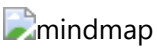
Chapter 3 - Storage and Retrieval

💡 when you give it some data, it should store it, and when you ask it again later, it should give the data back to you

Mind Map

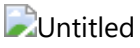


Data Structure

- log-structured storage engine vs page-oriented storage engine (B-trees)
- for an append-only log, update will just append to the end of the file so we need to look at the **last occurrence** for the latest data
- append-only log is very efficient for writes, but read will perform terribly - O(n), we need **one or more index.** Index is like additional metadata on the side, which acts as a signpost and help locate the data quickly. Indexes brings write **overhead** because it need to be updated for every write as well. we need well-chosen indexes (**often related to the query patterns**).

Hash Indexes

- keep an in-memory hash map where every key is mapped to a byte offset in the data file
- all keys need to fit in the available RAM
- it is great if the value for each key is updated frequently for a limited number of keys (so that they can fit into RAM) - leader board?
- logs (values) get large and we can break it into segments and perform **compaction (only keeping latest values)** later, we can also **merge** smaller segments into one at the same time. This should be done via **background thread** so that read/write are **not blocked**.

- each segment has its own in-memory hash table, we should check from the **most recent one.**
- file format: it's faster to use (length + raw content)
- records marked for deletion are also appended (**tombstone**) and are discarded during merging
- in-memory hash table can be lost when **crash**, we can
  - **rebuild** hash table using all segments (reverse engineering)
  - **persist snapshots** of hash tables on disk frequently
- use **checksums** to detect **partially written records**
- appending **order** is important so only **one thread** is allowed to write at a time.
- append-only log is better because

    - faster since sequential writes are faster than random write, especially in disks
    - easier concurrency control and crash recovery since the file is immutable

- append-only log has limitations

    - hash table must fit in memory
    - range queries are not efficient

SSTables and LSM-Trees

> Sorted String Table & Log-Structured Merge-Tree

- SSTable: when the **keys** are **sorted** and each key only appear once in each merged segment (compaction process ensures this)
- SSTable is better because:
    - merging is simple and efficient: two pointers, Can be performed even the file is bigger than RAM. when same key appears in both segments, always go with the later one
    - we do **NOT** need to keep the **entire** hash table anymore, just some sparse indexes for reference then we can perform binary search.
    - we can compress blocks of data before writing to disk to reduce I/O bandwidth
    - range query is efficient
- We keep an in-memory tree (red-black or AVL) called memtable (sorted), when it's getting larger than a threshold, we persist it into disk as a new and latest SSTable. background merging and compaction is still needed.
- We still need a separate WAL just for disaster recovery if the memtable has not been persisted yet.
- performance optimization:
    - it can be slow when searching for something does not exist: use bloom filter
    - determine the order and timing for compaction - **size tiered and leveled**

B-Trees

- B-trees break the database down into fixed-size blocks or pages, traditionally 4 KB in size (sometimes bigger), 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.
- leaf node contains the value or a reference for the value.
- The number of references to child pages in one page of the B-tree is called the branching factor, typically it is several hundred.
- update will locate the key update the value only, the reference remains unchanged
- adding new key requires finding 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, it is split into two half-full pages and the parent page is updated to account for the new subdivision.
- This algorithm ensures that the tree remains balanced: a B-tree with n keys always has a depth of O(log n).
- A four-level tree of 4 KB pages with a branching factor of 500 can store up to 256 TB. 500^4 & 4000B = 256*10^12
- Still need WAL to make the database resilient to crashes
- it's slower than SSTables since update is a real update physically
- Need latch (light-weighted lock) for concurrency control

💡 Locks and latches have different scopes and lifecycles. locks are a transaction synchronization mechanism while latches help synchronize processes or threads.

- optimizations:
    - use a copy-on-write scheme instead of WAL, update will create a new page and change reference
    - abbreviate key just for range
    - trees are sparse, left/right pointers can be used for range query without jumping back to parent pages.

Comparing B-Trees and LSM-Trees

- LSM-Trees are generally faster for writes and B-Trees are faster for reads
- B-trees write every piece of data at least twice - WAL and tree page and we have to write the entire page/block. LSM allow batch writes which is slightly better and also lower write amplification
- LSM compress better - smaller overall storage needed
- compaction in LSM can interfere with the performance, if the rate of compaction cannot keep up with write, unmerged segments will grow until you run out of disk space and reads will be slow since there are more segments to check.
- each key only appears once in B-Trees so transaction isolation is easier to achieve.

Other Indexing Structures

- It is also very common to have one or more secondary indexes
- keys are not unique for secondary indexes. we can either
    - by making each value in the index a list of matching row identifiers
    - by making each key unique by appending a row identifier to it.
- query by more than one key
    - concatenated index
    - multi-dimensional indexes: select * from XXX where .... AND .... AND....
- full-text search and fuzzy indexes: Lucene used inverted index stored with SSTables
- Some DBs are intended for cache only and they are small enough to fit in memory, or support persist function (like Redis)
- in-memory DB can implement data structure that is closer to the ones in programming languages, like Redis sets and PQ.

💡 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.

- we should use disk together with it as an append-only log for durability

Transaction Processing or Analytics?

- **OLTP**: An application typically looks up a small number of records by some key, using an index. Records are inserted or updated based on the user’s input.

- **OLAP**: Usually an analytic query needs to scan over a huge number of records, only reading a few columns per record, and calculates aggregate statistics (such as count, sum, or average) rather than returning the raw data to the user.
- a typical OLAP query only accesses a few (4 or 5) of hundreds of columns of a table at one time

Data Warehousing

- A data warehouse is a separate database that analysts can query to their hearts' content, without affecting OLTP operations. The data warehouse contains a read-only copy of the data in all the various OLTP systems in the company.
- Data is extracted from OLTP databases (using either a periodic data dump or a continuous stream of updates), transformed into an analysis friendly schema, cleaned up, and then loaded into the data warehouse. - ETL
- OLAP system can be optimized for analytic access pattern, which can be different from OLTP
- If the same aggregates are used by many different queries, it can be wasteful to crunch through the raw data every time - Materialized Views/Aggregates.

Stars and Snowflakes: Schemas for Analytics

- dimensional data modelling:
  - fact tables - individual events
  - dimension tables - who, what, where, when, how, and why of the event
  - joined via surrogate key
  - sub-dimension - snowflake
- Snowflake schemas are more normalized than star schemas

Column-Oriented Storage

- most OLAP query wants to access **some** columns for **all** rows
- In most OLTP databases, storage is laid out in a row-oriented fashion
- The idea behind column-oriented storage is simple: don't store all the values from one row together, but store all the values from each column together instead.
- column-oriented storage is very good for compression. - bitmap encoding
- Data Cubes and Materialized Views

Summary

- OLTP systems are typically user-facing, which means that they may see a huge volume of requests. In order to handle the load, applications usually only touch a small number of records in each query. The application requests records using some kind of key, and the storage engine uses an index to find the data for the requested key. Disk seek time is often the bottleneck here.
- Data warehouses and similar analytic systems are less well known, because they are primarily used by business analysts, not by end users. They handle a much lower volume of queries than OLTP systems, but each query is typically very demanding, requiring many millions of records to be scanned in a short time. Disk bandwidth (not seek time) is often the bottleneck here, and column-oriented storage is an increasingly popular solution for this kind of workload.

- OLTP:
    - log structed
        - LSM-trees, SSTables
    - update in place
        - B-tree