System Design

This note summarizes some important ideas in the book *Design Data-Intensive Application*, written by *Martin Kleppmann*. It also includes some of my real world practice of software development.

Preface

Data Intensive	The quantity, complexity and changing speed of data.
Key Ideas	**Scalability**, **Reliability** and **Maintainability**
Focus	Architecture and ways of thinking about the data system
Outline	- Fundamental ideas underpin the system design - Distributed system across machines - Heterogeneous systems that derive dataset from other datasets

Foundation of Data Systems

Building Blocks

Databases	Store data so that they, or another application and find it later
Caches	Remember the result of an expensive operation, to speed up reads
Search Indexes	Allow users to search data by keyword or filter it in various ways
Stream Processing	Send a message to another process, handled asynchronously
Batch Processing	Periodically crunch a large amount of accumulated data

- Trends:
    - New tools for data storage and processing merged for various use cases
    - Increasingly many apps have demanding that a single tool can't handle, so that tasks break down into ones that can be performed efficiently on a single tool.

Chapter 1: Reliable, Scalable and Maintainable Data Systems

Key Factors

Reliability	The system should work **correctly** even in the face of faults and errors
Scalability	There should be reasonable ways to deal with system **growth**
Maintainability	Many different people can work on maintaining an adapting the system **productively**

Reliability

- Fault-tolerance: one component of the system deviating from its spec, unlike failures when system as a whole stops providing the required service to users.
    - Beneficial to trigger the faults deliberately, e.g. Netflix's Chaos Monkey
- Hardware faults:
    - Add redundancy to the hardware components.
    - Trend to use software fault-tolerance techniques, compared to hardware redundancy
- Software errors:
    - The assumptions software make is somehow on longer true under some circumstances.
    - Carefully think about assumptions, testing, measuring, monitoring, process isolation, etc.
- Human errors:
    - Design systems that minimize opportunities for errors
    - Decouple the places where people make the most mistakes from where they can cause failure
    - Test thoroughly at all levels, from unit tests to integration tests.
    - Minimize the impact in the case of failure
    - Set up detailed and clear monitoring such as performance metrics and error rates.
    - Good management and training.

Scalability

- Def: the ability to cope with increased load.
- Describing Load:
    - The numbers to describe load, depending on the system.
    - E.g. RPS, ration of reads to writes, etc.
    - Twitter
        - Operations: post tweet and show tweet in home timeline.
        - Approaches:
            - Posting inserts to global database, each user's timeline fetches

        - Posting inserts to global database and caches to each user's timeline's "mailbox".
    - Parameters: the distribution of followers per user.
    - Newer approach: a hybrid of the two approaches.
- Describing Performance:
    - Throughput: the number of records we can process per second, for batch processing systems like Hadoop.
    - Response time:
        - The time between sending a request and receiving a response (NOT the same as latency, which is the duration the request is waiting to be handled).
        - Distribution of response time → medians, percentiles, and more.
            - Amazon cares about the 99th percentile: customer with the most products.
        - **S**ervice **L**evel **O**bjectives: metrics set expectations for clients of the service
    - It only takes one single slow backend request to slow down the entire end-user request.
- Approaches
    - Mixture of *scaling up* and *scaling out*. Elastic and manually detect load increase for scaling.
    - The architecture is specific to applications (common and rare operations), but there exist general-purpose building blocks.

Maintainability

- Majority of software cost: fix bugs, investigate errors, adapt to new platforms and more.
- Three design principles:
    - Operability: easy for operations to keep the system running smoothly.
        - Make routines easy and more time for high level activities
        - E.g. visibility to runtime and internal behaviour, support for automation and integration, good documentation, etc.
    - Simplicity: easy for new engineers to understand the system.
        - Manage complexity
        - Accidental complexity if it's not inherent in the problem that the software solves but arises only from the implementation.
        - Approach: **abstraction**
    - Evolvability: easy for engineers to make changes to system in the future, adapting for unanticipated use cases.
        - *Agile* development for data systems

Chapter 2: Data Models and Query Language

Data Models

- Data Model:
  - Influences how the software is written and how we think about the problem.
  - Each layer hides the complexity of the layers below it by providing a clean data model.

- **Relational Mode**l:
  - Data is organized into relations (tables), where each relation is an unordered collection of tuples (rows).
  - SQL and RDBMSes turn theory into reality
  - Hide implementation detail behind a cleaner interface: business data processing and then generalized well to broadly many.

- **NoSQL**:
  - Need for greater scalability, including large datasets or high write throughput.
  - Widespread preference for free and open source software
  - Specialized query operations that are not well supported
  - Frustration with restrictiveness of relational schemas and desire for dynamic and efficient one.

- Relation vs. Object
  - Object-relational mapping is needed to bridge object-oriented programming and relational models.
  - LinkedIn Profile
    - SQL model: user, positions, education, .. in separate tables and references
    - Structured data: multi-valued data to be stored within a single row.
    - Document: JSON or XML and let app to interprets its structure and content.
  - JSON advantages
    - Better locality, without joining or multi-queries of different tables
    - Tree structure, explicit

- Many to One, Many to Many
  - Information that is meaningful to humans is stored in only one place, and everything refers to it uses an id. Information may need to change, but id doesn't.
    - E.g. "Seattle" is stored as an id that refers to it.
    - Normalization: removes the duplication, no need to copy redundant human meaningful information and then update all of them.
  - Document model has weak support for joins, while in SQL it's easy.
  - In the long run, data tends to become more interconnected.

- History
    - IBM's *Information Management System* uses hierarchical model, similar to JSON model. Two prominent solutions back then:

    - Network model (discarded):
        - Generalized the tree model, each record could have multiple parents.
        - Access path: traverse the root record along these chains of links.
        - Requires manual access path selection, and code for querying and updating very complicated and inflexible.

    - Relational model:
        - By contrast to network model, lay out all the data in the open.
        - Query optimizer automatically decides which parts of the query to execute in which order and which indexes to use.
        - Query optimizers are complicated, but once built, they benefit for all → general purpose wins in the long run.

    - Document model:
        - Storing nested records within their parent records, but unlike network, the unique identifier *document reference* is resolved at read time by using a join or follow-up queries.

- **Document** vs Relational Model
    - Which leads to simpler code:
        - Depends on application
    - Schama:
        - Document model uses ***schema-on-read****,* while SQL uses ***schema-on-write***. For document, the application assumes some kind of structure that is not enforced by database.
    - Locality:
        - Only applies if you need large parts of the documents at the same time.
        - Recommend: keep documents small and avoid writes that increase the size of document.
        - Grouping related data together for locality is not limited to document model.
            - E.g. Google's Spanner, Oracle's multi-table index cluster tables, Bigtable data mode as in Cassandra and HBase.
    - A hybrid of document and relational model is the future.
        - Handle document-like data and also perform relational queries on it.

Query Language

- Declarative query language:
    - Specify the pattern of the data you want, but not *how* to achieve the goal.
    - Hides implementation details, makes it possible for database to introduce performance improvements without requiring any changes to queries.
    - SQL is more limited in functionality and thus it gives the database much more room for automatic optimizations.
    - Parallel execution suitable for declarative language. CPUs are faster because they have multiple cores.

- Imperative language
    - Tell the computer to perform certain operations in a certain order.

- Declarative Web Query: CSS, XSL vs. Javascript DOM. The former is much better (improve performance without rewriting codes, etc)

- MapReduce: the logic of the query is expressed with code snippets, which are called repeatedly by the processing framework.
    - *Map*: called once for every document that matches query and emits a key-value pair;
    - *Reduce*: group all values with same key and called reduced once.
    - Must be pure functions, low-level for distributed execution on cluster of machine. Powerful because we can write Javascript codes.
    - MongoDB: *aggregation pipeline* reinvents some subset of SQL

Graph-Like Data Models

- Property Graph
    - Store as two relational tables, one for vertices and one for edges.
    - Any vertex can be connected to any other.
    - Given vertex, can find both incoming and outgoing edges, and thus traverse the graph.
    - Different labels for different kinds of relationships, thus store several different kinds of information in a single graph.
    - Graphs are good for evolvability

- Graph Query
    - *Cypher Query Language:* query about the graphs.
    - SQL: recursive common table expressions, for variable length queries.

- Triple Stores: (subject, predicate, object)

    - The subject is a vertex, while object is a primitive data type (then predicate : object are key : value pairs,) or another vertex (then predicate is an edge).

- Semantic Web:
    - *Resource Description Framework:* different websites to publish data in a consistent format, allowing data to be combined into a web of data.
    - *SPARQL*: query language for triple-stores using RDF data model.
    - Foundation: Datalog query language, rules can be combined and reused in different queries.

- Graph vs. Network Model

Network (CODASYL)	Graph
Schema that specified which type to be nested within which other	Any vetice can connect → flexibility
Only way to reach particular record was to traverse one of the access path	Refer to any vertex by ID, use an index to find vertices
Children of a record are an ordered set	Not ordered
All queries are imperative	Most support high-level, declarative

Summary

- SQL: hierarchical not good for many to many → relational database model invented
- NoSQL: document and graph
    - Typically don't enforce an schema
- Schema: enforced on writing or handled on read.

Chapter 3: Storage and Retrieval

Database stores some data and then gives it back upon request. As programmers we need to select the storage engine that's optimized for different use cases.

Data Structures That Power Your Database

- Log: append-only sequence of records.

Index

- Index: additional structure that is derived from the primary data.
    - Tradeoff: well-chosen indexes speed up read queries, but every index slows down writes.

- HashIndex:
    - Keep an in-memory hash map where every key is mapped to a byte offset in the data file, location where the value can be found. E.g. *Bitcask*.
    - Limited space: break logs into segments. Each segment has its own in-memory hashmap. Lookup key by order.
    - Compaction: throw away duplicate keys in the log, keeping only the most recent update. Merging and compaction of frozen segments can be done in the background.
    - Adv: value for each key is updated frequently, but not too many distinct keys.
    - Implementation: binary format, deletion records, crash recovery as in-disk snapshot of hashmap, checksum for partially written records, concurrency control with single write thread.
    - Advantages and limitations:

Advantages	Limitations
Appending and segment merging are sequential write operations that are much faster on disk and SSD.	Hash table must fit in memory. On-disk hash map doesn't perform well.
Concurrency and crash recovery are much simpler if segment files are append-only and immutable.	Range queries are not efficient.
Merging old segments avoids fragmentation.	

- **Sorted String Table (SST)**: the sequence of key-value pairs is sorted by key. Each key only appears once within each merged segment file.
    - *Used by* LevelDB, Cassandra and HBase, *Inspired by* Google's Bigtable paper.

    - Merging segments is simple and efficient: like merge sort. Keep the most recent values for the same keys in different segments.
    - No longer need to keep all key in memory: only an in-memory index to tell the offsets for some of the keys, and it can be sparse. Search key within ranges.
    - Compression: each read request scan through several records so just group them. Reduce the I/O bandwidth use.
    - ***Log-structured Merge-Tree*** Algorithm:
        - When writing, add it to in-memory AVL or Red-black tree. Called *Memtable*.
        - When memtable gets bigger than some threshold, write it out to disk as an SSTable file.
        - To serve read request, try finding the key in the memtable, then in the most recent on-disk segment, then in the next-older segment.
        - Merge and compac in the background to combine segment files.
        - Avoid losing: keep a seperate log for the in-memory memtable.
    - Implementation:
        - Bloom filter to check key doesn't exist.
        - Size-tired compilation: newer and smaller SSTables are successively merged into older and larger ones.
        - Level compaction: key range in split up into smaller tables and older data is moved into separate levels.
    - Advantage:
        - Efficient range queries;
        - Disk writes are sequential so LSM supports high write throughput.

- **B-Trees**: keep key-value pairs sorted by key; break database into fixed-sized *blocks*, called pages, can refer to each other by location or address.
    - Most widely used: remain standard index implementation in almost all relational databases.
    - Algorithm: start from the root of the B-tree to look up a key in the index. Each child is responsible for a continuous range of keys and the keys between the references indicate the boundary.
    - Branching factor:
        - Number of references to child page in one page of the B-tree.
        - Depend on the amount of space required to store the page references and the range boundaries.
    - Always Balanced: B-tree with n keys always ahs depth of O(log(n)).
        - A four-level tree of 4KB pages with a branching factor of 500 can store up to 256TB.
    - Reliability:
        - Overwrite pages on disk with new data. → dangerous, some may need different pages to be written (split page upon insertion and update their parent).

        - Write-ahead log (WAL, or redo log): append-only file to which every B-tree modification must be written before it can be applied to the pages of the tree itself.
        - Concurrency control: *latches* (lightweight locks).
    - Optimization:
        - Copy-on-write: modified page written to a different location.
        - Save space by not storing the entire key.
        - Pages can be positioned anywhere on disk. Try to layout the leaf pages in sequential order.
        - Leaf pages may have references to its sibling pages to the left and right.

- LSM-Trees vs. B-Trees
    - General: LSM-Trees faster for writes, B-Trees faster for reads (LSM checks for different data structures and stages of compaction).
    - Advantage of LSM-trees:
        - Write amplification: LSM also needs to rewrite multiple times due to repeated compaction. → SSD can only overwrite blocks a limited number of times before wearing out.
            - B-tree must write every piece of data at least twice.
            - Write-heavy applications: write amplification has direct performance cost.
        - Sustain higher write throughput: sequential write compact SST rather than overwrite several pages.
        - Compressed better for smaller files on disk. Reduced fragmentation.
        - Many SSD's firmware internally uses a log-structured algorithm.
    - Downsides:
        - Compaction process can interfere with the performance of on-going reads and writes.
        - High write throughput: disk's finite bandwidth needs to be shared between the initial write and the compaction threads. Compaction rate may not keep up with writing.
        - B-tree's key exists in exactly one place, the locks can be directly attached to the tree.

- Other Index Structures
    - Secondary index:
        - Make each value in the index a list of matching row identifiers.
        - Make each entry unique by appending a row identifier to it.
    - Storing values within index:
        - Heapfile: reference to the row stored elsewhere. Avoids duplicating data when multiple secondary indexes are present. Each index just references a location in the heap file and the actual data is kept in one place.
        - Clustered index: store the indexed row directly within an index.

        - MySQL's innoDB
        - SQL Server
    - Covering index / index with included columns: a compromise between a clustered and non-clustered one.
        - Speed up reading, but require additional storage and can add overhead on writes.
- Multi-column indexes:
    - Concatenated index: combine several fields into one key by appending one after another.
    - Multi-dimensional indexes: important for geospatial data.
        - Can translate into a single number and then use *R-tree*. (PostgreSQL's Generalized Search Tree indexing facility)
        - E.g. color, weather and much more data.
- Full-text search and fuzzy indexes:
    - *Levenshtein automation*: in-memory index similar to a trie and search for words in a certain *edit distance*.
    - Other: information retrieval through machine learning and document classification.
- In-memory database:
    - Durability: achieved through special hardware, writing a log of changes, writing periodic snapshots to disk or by replicating the in-memory state to other machines.
        - E.g. VoltDB, MemSQL, Oracle TimesTen, RAMCloud
    - Performance: improvement by avoiding encoding in-memory data structures in a form that can be written to disk; NOT by not reading from disk, most disk don't read to read from disk.
    - Data models that are difficult to implement with disk-based indexes.
        - Redis: database-like interface to various data structures such as priority queue and sets.
    - Anti-caching: evict the least recently used data from memory to disk when not enough and when it's accessed again load it back into memory.
        - Similar to that of OS, but more efficient by dealing with individual records.
- Future changes: Non-volatile techniques.