Megastore and Spanner

Amir H. Payberah
amir@sics.se

Amirkabir University of Technology
(Tehran Polytechnic)

Motivation

- Storage requirements of today's interactive online applications.
  - Scalability (a billion internet users)
  - Rapid development
  - Responsiveness (low latency)
  - Durability and consistency (never lose data)
  - Fault tolerant (no unplanned/planned downtime)
  - Easy operations (minimize confusion, support is expensive)

Motivation

- Storage requirements of today's interactive online applications.
  - Scalability (a billion internet users)
  - Rapid development
  - Responsiveness (low latency)
  - Durability and consistency (never lose data)
  - Fault tolerant (no unplanned/planned downtime)
  - Easy operations (minimize confusion, support is expensive)

- These requirements are in conflict.

Motivation

- Relational DBMS, e.g., MySQL, MS SQL, Oracle RDB
  - Rich set of features
  - Difficult to scale to the massive amount of reads and writes.

Motivation

- Relational DBMS, e.g., MySQL, MS SQL, Oracle RDB
  - Rich set of features
  - Difficult to scale to the massive amount of reads and writes.

- NoSQL, e.g., BigTable, Dynamo, Cassandra
  - Highly Scalable
  - Limited API

NewSQL Databases

- NoSQL scalability + RDBMS ACID
- E.g., Megastore and Spanner

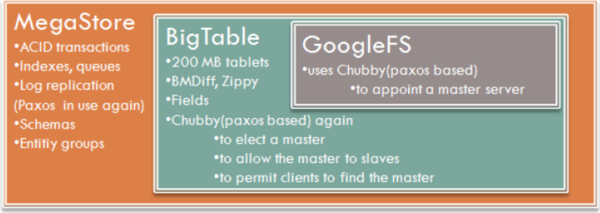
Megastore

Megastore

- Started in 2006 for app development at Google.
- Google's wide-area replicated data store.
- Adds (limited) transactions to wide-area replicated data stores.
- GMail, Google+, Android Market, Google App Engine, ...

Megastore

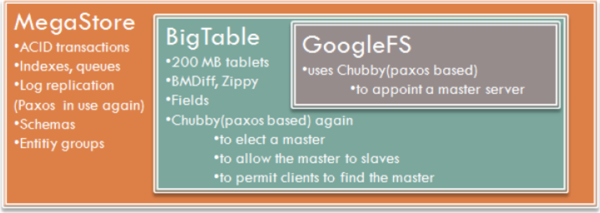
- Megastore layered on:
  - GFS (Distributed file system)
  - Bigtable (NoSQL scalable data store per datacenter)



[http://cse708.blogspot.jp/2011/03/megastore-providing-scalable-highly.html]

Megastore

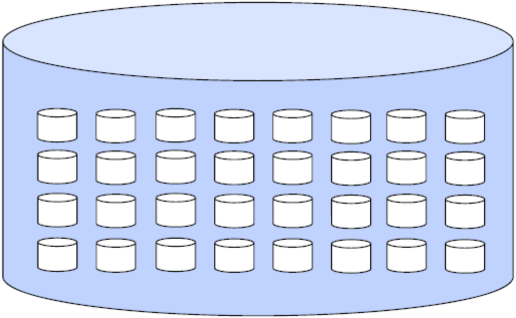
- Megastore layered on:
  - GFS (Distributed file system)
  - Bigtable (NoSQL scalable data store per datacenter)
- BigTable is cluster-level structured storage, while Megastore is geo-scale structured database.



[http://cse708.blogspot.jp/2011/03/megastore-providing-scalable-highly.html]

Entity Group (1/2)

- The data is partitioned into a collection of entity groups (EG).

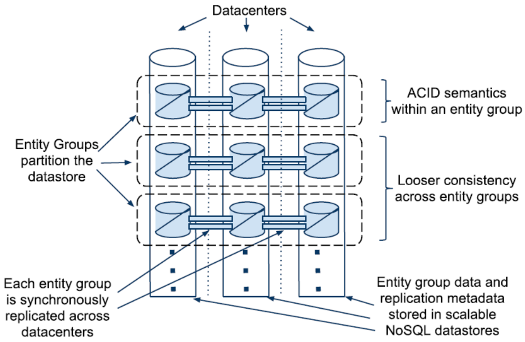


Entity Group (2/2)

Application	Entity Groups	Cross-EG Ops
Email	User accounts	none
Blogs	Users, Blogs	Access control, notifications, global indexes
Mapping	Local patches	Patch-spanning ops
Social	Users, Groups	Messages, relationships, notifications
Resources	Sites	Shipments

Entity Group Replication (1/2)

- Each entity group independently and synchronously replicated over a wide area.
- Megastore's replication system provides a single consistent view of the data stored in its underlying replicas.



Entity Group Replication (2/2)

- Synchronous replication: a low-latency implementation of paxos.

Entity Group Replication (2/2)

- Synchronous replication: a low-latency implementation of paxos.
- Basic paxos not used: poor match for high-latency links.

Entity Group Replication (2/2)

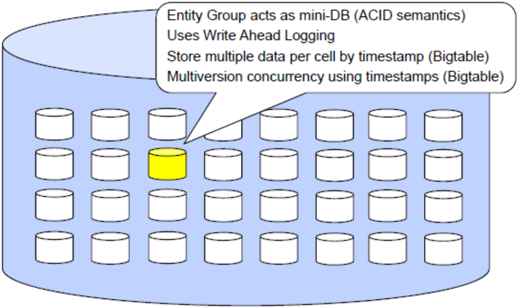
- Synchronous replication: a low-latency implementation of paxos.
- Basic paxos not used: poor match for high-latency links.
  - Writes require at least two inter-replica round-trips to achieve consensus: prepare round, accept round
  - Reads require one inter-replica round-trip: prepare round

Entity Group Replication (2/2)

- Synchronous replication: a low-latency implementation of paxos.
- Basic paxos not used: poor match for high-latency links.
  - Writes require at least two inter-replica round-trips to achieve consensus: prepare round, accept round
  - Reads require one inter-replica round-trip: prepare round
- Megastore uses a modified version of paxos: fast read, fast write

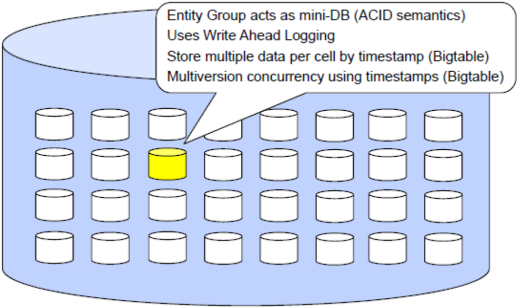
Entity Group Transaction (1/3)

- Within each EG: full ACID semantics



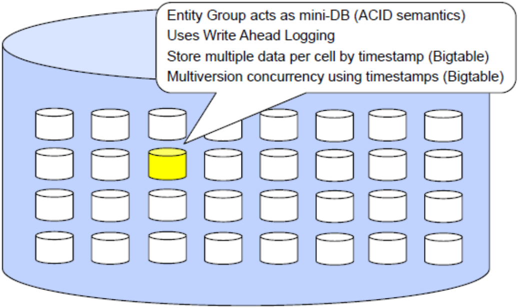
Entity Group Transaction (1/3)

- Within each EG: full ACID semantics
- Transaction management using Write Ahead Logging (WAL).



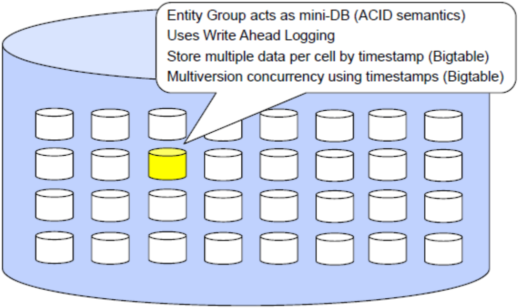
Entity Group Transaction (1/3)

- Within each EG: full ACID semantics
- Transaction management using Write Ahead Logging (WAL).
- BigTable feature: ability to store multiple data for same row/column with different timestamps.



Entity Group Transaction (1/3)

- Within each EG: full ACID semantics
- Transaction management using Write Ahead Logging (WAL).
- BigTable feature: ability to store multiple data for same row/column with different timestamps.
- Multiversion Concurrency Control (MVCC) in EGs.



Entity Group Transaction (2/3)

- Read consistency

Entity Group Transaction (2/3)

- Read consistency
  - Current: waits for uncommitted writes, then reads the last committed value.

Entity Group Transaction (2/3)

- Read consistency
  - Current: waits for uncommitted writes, then reads the last committed value.
  - Snapshot: doesn't wait, and reads the last committed values.

Entity Group Transaction (2/3)

- Read consistency
  - Current: waits for uncommitted writes, then reads the last committed value.
  - Snapshot: doesn't wait, and reads the last committed values.
  - Inconsistent reads: ignores the state of log and reads the last values directly (data may be stale).

Entity Group Transaction (3/3)

- Write consistency
  - Determine the next available log position.

Entity Group Transaction (3/3)

- Write consistency
  - Determine the next available log position.
  - Assigns mutations of WAL a timestamp higher than any previous one.

Entity Group Transaction (3/3)

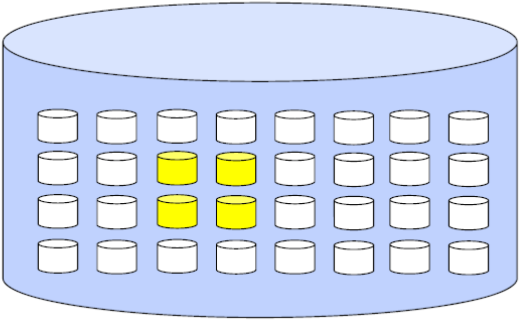
- Write consistency
  - Determine the next available log position.
  - Assigns mutations of WAL a timestamp higher than any previous one.
  - Employs paxos to settle the resource contention.

Entity Group Transaction (3/3)

- Write consistency
  - Determine the next available log position.
  - Assigns mutations of WAL a timestamp higher than any previous one.
  - Employs paxos to settle the resource contention.
  - Based on optimistic concurrency: in case of multiple writers to the same log position, only one will win, and the rest will notice the victorious write, abort, and retry their operations.

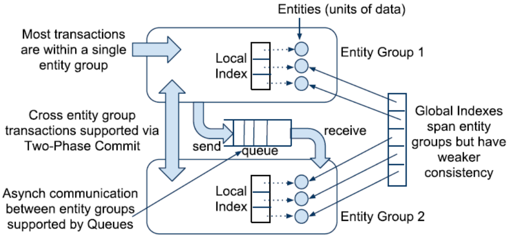
Across Entity Group Transaction (1/3)

- Across entity groups: limited consistency guarantees
- Two methods:
  - Asynchronous messaging (queue)
  - Two-Phase-Commit (2PC)



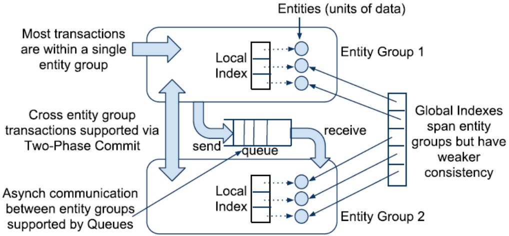
Across Entity Group Transaction (2/3)

- Queues
- Provide transactional messaging between EGs.
- Each message either is:
  - Synchronous: has a single sending and receiving entity group.
  - Asynchronous: has different sending and receiving entity group.
- Useful to perform operations that affect many EGs.



Across Entity Group Transaction (3/3)

- Two-Phase Commit
- Atomicity is satisfied.
- High latency



Spanner

Limitations of Existing Systems

- BigTable
  - Scalability
  - High throughput
  - High performance
  - Transactional scope limited to single row
  - Eventually-consistent replication support across data-centers

Limitations of Existing Systems

- Megastore
  - Replicated ACID transactions
  - Schematized semi-relational tables
  - Synchronous replication support across data-centers
  - Performance (poor write throughput)
  - Lack of query language

Solution: Google Spanner

- Bridging the gap between Megastore and Bigtable.
- SQL transactions + high throughput

Spanner

- Global scale database with strict transactional guarantees.

Spanner

- Global scale database with strict transactional guarantees.

- Global scale
  - Across datacenters
  - Scale up to millions of nodes, hundreds of datacenters, trillions of database rows

Spanner

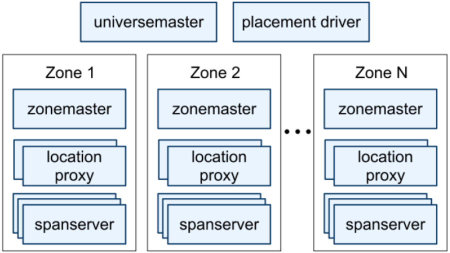
- Global scale database with strict transactional guarantees.

- Global scale
  - Across datacenters
  - Scale up to millions of nodes, hundreds of datacenters, trillions of database rows

- Strict transactional guarantees
  - General transactions (even inter-row)
  - Reliable even during wide-area natural disasters

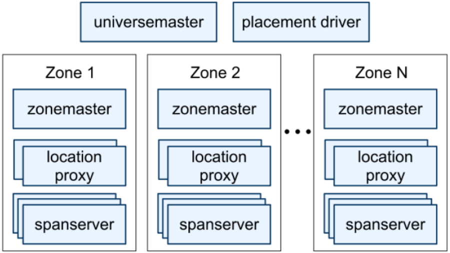
Spanner Implementation

Spanner Organization (1/2)



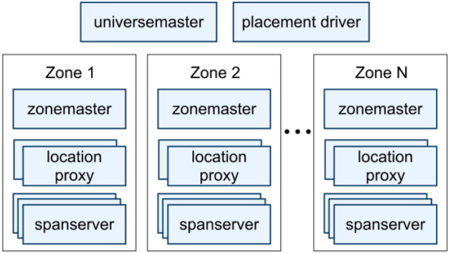
- Universe: Spanner deployment

Spanner Organization (1/2)



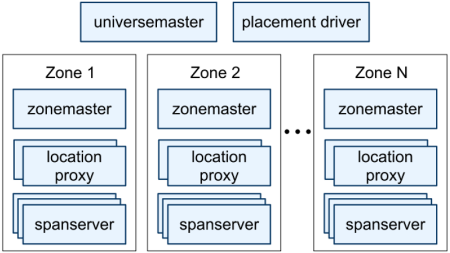
- Universe: Spanner deployment
- Zones: analogues to deployment of BigTable servers (unit of physical isolation)

Spanner Organization (1/2)



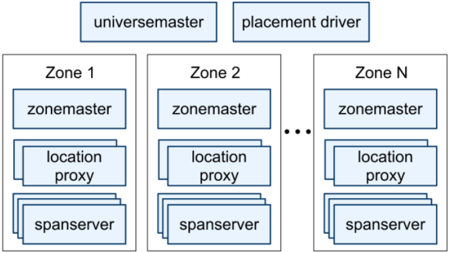
- Universe: Spanner deployment
- Zones: analogues to deployment of BigTable servers (unit of physical isolation)
  - One zonemaster: assigns data to spanservers

Spanner Organization (1/2)



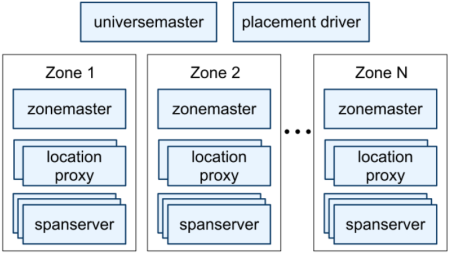
- Universe: Spanner deployment
- Zones: analogues to deployment of BigTable servers (unit of physical isolation)
    - One zonemaster: assigns data to spanservers
    - The proxies: used by clients to locate the spanservers assigned to serve their data

Spanner Organization (1/2)



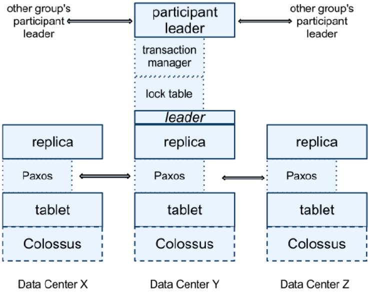
- Universe: Spanner deployment
- Zones: analogues to deployment of BigTable servers (unit of physical isolation)
  - One zonemaster: assigns data to spanservers
  - The proxies: used by clients to locate the spanservers assigned to serve their data
  - Thousands of spanservers: serve data to clients

Spanner Organization (2/2)



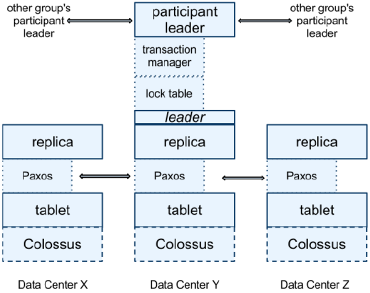
- The universe master: a console that displays status information about all the zones.
- The placement driver: handles automated movement of data across zones.

Spanserver Software Stack (1/4)



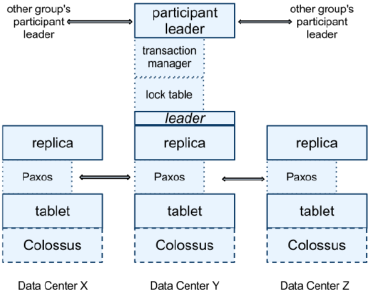
- Each spanserver is responsible for 100-1000 data structure instances, called tablet (similar to BigTable tablet).
- Tablet mapping: (key: string, timestamp:int64) $\rightarrow$ string
- Data and logs stored on Colossus (successor of GFS).

Spanserver Software Stack (2/4)



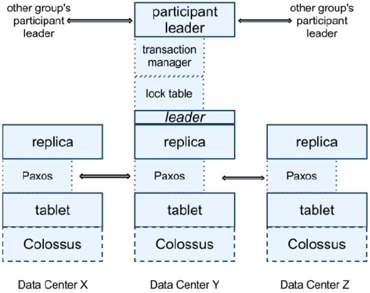
- A single paxos state machine on top of each tablet: consistent replication

Spanserver Software Stack (2/4)



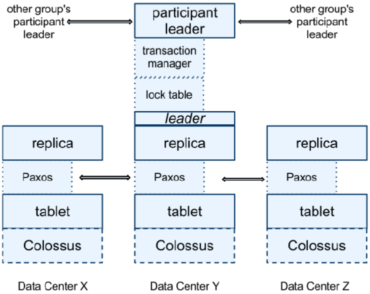
- A single paxos state machine on top of each tablet: consistent replication
- Paxos group: all machines involved in an instance of paxos.

Spanserver Software Stack (2/4)



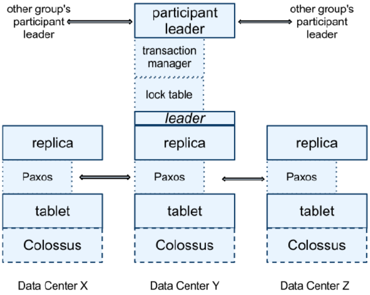
- A single paxos state machine on top of each tablet: consistent replication
- Paxos group: all machines involved in an instance of paxos.
- Paxos implementation supports long-lived leaders with time-based leader leases.

Spanserver Software Stack (3/4)



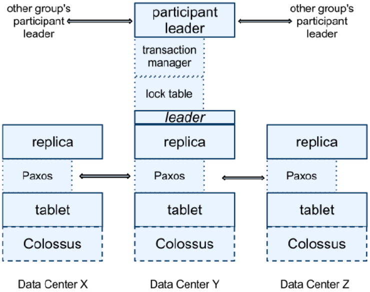
- Writes must initiate the paxos protocol at the leader.

Spanserver Software Stack (3/4)



- Writes must initiate the paxos protocol at the leader.
- Reads access state directly from the underlying tablet at any replica that is sufficiently up-to-date.

Spanserver Software Stack (4/4)



- Transaction manager: to support distributed transactions
  - At every replica that is a leader.

Transactions Involving Only One Paxos Group

- This is the case for most transactions.
- A long lived paxos leader.
  - The transaction manager: participant leader
  - The other replicas in the group: participant slaves

Transactions Involving Only One Paxos Group

- This is the case for most transactions.
- A long lived paxos leader.
  - The transaction manager: participant leader
  - The other replicas in the group: participant slaves
- A lock table for concurrency control.
  - Multiple concurrent transactions.

Transactions Involving Only One Paxos Group

- This is the case for most transactions.
- A long lived paxos leader.
  - The transaction manager: participant leader
  - The other replicas in the group: participant slaves
- A lock table for concurrency control.
  - Multiple concurrent transactions.
  - Maintained by paxos leader.

Transactions Involving Only One Paxos Group

- This is the case for most transactions.
- A long lived paxos leader.
  - The transaction manager: participant leader
  - The other replicas in the group: participant slaves
- A lock table for concurrency control.
  - Multiple concurrent transactions.
  - Maintained by paxos leader.
  - Maps ranges of keys to lock states.

Transactions Involving Only One Paxos Group

- This is the case for most transactions.
- A long lived paxos leader.
  - The transaction manager: participant leader
  - The other replicas in the group: participant slaves
- A lock table for concurrency control.
  - Multiple concurrent transactions.
  - Maintained by paxos leader.
  - Maps ranges of keys to lock states.
  - Two-phase locking.

Transactions Involving Only One Paxos Group

- This is the case for most transactions.
- A long lived paxos leader.
  - The transaction manager: participant leader
  - The other replicas in the group: participant slaves
- A lock table for concurrency control.
  - Multiple concurrent transactions.
  - Maintained by paxos leader.
  - Maps ranges of keys to lock states.
  - Two-phase locking.
  - Wound-wait for dead lock avoidance: young transaction dies if an older transaction needs a resource held by the young transaction.

Transactions Involving Only One Paxos Group

- This is the case for most transactions.
- A long lived paxos leader.
  - The transaction manager: participant leader
  - The other replicas in the group: participant slaves
- A lock table for concurrency control.
  - Multiple concurrent transactions.
  - Maintained by paxos leader.
  - Maps ranges of keys to lock states.
  - Two-phase locking.
  - Wound-wait for dead lock avoidance: young transaction dies if an older transaction needs a resource held by the young transaction.
- It can bypass the transaction manager.

Transactions Involving Multiple Paxos Groups

- One of the participant groups is chosen as the coordinator.
  - The participant leader of that group will be referred to as the coordinator leader.
  - The slaves of that group as coordinator slaves.

Transactions Involving Multiple Paxos Groups

- One of the participant groups is chosen as the coordinator.
  - The participant leader of that group will be referred to as the coordinator leader.
  - The slaves of that group as coordinator slaves.

- Group's leaders coordinate to perform two phase commit.

Transactions Involving Multiple Paxos Groups

- One of the participant groups is chosen as the coordinator.
  - The participant leader of that group will be referred to as the coordinator leader.
  - The slaves of that group as coordinator slaves.
- Group's leaders coordinate to perform two phase commit.
- The state of each transaction manager is stored in the underlying paxos group (and therefore is replicated).

Data Model and Directories

Data Model

- An application creates one or more databases in a universe.

Data Model

- An application creates one or more databases in a universe.
- Each database can contain an unlimited number of schematized tables.

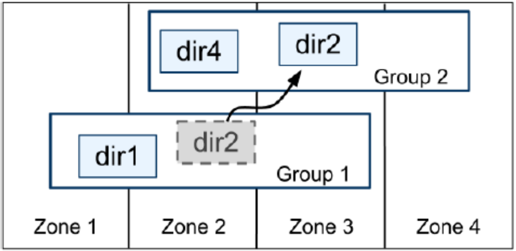
Data Model

- An application creates one or more databases in a universe.
- Each database can contain an unlimited number of schematized tables.
- Table
  - Rows and columns
  - Must have an ordered set one or more primary key columns
  - Primary key uniquely identifies each row

Data Model

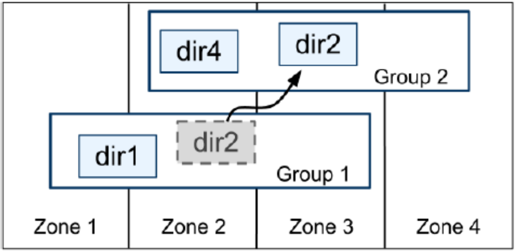
- An application creates one or more databases in a universe.
- Each database can contain an unlimited number of schematized tables.
- Table
  - Rows and columns
  - Must have an ordered set one or more primary key columns
  - Primary key uniquely identifies each row
- Hierarchies of tables
  - Tables must be partitioned by client into one or more hierarchies of tables
  - Table in the top: directory table

Directory (1/2)



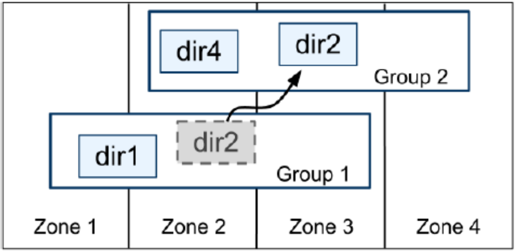
- Set of contiguous keys that share a common prefix.

Directory (1/2)



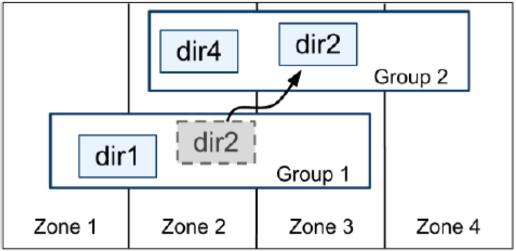
- Set of contiguous keys that share a common prefix.
- All data in a directory has the same replication configuration.

Directory (1/2)



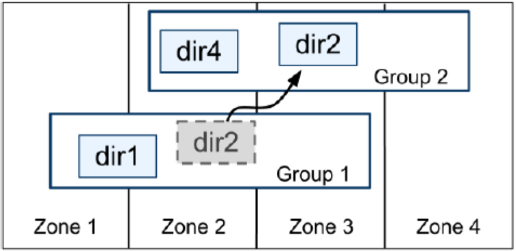
- Set of contiguous keys that share a common prefix.
- All data in a directory has the same replication configuration.
- The smallest unit whose geographic replication properties can be specified by an application.

Directory (1/2)



- Set of contiguous keys that share a common prefix.
- All data in a directory has the same replication configuration.
- The smallest unit whose geographic replication properties can be specified by an application.
- A Paxos group may contain multiple directories.

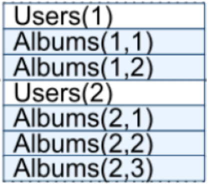
Directory (2/2)



- Spanner might move a directory:
  - To shed load from a paxos group.
  - To put directories that are frequently accessed together into the same group.
  - To move a directory into a group that is closer to its accessors.

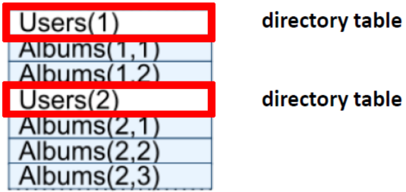
Example

```
CREATE TABLE Users {
  uid INT64 NOT NULL, email STRING
} PRIMARY KEY (uid), DIRECTORY;

CREATE TABLE Albums {
  uid INT64 NOT NULL, aid INT64 NOT NULL,
  name STRING
} PRIMARY KEY (uid, aid),
  INTERLEAVE IN PARENT Users ON DELETE CASCADE;
```



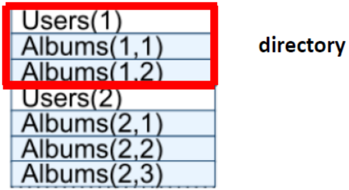
Example

```
CREATE TABLE Users {
  uid INT64 NOT NULL, email STRING
} PRIMARY KEY (uid), DIRECTORY;

CREATE TABLE Albums {
  uid INT64 NOT NULL, aid INT64 NOT NULL,
  name STRING
} PRIMARY KEY (uid, aid),
  INTERLEAVE IN PARENT Users ON DELETE CASCADE;
```



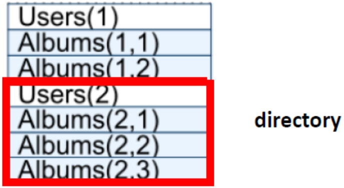
Example

```
CREATE TABLE Users {
  uid INT64 NOT NULL, email STRING
} PRIMARY KEY (uid), DIRECTORY;

CREATE TABLE Albums {
  uid INT64 NOT NULL, aid INT64 NOT NULL,
  name STRING
} PRIMARY KEY (uid, aid),
  INTERLEAVE IN PARENT Users ON DELETE CASCADE;
```



Example

```
CREATE TABLE Users {
  uid INT64 NOT NULL, email STRING
} PRIMARY KEY (uid), DIRECTORY;

CREATE TABLE Albums {
  uid INT64 NOT NULL, aid INT64 NOT NULL,
  name STRING
} PRIMARY KEY (uid, aid),
  INTERLEAVE IN PARENT Users ON DELETE CASCADE;
```



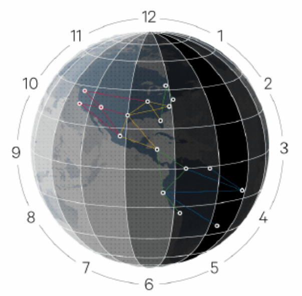
True Time and Consistency

Key Innovation

- Spanner knows what time it is.

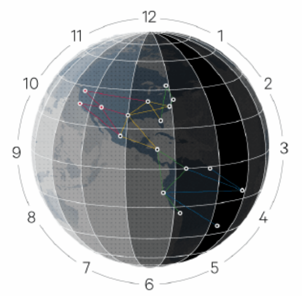
Time Synchronization (1/2)

- Is synchronizing time at the global scale possible?



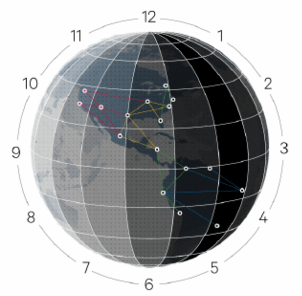
Time Synchronization (1/2)

- Is synchronizing time at the global scale possible?
- Synchronizing time within and between datacenters is extremely hard and uncertain.



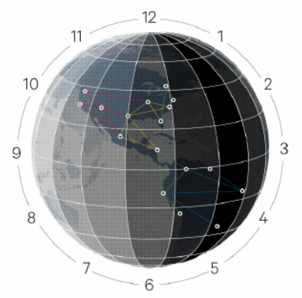
Time Synchronization (1/2)

- Is synchronizing time at the global scale possible?
- Synchronizing time within and between datacenters is extremely hard and uncertain.
- Serialization of requests is impossible at global scale.



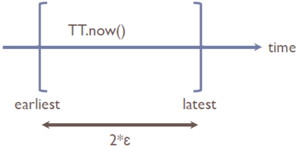
Time Synchronization (2/2)

- Idea: accept uncertainty, keep it small and quantify (using GPS and Atomic Clocks).



True Time API

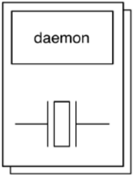
- TTinterval: is guaranteed to contain the absolute time during which TT.now() was invoked.



Method	Returns
TT.now()	TTinterval: [earliest, latest]
TT.after(t)	True if t has definitely passed
TT.before(t)	True if t has definitely not arrived

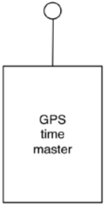
How TrueTime Is Implemented? (1/2)

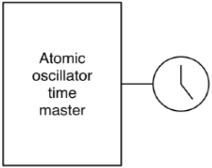
timeslave daemon per **machine**



set of time **master machines per datacenter**

majority of masters have
GPS receivers
with dedicated antennas



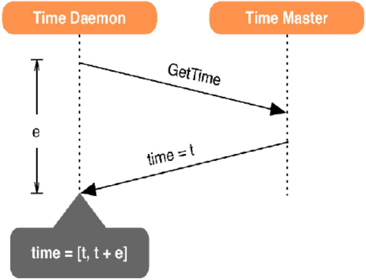


The remaining masters (which we refer to as **Armageddon masters**) are equipped with **atomic clocks.**

How TrueTime Is Implemented? (2/2)

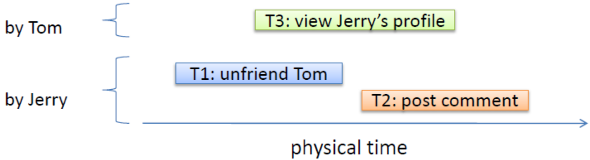
- Daemon polls variety of masters:
  - Chosen from nearby datacenters
  - From further datacenters
  - Armageddon masters

- Daemon polls variety of masters and reaches a consensus about correct timestamp.



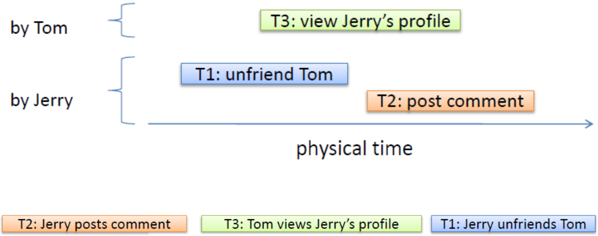
External Consistency (1/2)

- Jerry unfriends Tom to write a controversial comment.



External Consistency (1/2)

- Jerry unfriends Tom to write a controversial comment.



- If serial order is as above, Jerry will be in trouble!

External Consistency (2/2)

- External Consistency: Formally, If commit of T1 preceded the initiation of a new transaction T2 in wall-clock (physical) time, then commit of T1 should precede commit of T2 in the serial ordering also.

Snapshot Reads

- Read in past without locking.

Snapshot Reads

- Read in past without locking.
- Client can specify timestamp for read or an upper bound of timestamp.

Snapshot Reads

- Read in past without locking.
- Client can specify timestamp for read or an upper bound of timestamp.
- Each replica tracks a value called safe time t_{safe}, which is the maximum timestamp at which a replica is up-to-date.

Snapshot Reads

- Read in past without locking.
- Client can specify timestamp for read or an upper bound of timestamp.
- Each replica tracks a value called safe time t_{safe}, which is the maximum timestamp at which a replica is up-to-date.
- Replica can satisfy read at any $t \leq t_{safe}$.

Read-only Transactions

- Assign timestamp s_{read} and do snapshot read at s_{read}.
- $s_{read} = \texttt{TT.now().latest()}$
- It guarantees external consistency.

Read-Write Transactions (1/3)

- Leader must only assign timestamps within the interval of its leader lease.

Read-Write Transactions (1/3)

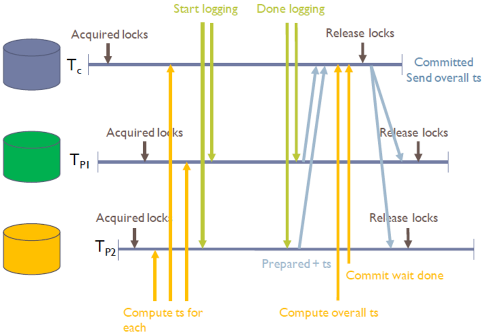
- Leader must only assign timestamps within the interval of its leader lease.
- Timestamps must be assigned in monotonically increasing order.

Read-Write Transactions (1/3)

- Leader must only assign timestamps within the interval of its leader lease.
- Timestamps must be assigned in monotonically increasing order.
- If transaction T1 commits before T2 starts, T2's commit timestamp must be greater than T1's commit timestamp.

Read-Write Transactions (2/3)

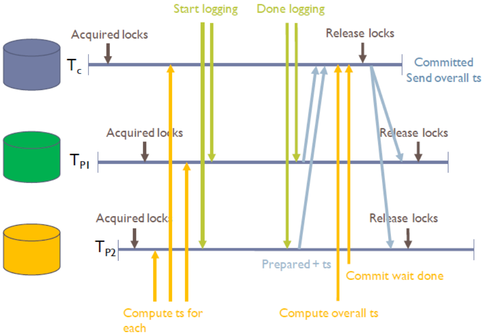
- Clients buffer writes.
- Client chooses a coordinate group that initiates two-phase commit.
- A non-coordinator-participant leader chooses a prepare timestamp and logs a prepare record through paxos and notifies the coordinator.



Read-Write Transactions (3/3)

- The coordinator assigns a commit timestamp s_i no less than all prepare timestamps and `TT.now().latest()`.

- The coordinator ensures that clients cannot see any data committed by T_i until `TT.after(`s_i`)` is true. This is done by commit wait (wait until absolute time passes s_i to commit).



Summary

Summary

- Megastore
- Entity Groups (EG)
- Within EG: using paxos - ACID
- Across EGs: using queue and two-phase commit

Summary

- Spanner
- Replica consistency: using paxos protocol
- Concurrency control: using two phase locking
- Transaction coordination: using two-phase commit
- Timestamps for transactions and data items

Questions?