Designing data-intensive applications

2017, Martin Kleppmann, first edition

Preface

- Terminology: **data-intensive systems** are those where data is the primary concern: e.g. the amounts of data, or complexity, or speed at which it is changing. It contrasts **compute-intensive systems**, where CPU cycles are the problem.
- The book wants to help engineers and architects to understand different technologies and their trade-offs.

1. Application characteristics

- Applications are usually built of standard blocks that have functionality such as: (P. 3)
    - Databases: store data so that it can be retrieved later in an efficient way, including search indices
    - Caches: remember the result of an expensive operation, speeding up reads
    - Stream processing / message queues: asynchronously handle messages, forwarding them to other systems
    - Batch processing: regularly do data crunching of large amounts of data
- Many of the systems on the market today are mixtures of functionality, e.g. Redis which is both a cache, database and a message queue.
- During development, many different questions come up, such as:
    - How to ensure correctness and completeness of stored data?
    - How to provide consistent and good performance, even under degraded conditions?
    - How to handle an increase in system load?
    - How to design a well-usable API?
- In data-intensive systems, the following application characteristics are desirable (P. 6):
    - Reliability: when the system works correctly (doing the right thing at the desired level of performance), even in case of software/hardware/human errors.
    - Scalability: reasonably react to growths in data volume, traffic volume or complexity
    - Maintainability: making sure that a system can be easily changed over time (adapting to changing environments and requirements, fixing bugs, adapting it to new platforms, keeping it operational), even when many different people (devs, ops) work on it.
- Details about **reliability**:
    - Fault = when a component of the system no longer behaves as specified (e.g. a bug). Failure = when the system as a whole stops providing the service to the user.
    - Systems that anticipate faults are called “fault-tolerant” or “resilient”. But as there

are so many different types of faults, you have to choose which specific fault types you want to “support”.
        - Hardware faults: the trend goes to distributing the system to many machines, and being able to tolerate the loss of entire machines.
        - Software faults: there is not much you can do, other than write tests carefully, and e.g. isolate parts of the system from one another, so that if one goes down (e.g. one process crashes), this won’t cause a domino effect.
        - Human faults: Humans are the main cause for outages (e.g. when they misconfigure the system accidentally). To reduce the risk, you should design the system to minimize the opportunities for error, or allow for quick and easy recovery from errors (e.g. via rollbacks, or gradual rollouts). Also, having detailed monitoring is important.
- Details about **scalability**:
    - You always make a system scalable according to a specific type of **load**. Load is described by **load parameters**, and they highly depend on your application. Examples are requests per second to a web server, or the ratio of reads to writes in a DB. Sometimes the average number matters, sometimes the peaks do.
    - You then need to establish how the resources (or changing them) relates to changes in the load, which is called **performance**. You can look at it in two ways (P. 13):
        - When you increase a load parameter, and keep system resources (CPU etc.) the same, how is the *performance* of the system affected?
        - If you increase a load parameter, how much do you need to increase the resources if you want to keep a stable level of *performance*?
    - Performance numbers might e.g. be **throughput** (records per second processed by a batch system), or the **response time** of a request (e.g. web server).
    - However, single numbers are meaningless. Usually you care about performance distributions / histograms (P. 14).
        - Often, the **mean** is not a good metric, percentiles are better.
    - Coping with load:
        - It is often the case that whenever you want to be able to support one order of magnitude (10x) of the current load tolerance, you need to completely (or partially) rewrite the architecture of your system.
        - In practice you often have a mixture of vertical and horizontal scaling approaches.
        - Elasticity (= fully *automated* up+down-scaling) is often not needed, and a manual scaling process can be sufficient, and they have fewer “surprises” (e.g. a high bill).
- Details about **maintainability**:
    - You can follow these 3 design principles for software systems to minimize maintenance pain:
        - Operability: make it easy for ops teams to keep the system running smoothly:

        - Provide good instrumentation and metrics
        - Don’t build in any dependency on concrete individual machines
        - Provide good documentation and an easy-to-understand operational model (if I do Y, X will happen)
        - Provide good default behavior, but also provide sensible configuration options.
        - Do self-healing where appropriate, but also give admins control to override it.
    - Simplicity: make it easy for new engineers to understand the system - remove as much complexity as possible.
    - Evolvability/modifiability/extensibility: make it easy to make changes to the system

2. Data models and query languages

Data models

- The most prominent models today are relational (SQL), and non-relational. The most common ones for the latter are document and graph models. Each model has assumptions about how it is being used. Some usage modes are easy, some are hard, some are impossible. (P. 28)
- NoSQL is not a particular technology. Driving forces behind NoSQL included:
    - Better scalability than relational DBs could offer, including very high write throughput scenarios
    - Need for specialized queries SQL did not support
    - Frustration that the relational schemas are so restrictive - many needed a more dynamic and expressive data model (P. 29)
- Difficulty of SQL/relational model: there is the object-relational mismatch: you need a translation between objects and the (multiple) tables the data is stored in (in a distributed way, when using a normalized schema). ORM frameworks hide most of this complexity, but cannot hide it completely (P. 30)
- Document DBs include MongoDB, RethinkDB, CouchDB, or Espresso.
- Arguments for Document data model (vs. relational model): P. 38
    - Better performance: data structures very closely reflect the objects, no JOINs are needed
    - Schema flexibility, which you might e.g. need when you pull data from other sources over which you have no control over, or if there are so many different kinds of entities that it does not make sense to design a separate table/schema for each one.
- Arguments against Document data models: (P. 38)
    - JOINs are not that well-supported (although DB implementations are catching up here, but the JOINs are still less performant than those of relational DB engines)
    - Relational models have better support for many-to-many and one-to-many

relations.
    - Document DBs are slow if each document you store in the DB is *huge* in size: the engine usually has to load the entire document into memory, even if you are just SELECTing a part of it. Also, when updating a document, the *entire* document has to be written to the DB again.
- Document DBs are *not* schemaless - there is always an implicit scha, but it is not enforced by the DB (when writing, or reading). Instead, the *application* implicitly imposes a schema when *reading*. In contrast, relational models enforce the schema on write (usually, there are exceptions such as SQLite).
- Over time, relational and document DBs are converging (P. 42).

Query languages

- The coarse classification of query languages can be “imperative vs. declarative”. SQL is *declarative*. OTOH, *imperative* would mean that you are writing code that iterates over data sets and (iteratively) manipulates the response-set/list.
- MapReduce:
    - Allows batch processing of large amounts of data, across many machines. No SQL DBs like MongoDB or CouchDB support MapReduce operations
    - The *map* (aka. collect) step transforms the raw input data in some form, e.g. extracting the month of a timestamp.
    - The *reduce* step (aka. *fold* or *inject*) performs some kind of aggregation on the data, e.g. computing the sum. (P. 46)
    - Map and reduce must be pure functions without side effects, so that the DB engine can run them anywhere, in any order (P. 48)
- Graph-like data models
    - Graphs have vertices/nodes and edges/relationships.
    - Examples include social graphs (people), web graph (web pages), or road/rail networks (vertices are junctions, edges are the roads/rails)
        - However, in general, graph structures are not limited to such kinds of *homo*geneous structures, for instance, a person could reference a website, etc.
    - Two common representation models are (P. 50)
        - **Property graph** model (used e.g. by Neo4j, Titan, InfiniteGraph)
            - Vertices have a unique identifier, a set of outgoing and incoming edges, and a collection of key-value pair properties.
            - Edges have a unique identifier, the “start”/”tail” vertex, the “end”/”head” vertex, a descriptive label, and (optional) a collection of key-value pair properties.
        - **Triple-store** model (used e.g. by AllegroGraph, Datomic, etc.)
            - Every information is stored in a simple three-part statement (*subject, predicate, object*), e.g. (jim, likes, beans).
            - The *subject* is like the vertex of a graph. The *object* can either be:
                - A value of a primitive data type (e.g. int). In this case, the

predicate and *object* together are like a K-V pair, being a property of the subject/vertex.
    - Another vertex in the graph. In this case, the *predicate* is an edge in the graph.
- Declarative query languages for graphs are e.g.
    - Cypher: used for property graphs, used in Neo4j
    - SPARQL: a query language for RDF triple-stores - it looks quite SQL-like (P. 59)
    - Datalog: much older than the other 2 query languages, from the 1980s, reminiscent of “Prolog”, where you define logical “rules” that build upon each other. Still used today as the foundation of Cascalog which is used to query large datasets in Hadoop. (P. 60)
- Theoretically, you can also put graph structures into *relational* databases, but it is not recommended to do so, because in relational DBs, you would like to know in advance which JOINs to make, but with graph structures, you want to have the flexibility to traverse edges once or more times (until you find a vertex you are looking for), i.e., the number of JOINs would not be fixed in advance. Modern SQL does have the “WITH RECURSIVE” to do this, but the queries look very bulky and are difficult to understand. (P. 53)
- Semantic web (P. 57): basic idea is that websites publish information as graphs in a machine-readable form. The Resource Description Framework (RDF) was chosen as the general notation format. The vision of semantic web was to become the “database of everything”, but it was not really implemented much in practice.
    - Turtle or JSON-LD are concrete formats to specify RDF-compatible graphs and their nodes and edges.
- In graph databases, vertices and edges are not ordered, but when querying, you can order the results (P. 60)
- Graph vs. document DBs (P. 63):
    - Document DBs are good for use cases where data comes as self-contained documents, and relationships are rare, or at most 1:1 or 1:n
    - Graph DBs: target use cases where anything is potentially related to anything

3. Storage and retrieval

This chapter explains how different databases work under the hood. You need this understanding to decide which kind of database to use, depending on the type of read/write workload and the types of queries your app needs to make.

Storage engines

- The two most common types of storage engines are **log-structured** vs. **page-oriented** engines. (P. 70)
- Log-structured:
    - A log is an *append-only* data file (P. 71)
    - In the real world, an always-growing log file would not be practical.

Consequently, the logs are broken down into segments, and there is regular background compaction happening for these segments (P. 73).
    - Append-only logs are very efficient for writing, but reading would be very slow, because you'd have to scan the entire log file to find the most recently updated value. To avoid this, you also maintain indices, of which there are several variants.
- Indices
    - No matter which index variant you choose, it will always slow down write operations. Therefore, indices are usually disabled by default, and you, the app developer, must explicitly define them, for the specific read queries that you want to speed up. (P. 71)
    - Hash indices are a common approach for key-value pair data. The index could e.g. be implemented as a hash map that maps from the key to the *storage address* of the actual data.
        - Limitations of hash tables:
            - They must fit in memory completely, so they are not suitable for huge amounts of data. Getting *disk*-based hash tables to work that perform well is very difficult, because of the random access I/O patterns, hash collisions, etc.
            - Range queries are not efficient: you cannot efficiently *scan* over a set of keys, such as foo000 - foo999. You would have to look up each key individually.
    - Log-Structured index (a.k.a. LSM, Log-Structured Merge-Tree) (P. 76-79)
        - Basic idea is to use an in-memory index structure, the "memtable", e.g. implemented as AVL-trees or Red-Black-trees, that make sure that the incoming writes are sorted/ordered automatically (because these trees do that automatically on insertion), and then regularly compact these memtables to disk, in the "SSTable" format (Sorted String table). The SSTable file format is a very simple serialization of the keys (and their value, e.g. the data pointer), s.t. all k-v pairs are written after another, the keys being are already sorted, and each key exists only once in a particular SSTable file.
        - Compacting multiple SSTable files can be done very efficiently, and does not require loading these files to memory: in a "merge-sort style", you simply read two SSTable files side-by-side, look at the first key in each file, and copy the lowest key (according to the sort order) to the output file, and repeat.
        - You can construct an (even partial) memtable for existing SSTable files quickly and easily.
        - Reading works like this: first check the memtable, then the most recent SSTable file, then the 2nd-most-recent SSTable file, etc.
        - To avoid that the writes are lost that are only stored in the pure-in-memory memtable (which is only "regularly" persisted), you can additionally write a recovery log file (in which the write keys are not sorted

according to the time, but simply by insertion order). After a DB crash, you can then use this log file to restore the memtable.

- B-trees (P. 79)
  - B-trees are much more popular than the Log-Structured Merge-Trees presented above. They are used in almost all relational DB engines, and many nonrelational ones.
  - While LSMs break the index down into segments, B-trees break it down into fixed-size blocks (a.k.a. pages). This is done to be more close to the hardware that also uses such pages/blocks. The pages contain the keys in sorted order, and can reference either other pages (that make a more fine-grained division of the index), or concrete values. See image on P. 80 for an example. The engine uses a predetermined "branching factor" to decide how many splits in the index key ranges there are (usually a branching factor is in the range of "several hundreds").
  - When inserting a new item, you first traverse down the tree to find the page/node where to insert the k-v pair. If that page/node has no space left, you need to split that node into two equally sized ones, and update the parent node/page (and should that page also be full already, you also have to update the parent's parent page, etc.).
- LSM-Trees vs. B-trees: as a general rule of thumb, LSM-trees are usually faster for writes, and B-trees faster for reads. The reading of LSM-trees is usually slower, because multiple SSTables need to be checked to find the value. However, in practice, the performance depends on your data and queries, so you should always benchmark anyway. When benchmarking, you should not only vary the load (e.g. items written/read per second), but also the duration, i.e. how long you keep up this rate. You might notice that the performance of a database engine deteriorates over extended time periods, e.g. because you didn't allow the DB engine to do sufficient background clean-up work (such as SSTable compaction). You should also make sure the benchmark happens on the hardware (e.g. SSD vs. HDD) that you will use in your final production system. (P. 84)
- A "clustered index" is one that also stores the actual data within the index data structure. A "nonclustered index" stores only the references to the data. A "covering index" is a compromise between the two, it stores some of a table's columns within the index. It has the name, because the index alone can already fully "cover" (in the sense of "provide for") the data requested in a query. (P. 86)
- Multi-column indices (P. 87): they cover *multiple* columns of a table.
  - The most simple implementation form is the *concatenated* index, where all the keys of the different columns are combined to a single key, by concatenating them.
  - A more complex form are *multidimensional* indices, e.g. for geospatial data. These are special implementations, e.g. using R-trees. Interestingly, you can "abuse" multidimensional indices also for non-geospatial data. For instance, for a weather app, you could build a multidimensional index

(date, temperature), to efficiently search for all observations during a specific year, where the temperature had a specific range.
        - Full-text search / fuzzy indices: these are special indices that allow for slight deviations (e.g. a specific edit distance) between the search term and the returned results.
- Pure in-memory DBs: (P. 88): traditional DBs have the advantage of requiring little RAM (which used to be expensive), but have a performance penalty, because they have to make sure that data is written to disk in a specific, hardware-friendly structure. However, since RAM has become much cheaper nowadays, we can also use pure in-memory DBs.
    - Some in-memory DBs, e.g. memcached, are purely meant as a *cache*, where data loss is acceptable if the process/machine is restarted.
    - However, in-memory DBs can still be persistent, e.g. by having a battery-powered RAM (to cover power outages), or by regularly persisting the entire memory content to disk in the background, as well as writing a log of changes to disk. The main disadvantage here is that recovery (at the server restart) takes quite a bit of time.
    - What makes in-memory DBs faster for reading is often not due to the fact that they don't read from disk. The OS keeps the most recently accessed disk pages in memory anyway, so a non-in-memory DB is also fast here. Instead, in-memory DBs can be faster because they don't have the overhead of encoding the in-memory data structures to/from a format that is suitable for disks.

OLTP vs. OLAP databases

- OLTP (On-Line Transaction Processing) databases are user-facing (i.e. users indirectly cause read and write queries). There are typically very many (short-lasting) queries, and they need to be completed very quickly, to keep the overall system response time low. The term "transaction" is part of the acronym for historic reasons (as first DBs were built for business transactions), not for technical reasons (it's not related to ACID-transactions). The community kept using the "transactional" term, s.t. it refers to a group of reads and write that form a logical unit. (P. 90)
- OLAP (On-Line Analytical Processing) DBs are not used by end-users, but by business analysts. The queries per second are much lower than for OLTP, but each query may be very expensive and take minutes or hours to complete. Usually, a query involves just a few columns, but millions/trillions of rows, and it computes some statistics/aggregates, such as the count, sum or average. The result of these queries are often put into reports that help the business make decisions (-> business intelligence). (P. 90/91)
- In principle, traditional DBs (which are OLTPs) can also be used for analytics (OLAP). . However, starting in the early 1990s, the term "data warehouse" (being an OLAP DB) was coined, being the dedicated database used for analytics. Reasons included: (P. 92)
    - Performance: OLAP-queries are expensive, and they could negatively impact the performance of the OLTP database, if these queries were simply run on the OLTP database directly.

    - Larger businesses have *many* OLTP DBs, and queries need to include all this data. Consequently, the idea is to transfer the data of all those DBs into a *single* (OLAP/warehouse) DB. Data warehouses are typically filled using the “ETL” pattern (extract, transform, load). This is either done periodically, as batch-processing, or it could also be done continuously via stream-processing. In ETL, *Extract* means to read the data from the OLTP-source (limited to the “newest” rows that needs to be pushed into the OLAP system), *Transform* refers to transforming the data to a OLAP-typical schema, *Load* means to insert the transformed data into the OLAP DB.
- The OLAP DB is basically a read-only copy, as the OLTPs still hold the “ground truth”.
- OLAP DB schemas are typically *relational*.
- Internally, OLAP DBs often store data quite differently, and DB products usually focus on being one type of database (OLTP or OLAP), but not both.
- Data warehouse vendors such as Teradata, Vertica, SAP HANA, and ParAccel have expensive commercial licenses. Amazon RedShift is a hosted version of ParAccel. Recently, many open source SQL-on-Hadoop projects have emerged; they compete with commercial data warehouse systems. Examples are Apache Hive, Spark SQL, Cloudera Impala, Facebook Presto, Apache Tajo, and Apache Drill (P. 93)
- Star and snowflakes schemas: most analytics DBs have a relational schema arranged like a star or a snowflake (P. 93-94):
    - Star schema: at the center you have the *fact* table, where each row is an event that occurred at some point of time. Some of the columns are attributes, some are foreign key references to other tables, the *dimension* tables. The dimension tables store the who/what/where/when/how/why of the event. The “star” schema has its name because, when visualizing it, the fact table is in the center, and it is surrounded by the dimension tables, and the connecting lines look like rays of a star.
    - Snowflake schema: a variant of a star schema, where the dimensions are further broken down into subdimensions. Snowflake schemas are more normalized than star schemas.
- Many (but not all) OLAP DBs are *column*-oriented storages: (P. 95)
    - A column-oriented storage does not store all the values of a *row* closely together on disk, but instead stores values of a *column* together on disk, e.g. each column being in a separate file.
    - The reason why column-DBs are better for OLAP queries:
        - Each *fact* table row has many (sometimes hundreds) of *columns*, but a typical analytical query only reads a few (e.g. 5) of these columns.
        - When an OLAP query tells the DB engine to compute aggregates, a *row*-oriented DB has to temporarily load every row (that matches the WHERE clause) into memory, loading *all* columns of that row (because that’s how the disk access works), which wastes resources/time. For queries involving just a few hundred/thousand rows, that wastefulness would not be a problem, but OLAPs typically contain millions or trillions of rows.
        - Most OLTPs are row-oriented.

    - Sidenote: The Parquet format is also column-oriented -> the “column/row-orientation” concept does not only apply to relational DBs.
    - Column-oriented storage can also more efficiently *compress* data (P. 97)

4. Data encoding and schema evolution

Problem description

- Every application’s code changes over time (due to changing requirements in the real world), and correspondingly, the internally-used data schemas also change.
- The core challenge is that schemas can diverge, because applications are deployed/updated at different points of time - but you still need the ability for newer software versions to read data written by older software versions (and vice versa). Example scenarios (P. 112):
    - *Rolling upgrades* for *server* applications that are spread to multiple nodes
    - *Client-side* applications are updated by users whenever they feel like it.
- The compatibility directions are:
    - Forward compatibility: older code can read data that was written by newer code.
        - This is tricky, because old code must be written s.t. it ignores additions made by newer versions of the code. In particular, if the old code reads an entity, updates it, and writes it back, you must ensure that unknown fields of the entity are still written back - in other words: you cannot completely ignore (as in “exclude”) the unknown fields during reading!
    - Backward compatibility: newer code can read data written by older code
- This chapter examines different binary/textual data formats for encoding (serializing) data, and how they handle schema changes.

Data serialization formats

- While applications often work with data using some proprietary layout in *memory*, they often need to serialize/encode the data to a byte stream, e.g. to save it as file, or transmit via the network to some other process. There are a ton of libraries that solve this problem:
    - ***Language-specific* formats**: (P. 113)
        - Many programming languages have built-in support, e.g. Java’s “Serializable” interface, or Python’s “pickle”.
        - **Often, these libraries are *not* a good choice**, for reasons like:
            - You are *limited to that programming language* (a process written in another language cannot read the data).
            - *Security issues*: the decoding process will instantiate arbitrary classes with arbitrary code.
            - *Versioning* of the serialization format is sometimes missing, or poorly implemented (e.g. when upgrading the programming

language)
                - Efficiency is sometimes bad (regarding run-time or produced message size).
    - ***Textual* formats** (P. 114)
        - Most prominent examples are JSON, XML and CSV
        - Their core advantage is that they are human-readable.
        - Problems with textual formats:
            - Serializing/deserializing *numbers* is often a problem. For instance, XML and CSV don't have numeric types (they are written as string). JSON does not distinguish integers from floats, and does not specify a precision. Also, each programming language may have different numeric concepts, e.g. differently many bytes to store a float/int/long/etc., and there may be unexpected data loss. For instance, JavaScript has problems with parsing numbers larger than 2^53.
            - JSON/XML don't support *binary strings* (that lack a character encoding)
            - CSV lacks schema definitions. XML/JSON do have schema languages, but they are very verbose and complex.
            - The serialized data is very inefficient (regarding CPU time) and large (transmission / storage costs), compared to *binary* formats explained below.
        - Textual formats are still a good choice for *public* APIs. They are the "lowest common denominator", and there's a ton of tooling / libraries available.
    - ***Binary* formats**: (P. 115)
        - Binary formats are a good choice for *company-internal* tools, or when you are working with very large amounts of data, where efficiency (w.r.t. space and time) becomes a serious concern.
        - There are several approaches for binary formats:
            - **Keep the data model, but change to binary encoding:** for instance, you can keep the (in practice often schema-less) data model of JSON/XML, but encode the data as binary: examples are BSON, BJSON, Smile, WBXML, Fast Infoset, MessagePack, etc. - because the schema is missing, the *object field names* (keys) must always be written into the serialized data.
            - **Predefined schema with static code generators:** the schema is baked into the app-code at compile-time. Examples are Apache Thrift, or Protocol buffers (protobuf). (P. 117)
                - You define a schema in a framework-proprietary IDL (interface definition language). You define classes, the field names and field data types, and also assign a "field tag" number for each field. A code generator that parses the IDL file and can generate entity classes (and

parser/decoder code) for all kinds of programming languages.
        - This format is more efficient than approaches such as BJSON, because the field names can be omitted in the serialized output - instead, only field tag numbers are written.
        - Handling of forward/backward compatibility: when new code adds fields, old code can simply *ignore* these fields (fields whose tag numbers it does not know). Thrift etc. allows configuring whether fields are required, optional, and whether they have default values. When adding a new field, you must make it optional (or make it required with a default value), or you would lose being backward compatible. Removing a field is like adding a field, but with backward/forward compatibility concerns being reversed. For instance, removing a (once) *required* field is not possible: old readers (where the field still is required) would consider such new serialized messages (that lack the removed field) to be incomplete and consequently reject or drop them unintentionally.
    - **Dynamic schema** (specified in an IDL) that is transmitted alongside the data - no statically-generated classes are needed (e.g. Apache Avro)
        - You somehow make sure that the process that writes data transmits its “writer schema” alongside the data to the reader (who also knows its reader schema, which may be newer/different compared to the writer schema). This schema-transmission could e.g. happen at the beginning of establishing a connection (when e.g. doing RPC), or writing the schema to the beginning of a file. The library (such as Avro) resolves differences of the writer and reader schema using some advanced algorithm.

Modes of data flow

- There are many ways for data to flow from one process to another (for which you need (de)serialization). The most common ways are: (P. 129)
    - Via databases:
        - Chances are very high that some processes use newer code, and some use older code. E.g. in a “rolling upgrade” scenario.
        - Consequently, you might be interested in both forward and backward compatibility.
    - Via service calls. There are different kinds of service implementations:
        - Web services: a service is a *web service* if HTTP is used as the

underlying transport mechanism. The most popular approaches are REST and SOAP. While SOAP is still used in many large enterprises, it has become much less popular, particular in smaller companies.
    - Among the reasons why REST is so popular is that it is much easier to debug (compared to binary formats), the vast amount of tooling (e.g. for testing, debugging, monitoring, etc.), being available in every programming language. (P. 135)
- RPCs: using ready-made libraries that somehow communicate between different processes (usually over TCP, but not necessarily over HTTP). Most libraries try to do this in a “transparent” way, s.t. It looks as if your code was calling *local* methods, even though they are *remote*. (P. 134)
    - RPC has been around since the 1970s. Older technologies, such as Java RMI or COM, are limited to a specific programming language or platform. Old systems like CORBA were cross-platform, but were very complex and did not provide backward or forward compatibility.
    - There are various problems with the fact that some RPC libraries try to hide the “remote connection” aspect:
        - A local call is predictable (it succeeds or fails). A network call is not - the request or response might be lost, or the remote end might be slow to respond, or be completely unavailable. You have to anticipate such things, e.g. by retrying the request, or building the remote functionality such that it is idempotent (to avoid problems when only the *response* of a RPC call is lost).
        - With a local function, you can pass references/pointers to objects stored in local memory. With RPC, you must serialize these objects (provided to the remote function as parameters). This can become problematic, e.g. for performance reasons (for complex objects), or in terms of functionality (e.g. out-parameters are not possible).
        - When doing RPC across different platforms or programming languages, the RPC library might hide the complexity of incompatible data types (e.g. that Python, C, Java, etc. all have slight deviations). Consequently, unexpected behavior, such as data loss due to rounding of numbers, might occur, which are hard to find or debug. Such problems can never occur in a local call.
- There are newer-generation RPC libraries
    - These have interdependencies between data serialization format and modes of data flow. For instance, Apache Thrift and Avro support RPC. gRPC is an implementation using Protocol Buffers under the hood.
    - They make the remoteness of the calls more explicit (not trying to

hide it), e.g. by exposing Futures. gRPC also supports streams, where a call can contain *multiple* requests & responses.
    - Via asynchronous message passing:
        - The message is not sent directly to a recipient, but goes through an intermediary, usually called *message broker*, *message queue*, or *message-oriented middleware* (MOM), which temporarily stores the message. (P. 137)
        - The message typically consists of some meta-data (e.g. a channel), and a byte-sequence for the payload. You will benefit most by using a data serialization format (for the payload) that is backward- and forward-compatible. (P. 138).
        - Advantages of having a message broker (over RPC): (P. 137)
            - The buffering of messages improves the system *reliability* (in case the recipient is temporarily unavailable, or because the broker can redeliver messages to a crashed process)
            - The sender does not need to know the destination of the receiver (e.g. IP:port), which is particularly useful in highly dynamic environments
            - Messages can be sent to *multiple* receivers.
            - There is a logical decoupling between sender and receiver (the sender does not care who exactly receives the messages).
        - Nowadays, there is a plethora of OSS for message brokers, e.g. RabbitMQ, ActiveMQ, HornetQ, NATS, Kafka. Chapter 11 has more details. (P. 137)
- (Distributed) Actor model (P. 138):
    - The *actor model* is a programming model for implementing concurrency. Instead of using threads, there are actors which have their own (local) state. Actors communicate with each other via asynchronous messages. Message delivery is assumed to be unreliable. An actor always processes one message at a time. There is usually some kind of framework that schedules actors.
    - The *distributed* version of the actor model simply schedules the actors onto *multiple nodes*, typically using a *message broker*. Popular frameworks include: Akka, Orleans, and Erlang OTP.

Part II: Distributed data

Reasons for why you want to distribute data to multiple machines: (P. 145)

- Scalability (as in: be able to handle more read/write requests/sec)
- High Availability (HA) / Fault tolerance
- Latency (for geo-distributed usage patterns)

However, keep in mind that while distributed data systems have above advantages, they also have many disadvantages (broadly speaking: higher complexity, and sometimes a limitation of how expressive the data model can be). (P. 147)

The two general ways to distribute data is **replication** and **partitioning**. Replication makes copies everywhere, partitioning is basically “sharding”. These mechanisms can also be combined (P. 147).

5. Replication

- A basic assumption for replication is that your total amount of data is so small that each individual instance/replica can hold an *entire* copy of the data set.
- Different implementations vary on their concrete approach for replication. Aspects include: (P. 151)
    - Algorithm: single-leader, multi-leader, leaderless
    - Synchronous vs. asynchronous replication
    - Handling of failing replicas

Single-leader replication

- One of the most common solutions in relational databases (e.g. MySQL, Postgres, Oracle), but also for a few NoSQL DBs (e.g. MongoDB or Rethink DB). It is also referred to as *active/passive* replication, or *master/slave* replication.
    - Leader-based replication is also found in other kinds of distributed systems, e.g. message brokers like Kafka.
- One of the replicas is the appointed **leader** (who accepts writes), all other replicas are **followers** (to which clients cannot write data to, but only read data from).
- The leader sends the write-changes to the followers in the form of a *replication log* or *change stream*. The followers take that log and update their local copy.
- Synchronous vs. asynchronous aspect: (P. 153)
    - In many relational DBs, the sync vs. async is a configurable option.
    - Sync replication advantage: follower is guaranteed to have an up-to-date copy (strong consistency) -> no data loss, if the leader suddenly fails.
    - Sync replication disadvantage: write ops would block if only a single follower becomes unavailable. This would make systems completely impractical. Therefore, in practice, “sync replication” means that only **one** of the followers is actually synchronous - the others are async. The system internally re-elects who is the sync follower, should one go down or become slow.
- **Handling node outages** (P. 156)
    - Nodes can go down (temporarily) for any kinds of (controlled or uncontrolled) reasons.
    - **Follower failure:** Basic idea is “catch up”: once up again, the follower reconnects to the leader and requests all data changes that occurred in the meantime. Once it has caught up, the follower accepts read requests again.
    - **Leader failure** via failover:
        - Failover means: one of the followers must be promoted to be the new leader, clients need to be told about the new leader, and the other

followers also need to know the new leader
    - There are many intricate details, e.g.:
        - How to reliably detect a failed leader? Timeouts are used, but they are not foolproof. What should the timeout value be? Consider that sometimes nodes are just temporarily slow, and starting a failover in a system that is already at load-capacity will make things even worse.
        - How to elect the new leader? It should be a follower who has the most up-to-date data.
        - If the old leader comes back, it has to be made to realize that it is no longer the leader and that it has to step down.
        - What to do if an old leader comes back (and the follower that became leader back then was not fully up-to-date): what to do with the writes the old leader still had saved? There might now be conflicts!
        - What to do in a “split brain” situation where two nodes think they are a leader and accept writes?

Problems with replication lag

- Replication lag = delay between a write to the leader and it being readable from a follower. (P. 162)
- Under normal circumstances, it is only a fraction of a second and not noticeable in practice. But when operating near capacity (or when having network problems), the lag can increase to seconds or minutes.
- Longer replication lag causes inconsistent behavior in applications that are not prepared for this.
    - Most typically, you are interested in “read-after-write” consistency, which is a weaker variant of “strong consistency”, because it is only concerned with the writes/reads of a *particular* user, not of all users. A technique to implement this is that when you read something that the user (or maybe someone else) recently modified, read it from the leader. Otherwise, read if from the follower. The client could send meta-data along with the query that contains the most recent write-timestamp of the client, and replicas then know whether they should answer the query (if they have received data stamped with newer or equal to that client-timestamp), or whether they have to redirect the query to another replica (P. 163)
    - Another kind of consistency you might want is “monotonic reads”, or “things cannot move backwards in time”, or “after reading newer data, you won’t re-read older data” (P. 165). A counter-example (where *monotonic reads* consistency is violated) is when a client first reads a list of 3 items from a more up-to-date follower, then refreshes the list and only gets 2 items from a more out-of-date follower. One way to achieve it is to always direct reads of a specific user to the same replica
    - Another kind of consistency is “consistent prefix reads” (P. 165), which says that

if a sequence of writes happened in a certain order, then anyone who reads those writes also must read them in that order". This is particularly a problem with sharded DBs, where each partition operates independently (while the writes ending up in different shards)

Multi-leader replication

- Also known as master-master or active-active (P. 168)
- The basic idea is that every leader is also a follower to other leaders.
- The typical use case is geo-distributed scenarios (P. 168).
    - Another one is where a client wants to work "offline first", and have a local cache of the data, e.g. CouchDB.
- It is also possible to *combine* the multi-leader with the single-leader (i.e., leader-follower) approach, e.g. having one leader per data center (and *within* each data center, you use the *single-leader* approach).
- Advantages of multi-leader over single-leader (P. 169) in a distributed data-center (DC) use case:
    - Better performance (in terms of lower latency for writes, because now writes are propagated to other DCs in the background, and in terms of high availability, because we can just always accept writes made in one DC even if all other leaders in the other DCs are currently down)
    - Tolerance of DC outages (no failover needed, as would be the case with "single-leader")
- Main disadvantage of multi-leader approaches: conflicts can arise, which somehow need to be solved
- Multi-leader replication is often implemented by *external* tools, e.g. Tungsten Replicator for MySQL/MariaDB, BDR for Postgresql, or GoldenGate for Oracle.
- Conflict handling (P. 172)
    - One approach is to *avoid* conflicts, e.g. by making sure that always a particular DC/replica is used for a reading+writing specific kinds of data (e.g. data of a particular user), but this is very susceptible to errors and edge cases (e.g. what happens if the user moves elsewhere and should now use a different DC?).
    - Every conflict resolution mechanism must ensure that the written data values eventually have the same *convergent* value. There are many techniques for this, such as: (P. 173)
        - LWW (Last Writer Wins), throwing away data from other writers (e.g. having the comparison be based on a random ID generated for each write, not necessarily on wall-clock-timestamps)
        - Merge the values together, e.g. by sorting them and concatenating the sort result
        - Record the conflict in a separate data structure that preserves all information - the application code then resolves the conflict later. This could (or could not) involve the user.
            - Conflict resolution at *write* time: Many multi-leader replication tools

also allow you to write conflict resolution handler callbacks, which cannot involve the user.
        - Conflict resolution at *read* time: When data is read, the DB returns all conflicting items to the application, which can then involve the user in merging the result. E.g. done in CouchDB.
    - Chapters 7+12 have more details on conflict detection & resolution.
- Replication topologies: (P. 175) Various topologies can be used. Most commonly, you have a peer-to-peer (all-to-all) topology, but you could also have a "ring-buffer" (circular) topology, or specific *tree*-shaped topology (e.g. *star*-topology).
    - Some of the topologies (e.g. ring buffer) have poor fault tolerance. The peer-to-peer topology has the best fault tolerance, but needs advanced mechanisms (such as version vectors) to properly detect the causality (happens-before relationship) of the distributed updates, to order the updates correctly.
    - It makes sense to thoroughly test a multi-leader database (including edge cases where nodes go down) to see whether it really behaves properly.

Leaderless replication

- Basic idea: *every* replica accepts writes from clients. (P. 177). Clients either send their writes to several replicas in parallel, or to a coordinator node (which then sends it to all replicas). The client assumes that a write was successful if it was confirmed by some kind of majority of nodes. Similarly, to read values, the client must read from several replicas in parallel and analyze the responses. Modern implementations use version vectors in the clients and servers to detect conflicts.
    - What was just described is "read repair". Another approach (that some DBs use *in addition* to read repair) is "anti-entropy process", where replicas send writes internally to each other to catch up, but the writes are not sent in any particular order (there is no *replication log* being used). (P. 179)
    - Note that there are also no "rollbacks", e.g. if 3 replicas accepted a write and 4 replicas did not, the write is not rolled back on those 3 replicas.
- Examples: DynamoDB, Riak, Cassandra, Voldemort.
- Writes when nodes are down (P. 178): leaderless DBs do not have a "failover" phase, see above for "read repair" and "anti-entropy process"
- The general approach for reading/writing is also called "quorum". If you say that you write to **w** replicas and read from **r** replicas, and there are a total of **n** replicas, then **r+w > n** must hold. (P. 179) In many DB implementations, r/w/n are configurable. It is common to make n an odd number (e.g. 3 or 5) and to set r = w = (n+1)/2.
    - To consider an extreme example: setting w=1 (write is already considered successful if it was received by just one of the replicas), then, by following above formula, r would have to equal to n - this makes sense, because to determine which one is the most up-to-date value, you'd have to read from *every* replica, to be sure to also read from the one replica that did receive the write. However, in general it is not a good choice to have r or w be equal to n, because then reading or writing requires all replicas to be up.

    - Note: in reality there may be many more than **n** nodes, and the **n** is simply chosen as the set of *designated* nodes (for a client, e.g. based on its geographic region). There is the concept of the “sloppy quorum” where the client will send writes/reads to other, non-designated nodes in case it cannot get a quorum from its **n** designated nodes. These other non-designated nodes that accepted a write will then attempt to asynchronously propagate the writes back to a designated node, once possible. (P. 183/184)
- With leaderless replication, the consistency guarantees are somewhat wonky. Therefore, you should only use such DBs if the requirements of your application can tolerate eventual consistency. You should understand the parameters w and r as only tweaking the *probability* of reading stale values, but not as strong guarantees.
    - See P 181/182 for explanations.
- Leaderless replication DBs are good for use cases that require HA, and low latency, but tolerate stale reads occasionally. (P. 183)

6. Partitioning

- The term “partition” is the most established term, but concrete products also use different terms, e.g. “shard”, or “region”, or “vnode” (Cassandra), or “tablet” (BigTable), or vBucket (Couchbase). (P. 199)
- Replication and partitioning are often combined, i.e., you do partitioning, but also replicate each partition to multiple nodes, to get HA. When using leader-follower replication, a node may store multiple partitions and you would typically configure that node to be the leader for one of the partitions, and a follower for all other partitions. (P. 200)

Partitioning of key-value data

- Deciding how to split data into multiple partitions is a difficult problem: you want to avoid a *skewed* distribution, because then the performance of your system suffers - an extreme example would be that one node gets almost all the load, while other nodes get almost no load at all. This one node would then be called a “hot spot”. (P. 201)
- Two commonly used approaches for partitioning are:
    - Partitioning by **key range**:
        - Assuming that your key can be sorted somehow (e.g. alphabetically/lexicographically), the basic idea is that you break down the entire range (e.g. AAAAA-ZZZZZ) into multiple sub-segments, and each such sub-segment is a partition.
        - The distribution should most commonly not be a *uniform* distribution, but consider the amount of data you expect to have in each range. For instance, if your keys are titles of books, then the partition A-D (books whose titles start with letters A to D) would contain many more books than the partition W-Z, and you would get a skewed distribution if you

always made each partition cover 4 letters of the alphabet.
        - Often DBs let you configure whether you want to choose the partitions manually, or have the DB figure it out dynamically. (P. 202)
        - Core advantage of this approach: you can do *range queries* efficiently, because the keys are already in sorted order.
        - Examples: HBase, RethinkDB (P. 212)
    - Partitioning by the **hash of the key**:
        - A hash function transforms any input (the key) into a uniformly distributed number range, e.g. 0 - 2^32.
        - The main disadvantage is that you can no longer do range queries over your keys *efficiently*, because the hashes of the (adjacent) keys are now scattered over all partitions. Some DBs still support range queries, but need to send them to all partitions and the DB engine then combines the results - in the end, you need to read your DB's documentation closely. (P. 204)
        - Even though key hashing reduces hot spots (= a particular partition becoming overloaded), you can still get hot spots if all the writes/reads are for the *same* key. DBs are unable to handle those hot spots well - you need to handle them in your application in some way. E.g. breaking down hot keys into multiple sub-keys, e.g. by adding numbers to the end of the key (and maintaining a break-down table so that you know how to combine the data again for *read* queries). (P. 206)

Handling secondary indices

- By "secondary index" we simply mean other indices that you create for columns other than the PK - the use case most often is the ability to search (or sort) using these other columns (e.g. "find all cars whose *color* has value *red*"). (P. 206)
- Given that the data (tables) is split into partitions in some way, the core problem is how to then split the secondary indices themselves into partitions, to get the best performance. There are two approaches:
    - Document-based partitioning (a.k.a. "local index"): (P. 207)
        - In this approach, each partition has its own collection of indices, which cover only the documents stored in that very same partition.
        - When reading/searching data, the read query must be sent to all partitions, and the DB engine must combine the results. This approach is also called "scatter & gather".
        - The main problem of this approach is performance: you get tail latency amplification. Tail latency (for an *individual* service) means that there is a latency distribution of responses being very fast (e.g. < 10 ms response time) in ~99% of the cases, but in 1% of the cases, you have a much higher latency, the "tail" latency (e.g. 100 ms). If you now have not one but *many* services, the probability of tail latencies increases (to more than 1%).

        - Databases such as MongoDB, Riak, Cassandra, Elasticsearch, SolrCloud or VoltDB use this approach.
    - Term-based partitioning (a.k.a. “global index”): (P. 208)
        - The DB engine creates a global index that covers the data of *all* partitions. Storing that global index only on one node would be bad (performance bottleneck, low availability), therefore this global secondary index is itself also being partitioned by “terms”, i.e, the index partitions are different from the partitions of the data (primary keys).
        - The term originally comes from full-text indices, where “terms” refer to all the words that occur in a document. A typical form for a term is *“<column name>: <value>”*, i.e., a (search) term also includes the <column name>, not just all the different values (that occur in the different rows of the table).
        - Partition can happen by…
            - the term itself: useful for efficient range queries
            - the hash of the term: improves the even distribution of the index among the partitions
        - The main advantage of this approach: reads are more efficient (compared to *document-based* partitioning): there is no scatter&gathering, but the query is sent to that partition that covers the term.
            - The main problem is that writes are slower: creating/updating a document now means that you may have to update several partitions of the *index*. Because this takes time (and there might be failures, reducing the availability), DB engines using this kind of approach typically update the secondary indices asynchronously.

Rebalancing partitions

- *Rebalancing* refers to the act of moving *load* (not necessarily just *data*, it could also be requests-rerouting) from one node to another. (P. 209)
- The motivation behind it: over the lifetime of an application, things change, such as
    - Higher load (which requires you to add more CPU via vertical or horizontal scaling)
    - Increased amounts of data (you need more disks or RAM)
    - Nodes fail occasionally (so other machines need to take over)
- Basic requirements for rebalancing:
    - After rebalancing, the *load* should be evenly distributed on the nodes
    - While rebalancing happens, the DB should still be readable and writable
    - Rebalancing should be as fast as possible, e.g. by moving only the minimal amount of data necessary
- Rebalancing strategies (P. 210)
    - **Fixed number of partitions**:
        - From the start you create many more partitions than there are nodes (e.g. 1000 partitions, even though you only plan to have 10 nodes), and you

assign multiple partitions to each node.
        - When adding more nodes to the cluster, these nodes “steal” a few partitions from each of the existing nodes. If a node is removed, the same happens in reverse.
        - You only move entire partitions between nodes.
        - If you have some nodes that are more powerful than others, they can also be assigned to more partitions than the other nodes.
        - This approach is used in DBs like Riak, Elasticsearch, Couchbase or Voldemort.
        - Downside of this approach: each partition has a management overhead, so it makes no sense to always choose e.g. 100’000 partitions (which would give you neatly “small” partitions that would also rebalance faster). There is no silver bullet here.
    - **Dynamic partitioning** (P. 212)
        - Especially suitable for DBs that partition by *key range* (see above, or P. 202), where a fixed number of partitions would make little sense, because if your data is skewed, then also the load would be very skewed.
        - Whenever a partition grows and exceeds a pre-configured size (e.g. 10 GB), it is split into two partitions.
        - Main advantage: partitions (and their overhead) adapt to your data
        - To avoid that you *initially* have only *one* partition (which could be overloaded), you can tell DB engines to do some kind of *pre-splitting* where you configure the DB engine your best guess of how the partitioning should look like
    - **Partition proportionally to nodes**
        - You configure a fixed number of partitions per node. Thus, the partition sizes grow together with the data
        - When a new node joins the cluster, it randomly chooses a fixed number of existing partitions to split, and then takes ownership of one half of each of those split partitions while leaving the other half of each partition in place. The randomization can produce unfair splits, but when averaged over a larger number of partitions, the new node ends up taking a fair share of the load from the existing nodes
        - E.g. used by Cassandra and Ketama
- Operations consideration: automatic vs. manual rebalancing (P. 213)
    - There is a fluid spectrum between “fully manual” and “fully automatic”. For instance, DBs like Couchbase or Riak make suggestions for a new partition assignment to the admin, who then has to approve it before it is applied.
    - Fully-automatic rebalancing is convenient, but dangerous because it is unpredictable. Since rebalancing is an expensive operation, it might happen too often and overload your nodes and the network.

Request routing

- Given that your DB has many nodes, the question of *routing* is: to which node (IP/port) does the client have to send its read/write request? (P. 214)
    - More generally, this problem is referred to as *service discovery* and it applies to many more areas, not just DBs.
- Different approaches are:
    - Clients may contact *any* node (e.g. using a round-robin load balancer). The node handles the request if it is in charge of the partition, otherwise it forwards the request to the correct node, and also forwards the reply back to the client.
    - Clients send the request to a routing tier first, which knows about all the nodes and forwards the request to the right node. That routing tier is like a partition-aware load balancer.
    - Clients themselves are aware of the partitions and connect to the correct node right away.
- No matter which approach you choose, it is important that the layer that decides which node to send a request to is actually correct about this decision. But nodes come and go all the time.
- Many distributed data systems rely on a coordination service such as ZooKeeper to keep track of this cluster meta-data, e.g. Solr, Kafka or HBase.
- Other DBs rely on a gossip protocol that happens directly between the nodes (e.g. Cassandra or Riak)

7. Transactions

- Transactions are a good mechanism to reduce the issue of “partial failures”, such as: (P. 221):
    - The software or hardware of the DB fails, e.g. in the middle of a sequence of write operations
    - The application (that uses the DB) crashes at some random point, e.g. after 2 of 4 of the DB operations it wants to make
    - The network is interrupted at any point of time, cutting the app off from the DB (or DB nodes from each other)
    - Several clients try to write to the DB at the same time, overwriting each other’s changes
- Transactions group several read/write operations into a logical unit, executed atomically, i.e., they are either all successful, or they all fail.
- Transactions simplify error handling for the application developer - they no longer need to worry about partial failure scenarios such as those presented above. (P. 222)
- Note: applications sometimes don’t need transactions!
- Single-object vs. multi-object operations (P. 228):
    - A transaction is usually a mechanism of grouping multiple operations that apply to *multiple* objects (P. 230).

    - Examples are:
        - *Foreign key* constraints between different DB tables, which must remain valid
        - In a document DB data model (which lacks the JOIN functionality), data is denormalized - transactions are useful to prevent denormalized data from becoming out of sync.
        - Databases with *secondary* indices: whenever you change values, these indices also need to be updated

ACID definitions

- ACID has been defined in 1983 as follows (P. 224+):
    - Atomicity: should have been called “abortability”: it means that if you abort a transaction, all writes from that transaction are undone.
    - Consistency: invariants (i.e., statements that always hold) about your data, that must always be true. These are defined by the application. For instance, in an accounting application, the credits and debits must be balanced. The application has to define and *ensure* the invariants - not the DB engine (so “C” in “ACID” does not belong there, it was added to be able to build a nice acronym).
    - Isolation: concurrently executing transactions are isolated from one another - there are many different levels of isolation, the strictest level being *serializable* transactions, which means that when the transactions have all been committed, the result is the same as if they had been run in (some) serial order (even though they actually executed in parallel).
    - Durability: the promise that once a transaction has committed successfully, any data it has written will definitely not be forgotten (exceptions are when hardware fails some time *after* it has reported that the data has been successfully written to it, and there are no backups)
- Unfortunately, in practice, the ACID implementation of DB #1 does not equal the ACID implementation of DB #2 (P. 223). The term “ACID compliance” has become an under-defined marketing term, because some subtle details are not clarified in the ACID definitions, so the DB vendors can choose their own interpretation.

Weaker isolation levels

- Because the *serializable* isolation level has severe performance problems, many DBs don’t implement it, but they instead implement weaker (non-serializable) isolation levels, which prevent some (but not all) concurrency issues
- Common problems with parallel transactions (and lack of isolation):
    - Dirty reads: transaction A sees data written by transaction B (even though B has not yet finished, i.e., has not yet committed these changes)
    - Dirty writes: both transactions A and B overwrite the value of the same data item (e.g. a specific row and column of a specific table) in parallel, and then A commits, thus overwriting the (not yet committed value) of B. If B were also

allowed to complete successfully, it would overwrite A's changes.

- Lost updates: if transaction A reads the value of a data item, then transaction B writes and commits a changed value for that data item, and then A also modifies and commits the value of that same data item. The value B changed is lost. (P. 243)
  - *Lost updates* are different to *dirty writes* because B commits even before A has done its first (not-yet-committed) write operation.
  - Approaches that addresses/fixes Lost updates specifically by forcing the write ops to be executed serially are:
    - *Atomic write operations* (P. 243), e.g. a special compare-and-set operations offered by the database, or putting the modification-math into the DB itself, e.g. `UPDATE counters SET value = value + 1`
    - *Explicit (row) locking*, e.g. with `SELECT … FOR UPDATE`, which causes the DB to lock all rows returned by the query. (P. 244)
  - Other approaches are to allow transactions that both write to the same rows in parallel, but once the user wants to commit, the transaction manager detects lost updates and refuses the commit. (P. 245)
- Non-repeatable reads (a.k.a. *read skew*): a *temporal* inconsistency where transaction A reads different values for a specific data item over the course of its lifetime (and A has not written to this value itself). Can e.g. happen if you have a sequence such as: A reads a value, transaction B overwrites and commits the value, A reads the value again. This is problematic whenever you have long-lasting read queries, e.g. analytic queries, integrity checks, or creating full database backups. (P. 238)
- Write skew: a generalization of the *lost updates* problem, where the updates of the transactions each affect *different* data items (whereas, with *lost updates*, they affect the *same* data item). (P. 248)
  - Can best be prevented using *explicit row locking*

Isolation level	**Solved issues/problems**	**Implementations that provide this isolation level**
Read uncommited	Prevents *dirty writes*, but not *dirty reads*.	Dirty writes are prevented using row-level locks.
Read committed	Prevents *dirty writes* and *dirty reads*. It is the default level in many traditional SQL DBs such as PostgreSQL, Oracle or MS SQL Server.	Dirty reads are usually not prevented by locks (because that would cause stalling of all transactions if there is only a single long-lasting transaction). The DB remembers both old committed values and new values set by the transaction

		that currently holds the write lock. Other transactions reading the value are given the old value.
Repeatable read (isolation level is not really standardized, nobody really knows what it means, it is e.g. confusingly called “serializable” in Oracle DB)	Prevents all of the above and non-repeatable reads.	Snapshot isolation (P. 238). Each transaction reads from a consistent snapshot created at the start of a transaction. Readers never block writers and vice versa.
Serializable	Prevents *all* problems	Implemented using approaches such as: ● Serially executing short transactions ● 2-phase locking ● Serial snapshot isolation Details are below

Serializable isolation level

- It offers the highest level of isolation for concurrent transactions, preventing all race conditions / problems.
- Approaches of implementation:
    - Serially executing short transactions
        - Basic idea: database only allows one transaction at a time (concurrency is not allowed). To avoid that the throughput is horrible (which would be the case if transactions were long-lasting, waiting for the application or user), the transactions are not allowed to be interactive. Instead, the application must submit the entire transaction code to the DB, as stored procedure (which executes very fast, without waiting for I/O) (P. 254)
        - The data processed in a transaction should fit into memory, for the transaction to be fast (P. 256)
        - Approach is e.g. implemented in VoltDB/H-Store or Redis
        - Stored procedures have a few disadvantages, e.g. that you are restricted to a small set of programming languages, there is almost no ecosystem of libraries, they are harder to debug, difficult to keep in version control or to deploy, more tricky to test, or integrate with monitoring systems (P. 255)
    - 2-phase locking (2PL)
        - Note: don’t confuse 2-phase commit (2PC) with 2PL!
        - Basic idea: multiple concurrent transactions are allowed to read the same data items in parallel (as long as no transaction is writing). But once a transaction wants to write to a data item, it must *exclusively lock* it. Readers lock writers, and vice versa. (P. 257)
        - This approach has its name because in the first phase, a transaction acquires a lock (while executing). The second phase, at the end of a

transaction, is when a transaction releases all its locks.
        - The DB engine automatically detects dead-locks between transactions and aborts one of the transactions (which the application then needs to retry).
        - Biggest disadvantage of this approach: performance is poor, not only because many of the concurrent transactions have to spend time waiting for other transactions to finish, but also because of the overhead of acquiring and releasing the locks. Thus, the overall latency of your database is hard to predict (P. 258/259).
    - Serializable snapshot isolation (SSI, P. 261)
        - A rather new approach that is not implemented by many DBs yet. It is an *optimistic* approach that has a very small performance overhead, still allowing for high levels of concurrency.
            - E.g. used in Postgres
        - The approach is very similar to *snapshot isolation* (used to guarantee *repeatable reads*), but with added checks (at the end of a transaction) that abort those transactions that would have violated the *serializable* isolation level. (P. 262)
        - SSI performs better than 2PL because there are no locks that transactions need to acquire/release. However, if you have a very write-heavy system, the percentage of aborted transactions will be quite high. (P. 265)

8. Issues with distributed systems

This chapter looks at all things that can (and will) go wrong, in theory and in practice, in a distributed system. It also looks for some mitigation strategies (in the sense of “don’t do/assume this or that”), but actual solution techniques are discussed in the follow-up chapters.

Faults and (partial) failures

- Distributed systems consist of many parts, and when only *some* (but not all) parts fail, this is known as *partial failure*. They can happen (non-deterministically) at any time, randomly. (P. 275)
- There are many kinds of hardware failures, such as: (P. 275)
    - Power loss (generator not kicking in) -> hardware is rebooted (=”power cycle”). Can apply to just a single rack, or even an entire data center (DC)
    - Backbone failures (entire DC becomes unreachable)
    - Networking hardware (e.g. switches) failing, e.g. causing network partitions in a DC
- With a *distributed system*, the approach generally is to anticipate such partial failures, and to tolerate them, keeping the system still operational. We need to build fault-tolerant mechanisms into the software. In other words: build a reliable system from unreliable components.
    - In contrast, a *supercomputer* (which is similarly powerful as a distributed system, computationally-speaking, but is still like a *single*-node machine) handles *partial failures* differently: it escalates it to a *total* failure: if something crashes, the entire

machine stops working (e.g. kernel panic).

- The larger a distributed system is, the more likely it is that *something* is broken. For a system with thousands of nodes, some nodes are *definitely* broken at any given time. (P. 276)
- It always pays off to artificially create partial failures in a testing scenario, to see what happens.

Unreliable networks

- The Internet and internal networks in a DC are of type *asynchronous packet-switched networks*. Here the network gives no guarantees as to *when* or *whether* a packet will arrive. (P. 278)
    - There are also other kinds of networks, *synchronous* ones, which are *circuit-switched* networks. They guarantee a maximum RTT, thanks to provisioning fixed resources (the circuit). However, in practice such synchronous networks are not used, because people favor that operations complete as fast as possible, which requires that the network can handle *bursty* traffic patterns. Synchronous networks would be a horrible waste of resources, because each circuit is given a *fixed* data rate, and to allow operations to complete quickly, you would have to reserve huge data rates per circuit, and the circuits would be sitting idle most of the time (and you would also get only a few circuits in total if each one is “fat”). In other words: *latency guarantees* of a circuit come at the cost of reduced *utilization* of the transmission medium, making it expensive. (P. 285/286)
- Things that can go wrong in an asynchronous packet network:
    - The request may get lost (e.g. unplugged network cable)
    - Request waits in some queue (e.g. when a router somewhere along the route is overloaded)
    - Network route is fine, but the remote node/process may have failed (e.g. power failure)
    - Remote node still lives, but can temporarily not respond (e.g. a “process pause”, see below, e.g. Garbage Collection cycle)
    - Remote node/process did process the request, generated a response, but the response was lost by the network (or the response was delayed due to some queue).
- Main problem: when a client does not receive a response, it can (mostly) not distinguish / determine which of the above issues applied. (P. 279)
    - There are some exceptions, e.g. specific error codes you get such as “destination not reachable” (no route to host) or “socket connection refused” (if the remote node lives, but the destination process is dead). Your client might also have access to (proprietary) information, e.g. by calling the proprietary API of a network switch that can tell the client about link failures on the hardware level.
- In practice, this inability to distinguish the underlying issue makes the task of *detecting faults* much more difficult (e.g. when a load balancer needs to stop sending requests to a

dead node, or when a single-leader replication DB needs to detect that the current leader has failed). (P. 280)

- The go-to approach to detect node failures is to use timeouts, but there is no silver bullet. Too short timeouts enable the system to be more responsive, but it might detect faults too quickly and declare nodes dead that are just slow. The effect is that the same actions are executed twice (by the old node declared dead, and a new node taking over the responsibilities). Transferring responsibility over to another node also makes the distributed system (as a whole) slower, temporarily. The worst case is “cascading failures”, where all nodes declare all other nodes as dead - everything stops working. (P. 281)
    - It is usually better to use *dynamic* instead of *constant* timeout values. The system continuously measures round trip times (and its variability / jitter) and automatically adjusts the timeout values, based on the *distribution* of response times. E.g. using a “Phi Accrual failure detector”. Something similar is done in TCP to determine the retransmission timeouts. (P. 284)
- In practice, network failures occur all the time, even with redundant network hardware. Human error (e.g. misconfiguring a router or switch) is the most common cause of outages. There are crazy things that can go wrong, e.g. a shark biting an undersea network cable, or an automated router firmware upgrade causing havoc. Also, just because a connection works in one direction, it does **not** imply that it also works in the opposite direction (one-sided packet loss). (P. 279)
- There are many places in a network request where **queueing** comes into play. Places of queueing include: (P. 282)
    - The network switch
    - The OS (when the application is temporarily not accepting new incoming packets for various reasons, including that the OS does not allocate CPU time slots to it, e.g. because of a *noisy neighbor*)
    - The VM hypervisor (if applicable), when a VM is temporarily suspended - the Virtual Machine Monitor (VMM) queues the packets in this case
    - TCP’s flow control (a.k.a. Congestion avoidance, or backpressure), where the *sender* queues packets before even sending them into the network.
- TCP vs UDP: UDP is good when you don’t need reliability or flow control - it’s good for use cases where delayed data is not useful.

Unreliable clocks

- Clocks are a very important concept in distributed systems, because they measure durations (time intervals), and they can provide (absolute) points of time. (P. 287)
- Distributed systems often need to determine a (global) order of events. This is made very difficult by the fact that message transmissions are unreliable and arbitrarily delayed. (P. 287)
- Using absolute points of time provided by (hardware or software) (wall) *clocks* is generally a bad idea. Clock synchronization is close to impossible to get right, as there are many reasons why it might fail (see P. 290). Hardware clocks drift unpredictably (e.g.

based on hardware temperature). The synchronization of clocks causes them to jump forward or backward in time, and they also may have poor resolution. (P. 288)
    - Reading absolute timestamps *should* be implemented as reading a confidence interval (e.g. Google's TrueTime API in Spanner does this). But in practice, the OS APIs will give you a very precise-looking (single) timestamp, because it is already quite difficult to even estimate the uncertainty interval. (P. 293)
- Alternatives to wall clocks are monotonic clocks and logical clocks:
    - Monotonic clocks are called this way because they are guaranteed to move forward in time. Their absolute value is meaningless (you always need to compute the difference). There are still issues with these clocks: the speed at which time of a monotonic clock moves forward is still determined by the hardware clock, which may be influenced by NTP (called "clock slewing" - not skewing!). The resolution is usually good (up to microseconds).
- Another problem is *process pauses* (P. 296), where the active thread is paused for longer periods of times (seconds, or even minutes).
    - There are many reasons for this:
        - Many programming language runtimes have a garbage collector. Even though the GC can be tuned, you still must assume the worst (long GC pauses)
        - A hypervisor might suspend a VM for an arbitrarily long time period (e.g. when migrating a VM to a different host).
        - End-user devices (e.g. laptops) might be suspended/resumed at any time by the user.
        - The OS context-switches to another thread. If the load is high (e.g. if there are many threads), the pause time might become quite long. Thread suspension happens also during I/O requests, and sometimes there might be I/O access at surprising places (e.g. Java's classloader when it lazily loads class files, or when you have disk swapping configured in the OS, and a memory page is not really in RAM but on disk and thus must be loaded into RAM first).
    - Threads are not notified of such pauses at all. (P. 298)
    - For single-node machines (and multi-threaded code) there are good tools for making applications thread-safe, e.g. mutexes, semaphores, etc. But there is often no good equivalent toolset for *distributed* systems. (P. 298)
    - There are techniques that allow you to eliminate process pauses, e.g. using hard real-time (operating) systems (RTOS) where all levels of the software stack are customized to provide real-time guarantees (with an upper bound). That is, not only the OS-provided stack, but also the programming language runtime above need to be optimized for real-time use cases, which only few languages do. Developing real-time applications is therefore very expensive, and thus it is only done in safety-critical embedded systems (e.g. airbag software). For "normal" commodity server software/hardware, implementing these real-time guarantees is not economical. (P. 299)

What a system knows, and what is really true and false

- Given that the observer (e.g. a node in a distributed system) has only very unreliable means for measuring the state of the system around it (it cannot even trust its own judgment, e.g. in case of *process pauses*), it is almost a philosophical question to ask: "what does the observer (really) *know* to be true or false in the system?". (P. 300)
- Because a *single* node cannot reliably judge a situation, distributed systems commonly rely on some kind of *quorum*, that is, some kind of (majority) voting among *multiple* nodes. (P. 301)
- What is often done is to design a *system model* (where you proclaim assumptions about the behavior of the system), and then design algorithms and prove that they behave correctly within the just-designed system model. Unfortunately, the system model is not the same as the real system (on real hardware), so you can only prove algorithms and models to be correct, but not actual implementations of them. (P. 300)
- The problem with *locking* (i.e., a distributed mutex) (P. 302): a simple/normal mutex (as known from single-node systems) does not work in a distributed system, because any single node cannot truly rely on knowing that it holds the lock. There are lease-based locks (which are time limited), but they are still unreliable (e.g. thanks to process pauses). A common approach to solve the problem are *fencing tokens*: whenever a lock server grants a lock or lease, it also returns a fencing token (e.g. a monotonically-increasing number). Whenever the lock is used to access a resource, the client also has to provide the fencing token, and the resource has to validate that this token is (still) valid.
- Distributed systems discussed in this book generally assume that no node is deliberately (or accidentally, due to a bug) "lying" to other nodes. That is, we assume that there are no *Byzantine faults*. Designing algorithms to be Byzantine-fault-tolerant is very complicated and therefore too costly. (P. 305)

9. Consistency and consensus

This chapter discusses some general approaches / algorithms / abstractions (such as *total order broadcast* or *consensus*) that provide certain consistency/consensus guarantees. These abstractions hide problems such as disk failures, crashes or race conditions. Whenever you have a working *implementation* of such an abstraction (suitable for your application), you can implement your application on top of this abstraction, reusing it. (P. 321)

Concrete example: given a solution for *consensus*, and you are developing a database (= the application), you can use consensus to elect a new leader in case the old leader fails.

Consistency guarantees

- Most of the *replicated* databases provide at least *eventual consistency*, which is a very weak guarantee. It is hard for developers to understand (who are traditionally used to stronger guarantees from traditional RDBMS). Developers must understand the limitations, and may not (accidentally) assume too many guarantees. (P. 323)
- There are many stronger consistency models, but their two common problems are (P. 323):
    - Worse performance
    - Less fault-tolerance and availability
- The strongest consistency guarantee is *linearizability / strong consistency / atomic consistency*. It means that the system behaves as if there was only *one* copy of the data, and that all operations on it are atomic. Reads are always including the most recent writes. (P. 324)
    - Use cases of linearizability: (P. 330/331)
        - Electing a *single* leader
        - Locking (e.g. to implement leader election - the lock holder is the leader)
        - Uniqueness guarantees
        - When multiple backend systems work on the same data (that was e.g. inserted into a distributed DB), and each backend system expects to read the most recent writes
    - Services such as ZooKeeper or etcd can implement many of the above use cases. The Java library Apache Curator provides higher-level recipes on top of ZooKeeper (like leader election, shared locks, barriers, etc.).
    - Note that you should think carefully whether your use case *really* needs linearizability. For instance, you may conceptually want a *uniqueness guarantee*, but in practice it might make more sense to allow “overbooking” and fix it after the fact (maybe compensating customers for their trouble with a coupon), because you can then use more scalable systems that generate more business (which generates more money than it costs you to compensate unhappy customers). (P. 331)
    - *Replication* is a common method to make a system fault-tolerant, but not every kind of replication mechanism can be made linearizable: (P. 333)
        - Multi-leader replication is not linearizable, because they asynchronously propagate writes and there could be conflicts.
        - Single-leader replication (DB) implementations may or may not be linearizable. E.g. if they use *snapshot isolation*, they are not. Also, in certain situations (such as split-brain) they are not.
        - Leaderless replication is usually not linearizable because of the internal techniques they use. For instance, *sloppy* quorums (that are usually used in practice instead of strict quorums) are not linearizable. And even strict quorums are not, because of timing issues, as illustrated on P. 334.
    - The costs of linearizability:
        - Poor availability: any system that is linearizable loses availability once there is a network partition. (P. 336)

        - Poor performance: it has been mathematically proven that the response time of read/write requests in linearizable systems is at least proportional to the uncertainty of delays in the network; this uncertainty is high in unreliable networks such as the Internet. (P. 338)
    - Side note about the CAP theorem (P. 337): it only considers one consistency model (linearizability) and one kind of fault (network partition), but it ignores network delays, dead nodes, or other trade-offs. Thus, it has little practical value when designing distributed systems. Today, it is only of “historical” interest.

Ordering guarantees

- Ordering of events or operations is an important concept in distributed systems (P. 339)
- If you consider *causality* (e.g. the “happens-before” relationship): causality requires ordering of the events, such that a *cause*-event comes before the *effect*-event. (P. 340)
    - Systems in which events are *causally ordered* are “causally consistent” (e.g. snapshot isolation)
    - However, *causal order* != *total order*. In a total order, you can compare *any* two events (of the set of all events) and therefore bring them into a sequence. With causal order, however, you only have kind of “tuples” that causally order *two specific* events. Mathematically-speaking, causal order is a “*partial* order”. (P. 341)
    - Linearizable systems require *total* order. Any system that is linearizable also will preserve causality correctly. (P. 342)
    - *Causally-consistent* systems are a popular choice because they don’t have the performance/availability-penalties of *linearizable* systems, but still offer the strongest possible consistency model that does not slow down due to network delays and remains available during network partitions. Causally-consistent systems are like a “middle-ground” between weak-consistency systems and linearizable systems.
    - The book (written in 2017) claims that, at the time, not many production-grade implementations of causally-consistent systems exist yet.
- Total Order broadcast (TOB): the basic idea of TOB is that linearizable systems do not only need to have a total order, but you also need to know *when* this total order is actually “finalized”. (P. 348). For instance, there are other, cheap ways (in terms of implementation/storage complexity, being cheaper than Version Vectors) such as *Lamport timestamps* (described on P. 346), which can also provide a “total order” for certain data structures (such as monotonically-increasing counters). But with such approaches, the total order only emerges once you have collected *all* read/write-operations in existence. In practice, this is too late - you need a finalized total order in near real-time. Therefore, the basic definition of TOB is that it is a protocol for exchanging messages between nodes.
    - TOB requirements:
        - Reliable delivery: no messages are lost - if a message is delivered to one node, it must be delivered to all nodes

        - Totally-ordered delivery: messages are delivered to every node in the same order
    - Consensus services such as ZooKeeper or etcd implement TOB (P. 349)
    - TOB is asynchronous: you do not know *when* a message is delivered. (P. 350) In contrast, linearizability is a *recency guarantee*. However, you can implement linearizability on top of a TOB mechanism/implementation. For instance, to implement a uniqueness constraint, a node writes a message <node ID, payload> to the TOB log and reads the log back - if its message appears in the log as first message with its own node ID and payload, the write is considered to be successful.

Distributed transactions and consensus

- On the surface, *consensus* seems like an easy-to-understand concept: you want several nodes to agree on something. However, there are many subtleties to this topic. (P. 352)
    - A formal definition is: one or more nodes *propose* values, and the consensus algorithm *decides* which one of these values should be used by everyone. (P. 364). The algorithm must satisfy the following properties (P. 365):
        - Uniform agreement: after the algorithm is done, every node decides on the same outcome
        - Integrity: once a value has been decided, it is not changed anymore
        - Validity: you can only decide for a value v if v has also been proposed
        - Termination: the algorithm must make progress - if some nodes fail, the other nodes must still reach a decision (using a majority) - thus there is a limit to the number of node failures the system can tolerate (fewer than half of the nodes may crash)
- Term definition “atomic commit”: when multiple nodes agree on the outcome of a transaction (commit vs. rollback). (P. 353)
- Research background: “the impossibility of consensus” (P. 353): the FLP impossibility theorem proves that there is no algorithm that can reach consensus, if there is the risk that a node may crash. In practice, nodes can crash, so it shouldn’t be possible to build consensus algorithms, right?
    - Wrong: the system model assumed in the FLP proof is a very restricted system that assumes a deterministic algorithm that is not allowed to use any clocks or timeouts. But in practice we can use those, and identify nodes that we *suspect* to have crashed. This lets us solve the consensus problem in practice, even though sometimes our suspicion about the node-lifeliness may be wrong.

2-Phase-Commit (2PC)

- 2PC is a consensus algorithm. It is used in many (traditional relational) DB products. Note that there are other algorithms based on e.g. Raft or Paxos, presented below, which have better fault-tolerance than 2PC.
- In the *atomic commit* problem, you must ensure that each node only commits once it is certain that all other nodes involved in the transaction will also commit. (P. 355)

- 2PC has a new component, the *coordinator* (a.k.a. *transaction manager*). It may be a separate service, but often it's just a library embedded into the application that wants to start a transaction.
- 2PC algorithm: (P. 356)
    - When the application wants to commit, the coordinator begins phase 1: it sends a *prepare* message to all nodes. If a node answers with "yes", it promises that it will be ready to commit the transaction in the 2nd phase (e.g. blocking any other transactions that would cause conflicts)
    - Phase 2: If all nodes replied with "yes", the coordinator sends out a *commit* message to all nodes. If one or more nodes replied "no", an *abort* message is sent to all nodes.
- Note: verbose details of the 2PC algorithm are found on P. 357
- Problems:
    - If the coordinator fails after it sent the *prepare* message, the whole system can become stuck, because the nodes must wait for a *commit* or *abort* message from the coordinator. (P. 358)
    - The coordinator itself is also *stateful*, it must reliably store enough data to be able to resume after a crash, e.g. the node responses it received for a *prepare* message. (P. 357) It also makes the coordinator a single point of failure (P. 363)
- There is a 3PC algorithm but it assumes networks with upper bounds on the delay for the processing in the nodes and for the network. (P. 359)

Fault-tolerant consensus algorithms

- 2PC is not very fault-tolerant. There are better alternatives, such as Viewstamped Replication (VSR), Paxos, Zab, or Raft (P. 366)
- Most of these algorithms don't work as described above in the abstract system model (where nodes *propose* values and then *decide* on one value). Instead, they decide on a *sequence* of values, which makes them TOB algorithms. (P. 366)
- Projects like ZooKeeper or etcd use such algorithms (like Raft) under the hood. They often have DB-like APIs, but they are not supposed to be used by you as an application developer directly. ZooKeeper etc. are designed to only store *small* amounts of data that fit into memory, and they have only a small number of nodes (to make consensus decisions more efficiently, e.g. 3, 5, or 7 nodes) - the data is replicated among all those nodes. (P. 370)
    - Your actual application (e.g. Kafka) can have many nodes (more than 7), and these nodes are then clients of one of the ZooKeeper nodes. In other words, you are outsourcing the coordination work to services like ZooKeeper. (P. 372)
    - Service discovery is another common use case for tools like ZooKeeper (e.g. looking up the (IP) address of a service - although there are often better (more performant) solutions for service discovery.
    - Membership service: keeping track of a set of nodes that is considered to be alive. Of course it can still happen that this list is not truly correct, e.g. if a node was prematurely declared dead.

Distributed transactions in practice

- Real-world distributed transactions have slower performance, e.g. MySQL becomes 10 slower, due to the additional file system sync calls and message round trip times. (P. 360)
- XA transactions: distributed transactions are typically done within a (homogeneous) DBMS/software (where all nodes run the same software). But sometimes you also want distributed transactions (atomic commits) in a *heterogeneous* system that consists of different applications, e.g. DBs or message brokers. There is the "X/Open XA" standard (eXtended Architecture) that implements 2PC transactions across heterogeneous technologies, with wide adoption, e.g. in relational DBs and message brokers. (P. 361) Note that XA needs to settle on the lowest common denominator of the features of all participating systems, and thus it cannot detect some problematic conditions, such as deadlocks (P. 364).

Interlude: Derived Data

- The next chapters look at how you can integrate different kinds of data systems which make up an entire distributed system. The different data systems are each optimized for different use cases (e.g. caches, indices, data stores, analytics systems, etc.), which is why you need to combine multiple ones (P. 385)
- The data systems can be categorized into 2 categories (P. 386):
    - Systems of record: they are the (single) source of truth which holds the authoritative version of the data. Data is typically written to this system first, often in normalized form.
    - Derived data systems: stores derived data (often denormalized) that has been transformed in some way, often from the *system of record*. The trade-off here is that we waste (disk) space (due to the copies) in order to get better performance.
- Most data system implementations (e.g. DB engines, query languages, etc.) are not inherently a system of record or a derived data system - they are just tools. It depends on how you choose to use them.
- There are three basic kinds of systems: online and offline systems, and a mixture of them (P. 389):
    - **Online systems** generally assume that a human triggered a request and is eagerly waiting for the response. Thus, the response time should be as low as possible, and (high) availability is very important.
    - **Offline systems**, a.k.a. batch processing systems, take large amounts of "bounded" input data (whose size is known at the start of the processing, and it does not change during processing), and produce some output data. The jobs can take a long time (even days), and there are no users *eagerly* waiting for the result. Here, *throughput* is important (processed items per some fixed time period), not so much the response time or availability.
    - **Nearline (Near-real-time) / stream processing systems**: they are somewhere between online and offline systems. Like batch processing jobs, but they usually

don’t take as long as batch jobs, and they are triggered by events of the system.

10. Batch processing

- The chapter looks at different kinds of batch processing systems:
    - UNIX tools
    - MapReduce
    - Dataflow engines
    - Graph processing engines
- Any kind of batch processing framework needs to solve the problems of *partitioning* (distributing data and computation to multiple nodes) and *fault tolerance* (recovery when individual tasks fail). (P. 429)

UNIX tools

- Basic idea: connect the inputs and outputs of several specialized tools (that do one thing well, e.g. sort, uniq, head, cat, awk, …) together, using pipes or redirect operators (< or >). (P. 394)
- Many of these tools are highly optimized w.r.t. memory use. That is, tools like `sort` transparently handle larger-than-memory datasets by temporarily writing data to disk. (P. 394).
- The uniform interface that allows passing data through these tools are simply file descriptors that contain streams of (uninterpreted) byte arrays. Since in UNIX many things are represented as files, the source/destination can also be *devices* or network *connections*. (P. 395) Often, tools support both loading from a specific file (path), or using stdin/stdout. The tools themselves are not aware of the complete chain of tools you are running, nor do they need to be.

MapReduce and HDFS

- MapReduce is similar to UNIX tools, but it distributes the computation across (potentially) thousands of nodes. (P. 397)
- Whereas UNIX tools use files or stdin/stdout as input and output, MapReduce jobs read and write files on a distributed file system. Hadoop and HDFS is a very popular implementation, it is an OSS re-implementation of Google FileSystem GFS. Various other distributed file systems exist, e.g. GlusterFS. (P. 398)
- Unlike HPC systems, MapReduce nodes do not need any special prerequisites. They only require a (TCP) connection to the other nodes.
- HDFS details: there is a central *NameNode* server that tracks the HDFS nodes and which file blocks are stored on which node, and of course many (file system) nodes. On each node there is a HDFS daemon process that offers an API to retrieve or store files on that node. To achieve HA, file blocks are replicated to multiple nodes. (P. 398)
- You need to implement two functions:
    - Mapper: it is called once per input record. It needs to extract and return 0-n key-

value pairs.
    - Reducer: The MapReduce framework collects all k-v pairs produced by all mappers, and calls the reducer function with an iterator over the collection of values of a specific key. From those, the reducer produces some output records.
- Basic job execution flow (P. 399/400):
    - 1) The MapReduce framework has an input format parser that parses input files stored on the distributed FS into input records. You typically provide a directory path on the distributed FS as input, all its contained files are processed.
    - 2) The MapReduce framework calls your mapper function which produces k-v pairs.
    - 3) MapReduce framework stores the k-v pairs of step 2 in a special way (using techniques similar to SSTables and LSM trees, see chapter 3 -> storage engine), resulting in pre-sorted files, which the MapReduce framework then efficiently feeds to the reducers.
    - 4) The MapReduce framework calls the reducer functions with sorted k-v pairs.
- As a user of MapReduce, you do not need to worry about the parallelization details. The mappers/reducers don't need to know where their input comes from or goes to (the MapReduce framework typically transports the code to where the data is, e.g. shipping JAR files, which is faster than sending data to the code). (P. 401)
- MapReduce *workflows*: because any individual MapReduce job can only do one round of sorting, people often chain multiple jobs together, which is called a *workflow*. However, MapReduce does not explicitly support workflows, which is why people create individual jobs, and set the output directory of one job to be the input directory of the next job. Various workflow schedulers are used in practice, e.g. Apache Airflow, or Spotify's Luigi. (P. 402) Because it is difficult to write complex multi-job MapReduce jobs, various abstraction-tools have emerged which are (nowadays) historic, such as Apache Pig, Apache Crunch, or Cascading (none of them have been maintained since 2018).
- Common use cases of MapReduce jobs (P. 411/412):
    - Build search indices
    - Build a database, e.g. for a recommender system
- The Avro format (see chapter 4) is often used in practice to store data in HDFS. (P. 414)
- In a sense, the HDFS is still like a "data lake" where MapReduce jobs convert unstructured data to structured data. (P. 415)
- The Hadoop ecosystem offers many more data processing models than MapReduce, e.g. SQL.

Other batch processing tools

- From a performance perspective, the main problem of MapReduce workflows with many jobs is that it always "materializes" / persists the intermediate results of each job. This takes time (also because the result data is *replicated* to several nodes), and could be considered a waste of storage. Also, a second job must always wait until the first job has *completely* finished (having written its intermediate output). (P. 420)
- *Dataflow engines* such as Apache Tez, Apache Spark or Apache Flink, which allow you

to model the flow of data through several processing stages, try to be better. Similar to MapReduce, they repeatedly call user-defined functions ("operators") which process one record at a time, and take care of the parallelization for you. However, these operator-functions are not the alternating map() and reduce() functions. Various optimizations are in place that speed up job execution significantly. Consequently, MapReduce-style workflows can be executed much faster with such dataflow engines than with e.g. Hadoop itself. (P. 421)
- For graph processing, there are dedicated algorithms and tools (e.g. Spark's GraphX API, or Apache Flink's Graph API called "Gelly"). They need to solve the problem (of answering a, say, GraphQL query) *iteratively*, because many problems require propagating information from one vertex to the other (think of queries where the scheduler needs to follow a series of edges). (P. 424/425)

11. Stream processing

- In practice, lots of data arrives as (unbounded) streams. Trying to solve them via *batch processing* (chapter 10) would be a bad idea. You would have to artificially divide the work into chunks (e.g. a day's worth of data), but then you would get the output quite delayed (which might be too slow for some users). (P. 439)
- What batch processing calls a "record" is called an "event" in stream processing. Events are a small, self-contained object with details of something that happened, usually including a wall clock timestamp.

Transmission of event streams

- "File names" in batch processing are the same as "topics" or "streams" in stream processing. (P. 441)
- Events could be encoded in various formats, e.g. strings, JSON, or in binary form. It could be appended to a file, stored in a table, or in a NoSQL datastore as a document. Events are transmitted over the network to other nodes for processing. Events are created by *producers* and read by *consumers/recipients/subscribers*. (P. 440). This usually happens (more efficiently) by *pushing* events, rather than pulling them (P. 441).
- This *publish/subscribe* model has a few interesting challenges (P. 441/442):
    - What if producers send messages faster than the consumer can handle them? Drop messages, buffer them in a queue, or apply backpressure. When buffering, how to handle an ever-growing queue? Persist messages from memory to disk? What about performance?
    - What if a node crashes or goes offline temporarily? Are messages lost (and could your application deal with that loss)? Achieving durability (e.g. by *immediately*) writing messages to disk does have a performance penalty.
- This *publish/subscribe* model can be implemented in various ways:
    - Via *direct messaging* between producers and consumers (P. 442): e.g. via UDP multicast, or via TCP or IP multicast (as done by brokerless libs such as ZeroMQ or nanomsg). Also, producers could connect directly to consumers (or vice

versa), such as done with webhooks.
        - The main caveat is that for these systems to work, producers and consumers need to be constantly online, or they will miss messages. (P. 443)
    - Via *message brokers* (P. 443) which run as servers to which consumers connect. Messages are put on a (usually *unbounded*) queue, thus consumers no longer need to be always online. Some brokers *persist* messages to disk (so that the broker can recover from a crash), some brokers only keep messages in memory. There are two basic variants of brokers:
        - **"Transient" brokers** that drop messages once they have been delivered (e.g. JMS/AMQP/etc.). Their caveat is that new subscribers will only get those messages that have been published since they joined. They are a good choice if you need *load balancing* message distribution (see bullet point below), e.g. because processing a single message is expensive, and assuming that ordering of messages is not important.
        - **Log-based brokers** that persist all events to disk - they also allow subscribers to process "old" messages published before they connected.
- Distributing events to multiple consumers subscribing to the same topic (P. 444): there are two basic approaches:
    - 1. *Load balancing*: events are evenly distributed to the consumers, e.g. in round-robin style.
    - 2. *Fan out*: each event is delivered to every consumer.
- Improving fault tolerance of message/event delivery (P. 445): brokers generally make use of acknowledgements and message-redelivery to tolerate crashing consumers. After a client has taken a message from the broker's queue, it must explicitly tell the broker that it has processed the message successfully (the "acknowledgement"). If this does not happen "in time" (using some timeout), the broker will deliver the message again (e.g. to other consumers, when using the *load balancing* approach). This can mean that you no longer get messages delivered to consumers in the same order as they were published!
- Details about *log-based* brokers (such as Kafka or Amazon Kinesis streams):
    - They are like a "hybrid" of databases (which let clients discover "old" data that was stored in the DB some time in the past, but DB implementations are not performant regarding subscriptions/pushing messages), and JMS-style message brokers that are transient/forgetting old events, but are efficient in their publish-delivery mechanism. (P. 447)
    - General idea: producers append messages to the end of a log, consumers read all messages until the log ends, and from then on they wait for notifications.
    - Scaling happens via *partitioning* (as discussed in chapter 6) of some form (the book does not give concrete examples), such that a specific *topic* is broken down into multiple partitions. In each partition, the broker assigns a monotonically-increasing counter for each message, a.k.a. *Offset*.
        - There are no ordering guarantees across different partitions!
    - Fault tolerance is achieved by replicating partitions to different nodes.

    - Achieving the *load balancing* message distribution mode (see above) is more tricky with log-based brokers, especially when using *partitions* (P. 448): you would usually assign entire partitions to consumers. But this has problems, e.g. a slow consumer would cause the entire system to slow down, because with this approach you don't let other (faster) consumers deal with the remaining messages stored in a partition.
    - Issue with fault tolerance and multiple partitions: it is not enough that a consumer internally maintains its offset, because when a consumer crashes, the broker would not know what to tell a new consumer (who takes over the work of the crashed one) regarding the offset from which it should continue. (P. 449)
    - In practice, log-based brokers cannot really offer *un*bounded queues, but they have to discard very old data once they run out of disk space. In practice, this would happen usually because a consumer is too slow or has crashed for too long, which is something you can monitor and write alerts for. (P. 450)

Bringing message broker features into (traditional) DBs

- Databases internally also have logs that contain changes, the write-ahead logs and replication logs. But they are not exposed via a public API (P. 454).
- *Change data capture* (CDC) is a topic of increasing interest: it means that you can observe all changes written to the DB, in order to apply them to some other system (e.g. an external index). Instead of all interested clients using a DB's "trigger" mechanism, you can instead have a single producer that is triggered, which then publishes the change events via a log-based message broker. (P. 455)
    - There are dedicated 3rd party implementations for traditional DBs like MySQL, e.g. Maxwell or Debezium, but also many DB vendors have Apache Kafka connectors. (P. 455)
    - There are also some DBs offering such CDC APIs *natively* (first class support), e.g. CouchDB or Firebase (P. 456)
    - Some of these tools also maintain the ability to retrieve a snapshot of a given point of time, allowing you to retrieve change events that happened *since* this snapshot has been created. (P. 455)
- A related approach to CDC is *Event sourcing* (from the Domain-Driven-Design community) (P. 457)
    - It applies the idea of capturing change events at a high level of abstraction (whereas CDC works at a low-level). What CDC essentially does is that the DB works on *mutable* data, and the DB engine extracts change events at the low level. With Event sourcing, the *application* already works at the event level - it creates and stores immutable event objects in the DB. Event sourcing records user actions themselves, rather than recording the *effects* of these actions.
    - Although there are specialized DB engines for event sourcing (e.g. Event Store), you can also use conventional DBs or log-based message brokers to build applications in this style.
    - However, reading/writing *only* these events would not work - applications and

users still (also) need a current state / snapshot that accumulates all events. However, this is done for performance reasons (to get a fast response), not for storage reasons - Event sourcing-based applications usually do want to keep all events for a very long time, for auditing etc.
        - Note that there is additional complexity due to the fact that the state/snapshot view needs to be updated *in lock-step* (or "in sync") with the events. This could be achieved via distributed transactions (that e.g. insert the event into the event store and also make sure that the view is updated), or with other tricks (e.g. having views be rebuilt asynchronously in the background e.g. every 10 minutes, and whenever we render a view, we dynamically compute the view, using the (slightly outdated) view and applying only the events from the event DB of the last 10 minutes.
- Advantages of immutable events (P. 461):
    - Auditability
    - Easier to debug code, because you can replay events
    - Easier to recover from errors (e.g. in a production system)
    - Higher degree of information: e.g. if 2 actions would cancel each other out (in the snapshot / state), it may still be useful to know about these 2 actions, e.g. for analytics
    - Ability to implement additional features with additional (state-based) views, in parallel to the production system (not negatively affecting it, e.g. not having to do schema migrations). In general, schema design becomes much more flexible, as you can always rebuild your DB (with a different schema) from the event logs.
- Disadvantages of immutable events (P. 463):
    - An ever-growing history of events will grow very large. Storage will become expensive, and the performance of log compaction and garbage collection may become degraded.
    - Sometimes you are required to delete data, e.g. for administrative reasons (GDPR etc.)

Processing of event streams

- There are a few options to process events:
    - 1) Write the data contained in the events to a different storage system, e.g. a DB, a cache, a (search) index, …. from which other clients can query the data.
    - 2) Push event data to users/humans, e.g. via mobile push notifications or via email
    - 3) Process one or more input streams to create one or more output streams, which in turn are then processed using options 1-3 again. **The rest of this section focuses on option 3.**.
- Uses of stream processing (P. 465-468):
    - *Complex event processing* (CEP): applies pattern-matching to the incoming event stream. You use some high-level declarative query language (e.g. SQL) to define the pattern. Once such a sequence is matched, the processing engine

produces a (complex) output event that contains details of the event pattern that was detected. There are a few commercial products available., e.g. IBM InfoSphere Streams, or SQLstream.
    - Stream analytics: rather than pattern matching, this usage mode computes aggregations and/or statistical metrics over a large number of events. E.g. the frequency of events, or a rolling average of a value, etc. Example implementations are Apache Storm, Spark Streaming, Flink, or Kafka Streams. Note that the boundary between CEP and stream analytics is blurry.
    - Maintaining materialized views: events are applied to a mutable state, e.g. caches, data warehouses/DBs, search indices, etc.
    - Searching on streams: can be broken down into “after-the-fact” search (when you want to search on past events) and “streaming search” (applying a search to each (future) event as it is added to the stream). It is possible to optimize the search, which is necessary, because it would be computationally expensive to brute-force-style apply *every* query to every document. Instead, you can also index the queries themselves. That index is a dictionary mapping from the terms (contained in a query) to the corresponding query: Whenever a document/event comes in, you turn it into a *query* that ORs all the terms in that document, and then use the query-index to find out which of the many existing queries you should actually run against that document. Details are available here.
- Handling *time* in stream processing is rather difficult:
    - There are different time stamps: event creation (timestamp created by the machine producing the event), event delivery (timestamp created by e.g. the broker), time of processing (timestamp from the processor’s local clock). (P. 469)
        - The clocks producing these timestamps are often not synchronized!
    - The time between *event creation* and *event delivery* may be rather large, e.g. due to network delays, or temporarily being offline. Also, the delivery may be out of order.
    - Answering a query such as “give me the rate of HTTP requests to my server averaged over the last 5 minutes” is difficult, because the result is constantly “in flux”, assuming that not all events of that “last 5 minutes” time window have arrived yet. These delayed events are called *straggler* events. You can e.g. handle them by ignoring them, but still tracking a metric of how many you have ignored, and alert you if the metric becomes bad. Or you can publish “corrections” of the search result. (P. 470)
- *Join queries* on streams: there are 3 different kinds of joins on streams (P. 473-475):
    - **Stream-stream joins**: the join operator is given 2 streams and searches for related events that occur within a time window (that you define). An example. One stream contains “user did a search” events, the other stream is “user clicked on search result”. The join operator locally maintains state for the events occurring within the predefined time window, usually in the form of a search index.
        - These 2 input streams might even be the *same* stream (“self-join”).
    - **Stream-table joins**: one input is a stream, the other input (“table”) is a

materialized view of a database of which the join operator keeps a *local* copy (for performance reasons - querying a remote DB on each incoming event would be too expensive). The join operator keeps this local copy of the materialized view up-to-date by subscribing to the DB's changelog (see "CDC" above). For each incoming event of the stream, the join operator queries the materialized view of the local DB copy and outputs an *enriched* event.
    - **Table-table joins**: both inputs are materialized-view-tables (as described above). The join operator creates a 3rd materialized view that is a (SQL-style) JOIN of the 2 incoming materialized view tables, producing an output stream of the changes to that 3rd materialized-view.
- Handling fault tolerance:
    - The general goal of fault tolerance is that the output should be the same as if nothing had gone wrong, even though in fact things have gone wrong. It should appear as if every event was processed exactly once. (P. 476)
    - In case of MapReduce-style *batch processing* (prev. chapter) this is easier to solve than for *stream processing*, because the input was bounded, and you would simply retry failed batch jobs (of intermediate stages). In stream processing, some tools (such as Spark Streaming) perform "micro-batching", breaking the stream up into smaller blocks, treating each block like a miniature batch process. Other solutions, e.g. Apache Flink, periodically generates rolling checkpoints of state, writing them to durable storage. If a stream operator crashes, it restarts from the most recent snapshot.

12. The future of data systems

- This is a meta-level chapter which reviews the design of distributed systems (particularly DB-like systems vs. data flow systems), and also goes into cultural/societal aspects.
- Given that systems are often made of several specialized components (DBs, message brokers, etc.), the analysis of the *data flow* is very important. Typically, some external actor (a machine or a user) initiates an action (e.g. a button click in your application's web UI), and this event then flows through the different components of our system. (P. 491)
    - One solution is to use distributed transactions, especially if several things (all) need to happen in parallel (e.g. manipulating a DB *and* writing data to a search index). The distributed transaction is needed because no individual component is "in charge" of keeping consistency, thus, the distributed transaction ensures it. The disadvantage is the complexity and lower throughput of distributed transactions (assuming you require strong consistency).
    - If you do need strong consistency (in terms of *data integrity*), but do not require an upper bound (regarding *timeliness*) after which a transaction is definitely finished, a better alternative is to use stream-processing systems (see previous chapter). You write the events generated by the external actor to a single source of truth (e.g. a *log-based broker*), which provides the total order of all events. Other components (e.g. the search index or a DB) subscribe to the log, and do

their work. You insert uniquely-generated end-to-end correlation IDs (already generated in your application, or at least in the message broker) to make operations idempotent. See also P. 523 for details. The advantage of this approach is that it can perform much better in terms of throughput and operational stability (P. 526)

- When using stream processing, you can also generate, store and process *events* whenever a user or external system reads from your system (that is: events are not just for *write*-operations). The stream processor can then both process read and write events, by writing the result to output streams. (P. 514)
    - When both read and write events are processed by the same stream processor, you can do a *stream-table join* (see above) between the stream of read queries and the database (that accumulates the write-ops).
    - Having read events improves the ability of user tracking.
    - However, storing the (probably large) volume of read events is costly and challenging in terms of handling the high throughput.
- Getting a completely *correct* application in practice is almost impossible. “Correctness” means that you have well-defined and well-understood semantics of the behavior of your application, even when facing various kinds of faults. However, it is already difficult to judge which transaction isolation level is the right one for our application. Things might look good during the initial tests, but subtle (hard-to-reproduce) problems still come up during operations of the production system. And you cannot even fully trust the claims of the vendors of DBs (or brokers, etc.), as e.g. the Jepsen DB benchmarks demonstrate, which often find flaws in DBs in the presence of network problems or crashes. And even if a DB was bug-free, your application code might use it in an incorrect way (doing the wrong calls, or configuring it wrong). (P. 515)
- A very common requirement is “exactly once” semantics, which guarantees that events are only processed exactly once, which is difficult because information passes through different nodes with (possibly) flaky connectivity where you cannot rely on requests or responses to be delivered as promised. While an *atomic commit* protocol can achieve this exactly-once requirement, another great way is to make operations *idempotent*, e.g. by using end-to-end operation identifiers. Exemplary, in a web app, you would generate a unique ID already in the app’s frontend, and that ID is passed on to all the other (distributed) components in your system. For instance, in a relational DB, you could have a “requests” table with a single “requestID” column whose values must be unique. In your transaction, you always first try to insert the request ID (submitted by the client) before doing any other table manipulations, thus failing early if the request ID was already committed as part of a previous transaction. (P. 518)
- Loosening constraints (P. 526-528): often the constraints of strong consistency are not really called for (e.g. having a unique username, or avoiding overbooking of a resource such as a hotel room). It is usually easier to allow for inconsistencies to happen and to compensate at a later point of time, e.g. by giving a gift or voucher to the end-user for their trouble due to a cancellation. This is particularly true in those cases where the business must have such “apology-mechanisms” in place anyway. Exemplary, an online shop may have miscounted their inventory, having sold you goods it doesn’t even have.

Or due to higher power (bad weather) a flight has to be canceled. Thus, if you have such an apology-mechanism in place already anyway (on the business level), it makes little sense to require strong consistency on the technological level.
    - Also, overbooking is an often-used (desirable) trick on the business-level anyway, e.g. overbooking hotel rooms because the business knows that there is a certain rate of (last-minute) cancellations anyway. If push comes to shove, they can offer you a room in the hotel next door.
    - Eventually, you still have to validate the consistency of your data (doing a delayed conflict resolution), but it can happen much later than the point of the event-generation (when the original write-op happened).
    - Having strong constraints in place only *reduces* the number of apologies you have to make, but that also reduces the performance and availability of your system, which in turn may *increase* the number of apologies you have to make (for the system outages).
- As discussed above (P. 515), you should expect bugs even in “battle-tested” DBs. A good mitigation strategy is to regularly/continuously audit the *integrity* of your data. (P. 530). This means to read the data and check that it makes sense. It also means to verify that backups are consistent, and that you can successfully restore them.
    - Some systems, such as HDFS or S3, have such data auditing integrated - they are assuming a certain failure rate of the underlying disks.
    - Doing data integrity auditing does cost some money, but it may save much more money which you would otherwise lose in a production incident that takes you a long time to fix, because you are having problems with the data restore process. Doing auditing *continuously* also helps you discover (and fix) bugs in your code more quickly (e.g. if the bug introduces integrity problems).
- Book closes with a lengthy section about responsibility we as developers have, touching on the ethics and rights of the end-users using our system (not summarized here).