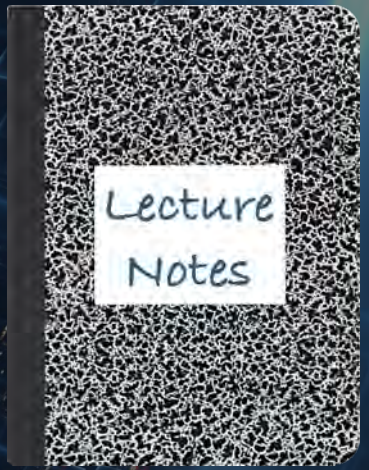
CS 417 – DISTRIBUTED SYSTEMS

# Week 9: Distributed Databases Part 2: Spanner




Paul Krzyzanowski

# Google Spanner

**Spanner uses:**

- Bigtable
- Paxos
- Two-phase commit
- Strict two-phase locking
- MVCC
- Clock synchronization

# Spanner

**Take Bigtable and add:**

- Familiar SQL-like multi-table, row-column data model
  - One primary key per table
- Synchronous replication (Bigtable was eventually consistent)
- Transactions across arbitrary rows

**Spanner**

- **Globally distributed multi-version database**
- ACID (general purpose transactions)
- Schematized tables (Semi-relational)
  - Built on top of a key-value based implementation
  - SQL-like queries
- Lock-free distributed read transactions

**Goal: make it easy for programmers to use !**

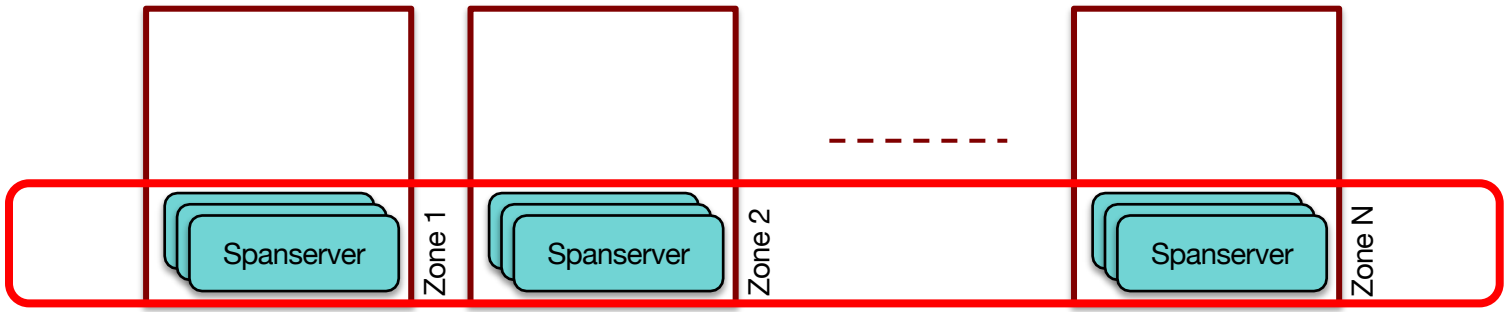
(Without having to deal with consistency models, locking, etc.)

# Spanner Components

- Tables sharded across rows into *splits*
- Splits stored in spanservers
- 1000s of spanservers per zone

Splits are what Bigtable called tablets

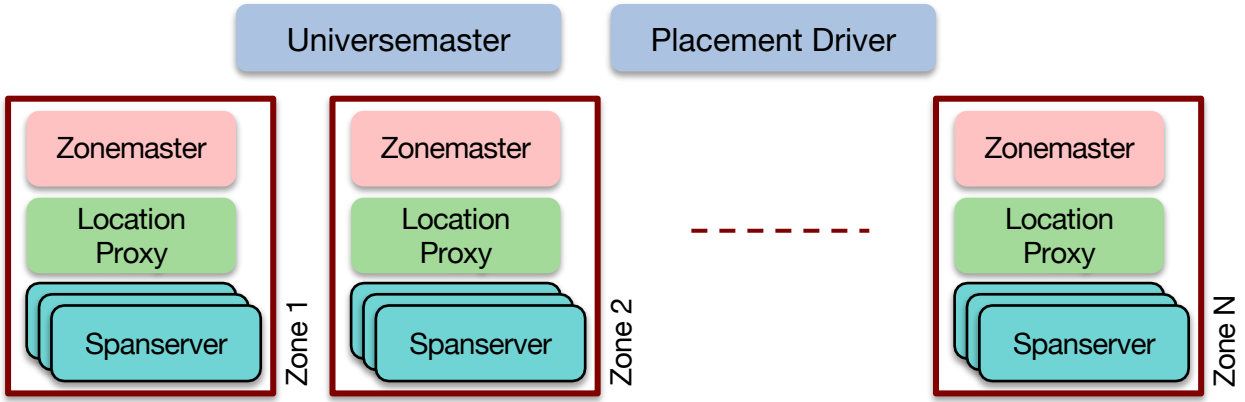
**Sharding:**
The general concept of breaking big data into smaller chunks



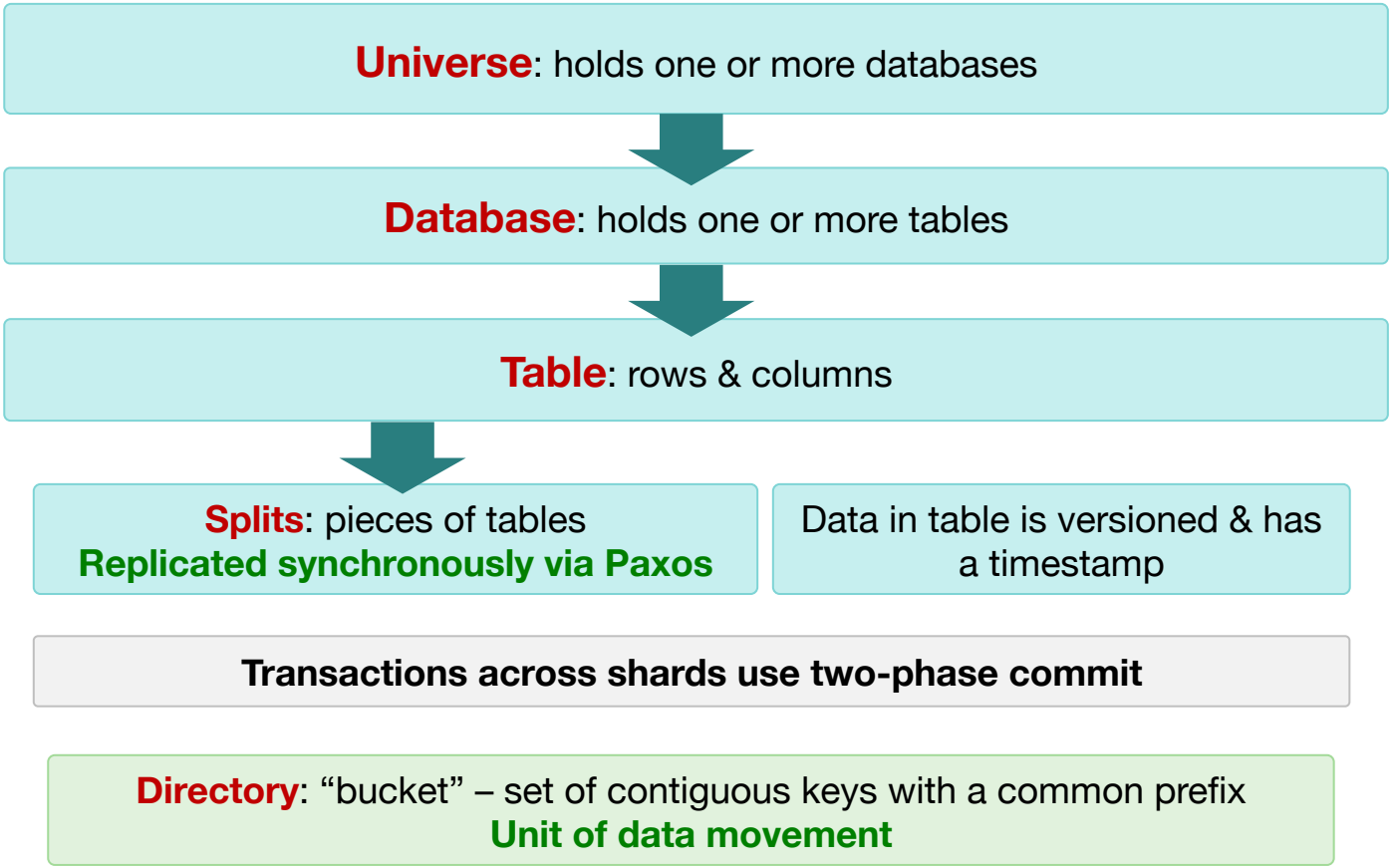
# Spanner Components

- Tables sharded across rows into *splits*
- Splits stored in spanservers
- 1000s of spanservers per zone

- Zonemaster
- Location proxies
- Universemaster
- Placement driver



# Data Storage



# Transactions

Provide ACID properties

- **Two-phase commit protocol** is used **outside** of a group of replicas
- Transactions are serialized: **strict 2-phase locking**

1. Acquire all locks
   – *do work (acquiring locks as needed)* –
2. **Get a commit timestamp**
3. Log the commit timestamp via Paxos consensus to the majority of replicas
4. Do the commit
   - Apply changes locally & to replicas
5. Release locks

# Even 2-Phase locking can be slow

## Read-write transactions

Spanner uses *read locks* and *write locks*

- *read locks* block behind *write locks*
- *write locks* block behind *read locks*

read/write transactions ⇒ wound-wait concurrency control

# 2-Phase locking can be slow

Multiversion concurrency (MVCC) for **snapshot reads**

- Snapshot of the database at a point in time
- Read old data without getting a lock
- Results are consistent
- Easy to run huge transactions that read data from a point in time

We need ***commit timestamps*** that will enable meaningful snapshots

# Getting good commit timestamps

- **Spanner: use physical timestamps**
  - If $T_1$ commits before $T_2$ then $T_1$ *must* get a smaller timestamp
  - Commit order matches global wall-time order

# TrueTime

- **Global wall-clock time** = time + interval of uncertainty
  - `TT.now().earliest` = time guaranteed to be <= current time
  - `TT.now().latest` = time guaranteed to be >= current time

Each data center has a
GPS receiver & atomic clock

# Commit Wait

We don’t know the *exact* time … but we can wait out the uncertainty

1. Acquire all locks

   – *do work (acquire locks as needed)* –

# Commit Wait

We don't know the *exact* time … but we can wait out the uncertainty

1. Acquire all locks

   – *do work (acquire locks as needed)* –

2. Get a commit timestamp: `t = TT.now().latest`

# Commit Wait

We don't know the *exact* time … but we can wait out the uncertainty

1. Acquire all locks

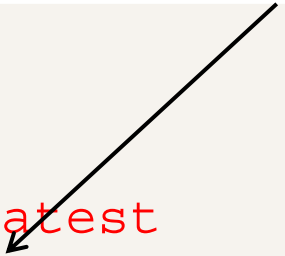
   – *do work (acquire locks as needed)* –

2. Get a commit timestamp: `t = TT.now().latest`
3. **Commit wait**: **wait until** `TT.now().earliest > t`

# Commit Wait

We don't know the *exact* time … but we can wait out the uncertainty

*average wait is ~4 ms*



1. Acquire all locks

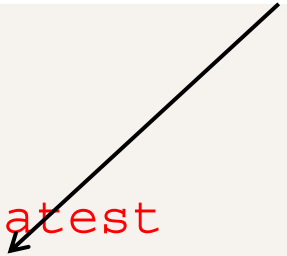
   *– do work (acquire locks as needed) –*

2. Get a commit timestamp: `t = TT.now().latest`
3. **Commit wait**: **wait until** `TT.now().earliest > t`

# Commit Wait

We don't know the *exact* time … but we can wait out the uncertainty

*average wait is ~4 ms*

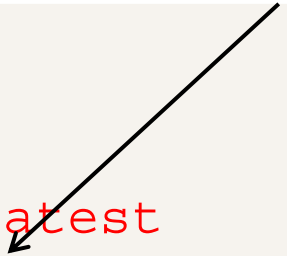


1. Acquire all locks
   – *do work (acquire locks as needed)* –
2. Get a commit timestamp: `t = TT.now().latest`
3. **Commit wait**: **wait until** `TT.now().earliest > t`
4. Commit
5. Release locks

# Commit Wait

We don't know the *exact* time … but we can wait out the uncertainty

*average wait is ~4 ms*



1. Acquire all locks
   – *do work (acquire locks as needed)* –
2. Get a commit timestamp: `t = TT.now().latest`
3. **Commit wait**: **wait until** `TT.now().earliest > t`
4. Commit
5. Release locks

**Guarantee**: If transaction T2 starts after transaction T1 commits, then T2 receives a later timestamp than T1. This is what gives Spanner external consistency

# Commit Wait and Replication

1. Acquire all locks

   – *do work (acquire locks as needed)* –

2. Get a commit timestamp: `t = TT.now().latest`

# Commit Wait and Replication

1. Acquire all locks

   – *do work (acquire locks as needed)* –

2. Get a commit timestamp: `t = TT.now().latest`

3. (a) Start consensus for replication

# Commit Wait and Replication

1. Acquire all locks

   – *do work (acquire locks as needed)* –

2. Get a commit timestamp: `t = TT.now().latest`

3. (a) Start consensus for replication
   (b) Commit wait (in parallel)

} **Make the replicas & wait for all to finish**

# Commit Wait and Replication

1. Acquire all locks
   – *do work (acquire locks as needed)* –
2. Get a commit timestamp: `t = TT.now().latest`
3. (a) Start consensus for replication
   (b) Commit wait (in parallel)

   } **Make the replicas & wait for all to finish**
4. Commit
5. Release locks

# Spanner Summary

- Semi-relational database of tables
- SQL-based query language
- Multi-version data model
- Synchronous replication using Paxos
- Distributed across many machines and many datacenters

# Are we breaking the rules?

- **Global ordering of transactions**
  - ***Systems cannot have globally synchronized clocks***
  - True: Spanner only requires bounded uncertainty and a wait before commit

# Are we breaking the rules?

- **Global ordering of transactions**
  - ***Systems cannot have globally synchronized clocks***
- **CAP theorem**
  - ***We cannot offer Consistency + Availability + Partition tolerance***

# Are we breaking the rules?

- **Global ordering of transactions**
  - ***Systems cannot have globally synchronized clocks***

- **CAP theorem**
  - ***We cannot offer Consistency + Availability + Partition tolerance***
  - Spanner is a CP system
    - If there is a partition, Spanner chooses C over A
    - In practice, users can feel they have a CA system

# Efficiency: You Don't Pay for What You Don't Use

**Spanner provides powerful guarantees,**
**but lets applications opt out of what they don't need**

- Transactions that touch only one split avoid two-phase commit
- Reads that can tolerate stale data avoid some of the extra work needed for the newest data
- Blind writes can be cheaper than read-modify-write transactions, though conflicting writes still require coordination
- Primary-key design affects performance: ordered keys help range scans, while randomized prefixes can reduce hotspots

# Spanner Conclusion

- ACID semantics not sacrificed
- Wide-area distributed transactions are built in
- TrueTime makes global timestamp ordering possible

# The End