Distributed databases: Spanner (Google) and Aurora (AWS)

Jinyang Li
What have we learnt last time

• ACID transactions
  – Atomicity (against crashes)
  – Isolation (semantics in the face of concurrency)

• Techniques for achieving A and I
  – A: WAL logging (REDO logging)
  – I: 2 phase locking for serializablity

• Distributed transaction commit
  – 2-phase commit
  – Not fault tolerant
Spanner’s goal

- A global, scalable database
  - General-purpose transactions (ACID)
  - SQL query language
  - Schematized tables
  - Semi-relational data model
- Global: data is replicated in multiple data centers
- Scalable: increase transaction read/write throughput by adding more machines
  - no more sharded mysql
Spanner’s setup: example social networking app

User posts
Friend lists

US
San Francisco
Seattle
Arizona
x1000

Brazil
Sao Paulo
Santiago
Buenos Aires
x1000

Spain
San Francisco
London
Paris
Berlin
Madrid
Lisbon
x1000

Russia
Moscow
Berlin
Krakow
x1000

OSDI 2012 (Spanner talk)
How to support distributed transactions?

A strawman

• Starting point: A simple impl. of local txs

```go
struct DB {
    data sync.Map[string]string
    locks sync.Map[string]LockStatus
}
struct Tx {
    tid int64
    writeset map[string]string
    readset map[string]interface{}
}

func (s *DB) StartTx() *Tx {
    return &Tx{rand.Int63()}
}
func (s *DB) Read(tx *Tx) {
}
func (s *DB) Write(tx *Tx) {
}
func (s *DB) CommitTx(tx *Tx) {
}
```
How to support distributed transactions?

A strawman

- Starting point: A simple impl. of local txs

```go
func (s *DB) Read(tx *Tx, key string) {
    while s.locks.Load(key) == WLOCKED {
        Wait() // deadlock detection needed
    }
    s.locks.Store(key, RLOCKED)
    tx.readset[key] = interface{}
    return s.data[key]
}

func (s *DB) Write(tx *Tx, key, value string) {
    tx.writeSet[key] = value
}

func (s *DB) CommitTx(tx *Tx) {
    for k, v := range tx.writeSet {
        while s.locks.Load(key) != IDLE {
            Wait() // deadlock detection needed
        }
        s.locks.Store(key, WLOCKED)
    }
    s.AppendToLog(tx, “Committed”)//
    for k, v := range tx.readSet {
        s.locks.Store(k, IDLE)
    }
    for k, v := range tx.writeSet {
        s.data.Store(k, v)
        s.locks.Store(k, IDLE)
    }
}
```

* use mutex to protect the atomicity of code block marked by []
Extending local transactions to distributed setting

Example Transaction: \( r = \text{Read}(X); \text{Write}(Y, r) \)

```
func (s *DB) Read(tx *Tx, key string) (value string, err error) {
    while s.locks.Load(key) == WLOCKED {
        return "", TRY_AGAIN_LATER
    }
    s.locks.Store(key, RLOCKED)
    return s.data.Load(key), nil
}
```

\( \text{tx.readset}[\text{key}]=\text{interface{}} \)

Server-X

Server-Y

\( \text{tx.tid} = 157 \)
Extending local transactions to distributed setting

Example Transaction: \( r = \text{Read}(X); \text{Write}(Y, r) \)
Extending local transactions to distributed setting

Example Transaction: \( r = \text{Read}(X); \text{Write}(Y, r) \)
Extending local transactions to distributed setting

Example Transaction: \( r = \text{Read}(X); \text{Write}(Y, r) \)
What can go wrong?

- Client fails after acquiring read locks
- Spanner’s solution:
  - Client sends keep-alive to servers holding read locks
  - Server times out held read locks
  - 2PC-prepare checks that read locks are still held before voting yes
What can go wrong?

• What if 2PC encounters irrecoverable failure?
• Spanner’s solution:
  – replicate both data and critical 2PC state using Paxos
2PC with Paxos

server-X (leader)
server-X'
server-Y (leader)
server-Y'

2PC-prepare
Paxos Accept
yes
Paxos Accept
yes
Paxos Accept

commit

2PC-participant logs yes-vote via Paxos
2PC-coordinator logs commit record via Paxos
2PC-participants logs commit record via Paxos
Other details

• Must the lock table be replicated via Paxos?
• Spanner’s solution:
  – No, lock table is only maintained by (Paxos) leader
  – Upon leader change
    • read locks are discarded (safe because 2PC-prepare checks for validity of read locks)
    • write locks are recovered (by scanning Paxos log for 2PC-prepare messages without corresponding 2PC-commit)
Spanner’s biggest innovation

• Strictly serializable read-only transactions
• Why read-only transactions?
  – Many application workloads are read-heavy
  – Facebook’s read vs. write ratio is 30:1 [SIGMETRICS’12]
• Our naive implementation is inefficient
  – read-only txs block read-write transactions
  – esp. bad if read-only txs read lots of objects
How to implement efficient read-only transactions

• Write-txs do not overwrite data, but write new versions of data
• Read-txs determine which versions of data to read using a timestamp
Example MVCC (local implementation)

```go
struct RWTx {
    tid int64
    writeset map[string]string
    readset map[string]interface{}
}

func (s *DB) StartRWTx() *RWTx {
    // same as before
}
func (s *DB) ReadRWTx(tx *RWTx) {
    // same as before
}
func (s *DB) WriteRWTx(tx *RWTx) {
    // same as before
}
func (s *DB) CommitRWTx(tx *RWTx) {
    //
}
```

```go
struct ROTx{
    readTS time.Time
}

func (s *DB) StartROTx() *ROTx {
}
func (s *DB) ReadROTx(tx *ROTx) {
    // same as before
}
```
Local MVCC (RW transactions)

```go
func (s *DB) CommitRWTx(tx *RWTx) {
    for k, v := range tx.writeset {
        while s.locks.Load(key) != IDLE {
            Wait()
        }
        s.locks.Store(key, WLOCKED)
    }
    s.AppendToLog(tx, commitTS, "Committed")
    for k, v := range tx.readset {
        s.locks.Store(k, IDLE)
    }
    for k, v := range tx.writeset {
        s.data[k] = append(s.data[k], Record{commitTS, v})
        s.locks[k] = IDLE
    }
}
```

commitTS := time.Now()

commitTS represents the serialization order
Local MVCC (RO transactions)

```go
func (s *DB) StartROTx() *ROTx {
    return &ROTx{time.Now()}
}
```

```go
func (s *DB) ReadROTx(tx *ROTx, key string) string {
    items := s.data.Load(key) // items is an array of data from old to new
    // read the largest version smaller than tx.readTS
    for i := len(items)-1; i >= 0; i-- {
        if items[i].commitTS < tx.readTS {
            return items[i].data
        }
    }
    return ""
}
```

T1: R(X)=0 R(Y)=0
T2: W(X)=1 W(Y)=1

T1 blocks T2 under 2PL but not under MVCC
Local MVCC (RO transactions)

```go
func (s *DB) StartROTx() *ROTx {
    return &ROTx{time.Now()}
}

func (s *DB) ReadROTx(tx *ROTx, key string) string {
    for {
        items, lockStatus := atomically retrieve items and lockStatus for the key
        if lockStatus == IDLE { break }
    }
    for i := len(items)-1; i >= 0; i-- {
        if items[i].commitTS < tx.readTS {
            return items[i].data
        }
    }
    return ""
}
```

Problem: Not all transactions less than readTS has finished writing.

```
lock X, lock Y, commitTS=1, W(X)=1, unlock X, W(Y)=1
readTS=2, R(X)=1, R(Y)=0??
```
Extending MVCC to distributed setting

• Timestamp requirement for strict serializability:
  – Timestamp for RW transactions obey the serialization order
  – Timestamp of a RO transaction must be larger than any committed RW transactions
Extending MVCC to distributed setting

• OK if there’s a centralized timestamp server

Omitted arrows: Paxos writes to persist “yes-vote”s and commit record of 2PC coordinator (X) and participant (Y)
What’s wrong if using clocks from different servers?

Problem: T2 starts after T1 finishes, but T2.readTS might be smaller than T1.commitTS due to clock difference
Why doesn’t Spanner use a central timestamp server?

• Central performance bottleneck?
• Added Latency
  – A server in european datacenter need to contact the timestamp server in US data center
Spanner’s time primitive: TrueTime

- **Core API:** `tt.now()`
  - It returns an error bound `[earliest, latest]`, so that `earliest < t_abs (true time) < latest`
  - Different servers have different error bounds
- **Auxiliary APIs:** `tt.After(t)`
  - `tt.After(t)` returns true if `t` is after true time
- **TrueTime** is implemented by synchronizing with time masters (GPS clock & atomic clock)
How RW transactions use TrueTime

• Coordinator sends 2PC-prepare:
  – each participant server attaches its tt.Now().latest

• If all vote yes, coordinator chooses commitTS as:
  – max(timestamps of 2PC-prepare replies, tt.Now().latest)

• **Spanner-commit-wait** → Coordinator waits till tt.After(commitTS) is true

• Coordinator logs commit status, then sends 2PC-commit
How RO transactions use TrueTime

• A RO transaction’s readTS is:
  – readTS = tt.Now().latest

• Why is this correct?
  – Prove: if T2 (ro) starts after T1 (rw) finishes, then
    T2.readTS > T1.commitTS

  T2.readTS > T2’s actual start time

  T2’s actual start time > T1’s actual finish time

  T1’s actual finish time > T1.commitTS

  because of tt.Now()’s error bound guarantee

  because of coordinator performs commit-wait
Spanner contains other optimizations

- Allow reads by RO transaction to be done at any replica, not just the Paxos leader
- Avoid blocking reads for single-read RO transaction
Spanner’s performance

• Setup: machines are within 1ms RTTs
• Best-case RW transaction latency (a single 2PC participant)
  – mean: 17ms, 99-th: 75ms
• Commit-wait latency: 5ms, Paxos latency: 9ms
Amazon’s Aurora

• Takes a very different approach than Spanner
• No-goals:
  – No infinite scalability → No distributed concurrency control
• Yes-goals:
  – High availability in the face of machine crash
  – High performance
Aurora’s motivation: Cloud customers running MySQL over EBS

Step 1, 3, 4 are sequential and synchronous (step 3: block-level software mirroring)
Performance negatives of MySQL over EBS

• Steps (1, 3, 4) are sequential and synchronous:
  – amplifies latency and jitter
• Many writes per user operation: e.g. double writes
  – Write-only workload benchmark
  – 7.4 I/Os per transactions
Aurora’s high level design

• Goal:
  – Do fewer IOs, less network traffic
  – minimize synchronization points

• The main insight:
  – REDO log captures db state
  – replicate the REDO log
    • offload REDO processing to storage

• Replicate REDO log using Raft/Paxos?
  – unnecessary: Aurora has a single writer
  – insufficient performance: log should be striped to for better performance
Aurora’s design

- Only log records are sent.
- Asynchronously replicated to 4 out of 6 replicas.
Replicas across Availability Zone

• 6 replicas across 3 availability zones
  – write-quorum = 4, survives either of the two failure scenarios below:
    • an entire AZ is down
    • 2 replicas are down
  – read-quorum = 3, survives either of the two failure scenarios below:
    • an entire AZ + another replica down
    • 3 replicas down
Segmented storage

• Database volume is partitioned into 10GB segments
  – 10TB db → 1000 segments

• Each segment → a potentially different replica set (Protection Group)

• Why segmented storage?
  – spread storage and processing over more nodes
  – faster recovery time
Segmented storage and REDO log replication

database (aka primary instance)

<table>
<thead>
<tr>
<th>LSN</th>
<th>PageNo</th>
<th>Action</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10...</td>
<td>insert record</td>
<td>555</td>
</tr>
<tr>
<td>2</td>
<td>33...</td>
<td>insert record</td>
<td>123</td>
</tr>
<tr>
<td>3</td>
<td>56...</td>
<td>delete record</td>
<td>10</td>
</tr>
</tbody>
</table>
Storage node processing

Step 1: Add received log to in-memory queue.

Step 2: Persist queue to disk and acknowledge.

Step 3: Identify gaps in log to disk and acknowledge (each record contains LSN of previous record for that PG).

Step 4: Gossip with peers to fill in gaps.

Step 5: Coalesce log records to new pages.

Step 6: Point in time snapshot.

Step 7: Garbage collection.

Step 8: Scrub and backup.
Aurora’s failure recovery

• Primary instance crashes, external service promotes another to be the new primary
• New primary needs to establish a consistent state
  – old primary did not complete write-quorum for some records → log may have holes
  – Only the last log record of a (mini)transaction represent consistent state
Aurora failure recovery

• New primary reads from a quorum in every PG to calculate VDL (volume durable LSN)
  – the highest consistent LSN below which all records have been received

• New primary issues a truncation request with range [VDL+1, VDL+Threshold]
  – all storage nodes discard log records in the range

• New primary performs UNDO recovery to unwind partial transactions

• storage nodes replay log on demand when handling read requests
Summary

• Distributed transactions in Spanner
  – 2PL + 2PC with Paxos for fault tolerance
  – use MVCC for efficient read-only transaction
  – Truetime ensures strict serializability without centralized timestamp server

• Distributed transactions are scalable, but not efficient

• Aurora has a single database writer, using striped replicated storage for REDO processing
Midterm statistics

Histogram of Midterm

Midterm