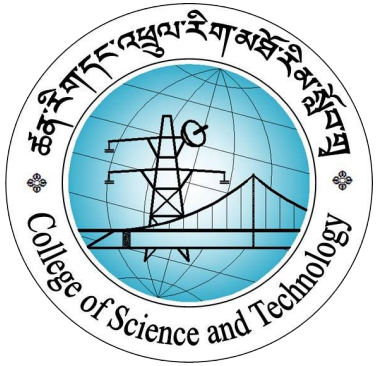


Royal University of Bhutan

DBS101 Database Systems Fundamentals

Lesson 11

Learning Outcomes

1. Understand data management in distributed databases.
2. Explain the BASE principles of non-relational databases.
3. Differentiate types of eventual consistency models.
4. Describe different types of non-relational databases.

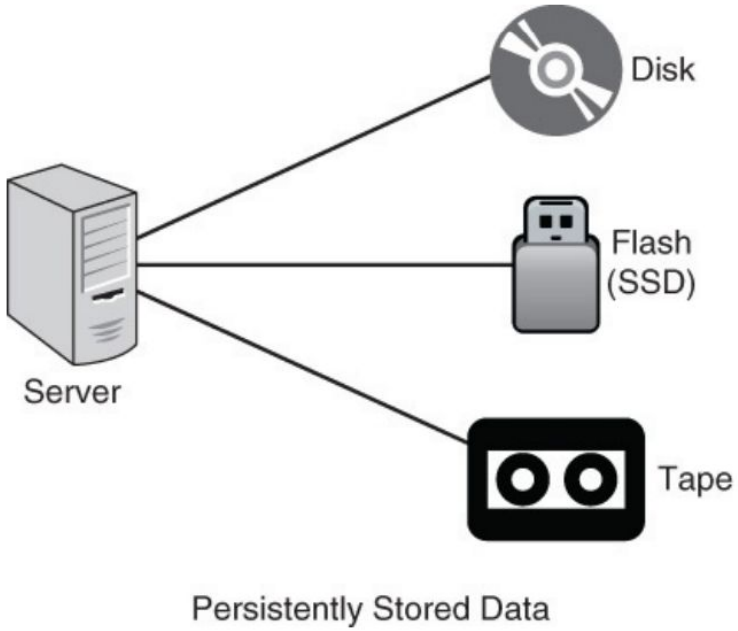
Data Management with Distributed Databases

Databases are designed to do two things: store data and retrieve data.

To meet these objectives, the database management systems must do three things:

- Store data persistently
- Maintain data consistency
- Ensure data availability

Data Management with Distributed Databases



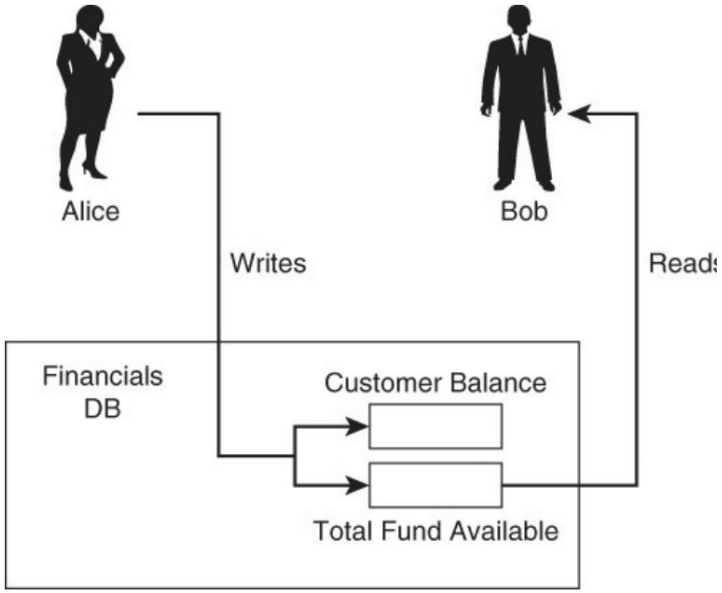


Figure 2.3 *Data should reflect a consistent state.*

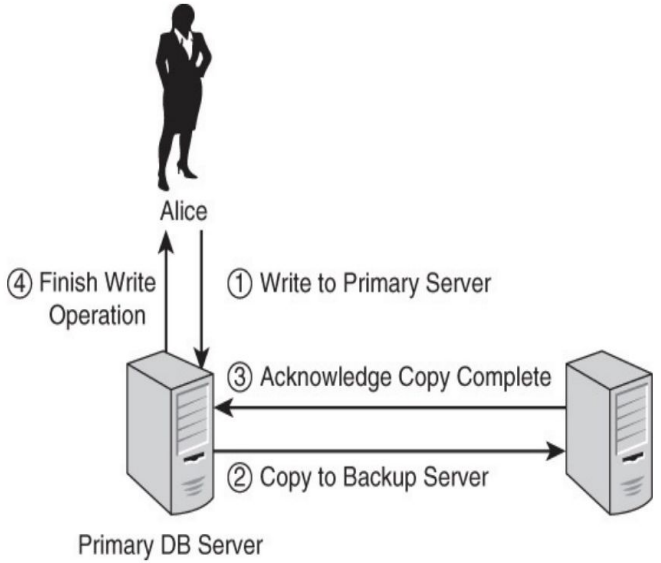


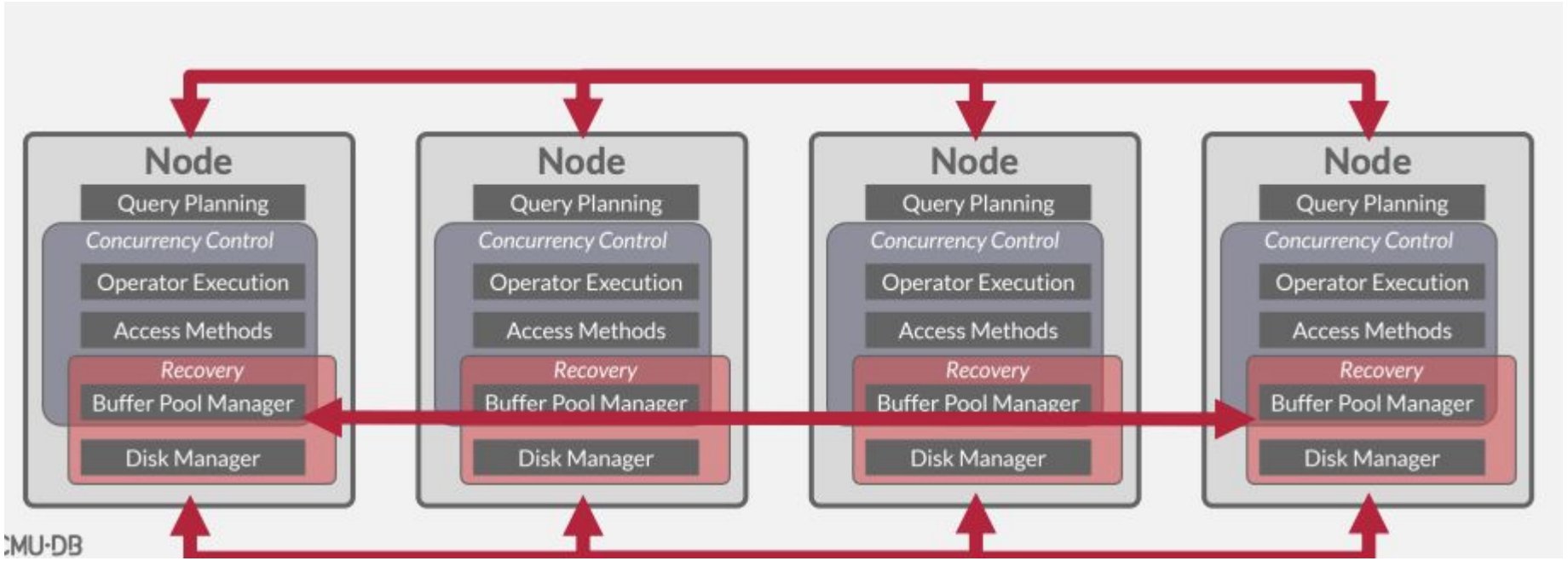
Figure 2.4 *Two-phase commits are complete only when operations on both databases are complete.*

Distributed DBMS

Database management systems can run on one or more computers.

When the database management system is running on multiple computers, it is called a distributed database.

Distributed Databases



Parallel vs. Distributed

Parallel DBMSs:

- Nodes are physically close to each other.
- Nodes connected with high-speed LAN.
- Communication cost is assumed to be small.

Distributed DBMSs:

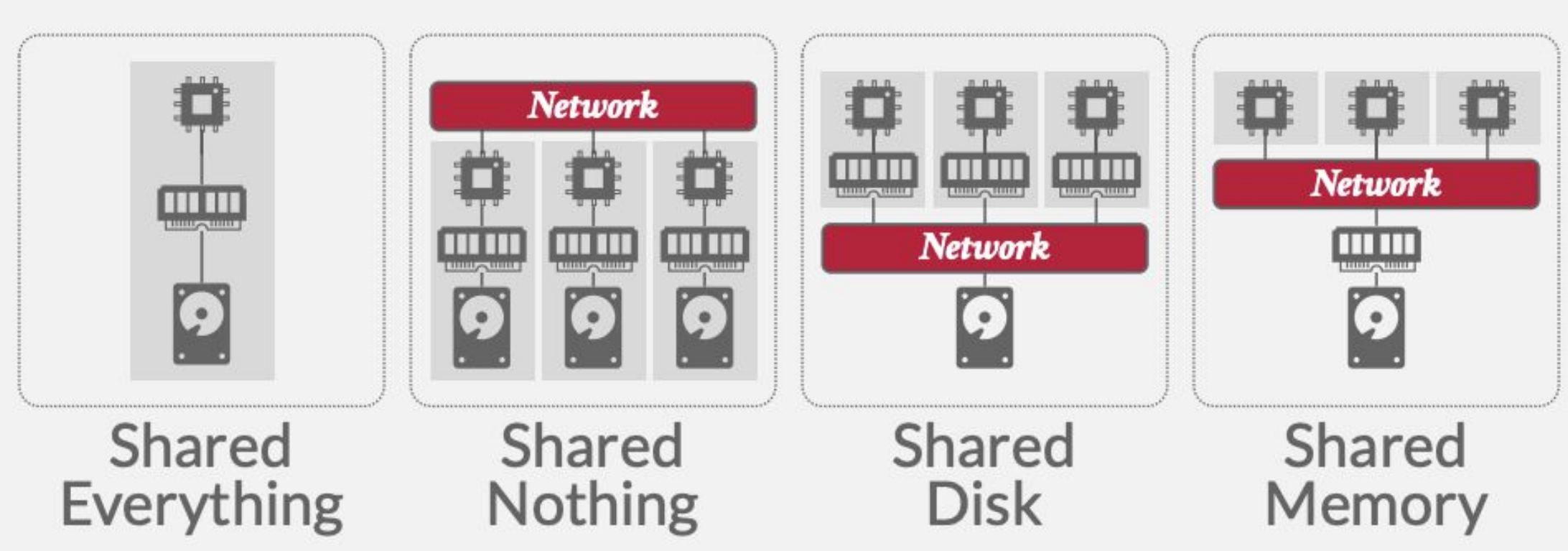
- Nodes can be far from each other.
- Nodes connected using public network.
- Communication cost and problems cannot be ignored.

System Architecture of Distributed Databases

A distributed DBMS system architecture specifies what shared resources are directly accessible to CPUs.

This affects how CPUs coordinate with each other and where they retrieve/store objects in the database.

System Architecture of Distributed Databases



Shared Nothing

Each DBMS node has its own CPU, memory, and local disk.

Nodes only communicate with each other via network.

- Better performance & efficiency.
- Harder to scale capacity.
- Harder to ensure consistency.

Shared Disk

Nodes access a single logical disk via an interconnect, but each have their own private memories.

- Scale execution layer independently from the storage layer.
- Nodes can still use direct attached storage as a slower/larger cache.
- This architecture facilitates data lakes and serverless systems.

Shared Memory

Nodes access a common memory address space via a fast interconnect.

- Each node has a global view of all the inmemory data structures.
- Can still use local memory / disk for intermediate results.

Early Distributed Database Systems

MUFFIN – UC Berkeley (1979)

SDD-1 – CCA (1979)

System R* – IBM Research (1984)

Gamma – Univ. of Wisconsin (1986)

NonStop SQL – Tandem (1987)

Data Transparency

Applications should not be required to know where data is physically located in a distributed DBMS.

- Any query that run on a single-node DBMS should produce the same result on a distributed DBMS.

In practice, developers need to be aware of the communication costs of queries to avoid excessively "expensive" data movement.

Database Partitioning

Split database across multiple resources:

- Disks, nodes, processors.
- Often called "sharding" in NoSQL systems.

The DBMS executes query fragments on each partition and then combines the results to produce a single answer.

The DBMS can partition a database physically (shared nothing) or logically (shared disk).

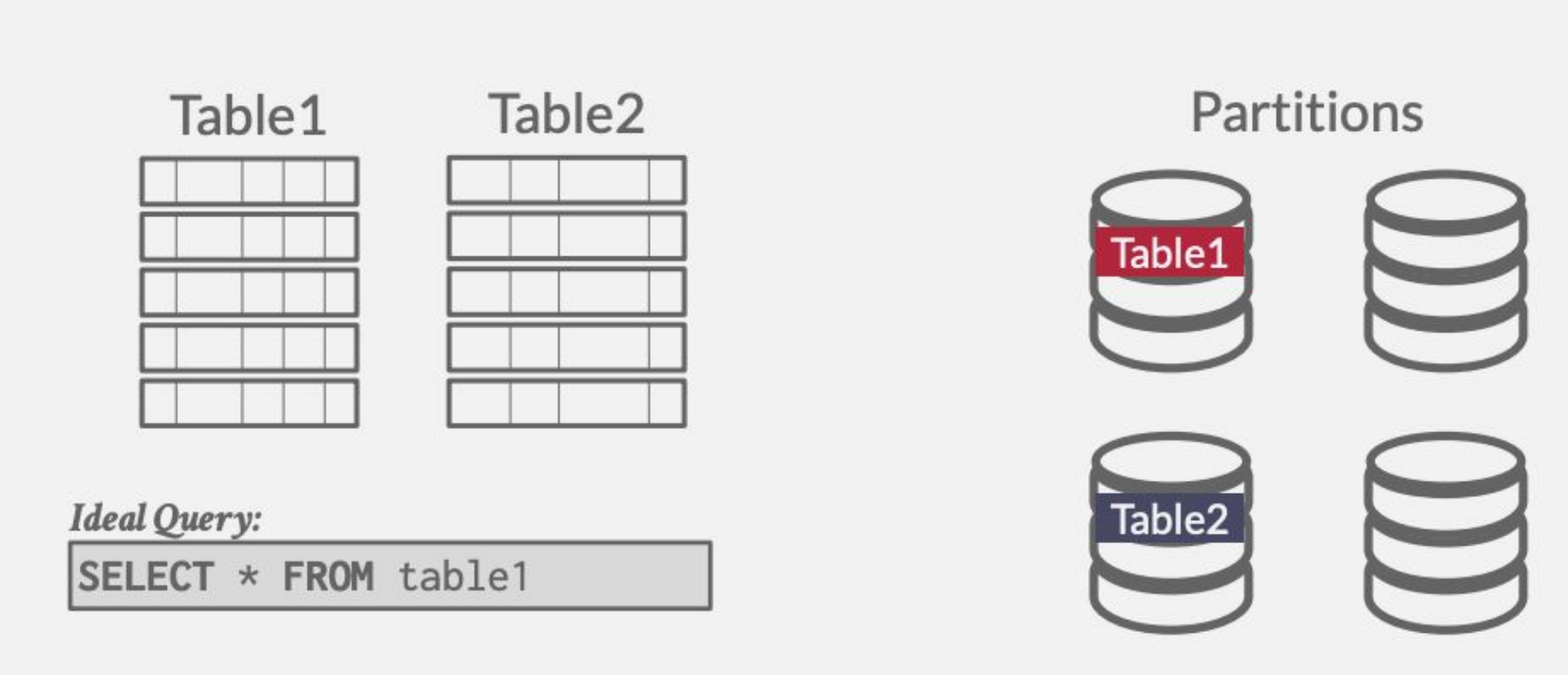
Naive Table Partitioning

Assign an entire table to a single node.

Assumes that each node has enough storage space for an entire table.

Ideal if queries never join data across tables stored on different nodes and access patterns are uniform.

Naive Table Partitioning



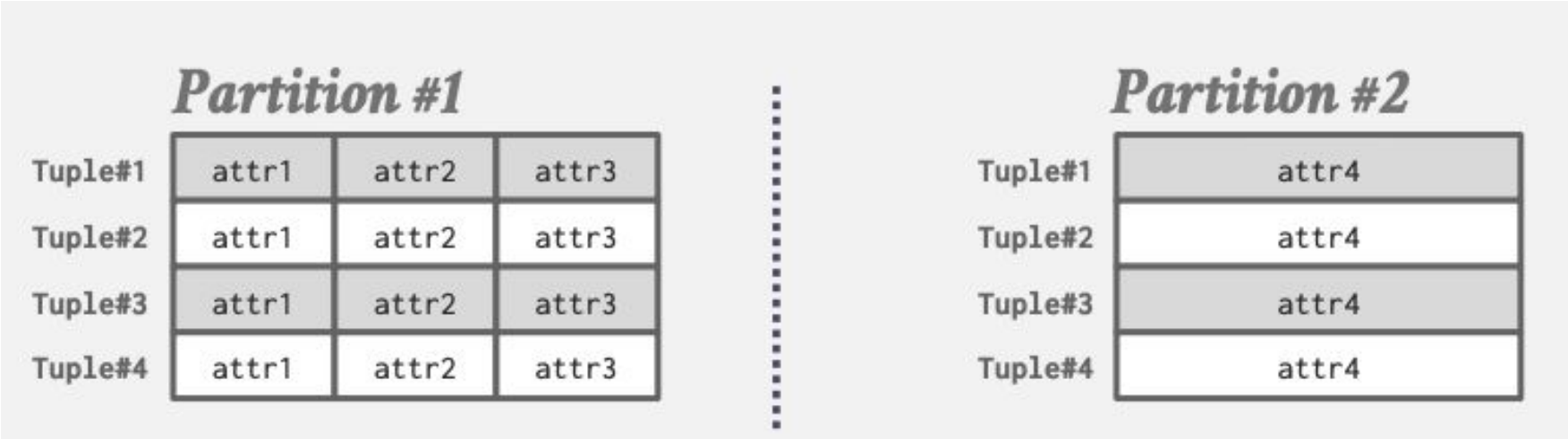
Vertical Partitioning

Split a table's attributes into separate partitions.

Must store tuple information to reconstruct the original record.

Vertical Partitioning

```
CREATE TABLE foo (
  attr1 INT,
  attr2 INT,
  attr3 INT,
  attr4 TEXT
);
```



Horizontal Partitioning

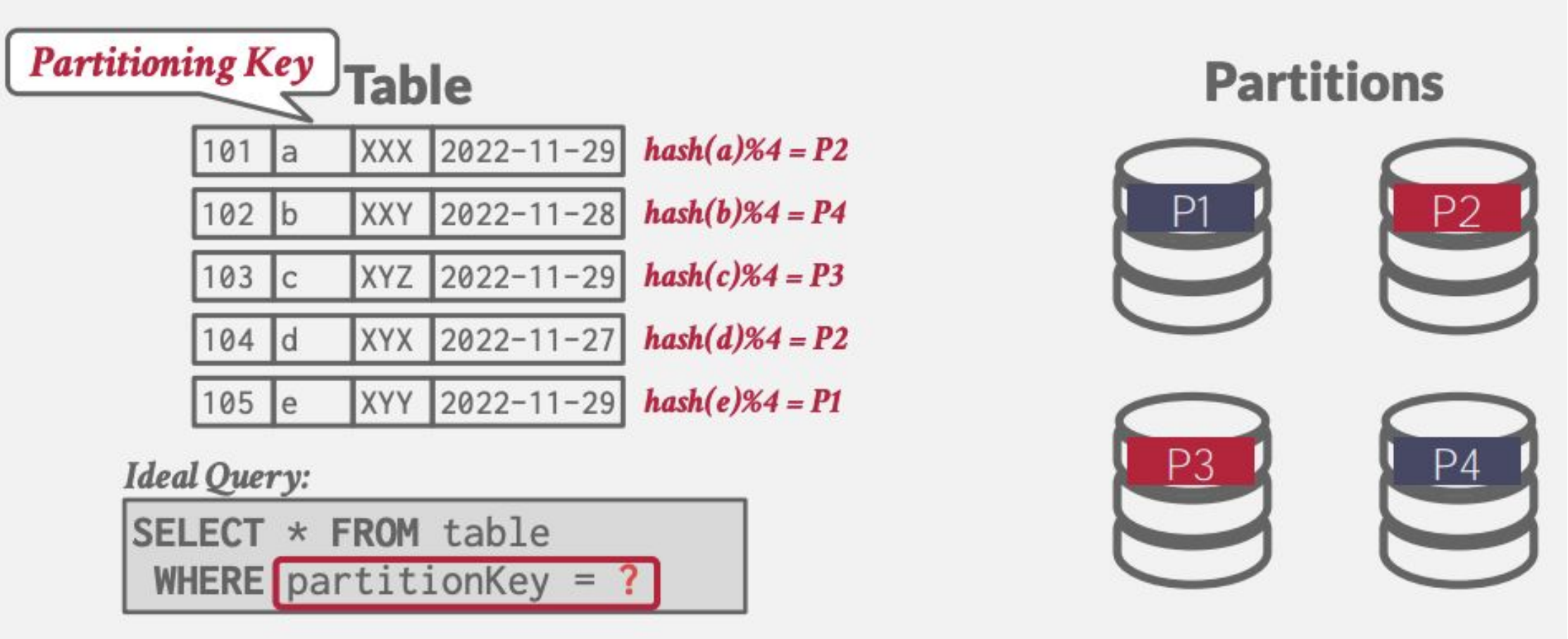
Split a table's tuples into disjoint subsets based on some partitioning key and scheme.

- Choose column(s) that divides the database equally in terms of size, load, or usage.

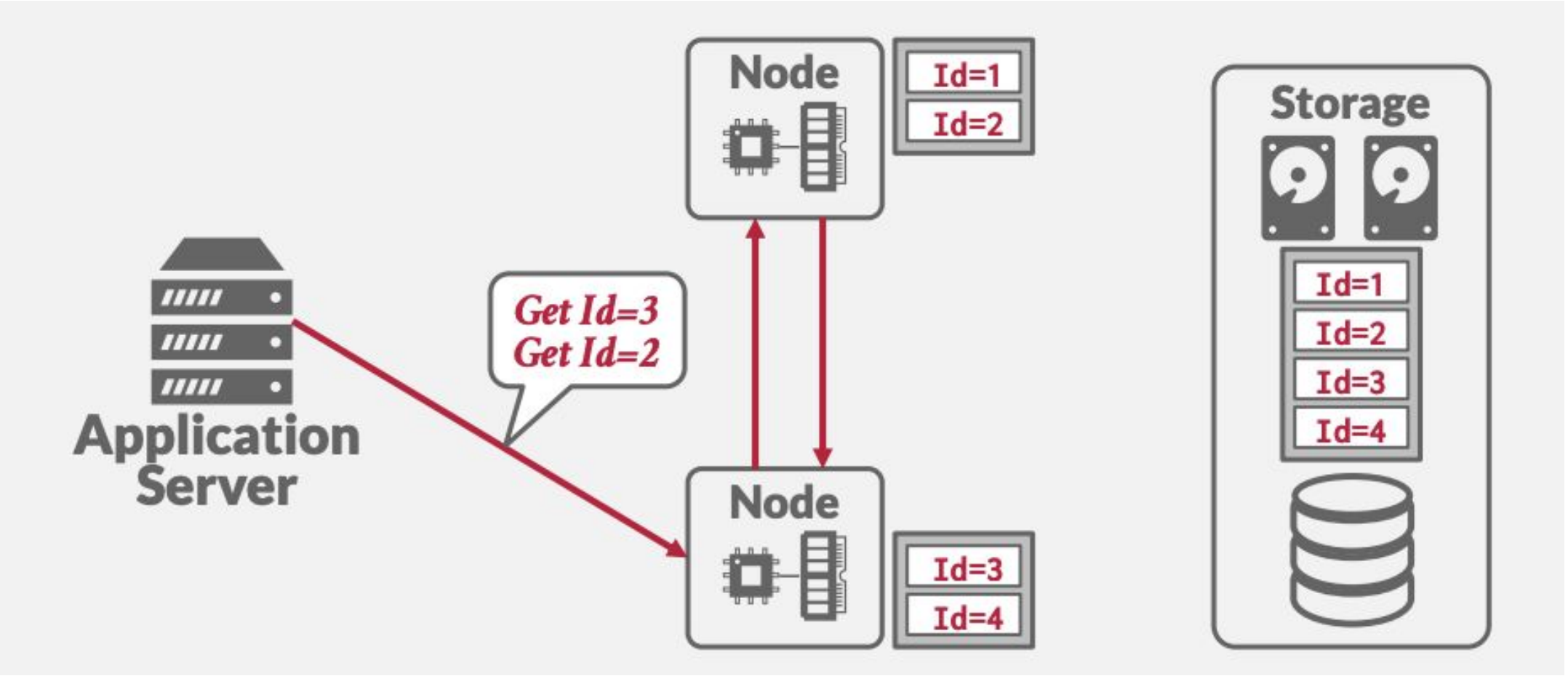
Partitioning Schemes:

- Hashing
- Ranges
- Predicates

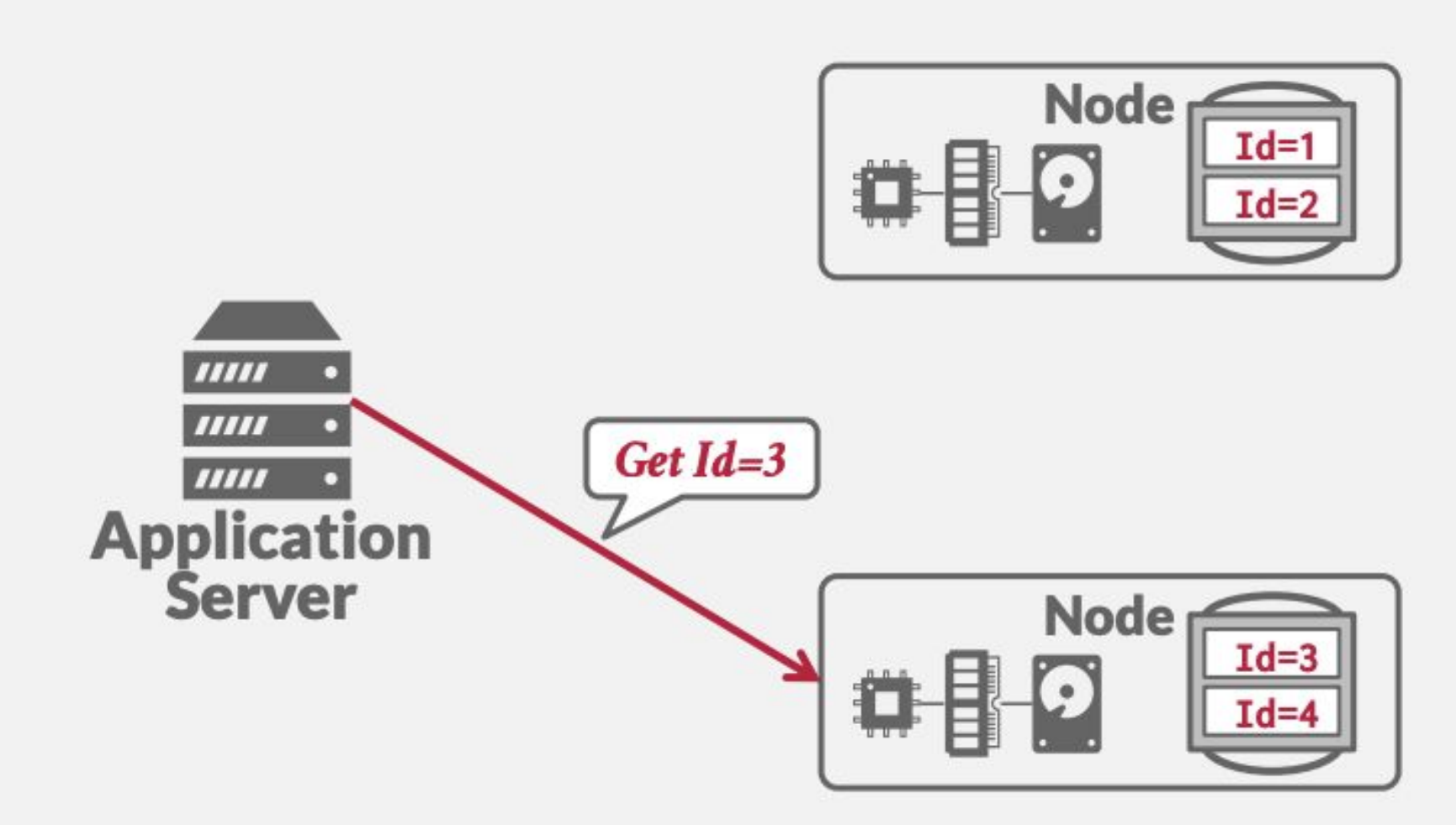
Horizontal Partitioning



Logical Partitioning



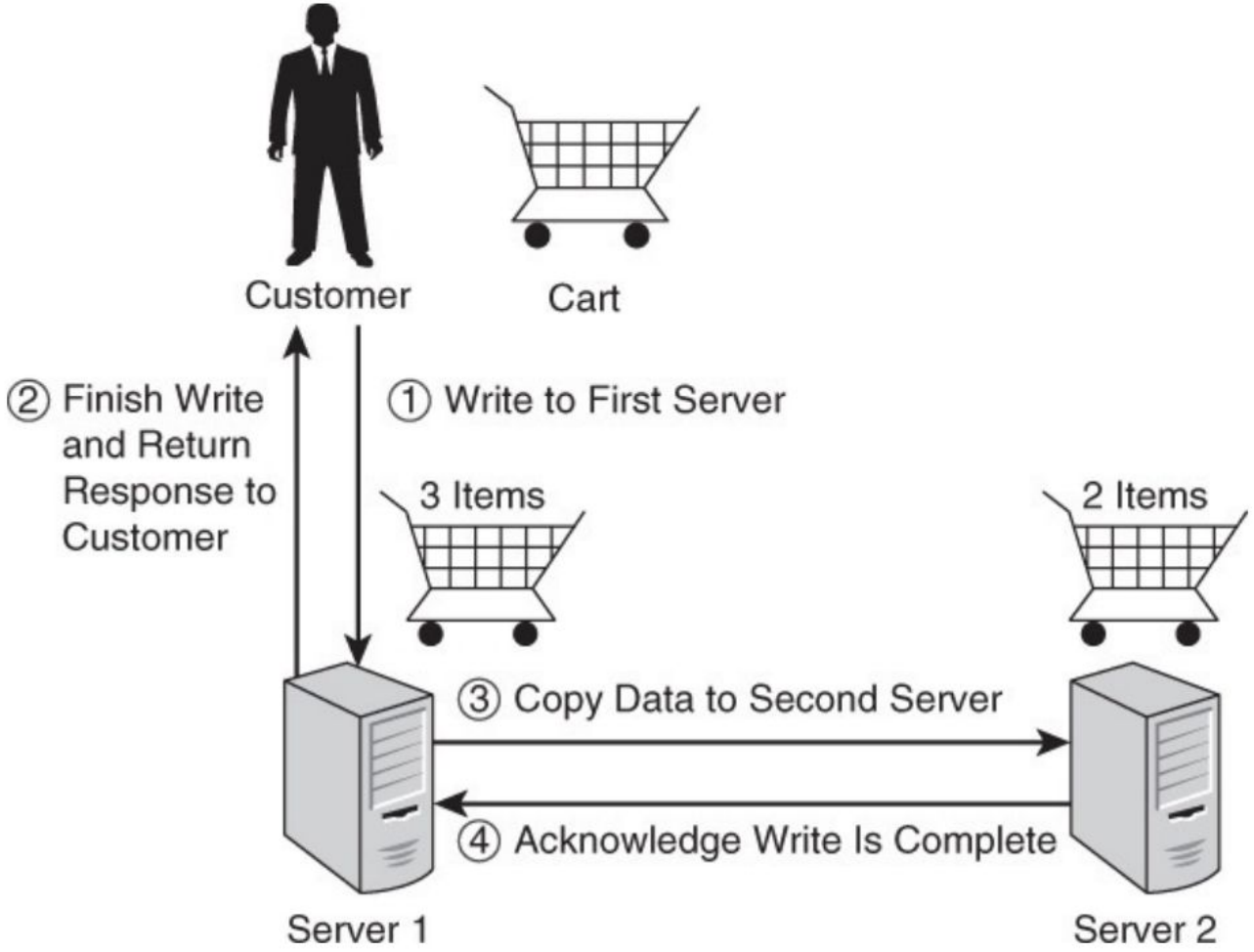
Physical Partitioning



Availability and Consistency in Distributed Databases

When two database servers/computers must keep consistent copies of data, they incur longer times to complete a transaction.

Availability and Consistency in Distributed Databases



Balancing Response Times, Consistency and Durability

NoSQL databases often implement **eventual consistency**; that is, there might be a period of time where copies of data have different values, but eventually all copies will have the same value.

Balancing Response Times, Consistency and Durability

NoSQL databases often use the concept of quorums when working with reads and writes.

A **quorum** is the number of servers that must respond to a read or write operation for the operation to be considered complete.

Balancing Response Times, Consistency and Durability

When a read is performed, the NoSQL database reads data from, potentially, multiple servers. Most of the time, all of the servers will have consistent data.

However, while the database copies data from one of the servers to the other servers storing replicas, the replica servers may have inconsistent data.

How do we make sure that responses to any operation is correct?

Balancing Response Times, Consistency and Durability

How do we make sure that responses to any operation is correct?

- For read operation: query all servers storing that data.

Balancing Response Times, Consistency and Durability

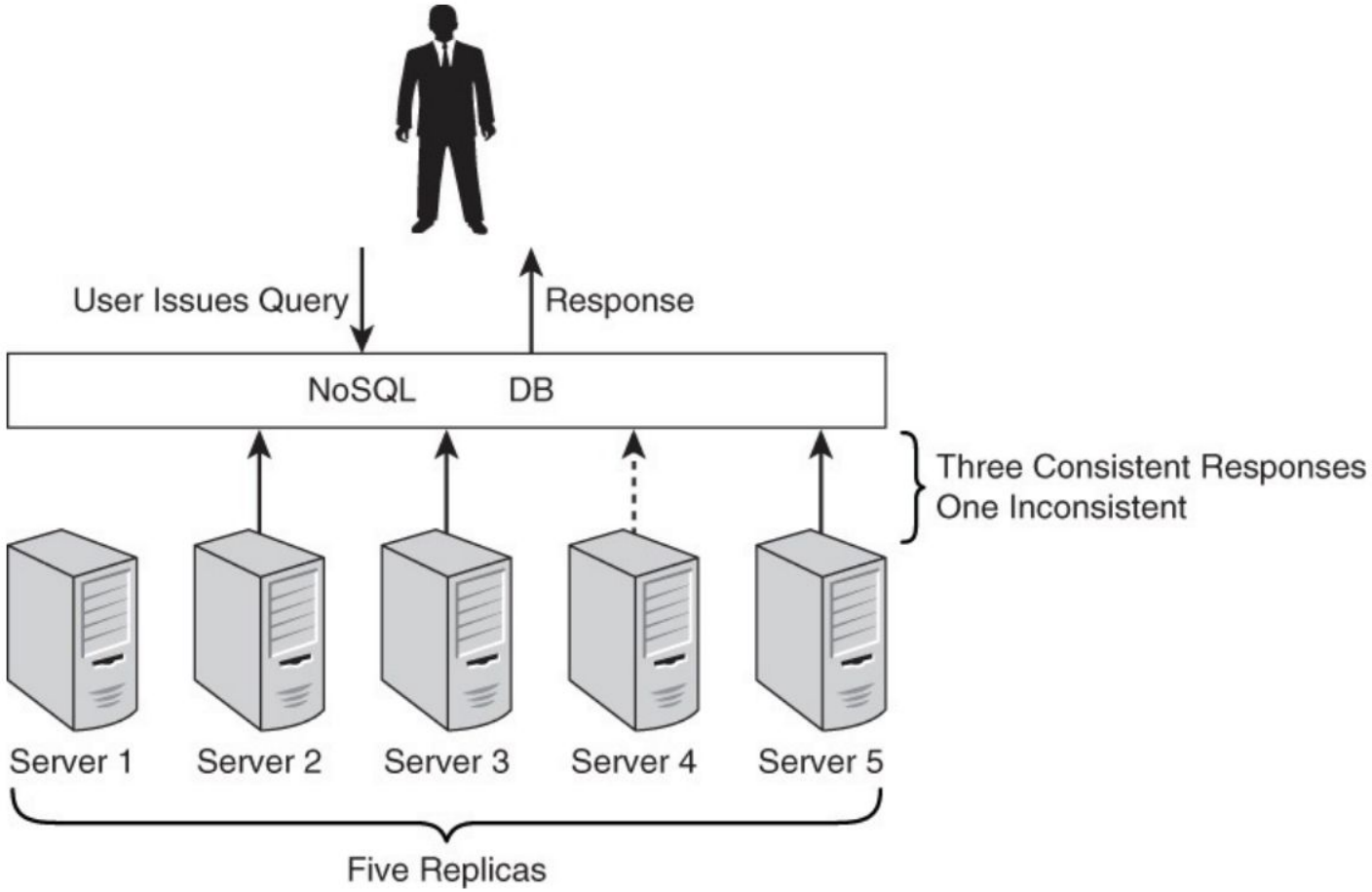


Figure 2.7 *NoSQL databases can mitigate the risk of inconsistent data by having servers vote on the correct response to a query.*

Consistency, Availability, and Partitioning: The CAP Theorem

The **CAP theorem**, also known as Brewer's theorem after the computer scientist who introduced it, states that **distributed databases cannot have consistency (C), availability (A), and partition protection (P) all at the same time.**

Consistency, Availability, and Partitioning: The CAP Theorem

Consistency refers to maintaining consistent copies of data on different servers.

Availability refers to providing a response to any query.

Partition protection means if a network that connects two or more database servers fails, the servers will still be available with consistent data.

Consistency, Availability, and Partitioning: The CAP Theorem

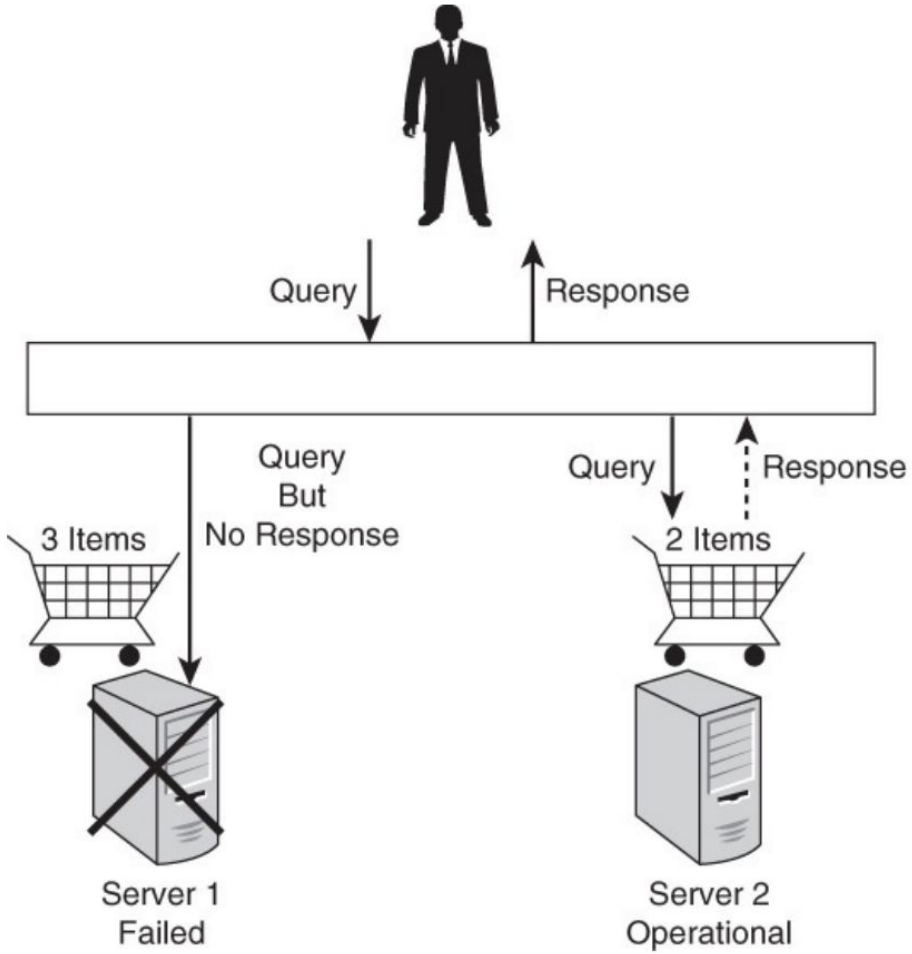


Figure 2.8 *Data can be available but not consistent.*

Consistency, Availability, and Partitioning: The CAP Theorem

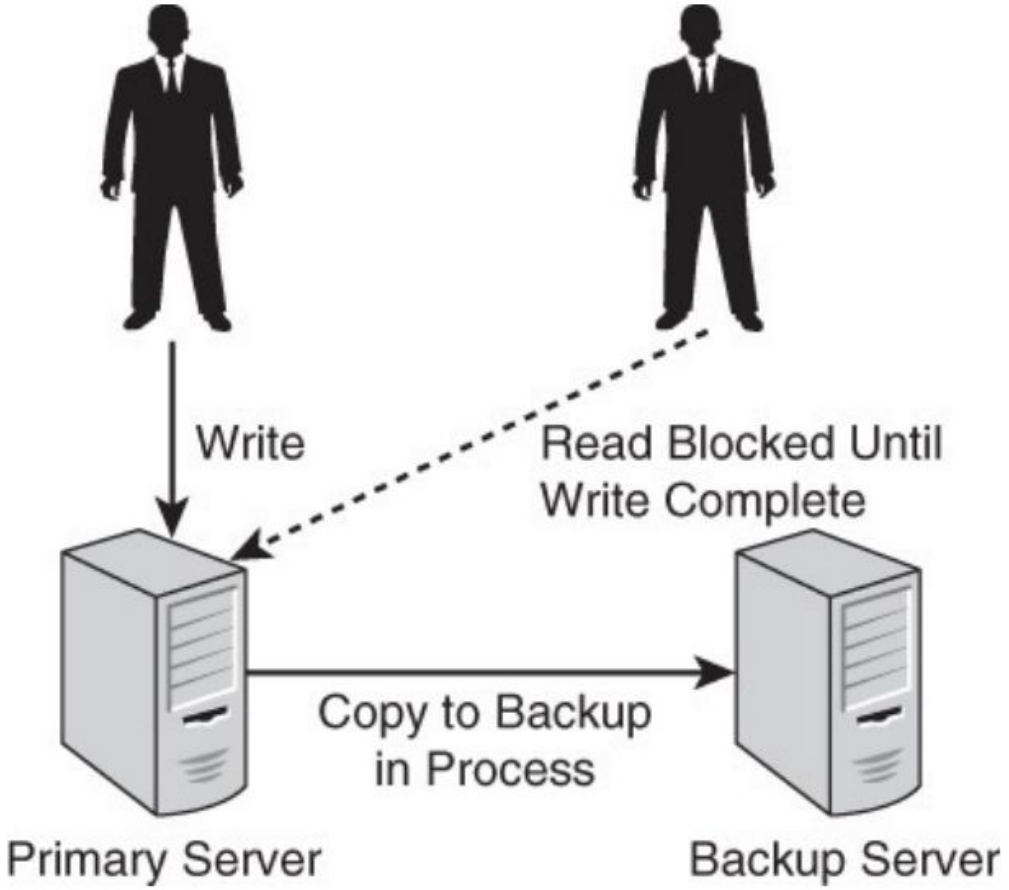


Figure 2.9 *Data can be consistent but not available.*

Consistency, Availability, and Partitioning: The CAP Theorem

Partition Protection: Refers to managing situations where servers are unable to communicate due to network failures.

In data management, partitioning can refer to various concepts. In the context of the CAP theorem, it relates to the inability to send messages between database servers.

If database servers running the same distributed database are partitioned due to network failure, it presents a dilemma. Allowing both servers to respond to queries preserves availability but risks inconsistency. Disabling one ensures consistency but sacrifices availability.

Consistency, Availability, and Partitioning: The CAP Theorem

- Network Partitions are uncommon.
- Most of the time database designers of NoSQL databases are conflicted with availability-consistency trade-offs.
- Designers of NoSQL database management systems have to determine how to balance varying needs for consistency, availability, and partitioning protection.
- NoSQL database designers can provide configuration mechanisms that allow users of the database to specify their preferred settings rather than making a single choice for all users of the database management system.

Consistency, Availability, and Partitioning: The CAP Theorem

- Application designers could make use of NoSQL database configuration options to make the availability-consistency trade-off decision at fine-grained levels, such as based on different types of data in the database.
- The only limitation is the configuration options provided in the NoSQL database management system used by the application.

ACID: Atomicity, Consistency, Isolation, and Durability

- Properties common to relational database management systems.

ACID: Atomicity, Consistency, Isolation, and Durability

Atomicity- describes a unit that cannot be further divided.

Consistency - In relational databases, this is known as strict consistency. In other words, a transaction does not leave a database in a state that violates the integrity of data.

Isolation - Isolated transactions are not visible to other users until transactions are complete

Durability - This means that once a transaction or operation is completed, it will remain even in the event of a power loss.

BASE: Basically Available, Soft State, Eventually Consistent

Properties associated with non-relational or NoSQL databases.

BA is for **basically available**. This means that there can be a partial failure in some parts of the distributed system and the rest of the system continues to function.

S is for **soft state**. Usually in computer science, the term soft state means data will expire if it is not refreshed. Here, in NoSQL operations, it refers to the fact that data may eventually be overwritten with more recent data. This property overlaps with the third property of BASE transactions, eventually consistent.

E is for **eventually consistent**. This means that there may be times when the database is in an inconsistent state.

Types of Eventual Consistency

1. **Casual Consistency** ensures that the database reflects the order in which operations were updated.

   For example, if Alice changes a customer's outstanding balance to $1,000 and one minute later Bob changes it to $2,000, all copies of the customer's outstanding balance will be updated to $1,000 before they are updated to $2,000.

Types of Eventual Consistency

2. Read-your-writes consistency means that once you have updated a record, all of your reads of that record will return the updated value. You would never retrieve a value inconsistent with the value you had written.

For example, let's say Alice updates a customer's outstanding balance to $1,500. The update is written to one server and the replication process begins updating other copies. During the replication process, Alice queries the customer's balance. She is guaranteed to see $1,500 when the database supports read your-writes consistency.

Types of Eventual Consistency

3. Session Consistency ensures read-your-writes consistency during a session.

A session is like a conversation between a client and a server or a user and the database. As long as the conversation continues, the database “remembers” all writes you have done during the conversation. If the session ends and you start another session with the same server, there is no guarantee it will “remember” the writes you made in the previous session. A session may end if you log off an application using the database or if you do not issue commands to the database for so long that the database assumes you no longer need the session and abandons it.

Types of Eventual Consistency

4. Monotonic Read Consistency ensures that if you issue a query and see a result, you will never see an earlier version of the value.

For example, let's assume Alice is yet again updating a customer's outstanding balance. The outstanding balance is currently $1,500. She updates it to $2,500. Bob queries the database for the customer's balance and sees that it is $2,500. If Bob issues the query again, he will see the balance is $2,500 even if all the servers with copies of that customer's outstanding balance have not updated to the latest value.

Types of Eventual Consistency

5. Monotonic Write Consistency ensures that if you were to issue several update commands, they would be executed in the order you issued them.

For example, lets say, x=100 and two write operations arrived in this order -120%X, X+100. Then 120%X will applied before X+100 resulting in 220. If monotonic writes condition is not met then X+100 could be applied before 120%X resulting in 240 which might be incorrect and violate some invariants such as X<225.

Database Paradigms

Next class:

6.5 Key-Value Pair Databases

6.6 Document Databases

6.7 Column Family Databases