Chapter 4. Storage and Retrieval

> *One of the miseries of life is that everybody names things a little bit wrong. And so it makes everything a little harder to understand in the world than it would be if it were named differently. A computer does not primarily compute in the sense of doing arithmetic. […] They primarily are filing systems.*
>
> —Richard Feynman, *Idiosyncratic Thinking* seminar (1985)

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 4th chapter of the final book. The GitHub repo for this book is *https://github.com/ept/ddia2-feedback*.

If you'd like to be actively involved in reviewing and commenting on this draft, please reach out on GitHub.

On the most fundamental level, a database needs to do two things: when you give it some data, it should store the data, and when you ask it again later, it should give the data back to you.

In Chapter 3 we discussed data models and query languages—i.e., the format in which you give the database your data, and the interface through which you can ask for it again later. In this chapter we discuss the same from the database's point of view: how the database can store the data that you give it, and how it can find the data again when you ask for it.

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 configure 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.

In particular, there is a big difference between storage engines that are optimized for transactional workloads (OLTP) and those that are optimized for analytics (we introduced this distinction in "Operational Versus Analytical Systems"). This chapter starts by examining two families of storage engines for OLTP: *log-structured* storage engines that write out immutable data files, and storage engines such as *B-trees* that update data in-place. These structures are used for both key-value storage as well as secondary indexes.

Later in "Data Storage for Analytics" we'll discuss a family of storage engines that is optimized for analytics, and in "Multidimensional and Full-Text Indexes" we'll briefly look at indexes for more advanced queries, such as text retrieval.

Storage and Indexing for OLTP

Consider the world's simplest database, implemented as two Bash functions:

```
#!/bin/bash

db_set () {
    echo "$1,$2" >> database
}

db_get () {
    grep "^$1," database | sed -e "s/^$1,//" | tail -n
}
```

These 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.

And it works:

```
$ db_set 12 '{"name":"London","attractions":["Big Ben",

$ db_set 42 '{"name":"San Francisco","attractions":["Gc

$ db_get 42
{"name":"San Francisco","attractions":["Golden Gate Bri
```

The 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. If you update a key several times, 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 (hence the `tail -n 1` in `db_get`):

```
$ db_set 42 '{"name":"San Francisco","attractions":["Ex

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

$ cat database
12,{"name":"London","attractions":["Big Ben","London Ey
42,{"name":"San Francisco","attractions":["Golden Gate
42,{"name":"San Francisco","attractions":["Exploratoriu
```

The `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 handling concurrent writes, reclaiming disk space so that the log doesn't grow forever, and handling partially written records when recovering from a crash), but the basic principle is the same. Logs are incredibly useful, and we will encounter them several times in this book.

NOTE

The word *log* is often used to refer to application logs, where an application outputs text that describes what's happening. In this book, *log* is used in the more general sense: an append-only sequence of records on disk. It doesn't have to be human-readable; it might be binary and intended only for internal use by the database system.

On the other hand, the `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*. In this chapter we will look at a range of indexing structures and see how they compare; the general idea is to structure the data in a particular way (e.g., sorted by some key) that makes it faster to locate the data you want. If you want to search the same data in several different ways, you may need several different indexes on different parts of the data.

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 consumes additional disk space and slows down writes, sometimes substantially [1]. For this reason, databases don't usually index everything by default, but require you—the person writing the application or administering the database—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 on writes than necessary.

Log-Structured Storage

To start, let's assume that you want to continue storing data in the append-only file written by `db_set`, and you just want to speed up reads. One way you could do this is by keeping a hash map in memory, in which every key is mapped to the byte offset in the file at which the most recent value for that key can be found, as illustrated in Figure 4-1.

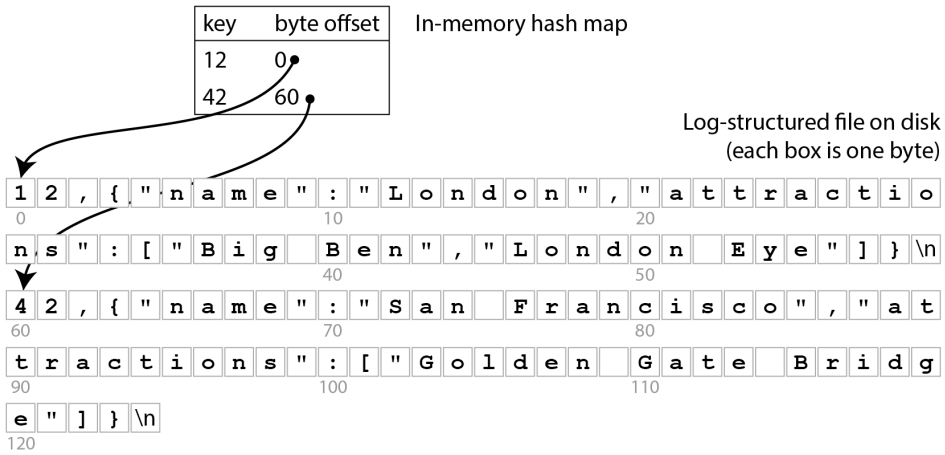

Figure 4-1. Storing a log of key-value pairs in a CSV-like format, indexed with an in-memory hash map.

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. When you want to look up a value, you use the hash map to find the offset in the log file, seek to that location, and read the value. If that part of the data file is already in the filesystem cache, a read doesn't require any disk I/O at all.

This approach is much faster, but it still suffers from several problems:

- You never free up disk space occupied by old log entries that have been overwritten; if you keep writing to the database you might run out of disk space.
- The hash map is not persisted, so you have to rebuild it when you restart the database—for example, by scanning the whole log file to find the latest byte offset for each key. This makes restarts slow if you have a lot of data.
- The hash table must fit in memory. In principle, you could maintain a hash table 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 [2].
- Range queries are not efficient. For example, you cannot easily scan over all keys between `10000` and `19999`—you'd have to look up each key

individually in the hash map.

The SSTable file format

In practice, hash tables are not used very often for database indexes, and instead it is much more common to keep data in a structure that is *sorted by key* [3]. One example of such a structure is a *Sorted String Table*, or *SSTable* for short, as shown in Figure 4-2. This file format also stores key-value pairs, but it ensures that they are sorted by key, and each key only appears once in the file.

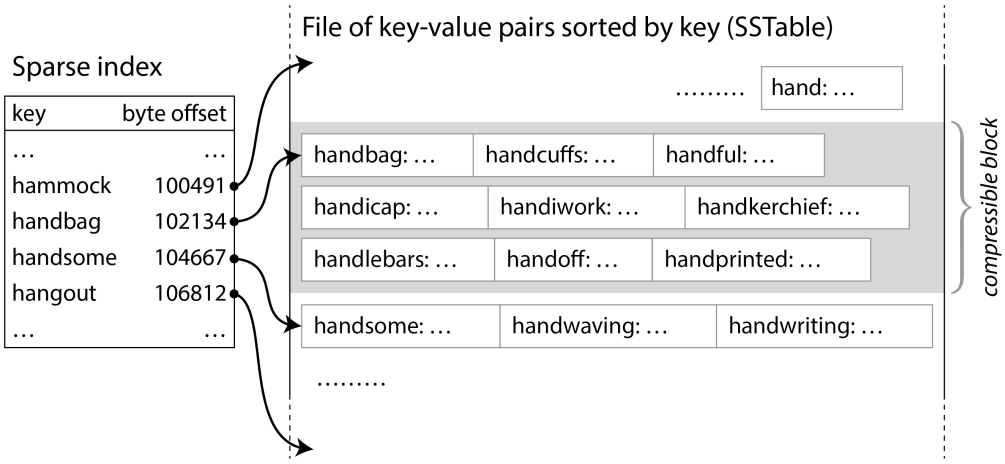

Figure 4-2. An SSTable with a sparse index, allowing queries to jump to the right block.

Now you do not need to keep all the keys in memory: you can group the key-value pairs within an SSTable into *blocks* of a few kilobytes, and then store the first key of each block in the index. This kind of index, which stores only some of the keys, is called *sparse*. This index is stored in a separate part of the SSTable, for example using an immutable B-tree, a trie, or another data structure that allows queries to quickly look up a particular key [4].

For example, in Figure 4-2, the first key of one block is `handbag`, and the first key of the next block is `handsome`. Now say you're looking for the key `handiwork`, which doesn't appear in the sparse index. Because of the sorting you know that `handiwork` must appear between `handbag` and `handsome`. This means you can seek to the offset for `handbag` and scan the file from there until you find `handiwork` (or not, if the key is not present in the file). A block of a few kilobytes can be scanned very quickly.

Moreover, each block of records can be compressed (indicated by the shaded area in Figure 4-2). Besides saving disk space, compression also reduces the I/O bandwidth use, at the cost of using a bit more CPU time.

Constructing and merging SSTables

The SSTable file format is better for reading than an append-only log, but it makes writes more difficult. We can't simply append at the end, because then the file would no longer be sorted (unless the keys happen to be written in ascending order). If we had to rewrite the whole SSTable every time a key is inserted somewhere in the middle, writes would become far too expensive.

We can solve this problem with a *log-structured* approach, which is a hybrid between an append-only log and a sorted file:

1. When a write comes in, add it to an in-memory ordered map data structure, such as a red-black tree, skip list [5], or trie [6]. With these data structures, you can insert keys in any order, look them up efficiently, and read them back in sorted order. This in-memory data structure is called the *memtable*.
2. When the memtable gets bigger than some threshold—typically a few megabytes—write it out to disk in sorted order as an SSTable file. We call this new SSTable file the most recent *segment* of the database, and it is stored as a separate file alongside the older segments. Each segment has a separate index of its contents. While the new segment is being written out to disk, the database can continue writing to a new memtable instance, and the old memtable's memory is freed when the writing of the SSTable is complete.
3. In order to read the value for some key, first try to find the key in the memtable and the most recent on-disk segment. If it's not there, look in the next-older segment, etc. until you either find the key or reach the oldest segment. If the key does not appear in any of the segments, it does not exist in the database.
4. From time to time, run a merging and compaction process in the background to combine segment files and to discard overwritten or deleted values.

Merging segments works similarly to the *mergesort* algorithm [5]. The process is illustrated in Figure 4-3: 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. If the same key appears in more than one input file, keep only the more recent value. This produces a new merged segment file, also sorted by key, with one value per key, and it uses minimal memory because we can iterate over the SSTables one key at a time.

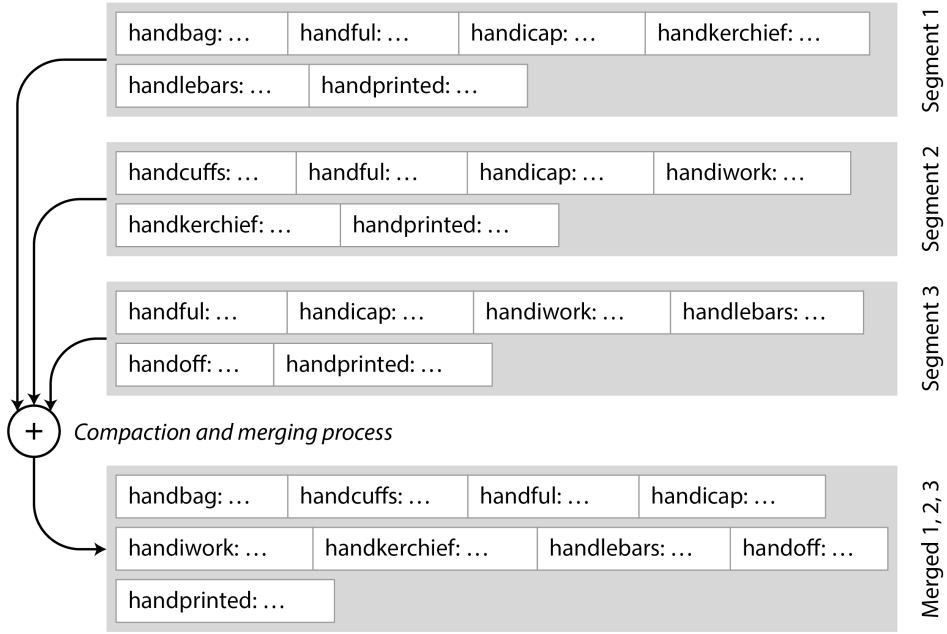

Figure 4-3. Merging several SSTable segments, retaining only the most recent value for each key.

To ensure that the data in the memtable is not lost if the database crashes, the storage engine keeps a separate log on disk to which every write is immediately appended. This log is not sorted by key, but that doesn't matter, because its only purpose is to restore the memtable after a crash. Every time the memtable has been written out to an SSTable, the corresponding part of the log can be discarded.

If you want to delete a key and its associated value, you have to append a special deletion record called a *tombstone* to the data file. When log segments are merged, the tombstone tells the merging process to discard any previous values for the deleted key. Once the tombstone is merged into the oldest segment, it can be dropped.

The algorithm described here is essentially what is used in RocksDB [7], Cassandra, Scylla, and HBase [8], all of which were inspired by Google's Bigtable paper [9] (which introduced the terms *SSTable* and *memtable*).

The algorithm was originally published in 1996 under the name *Log-Structured Merge-Tree* or *LSM-Tree* [10], building on earlier work on log-structured filesystems [11]. For this reason, storage engines that are based on the principle of merging and compacting sorted files are often called *LSM storage engines*.

In LSM storage engines, a segment file is written in one pass (either by writing out the memtable or by merging some existing segments), and thereafter it is immutable. The merging and compaction of segments can be

done in a background thread. While the merge is going on, we can still continue to serve reads using the input segments of the merge (as before, reads first look in the memtable and more recent segment files). When the merging process is complete, we switch read requests to using the new merged segment instead of the input segments, and then the input segment files can be deleted.

The segment files don't necessarily have to be stored on local disk: they are also well suited for writing to object storage. SlateDB and Delta Lake [12] take this approach, for example.

Having immutable segment files also simplifies crash recovery: if a crash happens while writing out the memtable or while merging segments, the database can just delete the unfinished SSTable and start afresh. The log that persists writes to the memtable could contain incomplete records if there was a crash halfway through writing a record, or if the disk was full; these are typically detected by including checksums in the log, and discarding corrupted or incomplete log entries. We will talk more about durability and crash recovery in Chapter 8.

Bloom filters

With LSM storage it can be slow to read a key that was last updated a long time ago, or that does not exist, since the storage engine needs to check several segment files. In order to speed up such reads, LSM storage engines often include a *Bloom filter* [13] in each segment, which provides a fast but approximate way of checking whether a particular key appears in a particular SSTable.

Figure 4-4 shows an example of a Bloom filter containing two keys and 16 bits (in reality, it would contain more keys and more bits). For every key in the SSTable we compute a hash function, producing a set of numbers that are then interpreted as indexes into the array of bits [14]. We set the bits corresponding to those indexes to 1, and leave the rest as 0. For example, the key `handbag` hashes to the numbers (2, 9, 4), so we set the 2nd, 9th, and 4th bits to 1. The bitmap is then stored as part of the SSTable, along with the sparse index of keys. This takes a bit of extra space, but the Bloom filter is generally small compared to the rest of the SSTable.

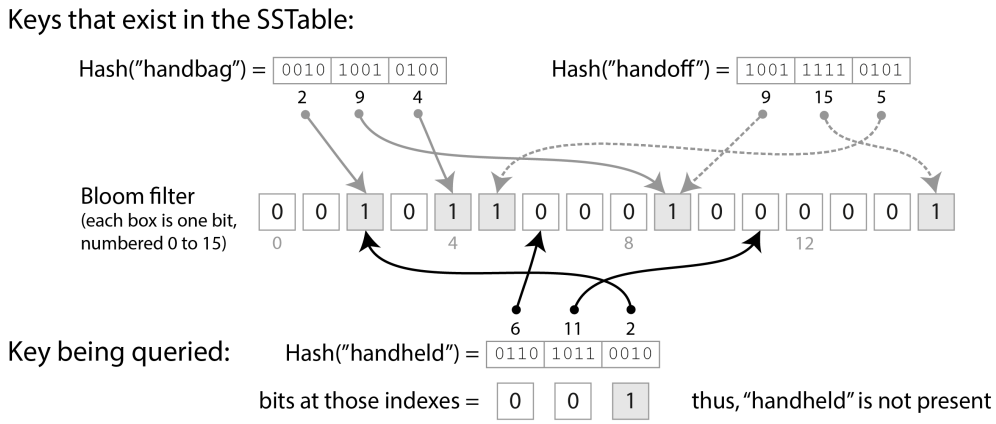

Figure 4-4. A Bloom filter provides a fast, probabilistic check whether a particular key exists in a particular SSTable.

When we want to know whether a key appears in the SSTable, we compute the same hash of that key as before, and check the bits at those indexes. For example, in Figure 4-4, we're querying the key `handheld`, which hashes to (6, 11, 2). One of those bits is 1 (namely, bit number 2), while the other two are 0. These checks can be made extremely fast using the bitwise operations that all CPUs support.

If at least one of the bits is 0, we know that the key definitely does not appear in the SSTable. If the bits in the query are all 1, it's likely that the key is in the SSTable, but it's also possible that by coincidence all of those bits were set to 1 by other keys. This case when it looks as if a key is present, even though it isn't, is called a *false positive*.

The probability of false positives depends on the number of keys, the number of bits set per key, and the total number of bits in the Bloom filter. You can use an online calculator tool to work out the right parameters for your application [15]. As a rule of thumb, you need to allocate 10 bits of Bloom filter space for every key in the SSTable to get a false positive probability of 1%, and the probability is reduced tenfold for every 5 additional bits you allocate per key.

In the context of an LSM storage engines, false positives are no problem:

- If the Bloom filter says that a key *is not* present, we can safely skip that SSTable, since we can be sure that it doesn't contain the key.
- If the Bloom filter says the key *is* present, we have to consult the sparse index and decode the block of key-value pairs to check whether the key really is there. If it was a false positive, we have done a bit of unnecessary work, but otherwise no harm is done—we just continue the search with the next-oldest segment.

Compaction strategies

An important detail is how the LSM storage chooses when to perform compaction, and which SSTables to include in a compaction. Many LSM-based storage systems allow you to configure which compaction strategy to use, and some of the common choices are [16, 17]:

Size-tiered compaction

Newer and smaller SSTables are successively merged into older and larger SSTables. For example, four 256 MB SSTables might be compacted into one 898 MB SSTable (the result is not 1024 MB due to deletions, overwrites, time-to-live (TTL) expirations, and so on). The SSTables containing older data can get very large, and merging them requires a lot of temporary disk space. The advantage of this strategy is that it can handle very high write throughput since most data is rewritten only a few times in larger sequential merges.

Leveled compaction

Instead of writing large SSTables, leveled compaction keeps SSTable sizes fixed and groups them into increasing "levels" (referred to as L0, L1, and so on). L0 contains the most recently written data. All levels beyond L0 contain key-range partitioned SSTables. For example, L1 might have two SSTables: the first with keys a-m and the second with n-z. Each level has its own size limit, and each level is larger than the level that preceeds it (e.g. L2 will be larger than L1). When a level's SSTables combine to exceed a maximum size limit, one or more SSTables from level i are merged into level i+1 and deleted from level i. This approach allows compaction to proceed more incrementally and use less disk space than the size-tiered strategy. Leveled compaction is more efficient for reads than size-tiered compaction because the storage engine needs to read fewer SSTables to check whether they contain the key.

As a rule of thumb, size-tiered compaction performs better if you have mostly writes and few reads, whereas leveled compaction performs better if your workload is dominated by reads. If you write a small number of keys frequently and a large number of keys rarely, then leveled compaction can also be advantageous [18]. Fortunately, most LSM-tree implementations provide a variety of compaction strategies for different workloads.

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. We discuss their performance characteristics in more detail in "Comparing B-Trees and LSM-Trees".

EMBEDDED STORAGE ENGINES

Many databases run as a service that accepts queries over a network, but there are also *embedded* databases that don't expose a network API. Instead, they are libraries that run in the same process as your application code, typically reading and writing files on the local disk, and you interact with them through normal function calls. Examples of embedded storage engines include RocksDB, SQLite, LMDB, DuckDB, and KùzuDB [19].

Embedded databases are very commonly used in mobile apps to store the local user's data. On the backend, they can be an appropriate choice if the data is small enough to fit on a single machine, and if there are not many concurrent transactions. For example, in a multitenant system in which each tenant is small enough and completely separate from others (i.e., you do not need to run queries that combine data from multiple tenants), you can potentially use a separate embedded database instance per tenant [20].

The storage and retrieval methods we discuss in this chapter are used in both embedded and in client-server databases. In Chapter 6 and Chapter 7 we will discuss techniques for scaling a database across multiple machines.

B-Trees

The log-structured approach is popular, but it is not the only form of key-value storage. The most widely used structure for reading and writing database records by key is the *B-tree*.

Introduced in 1970 [21] and called "ubiquitous" less than 10 years later [22], B-trees have stood the test of time very well. They remain the standard index implementation in almost all relational databases, and many nonrelational databases use them too.

Like SSTables, B-trees keep key-value pairs sorted by key, which allows efficient key-value lookups and range queries. But that's where the similarity

ends: B-trees have a very different design philosophy.

The log-structured indexes we saw earlier break the database down into variable-size *segments*, typically several megabytes or more in size, that are written once and are then immutable. By contrast, B-trees break the database down into fixed-size *blocks* or *pages*, and may overwrite a page in-place. A page is traditionally 4 KiB in size, but PostgreSQL now uses 8 KiB and MySQL uses 16 KiB by default.

Each page can be identified using a page number, which allows one page to refer to another—similar to a pointer, but on disk instead of in memory. If all the pages are stored in the same file, multiplying the page number by the page size gives us the byte offset in the file where the page is located. We can use these page references to construct a tree of pages, as illustrated in Figure 4-5.

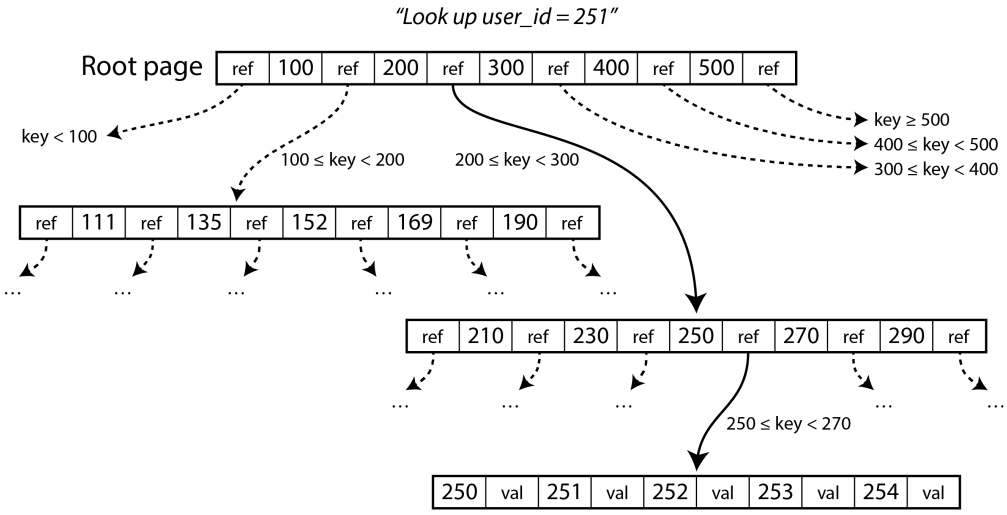

Figure 4-5. Looking up the key 251 using a B-tree index. From the root page we first follow the reference to the page for keys 200–300, then the page for keys 250–270.

One page is designated as the *root* of the B-tree; whenever you want to look up a key in the index, you start here. The page contains several keys and references to child pages. Each child is responsible for a continuous range of keys, and the keys between the references indicate where the boundaries between those ranges lie. (This structure is sometimes called a B+ tree, but we don't need to distinguish it from other B-tree variants.)

In the example in Figure 4-5, we are looking for the key 251, so we know that we need to follow the page reference between the boundaries 200 and 300. That takes us to a similar-looking page that further breaks down the 200–300 range into subranges. Eventually we get down to a page containing individual

keys (a *leaf page*), which either contains the value for each key inline or contains references to the pages where the values can be found.

The number of references to child pages in one page of the B-tree is called the *branching factor*. For example, in Figure 4-5 the branching factor is six. In practice, the branching factor depends on the amount of space required to store the page references and the range boundaries, but typically it is several hundred.

If you want to update the value for an existing key in a B-tree, you search for the leaf page containing that key, and overwrite that page on disk with a version that contains the new value. If you want to add a new key, you need to find the page whose range encompasses the new key and add it to that page. If there isn't enough free space in the page to accommodate the new key, the page is split into two half-full pages, and the parent page is updated to account for the new subdivision of key ranges.

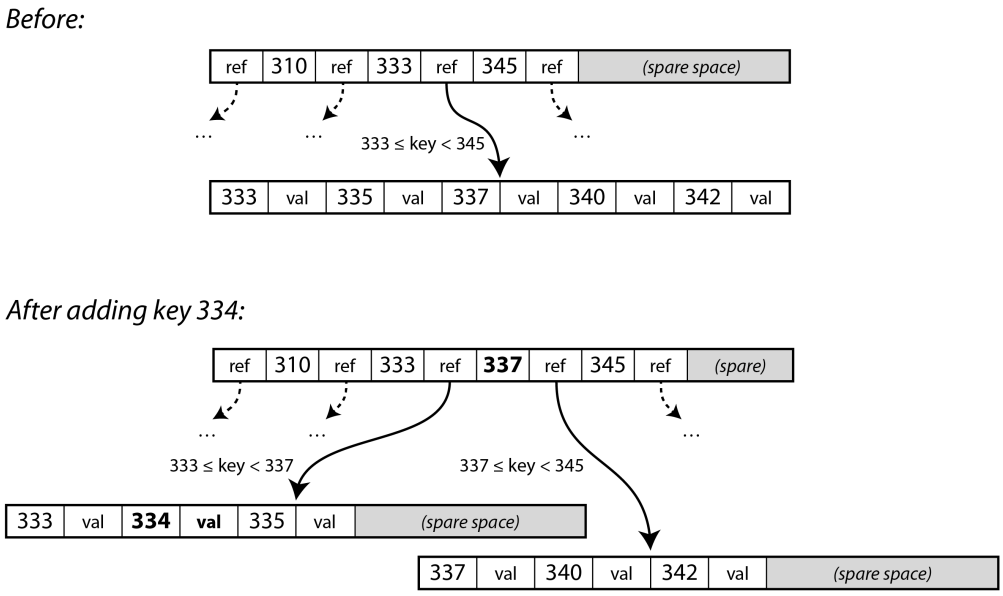

Figure 4-6. Growing a B-tree by splitting a page on the boundary key 337. The parent page is updated to reference both children.

In the example of Figure 4-6, we want to insert the key 334, but the page for the range 333–345 is already full. We therefore split it into a page for the range 333–337 (including the new key), and a page for 337–344. We also have to update the parent page to have references to both children, with a boundary value of 337 between them. If the parent page doesn't have enough space for the new reference, it may also need to be split, and the splits can continue all the way to the root of the tree. When the root is split, we make a new root above it. Deleting keys (which may require nodes to be merged) is more complex [5].

This algorithm ensures that the tree remains *balanced*: a B-tree with n keys always has a depth of $O(\log n)$. Most databases can fit into a B-tree that is three or four levels deep, so you don't need to follow many page references to find the page you are looking for. (A four-level tree of 4 KiB pages with a branching factor of 500 can store up to 250 TB.)

Making B-trees reliable

The basic underlying write operation of a B-tree is to overwrite a page on disk with new data. It is assumed that the overwrite does not change the location of the page; i.e., all references to that page remain intact when the page is overwritten. This is in stark contrast to log-structured indexes such as LSM-trees, which only append to files (and eventually delete obsolete files) but never modify files in place.

Overwriting several pages at once, like in a page split, is a dangerous operation: if the database crashes after only some of the pages have been written, you end up with a corrupted tree (e.g., there may be an *orphan* page that is not a child of any parent). If the hardware can't atomically write an entire page, you can also end up with a partially written page (this is known as a *torn page* [23]).

In order to make the database resilient to crashes, it is common for B-tree implementations to include an additional data structure on disk: a *write-ahead log* (WAL). This is an append-only file to which every B-tree modification must be written before it can be applied to the pages of the tree itself. When the database comes back up after a crash, this log is used to restore the B-tree back to a consistent state [2, 24]. In filesystems, the equivalent mechanism is known as *journaling*.

To improve performance, B-tree implementations typically don't immediately write every modified page to disk, but buffer the B-tree pages in memory for a while first. The write-ahead log then also ensures that data is not lost in the case of a crash: as long as data has been written to the WAL, and flushed to disk using the `fsync()` system call, the data will be durable as the database will be able to recover it after a crash [25].

B-tree variants

As B-trees have been around for so long, many variants have been developed over the years. To mention just a few:

- Instead of overwriting pages and maintaining a WAL for crash recovery, some databases (like LMDB) use a copy-on-write scheme [26]. A modified page is written to a different location, and a new version of the parent pages in the tree is created, pointing at the new location. This approach is also useful for concurrency control, as we shall see in "Snapshot Isolation and Repeatable Read".
- We can save space in pages by not storing the entire key, but abbreviating it. Especially in pages on the interior of the tree, keys only need to provide enough information to act as boundaries between key ranges. Packing more keys into a page allows the tree to have a higher branching factor, and thus fewer levels.
- To speed up scans over the key range in sorted order, some B-tree implementations try to lay out the tree so that leaf pages appear in sequential order on disk, reducing the number of disk seeks. However, it's difficult to maintain that order as the tree grows.
- Additional pointers have been added to the tree. For example, each leaf page may have references to its sibling pages to the left and right, which allows scanning keys in order without jumping back to parent pages.

Comparing B-Trees and LSM-Trees

As a rule of thumb, LSM-trees are better suited for write-heavy applications, whereas B-trees are faster for reads [27, 28]. However, benchmarks are often sensitive to details of the workload. You need to test systems with your particular workload in order to make a valid comparison. Moreover, it's not a strict either/or choice between LSM and B-trees: storage engines sometimes blend characteristics of both approaches, for example by having multiple B-trees and merging them LSM-style. In this section we will briefly discuss a few things that are worth considering when measuring the performance of a storage engine.

Read performance

In a B-tree, looking up a key involves reading one page at each level of the B-tree. Since the number of levels is usually quite small, this means that reads

from a B-tree are generally fast and have predictable performance. In an LSM storage engine, reads often have to check several different SSTables at different stages of compaction, but Bloom filters help reduce the number of actual disk I/O operations required. Both approaches can perform well, and which is faster depends on the details of the storage engine and the workload.

Range queries are simple and fast on B-trees, as they can use the sorted structure of the tree. On LSM storage, range queries can also take advantage of the SSTable sorting, but they need to scan all the segments in parallel and combine the results. Bloom filters don't help for range queries (since you would need to compute the hash of every possible key within the range, which is impractical), making range queries more expensive than point queries in the LSM approach [29].

High write throughput can cause latency spikes in a log-structured storage engine if the memtable fills up. This happens if data can't be written out to disk fast enough, perhaps because the compaction process cannot keep up with incoming writes. Many storage engines, including RocksDB, perform *backpressure* in this situation: they suspend all reads and writes until the memtable has been written out to disk [30, 31].

Regarding read throughput, modern SSDs (and especially NVMe SSDs that connect through the much faster Peripheral Component Interconnect Express (PCIe) bus rather than the SATA bus) can perform many independent read requests in parallel. Both LSM-trees and B-trees are able to provide high read throughput, but storage engines need to be carefully designed to take advantage of this parallelism [32].

Sequential versus random writes

With a B-tree, if the application writes keys that are scattered all over the key space, the resulting disk operations are also scattered randomly, since the pages that the storage engine needs to overwrite could be located anywhere on disk. On the other hand, a log-structured storage engine writes entire segment files at a time (either writing out the memtable or while compacting existing segments), which are much bigger than a page in a B-tree.

The pattern of many small, scattered writes (as found in B-trees) is called *random writes*, while the pattern of fewer large writes (as found in LSM-trees) is called *sequential writes*. Disks generally have higher sequential write

throughput than random write throughput, which means that a log-structured storage engine can generally handle higher write throughput on the same hardware than a B-tree. This difference is particularly big on spinning-disk hard drives (HDDs); on the solid state drives (SSDs) that most databases use today, the difference is smaller, but still noticeable (see "Sequential Versus Random Writes on SSDs").

SEQUENTIAL VERSUS RANDOM WRITES ON SSDS

On spinning-disk hard drives (HDDs), sequential writes are much faster than random writes: a random write has to mechanically move the disk head to a new position and wait for the right part of the platter to pass underneath the disk head, which takes several milliseconds—an eternity in computing timescales. However, SSDs (solid-state drives) including NVMe (Non-Volatile Memory Express, i.e. flash memory attached to the PCI Express bus) have now overtaken HDDs for many use cases, and they are not subject to such mechanical limitations.

Nevertheless, SSDs also have higher throughput for sequential writes than for random writes. The reason is that flash memory can be read or written one page (typically 4 KiB) at a time, but it can only be erased one block (typically 512 KiB) at a time. Some of the pages in a block may contain valid data, whereas others may contain data that is no longer needed. Before erasing a block, the controller must first move pages containing valid data into other blocks; this process is called *garbage collection* (GC) [33].

A sequential write workload writes larger chunks of data at a time, so it is likely that a whole 512 KiB block belongs to a single file; when that file is later deleted again, the whole block can be erased without having to perform any GC. On the other hand, with a random write workload, it is more likely that a block contains a mixture of pages with valid and invalid data, so the GC has to perform more work before a block can be erased [34, 35, 36].

The write bandwidth consumed by GC is then not available for the application. Moreover, the additional writes performed by GC contribute to wear on the flash memory; therefore, random writes wear out the drive faster than sequential writes.

Write amplification

With any type of storage engine, one write request from the application turns into multiple I/O operations on the underlying disk. With LSM-trees, a value is first written to the log for durability, then again when the memtable is written to disk, and again every time the key-value pair is part of a compaction. (If the values are significantly larger than the keys, this overhead can be reduced by storing values separately from keys, and performing compaction only on SSTables containing keys and references to values [37].)

A B-tree index must write every piece of data at least twice: once to the write-ahead log, and once to the tree page itself. In addition, they sometimes need to write out an entire page, even if only a few bytes in that page changed, to ensure the B-tree can be correctly recovered after a crash or power failure [38, 39].

If you take the total number of bytes written to disk in some workload, and divide by the number of bytes you would have to write if you simply wrote an append-only log with no index, you get the *write amplification*. (Sometimes write amplification is defined in terms of I/O operations rather than bytes.) In write-heavy applications, the bottleneck might be the rate at which the database can write to disk. In this case, the higher the write amplification, the fewer writes per second it can handle within the available disk bandwidth.

Write amplification is a problem in both LSM-trees and B-trees. Which one is better depends on various factors, such as the length of your keys and values, and how often you overwrite existing keys versus insert new ones. For typical workloads, LSM-trees tend to have lower write amplification because they don’t have to write entire pages and they can compress chunks of the SSTable [40]. This is another factor that makes LSM storage engines well suited for write-heavy workloads.

Besides affecting throughput, write amplification is also relevant for the wear on SSDs: a storage engine with lower write amplification will wear out the SSD less quickly.

When measuring the write throughput of a storage engine, it is important to run the experiment for long enough that the effects of write amplification become clear. When writing to an empty LSM-tree, there are no compactions going on yet, so all of the disk bandwidth is available for new writes. As the

database grows, new writes need to share the disk bandwidth with compaction.

Disk space usage

B-trees can become *fragmented* over time: for example, if a large number of keys are deleted, the database file may contain a lot of pages that are no longer used by the B-tree. Subsequent additions to the B-tree can use those free pages, but they can't easily be returned to the operating system because they are in the middle of the file, so they still take up space on the filesystem. Databases therefore need a background process that moves pages around to place them better, such as the vacuum process in PostgreSQL [25].

Fragmentation is less of a problem in LSM-trees, since the compaction process periodically rewrites the data files anyway, and SSTables don't have pages with unused space. Moreover, blocks of key-value pairs can better be compressed in SSTables, and thus often produce smaller files on disk than B-trees. Keys and values that have been overwritten continue to consume space until they are removed by a compaction, but this overhead is quite low when using leveled compaction [40, 41]. Size-tiered compaction (see "Compaction strategies") uses more disk space, especially temporarily during compaction.

Having multiple copies of some data on disk can also be a problem when you need to delete some data, and be confident that it really has been deleted (perhaps to comply with data protection regulations). For example, in most LSM storage engines a deleted record may still exist in the higher levels until the tombstone representing the deletion has been propagated through all of the compaction levels, which may take a long time. Specialist storage engine designs can propagate deletions faster [42].

On the other hand, the immutable nature of SSTable segment files is useful if you want to take a snapshot of a database at some point in time (e.g. for a backup or to create a copy of the database for testing): you can write out the memtable and record which segment files existed at that point in time. As long as you don't delete the files that are part of the snapshot, you don't need to actually copy them. In a B-tree whose pages are overwritten, taking such a snapshot efficiently is more difficult.

Multi-Column and Secondary Indexes

So far we have only discussed key-value indexes, which are like a *primary key* index in the relational model. A primary key uniquely identifies one row in a relational table, or one document in a document database, or one vertex in a graph database. Other records in the database can refer to that row/document/vertex by its primary key (or ID), and the index is used to resolve such references.

It is also very common to have *secondary indexes*. In relational databases, you can create several secondary indexes on the same table using the `CREATE INDEX` command, allowing you to search by columns other than the primary key. For example, in Figure 3-1 in Chapter 3 you would most likely have a secondary index on the `user_id` columns so that you can find all the rows belonging to the same user in each of the tables.

A secondary index can easily be constructed from a key-value index. The main difference is that in a secondary index, the indexed values are not necessarily unique; that is, there might be many rows (documents, vertices) under the same index entry. This can be solved in two ways: either by making each value in the index a list of matching row identifiers (like a postings list in a full-text index) or by making each entry unique by appending a row identifier to it. Storage engines with in-place updates, like B-trees, and log-structured storage can both be used to implement an index.

Storing values within the index

The key in an index is the thing that queries search by, but the value can be one of several things:

- If the actual data (row, document, vertex) is stored directly within the index structure, it is called a *clustered index*. For example, in MySQL's InnoDB storage engine, the primary key of a table is always a clustered index, and in SQL Server, you can specify one clustered index per table [43].
- Alternatively, the value can be a reference to the actual data: either the primary key of the row in question (InnoDB does this for secondary indexes), or a direct reference to a location on disk. In the latter case, the place where rows are stored is known as a *heap file*, and it stores data in no particular order (it may be append-only, or it may keep track of deleted

rows in order to overwrite them with new data later). For example, Postgres uses the heap file approach [44].

- A middle ground between the two is a *covering index* or *index with included columns*, which stores *some* of a table's columns within the index, in addition to storing the full row on the heap or in the primary key clustered index [45]. This allows some queries to be answered by using the index alone, without having to resolve the primary key or look in the heap file (in which case, the index is said to *cover* the query). This can make some queries faster, but the duplication of data means the index uses more disk space and slows down writes.

The indexes discussed so far only map a single key to a value. If you need to query multiple columns of a table (or multiple fields in a document) simultaneously, see "Multidimensional and Full-Text Indexes".

When updating a value without changing the key, the heap file approach can allow the record to be overwritten in place, provided that the new value is not larger than the old value. The situation is more complicated if the new value is larger, as it probably needs to be moved to a new location in the heap where there is enough space. In that case, either all indexes need to be updated to point at the new heap location of the record, or a forwarding pointer is left behind in the old heap location [2].

Keeping everything in memory

The data structures discussed so far in this chapter have all been answers to the limitations of disks. Compared to main memory, disks are awkward to deal with. With both magnetic disks and SSDs, data on disk needs to be laid out carefully if you want good performance on reads and writes. However, we tolerate this awkwardness because disks have two significant advantages: they are durable (their contents are not lost if the power is turned off), and they have a lower cost per gigabyte than RAM.

As RAM becomes cheaper, the cost-per-gigabyte argument is eroded. Many datasets are simply not that big, so it's quite feasible to keep them entirely in memory, potentially distributed across several machines. This has led to the development of *in-memory databases*.

Some in-memory key-value stores, such as Memcached, are intended for caching use only, where it's acceptable for data to be lost if a machine is

restarted. But other in-memory databases aim for durability, which can be achieved with special hardware (such as battery-powered RAM), by writing a log of changes to disk, by writing periodic snapshots to disk, or by replicating the in-memory state to other machines.

When an in-memory database is restarted, it needs to reload its state, either from disk or over the network from a replica (unless special hardware is used). Despite writing to disk, it's still an in-memory database, because the disk is merely used as an append-only log for durability, and reads are served entirely from memory. Writing to disk also has operational advantages: files on disk can easily be backed up, inspected, and analyzed by external utilities.

Products such as VoltDB, SingleStore, and Oracle TimesTen are in-memory databases with a relational model, and the vendors claim that they can offer big performance improvements by removing all the overheads associated with managing on-disk data structures [46, 47]. RAMCloud is an open source, in-memory key-value store with durability (using a log-structured approach for the data in memory as well as the data on disk) [48]. Redis and Couchbase provide weak durability by writing to disk asynchronously.

Counterintuitively, the performance advantage of in-memory databases is not due to the fact that they don't need to read from disk. Even a disk-based storage engine may never need to read from disk if you have enough memory, because the operating system caches recently used disk blocks in memory anyway. Rather, they can be faster because they can avoid the overheads of encoding in-memory data structures in a form that can be written to disk [49].

Besides performance, another interesting area for in-memory databases is providing data models that are difficult to implement with disk-based indexes. For example, Redis offers a database-like interface to various data structures such as priority queues and sets. Because it keeps all data in memory, its implementation is comparatively simple.

Data Storage for Analytics

The data model of a data warehouse is most commonly relational, because SQL is generally a good fit for analytic queries. There are many graphical data analysis tools that generate SQL queries, visualize the results, and allow

analysts to explore the data (through operations such as *drill-down* and *slicing and dicing*).

On the surface, a data warehouse and a relational OLTP database look similar, because they both have a SQL query interface. However, the internals of the systems can look quite different, because they are optimized for very different query patterns. Many database vendors now focus on supporting either transaction processing or analytics workloads, but not both.

Some databases, such as Microsoft SQL Server, SAP HANA, and SingleStore, have support for transaction processing and data warehousing in the same product. However, these hybrid transactional and analytical processing (HTAP) databases (introduced in "Data Warehousing") are increasingly becoming two separate storage and query engines, which happen to be accessible through a common SQL interface [50, 51, 52, 53].

Cloud Data Warehouses

Data warehouse vendors such as Teradata, Vertica, and SAP HANA sell both on-premises warehouses under commercial licenses and cloud-based solutions. But as many of their customers move to the cloud, new cloud data warehouses such as Google Cloud BigQuery, Amazon Redshift, and Snowflake have also become widely adopted. Unlike traditional data warehouses, cloud data warehouses take advantage of scalable cloud infrastructure like object storage and serverless computation platforms.

Cloud data warehouses tend to integrate better with other cloud services and to be more elastic. For example, many cloud warehouses support automatic log ingestion, and offer easy integration with data processing frameworks such as Google Cloud's Dataflow or Amazon Web Services' Kinesis. These warehouses are also more elastic because they decouple query computation from the storage layer [54]. Data is persisted on object storage rather than local disks, which makes it easy to adjust storage capacity and compute resources for queries independently, as we previously saw in "Cloud-Native System Architecture".

Open source data warehouses such as Apache Hive, Trino, and Apache Spark have also evolved with the cloud. As data storage for analytics has moved to data lakes on object storage, open source warehouses have begun to break apart [55]. The following components, which were previously integrated in a

single system such as Apache Hive, are now often implemented as separate components:

Query engine

Query engines such as Trino, Apache DataFusion, and Presto parse SQL queries, optimize them into execution plans, and execute them against the data. Execution usually requires parallel, distributed data processing tasks. Some query engines provide built-in task execution, while others choose to use third party execution frameworks such as Apache Spark or Apache Flink.

Storage format

The storage format determines how the rows of a table are encoded as bytes in a file, which is then typically stored in object storage or a distributed filesystem [12]. This data can then be accessed by the query engine, but also by other applications using the data lake. Examples of such storage formats are Parquet, ORC, Lance, or Nimble, and we will see more about them in the next section.

Table format

Files written in Apache Parquet and similar storage formats are typically immutable once written. To support row inserts and deletions, a table format such as Apache Iceberg or Databricks's Delta format are used. Table formats specify a file format that defines which files constitute a table along with the table's schema. Such formats also offer advanced features such as time travel (the ability to query a table as it was at a previous point in time), garbage collection, and even transactions.

Data catalog

Much like a table format defines which files make up a table, a data catalog defines which tables comprise a database. Catalogs are used to create, rename, and drop tables. Unlike storage and table formats, data catalogs such as Snowflake's Polaris and Databricks's Unity Catalog usually run as a standalone service that can be queried using a REST interface. Apache Iceberg also offers a catalog, which can be run inside a client or as a separate process. Query engines use catalog information when reading and writing tables. Traditionally, catalogs

and query engines have been integrated, but decoupling them has enabled data discovery and data governance systems (discussed in "Data Systems, Law, and Society") to access a catalog's metadata as well.

Column-Oriented Storage

As discussed in "Stars and Snowflakes: Schemas for Analytics", data warehouses by convention often use a relational schema with a big fact table that contains foreign key references into dimension tables. If you have trillions of rows and petabytes of data in your fact tables, storing and querying them efficiently becomes a challenging problem. Dimension tables are usually much smaller (millions of rows), so in this section we will focus on storage of facts.

Although fact tables are often over 100 columns wide, a typical data warehouse query only accesses 4 or 5 of them at one time (`"SELECT *"` queries are rarely needed for analytics) [52]. Take the query in Example 4-1: it accesses a large number of rows (every occurrence of someone buying fruit or candy during the 2024 calendar year), but it only needs to access three columns of the `fact_sales` table: `date_key` , `product_sk` , and `quantity` . The query ignores all other columns.

Example 4-1. Analyzing whether people are more inclined to buy fresh fruit or candy, depending on the day of the week

```
SELECT
  dim_date.weekday, dim_product.category,
  SUM(fact_sales.quantity) AS quantity_sold
FROM fact_sales
  JOIN dim_date    ON fact_sales.date_key   = dim_date.
  JOIN dim_product ON fact_sales.product_sk = dim_produ
WHERE
  dim_date.year = 2024 AND
  dim_product.category IN ('Fresh fruit', 'Candy')
GROUP BY
  dim_date.weekday, dim_product.category;
```

How can we execute this query efficiently?

In most OLTP databases, storage is laid out in a *row-oriented* fashion: all the values from one row of a table are stored next to each other. Document databases are similar: an entire document is typically stored as one contiguous sequence of bytes. You can see this in the CSV example of Figure 4-1.

In order to process a query like Example 4-1, you may have indexes on `fact_sales.date_key` and/or `fact_sales.product_sk` that tell the storage engine where to find all the sales for a particular date or for a particular product. But then, a row-oriented storage engine still needs to load all of those rows (each consisting of over 100 attributes) from disk into memory, parse them, and filter out those that don't meet the required conditions. That can take a long time.

The idea behind *column-oriented* (or *columnar*) storage is simple: don't store all the values from one row together, but store all the values from each *column* together instead [56]. If each column is stored separately, a query only needs to read and parse those columns that are used in that query, which can save a lot of work. Figure 4-7 shows this principle using an expanded version of the fact table from Figure 3-5.

NOTE

Column storage is easiest to understand in a relational data model, but it applies equally to nonrelational data. For example, Parquet [57] is a columnar storage format that supports a document data model, based on Google's Dremel [58], using a technique known as *shredding* or *striping* [59].

---

fact_sales table

date_key	product_sk	store_sk	promotion_sk	customer_sk	quantity	net_price	discount_price
240102	69	4	NULL	NULL	1	13.99	13.99
240102	69	5	19	NULL	3	14.99	9.99
240102	69	5	NULL	191	1	14.99	14.99
240102	74	3	23	202	5	0.99	0.89
240103	30	2	NULL	NULL	1	2.49	2.49
240103	30	3	NULL	NULL	3	14.99	9.99
240103	30	3	21	123	1	49.99	39.99
240103	30	8	NULL	233	1	0.99	0.99

Columnar storage layout

date_key:	240102, 240102, 240102, 240102, 240103, 240103, 240103, 240103
product_sk:	69, 69, 69, 74, 30, 30, 30, 30
store_sk:	4, 5, 5, 3, 2, 3, 3, 8
promotion_sk:	NULL, 19, NULL, 23, NULL, NULL, 21, NULL
customer_sk:	NULL, NULL, 191, 202, NULL, NULL, 123, 233
quantity:	1, 3, 1, 5, 1, 3, 1, 1
net_price:	13.99, 14.99, 14.99, 0.99, 2.49, 14.99, 49.99, 0.99
discount_price:	13.99, 9.99, 14.99, 0.89, 2.49, 9.99, 39.99, 0.99

Figure 4-7. Storing relational data by column, rather than by row.

The column-oriented storage layout relies on each column storing the rows in the same order. Thus, if you need to reassemble an entire row, you can take the 23rd entry from each of the individual columns and put them together to form the 23rd row of the table.

In fact, columnar storage engines don't actually store an entire column (containing perhaps trillions of rows) in one go. Instead, they break the table into blocks of thousands or millions of rows, and within each block they store the values from each column separately [60]. Since many queries are restricted to a particular date range, it is common to make each block contain the rows for a particular timestamp range. A query then only needs to load the columns it needs in those blocks that overlap with the required date range.

Columnar storage is used in almost all analytic databases nowadays [60], ranging from large-scale cloud data warehouses such as Snowflake [61] to single-node embedded databases such as DuckDB [62], and product analytics systems such as Pinot [63] and Druid [64]. It is used in storage formats such as Parquet, ORC [65, 66], Lance [67], and Nimble [68], and in-memory analytics formats like Apache Arrow [65, 69] and Pandas/NumPy [70]. Some time-series databases, such as InfluxDB IOx [71] and TimescaleDB [72], are also based on column-oriented storage.

Column Compression

Besides only loading those columns from disk that are required for a query, we can further reduce the demands on disk throughput and network bandwidth

by compressing data. Fortunately, column-oriented storage often lends itself very well to compression.

Take a look at the sequences of values for each column in Figure 4-7: they often look quite repetitive, which is a good sign for compression. Depending on the data in the column, different compression techniques can be used. One technique that is particularly effective in data warehouses is *bitmap encoding*, illustrated in Figure 4-8.

Column values:

product_sk:	69	69	69	69	74	30	30	30	30	29	31	31	30	30	30	68	69	69

Bitmap for each possible value:

product_sk = 29:	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0
product_sk = 30:	0	0	0	0	0	1	1	1	1	0	0	0	1	1	1	0	0	0
product_sk = 31:	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0
product_sk = 68:	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0
product_sk = 69:	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	1	1
product_sk = 74:	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0

Run-length encoding:

product_sk = 29:	9, 1	(9 zeros, 1 one, rest zeros)
product_sk = 30:	5, 4, 3, 3	(5 zeros, 4 ones, 3 zeros, 3 ones, rest zeros)
product_sk = 31:	10, 2	(10 zeros, 2 ones, rest zeros)
product_sk = 68:	15, 1	(15 zeros, 1 one, rest zeros)
product_sk = 69:	0, 4, 12, 2	(0 zeros, 4 ones, 12 zeros, 2 ones)
product_sk = 74:	4, 1	(4 zeros, 1 one, rest zeros)

Figure 4-8. Compressed, bitmap-indexed storage of a single column.

Often, the number of distinct values in a column is small compared to the number of rows (for example, a retailer may have billions of sales transactions, but only 100,000 distinct products). We can now take a column with *n* distinct values and turn it into *n* separate bitmaps: one bitmap for each distinct value, with one bit for each row. The bit is 1 if the row has that value, and 0 if not.

One option is to store those bitmaps using one bit per row. However, these bitmaps typically contain a lot of zeros (we say that they are *sparse*). In that case, the bitmaps can additionally be run-length encoded: counting the number of consecutive zeros or ones and storing that number, as shown at the bottom of Figure 4-8. Techniques such as *roaring bitmaps* switch between the two bitmap representations, using whichever is the most compact [73]. This can make the encoding of a column remarkably efficient.

Bitmap indexes such as these are very well suited for the kinds of queries that are common in a data warehouse. For example:

`WHERE product_sk IN (31, 68, 69):`

Load the three bitmaps for `product_sk = 31`, `product_sk = 68`, and `product_sk = 69`, and calculate the bitwise *OR* of the three bitmaps, which can be done very efficiently.

`WHERE product_sk = 30 AND store_sk = 3:`

Load the bitmaps for `product_sk = 30` and `store_sk = 3`, and calculate the bitwise *AND*. This works because the columns contain the rows in the same order, so the *k*th bit in one column's bitmap corresponds to the same row as the *k*th bit in another column's bitmap.

Bitmaps can also be used to answer graph queries, such as finding all users of a social network who are followed by user *X* and who also follow user *Y* [74]. There are also various other compression schemes for columnar databases, which you can find in the references [75].

NOTE

Don't confuse column-oriented databases with the *wide-column* (also known as *column-family*) data model, in which a row can have thousands of columns, and there is no need for all the rows to have the same columns [9]. Despite the similarity in name, wide-column databases are row-oriented, since they store all values from a row together. Google's Bigtable, Apache Accumulo, and HBase are examples of the wide-column model.

Sort Order in Column Storage

In a column store, it doesn't necessarily matter in which order the rows are stored. It's easiest to store them in the order in which they were inserted, since then inserting a new row just means appending to each of the columns. However, we can choose to impose an order, like we did with SSTables previously, and use that as an indexing mechanism.

Note that it wouldn't make sense to sort each column independently, because then we would no longer know which items in the columns belong to the same row. We can only reconstruct a row because we know that the *k*th item in one column belongs to the same row as the *k*th item in another column.

Rather, the data needs to be sorted an entire row at a time, even though it is stored by column. The administrator of the database can choose the columns by which the table should be sorted, using their knowledge of common queries. For example, if queries often target date ranges, such as the last month, it might make sense to make `date_key` the first sort key. Then the query can scan only the rows from the last month, which will be much faster than scanning all rows.

A second column can determine the sort order of any rows that have the same value in the first column. For example, if `date_key` is the first sort key in Figure 4-7, it might make sense for `product_sk` to be the second sort key so that all sales for the same product on the same day are grouped together in storage. That will help queries that need to group or filter sales by product within a certain date range.

Another advantage of sorted order is that it can help with compression of columns. If the primary sort column does not have many distinct values, then after sorting, it will have long sequences where the same value is repeated many times in a row. A simple run-length encoding, like we used for the bitmaps in Figure 4-8, could compress that column down to a few kilobytes—even if the table has billions of rows.

That compression effect is strongest on the first sort key. The second and third sort keys will be more jumbled up, and thus not have such long runs of repeated values. Columns further down the sorting priority appear in essentially random order, so they probably won't compress as well. But having the first few columns sorted is still a win overall.

Writing to Column-Oriented Storage

We saw in "Characterizing Transaction Processing and Analytics" that reads in data warehouses tend to consist of aggregations over a large number of rows; column-oriented storage, compression, and sorting all help to make those read queries faster. Writes in a data warehouse tend to be a bulk import of data, often via an ETL process.

With columnar storage, writing an individual row somewhere in the middle of a sorted table would be very inefficient, as you would have to rewrite all the compressed columns from the insertion position onwards. However, a bulk

write of many rows at once amortizes the cost of rewriting those columns, making it efficient.

A log-structured approach is often used to perform writes in batches. All writes first go to a row-oriented, sorted, in-memory store. When enough writes have accumulated, they are merged with the column-encoded files on disk and written to new files in bulk. As old files remain immutable, and new files are written in one go, object storage is well suited for storing these files.

Queries need to examine both the column data on disk and the recent writes in memory, and combine the two. The query execution engine hides this distinction from the user. From an analyst's point of view, data that has been modified with inserts, updates, or deletes is immediately reflected in subsequent queries. Snowflake, Vertica, Apache Pinot, Apache Druid, and many others do this [61, 63, 64, 76].

Query Execution: Compilation and Vectorization

A complex SQL query for analytics is broken down into a *query plan* consisting of multiple stages, called *operators*, which may be distributed across multiple machines for parallel execution. Query planners can perform a lot of optimizations by choosing which operators to use, in which order to perform them, and where to run each operator.

Within each operator, the query engine needs to do various things with the values in a column, such as finding all the rows where the value is among a particular set of values (perhaps as part of a join), or checking whether the value is greater than 15. It also needs to look at several columns for the same row, for example to find all sales transactions where the product is bananas and the store is a particular store of interest.

For data warehouse queries that need to scan over millions of rows, we need to worry not only about the amount of data they need to read off disk, but also the CPU time required to execute complex operators. The simplest kind of operator is like an interpreter for a programming language: while iterating over each row, it checks a data structure representing the query to find out which comparisons or calculations it needs to perform on which columns. Unfortunately, this is too slow for many analytics purposes. Two alternative approaches for efficient query execution have emerged [77]:

Query compilation

The query engine takes the SQL query and generates code for executing it. The code iterates over the rows one by one, looks at the values in the columns of interest, performs whatever comparisons or calculations are needed, and copies the necessary values to an output buffer if the required conditions are satisfied. The query engine compiles the generated code to machine code (often using an existing compiler such as LLVM), and then runs it on the column-encoded data that has been loaded into memory. This approach to code generation is similar to the just-in-time (JIT) compilation approach that is used in the Java Virtual Machine (JVM) and similar runtimes.

Vectorized processing

The query is interpreted, not compiled, but it is made fast by processing many values from a column in a batch, instead of iterating over rows one by one. A fixed set of predefined operators are built into the database; we can pass arguments to them and get back a batch of results [50, 75].

For example, we could pass the `product_sk` column and the ID of "bananas" to an equality operator, and get back a bitmap (one bit per value in the input column, which is 1 if it's a banana); we could then pass the `store_sk` column and the ID of the store of interest to the same equality operator, and get back another bitmap; and then we could pass the two bitmaps to a "bitwise AND" operator, as shown in Figure 4-9. The result would be a bitmap containing a 1 for all sales of bananas in a particular store.

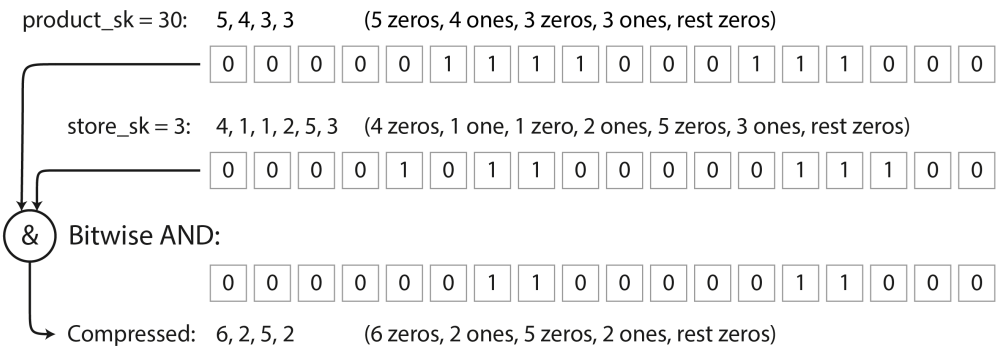

Figure 4-9. A bitwise AND between two bitmaps lends itself to vectorization.

The two approaches are very different in terms of their implementation, but both are used in practice [77]. Both can achieve very good performance by

taking advantages of the characteristics of modern CPUs:

- preferring sequential memory access over random access to reduce cache misses [78],
- doing most of the work in tight inner loops (that is, with a small number of instructions and no function calls) to keep the CPU instruction processing pipeline busy and avoid branch mispredictions,
- making use of parallelism such as multiple threads and single-instruction-multi-data (SIMD) instructions [79, 80], and
- operating directly on compressed data without decoding it into a separate in-memory representation, which saves memory allocation and copying costs.

Materialized Views and Data Cubes

We previously encountered *materialized views* in "Materializing and Updating Timelines": in a relational data model, they are table-like object whose contents are the results of some query. The difference is that a materialized view is an actual copy of the query results, written to disk, whereas a virtual view is just a shortcut for writing queries. When you read from a virtual view, the SQL engine expands it into the view's underlying query on the fly and then processes the expanded query.

When the underlying data changes, a materialized view needs to be updated accordingly. Some databases can do that automatically, and there are also systems such as Materialize that specialize in materialized view maintenance [81]. We will return to this topic in "Maintaining materialized views". Performing such updates means more work on writes, but materialized views can improve read performance in workloads that repeatedly need to perform the same queries.

Materialized aggregates are a type of materialized views that can be useful in data warehouses. As discussed earlier, data warehouse queries often involve an aggregate function, such as `COUNT`, `SUM`, `AVG`, `MIN`, or `MAX` in SQL. If the same aggregates are used by many different queries, it can be wasteful to crunch through the raw data every time. Why not cache some of the counts or sums that queries use most often? A *data cube* or *OLAP cube* does this by creating a grid of aggregates grouped by different dimensions [82]. Figure 4-10 shows an example.

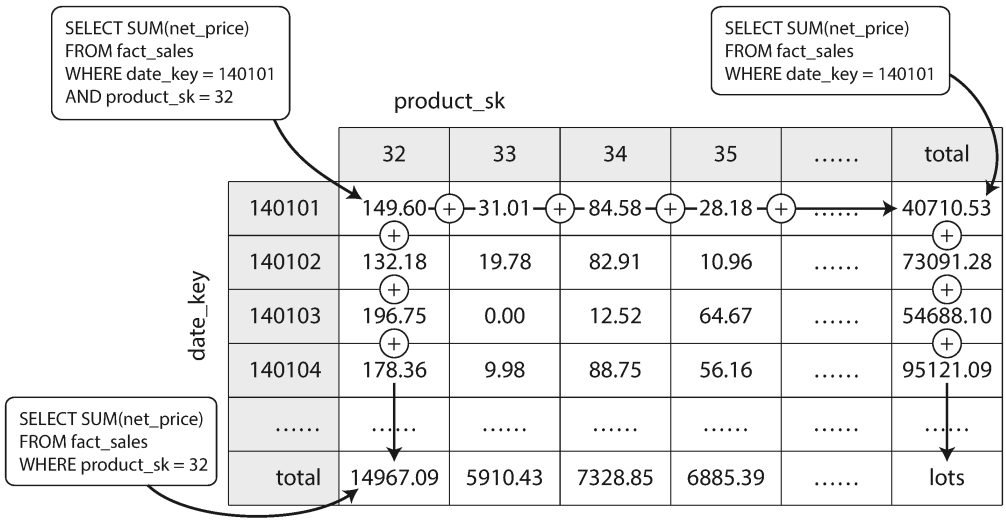

Figure 4-10. Two dimensions of a data cube, aggregating data by summing.

Imagine for now that each fact has foreign keys to only two dimension tables —in Figure 4-10, these are `date_key` and `product_sk`. You can now draw a two-dimensional table, with dates along one axis and products along the other. Each cell contains the aggregate (e.g., `SUM`) of an attribute (e.g., `net_price`) of all facts with that date-product combination. Then you can apply the same aggregate along each row or column and get a summary that has been reduced by one dimension (the sales by product regardless of date, or the sales by date regardless of product).

In general, facts often have more than two dimensions. In Figure 3-5 there are five dimensions: date, product, store, promotion, and customer. It's a lot harder to imagine what a five-dimensional hypercube would look like, but the principle remains the same: each cell contains the sales for a particular date-product-store-promotion-customer combination. These values can then repeatedly be summarized along each of the dimensions.

The advantage of a materialized data cube is that certain queries become very fast because they have effectively been precomputed. For example, if you want to know the total sales per store yesterday, you just need to look at the totals along the appropriate dimension—no need to scan millions of rows.

The disadvantage is that a data cube doesn't have the same flexibility as querying the raw data. For example, there is no way of calculating which proportion of sales comes from items that cost more than $100, because the price isn't one of the dimensions. Most data warehouses therefore try to keep as much raw data as possible, and use aggregates such as data cubes only as a performance boost for certain queries.

Multidimensional and Full-Text Indexes

The B-trees and LSM-trees we saw in the first half of this chapter allow range queries over a single attribute: for example, if the key is a username, you can use them as an index to efficiently find all names starting with an L. But sometimes, searching by a single attribute is not enough.

The most common type of multi-column index is called a *concatenated index*, which simply combines several fields into one key by appending one column to another (the index definition specifies in which order the fields are concatenated). This is like an old-fashioned paper phone book, which provides an index from (*lastname*, *firstname*) to phone number. Due to the sort order, the index can be used to find all the people with a particular last name, or all the people with a particular *lastname-firstname* combination. However, the index is useless if you want to find all the people with a particular first name.

On the other hand, *multi-dimensional indexes* allow you to query several columns at once. One case where this is particularly important is geospatial data. For example, a restaurant-search website may have a database containing the latitude and longitude of each restaurant. When a user is looking at the restaurants on a map, the website needs to search for all the restaurants within the rectangular map area that the user is currently viewing. This requires a two-dimensional range query like the following:

```
SELECT * FROM restaurants WHERE latitude  > 51.4946 ANI
                            AND longitude > -0.1162 ANI
```

A concatenated index over the latitude and longitude columns is not able to answer that kind of query efficiently: it can give you either all the restaurants in a range of latitudes (but at any longitude), or all the restaurants in a range of longitudes (but anywhere between the North and South poles), but not both simultaneously.

One option is to translate a two-dimensional location into a single number using a space-filling curve, and then to use a regular B-tree index [83]. More commonly, specialized spatial indexes such as R-trees or Bkd-trees [84] are used; they divide up the space so that nearby data points tend to be grouped in the same subtree. For example, PostGIS implements geospatial indexes as R-

trees using PostgreSQL's Generalized Search Tree indexing facility [85]. It is also possible to use regularly spaced grids of triangles, squares, or hexagons [86].

Multi-dimensional indexes are not just for geographic locations. For example, on an ecommerce website you could use a three-dimensional index on the dimensions (*red*, *green*, *blue*) to search for products in a certain range of colors, or in a database of weather observations you could have a two-dimensional index on (*date*, *temperature*) in order to efficiently search for all the observations during the year 2013 where the temperature was between 25 and 30°C. With a one-dimensional index, you would have to either scan over all the records from 2013 (regardless of temperature) and then filter them by temperature, or vice versa. A 2D index could narrow down by timestamp and temperature simultaneously [87].

Full-Text Search

Full-text search allows you to search a collection of text documents (web pages, product descriptions, etc.) by keywords that might appear anywhere in the text [88]. Information retrieval is a big, specialist topic that often involves language-specific processing: for example, several Asian languages are written without spaces or punctuation between words, and therefore splitting text into words requires a model that indicates which character sequences constitute a word. Full-text search also often involves matching words that are similar but not identical (such as typos or different grammatical forms of words) and synonyms. Those problems go beyond the scope of this book.

However, at its core, you can think of full-text search as another kind of multidimensional query: in this case, each word that might appear in a text (a *term*) is a dimension. A document that contains term *x* has a value of 1 in dimension *x*, and a document that doesn't contain *x* has a value of 0. Searching for documents mentioning "red apples" means a query that looks for a 1 in the *red* dimension, and simultaneously a 1 in the *apples* dimension. The number of dimensions may thus be very large.

The data structure that many search engines use to answer such queries is called an *inverted index*. This is a key-value structure where the key is a term, and the value is the list of IDs of all the documents that contain the term (the *postings list*). If the document IDs are sequential numbers, the postings list

can also be represented as a sparse bitmap, like in Figure 4-8: the *n*th bit in the bitmap for term *x* is a 1 if the document with ID *n* contains the term *x* [89].

Finding all the documents that contain both terms *x* and *y* is now similar to a vectorized data warehouse query that searches for rows matching two conditions (Figure 4-9): load the two bitmaps for terms *x* and *y* and compute their bitwise AND. Even if the bitmaps are run-length encoded, this can be done very efficiently.

For example, Lucene, the full-text indexing engine used by Elasticsearch and Solr, works like this [90]. It stores the mapping from term to postings list in SSTable-like sorted files, which are merged in the background using the same log-structured approach we saw earlier in this chapter [91]. PostgreSQL's GIN index type also uses postings lists to support full-text search and indexing inside JSON documents [92, 93].

Instead of breaking text into words, an alternative is to find all the substrings of length *n*, which are called *n*-grams. For example, the trigrams ($n = 3$) of the string `"hello"` are `"hel"`, `"ell"`, and `"llo"`. If we build an inverted index of all trigrams, we can search the documents for arbitrary substrings that are at least three characters long. Trigram indexes even allows regular expressions in search queries; the downside is that they are quite large [94].

To cope with typos in documents or queries, Lucene is able to search text for words within a certain edit distance (an edit distance of 1 means that one letter has been added, removed, or replaced) [95]. It does this by storing the set of terms as a finite state automaton over the characters in the keys, similar to a *trie* [96], and transforming it into a *Levenshtein automaton*, which supports efficient search for words within a given edit distance [97].

Vector Embeddings

Semantic search goes beyond synonyms and typos to try and understand document concepts and user intentions. For example, if your help pages contain a page titled "cancelling your subscription", users should still be able to find that page when searching for "how to close my account" or "terminate contract", which are close in terms of meaning even though they use completely different words.

To understand a document's semantics—its meaning—semantic search indexes use embedding models to translate a document into a vector of floating-point values, called a *vector embedding*. The vector represents a point in a multi-dimensional space, and each floating-point value represents the document's location along one dimension's axis. Embedding models generate vector embeddings that are near each other (in this multi-dimensional space) when the embedding's input documents are semantically similar.

NOTE

We saw the term *vectorized processing* in "Query Execution: Compilation and Vectorization". Vectors in semantic search have a different meaning. In vectorized processing, the vector refers to a batch of bits that can be processed with specially optimized code. In embedding models, vectors are a list of floating point numbers that represent a location in multi-dimensional space.

For example, a three-dimensional vector embedding for a Wikipedia page about agriculture might be [0.1, 0.22, 0.11]. A Wikipedia page about vegetables would be quite near, perhaps with an embedding of [0.13, 0.19, 0.24]. A page about star schemas might have an embedding of [0.82, 0.39, -0.74], comparatively far away. We can tell by looking that the first two vectors are closer than the third.

Embedding models use much larger vectors (often over 1,000 numbers), but the principles are the same. We don't try to understand what the individual numbers mean; they're simply a way for embedding models to point to a location in an abstract multi-dimensional space. Search engines use distance functions such as cosine similarity or Euclidean distance to measure the distance between vectors. Cosine similarity measures the cosine of the angle of two vectors to determine how close they are, while Euclidean distance measures the straight-line distance between two points in space.

Many early embedding models such as Word2Vec [98], BERT [99], and GPT [100] worked with text data. Such models are usually implemented as neural networks. Researchers went on to create embedding models for video, audio, and images as well. More recently, model architecture has become *multimodal*: a single model can generate vector embeddings for multiple modalities such as text and images.

Semantic search engines use an embedding model to generate a vector embedding when a user enters a query. The user's query and related context (such as a user's location) are fed into the embedding model. After the embedding model generates the query's vector embedding, the search engine must find documents with similar vector embeddings using a vector index.

Vector indexes store the vector embeddings of a collection of documents. To query the index, you pass in the vector embedding of the query, and the index returns the documents whose vectors are closest to the query vector. Since the R-trees we saw previously don't work well for vectors with many dimensions, specialized vector indexes are used, such as:

Flat indexes

Vectors are stored in the index as they are. A query must read every vector and measure its distance to the query vector. Flat indexes are accurate, but measuring the distance between the query and each vector is slow.

Inverted file (IVF) indexes

The vector space is clustered into partitions (called *centroids*) of vectors to reduce the number of vectors that must be compared. IVF indexes are faster than flat indexes, but can give only approximate results: the query and a document may fall into different partitions, even though they are close to each other. A query on an IVF index first defines *probes*, which are simply the number of partitions to check. Queries that use more probes will be more accurate, but will be slower, as more vectors must be compared.

Hierarchical Navigable Small World (HNSW)

HNSW indexes maintain multiple layers of the vector space, as illustrated in Figure 4-11. Each layer is represented as a graph, where nodes represent vectors, and edges represent proximity to nearby vectors. A query starts by locating the nearest vector in the topmost layer, which has a small number of nodes. The query then moves to the same node in the layer below and follows the edges in that layer, which is more densely connected, looking for a vector that is closer to the query vector. The process continues until the last layer is reached. As with IVF indexes, HNSW indexes are approximate.

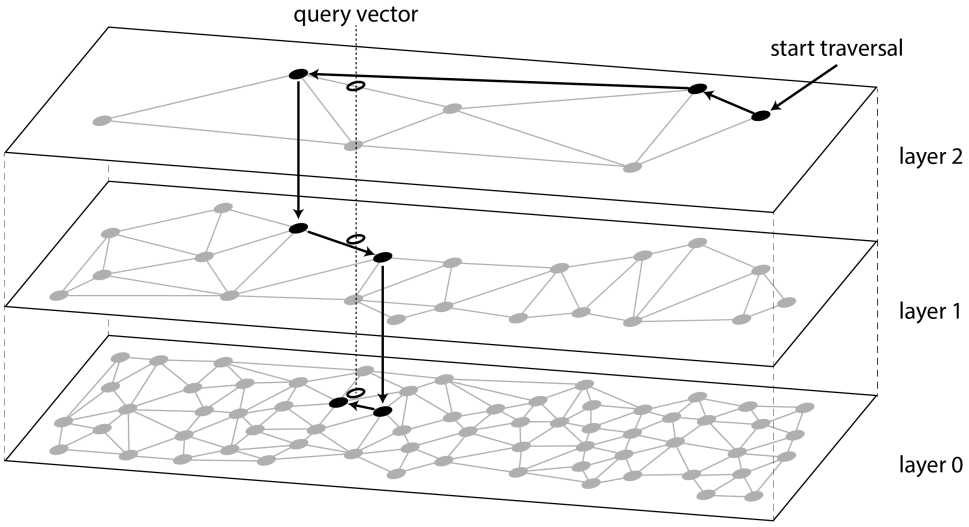

Figure 4-11. Searching for the database entry that is closest to a given query vector in a HNSW index.

Many popular vector databases implement IVF and HNSW indexes. Facebook's Faiss library has many variations of each [101], and PostgreSQL's pgvector supports both as well [102]. The full details of the IVF and HNSW algorithms are beyond the scope of this book, but their papers are an excellent resource [103, 104].

Summary

In this chapter we tried to get to the bottom of how databases perform storage and retrieval. What happens when you store data in a database, and what does the database do when you query for the data again later?

"Operational Versus Analytical Systems" introduced the distinction between transaction processing (OLTP) and analytics (OLAP). In this chapter we saw that storage engines optimized for OLTP look very different from those optimized for analytics:

- OLTP systems are optimized for a high volume of requests, each of which reads and writes a small number of records, and which need fast responses. The records are typically accessed via a primary key or a secondary index, and these indexes are typically ordered mappings from key to record, which also support range queries.
- Data warehouses and similar analytic systems are optimized for complex read queries that scan over a large number of records. They generally use a column-oriented storage layout with compression that minimizes the amount of data that such a query needs to read off disk, and just-in-time

compilation of queries or vectorization to minimize the amount of CPU time spent processing the data.

On the OLTP side, we saw storage engines from two main schools of thought:

- The log-structured approach, which only permits appending to files and deleting obsolete files, but never updates a file that has been written. SSTables, LSM-trees, RocksDB, Cassandra, HBase, Scylla, Lucene, and others belong to this group. In general, log-structured storage engines tend to provide high write throughput.
- The update-in-place approach, which treats the disk as a set of fixed-size pages that can be overwritten. B-trees, the biggest example of this philosophy, are used in all major relational OLTP databases and also many nonrelational ones. As a rule of thumb, B-trees tend to be better for reads, providing higher read throughput and lower response times than log-structured storage.

We then looked at indexes that can search for multiple conditions at the same time: multidimensional indexes such as R-trees that can search for points on a map by latitude and longitude at the same time, and full-text search indexes that can search for multiple keywords appearing in the same text. Finally, vector databases are used for semantic search on text documents and other media; they use vectors with a larger number of dimensions and find similar documents by comparing vector similarity.

As an application developer, if you're armed with this knowledge about the internals of storage engines, you are in a much better position to know which tool is best suited for your particular application. If you need to adjust a database's tuning parameters, this understanding allows you to imagine what effect a higher or a lower value may have.

Although this chapter couldn't make you an expert in tuning any one particular storage engine, it has hopefully equipped you with enough vocabulary and ideas that you can make sense of the documentation for the database of your choice.

FOOTNOTES

REFERENCES

[1] Nikolay Samokhvalov. How partial, covering, and multicolumn indexes may slow down UPDATEs in PostgreSQL. *postgres.ai*, October 2021. Archived at perma.cc/PBK3-F4G9

[2] Goetz Graefe. Modern B-Tree Techniques. *Foundations and Trends in Databases*, volume 3, issue 4, pages 203–402, August 2011. doi:10.1561/1900000028

[3] Evan Jones. Why databases use ordered indexes but programming uses hash tables. *evanjones.ca*, December 2019. Archived at perma.cc/NJX8-3ZZD

[4] Branimir Lambov. CEP-25: Trie-indexed SSTable format. *cwiki.apache.org*, November 2022. Archived at perma.cc/HD7W-PW8U. Linked Google Doc archived at perma.cc/UL6C-AAAE

[5] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein: *Introduction to Algorithms*, 3rd edition. MIT Press, 2009. ISBN: 978-0-262-53305-8

[6] Branimir Lambov. Trie Memtables in Cassandra. *Proceedings of the VLDB Endowment*, volume 15, issue 12, pages 3359–3371, August 2022. doi:10.14778/3554821.3554828

[7] Dhruba Borthakur. The History of RocksDB. *rocksdb.blogspot.com*, November 2013. Archived at perma.cc/Z7C5-JPSP

[8] Matteo Bertozzi. Apache HBase I/O – HFile. *blog.cloudera.com*, June 2012. Archived at perma.cc/U9XH-L2KL

[9] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A Distributed Storage System for Structured Data. At *7th USENIX Symposium on Operating System Design and Implementation* (OSDI), November 2006.

[10] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. The Log-Structured Merge-Tree (LSM-Tree). *Acta Informatica*, volume 33, issue 4, pages 351–385, June 1996. doi:10.1007/s002360050048

[11] Mendel Rosenblum and John K. Ousterhout. The Design and Implementation of a Log-Structured File System. *ACM Transactions on Computer Systems*, volume 10, issue 1, pages 26–52, February 1992. doi:10.1145/146941.146943

[12] Michael Armbrust, Tathagata Das, Liwen Sun, Burak Yavuz, Shixiong Zhu, Mukul Murthy, Joseph Torres, Herman van Hovell, Adrian Ionescu, Alicja Łuszczak, Michał

Świtakowski, Michał Szafrański, Xiao Li, Takuya Ueshin, Mostafa Mokhtar, Peter Boncz, Ali Ghodsi, Sameer Paranjpye, Pieter Senster, Reynold Xin, and Matei Zaharia. Delta Lake: High-Performance ACID Table Storage over Cloud Object Stores. *Proceedings of the VLDB Endowment*, volume 13, issue 12, pages 3411–3424, August 2020. doi:10.14778/3415478.3415560

[13] Burton H. Bloom. Space/Time Trade-offs in Hash Coding with Allowable Errors. *Communications of the ACM*, volume 13, issue 7, pages 422–426, July 1970. doi:10.1145/362686.362692

[14] Adam Kirsch and Michael Mitzenmacher. Less Hashing, Same Performance: Building a Better Bloom Filter. *Random Structures & Algorithms*, volume 33, issue 2, pages 187–218, September 2008. doi:10.1002/rsa.20208

[15] Thomas Hurst. Bloom Filter Calculator. *hur.st*, September 2023. Archived at perma.cc/L3AV-6VC2

[16] Chen Luo and Michael J. Carey. LSM-based storage techniques: a survey. *The VLDB Journal*, volume 29, pages 393–418, July 2019. doi:10.1007/s00778-019-00555-y

[17] Subhadeep Sarkar and Manos Athanassoulis. Dissecting, Designing, and Optimizing LSM-based Data Stores. Tutorial at *ACM International Conference on Management of Data* (SIGMOD), June 2022. Slides archived at perma.cc/93B3-E827

[18] Mark Callaghan. Name that compaction algorithm. *smalldatum.blogspot.com*, August 2018. Archived at perma.cc/CN4M-82DY

[19] Prashanth Rao. Embedded databases (1): The harmony of DuckDB, KùzuDB and LanceDB. *thedataquarry.com*, August 2023. Archived at perma.cc/PA28-2R35

[20] Hacker News discussion. Bluesky migrates to single-tenant SQLite. *news.ycombinator.com*, October 2023. Archived at perma.cc/69LM-5P6X

[21] Rudolf Bayer and Edward M. McCreight. Organization and Maintenance of Large Ordered Indices. Boeing Scientific Research Laboratories, Mathematical and Information Sciences Laboratory, report no. 20, July 1970. doi:10.1145/1734663.1734671

[22] Douglas Comer. The Ubiquitous B-Tree. *ACM Computing Surveys*, volume 11, issue 2, pages 121–137, June 1979. doi:10.1145/356770.356776

[23] Alex Miller. Torn Write Detection and Protection. *transactional.blog*, April 2025. Archived at perma.cc/G7EB-33EW

[24] C. Mohan and Frank Levine. ARIES/IM: An Efficient and High Concurrency Index Management Method Using Write-Ahead Logging. At *ACM International Conference on Management of Data* (SIGMOD), June 1992. doi:10.1145/130283.130338

[25] Hironobu Suzuki. The Internals of PostgreSQL. *interdb.jp*, 2017.

[26] Howard Chu. LDAP at Lightning Speed. At *Build Stuff '14*, November 2014. Archived at perma.cc/GB6Z-P8YH

[27] Manos Athanassoulis, Michael S. Kester, Lukas M. Maas, Radu Stoica, Stratos Idreos, Anastasia Ailamaki, and Mark Callaghan. Designing Access Methods: The RUM Conjecture. At *19th International Conference on Extending Database Technology* (EDBT), March 2016. doi:10.5441/002/edbt.2016.42

[28] Ben Stopford. Log Structured Merge Trees. *benstopford.com*, February 2015. Archived at perma.cc/E5BV-KUJ6

[29] Mark Callaghan. The Advantages of an LSM vs a B-Tree. *smalldatum.blogspot.co.uk*, January 2016. Archived at perma.cc/3TYZ-EFUD

[30] Oana Balmau, Florin Dinu, Willy Zwaenepoel, Karan Gupta, Ravishankar Chandhiramoorthi, and Diego Didona. SILK: Preventing Latency Spikes in Log-Structured Merge Key-Value Stores. At *USENIX Annual Technical Conference*, July 2019.

[31] Igor Canadi, Siying Dong, Mark Callaghan, et al. RocksDB Tuning Guide. *github.com*, 2023. Archived at perma.cc/UNY4-MK6C

[32] Gabriel Haas and Viktor Leis. What Modern NVMe Storage Can Do, and How to Exploit it: High-Performance I/O for High-Performance Storage Engines. *Proceedings of the VLDB Endowment*, volume 16, issue 9, pages 2090-2102. doi:10.14778/3598581.3598584

[33] Emmanuel Goossaert. Coding for SSDs. *codecapsule.com*, February 2014.

[34] Jack Vanlightly. Is sequential IO dead in the era of the NVMe drive? *jack-vanlightly.com*, May 2023. Archived at perma.cc/7TMZ-TAPU

[35] Alibaba Cloud Storage Team. Storage System Design Analysis: Factors Affecting NVMe SSD Performance (2). *alibabacloud.com*, January 2019. Archived at archive.org

[36] Xiao-Yu Hu and Robert Haas. The Fundamental Limit of Flash Random Write Performance: Understanding, Analysis and Performance Modelling.

dominoweb.draco.res.ibm.com, March 2010. Archived at perma.cc/8JUL-4ZDS

[37] Lanyue Lu, Thanumalayan Sankaranarayana Pillai, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. WiscKey: Separating Keys from Values in SSD-conscious Storage. At *4th USENIX Conference on File and Storage Technologies* (FAST), February 2016.

[38] Peter Zaitsev. Innodb Double Write. *percona.com*, August 2006. Archived at perma.cc/NT4S-DK7T

[39] Tomas Vondra. On the Impact of Full-Page Writes. *2ndquadrant.com*, November 2016. Archived at perma.cc/7N6B-CVL3

[40] Mark Callaghan. Read, write & space amplification - B-Tree vs LSM. *smalldatum.blogspot.com*, November 2015. Archived at perma.cc/S487-WK5P

[41] Mark Callaghan. Choosing Between Efficiency and Performance with RocksDB. At *Code Mesh*, November 2016. Video at youtube.com/watch?v=tgzkgZVXKB4

[42] Subhadeep Sarkar, Tarikul Islam Papon, Dimitris Staratzis, Zichen Zhu, and Manos Athanassoulis. Enabling Timely and Persistent Deletion in LSM-Engines. *ACM Transactions on Database Systems*, volume 48, issue 3, article no. 8, August 2023. doi:10.1145/3599724

[43] Lukas Fittl. Postgres vs. SQL Server: B-Tree Index Differences & the Benefit of Deduplication. *pganalyze.com*, April 2025. Archived at perma.cc/XY6T-LTPX

[44] Drew Silcock. How Postgres stores data on disk – this one's a page turner. *drew.silcock.dev*, August 2024. Archived at perma.cc/8K7K-7VJ2

[45] Joe Webb. Using Covering Indexes to Improve Query Performance. *simple-talk.com*, September 2008. Archived at perma.cc/6MEZ-R5VR

[46] Michael Stonebraker, Samuel Madden, Daniel J. Abadi, Stavros Harizopoulos, Nabil Hachem, and Pat Helland. The End of an Architectural Era (It's Time for a Complete Rewrite). At *33rd International Conference on Very Large Data Bases* (VLDB), September 2007.

[47] VoltDB Technical Overview White Paper. VoltDB, 2017. Archived at perma.cc/B9SF-SK5G

[48] Stephen M. Rumble, Ankita Kejriwal, and John K. Ousterhout. Log-Structured Memory for DRAM-Based Storage. At *12th USENIX Conference on File and Storage Technologies* (FAST), February 2014.

[49] Stavros Harizopoulos, Daniel J. Abadi, Samuel Madden, and Michael Stonebraker. OLTP Through the Looking Glass, and What We Found There. At *ACM International Conference on Management of Data* (SIGMOD), June 2008. doi:10.1145/1376616.1376713

[50] Per-Åke Larson, Cipri Clinciu, Campbell Fraser, Eric N. Hanson, Mostafa Mokhtar, Michal Nowakiewicz, Vassilis Papadimos, Susan L. Price, Srikumar Rangarajan, Remus Rusanu, and Mayukh Saubhasik. Enhancements to SQL Server Column Stores. At *ACM International Conference on Management of Data* (SIGMOD), June 2013. doi:10.1145/2463676.2463708

[51] Franz Färber, Norman May, Wolfgang Lehner, Philipp Große, Ingo Müller, Hannes Rauhe, and Jonathan Dees. The SAP HANA Database – An Architecture Overview. *IEEE Data Engineering Bulletin*, volume 35, issue 1, pages 28–33, March 2012.

[52] Michael Stonebraker. The Traditional RDBMS Wisdom Is (Almost Certainly) All Wrong. Presentation at *EPFL*, May 2013.

[53] Adam Prout, Szu-Po Wang, Joseph Victor, Zhou Sun, Yongzhu Li, Jack Chen, Evan Bergeron, Eric Hanson, Robert Walzer, Rodrigo Gomes, and Nikita Shamgunov. Cloud-Native Transactions and Analytics in SingleStore. At *ACM International Conference on Management of Data* (SIGMOD), June 2022. doi:10.1145/3514221.3526055

[54] Tino Tereshko and Jordan Tigani. BigQuery under the hood. *cloud.google.com*, January 2016. Archived at perma.cc/WP2Y-FUCF

[55] Wes McKinney. The Road to Composable Data Systems: Thoughts on the Last 15 Years and the Future. *wesmckinney.com*, September 2023. Archived at perma.cc/6L2M-GTJX

[56] Michael Stonebraker, Daniel J. Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Sam Madden, Elizabeth O'Neil, Pat O'Neil, Alex Rasin, Nga Tran, and Stan Zdonik. C-Store: A Column-oriented DBMS. At *31st International Conference on Very Large Data Bases* (VLDB), pages 553–564, September 2005.

[57] Julien Le Dem. Dremel Made Simple with Parquet. *blog.twitter.com*, September 2013.

[58] Sergey Melnik, Andrey Gubarev, Jing Jing Long, Geoffrey Romer, Shiva Shivakumar, Matt Tolton, and Theo Vassilakis. Dremel: Interactive Analysis of Web-Scale Datasets. At *36th International Conference on Very Large Data Bases* (VLDB), pages 330–339, September 2010. doi:10.14778/1920841.1920886

[59] Joe Kearney. Understanding Record Shredding: storing nested data in columns. *joekearney.co.uk*, December 2016. Archived at perma.cc/ZD5N-AX5D

[60] Jamie Brandon. A shallow survey of OLAP and HTAP query engines. *scattered-thoughts.net*, September 2023. Archived at perma.cc/L3KH-J4JF

[61] Benoit Dageville, Thierry Cruanes, Marcin Zukowski, Vadim Antonov, Artin Avanes, Jon Bock, Jonathan Claybaugh, Daniel Engovatov, Martin Hentschel, Jiansheng Huang, Allison W. Lee, Ashish Motivala, Abdul Q. Munir, Steven Pelley, Peter Povinec, Greg Rahn, Spyridon Triantafyllis, and Philipp Unterbrunner. The Snowflake Elastic Data Warehouse. At *ACM International Conference on Management of Data* (SIGMOD), pages 215–226, June 2016. doi:10.1145/2882903.2903741

[62] Mark Raasveldt and Hannes Mühleisen. Data Management for Data Science Towards Embedded Analytics. At *10th Conference on Innovative Data Systems Research* (CIDR), January 2020.

[63] Jean-François Im, Kishore Gopalakrishna, Subbu Subramaniam, Mayank Shrivastava, Adwait Tumbde, Xiaotian Jiang, Jennifer Dai, Seunghyun Lee, Neha Pawar, Jialiang Li, and Ravi Aringunram. Pinot: Realtime OLAP for 530 Million Users. At *ACM International Conference on Management of Data* (SIGMOD), pages 583–594, May 2018. doi:10.1145/3183713.3190661

[64] Fangjin Yang, Eric Tschetter, Xavier Léauté, Nelson Ray, Gian Merlino, and Deep Ganguli. Druid: A Real-time Analytical Data Store. At *ACM International Conference on Management of Data* (SIGMOD), June 2014. doi:10.1145/2588555.2595631

[65] Chunwei Liu, Anna Pavlenko, Matteo Interlandi, and Brandon Haynes. Deep Dive into Common Open Formats for Analytical DBMSs. *Proceedings of the VLDB Endowment*, volume 16, issue 11, pages 3044–3056, July 2023. doi:10.14778/3611479.3611507

[66] Xinyu Zeng, Yulong Hui, Jiahong Shen, Andrew Pavlo, Wes McKinney, and Huanchen Zhang. An Empirical Evaluation of Columnar Storage Formats. *Proceedings of the VLDB Endowment*, volume 17, issue 2, pages 148–161. doi:10.14778/3626292.3626298

[67] Weston Pace. Lance v2: A columnar container format for modern data. *blog.lancedb.com*, April 2024. Archived at perma.cc/ZK3Q-S9VJ

[68] Yoav Helfman. Nimble, A New Columnar File Format. At *VeloxCon*, April 2024.

[69] Wes McKinney. Apache Arrow: High-Performance Columnar Data Framework. At *CMU Database Group – Vaccination Database Tech Talks*, December 2021.

[70] Wes McKinney. Python for Data Analysis, 3rd Edition. O'Reilly Media, August 2022. ISBN: 9781098104023

[71] Paul Dix. The Design of InfluxDB IOx: An In-Memory Columnar Database Written in Rust with Apache Arrow. At *CMU Database Group – Vaccination Database Tech Talks*, May 2021.

[72] Carlota Soto and Mike Freedman. Building Columnar Compression for Large PostgreSQL Databases. *timescale.com*, March 2024. Archived at perma.cc/7KTF-V3EH

[73] Daniel Lemire, Gregory Ssi-Yan-Kai, and Owen Kaser. Consistently faster and smaller compressed bitmaps with Roaring. *Software: Practice and Experience*, volume 46, issue 11, pages 1547–1569, November 2016. doi:10.1002/spe.2402

[74] Jaz Volpert. An entire Social Network in 1.6GB (GraphD Part 2). *jazco.dev*, April 2024. Archived at perma.cc/L27Z-QVMG

[75] Daniel J. Abadi, Peter Boncz, Stavros Harizopoulos, Stratos Idreos, and Samuel Madden. The Design and Implementation of Modern Column-Oriented Database Systems. *Foundations and Trends in Databases*, volume 5, issue 3, pages 197–280, December 2013. doi:10.1561/1900000024

[76] Andrew Lamb, Matt Fuller, Ramakrishna Varadarajan, Nga Tran, Ben Vandiver, Lyric Doshi, and Chuck Bear. The Vertica Analytic Database: C-Store 7 Years Later. *Proceedings of the VLDB Endowment*, volume 5, issue 12, pages 1790–1801, August 2012. doi:10.14778/2367502.2367518

[77] Timo Kersten, Viktor Leis, Alfons Kemper, Thomas Neumann, Andrew Pavlo, and Peter Boncz. Everything You Always Wanted to Know About Compiled and Vectorized Queries But Were Afraid to Ask. *Proceedings of the VLDB Endowment*, volume 11, issue 13, pages 2209–2222, September 2018. doi:10.14778/3275366.3284966

[78] Forrest Smith. Memory Bandwidth Napkin Math. *forrestthewoods.com*, February 2020. Archived at perma.cc/Y8U4-PS7N

[79] Peter Boncz, Marcin Zukowski, and Niels Nes. MonetDB/X100: Hyper-Pipelining Query Execution. At *2nd Biennial Conference on Innovative Data Systems Research* (CIDR), January 2005.

[80] Jingren Zhou and Kenneth A. Ross. Implementing Database Operations Using SIMD Instructions. At *ACM International Conference on Management of Data* (SIGMOD), pages 145–156, June 2002. doi:10.1145/564691.564709

[81] Kevin Bartley. OLTP Queries: Transfer Expensive Workloads to Materialize. *materialize.com*, August 2024. Archived at perma.cc/4TYM-TYD8

[82] Jim Gray, Surajit Chaudhuri, Adam Bosworth, Andrew Layman, Don Reichart, Murali Venkatrao, Frank Pellow, and Hamid Pirahesh. Data Cube: A Relational Aggregation Operator Generalizing Group-By, Cross-Tab, and Sub-Totals. *Data Mining and Knowledge Discovery*, volume 1, issue 1, pages 29–53, March 2007. doi:10.1023/A:1009726021843

[83] Frank Ramsak, Volker Markl, Robert Fenk, Martin Zirkel, Klaus Elhardt, and Rudolf Bayer. Integrating the UB-Tree into a Database System Kernel. At *26th International Conference on Very Large Data Bases* (VLDB), September 2000.

[84] Octavian Procopiuc, Pankaj K. Agarwal, Lars Arge, and Jeffrey Scott Vitter. Bkd-Tree: A Dynamic Scalable kd-Tree. At *8th International Symposium on Spatial and Temporal Databases* (SSTD), pages 46–65, July 2003. doi:10.1007/978-3-540-45072-6_4

[85] Joseph M. Hellerstein, Jeffrey F. Naughton, and Avi Pfeffer. Generalized Search Trees for Database Systems. At *21st International Conference on Very Large Data Bases* (VLDB), September 1995.

[86] Isaac Brodsky. H3: Uber's Hexagonal Hierarchical Spatial Index. *eng.uber.com*, June 2018. Archived at archive.org

[87] Robert Escriva, Bernard Wong, and Emin Gün Sirer. HyperDex: A Distributed, Searchable Key-Value Store. At *ACM SIGCOMM Conference*, August 2012. doi:10.1145/2377677.2377681

[88] Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze. *Introduction to Information Retrieval*. Cambridge University Press, 2008. ISBN: 978-0-521-86571-5, available online at nlp.stanford.edu/IR-book

[89] Jianguo Wang, Chunbin Lin, Yannis Papakonstantinou, and Steven Swanson. An Experimental Study of Bitmap Compression vs. Inverted List Compression. At *ACM International Conference on Management of Data* (SIGMOD), pages 993–1008, May 2017. doi:10.1145/3035918.3064007

[90] Adrien Grand. What is in a Lucene Index? At *Lucene/Solr Revolution*, November 2013. Archived at perma.cc/Z7QN-GBYY

[91] Michael McCandless. Visualizing Lucene's Segment Merges. *blog.mikemccandless.com*, February 2011. Archived at perma.cc/3ZV8-72W6

[92] Lukas Fittl. Understanding Postgres GIN Indexes: The Good and the Bad. *pganalyze.com*, December 2021. Archived at perma.cc/V3MW-26H6

[93] Jimmy Angelakos. The State of (Full) Text Search in PostgreSQL 12. At *FOSDEM*, February 2020. Archived at perma.cc/J6US-3WZS

[94] Alexander Korotkov. Index support for regular expression search. At *PGConf.EU Prague*, October 2012. Archived at perma.cc/5RFZ-ZKDQ

[95] Michael McCandless. Lucene's FuzzyQuery Is 100 Times Faster in 4.0. *blog.mikemccandless.com*, March 2011. Archived at perma.cc/E2WC-GHTW

[96] Steffen Heinz, Justin Zobel, and Hugh E. Williams. Burst Tries: A Fast, Efficient Data Structure for String Keys. *ACM Transactions on Information Systems*, volume 20, issue 2, pages 192–223, April 2002. doi:10.1145/506309.506312

[97] Klaus U. Schulz and Stoyan Mihov. Fast String Correction with Levenshtein Automata. *International Journal on Document Analysis and Recognition*, volume 5, issue 1, pages 67–85, November 2002. doi:10.1007/s10032-002-0082-8

[98] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient Estimation of Word Representations in Vector Space. At *International Conference on Learning Representations* (ICLR), May 2013. doi:10.48550/arXiv.1301.3781

[99] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. At *Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, volume 1, pages 4171–4186, June 2019. doi:10.18653/v1/N19-1423

[100] Alec Radford, Karthik Narasimhan, Tim Salimans, and Ilya Sutskever. Improving Language Understanding by Generative Pre-Training. *openai.com*, June 2018. Archived at perma.cc/5N3C-DJ4C

[101] Matthijs Douze, Maria Lomeli, and Lucas Hosseini. Faiss indexes. *github.com*, August 2024. Archived at perma.cc/2EWG-FPBS

[102] Varik Matevosyan. Understanding pgvector's HNSW Index Storage in Postgres. *lantern.dev*, August 2024. Archived at perma.cc/B2YB-JB59

[103] Dmitry Baranchuk, Artem Babenko, and Yury Malkov. Revisiting the Inverted Indices for Billion-Scale Approximate Nearest Neighbors. At *European Conference on Computer Vision* (ECCV), pages 202–216, September 2018. doi:10.1007/978-3-030-01258-8_13

[104] Yury A. Malkov and Dmitry A. Yashunin. Efficient and robust approximate nearest neighbor search using Hierarchical Navigable Small World graphs. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, volume 42, issue 4, pages 824–836, April 2020. doi:10.1109/TPAMI.2018.2889473