Designing Data-Intensive Applications

Reliable, scalable, and maintainable applications

A data-intensive application is typically built from standard building blocks. They usually need to: * Store data (*databases*) * Speed up reads (*caches*) * Search data (*search indexes*) * Send a message to another process asynchronously (*stream processing*) * Periodically crunch data (*batch processing*)

- **Reliability**. To work *correctly* even in the face of *adversity*.
- **Scalability**. Reasonable ways of dealing with growth.
- **Maintainability**. Be able to work on it *productively*.

Reliability

Typical expectations: * Application performs the function the user expected * Tolerate the user making mistakes * Its performance is good * The system prevents abuse

Systems that anticipate faults and can cope with them are called *fault-tolerant* or *resilient*.

A fault is usually defined as one component of the system deviating from its spec, whereas *failure* is when the system as a whole stops providing the required service to the user.

You should generally **prefer tolerating faults over preventing faults**.

- **Hardware faults**. Until recently redundancy of hardware components was sufficient for most applications. As data volumes increase, more applications use a larger number of machines, proportionally increasing the rate of hardware faults. **There is a move towards systems that tolerate the loss of entire machines**. A system that tolerates machine failure can be patched one node at a time, without downtime of the entire system (*rolling upgrade*).
- **Software errors**. It is unlikely that a large number of hardware components will fail at the same time. Software errors are a systematic error within the system, they tend to cause many more system failures than uncorrelated hardware faults.
- **Human errors**. Humans are known to be unreliable. Configuration errors by operators are a leading cause of outages. You can make systems more reliable:

- Minimising the opportunities for error, peg: with admin interfaces that make easy to do the "right thing" and discourage the "wrong thing".
- Provide fully featured non-production *sandbox* environments where people can explore and experiment safely.
- Automated testing.
- Quick and easy recovery from human error, fast to rollback configuration changes, roll out new code gradually and tools to recompute data.
- Set up detailed and clear monitoring, such as performance metrics and error rates (*telemetry*).
- Implement good management practices and training.

Scalability

This is how do we cope with increased load. We need to succinctly describe the current load on the system; only then we can discuss growth questions.

Twitter example

Twitter main operations - Post tweet: a user can publish a new message to their followers (4.6k req/sec, over 12k req/sec peak) - Home timeline: a user can view tweets posted by the people they follow (300k req/sec)

Two ways of implementing those operations: 1. Posting a tweet simply inserts the new tweet into a global collection of tweets. When a user requests their home timeline, look up all the people they follow, find all the tweets for those users, and merge them (sorted by time). This could be done with a SQL `JOIN`. 2. Maintain a cache for each user's home timeline. When a user *posts a tweet*, look up all the people who follow that user, and insert the new tweet into each of their home timeline caches.

Approach 1, systems struggle to keep up with the load of home timeline queries. So the company switched to approach 2. The average rate of published tweets is almost two orders of magnitude lower than the rate of home timeline reads.

Downside of approach 2 is that posting a tweet now requires a lot of extra work. Some users have over 30 million followers. A single tweet may result in over 30 million writes to home timelines.

Twitter moved to an hybrid of both approaches. Tweets continue to be fanned out to home timelines but a small number of users with a very large number of followers are fetched separately and merged with that user's home timeline when it is read, like in approach 1.

Describing performance

What happens when the load increases: * How is the performance affected? * How much do you need to increase your resources?

In a batch processing system such as Hadoop, we usually care about *throughput*, or the number of records we can process per second.

> **Latency and response time**
>
> The response time is what the client sees. Latency is the duration that a request is waiting to be handled.

It's common to see the *average* response time of a service reported. However, the mean is not very good metric if you want to know your "typical" response time, it does not tell you how many users actually experienced that delay.

Better to use percentiles. * *Median* (*50th percentile* or *p50*). Half of user requests are served in less than the median response time, and the other half take longer than the median * Percentiles *95th*, *99th* and *99.9th* (*p95*, *p99* and *p999*) are good to figure out how bad your outliners are.

Amazon describes response time requirements for internal services in terms of the 99.9th percentile because the customers with the slowest requests are often those who have the most data. The most valuable customers.

On the other hand, optimising for the 99.99th percentile would be too expensive.

Service level objectives (SLOs) and *service level agreements* (SLAs) are contracts that define the expected performance and availability of a service. An SLA may state the median response time to be less than 200ms and a 99th percentile under 1s. **These metrics set expectations for clients of the service and allow customers to demand a refund if the SLA is not met.**

Queueing delays often account for large part of the response times at high percentiles. **It is important to measure times on the client side.**

When generating load artificially, the client needs to keep sending requests independently of the response time.

> **Percentiles in practice**
>
> Calls in parallel, the end-user request still needs to wait for the slowest of the parallel calls to complete. The chance of getting a slow call increases if an end-user request requires multiple backend calls.

Approaches for coping with load

- *Scaling up* or *vertical scaling*: Moving to a more powerful machine
- *Scaling out* or *horizontal scaling*. Distributing the load across multiple smaller machines.
- *Elastic* systems: Automatically add computing resources when detected load increase. Quite useful if load is unpredictable.

Distributing stateless services across multiple machines is fairly straightforward. Taking stateful data systems from a single node to a distributed setup can introduce a lot of complexity. Until recently it was common wisdom to keep your database on a single node.

Maintainability

The majority of the cost of software is in its ongoing maintenance. There are three design principles for software systems: * **Operability**. Make it easy for operation teams to keep the system running. * **Simplicity**. Easy for new engineers to understand the system by removing as much complexity as possible. * **Evolvability**. Make it easy for engineers to make changes to the system in the future.

Operability: making life easy for operations

A good operations team is responsible for * Monitoring and quickly restoring service if it goes into bad state * Tracking down the cause of problems * Keeping software and platforms up to date * Keeping tabs on how different systems affect each other * Anticipating future problems * Establishing good practices and tools for development * Perform complex maintenance tasks, like platform migration * Maintaining the security of the system * Defining processes that make operations predictable * Preserving the organisation's knowledge about the system

Good operability means making routine tasks easy.

Simplicity: managing complexity

When complexity makes maintenance hard, budget and schedules are often overrun. There is a greater risk of introducing bugs.

Making a system simpler means removing *accidental* complexity, as non inherent in the problem that the software solves (as seen by users).

One of the best tools we have for removing accidental complexity is *abstraction* that hides the implementation details behind clean and simple to understand APIs and facades.

Evolvability: making change easy

Agile working patterns provide a framework for adapting to change.

- *Functional requirements*: what the application should do
- *Nonfunctional requirements*: general properties like security, reliability, compliance, scalability, compatibility and maintainability.

Data models and query language

Most applications are built by layering one data model on top of another. Each layer hides the complexity of the layers below by providing a clean data model. These abstractions allow different groups of people to work effectively.

Relational model vs document model

The roots of relational databases lie in *business data processing*, *transaction processing* and *batch processing*.

The goal was to hide the implementation details behind a cleaner interface.

Not Only SQL has a few driving forces: * Greater scalability * preference for free and open source software * Specialised query optimisations * Desire for a more dynamic and expressive data model

With a SQL model, if data is stored in a relational tables, an awkward translation layer is translated, this is called *impedance mismatch*.

JSON model reduces the impedance mismatch and the lack of schema is often cited as an advantage.

JSON representation has better *locality* than the multi-table SQL schema. All the relevant information is in one place, and one query is sufficient.

In relational databases, it's normal to refer to rows in other tables by ID, because joins are easy. In document databases, joins are not needed for one-to-many tree structures, and support for joins is often weak.

If the database itself does not support joins, you have to emulate a join in application code by making multiple queries.

The most popular database for business data processing in the 1970s was the IBM's *Information Management System* (IMS).

IMS used a *hierarchical model* and like document databases worked well for one-to-many relationships, but it made many-to-,any relationships difficult, and it didn't support joins.

The network model

Standardised by a committee called the Conference on Data Systems Languages (CODASYL) model was a generalisation of the hierarchical model. In the tree structure of the hierarchical model, every record has exactly one parent, while in the network model, a record could have multiple parents.

The links between records are like pointers in a programming language. The only way of accessing a record was to follow a path from a root record called *access path*.

A query in CODASYL was performed by moving a cursor through the database by iterating over a list of records. If you didn't have a path to the data you wanted, you were in a difficult situation as it was difficult to make changes to an application's data model.

The relational model

By contrast, the relational model was a way to lay out all the data in the open" a relation (table) is simply a collection of tuples (rows), and that's it.

The query optimiser automatically decides which parts of the query to execute in which order, and which indexes to use (the access path).

The relational model thus made it much easier to add new features to applications.

The main arguments in favour of the document data model are schema flexibility, better performance due to locality, and sometimes closer data structures to the ones used by the applications. The relation model counters by providing better support for joins, and many-to-one and many-to-many relationships.

If the data in your application has a document-like structure, then it's probably a good idea to use a document model. The relational technique of *shredding* can lead unnecessary complicated application code.

The poor support for joins in document databases may or may not be a problem.

If you application does use many-to-many relationships, the document model becomes less appealing. Joins can be emulated in application code by making multiple requests. Using the document model can lead to significantly more complex application code and worse performance.

Schema flexibility

Most document databases do not enforce any schema on the data in documents. Arbitrary keys and values can be added to a document, when reading, **clients have no guarantees as to what fields the documents may contain.**

Document databases are sometimes called *schemaless*, but maybe a more appropriate term is *schema-on-read*, in contrast to *schema-on-write*.

Schema-on-read is similar to dynamic (runtime) type checking, whereas schema-on-write is similar to static (compile-time) type checking.

The schema-on-read approach if the items on the collection don't have all the same structure (heterogeneous) * Many different types of objects * Data determined by external systems

Data locality for queries

If your application often needs to access the entire document, there is a performance advantage to this *storage locality*.

The database typically needs to load the entire document, even if you access only a small portion of it. On updates, the entire document usually needs to be rewritten, it is recommended that you keep documents fairly small.

Convergence of document and relational databases

PostgreSQL does support JSON documents. RethinkDB supports relational-like joins in its query language and some MongoDB drivers automatically resolve database references. Relational and document databases are becoming more similar over time.

Query languages for data

SQL is a *declarative* query language. In an *imperative language*, you tell the computer to perform certain operations in order.

In a declarative query language you just specify the pattern of the data you want, but not *how* to achieve that goal.

A declarative query language hides implementation details of the database engine, making it possible for the database system to introduce performance improvements without requiring any changes to queries.

Declarative languages often lend themselves to parallel execution while imperative code is very hard to parallelise across multiple cores because it specifies instructions that must be performed in a particular order. Declarative languages specify only the pattern of the results, not the algorithm that is used to determine results.

Declarative queries on the web

In a web browser, using declarative CSS styling is much better than manipulating styles imperatively in JavaScript. Declarative languages like SQL turned out to be much better than imperative query APIs.

MapReduce querying

MapReduce is a programming model for processing large amounts of data in bulk across many machines, popularised by Google.

Mongo offers a MapReduce solution.

```
db.observations.mapReduce(
    function map() { 2
        var year  = this.observationTimestamp.getFullYear();
        var month = this.observationTimestamp.getMonth() + 1;
        emit(year + "-" + month, this.numAnimals); 3
    },
    function reduce(key, values) { 4
        return Array.sum(values); 5
    },
    {
        query: { family: "Sharks" }, 1
        out: "monthlySharkReport" 6
    }
);
```

The `map` and `reduce` functions must be *pure* functions, they cannot perform additional database queries and they must not have any side effects. These restrictions allow the database to run the functions anywhere, in any order, and rerun them on failure.

A usability problem with MapReduce is that you have to write two carefully coordinated functions. A declarative language offers more opportunities for a query optimiser to improve the performance of a query. For there reasons, MongoDB 2.2 added support for a declarative query language called *aggregation pipeline*

```
db.observations.aggregate([
    { $match: { family: "Sharks" } },
    { $group: {
        _id: {
            year:  { $year:  "$observationTimestamp" },
            month: { $month: "$observationTimestamp" }
        },
        totalAnimals: { $sum: "$numAnimals" }
    } }
]);
```

Graph-like data models

If many-to-many relationships are very common in your application, it becomes more natural to start modelling your data as a graph.

A graph consists of *vertices* (*nodes* or *entities*) and *edges* (*relationships* or *arcs*).

Well-known algorithms can operate on these graphs, like the shortest path between two points, or popularity of a web page.

There are several ways of structuring and querying the data. The *property graph* model (implemented by Neo4j, Titan, and Infinite Graph) and the *triple-store* model (implemented by Datomic, AllegroGraph, and others). There are also three declarative query languages for graphs: Cypher, SPARQL, and Datalog.

Property graphs

Each vertex consists of: * Unique identifier * Outgoing edges * Incoming edges * Collection of properties (key-value pairs)

Each edge consists of: * Unique identifier * Vertex at which the edge starts (*tail vertex*) * Vertex at which the edge ends (*head vertex*) * Label to describe the kind of relationship between the two vertices * A collection of properties (key-value pairs)

Graphs provide a great deal of flexibility for data modelling. Graphs are good for evolvability.

- *Cypher* is a declarative language for property graphs created by Neo4j
- Graph queries in SQL. In a relational database, you usually know in advance which joins you need in your query. In a graph query, the number if joins is not fixed in advance. In Cypher `:WITHIN*0...` expresses "follow a `WITHIN` edge, zero or more times" (like the `*` operator in a regular expression). This idea of variable-length traversal paths in a query can be expressed using something called *recursive common table expressions* (the `WITH RECURSIVE` syntax).

Triple-stores and SPARQL

In a triple-store, all information is stored in the form of very simple three-part statements: *subject*, *predicate*, *object* (peg: *Jim*, *likes*, *bananas*). A triple is equivalent to a vertex in graph.

The SPARQL query language

SPARQL is a query language for triple-stores using the RDF data model.

The foundation: Datalog

Datalog provides the foundation that later query languages build upon. Its model is similar to the triple-store model, generalised a bit. Instead of writing a triple (*subject*, *predicate*, *object*), we write as *predicate(subject, object)*.

We define *rules* that tell the database about new predicates and rules can refer to other rules, just like functions can call other functions or recursively call themselves.

Rules can be combined and reused in different queries. It's less convenient for simple one-off queries, but it can cope better if your data is complex.

Storage and retrieval

Databases need to do two things: store the data and give the data back to you.

Data structures that power up your database

Many databases use a *log*, which is append-only data file. Real databases have more issues to deal with tho (concurrency control, reclaiming disk space so the log doesn't grow forever and handling errors and partially written records).

> A *log* is an append-only sequence of records

In order to efficiently find the value for a particular key, we need a different data structure: an *index*. An index is an *additional* structure that is derived from the primary data.

Well-chosen indexes speed up read queries but every index slows down writes. That's why databases don't index everything by default, but require you to choose indexes manually using your knowledge on typical query patterns.

Hash indexes

Key-value stores are quite similar to the *dictionary* type (hash map or hash table).

Let's say our storage consists only of appending to a file. The simplest indexing strategy is to keep an in-memory hash map where every key is mapped to a byte offset in the data file. Whenever you append a new key-value pair to the file, you also update the hash map to reflect the offset of the data you just wrote.

Bitcask (the default storage engine in Riak) does it like that. The only requirement it has is that all the keys fit in the available RAM. Values can use more space than there is available in memory, since they can be loaded from disk.

A storage engine like Bitcask is well suited to situations where the value for each key is updated frequently. There are a lot of writes, but there are too many distinct keys, you have a large number of writes per key, but it's feasible to keep all keys in memory.

As we only ever append to a file, so how do we avoid eventually running out of disk space? **A good solution is to break the log into segments of certain size by closing the segment file when it reaches a certain size, and making subsequent writes to a new segment file. We can then perform *compaction* on these segments.** Compaction means throwing away duplicate keys in the log, and keeping only the most recent update for each key.

We can also merge several segments together at the sae time as performing the compaction. Segments are never modified after they have been written, so the merged segment is written to a new file. Merging and compaction of frozen segments can be done in a background thread. After the merging process is complete, we switch read requests to use the new merged segment instead of the old segments, and the old segment files can simply be deleted.

Each segment now has its own in-memory hash table, mapping keys to file offsets. In order to find a value for a key, we first check the most recent segment hash map; if the key is not present we check the second-most recent segment and so on. The merging process keeps the number of segments small, so lookups don't need to check many hash maps.

Some issues that are important in a real implementation: * File format. It is simpler to use binary format. * Deleting records. Append special deletion record to the data file (*tombstone*) that tells the merging process to discard previous values. * Crash recovery. If restarted, the in-memory hash maps are lost. You can recover from reading each segment but that would take long time. Bitcask speeds up recovery by storing a snapshot of each segment hash map on disk. * Partially written records. The database may crash at any time. Bitcask includes checksums allowing corrupted parts of the log to be detected and ignored. * Concurrency control. As writes are appended to the log in a strictly sequential order, a common implementation is to have a single writer thread. Segments are immutable, so they can be read concurrently by multiple threads.

Append-only design turns out to be good for several reasons: * Appending and segment merging are sequential write operations, much faster than random writes, especially on magnetic spinning-disks. * Concurrency and crash recovery are much simpler. * Merging old segments avoids files getting fragmented over time.

Hash table has its limitations too: * The hash table must fit in memory. It is difficult to make an on-disk hash map perform well. * Range queries are not efficient.

SSTables and LSM-Trees

We introduce a new requirement to segment files: we require that the sequence of key-value pairs is *sorted by key*.

We call this *Sorted String Table*, or *SSTable*. We require that each key only appears once within each merged segment file (compaction already ensures that). SSTables have few big advantages over log segments with hash indexes 1. **Merging segments is simple and efficient** (we can use algorithms like *mergesort*). When multiple segments contain the same key, we can keep the value from the most recent segment and discard the values in older segments. 2. **You no longer need to keep an index of all the keys in memory.** For a key like `handiwork`, when you know the offsets for the keys `handback` and `handsome`, you know `handiwork` must appear between those two. You can jump to the offset for `handback` and scan from there until you find `handiwork`, if not, the key is not present. You still need an in-memory index to tell you the offsets for some of the keys. One key for every few kilobytes of segment file is sufficient. 3. Since read requests need to scan over several key-value pairs in the requested range anyway, **it is possible to group those records into a block and compress it** before writing it to disk.

How do we get the data sorted in the first place? With red-black trees or AVL trees, you can insert keys in any order and read them back in sorted order. * When a write comes in, add it to an in-memory balanced tree structure (*memtable*). * When the memtable gets bigger than some threshold (megabytes), write it out to disk as an SSTable file. Writes can continue to a new memtable instance. * On a read request, try to find the key in the memtable, then in the most recent on-disk segment, then in the next-older segment, etc. * From time to time, run merging and compaction in the background to discard overwritten and deleted values.

If the database crashes, the most recent writes are lost. We can keep a separate log on disk to which every write is immediately appended. That log is not in sorted order, but that doesn't matter, because its only purpose is to restore the memtable after crash. Every time the memtable is written out to an SSTable, the log can be discarded.

Storage engines that are based on this principle of merging and compacting sorted files are often called LSM structure engines (Log Structure Merge-Tree).

Lucene, an indexing engine for full-text search used by Elasticsearch and Solr, uses a similar method for storing its *term dictionary*.

LSM-tree algorithm can be slow when looking up keys that don't exist in the database. To optimise this, storage engines often use additional *Bloom filters* (a memory-efficient data structure for approximating the contents of a set).

There are also different strategies to determine the order and timing of how SSTables are compacted and merged. Mainly two *size-tiered* and *leveled* compaction. LevelDB and RocksDB use leveled compaction, HBase use size-tiered, and Cassandra supports both. In size-tiered compaction, newer and smaller SSTables are successively merged into older and larger SSTables. In leveled compaction, the key range is split up into smaller SSTables and older data is moved into separate "levels", which allows the compaction to use less disk space.

B-trees

This is the most widely used indexing structure. B-tress keep key-value pairs sorted by key, which allows efficient key-value lookups and range queries.

The log-structured indexes break the database down into variable-size *segments* typically several megabytes or more. B-trees break the database down into fixed-size *blocks* or *pages*, traditionally 4KB.

One page is designated as the *root* and you start from there. The page contains several keys and references to child pages.

If you want to update the value for an existing key in a B-tree, you search for the leaf page containing that key, change the value in that page, and write the page back to disk. If you want to add new key, find the page and add it to the page. If there isn't enough free space in the page to accommodate the new key, it is split in two half-full pages, and the parent page is updated to account for the new subdivision of key ranges.

Trees remain *balanced*. A B-tree with n keys always has a depth of $O(\log n)$.

The basic underlying write operation of a B-tree is to overwrite a page on disk with new data. It is assumed that the overwrite does not change the location of the page, all references to that page remain intact. This is a big contrast to log-structured indexes such as LSM-trees, which only append to files.

Some operations require several different pages to be overwritten. When you split a page, you need to write the two pages that were split, and also overwrite their

parent. If the database crashes after only some of the pages have been written, you end up with a corrupted index.

It is common to include an additional data structure on disk: a *write-ahead log* (WAL, also know as the *redo log*).

Careful concurrency control is required if multiple threads are going to access, typically done protecting the tree internal data structures with *latches* (lightweight locks).

B-trees and LSM-trees

LSM-trees are typically faster for writes, whereas B-trees are thought to be faster for reads. Reads are typically slower on LSM-tress as they have to check several different data structures and SSTables at different stages of compaction.

Advantages of LSM-trees: * LSM-trees are typically able to sustain higher write throughput than B-trees, party because they sometimes have lower write amplification: a write to the database results in multiple writes to disk. The more a storage engine writes to disk, the fewer writes per second it can handle. * LSM-trees can be compressed better, and thus often produce smaller files on disk than B-trees. B-trees tend to leave disk space unused due to fragmentation.

Downsides of LSM-trees: * Compaction process can sometimes interfere with the performance of ongoing reads and writes. B-trees can be more predictable. The bigger the database, the the more disk bandwidth is required for compaction. Compaction cannot keep up with the rate of incoming writes, if not configured properly you can run out of disk space. * On B-trees, each key exists in exactly one place in the index. This offers strong transactional semantics. Transaction isolation is implemented using locks on ranges of keys, and in a B-tree index, those locks can be directly attached to the tree.

Other indexing structures

We've only discussed key-value indexes, which are like *primary key* index. There are also *secondary indexes*.

A secondary index can be easily constructed from a key-value index. The main difference is that in a secondary index, the indexed values are not necessarily unique. There are two ways of doing this: making each value in the index a list of matching row identifiers or by making a each entry unique by appending a row identifier to it.

Full-text search and fuzzy indexes

Indexes don't allow you to search for *similar* keys, such as misspelled words. Such *fuzzy* querying requires different techniques.

Full-text search engines allow synonyms, grammatical variations, occurrences of words near each other.

Lucene uses SSTable-like structure for its term dictionary. Lucene, the in-memory index is a finite state automaton, similar to a *trie*.

Keeping everything in memory

Disks have two significant advantages: they are durable, and they have lower cost per gigabyte than RAM.

It's quite feasible to keep them entirely in memory, this has lead to *in-memory* databases.

Key-value stores, such as Memcached are intended for cache only, it's acceptable for data to be lost if the machine is restarted. Other in-memory databases aim for durability, with special hardware, writing a log of changes to disk, writing periodic snapshots to disk or by replicating in-memory sate to other machines.

When an in-memory database is restarted, it needs to reload its state, either from disk or over the network from a replica. The disk is merely used as an append-only log for durability, and reads are served entirely from memory.

Products such as VoltDB, MemSQL, and Oracle TimesTime are in-memory databases. Redis and Couchbase provide weak durability.

In-memory databases can be faster because they can avoid the overheads of encoding in-memory data structures in a form that can be written to disk.

Another interesting area is that in-memory databases may provide data models that are difficult to implement with disk-based indexes.

Transaction processing or analytics?

A *transaction* is a group of reads and writes that form a logical unit, this pattern became known as *online transaction processing* (OLTP).

Data analytics has very different access patterns. A query would need to scan over a huge number of records, only reading a few columns per record, and calculates aggregate statistics.

These queries are often written by business analysts, and fed into reports. This pattern became known for *online analytics processing* (OLAP).

Data warehousing

A *data warehouse* is a separate database that analysts can query to their heart's content without affecting OLTP operations. It contains read-only copy of the dat in all various OLTP systems in the company. Data is extracted out of OLTP databases (through periodic data dump or a continuous stream of update), transformed into an analysis-friendly schema, cleaned up, and then loaded into the data warehouse (process *Extract-Transform-Load* or ETL).

A data warehouse is most commonly relational, but the internals of the systems can look quite different.

Amazon RedShift is hosted version of ParAccel. Apache Hive, Spark SQL, Cloudera Impala, Facebook Presto, Apache Tajo, and Apache Drill. Some of them are based on ideas from Google's Dremel.

Data warehouses are used in fairly formulaic style known as a *star schema*.

Facts are captured as individual events, because this allows maximum flexibility of analysis later. The fact table can become extremely large.

Dimensions represent the *who*, *what*, *where*, *when*, *how* and *why* of the event.

The name "star schema" comes from the fact than when the table relationships are visualised, the fact table is in the middle, surrounded by its dimension tables, like the rays of a star.

Fact tables often have over 100 columns, sometimes several hundred. Dimension tables can also be very wide.

Column-oriented storage

In a row-oriented storage engine, when you do a query that filters on a specific field, the engine will load all those rows with all their fields into memory, parse them and filter out the ones that don't meet the requirement. This can take a long time.

Column-oriented storage is simple: don't store all the values from one row together, but store all values from each *column* together instead. If each column is stored in a separate file, a query only needs to read and parse those columns that are used in a query, which can save a lot of work.

Column-oriented storage often lends itself very well to compression as the sequences of values for each column look quite repetitive, which is a good sign for compression. A technique that is particularly effective in data warehouses is *bitmap encoding*.

Bitmap indexes are well suited for all kinds of queries that are common in a data warehouse.

> Cassandra and HBase have a concept of *column families*, which they inherited from Bigtable.

Besides reducing the volume of data that needs to be loaded from disk, column-oriented storage layouts are also good for making efficient use of CPU cycles (*vectorised processing*).

Column-oriented storage, compression, and sorting helps to make read queries faster and make sense in data warehouses, where most of the load consist on large read-only queries run by analysts. The downside is that writes are more difficult.

An update-in-place approach, like B-tree use, is not possible with compressed columns. If you insert a row in the middle of a sorted table, you would most likely have to rewrite all column files.

It's worth mentioning *materialised aggregates* as some cache of the counts ant the sums that queries use most often. A way of creating such a cache is with a *materialised view*, on a relational model this is usually called a *virtual view*: a table-like object whose contents are the results of some query. A materialised view is an actual copy of the query results, written in disk, whereas a virtual view is just a shortcut for writing queries.

When the underlying data changes, a materialised view needs to be updated, because it is denormalised copy of the data. Database can do it automatically, but writes would become more expensive.

A common special case of a materialised view is know as a *data cube* or *OLAP cube*, a grid of aggregates grouped by different dimensions.

Encoding and evolution

Change to an application's features also requires a change to data it stores.

Relational databases conforms to one schema although that schema can be changed, there is one schema in force at any point in time. **Schema-on-read (or schemaless) contain a mixture of older and newer data formats.**

In large applications changes don't happen instantaneously. You want to perform a *rolling upgrade* and deploy a new version to a few nodes at a time, gradually working your way through all the nodes without service downtime.

Old and new versions of the code, and old and new data formats, may potentially all coexist. We need to maintain compatibility in both directions * Backward compatibility, newer code can read data that was written by older code. * Forward compatibility, older code can read data that was written by newer code.

Formats for encoding data

Two different representations: * In memory * When you want to write data to a file or send it over the network, you have to encode it

Thus, you need a translation between the two representations. In-memory representation to byte sequence is called *encoding* (*serialisation* or *marshalling*), and the reverse is called *decoding* (*parsing*, *deserialisation* or *unmarshalling*).

Programming languages come with built-in support for encoding in-memory objects into byte sequences, but is usually a bad idea to use them. Precisely because of a few problems. * Often tied to a particular programming language. * The decoding process needs to be able to instantiate arbitrary classes and this is frequently a security hole. * Versioning * Efficiency

Standardised encodings can be written and read by many programming languages.

JSON, XML, and CSV are human-readable and popular specially as data interchange formats, but they have some subtle problems: * Ambiguity around the encoding of numbers and dealing with large numbers * Support of Unicode character strings, but no support for binary strings. People get around this by encoding binary data as Base64, which increases the data size by 33%. * There is optional schema support for both XML and JSON * CSV does not have any schema

Binary encoding

JSON is less verbose than XML, but both still use a lot of space compared to binary formats. There are binary encodings for JSON (MesagePack, BSON, BJSON, UBJSON, BISON and Smile), similar thing for XML (WBXML and Fast Infoset).

Apache Thrift and Protocol Buffers (protobuf) are binary encoding libraries.

Thrift offers two different protocols: * **BinaryProtocol**, there are no field names like `userName`, `favouriteNumber`. Instead the data contains *field tags*, which are numbers (`1`, `2`) * **CompactProtocol**, which is equivalent to BinaryProtocol but it packs the same information in less space. It packs the field type and the tag number into the same byte.

Protocol Buffers are very similar to Thrift's CompactProtocol, bit packing is a bit different and that might allow smaller compression.

Schemas inevitable need to change over time (*schema evolution*), how do Thrift and Protocol Buffers handle schema changes while keeping backward and forward compatibility changes?

- **Forward compatible support**. As with new fields you add new tag numbers, old code trying to read new code, it can simply ignore not recognised tags.
- **Backwards compatible support**. As long as each field has a unique tag number, new code can always read old data. Every field you add after initial deployment of schema must be optional or have a default value.

Removing fields is just like adding a field with backward and forward concerns reversed. You can only remove a field that is optional, and you can never use the same tag again.

What about changing the data type of a field? There is a risk that values will lose precision or get truncated.

Avro

Apache Avro is another binary format that has two schema languages, one intended for human editing (Avro IDL), and one (based on JSON) that is more easily machine-readable.

You go go through the fields in the order they appear in the schema and use the schema to tell you the datatype of each field. Any mismatch in the schema between the reader and the writer would mean incorrectly decoded data.

What about schema evolution? When an application wants to encode some data, it encodes the data using whatever version of the schema it knows (*writer's schema*).

When an application wants to decode some data, it is expecting the data to be in some schema (*reader's schema*).

In Avro the writer's schema and the reader's schema *don't have to be the same*. The Avro library resolves the differences by looking at the writer's schema and the reader's schema.

Forward compatibility means you can have a new version of the schema as writer and an old version of the schema as reader. Conversely, backward compatibility means that you can have a new version of the schema as reader and an old version as writer.

To maintain compatibility, you may only add or remove a field that has a default value.

If you were to add a field that has no default value, new readers wouldn't be able to read data written by old writers.

Changing the datatype of a field is possible, provided that Avro can convert the type. Changing the name of a filed is tricky (backward compatible but not forward compatible).

The schema is identified encoded in the data. In a large file with lots of records, the writer of the file can just include the schema at the beginning of the file. On a database with individually written records, you cannot assume all the records will have the same schema, so you have to include a version number at the beginning of every encoded record. While sending records over the network, you can negotiate the schema version on connection setup.

Avro is friendlier to *dynamically generated schemas* (dumping into a file the database). You can fairly easily generate an Avro schema in JSON.

If the database schema changes, you can just generate a new Avro schema for the updated database schema and export data in the new Avro schema.

By contrast with Thrift and Protocol Buffers, every time the database schema changes, you would have to manually update the mappings from database column names to field tags.

Although textual formats such as JSON, XML and CSV are widespread, binary encodings based on schemas are also a viable option. As they have nice properties: * Can be much more compact, since they can omit field names from the encoded data. * Schema is a valuable form of documentation, required for decoding, you can be sure it is up to date. * Database of schemas allows you to check forward and backward compatibility changes. * Generate code from the schema is useful, since it enables type checking at compile time.

Modes of dataflow

Different process on how data flows between processes

Via databases

The process that writes to the database encodes the data, and the process that reads from the database decodes it.

A value in the database may be written by a *newer* version of the code, and subsequently read by an *older* version of the code that is still running.

When a new version of your application is deployed, you may entirely replace the old version with the new version within a few minutes. The same is not true in databases, the five-year-old data will still be there, in the original encoding, unless you have explicitly rewritten it. *Data outlives code*.

Rewriting (*migrating*) is expensive, most relational databases allow simple schema changes, such as adding a new column with a `null` default value without rewriting existing data. When an old row is read, the database fills in `null`s for any columns that are missing.

Via service calls

You have processes that need to communicate over a network of *clients* and *servers*.

Services are similar to databases, each service should be owned by one team. and that team should be able to release versions of the service frequently, without having to coordinate with other teams. We should expect old and new versions of servers and clients to be running at the same time.

Remote procedure calls (RPC) tries to make a request to a remote network service look the same as calling a function or method in your programming language, it seems convenient at first but the approach is flawed: * A network request is unpredictable * A network request it may return without a result, due a *timeout* * Retrying will cause the action to be performed multiple times, unless you build a mechanism for deduplication (*idempotence*). * A network request is much slower than a function call, and its latency is wildly variable. * Parameters need to be encoded into a sequence of bytes that can be sent over the network and becomes problematic with larger objects. * The RPC framework must translate datatypes from one language to another, not all languages have the same types.

There is no point trying to make a remote service look too much like a local object in your programming language, because it's a fundamentally different thing.

New generation of RPC frameworks are more explicit about the fact that a remote request is different from a local function call. Fiangle and Rest.li use *features* (*promises*) to encapsulate asyncrhonous actions.

RESTful API has some significant advantages like being good for experimentation and debugging.

REST seems to be the predominant style for public APIs. The main focus of RPC frameworks is on requests between services owned by the same organisation, typically within the same datacenter.

Via asynchronous message passing

In an *asynchronous message-passing* systems, a client's request (usually called a *message*) is delivered to another process with low latency. The message goes via an intermediary called a *message broker* (*message queue* or *message-oriented middleware*) which stores the message temporarily. This has several advantages

compared to direct RPC: * It can act as a buffer if the recipient is unavailable or overloaded * It can automatically redeliver messages to a process that has crashed and prevent messages from being lost * It avoids the sender needing to know the IP address and port number of the recipient (useful in a cloud environment) * It allows one message to be sent to several recipients * **Decouples the sender from the recipient**

The communication happens only in one direction. The sender doesn't wait for the message to be delivered, but simply sends it and then forgets about it (*asynchronous*).

Open source implementations for message brokers are RabbitMQ, ActiveMQ, HornetQ, NATS, and Apache Kafka.

One process sends a message to a named *queue* or *topic* and the broker ensures that the message is delivered to one or more *consumers* or *subscribers* to that queue or topic.

Message brokers typically don't enforce a particular data model, you can use any encoding format.

An *actor model* is a programming model for concurrency in a single process. Rather than dealing with threads (and their complications), logic is encapsulated in *actors*. Each actor typically represent one client or entity, it may have some local state, and it communicates with other actors by sending and receiving asynchronous messages. Message deliver is not guaranteed. Since each actor processes only one message at a time, it doesn't need to worry about threads.

In *distributed actor frameworks*, this programming model is used to scale an application across multiple nodes. It basically integrates a message broker and the actor model into a single framework.

- *Akka* uses Java's built-in serialisation by default, which does not provide forward or backward compatibility. You can replace it with something like Protocol Buffers and the ability to do rolling upgrades.
- *Orleans* by default uses custom data encoding format that does not support rolling upgrade deployments.
- In *Erlang OTP* it is surprisingly hard to make changes to record schemas.

What happens if multiple machines are involved in storage and retrieval of data?

Reasons for distribute a database across multiple machines: * Scalability * Fault tolerance/high availability * Latency, having servers at various locations worldwide

Replication

Reasons why you might want to replicate data: * To keep data geographically close to your users * Increase availability * Increase read throughput

The difficulty in replication lies in handling *changes* to replicated data. Popular algorithms for replicating changes between nodes: *single-leader*, *multi-leader*, and *leaderless* replication.

Leaders and followers

Each node that stores a copy of the database is called a *replica*.

Every write to the database needs to be processed by every replica. The most common solution for this is called *leader-based replication* (*active/passive* or *master-slave replication*). 1. One of the replicas is designated the *leader* (*master* or *primary*). Writes to the database must send requests to the leader. 2. Other replicas are known as *followers* (*read replicas*, *slaves*, *secondaries* or *hot stanbys*). The leader sends the data change to all of its followers as part of a *replication log* or *change stream*. 3. Reads can be query the leader or any of the followers, while writes are only accepted on the leader.

MySQL, Oracle Data Guard, SQL Server's AlwaysOn Availability Groups, MongoDB, RethinkDB, Espresso, Kafka and RabbitMQ are examples of these kind of databases.

Synchronous vs asynchronous

The advantage of synchronous replication is that the follower is guaranteed to have an up-to-date copy of the data that is consistent with the leader. The disadvantage is that it the synchronous follower doesn't respond, the write cannot be processed.

It's impractical for all followers to be synchronous. If you enable synchronous replication on a database, it usually means that *one* of the followers is synchronous, and the others are asynchronous. This guarantees up-to-date copy of the data on at least two nodes (this is sometimes called *semi-synchronous*).

Often, leader-based replication is asynchronous. Writes are not guaranteed to be durable, the main advantage of this approach is that the leader can continue processing writes.

Setting up new followers

Copying data files from one node to another is typically not sufficient.

Setting up a follower can usually be done without downtime. The process looks like: 1. Take a snapshot of the leader's database 2. Copy the snapshot to the follower node 3. Follower requests data changes that have happened since the snapshot was taken 4. Once follower processed the backlog of data changes since snapshot, it has *caught up*.

Handling node outages

How does high availability works with leader-based replication?

Follower failure: catchup recovery

Follower can connect to the leader and request all the data changes that occurred during the time when the follower was disconnected.

Leader failure: failover

One of the followers needs to be promoted to be the new leader, clients need to be reconfigured to send their writes to the new leader and followers need to start consuming data changes from the new leader.

Automatic failover consists: 1. Determining that the leader has failed. If a node does not respond in a period of time it's considered dead. 2. Choosing a new leader.

The best candidate for leadership is usually the replica with the most up-to-date changes from the old leader. 3. Reconfiguring the system to use the new leader. The system needs to ensure that the old leader becomes a follower and recognises the new leader.

Things that could go wrong: * If asynchronous replication is used, the new leader may have received conflicting writes in the meantime. * Discarding writes is especially dangerous if other storage systems outside of the database need to be coordinated with the database contents. * It could happen that two nodes both believe that they are the leader (*split brain*). Data is likely to be lost or corrupted. * What is the right time before the leader is declared dead?

For these reasons, some operation teams prefer to perform failovers manually, even if the software supports automatic failover.

Implementation of replication logs

Statement-based replication

The leader logs every *statement* and sends it to its followers (every `INSERT`, `UPDATE` or `DELETE`).

This type of replication has some problems: * Non-deterministic functions such as `NOW()` or `RAND()` will generate different values on replicas. * Statements that depend on existing data, like auto-increments, must be executed in the same order in each replica. * Statements with side effects may result on different results on each replica.

A solution to this is to replace any nondeterministic function with a fixed return value in the leader.

Write-ahead log (WAL) shipping

The log is an append-only sequence of bytes containing all writes to the database. The leader can send it to its followers. This way of replication is used in PostgresSQL and Oracle.

The main disadvantage is that the log describes the data at a very low level (like which bytes were changed in which disk blocks), coupling it to the storage engine.

Usually is not possible to run different versions of the database in leaders and followers. This can have a big operational impact, like making it impossible to have a zero-downtime upgrade of the database.

Logical (row-based) log replication

Basically a sequence of records describing writes to database tables at the granularity of a row: * For an inserted row, the new values of all columns. * For a deleted row, the information that uniquely identifies that column. * For an updated row, the information to uniquely identify that row and all the new values of the columns.

A transaction that modifies several rows, generates several of such logs, followed by a record indicating that the transaction was committed. MySQL binlog uses this approach.

Since logical log is decoupled from the storage engine internals, it's easier to make it backwards compatible.

Logical logs are also easier for external applications to parse, useful for data warehouses, custom indexes and caches (*change data capture*).

Trigger-based replication

There are some situations were you may need to move replication up to the application layer.

A trigger lets you register custom application code that is automatically executed when a data change occurs. This is a good opportunity to log this change into a separate table, from which it can be read by an external process.

Main disadvantages is that this approach has greater overheads, is more prone to bugs but it may be useful due to its flexibility.

Problems with replication lag

Node failures is just one reason for wanting replication. Other reasons are scalability and latency.

In a *read-scaling* architecture, you can increase the capacity for serving read-only requests simply by adding more followers. However, this only realistically works on asynchronous replication. The more nodes you have, the likelier is that one will be down, so a fully synchronous configuration would be unreliable.

With an asynchronous approach, a follower may fall behind, leading to inconsistencies in the database (*eventual consistency*).

The *replication lag* could be a fraction of a second or several seconds or even minutes.

The problems that may arise and how to solve them.

Reading your own writes

Read-after-write consistency, also known as *read-your-writes consistency* is a guarantee that if the user reloads the page, they will always see any updates they submitted themselves.

How to implement it: * **When reading something that the user may have modified, read it from the leader.** For example, user profile information on a social network is normally only editable by the owner. A simple rule is always read the user's own profile from the leader. * You could track the time of the latest update and, for one minute after the last update, make all reads from the leader. * The client can remember the timestamp of the most recent write, then the system can ensure that the replica serving any reads for that user reflects updates at least until that timestamp. * If your replicas are distributed across multiple datacenters, then any request needs to be routed to the datacenter that contains the leader.

Another complication is that the same user is accessing your service from multiple devices, you may want to provide *cross-device* read-after-write consistency.

Some additional issues to consider: * Remembering the timestamp of the user's last update becomes more difficult. The metadata will need to be centralised. * If replicas are distributed across datacenters, there is no guarantee that connections from different devices will be routed to the same datacenter. You may need to route requests from all of a user's devices to the same datacenter.

Monotonic reads

Because of followers falling behind, it's possible for a user to see things *moving backward in time*.

When you read data, you may see an old value; monotonic reads only means that if one user makes several reads in sequence, they will not see time go backward.

Make sure that each user always makes their reads from the same replica. The replica can be chosen based on a hash of the user ID. If the replica fails, the user's queries will need to be rerouted to another replica.

Consistent prefix reads

If a sequence of writes happens in a certain order, then anyone reading those writes will see them appear in the same order.

This is a particular problem in partitioned (sharded) databases as there is no global ordering of writes.

A solution is to make sure any writes casually related to each other are written to the same partition.

Solutions for replication lag

Transactions exist so there is a way for a database to provide stronger guarantees so that the application can be simpler.

Multi-leader replication

Leader-based replication has one major downside: there is only one leader, and all writes must go through it.

A natural extension is to allow more than one node to accept writes (*multi-leader*, *master-master* or *active/active* replication) where each leader simultaneously acts as a follower to the other leaders.

Use cases for multi-leader replication

It rarely makes sense to use multi-leader setup within a single datacenter.

Multi-datacenter operation

You can have a leader in *each* datacenter. Within each datacenter, regular leader-follower replication is used. Between datacenters, each datacenter leader replicates its changes to the leaders in other datacenters.

Compared to a single-leader replication model deployed in multi-datacenters * **Performance.** With single-leader, every write must go across the internet to wherever the leader is, adding significant latency. In multi-leader every write is processed in the local datacenter and replicated asynchronously to other datacenters. The network delay is hidden from users and perceived performance may be better. * **Tolerance of datacenter outages.** In single-leader if the datacenter with the leader fails, failover can promote a follower in another datacenter. In multi-leader, each datacenter can continue operating independently from others. * **Tolerance of network problems.** Single-leader is very sensitive to problems in this inter-datacenter link as writes are made synchronously over this link. Multi-leader with asynchronous replication can tolerate network problems better.

Multi-leader replication is implemented with Tungsten Replicator for MySQL, BDR for PostgreSQL or GoldenGate for Oracle.

It's common to fall on subtle configuration pitfalls. Autoincrementing keys, triggers and integrity constraints can be problematic. Multi-leader replication is often considered dangerous territory and avoided if possible.

Clients with offline operation

If you have an application that needs to continue to work while it is disconnected from the internet, every device that has a local database can act as a leader, and there will be some asynchronous multi-leader replication process (imagine, a Calendar application).

CouchDB is designed for this mode of operation.

Collaborative editing

Real-time collaborative editing applications allow several people to edit a document simultaneously. Like Etherpad or Google Docs.

The user edits a document, the changes are instantly applied to their local replica and asynchronously replicated to the server and any other user.

If you want to avoid editing conflicts, you must the lock the document before a user can edit it.

For faster collaboration, you may want to make the unit of change very small (like a keystroke) and avoid locking.

Handling write conflicts

The biggest problem with multi-leader replication is when conflict resolution is required. This problem does not happen in a single-leader database.

Synchronous vs asynchronous conflict detection

In single-leader the second writer can be blocked and wait the first one to complete, forcing the user to retry the write. On multi-leader if both writes are successful, the conflict is only detected asynchronously later in time.

If you want synchronous conflict detection, you might as well use single-leader replication.

Conflict avoidance

The simplest strategy for dealing with conflicts is to avoid them. If all writes for a particular record go through the sae leader, then conflicts cannot occur.

On an application where a user can edit their own data, you can ensure that requests from a particular user are always routed to the same datacenter and use the leader in that datacenter for reading and writing.

Converging toward a consistent state

On single-leader, the last write determines the final value of the field.

In multi-leader, it's not clear what the final value should be.

The database must resolve the conflict in a *convergent* way, all replicas must arrive a the same final value when all changes have been replicated.

Different ways of achieving convergent conflict resolution. * Five each write a unique ID (timestamp, long random number, UUID, or a has of the key and value), pick the write with the highest ID as the *winner* and throw away the other writes. This is known as *last write wins* (LWW) and it is dangerously prone to data loss. * Give each replica a unique ID, writes that originated at a higher-numbered replica always take precedence. This approach also implies data loss. * Somehow merge the values together. * Record the conflict and write application code that resolves it a to some later time (perhaps prompting the user).

Custom conflict resolution

Multi-leader replication tools let you write conflict resolution logic using application code.

- **On write.** As soon as the database system detects a conflict in the log of replicated changes, it calls the conflict handler.
- **On read.** All the conflicting writes are stored. On read, multiple versions of the data are returned to the application. The application may prompt the user or automatically resolve the conflict. CouchDB works this way.

Multi-leader replication topologies

A *replication topology* describes the communication paths along which writes are propagated from one node to another.

The most general topology is *all-to-all* in which every leader sends its writes to every other leader. MySQL uses *circular topology*, where each nodes receives writes from one node and forwards those writes to another node. Another popular topology has the shape of a *star*, one designated node forwards writes to all of the other nodes.

In circular and star topologies a write might need to pass through multiple nodes before they reach all replicas. To prevent infinite replication loops each node is given a unique identifier and the replication log tags each write with the identifiers of the nodes it has passed through. When a node fails it can interrupt the flow of replication messages.

In all-to-all topology fault tolerance is better as messages can travel along different paths avoiding a single point of failure. It has some issues too, some network links may be faster than others and some replication messages may "overtake" others. To order events correctly. there is a technique called *version vectors*. PostgresSQL BDR does not provide casual ordering of writes, and Tungsten Replicator for MySQL doesn't even try to detect conflicts.

Leaderless replication

Simply put, any replica can directly accept writes from clients. Databases like look like Amazon's in-house *Dynamo* datastore. *Riak*, *Cassandra* and *Voldemort* follow the *Dynamo style*.

In a leaderless configuration, failover does not exist. Clients send the write to all replicas in parallel.

Read requests are also sent to several nodes in parallel. The client may get different responses. Version numbers are used to determine which value is newer.

Eventually, all the data is copied to every replica. After a unavailable node come back online, it has two different mechanisms to catch up: * **Read repair.** When a client detect any stale responses, write the newer value back to that replica. * **Anti-entropy process.** There is a background process that constantly looks for differences in data between replicas and copies any missing data from one replica to he other. It does not copy writes in any particular order.

Quorums for reading and writing

If there are n replicas, every write must be confirmed by w nodes to be considered successful, and we must query at least r nodes for each read. As long as $w + r > n$, we expect to get an up-to-date value when reading. r and w values are called *quorum* reads and writes. Are the minimum number of votes required for the read or write to be valid.

A common choice is to make n and odd number (typically 3 or 5) and to set $w = r = (n + 1)/2$ (rounded up).

Limitations: * Sloppy quorum, the w writes may end up on different nodes than the r reads, so there is no longer a guaranteed overlap. * If two writes occur concurrently, and is not clear which one happened first, the only safe solution is to merge them. Writes can be lost due to clock skew. * If a write happens concurrently with a read, the write may be reflected on only some of the replicas. * If a write succeeded on some replicas but failed on others, it is not rolled back on the replicas where it succeeded. Reads may or may not return the value from that write. * If a node carrying a new value fails, and its data is restored from a replica carrying an old value, the number of replicas storing the new value may break the quorum condition.

Dynamo-style databases are generally optimised for use cases that can tolerate eventual consistency.

Sloppy quorums and hinted handoff

Leaderless replication may be appealing for use cases that require high availability and low latency, and that can tolerate occasional stale reads.

It's likely that the client won't be able to connect to *some* database nodes during a network interruption. * Is it better to return errors to all requests for which we cannot reach quorum of w or r nodes? * Or should we accept writes anyway, and write them to some nodes that are reachable but aren't among the n nodes on which the value usually lives?

The latter is known as *sloppy quorum*: writes and reads still require w and r successful responses, but those may include nodes that are not among the designated n "home" nodes for a value.

Once the network interruption is fixed, any writes are sent to the appropriate "home" nodes (*hinted handoff*).

Sloppy quorums are useful for increasing write availability: as long as any w nodes are available, the database can accept writes. This also means that you cannot be sure to read the latest value for a key, because it may have been temporarily written to some nodes outside of n.

Multi-datacenter operation

Each write from a client is sent to all replicas, regardless of datacenter, but the client usually only waits for acknowledgement from a quorum of nodes within its local

datacenter so that it is unaffected by delays and interruptions on cross-datacenter link.

Detecting concurrent writes

In order to become eventually consistent, the replicas should converge toward the same value. If you want to avoid losing data, you application developer, need to know a lot about the internals of your database's conflict handling.

- **Last write wins (discarding concurrent writes).** Even though the writes don' have a natural ordering, we can force an arbitrary order on them. We can attach a timestamp to each write and pick the most recent. There are some situations such caching on which lost writes are acceptable. If losing data is not acceptable, LWW is a poor choice for conflict resolution.
- **The "happens-before" relationship and concurrency.** Whether one operation happens before another operation is the key to defining what concurrency means. **We can simply say that to operations are *concurrent* if neither happens before the other.** Either A happened before B, or B happened before A, or A and B are concurrent.

Capturing the happens-before relationship

The server can determine whether two operations are concurrent by looking at the version numbers. * The server maintains a version number for every key, increments the version number every time that key is written, and stores the new version number along the value written. * Client reads a key, the server returns all values that have not been overwrite, as well as the latest version number. A client must read a key before writing. * Client writes a key, it must include the version number from the prior read, and it must merge together all values that it received in the prior read. * Server receives a write with a particular version number, it can overwrite all values with that version number or below, but it must keep all values with a higher version number.

Merging concurrently written values

No data is silently dropped. It requires clients do some extra work, they have to clean up afterward by merging the concurrently written values. Riak calls these concurrent values *siblings*.

Merging sibling values is the same problem as conflict resolution in multi-leader replication. A simple approach is to just pick one of the values on a version number or timestamp (last write wins). You may need to do something more intelligent in application code to avoid losing data.

If you want to allow people to *remove* things, union of siblings may not yield the right result. An item cannot simply be deleted from the database when it is removed, the system must leave a marker with an appropriate version number to indicate that the item has been removed when merging siblings (*tombstone*).

Merging siblings in application code is complex and error-prone, there are efforts to design data structures that can perform this merging automatically (CRDTs).

Version vectors

We need a version number *per replica* as well as per key. Each replica increments its own version number when processing a write, and also keeps track of the version numbers it has seen from each of the other replicas.

The collection of version numbers from all the replicas is called a *version vector*.

Version vector are sent from the database replicas to clients when values are read, and need to be sent back to the database when a value is subsequently written. Riak calls this *casual context*. Version vectors allow the database to distinguish between overwrites and concurrent writes.

Partitioning

Replication, for very large datasets or very high query throughput is not sufficient, we need to break the data up into *partitions* (*sharding*).

Basically, each partition is a small database of its own.

The main reason for wanting to partition data is *scalability*, query load can be load cabe distributed across many processors. Throughput can be scaled by adding more nodes.

Partitioning and replication

Each record belongs to exactly one partition, it may still be stored on several nodes for fault tolerance.

A node may store more than one partition.

Partition of key-value data

Our goal with partitioning is to spread the data and the query load evenly across nodes.

If partition is unfair, we call it *skewed*. It makes partitioning much less effective. A partition with disproportionately high load is called a *hot spot*.

The simplest approach is to assign records to nodes randomly. The main disadvantage is that if you are trying to read a particular item, you have no way of knowing which node it is on, so you have to query all nodes in parallel.

Partition by key range

Assign a continuous range of keys, like the volumes of a paper encyclopaedia. Boundaries might be chose manually by an administrator, or the database can choose them automatically. On each partition, keys are in sorted order so scans are easy.

The downside is that certain access patterns can lead to hot spots.

Partitioning by hash of key

A good hash function takes skewed data and makes it uniformly distributed. There is no need to be cryptographically strong (MongoDB uses MD5 and Cassandra uses Murmur3). You can assign each partition a range of hashes. The boundaries can be evenly spaced or they can be chosen pseudorandomly (*consistent hashing*).

Unfortunately we lose the ability to do efficient range queries. Keys that were once adjacent are now scattered across all the partitions. Any range query has to be sent to all partitions.

Skewed workloads and relieving hot spots

You can't avoid hot spots entirely. For example, you may end up with large volume of writes to the same key.

It's the responsibility of the application to reduce the skew. A simple technique is to add a random number to the beginning or end of the key.

Splitting writes across different keys, makes reads now to do some extra work and combine them.

Partitioning and secondary indexes

The situation gets more complicated if secondary indexes are involved. A secondary index usually doesn't identify the record uniquely. They don't map neatly to partitions.

Partitioning secondary indexes by document

Each partition maintains its secondary indexes, covering only the documents in that partition (*local index*).

You need to send the query to *all* partitions, and combine all the results you get back (*scatter/gather*). This is prone to tail latency amplification and is widely used in MongoDB, Riak, Cassandra, Elasticsearch, SolrCloud and VoltDB.

Partitioning secondary indexes by term

We construct a *global index* that covers data in all partitions. The global index must also be partitioned so it doesn't become the bottleneck.

It is called the *term-partitioned* because the term we're looking for determines the partition of the index.

Partitioning by term can be useful for range scans, whereas partitioning on a hash of the term gives a more even distribution load.

The advantage is that it can make reads more efficient: rather than doing scatter/gather over all partitions, a client only needs to make a request to the partition containing the term that it wants. The downside of a global index is that writes are slower and complicated.

Rebalancing partitions

The process of moving load from one node in the cluster to another.

Strategies for rebalancing: * **How not to do it: Hash mod n.** The problem with *mod N* is that if the number of nodes *N* changes, most of the keys will need to be moved from one node to another. * **Fixed number of partitions.** Create many more partitions than there are nodes and assign several partitions to each node. If a node is added to the cluster, we can *steal* a few partitions from every existing node until partitions are fairly distributed once again. The number of partitions does not change, nor does the assignment of keys to partitions. The only thing that change is the assignment of partitions to nodes. This is used in Riak, Elasticsearch, Couchbase, and Voldemport. **You need to choose a high enough number of partitions to accomodate future growth.** Neither too big or too small. * **Dynamic partitioning.** The number of partitions adapts to the total data volume. An empty database starts with an empty partition. While the dataset is small, all writes have to processed by a single node while the others nodes sit idle. HBase and MongoDB allow an initial set of partitions to be configured (*pre-splitting*). * **Partitioning proportionally to nodes.** Cassandra and Ketama make the number of partitions proportional to the number of nodes. Have a fixed number of partitions *per node*. This approach also keeps the size of each partition fairly stable.

Automatic versus manual rebalancing

Fully automated rebalancing may seem convenient but the process can overload the network or the nodes and harm the performance of other requests while the rebalancing is in progress.

It can be good to have a human in the loop for rebalancing. You may avoid operational surprises.

Request routing

This problem is also called *service discovery*. There are different approaches: 1. Allow clients to contact any node and make them handle the request directly, or forward the request to the appropriate node. 2. Send all requests from clients to a routing tier first that acts as a partition-aware load balancer. 3. Make clients aware of the partitioning and the assignment of partitions to nodes.

In many cases the problem is: how does the component making the routing decision learn about changes in the assignment of partitions to nodes?

Many distributed data systems rely on a separate coordination service such as ZooKeeper to keep track of this cluster metadata. Each node registers itself in ZooKeeper, and ZooKeeper maintains the authoritative mapping of partitions to nodes. The routing tier or the partitioning-aware client, can subscribe to this information in ZooKeeper. HBase, SolrCloud and Kafka use ZooKeeper to track partition assignment. MongoDB relies on its own *config server*. Cassandra and Riak take a different approach: they use a *gossip protocol*.

Parallel query execution

Massively parallel processing (MPP) relational database products are much more sophisticated in the types of queries they support.

Transactions

Implementing fault-tolerant mechanisms is a lot of work.

The slippery concept of a transaction

Transactions have been the mechanism of choice for simplifying these issues. Conceptually, all the reads and writes in a transaction are executed as one operation:

either the entire transaction succeeds (*commit*) or it fails (*abort*, *rollback*).

The application is free to ignore certain potential error scenarios and concurrency issues (*safety guarantees*).

ACID

- **Atomicity.** Is *not* about concurrency. It is what happens if a client wants to make several writes, but a fault occurs after some of the writes have been processed. *Abortability* would have been a better term than *atomicity*.
- **Consistency.** *Invariants* on your data must always be true. The idea of consistency depends on the application's notion of invariants. Atomicity, isolation, and durability are properties of the database, whereas consistency (in an ACID sense) is a property of the application.
- **Isolation.** Concurrently executing transactions are isolated from each other. It's also called *serializability*, each transaction can pretend that it is the only transaction running on the entire database, and the result is the same as if they had run *serially* (one after the other).
- **Durability.** Once a transaction has committed successfully, any data it has written will not be forgotten, even if there is a hardware fault or the database crashes. In a single-node database this means the data has been written to nonvolatile storage. In a replicated database it means the data has been successfully copied to some number of nodes.

Atomicity can be implemented using a log for crash recovery, and isolation can be implemented using a lock on each object, allowing only one thread to access an object at any one time.

A transaction is a mechanism for grouping multiple operations on multiple objects into one unit of execution.

Handling errors and aborts

A key feature of a transaction is that it can be aborted and safely retried if an error occurred.

In datastores with leaderless replication is the application's responsibility to recover from errors.

The whole point of aborts is to enable safe retries.

Weak isolation levels

Concurrency issues (race conditions) come into play when one transaction reads data that is concurrently modified by another transaction, or when two transactions try to simultaneously modify the same data.

Databases have long tried to hide concurrency issues by providing *transaction isolation*.

In practice, is not that simple. Serializable isolation has a performance cost. It's common for systems to use weaker levels of isolation, which protect against *some* concurrency issues, but not all.

Weak isolation levels used in practice:

Read committed

It makes two guarantees: 1. When reading from the database, you will only see data that has been committed (no *dirty reads*). Writes by a transaction only become visible to others when that transaction commits. 2. When writing to the database, you will only overwrite data that has been committed (no *dirty writes*). Dirty writes are prevented usually by delaying the second write until the first write's transaction has committed or aborted.

Most databases prevent dirty writes by using row-level locks that hold the lock until the transaction is committed or aborted. Only one transaction can hold the lock for any given object.

On dirty reads, requiring read locks does not work well in practice as one long-running write transaction can force many read-only transactions to wait. For every object that is written, the database remembers both the old committed value and the new value set by the transaction that currently holds the write lock. While the transaction is ongoing, any other transactions that read the object are simply given the old value.

Snapshot isolation and repeatable read

There are still plenty of ways in which you can have concurrency bugs when using this isolation level.

Nonrepeatable read or *read skew*, when you read at the same time you committed a change you may see temporal and inconsistent results.

There are some situations that cannot tolerate such temporal inconsistencies: * **Backups.** During the time that the backup process is running, writes will continue to be made to the database. If you need to restore from such a backup, inconsistencies can become permanent. * **Analytic queries and integrity checks.** You may get nonsensical results if they observe parts of the database at different points in time.

Snapshot isolation is the most common solution. Each transaction reads from a *consistent snapshot* of the database.

The implementation of snapshots typically use write locks to prevent dirty writes.

The database must potentially keep several different committed versions of an object (*multi-version concurrency control* or MVCC).

Read committed uses a separate snapshot for each query, while snapshot isolation uses the same snapshot for an entire transaction.

How do indexes work in a multi-version database? One option is to have the index simply point to all versions of an object and require an index query to filter out any object versions that are not visible to the current transaction.

Snapshot isolation is called *serializable* in Oracle, and *repeatable read* in PostgreSQL and MySQL.

Preventing lost updates

This might happen if an application reads some value from the database, modifies it, and writes it back. If two transactions do this concurrently, one of the modifications can be lost (later write *clobbers* the earlier write).

Atomic write operations

A solution for this it to avoid the need to implement read-modify-write cycles and provide atomic operations such us

```
UPDATE counters SET value = value + 1 WHERE key = 'foo';
```

MongoDB provides atomic operations for making local modifications, and Redis provides atomic operations for modifying data structures.

Explicit locking

The application explicitly lock objects that are going to be updated.

Automatically detecting lost updates

Allow them to execute in parallel, if the transaction manager detects a lost update, abort the transaction and force it to retry its read-modify-write cycle.

MySQL/InnoDB's repeatable read does not detect lost updates.

Compare-and-set

If the current value does not match with what you previously read, the update has no effect.

```
UPDATE wiki_pages SET content = 'new content'
  WHERE id = 1234 AND content = 'old content';
```

Conflict resolution and replication

With multi-leader or leaderless replication, compare-and-set do not apply.

A common approach in replicated databases is to allow concurrent writes to create several conflicting versions of a value (also know as *siblings*), and to use application code or special data structures to resolve and merge these versions after the fact.

Write skew and phantoms

Imagine Alice and Bob are two on-call doctors for a particular shift. Imagine both the request to leave because they are feeling unwell. Unfortunately they happen to click the button to go off call at approximately the same time.

```
ALICE                                   BOB

┌─ BEGIN TRANSACTION                    ┌─ BEGIN TRANSACTION
│                                       │
├─ currently_on_call = (                ├─ currently_on_call = (
│   select count(*) from doctors        │    select count(*) from doctors
│   where on_call = true                │    where on_call = true
│   and shift_id = 1234                 │    and shift_id = 1234
│  )                                    │  )
│  // now currently_on_call = 2         │  // now currently_on_call = 2
│                                       │
├─ if (currently_on_call  2) {          │
│    update doctors                     │
│    set on_call = false                │
│    where name = 'Alice'               │
│    and shift_id = 1234                │
│  }                                    │
│                                       ├─ if (currently_on_call >= 2) {
└─ COMMIT TRANSACTION                   │    update doctors
                                        │    set on_call = false
                                        │    where name = 'Bob'
                                        │    and shift_id = 1234
                                        │  }
                                        │
                                        └─ COMMIT TRANSACTION
```

Since database is using snapshot isolation, both checks return 2. Both transactions commit, and now no doctor is on call. The requirement of having at least one doctor has been violated.

Write skew can occur if two transactions read the same objects, and then update some of those objects. You get a dirty write or lost update anomaly.

Ways to prevent write skew are a bit more restricted: * Atomic operations don't help as things involve more objects. * Automatically prevent write skew requires true serializable isolation. * The second-best option in this case is probably to explicitly lock the rows that the transaction depends on. ```sql BEGIN TRANSACTION;

SELECT * FROM doctors WHERE on_call = true AND shift_id = 1234 FOR UPDATE;

UPDATE doctors SET on_call = false WHERE name = 'Alice' AND shift_id = 1234;

COMMIT; ```

Serializability

This is the strongest isolation level. It guarantees that even though transactions may execute in parallel, the end result is the same as if they had executed one at a time, *serially*, without concurrency. Basically, the database prevents *all* possible race conditions.

There are three techniques for achieving this: * Executing transactions in serial order * Two-phase locking * Serializable snapshot isolation.

Actual serial execution

The simplest way of removing concurrency problems is to remove concurrency entirely and execute only one transaction at a time, in serial order, on a single thread. This approach is implemented by VoltDB/H-Store, Redis and Datomic.

Encapsulating transactions in stored procedures

With interactive style of transaction, a lot of time is spent in network communication between the application and the database.

For this reason, systems with single-threaded serial transaction processing don't allow interactive multi-statement transactions. The application must submit the entire transaction code to the database ahead of time, as a *stored procedure*, so all the data required by the transaction is in memory and the procedure can execute very fast.

There are a few pros and cons for stored procedures: * Each database vendor has its own language for stored procedures. They usually look quite ugly and archaic from today's point of view, and they lack the ecosystem of libraries. * It's harder to debug, more awkward to keep in version control and deploy, trickier to test, and difficult to integrate with monitoring.

Modern implementations of stored procedures include general-purpose programming languages instead: VoltDB uses Java or Groovy, Datomic uses Java or Clojure, and Redis uses Lua.

Partitioning

Executing all transactions serially limits the transaction throughput to the speed of a single CPU.

In order to scale to multiple CPU cores you can potentially partition your data and each partition can have its own transaction processing thread. You can give each CPU core its own partition.

For any transaction that needs to access multiple partitions, the database must coordinate the transaction across all the partitions. They will be vastly slower than single-partition transactions.

Two-phase locking (2PL)

> Two-phase locking (2PL) sounds similar to two-phase *commit* (2PC) but be aware that they are completely different things.

Several transactions are allowed to concurrently read the same object as long as nobody is writing it. When somebody wants to write (modify or delete) an object, exclusive access is required.

Writers don't just block other writers; they also block readers and vice versa. It protects against all the race conditions discussed earlier.

Blocking readers and writers is implemented by a having lock on each object in the database. The lock is used as follows: * if a transaction want sot read an object, it must first acquire a lock in shared mode. * If a transaction wants to write to an object, it must first acquire the lock in exclusive mode. * If a transaction first reads and then writes an object, it may upgrade its shared lock to an exclusive lock. * After a transaction has acquired the lock, it must continue to hold the lock until the end of the transaction (commit or abort). **First phase is when the locks are acquired, second phase is when all the locks are released.**

It can happen that transaction A is stuck waiting for transaction B to release its lock, and vice versa (*deadlock*).

The performance for transaction throughput and response time of queries are significantly worse under two-phase locking than under weak isolation.

A transaction may have to wait for several others to complete before it can do anything.

Databases running 2PL can have unstable latencies, and they can be very slow at high percentiles. One slow transaction, or one transaction that accesses a lot of data and acquires many locks can cause the rest of the system to halt.

Predicate locks

With *phantoms*, one transaction may change the results of another transaction's search query.

In order to prevent phantoms, we need a *predicate lock*. Rather than a lock belonging to a particular object, it belongs to all objects that match some search condition.

Predicate locks applies even to objects that do not yet exist in the database, but which might be added in the future (phantoms).

Index-range locks

Predicate locks do not perform well. Checking for matching locks becomes time-consuming and for that reason most databases implement *index-range locking*.

It's safe to simplify a predicate by making it match a greater set of objects.

These locks are not as precise as predicate locks would be, but since they have much lower overheads, they are a good compromise.

Serializable snapshot isolation (SSI)

It provides full serializability and has a small performance penalty compared to snapshot isolation. SSI is fairly new and might become the new default in the future.

Pesimistic versus optimistic concurrency control

Two-phase locking is called *pessimistic* concurrency control because if anything might possibly go wrong, it's better to wait.

Serial execution is also *pessimistic* as is equivalent to each transaction having an exclusive lock on the entire database.

Serializable snapshot isolation is *optimistic* concurrency control technique. Instead of blocking if something potentially dangerous happens, transactions continue anyway, in the hope that everything will turn out all right. The database is responsible for checking whether anything bad happened. If so, the transaction is aborted and has to be retried.

If there is enough spare capacity, and if contention between transactions is not too high, optimistic concurrency control techniques tend to perform better than pessimistic ones.

SSI is based on snapshot isolation, reads within a transaction are made from a consistent snapshot of the database. On top of snapshot isolation, SSI adds an algorithm for detecting serialization conflicts among writes and determining which transactions to abort.

The database knows which transactions may have acted on an outdated premise and need to be aborted by: * **Detecting reads of a stale MVCC object version.** The database needs to track when a transaction ignores another transaction's writes due to MVCC visibility rules. When a transaction wants to commit, the database checks whether any of the ignored writes have now been committed. If so, the transaction must be aborted. * **Detecting writes that affect prior reads.** As with two-phase locking, SSI uses index-range locks except that it does not block other transactions. When a transaction writes to the database, it must look in the indexes for any other transactions that have recently read the affected data. It simply notifies the transactions that the data they read may no longer be up to date.

Performance of serializable snapshot isolation

Compared to two-phase locking, the big advantage of SSI is that one transaction doesn't need to block waiting for locks held by another transaction. Writers don't block readers, and vice versa.

Compared to serial execution, SSI is not limited to the throughput of a single CPU core. Transactions can read and write data in multiple partitions while ensuring serializable isolation.

The rate of aborts significantly affects the overall performance of SSI. SSI requires that read-write transactions be fairly short (long-running read-only transactions may be okay).

The trouble with distributed systems

Faults and partial failures

A program on a single computer either works or it doesn't. There is no reason why software should be flaky (non deterministic).

In a distributed systems we have no choice but to confront the messy reality of the physical world. There will be parts that are broken in an unpredictable way, while others work. Partial failures are *nondeterministic*. Things will unpredicably fail.

We need to accept the possibility of partial failure and build fault-tolerant mechanism into the software. **We need to build a reliable system from unreliable components.**

Unreliable networks

Focusing on *shared-nothing systems* the network is the only way machines communicate.

The internet and most internal networks are *asynchronous packet networks*. A message is sent and the network gives no guarantees as to when it will arrive, or whether it will arrive at all. Things that could go wrong: 1. Request lost 2. Request waiting in a queue to be delivered later 3. Remote node may have failed 4. Remote node may have temporarily stoped responding 5. Response has been lost on the network 6. The response has been delayed and will be delivered later

If you send a request to another node and don't receive a response, it is *impossible* to tell why.

The usual way of handling this issue is a *timeout*: after some time you give up waiting and assume that the response is not going to arrive.

Nobody is immune to network problems. You do need to know how your software reacts to network problems to ensure that the system can recover from them. It may make sense to deliberately trigger network problems and test the system's response.

If you want to be sure that a request was successful, you need a positive response from the application itself.

If something has gone wrong, you have to assume that you will get no response at all.

Timeouts and unbounded delays

A long timeout means a long wait until a node is declared dead. A short timeout detects faults faster, but carries a higher risk of incorrectly declaring a node dead (when it could be a slowdown).

Premature declaring a node is problematic, if the node is actually alive the action may end up being performed twice.

When a node is declared dead, its responsibilities need to be transferred to other nodes, which places additional load on other nodes and the network.

Network congestion and queueing

- Different nodes try to send packets simultaneously to the same destination, the network switch must queue them and feed them to the destination one by one. The switch will discard packets when filled up.
- If CPU cores are busy, the request is queued by the operative system, until applications are ready to handle it.
- In virtual environments, the operative system is often paused while another virtual machine uses a CPU core. The VM queues the incoming data.
- TCP performs *flow control*, in which a node limits its own rate of sending in order to avoid overloading a network link or the receiving node. This means additional queuing at the sender.

You can choose timeouts experimentally by measuring the distribution of network round-trip times over an extended period.

Systems can continually measure response times and their variability (*jitter*), and automatically adjust timeouts according to the observed response time distribution.

Synchronous vs ashynchronous networks

A telephone network estabilishes a *circuit*, we say is *synchronous* even as the data passes through several routers as it does not suffer from queing. The maximum end-to-end latency of the network is fixed (*bounded delay*).

A circuit is a fixed amount of reserved bandwidth which nobody else can use while the circuit is established, whereas packets of a TCP connection opportunistically use whatever network bandwidth is available.

Using circuits for bursty data transfers wastes network capacity and makes transfer unnecessary slow. By contrast, TCP dinamycally adapts the rate of data transfer to the available network capacity.

We have to assume that network congestion, queueing, and unbounded delays will happen. Consequently, there's no "correct" value for timeouts, they need to be determined experimentally.

Unreliable clocks

The time when a message is received is always later than the time when it is sent, we don't know how much later due to network delays. This makes difficult to determine the order of which things happened when multiple machines are involved.

Each machine on the network has its own clock, slightly faster or slower than the other machines. It is possible to synchronise clocks with Network Time Protocol (NTP).

- **Time-of-day clocks**. Return the current date and time according to some calendar (*wall-clock time*). If the local clock is toof ar ahead of the NTP server, it may be forcibly reset and appear to jump back to a previous point in time. **This makes it is unsuitable for measuring elapsed time.**
- **Monotonic clocks**. Peg: `System.nanoTime()` . They are guaranteed to always move forward. The difference between clock reads can tell you how much time elapsed beween two checks. **The *absolute* value of the clock is meaningless.** NTP allows the clock rate to be speeded up or slowed down by up to 0.05%, but **NTP cannot cause the monotonic clock to jump forward or backward**. **In a distributed system, using a monotonic clock for measuring elapsed time (peg: timeouts), is usually fine**.

If some piece of sofware is relying on an accurately synchronised clock, the result is more likely to be silent and subtle data loss than a dramatic crash.

You need to carefully monitor the clock offsets between all the machines.

Timestamps for ordering events

It is tempting, but dangerous to rely on clocks for ordering of events across multiple nodes. This usually imply that *last write wins* (LWW), often used in both multi-leader replication and leaderless databases like Cassandra and Riak, and data-loss may happen.

The definition of "recent" also depends on local time-of-day clock, which may well be incorrect.

Logical clocks, based on counters instead of oscillating quartz crystal, are safer alternative for ordering events. Logical clocks do not measure time of the day or elapsed time, only relative ordering of events. This contrasts with time-of-the-day and monotic clocks (also known as *physical clocks*).

Clock readings have a confidence interval

It doesn't make sense to think of a clock reading as a point in time, it is more like a range of times, within a confidence internval: for example, 95% confident that the time now is between 10.3 and 10.5.

The most common implementation of snapshot isolation requires a monotonically increasing transaction ID.

Spanner implements snapshot isolation across datacenters by using clock's confidence interval. If you have two confidence internvals where

```
A = [A earliest, A latest]
B = [B earliest, B latest]
```

And those two intervals do not overlap (`A earliest` < `A latest` < `B earliest` < `B latest`), then B definetively happened after A.

Spanner deliberately waits for the length of the confidence interval before commiting a read-write transaction, so their confidence intervals do not overlap.

Spanner needs to keep the clock uncertainty as small as possible, that's why Google deploys a GPS receiver or atomic clock in each datacenter.

Process pauses

How does a node know that it is still leader?

One option is for the leader to obtain a *lease* from other nodes (similar ot a lock with a timeout). It will be the leader until the lease expires; to remain leader, the node must periodically renew the lease. If the node fails, another node can takeover when it expires.

We have to be very careful making assumptions about the time that has passed for processing requests (and holding the lease), as there are many reasons a process would be paused: * Garbage collector (stop the world) * Virtual machine can be suspended * In laptops execution may be suspended * Operating system context-switches * Synchronous disk access * Swapping to disk (paging) * Unix process can be stopped (`SIGSTOP`)

You cannot assume anything about timing

Response time guarantees

There are systems that require software to respond before a specific *deadline* (*real-time operating system, or RTOS*).

Library functions must document their worst-case execution times; dynamic memory allocation may be restricted or disallowed and enormous amount of testing and measurement must be done.

Garbage collection could be treated like brief planned outages. If the runtime can warn the application that a node soon requires a GC pause, the application can stop sending new requests to that node and perform GC while no requests are in progress.

A variant of this idea is to use the garbage collector only for short-lived objects and to restart the process periodically.

Knowledge, truth and lies

A node cannot necessarily trust its own judgement of a situation. Many distributed systems rely on a *quorum* (voting among the nodes).

Commonly, the quorum is an absolute majority of more than half of the nodes.

Fencing tokens

Assume every time the lock server grant sa lock or a lease, it also returns a *fencing token*, which is a number that increases every time a lock is granted (incremented by the lock service). Then we can require every time a client sends a write request to the storage service, it must include its current fencing token.

The storage server remembers that it has already processed a write with a higher token number, so it rejects the request with the last token.

If ZooKeeper is used as lock service, the transaciton ID `zcid` or the node version `cversion` can be used as a fencing token.

Byzantine faults

Fencing tokens can detect and block a node that is *inadvertently* acting in error.

Distributed systems become much harder if there is a risk that nodes may "lie" (*byzantine fault*).

A system is *Byzantine fault-tolerant* if it continues to operate correctly even if some of the nodes are malfunctioning. * Aerospace environments * Multiple participating organisations, some participants may attempt ot cheat or defraud others

Consistency and consensus

The simplest way of handling faults is to simply let the entire service fail. We need to find ways of *tolerating* faults.

Consistency guarantees

Write requests arrive on different nodes at different times.

Most replicated databases provide at least *eventual consistency*. The inconsistency is temporary, and eventually resolves itself (*convergence*).

With weak guarantees, you need to be constantly aware of its limitations. Systems with stronger guarantees may have worse performance or be less fault-tolerant than systems with weaker guarantees.

Linearizability

Make a system appear as if there were only one copy of the data, and all operaitons on it are atomic.

- `read(x) => v` Read from register *x*, database returns value *v*.
- `write(x,v) => r` *r* could be *ok* or *error*.

If one client read returns the new value, all subsequent reads must also return the new value.

- `cas(x_old, v_old, v_new) => r` an atomic *compare-and-set* operation. If the value of the register *x* equals *v_old*, it is atomically set to *v_new*. If `x != v_old` the registers is unchanged and it returns an error.

Serializability: Transactions behave the same as if they had executed *some* serial order.

Linearizability: Recency guarantee on reads and writes of a register (individual object).

Locking and leader election

To ensure that there is indeed only one leader, a lock is used. It must be linearizable: all nodes must agree which nodes owns the lock; otherwise is useless.

Apache ZooKeepr and etcd are often used for distributed locks and leader election.

Constraints and uniqueness guarantees

Unique constraints, like a username or an email address require a situation similiar to a lock.

A hard uniqueness constraint in relational databases requires linearizability.

Implementing linearizable systems

The simplest approach would be to have a single copy of the data, but this would not be able to tolerate faults.

- Single-leader repolication is potentially linearizable.
- Consensus algorithms is linearizable.
- Multi-leader replication is not linearizable.
- Leaderless replication is probably not linearizable.

Multi-leader replication is often a good choice for multi-datacenter replication. On a network interruption betwen data-centers will force a choice between linearizability and availability.

With multi-leader configuraiton, each data center can operate normally with interruptions.

With single-leader replication, the leader must be in one of the datacenters. If the application requires linearizable reads and writes, the network interruption causes the application to become unavailable.

- If your applicaiton *requires* linearizability, and some replicas are disconnected from the other replicas due to a network problem, the some replicas cannot process request while they are disconnected (unavailable).
- If your application *does not require*, then it can be written in a way tha each replica can process requests independently, even if it is disconnected from other replicas (peg: multi-leader), becoming *available*.

If an application does not require linearizability it can be more tolerant of network problems.

The unhelpful CAP theorem

CAP is sometimes presented as *Consistency, Availability, Partition tolerance: pick 2 out of 3*. Or being said in another way *either Consistency or Available when Partitioned*.

CAP only considers one consistency model (linearizability) and one kind of fault (*network partitions*, or nodes that are alive but disconnected from each other). It doesn't say anything about network delays, dead nodes, or other trade-offs. CAP has been historically influential, but nowadays has little practical value for designing systems.

The main reason for dropping linearizability is *performance*, not fault tolerance. Linearizabilit is slow and this is true all the time, not on only during a network fault.

Ordering guarantees

Cause comes before the effect. Causal order in the system is what happened before what (*causally consistent*).

Total order allows any two elements to be compared. Peg, natural numbers are totally ordered.

Some cases one set is greater than another one.

Different consistency models:

- Linearizablity. *total order* of operations: if the system behaves as if there is only a single copy of the data.
- Causality. Two events are ordered if they are causally related. Causality defines *a partial order*, not a total one (incomparable if they are concurrent).

Linearizability is not the only way of preserving causality. **Causal consistency is the strongest possible consistency model that does not slow down due to network delays, and remains available in the face of network failures.**

You need to know which operation *happened before*.

In order to determine the causal ordering, the database needs to know which version of the data was read by the application. **The version number from the prior operation is passed back to the database on a write.**

We can create sequence numbers in a total order that is *consistent with causality*.

With a single-leader replication, the leader can simply increment a counter for each operation, and thus assign a monotonically increasing sequence number to each operation in the replication log.

If there is not a single leader (multi-leader or leaderless database): * Each node can generate its own independent set of sequence numbers. One node can generate only odd numbers and the other only even numbers. * Attach a timestamp from a time-of-day clock. * Preallocate blocks of sequence numbers.

The only problem is that the sequence numbers they generate are *not consistent with causality*. They do not correctly capture ordering of operations across different nodes.

There is simple method for generating sequence numbers that *is* consistent with causality: *Lamport timestamps*.

Each node has a unique identifier, and each node keeps a counter of the number of operations it has processed. The lamport timestamp is then simply a pair of (*counter*, *node ID*). It provides total order, as if you have two timestamps one with a greater counter value is the greater timestamp. If the counter values are the same, the one with greater node ID is the greater timestamp.

Every node and every client keeps track of the *maximum* counter value it has seen so far, and includes that maximum on every request. When a node receives a request of response with a maximum counter value greater than its own counter value, it inmediately increases its own counter to that maximum.

As long as the maximum counter value is carried along with every operation, this scheme ensure that the ordering from the lamport timestamp is consistent with causality.

Total order of oepration only emerges after you have collected all of the operations.

Total order broadcast: * Reliable delivery: If a message is delivered to one node, it is delivered to all nodes. * Totally ordered delivery: Mesages are delivered to every node in the same order.

ZooKeeper and etcd implement total order broadcast.

If every message represents a write to the database, and every replica processes the same writes in the same order, then the replcias will remain consistent with each other (*state machine replication*).

A node is not allowed to retroactgively insert a message into an earlier position in the order if subsequent messages have already been dlivered.

Another way of looking at total order broadcast is that it is a way of creating a *log*. Delivering a message is like appending to the log.

If you have total order broadcast, you can build linearizable storage on top of it.

Because log entries are delivered to all nodes in the same order, if therer are several concurrent writes, all nodes will agree on which one came first. Choosing the first of the conflicting writes as the winner and aborting later ones ensures that all nodes agree on whether a write was commited or aborted.

This procedure ensures linearizable writes, it doesn't guarantee linearizable reads.

To make reads linearizable: * You can sequence reads through the log by appending a message, reading the log, and performing the actual read when the message is delivered back to you (etcd works something like this). * Fetch the position of the latest log message in a linearizable way, you can query that position to be delivered to you, and then perform the read (idea behind ZooKeeper's `sync()`). * You can make your read from a replica that is synchronously updated on writes.

For every message you want to send through total order broadcast, you increment-and-get the linearizable integer and then attach the value you got from the register as a sequence number to the message. YOu can send the message to all nodes, and the recipients will deliver the message consecutively by sequence number.

Distributed transactions and consensus

Basically *getting several nodes to agree on something*.

There are situations in which it is important for nodes to agree: * Leader election: All nodes need to agree on which node is the leader. * Atomic commit: Get all nodes to agree on the outcome of the transacction, either they all abort or roll back.

Atomic commit and two-phase commit (2PC)

A transaction either succesfully *commit*, or *abort*. Atomicity prevents half-finished results.

On a single node, transaction commitment depends on the *order* in which data is writen to disk: first the data, then the commit record.

2PC uses a coordinartor (*transaction manager*). When the application is ready to commit, the coordinator begins phase 1: it sends a *prepare* request to each of the nodes, asking them whether are able to commit.

- If all participants reply "yes", the coordinator sends out a *commit* request in phase 2, and the commit takes place.
- If any of the participants replies "no", the coordinator sends an *abort* request to all nodes in phase 2.

When a participant votes "yes", it promises that it will definitely be able to commit later; and once the coordiantor decides, that decision is irrevocable. Those promises ensure the atomicity of 2PC.

If one of the participants or the network fails during 2PC (prepare requests fail or time out), the coordinator aborts the transaction. If any of the commit or abort request fail, the coordinator retries them indefinitely.

If the coordinator fails before sending the prepare requests, a participant can safely abort the transaction.

The only way 2PC can complete is by waiting for the coordinator to revover in case of failure. This is why the coordinator must write its commit or abort decision to a transaction log on disk before sending commit or abort requests to participants.

Three-phase commit

2PC is also called a *blocking* atomic commit protocol, as 2Pc can become stuck waiting for the coordinator to recover.

There is an alternative called *three-phase commit* (3PC) that requires a *perfect failure detector*.

Distributed transactions carry a heavy performance penalty due the disk forcing in 2PC required for crash recovery and additional network round-trips.

XA (X/Open XA for eXtended Architecture) is a standard for implementing two-phase commit across heterogeneous technologies. Supported by many traditional relational databases (PostgreSQL, MySQL, DB2, SQL Server, and Oracle) and message brokers (ActiveMQ, HornetQ, MSQMQ, and IBM MQ).

The problem with *locking* is that database transactions usually take a row-level exclusive lock on any rows they modify, to prevent dirty writes.

While those locks are held, no other transaction can modify those rows.

When a coordinator fails, *orphaned* in-doubt transactions do ocurr, and the only way out is for an administrator to manually decide whether to commit or roll back the transaction.

Fault-tolerant consensus

One or more nodes may *propose* values, and the consensus algorithm *decides* on those values.

Consensus algorithm must satisfy the following properties: * Uniform agreement: No two nodes decide differently. * Integrity: No node decides twice. * Validity: If a node decides the value *v*, then *v* was proposed by some node. * Termination: Every node that does not crash eventually decides some value.

If you don't care about fault tolerance, then satisfying the first three properties is easy: you can just hardcode one node to be the "dictator" and let that node make all of the decisions.

The termination property formalises the idea of fault tolerance. Even if some nodes fail, the other nodes must still reach a decision. Termination is a liveness property, whereas the other three are safety properties.

The best-known fault-tolerant consensus algorithms are Viewstamped Replication (VSR), Paxos, Raft and Zab.

Total order broadcast requires messages to be delivered exactly once, in the same order, to all nodes.

So total order broadcast is equivalent to repeated rounds of consensus: * Due to agreement property, all nodes decide to deliver the same messages in the same order. * Due to integrity, messages are not duplicated. * Due to validity, messages are not corrupted. * Due to termination, messages are not lost.

Single-leader replication and consensus

All of the consensus protocols dicussed so far internally use a leader, but they don't guarantee that the lader is unique. Protocols define an *epoch number* (*ballot number* in Paxos, *view number* in Viewstamped Replication, and *term number* in Raft). Within each epoch, the leader is unique.

Every time the current leader is thought to be dead, a vote is started among the nodes to elect a new leader. This election is given an incremented epoch number, and thus epoch numbers are totallly ordered and monotonically increasing. If there is a conflic, the leader with the higher epoch number prevails.

A node cannot trust its own judgement. It must collect votes from a *quorum* of nodes. For every decision that a leader wants to make, it must send the proposed value to the other nodes and wait for a quorum of nodes to respond in favor of the proposal.

There are two rounds of voting, once to choose a leader, and second time to vote on a leader's proposal. The quorums for those two votes must overlap.

The biggest difference with 2PC, is that 2PC requires a "yes" vote for *every* participant.

The benefits of consensus come at a cost. The process by which nodes vote on proposals before they are decided is kind of synchronous replication.

Consensus always require a strict majority to operate.

Most consensus algorithms assume a fixed set of nodes that participate in voting, which means that you can't just add or remove nodes in the cluster. *Dynamic membership* extensions are much less well understood than static membership algorithms.

Consensus systems rely on timeouts to detect failed nodes. In geographically distributed systems, it often happens that a node falsely believes the leader to have failed due to a network issue. This implies frequest leader elecctions resulting in terrible performance, spending more time choosing a leader than doing any useful work.

Membership and coordination services

ZooKeeper or etcd are often described as "distributed key-value stores" or "coordination and configuration services".

They are designed to hold small amounts of data that can fit entirely in memory, you wouldn't want to store all of your application's data here. Data is replicated across all the nodes using a fault-tolerant total order broadcast algorithm.

ZooKeeper is modeled after Google's Chubby lock service and it provides some useful features: * Linearizable atomic operations: Usuing an atomic compare-and-set operation, you can implement a lock. * Total ordering of operations: When some resource is protected by a lock or lease, you need a *fencing token* to prevent clients from conflicting with each other in the case of a process pause. The fencing token is some number that monotonically increases every time the lock is acquired. * Failure detection: Clients maintain a long-lived session on ZooKeeper servers. When a ZooKeeper node fails, the session remains active. When ZooKeeper declares the session to be dead all locks held are automatically released. * Change notifications: Not only can one client read locks and values, it can also watch them for changes.

ZooKeeper is super useful for distributed coordination.

ZooKeeper/Chubby model works well when you have several instances of a process or service, and one of them needs to be chosen as a leader or primary. If the leader fails, one of the other nodes should take over. This is useful for single-leader databases and for job schedulers and similar stateful systems.

ZooKeeper runs on a fixed number of nodes, and performs its majority votes among those nodes while supporting a potentially large number of clients.

The kind of data managed by ZooKeeper is quite slow-changing like "the node running on 10.1.1.23 is the leader for partition 7". It is not intended for storing the runtime state of the application. If application state needs to be replicated there are other tools (like Apache BookKeeper).

ZooKeeper, etcd, and Consul are also often used for *service discovery*, find out which IP address you need to connect to in order to reach a particular service. In cloud environments, it is common for virtual machines to continually come an go, you often don't know the IP addresses of your services ahead of time. Your services when they start up they register their network endpoints ina service registry, where they can then be found by other services.

ZooKeeper and friends can be seen as part of a long history of research into *membership services*, determining which nodes are currently active and live members of a cluster.

Batch processing

- Service (online): waits for a request, sends a response back
- Batch processing system (offline): takes a large amount of input data, runs a *job* to process it, and produces some output.
- Stream processing systems (near-real-time): a stream processor consumes input and produces outputs. A stream job operates on events shortly after they happen.

Batch processing with Unix tools

We can build a simple log analysis job to get the five most popular pages on your site

```
cat /var/log/nginx/access.log |
  awk '{print $7}' |
  sort             |
  uniq -c          |
  sort -r -n       |
  head -n 5        |
```

You could write the same thing with a simpel program.

The difference is that with Unix commands automatically handle larger-than-memory datasets and automatically paralelizes sorting across multiple CPU cores.

Programs must have the same data format to pass information to one another. In Unix, that interface is a file (file descriptor), an ordered sequence of bytes.

By convention Unix programs treat this sequence of bytes as ASCII text.

The unix approach works best if a program simply uses `stdin` and `stdout`. This allows a shell user to wire up the input and output in whatever way they want; the program doesn't know or care where the input is coming from and where the output is going to.

Part of what makes Unix tools so successful is that they make it quite easy to see what is going on.

Map reduce and distributed filesystems

A single MapReduce job is comparable to a single Unix process.

Running a MapReduce job normally does not modify the input and does not have any side effects other than producing the output.

While Unix tools use `stdin` and `stdout` as input and output, MapReduce jobs read and write files on a distributed filesystem. In Hadoop, that filesystem is called HDFS (Haddoop Distributed File System).

HDFS is based on the *shared-nothing* principe. Implemented by centralised storage appliance, often using custom hardware and special network infrastructure.

HDFS consists of a daemon process running on each machine, exposing a network service that allows other nodes to access files stored on that machine. A central server called the *NameNode* keeps track of which file blocks are stored on which machine.

File blocks are replciated on multiple machines. Reaplication may mean simply several copies of the same data on multiple machines, or an *erasure coding* scheme such as Reed-Solomon codes, which allow lost data to be recovered.

MapReduce is a programming framework with which you can write code to process large datasets in a distributed filesystem like HDFS. 1. Read a set of input files, and break it up into *records*. 2. Call the mapper function to extract a key and value from each input record. 3. Sort all of the key-value pairs by key. 4. Call the reducer function to iterate over the sorted key-value pairs.

- Mapper: Called once for every input record, and its job is to extract the key and value from the input record.
- Reducer: Takes the key-value pairs produced by the mappers, collects all the values belonging to the same key, and calls the reducer with an interator over that collection of vaues.

MapReduce can parallelise a computation across many machines, without you having ot write code to explicitly handle the parallelism. THe mapper and reducer only operate on one record at a time; they don't need to know where their input is coming from or their output is going to.

In Hadoop MapReduce, the mapper and reducer are each a Java class that implements a particular interface.

The MapReduce scheduler tries to run each mapper on one of the machines that stores a replica of the input file, *putting the computation near the data*.

The reduce side of the computation is also partitioned. While the number of map tasks is determined by the number of input file blocks, the number of reduce tasks is configured by the job author. To ensure that all key-value pairs with the same key end up in the same reducer, the framework uses a hash of the key.

The dataset is likely too large to be sorted with a conventional sorting algorithm on a single machine. Sorting is performed in stages.

Whenever a mapper finishes reading its input file and writing its sorted output files, the MapReduce scheduler notifies the reducers that they can start fetching the output files from that mapper. The reducers connect to each of the mappers and download the files of sorted key-value pairs for their partition. Partitioning by reducer, sorting and copying data partitions from mappers to reducers is called *shuffle*.

The reduce task takes the files from the mappers and merges them together, preserving the sort order.

MapReduce jobs can be chained together into *workflows*, the output of one job becomes the input to the next job. In Hadoop this chaining is done implicitly by directory name: the first job writes its output to a designated directory in HDFS, the second job reads that same directory name as its input.

Compared with the Unix example, it could be seen as in each sequence of commands each command output is written to a temporary file, and the next command reads from the temporary file.

It is common in datasets for one record to have an association with another record: a *foreign key* in a relational model, a *document reference* in a document model, or an *edge* in graph model.

If the query involves joins, it may require multiple index lookpus. MapReduce has no concept of indexes.

When a MapReduce job is given a set of files as input, it reads the entire content of all of those files, like a *full table scan*.

In analytics it is common to want to calculate aggregates over a large number of records. Scanning the entire input might be quite reasonable.

In order to achieve good throughput in a batch process, the computation must be local to one machine. Requests over the network are too slow and nondeterministic. Queries to other database for example would be prohibitive.

A better approach is to take a copy of the data (peg: the database) and put it in the same distributed filesystem.

MapReduce programming model has separated the physical network communication aspects of the computation (getting the data to the right machine) from the application logic (processing the data once you have it).

In an example of a social network, small number of celebrities may have many millions of followers. Such disproportionately active database records are known as *linchpin objects* or *hot keys*.

A single reducer can lead to significant *skew* that is, one reducer that must process significantly more records than the others.

The *skewed join* method in Pig first runs a sampling job to determine which keys are hot and then records related to the hot key need to be replicated to *all* reducers handling that key.

Handling the hot key over several reducers is called *shared join* method. In Crunch is similar but requires the hot keys to be specified explicitly.

Hive's skewed join optimisation requries hot keys to be specified explicitly and it uses map-side join. If you *can* make certain assumptions about your input data, it is possible to make joins faster. A MapReducer job with no reducers and no sorting, each mapper simply reads one input file and writes one output file.

The output of a batch process is often not a report, but some other kind of structure.

Google's original use of MapReduce was to build indexes for its search engine. Hadoop MapReduce remains a good way of building indexes for Lucene/Solr.

If you need to perform a full-text search, a batch process is very effective way of building indexes: the mappers partition the set of documents as needed, each reducer builds the index for its partition, and the index files are written to the distributed filesystem. It pararellises very well.

Machine learning systems such as clasifiers and recommendation systems are a common use for batch processing.

Key-value stores as batch process output

The output of those batch jobs is often some kind of database.

So, how does the output from the batch process get back into a database?

Writing from the batch job directly to the database server is a bad idea: * Making a network request for every single record is magnitude slower than the normal throughput of a batch task. * Mappers or reducers concurrently write to the same output database an it can be easily overwhelmed. * You have to worry about the results from partially completed jobs being visible to other systems.

A much better solution is to build a brand-new database *inside* the batch job an write it as files to the job's output directory, so it can be loaded in bulk into servers that handle read-only queries. Various key-value stores support building database files in MapReduce including Voldemort, Terrapin, ElephanDB and HBase bulk loading.

By treating inputs as immutable and avoiding side effects (such as writing to external databases), batch jobs not only achieve good performance but also become much easier to maintain.

Design principles that worked well for Unix also seem to be working well for Hadoop.

The MapReduce paper was not at all new. The sections we've seen had been already implemented in so-called *massively parallel processing* (MPP) databases.

The biggest difference is that MPP databases focus on parallel execution of analytic SQL queries on a cluster of machines, while the combination of MapReduce and a distributed filesystem provides something much more like a general-purpose operating system that can run arbitraty programs.

Hadoop opened up the possibility of indiscriminately dumpint data into HDFS. MPP databases typically require careful upfront modeling of the data and query patterns before importing data into the database's proprietary storage format.

In MapReduce instead of forcing the producer of a dataset to bring it into a standarised format, the interpretation of the data becomes the consumer's problem.

If you have HDFS and MapReduce, you *can* build a SQL query execution engine on top of it, and indeed this is what the Hive project did.

If a node crashes while a query is executing, most MPP databases abort the entire query. MPP databases also prefer to keep as much data as possible in memory.

MapReduce can tolerate the failure of a map or reduce task without it affecting the job. It is also very eager to write data to disk, partly for fault tolerance, and partly because the dataset might not fit in memory anyway.

MapReduce is more appropriate for larger jobs.

At Google, a MapReduce task that runs for an hour has an approximately 5% risk of being terminated to make space for higher-priority process.

Ths is why MapReduce is designed to tolerate frequent unexpected task termination.

Beyond MapReduce

In response to the difficulty of using MapReduce directly, various higher-level programming models emerged on top of it: Pig, Hive, Cascading, Crunch.

MapReduce has poor performance for some kinds of processing. It's very robust, you can use it to process almost arbitrarily large quantities of data on an unreliable multi-tenant system with frequent task terminations, and it will still get the job done.

The files on the distributed filesystem are simply *intermediate state*: a means of passing data from one job to the next.

The process of writing out the intermediate state to files is called *materialisation*.

MapReduce's approach of fully materialising state has some downsides compared to Unix pipes:

- A MapReduce job can only start when all tasks in the preceding jobs have completed, whereas rocesses connected by a Unix pipe are started at the same time.
- Mappers are often redundant: they just read back the same file that was just written by a reducer.
- Files are replicated across several nodes, which is often overkill for such temporary data.

To fix these problems with MapReduce, new execution engines for distributed batch computations were developed, Spark, Tez and Flink. These new ones can handle an entire workflow as one job, rather than breaking it up into independent subjobs (*dataflow engines*).

These functions need not to take the strict roles of alternating map and reduce, they are assembled in flexible ways, in functions called *operators*.

Spark, Flink, and Tex avoid writing intermediate state to HDFS, so they take a different approach to tolerating faults: if a machine fails and the intermediate state on that machine is lost, it is recomputed from other data that is still available.

The framework must keep track of how a given piece of data was computed. Spark uses the resilient distributed dataset (RDD) to track ancestry data, while Flink checkpoints operator state, allowing it to resume running an operator that ran into a fault during its execution.

Graphs and iterative processing

It's interesting to look at graphs in batch processing context, where the goal is to perform some kind of offline processing or analysis on an entire graph. This need often arises in machine learning applications such as recommednation engines, or in ranking systems.

"repeating until done" cannot be expressed in plain MapReduce as it runs in a single pass over the data and some extra trickery is necessary.

An optimisation for batch processing graphs, the *bulk synchronous parallel* (BSP) has become popular. It is implemented by Apache Giraph, Spark's GraphX API, and Flink's Gelly API (_Pregel model, as Google Pregel paper popularised it).

One vertex can "send a message" to another vertex, and typically those messages are sent along the edges in a graph.

The difference from MapReduce is that a vertex remembers its state in memory from one iteration to the next.

The fact that vertices can only communicate by message passing helps improve the performance of Pregel jobs, since messages can be batched.

Fault tolerance is achieved by periodically checkpointing the state of all vertices at the end of an interation.

The framework may partition the graph in arbitrary ways.

Graph algorithms often have a lot of cross-machine communication overhead, and the intermediate state is often bigger than the original graph.

If your graph can fit into memory on a single computer, it's quite likely that a single-machine algorithm will outperform a distributed batch process. If the graph is too big to fit on a single machine, a distributed approach such as Pregel is unavoidable.

Stream processing

We can run the processing continuously, abandoning the fixed time slices entirely and simply processing every event as it happens, that's the idea behind *stream processing*. Data that is incrementally made available over time.

Transmitting event streams

A record is more commonly known as an *event*. Something that happened at some point in time, it usually contains a timestamp indicating when it happened acording to a time-of-day clock.

An event is generated once by a *producer* (*publisher* or *sender*), and then potentially processed by multiple *consumers* (*subcribers* or *recipients*). Related events are usually grouped together into a *topic* or a *stream*.

A file or a database is sufficient to connect producers and consumers: a producer writes every event that it generates to the datastore, and each consumer periodically polls the datastore to check for events that have appeared since it last ran.

However, when moving toward continual processing, polling becomes expensive. It is better for consumers to be notified when new events appear.

Databases offer *triggers* but they are limited, so specialised tools have been developed for the purpose of delivering event notifications.

Messaging systems

Direct messaging from producers to consumers

Within the *publish*/*subscribe* model, we can differentiate the systems by asking two questions: 1. *What happens if the producers send messages faster than the consumers can process them?* The system can drop messages, buffer the messages in a queue, or apply *backpressure* (*flow control*, blocking the producer from sending more messages). 2. *What happens if nodes crash or temporarily go offline, are any messages lost?* Durability may require some combination of writing to disk and/or replication.

A number of messaging systems use direct communication between producers and consumers without intermediary nodes: * UDP multicast, where low latency is important, application-level protocols can recover lost packets. * Brokerless messaging libraries such as ZeroMQ * StatsD and Brubeck use unreliable UDP messaging for collecting metrics * If the consumer expose a service on the network, producers can make a direct HTTP or RPC request to push messages to the consumer. This is the idea behind webhooks, a callback URL of one service is registered with another service, and makes a request to that URL whenever an event occurs

These direct messaging systems require the application code to be aware of the possibility of message loss. The faults they can tolerate are quite limited as they assume that producers and consumers are constantly online.

If a consumer if offline, it may miss messages. Some protocols allow the producer to retry failed message deliveries, but it may break down if the producer crashes losing the buffer or messages.

Message brokers

An alternative is to send messages via a *message broker* (or *message queue*), which is a kind of database that is optimised for handling message streams. It runs as a server, with producers and consumers connecting to it as clients. Producers write messages to the broker, and consumers receive them by reading them from the broker.

By centralising the data, these systems can easily tolerate clients that come and go, and the question of durability is moved to the broker instead. Some brokers only keep messages in memory, while others write them down to disk so that they are not lost inc ase of a broker crash.

A consequence of queueing is that consuemrs are generally *asynchronous*: the producer only waits for the broker to confirm that it has buffered the message and does not wait for the message to be processed by consumers.

Some brokers can even participate in two-phase commit protocols using XA and JTA. This makes them similar to databases, aside some practical differences: * Most message brokers automatically delete a message when it has been successfully delivered to its consumers. This makes them not suitable for long-term storage. * Most message brokers assume that their working set is fairly small. If the broker needs to buffer a lot of messages, each individual message takes longer to process, and the overall throughput may degrade. * Message brokers often support some way of subscribing to a subset of topics matching some pattern. * Message brokers do not support arbitrary queries, but they do notify clients when data changes.

This is the traditional view of message brokers, encapsulated in standards like JMS and AMQP, and implemented in RabbitMQ, ActiveMQ, HornetQ, Qpid, TIBCO Enterprise Message Service, IBM MQ, Azure Service Bus, and Google Cloud Pub/Sub.

When multiple consumers read messages in the same topic, to main patterns are used: * Load balancing: Each message is delivered to *one* of the consumers. The broker may assign messages to consumers arbitrarily. * Fan-out: Each message is delivered to *all* of the consumers.

In order to ensure that the message is not lost, message brokers use *acknowledgements*: a client must explicitly tell the broker when it has finished processing a message so that the broker can remove it from the queue.

The combination of laod balancing with redelivery inevitably leads to messages being reordered. To avoid this issue, youc an use a separate queue per consumer (not use the load balancing feature).

Partitioned logs

A key feature of barch process is that you can run them repeatedly without the risk of damaging the input. This is not the case with AMQP/JMS-style messaging: receiving a message is destructive if the acknowledgement causes it to be deleted from the broker.

If you add a new consumer to a messaging system, any prior messages are already gone and cannot be recovered.

We can have a hybrid, combining the durable storage approach of databases with the low-latency notifications facilities of messaging, this is the idea behind *log-based message brokers*.

A log is simply an append-only sequence of records on disk. The same structure can be used to implement a message broker: a producer sends a message by appending it to the end of the log, and consumer receives messages by reading the log sequentially. If a consumer reaches the end of the log, it waits for a notification that a new message has been appended.

To scale to higher throughput than a single disk can offer, the log can be *partitioned*. Different partitions can then be hosted on different machines. A topic can then be defined as a group of partitions that all carry messages of the same type.

Within each partition, the broker assigns monotonically increasing sequence number, or *offset*, to every message.

Apache Kafka, Amazon Kinesis Streams, and Twitter's DistributedLog, are log-based message brokers that work like this.

The log-based approach trivially supports fan-out messaging, as several consumers can independently read the log reading without affecint each other. Reading a message does not delete it from the log. To eachieve load balancing the broker can assign entire partitions to nodes in the consumer group. Each client then consumes *all* the messages in the partition it has been assigned. This approach has some downsides. * The number of nodes sharing the work of consuming a topic can be at most the number of log partitions in that topic. * If a single message is slow to process, it holds up the processing of subsequent messages in that partition.

In situations where messages may be expensive to process and you want to pararellise processing on a message-by-message basis, and where message ordering is not so important, the JMS/AMQP style of message broker is preferable. In situations with high message throughput, where each message is fast to process and where message ordering is important, the log-based approach works very well.

It is easy to tell which messages have been processed: al messages with an offset less than a consumer current offset have already been processed, and all messages with a greater offset have not yet been seen.

The offset is very similar to the *log sequence number* that is commonly found in single-leader database replication. The message broker behaves like a leader database, and the consumer like a follower.

If a consumer node fails, another node in the consumer group starts consuming messages at the last recorded offset. If the consumer had processed subsequent messages but not yet recorded their offset, those messages will be processed a second time upon restart.

If you only ever append the log, you will eventually run out of disk space. From time to time old segments are deleted or moved to archive.

If a slow consumer cannot keep with the rate of messages, and it falls so far behind that its consumer offset poitns to a deleted segment, it will miss some of the messages.

The throughput of a log remains more or less constant, since every message is written to disk anyway. This is in contrast to messaging systems that keep messages in memory by default and only write them to disk if the queue grows too large: systems are fast when queues are short and become much slower when they start writing to disk, throughput depends on the amount of history retained.

If a consumer cannot keep up with producers, the consumer can drop messages, buffer them or applying backpressure.

You can monitor how far a consumer is behind the head of the log, and raise an alert if it falls behind significantly.

If a consumer does fall too far behind and start missing messages, only that consumer is affected.

With AMQP and JMS-style message brokers, processing and acknowledging messages is a destructive operation, since it causes the messages to be deleted on the broker. In a log-based message broker, consuming messages is more like reading from a file.

The offset is under the consumer's control, so you can easily be manipulated if necessary, like for replaying old messages.

Databases and streams

A replciation log is a stream of a database write events, produced by the leader as it processes transactions. Followers apply that stream of writes to their own copy of the database and thus end up with an accurate copy of the same data.

If periodic full database dumps are too slow, an alternative that is sometimes used is *dual writes*. For example, writing to the database, then updating the search index, then invalidating the cache.

Dual writes have some serious problems, one of which is race conditions. If you have concurrent writes, one value will simply silently overwrite another value.

One of the writes may fail while the other succeeds and two systems will become inconsistent.

The problem with most databases replication logs is that they are considered an internal implementation detail, not a public API.

Recently there has been a growing interest in *change data capture* (CDC), which is the process of observing all data changes written to a database and extracting them in a form in which they can be replicated to other systems.

For example, you can capture the changes in a database and continually apply the same changes to a search index.

We can call log consumers *derived data systems*: the data stored in the search index and the data warehouse is just another view. Change data capture is a mechanism for ensuring that all changes made to the system of record are also reflected in the derived data systems.

Change data capture makes one database the leader, and turns the others into followers.

Database triggers can be used to implement change data capture, but they tend to be fragile and have significant performance overheads. Parsing the replication log can be a more robust approach.

LinkedIn's Databus, Facebook's Wormhole, and Yahoo!'s Sherpa use this idea at large scale. Bottled Watter implements CDC for PostgreSQL decoding the write-ahead log, Maxwell and Debezium for something similar for MySQL by parsing the binlog, Mongoriver reads the MongoDB oplog, and GoldenGate provide similar facilities for Oracle.

Keeping all changes forever would require too much disk space, and replaying it would take too long, so the log needs to be truncated.

You can start with a consistent snapshot of the database, and it must correspond to a known position or offset in the change log.

The storage engine periodically looks for log records with the same key, throws away any duplicates, and keeps only the most recent update for each key.

An update with a special null value (a *tombstone*) indicates that a key was deleted.

The same idea works in the context of log-based mesage brokers and change data capture.

RethinkDB allows queries to subscribe to notifications, Firebase and CouchDB provide data synchronisation based on change feed.

Kafka Connect integrates change data capture tools for a wide range of database systems with Kafka.

Event sourcing

There are some parallels between the ideas we've discussed here and *event sourcing*.

Similarly to change data capture, event sourcing involves storing all changes to the application state as a log of change events. Event sourcing applyies the idea at a different level of abstraction.

Event sourcing makes it easier to evolve applications over time, helps with debugging by making it easier to understand after the fact why something happened, and guards against application bugs.

Specialised databases such as Event Store have been developed to support applications using event sourcing.

Applications that use event sourcing need to take the log of evetns and transform it into application state that is suitable for showing to a user.

Replying the event log allows you to reconstruct the current state of the system.

Applications that use event sourcing typically have some mechanism for storing snapshots.

Event sourcing philosophy is careful to distinguis between *events* and *commands*. When a request from a user first arrives, it is initially a command: it may still fail (like some integrity condition is violated). If the validation is successful, it becomes an event, which is durable and immutable.

A consumer of the event stream is not allowed to reject an event: Any validation of a command needs to happen synchronously, before it becomes an event. For example, by using a serializable transaction that atomically validates the command and publishes the event.

Alternatively, the user request to serve a seat could be split into two events: first a tentative reservation, and then a separate confirmation event once the reservation has been validated. This split allows the validation to take place in an asynchronous process.

Whenever you have state changes, that state is the result of the events that mutated it over time.

Mutable state and an append-only log of immutable events do not contradict each other.

As an example, financial bookkeeping is recorded as an append-only *ledger*. It is a log of events describing money, good, or services that have changed hands. Profit and loss or the balance sheet are derived from the ledger by adding them up.

If a mistake is made, accountants don't erase or change the incorrect transaction, instead, they add another transaction that compensates for the mistake.

If buggy code writes bad data to a database, recovery is much harder if the code is able to destructively overwrite data.

Immutable events also capture more information than just the current state. If you persisted a cart into a regular database, deleting an item would effectively loose that event.

You can derive views from the same event log, Druid ingests directly from Kafka, Pistachio is a distributed key-value sotre that uses Kafka as a commit log, Kafka Connect sinks can export data from Kafka to various different databases and indexes.

Storing data is normally quite straightforward if you don't have to worry about how it is going to be queried and accessed. You gain a lot of flexibility by separating the form in which data is written from the form it is read, this idea is known as *command query responsibility segregation* (CQRS).

There is this fallacy that data must be written in the same form as it will be queried.

The biggest downside of event sourcing and change data capture is that consumers of the event log are usually asynchronous, a user may make a write to the log, then read from a log derived view and find that their write has not yet been reflected.

The limitations on immutable event history depends on the amount of churn in the dataset. Some workloads mostly add data and rarely update or delete; they are wasy to make immutable. Other workloads have a high rate of updates and deletes on a comparaively small dataset; in these cases immutable history becomes an issue because of fragmentation, performance compaction and garbage collection.

There may also be circumstances in which you need data to be deleted for administrative reasons.

Sometimes you may want to rewrite history, Datomic calls this feature *excision*.

Processing Streams

What you can do with the stream once you have it: 1. You can take the data in the events and write it to the database, cache, search index, or similar storage system, from where it can thenbe queried by other clients. 2. You can push the events to users in some way, for example by sending email alerts or push notifications, or to a real-time dashboard. 3. You can process one or more input streams to produce one or more output streams.

Processing streams to produce other, derived streams is what an *operator job* does. The one crucial difference to batch jobs is that a stream never ends.

Complex event processing (CEP) is an approach for analising event streams where you can specify rules to search for certain patterns of events in them.

When a match is found, the engine emits a *complex event*.

Queries are stored long-term, and events from the input streams continuously flow past them in search of a query that matches an event pattern.

Implementations of CEP include Esper, IBM InfoSphere Streams, Apama, TIBCO StreamBase, and SQLstream.

The boundary between CEP and stream analytics is blurry, analytics tends to be less interested in finding specific event sequences and is more oriented toward aggregations and statistical metrics.

Frameworks with analytics in mind are: Apache Storm, Spark Streaming, Flink, Concord, Samza, and Kafka Streams. Hosted services include Google Cloud Dataflow and Azure Stream Analytics.

Sometimes there is a need to search for individual events continually, such as full-text search queries over streams.

Message-passing ystems are also based on messages and events, we normally don't think of them as stream processors.

There is some crossover area between RPC-like systems and stream processing. Apache Storm has a feature called *distributed RPC*.

In a batch process, the time at which the process is run has nothing to do with the time at which the events actually occurred.

Many stream processing frameworks use the local system clock on the processing machine (*processing time*) to determine windowing. It is a simple approach that breaks down if there is any significant processing lag.

Confusing event time and processing time leads to bad data. Processing time may be unreliable as the stream processor may queue events, restart, etc. It's better to take into account the original event time to count rates.

You can never be sure when you have received all the events.

You can time out and declare a window ready after you have not seen any new events for a while, but it could still happen that some events are delayed due a network interruption. You need to be able to handle such *stranggler* events that arrive after the window has already been declared complete.

1. You can ignore the stranggler events, tracking the number of dropped events as a metric.
2. Publish a *correction*, an updated value for the window with stranglers included. You may also need to retrat the previous output.

To adjust for incofrrect device clocks, one approach is to log three timestamps: * The time at which the event occurred, according to the device clock * The time at which the event was sent to the server, according to the device clock * The time at which the event was received by the server, according to the server clock.

You can estimate the offset between the device clock and the server clock, then apply that offset to the event timestamp, and thus estimate the true time at which the event actually ocurred.

Several types of windows are in common use: * Tumbling window: Fixed length. If you have a 1-minute tumbling window, all events between 10:03:00 and 10:03:59 will be grouped in one window, next window would be 10:04:00-10:04:59 * Hopping window: Fixed length, but allows windows to overlap in order to provide some smoothing. If you have a 5-minute window with a hop size of 1 minute, it would contain the events between 10:03:00 and 10:07:59, next window would cover 10:04:00-10:08:59 * Sliding window: Events that occur within some interval of each other. For example, a 5-minute sliding window would cover 10:03:39 and 10:08:12 because they are less than 4 minutes apart. * Session window: No fixed duration. All events for the same user, the window ends when the user has been inactive for some time (30 minutes). Common in website analytics

The fact that new events can appear anytime on a stream makes joins on stream challenging.

Stream-stream joins

You want to detect recent trends in searched-for URLs. You log an event containing the query. Someone clicks one of the search results, you log another event recording the click. You need to bring together the events for the search action and the click action.

For this type of join, a stream processor needs to maintain *state*: All events that occurred in the last hour, indexed by session ID. Whenever a search event or click event occurs, it is added to the appropriate index, and the stream processor also checks the other index to see if another event for the same session ID has already arrived. If there is a matching event, you emit an event saying search result was clicked.

Stream-table joins

Sometimes know as *enriching* the activity events with information from the database.

Imagine two datasets: a set of usr activity events, and a database of user profiles. Activity events include the user ID, and the the resulting stream should have the augmented profile information based upon the user ID.

The stream process needs to look at one activity event at a time, look up the event's user ID in the database, and add the profile information to the activity event. THe database lookup could be implemented by querying a remote database., however this would be slow and risk overloading the database.

Another approach is to load a copy of the database into the stream processor so that it can be queried locally without a network round-trip. The stream processor's local copy of the database needs to be kept up to date; this can be solved with change data capture.

Table-table join

The stream process needs to maintain a database containing the set of followers for each user so it knows which timelines need to be updated when a new tweet arrives.

Time-dependence join

The previous three types of join require the stream processor to maintain some state.

If state changes over time, and you join with some state, what point in time do you use for the join?

If the ordering of events across streams is undetermined, the join becomes nondeterministic.

This issue is known as *slowly changing dimension* (SCD), often addressed by using a unique identifier for a particular version of the joined record. For example, we can turn the system deterministic if every time the tax rate changes, it is given a new identifier, and the invoice includes the identifier for the tax rate at the time of sale. But as a consequence makes log compation impossible.

Fault tolerance

Batch processing frameworks can tolerate faults fairly easy:if a task in a MapReduce job fails, it can simply be started again on another machine, input files are immutable and the output is written to a separate file.

Even though restarting tasks means records can be processed multiple times, the visible effect in the output is as if they had only been processed once (*exactly-once-semantics* or *effectively-once*).

With stream processing waiting until a tasks if finished before making its ouput visible is not an option, stream is infinite.

One solution is to break the stream into small blocks, and treat each block like a minuature batch process (*micro-batching*). This technique is used in Spark Streaming, and the batch size is typically around one second.

An alternative approach, used in Apache Flint, is to periodically generate rolling checkpoints of state and write them to durable storage. If a stream operator crashes, it can restart from its most recent checkpoint.

Microbatching and chekpointing approaches provide the same exactly-once semantics as batch processing. However, as soon as output leaves the stream processor, the framework is no longer able to discard the output of a failed batch.

In order to give appearance of exactly-once processing, things either need to happen atomically or none of must happen. Things should not go out of sync of each other. Distributed transactions and two-phase commit can be used.

This approach is used in Google Cloud Dataflow and VoltDB, and there are plans to add similar features to Apache Kafka.

Our goal is to discard the partial output of failed tasks so that they can be safely retired without taking effect twice. Distributed transactions are one way of achieving that goal, but another way is to rely on *idempotence*.

An idempotent operation is one that you can perform multiple times, and it has the same effect as if you performed it only once.

Even if an operation is not naturally idempotent, it can often be made idempotent with a bit of extra metadata. You can tell wether an update has already been applied.

Idempotent operations can be an effective way of achieving exactly-once semantics with only a small overhead.

Any stream process that requires state must ensure tha this state can be recovered after a failure.

One option is to keep the state in a remote datastore and replicate it, but it is slow.

An alternative is to keep state local to the stream processor and replicate it periodically.

Flink periodically captures snapshots and writes them to durable storage such as HDFS; Samza and Kafka Streams replicate state changes by sending them to a dedicated Kafka topic with log compaction. VoltDB replicates state by redundantly processing each input message on several nodes.

The future of data systems

Data integration

Updating a derived data system based on an event log can often be made determinisitic and idempotent.

Distributed transactions decide on an ordering of writes by using locks for mutual exclusion, while CDC and event sourcing use a log for ordering. Distributed transactions use atomic commit to ensure exactly once semantics, while log-based systems are based on deterministic retry and idempotence.

Transaction systems provide linearizability, useful guarantees as reading your own writes. On the other hand, derived systems are often updated asynchronously, so they do not by default offer the same timing guarantees.

In the absence of widespread support for a good distributed transaction protocol, log-based derived data is the most promising approach for integrating different data systems.

However, as systems are scaled towards bigger and more coplex worloads, limitiations emerge: * Constructing a totally ordered log requires all events to pass through a *single leader node* that decides on the ordering. * An undefined ordering of events that originate on multiple datacenters. * When two events originate in different services, there is no defined order for those events. * Some applications maintain client-side state. Clients and servers are very likely to see events in different orders.

Deciding on a total order of events is known as *total order broadcast*, which is equivalent to consensus. It is still an open research problem to design consensus algorithms that can scale beyond the throughput of a single node.

Batch and stream processing

The fundamental difference between batch processors and batch processes is that the stream processors operate on unbounded datasets whereas batch processes inputs are of a known finite size.

Spark performs stream processing on top of batch processing. Apache Flink performs batch processing in top of stream processing.

Batch processing has a quite strong functional flavour. The output depends only on the input, there are no side-effects. Stream processing is similar but it allows managed, fault-tolerant state.

Derived data systems could be maintained synchronously. However, asynchrony is what makes systems based on event logs robust: it allows a fault in one part of the system to be contained locally.

Stream processing allows changes in the input to be reflected in derived views with low delay, whereas batch processing allows large amounts of accumulated historical data to be reprocessed in order to derive new views onto an existing dataset.

Derived views allow *gradual* evolution. If you want to restructure a dataset, you do not need to perform the migration as a sudden switch. Instead, you can maintain the old schema and the new schema side by side as two independent derived views onto the same underlying data, eventually you can drop the old view.

Lambda architecture

The whole idea behind lambda architecture is that incoming data should be recorded by appending immutable events to an always-growing dataset, similarly to event sourcing. From these events, read-optimised vuews are derived. Lambda architecture proposes running two different systems in parallel: a batch processing system such as Hadoop MapReduce, and a stream-processing system as Storm.

The stream processor produces an approximate update to the view: the batch processor produces a corrected version of the derived view.

The stream process can use fast approximation algorithms while the batch process uses slower exact algorithms.

Unbundling databases

Creating an index

Batch and stream processors are like elaborate implementations of triggers, stored procedures, and materialised view maintenance routines. The derived data systems they maintain are like different index types.

There are two avenues by which different storate and processing tools can nevertheless be composed into a cohesive system: * Federated databases: unifying reads. It is possible to provide a unified query interface to a wide variety of underlying storate engines and processing methods, this is known as *federated database* or *polystore*. An example is PostgreSQL's *foreign data wrapper*. * Unbundled databases: unifying writes. When we compose several storage systems, we need to ensure that all data changes end up in all the right places, even in the face of faults, it is like *unbundling* a database's index-maintenance features in a way that can synchronise writes across disparate technologies.

Keeping the writes to several storage systems in sync is the harder engineering problem.

Synchronising writes requires distributed transactions across heterogeneous storage systems which may be the wrong solution. An asynchronous event log with idempotent writes is a much more robust and practical approach.

The big advantage is *loose coupling* between various components: 1. Asynchronous event streams make the system as a whole more robust to outages or performance degradation of individual components. 2. Unbundling data systems allows different software components and services to be developed, improved and maintained independently from each other by different teams.

If there is a single technology that does everything you need, you're most likely best off simply using that product rather than trying to reimplement it yourself from lower-level components. The advantages of unbundling and composition only come into the picture when there is no single piece of software that satisfies all your requirements.

Separation of application code and state

It makes sense to have some parts of a system that specialise in durable data storage, and other parts that specialise in running application code. The two can interact while still remaining independent.

The trend has been to keep stateless application logic separate from state management (databases): not putting application logic in the database and not putting persistent state in the application.

Dataflow, interplay between state changes and application code

Instead of treating the database as a passive variable that is manipulated by the application, application code responds to state changes in one place by triggering state changes in another place.

Stream processors and services

A customer is purchasing an item that is priced in one currency but paid in another currency. In order to perform the currency conversion, you need to know the current exchange rate.

This could be implemented in two ways: * Microservices approach, the code that processes the purchase would probably wuery an exchange-rate service or a database in order to obtain the current rate for a particular currency. * Dataflow approach, the code that processes purchases would subscribe to a stream of exchange rate updates ahead of time, and record the current rate in a local database whenever it changes. When it comes to processing the purchase, it only needs to query the local database.

The dataflow is not only faster, but it is also more robust to the failure of another service.

Observing derived state

Materialised views and caching

A full-text search index is a good example: the write path updates the index, and the read path searches the index for keywords.

If you don't have an index, a search query would have to scan over all documents, which is very expensive. No index means less work on the write path (no index to update), but a lot more work on the read path.

Another option would be to precompute the search results for only a fixed set of the most common queries. The uncommon queries can still be served from the inxed. This is what we call a *cache* although it could also be called a materialised view.

Read are events too

It is also possible to represent read requests as streams of events, and send both the read events and write events through a stream processor; the processor responds to read events by emiting the result of the read to an output stream.

It would allow you to reconstruct what the user saw before they made a particular decision.

Enables better tracking of casual dependencies.

Aiming for correctness

If your application can tolerate occasionally corrupting or losing data in unpredictable ways, life is a lot simpler. If you need stronger assurances of correctness, the serializability and atomic commit are established approaches.

While traditional transaction approach is not going away, there are some ways of thinking about correctness in the context of dataflow architectures.

The end-to-end argument for databases

Bugs occur, and people make mistakes. Favour of immutable and append-only data, because it is easier to recover from such mistakes.

We've seen the idea of *exactly-once* (or *effectively-once*) semantics. If something goes wrong while processing a message, you can either give up or try again. If you try again, there is the risk that it actually succeeded the first time, the message ends up being processed twice.

Exactly-once means arranging the computation such that the final effect is the same as if no faults had occurred.

One of the most effective approaches is to make the operation *idempotent*, to ensure that it has the same effect, no matter whether it is executed once or multiple times. Idempotence requires some effort and care: you may need to maintain some additional metadata (operation IDs), and ensure fencing when failing over from one node to another.

Two-phase commit unfortunately is not sufficient to ensure that the transaction will only be executed once.

You need to consider *end-to-end* flow of the request.

You can generate a unique identifier for an operation (such as a UUID) and include it as a hidden form field in the client application, or calculate a hash of all the relevant form fields to derive the operation ID. If the web browser submits the POST request twice, the two requests will have the same operation ID. You can then pass that operation ID all the way through to the database and check that you only ever execute one operation with a given ID. You can then save those requests to be processed, uniquely identified by the operation ID.

Is not enough to prevent a user from submitting a duplicate request if the first one times out. Solving the problem requires an end-to-end solution: a transaction indentifier that is passed all the way from the end-user client to the database.

Low-level reliability mechanisms such as those in TCP, work quite well, and so the remaining higher-level faults occur fairly rarely.

Transactions have long been seen as a good abstraction, they are useful but not enough.

It is worth exploring F=fault-tolerance abstractions that make it easy to provide application-specific end-to-end correctness properties, but also maintain good performance and good operational characteristics.

Enforcing constraints

Uniqueness constraints require consensus

The most common way of achieving consensus is to make a single node the leadder, and put it in charge of making all decisions. If you need to tolerate the leader failing, you're back at the consensus problem again.

Uniqueness checking can be scaled out by partitioning based on the value that needs to be unique. For example, if you need usernames to be unique, you can partition by hash or username.

Asynchronous multi-master replication is ruled out as different masters concurrently may accept conflicting writes, so values are no longer unique. If you want to be able to immediately reject any writes that would violate the constraint, synchronous coordination is unavoidable.

Uniqueness in log-based messaging

A stream processor consumes all the messages in a log partition sequentially on a single thread. A stream processor can unambiguously and deterministically decide which one of several conflicting operations came first. 1. Every request for a username is encoded as a message. 2. A stream processor sequentially reads the requests in the log. For every request for a username tht is available, it records the name as taken and emits a success message to an output stream. For every request for a username that is already taken, it emits a rejection message to an output stream. 3. The client waits for a success or rejection message corresponding to its request.

The approach works not only for uniqueness constraints, but also for many other kinds of constraints.

Multi-partition request processing

There are potentially three partitions: the one containing the request ID, the one containing the payee account, and one containing the payer account.

The traditional approach to databases, executing this transaction would require an atomic commit across all three partitions.

Equivalent correctness can be achieved with partitioned logs, and without an atomic commit.

1. The request to transfer money from account A to account B is given a unique request ID by the client, and appended to a log partition based on the request ID.
2. A stream processor reads the log of requests. For each request message it emits two messages to output streams: a debit instruction to the payer account A (partitioned by A), and a credit instruction to the payee account B (partitioned by B). The original request ID is included in those emitted messages.
3. Further processors consume the streams of credit and debit instructions, deduplicate by request ID, and apply the chagnes to the account balances.

Timeliness and integrity

Consumers of a log are asynchronous by design, so a sender does not wait until its message has been proccessed by consumers. However, it is possible for a client to wait for a message to appear on an output stream.

Consistency conflates two different requirements: * Timeliness: users observe the system in an up-to-date state. * Integrity: Means absence of corruption. No data loss, no contradictory or false data. The derivation must be correct.

Violations of timeless are "eventual consistency" whereas violations of integrity are "perpetual inconsistency".

Correctness and dataflow systems

When processing event streams asynchronously, there is no guarantee of timeliness, unless you explicitly build consumers that wait for a message to arrive before returning. But integrity is in fact central to streaming systems.

Exactly-once or *effectively-once* semantics is a mechanism for preserving integrity. Fault-tolerant message delivery and duplicate supression are important for maintaining the integrity of a data system in the face of faults.

Stream processing systems can preserve integrity without requireing distributed transactions and an atomic commit protocol, which means they can potentially achieve comparable correctness with much better performance and operational robustness. Integrity can be achieved through a combination of mechanisms: * Representing the content of the write operation as a single message, this fits well with event-sourcing * Deriving all other state updates from that single message using deterministic derivation functions * Passing a client-generated request ID, enabling end-to-end duplicate supression and idempotence * Making messages immutable and allowing derived data to be reprocessed from time to time

In many businesses contexts, it is actually acceptable to temporarily violate a constraint and fix it up later apologising. The cost of the apology (money or reputation), it is often quite low.

Coordination-avoiding data-systems

1. Dataflow systems can maintain integrity guarantees on derived data without atomic commit, linearizability, or synchronous cross-partition coordination.
2. Although strict uniqueness constraints require timeliness and coordination, many applications are actually fine with loose constraints than may be temporarily violated and fixed up later.

Dataflow systems can provide the data management services for many applications without requiring coordination, while still giving strong integrity guarantees. *Coordination-avoiding* data systems can achieve better performance and fault tolerance than systems that need to perform synchronous coordination.

Trust, but verify

Checking the integrity of data is know as *auditing*.

If you want to be sure that your data is still there, you have to actually read it and check. It is important to try restoring from your backups from time to time. Don't just blindly trust that it is working.

Self-validating or *self-auditing* systems continually check their own integrity.

ACID databases has led us toward developing applications on the basis of blindly trusting technology, neglecting any sort of auditability in the process.

By contrast, event-based systems can provide better auditability (like with event sourcing).

Cryptographic auditing and integrity checking often relies on *Merkle trees*. Outside of the hype for cryptocurrencies, *certificate transparency* is a security technology that relies on Merkle trees to check the validity of TLS/SSL certificates.

Doing the right thing

Many datasets are about people: their behaviour, their interests, their identity. We must treat such data with humanity and respect. Users are humans too, and human dignitity is paramount.

There are guidelines to navigate these issues such as ACM's Software Engineering Code of Ethics and Professional Practice

It is not sufficient for software engineers to focus exclusively on the technology and ignore its consequences: the ethical responsibility is ours to bear also.

In countries that respect human rights, the criminal justice system presumes innocence until proven guilty; on the other hand, automated systems can systematically and artbitrarily exclude a person from participating in society without any proof of guilt, and with little chance of appeal.

If there is a systematic bias in the input to an algorithm, the system will most likely learn and amplify bias in its output.

It seems ridiculous to believe that an algorithm could somehow take biased data as input and produce fair and impartial output from it. Yet this believe often seems to be implied by proponents of data-driven decision making.

If we want the future to be better than the past, moral imagination is required, and that's something only humans can provide. Data and models should be our tools, not our masters.

If a human makes a mistake, they can be held accountable. Algorithms make mistakes too, but who is accountable if they go wrong?

A credit score summarises "How did you behave in the past?" whereas predictive analytics usually work on the basis of "Who is similar to you, and how did people like you behave in the past?" Drawing parallels to others' behaviour implies stereotyping people.

We will also need to figure outhow to prevent data being used to harm people, and realise its positive potential instead, this power could be used to focus aid an support to help people who most need it.

When services become good at predicting what content users want to se, they may end up showing people only opinions they already agree with, leading to echo chambers in which stereotypes, misinformation and polaristaion can breed.

Many consequences can be predicted by thinking about the entire system (not just the computerised parts), an approach known as *systems thinking*.

Privacy and tracking

When a system only stores data that a user has explicitly entered, because they want the system to store and process it in a certain way, the system is performing a service for the user: the user is the customer.

But when a user's activity is tracked and logged as a side effect of other things they are doing, the relationship is less clear. The service no longer just does what the users tells it to do, but it takes on interests of its own, which may conflict with the user's interest.

If the service is funded through advertising, the advertirsers are the actual customers, and the users' interests take second place.

The user is given a free service and is coaxed into engaging with it as much as possible. The tracking of the user serves the needs of the advertirses who are funding the service. This is basically *surveillance*.

As a thougt experiment, try replacing the word *data* with *surveillance*.

Even themost totalitarian and repressive regimes could only dream of putting a microphone in every room and forcing every person to constantly carry a device capable of tracking their location and movements. Yet we apparently voluntarily, even enthusiastically, throw ourselves into this world of total surveillance. The difference is just that the data is being collected by corporations rather than government agencies.

Perhaps you feel you have nothing to hide, you are totally in line with existing power structures, you are not a marginalised minority, and you needn't fear persecution. Not everyone is so fortunate.

Without understanding what happens to their data, users cannot give any meaningful consent. Often, data from one user also says things about other people who are not users of the service and who have not agreed to any terms.

For a user who does not consent to surveillance, the only real alternative is simply to not user the service. But this choice is not free either: if a service is so popular that it is "regarded by most people as essential for basic social participation", then it is not reasonable to expect people to opt out of this service. Especially when a service has network effects, there is a social cost to people choosing *not* to use it.

Declining to use a service due to its tracking of users is only an option for the small number of people who are priviledged enough to have the time and knowledge to understand its privacy policy, and who can affort to potentially miss out on social participation or professional opportunities that may have arisen if they ahd participated in the service. For people in a less priviledged position, there is no meaningful freedom of choice: surveillance becomes inescapable.

Having privacy does not mean keeping everything secret; it means having the freedom to choose which things to reveal to whom, what to make public, and what to keep secret.

Companies that acquire data essentially say "trust us to do the right thing with your data" which means that the right to decide what to reveal and what to keep secret is transferred from the individual to the company.

Even if the service promises not to sell the data to third parties, it usually grants itself unrestricted rights to process and analyse the data internally, often going much further than what is overtly visible to users.

If targeted advertising is what pays for a service, then behavioral data about people is the service's core asset.

When collecting data, we need to consider not just today's political environment, but also future governments too. There is no guarantee that every government elected in the future will respect human rights and civil liberties, so "it is poor civic hygiene to install technologies that could someday facilitate a police state".

To scrutinise others while avoiding scrutiny oneself is one of the most important forms of power.

In the industrial revolution tt took a long time before safeguards were established, such as environmental protection regulations, safety protocols for workplaces, outlawing child labor, and health inspections for food. Undoubtedly the cost of doing business increased when factories could no longer dump their waste into rivers, sell tainted foods, or exploit workers. But society as a whole benefited hugely, and few of us would want to return to a time before those regulations.

We should stop regarding users as metrics to be optimised, and remember that they are humans who deserve respect, dignity, and agency. We should self-regulate our data collection and processing practices in order to establish an maintain the trust of the people who depend on our software. And we should take it upon ourselves to educate end users about how their data is used, rather than keeping them in the dark.

We should allow each individual to maintain their privacy, their control over their own data, and not steal that control from them through surveillance.

We should not retain data forever, but purge it as soon as it is no longer needed.