Distributed Database Models

Reading

- NoSQL Distilled[1] chapter 4

Reading questions

- What is **sharding**?
- What are the factors to consider when deciding how to assign the data onto different shards?
- What are some possible ways to attempt to balance the load when handling multiple shards?
- Does sharding benefit performance or reliability?
- How does master-slave replication work? Where can reads/writes happen?
- How does peer-to-peer replication work? Where can reads/writes happen?
- What use-cases is master-slave replication suitable for?
- What use-cases is peer-to-peer replication suitable for?
- What are main differences between master-slave replication and peer-to-peer replication?
- How can peer-to-per replication avoid inconsistent writes?

Notes

We discuss here the various possibilities when it comes to expanding our database capabilities by using multiple servers.

Replication vs Sharding

First of all there are two *orthogonal* approaches to server expansion. We can have both in place, or just one, or neither.

Sharding Sharded database servers each contain a part of the overall data, i.e. they store different data on separate nodes. This division could be either via storing different tables, or more likely storing different rows for each table. For example in our evaluations database we may have a different shard for each academic department or each year. Sharding works great when the different queries performed each only need to talk to one shard. But it breaks down when for instance joins need to be performed across multiple shards.

Sharding works well with an aggregate-oriented approach, where different aggregates can be stored on different shards.

[1] http://learning.acm.org/books/book_detail.cfm?id=2381014&type=safari

Replication Replicated servers contain identical copies of the entire database. They are in that sense like "mirrors". Having multiple identical copies means that multiple queries can be served at the same time, but it also means that some amount synchronization needs to be in place to keep the different replicas in sync.

Replication improves *resilience*, as the data is now stored on multiple nodes.

Master-Slave Replication

In a master-slave replication setup we have one server, considered the **master**. All writes happen on that server. There are also multiple **slave** servers, that receive updates from the master. Reads can happen from either the master or the slaves, so this system distributes reads well across multiple nodes. It is however still constrained by having writes happen in only one place.

This design offers **read resilience**. Even if one or more of the servers fails, the remaining servers can keep offering read access. This can help a lot with read-heavy applications, but will offer little benefit to write-intensive applications.

As the slaves are exact replicas of the master server, one of them can assume the role of the master in case the master fails. In fact most of the time you can simply create a set of nodes and have them automatically decide who would be the master.

There are some consistency issues that occur due to the delay in updating between master and slaves. We will discuss those later.

Peer-to-peer Replication

Peer-to-peer replication aims to address the fact that the master can be a severe bottleneck on master-slave replication setups. In a peer-to-peer replication setup the various nodes are all "equals". Any node can accept reads as well as writes, and they communicate these writes to each other.

The biggest advantage of this setup of course is its read and write resilience. One node's failure does not cause problems, as the remaining nodes can continue their work without losing a beat.

The biggest problem that arises is that of consistency. For example we may have conflicting write requests that come on different nodes, and then those nodes attempt to communicate those requests to the rest of the nodes. This could lead to considerable inconsistencies.

There are various ways to resolve this, that we will discuss in more details later. The most standard approach would be to have the replicas communicate their writes first before they "accept" them. Once a majority of the replicas has confirmed a write, it can now be considered as having been successfully performed and a response sent to the client. This requires a certain amount of network traffic in coordinating these writes.

This is a common tradeoff between consistency and availability, and we will return to it later.