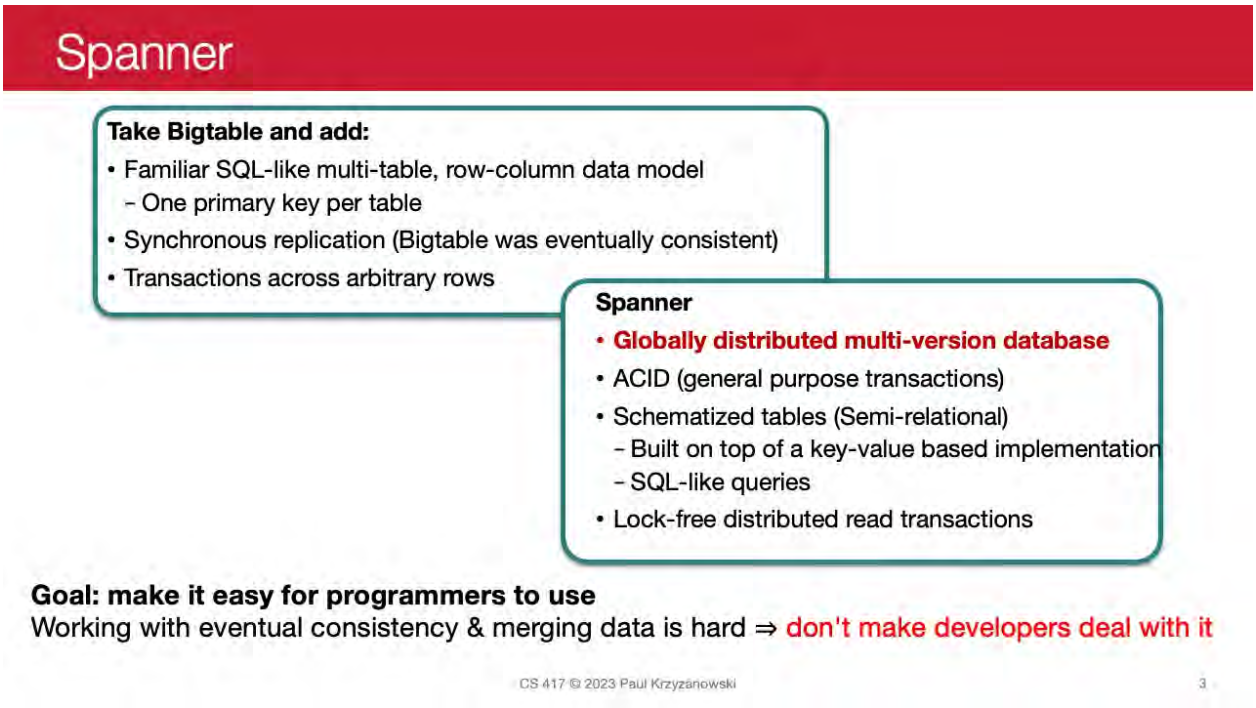
Distributed Systems Week 9: Spanner

Paul Krzyzanowski
November 7, 2023

We discussed the designs of Bigtable and Cassandra. Now we will look at some of the highlights of Spanner, which is Google's globally distributed database. It combines elements of Bigtable, clock synchronization, the two-phase commit protocol, strict two-phase locking, wound-wait concurrency control, multi-version concurrency control, and state machine replication.



Bigtable and Spanner

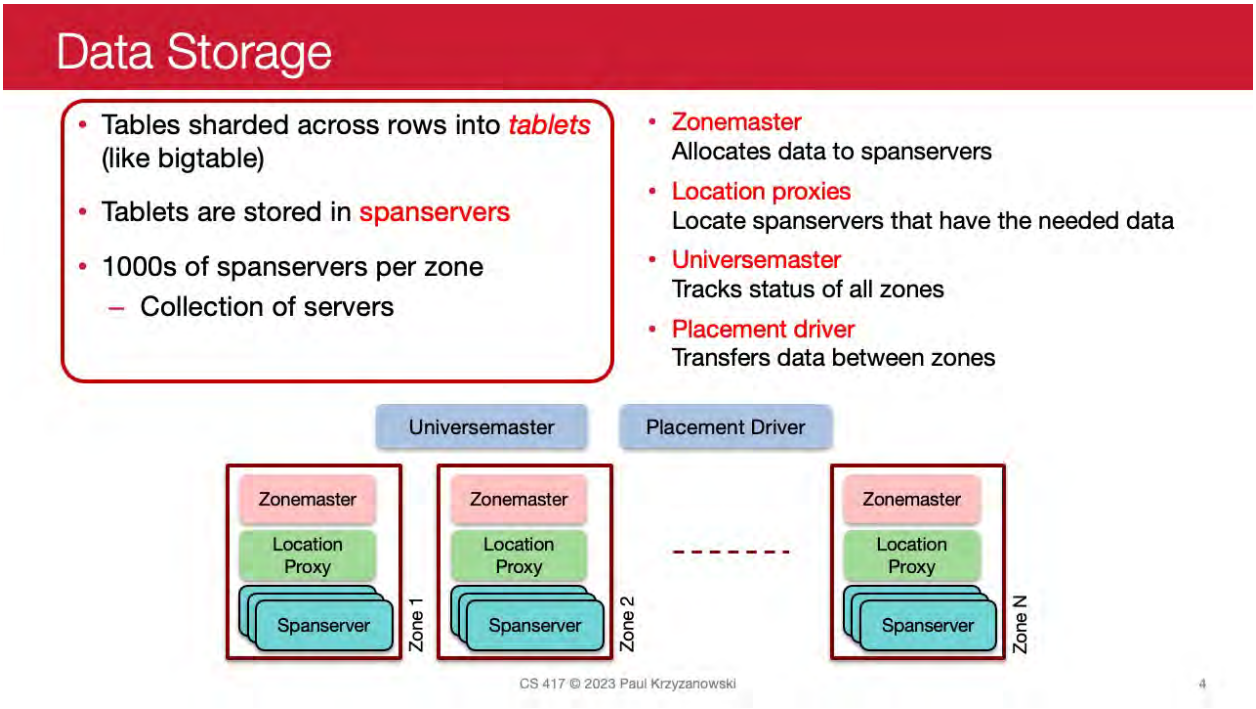
We recently looked at the architecture of Bigtable, which was designed to allow us to store vast amounts of data all structured as one really big table.

Bigtable

Bigtable gave us a single table that had rows and columns. It was essentially a two-dimensional grid with a third dimension of time that gave us versions of objects in each cell. If you created multiple tables, the software did nothing to link them together like a relational database would. You would be responsible for all that, including all locking and distributed commit operations that would be needed to keep data consistent. This was tedious, difficult, and error-prone.

Spanner

panner is basically the evolution of Bigtable into a huge, distributed, multi-table database. We can look at Spanner as a collection of Bigtables. Unlike Bigtable's eventual consistency model for replication, Spanner will provide consistent updates across multiple tables and replicas.



Spanner architecture: data storage

Let's look at how Spanner is organized. Each table is broken up across groups of rows into a bunch of tablets.

Tablets are stored on spanservers. You can think of them as similar to the tablet servers in Bigtable. Each spanserver holds multiple tablets. Note that these are just compute servers that access a distributed filesystem.

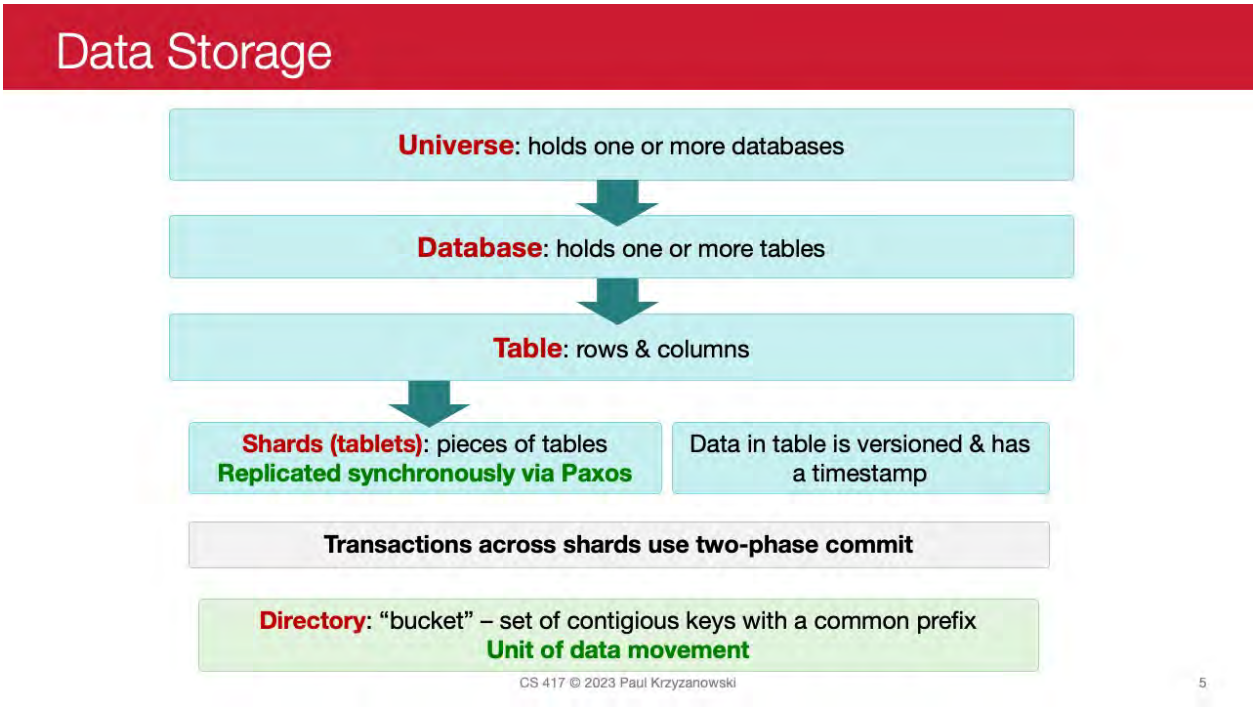
Then we have a bunch of zones. A zone is just a collection of servers that can all run independently. Think of it as a datacenter. You can have thousands of spanservers sitting inside each zone.

Coordinating the activity in a zone is the zonemaster. The zonemaster allocates data to spanservers. It tells each spanserver what it has to do and whether tablets need to be moved to different spanservers.

Each zone has a location proxies. Location proxies are responsible for locating the spanservers that contain the needed data.

Globally, outside of each individual zone, we have a universe master. It tracks the status of Spanner across multiple zones – that is, multiple data centers.

Working alongside the universe master is a placement driver. This is a high-level allocator that manages the movement of data between different zones.



Data storage hierarchy

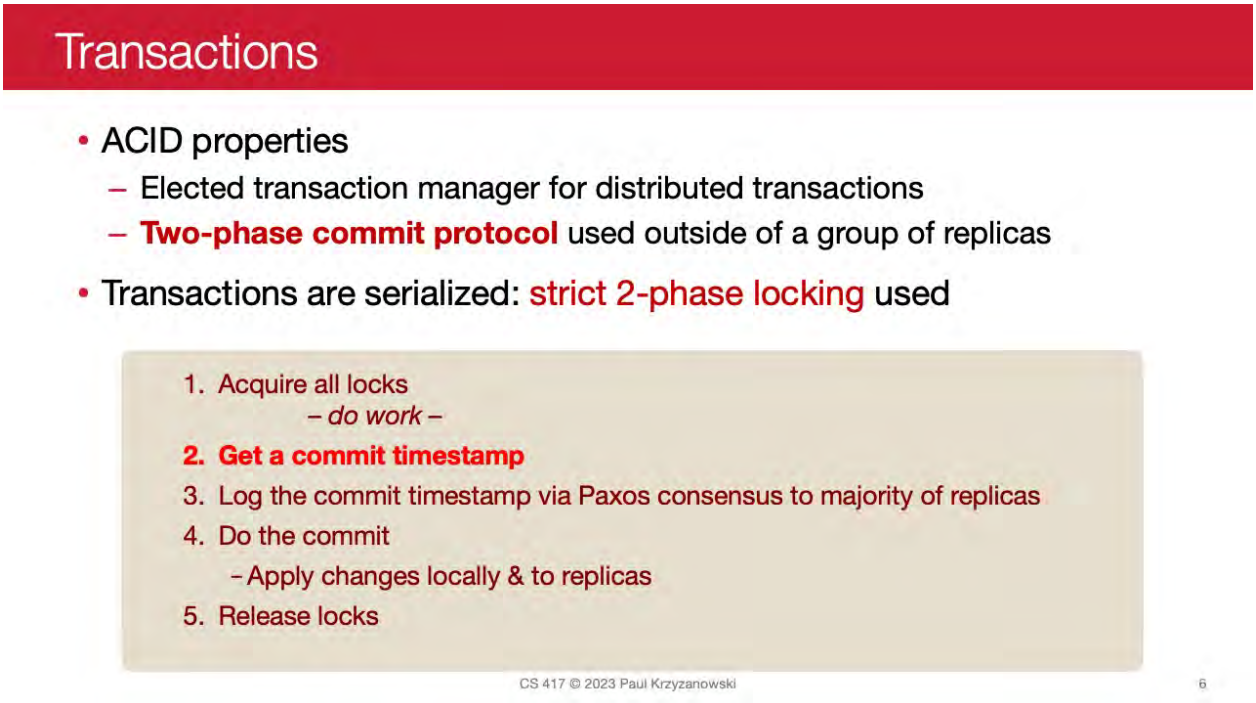
We can look at the hierarchy this way:

The universe holds one or more databases. This is the collection of everything we have. A database - a single instance - holds one or more tables if it's just one table it's kind of similar in concept to Bigtable. The table contains rows and columns kind of like we saw in Bigtable.

Each table is a collection of tablets, also called shards. Tablets are pieces of a table broken across row boundaries. For fault tolerance, tablets are replicated across multiple servers. Spanner uses Paxos, which is a fault-tolerant consensus protocol similar to Raft to do synchronous replication of data onto multiple servers.

All the data inside a table is versioned so we have a timestamp for every version of the data. Any transactions that touch multiple shards or interact with different tables will use a two-phase commit protocol.

Spanner also has the concept of a directory, which is an unfortunate name since it has nothing to do with filesystem directories. A directory can be thought of as a bucket in a tablet — a set of contiguous keys that share a common prefix. This sounds a lot like a tablet in Bigtable. In Spanner, a tablet doesn't necessarily hold a contiguous sorted set of rows like it does in Bigtable. Instead, a directory structure allows contiguous rows from different tables to be interleaved together to make some local join operations across those tables to be efficient. The directory is the unit of data movement.



Transactions

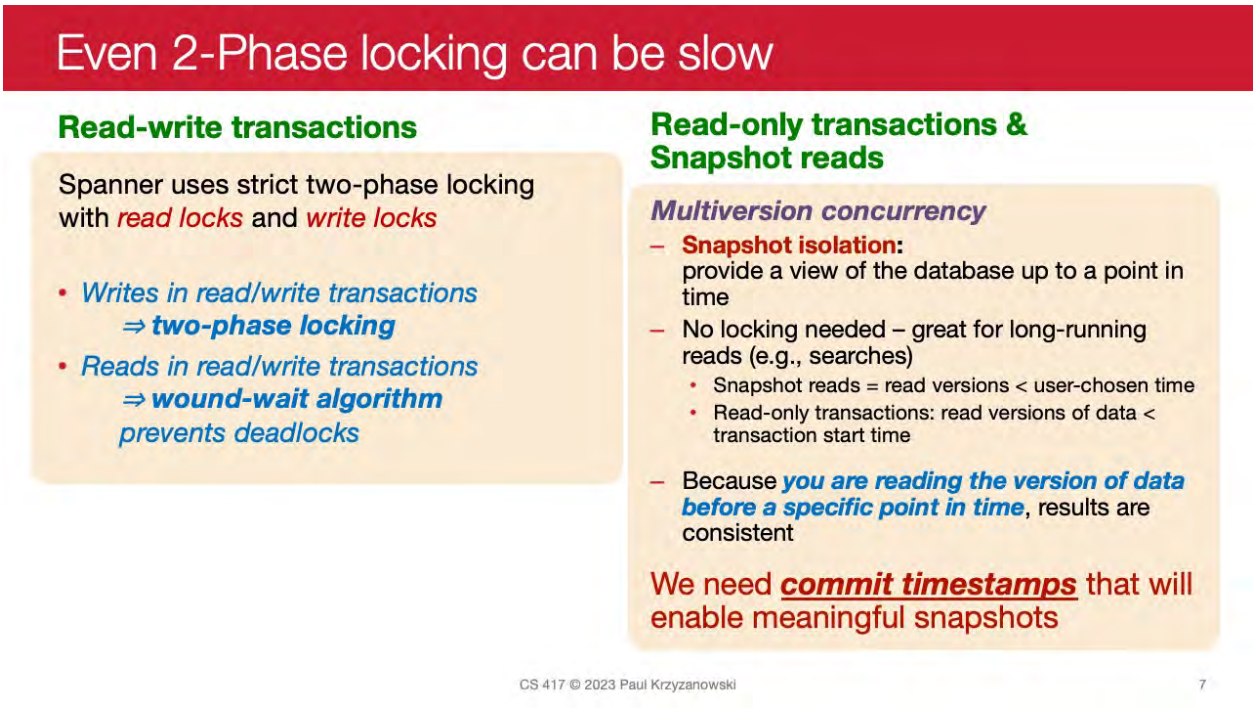
Unlike Bigtable — because Spanner was designed to be transactional and handle operations across multiple rows and across multiple tables — Spanner provides ACID semantics.

To provide this model of consistency, it uses the two-phase commit protocol for distributed transactions using an elected transaction manager. To guarantee ACID semantics, one transaction must not interfere with the effects of another transaction if the transactions are running concurrently. Spanner uses strict two-phase locking to ensure that a transaction does not read or write data that is being used by another transaction.

We saw how two-phase locking works. The transaction first acquires locks on the rows it will access. Acquiring all the needed locks is the growing phase. The transaction then does its work and releases the locks when it commits.

In Spanner, before the transaction commits, it will first get a *commit timestamp*. The commit timestamp reflects the serialization order of transactions – their logical sequencing. This applies across all transactions across all systems globally.

Completing thecommit means that each transaction makes its modifications permanent. The transaction then propagates its changes to all the replicas so they could be made permanent on the replicas during the commit. At this point, all the fields that the transaction is modifying are locked — so no one else can see those changes. When it's finally done and all the replication is finished, then the transaction releases the locks and everyone can see the changes.



Improving concurrency

The problem with two-phase locking is it could be slow and reduce the amount of concurrency. If we're modifying a lot of rows of data, then nobody else will be able to access that data.

Read-write transactions

We've seen ways to improve concurrency. We can use separate read locks and write locks. Spanner does this for transactions that modify data (read-write transactions).

If all we're doing is reading a data element and not modifying it there's no harm in having other transactions read the same data. You can have multiple *read locks* on the same data at any given time. If you're writing a data element, a *write lock* gives you exclusive access to the data.

Read and *write* locks have a blocking property. Read locks can block behind write locks: if a transaction is doing a write, it will have a *write lock* so nobody else can read that data.

And similarly, *write locks* block behind *read locks*. If a transaction needs to modify data, it will have to wait until any transactions that are reading the data have finished reading and released the lock so they won't see different versions of the data.

Strict two-phase locking

Spanner uses different forms of concurrency control based on what the transaction needs to do. If it's modifying data, Spanner uses read-write locks and strict two-phase locking, which is a pessimistic approach to concurrency control.

Wound-wait deadlock prevention

To avoid deadlock in reading data, Spanner uses the wound-wait algorithm. With this algorithm, there is a chance the transaction would need to be aborted.

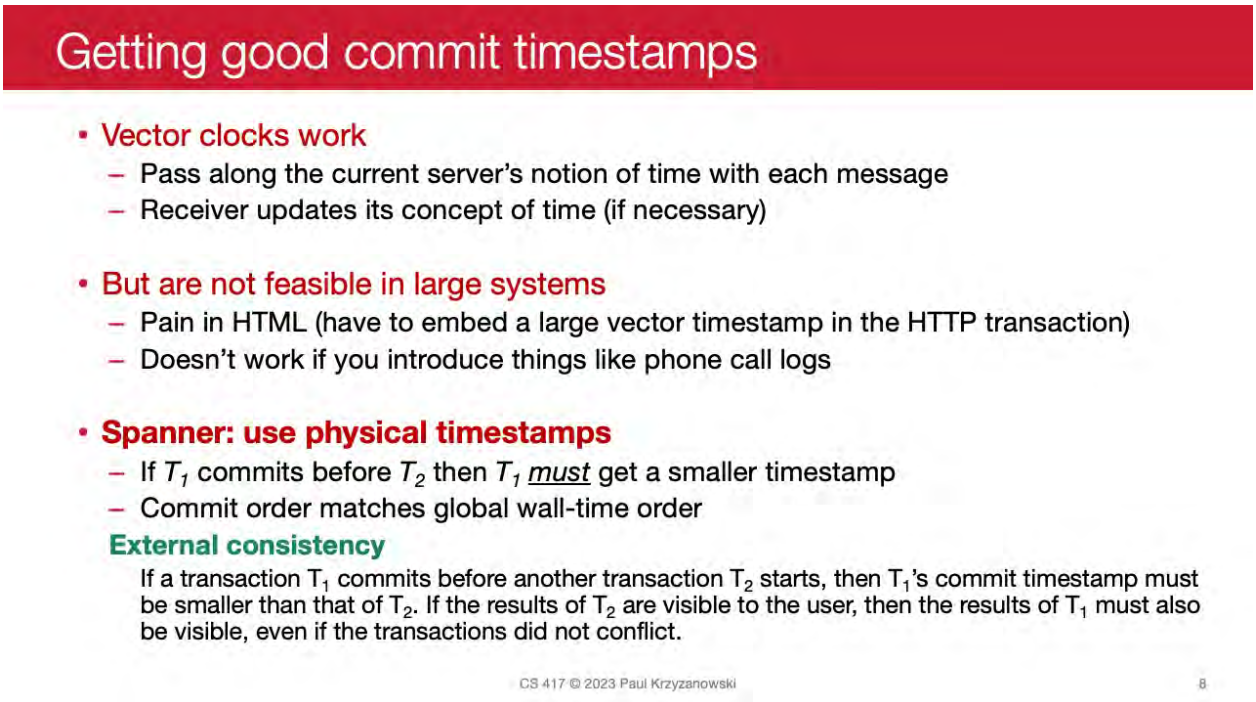
Read-only transactions and snapshot reads

For read-only transactions, Spanner takes advantage of the fact that each cell stores older versions of data, each with a timestamp of the transaction that modified it. This allows it to implement multi-version concurrency for read-only transactions.

With multi-version concurrency, we can present a view of the database for any transaction the way it looked at a specific point in time. This makes long-running reads, such as searching through a really huge set of data non-disruptive. They don't hold up anything else – they are just reading versions of data as it existed at a particular point in time.

Because a transaction is reading data before some point in time, all the data will be consistent. Note that we need good timestamps for this to work, which we will look at shortly. If a transaction defines itself as being read-only, it will be presented with versions of data that are no newer than the start time of the transaction. Snapshot isolation is the same as a read-only transaction except that the user can specify the point in time.

To implement this, we need to generate commit timestamps that will allow us to have meaningful snapshots.



Getting good commit timestamps

To get useful commit timestamps we could use vector clocks. To use vector clocks, we pass along the server's current concept of time along with every message that we send to other servers and every receiver will have to update its concept of time based on the vector.

A problem with vector timestamps is they're often not practical in large systems. A vector represents all the components that are involved in the transaction so if you're looking at having thousands of systems your vector is going to have thousands of elements in it and that becomes inefficient to store and move around.

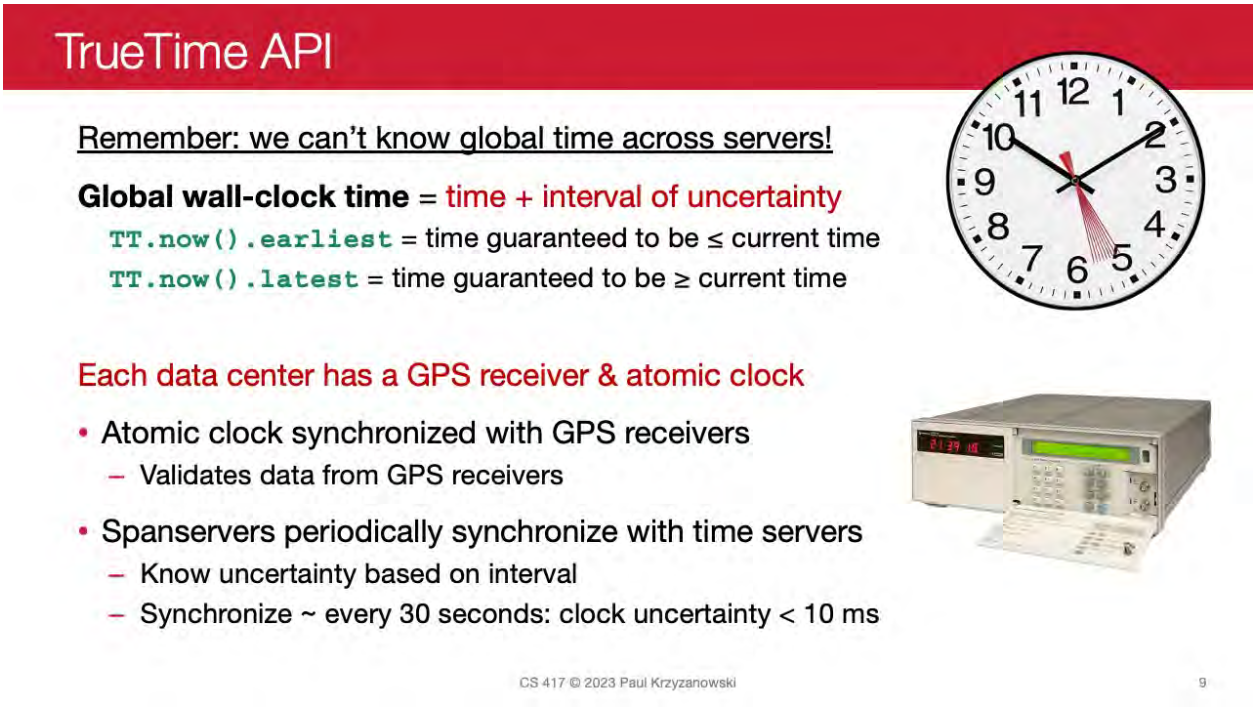
Moreover, if you're communicating with the system using something like HTML over HTTP (e.g., capturing website click data) that becomes even messier because now you must embed these potentially large vector timestamps inside HTTP transactions. If you're doing applications such as using phone call logging then you even have no place to even put these vector timestamps.

What Spanner did was go against this common wisdom and use physical timestamps. When we looked at timestamps, we dismissed physical timestamps because you cannot do global time ordering because no two systems can be guaranteed to synchronize to the exact same time.

Spanner provides a property of **external consistency**, also called linearizability to its transactions. With external consistency, the requirement is that if we a transactions T_1 commits

before another transaction T_2 then transaction T_1 must get a smaller timestamp than transaction T_2.

If T_1 has a smaller timestamp than T_2, then the commit order must match this timestamp order. The timestamps, in turn, are not logical clocks but refer to the physical time — referred to as wall time order. Wall time is the time you see on the clock on the wall — it's the real time.



TrueTime API

To implement external consistency, Spanner makes use of TrueTime via a TrueTime API. Keep in mind that we can never know the real global time consistently across servers. We cannot guarantee that we can synchronize the clock on every single server to the exact nanosecond.

What Spanner does is define the global wall clock time.The wall clock time is the current time plus some interval of uncertainty. The TrueTime API — abbreviated as TT — doesn't give us a specific time. It doesn't tell us what the time really is but it gives us two times: the earliest time and the latest time that could possibly be right now.

TT.now().earliest is the time that is guaranteed to be less than or equal to the current time and TT.now().latest is the timestamp that is guaranteed to be greater than or equal to the current time. TrueTime provides us with an interval of time and we want that interval to be small in order to minimize this range of uncertainty.

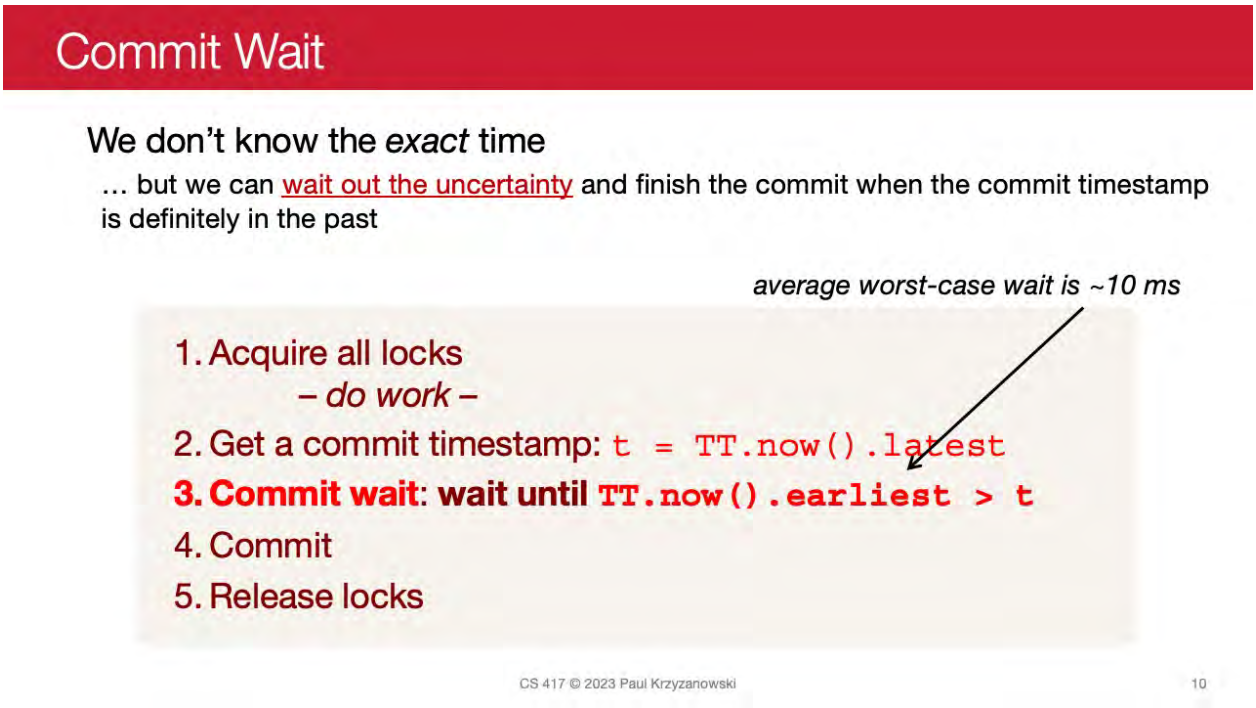
To make that window as small as possible, every data center that runs Spanner – which is every Google data center — has both a GPS receiver and an atomic clock. The atomic clock is periodically synchronized from GPS receivers. This atomic clock provides fault tolerance in case the system cannot read the GPS receiver – or if the entire GPS system is jammed. By having the atomic clock at each data center, we don't incur the latency of synchronizing from remote sources.

All servers in Spanner periodically synchronize with these local time servers that, in turn, synchronize their time directly from the GPS receiver or atomic clock.

Every spanserver knows what the timestamp's amount of uncertainty is. We saw how we get with Christian's algorithm — when you synchronize from another server and know the network latency and the accuracy of the clock source, you can compute the error of the synchronization. The error also takes into account how long ago the time server last synchronized. TrueTime uses Marzullo's algorithm, which was developed to choose an accurate time from multiple noisy time sources and account for errors.

The TrueTime API provides us with that window of uncertainty. To make the window as small as possible, systems synchronize from highly accurate local time sources and typically synchronize approximately every 30 seconds.

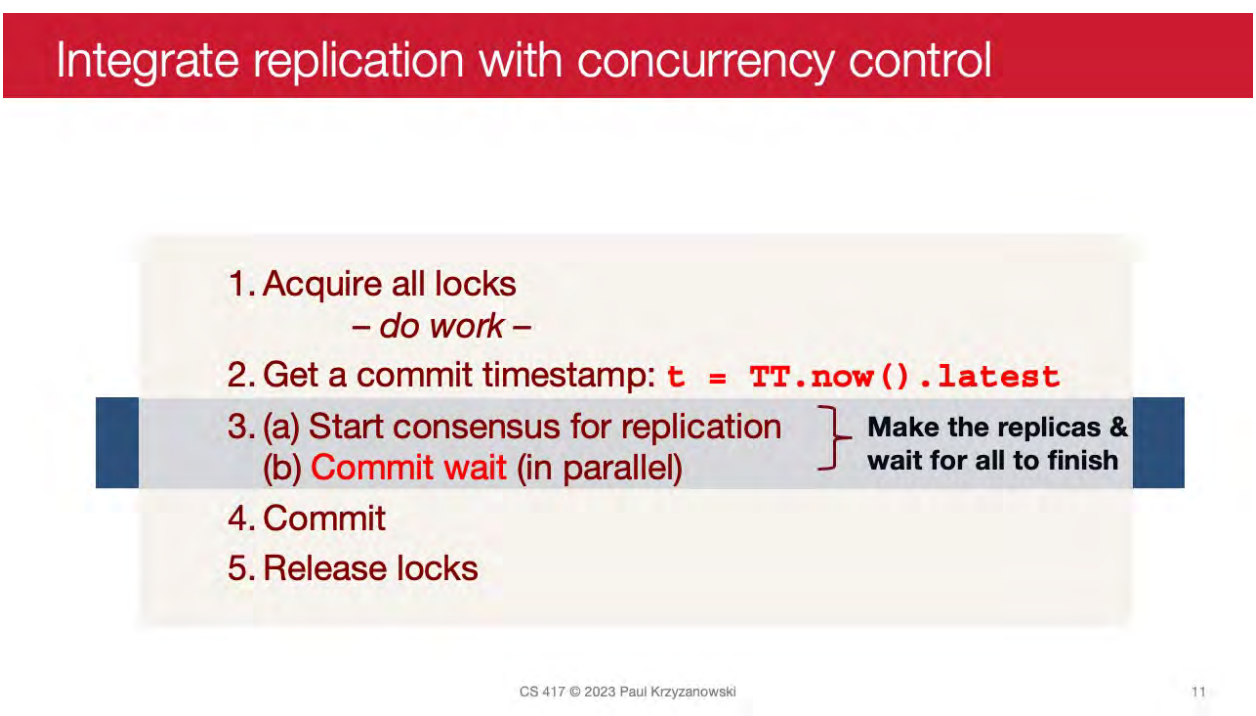
The clock uncertainty on a server is usually within about 10 milliseconds so TrueTime gives us an earliest time and the latest time and the difference between these two times is usually approximately 10 milliseconds.



Commit wait

Let's look at how commits work with the use of the TrueTime API. Remember that we don't know the exact time but we can wait out TrueTime's region of uncertainty. We still take the same steps to commit.

1. A transaction first acquires locks and does whatever work is needed for the transaction.
2. Then the transaction gets a commit timestamp. This is the latest time: TT.now().latest. It's the latest time that it can possibly be right now.
3. The next thing we do is a **commit wait**. A commit wait means we do nothing but wait until we're sure that that commit timestamp is now sometime in the past. We do this by waiting until TT.now().earliest has passed the timestamp we recorded earlier. The average worst-case wait time is going to be around 10 milliseconds, so we really don't have to wait all that long.
4. When that's done, we do the normal commit process and then release the locks. Now we're done with the transaction.



Integrate replication

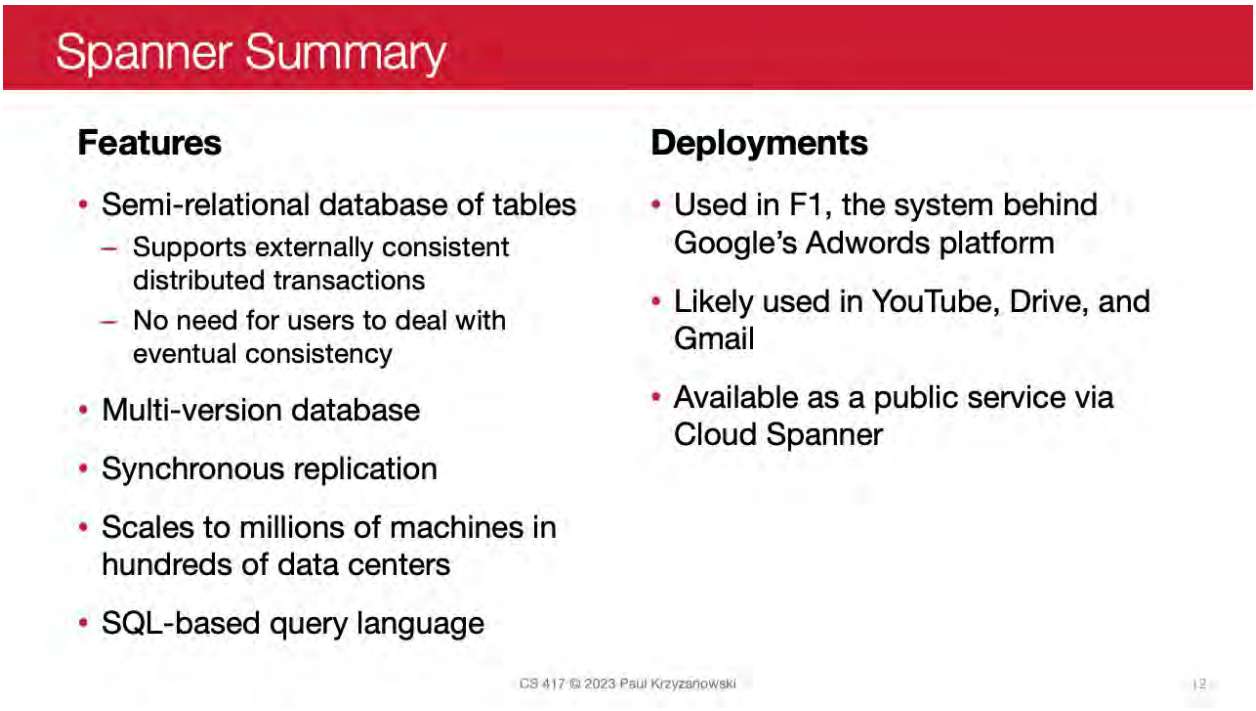
Spanner integrates replication because we need fault tolerance. It goes through the same steps we discussed previously: we acquire all the locks we need and do the work for the transaction. When the work is done, we get the commit timestamp. This is the timestamp of the latest time that it currently can be.

Now we prepare for replication since we have no more changes to the data since we are ready to commit. Replication uses a consensus protocol for fault tolerance. Spanner uses Paxos but we can do the same thing with Raft state machine replication.

The replication protocol ensures that all the replicas have the changes that we made. We also send the commit timestamp to each replica so they can properly version all the data that each replica is modifying.

When the replication is complete, we have all the replicas including ourselves do a commit wait in parallel. The commit wait will delay the commit until the timestamp that we were given for the commit is definitely in the past.

When that's done, we finish our commit, release our locks and we're done with the transaction.



Summary

Features

Spanner is a huge-scale semi-relational database made of tables. Unlike Bigtable, it supports multiple tables and users can apply SQL queries on these tables.

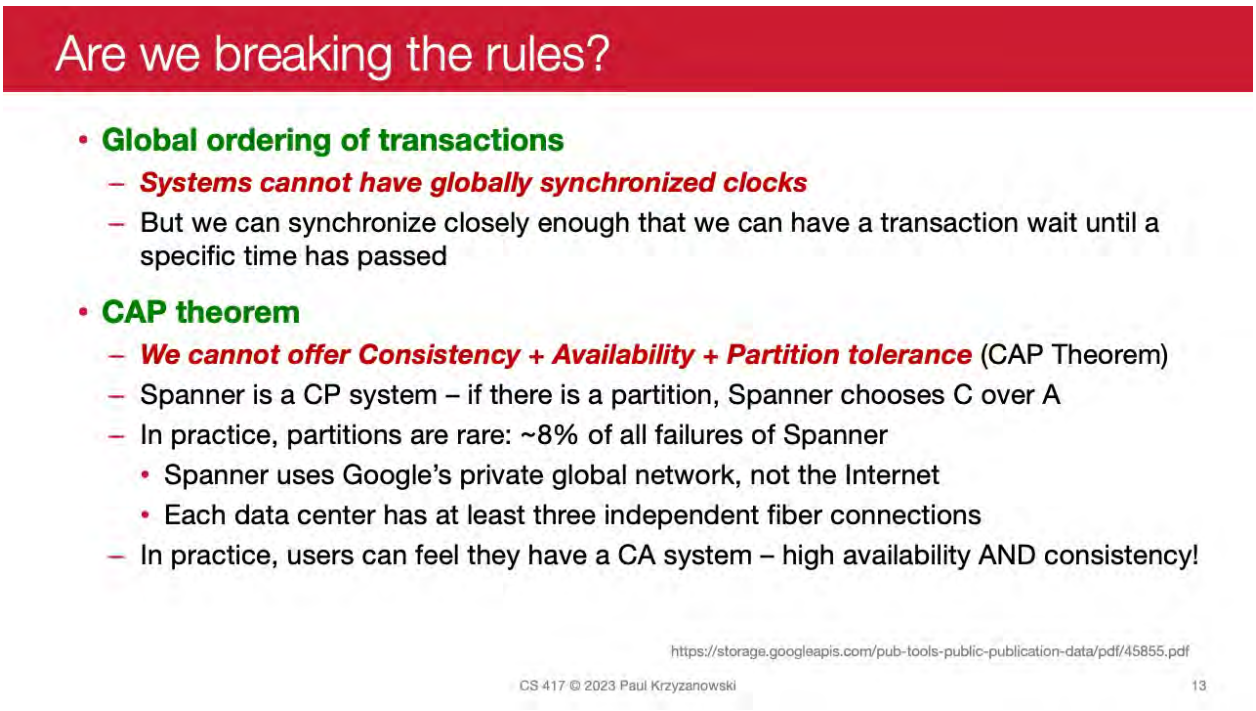
Spanner gives us an externally consistent set of distributed transactions so anyone anywhere in the world who looks at the database sees a consistent view of the database even if they are doing long-running searches — because each transaction is always looking at a certain point in time. The transaction sees data that was valid when the transaction started so it will not see any later changes.

Because of that, users don't have to try to deal with the problems of eventual consistency models. They always see consistent data. To support this, Spanner is a multiversion database: each record in every row and column across all tables is a cell that contains multiple versions of timestamped data.

Spanner also supports synchronous replication so all those tables can be replicated across multiple systems. They are replicated in a way where no one will access inconsistent versions of data.

Deployments

Spanner is designed to scale to support millions of machines and hundreds of data centers. It's used in various services within Google. For instance, it's used in F1, which is the internal system behind Google's AdWords platform. It may also be used in Gmail and Google search and many other services. Google just hasn't disclosed exactly where it is deployed.



Are we breaking the rules?

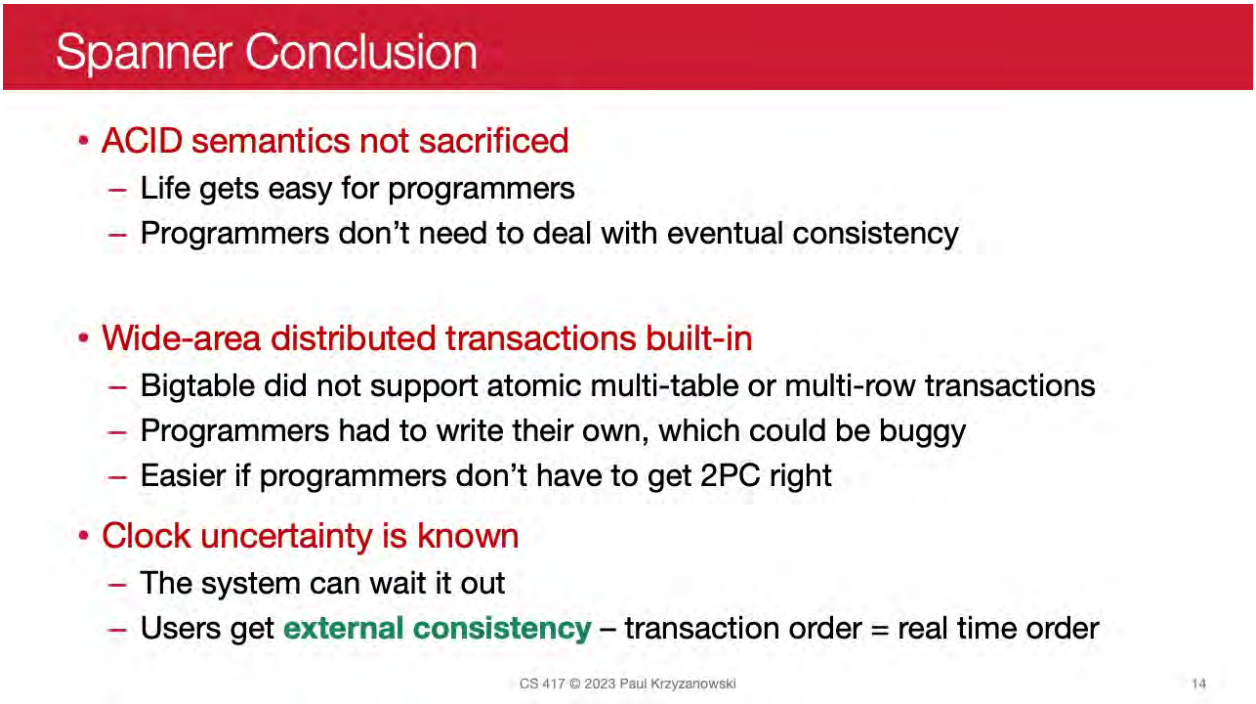
In some ways, Spanner looks too good to be true. It provides us with consistent global time ordering of transactions, and yet we know systems cannot have globally synchronized clocks. But what we can do is synchronize the clocks closely enough so that we can wait to be sure that we have a particular time that has passed. Spanner simply waits until the transaction timestamp is definitely in the past thus creates the appearance of global time ordering.

We also have to consider the CAP theorem, which states that we cannot offer both consistency and availability when partitions may occur. In some ways, it seems like Spanner is breaking the rules because it looks like it's giving us a highly available system that also is completely consistent.

In reality, Spanner is a CP system. It gives us consistency above everything else. If there is a partition, Spanner chooses consistency over availability. Replication and commits will be delayed. The eventual consistency model (AP) became popular because we specifically did not want to wait in these circumstances. We wanted to prioritize high availability over consistency.

We can argue that we don't get true high availability with Spanner. But with the design of data centers in the Google environment, along with the redundant networks that connect data centers together, partitions are rare. Google's data centers connect through Google's private global network and do not rely on the worldwide public Internet. Every data center has at least three independent fiber connections to other data centers and there's a lot of redundancy inside each data center.

In practice, partitions are pretty rare in this environment. They account for approximately 8% of all failures of Spanner. So, partitions do occur and you have to design for them. When they occur, transactions will wait and consistency will win out over availability. Users feel like they get both consistency and high availability because partitions hardly ever occur.



Spanner conclusion

To conclude, Spanner provides a system where ACID semantics are not sacrificed. Programmers can really feel like they're using a traditional database. They get all the consistency they expect in a traditional database.

Life gets easy for programmers because they don't have to worry about accessing inconsistent data. They also don't have to program custom own solutions to deal with unwanted eventual consistency.

Wide-area distributed transactions are built into the architecture of the system. Bigtable didn't support distributed transactions. It supported single row transactions and programmers had to write their own if they wanted consistency for transactions across multiple rows or tables.

Spanner supports this and programmers don't have to worry about getting a two-phase commit protocol or concurrency control mechanisms implemented correctly. Clock synchronization and inconsistencies are handled automatically by the framework, giving users a property of external consistency – that transaction timestamps reflect real-time ordering.

The End.
