Data-Intensive Computing with MapReduce

Session 12: NoSQL

Jimmy Lin
University of Maryland
Thursday, April 18, 2013

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
- MapReduce won't do!

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

Three Core Ideas

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

We got 99 problems…

- How do we keep replicas in sync?
- How do we synchronize transactions across multiple partitions?
- What happens to the cache when the underlying data changes?


Relational Databases
… to the rescue!
Source: images.wikia.com/batman/images/b/b1/Bat_Signal.jpg

What do RDBMSes provide?

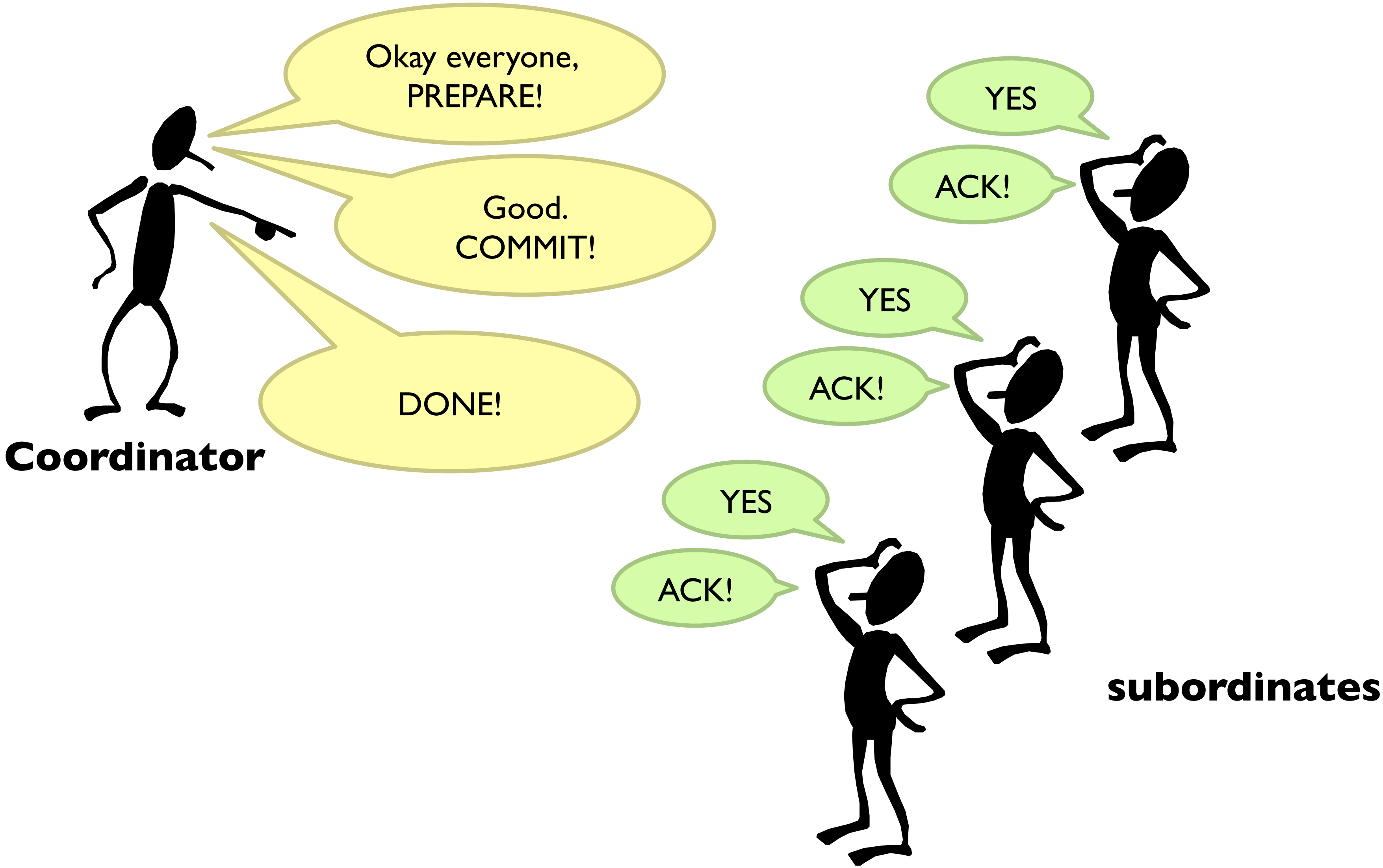
- Relational model with schemas
- Powerful, flexible query language
- Transactional semantics: ACID
- Rich ecosystem, lots of tool support

How do RDBMSes do it?

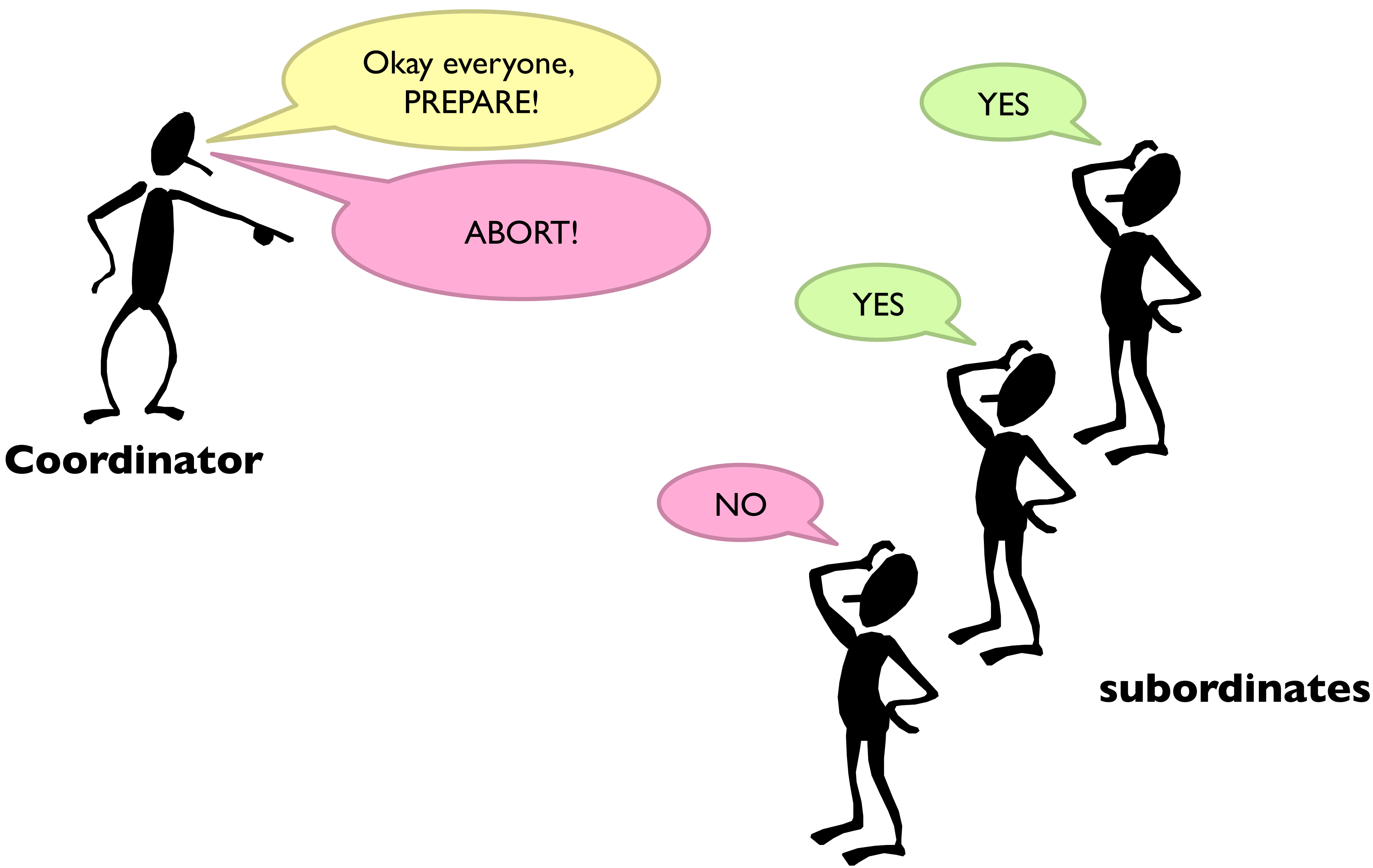
- Transactions on a single machine: (relatively) easy!
- Partition tables to keep transactions on a single machine
  - Example: partition by user
- What about transactions that require multiple machine?
  - Example: transactions involving multiple users

Solution: Two-Phase Commit

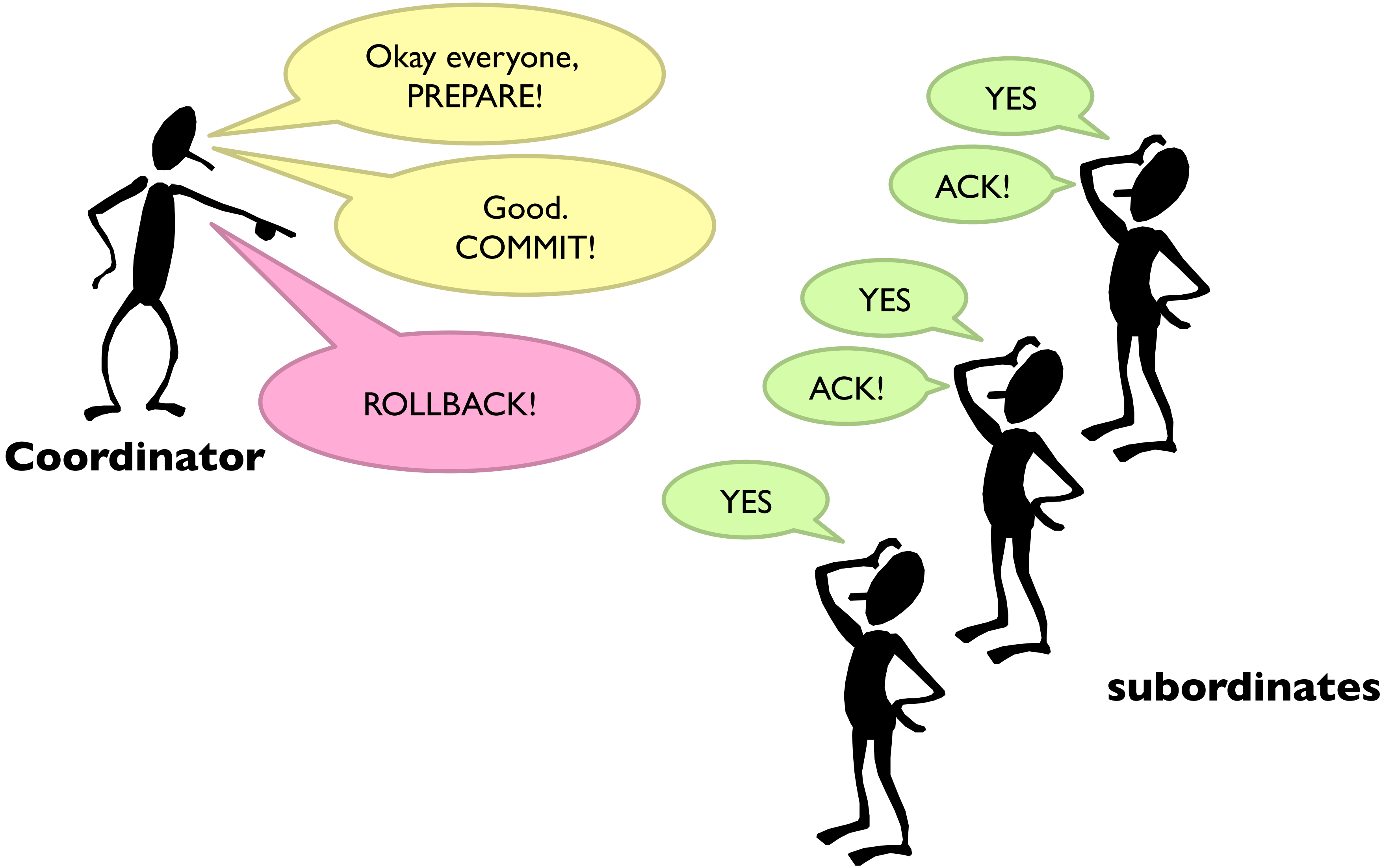
2PC: Sketch


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!
ROLLBACK!
Coordinator
YES
ACK!
YES
ACK!
YES
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?

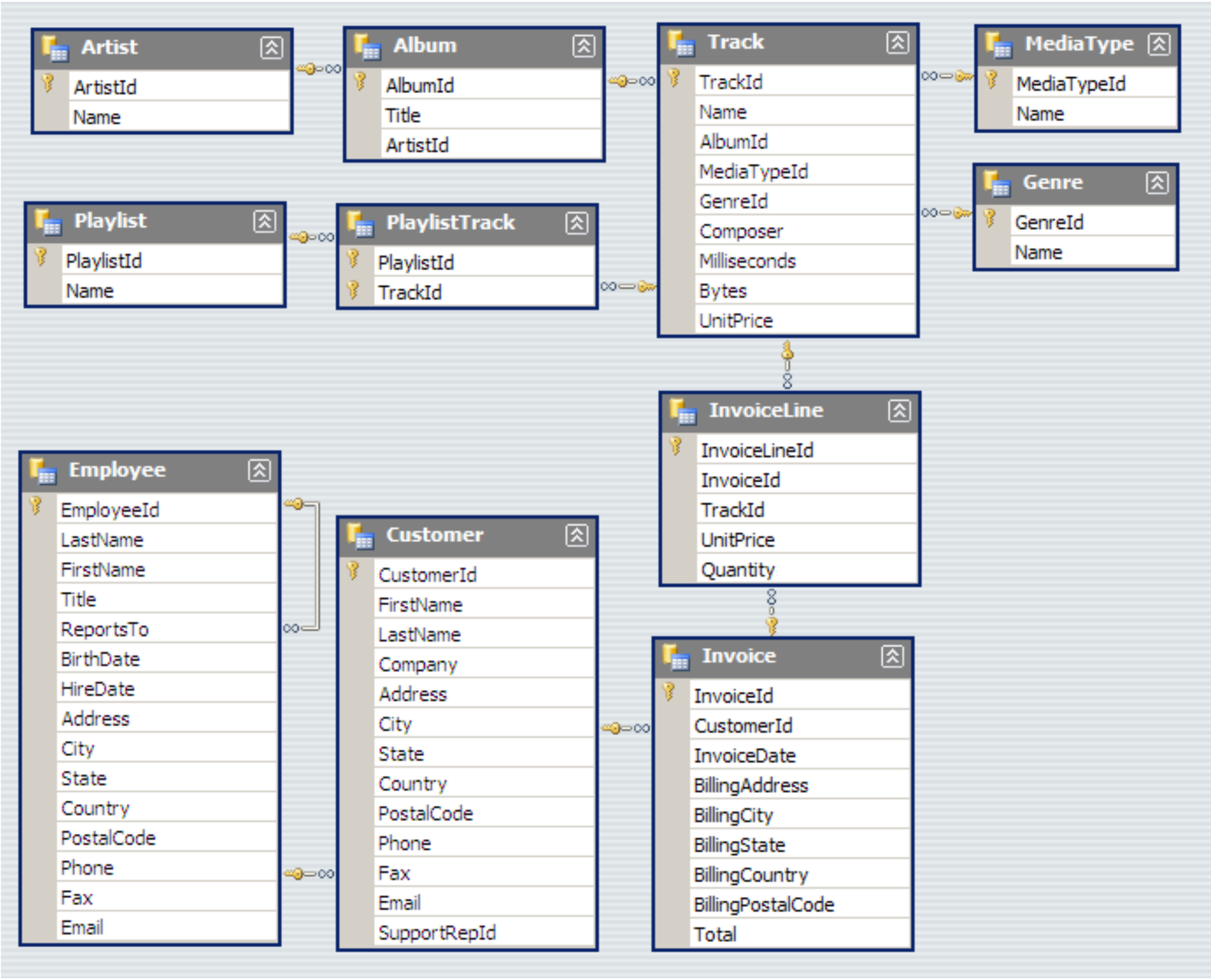
Solution: Paxos!

(details beyond scope of this course)

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: 2PC is slow!
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

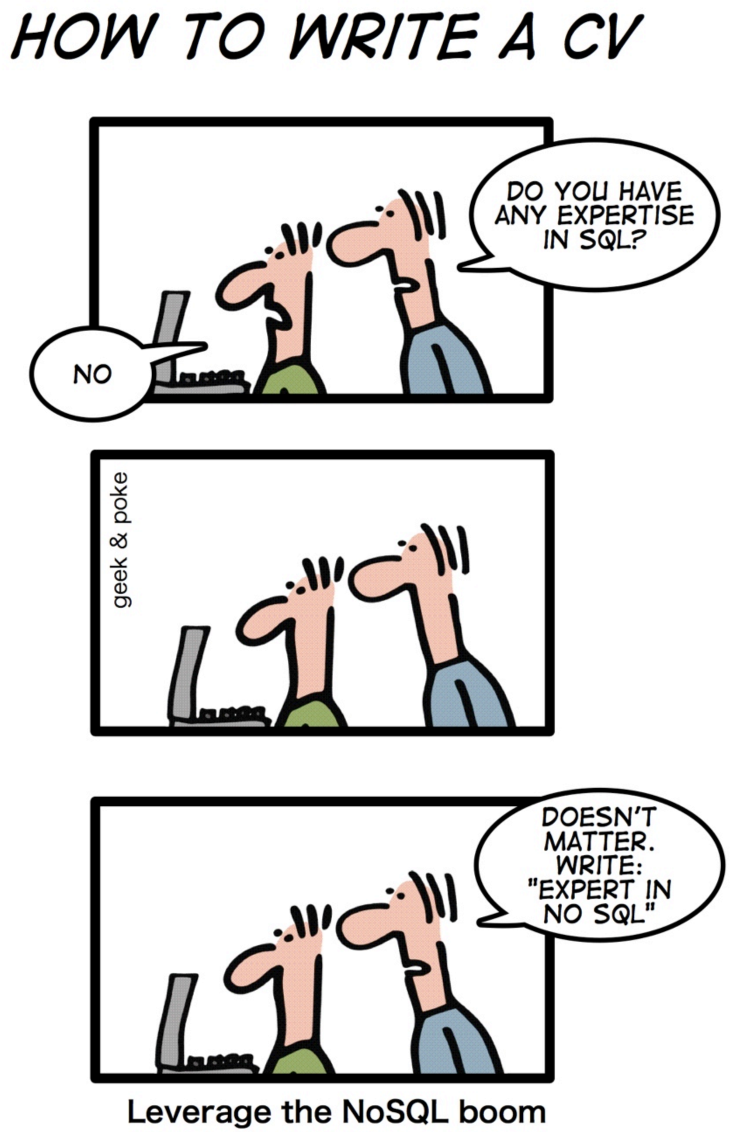
What if we want *a la carte*?

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?

Enter… NoSQL!



Source: geekandpoke.typepad.com/geekandpoke/2011/01/nosql.html

NoSQL (Not only SQL)

1. Horizontally scale "simple operations"
2. Replicate/distribute data over many servers
3. Simple call interface
4. Weaker concurrency model than ACID
5. Efficient use of distributed indexes and RAM
6. Flexible schemas

Source: Cattell (2010). Scalable SQL and NoSQL Data Stores. *SIGMOD Record*.

(Major) Types of NoSQL databases

- Key-value stores
- Column-oriented databases
- Document stores
- Graph databases

Key-Value Stores

Source: Wikipedia (Keychain)

Key-Value Stores: Data Model

- Stores associations between keys and values
- Keys are usually primitives
  - For example, ints, strings, raw bytes, etc.
- Values can be primitive or complex: usually opaque to store
  - Primitives: ints, strings, etc.
  - Complex: JSON, HTML fragments, etc.

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?

Key-Value Stores: Implementation

- Non-persistent:
  - Just a big in-memory hash table
- Persistent
  - Wrapper around a traditional RDBMS

What if data doesn't fit on a single machine?

Simple Solution: Partition!

- Partition the key space across multiple machines
  - Let's say, hash partitioning
  - For *n* machines, store key *k* at machine *h(k)* mod *n*
- Okay… But:
  1. How do we know which physical machine to contact?
  2. How do we add a new machine to the cluster?
  3. What happens if a machine fails?

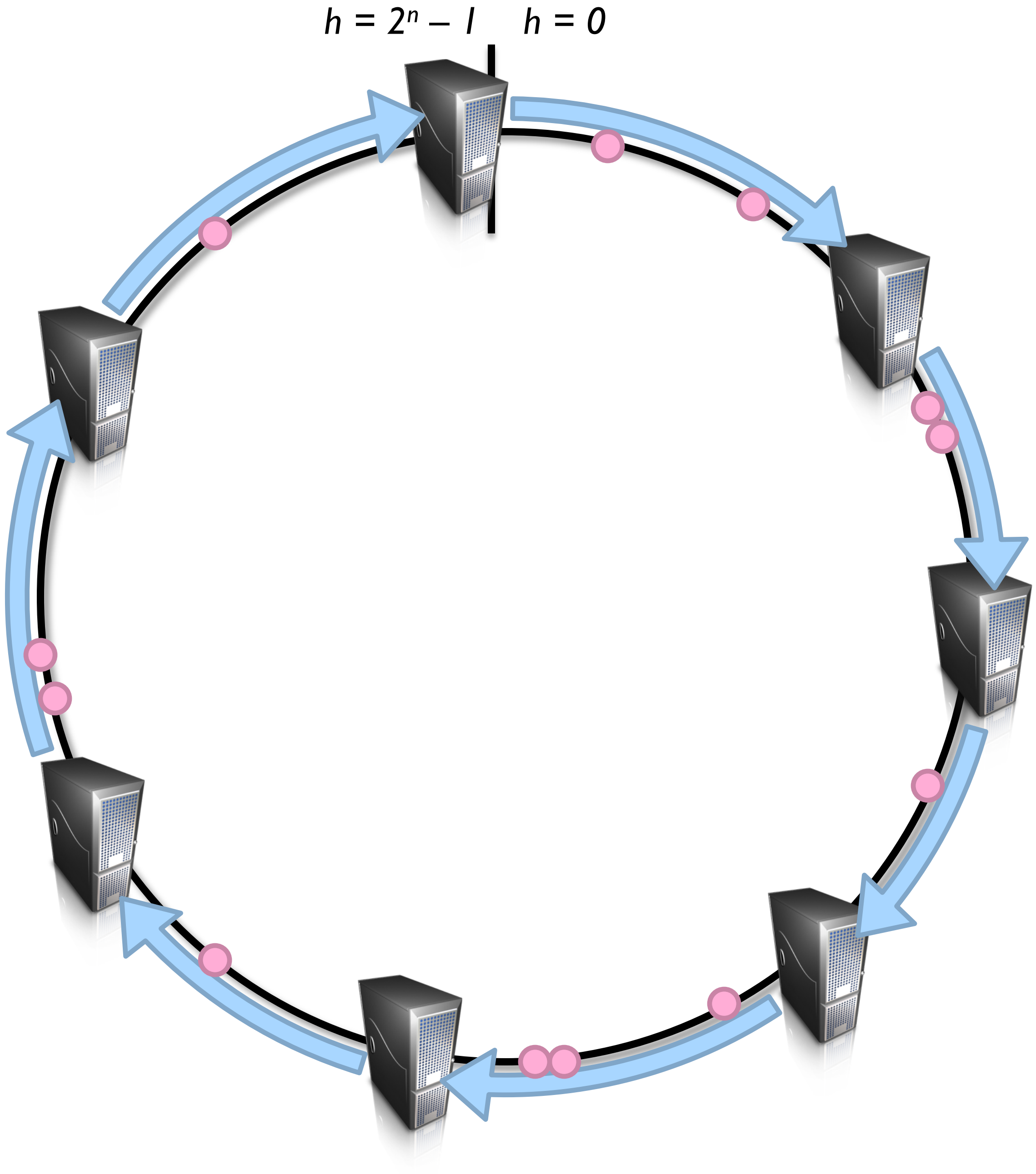
See the problems here?

Clever Solution

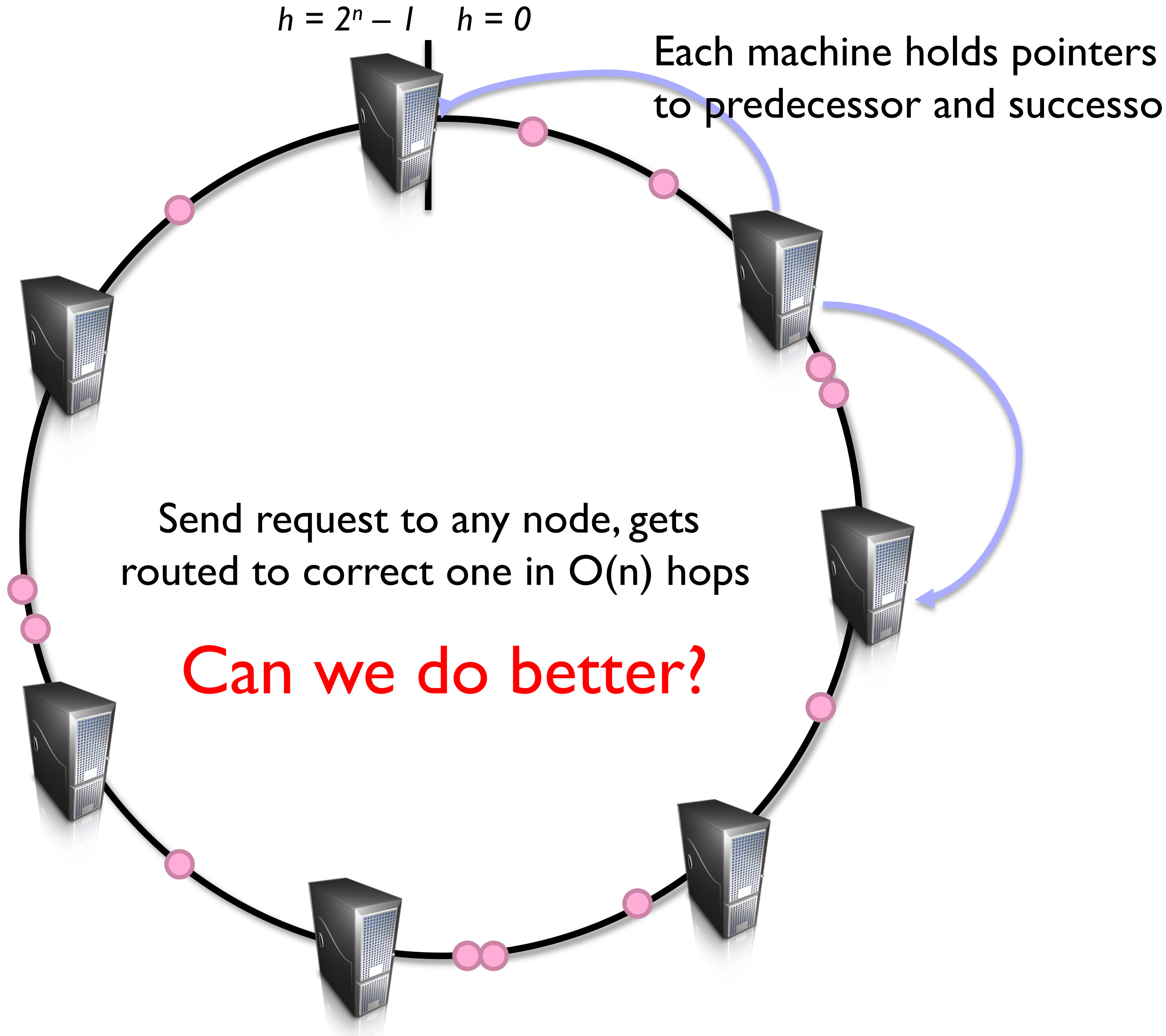
- Hash the keys
- Hash the machines also!

Distributed hash tables!

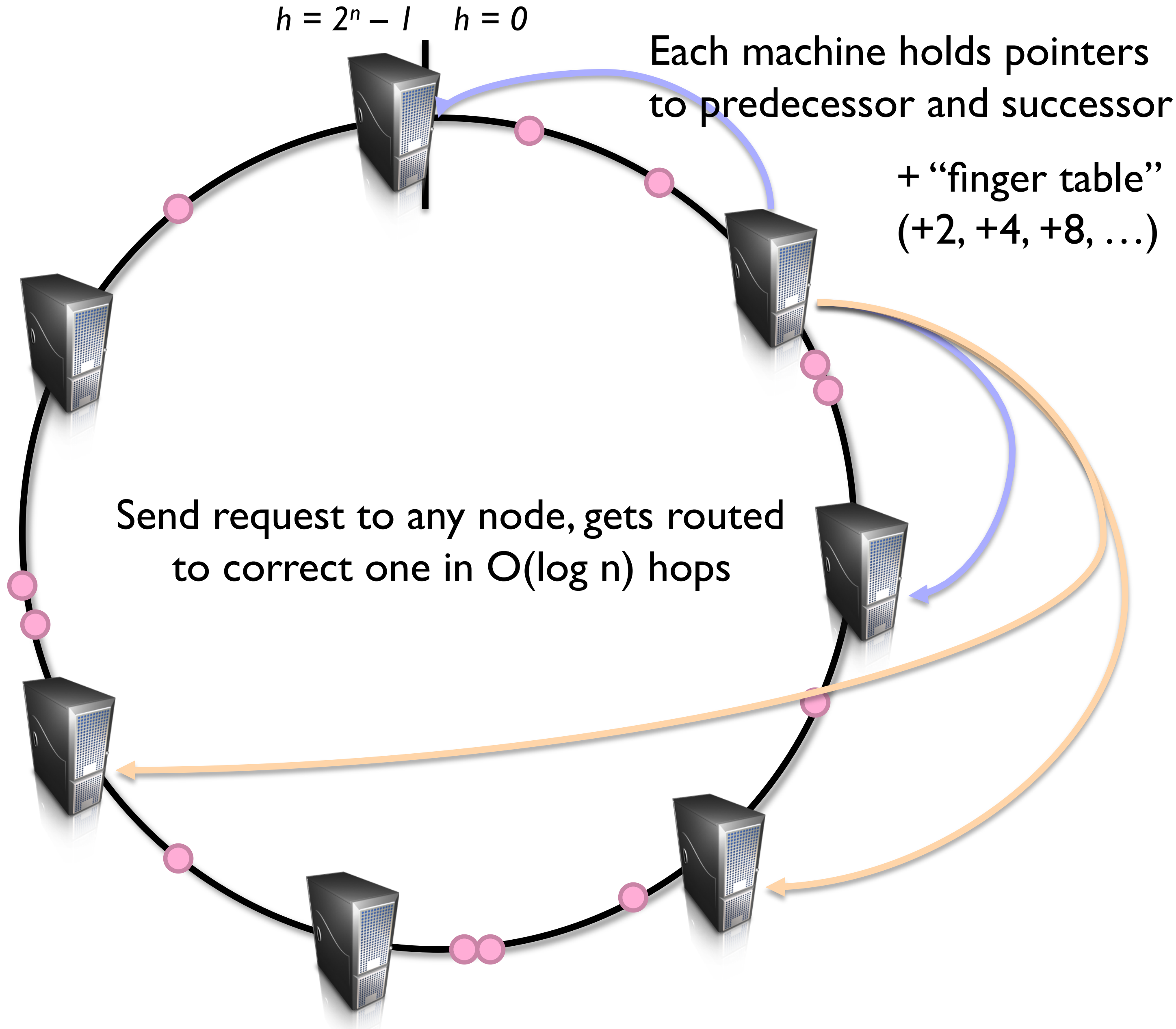
(following combines ideas from several sources…)



$h = 2^n - 1$
$h = 0$

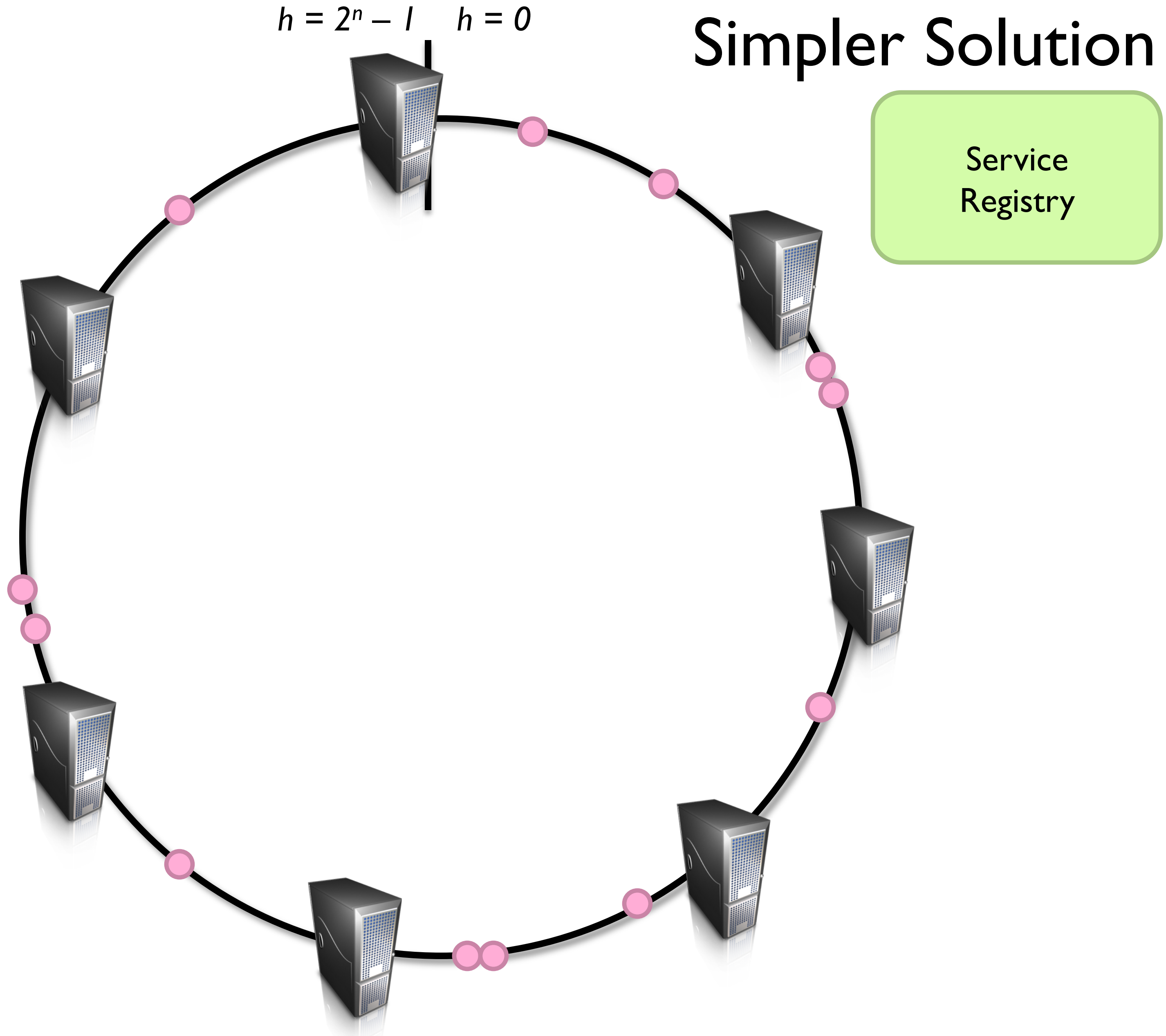


Routing: Which machine holds the key?



Routing: Which machine holds the key?

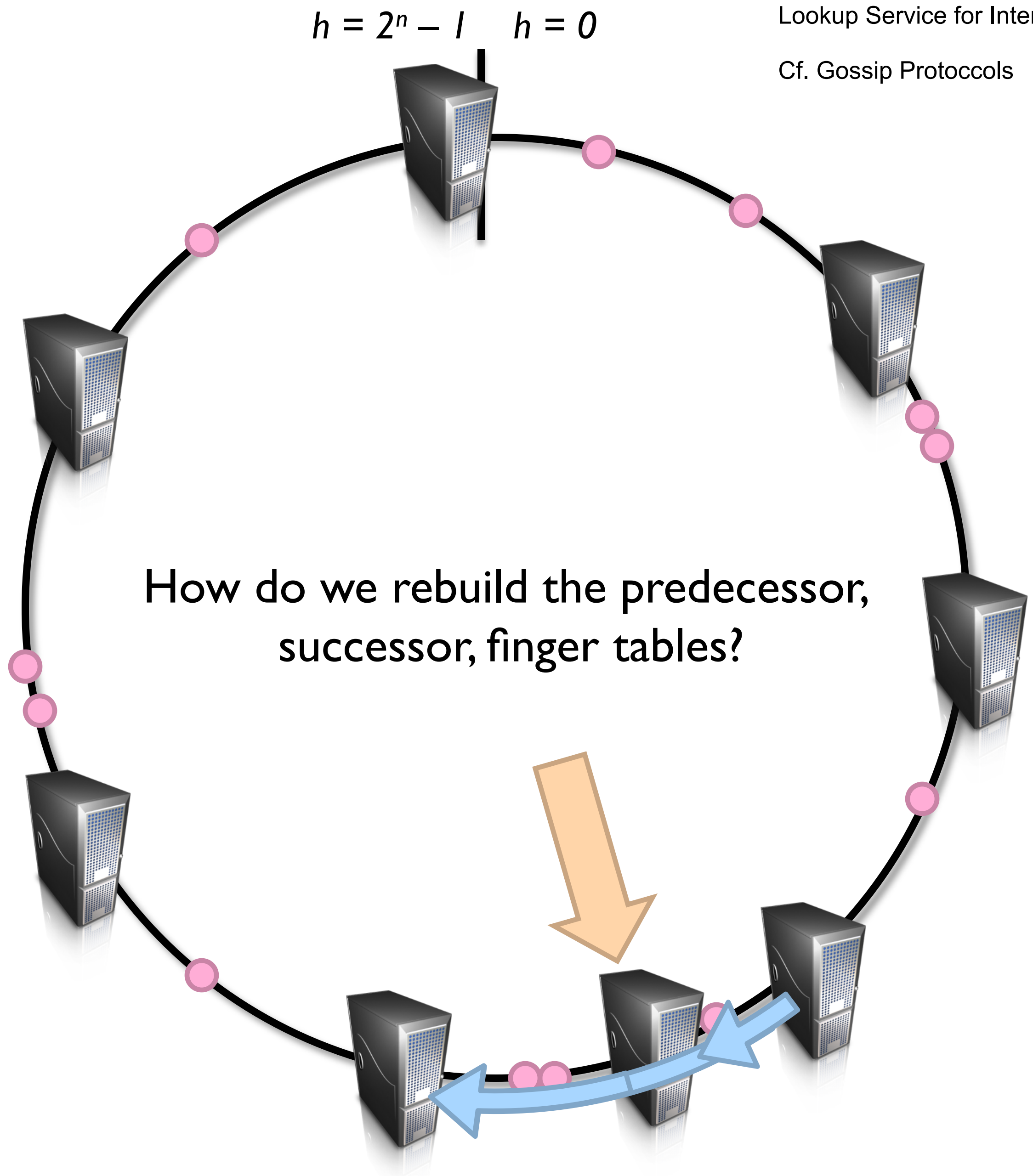
Simpler Solution



Routing: Which machine holds the key?

Stoica et al. (2001). Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications. *SIGCOMM.*

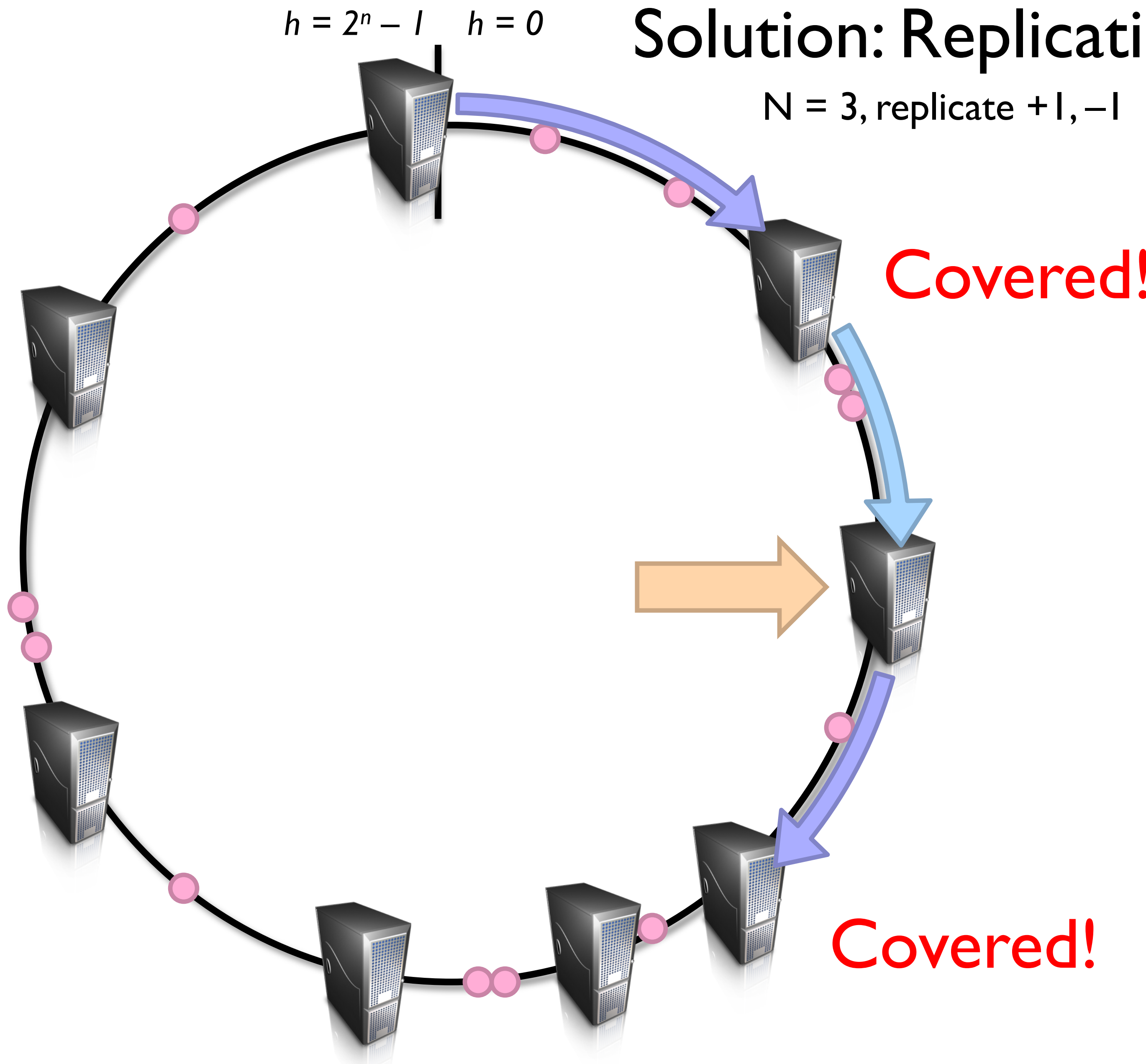
Cf. Gossip Protoccols



New machine joins: What happens?

Solution: Replication

N = 3, replicate +1, –1



Machine fails: What happens?

How to actually replicate? Later…

Another Refinement: Virtual Nodes

- Don’t directly hash servers
- Create a large number of virtual nodes, map to physical servers
  - Better load redistribution in event of machine failure
  - When new server joins, evenly shed load from other servers

Bigtable

Source: Wikipedia (Table)

Data Model

- A table in Bigtable is a sparse, distributed, persistent multidimensional sorted map
- Map indexed by a row key, column key, and a timestamp
  - (row:string, column:string, time:int64) → uninterpreted byte array
- Supports lookups, inserts, deletes
  - Single row transactions only

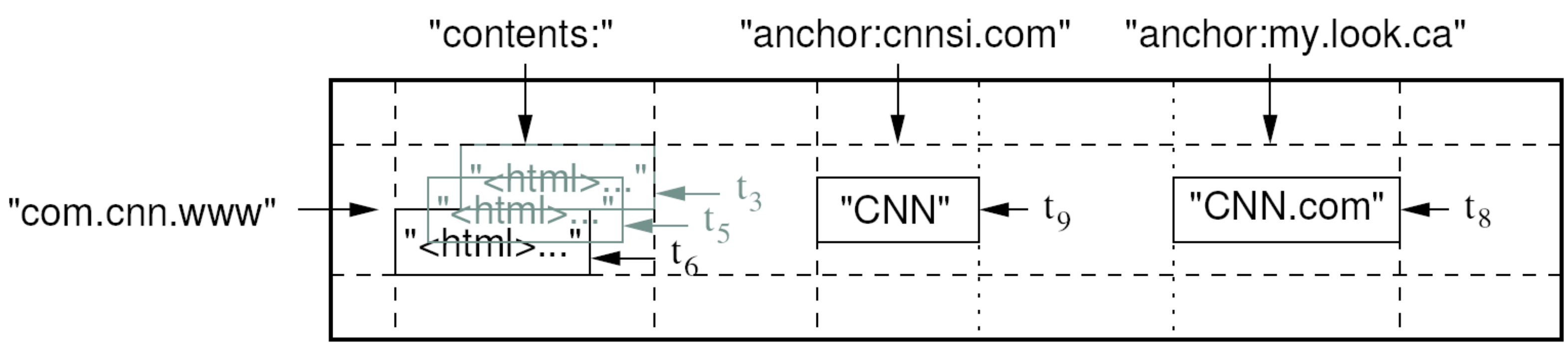


Image Source: Chang et al., OSDI 2006

Rows and Columns

- Rows maintained in sorted lexicographic order
  - Applications can exploit this property for efficient row scans
  - Row ranges dynamically partitioned into tablets
- Columns grouped into column families
  - Column key = *family:qualifier*
  - Column families provide locality hints
  - Unbounded number of columns

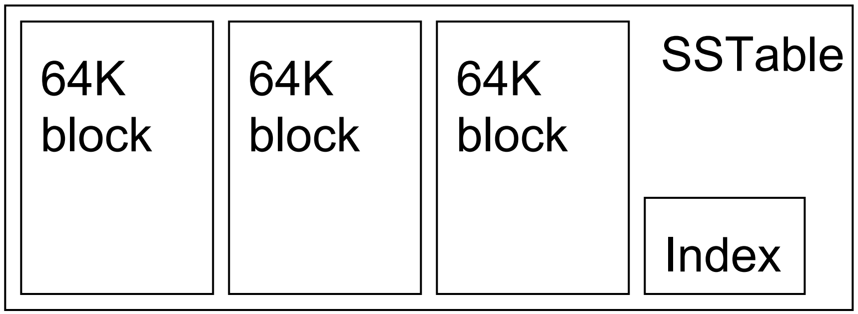
At the end of the day, it's all key-value pairs!

Bigtable Building Blocks

- GFS
- Chubby
- SSTable

SSTable

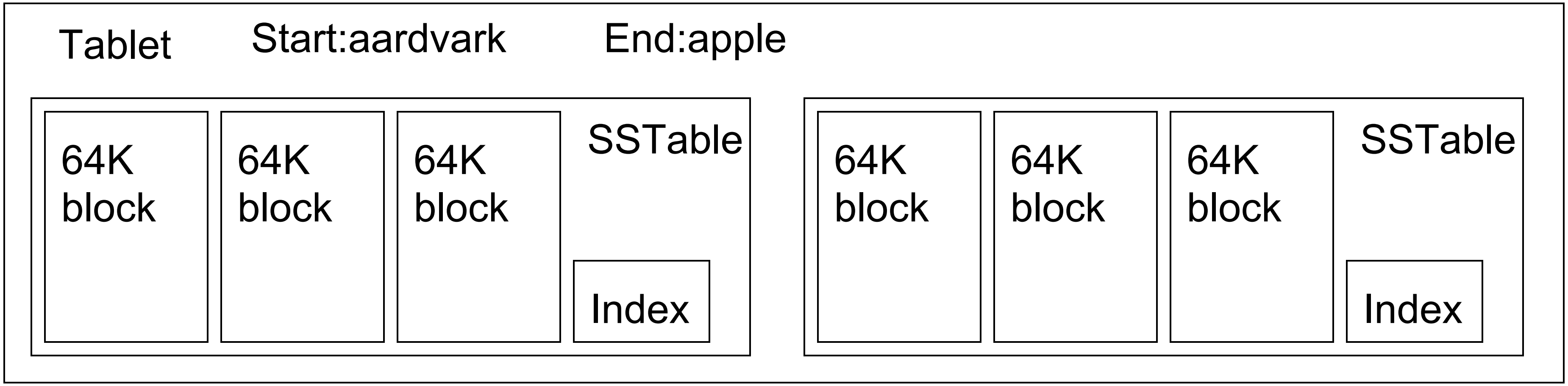
- Basic building block of Bigtable
- Persistent, ordered immutable map from keys to values
  - Stored in GFS
- Sequence of blocks on disk plus an index for block lookup
  - Can be completely mapped into memory
- Supported operations:
  - Look up value associated with key
  - Iterate key/value pairs within a key range



Source: Graphic from slides by Erik Paulson

Tablet

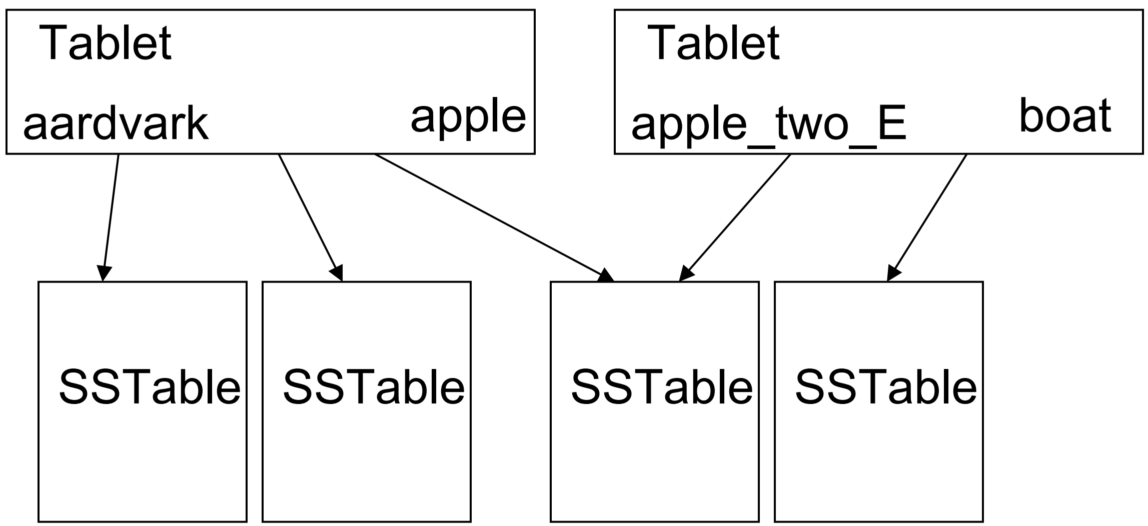
- Dynamically partitioned range of rows
- Built from multiple SSTables



Source: Graphic from slides by Erik Paulson

Table

- Multiple tablets make up the table
- SSTables can be shared



Source: Graphic from slides by Erik Paulson

Architecture

- Client library
- Single master server
- Tablet servers

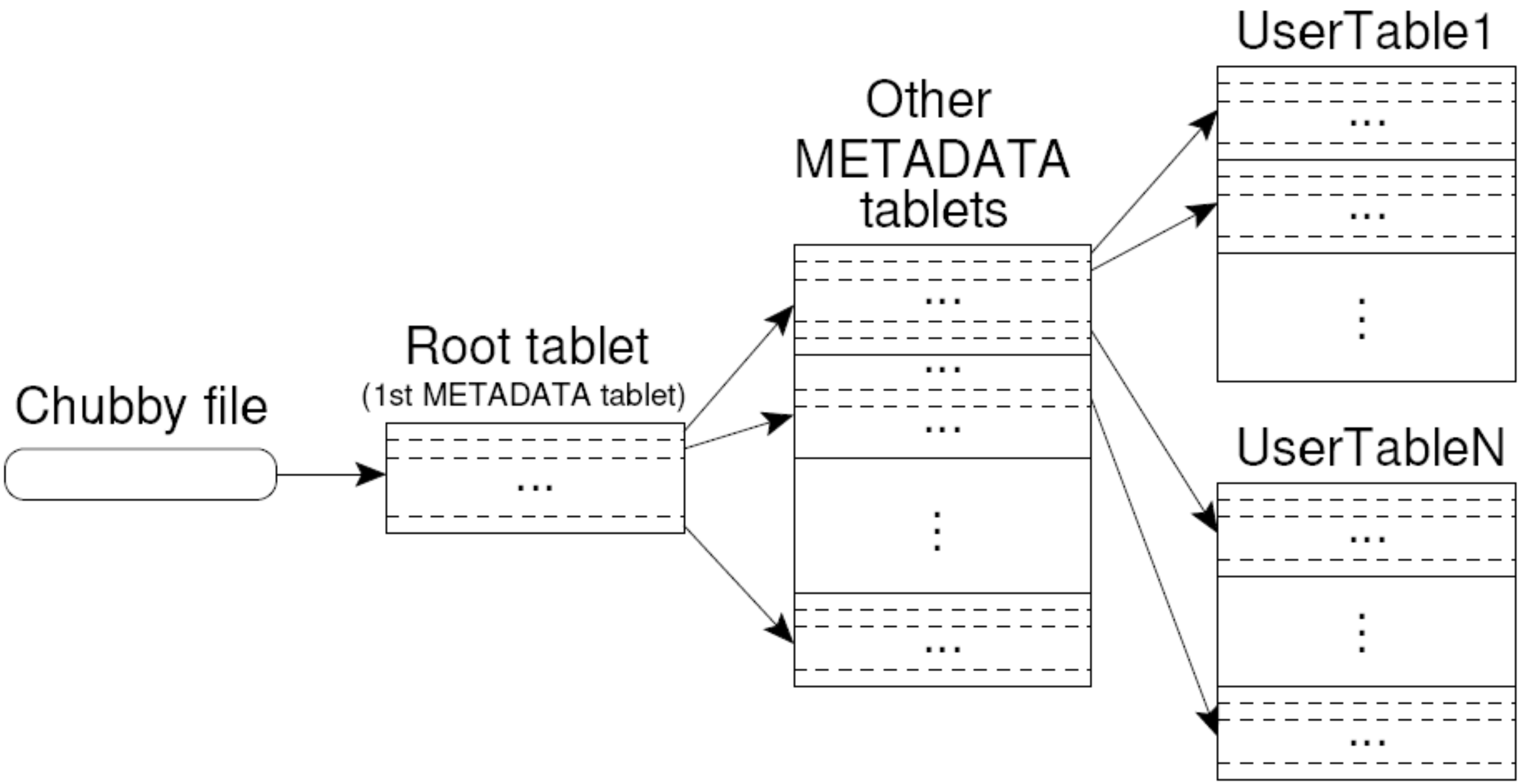
Bigtable Master

- Assigns tablets to tablet servers
- Detects addition and expiration of tablet servers
- Balances tablet server load
- Handles garbage collection
- Handles schema changes

Bigtable Tablet Servers

- Each tablet server manages a set of tablets
  - Typically between ten to a thousand tablets
  - Each 100-200 MB by default
- Handles read and write requests to the tablets
- Splits tablets that have grown too large

Tablet Location



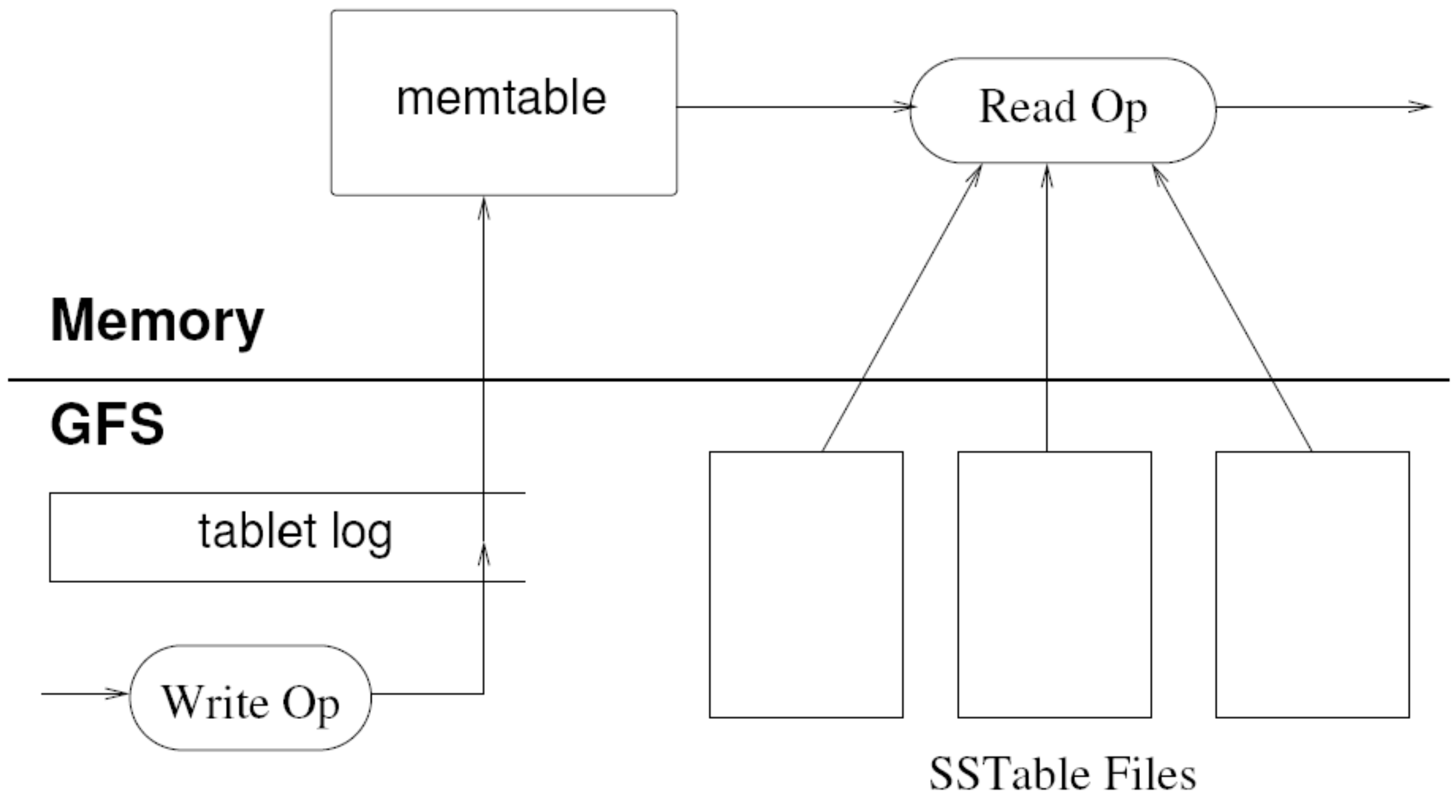
Upon discovery, clients cache tablet locations

Image Source: Chang et al., OSDI 2006

Tablet Assignment

- Master keeps track of:
  - Set of live tablet servers
  - Assignment of tablets to tablet servers
  - Unassigned tablets
- Each tablet is assigned to one tablet server at a time
  - Tablet server maintains an exclusive lock on a file in Chubby
  - Master monitors tablet servers and handles assignment
- Changes to tablet structure
  - Table creation/deletion (master initiated)
  - Tablet merging (master initiated)
  - Tablet splitting (tablet server initiated)

Tablet Serving



"Log Structured Merge Trees"

Image Source: Chang et al., OSDI 2006

Compactions

- Minor compaction
  - Converts the memtable into an SSTable
  - Reduces memory usage and log traffic on restart
- Merging compaction
  - Reads the contents of a few SSTables and the memtable, and writes out a new SSTable
  - Reduces number of SSTables
- Major compaction
  - Merging compaction that results in only one SSTable
  - No deletion records, only live data

Bigtable Applications

- Data source and data sink for MapReduce
- Google's web crawl
- Google Earth
- Google Analytics

HBase

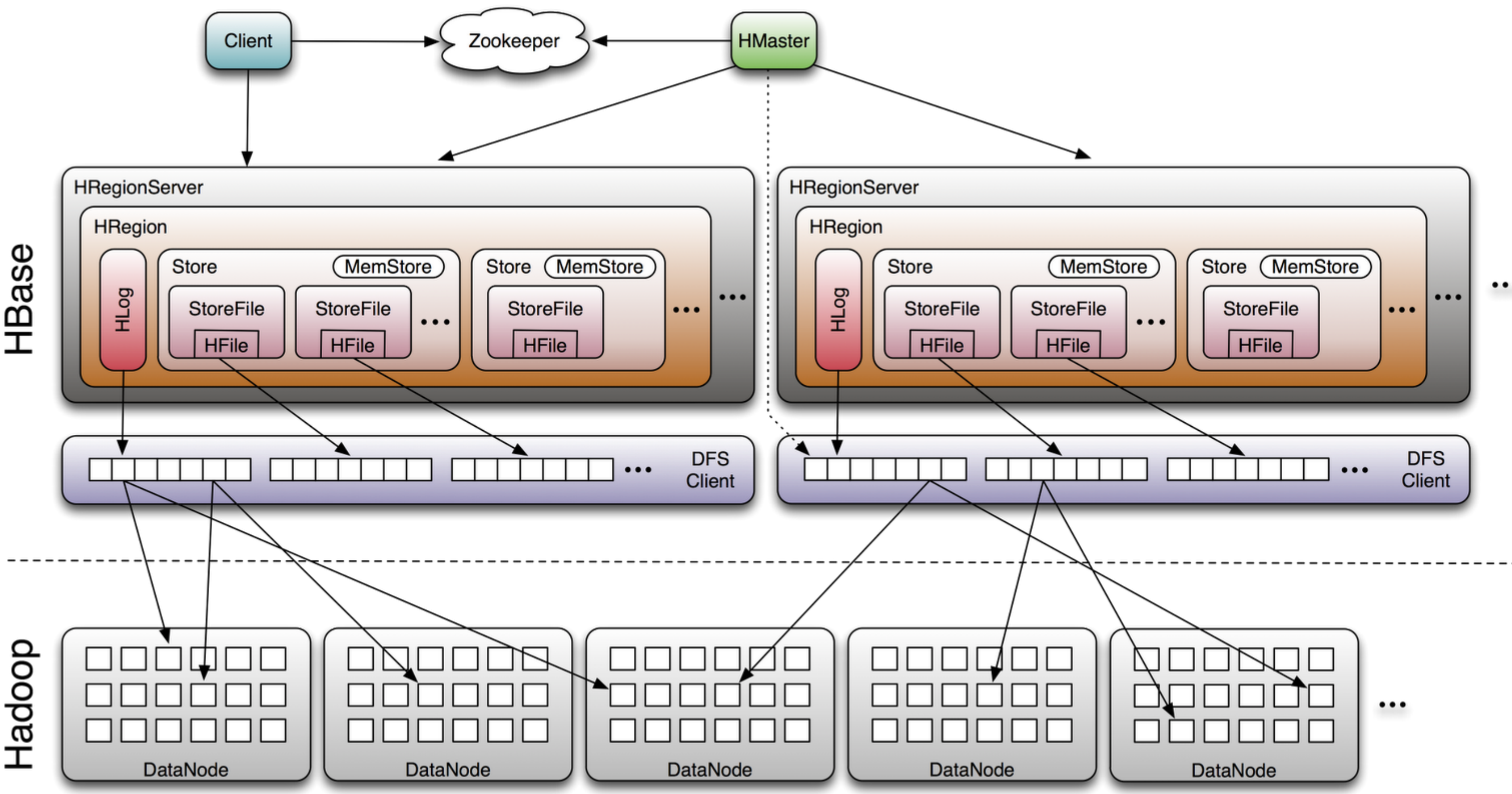


Image Source: http://www.larsgeorge.com/2009/10/hbase-architecture-101-storage.html

Challenges

- Keep it simple!
- Keep it flexible!
- Push functionality onto application

… and it's still hard to get right!
Example: splitting and compaction storms

Back to consistency...

Consistency 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
- Replication
  - For robustness (availability)
  - For throughput

Let's just focus on this

- Caching
  - For latency

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

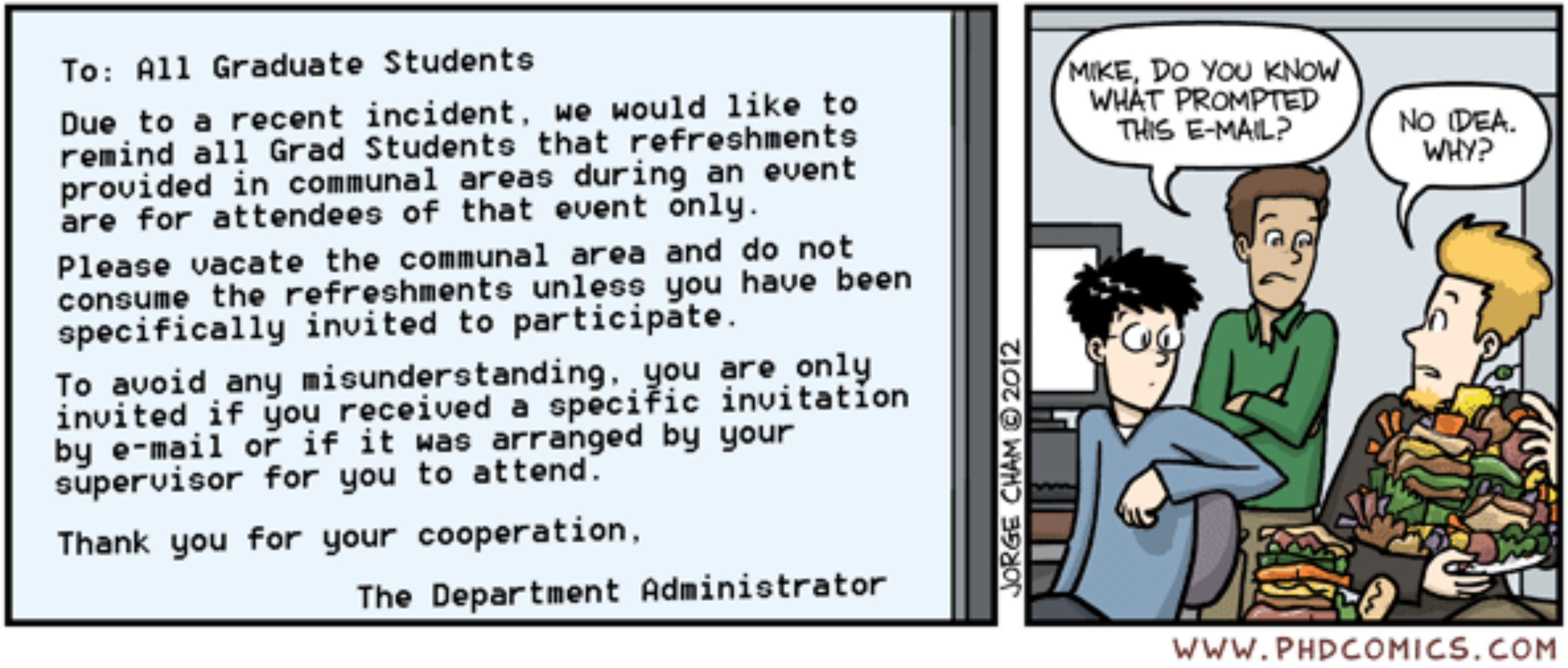
Replication possibilities

- 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

Three Core Ideas

- Partitioning (sharding)
  - For scalability
  - For latency

Quick look at this

- Replication
  - For robustness (availability)
  - For throughput
- Caching
  - For latency

“Unit of Consistency”

- Single record:
  - Relatively straightforward
  - Complex application logic to handle multi-record transactions
- Arbitrary transactions:
  - Requires 2PC/Paxos
- 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.

Three Core Ideas

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

This is really hard!

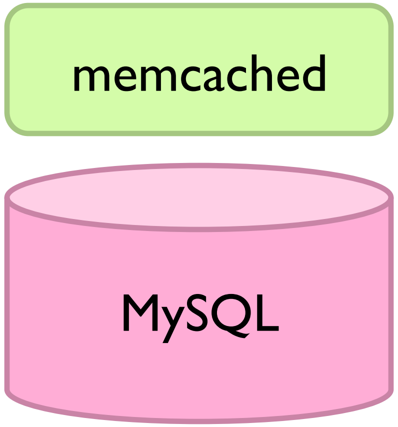

Now imagine multiple datacenters…
What's different?
Source: Google

Three Core Ideas

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

Quick look at this

Facebook Architecture



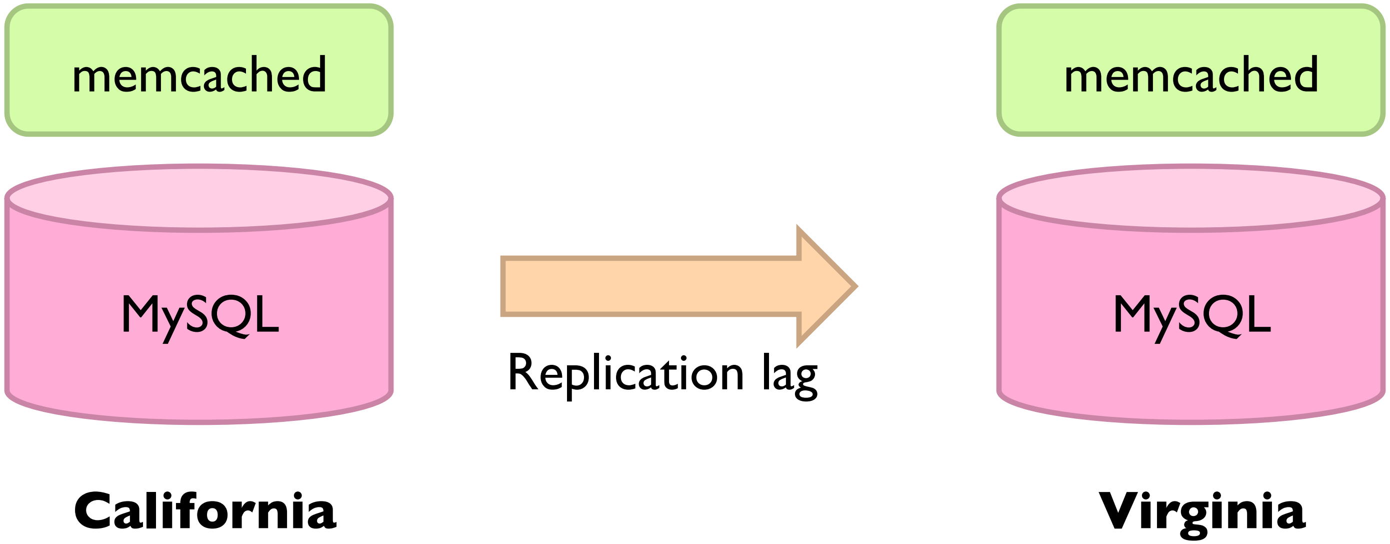
Read path:
Look in memcached
Look in MySQL
Populate in memcached

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

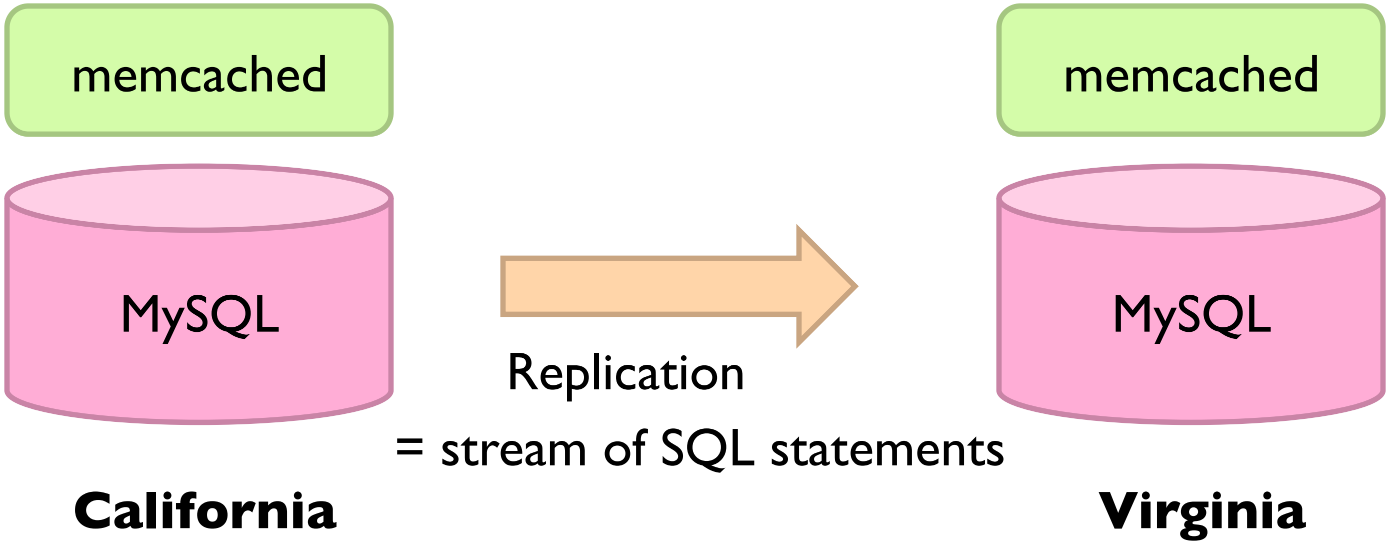
Facebook Architecture: Multi-DC



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
- Replication
  - For robustness (availability)
  - For throughput

Get's go back to this again

- Caching
  - For latency

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

Have our cake and eat it too?

Google's Spanner

- Features:
  - Full ACID translations across multiple datacenters, across continents!
  - External consistency: wrt globally-consistent timestamps!
- How?
  - TrueTime: globally synchronized API using GPSes and atomic clocks
  - Use 2PC but use Paxos to replicate state
- Tradeoffs?

Questions?

Source: Wikipedia (Japanese rock garden)