At its most fundamental, a database needs to do two things

- when you give it data, it should store it
- when you ask it again later, it should give the data back to you

In the previous chapter, we explored the scenario from the pov of the developer -- in what format you give the db your data, and the mechanism by which you can ask for it again.

Here, we discuss the same scenario from the pov of the database

- how can we store the data that we are given?
- how can we find it again when asked for it?

There is an especially large difference between storage engines optimized for transactional workloads and those optimized for analytics

- Column oriented stores are a family of storage engines that is optimized for analytics

but we begin by looking at storage engines in two kinds of db we have already discussed: Relational, and NoSQL.
There are two families of storage engine:
1. log structured storage engines
2. page oriented storage engines, such as b-trees.

Log structured storage engines

Data Structures that Power A Database

Appending to a file in general is efficient -- so many databases internally use a *log,* which is an append-only data file/sequence of records

- need to tack on concurrency control, reclaiming disk space, error handling, etc

Without an index, a naive lookup in a database is an O(n) operation

- an index is a data structure that reduces this cost -- the general idea is to keep some additional metadata on the side, which acts as a signpost and helps you to locate the data you want

An index is an additional structure that is derived from the pimary data.

- mantaining this additional structure does incur some overhead on writes, for which the best possible performance comes from just appending to a file
- but since we need to update our index/indices on every write, we incur some overhead

This is an important trade-off in the storage systems. Well designed indices speed up read queries, but every index slows down writes.

- this is why db's usually dont index everything by default, instead asking the app developer to choose indexes manually

Hash indexes

- use an in-memory hash map that stores the byte offset for every key
- whenever a new key is added to the db, also add it and the byte offset to the hash map
- on read, use the hash map to find the offset in the data file, seek to that location, and read the value
- this is what Bitcask, the storage engine for Riak, does
- useful when you can keep all the keys in memory

Q: if you keep appending to the log file, how do you avoid running out of disk space?

- break the log into segments of a certain size by closing a segment file when it reaches a certain size, then make subsequent writes to a new segment file
- then perform compaction on these segments: throw away duplicate keys and keep only the most recent update for every key
- this usually makes segments smaller, so you can even merge several segments together while performing the compaction
  - since segments are never modified after they are written, the new merged segment is written to a new file
- the merging and compaction of segments can be done in a background trhead, and while it is happening, we can serve read requests from the old segment files, and write requests to the latest segment file
- after compaction, just swtich read requests to use the new merged segment -- and delete the old segments
- each segment gets its own in-memory has table mapping keys to file offsets
- to find a value for a key
  - first check the most recent segment's hash map
  - then the 2nd most recent
  - and so on

Record deletion: add a deletion record to the data file, so the db knows to delete previous records for the same key during compaction.

Crash recovery: if the db is restarted, all the in-memory hash maps are lost. Bitcask stores on-disk snapshots of each segment's hash map, which can be loaded into memory quickly on reboot

Concurrency control: have only one writer thread. Segments are append-only and otherwise immutable, so they can be read concurrently by multiple threads

Question: why do append-only instead of just modifying the file in-place by overwriting the old value with the new one?
Answer:
- appending and segment merging are sequential write operations, which are much faster than random writes -- though this is less advantageous with flash-based SSDs
    - concurrency control is easier if segment files are immutable
    - merging old segments avoids the problem of data files getting fragmentted over time

But the hash-map has some limitations
- all the keys need to fit in memory -- this may not be true with a large number of keys
- range queries are not efficient -- you need to look up each key individually in the hashmap

SSTables and LSM Trees
Earlier, each log-structured storage segment is a sequence of k-v pairs that appear in the order that they were written, and values later in the log take precedence over values for the same key from earlier in the log.
- Apart from this, there is no meaning to the ordering of key-value pairs in the log

What if we require that the sequence of k-v pairs is instead sorted by key? ≡ Sorted String Table, or SSTable
- with this format, we cannot append new k-v pairs to a segment immediately since writes can occur in any order
- but how can we make on-disk writes sequential?

Advantages of SSTables over log segments with hash indexes:
1. merging segments is simple and efficient, even if the files are bigger than the available memory
    - the approach is like mergesort -- read input files side by side, look at the first key in each file, copy the lowest key into the output file
        - if the same key appears in several input segments, pick the value from the most recent segment and discard the others

2. To find a particular key in the file, we no longer an index of *all* the keys in memory. INstead, since the keys are sorted in each segment file, you can use a sparse in-memory index and look between indexed keys
- if the keys and vals all had fixed length, you wouldn't need any in-memory index at all -- could just binary search on a segment file -- but as records are variable-length, it is difficult to know when one record ends and the next one starts
3. Can compress records for keys between subsequent indexed keys -- so each entry of the sparse in-memory index then points to the start of a compressed block

But how you get the data to be sorted by key in the first place?

- can maintain a sorted data structure on disk -- like B-Trees do
- but maintaining it in-memory is easier -- using red-black trees, or AVL trees
  - with these data structures, you can insert keys in any order, and read them back in sorted order
  - then flush the data to disk at a certain checkpoint!

So:

- when a wrtie comes in, add it to an in-memory balanced tree data structure -- sometimes called a *memtable*
- when the memtable grows bigger than some threshold -- typically a few MB -- write it to disk as a SSTable file
  - the tree already has the keys sorted in order!
  - the new SSTable file becomes the most recent segment in the database!
  - while this flush happens, writers write to a new in-memory tree structure
- to serve a read req:
  - first try and find the key in the memtable
  - then in the most recent on-disk segment
  - and so on
- from time to time, run compaction
- to handle db crashes: maintain an on-disk log to which every write is first appended
  - can use this to reconstruct the most recent in-memory tree structure on reboot

Q: do we maintain the sparse in-memory mapping for each segment that is written to disk?

- presumably the answer is yes?

This is essentially how RocksDB and LevelDB work. Google Bigtable was the first to introduce SSTable and memtable concepts.

- LevelDB use level-based compaction -- hence the name
- Cassandra supports this and size-based compaction

Originally, this indexing structure - keeping a cascade of SSTables that are merged in the background - was called Log-Structured Merge-Tree, or LSM-Tree

Lucene, an indexing engine for full-text serarch used by ElasticSearch and Solr, uses a similar method for storing its term dictionary

- a full-text index is more complex than a key-value index, but the idea is similar: given a word in search query, find all the documents that mention the word
- uses a key-value structure
  - the key is the word (a *term)*
  - the value is the list of IDs of all the documents that contain the word (the *postings*

list)
- in Lucne, the mapping from term → postings is kept in SSTable-like sorted files, which are merged in the background as needed

Q: what does an update look like here? if there is a new document added? since existing records are immutable, do you need to recreate the mapping from scratch every time there is an update to the document base?

Bloom filters are typically used as a performance optimization -- else you need to check the memtable and then every segment all the way to the oldest to find a key that is not in the database

This scheme works well even when the dataset is much bigger than the available memory, and supports range queries well since the keys are sorted on write. And because disk writes are sequential, the LSM-tree supports high write throughput.

Page oriented storage engines

B-Trees

The most widely used indexing strucutre is the B-tree, and is quite difference from the log-structured indices we have seen so far. Introduced in 1970 but still remain the standard index implementation for almost all relational databases and many non-relational dbs as well

Like SSTables, B-trees keep key-value pairs sorted by key, which allows efficient key-value lookups and range queries. But they are very different beyond that.
- log strucutres use an append only log file
- page strucutres modify pages on disk in-place

The log-strucutred indices earlier break the database down into variable-size segments, typically several MB, and always write a segment sequentially.
In contrast, B-trees break the database down into fixed-size blocks or pages, traditionally 4KB in size, and read or write one page at a time.
- this design corresponds more closely to the underlying hardware, as disks are also arranged in fixed-size blocks

Each page is identified using an address or location, which allows one page to refer to another -- like a pointer, but on disk. We can use these page references to construct a tree of pages.

One page is designated the root page -- all lookups start here. It 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.

The data/key-value pairs themselves are only stored in the terminal *leaf pages*

The number of references to child pages in one page of the B-tree is called the *branching factor*

- typically, this is in the several hundred -- it depends on the amount of space required to store the page references and range boundaries

To update the value for an existing key in a B-tree

- seach for the leaf page containing that key
- change the value in that page
- write the page back to disk, with any references to the page remaining valid
- update in place, vs the SSTable scheme, which is append only and ignores all but the last

To add a new key

- find the page whose range encompasses the new key
- if there is space in the page, add the key
- if not, split the page into the two half-full pages, and update the parent page to account for the new subdivision of key ranges
- this algorithm keeps the tree balanced -- a B-tree with n keys always has a depth of O(logn)
- typically the depth of a B-tree database is about 3 or 4

To delete a key?? how do you keep the tree balanced?

B-trees usually implement a write ahead log (WAL), an append-only log to which every B-tree modification must be written before it is applied to the pages of the tree itself

Concurrency control: since you can have multiple threads accessing the B-tree at the same time, you typically need some kind of locking/latching

- more complicated than with log structured approaches, because they do all the merging and compaction in the background without interfering with incoming queries

Advantages of LSM-Trees
1. A B-tree must write every piece of data twice: once to the WAL, once to the page itself (perhaps more times if pages are split), even if only a few bytes in the page are modified. LSM-Trees also rewrite data multiple times due to merging and compaction - *write*

amplification is the effect of one write to the db causing multiple writes to disk over the course of the db's lifetime - this is especially a concern with SSDs, which can overwrite blocks only a limited number of times before wearing out
LSM-Trees typically have lower write amplification, and they usually have to sequentially write SSTable files rather than overwriting pages in the tree via random writes -- especially advantageous with magnetic hard drives

2. LSM-Trees compress better than B-Trees and produce smaller files on disk.
3. B-tree storage engines often leave disk space unused due to fragmentation -- when a page has leftover space. On the other hand, LSM-Trees do regular compaction to reduce fragmentation and max disk utilization

Downsides of LSM-Trees

1. Compaction can sometimes interfere with the performance of ongoing reads and writes -- disks have fixed write bandwidth, so sometime srequests may have to wait till disk finishes a compaction operation
2. The larger the db gets, more disk resources are needed for compaction. This can impact high write throughput workflows
3. Compaction has to be monitored to keep up with the rate of incoming writes
4. Multiple copies of the same key, which makes B-trees more attrative for dbs that want to offer strong transactional semantics -- eg transaction isolation using locks is easier to do with B-tree

Other indexing structures

Secondary indices
- indexed values are not necessarily unique, unlike with primary indices
- can used both B-trees and log-strucutred indexes

Storing values within the index
- the key in an index is the thing that queries search for. But the value can be either:
  - the row data itself
  - or a reference to where the row data is stored
- in the latter case, the place where rows are stored is called the *heap file*
  - often used when secondary indices are present -- avoids duplication, since indexes can just refer to a location in the heap file, where the data is stored (once)
    - instead of storing the same data multiple times per index
- but sometimes the hop from index → heap file is too expensive for read performance

    - so store the indexed row itself in an index -- clustered index

Spectrum from: clsutered index (store all row data in the index) -- covering index ≡ index with included columns i.e. store *some* of the table columns in the index -- non clustered index (only store references to the data in the index)

Multi column indexes

- so far, only discussed looking up rows by a single key
- to query data across multiple columns in a table/multiple fields in a document, this is not sufficient
- most common multi column index is a *concatenated* index -- combine several fields into one key by appending one column to another
- multi dimensional indices are more generalizable way of querying several columns at once -- especially important for geospatial data
    - where you need to query by latitude AND longitude
    - a standard B-tree or LSM-Tree will not suffice
        - they will search across either latitude or longitude, but not both
    - options
        - translate lat, long into a single number using a space filling curve, then use a b-tree index
        - specialized spatial indices like R-trees
    - 2D indices as a more general solution (beyond geospatial data)

Full text search and fuzzy indices

- Lucene uses an SSTable-like structure for its term dictionary
- the structure requires a small in-memory index that tells queries at which offset in the sorted file they need to look for a key
    - in LevelDB, the in-memory index is a sparse collection of some of the keys in the SSTable
    - in Lucene, the in-memory index is a finite state automaton over the characters in the keys, similar to a trie

Keep everything in memory

- cost-per-GB of memory is much lower now
- some in-memory dbs are used for caching only, where it is acceptable for data to be lost if a machine is restarted -- e.g. Memcached
- others aim for durability
    - hardware improvements like battery powered RAM
    - use a WAL pushed to disk
    - snapshots to disk
    - replicate in-memory db on other machines

- VoltDb, TimesTen are in-memory with a relational data model
- Redis provides weak durability by writing to disk periodically
- the perf boost is NOT from not having to read from disk. Rather it is from avoiding the overhead of encoding in-memory data in a form that can be written to disk
- *anti-caching dbs*: evict least recentyl used data to disk, load it back when necessary

Transaction Processing vs Analytics Processing

Difference between transaction processing and analytics processing

- transaction processing in applications is usually interactive, smaller data scale, look up small number of records by some key
  - low-latency reads and writes necessary
    - as opposed to batch processing jobs, which only run periodically
  - data typically represents latest state of data
  - OLTP -- on line TP
  - the bottleneck is disk seek time
- analytics usually needs to scan over huge buymber of records, reading only a few columns per record, and return aggregates
  - data typically represents a history of events that happened over time
  - to answet quetions like
    - what was total revenue in Jan
    - what brands are most purcahsed together in the summer
    - etc
  - OLAP
  - the bottleneck is disk bandwidth
- Data Warehouses typically used to store data for analytics

Data Warehousing

- typically a separate db that analysts query without affecting OLTP operations
- Extract-Transform-Load is the process of
  - getting getting data from OLTP systems, either as a periodic data dump or a continuouis stream of updates
  - into an analytics friendly schema
  - cleaned up
  - and loaded into the data warehouse
- don't *need* a separate data warehouse system but it can be useful -- can optimize the warehouse for analytics workflows
  - SQL is generalyl a good fit for both!
  - some products like MSFT SQL Server and SAP Hana support both transaction processing and data warehousing

- Data warehouse vendros like Teradata, Vertica, Redshift
  - and a bunch of SQL-on-Hadoop projects -Apache Hive, Spark SQL, Cloudera Impala, Fb Presto....
    - take ideas from Google Dremel

Analytics Schemas

- typically the *star schema*
- a *fact table* with all the events that occurred at a particualr time
- then *dimension* tables with further information on the when what who where why how of each event
- typically join in dim tables into the fact table when you need something

Column Oriented Storage

- Although fact tables are often 100s of columns wide, most analytics queries only need to access a few columns at a time! and ignore all the other columns
- in tmost OLTP databases, data is laid out in a *row oriented* fashion -- all the values from one row on a table are stored next to each other
  - document dbs are similar -- an entire document is stored continuouisly
- the idea behind column oriented storage is simple -- dont store all the values from a row together, but all the values from a column together
  - if each column is stored in a separate file, a query needs to only read and parse those columns that are used in that query, which saves a lot of work
- each column file contains the rows in the same order
- compression via bitmap encoding -- bitmap of 1/0 to indicate which row a column value is in
  - see Abadi paper for overview of compression mechanisms

Cassandra and HBase have *column families* concept - but this is not column-oriented -- within each column family, they store all columns from a row together, with a row key -- this is similar to Bigtable

Sort orders for columns based on indexes

- C-Store uses a different sort order index for each replica of the data!

Writing to column oriented storage

- Column oriented storage makes reads quicker in a data warehouse settings
- but it can make writes more difficult (since we still need to write entire rows, meaning you need to touch every column file with the update)
- An update-in-place approach like B-trees is not possible with compressed columns

    - to insert a row in the middle of a sorted table, you need to touch and rewrite all the column files consistently, as a row is identified by its position in the column store
- However, you can use LSM-trees!
    - all writes go to an in-memory store where they are added to sorted structure, then prepared for writing to disk -- doesnt matter if the in-memory store is row or column oriented
    - when enough writes have accumulated, they are merged with the column files on disk and written to new files in bulk

Data cubes = grid of aggreagates groups by different dimensions