CS4225/CS5425 Big Data Systems for Data Science

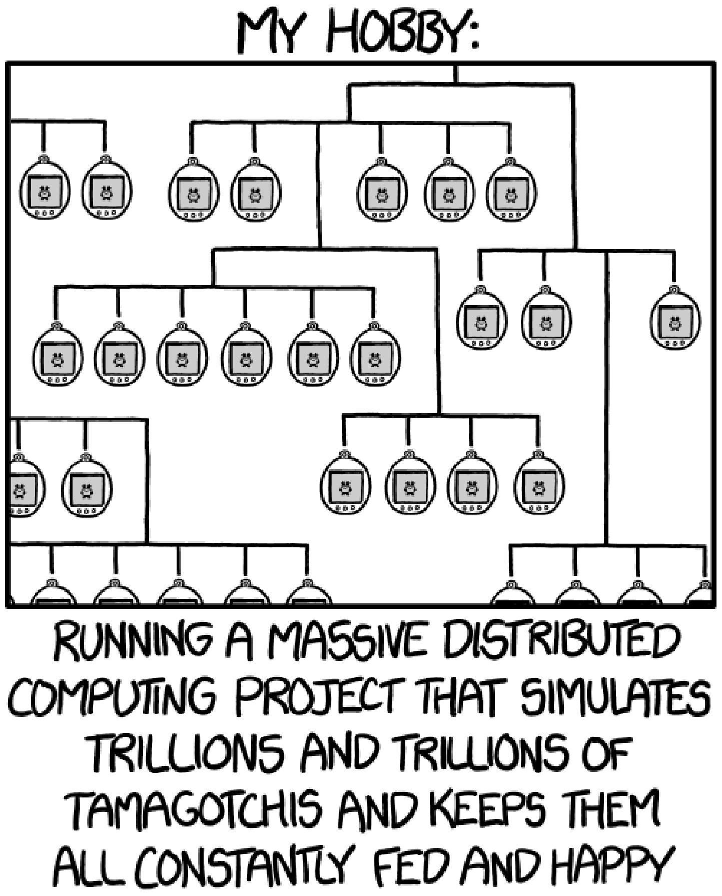
NoSQL and Distributed Databases 2

[Optional, for your own study]

Bingsheng He
School of Computing
National University of Singapore
hebs@comp.nus.edu.sg

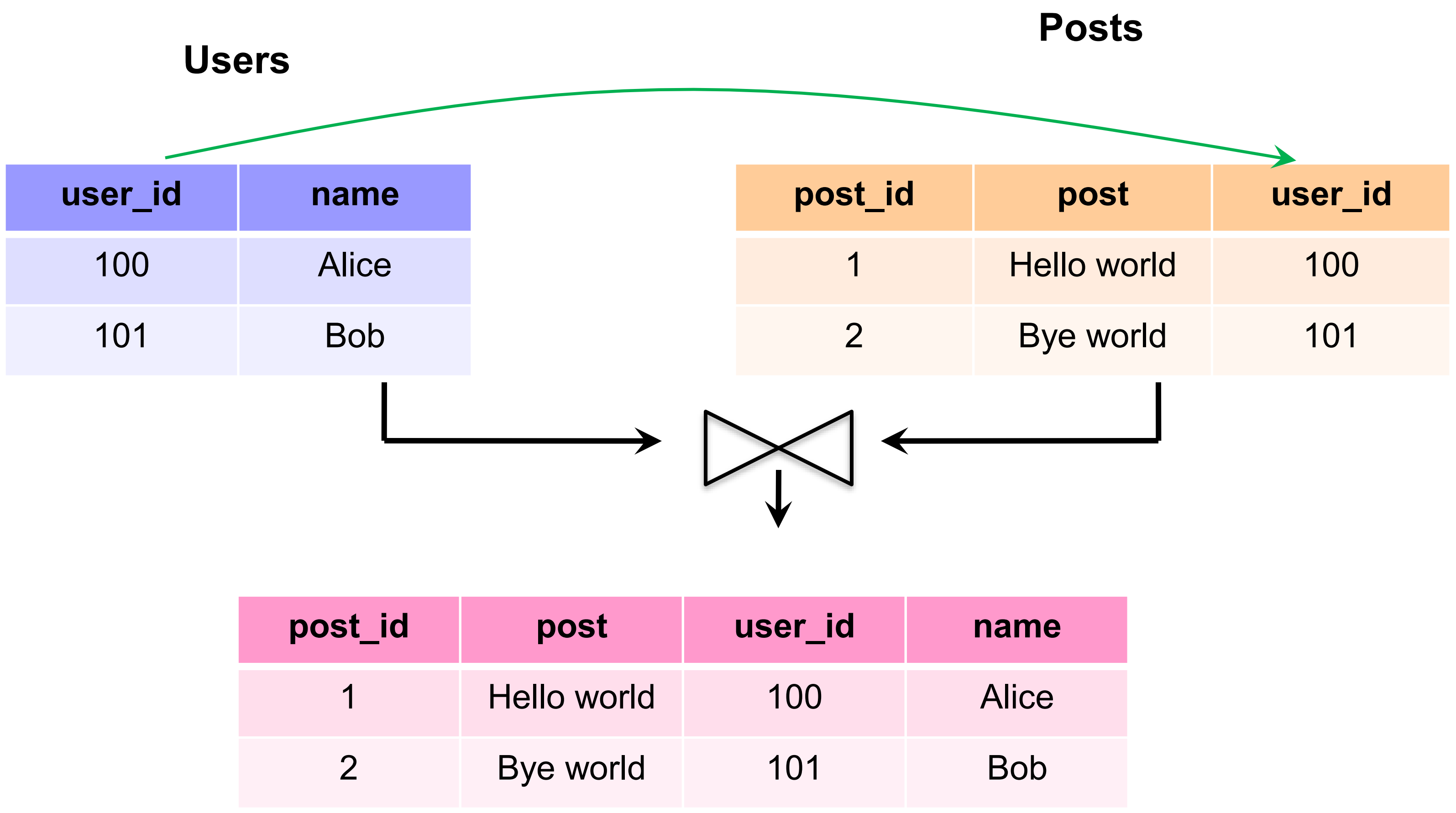
Learning Outcomes

- **Duplication in NoSQL**
- Basic Concepts of Distributed Databases
- Data Partitioning


MY HOBBY:
RUNNING A MASSIVE DISTRIBUTED COMPUTING PROJECT THAT SIMULATES TRILLIONS AND TRILLIONS OF TAMAGOTCHIS AND KEEPS THEM ALL CONSTANTLY FED AND HAPPY

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

Recall: Joins in RDBMS



Users

user_id	name
100	Alice
101	Bob

Posts

post_id	post	user_id
1	Hello world	100
2	Bye world	101

$\bowtie$

post_id	post	user_id	name
1	Hello world	100	Alice
2	Bye world	101	Bob

Duplication: displaying posts + usernames

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"
  },
]
```

- Answer 1: some NoSQL databases do support joins (e.g. later versions of MongoDB). But generally not as advanced as RDBMS joins (e.g., in terms of algorithms / query optimization)
- Answer 2: Duplication (i.e. ‘denormalization’)
  - ‘Storage is cheap: why not just duplicate data to improve efficiency?’
  - Tables are designed around the queries we expect to receive. This is beneficial especially when we mostly need to process a fixed type of queries
  - Leads to a new problem: what if user changes their name? (the change needs to be propagated to multiple tables)

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

Pros & Cons of NoSQL Systems

Pros

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

+ **Horizontal scalability**

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

Cons

- **Less sophisticated query language**: SQL has highly optimized and advanced querying capabilities; NoSQL key-value stores lack complex querying abilities

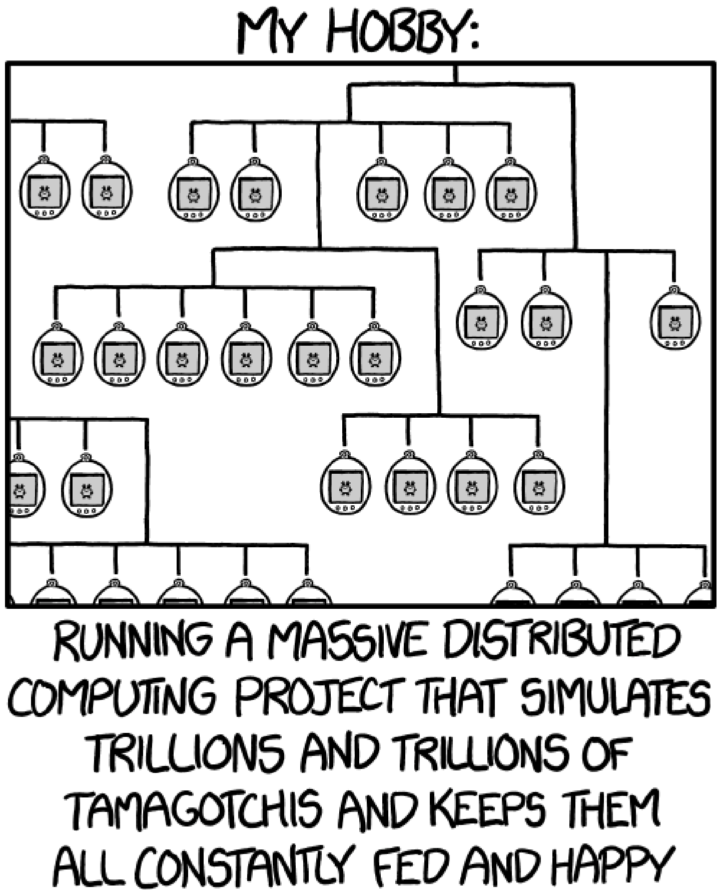
- **Weaker consistency guarantees / denormalization**: application may receive stale data that may need to be handled on the application side

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

- Duplication in NoSQL
- **Basic Concepts of Distributed Databases**
- Data Partitioning


MY HOBBY:
RUNNING A MASSIVE DISTRIBUTED COMPUTING PROJECT THAT SIMULATES TRILLIONS AND TRILLIONS OF TAMAGOTCHIS AND KEEPS THEM ALL CONSTANTLY FED AND HAPPY

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

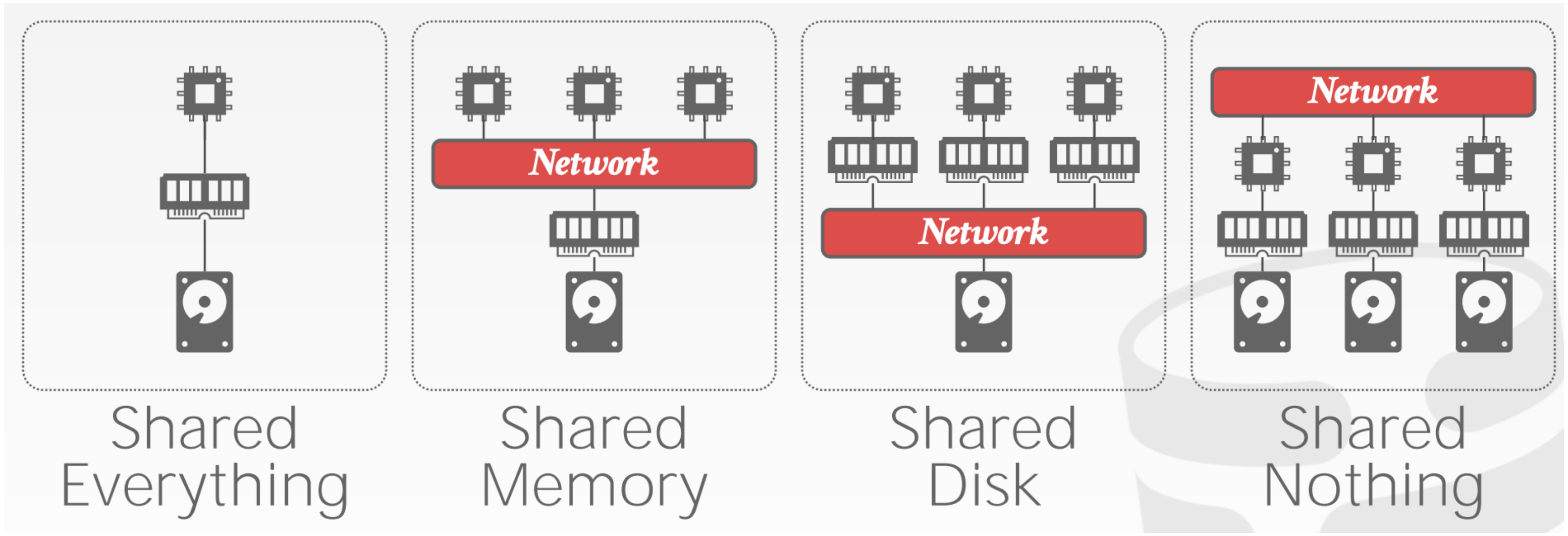
Introduction to Distributed Databases

- 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 serve 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.

Distributed Database Architectures



Examples:	Single node DBMS	Supercomputers	Cloud databases	NoSQL (mostly)

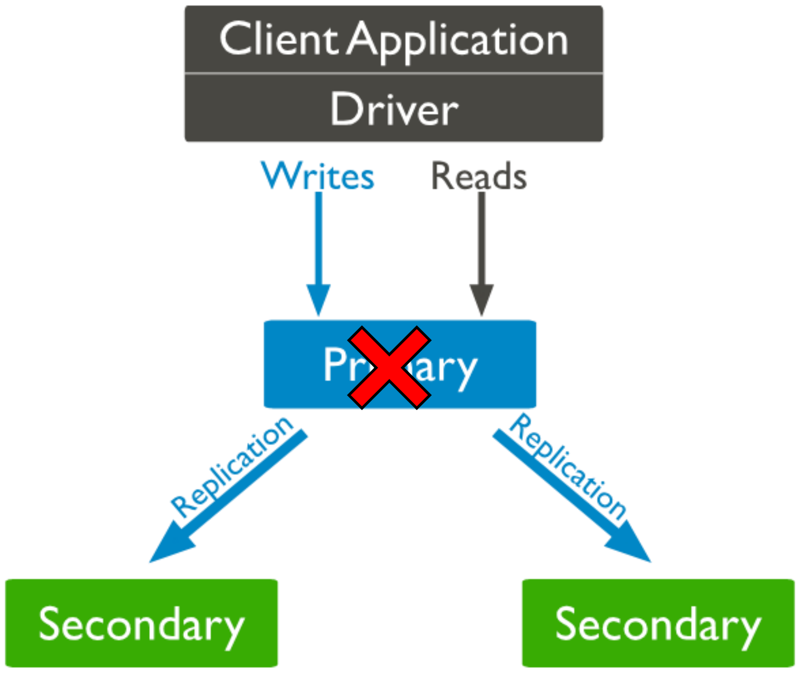
ORACLE RAC
snowflake

Cockroach LABS
H-Store
cassandra
redis
mongoDB
CouchDB relax
MySQL Cluster

Credit: Andy Pavlo (CMU CS15-446)

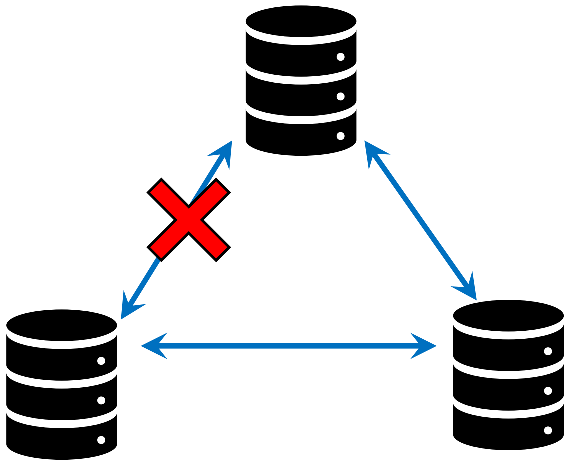
Assumptions of Distributed Databases

- Nodes can go down (requiring the system to have sufficient redundancy; just as we observed in Hadoop, HDFS, NoSQL, etc.)

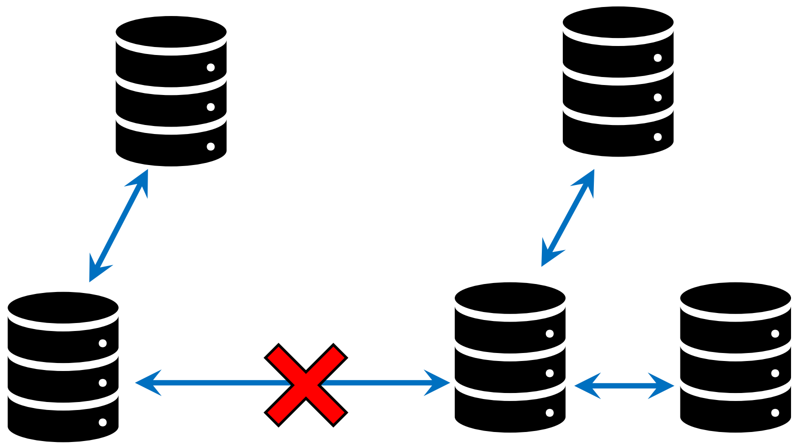


Assumptions of Distributed Databases

- Network connections between some nodes may fail



- In some cases, connection failures cause the network to split into 2 parts which are completely cut off from each other. This is called a **partition.**

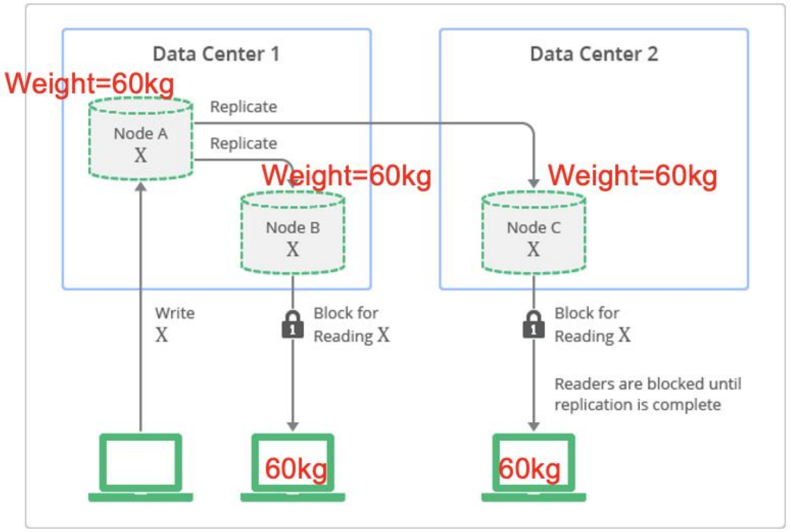


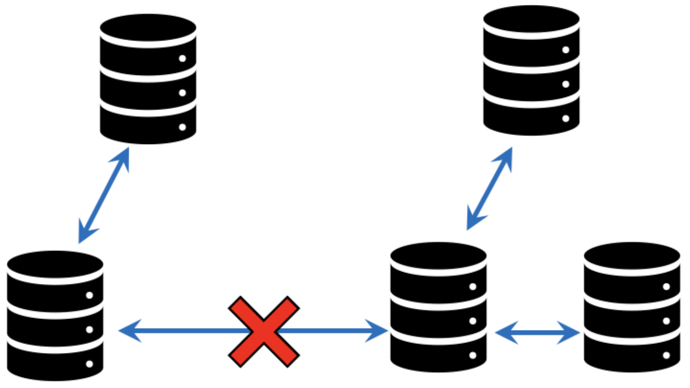
Assumptions of Distributed Databases

- 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)

CAP Theorem

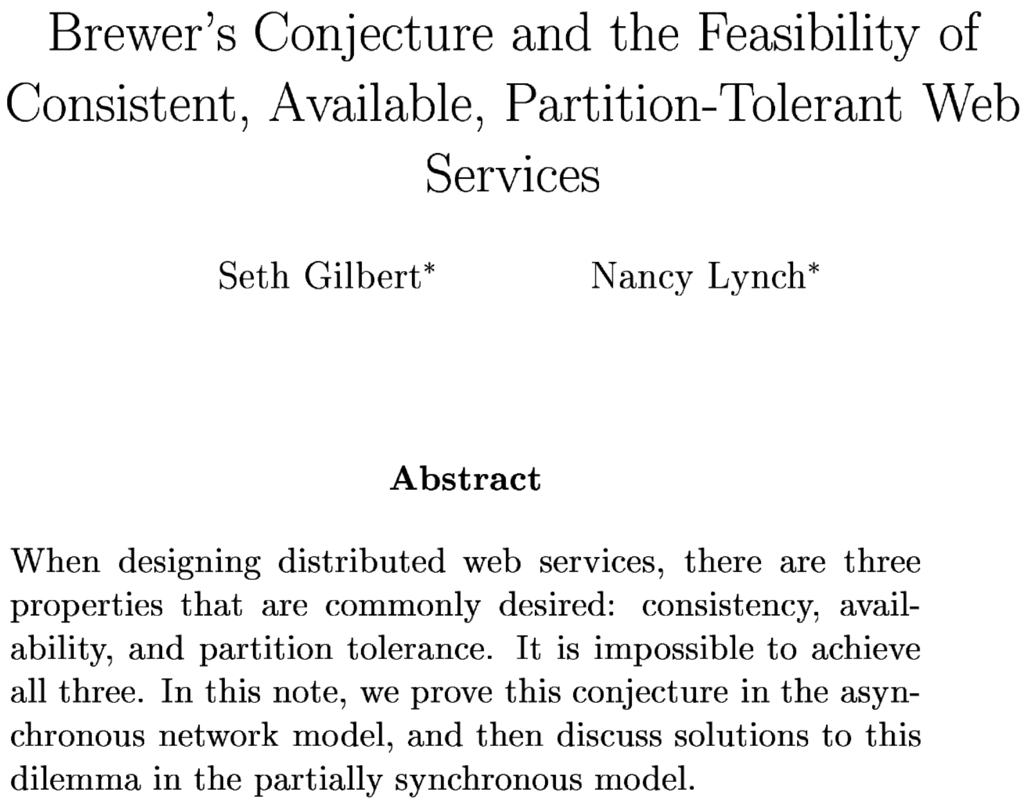
- **Consistency**: Every read receives the most recent write or an error.
  - Basically same as the notion of "Strong Consistency" we have discussed
- **Availability**: Every request to a node in the system must lead to a response.
- **Partition Tolerance**: The system continues to operate even when network failures cause a partition into sub-networks.





CAP Theorem

A distributed system cannot have all 3 of Consistency, Availability, and Partition Tolerance.

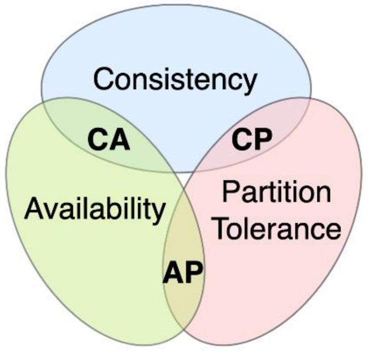


Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services

Seth Gilbert* Nancy Lynch*

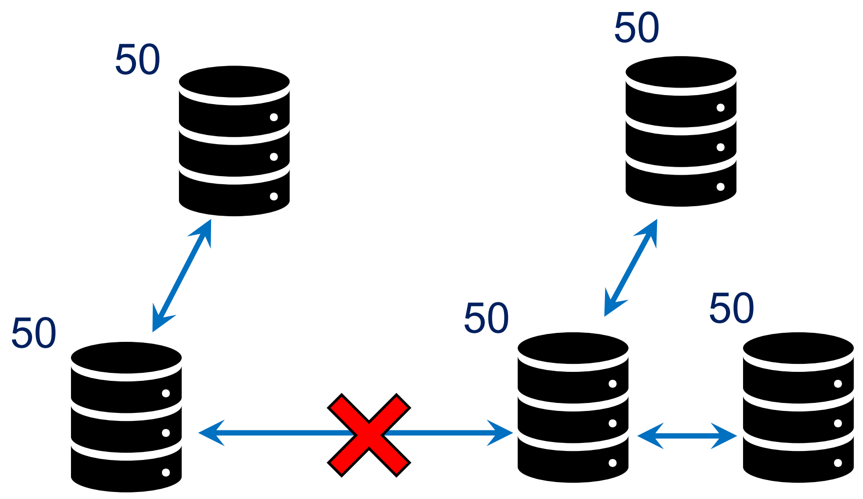
Abstract

When designing distributed web services, there are three properties that are commonly desired: consistency, availability, and partition tolerance. It is impossible to achieve all three. In this note, we prove this conjecture in the asynchronous network model, and then discuss solutions to this dilemma in the partially synchronous model.



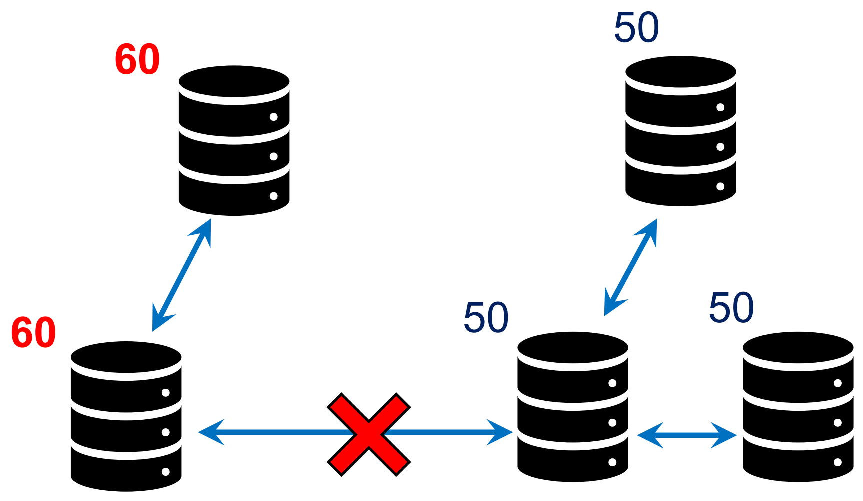
CAP Theorem: Intuitive Explanation

- Suppose we have some data (e.g., 50), and a network partition occurs.



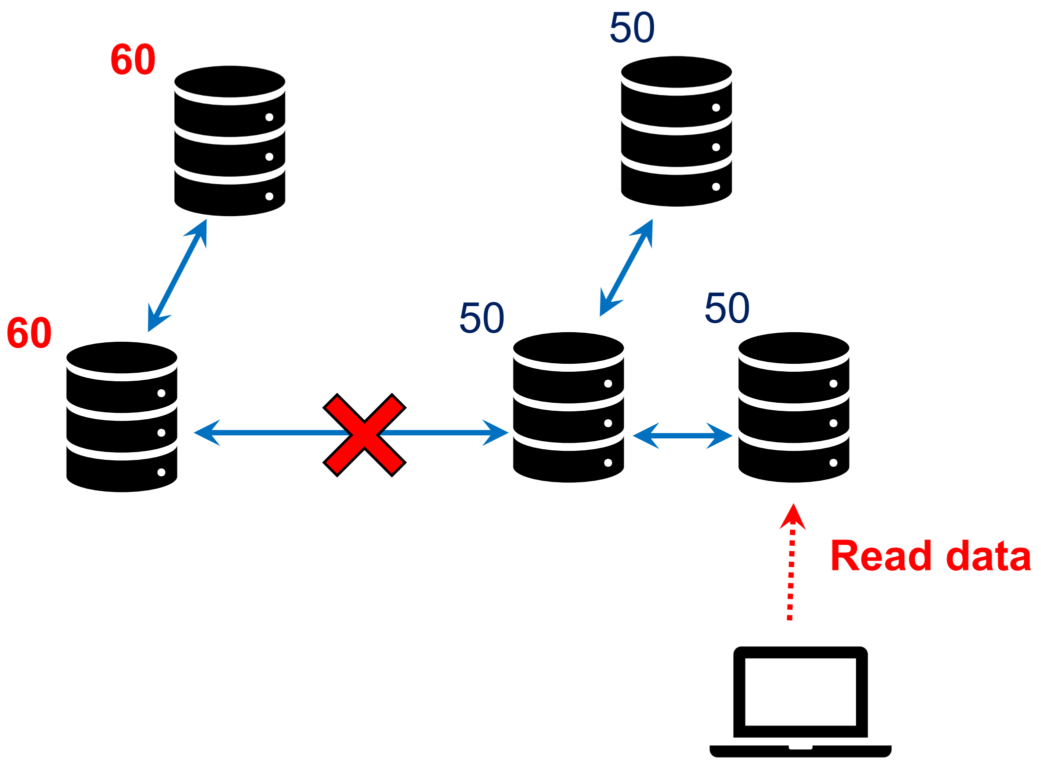
CAP Theorem: Intuitive Explanation

- We make a write to partition 1 (e.g., 50 → 60). The information can be propagated in partition 1, but not to partition 2.



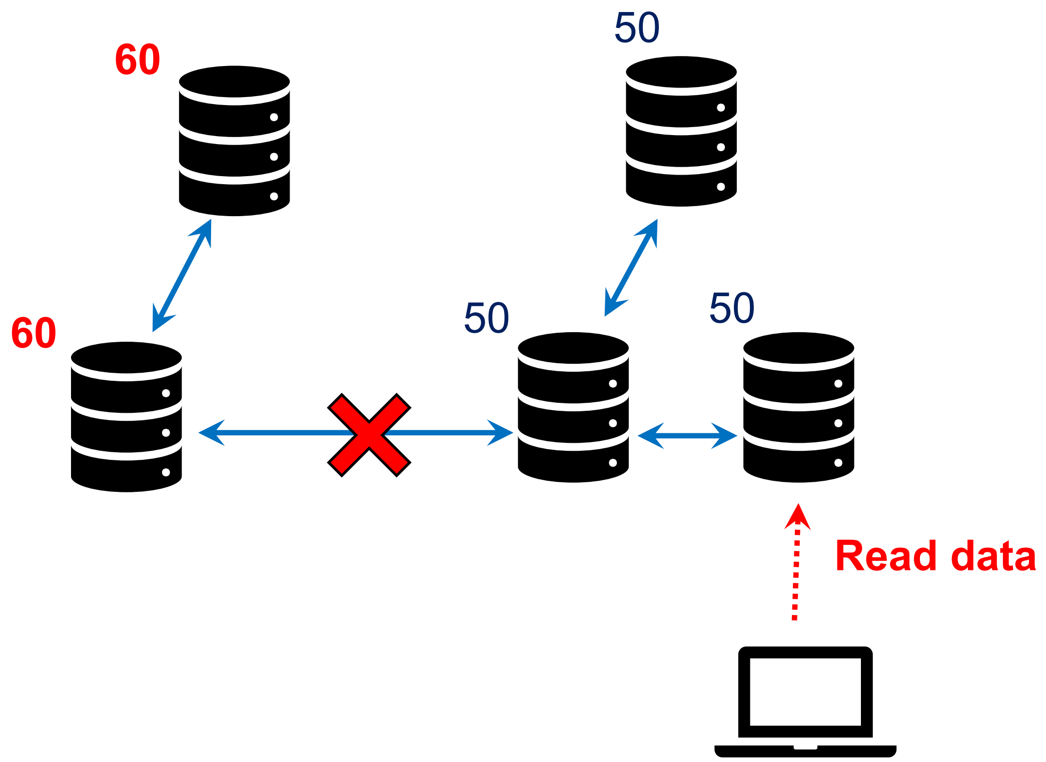
CAP Theorem: Intuitive Explanation

- Now suppose we try to read from partition 2.



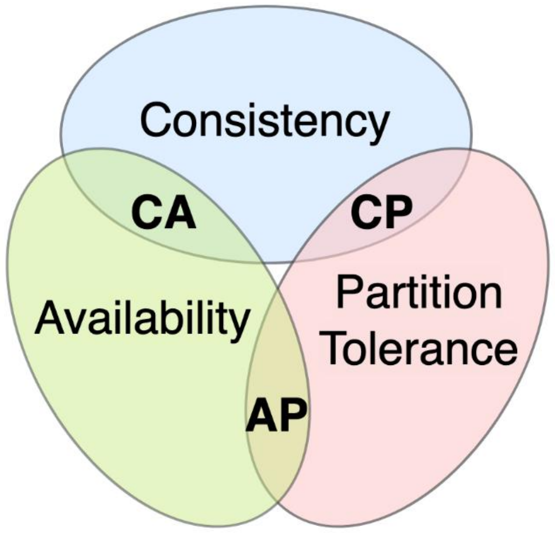
CAP Theorem: Intuitive Explanation

- Now we have 2 choices:
  - Return the outdated value of 50, which **sacrifices Consistency** (as 50 is not the most recently written value)
  - Or: return an error (or no response), which **sacrifices Availability**



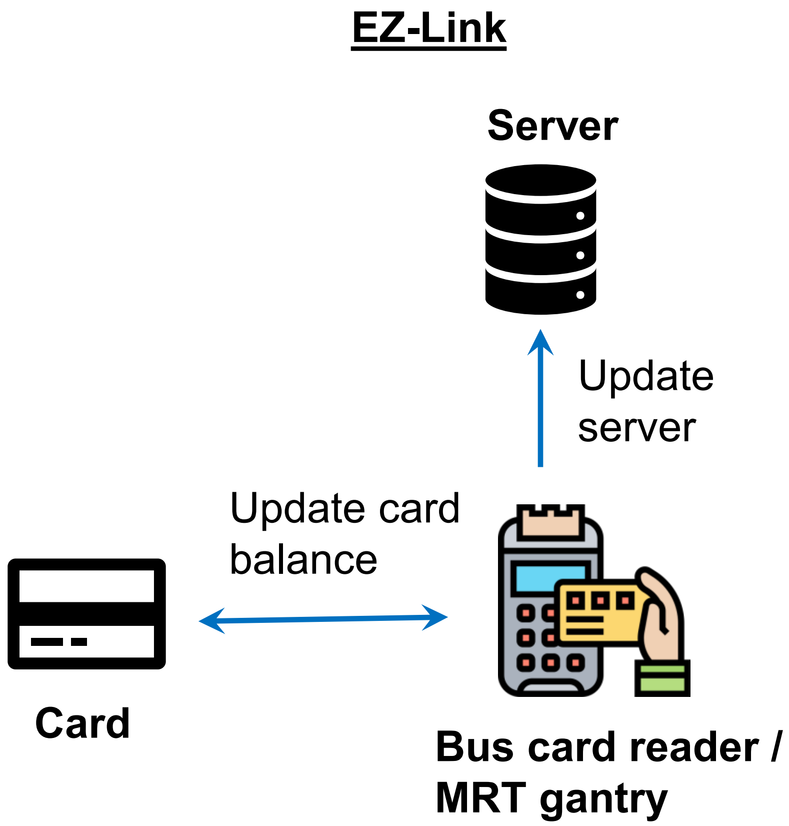
CAP Theorem: Conclusion

- In the presence of partitions, distributed systems have to sacrifice either Consistency or Availability
  - Generally, NoSQL systems tend to sacrifice Consistency, while (distributed) RDBMS tend to sacrifice Availability. But this is not a universal rule, e.g. both NoSQL and RDBMS can have “tunable consistency” levels.
  - More accurate to say that whether RDBMS or NoSQL, tuning a system for (Strong) Consistency results in sacrificing Availability



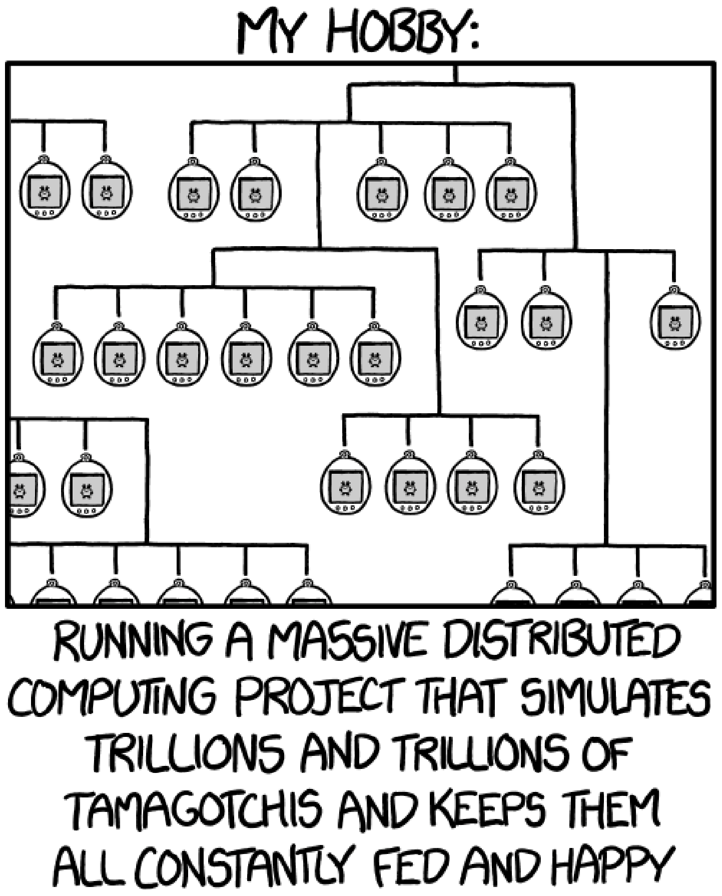
CAP Theorem: EZLink Example

- Suppose we are in a bus which loses internet connection ("**partition**")
- When the user tries to scan the card, the system lets them through, prioritizing **availability** but sacrificing **consistency** (i.e., card balance on server is temporarily outdated)



Today's Plan

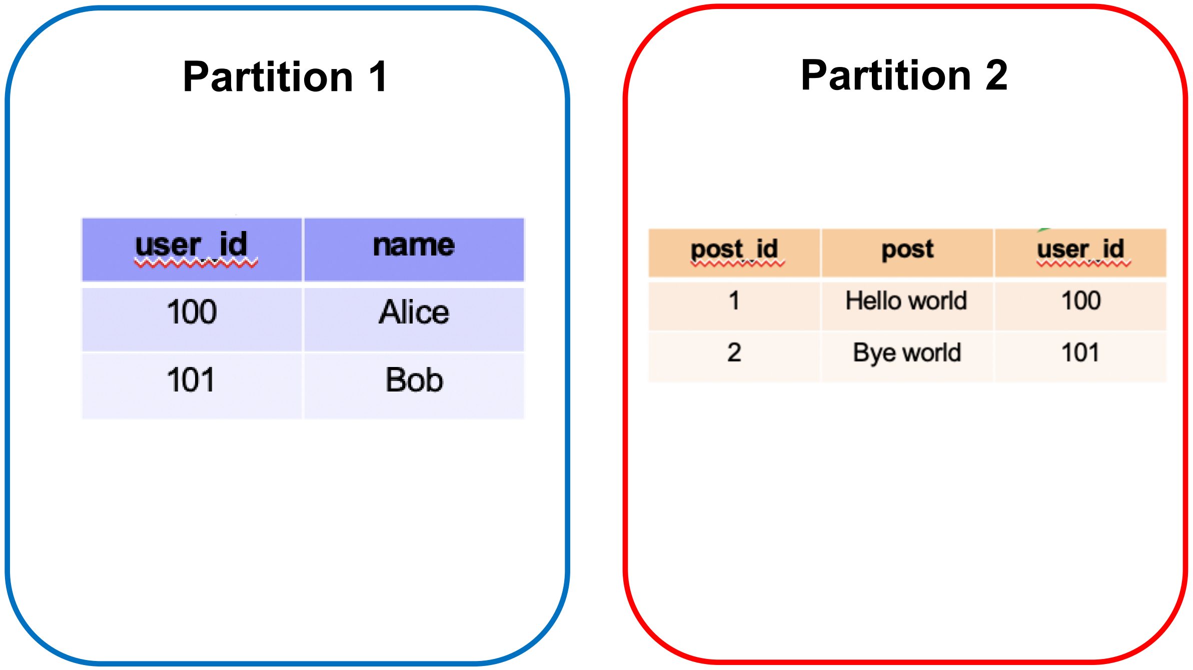
- Duplication in NoSQL
- Basic Concepts of Distributed Databases
- **Data Partitioning**


MY HOBBY:
RUNNING A MASSIVE DISTRIBUTED COMPUTING PROJECT THAT SIMULATES TRILLIONS AND TRILLIONS OF TAMAGOTCHIS AND KEEPS THEM ALL CONSTANTLY FED AND HAPPY

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

Table Partitioning

- Put different tables (or collections) on different machines



Partition 1

user_id	name
100	Alice
101	Bob

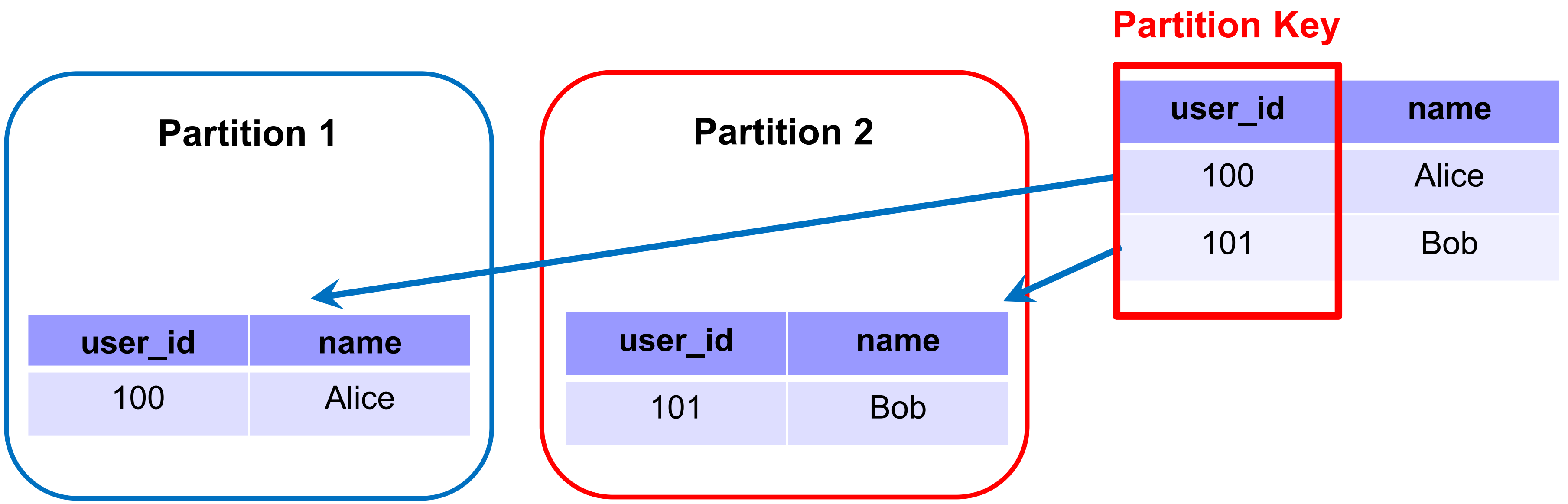
Partition 2

post_id	post	user_id
1	Hello world	100
2	Bye world	101

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

Horizontal Partitioning

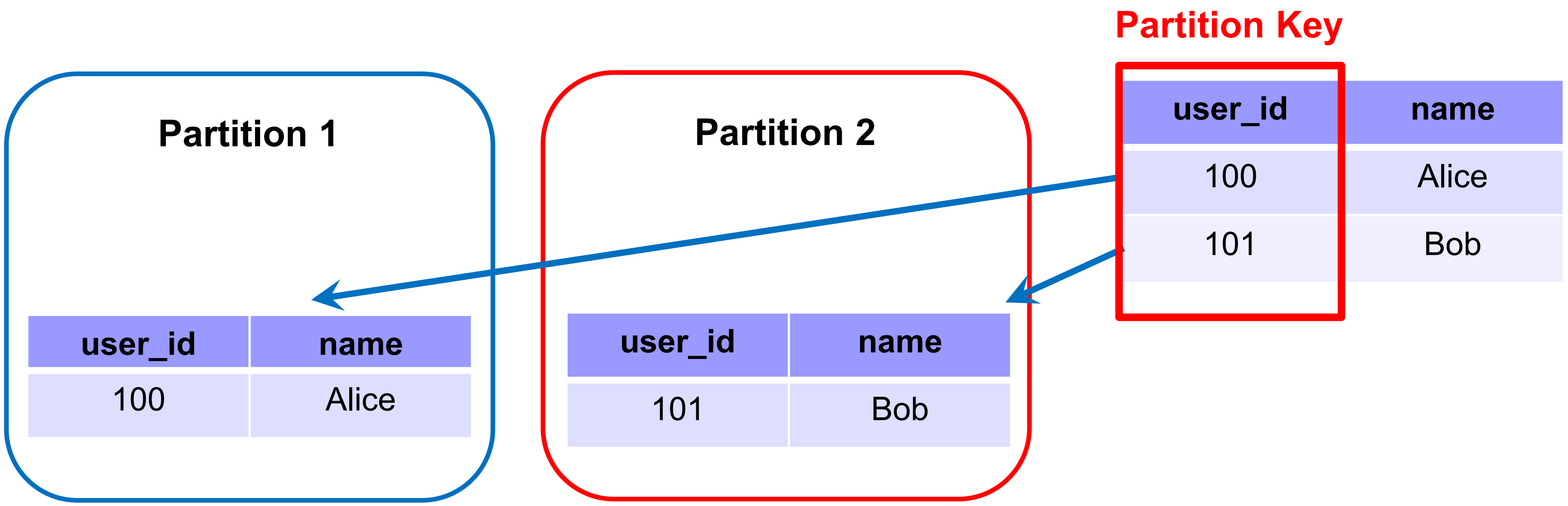
- Different tuples are stored in different nodes



- 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



user_id	name
100	Alice
101	Bob

Partition 1:

user_id	name
100	Alice

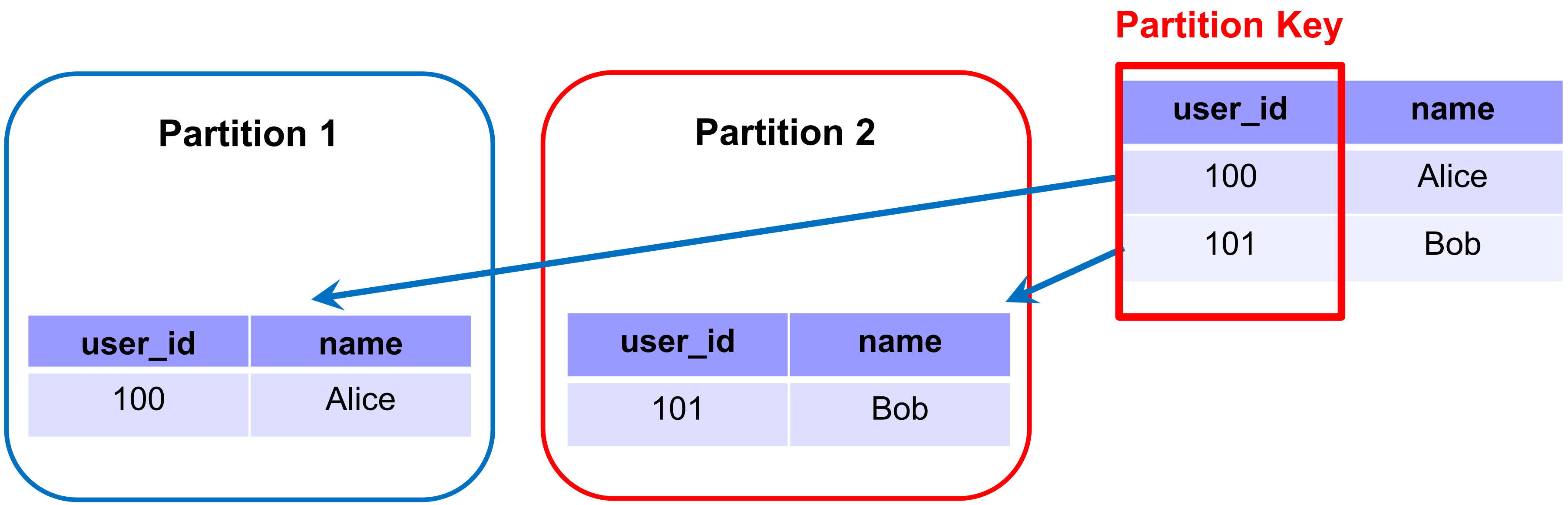
Partition 2:

user_id	name
101	Bob

- Q: imagine you have an e-commerce company, which stores a document database containing customer information. You use each user's city, city_id as a partition key; when is this good / bad?

Horizontal Partitioning

- Different tuples are stored in different nodes



user_id	name
100	Alice
101	Bob

Partition 1:

user_id	name
100	Alice

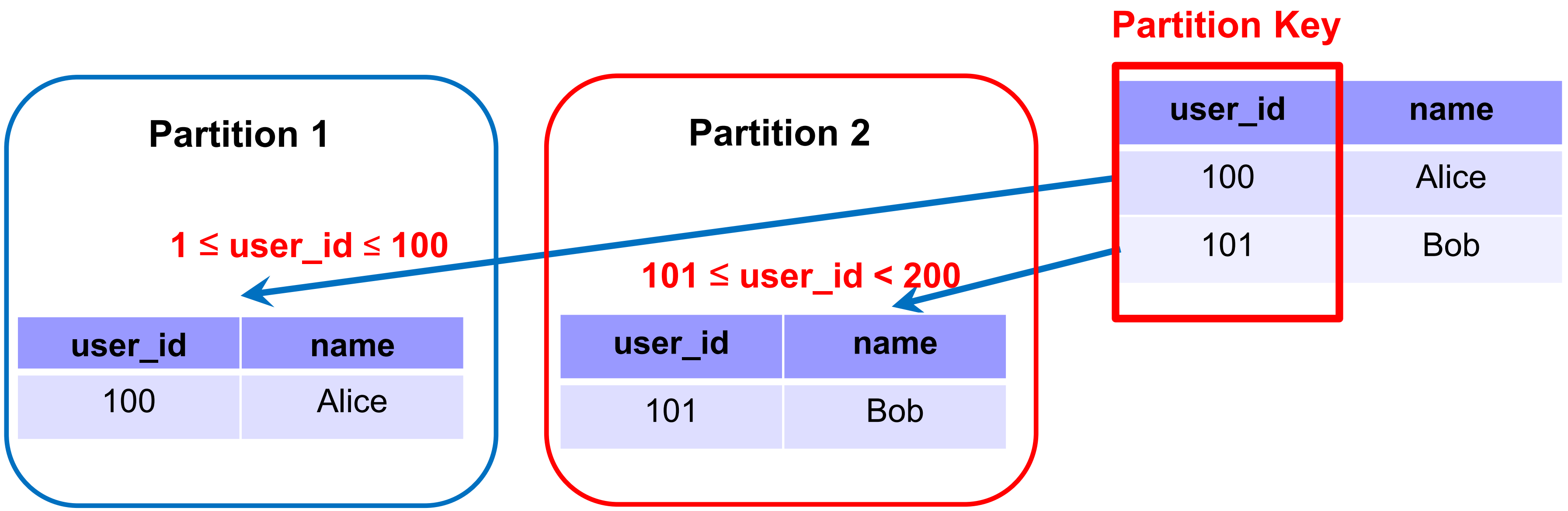
Partition 2:

user_id	name
101	Bob

- Q: imagine you have an e-commerce company, which stores a document database containing customer information. You use each user's city, city_id as a partition key; when is this good / bad?
- A: Good if we mostly query data only by cities, or read/write data for individual cities, allowing partition pruning. 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



user_id	name
100	Alice
101	Bob

Partition 1:

user_id	name
100	Alice

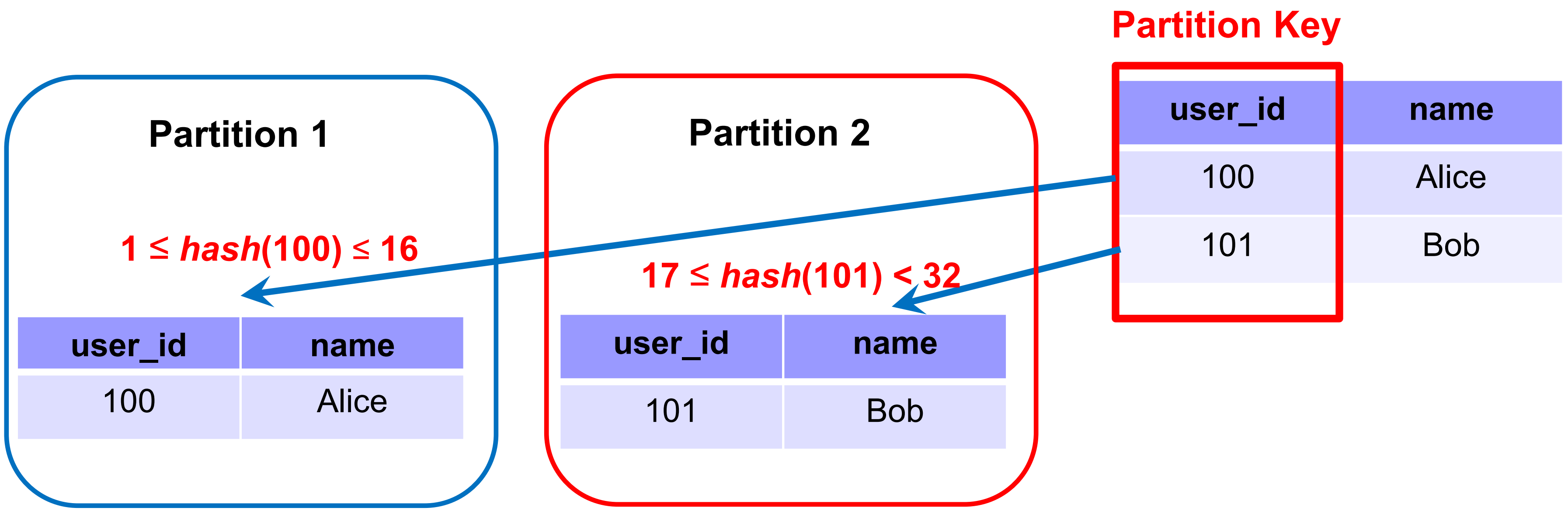
Partition 2:

user_id	name
101	Bob

- **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



user_id	name
100	Alice
101	Bob

Partition 1:

user_id	name
100	Alice

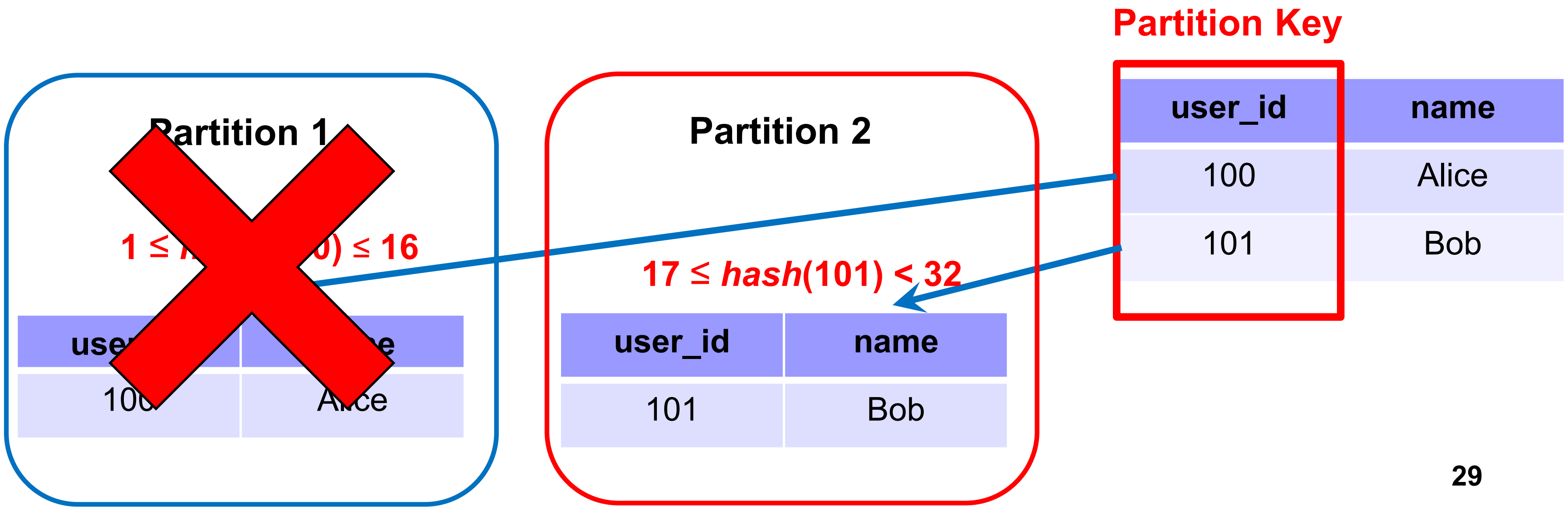
Partition 2:

user_id	name
101	Bob

- **Hash Partition**: hash partition key, then divide that into partitions based on ranges
  - Hash function automatically spreads out partition key values roughly evenly

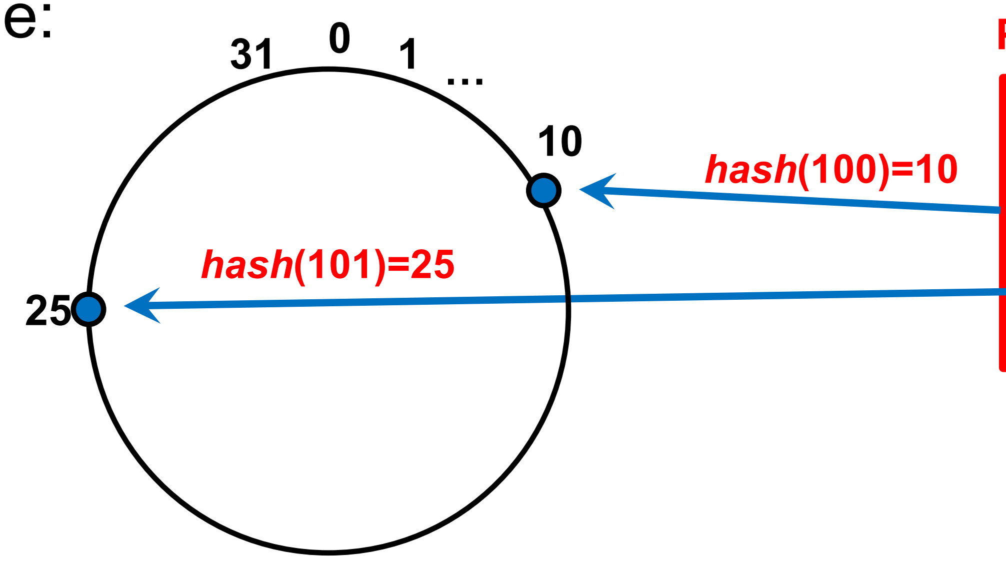
Consistent Hashing: Motivation

- **Weakness of previous approaches**: in hash / range partitioning, how do we add / remove a node (say, if we want to partition over more nodes, or if a node goes down)
  - If we completely redo the partition, a lot of data may have to be moved around, which is inefficient.
  - Solution: consistent hashing (covered next)



Consistent Hashing

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

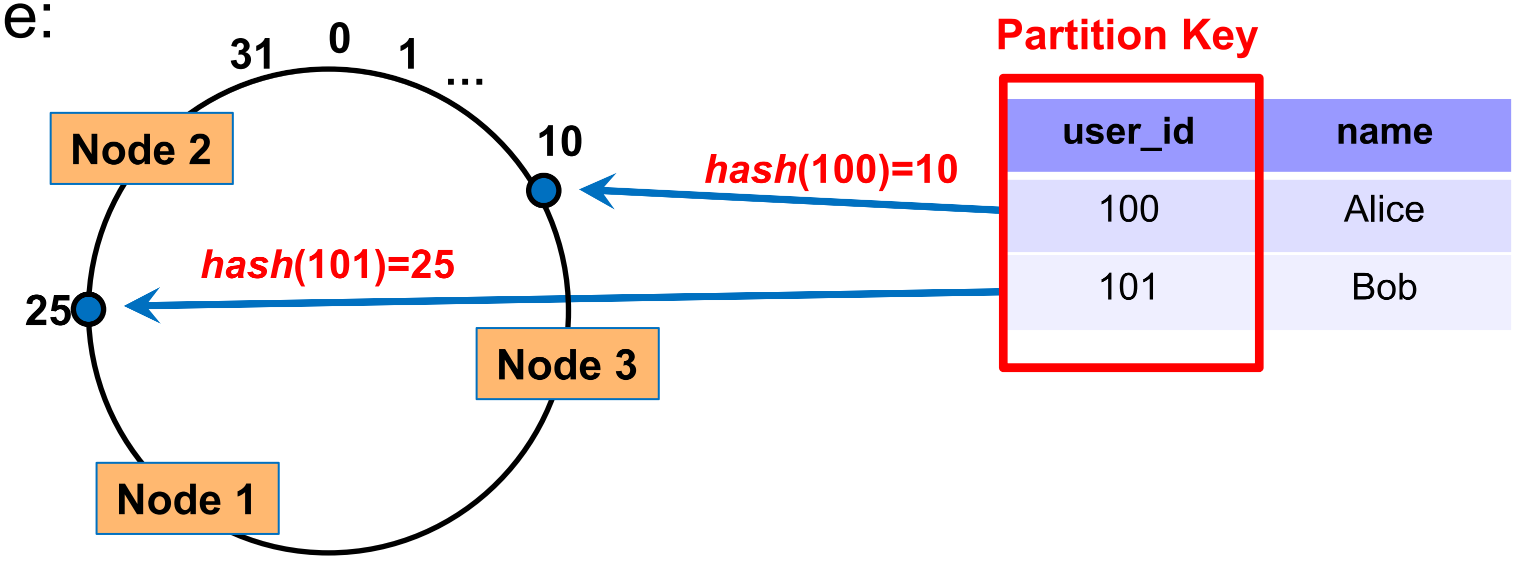


Partition Key

user_id	name
100	Alice
101	Bob

Consistent Hashing

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

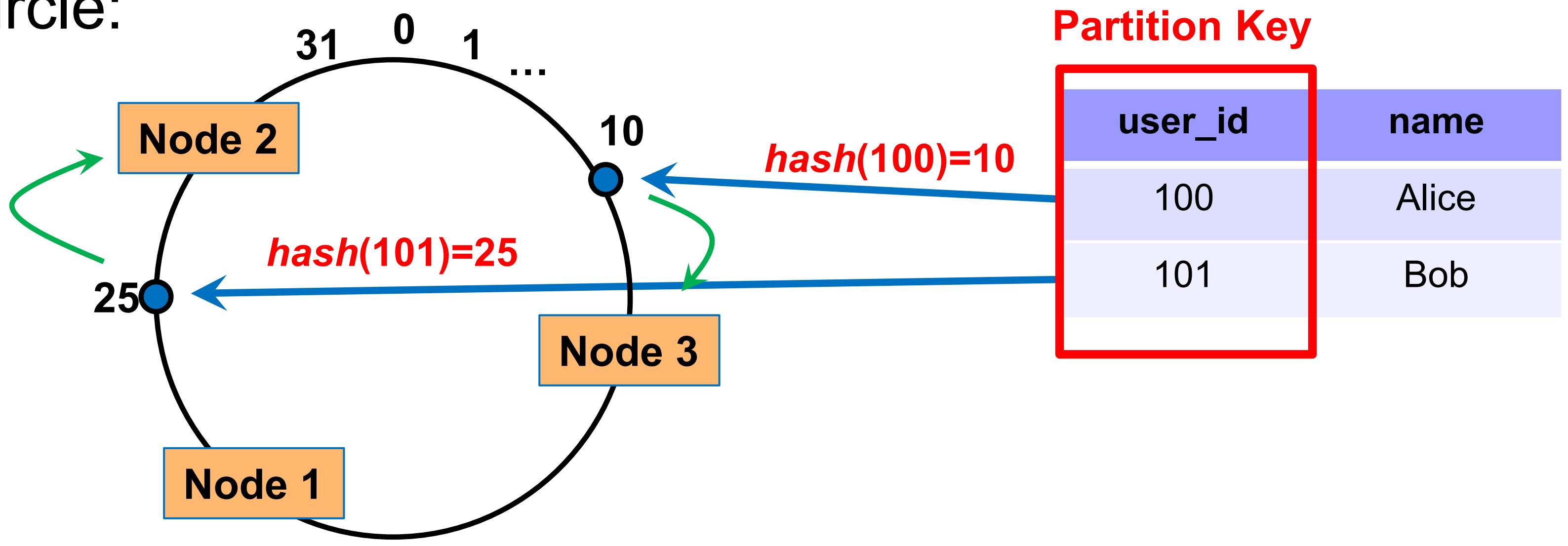


user_id	name
100	Alice
101	Bob

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

Consistent Hashing

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

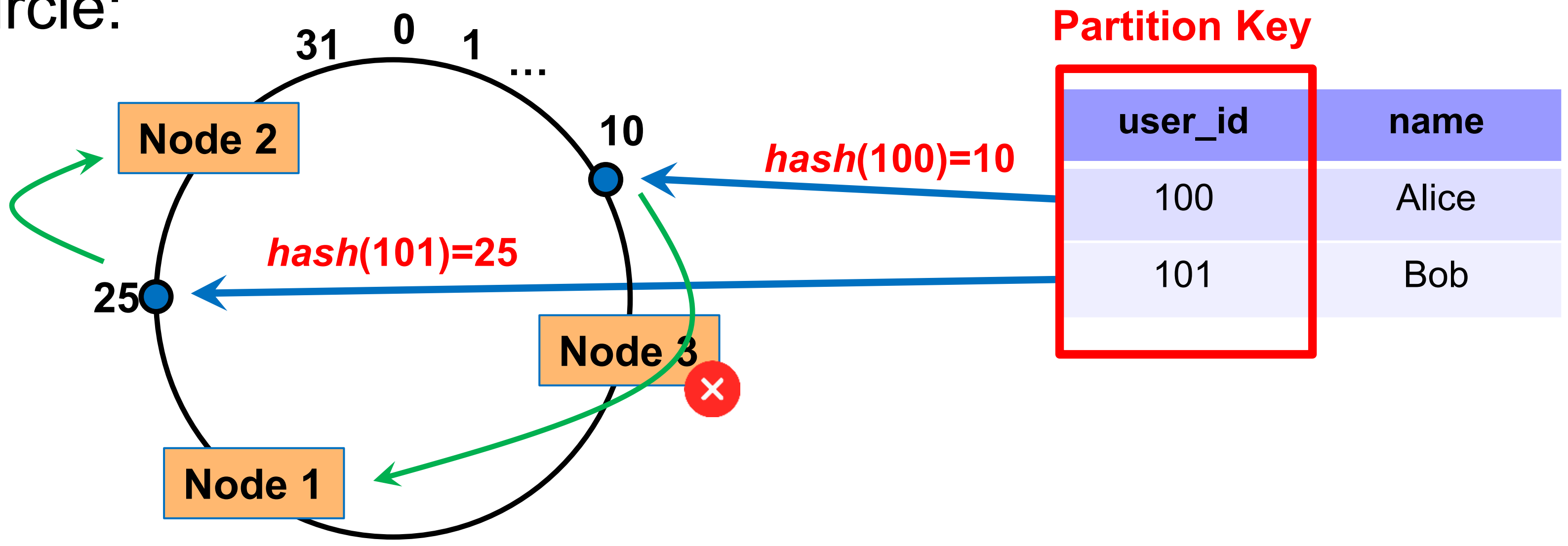


user_id	name
100	Alice
101	Bob

- **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:

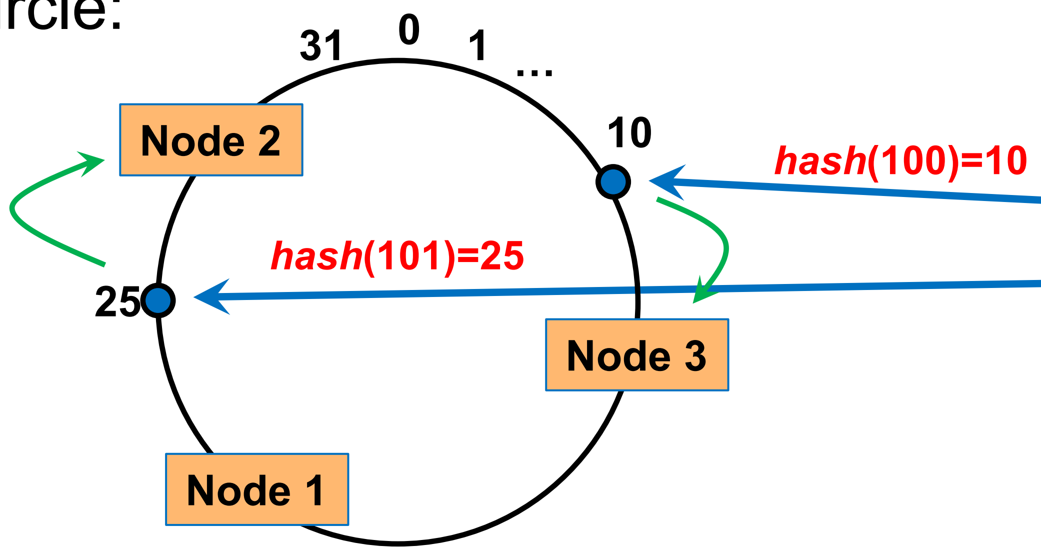


user_id	name
100	Alice
101	Bob

- 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:



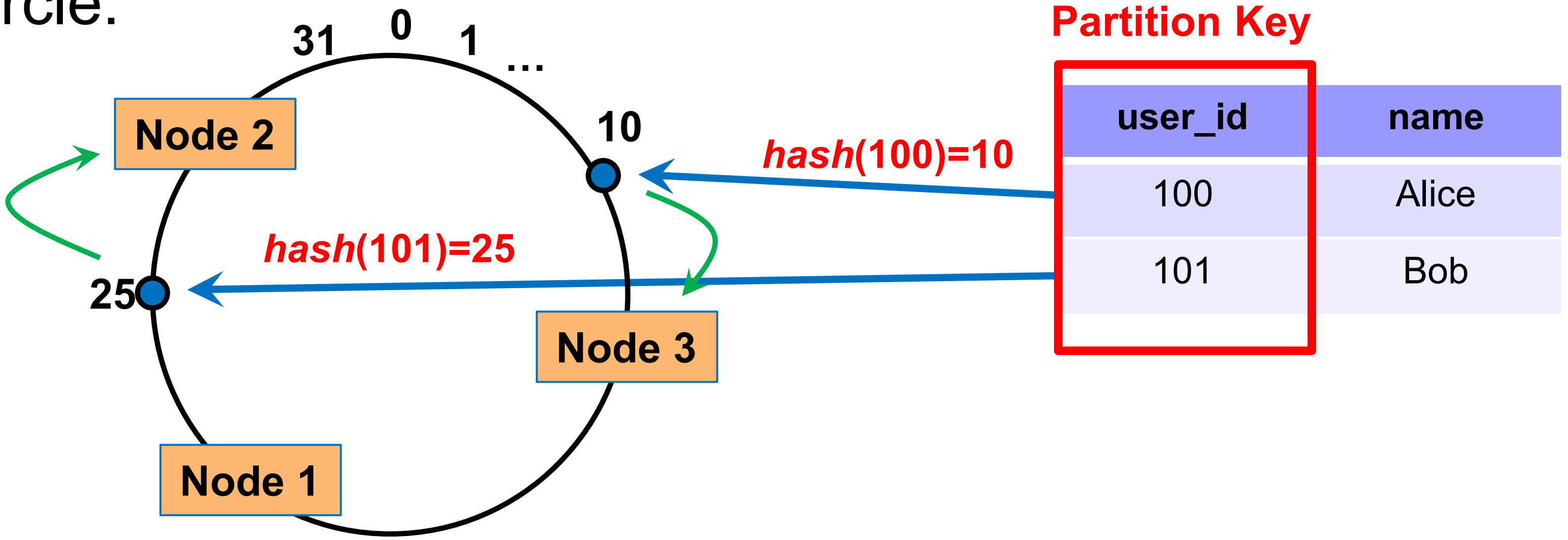
Partition Key

user_id	name
100	Alice
101	Bob

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

Consistent Hashing

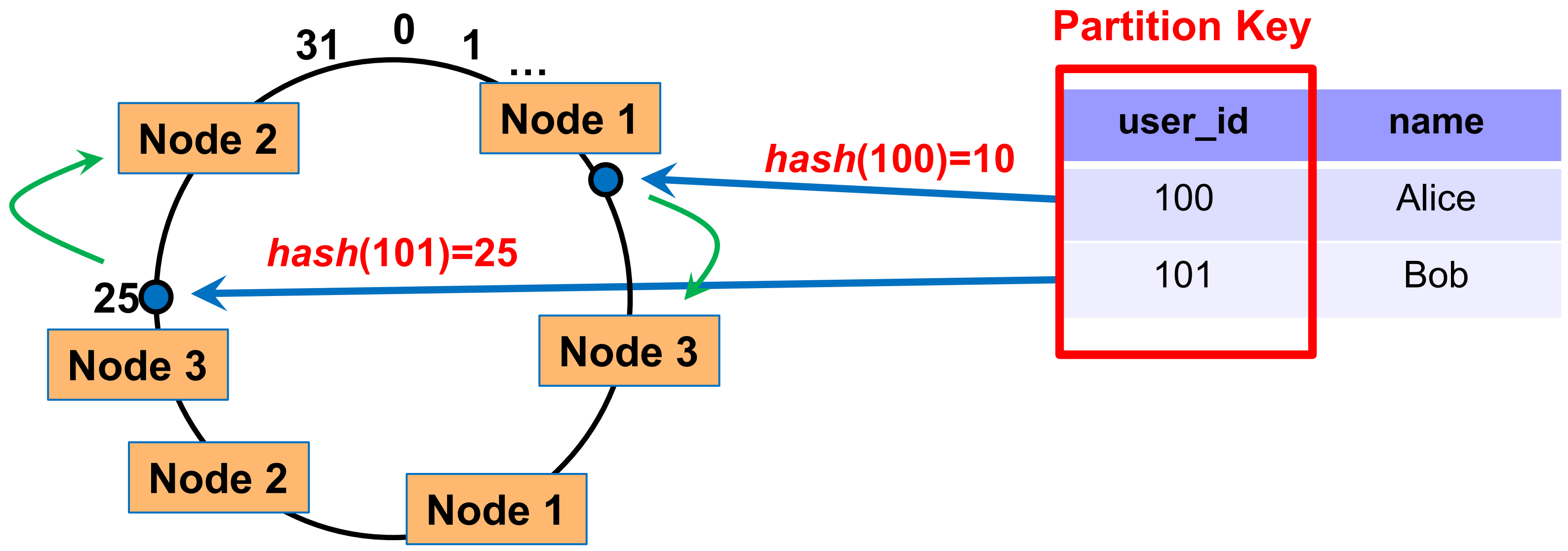
- Think of the output of the hash function as lying on a circle:



user_id	name
100	Alice
101	Bob

- **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



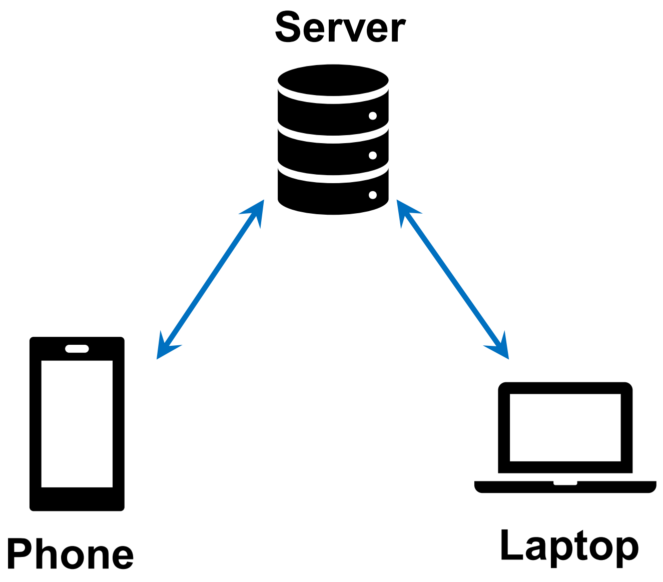
user_id	name
100	Alice
101	Bob

- **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.

When poll is active respond at **PollEv.com/cs4225**

Q1: Consider the iCloud example discussed last lecture. After taking a photo, iCloud tries to sync it with the server. If the connection is down, it keeps trying every few seconds. Which of CAP does this sacrifice?

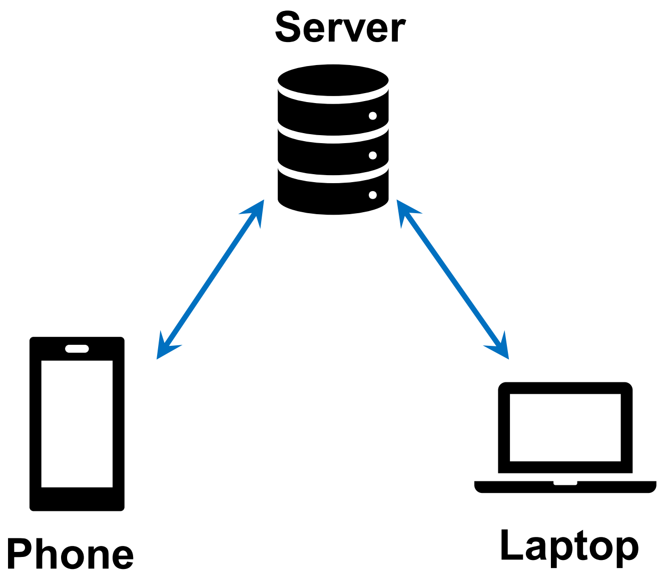
1. C
2. A
3. P



When poll is active respond at **PollEv.com/cs4225**

Q1: Consider the iCloud example discussed last lecture. After taking a photo, iCloud tries to sync it with the server. If the connection is down, it keeps trying every few seconds. Which of CAP does this sacrifice?

1. [✓ C]
2. A
3. P



When poll is active respond at **PollEv.com/cs4225**

Q2: True/False:

1. The “Consistency” in CAP refers to Strong Consistency.
2. Database systems which satisfy the ACID “Consistency” guarantee must satisfy Strong Consistency.
3. Range partitioning is suited for time series data where we often need to read data within periods of time.
4. In consistent hashing, adding or removing nodes both lead to relatively little movement of data across the database.

When poll is active respond at **PollEv.com/cs4225**

Q2: True/False:

1. The “Consistency” in CAP refers to Strong Consistency.
2. Database systems which satisfy the ACID “Consistency” guarantee must satisfy Strong Consistency.
3. Range partitioning is suited for time series data where we often need to read data within periods of time.
4. In consistent hashing, adding or removing nodes both lead to relatively little movement of data across the database.

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
  - Further readings
  - Supplementary slides

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 in the final answer, your 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 not 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 not in docIDs:
    emit(token, 1)
}

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

<u>Explanation</u>:

- 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

<u>Combiner:</u>

- 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 network I/O).

Acknowledgement

- Slides adopted/revised from
  - Jimmy Lin, http://lintool.github.io/UMD-courses/bigdata-2015-Spring/
  - Bryan Hooi
- Some slides are also adopted/revised from
  - Jure Leskovec, Anand Rajaraman, and Jeffrey David Ullman. 2014. Mining of Massive Datasets (2nd ed.). Cambridge University Press. http://www.mmds.org/