May 1, 2013

Lecture 20/21 -- Google Spanner

Quiz 2 next week based on Lectures 13--22 (no OCC, but streaming).

<u>What is the main idea behind Spanner?</u>

To build a transactional storage system replicated in the wide area.

<u>Why are we reading this?</u>

Contrary to conventional wisdom, Google says that running 2PC and synchronous replication is practical in the wide area. Although individual transaction latencies are high (due to multiple network round trips), the overall system throughput can still be good.

Report using it to support the ad system. A bit unclear what this means, but presumably this is the database that records information about each customer who buys ads on Google, how much they paid, how many (and which?) ads were served, click through rate, etc. This is a very large database, but each customer doesn't necessarily have a ton of data or require particularly low latency.

<u>What are the key ideas?</u>

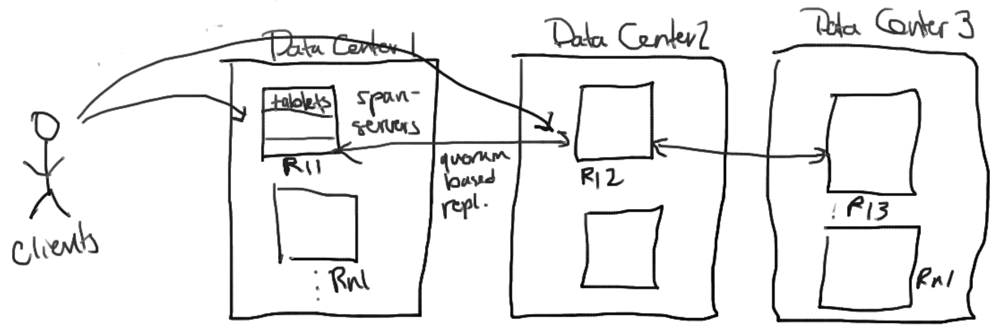
- Relational data model with SQL and general purpose transactions (interesting to see that Google has evolved from MapReduce to a general purpose database system!)

- "External consistency" -- transactions can be ordered by their commit time, and commit times correspond to real world notions of time

- Paxos-based replication, with number of replicas and distance to replicas controllable.

- Data partitioned across thousands of servers

<u>Diagram:</u>



Basic model is just a distributed database, running 2PC and locking to coordinate transactions. Each node ("spanserver") is actually replicated using Paxos algorithm.

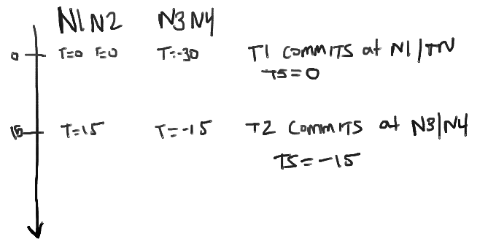
First, let's look at how external consistency works.

Key property: all writes in a transaction are globally ordered by timestamp

Additionally, if the start of a transaction T2 occurs after the commit of a transaction T1, then the commit timestamp of T2 must be greater than the commit timestamp of T1.

This sounds obvious, but if machines aren't time synchronized, then not guaranteed to be true.

For example, suppose there are two nodes N1 and N2 running T1 and two nodes N3 and N4 running T2. Suppose N1 and N2 are running 30 seconds ahead N3 and N4. Suppose T1 commits at time T=0 on node N1/N2 (time T=-30 on N3/N4). If T2 commits 15 seconds later, and is assigned time T=-15 by N3/N4, then T2 will appear to have a commit time that is earlier than T1 to an external observer (or another replica trying to recover).
Diagram:

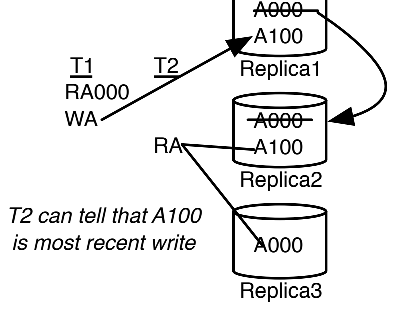


Why is this a problem?

Suppose T1 and T2 both wrote some data item A. Then a replica that is recovering may apply T1's writes to A (because they have a later time stamp) than T2's. If everyone agrees on timestamps then we can just use timestamps to determine the most recent write to each objects.

How else can we solve this?

Using vector clocks, as Adam discussed Monday. Each node is assigned a monotonically increasing sequence number. Example:



What is wrong with vector clocks? Nothing, except that the values in vectors don't correspond to any notion of real world time (it's not obvious how to get a real world ordering of T1/T2 from the vector values.)

Ok, so how do we ensure that all nodes have consistent clock values?

Use time synchronization (GPS + atomic clocks on some nodes), plus network time synchronization protocols (where nodes exchange times with each other and adjust their clocks accordingly.)

But can't there still be small differences between clocks on nodes?

Yes! They define an API called TrueTime that is able to estimate the accuracy of a node's clock, guaranteeing that a timestamp T lies in an interval between two actual times *T.earliest* and *T.latest*

Suppose T1 commits at time t1 on N1, and T2 commits at time t2 on N2. We want to ensure that T1 is assigned a commit time that is before T2's. How do we do this?

We could do it trivially by ensuring that T2 doesn't commit until after t1.latest , e.g., that t2.earliest > t1.latest. But then N1 and N2 would have to know about all of each other's transactions.

Instead, if we just ensure that T1 holds its locks until t1.latest, then we can ensure that T1 commits before T2 commits.

Paxos-based replication

Why do we need this?

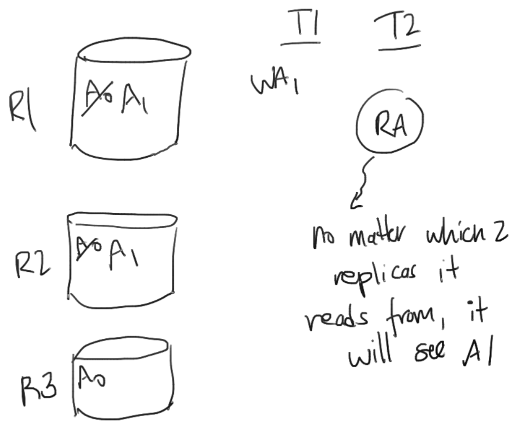
We want to be able to tolerate the failure / disconnection of some nodes / data centers.

Idea is to ensure that all writes and reads go to a "quorum" of nodes.

What's a quorum? Generally a simple majority. E.g., if we have N nodes, read and write to floor(N/2) + 1 nodes.

(In reality, we send write and read request to all nodes, but only wait for a quorum of them to complete.)

We saw the basics of quorum-based replication last time. Example with 3 nodes. We send all writes to 2 nodes, all reads to 2 nodes.

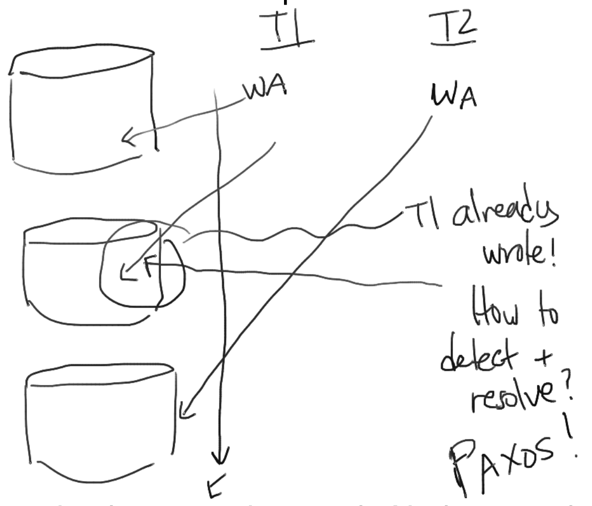


These nodes form a "replicated state machine" -- idea is that each node processes requests in the same order, which will ensure that all nodes have identical state and can be used to process any read / write request.

Paxos is a principled way to make this approach work. Why is this tricky?

Suppose three nodes all have a consistent view of object A w / version 1.

Suppose two concurrent writes to object A are sent to nodes 1 and 2, respectively. They both attempt to propagate the write.   Example:



(Node 1 "wins" and it's version becomes installed. Node 2 needs to know to discard its write because it couldn't get a quorum to agree to accept it, and needs to apply node 1's write.)

To handle this, we need a multi-round protocol kind of like two phase commit (but unlike 2PC we don't need to write to all nodes, just a quorum!)

Basically, each "proposer" of a new value for a record tries to get a quorum of replicas to agree to accept its value, and then commit its value to that quorum. This ensures that some proposer will "win" by writing to a quorum (and that no one else will be able to get a quorum.)

But protocol is more subtle than that because any of the replicas or proposers could fail at any point, and we don't want replicas that have agreed to accept a value to be unable to make progress / have to wait for failed proposer to come back online. (As would happen in 2PC if the coordinator failed.)

Not going to describe or analyze protocol in detail -- take 6.828 to learn about it more.

Spanner uses a generalization of Paxos know as "multi-Paxos":

Each spanserver runs a single instance of Paxos for each "tablet" of data it stores. It uses Paxos to agree on a "group leader". All writes will be done by the leader to replicas. This means that there are never concurrent outstanding writes to the same object within a replica group as long as the leader is stable, so only one round of communication is needed between leader and replicas to confirm writes.

<u>What if leader fails?</u>

Need to elect a new leader, which they do using timeouts. Each leader has a fixed "lease" on leadership, which it must renew periodically. In effect each replicas keeps track of last time it receive a write or a request to extend the least from the leader. If a replica hasn't heard from leader in some period, it will attempt to elect a new leader using Paxos. If a quorum of nodes agree that they haven't heard from the leader, then a new leader is chosen.

<u>Ok, so how does Spanner distribute data?</u>

Logically, Spanner consists of a collection of database tables.

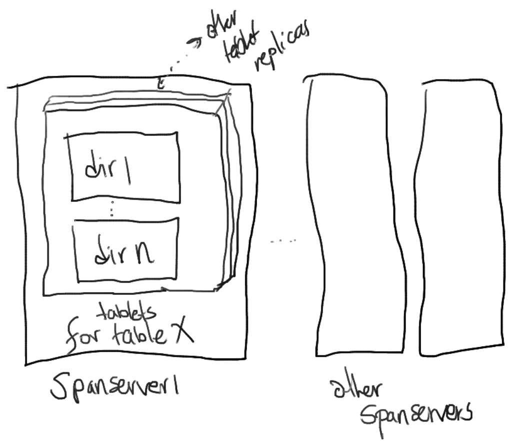
Data in a table is divided into a number of "tablets"

Each "spanserver" contains a number of tablets.

Each tablet divided into "directories"

Directories are unit of replication, and can be moved across servers to balance load. This is apparently done automatically.

Diagram:



Physical layout is done in user specified hierarchies. For example, given table of emp and depts, user can say that emps should be stored in depts:

```
create table depts {
did int, ...
};

create table emps {
did int,
eid int,
} interleave in parent dept
```

This will create one directory per dept.

<u>Why do they do this?</u>

Presumably to promote locality of data for key-FK joins (although they don't really say).

<u>How are transactions executed?</u>

Using two phase locking locking and two-phase commit.

<u>How do you assign timestamps to transactions?</u>

Each participant chooses a timestamp when it receives a prepare message that is greater than the timestamp of any transaction it has previously committed.

Coordinator chooses a commit timestamp T larger that the prepare time stamps of any participants, and coordinators and participants wait until T.latest before releasing their locks, to ensure any transactions that begin after the transaction commits will have a later commit timestamp.

How is the performance?

On a 1 ms network, latencies are ~15ms, and fairly independent of replicas (a lot less for read only transactions).

Throughput is ~2.5 K transactions per sec for one set of spanservers. (Not awesome, for this kind of workload.)

Relatively low relatively to what you would get from an optimized in memory database.

Like most of Google's work -- performance is not essential. Presumably this scales well with span servers (at least for ad workload).

Re: ads -- in 24 hours they are only averaging about 350 writes / sec, with latencies in the 10's -- 100's of msecs.