Data Structures That Power Databases

Why should you, as an application developer, care how the database handles storage

and retrieval internally? You’re probably not going to implement your own storage

engine from scratch, but you *do* need to select a storage engine that is appropriate for

your application, from the many that are available. In order to tune a storage engine

to perform well on your kind of workload, you need to have a rough idea of what the

storage engine is doing under the hood.

We will examine two families of storage engines: ***log-structured* storage engines**, and ***page-oriented* storage engines** such as B-trees.

# Introduction

Consider the world’s simplest database. two functions implement a key-value store. You can call db\_set key value, which will store key and value in the database. The key and value can be (almost) anything you like—for example, the value could be a JSON document. You can then call db\_get key, which looks up the most recent value associated with that particular key and returns it.

The underlying storage format is very simple: a text file where each line contains a

key-value pair, separated by a comma (roughly like a CSV file, ignoring escaping

issues). Every call to db\_set appends to the end of the file, so if you update a key several

times, the old versions of the value are not overwritten—you need to look at the

last occurrence of a key in a file to find the latest value.

**Our db\_set function actually has pretty good performance** for something that is so

simple, because appending to a file is generally very efficient. Similar to what db\_set does, many databases internally use **a log, which is an append-only data file**. Real databases have more issues to deal with (such as concurrency control, reclaiming disk space so that the log doesn’t grow forever, and handling errors and partially written records), but the basic principle is the same. Logs are incredibly useful, and we will encounter them several times in the rest of this book.

In this book, ***log* is used in the more general sense: an append-only sequence of records**. It doesn’t have to be human-readable; it might be binary and intended only for other programs to read.

**On the other hand, our db\_get function has terrible performance** if you have a large number of records in your database. Every time you want to look up a key, db\_get has to scan the entire database file from beginning to end, looking for occurrences of the key. In algorithmic terms, the cost of a lookup is ***O*(*n*):** if you double the number of records *n* in your database, a lookup takes twice as long. That’s not good.

**In order to efficiently find the value for a particular key in the database, we need a different data structure: an *index****.* the **general idea** behind them **is to keep some**

**additional metadata on the side**, which acts as a signpost and **helps you to locate the**

**data you want.**

An index is an *additional* structure that is derived from the primary data. Many databases

allow you to add and remove indexes, and **this doesn’t affect the contents of the**

**database;** **it only affects the performance of queries**. Maintaining additional structures

incurs overhead, especially on writes. For writes, it’s hard to beat the performance of

simply appending to a file, because that’s the simplest possible write operation. **Any**

**kind of index usually slows down writes, because the index also needs to be updated**

**every time data is written.**

This is an important trade-off in storage systems: **well-chosen indexes speed up read**

**queries, but every index slows down writes**. For this reason, databases don’t usually

index everything by default, but require you—the application developer or database

administrator—to choose indexes manually, using your knowledge of the application’s

typical query patterns. You can then choose the indexes that give your application

the greatest benefit, without introducing more overhead than necessary.

# Hash Indexes

Let’s get back to our simple key-value storage we talked about in the introduction:

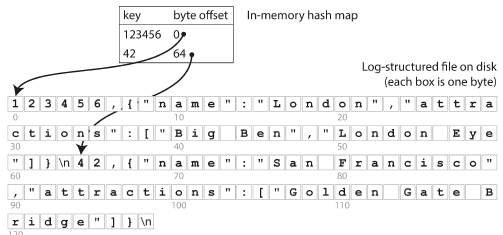
$ cat database

123456,{"name":"London","attractions":["Big Ben","London Eye"]}

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

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

the simplest possible indexing strategy is this: keep **an in-memory hash map** where every **key is mapped to a byte offset in the data file(on disk)**—the location at which the value can be found, as illustrated in Figure 3-1. Whenever you append a new key-value pair to the file, you also update the hash map to reflect the offset of the data you just wrote (this works both for inserting new keys and for updating existing keys). **When you want to look up a value, use the hash map to find the offset in the data file, seek to that location, and read the value.**



since the hash map is kept completely in memory. The **values** can use more space than there is available memory, since they **can be loaded from disk with just one disk seek**. If that part of the data file is already in the filesystem cache, a read doesn’t require any disk I/O at all.

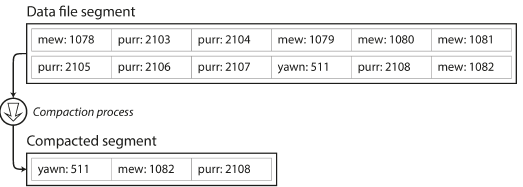
A storage engine like Bitcask is well suited to situations where the value for each key is updated frequently. For example, the key might be the URL of a cat video, and the value might be the number of times it has been played (incremented every time someone hits the play button). In this kind of workload**, there are a lot of writes, but there are not too many distinct keys—you have a large number of writes per key, but it’s feasible to keep all keys in memory.**

As described so far, we only ever append to a file—so how do we avoid eventually

running out of disk space? A good solution is to break the log into segments of a certain

size by closing a segment file when it reaches a certain size, and making subsequent

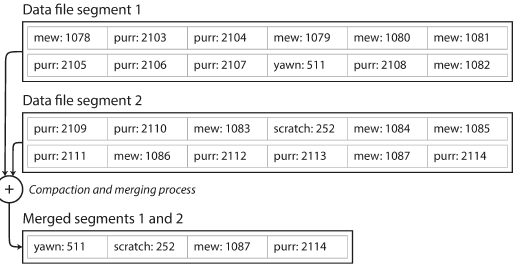
writes to a new segment file. We can then perform ***compaction* on these segments**. Compaction means throwing away duplicate keys in the log, and keeping only the most recent update for each key.



Moreover, since compaction often makes segments much smaller (assuming that a

key is overwritten several times on average within one segment), we can also merge

several segments together at the same time as performing the compaction. **Segments are never modified after they have been written, so the merged segment is written to a new file.** The merging and compaction of frozen segments can be done in a background thread, and while it is going on, we can still continue to serve read and write requests as normal, using the old segment files. After the merging process is complete, we switch read requests to using the new merged segment instead of the old segments—and then the old segment files can simply be deleted.



**Each segment now has its own in-memory hash table**, mapping keys to file offsets. In

order to find the value for a key, we first check the most recent segment’s hash map;

if the key is not present, we check the second-most-recent segment, and so on. The

merging process keeps the number of segments small, so lookups don’t need to check many hash maps.

Lots of detail goes into making this simple idea work in practice. Briefly, some of the issues that are important in a real implementation are:

* **File format**

CSV is not the best format for a log. It’s faster and simpler to use a binary format that first encodes the length of a string in bytes, followed by the raw string (without need for escaping).

* **Deleting records**

If you want to delete a key and its associated value**, you have to append a special**

**deletion record to the data file** (sometimes **called a tombstone**). When log segments

are merged, the tombstone tells the merging process to discard any previous

values for the deleted key.

* **Crash recovery**

If the database is restarted, **the in-memory hash maps are lost**. In principle, you

can restore each segment’s hash map by reading the entire segment file from

beginning to end and noting the offset of the most recent value for every key as

you go along. However, that might take a long time if the segment files are large,

which would make server restarts painful**. Bitcask speeds up recovery by storing a snapshot of each segment’s hash map on disk**, which can be loaded into memory more quickly.

* **Partially written records**

The database may crash at any time, including halfway through appending a

record to the log. **Bitcask files include checksums, allowing such corrupted parts**

**of the log to be detected and ignored**.

* **Concurrency control**

As writes are appended to the log in a strictly sequential order, a common implementation

choice is to have **only one writer thread**. Data file segments are append-only and otherwise immutable, so they can be read concurrently by multiple threads.

An append-only log seems wasteful at first glance: **why don’t you update the file in**

**place, overwriting the old value with the new value?** But an append-only design turns out to be good for several reasons:

* Appending and segment merging are sequential write operations, which are generally **much faster than random writes**, especially on magnetic *spinning-disk hard drives*. To some extent sequential writes are also preferable on flash-based *solid-state drives* (SSDs). We will discuss this issue further in “Comparing B-Trees and LSM-Trees”.
* Concurrency and crash recovery are much simpler if segment files are append-only or immutable. For example, you don’t have to worry about the case where a crash happened while a value was being overwritten, leaving you with a file containing part of the old and part of the new value spliced together.
* Merging old segments avoids the problem of data files getting fragmented over time.

However, the hash table index also has limitations:

* **The hash table must fit in memory**, so if you have a very large number of keys, you’re out of luck. In principle, you could maintain a hash map on disk, but unfortunately it is difficult to make an on-disk hash map perform well. **It requires a lot of random access I/O**, it is expensive to grow when it becomes full, and hash collisions require fiddly logic.
* **Range queries are not efficient**. For example, you cannot easily scan over all keys between kitty00000 and kitty99999—you’d have to look up each key individually in the hash maps. In the next section we will look at an indexing structure that doesn’t have those limitations.

# SSTables and LSM-Trees

In Figure 3-3 (the figure from last section showing merging process), each log-structured storage segment is a sequence of key-value pairs.

**These pairs appear in the order that they were written**, and **values later in the log take precedence over values for the same key earlier in the log.** Apart from that, the orderof key-value pairs in the file does not matter.

Now we can make a simple change to the format of our segment files: we require that **the sequence of key-value pairs is *sorted by key*.** At first glance, that requirement **seems to break our ability to use sequential writes**, but we’ll get to that in a moment.

We call this format **Sorted String Table, or SSTable** for short. We also require that

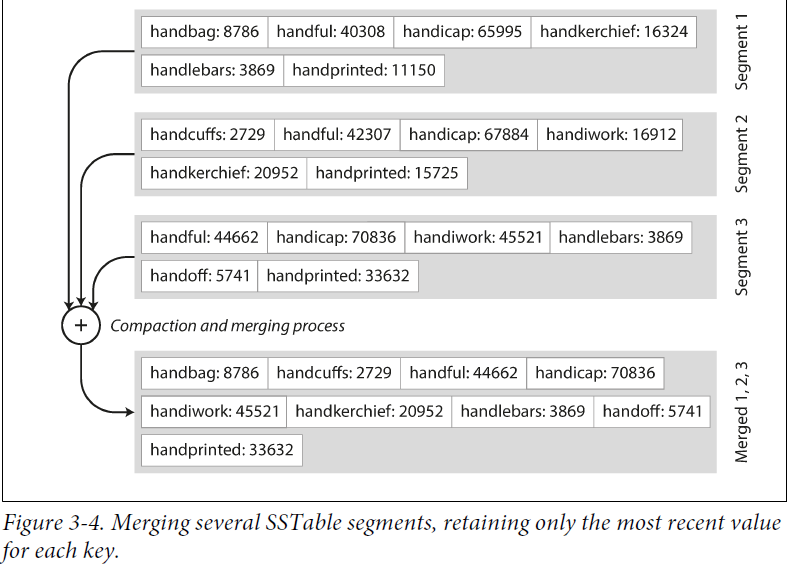
each key only appears once within each merged segment file (the compaction process

already ensures that). **SSTables have several big advantages over log segments with**

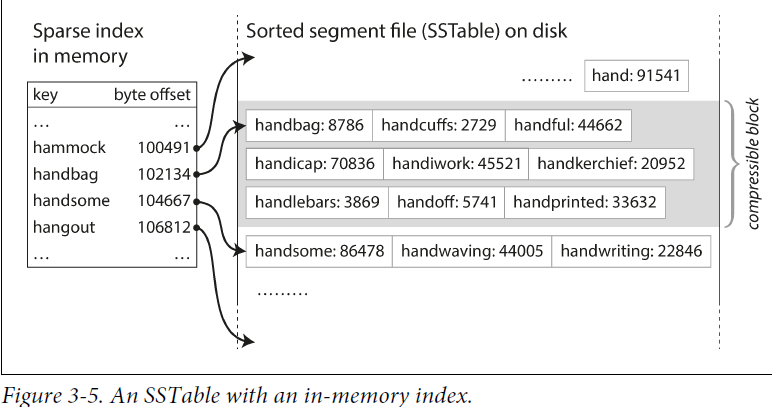
**hash indexes:**

1. Merging segments is simple and efficient, even if the files are bigger than the available memory. The approach is like the one used in the *merge-sort* algorithm and is illustrated in Figure 3-4. you start reading the input files side by side, look at the first key in each file, copy the lowest key (according to the sort order) to the output file, and repeat. **This produces a new merged segment file, also sorted by key.**

What if the same key appears in several input segments? Remember that each segment contains all the values written to the database during some period of time. This means that all the values in one input segment must be more recent than all the values in the other segment (assuming that we always merge adjacent segments). When multiple segments contain the same key, we can keep the value from the most recent segment and discard the values in older segments:



2. In order to find a particular key in the file, you no longer need to keep an index of all the keys in memory. See Figure 3-5 for an example: say you’re looking for the key handiwork, but you don’t know the exact offset of that key in the segment file. However, you do know the offsets for the keys, *handbag* and *handsome*, and because of the sorting you know that *handiwork* must appear between those two. This means you can jump to the offset for *handbag* and scan from there until you find *handiwork* (or not, if the key is not present in the file).



You still need an in-memory index to tell you the offsets for some of the keys, but it can be sparse: one key for every few kilobytes of segment file is sufficient, because a few kilobytes can be scanned very quickly.

As a side note: If all keys and values had a fixed size, you could use binary search on a segment file and avoid the in-memory index entirely. However, they are usually variable-length in practice, which makes it difficult to tell where one record ends and the next one starts if you don’t have an index.

3. Since read requests need to scan over several key-value pairs in the requested range anyway, it is possible to group those records into a block and compress it before writing it to disk (indicated by the shaded area in Figure 3-5). **Each entry of the sparse in-memory index then points at the start of a compressed block**. Besides saving disk space, compression also reduces the I/O bandwidth use.

## Constructing and Maintaining SSTables

how do you get your data to be sorted by key in the first place? Our incoming writes can occur in any order.

Maintaining a sorted structure on disk is possible (see “B-Trees” on page 79), but maintaining it in memory is much easier. There are plenty of well-known tree data structures that you can use, such as red-black trees or AVL trees(These two are self-balancing trees) [2]. With these data structures, you can insert keys in any order and read them back in sorted order.

We can now make our storage engine work as follows:

* When a write comes in, add it to **an in-memory balanced tree data structure** (for example, a red-black tree). This in-memory tree is sometimes called a *memtable*.
* When the memtable gets bigger than some threshold—typically a few megabytes —write it out to disk as an SSTable file. This can be done efficiently because the tree already maintains the key-value pairs sorted by key. The new SSTable file becomes the most recent segment of the database. While the SSTable is being written out to disk, writes can continue to a new memtable instance.
* In order to serve a read request, first try to find the key in the memtable, then in the most recent on-disk segment, then in the next-older segment, etc.
* From time to time, run a merging and compaction process in the background to combine segment files and to discard overwritten or deleted values.

This scheme works very well. It only suffers from **one problem: if the database crashes, the most recent writes (which are in the memtable but not yet written out to disk) are lost**. In order to avoid that problem, we can **keep a separate log on disk** to which every write is immediately appended, just like in the previous section. That log is not in sorted order, but that doesn’t matter, because **its only purpose is to restore the memtable after a crash.** Every time the **memtable is written out to an SSTable, the corresponding log can be discarded.**

## Making an LSM-Tree out of an SSTables

The algorithm described here is essentially what is used in **LevelDB [6] and RocksDB** [7], **key-value storage engine libraries** that are designed to be embedded into other applications. Among other things, LevelDB can be used in Riak as an alternative to Bitcask. Similar storage engines are used in **Cassandra and HBase** [8], both of which were inspired by Google’s Bigtable paper [9] (which introduced the terms *SSTable* and *memtable*).

Originally this indexing structure was described by Patrick O’Neil et al. under the name *Log-Structured Merge-Tree* (or LSM-Tree) [10], building on earlier work on log structured filesystems [11]. Storage engines that are based on this principle of merging and compacting sorted files are often called LSM storage engines.

Lucene, an indexing engine for full-text search used by Elasticsearch and Solr, uses a similar method for storing its *term dictionary* [12, 13]. A full-text index is much more complex than a key-value index but is based on a similar idea: given a word in a search query, find all the documents (web pages, product descriptions, etc.) that mention the word. This is implemented with a key-value structure where the key is a word (a *term*) and the value is the list of IDs of all the documents that contain the word (the *postings list*). In Lucene, this mapping from term to postings list is kept in SSTable-like sorted files, which are merged in the background as needed [14].

## Performance Optimizations

As always, a lot of detail goes into making a storage engine perform well in practice:

* For example, the LSM-tree algorithm can be slow when looking up keys that do not exist in the database: you have to check the memtable, then the segments all the way back to the oldest (possibly having to read from disk for each one) before you can be sure that the key does not exist. In order to optimize this kind of access, storage engines often use additional *Bloom filters* [15]. (A Bloom filter is a memory-efficient data structure for approximating the contents of a set. It can tell you if a key does not appear in the database, and thus saves many unnecessary disk reads for nonexistent keys.)
* There are also different strategies to determine the order and timing of how SSTables are compacted and merged. The most common options are *size-tiered* and *leveled* compaction. LevelDB and RocksDB use leveled compaction (hence the name of LevelDB), HBase uses size-tiered, and Cassandra supports both [16]. In size-tiered compaction, newer and smaller SSTables are successively merged into older and larger SSTables. In leveled compaction, the key range is split up into smaller SSTables and older data is moved into separate “levels,” which allows the compaction to proceed more incrementally and use less disk space.

Even though there are many subtleties, the basic idea of LSM-trees—keeping a cascade of SSTables that are merged in the background—is simple and effective. Even when the dataset is much bigger than the available memory it continues to work well. Since data is stored in sorted order, you can efficiently perform range queries (scanning all keys above some minimum and up to some maximum), and because the disk writes are sequential the LSM-tree can support remarkably high write throughput.

# B-Trees