Notes on *Designing Data-Intensive Applications* by Martin Kleppmann

Thomas Monson

Contents

1 Reliable, Scalable, and Maintainable Applications

1.1 Thinking About Data Systems

Redis is a datastore that is also used as a message queue. Apache Kafka is a message queue with database-like durability guarantees.

Data-intensive applications have requirements that are too demanding for a single tool to fulfill. Work is broken down into tasks that can be performed efficiently by a single tool, and those tools are stitched together with application code.

1.2 Reliability

Reliability: The system should continue to work correctly (performing the correct function at the desired level of performance) even in the face of adversity (hardware faults, software faults, human error).

A *fault* is something that can go wrong, such as a component deviating from its spec. A *failure* is when a system stops providing a required service to users. A system that anticipates faults and prevents them from causing failures is called *fault-tolerant* or *resilient*.

It is good practice to simulate faults in a system to test whether it will be able to handle faults that occur naturally (one tool for this is Netflix's *Chaos Monkey*).

1.2.1 Hardware Faults

Hard disks are reported as having a mean time to failure (MTTF) of about 10 to 50 years. Thus, on a storage cluster with 10,000 disks, we should expect on average one disk to die per day.

The traditional response to hardware faults is component redundancy. This makes total failure of a single machine rare. However, data-intensive applications are built on distributed systems, so it is more important that they

be able to tolerate the loss of entire machines.

A single-server systems requires planned downtime for upgrades. A system that can tolerate machine failure can be patched one node at a time, without downtime of the entire system (*rolling upgrade*).

Hardware faults are typically random and uncorrelated.

1.2.2 Software Faults

A systematic fault in software can cause cascading failures, where a fault in one component triggers a fault in another component, which in turn triggers further faults.

1.2.3 Human Errors

Use well-designed abstractions. Use sandbox environments during development. Do all kinds of testing. Roll out changes gradually and make it easy to roll them back. Set up detailed and clear monitoring, such as performance metrics and error rates.

1.2.4 How Important Is Reliability?

No notes.

1.3 Scalability

Scalability: As the system grows (in data volume, traffic volume, or complexity), there should be reasonable ways of dealing with that growth.

Discussing scalability means considering questions like:

- "If the system grows in a particular way, what are our options for coping with the growth?"
- "How can we add computing resources to handle the additional load?"

1.3.1 Describing Load

Load can be described with a few numbers called *load parameters*. The choice of parameter depends on the system. Some examples include: requests/sec, ratio of reads to writes, number of concurrent users, cache hit rate. The parameters may describe average-case or worst-case metrics.

Twitter users can post tweets or load their home timeline to read tweets. Twitter's scaling challenge is due to *fan-out*—each user follows and is followed by many people. There are two ways of implementing these operations:

1. On write, insert the new tweet into a global collection of tweets. On read, get all people the user follows, get all tweets for each of those users, and merge the tweets (sorted by time).

2. Maintain a cache for each user's timeline. On write, get all followers and insert the new tweet into their caches. On read, read the cache.

The first approach does more work on read. The second approach does more work on write. There are two orders of magnitude more reads than writes, which is why Twitter switched from the first to the second approach. But some users, like celebrities, have tens of millions of followers, which would require tens of millions of fan-out writes per tweet post. For these users, Twitter uses the first approach.

In this case, distribution of followers per user is a key load parameter, since it determines the fan-out load.

1.3.2 Describing Performance

Ways to investigate performance under increased load:

- When you increase a load parameter and keep the system resources unchanged, how is the performance of your system affected?

- When you increase a load parameter, how much do you need to increase the resources if you want to keep performance unchanged?

Performance numbers:

- *Throughput*: the number of units of information that can be processed in a given time (e.g. 1000 records/second)
- *Response time*: the time between a client sending a request and receiving a response
- *Service time*: the time to process a request
- *Latency*: the time a request is waiting to be processed (during which it is *latent*)

(Latency and response time are often used synonymously, but they are not the same.)

Response time varies, even for a request made multiple times. Thus, you should think of response time as a distribution of values that you can measure.

The median response time (*p50*) is a good measure of how long users typically have to wait. High percentiles of response times (*tail latencies*) represent the requests that take the longest to serve. For example, the 99th and 99.9th percentiles (*p99* and *p999*) are slower than 99 out of 100 requests and 999 out of 1000 requests respectively.

Amazon describes response time requirements in terms of p999 because the slowest requests often come from the most valuable customers (they have more data on their account because they buy more). Percentiles are also used in contracts to define whether or not a service is considered *up*.

As a server can only process a small number of things in parallel, it only takes a small number of slow requests to hold up the processing of subsequent requests (*head-of-line blocking*).

If a single end-user request calls backend services multiple times, the chance of getting a slow call increases. Even if the calls are made in parallel, the end-user request still needs to wait for the slowest call to complete. This effect, known as *tail latency amplification*, causes a higher proportion of end-user requests to be slow.

1.3.3 Approaches for Coping with Load

Some systems are *elastic*, meaning that they can automatically add computing resources when they detect a load increase, whereas other systems are scaled manually. An elastic system is useful if load is unpredictable, but manually scaled systems are simpler.

Distributing stateless services across multiple nodes is easy; distributing stateful data systems across multiple nodes introduces a lot of complexity.

The architecture of large-scale systems is usually highly specific to the application. There is no generic scaling solution. Potential challenges: volume of reads, volume of writes, volume of data to store, complexity of the data, response time requirements, access patterns.

A system designed to handle 100k 1kB requests/sec and a system designed to handle 3 2GB requests/sec look very different, even though they have the same throughput.

An architecture that scales well for a particular application is built around assumptions of which operations will be common and which will be rare—the load parameters. If these assumptions are wrong, the application will not scale.

1.4 Maintainability

Maintainability: Over time, many different people will work on the system (engineering and operations, both maintaining current behavior and adapting the system to new use cases), and they should all be able to work on it productively.

1.4.1 Operability: Making Life Easy for Operations

Operability: Make it easy for operations teams to keep the system running smoothly.

Provide visibility into runtime behavior with good monitoring. Support automation and integration with standard tools. Avoid dependency on single

machines. Provide good documentation and operational model ("if I do X, Y will happen"). Provide good default behavior and freedom to override defaults.

1.4.2 Simplicity: Managing Complexity

Simplicity: Make it easy for new engineers to understand the system by removing as much complexity as possible from the system.

Making a system simpler does not necessarily mean reducing its functionality; it can also mean reducing *accidental* complexity. Complexity is accidental if it is not inherent in the problem that the software solves but arises only from the implementation.

Abstraction is a great tool for removing accidental complexity. Reuse of abstractions is efficient and can improve performance.

1.4.3 Evolvability: Making Change Easy

Evolvability: Make it easy for engineers to make changes to the system in the future, adapting it for unanticipated use cases as requirements change. Also known as *extensibility*, *modifiability*, or *plasticity*.

"Agility at the data system level"

1.5 Summary

No notes.

2 Data Models and Query Languages

Data models are layered on top of each other. For example:

1. Entities in the real world are modeled in terms of data structures, and those data structures are manipulated in application code.
2. The data structures are stored in a database as documents, tables, graph models, etc.

3. The documents/tables/graphs are encoded in storage media in such a way that the database can efficiently query and manipulate it.
4. The bytes of the encoded data are represented in terms of electrical current, pulses of light, magnetic fields, etc.

Every data model embodies assumptions about how it is going to be used. The choice of data model has a profound effect on what the application above it can and cannot do.

2.1 Relational Model Versus Document Model

In the *relational model* (SQL), data is organized into *relations* (tables), where each relation is an unordered collection of *tuples* (rows). Relational databases were originally used for business data processing, such as transaction processing and batch processing.

2.1.1 The Birth of NoSQL

Interest in NoSQL databases was motivated by:

- Need for greater scalability
- Preference for FOSS
- Need for specialized query operations
- Desire for a less restrictive, dynamic data model

The idea that relational databases will continue to be used alongside nonrelational databases is known as *polyglot persistence*.

2.1.2 The Object-Relational Mismatch

If the application code uses objects and the data is stored in relational tables, there must be an awkward translation layer between the objects and the tables (*impedance mismatch*). Object-relational mappers (ORMs) reduce the complexity of this translation layer.

One user can have many job positions, many periods of education, and many pieces of contact information. These *one-to-many relationships* can be expressed in various ways:

- Put positions, education, and contact information in separate tables, with a foreign key reference to the users table
- Represent positions, education, and contact information as separate JSON/XML documents and store them in "structured datatype" columns (database can query and index)
- Represent positions, education, and contact information as separate JSON/XML documents and store them in text columns, letting the application interpret the structure and content (database cannot query or index)

It is easier to fetch a whole user profile if the profile is stored as a document than if it is stored as a collection of tables (the document has better *locality*).

One-to-many relationships form a tree structure.

2.1.3 Many-to-One and Many-to-Many Relationships

Whether you store data as an ID or as text is a question of duplication. Do you want to store the string "Chicago" one time and have all Chicagoans point to that string when asked where they are from or do you want to let every Chicagoan store their own "Chicago" as an answer to that question? If data is duplicated and needs to be changed in the future, all redundant copies need to be updated.

Removing such duplication is the key idea behind *normalization*. But this requires *many-to-one relationships* (many people live in one city). One-to-many relationships are natural for documents, but many-to-one/many relationships are not.

Many-to-one/many relationships are easy in SQL (with joins) and hard in NoSQL.

Most NoSQL databases do not support joins, so, to support many-to-one/many relationships, joins will have to be simulated in application code by making multiple queries.

2.1.4 Are Document Databases Repeating History?

IBM's Information Management System (IMS) used the *hierarchical model* (similar to JSON, tree structure). It had the same problems with representing many-to-one/many relationships.

The *network model* (*CODASYL model*) tried to support many-to-one/many relationships by allowing child records to have multiple parents. Thus, a record could have several different *access paths*. This is bad because the application code would have to keep track of all of the various paths one could traverse to get to a record. Querying was difficult if you didn't have the access path.

By contrast, the *relational model* does not use nested structures. It has a *query optimizer* that decides what parts of the query to execute in which order and which indexes to use. This is analogous to the access path, but it is abstracted from the application developer.

Like the hierarchical model, document databases represent one-to-many relationships by nesting records instead of by relating tables. Relational and document databases represent many-to-one/many relationships in similar ways: the *one* item is referenced by *foreign key* in relational databases or by *document reference* in document databases, and these references are resolved at read time by means of a join or follow-up queries.

2.1.5 Relational Versus Document Databases Today

Which data model leads to simpler application code? It depends on the structure of your data. If your data is tree-like (one-to-many relationships) and loaded all at once, the document model is probably a good choice.

Shredding (splitting a document into multiple tables) can lead to cumbersome schemas and complicated application code.

If your data has many-to-many relationships (joins are required), the document model is less appealing. You can reduce the need for joins by denormalizing data (consistency issues) or emulate joins in application code by making multiple requests (additional code complexity, slow).

Document databases are *schema-on-read* (the structure of the data is implicit and only interpreted when the data is read). Relational databases are *schema-on-write* (the structure of the data is explicit and the database ensures all written data conforms to it). This is analogous to dynamic (run-time) vs. static (compile-time) type checking.

If you want to add new fields to your data, schema-on-read allows you to just start writing documents with new fields. Application code handles the case where old documents are read. For example:

```
if (user && user.name && !user.first_name) {
    // Old documents don't have first_name
    user.first_name = user.name.split(" ")[0];
}
```

On the other hand, schema-on-write requires you to add a new column to a table and rewrite every row to add the updated values (or you can leave the values as `NULL` and update on read, similar to a document database). This is slow and may result in downtime.

Schema-on-read is good for data that is heterogenous. This may be the case if you are working with many different kinds of objects or if the structure of your data is determined by external systems over which you have no control. Schema-on-write is good for homogeneous data.

A document is usually stored as a single continuous string. If your application often needs to access the entire document, there is a performance advantage to this storage locality.

Most relational databases have support for XML or JSON documents, and many document databases have support for relational-like joins.

2.2 Query Languages for Data

An *imperative* language tells the computer to perform certain operations in a certain order. A *declarative* query language specifies the pattern of the data you want, but not how to achieve that goal (the query optimizer handles

that). It allows the database system to introduce performance improvements without requiring any changes to queries. Declarative query languages are also more easily parallelizable.

2.2.1 Declarative Queries on the Web

Say you want to change the background color of a navigation item on a website when the user navigates to that page. This can be done imperatively (JavaScript) or declaratively (CSS). If done declaratively, the browser will automatically detect when the navigation item is no longer "selected," and it will remove the background color accordingly. Declarative code would also be able to receive performance improvements without changing the code.

2.2.2 MapReduce Querying

MapReduce is a programming model for processing large amounts of data in bulk across many machines (*distributed execution*). It is supported by some NoSQL databases. It is somewhere in between imperative and declarative.

In MapReduce, you define the logic of two pure functions, `map` and `reduce`. Then the processing framework repeatedly calls those functions.

The MongoDB MapReduce feature takes four parameters: a `map()` function, a `reduce(key, values)` function, a `query` condition (specified declaratively), and `out` (an output collecton). `map` is called once for every document that matches `query` (within `map`, the document is referred to with the `this` keyword). `map` emits a key-value pair. The key-value pairs emitted by `map` are grouped by key. For all key-value pairs with the same key, `reduce` is called once. Each `reduce` call writes a return value to `out`.
A distributed implementation of SQL can be written as a pipeline of MapReduce queries or without MapReduce at all.

MongoDB also has a declarative query language called the *aggregation pipeline* (more opportunities for query optimizer to improve performance). It is like SQL with a JSON-based syntax—a SQL-like language inside of a NoSQL system!

2.3 Graph-Like Data Models

Graph data models are a natural choice when many-to-many relationships are very common in your data. Graphs can store homogeneous or heterogenous data (verticies may or may not represent different kinds of objects, edges may or may not represent different kinds of relationships).

2.3.1 Property Graphs

In the property graph model, each vertex consists of:

- A unique identifier
- A set of outgoing edges
- A set of incoming edges
- A collection of properties (key-value pairs)

Each edge consists of:

- A unique identifier
- The vertex at which the edge starts (the *tail vertex*)
- The vertex at which the edge ends (the *head vertex*)
- A label to describe the kind of relationship between the two verticies
- A collection of properties (key-value pairs)

Any vertex can be connected with any other vertex. You can traverse the graph forward and backward by accessing the outgoing and incoming edges of any vertex. You can store heterogenous data.

One way of representing a property graph is with relational tables (one for verticies, one for edges).

Graphs are good for evolvability.

2.3.2 The Cypher Query Language

Cypher is a declarative query language for property graphs, created for the Neo4j graph database.

The following Cypher query adds some verticies and edges to the database:

```
CREATE
  (NAmerica:Location {name:'North America', type:'continent'}),
  (USA:Location      {name:'United States', type:'country'}),
  (Idaho:Location    {name:'Idaho',         type:'state'}),
  (Lucy:Person       {name:'Lucy'}),
  (Idaho) -[:WITHIN]->  (USA) -[:WITHIN]-> (NAmerica),
  (Lucy)  -[:BORN_IN]-> (Idaho)
```

The following Cypher query finds people who emigrated from the US to Europe:

```
MATCH
  (person) -[:BORN_IN]->  () -[:WITHIN*0..]-> (us:Location {name:'United States
  (person) -[:LIVES_IN]-> () -[:WITHIN*0..]-> (eu:Location {name:'Europe'})
RETURN person.name
```

This query could be executed by starting with people and working toward the locations or it could be executed by starting with the locations and working toward people. This is up to the query optimizer.

2.3.3 Graph Queries in SQL

In a relational database, you usually know in advance which joins you need in your query. In a graph query, you may need to traverse a variable number of edges before you find the target vertex (the number of joins is not fixed in advance). The `() -[:WITHIN*0]-> ()` Cypher rule expresses a variable-length traversal path.

SQL can express such a path using *recursive common table expressions*, but the query will be long and complicated. A graph query language is more appropriate for this use case.

2.3.4 Triple-Stores and SPARQL

In the triple-store model, all information is stored in the form of a three-part statement: (*subject*, *predicate*, *object*). The subject is a vertex. The object is one of two things:

1. A primitive value. In this case, the predicate and object form a key-value property of the subject vertex. For example, (*lucy*, *age*, *33*) represents a vertex `lucy` with properties {`"age": 33`}.

2. Another vertex. In this case, the predicate is an edge, the subject is the tail vertex, and the object is the head vertex. For example, in (*lucy*, *marriedTo*, *alain*), the vertex `lucy` is connected to the vertex `alain` by an edge with label `marriedTo`.

Here's an example of graph data written as triples in a format called *Turtle*:

```
@prefix : <urn:example:>.
_:lucy     a       :Person.
_:lucy     :name   "Lucy".
_:lucy     :bornIn _:idaho.
_:idaho    a       :Location.
_:idaho    :name   "Idaho".
_:idaho    :type   "state".
_:idaho    :within _:usa.
_:usa      a       :Location.
_:usa      :name   "United States".
_:usa      :type   "country".
_:usa      :within _:namerica.
_:namerica a       :Location.
_:namerica :name   "North America".
_:namerica :type   "continent".
```

The *Resource Description Framework* (RDF) is a standard for describing metadata. It makes statements about resources in the form of triples. It was originally designed to support the *semantic web*, the idea of publishing machine-readable information on websites alongside the human-readable information that already exists there. The RDF data model does not have to be used for projects related to the semantic web; it is also just a good model

for graph data.

In RDF, the subject, predicate, and object of a triple are often URIs. This is done to avoid naming conflicts, like the word `lives_in` having a different meaning on your machine than it has on another person's machine.

SPARQL is a query language for triple-stores using the RDF data model. The following SPARQL query finds people who emigrated from the US to Europe:

```
PREFIX : <urn:example:>

SELECT ?personName WHERE {
  ?person :name ?personName
  ?person :bornIn  / :within* / :name "United States".
  ?person :livesIn / :within* / :name "Europe".
}
```

2.3.5 The Foundation: Datalog

Datalog (a subset of Prolog) is a declarative logic programming language that is often used as a query language. Its data model is similar to the triple-store model, but its triples (known as *facts*) are written in the form *predicate*(*subject*, *object*).

Here's an example of graph data represented as Datalog facts:

```
name(namerica, 'North America').
type(namerica, continent).

name(usa, 'United States').
type(usa, country).
within(usa, namerica).

name(idaho, 'Idaho').
type(idaho, state).
within(idaho, usa).
```

```
name(lucy, 'Lucy').
born_in(lucy, idaho).
```

The following Datalog query finds people who emigrated from the US to Europe:

```
within_recursive(Location, Name) :- name(Location, Name).     /* Rule 1 */

within_recursive(Location, Name) :- within(Location, Via),    /* Rule 2 */
                                    within_recursive(Via, Name).

migrated(Name, BornIn, LivingIn) :- name(Person, Name),       /* Rule 3 */
                                    born_in(Person, BornLoc),
                                    within_recursive(BornLoc, BornIn),
                                    lives_in(Person, LivingLoc),
                                    within_recursive(LivingLoc, LivingIn).

?- migrated(Who, 'United States', 'Europe').
/* Who = 'Lucy'. */
```

Datalog defines *rules* that tell the database about new predicates (e.g. `within_recursive` and `migrated`). They are not stored in the database but are derived from data or other rules.

Words that start with an uppercase letter are variables, and predicates match with existing triples. For example, `name(Location, Name)` matches `name(namerica, 'North America')` with variable bindings `Location = namerica` and `Name = 'North America'`.

A rule applies if there is a match for *all* predicates on the righthand side of the `:-` operator. If the rule applies, it is as if the lefthand side of the `:-` is added to the database.

Here's an example. Repeated application of Rules 1 and 2 will tell you all the locations contained within North America:

1. `name(namerica, 'North America')` $\Longrightarrow$ `within_recursive(namerica, 'North America')`

2. `within(usa, namerica)` $\wedge$ `within_recursive(namerica, 'North America')` $\implies$ `within_recursive(usa, 'North America')`

3. `within(idaho, usa)` $\wedge$ `within_recursive(usa, 'North America')` $\implies$ `within_recursive(idaho, 'North America')`

Rule 3 can then be used to find people `Who` who were born in some location `BornIn` and live in some location `LivingIn`.

2.4 Summary

Some specialized data models:

- Genome databases optimized for *sequence-similarity search*
- Systems for analysis of petabyte-scale data (e.g. ROOT for particle physics)
- Levenshtein automata used to power *full-text search* (e.g. Apache Lucene)

3 Storage and Retrieval

3.1 Data Structures That Power Your Database

A simple key-value store can be implemented with these Bash functions:

```
#!/bin/bash

db_set () {
    echo "$1,$2" >> database
}

db_get () {
    grep "^$1," database | sed -e "s/^$1,//" | tail -n 1
}
```

Data would be stored in an append-only file (a *log*), and outdated key-value pairs would remain in the file. `db_set` has good performance because appending is efficient. `db_get` has bad performance because it performs a linear search for every lookup. It could be improved with an *index*.

An *index* is an additional structure that holds metadata derived from the primary data in the database. It helps you locate data but it needs to be updated when the database changes. **Well-chosen indexes speed up read queries, but every index slows down writes.**

3.1.1 Hash Indexes

A *hash index* is an index implemented with a hash map. You could maintain an in-memory hash map that maps keys to byte offsets. When searching for the value associated with a key, the hash map will tell you how far into the file (on disk) the key-value pair is located. All of the keys must fit within the available memory. An append-only storage engine with a hash index would be appropriate for a system with many writes per key.

Only appending means that the database will be full of outdated entries. How do we fix this inefficient use of disk space?

- Split the log into *segments* of a certain size; when the current segment is full, write to a new segment file.
- Perform *compaction* on the segments (delete outdated key-value pairs).
- Merge smaller compacted segments together.

Segments cannot be modified after being written; compaction/merging must create a new file. The compaction/merging process can be done in a background thread. Meanwhile, reads and writes can be served with the old segment files.

Each segment will have its own in-memory hash map. To find the value associated with a key, search the segments for the key in order from most recent to least recent.

Other concerns:

- Logs should not be in CSV format, they should be binary, storing the byte-length of a string, followed by the string
- To delete a key, you must append a special deletion record (a *tombstone*) to the log. Values for the deleted key that come before this tombstone will be deleted during compaction.
- If the server crashes, the in-memory hash maps will be lost. Consider storing snapshots of the maps on disk.
- The database may crash while appending a record. Consider including checksums in the log to determine whether a record is corrupted and should be ignored.
- Concurrency: one write thread (writes are strictly sequential), multiple read threads (segments are append-only and immutable)

Append-only writing has advantages over overwriting. Appending is a sequential write operation, which is much faster than a random write operation, especially on HDDs. Because values cannot be overwritten, there is no chance that a file will be half-overwritten after a crash.

Hash indexes do not facilitate efficient *range queries* (reading all records with values within some range).

3.1.2 SSTables and LSM-Trees

What if the key-value pairs in the segments were sorted by key? This is called *SSTable* or *Sorted String Table* format. SSTable segments have many advantages over log-structured segments with hash indexes:

1. Merging is efficient. Look at the first key in each segment, copy the lowest key to the output file, repeat. The merged segment is sorted by key. (A given key can only appear once in each segment, but multiple segments can contain the same key. If the same keys are being compared, copy the value from the newer segment and discard the value from the older segment.)
2. You don't need to keep an index for every key. You can just keep indexes for some keys (maybe one every few kilobytes) and scan between

two adjacent indexes when searching for a key that would fall between them in the sorted order.

3. Key-value pairs that fall between indexes can be grouped into a block and compressed before being written to disk.

How can the segments be sorted when writes are append-only and can come in any order? By adding new records to a self-balancing binary search tree (such as a red-black tree or AVL tree) in memory (insert in any order, read in sorted order). This in-memory tree is called the *memtable*.

When the memtable gets bigger than some threshold, write it to disk as an SSTable file (reading the memtable in sorted order is O(n)). While the memtable is being written to disk, new writes can be sent to a new memtable.

To serve a read request, search the memtable first, then the most recent on-disk segment, then the segment before that, ...

If the server crashes, the memtable will be lost. To handle this, keep a separate log on disk to which every write is appended. It will contain the same records as the memtable but not in sorted order. It can be sorted later, if a crash happens. Once the memtable is written to disk, this log can be deleted.

This indexing structure is known as a *log-structured merge tree* (LSM-tree). Storage engines based on the idea of compacting and merging sorted files are called LSM storage engines.

If a key is not in the database, searching for it will take a long time. Storage engines optimize this operation by using *Bloom filters*, data structures for approximating the contents of a set.

There are different ways to determine the order and timing of how SSTables are compacted and merged. 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 proceed more incrementally and use less disk space.

LSM-trees can efficiently perform range queries, and they support high write throughput because they write data sequentially.

3.1.3 B-Trees

Like SSTables, B-trees keep key-value pairs sorted by key (efficient lookup and range queries). They have been around since 1970, and they are very popular.

B-trees break the database down into fixed-size *blocks* or *pages*, traditionally 4 KB in size (sometimes bigger), and read or write one page at a time.

A page can be identified with a reference (address on disk). A page contains (alternating) keys and references to other pages.

For example, when looking for the value associated with key `251` in the following page:

100	ref1	200	ref2	300

You would follow `ref2` to the page that contains all keys greater than or equal to `200` and less than or equal to `300`.

Eventually, you will get to a *leaf page*, which either contains (alternating) keys and their values or (alternating) keys and references to the pages where the values can be found.

The number of page references in a page is known as the *branching factor*. It depends on the space required to store references and range boundaries but is typically several hundred.

- *To update a value for a key*: search for leaf page containing key, change value in page, write page back to disk.
- *To add a new key*: search for page whose range encompasses key, add key. If not enough space in page, split page into two half-full pages, update parent page with references to two new pages.

The tree will remain balanced. Most databases can fit into a B-tree with three or four levels (a four-level tree of 4 KB pages with branching factor of 500 can store up to 256 TB).

Unlike the LSM-tree, the B-tree actually modifies files in-place. When a page is split, it requires three pages to be overwritten. If the database crashes during this operation, an orphan page may be created. To prevent this, B-tree implementations often have an additional data structure on disk called a *write-ahead log* (WAL) or *redo log*, where B-tree modifications are appended before they are applied to the pages of the tree itself.

Updating pages in-place $\implies$ concurrency control measures are required (e.g. *latches*).

B-Tree Optimizations:

- Instead of overwriting pages and maintaining a WAL, you could write a modified page to a new location and overwrite the parent page to point to the modified page.
- Abbreviate keys (e.g. `1024` and `1048` could be stored as `24` and `48`). More keys in a page $\implies$ higher branching factor $\implies$ fewer levels.
- Many B-tree implementations try to keep the leaf pages in sequential order to support fast range queries. This becomes difficult as the tree grows. LSM-trees are better at this.
- Add additional pointers, like giving leaf pages left and right pointers to sibling leaf pages.

3.1.4 Comparing B-Trees and LSM-Trees

As a rule of thumb, LSM-trees are faster for writes and B-trees are faster for reads. However, in practice, benchmarks are inconclusive, and you should measure the performance of both under your actual workload.

A B-tree index must write each piece of data at least twice, to the WAL and the page. And the entire page has to be written, even if only a few bytes changed. A log-structured index also rewrites data multiple times, during

compaction and merging.

One write resulting in multiple writes to disk is called *write amplification*. In write-heavy applications, this can be a bottleneck on write throughput. LSM-trees can typically sustain higher write throughput than B-trees (lower write amplification, sequentially writing compact SSTables vs/. overwriting several pages).

LSM-trees can be compressed better. B-trees leave some disk space unused due to fragmentation.

The compaction process of log-structured storage can sometimes interfere with incoming reads and writes (they have to wait for it to finish).

A log-structured storage engine must share write throughput between initial writes and compaction writes. As the database grows, more of the disk bandwidth is needed for compaction. If compaction cannot keep up with writes (high write throughput, poor configuration), the number of unmerged segments will grow until disk space runs out. Reads will also slow down because segments are not compacted.

In a B-tree, each key exists in exactly one place in the index. Transactional isolation is often implemented using locks on ranges of keys, and these locks can be directly attached to a B-tree.

3.1.5 Other Indexing Structures

A *primary index* is an index where each key uniquely identifies a row in a relational table, a document in a document store, or a vertex in a graph. A *secondary index* is an index where keys are not necessarily unique. In a relational database, you can create secondary indexes with the `CREATE INDEX` command, and they are often crucial for performing joins efficiently.

LSM-trees and B-trees are used to implement key-value indexes, but secondary indexes output multiple values. We can implement a secondary index with a key-value index either by:

1. making each value a list of row identifiers, or

2. making each key unique by appending a row identifier to it.

Values in an index are usually references to primary data (rows). In this case, the rows themselves are stored in an unordered *heap file*, and the values refer to some location in the heap file. This avoids duplicating data when multiple secondary indexes are present.

A row in a heap file can be overwritten in-place if the new value is not larger than the old value. If it is, the row needs to written somewhere else in the heap, so either all indexes need to be updated or a forwarding pointer must be left behind at the old location.

If the hop from index to heap file impacts reads too much, you can store the rows directly in the index as its values (as originals or copies). This is known as a *clustered index*. Or if you want to store only some of the table's columns in the index, you can do so with a *covering index* or *index with included columns* (some queries can be answered with the index alone, the index *covers* these queries).

Multi-column indexes

What if you need to query multiple columns of a table simultaneously (i.e. input two keys)? You could do this with an LSM-tree or B-tree by simply concatinating the values of multiple columns to form keys. This is the most common type of multi-column index—a *concatenated index*. However, in such an index with keys of the form `(lastname, firstname)`, queries by *lastname* will be efficient, but queries by *firstname* will not.

For efficient multi-column range queries, *multi-dimensional indexes* are a better choice. They can do queries like this efficiently:

```
SELECT * FROM restaurants WHERE latitude  > 51.4946 AND latitude  < 51.5079
                            AND longitude > -0.1162 AND longitude < -0.1004;
```

Such indexes are usually implemented with *R-trees*. You could also use a *UB-tree*, which is a data structure that uses a space-filling curve to transform multivariate keys into univariate keys and then indexes those univariate keys in a B-tree. Multi-dimensional indexes are not necessarily spatial. They could be used to search for products with a color that falls within a

constrained space in the RGB color domain.

Full-text search and fuzzy indexes

What if you need to query based on a given key and on keys that are similar to the given key? This is called *fuzzy querying*, and it is often used in full-text search. Given an initial word, it can be used to find synonyms, grammatical variations, words nearby in the text, or words of a certain *edit distance* away.

Keeping everything in memory

Indexes are answers to the limitations of disks (and SSDs). Physical storage location on these media affects their read/write performance. What if we instead store everything in RAM, where physical storage location does not affect read/write performance?

HDDs and SSDs are non-volatile and cheaper than RAM. However, RAM is getting cheaper and its data can be made more durable by means of battery power, logging changes on disk, writting snapshots to disk, or replication of state to other machines.

In-memory databases are faster because they do not need to encode in-memory data structures in a form that can be stored on disk.

An in-memory database can support datasets that are larger than the available memory. It can do this with *anti-caching*, which is the process of evicting the least recently used data from memory to disk when memory is full and loading it back into memory when it is accessed again. However, any in-memory indexes must fit entirely within memory.

Implementations: VoltDB, MemSQL, Oracle TimesTen, RAMCloud, Redis, Couchbase

3.2 Transaction Processing or Analytics?

In the early days of businesses using databases, a write typically corresponded to a commercial transaction taking place. Later, writes were used to represent many different things—postings comments, performing actions in a game, etc. The term *transaction* came to be used to describe any group of

reads and writes that forms a logical unit.

Databases are typically used for either *online transaction processing* (OLTP) or *online analytics processing* (OLAP), where *online* roughly means *interactive*. OLTP is used for serving user requests, OLAP is used for supporting business intelligence.

OLTP typically reads a small number of records per query, by giving a key to an index. It supports low-latency reads and writes. Its data represents the current state of a system.

OLAP typically scans over a large number of records per query, reads only a few columns per record, and calculates aggregate statistics (e.g. count, sum, average). It writes data in bulk or in a continuous stream of updates. It often makes use of *batch processing* (latency can be high). Its data represents a history of events.

3.2.1 Data Warehousing

OLTP systems need to be highly available and low-latency. Running expensive analytics queries on them can harm the performance of concurrently executing transactions. Thus, there was a trend in the 1990s for companies to copy data from their OLTP databases over to a *data warehouse*, where analysts could perform their queries without impacting transactional performance. To do this, data is *extracted* (periodically or continuously), *transformed* into an analysis-friendly format, and *loaded* into the warehouse via a process called *ETL*.

Data warehousing is more important for large companies. But a big advantage to using a separate data warehouse for analytics is that you can optimize its storage engine for analytic access patterns.

Implementations:

- *Commercial data warehouses*: Teradata, Vertica, Microsoft SQL Server, SAP HANA, Amazon RedShift (ParAccel)
- *Open-source SQL-on-Hadoop*: Apache Hive, Spark SQL, Impala, Tajo, Drill; Facebook Presto

3.2.2 Stars and Snowflakes: Schemas for Analytics

There is less data model diversity for analytics than there is for transaction processing. Many data warehouses use a *star schema* (*dimensional modeling*).

In a star schema, there is a single, central *fact table* that is surrounded by *dimension tables*. Each row of the fact table represents an event (such as a customer's purchase of a product). Columns in the fact table can either be *attributes* (values, such as the price at which the product was sold) or *dimensions* (foreign key references to dimension tables, such as a reference to a record in a product table).

One variation of this template is the *snowflake schema*, where dimensions can be further broken down into subdimensions. For example, a fact table could have a product dimension, and that product dimension table could have a brand dimension or a category dimension.

3.3 Column-Oriented Storage

Fact tables can sometimes have trillions of rows. They often have over 100 columns, but typical analytics queries access only a few columns at a time.

In most OLTP databases, storage is *row-oriented*: all the values from one row of a table are stored next to each other. Document databases are similar: a document is stored as a contiguous sequence of bytes.

It is inefficient to do an analytics query on row-oriented storage. You would have to scan through over 100 attributes per row, the majority of which would not be used. Once the rows are in memory, you would have to parse them and filter out the ones that don't match the query criteria.

It is more efficient to do an analytics query on *column-oriented* storage. Store the values from each column together on disk. Only read the columns that are used in the query.

Each column file contains its data in the order of the rows (e.g. to get the 23rd row, read the 23rd value from each column file).

Column-oriented storage can also be used for non-relational data. For example, Parquet is a columnar storage format that supports documents.

3.3.1 Column Compression

Repetitive sequences of column values can be compressed to lower demands on disk throughput. One way to do this is to encode column values as bitmaps to create a *bitmap index* and then compress the bitmaps using *run-length encoding*. One bitmap is created for each distinct value.

Bitmap indexes work well for low-cardinality columns, which have a small number of distinct values relative to the total number of records. As cardinality increases, the number of bitmaps increases, the bitmaps become sparse, and run-length encoding becomes more effective.

Column values:

V	69	69	74	31	31	31	29	30	30	31	31	31	68	69	69

Bitmap for each value:

29	0	0	0	0	0	0	**1**	0	0	0	0	0	0	0	0
30	0	0	0	0	0	0	0	**1**	**1**	0	0	0	0	0	0
31	0	0	0	**1**	**1**	**1**	0	0	0	**1**	**1**	**1**	0	0	0
68	0	0	0	0	0	0	0	0	0	0	0	0	**1**	0	0
69	**1**	**1**	0	0	0	0	0	0	0	0	0	0	0	**1**	**1**
74	0	0	**1**	0	0	0	0	0	0	0	0	0	0	0	0

Run-length encoding:

29:	6, 1	(6 zeros, 1 one, rest zeros)
30:	7, 2	(7 zeros, 2 ones, rest zeros)
31:	3, 3, 3, 3	(3 zeros, 3 ones, 3 zeros, 3 ones, rest zeros)
68:	12, 1	(12 zeros, 1 one, rest zeros)
69:	13, 2	(13 zeros, 2 ones)
74:	2, 1	(2 zeros, 1 one, rest zeros)

Bitmap indexes can perform bitwise logical operations within and across columns, which is very helpful for analytics workloads.

- `WHERE product_sk IN (30, 68, 69):` (bitwise *OR*)
- `WHERE product_sk = 31 and store_sk = 3:` (bitwise *AND*)

For data warehouse queries, both disk-to-memory bandwidth and memory-to-CPU bandwidth are bottlenecks. Compressed column-oriented storage layouts help with both. Because the data is compressed, more rows from a column can fit into the CPU's L1 cache. Because rows do not need to be parsed for relevant data, the CPU can iterate over them in a tight loop. Bitwise logical operators can operate on the compressed data directly via *vectorized processing*.

3.3.2 Sort Order in Column Storage

Column data can be stored simply by order of insertion, or we can impose some other order upon it (as we did with SSTables for row data).

During sorting, elements in all of the column files must be rearranged in exactly the same way, to preserve row structure. Thus, rows can be sorted according to one column's sorted order. For example, transactions could be sorted by date. This would improve queries over date ranges.

A second column can determine the sort order of any rows that have the same value in the first column. For example, transactions on the same date can be sorted by product name. This would improve queries that group or filter transactions by product name within a certain date range. You can define a third sort column, a fourth sort column, and so on...

If the primary sort column has low cardinality, then it will have long sequences of repeated values that can be compressed with run-length encoding. Compression works for second and third sort columns and so on, but its effectiveness decreases as you move down the sort priority.

Different queries benefit from different sort orders, so you could store the same data sorted in different ways to improve different kinds of queries. And because you need to replicate your data across multiple machines for redundancy anyway, you might as well sort the replicas in different ways.

3.3.3 Writing to Column-Oriented Storage

Update-in-place writing (what B-trees do) is not viable for compressed columns because all of the column files would have to be rewritten. LSM-trees can be used to write to column storage. Queries must take into account both the data stored on disk and the recently written data in the LSM-tree in memory.

3.3.4 Aggregation: Data Cubes and Materialized Views

Analytics queries often use aggregate functions in SQL (e.g. `COUNT`, `SUM`, `AVG`, `MIN`, `MAX`). It makes sense to cache the *materialized aggregates* that are used most often by queries.

A *view* in a relational database is a table-like object that stores an underlying query that is executed dynamically when a user reads from the view.

A *materialized view* is a table-like object that stores an actual copy of its query results. Materialized views need to be updated when the underlying data changes, which is why they are mainly used in OLAP, not OLTP, systems.

A *data cube* is a type of materialized view. It is a grid of materialized aggregates grouped by different dimensions.

Consider a fact table where each fact has foreign key references to two dimensions (e.g. *date* and *product*). You can draw a two-dimensional table with dates on axis and products on the other axis. Each cell contains an aggregate (e.g. `SUM`) of an attribute (e.g. `net_price`) for all facts with that date-product combination. You can apply the same aggregate along each row or column to get summary reduced by one dimension (e.g. sales on this date, regardless of product).

	32	33	34	35	...	total
140101	149.60	31.01	84.58	28.18		40710.53
140102	132.18	19.78	82.91	10.96		73091.28
140103	196.75	0.00	12.52	64.67		54688.10
140104	178.36	9.98	88.75	56.16		95121.09
...					...	
total	14967.09	5910.43	7328.85	6885.39		lots

Facts can have more than two dimensions. Data cubes can be n-dimensional.

Materialized data cubes can make certain queries very fast because the answers are precomputed, but querying a data cube is not as flexible as querying the database itself.

3.4 Summary

Disk seek time is often the bottleneck for OLTP systems. Disk bandwidth is often the bottleneck for OLAP systems.

The key idea of log-structured storage engines: they turn random-access writes into sequential writes, which enables higher write throughput on HDDs (no seeking required) and SSDs.

In OLAP, queries sequentially scan across a large number of rows, so indexes are not very helpful. It's more helpful to minimize the amount of data that the query needs to read from disk.

4 Encoding and Evolution

A change in an application's features generally requires a change in the data it stores. Changes usually need to be made gradually.

For server-side applications, a *rolling upgrade* or *staged rollout* deploys new code to a few nodes at a time. For client-side applications, users may update their software at different times. In both cases, old and new versions of the code and data may coexist in the system. We need compatibility in both

directions.

Backward compatibility: newer code can read data written by older code
Forward compatibility: older code can read data written by newer code

4.1 Formats for Encoding Data

The translation of an in-memory structure to a byte sequence that can be stored on disk or sent over a network is called *encoding* or *serialization* or *marshalling*. The reverse is called *decoding* or *deserialization* or *unmarshalling*.

4.1.1 Language-Specific Formats

Language-specific encodings are typically a bad idea. They lock you into using the same language, introduce security problems through the decoding process (which needs to be able to instantiate arbitrary classes), and do not handle compatibility well.

4.1.2 JSON, XML, and Binary Variants

JSON, XML, and CSV are textual, human-readable formats. They are popular and, despite their problems, will likely continue to be used for data interchange between organizations that need to agree on a common format.

Problems:

- JSON and XML have optional schema support, CSV does not. If schemas are not used, then data will have to be interpreted (decoded) manually in the application code.
- XML and CSV do not distinguish between numbers and digit strings. JSON does not distinguish between integers and floating-point numbers. Integers larger than 2^{53} cannot be exactly represented in an IEEE 754 double-precision floating-point number, so such numbers become inaccurate when parsed in an language that uses floating-point numbers (such as JavaScript).
- JSON and XML do not support binary strings (sequences of bytes without a character encoding). One workaround is to encode binary

data as text using Base64 and define the value as Base64-encoded in the schema (increases data size by 33%.

Some binary encodings for JSON: MessagePack, BSON, BJSON, UBJSON, BISON, Smile. Some binary encodings for XML: WBXML, Fast Infoset.

These binary encodings made specifically for JSON and XML do not prescribe a schema because JSON and XML do not require one. Thus, they need to include all the object field names in the encoded data. This means that they will not save much space compared to their textual counterparts.

4.1.3 Thrift and Protocol Buffers

Apache Thrift (developed at Facebook) and Protocol Buffers (protobuf, developed at Google) are binary encoding libraries that require a schema. This means that they do not have to encode the field names. Instead, they can encode *field tags* (numerical aliases) that are defined in the schema. This saves space.

Thrift and Protocol Buffers each come with a code generation tool that inputs a schema definition and outputs classes that implement the schema in various programming languages. Application code can call this generated code to encode and decode records of the schema.

Thrift has multiple encoding formats, such as BinaryProtocol and CompactProtocol. CompactProtocol improves upon BinaryProtocol by packing the field type and tag into a single byte and by using variable-length integers.

The `required` and `optional` keywords in the Thrift and Protocol Buffers IDLs do not affect data is encoded. The `required` keyword simply enables a runtime check that fails if the field is not set.

Field names in a schema can be changed with no issue, as encoded data never refers to field names, only field tags. Changing a field tag would make all existing encoded data invalid.

To maintain forward compatibility:

- New fields that are added to a schema must have new field tags. Old code must be able to skip over fields with new tags that it does not recognize.
- Tags from fields that have been removed can never be reused. New code must be able to skip these fields in old data.

To maintain backward compatibility:

- New fields that are added to a schema cannot be required. New code must be able to read old data that does not have these fields. Thus, all fields added after initial deployment must be optional or have a default value.
- Only optional fields can be removed. Old code must be able to read new data that does not have these fields.

Changing the datatype of a field in a schema may cause values to lose precision or be truncated. For example, if a 32-bit integer is updated to a 64-bit integer, new code will be able to fill in missing bits with zeros, but old code will not be able to correctly read any values that do not fit in 32 bits.

In Protocol Buffers, an `optional` field (single-valued) can be changed into a `repeated` field (multi-valued). New code reading old data will see a list with zero or one elements. Old code reading new data will see only the last element of the list. Because Thrift has a dedicated list type, it does not support the same evolution from single- to multi-valued fields, but it does support nested lists.

4.1.4 Avro

Avro improves on Thrift and Protocol Buffers by getting rid of field tags and datatype bytes entirely. To parse the binary data, you go through the fields in the order that they appear in the schema to find the datatype of each field. Thus, binary data can only be properly decoded if the reader is using the exact same schema that the writer used to encode the data.

If the *writer's schema* and the *reader's schema* differ, Avro resolves their differences. If fields are out of order, it matches them by field name. If a field appears in the writer's schema but not the reader's schema, it is ignored.

If a field appears in the reader's schema but not the writer's schema, it is assigned the default value defined in the reader's schema.

Unlike Thrift and Protocol Buffers, Avro does not have `optional` and `required` markers. Instead, it has union types and default values. To maintain backward and forward compatibility, you must only add or remove fields with default values. To specify that a field is nullable (optional), you must assign it a union type (e.g. `union { null, long }`).

To resolve schema differences, the reader needs to know the writer's schema. It would be too space-inefficient to send the schema along with every record, but it could reasonably be sent as part of processing many records. Some examples:

- A large file containing many records encoded with the same schema (an *object container file*) can include the schema once.
- A database containing records encoded with different schemas at different times can store the different schema versions, mark the records with a schema version number, and fetch the correct schema when decoding.
- When two processes are communicating over a bidirectional network connection, they can agree on a schema to use for the lifetime of the connection.

An Avro schema does not use tag numbers. That makes it a better choice for dynamically generated schemas as differences between schemas can be resolved by field name instead of by tag number. When a database schema changes, a new Avro schema can be generated from it without worrying about maintaining consistent tag numbers.

Avro provides optional code generation for statically typed programming languages (create classes with a type that conforms to the Avro schema, put the decoded data into objects that can be manipulated in memory).

4.1.5 The Merits of Schemas

Binary encodings based on schemas:

- Are compact (because they do not have to encode field names)

- Are self-documenting (the schema describes the encoding and is required for decoding)
- Help check for backward and forward compatibility of recent schema changes when previous schema versions are stored in a database
- Can generate code from the schema in a statically typed programming language

4.2 Modes of Dataflow

4.2.1 Dataflow Through Databases

Consider this scenario:

- a new writer *p1* encodes an object and writes a record with a new field to a database
- an old reader *p2* decodes the record and reads the data into an in-memory object, ignoring the new field
- *p2* updates one of the object's values, encodes the object (without the new field), and writes the record to the database
- the new field is lost

Conclusion: you may need to ensure that your application code does not break forward compatibility.

Most relational databases allow simple schema changes, such as adding a column with a null default value, without rewriting existing data. Schema evolution like this allows an entire database to appear as if it was encoded with a single schema, even though the underlying storage may contain records encoded with various historical versions of the schema.

Let's say you want to make a copy of an entire database for archival purposes or for analysis in a warehouse. You should consider:

- Encoding the copy using a single schema (data could then be stored in an Avro object container file)
- Encoding the copy in a column-oriented format like Parquet, if you intend to do analysis on it

4.2.2 Dataflow Through Services: REST and RPC

Some examples of clients: web browsers, native applications, client-side JavaScript applications running inside of a web browser (*Ajax*), servers making requests of other servers or databases.

In a *service-oriented architecture* (SOA) or *microservices architecture*, each service is owned by one team, is independently deployable, and is used by other teams. Old and new versions of servers and clients may be running at the same time, so the data encoding used by servers and clients must be compatible across versions of the service API.

When HTTP is used to talk to a service, it is called a *web service*. But web services are not only used on the web. Some examples of web services:

- A user application making requests to a service over HTTP.
- A service making requests to another service owned by the same organization, often within the same datacenter.
- A service making requests to a service owned by a different organization (data interchange).

There are many ways to make an API call over a network. All of them involve making a *remote procedure call* (RPC), which is when a machine calls a function that is defined on a different machine (in a separate address space) in its network.

There are two popular approaches to web services: *REST* and *SOAP*. REST is not a protocol, but a design philosophy that builds on the principles of HTTP. SOAP is an XML-based protocol for making network API requests.

A SOAP API is described using Web Services Description Language (WSDL). A RESTful API can be described using OpenAPI (Swagger).

The RPC model tries to make sending a request to a remote network service look the same as calling a local function (*location transparency*). This is a flawed approach because they are fundamentally different.

Ways that remote function calls are different than local function calls:

- Unlike a local call, a remote call is unpredictable and may fail due to problems outside of your control.
- A local call either returns a result, throws an exception, or never returns (infinite loop or crash). In addition to these outcomes, a remote call can also return without a result, due to *timeout*.
- If a remote call fails, it could be that the request got through but the response was lost. If you retry the remote call, you will cause the actions to be performed twice, unless you build a mechanism for deduplication (idempotence) into the protocol.
- A remote call is much slower than a local call, and its latency is wildly variable.
- The arguments of a remote call must be encoded into a sequence of bytes.
- A remote service may be implemented in a different programming language than the client application making the call. In this case, the RPC framework must translate datatypes from one language to another. This can cause issues with numbers.

Thrift and Avro support RPC. gRPC is an RPC implementation that uses Protocol Buffers.

A custom RPC protocol could have better performance than REST, but mostly only in the context of making intra-organizational requests (typically within the same datacenter).

For dataflow through services, it is reasonable to assume that all servers will be updated before any client is updated. Requests must be backward-compatible, responses must be forward-compatible.

The compatibility properties of an API depend on the encoding it uses:

- RESTful APIs usually use JSON and URI-encoded request parameters. Adding optional request parameters and adding new response fields both maintain compatibility.
- SOAP APIs use XML. Evolution is possible but tricky.

- Thrift, gRPC, and Avro RPCs can be evolved according to the compatibility rules of their respective encodings.

If the provider of a service cannot force its clients to update, it will need to maintain compatibility indefinitely. If a compatibility-breaking change is needed, the provider will probably have to maintain multiple versions of the service (clients will select a version to use via URL, header, or selection data stored server-side).

4.2.3 Message-Passing Dataflow

In an *asynchronous message-passing system*, a *producer* sends a message to a *message broker* or *message queue* or *topic*, which stores the message temporarily, and the broker sends the message to one or more *consumers* or *subscribers*. The topic is one-way; producers typically do not expect a response.

Advantages of using a message broker compared to direct RPC:

- Acts as a buffer if the recipient is currently unavailable
- Can automatically redeliver messages
- Abstracts away the IP address and port number of the recipient. Logically decouples sender from recipient.
- One-to-many message distribution (multicast)

Topics can be chained together. A consumer can send a message back to a producer via a *reply queue*. If an old consumer republishes messages, you may need to make sure that it does not erase fields added by a new producer.

A message broker typically does not enfore a particular data model or encoding.

The *actor model* is a programming model for concurrency in a single process. Each actor represents one entity, it may have local (non-shared) state, and it communicates with other actors by sending and receiving asynchronous messages. Messages can be lost. Because actors process only one message at

a time, they can be scheduled independently.

In a *distributed actor framework*, messages can be sent between actors on the same node or across different nodes. In the latter case, messages will be transparently encoded, sent over the network, and decoded on the other side. If you want to be able to perform a rolling upgrade for your actor-based application, you have to ensure backward and forward compatibility.

Distributed actor frameworks: Akka, Orleans, Erlang OTP.

4.3 Summary

Assume we have a service that is replicated across multiple nodes. We would like to be able to perform rolling upgrades on this system because they allow new versions of a service to be released without downtime (encouraging frequent small releases over rare big releases) and make deployments less risky (faulty releases can be rolled back before they affect all users). To do this, we need to ensure all data flowing in the system is encoded in a way that provides backward and forward compatibility.

Language-specific encodings are bad. Textual encodings are okay, but popular. Binary encodings are compact, efficient, and have clearly defined compatibility semantics.

5 Replication

Replication means storing the same data on multiple machines that are connected via a network. You might want to do this for:

- Geographic proximity (reduced latency)
- Resiliency (increased availability)
- Scale (increased read throughput)

The hard part of replication is handling changes to replicated data.

5.1 Leaders and Followers

Single-leader replication:

1. One replica is the *leader*. All writes are sent to the leader, which writes the data to its local storage.
2. All other replicas are *followers*. When the leader writes to its storage, it also sends the changes to its followers as part of a *replication log* or *change stream*. Each follower applies the writes in the same order as on the leader.
3. Reads can go to any replica.

5.1.1 Synchronous Versus Asynchronous Replication

Synchronous replication: the leader waits until a follower confirms that it received the write before reporting success to the client

- Follower is guaranteed to have up-to-date, consistent data
- If the follower does not respond, the write cannot be processed. The leader must block all writes until the follower is available.

Asynchronous replication: the leader does not wait for a response from the follower

- Leader can continue to process writes, even if all of its followers have fallen behind
- If the leader fails, any writes that have not been replicated are lost
- Writes are not durable, even though a client may have been told a write was successful
- Very popular configuration

Semi-synchronous replication: one follower is synchronous

- Two nodes are guaranteed to have an up-to-date copy of the data
- If the synchronous follower is slow, one of the asynchronous followers is made synchronous

5.1.2 Setting Up New Followers

1. Take a consistent snapshot of the leader's database at some point in time (preferably without locking the database).
2. Copy the snapshot to the new follower node.
3. The follower connects to the leader and requests all changes since the snapshot was taken. The snapshot is associated with a position in the leader's replication log (*log sequence number* or *binlog coordinates*).
4. When the follower processes the backlog of changes, it has *caught up.*

5.1.3 Handling Node Outages

If a follower crashes, it knows the last transaction that was processed before the crash. It can connect to the leader and request all changes that have occurred since that transaction.

If the leader fails, one of the followers needs to become the leader. This is called *failover*, and it usually involves the following steps:

1. *Determining that the leader has failed.* Nodes send heartbeat messages to each other. If the leader does not respond for some period of time, it is assumed to be dead.
2. *Choosing a new leader.* Can be done in various ways, such as via election or *controller node.* The best candidate is usually the most up-to-date follower (minimizes data loss). This is a consensus problem.
3. *Reconfiguring the system to use the new leader.* Clients need to send their requests to the new leader. If the old leader come back, it needs to become a follower.

Potential failover problems:

- If asynchronous replication is used, the new leader might not have all the writes from the old leader. If the old leader come back, what should happen to those writes? The new leader may have received conflicting writes in the meantime. The most common solution is to discard the writes, which may violate the client's durability expectations.

- Discarding writes can be dangerous if other storage systems need to be coordinated with the database contents. The writes could have been successfully written to a cache, for example. Now, the cache and database are inconsistent.
- Split brain
- Timeouts

There are no easy solutions to these problems. For this reason, some operations teams prefer to perform failovers manually.

5.1.4 Implementation of Replication Logs

5.2 Problems with Replication Lag

5.2.1 Reading Your Own Writes

5.2.2 Monotonic Reads

5.2.3 Consistent Prefix Reads

5.2.4 Solutions for Replication Lag

5.3 Multi-Leader Replication

5.3.1 Use Cases for Multi-Leader Replication

5.3.2 Handling Write Conflicts

5.3.3 Multi-Leader Replication Topologies

5.4 Leaderless Replication

5.4.1 Writing to the Database When a Node Is Down

5.4.2 Limitations of Quorum Consistency

5.4.3 Sloppy Quorums and Hinted Handoff

5.4.4 Detecting Concurrent Writes

5.5 Summary

A Appendix for Chapter 4

Below are a number of binary encodings (byte sequences) of the following JSON record:

```
{
    "userName": "Martin",
    "favoriteNumber": 1337,
    "interests": ["daydreaming", "hacking"]
}
```

Strings are encoded according to UTF-8.

A.1 MessagePack

66 bytes:

- `83` (object, 3 entries)
- `a8` (string, length 8)
- `75 73 65 72 4e 61 6d 65` (`userName`)
- `a6` (string, length 6)
- `4d 61 72 74 69 6e` (`Martin`)
- `ae` (string, length 14)
- `66 61 76 6f 72 69 74 65 4e 75 6d 62 65 72` (`favoriteNumber`)
- `cd` (uint16)
- `05 39` (`1337`)
- `a9` (string, length 9)
- `69 6e 74 65 72 65 73 74 73` (`interests`)
- `92` (array, 2 entries)
- `ab` (string, length 11)

- `64 61 79 64 72 65 61 6d 69 6e 67` (`daydreaming`)
- `a7` (string, length 7)
- `68 61 63 6b 69 6e 67` (`hacking`)

A.2 Thrift

A schema for the JSON record, written in the Thrift interface description language (IDL), would look like this:

```
struct Person {
    1: required string       userName,
    2: optional i64          favoriteNumber,
    3: optional list<string> interests
}
```

A.2.1 BinaryProtocol

59 bytes:

- `0b` (type 11, string)
- `00 01` (field tag = 1)
- `00 00 00 06` (length 6)
- `4d 61 72 74 69 6e` (`Martin`)
- `0a` (type 10, i64)
- `00 02` (field tag = 2)
- `00 00 00 00 00 00 05 39` (`1337`)
- `0f` (type 15, list)
- `00 03` (field tag = 3)
- `0b` (item type 11, string)
- `00 00 00 02` (2 list items)

- `00 00 00 0b` (length 11)
- `64 61 79 64 72 65 61 6d 69 6e 67` (`daydreaming`)
- `00 00 00 07` (length 7)
- `68 61 63 6b 69 6e 67` (`hacking`)
- `00` (end of struct)

A.2.2 CompactProtocol

34 bytes:

- `18` (field tag = 1; type 8, string)
- `06` (length 6)
- `4d 61 72 74 69 6e` (`Martin`)
- `16` (field tag += 1; type 6, i64)
- `f2 14` (`1337`, see p.119 of original text for how)
- `19` (field tag += 1; type 9, list)
- `28` (2 list items; item type 8, string)
- `0b` (length 11)
- `64 61 79 64 72 65 61 6d 69 6e 67` (`daydreaming`)
- `07` (length 7)
- `68 61 63 6b 69 6e 67` (`hacking`)
- `00` (end of struct)

A.3 Protocol Buffers

A schema for the JSON record, written in the Protocol Buffers interface description language (IDL), would look like this:

```
message Person {
    required string user_name      = 1
    optional int64 favorite_number = 2
    repeated string interests      = 3
}
```

33 bytes:

- `0a` (field tag = 1; type 2, string `00001 010`)
- `06` (length 6)
- `4d 61 72 74 69 6e` (`Martin`)
- `10` (field tag = 2; type 0, varint `00010 000`)
- `b9 0a` (`1337`, see p.120 of original text for how)
- `1a` (field tag = 3; type 2, string `00011 010`)
- `0b` (length 11)
- `64 61 79 64 72 65 61 6d 69 6e 67` (`daydreaming`)
- `1a` (field tag = 3; type 2, string `00011 010`)
- `07` (length 7)
- `68 61 63 6b 69 6e 67` (`hacking`)

A.4 Avro

```
record Person {
    string               userName;
    union { null, long } favoriteNumber;
    array<string>        interests;
}
```

32 bytes:

- `0c` (length 6)
- `4d 61 72 74 69 6e` (`Martin`)
- `02` (union branch 1 long)
- `f2 14` (`1337`, see p.123 of original text for how)
- `04` (2 array items)
- `16` (length 11)
- `64 61 79 64 72 65 61 6d 69 6e 67` (`daydreaming`)
- `0e` (length 7)
- `68 61 63 6b 69 6e 67` (`hacking`)
- `00` (end of array)

B Big Ideas

Well-chosen indexes speed up read queries, but every index slows down writes.

C Glossary

Application programming interface (API):

A key principle of design is to prohibit access to all resources by default. An *interface* is a well-defined entry point, or a set thereof, that allows access to resources. An API is an interface that supports application programming. It is an interface between computer programs.

API documentation tells you how to use an API (information on endpoints, parameters, response formats, and authentication requirements).

An API specification is a technical description of an API's behavior and the data models it uses. It is usually written in JSON or XML according to a

standard (such as OpenAPI or Swagger). It tells you how to implement an API.

Database index:

An index is a tool that is used for locating information or resources. You give it information about what you're looking for, it gives you information on where you can find what you're looking for.

From the perspective of a database user: An index is a database object that improves the speed of data retrieval at the cost of additional writes and storage space. It is a map whose values are primary data in the database.

From the perspective of a storage engine developer: An index is a data structure auxiliary to the primary database that contains metadata derived from the primary data. It is a map whose values are references to locations on disk where primary data is stored. (In the case of a *clustered index*, the values are the primary data themselves.)

In publishing, an index is a collection of words or phrases (*headings*) and associated *pointers* (such as page numbers, which are a form of locational metadata) to useful information related to the headers.

Foreign key: a column in a table that refers to a primary key in some other table.