Spanner

Overview

- Spanner is a scalable, multi-version, globally distributed, replicated database
- Supports externally-consistent (linearizable) distributed transactions (reads and writes)
- Shards data across many Paxos state machines in datacenters around the world
  - Automatically reshards as amount of data / number of servers change
- Designed to scale to millions of machines and trillions of rows
- Used for high availability, even in the face of natural disasters
  - Replicates data within or even across continents
  - Replication constraints can control how far replicas are from each other, how far data is from users, how many replicas are maintained
- Supports general-purpose transactions and a SQL-based query language
- Assigns commit timestamps that reflect a serialization order
  - Based on looking at a clock and its uncertainty, which is factored into how long Spanner waits to ensure that it is accurate
  - Uses the TrueTime API for looking at the clock

Implementation

- A Spanner deployment is called a universe, and there are few running universes
  - Organized into a set of zones where each zone is roughly a Bigtable deployment
  - The set of zones is the set of locations across which data can be replicated
    - Each has a zonemaster and hundreds to thousands of spanservers
    - Zonemaster assigns data to spanservers, and the spanservers serve data
    - They also have location proxies which are used by clients to locate the spanservers assigned to their data
  - The universe master and placement driver are external singletons that are not in a zone
    - Universe master is a console for displaying status info about all zones for interactive debugging
    - Placement driver handles automated movement of data across zones
      - This occurs to meet updated replication constraints or balance load
      - Happens on the timescale of minutes

Spanservers

- A spanserver is distributed across multiple datacenters (zones)
- Each spanserver is responsible for 100-1000 instances of a datastructure called a tablet
  - A tablet implements a bag of mappings from `(key: string, timestamp: int64) -> string`
  - Because each mapping contains a key and a timestamp, it's more of a multi-version database than a key-value store
- A tablet's state is stored in a set of B-tree-like files and a write-ahead log on a distributed file system called Colossus
- Each spanserver implements a Paxos state machine on top of each tablet
  - Used to keep this bag of mappings constantly replicated
- Each spanserver can be thought of as a shard of the data
- Also includes a lock table for concurrency control and a transaction manager to support distributed transactions
  - Transaction manager can be bypassed if a transaction only involves a single shard
- Therefore when clients read their closest replica, they can go to a member of the Paxos cluster and read from them

- Challenges: data on this member may not be fresh and transactions that use multiple shards (spanservers) need to be serializable

- General idea is that each spanserver has some participant leader which is not necessarily the same as the Paxos leader
  - Participant leader stores the locks and transaction manager
- Tells the Paxos replica group (which is a separate bit of running software) the commands it wants it to replicate

Read-Write Transactions

- Example: $x = x + 1; y = y - 1$
  - This entire thing should happen as a single atomic action
- Main idea is that we want to use two-phase commit with the Paxos-replicated participants
- Client picks a unique transaction ID and sends each read to Paxos leader of relevant shard
  - Each shard acquired lock on relevant record
  - Read locks aren't replicated in Paxos so shard leader failure -> abort
- Client keeps the write private until it receives all of the reads
  - Afterwards, it then chooses a Paxos group to act as the two phase commit transaction coordinator (TC)
  - Sends writes to shard leaders which each:
    - Acquire locks on written records
    - Log a prepare record in Paxos to replicate the new written value
    - Tells TC either that is prepared or that it crashed and lost the lock table
  - The TC then decides whether to commit or abort
    - Logs the decision and then tells participant leaders / client the results
  - Each participant leader:
    - Logs the TC's decision via Paxos, performs the write (if not aborted), and then releases the locks
- Since a transaction coordinator is backed by Paxos, we won't have problem with 2PC of TC failing
- This takes a long time (on the order of 100 ms), but with many shards and many clients, we can get high throughput

Read-Only Transactions

- We want this to be faster than read-write transactions (xactions)
- We read from local replicas and avoid Paxos / cross-datacenter messages
- To deal with correctness, we need it to "fit between read/write xactions"
- We can't just read the latest committed values if we want to read multiple things in the same transaction

```
T1:  Wx  Wy  C
T2:                Wx  Wy  C
T3:            Rx             Ry
```

  - T3 will not see a linearizable order in the transaction
- Idea is to synchronize all computer's clocks and assign each transaction a timestamp (Snapshot Isolation)
  - For read/write we use the commit time
  - For read-only we use the start time
- We want the results to occur as if each of them happened in the order of those timestamps
  - All read/only transactions will only look at read/write transactions that committed before it started
  - This is why every tablet included every version of the data at the different timestamps

---

- If T3 reads from a replica that hasn't seen T1's write (i.e. it wasn't in Paxos majority) then we need to fix this
  - Replicas use an idea called "safe time"
  - Before serving a read at time 20, the replica must see Paxos write for time > 20
    - It won't use this write in its decision making, but at least this way it knows it has seen all writes < 20

- Also has to delay if there are prepared but uncommitted transactions with a
- This creates a scenario where the read/only transactions are usually fast

Synchronized Clocks

- This creates a problematic scenario for only r/o xactions if clocks aren't synchronized
  - r/w xactions are fine because they're based off of the locks
- The TrueTime API yields a time interval for each clock that guarantees the correct time is somewhere in the interval
  - Intervals are usually microseconds but can go up to 10+ milliseconds
- Using these we can make sure that we have the correct semantics
  - For r/o transactions, we measure the interval at the start of the transaction
  - For r/w, we measure when the commit begins
  - For the measured interval, we always look at the latest possible measured time
- Before completing a r/w transaction's commit, we wait until this timestamp is less than the earliest possible current time
  - This guarantees that the recorded timestamp has definitely passed
- Then for r/o we just compare its timestamp to make sure we have already seen a recent write (just read from the database)