Spanner (Jet.com)

Spanner is the world's best database. It's available on Google Cloud as Cloud Spanner, powers major Google products like Adwords, is semi-implemented open-source as CockroachDB (unrelated to Google), and has been around since 2012.

Why is it so good? Well, it excels at three main things that are pretty hard to do well:

1. External consistency
2. Transparent sharding / load balancing
3. Client-friendly, intelligent SQL semantics

We'll be focusing on the first one and how they're achieved by Spanner. The 2012 white paper details the first two, and the 2017 SQL white paper details the last one. If interest is there, we can deep-dive on the third one too.

Starting with the first bullet - I'll let the white paper speak for itself:

> Spanner has two features that are difficult to implement in a distributed database: it provides externally consistent reads and writes, and globally-consistent reads across the database at a timestamp. These features enable Spanner to support consistent backups, consistent MapReduce executions [12], and atomic schema updates, all at global scale, and even in the presence of ongoing transactions.

Consistency, as we know from Designing Data-Driven Applications, is HARD. How does Spanner achieve this? In short - TrueTime! But let's review the software stack first.

Implementation

Spanner's organizational structure:

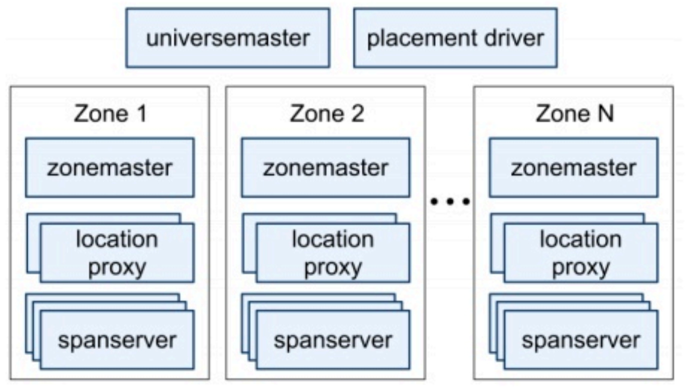


Figure 1: Spanner server organization.

- One Spanner deploy is a `universe`.
- `Zones` are an administrative bundle, and can be deployed across datacenters
- A zone's `zonemaster` handles data placement on spanservers, which each zone has many of.
- A `spanserver` actually serves data to clients.
- A zone's `location proxy` handles routing of data requests to a particular spanserver that owns that data.

Spanservers in Detail

Each spanserver owns 100 - 1000 instances of a `tablet` data structure (like BigTable) — each tablet is simply a bag of:

> (key: String, timestamp: int64) -> string

All tablet data is stored via B-Tree-like structures on disk, as well as a write-ahead log; these both sit on a DFS called Colossus (successor to GFS).

Here's where Paxos state machines enter — **we use them to ensure consistent replication between tablets!** Each tablet has a Paxos state machine with long-lived leaders and time-based leader leases (10 seconds).

This is a lot of nomenclature with a lot of theory behind it — diagram is easier:

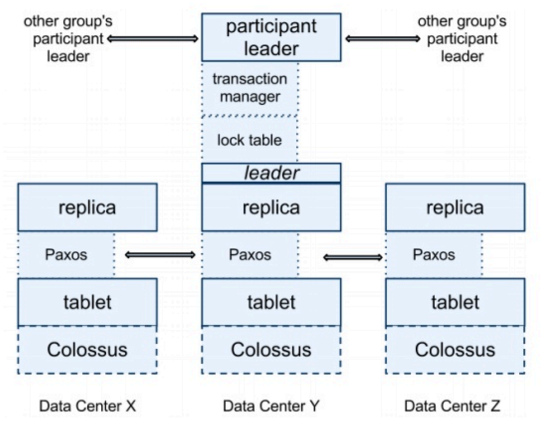


Figure 2: Spanserver software stack.

Other Bits

Directories are groups of keys within a tablet. They can be moved around and colocated based on access patterns:

> A directory is the unit of data placement. All data in a directory has the same replication configuration. When data is moved between Paxos groups, it is moved directory by directory, as shown in Figure 3. Spanner might move a directory to shed load from a Paxos group; to put directories that are frequently accessed together into the same group; or to move a directory into a group that is closer to its accessors. Directories can be moved while client operations are ongoing. One could expect that a 50MB directory can be moved in a few seconds.

About transparent replica management under load:

> As a globally-distributed database, Spanner provides
> several interesting features. First, the replication configurations for data can be dynamically controlled at a fine grain by applications. Applications can specify constraints to control which datacenters contain which data, how far data is from its users (to control read latency), how far replicas are from each other (to control write latency), and how many replicas are maintained (to control durability, availability, and read performance). Data can also be dynamically and transparently moved between datacenters by the system to balance resource usage
> across datacenters.

Consistency and Consensus

In a concurrent system with events happening independently, any two events `A` and `B` can be ordered in one of the two following ways:

1. **Partially ordered** - either A `happens-before` B, B `happens-before` A, **OR** they happened concurrently.
2. **Totally ordered** - every operation is ordered with respect to others, i.e. there is no concurrency.

A partial order is implied by a total order (is "stronger", and thus contains), and a total order can be created out of a partial order (from DDIA):

> In particular, we can create sequence numbers in a total order that is consistent with causality if we promise

> that if operation A causally happened before B, then A occurs before B in the total order (A has a lower sequence number than B). Concurrent operations may be ordered arbitrarily. Such a total order captures all the causality information, but also imposes more ordering than strictly required by causality.

Kleppmann goes to lengths to differentiate the concept of *linearizability* - the ability to treat your data totally ordered, like there's only one source of data - from the concept of *causality* - treating your data as partially ordered. We won't focus too much on this today because, as we'll see, Spanner is sneaky and CREATES linearizability from causality - derives a total order from a partial order - using TrueTime and Paxos!

Let's take a first stab at creating a totally ordered system of events by using *Lamport timestamps*. Lamport timestamps are simply [id, nodeID] pairs — the `id` could be some monotonic integer, and the `nodeID` can be the same. This can be used to create a total order — (3, 3) > (3, 2) because we say so. BUT! That total order is only available for introspection and decision-making *after* a set of concurrent operations have been finalized. In order to make game-time decisions on concurrent operations, all clients and nodes must agree on when a total order has been finalized, and this requires game-time awareness of what other operations nodes are concurrently performing. In other words, you need to *broadcast* that ordering information!

All this to say - in order to have consistent ordering in a distributed system, you must strengthen *total order* to **total order broadcast**.

Total order broadcast is an attempt to get what single-leader replication offers - a total order of operations - at a fault-tolerant scale in a distributed system. In order for any algorithms to work on that front, we need two guarantees from our nodes:

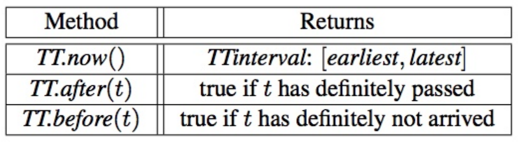
1. Reliable delivery - no messages are lost
2. Totally ordered delivery - messages from a node are delivered in the same order to all receiving nodes

Total order broadcast and linearizability are interchangeable, and both are a form of creating consensus in distributed systems! So to get their power, we need a a consensus algorithm like **two-phase commit**.

Some consensus algorithms: Paxos is the classic one, and one we'll cover here a bit more. Other examples are Raft and Zab (Zab is used by ZooKeeper).

TrueTime

The TrueTime API:



Method	Returns
TT.now()	*TTinterval*: [*earliest*, *latest*]
TT.after(t)	true if t has definitely passed
TT.before(t)	true if t has definitely not arrived

Table 1: TrueTime API. The argument t is of type *TTstamp*.

TrueTime relies on two kinds of clocks: `GPS clocks` and `atomic clocks`. Both have different failure modes (GPS: radio interference, atomic clocks: frequency drift), so both are maintained for redundancy and fault-tolerance.

TrueTime is physically maintained by two kinds of masters, corresponding to these two kinds of clocks:

- GPS masters
- Armageddon masters (atomic clocks)

Armageddon masters advertise out worst-case frequency-based clock drift in between synchronizations. GPS masters don't advertise any real time drift.

Each Spanner node has a time daemon attached to it that periodically polls these different masters to synchronize. The daemon uses *Marzullo's algorithm* to detect liars, and machines can evict themselves from the cluster if they detect that they themselves are a liar (local clock).

Between synchronizations, daemons will advertise a slowly-increasing clock drift ϵ. This ends up as a sawtooth function of clock drift between 1 - 7 ms between synchronizations, which are every 30 seconds. This is important.

Linearizability via TrueTime

Spanner uses TrueTime and Paxos to enforce external consistency (what Spanner defines as linearizability) on all read-write transactions. Here's how.

Remember that data is actually served from an individual spanserver, which consists of 100 - 1000 tablets, which in turn each have their own Paxos state machine. Each tablet's KV pairs are replicated via Paxos, and the set of replicas of a given tablet is called a *Paxos group*.

Okay - so when a RW transaction comes in, it is routed to the appropriate Paxos group leader that owns that data involved. Behind the scenes, a Paxos group is managing leader election every ten seconds, with leadershop extended by a successful R/W transaction. When a transaction comes in to the leader, that leader kicks off the Paxos protocol within its group. Per normal Paxos - remember that multiple transactions could be occuring at the leader nearly simultaneously, and the Paxos protocol might fail at one or more of the replicas ("acceptors").

The Paxos protocol uses *two-phase locking* to do the actual Paxos write - this means that the timestamp assigned to the transaction occurs after all locks are acquired, and before any locks are released.

Here's the important part - the **Spanner monotonicity invariant**. A single Paxos leader can of course assign monotonically increasing timestamps once it's acquired all locks via the Paxos protocol - central point of coordination! So it does so, and uses TrueTime to basically **create a total order!**

The technical proof is a bit overkill, but in short - if a Paxos write event occurs at time `A`, the timestamp assigned to that event is guaranteed to be **no less than TT.now().latest**. This means that Spanner is essentially spacing out its Paxos timestamp writes based on TrueTime clock drift / uncertainty to **create** external consistency / a total order.

tl;dr, I'd say this - Spanner takes concurrent RW transactions and uses Paxos to decide on a order. The order is reflected in the timestamps, which are always spaced by TrueTime's running calculation of ϵ per time daemon. This allows for external consistency - all replicas will have the same data, with timestamps agreed upon with disjointness and monotonicity to serve and manage all future transactions (as well as snapshot reads!).

This timestamp monotonicity guarantee allows Spanner to serve reads at any replica that is *sufficiently up to date*. Boom, done with RO transactions.

How does one handle transactions that involve multiple Paxos groups / tablets? If a transaction involves only one

Paxos group, it's sufficient to have a tablet and a corresponding lock table; if it involves multiple Paxos groups, then we need a `transaction manager`, which coordinates with the participant leaders (now slaves) of the other Paxos groups.