CS4225/CS5425 Big Data Systems for Data Science

NoSQL and Basics of Distributed Databases

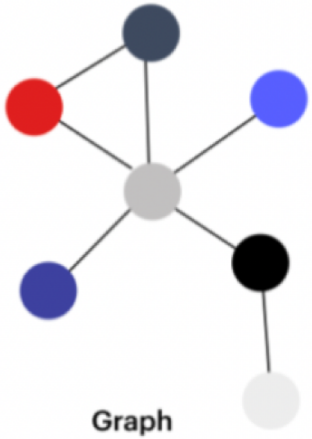
Bryan Hooi
School of Computing
National University of Singapore
bhooi@comp.nus.edu.sg

School of Computing

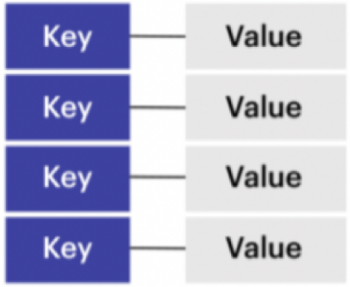
Recap: NoSQL

- NoSQL refers to non-relational databases
- NoSQL has come to stand for "Not Only SQL", i.e. using relational and non-relational databases alongside one another

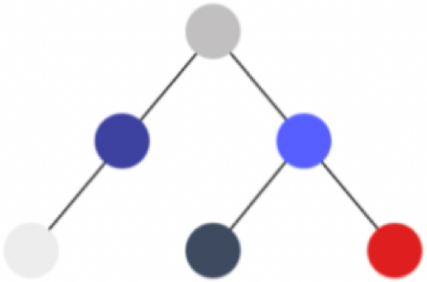
Column



Graph



Key-Value



Document

Recap: Key-Value Stores

Phone directory

Key	Value
Paul	(091) 9786453778
Greg	(091) 9686154559
Marco	(091) 9868564334

MAC table

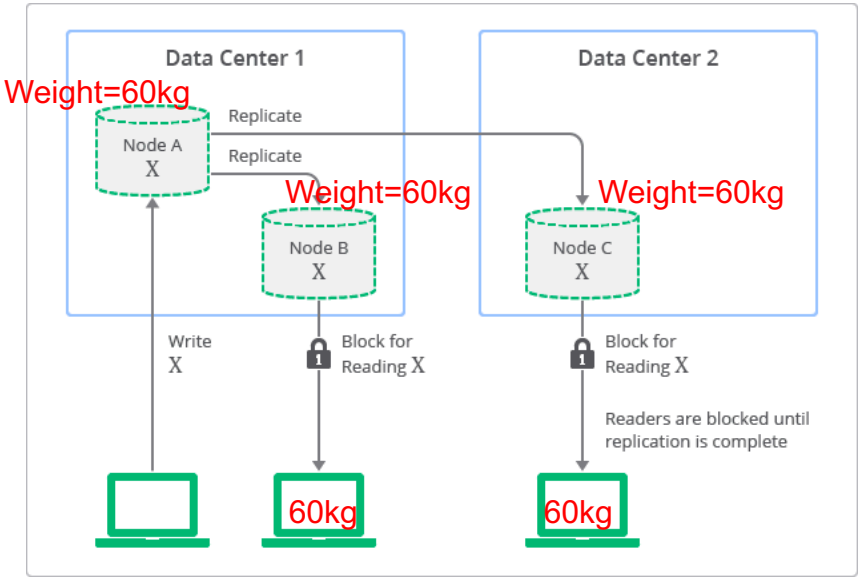
Key	Value
10.94.214.172	3c:22:fb:86:c1:b1
10.94.214.173	00:0a:95:9d:68:16
10.94.214.174	3c:1b:fb:45:c4:b1

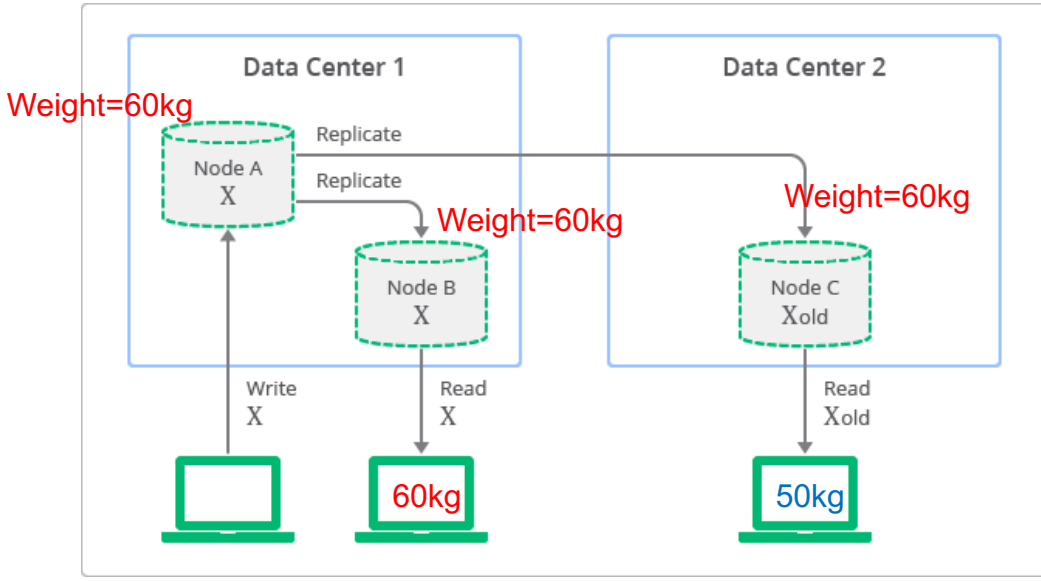
- Operations: get, put, multi-get, multi-put, range queries
- Types: persistent, non-persistent (in-memory)
- Very fast reads and writes – suitable for cases like caches / user profile info, where we usually need to read / write all the data from an object at a time

Recap: Document Stores

- **Collections** have multiple **documents** (i.e. JSON object)
- Operations: CRUD (create, update, read, destroy) – similar to SQL queries, but without fixed schema
- Suitable for queries involving the content of each document

Recap: Strong vs Eventual Consistency





Strong consistency: any reads immediately after an update must give the same result on all observers

Eventual consistency: if the system is functioning and we wait long enough, eventually all reads will return the last written value

Clarification: Strong Consistency vs ACID

- If you've taken a database class, you may recall that the C in ACID stands for "Consistency"
  - This is almost completely unrelated with "Strong Consistency"!
    - "Consistency" in ACID: means the database must always obey certain predefined constraints (e.g. inventory counts cannot be negative)
    - Strong Consistency (in distributed systems): any reads immediately after an update give the same result on all observers
  - The names came from different communities (ACID from the database community; Strong Consistency from distributed systems)
  - Note: since we didn't discuss ACID much in this class, we consider it as not required knowledge for exam purposes

Recap: Duplication

Users

```
[
  {
    "user_id": 100,
    "name": "Alice"
  },
  {
    "user_id": 101,
    "name": "Bob"
  },
]
```

Posts

```
[
  {
    "post_id": "1",
    "post": "Hello world",
    "user_id": 100
  },
  {
    "post_id": "2",
    "post": "Bye world",
    "user_id": 101
  },
]
```


Posts (+ username)

```
[
  {
    "post_id": "1",
    "post": "Hello world",
    "user_id": 100,
    "name": "Alice",
  },
  {
    "post_id": "2",
    "post": "Bye world",
    "user_id": 101,
    "name: "Bob"
  },
]
```

- 'Storage is cheap: why not just duplicate data to improve efficiency?'
- Tables are designed around the queries we expect to receive
- Leads to a new problem: what if user changes their name? (this needs to be propagated to multiple tables)

Recap: Pros & Cons of NoSQL Systems

Pros

+ **Flexible / dynamic schema**: suitable for less well-structured data

+ **Horizontal scalability**: we will discuss this more next week

+ **High performance and availability**: due to their relaxed consistency model and fast reads / writes

Cons

- **No declarative query language**: query logic (e.g. joins) may have to be handled on the application side, which can add additional programming

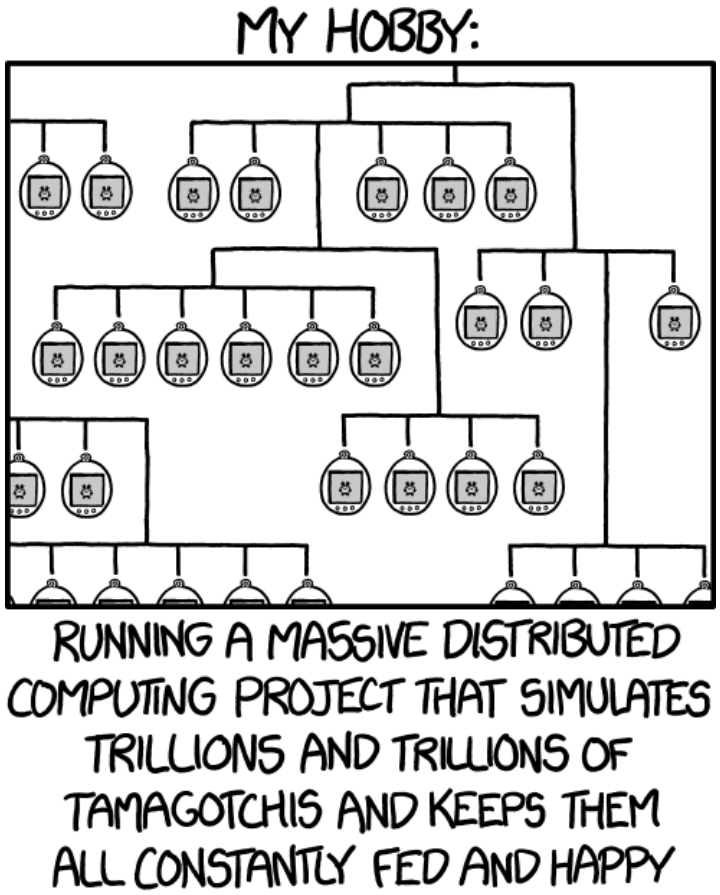
- **Weaker consistency guarantees**: application may receive stale data that may need to be handled on the application side

Conclusion: no one size fits all. Depends on needs of application: whether denormalization is suitable; complexity of queries (joins vs simple read/writes); importance of consistency (e.g. financial transactions vs tweets); data volume / need for availability.

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

Today's Plan

- **Basic Concepts of Distributed Databases**
- Data Partitioning
- Query Processing in NoSQL



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

Introduction and Motivation

- In today's lecture, we focus on how NoSQL systems are implemented as distributed databases.
- Many of today's concepts will be relevant to distributed databases more generally, including both NoSQL, as well as relational databases.
- Why distributed databases?
  - **Scalability**: allow database sizes to scale simply by adding more nodes
  - **Availability / Fault Tolerance:** if one node fails, others can still server requests
  - **Latency**: generally, each request is served by the closest replica, reducing latency, particularly when the database is distributed over a wide geographical area (e.g., globally)

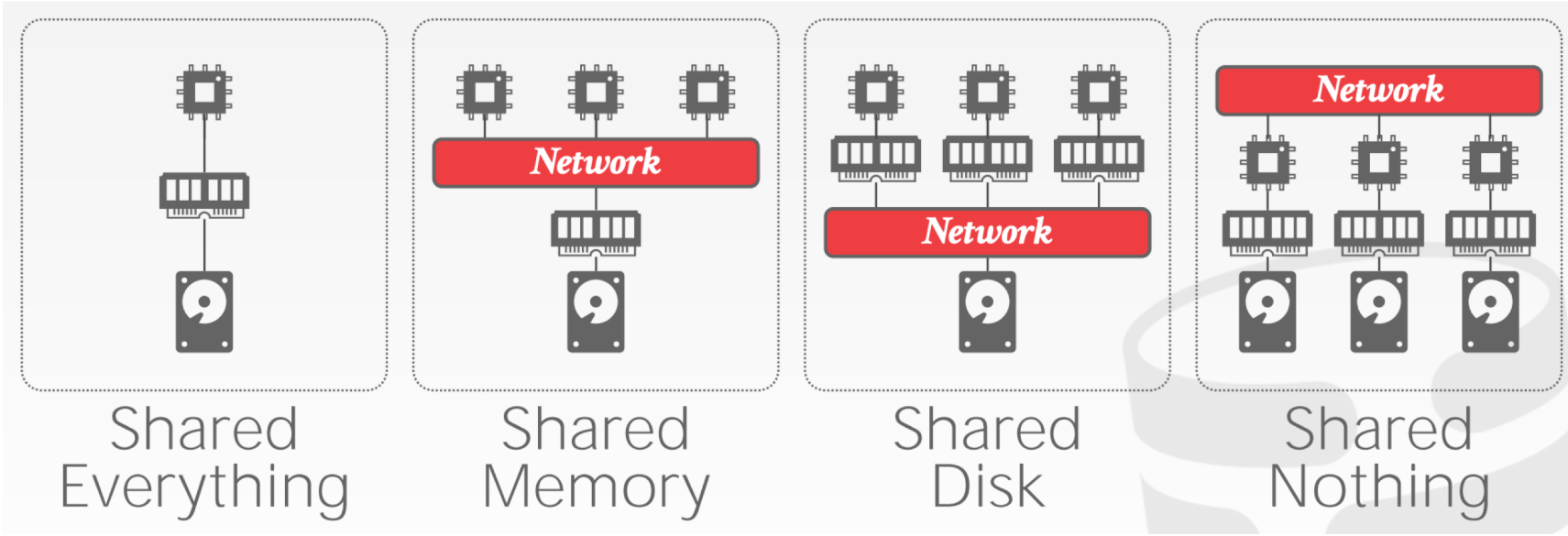
Data Transparency

- Users should not be required to know how the data is physically distributed, partitioned, or replicated.
- A query that works on a single node database should still work on a distributed database.

Assumption of Distributed Databases

- All nodes in a distributed database are well-behaved (i.e. they follow the protocol we designed for them; not 'adversarial' or trying to corrupt the database)
  - Out of syllabus: If we cannot trust the other nodes, we need a 'Byzantine Fault Tolerant' protocol, which allows the system to function correctly even when some of its nodes behave arbitrarily (including maliciously). Blockchains achieve this goal (using a decentralized ledger, and various consensus mechanisms)

Distributed Database Architectures



Examples:

Single node DBMS

Supercomputers

Cloud databases

NoSQL (mostly)

ORACLE RAC

snowflake

Cockroach LABS

H-Store

redis

cassandra

mongoDB

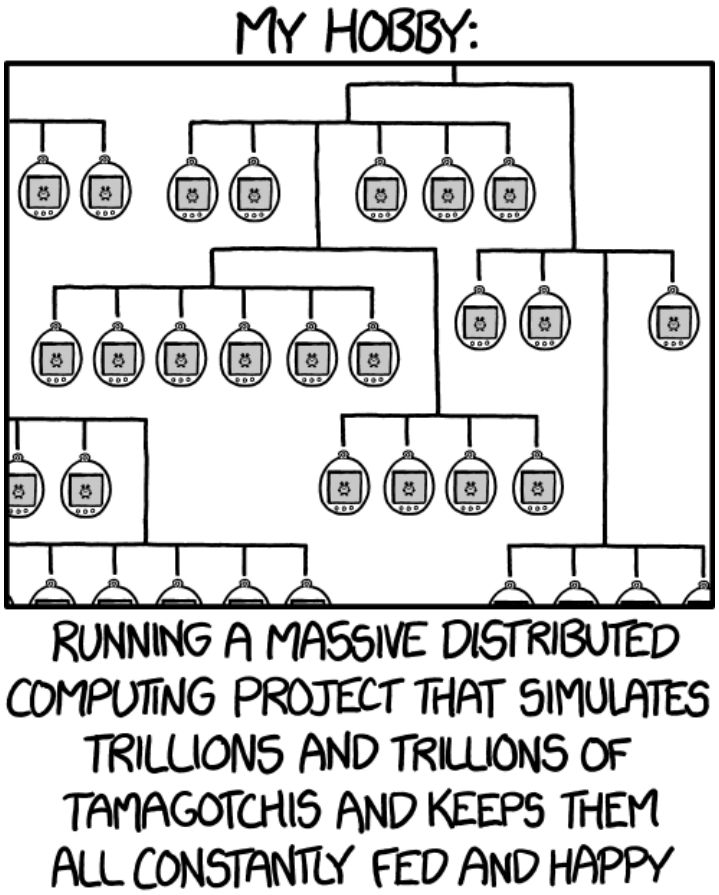
CouchDB

MySQL Cluster

Credit: Andy Pavlo (CMU CS15-446)

Today's Plan

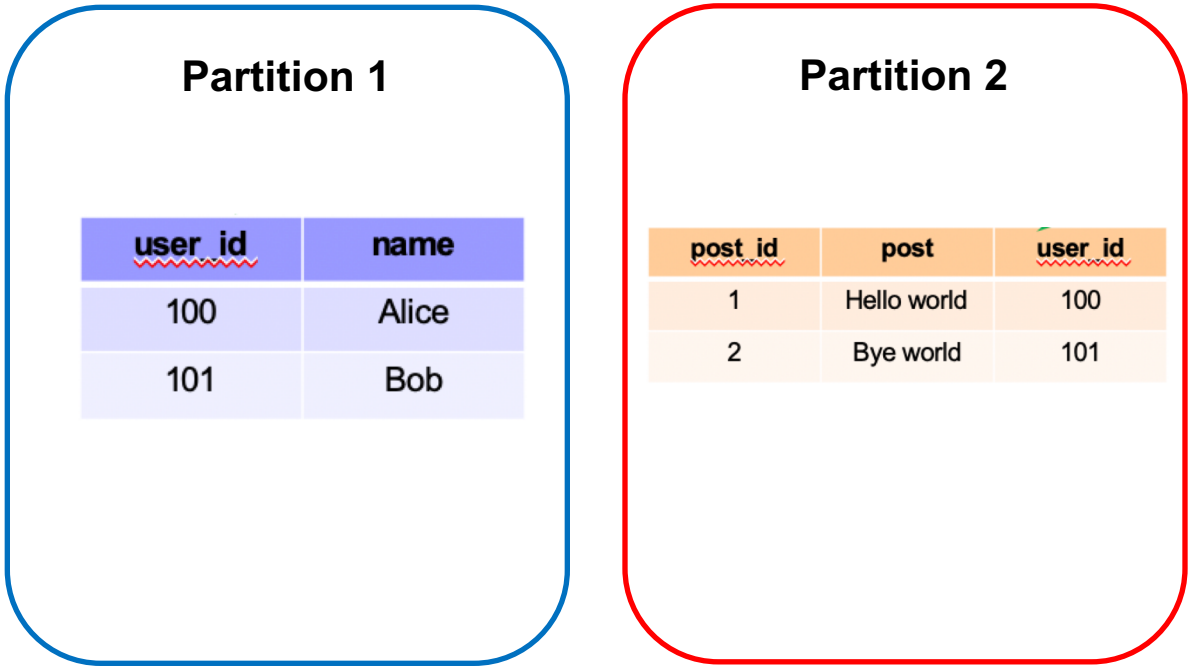
- Basic Concepts of Distributed Databases
- **Data Partitioning**
- Query Processing in NoSQL



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

Table Partitioning

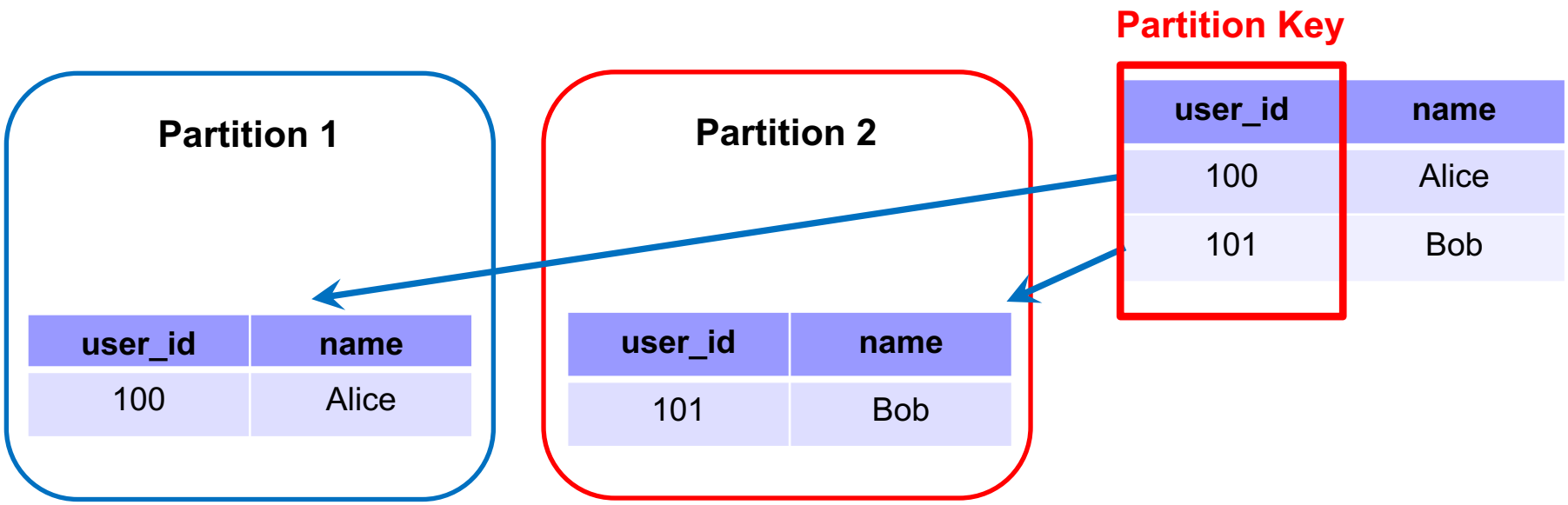
- Put different tables (or collections) on different machines



- **Problem**: scalability – each table cannot be split across multiple machines

Horizontal Partitioning

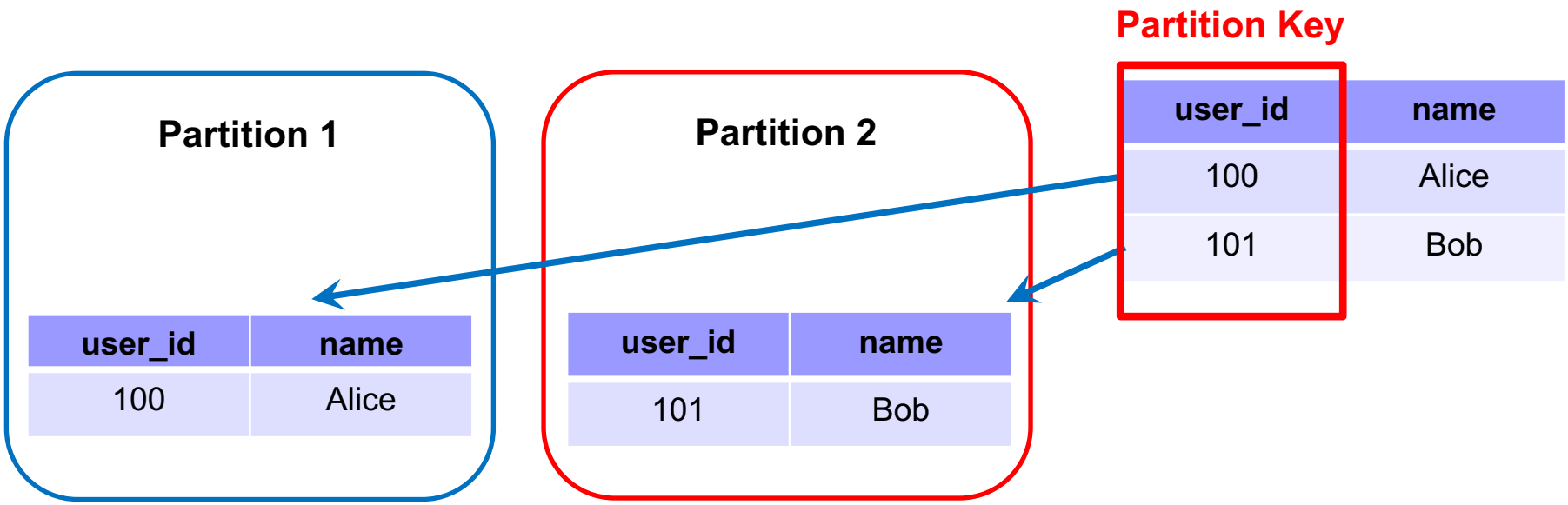
- Different tuples are stored in different nodes



- Also called "sharding"
- **Partition Key** (or "shard key") is the variable used to decide which node each tuple will be stored on: tuples with the same shard key will be on the same node
  - **How to choose partition key?** If we often need to filter tuples based on a column, or "group by" a column, then that column is a suitable partition key
  - Example: if we filter tuples by user_id=100, and user_id is the partition key, then all the user_id=100 data will be on the same partition. Data from other partitions can be ignored (called '**partition pruning**'), which saves time as we don't have to scan these tuples.

Horizontal Partitioning

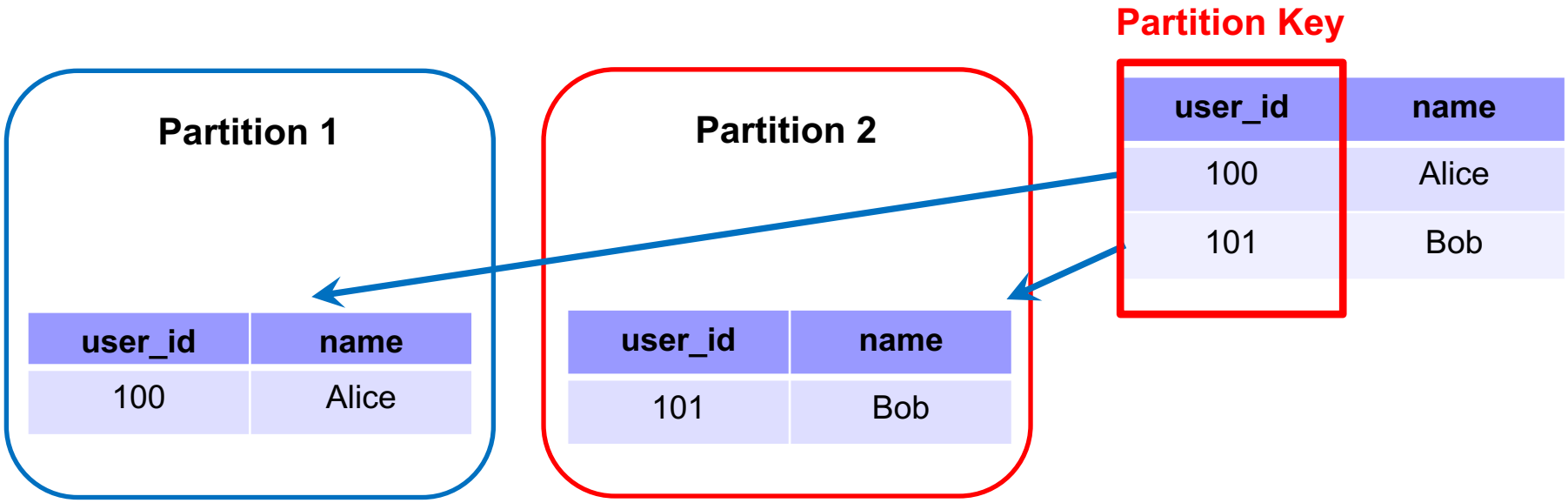
- Different tuples are stored in different nodes



- **Q**: imagine using each user's city, city_id as a partition key; when is this good / bad?

Horizontal Partitioning

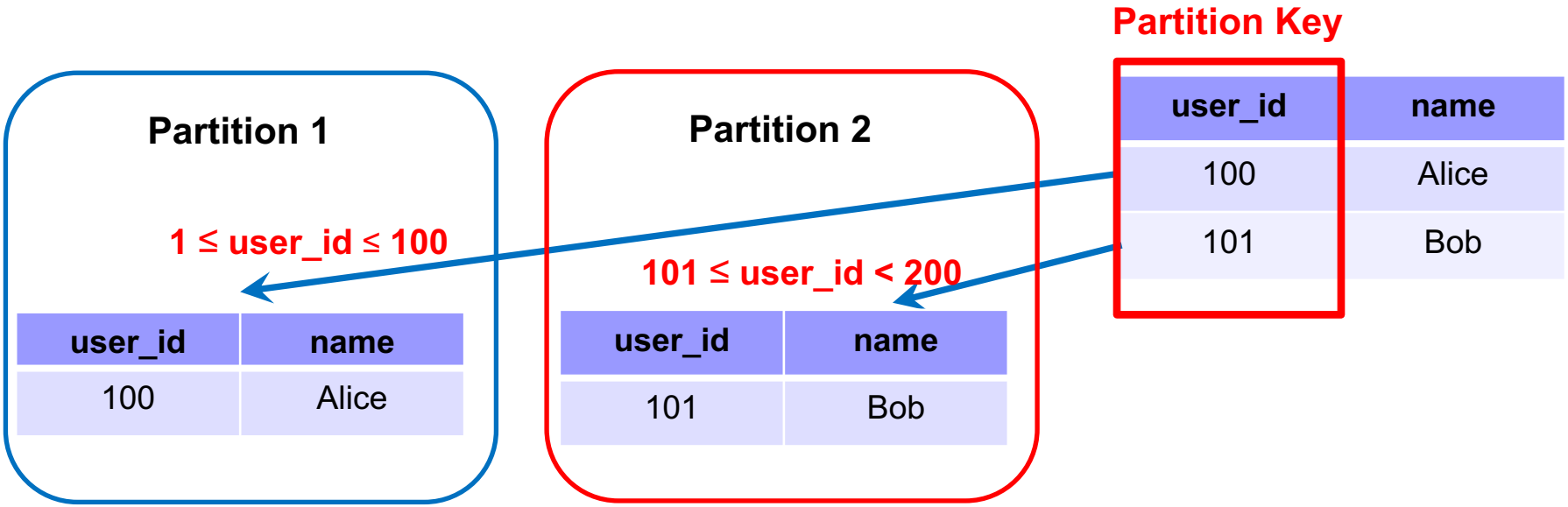
- Different tuples are stored in different nodes



- **Q**: imagine using each user's city, city_id as a partition key; when is this good / bad?
- **A**: good if we mostly aggregate data only within individual cities. Bad if there are too few cities (called **low cardinality**), and this causes a lack of scalability. A similar problem occurs if some cities have too **high frequency** (e.g. most users are from Singapore). Sometimes, these can be mitigated using **composite keys** (a combination of multiple keys)

Horizontal Partitioning – Range Partition

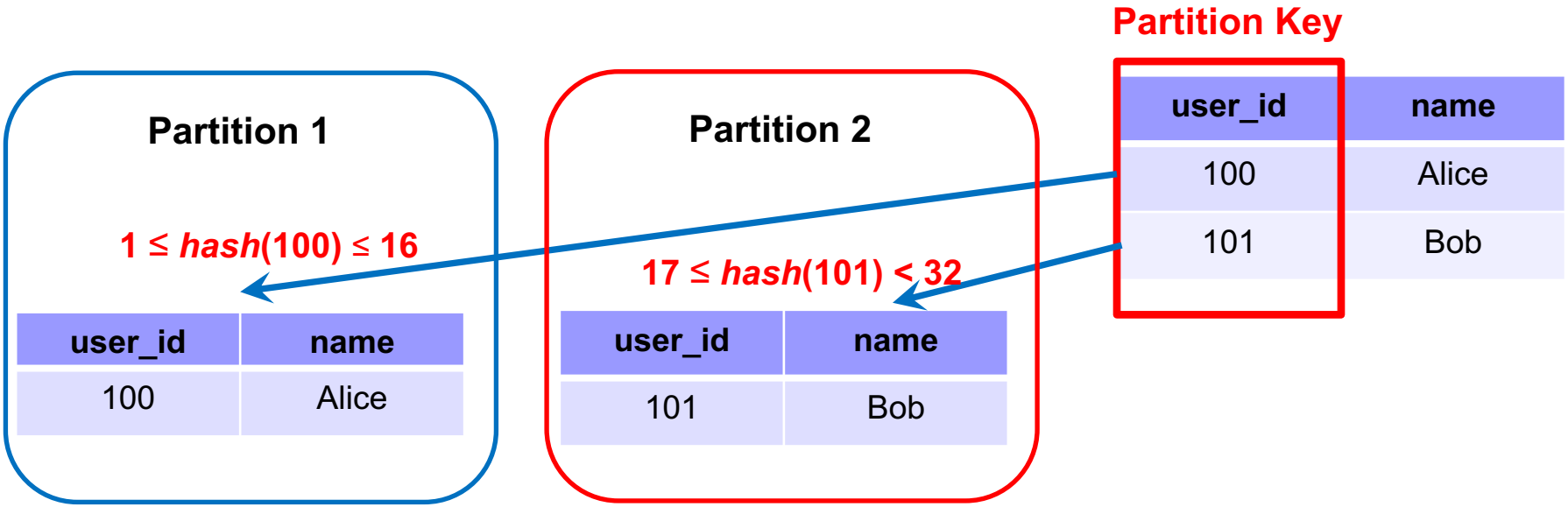
- Different tuples are stored in different nodes



- **Range Partition**: split partition key based on range of values
  - Beneficial if we need range-based queries. In the above example, if the user queries for user_id < 50, all the data in partition 2 can be ignored ('partition pruning'); this saves a lot of work
  - But: range partitioning can lead to imbalanced shards, e.g. if many rows have user_id = 0
  - Splitting the range is automatically handled by a balancer (it tries to keep the shards balanced)

Horizontal Partitioning – Hash Partition

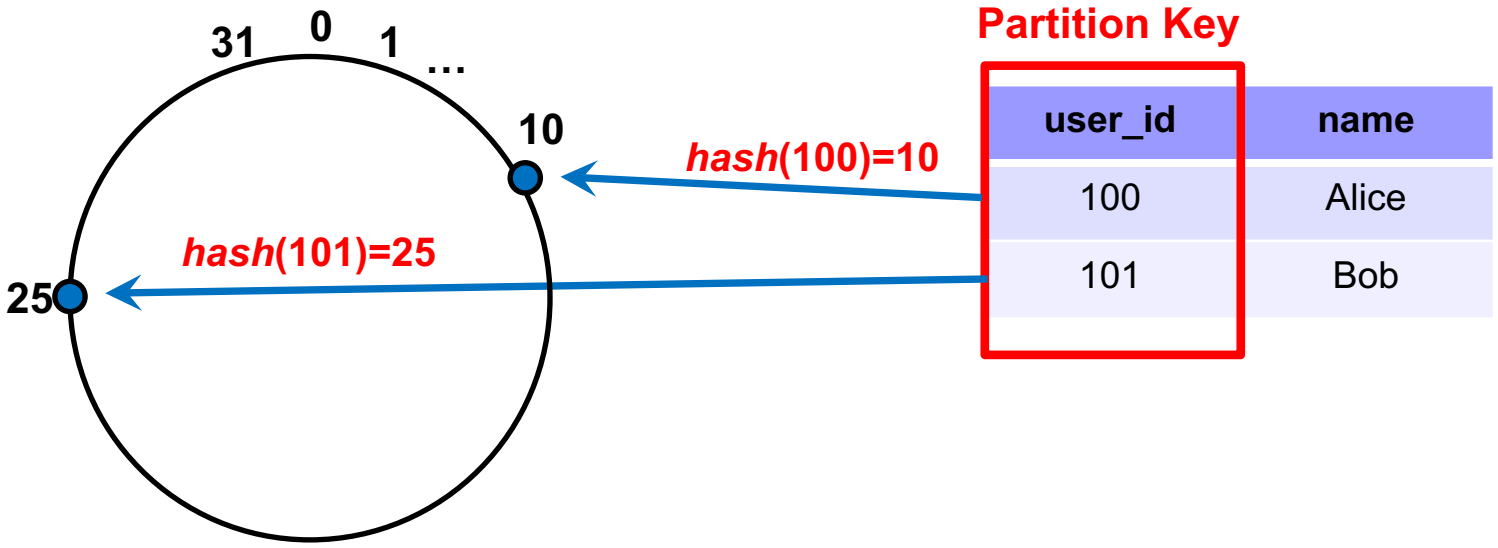
- Different tuples are stored in different nodes



- **Hash Partition**: hash partition key, then divide that into partitions based on ranges
  - Hash function automatically spreads out partition key values roughly evenly
  - **Question:** in previous approaches, how to add / remove a node? If we completely redo the partition, a lot of data may have to be moved around, which is inefficient.
  - **Answer**: consistent hashing

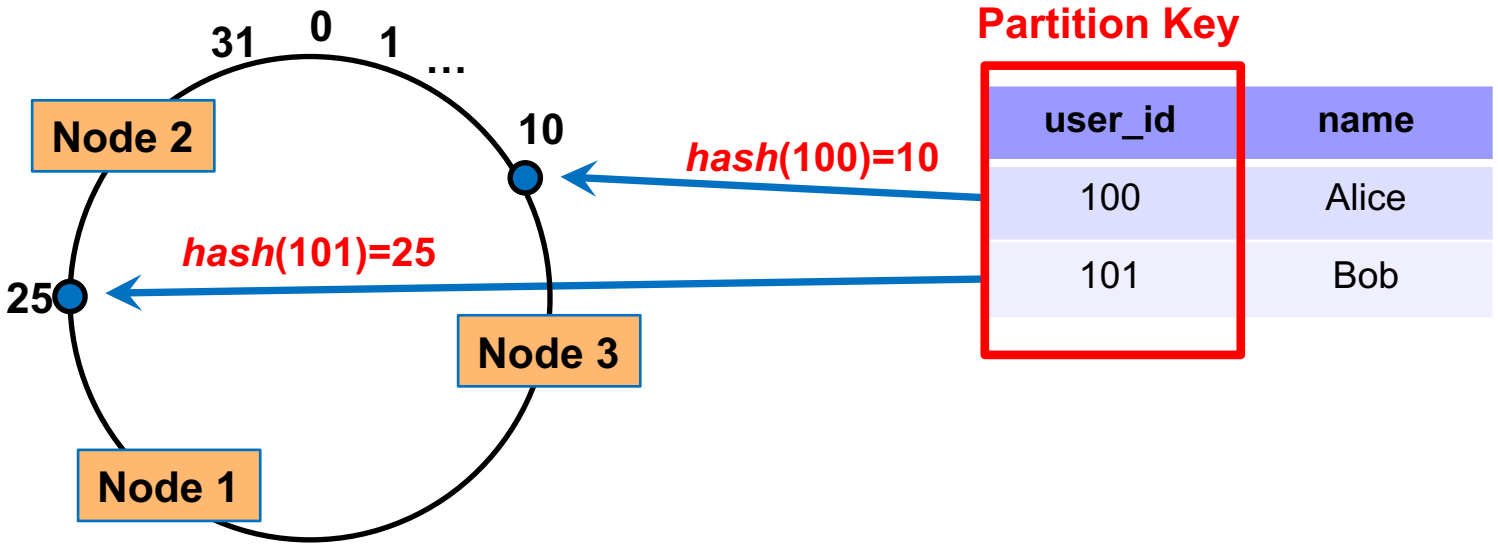
Consistent Hashing

- Think of the output of the hash function as lying on a circle:



Consistent Hashing

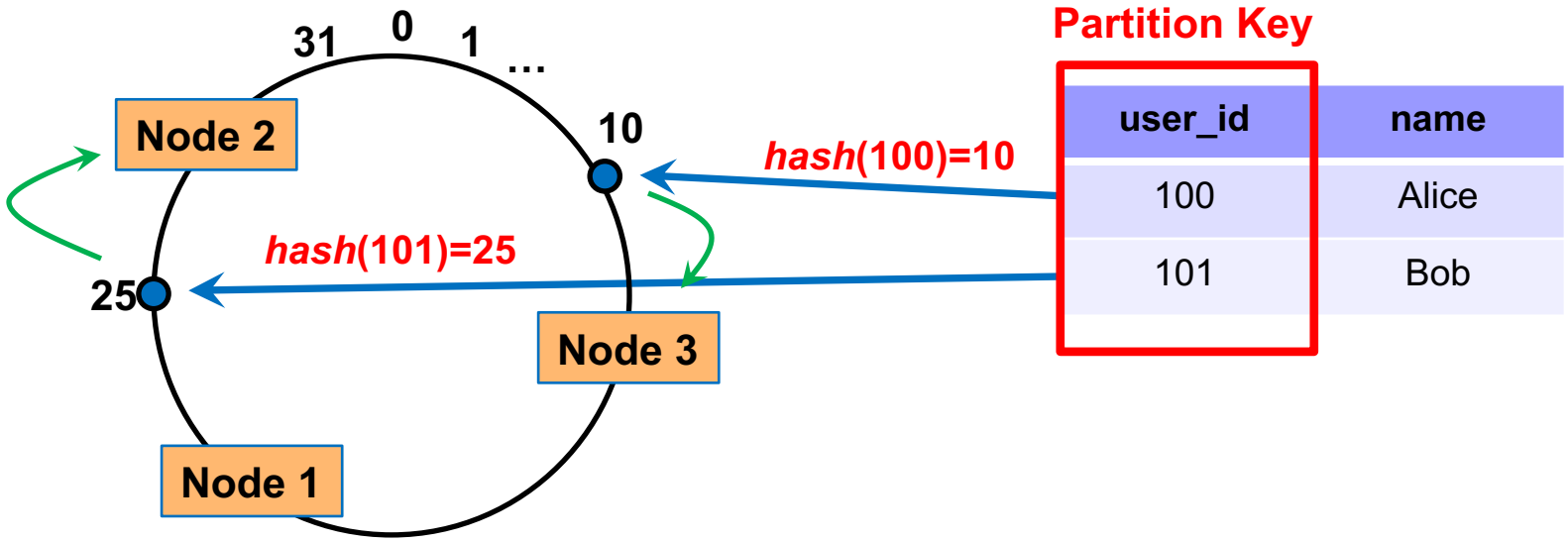
- Think of the output of the hash function as lying on a circle:



- **How to partition**: each node has a 'marker' (rectangles)

Consistent Hashing

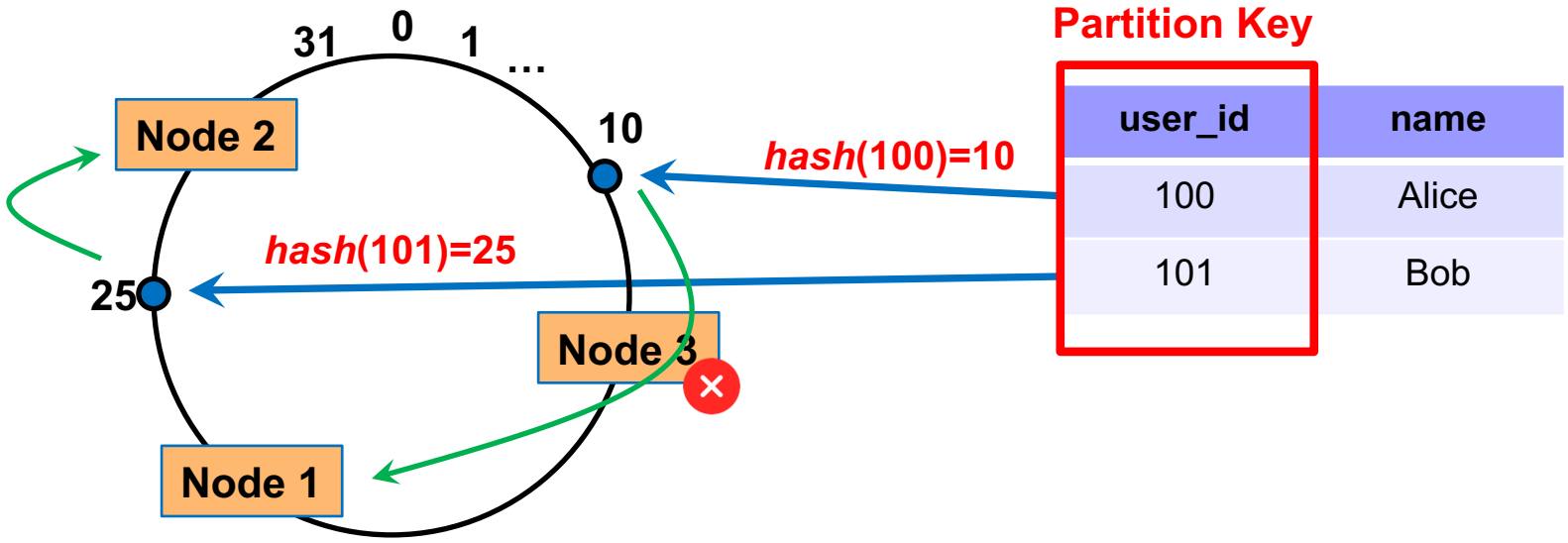
- Think of the output of the hash function as lying on a circle:



- **How to partition**: each node has a 'marker' (rectangles)
  - Each tuple is placed on the circle, and assigned to the node that comes clockwise-after it

Consistent Hashing

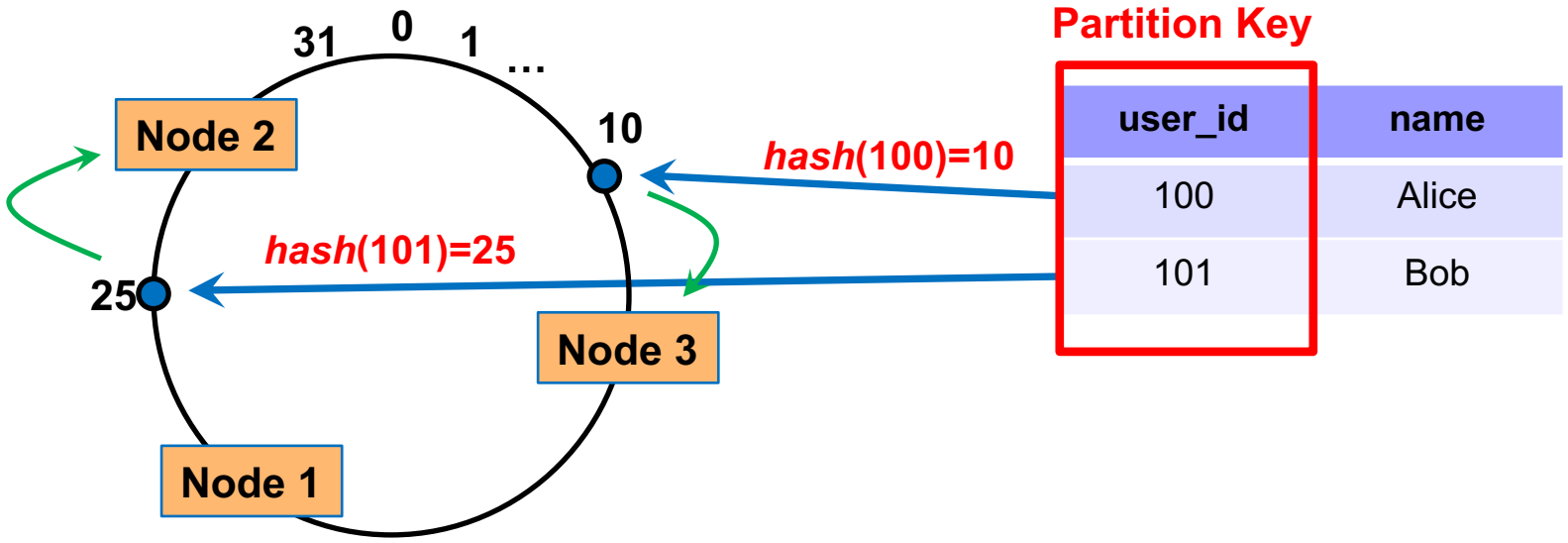
- Think of the output of the hash function as lying on a circle:



- To **delete a node**, we simply re-assign all its tuples to the node clock-wise after this one

Consistent Hashing

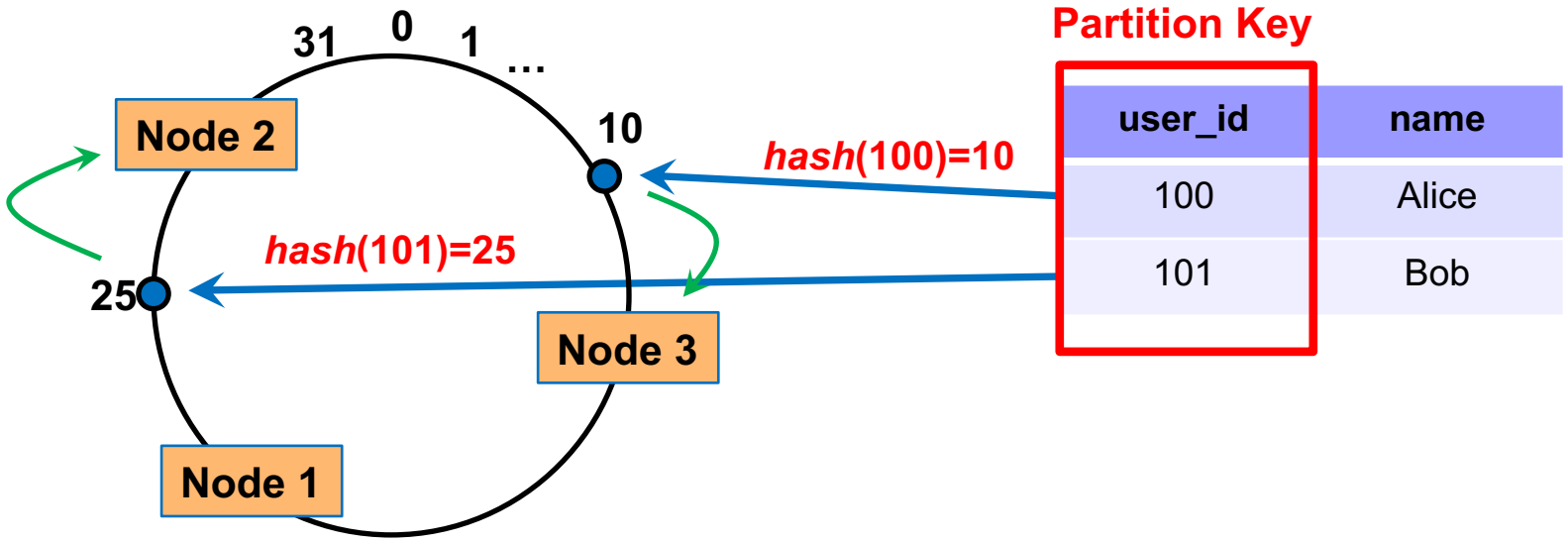
- Think of the output of the hash function as lying on a circle:



- Similarly, to **add a node**, we add a new marker, and re-assigning all tuples which now belong to the new node

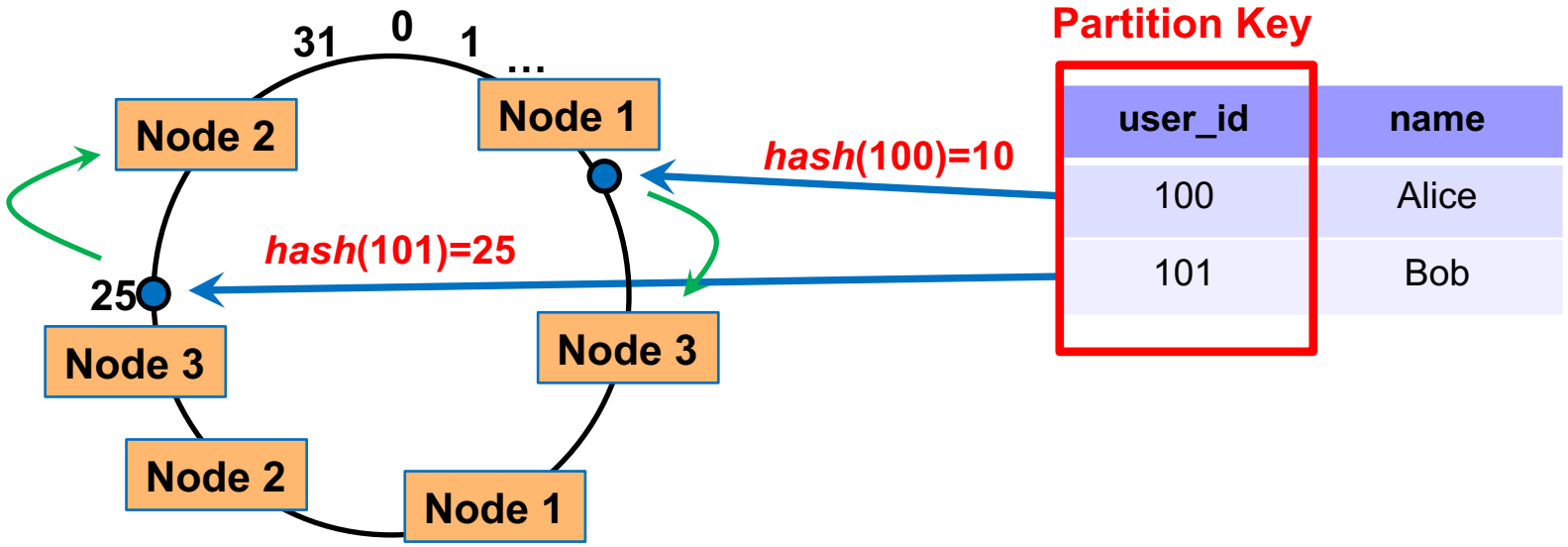
Consistent Hashing

- Think of the output of the hash function as lying on a circle:



- **Simple replication strategy**: replicate a tuple in the next few (e.g. 2) additional nodes clockwise after the primary node used to store it

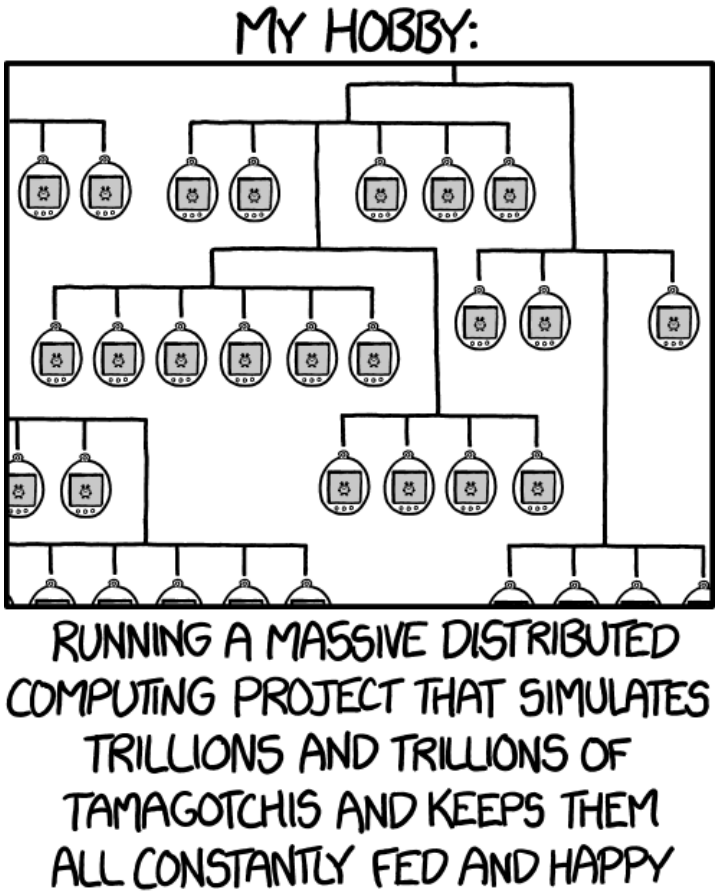
Consistent Hashing



- **Multiple markers**: we can also have multiple markers per node. For each tuple, we still assign it to the marker nearest to it in the clockwise direction.
- Benefit: when we remove a node (e.g. node 1), its tuples will not all be reassigned to the same node. So, this can balance load better.

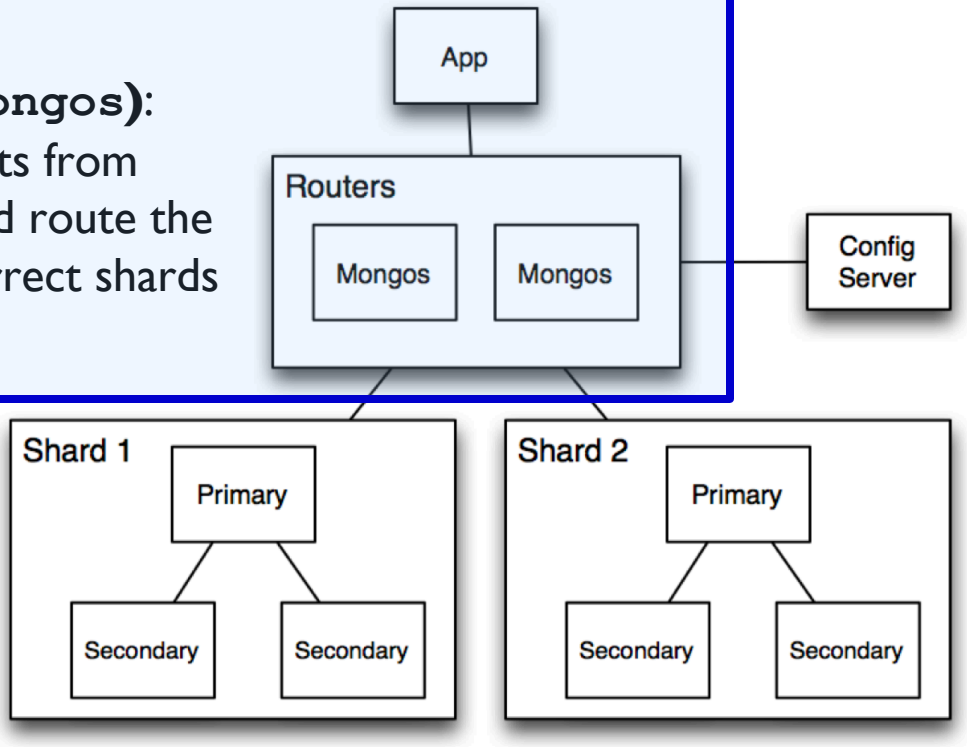
Today's Plan

- Basic Concepts of Distributed Databases
- Data Partitioning
- **Query Processing in NoSQL**



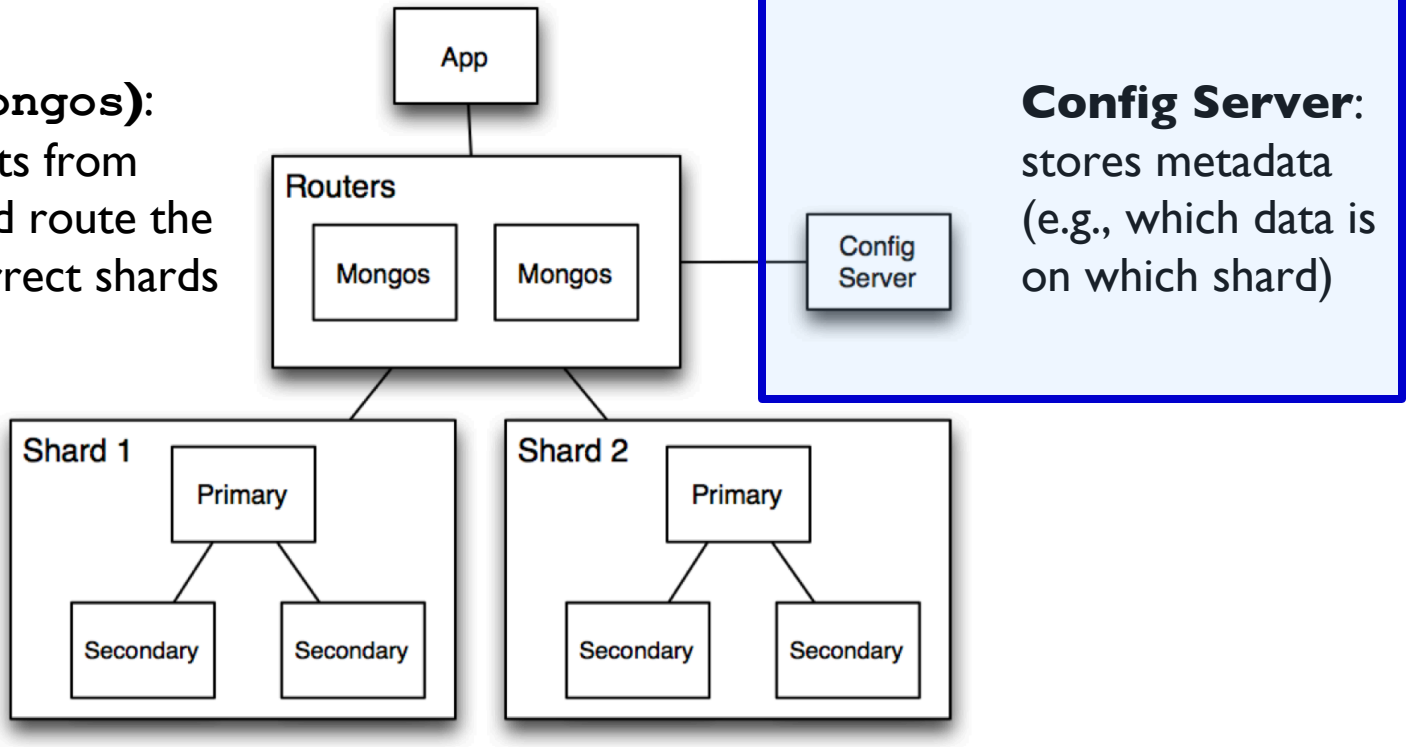
Architecture of MongoDB

Routers (`mongos`): handle requests from application and route the queries to correct shards



Architecture of MongoDB

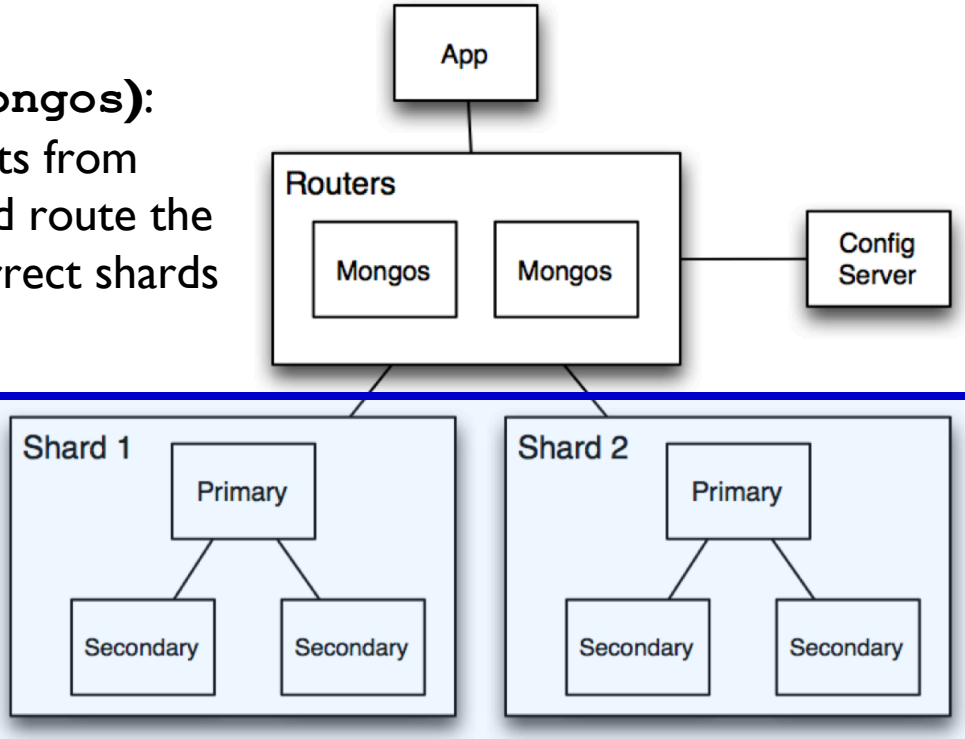
Routers (`mongos`): handle requests from application and route the queries to correct shards



Config Server: stores metadata (e.g., which data is on which shard)

Architecture of MongoDB

Routers (`mongos`): handle requests from application and route the queries to correct shards

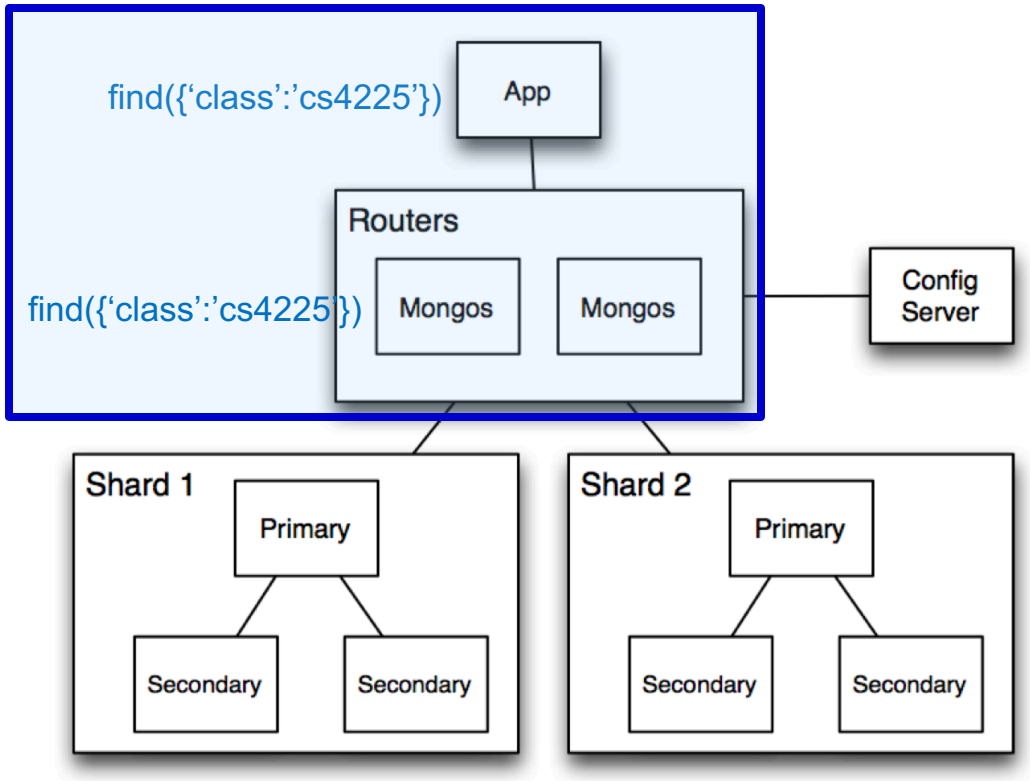


Config Server: stores metadata (e.g., which data is on which shard)

Replica sets: multiple processes which maintain the same data, for redundancy

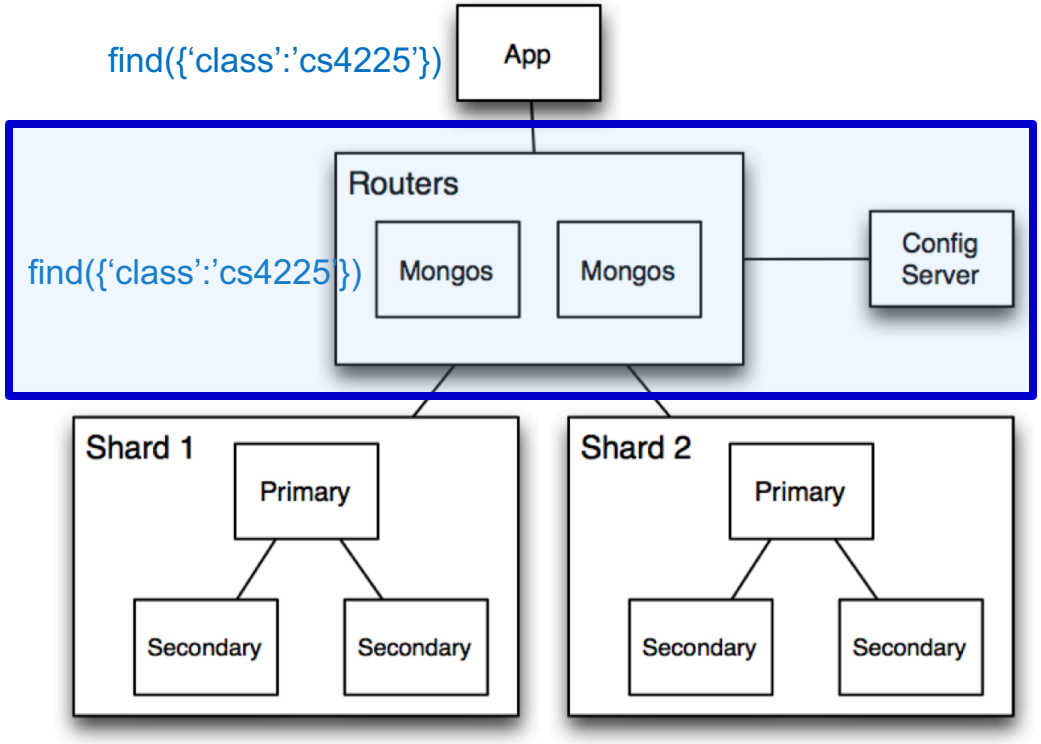
Each replica set is responsible for a subset of the database, known as a **shard** (this is a **horizontal partitioning** scheme)

Example of Read or Write Query



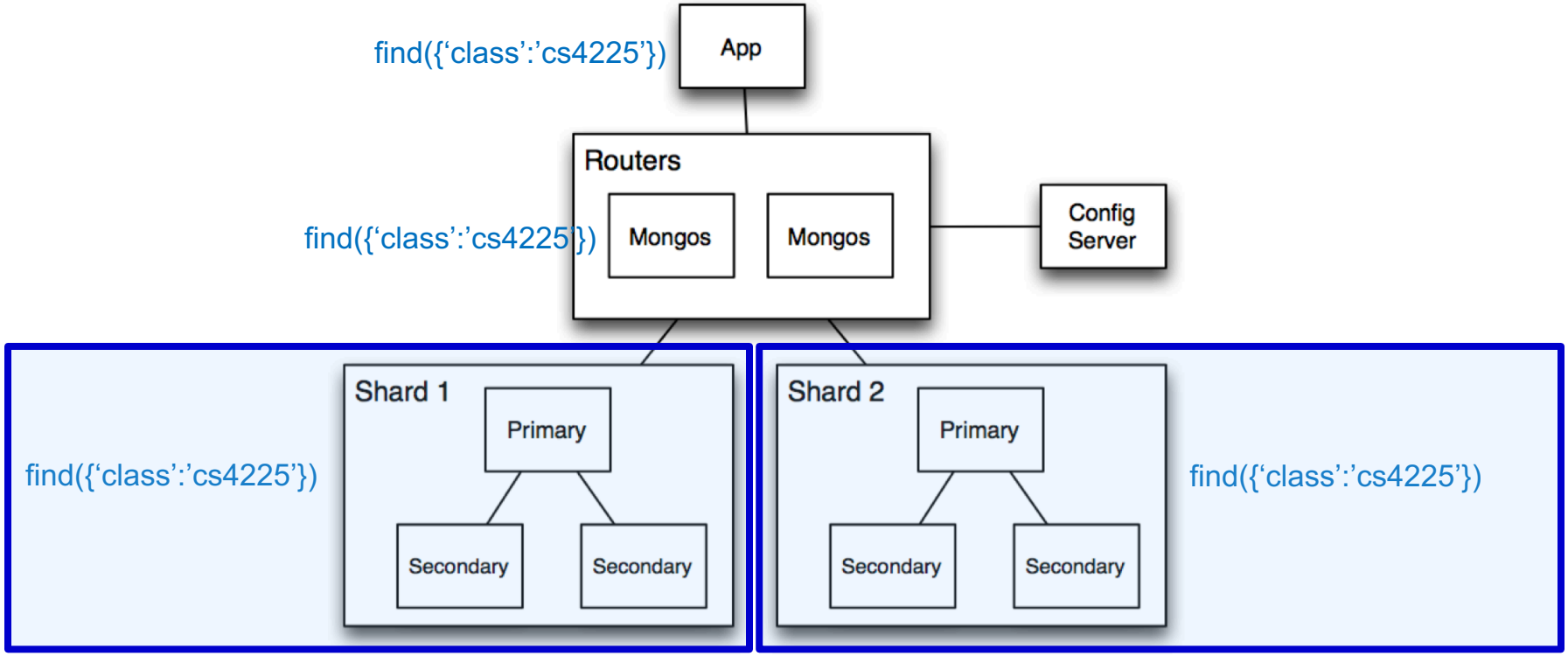
1. **Query is issued to a router (`mongos`) instance**

Example of Read or Write Query



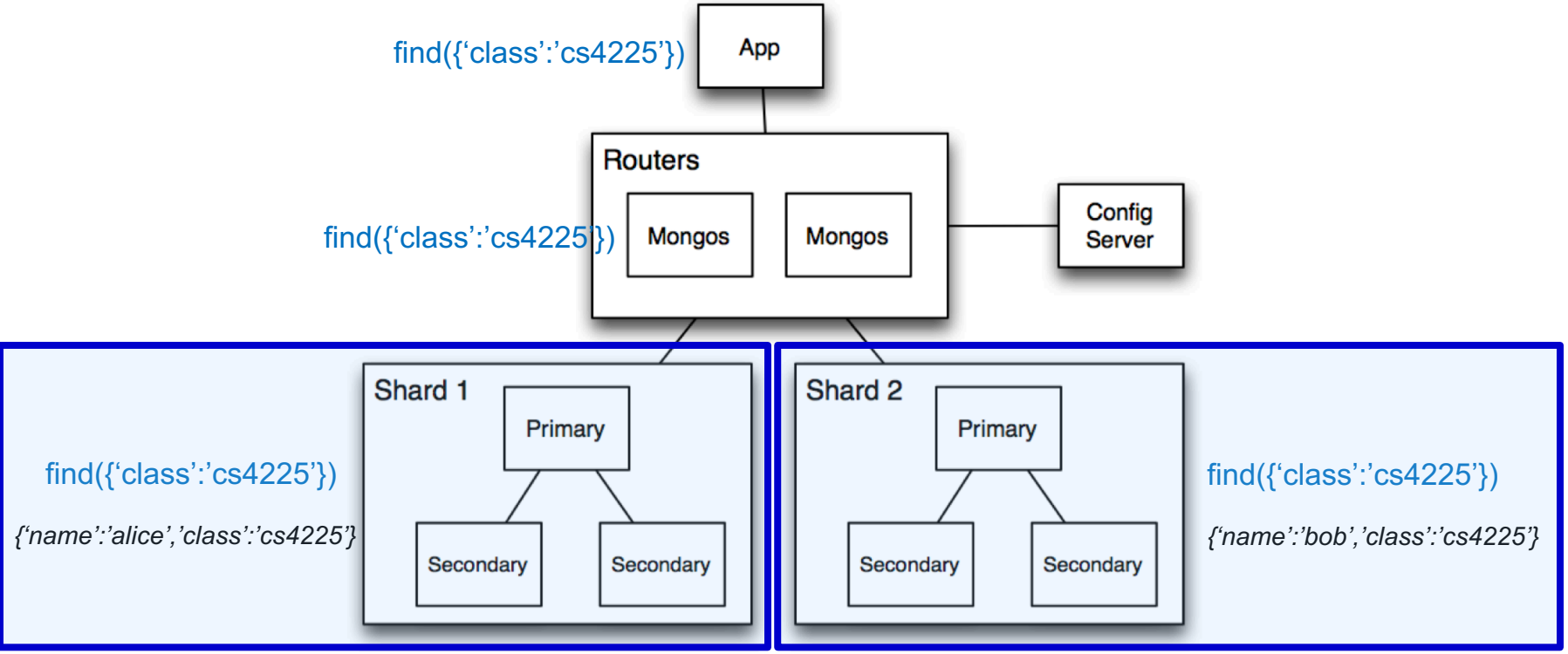
1. Query is issued to a router (`mongos`) instance
2. **With help of config server, `mongos` determines which shards to query**

Example of Read or Write Query



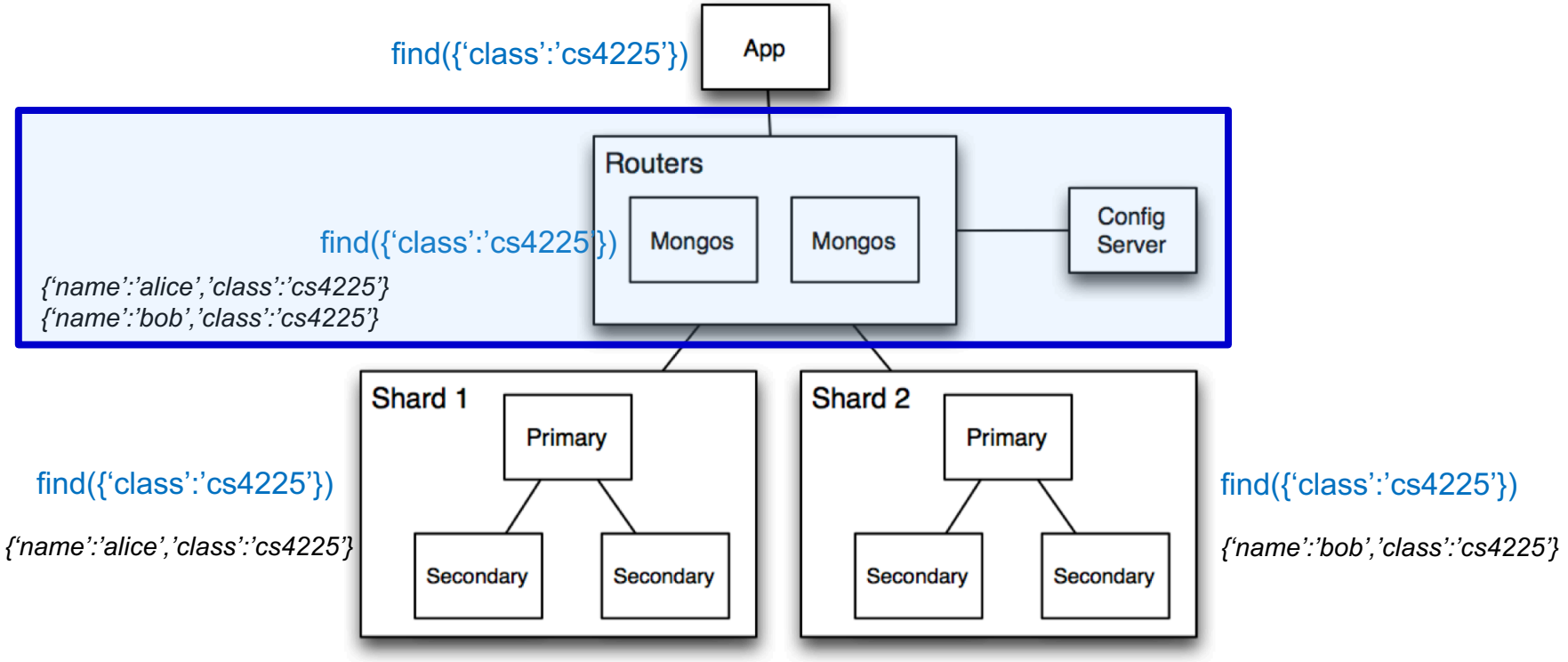
1. Query is issued to a router (`mongos`) instance
2. With help of config server, `mongos` determines which shards to query
3. **Query is sent to relevant shards (*partition pruning*)**
   - Example: when reading a specific value of the shard key, the config server can determine that the query only needs to go to one shard (the one that contains that value of the shard key); writes are similar
   - But if the query is based on a key other than the shard key, it is relevant to all shards, and thus will go to all shards

Example of Read or Write Query



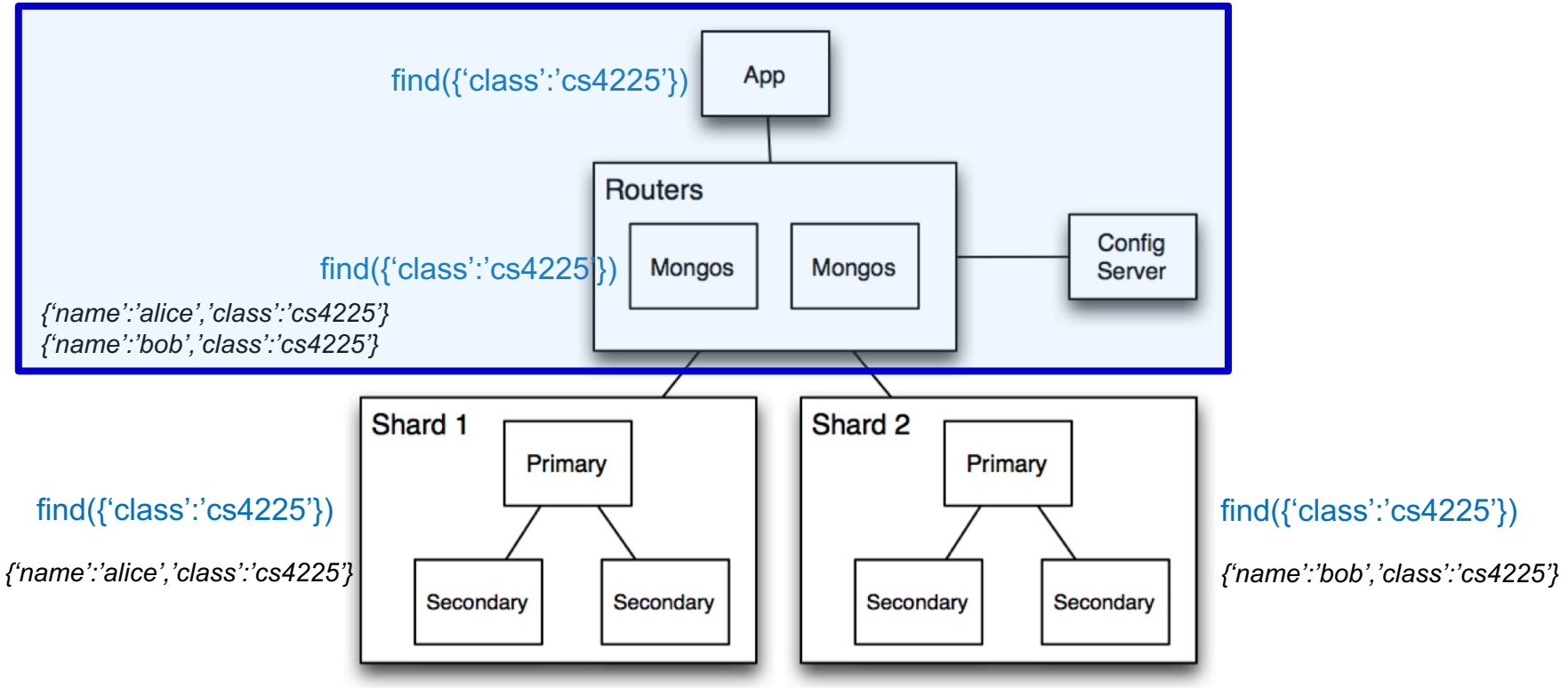
1. Query is issued to a router (`mongos`) instance
2. With help of config server, `mongos` determines which shards to query
3. Query is sent to relevant shards (*partition pruning*)
4. **Shards run query on their data, and send results back to `mongos`**

Example of Read or Write Query



1. Query is issued to a router (`mongos`) instance
2. With help of config server, `mongos` determines which shards to query
3. Query is sent to relevant shards (*partition pruning*)
4. Shards run query on their data, and send results back to `mongos`
5. `mongos` merges the query results and returns the merged results to the application

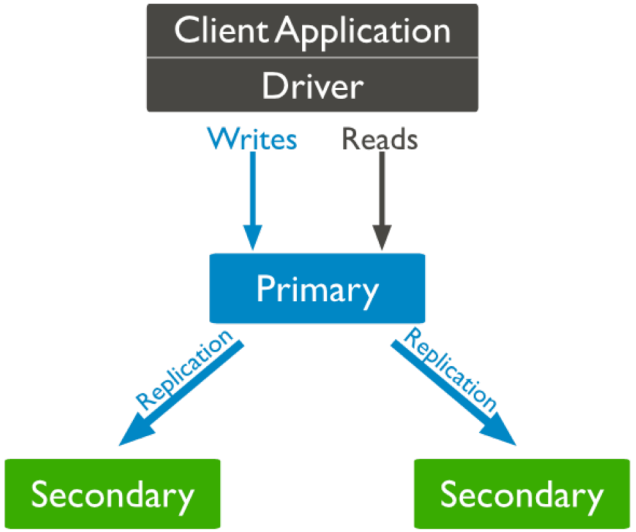
Example of Read or Write Query



1. Query is issued to a router (`mongos`) instance
2. With help of config server, `mongos` determines which shards to query
3. Query is sent to relevant shards (*partition pruning*)
4. Shards run query on their data, and send results back to `mongos`
5. `mongos` merges the query results and returns the merged results to the application

Replication in MongoDB

- Common configuration: 1 primary, 2 secondaries
- **Writes**:
  - The "Primary" receives all write operations
  - Records writes onto its "operation log"
  - Secondaries will then replicate this operation log, apply it to their local copies of the data (thus ensuring data is synchronized), then acknowledge the operation
- **Reads**:
  - The user can configure the "read preference", which decides whether we can read from secondaries (this is the default), or the primary
  - Allowing reading from secondaries can decrease latency and distribute load (improving throughput), but allows for reading stale data (it has only "eventual consistency")
- **Elections**:
  - If the primary node fails, the nodes "conduct an election", which is a protocol to choose one of the secondaries to be promoted to primary



Conclusion: Reasons for Scalability & Performance of NoSQL

- **Horizontal partitioning**: as we get more and more data, we can simply partition it into more and more shards (even if individual tables become very large)
  - Horizontal partitioning improves speed due to parallelization.
- **Duplication (i.e. denormalization):** Unlike relational DBs where queries may require looking up multiple tables (joins), using duplication in NoSQL allows queries to go to only 1 collection.
- **Relaxed consistency guarantees**: prioritize availability over consistency – can return slightly stale data

First Half Wrap-up + Exam Info

- Questions generally focus on understanding and application:
  - Integrative: Require you to combine knowledge from different chapters of the textbook
  - "Application": Require you to apply your knowledge of fundamental concepts to reasonably practical scenarios.
  - "Why not": Example, Tommy proposed a solution A to solve problem B in the lecture. Tell me what is the problem with solution A and how to overcome this problem
- In general: if you have successfully understood the concepts we have discussed in lecture, you should be able to answer the test questions.
  - General focus is on understanding of concepts / principles and understanding when various algorithms / systems should be used; not on knowing particular details of specific software implementations

Scope of Exam

- **Scope**: content discussed in the lecture which appears in slides
- **Out of scope**:
  - Content only found in the notes underneath the slides (these can be ignored)
  - Content only discussed in response to student questions (i.e. beyond the scope of the slides)
  - Content marked with "Details" or that I mentioned is out of scope
  - If there are any tasks which ask for code, they will allow pseudocode. As long as your pseudocode is reasonable / understandable by the grader, we will accept it.
  - Historical details

Tips for Designing MapReduce Programs

- In exam, only pseudocode will be required (if you want to write in Python / Java, that is fine too). As long as your pseudocode is "reasonable" (understandable to the grader), it will be accepted
- **Think about what key the mapper should emit**: this key will be used to group the data for the reducers
- **Then think about what value the mapper should emit**: this should give the reducers all the information they need to perform their task
- **Evaluating efficiency**: main considerations are the disk and network I/O (determined by the amount of data emitted by mappers, but possibly reduced by combiners), and the memory working set (determined by the mapper's intermediate state)

MapReduce: Example

Given two documents (documents 0 and 1), we want to find all k-shingles in document 0 but not in document 1 (that is, all shingles which appear at least once in document 0, but do not appear at all in document 1), where **k=1**. We receive our documents line by line, receiving input key-value pairs of the form <*docID*, *line*>, where *docID* (an integer of either 0 or 1) is the document ID being read, and *line* is a **space-separated string** containing a line of the document. For each k-shingle appearing in either document, our reduce function should emit a tuple (only once) of the form <*shingle, 1*>. Show pseudo-code for how you would use MapReduce to find all such shingles. You can assume that the input (in *line*) is 'clean'; e.g. no duplicate spaces or spaces at the start / end of the line, and no characters other than letters and spaces are present. You can assume the existence of a string splitting function of your choice, e.g. split().

Bonus: show how to use a combiner (not in-mapper combiner) to speed up the program.

```
map (docID, line) {
  /* your pseudo code*/
  /* output the map results by calling the API, emit(specify your map output)*/
}
reduce (/* specify your input to reducer */) {
  /* your pseudo code */
  /* output the map results by calling the API, emit(shingle, 1). */
}
```

MapReduce: Example

```
map (docID, line) {
  tokens = line.split()
  for token in tokens:
    emit(token, docID)
}

reduce (token, docIDs[]) {
  if 0 in docIDs and 1 in docIDs:
    emit(token, 1)
}
```

Explanation:

- 1-shingles are just words (or tokens)
- The map() function should use words as keys, so that each reduce() will handle one word
- The map() needs to emit the docID as value, since the reduce() needs this information

MapReduce: Example

```
map (docID, line) {
  tokens = line.split()
  for token in tokens:
    emit(token, docID)
}

reduce (token, docIDs[]) {
  if 0 in docIDs and 1 in docIDs:
    emit(token, 1)
}

combine (token, docIDs[]) {
  if 0 in docIDs:
    emit(token, 0)
  if 1 in docIDs:
    emit(token, 1)
}
```

Explanation:

- 1-shingles are just words (or tokens)
- The map() function should use words as keys, so that each reduce() will handle one word
- The map() needs to emit the docID as value, since the reduce() needs this information

Combiner:

- The combine() function combines multiple (token, 0) tuples into a single (token, 0) tuple, and similarly combines multiple (token, 1) tuples into a single (token, 1) tuple.
- Thus it does not change the final output, but speeds up the program by reducing the number of such tuples emitted (and thus the disk and memory I/O).

Good luck with the 2nd half of the class, and the exam!