Introduction to Spanner

Learn why do we need a strongly consistent, distributed database where replicas can be anywhere on Earth.

We'll cover the following

A history of distributed databases

It was a system designer’s dream to build a globally distributed database with all the good features of a traditional relational database like strong consistency, ability to do complex transactions, consistent snapshots, and many more. However, achieving the features above with good performance and high availability proved hard. In pursuit of that dream, we had many different kinds of NoSQL databases.

We had a significant leap forward in realizing this dream with Google’s Spanner system. It is interesting how Spanner controlled the skew on clocks and utilized high-quality network infrastructure, to provide a globally distributed database with strongly consistent reads and writes. We will study this fascinating innovation in this chapter in detail.

Motivation

NoSQL databases are widely used for their benefits like flexible and evolving data models, scalability, and high performance. Even though NoSQL prioritizes scalability and performance, it is unable to ensure strong data consistency (primarily due to the challenges of the CAP theorem).

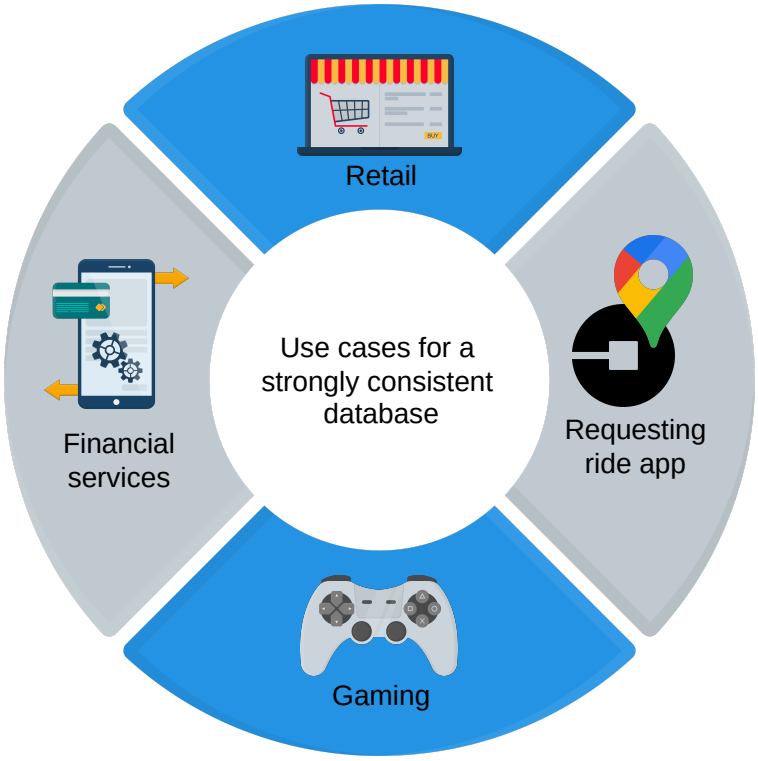
When NoSQL databases prioritize scalability, performance, and availability, they often trade off strong data consistency, which is a consequence of the CAP and PACELC theorems.

For example, re-entering the same dataset in NoSQL databases might be accepted without an error being thrown, but relational databases prevent duplicate rows from being added via integrity checks.

Most NoSQL solutions trade off strong consistency for high availability under network partitions (CAP theorem). Some applications' classes are hard to develop using loose consistency guarantees, for example, eventual consistency. The following use cases where strong consistency and high availability are the requirements of the applications:

- **Financial services:** Regulatory constraints and high client expectations make this industry particularly challenging. There must be constant, error-free communication between banks, fintech companies, and government agencies. Applications like payment gateways and online banking have the added pressure of processing hundreds of millions of transactions reliably while also applying complex anti-fraud and settlement measures. Coping with this constant influx of data requires meticulous re-architecturing of legacy systems. NoSQL cannot provide strong consistency with high availability, which is why using a relational database is a better choice.
- **Retail:** In today's omnichannel retail environment, a retailer's database must be able to manage online orders and returns as well as in-store pickups, loyalty programs, retail supply chains, inventories, and more. Dynamic pricing and just-in-time delivery both make inventory management more difficult. A potential customer is lost to a rival business when one's business runs out of goods. It is also important to anticipate seasonal volume increases. Dealing

with sudden spikes in demand needs elastic scaling, which means paying for more resources only for as long as we're using them. The billing and inventory management systems should be able to keep up with variable traffic patterns all the time (normal and spike cases).



Use cases for a strongly consistent database

- **Requesting ride app:** In today's world, technology is very advanced. It requires the algorithms and machines to perform real-time decision-making to ease the user experience. For example, Google Maps require real-time route calculation. It also needs to optimize the routes according to real-time data analysis. Moreover, delivering monitoring and performance insights data for analysis is also critical. Therefore, we need to manage a huge amount of data with availability to improve user experience. For example, Uber started using Spanner, which is a strongly consistent database with relational properties. Uber needed horizontal scalability and reduced operating overhead without sacrificing transactional integrity, and Spanner was able to facilitate them.
- **Gaming:** No other sector puts databases' scalability to the test like the gaming business. Millions of people may try out a new game on its release day, and every one of them is a potential source of income via in-game microtransactions and premium content. This group has zero tolerance for

boredom or delays. Customers can and will go elsewhere if there are delays in accessing a game, or the experience varies depending on where they are located. Therefore, there is a need to scale out to handle spikes in traffic.

Google Spanner

Google Spanner is a large-scale database system. It accommodates worldwide OLTP (online transaction processing) installations, SQL semantics, horizontal scalability, high availability, and transactional consistency.

Accessibility and reliability are two key selling points for Google Spanner—a cloud database service. These properties are commonly considered in conflict with one another, with data designers typically making decisions to promote either availability or consistency. The CAP theorem is the most eloquent characterization of the trade-off. It has served as the theoretical foundation for the widespread adoption of NoSQL databases to address the twin challenges of high availability and scalability in modern online and cloud architectures. Google Spanner blends SQL and NoSQL characteristics to ensure system availability and data consistency.

A strongly consistent database is a need for ever-changing technology. We will learn to design such a database in this chapter.

> **Note:** One can argue that we can create a viable solution using a NoSQL database for many, if not all use cases. However, with NoSQL solutions, programmers often need to understand how the underlying data store has made different trade-offs and other assumptions. On the other hand,

> strongly consistent databases provide a simpler end-programmer interface where we have a standardized SQL dialect to communicate with the database, and programmers might take the underlying system as a black box. Reasoning about a program based on a strongly consistent store is much easier compared to other consistency models.

Requirements

The requirements for designing a highly available and strongly consistent database are as follows:

Functional requirements

We want to provide relational database-like data manipulation capabilities. A few requirements include:

- **Isolated transaction:** We want that all committed transactions are treated as if they were processed sequentially and in order rather than in parallel.
- **Efficient data reading:** We want non-blocking reads like taking data snapshots in the past. We also want lock-free read transactions and always want to read an up-to-date value from the database.
- **Atomic commit across shards:** The massive amount of data is not stored on a single node. Instead, it is split into subsets of data and assigned to a replica. This way, we keep data on multiple nodes. It means we can have a transaction that needs to read and write data on multiple nodes. In such a case, we need an atomic commit, that is, the change is committed or aborted on all nodes.

Non-functional requirements

The non-functional requirements are as follows:

- **Consistency:** The system should support strong consistency. It means that any changes made to the data are replicated across all nodes synchronously and searched with the same results every time.

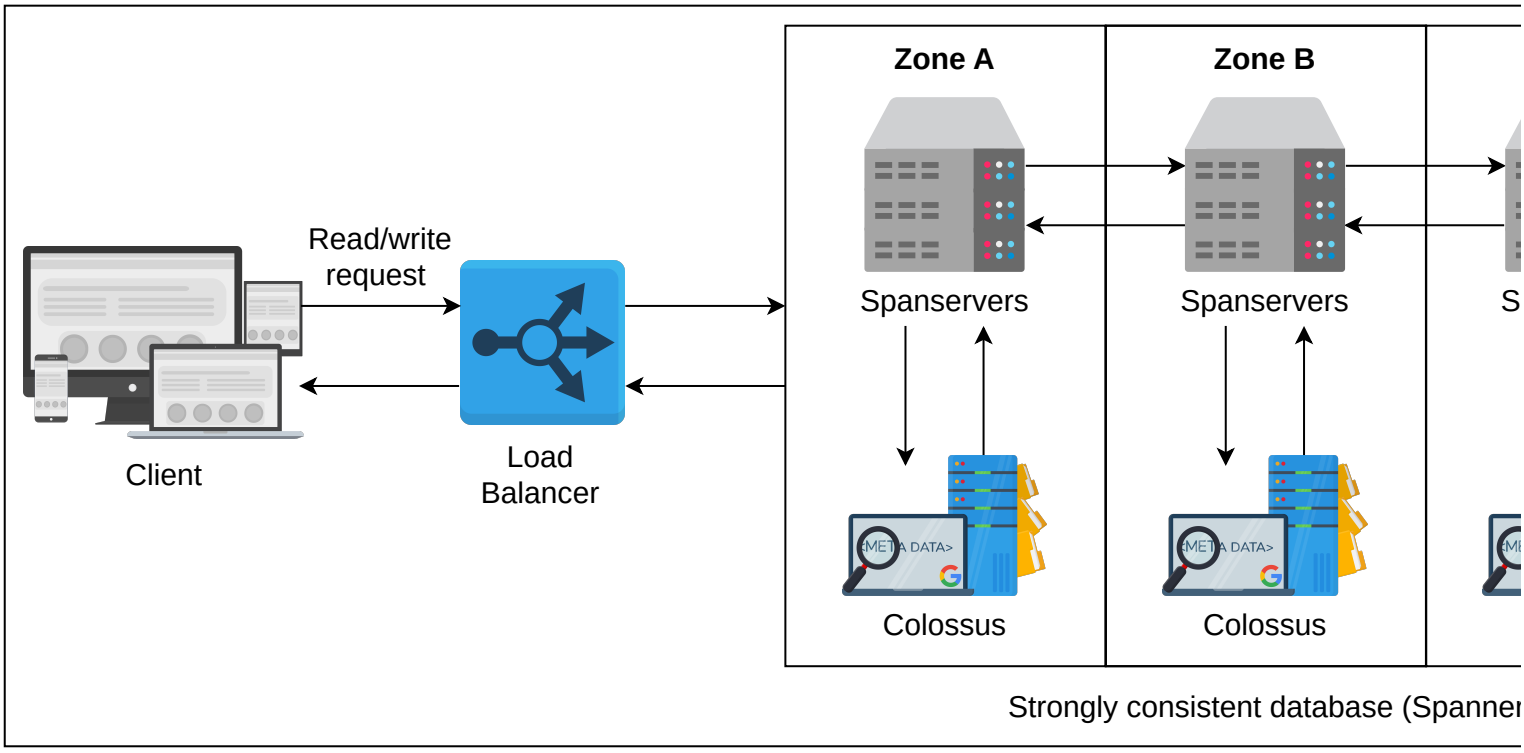
- **Scalability:** The database should be scalable to accommodate billions of machines and millions of users over hundreds of data centers.
- **Availability:** The system should be highly available. High availability requires that we need to ensure continuous service. It should provide up to 99.999% availability.
- **Fault tolerance:** The system should operate uninterrupted despite server or component failures.

High-level design

The components involved in the high-level design of our system are as follows:

- **Client:** This is the Spanner system user that utilizes the system's database-like functions.
- **Load balancer:** This balances the client's load.
- **Zone:** A zone consists of a single zone handler and between 100 and a few thousand servers. A data center can contain one or more than one zone.
- **Spanserver:** This consists of tablets. A Spanner tablet is a data structure like Bigtable's tablet. Each server has between 100 and 1000 instances of tablets.
- **Colossus:** This is a distributed file system that stores a tablet's state in a format similar to B-tree-like and write-ahead logs.

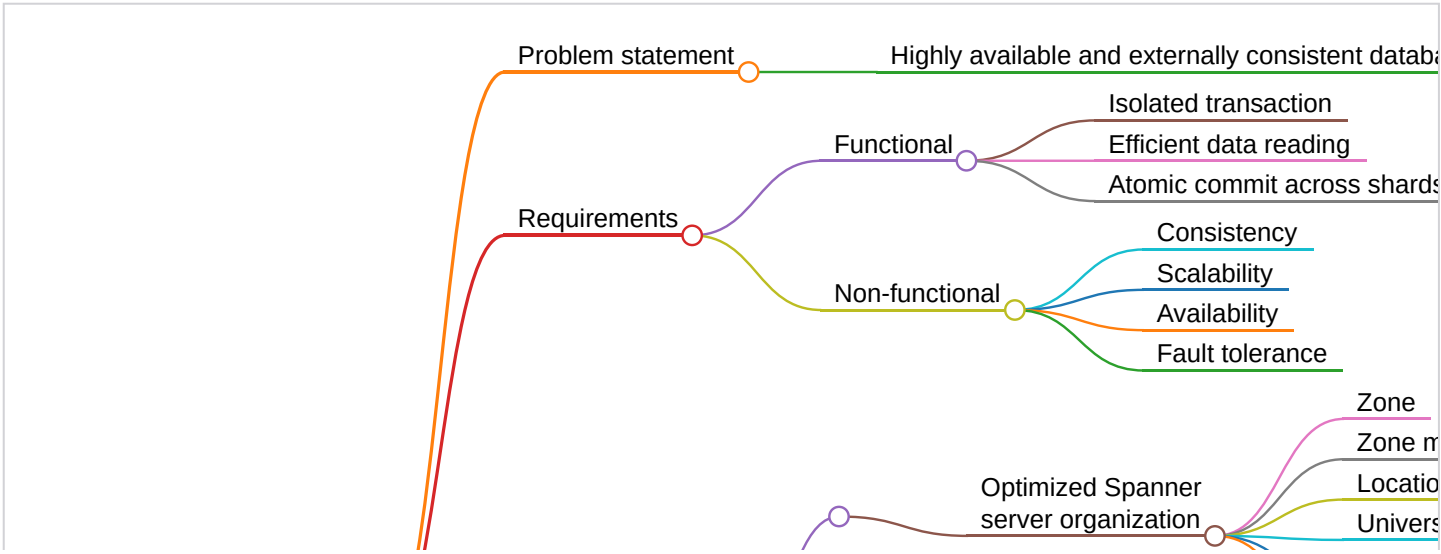
The following illustration shows Spanner's high-level design. The client requests Spanner. The request is routed by the load balancer to one of the servers. The spanservers are divided over zones. The spanserver that receives the request processes it, saves the data to Colossus, and returns a response to the client.



Spanner's high-level design

Bird's eye view

In the next lesson, we will discuss our strongly consistent database in detail. The following concept map is a quick summary of that.


Problem statement
Highly available and externally consistent databa
Requirements
Functional
Isolated transaction
Efficient data reading
Atomic commit across shards
Non-functional
Consistency
Scalability
Availability
Fault tolerance
Optimized Spanner
server organization
Zone
Zone m
Locatio
Univers