数据存储与检索

> 👣 数据库是一个老生常谈的话题，我们通常会直接讨论现有的一些数据库的底层设计，和一些实现细节。这里我们更多的是而从宏观的角度上看，**数据库中的数据是怎么存储的，为什么这么设计？它的性能如何？我们怎么优化？**

驱动数据库的数据结构

世界上最简单的数据库可以用两个 Bash 函数实现：

Bash

```bash
#!/bin/bash
db_set () {
        echo "$1,$2" >> database
}
db_get () {
        grep "^$1," database | sed -e "s/^$1,//" | tail -n 1
}
```

执行 `db_set key value` ，会将 **键（key）和值（value）** 存储在数据库中。键和值可以是你喜欢的任何东西，例如，值可以是 JSON 文档。然后调用 `db_get key` ，查找与该键关联的最新值并将其返回。

麻雀虽小，五脏俱全：

Bash

```bash
$ db_set 123456 '{"name":"London","attractions":["Big Ben","London Eye"]}' $

$ db_set 42 '{"name":"San Francisco","attractions":["Golden Gate Bridge"]}'

$ db_get 42
{"name":"San Francisco","attractions":["Golden Gate Bridge"]}
```

底层的存储格式非常简单：一个文本文件，每行包含一条逗号分隔的键值对（忽略转义问题的话，大致与 CSV 文件类似）。每次对 `db_set` 的调用都会向文件末尾追加记录，所以更新键的时候旧版本的值不会被覆盖 —— 因而查找最新值的时候，需要找到文件中键最后一次出现的位置（因此 `db_get` 中使用了 `tail -n 1` 。）

Bash

```
$ db_set 42 '{"name":"San Francisco","attractions":["Exploratorium"]}'

$ db_get 42
{"name":"San Francisco","attractions":["Exploratorium"]}

$ cat database
123456,{"name":"London","attractions":["Big Ben","London Eye"]}
42,{"name":"San Francisco","attractions":["Golden Gate Bridge"]}
42,{"name":"San Francisco","attractions":["Exploratorium"]}
```

`db_set` 函数对于**极其简单的场景其实有非常好的性能**，因为在文件尾部追加写入通常是非常高效的。与 `db_set` 做的事情类似，许多数据库在内部使用了**日志**，也就是一个 **append-only** 的数据文件。真正的数据库有更多的问题需要处理（如并发控制，回收磁盘空间以避免日志无限增长，处理错误与部分写入的记录），但基本原理是一样的。

另一方面，如果这个数据库中有着大量记录，则这个 `db_get` 函数的性能会非常糟糕。每次你想查找一个键时，`db_get` 必须从头到尾扫描整个数据库文件来查找键的出现。查找的开销是 `O(n)`：如果数据库记录数量 n 翻了一倍，查找时间也要翻一倍。这就不好了。

怎么加速 `get` 的性能?

为了高效查找数据库中特定键的值，我们需要一个数据结构：**索引**。索引背后的大致思想是，保存一些额外的元数据作为路标，帮助你找到想要的数据。如果想在同一份数据中以几种不同的方式进行搜索，那么你也许需要不同的索引，建在数据的不同部分上。

索引是从主数据衍生的**附加**结构。许多数据库允许添加与删除索引，这不会影响数据的内容，它只影响查询的性能。维护额外的结构会产生开销，特别是在写入时。写入性能很难超过简单地追加写入文件，因为追加写入是最简单的写入操作。**任何类型的索引通常都会减慢写入速度，因为每次写入数据时都需要更新索引**。

> 💡 这是存储系统中一个重要的权衡：精心选择的索引加快了读查询的速度，但是每个索引都会拖慢写入速度。因为这个原因，数据库默认并不会索引所有的内容，而需要你通过对应用查询模式的了解来手动选择索引。你可以选择能为应用带来最大收益，同时又不会引入超出必要开销的索引。

哈希索引

让我们从 **键值数据（key-value Data）** 的索引开始。对于更复杂的索引来说，这是一个有用的构建模块。

键值存储与在大多数编程语言中可以找到的 **dictionary 的数据类型非常相似，通常字典都是用散列映射**实现的。既然我们已经有**内存中**数据结构 —— hash map，为什么不使用它来索引在**磁盘上**的数

据呢？

假设我们的数据存储只是一个追加写入的文件，就像前面的例子一样。那么最简单的索引策略就是：保留一个内存中的哈希映射，其中每个键都映射到一个数据文件中的字节偏移量，指明了可以找到对应值的位置，如下图所示。当你将新的键值对追加写入文件中时，要更新散列映射，以反映刚刚写入的数据的偏移量（这同时适用于插入新键与更新现有键）。当你想查找一个值时，使用哈希映射来查找数据文件中的偏移量，**寻找（seek）** 该位置并读取该值。

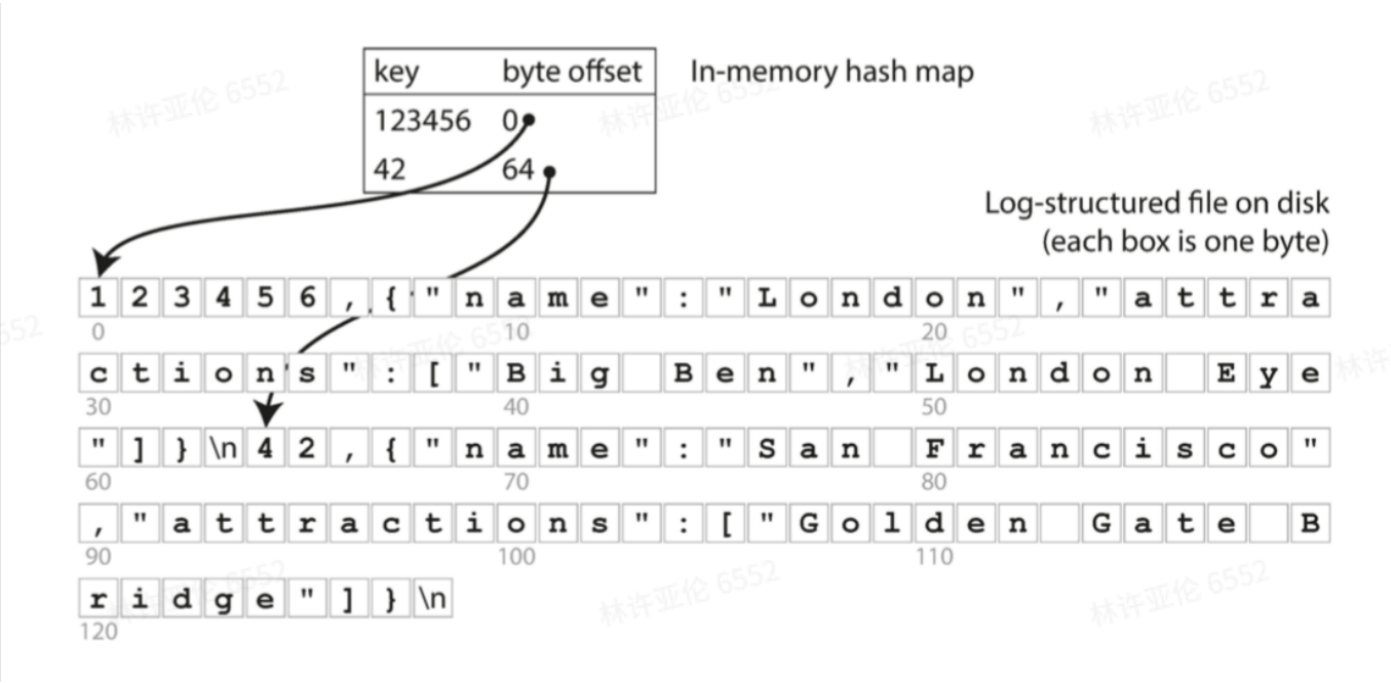


图 以类 CSV 格式存储键值对的日志，并使用内存哈希映射进行索引

现实中，Bitcask 实际上就是这么做的（Riak 中默认的存储引擎）。Bitcask 提供高性能的读取和写入操作，但所有键必须能放入可用内存中，因为哈希映射完全保留在内存中。这些值可以使用比可用内存更多的空间，因为可以从磁盘上通过一次 `seek` 加载所需部分，如果数据文件的那部分已经在文件系统缓存中，则读取根本不需要任何磁盘 I/O。

像 Bitcask 这样的存储引擎非常适合每个键的值经常更新的情况。例如，键可能是视频的 URL，值可能是它播放的次数（每次有人点击播放按钮时递增）。在这种类型的工作负载中，有很多写操作，但是没有太多不同的键 —— 每个键有很多的写操作，但是将所有键保存在内存中是可行的。

直到现在，我们只是追加写入一个文件 —— 所以如何避免最终用完磁盘空间？一种好的解决方案是，将日志分为特定大小的段（segment），当日志增长到特定尺寸时关闭当前段文件，并开始写入一个新的段文件。然后，我们就可以对这些段进行**压缩**。压缩意味着在日志中丢弃重复的键，只保留每个键的最近更新。

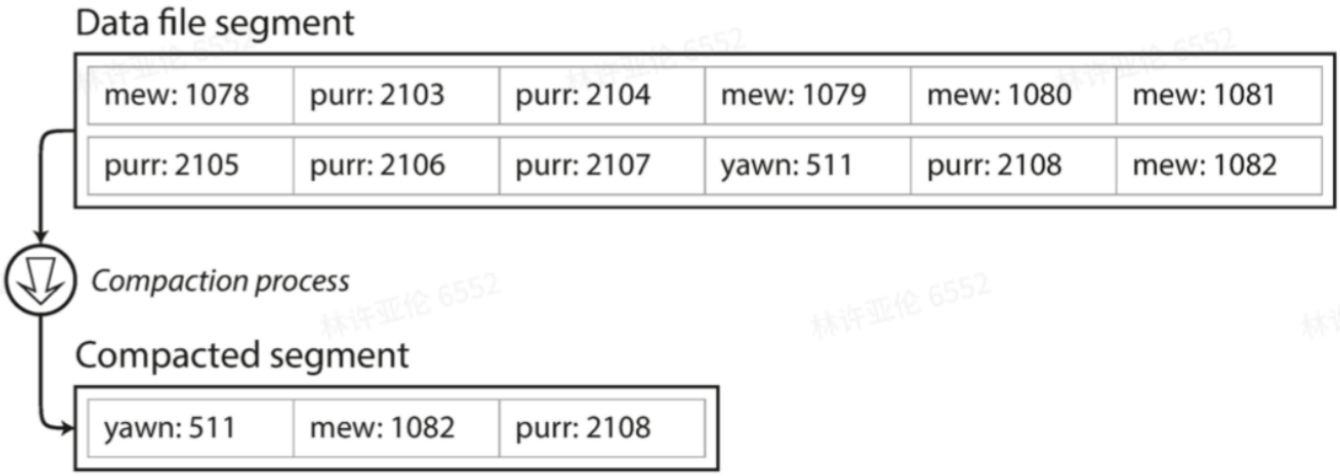
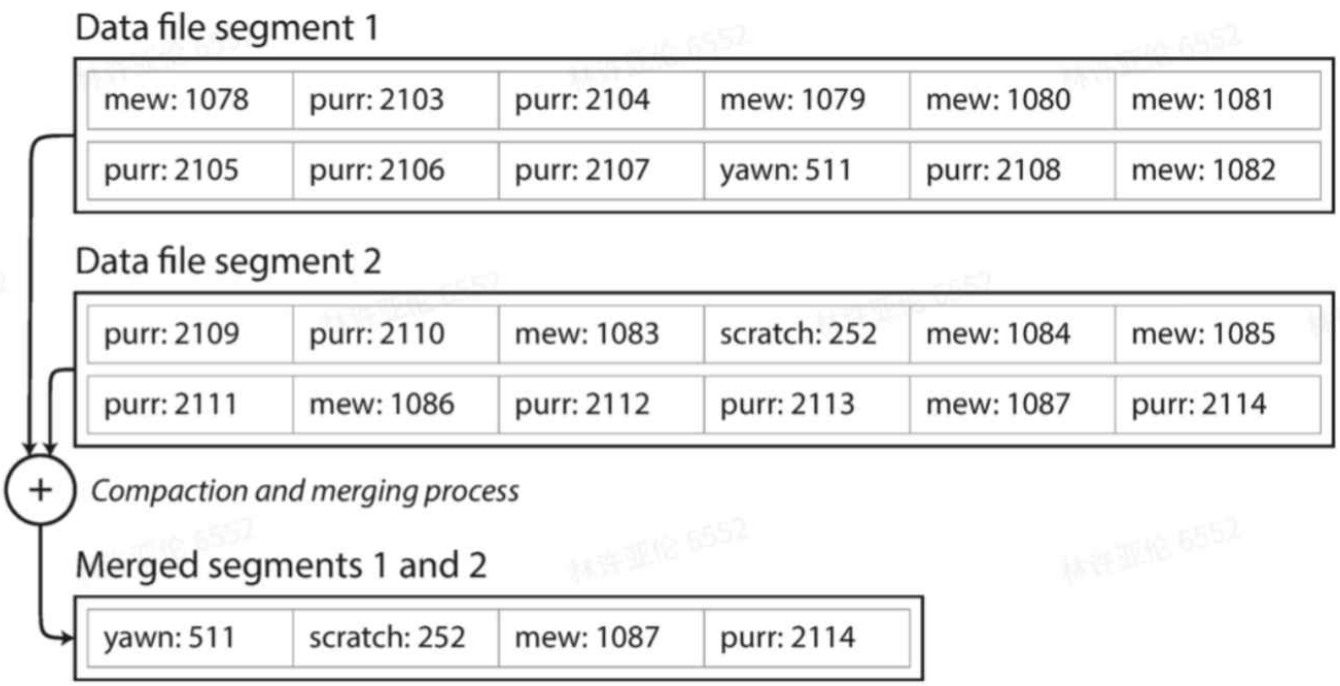


图 压缩键值更新日志（统计猫视频的播放次数），只保留每个键的最近值

而且，由于压缩经常会使得段变得很小（假设在一个段内键被平均重写了好几次），我们也可以在执行压缩的同时将多个段合并在一起，如下图所示。段被写入后永远不会被修改，所以合并的段被写入一个新的文件。冻结段的合并和压缩可以在后台线程中完成，在进行时，我们仍然可以继续使用旧的段文件来正常提供读写请求。合并过程完成后，我们将读取请求转换为使用新的合并段而不是旧段 —— 然后可以简单地删除旧的段文件。



每个段现在都有自己的内存散列表，将键映射到文件偏移量。**为了找到一个键的值，我们首先检查最近段的哈希映射；如果键不存在，我们检查第二个最近的段，依此类推**。合并过程保持细分的数量，所以查找不需要检查许多哈希映射。 大量的细节进入实践这个简单的想法工作。简而言之，一些真正实施中重要的问题是：

删除记录

如果要删除一个键及其关联的值，则必须在数据文件（有时称为逻辑删除）中附加一个特殊的删除记录。当日志段被合并时，逻辑删除告诉合并过程放弃删除键的任何以前的值。

崩溃恢复

如果数据库重新启动，则内存散列映射将丢失。原则上，您可以通过从头到尾读取整个段文件并在每次按键时注意每个键的最近值的偏移量来恢复每个段的哈希映射。但是，如果 segment 文件很大，这可能需要很长时间。解决方式也很直接，快照。 Bitcask 通过存储加速恢复磁盘上每个段的哈希映射的快照，可以更快地加载到内存中。

部分写入记录

数据库可能随时崩溃，包括将记录附加到日志中途。 Bitcask 文件包含校验和，允许检测和忽略日志的这些损坏部分。

并发控制

由于写操作是以严格顺序的顺序附加到日志中的，所以常见的实现选择是只有一个写入器线程。数据文件是不可变的，所以它们可以被多个线程同时读取。

乍一看，**只追加日志**看起来很浪费：**为什么不更新文件，用新值覆盖旧值？**这样设计的原因有几个：

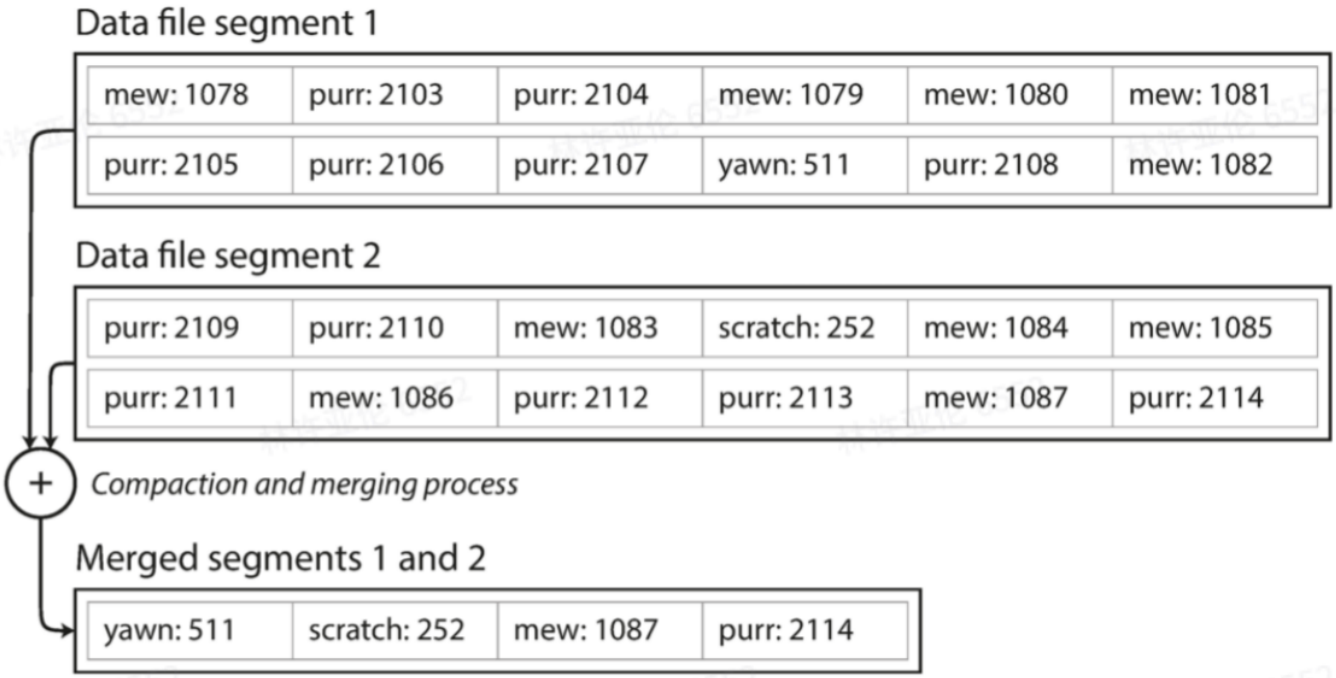
- 追加和分段合并是顺序写入操作，通常比随机写入快得多，尤其是在磁盘旋转硬盘上。在某种程度上，顺序写入在基于闪存的 **固态硬盘（SSD）** 上也是优选的；
- 如果段文件是不可变的，并发和崩溃恢复就简单多了。例如，不必担心在覆盖值时发生崩溃的情况，而将包含旧值和新值的一部分的文件保留在一起；
- 合并旧段可以避免数据文件随着时间的推移而分散的问题。

但是，哈希表索引也有局限性：

- 散列表必须能放进内存：如果你有非常多的键，那真是倒霉。原则上可以在磁盘上保留一个哈希映射，不幸的是磁盘哈希映射很难表现优秀。它需要大量的随机访问 I/O，当它变满时增长是很昂贵的，并且散列冲突需要很多的逻辑；
- 范围查询效率不高。例如，无法轻松扫描 kitty00000 和 kitty99999 之间的所有键 —— 必须在散列映射中单独查找每个键。

因此，我们需要没有这些限制的索引结构。

SSTables 和 LSM Tree

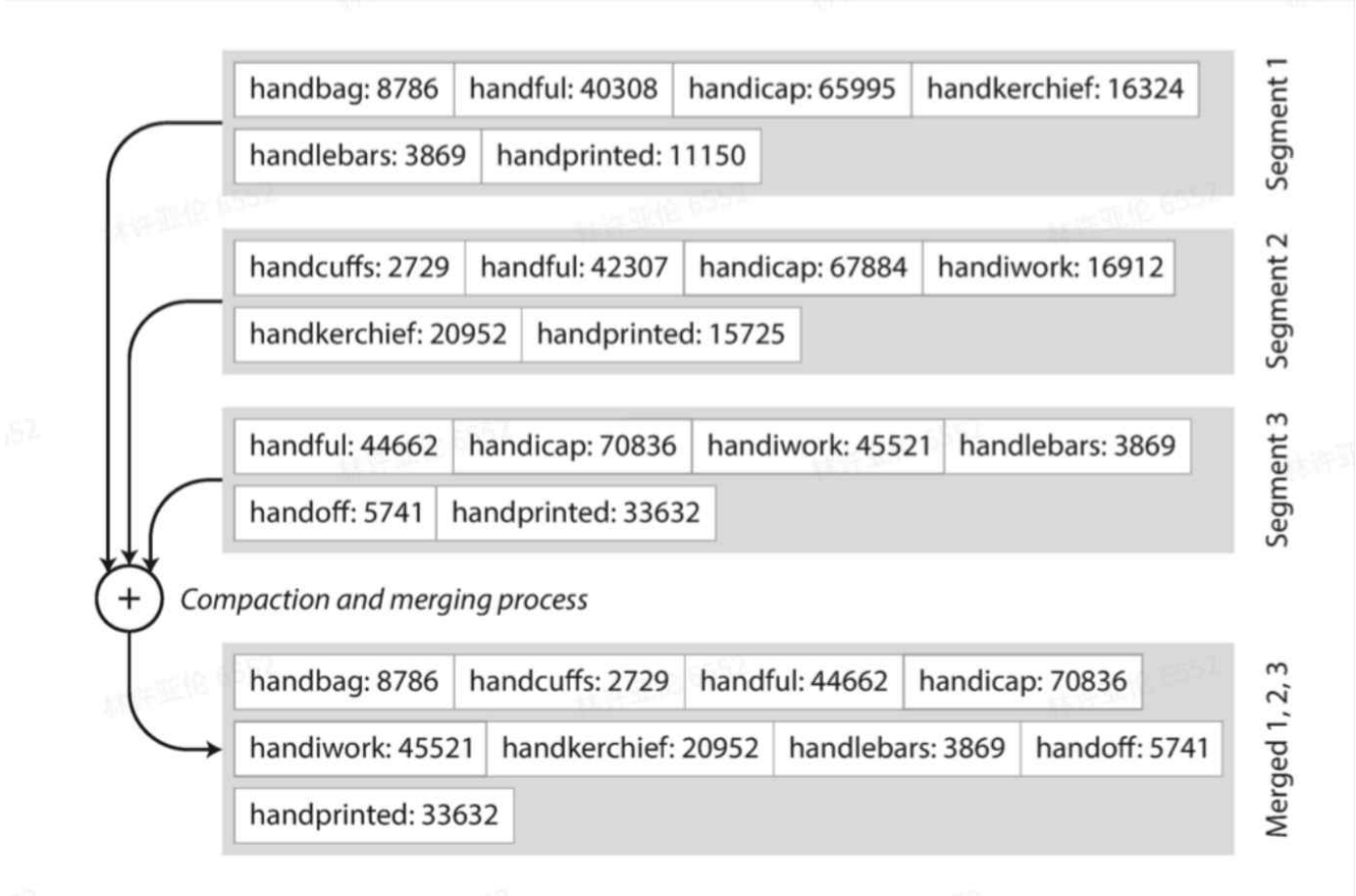


在上图中，每个日志结构存储段都是一系列键值对。这些对按照它们写入的顺序出现，日志中稍后的值优先于日志中较早的相同键的值。除此之外，文件中键值对的顺序并不重要。

为了解决上节提出的哈希索引的两个问题，现在我们可以对段文件的格式做一个简单的改变：**我们要求键值对的序列按键排序**。乍一看，这个要求似乎打破了我们使用顺序写入的能力，但是我们马上就会明白这一点。

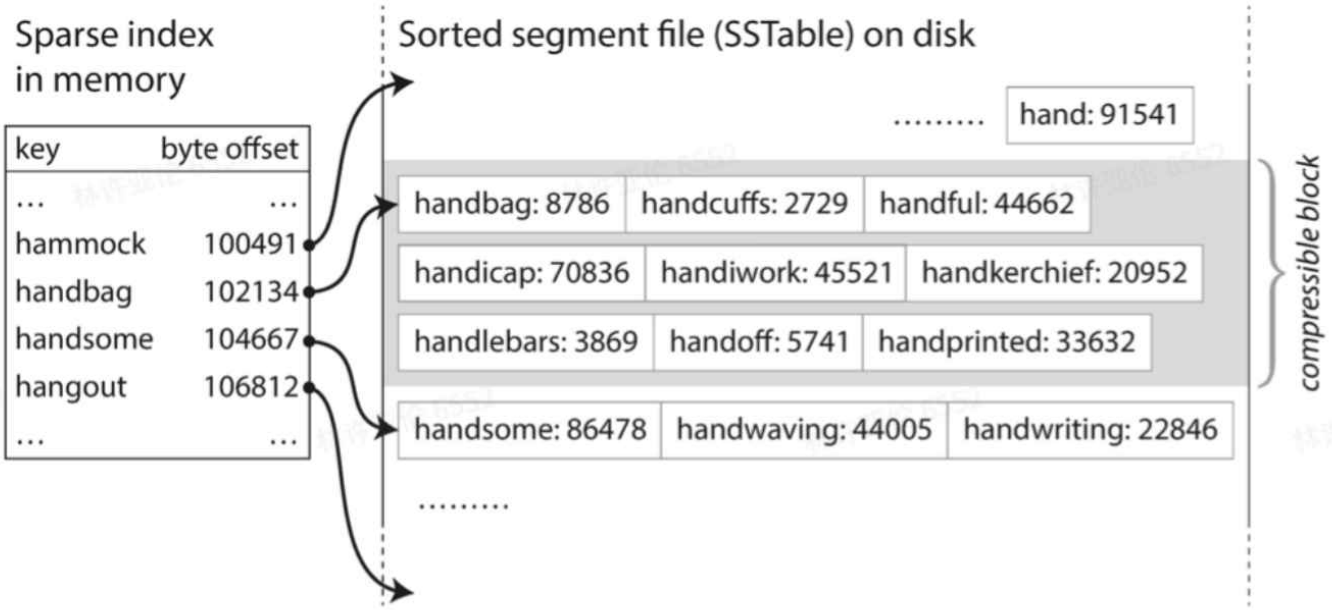
我们把这个格式称为 **Sorted String Table**，简称 SSTable。我们还要求每个键只在每个合并的段文件中出现一次（压缩过程已经保证）。与使用散列索引的日志段相比，SSTable 有几个很大的优势：

1. 合并段是简单而高效的，即使文件大于可用内存。这种方法就像归并排序算法中使用的方法一样，开始并排读取输入文件，查看每个文件中的第一个键，复制最低键（根据排序顺序）到输出文件，并重复。这产生一个新的合并段文件，也按键排序。



如果在几个输入段中出现相同的键，该怎么办？每个段都包含在一段时间内写入数据库的所有值。这意味着一个输入段中的所有值一定比另一个段中的所有值更新（假设我们总是合并相邻的段）。当多个段包含相同的键时，我们可以保留最近段的值，并丢弃旧段中的值。

2. 为了在文件中找到一个特定的键，不再需要保存内存中所有键的索引。以下图为例：假设你正在内存中寻找键 `handiwork`，但是你不知道段文件中该关键字的确切偏移量。然而，你知道 `handbag` 和 `handsome` 的偏移，而且由于排序特性，你知道 `handiwork` 必须出现在这两者之间。这意味着可以跳到 `handbag` 的偏移位置并从那里扫描，直到找到 `handiwork`（或没找到，如果该文件中没有该键）。



我们仍然需要一个内存索引来记录一些键的偏移量，但它可能很稀疏：每几千字节的段文件就有一个键就足够了，因为几千字节可以很快被扫描。

3. 由于读取请求无论如何都需要扫描所请求范围内的多个键值对，因此可以将这些数据 item 分组到一个个 block 中，在将其写入磁盘之前对其进行压缩（如上图中的阴影区域所示）。稀疏内存中索引的每个条目都指向压缩块的开始处。除了节省磁盘空间之外，压缩还可以减少 IO 带宽的使用。

构建和维护 SSTables

到目前为止，但是如何让你的数据首先被按键排序呢？

在磁盘上维护有序结构是可能的（比如 B 树），但在内存保存则要容易得多。有许多可以使用的树形数据结构，例如红黑树或 AVL 树。使用这些数据结构，就可以按任何顺序插入键，并按排序顺序读取它们。

现在我们可以使我们的存储引擎工作如下：

- 写入时，将其添加到内存中的平衡树数据结构（例如，红黑树）。这个内存树有时被称为**内存表（memtable）**。
- 当**内存表**大于某个阈值（通常为几兆字节）时，将其作为 SSTable 文件写入磁盘。这可以高效地完成，因为树已经维护了按键排序的键值对。新的 SSTable 文件成为数据库的最新部分。当 SSTable 被写入磁盘时，写入可以继续到一个新的内存表实例。
- 为了提供读取请求，首先尝试在内存表中找到关键字，然后在最近的磁盘段中，然后在下一个较旧的段中找到该关键字。
- 有时会在后台运行合并和压缩过程以组合段文件并丢弃覆盖或删除的值。

这个方案效果很好。它只会遇到一个问题：如果数据库崩溃，则最近的写入（在内存表中，但尚未写入磁盘）将丢失。为了避免这个问题，我们可以在磁盘上维护一个临时日志文件，每个写入都会立即

被附加到磁盘上，就像在前一节中一样。该日志不是按排序顺序，但这并不重要，因为它的唯一目的是在崩溃后恢复内存表。每当内存表写出到 SSTable 时，相应的日志都可以被丢弃。

用 SSTables 制作 LSM 树

这里描述的算法本质上是 LevelDB 和 RocksDB 中使用的键值存储引擎库，被设计嵌入到其他应用程序中。除此之外，LevelDB 可以在 Riak 中用作 Bitcask 的替代品。在 Cassandra 和 HBase 中使用了类似的存储引擎，这两种引擎都受到了 Google 的 Bigtable（引入了 SSTable 和 memtable）的启发。

最初这种索引结构是由 Patrick O'Neil 等人介绍的，且被命名为 LSM Tree，它是基于更早之前的日志结构文件系统来构建的。基于这种合并和压缩排序文件原理的存储引擎通常被称为 LSM 存储引擎。

LSM Tree 上的性能优化

与往常一样，**大量的细节使得存储引擎在实践中表现良好**。

例如，当查找数据库中不存在的键时，LSM Tree 可能会很慢：首先必须检查内存表，然后将这些段一直遍历，直到搜索到最旧的数据（可能必须从磁盘读取每一个），然后才能确定键不存在。为了优化这种访问，存储引擎通常使用额外的布隆过滤器。（布隆过滤器是用于近似集合内容的内存高效数据结构，从而为不存在的键节省许多不必要的磁盘读取操作。）

还有不同的策略来确定 SSTables 如何被压缩和合并的顺序和时间。最常见的选择是 size-tiered 和 leveled compaction。LevelDB 和 RocksDB 使用 leveled compaction（LevelDB 因此得名），HBase 使用 size-tiered。对于 sized-tiered，较新和较小的 SSTables 相继被合并到较旧的和较大的 SSTable 中。对于 leveled compaction，key 范围被拆分到较小的 SSTables，而较旧的数据被移动到单独的层级，这使得压缩能够更加增量地进行，并且使用较少的磁盘空间。

即使有许多微妙的东西，LSM 树的基本思想，即**保存一系列在后台合并的 SSTables** 是简单而有效的。即使数据集比可用内存大得多，它仍能继续正常工作。由于数据按排序顺序存储，因此可以高效地执行范围查询（扫描所有高于某些最小值和最高值的所有键），并且因为磁盘写入是连续的，所以 LSM 树可以支持非常高的写入吞吐量。

B 树

> 亚辉在此前对索引和 B 树有详细的介绍，见 indexes。

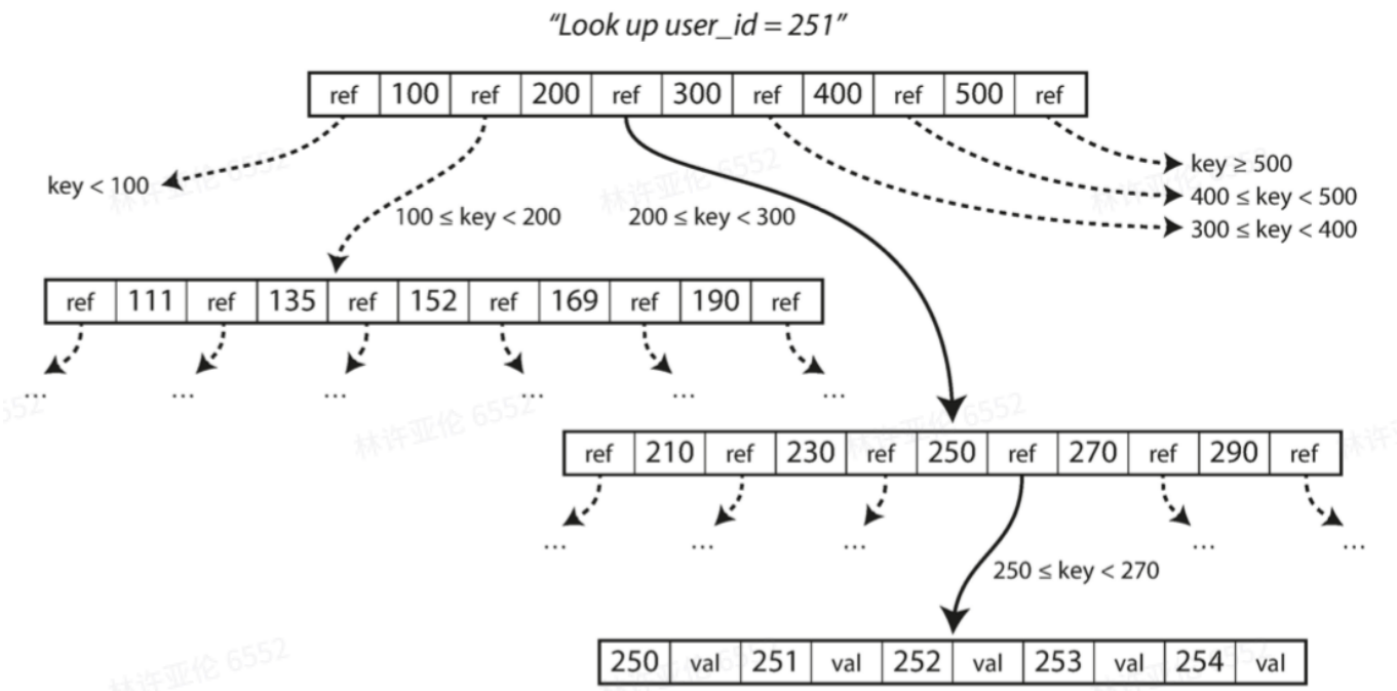
刚才讨论的日志结构索引正处在逐渐被接受的阶段，但它们并不是最常见的索引类型。使用最广泛的索引结构在 1970 年被引入，不到 10 年后变得 “无处不在”，B 树经受了时间的考验。在几乎所有的关系数据库中，它们仍然是标准的索引实现，许多非关系数据库也使用它们。

像 SSTables 一样，**B 树保持按键排序的键值对，这允许高效的键值查找和范围查询**。但除此之外，B 树有着非常不同的设计理念。

我们前面看到的日志结构索引将数据库分解为可变大小的段，通常是几兆字节或更大的大小，并且总是按顺序编写 segment。相比之下，B 树将数据库分解成固定大小的块或页面，传统上大小为 4KB

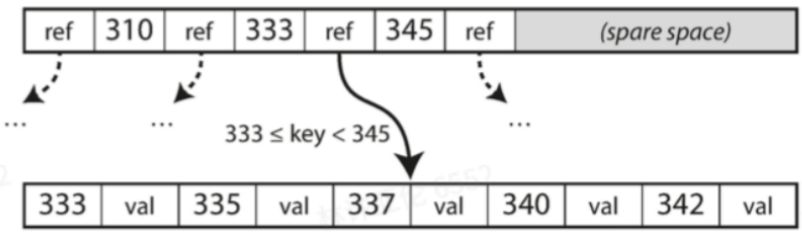
（有时会更大），并且一次只能读取或写入一个页面。这种设计更接近于底层硬件，因为磁盘也被安排在固定大小的块中。

每个页面都可以使用地址或位置来标识，这允许一个页面引用另一个页面 —— 类似于指针，但指向的是磁盘而不是在内存中。我们可以使用这些页面引用来构建一个页面树：

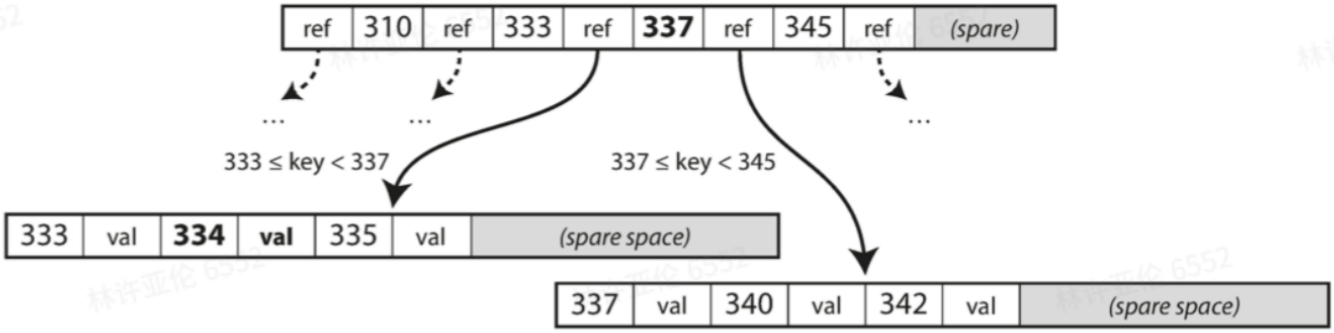


一个页面会被指定为 B 树的根；在索引中查找一个键时，就从这里开始。该页面包含几个键和对子页面的引用。每个子页面负责一段连续范围的键，引用之间的键，指明了引用子页面的键范围。

如果要更新 B 树中现有键的值，则搜索包含该键的叶子页，更改该页中的值，并将该页写回到磁盘，如果你想添加一个新的键，你需要找到其范围包含新键的页面，并将其添加到该页面。如果页面中没有足够的可用空间容纳新键，则将其分成两个半满叶子页面，并更新父页面以解释键范围的新分区，如下图：



After adding key 334:



该算法确保树保持平衡：具有 n 个键的 B 树总是具有 `O(log n)` 的深度。大多数数据库可以放入一个三到四层的 B 树。 （分支因子为 500 的 4KB 页面的四级树可以存储多达 256TB 。）

让 B 树更可靠

B 树的基本底层写操作是用新数据覆盖磁盘上的页面。假定覆盖不改变页面的位置：即，当页面被覆盖时，对该页面的所有引用保持完整。这与日志结构索引（如 LSM 树）形成鲜明对比，后者只附加到文件（并最终删除过时的文件），但从不修改文件。

可以考虑将硬盘上的页面覆盖为实际的硬件操作。在磁性硬盘驱动器上，这意味着将磁头移动到正确的位置，等待旋转盘上的正确位置出现，然后用新的数据覆盖适当的扇区。

而且，一些操作需要覆盖几个不同的页面。例如，如果因为插入导致页面过满而拆分页面，则需要编写已拆分的两个页面，并覆盖其父页面以更新对两个子页面的引用。这是一个危险的操作，因为如果数据库在仅有一些页面被写入后崩溃，那么最终将导致一个损坏的索引（例如，可能有一个孤儿页面不是任何父项的子项） 。

为了使数据库对崩溃具有韧性，B 树实现通常会带有一个额外的磁盘数据结构：**WAL, write-ahead-log**（也称为**重做日志（redo log）**）。这是一个仅追加的文件，每个 B 树修改都可以应用到树本身的页面上。当数据库在崩溃后恢复时，这个日志被用来使 B 树恢复到一致的状态。

比较 B 树和 LSM 树

尽管 B 树实现通常比 LSM 树实现更成熟，但 LSM 树由于其性能特点也非常有趣。根据经验，通常 LSM 树的写入速度更快，而 B 树的读取速度更快。 LSM 树上的读取通常比较慢，因为它们必须检查几种不同的数据结构和不同压缩层级的 SSTables。

LSM 树的优点

B 树索引必须至少两次写入每一段数据：一次写入预先写入日志，一次写入树页面本身（也许再次分页）。即使在该页面中只有几个字节发生了变化，也需要一次编写整个页面的开销。

由于反复压缩和合并 SSTables，日志结构索引也会重写数据。这种影响 被称为**写放大**。

在写入繁重的应用程序中，性能瓶颈可能是数据库可以写入磁盘的速度。在这种情况下，写放大会导致直接的性能代价：存储引擎写入磁盘的次数越多，可用磁盘带宽内的每秒写入次数越少。

而且，LSM 树通常能够比 B 树支持更高的写入吞吐量，部分原因是它们有时具有较低的写放大（尽管这取决于存储引擎配置和工作负载），部分是因为它们顺序地写入紧凑的 SSTable 文件而不是必须覆盖树中的几个页面。这种差异在磁性硬盘驱动器上尤其重要，顺序写入比随机写入快得多。

LSM 树可以被压缩得更好，因此经常比 B 树在磁盘上产生更小的文件。 B 树存储引擎会由于分割而留下一些未使用的磁盘空间：当页面被拆分或某行不能放入现有页面时，页面中的某些空间仍未被使用。由于 LSM 树不是面向页面的，并且定期重写 SSTables 以去除碎片，所以它们具有较低的存储开销，特别是当使用分层压缩时。

LSM 树的缺点

日志结构存储的缺点是**压缩过程有时会干扰正在进行的读写操作**。尽管存储引擎尝试逐步执行压缩而不影响并发访问，但是磁盘资源有限，所以很容易发生请求需要等待磁盘完成昂贵的压缩操作。对吞吐量和平均响应时间的影响通常很小，但是在更高百分比的情况下，对日志结构化存储引擎的查询响应时间有时会相当长，而 B 树的行为则相对更具可预测性。

压缩的另一个问题出现在高写入吞吐量：磁盘的有限写入带宽需要在初始写入（记录和刷新内存表到磁盘）和在后台运行的压缩线程之间共享。写入空数据库时，可以使用全磁盘带宽进行初始写入，但数据库越大，压缩所需的磁盘带宽就越多。

如果写入吞吐量很高，并且压缩没有仔细配置，压缩跟不上写入速率。在这种情况下，磁盘上未合并段的数量不断增加，直到磁盘空间用完，读取速度也会减慢，因为它们需要检查更多段文件。

B 树的一个优点是每个键只存在于索引中的一个位置，而 LSM Tree 可能在不同的段中有相同键的多个副本。这个方面使得 B 树在提供强大的事务语义的数据库中很有吸引力：在许多关系数据库中，事务隔离是通过在键范围上使用锁来实现的，在 B 树索引中，这些锁可以直接连接到树上。

CLP: Efficient and Scalable Search on Compressed Text Logs

CLP: Efficient and Scalable Search on Compressed Text Logs · Issue #15 ····

https://www.usenix.org/system/files/osdi21-rodrigues.pdf osdi2021

https://github.com/linxuyalun/paper-reading/issues/15