The Algorists

https://www.thealgorists.com

Replication Lag: A Problem Faced in Eventual Consistency and Asynchronous Replication, and Some Work-Arounds

DECEMBER 26, 2017JULY 20, 2020 / EFFICIENTCODEBLOG

For more such interesting technical contents, please feel free to visit *The Algorists! (https://www.thealgorists.com/Algo)*

While building a scalable distributed system which uses eventual consistency and asynchronous replication, a lot of problems arise due to the replication lag (the fact that all the followers may not be totally synchronized and up-to-date with all the writes happened on the leader), some of which are discussed below along with some possible solutions:

- **Reading Your Own Writes / *Read-after-write* Consistency / *Read-your-writes* Consistency:** Many applications let the user submit some data and then view what they have submitted. For an example, **it might be a comment on a discussion thread**. When new data is submitted, it must be sent to the leader, but when the user views the data, it can be read from a follower.With asynchronous replication, there is a problem: if the user views the data shortly after making a write, the new data may not yet have reached the replica. To the user, it looks as though **the data they submitted was lost,** so they would be understandably unhappy.**Reading Your Own Writes** or ***Read-after-write* Consistency** or ***Read-your-writes* Consistency** comes to aid in such situations. This is a guarantee that is the user reloads the page, they will always see any updates they submitted **themselves. It makes no promise about other users:** other users' updates may not be visible until some later time. However, it reassures the user that their own input has been saved correctly.*Read-your-own-write* can be implemented in several ways as discussed below:1. **Always Read A User's Own Modifiable Data From Leader:**
  When reading something that the user may have modified, read it from the leader; otherwise, read it from a follower. This requires that you have some way of knowing whether something might have been modified, **without actually querying it**. For example:

  user profile information on a social network is normally only editable by the owner of the profile. Thus, a simple rule could be: **always read the user's own profile from the leader**.

2. **Read only from Leader for 1 minute following an update:**

If most things in the application is editable by the user, then the above approach won't be effective, as most things would have to be read from the leader, negating the benefit of ***read scaling.*** In that case, we need some other criteria to decide whether to read from the leader. For example, **you could track the time of the last update and, for one minute after the last update, make all the reads from the leader.** You could also monitor the ***replication lag*** on followers and prevent queries on any followers that is more than one minute behind the leader.

3. **While reading from a follower node, make sure the follower has all the updates at least till the timestamp of the last update made by that user:**

The client can remember the timestamp of its most recent write, then the system can ensure that the replica serving any reads for that user reflects updates at least until that timestamp. **If a replica is not sufficiently up-to-date, either the read can be handled by another replica or the query can wait until the replica has caught up.** The timestamp could be *logical timestamp* (like *log sequence number*) or the actual system clock (in which case we need to keep in mind the problems with *clock skew*).

We also need to keep in mind that in case we have multi-datacenter operation, we would need to have additional complexities along with that discussed above. Any request that needs to be served by the leader must be routed to the datacenter that contains the leader. **In case the users access the system from different kinds of devices, say PC, mobile phone, tablets, and if your approach requires reading from the same leader, then you may first need to route requests from all of a user's devices to the same datacenter**.

Timestamp (either logical or physical) needs to be centralized:

Approach that requires remembering the timestamp of the user's last update become more difficult because the code running on one device doesn't know what updates have taken place from other devices. This metadata will need to be **centralized.**

- **Monotonic Reads (Each User Reads From A Fixed Replica):**

Monotonic Reads address the anomaly that a user can see things ***moving backward in time*** when reading from asynchronous followers.
This can happen if a user makes several reads from different replicas.

An interesting example:

Say in a discussion thread, user 1234 adds a comment. Now lets suppose another user 5678 makes the same query to fetch the comments twice, first to a follower with little lag, then to a follower with greater lag. This scenario is quite likely if the user 5678 refreshes a web page and each request is routed to a random server. The first query returns the comment recently added by the user 1234, but the second one does not return that because the lagging follower has not yet picked up that write. In effect, the second query is **observing the system at an earlier point in time** than the first query. No doubt that it's very confusing for the user 5678 if he first sees user 1234's comment and then see it disappear.

Monotonic Reads is a guarantee that this kind of anomaly doe not happen. ***Monotonic Read*** **is a lesser guarantee than** ***Strong Consistency,*** **but stronger guarantee than** ***Eventual Consistency.*** When you read data, you may see an old value; **monotonic reads only means that if one user makes several reads in sequence, they will not see time go backwards, i.e., they will not read older data after having read a newer value.**

One way of achieving monotonic reads is to make sure that each user always makes their reads from the same replica. The replica can be chosen based on a hash of their user ID. However if that replica fails then the user's queries will need to be rerouted to another replica.

- **Consistent Prefix Reads (Only happens in Sharded/Partitioned Distributed Databases) :**

One Proposed Solution:
Send any writes that are causally related to each other to the same shard or partition.

Sometime we may have situation concerning ***violation of causality.*** If two data are causally dependent on each other then the order in the data was written is very important, without which the data would make no sense.
For example, see the below conversation:

```
Anthony: Hi Michael, did you go visit Victoria yesterday?
Michael: Yes I did visit her yesterday.
```

The order in which someone receives the above conversation is very important. If a user receives the above conversation in a different ordering such as below then it would make no sense:

```
Michael: Yes I did visit her yesterday.
Anthony: Hi Michael, did you go visit Victoria yesterday?
```

To the observer it looks like Michael is answering the question before Anthony has even asked it.

Preventing this kind of anomaly requires a special type of guarantee: ***Consistent Prefix Reads.***

> ***Consistent Prefix Reads*** **says that if a sequence of writes happens in a certain order, thenanyone reading those writes will see them appear in the same order.**

This is a particular problem in ***partitioned (sharded) databases.*** If the database always applies writes in the same order, reads always see a consistent prefix, so this anomaly cannot happen. However, in many distributed databases, different partitions operate independently, so there is no global ordering of writes: when a user reads from the database, they may see some parts of the database in an older state and some in a newer state.

> One solution is to **make sure that any writes that are causally related to each other are written to the same partition.** But in practice, in some applications that cannot be done effectively.

There are also algorithms like "happens-before" that explicitly keep track of causal dependencies.

For more such interesting technical contents, please feel free to visit *The Algorists! (https://www.thealgorists.com/Algo)*

Categories: Computer Science, Scalable Systems