Chapter 3 - Design scalable systems

Type	Designing Data-Intensive Applicaitons
Content	
Created time	@October 5, 2022 12:40 PM
Last edited time	@October 6, 2022 3:00 PM
Resources	
💼 Projects	

Storage and Retrieval

> If you keep thinks tidility ordered, you’re too lazy to go searching. - German Proverb

💡 On the fundamental level a database has to do two things. When you give it data, it should store it, and when you ask for that data back, it should give it to you.

💡 We should understand how databases work under the hood, as it will inform it’s ability to perform under different workloads, and use cases.

- In particluar, transactional queries vs analytical ones

✏️ **Log:** An append only sequence of records.

Indexing

✏️ Index: An additional structure that is derived from the primary data, that is used to enable efficient search.

💡 A common rule with indexes is that well chosen indexes can accelerate reads, but it also slow down writes.

- Simply appending data to a file is the quickest way to write, as it's the simplest write operation

Hash indexes

This is the index for key-value data. Key-value stores are usually implemented using hash tables.

💡 A hash table is a popular data structure used for key-value look up. When given a key, it will store it, and return a value.

The hash table will implement a hash function that converts the key into an integer and remaps into index. String → hash code → index. This is to lower the amount storage needed.

💡 A hash index works by keep an in-memory hash map (hash table) where every key is mapped to a byte offset in the data file, where the value can be found. Appending a key-value pair would update the hash map tup reflect the offset of the written data.

Segmentation

💡 **Segmentation** is the process of breaking the log into certain segments of a certain size, which will close when full, and write to the next segment. We will then perform **compaction**, by throwing away duplicate historical keys in each segments and keeping only the most updated ones in the compacted segment.

- Segments are immutable
- Segments can also be compacted and merged into a new segment.
    - This can be done in a background thread, so while this is going on, we can still serve read requests using the old segments

Implementation of hash indexes problems:

- File format: CSV may not be the best choice it is faster and simpler to store the data in binary format (sequence of 1s and 0s)
- Deleting records: Deletion is done by appending a delete record to the file, and will indicate during the mergin process to discard any values for the deleted key
- Crash recovery: If the database is restarted, in memory hash maps are lost. This can be solved by storing a snapshot of each segments hash map on disk
- Partially written records: If it crashes when being appended to a log, you can be left with corrupted logs. Checksums can mitigate this

- Concurrency control: Due to the sequential nature of logs, you can allocate writes to a single thread.

Advantages of append only design:

- Much faster due to sequential write operations
- Concurrency and crash recovery are easier due to segment immutability
- Merging old segments removes the problem of data files being fragmented over time

Disadvantages of append only design:

- Hash table must fit in memory
- Range queries are not efficient

> ✏️ **Thread:** A small set of instructions designed to be schduled and executed by the CPU regardless of parents processes.

> ✏️ **Data Fragmentation:** This is when data that is stored across multiple locations, which could cause major overheads if they are needed together.

SSTables

> 💡 The major difference for this database design is that the sequence of key-value pairs is sorted by key. Known as Sorted String Table.

- Cannot append new key-value pairs to the segment immediately as writes can occur in any order

SSTables advantages over log segments with hash indexes:

- Merging segments is simple and efficient. Look at the first key in each file, copy the lowest key according to the sort order to the output file an repeat. Creating a new merged segment file sorted by key.
    - When there's several duplicate keys across several segments we simply keep the value of the key from the most recent segment
- You do not need to keep all the keys in memory, you still need an in-memory index that keeps the offsets of some of the keys, but due to how it's sorted you can keep it sparse, and find values within a range.
    - Handicap is between handbag and handiwork

How do you sort your data before a write ?

> 💡 Mainting a sorted a structure in disk is possible, but maintaining it in memory is much easier.

- With the SSTable structure you can insert keys in any order and read them back in sorted order
- They will then be written to a on disk SSTable which is sorted

Problem: If the database crashes the most recent writes (still in memory) but not yet written to disk will be lost, we keep a seperate log on disk to which every write is appended.

Disadvantages of SSTables:

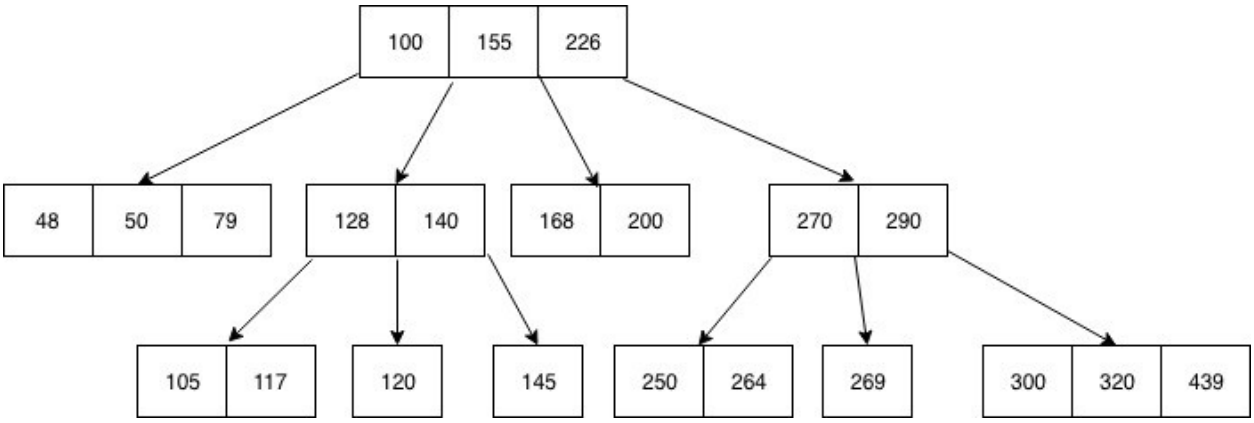
- Can be slow when looking up values that do not exist, you have the check the memtable and then all the keys on disk.

- [ ] Look up LSM algos

B-trees

> 💡 The most common and widely used indexing structure is the B-tree. They remain the standard index implementation in almost all relational and some non relational databases.

- Keep key-value pairs sorted by key
- While the log-structured indexes broke databases down into segments where writes and done sequentially, B-trees break databases down into fixed-size blocks or pages, and read one page at a time.
- This design is more in line with the underlying hardware, as disks are also arranged in fixed size blocks.



B tree visualized

💡 Each page can be identified using an **address** or location, which allows one page to refer to another on the disk. These page references can constructed as a tree of pages.

One page is designated as the **root** of the B-tree; whenever you want to look up a key in an index you start at the root, this page contains several keys and references to other child pages. The keys between references act as boundaries.

A **child page** is that contains individual keys is known as a leaf page, which either contains the value for each key inline or references to the individual pages where the values can be found.

✏️ **Branching factor:** The number of references to child pages in one page.

- Updating the value of an existing key consists of find the leaf page containing the key, change the values in that page and write page back to disk
- Adding a value includes adding the page within the range that encompasses it

⚠️ The algorithm ensures that the tree remains balanced: a B tree with n keys always has a depth of O(log n).

- The basic underlying write operation on a B-tree is to overwrite a page on disk with new data
    - It is assumed that the location of the overwritten page stays the same, which is in contast to LSM-trees, which only append to files.

☐ How do hard drives work?

> 💡 Write-ahead log is an append only file, in which every B-tree modification must be written before it can be applied to the pages of the tree, this ensures the database is resilient to crashes.

☐ What is concurrency?

- B-trees have been optimized over the years

B-trees vs LSM-trees

LSM trees are typically faster for writes, whereas B trees are though to be faster for reads.

Advantages of LSM trees:

- A B-tree index must every piece of data at least twice: once to the write ahead log, and once to the tree page itself. Also there is overhead from having to write an entire page at a time.
- LSM trees can sustain higher write thoughput, due to having lower write amplification
- LSM trees can be compressed better, and produce smaller files on disk than B-trees
- Faster on sequential writes, depends on hard drive

Downsides of LSM trees:

- Higher response time

Secondary indexes

Key-value indexes which are like primary key index in the relational model, is not the only index type.

> 💡 Secondary indexes allow you to perfom more efficient queries on other columns that are not necessary the primary key. CREATE INDEX command on SQL

Multi-column indexes

> 💡 Multi-column indexes allows you to query several columns at once, which is important for geospatial data. The index would translate a two-dimensioanl location into a single number, and then use a regular B-tree index.

- [ ] What is an R-tree, relevant to geospatial analysis

Fuzzy indexes

> 💡 All other indexes assume you have the exact data and allow you to query for exact values of a key. Fuzzy indexes allows you to search for words that are approximately similar.

- [ ] What is an in memory database, wand why are they superior

Transaction processing or Analytics?

Transaction queries

💡 In the early days a write to a database typically corresponded to a commercial transaction taking place. As databases expanded into areas that did not involve an exchange of money the term *transaction* stuck, referring to a group of reads and writes that form a logical unit.

✏️ Online transaction processing (OLTP): An access pattern of records being inserted and updated based on the user's input.

Analytical queries

💡 Databases however, also started becoming particularly used for data analytics, which has very different access patterns. Usually an analytical query scans over a huge number of records, only reading a few columns per record, and calculates aggregate statistics, rather than returning raw data.

✏️ Online Analytic processing (OLAP): An access pattern usually used for business intelligence including large queries scanning over many of records, across several columns.

Differences between transactional and analytical queries

Property	OLTP	OLAP
Read pattern	small number of records per query, fetched by key	Aggregate over large number of records
Write pattern	Random-access, low-latency writes from user input	Bulk import (ETL) or event stream
Data representation	Latest state of data	History of events
Dataset size	GB to TB	TB to PB

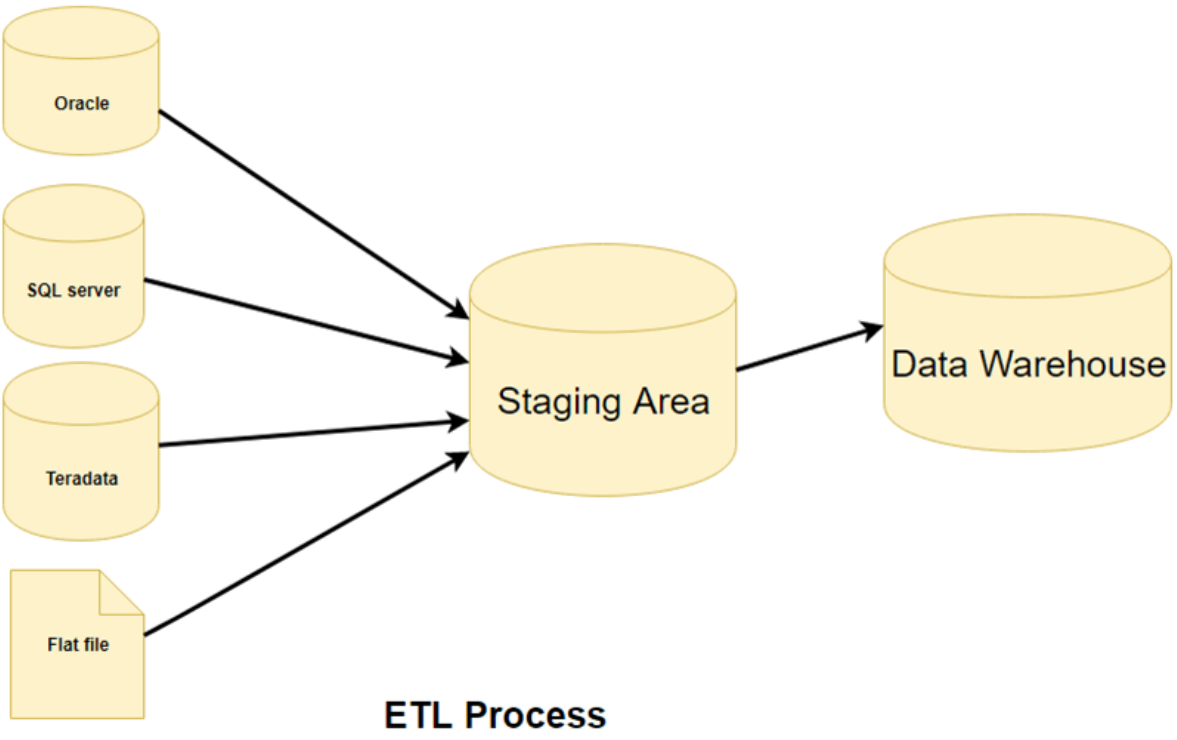
Primarily used by	End users, via web app	Internal analysts

Data warehousing

A data warehouse is a seperate database that analysts can query to their hearts content without affecting OLTP operations.

It contains a read-only copy of the data within a OLTP system in the company. Data is extracted from all OLTP database, transformed into an analysis-friendly schema, cleaned up and then loaded into the data warehouse.

This process is called ETL.



Outline of ETL into a data warehouse

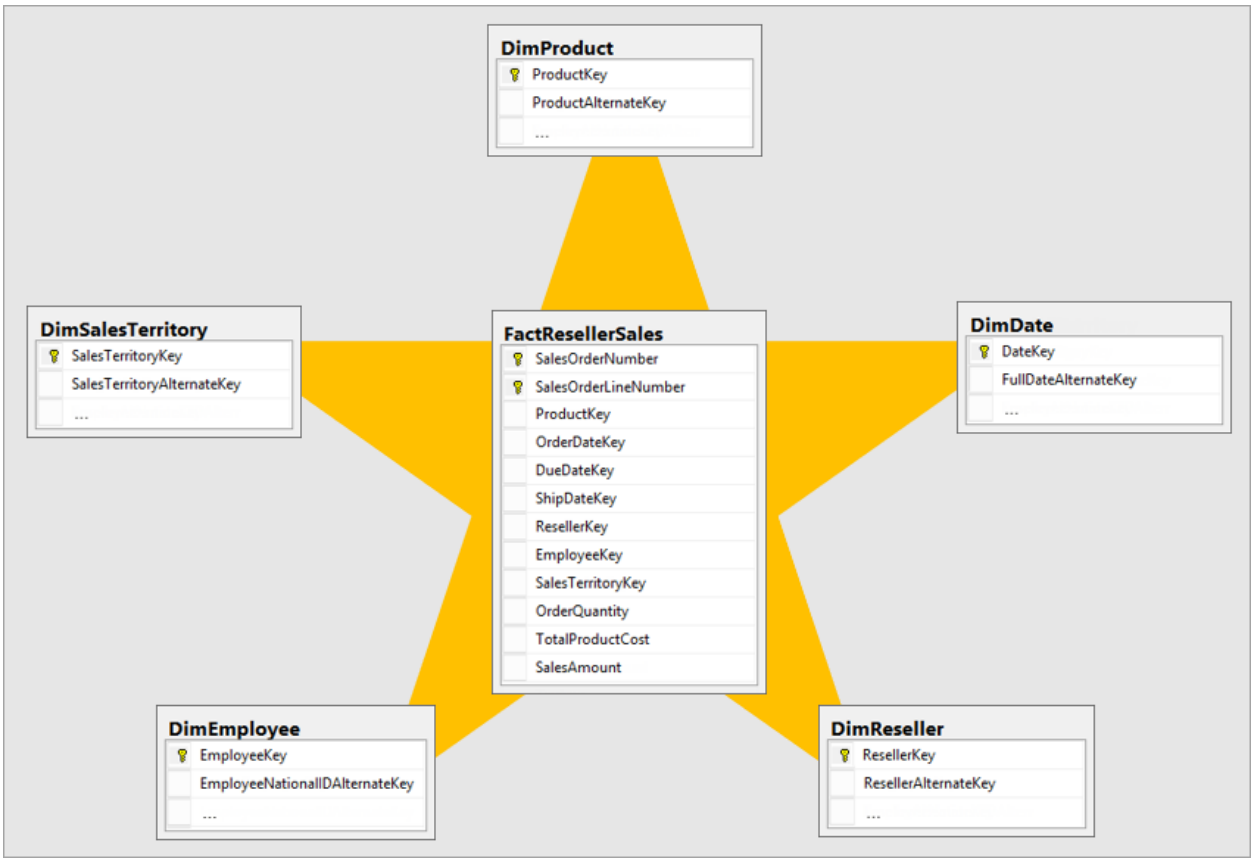
> 💡 The reason we use a data warehouse rather than querying OLTP systems directly for analytics, is that data can be optimized for analytic access patterns.

- Both a data warehouse and OLTP relational database look similar, because both have a SQL interface, but the internals look quite different.
  - Data warehouse would store column based data
  - OLTP system will store row based data

Schemas for analytics: Stars and Snowflakes

> 💡 There is a much less diverse range of schemas for analytical data models. Star schema and snowflake schema.

Star schema



Star schema

The star schema revolves around a fact table, each row in the fact table represents an event that occured at a particular time. Some columns in the fact table can be attributes but other columns are foreign key references to other tables called **dimension tables.**

Dimension tables: These tables represent the who, what, where, when, how and why of the event that occured in the fact table.

Snowflake schema

A snowflake schema is more normalized, but often analysts prefer the star schema as they are easier to work with.

Row based data vs column based data

The primary difference in row vs column based data is in how the data is stored, this will have an impact on the use cases of the data.

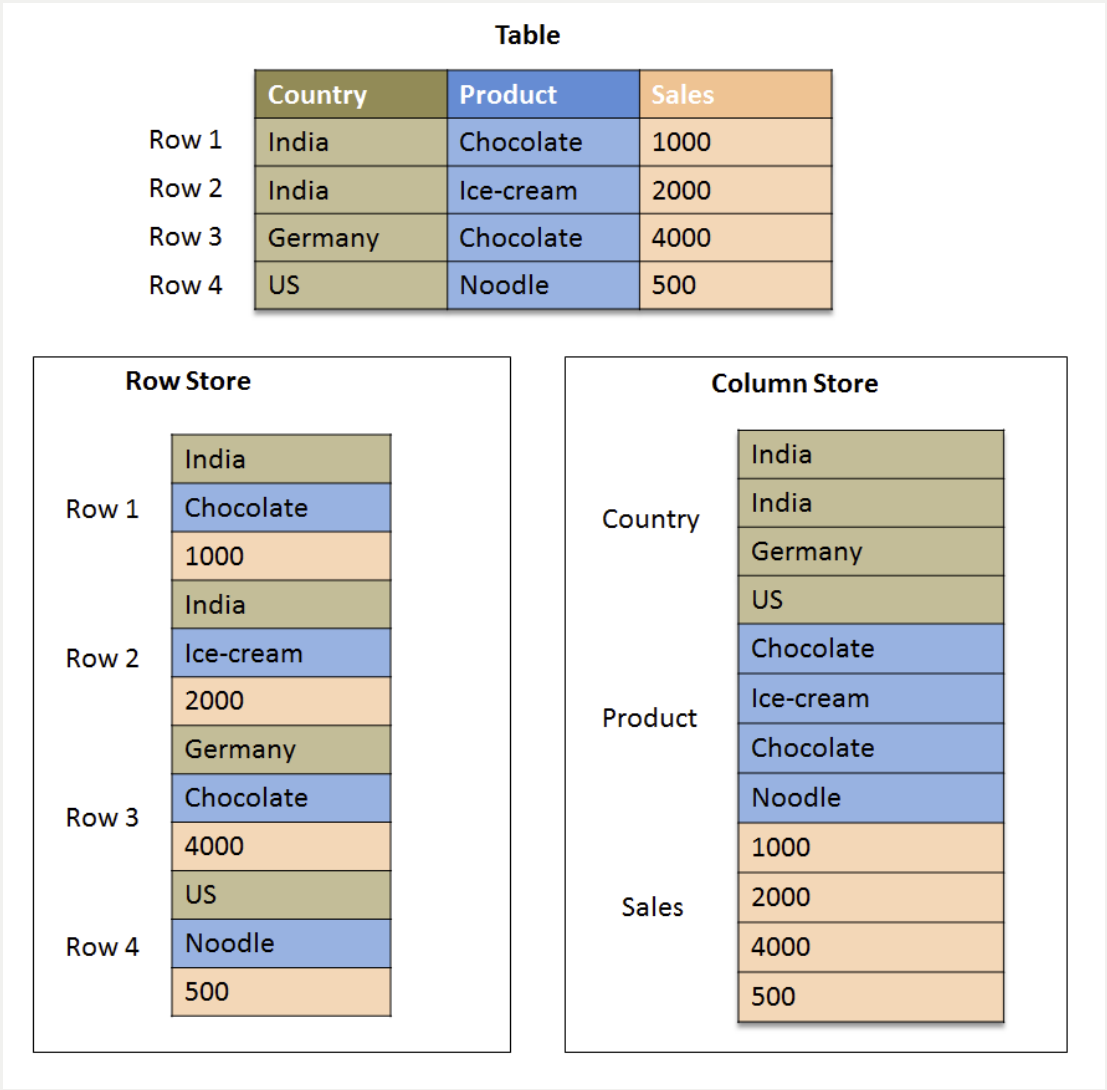


Figure 1: Row based data vs column based data, underlying storage

On the storage disk, data is grouped together within a series of blocks. It is cheap and quick for the computer to look inside an individual block to see the data, however it becomes more expensive and longer if a query requires the computer to check through several boxes. This is important to understand in highlighting the difference between row and column based data storage.

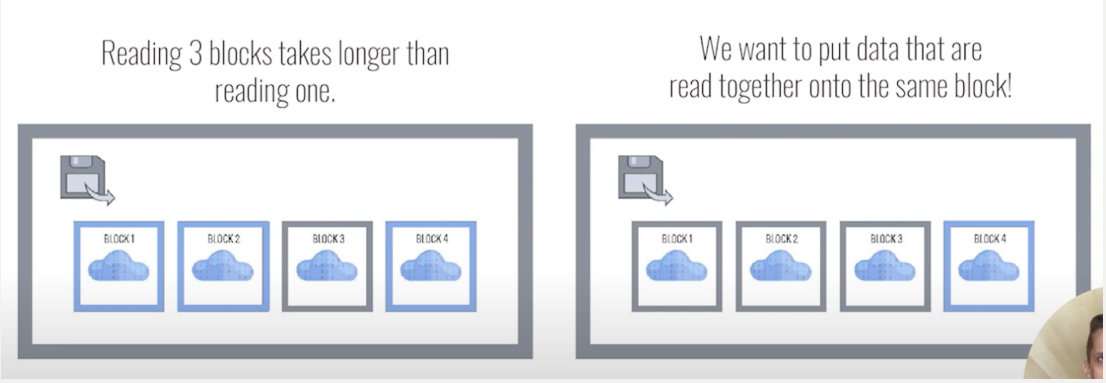


Figure 2: How data is stored on the disk.

Row based: As figure 1 suggests row based data is stored together under the hood based on the index of the row, all columns associated with that row's index is grouped together. For example: Under the hood, if we made an SQL query that wanted to get all the values relating to India, then in a row based data structure, that would be a quick process as all of that data is stored in the same block.

Column based: As figure 1 suggests columb based data is stored together under the hood based on the associated columns, and it the computer will only know which row it belongs to based on the values position within that list of column values. Example: If we made an SQL query that wanted to count the number of unique countries in the table, then that would be a quick process as all of the columns relating to a country is stored in the same block.

As we mentioned, queries that force the computer to read several blocks at a time take longer and are more expensive, and this basic principles explains why both types of data structures are optimized for different use cases.

Column based is better for analytical queries, where a lot of aggregation functions would be applied to column wide data.

```
SELECT COUNT(countries) from table;
```

Row based data is optimized for transaction based queries, where we might want to see all values associated to a single row.

```
SELECT * from table from where countries = 'india';
```

	Column based	Row based
Pros	- Only read in relevant data	- Easy to add data, as it just appends to the latest row
Con	- Harder to add data, as it needs to add to each block based on index	- Brings in unnecessary data

Column based data is also much better suited for compression, using bitmap encoding.

- [ ] What is bitmap encoding

Aggregation

Columnar databases can be much faster for ad-hoc analytical queries. Data warehouses can implement a materialized view known as a data cube or OLAP cube which is a grid of aggregates grouped by different dimensions.

Certain aggregate queries will become very fast, as they have been pre-computed.