Designing Data Intensive Applications

CHAPTER - 3

Storage And Retrieval

How will data be stored in database?

How data will be retrieved from database?

Data Structures that power your Database

Why do I care what's happening in backend? ⇒ It helps in selecting a data storage for your system. There are lot of options these days 😊

Log file → Assume you're keeping a key value pair and store in text file in append mode.

For any new write query, you'll just append an entry.

How will read value for given key?

↳ Scan entire file for occurence of key? → that will be O(n) operation

↳ It's not efficient if data grows to large scale

↳ You'll need indexes (watch youtube video on Indexes on channel)

create indexes for data written.

↳ Should I create indexes for all the data? It will make read faster

→ NO, it will make write slower. Along with append operation, you also need to write data to indexes.

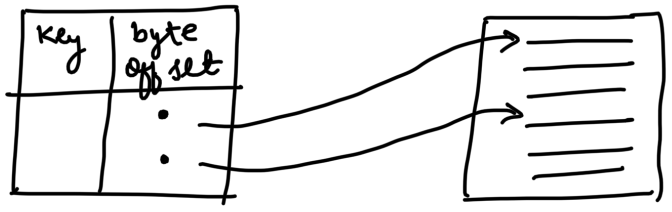
Hash Indexes → How will you index data on disk?

Assumption - Data storage provides only append operation on file

Indexing Strategy →

↳ Create in-memory hashMap.

↳ Every key is mapped to byte offset in file.



↳ It is used by Bit cask (Storage engine in Riak)

↳ In memory Hash map helps if there are limited no of keys.

↳ Example - maintain a counter how many times video is played.

↳ How will you ensure you're not running out of disk space (since you're only appending)

↳ ① break the log into segments of certain size.

↳ ② perform compaction - remove duplicate keys in log and only keep recent update.

V1:23	V2:25	V1:31
V3:73	V1:99	V2:50

→

V1:99	V2:50	V3:73

↳ Each segment has its own in-memory hash Table.

↳ Find value for key: check most recent segment's hash map.

↳ If not found, check 2nd most recent.

Implementation Details

① File format – Binary → encodes the length of string in bytes, followed by raw string.

② Record Deletion – append special deletion record to data file.
↳ called Tombstone {watch Discord Msg storage video}

③ Crash Recovery – Maps in memory will be lost in events of crash.
↳ keep snapshot of segment's hash map on disk.

④ Partially written records – use checksums, helps in identifying corrupted parts and delete them.

⑤ Concurrency – Data files are append only → immutable.
↳ can be read concurrently.

Why not update the key instead of appending?

① Much faster than random writes, especially on magnetic spinning disks.
② concurrency and crash recovery is simple.

Limitations

① Hash Table must fit in memory.
② Range queries are not efficient.

SSTables and LSM-Trees

optimizing the limitations mentioned above ↑
↳ change the format of how segment files are stored
↳ make sequence of key-value pair "sorted by key"
↑
This is called Sorted String Table (SSTable)

↳ Each key should appear only once in merged segment.
(handled in compaction process)

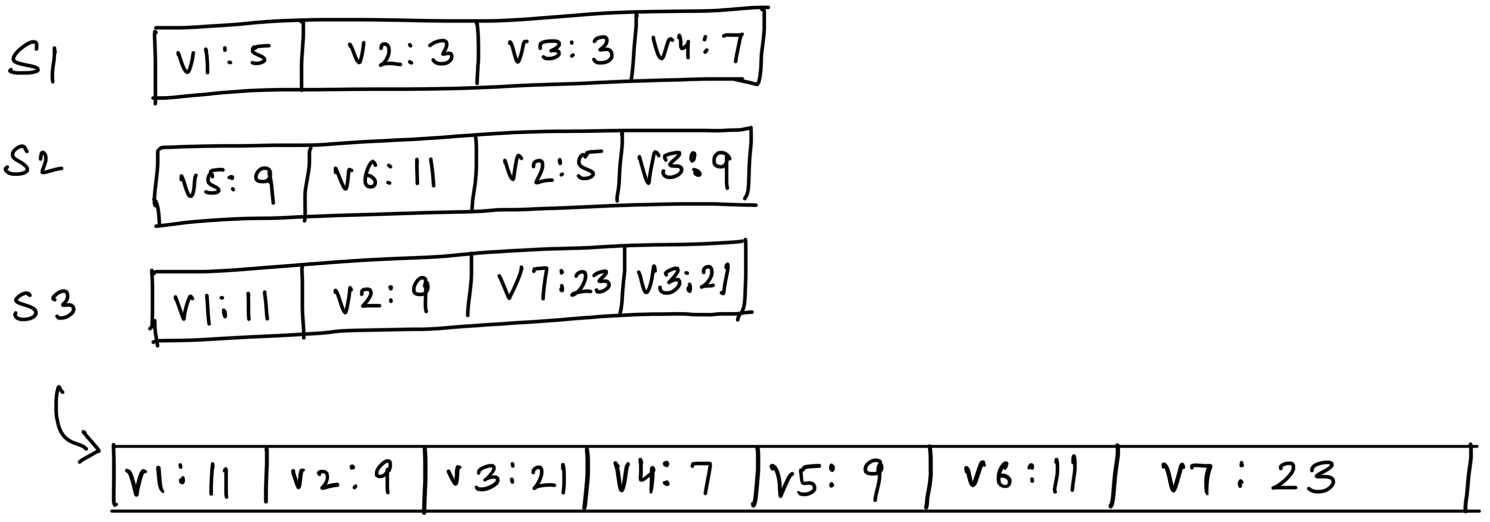
(A) Merging of segments → algorithm similar to merge sort.

① Read input file side by side.

② look for first key in each file.

③ Copy lowest key to output file & repeat again.

④ If same key in multiple segments → pick value from most recent segment.



Search for a key

↳ ① Inmemory index is not required for maintaining offset for key.

② maintain offset for some of keys (sparse index) and they traverse between these offsets to find particular key.

③ Group the records for keys you're maintaining in Inmemory index. ↳ saves disk space and reduce I/O.

Construction & maintenance of SSTables

For any write query –

① Add it to in-memory balanced tree structure. – called memtable.

② Write memtable to disk after certain threshold (say 1MB) – called SSTable

③ For read query, search in memtable, then most recent segment & so on.

④ Run merge & compaction time to time.

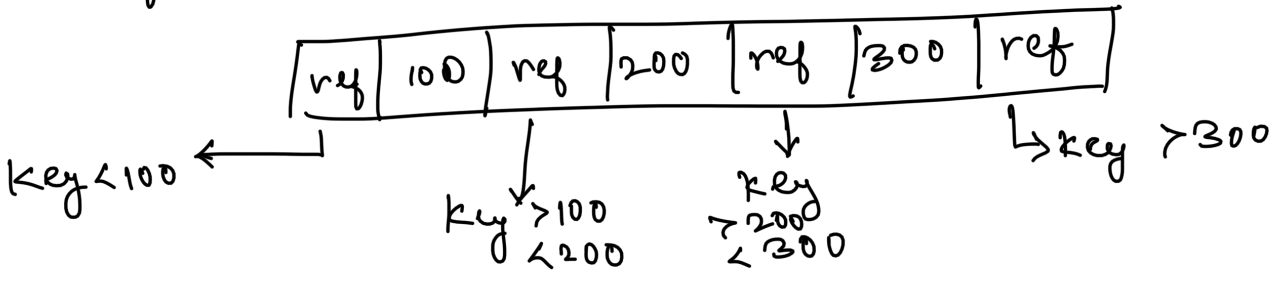
Performance of Database built on LSM Tree

① Reads can be slow as you traverse from one segment to another.
↳ Bloom Filters are used → data structure helpful in approximating contents of set.
Example → will tell if key is present in dB or not saving traversing DB.

② Compaction & merging → how to determine order & Timing.

Size-Tiered	leveled compaction
HBase. Cassandra uses both.	LevelDB, RocksDB
smaller SSTables are successively merged into larger SSTables.	key range is split up into small SSTable and older data moved to separate levels.

BTree → It is not "Binary" Tree.
↳ key value pairs sorted by key. → on disk
↳ BTree break database into fixed size blocks.
↳ Each Block can be identified by address, using which one page refer to another.



Handling Crashes in BTree → what if hardware crashed at time of any update?
It maintains additional data structure → WAL (Write ahead Log)
↓
append only file - every modification to BTree is written here before actual update.

Write amplification → one write to DB resulting in multiple writes on disk over course of DB's lifetime.

LSM Tree	BTree
① Typically faster for writes ② It can sustain higher write throughput due to less write amplification ③ Can be compressed better ④ Compaction process can affect ongoing read/write. ↳ more observable at high throughput.	① Typically faster for reads. ② More write overhead, update existing page. ③ leaves some space unused due to fragmentation. ④ Each key is present a single place. ↳ offer strong transactional support

Full Text Search and Fuzzy Indexes

↳ Lucene example ElasticSearch

ex. to search for mis-spelled words or synonyms.

In-memory Databases

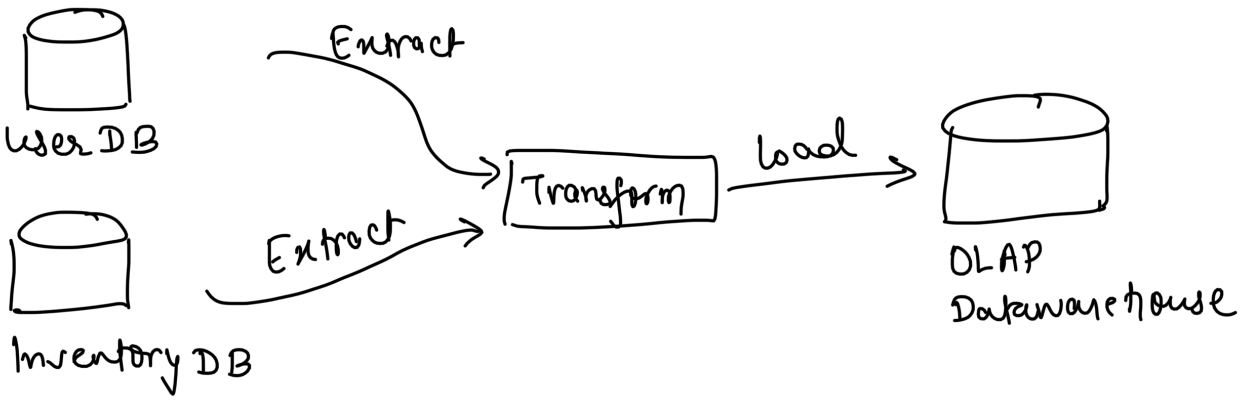
memcached → Indended for caching purpose, it is ok to loose data if machine is restarted.

↳ Inmemory databases for durability → dump periodic snapshots to disk.

↳ Redis and couchbase provide weak durability by writing to disk asynchronously.

Your use-case → Transaction Processing or Analytics ?

OLTP	OLAP
① Read - small number of records per query fetched by key ② write - Random access, low latency writes ③ GB to TB	① Aggregate over large set of data. ② ETL ③ TB to PB



OLTP databases

Column Oriented Data Storage

1. The table can be >100 columns wide but we might be accessing only 4 to 5 columns at a time.
2. To get the data, you need to load all rows matching criteria to these columns
3. Column Oriented → Store values from each column together instead of rows.
4. Column Compression is easier → there are chances that values in column are same for different rows.
5. Example – Cassandra, HBase
   - ↳ not strongly column oriented.
     - → they store all columns for row together along with rowkey.
     - → don't use column compression.

Subscribe for more such content.

YouTube channel- MsDeep Singh

Happy Learning ☺