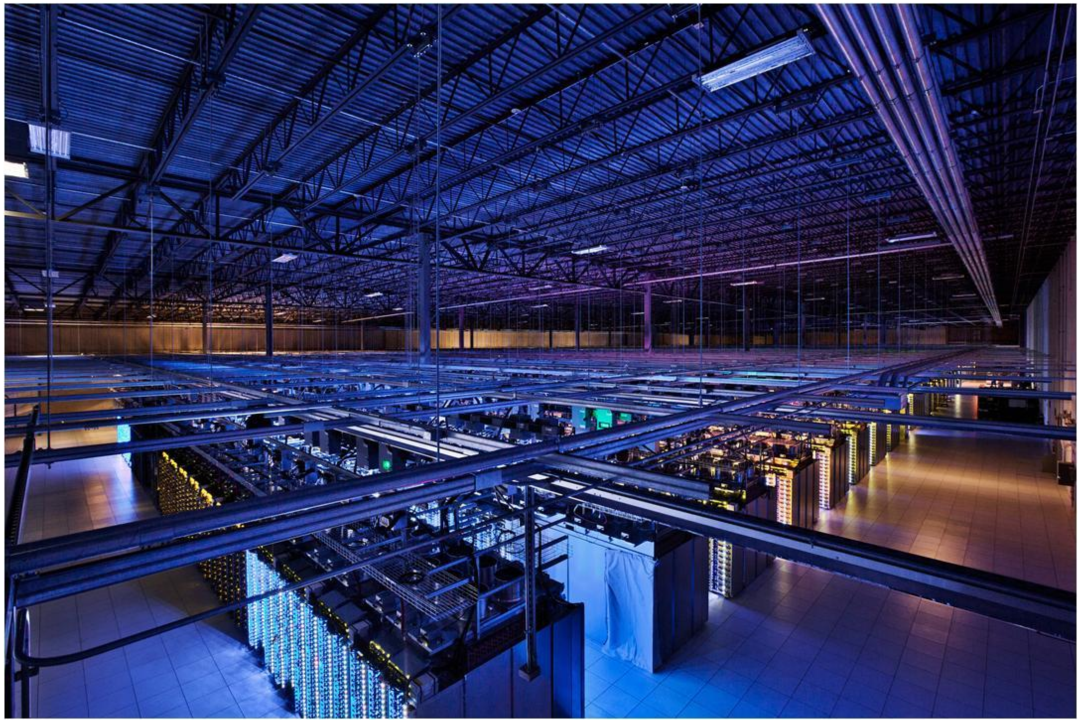
Big Data for Data Science

noSQL: BASE vs ACID



Centrum Wiskunde & Informatica

THE NEED FOR SOMETHING DIFFERENT

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!

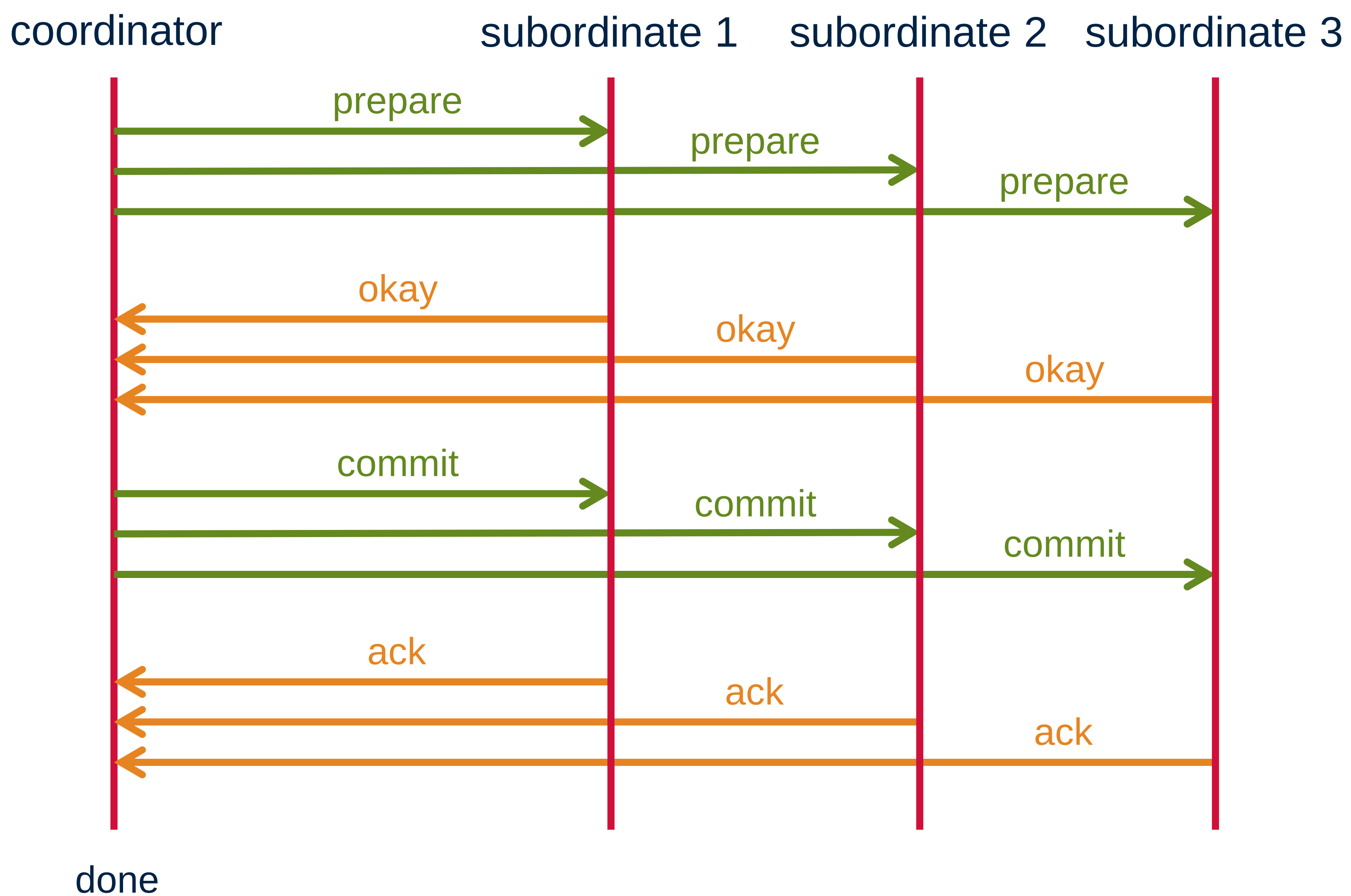
- Three core ideas
  - Partitioning (sharding)
    - For scalability
    - For latency
  - Replication
    - For robustness (availability)
    - For throughput
  - Caching
    - For latency

- 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

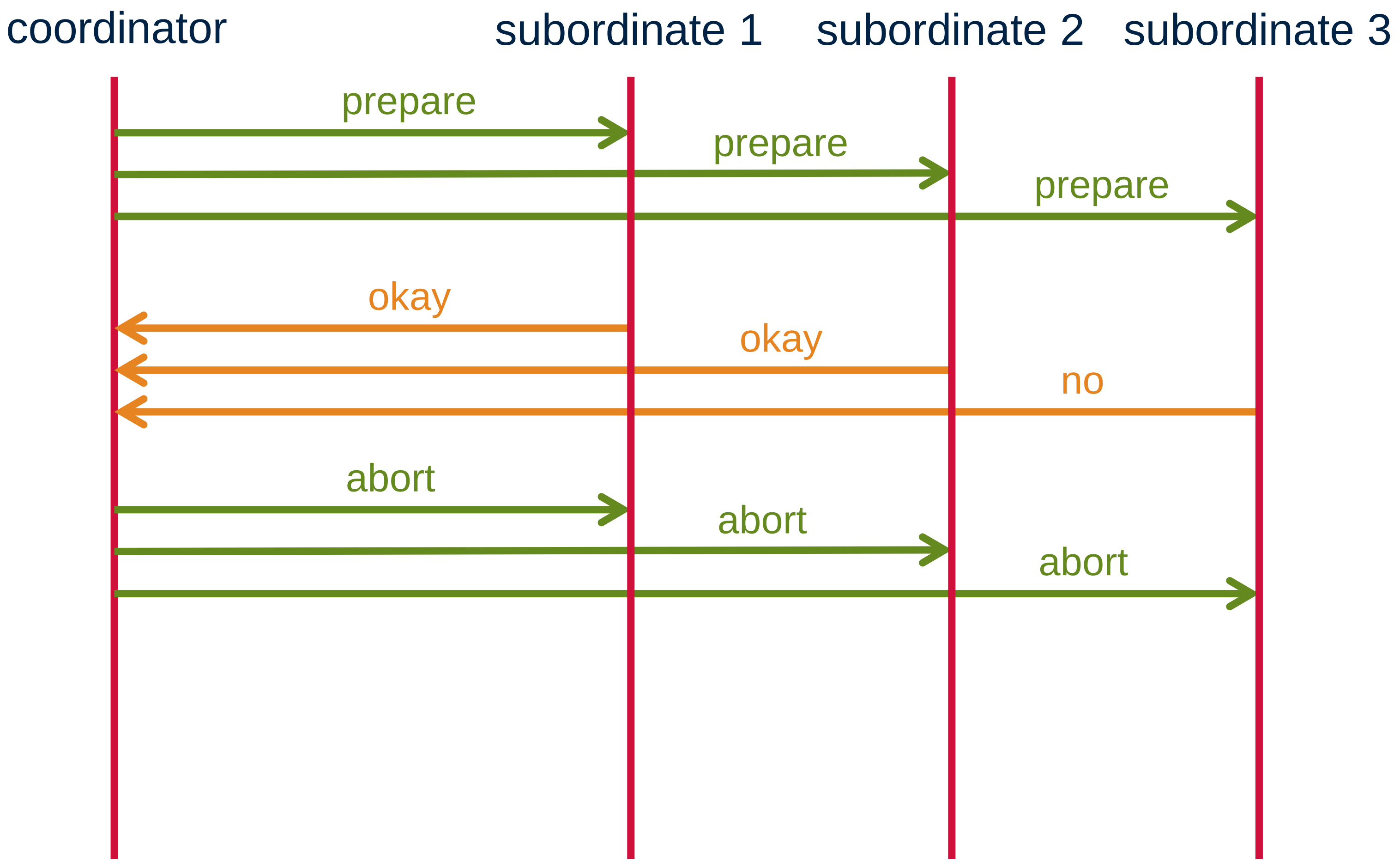
- RDBMSs provide
  - Relational model with schemas
  - Powerful, flexible query language
  - Transactional semantics: ACID
  - Rich ecosystem, lots of tool support
- Great, I'm sold! How do they do this?
  - Transactions on a single machine: (relatively) easy!
  - Partition tables to keep transactions 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
  - Two-phase commit (2PC)

2PC (two phase commit)

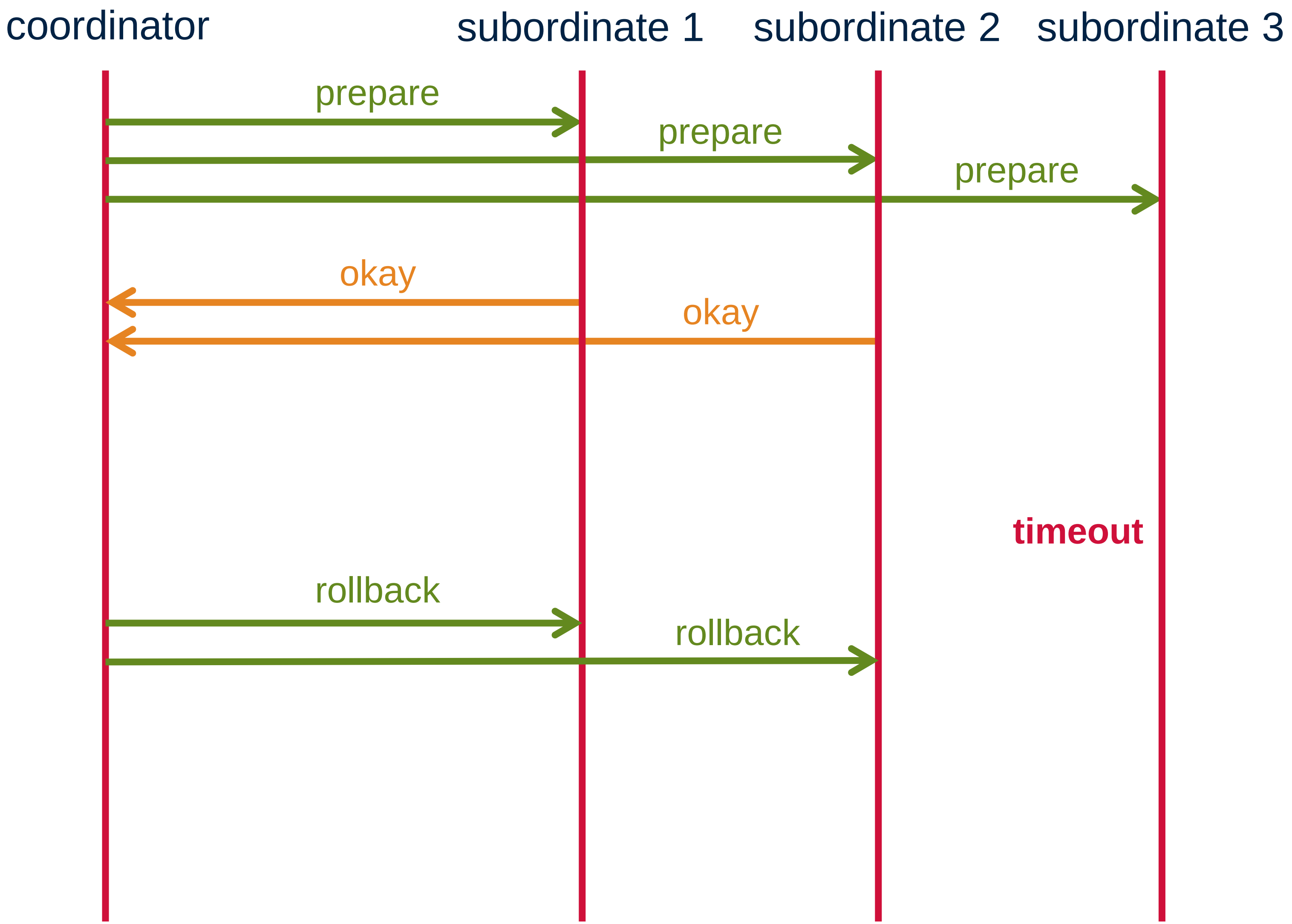


CWI Centrum Wiskunde & Informatica

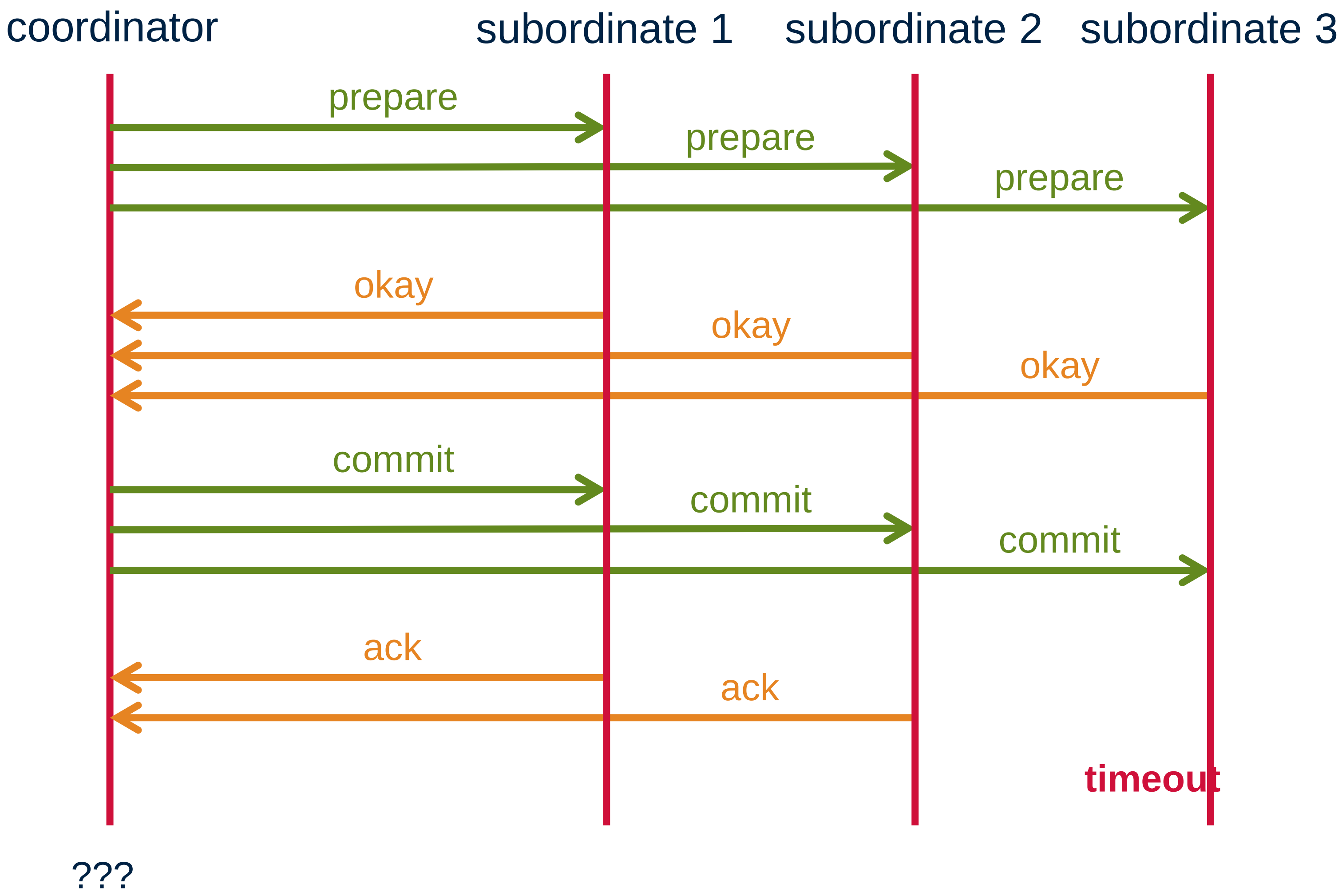
2PC abort



2PC rollback



2PC commit



coordinator
subordinate 1
subordinate 2
subordinate 3
prepare
prepare
prepare
okay
okay
okay
commit
commit
commit
ack
ack
timeout
???

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 ACID 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

Centrum Wiskunde & Informatica

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?

Centrum Wiskunde & Informatica

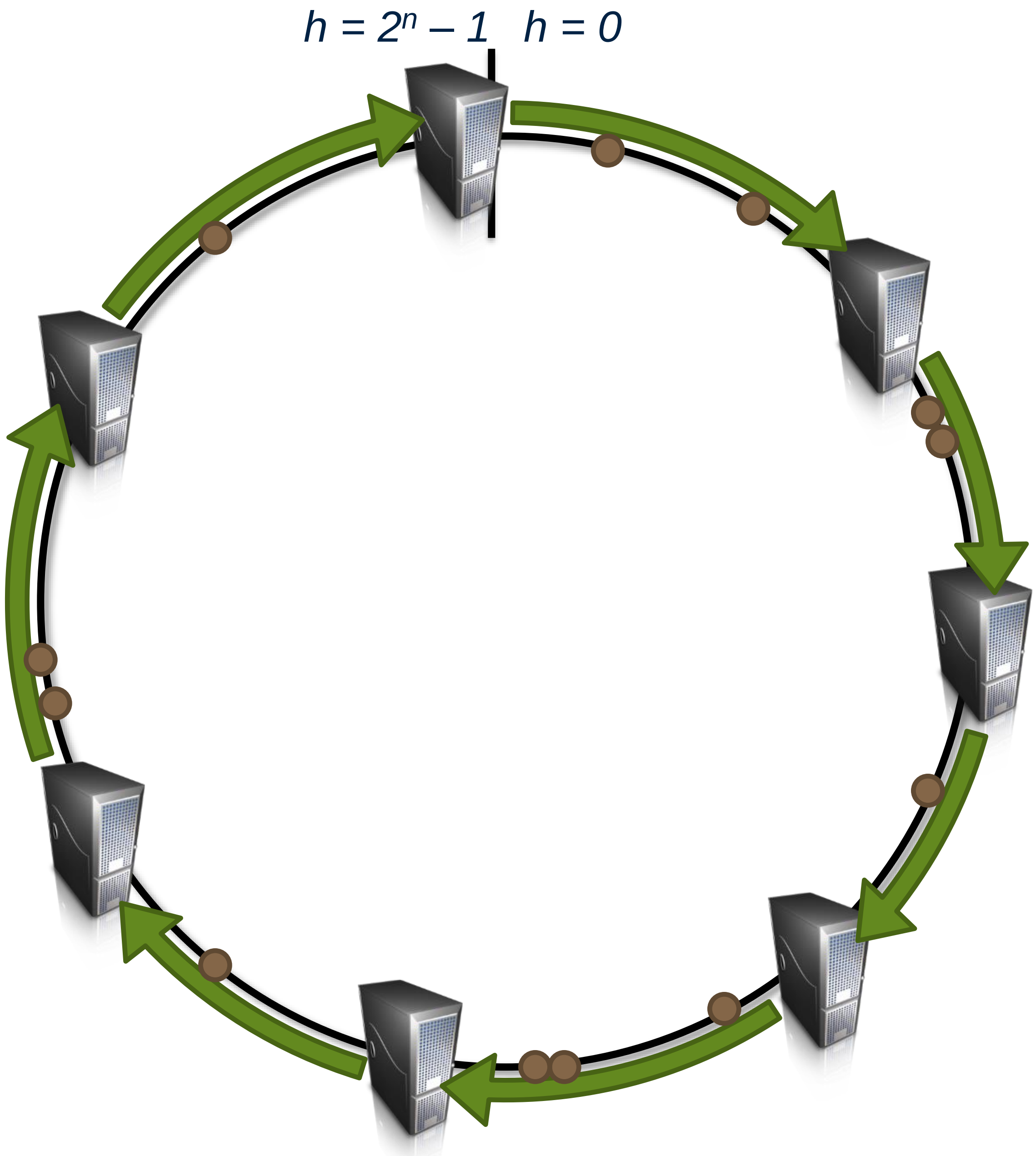
Key-value stores: implementation

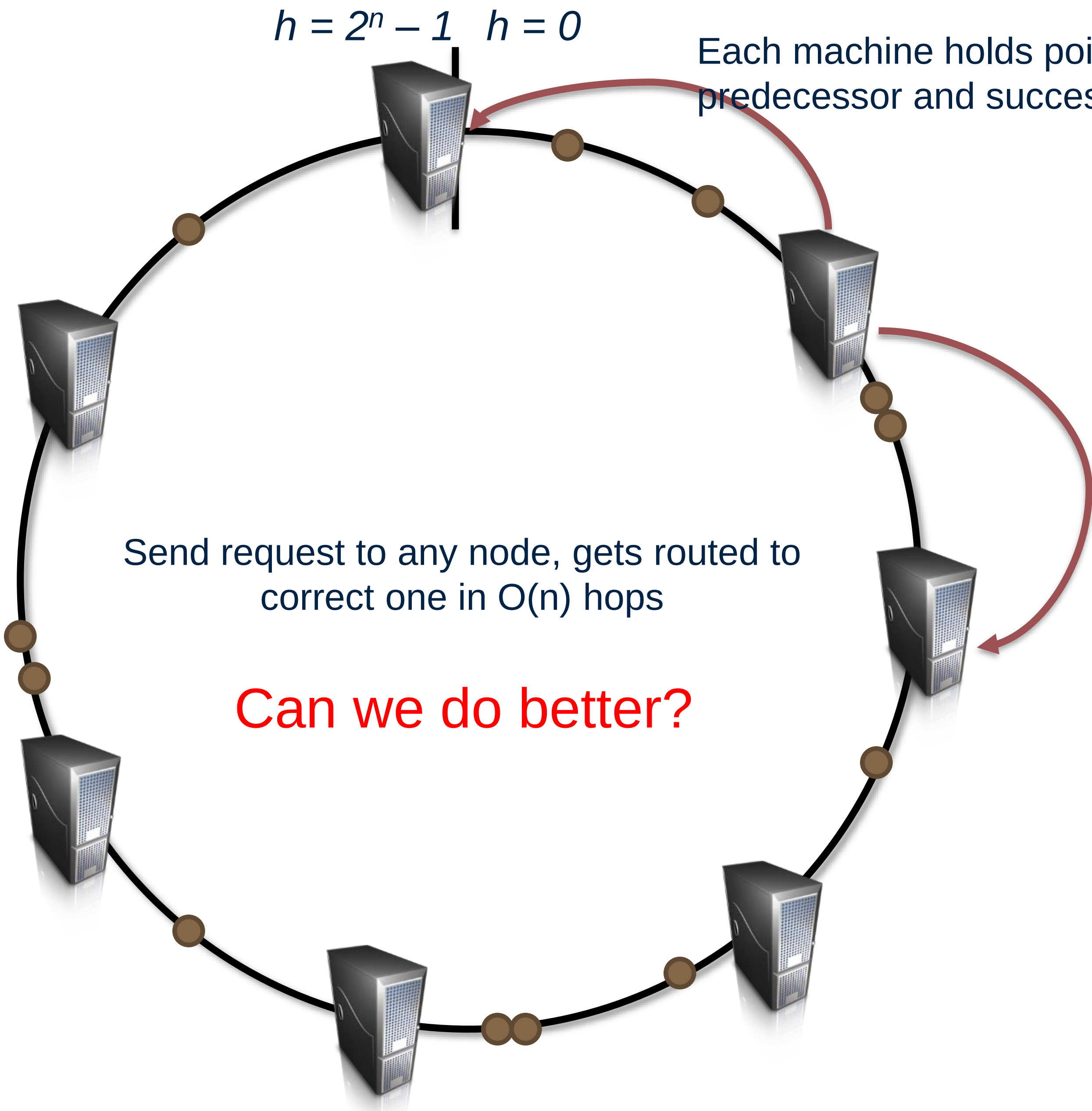
- Non-persistent:
  - Just a big in-memory hash table
- Persistent
  - Wrapper around a traditional RDBMS
- But what if data does not fit on a single machine?

Dealing with scale

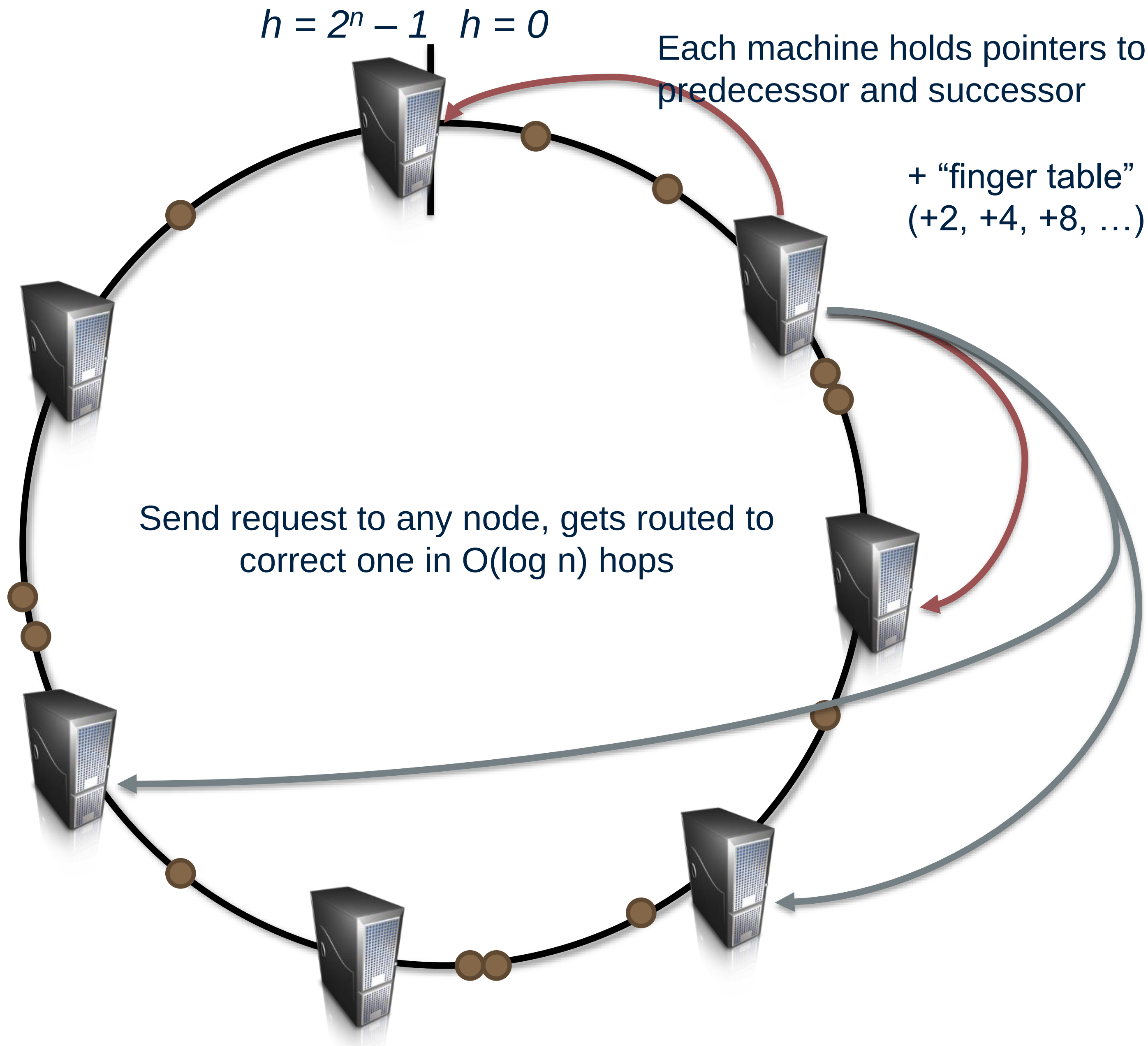
- Partition the key space across multiple machines
  - Let's say, hash partitioning
  - For *n* machines, store key *k* at machine *h(k)* mod *n*
- Okay… but:
  1. How do we know which physical machine to contact?
  2. How do we add a new machine to the cluster?
  3. What happens if a machine fails?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

DISTRIBUTED HASH TABLES: CHORD


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



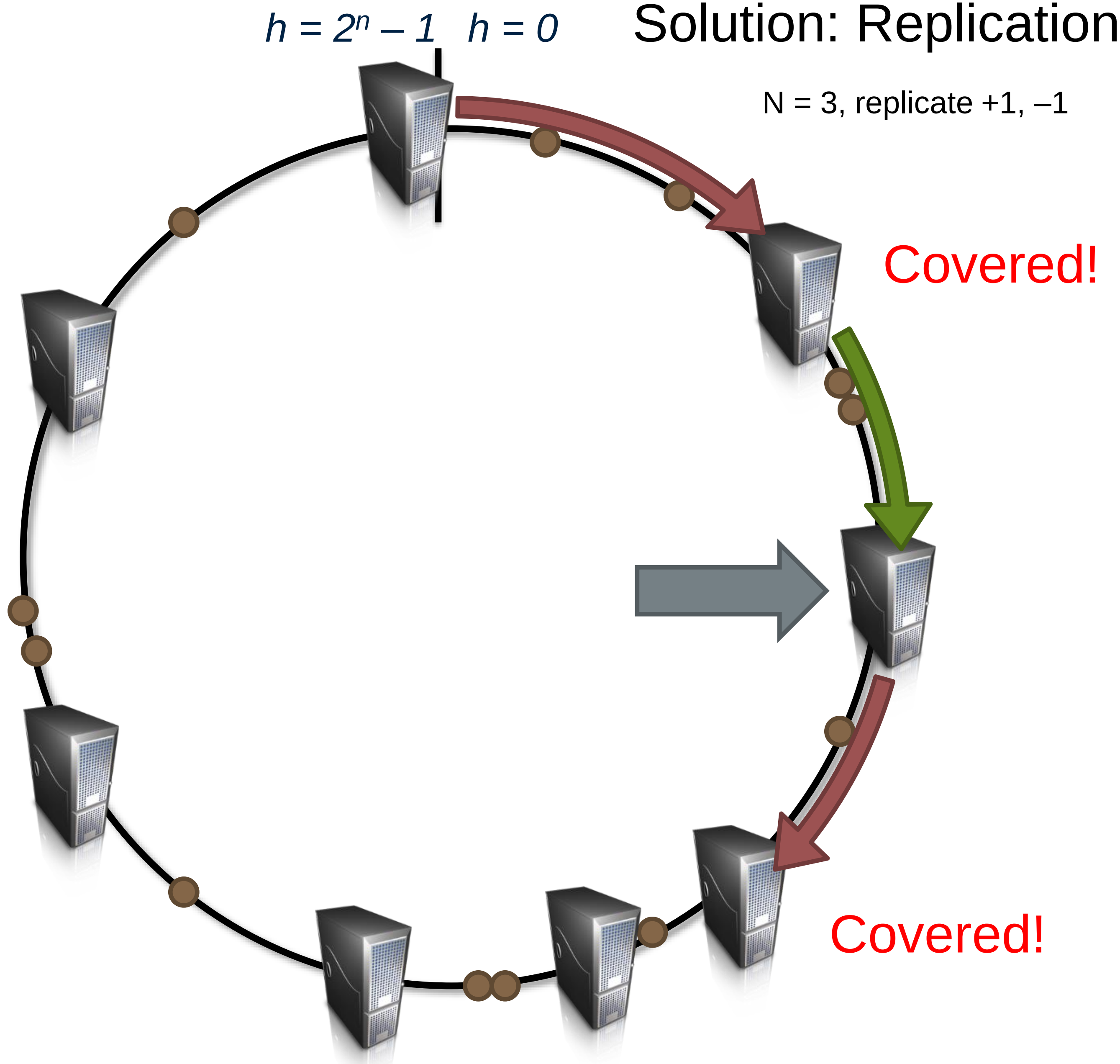
Routing: which machine holds the key?



Routing: which machine holds the key?

New machine joins: what happens?

Solution: Replication



Machine fails: what happens?

Centrum Wiskunde & Informatica

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?

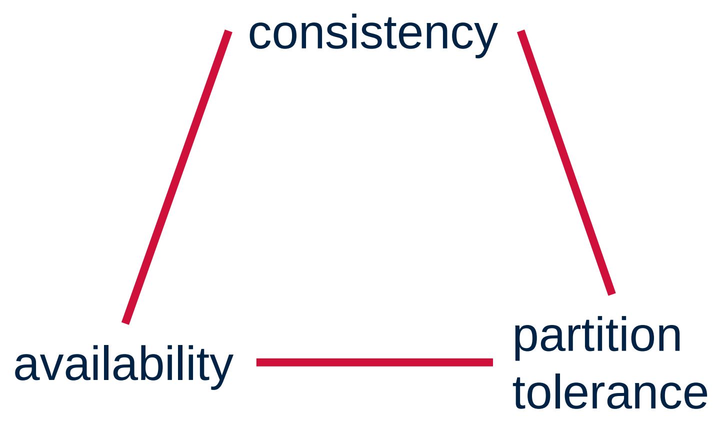
Three core ideas

- 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



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

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.

Centrum Wiskunde & Informatica

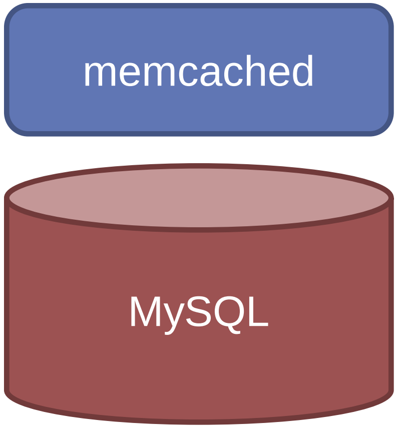
Three core ideas

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

Quick look at this

CWI
Centrum Wiskunde & Informatica

Facebook architecture

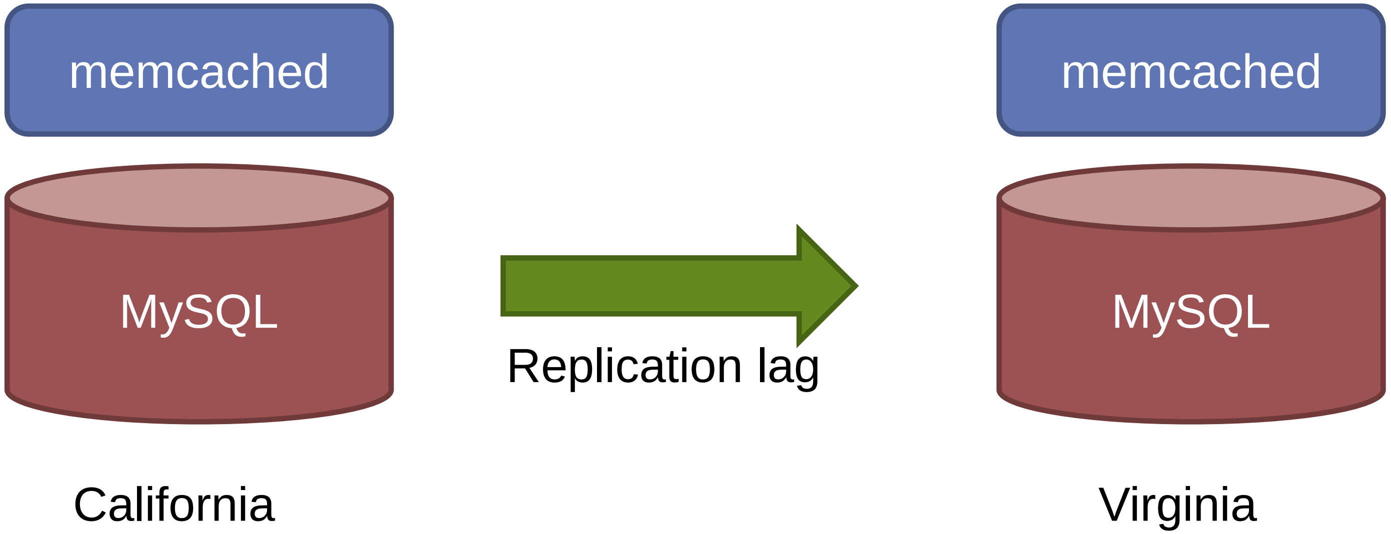


Read path:
Look in memcached
Look in MySQL
Populate in memcached

Write path:
Write in MySQL
Remove in memcached

Subsequent read:
Look in MySQL
Populate in memcached ✔

Facebook architecture: multi-DC



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!

THE BASE METHODOLOGY

Methodology versus model?

- An apples and oranges debate that has gripped the cloud community
  - A methodology is a way of doing something
    - For example, there is a methodology for starting fires without matches using flint and other materials
  - A model is really a mathematical construction
    - We give a set of definitions (i.e., fault-tolerance)
    - Provide protocols that provably satisfy the definitions
    - Properties of model, hopefully, translate to application-level guarantees

The ACID model

- A model for correct behavior of databases
- Name was coined (no surprise) in California in 60's
  - Atomicity
    - Either it all succeeds, or it all fails
    - Even if transactions have multiple operations, the rest of the world will either see all effects simultaneously (success), or no effects (failure)
  - Consistency
    - A transaction that runs on a correct database leaves it in a correct state
  - Isolation
    - It looks as if each transaction rusn all by itself.
    - Transactions are shielded from other transactions running concurrently
  - Durability
    - Once a transaction commits, updates cannot be lost or rolled back
    - Everything is permanent

ACID as a methodology

- We teach it all the time in our database courses
- We use it when developing systems
  - We write transactional code
  - System executes this code in an all-or-nothing way

```
Begin
    let employee t = Emp.Record("Tony");
    t.status = "retired";
    ∀ customer c: c.AccountRep=="Tony" →
        c.AccountRep = "Sally";
Commit;
```

Begin signals the start of the transaction

Body of the transaction performs reads and writes atomically

Commit asks the database to make the effects permanent. If a crash happens before this, or if the code executes **Abort**, the transaction rolls back and leaves no trace

Why is ACID helpful?

- Developer does not need to worry about a transaction leaving some sort of partial state
  - For example, showing Tony as retired and yet leaving some customer accounts with him as the account rep
- Similarly, a transaction cannot glimpse a partially completed state of some concurrent transaction
  - Eliminates worry about transient database inconsistency that might cause a transaction to crash
  - Analogous situation
    - Thread *A* is updating a linked list and thread *B* tries to scan the list while *A* is running
    - What if *A* breaks a link?
    - *B* is left dangling, or following pointers to nowhere-land

Serial and serialisable execution

- A serial execution is one in which there is at most one transaction running at a time, and it always completes via commit or abort before another starts
- Serialisability is the illusion of serial execution
  - Transactions execute concurrently and their operations interleave at the level of database accesses to primary data
  - Yet a database is designed to guarantee an outcome identical to some serial execution: it masks concurrency
    - This is achieved though some combination of locking and snapshot isolation

All ACID implementations have costs

- Locking mechanisms involve competing for locks
  - Overheads associated with maintaining locks
  - Overheads associated with duration of locks
  - Overheads associated with releasing locks on Commit
- Snapshot isolation mechanisms uses fine-grained locking for updates
  - But also have an additional version based way of handing reads
  - Forces database to keep a history of each data item
  - As a transaction executes, picks the versions of each item on which it will run

These costs are not so small

This motivates BASE

- Proposed by eBay researchers
  - Found that many eBay employees came from transactional database backgrounds and were used to the transactional style of thinking
  - But the resulting applications did not scale well and performed poorly on their cloud infrastructure
- Goal was to guide that kind of programmer to a cloud solution that performs much better
  - BASE reflects experience with real cloud applications
  - Opposite of ACID

D. Pritchett. BASE: An Acid Alternative. ACM Queue, July 28, 2008.

Centrum Wiskunde & Informatica

Not a model, but a methodology

- BASE involves step-by-step transformation of a transactional application into one that will be far more concurrent and less rigid
  - But it does not guarantee ACID properties
  - Argument parallels (and actually cites) CAP: they believe that ACID is too costly and often, not needed

BASE stands for *Basically Available Soft-State Services with Eventual Consistency*

Terminology

- Basically Available: Like CAP, goal is to promote rapid responses.
  - BASE papers point out that in data centers partitioning faults are very rare and are mapped to crash failures by forcing the isolated machines to reboot
  - But we may need rapid responses even when some replicas can't be contacted on the critical path

- Soft state service: Runs in first tier
  - Cannot store any permanent data
  - Restarts in a clean state after a crash
  - To remember data either replicate it in memory in enough copies to never lose all in any crash or pass it to some other service that keeps hard state

- Eventual consistency: OK to send optimistic answers to the external client
  - Could use cached data (without checking for staleness)
  - Could guess at what the outcome of an update will be
  - Might skip locks, hoping that no conflicts will happen
  - Later, if needed, correct any inconsistencies in an offline cleanup activity

How BASE is used

- Start with a transaction, but remove Begin/Commit
  - Now fragment it into steps that can be done in parallel, as much as possible
  - Ideally each step can be associated with a single event that triggers that step: usually, delivery of a multicast
- Leader that runs the transaction stores these events in a message queuing middleware system
  - Like an email service for programs
  - Events are delivered by the message queuing system
  - This gives a kind of all-or-nothing behavior

BASE in action

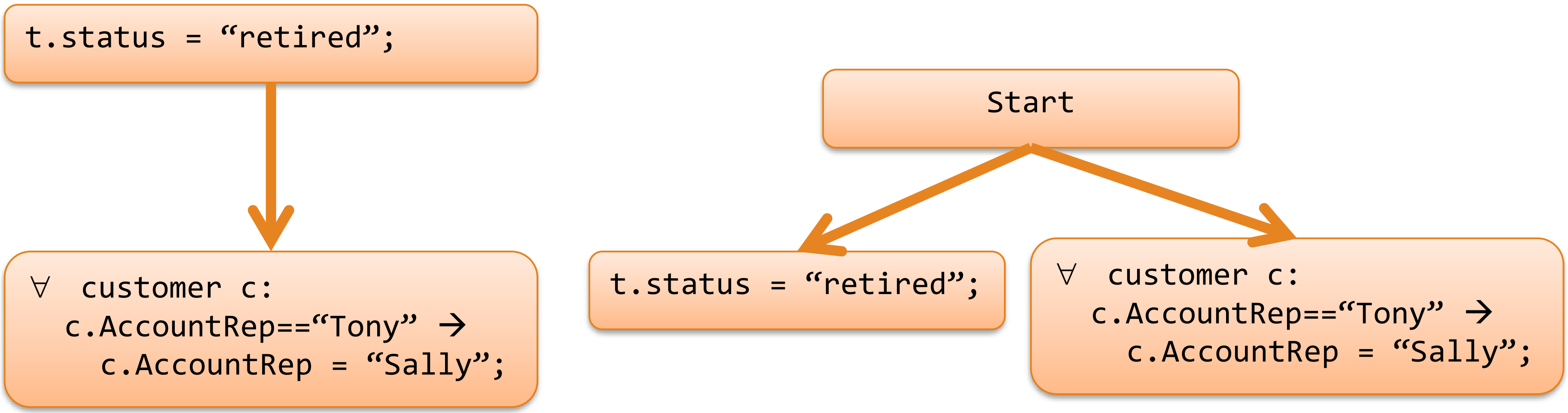
```
t.status = “retired”;
```

↓

```
∀ customer c:
  c.AccountRep==“Tony” →
    c.AccountRep = “Sally”;
```

```
Begin
    let employee t = Emp.Record(“Tony”);
    t.status = “retired”;
    ∀ customer c: c.AccountRep==“Tony” →
        c.AccountRep = “Sally”;
Commit;
```

BASE in action



- BASE suggestions
  - Consider sending the reply to the user before finishing the operation
  - Modify the end-user application to mask any asynchronous side-effects that might be noticeable
    - In effect, weaken the semantics of the operation and code the application to work properly anyhow
  - Developer ends up thinking hard and working hard!

Before BASE… and after

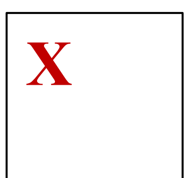
- Code was often much too slow
  - Poor scalability
  - End-users waited a long time for responses

- With BASE
  - Code itself is way more concurrent, hence faster
  - Elimination of locking, early responses, all make end-user experience snappy and positive
  - But we do sometimes notice oddities when we look hard

BASE side-effects

- Suppose an eBay auction is running fast and furious
  - Does every single bidder necessarily see every bid?
  - And do they see them in the identical order?
- Clearly, everyone needs to see the winning bid
- But slightly different bidding histories should not hurt much, and if this makes eBay 10x faster, the speed may be worth the slight change in behaviour!

- Upload a YouTube video, then search for it
  - You may not see it immediately
- Change the initial frame (they let you pick)
  - Update might not be visible for an hour

- Access a FaceBook page when your friend says she has posted a photo from the party
  - You may see an 

AMAZON DYNAMO

BASE in action: Dynamo

- Amazon was interested in improving the scalability of their shopping cart service

- A core component widely used within their system
  - Functions as a kind of key-value storage solution
  - Previous version was a transactional database and, just as the BASE folks predicted, was not scalable enough
  - Dynamo project created a new version from scratch

Dynamo approach

- Amazon made an initial decision to base Dynamo on a Chord-like Distributed Hash Table (DHT) structure
  - Recall Chord and its O(log n) routing ability
- The plan was to run this DHT in tier 2 of the Amazon cloud system
  - One instance of Dynamo in each Amazon data centre and no linkage between them
- This works because each data centre has ownership for some set of customers and handles all of that person's purchases locally
  - Coarse-grained sharding/partitioning

The challenge

- Amazon quickly had their version of Chord up and running, but then encountered a problem
- Chord was not very tolerant to delays
  - If a component gets slow or overloaded, the hash table was heavily impacted
- Yet delays are common in the cloud (not just due to failures, although failure is one reason for problems)
- So how could Dynamo tolerate delays?

The Dynamo idea

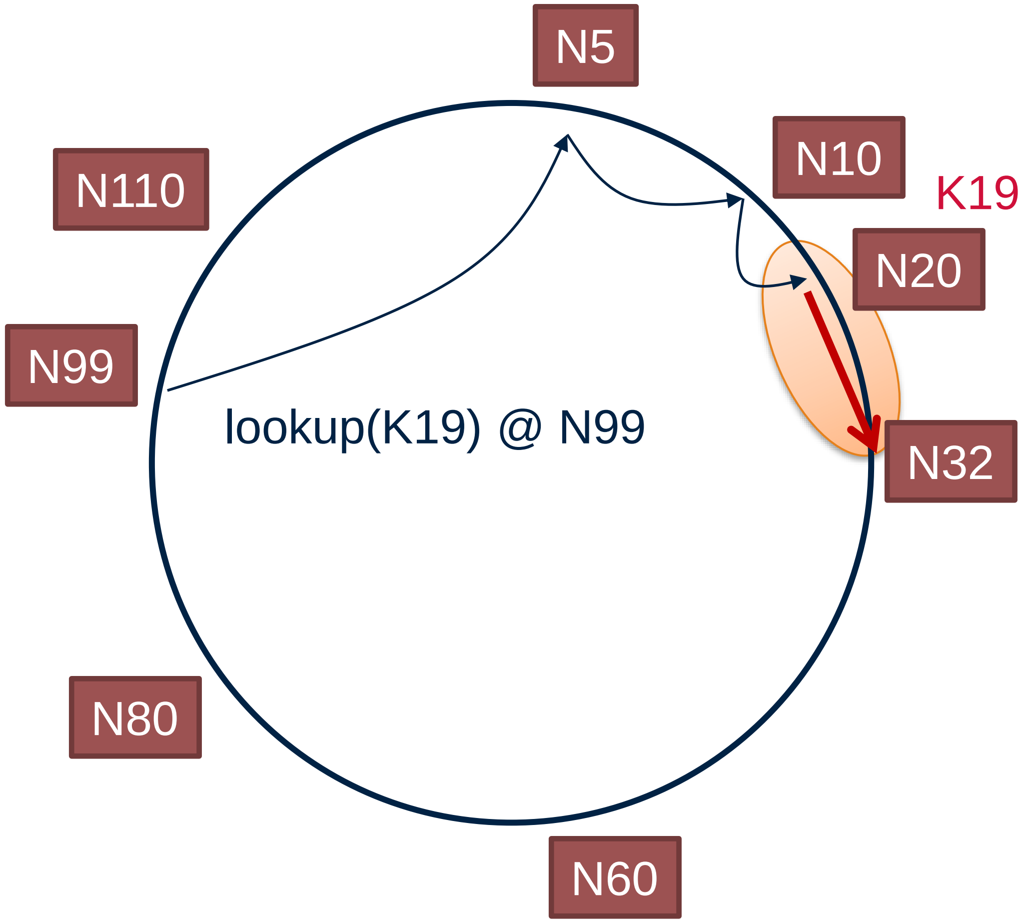
- The key issue is to find the node on which to store a key-value tuple, or one that has the value
- Routing can tolerate delay fairly easily
  - Suppose node K wants to use the finger table to route to node $K+2^i$ and gets no acknowledgement
  - Then Dynamo just tries again with node $K+2^{i-1}$
  - This works at the cost of a slight stretch in the routing path, in the rare cases when it occurs

What if the actual owner node fails?

- Suppose that we reach the point at which the next hop should take us to the owner for the hashed key
- But the target does not respond
  - It may have crashed, or have a scheduling problem (overloaded), or be suffering some kind of burst of network loss
  - All common issues in Amazon’s data centres
- Then they do the Get/Put on the next node that actually responds even if this is the wrong one
  - Chord will repair

Dynamo example

- Ideally, this strategy works perfectly
  - Chord normally replicates a key-value pair on a few nodes, so we would expect to see several nodes that know the current mapping: a shard
  - After the intended target recovers, the repair code will bring it back up to date by copying key-value tuples
- But sometimes Dynamo jumps beyond the target range and ends up in the wrong shard



Consequences of misrouting (and mis-storing)

- If this happens, Dynamo will eventually repair itself
  - But meanwhile, some slightly confusing things happen
- Put might succeed, yet a Get might fail on the key
- Could cause user to buy the same item twice
  - This is a risk they are willing to take because the event is rare and the problem can usually be corrected before products are shipped in duplicate

Werner Vogels on BASE

- He argues that delays as small as 100ms have a measurable impact on Amazon's income!
  - People wander off before making purchases
  - So snappy response is king
- True, Dynamo has weak consistency and may incur some delay to achieve consistency
  - There isn't any real delay bound
  - But they can hide most of the resulting errors by making sure that applications which use Dynamo don't make unreasonable assumptions about how Dynamo will behave

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
  - Cost of ACID in RDBMSs
  - Key,Value APIs
  - Caching, Replication,Partitioning
- BASE is a widely popular alternative to transactions (ACID)
  - Basically Available Soft-State Services with Eventual Consistency
  - Used (mostly) for first tier cloud applications
  - Weakens consistency for faster response, later cleans up
  - Consistency is eventual, not immediate
  - Complicates the work of the application developer
  - eBay, Amazon Dynamo shopping cart both use BASE