Spanner

December 5, 2024

1 Introduction

1. 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.
2. Spanner provides externally consistent reads and writes, and globally-consistent reads across the database at a timestamp.

2 Implementation

A Spanner deployment is called a **universe**.

Spanner is organized as a set of **zones**, where each zone is the rough analog of a deployment of Bigtable servers. Zones are the unit of administrative deployment.

- The set of zones is also the set of locations across which data can be replicated.
- Zones can be added to or removed from a running system as new datacenters are brought into service and old ones are turned off, respectively.

- Zones are also the unit of physical isolation: there may be one or more zones in a datacenter, for example, if different applications' data must be partitioned across different sets of servers in the same datacenter.

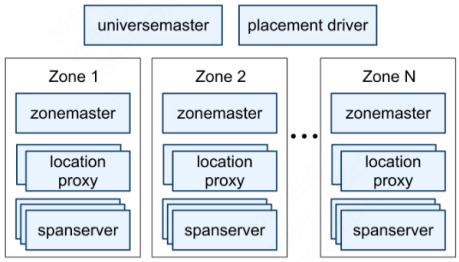


Figure 1: Spanner server organization.

A zone has one **zonemaster** and between one hundred and several thousand **spanservers**. The former assigns data to spanservers; the latter serve data to clients. The per-zone **location proxies** are used by clients to locate the spanservers assigned to serve their data. The **universe master** and the **placement driver** are currently singletons.

- The universe master is primarily a console that displays status information about all the zones for interactive debugging.

- The placement driver handles automated movement of data across zones on the timescale of minutes. The placement driver periodically communicates with the spanservers to find data that needs to be moved, either to meet updated replication constraints or to balance load.

2.1 Spanserver Software Stack

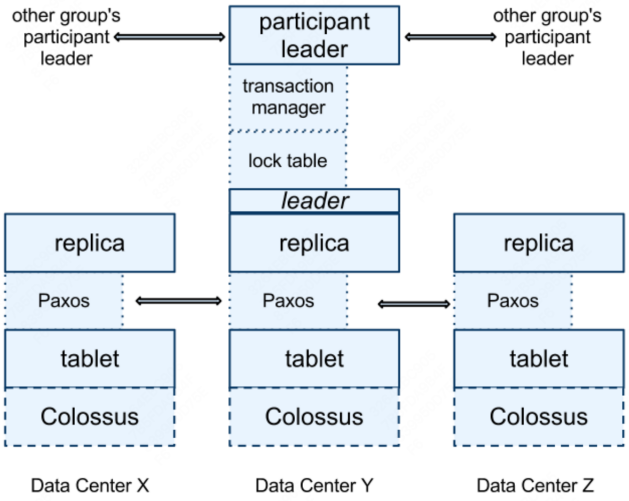


Figure 2: Spanserver software stack.

At the bottom, each spanserver is responsible for between 100 and 1000 instances of a data structure called **tablet**. A tablet implements a bag of the following mappings:

$$(\text{key:string}, \text{timestamp:int64}) \rightarrow \text{string}$$

A tablet's state is stored in set of B-tree-like files and a write-ahead log, all on a distributed filesystem called Colossus.

To support replication, each spanserver implements a single Paxos state machine on top of each tablet. Each state machine stores its metadata and log in its corresponding tablet. Our Paxos implementation supports long-lived leaders with time-based leader leases, whose length defaults to 10 seconds. The current Spanner implementation logs every Paxos write twice: once in the tablet's log, and once in the Paxos log. This choice was made out of expediency, and **we are likely to remedy this eventually**. Our implementation of Paxos is pipelined, so as to improve Spanner's throughput in the presence of WAN latencies; but writes are applied by Paxos in order.

The Paxos state machines are used to implement a consistently replicated bag of mappings. The key-value mapping state of each replica is stored in its corresponding tablet. Writes must initiate the Paxos protocol at the leader; reads access state directly from the underlying tablet at any

replica that is sufficiently up-to-date. The set of replicas is collectively a **Paxos group**.

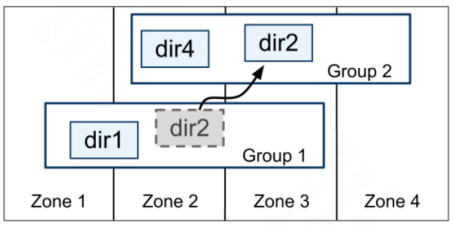
At every replica that is a leader, each spanserver implements a **lock table** to implement concurrency control. The lock table contains the state for two-phase locking: it maps ranges of keys to lock states. (*Note that having a long-lived Paxos leader is critical to efficiently managing the lock table.*) In both Bigtable and Spanner, we designed for long-lived transactions (for example, for report generation, which might take on the order of minutes), which perform poorly under optimistic concurrency control in the presence of conflicts. Operations that require synchronization, such as transactional reads, acquire locks in the lock table; other operations bypass the lock table.

At every replica that is a leader, each spanserver also implements a **transaction manager** to support distributed transactions. The transaction manager is used to implement a **participant leader**; the other replicas in the group will be referred to as **participant slaves**. If a transaction involves only one Paxos group (as is the case for most transactions), it can bypass the transaction manager, since the lock table and Paxos together provide transactionality. If a transaction involves more than one Paxos group, those groups' leaders coordinate to perform two-phase commit. One of the participant groups is chosen as the coordinator: the participant leader of that group will be referred to as the **coordinator leader**, and the slaves of that group as **coordinator slaves**. The state of each transaction manager is stored in the underlying Paxos group (and therefore is replicated).

2.2 Directories and Placement

On top of the bag of key-value mappings, the Spanner implementation supports a bucketing abstraction called a **directory**, which is a set of contiguous keys that share a common prefix. Supporting directories allows applications to control the locality of their data by choosing keys carefully.

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 2.2. 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.



Spanner tablet is a container that may encapsulate multiple partitions of the row space. We made this decision so that it would be possible to colocate multiple directories that are frequently accessed together.

Movedir is the background task used to move directories between Paxos groups. Movedir is also used to add or remove replicas to Paxos groups because Spanner does not yet support in-Paxos configuration changes. Movedir is not implemented as a single transaction, so as to avoid blocking ongoing reads and writes on a bulky data move. Instead, movedir registers the fact that it is starting to move data and moves the data in the background. When it has moved all but a nominal amount of the data, it uses a transaction to atomically move that nominal amount and update the metadata for the two Paxos groups.

A directory is also the smallest unit whose geographic-replication properties (or **placement**, for short) can be specified by an application. The design of our placement-specification language separates responsibilities for managing replication configurations. Administrators control two dimensions: the number and types of replicas, and the geographic placement of those replicas. They create a menu of named options in these two dimensions (e.g., North America, replicated 5 ways with 1 witness). An application controls how data is replicated, by tagging each database and/or individual directories with a combination of those options. For example, an application might store each end-user's data in its own directory, which would enable user A's data to have three replicas in Europe, and user B's data to have five replicas in North America.

Spanner will shard a directory into multiple **fragments** if it grows too large. Fragments may be served from different Paxos groups (and therefore different servers). Movedir actually moves fragments, and not whole directories, between groups.

2.3 Data Model

Spanner exposes the following set of data features to applications: a data model based on schematized semi-relational tables, a query language, and general-purpose transactions.

The application data model is layered on top of the directory-bucketed key-value mappings supported by the implementation. An application creates one or more **databases** in a universe. Each database can contain an unlimited number of schematized **tables**. Tables look like relational-database tables, with rows, columns, and versioned values.

Spanner's data model is not purely relational, in that rows must have names. More precisely, every table is required to have an ordered set of one or more primary-key columns. This requirement is where Spanner still looks like a key-value store: the primary keys form the name for a row, and each table defines a mapping from the primary-key columns to the non-primary-key columns.

```
1  CREATE TABLE Users {
2    uid INT64 NOT NULL, email STRING
3  } PRIMARY KEY (uid), DIRECTORY;
4
5  CREATE TABLE Albums {
6    uid INT64 NOT NULL, aid INT64 NOT NULL,
7    name STRING
8  } PRIMARY KEY (uid, aid),
9    INTERLEAVE IN PARENT Users ON DELETE CASCADE;
```

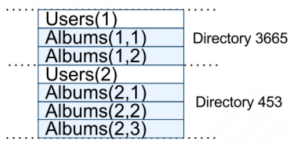


Figure 4: Example Spanner schema for photo metadata, and the interleaving implied by `INTERLEAVE IN`.

Every Spanner database must be partitioned by clients into one or more hierarchies of tables. The table at the top of a hierarchy is a **directory table**. Each row in a directory table with key K, together with all of the rows in descendant tables that start with K in lexicographic order, forms a directory.

This interleaving of tables to form directories is significant because it allows clients to describe the locality relationships that exist between multiple tables, which is necessary for good performance in a sharded, distributed database.

3 TrueTime

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

Denote the absolute time of an event e by the function $t_{abs}(e)$. TrueTime guarantees that for an invocation $tt = TT.now()$, $tt.earliest \leq t_{abs}(e_{now}) \leq tt.latest$, where e_{now} is the invocation event.

TrueTime is implemented by a set of **time master** machines per datacenter and a **timeslave daemon** per machine. The majority of masters have GPS receivers with dedicated antennas; these masters are separated physically to reduce the effects of antenna failures, radio interference, and spoofing. The remaining masters (which we refer to as **Armageddon masters**) are equipped with atomic clocks. An atomic clock is not that expensive: the cost of an Armageddon master is of the same order as that of a GPS master. All masters' time references are regularly compared against each other. Each master also cross-checks the rate at which its reference advances time against its own local clock, and evicts itself if there is substantial divergence. Between synchronizations, Armageddon masters advertise a slowly increasing time uncertainty that is derived from conservatively applied worst-case clock drift. GPS masters advertise uncertainty that is typically close to zero.

Every daemon polls a variety of masters to reduce vulnerability to errors from any one master. Some are GPS masters chosen from nearby datacenters; the rest are GPS masters from farther datacenters, as well as some Armageddon masters. Daemons apply a variant of Marzullo's algorithm to detect and reject liars, and synchronize the local machine clocks to the non-liars. To protect against broken local clocks, machines that exhibit frequency excursions larger than the worst-case bound derived from component specifications and operating environment are evicted.

Between synchronizations, a daemon advertises a slowly increasing time uncertainty. ϵ is derived from conservatively applied worst-case local clock drift. ϵ also depends on time-master uncertainty and communication delay to the time masters.

In our production environment, λ is typically a sawtooth function of time, varying from about 1 to 7 ms over each poll interval. λ is therefore 4 ms most of the time. The daemon's poll interval is currently 30 seconds, and the current applied drift rate is set at 200 microseconds/second, which

together account for the sawtooth bounds from 0 to 6 ms. The remaining 1 ms comes from the communication delay to the time masters.

4 Concurrency Control

4.1 Timestamp Management

Operation	Concurrency Control	Replica Required
Read-Write Transaction	pessimistic	leader
Read-Only Transaction	lock-free	leader for timestamp, any for re
Snapshot Read, client-provided timestamp	lock-free	any
Snapshot Read, client-provided bound	lock-free	any

A read-only transaction must be predeclared as not having any writes.

4.1.1 Paxos Leader Leases

Spanner's Paxos implementation uses **timed leases** to make leadership long-lived (10 seconds by default). A potential leader sends requests for timed lease votes; upon receiving a quorum of lease votes the leader knows it has a lease. A replica extends its lease vote implicitly on a successful write, and the leader requests lease-vote extensions if they are near expiration. Define a leader's **lease interval** as starting when it discovers it has a quorum of lease votes, and as ending when it no longer has a quorum of lease votes (because some have expired). Spanner depends on the following **disjointness invariant**:

For each Paxos group, each Paxos leader's lease interval is disjoint from every other leader's.

The Spanner implementation permits a Paxos leader to abdicate by releasing its slaves from their lease votes. To preserve the disjointness invariant, Spanner constrains when abdication is permissible. Define s_{max} to be the maximum timestamp used by a leader. Subsequent sections will describe when smax is advanced. Before abdicating, a leader must wait until $TT.after(s_{max})$ is true.

4.1.2 Assining Timestamps to RW Transactions

Transactional reads and writes use two-phase locking. As a result, they can be assigned timestamps at any time when all locks have been acquired,

but before any locks have been released. For a given transaction, Spanner assigns it the timestamp that Paxos assigns to the Paxos write that represents the transaction commit.

Spanner depends on the following **monotonicity invariant**:

Within each Paxos group, Spanner assigns timestamps to Paxos writes in monotonically increasing order, even across leaders

This invariant is enforced across leaders by making use of the disjointness invariant: a leader must only assign timestamps within the interval of its leader lease. Note that whenever a timestamp s is assigned, s_{max} is advanced to s to preserve disjointness.

Spanner also enforces the following **external consistency invariant**:

If the start of a transaction T_2 occurs after the commit of a transaction T_1 , then the commit timestamp of T_2 must be greater than the commit timestamp of T_1.

Define the start and commit events for a transaction T_1 by e_i^{start} and e_i^{commit}; and the commit timestamp of a transaction T_i by s_i. The invariant becomes

$$t_{abs}(e_1^{commit}) < t_{abs}(e_2^{start}) \Rightarrow s_1 < s_2$$

The protocol for executing transactions and assigning timestamps obeys *two* rules, which together guarantee this invariant, as shown below. Define the arrival event of the commit request at the coordinator leader for a write T_i to be e_i^{server}.

- **Start**: The coordinator leader for a write T_i assigns a commit timestamp s_i no less than the value of $TT.now().latest$, computed after e_i^{server}
- **Commit Wait**: The coordinator leader ensures that clients cannot see any data committed by T_i until $TT.after(s_i)$ is true. Commit wait ensures that s_i is less than the absolute commit time of T_i, or $s_i < t_{abs}(e_i^{commit})$. Now we get

$$\begin{aligned} s_1 &< t_{abs}(e_1^{commit}) && \text{(commit wait)} \\ t_{abs}(e_1^{commit}) &< t_{abs}(e_2^{start}) && \text{(assumption)} \\ t_{abs}(e_2^{start}) &\leq t_{abs}(e_2^{server}) && \text{(causality)} \\ t_{abs}(e_2^{server}) &\leq s_2 && \text{(start)} \\ s_1 &< s_2 \end{aligned}$$

4.1.3 Serving Reads at a Timestamp

The monotonicity invariant allows Spanner to correctly determine whether a replica's state is sufficiently up-to-date to satisfy a read.

Every replica tracks a value called **safe time** t_{safe} which is the maximum timestamp at which a replica is up-to-date. A replica can satisfy a read at a timestamp t if $t \leq t_{safe}$.

Define $t_{safe} = \min(t_{safe}^{Paxos}, t_{safe}^{TM})$, where each Paxos state machine has a safe time t_{safe}^{Paxos} and each transaction manager has a safe time t_{safe}^{TM}.

5 Problems

1. 2.3: Why?

6 References

References