Contents

1 The Architecture **2**
1.1 Layered Structure 2
1.2 Directories and Data Model 3
1.3 Considerations for Two-Phase Commit 3

2 Concurrency Control **3**
2.1 Paxos Group States 4
2.2 Committing Read-Write Transactions 5
2.3 Read-Only Transactions 6

Spanner Transaction and Replication

Junlong Gao

The Google's paper on building a "globally consistent" database is a dense one encapsulating multiple interesting ideas. It uses locking and two phase commit protocol to ensure serializability but the most important, and significant contribution of this paper is how a assumed "absolute time ordering" can be used to achieve linearizability [1].

This document tries to highlight the ideas described in the paper, with a focus on how the entire system achieves the external consistency guarantee: serializability with linearizability .

1 The Architecture

1.1 Layered Structure

Logically speaking, Spanner has a very clear layered structure:

- The lowest layer is the persistent storage. Important design decision revolves around if it is supposed to be a shared distributed storage or not. Google uses Colossus (the successor of Google File System), a distributed file system.
- Then is the tablet, a local key-value store. One important aspect is the keys are versioned:
$$(\mathsf{key} : \mathsf{string}, \mathsf{version} : \mathsf{timestamp}) \rightarrow \mathsf{val} : \mathsf{string}$$
- Then is the replication layer. Spanner uses Paxos to replicate and coordinate access of the key-value pairs in the tablets. The nodes responsible for replicating a particular set of key-value pairs is called a Paxos group, or a group, for short.
- Then is the transaction layer. Not every node is responsible for the transaction implementation at all times. Only the leader in a replication group needs to implement transaction support for the data in that group. This implies maintaining the lock table, participate in a transaction two-phase commit, recording prepare and commit messages.
- Then is the distribution layer. In Spanner, nodes are placed in different zones and there is a service *zonemaster* for tracking nodes in each zone. Another *placement driver* is responsible for tracking and balancing across zones.

- Finally, the entire deployment, including all the zones and the placement driver is named a single *universe*. A *universemaster* service is responsible for providing monitoring and interactive management for the entire deployment.

1.2 Directories and Data Model

A *Directory* is a range of key-value pairs, sharing some prefix. This is the smallest unit for replication, data re-balancing, and have application level configurable replication factor. One Paxos group can replicate multiple directories (since one node is one replica, note in the paper Google gave up implementing multiple Paxos state machine in the same node) so each replica's tablet holds multiple directories. Since there are many server nodes in a typical deployment, the placement driver and the zonemasters will be able to group server nodes to form Paxos groups of size 3, 5 or more depending on application configurations.

1.3 Considerations for Two-Phase Commit

The two-phase commit has been regarded as "anti-availability" due to its various shortcomings in terms of availability requirements for both participants, coordinator, and even the clients [4]. Some storage service tries to code around it and provide a weaker isolation level (if it supports some form of transactions) and weaker consistency level for the sake of better availability [2].

Google's work on Spanner shows a promising result: when building two-phase commit on a much higher level of availability unit: a Paxos group, two-phase commit not only becomes usable, but also allows us to harness the good old transaction processing studies on isolation: two-phase locking and well-ordered locking enables serializability [3].

2 Concurrency Control

The techniques for ensuring serializability is well-discussed in [3]. It is the problem of linearizability and how it is solved makes Spanner so interesting. linearizability requires system reads and writes respects "real-time" ordering, in that the reads should reflect the writes happened before them, and the writes should be applied in the order of their commit timestamps.

At a high level, Spanner commit protocol still is a variant of two-phase commit. To ensure external consistency, Spanner timestamps every mutation (writes) of the database, and make sure that these mutations are available for subsequent reads/writes only after the absolute real-time has passed those mutations' commit timestamps.

To serve reads, a read-only transaction T will carry a timestamp $\mathsf{t}_{\mathsf{read}}$ for locating the point in time of the reading, so that Spanner can ensure the data is no earlier than this timestamp. When $\mathsf{t}_{\mathsf{read}} \geq \mathsf{t}_{\mathsf{abs}}(\mathsf{T}_{\mathsf{commit}})$, the reads will always reflect the latest writes.

Roughly speaking, the modification of the two-phase commit done in Spanner is for negotiating a lower bound for the timestamps of each committing transaction. The

negotiated timestamp is then used by each replica to keep track if it is safe to serve a read when there are in-flight transactions. Also, after two-phase commit is completed, the latest committed mutation will be used to keep track if this replica has up-to-date states for reading.

2.1 Paxos Group States

The smallest unit for participating two-phase commit is a Paxos group. Each group has a leader, and it also maintains the lock table and works as the participating transaction manager for the group's other replicas.

The Paxos group maintains a replicated log, which is used to persisting commit or prepares records. When a transaction is committed, sometimes later, the replicas of the group is allowed to tail this log and apply the commit message onto its local persisted state machine. In the case of Spanner, they use a B^+-tree as both the index structure and the data storage for this state machine.

Each replica in the group will have the following internal states in addition to the Paxos log and the state machine:

- t_{safe}^{TM} This timestamp indicates a strict lower-bound for all in-flight transactions' commit timestamp. The meaning of this timestamp is, if there is a read translation at time s_{read} and $s_{read} \leq t_{safe}^{TM}$, then the in-flight transactions cannot interfere this read as the mutation will happen in a future timestamp with respect to s_{read}. The transaction protocol entails how this state is maintained for each committed transition.

- t_{safe}^{Paxos} This timestamp indicates a lower-bound for the next future transaction's commit timestamp. In other words, the next mutation's commit point cannot be earlier than t_{safe}^{Paxos}. The meaning of this timestamp is, if there is a read translation at time s_{read} and $s_{read} \leq t_{safe}^{Paxos}$, then the replica can safely serve a read by reading the persisted state machine, since the replica knows the next mutation will happen in a future time with respect to s_{read}.

- t_{latest} This timestamp indicates the latest committed record in the Paxos log. One may argue that this is redundant since we can trivially have $t_{safe}^{Paxos} = t_{latest}$. In fact, Spanner makes it $t_{safe}^{Paxos} \geq t_{latest}$, for the reasons to be entailed later.

- n_{latest} This is a sequence number representing the latest commit record in the Paxos log.

- $\mathsf{MinNextTS}(n)$ This maps a sequence number to a timestamp. $\mathsf{MinNextTS}(n)$ gives a lower-bound for the committing timestamp this replica will propose for commit record $n + 1$. This is used to compute t_{safe}^{Paxos}.

2.2 Committing Read-Write Transactions

the transaction involves both read and write are declared by the client and marked as rw transactions. The rough transaction T_i's committing process is like this:

- The client identifies all the groups involved in T_i, and pick one group g_c as the coordinator, then the rest of the groups g_j become the participants for this two-phase commit. From now on, each group is identified by its Paxos leader node. i.e. g is the leader node for that group, not all the replicas in the group.

- The client sends the commit message to g_c, and all buffered writes to each corresponding group. Since different groups are usually in different zones connected by WAN, let the client drive the commit process avoids sending these buffered data twice over the WAN.

- The non-coordinator groups g_j will then start preparing for T_i. It first acquires the locks in the lock table, then proposes a committing timestamp to the coordinator. This timestamp is called "prepare timestamp" $s_{i,g}^{prepare}$ and will be logged as a prepare message in its Paxos log. $s_{i,g}^{prepare}$ has to satisfy:

$$s_{i,g}^{prepare} \geq t_{safe,g}^{Paxos} \tag{1}$$

  So that when the transaction commits, $t_{safe,g}^{Paxos}$ serves its purpose.

- The coordinating group g_c will also acquires locks in the group for T_i. After it receives all the prepare messages, it will pick a "committing timestamp" s_i such that

$$\begin{aligned} \forall g, s_i &\geq s_{i,g}^{prepare} \\ s_i &\geq TT.now().latest \\ s_i &\geq t_{latest} \end{aligned} \tag{2}$$

  Then the transaction is now committed, the coordinator writes a commit message to its own Paxos log.

- Here is the real trick: after coordinator declares commit for T_i, it does not immediately make it visible to the rest of the participant group *nor should it allow for the locks to be released*. Instead, it simply waits. While the rest of the participants are waiting for the coordinator to declare commit, the coordinator just wait until $TT.after(s_i)$ is true. It is only after this is true, the coordinator can broadcast the commit message to the rest of the participants, and they can apply this commit message to their Paxos log, and subsequently, their state machines, to make this commit visible and release the locks. This imposes the real-time ordering for subsequent read or write transactions. To wit, consider another transaction T_j issued by the client such that:

$$t_{abs}(T_j^{start}) \geq t_{abs}(T_i^{commit})$$

and, by definition, the real time commit point $t_{abs}(T_i^{commit})$ is when T_i is visible, which satisfies:

$$s_i \leq t_{abs}(T_i^{commit})$$

Thanks to the commit wait, and we can thus conclude:

$$s_j \geq t_{abs}(T_j^{start}) \geq t_{abs}(T_i^{commit}) \geq s_i$$

In other words, the internal ordering of T_i and T_j mutating the state machine respects the real time ordering.

2.3 Read-Only Transactions

To serve read, read transaction will carry a timestamp s_{read} to specify how up-to-date the read should be served. Setting $s_{read} = TT.now.latest$ will ensure linearizability . It is now a matter of how each replica keeps track of how up-to-date it is when serving the read at s_{read}. There are nature problem arises:

- Each replica for the same group will apply commit record at different speed to the persistent layer, and thus is not up-to-date with respect to each other within the group.
- When a read-only transaction expands multiple groups, only all the groups are up-to-date can they serve the read at s_{read}.
- When each replica is handling some in-flight transaction (i.e. a prepare message is present), the replica needs to know if this transaction, after it is committed, can interfere with the read at s_{read}. If the in-flight transaction has a timestamp $s \leq s_{read}$, then this replica cannot serve read until the transaction commits.
- Finally, if the replica knows its last committed transaction timestamp $t_{latest} \geq s_{read}$, then its state is up-to-date, but if $t_{latest} < s_{read}$, can it serve the read? Once the system stops writing and only see read requests, eventually $t_{latest} < s_{read}$. How can the replica tell if it is OK to serve read?

To answer these questions, Spanner uses two states t_{safe}^{TM} and t_{safe}^{Paxos} to help determine if read at s_{read} can be safely served or not. The high-level idea is to track committed and in-flight transaction timestamps, and derive a lower-bound timestamp for both of them. If the read timestamp is no later than the lower-bound, then the read is safe to serve.

t_{safe}^{TM} is derived in the following manner: the replica tracks all the prepare transactions T_i it current sees without commit. If there is no such T_i, then $t_{safe}^{TM} = \infty$. Otherwise, recall that each replica will see the prepare message in the Paxos group. For each such message uncommitted, $t_{g,i}^{prepare} \leq t_{g,i}^{commit}$ guaranteed by the read-write transaction commit protocol. Then it is suffice to set $t_{safe}^{TM} = \min_i\{t_{g,i}^{prepare}\} - 1$.

t_{safe}^{Paxos} is considerably trickier. It is tempting for each replica to set $t_{safe}^{Paxos} = t_{latest}$, since timestamp marches forward trivially. But this cannot work since when there is no

more write transaction, future reads cannot be served. Even for a read that happens before some latest write, the read can span to multiple groups and some groups did not participate that write transaction. A more practical way to look at t_{safe}^{Paxos} is to regard it as the lower-bound for the next write operation (at sequence number $n+1$ provided the current latest applied commit record is n). As long as s_{read} is before the future write timestamp, no mutation can happen between s_{read} and the future write. Thus the read can be served. Recall that each group honors its current t_{safe}^{Paxos} when proposing a commit timestamp for the prepare message, so the next write timestamp can be enforced. The leader simply keeps a mapping $MinNextTS(n)$ that maps n to the next write $n+1$'s timestamp lower-bound, and uses it to advance t_{safe}^{Paxos} in the group. Such a mapping is done by extending $MinNextTS(n)$ every 8 seconds. i.e. The next proposed commit timestamp will be at most 8 seconds ahead from $TT.now.latest$. During this 8 second period, the replica can safely serve reads with $s_{read} = TT.now.latest$.

But how long should we wait? Does that mean the write transaction needs to wait for 8 seconds in the worst case by the coordinator? A leader should not use $MinNextTS(n)$ blindly, since this enforces a lower-bound for commit timestamp (and the coordinator needs to wait for that to be passed). Only when a read occurred and t_{safe}^{Paxos} is out of sync with respect to current $TT.now$ too far, the leader can use $MinNextTS(n)$ to advance t_{safe}^{Paxos} to serve the read (but the writes in this timeframe are poisoned by at most 8 seconds).

Also, non-conflict reads can be arbitrarily delayed by the (lack of) advancement of the above two states. For example, prepared yet not committed transition timestamps gives a false conflict if their timestamps are too early. Spanner's answer is to use a more fine-grained table for the key ranges' t_{safe}^{TM} and t_{safe}^{Paxos}. So that when the read transaction comes, non-conflict ranges' timestamps do not contribute to the comparison, only the overlapping ranges' matter matters.

References

[1] Brian F. Cooper. Spanner. *Proceedings of the 6th International Systems and Storage Conference on - SYSTOR '13*, 2013.

[2] Seth Gilbert and Nancy Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *Acm Sigact News*, 33(2):51–59, 2002.

[3] Jim Gray and Andreas Reuter. *Transaction processing: concepts and techniques*. Elsevier, 1992.

[4] Pat Helland. Life beyond distributed transactions: an apostate's opinion. In *CIDR*, volume 2007, pages 132–141, 2007.