CS 61:
Database Systems

Distributed systems

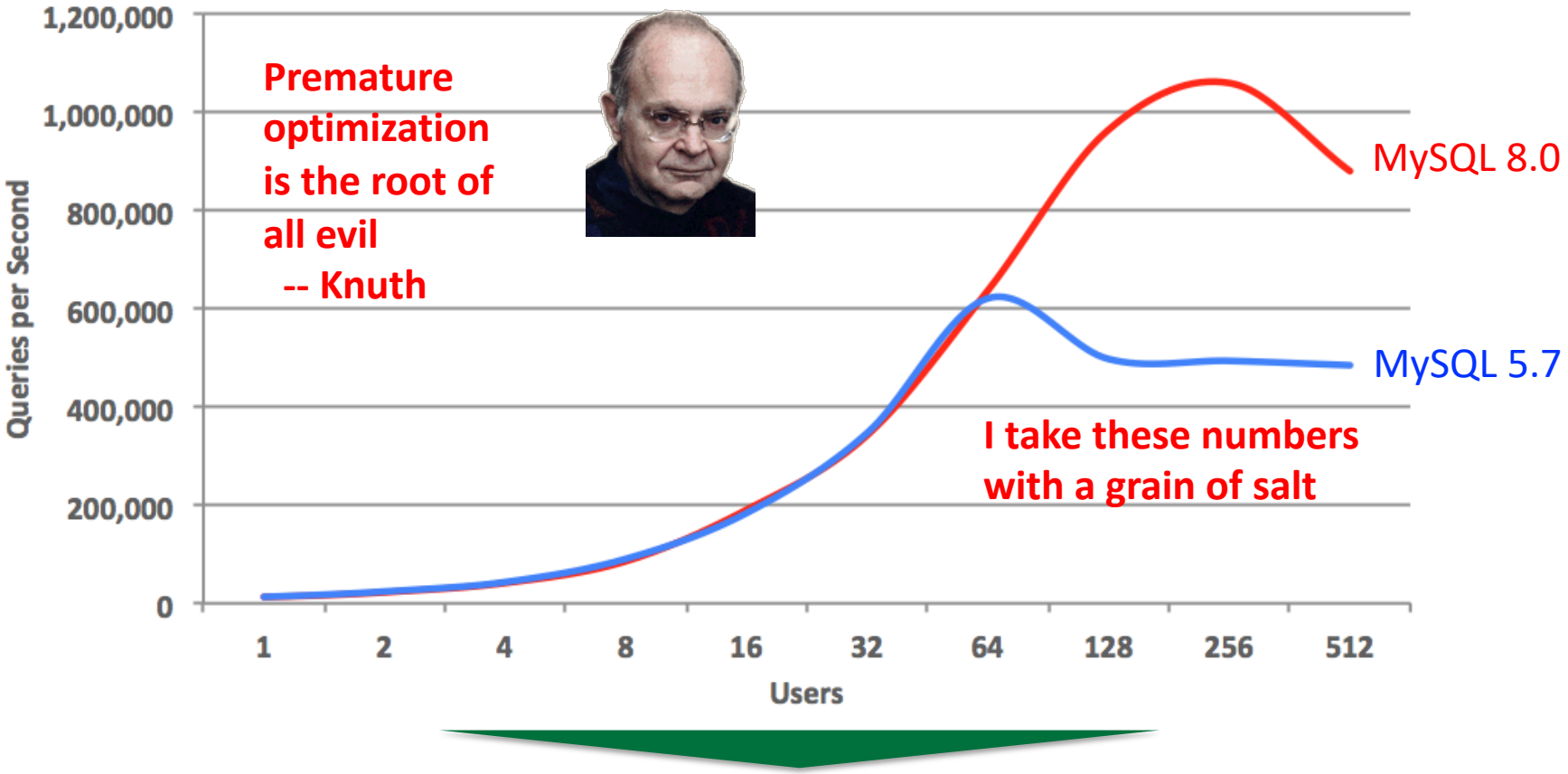
Agenda

1. Centralized systems

2. Distributed systems
   - High availability
   - Scalability

3. MongoDB

A single database can handle many thousands of transactions per second



Your start up that you're certain will be a smashing success in the market is unlikely to overwhelm a database running on a single reasonable server for quite some time

-- Pierson

Scale vertically – get a bigger box

Scale horizontally – get more boxes

Source: https://www.mysql.com/why-mysql/benchmarks/

Let's estimate performance

Assume:

Each user interaction takes 10 queries on average (normalization)

Average of 30 user interactions/visitor

Database can handle 100,000 queries per second

If you exceed these numbers, you'll need some help from someone who took more than an introductory database class!

Max user interactions/second:

$$\frac{100{,}000 \text{ queries}}{\text{second}} \times \frac{1 \text{ interaction}}{10 \text{ queries}} = \frac{10{,}000 \text{ interactions}}{\text{second}}$$

Max user interactions/day:

$$\frac{10{,}000 \text{ interactions}}{\text{second}} \times \frac{60*60*24 \text{ seconds}}{\text{day}} = \frac{864\text{M interactions}}{\text{day}}$$

Max user visits/day:

$$\frac{864\text{M interactions}}{\text{day}} \times \frac{1 \text{ visitor}}{30 \text{ interactions}} = \frac{28.8\text{M visitors}}{\text{day}}$$

Source: percona.com

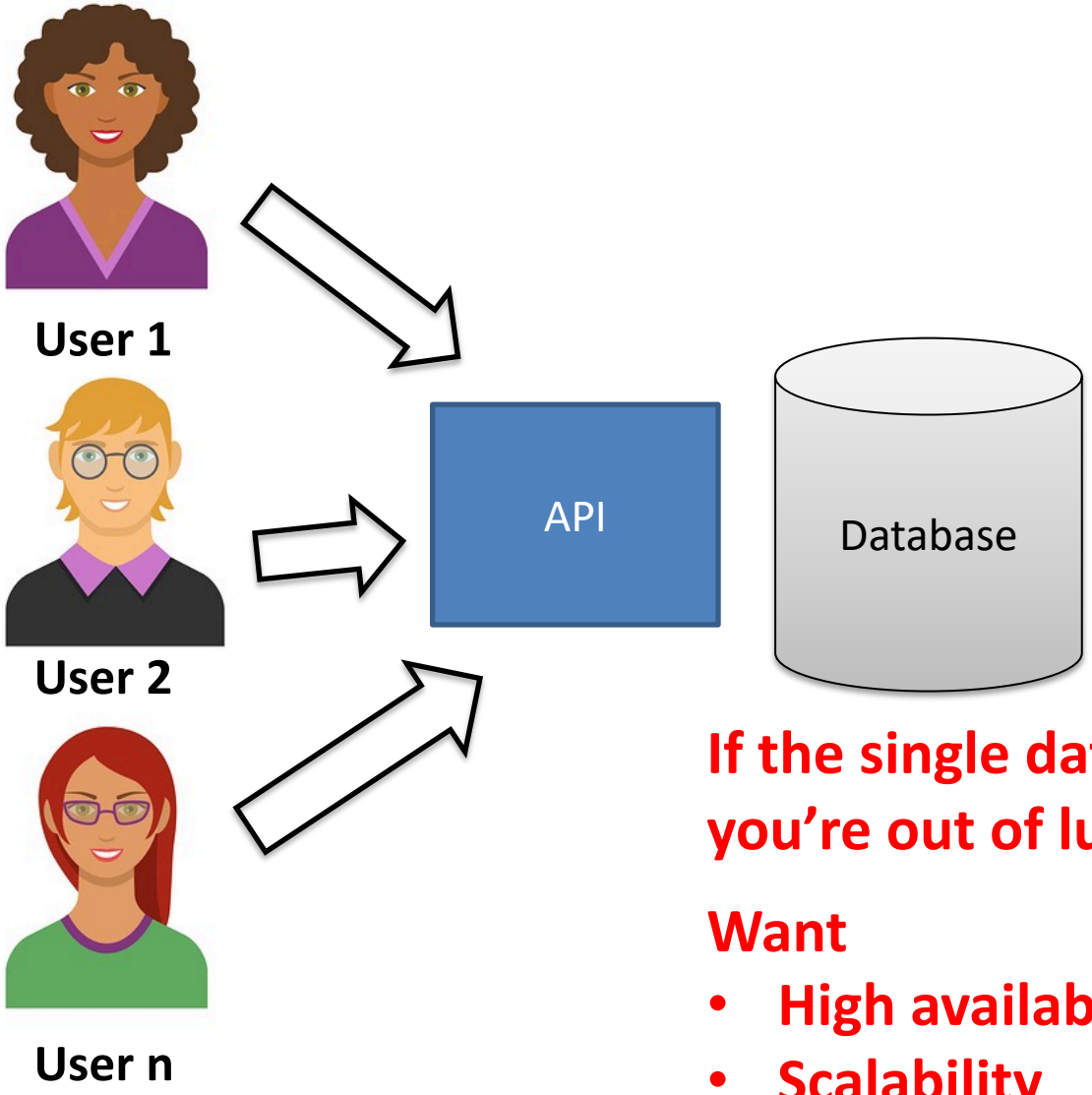
Agenda

1. Centralized systems

2. Distributed systems
   - High availability
   - Scalability

3. MongoDB

With one database, you've put all you eggs in one basket!



If the single database fails, you're out of luck

Want

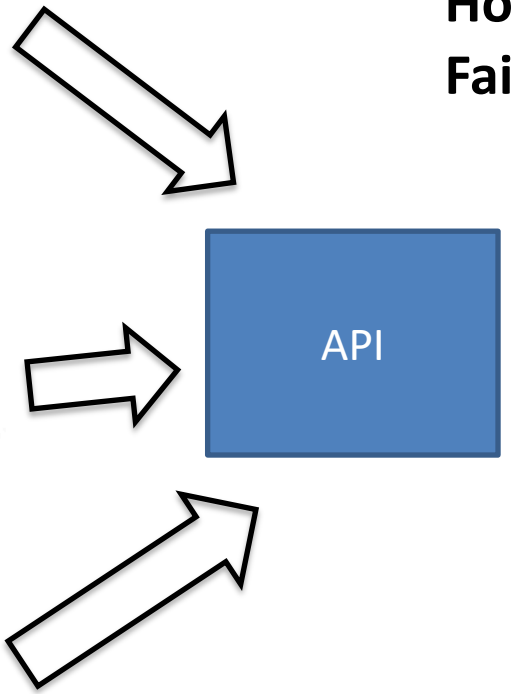
- **High availability**
- **Scalability**

With SANs you can have real-time, block-level replication to another database

User 1

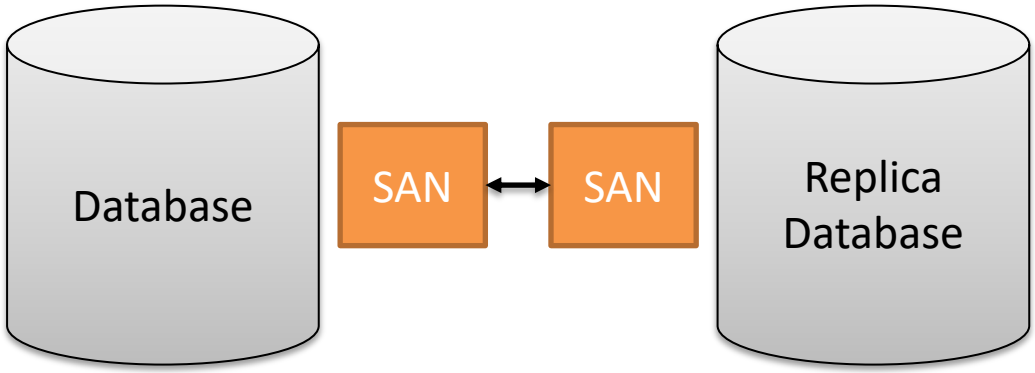
User 2

User n



Replica ideally located offsite

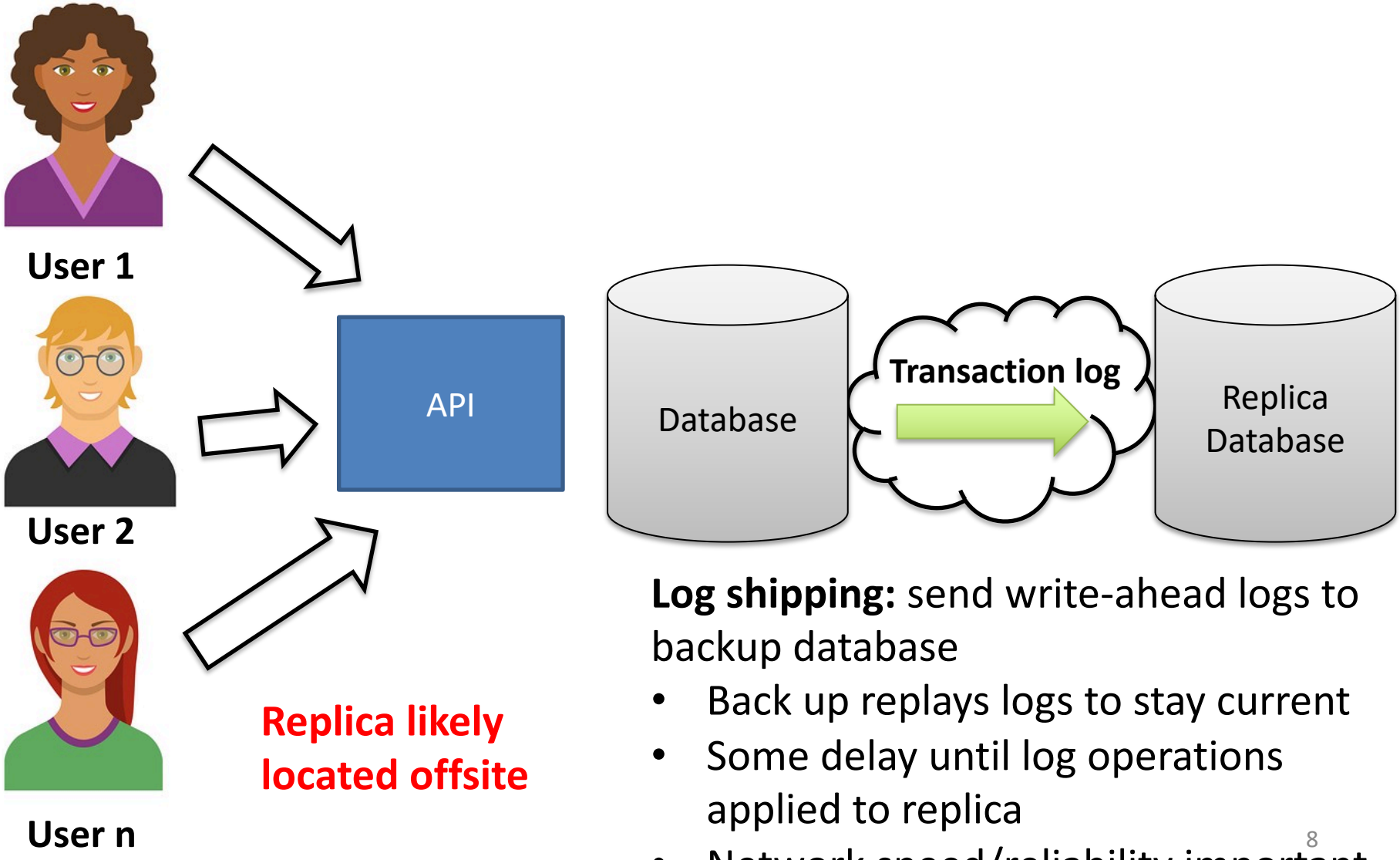
Cold standby: replica current to a point in time
Warm standby: replica kept current
Hot standby: replica is open for read-only ops
Failover: replica takes over for primary



SAN have block-level access

- Change made to one SAN immediately replicated to another SAN
- API accesses back up database if primary fails
- Expensive, but real-time

Log shipping is another, often more cost-effective high availability solution



Log shipping: send write-ahead logs to backup database

- Back up replays logs to stay current
- Some delay until log operations applied to replica
- Network speed/reliability important

Partitioning can help with scalability when data become large

Partitioning

Horizontal partitioning (aka sharding) slices data by rows and spreads across multiple nodes

Vertical partitioning slices data by columns

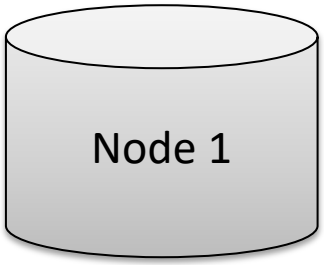
Logical data view

ID	Name	Salary
100	Alice	100,000
200	Bob	90,000
300	Charlie	85,000
…		

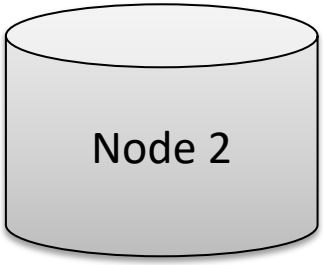
Partitioning increases capacity

- Each machine may be small, but only handles a subset of overall data
- Tradeoff: increased complexity

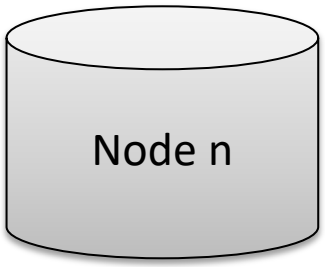
Global distributed schema keeps track of data locations



ID	Name	Salary
100	Alice	100,000
…		



ID	Name	Salary
200	Bob	90,000
…		



ID	Name	Salary
300	Charlie	85,000
…		

Adapted from Coronel and Morris

Sharding horizontally partitions data based on an attribute chosen as the shard key

Partitioning

Choose an attribute to serve as the shard key

- Shard key difficult change once database is partitioned
- Want cardinality > number of shards

Logical data view

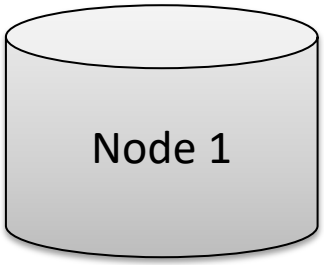
ID	Name	Salary
100	Alice	100,000
200	Bob	90,000
300	Charlie	85,000
…		

Hashing:

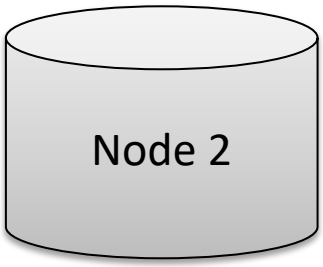
- Hash shard key to get replica number like CS10 hash table

Range partitioning:

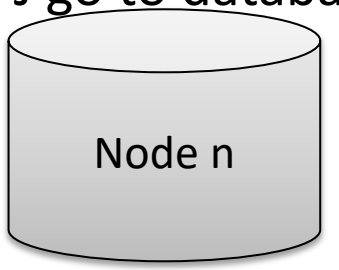
- Distribute based on a partition key (Names A-E go to database 1, F-J go to database 2, …



ID	Name	Salary
100	Alice	100,000
…		



ID	Name	Salary
200	Bob	90,000
…		



ID	Name	Salary
300	Charlie	85,000
…		

Adapted from Coronel and Morris

Problem: queries may affect multiple shards; use Two-Phase Commit (2PC)

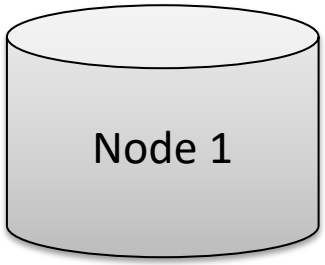
Two-phase commit (2PC) protocol

UPDATE Employee
SET Salary = Salary * 1.05

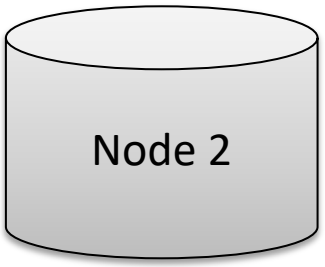
Two-phase commit: DO-UNDO-REDO protocol

- **DO:** Record before and after values in write ahead transaction log
- **UNDO:** Reverses operation using transaction log
- **REDO:** redoes an operation written by DO

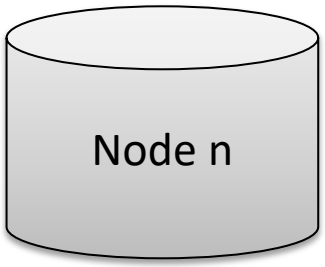
One node chosen as coordinator



ID	Name	Salary
100	Alice	100,000
…		



ID	Name	Salary
200	Bob	90,000
…		



ID	Name	Salary
300	Charlie	85,000
…		

Adapted from Coronel and Morris

Problem: queries may affect multiple shards; use Two-Phase Commit (2PC)

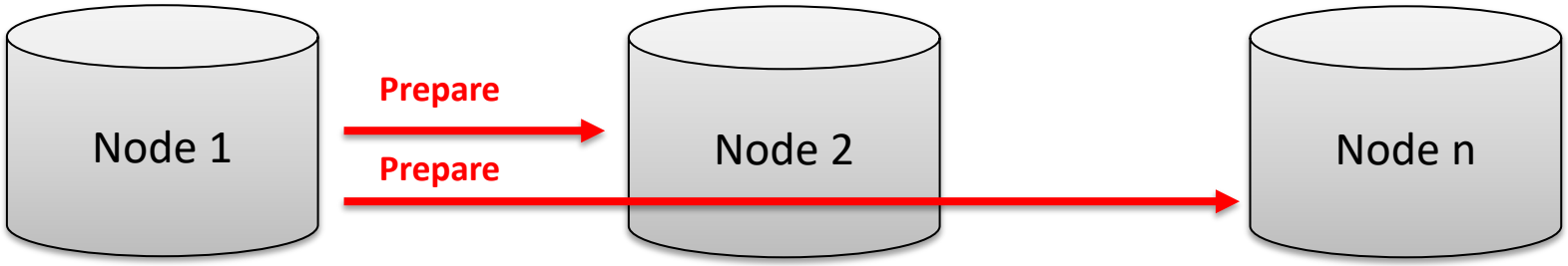
Two-phase commit (2PC) protocol

UPDATE Employee
SET Salary = Salary * 1.05

Phase 1: Preparation

- Coordinator send Prepare to Commit message to all subordinate nodes
- Subordinates write transaction log and send acknowledgement to coordinator
- Coordinator ensures all nodes ready to commit or aborts

One node chosen as coordinator



ID	Name	Salary
100	Alice	100,000
…		

ID	Name	Salary
200	Bob	90,000
…		

ID	Name	Salary
300	Charlie	85,000
…		

Adapted from Coronel and Morris

Problem: queries may affect multiple shards; use Two-Phase Commit (2PC)

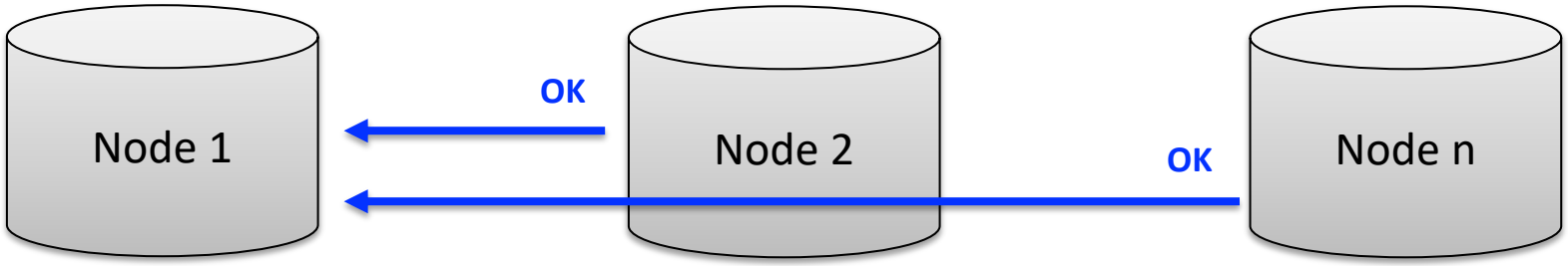
Two-phase commit (2PC) protocol

UPDATE Employee
SET Salary = Salary * 1.05

Phase 1: Preparation

- Coordinator send Prepare to Commit message to all subordinate nodes
- Subordinates write transaction log and send acknowledgement to coordinator
- Coordinator ensures all nodes ready to commit or aborts

One node chosen as coordinator



ID	Name	Salary
100	Alice	100,000
…		

ID	Name	Salary
200	Bob	90,000
…		

ID	Name	Salary
300	Charlie	85,000
…		

Adapted from Coronel and Morris

Problem: queries may affect multiple shards; use Two-Phase Commit (2PC)

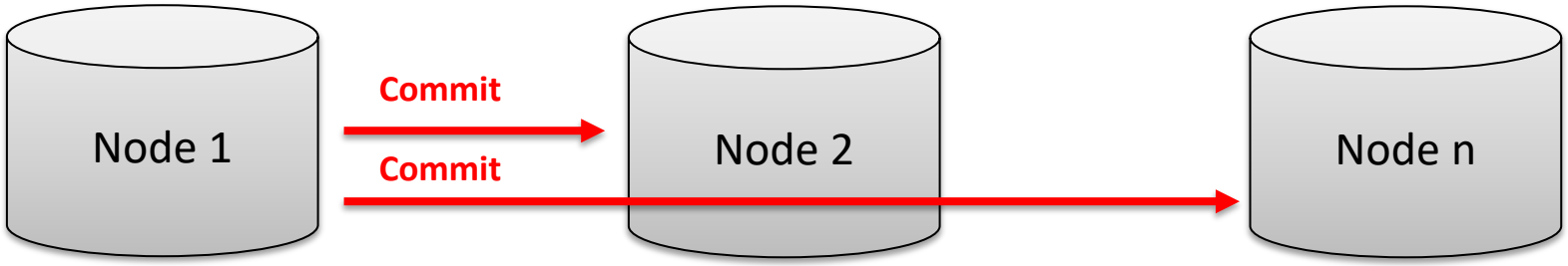
Two-phase commit (2PC) protocol

UPDATE Employee
SET Salary = Salary * 1.05

Phase 2: Commit

- Coordinator broadcasts commit message
- Each subordinate updates with DO
- Subordinates reply with COMMITTED or NOT COMMITTED
- If any nodes reply NOT COMMITTED, then UNDO followed by REDO

One node chosen as coordinator



ID	Name	Salary
100	Alice	100,000
…		

ID	Name	Salary
200	Bob	90,000
…		

ID	Name	Salary
300	Charlie	85,000
…		

Adapted from Coronel and Morris

Problem: queries may affect multiple shards; use Two-Phase Commit (2PC)

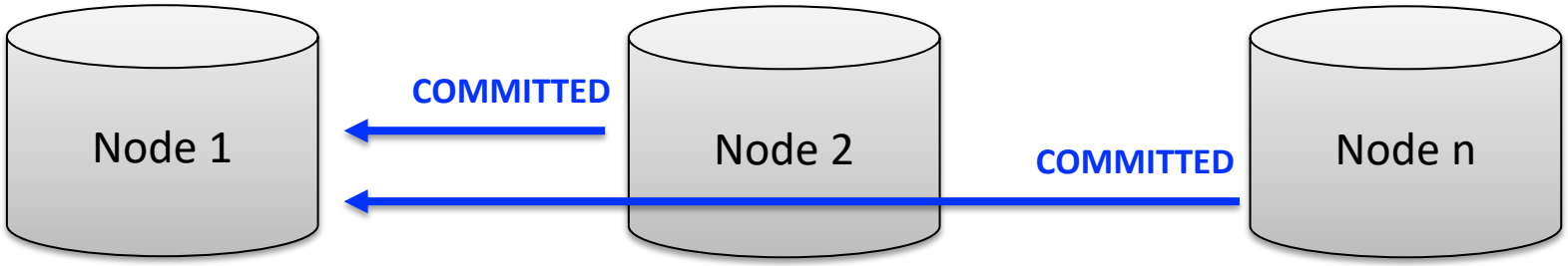
Two-phase commit (2PC) protocol

UPDATE Employee
SET Salary = Salary * 1.05

Phase 2: Commit

- Coordinator broadcasts commit message
- Each subordinate updates with DO
- Subordinates reply with COMMITTED or NOT COMMITTED
- If any nodes reply NOT COMMITTED, then UNDO followed by REDO

One node chosen as coordinator



ID	Name	Salary
100	Alice	100,000
…		

ID	Name	Salary
200	Bob	90,000
…		

ID	Name	Salary
300	Charlie	85,000
…		

Adapted from Coronel and Morris

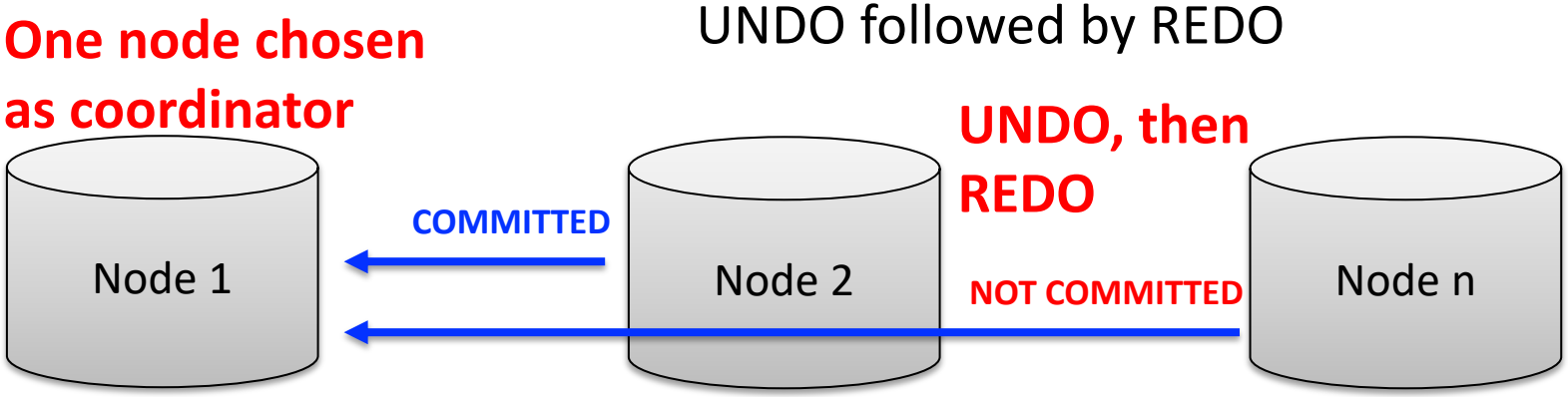
Problem: queries may affect multiple shards; use Two-Phase Commit (2PC)

Two-phase commit (2PC) protocol

UPDATE Employee
SET Salary = Salary * 1.05

Phase 2: Commit

- Coordinator broadcasts commit message
- Each subordinate updates with DO
- Subordinates reply with COMMITTED or NOT COMMITTED
- If any nodes reply NOT COMMITTED, then UNDO followed by REDO



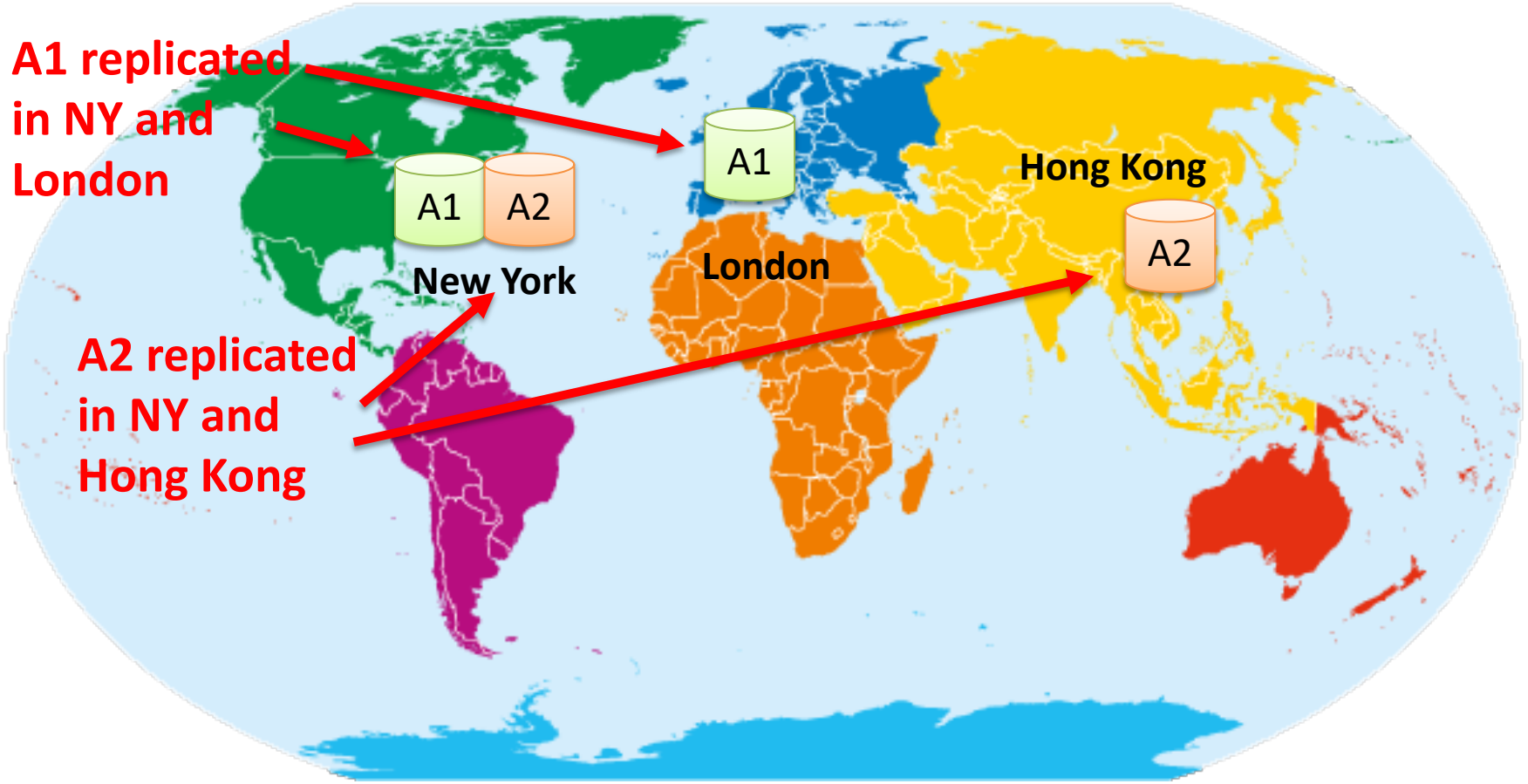
ID	Name	Salary
100	Alice	100,000
...		

ID	Name	Salary
200	Bob	90,000
...		

ID	Name	Salary
300	Charlie	85,000
...		

Adapted from Coronel and Morris

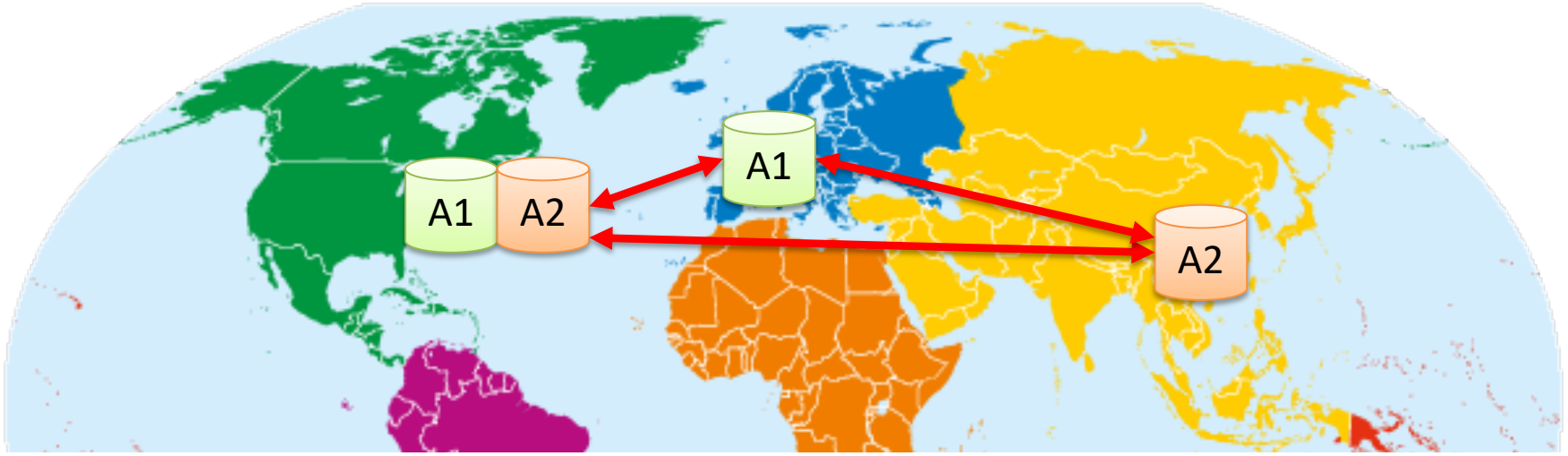
Data might be replicated to several nodes located across the globe



Data replication scenarios

- **Fully replicated:** multiple copies of each database partition at multiple sites
- **Partially replicated**: multiple copies of some database partitions at multiple sites
- **Unreplicated:** stores each database partition at a single site

All replicated nodes should have same data, but network latency raises issues

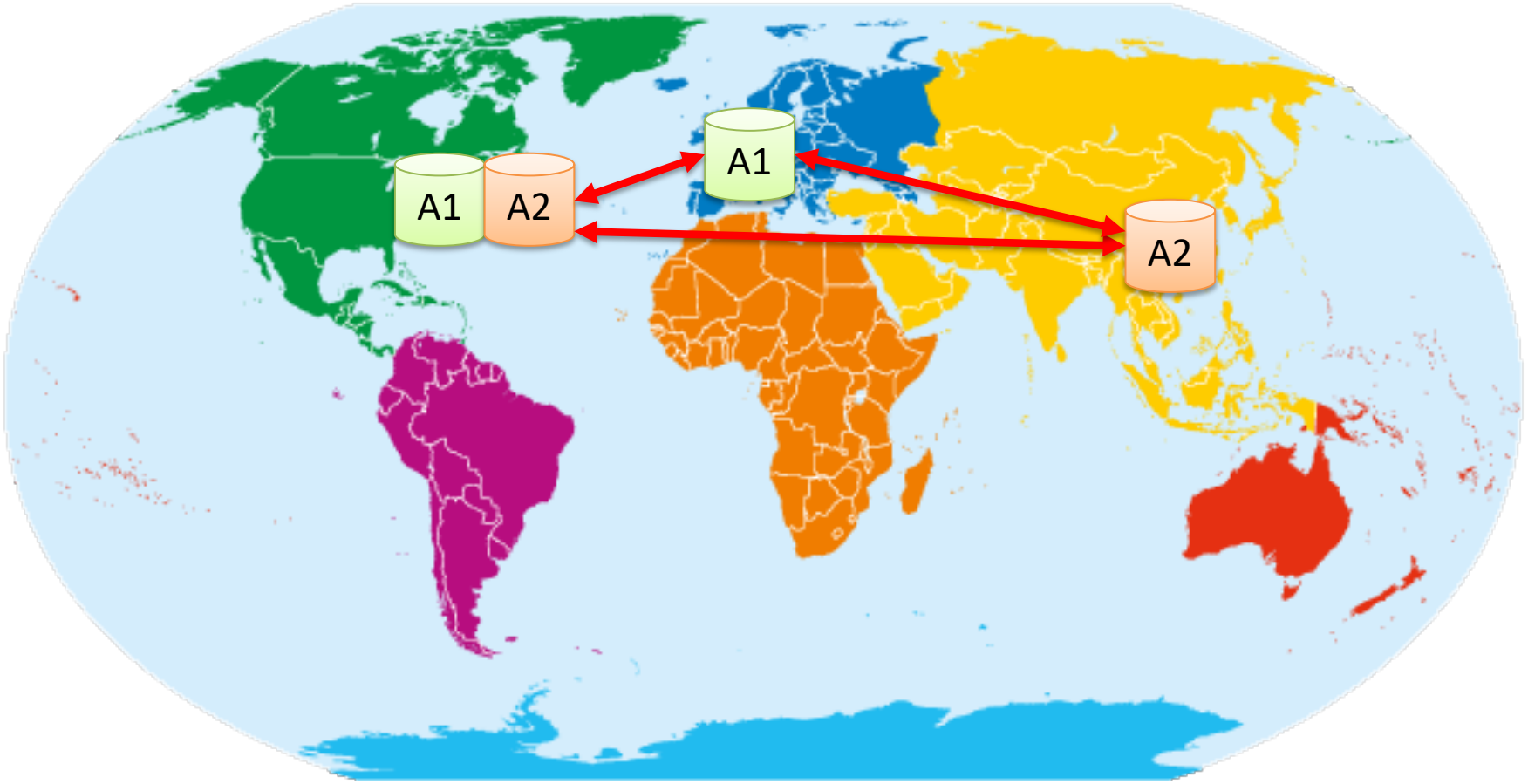


A potential problem: consistency rule says all copies must be identical when data changes are made

Read operations just query the nearest replica

- **Push replication** (focus on consistency)
  - After data update, send changes to all replicas
  - Data unavailable until changes to propagate across all copies, but data is always consistent across copies
- **Pull replication** (focus on availability)
  - Send message to all replicas, they decide when to apply change
  - Data is available, but not consistent until changes propagate

The network may be partitioned by communication breaks

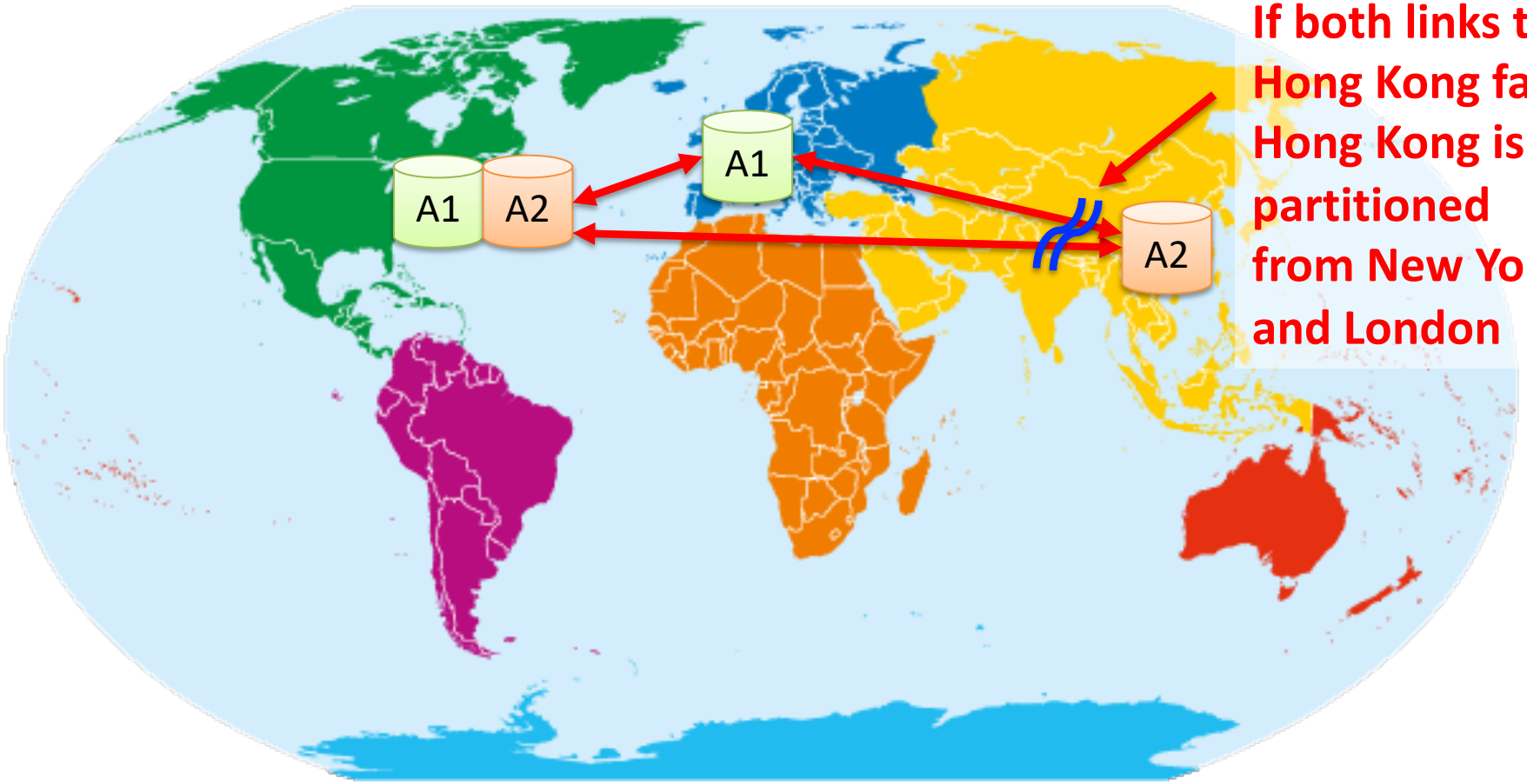


The network may have multiple communication links between each node

If one link fails, other nodes will still be reachable

Multiple link failure, however, may separate some nodes – called a network partition

The network may be partitioned by communication breaks



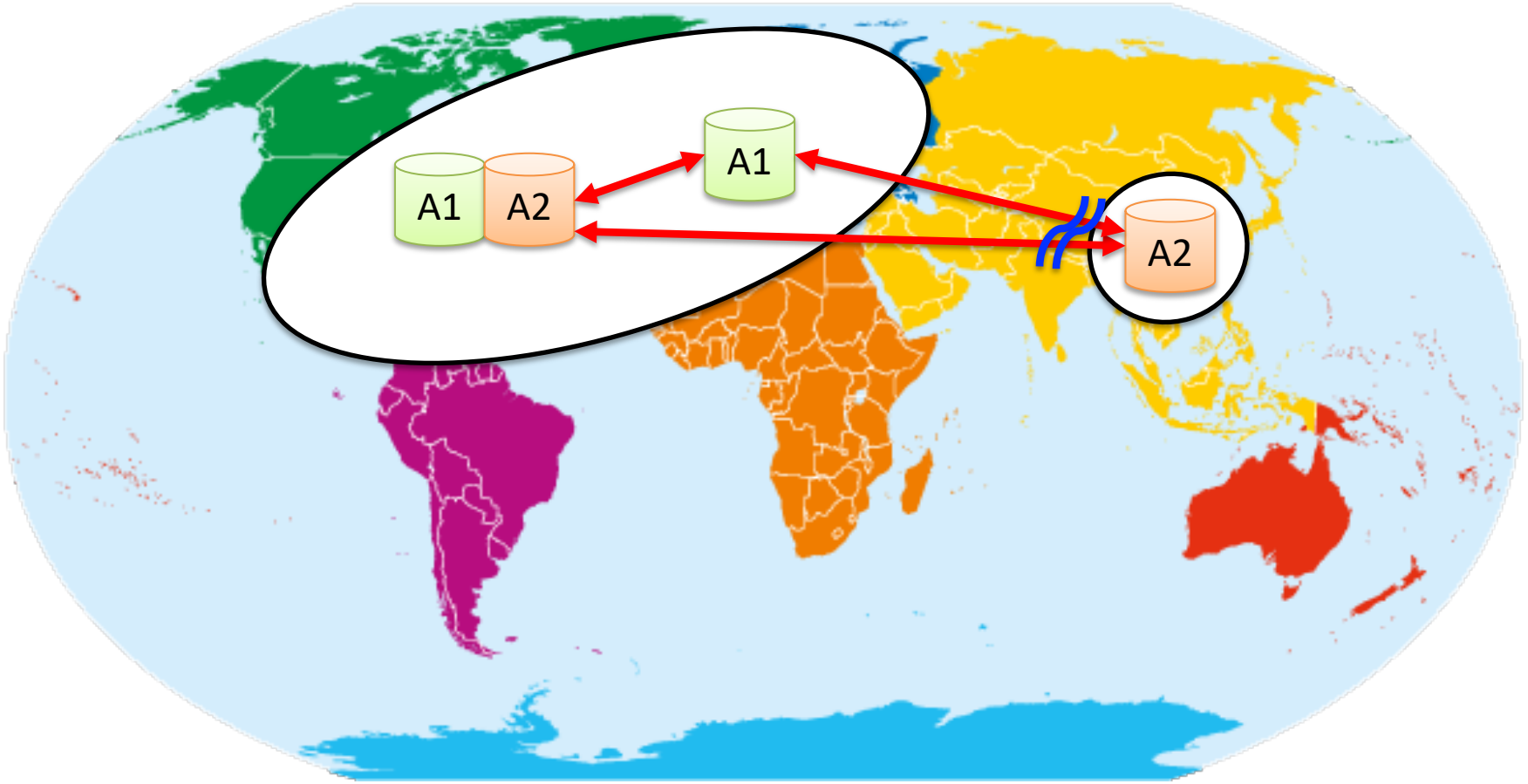
If both links to Hong Kong fail, Hong Kong is partitioned from New York and London

The network may have multiple communication links between each node

If one link fails, other nodes will still be reachable

Multiple link failure, however, may separate some nodes – called a network partition

The network may be partitioned by communication breaks



The network may have multiple communication links between each node

If one link fails, other nodes will still be reachable

Multiple link failure, however, may separate some nodes – called a network partition

It impossible to be consistent, available, and partition tolerant simultaneously

CAP theorem

The CAP theorem showed that it is impossible to have three desirable properties at the same time in distributed systems

Consistency

- All nodes should return the same data at the same time
- Replicas should be immediately updated
- Network latency means this cannot happen

Availability

- A request is always fulfilled by the system
- No request is ever lost

Partition tolerant

- The system will operate even if nodes fail
- Operations that are lost due to node failure are picked up by other nodes
- The system will only fail if all nodes fail

Trade-off between consistency and availability

BASE rather than ACID: Basically Available, Soft state, Eventually consistent (BASE)

Data changes are not immediate but propagate slowly through the system until all replicas are eventually consistent

Eric A. Brewer, "Towards robust distributed systems," *Principles of Distributed Computing*, ACM, July 2000.

Agenda

1. Centralized systems

2. Distributed systems
   - High availability
   - Scalability

3. MongoDB

MongoDB is a NoSQL database designed for high availability and scalability

MongoDB is a document database designed for scalability and flexibility

- MongoDB is “schemaless”
  - Stores data in JSON-like documents
  - Fields can vary from document to document
  - Data structure can change over time
- MongoDB is a distributed database at its core
  - High availability (replication)
  - Horizontal scaling (sharding)
  - Geographic distribution built in
- MongoDB is free to use
  - Cloud-based version at MongoDB Atlas (https://www.mongodb.com/cloud/atlas)

Adapted from Mongo documentation

Mongo solves the high availability problem using two different types of nodes

Mongo replication

Replica set: Group of database servers that provide high availability and redundancy using two different types of nodes

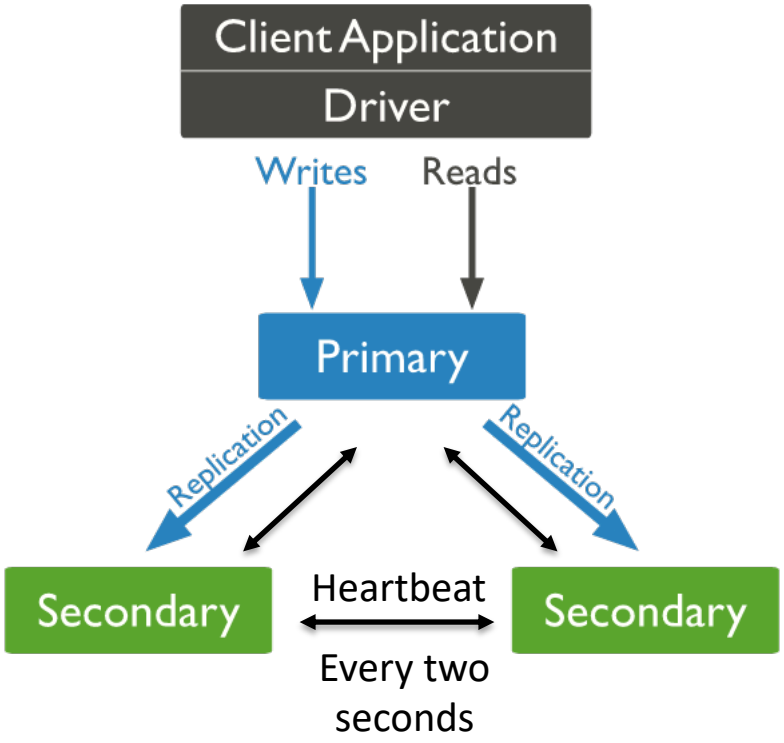
- **Primary:** Receives all write operations
- **Secondary:** replicate operations from primary to maintain an identical data set

Mongo recommends at least a three-member replica set (more even better)

All members of replica set can accept read operations

Replica sets store the same data in all nodes

Purpose: redundancy for high availability



Adapted from Mongo documentation

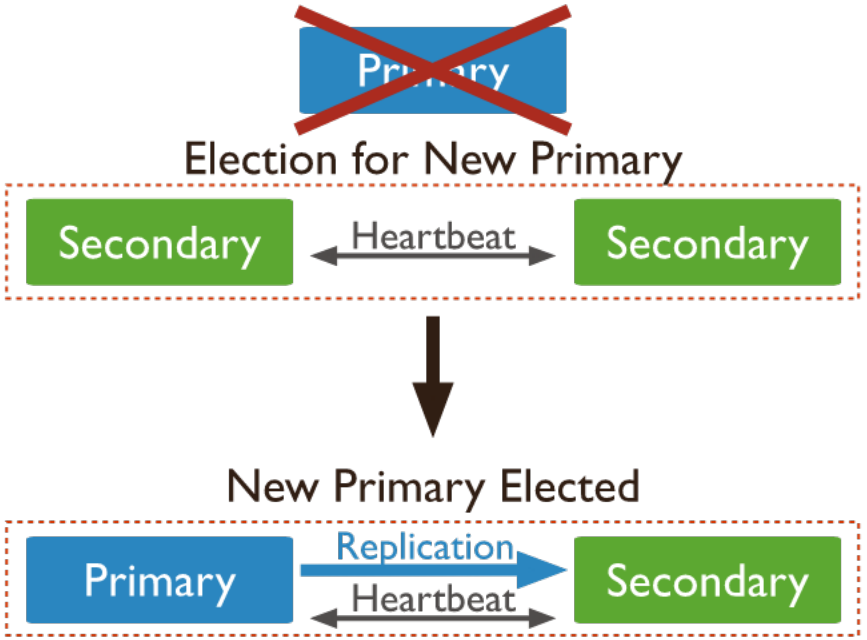
If the primary nodes fails, other nodes hold an election to pick a new primary

Mongo replication

Replica sets use elections to determine which set member will be primary

Elections triggered:

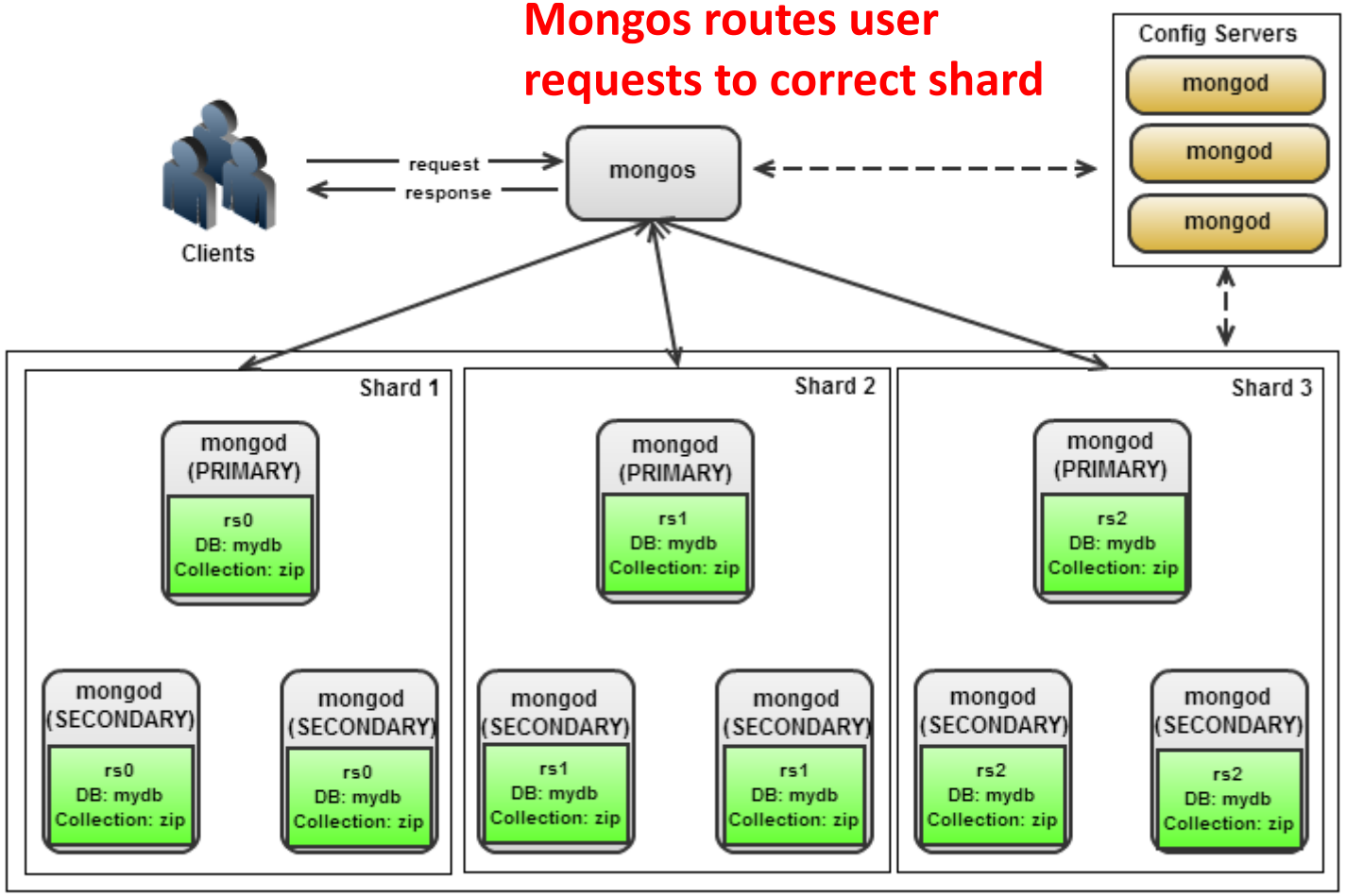
- Start up
- New node added to set
- Secondary member looses connectivity to primary for 10 seconds (default)
- Called automatic failover when new primary takes over



Adapted from Mongo documentation

Mongo shards data across multiple nodes, each shard can be replicated

Replication with sharding



Config servers keep track of data location in shards

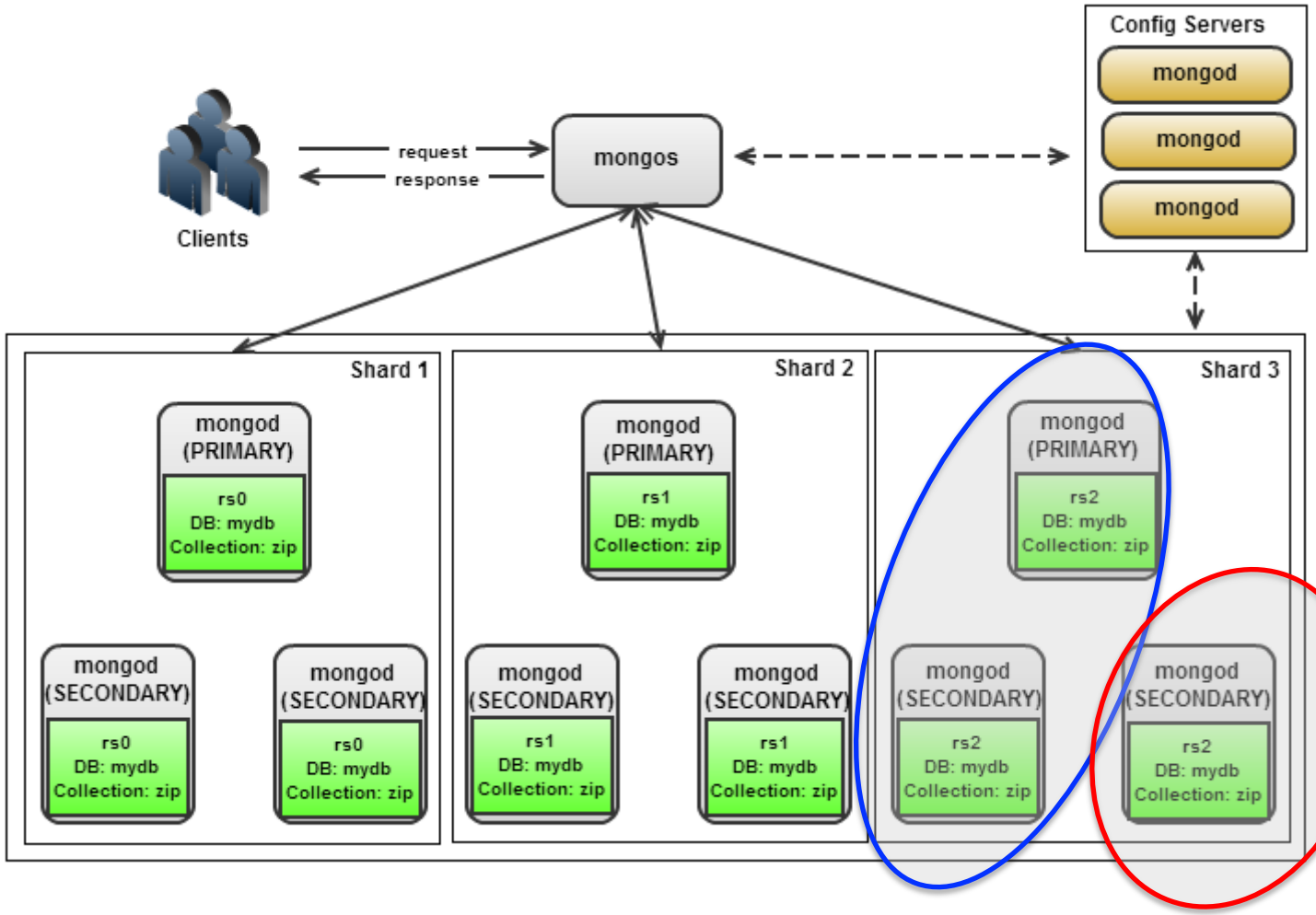
Data is split into three shards based on shard key for scalability

Each shard is replicated across three nodes for high availability

Source: https://severalnines.com/blog/turning-mongodb-replica-set-sharded-cluster

If network is partitioned, behavior depends on primary's location

Replication with sharding



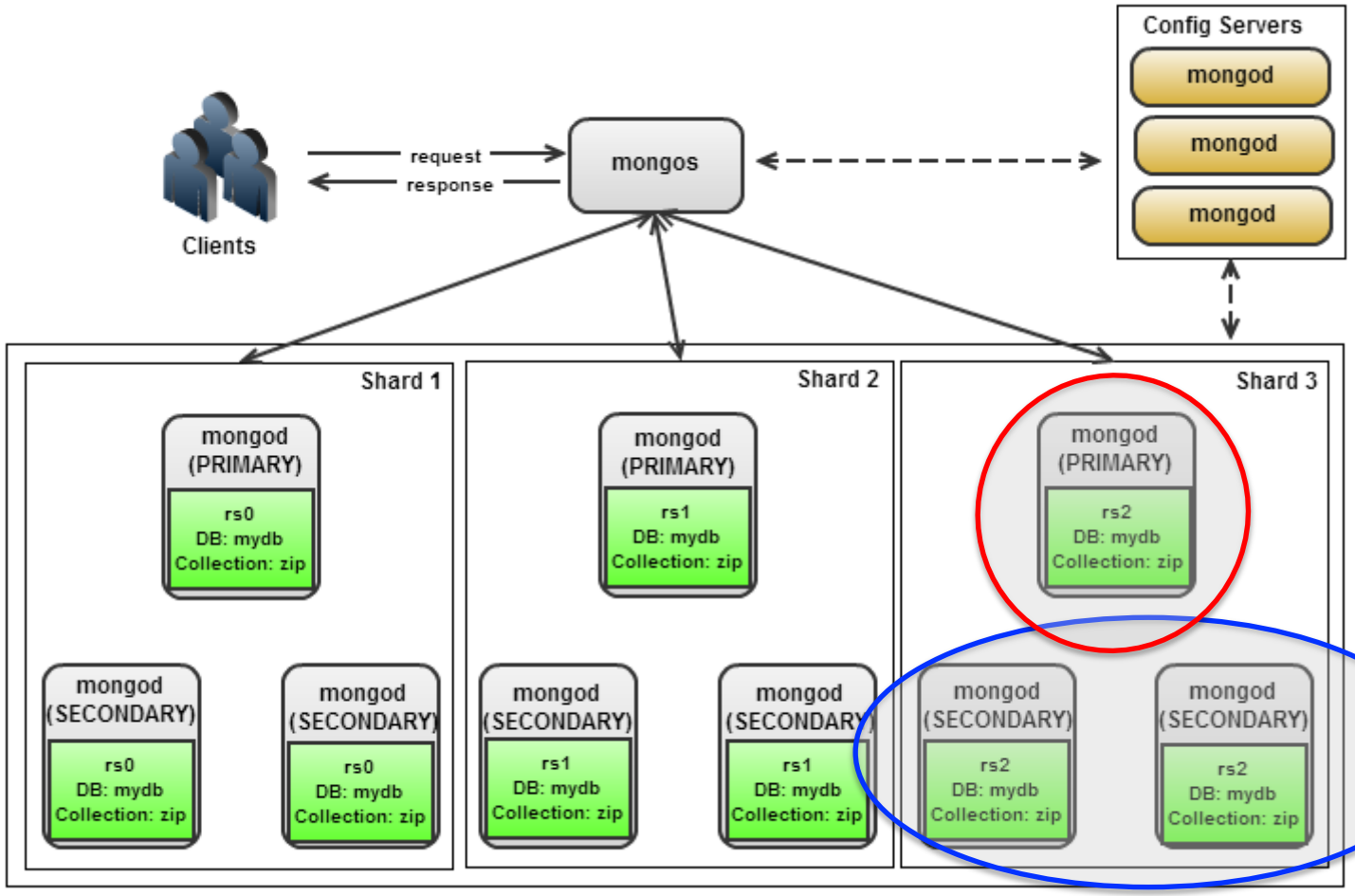
If primary in partition with majority nodes, primary continues

Node in non-majority partition stays as secondary (read only)

Source: https://severalnines.com/blog/turning-mongodb-replica-set-sharded-cluster

If network is partitioned, behavior depends on primary's location

Replication with sharding



If primary in partition in non-majority partition, steps down as primary

Election in majority partition to choose new primary

Source: https://severalnines.com/blog/turning-mongodb-replica-set-sharded-cluster

Homework: get access to a MongoDB database

Create a cloud-based MongoDB database

1. Create a free cloud-based account at Mongo Atlas:
   https://www.mongodb.com/cloud/atlas
2. Install mongo shell to interact with your database

   Mac (assumes you have brew installed):
   brew tap mongodb/brew
   brew install mongodb-community-shell

   Windows:
   https://www.mongodb.com/download-center/community (install only shell)
3. Optional: install Compass (like MySQL Workbench):
   https://www.mongodb.com/products/compass

OR

Install MongoDB locally on your machine

https://docs.mongodb.com/manual/installation/ (includes shell)