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. Similarly 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:

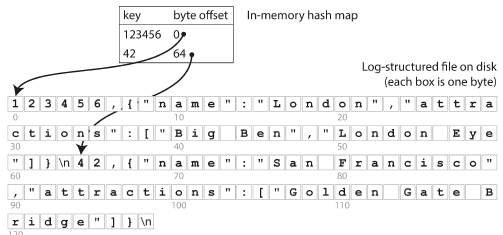
$ cat database

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.

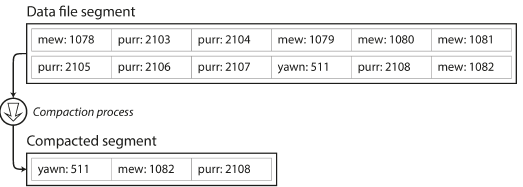
A storage engine like Bitcask is well suited to situations where the value for each key is updated frequently. For example, the key might be the URL of a cat video, and the value might be the number of times it has been played (incremented every time someone hits the play button). In this kind of workload**, 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

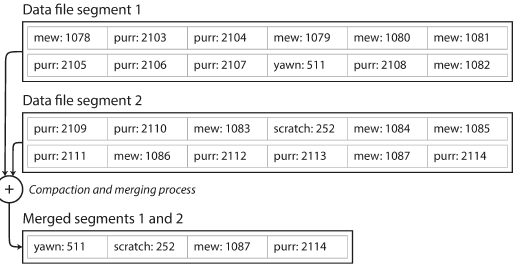
writes to a new segment file. We can then perform ***compaction* on these segments**. Compaction means throwing away duplicate keys in the log, and keeping only the most recent update for each key.



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