[SYSTEM-DESIGN]

Designing Data-Intensive Applications – Chapter 1: Reliable, Scalable, and Maintainable Applications

Posted by CHARLES *on* 2020-04-02

《Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems》 https://amzn.to/2WYphy6

Reliability

- Continuing to work correctly, even when things go wrong.
  - aka. *fault-tolerant or resilient.*
- **Fault vs. Failure** (fault means one component failed), design system that able tolerance the fault to prevent failure;
- **Hardware Faults**: Redundancy is the key;
- **Software Errors**: could be more troublesome;
  - Developers need to make assumptions and interactions carefully.
- **Human Errors**: Design to minimize chances of errors; decouple; Test thoroughly; Quick recover ability; monitoring(performance/error rate), aka Telemetry; Good management practice/training;

Scalability:

- A system's ability to cope with increased load.
- **Describing Load**: defined by "load parameters", it depend on architecture of your system, e.g.
  - the **requests per second** to a web server,
  - the **ratio of reads to writes** in a database,
  - the **number of simultaneously active users** in a chatroom,

- the **hit rate** on a **cache**, etc.
- Twitter hybrid approach: fanned-out vs. fetched

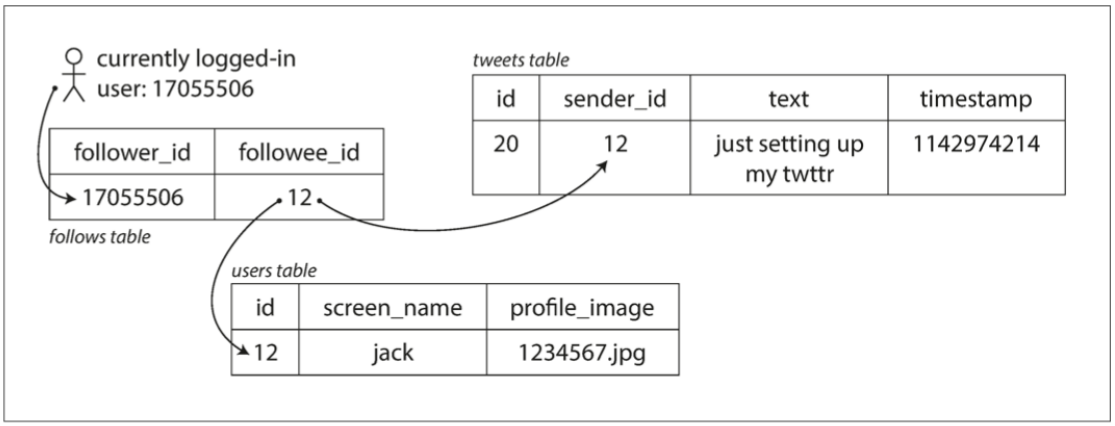


Figure 1-2. Simple relational schema for implementing a Twitter home timeline.

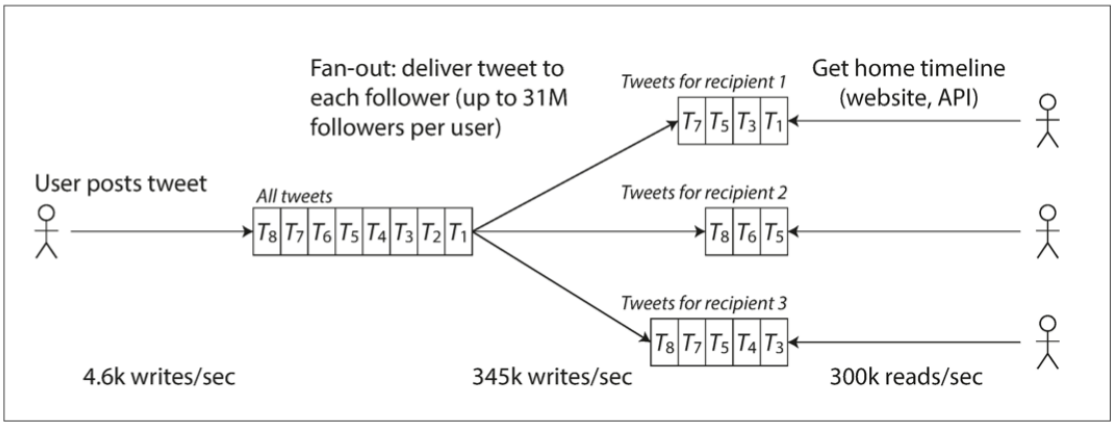


Figure 1-3. Twitter's data pipeline for delivering tweets to followers, with load parameters as of November 2012 [16].

- the average rate of published tweets is almost two orders of magnitude lower than the rate of home timeline reads → Fanned-Out approach
  - On average, a tweet is delivered to about 75 followers, so 4.6k tweets per second become 345k writes per second to the home timeline caches.
  - Moved to Hybrid approach to handle Justin-Biber effects.
- **Describing Performance**: When you increase a load parameter, keep the same system resources, what will happen ? How much do you need to increase resources to keep up the performance ?
  - E.g. "**response time**"— the time between a client sending a request and receiving a response.
  - **Latency vs. response time**: Latency is the duration that a request is waiting to be handled—during which it is latent, awaiting service.
  - Better use "**percentiles**" to measure response time, e.g. median(p50)

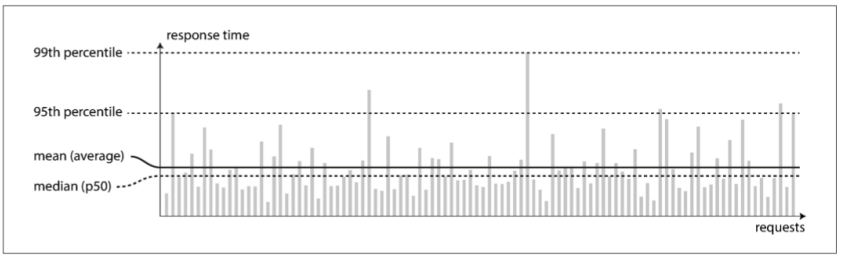


Figure 1-4. Illustrating mean and percentiles: response times for a sample of 100 requests to a service.

- High percentiles of response times, also known as **tail latencies**, are important because they directly affect users' experience of the service.
- Amazon describes response time requirements for internal services in terms of the 99.9th percentile (affects 1 in 1,000 requests).
  - On the other hand, optimizing the 99.99th percentile (the slowest 1 in 10,000 requests) was deemed too expensive and to not yield enough benefit for Amazon's purposes.
- Amazon has also observed that a 100 ms increase in response time reduces sales by 1% [20], and others report that a 1-second slowdown reduces a customer satisfaction metric by 16%
- **It is important to measure response times on the client side**.

- **Approaches for Coping with Load**:
  - Good architectures usually involve a pragmatic mixture of approaches:**Scaling up**(Vertical) & **Scaling out**(horizontal)
    - Scaling out:**Stateless** service is fairly straightforward compared to **Stateful** service.
  - It is conceivable that distributed data systems will become the default in the future.
  - The architecture of systems that operate at large scale is usually highly specific to the application.
  - An scalable architecture usually built from general-purpose building blocks arranged in familiar patterns.

Maintainability:

- **Operability**: Make it easy for operations teams to keep the system running smoothly.
- **Simplicity**: Make it easy for new engineers to understand the system,
  - By removing as much complexity as possible from the system. (Note: this is not the same as simplicity of the user interface.)
  - Making a system simpler DOES NOT necessarily mean reducing its functionality;
  - **Complexity**: as accidental if it is not inherent in the problem that the software solves (as seen by the users) but arises only from the implementation.
    - One of the best tools we have for removing accidental complexity is**abstraction**. (However,finding good abstractions is very hard.)
- **Evolvability**:Make it easy for engineers to make changes to the system in the future, adapting it for unanticipated use cases as requirements change. Also known as **extensibility**, **modifiability**, or **plasticity**.
  - **Agile** working patterns provide a framework for adapting to change
  - simple and easy-to-understand systems are usually easier to modify than complex

ones (Linked to the previous point – **Maintainability**)

Summary

- **functional requirements**: (what it should do, such as allowing data to be stored, retrieved, searched, and processed in various ways),
- **nonfunctional requirements**: (general properties like security, reliability, compliance, scalability, compatibility, and maintainability)
- **Reliability** means making systems work correctly, even when faults occur.
- **Scalability** means having strategies for keeping performance good, even when load increases.
- **Maintainability** has many facets, but in essence it's about making life better for the engineering and operations teams who need to work with the system.

《Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems》https://amzn.to/2WYphy6

PREVIOUS POST

Designing Data Intensive Applications – Chapter 3: Storage and Retrieval

NEXT POST

Designing Data-Intensive Applications – Chapter 2: Data Models and Query Languages

One Comment

ADD YOURS

Pingback: Design Resources – GEEKSTER

[SYSTEM-DESIGN]

Designing Data-Intensive Applications – Chapter 2: Data Models and Query Languages

Posted by CHARLES *on* 2020-04-02

《Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems》 https://amzn.to/2WYphy6

- **Relational DB:** MS SQL, MySQL, IBM DB2, PostgreSQL, SQLite etc.
- **Document DB:** Cassandra, HBase, Google Spanner, RethinkDB, MongoDB etc.
- **Graph DB**: Neo4j,Titan,InfiniteGraph,AllegroGraph,Cypher,SPARQL,Gremlin,Pregel
- **Data Models**:
  - Not only on how the software is written, but also on how we think about the problem that we are solving.
  - Each layer hides the complexity of the layers below it by providing a clean data model.

Relational Model vs. Document Model:

- **Relational Model**: data is organized into relations (called tables in SQL), where each relation is an unordered collection of tuples (rows in SQL)
- **Birth of NoSQL**: (high write)
  - A need for **greater scalability** than relational databases can easily achieve, including very large datasets or very **high write throughput**.
  - **Polyglot persistence**: Hybrid → SQL + NoSQL
- **The Object-Relational Mismatch**: The disconnect between the models is sometimes called an impedance mismatch.
- **Many-to-One and Many-to-Many Relationships**:
  - When you store the text directly, you are duplicating the human-meaningful information in every record that uses it.
  - Removing such duplication is the key idea behind**normalization** in databases.

- **The relational model**
  - The relational model thus made it much easier to add new features to applications.
  - You only need to build a query optimizer once, and then all applications that use the database can benefit from it.
  - Original **Document DB** is Good for One-To-Many, but Not good for Many-to-One;
- **Relational vs. Document Databases Today(Data Model)**:
  - **Document DB vs. Relational DB**
    - **Document DB**: The main arguments in favor of the document data model are **schema flexibility**, **better performance due to locality**, and that for some applications it is closer to the data structures used by the application.
    - **Relational DB**: The relational model counters by providing b**etter support for joins**, and many-to-one and many-to-many relationships.
  - **Which data model leads to simpler application code**? it depends on the kinds of relationships that exist between data items.
    - If the data in your application has a document-like structure (i.e., a tree of one-to-many relationships, where typically the entire tree is loaded at once), then it's probably a good idea to use a document model.
      - **Limitation**: deeply nested; joins; many-to-many
  - **Schema flexibility in the document model**:
    - **Schema-on-read** (DocumentDB, e.g. dynamic runtime type checking) vs.**Schema-on-write** (Relational DB, e.g. static compile time type checking)
      - E.g. default of NULL and fill it in at read time, like it would with a document database.
  - **Data locality for queries**:
    - If your application often needs to access the entire document (for example, to render it on a web page), there is a performance advantage to this *storage locality*.
    - The idea of grouping related data together for locality is not limited to the document model. (e.g. Spanner DB, Oracle, Bigtable)
  - **Convergence of document and relational databases**:
    - It seems that relational and document databases are becoming more similar over time, and that is a good thing: the data models complement each other.

Query Languages for Data:

- SQL is a *declarative* query language, whereas IMS and CODASYL query the database using *imperative* code.
  - **Declarative** query language: hides implementation details of the database engine; often lend themselves to parallel execution.(which is important)
  - **Declarative Queries on the Web**:
    - E.g. CSS/XSL vs. DOM API
  - **MapReduce Querying**: MapReduce is a programming model for processing large amounts of data in bulk across many machines, popularized by Google. (e.g. MongoDB)
    - MapReduce is neither a declarative query language nor a fully imperative query API, but somewhere in between.

Graph-Like Data Models:

- A graph consists of two kinds of objects: vertices (also known as nodes or entities) and edges (also known as relationships or arcs).

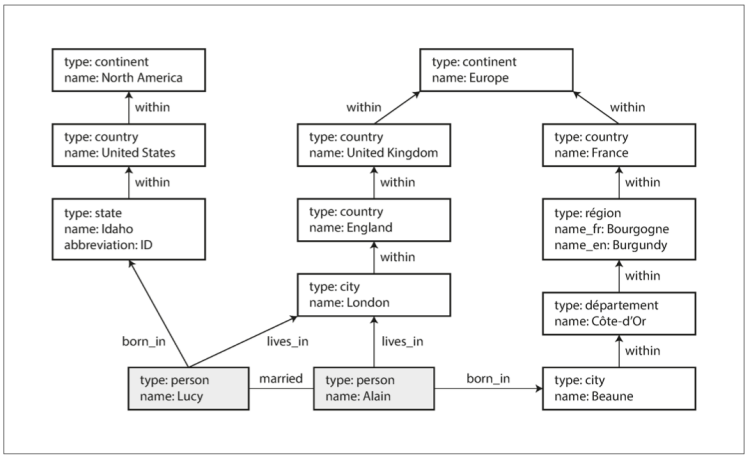


Figure 2-5. Example of graph-structured data (boxes represent vertices, arrows represent edges).

- **Good for**: If your application has mostly one-to-many relationships (tree-structured data) or no relationships between records.
- **Property graph** model (implemented by Neo4j, Titan, and InfiniteGraph): No schema restricts; travers; maintaining a clean model;
  - **Cypher Query Language(**Neo4j**)**: declarative query language for property graphs
  - **Graph Queries in SQL**: possible but difficult;
- **Triple-store** model (implemented by Datomic, AllegroGraph, and others): mostly equivalent to the property graph model.
  - In a triple-store, all information is stored in the form of very simple three-part statements: (subject, predicate, object). The subject of a triple is equivalent to a vertex in a graph.
  - **The semantic web**: The triple-store data model is completely independent of the semantic web(e.g. Datomic)
  - **RDF(Resource Description Framework) data model**:
    - Tool – Apache Jena
- **Declarative query languages** for graphs: Cypher, SPARQL, and Datalog.
  - **The SPARQL("sparkle") query language**: SPARQL is a query language for triple-stores using the RDF data model
  - **Datalog**: much older language than SPARQL or Cypher, a foundation of later query language (e.g. Datomic, Cascalog),
    - Subset of Prolog.
    - Similar to the triple-store model, generalized a bit. Instead of writing a triple as (subject, predicate, object), we write it as predicate(subject, object).
- **Imperative graph query languages** such as Gremlin and graph processing frameworks like Pregel

Summary

- Historically, data started out being represented as one big tree (the hierarchical model), but that wasn't good for representing many-to-many relationships, so the relational

model was invented to solve that problem.

- New non-relational "NoSQL" datastores have **diverged** in **two main directions**:
  - **Document databases** target use cases:
    - where data comes in self-contained documents;
    - relationships between one document and another are rare.
  - **Graph databases** go in the opposite direction, targeting use cases:
    - where anything is potentially related to everything.
- **Document** and **Graph databases** typically **don't enforce a schema** for the data they store, which can make it easier to adapt applications to changing requirements
- schema is **explicit** (enforced on write) or **implicit** (handled on read).

《Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems》 https://amzn.to/2WYphy6

PREVIOUS POST

Designing Data-Intensive Applications – Chapter 1: Reliable, Scalable, and Maintainable Applications

NEXT POST

Designing Data-Intensive Applications – Chapter 4: Encoding and Evolution

Leave a Reply

[SYSTEM-DESIGN]

Designing Data Intensive Applications – Chapter 3: Storage and Retrieval

Posted by CHARLES *on* 2020-04-02

<Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems>

(If you keep things tidily ordered, you're just too lazy to go searching.) —German proverb

- DB needs to do two things:
  - when you give it some data, it should store the data;
  - when you ask it again later, it should give the data back to you.
- How we can store the data that we're given, and how we can find it again when we're asked for it.
  - There is a big difference between storage engines that are optimized for **transactional** workloads and those that are optimized for **analytics**.
- **Two Type of DB Engine**
  - **log-structured** storage engines: (Log: an append-only sequence of records.)
    - many databases internally use a log, which is an append-only data file;
  - **page-oriented** storage engines such as B-trees(more popular)**:**

Data Structures That Power Your Database:

- **Log-Structured** need an Index to speed up look up time: An index is an additional structure that is derived from the primary data;
  - Maintaining additional structures incurs overhead, especially on writes, because it need updated every time on write;
  - Application developer or database administrator to choose indexes manually.
- **Segment Files:** break the log-file into segment
  - **Merging/Compaction:** means throwing away duplicate keys in the log, and keeping

only the most recent update for each key.

- **Hash Indexes**: basically use a HashMap(HashTable) to store the index in memory for speed up the lookup time:
  - Limitation: The hash table must fit in memory; Range queries are not efficient.
- **SSTables and LSM-Trees**:
  - **SSTable**: (We call this format **Sorted String Table**, or **SSTable** for short)
    - We require that the sequence of key-value pairs is sorted by key;
    - We also require that each key only appears once within each merged segment file;
    - **Advantages**:
      - Merging segments is simple and efficient, even if the files are bigger than the available memory(kind like mergesort);
      - no longer need to keep an index of all the keys in memory;
      - Since read requests need to scan over several key-value pairs in the requested range anyway, it is possible to group those records into a block and compress it before writing it to disk.
    - **Constructing and maintaining SSTables**: e.g. LevelDB, RocksDB,Cassandra and HBase
      - How to keep it sorted: in-memory tree is sometimes called a **memtable**, e.g. red-black tree or AVL tree;
      - When memtable gets bigger than some threshold, write it to disk;
      - serve a read request: start from memtable → most recent segment → other segments;
  - **Making an LSM-tree out of SSTables**:
    - LSM-tree: **Log-Structured Merge-Tree** (or LSM-Tree)
    - Storage engines that are based on this principle of **merging** and **compacting** sorted files are often called LSM storage engines;
    - **Lucene**, an indexing engine for full-text search used by **Elasticsearch** and **Solr**.
  - **Performance optimizations**:
    - **Bloom filters**: a memory-efficient data structure for approximating the contents of a set. It can tell you if a key does/may not appear in the database, and thus saves many unnecessary disk reads for nonexistent keys.
    - **Size-tiered**: newer and smaller SSTables are successively merged into older and larger SSTables. (e.g.HBase, Cassandra)
    - **leveled compaction**: the key range is split up into smaller SSTables and older data is moved into separate “levels,” which allows the compaction to proceed more incrementally and use less disk space. (e.g. LevelDB and RocksDB, Cassandra)
- **B-Trees(most popular)**:
  - They remain the standard index implementation in almost all relational databases, and many non-relational databases use them too;
  - Like SSTables, B-trees keep key-value pairs sorted by key, but B-trees have a very different design philosophy.
  - 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. (Corresponding to Hardware structure, HDD)

    - The number of references to child pages in one page of the B-tree is called the **branching factor**. Growing a B-tree by splitting a page
    - The tree remains balanced: a B-tree with n keys always has a depth of O(log n).
        - Fun fact: A four-level tree of 4 KB pages with a branching factor of 500 can store up to 256 TB.
    - **Making B-trees reliable**:
        - Common for B-tree implementations to include an additional data structure on disk: a write-ahead log (**WAL**, also known as a redo log).
            - This 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.
            - protecting the tree's data structures with latches (lightweight locks)
    - **B-tree optimizations**:
        - Instead of overwriting pages and maintaining a WAL for crash recovery, some databases (like LMDB) use a copy-on-write scheme;
        - We can save space in pages by not storing the entire key, but abbreviating it.
- **Comparing B-Trees and LSM-Trees**:
    - **LSM-trees** are typically faster for writes;
        - **Pros** of LSM-trees:
            - sometimes have **lower write amplification**
            - LSM-trees can be **compressed** better
            - they have **lower storage overheads**, especially when using leveled compaction.
        - **Write Amplification**: Rewrite data multiple times due to repeated compaction and merging of SSTables.
        - **Cons** of LSM-trees:
            - The compaction process can sometimes interfere with the performance of ongoing reads and writes.
            - the bigger the database gets, the more disk bandwidth is required for compaction.
            - If write throughput is high and compaction is not configured carefully, it can happen that compaction cannot keep up with the rate of incoming writes.
    - **B-trees** are thought to be faster for reads;
        - This aspect makes B-trees attractive in databases that want to offer strong transactional semantics: in many relational databases, transaction isolation is implemented using locks on ranges of keys, and in a B-tree index, those locks can be directly attached to the tree.
- **Other Indexing Structures**:
    - **key-value indexes**, which are like a primary key index in the relational model.
    - **Secondary indexes**, which are often crucial for performing joins efficiently.
        - Either way, both B-trees and log-structured indexes can be used as secondary indexes.
    - **Storing values within the index**:
        - The Value: it could be the actual row (document, vertex) in question, or it could be a reference to the row stored elsewhere(Heap file).
            - Each index just references a location in the heap file, and the actual data is

kept in one place.(avoid duplication when multiple secondary indexes)

- **Cluster Index**: it can be desirable to store the indexed row directly within an index.
- **Covering index** or index with included columns: A compromise between
  - a clustered index (storing all row data within the index)
  - a nonclustered index (storing only references to the data within the index), which stores some of a table's columns within the index.
- **Multi-column indexes**:
  - **concatenated index**: which simply combines several fields into one key by appending one column to another.
  - **Multi-dimensional indexes** are a more general way of querying several columns at once, which is particularly important for geospatial data
- **Full-text search and fuzzy indexes**:
  - In Lucene, the in-memory index is a finite state automaton over the characters in the keys, similar to a trie; (C: aka. reverse index)
    - Levenshtein automaton, which supports efficient search for words within a given edit distance(add/remove/update)

- **Keeping everything in memory**: (e.g. Memcached, VoltDB, MemSQL, and Oracle TimesTen, RAMCloud,Redis and Couchbase)
  - Many datasets are simply not that big, so it's quite feasible to keep them entirely in memory,potentially distributed across several machines. This has led to the development of **in-memory databases**.
  - **Persistence**: special hardware (such as battery-powered RAM), by writing a log of changes to disk, by writing periodic snapshots to disk, or by replicating the in-memory state to other machines.
  - 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.
  - Providing data models that are difficult to implement with disk-based indexes. (e.g. Redis with Priority-Queue and Sets)

Transaction Processing or Analytics

- **ACID (Atomicity, Consistency, Isolation, and Durability)** properties.
- online transaction processing (**OLTP**) vs. online analytical processing (**OLAP**)

Table 3-1. Comparing characteristics of transaction processing versus analytic systems

Property	Transaction processing systems (OLTP)	Analytic systems (OLAP)
Main read pattern	Small number of records per query, fetched by key	Aggregate over large number of records
Main write pattern	Random-access, low-latency writes from user input	Bulk import (ETL) or event stream
Primarily used by	End user/customer, via web application	Internal analyst, for decision support
What data represents	Latest state of data (current point in time)	History of events that happened over time
Dataset size	Gigabytes to terabytes	Terabytes to petabytes

- Databases started being increasingly used for data analytics, which has very different access patterns.
  - These queries are often written by business analysts, and feed into reports that help the management of a company make better decisions (business intelligence **BI**)
- **Data Warehousing**: is a separate database that analysts can query to their hearts' content, without affecting OLTP operations. contains a **read-only copy** of the data in all

the various OLTP systems in the company.
    - Data is extracted from OLTP databases, This process is known as Extract–Transform–Load (ETL).
    - Data warehouse can be optimized for analytic access patterns.
- **The divergence between OLTP databases and data warehouses**:
    - Many database vendors now focus on supporting either transaction processing or analytics workloads, but not both. (e.g. Teradata, Vertica, SAP HANA, and ParAccel(RedShift from AWS), SQL-on-Hadoop, Apache Hive, Spark SQL, Cloudera Impala, Facebook Presto, Apache Tajo, and Apache Drill, Google Dremel)

Stars and Snowflakes: Schemas for Analytics

- **Star schema** (also known as dimensional modeling): The name "star schema" comes from the fact that when the table relationships are visualized, the fact table is in the middle, surrounded by its dimension tables; the connections to these tables are like the rays of a star.
    - At the center of the schema is a so-called **fact table**;
        - fact tables often have over 100 columns, sometimes several hundred;
    - Some of the columns in the fact table are attributes, e.g. price at which the product was sold and the cost of buying it from the supplier;
    - Other columns in the fact table are foreign key references to other tables, called **dimension tables**.
        - The dimensions represent the **who, what, where, when, how, and why** of the event.
- **Snowflake Schema**: where dimensions are further broken down into subdimensions.
- **Column-Oriented Storage**:
    - **Row-Oriented fashion**: all the values from one row of a table are stored next to each other.
    - The idea behind **Column-Oriented storage**: don't store all the values from one row together, but store all the values from each column together instead.
        - If each column is stored in a separate file, a query only needs to read and parse those columns that are used in that query, which can save a lot of work.
    - The column-oriented storage layout relies on each column file containing the rows in the **same order**.
    - **Column Compression**: we can further reduce the demands on disk throughput by compressing data(e.g. bitmap encoding)
        - We can now take a column with n distinct values and turn it into n separate bitmaps: one bitmap for each distinct value, with one bit for each row. The bit is 1 if the row has that value, and 0 if not.
        - If n is very small (for example, a country column may have approximately 200 distinct values), those bitmaps can be stored with one bit per row.
        - if n is bigger, there will be a lot of zeros in most of the bitmaps (we say that they are sparse), the bitmaps can additionally be run-length encoded.
    - **Memory bandwidth and vectorized processing**:
        - Operators, such as the bitwise AND and OR described previously, can be

designed to operate on such chunks of compressed column data directly.

- **Sort Order in Column Storage**:
  - The data needs to be sorted an entire row at a time, even though it is stored by column.
  - A second column can determine the sort order of any rows that have the same value in the first column.
  - It can help with compression of columns.
  - **Several different sort orders**:
    - Different queries benefit from different sort orders, so why not store the same data sorted in several different ways.
    - Having multiple sort orders in a column-oriented store is a bit similar to having multiple secondary indexes in a row-oriented store.
- **Writing to Column-Oriented Storage**:
  - Column-oriented storage, compression, and sorting all help to make those read queries faster.
  - However, they have the downside of making writing more difficult.
    - One solution: LSM-trees. All writes first go to an in-memory store, where they are added to a sorted structure and prepared for writing to disk. (e.g. Vertica)

- **Aggregation**: **Data Cubes** and **Materialized Views**;
  - **Aggregate functions**: such as COUNT, SUM, AVG, MIN, or MAX in SQL.
  - **materialized view**: In a relational data model, it is often defined like a standard (virtual) view: a table-like object whose contents are the results of some query. (Not often in OLTP)
  - Special case of Materialized View: data cube or OLAP cube. It is a grid of aggregates grouped by different dimensions.
    - **The advantage of a materialized data cube** is that certain queries become very fast because they have effectively been precomputed.
    - **The disadvantage is that a data cube** doesn't have the same flexibility as querying the raw data.
    - use aggregates such as data cubes only as a performance boost for certain queries.

Summary

- Storage engine two broad categories:
  - Optimized for **transaction** processing (OLTP)
    - **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.
  - Optimized for **analytics** (OLAP).
    - 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.
    - **Indexes are much less relevant**. Instead it becomes important to encode data very compactly, to minimize the amount of data that the query needs to read from disk. (column-oriented storage helps achieve this goal)
- On the OLTP side, we saw storage engines from two main schools of thought:
    - The **log-structured** school, which only permits appending to files and deleting obsolete files, but never updates a file that has been written.
        - Bitcask, SSTables, LSM-trees, LevelDB, Cassandra, HBase, Lucene, and others belong to this group.
    - The **update-in-place** school, which treats the disk as a set of fixed-size pages that can be overwritten.
        - B-trees are the biggest example of this philosophy, being used in all major relational databases and also many non-relational ones.
    - Their **key idea** is that they systematically turn random-access writes into sequential writes on disk, which enables higher write throughput due to the performance characteristics of hard drives and SSDs.

<Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems>

PREVIOUS POST

Happy #chinesenewyear #yearofthepig #piggy

NEXT POST

Designing Data-Intensive Applications – Chapter 1: Reliable, Scalable, and Maintainable Applications

[SYSTEM-DESIGN]

Designing Data-Intensive Applications – Chapter 4: Encoding and Evolution

Posted by CHARLES *on* 2020-04-06

<Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems>

Everything changes and nothing stands still. —Heraclitus of Ephesus, as quoted by Plato in Cratylus (360 BCE)

- Applications inevitably change over time..
- schema-on-read ("**schemaless**") databases don't enforce a schema, so the database can contain a mixture of older and newer data formats written at different times
- When data changes usually you will need code change, but in a large application, code changes often cannot happen instantaneously.
  - With server-side applications you may want to perform a**rolling upgrade** (also known as a **staged rollout**)
  - With client-side applications you're at the mercy of the user, who may not install the update for some time.
- We need to maintain compatibility in both directions
  - **Backward compatibility**: Newer code can read data that was written by older code. (relative easy, since you already knew the existing old data schema)
  - **Forward compatibility**: Older code can read data that was written by newer code. (relative harder, it requires older code to ignore additions made by a newer version of the code)

Formats for Encoding Data:

- The translation from the in-memory representation to a byte sequence is called **encoding** (also known as **serialization** or **marshalling**), and the reverse is called **decoding**

(**parsing**, **deserialization**, **unmarshalling**).

- **Language-Specific Formats**: (bad idea, usually don't recommended)
  - Java has **java.io.Serializable**, Ruby has **Marshal**, Python has **pickle**, and so on. Many third-party libraries also exist, such as **Kryo** for Java.
  - Problems:
    - The encoding is often tied to a particular programming language, and reading the data in another language is very difficult.
    - In order to restore data in the same object types, the decoding process needs to be able to instantiate arbitrary classes.(Potential security risk)
    - **Versioning** data is often an afterthought in these libraries
    - **Efficiency** (CPU time taken to encode or decode, and the size of the encoded structure) is also often an afterthought
- **JSON, XML, and Binary Variants**:
  - Problems:
    - There is a lot of ambiguity around the encoding of numbers. This is a problem when dealing with large numbers(e.g. 2^53);
    - JSON and XML have good support for Unicode character strings (i.e., human-readable text), but they don't support binary strings
    - There is optional schema support for both XML and JSON (but complicated to learn and use)
    - CSV does not have any schema, so it is up to the application to define the meaning of each row and column.
  - Despite these flaws, JSON, XML, and CSV are good enough for many purposes.
  - **The difficulty of getting different organizations to agree on anything outweighs most other concerns**;
- **Binary encoding**: For data that is used only internally within your organization, there is less pressure to use a lowest-common-denominator encoding format
  - JSON is verbose but it use lots of space, hence it led to the development of a profusion of binary encodings for JSON (MessagePack, BSON, BJSON, UBJSON, BISON, and Smile, to name a few) and for XML (WBXML and Fast Infoset, for example) [but they wasn't save too much space neither]
- **Thrift and Protocol Buffers**: Apache Thrift [by Facebook] and Protocol Buffers (protobuf) [by Google] are binary encoding libraries that are based on the same principle.
  - Both Thrift and Protocol Buffers
    - require a schema for any data that is encoded.
    - each come with a code generation tool
  - **Thrift** has two different binary encoding formats,
    - BinaryProtocol
    - CompactProtocol (and DenseProtocol only support C++)
  - **Protocol** only have Binary encoding;
    - does not have a list or array data type, but instead has a repeated marker for fields (which is a third option alongside required and optional).
  - **Field tags and schema evolution**:
    - to maintain backward compatibility, every field you add after the initial

deployment of the schema must be optional or have a default value.
        - you can only remove a field that is optional (a required field can never be removed), and you can never use the same tag number again
    - **Data Types and schema evolution**;
- **Apache Avro**: (subproject of Hadoop) is another binary encoding format that is interestingly different from Protocol Buffers and Thrift.
    - **The writer's schema and the reader's schema**:
        - The key idea with Avro is that the writer's schema and the reader's schema don't have to be the same—they only need to be compatible.
    - **Schema evolution rules**:
        - To maintain compatibility, you may only add or remove a field that has a default value.
    - In Avro:
        - if you want to allow a field to be null, you have to use a union type. For example, union { null, long, string } field
        - Doesn't have optional and required markers.
        - Change field name: the reader's schema can contain aliases for field names, so it can match an old writer's schema field names against the aliases.
    - A database of schema versions is a useful thing to have in any case, since it acts as documentation and gives you a chance to check schema compatibility.
    - **Dynamically generated schemas**
    - Code generation and dynamically typed languages:
        - After a schema has been defined, you can generate code that implements this schema in a programming language of your choice.
- **The Merits of Schemas:**
    - schema evolution allows the same kind of flexibility as schemaless/ schema-on-read JSON databases provide, while also providing better guarantees about your data and better tooling.

Modes of Dataflow

- Forward and backward compatibility, which are important for **evolvability** (upgrade system independently, and not having to change everything at once)
- **Via Databases**:
    - **Different values written at different times**: A database generally allows any value to be updated at any time. (aka, **data outlives code**) using Schema evolution rules.
    - **Archival storage**: use latest schema
        - encode the data in an analytics-friendly column-oriented format such as Parquet
- **Via Synchronized Service call**: **REST** or **RPC**
    - **REST** seems to be the predominant style for **public** APIs.
    - **RPC** frameworks are on requests between services owned by the same organization, typically within the same datacenter.
    - The most common arrangement is to have two roles: **clients** and **servers**.
        - The API exposed by the server is known as a **service**.
    - Moreover, a server can itself be a client to another service (e.g. a web app server acts

as client to a database)
    - **service-oriented architecture (SOA)**, more recently refined and rebranded as **microservices architecture**;
    - A key design goal of a **service-oriented/microservices** architecture is to **make the application easier to change and maintain by making services independently deployable and evolvable**.
- **Web services**: **REST** and **SOAP**
    - **REST** is not a protocol, but rather a **design philosophy** that builds upon the principles of HTTP. An API designed according to the principles of REST is called RESTful.
        - A definition format such as **OpenAPI**, also known as **Swagger**, can be used to describe RESTful APIs and produce documentation.
    - **SOAP** is an XML-based protocol for making network API requests.
        - The API of a SOAP web service is described using an XML-based language called the **Web Services Description Language, or WSDL**, WSDL enables code generation so that a client can access a remote service using local classes and method calls; (good for static typed language like Java)
- **RPC**:
    - The RPC model tries to make a request to a remote network service look the same as calling a function or method in your programming language, within the same process (this abstraction is called **location transparency**).
    - Although RPC seems convenient at first, the approach is **fundamentally flawed**. A network request is very different from a local function call.
        - A network request is unpredictable;
        - A network response is unpredictable;
        - Idempotence is required for re-try;
        - A network request is much slower than a function call, and its latency is also wildly variable;
    - **Current directions for RPC**: Thrift and Avro come with RPC support included, gRPC is an RPC implementation using Protocol Buffers, Finagle also uses Thrift, and Rest.li uses JSON over HTTP.
        - Custom RPC protocols with a binary encoding format can achieve better performance than something generic like JSON over REST;
        - But REST it is good for experimentation and debugging;
    - Data encoding and evolution for RPC
- **Via Asynchronous message passing**: asynchronous message-passing systems, which are somewhere between RPC and databases.
    - **Advantages**:
        - Act as buffer;
        - Auto-redeliver message;
        - Allow message fan-out;
        - Decouple the sender and recipient;
    - **Disadvantages**:
        - message-passing communication is usually one-way: a sender normally doesn't expect to receive a reply to its messages

        - communication pattern is asynchronous: the sender doesn't wait for the message to be delivered, but simply sends it and then forgets about it
    - **Message brokers**: open source implementations such as **RabbitMQ**, ActiveMQ, HornetQ, NATS, and Apache **Kafka** have become popular
        - one process sends a message to a named **queue** or **topic**, and the **broker** ensures that the message is delivered to one or more **consumers** of or **subscribers** to that **queue** or **topic**.
        - There can be many **producers** and many **consumers** on the same topic.
        - A topic provides only one-way dataflow. However, a consumer may itself publish messages to another topic (so you can chain them together), or to a reply queue that is consumed by the sender of the original message (allowing a request/response dataflow, similar to RPC)
        - **Distributed actor frameworks**: The actor model is a programming model for concurrency in a single process. (e.g. Akka, Orleans, Erlang OTP)
            - A distributed actor framework essentially integrates a message broker and the actor programming model into a single framework.

Summary:

- many services need to support rolling upgrades → evolvability
    - released without downtime
    - make deployments less risky
- all data flowing around the system is encoded in a way that provides backward compatibility (new code can read old data) and forward compatibility (old code can read new data).
    - data encoding formats and it pros and cons;
- several modes of dataflow, illustrating different scenarios in which data encodings are important.
    - **Databases**, where the process writing to the database encodes the data and the process reading from the database decodes it.
    - **RPC and REST APIs**, where the client encodes a request, the server decodes the request and encodes a response, and the client finally decodes the response
    - **Asynchronous message passing** (using message brokers or actors), where nodes communicate by sending each other messages that are encoded by the sender and decoded by the recipient

<Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems>

[SYSTEM-DESIGN]

Designing Data-Intensive Applications – Chapter 5: Replication

Posted by CHARLES *on* 2020-04-08

<Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems>

The major difference between a thing that might go wrong and a thing that cannot possibly go wrong is that when a thing that cannot possibly go wrong goes wrong it usually turns out to be impossible to get at or repair. —Douglas Adams, Mostly Harmless (1992)

- **Replication** means keeping a copy of the same data on multiple machines that are connected via a network.
  - To keep data geographically close to your users (**reduce latency**)
  - To allow the system to continue working even if some of its parts have failed (**increase availability**)
  - To scale out the number of machines that can serve read queries (**increase read throughput**)
- **All of the difficulty in replication lies in handling changes to replicated data**;
- three popular algorithms for replicating changes between nodes:**single-leader, multi-leader, and leaderless** replication.

Leaders and Followers:

- Each node that stores a copy of the database is called a**replica**;
- How do we ensure that all the data ends up on all the replicas?
- The most common solution for this is called leader-based replication (also known as active/passive or master/slave replication)
  - One of the replicas is designated the **leader** (also known as **master** or **primary**).
  - The other replicas are known as**followers** (read replicas, **slaves**, secondaries, or hot

standbys).
    - When a client wants to read from the database, it can query either the leader or any of the followers.
    - E.g.
        - **Relational DB**: PostgreSQL (since version 9.0), MySQL, Oracle Data Guard [2], and SQL Server's AlwaysOn Availability Groups,
        - **Non-Relational DB**: MongoDB, RethinkDB, and Espresso
        - **Message Broker**: Kafka and RabbitMQ highly available queues
- **Synchronous vs. Asynchronous Replication**: In relational databases, this is often a configurable option; other systems are often hardcoded to be either one or the other.
    - Most database systems apply changes to followers in less than a second.
    - **Synchronous**:
        - Advantages: The follower is guaranteed to have an up-to-date copy of the data that is consistent with the leader.
        - Disadvantages: if the synchronous follower doesn't respond (because it has crashed, or there is a network fault, or for any other reason), the write cannot be processed
        - **impractical** for all followers to be synchronous: any one node outage would cause the whole system to grind to a halt.
    - **Semi-synchronous**: leader and one synchronous follower.
    - **Often**, leader-based replication is configured to be completely **Asynchronous**.
        - a fully asynchronous configuration has the advantage that the leader can continue processing writes, even if all of its followers have fallen behind.
        - especially if there are many followers or if they are geographically distributed.
- **Setting Up New Followers**:
    - How do you ensure that the new follower has an accurate copy of the leader's data?
    - Four Step process: 1, take snapshot; 2, copy snapshot; 3, carry-over new change by using leader's replication log(e.g. log sequence number, and MySQL calls it the binlog coordinates.); 4, after caught up, keep process data from leader;
        - The practical steps of setting up a follower vary significantly by database.
- **Handling Node Outages**: our goal is to keep the system as a whole running despite individual node failures, and to keep the impact of a node outage as small as possible.
    - How do you achieve high availability with leader-based replication?
    - **Follower failure: Catch-up recovery**:
        - from its log, it knows the last transaction that was processed before the fault occurred.
    - **Leader failure: Failover**:
        - **Failover**: one of the followers needs to be promoted to be the new leader, clients need to be reconfigured to send their writes to the new leader, and the other followers need to start consuming data changes from the new leader.
        - **Automatic failover** process usually consists of the following steps:
            - 1. Determining that the leader has failed. if a node doesn't respond for some period of time—say, 30 seconds—it is assumed to be dead;
            - 2. Choosing a new leader, through election process or controller node selection;

        - 3. Reconfiguring the system to use the new leader. The system needs to ensure that the old leader becomes a follower and recognizes the new leader.
    - **Failover is fraught with things that can go wrong**
        - If asynchronous replication is used, the new leader may not have received all the writes from the old leader before it failed.
        - Discarding writes is especially dangerous if other storage systems outside of the database need to be coordinated with the database contents.
        - In certain fault scenarios, it could happen that two nodes both believe that they are the leader.– aka. **split brain** which will cause conflict data.
        - What is the right timeout before the leader is declared dead?
    - There are no easy solutions to these problems. For this reason, some operations teams prefer to perform failovers manually.
    - These issues—node failures; unreliable networks; and trade-offs around replica consistency, durability, availability, and latency—are in fact fundamental problems in distributed systems.
- **Implementation of Replication Logs**:
    - How does leader-based replication work under the hood?
    - **Statement-based replication(Not Recommended)**:
      (e.g. early version of MySQL)
        - Simplest case, the leader logs every write request (statement) that it executes and sends that statement log to its followers.
        - Potential Issues with this approach:
            - Any statement that calls a nondeterministic function, such as NOW() will get different values;
            - If statements use an auto-incrementing column, or if they depend on the existing data in the database
            - Statements that have side effects (e.g., triggers, stored procedures, user-defined functions)
    - **Write-ahead log (WAL) shipping**: (e.g. PostgreSQL and Oracle)
        - the log is an append-only sequence of bytes containing all writes to the database. Besides writing the log to disk, the leader also sends it across the network to its followers.
        - The main disadvantage is:
            - The log describes the data on a very low level: a WAL contains details of which bytes were changed in which disk blocks. → **closely coupled to the storage engine**.
    - **Logical (row-based) log replication**:
        - An alternative is to use different log formats for replication and for the storage engine, which allows the replication log to be **decoupled from the storage engine internals**.
        - Since a **logical log** is decoupled from the storage engine internals, it can more easily be kept backward compatible, allowing the leader and the follower to run different versions of the database software, or even different storage engines.
        - A logical log format is also easier for external applications to parse.Useful if you want to send the contents of a database to an external system(e.g. **CDC** change

data capture)

- **Trigger-based replication**: (e.g. Goldengate for Oracle)
  - if you want to only replicate a subset of the data, or want to replicate from one kind of database to another, or if you need conflict resolution logic then you may need to move replication up to the application layer.
  - An alternative is to use features that are available in many relational databases: triggers and stored procedures.
    - A trigger lets you register custom application code that is automatically executed when a data change (write transaction) occurs in a database system.
  - Trigger-based replication typically has greater overheads than other replication methods, and is more prone to bugs and limitations than the database's built-in replication.
    - But, provide greater flexibility;

Problems with Replication lag:

- Leader-based replication requires all writes to go through a single node, but read-only queries can go to any replica.
- **For Read heavy application**: create many followers, and distribute the read requests across those followers. You can increase the capacity for serving read-only requests simply by adding more followers.
- **Eventual Consistent:** This inconsistency is just a temporary state—if you stop writing to the database and wait a while, the followers will **eventually** catch up and become consistent with the leader.
  - The term "**eventually**" is deliberately vague: in general, there is no limit to how far a replica can fall behind.
- **Reading Your Own Writes**:
  - A user makes a write, followed by a read from a stale replica. To prevent this anomaly, we need **read-after-write consistency**.
  - It makes no promises about other users: other users' updates may not be visible until some later time. However, it reassures the user that their own input has been saved correctly
  - How can we implement read-after-write consistency in a system with leader-based replication? e.g.
    - When reading something that the user may have modified, read it from the leader; otherwise, read it from a follower.
    - You could track the time of the last update and, for one minute after the last update, make all reads from the leader;
    - The client can remember the timestamp of its most recent write;
    - Any request that needs to be served by the leader must be routed to the datacenter that contains the leader.
  - **cross-device read-after-write consistency**: if the user enters some information on one device and then views it on another device, they should see the information they just entered.
    - If your approach requires reading from the leader, you may first need to route

requests from all of a user's devices to the same datacenter.

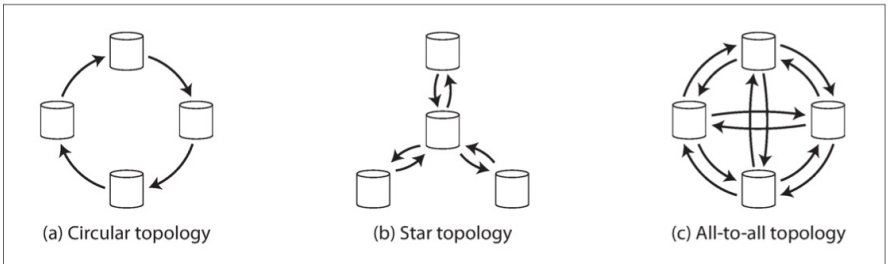
- **Monotonic Reads**:
  - when reading from asynchronous followers is that it's possible for a user to see things moving backward in time. This can happen if a user makes several reads from different replicas.
  - **Monotonic reads**: It's a lesser guarantee than strong consistency, but a stronger guarantee than eventual consistency.
    - One way of achieving monotonic reads is to make sure that each user always **makes their reads from the same replica**.
- **Consistent Prefix Reads**:
  - violation of causality. E.g. If some partitions are replicated slower than others, an observer may see the answer before they see the question.
  - **consistent prefix reads**: This guarantee says that if a sequence of writes happens in a certain order, then anyone reading those writes will see them appear in the same order.
    - This is a particular problem in partitioned (sharded) databases,
- **Solutions for Replication Lag**:
  - Stronger guarantees;
  - **Transactions**: they are a way for a database to provide stronger guarantees so that the application can be simpler.
  - **Single-node transactions** have existed for a long time, but not for distributed systems.

Multi-Leader Replication:

- Leader-based replication has one major downside: there is only one leader, and all writes must go through it. → Single point of failure;
- **Multi-leader configuration** (also known as master–master or active/active replication): A natural extension of the leader-based replication model is to allow more than one node to accept writes.
  - Replication the same: each node that processes a write must forward that data change to all the other nodes.
- **Use Cases for Multi-Leader Replication**:
  - It rarely makes sense to use a multi-leader setup within a single datacenter, because the benefits rarely outweigh the added complexity.
  - Some databases support multi-leader configurations by default, but it is also often implemented with external tools. (e.g. Tungsten Replicator for MySQL, BDR for PostgreSQL, and GoldenGate for Oracle)
  - **Multi-datacenter operation**:
    - In a multi-leader configuration, you can have **a leader in each datacenter**. Each datacenter's leader replicates its changes to the leaders in other datacenters.
  - **Advantages:** compare with Single-Leader design:
    - Performance: reduced latency;
    - Tolerance of datacenter outages:
    - Tolerance of network problems;

        - A multi-leader configuration with asynchronous replication can usually tolerate network problems better: a temporary network interruption does not prevent writes being processed.
    - **Downside**:
        - the same data may be concurrently modified in two different datacenters, and those **write conflicts** must be resolved.
        - As multi-leader replication is a somewhat retrofitted feature in many databases, there are often subtle configuration pitfalls and surprising interactions with other database features.
    - **Clients with offline operation**: E.g. calendar, mobile devices etc. (CouchDB)
        - In this case, every device has a local database that acts as a leader (it accepts write requests), and there is an asynchronous multi-leader replication process (sync) between the replicas of your calendar on all of your devices.
        - multi-leader replication is a tricky thing to get right.
    - **Collaborative editing**:
        - Real-time collaborative editing applications allow several people to edit a document simultaneously. E.g. Etherpad, Google Docs.
        - For faster collaboration, you may want to make the unit of change very small (e.g., a single keystroke) and avoid locking.
- **Handling Write Conflicts**:
    - The **biggest problem** with multi-leader replication is that write conflicts can occur, which means that conflict resolution is required.
    - **Synchronous versus asynchronous conflict detection**:
        - the conflict is only detected asynchronously at some later point in time
        - If you want synchronous conflict detection, you might as well just use single-leader replication.
    - **Conflict avoidance**:
        - The simplest strategy for dealing with conflicts is to **avoid** them: if the application can ensure that all writes for a particular record go through the same leader, then conflicts cannot occur.
    - **Converging toward a consistent state**:
        - the database must resolve the conflict in a **convergent** way, which means that all replicas must arrive at the same final value when all changes have been replicated.
        - **Ways of achieving convergent conflict resolution**:
            - Give **each write a unique ID** (e.g., a timestamp, a long random number, a UUID, or a hash of the key and value), pick the write with the highest ID as the winner, and throw away the other writes. (**LWW — Last Write Wins**)
                - This approach is popular, it is dangerously prone to data loss;
            - Give **each replica a unique ID** (also potential data loss);
            - Somehow **merge** the values together;
            - **Record** the conflict in an explicit data structure that preserves all information, and write application code that **resolves** the conflict at some later time (e.g. prompt user)
        - **Custom conflict resolution logic**:

        - Most multi-leader replication tools let you write conflict resolution logic using application code.
            - On Write: As soon as the database system detects a conflict in the log of replicated changes, it calls the conflict handler. (e.g. Bucardo)
            - On Read: When a conflict is detected, all the conflicting writes are stored. The next time the data is read, these multiple versions of the data are returned to the applica- tion. (e.g. CouchDB)
    - **Automatic Conflict Resolution**:
        - **Conflict-free replicated data** types (CRDTs, two-way merge) (e.g. Riak 2.0)
        - **Mergeable persistent data** structures (similar to git, perform three-way merge)
        - Operational transformation: It was designed particularly for concurrent editing of an ordered list of items, such as the list of characters that constitute a text document. (Etherpad, Google Docs)
- **Multi-Leader Replication Topologies**:
    - A replication topology describes the communication paths along which writes are propagated from one node to another.



    - The most general topology is all-to-all, in which every leader sends its writes to every other leader.
    - In circular and star topologies, a write may need to pass through several nodes before it reaches all replicas.
        - Problem: if just one node fails, it can interrupt the flow of replication messages between other nodes, causing them to be unable to communicate until the node is fixed.
    - Fault tolerance of a more densely connected topology (such as all-to-all) is better because it allows messages to travel along different paths, avoiding a single point of failure.
        - Problem: some network links may be faster than others (e.g., due to network congestion), with the result that some replication messages may “overtake” others.

Leaderless Replication:

- Amazon used it for its in-house **Dynamo system**.vi Riak, Cassandra, and Voldemort are open source datastores with leaderless replication models inspired by Dynamo, so this kind of database is also known as Dynamo-style. (C:DynamoDB, singler-leader?)
- In some leaderless implementations, the client directly sends its writes to several replicas, while in others, a **coordinator** node does this on behalf of the client.
- **Writing to the Database When a Node Is Down**
    - In a leaderless configuration, failover does not exist.

- A **quorum write, quorum read, and read repair** after a node outage.
- When a client reads from the database, it doesn't just send its request to one replica: **read requests are also sent to several nodes in parallel**.
- **Read repair and anti-entropy**:
  - The replication scheme should ensure that eventually all the data is copied to every replica.
  - **Read repair**: When a client makes a read from several nodes in parallel, it can detect any stale responses. (when read is heavy)
  - **Anti-entropy process**: In addition, some datastores have a background process that constantly looks for differences in the data between replicas and copies any missing data from one replica to another. (maybe delay)
- **Quorums for reading and writing**:
  - As long as **w + r > n**, we expect to get an up-to-date value when reading, because at least one of the r nodes we're reading from must be up to date.
  - **Reads and writes that obey these r and w values are called quorum reads and writes**:
    - You can think of **r** and **w** as the minimum number of votes required for the read or write to be valid.
  - In Dynamo-style databases, the parameters n, w, and r are typically configurable. A common choice is to make **n** an odd number (typically 3 or 5) and to set w = r = (n + 1) / 2 (rounded up).
  - The quorum condition, w + r > n, allows the system to tolerate unavailable nodes as follows:
    - If w < n, we can still process writes if a node is unavailable.
    - If r < n, we can still process reads if a node is unavailable.
    - With n = 3, w = 2, r = 2 we can tolerate one unavailable node.
    - With n = 5, w = 3, r = 3 we can tolerate two unavailable nodes.
    - The parameters w and r determine how many nodes we wait for
  - If fewer than the required w or r nodes are available, writes or reads return an error.
- **Limitations of Quorum Consistency**:
  - If you have **n replicas**, and you choose w and r such that w + r > n, you can generally expect every read to return the most recent value written for a key.
    - because the set of nodes to which you've written and the set of nodes from which you've read must overlap
  - Often, r and w are chosen to be a majority (more than n/2) of nodes, because that ensures w + r > n while still tolerating up to n/2 node failures.
  - With a smaller w and r:
    - more likely to read stale values
    - But allows lower latency and higher availability: if there is a network interruption and many replicas become unreachable, there's a higher chance that you can continue processing reads and writes.
  - Even with w + r > n, there are many edge cases that could cause problems
    - Thus, although quorums appear to guarantee that a read returns the latest written value, in practice it is not so simple.

- Dynamo-style databases are generally optimized for use cases that can tolerate **eventual consistency**.
    - Stronger guarantees generally require transactions or consensus.
- **Monitoring staleness**:
    - For leader-based replication, the database typically exposes metrics for the replication lag, which you can feed into a monitoring system.
    - In systems with leaderless replication, there is no fixed order in which writes are applied, which makes monitoring more difficult.
    - Eventual consistency is a deliberately vague guarantee, but for operability it's important to be able to quantify "eventual."

- **Sloppy Quorums and Hinted Handoff**:
    - Databases with appropriately configured quorums can tolerate the failure of individual nodes without the need for failover.
    - Leaderless replication appealing for use cases that require high availability and low latency, and that can tolerate occasional stale reads.
    - **Sloppy quorum**: writes and reads still require w and r successful responses, but those may include nodes that are not among the designated n "home" nodes for a value. (e.g. Riak enabled default, Cassandra, Voldemort disabled)
        - particularly useful for increasing write availability: as long as any w nodes are available, the database can accept writes.
        - isn't a quorum at all in the traditional sense. It's only an assurance of durability.
    - **Hinted handoff**: Once the network interruption is fixed, any writes that one node temporarily accepted on behalf of another node are sent to the appropriate "home" nodes.
    - **Multi-datacenter operation**:
        - **Cassandra** and Voldemort implement their multi-datacenter support within the normal leaderless model: the number of replicas n includes nodes in all datacenters, and in the configuration you can specify how many of the n replicas you want to have in each datacenter.
        - **Riak** keeps all communication between clients and database nodes local to one datacenter, so n describes the number of replicas within one datacenter.
        - Cross-datacenter replication between database clusters happens asynchronously in the background, in a style that is similar to multi-leader replication
    - **Detecting Concurrent Writes**:
        - **Dynamo-style databases** allow several clients to concurrently write to the same key, which means that conflicts will occur even if strict quorums are used.
        - In order to become eventually consistent, the replicas should converge toward the same value.
        - **Last write wins (discarding concurrent writes)**:
            - declare that each replica need only store the most "recent" value and allow "older" values to be overwritten and discarded.
            - writes are concurrent, so their order is undefined.
            - Even though the writes don't have a natural ordering, we can force an arbitrary order on them.

        - LWW achieves the goal of eventual convergence, but at the cost of durability
            - If losing data is not acceptable, LWW is a poor choice for conflict resolution.
    - **The "happens-before" relationship and concurrency**:
        - An operation A happens before another operation B if B knows about A, or depends on A, or builds upon A in some way.
        - **Concurrency, Time, and Relativity**:
            - For defining **concurrency**, exact time doesn't matter: **we simply call two operations concurrent if they are both unaware of each other, regardless of the physical time at which they occurred.**
    - **Capturing the happens-before relationship**:
        - server can determine whether two operations are concurrent by looking at the **version numbers**—it does not need to interpret the value itself (so the value could be any data structure).
        - When a write includes the version number from a prior read, that tells us which previous state the write is based on.
    - **Merging concurrently written values**:
        - if several operations happen concurrently, clients have to clean up afterward by merging the concurrently written values. (aka siblings in Riak)
            - Merging sibling values is essentially the same problem as conflict resolution in multi-leader replication
        - a reasonable approach to merging siblings is to just take the union.
    - **Version vectors**: (C: allow data to diverge and converge,like git)
        - We need to use a version number per replica as well as per key.
        - Each replica increments its own version number when processing a write, and also keeps track of the version numbers it has seen from each of the other replicas.
        - **Version vector**: The collection of version numbers from all the replicas.
            - most interesting is probably the dotted version vector, which is used in Riak 2.0
        - The version vector allows the database to distinguish between overwrites and concurrent writes.
        - The version vector structure ensures that it is safe to read from one replica and subsequently write back to another replica.
        - **Version vectors and vector clocks**:
            - A version vector is sometimes also called a vector clock, even though they are not quite the same.
            - when comparing the state of replicas, version vectors are the right data structure to use.

Summary:

- **Replication** can serve several purposes:
    - **High availability**: Keeping the system running, even when one machine (or several machines, or an entire datacenter) goes down

    - **Disconnected operation**: Allowing an application to continue working when there is a network interruption
    - **Latency**: Placing data geographically close to users, so that users can interact with it faster
    - **Scalability**: Being able to handle a higher volume of reads than a single machine could handle, by performing reads on replicas
- Despite being a simple goal, keeping a copy of the same data on several machines' replication turns out to be a remarkably tricky problem.
- Three main approaches to replication:
    - **Single-leader** replication
        - Clients send all writes to a single node (the leader), which sends a stream of data change events to the other replicas (followers). Reads can be performed on any replica, but reads from followers might be stale.
    - **Multi-leader** replication
        - Clients send each write to one of several leader nodes, any of which can accept writes. The leaders send streams of data change events to each other and to any follower nodes.
    - **Leaderless** replication
        - Clients send each write to several nodes, and read from several nodes in parallel in order to detect and correct nodes with stale data.
- Each approach has advantages and disadvantages.
    - Single-leader:
        - s popular because it is fairly easy to understand and there is no conflict resolution to worry about.
    - Multi-Leader & Leaderless:
        - Can be more robust in the presence of faulty nodes, network interruptions, and latency spikes—at the cost of being harder to reason about and providing only very weak consistency guarantees.
- Replication can be **synchronous** or **asynchronous**, which has a profound effect on the system behavior when there is a fault.
    - It's important to figure out what happens when replication lag increases and servers fail.
- Consistency models which are helpful for deciding how an application should behave under replication lag:
    - Read-after-write consistency;
    - Monotonic reads;
    - Consistent prefix reads;
- **Concurrency** issues that are inherent in multi-leader and leaderless replication approaches: because they allow multiple writes to happen concurrently, conflicts may occur.

<Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems>

[SYSTEM-DESIGN]

Designing Data Intense Application – Chapter 6: Partitioning

Posted by CHARLES *on* 2020-04-09

<Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems>

- The main reason for wanting to partition data is **scalability**.
- For very **large datasets**, or very **high throughput**, that is not sufficient: we need to break the data up into **partitions, also known as sharding**.
  - What we call a **partition** here is called a **shard** in MongoDB, Elasticsearch, and SolrCloud; it's known as a **region** in HBase, a **tablet** in Bigtable, a **vnode** in Cassandra and Riak, and a **vBucket** in Couchbase.
- Recently rediscovered by NoSQL databases and Hadoop-based data warehouses.

Partitioning and Replication:

- **Partitioning** is usually combined with **replication** so that copies of each partition are stored on multiple nodes.
- A node may store more than one partition.
  - Each partition's leader is assigned to one node, and its followers are assigned to other nodes.
  - Each node may be the leader for some partitions and a follower for other partitions.

Partitioning of Key-Value Data:

- How do you decide which records to store on which nodes?
- If the partitioning is unfair, so that some partitions have more data or queries than others, we call it **skewed**.
  - A partition with disproportionately high load is called a **hot spot**.

- Avoid Hotspot:
  - The simplest approach for avoiding hot spots would be to assign records to nodes randomly.
  - **disadvantage**: when you're trying to read a particular item, you have no way of knowing which node it is on, so you have to query all nodes in parallel.
- **Partitioning by Key Range**:
  - One way of partitioning is to assign a continuous range of keys (from some minimum to some maximum) to each partition, like the volumes of a paper encyclopedia
  - The ranges of keys are not necessarily evenly spaced, because your data may not be evenly distributed.
    - In order to distribute the data evenly, the partition boundaries need to adapt to the data.
    - Automatically choose: Bigtable, HBase, RethinkDB and MongoDB(<2.4)
  - Within each partition, we can keep keys in sorted order (see "SSTables and LSM-Trees" on page 76).
- **Partitioning by Hash of Key**: (aka. **Consistent Hashing**)
  - Because of this risk of skew and hot spots, many distributed datastores use a hash function to determine the partition for a given key.
  - A good hash function takes skewed data and makes it uniformly distributed.
    - For partitioning purposes, the hash function need NOT be cryptographically strong: e.g. **Cassandra** and **MongoDB** use **MD5**, and Voldemort uses the FowlerNoll–Vo function
  - Once you have a suitable hash function for keys, you can assign each partition a range of hashes (rather than a range of keys), and every key whose hash falls within a partition's range will be stored in that partition

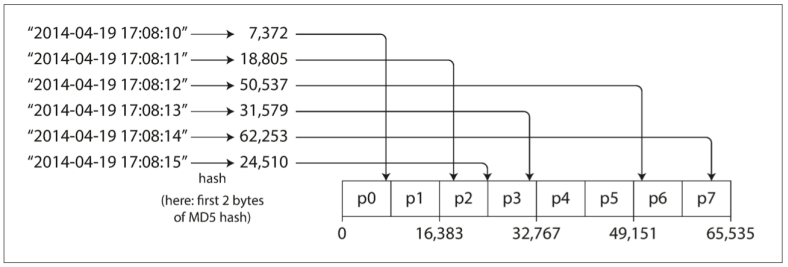


Figure 6-3. Partitioning by hash of key.

  - pseudo-randomly (in which case the technique is sometimes known as **consistent hashing**)
- **Consistent Hashing**:
  - is a way of evenly distributing load across an internet-wide system of caches such as a content delivery network (CDN).
  - this particular approach actually doesn't work very well for databases [8], so it is rarely used in practice. (C: Seems totally not aligned with what some YouTuber was saying ?)
- Downside is we lose a nice property of key-range partitioning: the ability to do efficient range queries. (C: this is very similar that when Partition a MQ, the order of the message will not be aligned)
  - In MongoDB, if you have enabled hash-based sharding mode, any range query has to be sent to all partitions.

    - Range queries on the primary key are NOT supported by Riak, Couchbase, or Voldemort.
- Cassandra achieves a compromise between the two partitioning strategies.
    - A table in Cassandra can be declared with a compound primary key consisting of several columns.
    - Only the first part of that key is hashed to determine the partition, but the other columns are used as a concatenated index for sorting the data in Cassandra's SSTables.
    - The concatenated index approach enables an elegant data model for one-to-many relationships.
- **Skewed Workloads and Relieving Hot Spots**:
    - most data systems are not able to automatically compensate for such a highly skewed workload, so it's the responsibility of the application to reduce the skew.(e.g. Celebrity posts)
        - if one key is known to be very hot, a simple technique is to**add a random number** to the beginning or end of the key.
        - However, having split the writes across different keys, any reads now have to do additional work, as they have to read the data from all 100 keys and combine it.

Partitioning and Secondary Indexes:

- The situation becomes more complicated if secondary indexes are involved.
- A secondary index usually doesn't identify a record uniquely but rather is a way of searching for occurrences of a particular value.
- Secondary indexes are the bread and butter of relational databases, and they are common in document databases too.
- Many key-value stores (such as HBase and Voldemort) have avoided secondary indexes
    - but some (such as Riak) have
- Secondary indexes are the **raison d'être** of search servers such as Solr and Elasticsearch.
- **The problem with secondary indexes is that they don't map neatly to partitions**.
- There are two main approaches to partitioning a database with secondary indexes: **document-based partitioning** and **term-based partitioning**.
- **Partitioning Secondary Indexes by Document**: (e.g. Used car selling site)
    - Widely used: **MongoDB**, Riak, **Cassandra**, Elasticsearch, SolrCloud, and VoltDB.
    - In this indexing approach, each partition is completely separate: each partition maintains its own secondary indexes, covering only the documents in that partition.

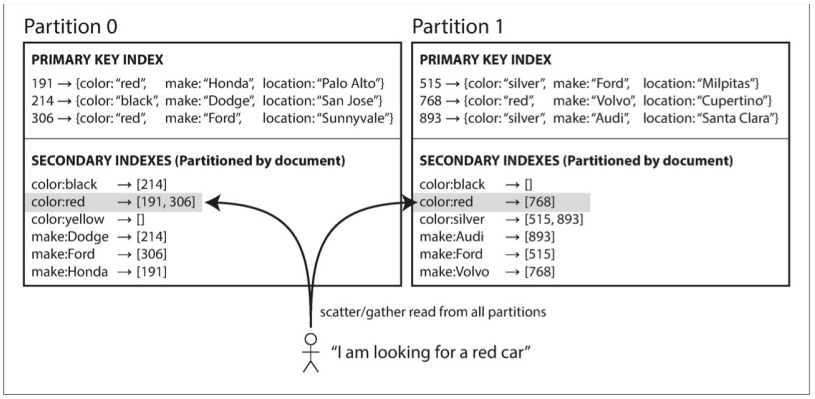


Figure 6-4. Partitioning secondary indexes by document.

- A document-partitioned index is also known as a local index (as opposed to a global index, described in the next section)
  - you need to send the query to all partitions, and combine all the results you get back.
  - This approach to querying a partitioned database is sometimes known as **scatter/gather**, and it can make read queries on secondary indexes quite expensive. (prone to tail latency amplification)
- **Partitioning Secondary Indexes by Term**:
  - We can construct a **global index** that covers data in all partitions.
  - A global index must also be partitioned, but it can be partitioned differently from the primary key index.
  - We call this kind of index term-partitioned, because the term we're looking for determines the partition of the index.
  - **Pros** it can make reads more efficient:
    - rather than doing scatter/gather over all partitions, a client only needs to make a request to the partition containing the term that it wants.
  - **Cons** it writes are slower and more complicated,
    - because a write to a single document may now affect multiple partitions of the index (every term in the document might be on a different partition, on a different node).
  - In practice, updates to global secondary indexes are often asynchronous.

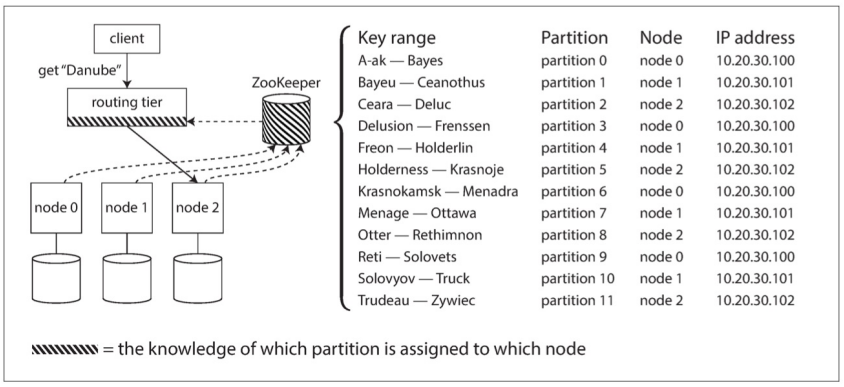
Rebalancing Partitions:

- The process of moving load from one node in the cluster to another is called **rebalancing**.
- **Minimum requirements**:
  - the load (data storage, read and write requests) should be shared fairly between the nodes in the cluster
  - While rebalancing is happening, the database should continue accepting reads and writes.
  - No more data than necessary should be moved between nodes. (fast / minimize load)
- **Strategies for Rebalancing**:
  - **How NOT to do it: hash mod N**
    - The problem with the mod N approach is that if the number of nodes N changes, most of the keys will need to be moved from one node to another.
  - **1, Fixed number of partitions**: used in Riak, Elasticsearch, Couchbase, and Voldemort;
    - create many more partitions than there are nodes, and assign several partitions to each node.
      - if a node is added to the cluster, the new node can steal a few partitions from every existing node until partitions are fairly distributed once again.
    - the number of partitions configured at the outset is the maximum number of nodes you can have
  - **2, Dynamic partitioning**: key range–partitioned databases such as HBase(in HDFS) and RethinkDB create partitions dynamically. **MongoDB**(>2.4)

    - Each partition is assigned to one node, and each node can handle multiple partitions, like in the case of a fixed number of partitions.
    - **An advantage** of dynamic partitioning is that the number of partitions adapts to the total data volume.
    - **A caveat** is that an empty database starts off with a single partition, since there is no a priori information about where to draw the partition boundaries. (mitigated by "pre-splitting")
  - **3, Partitioning proportionally to nodes**: used by **Cassandra** and Ketama
    - in other words, to have a fixed number of partitions per node. This approach also keeps the size of each partition fairly stable.
- **Operations: Automatic or Manual Rebalancing**:
  - Couchbase, Riak, and Voldemort generate a suggested partition assignment automatically, but require an administrator to commit it before it takes effect.
  - Fully automated rebalancing can be convenient, but it is unpredictable. (consider rebalancing is a large overhaul), especially dangerous when combined with automatic failure detection.

Request Routing: e.g. ZooKeeper

- When a client wants to make a request, how does it know which node to connect to?
  - This is an instance of a more general problem called **service discovery**, which isn't limited to just databases. (Common problem among "high availability" network applications)
- Few different approaches:
  - Allow clients to contact any node (e.g., via a round-robin load balancer).
  - Send all requests from clients to a routing tier first, which determines the node that should handle each request and forwards it accordingly.
  - Require that clients be aware of the partitioning and the assignment of partitions to nodes.
- How does the component making the routing decision (which may be one of the nodes, or the routing tier, or the client) learn about changes in the assignment of partitions to nodes?
  - Hard to handle, But many distributed data systems rely on a separate coordination service such as **ZooKeeper** to keep track of this cluster metadata.



Key range	Partition	Node	IP address
A-ak — Bayes	partition 0	node 0	10.20.30.100
Bayeu — Ceanothus	partition 1	node 1	10.20.30.101
Ceara — Deluc	partition 2	node 2	10.20.30.102
Delusion — Frenssen	partition 3	node 0	10.20.30.100
Freon — Holderlin	partition 4	node 1	10.20.30.101
Holderness — Krasnoje	partition 5	node 2	10.20.30.102
Krasnokamsk — Menadra	partition 6	node 0	10.20.30.100
Menage — Ottawa	partition 7	node 1	10.20.30.101
Otter — Rethimnon	partition 8	node 2	10.20.30.102
Reti — Solovets	partition 9	node 0	10.20.30.100
Solovyov — Truck	partition 10	node 1	10.20.30.101
Trudeau — Zywiec	partition 11	node 2	10.20.30.102

Figure 6-8. Using ZooKeeper to keep track of assignment of partitions to nodes.

  - **ZooKeeper** maintains the authoritative mapping of partitions to nodes.Whenever a partition changes ownership, or a node is added or removed, ZooKeeper notifies the

routing tier so that it can keep its routing information up to date.
    - LinkedIn's Espresso uses Helix for cluster management (which in turn relies on ZooKeeper).
    - HBase, SolrCloud, and **Kafka** also use ZooKeeper to track partition assignment
        - **MongoDB** has a similar architecture
        - (C: Latest version of Kafka seems removed ZooKeeper)
    - **Cassandra** and Riak take a different approach: they use a **gossip protocol** among the nodes to disseminate any changes in cluster state.
        - This model puts more complexity in the database nodes but avoids the dependency on an external coordination service such as ZooKeeper.
    - Couchbase does not rebalance automatically, which simplifies the design.
- **Parallel Query Execution**:
    - **massively parallel processing** (MPP) relational database products, often used for analytics, are much more sophisticated in the types of queries they support.

Summary:

- The goal of partitioning is to spread the data and query load evenly across multiple machines, avoiding hot spots (nodes with disproportionately high load).
- two main approaches to partitioning
    - **Key range partitioning**, where keys are sorted, and a partition owns all the keys from some minimum up to some maximum.
        - partitions are typically rebalanced dynamically by splitting
        - Risk of hot spots
    - **Hash partitioning**, where a hash function is applied to each key, and a partition owns a range of hashes.
        - partitioning by hash, it is common to create a fixed number of partitions in advance
        - Destroy the ordering of keys
- A secondary index also needs to be partitioned:
    - **Document-partitioned indexes (local indexes)**, where the secondary indexes are stored in the same partition as the primary key and value.
    - **Term-partitioned indexes (global indexes)**, where the secondary indexes are partitioned separately, using the indexed values.
- Operations that need to write to several partitions can be difficult to reason about: for example, what happens if the write to one partition succeeds, but another fails?

<Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems>

[SYSTEM-DESIGN]

Designing Data Intense Application – Chapter 7:Transactions

Posted by CHARLES *on* 2020-04-12

<Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems>

Some authors have claimed that general two-phase commit is too expensive to support, because of the performance or availability problems that it brings. We believe it is better to have application programmers deal with performance problems due to overuse of transactions as bottlenecks arise, rather than always coding around the lack of transactions.

—James Corbett et al.,**Spanner: Google's Globally-Distributed Database** (2012)

- A **transaction** is a way for an application to group several reads and writes together into a logical unit.
  - Conceptually, all the reads and writes in a transaction are executed as one operation: either the entire transaction succeeds (commit) or it fails (abort, rollback).
- By using transactions, the application is free to ignore certain potential error scenarios and concurrency issues, because the database takes care of them instead (we call these safety guarantees)
  - vs. **higher performance** or **higher availability**
- **How do you figure out whether you need transactions**?
  - In order to answer that question, we first need to understand exactly what safety guarantees transactions can provide, and what costs are associated with them.

The Slippery Concept of a Transaction

- Almost all relational databases today, and some non-relational databases, support transactions. E.g. MySQL, MSSQL, Oracle, PostgreSQL etc.

- **The Meaning of ACID**:
  - The safety guarantees provided by **transactions** are often described by the well-known acronym **ACID**, which stands for **Atomicity, Consistency, Isolation, and Durability**.
  - Systems that do not meet the ACID criteria are sometimes called **BASE**, which stands for **Basically Available, Soft state, and Eventual consistency**.
    - This is even more vague than the definition of ACID.
  - **Atomicity**: atomic refers to something that cannot be broken down into smaller parts
    - If the writes are grouped together into an **atomic transaction**, and the transaction cannot be completed (committed) due to a fault, then the transaction is aborted and the database must discard or undo any writes it has made so far in that transaction.
  - **Consistency**: refers to an application-specific notion of the database being in a "good state."
    - You have certain statements about your data (invariants) that must always be true—e.g. In an accounting system, credits and debits across all accounts must always be balanced.
      - However, this depends on the application's notion of invariants.
    - **Atomicity, isolation, and durability** are properties of the database, whereas **consistency** (in the ACID sense) is a property of the application.
  - **Isolation**: concurrently executing transactions are isolated from each other: they cannot step on each other's toes. (old term: Serializability)
    - In practice, serializable isolation is rarely used, because it carries a performance penalty.
  - **Durability**: the promise that once a transaction has committed successfully, any data it has written will not be forgotten, even if there is a hardware fault or the database crashes.
    - **In a single-node database**, durability typically means that the data has been written to non-volatile storage such as a hard drive or SSD.
    - **In a replicated database**, durability may mean that the data has been successfully copied to some number of nodes.
    - perfect durability does not exist. (e.g. What if earth were destroyed)
  - **Replication and Durability**:
    - In practice, there is no one technique that can provide absolute guarantees. There are only various risk-reduction techniques, including writing to disk, replicating to remote machines, and backups—and they can and should be used together.
- **Single-Object and Multi-Object Operations**:
  - Multi-object transactions are often needed if several pieces of data need to be kept in sync. (e.g. Email app, with "unread" counter, a way of denormalization)
  - **Violating isolation**: one transaction reads another transaction's uncommitted writes (a "dirty read").
  - **Multi-object transactions** require some way of determining which read and write operations belong to the same transaction.
    - On the other hand, many non-relational databases don't have such a way of

grouping operations together.
    - **Single-object writes**:
        - Increment operation, which removes the need for a read-modify-write cycle.
        - **Compare-and-set** and other single-object operations have been dubbed "light-weight transactions" or even "ACID" for marketing purposes
    - **The need for multi-object transactions**:
        - Many distributed datastores have abandoned multi-object transactions because they are difficult to implement across partitions, and they can get in the way in some scenarios where very high availability or performance is required.
        - In a relational data model, a row in one table often has a foreign key reference to a row in another table.
        - In a document data model, the fields that need to be updated together are often within the same document, which is treated as a single object
        - In databases with secondary indexes (almost everything except pure key-value stores), the indexes also need to be updated every time you change a value.
- **Handling errors and aborts**:
    - A key feature of a transaction is that it can be aborted and safely retried if an error occurred.
        - if the database is in danger of violating its guarantee of atomicity, isolation, or durability, it would rather abandon the transaction entirely than allow it to remain half-finished.
        - But datastores with leaderless replication won't obey such rules but work much more on a "best effort" basis, so it's the application's responsibility to recover from errors.
    - Although retrying an aborted transaction is a simple and effective error handling mechanism, it isn't perfect.

Weak Isolation Levels

- If two transactions don't touch the same data, they can safely be run in parallel, because neither depends on the other.
- databases have long tried to hide concurrency issues from application developers by providing **transaction isolation**.
    - **serializable isolation** means that the database guarantees that transactions have the same effect as if they ran serially (i.e., one at a time, without any concurrency)
- **Serializable isolation** has a performance cost, and many databases don't want to pay that price
    - therefore common for systems to use weaker levels of isolation, which protect against some concurrency issues, but not all.
- Rather than blindly relying on tools, we need to develop a good understanding of the kinds of concurrency problems that exist, and how to prevent them.
- **Several weak (non-serializable) isolation levels that are used in practice**:
    - **Read Committed;**
    - **Snapshot Isolation and Repeatable Read;**
    - **Preventing Lost Updates;**

    - **Write Skew and Phantoms;**
- **Read Committed**: The most basic level of transaction isolation with two guarantees(default setting in Oracle 11g, PostgreSQL, SQL Server 2012, MemSQL, and many other databases)
    - 1. When reading from the database, you will only see data that has been committed (no dirty reads).
    - 2. When writing to the database, you will only overwrite data that has been committed (no dirty writes).
    - **No dirty reads**:
        - Imagine a transaction has written some data to the database, but the transaction has not yet committed or aborted. Can another transaction see that uncommitted data? If yes, that is called a dirty read;
    - **No dirty writes**:
        - what happens if the earlier write is part of a transaction that has not yet committed, so the later write overwrites an uncommitted value? This is called a dirty write.
        - by delaying the second write until the first write's transaction has committed or aborted.
    - **Implementing read committed**:
        - Databases **prevent dirty writes** by using **row-level locks**: when a transaction wants to modify a particular object (row or document), it must first acquire a lock on that object.
        - Most databases **prevent dirty reads** using the approach: for every object that is written, the database remembers both the old committed value and the new value set by the transaction that currently holds the write lock.
- **Snapshot Isolation and Repeatable Read**: (it is supported by PostgreSQL, MySQL with the InnoDB storage engine, Oracle, SQL Server, and others)
    - Snapshot isolation is a popular feature;
    - **Non-repeatable read or read skew**: if Alice were to read the balance of account 1 again at the end of the transaction, she would see a different value ($600) than she saw in her previous query.
    - **Snapshot isolation** is the most common solution to this problem. The idea is that each transaction reads from a **consistent snapshot** of the database—that is, the transaction sees all the data that was committed in the database at the start of the transaction.(C: similar with Version Vector?)
        - Even if the data is subsequently changed by another transaction, each transaction sees only the old data from that particular point in time.
    - **Snapshot isolation** is a boon for long-running, read-only queries such as backups and analytics.
    - **Implementing snapshot isolation**:
        - Like read committed isolation, implementations of snapshot isolation typically use write locks to prevent dirty writes;
            - write can block the progress of another transaction that writes to the same object.
            - However, reads do not require any locks.

        - From a performance point of view, a key principle of snapshot isolation is readers never block writers, and writers never block readers.
        - Because it maintains several versions of an object side by side, this technique is known as **Multi-Version Concurrency Control (MVCC)**.
    - **Visibility rules for observing a consistent snapshot**:
        - When a transaction reads from the database, transaction IDs are used to decide which objects it can see and which are invisible.
        - an object is visible if both of the following conditions are true:
            - At the time when the reader's transaction started, the transaction that created the object had already committed.
            - The object is not marked for deletion, or if it is, the transaction that requested deletion had not yet committed at the time when the reader's transaction started.
        - By never updating values in place but instead creating a new version every time a value is changed, the database can provide a consistent snapshot while incurring only a small overhead
        - **Indexes and snapshot isolation**: (?)
            - append-only/copy-on-write variant that does not overwrite pages of the tree when they are updated, but instead creates a new copy of each modified page.
    - **Repeatable read and naming confusion**:
        - Many databases that implement it call it by different names. In Oracle it is called serializable, and in PostgreSQL and MySQL it is called repeatable read.
        - It defines repeatable read, which looks superficially similar to snapshot isolation. (C: Because the concept of "snapshot-isolation" haven't invented yet back then)
        - As a result, nobody really knows what repeatable read means.
- **Preventing Lost Updates**:
    - There are several other interesting kinds of conflicts that can occur between concurrently writing transactions. The best known of these is the **lost update** problem.
    - **The lost update problem can occur** if an application reads some value from the database, modifies it, and writes back the modified value (a read-modify-write cycle).
        - If two transactions do this concurrently, one of the modifications can be lost, because the second write does not include the first modification.
        - E.g.
            - 1, Incrementing a counter or updating an account balance;
            - 2, Making a local change to a complex value;
            - 3, Two users editing a wiki page at the same time
    - **Atomic write operations**: (e.g. MongoDB, Redis)
        - Many databases provide atomic update operations, which remove the need to implement read-modify-write cycles in application code.
            - Not all writes can easily be expressed in terms of atomic operations—for example, updates to a wiki page involve arbitrary text editing
        - **cursor stability**: Atomic operations are usually implemented by taking an exclusive lock on the object when it is read so that no other transaction can read it until the update has been applied.

        - Another option is to simply force all atomic operations to be executed on a single thread.
    - **Explicit locking**:
        - if the database's built-in atomic operations don't provide the necessary functionality, it is for the application to explicitly lock objects that are going to be updated.
            - E.g. The **FOR UPDATE** clause indicates that the database should take a lock on all rows returned by this query.
    - **Automatically detecting lost updates**: (e.g. PostgreSQL, Oracle, SQL)
        - An alternative is to allow them to execute in parallel and, if the transaction manager detects a lost update, abort the transaction and force it to retry its read-modify-write cycle.
    - **Compare-and-set**:
        - In databases that don't provide transactions, you sometimes find an atomic **compare-and-set** operation.
            - avoid lost updates by allowing an update to happen only if the value has not changed since you last read it.
    - **Conflict resolution and replication**:
        - Locks and compare-and-set operations assume that there is a single up-to-date copy of the data. Thus, techniques based on locks or compare-and-set do not apply in this context.
        - **Solution**: allow concurrent writes to create several conflicting versions of a value (also known as siblings), and to use application code or special data structures to resolve and merge these versions after the fact.
        - The last write wins (LWW) conflict resolution method is prone to lost updates and Unfortunately, LWW is the default in many replicated databases.
- **Write Skew and Phantoms**: (e.g. Hospital doctor on call schedule App)
    - **Write skew**. It is neither a dirty write nor a lost update, because the two transactions are updating two different objects (Alice's and Bob's on-call schedule, respectively).
        - You can think of Write skew as a generalization of the lost update problem.
    - **Write skew** can occur if two transactions read the same objects, and then update some of those objects (different transactions may update different objects).
        - Automatically preventing write skew requires true serializable isolation.
        - The second-best option in this case is probably to explicitly lock the rows that the transaction depends on.
    - **More examples of write skew**:
        - Meeting room booking system;
        - Multiplayer game;
        - Claiming a username; (C:reason why website tend to use Email as username, because Email usually guarantee to be unique) Unique constraint could solve this problem;
        - Preventing double-spending;
    - **Phantoms causing write skew**:
        - where a write in one transaction changes the result of a search query in another transaction, is called a **phantom**.

- **Materializing conflicts**: (last resort, A serializable isolation level is much preferable in most cases.)
  - If the problem of phantoms is that there is no object to which we can attach the locks, perhaps we can artificially introduce a lock object into the database?
  - This approach is called **materializing conflicts**, because it takes a phantom and turns it into a lock conflict on a concrete set of rows that exist in the database.

Serializability

- **Serializable isolation** is usually regarded as the strongest isolation level.
  - It guarantees that even though transactions may execute in parallel, the end result is the same as if they had executed one at a time, serially, without any concurrency.
- Three techniques of implementation: (Focus one node in this chapter)
  - Literally executing transactions in a **Serial-order**;
  - **Two-phase locking**, which for several decades was the only viable option;
  - **Optimistic concurrency control** techniques such as **serializable snapshot isolation**;
- **Actual Serial Execution**: (e.g. Redis, VoltDB/H-Store, Datomic)
  - The simplest way of avoiding concurrency problems is to**remove the concurrency entirely**: to execute only one transaction at a time, in serial order, on a single thread.
  - Two Reasons why this starting pick momentum late until 2007:
    - RAM became cheap enough that for many use cases is now feasible to keep the entire active dataset in memory.
    - Database designers realized that OLTP transactions are usually short and only make a small number of reads and writes
  - Sometimes perform better than a system that supports concurrency, because it can avoid the coordination overhead of locking, but its throughput is limited to that of a single CPU core.
  - **Encapsulating transactions in stored procedures**:
    - systems with single-threaded serial transaction processing don't allow interactive multi-statement transactions.
      - Instead, the application must submit the entire transaction code to the database ahead of time, as a **stored procedure**.
    - **Pros and cons of stored procedures**:
      - Each database vendor has its own language for stored procedures (Oracle has PL/ SQL, SQL Server has T-SQL, PostgreSQL has PL/pgSQL, etc.).
      - Code running in a database is difficult to manage: compared to an application server;
      - A database is often much more performance-sensitive than an application server, because a single database instance is often shared by many application servers. (C: wider "negative" impact if a badly written SP)
    - Modern implementations of stored procedures have abandoned PL/SQL and use existing general-purpose programming languages instead: VoltDB uses Java or Groovy, Datomic uses Java or Clojure, and Redis uses Lua.
  - **Partitioning**:
    - For applications with high write throughput, the single-threaded transaction

processor can become a serious bottleneck.
    - If you can find a way of partitioning your dataset so that each transaction only needs to read and write data within a single partition, then each partition can have its own transaction processing thread running independently from the others.
    - Whether transactions can be single-partition depends very much on the structure of the data used by the application.
  - **Summary of serial execution**:
    - a viable way of achieving serializable isolation within certain constraints:
      - Every transaction must be small and fast, because it takes only one slow transaction to stall all transaction processing.
      - It is limited to use cases where the active dataset can fit in memory. (C: use anti-caching if the dataset is in disk)
      - Write throughput must be low enough to be handled on a single CPU core or transactions need to be partitioned without requiring cross-partition coordination.
      - Cross-partition transactions are possible, but there is a hard limit to the extent to which they can be used.
- **Two-Phase Locking (2PL)**: (2PL is not 2PC)
  - Several transactions are allowed to concurrently read the same object as long as nobody is writing to it. But as soon as anyone wants to write (modify or delete) an object, exclusive access is required:
    - If transaction A has read an object and transaction B wants to write to that object, B must wait until A commits or aborts before it can continue. (This ensures that B can't change the object unexpectedly behind A's back.)
    - If transaction A has written an object and transaction B wants to read that object, B must wait until A commits or aborts before it can continue. (Reading an old version of the object, like in Figure 7-1, is not acceptable under 2PL.)
  - In 2PL, writers don't just **block other writers; they also block readers**and vice versa.
    - because 2PL provides serializability, it protects against all the race conditions discussed earlier, including lost updates and write skew.
  - **Implementation of two-phase locking**:
    - Shared and Exclusive Lock;
    - The first phase (while the transaction is executing) is when the locks are acquired, and the second phase (at the end of the transaction) is when all the locks are released.
    - Since so many locks are in use, it can happen quite easily that transaction A is stuck waiting for transaction B to release its lock, and vice versa. → **deadlock**
      - The database automatically detects deadlocks between transactions and aborts one of them so that the others can make progress.
      - Application need retry after abort
  - **Performance of two-phase locking**:
    - transaction throughput and response times of queries are significantly worse under two-phase locking than under weak isolation.
    - databases running 2PL can have quite unstable latencies, and they can be very

slow at high percentiles

- **Predicate locks**:
  - A database with serializable isolation must prevent phantoms.
  - **Predicate lock**: It works similarly to the shared/exclusive lock described earlier, but rather than belonging to a particular object (e.g., one row in a table), it belongs to all objects that match some search condition.
  - A predicate lock applies even to objects that do not yet exist in the database, but which might be added in the future (phantoms).
- **Index-range locks**:
  - Unfortunately, predicate locks do not perform well: if there are many locks by active transactions, checking for matching locks becomes time-consuming.
  - Most databases with 2PL actually implement **index-range locking** (also known as next-key locking), which is a simplified approximation of predicate locking.
  - Index-range locks are not as precise as predicate locks would be, but since they have much lower overheads, they are a good compromise.

- **Serializable Snapshot Isolation (SSI)**: (new default in the future)
  - It provides full **serializability**, but has only a small performance penalty compared to **snapshot isolation**.
    - SSI is fairly new: it was first described in 2008 and is the subject of Michael Cahill's PhD thesis;
  - **Pessimistic vs. Optimistic concurrency control**:
    - Two-phase locking is a so-called **pessimistic** concurrency control mechanism;
      - Serial execution is, in a sense, pessimistic to the extreme.
    - Serializable snapshot isolation is an **optimistic** concurrency control technique
  - Contention can be reduced with commutative atomic operations
  - SSI is based on snapshot isolation—that is, all reads within a transaction are made from a consistent snapshot of the database.
    - On top of snapshot isolation,**SSI adds an algorithm for detecting serialization conflicts among writes and determining which transactions to abort**
  - **Decisions based on an outdated premise**:
    - How does the database know if a query result might have changed? There are two cases to consider:
      - **Detecting reads** of a stale MVCC object version (uncommitted write occurred before the read)
      - **Detecting writes** that affect prior reads (the write occurs after the read)
  - **Detecting stale MVCC reads**:
    - In order to prevent this anomaly, the database needs to track when a transaction ignores another transaction's writes due to MVCC visibility rules.

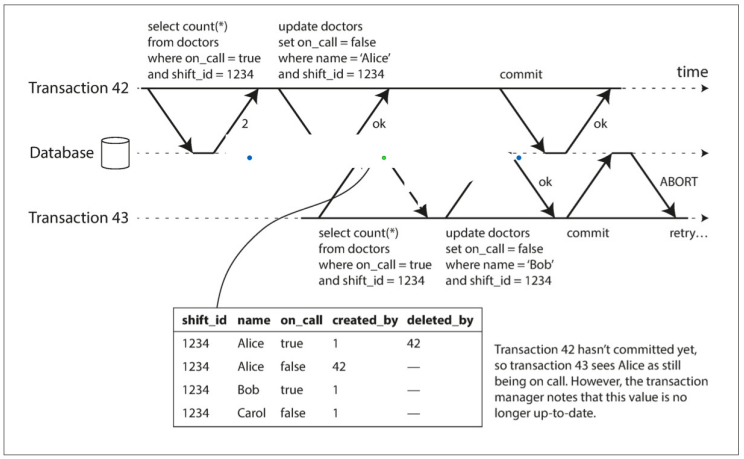


Figure 7-10. Detecting when a transaction reads outdated values from an MVCC snapshot.

- By avoiding unnecessary aborts, SSI preserves snapshot isolation's support for long-running reads from a consistent snapshot.

- **Detecting writes that affect prior reads**:
  - The second case to consider is when another transaction modifies data after it has been read.

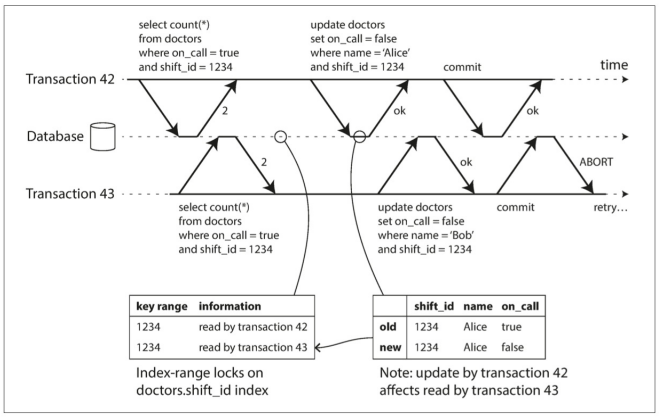


Figure 7-11. In serializable snapshot isolation, detecting when one transaction modifies another transaction's reads.

  - When a transaction writes to the database, it must look in the indexes for any other transactions that have recently read the affected data.
    - This process is similar to acquiring a write lock on the affected key range, but rather than blocking until the readers have committed, the lock acts as a **tripwire**: it simply notifies the transactions that the data they read may no longer be up to date.

- **Performance of serializable snapshot isolation**:
  - Compared to two-phase locking, the big advantage of serializable snapshot isolation is that one transaction doesn't need to block waiting for locks held by another transaction.
    - Like under snapshot isolation, writers don't block readers, and vice versa.
  - The rate of aborts significantly affects the overall performance of SSI.

Summary

- Transactions are an abstraction layer that allows an application to pretend that certain concurrency problems and certain kinds of hardware and software faults don't exist.
- Concurrency control: (isolation levels)

    - Read committed,
    - Snapshot isolation (sometimes called Repeatable read),
    - Serializable.
- Race conditions:
    - Dirty reads
    - Dirty writes
    - Read skew (non-repeatable reads)
        - Solved by: snapshot isolation which implemented with MVCC;
    - Lost updates
    - Write skew
        - Solved by: serializable isolation
    - Phantom reads
        - Solved by: index-range locks
- Only Serializable Isolation protects against all of these issues. We discussed three different approaches to implementing serializable transactions:
    - Literally executing transactions in a serial order
    - Two-phase locking
    - Serializable snapshot isolation (**SSI**) (C:best approach)

<Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems>

PREVIOUS POST

Designing Data Intense Application – Chapter 6: Partitioning

NEXT POST

Designing Data Intense Application – Chapter 8: The Trouble with Distributed Systems

[SYSTEM-DESIGN]

Designing Data Intense Application – Chapter 8: The Trouble with Distributed Systems

Posted by CHARLES *on* 2020-04-14

<Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems>

- This chapter is a thoroughly pessimistic and depressing overview of things that may go wrong in a distributed system.
  - Networks issues
  - Clocks & timing issues

Faults and Partial Failures

- Single machine software is **deterministic**;
  - An individual computer with good software is usually either fully functional or entirely broken, but not something in between.
- In distributed systems, we are no longer operating in an idealized system model—we have no choice but to confront the messy reality of the physical world.
- The difficulty is that **partial failures** are **non-deterministic**: if you try to do anything involving multiple nodes and the network, it may sometimes work and sometimes unpredictably fail.
  - This **non-determinism** and possibility of **partial failures** is what makes distributed systems hard to work with.
- **Cloud-Computing and Super-Computing**: (C: Vertical scaling vs. Horizontal Scaling)
  - High-performance computing (HPC) vs. Cloud Computing
    - supercomputer is more like a single-node computer than a distributed system: it deals with partial failure by letting it escalate into total failure
  - If we want to make distributed systems work, we must accept the possibility of partial failure and build fault-tolerance mechanisms into the software.

    - build a reliable system from unreliable components.
  - Even in smaller systems consisting of only a few nodes, it's important to think about partial failure.
  - The fault handling must be part of the software design, and you (as operator of the software) need to know what behavior to expect from the software in the case of a fault.
    - important to consider a wide range of possible faults—even fairly unlikely ones.
  - In distributed systems, **suspicion**, **pessimism**, and **paranoia** pay off.

Unreliable Networks

- the distributed systems we focus on in this book are**shared-nothing systems**: i.e., a bunch of machines connected by a network.
  - Why ?
    - it's comparatively cheap because it requires no special hardware,
    - it can make use of commoditized cloud computing services, and
    - it can achieve high reliability through redundancy across multiple geographically distributed datacenters.
- The internet and most internal networks in data centers (often Ethernet) are asynchronous packet networks. If you send a request and expect a response, many things could go wrong.

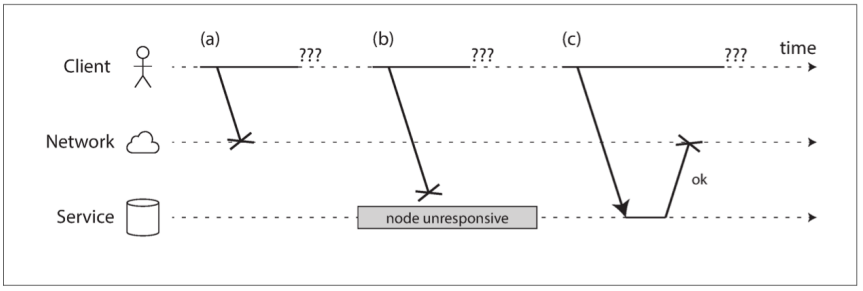


Figure 8-1. If you send a request and don't get a response, it's not possible to distinguish whether (a) the request was lost, (b) the remote node is down, or (c) the response was lost.

  - If you send a request to another node and don't receive a response, it is impossible to tell why.
    - One Solution: use Timeout.
- **Network Faults in Practice**:
  - Nevertheless, nobody is immune from network problems. Handling network faults doesn't necessarily mean tolerating them.
  - You do need to know how your software reacts to network problems and ensure that the system can recover from them.
    - "**Chaos Monkey**"(e.g. Netflix) is needed to test this "**Reliability**".
- **Detecting Faults**: (C: **Gossip** Protocol, send request/signal across each other and then broadcasting; or **ZooKeeper**)
  - Many systems need to automatically detect faulty nodes.
  - Unfortunately, the uncertainty about the network makes it difficult to tell whether a node is working or not.
- **Timeouts and Unbounded Delays**:
  - If a timeout is the only sure way of detecting a fault, then how long should the

timeout be?
    - There is unfortunately no simple answer.
- **Prematurely declaring a node dead is problematic**:
    - if the node is actually alive and in the middle of performing some action (for example, sending an email), and another node takes over, the action may end up being performed twice.
    - When a node is declared dead, its responsibilities need to be transferred to other nodes, which places additional load on other nodes and the network.
- **Asynchronous networks have unbounded delays** (that is, they try to deliver packets as quickly as possible, but there is no upper limit on the time it may take for a packet to arrive)
- **Network congestion and queueing**:
    - The variability of packet delays on computer networks is most often due to queueing. (e.g. Network, CPU, VMs, TCP flow-control etc)
    - **TCP** considers a packet to be lost if it is not acknowledged within some timeout (which is calculated from observed round-trip times), and lost packets are automatically retransmitted.
    - **TCP vs. UDP**: (C: delayed message meaningless/worthless → UDP)
        - Trade-off between reliability and variability of delays: as UDP does not perform flow control and does not retransmit lost packets.
    - You can only choose timeouts experimentally.
    - Systems can continually measure response times and their variability (jitter), and automatically adjust: Phi Accrual failure detector, which is used for example in Akka and Cassandra. TCP retransmission timeouts also work similarly.
- **Synchronous vs. Asynchronous Networks**:
    - Phone call network is **synchronous**: even as data passes through several routers, it does not suffer from queueing, because the 16 bits of space for the call have already been **reserved** in the next hop of the network.
        - **bounded delay**: No queueing, the maximum end-to-end latency of the network is fixed.
    - Can we not simply make network delays predictable?
        - Ethernet and IP are packet-switched protocols, which suffer from queueing and thus unbounded delays in the network. These protocols do not have the concept of a circuit.
        - Why do datacenter networks and the internet use packet switching?
            - The answer is that they are **optimized for bursty traffic**.
    - With careful use of quality of service (QoS, prioritization and scheduling of packets) and admission control (rate-limiting senders), it is possible to emulate circuit switching on packet networks, or provide statistically bounded delay.
    - **Latency and Resource Utilization**:
        - Phone line: resource is divided up in a static way
        - Internet: shares network bandwidth dynamically
            - This approach has the downside of queueing, but the advantage is that it **maximizes utilization of the wire**.
            - A similar situation arises with CPUs.

- **Latency guarantees** are achievable in certain environments, if resources are statically partitioned (e.g., dedicated hardware and exclusive bandwidth allocations).
  - Comes at the cost of reduced utilization
- **Variable delays** in networks are not a law of nature, but simply the result of a **cost/benefit trade-off**.

Unreliable Clocks

- **Clocks and Time** are important. Applications depend on clocks in various ways.
- In a distributed system, time is a tricky business, because communication is not instantaneous.
- Each machine on the network has its own clock, which is an actual hardware device: usually a **quartz crystal oscillator**. But these devices are **not perfectly accurate**, so each machine has its own notion of time.
  - It is possible to synchronize clocks to some degree: the most commonly used mechanism is the **Network Time Protocol (NTP)**
- **Monotonic vs. Time-of-Day Clocks**: Modern computers have at least two different kinds of clocks: a **time-of-day** clock and a **monotonic** clock.
  - **Time-of-day clocks**: it returns the current date and time according to some calendar (also known as wall-clock time).
    - Time-of-day clocks are usually synchronized with NTP, which means that a timestamp from one machine (ideally) means the same as a timestamp on another machine.
  - **Monotonic clocks**: is suitable for measuring a duration (time interval), such as a timeout or a service's response time.
    - they are **guaranteed to always move forward** (whereas a time-of- day clock may jump back in time).
    - NTP may adjust the frequency at which the monotonic clock moves forward (this is known as slewing the clock) if it detects that the computer's local quartz is moving faster or slower than the NTP server.
- **Clock Synchronization and Accuracy**:
  - Monotonic clocks don't need synchronization, but time-of-day clocks need to be set according to an **NTP** server or other external time source in order to be useful.
    - Hardware clocks and NTP can be fickle beasts.
  - It is possible to achieve very good clock accuracy if you care about it sufficiently to invest significant resources.
    - E.g. using GPS receivers, the Precision Time Protocol (PTP), and careful deployment and monitoring
- **Relying on Synchronized Clocks**:
  - Robust software needs to be prepared to deal with incorrect clocks.
    - incorrect clocks easily go unnoticed.
  - If some piece of software is relying on an accurately synchronized clock, the result is more likely to be silent and subtle data loss than a dramatic crash.
  - Thus, if you use software that requires synchronized clocks, it is essential that you

also carefully monitor the clock offsets between all the machines.

- **Timestamps for ordering events**:
  - Consider one particular situation in which it is tempting, but dangerous, to rely on clocks: ordering of events across multiple nodes. E.g. (LWW)

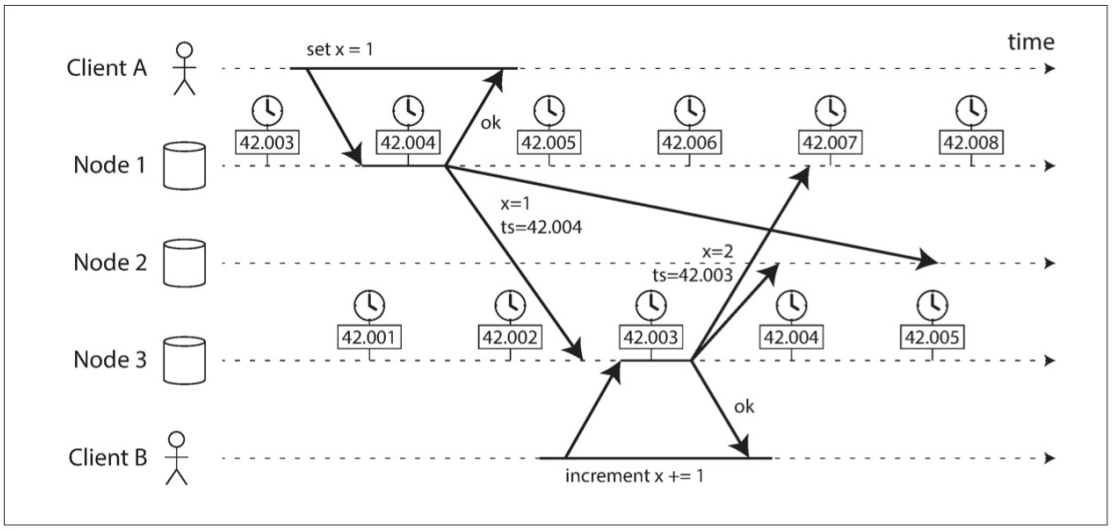


Figure 8-3. The write by client B is causally later than the write by client A, but B's write has an earlier timestamp.

- This conflict resolution strategy is called last write wins (LWW), and it is widely used in both multi-leader replication and leaderless databases such as Cassandra and Riak. (C: write to cache first?)
  - DB writes can mysteriously disappear;
  - LWW can't distinguish between writes that occurred sequentially in quick succession;
  - It is possible for two nodes to independently generate writes with the same timestamp
- For correct ordering, You would **need the clock source to be significantly more accurate than the thing you are measuring** (namely network delay).
- So-called **logical clocks**, which are based on incrementing counters rather than an oscillating quartz crystal, are a safer alternative for ordering events:
  - Logical clocks only measure the relative ordering of events (whether one event happened before or after another).
- **Clock readings have a confidence interval**:
  - Thus, it doesn't make sense to think of a clock reading as a point in time—it is more like a **range of times**, within a confidence interval.
  - Google's TrueTime API in Spanner, which explicitly reports the confidence interval on the local clock.
    - When you ask it for the current time, you get back two values: [earliest, latest]
- **Synchronized clocks for global snapshots**:
  - With lots of small, rapid transactions, creating transaction IDs in a distributed system becomes an untenable bottleneck.
  - Can we use the timestamps from synchronized time-of-day clocks as transaction IDs?
    - If we could get the synchronization good enough, they would have the right

properties: later transactions have a higher timestamp.
    - Spanner needs to keep the clock uncertainty as small as possible;
        - For this purpose, Google deploys a **GPS receiver or atomic clock** in each datacenter, allowing clocks to be synchronized to within about **7 ms**.
- **Process Pauses**:
    - Is it crazy to assume that a thread might be paused for so long? Unfortunately not. E.g.
        - Many programming language runtimes (such as the Java Virtual Machine) have a garbage collector (GC) that occasionally needs to stop all running threads.
        - In virtualized environments, a virtual machine can be suspended (pausing the execution of all processes and saving the contents of memory to disk) and resumed (restoring the contents of memory and continuing execution).
        - If the application performs synchronous disk access, a thread may be paused waiting for a slow disk I/O operation to complete
    - All of these occurrences can preempt the running thread at any point and resume it at some later time, without the thread even noticing.
- **Response time guarantees**:
    - In some systems, there is a specified deadline by which the software must respond; if it doesn't meet the deadline, that may cause a failure of the entire system. These are so-called **Hard real-time systems**.
    - A **real-time operating system (RTOS)** that allows processes to be scheduled with a guaranteed allocation of CPU time in specified intervals is needed;
    - For most server-side data processing systems, real-time guarantees are simply not economical or appropriate.
- **Limiting the impact of garbage collection**:
    - An emerging idea is to **treat GC pauses like brief planned outages** of a node, and to let other nodes handle requests from clients while one node is collecting its garbage.

Knowledge, Truth, and Lies

- For Distributed System: there is **no shared memory**, only **message passing** via an unreliable network with variable delays, and the systems may suffer from partial failures, unreliable clocks, and processing pauses.
    - A node in the network cannot know anything for sure—it can only make guesses based on the messages it receives (or doesn't receive) via the network.
    - A node can only find out what state another node is in (what data it has stored, whether it is correctly functioning, etc.) by exchanging messages with it.
- Although it is possible to make software well behaved in an unreliable system model, it is not straightforward to do so.
- **The Truth Is Defined by the Majority**:
    - Network with an asymmetric fault: a node is able to receive all messages sent to it, but any outgoing messages from that node are dropped or delayed.
    - In a distributed system a node cannot necessarily trust its own judgment of a situation.
    - Instead, many distributed algorithms rely on a **Quorum**, that is, voting among the

nodes: decisions require some minimum number of votes from several nodes in order to reduce the dependence on any one particular node.

- Most commonly, the quorum is an absolute **majority of more than half the nodes**.

- **The leader and the lock**:

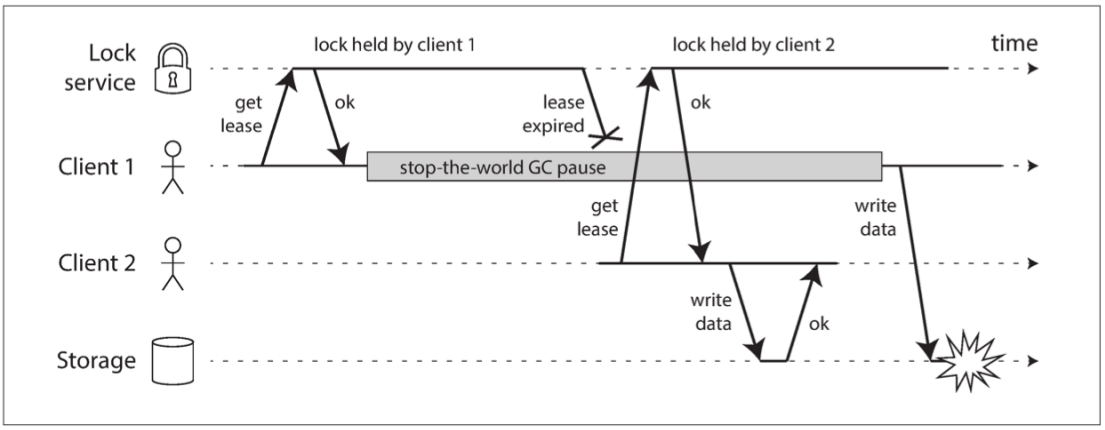


Figure 8-4. Incorrect implementation of a distributed lock: client 1 believes that it still has a valid lease, even though it has expired, and thus corrupts a file in storage.

- **Fencing tokens**:
  - we need to ensure that a node that is under a false belief of being "the chosen one" cannot disrupt the rest of the system. A fairly simple technique that achieves this goal is called **fencing**.

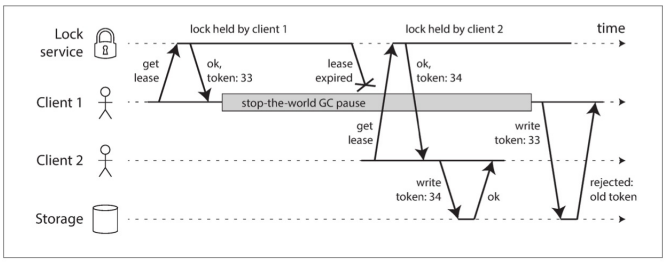


Figure 8-5. Making access to storage safe by allowing writes only in the order of increasing fencing tokens.

  - If ZooKeeper is used as a lock service, the transaction ID zxid or the node version cversion can be used as a fencing token.

- **Byzantine Faults**:
  - Distributed systems problems become much harder if there is a risk that nodes may "**lie**" (send arbitrary faulty or corrupted responses)
    - E.g. if a node may claim to have received a particular message when in fact it didn't. (Such behavior is known as a **Byzantine fault**, and the problem of reaching consensus in this untrusting environment is known as the **Byzantine Generals Problem**)
  - A system is **Byzantine fault-tolerant** if it continues to operate correctly even if some of the nodes are malfunctioning and not obeying the protocol, or if malicious attackers are interfering with the network.
  - **Weak forms of lying**:
    - it can be worth adding mechanisms to software that guard against weak forms of "lying"
      - E.g. invalid messages due to hardware issues, software bugs, and misconfiguration

        - Solutions: e.g.
            - **checksums** in the application-level protocol;
            - A publicly accessible application must carefully **sanitize** any inputs from users;
            - NTP clients can be configured with multiple server addresses.
- **System Model and Reality**:
    - Many algorithms have been designed to solve distributed systems problems,
        - Algorithms need to tolerate the various faults of distributed systems that we discussed in this chapter.
        - Algorithms need to be written in a way that does not depend too heavily on the details of the hardware and software configuration on which they are run.
    - This requires that we somehow formalize the kinds of faults that we expect to happen in a system. We do this by defining a **system model**, which is an abstraction that describes what things an algorithm may assume.
    - **Timing assumptions**, three system models are in common use
        - **Synchronous** model: assumes bounded network delay, bounded process pauses, and bounded clock error.
        - **Partially synchronous** model*: means that a system behaves like a synchronous system most of the time, but it sometimes exceeds the bounds for network delay, process pauses, and clock drift.
        - **Asynchronous** model: an algorithm is not allowed to make any timing assumptions—in fact, it does not even have a clock (so it cannot use timeouts)
    - **Node failures**. The three most common system models for nodes are:
        - **Crash-stop faults**:an algorithm may assume that a node can fail in only one way, namely by crashing.
        - **Crash-recovery faults***: We assume that nodes may crash at any moment, and perhaps start responding again after some unknown time.
        - **Byzantine (arbitrary) faults**: Nodes may do absolutely anything, including trying to trick and deceive other nodes.
    - For modeling real systems, the partially synchronous model with crash-recovery faults is generally **the most useful model**.
    - **Correctness of an algorithm**:
        - **An algorithm is correct in some system model** if it always satisfies its properties in all situations that we assume may occur in that system model.
    - **Safety and Liveness**:
        - What distinguishes the two kinds of properties?
            - A giveaway is that liveness properties often include the word "**eventually**" in their definition. (And yes, you guessed it— eventual consistency is a liveness property.)
        - **Safety:** is often informally defined as **nothing bad happens**,
        - **Liveness:** as something **good eventually happens**.
        - In the example, **uniqueness** and **monotonic sequence** are safety properties, but **availability** is a liveness property.
        - **If a safety property is violated**, we can point at a particular point in time at which it was broken (e.g., if the uniqueness property was violated, we can identify the particular operation in which a duplicate fencing token was returned).

        - After a safety property has been violated, the violation cannot be undone—the damage is already done.
    - **A liveness property works the other way round**: it may not hold at some point in time (e.g., a node may have sent a request but not yet received a response),
        - but there is always hope that it may be satisfied in the future (namely by receiving a response).
    - An advantage of distinguishing between **safety** and **liveness** properties is that it helps us deal with difficult system models.
- **Mapping system models to the real world**:
    - **Abstract system models** are incredibly helpful for distilling down the complexity of real systems to a manageable set of faults that we can reason about, so that we can understand the problem and try to solve it systematically.
    - Proving an algorithm correct does not mean its implementation on a real system will necessarily always behave correctly.

Summary

- In this chapter we have discussed a wide range of problems that can occur in distributed systems.
    - Network delay;
    - Node out of sync (even with NTP)
    - Pause of execution (maybe due to GC)
- Such **partial failures** can occur is the defining characteristic of distributed systems.
- In distributed systems, we try to build tolerance of partial failures into software, so that the system as a whole may continue functioning even when some of its constituent parts are broken.
- To tolerate faults, the first step is to detect them, but even that is hard.
- Once a fault is detected, making a system tolerate it is not easy either: there is no global variable, no shared memory, no common knowledge or any other kind of shared state between the machines. (e.g. Solution: Quorum to agree)
- If you can avoid opening Pandora's box and simply keep things on a single machine, it is generally worth doing so.

<Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems>

[SYSTEM-DESIGN]

Designing Data Intense Application – Chapter 9: Consistency and Consensus

Posted by CHARLES *on* 2020-04-18

<Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems>

Is it better to be alive and wrong or right and dead? —Jay Kreps, A Few Notes on Kafka and Jepsen (2013)

- In this chapter, we will talk about some examples of algorithms and protocols for building **fault-tolerant distributed systems**.
- We will assume that all the problems from Chapter 8 can occur:
  - packets can be lost, reordered, duplicated,
  - arbitrarily delayed in the network;
  - clocks are approximate at best;
  - and nodes can pause (e.g., due to garbage collection) or crash at any time.
- The best way of building fault-tolerant systems is to find some**general-purpose abstractions with useful guarantees**, implement them once, and then let applications rely on those guarantees.
- This is the same approach as we used with transactions in Chapter 7: by using a transaction, the application can pretend:
  - that there are no crashes (**atomicity**),
  - that nobody else is concurrently accessing the database (**isolation**),
  - that storage devices are perfectly reliable (**durability**).
- One of the most important abstractions for distributed systems is**consensus**: that is, getting all of the nodes to agree on something.
  - Once you have an implementation of consensus, applications can use it for various purposes.
  - Correct implementations of consensus help avoid problems.

Consistency Guarantees (Weak vs. Strong)

- Inconsistencies occur no matter what replication method the database uses (single-leader, multi-leader, or leaderless replication).
  - Most replicated databases provide at least **eventual consistency**. A better name for eventual consistency may be **convergence**, as we expect all replicas to eventually converge to the same value.
  - **Weak guarantee**: it doesn't say anything about when the replicas will converge.
- **Eventual consistency** is hard for application developers because it is so different from the behavior of variables in a normal single-threaded program.
- Systems with **Stronger guarantees** may have worse performance or be less fault-tolerant than systems with weaker guarantees.
- **Distributed consistency models** vs. **Hierarchy of transaction isolation levels**
  - **transaction isolation** is primarily about avoiding race conditions due to concurrently executing transactions;
  - **distributed consistency** is mostly about coordinating the state of replicas in the face of delays and faults.

Linearizability(C: SLOW,not used very often in practice)

- Idea behind **Linearizability** (also known as **atomic consistency**, **strong consistency**, **immediate consistency**, or **external consistency**)
  - **The basic idea is to make a system appear as if there were only one copy of the data, and all operations on it are atomic**. (even though there may be multiple replicas in reality, the application does not need to worry about them.)
- **In a linearizable system**, as soon as one client successfully completes a write, all clients reading from the database must be able to see the value just written.
  - In other words, linearizability is a **recency guarantee**.

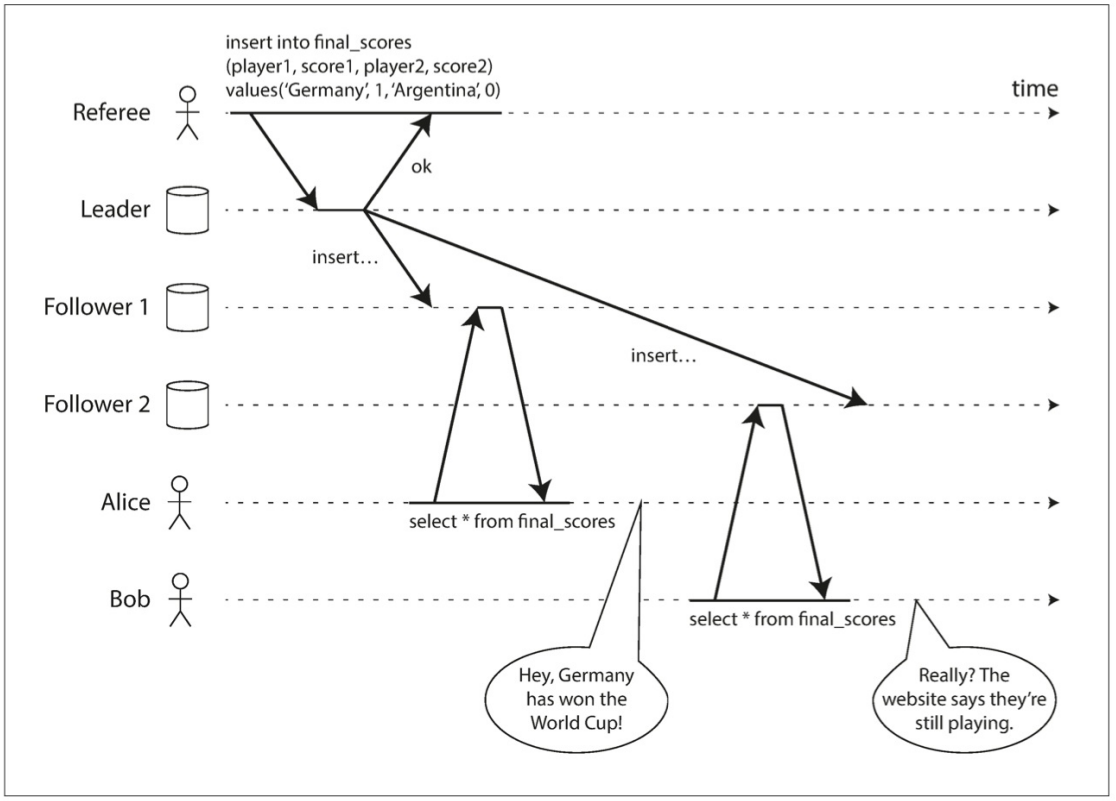


Figure 9-1. This system is not linearizable, causing football fans to be confused.

- **What Makes a System Linearizable**? To make a system appear as if there is only a single copy of the data.

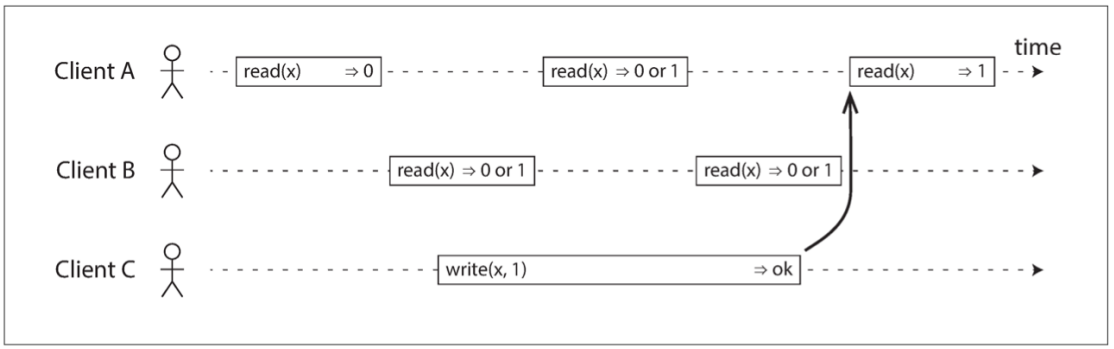


Figure 9-2. If a read request is concurrent with a write request, it may return either the old or the new value.

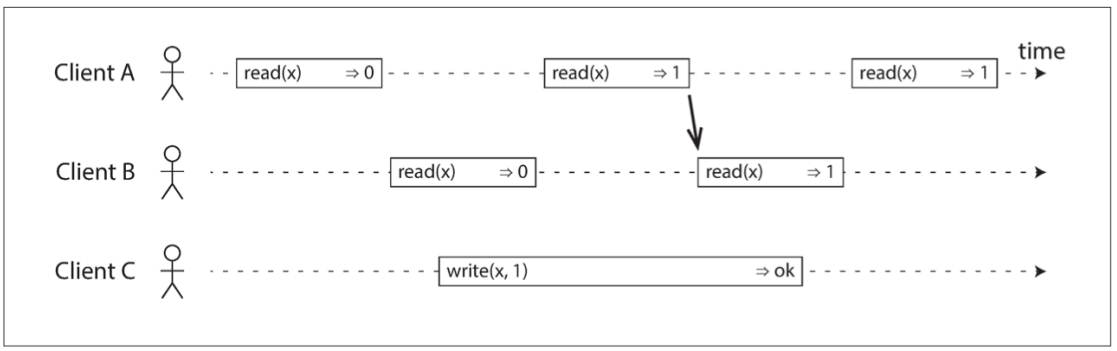


Figure 9-3. After any one read has returned the new value, all following reads (on the same or other clients) must also return the new value.

- **atomic compare-and-set** (**cas**);

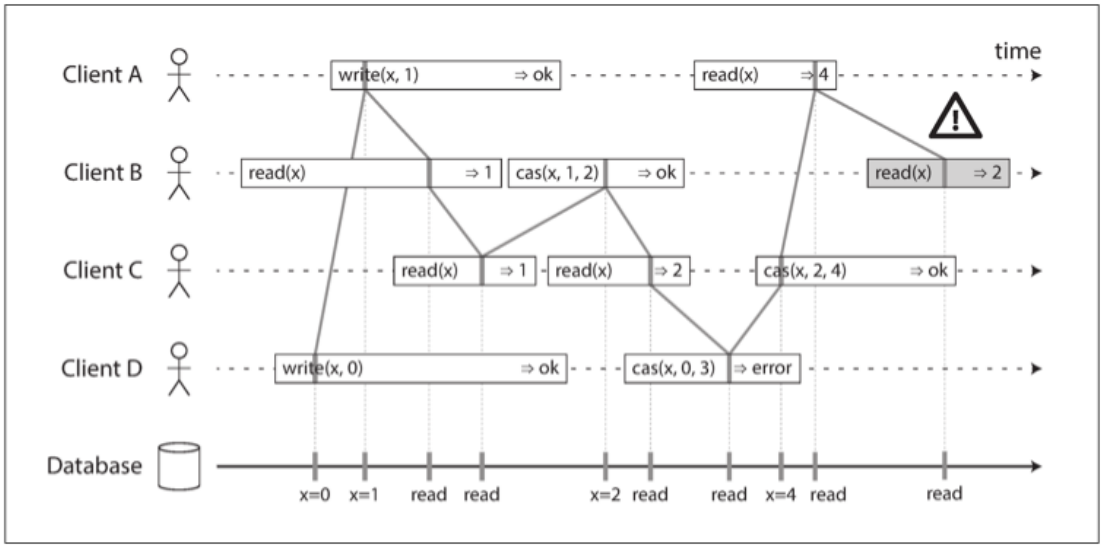


Figure 9-4. Visualizing the points in time at which the reads and writes appear to have taken effect. The final read by B is not linearizable.

- **Linearizability vs. Serializability**:
  - **Serializability** is an **isolation property of transactions**, where every transaction may read and write multiple objects (rows, documents, records).
  - **Linearizability** is a recency guarantee on reads and writes of a register (an individual object).
    - 2PL & actual serial execution are typically linearizable.
    - **serializable snapshot isolation** is **NOT** linearizable.
  - A database may provide both **serializability** and **linearizability**, and this combination is known as strict serializability or strong one-copy serializability.
- **Relying on Linearizability**:
  - In what circumstances is linearizability useful?
  - **Locking and leader election**: (e.g. ZooKeeper, etcd)
    - Lock must be linearizable: all nodes must agree which node owns the lock; otherwise it is useless.
    - Distributed locking is also used at a much more granular level in some distributed databases, such as Oracle Real Application Clusters (RAC)
- **Constraints and Uniqueness guarantees**:
  - If you want to enforce this constraint as the data is written (such that if two people try to concurrently create a user or a file with the same name, one of them will be returned an error), you need **linearizability**. (C: kind like “lock”)
  - a hard uniqueness constraint, such as the one you typically find in relational databases, requires linearizability.
- **Cross-channel timing dependencies**:
  - The linearizability violation was only noticed because there was an additional communication channel in the system.

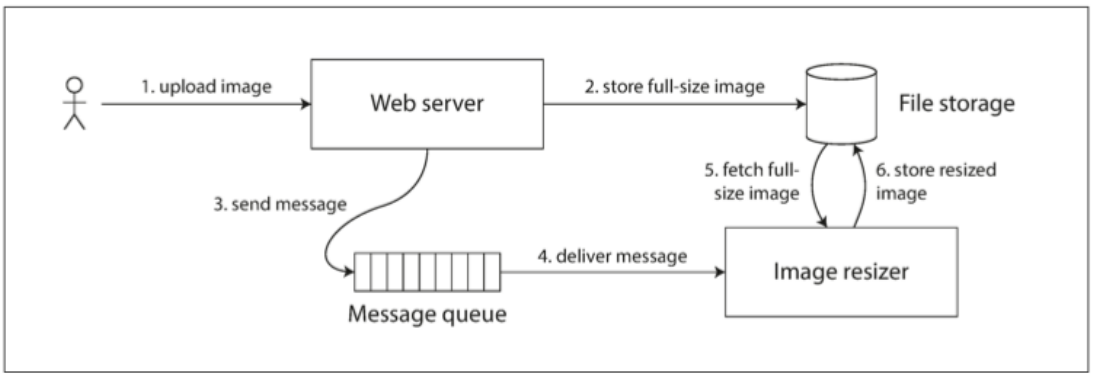


Figure 9-5. The web server and image resizer communicate both through file storage and a message queue, opening the potential for race conditions.

- If the file storage service is linearizable, then this system should work fine.
- If it is not linearizable, there is the risk of a race condition:
  - the message queue (steps 3 and 4 in Figure 9-5) might be faster than the internal replication inside the storage service.
- This problem arises because there are two different communication channels between the web server and the resizer: the file storage and the message queue.
- **Implementing Linearizable Systems**:
  - **Replication & Linearizable**:
    - Single-leader replication (potentially linearizable);
    - Consensus algorithms (linearizable) [e.g. ZooKeeper and etcd]
    - Multi-leader replication (not linearizable)
    - Leaderless replication (probably not linearizable)
  - **Linearizability and quorums**:
    - it seems as though strict quorum reads and writes should be linearizable in a Dynamo-style model. However, when we have variable network delays, it is possible to have race conditions.

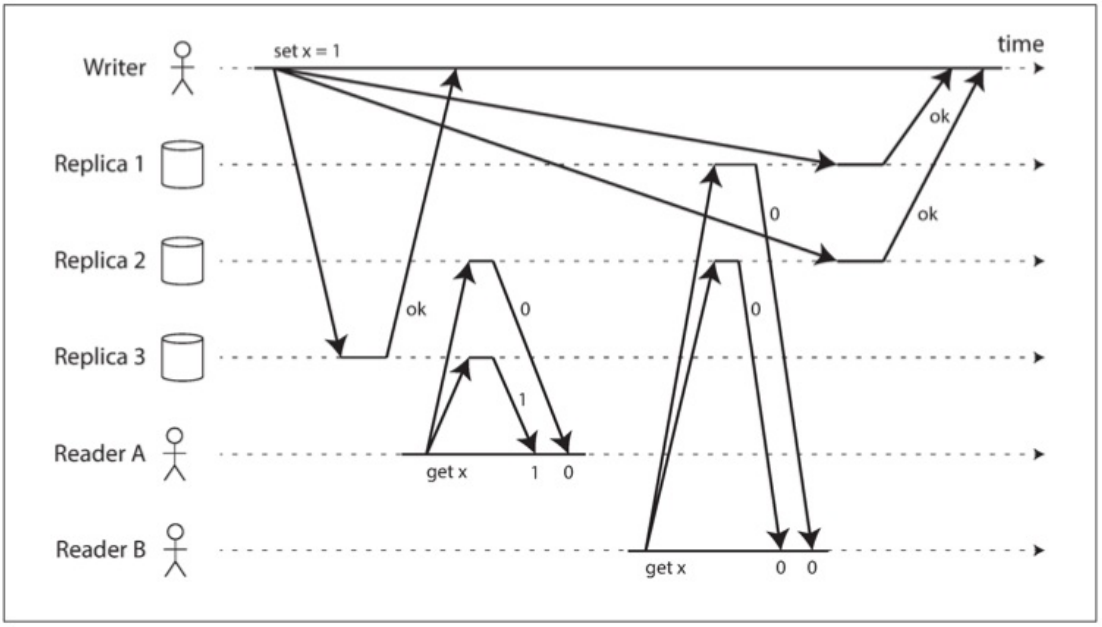


Figure 9-6. A nonlinearizable execution, despite using a strict quorum.

- it is safest to assume that a leaderless system with Dynamo-style replication does not

provide linearizability.

- **The Cost of Linearizability**:

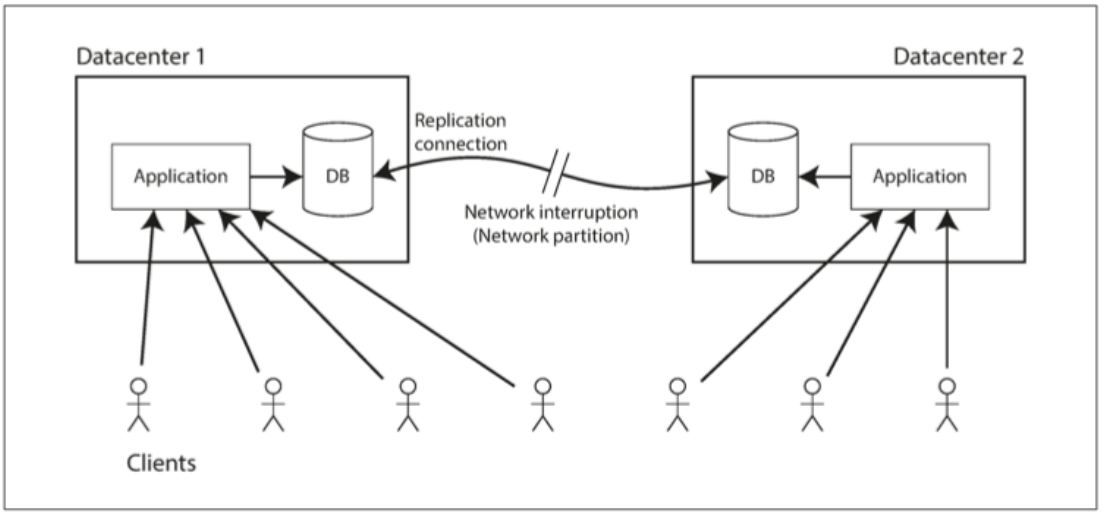


Figure 9-7. A network interruption forcing a choice between linearizability and availability.

- **The CAP theorem**: (very narrow scope)
  - CAP is sometimes presented as **Consistency**, **Availability**, **Partition tolerance**: pick 2 out of 3.
  - **The Unhelpful CAP Theorem**:
    - Unfortunately, putting it this way is misleading because **network partitions are a kind of fault**, so they aren't something about which you have a choice: **they will happen whether you like it or not**.
    - A better way of phrasing CAP would be either **Consistent** or **Available** when **Partitioned**. (aka, CP or AP)
    - All in all, there is a lot of misunderstanding and confusion around CAP, and **it does not help us understand systems better, so CAP is best avoided**.
- **Linearizability and network delays**:
  - Although linearizability is a useful guarantee, surprisingly few systems are actually linearizable in practice.
  - Many distributed databases that choose not to provide linearizable guarantees: they do so primarily to increase performance, not so much for fault tolerance.
  - Linearizability is **slow**—and this is true all the time, not only during a network fault.

Ordering Guarantees

- Ordering has been a recurring theme in this book(e.g. Order of writes, Serializability, timestamps and clocks etc.)
- There are deep connections between **ordering**, **linearizability**, and **consensus**.
- **Ordering and Causality**:
  - There are several reasons why ordering keeps coming up, and one of the reasons is that it helps preserve **causality**.
    - E.g. causal dependency between the question and the answer;
  - **Causality imposes an ordering on events**: cause comes before effect;

        - a message is sent before that message is received;
        - the question comes before the answer.
    - If a system obeys the ordering imposed by causality, we say that it is **causally consistent**.
        - E.g. snapshot isolation provides causal consistency
    - **The causal order is NOT a total order**:
        - A **total order** allows any two elements to be compared, so if you have two elements, you can always say which one is greater and which one is smaller.
        - However, mathematical sets are not totally ordered: is {a, b} greater than {b, c}?
            - We say they are **incomparable**, and therefore mathematical sets are **partially ordered**.
        - **Linearizability**: In a linearizable system, we have a total order of operations.
        - **Causality**: two events are ordered if they are causally related (one happened before the other), but they are incomparable if they are concurrent. Hence **causality defines a partial order**.
    - There are **no concurrent operations in a linearizable datastore**: there must be a single timeline along which all operations are **totally ordered**.
    - **Distributed VCS** such as Git, their version histories are very much like the graph of causal dependencies.
        - Often one commit happens after another, in a straight line, but sometimes you get branches (when several people concurrently work on a project), and merges are created when those concurrently created commits are combined.
    - **Linearizability is stronger than causal consistency**:
        - What is the relationship between the causal order and linearizability?
            - **linearizability implies causality**: any system that is linearizable will preserve causality correctly.
        - **Causal Consistency** is the **Strongest** possible consistency model that does **NOT slow down** due to network delays, and remains available in the face of network failures.
        - In many cases, systems that appear to require linearizability in fact only really require causal consistency, which can be implemented more efficiently.
    - **Capturing causal dependencies**:
        - In order to maintain causality, you need to know which operation happened before which other operation.
        - In order to determine causal dependencies, we need some way of describing the "knowledge" of a node in the system.
        - The techniques for determining which operation happened before which other operation are similar to what we discussed in "Detecting Concurrent Writes"
        - In order to determine the causal ordering, the database needs to know which version of the data was read by the application. (e.g. Version Vectors, or conflict detection of SSI)
- **Sequence Number Ordering**:
    - Although causality is an important theoretical concept, actually keeping track of all causal dependencies can become impractical.
    - Better way: we can use sequence numbers or timestamps to order events.

- A **timestamp** need not come from a time-of-day clock but "**Logical Clock**".
- Such sequence numbers or timestamps are compact (only a few bytes in size), and they provide a total order.

- **Non-causal sequence number generators**:
  - If there is not a single leader (perhaps because you are using a multi-leader or leaderless database, or because the database is partitioned), it is less clear how to generate sequence numbers for operations.
    - Each node can generate its own independent set of sequence numbers. (e.g. one node odd, another node even; unique node identifier);
    - You can attach a timestamp from a time-of-day clock (physical clock) to each operation;
    - You can preallocate blocks of sequence numbers.
  - They all have a problem: **the sequence numbers they generate are not consistent with causality.**
- **Lamport timestamps:**
  - a simple method for **generating sequence numbers that is consistent with causality**.
  - **The Lamport timestamp is then simply a pair of (counter, node ID).**

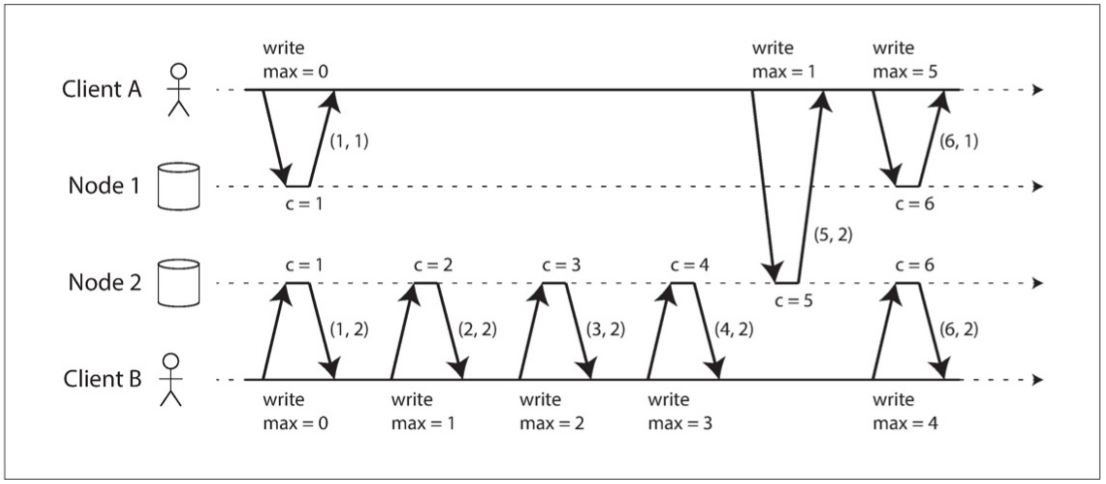


Figure 9-8. Lamport timestamps provide a total ordering consistent with causality.

- A Lamport timestamp bears no relationship to a physical time-of-day clock, but it provides total ordering:
  - if you have two timestamps, the one with a greater counter value is the greater timestamp;
  - if the counter values are the same, the one with the greater node ID is the greater timestamp.
- Key idea: **every node and every client keeps track of the maximum counter value it has seen so far, and includes that maximum on every request**.
  - When a node receives a request or response with a maximum counter value greater than its own counter value, it immediately increases its own counter to that maximum. (C: kind like "Read Repair"?)
- As long as the maximum counter value is carried along with every operation, this scheme ensures that the ordering from the Lamport timestamps is consistent with causality, because every causal dependency results in an increased timestamp.

- vs. **Version Vector**:
  - version vectors can distinguish whether two operations are concurrent or whether one is causally dependent on the other,
  - whereas Lamport timestamps always enforce a total ordering.
- **Timestamp ordering is not sufficient**:
  - The problem here is that the total order of operations only emerges after you have collected all of the operations.
  - To conclude: in order to implement something like a uniqueness constraint for usernames, it's not sufficient to have a total ordering of operations—you also need to know when that order is finalized.
    - This idea of knowing when your total order is finalized is captured in the topic of **total order broadcast**.
- **Total Order Broadcast**:
  - If your program runs only on a single CPU core, it is easy to define a total ordering of operations: it is simply the order in which they were executed by the CPU. (but it is (no-surprising) tricky in distributed-system)
  - In the distributed systems literature, this problem is known as**total order broadcast or atomic broadcast**.
  - Total order broadcast → protocol for exchanging messages between nodes. With two safety properties:
    - **Reliable delivery**: No messages are lost, if a message is delivered to one node, it is delivered to all nodes.
    - **Totally ordered delivery**: Messages are delivered to every node in the same order.
  - **Using total order broadcast**: Log-style (e.g. Consensus services such as ZooKeeper and etcd)
    - there is a strong connection between **total order broadcast and consensus**.
    - Use cases:
      - Can be used for DB replication (aka. State machine replication)
      - Can be used to implement serializable transactions.
      - Useful for implementing a lock service that provides fencing tokens. (e.g. zxid in ZooKeeper)
    - **Total order broadcast** is that the order is fixed at the time the messages are delivered.
    - Another way of looking at total order broadcast is that it is a way of creating a log (as in a replication log, transaction log, or write-ahead log)
      - delivering a message is like appending to the log.
  - **Implementing linearizable storage using total order broadcast**:
    - **Total order broadcast** is asynchronous: messages are guaranteed to be delivered reliably in a fixed order, but there is no guarantee about when a message will be delivered (so one recipient may lag behind the others).
      - By contrast, **linearizability** is a recency guarantee: a read is guaranteed to see the latest value written.
    - While this procedure ensures **linearizable writes**, it doesn't guarantee linearizable reads. (due to async. update)

        - To be precise, the procedure described here provides **sequential consistency**, sometimes also known as **timeline consistency**, a slightly weaker guarantee than linearizability.
        - There are ways to make reads linearizable tho.
    - **Implementing total order broadcast using linearizable storage**:
        - The easiest way is to assume you have a linearizable register that stores an integer and that has an atomic increment-and-get operation.
            - Alternatively, an atomic compare-and-set operation would also do the job.
        - It can be proved that a linearizable compare-and-set (or increment-and-get) register and total order broadcast are both equivalent to consensus.

Distributed Transactions and Consensus

- **Consensus** is one of the most important and fundamental problems in distributed computing: the goal is simply to get several nodes to agree on something.
- Why is Consensus important ?
    - Leader election:
    - Atomic commit: aka. atomic commit problem.
- **The Impossibility of Consensus**: FLP-result with very restrictive rules.
- **Atomic Commit and Two-Phase Commit(2PC)**:
    - Atomicity prevents failed transactions from littering the database with half-finished results and half-updated state.
    - **From single-node to distributed atomic commit**:
        - a node must only commit once it is certain that all other nodes in the transaction are also going to commit.
    - **Introduction to two-phase commit**:
        - Two-phase commit is an algorithm for **achieving atomic transaction** commit across multiple nodes.

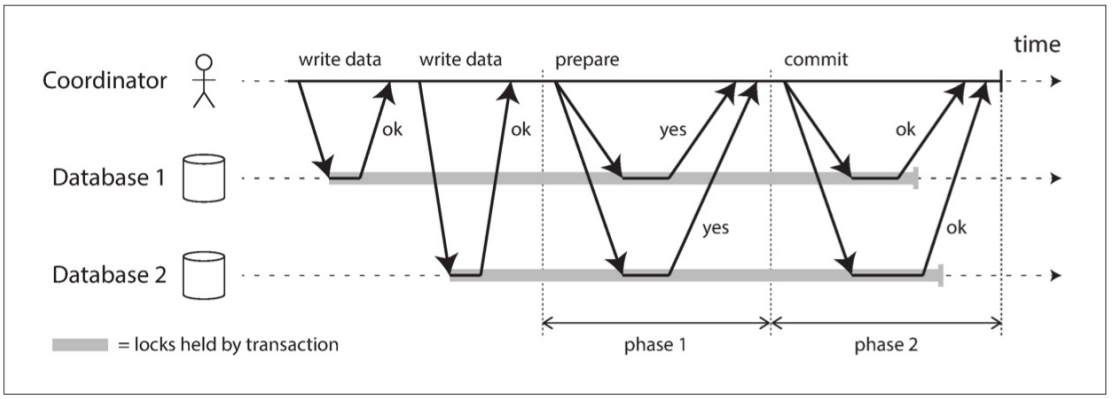


Figure 9-9. A successful execution of two-phase commit (2PC).

- 2PC uses a new component that does not normally appear in single-node transactions: a **coordinator** (aka. **transaction manager**).
- (This process is somewhat like the traditional marriage ceremony in Western cultures.)
- **A system of promises**:
    - There is no more going back: if the decision was to commit, that decision must be enforced, no matter how many retries it takes.

    - If a participant has crashed in the meantime, the transaction will be committed when it recovers—since the participant voted "yes," it cannot refuse to commit when it recovers.
- **Coordinator failure**:
    - If the coordinator fails before sending the prepare requests, a participant can safely abort the transaction.
    - But once the participant has received a prepared request and voted "yes," it can no longer abort unilaterally—**it must wait to hear back from the coordinator**.
    - If the coordinator crashes or the network fails at this point, the participant can do nothing but wait.
        - A participant's transaction in this state is called **in doubt or uncertain**.

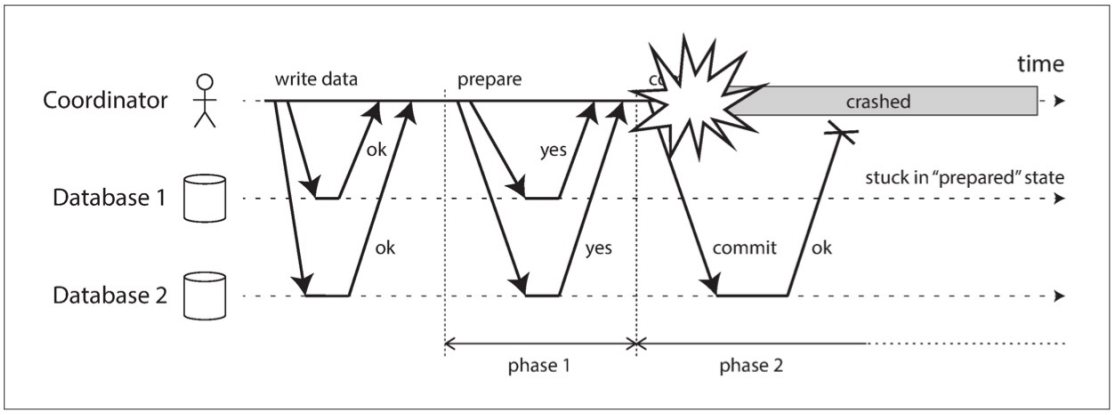


Figure 9-10. The coordinator crashes after participants vote "yes." Database 1 does not know whether to commit or abort.

- When the coordinator recovers, it determines the status of all in-doubt transactions by reading its transaction log.
- Thus, the commit point of 2PC comes down to a regular single-node atomic commit on the coordinator.
- **Three-phase commit**:
    - Two-phase commit is called a blocking atomic commit protocol due to the fact that 2PC can become stuck waiting for the coordinator to recover.
    - 3PC assumes a network with bounded delay and nodes with bounded response times.
    - Non-blocking atomic commit requires a perfect failure detector.
- **Distributed Transactions in Practice**:
    - Many cloud services choose not to implement distributed transactions due to the operational problems they engender.
    - Some implementations of distributed transactions carry a heavy performance penalty
        - E.g. distributed transactions in MySQL are reported to be over 10 times slower than single-node transactions.
    - **Two quite different types of distributed transactions**:
        - **Database-internal distributed transactions**:
            - Some distributed databases (i.e., databases that use replication and partitioning in their standard configuration) support internal transactions

among the nodes of that database.
    - **Heterogeneous distributed transactions**:
        - In a heterogeneous transaction, the participants are two or more different technologies: e.g. two databases from different vendors, or even non-database systems such as message brokers.
- **Exactly-once message processing**:
    - Heterogeneous distributed transactions allow diverse systems to be integrated in powerful ways.
        - E.g. Message Queue work with DB.
    - by atomically committing the message and the side effects of its processing, we can ensure that the message is effectively processed exactly once, even if it required a few retries before it succeeded.
- **XA transactions**: X/Open XA (short for eXtended Architecture)
    - a standard for implementing **two-phase commit across heterogeneous technologies**.
        - XA is not a network protocol—it is merely a C API for interfacing with a transaction coordinator. (e.g. JTA → JDBC, JMS)
    - XA is supported by many traditional relational databases (including PostgreSQL, MySQL, DB2, SQL Server, and Oracle) and message brokers (including ActiveMQ, HornetQ, MSMQ, and IBM MQ).
- **Holding locks while in doubt**:
    - database transactions usually take a row-level exclusive lock on any rows they modify, to prevent dirty writes.
- **Recovering from coordinator failure**:
    - In theory, if the coordinator crashes and is restarted, it should cleanly recover its state from the log and resolve any in-doubt transactions.
    - However, in practice, orphaned in-doubt transactions do occur — that is, transactions for which the coordinator cannot decide the outcome for whatever reason.
        - The only way out is for an administrator to manually decide whether to commit or roll back the transactions. (potentially requires a lot of manual effort)
    - Many XA implementations have an emergency escape hatch called **heuristic decisions**. (a euphemism for probably breaking atomicity)
- **Limitations of distributed transactions**:
    - **Transaction coordinator** is itself a kind of database (in which transaction outcomes are stored), and so it needs to be approached with the same care as any other important database. (It like Single Point of Failure)
    - **Distributed transactions** thus have a **tendency of amplifying failures**, which runs counter to our goal of building fault-tolerant systems.
- **Fault-Tolerant Consensus**:
    - **Consensus problem**: one or more nodes may propose values, and the consensus algorithm decides on one of those values.
    - **Consensus algorithm must satisfy the following properties**: (Termination is a **liveness** property, whereas the other three are **safety** properties)

    - **Uniform agreement**:No two nodes decide differently.
    - **Integrity**:No node decides twice.
    - **Validity**:If a node decides value v, then v was proposed by some node.
    - **Termination**:Every node that does not crash eventually decides some value. (formalizes the idea of fault tolerance)
- **Core idea**: everyone decides on the same outcome, and once you have decided, you cannot change your mind. (C: ALP 13, Disagree & Commit)
- The **termination property** is subject to the assumption that fewer than half of the nodes are crashed or unreachable.
- Most consensus algorithms assume that there are no Byzantine faults.
- **Consensus algorithms and total order broadcast**
    - The best-known fault-tolerant consensus algorithms are Viewstamped Replication (VSR), **Paxos**, Raft, and Zab;
    - They decide on a sequence of values, which makes them **total order broadcast** algorithms, as discussed previously.
- **Single-leader replication and consensus**:
- **Epoch numbering and quorums**:
    - All of the consensus protocols discussed so far internally use a leader in some form or another, but they don't guarantee that the leader is unique.
    - Instead, they can make a **weaker guarantee**:
        - The protocols define an **epoch number** (called the **ballot** number in Paxos, **view** number in Viewstamped Replication, and **term** number in Raft) and guarantee that within each epoch, the leader is unique.
    - How does a leader know that it hasn't been ousted by another node?
        - It must collect votes from a quorum of nodes.
        - The quorum typically, but not always, consists of a majority of nodes.
    - We have two rounds of voting: **once** to choose a leader, and a **second** time to vote on a leader's proposal.
        - **Key insight is** that the quorums for those two votes must overlap: if a vote on a proposal succeeds, at least one of the nodes that voted for it must have also participated in the most recent leader election.
    - vs. **2PC** which the coordinator is not elected.
        - Moreover, consensus algorithms define a recovery process by which nodes can get into a consistent state after a new leader is elected, ensuring that the safety properties are always met.
- **Limitations of consensus**:
    - Consensus algorithms are a huge breakthrough for distributed systems:
        - they bring concrete safety properties (agreement, integrity, and validity) to systems where everything else is uncertain,
        - they nevertheless remain fault-tolerant (able to make progress as long as a majority of nodes are working and reachable).
        - they can also implement linearizable atomic operations in a fault-tolerant way
    - But everything come at a cost:
    - Consensus systems **always require a strict majority to operate**.(1 out of 3, 2 out of 5 etc.)

    - Most consensus algorithms **assume** a fixed set of nodes that participate in voting, which means that you can't just add or remove nodes in the cluster.
    - Consensus systems generally rely on timeouts to detect failed nodes.(within highly variable network delays, this could be lots of false failed) result in frequent leader elections result in terrible performance.
    - Sometimes, consensus algorithms are particularly **sensitive to network problems**.
- **Membership and Coordination Services**:
    - Projects like **ZooKeeper** or **etcd** are often described as "distributed key-value stores" or "coordination and configuration services." (useful for distributed coordination)
    - ZooKeeper not for general-purpose databases, but indirectly via some other project.
        - E.g. HBase, Hadoop YARN, OpenStack Nova, and **Kafka** all rely on ZooKeeper running in the background.
    - ZooKeeper and etcd are designed to hold small amounts of data that can fit entirely in memory (although they still write to disk for durability).
        - That small amount of data is replicated across all the nodes using a **fault-tolerant total order broadcast** algorithm.
    - **total order broadcast is just what you need for database replication**:
        - if each message represents a write to the database, applying the same writes in the same order keeps replicas consistent with each other.
    - ZooKeeper not only **total order broadcast** (and hence consensus), but also:
        - **Linearizable atomic operations**: Using an atomic compare-and-set operation, you can implement a lock;
        - **Total ordering of operations**: fencing token -> monotonically increasing transaction ID (zxid) and version number (cversion);
        - **Failure detection**: the client and server periodically exchange heartbeats to check that the other node is still alive;
        - **Change notifications**: Not only can one client read locks and values that were created by another client, but it can also watch them for changes. (C: this featured used a lots for monitoring services)
    - **Allocating work to nodes**:
        - Useful for single-leader databases, but it's also useful for job schedulers and similar stateful systems.
        - Partitioned resources (database, message streams, file storage, distributed actor system, etc.) and need to decide which partition to assign to which node.
        - If done correctly, this approach allows the application to automatically recover from faults without human intervention.
        - ZooKeeper runs on a fixed number of nodes (usually three or five) and performs its majority votes among those nodes while supporting a potentially large number of clients.
        - If the application state needs to be replicated from one node to another, other tools (such as Apache BookKeeper) can be used.
    - **Service discovery**:
        - **ZooKeeper**, **etcd**, and **Consul** are also often used for service discovery—that is, to

find out which IP address you need to connect to in order to reach a particular service.
    - In cloud data center environments, where it is common for virtual machines to continually come and go, you often don't know the IP addresses of your services ahead of time.
        - Instead, you can configure your services such that when they start up they register their network endpoints in a service registry, where they can then be found by other services.
    - Some consensus systems support read-only caching replicas.
- **Membership services**:
    - **ZooKeeper** and **friends** can be seen as part of a long history of research into membership services.
    - **A membership service** determines which nodes are currently active and live members of a cluster.

Summary

- In this chapter we examined the topics of **consistency** and **consensus** from several different angles.
    - **Linearizability**(Consistency Model):
        - its goal is to make replicated data appear as though there were only a single copy, and to make all operations act on it atomically. (downside: being slow especially when there is large network delays)
    - **Causality**(weaker consistency model):
        - which imposes an ordering on events in a system (what happened before what, based on cause and effect).
        - **Causal consistency** does not have the coordination overhead of linearizability and is much less sensitive to network problems.
- We saw that achieving **consensus** means deciding something in such a way that all nodes agree on what was decided, and such that the decision is irrevocable.
    - Wide range of problems are actually reducible to consensus and are equivalent to each other:
        - **Linearizable compare-and-set registers**:The register needs to atomically decide whether to set its value, based on whether its current value equals the parameter given in the operation.
        - **Atomic transaction commit**:A database must decide whether to commit or abort a distributed transaction.
        - **Total order broadcast**:The messaging system must decide on the order in which to deliver messages.
        - **Locks and leases**:When several clients are racing to grab a lock or lease, the lock decides which one successfully acquired it.
        - **Membership/Coordination service**:Given a failure detector (e.g., timeouts), the system must decide which nodes are alive, and which should be considered dead because their sessions timed out.
        - **Uniqueness constraint**:When several transactions concurrently try to create

conflicting records with the same key, the constraint must decide which one to allow and which should fail with a constraint violation.

- How to handle leader-failure:
  - 1, Wait for the leader to recover, and accept that the system will be blocked in the meantime.
  - 2, Manually fail over by getting humans to choose a new leader node and reconfigure the system to use it.
  - 3, Use an algorithm to automatically choose a new leader.
- Tools like **ZooKeeper** play an important role in providing an "outsourced" **consensus**, **failure detection**, and **membership service** that applications can use.
- Not every system necessarily requires consensus: for example, **leaderless** and **multi-leader** replication systems typically do not use global consensus.

<Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems>

PREVIOUS POST

Designing Data Intense Application – Chapter 8: The Trouble with Distributed Systems

NEXT POST

Designing Data Intense Application – Chapter 10: Batch Processing

[SYSTEM-DESIGN]

Designing Data Intense Application – Chapter 10: Batch Processing

Posted by CHARLES *on* 2020-04-22

<Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems>

A system cannot be successful if it is too strongly influenced by a single person. Once the initial design is complete and fairly robust, the real test begins as people with many different viewpoints undertake their own experiments. —Donald Knuth

- Three different types of systems:
  - **Services (online systems)**: Response time is usually the primary measure of performance of a service and availability is often very important. (e.g. API)
  - **Batch processing systems (offline systems)**: primary performance measure of a batch job is usually throughput (the time it takes to crunch through an input dataset of a certain size). (e.g. MapReduce)
  - **Stream processing systems (near-real-time systems)**: As stream processing builds upon batch processing. (e.g. Kafka)
- As we shall see in this chapter,**batch processing** is an important building block in our quest to build reliable, scalable, and maintainable applications. (e.g. MapReduce → Hadoop, CouchDB, and MongoDB)

Batch Processing with Unix Tools

- **Simple Log Analysis**
  - Surprisingly many data analyses can be done in a few minutes using some combination of **awk, sed, grep, sort, uniq, and xargs**, and they perform surprisingly well.
  - **Chain of commands vs. Custom program**:

        - Instead of the chain of Unix commands, you could write a simple program to do the same thing. (e.g. Ruby, Python)
    - **Sorting vs. in-memory aggregation**:
        - The sort utility in GNU Coreutils (Linux) automatically handles larger-than-memory datasets by spilling to disk, and automatically parallelized sorting across multiple CPU cores.
            - This means that the simple chain of Unix commands we saw earlier easily scales to large datasets, without running out of memory. (Limited by desk speed)
- **The Unix Philosophy**
    - Doug McIlroy from 1964 "We should have some ways of connecting programs like [a] garden hose—screw in another segment when it becomes necessary to massage data in another way. This is the way of I/O also."
    - The idea of **connecting programs with pipes** became part of what is now known as the **Unix philosophy**—a set of design principles that became popular among the developers and users of Unix.
    - This approach—automation, rapid prototyping, incremental iteration, being friendly to experimentation, and breaking down large projects into manageable chunks—sounds remarkably like the **Agile** and **DevOps** movements of today.
    - **A uniform interface**:
        - In Unix, that interface is a file (or, more precisely, a file descriptor).
            - Another example of a uniform interface is URLs and HTTP, the foundations of the web.
        - A file is just an ordered sequence of bytes.
    - **Separation of logic and wiring**:
        - Another characteristic feature of Unix tools is their use of standard input (stdin) and standard output (stdout).
        - Separating the input/output wiring from the program logic makes it easier to **compose small tools into bigger systems**.
    - **Transparency and experimentation**:
        - Part of what makes Unix tools so successful is that they make it quite easy to see what is going on.
        - the biggest limitation of Unix tools is that they run only on a single machine—and that's where tools like **Hadoop** come in.

MapReduce and Distributed Filesystems

- **MapReduce** is a bit like Unix tools, but distributed across potentially thousands of machines.
    - Like Unix tools, it is a fairly blunt, brute-force, but surprisingly effective tool.
- A single **MapReduce** job is comparable to a single Unix process: it takes one or more inputs and produces one or more outputs.
- Instead of stdin or stdout, MapReduce jobs read and write files on a**distributed filesystem**. (e.d. **HDFS** for Hadoop, open source version of **GFS**)
    - **GlusterFS** and the Quantcast File System (**QFS**). Object storage services such as

Amazon S3, Azure Blob Storage, and OpenStack Swift are similar in many ways.

- **HDFS** is based on the **shared-nothing principle** (see the introduction to Part II), in contrast to the shared-disk approach of Network Attached Storage (NAS) and Storage Area Network (SAN) architectures.
  - shared-nothing approach requires no special hardware, only computers connected by a conventional data center network.
- **HDFS** consists of a **daemon process** running on each machine, exposing a network service that allows other nodes to access files stored on that machine (assuming that every general-purpose machine in a datacenter has some disks attached to it).
  - A central server called the **NameNode** keeps track of which file blocks are stored on which machine.
  - Thus, **HDFS** conceptually creates one big filesystem that can use the space on the disks of all machines running the daemon.
- In order to tolerate machine and disk failures, **file blocks are replicated on multiple machines**.
  - ensure coding scheme such as Reed–Solomon codes, kind like RAID.
- **HDFS has scaled well**: at the time of writing, the biggest HDFS deployments run on tens of thousands of machines, with combined storage capacity of hundreds of petabytes.
- **MapReduce Job Execution**:
  - **MapReduce is a programming framework** with which you can write code to process large datasets in a distributed filesystem like HDFS.
  - To create a MapReduce job, you need to implement two callback functions:
    - **Mapper**: The mapper is called once for every input record, and its job is to extract the key and value from the input record.
    - **Reducer**: The MapReduce framework takes the key-value pairs produced by the mappers, collects all the values belonging to the same key, and calls the reducer with an iterator over that collection of values.
  - Viewed like this;
    - The role of the **mapper** is to prepare the data by putting it into a form that is suitable for sorting.
    - The role of the **reducer** is to process the data that has been sorted.
  - **Distributed execution of MapReduce**:

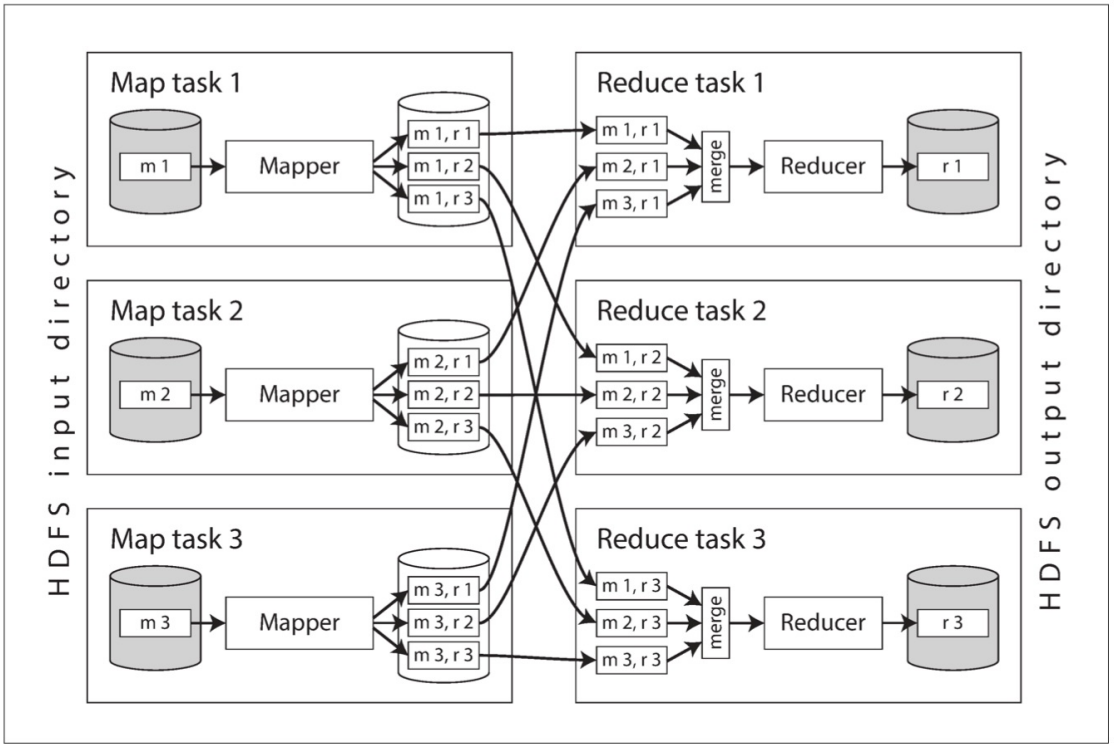


Figure 10-1. A MapReduce job with three mappers and three reducers.

- parallelization is based on partitioning
- **Putting the computation near the data**: it saves copying the input file over the network, reducing network load and increasing locality.
- The reduce side of the computation is also partitioned.
- The key-value pairs must be sorted, but the dataset is likely too large to be sorted with a conventional sorting algorithm on a single machine. Instead, the sorting is performed in stages.
- The process of partitioning by reducer, sorting, and copying data partitions from mappers to reducers is known as the **shuffle**.
- **MapReduce workflows**:
  - it is very common for MapReduce jobs to be **chained** together into workflows, such that the output of one job becomes the input to the next job.
    - this chaining is done **implicitly** by directory name:
    - First job must be configured to write its output to a designated directory in HDFS,
    - Second job must be configured to read that same directory name as its input.
  - Various **workflow schedulers** for Hadoop have been developed, including Oozie, Azkaban, Luigi, Airflow, and Pinball
    - Workflows consisting of 50 to 100 MapReduce jobs are common when building recommendation systems.
  - Various **higher-level tools** for Hadoop, such as Pig, Hive, Cascading, Crunch, and FlumeJava.
- **Reduce-Side Joins and Grouping**:
  - A **foreign key** in a relational model, a **document reference** in a document model, or **an edge** in a graph model.
  - MapReduce has no concept of indexes—at least not in the usual sense.
  - **Example: analysis of user activity events**:

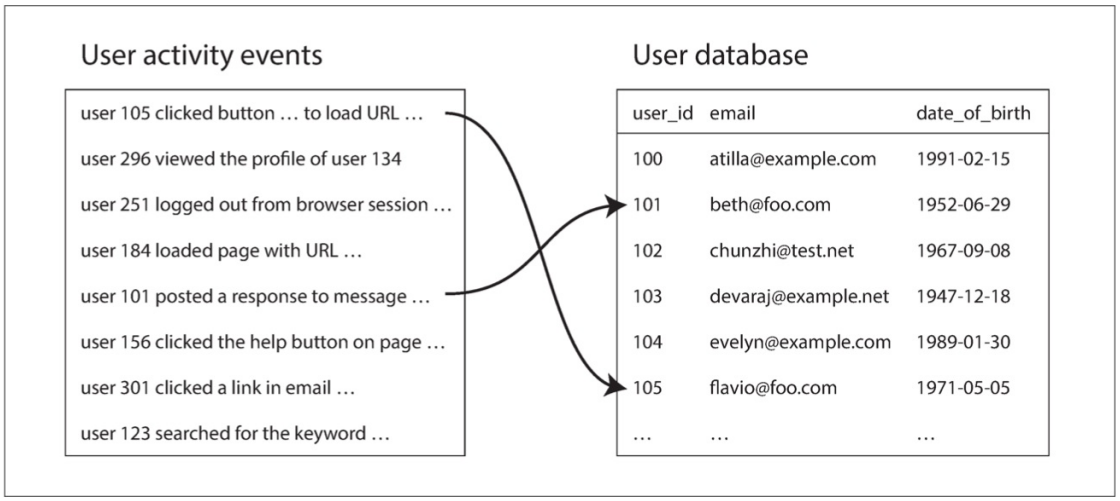


Figure 10-2. A join between a log of user activity events and a database of user profiles.

- **Star schema**: the log of events is the fact table, and the user database is one of the dimensions.
- In order to achieve good throughput in a batch process, the computation must be (as much as possible) local to one machine.
- **Sort-merge joins**:

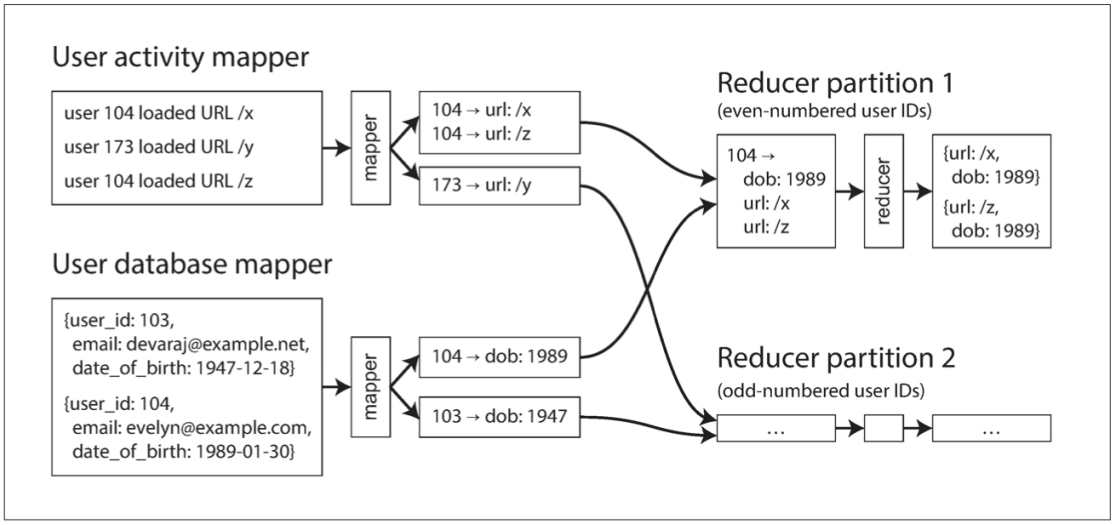


Figure 10-3. A reduce-side sort-merge join on user ID. If the input datasets are partitioned into multiple files, each could be processed with multiple mappers in parallel.

- The effect is that all the activity events and the user record with the same user ID become adjacent to each other in the reducer input. (along with secondary sort)
- **sort-merge join**: Since the reducer processes all of the records for a particular user ID in one go, it only needs to keep one user record in memory at any one time, and it never needs to make any requests over the network. (C: this explained why ETL tool Pentaho/Kettle need to always sort the value before "Merge Join Row")
- **Bringing related data together in the same place**:
  - One way of looking at this architecture is that mappers "send messages" to the reducers.
    - When a mapper emits a key-value pair, the **key acts like the destination address** to which the value should be delivered.
  - Using the **MapReduce programming model** has **separated** the physical network

communication aspects of the computation (getting the data to the right machine) **from** the application logic (processing the data once you have it).

- **GROUP BY**:
  - **The simplest way** is to set up the mappers so that the key-value pairs they produce use the desired grouping key.
  - **Another common use** for grouping is collating all the activity events for a particular user session, in order to find out the sequence of actions that the user took—a process called **sessionization**. (e.g. for A/B testing)
- **Handling skew**:
  - The pattern of "bringing all records with the same key to the same place" breaks down if there is a very large amount of data related to a single key.
    - e.g. celebrities in SNS, Such disproportionately active database records are known as linchpin objects or hot keys.
  - If a join input has hot keys, there are a few algorithms you can use to compensate. (e.g. skewed join method in Pig, sharded join method in Crunch)
  - Hive's skewed join optimization takes an alternative approach.
  - When grouping records by a hot key and aggregating them, you can perform the grouping in two stages.
- **Map-Side Joins**:
  - The reduce-side approach has the advantage that you do not need to make any assumptions about the input data.
  - if you can **make certain assumptions** about your input data, it is possible to **make joins faster** by using a so-called **map-side join**.
    - This approach uses a cut-down MapReduce job in which there are no reducers and no sorting.
  - **Broadcast hash joins**:
    - The simplest way of performing a map-side join applies in the case where a **large dataset is joined with a small dataset**.
      - the small dataset needs to be small enough that it can be loaded entirely into memory in each of the mappers
    - This simple but effective algorithm is called a **broadcast hash join**:
      - The word **broadcast** reflects the fact that each mapper for a partition of the large input reads the entirety of the small input (so the small input is effectively "**broadcast**" to all partitions of the large input), and the word **hash** reflects its use of a hash table.
      - E.g. Pig (under the name "replicated join"), Hive ("MapJoin"), Cascading, and Crunch, Data-warehouse engine Impala.
    - Instead of loading the small join input into an in-memory hash table, an alternative is to store the small join input in a read-only index on the local disk. (fit in OS' page cache, almost as fast as memory)
  - **Partitioned hash joins**: (e.g. bucketed map joins in Hive)
    - If the inputs to the map-side join are partitioned in the same way, then the hash join approach can be applied to each partition independently.
    - This approach only works if both of the join's inputs have the same number of partitions, with records assigned to partitions based on the same key and the

same hash function.
    - **Map-side merge joins**:
        - not only partitioned in the same way, but also sorted based on the same key.
    - **MapReduce workflows with map-side joins**:
        - When the output of a MapReduce join is consumed by downstream jobs, the choice of map-side or reduce-side join affects the structure of the output.
        - Knowing about the physical layout of datasets in the distributed filesystem becomes important when optimizing join strategies.
            - In the Hadoop ecosystem, this kind of **metadata** about the partitioning of datasets is often maintained in **HCatalog** and the **Hive metastore**.
- **The Output of Batch Workflows**:
    - Where does batch processing fit in?
        - It is not transaction processing, nor is it analytics. It is **closer to analytics**, in that a batch process typically scans over large portions of an input dataset.
    - The output of a batch process is often not a report, but some other kind of structure.
    - **Building search indexes**: (documents in, indexes out.)
        - Google's original use of MapReduce was to build indexes for its search engine, which was implemented as a workflow of 5 to 10 MapReduce jobs. (e.g. still used today by **Lucene/Solr**)
        - Recall **full-text search index**: it is a file (the term dictionary) in which you can efficiently look up a particular keyword and find the list of all the document IDs containing that keyword (the postings list).
    - **Key-value stores as batch process output**: (database files in, database out)
        - Another common use for batch processing is to build **machine learning systems** such as **classifiers** (e.g., spam filters, anomaly detection, image recognition) and **recommendation systems** (e.g., people you may know, products you may be interested in, or related searches)
        - Build a brand-new database inside the batch job and write it as files to the job's output directory in the distributed filesystem, just like the search indexes in the last section.
            - Various key-value stores support building database files in MapReduce jobs, including Voldemort, Terrapin, ElephantDB, and HBase bulk loading.
    - **Philosophy of batch process outputs**:
        - In the process, the **input is left unchanged**, any previous **output is completely replaced** with the new output, and there are no other side effects.
        - By treating **inputs as immutable** and avoiding side effects (such as writing to external databases), batch jobs not only achieve good performance but also become much easier to maintain.
        - On Hadoop, some of those low-value syntactic conversions are eliminated by using more structured file formats: e.g. Avro, Parquet
- **Comparing Hadoop to Distributed Databases**:
    - **Hadoop** is somewhat like a distributed version of Unix, where**HDFS** is the filesystem and MapReduce is a quirky implementation of a Unix process(which happens to always run the sort utility between the map phase and the reduce phase).
    - **MapReduce** and a **Distributed Filesystem** provides something much more like a

general-purpose operating system that can run arbitrary programs.

- **Diversity of storage**:
  - Databases require you to structure data according to a particular model (e.g., relational or documents),
    - whereas files in a distributed filesystem are just byte sequences, which can be written using any data model and encoding.
  - Collecting data in its raw form, and worrying about schema design later, allows the data collection to be speeded up (a concept sometimes known as a "**data lake**" or "**enterprise data hub**").
    - Aka. **sushi principle**: "**raw (data) is better**"
  - Indiscriminate data dumping shifts the burden of interpreting the data from producer to consumer's problem (schema-on-read approach).
  - There may not even be one ideal data model, but rather different views onto the data that are suitable for different purposes.
  - Data modeling still happens, but it is in a separate step, decoupled from the data collection.
    - This decoupling is possible because a distributed filesystem supports data encoded in any format.
- **Diversity of processing models**:
  - MapReduce gave engineers the ability to easily run their own code over large datasets.
  - Sometimes having two processing models, SQL and MapReduce, was not enough.
  - The system is flexible enough to support a diverse set of workloads within the same cluster.
  - Not having to move data around makes it a lot easier to derive value from the data, and a lot easier to experiment with new processing models.
- **Designing for frequent faults**:
  - When comparing MapReduce to MPP databases, two more differences in design approach stand out: **the handling of faults** and **the use of memory and disk**.
    - **MPP** databases prefer to keep as much data as possible in memory (e.g., using hash joins) to avoid the cost of reading from disk.
    - **MapReduce** is very eager to write data to disk, partly for fault tolerance, and partly on the assumption that the dataset will be too big to fit in memory anyway.
  - Overcommitting resources in turn allows better utilization of machines and greater efficiency compared to systems that segregate production and non-production tasks.
  - It's not because the hardware is particularly unreliable, it's because the freedom to arbitrarily terminate processes enables better resource utilization in a computing cluster.
    - Among open source cluster schedulers, preemption is less widely used. (e.g. YARN's CapacityScheduler)

Beyond MapReduce

- Depending on the volume of data, the structure of the data, and the type of processing being done with it, other tools may be more appropriate for expressing a computation.
- Implementing a complex processing job using the raw MapReduce APIs is actually quite hard and laborious—for instance, you would need to implement any join algorithms from scratch.
- In response to the difficulty of using MapReduce directly, various higher-level programming models (Pig, Hive, Cascading, Crunch) were created as abstractions on top of MapReduce.
- **Materialization of Intermediate State**:
  - Publishing data to a well-known location in the distributed file system allows loose coupling so that jobs don't need to know who is producing their input or consuming their output.
  - **Intermediate state**: a means of passing data from one job to the next. (not shared between job or team)
    - The process of writing out this intermediate state to files is called **materialization**. (C: recall "materialized views" from previously)
    - In contrast: **Pipes** do not fully materialize the intermediate state, but instead **stream** the output to the input incrementally, using only a small in-memory buffer.
  - MapReduce's approach of **fully materializing intermediate state** has downsides compared to Unix pipes: (C: which derived from its advantages)
    - A MapReduce job can only start when all tasks in the preceding jobs (that generate its inputs) have completed;
    - Mappers are often redundant: they just read back the same file that was just written by a reducer, and prepare it for the next stage of partitioning and sorting.
    - Storing intermediate state in a distributed file system means those files are replicated across several nodes, which is often overkill for such temporary data.
  - **Dataflow engines**:
    - New execution engine created to solve previou problems. E.g.**Spark**, Tez, and Flink.
      - They handle an entire workflow as one job, rather than breaking it up into independent subjobs.
    - Since they explicitly model the flow of data through several processing stages, these systems are known as **dataflow engines**.
    - Offers several advantages compared to the MapReduce model:
      - Expensive work such as sorting need only be performed in places where it is actually required.
      - There are no unnecessary map tasks
      - Can make locality optimizations.
      - It is usually sufficient for intermediate state between operators to be kept in memory or written to local disk.
      - Operators can start executing as soon as their input is ready;
      - Existing Java Virtual Machine (JVM) processes can be reused to run new operators, reducing startup overheads compared to MapReduce (which launches a new JVM for each task).

- **Fault tolerance**:
  - An advantage of fully materializing intermediate state to a distributed file system is that it is durable, which makes fault tolerance fairly easy.
    - if a task fails, it can just be restarted on another machine and read the same input again from the filesystem.
  - **Spark, Flink, and Tez** avoid writing intermediate state to HDFS, so they take a different approach to tolerating faults:
    - if a machine fails and the intermediate state on that machine is lost, it is **recomputed** from other data that is still available.
  - To enable this **recomputation**, the framework must keep track of how a given piece of data was computed—which input partitions it used, and which operators were applied to it.
    - When recomputing data, it is important to know whether the computation is **deterministic**.
      - The solution in the case of non-deterministic operators is normally to kill the downstream operators as well, and run them again on the new data.
    - In order to avoid such cascading faults, **it is better to make operators deterministic**.
  - Recovering from faults by recomputing data is not always the right answer:
    - if the intermediate data is much smaller than the source data, or if the computation is very CPU-intensive, it is probably cheaper to materialize the intermediate data to files than to recompute it.
- **Discussion of materialization**:
  - **Flink** especially is built around the idea of pipelined execution: that is, incrementally passing the output of an operator to other operators, and not waiting for the input to be complete before starting to process it.

- **Graphs and Iterative Processing**:
  - In graph processing, the data itself has the form of a graph.
  - This need often arises in machine learning applications such as recommendation engines, or in ranking systems. (e.g. PageRank)
  - Iterative style: (works, but very inefficient with MapReduce)
    - 1. An external scheduler runs a batch process to calculate one step of the algorithm.
    - 2. When the batch process completes, the scheduler checks whether it has finished.
    - 3. If it has not yet finished, the scheduler goes back to step 1 and runs another round of the batch process.
  - **The Pregel processing model**:
    - As an optimization for batch processing graphs, the **bulk synchronous parallel (BSP) model** of computation has become popular. Aka. "**Pregel model**".
      - (implemented by Apache Giraph, Spark's GraphX API, and Flink's Gelly API.)
    - **Idea behind Pregel**: one vertex can "send a message" to another vertex, and typically those messages are sent along the edges in a graph.
    - In each iteration, a function is called for each vertex, passing it all the messages that were sent to it—much like a call to the reducer. (It's a bit similar to the actor

model)
    - **Fault tolerance**:
        - Pregel implementations **guarantee that messages are processed exactly once** at their destination vertex in the following iteration.
        - This fault tolerance is achieved by periodically **check-pointing** the state of all vertices at the end of an iteration. (i.e., writing their full state to durable storage.)
    - **Parallel execution**:
        - A vertex does not need to know on which physical machine it is executing; when it sends messages to other vertices, it simply sends them to a vertex ID.
- **High-Level APIs and Languages**:
    - **Spark** and **Flink** also include their own high-level dataflow APIs, often taking inspiration from **FlumeJava**.
    - These dataflow APIs generally use relational-style building blocks to express a computation:
        - joining datasets on the value of some field;
        - grouping tuples by key;
        - filtering by some condition;
        - and aggregating tuples by counting, summing, or other functions.
    - **The move toward declarative query languages**:
        - The choice of join algorithm can make a big difference to the performance of a batch job;
        - This is possible if joins are specified in a declarative way: the application simply states which joins are required, and the query optimizer decides how they can best be executed.
        - Hive, Spark DataFrames, and Impala also use **vectorized execution**: iterating over data in a tight inner loop that is friendly to CPU caches, and avoiding function calls.
        - batch processing frameworks begin to look more like **MPP databases** (and can achieve comparable performance) they retain their flexibility advantage.
    - **Specialization for different domains**:
        - Another domain of increasing importance is **statistical** and **numerical** algorithms, which are needed for machine learning applications such as classification and recommendation systems.

Summary

- In this chapter we explored the topic of batch processing.
- In the Unix world, the uniform interface that allows one program to be composed with another in **files** and **pipes**;
    - In **MapReduce**, that interface is a distributed file system.
- Dataflow engines add their own pipe-like data transport mechanisms to avoid materializing intermediate state to the distributed file system, but the initial input and final output of a job is still usually HDFS.
- The two main problems that distributed batch processing frameworks need to solve are:
    - **Partitioning**: In MapReduce, mappers are partitioned according to input file blocks.

    - **Fault tolerance**: MapReduce frequently writes to disk, which makes it easy to recover from an individual failed task.
- **Join algorithms for MapReduce**:
    - **Sort-merge joins**: Each of the inputs being joined goes through a mapper that extracts the join key.
    - **Broadcast hash joins**: One of the two join inputs is small, so it is not partitioned and it can be entirely loaded into a hash table.
    - **Partitioned hash joins**: If the two join inputs are partitioned in the same way (using the same key, same hash function, and same number of partitions), then the hash table approach can be used independently for each partition.
- Distributed batch processing engines have a deliberately restricted programming model: **callback functions** (such as mappers and reducers) are assumed to be**stateless** and to have **no externally visible side effects** besides their designated output.
- Does **not need to worry about implementing fault-tolerance mechanisms**: the framework can guarantee that the final output of a job is the same as if no faults had occurred, even though in reality various tasks perhaps had to be retried.
- The output is derived from the input. And the input data is**bounded**: it has a known, fixed size (for example, it consists of a set of log files at some point in time, or a snapshot of a database's contents).

<Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems>

PREVIOUS POST

Designing Data Intense Application – Chapter 9: Consistency and Consensus

NEXT POST

Designing Data Intense Application – Chapter 11: Stream Processing

[SYSTEM-DESIGN]

Designing Data Intense Application – Chapter 11: Stream Processing

Posted by CHARLES *on* 2020-05-01

<Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems>

A complex system that works is invariably found to have evolved from a simple system that works. The inverse proposition also appears to be true: A complex system designed from scratch never works and cannot be made to work.

—John Gall, Systemantics (1975)

- Batch process is under the assumption all the data is **Bounded**, which means we know the finite size of the data we are dealing with, so it is known when the job is finished.
- In reality, a lot of the data is unbounded. (because data keeps generated every second). This issue will force the batch process to divide data into "chunks".
  - Which means the result/derived data is **delayed** based on the interval you are chosen. (this is too slow for impatient users)
- Continuously data processing without interruptions(break into chunks) is the idea behind **streaming processing**.
  - Think of "Stream" as a never stop flow of water/river that keep feeding data in;
  - E.g. (stdin/stdout, file inputstream, TCP stream etc.)
  - "Event Stream" as a data management mechanism

Transmitting Event Streams

- Batch processing: Input are **Files** vs. Streaming Processing: Input are **Events**;
- What is **Event**: a small, self-contained, immutable object containing the details of

something that happened at some point in time.

- Event generated by a producer(Publisher/Sender) and then processed by multiple consumers(subscribers/recipients).
  - Related "events" are grouped together by **Topic/Stream**.
- Batch vs. Stream processing is kind like the difference between Pull and Push.
  - Batch: is the consumer keep Pull event from DB.
  - Stream: is the DB keep push Event to Consumer.
- Traditional DB/(RDMS) is not designed for "Stream/Event" processing
- **Message Systems**:
  - A producer sends a message containing the event, which is then pushed to consumers.
  - MQ vs. Unix Pipe or TCP
    - MQ allows many-to-many relationships (Producer vs. Consumer)
    - As Unix Pipe & TCP is usually one-to-one
  - What happens if the producers send messages faster than the consumers can process them? Three options
    - Drop message; Queue ; backpressure (flow control)
    - What if Queue is full ?
  - What happens if nodes crash or temporarily go offline—are any messages lost?
    - Still a trade off between C & A (Consistency and Availability)
  - **Direct messaging from producers to consumers**: (prone to loss data)
    - **UDP multicast**: used in financial industry for streams such as stock market feeds;
    - **Brokerless message library**: ZeroMQ.
    - **StatsD & Brubeck**: UDP messaging.
  - **Message brokers**: (aka. Message Queue, e.g. ActiveMQ, RabbitMQ)
    - A special type of DB that optimized for Message Streams;
    - Producer → Broker → Consumer
    - Better fault tolerance; Message could be persist to disk;
    - It all asynchronously;
  - **Message brokers compared to databases**: (JMS, AMQP)
    - In MQ Some even support 2PC (two-phase commit);
    - In MQ, data is deleted right-after message is been consumed;
    - In MQ, usually assume the Queue is short.
    - DB Secondary Indexes vs. MQ subset of topics
    - MQ has no support for queries, but notify clients when data change
    - E.g. RabbitMQ, ActiveMQ, HornetQ, Qpid, TIBCO Enterprise Message Service, IBM MQ, Azure Service Bus, and Google Cloud Pub/Sub.
  - **Multiple consumers**:
    - **Load balancing**: arbitrarily assigned to worker/consumer; Good for parallel processing expensive work-load.
    - **Fan-out**: Each message is sent to all consumers/worker; (topic subscription in JMS, exchange bindings in AMQP)

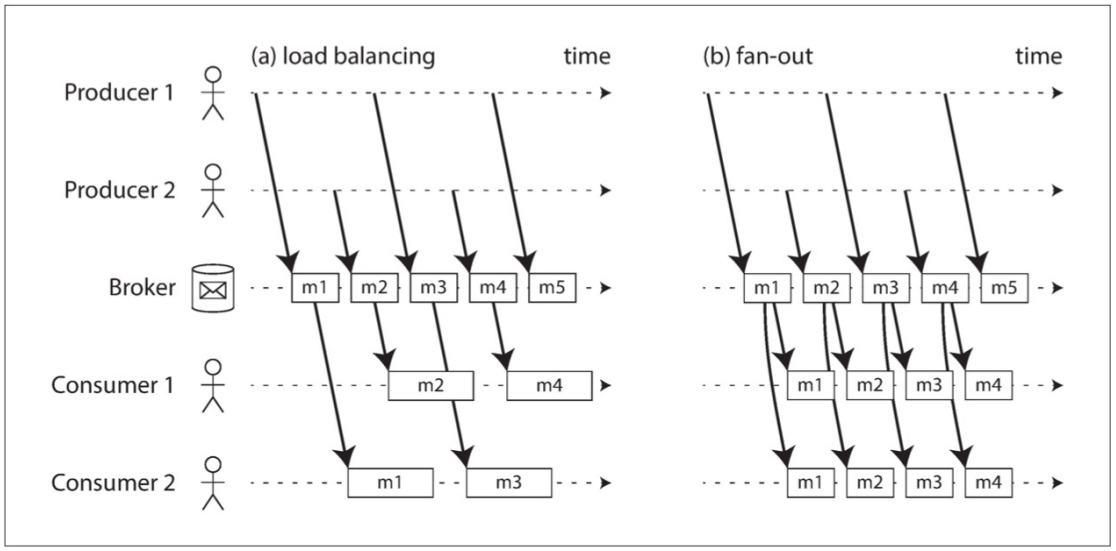


Figure 11-1. (a) Load balancing: sharing the work of consuming a topic among consumers; (b) fan-out: delivering each message to multiple consumers.

- Two patterns could be combined.
- **Acknowledgments and redelivery**:
  - **acknowledgments**: a confirmation from a client that it has finished processing a message so that the broker can remove it from the queue.
    - Note: due to network issues, the ack. Could be lost, then cause the ordering of the message change (C: In general, when you using Queue, you shouldn't have cared about the order at first place)
- **Partitioned Logs**:
  - **Transient messaging mindset**: transient operation that leaves no permanent trace. (Which is totally opposite than DB or FileSystem)
  - **MQ is NOT idempotent**: receiving a message is destructive if the acknowledgment causes it to be deleted from the broker;
  - Why can we not have a hybrid, combining the durable storage approach of databases with the low-latency notification facilities of messaging?
    - Yes, **log-based message brokers**.
  - **Using logs for message storage**:
    - Log: is simply an append-only sequence of records on disk.
    - **Log-based Message broker**: A producer sends a message by appending it to the end of the log, and a consumer receives messages by reading the log sequentially.
    - **This log can be partitioned**.
    - Each message is offset by a sequence number. (totally ordered)
      - Note: no ordering guarantee across partitions tho.

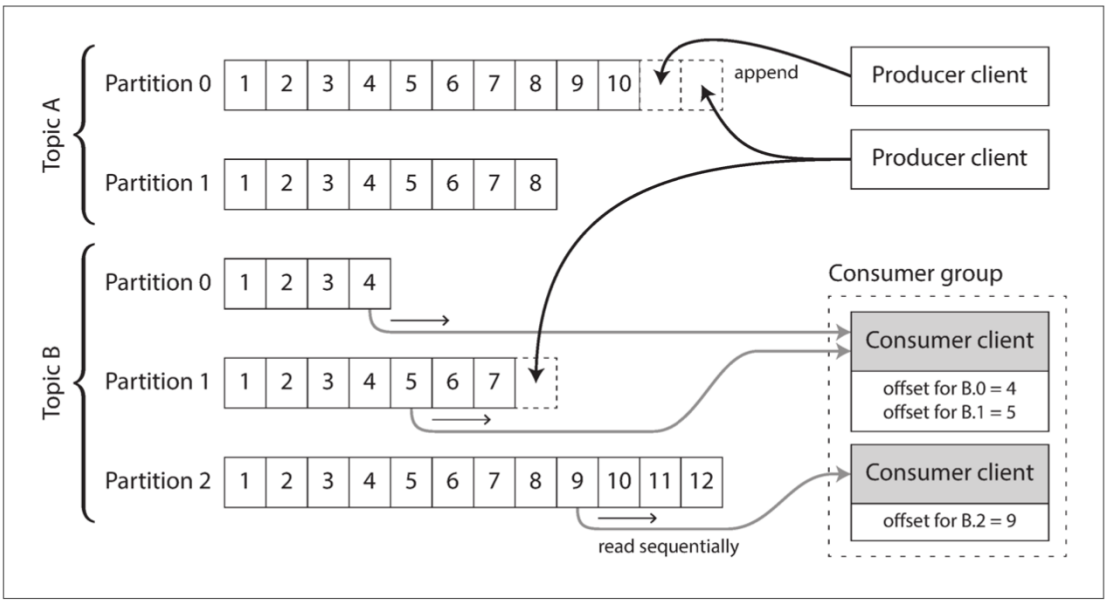


Figure 11-3. Producers send messages by appending them to a topic-partition file, and consumers read these files sequentially.

- E.g. Apache **Kafka**, Amazon Kinesis Streams, and Twitter's DistributedLog. (millions of MPS by partitioning)
- **Logs compared to traditional messaging***:
  - the broker can assign entire partitions to nodes in the consumer group, Then each client consumes all the messages in the partitions it has been assigned.
  - **JMS/AMQP style of message broker is preferable**: Message is expensive, parallel processing, order doesn't matter.
  - **Log-based approach**: high message throughput, where each message is fast to process and where message ordering is important.
- **Consumer offsets**:
  - Similar to "log sequence number" in single-leader DB replication;
  - The message broker behaves like a leader database, and the consumer like a follower.
- **Disk space usage**:
  - To prevent run out of storage, log is divided into segments, old segments are deleted or archived.
  - **"Log" is kind of like a bounded-size buffer**, if the consumer can't keep up, the old message will be discarded. (aka. Circular buffer, ring buffer)
    - E.g. 6T HDD with 150MB/s write speed can buffer up to 11 hrs of messages.
- **When consumers cannot keep up with producers**:
  - Three choices: **dropping**, **buffering** or **backpressure**(flow-control)
  - Consumers are independent from each other;
- **Replaying old messages**:
  - it is a read-only operation that does not change the log.
  - Offset is under the consumer's control. So, it has the freedom to go back to previous data/offset.
  - This made it easier for integration with other dataflows.

Databases and Streams

- DB and Stream are correlated deeper; (e.g. write is an event)
- The replication log is a stream of database write events, produced by the leader as it processes transactions.
- **Keeping Systems in Sync**:
  - Often need to combine several different technologies in order to satisfy their requirements. (Write/Read/Search/Analytics etc.)
  - It is essential to keep all the data in-sync. if an item is updated in the database, it also needs to be updated in the cache, search indexes, and data warehouse. (usually through ETL processes.)
    - Or "Dual writes", which prone to issue like race-conditiotion;

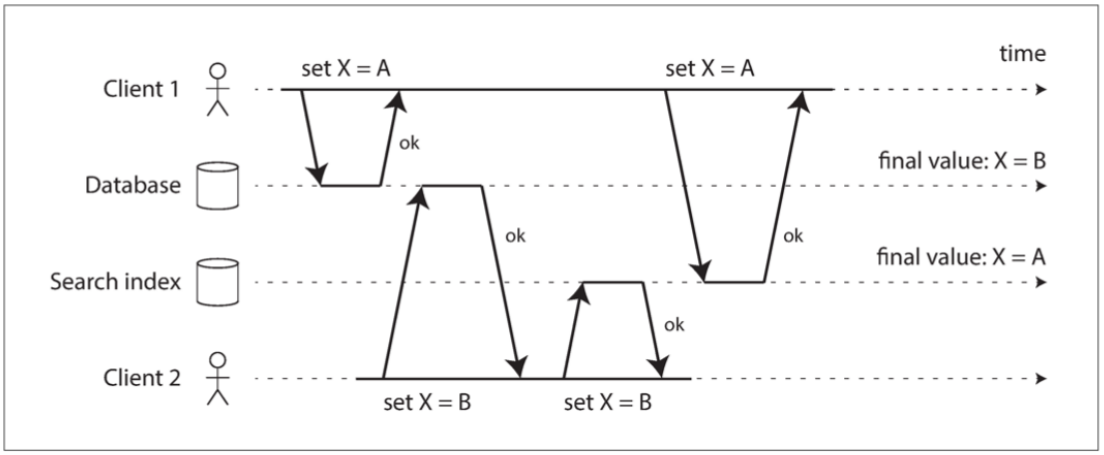


Figure 11-4. In the database, X is first set to A and then to B, while at the search index the writes arrive in the opposite order.

- The key is to determine if we can have only one "Source of Truth" (aka. Leader)
- **Change Data Capture**:
  - CDC(Change Data Capture): is a process that extracts DB changes and puts it into other systems.
    - E.g. in the form of "Stream"

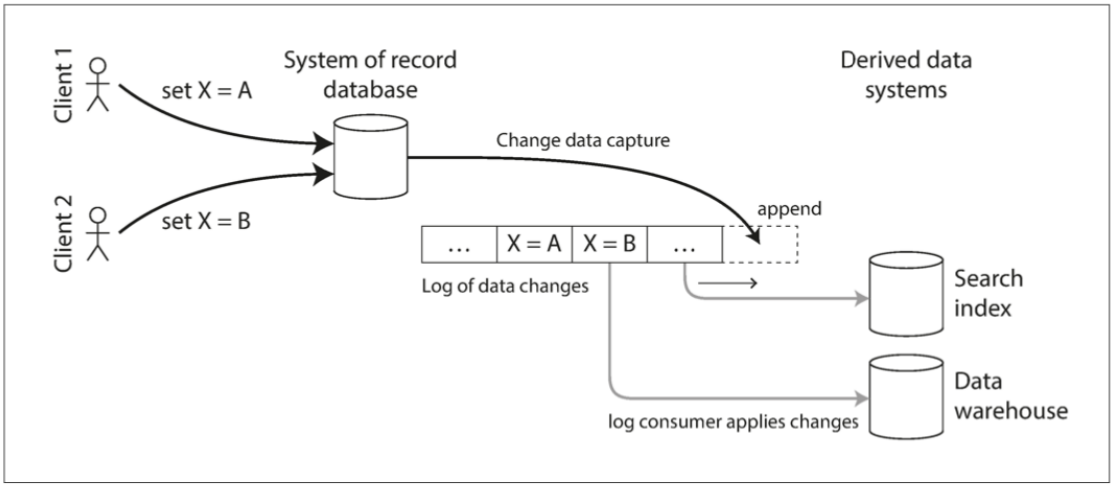
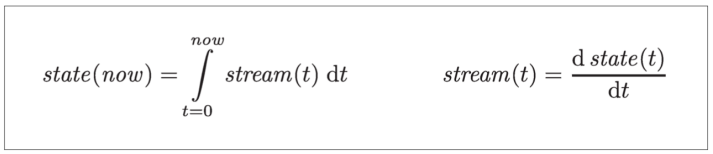


Figure 11-5. Taking data in the order it was written to one database, and applying the changes to other systems in the same order.

- **Implementing change data capture**:
  - We call any "log consumer" a "derived data system". The idea behind CDC is to

ensure all those "derived data systems" got the up-to-date changes.
    - Essentially, CDC makes one database the **leader** (the one from which the changes are captured), and turns the others into followers.
    - Potential Mechanism: DB Triggers(Con:performance overheads), Parsing replication log(Con:schema changes);
    - Usually done asynchronously → replication lag
- **Initial Snapshot**:
- **Log compaction**: (e.g. Apache Kafka)
    - This is used when need add a new "derived data system"
    - This allows the message broker to be used for durable storage, not just for transient messaging.
- **API support for change streams**: (e.g. RethinkDB, FIreBase, CouchDB, Meteor, VoltDB)
    - DBs engine started to support change streams;
    - A table will hold transactions but can't be queried.
- **Event Sourcing**:
    - **Event Sourcing**: a technique that was developed in the domain-driven design (DDD) community.
    - **CDC vs. ES**(Event Sourcing): different level of abstraction.
        - **CDC**: application isn't aware of CDC occurring, so it happens at a lower level.
        - **ES**: reflect things that happened at the application level.
    - **ES is a powerful technique for data modeling**, because it makes more sense to record a user's action as immutable events, rather than the effect of the actions on a mutable DB.
    - Event sourcing is similar to the chronicle data model. (or "fact table").
    - **Deriving current state from the event log**
        - Applications need to take the events and transform it to an application state that is suitable for the user to view. (deterministic)
    - **Commands and events**:
        - First it comes as "Command" and then after it successfully executed, it becomes "Event" which is durable and immutable.
            - when the event is generated, it becomes a fact.
        - A consumer of the event stream is not allowed to reject an event;
        - Any validation of a command needs to happen synchronously, before it becomes an event.
- **State, Streams, and Immutability**:
    - **Immutability** is also what makes event sourcing and change data capture powerful.
    - Whenever you have a state that changes, that state is the result of the events that mutated it over time.
    - **mutable state** and an **append-only log** of immutable events do not contradict each other: they are two sides of the same coin.
        - the **changelog**, represents the evolution of state over time.
    - In terms of mathematical:
        - **application state** is what you get when you **integrate** an event stream over time;
        - **a change stream** is what you get when you **differentiate** the state by time;



$$state(now) = \int_{t=0}^{now} stream(t)\ \mathrm{d}t \qquad stream(t) = \frac{\mathrm{d}\ state(t)}{\mathrm{d}t}$$

Figure 11-6. The relationship between the current application state and an event stream.

- **Quote from Pat Helland**:
  - *Transaction logs record all the changes made to the database. High-speed appends are the only way to change the log. From this perspective, the contents of the database hold a caching of the latest record values in the logs.* ***The truth is the log****. The database is a cache of a subset of the log. That cached subset happens to be the latest value of each record and index value from the log.*
- **Log compaction**: bridging the distinction between log and DB state. it retains only the latest version of each record, and discards overwritten versions.
- **Advantages of immutable events**: (e.g. Account ledger)
  - Particularly important in financial systems, it is also beneficial for many other systems.
  - Capture more information than just the current state.
  - E.g. Shopping cart history with append-only event could help analytic in the future;
- **Deriving several views from the same event log**
  - You can derive several different read-oriented representations from the same log of events. (C: This idea is similar to the talk about Apache Kafka,built application around the Kafka stream)
  - Having an explicit translation step from an event log to a database makes it easier to evolve your application over time. (C: enable old & new system running side by side)
  - **Command Query Responsibility Segregation (CQRS)**: you gain a lot of flexibility by separating the form in which data is written from the form it is read.
- **Concurrency control**:
  - The biggest downside of event sourcing and change data capture is asynchronous. → cause delay.
  - Potential Solutions:
    - “Reading your own writes”
    - “Implementing linearizable storage using total order broadcast”
- **Limitations of immutability**:
  - Truly deleting data could be difficult because data live in many places.
  - Deletion is more a matter of “making it harder to retrieve the data” than actually “making it impossible to retrieve the data.”

Processing Streams

- Where streams come from (user activity events, sensors, and writes to databases).
- How streams are transported (through direct messaging, via message brokers, and in event logs).
- Last question is **What can you do with Stream**? three major options

    - 1, write it to a database, cache, search index, or similar storage system so it can be used by other applications/clients/systems.
        - Kind like **maintaining materialized views**.
    - 2, push the events to users directly;
    - 3, process one or more input streams to produce one or more output streams. (pipelining); **processing streams to produce other, derived streams**.
- A block of Code that processes streams so called "Operator" or "Job". (kind like Unique processes or MapReduce job)
    - Since the Stream never ends(unbounded), so sorting doesn't make sense here, neither does sort-merge joins will be used.
    - Fault-tolerance mechanisms also need to be revised.
- **Use of Stream Processing**
    - **Monitoring System**: Fraud detection, Trading System, Manufacturing System, Military and Intelligence systems.
    - **Complex event processing(CEP)**: emerged from the 90s
        - CEP allows you to specify rules to search for certain patterns of events in a stream.
    - **Stream analytics**: (e.g. Apache Storm, Spark Streaming, Flink, Concord, Samza, and Kafka Streams, Google Cloud Dataflow and Azure Stream Analytics)
        - More oriented toward aggregations and statistical metrics over a large number of events;
        - Stream analytics systems sometimes use **probabilistic algorithms**, such as Bloom filters.
    - **Maintaining materialized views**:
        - Derived data systems can be treated as maintaining materialized views.
    - **Search on streams**:
        - The percolator feature of Elasticsearch is one option for implementing this kind of stream search.
    - **Message passing and RPC**:
- **Reasoning About Time**
    - Time "window"
    - Using the timestamps in the events allows the processing to be **deterministic**.
    - **Event time versus processing time**: (e.g. Star War movies)
        - Processing may be delayed.
        - Confusing event time and processing time leads to bad data.

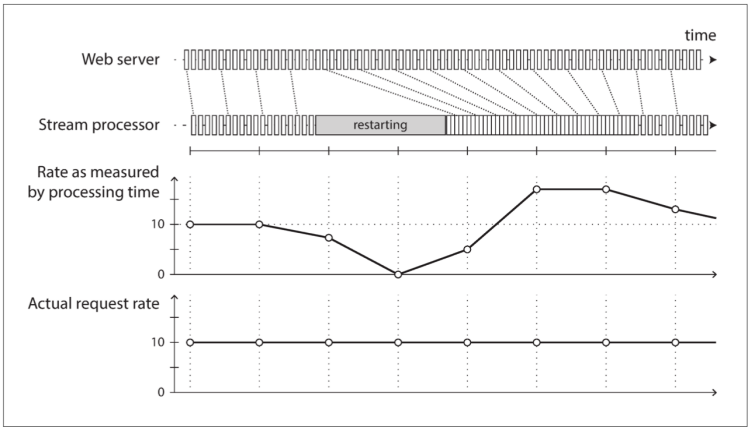


Figure 11-7. Windowing by processing time introduces artifacts due to variations in processing rate.

- **Knowing when you're ready**:
  - need to be able to handle such**straggler** events that arrive after the window has already been declared complete.
    - 1, Ignore the straggler events;
    - 2, Publish a correction;
- **Whose clock are you using, anyway**?
  - Need address Incorrect device clocks, log three timestamps:
    - The time at which the event occurred, according to the device clock
    - The time at which the event was sent to the server, according to the device clock
    - The time at which the event was received by the server, according to the server clock
- **Types of windows**:
  - Tumbling window: fixed length, and every event belongs to exactly one window.
  - Hopping window: fixed length, but allows windows to overlap in order to provide some smoothing.
  - Sliding window: contains all the events that occur within some interval of each other.
  - Session window: has no fixed duration. But, grouping together all events relative to the same user that occur closely together in time. (e.g. website analytics)

- **Stream Joins**
  - Similar to batch jobs; However, since new events can appear anytime on a stream makes joins on streams more challenging than in batch jobs.
  - three different types of joins: **stream-stream joins**, **stream-table joins**, and **table-table joins**.
  - **Stream-stream join (window join)**:
    - a stream processor needs to maintain state.
  - **Stream-table join (stream enrichment)**:
    - Enriching the activity events with information from the database.
    - Instead of performing remote SQL queries, we can cache up a copy of DB. (In Memory hashtable or local disk index)
      - Need CDC to ensure the stream data is up-to-date;
    - A stream-table join is actually very similar to a stream-stream join, but in this case we have "table changelog stream" involved.

    - **Table-table join (materialized view maintenance)**: (e.g. Tweets)
        - it maintains a materialized view for a query that joins two tables.
    - **Time-dependence of joins**:
        - **Common**: they all require the stream processor to maintain some state based on one join input, and query that state on messages from the other join input.
        - If the state changes over time, and you join with some state, what point in time do you use for the join ? (e.g. sales Tax calculation)
        - If the ordering of events across streams is undetermined, the join becomes nondeterministic;
        - **slowly changing dimension (SCD)**: addressed by using a unique identifier for a particular version of the joined record. (but this approach made log compaction impossible, because we need retain all version of the records)
- **Fault Tolerance**
    - You can't wait until a stream is finished to validate its output/result, since all the stream is unbounded and will never really finish/complete.
    - **Microbatching and checkpointing**:
        - **Microbatching**: break the stream into small blocks, and treat each block like a miniature batch process. (e.g. **Spark** Streaming) usually one second interval.
            - Smaller the batches size the greater overhead.
            - Larger batches size means longer delay of results.
            - implicitly provides a tumbling window equal to the batch size
        - **Checkpointing**: triggered by barriers in the message stream, similar to the boundaries between microbatches, but without forcing a particular window size. (e.g. Apache **Flink**)
        - Both approaches won't prevent external side effects after the results have been written into External Systems.
    - **Atomic commit revisited**:
        - Achieve "Exactly-Once" processing without transactions across heterogeneous technologies.
        - **Idempotence**:
            - Distributed transactions are one way of achieving that goal, but another way is to rely on **idempotence**.
            - if an operation is not naturally idempotent, it can often be made idempotent with a bit of extra metadata. (e.g. Kafka with some offset value)
        - **Rebuilding state after a failure**:
            - keep state local to the stream processor, and replicate it periodically.
            - sometimes the state can be rebuilt from the input streams.

Summary

- Discussed **event streams**, what purposes they serve, and how to process them.
    - Similar to "batch processing" but unbounded.
    - **message brokers** and **event logs** serve as the streaming equivalent of a filesystem.
- Two types of Message brokers:
    - **AMQP/JMS-style message broker**: exact order is not important

    - **Log-based message broker**: order is kept.
        - Similar to log-structured storage engines
- Where streams come from ?
    - user activity events, sensors providing periodic readings, and data feeds (e.g., market data in finance)
    - writes to a database as a stream: capture the changelog
        - Change Data Capture
        - Event Sourcing
- DB as streams is very useful for integrating different systems
    - E.g. search indexes, caches, and analytics systems.
- Stream joins and fault tolerance: By maintaining state as streams and replaying messages
    - searching for event patterns (complex event processing),
    - computing windowed aggregations (stream analytics),
    - keeping derived data systems up to date (materialized views).
- three types of joins:
    - Stream-stream joins
    - Stream-table joins
    - Table-table joins
- fault tolerance and exactly-once semantics
    - microbatching,
    - checkpointing,
    - transactions,
    - idempotent writes.

<Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems>

PREVIOUS POST

Designing Data Intense Application – Chapter 10: Batch Processing

NEXT POST

Designing Data Intense Application – Chapter 12: The Future of Data Systems

[SYSTEM-DESIGN]

Designing Data Intense Application – Chapter 12: The Future of Data Systems

Posted by CHARLES *on* 2020-05-15

<Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems>

If a thing be ordained to another as to its end, its last end cannot consist in the preservation of its being. Hence a captain does not intend as a last end, the preservation of the ship entrusted to him, since a ship is ordained to something else as its end, viz. to navigation. (Often quoted as: **If the highest aim of a captain was the preserve his ship, he would keep it in port forever**) — St. Thomas Aquinas, Summa Theologica (1265–1274)

- **Goal** for the book was: how to create applications and systems that are **reliable**, **scalable**, and **maintainable**.
- Our goal is to discover how to design applications that are better than the ones of today — **robust, correct, evolvable, and ultimately beneficial to humanity**.

Data Integration

- Common theme of the book: **for any given problem, there are several solutions**. (C: As it should be for anything in life)
- Challenges:
  - 1, mapping between tools and circumstances;
  - 2, in complex applications, data is often used in several different ways.
- **Combining Specialized Tools by Deriving Data**
  - The need for data integration often only becomes apparent if you zoom out and consider the dataflows across an entire organization.
  - **Reasoning about dataflows**:
    - Need be very clear about the INs/OUTs of the data;

    - **Derived data vs. distributed transactions**:
        - At an abstract level, they achieve a similar goal by different means.
    - **The limits of total ordering**:
        - In formal terms, deciding on a total order of events is known as total order broadcast, which is equivalent to consensus.
    - **Ordering events to capture causality**:
- **Batch and Stream Processing**:
    - The **goal of data integration** is to make sure that data ends up in the right form in all the right places.
    - There are also many detailed differences in the ways the processing engines are implemented, but these distinctions are beginning to blur.
    - **Maintaining derived state**:
        - Batch processing has a quite strong functional flavor.
        - Asynchrony is what makes systems based on event logs robust.
    - **Reprocessing data for application evolution**:
        - Reprocessing existing data provides a good mechanism for maintaining a system. (e.g. Schema Migrations on Railways, with three rails)
        - Derived **views** allow gradual evolution. (Because underlying data hasn't changed)
            - Every stage of the process is easily reversible if something goes wrong. (C: echo Jeff Bezos' type 1 or 2 decision)
    - **The lambda architecture**: combine stream and batch processing.
        - The lambda architecture proposes running two different systems in parallel:
            - a batch processing system such as Hadoop MapReduce,
            - a separate stream processing system such as Storm.
        - Potential problems:
            - Dual-maintenance;
            - Data need to be merged in order to produce results;
            - Reprocessing data is expensive on large datasets;
    - **Unifying batch and stream processing**:
        - allowing both batch computations (reprocessing historical data) and stream computations (processing events as they arrive) to be implemented in the same system.

Unbundling Databases

- At a most abstract level, **databases, Hadoop, and operating systems** all perform the same functions: they store some data, and they allow you to process and query that data.
    - at their core, both are "information management" systems
- Unix vs. Relationship Model
- **Composing Data Storage Technologies**:
    - Similar features that built for both "DBs" and "batch & stream processing";
    - **Creating an index**:
        - Whenever you run CREATE INDEX, the database essentially **reprocesses** the existing dataset and **derives** the index as a new view onto the existing data.

- **The meta-database of everything**:
  - the dataflow across an entire organization starts looking like one huge database.
  - Dataflow between subsystem is like keep indexes, materialized views up to date; (regardless is batch, stream or ETL)
  - Two cohesive system in the future:
    - **Federated databases: unifying reads**; (e.g. PostgreSQL's foreign data wrapper)
      - Application still able to access low-level data engine;
      - User can grab combined data from the federated interface;
    - **Unbundled databases: unifying writes**;
- **Making unbundling work**:
  - **Federation** and **unbundling** are two sides of the same coin: composing a reliable, scalable, and maintainable system out of diverse components.
  - Synchronizing writes is definitely a harder problem to solve.
  - Asynchronous event log with idempotent write**over** distributed transactions.
  - log-based integration is loose coupling between the various components.
    - 1, at a system level, asynchronous event streams make the system more robust and high-fault tolerance;
    - 2, at a human level, unbundling data systems allows subsystems to be more elastic/maintainable.
- **Unbundled vs. integrated systems**:
  - Specialized query engines will continue to be important for particular workloads.
  - It's about breadth, not depth.
  - If there is one software/system that can satisfy all your needs, then use it. Unless there isn't one, then you should consider unbundling/composition.
- **What's missing**?
  - We don't yet have the unbundled-database equivalent of the Unix shell. (e.g. declare mysql | elasticsearch, by analogy to Unix pipes)
  - E.g. differential dataflow
- **Designing Applications Around Dataflow**:
  - "database inside-out" approach, more like a "design pattern" rather than a "new architecture"
  - **Goal**: when a record in a database changes, we want any index for that record to be automatically updated, and any cached views or aggregations that depend on the record to be automatically refreshed. (And you don't need worry about the technical details)
  - **Application code as a derivation function**:
    - When one dataset is derived from another, it goes through some kind of transformation function. (e.g. secondary index, full-text search, ML system, cache)
    - Custom code within the database is the part where most of them struggle.
  - **Separation of application code and state**:
    - some parts of a system that specialize in durable data storage, and other parts that specialize in running application code. (e.g. Web application service, most of them are stateless)

- Keep stateless application logic separate from the state management(database)
  - not putting application logic in the database and not putting persistent state in the application
  - DB acts as a kind of **mutable shared variable** that can be accessed synchronously over the network.
- usually you can't subscribe to changes in a mutable variable, you can only read it periodically. (e.g. unlike Spreadsheet)
  - But you can implement your own notification —> aka. Observer pattern.
  - (C: this subscribe model also used by the Flutter state management package "Provider", it in a way of subscribe change and notify the UI to render)
  - Similar approach goes to DBs (subscribing to change only just beginning as a feature)

- **Dataflow: Interplay between state changes and application code**:
  - Application code responds to state changes in one place by triggering state changes in another place.
  - (C: This "unbundling" approach is very similar on the talk I watched that built systems around Apache Kafka)
  - Maintaining derived data is not the same as asynchronous job execution
    - 1, the order of state changes is often important.
    - 2, fault tolerance is the key
- **Stream processors and services**:
  - service-oriented architecture(SOA) over a single monolithic application is primarily organizational scalability through loose coupling.
  - Composing **stream operators** into dataflow systems has a lot of similar characteristics to the microservices approach.
    - But, it is one-directional, asynchronous message streams rather than synchronous request/response interactions.
  - "**The fastest and most reliable network request is no network request at all!**' (C: so make everything as local as possible)
- Subscribing to a stream of changes, rather than querying the current state when needed, brings us closer to a spreadsheet-like model of computation:

- **Observing Derived State**:

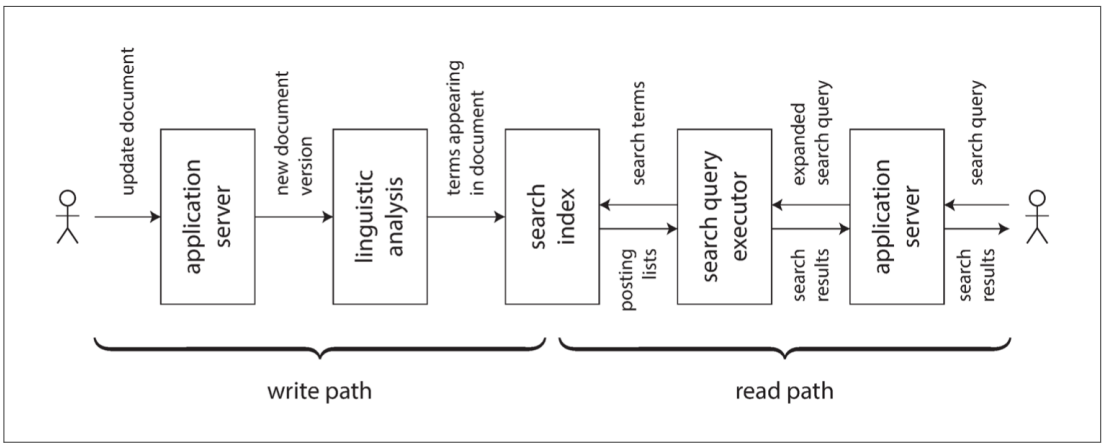


Figure 12-1. In a search index, writes (document updates) meet reads (queries).

- Taken together, the write path and the read path encompass the whole journey of the data, from the point where it is collected to the point where it is consumed (probably by another human).
  - Write Path: pre-computed → eagerly → eager evaluation
  - Read Path: lazy evaluation.
- **Materialized views and caching**:
  - The role of caches, indexes, and materialized views is simple: they shift the boundary between the read path and the write path.
- **Stateful, offline-capable clients**:
  - "single-page" JavaScript web apps have gained a lot of stateful capa- bilities. (Same as Mobile Apps)
    - client-side user interface interaction and
    - persistent local storage in the web browser
  - we can think of the on-device state as a cache of state on the server.
- **Pushing state changes to clients**:
  - Actively pushing states changes all the way to client devices means extending the write path all the way to the end users.
  - Each device is a small subscriber to a small stream of events.
- **End-to-end event streams**: (e.g. React, Flux, Redux)
  - Client-side state is managed by subscribing a stream of events (Event Sourcing)
  - Allow a server to push state-change events into this client-side event pipeline.
    - state changes could flow through an end-to-end write path.
  - We need to rethink the way we build our system: moving away from request/ response interaction and toward publish/subscribe dataflow.
  - **keep in mind the option of subscribing to changes, not just querying the current state**.
- **Reads are events too**:
  - Writing read events to durable storage thus enables better tracking of causal dependencies
- **Multi-partition data processing**:

Aiming for Correctness

- Stateless services are much more easier to correct than Stateful;
- **The End-to-End Argument for Databases**:
  - Data system with strong safety properties can't guarantee the application is free from data loss or corruption.
    - Application bugs occur, humans make mistakes.
  - "Remove the ability of faulty code to destroy good data" → the best way to fix mistakes is to avoid it in the first place.
  - **Exactly-once execution of an operation**:
    - "Idempotent" → but some scenarios require some effort and care to make the operation "idempotent".
  - **Duplicate suppression**:
  - **Operation identifiers**:

- In order to achieve operation idempotent, we need consider the end-to-end flow of the request. (e.g. UUID for each operation)

Example 12-2. Suppressing duplicate requests using a unique ID

```
ALTER TABLE requests ADD UNIQUE (request_id);

BEGIN TRANSACTION;

INSERT INTO requests
  (request_id, from_account, to_account, amount)
  VALUES('0286FDB8-D7E1-423F-B40B-792B3608036C', 4321, 1234, 11.00);

UPDATE accounts SET balance = balance + 11.00 WHERE account_id = 1234;
UPDATE accounts SET balance = balance - 11.00 WHERE account_id = 4321;

COMMIT;
```

- "Requests" table acting like an "event log".
- **The end-to-end argument**:
  - End-to-End solution: a transaction identifier that is passed all the way from the end-user client to the database.
  - Checking data integrity: Checksums.
  - Only end-to-end encryption and authentication can protect against all of these things.
  - low-level reliability features are not by themselves sufficient to ensure end-to-end correctness
- **Applying end-to-end thinking in data systems**:
  - Currently there still no good solution to wrap low-level reliability mechanisms around high-level faults.
  - Transaction is great but expensive, especially when it comes down to heterogeneous storages.
    - It made a wide range of possible issues (concurrent writes, constraint violations, crashes, network interruptions, disk failures) and collapsed them down to two possible outcomes: **commit** or **abort**.
- **Enforcing Constraints**:
  - E.g. Uniqueness
  - **Uniqueness constraints require consensus**:
    - want to be able to immediately reject any writes that would violate the constraint, synchronous coordination is unavoidable
  - **Uniqueness in log-based messaging**:
    - "Total order broadcast" → consensus
    - Linearizable storage using total order broadcast.
  - **Multi-partition request processing**:
    - correctness can be achieved with partitioned logs, and without an atomic commit. (By starting at 'Single-object' writes, and then the single-object will derive other messages/events to execute.)

- **Timeliness and Integrity**:
  - **Consistency** conflates two different requirements:
    - **Timeliness**: means ensuring that users observe the system in an up-to-date state.
    - **Integrity**: means absence of corruption. (ACID → ACD)
  - Slogan:
    - violations of **timeliness** are "eventual consistency,"
    - violations of **integrity** are "perpetual **in**consistency."
  - In general, **Integrity is much more important than timeliness**.
    - E.g. Credit Card transactions don't show within 24hrs matter far less if you got charged in the wrong amount.
  - **Correctness of dataflow systems**:
    - **ACID transactions** usually provide both **timeliness** (e.g., linearizability) and **integrity** (e.g., atomic commit) guarantees.
    - Integrity is central to streaming systems.
    - Exactly-once or effectively-once semantics is a mechanism for preserving integrity.
    - reliable stream processing systems can preserve integrity without requiring distributed transactions and an atomic commit protocol. By utilize the following mechanisms:
      - Event Sourcing;
      - Deterministic derivation functions;
      - End-to-end duplicate suppression and idempotence;
      - Immutable messages.
  - **Loosely interpreted constraints**:
    - many real applications can actually get away with much weaker notions of uniqueness. (e.g. user registration, back-order, over-booking etc.)
    - These applications do require integrity But they don't require timeliness on the enforcement of the constraint.
  - **Coordination-avoiding data systems**:
    - **coordination-avoiding data systems** have a lot of appeal: they can achieve better performance and fault tolerance than systems that need to perform synchronous coordination.
    - It boils down to Trade-offs
- **Trust, but Verify**:
  - System models take a binary approach toward faults. Happen vs. Never Happen
    - In reality, it is about probabilities: more likely vs. less likely.
  - **Maintaining integrity in the face of software bugs**:
    - E.g. MySQL & PostgreSQL
    - Many applications don't even correctly use the features that databases offer for preserving integrity, such as foreign key or uniqueness constraints.
    - Consistency in the sense of ACID assumes everything starts off in a consistent state. (also we assume there is bug-free application code)
  - **Don't just blindly trust what they promise**:
    - Checking the integrity of data is known as **auditing**.
    - E.g. large-scale storage systems such as HDFS and Amazon S3 do not fully trust

disks. (their have background processes keep checking and compare replicas)
        - Trust the disk most of the time, but not always.
    - Read → Check
- **A culture of verification**:
    - Under the "weak consistent" culture, "trust but verify" seems to become more important to implement.
- **Designing for auditability**:
    - Event-based systems can provide better auditability.
    - A deterministic and well-defined data-flow also makes it easier to debug and trace the execution of a system.
- **The end-to-end argument again**:
    - Checking the integrity of data systems is best done in an end-to-end fashion.
    - End-to-End ensured the widest coverage through the whole system(disks,networks,services and algorithms)
- **Tools for auditable data systems**: (C: Very interesting idea)
    - use **cryptographic tools** to prove the integrity of a system in a way that is robust to a wide range of hardware and software issues, and even potentially malicious actions. (e.g. Cryptocurrencies, blockchains, and distributed ledger technologies)
        - Essentially, they are distributed databases, with a data model and transaction mechanism, in which different replicas can be hosted by mutually untrusting organizations.
    - Cryptographic auditing and integrity checking often relies on **Merkle trees**.
        - **certificate transparency** is a security technology that relies on Merkle trees to check the validity of TLS/SSL certificates.

Doing the Right Thing

- Every system is built for a purpose; every action we take has both intended and unintended consequences.
- We have a responsibility to carefully consider those consequences and to consciously decide what kind of world we want to live in.
- Many datasets are about **people**: their behavior, their interests, their identity.
    - Treat users with humanity and respect.
- A technology is not good or bad in itself—what matters is how it is used and how it affects people.
- **Predictive Analytics**: (e.g. Weather, diseases, load, insurance etc.)
    - "Algorithmic prison" automated systems can systematically and arbitrarily exclude a person from participating in society without any proof of guilt, and with little chance of appeal. (C: People saying nowadays the world is more divided, maybe we can think it was caused by "algorithms"?)
    - **Bias and discrimination**:
        - "**machine learning is like money laundering for bias**"
        - Predictive analytics systems merely extrapolate from the past.
        - Data and models should be our tools, not our masters.
    - **Responsibility and accountability**:

        - Automated decision making opens the question of **responsibility and accountability**.
        - A credit score summarizes "How did you behave in the past?"
            - whereas **predictive analytics** usually work on the basis of "Who is similar to you, and how did people like you behave in the past?"
            - if a decision is incorrect due to erroneous data, recourse is almost impossible.
        - Much data is statistical in nature, which means that even if the**probability distribution** on the whole is correct, individual cases may well be wrong.
            - the output of a prediction system is probabilistic and may well be wrong in individual cases.
        - A blind belief in the supremacy of data for making decisions is not only delusional, it is positively dangerous.
    - **Feedback loops**:
        - Services only show you what you want to see → lead to "echo chambers" like stereotype, misinformation and polarization.
        - self-reinforcing feedback loops. → downward spiral.
        - "Systems thinking"
- **Privacy and Tracking**:
    - Data collection and tracking bring side consequences;
    - User tracking benefits users but also providing side effects.
    - "surveillance" users → the Consumer becomes the second citizen of the services/apps, but the **advertiser becomes the real user**.
    - **Surveillance**:
        - Replace "data" with "surveillance" e.g. "Designing Surveillance Intensive Applications"
    - **Consent and freedom of choice**:
        - Without understanding what happens to their data, users cannot give any meaningful consent.
        - It is not reciprocity between the services and users.
        - The relationship between the service and the user is very asymmetric and one-sided.
            - It is all set by the service provider, not the user.
    - **Privacy and use of data**:
        - Having privacy does not mean keeping everything secret; **it means having the freedom to choose which things to reveal to whom, what to make public, and what to keep secret**.
        - Privacy didn't erode but transfer to data collector;
    - **Data as assets and power**:
        - behavioral data is a byproduct of users interacting with a service → "data exhaust".
            - Or, we can think another way around, the service provider is using their service to lure users interaction/behavioral data.
        - Startups are valued by their user numbers, by "eyeballs" i.e., by their surveillance capabilities.
        - Need consider the future of the data on how it could be used or abused.

- **Remembering the Industrial Revolution**:
  - Data is the defining feature of the information age.
  - The good and bad that was brought by the Industrial Revolution → the same goes to the Information Revolution.
    - Data is the pollution problem of the information age, and protecting privacy is the environmental challenge. — Bruce Scheneier.
- **Legislation and self-regulation**:
  - Existing law from 1995 seems totally opposite with today's need of BigData.
  - Opportunities/Risk vs. over regulation
  - Purging vs. Immutability

Summary

- No one single tool can fit all;
  - Bundle is able to help
- Data integration problem can be solved by "batch processing" and "event streams"
  - Let the data flow
- Systems of record vs. Derived data
  - Keep it loosely coupled;
- Expressing dataflows as transformations from one dataset to another also helps evolve applications.
- Unbundling the components of a database, but composing loosely coupled components when building applications
- Derived state can be updated by observing changes
  - Bring the state to the end-user device.
- Strong integrity guarantees with
  - Asynchronous event processing
  - End-to-end operation identifiers to achieve idempotent or by checking constraints async.
  - This is much better than distributed transactions
- Audits for data integrity check
- Goal, built a world that treats people with humanity and respect.

<Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems>