THE UNIVERSITY *of* EDINBURGH
informatics

Extreme Computing

NoSQL

PREVIOUSLY: BATCH

Query most/all data
Results Eventually

NOW: ON DEMAND

Single Data Points
Latency Matters

One problem, three ideas

- 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!

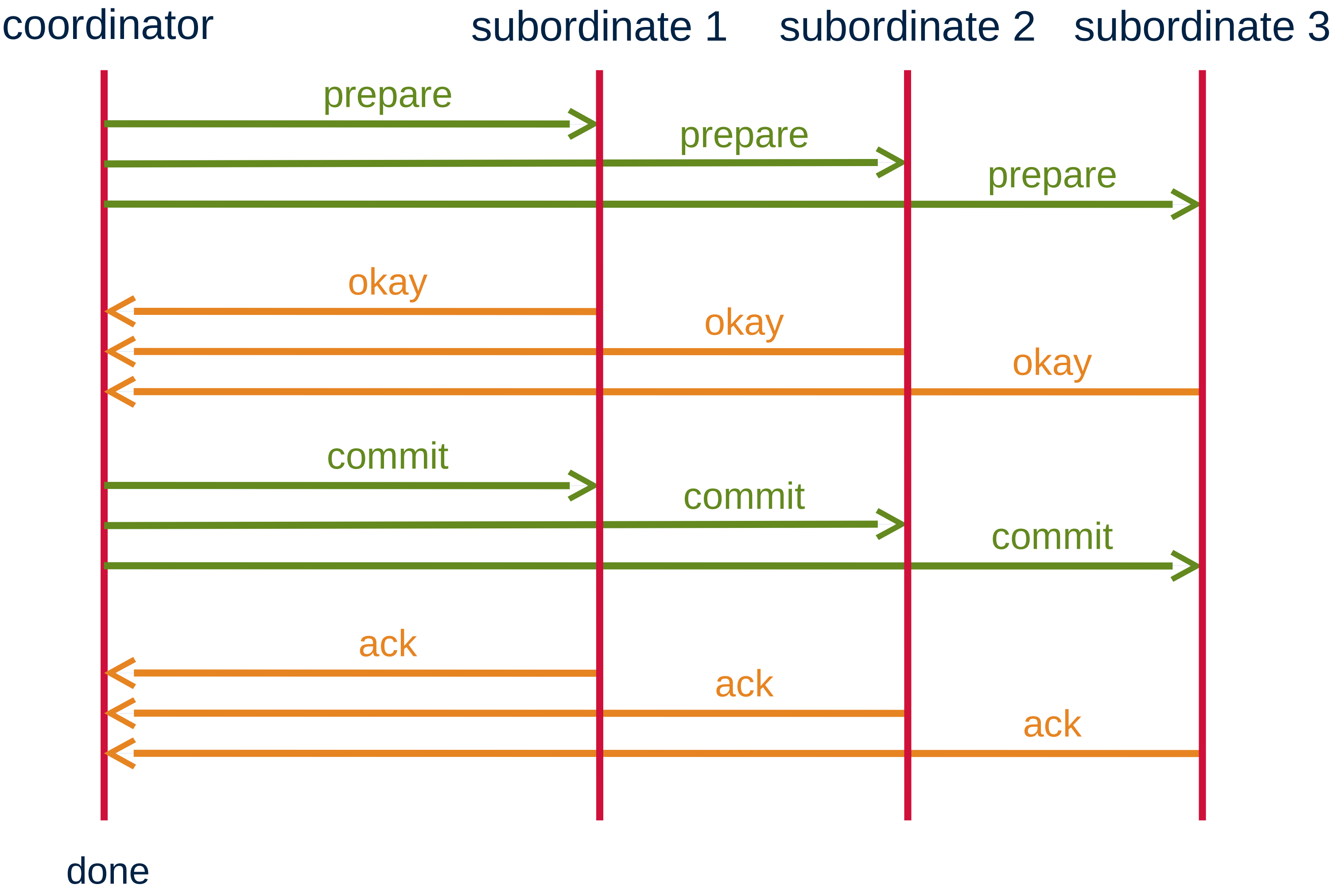
- Three core ideas
  - Partitioning (sharding)
    - For scalability
    - For latency
  - Replication
    - For robustness (availability)
    - For throughput
  - Caching
    - For latency

- Three more problems
  - How do we synchronise partitions?
  - How do we synchronise replicas?
  - What happens to the cache when the underlying data changes?

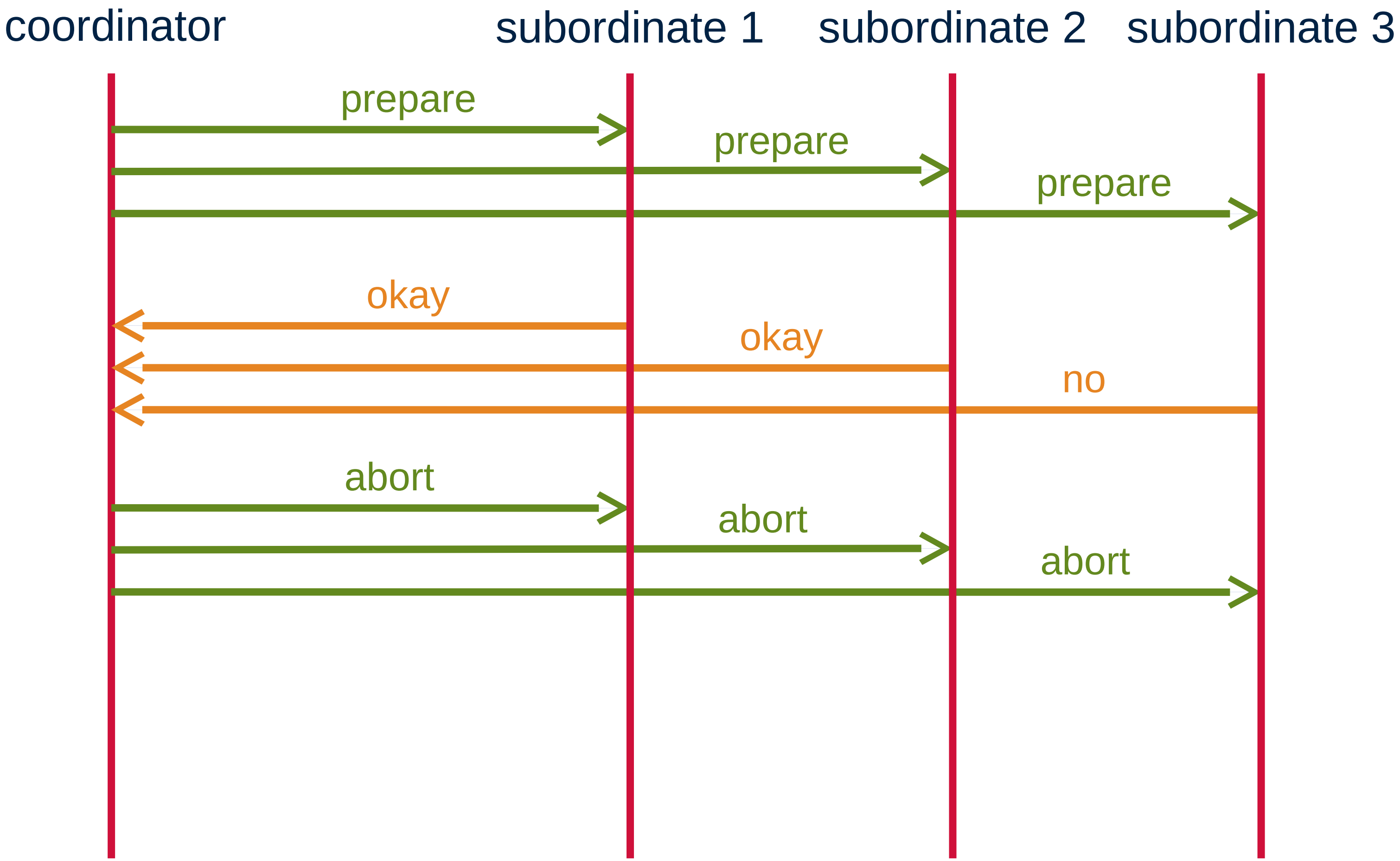
Relational databases to the rescue

- RDBMSs provide
  - Relational model with schemas
  - Powerful, flexible query language
  - Transactional semantics
  - Rich ecosystem, lots of tool support
- Great, I'm sold! How do they do this?
  - 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
- Need a new distributed protocol (but remember two generals)
  - Two-phase commit (2PC)

2PC commit


coordinator
subordinate 1
subordinate 2
subordinate 3
prepare
prepare
prepare
okay
okay
okay
commit
commit
commit
ack
ack
ack
done

2PC abort



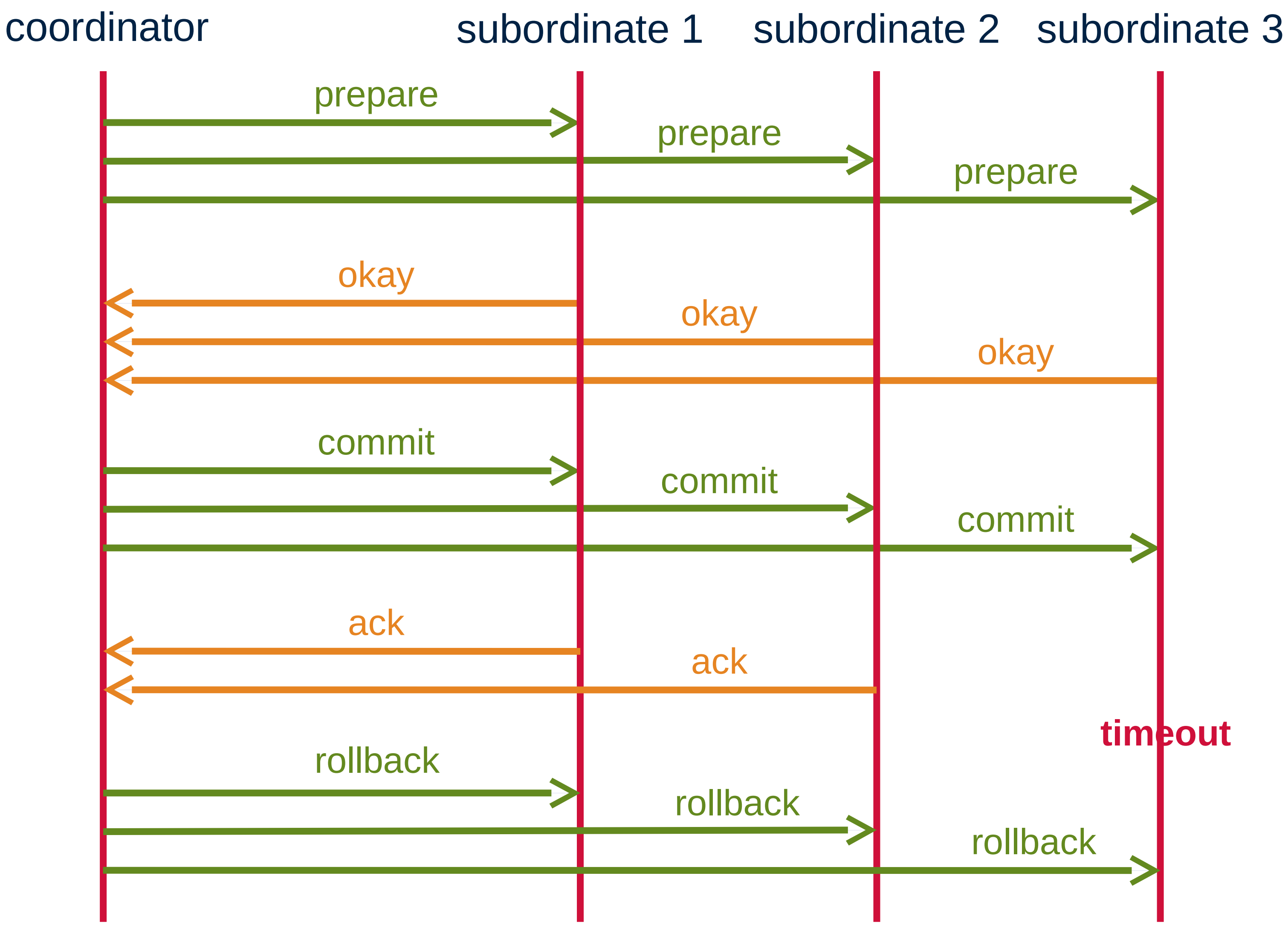
coordinator

subordinate 1

subordinate 2

subordinate 3

2PC rollback



2PC: assumptions and limitations

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

Solution: Paxos!

(details beyond scope of this course)

Problems with RDBMSs

- Must design from the beginning
  - Difficult and expensive to evolve
- True transactions implies two-phase commit
  - Slow!
- Databases are expensive
  - Distributed databases are even more expensive

What do RDBMSs provide?

- Relational model with schemas
- Powerful, flexible query language
- Transactional semantics: ACID
- Rich ecosystem, lots of tool support
- Do we need all these?
  - What if we selectively drop some of these assumptions?
  - 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?

Solution: NoSQL

NoSQL

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

- The “No” in NoSQL used to mean No
- Supposedly now it means “Not only”
- Four major types of NoSQL databases
  - Key-value stores
  - Column-oriented databases
  - Document stores
  - Graph databases

KEY-VALUE STORES

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
- But what if data does not fit on a single machine?

Dealing with scale

- 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?
- We need something better
  - Hash the keys
  - Hash the machines
  - Distributed hash tables

BIGTABLE

BigTable: 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

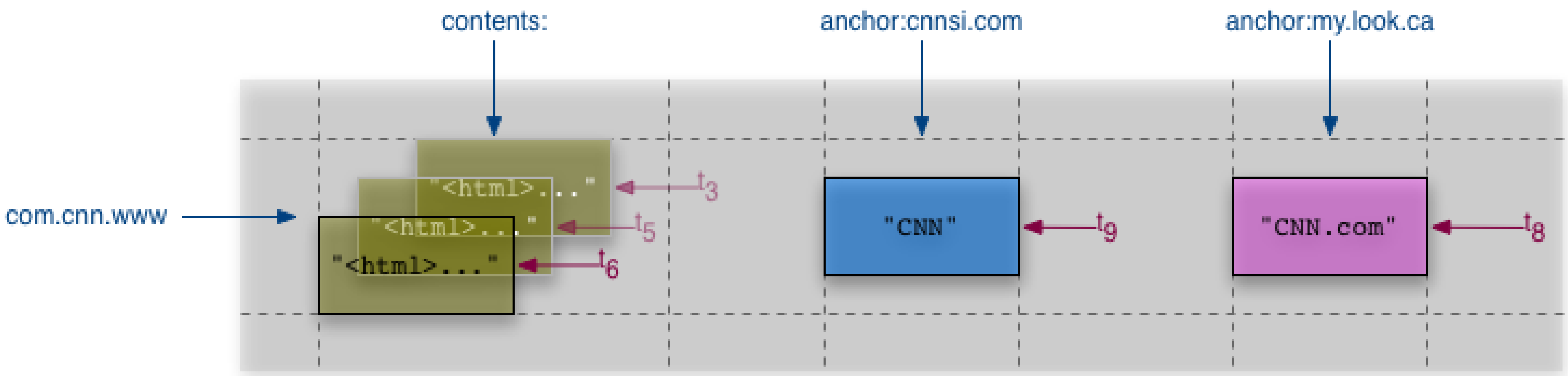


Image Source: Chang et al., OSDI 2006

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

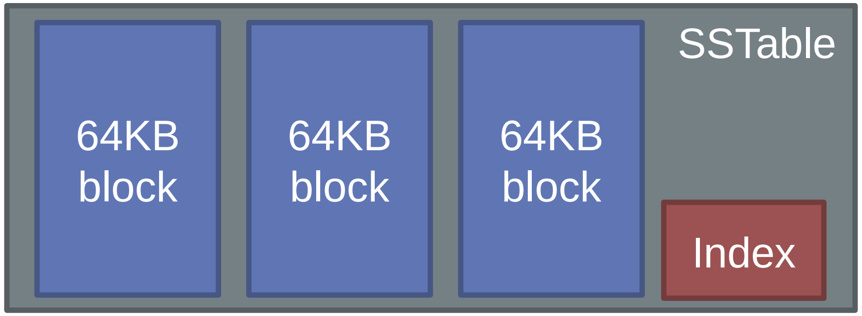
At the end of the day, it’s all key-value pairs!

BigTable building blocks

- GFS
- Chubby
- SSTable

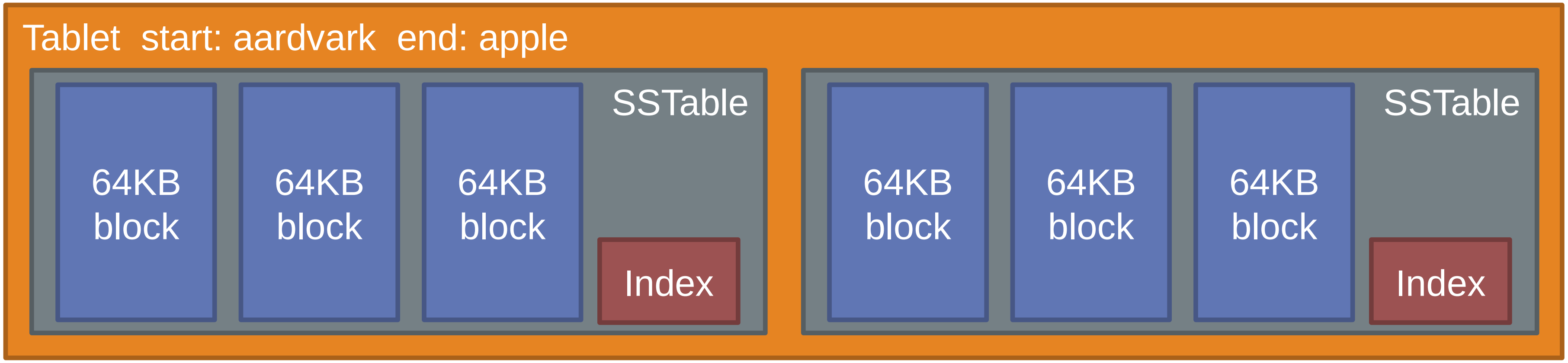
SSTable

- 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



Tablets and tables

- Dynamically partitioned range of rows
- Built from multiple SSTables

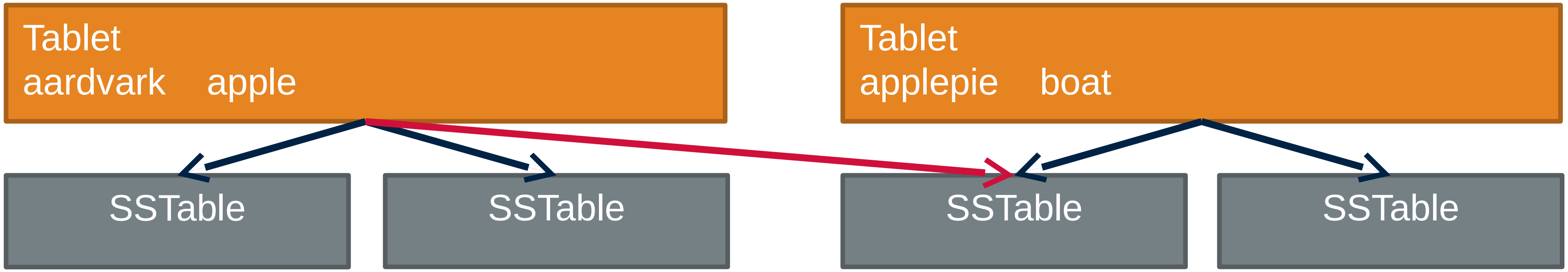


Tablet start: aardvark end: apple

SSTable: 64KB block, 64KB block, 64KB block, Index

SSTable: 64KB block, 64KB block, 64KB block, Index

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



Tablet
aardvark apple

Tablet
applepie boat

SSTable

SSTable

SSTable

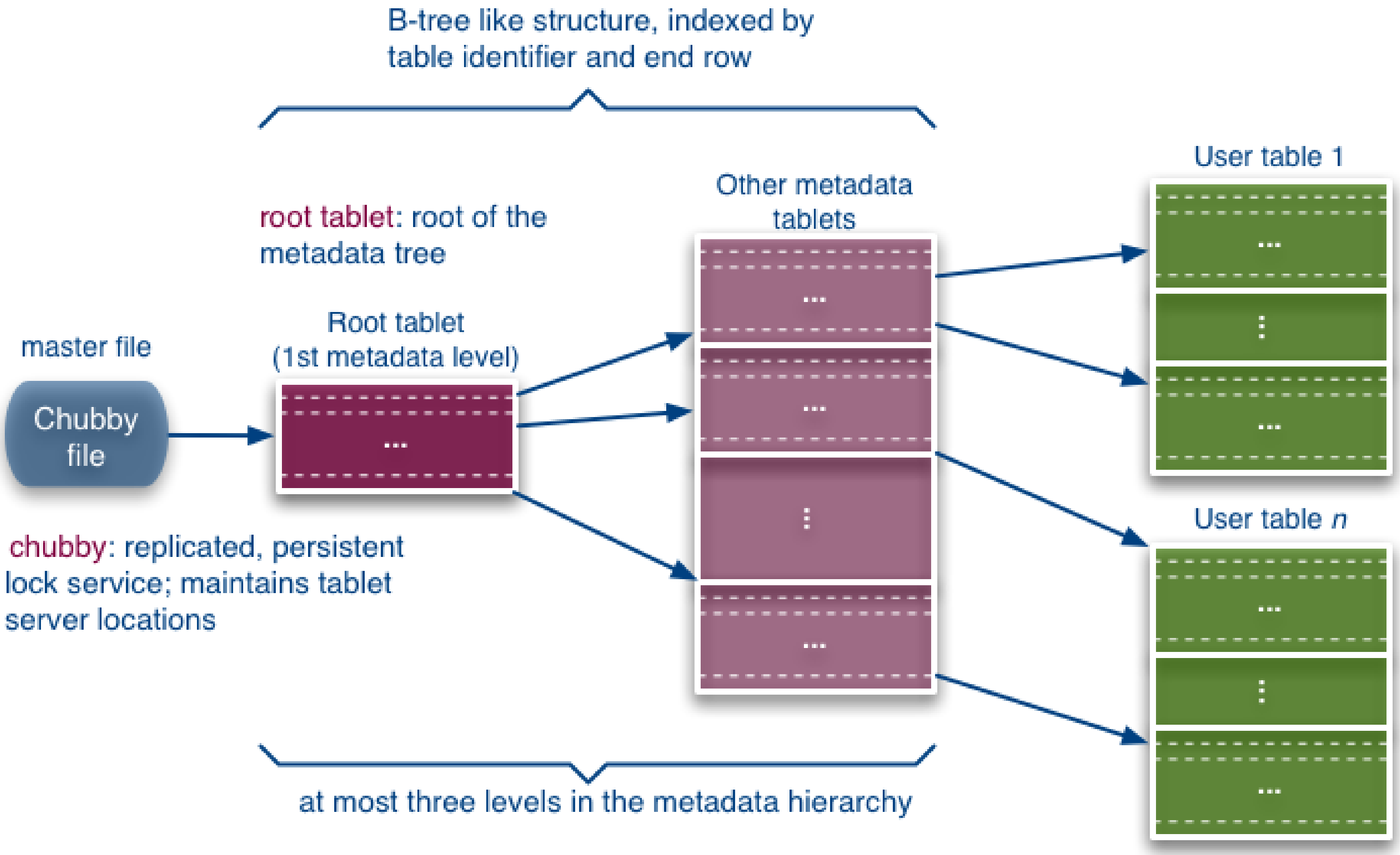
SSTable

Source: Graphic from slides by Erik Paulson

Notes on the architecture

- Similar to GFS
    - Single master server, multiple 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 evolution
- Bigtable tablet servers
    - Each tablet server manages a set of tablets
        - Typically between ten to a thousand tablets
        - Each 100-200MB by default
- Handles read and write requests to the tablets
    - Splits tablets when they grow too large

Location dereferencing



B-tree like structure, indexed by table identifier and end row

root tablet: root of the metadata tree

master file

Root tablet (1st metadata level)

Other metadata tablets

User table 1

User table *n*

chubby: replicated, persistent lock service; maintains tablet server locations

at most three levels in the metadata hierarchy

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 and I/O flow

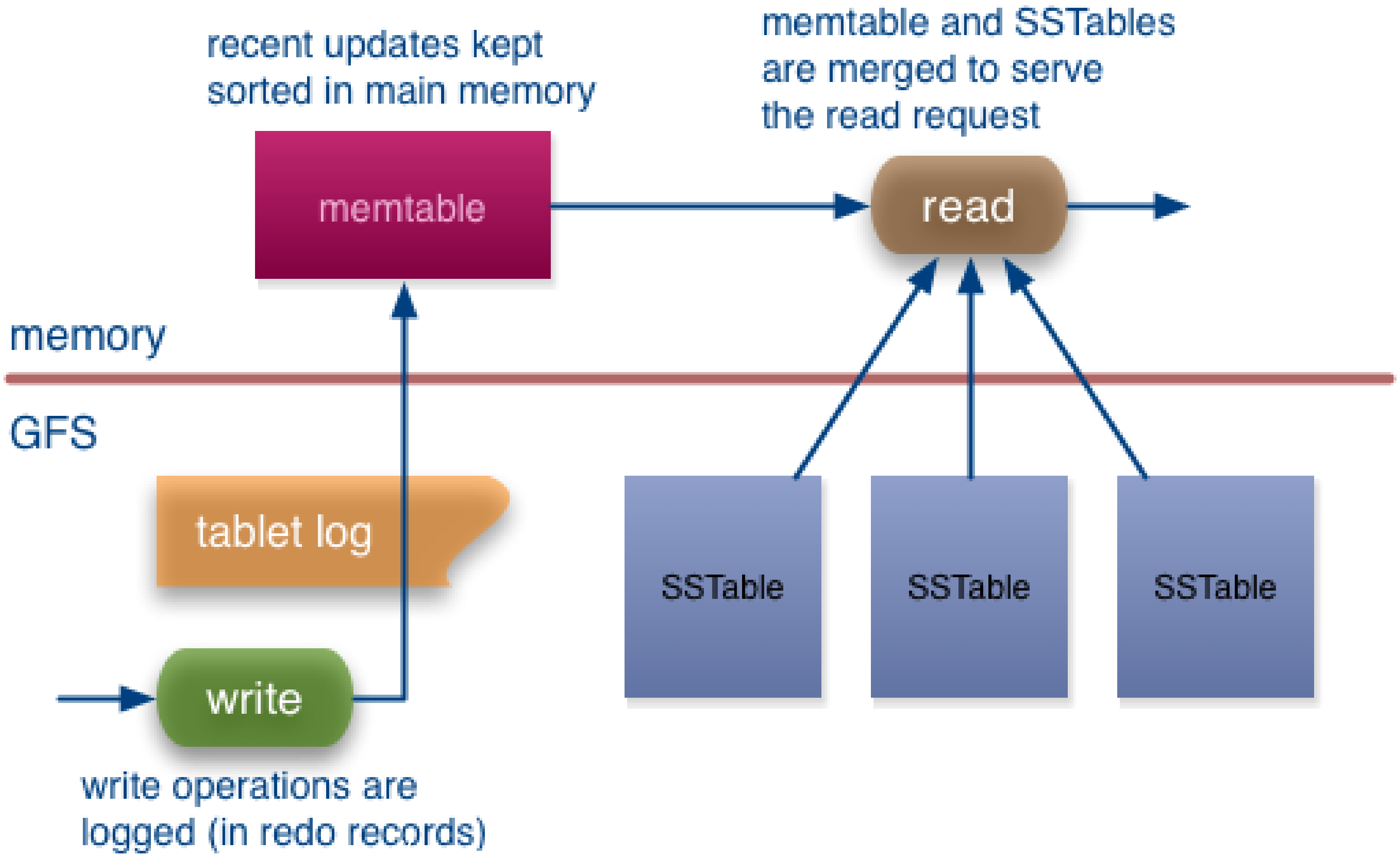


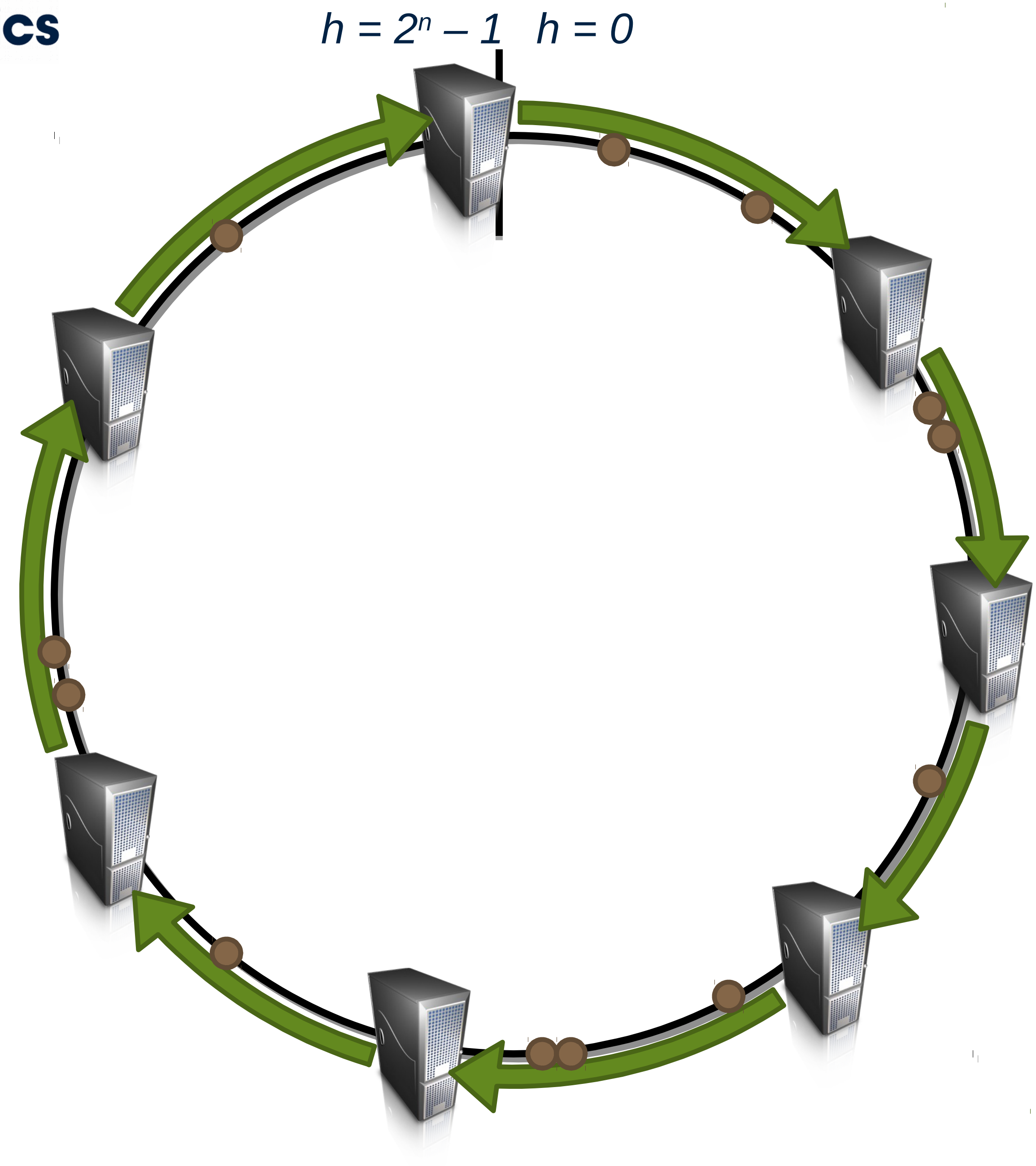
Image Source: Chang et al., OSDI 2006

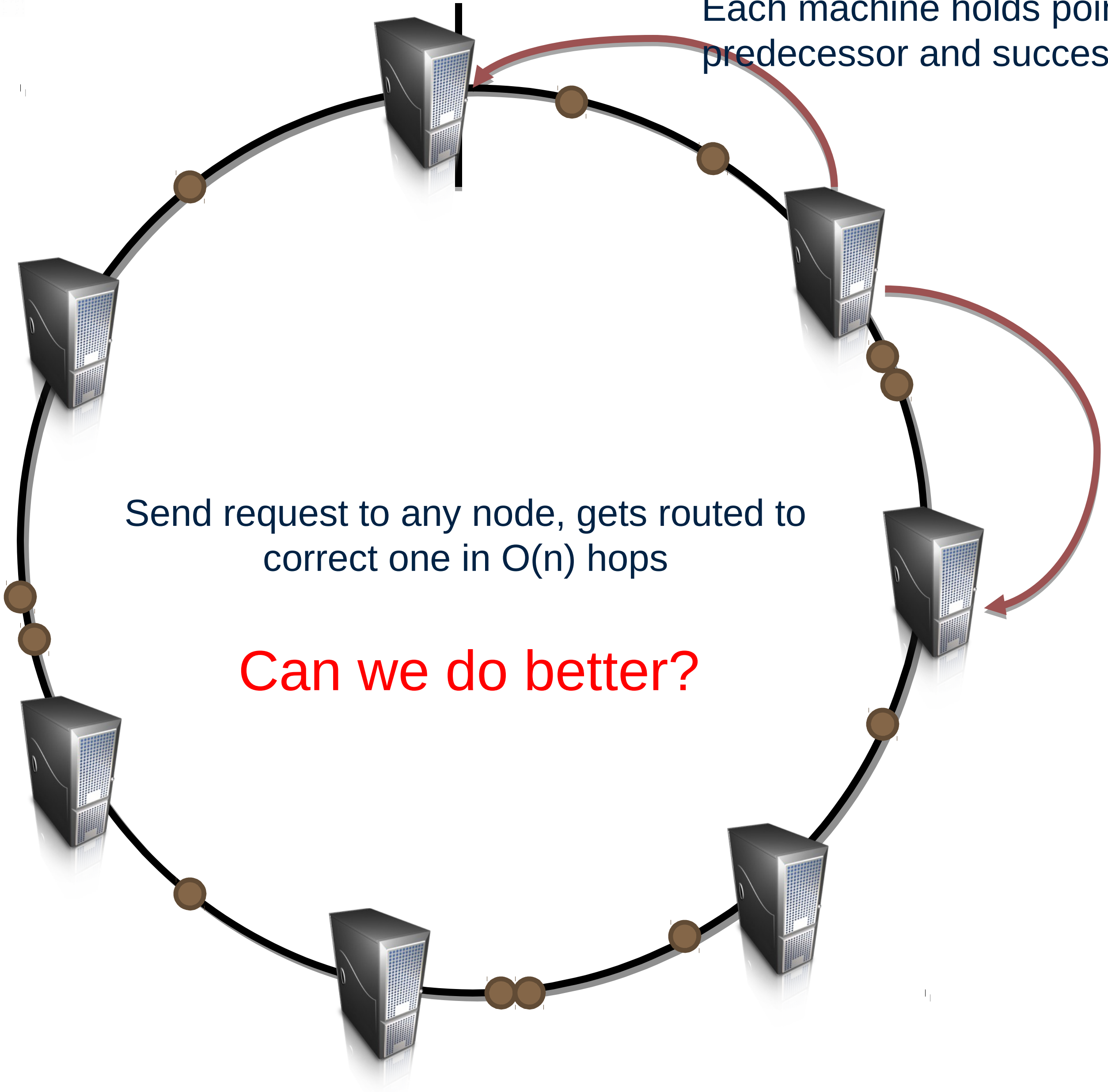
Tablet management

- 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

DISTRIBUTED HASH TABLES: CHORD

THE UNIVERSITY *of* EDINBURGH
informatics



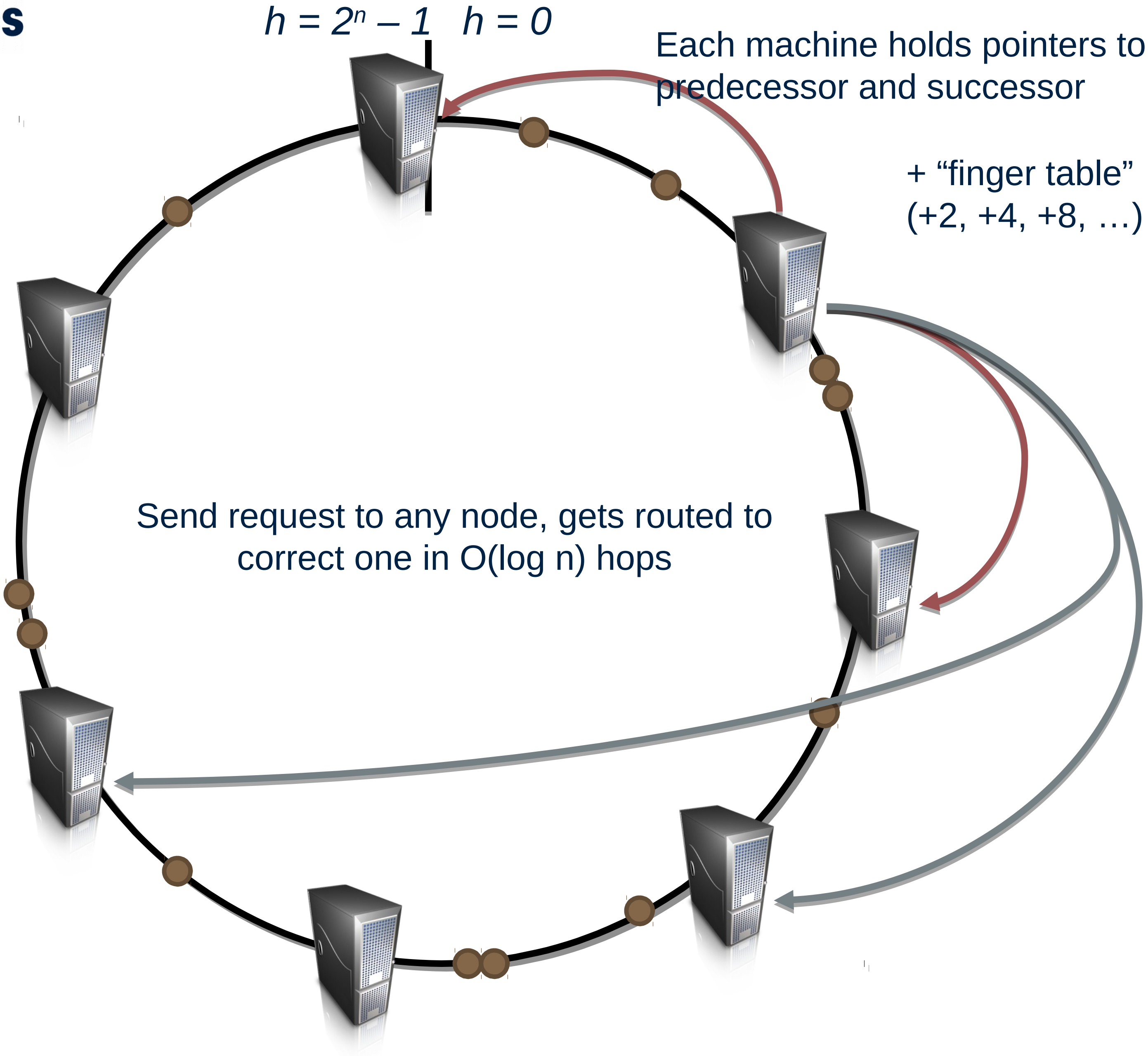


Each machine holds pointers to predecessor and successor

Send request to any node, gets routed to correct one in O(n) hops

Can we do better?

Routing: which machine holds the key?



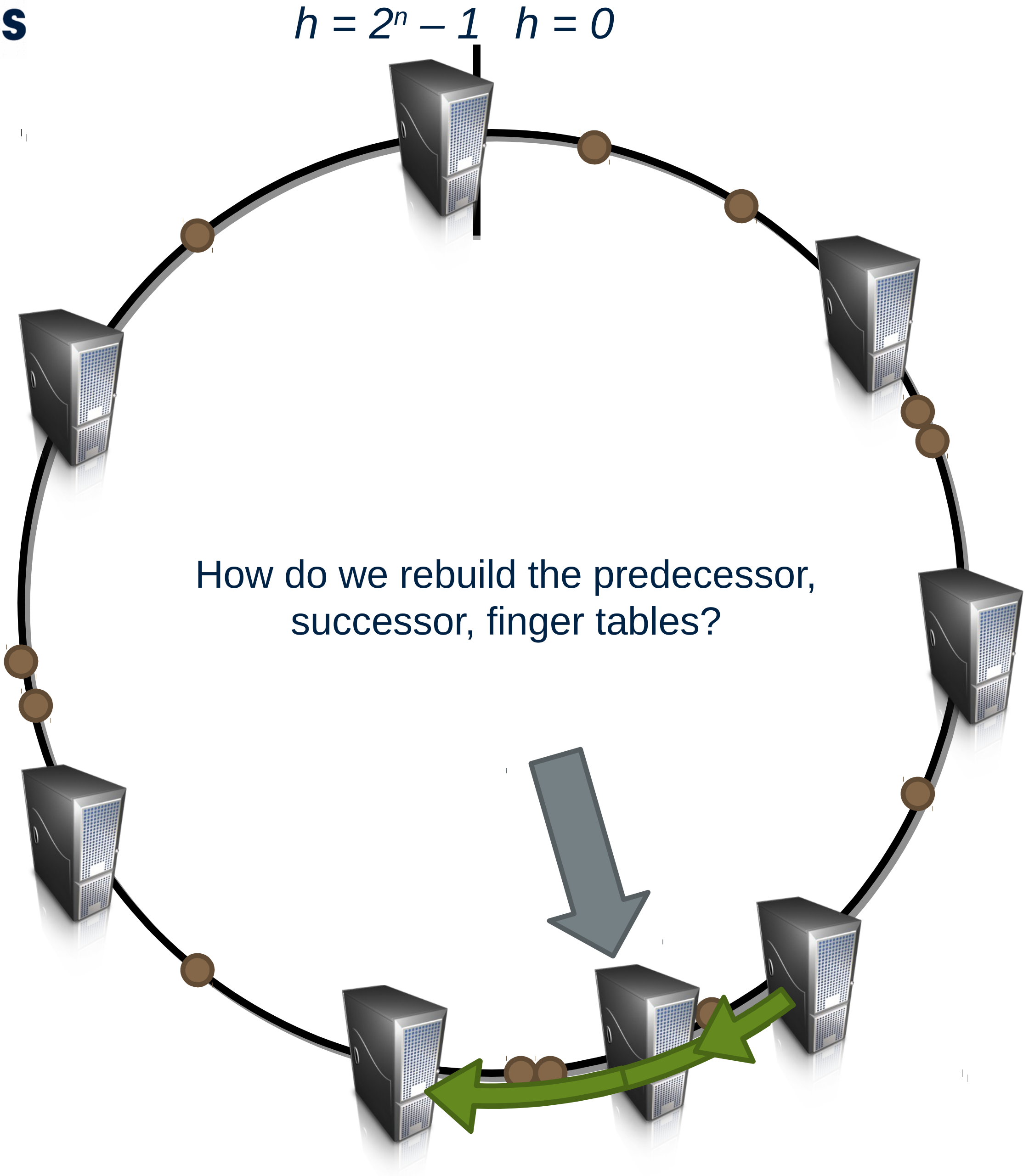
$h = 2^n - 1$ *$h = 0$*

Each machine holds pointers to predecessor and successor

+ “finger table” (+2, +4, +8, …)

Send request to any node, gets routed to correct one in O(log n) hops

Routing: which machine holds the key?

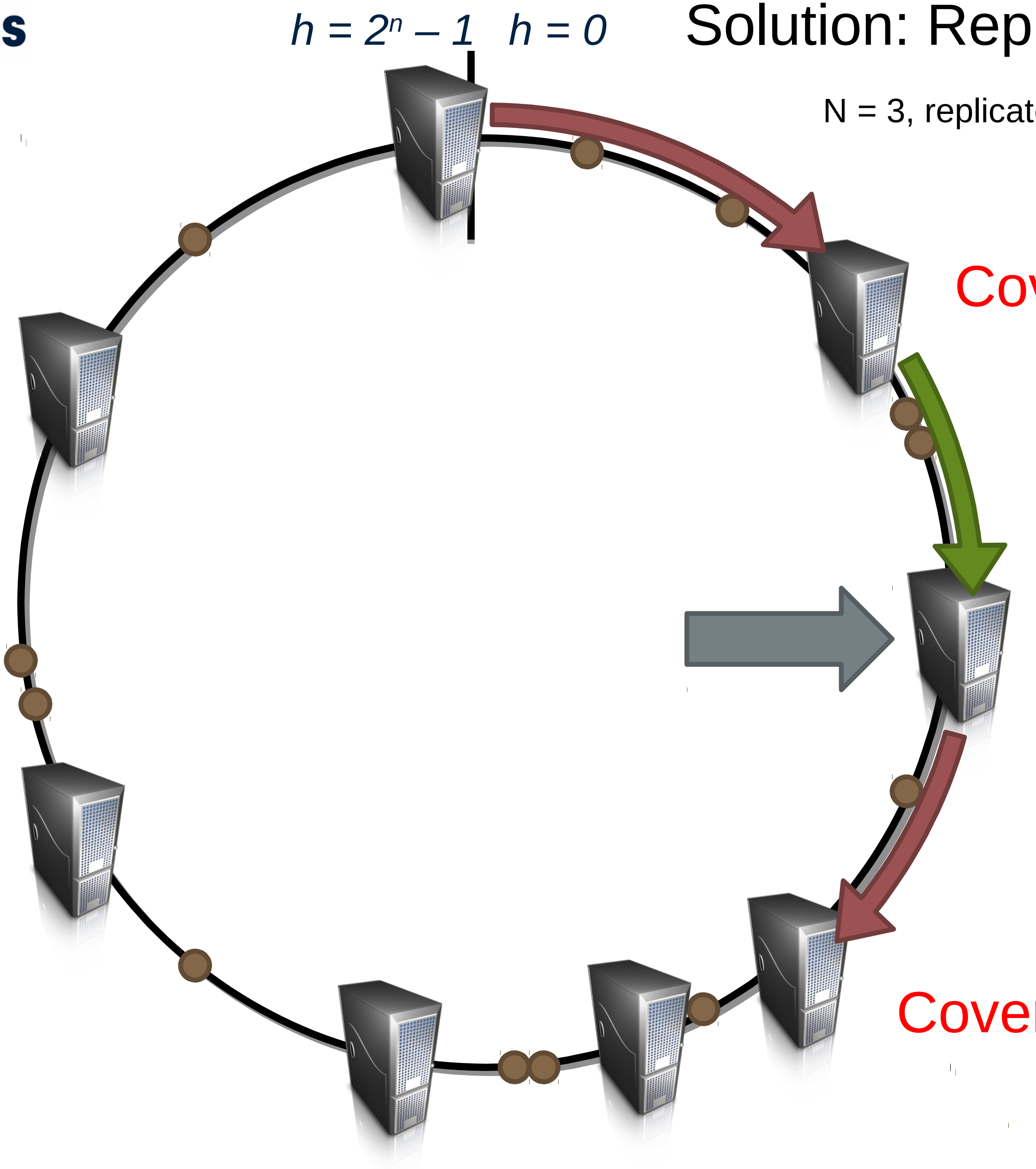


How do we rebuild the predecessor, successor, finger tables?

New machine joins: what happens?

Solution: Replication

N = 3, replicate +1, –1



Machine fails: what happens?

CONSISTENCY IN KEY-VALUE STORES

Focus on consistency

- People you do not 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

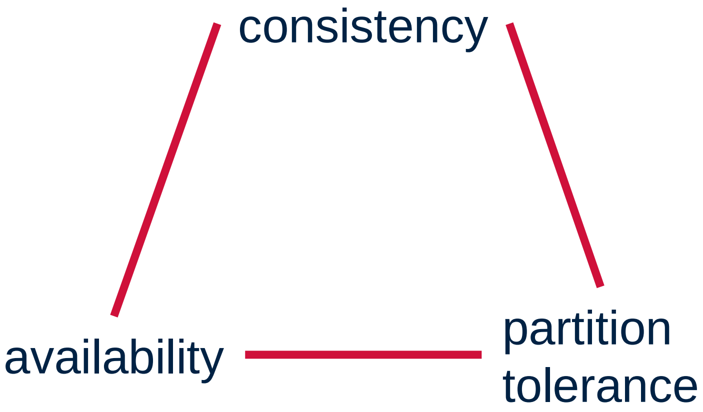
- Partitioning (sharding)
  - For scalability
  - For latency
- Replication
  - For robustness (availability)
  - For throughput
- Caching
  - For latency

We'll shift our focus here

(Re)CAP

- CAP stands for **C**onsistency, **A**vailability, **P**artition tolerance
  - Consistency: all nodes see the same data at the same time
  - Availability: node failures do not prevent system operation
  - Partition tolerance: link failures do not prevent system operation

- Largely a conjecture attributed to Eric Brewer
- *A distributed system can satisfy any two of these guarantees at the same time, but not all three*
- You can't have a triangle; pick any one side



consistency

availability

partition tolerance

CAP Tradeoffs

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

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”

Three core ideas

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

Quick look at this

Unit of consistency

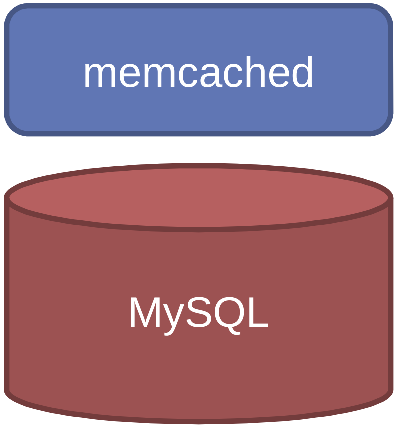
- Single record:
  - Relatively straightforward
  - Complex application logic to handle multi-record transactions
- Arbitrary transactions:
  - Requires 2PC/Paxos
- Middle ground: entity groups
  - Groups of entities that share affinity
  - Co-locate entity groups
  - Provide transaction support within 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

Quick look at this

Facebook architecture

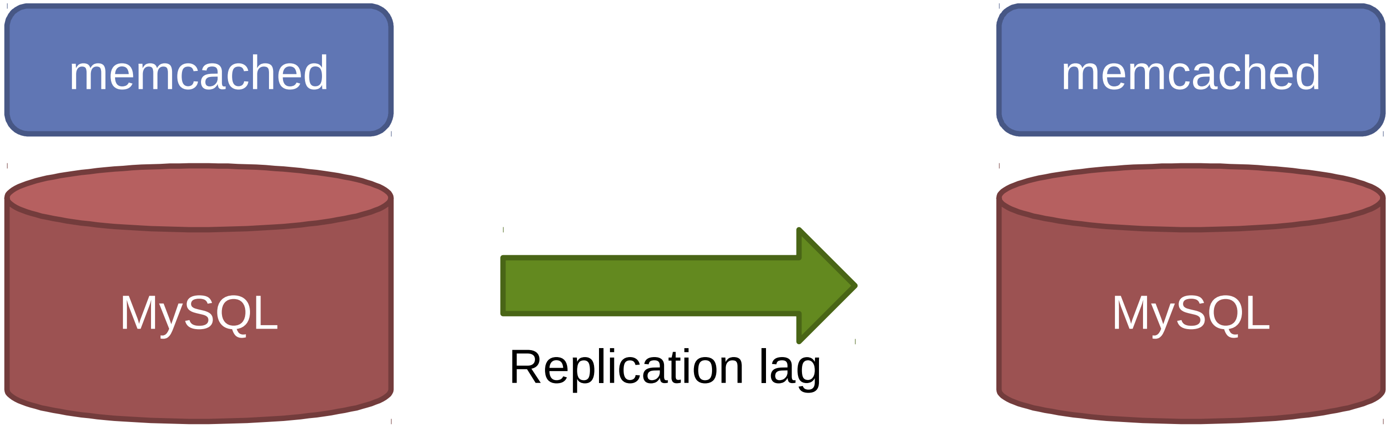


Read path:
Look in memcached
Look in MySQL
Populate in memcached

Write path:
Write in MySQL
Remove in memcached

Subsequent read:
Look in MySQL ✓
Populate in memcached

Facebook architecture: multi-DC



California

Virginia

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!

Three Core Ideas

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

Let’s go back to this again

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

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?

Summary

- Described the basics of NoSQL stores
- Discussed the benefits and detriments of RDBMSs
- Introduced various kinds of non-relational stores
    - Distributed hash tables (Chord)
    - Wide-column stores (BigTable)
- Introduced caching and replication
    - Addressed some of the associated problems
- Discussed real-world use cases