢复时, 能够保证不会丢失数据。因为在将记录写入内存的 Memtable 之前, 会先写入 log 文件, 这样即使系统发生故障, Memtable 中的数据没有来得及 Dump 到磁盘的 SSTable 文件, LevelDB 也可以根据 log 文件恢复内存的 Memtable 数据结构内容, 不会造成系统丢失数据, 在这点上 LevelDB 和 Bigtable 是一致的。

下面我们带大家看看 log 文件的具体物理和逻辑布局是怎样的, LevelDB 对于一个 log 文件, 会把它切割成以 32K 为单位的物理 Block, 每次读取的单位以一个 Block 作为基本读取单位, 下图展示的 log 文件由 3 个 Block 构成, 所以从物理布局来讲, 一个 log 文件就是由连续的 32K 大小 Block 构成的。



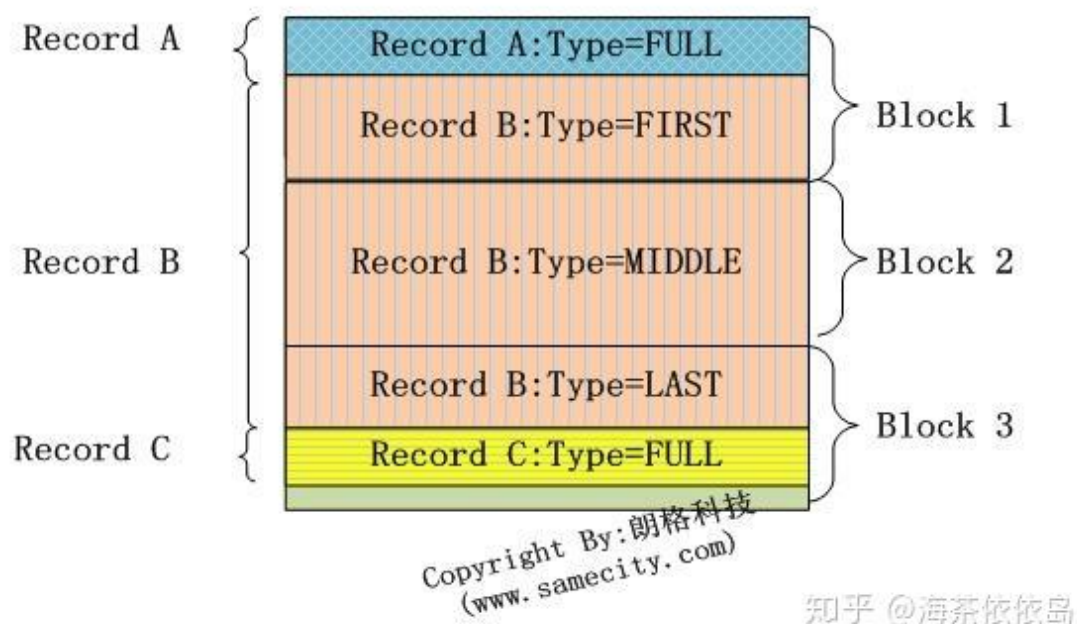


图 3.1 log 文件布局

在应用的视野里是看不到这些 Block 的，应用看到的是一系列的 Key:Value 对，在 LevelDB 内部，会将一个 Key:Value 对看做一条记录的数据，另外在这个数据前增加一个记录头，用来记载一些管理信息，以方便内部处理，图 3.2 显示了一个记录在 LevelDB 内部是如何表示的。



图 3.2 记录结构

记录头包含三个字段，ChechSum 是对“类型”和“数据”字段的校验码，为了避免处理不完整或者是被破坏的数据，当 LevelDB 读取记录数据时候会对数据进行校验，如果发现和存储的 CheckSum 相同，说明数据完整无破坏，可以继续后续流程。“记录长度”记载了数据的大小，“数据”则是上面讲的 Key:Value 数值对，“类型”字段则指出了每条记录的逻辑结构和 log 文件物理分块结构之间的关系，具体而言，主要有以下四种类型：FULL/FIRST/MIDDLE/LAST。

如果记录类型是 FULL，代表了当前记录内容完整地存储在一个物理 Block 里，没有被不同的物理 Block 切割开；如果记录被相邻的物理 Block 切割开，则类型会是其他三种类型中的一种。我们以图 3.1 所示的例子来具体说明。

假设目前存在三条记录，Record A，Record B 和 Record C，其中 Record A 大小为 10K，Record B 大小为 80K，Record C 大小为 12K，那么其在 log 文件中的逻辑布局会如图 3.1 所示。Record A 是图中蓝色区域所示，因为大小为  $10K < 32K$ ，能够放在一个物理 Block 中，所以其类型为 FULL；Record B 大小为 80K，而 Block 1 因为放入了 Record A，所以还剩下 22K，不足以放下 Record B，所以在 Block 1 的剩余部分放入 Record B 的开头一部分，类型标识为 FIRST，代表了是一个记录的起始部分；Record B 还有 58K 没有存储，这些只能依次放在后续的物理 Block 里面，因为 Block 2

大小只有 32K，仍然放不下 Record B 的剩余部分，所以 Block 2 全部用来放 Record B，且标识类型为 MIDDLE，意思是这是 Record B 中间一段数据；Record B 剩下的部分可以完全放在 Block 3 中，类型标识为 LAST，代表了这是 Record B 的末尾数据；图中黄色的 Record C 因为大小为 12K，Block 3 剩下的空间足以全部放下它，所以其类型标识为 FULL。

从这个小例子可以看出逻辑记录和物理 Block 之间的关系，LevelDB 一次物理读取为一个 Block，然后根据类型情况拼接出逻辑记录，供后续流程处理。

## **LevelDB 日知录之四：SSTable 文件**

SSTable 是 Bigtable 中至关重要的一块，对于 LevelDB 来说也是如此，对 LevelDB 的 SSTable 实现细节的了解也有助于了解 Bigtable 中一些实现细节。

本节内容主要讲述 SSTable 的静态布局结构，我们曾在“LevelDB 日知录之二：整体架构”中说过，SSTable 文件形成了不同 Level 的层级结构，至于这个层级结构是如何形成的我们放在后面 Compaction 一节细说。本节主要介绍 SSTable 某个文件的物理布局 and 逻辑布局结构，这对了解 LevelDB 的运行过程很有帮助。

LevelDB 不同层级有很多 SSTable 文件（以后缀 .sst 为特征），所有 .sst 文件内部布局都是一样的。上节介绍 log 文件是物理分块的，SSTable 也一样会将文件划分为固定大小的物理存储块，但是两者逻辑布局大不相同，根本原因是：log 文件中的记录是 key 无序的，即先后记录的 key 大小没有明确大小关系，而 .sst 文件内部则是根据记录的 key 由小到大排列的，从下面介绍的 SSTable 布局可以体会到 key 有序是为何如此设计 .sst 文件结构的关键。

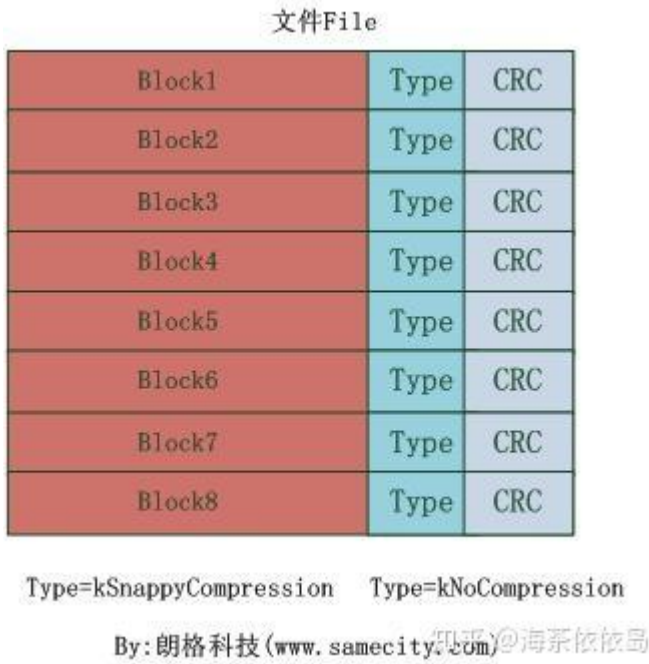


图 4.1 .sst 文件的分块结构

图 4.1 展示了一个 .sst 文件的物理划分结构，同 log 文件一样，也是划分为固定大小的存储块，每个 Block 分为三个部分，红色部分是数据存储区，蓝色的 Type 区用于标识数据存储区是否采用了数据

压缩算法（Snappy 压缩或者无压缩两种），CRC 部分则是数据校验码，用于判别数据是否在生成和传输中出错。

以上是 .sst 的物理布局，下面介绍 .sst 文件的逻辑布局，所谓逻辑布局，就是说尽管大家都是物理块，但是每一块存储什么内容，内部又有什么结构等。图 4.2 展示了 .sst 文件的内部逻辑解释。

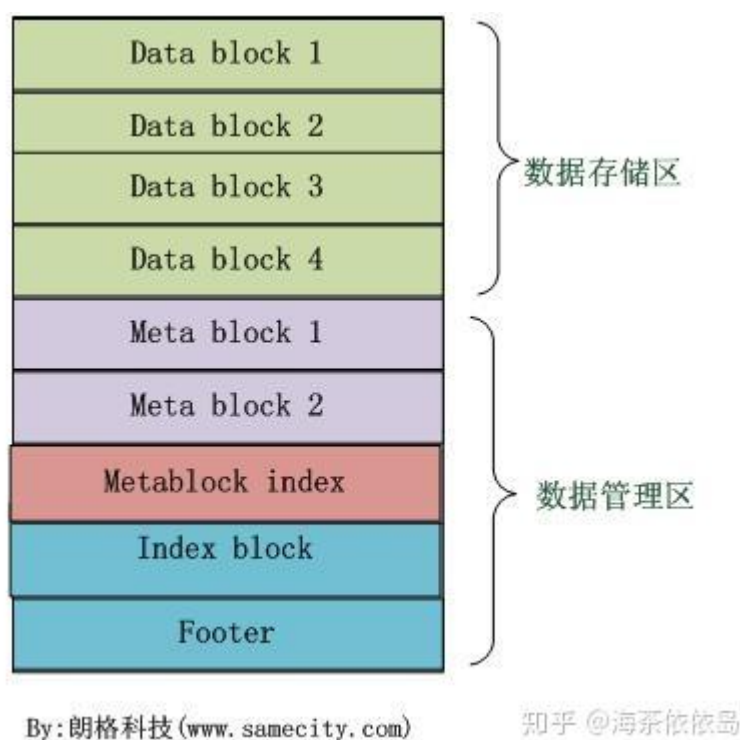


图 4.2 逻辑布局

从图 4.2 可以看出，从大的方面，可以将 .sst 文件划分为数据存储区和数据管理区，数据存储区存放实际的 Key:Value 数据，数据管理区则提供一些索引指针等管理数据，目的是更快速便捷的查找相应的记录。两个区域都是在上述的分块基础上的，就是说文件的前面若干

块实际存储 KV 数据，后面数据管理区存储管理数据。管理数据又分为四种不同类型：紫色的 Meta Block，红色的 MetaBlock 索引和蓝色的数据索引块以及一个文件尾部块。

LevelDB 1.2 版对于 Meta Block 尚无实际使用，只是保留了一个接口，估计会在后续版本中加入内容，下面我们看看数据索引区和文件尾部 Footer 的内部结构。

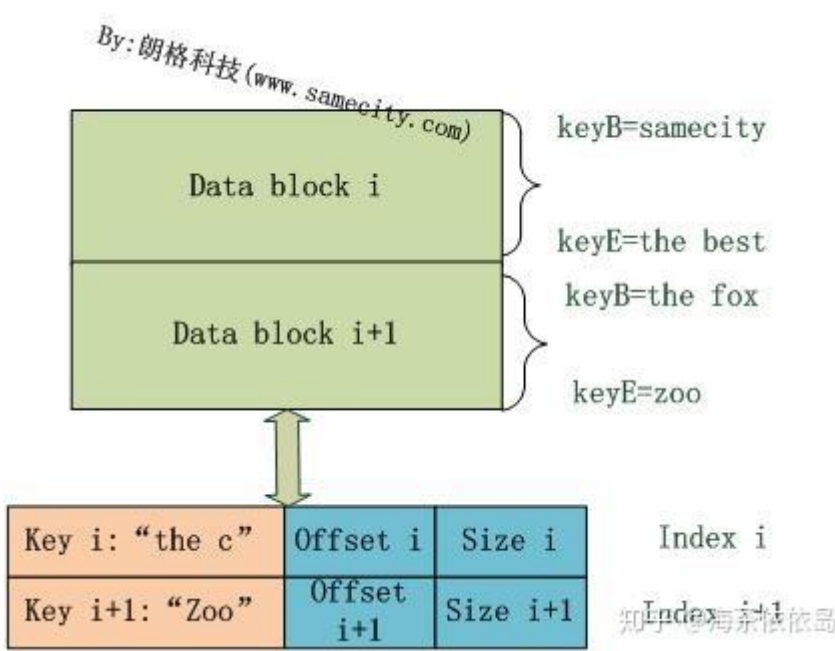


图 4.3 数据索引

图 4.3 是数据索引的内部结构示意图。再次强调一下，Data Block 内的 KV 记录是按照 key 由小到大排列的，数据索引区的每条记录是对某个 Data Block 建立的索引信息，每条索引信息包含三个内容，以图 4.3 所示的数据块 i 的索引 Index i 来说：红色部分的第一个字段记载大于等于数据块 i 中最大的 key 值的那个 key，第二

个字段指出数据块  $i$  在 .sst 文件中的起始位置，第三个字段指出 Data Block  $i$  的大小（有时候是有数据压缩的），后面两个字段好理解，是用于定位数据块在文件中的位置的，第一个字段需要详细解释一下，在索引里保存的这个 key 值未必一定是某条记录的 key，以图 4.3 的例子来说，假设数据块  $i$  的最小 key= "samecity"，最大 key= "the best"；数据块  $i+1$  的最小 key= "the fox"，最大 key= "zoo"，那么对于数据块  $i$  的索引 Index  $i$  来说，其第一个字段记载大于等于数据块  $i$  的最大 key( "the best" ) 同时要小于数据块  $i+1$  的最小 key( "the fox" )，所以例子中 Index  $i$  的第一个字段是： "the c"，这个是满足要求的；而 Index  $i+1$  的第一个字段则是 "zoo"，即数据块  $i+1$  的最大 key。

文件末尾 Footer 块的内部结构见图 4.4，Metaindex\_handle 指出了 metaindex block 的起始位置和大小；index\_handle 指出了 index Block 的起始地址和大小；这两个字段可以理解为索引的索引，是为了正确读出索引值而设立的，后面跟着一个填充区和魔数。

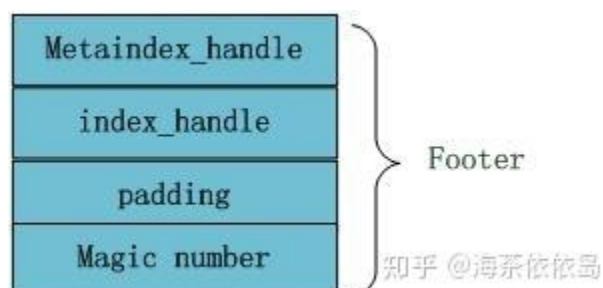


图 4-4 Footer

上面主要介绍的是数据管理区的内部结构，下面我们看看数据区的一个 Block 的数据部分内部是如何布局的（图 4.1 中的红色部分），图 4.5 是其内部布局示意图。

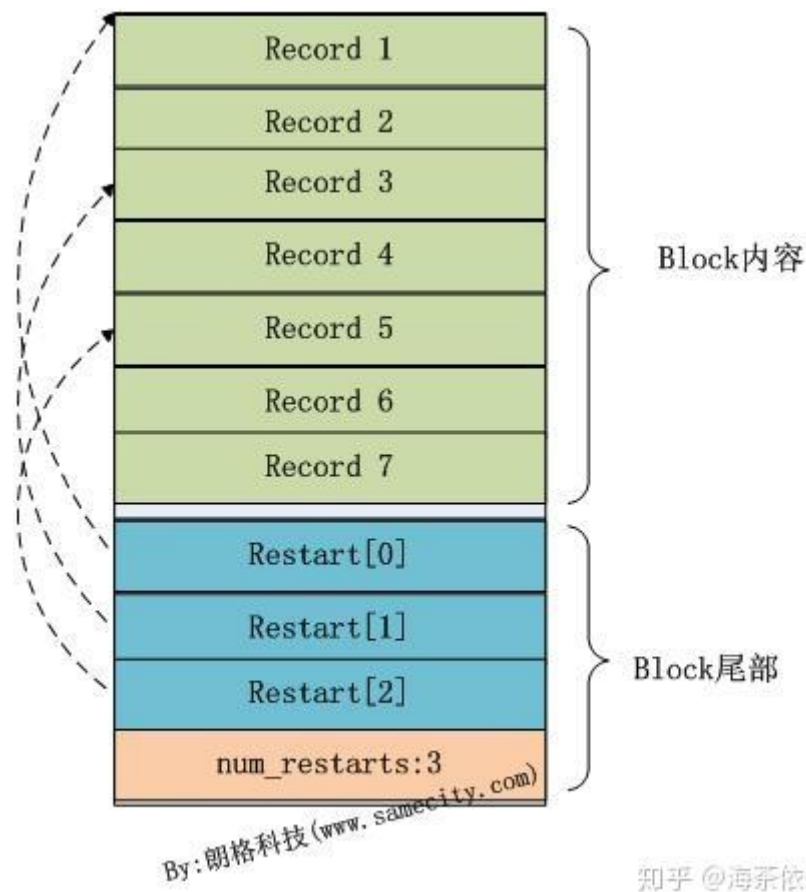


图 4.5 数据 Block 内部结构

从图中可以看出，其内部也分为两个部分，前面是一个个 KV 记录，其顺序是根据 key 值由小到大排列的，在 Block 尾部则是一些“重启点”（Restart Point），其实是一些指针，指出 Block 内容中的一些记录位置。



“重启点”是干什么的呢？我们一再强调，Block 内容里的 KV 记录是按照 key 大小有序的，这样的话，相邻的两条记录很可能 key 部分存在重叠，比如 key i= “the car” ， key i+1= “the color” ，那么两者存在重叠部分 “the c” ，为了减少 key 的存储量，key i+1 可以只存储和上一条 key 不同的部分 “olor” ，两者的共同部分从 key i 中可以获得。记录的 key 在 Block 内容部分就是这么存储的，主要目的是减少存储开销。“重启点”的意思是：在这条记录开始，不再采取只记载不同的 key 部分，而是重新记录所有的 key 值，假设 key i+1 是一个重启点，那么 key 里面会完整存储 “the color” ，而不是采用简略的 “olor” 方式。Block 尾部就是指出哪些记录是这些重启点的。



知乎 @海茶依依岛

图 4.6 记录格式

在 Block 内容区，每个 KV 记录的内部结构是怎样的？图 4.6 给出了其详细结构，每个记录包含 5 个字段：key 共享长度，比如上面的 “olor” 记录，其 key 和上一条记录共享的 key 部分长度是 “the c” 的长度，即 5；key 非共享长度，对于 “olor” 来说，

是 4; value 长度指出 Key:Value 中 value 的长度, 在后面的 value 内容字段中存储实际的 value 值; 而 key 非共享内容则实际存储 “olor” 这个 key 字符串。

上面讲的这些就是 .sst 文件的全部内部奥秘。

## LevelDB 日知录之五: MemTable 详解

LevelDB 日知录前述小节大致讲述了磁盘文件相关的重要静态结构, 本小节讲述内存中的数据结构 Memtable, Memtable 在整个体系中的重要地位也不言而喻。总体而言, 所有 KV 数据都是存储在 Memtable, Immutable Memtable 和 SSTable 中的, Immutable Memtable 从结构上讲和 Memtable 是完全一样的, 区别仅仅在于其是只读的, 不允许写入操作, 而 Memtable 则是允许写入和读取的。当 Memtable 写入的数据占用内存到达指定数量, 则自动转换为 Immutable Memtable, 等待 Dump 到磁盘中, 系统会自动生成新的 Memtable 供写操作写入新数据, 理解了 Memtable, 那么 Immutable Memtable 自然不在话下。

LevelDB 的 MemTable 提供了将 KV 数据写入, 删除以及读取 KV 记录的操作接口, 但是事实上 Memtable 并不存在真正的删除操作, 删除某个 key 的 value 在 Memtable 内是作为插入一条记录实施的, 但是会打上一个 key 的删除标记, 真正的删除操作是 Lazy

的，会在以后的 Compaction 过程中去掉这个 KV。

需要注意的是，LevelDB 的 Memtable 中 KV 对是根据 key 大小有序存储的，在系统插入新的 KV 时，LevelDB 要把这个 KV 插到合适的位置上以保持这种 key 有序性。其实，LevelDB 的 Memtable 类只是一个接口类，真正的操作是通过背后的 SkipList 来做的，包括插入操作和读取操作等，所以 Memtable 的核心数据结构是一个 SkipList。

SkipList 是由 William Pugh 发明。他在 Communications of the ACM June 1990, 33(6) 668-676 发表了 Skip lists: a probabilistic alternative to balanced trees，在该论文中详细解释了 SkipList 的数据结构和插入删除操作。

SkipList 是平衡树的一种替代数据结构，但是和红黑树不相同的是，SkipList 对于树的平衡的实现是基于一种随机化的算法的，这样也就是说 SkipList 的插入和删除的工作是比较简单的。

关于 SkipList 的详细介绍可以参考这篇文章：

[\\_cnblogs.com/xuqiang/arc\\_](http://cnblogs.com/xuqiang/arc)，讲述的很清楚，LevelDB 的 SkipList 基本上是一个具体实现，并无特殊之处。

SkipList 不仅是维护有序数据的一个简单实现，而且相比较平衡树来说，在插入数据的时候可以避免频繁的树节点调整操作，所以写入效率是很高的，LevelDB 整体而言是个高写入系统，SkipList 在其中应该也起到了很重要的作用。Redis 为了加快插入操作，也使用了 SkipList 来作为内部实现数据结构。

## **LevelDB 日知录之六 写入与删除记录**

在之前的五节 LevelDB 日知录中，我们介绍了 LevelDB 的一些静态文件及其详细布局，从本节开始，我们看看 LevelDB 的一些动态操作，比如读写记录，Compaction，错误恢复等操作。

本节介绍 LevelDB 的记录更新操作，即插入一条 KV 记录或者删除一条 KV 记录。LevelDB 的更新操作速度是非常快的，源于其内部机制决定了这种更新操作的简单性。

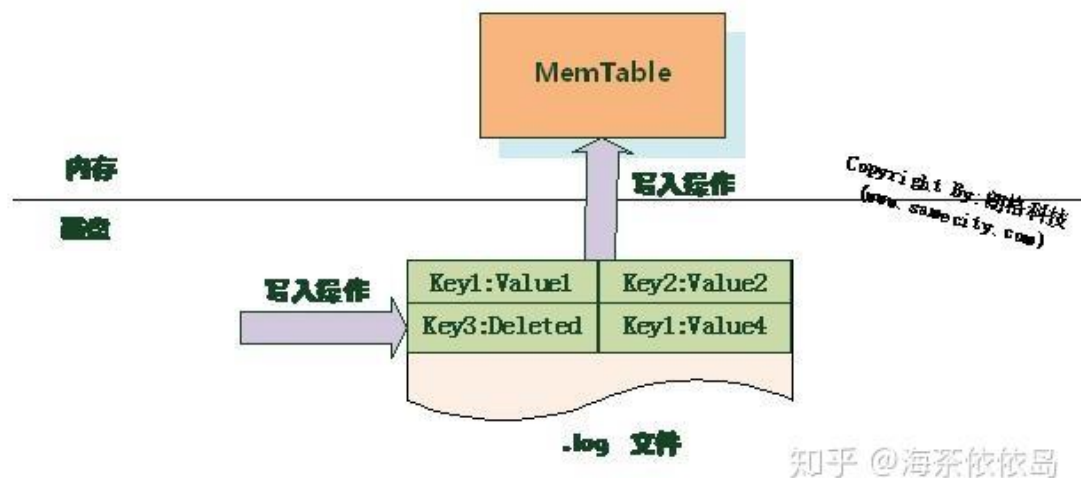


图 6.1 LevelDB 写入记录

图 6.1 是 LevelDB 如何更新 KV 数据的示意图，从图中可以看出，对于一个插入操作  $\text{Put}(\text{Key}, \text{Value})$  来说，完成插入操作包含两个具体步骤：首先是将这条 KV 记录以顺序写的方式追加到之前介绍过的 log 文件末尾，因为尽管这是一个磁盘读写操作，但是文件的顺序追加写入效率是很高的，所以并不会导致写入速度的降低；第二个步骤是：如果写入 log 文件成功，那么将这条 KV 记录插入内存中的 Memtable 中，前面介绍过，Memtable 只是一层封装，其内部其实是一个 key 有序的 SkipList 列表，插入一条新记录的过程也很简单，即先查找合适的插入位置，然后修改相应的链接指针将新记录插入即可。完成这一步，写入记录就算完成了，所以一个插入记录操作涉及一次磁盘文件追加写和内存 SkipList 插入操作，这是为何

LevelDB 写入速度如此高效的根本原因。

从上面的介绍过程中也可以看出：log 文件内是 key 无序的，而 Memtable 中是 key 有序的。那么如果是删除一条 KV 记录呢？对于 LevelDB 来说，并不存在立即删除的操作，而是与插入操作相同的，区别是，插入操作插入的是 Key:Value 值，而删除操作插入的是 “Key:删除标记”，并不真正去删除记录，而是后台 Compaction 的时候才去做真正的删除操作。

LevelDB 的写入操作就是如此简单。真正的麻烦在后面将要介绍的读取操作中。

## **LevelDB 日知录之七：读取记录**

LevelDB 是针对大规模 Key/Value 数据的单机存储库，从应用的角度来看，LevelDB 就是一个存储工具。而作为称职的存储工具，常见的调用接口无非是新增 KV，删除 KV，读取 KV，更新 key 对应的 value 值这么几种操作。LevelDB 的接口没有直接支持更新操作的接口，如果需要更新某个 key 的 value，你可以选择直接生猛地插入新的 KV，保持 key 相同，这样系统内的 key 对应的 value 就会被更新；或者你可以先删除旧的 KV，之后再插入新的 KV，这样比较委婉地完成 KV 的更新操作。

假设应用提交一个 key 值，下面我们看看 LevelDB 是如何从存储的数据中读出其对应的 value 值的。图 7-1 是 LevelDB 读取过程的整体示意图。

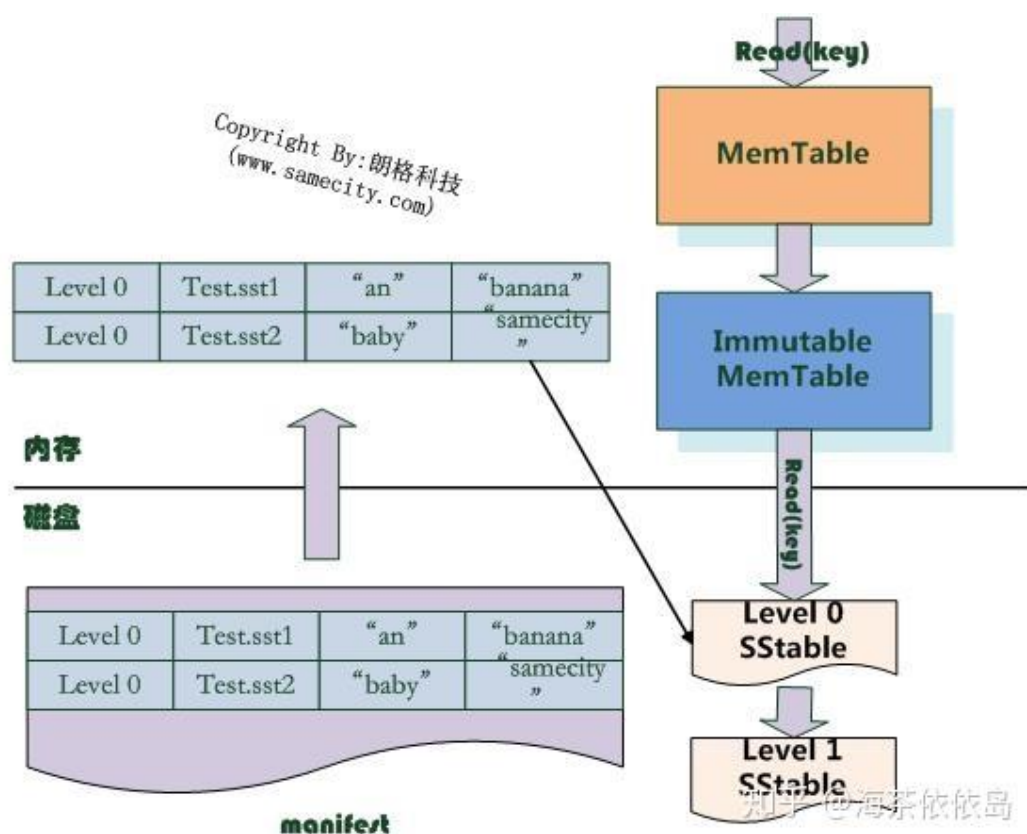


图 7-1 LevelDB 读取记录流程

LevelDB 首先会去查看内存中的 Memtable，如果 Memtable 中包含 key 及其对应的 value，则返回 value 值即可；如果在 Memtable 没有读到 key，则接下来到同样处于内存中的 Immutable Memtable 中去读取，类似地，如果读到就返回，若是没有读到，那么只能万般无奈下从磁盘中的大量 SStable 文件中查

找。因为 SSTable 数量较多，而且分成多个 Level，所以在 SSTable 中读数据是相当蜿蜒曲折的一段旅程。总的读取原则是这样的：首先从属于 Level 0 的文件中查找，如果找到则返回对应的 value 值，如果没有找到那么到 Level 1 中的文件中去找，如此循环往复，直到在某层 SSTable 文件中找到这个 key 对应的 value 为止（或者查到最高 Level，查找失败，说明整个系统中不存在这个 key）。

那么为什么是从 Memtable 到 Immutable Memtable，再从 Immutable Memtable 到文件，而文件中为何是从低 Level 到高 Level 这么一个查询路径呢？道理何在？之所以选择这么个查询路径，是因为从信息的更新时间来说，很明显 Memtable 存储的是最鲜的 KV 对；Immutable Memtable 中存储的 KV 数据对的新鲜程度次之；而所有 SSTable 文件中的 KV 数据新鲜程度一定不如内存中的 Memtable 和 Immutable Memtable 的。对于 SSTable 文件来说，如果同时在 Level L 和 Level L+1 找到同一个 key，level L 的信息一定比 level L+1 的要新。也就是说，上面列出的查找路径就是按照数据新鲜程度排列出来的，越新鲜的越先查找。

为啥要优先查找新鲜的数据呢？这个道理不言而喻，举个例子。比如我们先往 LevelDB 里面插入一条数据 {key="[samecity.com](http://www.samecity.com)" value="朗格科技"}，过了几天，samecity 网站改名为：69 同城，



此时我们插入数据{key="\_samecity.com" value="69 同城"}, 同样的 key, 不同的 value; 逻辑上理解好像 LevelDB 中只有一个存储记录, 即第二个记录, 但是在 LevelDB 中很可能存在两条记录, 即上面的两个记录都在 LevelDB 中存储了, 此时如果用户查询 key="\_samecity.com", 我们当然希望找到最新的更新记录, 也就是第二个记录返回, 这就是为何要优先查找新鲜数据的原因。

前文有讲: 对于 SSTable 文件来说, 如果同时在 Level L 和 Level L+1 找到同一个 key, Level L 的信息一定比 Level L+1 的要新。这是一个结论, 理论上需要一个证明过程, 否则会招致如下的问题: 为神马呢? 从道理上讲呢, 很明白: 因为 Level L+1 的数据不是从石头缝里蹦出来的, 也不是做梦梦到的, 那它是从哪里来的? Level L+1 的数据是从 Level L 经过 Compaction 后得到的 (如果您不知道什么是 Compaction, 那么.....也许以后会知道的), 也就是说, 您看到的现在的 Level L+1 层的 SSTable 数据是从原来的 Level L 中来的, 现在的 Level L 比原来的 Level L 数据要新鲜, 所以可证, 现在的 Level L 比现在的 Level L+1 的数据要新鲜。

SSTable 文件很多, 如何快速找到 key 对应的 value 值? 在 LevelDB 中, Level 0 一直都爱搞特殊化, 在 Level 0 和其它 Level 中查找某个 key 的过程是不一样的。因为 level 0 下的不同文件可能 key 的范围有重叠, 某个要查询的 key 有可能多个文件都包含, 这

样的话 LevelDB 的策略是先找出 level 0 中哪些文件包含这个 key (manifest 文件中记载了 Level 和对应的文件及文件里 key 的范围信息, LevelDB 在内存中保留这种映射表), 之后按照文件的新鲜程度排序, 新的文件排在前面, 之后依次查找, 读出 key 对应的 value。而如果是非 Level 0 的话, 因为这个 Level 的文件之间 key 是不重叠的, 所以只从一个文件就可以找到 key 对应的 value。

最后一个问题, 如果给定一个要查询的 key 和某个 key range 包含这个 key 的 SSTable 文件, 那么 LevelDB 是如何进行具体查找过程的呢? LevelDB 一般会先在内存中的 Cache 中查找是否包含这个文件的缓存记录, 如果包含, 则从缓存中读取; 如果不包含, 则打开 SSTable 文件, 同时将这个文件的索引部分加载到内存中并放入 Cache 中。这样 Cache 里面就有了这个 SSTable 的缓存项, 但是只有索引部分在内存中, 之后 LevelDB 根据索引可以定位到哪个 Data Block 会包含这条 key, 从文件中读出这个 Block 的内容, 再根据记录一一比较, 如果找到则返回结果, 如果没有找到, 那么说明这个 Level 的 SSTable 文件并不包含这个 key, 所以到下一级别的 SSTable 中去查找。

从之前介绍的 LevelDB 的写操作和这里介绍的读操作可以看出, 相对写操作, 读操作处理起来要复杂很多, 所以写的速度必然要远远高于读数据的速度, 也就是说, **LevelDB 比较适合写操作多于读操作的**

**应用场合。而如果应用是很多读操作类型的，那么顺序读取效率会比较高，因为这样大部分内容都会在缓存中找到，尽可能避免大量的随机读取操作。**

## **LevelDB 日知录之八：Compaction 操作**

前文有述，对于 LevelDB 来说，写入记录操作很简单，删除记录仅仅写入一个删除标记就算完事，但是读取记录比较复杂，需要在内存以及各个层级文件中依照新鲜程度依次查找，代价很高。为了加快读取速度，LevelDB 采取了 compaction 的方式来对已有的记录进行整理压缩，通过这种方式，来删除掉一些不再有效的 KV 数据，减小数据规模，减少文件数量等。

LevelDB 的 compaction 机制和过程与 Bigtable 所讲述的是基本一致的，Bigtable 中讲到三种类型的 compaction: minor, major 和 full。所谓 minor Compaction，就是把 Memtable 中的数据导出到 SSTable 文件中；major compaction 就是合并不同层级的 SSTable 文件，而 full compaction 就是将所有 SSTable 进行合并。

LevelDB 包含其中两种，minor 和 major。

我们将为大家详细叙述其机理。

先来看看 minor Compaction 的过程。Minor compaction 的目的是当内存中的 Memtable 大小到了一定值时，将内容保存到磁盘文件中，图 8.1 是其机理示意图。

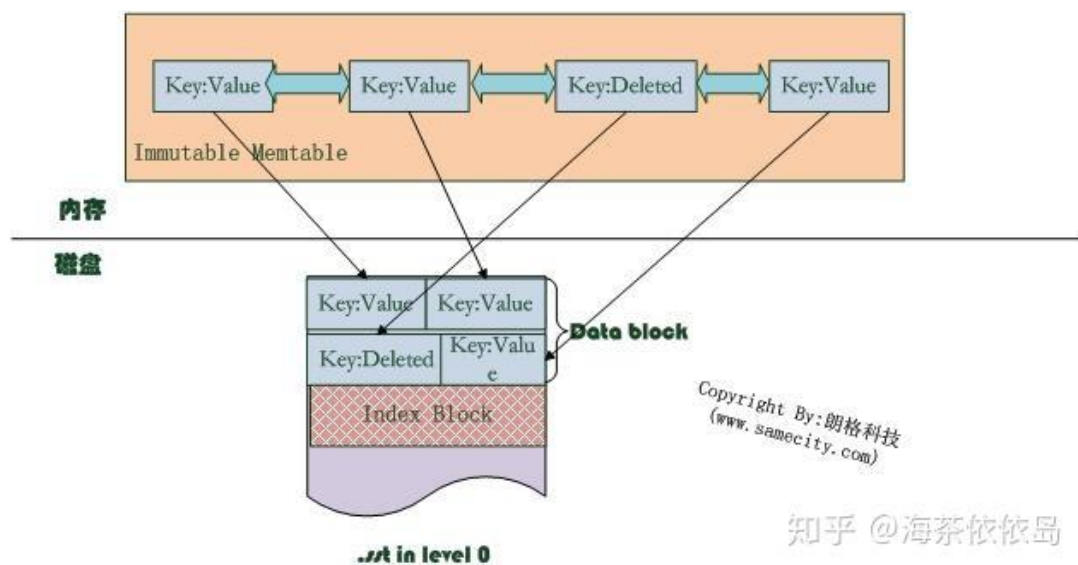


图 8.1 minor compaction

从 8.1 可以看出，当 Memtable 数量到了一定程度会转换为 Immutable Memtable，此时不能往其中写入记录，只能从中读取 KV 内容。之前介绍过，Immutable Memtable 其实是一个多层级队列 SkipList，其中的记录是根据 key 有序排列的。所以这个 minor compaction 实现起来也很简单，就是按照 Immutable Memtable 中记录由小到大遍历，并依次写入一个 Level 0 的新建 SSTable 文件中，写完后建立文件的 index 数据，这样就完成了一次 minor

compaction。从图中也可以看出，对于被删除的记录，在 minor compaction 过程中并不真正删除这个记录，原因也很简单，这里只知道要删掉 key 记录，但是这个 KV 数据在哪里？那需要复杂的查找，所以在 minor compaction 的时候并不做删除，只是将这个 key 作为一个记录写入文件中，至于真正的删除操作，在以后更高级别的 compaction 中会去做。

当某个 Level 下的 SSTable 文件数目超过一定设置值后，LevelDB 会从这个 Level 的 SSTable 中选择一个文件 ( $\text{Level} > 0$ )，将其和高一层级的  $\text{Level} + 1$  的 SSTable 文件合并，这就是 major compaction。

我们知道在大于 0 的层级中，每个 SSTable 文件内的 key 都是由小到大有序存储的，而且不同文件之间的 key 范围（文件内最小 key 和最大 key 之间）不会有任何重叠。Level 0 的 SSTable 文件有些特殊，尽管每个文件也是根据 key 由小到大排列，但是因为 Level 0 的文件是通过 minor compaction 直接生成的，所以任意两个 Level 0 下的两个 sstable 文件可能在 key 范围上有重叠。所以在做 major compaction 的时候，对于大于 Level 0 的层级，选择其中一个文件就行，但是对于 Level 0 来说，指定某个文件后，本 Level 中很可能有其它 SSTable 文件的 key 范围和这个文件有重叠，这种情况下，要找出所有有重叠的文件和 Level 1 的文件进行合

并，即 Level 0 在进行文件选择的时候，可能会有多个文件参与 major compaction。

LevelDB 在选定某个 Level 进行 compaction 后，还要选择是具体哪个文件要进行 compaction，LevelDB 在这里有个小技巧，就是说轮流来，比如这次是文件 A 进行 compaction，那么下次就是在 key range 上紧挨着文件 A 的文件 B 进行 compaction，这样每个文件都会有机会轮流和高层 Level 的文件进行合并。

如果选好了 Level L 的文件 A 和 Level L+1 的文件进行合并，那么问题又来了，应该选择 Level L+1 哪些文件进行合并？LevelDB 选择 L+1 层中和文件 A 在 key range 上有重叠的所有文件来和文件 A 进行合并。

也就是说，选定了 Level L 的文件 A，之后在 Level L+1 中找到了所有需要合并的文件 B，C，D.....等等。剩下的问题就是具体是如何进行 major 合并的？就是说给定了一系列文件，每个文件内部是 key 有序的，如何对这些文件进行合并，使得新生成的文件仍然 key 有序，同时抛掉哪些不再有价值的 KV 数据。

图 8.2 说明了这一过程。

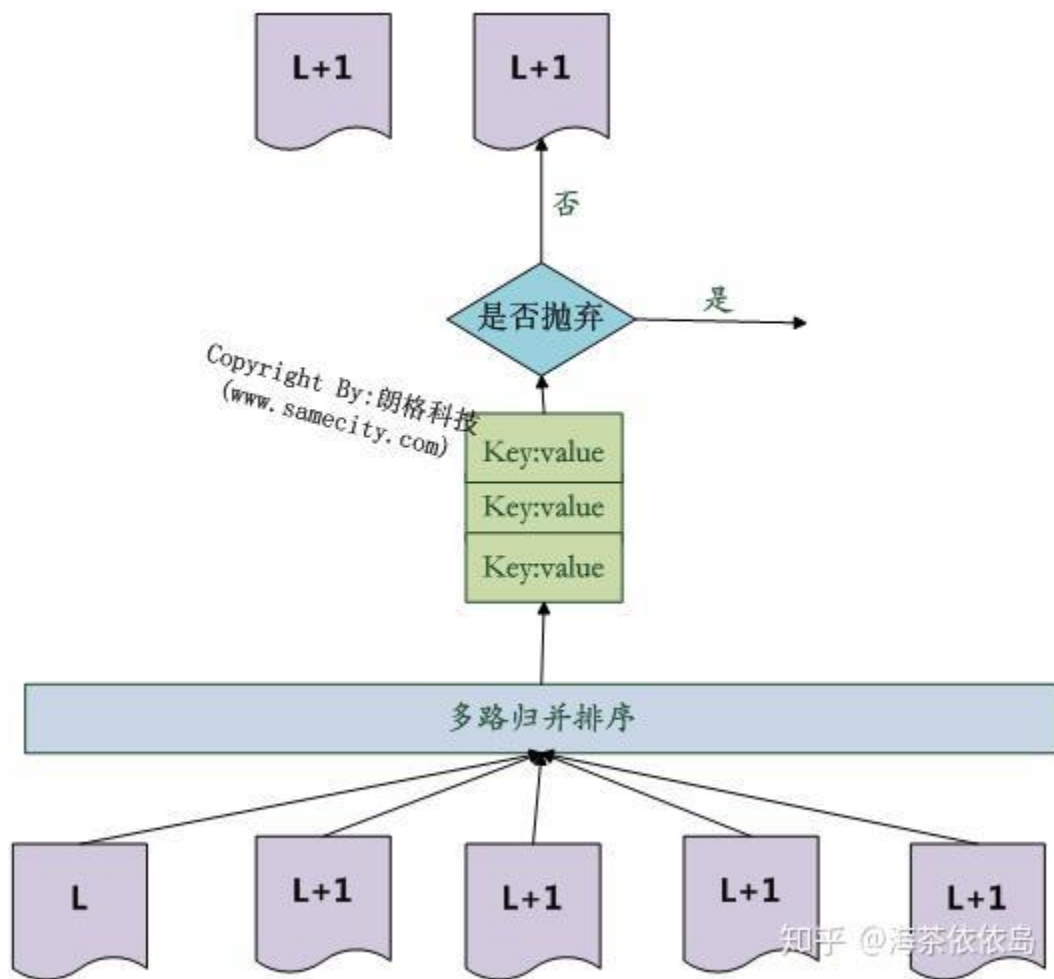


图 8.2 SSTable Compaction

Major compaction 的过程如下：对多个文件采用多路归并排序的方式，依次找出其中最小的 key 记录，也就是对多个文件中的所有记录重新进行排序。之后采取一定的标准判断这个 key 是否还需要保存，如果判断没有保存价值，那么直接抛掉，如果觉得还需要继续保存，那么就将其写入 Level L+1 层中新生成的一个 SSTable 文件中。就这样对 KV 数据——处理，形成了一系列新的 L+1 层数据文件，之

前的 L 层文件和 L+1 层参与 compaction 的文件数据此时已经没有任何意义了，所以全部删除。这样就完成了 L 层和 L+1 层文件记录的合并过程。

那么在 major compaction 过程中，判断一个 KV 记录是否抛弃的标准是什么呢？其中一个标准是：对于某个 key 来说，如果在小于 L 层中存在这个 key，那么这个 KV 在 major compaction 过程中可以抛掉。因为我们前面分析过，对于层级低于 L 的文件中如果存在同一 key 的记录，那么说明对于 key 来说，有更新鲜的 value 存在，那么过去的 value 就等于没有意义了，所以可以删除。

## **LevelDB 日知录之九 LevelDB 中的 Cache**

书接前文，前面讲过对于 LevelDb 来说，读取操作如果没有在内存的 Memtable 中找到记录，要多次进行磁盘访问操作。假设最优情况，即第一次就在 Level 0 中最新的文件中找到了这个 key，那么也需要读取 2 次磁盘，一次是将 SSTable 的文件中的 index 部分读入内存，这样根据这个 index 可以确定 key 是在哪个 block 中存储；第二次是读入这个 block 的内容，然后在内存中查找 key 对应的 value。

LevelDB 中引入了两个不同的 Cache: Table Cache 和 Block



Cache。其中 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。

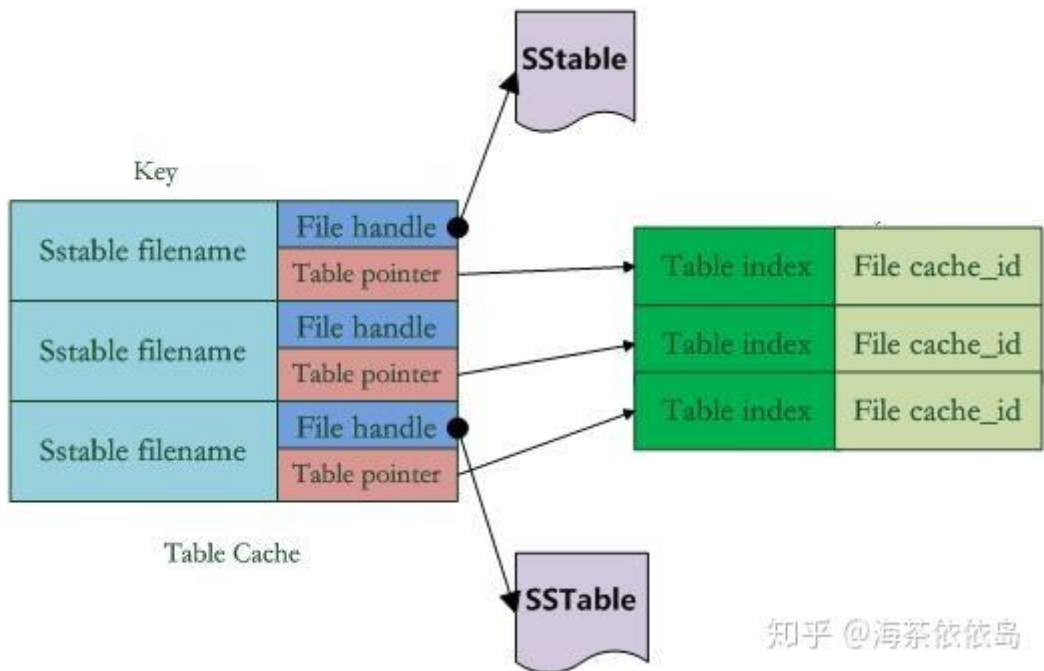


图 9.1 Table Cache

图 9.1 是 Table Cache 的结构。在 Cache 中，key 值是 SStable 的文件名称，value 部分包含两部分，一个是指向磁盘打开的 SStable 文件的文件指针，这是为了方便读取内容；另外一个是指向内存中这个 SStable 文件对应的 table 结构指针，table 结构在内存中，保存了 SStable 的 index 内容以及用来指示 block cache 用的 cache\_id，当然除此外还有其它一些内容。

比如在 get(key) 读取操作中，如果 LevelDB 确定了 key 在某个

Level 下某个文件 A 的 key range 范围内，那么需要判断是不是文件 A 真的包含这个 KV。此时，LevelDB 会首先查找 Table Cache，看这个文件是否在缓存里，如果找到了，那么根据 index 部分就可以查找是哪个 block 包含这个 key。如果没有在缓存中找到文件，那么打开 SSTable 文件，将其 index 部分读入内存，然后插入 Cache 里面，去 index 里面定位哪个 block 包含这个 key。如果确定了文件哪个 block 包含这个 key，那么需要读入 block 内容，这是第二次读取。

File cache_id+block_offset	block 内容
File cache_id+block_offset	block 内容
File cache_id+block_offset	block 内容
File cache_id+block_offset	block 内容

Block Cache 知乎 @海茶依依岛

图 9.2 Block Cache

Block Cache 是为了加快这个过程的，图 9.2 是其结构示意图。其中的 key 是文件的 cache\_id 加上这个 block 在文件中的起始位置 block\_offset。而 value 则是这个 Block 的内容。

如果 LevelDB 发现这个 block 在 block cache 中，那么可以避免读取数据，直接在 cache 里的 block 内容里面查找 key 的 value 就行，如果没找到呢？那么读入 block 内容并把它插入 block cache 中。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，您可以推断下其效率如何。

## LevelDB 日知录之十 Version、VersionEdit、VersionSet

Version 保存了当前磁盘以及内存中所有的文件信息，一般只有一个 Version 叫做 "current" version（当前版本）。LevelDB 还保存了一系列的历史版本，这些历史版本有什么作用呢？

当一个 Iterator 创建后，Iterator 就引用到了 current version(当前版本)，只要这个 Iterator 不被 delete 那么被 Iterator 引用的版本就会一直存活。这就意味着当你用完一个 Iterator 后，需要及时删除它。

当一次 Compaction 结束后（会生成新的文件，合并前的文件需要删除），LevelDB 会创建一个新的版本作为当前版本，原先的当前版本就会变为历史版本。

VersionSet 是所有 Version 的集合，管理着所有存活的 Version。

VersionEdit 表示 Version 之间的变化，相当于 delta 增量，表示有增加了多少文件，删除了文件。下图表示他们之间的关系。

Version0 + VersionEdit --> Version1

VersionEdit 会保存到 MANIFEST 文件中，当做数据恢复时就会从 MANIFEST 文件中读出来重建数据。

LevelDB 的这种版本的控制，让我想到了双 buffer 切换，双 buffer 切换来自于图形学中，用于解决屏幕绘制时的闪屏问题，在服务器编程中也有用处。

比如我们的服务器上有一个字典库，每天我们需要更新这个字典库，我们可以新开一个 buffer，将新的字典库加载到这个新 buffer 中，等到加载完毕，将字典的指针指向新的字典库。

LevelDB 的 version 管理和双 buffer 切换类似，但是如果原 version 被某个 iterator 引用，那么这个 version 会一直保持，直到没有被任何一个 iterator 引用，此时就可以删除这个 version。

# LevelDB 和 RocksDB 结构详解

阅读本文之前建议对 LSM 树有一定的认识。本文将介绍 LSM Tree 的主流实现，即 LevelDB 和 RocksDB 作为 KV 数据库。

两者对外提供的主要接口基本一致，就是包含以下 5 个基本操作

- get(K) 查找 key K 对应的 value
- put(K,V) 插入键值对 (K, V)
- update(K,V) 查找 key K 对应的 value 更新为 V
- delete(K) 删除 key K 对应的条目
- scan(K1,K2) 得到从 K1 到 K2 的所有 key 和 value

我们将从静态和动态角度去介绍两个数据库

先从 LevelDB 开始，相对好理解。毕竟 RocksDB 是在 levelDB 上做的改进

## LevelDB

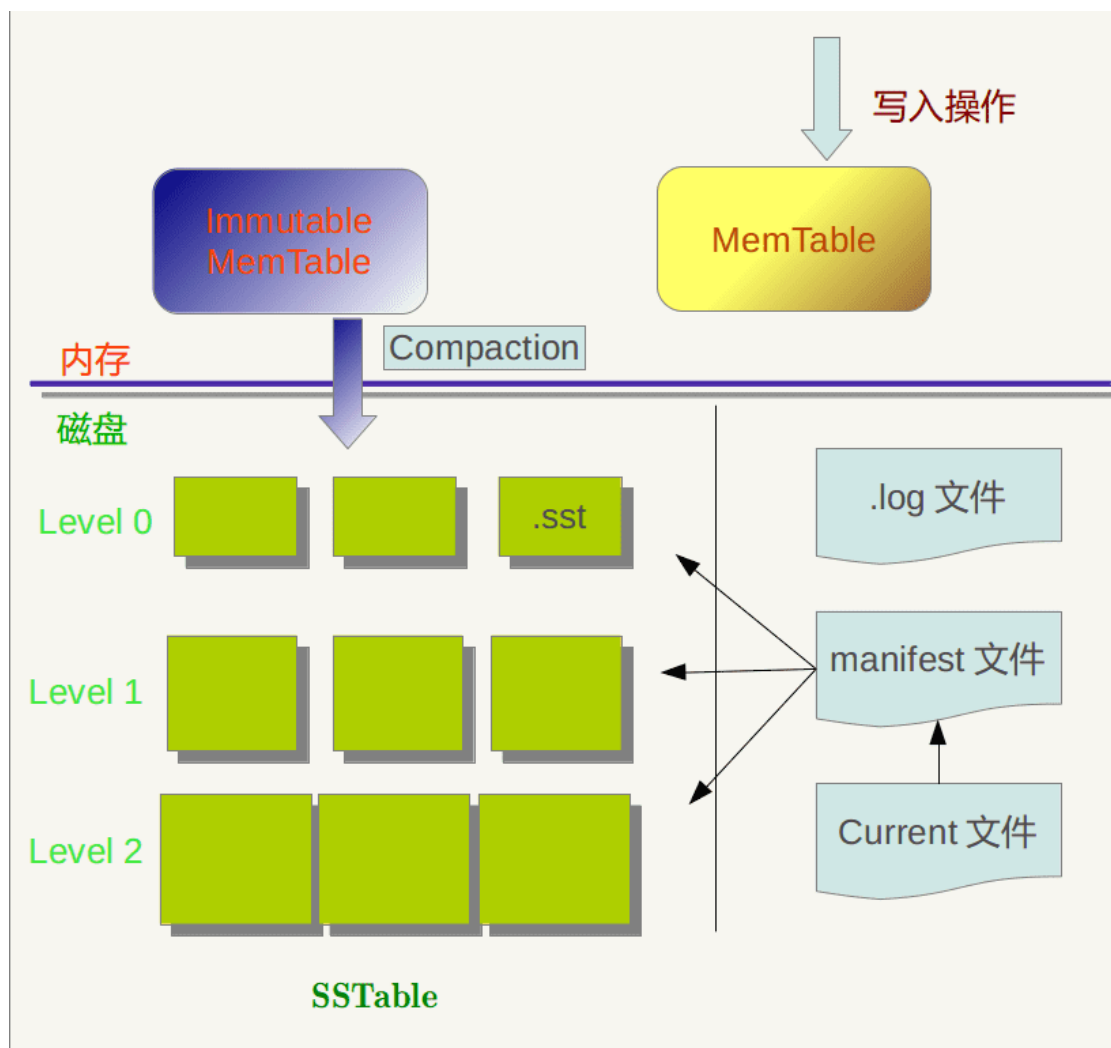
---

接下从静态和动态角度去介绍

静态视角：假想整个系统正在运行过程中（不断插入删除读取数据），此时我们给 LevelDb 照相，从照片可以看到之前系统的数据在内存和磁盘中是如何分布的，处于什么状态等。

动态视角：了解系统是如何写入一条记录，读出一条记录，删除一条记录的，同时也包括除了这些接口操作外的内部操作比如 compaction，系统运行时崩溃后如何恢复系统等方面。

## 架构 - 静态视角



## 内存（对应 LSM 的 C0）

- o memtable
  - o 结构: **SKIPLIST**，和 immutable memtable 完全相同。

- 读写：允许读写
  - 功能：当 Memtable 写入的数据占用内存到达指定数量，则自动转换为 Immutable Memtable，等待 Dump 到磁盘中，系统会自动生成新的 Memtable 供写操作写入新数据
  - 删除：并不存在真正的删除操作,删除某个 Key 的 Value 在 Memtable 内是作为插入一条记录实施的, 但是会打上一个 Key 的删除标记, 真正的删除操作是 Lazy 的, 会在以后的 Compaction 过程中去掉这个 KV
  - 重启时：会从 log 中恢复。
- immutable memtable
    - 正在进行写入磁盘操作的 memtable

## 磁盘中

---

- log(WAL)
  - 属于 write-ahead-log, 每个写入操作, 都会往 log 尾部添加一个完整的 kv 记录
  - 主作用是故障恢复, 可以用于恢复 memtable 和 immutable memtable
- current

current 指出当前有效的那个 manifest 是哪个

随着 Compaction 进行, sstable 变化, manifest 会记录这些变化

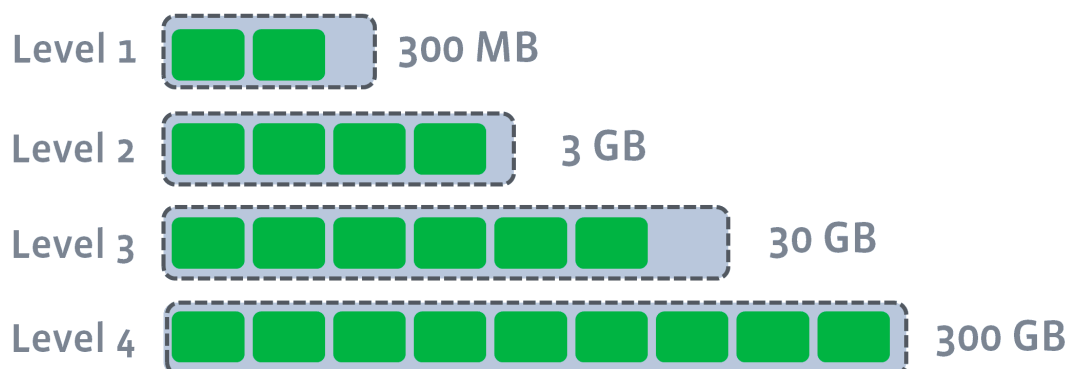
- manifest

记载 sst 各文件的信息(LEVEL, NAME, MIN\_KEY, MAX\_KEY)

Level 0	Test1.sst	"abc"	"hello"
Level 0	Test2.sst	"bbc"	"world"

Manifest

- SST 文件 (Semi-sort table)   **(对应 LSM 的 C1-N)**
  - 文件中 key 有序存储，存储一个范围(K1,K2)之间的键值对
  - Level 0 的 SST 之间可能存在 key 重叠，Level 1+ 的 SST 不会有 key 重叠
  - 所有文件是一种层级结构，第一层为 Level 0，第二层为 Level 1，层级逐渐增高，每一层的容量也会增大，这也是称为 LevelDB 的原因。这 level 的设计上 LevelDB 和 RocksDB 完全一致。其大小关系往往是每个级别差 10 倍，如下图：



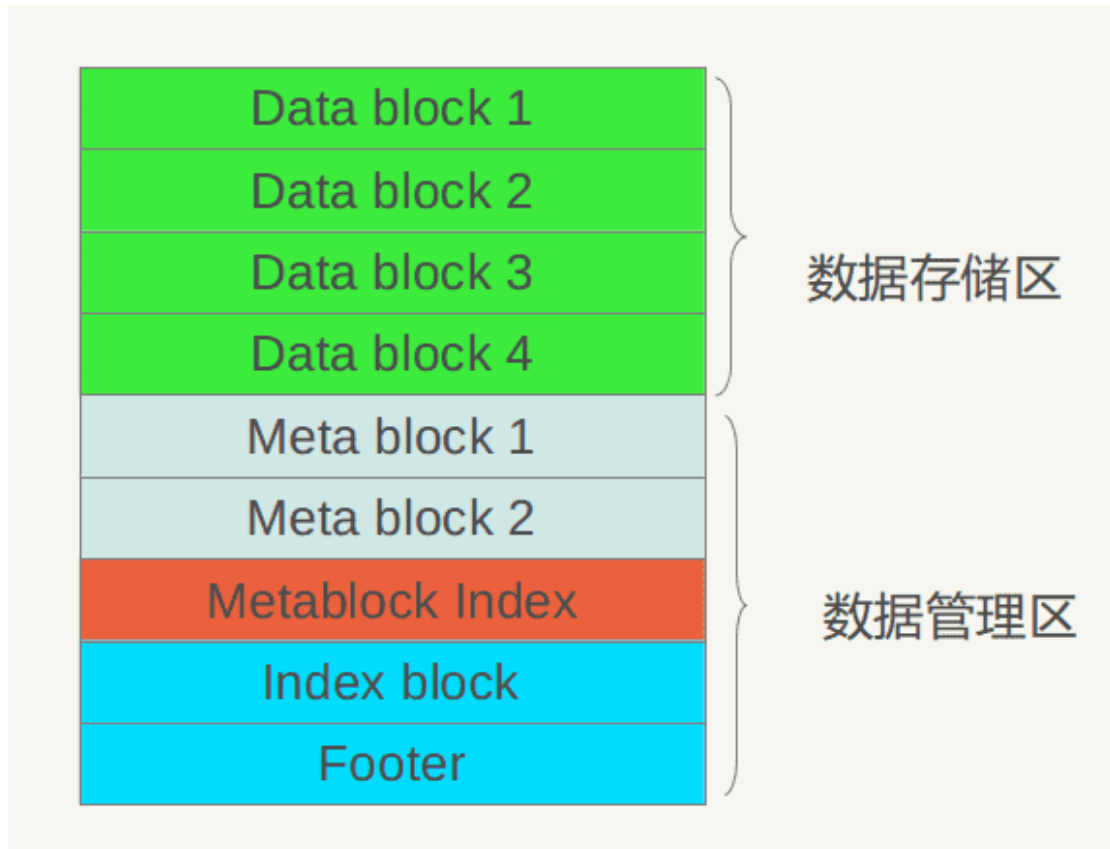
## SST 文件 (对应 LSM 的 C1-N)



物理布局

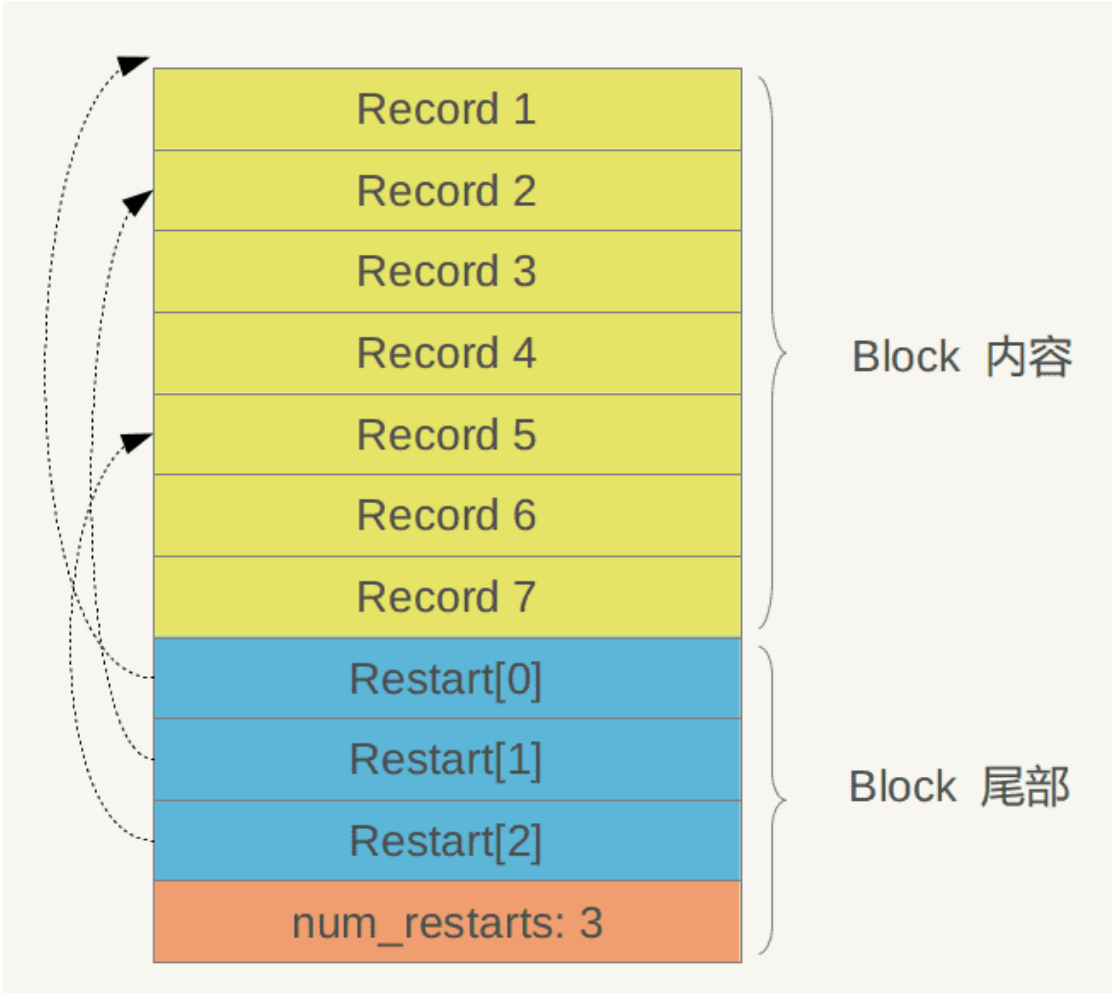
Block 1	Type	CRC
Block 2	Type	CRC
Block 3	Type	CRC
Block 4	Type	CRC
Block 5	Type	CRC
Block 6	Type	CRC
Block 7	Type	CRC
Block 8	Type	CRC

## 逻辑结构



数据存储区

存储实际的 key-value，单个 block 里面内容如下

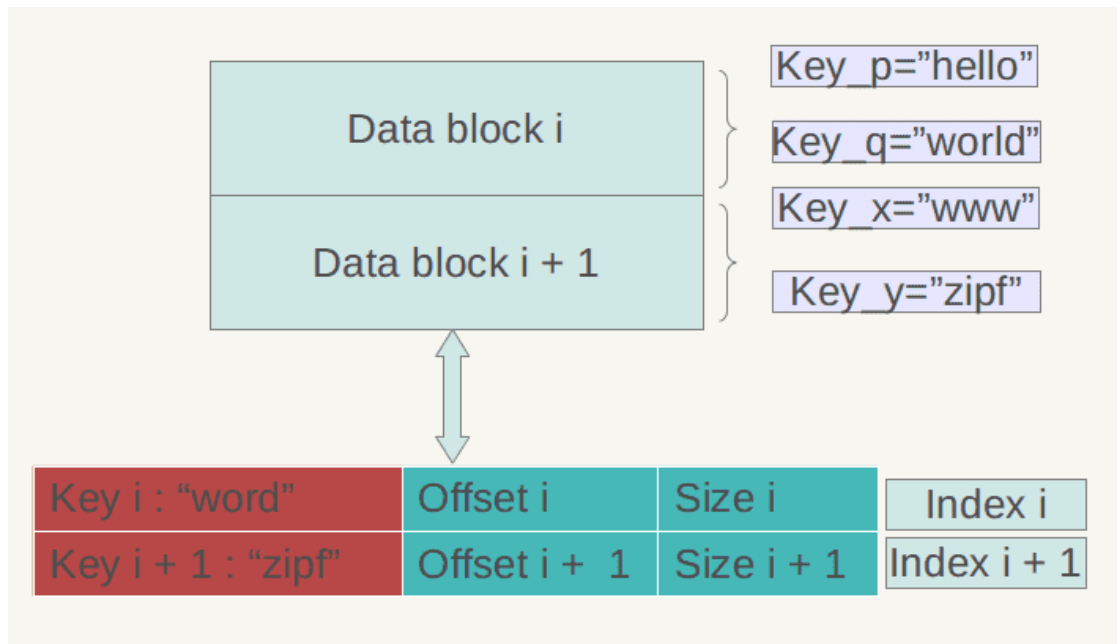


“重启点”（Restart Point），其实是一些指针，**为了降低数据冗余** 指出 Block 内容中的一些记录位置。在这条记录开始，不再采取只记载不同的 Key 部分，而是重新记录所有的 Key 值。

Record i	key共享长度	key非共享长度	value长度	key非共享内容	value内容
Record i+1	key共享长度	key非共享长度	value长度	key非共享内容	value内容

## 数据管理区

- meta block : 记录这个 SST 文件的一些元信息, 比如 record 个数, 数据大小等
- footer : 指向 (索引) index block 的 index
- **index block**: 指向 data block 在文件中的地址, 用于查找数据在哪个 block 内



## 操作 - 动态视角

### 写 - 插入、更新、删除数据

- 插入: 同步操作就是 (决定延迟), 就是先顺序写入 log (内部无序), 然后写入 memtable。异步写入后续就是要看 compaction。
- 更新: 当做插入处理。
- 删除: 不是直接删除, 而是加入了一个删除标记

## compaction

目的：完成数据的沉降，同时由于之前的插入、删除操作，数据有很多的冗余。

压缩，然后删除掉一些不再有效的 KV 数据，减小数据规模，减少文件数。

### flush (Memtable -> LEVEL0)

- 触发条件： memtable 大小超过阈值
- 结果： 其过程就是把 immutable memtable 简单持久化一个新 sst
- 过程：依次写入 sst，然后新建索引（所以一个 level 0 sst 数据大小和 immutable memtable 是一样的）
- 注意： 因为不考虑其他 sst，所以 level0 的 sst 键会存在会重叠

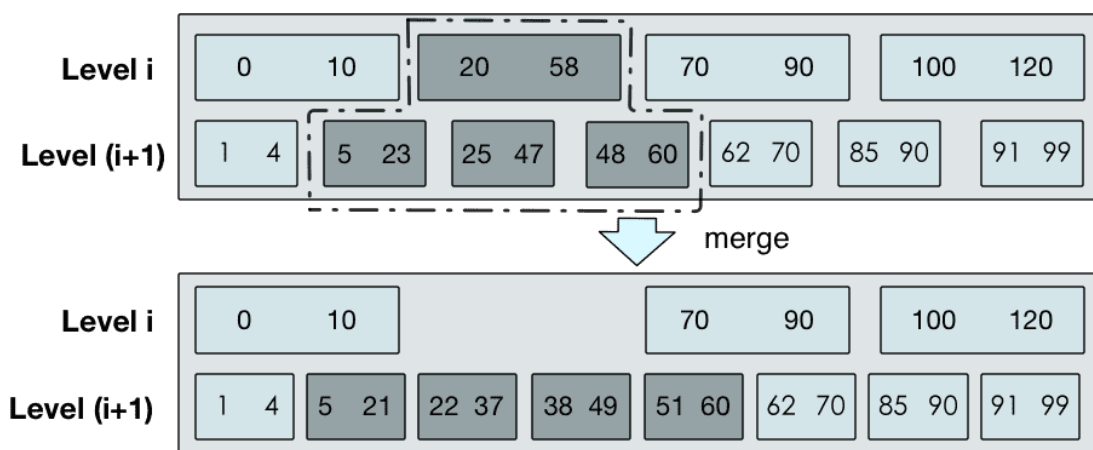
### compaction (Level i -> Level i+1)

就是按文件产生次序轮询，合并之后原文件就失效等待删除，新的文件生效

(manifest 记录)

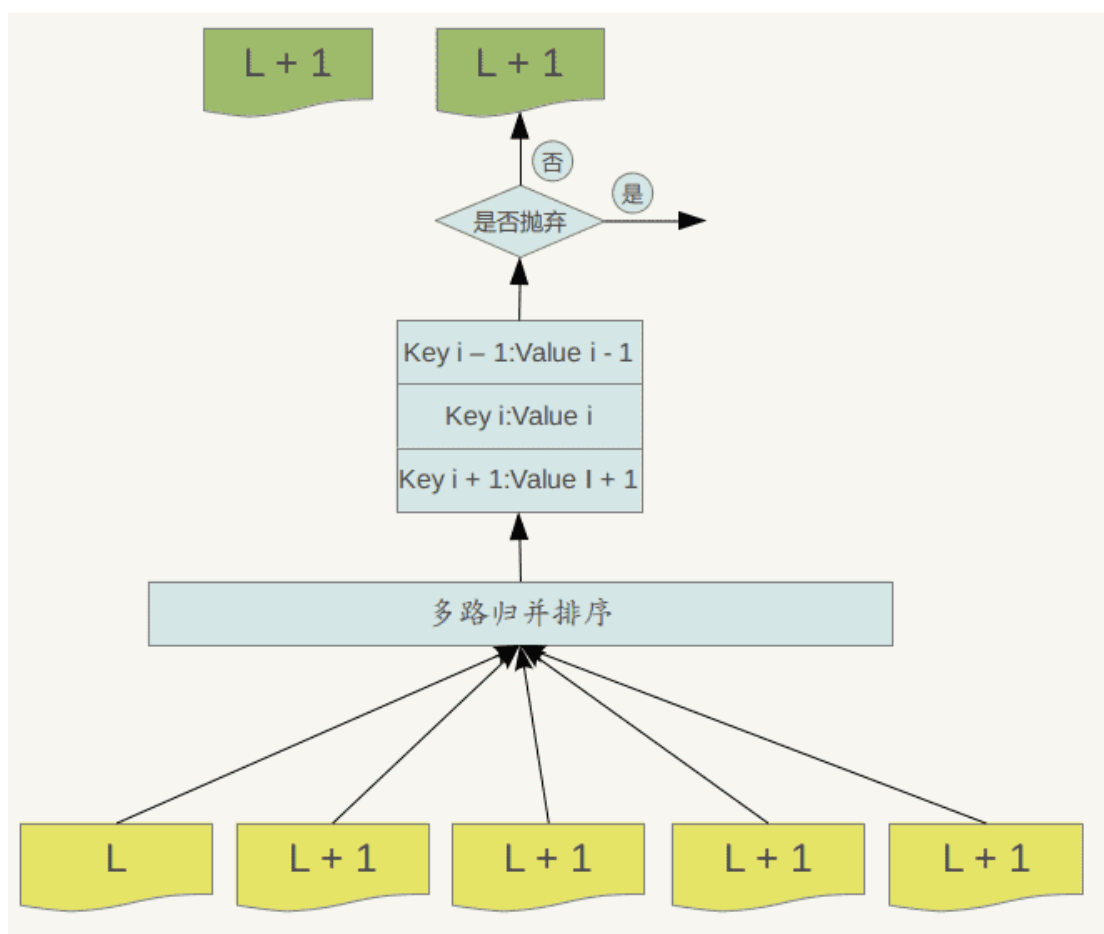
- 触发条件 level i 大小或者文件个数超过阈值
- 过程：以 level i 的一个文件为驱动，在 level i 找和 level i+1 找存在 key 重叠的 sst，然后滚动合并
  - 当 i 是 0 时，因为 leveli 文件之间存在重叠，所以是 leveli 和 leveli+1 的多个文件一起合并。
  - 当 i 是 1 时，当因为 leveli 的 sst 之间没有重叠，所以就是一个 leveli 的文件和多个 L+1 合并

- 结果图示( $i \geq 1$  情况):



- 合并过程图示 (即一个多路归并排序,图中 L 代表 level i 的文件, L+1 代表 level i+1

的文件):



## 读

先内存后磁盘，硬盘从上层至下层。（找到数据就返回，因为 level 小的里面的记录肯定是最新的）

先在 memtable 里面找,如果找不到则从磁盘文件中,对于磁盘上的每一个 level, 查找有三级：

1. sst 定位：优先新鲜(level 小的)sst，唯一特殊的就是 level 0 不同 sst 可能会有 key 重叠，所以要在 manifest 中找到符合条件的文件，按**从新到旧的顺序**依次查找。
2. sst 内查找：先看 sst 的索引是否已经加载到 Cache，没有就加载,然后定位 block
3. block 内查找：先在 restart points 上找，然后 restart points 之间顺序查找

## 缓存 Cache

为了加快查询，数据库都会将数据放入内存，来降低重复访问同一段数据带来的开销。

如：

1. manifest 内容存在缓存中
2. 打开一个 sst 时，其索引块也会加载到缓存中
3. 在一个 block 内查找时，也会将 block 加入到缓存

## RocksDB

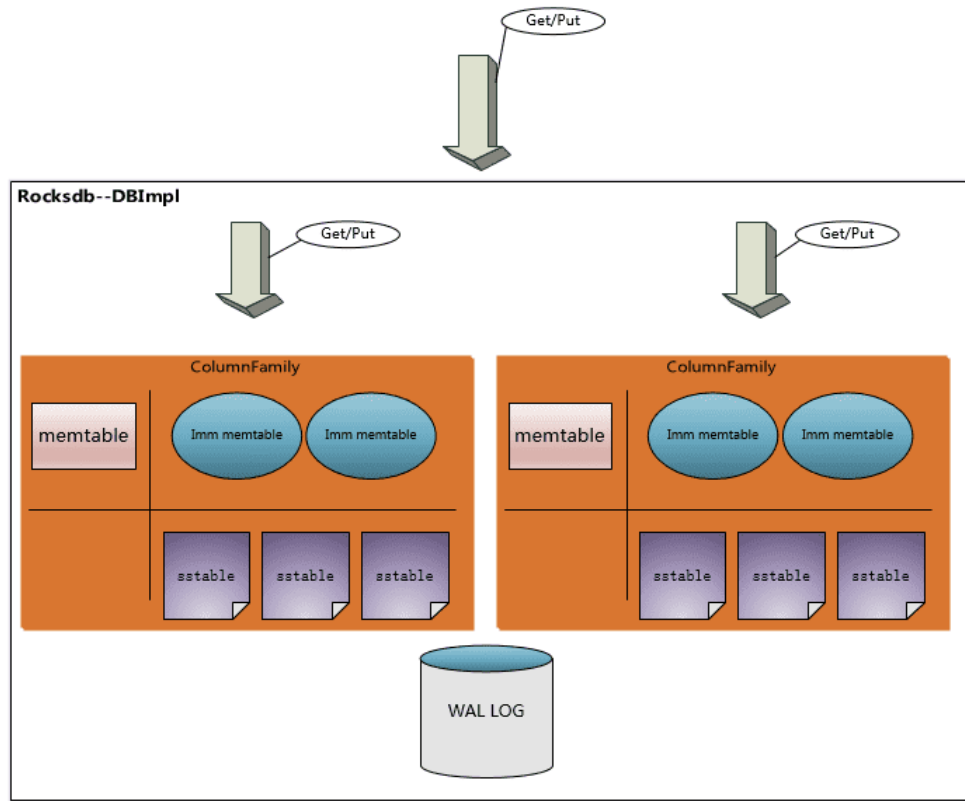
---

结构和 levelDB 大同小异，只是多了一些改进

1. 增加了 column family, 有了列簇的概念, 可把一些相关的 key 存储在一起
2. 内存中有多个 immutable memtable, 可防止 LevelDB 中的 write stall
3. 可支持多线程同时 compaction, 理论上多线程同时 compaction 会比一个线程 compaction 要快 (TODO 执行 compaction 的中心在哪)
4. 支持 TTL
5. flush 与 compaction 分开不同的线程池来调度, 并具有不同的优先级, flush 要优于 compaction, 这样可以加快 flush, 防止 stall
6. 对 SSD 存储做了优化, 可以以 in-memory 方式运行
7. 增加了对 write ahead log (WAL) 的管理机制, 更方便管理 WAL, WAL 是 binlog 文件
8. 支持多种不同的 compaction 策略



## 结构



## 特性

同 LSM Tree，唯一区别在于每一层分成了多个文件。

- 优点：
  - 写延迟低
  - 访问新数据更快，适合时序、实时存储
  - 空间放大率低
- 缺点：
  - 写放大、读放大高
  - 读放大

- 磁盘上修改数据的粒度必须是文件

请设计一个 Key-Value 存储引擎(Design a key-value store)。

这是一道频繁出现的题目，个人认为也是一道很好的题目，这题纵深非常深，内行的人可以讲的非常深。

首先讲两个术语，**数据库**和**存储引擎**。数据库往往是一个比较丰富完整的系统，提供了 SQL 查询语言，事务和水平扩展等支持。然而存储引擎则是小而精，纯粹专注于单机的读/写/存储。一般来说，数据库底层往往会使用某种存储引擎。

目前开源的 KV 存储引擎中，RocksDB 是流行的一个，MongoDB 和 MySQL 底层可以切换到 RocksDB，TiDB 底层直接使用了 RocksDB。大多数分布式数据库的底层不约而同的都选择了 RocksDB。

RocksDB 最初是从 LevelDB 进化而来的，我们先从简单一点的 LevelDB 入手，借鉴它的设计思路。

## LevelDB 整体结构

---

有一个反直觉的事情是，**内存随机写甚至比硬盘的顺序读还要慢**，磁盘随机写就更慢了，说明我们要避免随机写，最好设计成顺序写。因此好的 KV 存储引擎，都在尽量避免更新操作，把更新和删除操作转化为顺序写操作。LevelDB 采用了一种 SSTable 的数据结构来达到这个目的。

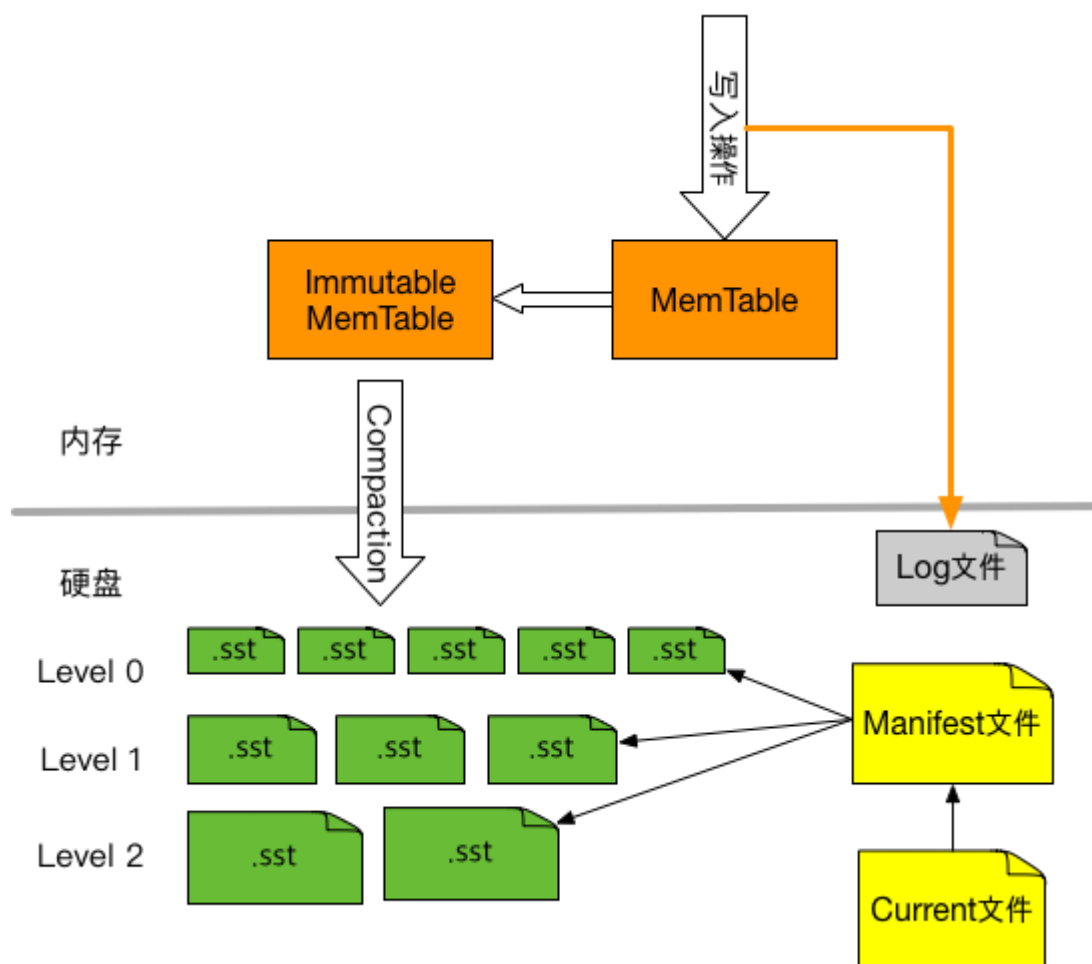
SSTable(Sorted String Table)就是一组按照 key 排序好的 key-value 对, key 和 value 都是字节数组。SSTable 既可以在内存中, 也可以在硬盘中。

SSTable 底层使用 LSM Tree(Log-Structured Merge Tree)来存放有序的 key-value 对。

LevelDB 整体由如下几个组成部分,

1. MemTable。即内存中的 SSTable, 新数据会写入到这里, 然后批量写入磁盘, 以此提高写的吞吐量。
2. Log 文件。写 MemTable 前会写 Log 文件, 即用 WAL(Write Ahead Log)方式记录日志, 如果机器突然掉电, 内存中的 MemTable 丢失了, 还可以通过日志恢复数据。WAL 日志是很多传统数据库例如 MySQL 采用的技术, 详细解释可以参考[数据库如何用 WAL 保证事务一致性? - 知乎专栏](#)。
3. Immutable MemTable。内存中的 MemTable 达到指定的大小后, 将不再接收新数据, 同时会有新的 MemTable 产生, 新数据写入到这个新的 MemTable 里, Immutable MemTable 随后会写入硬盘, 变成一个 SST 文件。
4. **SSTable** 文件。即硬盘上的 SSTable, 文件尾部追加了一块索引, 记录 key->offset, 提高随机读的效率。SST 文件为 Level 0 到 Level N 多层, 每一层包含多个 SST 文件; 单个 SST 文件容量随层次增加成倍增长; Level0 的 SST 文件由 Immutable MemTable 直接 Dump 产生, 其他 Level 的 SST 文件由其上一层的文件和本层文件归并产生。
5. Manifest 文件。Manifest 文件中记录 SST 文件在不同 Level 的分布, 单个 SST 文件的最大最小 key, 以及其他一些 LevelDB 需要的元信息。

6. Current 文件。从上面的介绍可以看出，LevelDB 启动时的首要任务就是找到当前的 Manifest，而 Manifest 可能有多个。Current 文件简单的记录了当前 Manifest 的文件名。



LevelDB 的一些核心逻辑如下，

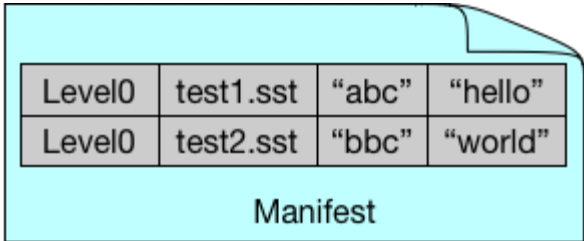
1. 首先 SST 文件尾部的索引要放在内存中，这样读索引就不需要一次磁盘 IO 了
2. 所有读要先查看 **MemTable**，如果没有再查看内存中的索引
3. 所有写操作只能写到 **MemTable**，因为 SST 文件不可修改
4. 定期把 **Immutable MemTable** 写入硬盘，成为 **SSTable** 文件，同时新建一个 **MemTable** 会继续接收新来的写操作

5. 定期对 **SSTable** 文件进行合并
6. 由于硬盘上的 **SSTable** 文件是不可修改的，那怎么更新和删除数据呢？对于更新操作，追加一个新的 key-value 对到文件尾部，由于读 **SSTable** 文件是从前向后读的，所以新数据会最先被读到；对于删除操作，追加“墓碑”值(tombstone)，表示删除该 key，在定期合并 **SSTable** 文件时丢弃这些 key，即可删除这些 key。

## Manifest 文件

---

Manifest 文件记录各个 SSTable 各个文件的管理信息，比如该 SST 文件处于哪个 Level，文件名称叫啥，最小 key 和最大 key 各自是多少，如下图所示，

A diagram of a Manifest file structure. It consists of a light blue rounded rectangle with a folded top-right corner. Inside, there is a table with two rows and four columns. The first row contains 'Level0', 'test1.sst', '“abc”', and '“hello”'. The second row contains 'Level0', 'test2.sst', '“bbc”', and '“world”'. Below the table, the word 'Manifest' is centered.

Level0	test1.sst	“abc”	“hello”
Level0	test2.sst	“bbc”	“world”

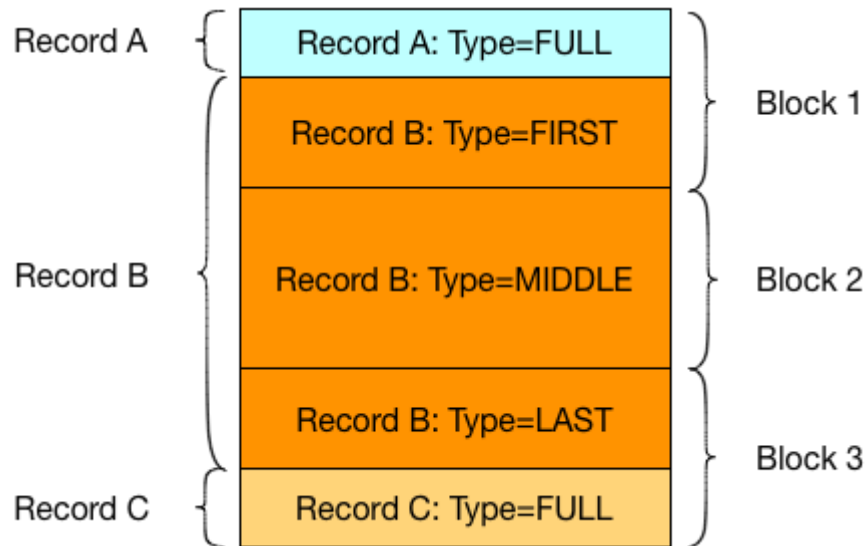
Manifest

## Log 文件

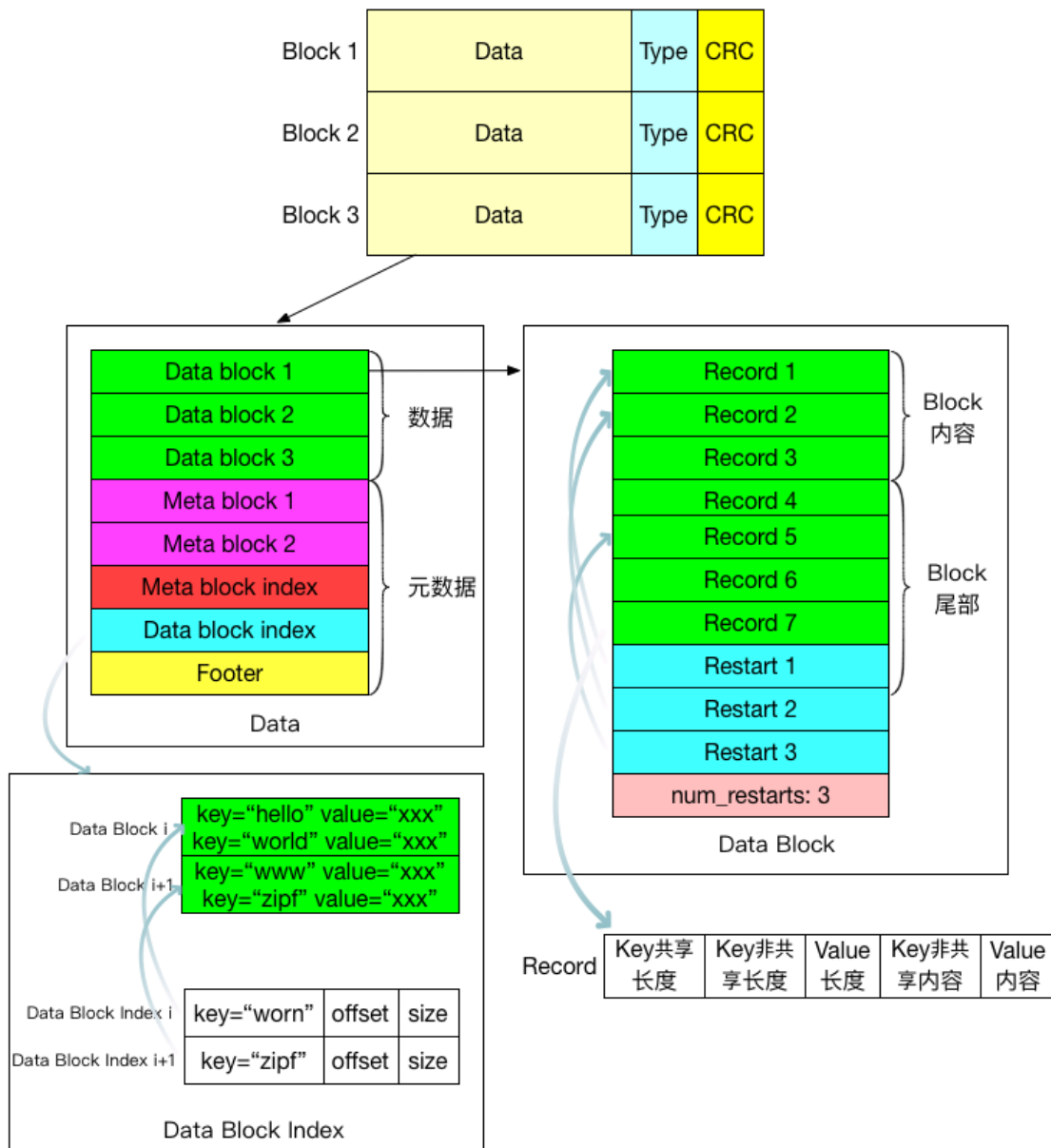
---

Log 文件主要作用是系统发生故障时，能够保证不会丢失数据。因为在数据写入内存中的 MemTable 之前，会先写入 Log 文件，这样即使系统发生故障，MemTable 中的数据没有来得及 Dump 到磁盘，LevelDB 也可以根据 log 文件恢复内存中的 MemTable，不会造成系统丢失数据。这个方式就叫做 WAL(Write Ahead Log)，很多传统数据库例如 MySQL 也使用了 WAL 技术来记录日志。

每个 Log 文件由多个 block 组成，每个 block 大小为 32K，读取和写入以 block 为基本单位。下图所示的 Log 文件包含 3 个 Block，



## SSTable



## MemTable

MemTable 是内存中的数据结构，存储的内容跟硬盘上的 SSTable 一样，只是格式不一样。Immutable MemTable 的内存结构和 Memtable 是完全一样的，区别仅仅在于它是只读的，而 MemTable 则是允许写入和读取的。当 MemTable 写入的数据占用内存到达指定大小，则自动转换为 Immutable Memtable，等待 Dump 到磁盘中，系统会自动生成一个新的 MemTable 供

写操作写入新数据，理解了 MemTable，那么 Immutable MemTable 自然不在话下。

MemTable 里的数据是按照 key 有序的，因此当插入新数据时，需要把这个 key-value 对插入到合适的位置上，以保持 key 有序性。MemTable 底层的核心数据结构是一个跳表(Skip List)。跳表是红黑树的一种替代数据结构，具有更高的写入速度，而且实现起来更加简单，请参考[跳表\(Skip List\)](#)。

前面我们介绍了 LevelDB 的一些内存数据结构和文件，这里开始介绍一些动态操作，例如读取，写入，更新和删除数据，分层合并，错误恢复等操作。

## 添加、更新和删除数据

---

LevelDB 写入新数据时，具体分为两个步骤：

1. 将这个操作顺序追加到 log 文件末尾。尽管这是一个磁盘操作，但是文件的顺序写入效率还是跟高的，所以不会降低写入的速度
2. 如果 log 文件写入成功，那么将这条 key-value 记录插入到内存中 MemTable。

LevelDB 更新一条记录时，并不会本地修改 SST 文件，而是会作为一条新数据写入 MemTable，随后会写入 SST 文件，在 SST 文件合并过程中，新数据会处于文件尾部，而读取操作是从文件尾部倒着开始读的，所以新值一定会最先被读到。



LevelDB 删除一条记录时，也不会修改 SST 文件，而是用一个特殊值(墓碑值，tombstone)作为 value，将这个 key-value 对追加到 SST 文件尾部，在 SST 文件合并过程中，这种值的 key 都会被忽略掉。

核心思想就是把写操作转换为顺序追加，从而提高了写的效率。

## 读取数据

---

读操作使用了如下几个手段进行优化：

- MemTable + SkipList
- Binary Search(通过 manifest 文件)
- 页缓存
- bloom filter
- 周期性分层合并

## 分层合并(Levelled Compaction)

---

# SSTable and Log Structured Storage: LevelDB

If Protocol Buffers is the [lingua franca of individual data record at Google](#), then the Sorted String Table (SSTable) is one of the most popular outputs for storing, processing, and exchanging datasets. As the name itself implies, an **SSTable** is a simple abstraction to efficiently store large numbers of key-value pairs while optimizing for high throughput, sequential read/write workloads.

Unfortunately, the SSTable name itself has also been overloaded by the industry to refer to services that go well beyond just the sorted table, which has only added unnecessary confusion to what is a very simple and a useful data structure on its own. Let's take a closer look under the hood of an SSTable and how LevelDB makes use of it.

## SSTable: Sorted String Table

Imagine we need to process a large workload where the input is in Gigabytes or Terabytes in size. Additionally, we need to run multiple steps on it, which must be performed by different binaries - in other words, imagine we are running a sequence of Map-Reduce jobs! Due to size of input, reading and writing data can dominate the running time. Hence, random reads and writes are not an option, instead we will want to stream the data in and once we're done, flush it back to disk as a streaming operation. This way, we can [amortize the disk I/O costs](#). Nothing revolutionary, moving right along.

A "Sorted String Table" then is exactly what it sounds like, it is a file which contains a set of arbitrary, sorted key-value pairs inside. Duplicate keys are fine, there is no need for "padding" for keys or values, and keys and values are arbitrary blobs. Read in the entire file sequentially and you have a sorted index. Optionally, if the file is very large, we can also prepend, or create a standalone `key:offset` index for fast access. **That's all an SSTable is: very simple, but also a very useful way to exchange large, sorted data segments.**

## SSTable and BigTable: Fast random access?

Once an SSTable is on disk it is effectively immutable because an insert or delete would require a large I/O rewrite of the file. Having said that, for static indexes it is a great solution: read in the index, and you are always one disk seek away, or simply `mmap` the entire file to memory. Random reads are fast and easy.

Random writes are much harder and expensive, that is, unless the entire table is in memory, in which case we're back to simple pointer manipulation. Turns out, this is the very problem that [Google's BigTable](#) set out to solve: fast read/write access for petabyte datasets in size, backed by SSTables underneath. How did they do it?

# SSTables and Log Structured Merge Trees

We want to preserve the fast read access which SSTables give us, but we also want to support fast random writes. Turns out, we already have all the necessary pieces: random writes are fast when the SSTable is in memory (let's call it `MemTable`), and if the table is immutable then an on-disk SSTable is also fast to read from. Now let's introduce the following conventions:

1. On-disk `SSTable` indexes are always loaded into memory
2. All writes go directly to the `MemTable` index
3. Reads check the `MemTable` first and then the `SSTable` indexes
4. Periodically, the `MemTable` is flushed to disk as an `SSTable`
5. Periodically, on-disk SSTables are "collapsed together"

What have we done here? Writes are always done in memory and hence are always fast. Once the `MemTable` reaches a certain size, it is flushed to disk as an immutable `SSTable`. However, we will maintain all the `SSTable` indexes in memory, which means that for any read we can check the `MemTable` first, and then walk the sequence of `SSTable` indexes to find our data. Turns out, we have just reinvented the "[The Log-Structured Merge-Tree](#)" (**LSM Tree**), described by Patrick O'Neil, and this is also the very mechanism behind "[BigTable Tablets](#)".

## LSM & SSTables: Updates, Deletes and Maintenance

This "LSM" architecture provides a number of interesting behaviors: **writes are always fast** regardless of the size of dataset (append-only), and **random reads are either served from memory or require a quick disk seek**. However, what about updates and deletes?

Once the `SSTable` is on disk, it is immutable, hence updates and deletes can't touch the data. Instead, a more recent value is simply stored in `MemTable` in case of update, and a "*tombstone*" record is appended for deletes. Because we check the indexes in sequence, future reads will find the updated or the tombstone record without ever reaching the older values! Finally, having hundreds of on-disk SSTables is also not a great idea, hence **periodically we will run a process to merge the on-disk SSTables, at which time the update and delete records will overwrite and remove the older data**.

## SSTables and LevelDB

Take an `SSTable`, add a `MemTable` and apply a set of processing conventions and what you get is a nice database engine for certain type of workloads. In fact, Google's BigTable, Hadoop's HBase, and Cassandra amongst others are all using a variant or a direct copy of this very architecture.

Simple on the surface, but as usual, implementation details matter a great deal. Thankfully, [Jeff Dean](#) and [Sanjay Ghemawat](#), the original contributors to the `SSTable` and BigTable infrastructure at Google [released LevelDB earlier last year](#), which is more or less an exact replica of the architecture we've described above:

- `SSTable` under the hood, `MemTable` for writes
- Keys and values are arbitrary byte arrays
- Support for Put, Get, Delete operations
- Forward and backward iteration over data
- Built-in [Snappy compression](#)

Designed to be the engine for [IndexDB in WebKit](#) (aka, embedded in your browser), it is [easy to embed](#), [fast](#), and best of all, takes care of all the `SSTable` and `MemTable` flushing, merging and other gnarly details.

## Working with LevelDB: Ruby

LevelDB is a library, not a standalone server or service - although you could easily implement one on top. To get started, grab your favorite language bindings ([ruby](#)), and let's see what we can do:

```
require 'leveldb' # gem install leveldb-ruby
```

```
db = LevelDB::DB.new "/tmp/db"
```

```
db.put "b", "bar"
```

```
db.put "a", "foo"
```

```
db.put "c", "baz"
```

```
puts db.get "a" # => foo
```

```
db.each do |k,v|
```

```
  p [k,v] # => ["a", "foo"], ["b", "bar"], ["c", "baz"]
```

```
end
```

```
db.to_a # => [ ["a", "foo"], ["b", "bar"], ["c", "baz"] ]
```

We can store keys, retrieve them, and perform a range scan all with a few lines of code. The mechanics of maintaining the MemTables, merging the SSTables, and the rest is taken care for us by LevelDB - nice and simple.

## LevelDB in WebKit and Beyond

SSTable is a very simple and useful data structure - a great bulk input/output format. However, what makes the SSTable fast (sorted and immutable) is also what exposes its very limitations. To address this, we've introduced the idea of a MemTable, and a set of "log structured" processing conventions for managing the many SSTables.

All simple rules, but as always, implementation details matter, which is why LevelDB is such a nice addition to the open-source database engine stack. Chances are, **you will soon find LevelDB embedded in your browser, on your phone, and in many [other places](#)**. Check out the [LevelDB source](#), scan the [docs](#), and take it for a spin.

## Log Structured Merge Trees(LSM) 原理

十年前，谷歌发表了 “BigTable” 的论文，论文中很多很酷的方面之一就是它所使用的文件组织方式，这个方法更一般的名字叫 Log Structured-Merge Tree。

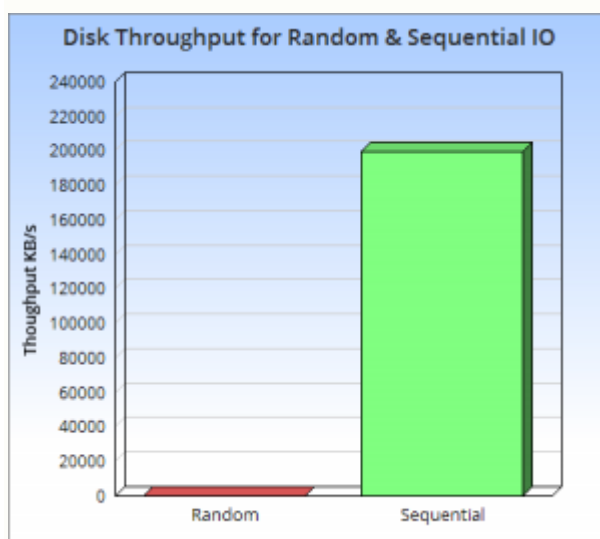
LSM 是当前被用在许多产品的文件结构策略：HBase, Cassandra, LevelDB, SQLite,甚至在 mangodb3.0 中也带了一个可选的 LSM 引擎（Wired Tiger 实现的）。

LSM 有趣的地方是他抛弃了大多数数据库所使用的传统文件组织方法，实际上，当你第一次看它是违反直觉的。

## 背景知识

简单的说，LSM 被设计来提供比传统的 B+树或者 ISAM 更好的写操作吞吐量，通过消去随机的本地更新操作来达到这个目标。

那么为什么这是一个好的方法呢？这个问题的本质还是磁盘随机操作慢，顺序读写快的老问题。这二种操作存在巨大的差距，无论是磁盘还是 SSD。



上图很好的说明了这一点，他们展现了一些反直觉的事实，顺序读写磁盘（不管是 SATA 还是 SSD）快于随机读写主存，而且快至少三个数量级。这说明我们要避免随机读写，最好设计成顺序读写。

所以，让我们想想，如果我们对写操作的吞吐量敏感，我们最好怎么做？一个好的办法是简单的将数据添加到文件。这个策略经常被使用在日志或者堆文件，因为他们是完全顺序的，所以可以提供非常好的写操作性能，大约等于磁盘的理论速度，也就是 200~300 MB/s。

因为简单和高效，基于日志的策略在大数据之间越来越流行，同时他们也有一些缺点，从日志文件中读一些数据将会比写操作需要更多的时间，需要倒序扫描，直接找到所需的内容。

这说明日志仅仅适用于一些简单的场景：1. 数据是被整体访问，像大部分数据库的 WAL(write-ahead log) 2. 知道明确的 offset，比如在 Kafka 中。

所以，我们需要更多的日志来为更复杂的读场景（比如按 key 或者 range）提供高效的性能，这儿有 4 个方法可以完成这个，它们分别是：

1. 二分查找：将文件数据有序保存，使用二分查找来完成特定 key 的查找。
2. 哈希：用哈希将数据分割为不同的 bucket
3. B+树：使用 B+树 或者 ISAM 等方法，可以减少外部文件的读取
4. 外部文件：将数据保存为日志，并创建一个 hash 或者查找树映射相应的文件。

所有的方法都可以有效的提高了读操作的性能（最少提供了  $O(\log(n))$ ），但是，却丢失了日志文件超好的写性能。上面这些方法，都强加了总体的结构信息在数据上，数据被按照特定的方式放置，所以可以很快的找到特定的数据，但是却对写操作不友善，让写操作性能下降。

更糟糕的是，当我们需要更新 hash 或者 B+树的结构时，需要同时更新文件系统中特定的部分，这就是上面说的比较慢的随机读写操作。**这种随机的操作要尽量减少。**

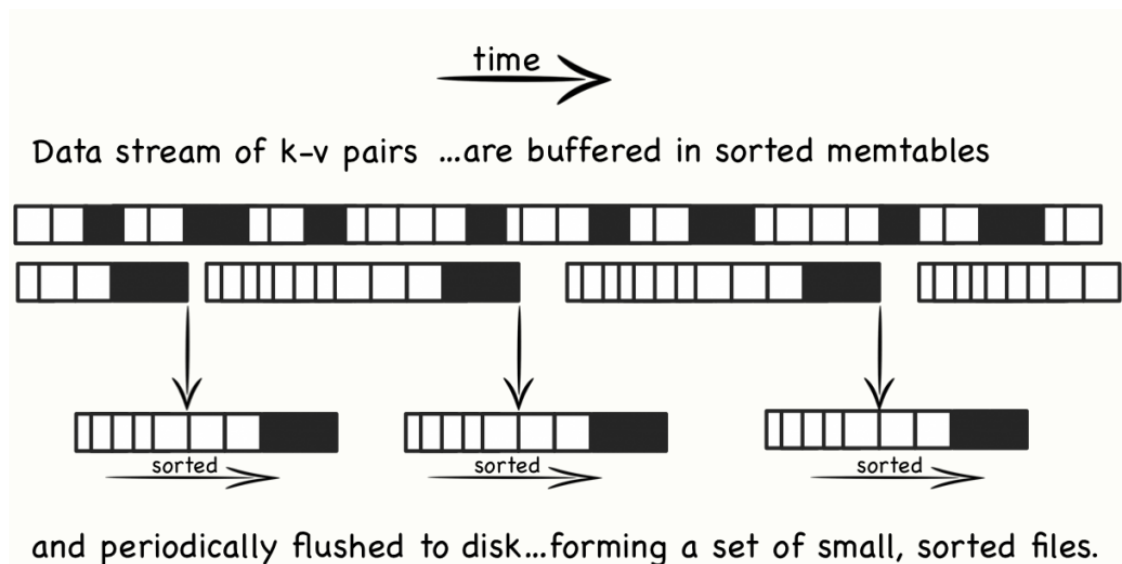
所以这就是 LSM 被发明的原理， LSM 使用一种不同于上述四种的方法，保持了日志文件写性能，以及微小的读操作性能损失。本质上就是让所有的操作顺序化，而不是像散弹枪一样随机读写。

很多树结构可以不用 update-in-place，最流行就是 [append-only Btree](#)，也称为 Copy-On-Write Tree。他们通过顺序的在文件末尾重复写对结构来实现写操作，之前的树结构的相关部分，包括最顶层结点都会变成孤结点。尽管通过这种方法避免了本地更新，但是因为每个写操作都要重写树结构，放大了写操作，降低了写性能。

## The Base LSM Algorithm

从概念上说，最基本的 LSM 是很简单的。将之前使用一个大的查找结构（造成随机读写，影响写性能），变换为将写操作顺序的保存到一些相似的有序文件（也就是 sstable)中。所以每个文件包含短时间内的一些改动。因为文件是有序的，所以之后查找也会很快。文件是不可修改的，他们永远不会被更新，新的更新操作只会写到新的文件中。读操作检查很多的文件。通过周期性的合并这些文件来减少文件个数。





让我们更具体的看看，当一些更新操作到达时，他们会被写到内存缓存（也就是 memtable）中，memtable 使用树结构来保持 key 的有序，在大部分的实现中，memtable 会通过写 WAL 的方式备份到磁盘，用来恢复数据，防止数据丢失。当 memtable 数据达到一定规模时会被刷新到磁盘上的一个新文件，重要的是系统只做了顺序磁盘读写，因为没有文件被编辑，新的内容或者修改只用简单的生成新的文件。

所以越多的数据存储到系统中，就会有越多的不可修改的，顺序的 sstable 文件被创建，它们代表了小的，按时间顺序的修改。

因为比较旧的文件不会被更新，重复的纪录只会通过创建新的纪录来覆盖，这也就产生了一些冗余的数据。

所以系统会周期的执行合并操作（compaction）。合并操作选择一些文件，并把他们合并到一起，移除重复的更新或者删除纪录，同时也会删除上述的冗

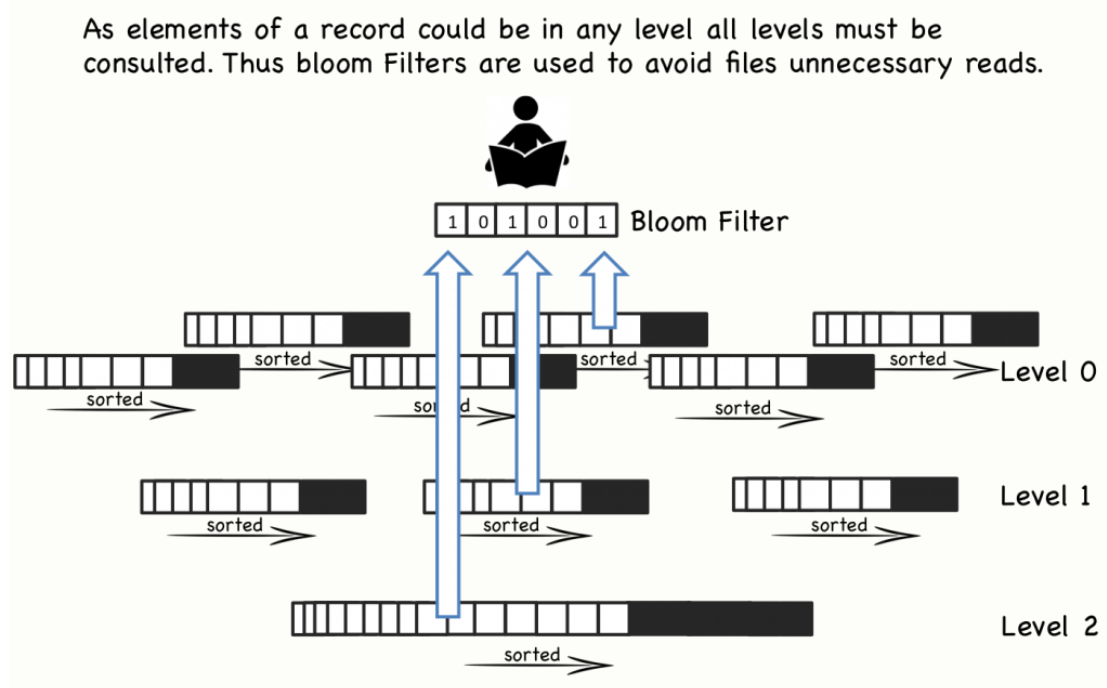
余。更重要的是，通过减少文件个数的增长，保证读操作的性能。因为 sstable 文件都是有序结构的，所以合并操作也是非常高效的。

当一个读操作请求时，系统首先检查内存数据(memtable)，如果没有找到这个 key，就会逆序的一个一个检查 sstable 文件，直到 key 被找到。因为每个 sstable 都是有序的，所以查找比较高效( $O(\log N)$ )，但是读操作会变的越来越慢随着 sstable 的个数增加，因为每一个 sstable 都要被检查。（ $O(K \log N)$ , K 为 sstable 个数，N 为 sstable 平均大小）。

所以，读操作比其它本地更新的结构慢，幸运的是，有一些技巧可以提高性能。最基本的方法就是页缓存（也就是 leveldb 的 TableCache，将 sstable 按照 LRU 缓存在内存中）在内存中，减少二分查找的消耗。LevelDB 和 BigTable 是将 block-index 保存在文件尾部，这样查找就只要一次 IO 操作，如果 block-index 在内存中。一些其它的系统则实现了更复杂的索引方法。

即使有每个文件的索引，随着文件个数增多，读操作仍然很慢。通过周期的合并文件，来保持文件的个数，因些读操作的性能在可接收的范围内。即便有了合并操作，读操作仍然会访问大量的文件，大部分的实现通过布隆过滤器来避免大量的读文件操作，布隆过滤器是一种高效的方法来判断一个 sstable 中是否包含一个特定的 key。（如果 bloom 说一个 key 不存在，就一定不存在，而当 bloom 说一个文件存在是，可能是不存在的，只是通过概率来保证）

所有的写操作都被分批处理，只写到顺序块上。另外，合并操作的周期操作会对 IO 有影响，读操作有可能会访问大量的文件（散乱的读）。这简化了算法工作的方法，我们交换了读和写的随机 IO。这种折衷很有意义，我们可以通过软件实现的技巧像布隆过滤器或者硬件（大文件 cache）来优化读性能。

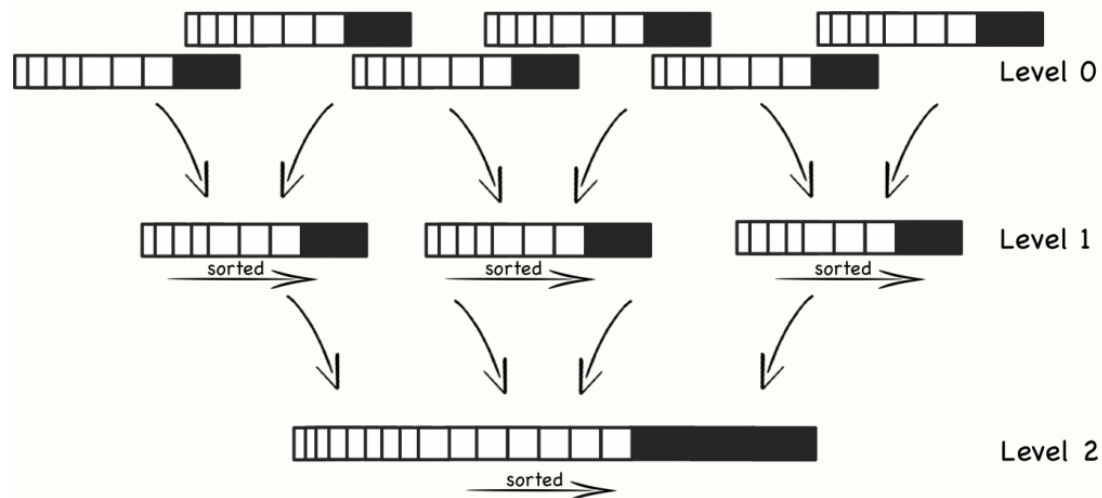


## Basic Compaction

为了保持 LSM 的读操作相对较快，维护并减少 sstable 文件的个数是很重要的，所以让我们更深入的看一下合并操作。这个过程有点儿像一般垃圾回收算法。

当一定数量的 sstable 文件被创建，例如有 5 个 sstable，每一个有 10 行，他们被合并为一个 50 行的文件（或者更少的行数）。这个过程一直持续着，当更多的有 10 行的 sstable 文件被创建，当产生 5 个文件时，它们就被合并到 50 行的文件。最终会有 5 个 50 行的文件，这时会将这 5 个 50 行的文件合并

成一个 250 行的文件。这个过程不停的创建更大的文件。像下图：



Compaction continues creating fewer, larger and larger files

上述的方案有一个问题，就是大量的文件被创建，在最坏的情况下，所有的文件都要搜索。

## Levelled Compaction

更新的实现，像 LevelDB 和 Cassandra 这个问题的方法是：实现了一个分层的，而不是根据文件大小来执行合并操作。这个方法可以减少在最坏情况下需要检索的文件个数，同时也减少了一次合并操作的影响。

按层合并的策略相对于上述的按文件大小合并的策略有二个关键的不同：

1. 每一层可以维护指定的文件个数，同时保证不让 key 重叠。也就是说把 key 分区到不同的文件。因此在一层查找一个 key，只用查找一个文件。  
第一层是特殊情况，不满足上述条件，key 可以分布在多个文件中。

2. 每次，文件只会被合并到上一层的一个文件。当一层的文件数满足特定个数时，一个文件会被选出并合并到上一层。这明显不同与另一种合并方式：一些相近大小的文件被合并为一个大文件。

这些改变表明按层合并的策略减小了合并操作的影响，同时减少了空间需求。

除此之外，它也有更好的读性能。但是对于大多数场景，总体的 IO 次数变的更多，一些更简单的写场景不适用。

## 总结

所以，LSM 是日志和传统的单文件索引 (B+ tree, Hash Index) 的中立，他提供一个机制来管理更小的独立的索引文件(sstable)。

通过管理一组索引文件而不是单一的索引文件，LSM 将 B+ 树等结构昂贵的随机 IO 变的更快，而代价就是读操作要处理大量的索引文件(sstable)而不是一个，另外还是一些 IO 被合并操作消耗。

如果还有不明白的，这还有一些其它的好的介绍。 [here](#) and [here](#)

## 关于 LSM 的一些思考

为什么 LSM 会比传统单个树结构有更好的性能？

我们看到 LSM 有更好的写性能，同时 LSM 还有其它一些好处。 sstable 文件是不可修改的，这让对他们的锁操作非常简单。一般来说，唯一的竞争资源就是 memtable，相对来说需要相对复杂的锁机制来管理在不同的级别。

所以最后的问题很可能是以写为导向的压力预期如何。如果你对 LSM 带来的写性能的提高很敏感，这将会很重要。大型互联网企业似乎很看中这个问题。

Yahoo 提出因为事件日志的增加和手机数据的增加，工作场景为从 read-heavy 到 read-write。。许多传统数据库产品似乎更青睐读优化文件结构。

因为可用的内存的增加，通过操作系统提供的大文件缓存，读操作自然会被优化。写性能（内存不可提高）因此变成了主要的关注点，所以采取其它的方法，硬件提升为读性能做的更多，相对于写来说。因此选择一个写优化的文件结构很有意义。

理所当然的，LSM 的实现，像 LevelDB 和 Cassandra 提供了更好的写性能，相对于单树结构的策略。

## Beyond Levelled LSM

这有更多的工作在 LSM 上， Yahoo 开发了一个系统叫作 Pnuts， 组合了 LSM 与 B 树，提供了更好的性能。我没有看到这个算法的开放的实现。 IBM 和 Google 也实现了这个算法。也有相关的策略通过相似的属性，但是是通过维护一个拱形的结构。如 Fractal Trees， Stratified Trees.

这当然是一个选择，数据库利用大量的配置，越来越多的数据库为不同的工作场景提供插件式引擎。 Parquet 是一个流行的 HDFS 的替代，在很多相对的文面做的好很（通过一个列格式提高性能）。MySQL 有一个存储抽象，支持大量的存储引擎的插件，例如 Toku (使用 fractal tree based index)。

Mongo3.0 则包含了支持 B+和 LSM 的 Wired Tiger 引擎。许多关系数据库可以配置索引结构，使用不同的文件格式。

考虑被使用的硬件，昂贵的 SSD，像 FusionIO 有更好的随机写性能，这适合本地更新的策略方法。更便宜的 SSD 和机械盘则更适合 LSM。

## 延伸阅读

- There is a nice introductory post [here](#).
- The LSM description in this [paper](#) is great and it also discusses interesting extensions.
- These three posts provide a holistic coverage of the algorithm: [here](#), [here](#) and [here](#).
- The original Log Structured Merge Tree paper [here](#). It is a little hard to follow in my opinion.
- The Big Table paper [here](#) is excellent.
- [LSM vs Fractal Trees](#) on High Scalability.
- Recent work on [Diff-Index](#) which builds on the LSM concept.

- [Jay](#) on SSDs and the benefits of LSM

# 存储引擎技术架构与内幕

## (leveldb-1)

今年年初做毕设那段时间, 曾涉及一些关于数据存储方面的技术, 着手实现时也发现了不少乐趣, 这里就梳理一下思路, 做个总结, 也尽量让人读完之后有个全面的了解, 感谢工业界和学术界产出的详实资料文献和笔录经验, 给我学习过程提供了莫大便利! 其中过于底层的技术的原理我可能没法做到了然于胸, 当然我会持续深入学习那些不甚了解/拿不准的东西, 不断完善知识技能树.

## 几个术语

来区分这两个专有名词:

- 数据库
- 存储引擎

数据库往往是一个比较丰富完整的系统, 目的是大而全, 提供了丰富的查询和一系列复杂的数据操作逻辑, 或者是在网络层面, 水平扩展等支持.

然而一个存储引擎目标则是小而精, 因为它是纯粹专注于**读/写/存储**, 一般来说, 直接使用存储引擎的是像数据这样的上层存储系统.

## 那么来思考几个问题

如果是你, 你要如何:

- 设计一个 k-v 存储引擎?
- 支持随机读写?
- 高效的数据操作?
- 持久化数据?
- 错误容忍?



ok, 我知道你也不是十分了解~ 我也非专业, 那么我们就看看人家是怎么考虑这些问题的~~.

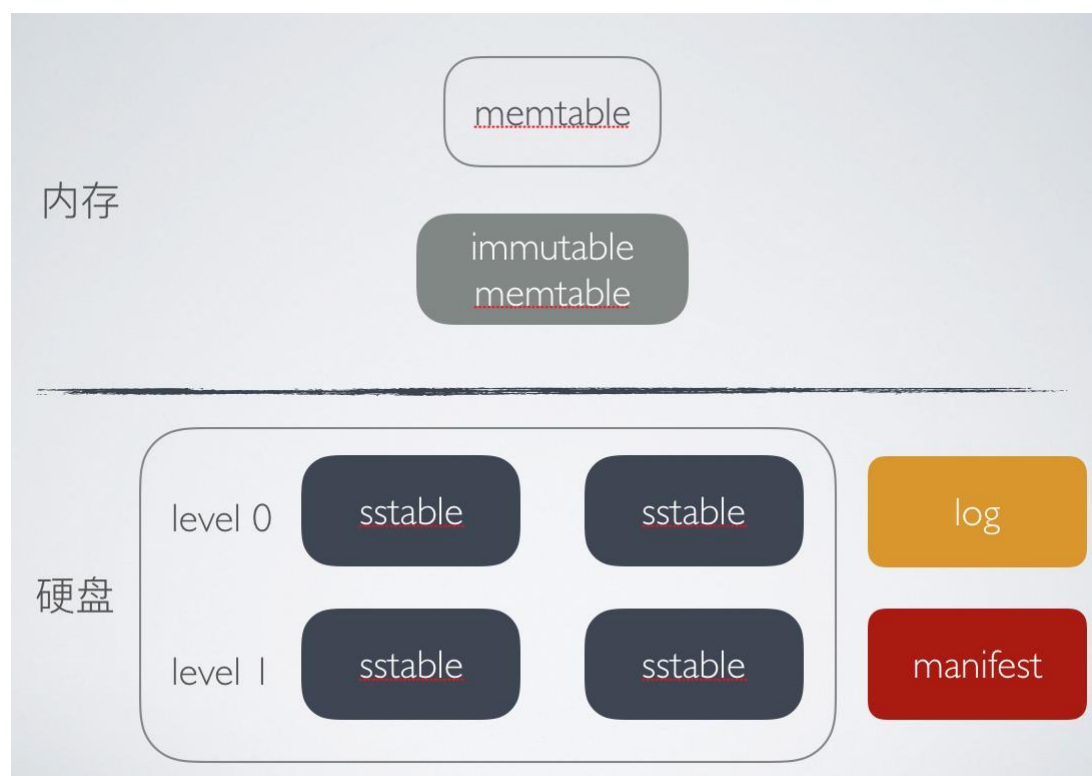
## 从 **leveldb** 讲起

为啥说 **leveldb**? 因为我觉得它是个不错的学习例子, 代码和官方文档都详细说明了一个简单的 k-v 引擎在设计上应该考虑哪些, 注意什么.

没听过 **leveldb** 也不要紧, 几句话介绍一下: **google** 亲儿子, **BigTable** 核心技术, **Chrome IndexedDB** 的内核. 具备极高的写入效率, 同时有不俗的读性能.

## 结构规划

了解一个系统的普遍方法就是先了解其大体轮廓和工作方法. 为了了解 **leveldb** 是如何工作的, 我们首先来看它的结构规划是怎样的.



根据此图先初步说一下它的读写流程:

- **write:** 写入 **log** 文件一条操作(数据更新)记录, 再往 **memtable** 里写入 k-v 对.
- **read:** 读 **memtable** (没找到)→ 读 **immutable memtable** (没找到)→ 读 **sstable** (没找到)→ 空

没错, 每次操作就这么简单的流程. 接下来就来谈谈这样设计的优劣原由以及各个组件所起的作用, 从而讲一下设计一个存储引擎的思路.

## 内幕

虽说一次读写就那么几步, 但是他们的背后究竟发生了什么? 换句话说, 既然 `leveldb` 作为一个具备高性能写操作的存储引擎, 是什么提供的性能优势? 来看下操作内幕.

## memtable

`memtable` 是个内存数据结构, 也是读写操作的入口.

作为一个存储引擎, 读无非就是读若干个 `key`, 然后返回对应的 `value`; 而写就是存下这些 `key-value` 对.

所以 `memtable` 中存的每条数据也都是一个键值对.

关键点是 `memtable` 中的数据是按 `key` 的字母表顺序排序的.

然而对一个具备随机读能力的排序结构执行插入操作往往是有开销的, 通常写瓶颈也都是集中在这里. 但是也有很多数据结构提供了加速随机写, 比如链表, AVL 树, B 树, `skiplist`. `memtable` 的核心结构就是用了跳表.

结构简单, 概率上近似  $O(\log N)$  读写复杂度的跳表在很多存储系统里得到应用, 比如 `elasticsearch` 的高速搜索就是基于跳表, 位图以及倒排索引实现的. `memtable` 采用这种结构初步实现快速读写.

接下来, 既然是存储引擎, 那么肯定能持久化数据, 而 `leveldb` 核心技术就是围绕持久化过程而构建的.

## Log Structed Merge Tree

我们在谈论 `leveldb` 的存储方案之前, 有必要先介绍一下另外一种数据结构: `lsm tree`.

`lsm tree` 是针对写入速度瓶颈问题而提出的. `mysql` 这种数据库的存储引擎使用了 B+ 树来持久化数据, B+ 树是一个索引树, 可以说是同时考虑了读写均衡, 其结构上对树高进行了优化, 搜索耗时相比 AVL 树降下来. 然而问题依然是前面我们谈到的 "对一个随机读优化的排序结构执行随机写是有很大开销的", 所以对那些需要高频写操作的系统来讲, B+ 树作为存储结构可能并不合适.

这时 `lsm tree` 可能是个更好的选择, 它是一种类似日志的数据结构, 将随机写变为顺序写, 核心思想是:

- 对变更进行批量 & 延时处理
- 通过归并排序将更新迁移到硬盘上

自从上世纪某年代一篇关于日志结构文件系统的论文发表之后, 有人基于这个想法提出了解决写瓶颈的问题: 使用顺序写(追加)替代随机写. 为什么要变为顺序写呢? 因为顺序写不需要多次寻址, 速度能达到硬盘理论传输速度, 而随机写则受限于硬盘寻址速度.

这种思想跟当今的 `immutable` 结构, 函数式中的不变量, 比特币区块链如出一辙, 因为跟日志很像, 所以称之为 "日志结构合并树", 原论文花了 30 来页从数据结构, 操作策略, 低层次磁盘优化等等方面阐述了 `lsm tree`. (我读 5 页就放弃了...)

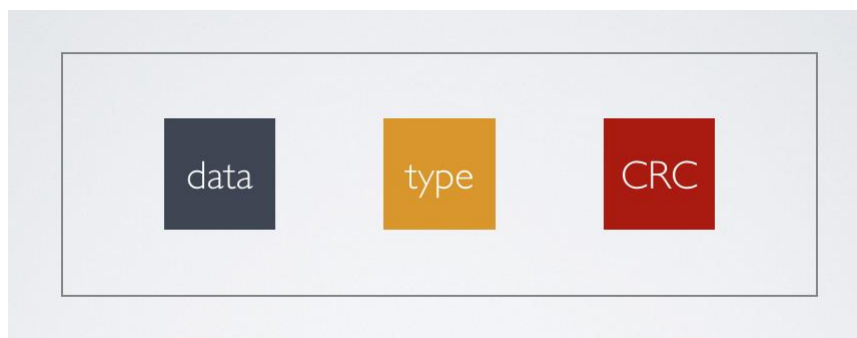
`leveldb` 在持久化上借鉴了 `lsm tree` 的设计. 具体实现就是 `memtable + sstable`.

## sstable

`sstable` 全名 `sort-string table`, `bigtable` 使用的存储技术. 顾名思义, `sstable` 中的数据都是有序的.

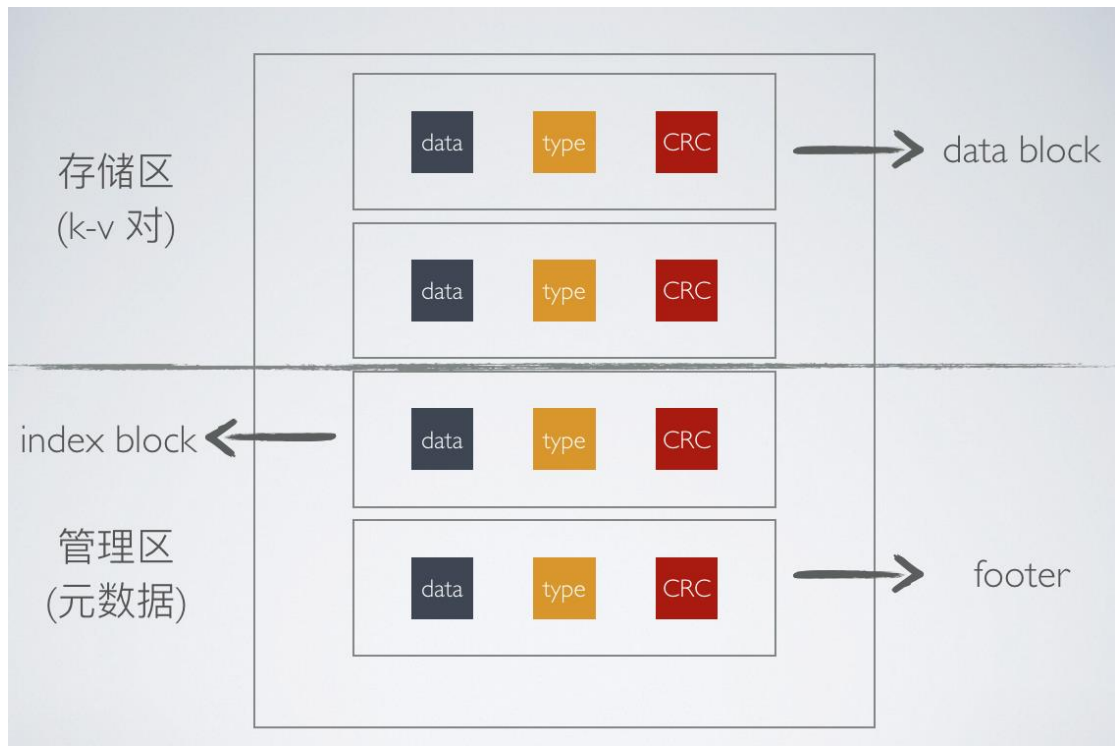
除了日志之外, `leveldb` 的数据统统存储在 `sstable` 中. 对于 `sstable`, 我们先来看下其布局.

每个 `sstable` 内部按 `block` 划分. 物理布局如图.



`type` 用于表示 `data` 域采用何种压缩算法. (因为 `leveldb` 使用了 `snappy` 压缩算法)  
`crc` 循环冗余编码用于数据的检错. 这个 IP 数据报头也有用到.

而在逻辑视图上, 将整个 `sstable` 看成是由 存储区 和 管理区 组成:



下面分别介绍不同种类的 block 作用.

### index block



index block 中的每条记录会对某个 data block 建立索引.

key 保存大于等于前一个 data block 中最大 key 并且小于下一个 data block 中最小 key 的一个值.

offset 指明被索引的 data block 的起始位置在 sstable 中的偏移量.  
size 记录 data block 大小.

## footer

footer 是整个 sstable 的末尾 block, 记录了 indexblock 的起始偏移量和大小, 很容易看出其存在意义就是为了方便读取 index block.

## data block

对于重中之重 —— 数据块, 其 data 域存储的就是所有的 k-v.  
更具体来说, 是 data 域的 record 段存储的. record 里的 k-v 对全部按 key 有序排列.



**record** 字段也并不是简单的存储了 k-v, 而是将 key 分解为前缀和后缀存储.  
为何如此复杂设计? 原因很简单, 压缩存储, 尽量减少重复数据占用的空间, 思路类似 trie 树 (公共前缀树).

而上面的 **restart** 字段表示从某条记录开始重新存储新的前缀, 并且该条记录完整存储一个 key. (切换公共前缀)

## sstable 跟 lsm tree 啥关系?

之前提到了 leveldb 借鉴 lsm-tree 实现了顺序写结构, 然后紧接着介绍 sstable, 那么 sstable 如何实现的 lsm-tree 这种思想? 接下来就不得不提 leveldb 中的操作策略了.

## **sstable(s) 是复数**

看过第一幅图你应该发现了, **sstable** 不仅仅有一个, 而是一批.

leveldb 将 **sstable** 划分为了不同层次(level). level  $i+1$  层的 **sstables** 由 level  $i$  层的 **sstables** merge 得来. 而最上层称为 level 0.

那么第 0 层的 **sstables** 是怎么来的呢?

## **memtable + sstable = lsm tree**

- **memtable** → **immutable memtable** (内存中)
- **immutable memtable** → (level 0) **sstable** (内存 → 外存)

还记得最开始提到的 **memtable**? 那个读写入口数据结构. leveldb 给了他一个阈值, 但凡 **memtable** 里的数据大小达到了阈值, 后台任务就将 **memtable** 标记为只读, 也就是变成了 **immutable memtable**.

然后创建一个新的 **memtable** 用于接下来的读写.

**immutable memtable** 经过一段时间会被迁移到硬盘上, 成为 **sstable**, 这一过程称之为 **compaction**.

因为 **memtable** 的结构是有序的, 因此 **sstable** 也是有序存储的.

## **compaction**

**compaction** 是执行 lsm-tree 中 merge 的过程.

对于不同应用情况, 分为 **minor compaction** 和 **major compaction**.

### **minor compaction**

**minor compaction** 用于内存到外存的迁移过程. 就是简单的遍历跳表, 依次写入新的 **sstable record**, 最后建立 **index block** 并完善一些其他的重要元信息. 这也就是从 **immutable memtable** 到 level 0 **sstable** 的迁移.

### **major compaction**

**major compaction** 用于 level 之间的迁移.

当某个 level 的 **sstables** 数量超过一个给定的阈值, 就会触发 **major compaction**.

这里有两点差异:

- 对 level  $> 0$  的 **sstables**, 选择其中一个 **sstable** 与 下一层 **sstables** 做合并.

- 对 level = 0 的 sstables, 在选择一个 sstable 后, 还需要找出所有与这个 sstable 有 key 范围重叠的 sstables, 最后统统与 level 1 的 sstables 做合并.

不知之前是否注意到, level 0 的 sstables 可能有键范围的重合. (因为 level 0 的 sstable 是直接由 memtable 变过来的, 而不同的 memtable 之间并没有约束 key 必须独立.)

## merge 策略

compaction 过程中有提到选择 sstable, 如何选择? 这就看 leveldb 的 merge 策略了:

- level i 按顺序选择.
- level i+1 选择所有与 level i 所选 sstable(s) 有 key 范围重叠的 sstables.
- 将这些 sstables 做 K 路归并排序. 对于相同的 key, 只保留最新的(上层 sstable, 同层中新的 sstable).
- 清除参与此次 merge 的所有 sstables, 保留新的 sstable.

有了前面提到的 compaction 第二个约束和 merge 策略, 间接解释了 level 0 和其他 level 不同的原因: 在 level 0 的 compaction 后, level 1 产生的 sstable 是没有 key 范围重叠的, 因此向高层 level 的 compaction 也不会有同一个 level 下 key 范围重叠的 sstable 产生.

到这里, leveldb 如何实现的 lsm-tree 就比较明朗了: 内存结构中写数据, 每次写只考虑当前 memtable, 不涉及其他结构的更新, 内存数据顺序迁移到外存中, 形成一个日志结构, 这就是 leveldb 顺序写的思路.

### 关于 k 路归并

就是我们上数据结构课学的那个外部排序. 这里有个简单的思路: 用最小堆实现.

- k 个有序的 sstable 取最小 key, 组成最小堆, 取定点 key, 用其所在 sstable 的下一个 key 填充(当一个 sstable 所有的 key 都被取完, 则从堆的末尾取 key), 然后重新调整使其符合堆性质, 如此循环往复.
- 而所有取出的最小堆顶点依次组成一个列表, 就形成新的 sstable 了.

## 如何做到高效的随机写?

到此为止, leveldb 的写入策略就介绍完了, 这里做个总结:

1. 内存 + 跳表
2. 外存中追加, 使用顺序写

## 读取效率如何?

说完了写, 那么并不具备读优势的 `leveldb` 读取是怎样设计的呢?

毕竟凡事很难达到两全, 顺序写的这种结构就注定了丧失读效率(需要多次遍历寻址). 然而 `leveldb` 仍然费尽心思在代码层面而非算法层面做了优化.

查询过程中优化的顺序如下:

- 内存 + skiplist
- sstables B-search (通过 manifest 文件)
- 页缓存
- bloom filter

(周期性 compaction 做辅助)

在查询 level 0 sstables 时, 因为不同 sstables 的 key 范围可能有重叠, 所以只能把但凡包含 key 的 sstables 找出来, 排序取最新的 (manifest 文件记录了当前所有 sstable 的信息, 包括文件名, 所处 level, key 的范围等等). 因为 sstable 是有序的, 所以在 level > 0 寻找 sstable 这一过程可以应用二分搜索完成. 找到 sstable 后, 读出其 index block, 加载到内存作为 table cache, 再根据索引去查具体的 data block. 而 data block 也包含了一组 k-v, 为了提高定位 key 的效率, `leveldb` 又引入了布隆过滤器这一数据结构.

## bloom filter

这里引用一下数学之美中的介绍: 它是一个判断元素是否在集合中的高效做法, 常用于垃圾邮件过滤.

优势:

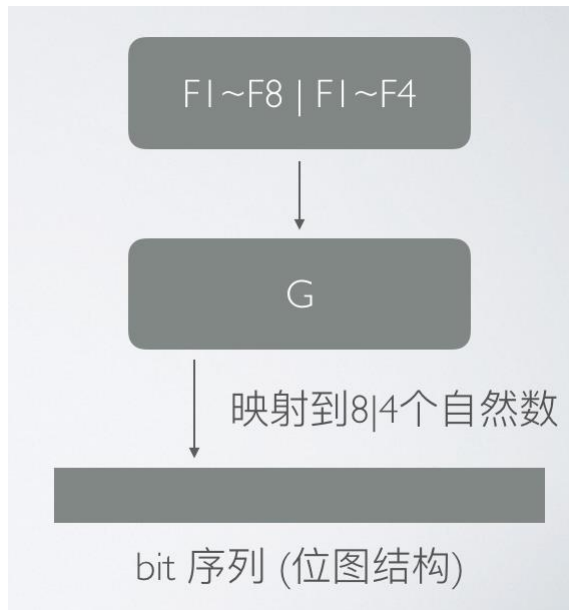
- + 快速
- + 节省空间(为散列表的  $1/8$  或者  $1/4$ )
- + 不存储数据(安全性)

劣势:

- + 存在误判率 (但是微乎其微, 也可以用白名单方法解决)

原理简单说下:





$f_1 \sim f_8$  和  $G$  为不同的随机数产生器, 把一个元素通过  $f$  计算分别得到  $4/8$  个值, 然后分别通过  $G$  映射到  $4/8$  个自然数, 这些数用位图的方法存储在一个 **bit** 序列中(置 1 的索引), 因此判断时只要检查这几个位置的值是否全为一即可。

有了 bloom filter, 在 block 的判定中就可以快速跳过不含有目标 key 的 block 了, 在一定程度上减少了磁盘的随机访问次数. leveldb 中 bloom filter 是针对每个 sstable block 而建立的.

## 如何节省存储空间?

做到了读写优化后, 节省空间也是要考虑的, 所幸基于这种设计架构, 已经做到了把空间压缩, 具体:

- 公共前缀 key
- sstables compaction
- snappy 压缩

## 综上

我们可以看到 leveldb 在存储引擎设计/实现过程中对各个层面做出的优化, 包括:

- 架构层面
- 算法层面
- 代码层面

## 容错 & 数据恢复

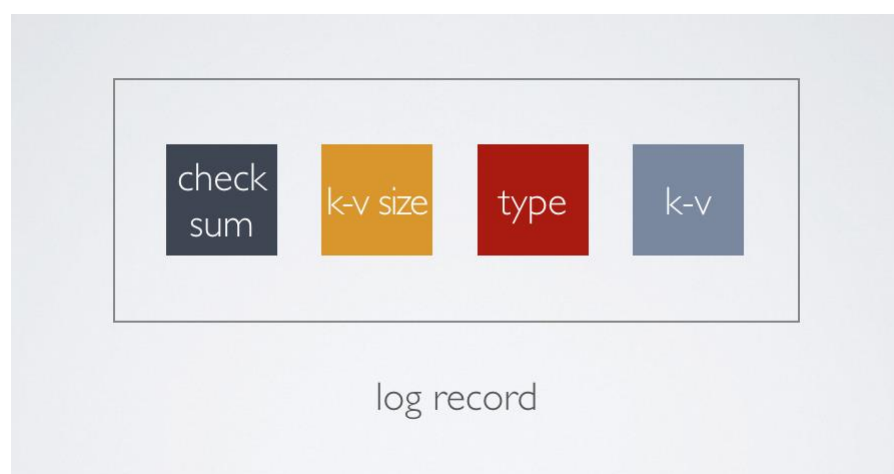
对于一个存储引擎来说, 健壮性也是必不可少的, 进程意外退出或者机器故障很容易导致数据断层(丢失), 对此, `leveldb` 在每次写操作前都会在 `log` 文件里追加一条日志.

`log` 跟 `sstable` 很像, 因为 `log` 就是日志.

唯一的区别是 `log` 忠实按照操作顺序记录, 因此不考虑 `key` 是否有序.

## 日志文件

`log` 文件的单位是固定大小的 `block`. 每个 `block` 中包含了一系列记录.



其中 `type` 有以下四种植:

- `full`: 该记录完整存储在一个 `block` 里
- `first`: 该记录的开头部分
- `middle`: 该记录的中间部分
- `last`: 该记录的最后部分

`leveldb` 会根据类型拼接出逻辑记录, 供后续处理.

## what's more

除了前面讲的技术, `leveldb` 还有更多可以深入研究的点, 比如快照(snapshot)和版本控制(数据恢复)技术, 这里暂时没有研究, 就不扩展了.

## 引用

- log 文件格式: [https://raw.githubusercontent.com/google/leveldb/master/doc/log\\_format.txt](https://raw.githubusercontent.com/google/leveldb/master/doc/log_format.txt)
- sstable 文件格式: [https://raw.githubusercontent.com/google/leveldb/master/doc/table\\_format.txt](https://raw.githubusercontent.com/google/leveldb/master/doc/table_format.txt)
- 设计要点: <https://rawgit.com/google/leveldb/master/doc/impl.html>
- 多路归并: <https://zh.wikipedia.org/wiki/%E5%A4%96%E6%8E%92%E5%BA%8F>
- 跳表: <https://zh.wikipedia.org/wiki/%E8%B7%B3%E8%B7%83%E5%88%97%E8%A1%A8>
- leveldb 实现原理: <http://www.cnblogs.com/haippy/archive/2011/12/04/2276064.html>

然后这里有个自己设计的存储(读优化), 当然其架构和性能跟 leveldb 是不能比的, 不过是在研究 leveldb 之前写的, 之后发现好多地方的设计竟然有异曲同工之妙, 我也偷偷的高兴一阵子~~

- Hive-fs (<https://github.com/abbshr/hive-fs>)
- Leviathan (<https://github.com/abbshr/Leviathan/>)

## To Be Continued ...

我感觉上文中还有一些概念和技术的说明比较模糊, 并且也只介绍了 leveldb. 不过存储技术也是在下今后的研究方向之一, 我要让这篇日志会变得更加严谨, 内容更充实.