Consistency

Reading

- NoSQL Distilled[1] chapters 5, 6

Reading questions

- What is a write-write conflict? Provide various examples of such conflicts. Include situations where the precise way the conflict is resolved leads to incorrect results.
- What are the pessimistic and optimistic approaches to conflict resolution?
- What is logical consistency in read-write conflicts, and how can we handle it?
- What is replication consistency in read-write conflicts, and how can we handle it?
- What are **sticky sessions**? What other term we use in this context? What problem do these solve?
- What does the CAP theorem say? What is its importance?
- Design examples of particular situations where inconsistent writes can be tolerated by the system. Some examples are discussed in section 5.3.1.
- Describe situations where you may be willing to sacrifice durability/persistence, and the benefits of doing so. (section 5.4)
- Describe some possible options in a quorum system with $N = 5$ replicas, and some of the tradeoffs in the various cases.
- What is the point and use cases for **version stamps**? (section 6.2)
- Describe some systems that can be used as version stamps, and their tradeoffs.
- What are **vector stamps**? (section 6.2)

Notes

Write-Write Conflicts

Consistency is of primary importance on all database systems. It aims at handling conflicts in a meaningful way. The most typical example of such conflicts are the so-called write-write conflicts:

> A **write-write conflict** occurs when two clients, A and B, both attempt to update the same resource, with different values. These two attempts must be serialized, with the result that attempt A will get overwritten by attempt B.
>
> In effect what happens is that attempt B was based on the values present before attempt A, and yet happens after it. This is an inconsistency.

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

There are two main approaches to dealing with consistency problems:

Pessimistic The pessimistic approach tries to prevent conflicts at all costs. In the case of write-write conflicts it would do so by establishing a *write lock* when the first request comes in. This way client B cannot attempt to do anything until client A has finished with their write and released the lock.

Pessimistic approaches tend to sacrifice availability/liveness for consistency/safety, as working with the locks takes time and slows down the client's response.

Optimistic The optimistic approach allows conflicts to occur, but detects them and takes some later action to sort them out. A common approach for that is a *conditional update*, where each client tests the value before attempting to update, and fails if the value has changed. In the basic example above, this would cause the B client's update to fail.

Another optimistic approach is to let both writes "occur", but record that they are in conflict, and have in place some rules for resolving such conflicts. This requires specific domain knowledge.

Optimistic approaches sacrifice consistency/safety for availability/liveness.

In cases with replication further complications might ensue, regarding the order of serialization: If two nodes both receive the requests from clients A and B, but perform them in opposite orders, then we end up with one node where A's request went through and another node where B's request went through, and these nodes must now reconsile themselves.

Read-Write Conflicts

Read-Write conflicts come in two flavors. The first is a conflict that violates what we would call **logical consistency**:

> Client A needs to update two resources, X, and Y, that are related. The client updates resource X, but before he gets a chance to update Y client B reads both X and Y. Now client Y is reading an inconsistent state between the two resources.

This situation is usually handled via **transactions**: Both updates to X and Y are treated as one transaction, and they are both performed before the other client has a chance to do anything.

In NoSQL systems this is still possible as long as these changes are part of the *same* aggregate. You can change multiple parts of the same aggregate in one atomic transaction. But changing multiple aggregates will typically require multiple transactions, resulting in an *inconsistency window* during which another client might read inconsistent data.

The other kind of conflict is related to **replication consistency**:

> Client A updates a resource X in one node, and that update propagates through the replicas, but it arrives at the different replicas at different times. Clients accessing the different replicas might see different values for X, even if they access it at the same time, because the update might have arrived at the one replica but not the other.

This is a situation where the system would be **eventually consistent**, as the replicas will eventually all receive the update. But there is a time window during which the data on some replicas is effectively *stale*.

Some times we would be content with a weaker sort of consistency, namely **session consistency**. If we make some changes, we expect that we should be immediately able to see those changes, even if it may take longer for these changes to propagate to everyone else. This is also called **read-your-writes** consistency. This is often achieved via **sticky sessions**, i.e. all operations related to a particular user session being processed by the same node. This is also called **session affinity**.

Relaxing Consistency: The CAP Theorem

The CAP theorem relates 3 notions: **Consistency**, **Availability** and **Partition Tolerance**.

In this context, *availability* means that “every request received by a nonfailing node must result in a response.

Partition tolerance means that the cluster can survive communications breakages that partitions the cluster into sets of servers that can talk to each other but to no servers from other sets. For instance a break in satellite communications could leave a world-wide network into groups of servers on each continent. Each server can connect to any other server on the same continent, but can no longer connect to servers on other continents.

Essentially every distributed system must provide partition tolerance.

In it’s simplified form, the CAP theorem says the following:

> **CAP Theorem**
>
> Given the three properties of Consistency, Availability, and Partition Tolerance, you can only achieve two of the three.
>
> More precisely, on a system that may suffer partitions, as all distributed systems may do, you have to trade off consistency vs availability. It does not have to be one or the other: You often sacrifice some consistency to obtain more availability.

Let us consider a simple example, of two nodes, A and B, linked together. And supposed we attempt to make a write on node A. Then in order to provide consistency, that node must communicate with node B first and confirm that the write is safe before accepting it. But this means that if the link between A and B is broken, then node A cannot possibly allow the write to happen. And this breaks availability.

If the two nodes follow a “master-slave” paradigm, where B is the master and A is the slave, then B can always be available for reads and writes. But in order for A to successfully write it must first send the request to B, which would be impossible if the link between them had broken. In that case also B may show inconsistent results.

On the other end of the spectrum we could have node A accept any writes without consulting with node B. This provides availability. But it can break consistency when conflicting updates occur simultaneously to both nodes.

This may still be OK if the system has some way to handle inconsistent updates. Depending on the system, this may still be OK. For instance, overbooking a flight by a few passengers may be manageable, or having multiple items enter the same shopping cart in Amazon may be resolveable when the user tries to check out.

> Certain inconsistent writes may be manageable based on your system’s logic. You do not always have to sacrifice availability in order to achieve (occasionally unneeded) consistency.

Quorums

Given the possibility of node failures in a replication scenario, one important consideration is how to ensure consistency. For instance if a node is processing a write, but the node fails before it can communicate that write to the other nodes, then that write may not happen. Or when the node manages to connect back to the system, that node’s changes might now conflict with changes elsewhere in the system.

A way to minimize this problem is to use a **write quorum**, where we only consider the write operation performed when over half the nodes have processed it. This way if we have multiple conflicting changes, only one may reach a quorum.

There is a similar question with achieving **read quorum**. How many nodes do we need to contact in order to read something and be sure that we are reading its accurate value? The simple idea is that we need to make sure that we read from enough nodes so that if there was a write going on, we would talk to some of the nodes involved in that write. In order to make that more precise, let’s set some more precise terminology:

N The **replication factor** N is the number of replicas we have. Our system may be broken into many shards and replicas, we only consider here how many copies of a piece of data we have.

W The **write quorum** number is the number of replicas that must accept a write before we consider it done. The bigger W is, the slower writes are.

R The **read quorum** number is the number of replicas we must read a value from before we consider it guaranteed to be the correct value. The bigger R is, the slower reads are.

Based on this we have some key conditions:

- In order to avoid write-write conflicts and ensure strong write consistency, we must have $W > N/2$.

- In order to avoid read-write conflicts and ensure strong read consistency, we must have $W + R > N$.

As an example, let us suppose that $N = 3$, a popular choice. Then we have a number of options for W and R:

- $W = 2$ and $R = 2$. This leads to balanced speeds for reads and writes. Ensures strong consistency.
- $W = 3$ and $R = 1$. This leads to faster reads at the cost of slower writes. Ensures strong consistency. You also cannot tolerate a failing node for writes.
- $W = 2$ and $R = 1$. This ensures write-consistency, fast reads, and somewhat fast writes, but allows for the possibility of stale reads, if the node we read from happens to be different from the two nodes we write to.