What every developer should know about database consistency

July 31, 2020

Imagine assigning some value to a variable, reading it back immediately after, and finding out that somehow the write had no effect at all - madness!

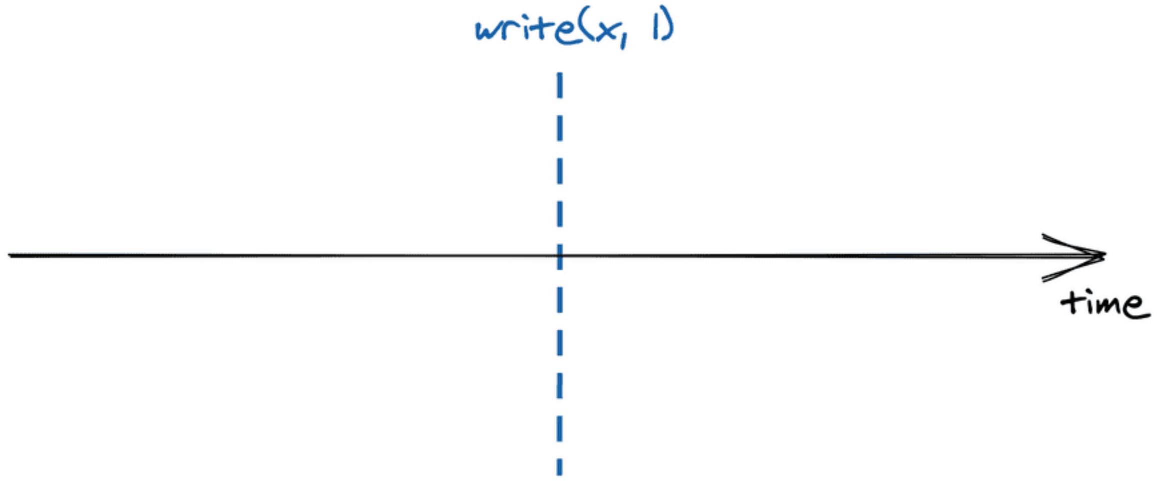
```
x = 42
assert(x == 42)  # throws exception
```

This is precisely what can happen when you are using a distributed data store with weak consistency guarantees. “But wait, aren’t databases supposed to take care of consistency issues for me?” I hear you ask. Whether an update becomes visible sooner rather than later depends on the guarantees offered by the database.

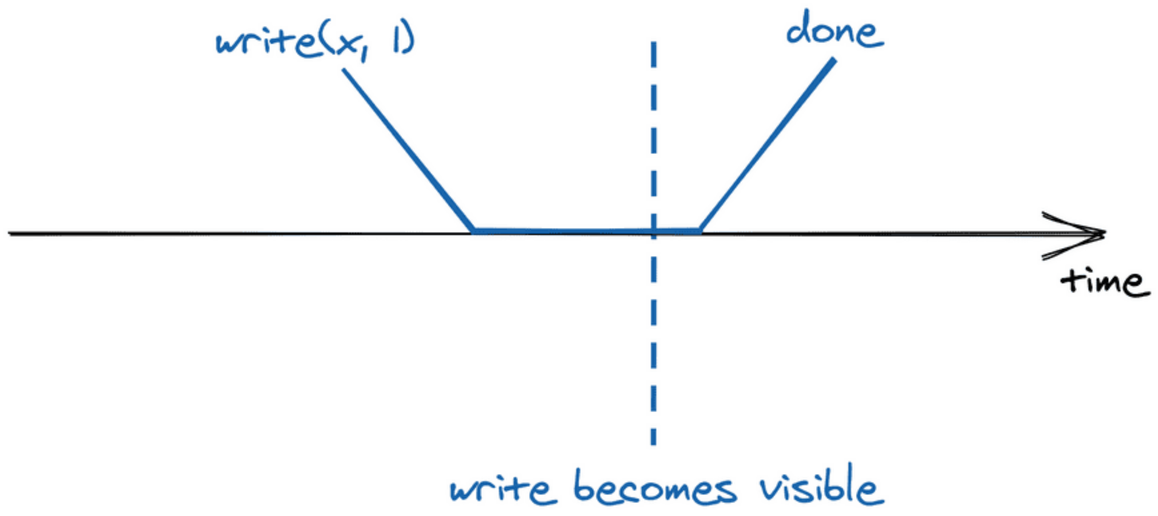
Some databases come with counter-intuitive consistency guarantees to provide high availability and performance. Others have knobs that allow you to chose whether you want better performance or stronger guarantees, like Azure’s Cosmos DB and Cassandra. Because of that, you need to know what the trade-offs are.

Anatomy of a database request

Let’s take a look at what happens when you send a request to a database. In an ideal world, your request executes instantaneously:



But we don't live an ideal world - your request needs to reach the data store, which then needs to process the request and finally send back a response to you. All these actions take time and are not instantaneous:



The best guarantee a database can provide is that the request executes somewhere between its invocation and completion time. You might think that this doesn't look like a big deal - after all, it's what you are used to when writing single-threaded applications - if you assign 1 to x and read its value right after, you expect to find 1 in there, assuming there is no other thread writing to the same variable. But, once you start dealing with data stores that replicate their state on multiple machines for high availability and scalability, all bets are off. To understand why that's the case, we will explore the trade-offs a system designer has to make to implement reads in a simplified model of a distributed database.

Suppose we have a distributed key-value store, which is composed of a set of replicas. The replicas elect a leader among themselves, which is the only node that can accept writes. When the leader receives a write request, it broadcasts it asynchronously to the other replicas. Although all replicas receive the same updates in the same order, they do so at different times.

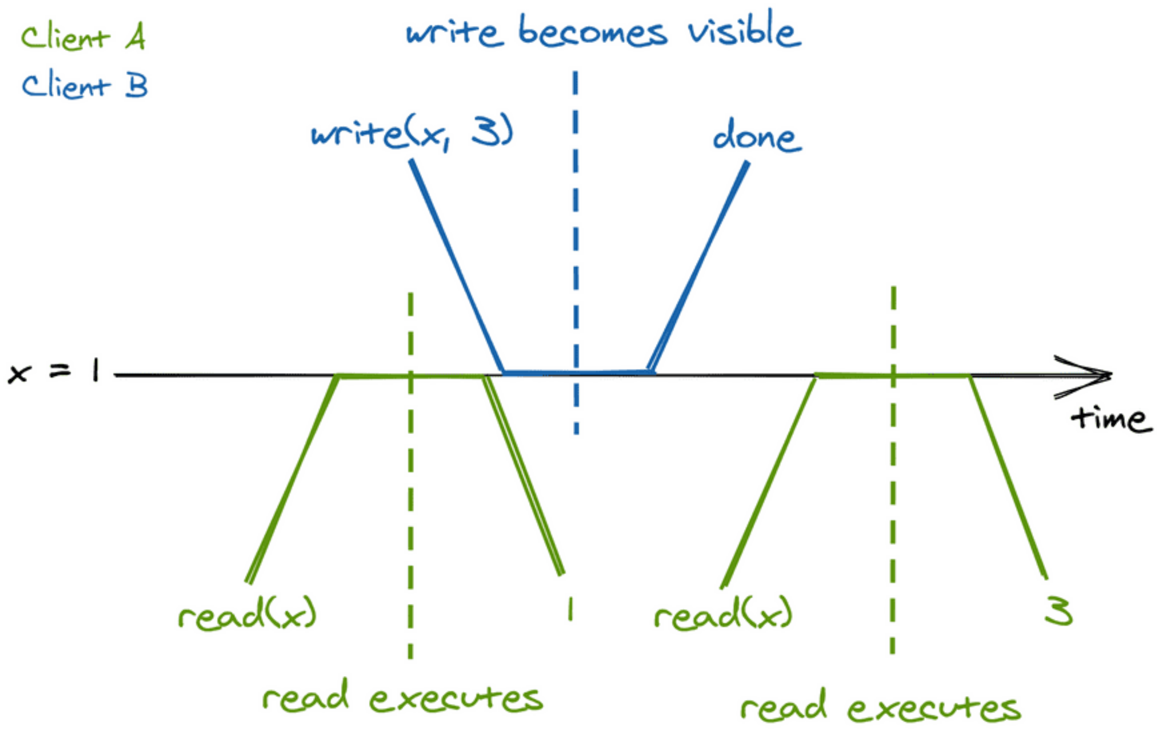
You are tasked to come up with a strategy to handle read requests - how would you go about it? Well, a read can potentially be served by the leader or a replica. If all reads were to go through the leader, the throughput would be limited by what a single node can handle. Alternatively, any replica could serve any read request - that would definitely scale, but then two clients, or observers, could have a different view of the system's state, as replicas can lag behind the leader and between them.

Intuitively, there is a trade-off between how consistent the observers' views of the system are, and the system's performance and availability. To understand this relationship, we need to define precisely what we mean by consistency. We will do so with the help of consistency models, which formally define the possible views of the system's state observers can experience.

Strong Consistency

If clients send writes and reads exclusively to the leader, then every request appears to take place atomically at a very specific point in time as if there was a single copy of the data. No matter how many replicas there are or how far behind they are lagging, as long as the clients always query the leader directly, from their point of view there is a single copy of data.

Because a request is not served instantaneously, and there is a single node serving it, the request executes somewhere between its invocation and completion time. Another way to think about it is that once a request completes, it's side-effects are visible to all observers:



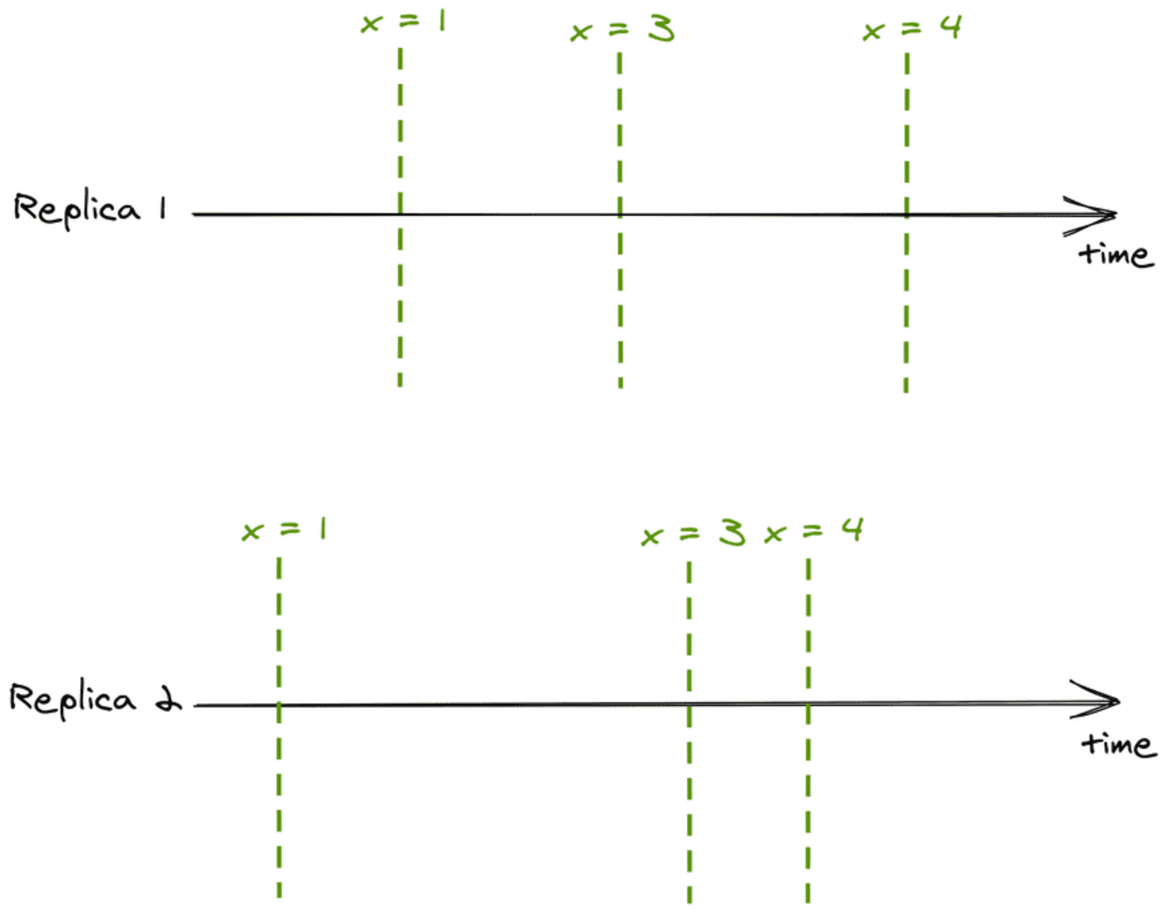
Since a request becomes visible to all other participants between its invocation and completion time, there is a real-time guarantee that must be enforced - this guarantee is formalized by a consistency model called linearizability, or strong consistency. Linearizability is the strongest consistency guarantee a system can provide for single-object requests.

What if the client sends a read request to the leader, but by the time the request gets there, the server that received the request thinks it's still the leader, but it actually was deposed? If the ex-leader was to process the request, the system would no longer be strongly consistent. To guard against this case, the presumed leader first needs to contact a majority of the replicas to confirm whether it still is the leader. Only then is it allowed to execute the request and send back the response to the client. This considerably increases the time required to serve a read.

Sequential Consistency

So far, we have discussed serializing all reads through the leader. But doing so creates a single chokepoint, which limits the system's throughput. On top of that, the leader needs to contact a majority of replicas to handle a read. To increase the read performance, we could allow the replicas to handle requests as well.

Even though a replica can lag behind the leader, it will always receive new updates in the same order as the leader. If a client A only ever queries replica 1, and client B only ever queries replica 2, the two clients will see the state evolving at different times, as replicas are not entirely in sync:



The consistency model in which operations occur in the same order for all observers, but doesn't provide any real-time guarantee about when an operation's side-effect becomes visible to the observers, is called sequential consistency. The lack of real-time guarantees is what differentiates sequential consistency with linearizability.

A simple application of this model is a producer/consumer system synchronized with a queue - a producer node writes items to the queue, which a consumer reads. The producer and the consumer see the items in the same order, but the consumer lags behind the producer.

Eventual Consistency

Although we managed to increase the read throughput, we had to pin clients to replicas - what if a replica goes down? We could increase the availability of the store by allowing a client to query any replica. But, this comes at a steep price in terms of consistency. Say there are two replicas 1 and 2, where replica 2 lags behind replica 1. If a client queries replica 1 and right after replica 2, it will see a state from the past, which can be very confusing. The only guarantee a client has is that eventually, all replicas will converge to the final state if the writes to the system stop. This consistency model is called eventual consistency.

It's challenging to build applications on top of an eventually consistent data store because the behavior is different from the one you are used to when writing single-threaded applications. Subtle bugs can creep up that are hard to debug and to reproduce. Yet, in eventual consistency's defense, not all applications require linearizability. You need to make the conscious choice whether the guarantees offered by your data store, or lack thereof, satisfy your application's requirements. An eventually consistent store is perfectly fine if you want to keep track of the number of users visiting your website, as it doesn't really matter if a read returns a number that is slightly out of date. But for a payment processor, you definitely want strong consistency.

PACELC Theorem

There are more consistency models than the ones presented in this post. Still, the main intuition behind them is the same: the stronger the consistency guarantees are, the higher the latency of individual operations is, and the less available the store becomes when failures happen. This relationship is formalized by the PACELC theorem. It states that in case of network partitioning (P) in a distributed computer system, one has to choose between availability (A) and consistency (C), but else (E), even when the system is running normally in the absence of partitions, one has to choose between latency (L) and consistency (C).

Written by Roberto Vitillo

← The second chapter of Understanding Distributed Systems is out

How to conduct a system design interview →