Storage And Retrieval

Arnav Gupta

January 27, 2025

Contents

1 Introduction

Knowing how a DB handles storage helps when choosing an appropriate storage engine.

2 Data Structures that Power your Database

Log: an append-only sequence of records

To efficiently find the value for a particular key, an **index** can be helpful. Indexes are derived from the primary data.

Indexes are additional to the primary data, which has overhead. Indexes speed up reads but slow down writes.

Application developers should choose indexes manually based on the application's typical query patterns.

2.1 Hash Indexes

Key-value stores are similar to dictionaries, which are usually implemented as hash maps.

Can index a key-value store using a hash map to store the byte offset in the data file where the value can be found.

On writes, write to the hash map as well (based on where data was inserted).

On reads, read from the hash map to get the offset where the record is.

To avoid running out of space, break log into segments and close a segment file when it reaches a certain size. Perform **compaction** on segments, throwing away duplicate keys in the log and merging segments if possible. Since segments are never modified after writing, merged segments are written to a new file.

Each segment has an in-memory hash table, mapping keys to file offsets. Hash maps are checked in order of recency.

Having logs be append-only helps with speed, concurrency, crash recovery, and file fragmenting.

Hash table must fit in memory, so cannot have to many keys. Also, range queries are not efficient (lookups are independent).

2.2 SSTables and LSM-Trees

In an **SSTable**, sort the key-value pairs in each segment by key.

This allows simpler merging, removes the need to keep an index for all keys in memory, and allows the use of blocks of key-value pairs for compression.

2.2.1 Constructing and Maintaining SSTables

Can use a tree data structure (like red-black tree or AVL tree) so maintain sorted order. This structure is maintained in memory, in a **memtable**.

When the memtable reaches some size threshold, it is written to disk, becoming the most recent segment.

Only issue is with crashes (since memtable in-memory). Can solve with a log on disk to which every write is appended, used to restore the memtable after a crash.

2.2.2 Making an LSM-Tree out of SSTables

This indexing structure is called an **LSM-Tree**.

2.2.3 Performance Optimizations

Use **Bloom filters** to optimize checking if keys do not exist in the DB.

Types of compaction include:

- **size-tiered**: newer SSTables successively merged into older and larger SSTables
- **leveled**: key range split up into smaller SSTables and older data moved into separate levels

LSM-Tree works well even when data is bigger than available memory. Also better at range queries since data is sorted.

2.3 B-Trees

Most widely used indexing structure.

DB broken down into fixed-size blocks/pages. Design corresponds to underlying hardware.

Root of B-Tree is a page and contains keys and references to child pages. Keys are a continuous range, indicating where boundaries between ranges lie.

A leaf page contains the values or references to the values.

Branching factor: number of references to child pages in 1 page of the B-tree

If not enough free space in a page to accommodate a new key, it is split into two half-full pages. This ensures the tree remains **balanced**.

A B-Tree with n keys always has depth of $O(\log(n))$.

2.3.1 Making B-Trees Reliable

B-Tree write overwrites a page on disk with new data, but overwrite does not change the location of the page.

To make the DB resilient to crashes, a **write-ahead log** is used, which 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.

For concurrency, **latches** are used to protect the tree's data structures.

2.3.2 B-Tree Optimizations

Some DBs use a copy-on-write scheme where a modified page is rewritten to a different location and pointers are updated.

The key can be abbreviated to save space.

Some B-tree implementations place leaf pages in sequential order on disk.

Some B-trees have sibling pointers.

Fractal trees reduce disk seeks.

2.4 Comparing B-Trees and LSM-Trees

LSM-trees are faster for writes, B-trees are faster for reads.

2.4.1 Advantages of LSM-Trees

Less writes required due to **write amplification**: one write to the DB results in multiple writes to the disk.

Sequentially write compact SSTable files rather than overwriting several pages in the tree.

Better compression and less fragmentation due to compaction.

2.4.2 Downsides of LSM-Trees

Compaction can interfere with the performance of ongoing reads and writes.

Write bandwidth must be shared between the initial write and compaction threads running in the background.

It is possible that compaction cannot keep up with the rate of incoming writes, so there are too many segment files.

B-trees are better for transactions.

2.5 Other Indexing Structures

Key-value relies on **primary key**. **Secondary indexes** are also possible.

A secondary index can be constructed from a key-value index, but keys are not unique.

2.5.1 Storing Values within the Index

Heap file: where rows are stored, in no particular order and no duplicated data

Indexes just reference a location in the heap file.

Records can be overwritten in place in the heap file.

Clustered index: store indexed row directly within an index

Covering index: stores some of a table's columns within the index

Clustered and covering indexes speed up reads but add overhead on writes.

2.5.2 Multi-Column Indexes

Concatenated index: combine several fields into one key by appending one column to another

B-tree and LSM-tree indexes don't answer multi-dimensional queries efficiently. R-trees are better for these.

2.5.3 Full-Text Search and Fuzzy Indexes

Fuzzy querying: searching for similar keys

Can use a SSTable-like structure or use a trie.

Levenshtein automaton: supports efficient search for words within a given edit distance

2.5.4 Keeping Everything in Memory

As RAM gets cheaper, this has led to **in-memory databases**.

Some used for caching, some more durable.

Writing to disk is still done for durability, but data is loaded from memory.

In-memory databases are faster because they avoid overheads of encoding in-memory data structures in a form that can be written to disk.

Anti-caching allows datasets larger than available memory using a scheme similar to virtual memory.

3 Transaction Processing vs Analytics

Transaction processing: low latency reads and writes

Data analytics have different access patterns than transactions.

Data warehouse: separate database to run analytics on

3.1 Data Warehousing

Using a data warehouse does not affect OLTP operations since it is read-only and can be processed through ETL transformations.

Data warehouse can be optimized for analytic access patterns.

3.1.1 Divergence Between OLTP DBs and Data Warehouses

OLTP and data warehouse have SQL query interface, but internally are optimized for different query patterns.

3.2 Stars and Snowflakes: Schemas for Analytics

Star schema: has fact table that represents events at times and dimension tables that describe events

Snowflake schema: variant of star schema where dimensions are further broken down into subdimensions (more normalized)

4 Column-Oriented Storage

Typical data warehouse query only accesses a small fraction of columns at a time.

With **column-oriented storage**, all values from a column are stored together rather than all values from a row. This can be helpful for fact tables where indexes only exist on a small number of columns.

Relies on each column file containing rows in the same order.

4.1 Column Compression

Can compress column representation with **bitmap encoding**, where each distinct value has a bitmap. This is well suited for the queries common in data warehouses.

4.1.1 Memory Bandwidth and Vectorized Processing

Bottlenecks include getting data from disk into memory, efficiently using bandwidth from main memory into CPU cache, avoiding branch mispredictions, and making use of SIMD instructions.

With column compression, more rows from a column fit in the same amount of L1 cache.

Vectorized processing: operators designed to operate on such chunks of compressed column data directly

4.2 Sort Order in Column Storage

Order doesn't necessarily matter.

DB admin can choose the columns by which a table should be sorted.

Sorted order can help with column compression, since compression is strongest on first key.

4.2.1 Several Different Sort Orders

Can store redundant data in different ways to use the version best fit to a query.

4.3 Writing to Column-Oriented Storage

Writes are more difficult in column-oriented storage.

LSM-trees can be used (with compaction) since update in-place will not work.

Queries need to examine column data on disk and recent writes in memory.

4.4 Aggregation: Data Cubes and Materialized Views

Materialized view: copy of query results, written to disk

When underlying data changes, a materialized view needs to be updated.

Data cube: grid of aggregates grouped by different dimensions

With a materialized data cube, certain queries become very fast since they have been precomputed.

Data cube doesn't have the same flexibility as querying raw data.