# Spanner: Google's Globally Distributed Database

# Spanner

- Google's globally distributed multi-versioned database
- General purpose transactions (ACID)
- Schematized semi-relational tables, SQL-like queries
- Synchronous replication
- Lock-free distributed read transaction
- External consistency at global scale

# Basic Data Storage Idea

- Shard the database across rows
- Store shards in replica sets (group) geographically distributed
  - Single datacentre, across datacenters spanning a continent, across continents
  - Paxos based replication within each set
- Admin controls number, type, and geo-placement of replicas
  - Ex: 5-way replication in North America, 3-way replication in Europe etc.
- Application can specify
  - Which data goes to which DC
  - How far data is from users (controls read latency)
  - How far replicas are from each other (controls write latency)

- Tablet
  - Set of (key, timestamp, value)
    - Set of rows
  - Stored in B-tree like files along with Write-Ahead-Log in an underlying Google Colossus filesystem
- Directory
  - A bucketing abstraction on top of the key-value mapping
  - Unit of data placement and movement
  - Set of contiguous keys with the same “prefix”
  - Smallest unit whose geographical replication placement can be specified by the application
- A tablet can contain many directories
  - Need not be lexicographically contiguous partitions of the row space

**Universe**: holds one or more databases

**Database**: holds one or more tables

**Table**: rows & columns

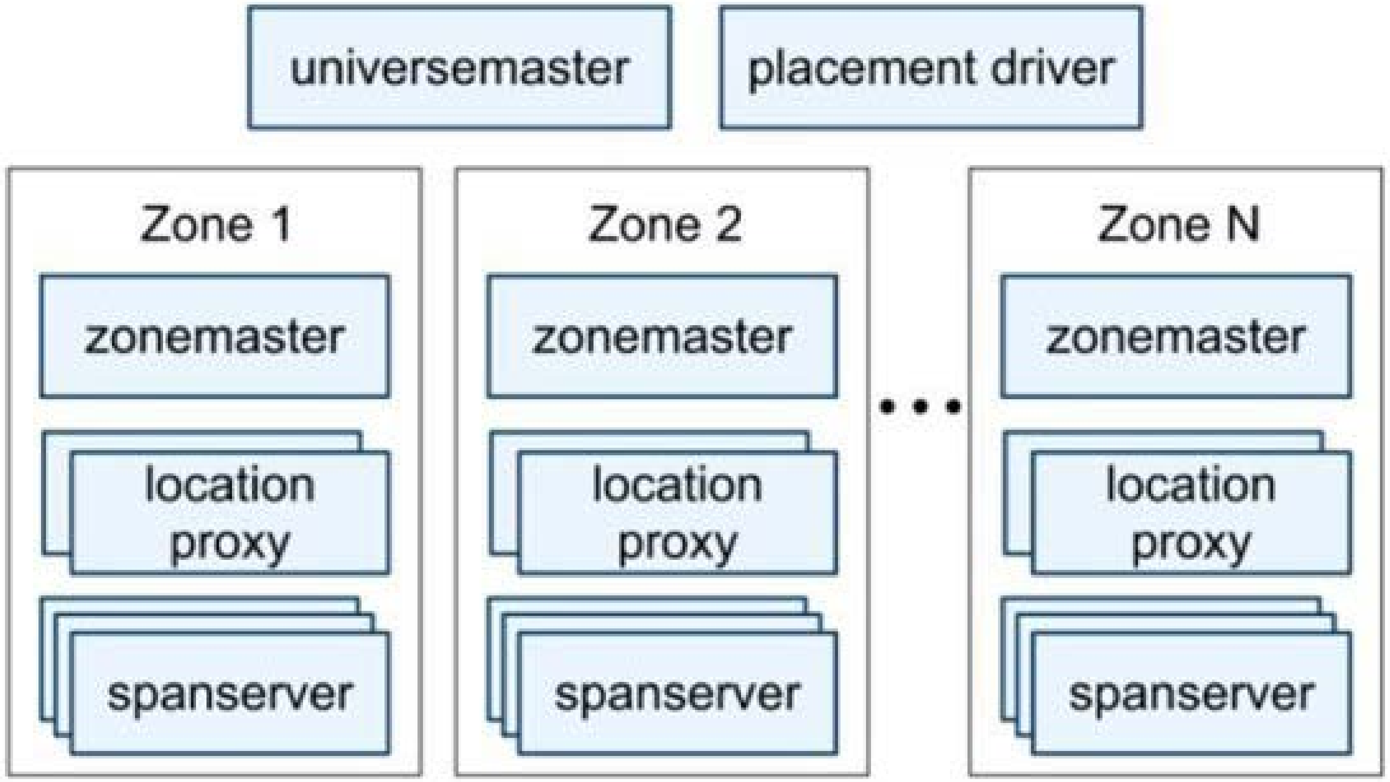
**Shards (tablets)**: pieces of tables
**Replicated synchronously via Paxos**

Data in table is versioned & has a timestamp

**Transactions across shards use two-phase commit**

**Directory**: “bucket” – set of contigious keys with a common prefix
**Unit of data movement between Paxos groups**

# Basic Architecture



# Universe

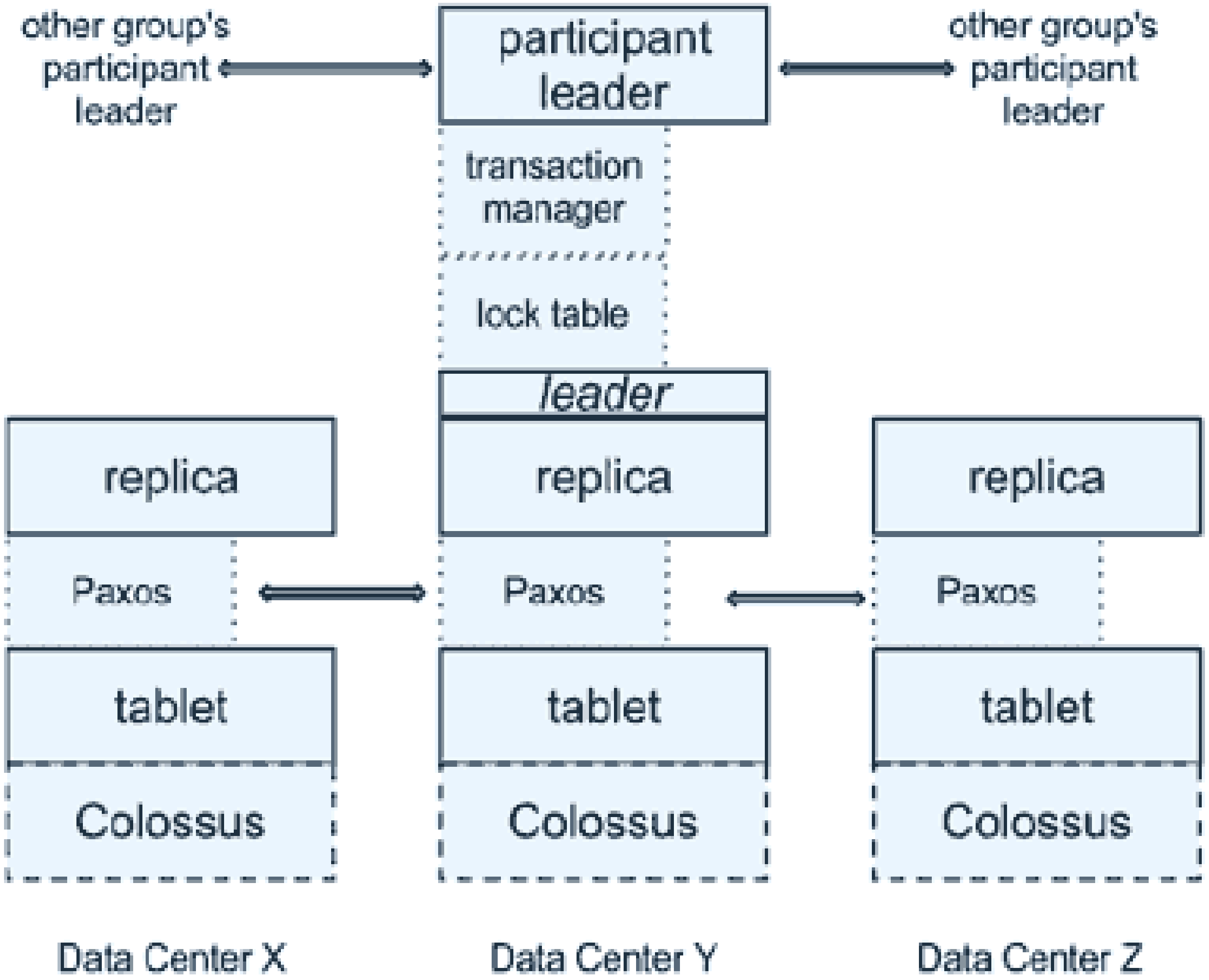
- One deployment of spanner
- Holds one or more databases created by application
- Has
  - A universemaster
    - Console to display/monitor status of all zones for debugging
  - A placement driver
    - Controls auto movement of data across zones
      - May need to be moved for load balancing, addition/deletion of replicas, grouping data with similar access patterns etc.
    - Communicates with spanservers to decide what data to move when

# Zones

- Set of zones = set of locations for replicating data
- Zones can be dynamically added or removed
- Unit of physical isolation
  - Data of different applications can be in different zones, even within the same DC
- Has
  - One zonemaster
    - Assigns data to spanservers
  - Few Location Proxies
    - Used by clients to find spanservers assigned to serve their data
  - 100’s to 1000’s of spanservers
    - Actually stores data

# Spanserver

- Each spanserver stores
  - 100-1000 tablets
  - One Paxos protocol instance running for each tablet
  - Paxos stores its metadata and log in the same tablet
  - Each Paxos write is logged in both Paxos log and tablet log
  - Writes applied by Paxos in order
    - Writes invoke Paxos protocol at leader
  - Reads access state at underlying tablet at any replica that is sufficiently "up-to-date"
  - The set of replicas of a tablet is collectively called a Paxos group


other group's participant leader
participant leader
other group's participant leader
transaction manager
lock table
leader
replica
replica
replica
Paxos
Paxos
Paxos
tablet
tablet
tablet
Colossus
Colossus
Colossus
Data Center X
Data Center Y
Data Center Z

- Concurrency control
  - Leader implements a lock table
    - Two-phase locking
- Supporting distributed transactions
  - Necessary when a transaction spans across groups
  - Leader implements a transaction manager
    - If a transaction involves only one Paxos group, no transaction manager needed
  - Transaction managers of the group coordinate to do two-phase commit
    - One group is chosen as coordinator, that leader is “coordinator leader”
    - Others are “coordinator slaves”
  - Transaction manager state is also stored in Paxos group
    - So sort of replicated two-phase commit, avoids blocking when coordinator fails

# Data Model

- Schematized semi-relational model
- SQL like query language
- Synchronous replication
- Data model layered on top of directory-bucketed key-value mapping
- Application creates one or more databases in an universe
    - Each DB can contain an unlimited number of schematized tables
    - Tables look like relational tables with rows, columns, and versioned values

# TrueTime

- Spanner's implementation global wall-clock time
- Main interesting idea
  - If you ask the local clock for a time, it does not give a single time, it gives an interval within which the global time will lie
- Three APIs
  - TT.now() – returns an interval [earliest, latest]
    - Guaranteed to contain the absolute time at which the TT.now() was invoked
  - TT.before(t)
    - Returns true if t has definitely passed
  - TT.after(t)
    - Returns true if t has definitely not arrived

- Time implemented by GPS and Atomic clocks
- Set of time master m/cs per DC
  - Most have GPS with dedicated antenna
  - Some have Atomic Clocks
- Time slave daemon per machine
  - Pulls time from a variety of time masters
    - Including from other DCs, near and far
  - Applies a version of Marzullo's algorithm to
    - Detect and reject liars
    - Synchronize local machine clock with non-liars

# External Consistency

- Consistency requirement
  - If start of $T_2$ occurs after commit of $T_1$, then $TS(T_2) > TS(T_1)$
- How do we enforce external consistency?
  - Enabler: Interval-based global time from TrueTime API.
  - Two rules for executing transactions and assigning timestamps
    - Start: Coordinating leader for a write $T_i$ assigns a commit timestamp $s_i$ >= TT.now().latest computed after the arrival time of the commit request at the coordinating leader of Ti
    - Commit Wait: Coordinating leader ensures that client cannot see any data committed by $T_i$ until TT.after($s_i$) is true

# Types of Transactions

- Read-only
  - Must be pre-defined by clients as read-only
  - Execution after a system chooses a timestamp without locking
    - Does not block incoming writes
  - Can be from any replica that is sufficiently up-to-date
- Snapshot read
  - Lock free read of a past version of data
  - Client can specify a timestamp or a max acceptable staleness
- Read-write transactions

# Assigning Timestamps for Read-Only Transactions

- Find scope of the reads
  - All keys that will be read in the transaction
- If the scope's values are served by a single Paxos group
  - Client issues a read-only transaction to that group's leader
  - The leader assign a timestamp that is greater than the last committed write at the group
- If not
  - Leaders can coordinate to assign a timestamp based on the last committed transactions at each group
  - Spanner approach: Assign TT.now().latest
    - No coordination is needed, so fast

# Serving Read at a Timestamp

- Read from a sufficiently up-to-date replica
- What is up-to-date?
  - Define a safe time which is the maximum time at which a replica is up-to-date
    - Safe time = min(safe time of Paxos state machine, safe time of transaction manager)
    - Safe time of Paxos = timestamp of highest applied Paxos write
    - Safe time of TM = min(prepare timestamp of all transactions prepared at the group) – 1
      - Coordinating leader ensures that any transactions commit timestamp is >= prepare timestamps of the transaction over all groups involved in the transaction
      - So choosing this as safe time says that no further writes will happen before this
      - Known from local Paxos log. So fast again.
  - Replica can satisfy a read at a timestamp t if t is <= the safe time

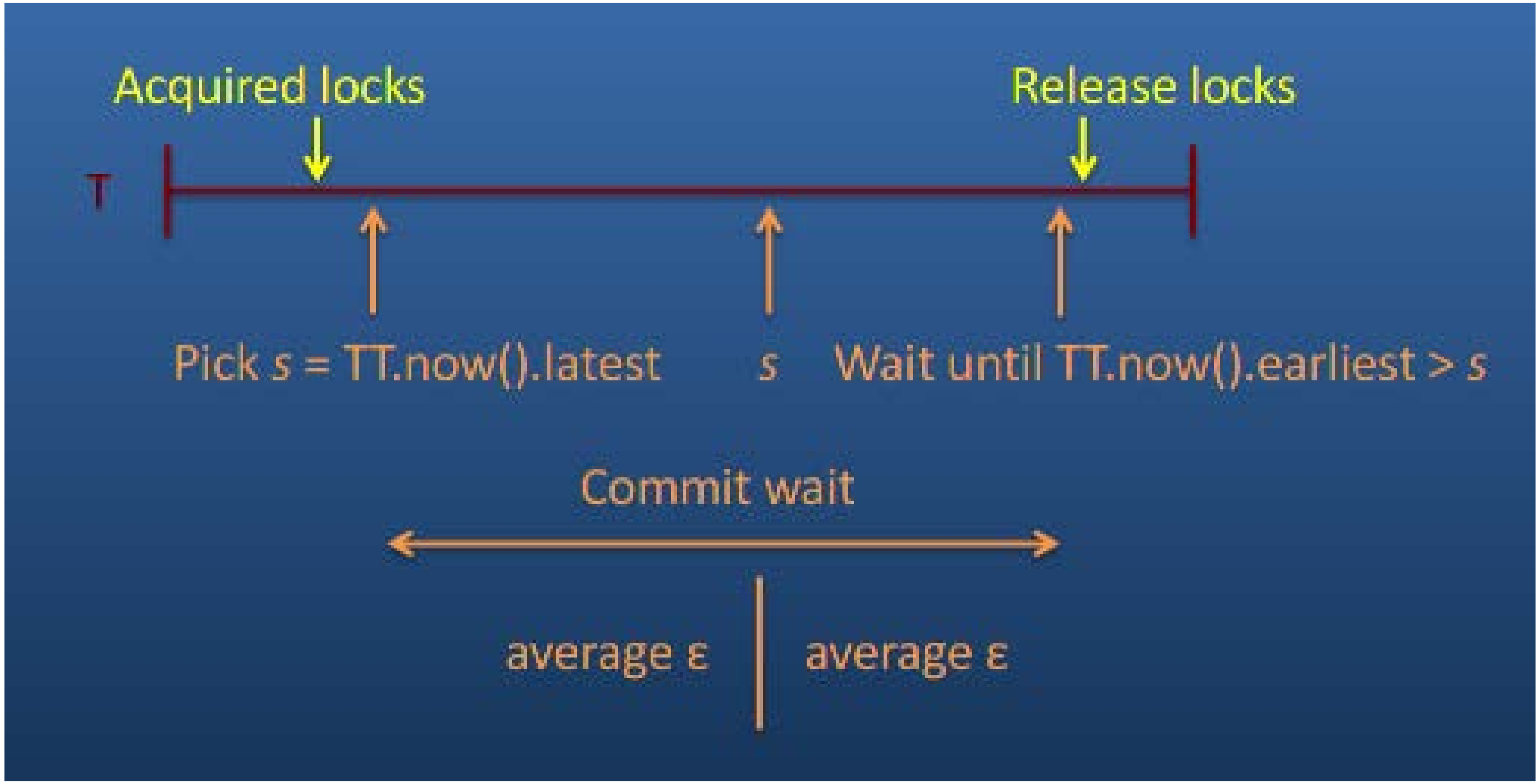
# Read-Write Transactions

- Client issues reads to leader replicas of appropriate groups. These acquire read locks and read the most recent data.
- Once reads are completed and writes are buffered (at the client), client chooses a coordinator leader and sends the identity of the leader along with buffered writes to participant leaders.
- Non-coordinator participant leaders
  - Acquire write locks
  - Choose a prepare timestamp larger than any previous transaction timestamps and log a prepare record in Paxos.
  - Notify coordinator of chosen timestamp

# Aassigning timestamp to a RW transaction

- Can be assigned  a timestamp anytime after it has acquired all logs and before it has released any
- Assigned by coordinating leader
- Assign a timestamp that is
  - >= all prepare timestamps received from coordinators
  - > TT.now().latest at the time the coordinating leader received its commit message from the client
  - Greater than any timestamp assigned to any previous transaction by the leader
- Log the commit/abort record through Paxos, but delay sending decision to other coordinators until TT.after(s), where s is the timestamp assigned
  - Guarantees s has elapsed

# Commit Wait


Acquired locks
Release locks
T
Pick s = TT.now().latest
s
Wait until TT.now().earliest > s
Commit wait
average ε
average ε

# Implications of TrueTime

- The larger the uncertainty bound from TrueTime, the longer commit wait period.
- Commit wait will slow down dependent transactions, since locks are held during commit wait.
- So, as time gets less certain, Spanner gets slower.
- Attack Vector: you can cause very long commit wait periods – slow the system down by messing with the clock.