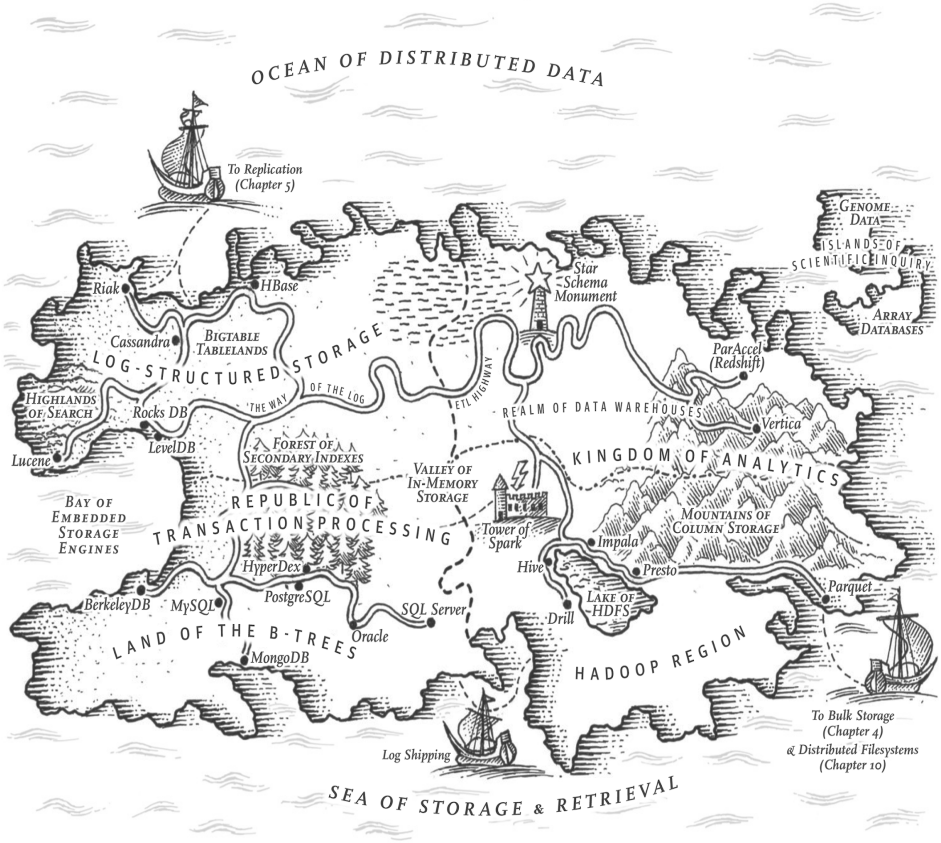
Designing Data-Intensive Applications

Chapter 3: Storage and Retrieval



Introduction

- As a motivating example, the author shows that a simple database that simply appends key-value pairs to a *log*
    - Note that the log is append-only. To update a key, you append a new entry, and the previous entry is supposed to be ignored now.
    - This system will have very fast writes, but linear performance on reads, which is pretty bad if you have millions, billions, or more entries.
- Additional data structures to help you find things for reads are referred to as *indexes*
    - An important trade-off in storage systems is good indexes speed up reads, but every index slows down writes

Hash Indexes

- Key-value stores can be implemented with a hash map (a.k.a. hash table) just like the dictionary type in Python and other languages.
- The hash map lives in memory and points to the offset in the log file on disk where the key-value entry is located

- This is enough to build a viable system as long as all of your keys fit in memory: writes are still append-only in one location and are fast, and hash map lookups are roughly constant time so are very fast
- More realistically, you probably want to keep your log file from consuming crazy amounts of disk space
    - Break your log file into segments every so often
    - In the background perform compaction where you eliminate duplicated keys, and optionally merge compacted log files together
    - Now, reads require sequentially looking for the key in the hash map for each log file segments, from most recent to oldest
        - For this reason, you want compaction to keep the number of log segments small
- Other details that need to be handled in the real world include deleting records, crash recovery, and concurrency control

SSTables and LSM-Trees

- A limitation of hash indexes is that they are fast for reading a single key, but not a range of consecutive keys
- A *Sorted String Table* (SSTable) differs from the above simple hash indexed log files in that each file segment is in sorted order. Sorted files support reading ranges.
- A *Log-Structured Merge-Tree* (LSM-tree) uses two (or more) tiers, with the last tier being SSTables
    - We save up a bunch of records in memory (the initial tier) before writing them to a file segment in sorted order
        - Several options to manage sorted records in memory, such as red-black trees, which is then called a *memtable*
    - Instead of a full hash map per segment, we can have a partial index of offsets
        - Partial indexes take advantage of the sorted order of file segments, so keys near each other in the sort order will be near each other in the file
    - Merging file segments is a fast linear time, minimal memory algorithm
- Individual writes are as fast or faster than hash indexes. Reads have a little more overhead, but likely to have similar performance.
- Note that now we have two background processes: in addition to segment compaction, we have periodic writing of batches of the memtable to a new file segment.
- Optimizations include:
    - Using *Bloom filters* to identify most keys that don't exist
    - Size-tiered compaction simply merges newer and smaller segments into older and larger SSTables
    - Leveled compaction raises the threshold for compaction as the segments get older and bigger
    - *Clarification:* for the above two compaction strategies, "older" refers to the age of the original records inside the segments, not how recently the file was compacted
    - Breaking up the key range into a collection of sub-ranges, with each sub-range having its own set of segments

B-Trees

- B-trees are a general form of balanced trees (the “B” comes from the word balance)
- Unlike log-structured storage, B-trees do not create duplicate entries; instead they allow deletes and updates in place
- B-trees break the database into fixed size *pages*, and store the pages on disk in a tree structure
- If you’re familiar with trees, both reads and writes are logarithmic time. When the branching factor *b* is in the hundreds, the log base *b* of ten billion is still under four.
    - Tree searches start at the root
    - Each page breaks up the key range into up to *b* smaller sub-ranges
    - You repeatedly follow pointers/references to sub-ranges until you get to a leaf
- B-trees are guaranteed to stay balanced, however occasionally a node split happens (which can also cause additional node splits up toward the root). This adds cost to the average write, but for shallow B-trees the cost doesn’t add much.
- Making B-trees reliable
    - Writing in place and page splits are dangerous operations to be interrupted by a server crash
    - Most B-tree implementations use a *write-ahead log* (here’s that append-only log data structure again!) on disk. Modifications are written here prior to updating the B-tree
- Concurrency control is needed, typically with lightweight locks called *latches*
- Optimizations include:
    - Abbreviating long keys, especially in the interior of the tree
    - Laying out tree pages in sequential order, in larger blocks, or using page maps
    - Additional pointers in the tree, such as left/right sibling pointers

Comparing B-Trees and LSM-Trees

- B-trees are a mature, well established technology providing good performance across a range of workloads, but LSM-trees have some interesting properties
- In particular, LSM-trees are typically faster for writes, so are interesting for write-heavy applications
- Conventional wisdom is B-trees are faster for reads, but may depend on workload
- Advantages of LSM-trees include:
    - Generally less write amplification - SSTable compaction versus B-trees writing entire pages even if only one record changed
    - LSM-trees can be compressed better and don’t suffer as much internal fragmentation as B-trees
- Downsides of LSM-trees include:
    - Background compaction processes can sometimes interfere with/slow writes
    - In the extreme, if write activity is high enough, compaction may not be able to keep up. If number of SSTables grows, performance will degrade. Typically, administrators will have to explicitly monitor for this scenario.

    - Transactional functionality (covered in chapter 7) is easier to implement with B-trees given that each key exists only once, and locks are straightforward to implement on ranges of keys.

Other Indexing Structures

- RDBMS often allow one clustered index per table, where the records are stored in the leaves of the index tree
- Secondary indexes can supplement the primary key
    - Secondary indexes usually don't contain the data, just a reference to the location
        - Actual record storage can also be in a heap file, which is unordered
- Multi-column indexes are also used, sometimes as covering indexes
- Other kinds of indexes include full-text and spatial (multi-dimensional) indexes

In-memory databases

- With RAM cost decreases, in-memory databases such as Memcached become viable
- Author states that the performance advantage of in-memory databases is not primarily due to avoiding slow reads from disk, but rather because they avoid the overhead of encoding data for writing to disk
- The *anti-caching* approach evicts LRU data from memory to disk without the overhead of durable on-disk data structures, akin to OS swap files

Transaction Processing or Analytics

- Early days of business data processing was dominated by commercial transactions, leading to the term *online transaction processing* (OLTP)
- Transactional workload typically see small numbers of records being looked up, and mostly single records inserted or updated based on user input
- Patterns for analytics (usage of term OLAP has diminished) differ in the common case being reads across large numbers of records, and often aggregate numbers (e.g. sum of revenue) being calculated. In the past, majority of writes might be bulk loads, but streaming becoming more common

Data Warehousing

- To guard low latency performance of OLTP systems, organizations started creating separate systems called data warehouses for analytic workloads
- A read-only copy of data was extracted from OLTP systems, cleaned & made consistent, and loaded into the data warehouse, using what came to be known as the *Extract-Transform-Load* (ETL) process
- The data model for data warehouses was relational for a long time, but divergence to other models such as Hadoop has occurred, independent of whether SQL is still the query language

Stars and Snowflakes: Schemas for Analytics

- Many data warehouses use a *star schema*, also known as *dimensional modeling*

    - At the center of the schema is a fact table, with *facts* representing individual events, usually very wide with lots of columns
    - Facts have event-specific data elements in some columns, but foreight key references (e.g. product ID) in others
    - The other, typically smaller, tables surrounding the fact table are *dimension tables*
- A visual diagram of dimension tables around a central fact table, often with relationship arrows, looks a lot like a star
- If dimensions are stored more normalized, e.g. countries having subdimensions for regions, and possibly regions broken into states, then the addition branching makes the diagram look more like a snowflake, thus the term *snowflake schema*
- A data warehouse may have multiple fact tables, thus multiple stars/snowflakes

Column-Oriented Storage

- Data warehouses are common in large organizations, so it becomes a challenge how to optimize queries against fact tables with billions or trillions of rows
- Taking advantage of the fact that typically each analytic query only needs a small number of the many columns in a fact table, you can reduce reads by physically storing chunks of column data together, instead of physically storing chunks of records (rows)
- Columns from the same fact table must all store the rows in the same order
- Column compression provides additional benefit:
    - The number of distinct values in a column is usually small compared to the number of rows
    - You can use bitmap encoding (a form of one-hot encoding) and run-length encoding (RLE) to shrink the amount of space needed to store a column's contents
- Vectorized processing improves performance:
    - Modern CPUs have complex instruction sets, including ability to perform simple SIMD operations like AND and OR against L1 cache much faster than explicit loops over individual data elements
- Sort order in column storage
    - Just as vanilla log files are append-only in no particular order, but SSTables are sorted, you can sort column store data
    - Sorted order will help with compression of the sort key columns
    - C-Store implemented replication with different replicas having different sort orders. In the common case, you can choose a particular replica if its sort order matches the range criteria of your query
- Writes to column stores are more complex
    - As with LSM-trees, we can have a two-level structure where batches of data are accumulated in memory, then periodically merged and written to disk. The sort order and compression work is batched behind the scenes.

Aggregation: Data Cubes and Materialized Views

- If you know certain aggregations are going to be frequently requested, e.g. sum of sales per department per store per day, you can save a physical copy of that data in a *materialized view*
- A data *cube* stores not just one view, but rather aggregations by multiple dimensions, e.g. all totals of sales by department, totals by store, totals by day, totals by department & store, totals by department & day, totals by store & day, and totals by department & store & day. If you visualize three dimensions, as in this example, you arrange all of the totals by department & store & day in a cube. Two-element subtotals would be sums by row, column, or stack. One-element subtotals would be sums across squares of numbers.

Summary

- This chapter focused on relational versus log-structured storage
    - Relational does update in place, keeping only one copy of each record, but requiring random access (aggregated in pages)
    - Log-based concentrates writes into fewer, sequential chunks, but requires multiple data structures and obsolete copies of records can complicate more advanced use cases
- Analytic workloads were discussed as a special case which can be optimized
- Note that this chapter focused on the options for vertical scaling. When we get to Part II of the book, issues around horizontal scaling can be introduced.

SQLite - A simple case study

- Simple RDBMS
- Entire database is stored as a single file
- Supports very limited concurrency
    - File is locked when any write is performed, and all other actions must wait
    - Multiple reads can be performed concurrently
- Current version supports full text indexes and spatial indexes
- Data storage is pretty straightforward:
    - Tables are B-trees
    - Indexes are B-trees
    - The `sqlite_master` table lists all objects
- Using the word “index” loosely to describe other metadata, other indexes maintained by SQLite add cost to the average write, and sometimes provide other performance gains
    - Freelist pages
    - Pointer map pages - backward pointers (to parent/predecessor) used for defragmentation
    - B-tree page headers
    - B-tree pointer array - the tracking structure of cells (keys, pointers, and payload) within a B-tree page

- A `sqlite_sequence` table - autoincrement tracking table
- Statistics tables used by the query planner