Distributed Transactions

Dan Ports, CSEP 552

Today

- Bigtable (from last week)
- Overview of transactions
- Two approaches to adding transactions to Bigtable: MegaStore and Spanner
- Latest research: TAPIR

Bigtable

- stores (semi)-structured data
    - e.g., URL -> contents, metadata, links
    - e.g., user > preferences, recent queries
- really large scale!
    - capacity: 100 billion pages * 10 versions => 20PB
    - throughput: 100M users, millions of queries/sec
    - latency: can only afford a few milliseconds per lookup

Why not use a commercial DB?

- Scale is too large, and/or cost too high
- Low-level storage optimizations help
  - data model exposes locality, performance tradeoff
  - traditional DBs try to hide this!
- Can remove “unnecessary” features
  - secondary indexes, multirow transactions, integrity constraints

Data Model

- a big, sparse, multidimensional sorted table
- (row, column, timestamp) -> contents
- fast lookup on a key
- rows are ordered lexicographically, so scans in order

Consistency

- Is this an ACID system?
- Durability and atomicity: via commit log in GFS
- Strong consistency:
  operations get processed by a single server in order
- Isolated transactions:
  single-row only, e.g., compare-and-swap

Implementation

- Divide the table into tablets (~100 MB) grouped by a range of sorted rows
- Each tablet is stored on a tablet server that manages 10-1000 tablets
- Master assigns tablets to servers, reassigns when servers are new/crashed/overloaded, splits tablets as necessary
- Client library responsible for locating the data

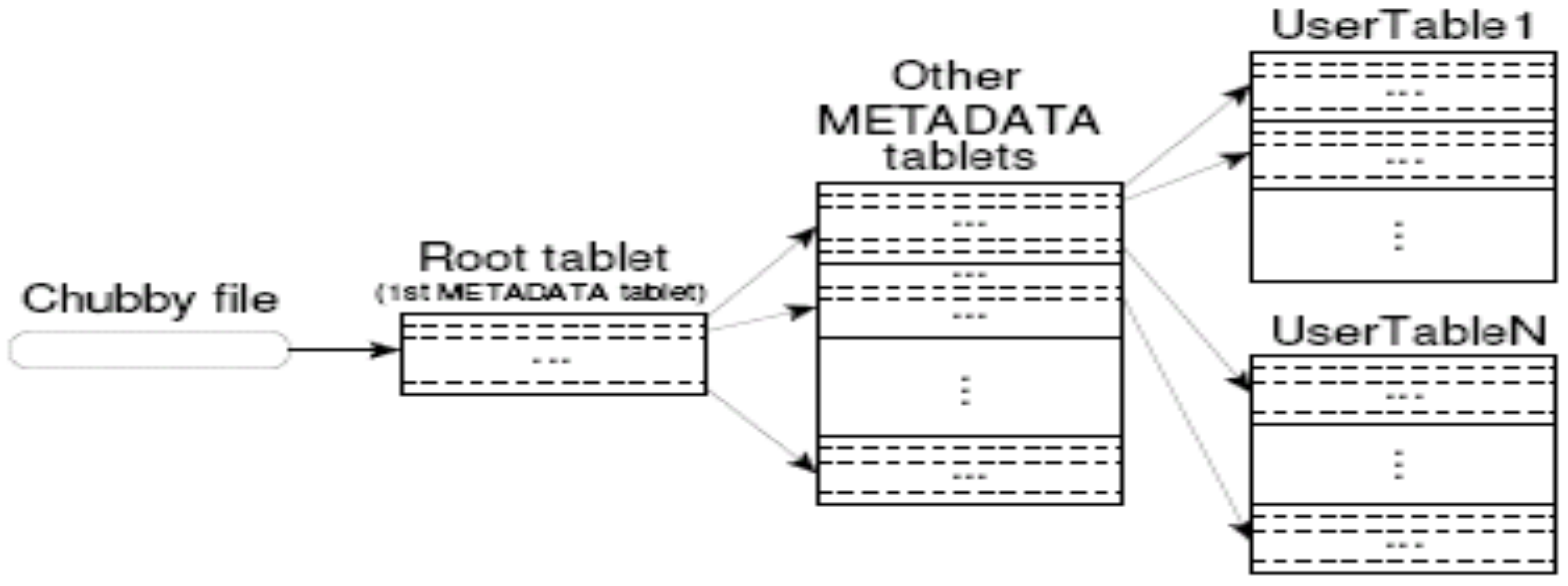
Is this just like GFS?

Is this just like GFS?

- Same general architecture, but…
- can leverage GFS and Chubby!
- tablet servers and master are basically stateless
  - tablet data is stored in GFS,
    coordinated via Chubby
  - master serves most config data in Chubby

Is this just like GFS?

- Scalable metadata assignment
- Don't store the entire list of row -> tablet -> server mappings in the master
- 3-level hierarchy
  entries are location: ip/port of relevant server



Fault tolerance

- If a tablet server fails (while storing ~100 tablets)
  - reassign each tablet to another machine
  - so 100 machines pick up just 1 tablet each
  - tablet SSTables & log are in GFS

- If the master fails
  - acquire lock from Chubby to elect new master
  - read config data from Chubby
  - contact all tablet servers to ask what they're responsible for

Bigtable in retrospect

- Definitely a useful, scalable system!
- Still in use at Google, motivated lots of NoSQL DBs
- Biggest mistake in design (per Jeff Dean, Google): not supporting distributed transactions!
    - became really important w/ incremental updates
    - users wanted them, implemented themselves, often incorrectly!
    - at least 3 papers later fixed this — two next week!

Transactions

- Important concept for simplifying reasoning about complex actions
- Goal: group a set of individual operations
  (reads and writes) into an atomic unit
  - e.g., checking_balance -= 100, savings_balance += 100
- Don't want to see one without the others
  - even if the system crashes (atomicity/durability)
  - even if other transactions are running concurrently (isolation)

Traditional transactions

- as found in a single-node database
- atomicity/durability: write-ahead logging
  - write each operation into a log on disk
  - write a commit record that makes all ops commit
  - only tell client op is done after commit record written
  - after a crash, scan log and redo any transaction with a commit record; undo any without

Traditional transactions

- isolation: concurrency control
  - simplest option: only run one transaction at a time!
  - standard (better) option: two-phase locking
    - keep a lock per object / DB row,
      usually single-writer / multi-reader
    - when reading or writing, acquire lock
    - hold all locks until after commit, then release

Transactions are hard

- definitely oversimplifying: see a database textbook on how to get the single-node case right
- …but let’s jump to an even harder problem:
  distributed transactions!
- What makes distributed transactions hard?
  - savings_bal and checking_bal might be stored on different nodes
  - they might each be replicated or cached
  - need to coordinate the ordering of operations across copies of data too!

Correctness for isolation

- usual definition: serializability
  each transaction's reads and writes are consistent with running them in a serial order, *one transaction at a time*

- sometimes: strict serializability = linearizability
  same definition + real time component

- two-phase locking on a single-node system provides strict serializability!

Weaker isolation?

- we had weaker levels of consistency:
  causal consistency, eventual consistency, etc
- we can also have weaker levels of *isolation*
- these allow various anomalies:
  behavior not consistent with executing serially
- snapshot isolation, repeatable read,
  read committed, etc

Weak isolation vs weak consistency

- at strong consistency levels, these are the same: serializability, linearizability/strict serializability

- weaker isolation: operations aren't necessarily atomic

  ```
  A:   savings -= 100                              checking += 100
  B:                    read savings, checking
  ```

  but all agree on what sequence of events occurred!

- weaker consistency: operations are atomic, but different clients might see different order

  ```
  A sees: s -= 100; c += 100;     read s,c
  B sees: read s,c;      s -= 100; c += 100
  ```

Two-phase commit

- model: DB partitioned over different hosts, still only one copy of each data item; one coordinator per transaction
- during execution: use two-phase locking as before; acquire locks on all data read/written
- to commit, coordinator first sends prepare message to all shards; they respond prepare_ok or abort
  - if prepare_ok, they *must* be able to commit transaction later; past last chance to abort.
  - Usually requires writing to durable log.
- if all prepare_ok, coordinator sends commit to all; they write commit record and release logs

Is this the end of the story?

- Availability: what do we do if either some shard or the coordinator fails?
  - generally: 2PC is a blocking protocol, can’t make progress until it comes back up
  - some protocols to handle specific situations, e.g., coordinator recovery
- Performance: can we really afford to take locks and hold them for the entire commit process?

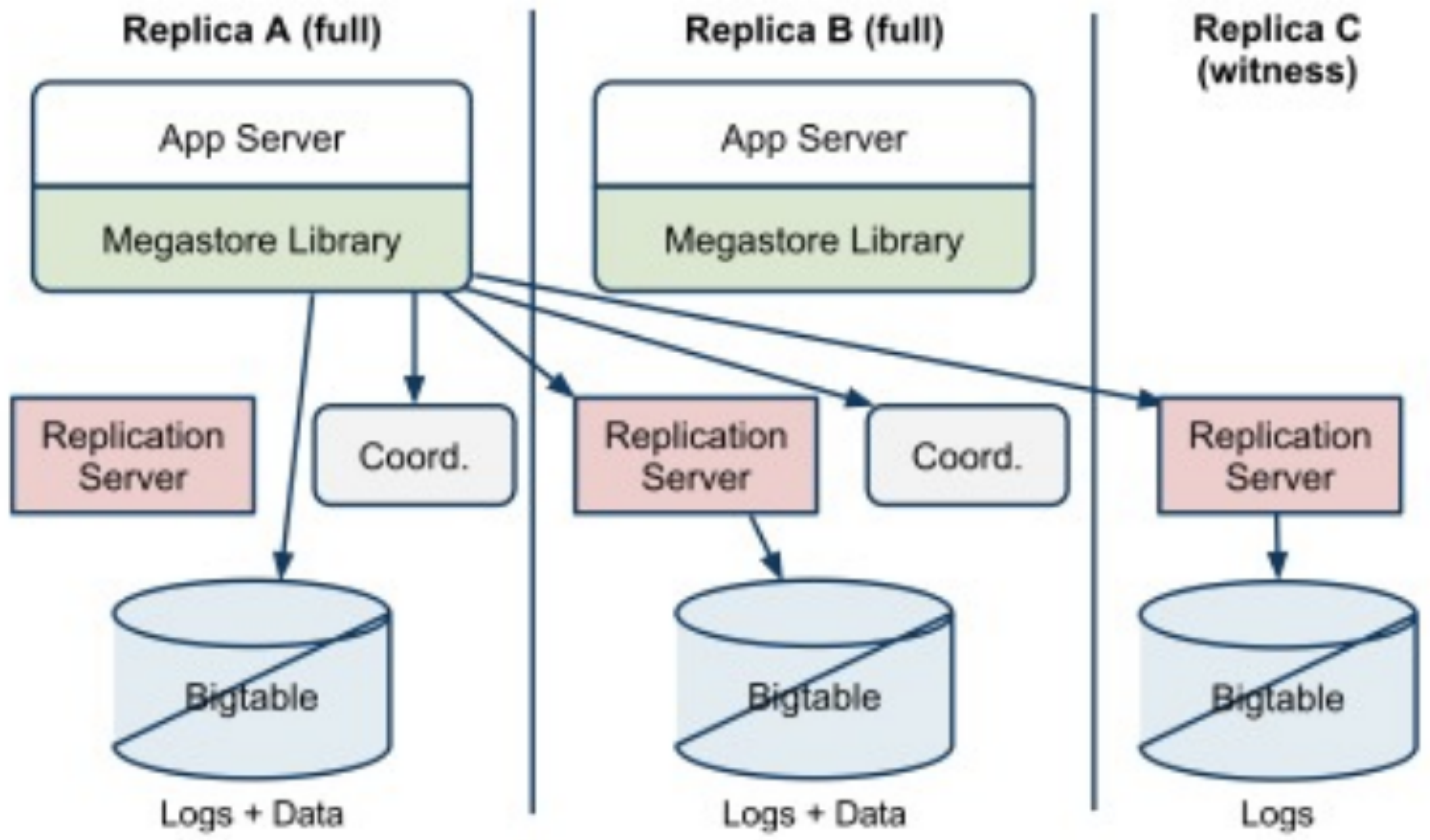
MegaStore

- Subsequent storage system to Bigtable
  - provide an interface that looks more like SQL
  - provide multi-object transactions
- Paper doesn’t make it clear how it was used, but:
  - later revealed: GMail, Picasa, Calendar
  - available through Google App Engine

Conventional wisdom

- Hard to have both consistency and performance in the wide area
    - consistency requires expensive communication to coordinate
- Hard to have both consistency and availability in the wide area
    - need 2PC across data; what about failures and partitions?
- One solution: relaxed consistency [next week]
- MegaStore: try to have it all!

MegaStore architecture

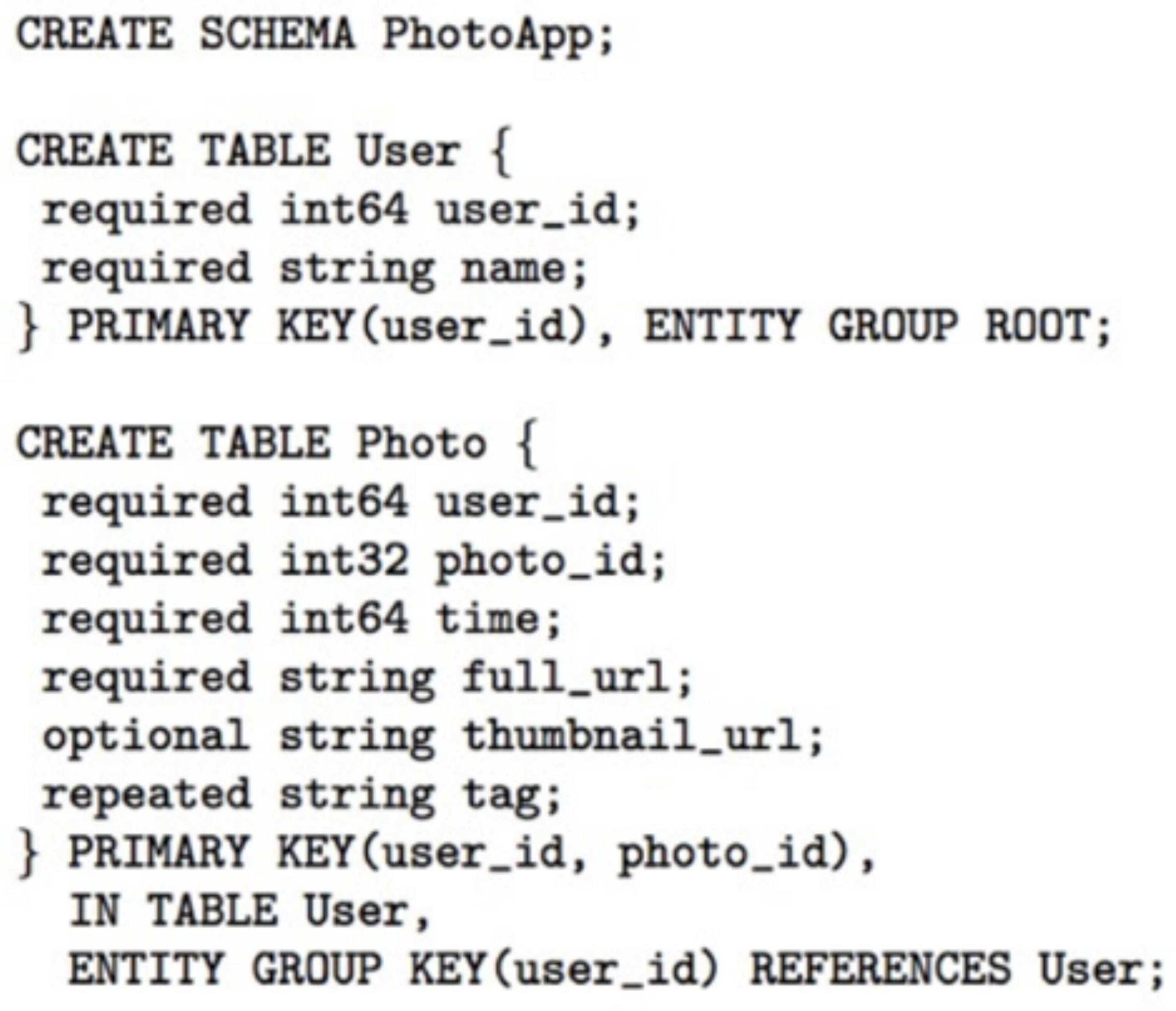


Setting

- browser web requests may arrive at any replica
    - i.e., application server at any replica
- no designated primary replica
- so could easily be concurrent transactions on same data from multiple replicas!

Data model

- Schema: set of tables containing set of entities containing set of properties
- Looks basically like SQL, but:
    - annotations about which data are accessed together (IN TABLE, etc)
    - annotations about which data can be updated together (entity groups)

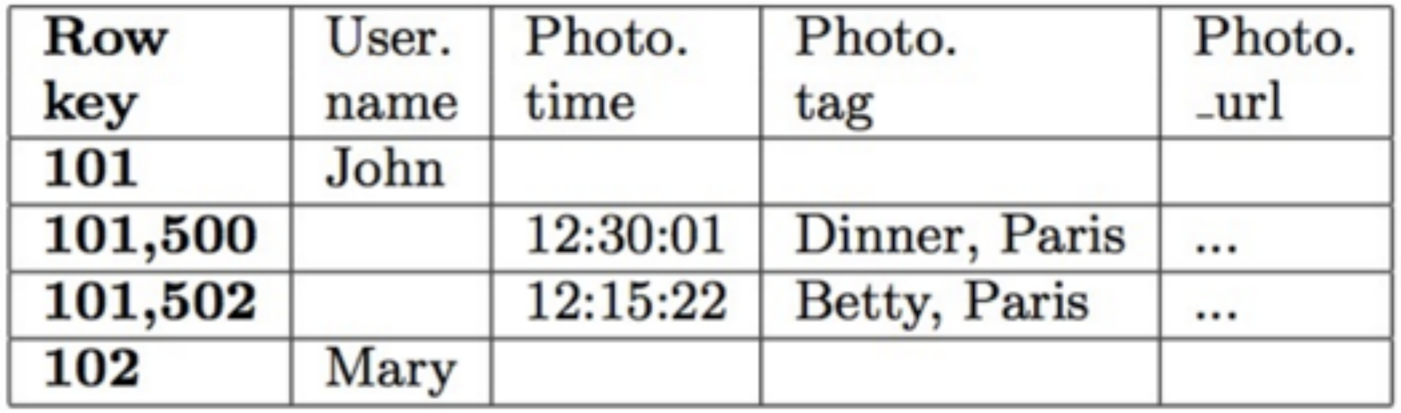


```
CREATE SCHEMA PhotoApp;

CREATE TABLE User {
 required int64 user_id;
 required string name;
} PRIMARY KEY(user_id), ENTITY GROUP ROOT;

CREATE TABLE Photo {
 required int64 user_id;
 required int32 photo_id;
 required int64 time;
 required string full_url;
 optional string thumbnail_url;
 repeated string tag;
} PRIMARY KEY(user_id, photo_id),
  IN TABLE User,
  ENTITY GROUP KEY(user_id) REFERENCES User;
```

Aside: a DB view

- Key principle of relational DBs: *data independence*
  users specify schema for data and what they want to do; DB figures out how to run it
- Consequence: performance is not transparent
  - easy to write a query that will take forever!
    especially in the distributed case!
- MegaStore argument is non-traditional
  - make performance choices explicit
  - make the user implement expensive things like joins themselves!

Translating schema to Bigtable



Row key	User. name	Photo. time	Photo. tag	Photo. _url
101	John			
101,500		12:30:01	Dinner, Paris	...
101,502		12:15:22	Betty, Paris	...
102	Mary			

- use row key as primary ID for Bigtable
- carefully select row keys so that related data is lexicographically close => same tablet
- embed related data that's accessed together

Entity groups

- transactions can only use data within a single entity group
- one row or a set of related rows, defined by application
  - e.g., all my gmail messages in 1 entity group
- example transaction:
  move message 321 from Inbox to Personal
- not possible as a transaction:
  deliver message to Dan, Haichen, Adriana

Implementing Transactions

- each entity group has a transaction log, stored in Bigtable
- data in Bigtable is the result of executing log operations
- to commit a transaction, create a log entry with its updates, use Paxos to agree that it’s the next entry in the log
- basically like lab 3, except that log entries are transactions instead of individual operations

Implementing Transactions

- find highest Paxos log entry number (N)
- read data from local Bigtable
- accumulate writes in temporary storage
- create a log entry w/ the set of writes
- use Paxos to agree that this is log entry N+1
- apply writes in log entry to Bigtable data

What does this mean?

- need to wait for inter-data-center messages to commit
- only a majority of replicas need to respond
- some replicas might be missing log entries; need to wait for them
- no leader in Paxos (unlike Chubby),
  so conflicts are definitely possible!

Concurrent transactions

- suppose two concurrent transactions try to commit, both modify X
- MegaStore must abort (at least) one!
- achieves this by catching conflicts during Paxos agreement
  - Paxos allows only one to write log entry N+1
  - the other application server will have to retry whole transaction

Concurrent transactions

- Does this work to prevent conflicts?
- Yes, but…
  it actually prevents **any** concurrency within an entity group
  - even if the transactions don’t actually conflict
  - traditional DB locking would do better:
    would allow concurrency on non-overlapping data

More Paxos tricks

- Distinguishing a proposer
  - Multi-Paxos and Chubby have a leader
  - MegaStore lets the last writer to an entity group be the distinguished proposer for the next entry
- Witnesses
  - Paxos requires 2f+1 replicas to tolerate f failures
  - but can have f of them that just record log entries, don’t actually apply the log
  - can be promoted to full replicas if one fails

What about transactions *across* entity groups?

- Don’t do that?
- Rely on two-phase commit?

Performance

- a couple transactions per second per entity group
- 10s of milliseconds for a read
- 100s of milliseconds for a write

- Is this OK?

Spanner

- Subsequent system [2012] from Google
- backend for the F1 database, which runs the ad system
- addresses limitations of MegaStore
  - no restriction on transaction scope (no entity groups)
  - more than one concurrent transaction at a time!
  - lock-free read-only transactions

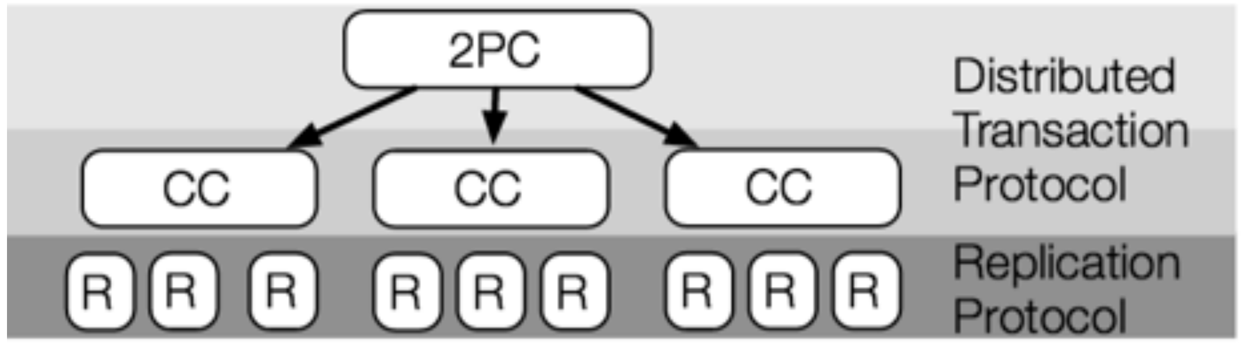
Example: social network

- simple schema: user posts, and friends lists
- but sharded across thousands of machines
- each replicated across multiple continents

Example: social network

- example: generate page of friends' recent posts
- what if I remove friend X, post mean comment?
    - maybe he sees old version of friends list,
      new version of my posts?
- How can we solve this with locking?
    - acquire read locks on friends list, and on each friend's posts
    - prevents them from being modified concurrently
    - but potentially really slow?

Spanner architecture



- Each shard is stored in a Paxos group
    - replicated across data centers
    - has a (relatively long-lived) leader
- Transactions span Paxos groups using 2PC
    - use 2PC for transactions
    - leader of each Paxos group tracks locks
    - one group leader becomes the 2PC coordinator, others participants

Some authors have claimed that general two-phase commit is too expensive to support, because of the performance or availability problems that it brings.
We believe it is better to have application programmers deal with performance problems due to overuse of transactions as bottlenecks arise, rather than always coding around the lack of transactions.
Running two-phase commit over Paxos mitigates the availability problems

Basic 2PC/Paxos approach

- during execution, read and write objects
  - contact the appropriate Paxos group leader, acquire locks
- client decides to commit, notifies the coordinator
  - coordinator contacts all shards, sends PREPARE message
  - they Paxos-replicate a prepare log entry (including locks),
  - vote either ok or abort
- if all shards vote OK, coordinator sends commit message
  - each shard Paxos-replicates commit entry
  - leader releases locks

Basic 2PC/Paxos approach

- Note that this is really the same as basic 2PC from before
- Just replaced writes to a log on disk with writes to a Paxos replicated log!
- It is linearizable (= strict serializable = externally consistent)

- So what’s left?
  - Lock-free read-only transactions

Lock-free r/o transactions

- Key idea: assign meaningful timestamp to transaction
  - such that timestamps are enough to order transactions meaningfully
- Keep a history of versions around on each node
- Then, reasonable to say:
  r/o transaction X reads at timestamp 10

TrueTime

- Common misconception: the magic here is fancy hardware (atomic clocks, GPS receivers)
  - this is actually relatively standard (see NTP)

- Actual key idea:
  expose the *uncertainty* in the clock value

TrueTime API

- interval = TT.now()
- “correct” time is “guaranteed” to be between interval.latest and interval.earliest
- What does this actually mean?

Implementing TrueTime

- time masters (GPS, atomic clocks) in each data center
- NTP or similar protocol syncs with multiple masters, rejects outliers
- TrueTime returns the local clock value, plus uncertainty
    - uncertainty = time since last sync * 200 usec/sec

Assigning transaction timestamps

- When in the basic protocol should we assign a transaction its timestamp?
- What timestamp should we give it?
- How do we know that timestamp is consistent with global time?

Basic 2PC/Paxos approach

- during execution, read and write objects
    - contact the appropriate Paxos group leader, acquire locks
- client decides to commit, notifies the coordinator
    - coordinator contacts all shards, sends PREPARE message
    - they Paxos-replicate a prepare log entry (including locks),
    - vote either ok or abort
- if all shards vote OK, coordinator sends commit message
    - each shard Paxos-replicates commit entry
    - leader releases locks

Assigning transaction timestamps

- When in the basic protocol should we assign a transaction its timestamp?
    - any time between when all locks acquired and first lock released
- What timestamp should we give it?
    - a time TrueTime thinks is in the future, TT.latest
- How do we know that timestamp is consistent with global time?
    - *commit wait*: wait until TrueTime knows it’s now in the past
    - thus: we know that when that timestamp was correct,
      we’re holding the locks

Timestamp details

- Coordinator actually collects several TrueTime timestamps:
  - timestamps from when each shard executed prepare
  - timestamp that coordinator received commit request from client
  - highest timestamp of any previous transaction
- Actually picks a timestamp greater than max of these, waits for it to be in the past

Commit wait

- What does this mean for performance?
- Larger TrueTime uncertainty bound
  => longer commit wait
- Longer commit wait => locks held longer
  => can’t process conflicting transactions
  => lower throughput
- i.e., if time is less certain, Spanner is slower!

What does this buy us?

- Can now do a read-only transaction at a particular timestamp, have it be meaningful
- Example: pick a timestamp T in the past,
  read version w/ timestamp T from all shards
    - since T is in the past, they will never accept a transaction with timestamp < T
    - don’t need locks while we do this!
- What if we want the current time?

What if TrueTime fails?

- Google argument: picked using engineering considerations, less likely than a total CPU failure
- But what if it went wrong anyway?
  - can cause very long commit wait periods
  - can break ordering guarantees, no longer externally consistent
  - but system will always be serializable: gathering many timestamps and taking the max is a Lamport clock