Big Data Infrastructure

CS 489/698 Big Data Infrastructure (Winter 2016)

Week 9: Mutable State (2/2)

March 17, 2016

Jimmy Lin
David R. Cheriton School of Computer Science
University of Waterloo

These slides are available at http://lintool.github.io/bigdata-2016w/

The Fundamental Problem

- We want to keep track of *mutable* state in a *scalable* manner
- Assumptions:
  - State organized in terms of many "records"
  - State unlikely to fit on single machine, must be distributed

(note: much of this material belongs in a distributed systems or databases course)

Motivating Scenarios

- Money shouldn't be created or destroyed:
  - Alice transfers $100 to Bob and $50 to Carol
  - The total amount of money after the transfer should be the same
- Phantom shopping cart:
  - Bob removes an item from his shopping cart…
  - Item still remains in the shopping cart
  - Bob refreshes the page a couple of times… item finally gone

Motivating Scenarios

- People you don't want seeing your pictures:
  - Alice removes mom from list of people who can view photos
  - Alice posts embarrassing pictures from Spring Break
  - Can mom see Alice's photo?
- Why am I still getting messages?
  - Bob unsubscribes from mailing list
  - Message sent to mailing list right after
  - Does Bob receive the message?

Three Core Ideas

- Partitioning (sharding)
  - For scalability
  - For latency

Need distributed transactions!

- Replication
  - For robustness (availability)
  - For throughput

Need replica coherence protocol!

- Caching
  - For latency

Need cache coherence protocol!

How to address?

Source: images.wikia.com/batman/images/b/b1/Bat_Signal.jpg

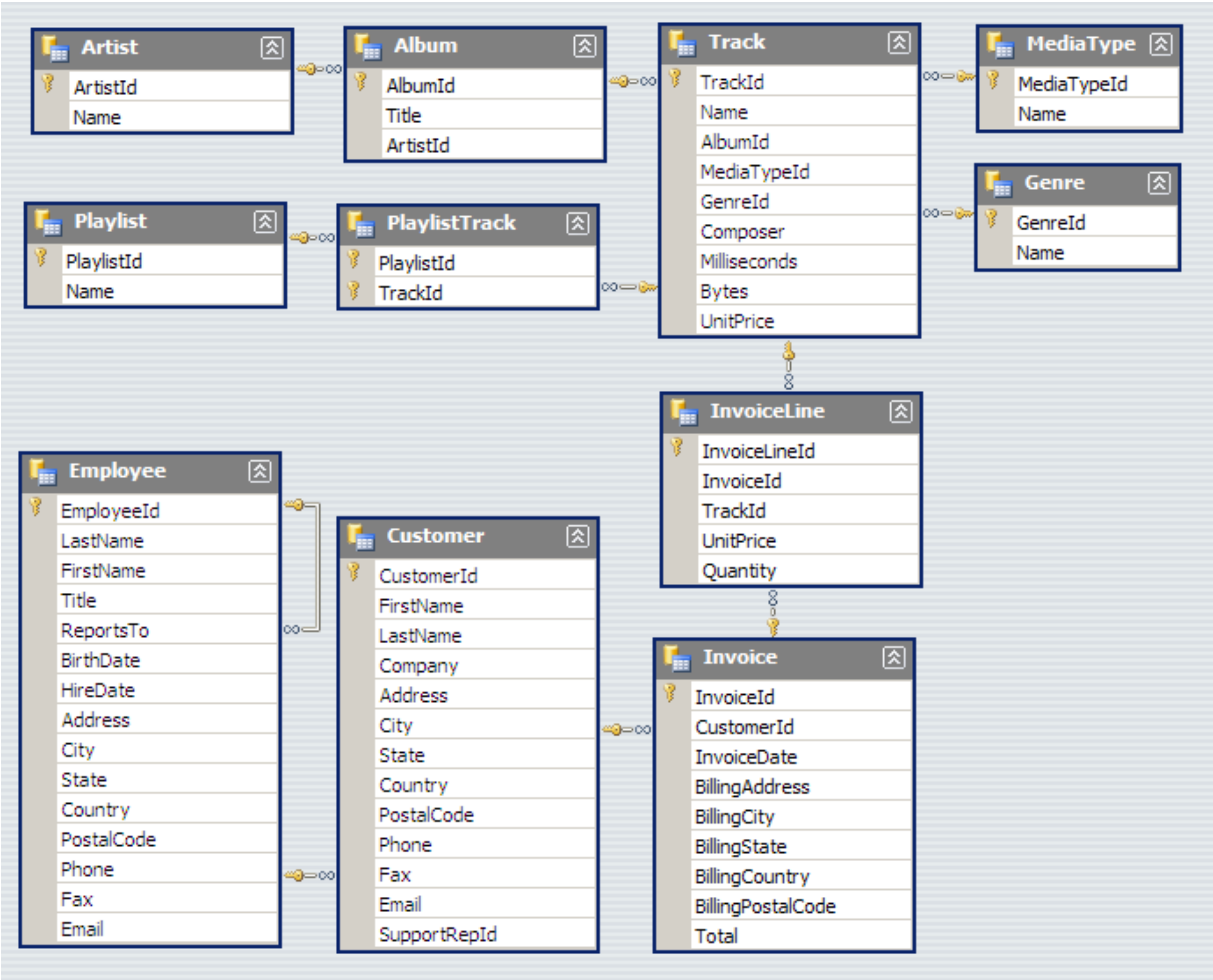
What do RDBMSes provide?

- Relational model with schemas
- Powerful, flexible query language
- Transactional semantics: ACID
- Rich ecosystem, lots of tool support

RDBMSes: Pain Points

Source: www.flickr.com/photos/spencerdahl/6075142688/

#1: Must design up front, painful to evolve



Note: Flexible design doesn't mean *no* design!


#2: Pay for ACID!
Source: Wikipedia (Tortoise)


#3: Cost!
Source: www.flickr.com/photos/gnusinn/3080378658/

What do RDBMSes provide?

- Relational model with schemas
- Powerful, flexible query language
- Transactional semantics: ACID
- Rich ecosystem, lots of tool support

What if we want *a la carte*?

Source: www.flickr.com/photos/vidiot/18556565/

Features *a la carte*?

- What if I'm willing to give up consistency for scalability?
- What if I'm willing to give up the relational model for something more flexible?
- What if I just want a cheaper solution?

Three Core Ideas

- Partitioning (sharding)
  - For scalability
  - For latency

Need distributed transactions!

- Replication
  - For robustness (availability)
  - For throughput

Need replica coherence protocol!

- Caching
  - For latency

Need cache coherence protocol!

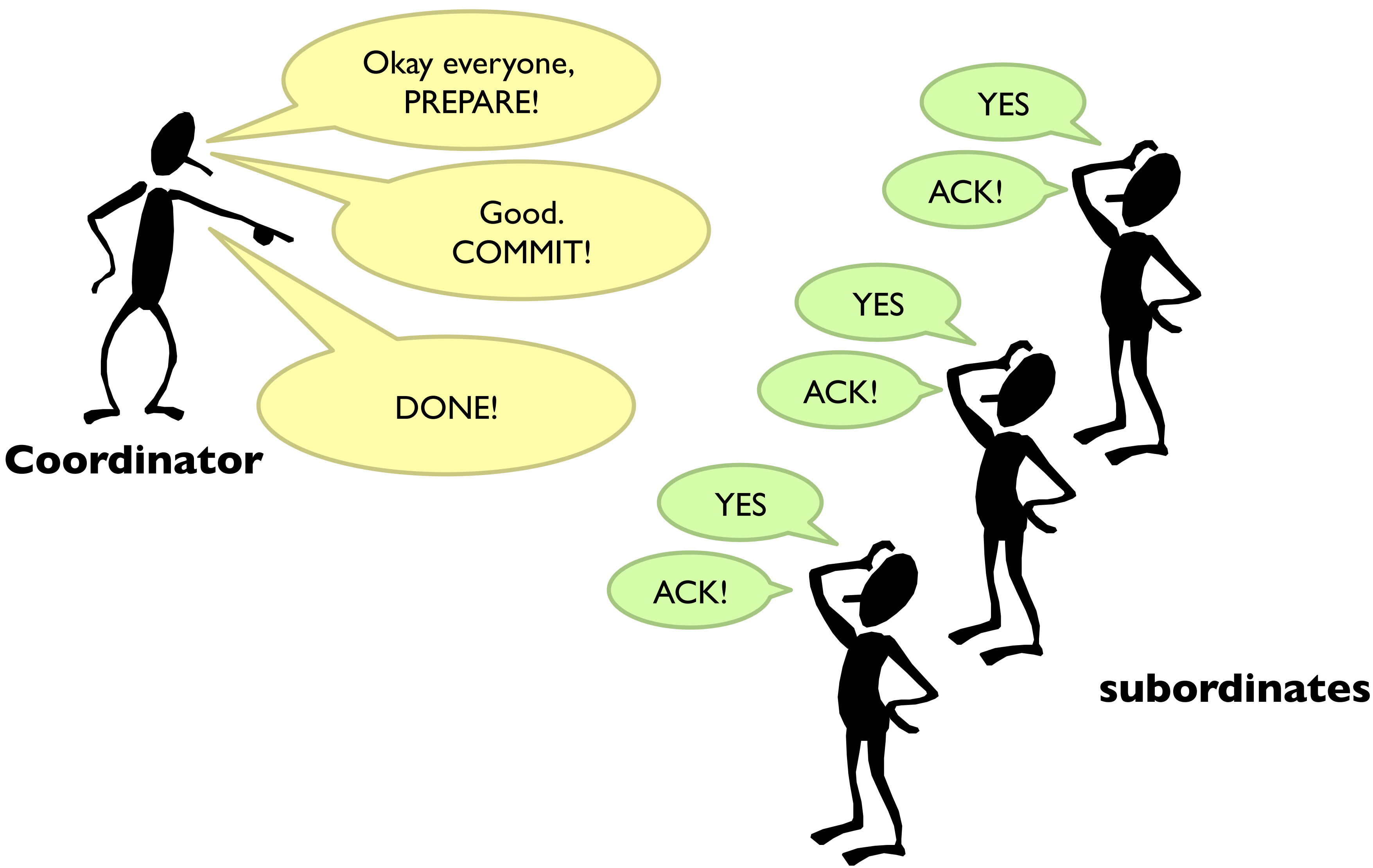
Motivating application?

How do RDBMSes do it?

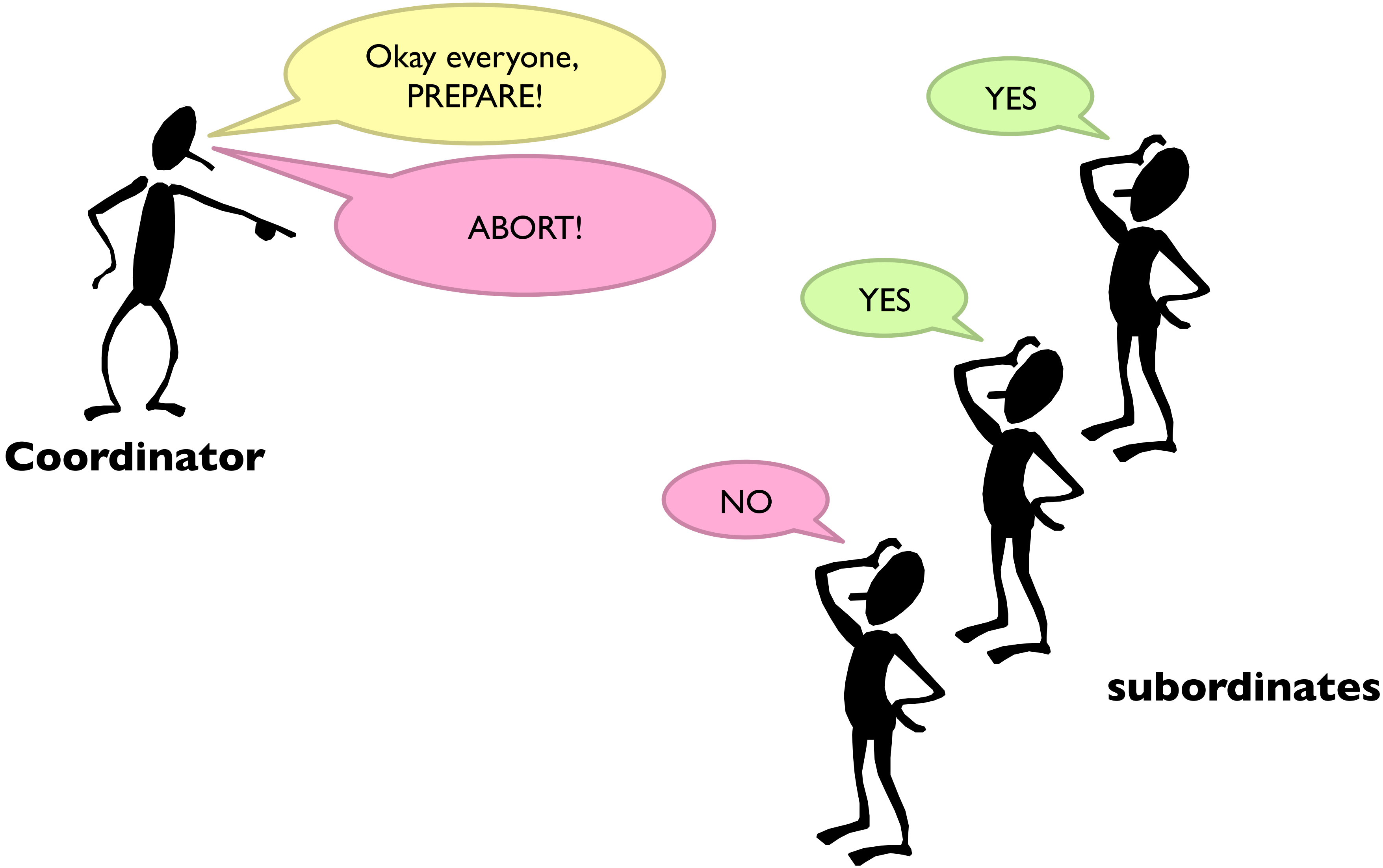
- Transactions on a single machine: (relatively) easy!
- Partition tables to keep transactions on a single machine
  - Example: partition by user
- What about transactions that require multiple machine?
  - Example: transactions involving multiple users

Solution: Two-Phase Commit

2PC: Sketch

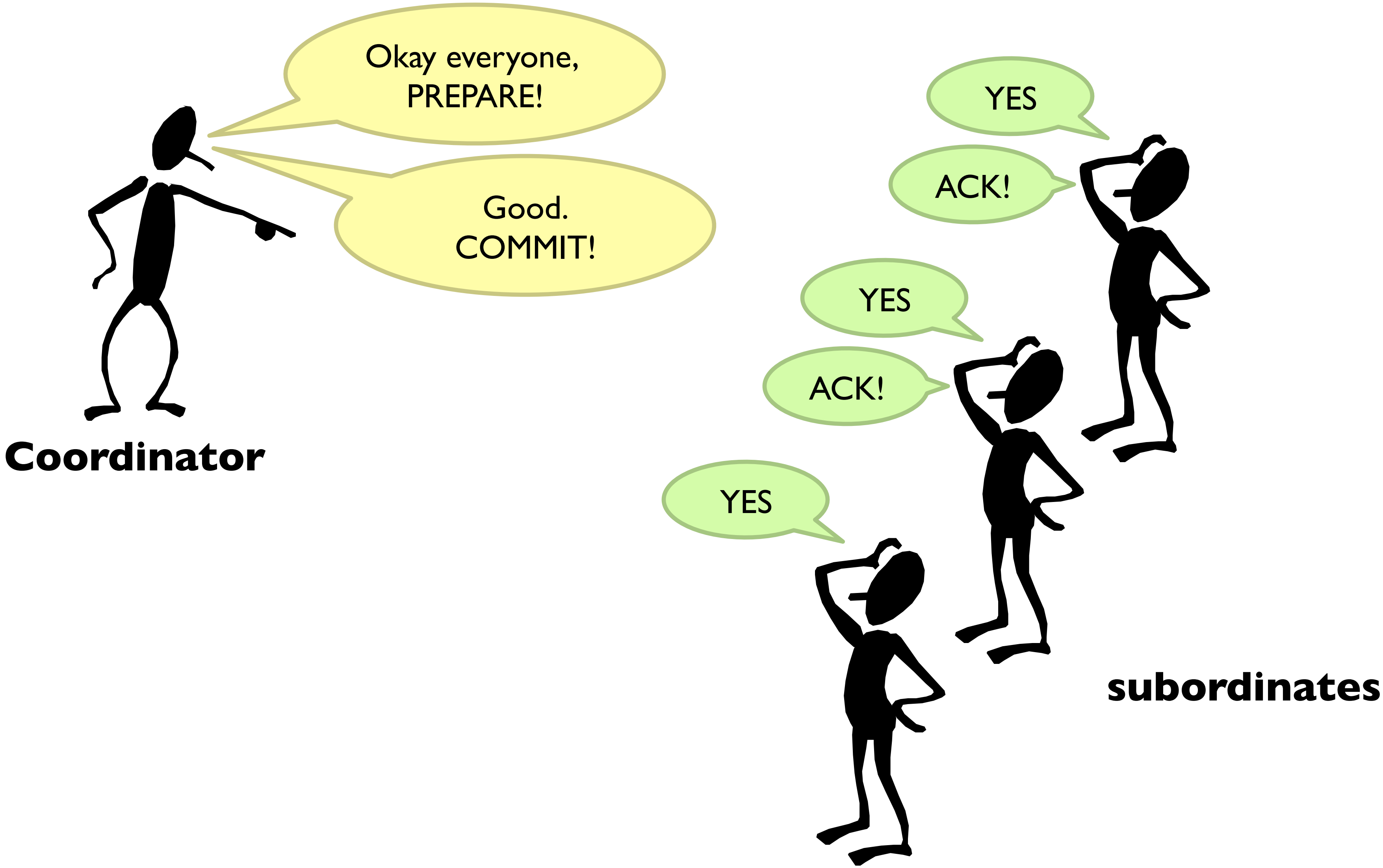

Okay everyone,
PREPARE!
Good.
COMMIT!
DONE!
Coordinator
YES
ACK!
YES
ACK!
YES
ACK!
subordinates

2PC: Sketch



Okay everyone, PREPARE!
ABORT!
YES
YES
NO
Coordinator
subordinates

2PC: Sketch


Okay everyone,
PREPARE!
Good.
COMMIT!
YES
ACK!
YES
ACK!
YES
Coordinator
subordinates

2PC: Assumptions and Limitations

- Assumptions:
  - Persistent storage and write-ahead log at every node
  - WAL is never permanently lost
- Limitations:
  - It's blocking and slow
  - What if the coordinator dies?

Beyond 2PC: Paxos!
(details beyond scope of this course)

Remember this?

Key-Value Stores: Operations

- Very simple API:
  - Get – fetch value associated with key
  - Put – set value associated with key
- Optional operations:
  - Multi-get
  - Multi-put
  - Range queries
- Consistency model:
  - Atomic puts (usually)
  - Cross-key operations: who knows?

"Unit of Consistency"

- Single record:
  - Relatively straightforward
  - Complex application logic to handle multi-record transactions
- Arbitrary transactions:
  - Requires 2PC
- Middle ground: entity groups
  - Groups of entities that share affinity
  - Co-locate entity groups
  - Provide transaction support within entity groups
  - Example: user + user's photos + user's posts etc.

Where have we learned this trick before?

Three Core Ideas

- Partitioning (sharding)
  - For scalability
  - For latency

Need distributed transactions!

- Replication
  - For robustness (availability)
  - For throughput

Need replica coherence protocol!

- Caching
  - For latency

Need cache coherence protocol!

CAP “Theorem” (Brewer, 2000)

Consistency

Availability

Partition tolerance

… pick two

CAP Tradeoffs

- CA = consistency + availability
  - E.g., parallel databases that use 2PC
- AP = availability + tolerance to partitions
  - E.g., DNS, web caching

Is this helpful?

- CAP not really even a “theorem” because vague definitions
  - More precise formulation came a few years later

Source: Abadi (2012) Consistency Tradeoffs in Modern Distributed Database System Design. IEEE Computer, 45(2):37-42

Abadi Says…

- CP makes no sense!
- CAP says, in the presence of P, choose A or C
  - But you'd want to make this tradeoff even when there is no P
- Fundamental tradeoff is between consistency and latency
  - Not available = (very) long latency

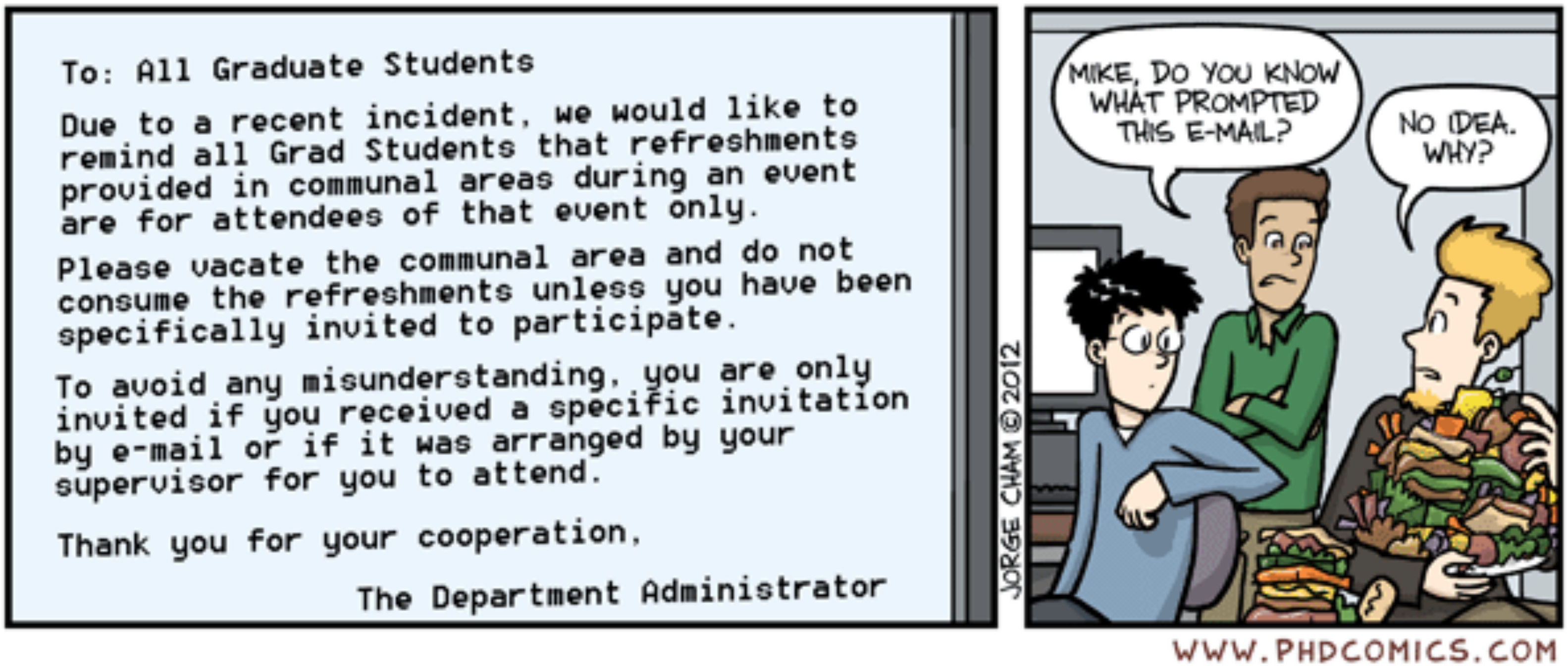
Replication possibilities

- Update sent to all replicas at the same time
  - To guarantee consistency you need something like Paxos
- Update sent to a master
  - Replication is synchronous
  - Replication is asynchronous
  - Combination of both
- Update sent to an arbitrary replica

All these possibilities involve tradeoffs!
"eventual consistency"

Move over, CAP

- PACELC ("pass-elk")
- PAC
  - If there's a partition, do we choose A or C?
- ELC
  - Otherwise, do we choose latency or consistency?



Morale of the story: there's no free lunch!

Source: www.phdcomics.com/comics/archive.php?comicid=1475

HBase

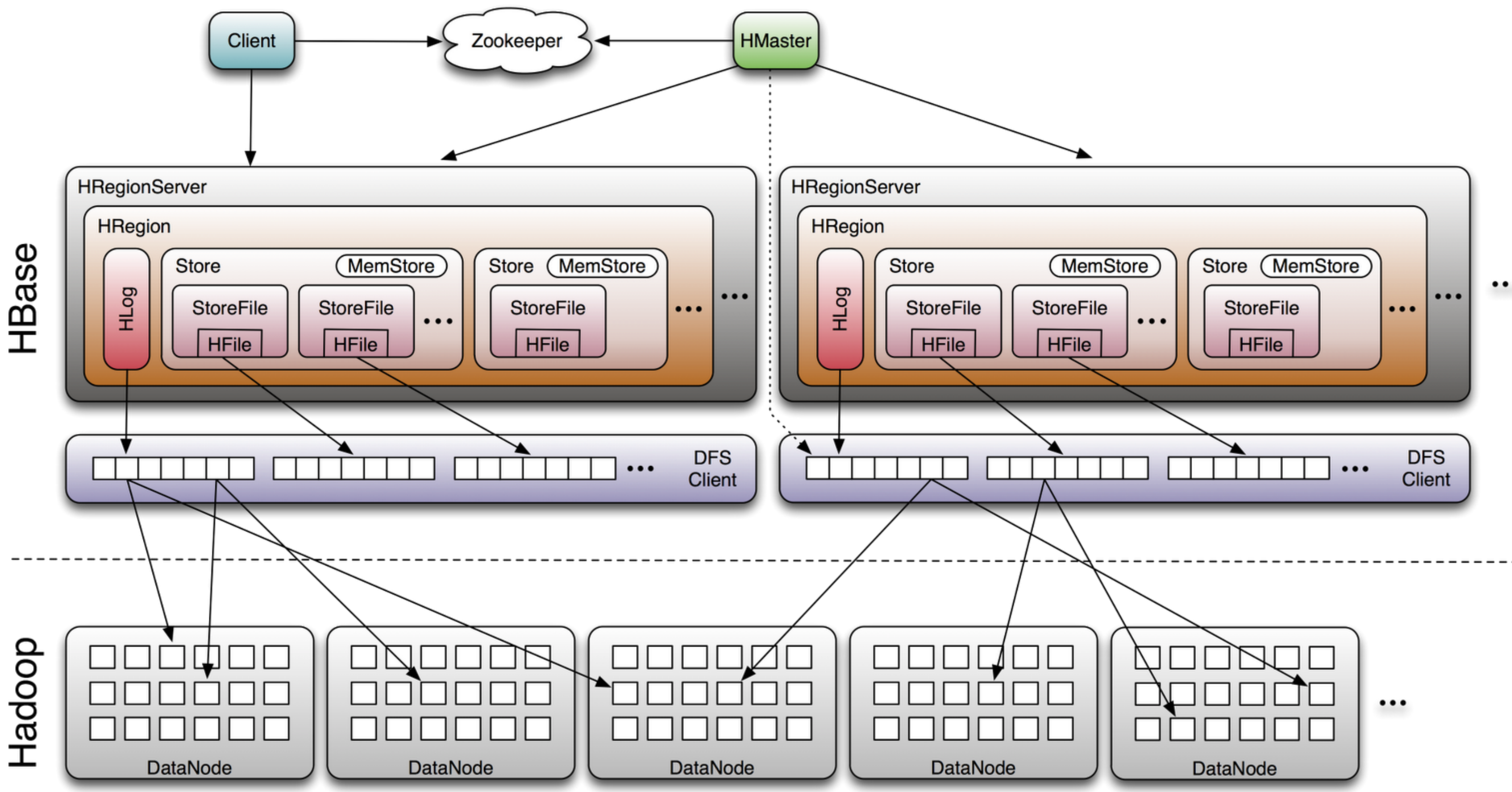


Image Source: http://www.larsgeorge.com/2009/10/hbase-architecture-101-storage.html

Three Core Ideas

- Partitioning (sharding)
  - For scalability
  - For latency

Need distributed transactions!

- Replication
  - For robustness (availability)
  - For throughput

Need replica coherence protocol!

- Caching
  - For latency

Need cache coherence protocol!

This is really hard!


Now imagine multiple datacenters…
What’s different?
Source: Google

Three Core Ideas

- Partitioning (sharding)
  - For scalability
  - For latency

Need distributed transactions!

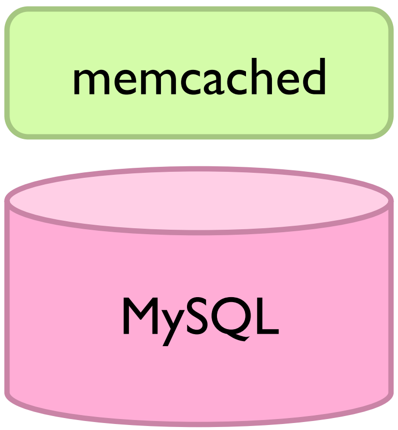
- Replication
  - For robustness (availability)
  - For throughput

Need replica coherence protocol!

- Caching
  - For latency

Need cache coherence protocol!

Facebook Architecture



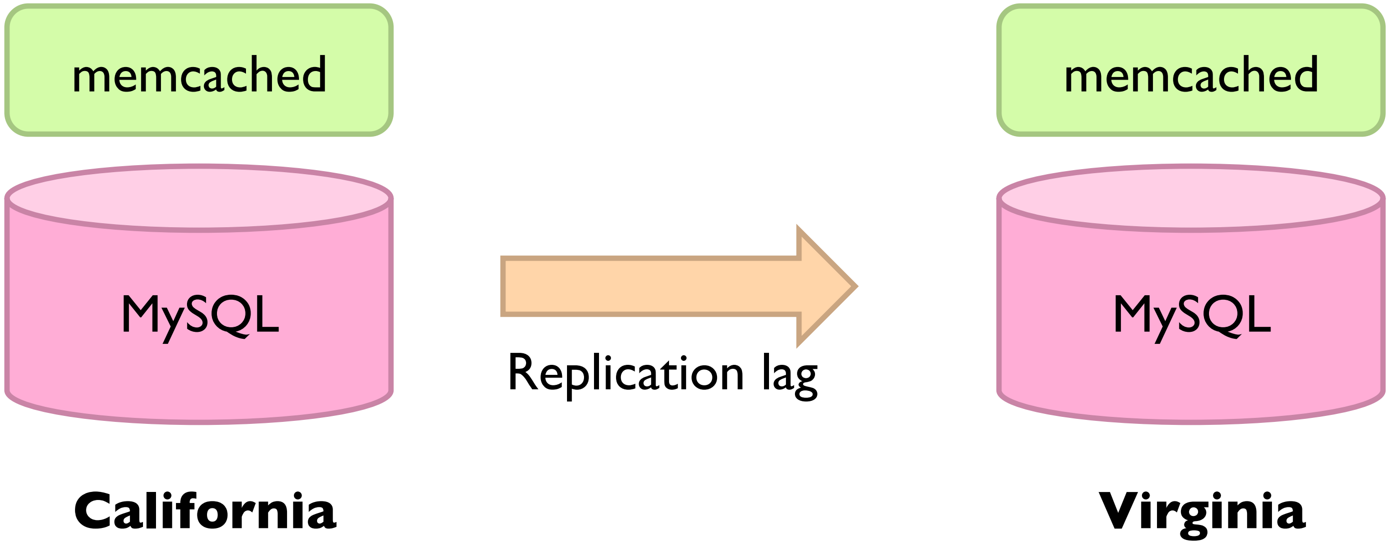
Read path:
Look in memcached
Look in MySQL
Populate in memcached

Write path:
Write in MySQL
Remove in memcached

Subsequent read:
Look in MySQL ✔
Populate in memcached

Source: www.facebook.com/note.php?note_id=23844338919

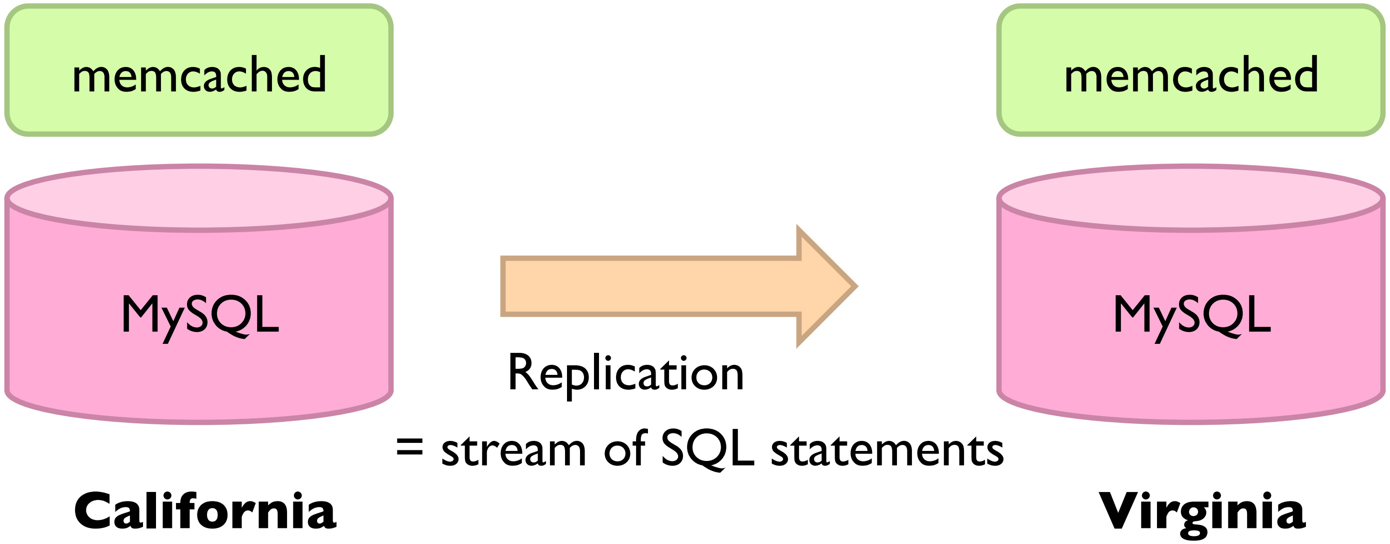
Facebook Architecture: Multi-DC



1. User updates first name from “Jason” to “Monkey”.
2. Write “Monkey” in master DB in CA, delete memcached entry in CA and VA.
3. Someone goes to profile in Virginia, read VA slave DB, get “Jason”.
4. Update VA memcache with first name as “Jason”.
5. Replication catches up. “Jason” stuck in memcached until another write!

Source: www.facebook.com/note.php?note_id=23844338919

Facebook Architecture



Solution: Piggyback on replication stream, tweak SQL

```
REPLACE INTO profile (`first_name`) VALUES ('Monkey’)
WHERE `user_id`='jsobel' MEMCACHE_DIRTY 'jsobel:first_name'
```

Source: www.facebook.com/note.php?note_id=23844338919

Three Core Ideas

- Partitioning (sharding)
  - For scalability
  - For latency

Need distributed transactions!

- Replication
  - For robustness (availability)
  - For throughput

Need replica coherence protocol!

- Caching
  - For latency

Need cache coherence protocol!

Yahoo's PNUTS

- Yahoo's globally distributed/replicated key-value store
- Provides *per-record* timeline consistency
  - Guarantees that all replicas provide all updates in same order
- Different classes of reads:
  - Read-any: may time travel!
  - Read-critical(required version): monotonic reads
  - Read-latest

PNUTS: Implementation Principles

- Each record has a single master
  - Asynchronous replication across datacenters
  - Allow for synchronous replicate within datacenters
  - All updates routed to master first, updates applied, then propagated
  - Protocols for recognizing master failure and load balancing
- Tradeoffs:
  - Different types of reads have different latencies
  - Availability compromised when master fails and partition failure in protocol for transferring of mastership

Three Core Ideas

- Partitioning (sharding)
  - For scalability
  - For latency
- Replication
  - For robustness (availability)
  - For throughput
- Caching
  - For latency

Need distributed transactions!

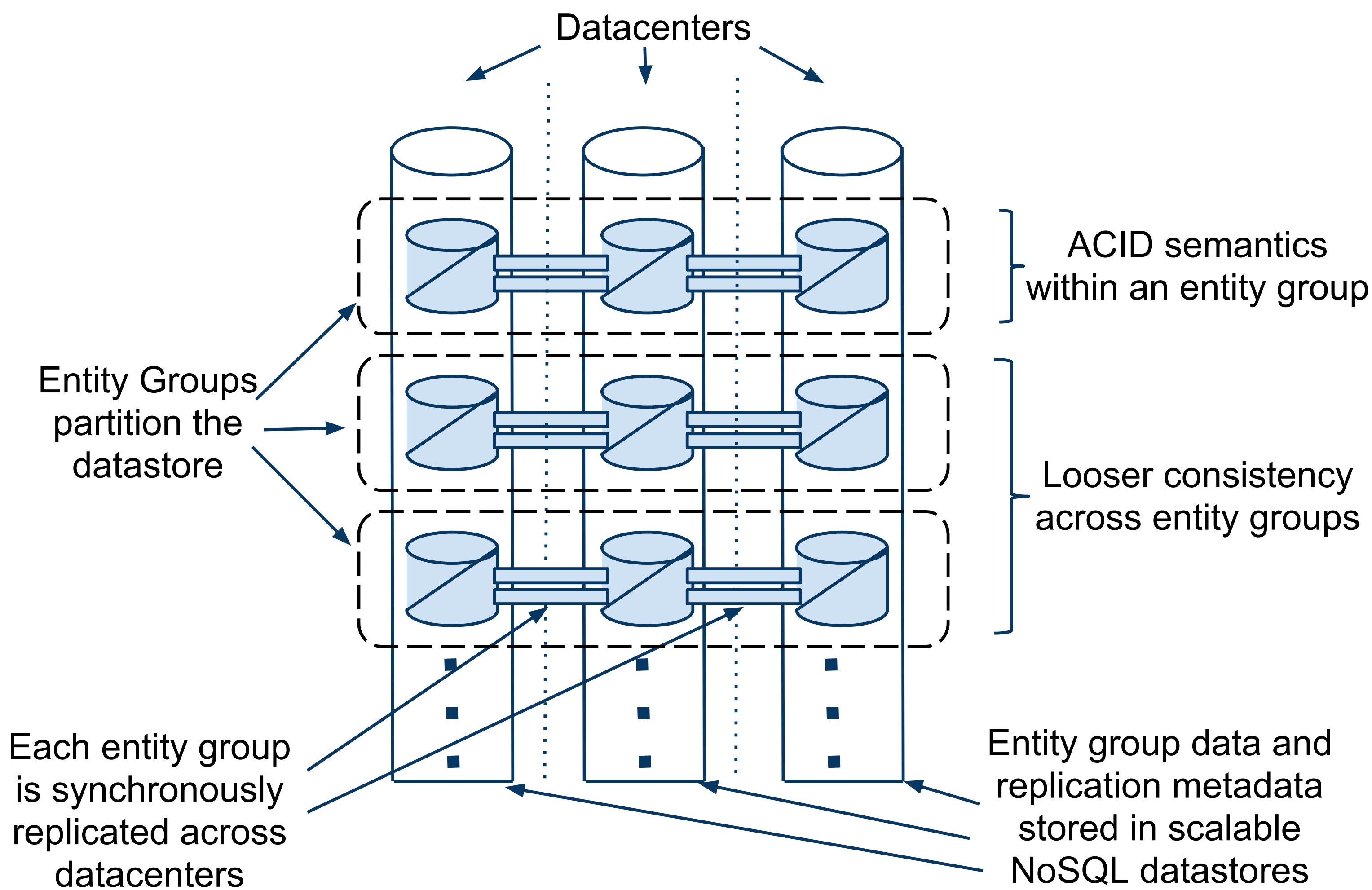
Have our cake and eat it too?

Need replica coherence protocol!

Need cache coherence protocol!

Source: Wikipedia (Cake)

Google's Megastore



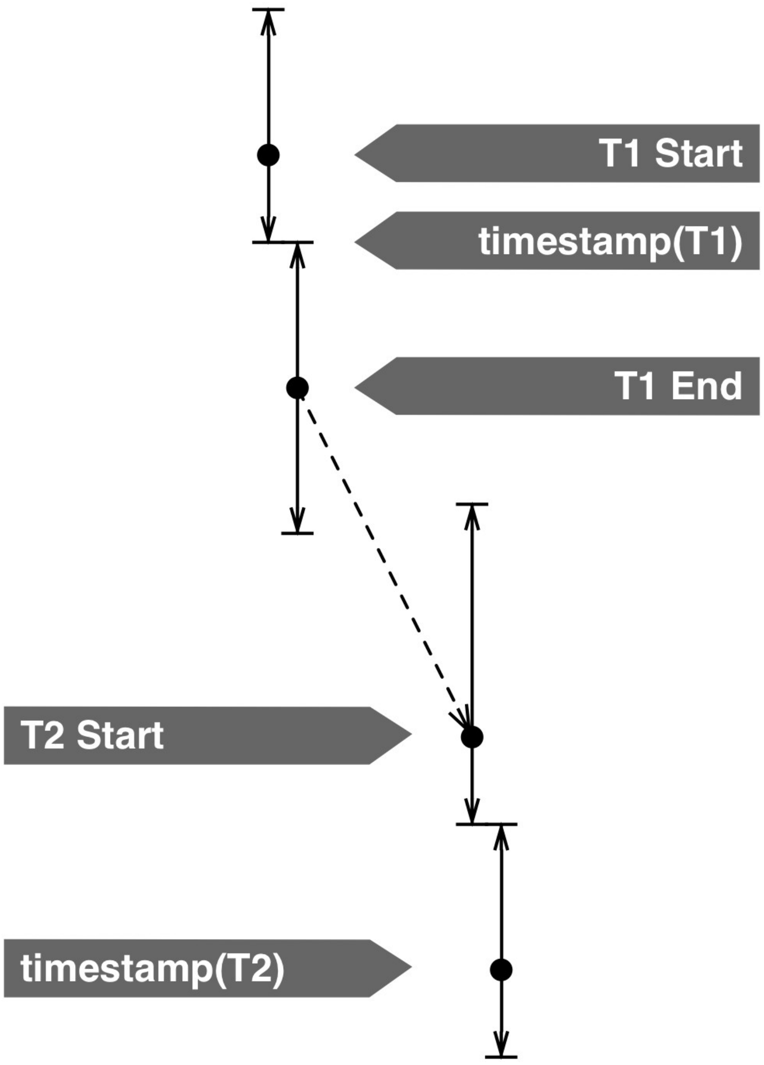
Source: Baker et al., CIDR 2011

Google's Spanner

- Features:
  - Full ACID translations across multiple datacenters, across continents!
  - External consistency (= linearizability):
    system preserves *happens-before* relationship among transactions
- How?
  - Given write transactions A and B, if A *happens-before* B, then timestamp(A) < timestamp(B)

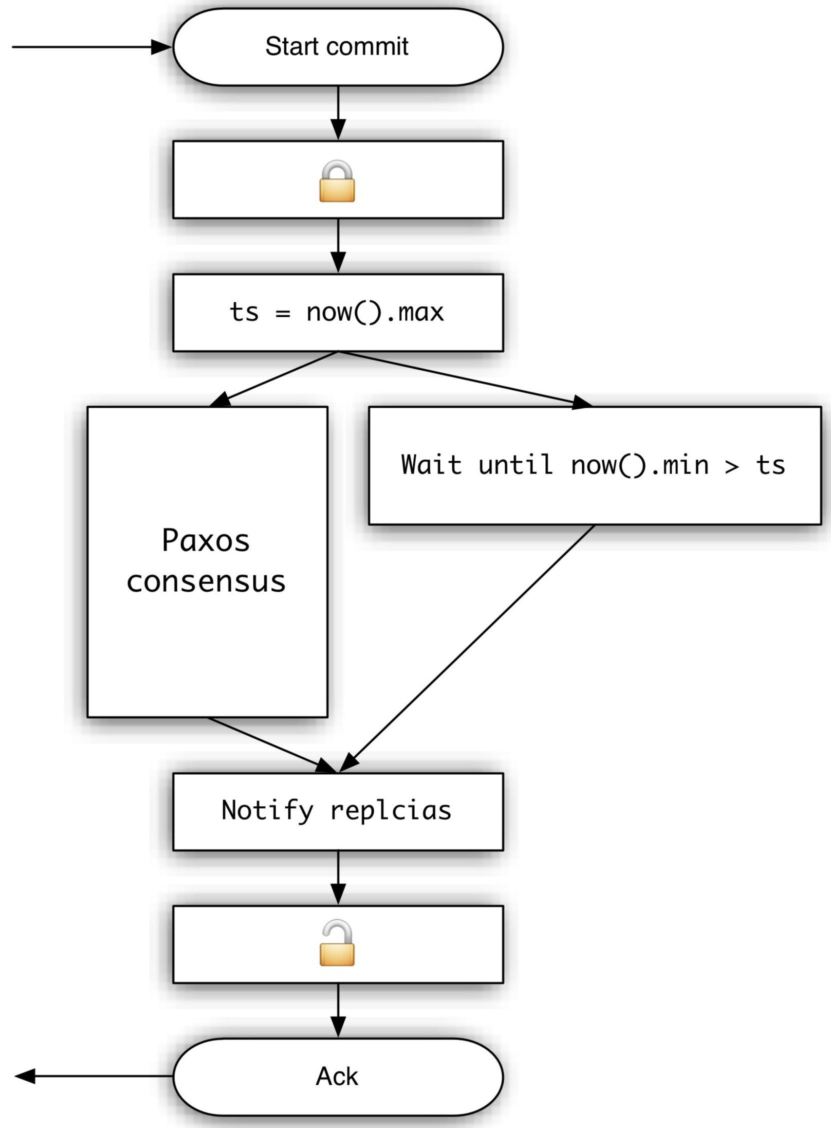
Source: Llyod, 2012

Why this works



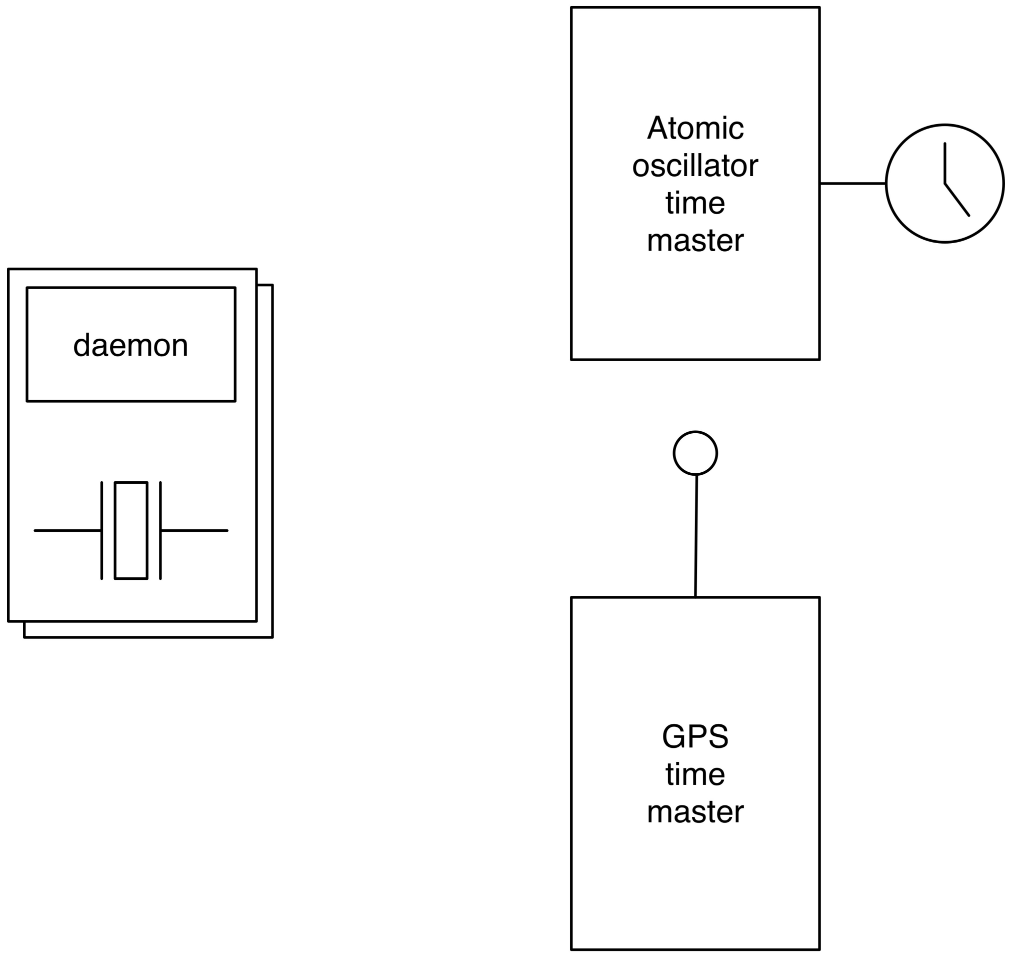
Source: Llyod, 2012

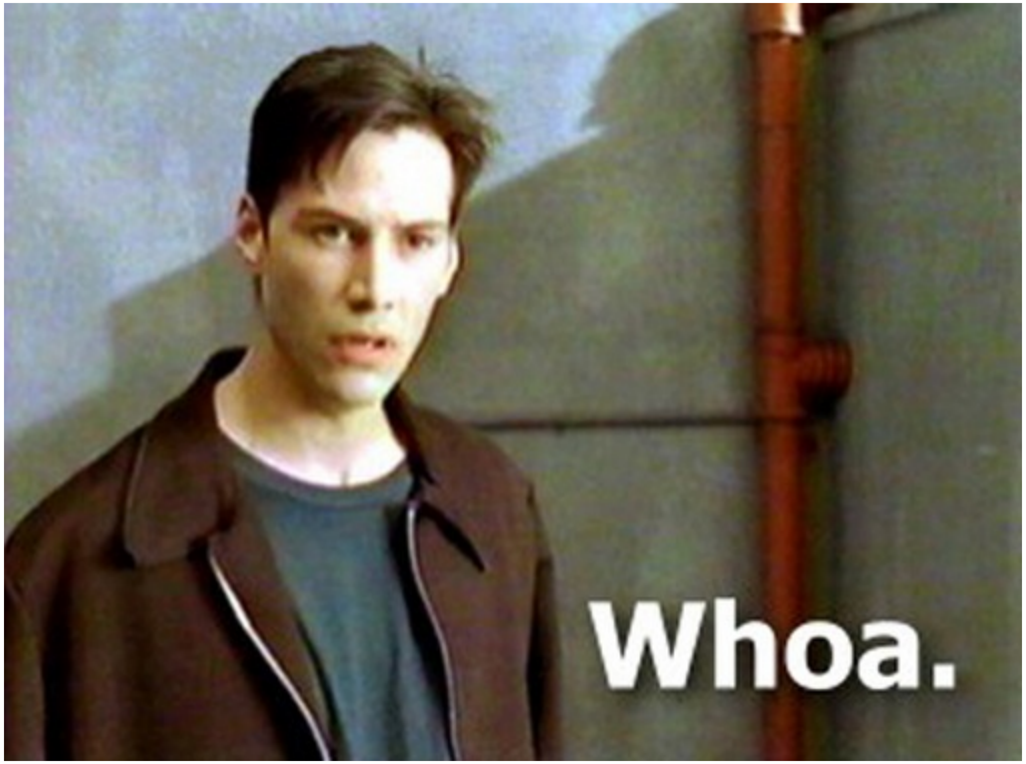
TrueTime → write timestamps



Source: Llyod, 2012

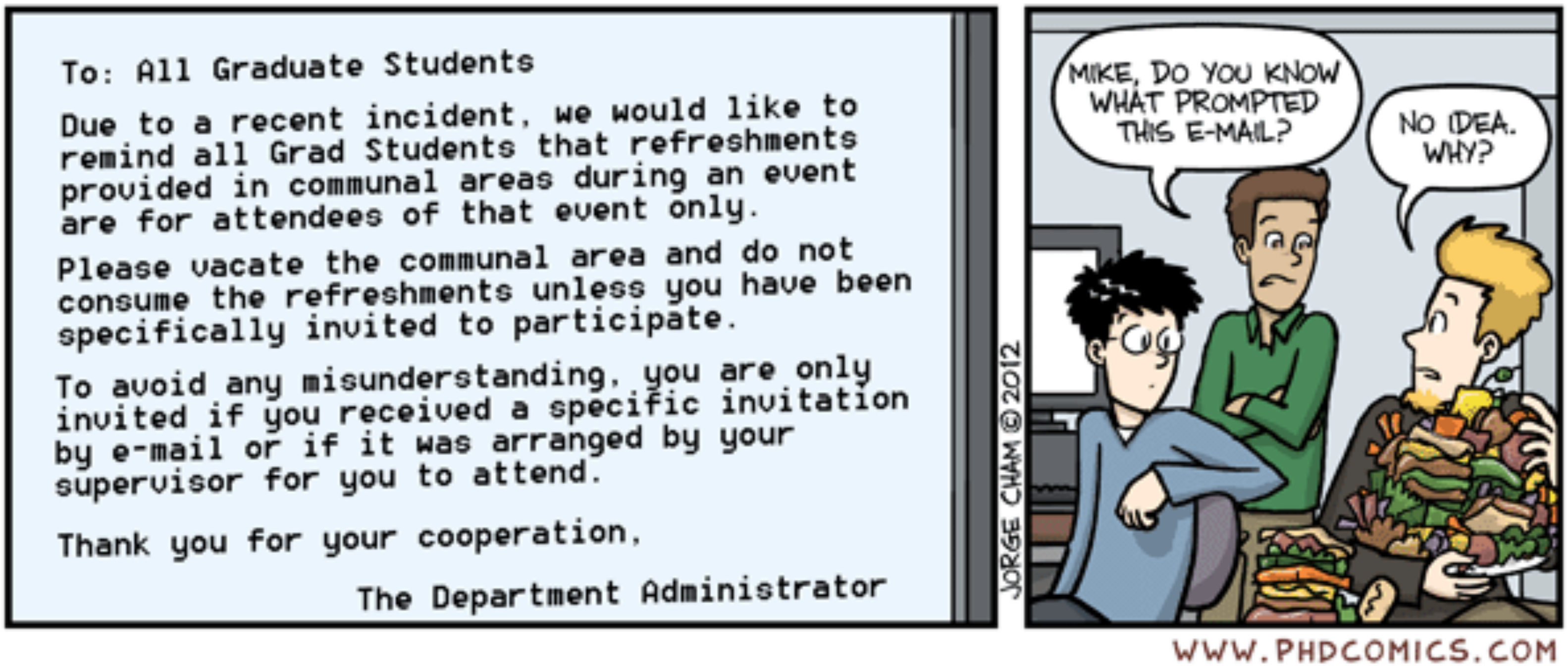
TrueTime





What's the catch?

Source: The Matrix



Morale of the story: there's no free lunch!

Source: www.phdcomics.com/comics/archive.php?comicid=1475

Questions?

Source: Wikipedia (Japanese rock garden)