Carnegie Mellon University

Distributed OLTP Databases (Part I)

Lecture #22

Database Systems
15-445/15-645
Fall 2018

Andy Pavlo
Computer Science
Carnegie Mellon Univ.

ADMINISTRIVIA

Project #3: TODAY @ 11:59am

Homework #5: Monday Dec 3rd @ 11:59pm

Project #4: Monday Dec 10th @ 11:59pm

Extra Credit: Wednesday Dec 12th @11:59pm

Final Exam: Sunday Dec 16th @ 8:30am

ADMINISTRIVIA

Monday Dec 3rd – VoltDB Lecture

→ Dr. Ethan Zhang (Lead Engineer)

Wednesday Dec 5th – Potpourri + Review

→ Vote for what system you want me to talk about.

→ https://cmudb.io/f18-systems

Wednesday Dec 5th – Extra Credit Check

→ Submit your extra credit assignment early to get feedback from me.

UPCOMING DATABASE EVENTS

Swarm64 Tech Talk

→ Thursday November 29th @ 12pm

→ GHC 8102 ← Different Location!

VoltDB Research Talk

→ Monday December 3rd @ 4:30pm

→ GHC 8102

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.

DISTRIBUTED DBMSs

Use the building blocks that we covered in single-node DBMSs to now support transaction processing and query execution in distributed environments.

→ Optimization & Planning
→ Concurrency Control
→ Logging & Recovery

OLTP VS. OLAP

On-line Transaction Processing (OLTP):

→ Short-lived read/write txns.
→ Small footprint.
→ Repetitive operations.

On-line Analytical Processing (OLAP):

→ Long-running, read-only queries.
→ Complex joins.
→ Exploratory queries.

TODAY'S AGENDA

System Architectures

Design Issues

Partitioning Schemes

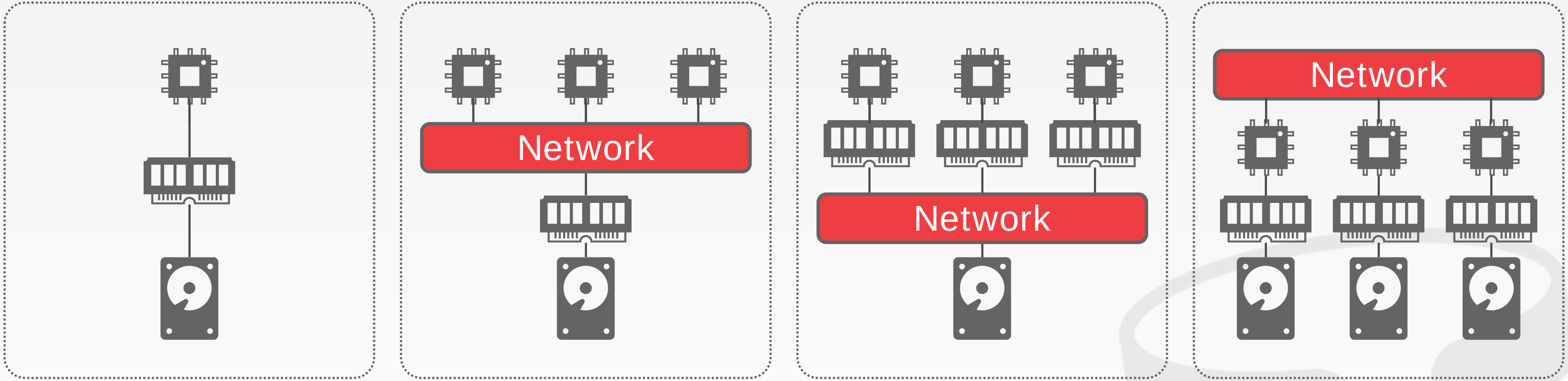
Distributed Concurrency Control

SYSTEM ARCHITECTURE

A DBMS's 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



Shared Everything

Shared Memory

Shared Disk

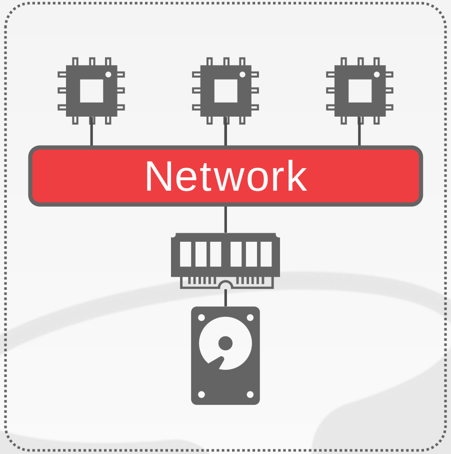
Shared Nothing

SHARED MEMORY

CPUs have access to common memory address space via a fast interconnect.

→ Each processor has a global view of all the in-memory data structures.

→ Each DBMS instance on a processor has to "know" about the other instances.

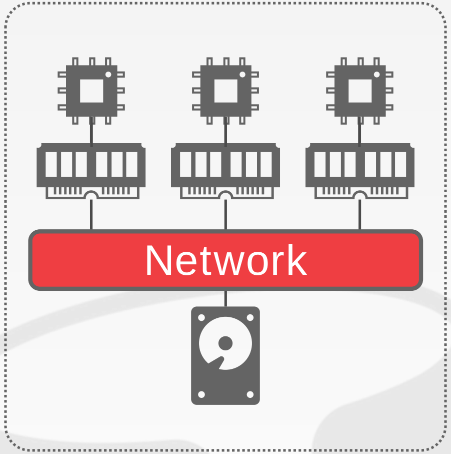


SHARED DISK

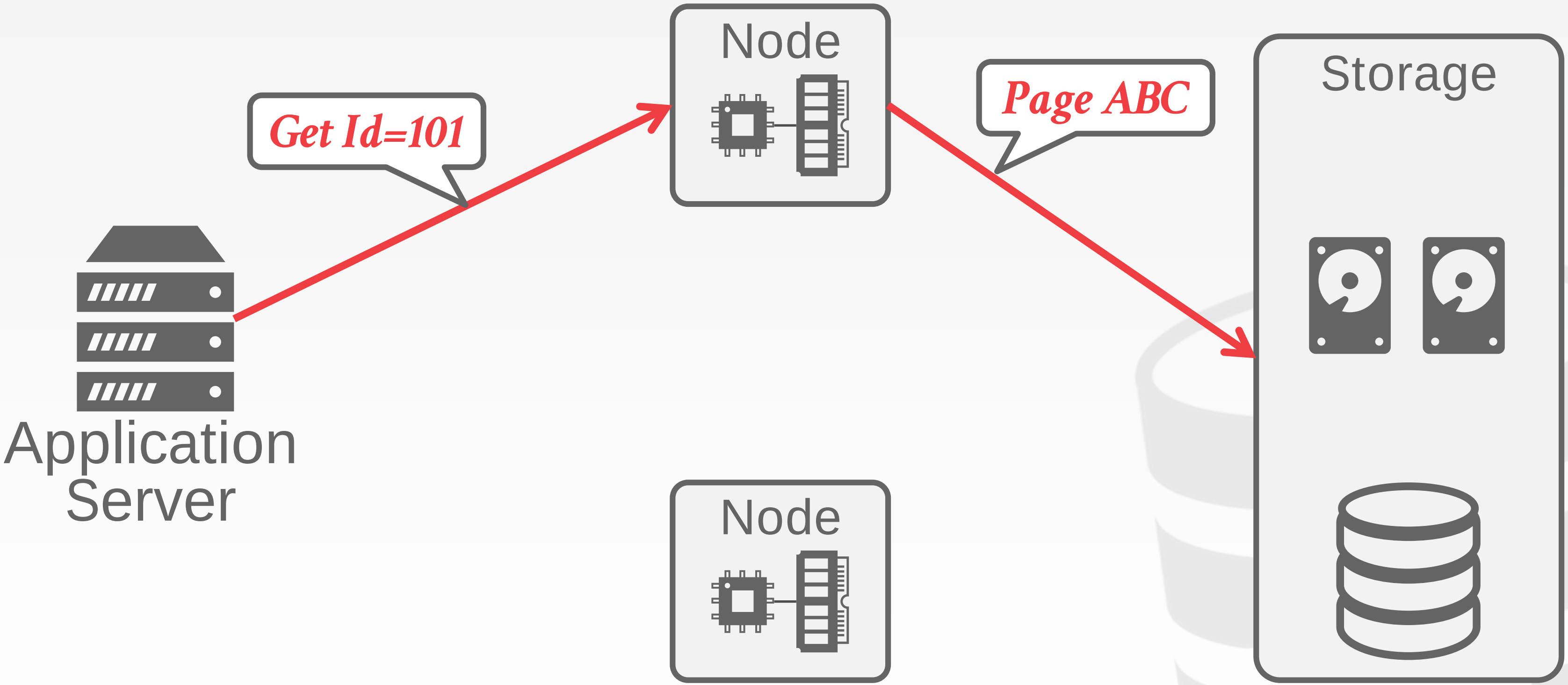
All CPUs can access a single logical disk directly via an interconnect but each have their own private memories.

→ Can scale execution layer independently from the storage layer.

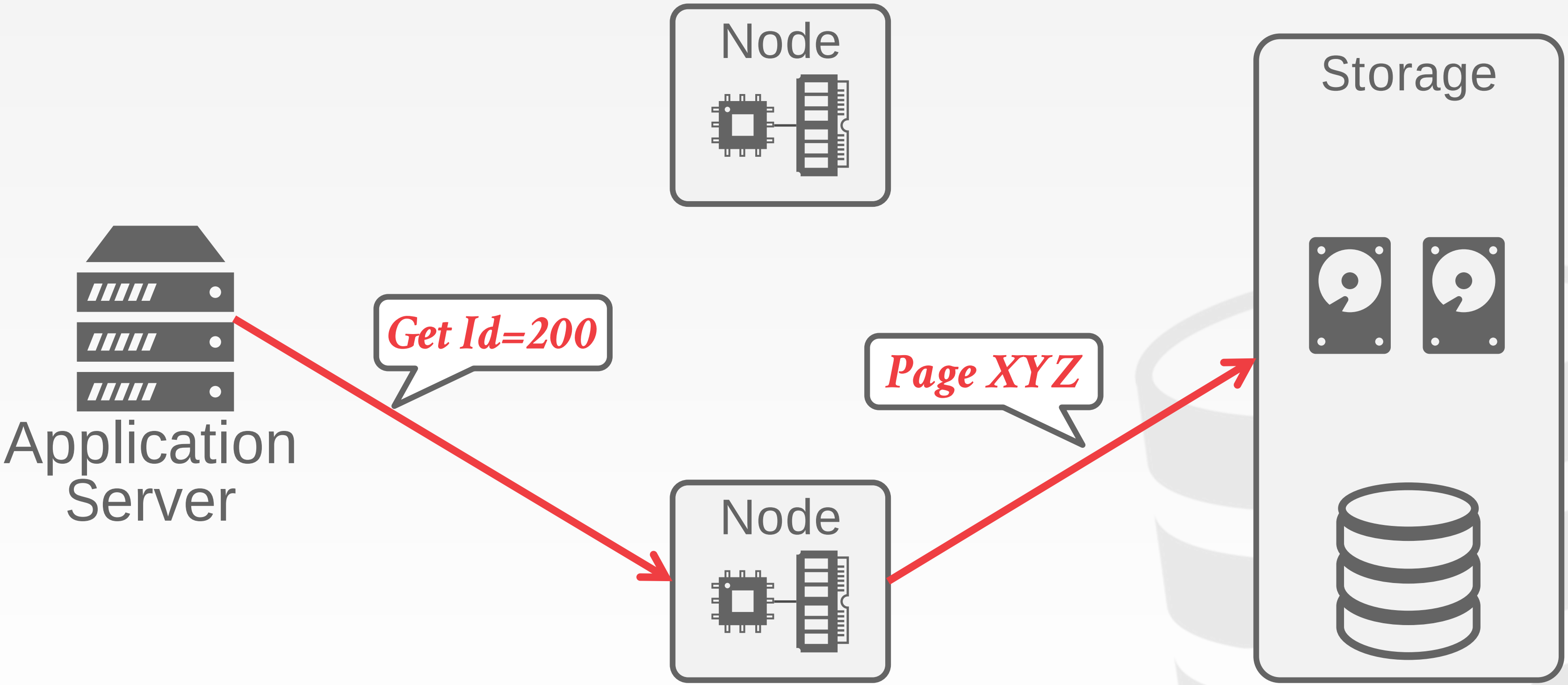
→ Have to send messages between CPUs to learn about their current state.



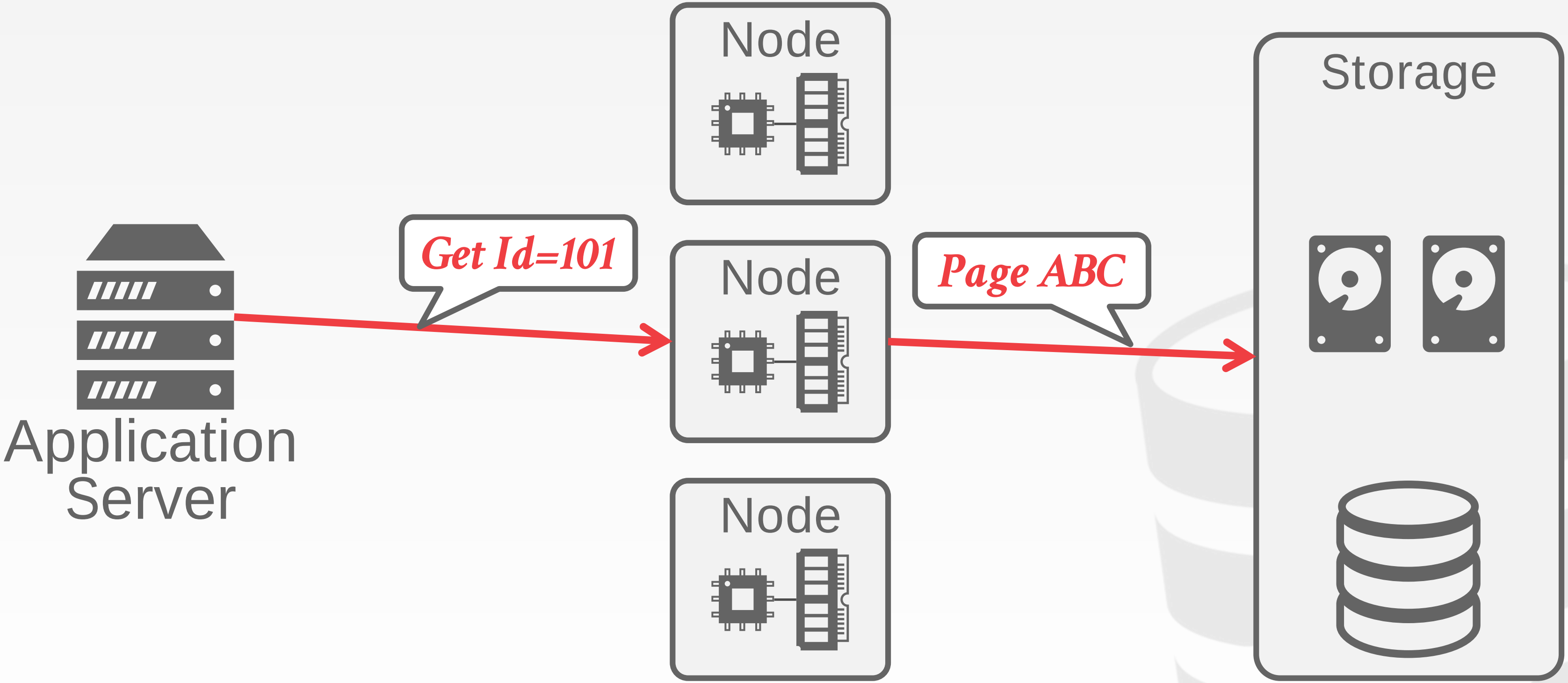
SHARED DISK EXAMPLE



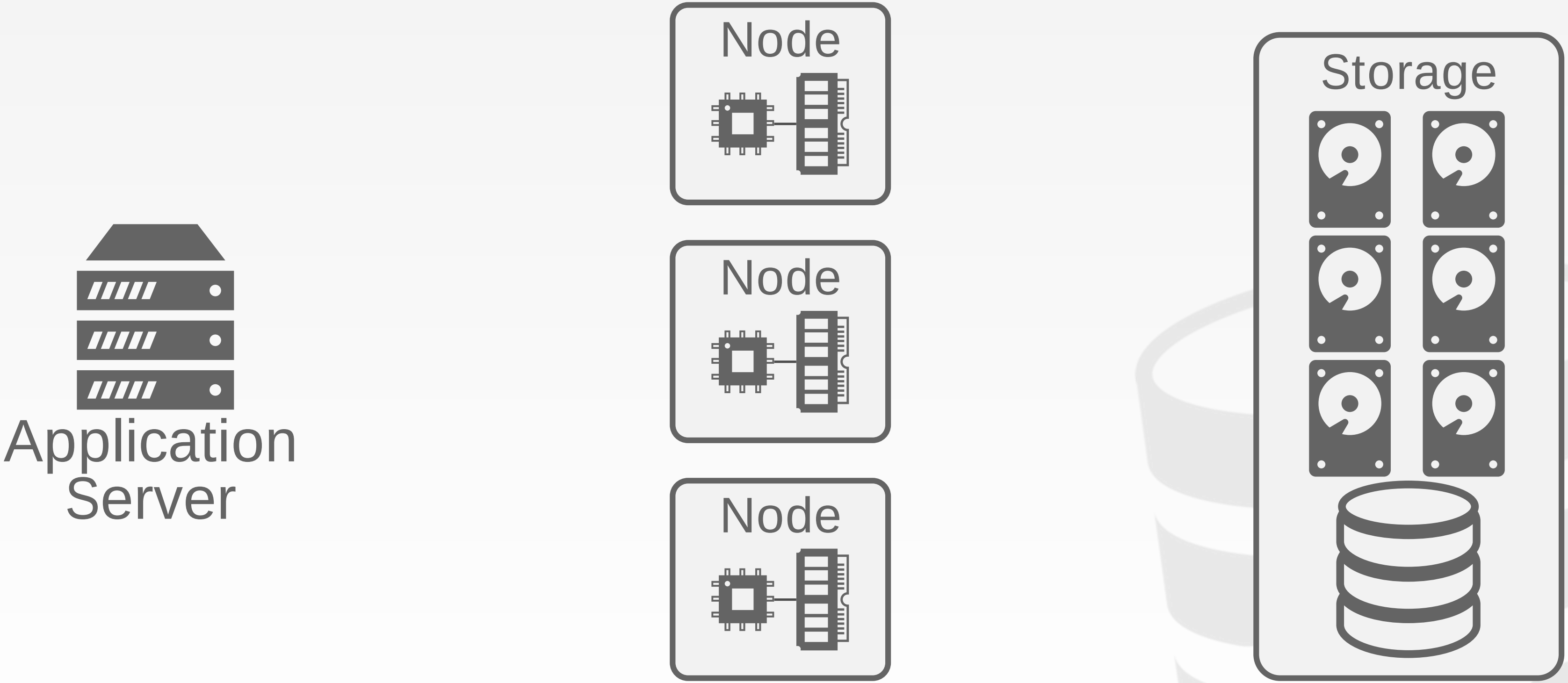
SHARED DISK EXAMPLE



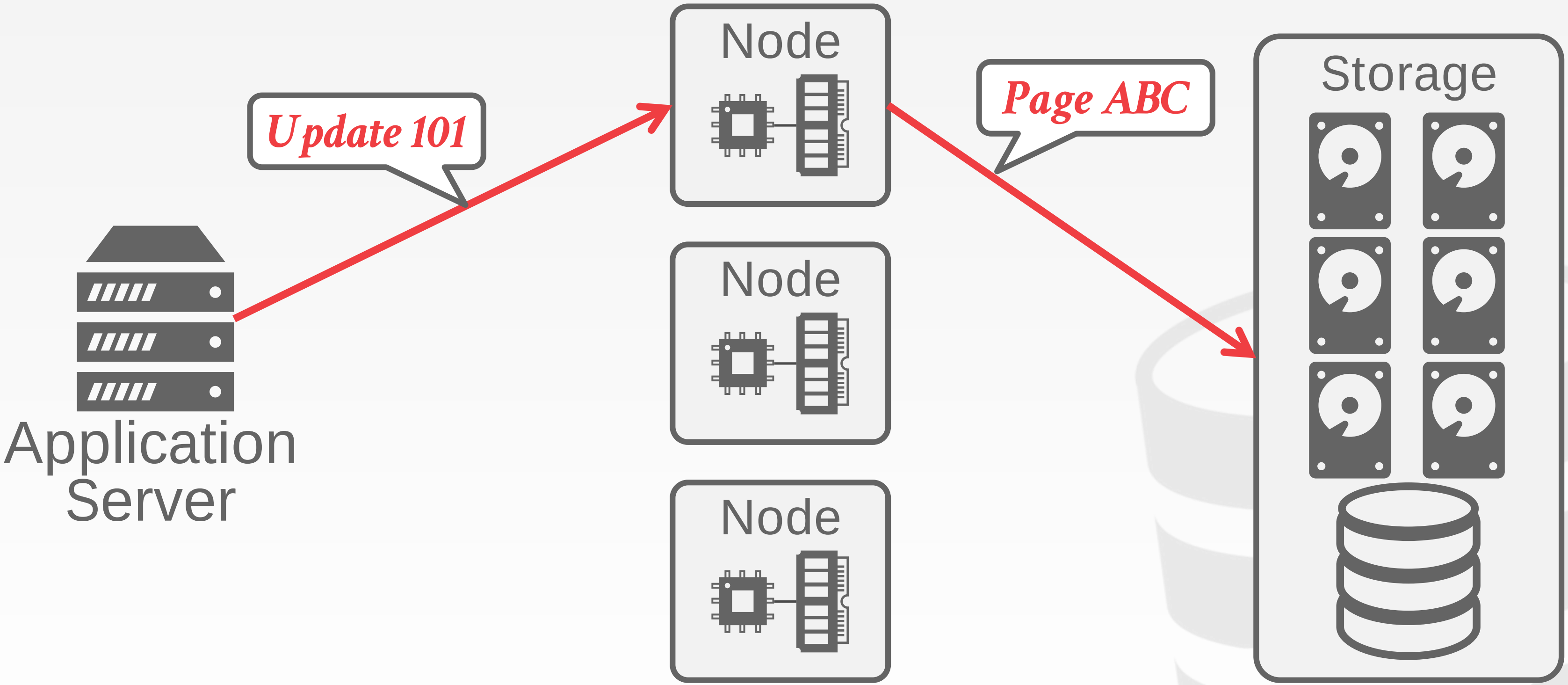
SHARED DISK EXAMPLE



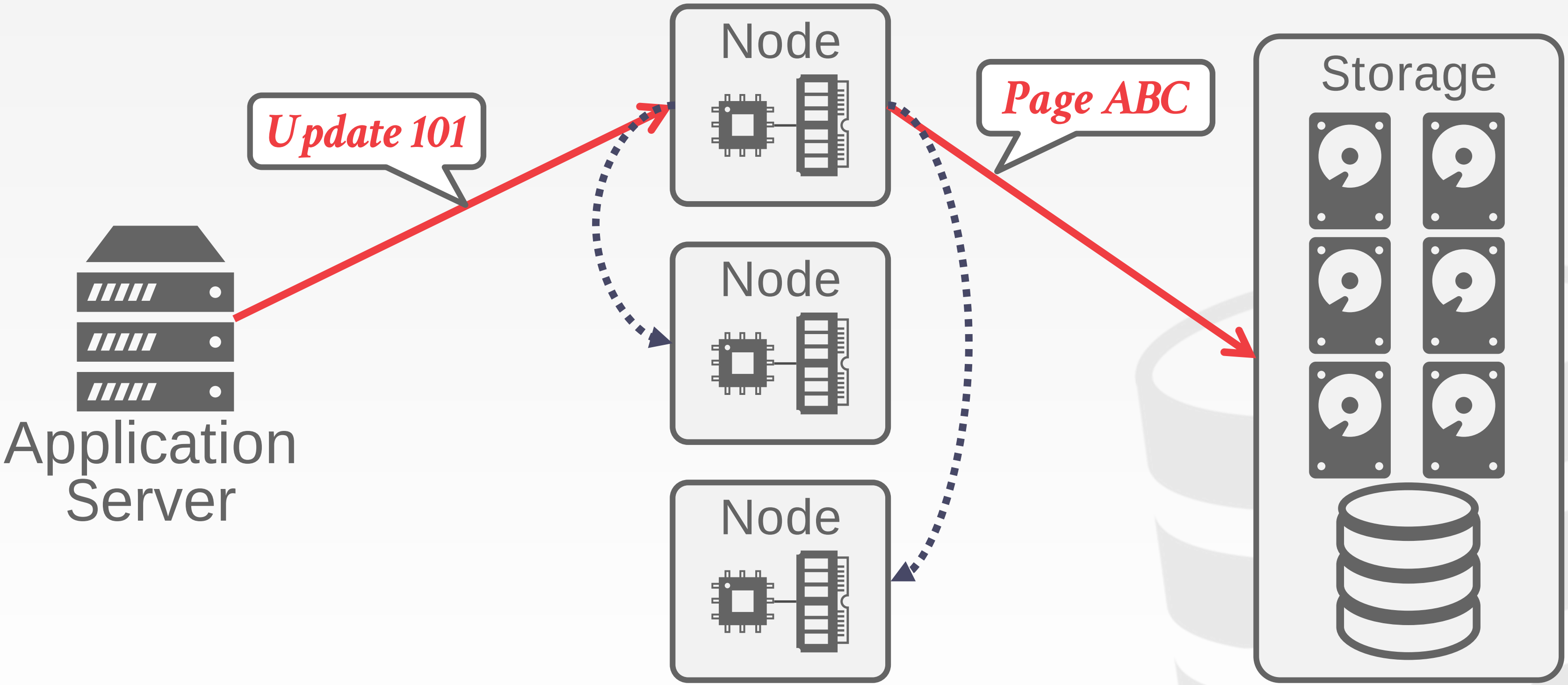
SHARED DISK EXAMPLE



SHARED DISK EXAMPLE



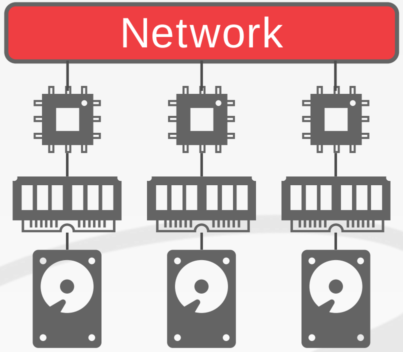
SHARED DISK EXAMPLE



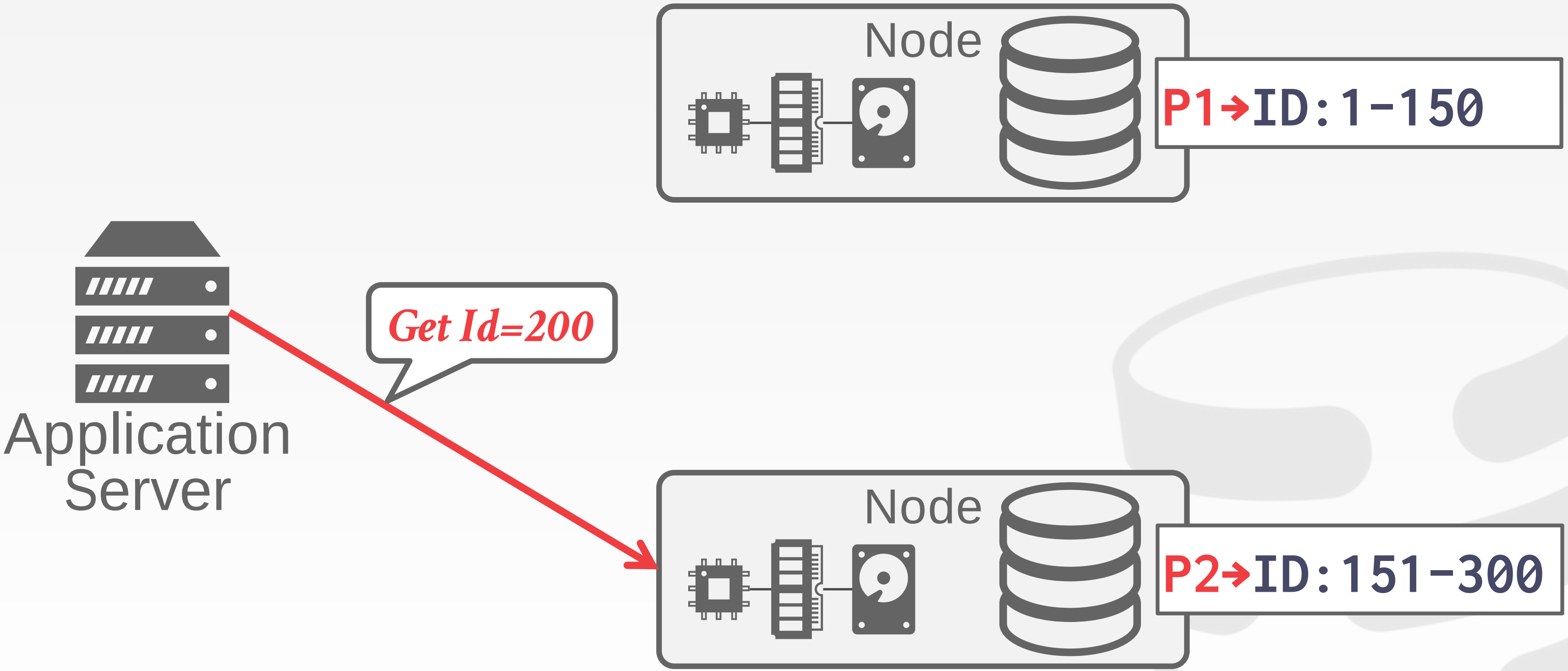
SHARED NOTHING

Each DBMS instance has its own CPU, memory, and disk.

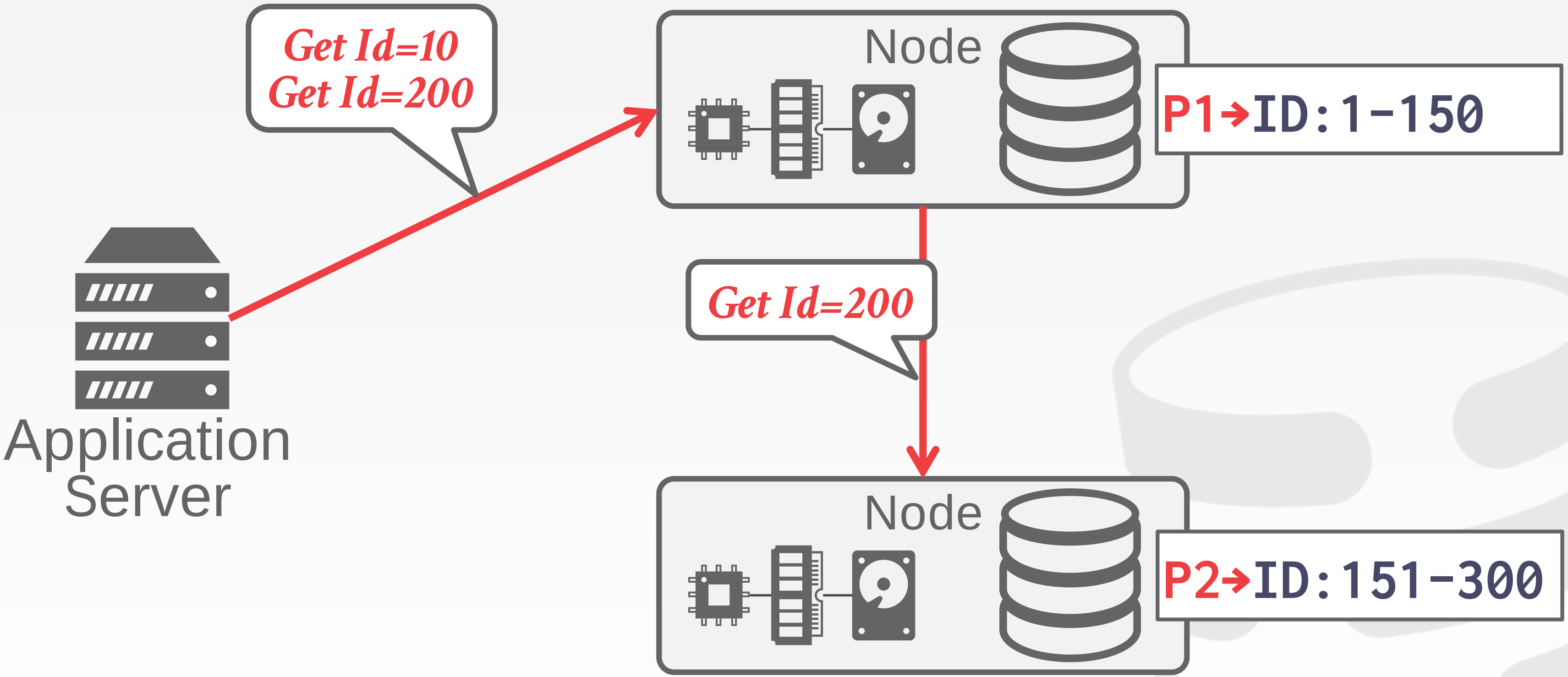
Nodes only communicate with each other via network.
→ Easy to increase capacity.
→ Hard to ensure consistency.



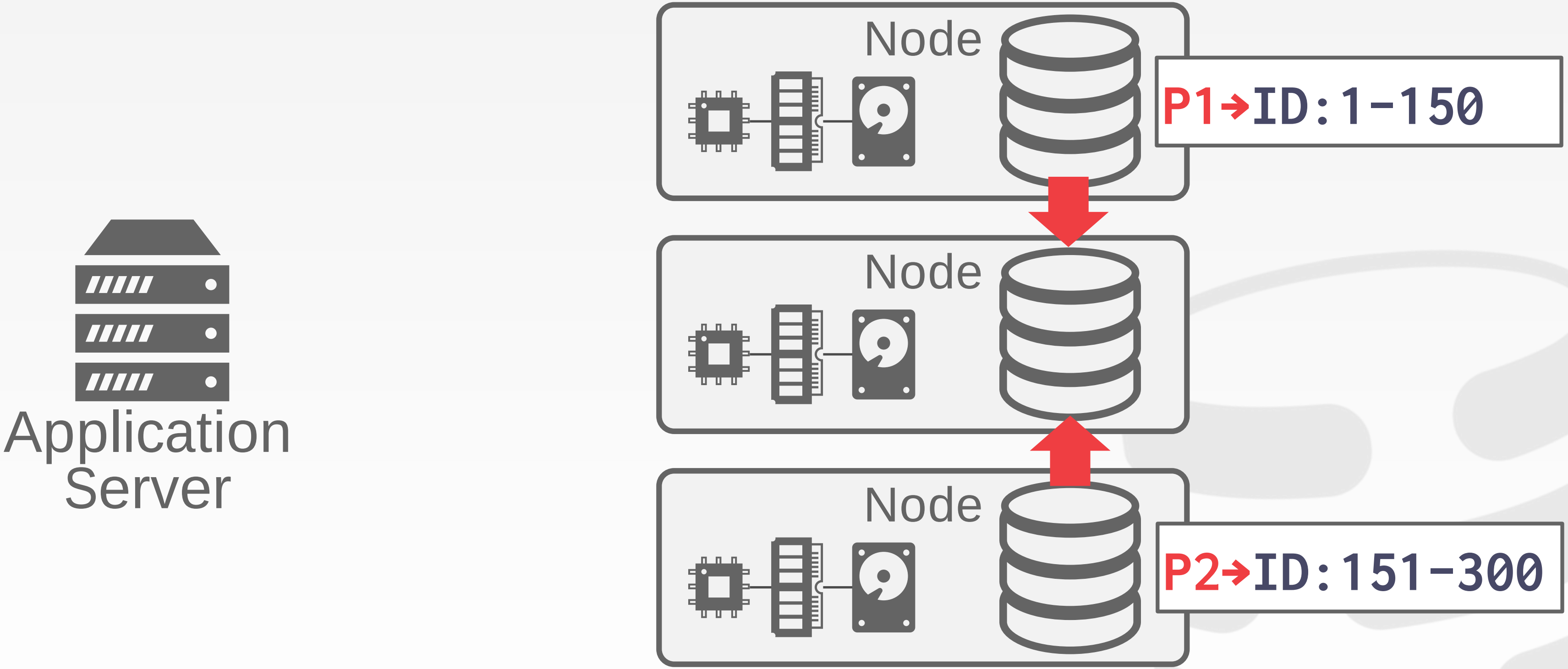
SHARED NOTHING EXAMPLE



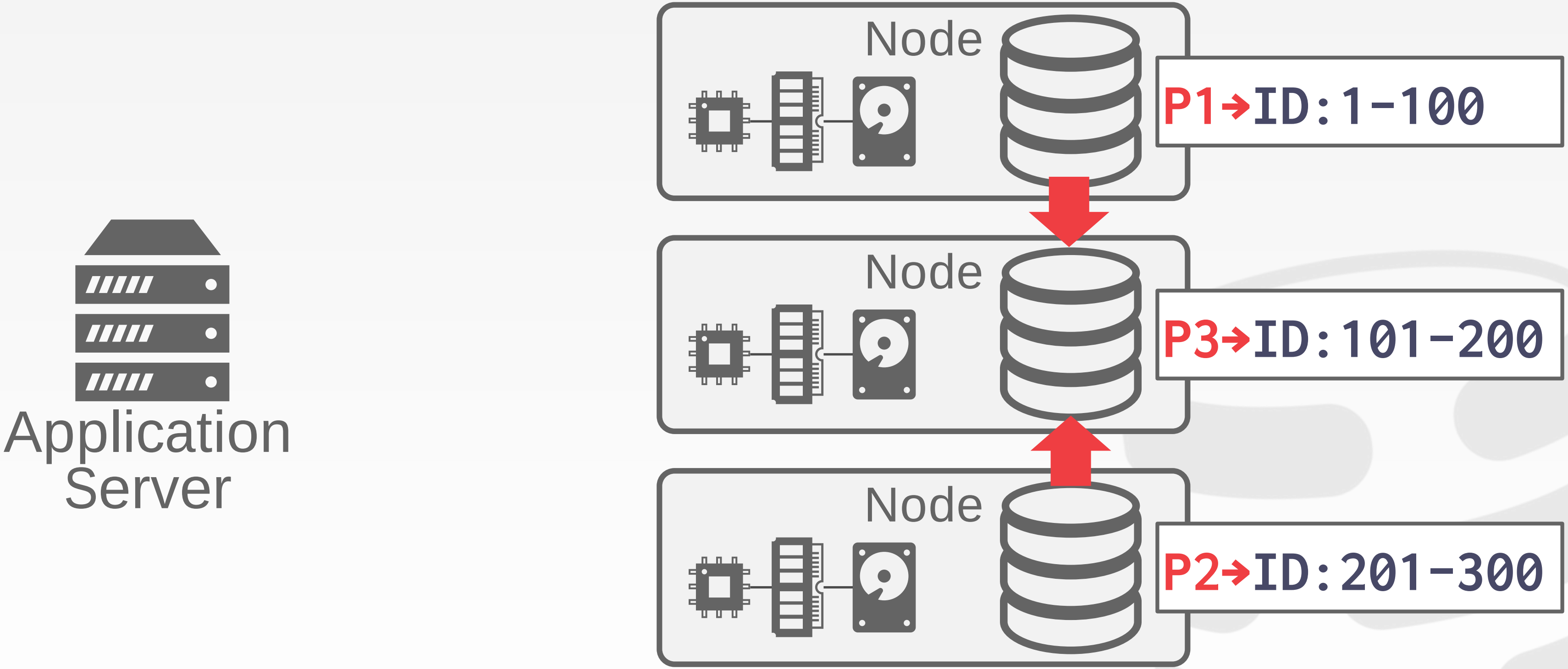
SHARED NOTHING EXAMPLE



SHARED NOTHING EXAMPLE



SHARED NOTHING EXAMPLE



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)


Stonebraker


Bernstein


Mohan


DeWitt


Gray

DESIGN ISSUES

How does the application find data?

How to execute queries on distributed data?

→ Push query to data.

→ Pull data to query.

How does the DBMS ensure correctness?

HOMOGENOUS VS. HETEROGENOUS

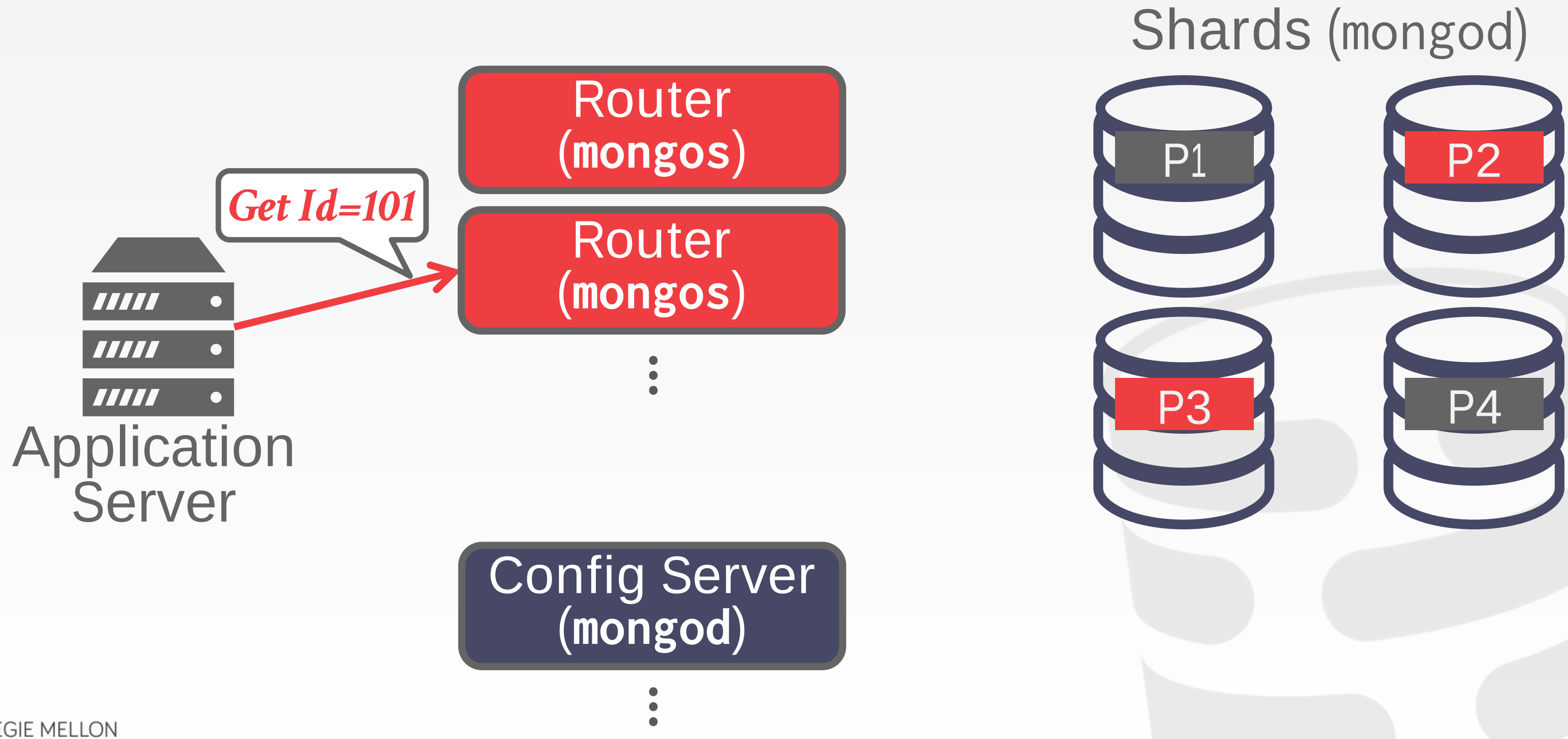
Approach #1: Homogenous Nodes

→ Every node in the cluster can perform the same set of tasks (albeit on potentially different partitions of data).
→ Makes provisioning and failover "easier".

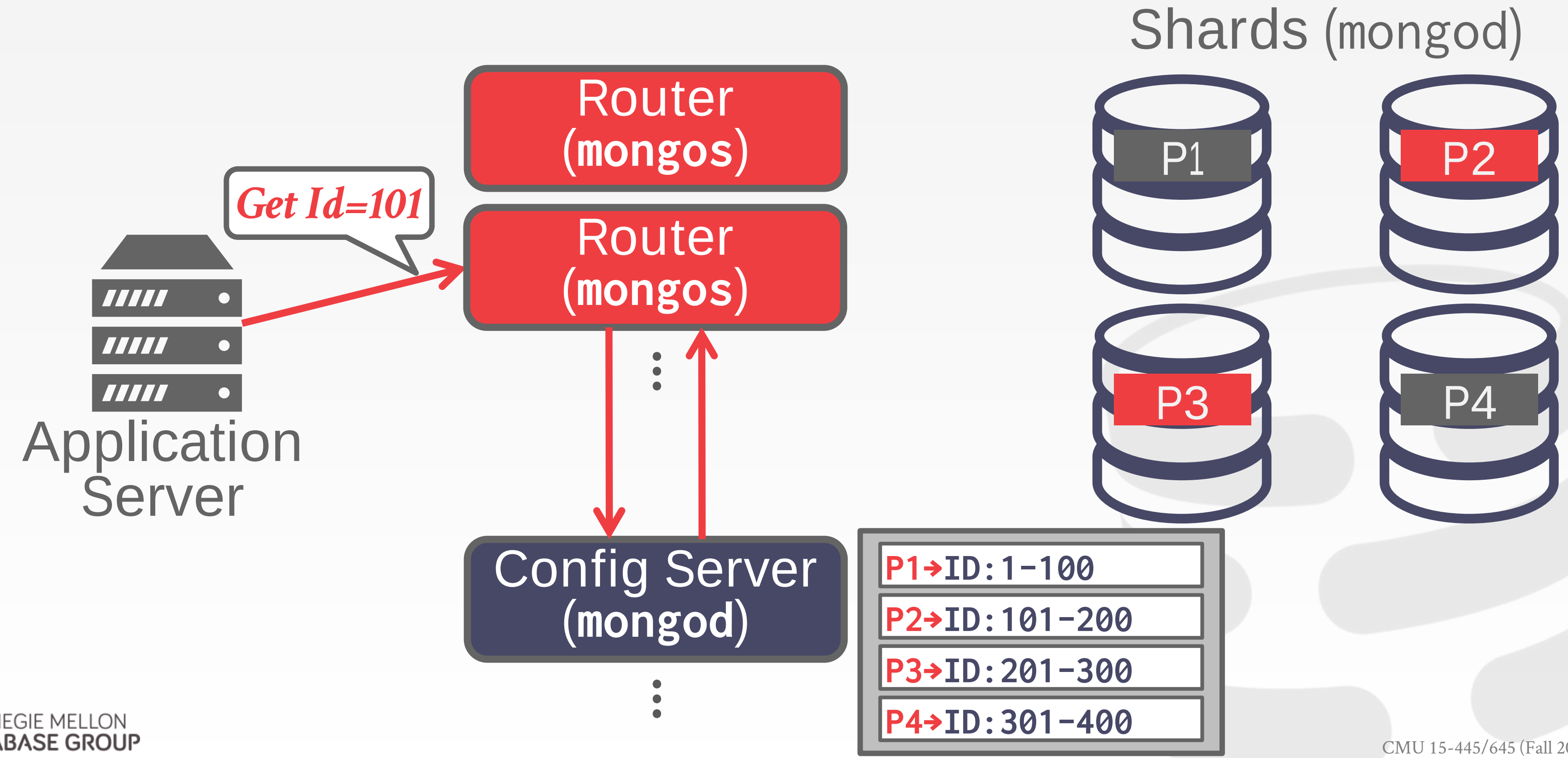
Approach #2: Heterogenous Nodes

→ Nodes are assigned specific tasks.
→ Can allow a single physical node to host multiple "virtual" node types for dedicated tasks.

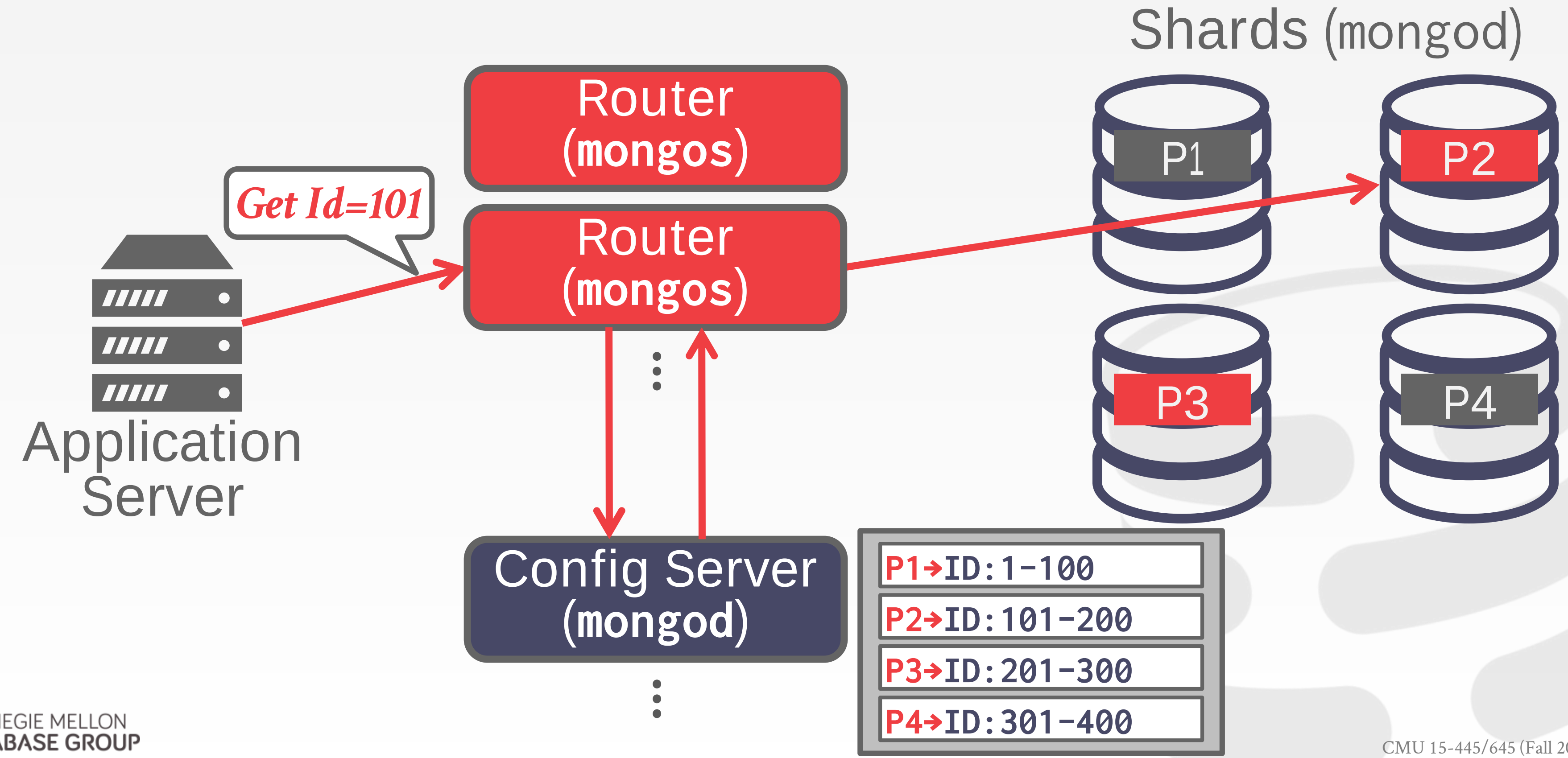
MONGODB CLUSTER ARCHITECTURE



MONGODB CLUSTER ARCHITECTURE



MONGODB CLUSTER ARCHITECTURE



DATA TRANSPARENCY

Users should not be required to know where data is physically located, how tables are **partitioned** or **replicated**.

A SQL query that works on a single-node DBMS should work the same on a distributed DBMS.

DATABASE PARTITIONING

Split database across multiple resources:

→ Disks, nodes, processors.

→ Sometimes called "sharding"

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

NAÏVE TABLE PARTITIONING

Each node stores one and only table.

Assumes that each node has enough storage space for a table.

NAÏVE TABLE PARTITIONING

Table1

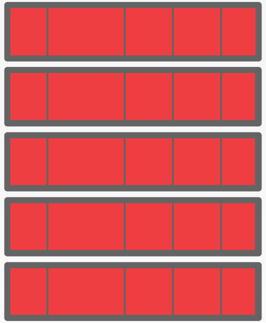
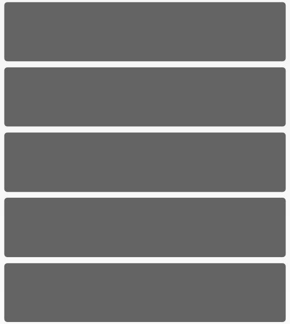
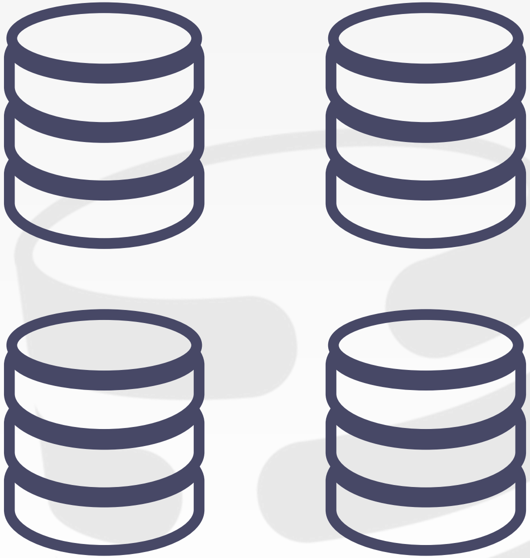


Table2



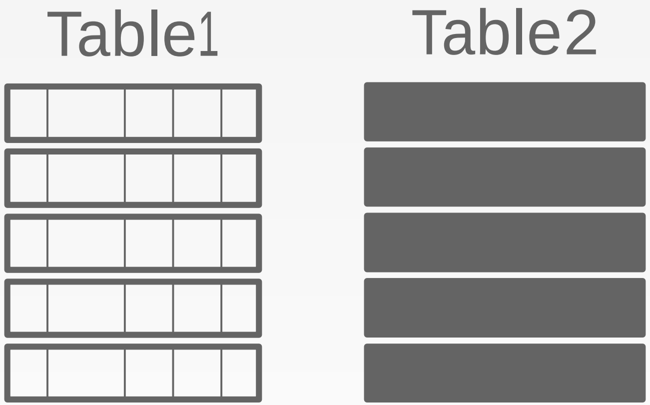
Partitions

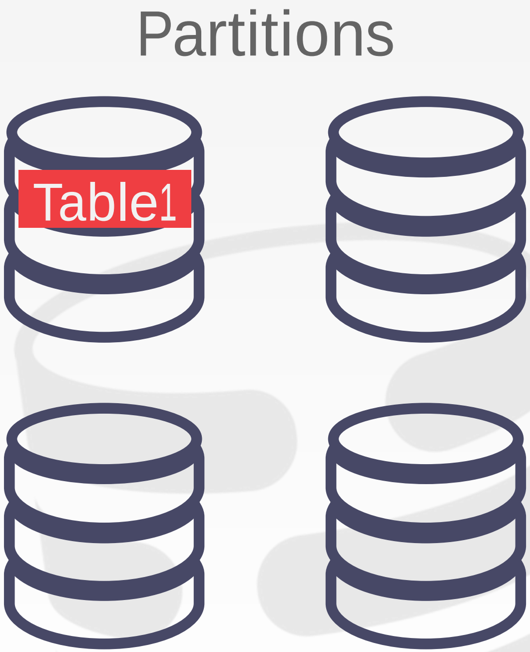


Ideal Query:

```
SELECT * FROM table
```

NAÏVE TABLE PARTITIONING

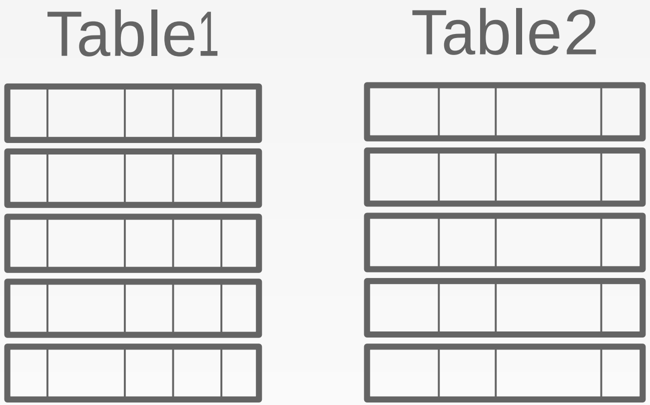


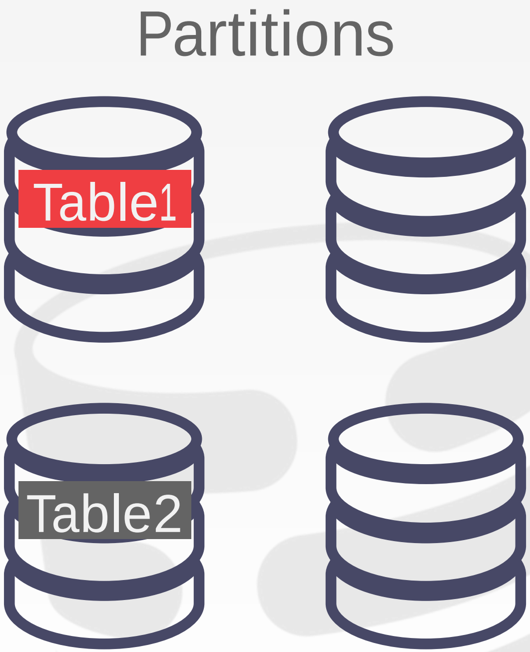


Ideal Query:

```
SELECT * FROM table
```

NAÏVE TABLE PARTITIONING





Ideal Query:

```
SELECT * FROM table
```

HORIZONTAL PARTITIONING

Split a table's tuples into disjoint subsets.
→ Choose column(s) that divides the database equally in terms of size, load, or usage.
→ Each tuple contains all of its columns.
→ Hash Partitioning, Range Partitioning

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

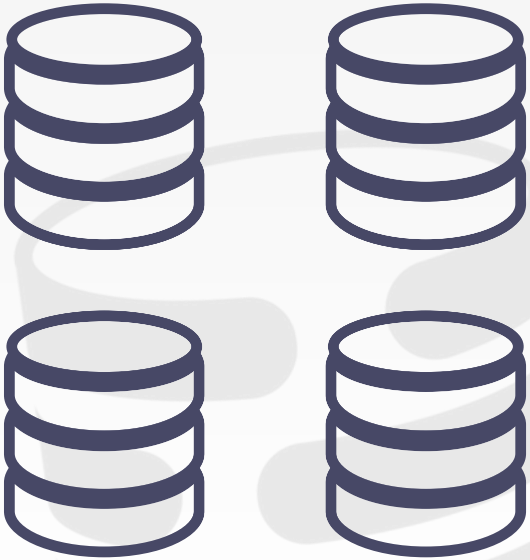
HORIZONTAL PARTITIONING

Partitioning Key

Table1

101	a	XXX	2017-11-29	hash(a)%4 = P2
102	b	XXY	2017-11-28	hash(b)%4 = P4
103	c	XYZ	2017-11-29	hash(c)%4 = P3
104	d	XYX	2017-11-27	hash(d)%4 = P2
105	e	XYY	2017-11-29	hash(e)%4 = P1

Partitions



Ideal Query:

```
SELECT * FROM table
 WHERE partitionKey = ?
```

HORIZONTAL PARTITIONING

Partitioning Key

Table1

101	a	XXX	2017-11-29
102	b	XXY	2017-11-28
103	c	XYZ	2017-11-29
104	d	XYX	2017-11-27
105	e	XYY	2017-11-29

hash(a)%4 = P2

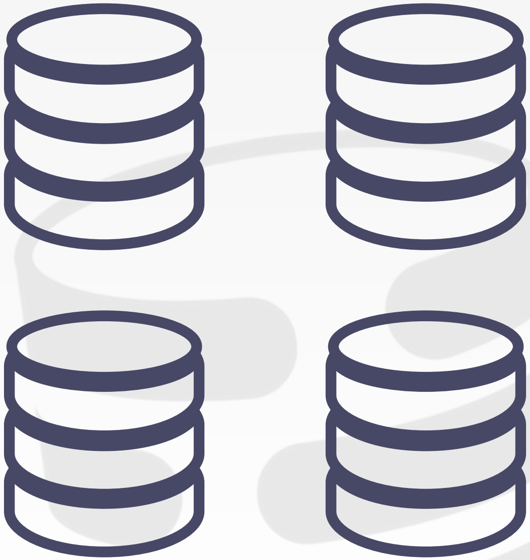
hash(b)%4 = P4

hash(c)%4 = P3

hash(d)%4 = P2

hash(e)%4 = P1

Partitions



Ideal Query:

```
SELECT * FROM table
 WHERE partitionKey = ?
```

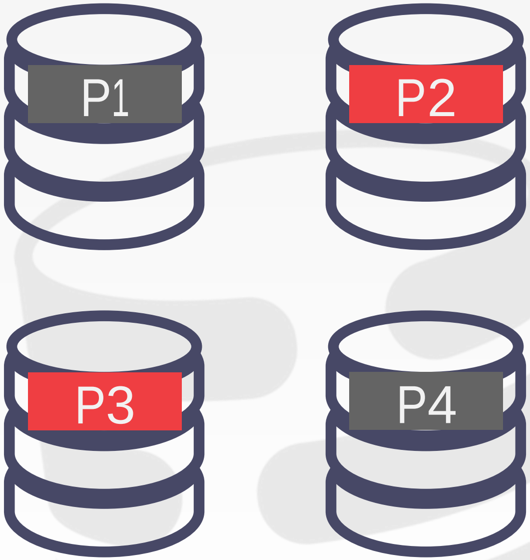
HORIZONTAL PARTITIONING

Partitioning Key

Table1

101	a	XXX	2017-11-29	hash(a)%4 = P2
102	b	XXY	2017-11-28	hash(b)%4 = P4
103	c	XYZ	2017-11-29	hash(c)%4 = P3
104	d	XYX	2017-11-27	hash(d)%4 = P2
105	e	XYY	2017-11-29	hash(e)%4 = P1

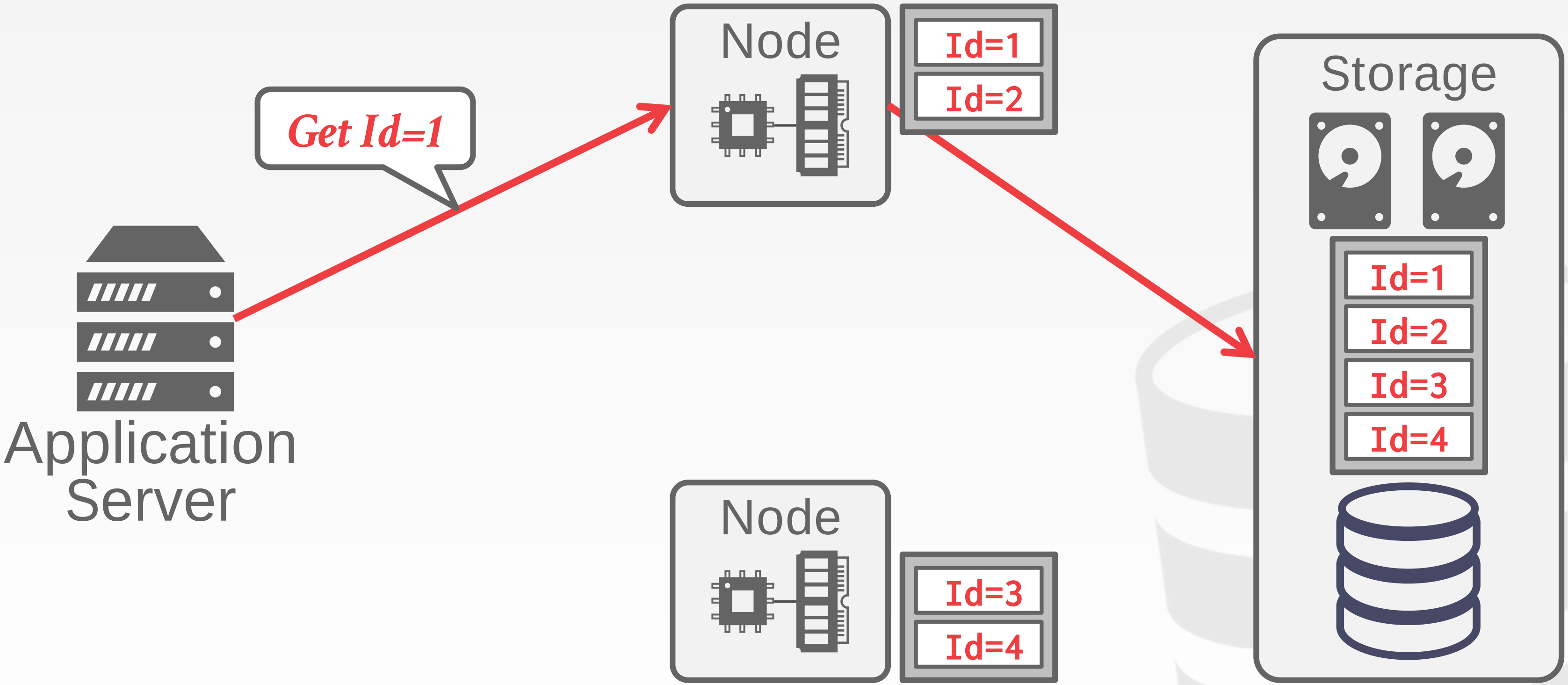
Partitions



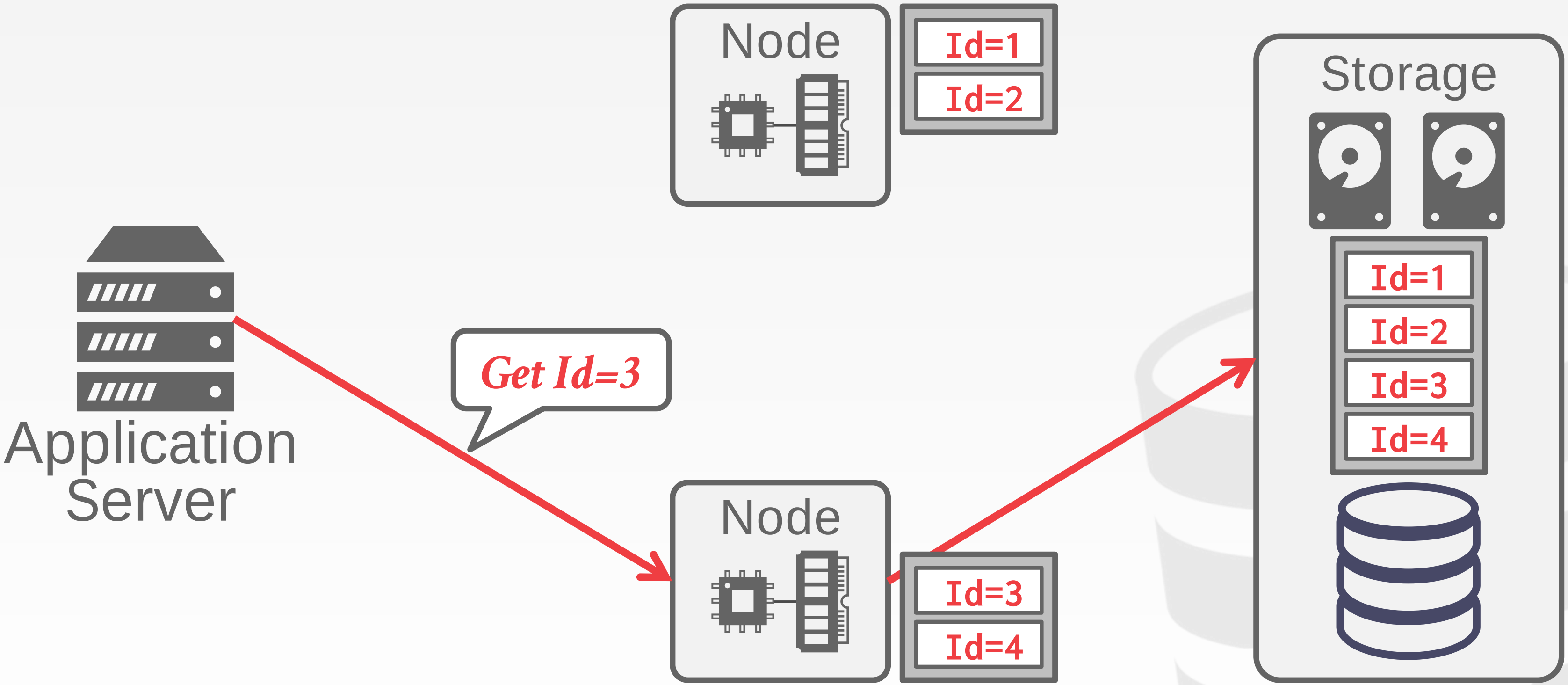
Ideal Query:

```
SELECT * FROM table
 WHERE partitionKey = ?
```

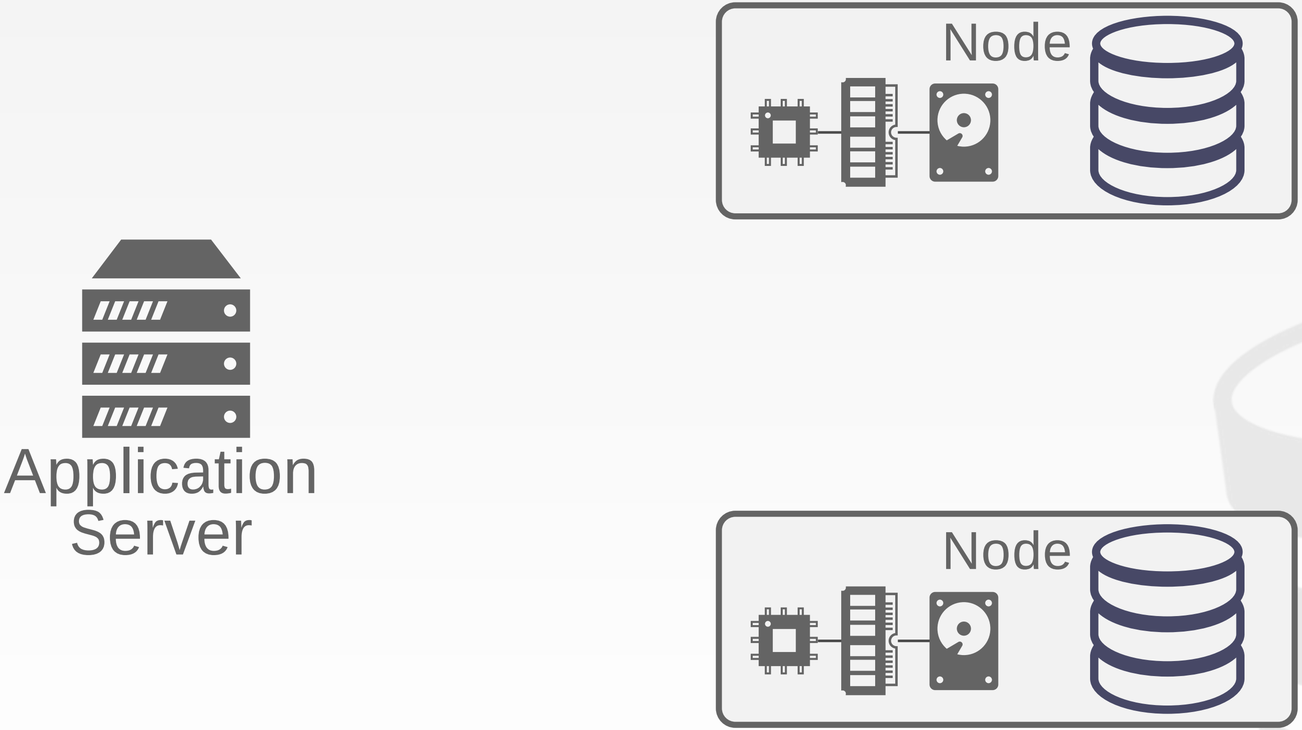
LOGICAL PARTITIONING



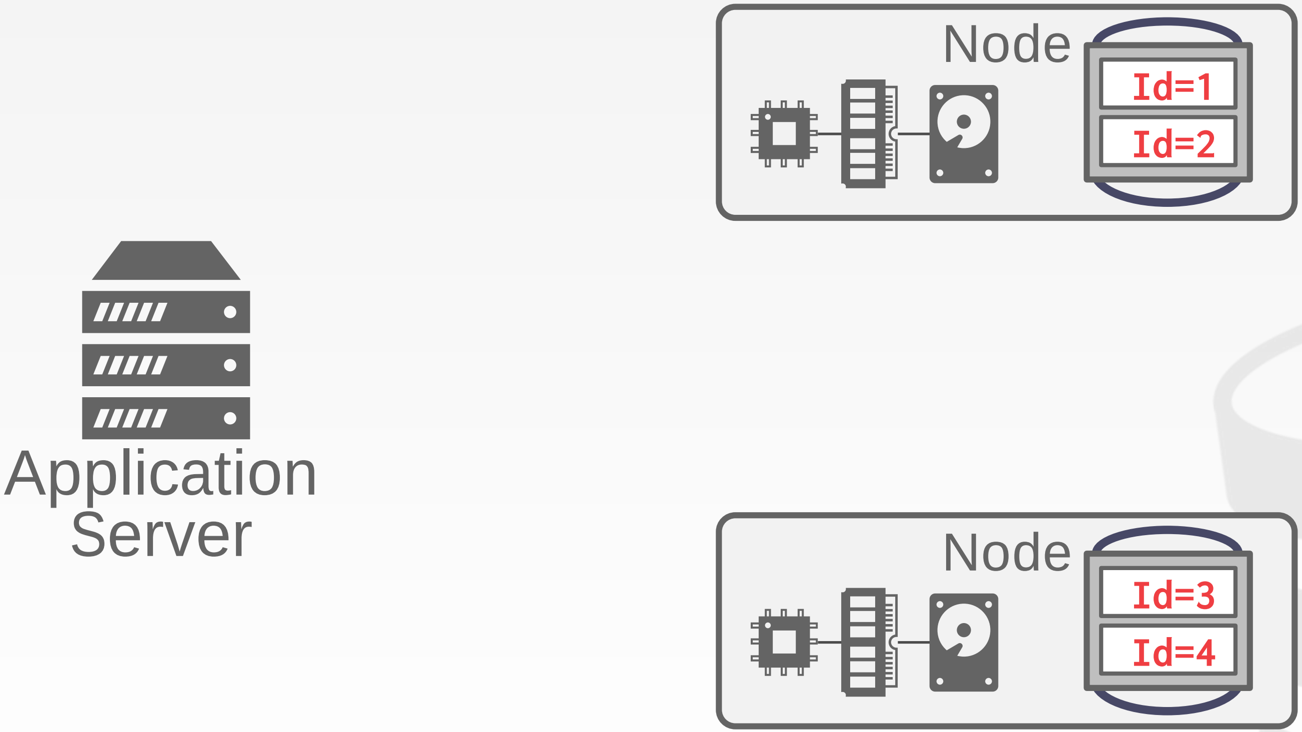
LOGICAL PARTITIONING



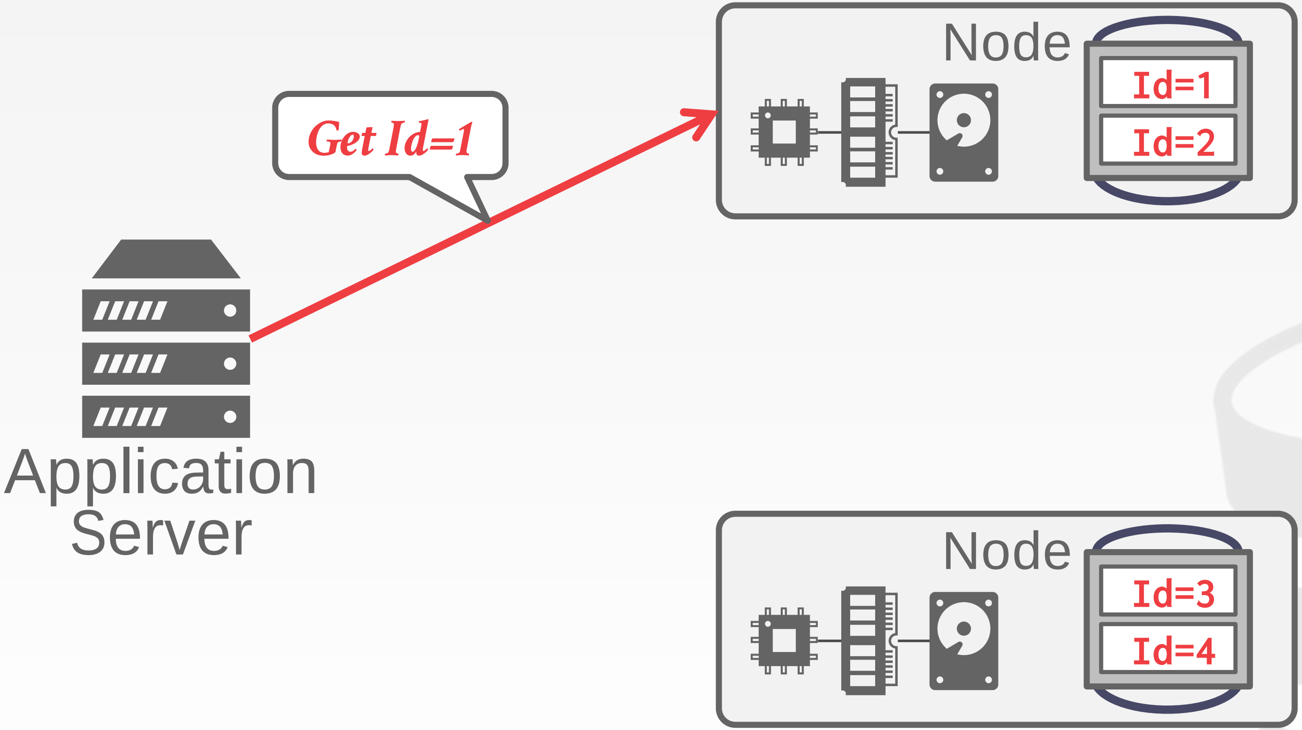
PHYSICAL PARTITIONING



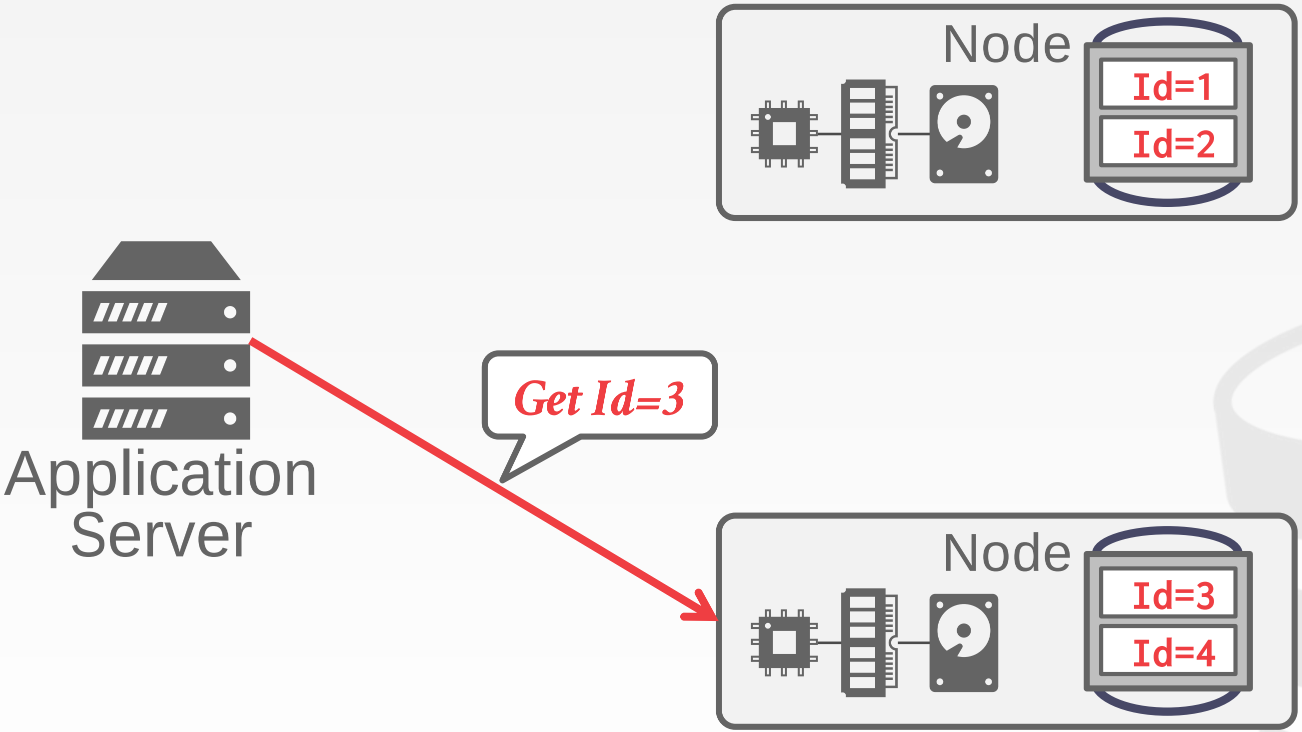
PHYSICAL PARTITIONING



PHYSICAL PARTITIONING



PHYSICAL PARTITIONING



SINGLE-NODE VS. DISTRIBUTED

A **single-node** txn only accesses data that is contained on one partition.

→ The DBMS does not need coordinate the behavior concurrent txns running on other nodes.

A **distributed** txn accesses data at one or more partitions.

→ Requires expensive coordination.

TRANSACTION COORDINATION

If our DBMS supports multi-operation and distributed txns, we need a way to coordinate their execution in the system.

Two different approaches:
→ **Centralized**: Global "traffic cop".
→ **Decentralized**: Nodes organize themselves.

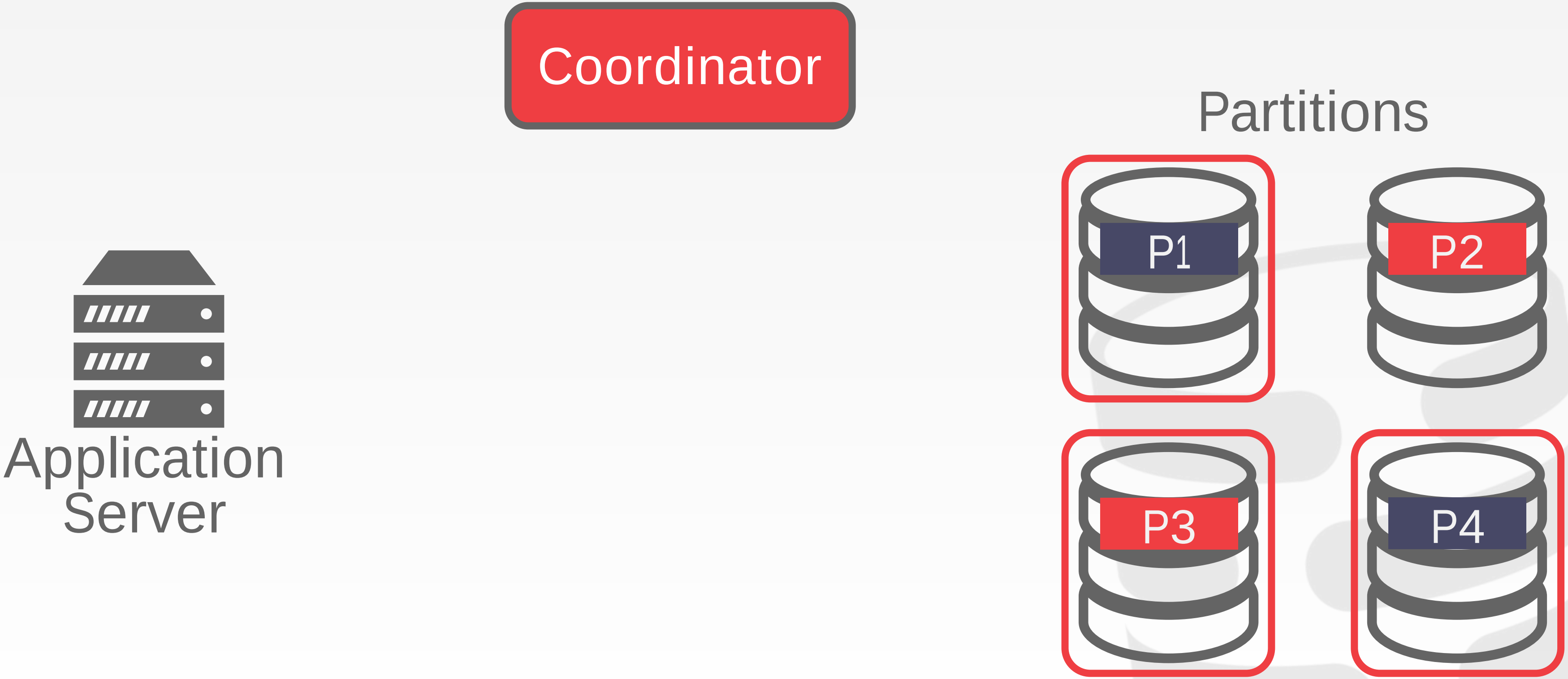
TP MONITORS

Example of a centralized coordinator.

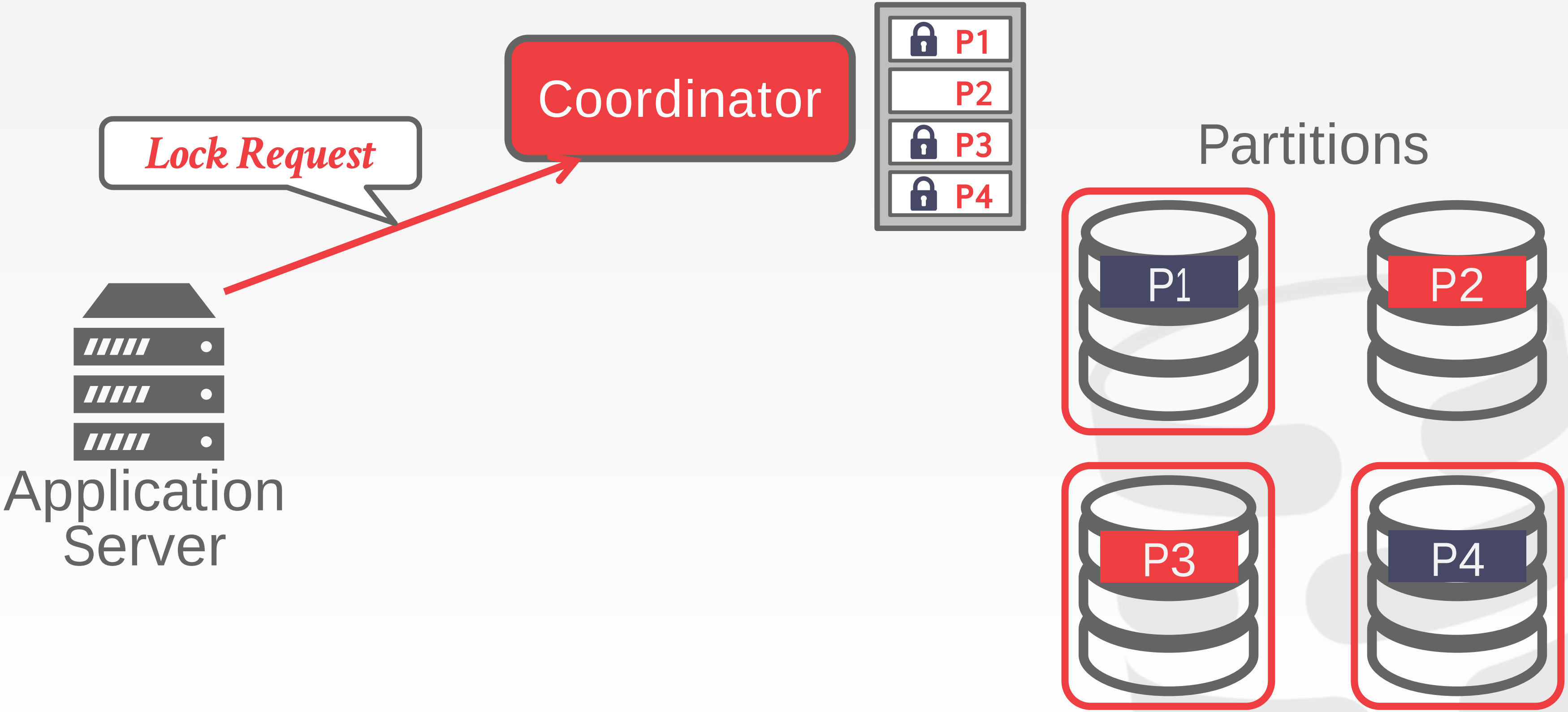
Originally developed in the 1970-80s to provide txns between terminals and mainframe databases.
→ Examples: ATMs, Airline Reservations.

Many DBMSs now support the same functionality internally.

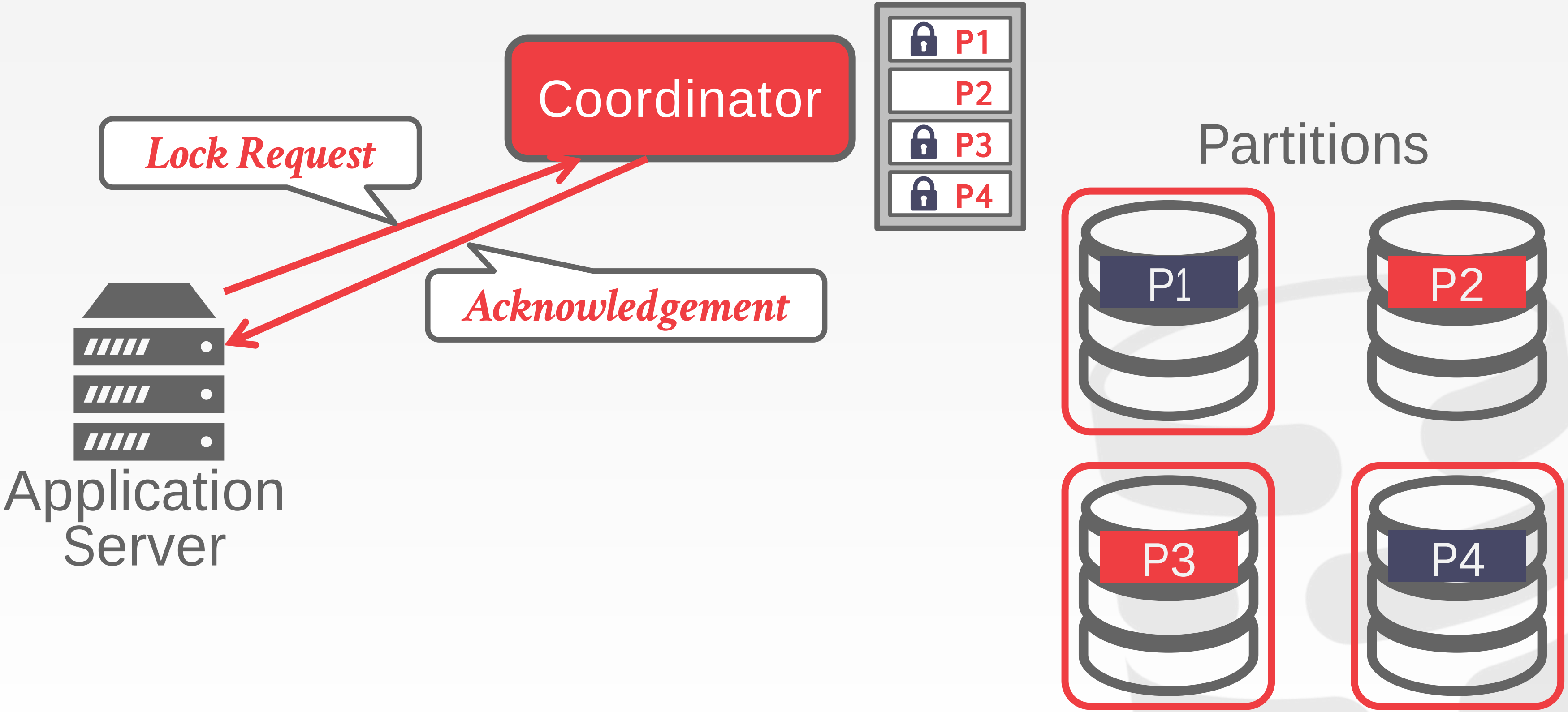
CENTRALIZED COORDINATOR



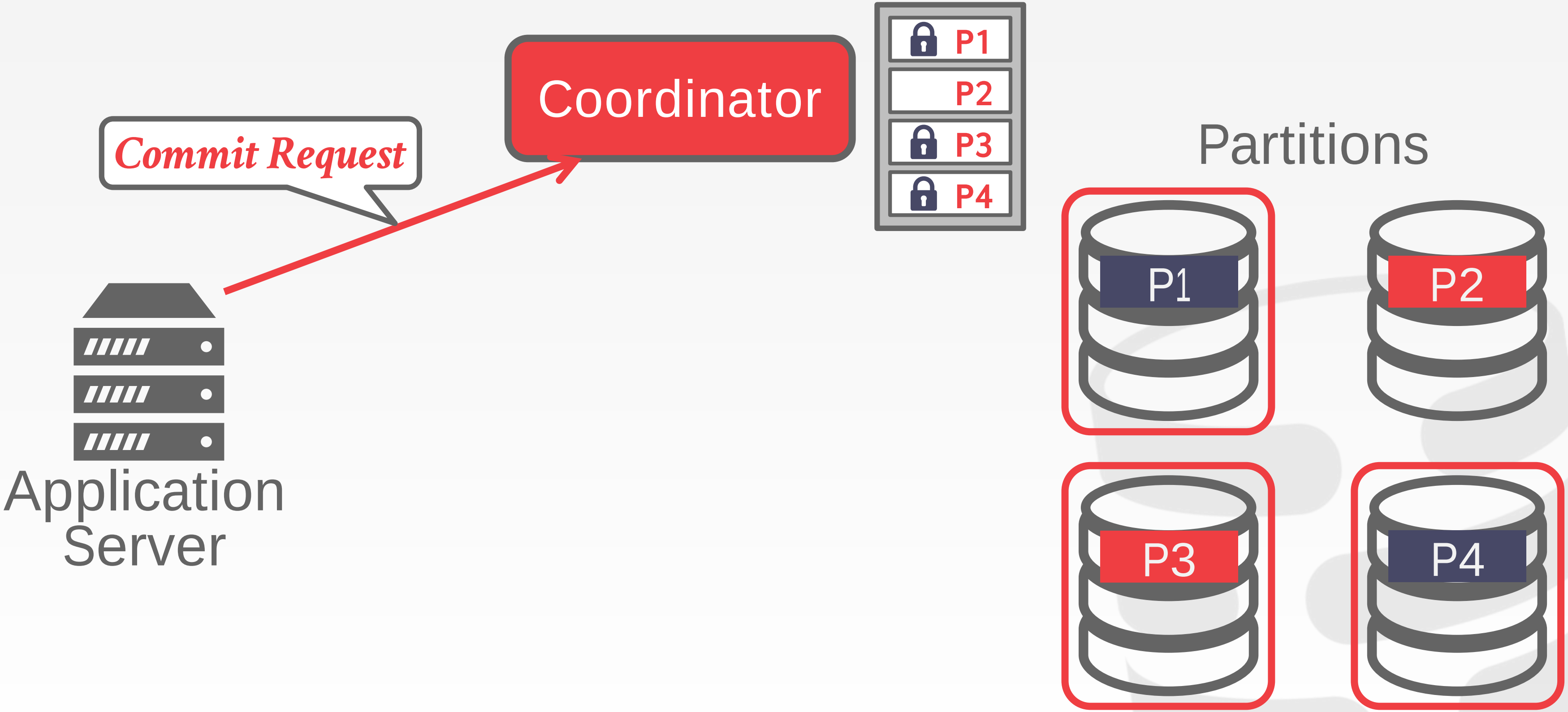
CENTRALIZED COORDINATOR



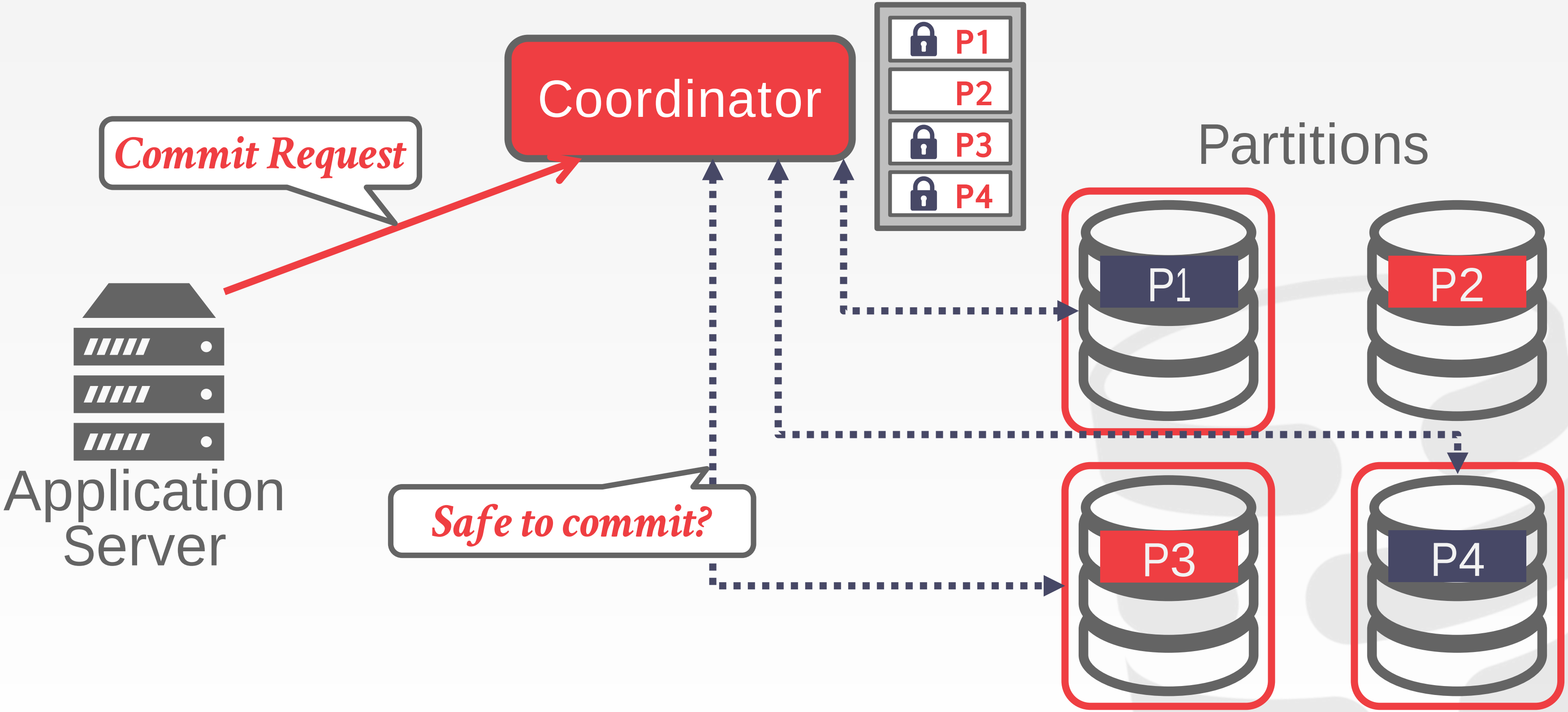
CENTRALIZED COORDINATOR



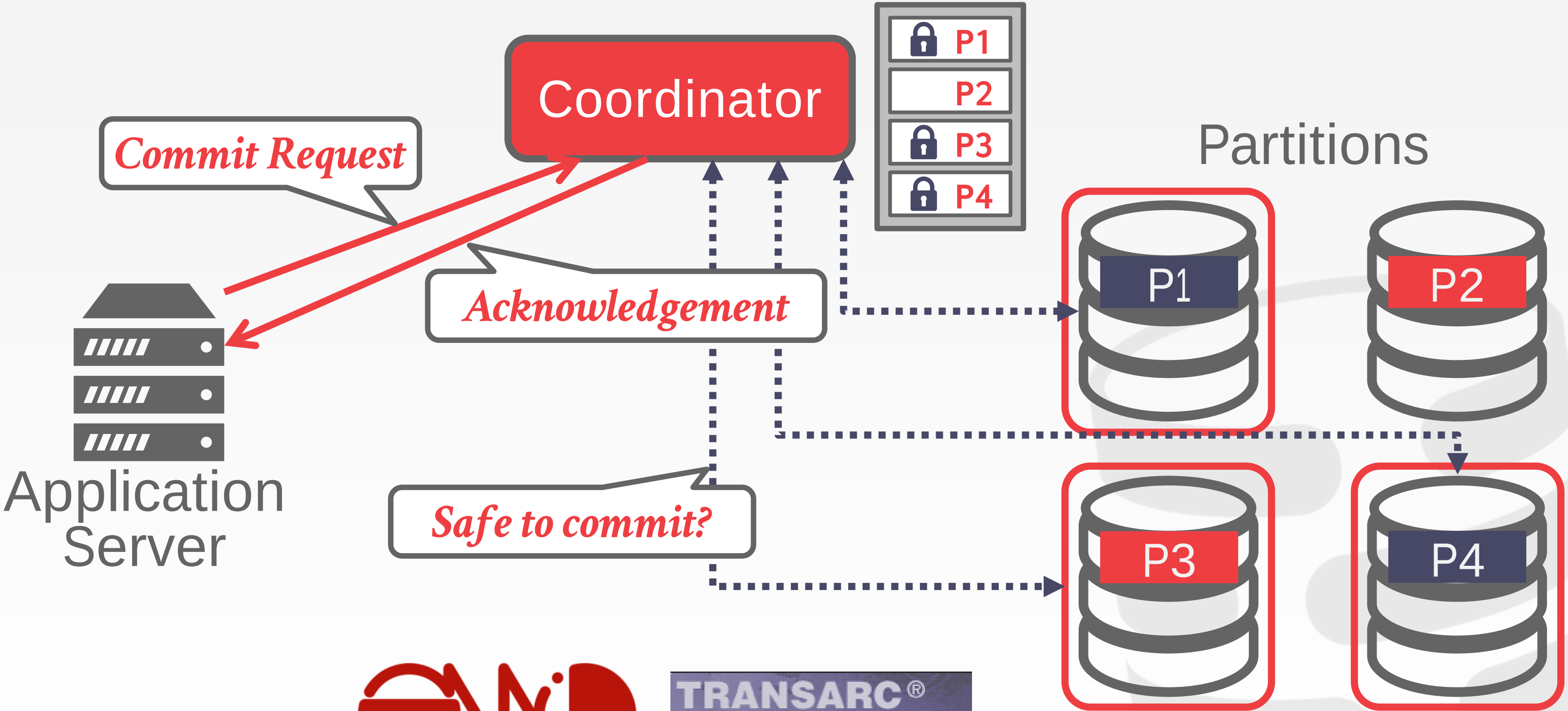
CENTRALIZED COORDINATOR



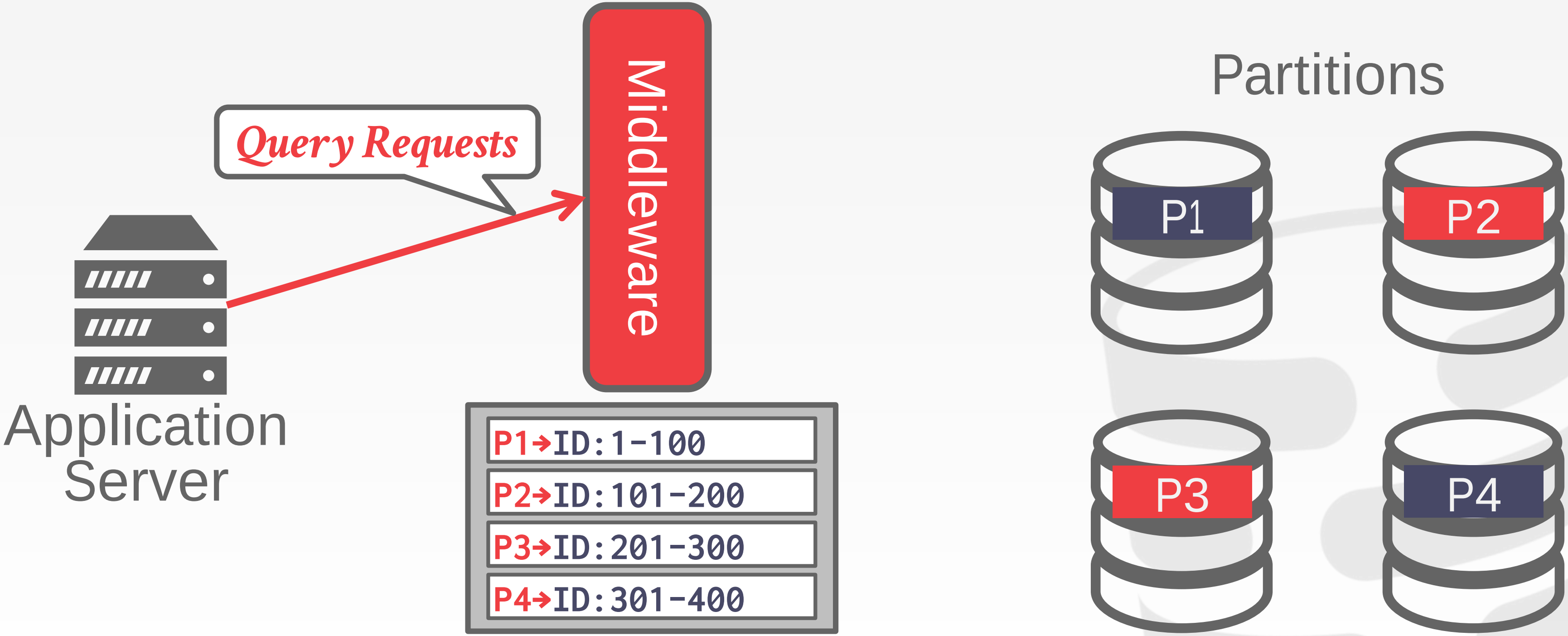
CENTRALIZED COORDINATOR



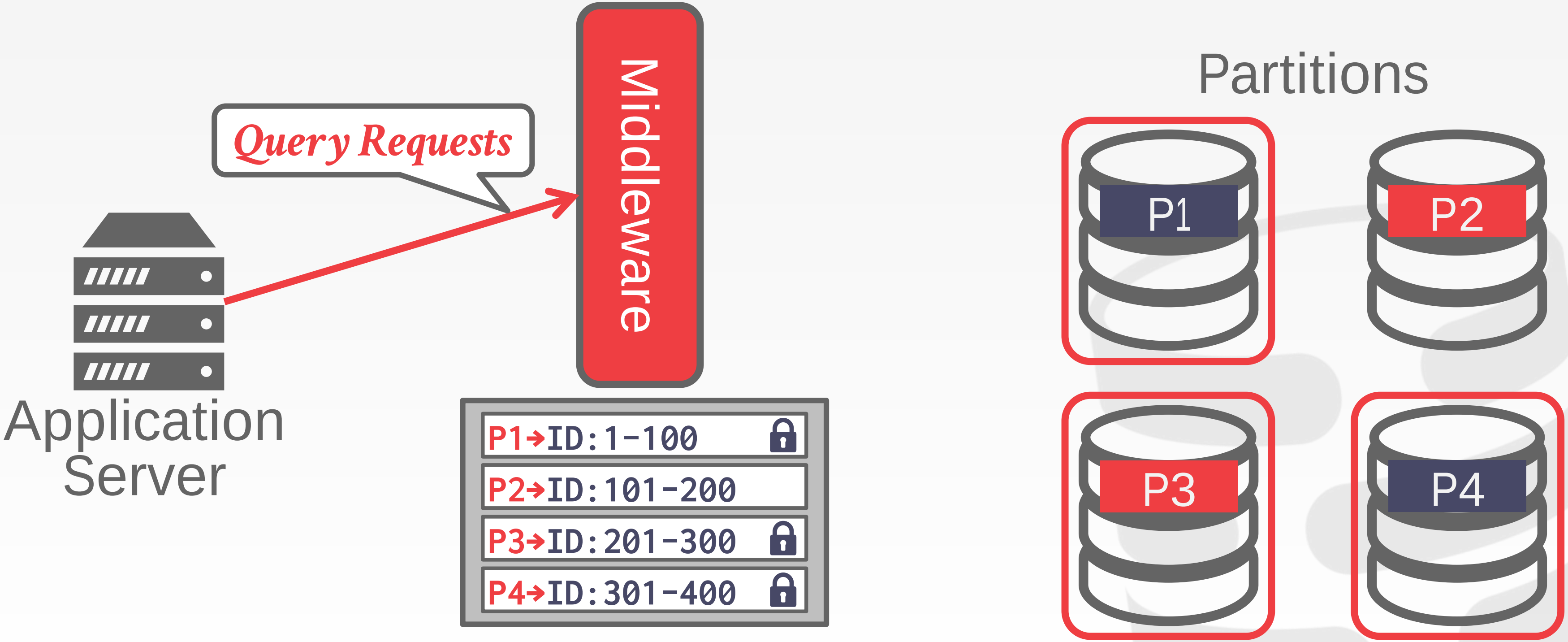
CENTRALIZED COORDINATOR



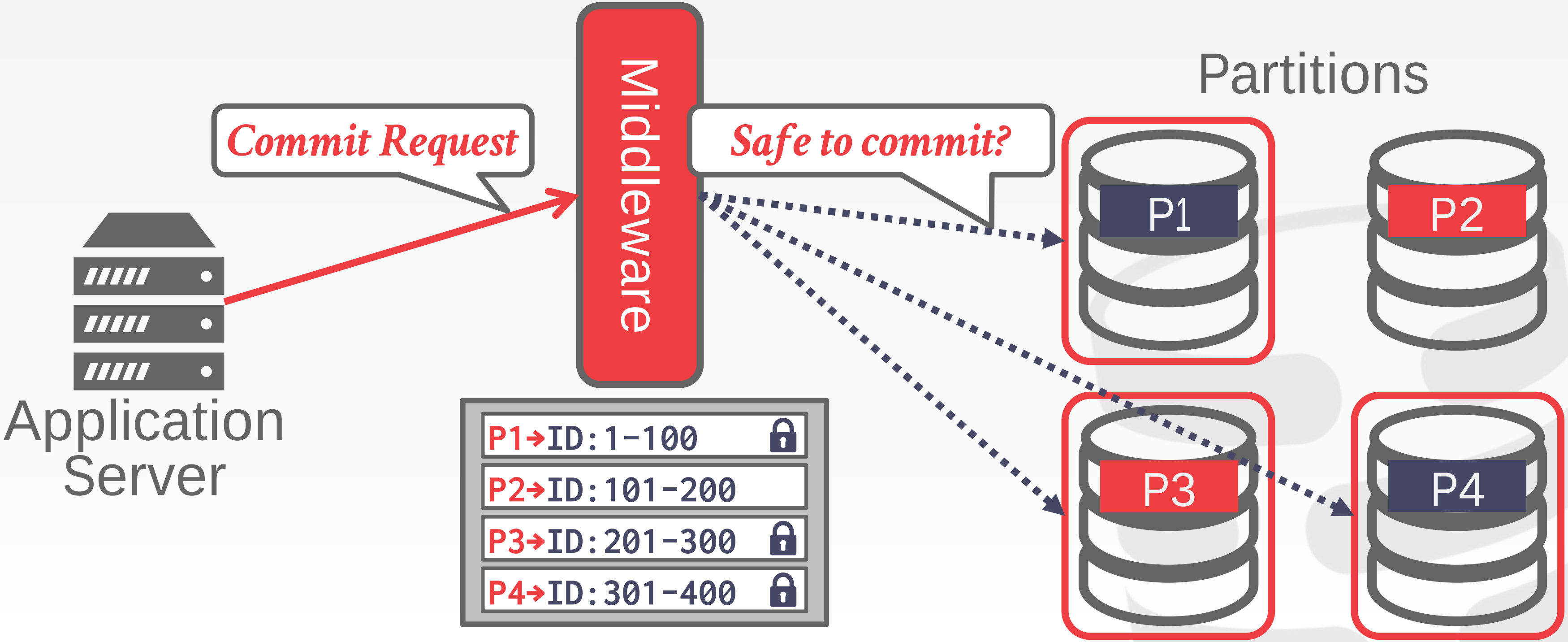
CENTRALIZED COORDINATOR



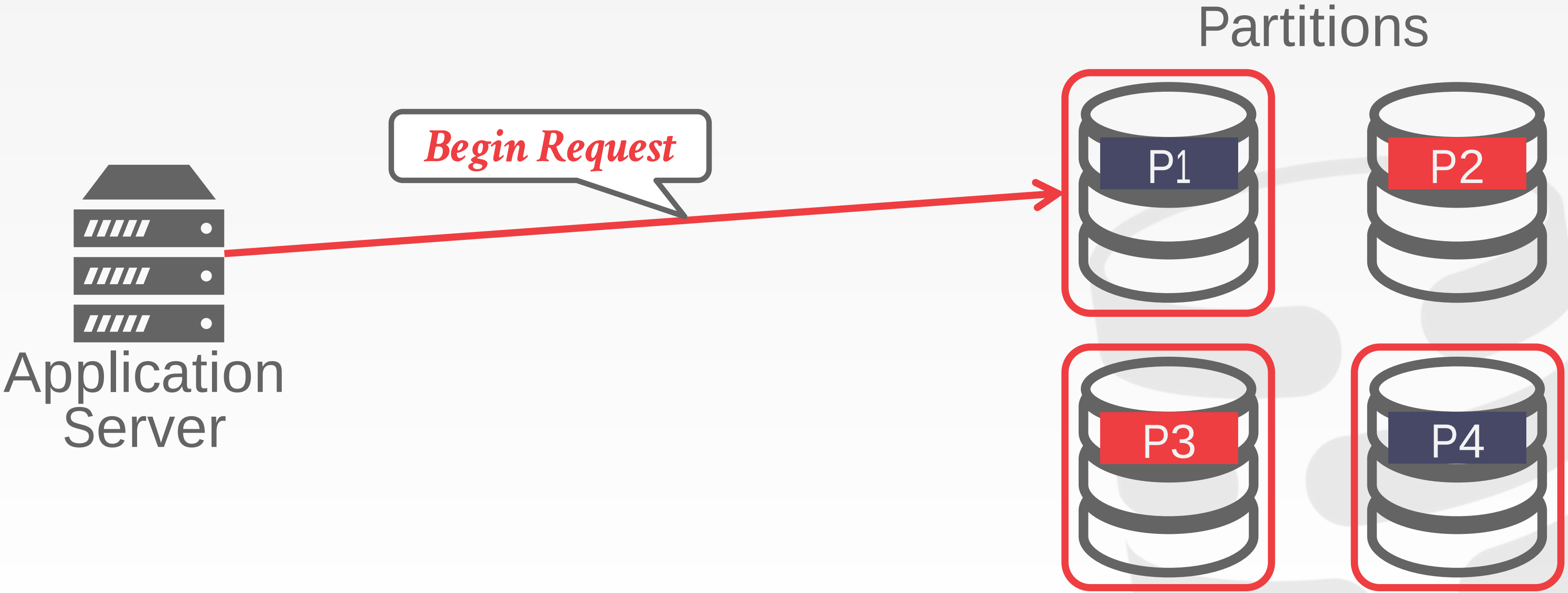
CENTRALIZED COORDINATOR



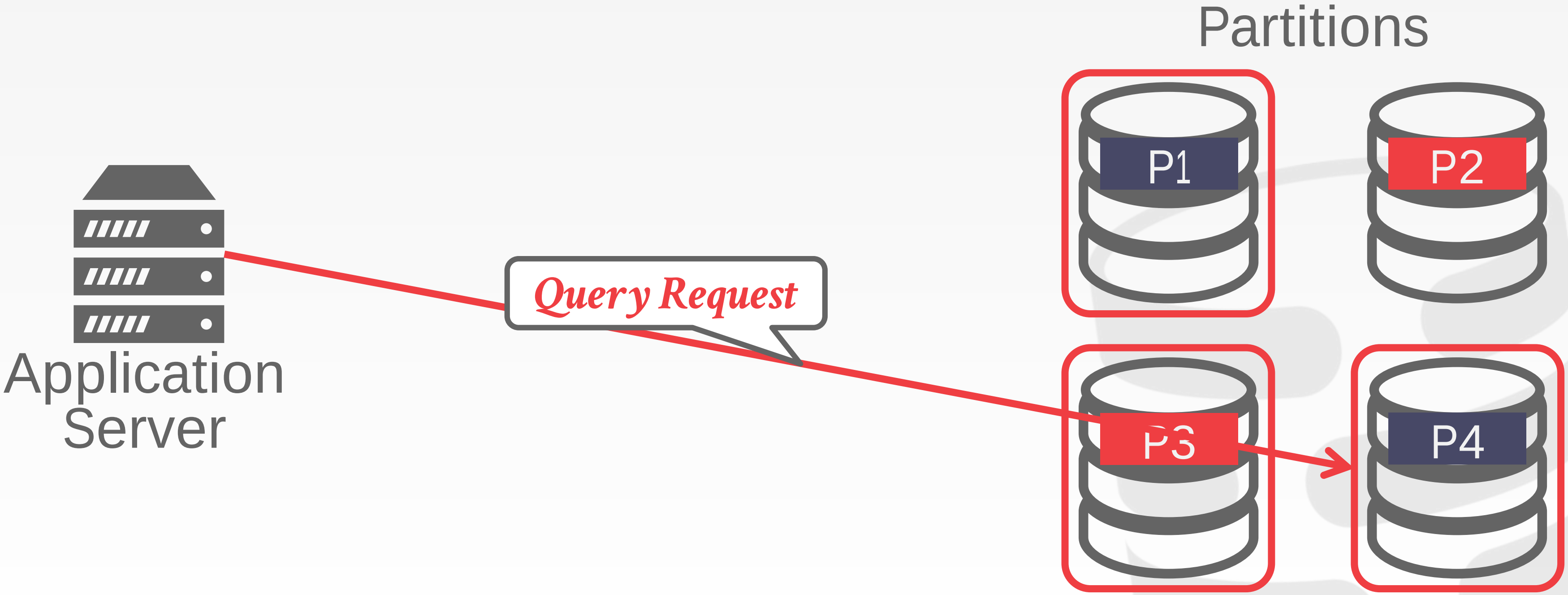
CENTRALIZED COORDINATOR



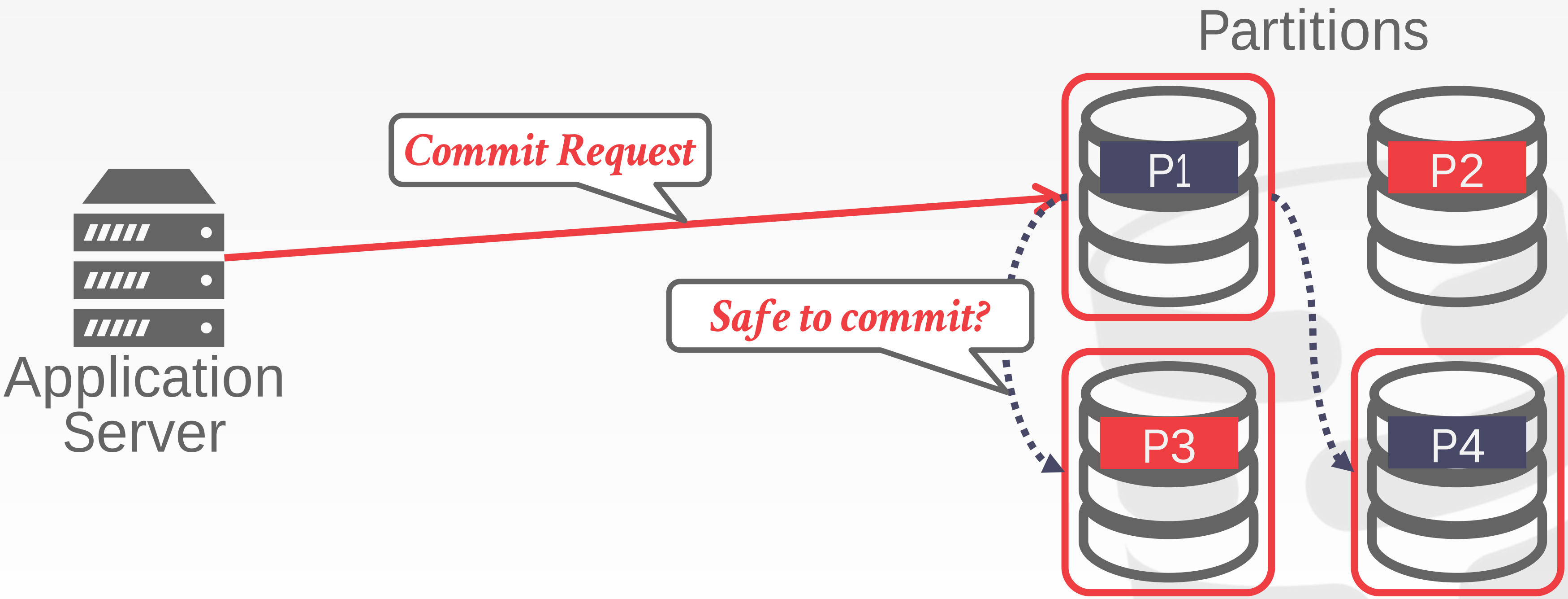
DECENTRALIZED COORDINATOR



DECENTRALIZED COORDINATOR



DECENTRALIZED COORDINATOR



DISTRIBUTED CONCURRENCY CONTROL

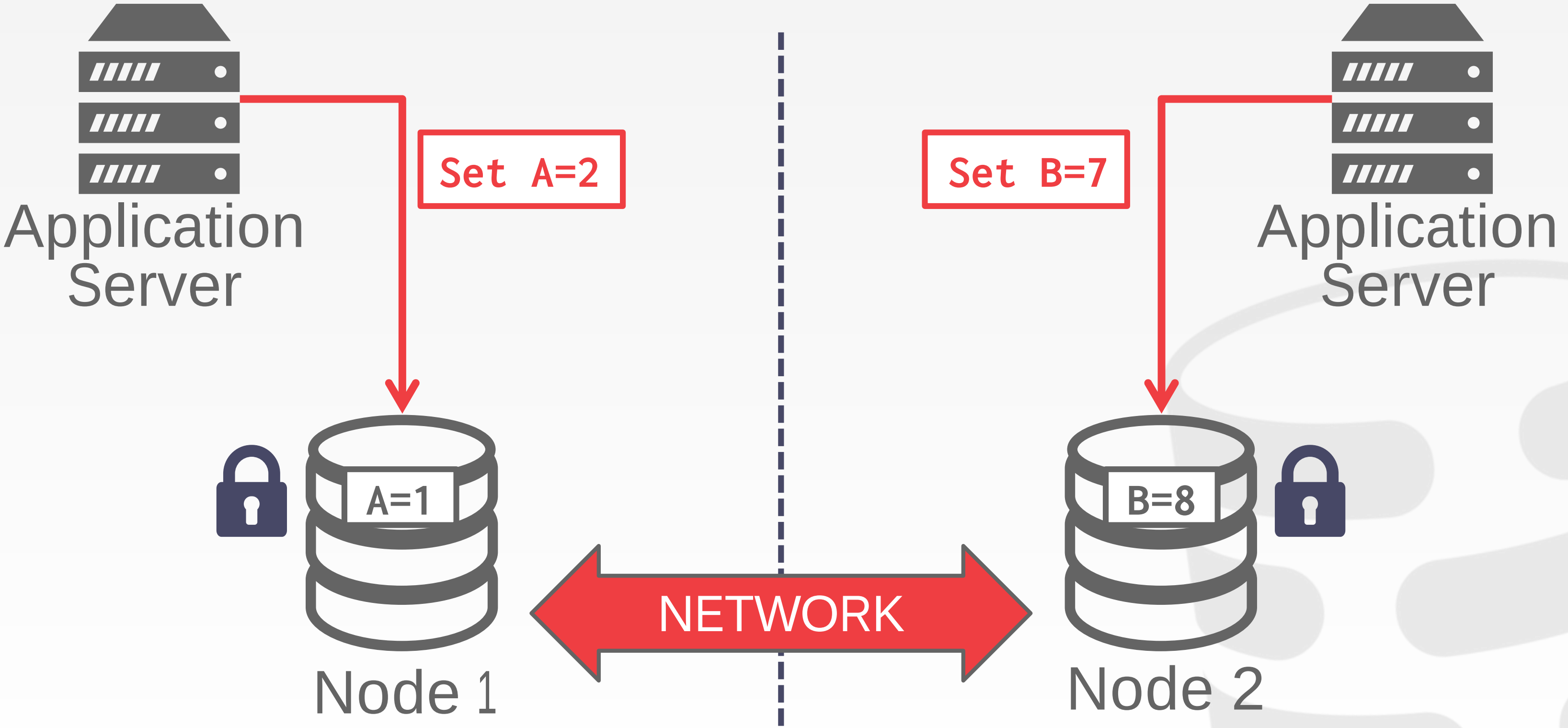
Need to allow multiple txns to execute simultaneously across multiple nodes.

→ Many of the same protocols from single-node DBMSs can be adapted.

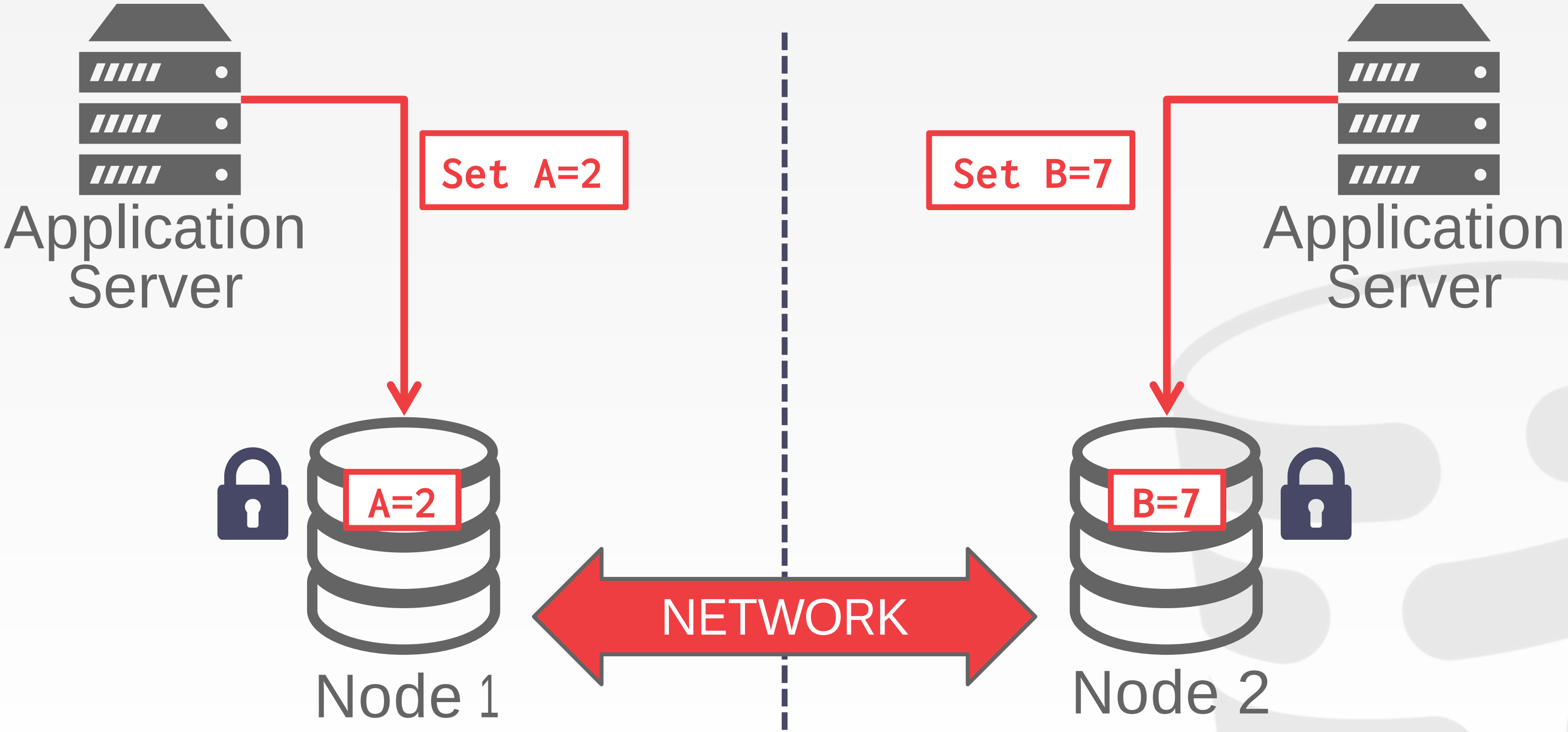
This is harder because of:

→ Replication.
→ Network Communication Overhead.
→ Node Failures.
→ Clock Skew.

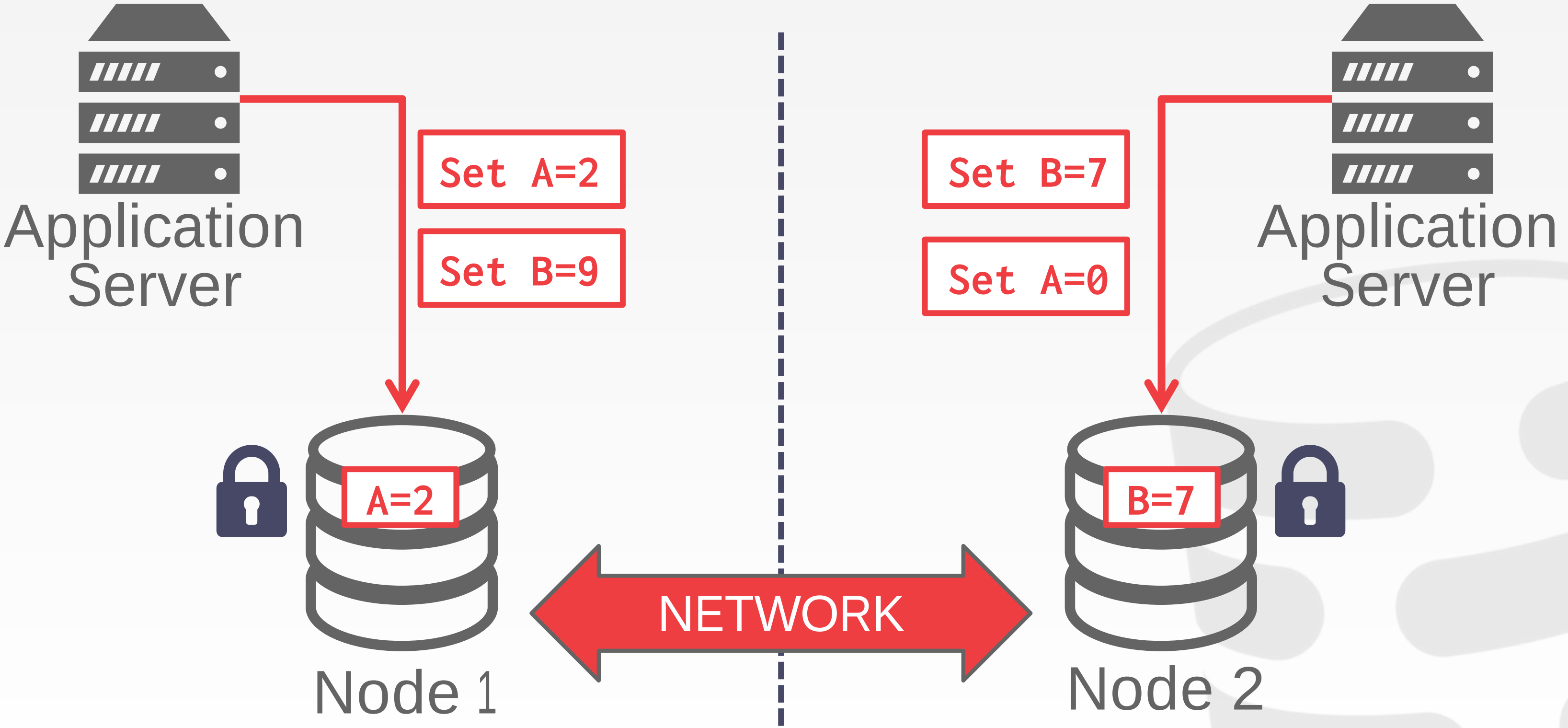
DISTRIBUTED 2PL



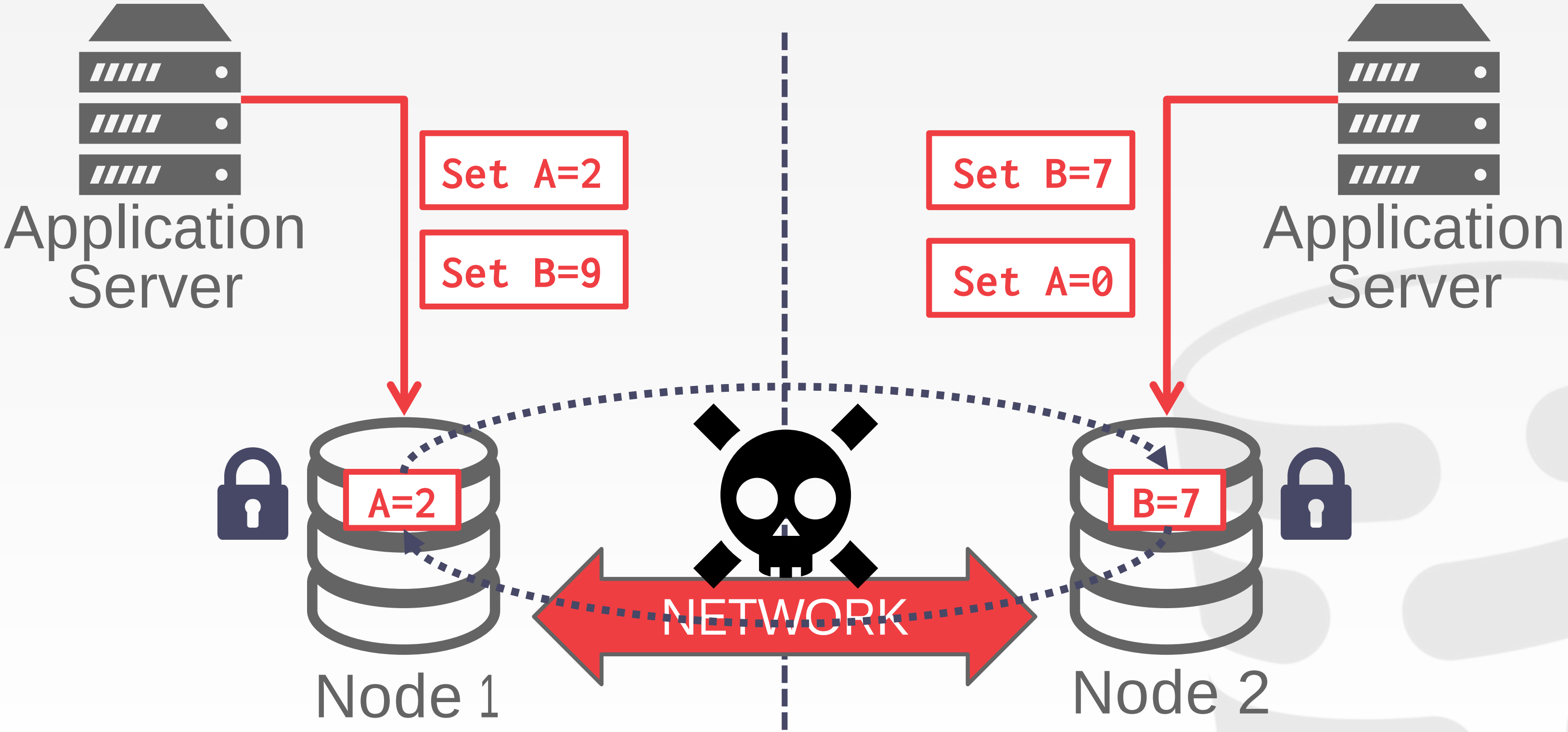
DISTRIBUTED 2PL



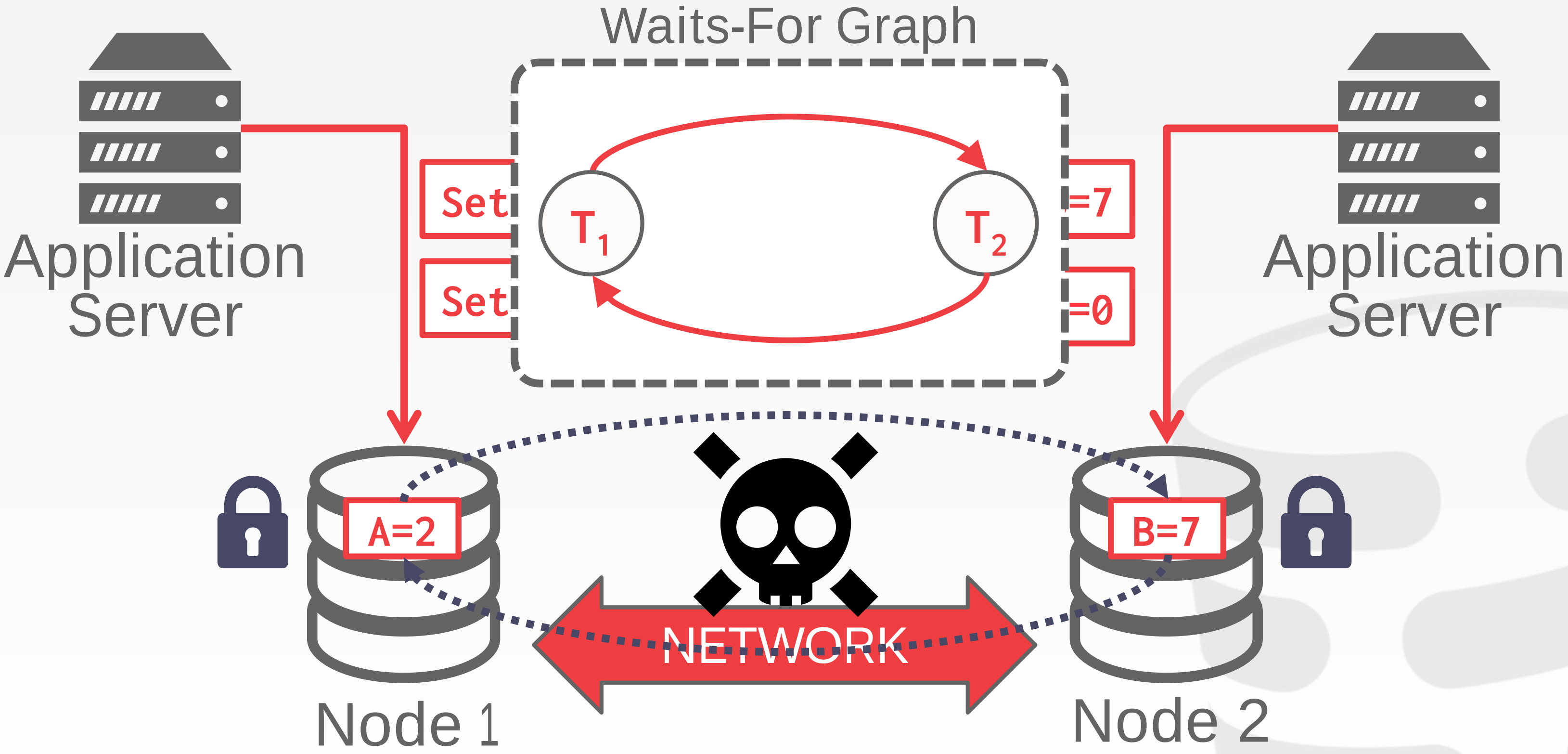
DISTRIBUTED 2PL



DISTRIBUTED 2PL



DISTRIBUTED 2PL



OBSERVATION

We have not discussed how to ensure that all nodes agree to commit a txn and then to make sure it does commit if we decide that it should.

→ What happens if a node fails?
→ What happens if our messages show up late?

ATOMIC COMMIT PROTOCOL

When a multi-node txn finishes, the DBMS needs to ask all of the nodes involved whether it is safe to commit.

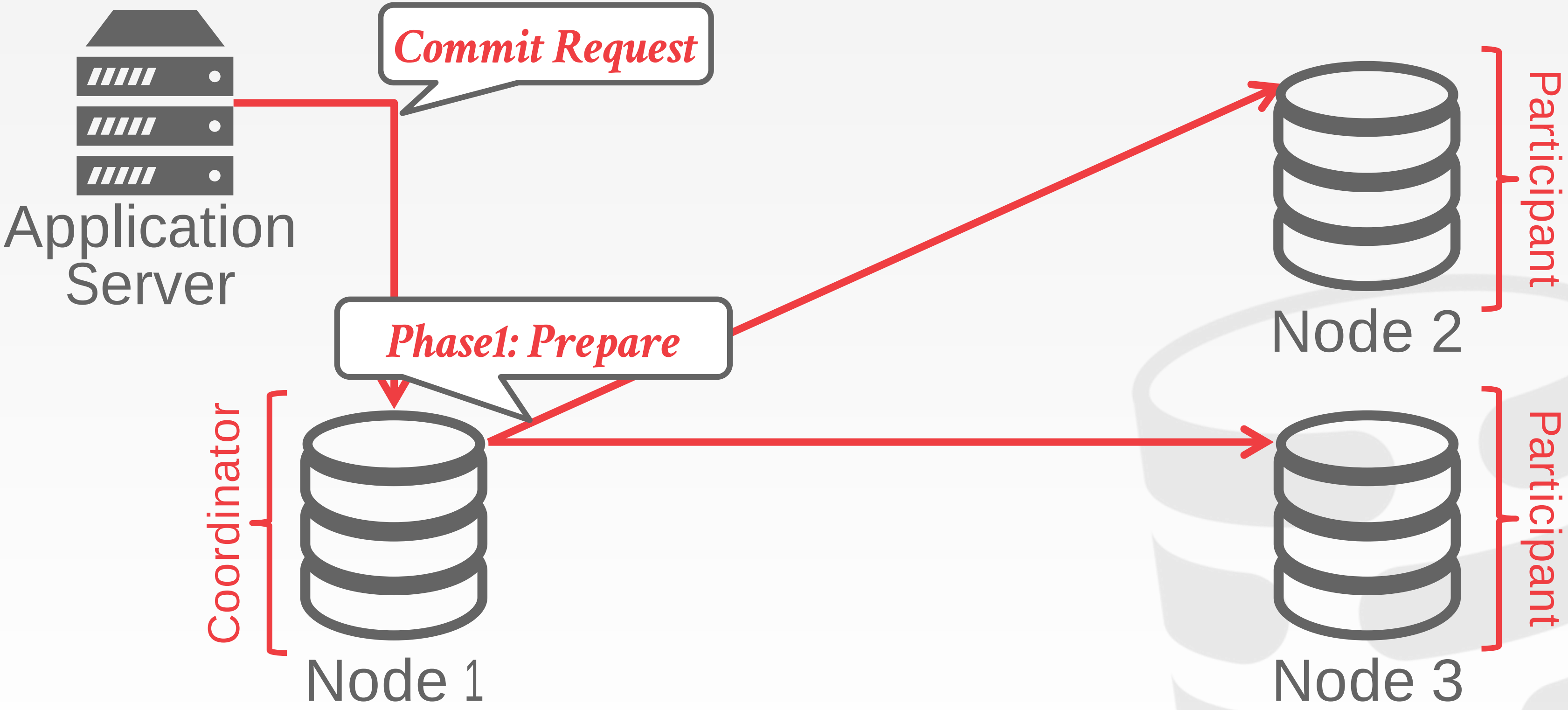
→ All nodes must agree on the outcome

Examples:

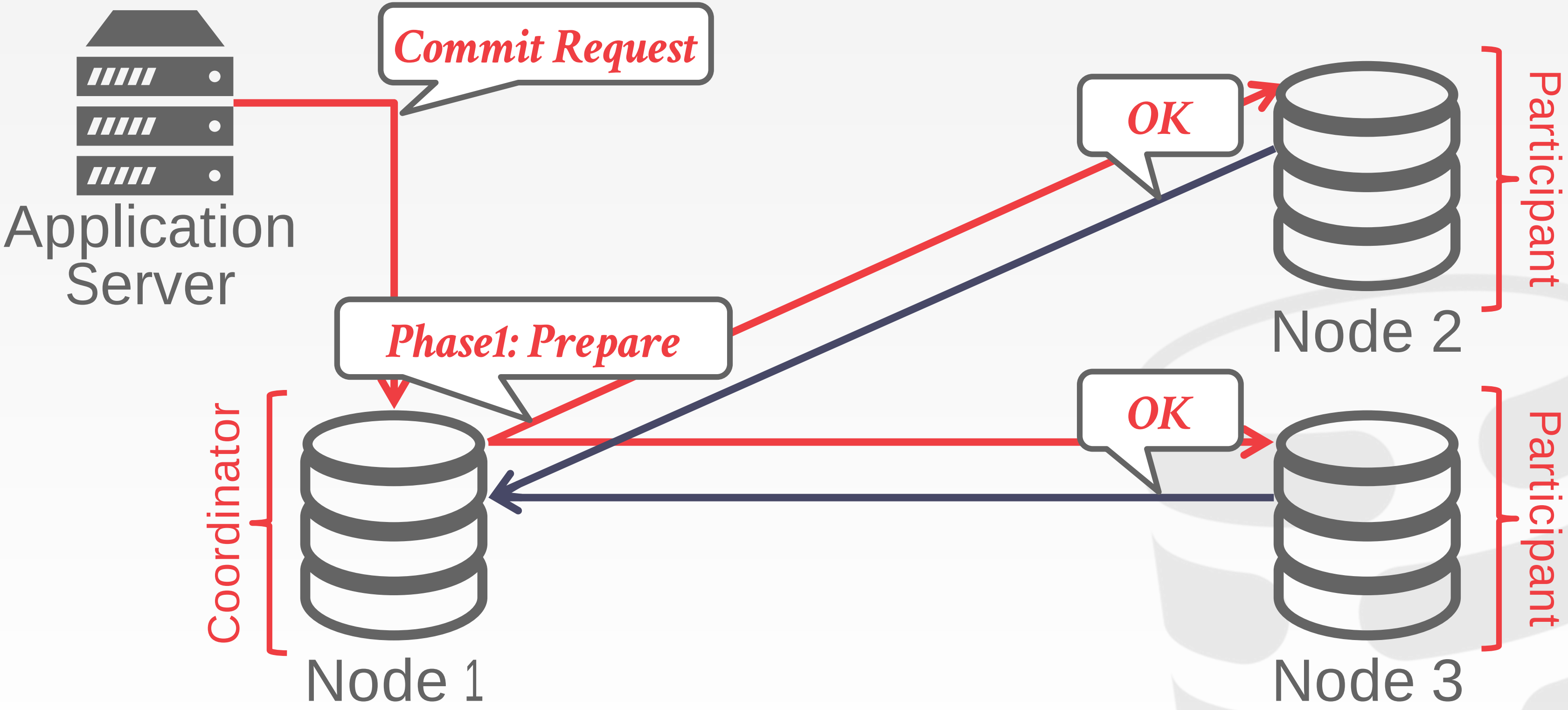
→ Two-Phase Commit
→ Three-Phase Commit (not used)
→ Paxos
→ Raft
→ ZAB (Apache Zookeeper)

TWO-PHASE COMMIT (SUCCESS)

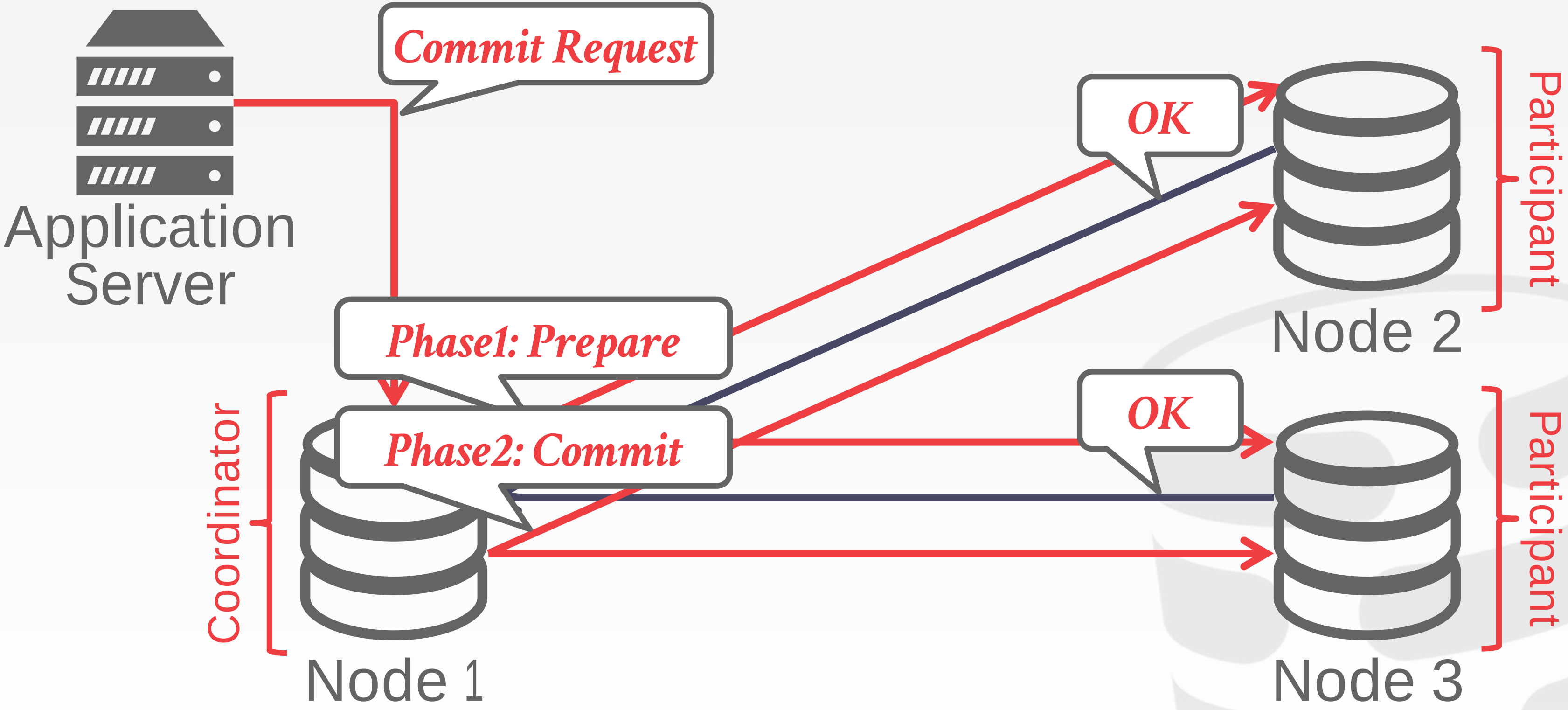
TWO-PHASE COMMIT (SUCCESS)



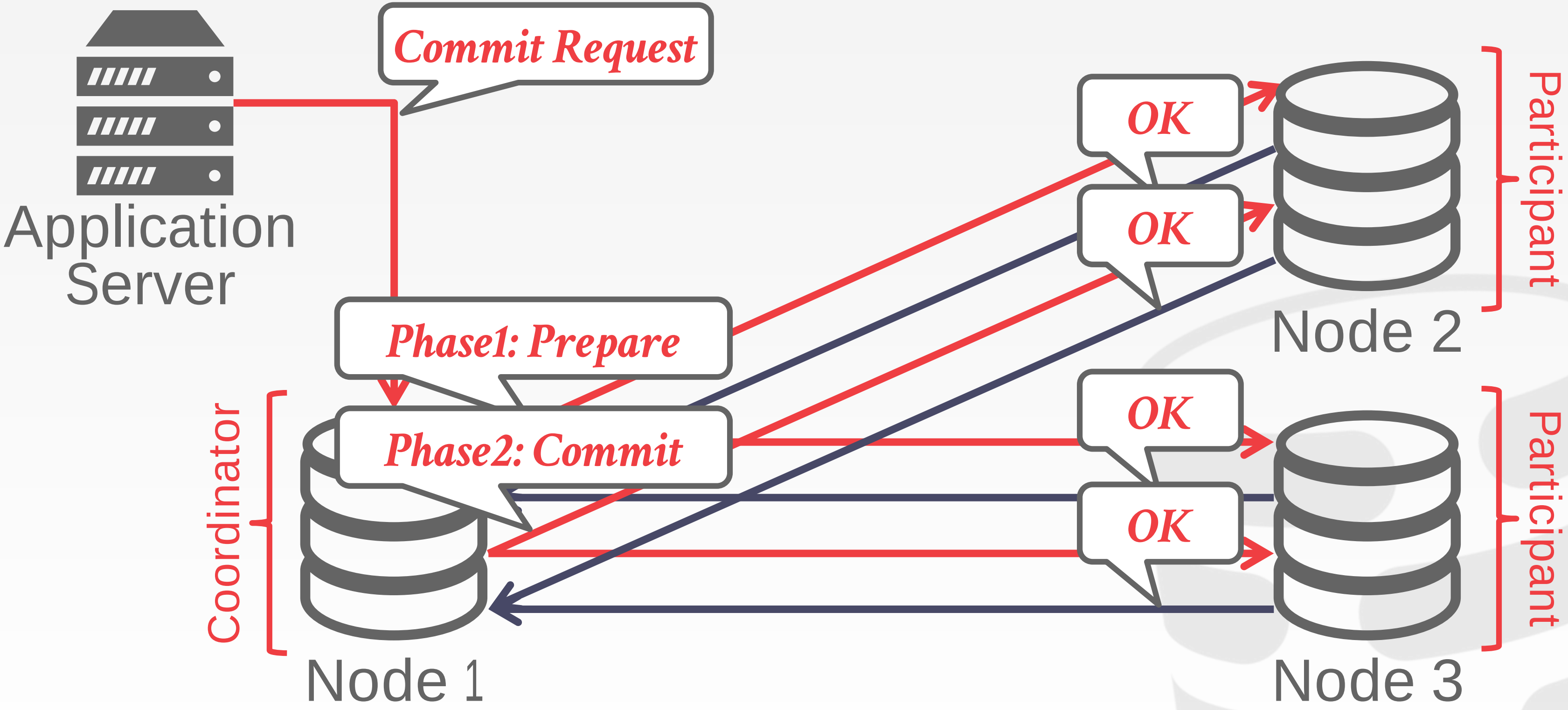
TWO-PHASE COMMIT (SUCCESS)



TWO-PHASE COMMIT (SUCCESS)

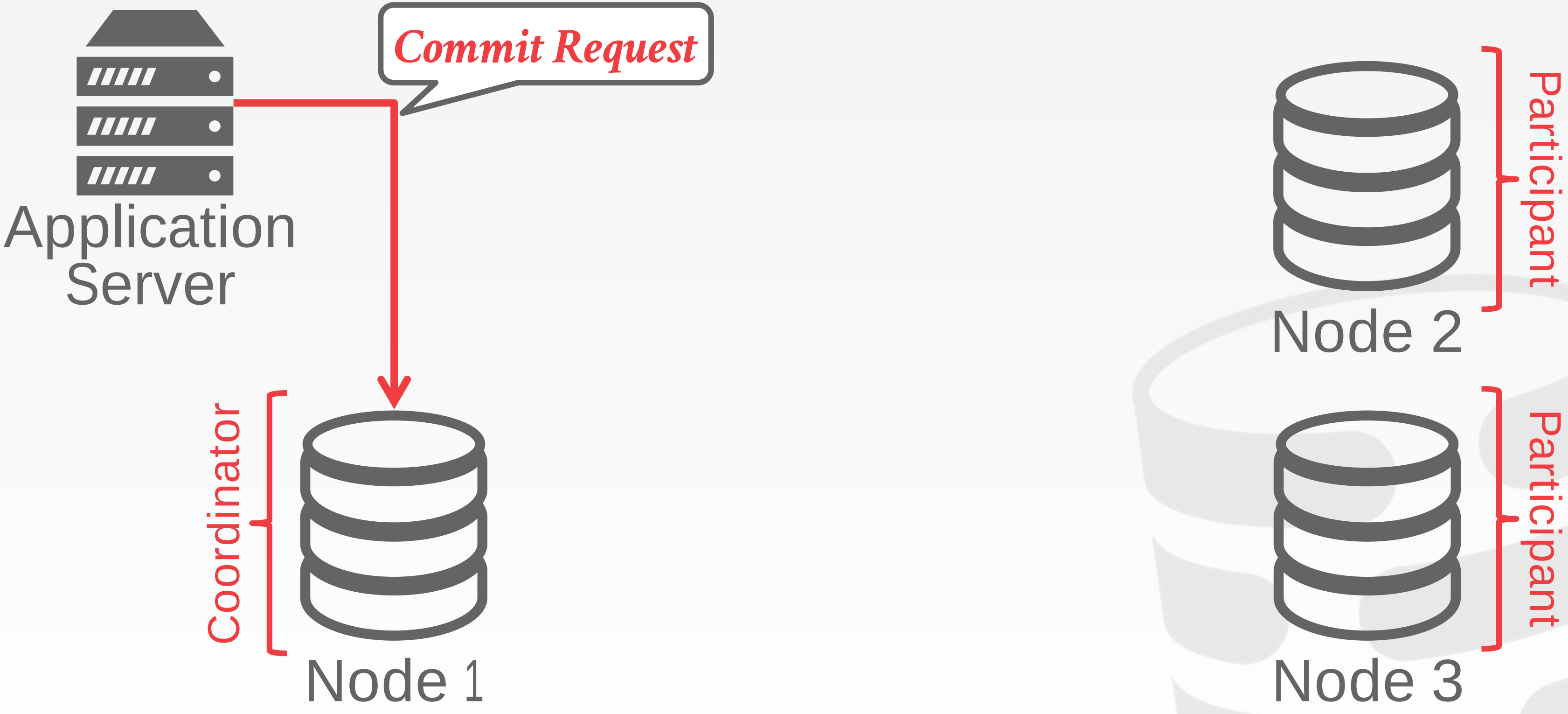


TWO-PHASE COMMIT (SUCCESS)

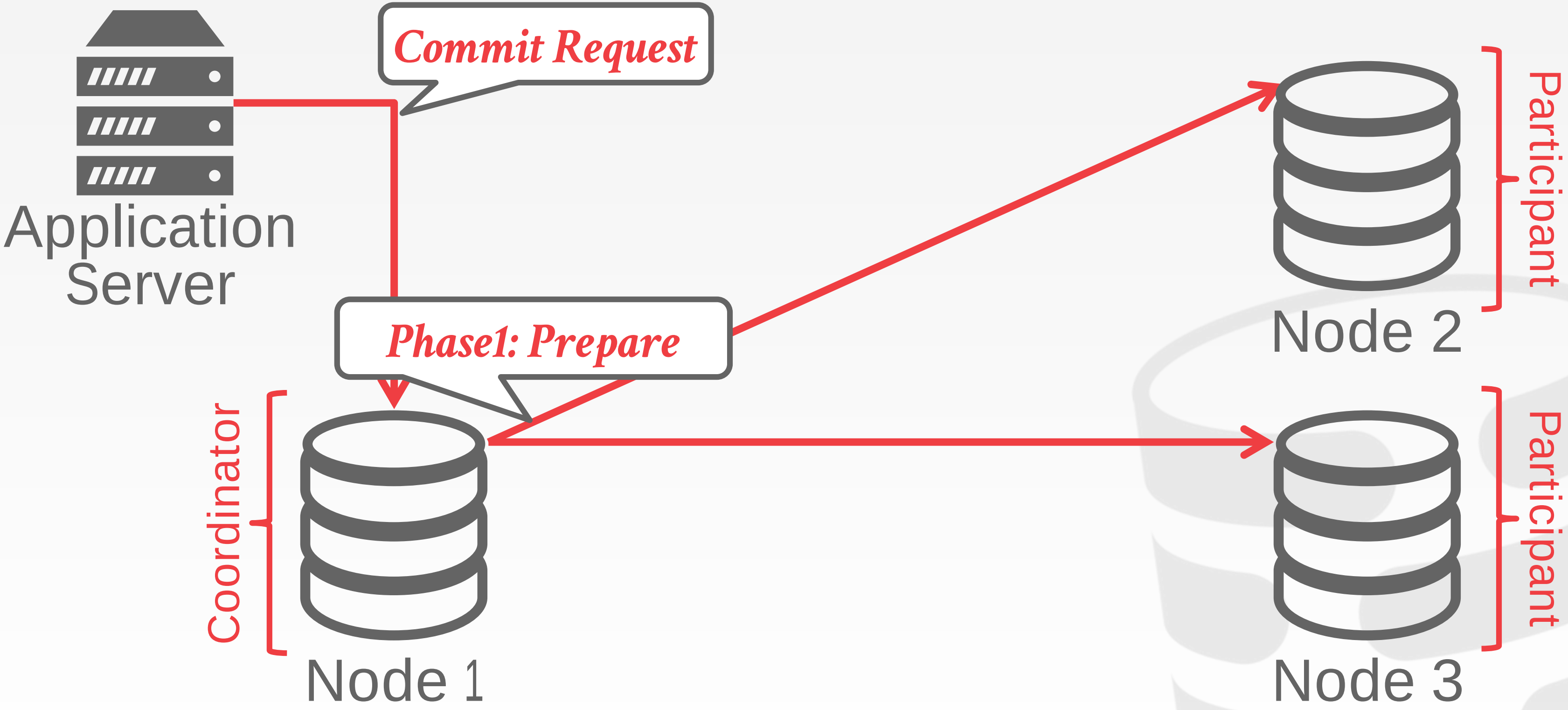


TWO-PHASE COMMIT (SUCCESS)

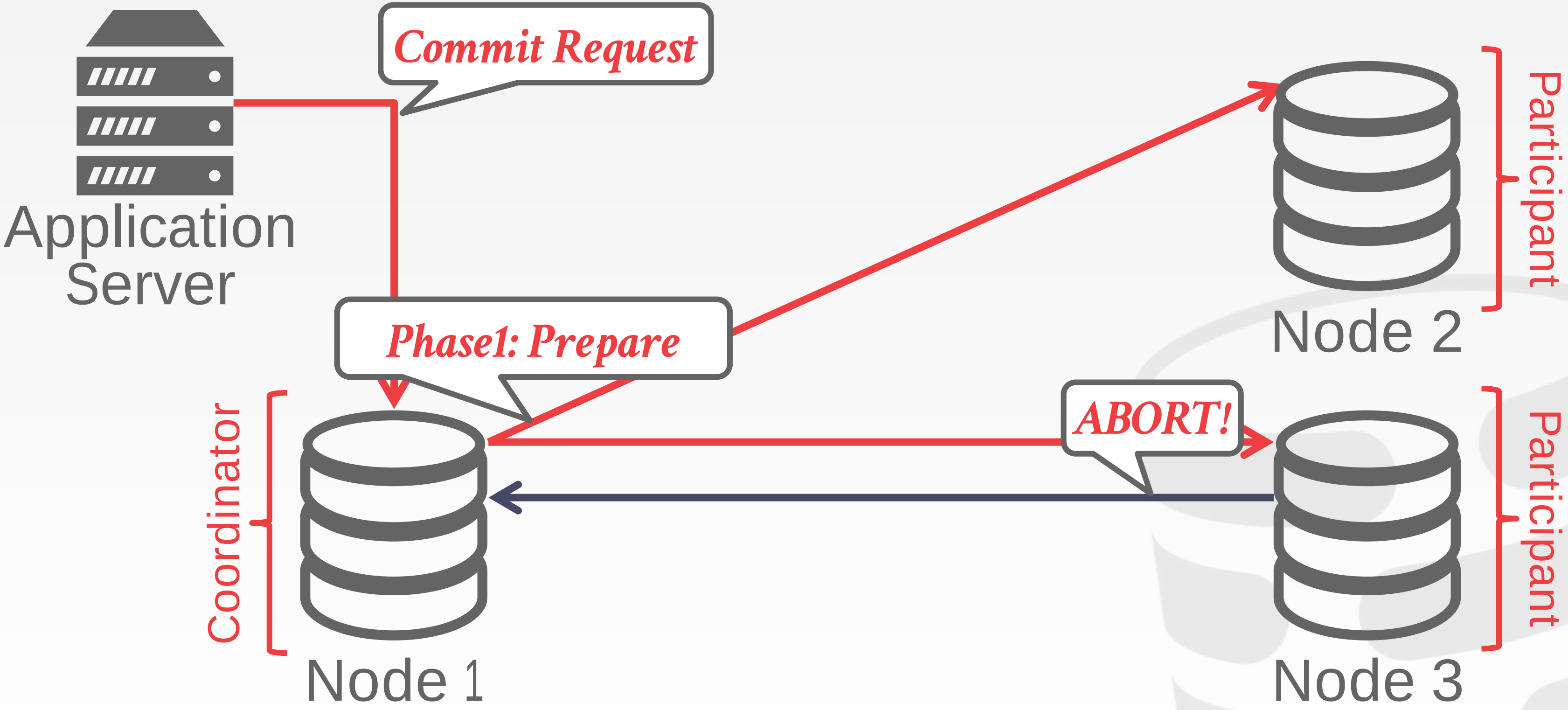
TWO-PHASE COMMIT (ABORT)



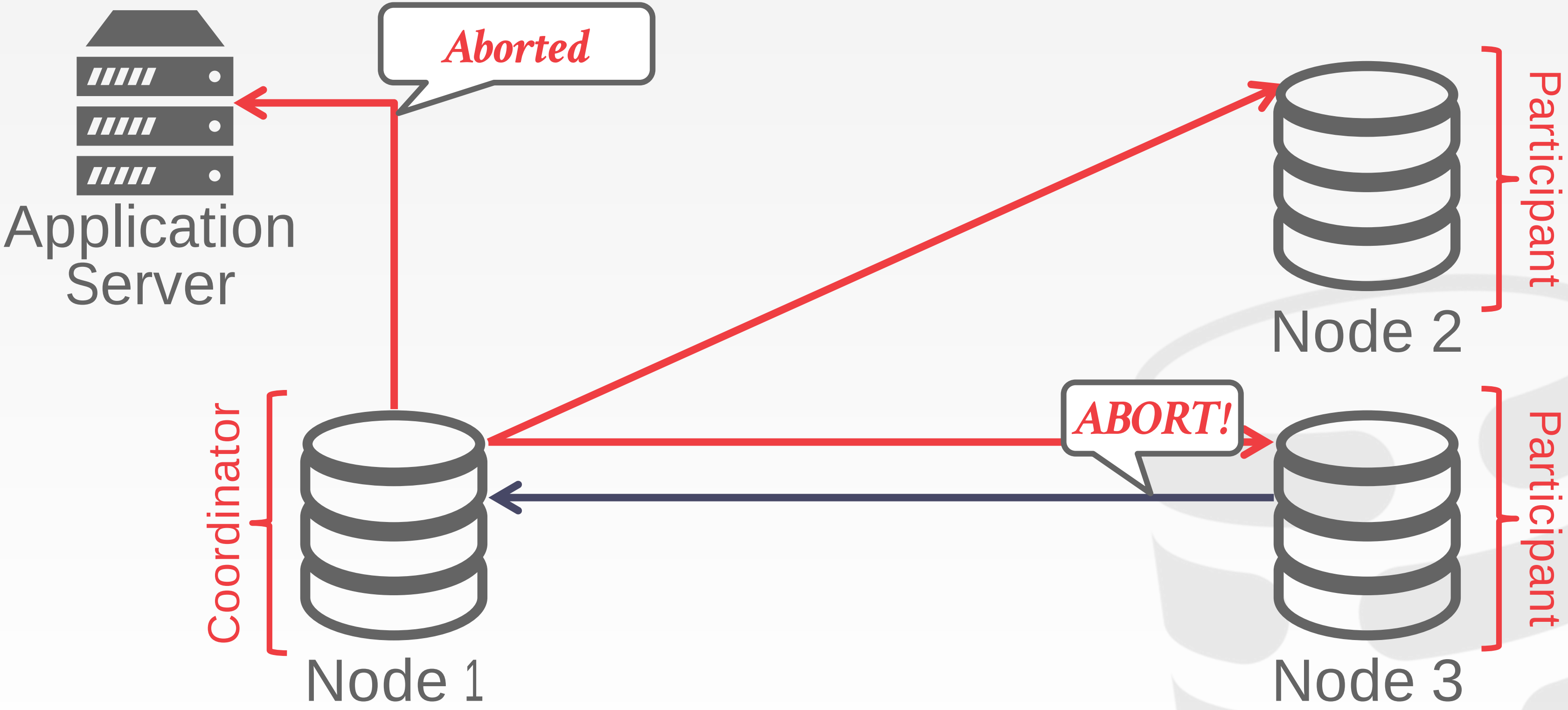
TWO-PHASE COMMIT (ABORT)



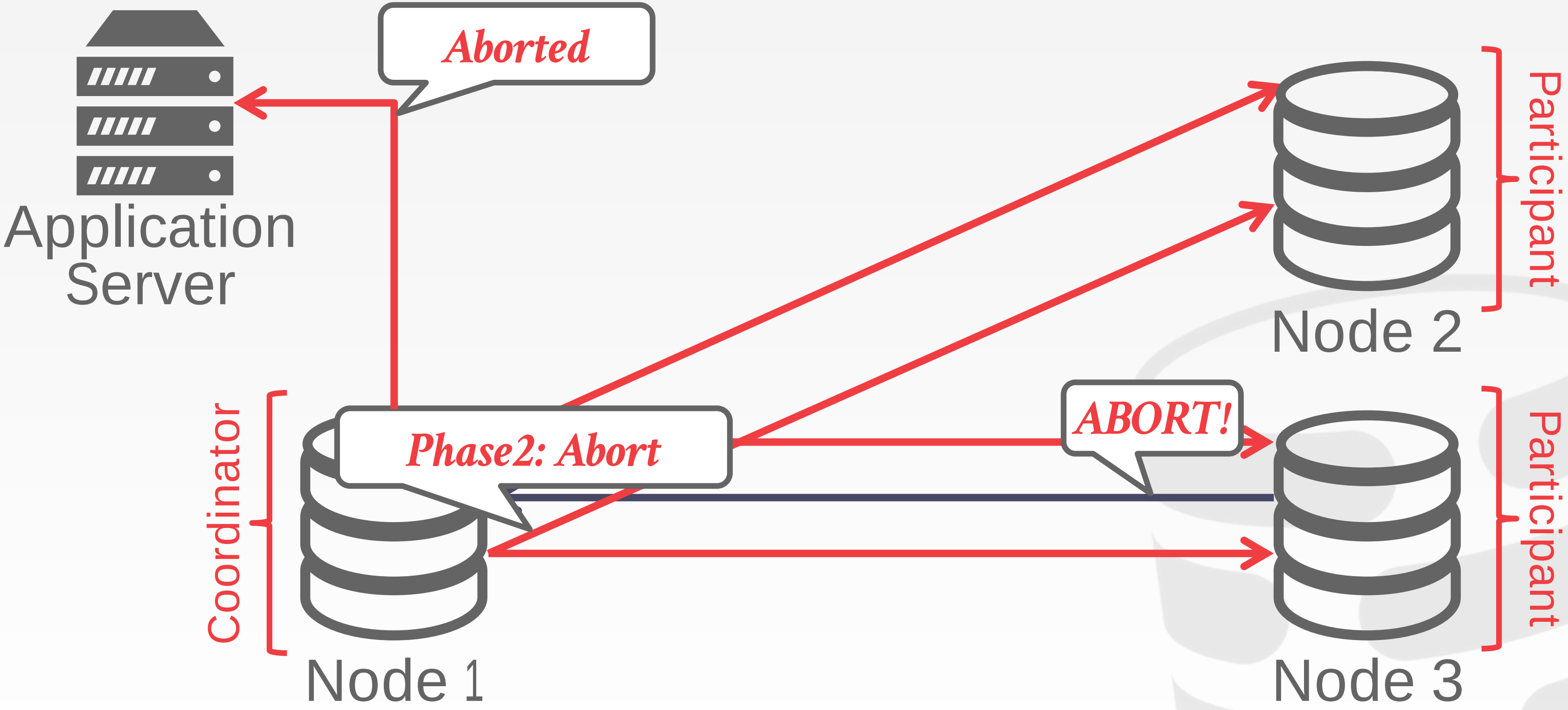
TWO-PHASE COMMIT (ABORT)



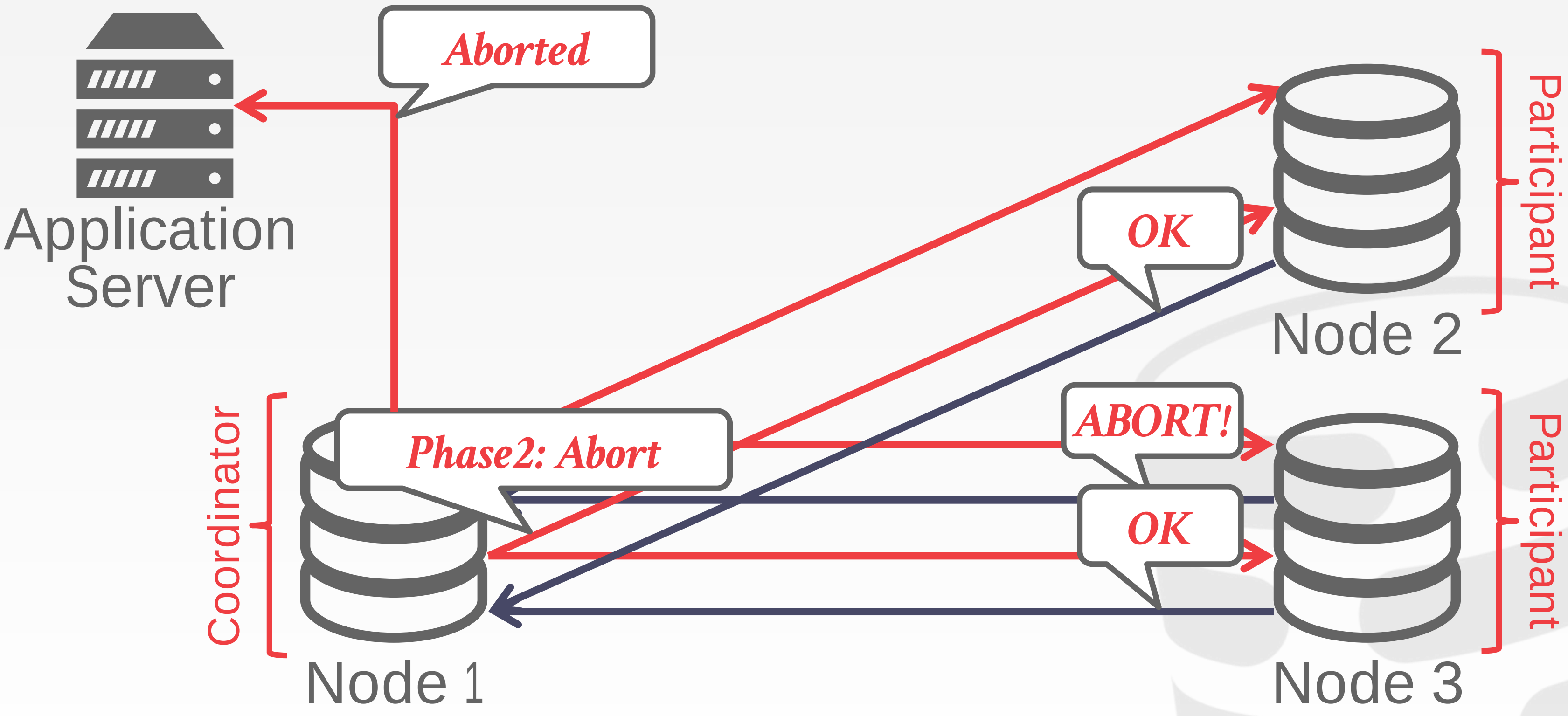
TWO-PHASE COMMIT (ABORT)



TWO-PHASE COMMIT (ABORT)



TWO-PHASE COMMIT (ABORT)



2PC OPTIMIZATIONS

Early Prepare Voting

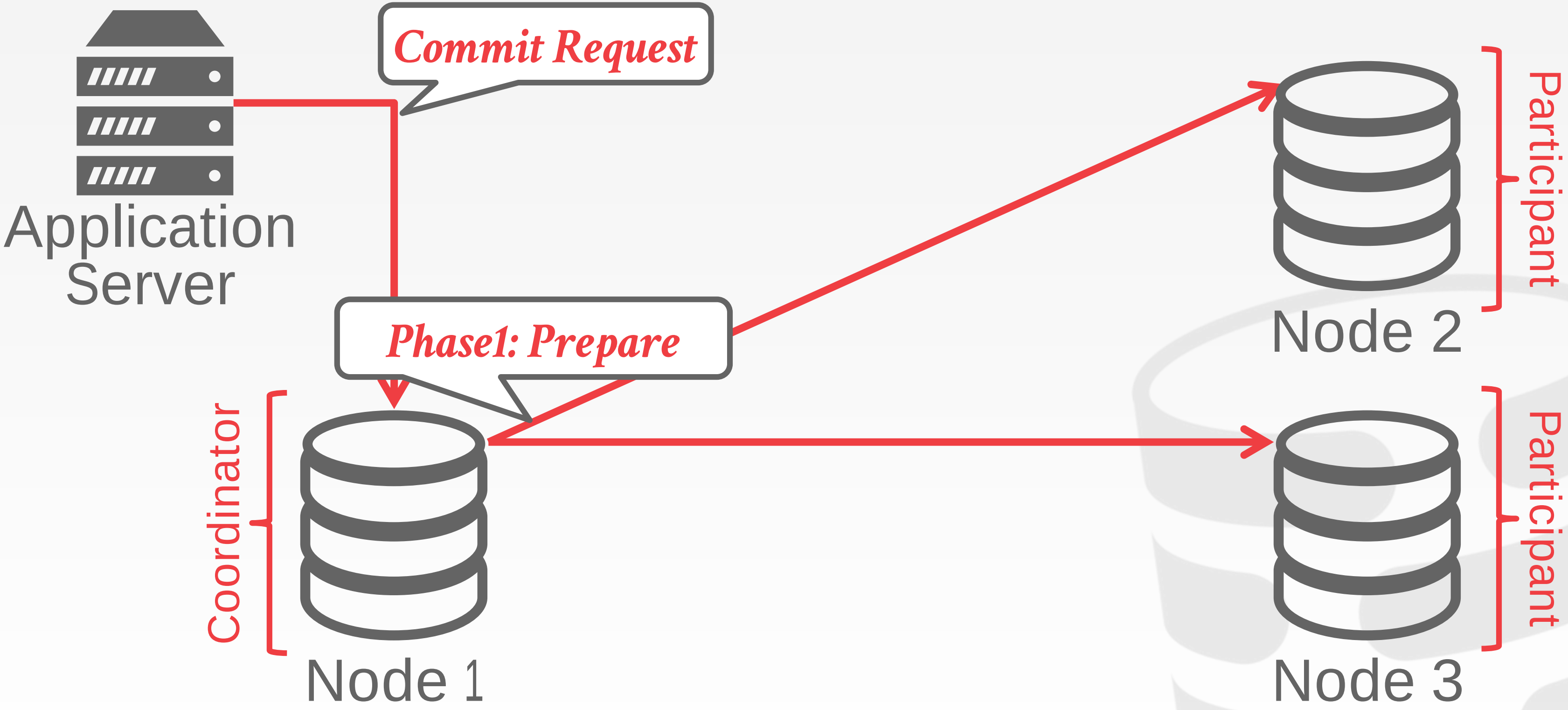
→ If you send a query to a remote node that you know will be the last one you execute there, then that node will also return their vote for the prepare phase with the query result.

Early Acknowledgement After Prepare

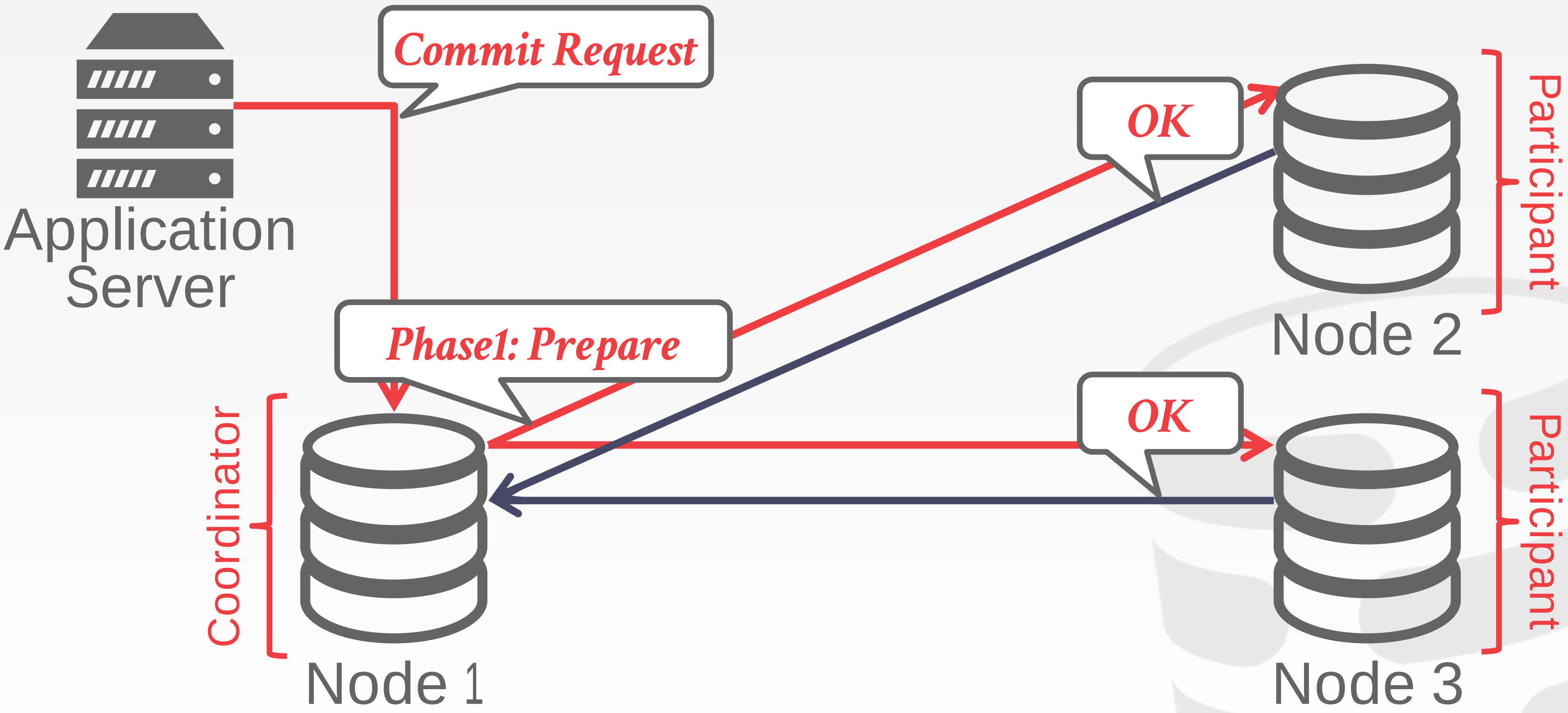
→ If all nodes vote to commit a txn, the coordinator can send the client an acknowledgement that their txn was successful before the commit phase finishes.

EARLY ACKNOWLEDGEMENT

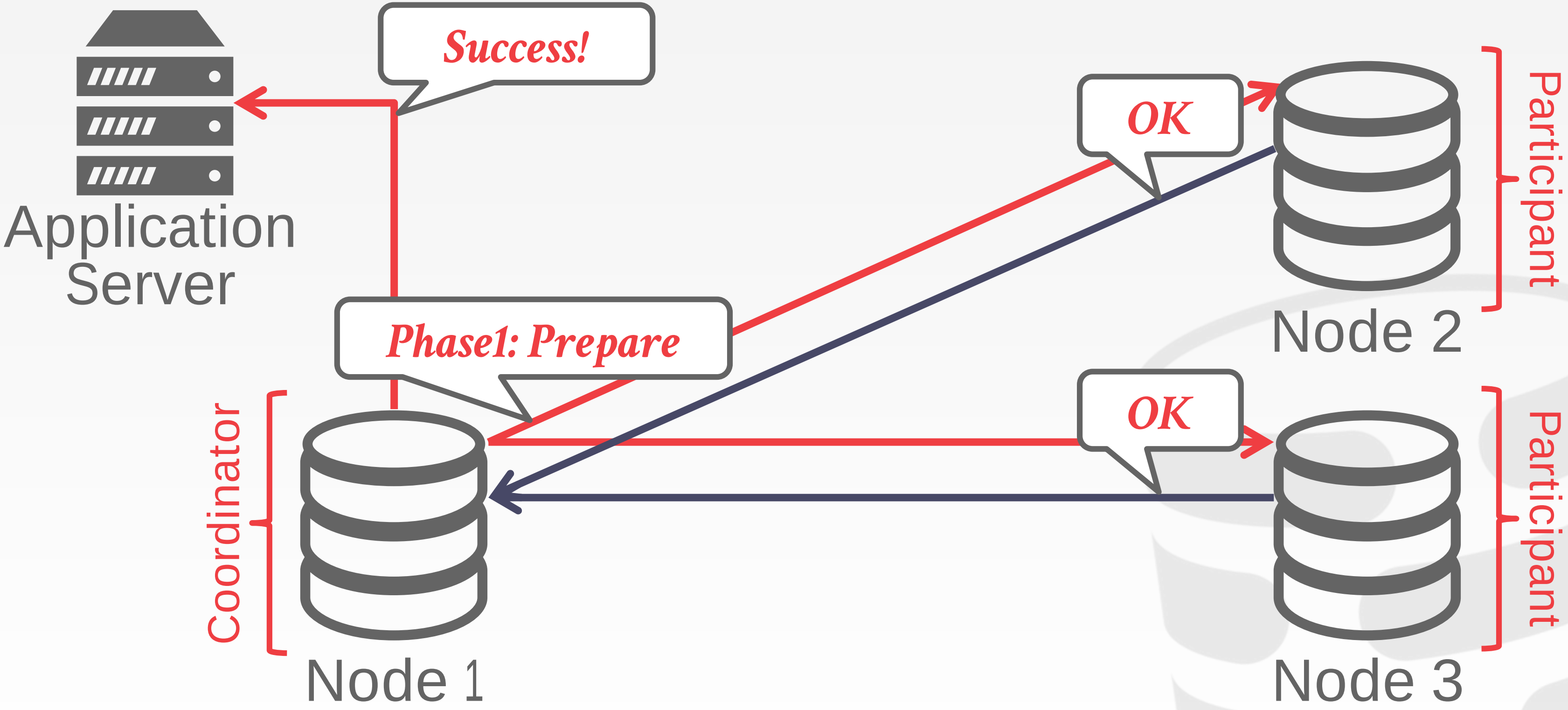
EARLY ACKNOWLEDGEMENT



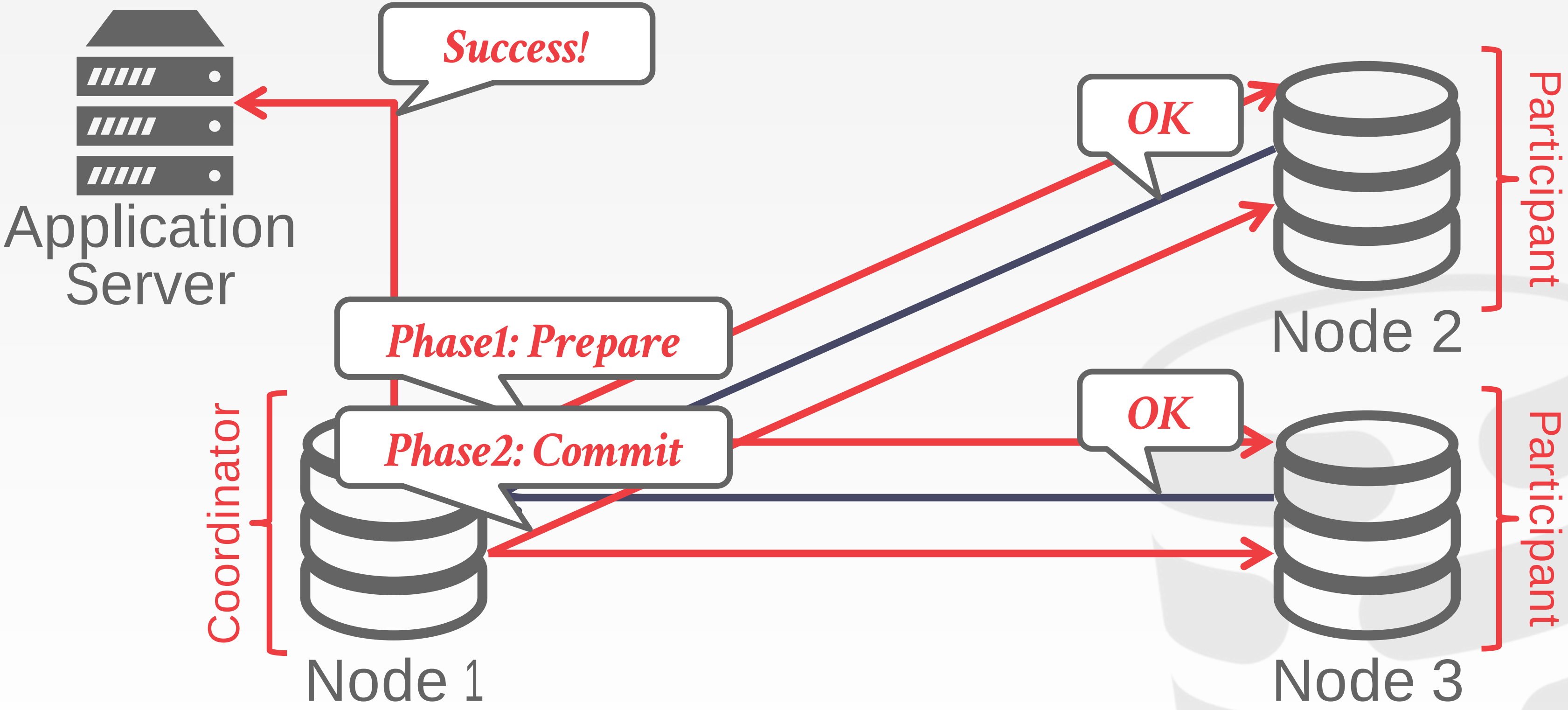
EARLY ACKNOWLEDGEMENT



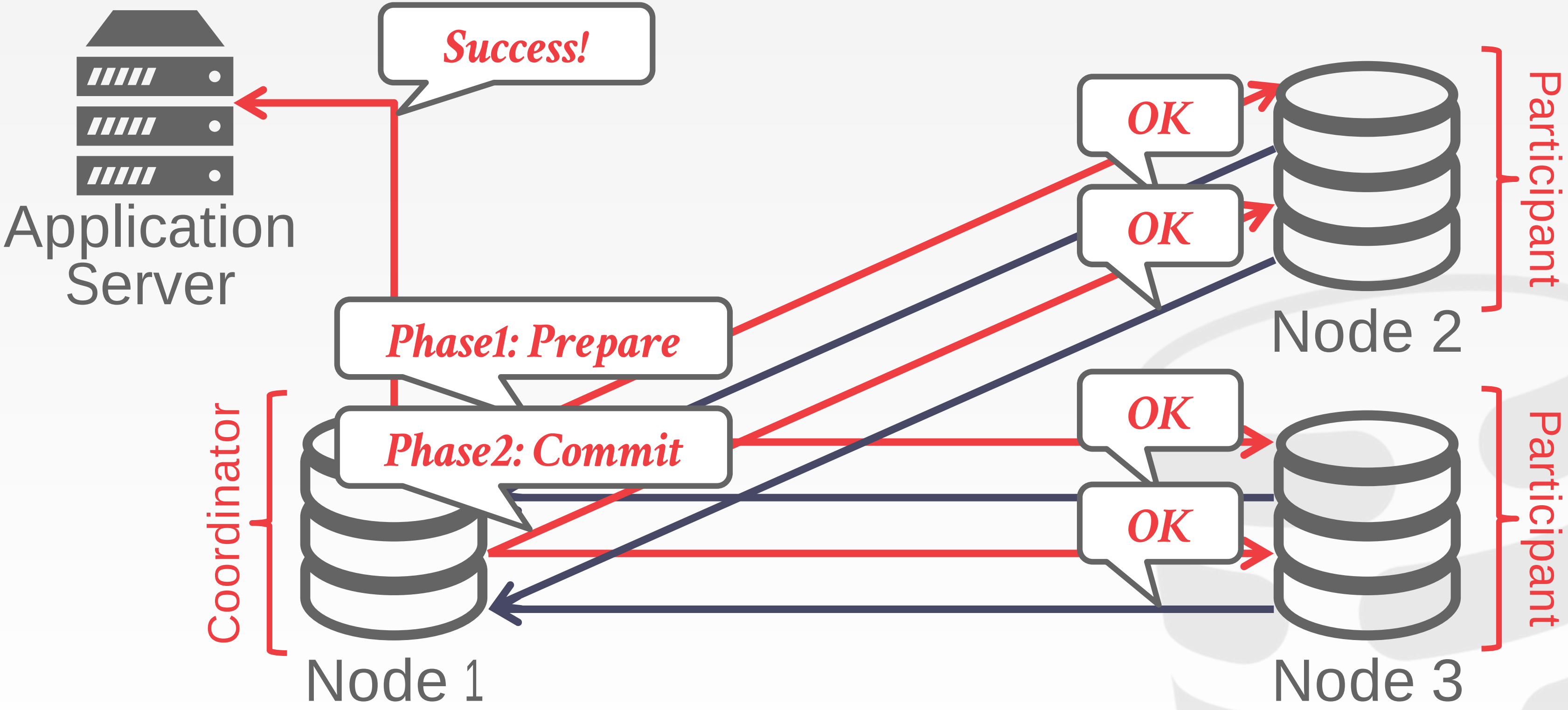
EARLY ACKNOWLEDGEMENT



EARLY ACKNOWLEDGEMENT



EARLY ACKNOWLEDGEMENT



TWO-PHASE COMMIT

Each node has to record the outcome of each phase in a stable storage log.

What happens if coordinator crashes?

→ Participants have to decide what to do.

What happens if participant crashes?

→ Coordinator assumes that it responded with an abort if it hasn't sent an acknowledgement yet.

The nodes have to block until they can figure out the correct action to take.

PAXOS

Consensus protocol where a coordinator proposes an outcome (e.g., commit or abort) and then the participants vote on whether that outcome should succeed.

Does not block if a majority of participants are available and has provably minimal message delays in the best case.

→ First correct protocol that was provably resilient in the face asynchronous networks

2PC VS. PAXOS

Two-Phase Commit

→ Blocks if coordinator fails after the prepare message is sent, until coordinator recovers.

Paxos

→ Non-blocking as long as a majority participants are alive, provided there is a sufficiently long period without further failures.

CONCLUSION

I have barely scratched the surface on distributed txn processing…

It is **really** hard to get right.

More info (and humiliation):
→ Kyle Kingsbury's Jepsen Project

NEXT CLASS

Replication

CAP Theorem

Real-World Examples