1. Reliable, Scalable, and Maintainable Applications

Many applications today are *data-intensive*, as opposed to *compute-intensive*.

For example, many applications need to:

- Store data so that they, or another application, can find it again later (*databases*)
- Remember the result of an expensive operation, to speed up reads (*caches*)
- Allow users to search data by keyword or filter it in various ways (*search indexes*)
- Send a message to another process, to be handled asynchronously (*stream processing*)
- Periodically crunch a large amount of accumulated data (*batch processing*)

You usually create a new, special-purpose data system from smaller, general-purpose components.

Reliability is continuing to work correctly, even when things go wrong. The things that can go wrong are called *faults*, and 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 a *failure* is when the system, as a whole, stops providing the required service to the user.

Scalability is the term we use to describe a system's ability to cope with increased load. Scalability is based on load parameters and performance.

Performance Measure: The measures of *95th*, *99th*, and *99.9th* percentiles are common (abbreviated *p95*, *p99*, and *p999*). They are the response time thresholds at which 95%, 99%, or 99.9% of requests are faster than that particular threshold. For example, if the 95th percentile response time is 1.5 seconds, that means 95 out of 100 requests take less than 1.5 seconds, and 5 out of 100 requests take 1.5 seconds or more.

Approaches for coping with Load: S*caling up* (*vertical scaling*, moving to a more powerful machine) and *scaling out* (*horizontal scaling*, distributing the load across multiple smaller machines).

Accidental Complexity Vs Essential Complexity: We define complexity as accidental if it is not inherent in the problem that the software solves (as seen by the users) but arises only from the implementation. If the complexity is inherent to the problem, it is called the Accidental Complexity.

2. Data Models and Query Language

Data model is the format in which the application developer gives the data to the database. Query language is the mechanism by which the application developer can ask for it again later.

Most applications are built by layering one data model on top of another. For each layer, the key question is: how is it *represented* in terms of the next-lower layer?

Objects, Data Structs -> JSON RDBMS NoSQL -> Bytes -> Magnetic fields, electric pulses etc.

In a complex application there may be more intermediary levels, such as APIs built upon APIs, but the basic idea is still the same: each layer hides the complexity of the layers below it by providing a clean data model.

We will look at relational, document and graph data models and query languages in this section.

It seems likely that in the foreseeable future, relational databases will continue to be used alongside a broad variety of nonrelational datastores—an idea that is called ***polyglot persistence***

If data is stored in relational tables, an awkward translation layer is required between the objects in the application code and the database model of tables, rows, and columns. The disconnect between the models is sometimes called an ***impedance mismatch***

When it comes to representing many-to-one and many-to-many relationships, relational and document databases are not fundamentally different: in both cases, the related item is referenced by a unique identifier, which is called a ***foreign key*** **in the relational model** and a ***document reference*** **in the document model.**

Why a foreign key? The advantage of using an ID is that because it has no meaning to humans, it never needs to change: the ID can remain the same, even if the information it identifies changes. Anything that is meaningful to humans may need to change sometime in the future—and if that information is duplicated, all the redundant copies need to be updated.

The main arguments in favor of the document data model are schema flexibility, better performance due to locality, and that for some applications it is closer to the data structures used by the application. The relational model counters by providing better support for joins, and many-to-one and many-to-many relationships.

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

A document is usually stored as a single continuous string , encoded as JSON, XML, or a binary variant thereof (such as MongoDB's BSON). If your application often needs to access the entire document there is a performance advantage to this *storage locality*. If data is split across multiple tables multiple index lookups are required to retrieve it all, which may require more disk seeks and take more time.

The locality advantage only applies if you need large parts of the document at the same time. The database typically needs to load the entire document, even if you access only a small portion of it, which can be wasteful on large documents. On updates to a document, the entire document usually needs to be rewritten—only modifications that don't change the encoded size of a document can easily be performed in place. For these reasons, it is generally recommended that you keep documents fairly small and avoid writes that increase the size of a document.

An **imperative language** tells the computer to perform certain operations in a certain order. You can imagine stepping through the code line by line, evaluating conditions, updating variables, and deciding whether to go around the loop one more time.

In a **declarative query language**, like SQL or relational algebra, you just specify the pattern of the data you want—what conditions the results must meet, and how you want the data to be transformed but not *how* to achieve that goal. It is up to the database system's query optimizer to decide.

In a web browser, using declarative CSS styling is much better than manipulating styles imperatively in JavaScript.

3. Storage Engines

The two most popular families of storage engines: *log-structured* storage engines such as SStables, and *page-oriented* storage engines such as B-trees.

A naïve approach to storing key-value data is to simply append each new entry to an immutable text file. This method excels in write performance, as new data is continuously appended without modifying existing content. However, retrieval operations become inefficient, requiring **O(N)** time complexity, which is unscalable as the dataset grows.

To enhance read performance, databases introduce an **index**, a secondary data structure derived from the primary data file. The index provides an efficient lookup mechanism, allowing queries to retrieve values with significantly lower latency. Notably, indexes in most databases can be added or removed without altering the underlying data, as they only influence query performance. This introduces a fundamental trade-off in storage system design:

- **Indexes accelerate read queries** by reducing lookup time.
- **Indexes slow down write operations**, as each new record requires index updates.

Hash Table Indexing

A basic indexing strategy is to use a **hash table**, where each key from the primary data file maps to an **offset location** in the file. This allows direct access to the value, significantly reducing read time.

Compaction and Merging

To manage storage efficiently, databases periodically **compact** the primary data by retaining only the most recent versions of updated records. During this process, fragmented data segments are **merged** into a consolidated file, reducing redundant storage. Each time compaction and merging occur, the **hash table index must be updated** to reflect the new offsets of keys.

However, this approach presents two major challenges:

1. **Memory Overhead** – Since the hash table maintains a 1:1 mapping with the records in the primary data file, it can grow excessively large.
2. **No Support for Range Queries** – Hash tables are optimized for exact key lookups but are inefficient for queries that require scanning a range of keys.

We can address these two problems by creating a **Sorted String Table** (SSTable). The sort for the SSTable can be done through

4. Encoding and Schema Evolution

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.

Backward compatibility is normally not hard to achieve: as author of the newer code, you know the format of data written by older code, and so you can explicitly handle it (if necessary, by simply keeping the old code to read the old data). Forward compatibility can be trickier, because it requires older code to ignore additions made by a newer version of the code.

Programs usually work with data in (at least) two different representations:

1. In memory, data is kept in objects, structs, lists, arrays, hash tables, trees, and so on. These data structures are optimized for efficient access and manipulation by the CPU (typically using pointers).
2. When you want to write data to a file or send it over the network, you have to encode it as some kind of self-contained sequence of bytes (for example, a JSON document). Since a pointer wouldn't make sense to any other process, this sequence-of-bytes representation looks quite different from the data structures that are normally used in memory.

Thus, we need some kind of translation between the two representations. The translation from the in-memory representation to a byte sequence is called ***encoding*** (also known as ***serialization*** or ***marshalling***), and the reverse is called ***decoding*** (***parsing***, ***deserialization***, ***unmarshalling***).

Serialization can be achieved in three ways.

1. Language specific formats - For eg., Java has java.io.Serializable, Python has pickle etc.
2. JSON, XML and Binary Variants like Message Pack (schema less), Thrift and Buffer protocols, Avro etc.

Schemas inevitably need to change over time. We call this ***schema evolution***.

With Avro, when an application wants to encode some data (to write it to a file or database, to send it over the network, etc.), it encodes the data using whatever version of the schema it knows about—for example, that schema may be compiled into the application. This is known as the ***writer's schema***.

When an application wants to decode some data (read it from a file or database, receive it from the network, etc.), it is expecting the data to be in some schema, which is known as the ***reader's schema***.

H ow does the reader know the writer's schema with which a particular piece of data was encoded? We can't just include the entire schema with every record, because the schema would likely be much bigger than the encoded data, making all the space savings from the binary encoding futile.

Forward compatibility means that 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.

The answer depends on the context in which Avro is being used.

1. **Large file with lots of records:** In this case, the writer of that file can just include the writer's schema once at the beginning of the file. Avro specifies a file format (object container files) to do this.
2. **Database with individually written records:** In a database, different records may be written at different points in time using different writer's schemas. The simplest solution is to include a version number at the beginning of every encoded record, and to keep a list of schema versions in your database.
3. **Sending records over a network connection:** When two processes are communicating over a bidirectional network connection, they can negotiate the schema version on connection setup and then use that schema for the lifetime of the connection.

Avro can dynamically generate schemas whereas Thirift and Buffer protocols cannot.

Some of the most common ways how data flows between processes:

- Via databases
- Via service calls
- Via asynchronous message passing

Web services:

When HTTP is used as the underlying protocol for talking to the service, it is called a *web service*. This is perhaps a slight misnomer, because web services are not only used on the web, but in several different contexts.

There are two popular approaches to web services: ***REST*** and ***SOAP***.

REST is not a protocol, but rather a design philosophy that builds upon the principles of HTTP. It emphasizes simple data formats, using URLs for identifying resources and using HTTP features for cache control, authentication, and content type negotiation. An API designed according to the principles of REST is called *RESTful*.

By contrast, SOAP is an XML-based protocol for making network API requests. Although it is most commonly used over HTTP, it aims to be independent from HTTP and avoids using most HTTP features. The API of a SOAP web service is described using an XML-based language called the Web Services Description Language, or WSDL. WSDL enables code generation so that a client can access a remote service using local classes and method calls.

The problems with remote procedure calls (RPCs)

1. A local function call is predictable and either succeeds or fails, depending only on parameters that are under your control where as a network request is not.
2. A local function call either returns a result, or throws an exception, when it isn't able to process. But a network can process and fail while sending the response back.
3. A network request is much slower than a function call, and its latency is also wildly variable
4. When you call a local function, you can efficiently pass it references (pointers) to objects in local memory. When you make a network request, all those parameters need to be encoded into a sequence of bytes that can be sent over the network

Distributed Data

There are various reasons why you might want to distribute a database across multiple machines:

Scalability

If your data volume, read load, or write load grows bigger than a single machine can handle, you can potentially spread the load across multiple machines.

Fault tolerance/high availability

If your application needs to continue working even if one machine goes down, you can use multiple machines to give you redundancy. When one fails, another one can take over.

Latency

If you have users around the world, you might want to have servers at various locations worldwide so that each user can be served from a data center that is geographically close to them. That avoids the users having to wait for network packets to travel halfway around the world.

Scaling Strategies

Many CPUs, many RAM chips, and many disks can be joined together under one operating system, and a fast interconnect allows any CPU to access any part of the memory or disk. This is called **Shared Memory Architecture**. The problem with a shared-memory approach is that the cost grows faster than linearly. Another approach is the **shared-disk architecture**, which uses several machines with independent CPUs and RAM, but stores data on an array of disks that is shared between the machines, which are connected via a fast network. This architecture can be used for some data warehousing workloads, but contention and the overhead of locking limit the scalability of the shared-disk approach.

Shared-nothing architectures (sometimes called *horizontal scaling* or *scaling out*) have gained a lot of popularity. In this approach, each machine or virtual machine running the database software is called a *node*. Each node uses its CPUs, RAM, and disks independently. Any coordination between nodes is done at the software level, using a conventional network. While a distributed shared-nothing architecture has many advantages, it usually also incurs additional complexity for applications and sometimes limits the expressiveness of the data models you can use. We focus on shared-nothing architectures—not because they are necessarily the best choice for every use case, but rather because they require the most caution from you, the application developer.

5. Replication

There are two common ways data is distributed across multiple nodes:

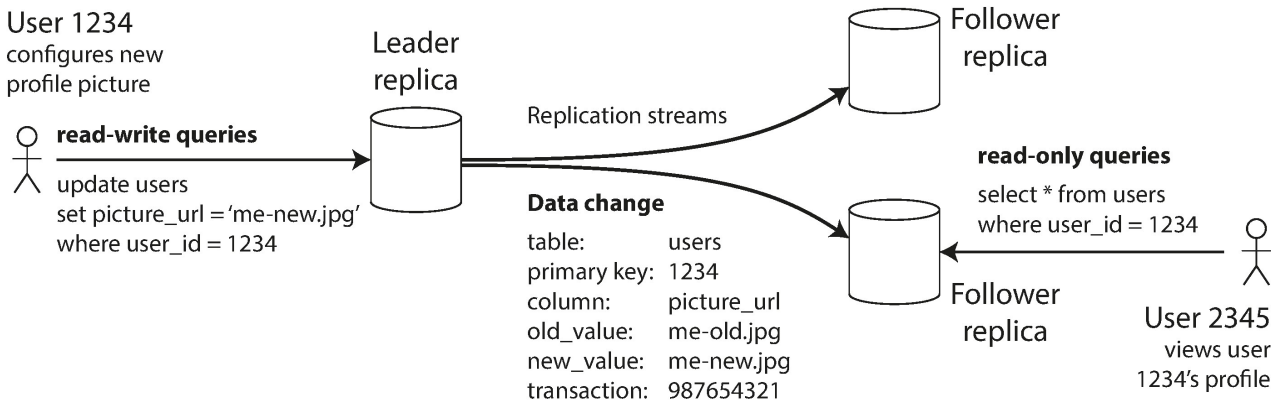
Replication: Keeping a copy of the same data on several different nodes, potentially in different locations. Replication provides redundancy: if some nodes are unavailable, the data can still be served from the remaining nodes. Replication can also help improve performance.

Partitioning: Splitting a big database into smaller subsets called *partitions* so that different partitions can be assigned to different nodes (also known as *sharding*).

All of the difficulty in replication lies in handling *changes* to replicated data, and that's what this chapter is about. We will discuss three popular algorithms for replicating changes between nodes**: *single-leader*, *multi-leader***, and ***leaderless*** replication.

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

Single Leader



One replica is designated the *leader* (also known as *master* or *primary*). Other replicas are known as *followers* (*read replicas*, *slaves*, *secondaries*, or *hot standbys*). Each follower takes the replication log from the leader and updates its local copy of the database accordingly. An important detail of a replicated system is whether the replication happens *synchronously* or *asynchronously*.

The disadvantage of synchronous replication is that if the follower doesn't respond, the write cannot be processed. The leader will block all writes and wait until the synchronous replica is available again. For that reason, it is impracticable for all followers to be synchronous. If the synchronous follower becomes unavailable or slow, one of the asynchronous followers is made

synchronous. This guarantees that you have an up-to-date copy of the data on at least two nodes: the leader and one synchronous follower.

Follower failure is easy to manage. The follower just has to catchup with the leader from the last snapshot until the most recent update in the replication log.

But the leader failure is complicated. usually consists of the following steps:

1. Determining that the leader has failed.
2. Choosing the new leader.
3. Reconfiguring the system to use the new leader.
4. Making the old leader to accept the new leader when it comes up. Avoiding split brain.

Implementation of Replication

1. **Statement based replication**. *Disadvantage*: Non deterministic functions generate different values in replicas.
2. **Write ahead log (WAL) shipping** (in B-Tree storage engines and Log Segments in SSTable). *Disadvantage*: Logs are at a granular, byte level details and will make replication tightly coupled with replication and demands all nodes run the same version of software.
3. **Logical log replication**: To avoid the above tight coupling we create a separate log called logical log for replication. It will have values of all columns for inserted rows, unique identifier for deleted rows and values of changed columns along with unique identifier for updated rows.

A logical log format is also easier for external applications to parse. This aspect is useful if you want to send the contents of a database to an external system, such as a data warehouse for offline analysis, or for building custom indexes and caches. This technique is called ***change data capture.***

Specificity in Replication: If you want to only replicate a subset of the data, then you may need to move replication up to the application layer. This can be done by tools like Oracle GoldenGate, which interprets the database log, or by triggers and stored procedures.

The lag in replication a.k.a *eventual consistency* is a problem when

1. **Reading your own writes**. We can address this by routing the writer to leader node even for reading until eventual consistency is reached. This is called *read-your-writes* or *read-after-write consistency.*

2. **When user makes several reads**, it is possible if a user makes several reads from different replicas.