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. 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.