Distributed Distributed Systems

L17: Distributed Transactions
.... Applications......
Why does each KV have a VC? How does a RM determine sufficient Dependency exist to update?

Set(k, v, ts) function
Returns (ts)

Get(k, ts) function
Returns (v, ts)
• Given a "Partition", you must pick between "Availability" and "Consistency"
  • Pick Consistently: Some clients (not all) can change "data consistently"
  • Pick Availability: All clients can change data but “inconsistently”

• C: Consistency (Linearizable)
• A: Availability
• P: Partition tolerance
Distributed Transactions
How do you Change Cluster Size?

Goal: add two new servers to cluster

Challenge: consistently get all servers to agree on new cluster size

Case: During cluster update, leader fails AND different servers have different notions of size → multiple leaders
How do you Change Cluster Size?

Goal: add two new servers to cluster

Challenge: consistently get all servers to agree on new cluster size

Case: During cluster update, leader fails AND different servers have different notions of size → multiple leaders

Solution: Need a protocol to consistently update the cluster
Introduction

A transaction in Cloud Spanner is a set of reads and writes that execute atomically at a single logical point in time across columns, rows, and tables in a database.

Cloud Spanner supports these transaction modes:

- **Locking read-write.** This type of transaction is the only transaction type that supports writing data into Cloud Spanner. These transactions rely on pessimistic locking and, if necessary, two-phase commit. Locking read-write transactions may abort, requiring the application to retry.

- **Read-only.** This transaction type provides guaranteed consistency across several reads, but does not allow writes. Read-only transactions can be configured to read at timestamps in the past. Read-only transactions do not need to be committed and do not take locks.

https://cloud.google.com/spanner/docs/transactions
Optimistic or pessimistic locking – Which one should you pick?

Don Pinto, Principal Product Manager, Couchbase on December 16, 2014

https://blog.couchbase.com/optimistic-or-pessimistic-locking-which-one-should-you-pick/
• Replication
  • Lazy, Passive, Active
  • Consistency for a single shard

• Distributed Transaction
  • Consistent/Atomic change to data in multiple shards
  • Multiple shards ➔ Can use traditional replication techniques

Clients send requests To all replicas
What is Transaction?

- A set of operations that need to be performed together.
  - Example 1: transferring money between accounts
  - Example 2: shopping cart checkout

Initially Theo and Rodrigo have $100.
Goal: Transfer $50 from Rodrigo to Theo.

Read(R)
Update(R, $50)
Read(T)
Update(T, $150)
What is Transaction?

- A set of operations that need to be performed together.
  - Example 1: transferring money between accounts
  - Example 2: shopping cart checkout

Initially Theo and Rodrigo have $100.
Goal: Transfer $50 from Rodrigo to Theo.
Ideal: either the whole 4 operations happen or none happen
Worst case: only a subset occur

<table>
<thead>
<tr>
<th>Operation</th>
<th>Node</th>
<th>Shard</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read(R)</td>
<td>Node A</td>
<td>Shard 1</td>
</tr>
<tr>
<td>Update(R, $50)</td>
<td>Node B</td>
<td>Shard 1</td>
</tr>
<tr>
<td>Read(T)</td>
<td>Node C</td>
<td>Shard 1</td>
</tr>
<tr>
<td>Update(T, $150)</td>
<td>Node D</td>
<td>Shard 2</td>
</tr>
<tr>
<td></td>
<td>Node E</td>
<td>Shard 2</td>
</tr>
<tr>
<td></td>
<td>Node F</td>
<td>Shard 2</td>
</tr>
</tbody>
</table>

Rodrigo is in shard 1
Theo is in shard 2
Read(R)
Update(R, R-$50)
Read(T)
Update(T, T+$50)

Read(T)
Update(T, T-$150)
Read(K)
Update(K, K+$150)
Transaction Background

**ACID Semantics**
- Atomicity
- Consistency
- Isolation
- Durability

- All or nothing semantics: all operations succeeds or fails.
- Transitions from one consistent state to another consistent state
- Intermediate states are not exposed to the outside world (no partial writes are exposed)
- Results of a ‘committed’ transactions persists after the transaction (and through failures)

**Transactions are easy for traditional databases**
- Traditional databases are on a single server → failure is “all-or-nothing”
  - All components of the transactions fail
- Distributed transactions → different components can fail
  - Need to provides Transaction semantics when any subset of the components can fail
Distributed Transactions Semantics

- Read(R)
- Update(R, R-$50)
- Read(T)
- Update(T, T+$50)
Distributed Transactions Semantics

- **Transaction Manager**
  - Server in charge of orchestrating the transaction

- **Steps for transaction**
  - Client initiates a transaction
    - TM gives client a Transaction ID (TID)
  - Client submits operations to TM
    - TM relays operations to replicas
  - Client commits transaction
    - TM performs two phase commit
Distributed Transactions Semantics

- **Transaction Manager**
  - Server in charge of orchestrating the transaction

- **Steps for transaction**
  - Client initiates a transaction
    - TM gives client a Transaction ID (TID)
  - Client submits operations on TID
    - TM relays operations to replicas
  - Client commits
    - TM performs two-phase commit

**Client-Side Code:**

```python
Client-Side Code:
tid = openTransaction();
RVal = a.get(tid, Rodrigo);
a.update(tid, Rodrigo, RVal - 50);
Tval = b.get(tid, Theo);
b.update(tid, Theo, Tval + 50);
closeTransaction(tid) or abortTransaction(tid)
```

**Replica Manager**

- Prepare operations
- Store operations locally in log
- **But do not commit** operations

**Transaction Manager**

- TM hands out TIDs
- TM manages and relays operations to Replica Leaders
- TM keeps track of all Replicas involved in the transactions
One Phase Commit
• On client commit
  • T.M. retries until success

• Why is this bad?
Two Phase Commit
Two Phase Commit

• Provides Atomicity and Consistency
  • NOT Isolation and Durability

• Assumptions: each server maintains a transaction log
  • Transaction log is stored in persistent memory
  • If failure → items in Transaction Log survives

• Terminology changes:
  • Coordinator ← The transaction Manager
  • Participant ↔ Leader of a replica
Two Phase Commit

• Phase 1:
  • Coordinator sends request for votes
  • Participants vote

• Phase 2:
  • Coordinator counts votes
  • Coordinator informs of transaction status

• At the end of Phase 2: either all participants commit or all abort
State Diagrams for Two Phase Commit

Coordinator

Init
- app commit \rightarrow req votes
- all commit \rightarrow commit

Wait
- any abort \rightarrow abort
- commit \rightarrow commit

Abort

Commit

Participant

Init
- vote req/commit
- vote req/abort

Uncertain
- abort/ack
- commit/ack

Abort

Commit

Coordinator

Ready?

Vote: Commit or Abort

Response
Two Phase Commit

- Phase 1:
  - Coordinator sends request for votes
  - Participants vote

- Phase 2:
  - Coordinator counts votes
  - Coordinator informs of transaction status

At the end of Phase 2: either all participants commit or all abort.
Two Phase Commit With Failures

• What is the impact of failures on 2PC?

• 2PC is synchronous
  • Failure == node failure or network failure
  • Failure --> the protocol blocks/stalls

Coordinator fails, participants will be uncertain (waiting)

Participant fails, coordinator will be waiting
What is the Impact of Failures?

Abort
Commit
Wait
Init

Abort
Commit

Vote: Commit or Abort
Ready?
Response

Coordinator
Participants

Commit/ack
Abort/ack
vote req/abort
vote req/commit
init

app commit → req votes
any abort → abort
all commit → commit
Crash Points

Coordinator

Init

app commit/vote req

Wait

any abort/abort

Abort

Commit

Commit/commit

Participant

Init

ginition

vote req/commit

Uncertain

vote req/abort

Abort

Commit

Abort/ack

Commit/ack

Coordinator fails, participants will be uncertain (waiting)
Participant fails, coordinator will be waiting