Distributed Systems

Day 13: Distributed Transaction

“To Be or Not to Be... Distributed .. Transactions”
Summary

• Background on Transactions
  • ACID Semantics

• Distribute Transactions
  • Terminology: Transaction manager, Coordinator, Participant
  • Two Phase Commit
    • Adding Isolation with Locks: optimistic V. pessimistic
    • Performance Issues
  • Consistency Models
    • Serializability Versus Linearizability
Transactions

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
• 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

Partition data into shards, maps shards to server with consistent hashing
Maintain multiple copies for fault tolerance and to reduce latency
Clients send requests to all replicas

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

Read(R)
Update(R, $50)
Read(T)
Update(T, $150)

Rodrigo is in shard 1
Theo is in shard 2
Transaction Background

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

- **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 only a subset of the components fail

<table>
<thead>
<tr>
<th>All or nothing semantics: all operations succeeds or fails.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Transitions from one consistent state to another consistent state</td>
</tr>
<tr>
<td>Intermediate states are not exposed to the outside world (no partial writes are exposed)</td>
</tr>
<tr>
<td>Results of a ‘committed’ transactions persists after the transaction (and through failures)</td>
</tr>
</tbody>
</table>
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 to TM
    • TM relays operations to replicas
  • Client commits transaction
    • TM performs two phase commit

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

**Transaction Manager**
- TM hands out TIDs
- TM manages and relays operations to Replica Leaders
- TM keeps track of all Replicas involved in the transactions

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

Node A — Node D
Shard 1 — Shard 2
Node C — Node F
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
Coordinator State Diagram for Two-Phase Commit

Diagram:

- **Init**
  - app commit/vote req
- **Wait**
  - any abort/abort
  - all commit/commit
- **Abort**
- **Commit**

Coordinator

- **Ready?**
- **Vote: Commit or Abort**
- **Response**

Participants
Participant State Diagram for Two-Phase Commit

Coordinator

Participant

Make a change
Commit or Abort
Response

Init
Uncertain
Abort
Commit

vote req/commit
vote req/abort
abort/ack
commit/ack
State Diagrams for Two Phase Commit

Coordinator

Participant
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.

BRACE YOURSELF

FAILURES ARE COMING
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
Crash Points

Coordinator

Init

Wait

Abort

Commit

app commit/vote req

any abort/abort

all commit/commit

Participant

Init

Uncertain

Abort

Commit

vote req/abort

vote req/commit

abort/ack

commit/ack
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/octals

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

• Detect failure using Timeouts
  • Is this model Synchronous or Asynchronous?
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

• Detect failure using Timeouts
  • Coordinator detects participant failure and assumes ABORT \(\rightarrow\) Transaction terminates
  • Participant detects Coordinator failure
    • Why Can’t Participant automatically ABORT?
Participant Recovery from Coordinator Failure

- Participant in Uncertain state
  - waiting for coordinator to say “commit” or “abort”

- It detects failure of coordinator
  - Using timeout

- Participant in Uncertain state
  - can’t assume either outcome
    - BAD things happen if participant makes wrong assumptions
  - waits for coordinator to restart
    - On restart contact coordinator for final outcome
Participant Recovery from Coordinator Failure

- Participant in Uncertain state
  - waiting for coordinator to say “commit” or “abort”

- It detects failure of coordinator
  - Using timeout

- Participant in Uncertain state
  - can’t assume either outcome
    - BAD things happen if participant makes wrong assumptions
  - waits for coordinator to restart
    - On restart contact coordinator for final outcome

---

**AS COORDINATOR, I DON’T ALWAYS FAIL**

**BUT WHEN I FAIL, I BLOCK THE TRANSACTION**
## Coordinator Recovery from Participant Failure

<table>
<thead>
<tr>
<th>Diagram</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Init</td>
<td>ap commit/vote req</td>
</tr>
<tr>
<td>Wait</td>
<td>any abort/abort all commit/commit</td>
</tr>
<tr>
<td>Abort</td>
<td></td>
</tr>
<tr>
<td>Commit</td>
<td></td>
</tr>
</tbody>
</table>

- Coordinator in wait state
  - waiting for participant to say “commit” or “abort”

- It detects failure of participant
  - Using timeout

- Coordinator assumes they said ‘no’
  - Takes no response as an abort
  - Abort transaction!!

- If participant Fails, Coordinator can make progress
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

- Detect failure using Timeouts
  - Coordinator detects participant failure and assumes ABORT \(\rightarrow\) transaction terminates
  - Participant detects Coordinator failure
    - The participant must wait for coordinator
    - The transaction is stalled!

Coordinator fails, participants will be uncertain (waiting)
Participant fails, coordinator will be waiting
Two Phase Commit and CAP Theorem!

- During a partition
  - Does 2PC pick Availability or Consistency?

Diagram:

- Coordinator
- Participant
- Network Partition
Theorem

- **Consistency (Linearizable)**
- **Availability**
- **Partition tolerance**

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
Two Phase Commit and CAP Theorem!

- During a partition
  - Does 2PC pick Availability or Consistency?
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

• Detect failure using Timeouts
  • Coordinator detects participant assumes ABORT
  • Participant detects Coordinator failure
    • Why Can’t Participant automatically ABORT?

Coordinator fails, participants will be waiting
Participant fails, coordinator will be waiting
Two Phase Commit With Failures

• What is the impact of failures on 2PC?

• 2PC is synchronous
  - Failure: node failure or network failure
  - Failure blocks/stalls
  - Failure → protocol blocks/stalls

  - Failure:
    - Fails, participants will be waiting
    - Fails, coordinator will be waiting

Using Timeouts
  - Detects participant failure
    - Participant aborts
  - Detects Coordinator failure
    - Participant automatically aborts?

Participant ABORT
  - Eventually transaction ends

Ready to Commit?
  - Participant

Vote [Yes/No]
  - Participant

Count Votes!!!!!!
  - Participant

Abort/Commit?
  - Participant

Participant that voted NO can abort
  - However, Voted yes can not ABORT

Transaction ends

Participant that voted NO can abort
  - However, Voted yes can not ABORT

Vote [Yes/No]
  - Participant

Count Votes!!!!!!
  - Participant
Two Phase Commit: Adding Isolation with Locks
RocksDB supports Transactions when using a TransactionDB or OptimisticTransactionDB. Transactions have a simple BEGIN/COMMIT/ROLLBACK api and allow applications to modify their data concurrently while letting RocksDB handle the conflict checking. RocksDB supports both pessimistic and optimistic concurrency control.

https://github.com/facebook/rocksdb/wiki/Transactions
Concurrency Modes and Isolation Levels

Whenever `TRANSACTIONAL` atomicity mode is configured, Ignite supports `OPTIMISTIC` and `PESSIMISTIC` concurrency modes for transactions. Concurrency level determines when an entry-level transaction lock should be acquired - at the time of data access or during the `prepare` phase. Locking prevents concurrent access to an object. For example, when you attempt to update a ToDo list item with pessimistic locking, the server places a lock on the object until you either commit or rollback the transaction so that no other transaction or operation is allowed to update the same entry. Regardless of the concurrency level used in a transaction, there exists a moment in time when all entries enlisted in the transaction are locked before the commit.

https://apacheignite.readme.io/docs/concurrency-modes-and-isolation-levels
Pessimistic Versus Optimistic Locking

• **Trade-off: concurrency versus isolation**

**Pessimistic:**
- Get all locks before transaction
- Release all locks after transaction
  - Release locks after commit/abort
- Prevents anyone else from using the data during transaction
  - Locks prevent read/write of data
  - Locks stop other transactions

**Optimistic**
- No locks
- Get a copy of data before transaction
- After transaction check to make sure data has not changed
  - If the data changed then ABORT!!!!
  - Data changes means someone else changed the data
Pessimistic Versus Optimistic Locking

- Trade-off: concurrency versus isolation

**Pessimistic**
- Low level of concurrency
- Low performance
- Sequential ordering of transactions

**Optimistic**
- High level of concurrency
- High throughput: especially if all reads
- Many Transactions will abort if many writes
Two Phase Commit: Practical Issues
Practical Performance Issues with 2PC

• Synchronization: 2PC Overheads
  • Multiple "rounds" of communication
  • Three rounds of communication
    • $3(N)$ messages
  • During these rounds resources are frozen
Practical Performance Issues with 2PC

• Blocking: 2PC
  • During failure → 2PC can block
  • When 2PC blocks → then other transactions are unable to progress
Practical Performance Issues with 2PC

• Synchronization: 2PC Overheads
  • Multiple "rounds" of communication
  • Three rounds of communication
    • 3(N) messages
  • During these rounds resources are frozen

• Blocking: 2PC
  • During failure → 2PC can block
  • When 2PC blocks → then other transactions are unable to progress

Distributed Transaction No Longer Considered Dead!

2007: Avoid Distributed transactions

2012: Google’s Spanner Dist. Transaction!!!

2019: MS’s Orleans Dist. Transaction to the Cloud!!!

https://thenewstack.io/microsoft-orleans-brings-distributed-transactions-to-cloud/
Distributed Transactions and ACID
How do you get ACID in Distributed Transactions?

• Two Phase Commit $\rightarrow$ A + C
  • 2PC: atomic and consistent change from one state to another state

• Two Phase Commit + Locks $\rightarrow$ A + C + I
  • Locks provide isolation by preventing concurrent transaction from accessing data

• Two Phase Commit + Locks + Logs $\rightarrow$ A + C + I + D
  • Logs ensure the transactions persists
Consistency Models Revisited
Consistency Spectrum

- **Strong Consistency**: Faster but harder to program
- **Weak Consistency**: Slower but easier to program

### Levels of Consistency
- **Strict Serializability**
- **Sequential Consistency**
- **Linearizability**
- **Causal Consistency**
- **Eventual Consistency**
Strict Serializability

• Total order + FIFO + “Time” → for a transaction
  • After a transaction commits, all future reads will see committed data

• Requires 2PC + pessimistic Locks
  • Low performance: reads/write have high latency and low throughput
Strict Serializability

- Total order + FIFO + "Time" -> for a transaction
  - After a transaction commits, all future reads will see committed data

- Requires 2PC + pessimistic Locks
  - Low performance: reads/write have high latency and low throughput

Strict Serializability V. Linearizability

- Strict Serializability = Linearizability for Transactions
- Linearizability = Total order + FIFO + Real time for individual operations
- Strict Serializability = Total order + FIFO + Real time for transactions (groups of operations)
Summary

• Background on Transactions
  • ACID Semantics

• Distribute Transactions
  • Terminology: Transaction manager, Coordinator, Participant
  • Two Phase Commit
    • Adding Isolation with Locks: optimistic v. pessimistic
    • Performance Issues
  • Consistency Models
    • Serializability Versus Linearizability