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')` $\implies$ `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

A Big Ideas