Distributed Systems

Where We are Going

- CAP Theorem
- Consistency levels
- Distributed Key-Value Stores
  - Fast Array of Wimpy Nodes (FAWN)
- Serializability and Its Discontents
  - access anomalies
**CAP Theorem**

- **Consistency**
  - staleness

- **Availability**
  - will always respond

- **Partition Tolerance**
  - surviving network partitions

*pick any two…*

---

**Consistency**

- eventual consistency
  - almost no constraints
  - if updates stop, values must converge

- causal consistency
  - partial order over operations
  - *local* property

- linearizability
  - every access totally ordered
  - *externalizable*: order agrees with external observation
  - every read returns latest value
Eventual Consistency (EC)

- eventual consistency
  - all replicas will *eventually* converge
- why do we use it?
  - can answer queries locally, immediately
    - very fast
    - partition tolerant
    - not very consistent

Causal Consistency (CC)

- partial order
  - replicas might converge (causal+)
- why do we use it?
  - pretty fast
  - partition tolerant
  - more consistent than EC

**Example:**

- **R1**: x=0
- **R2**: w(x)2, r(x)2
- **R3**: w(y)2, r(y)2
  - R3 not guaranteed to see y=2
  - But if so, x must be “1”
Linearizability (Lin)

- gold standard!
  - all accesses totally ordered
  - no reads are stale
  - external observer agrees with that total order
- why don’t we use it all the time?
  - very slow
    - need to access all replica to get most recent version
  - doesn’t deal well with partitions
    - can’t access all replicas if network down
  - not always available

COPS

- linearizable within cluster
  - each key mapped to specific replica in each cluster
- causal consistency across clusters
  - info pushed asynchronously between clusters
COPS and causal consistency

- reads have dependencies
  - all those write present in local store when read done
  - tracks causal order among writes
  - writes pushed to other clusters in causal order
- how?
  - “nearest”

```
Client 1  put(x,1) → put(y,2) → put(x,3)
          ↓
Client 2  get(y)=2 → put(x,4)
          ↓
Client 3  get(x)=4 → put(z,5)
          ↓
Time
```

COPS nuts and bolts

- dependencies formalized by put_after
COPS nuts and bolts

- "context" implemented incrementally as set of nearest neighbors
- read:
  - return value added to context
- write:
  - context added to write
  - context zero'd out
  - context set to new write
  - propagated with writes to remote clusters

COPS causal not always enough

Alice:
- permissions = "world"
- post1: “all is good”
- permissions -= “boss”
- post2: “I hate my job!”

Alice gets fired…

*TOCTOU* vulnerability (time-of-check-to-time-of-use)
COPS-GT read transactions

- make set of reads
- ensure that we have latest version in dependences

```plaintext
# @param keys list of keys
# @param ctx_id context id
# @return values list of values

function get_trans(keys, ctx_id):
    # Get keys in parallel (first round)
    for k in keys
        results[k] = get_by_version(k, LATEST)
    # Calculate causally correct versions (ccv)
    for k in keys
        ccv[k] = max(ccv[k], results[k].vers)
        for dep in results[k].deps
            if dep in keys
                ccv[dep] = max(ccv[dep], dep.vers)
    # Get needed ccvs in parallel (second round)
    for k in keys
        if ccv[k] > results[k].vers
            results[k] = get_by_version(k, ccv[k])
    # Update the metadata stored in the context
    update_context(results, ctx_id)
    # Return only the values to the client
    return extract_values(results)
```

- Alice keeps job
  - that she hates

COPS

- linearizable in data center
- causal between data centers
- both are good use cases
Serializability

- What is it good for?
  - Huh! Good god, what is it good for - absolutely nothing, say it again ...
  - Isolation between transactions
  - Means we preserve consistency
  - AD
- But!
  - Do external observers agree with this order?

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
<th>T3</th>
</tr>
</thead>
<tbody>
<tr>
<td>start</td>
<td>w(x)1</td>
<td>start</td>
</tr>
<tr>
<td>w(x)2</td>
<td></td>
<td>commit</td>
</tr>
<tr>
<td>commit</td>
<td>start</td>
<td></td>
</tr>
<tr>
<td>r(x)?</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Serializability and anomalies

- The immortal write

History

- Write name: "Danny"
- Write name: "Denny"
- Write name: "Denny"
- Goes back in time

Now

- Writes only see value of final write in serial order: "Danny"

http://dbmsmusings.blogspot.com/
Serializability and anomalies

- the stale read

![Diagram showing serializability and anomalies: the stale read](image)

Serializability and anomalies

- the causal reverse

![Diagram showing serializability and anomalies: the causal reverse](image)

Transfer of $1,000,000 from A to B in two transactions might see $2,000,000 total!
Serializability and anomalies

- avoiding time travel anomalies
- **strict serializability**
  - all guarantees of one-copy serializability (1CS)
  - if \( T_2 \) starts after \( T_1 \) finishes (real time)
    - \( T_2 \) serialized *after* \( T_1 \)

<table>
<thead>
<tr>
<th>System Guarantee</th>
<th>Immortal write</th>
<th>Stale read</th>
<th>Causal reverse</th>
</tr>
</thead>
<tbody>
<tr>
<td>ONE COPY SERIALIZABLE</td>
<td>Possible</td>
<td>Possible</td>
<td>Possible</td>
</tr>
<tr>
<td>STRONG SESSION SERIALIZABLE</td>
<td>Possible (but not within same session)</td>
<td>Possible (but not within same session)</td>
<td>Possible (but not within same session)</td>
</tr>
<tr>
<td>STRONG WRITE SERIALIZABLE</td>
<td>Not Possible</td>
<td>Possible</td>
<td>Not Possible</td>
</tr>
<tr>
<td>STRONG PARTITION SERIALIZABLE</td>
<td>Not Possible</td>
<td>Not Possible</td>
<td>Possible</td>
</tr>
<tr>
<td>STRICT SERIALIZABLE</td>
<td>Not Possible</td>
<td>Not Possible</td>
<td>Not Possible</td>
</tr>
</tbody>
</table>

Serializability and anomalies

- both time travel and isolation level anomalies

<table>
<thead>
<tr>
<th>System Guarantee</th>
<th>Dirty read</th>
<th>Non-repeateable read</th>
<th>Phantom Read</th>
<th>Write Skew</th>
<th>Immortal write</th>
<th>Stale read</th>
<th>Causal reverse</th>
</tr>
</thead>
<tbody>
<tr>
<td>READ UNCOMMITTED</td>
<td>Possible</td>
<td>Possible</td>
<td>Possible</td>
<td>Possible</td>
<td>Possible</td>
<td>Possible</td>
<td>Possible</td>
</tr>
<tr>
<td>READ COMMITTED</td>
<td>Not Possible</td>
<td>Possible</td>
<td>Possible</td>
<td>Possible</td>
<td>Possible</td>
<td>Possible</td>
<td>Possible</td>
</tr>
<tr>
<td>REPEATABLE READ</td>
<td>Not Possible</td>
<td>Not Possible</td>
<td>Possible</td>
<td>Possible</td>
<td>Possible</td>
<td>Possible</td>
<td>Possible</td>
</tr>
<tr>
<td>SNAPSHOT ISOLATION</td>
<td>Not Possible</td>
<td>Not Possible</td>
<td>Not Possible</td>
<td>Possible</td>
<td>Possible</td>
<td>Possible</td>
<td>Possible</td>
</tr>
<tr>
<td>SERIALIZABLE / ONE COPY SERIALIZABLE / STRONG SESSION SERIALIZABLE</td>
<td>Not Possible</td>
<td>Not Possible</td>
<td>Not Possible</td>
<td>Not Possible</td>
<td>Possible</td>
<td>Possible</td>
<td>Possible</td>
</tr>
<tr>
<td>STRONG WRITE SERIALIZABLE</td>
<td>Not Possible</td>
<td>Not Possible</td>
<td>Not Possible</td>
<td>Not Possible</td>
<td>Not Possible</td>
<td>Possible</td>
<td>Not Possible</td>
</tr>
<tr>
<td>STRONG PARTITION SERIALIZABLE</td>
<td>Not Possible</td>
<td>Not Possible</td>
<td>Not Possible</td>
<td>Not Possible</td>
<td>Not Possible</td>
<td>Not Possible</td>
<td>Possible</td>
</tr>
<tr>
<td>STRICT SERIALIZABLE</td>
<td>Not Possible</td>
<td>Not Possible</td>
<td>Not Possible</td>
<td>Not Possible</td>
<td>Not Possible</td>
<td>Not Possible</td>
<td>Not Possible</td>
</tr>
</tbody>
</table>
What’s Spanner?

- **Solution**: Spanner is a scalable, multi-version, globally-distributed, and synchronously-replicated database at Google.

- Distributed multi-version database
  - General-purpose transactions (ACID)
  - SQL query language (extend to support Protobuf)
  - Schematized tables
  - Semi-relational data model (MetaStore)
  - Removes the need to manually partition data

Spanner Organization

- A set of zones (deployment unit)
  - dynamic added or remove
  - also the unit of physical isolation.
- Each Zone
  - Zonemaster (1)
  - Spanserver a*-100-1000
- Location Proxy
  - Help user to locate Spanserver
- Placement Driver
  - Handles data movement across zones
- Universe master
  - Status info for all zones for debugging

**Figure 1**: Spanner server organization.
Concurrency controls

- **2 Phase Locking**
  - As soon as you release the first lock, you can’t obtain any more
  - Growing phase and Shrink phase
- **OCC**
  - Just let the txns run (no lock)
  - Check at end if txns conflict --- "validation"
- **MVCC**
  - snapshot isolation (multiversion)
  - Write doesn’t block read
- **Spanner is designed for long-lived transactions**
  - OCC performs poorly due to confliction

**Key Innovation: True Time**

Spanner knows what time is it
Is Synchronizing Time at the Global Scale Possible?

- synchronizing time within and between datacenters is extremely hard and uncertain

- serialization of requests is impossible at global scale

- Idea: Accept uncertainty, keep it small and quantify (using GPS and Atomic Clocks)
Time References

GPS: The GPS receivers, much like the one in your cell phone, grab the time from various satellites orbiting the globe.

Atomic clocks: keep their own time

Clock skew averages 4ms.

TrueTime

- TrueTime represents time as a TTinterval: [earliest, latest]
- A TTinterval with bounded time uncertainty
- A TTinterval is guaranteed to contain the absolute time during which TT.now() was invoked.

\[ tt.earliest \leq t_{abs}(e_{now}) \leq tt.latest \]

- Novel TrueTime APIs distribute a globally synchronized time
How TrueTime Is Implemented?

- The underlying time references used by TrueTime are **GPS and atomic clocks**
- a set of time master machines **per datacenter**
  - Antennas masters (GPS)
  - Armageddon masters (atomic clock)
- a timeslave daemon **per machine**
  - polls a variety of masters to reduce vulnerability to errors from any one master.

Timestamps, Global Clock

- **Strict two-phase locking for write transactions**
- **Assign timestamp while locks are held**

```
T
```

Acquired locks

Pick s = now()

Release locks
Timestamp Invariants

- Timestamp order == commit order
  - $T_1$
  - $T_2$
- Timestamp order respects global wall-time order
  - $T_1$
  - $T_2$

Timestamps and TrueTime

- Acquired Jocks
- Release locks
- $T$
- $s = \text{TT.now().latest}$
- $s = \text{TT.now().earliest} > s$
- Commit wait
  - average $\xi$
  - average $\xi$
Distributed Systems == Good

- **Good in:**
  - underlies most all big system
  - continuing to grow
  - fun

- **Anomalies happen in the distributed world**
  - most systems do not handle well
  - a few do (Spanner, SLOG (abadi)), maybe Cockroachdb
  - many opportunities to do interesting work in this area