Carnegie Mellon University

21 22

Introduction to Distributed Databases

Intro to Database Systems
15-445/15-645
Fall 2021

LM
Lin Ma
Computer Science Carnegie Mellon University

ADMINISTRIVIA

Homework #5: Will be released on Monday Nov 22nd. It is due Dec 2nd @ 11:59pm.

Project #4: Will be released today. It is due Dec 5th @ 11:59pm.

UPCOMING DATABASE TALK

Fluree - Cloud-Native Ledger Graph Database

→ Mon Nov 15th @ 4:30pm ET

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

TODAY'S AGENDA

System Architectures

Design Issues

Partitioning Schemes

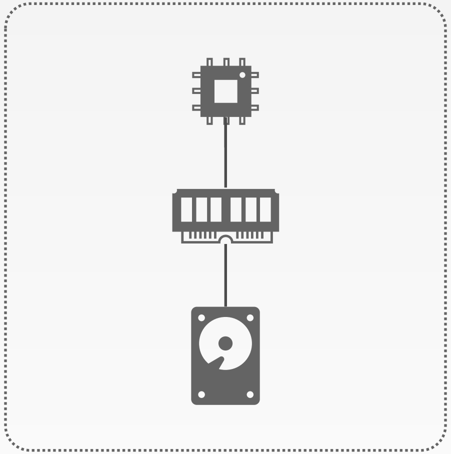
Distributed Concurrency Control

SYSTEM ARCHITECTURE

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

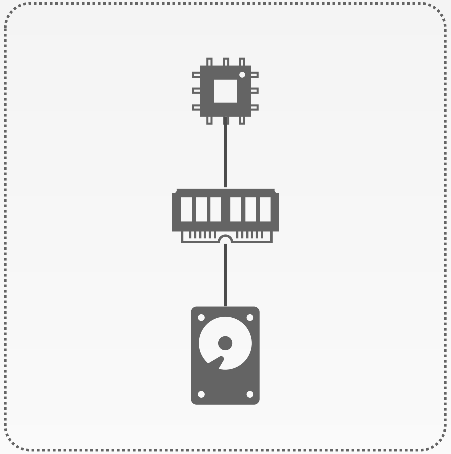
This affects how CPUs coordinate with each other and where they retrieve/store objects in the database.

SYSTEM ARCHITECTURE

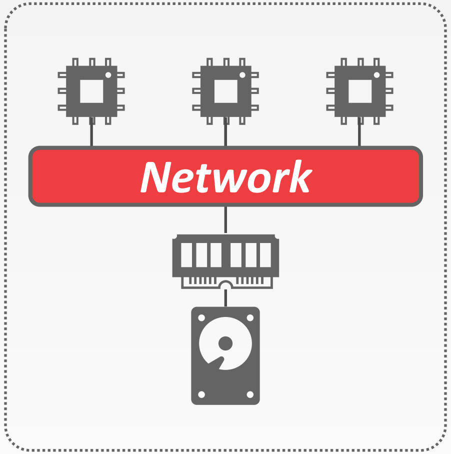


Shared Everything

SYSTEM ARCHITECTURE

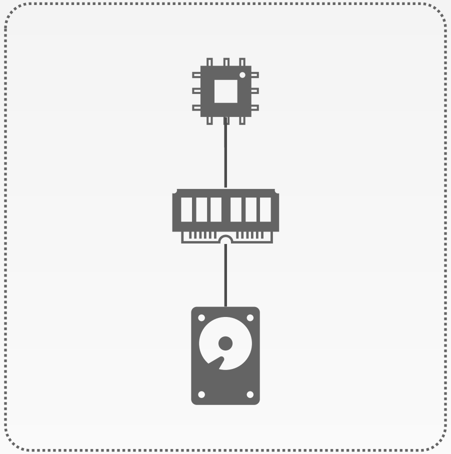


Shared Everything

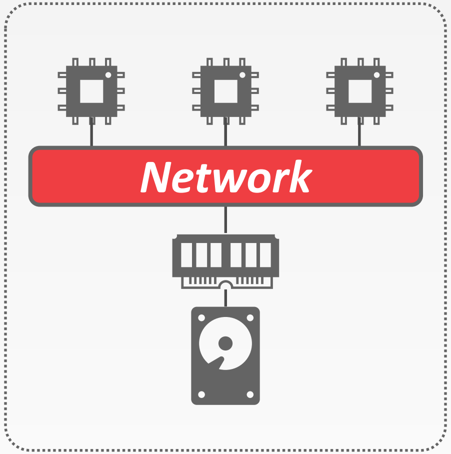


Shared Memory

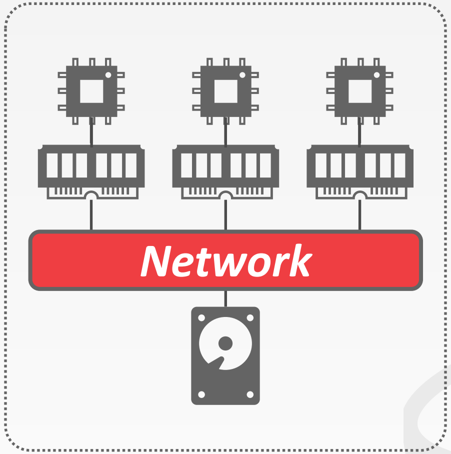
SYSTEM ARCHITECTURE



Shared Everything

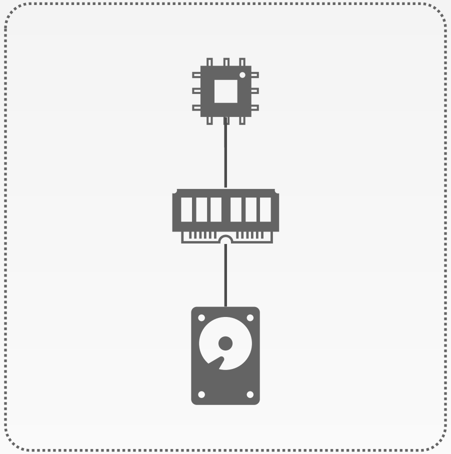


Shared Memory

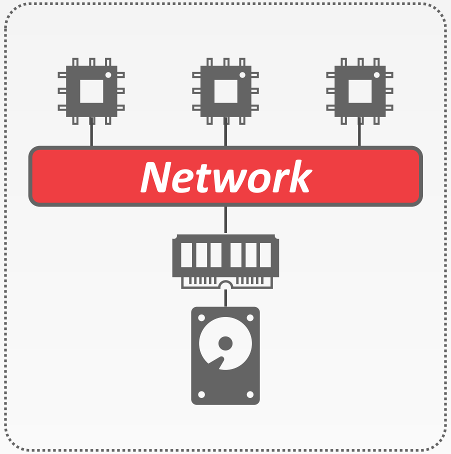


Shared Disk

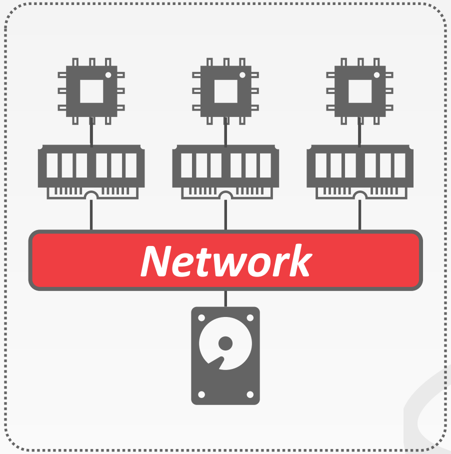
SYSTEM ARCHITECTURE



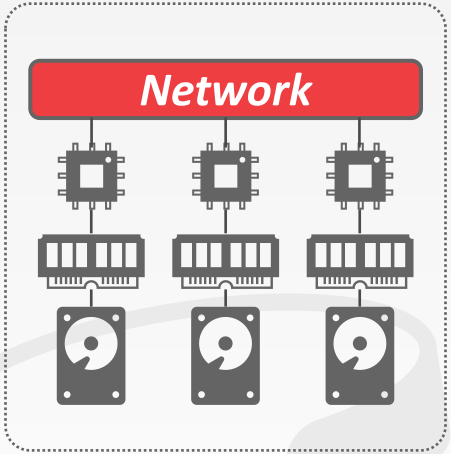
Shared Everything



Shared Memory



Shared Disk



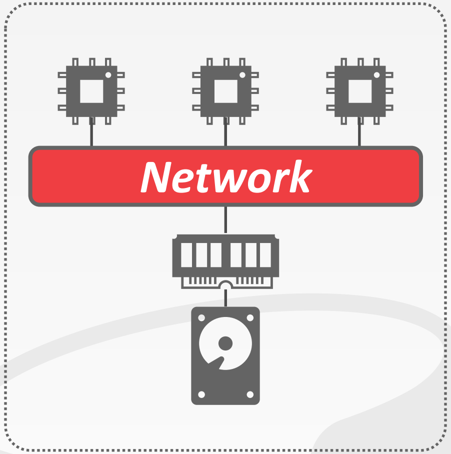
Shared Nothing

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.

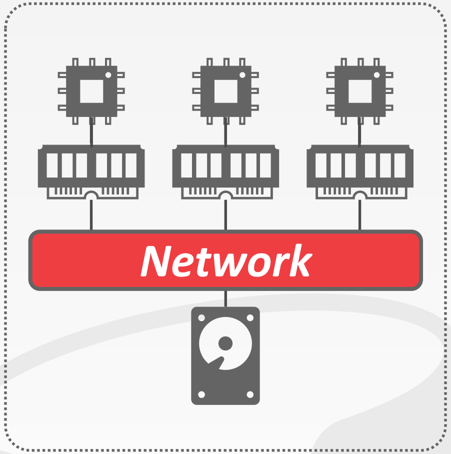


SHARED DISK

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.

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

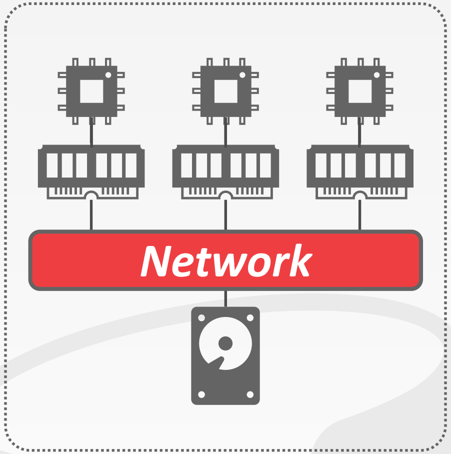


SHARED DISK

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

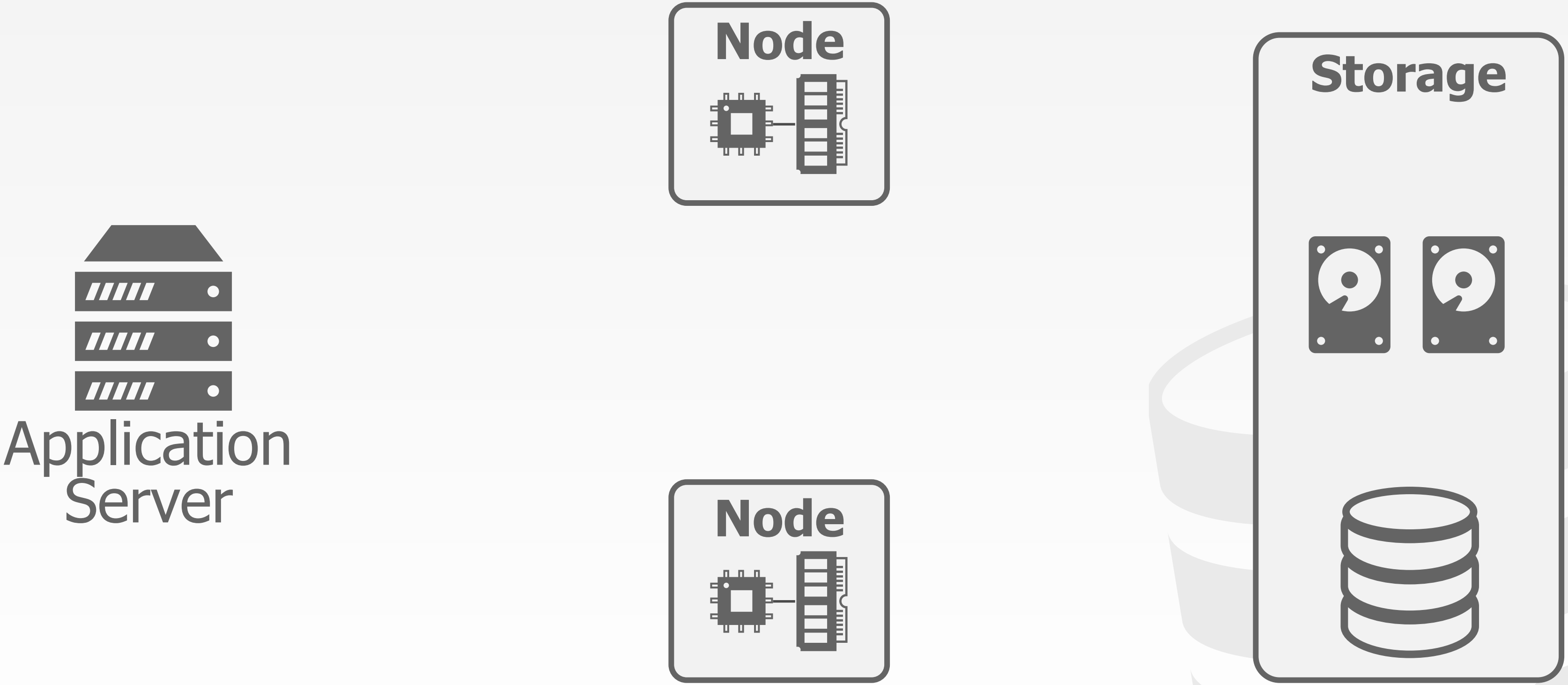
→ Can scale execution layer independently from the storage layer.

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

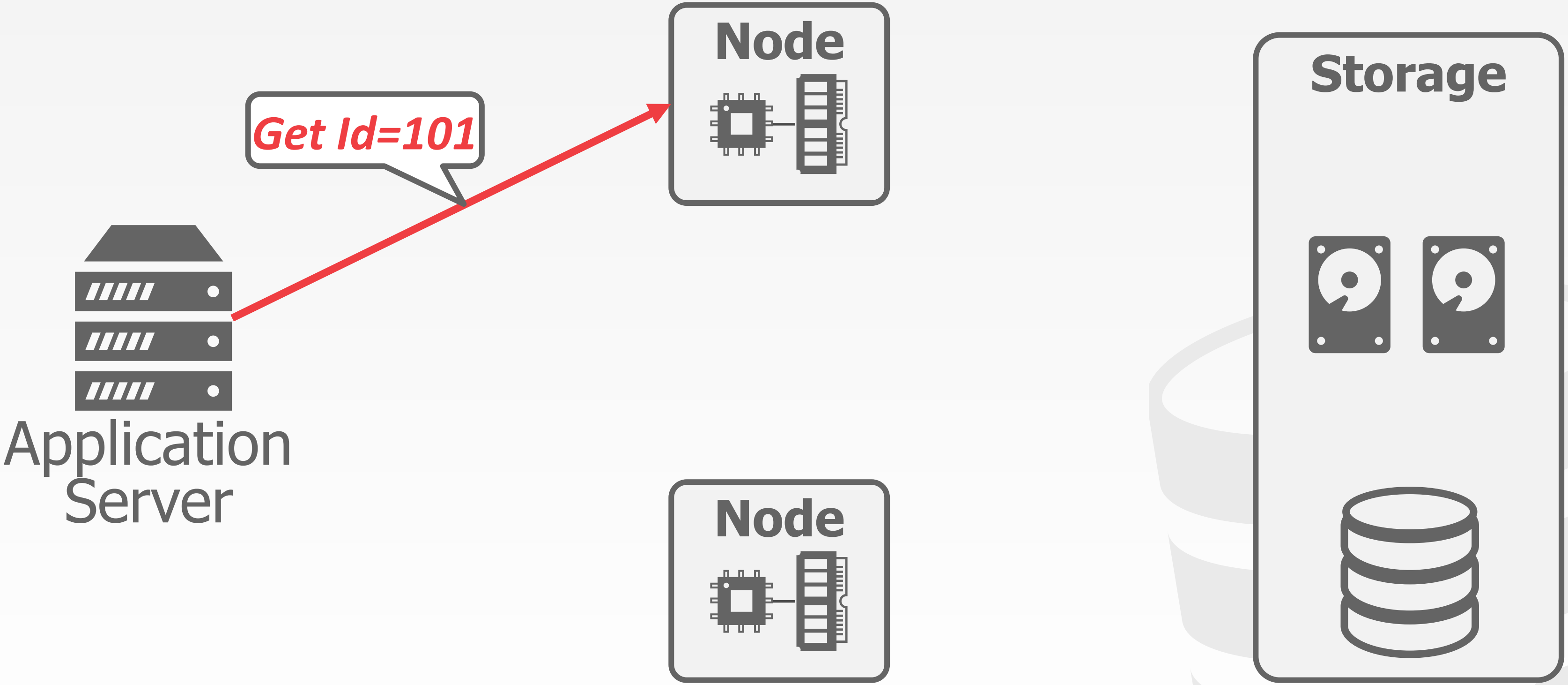


ROCKSET, ORACLE EXADATA, sqrrl, presto, cloudera IMPALA, NUODB, amazon REDSHIFT, YugaByte, ORACLE RAC, splice MACHINE, Hortonworks STINGER, APACHE HBASE, snowflake, Google Spanner, Amazon Aurora

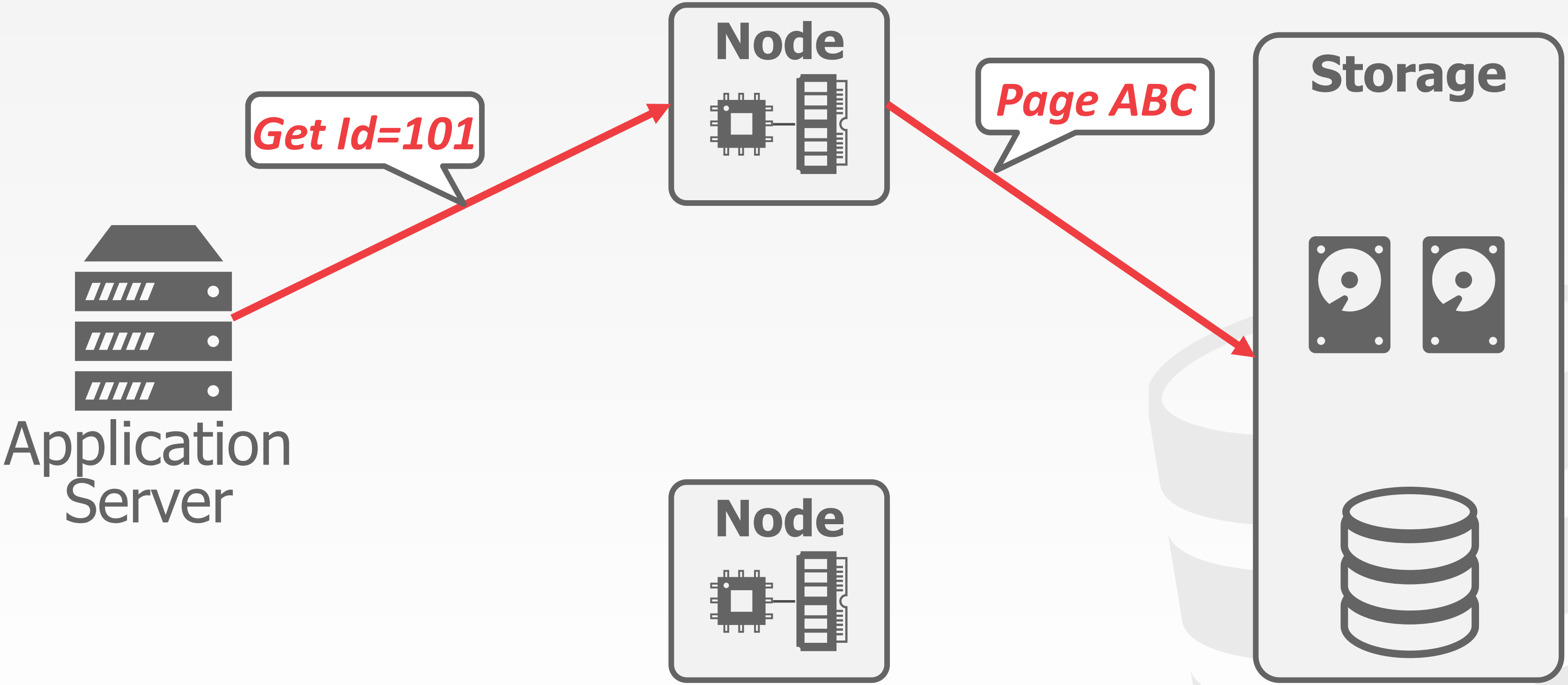
SHARED DISK EXAMPLE



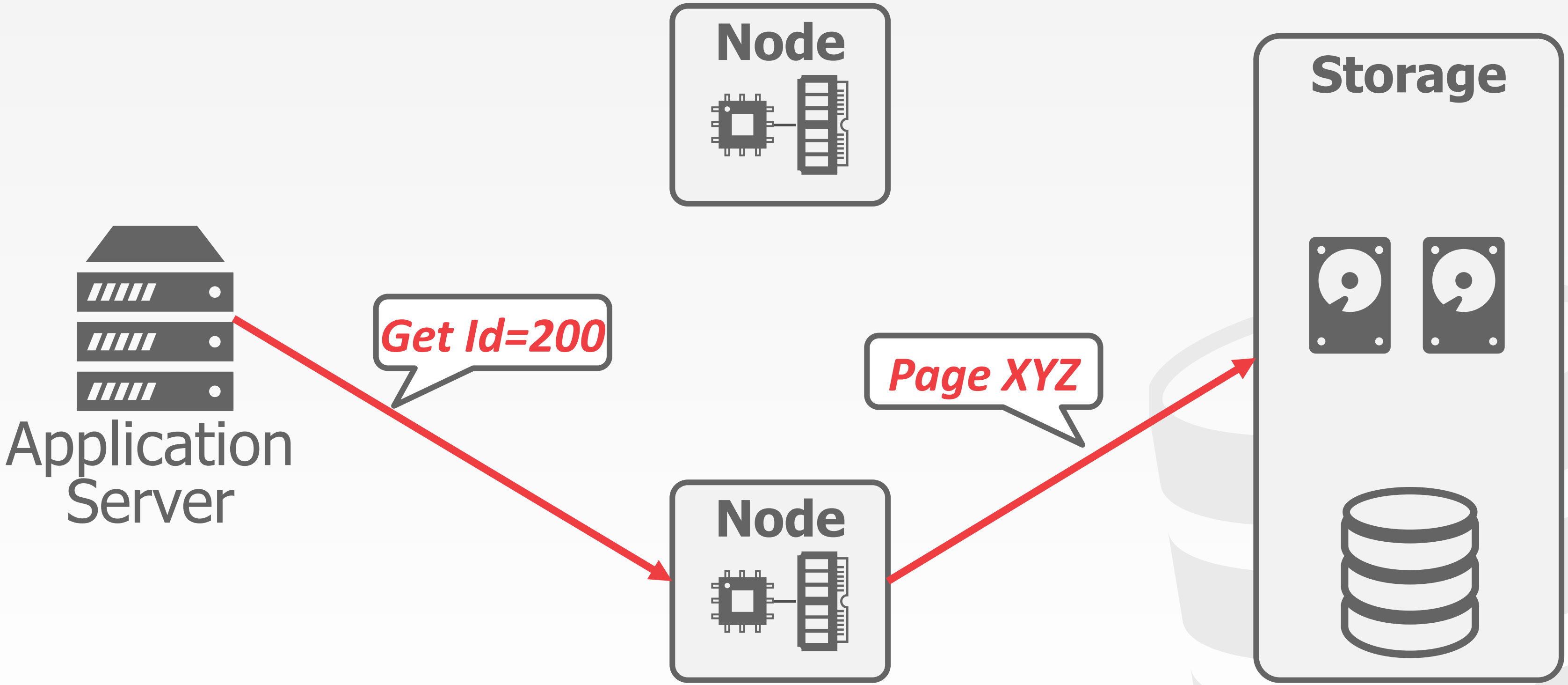
SHARED DISK EXAMPLE



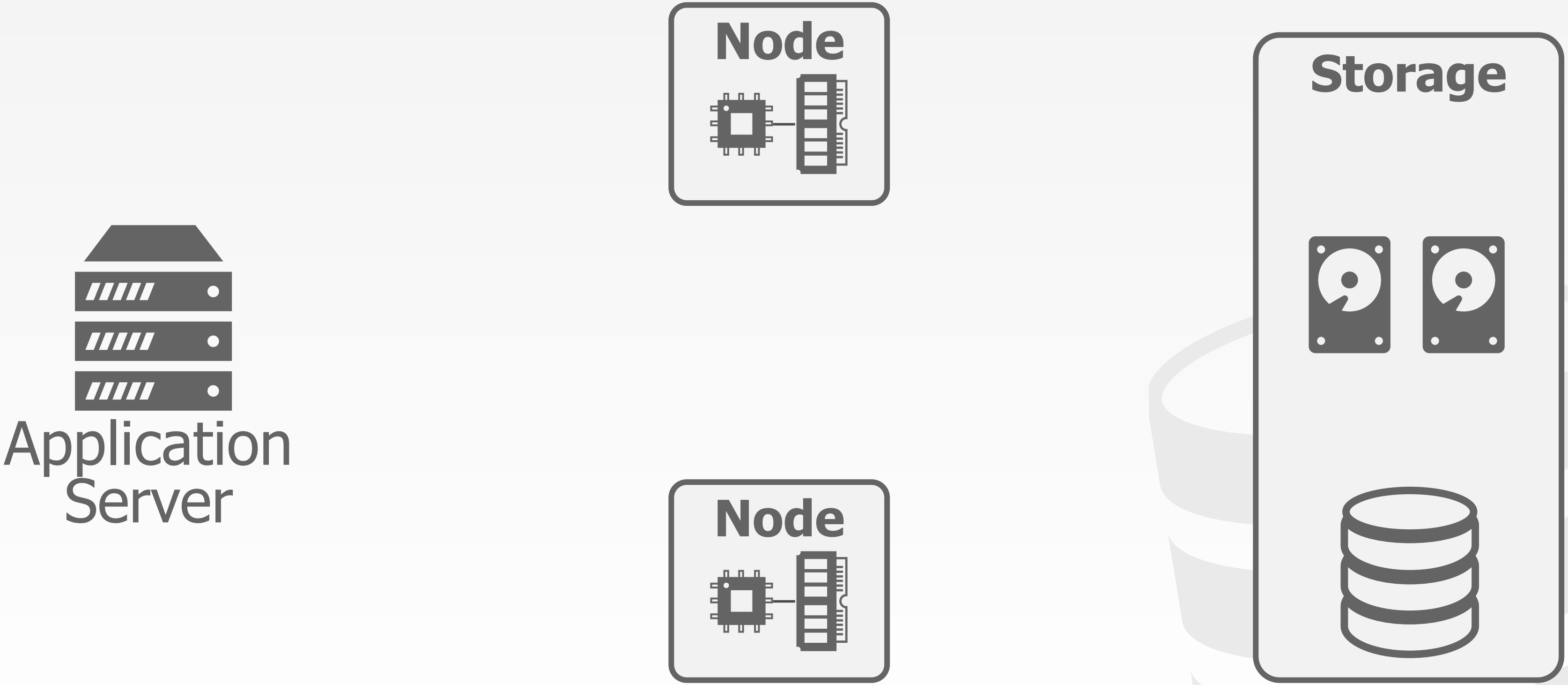
SHARED DISK EXAMPLE



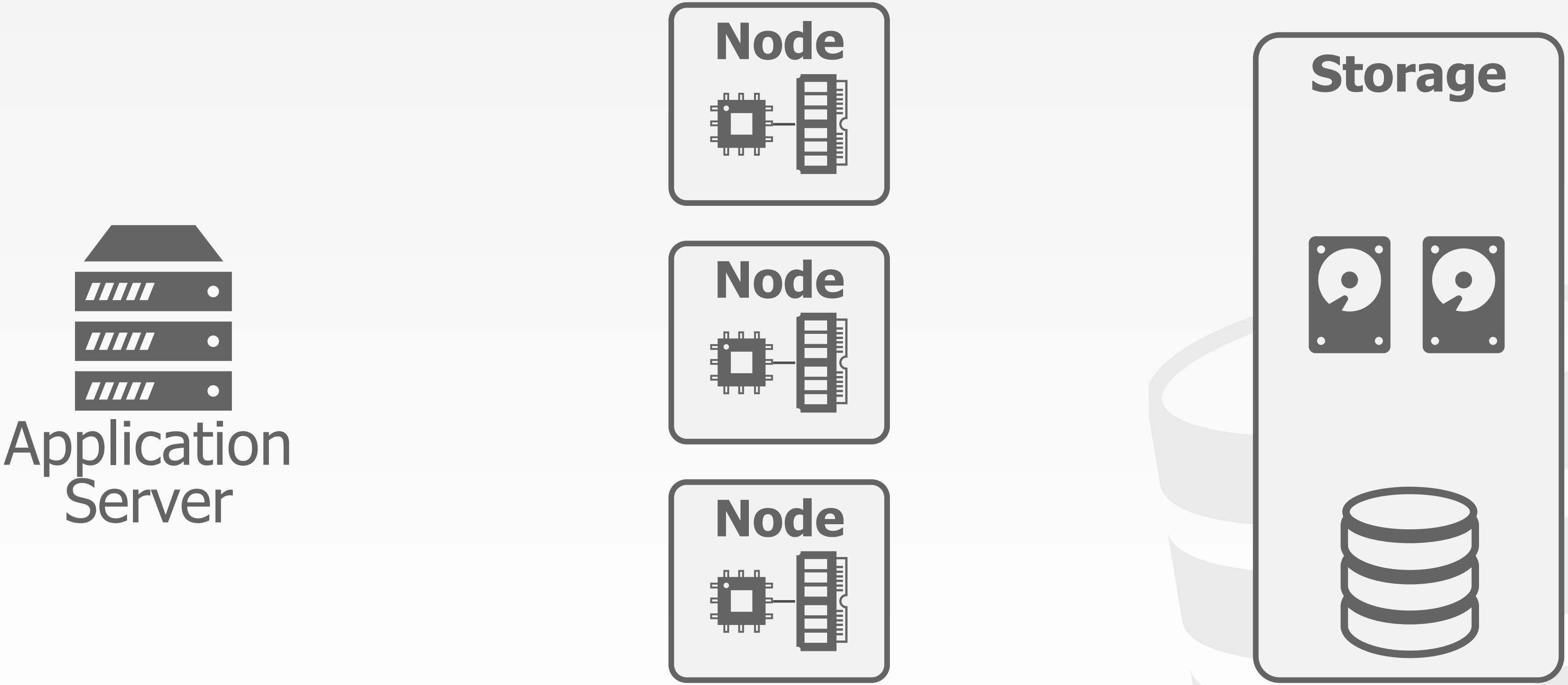
SHARED DISK EXAMPLE



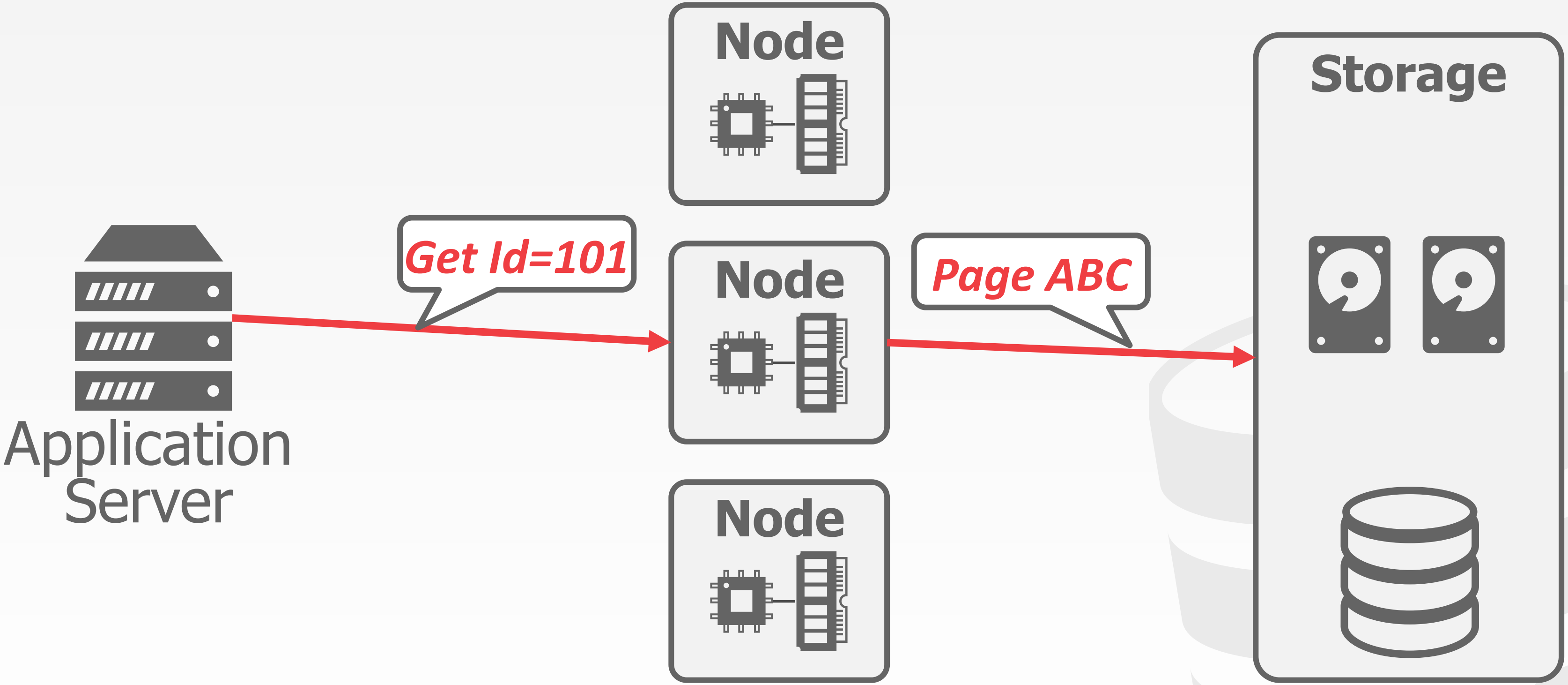
SHARED DISK EXAMPLE



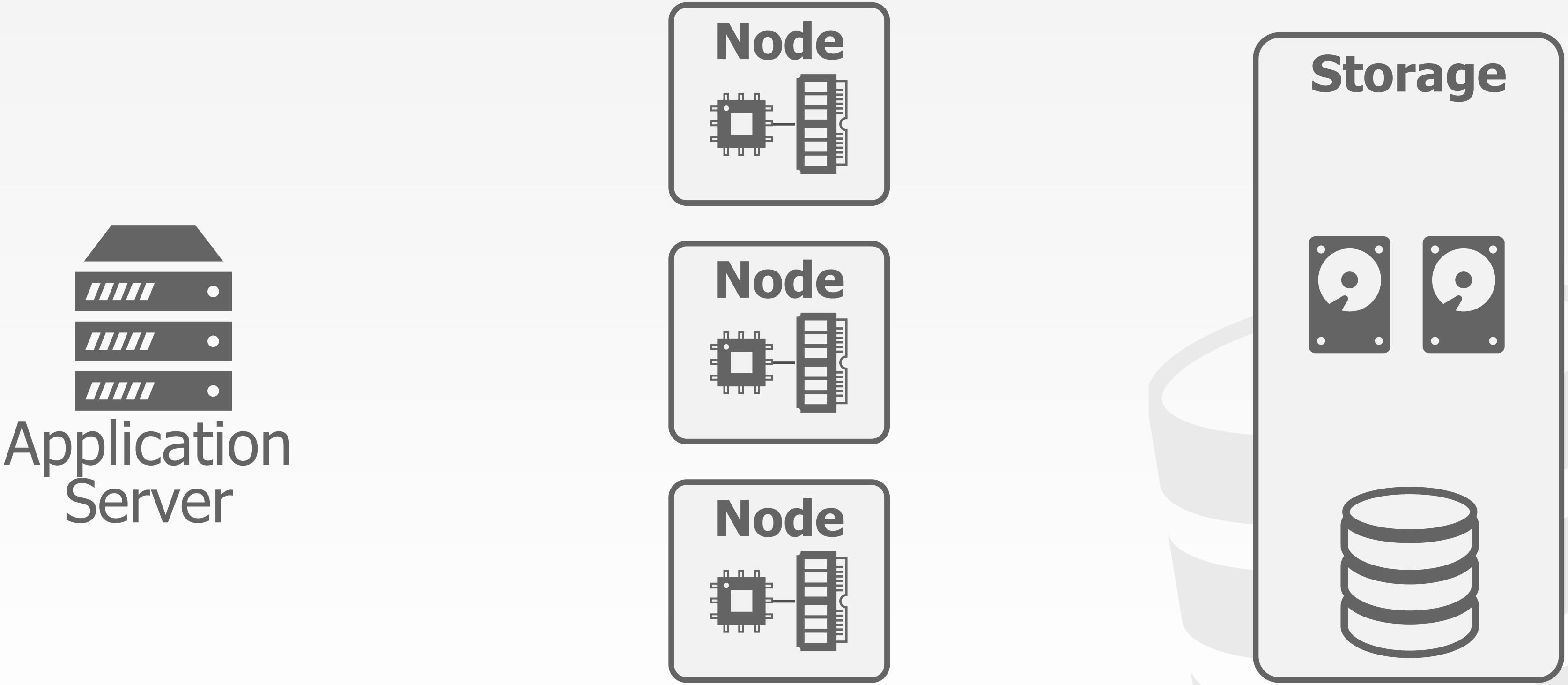
SHARED DISK EXAMPLE



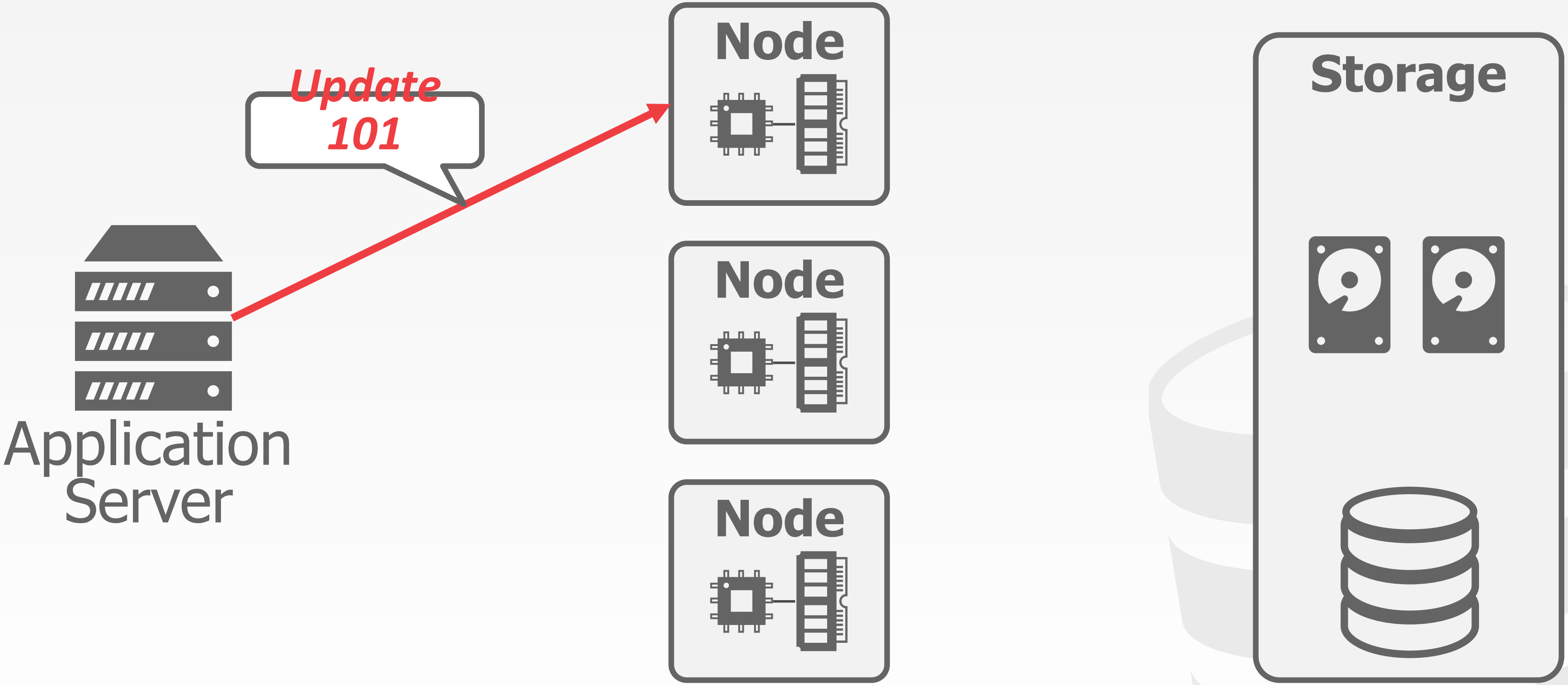
SHARED DISK EXAMPLE



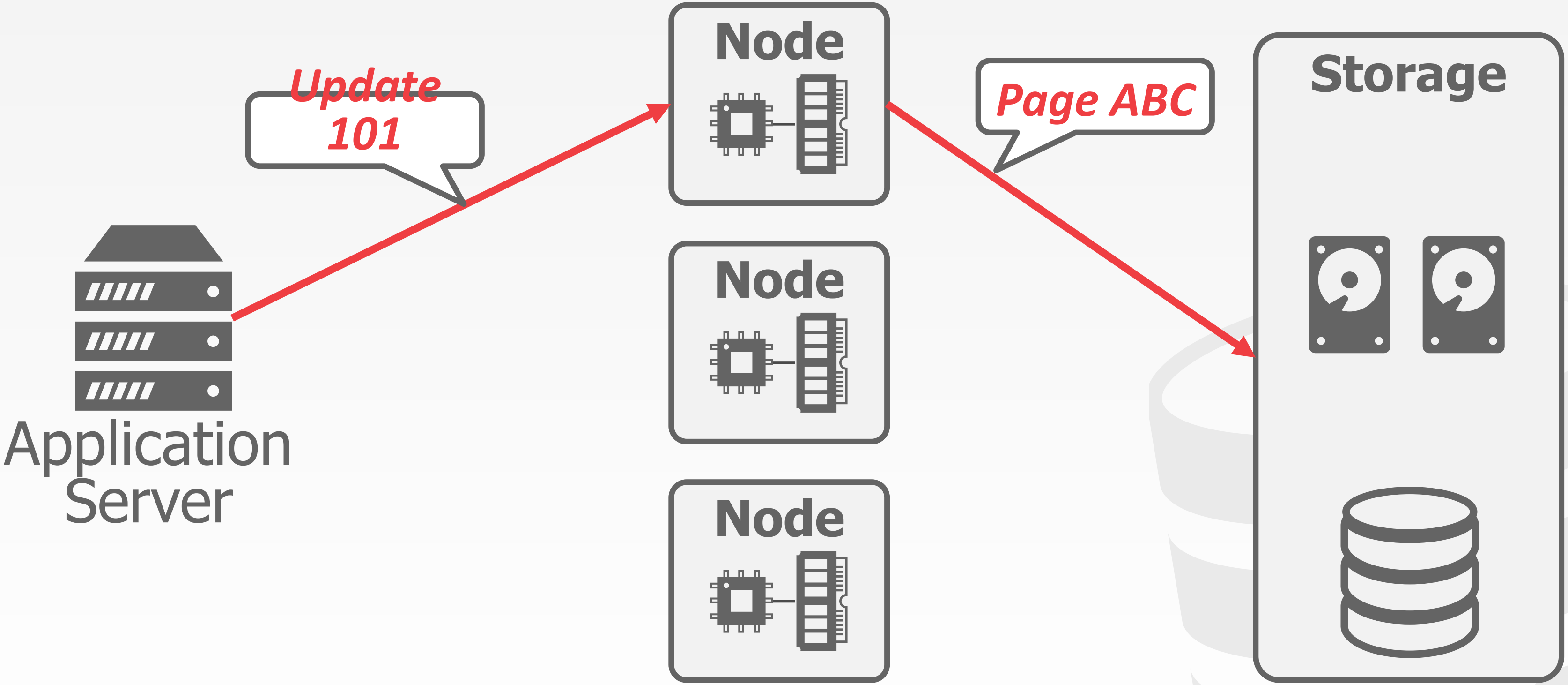
SHARED DISK EXAMPLE



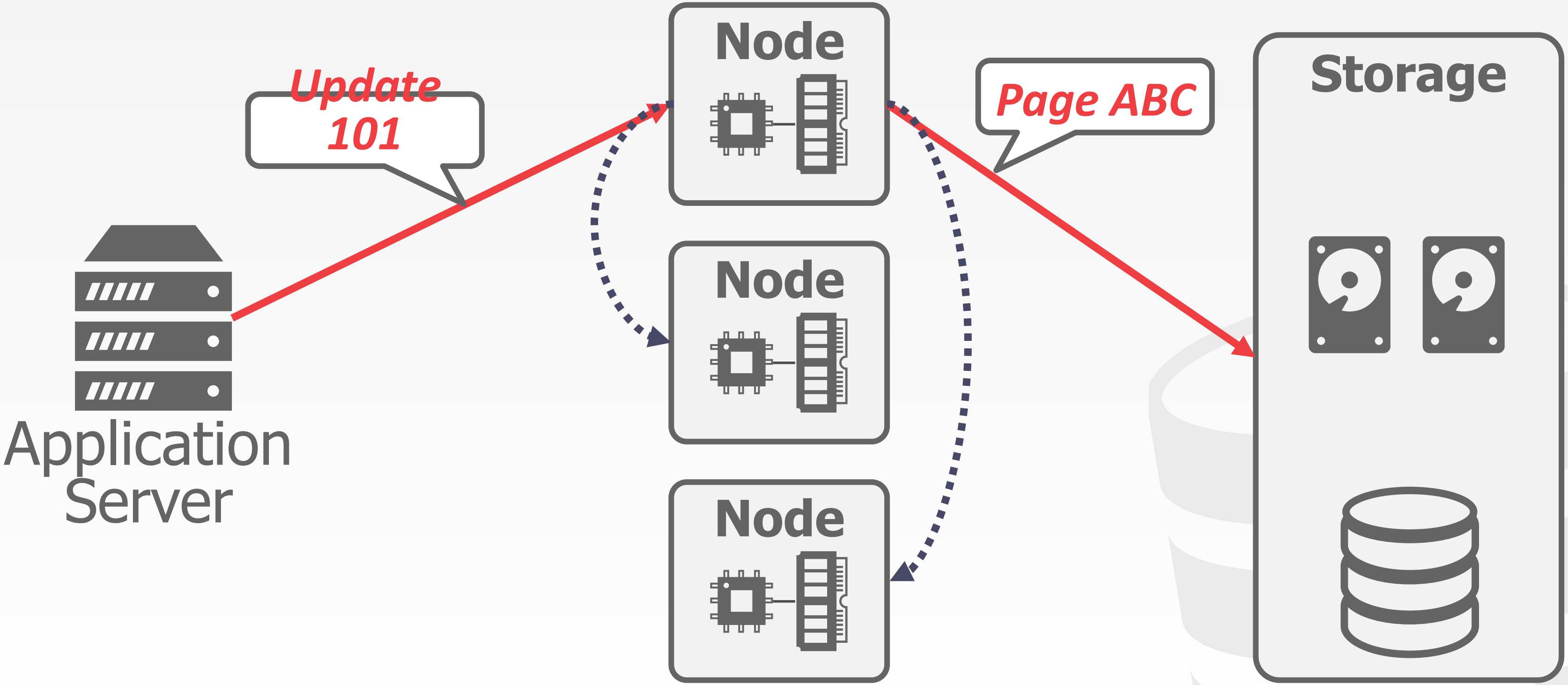
SHARED DISK EXAMPLE



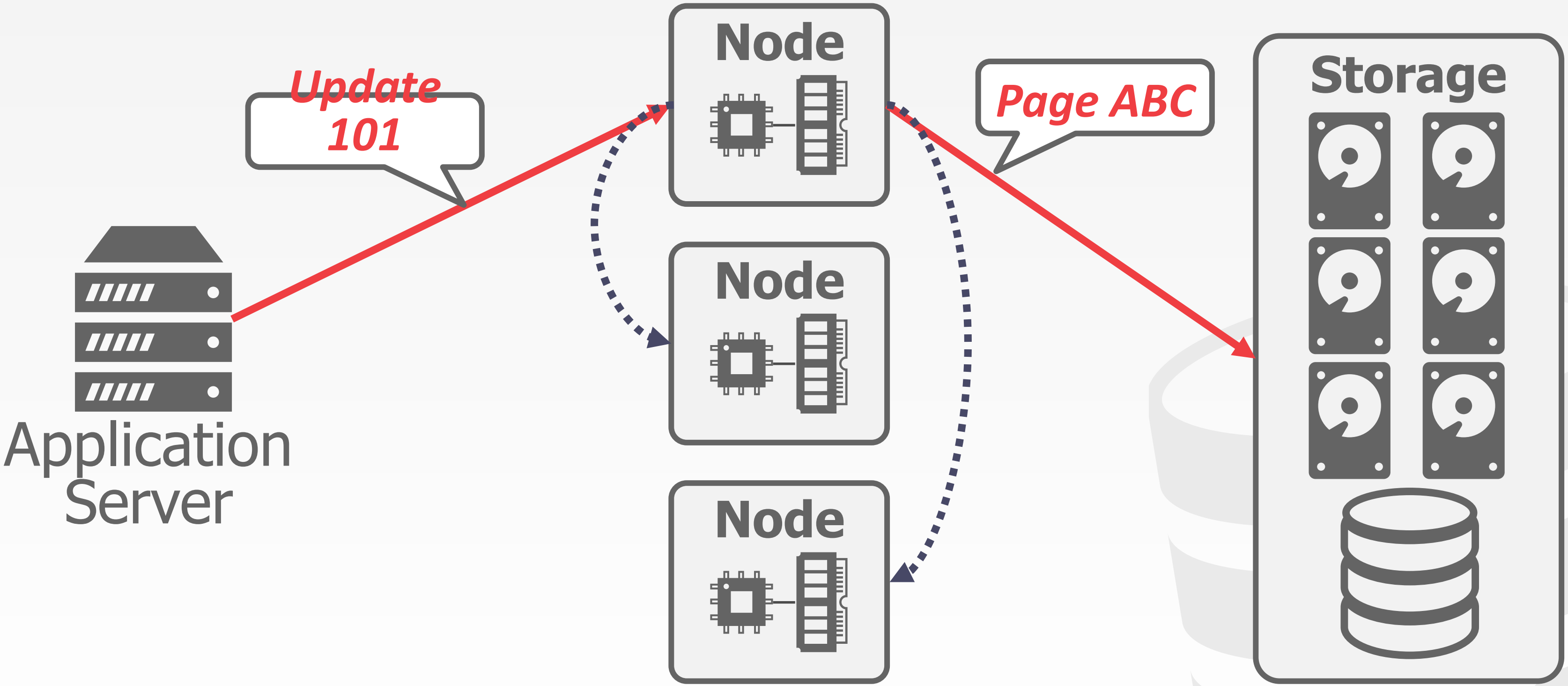
SHARED DISK EXAMPLE



SHARED DISK EXAMPLE



SHARED DISK EXAMPLE

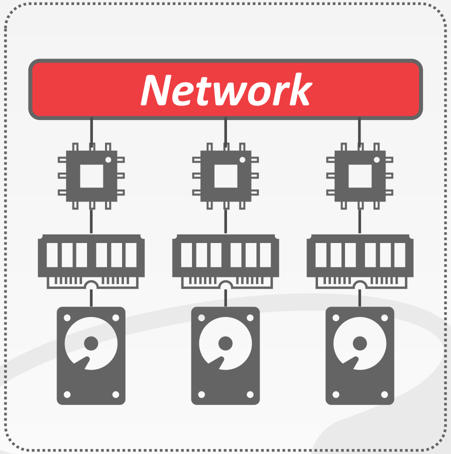


SHARED NOTHING

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

Nodes only communicate with each other via network.

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



SHARED NOTHING

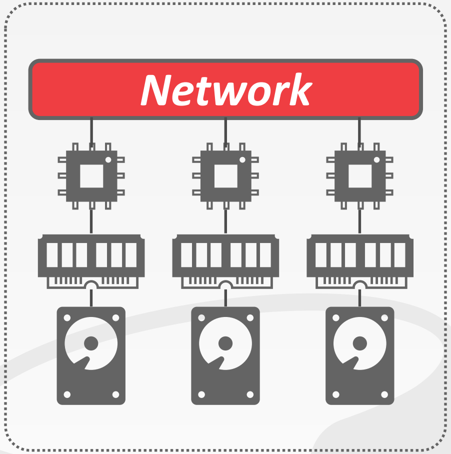
Each DBMS instance has its own CPU, memory, and disk.

Nodes only communicate with each other via network.

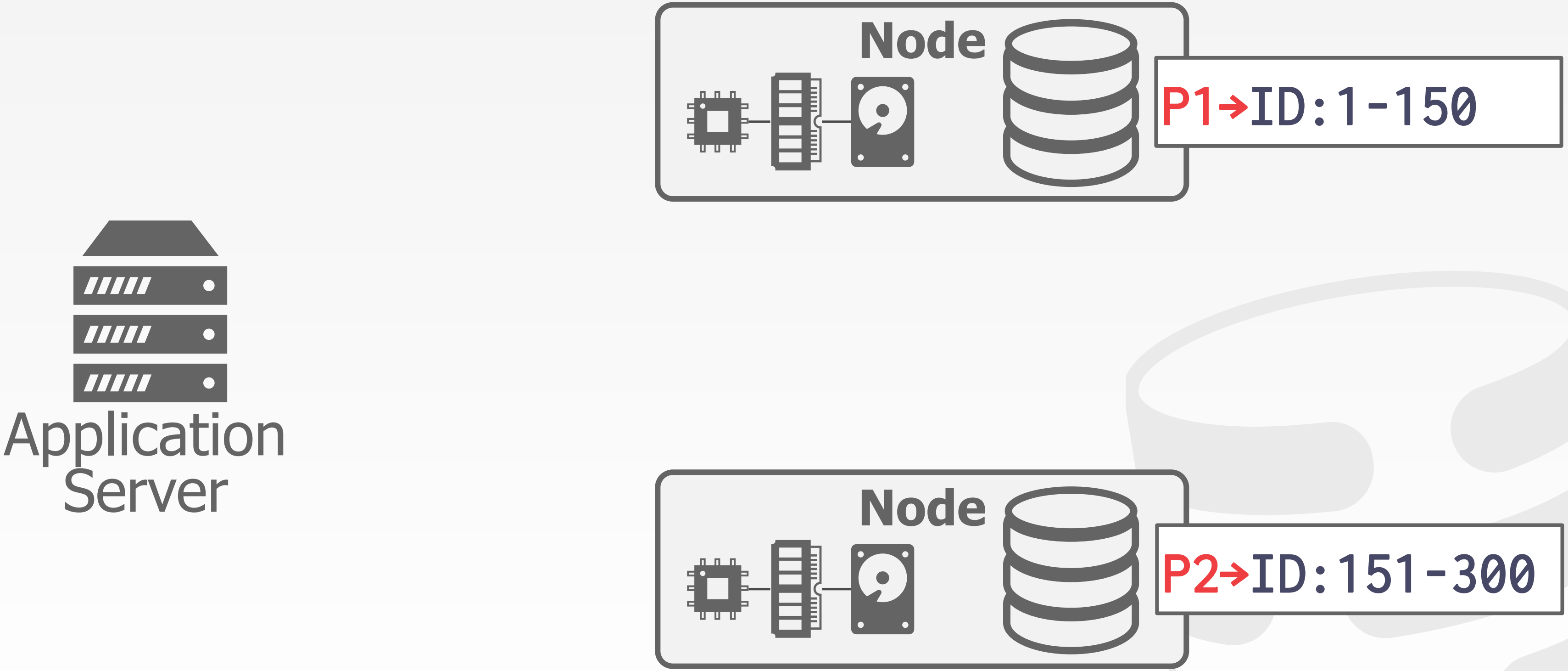
→ Harder to scale capacity.

→ Harder to ensure consistency.

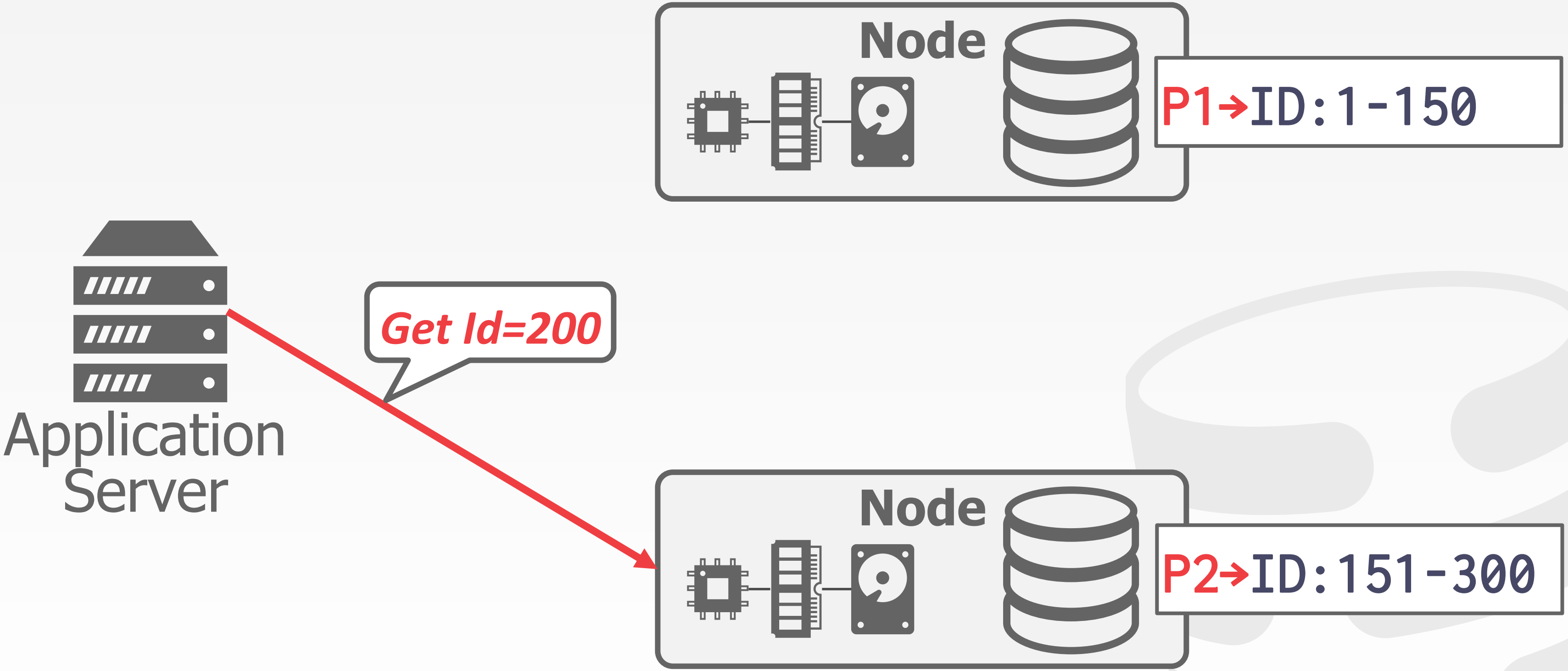
→ Better performance & efficiency.



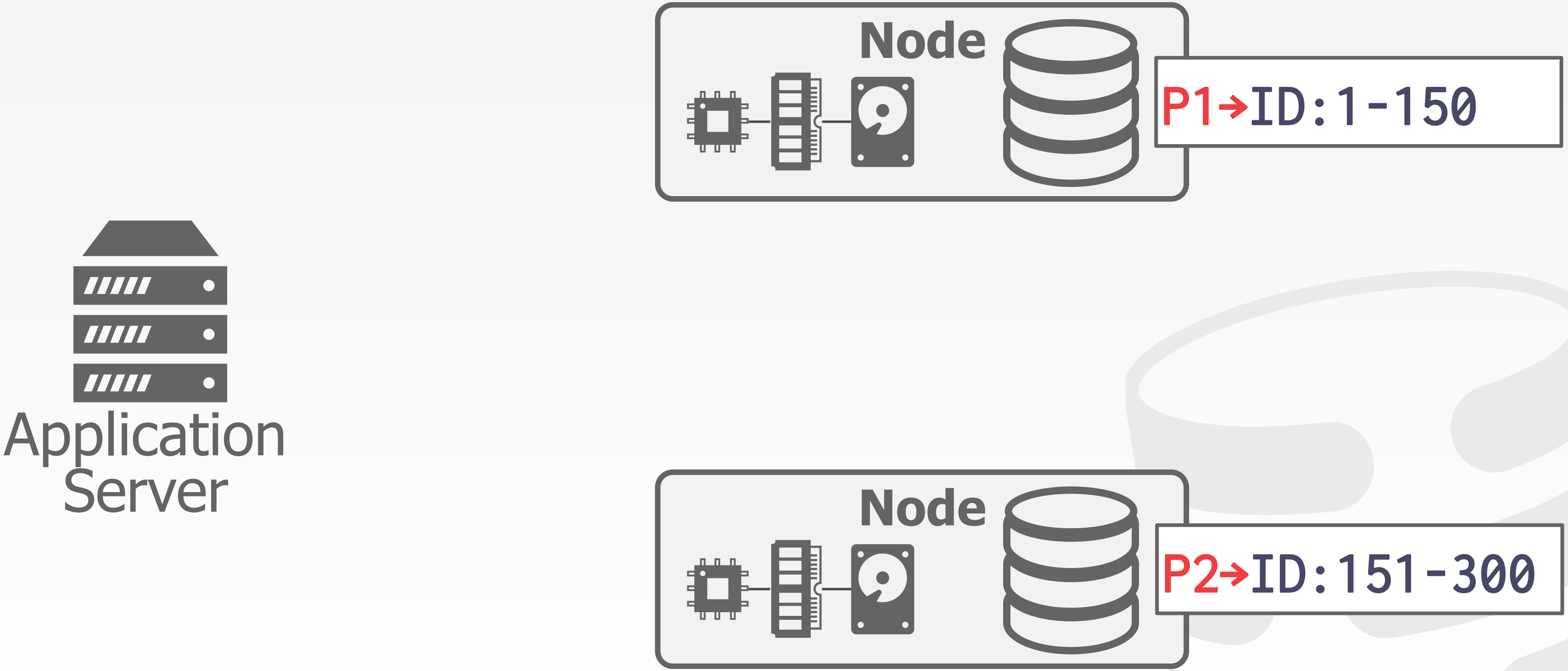
SHARED NOTHING EXAMPLE



SHARED NOTHING EXAMPLE



SHARED NOTHING EXAMPLE



Node

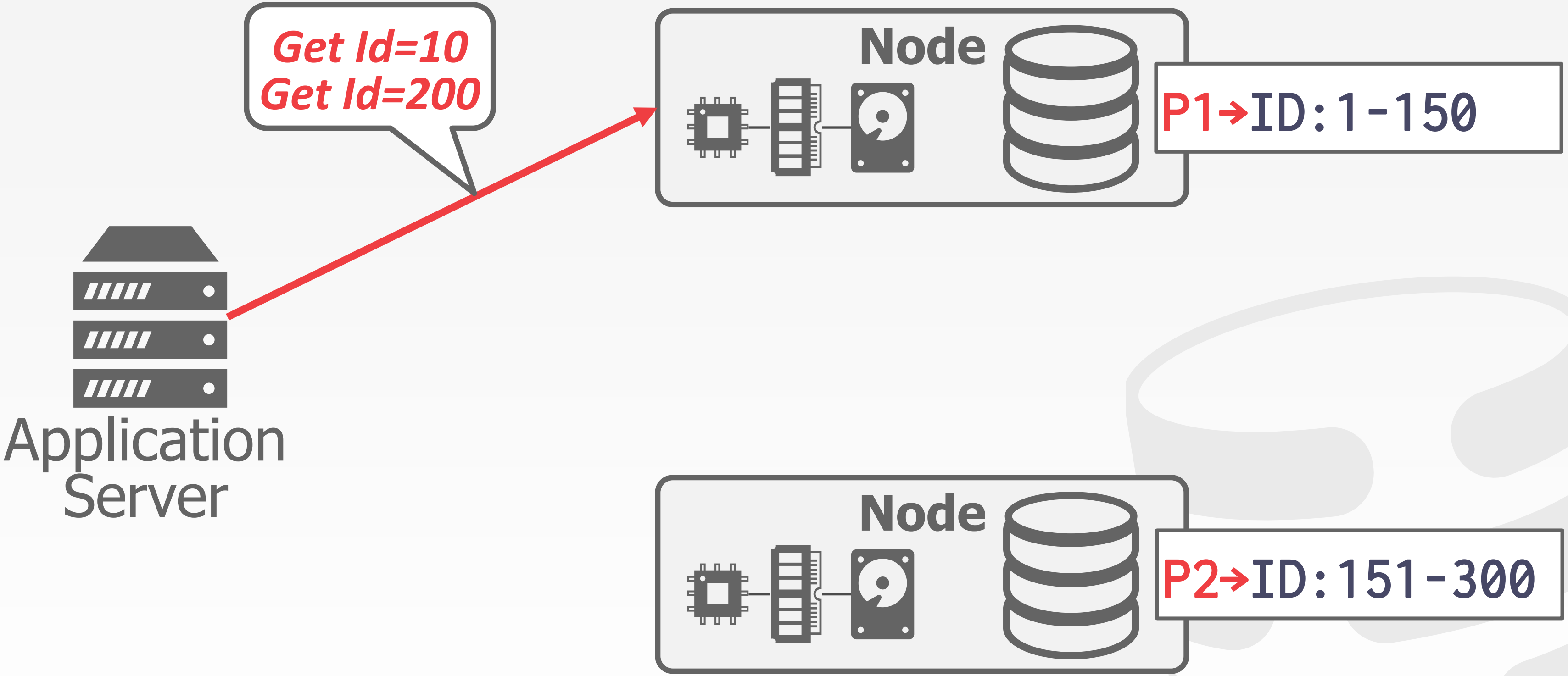
P1→ID:1-150

Application Server

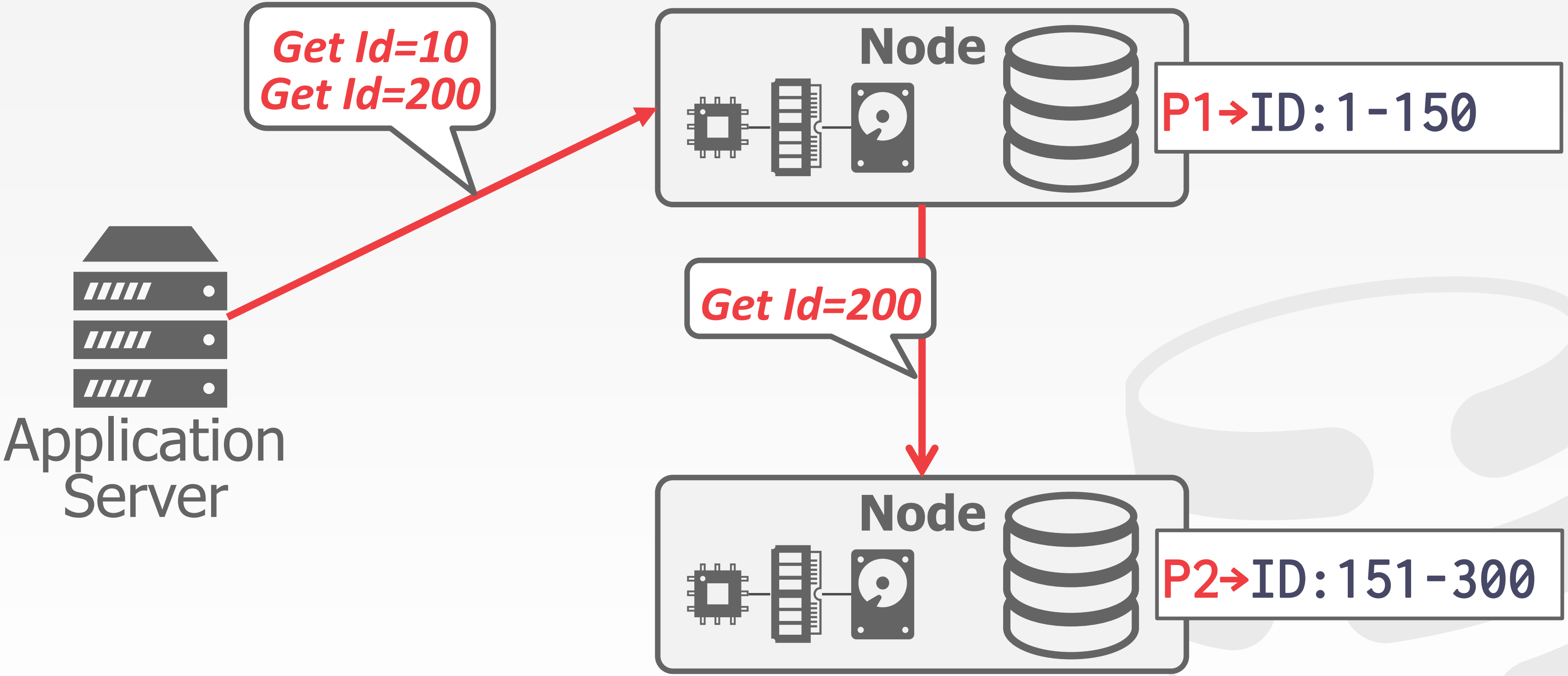
Node

P2→ID:151-300

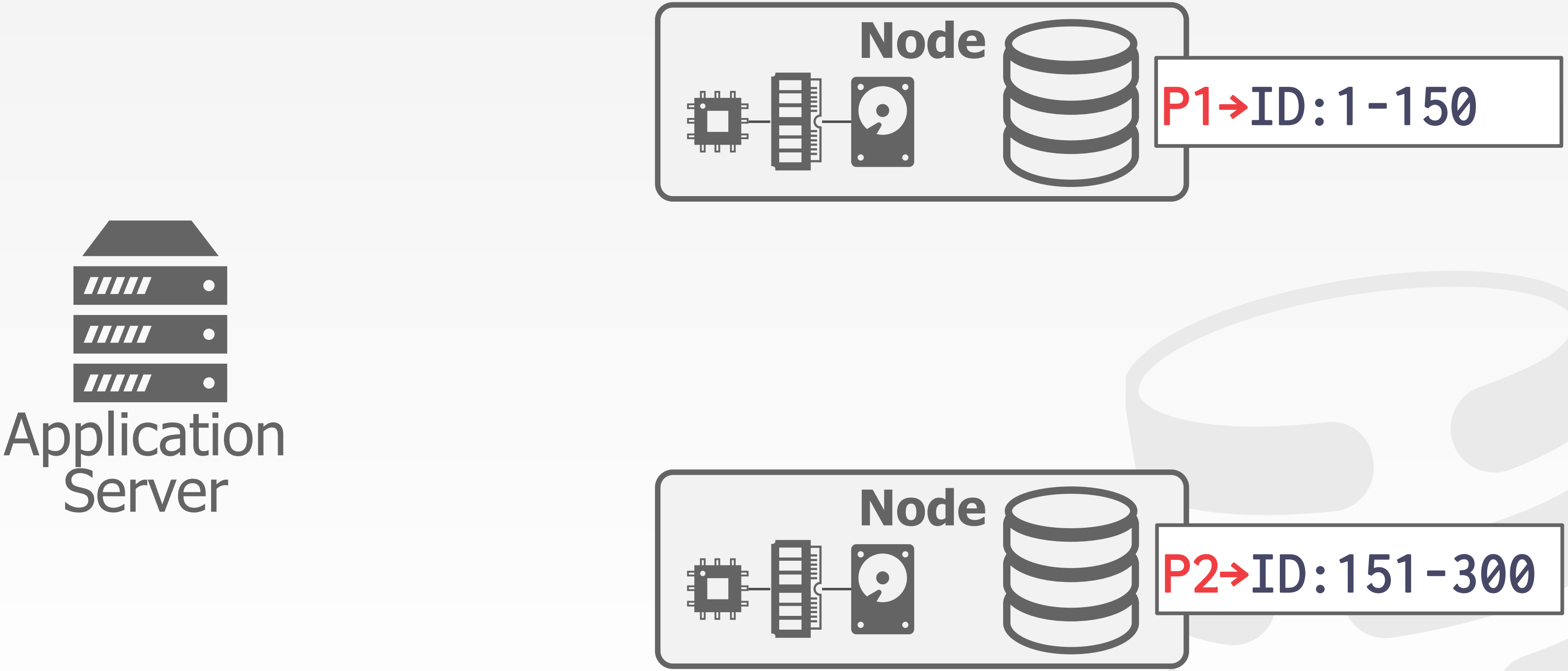
SHARED NOTHING EXAMPLE



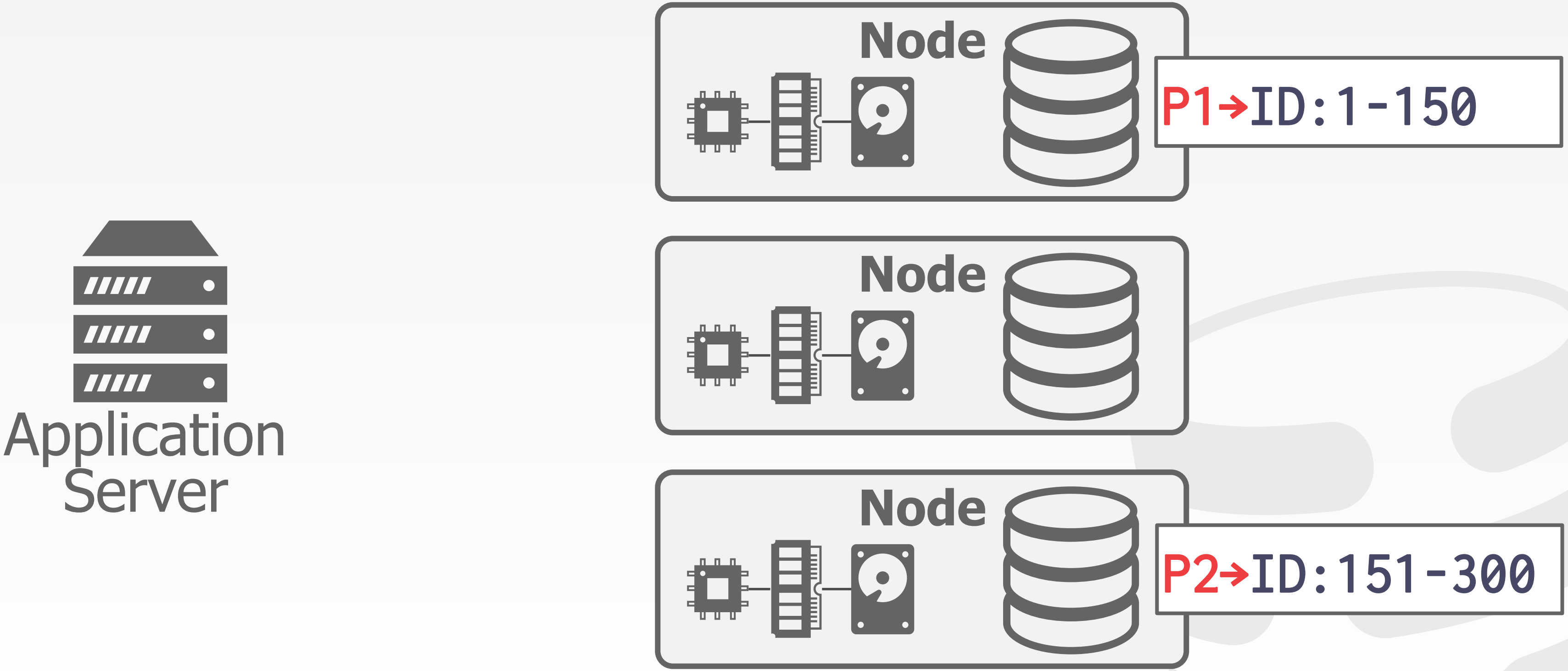
SHARED NOTHING EXAMPLE



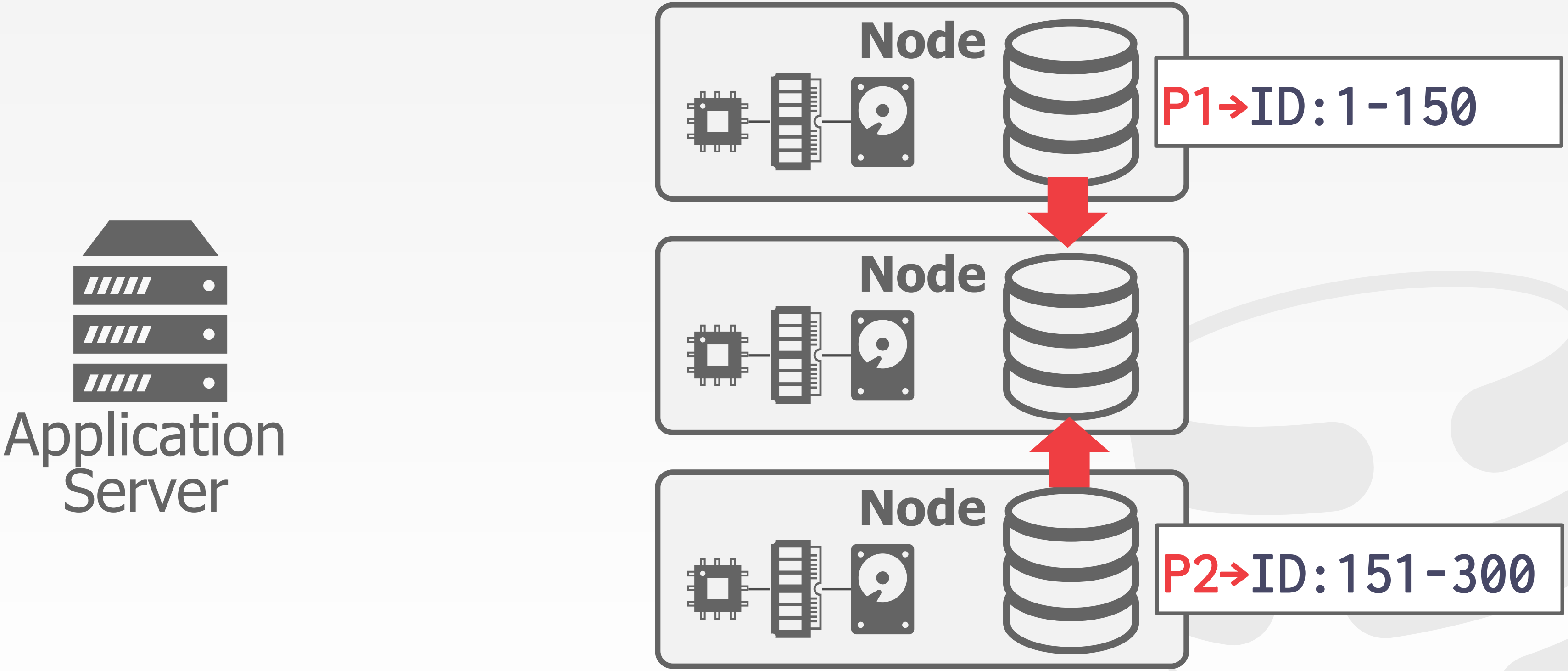
SHARED NOTHING EXAMPLE



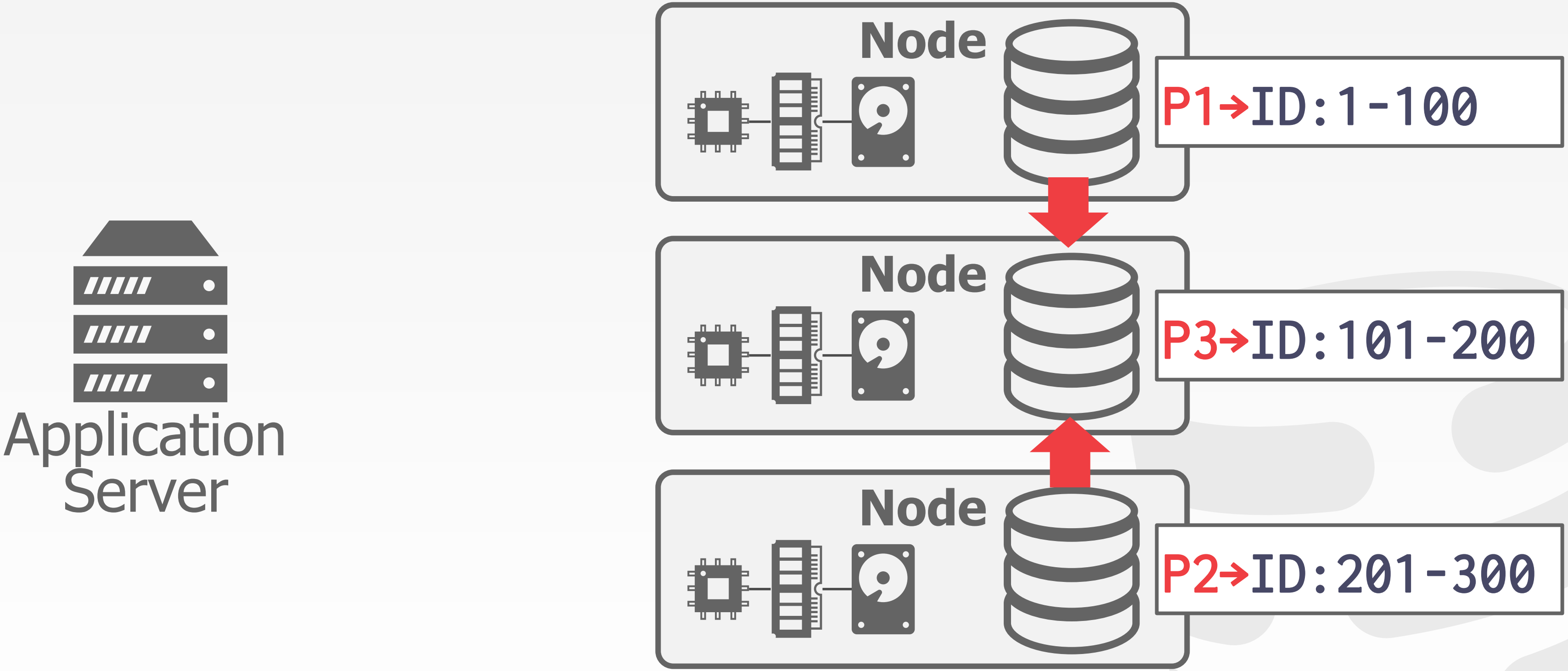
SHARED NOTHING EXAMPLE



SHARED NOTHING EXAMPLE



SHARED NOTHING EXAMPLE



EARLY DISTRIBUTED DATABASE SYSTEMS

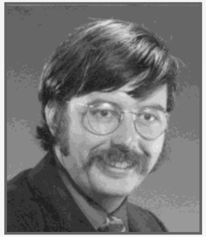
MUFFIN – UC Berkeley (1979)

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).

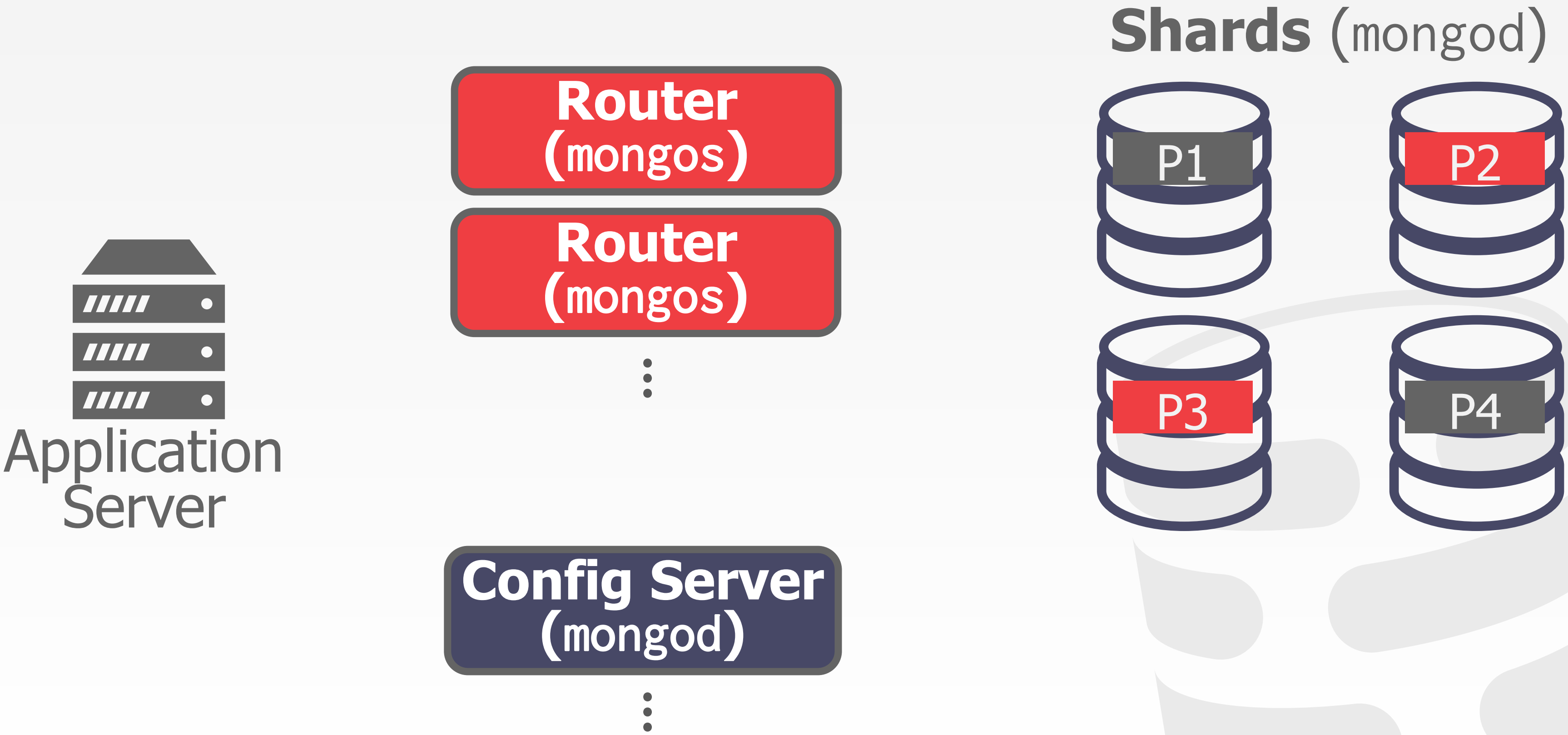
→ Makes provisioning and failover "easier".

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 HETEROGENOUS ARCHITECTURE



Shards (mongod)

Router (mongos)

Router (mongos)

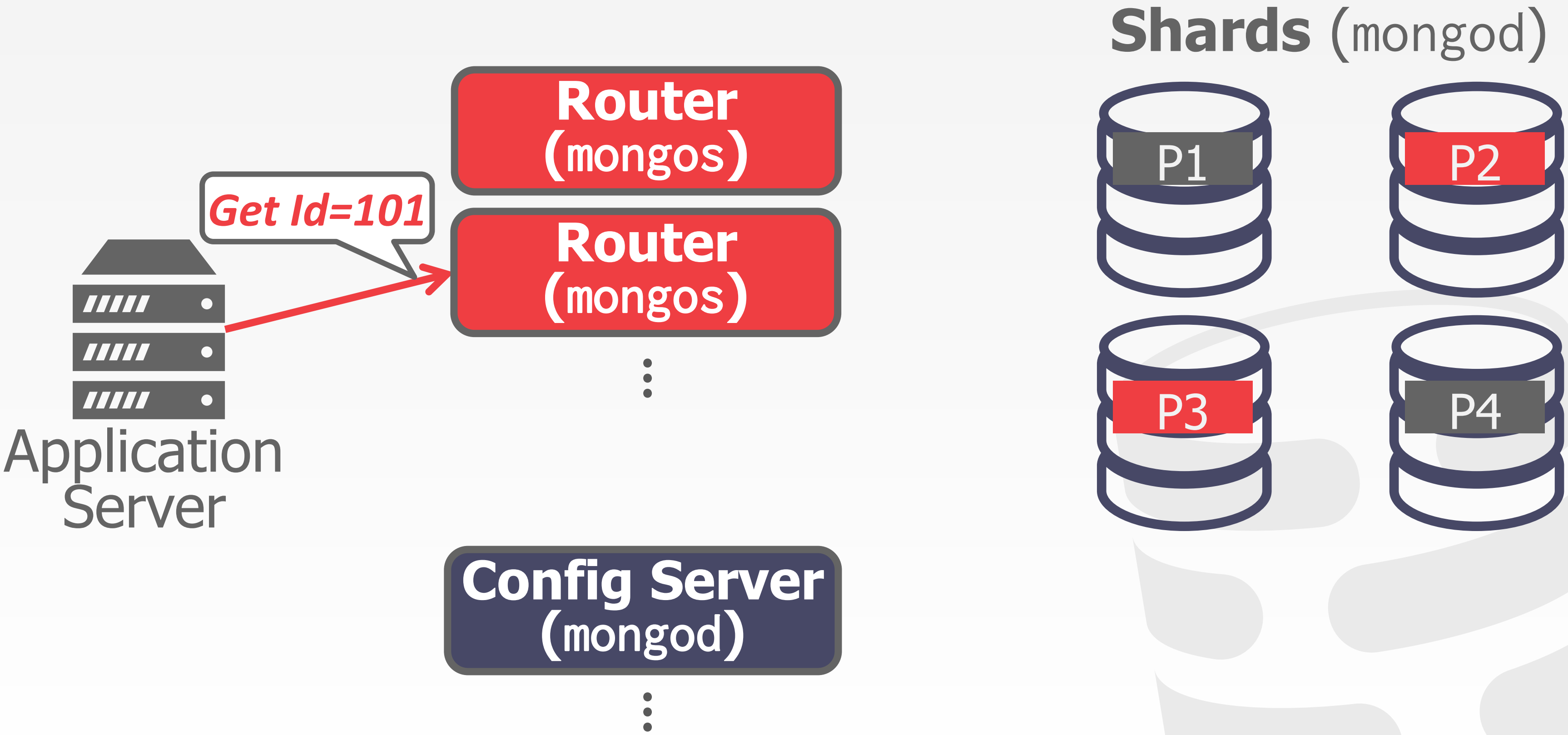
⋮

Application Server

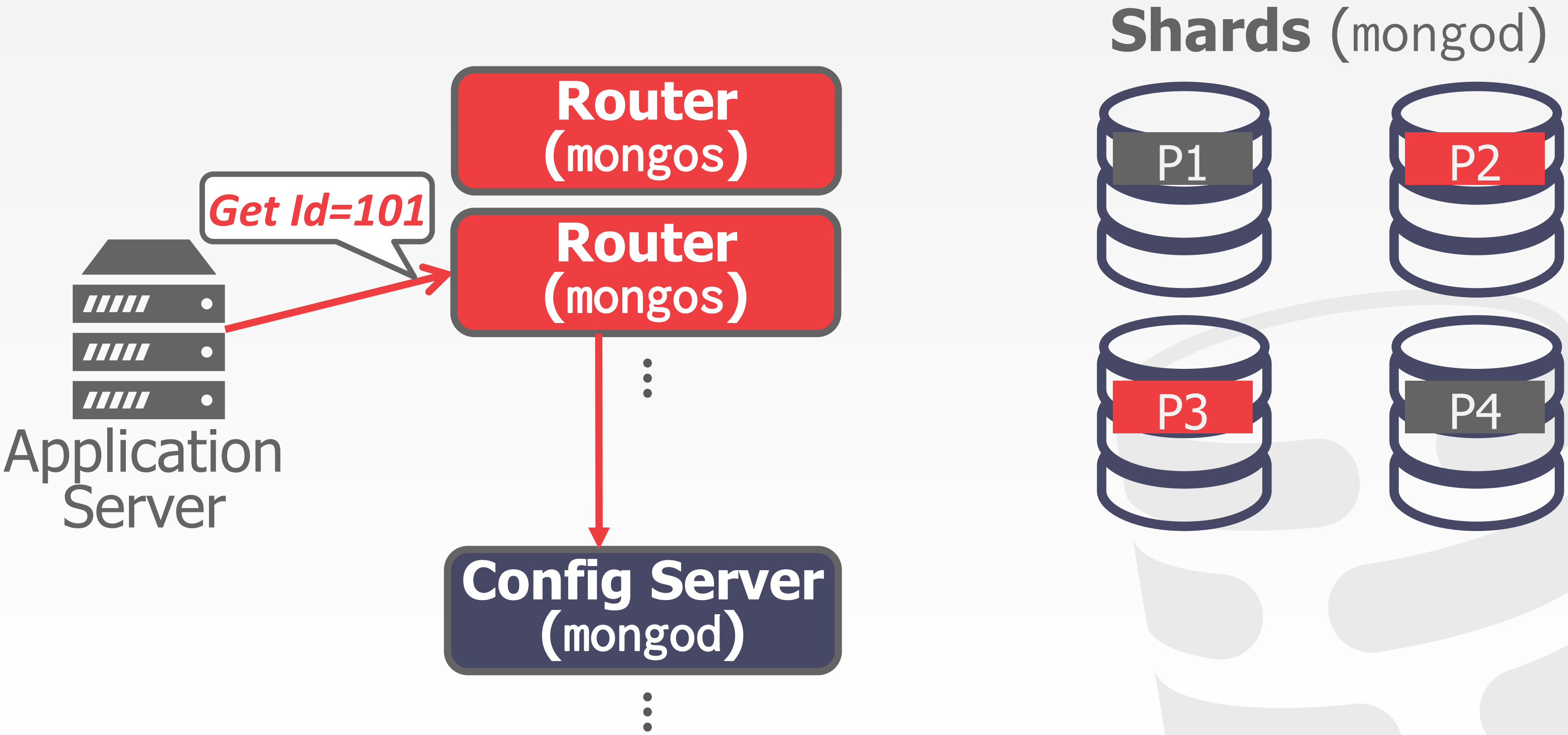
Config Server (mongod)

⋮

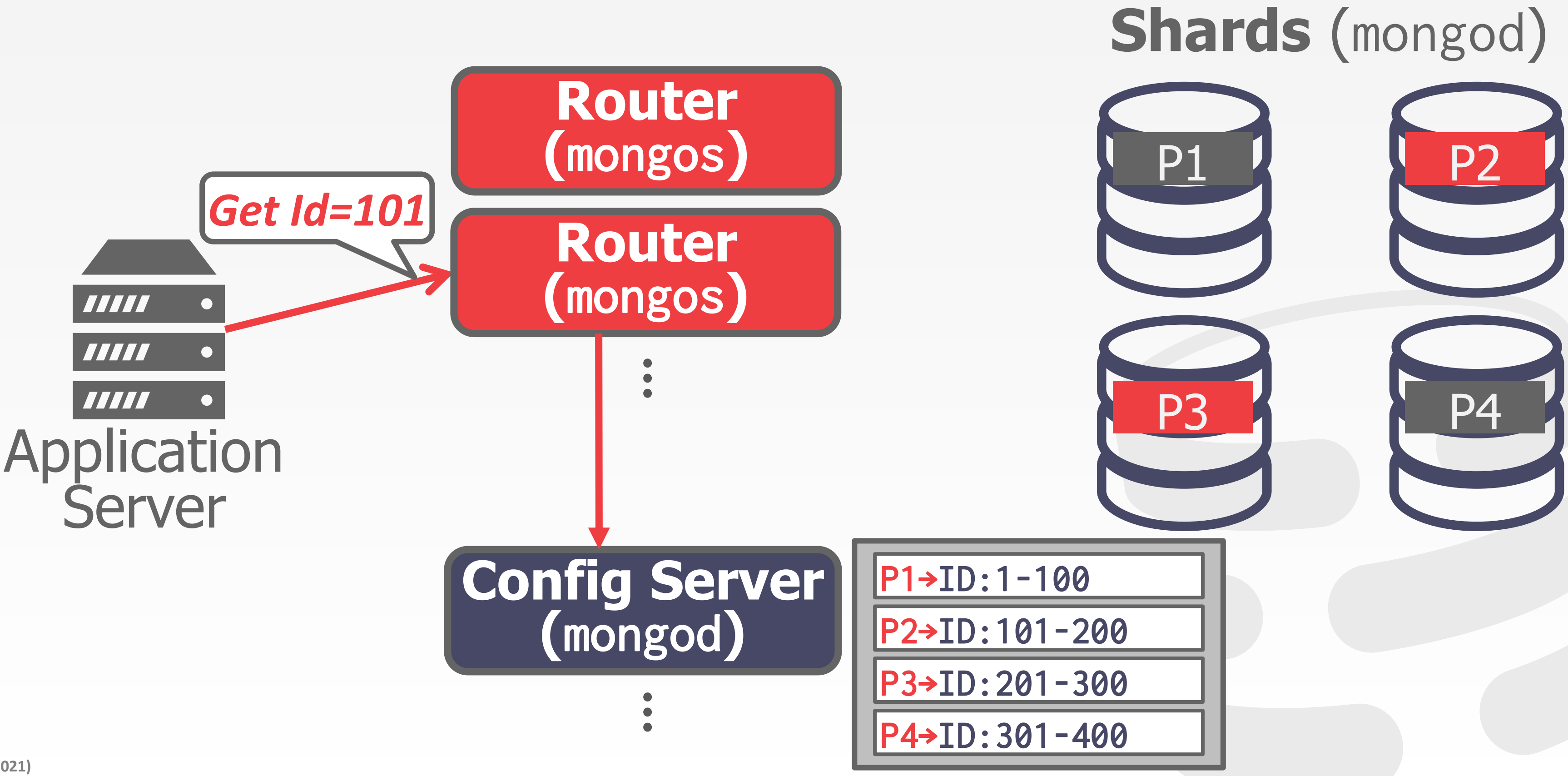
MONGODB HETEROGENOUS ARCHITECTURE



MONGODB HETEROGENOUS ARCHITECTURE



MONGODB HETEROGENOUS ARCHITECTURE



Shards (mongod)

P1

P2

P3

P4

Router (mongos)

Router (mongos)

Get Id=101

Application Server

Config Server (mongod)

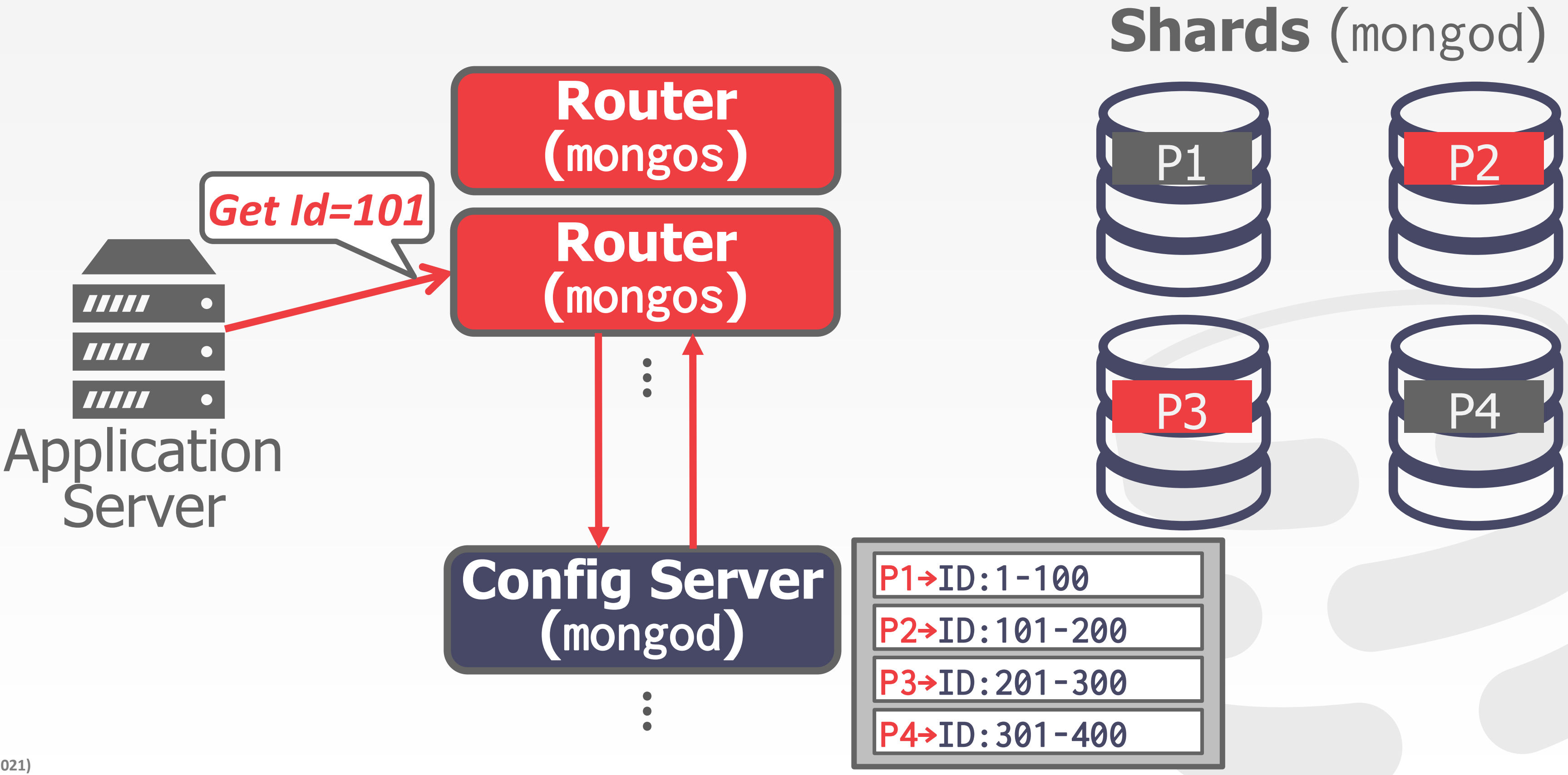
P1→ID:1-100

P2→ID:101-200

P3→ID:201-300

P4→ID:301-400

MONGODB HETEROGENOUS ARCHITECTURE



Shards (mongod)

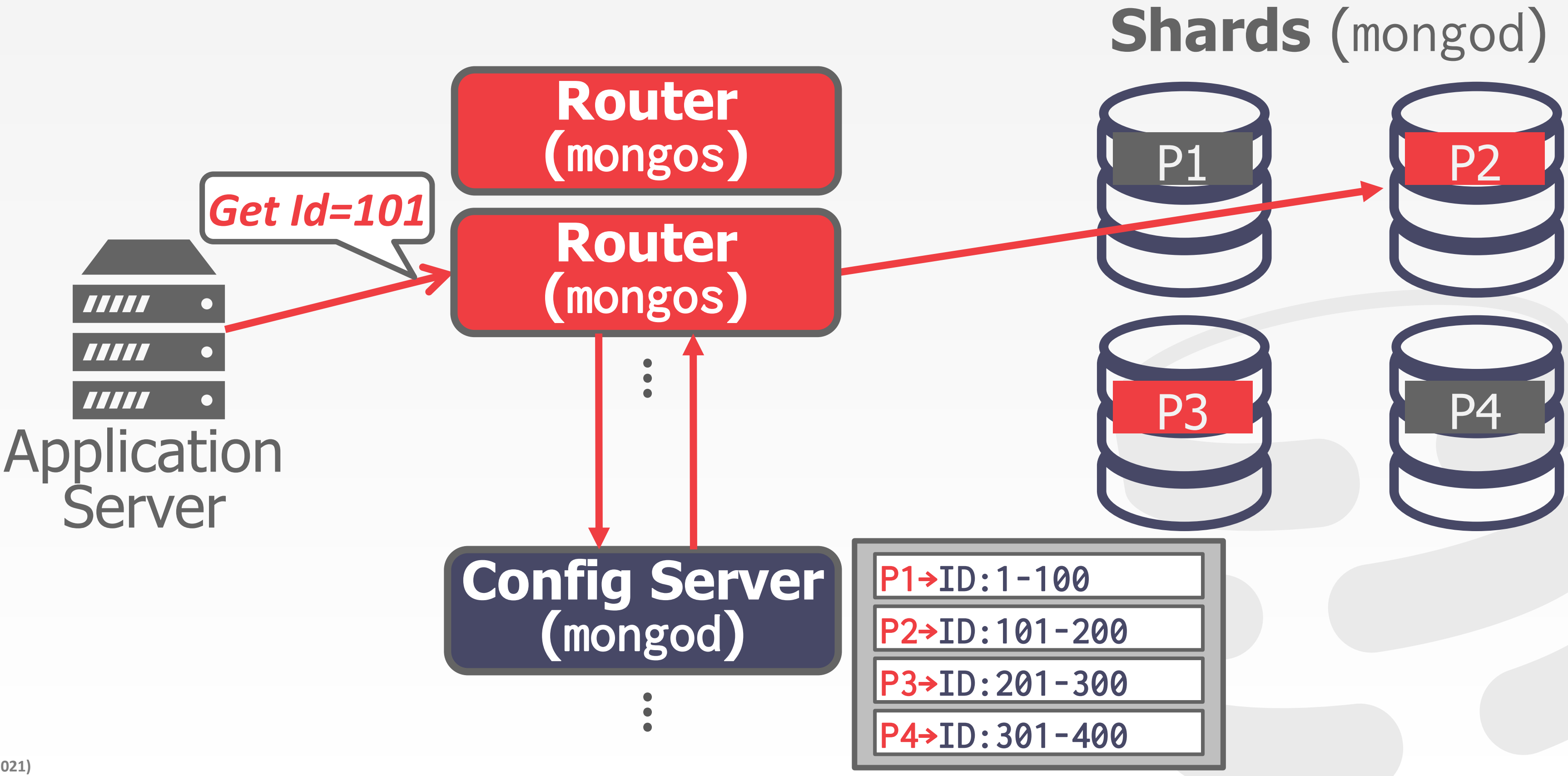
P1→ID:1-100

P2→ID:101-200

P3→ID:201-300

P4→ID:301-400

MONGODB HETEROGENOUS ARCHITECTURE



P1→ID:1-100
P2→ID:101-200
P3→ID:201-300
P4→ID:301-400

DATA TRANSPARENCY

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

A 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.

→ Often called "sharding" in NoSQL systems.

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

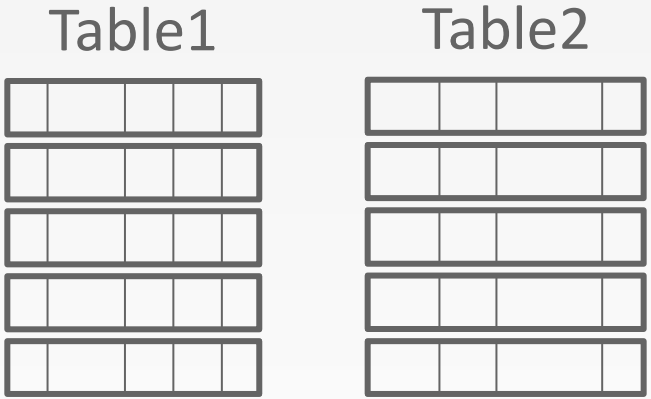
NAÏVE TABLE PARTITIONING

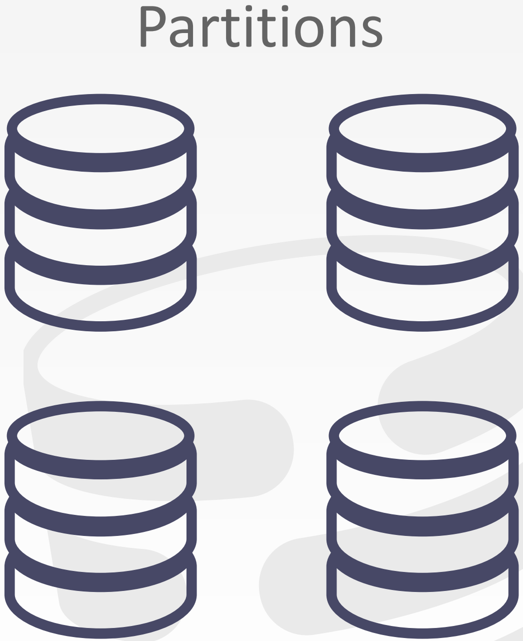
Assign an entire table to a single node.

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.

NAÏVE TABLE PARTITIONING

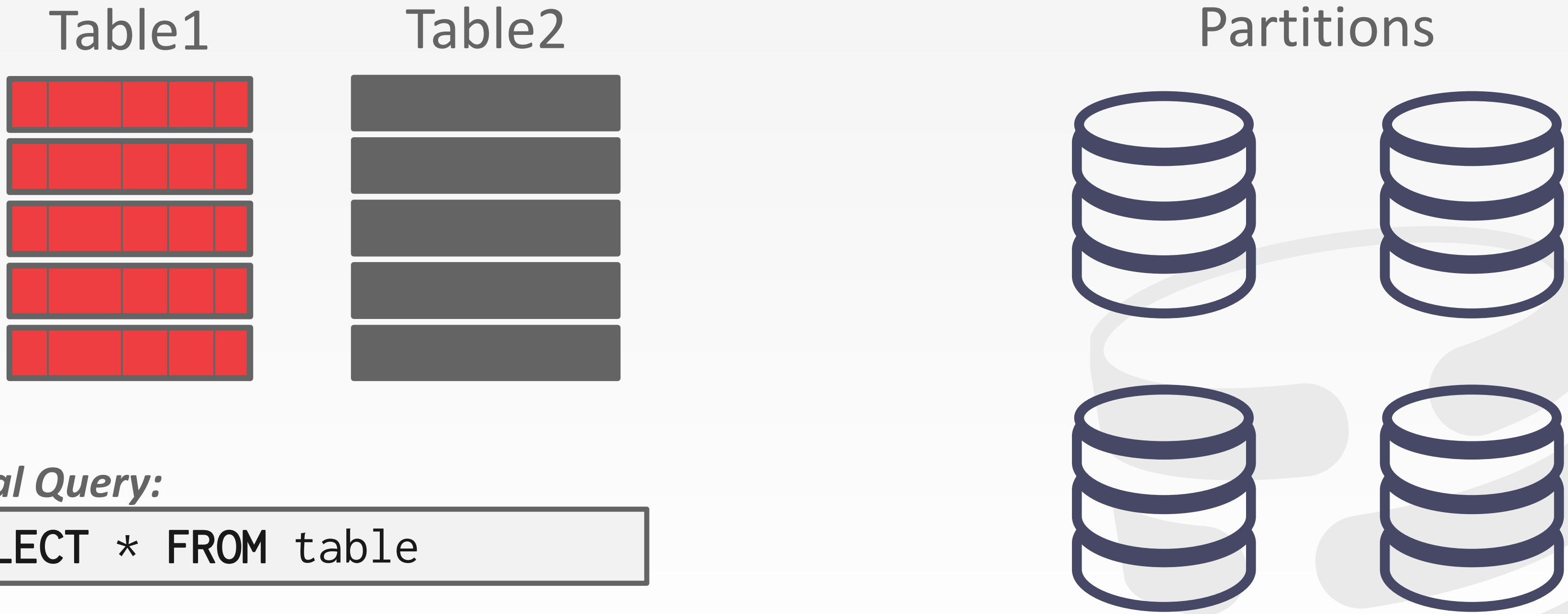




Ideal Query:

```
SELECT * FROM table
```

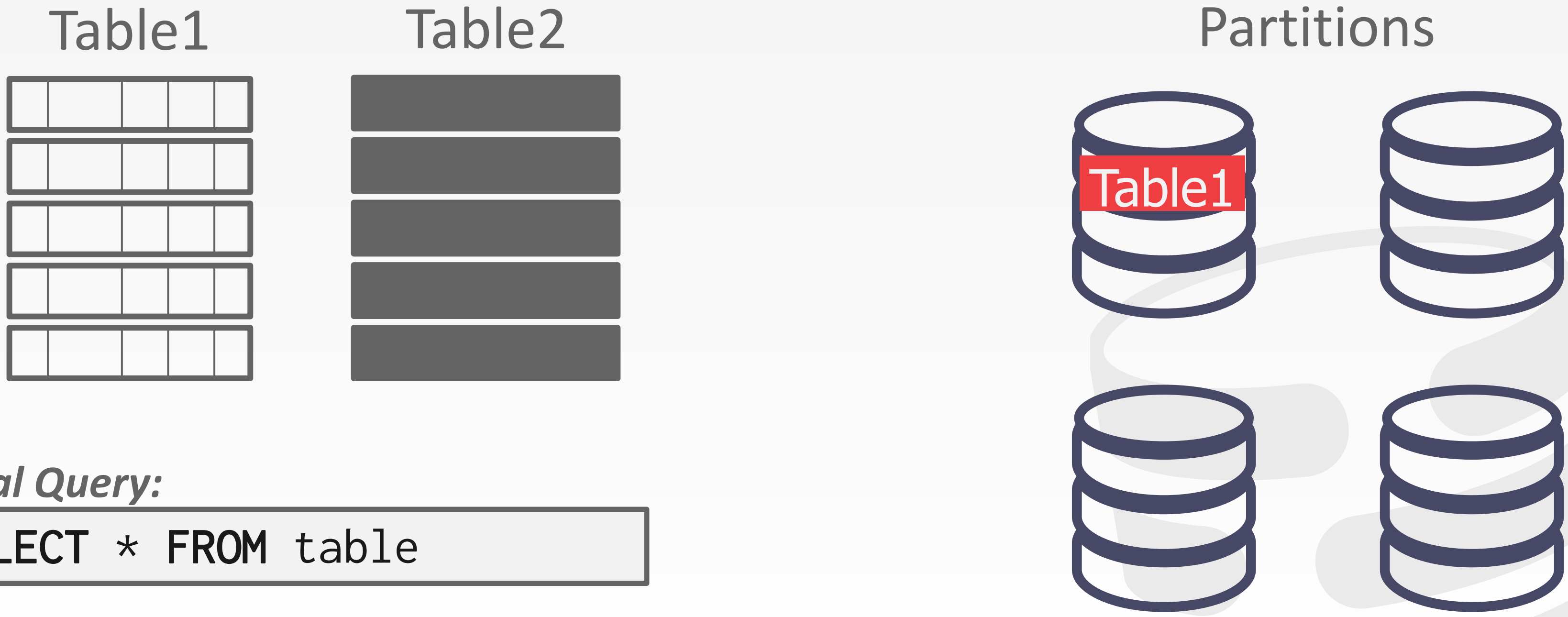
NAÏVE TABLE PARTITIONING



Ideal Query:

```
SELECT * FROM table
```

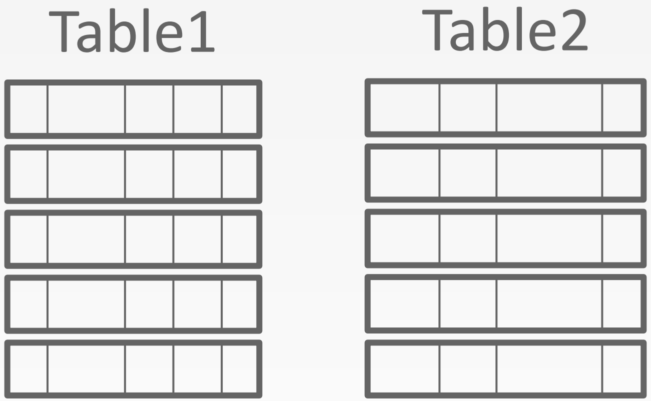
NAÏVE TABLE PARTITIONING

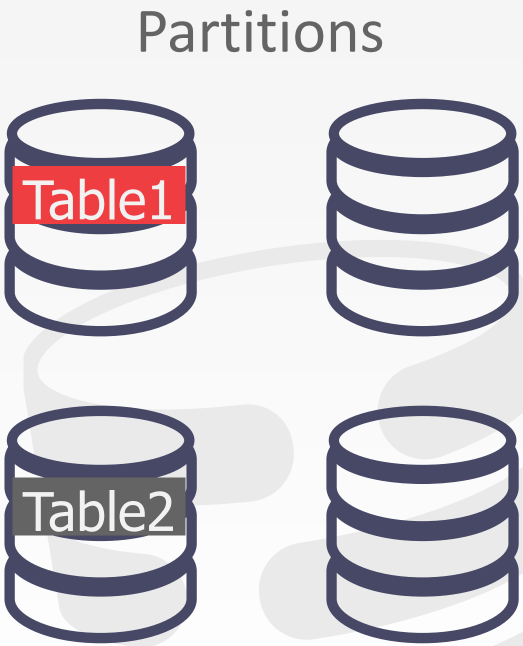


Ideal Query:

```
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.

→ Hash Partitioning, Range Partitioning

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

HORIZONTAL PARTITIONING

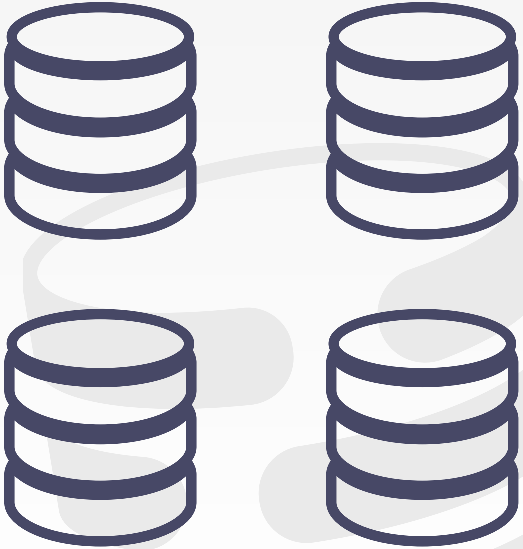
Table1

101	a	XXX	2019-11-29
102	b	XXY	2019-11-28
103	c	XYZ	2019-11-29
104	d	XYX	2019-11-27
105	e	XYY	2019-11-29

Ideal Query:

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

Partitions



HORIZONTAL PARTITIONING

Partitioning Key

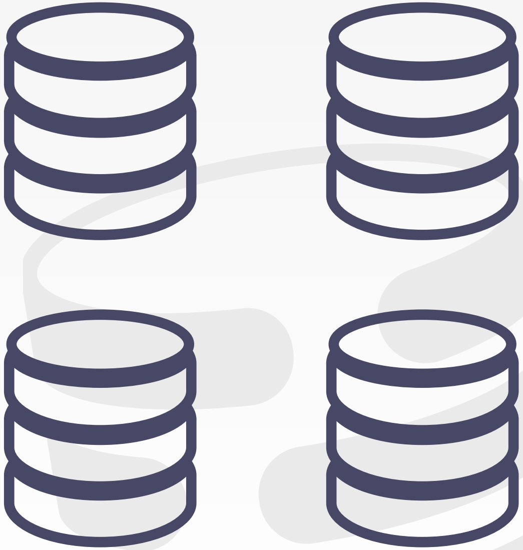
Table1

101	a	XXX	2019-11-29
102	b	XXY	2019-11-28
103	c	XYZ	2019-11-29
104	d	XYX	2019-11-27
105	e	XYY	2019-11-29

Ideal Query:

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

Partitions



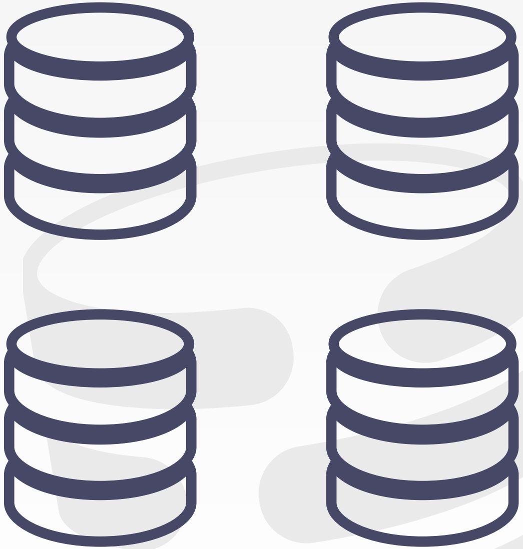
HORIZONTAL PARTITIONING

Partitioning Key

Table1

101	a	XXX	2019-11-29	***hash(a)%4 = P2***
102	b	XXY	2019-11-28	***hash(b)%4 = P4***
103	c	XYZ	2019-11-29	***hash(c)%4 = P3***
104	d	XYX	2019-11-27	***hash(d)%4 = P2***
105	e	XYY	2019-11-29	***hash(e)%4 = P1***

Partitions



Ideal Query:

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

HORIZONTAL PARTITIONING

Partitioning Key

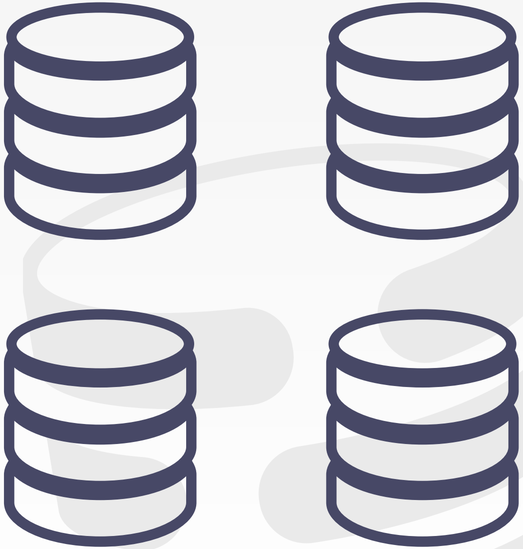
Table1

101	a	XXX	2019-11-29	*hash(a)%4 = P2*
102	b	XXY	2019-11-28	*hash(b)%4 = P4*
103	c	XYZ	2019-11-29	*hash(c)%4 = P3*
104	d	XYX	2019-11-27	*hash(d)%4 = P2*
105	e	XYY	2019-11-29	*hash(e)%4 = P1*

Ideal Query:

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

Partitions



HORIZONTAL PARTITIONING

Partitioning Key

Table1

101	a	XXX	2019-11-29
102	b	XXY	2019-11-28
103	c	XYZ	2019-11-29
104	d	XYX	2019-11-27
105	e	XYY	2019-11-29

hash(a)%4 = P2

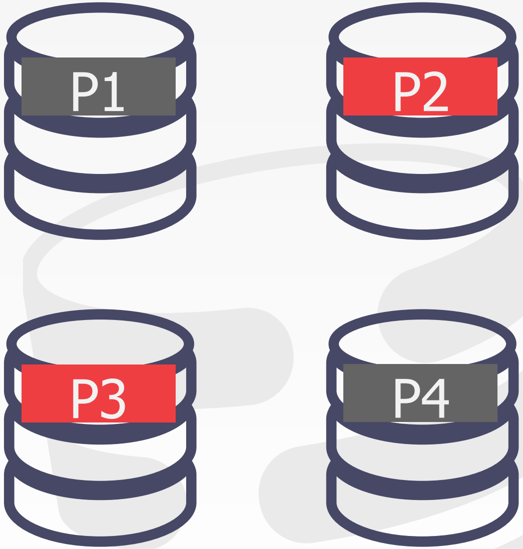
hash(b)%4 = P4

hash(c)%4 = P3

hash(d)%4 = P2

hash(e)%4 = P1

Partitions



Ideal Query:

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

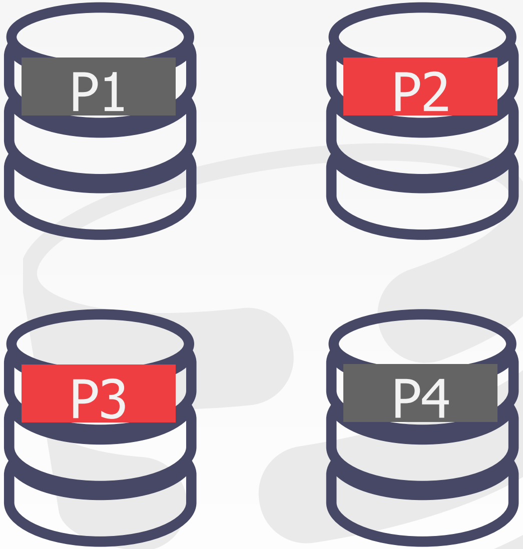
HORIZONTAL PARTITIONING

Partitioning Key

Table1

101	a	XXX	2019-11-29	***hash(a)%4 = P2***
102	b	XXY	2019-11-28	***hash(b)%4 = P4***
103	c	XYZ	2019-11-29	***hash(c)%4 = P3***
104	d	XYX	2019-11-27	***hash(d)%4 = P2***
105	e	XYY	2019-11-29	***hash(e)%4 = P1***

Partitions



Ideal Query:

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

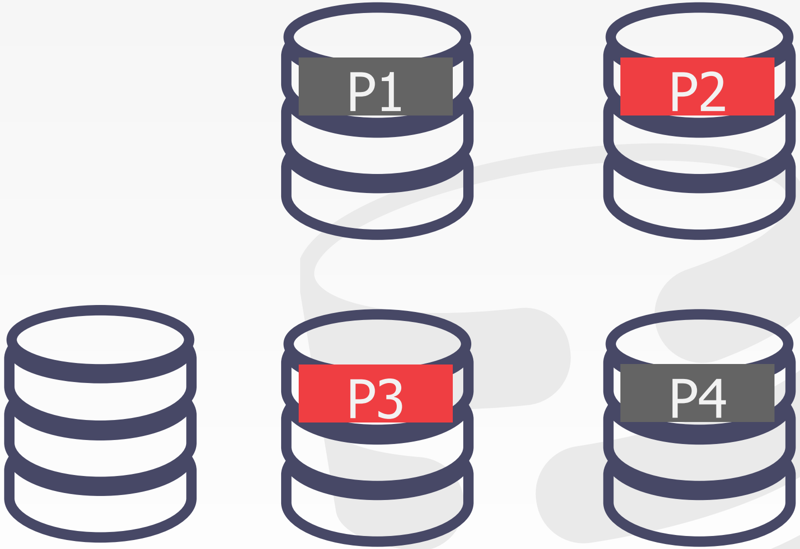
HORIZONTAL PARTITIONING

Partitioning Key

Table1

101	a	XXX	2019-11-29	*hash(a)%4 = P2*
102	b	XXY	2019-11-28	*hash(b)%4 = P4*
103	c	XYZ	2019-11-29	*hash(c)%4 = P3*
104	d	XYX	2019-11-27	*hash(d)%4 = P2*
105	e	XYY	2019-11-29	*hash(e)%4 = P1*

Partitions



Ideal Query:

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

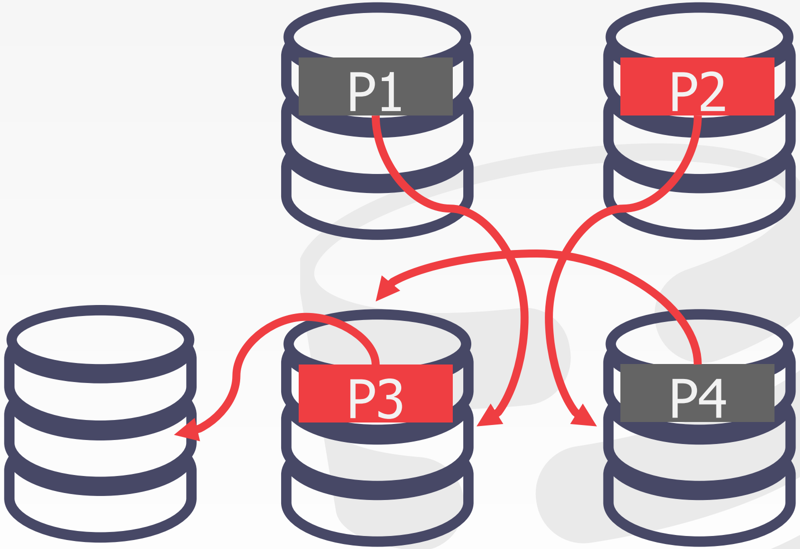
HORIZONTAL PARTITIONING

Partitioning Key

Table1

101	a	XXX	2019-11-29	***hash(a)%5 = P4***
102	b	XXY	2019-11-28	***hash(b)%5 = P3***
103	c	XYZ	2019-11-29	***hash(c)%5 = P5***
104	d	XYX	2019-11-27	***hash(d)%5 = P1***
105	e	XYY	2019-11-29	***hash(e)%5 = P3***

Partitions

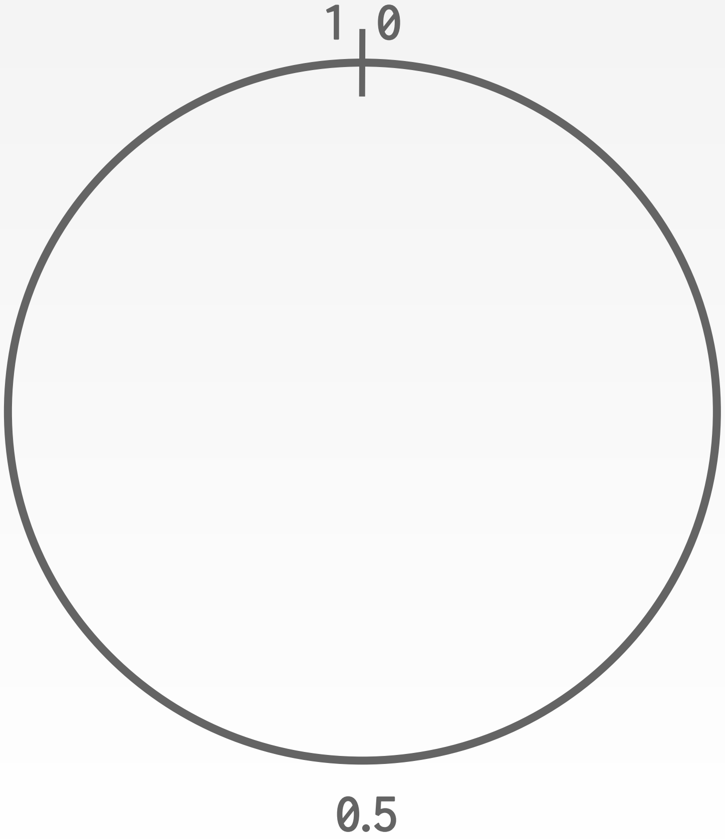


Ideal Query:

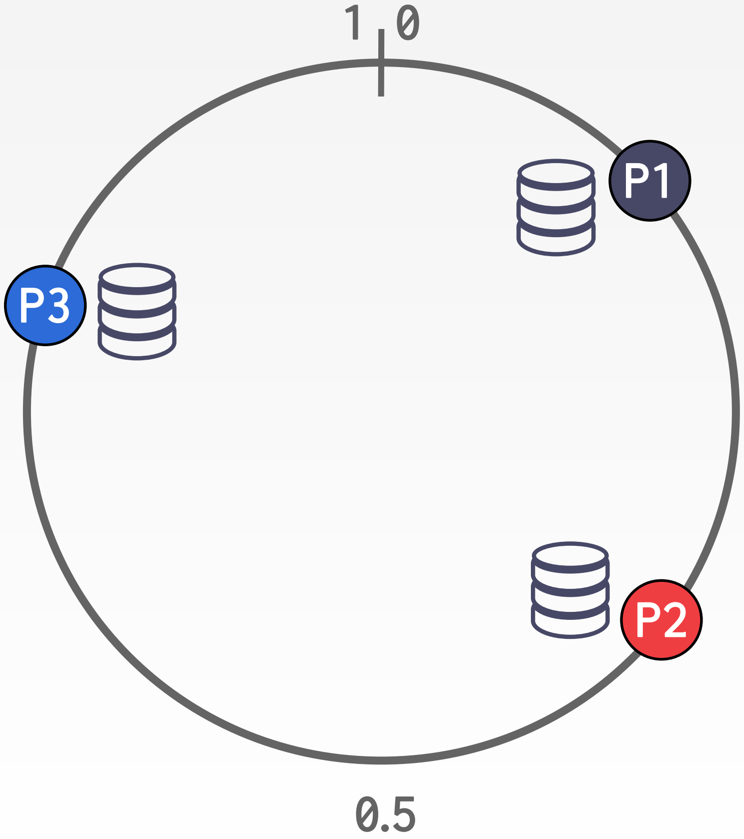
```
SELECT * FROM table
 WHERE partitionKey = ?
```

CONSISTENT HASHING

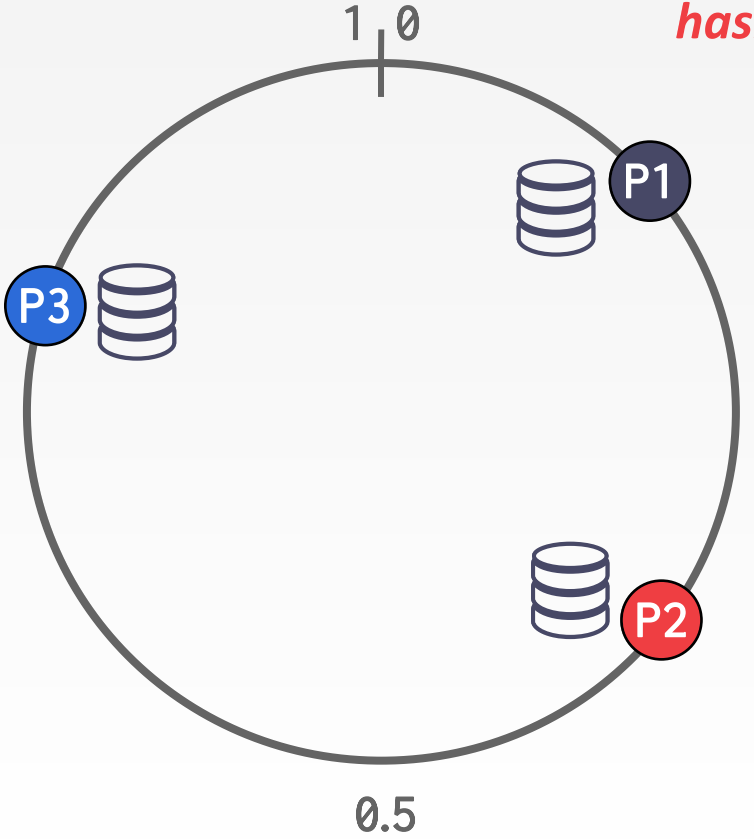
CONSISTENT HASHING



CONSISTENT HASHING

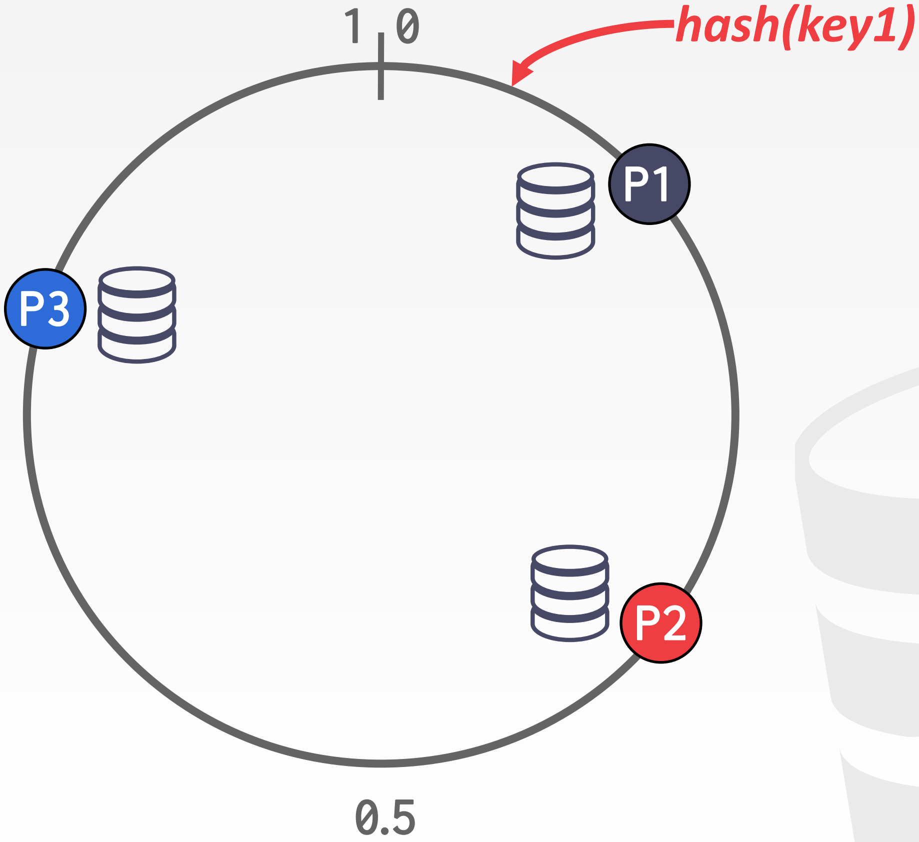


CONSISTENT HASHING

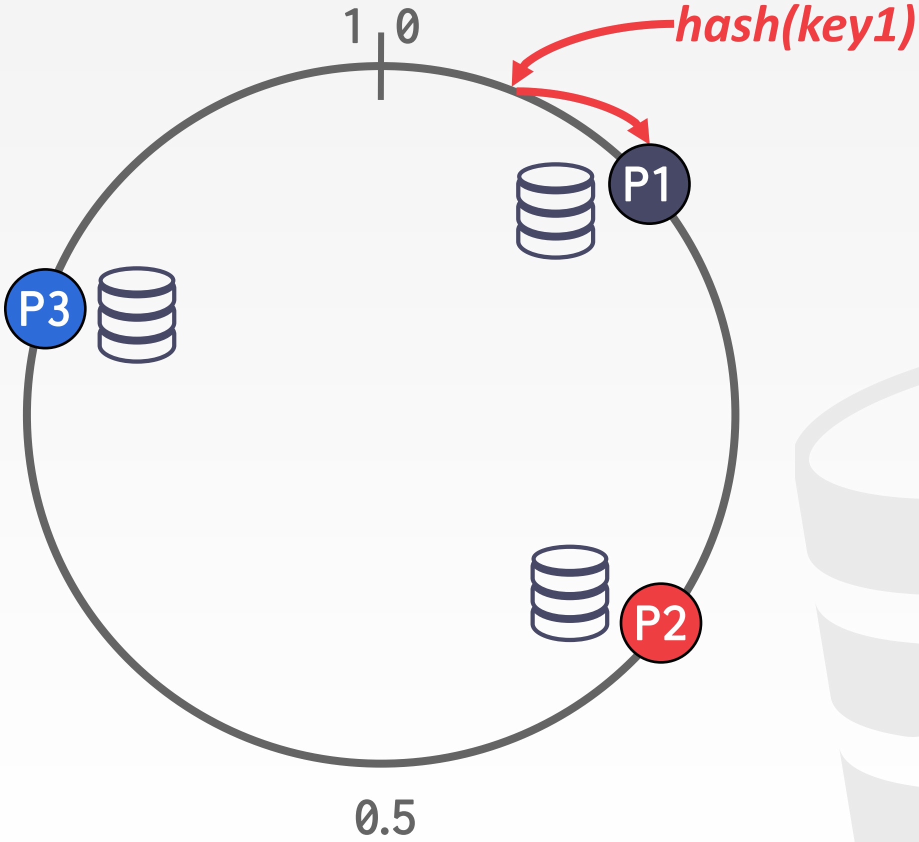


hash(key1)

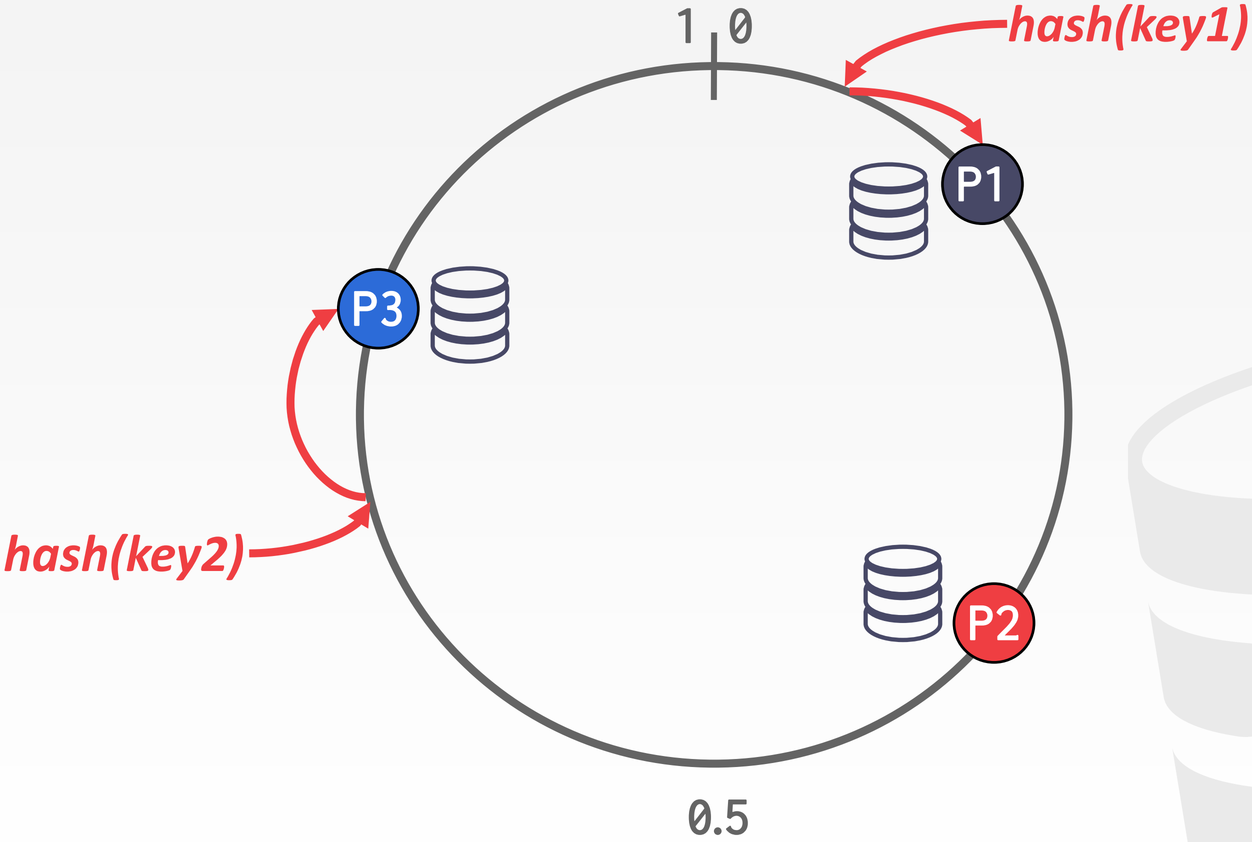
CONSISTENT HASHING



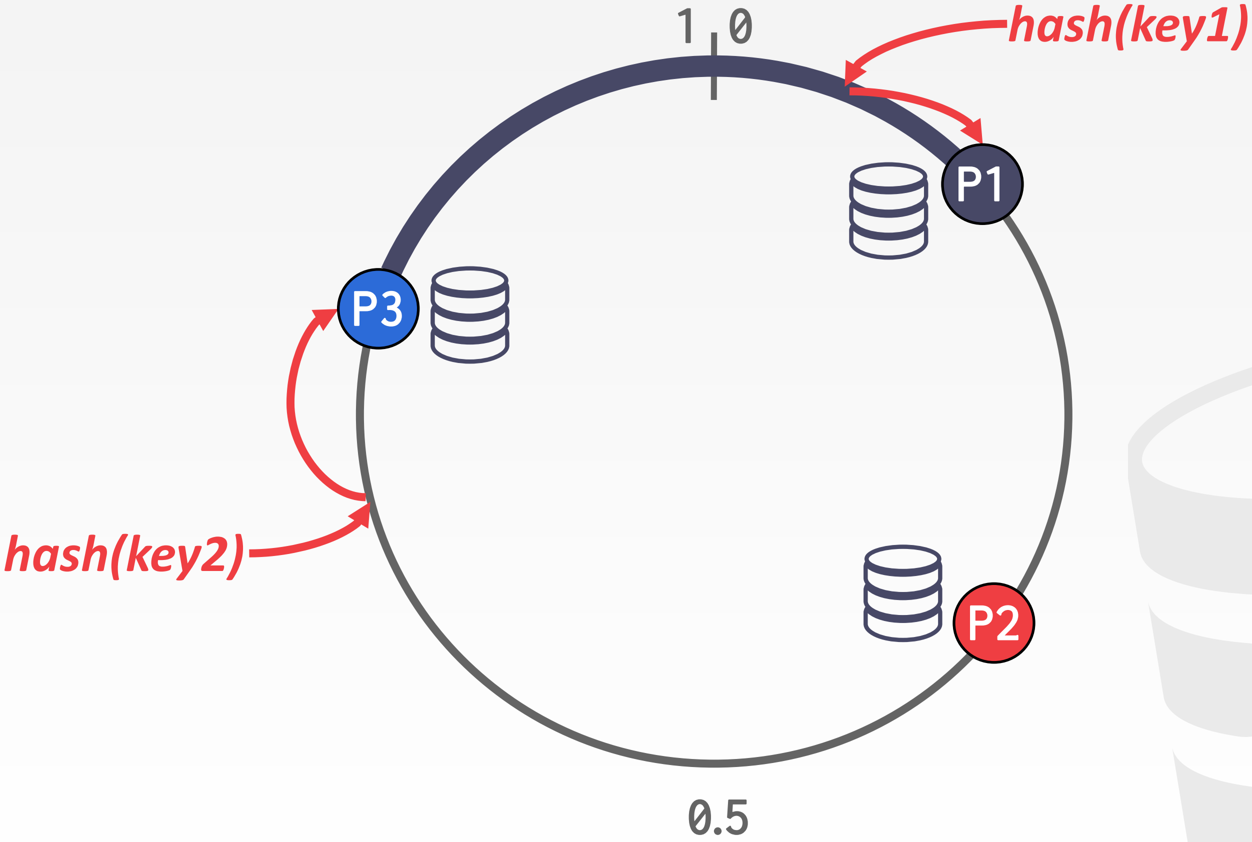
CONSISTENT HASHING



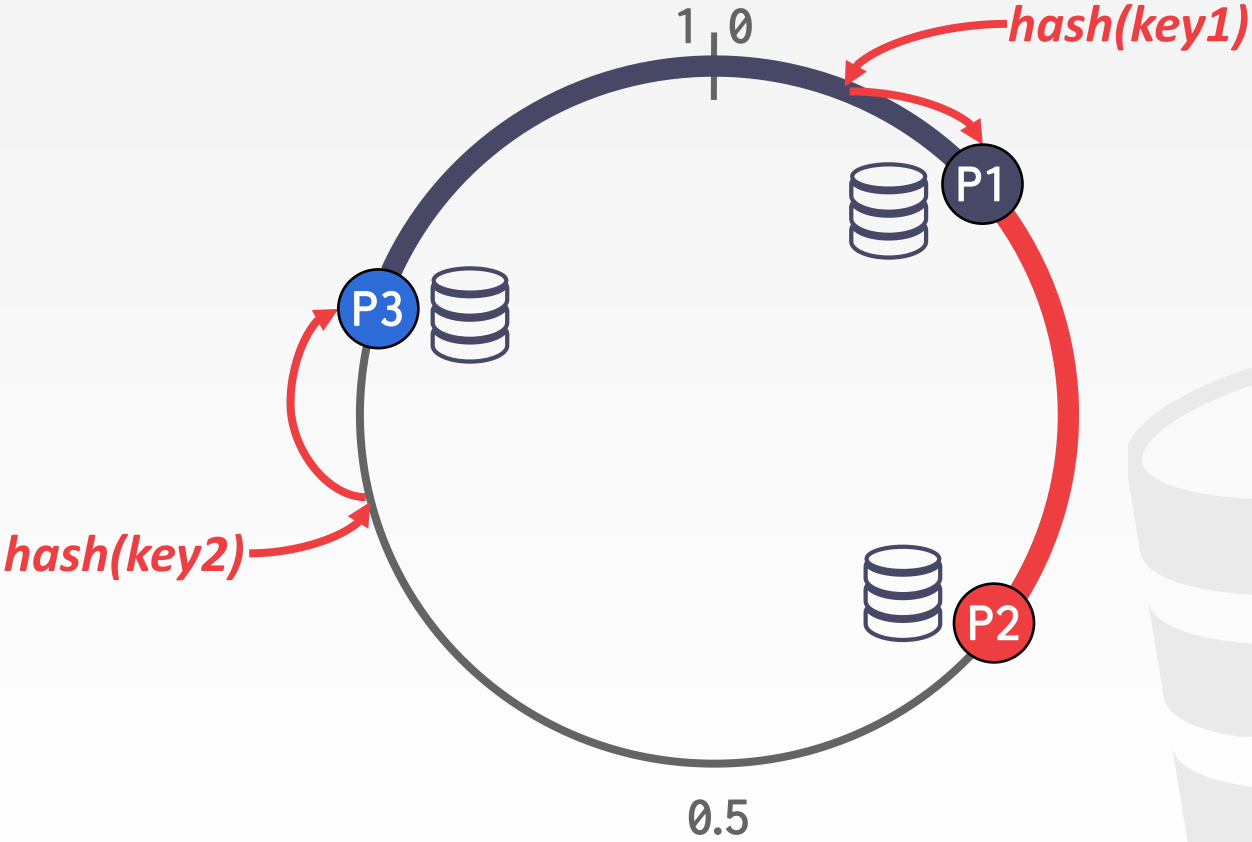
CONSISTENT HASHING



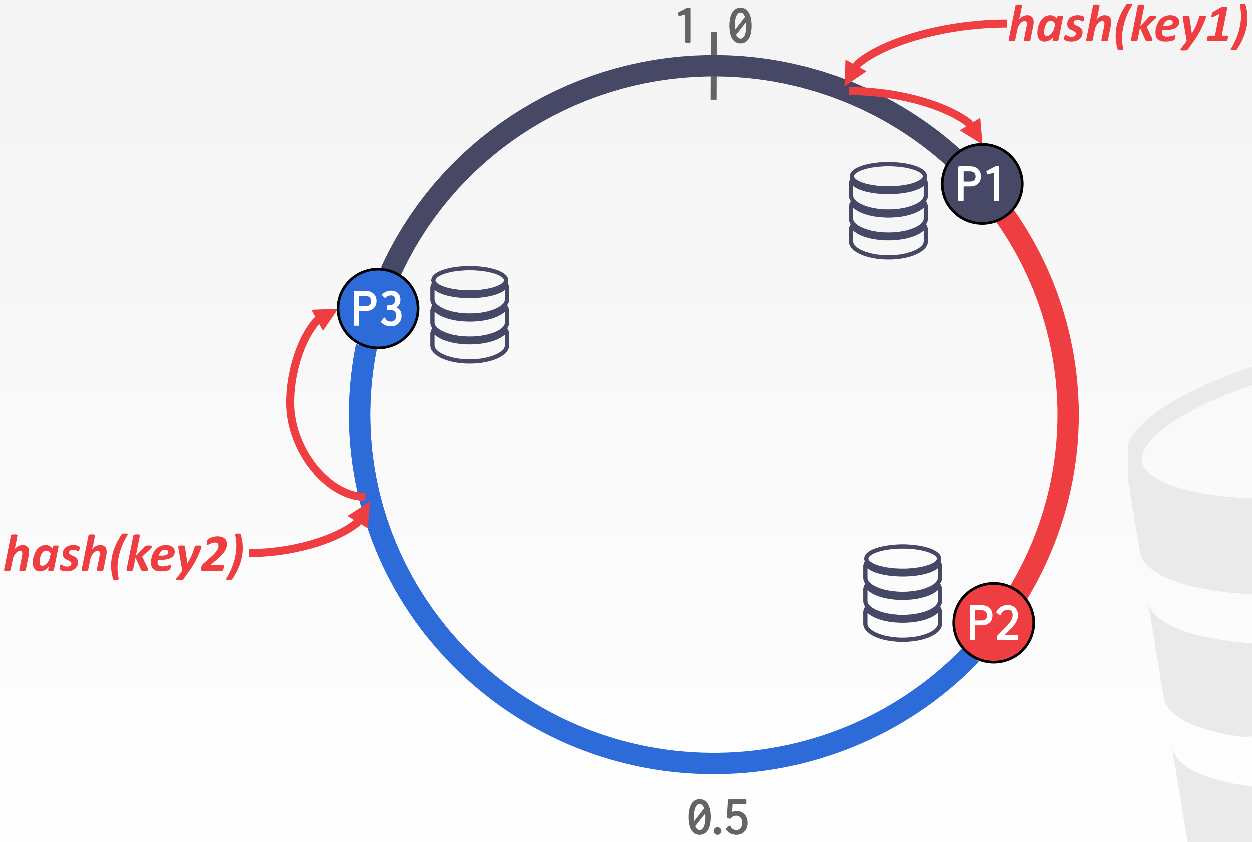
CONSISTENT HASHING



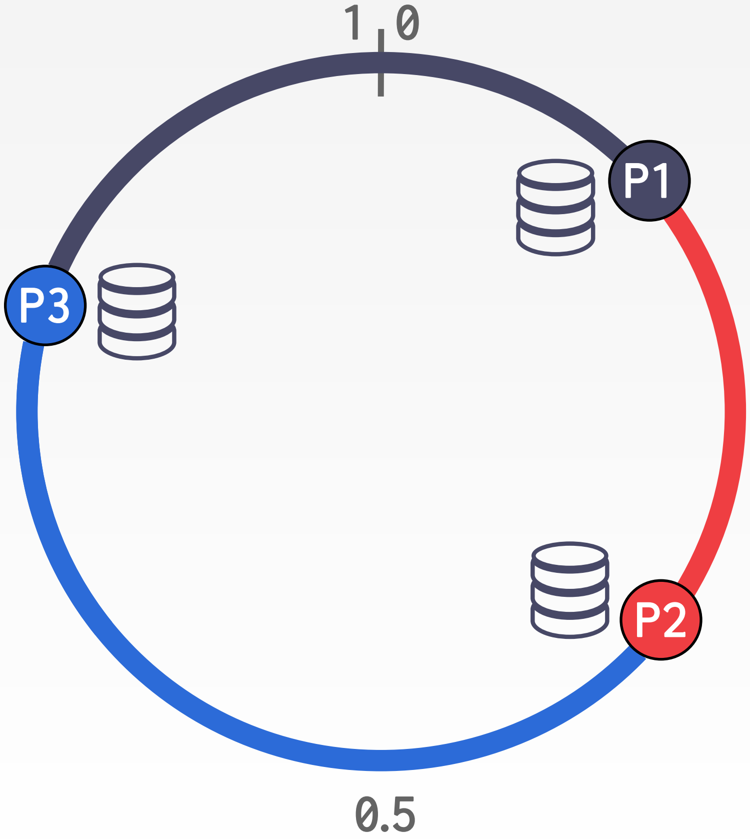
CONSISTENT HASHING



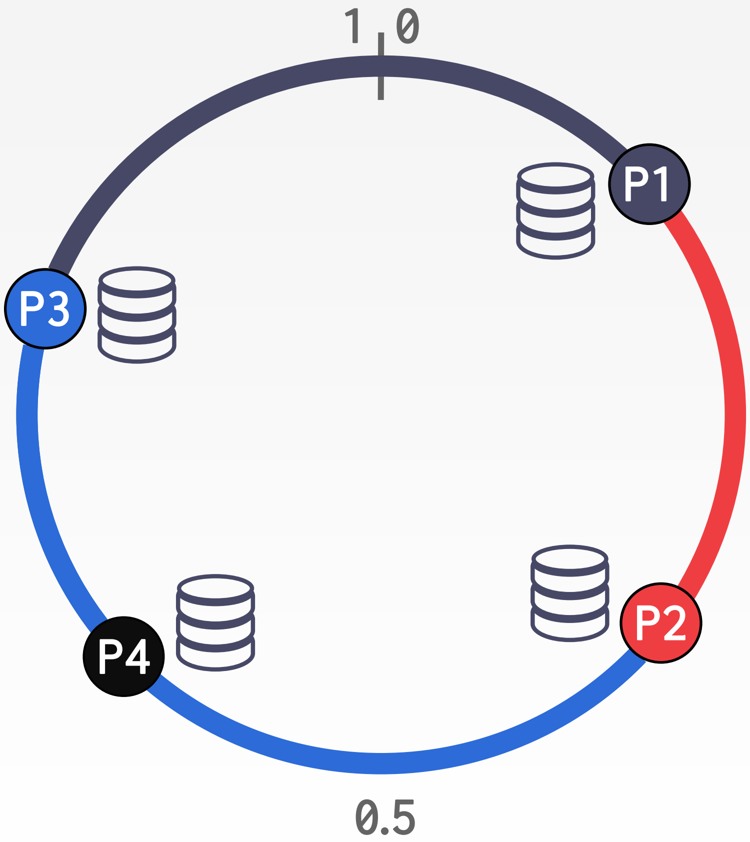
CONSISTENT HASHING



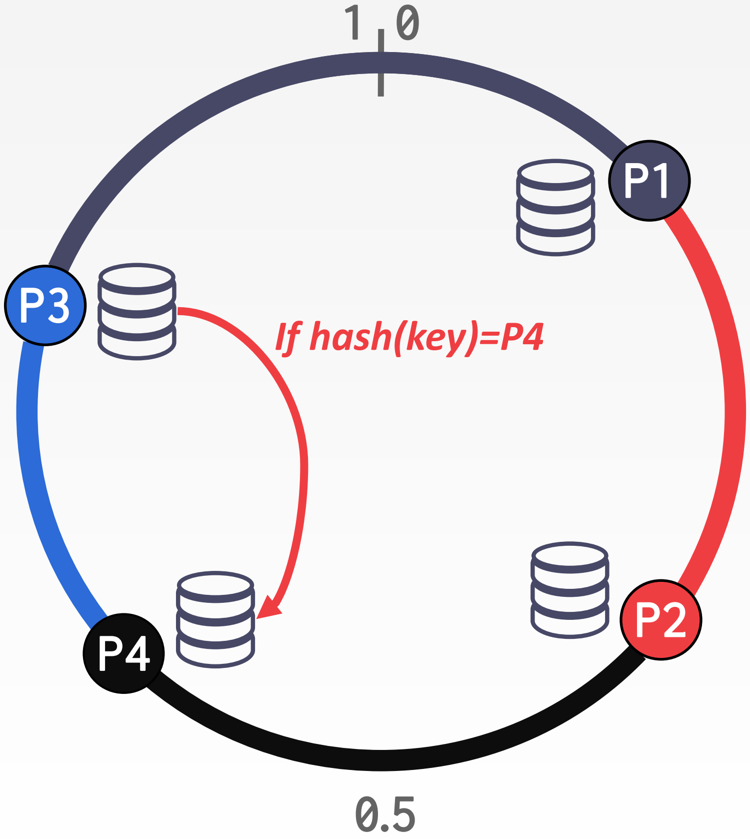
CONSISTENT HASHING



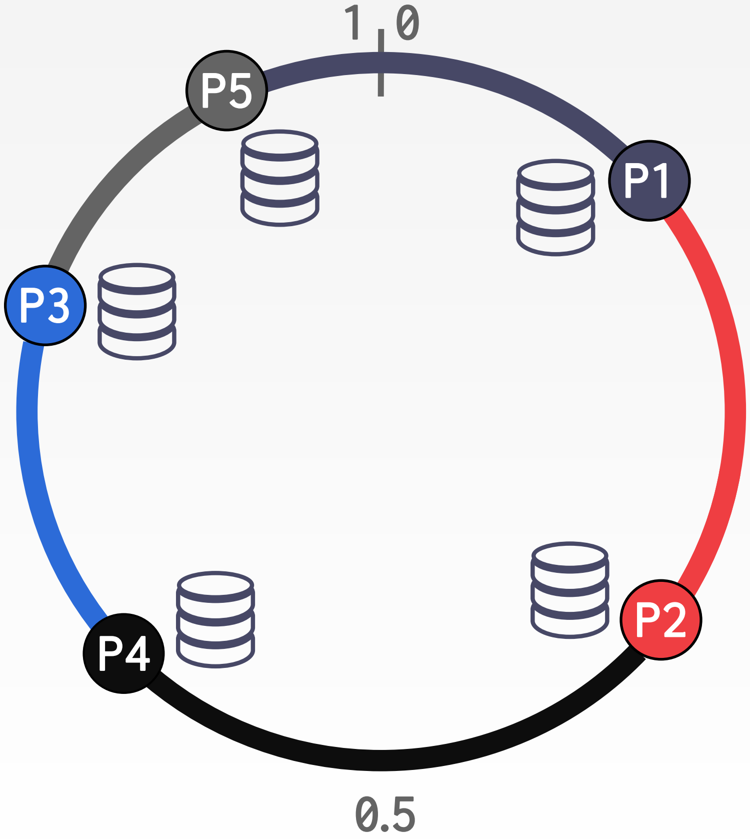
CONSISTENT HASHING



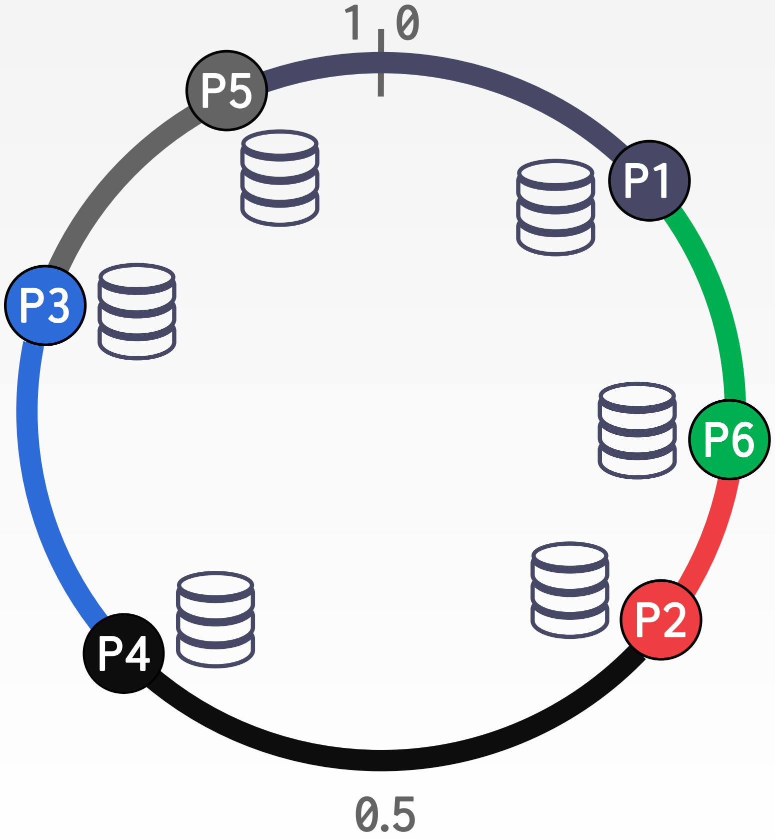
CONSISTENT HASHING



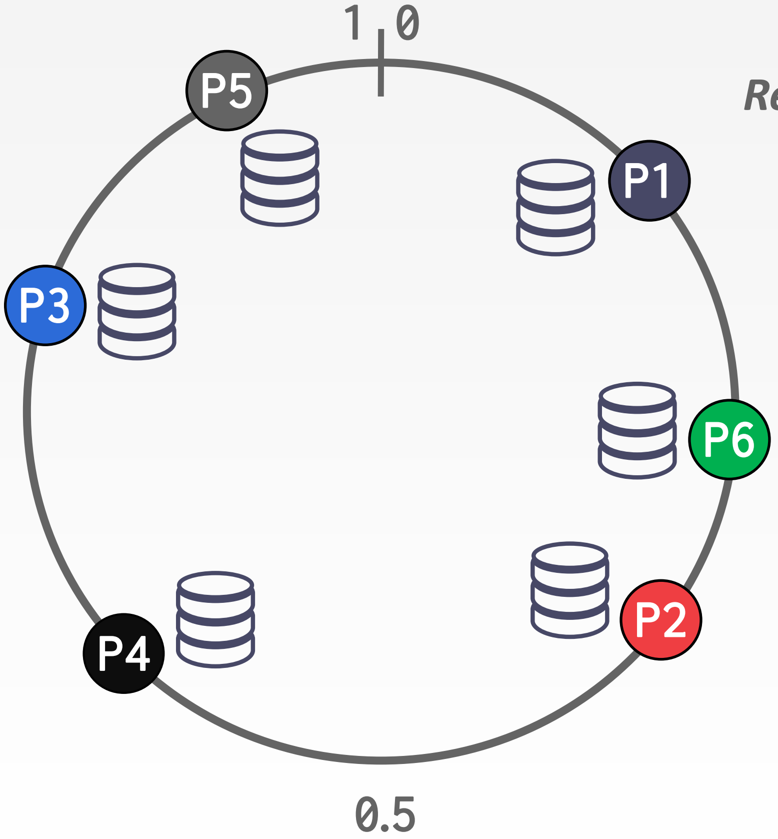
CONSISTENT HASHING



CONSISTENT HASHING

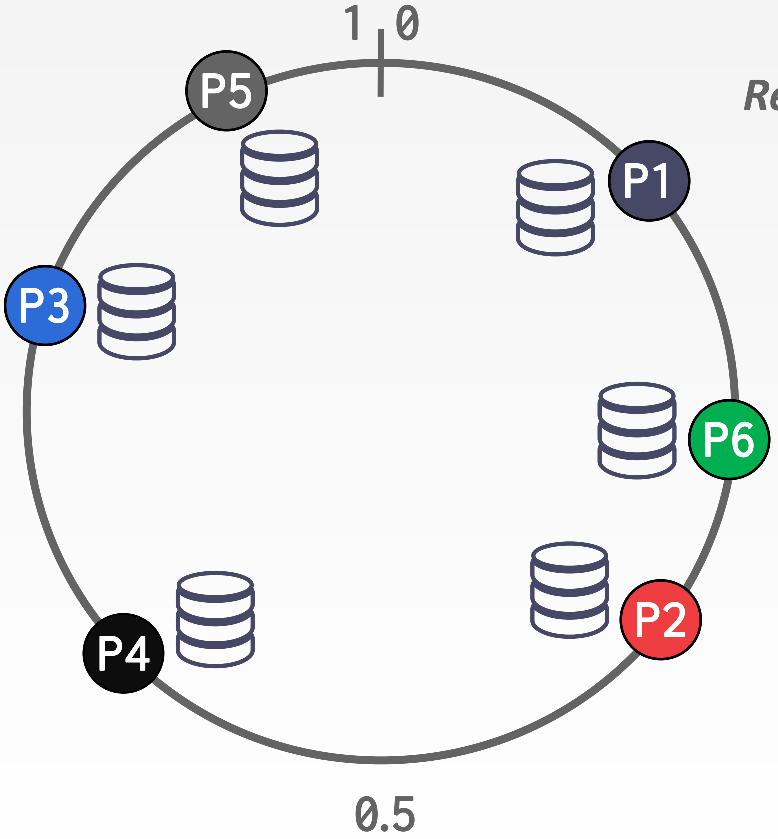


CONSISTENT HASHING



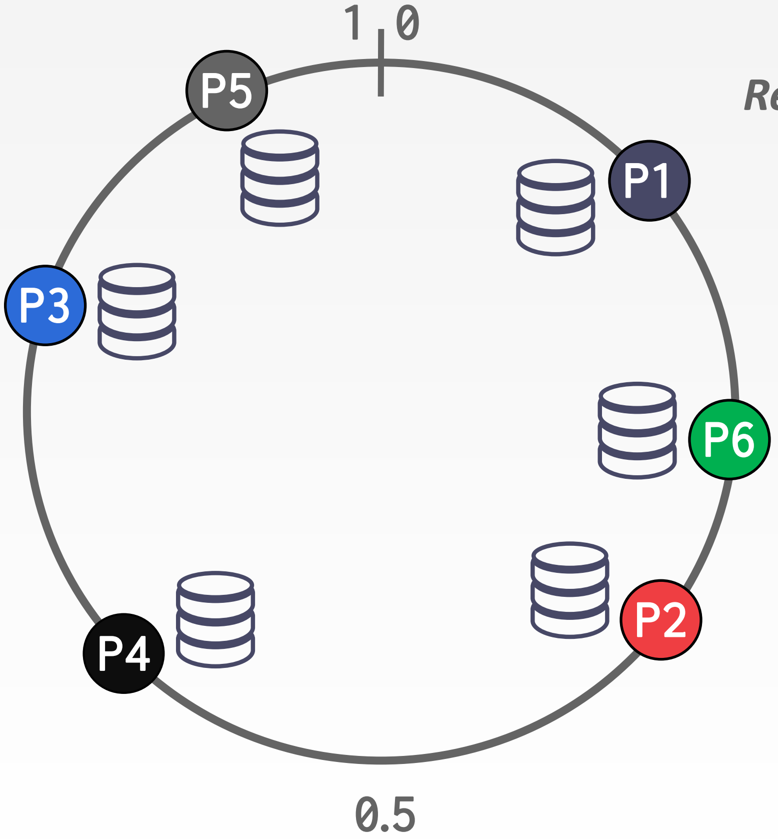
Replication Factor = 3

CONSISTENT HASHING



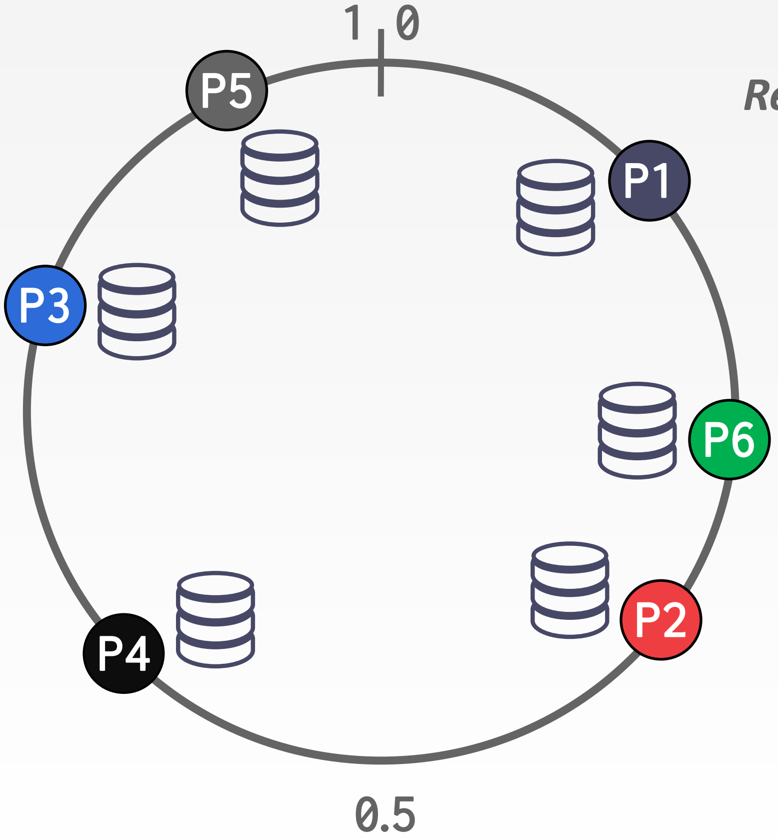
Replication Factor = 3

CONSISTENT HASHING



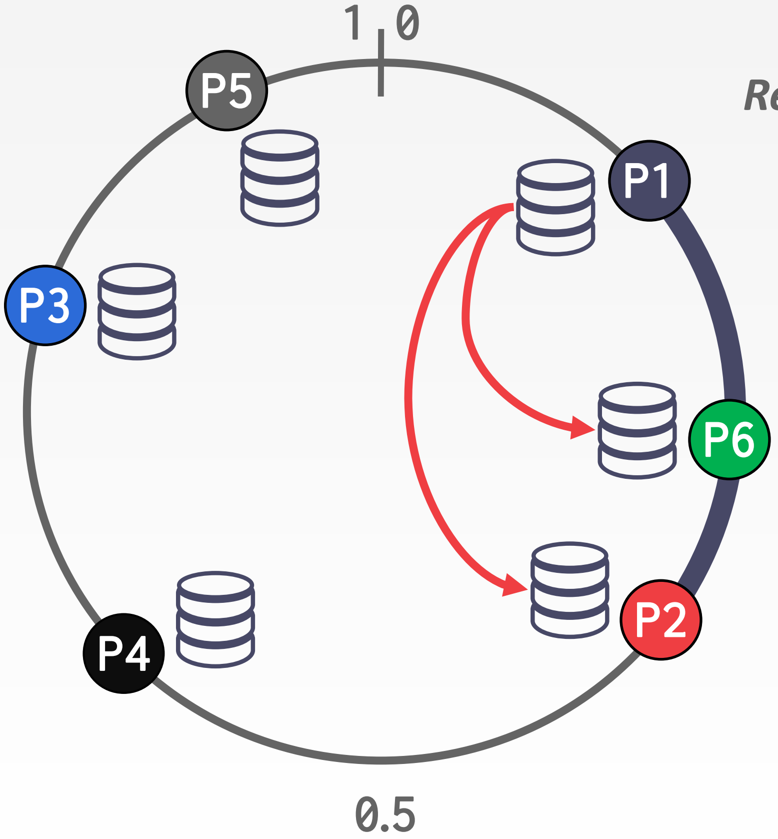
Replication Factor = 3

CONSISTENT HASHING



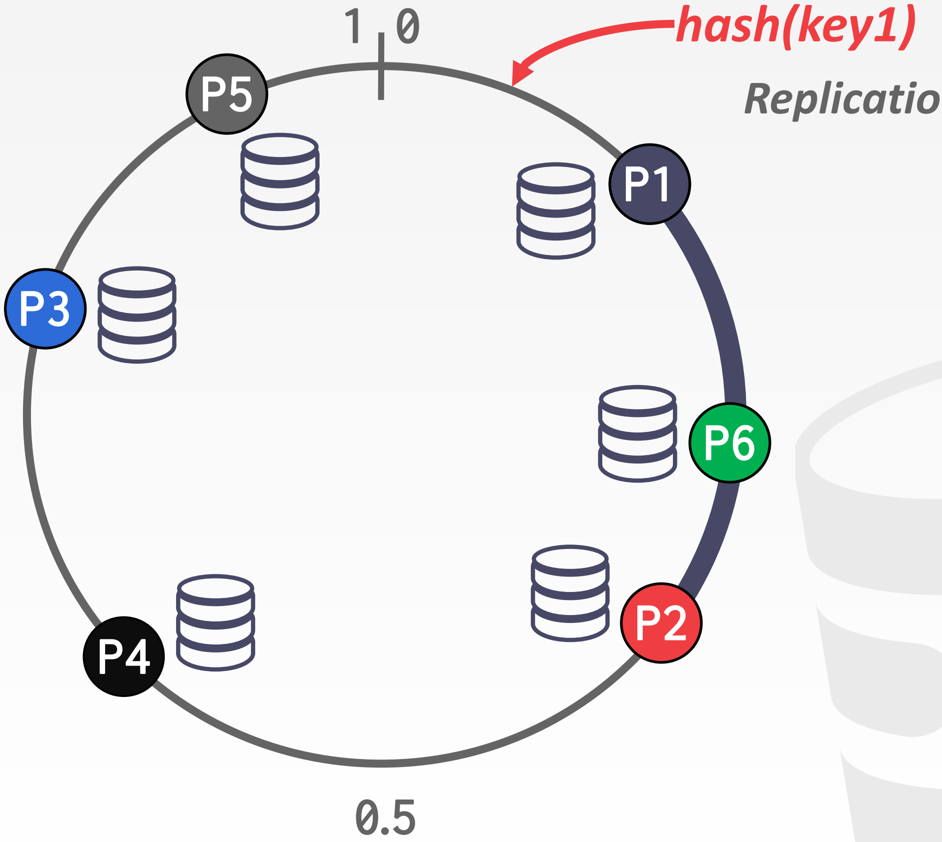
Replication Factor = 3

CONSISTENT HASHING



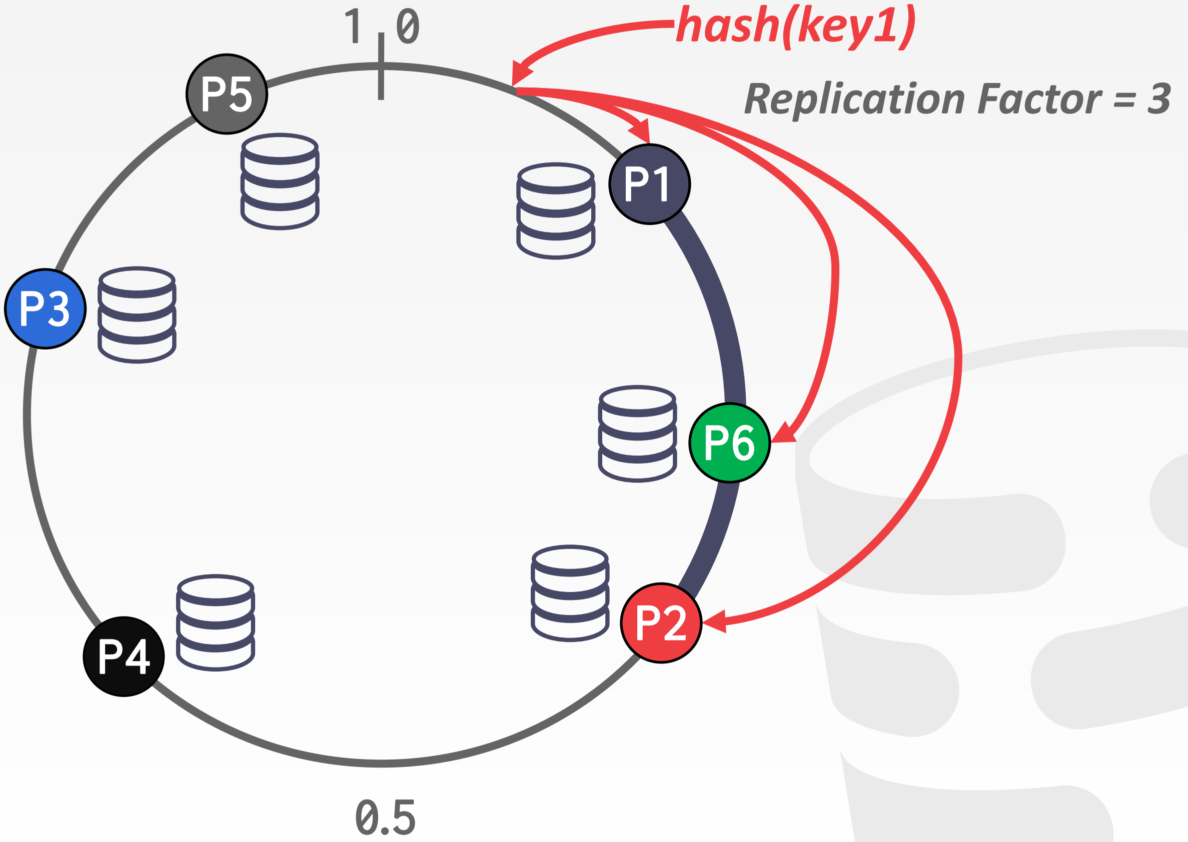
Replication Factor = 3

CONSISTENT HASHING



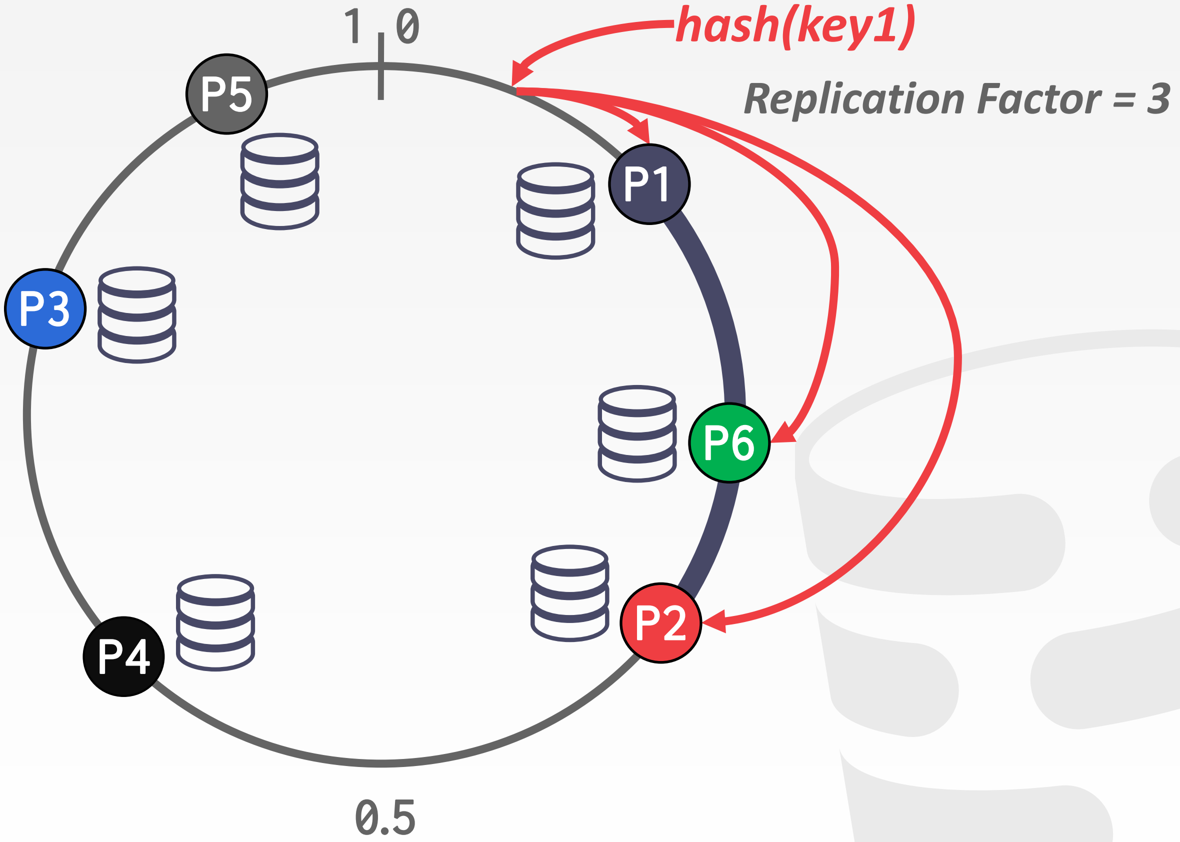
Replication Factor = 3

CONSISTENT HASHING

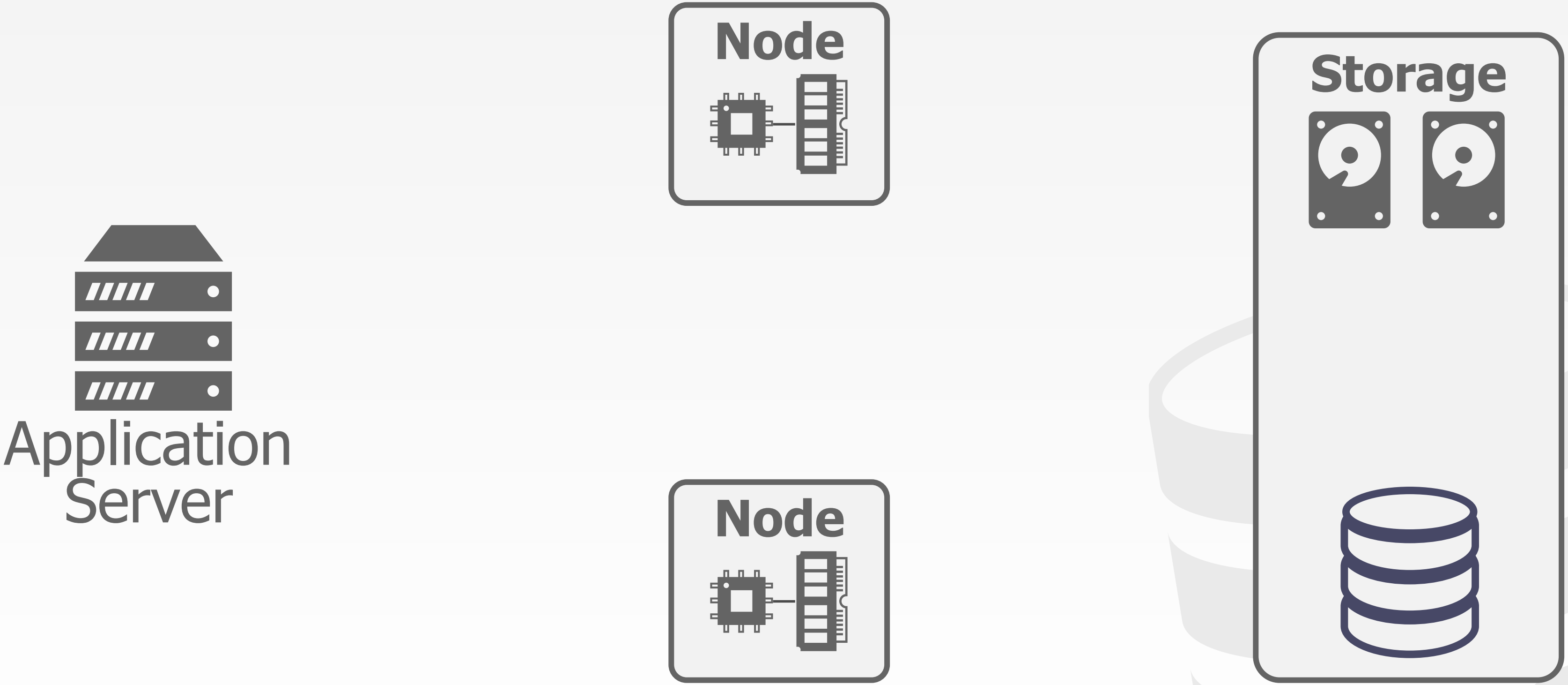


Replication Factor = 3

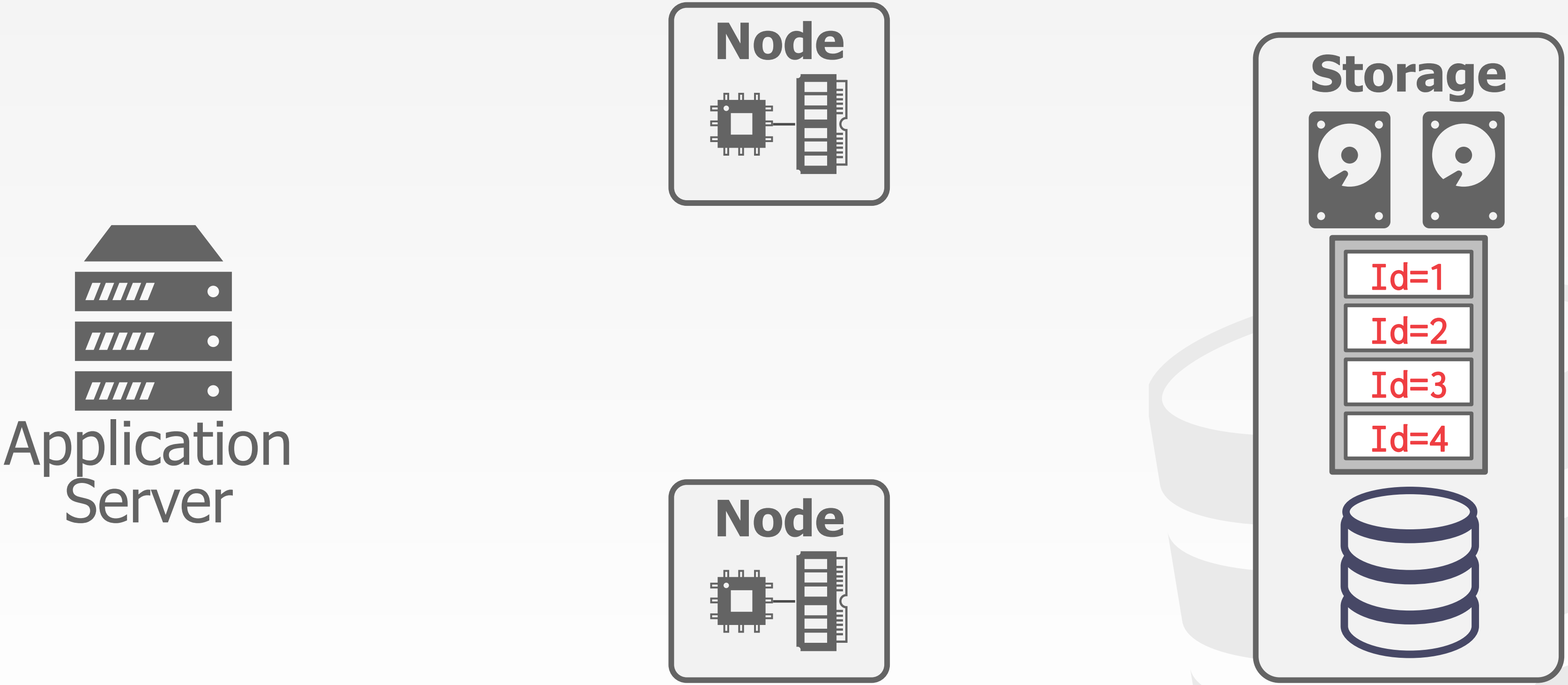
CONSISTENT HASHING



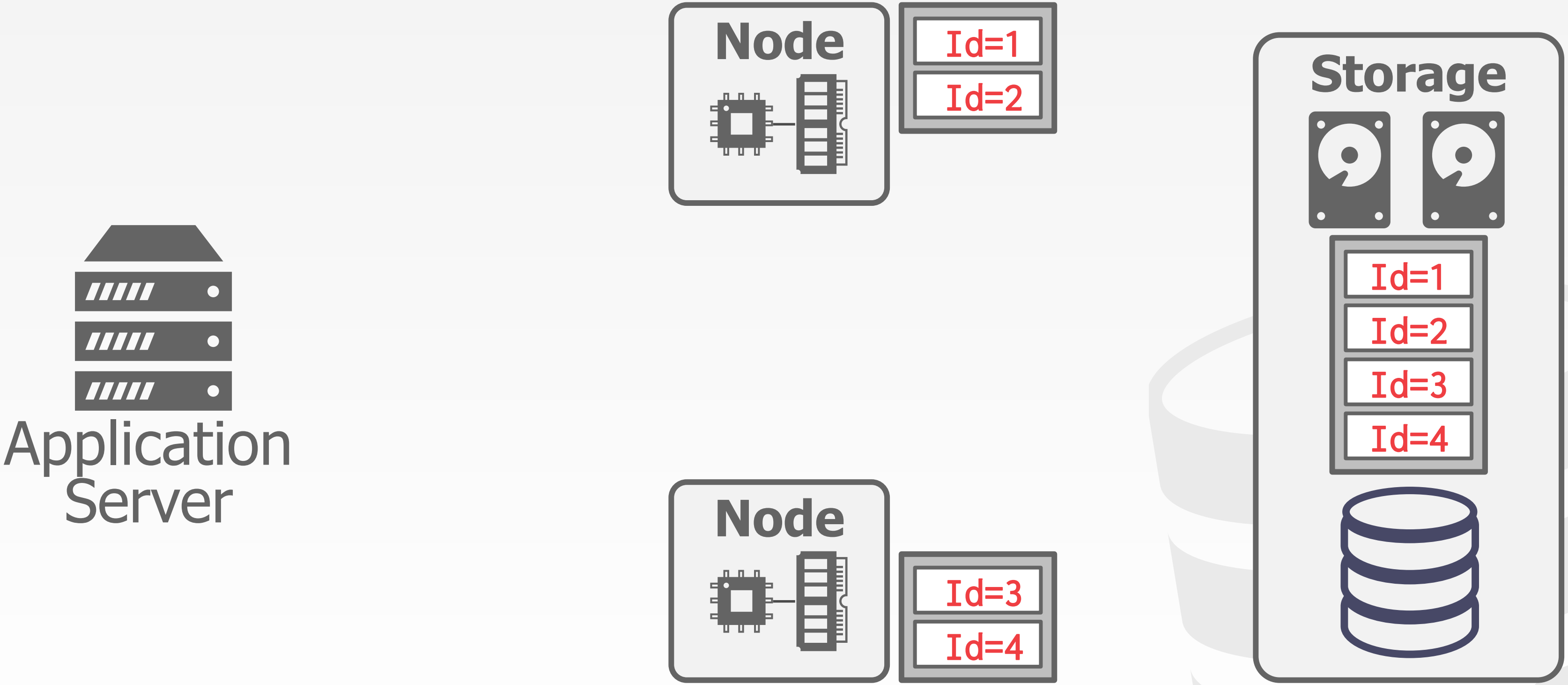
LOGICAL PARTITIONING



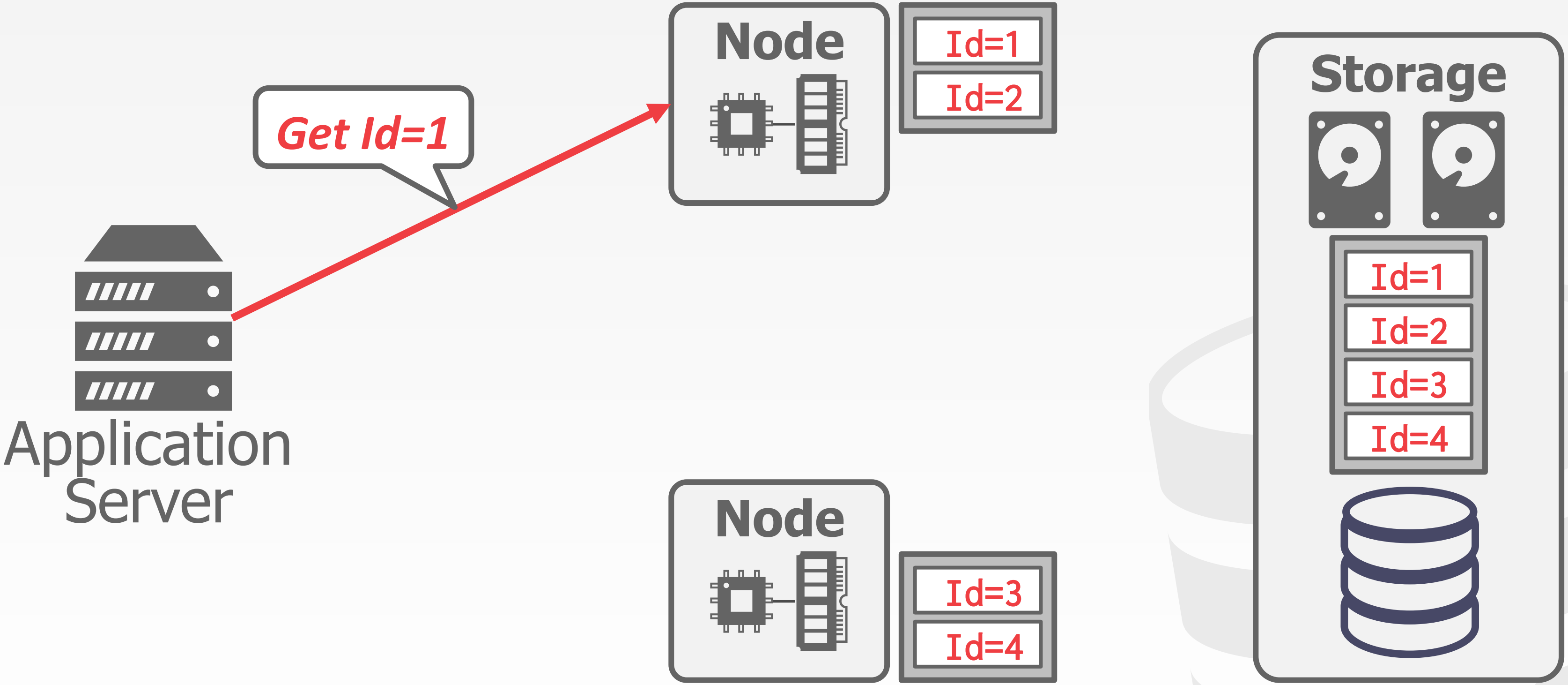
LOGICAL PARTITIONING



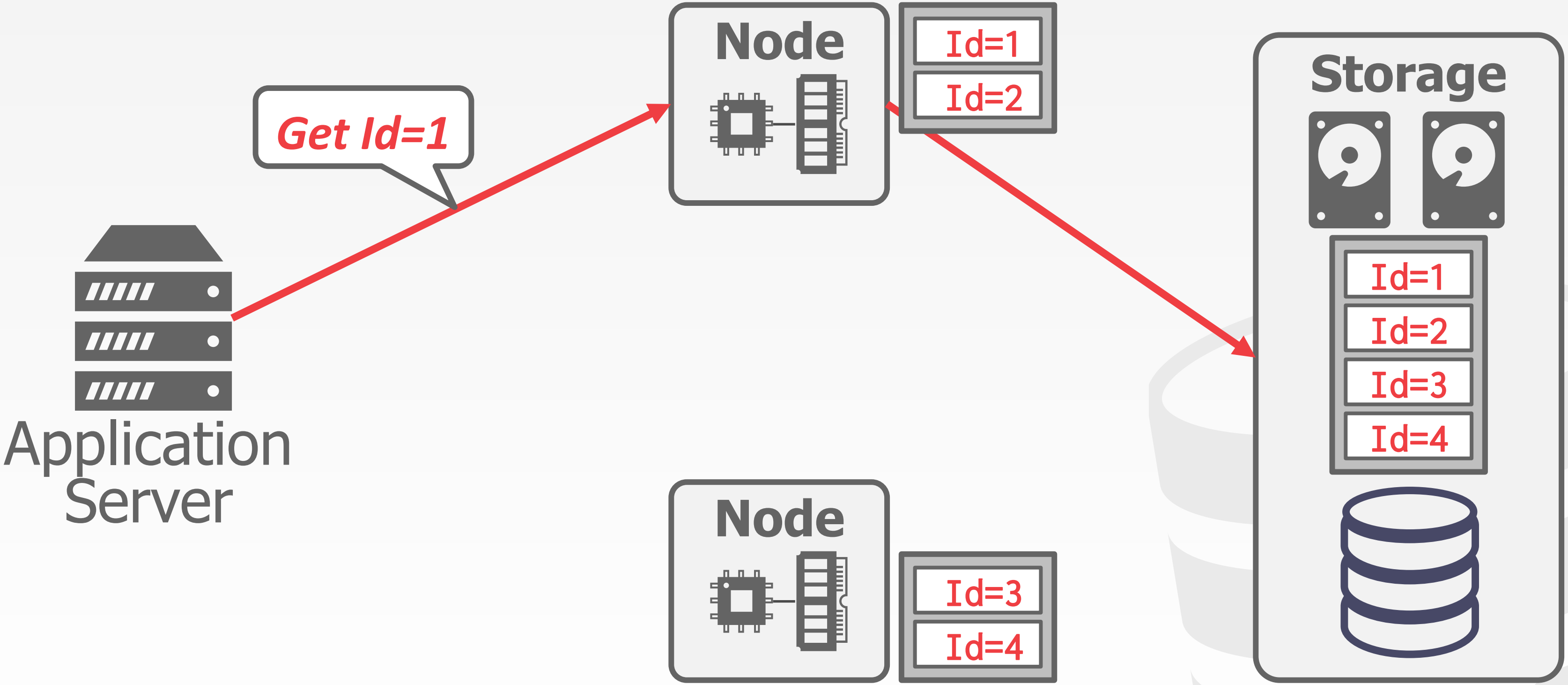
LOGICAL PARTITIONING



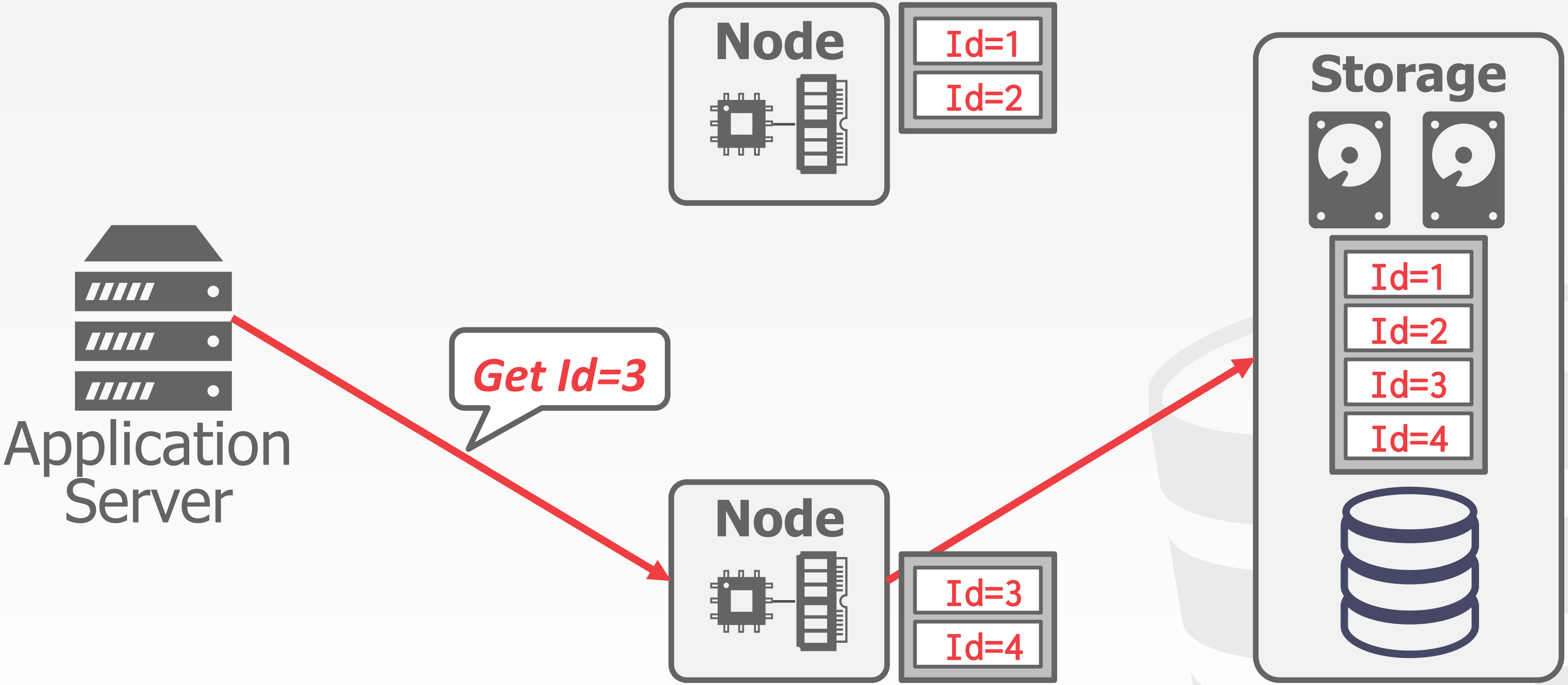
LOGICAL PARTITIONING



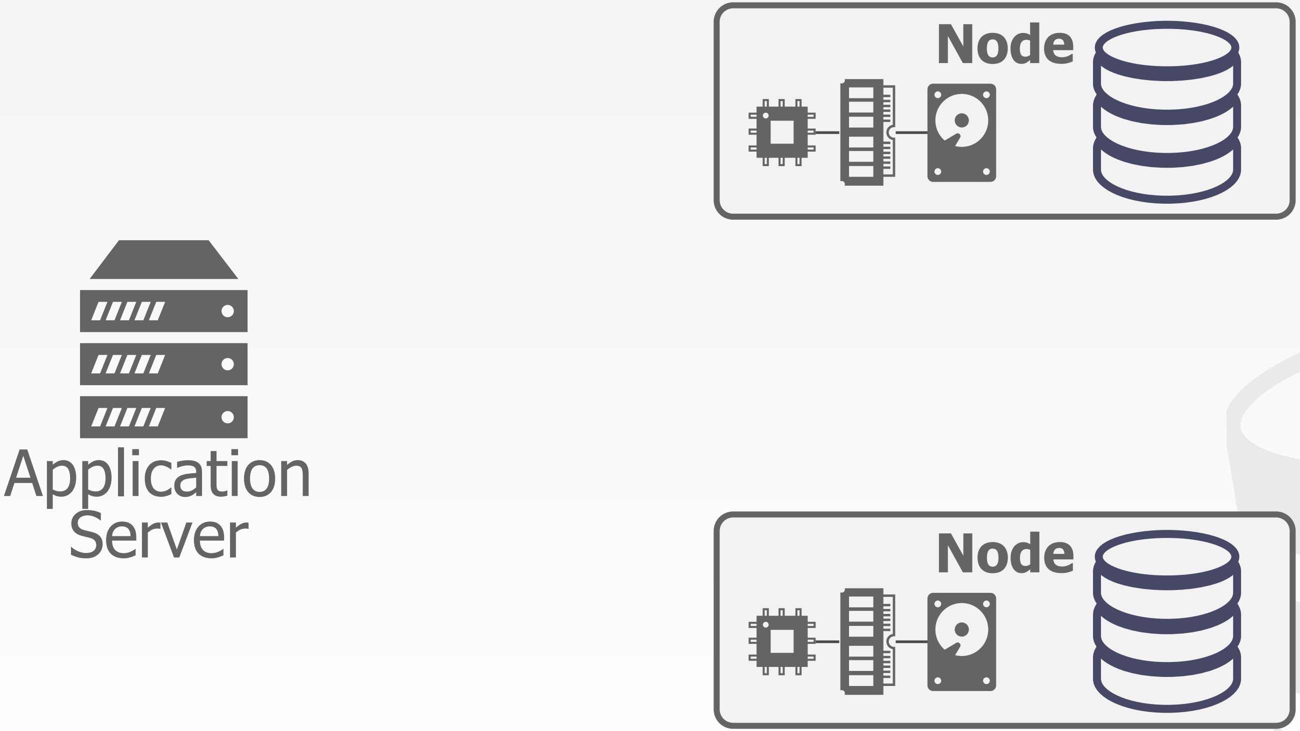
LOGICAL PARTITIONING



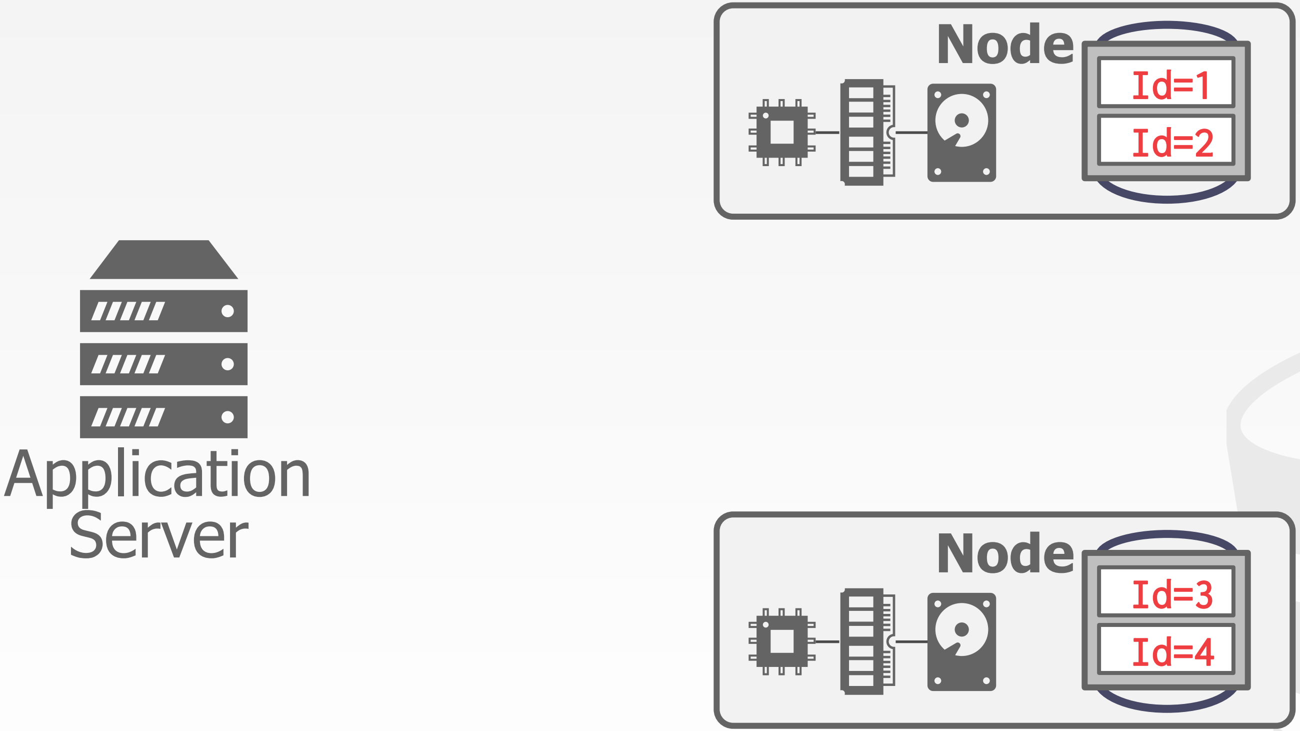
LOGICAL PARTITIONING



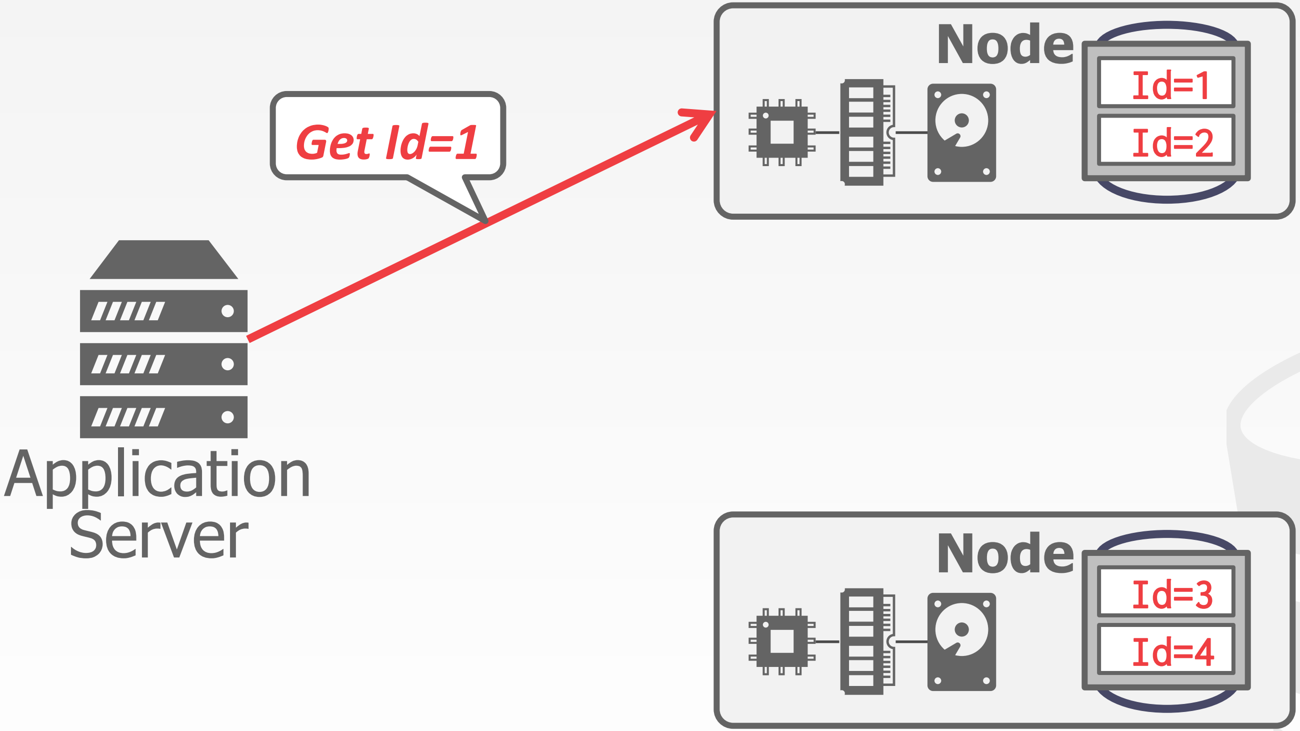
PHYSICAL PARTITIONING



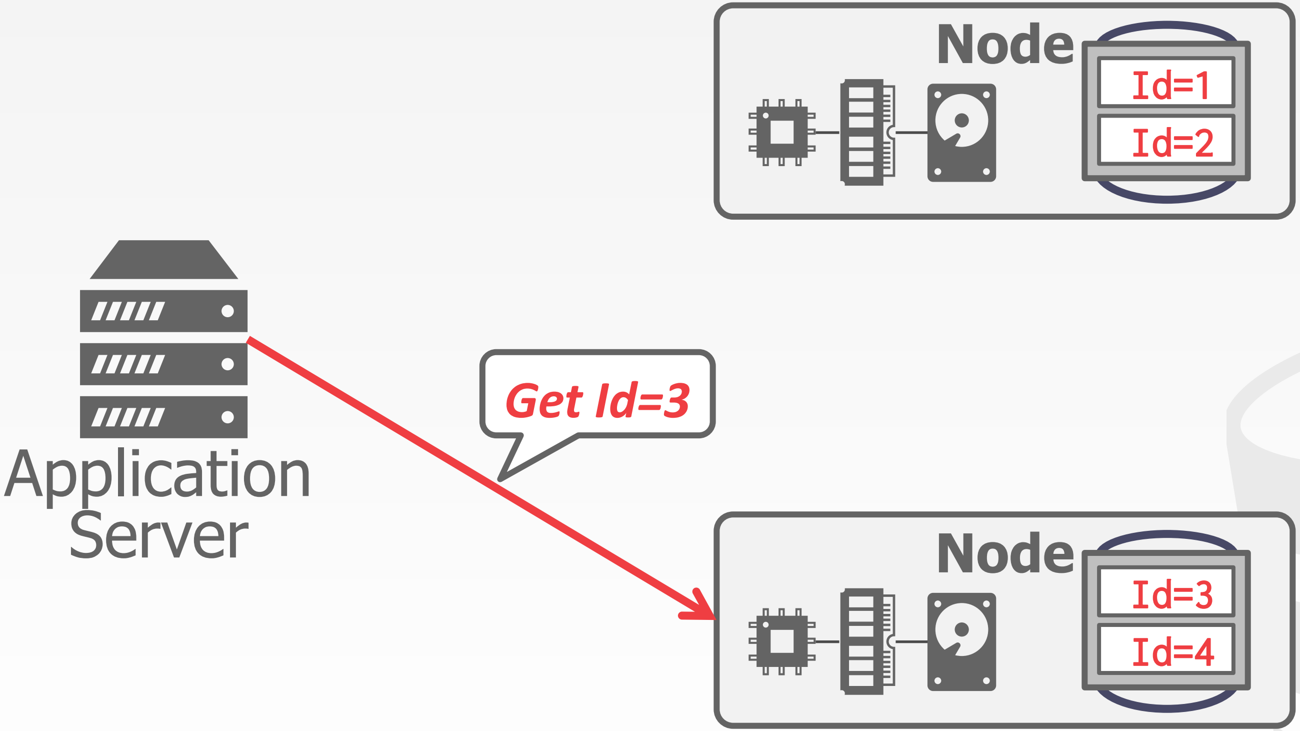
PHYSICAL PARTITIONING



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

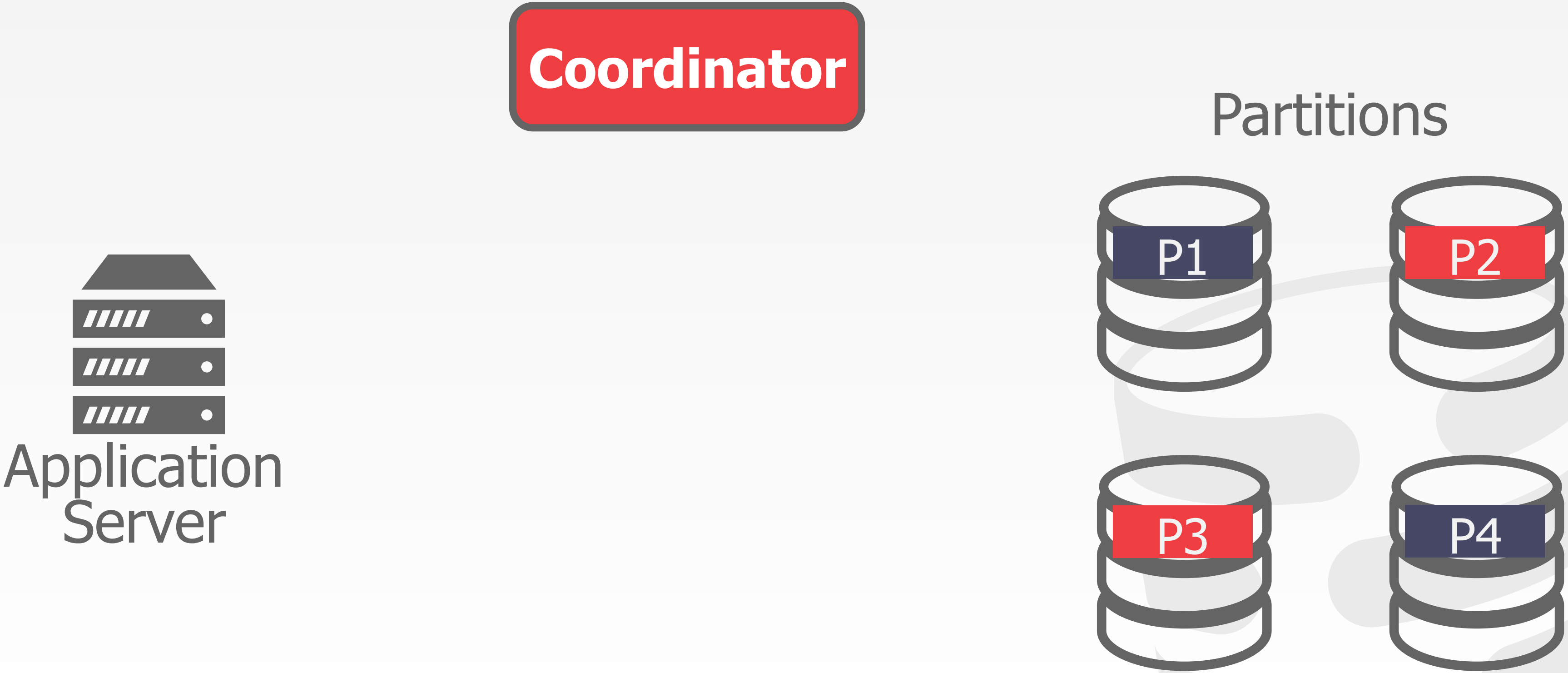
A **TP Monitor** is an example of a centralized coordinator for distributed DBMSs.

Originally developed in the 1970-80s to provide txns between terminals and mainframe databases.

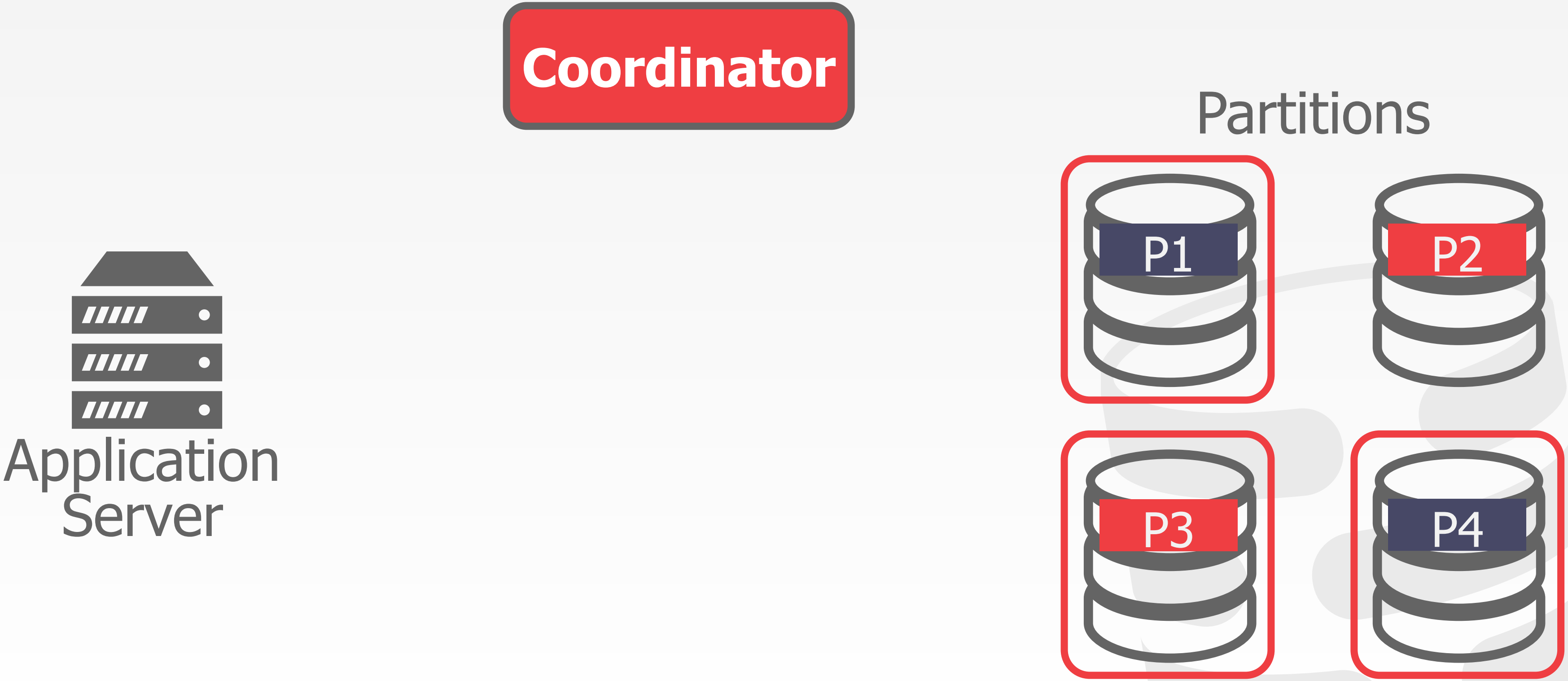
→ Examples: ATMs, Airline Reservations.

Many DBMSs now support the same functionality internally.

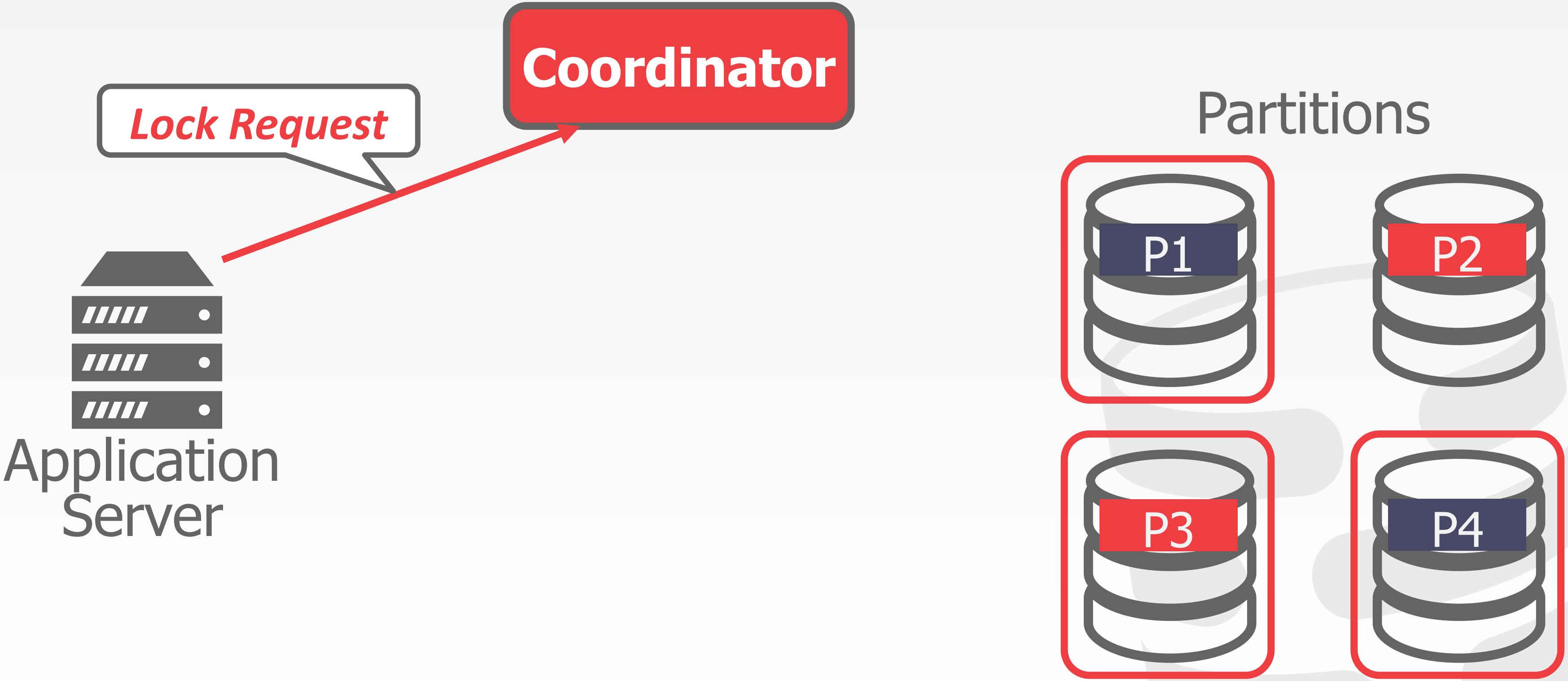
CENTRALIZED COORDINATOR



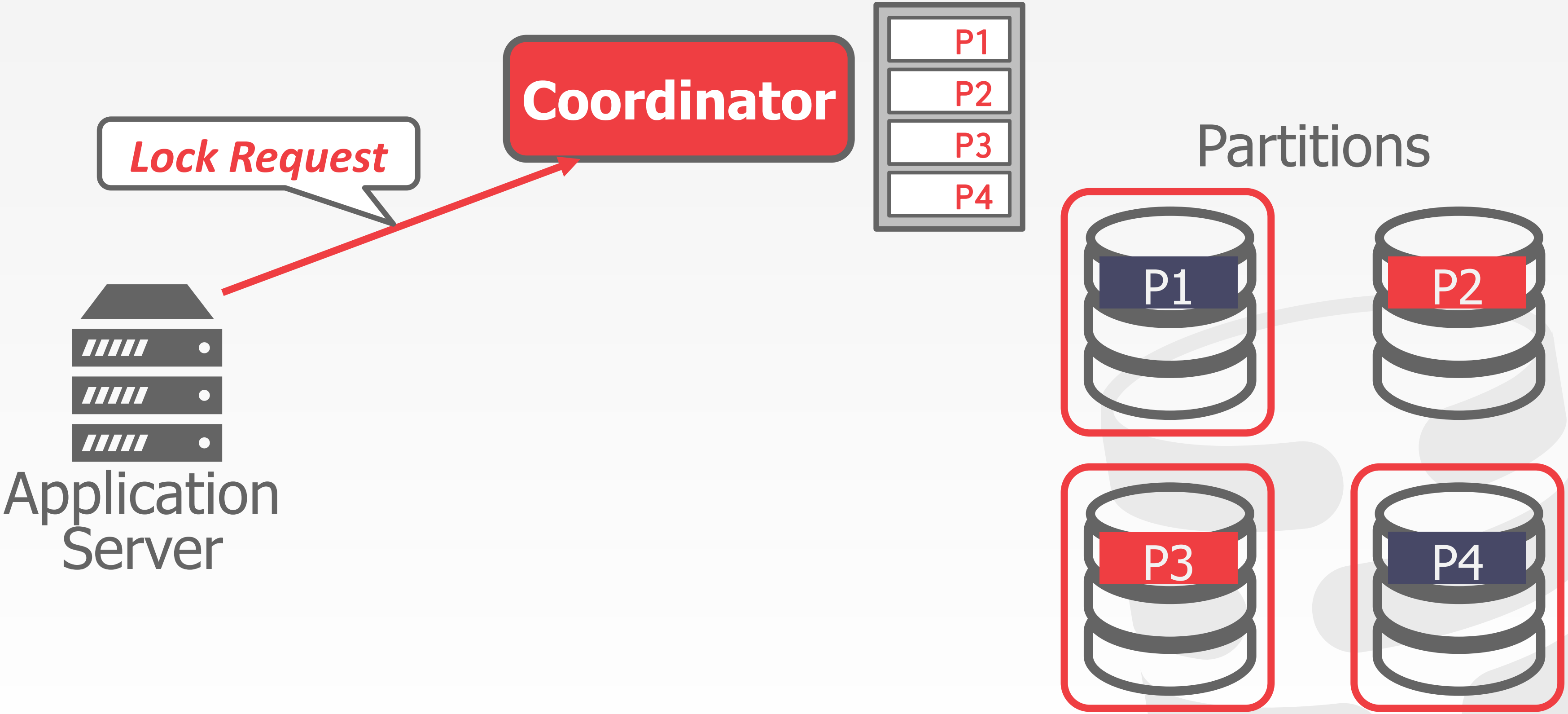
CENTRALIZED COORDINATOR



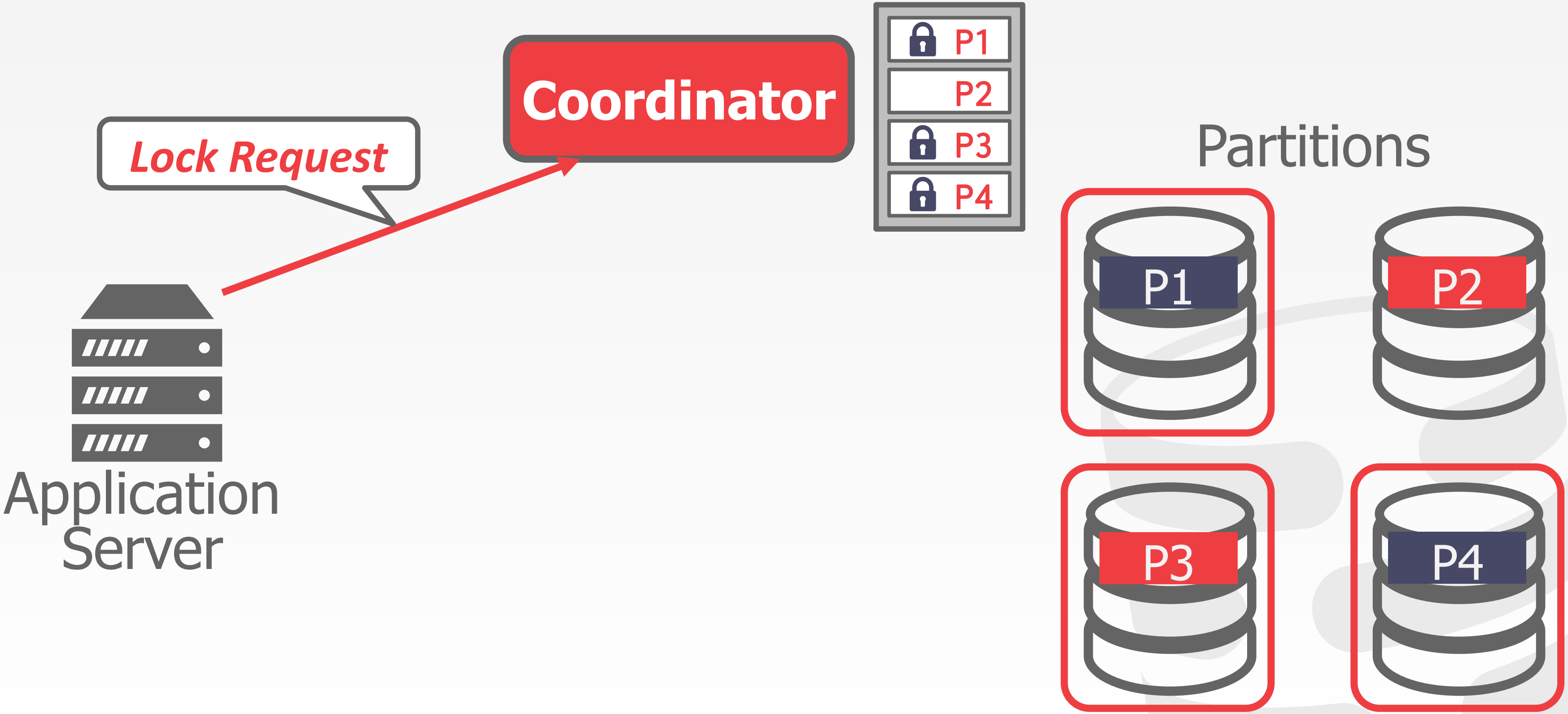
CENTRALIZED COORDINATOR



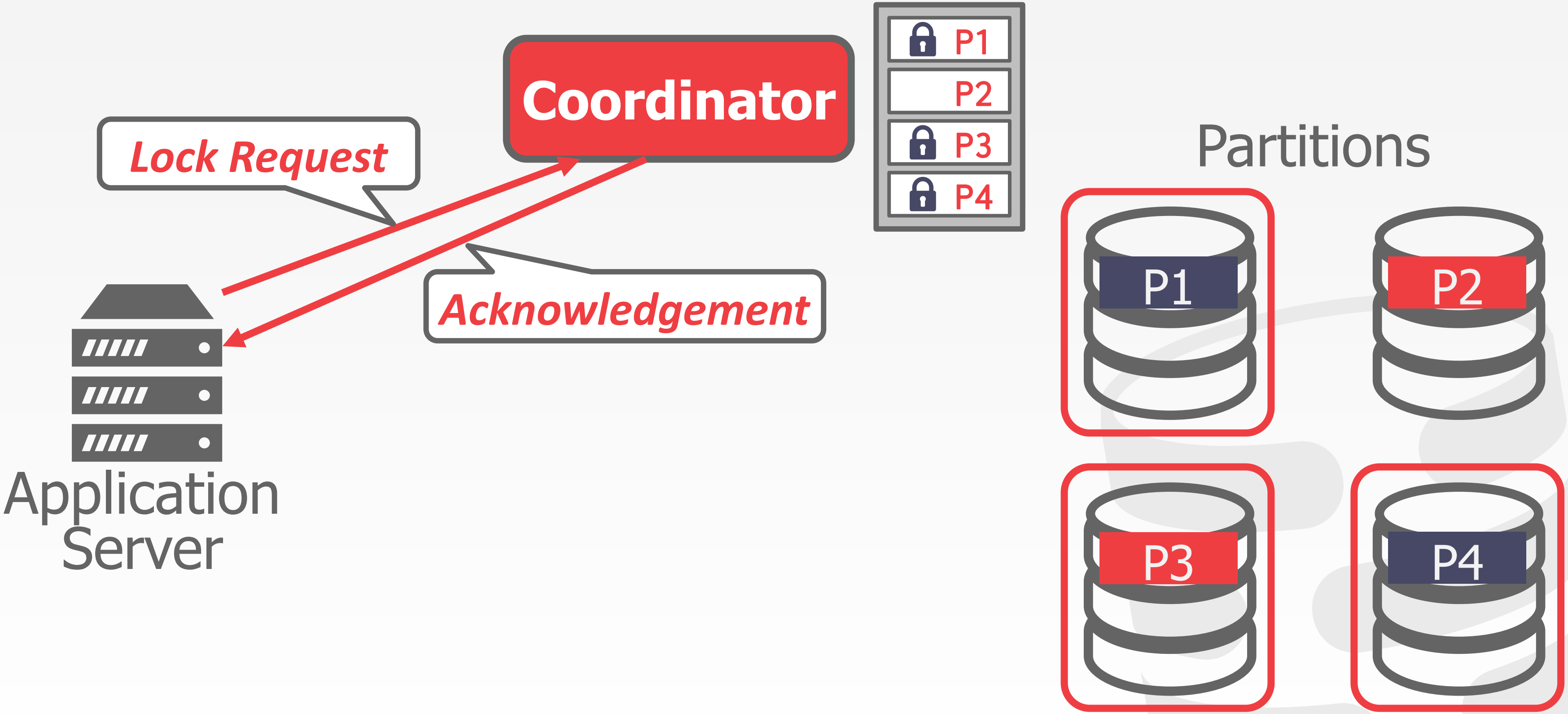
CENTRALIZED COORDINATOR



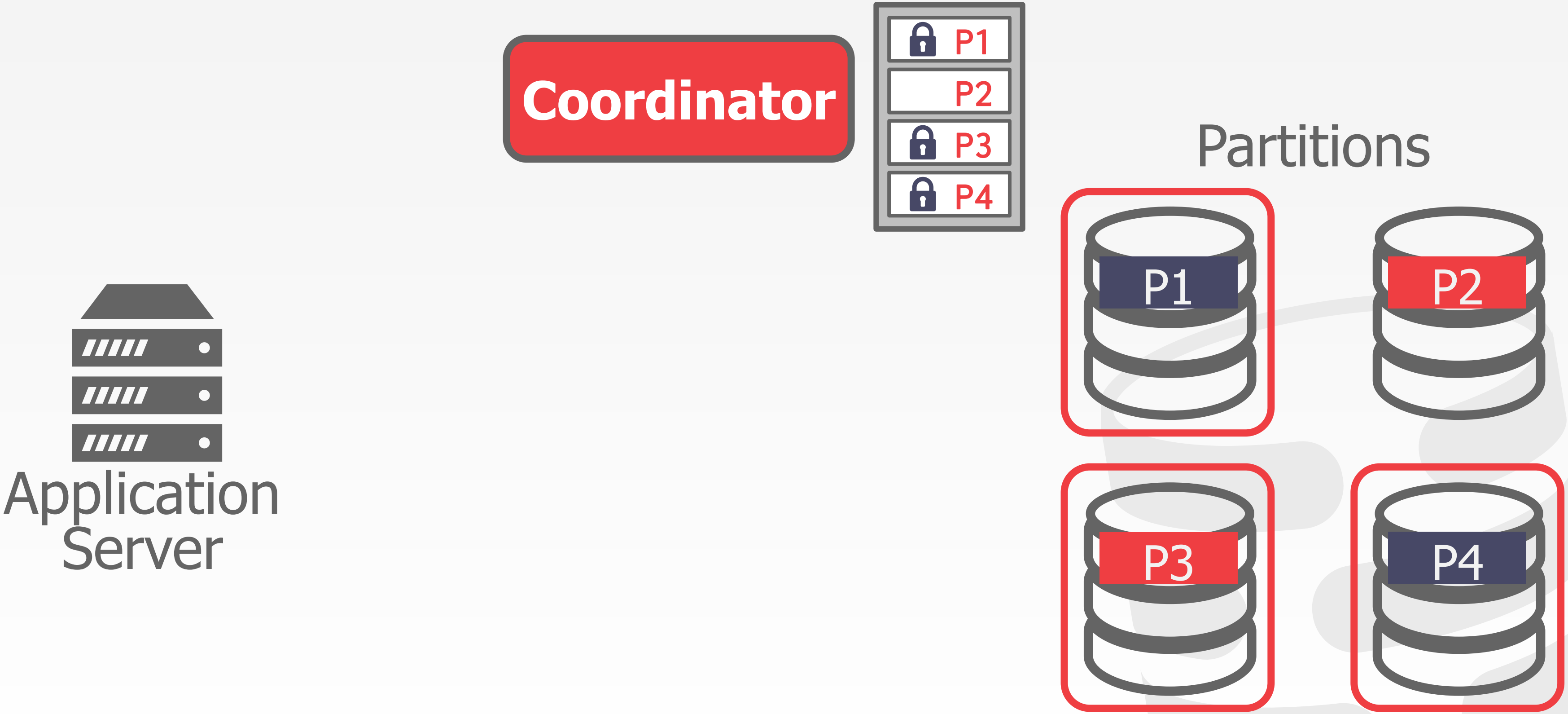
CENTRALIZED COORDINATOR



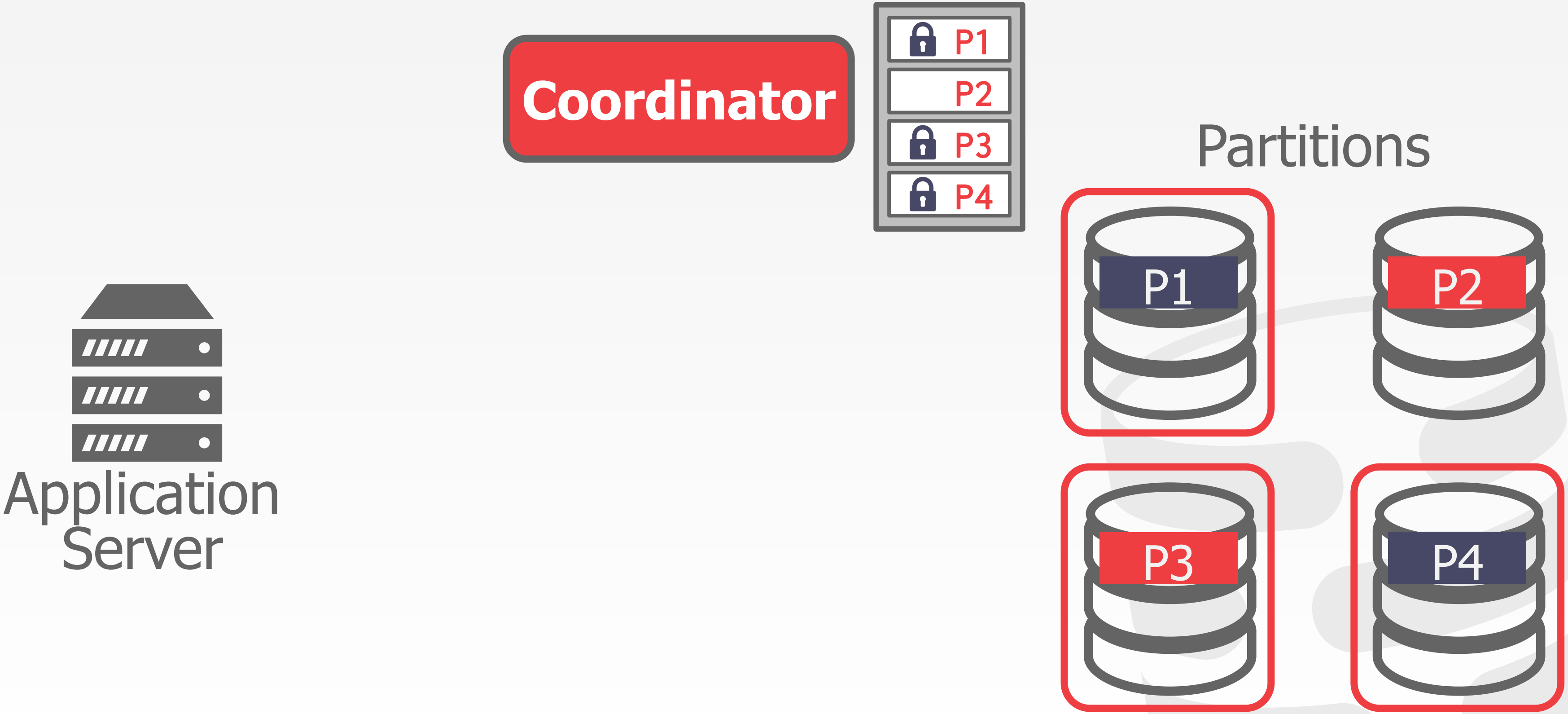
CENTRALIZED COORDINATOR



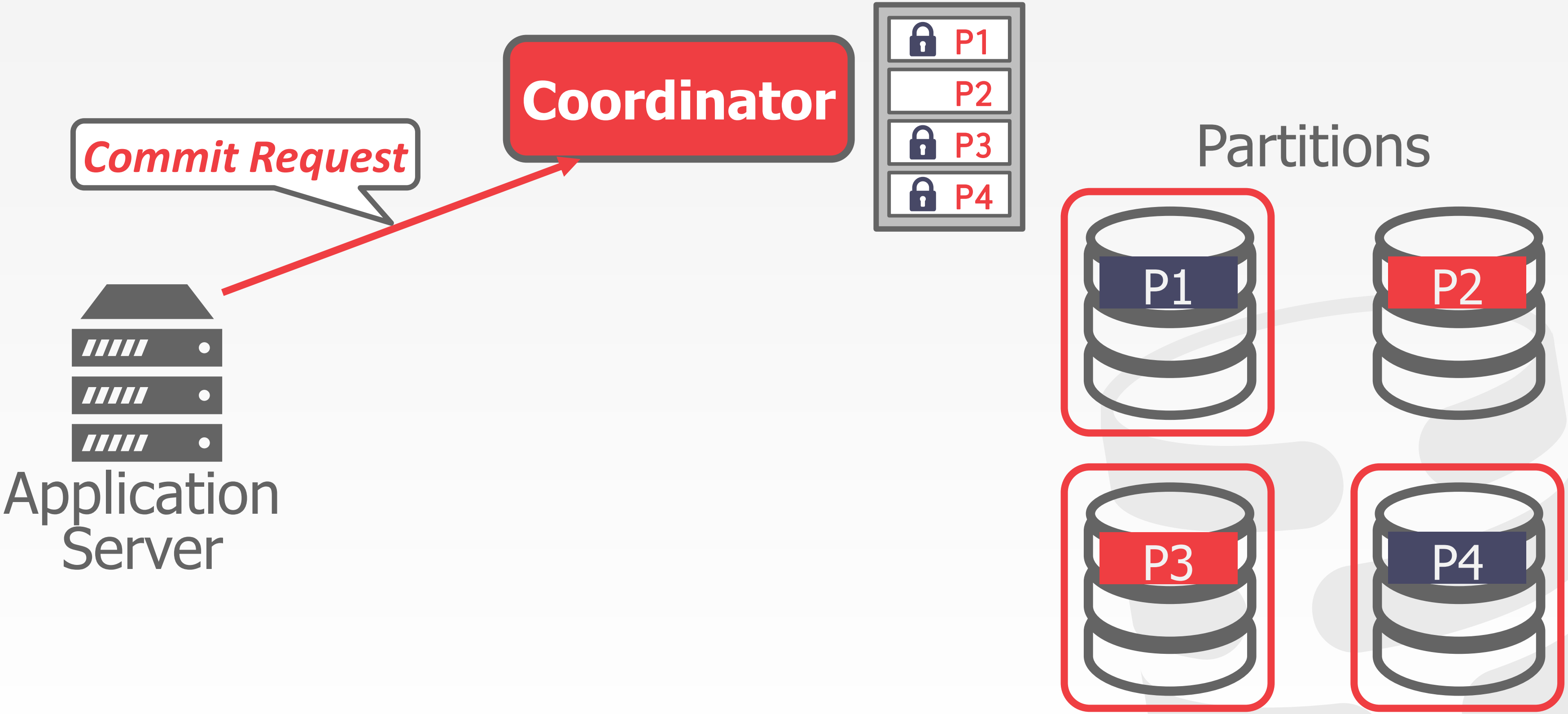
CENTRALIZED COORDINATOR



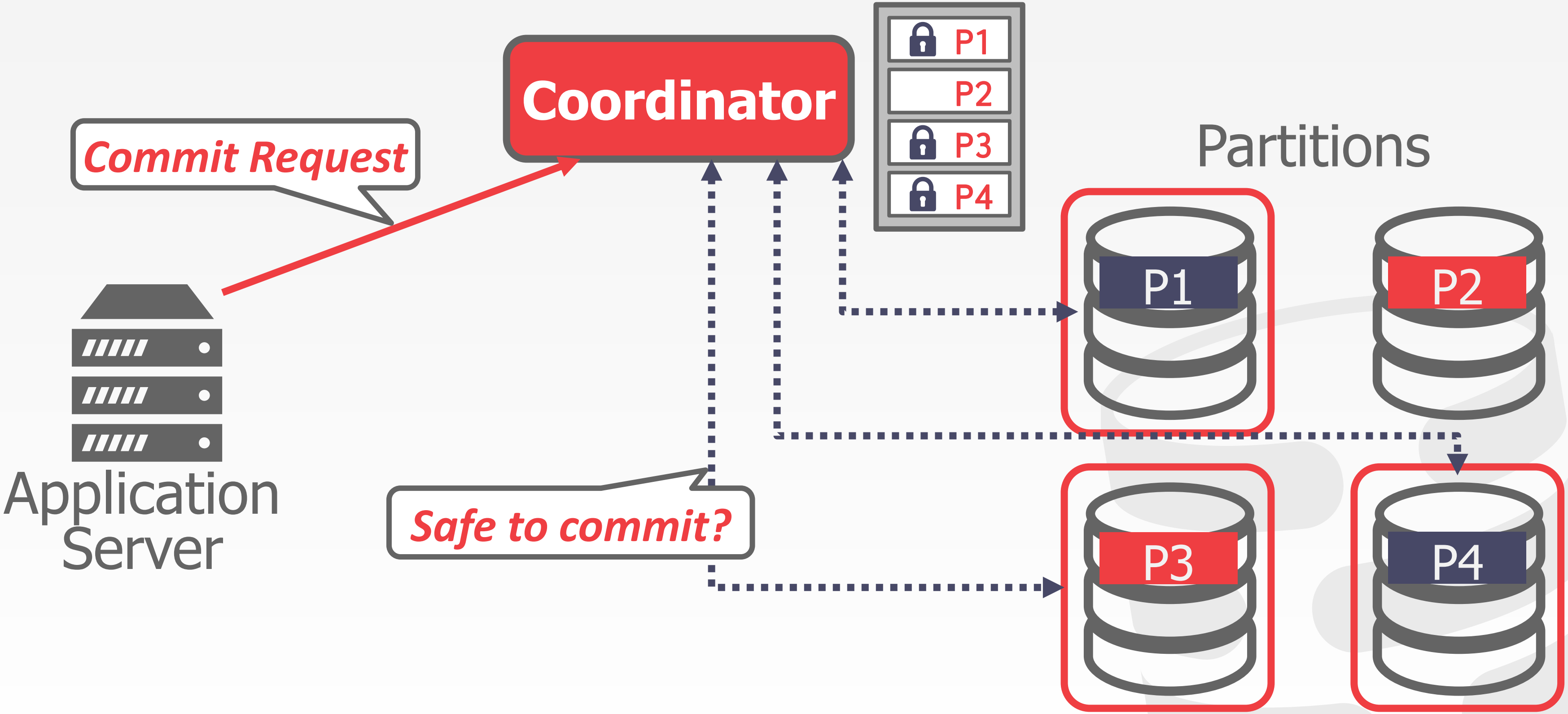
CENTRALIZED COORDINATOR



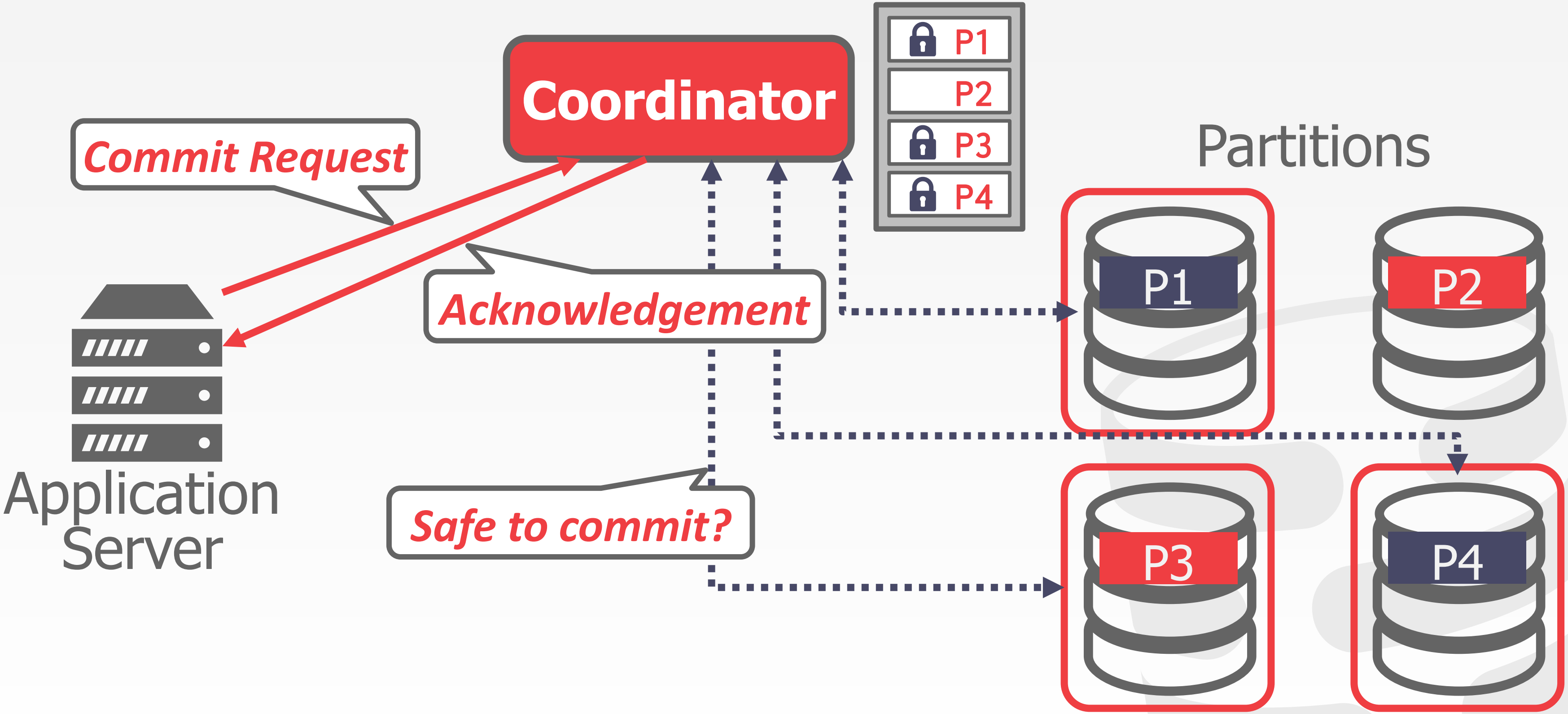
CENTRALIZED COORDINATOR



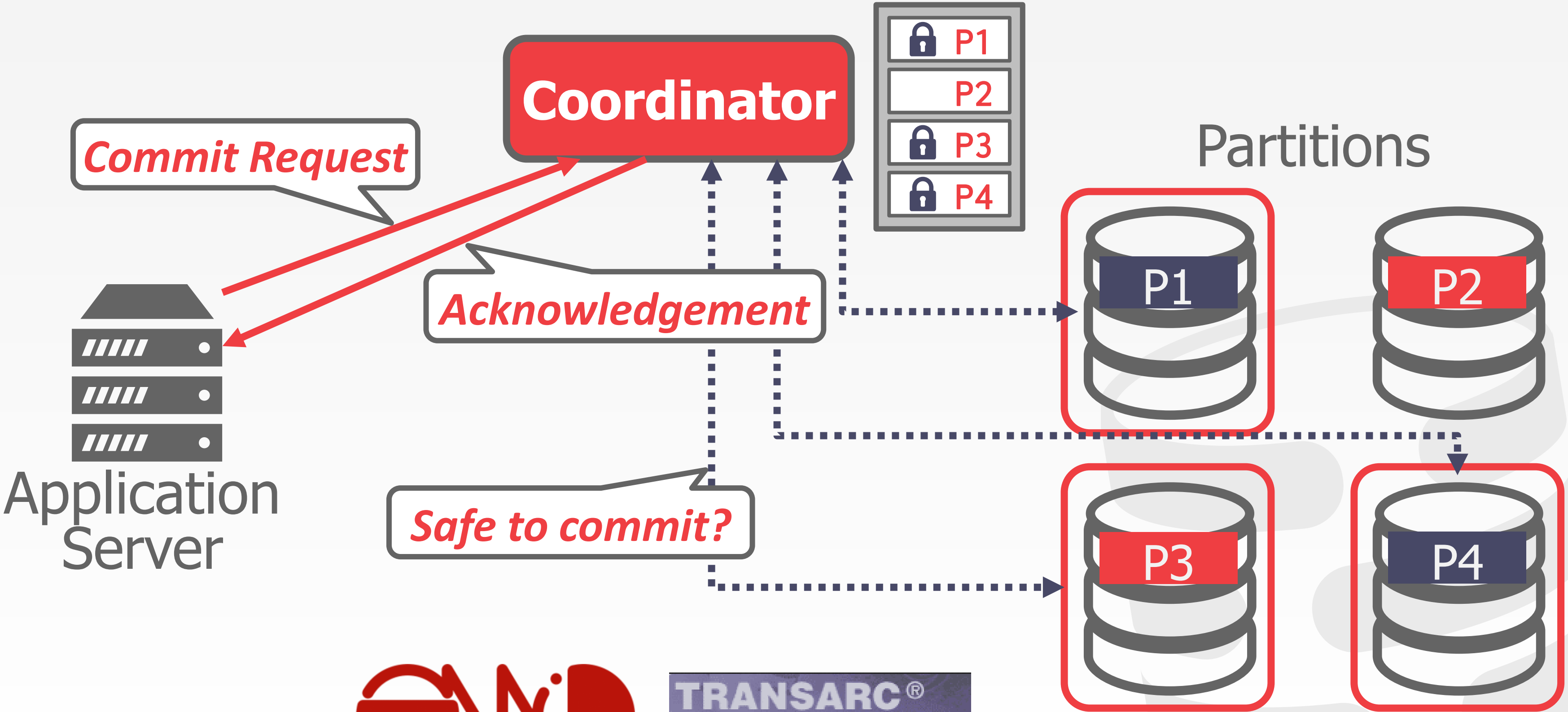
CENTRALIZED COORDINATOR



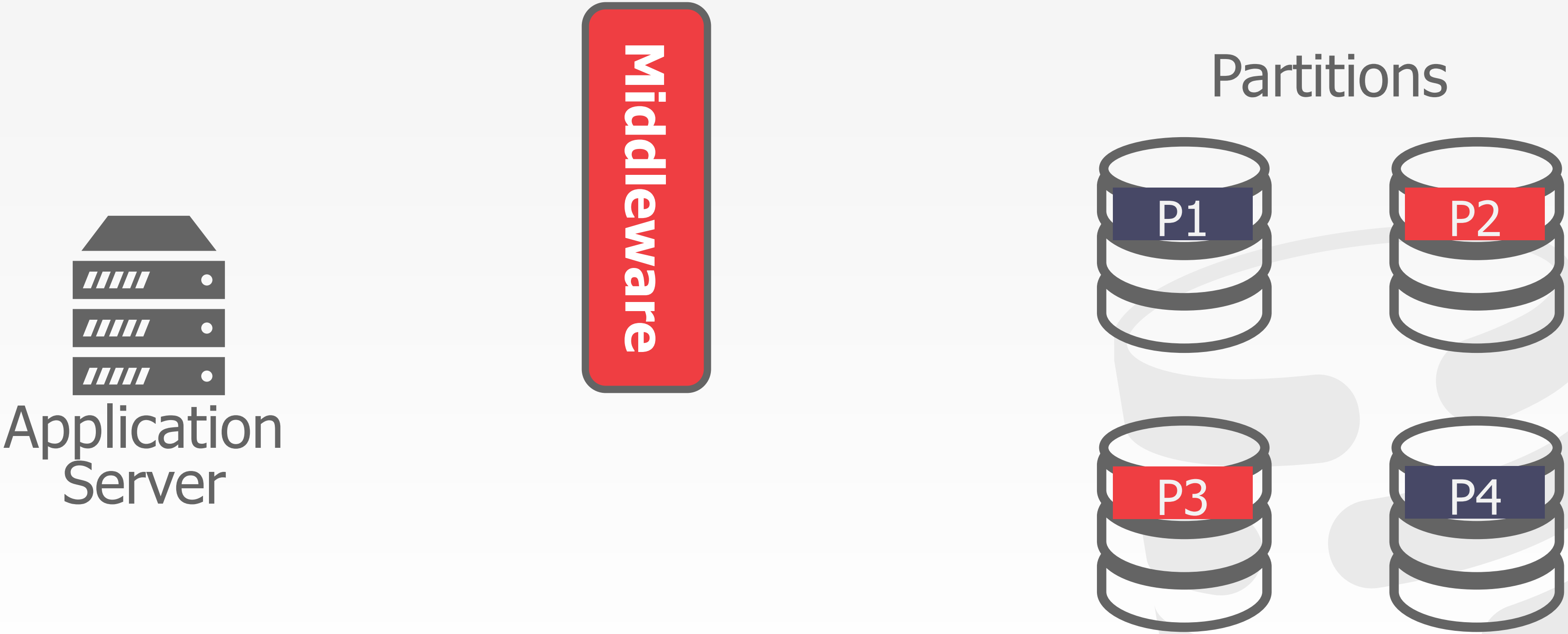
CENTRALIZED COORDINATOR



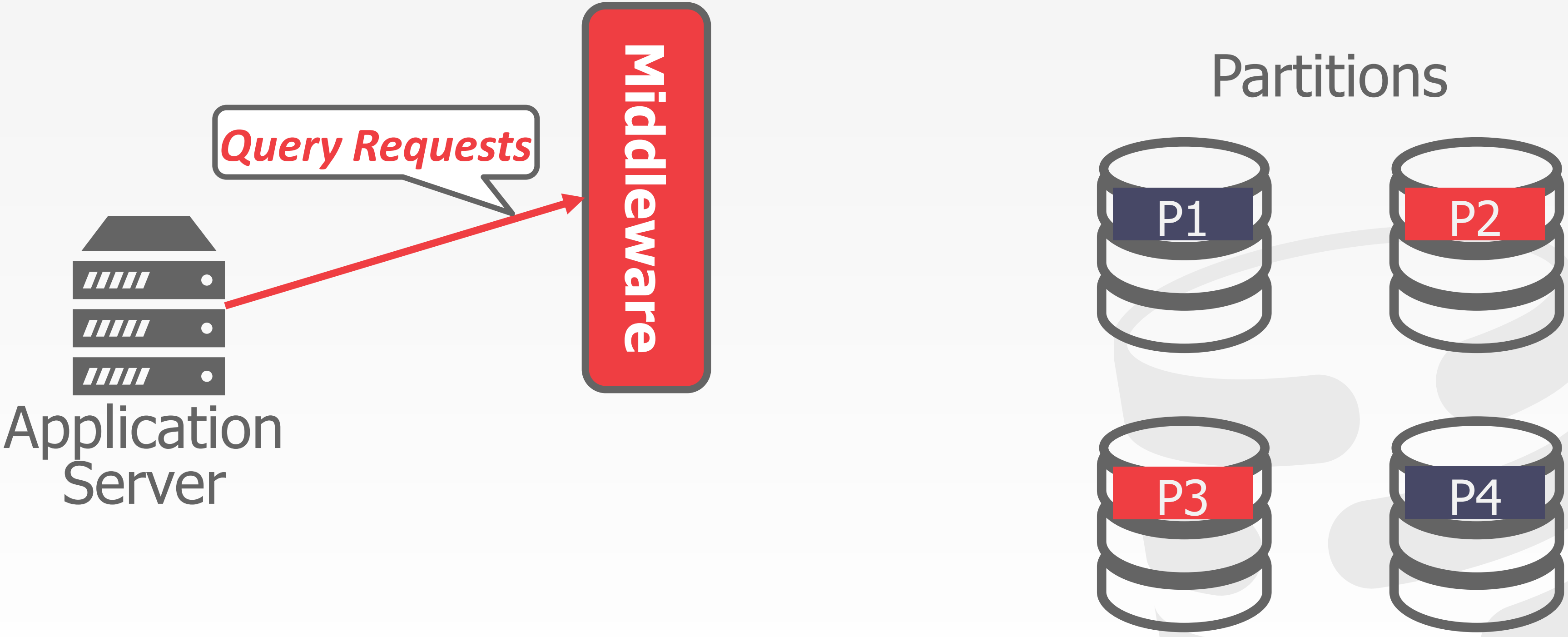
CENTRALIZED COORDINATOR



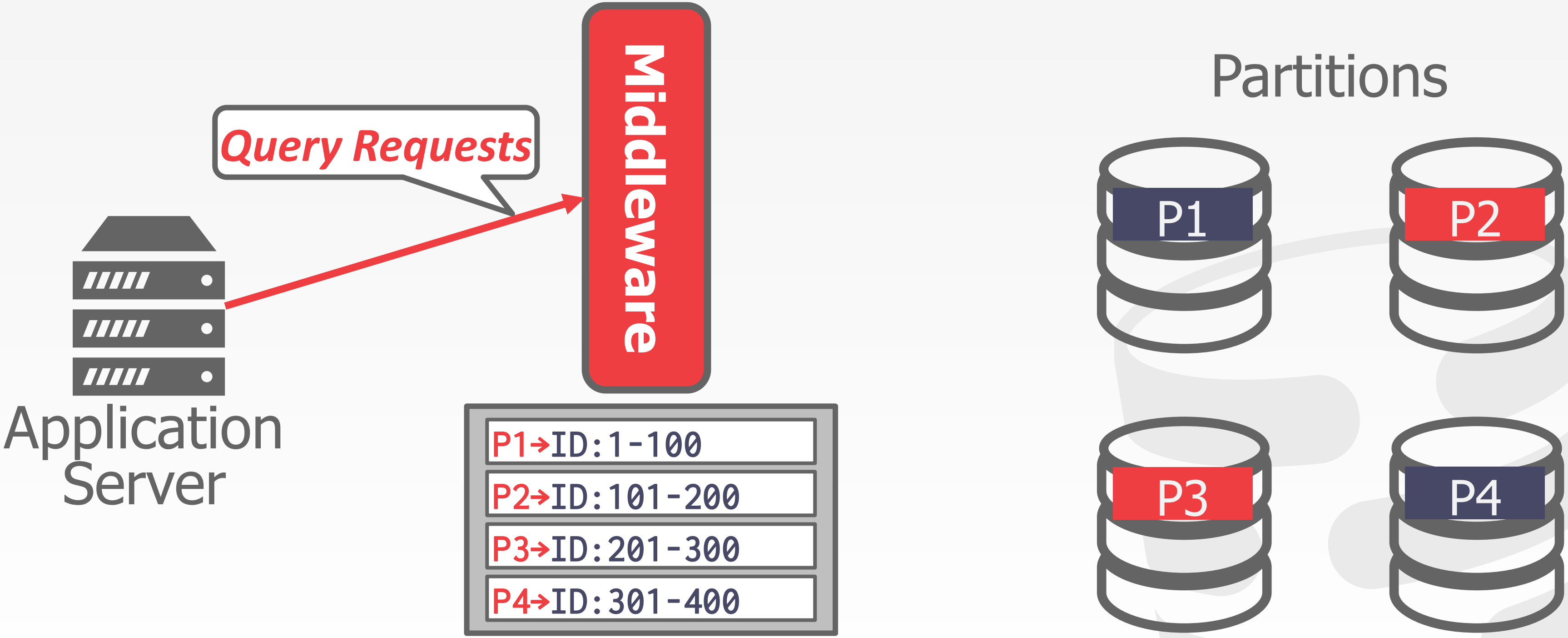
CENTRALIZED COORDINATOR



CENTRALIZED COORDINATOR

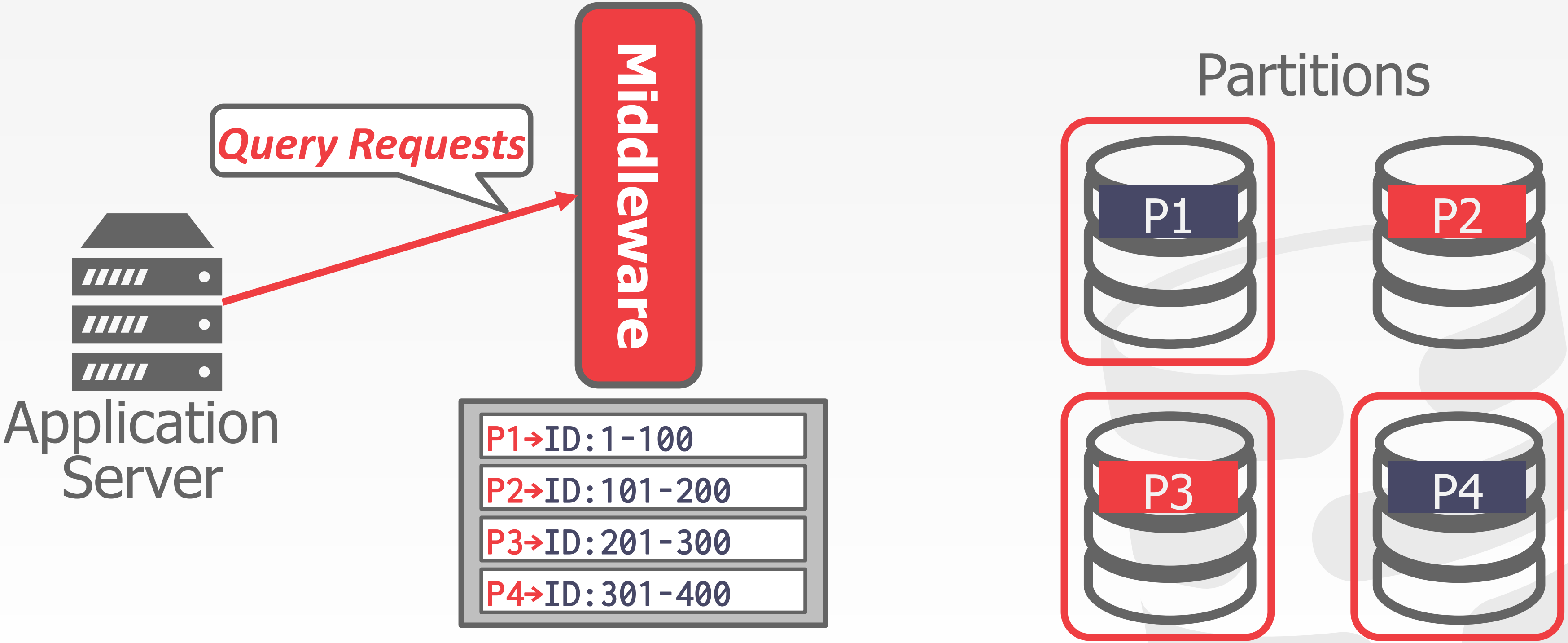


CENTRALIZED COORDINATOR



P1→ID:1-100
P2→ID:101-200
P3→ID:201-300
P4→ID:301-400

CENTRALIZED COORDINATOR



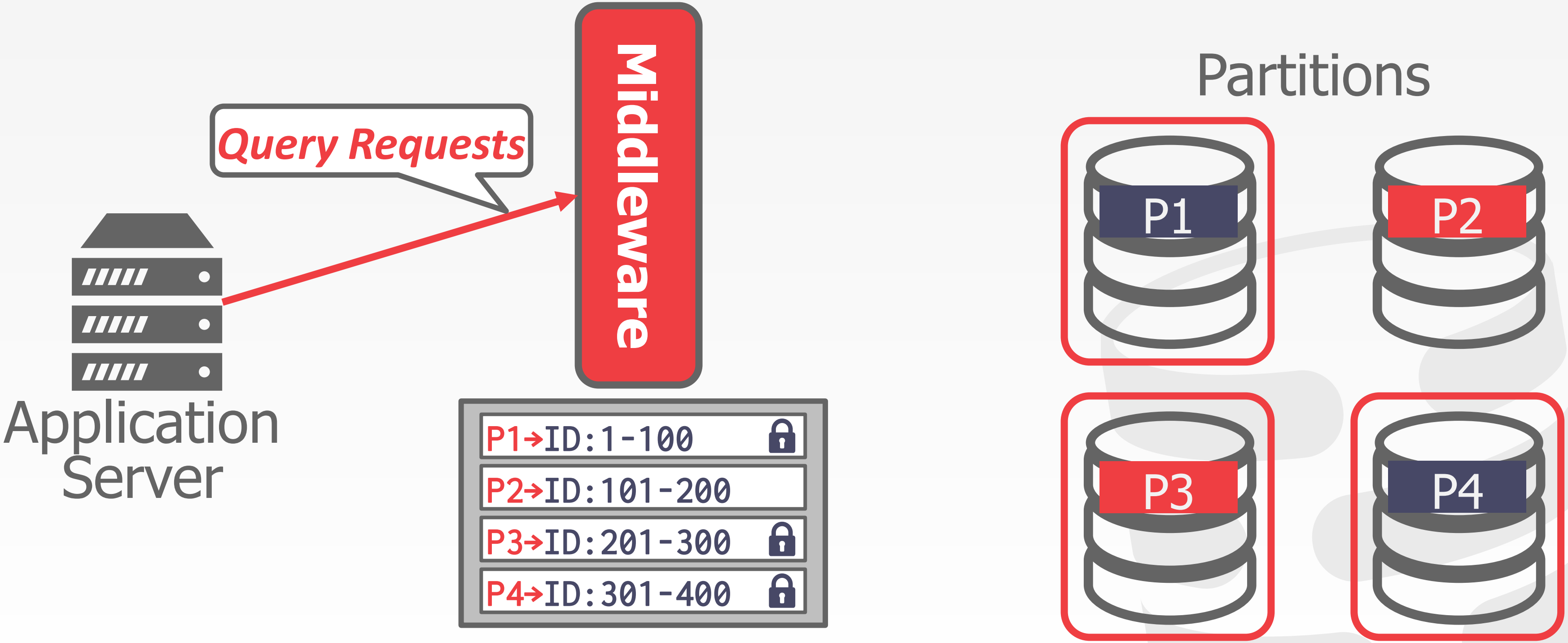
P1→ID:1-100

P2→ID:101-200

P3→ID:201-300

P4→ID:301-400

CENTRALIZED COORDINATOR



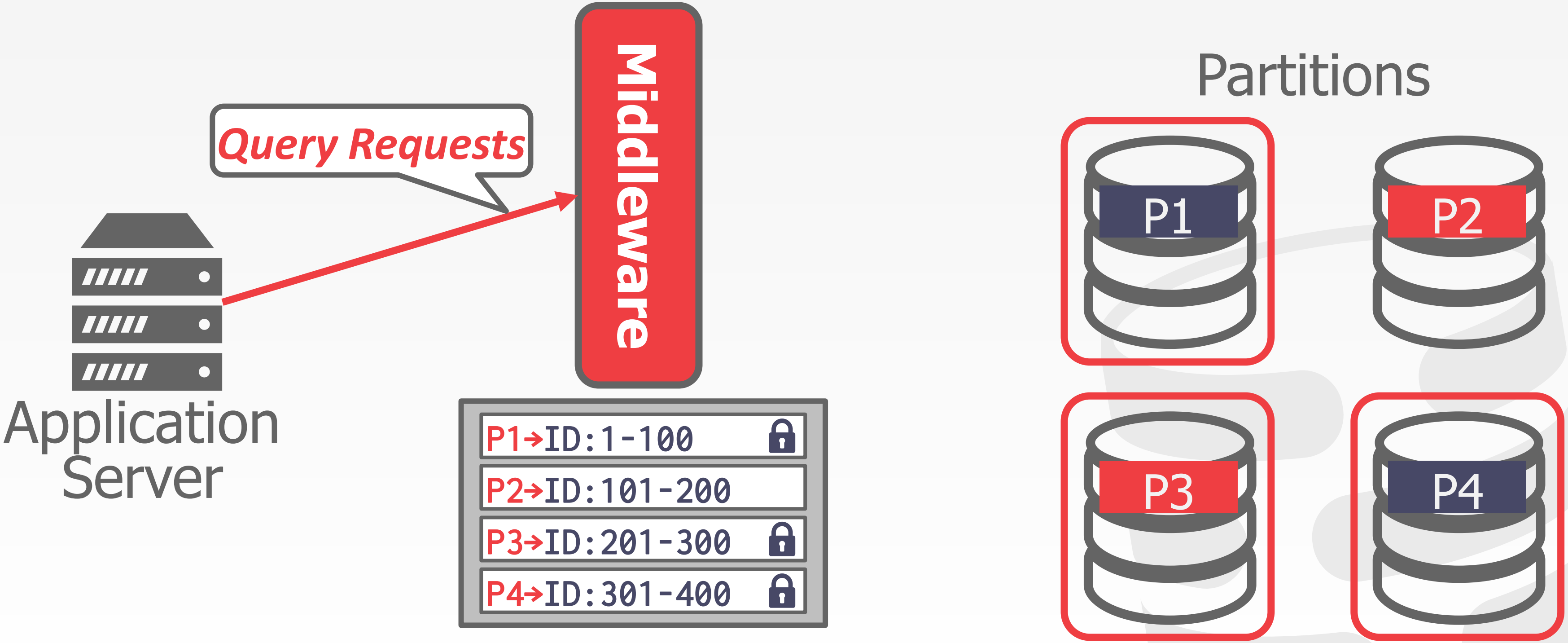
P1→ID:1-100 🔒

P2→ID:101-200

P3→ID:201-300 🔒

P4→ID:301-400 🔒

CENTRALIZED COORDINATOR



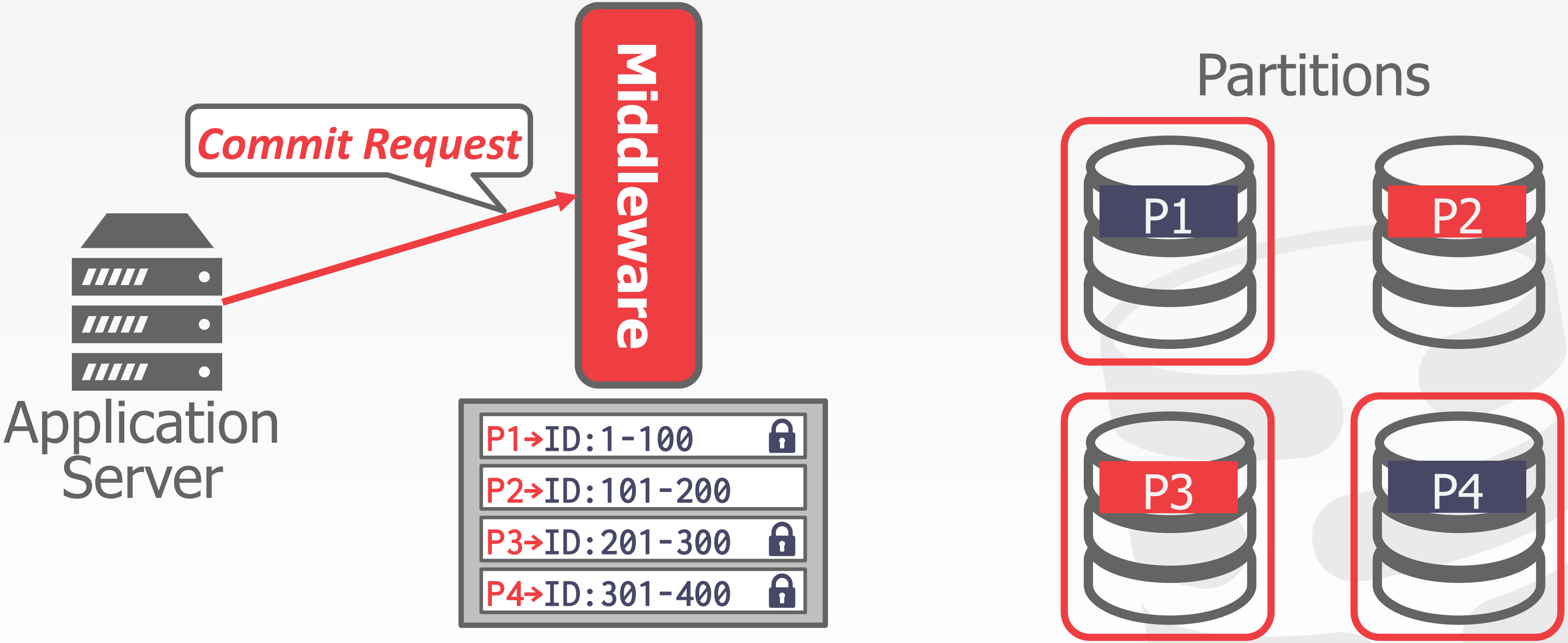
P1→ID:1-100 🔒

P2→ID:101-200

P3→ID:201-300 🔒

P4→ID:301-400 🔒

CENTRALIZED COORDINATOR



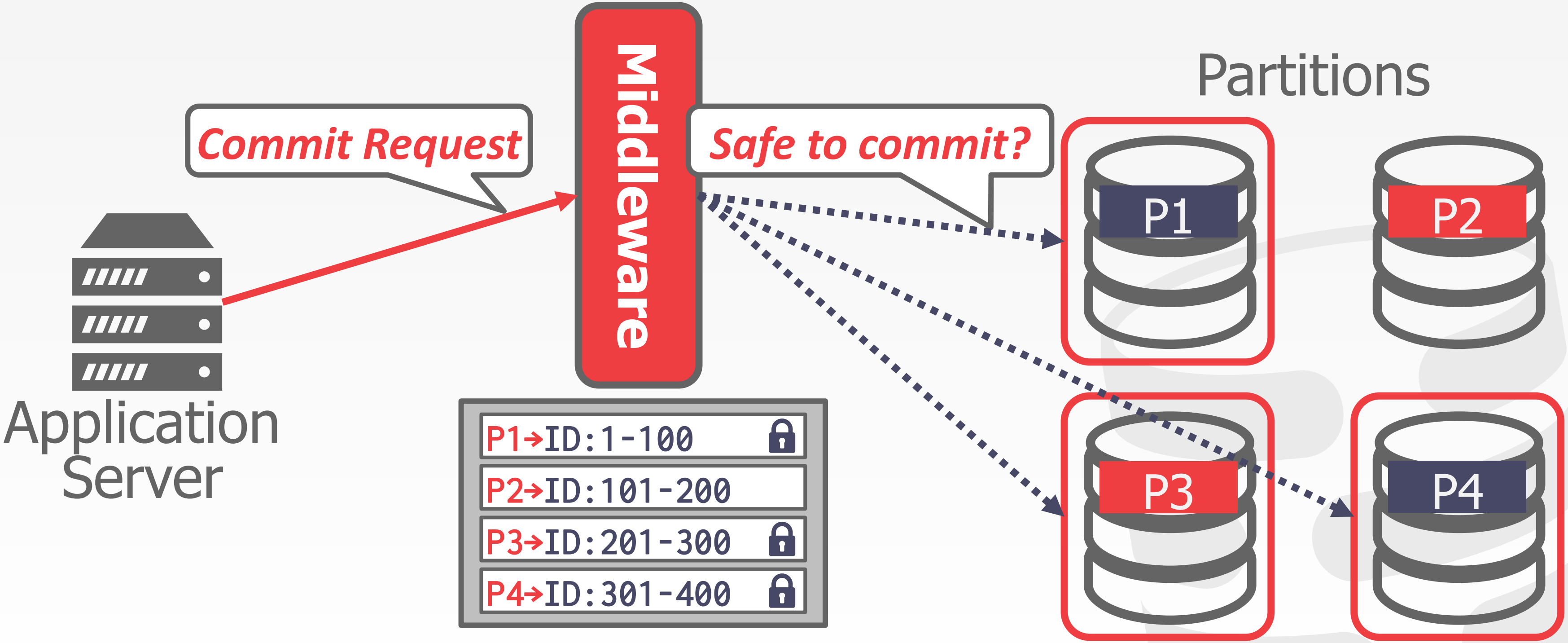
P1→ID:1-100 🔒

P2→ID:101-200

P3→ID:201-300 🔒

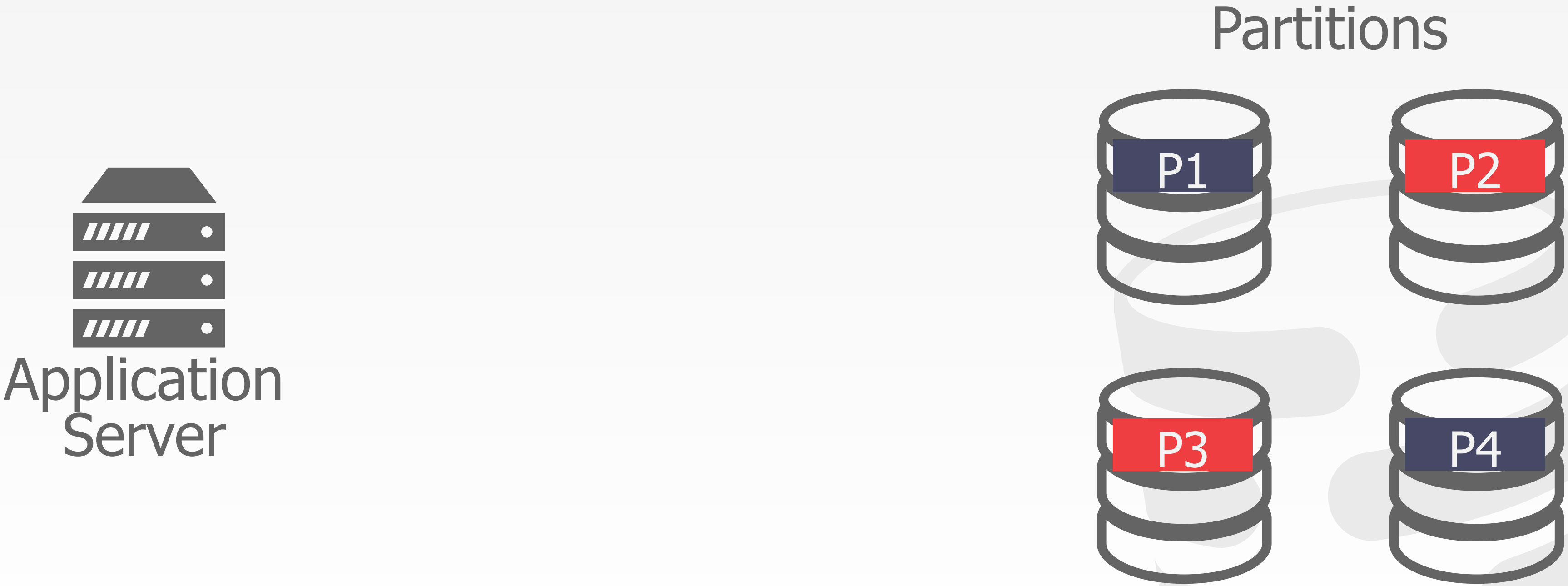
P4→ID:301-400 🔒

CENTRALIZED COORDINATOR

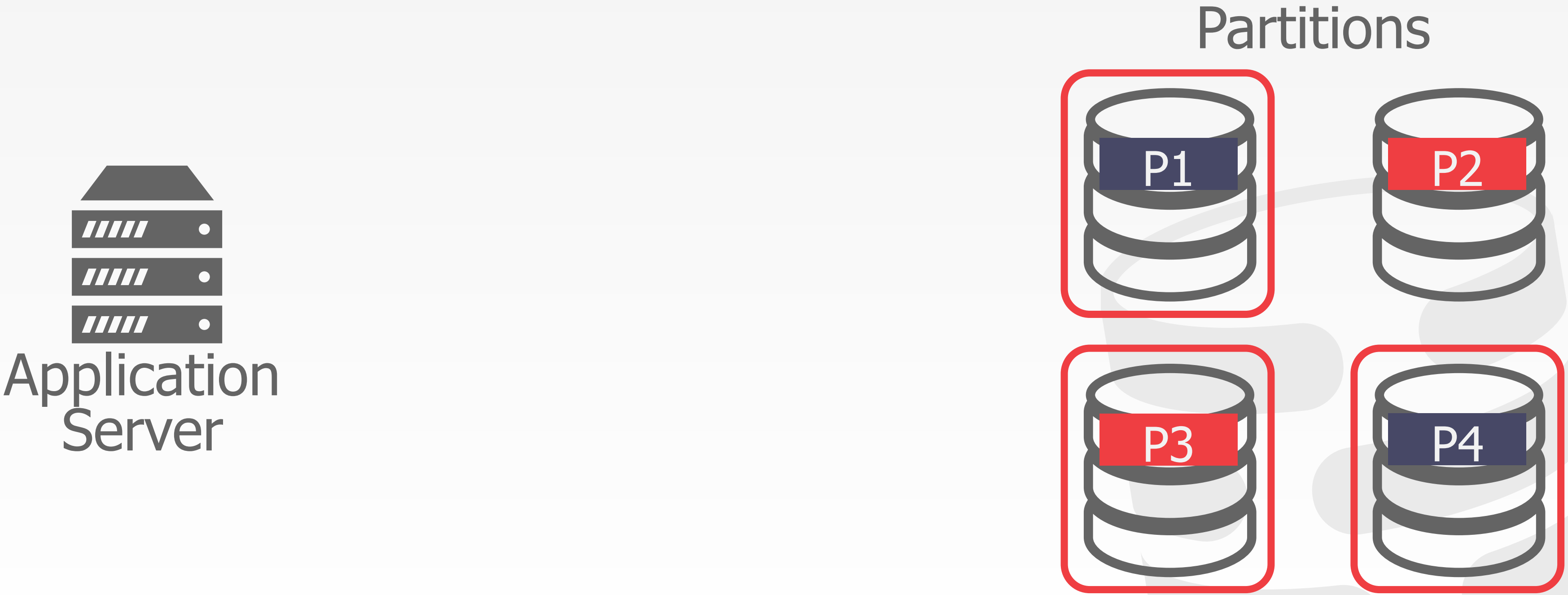


P1→ID:1-100 🔒
P2→ID:101-200
P3→ID:201-300 🔒
P4→ID:301-400 🔒

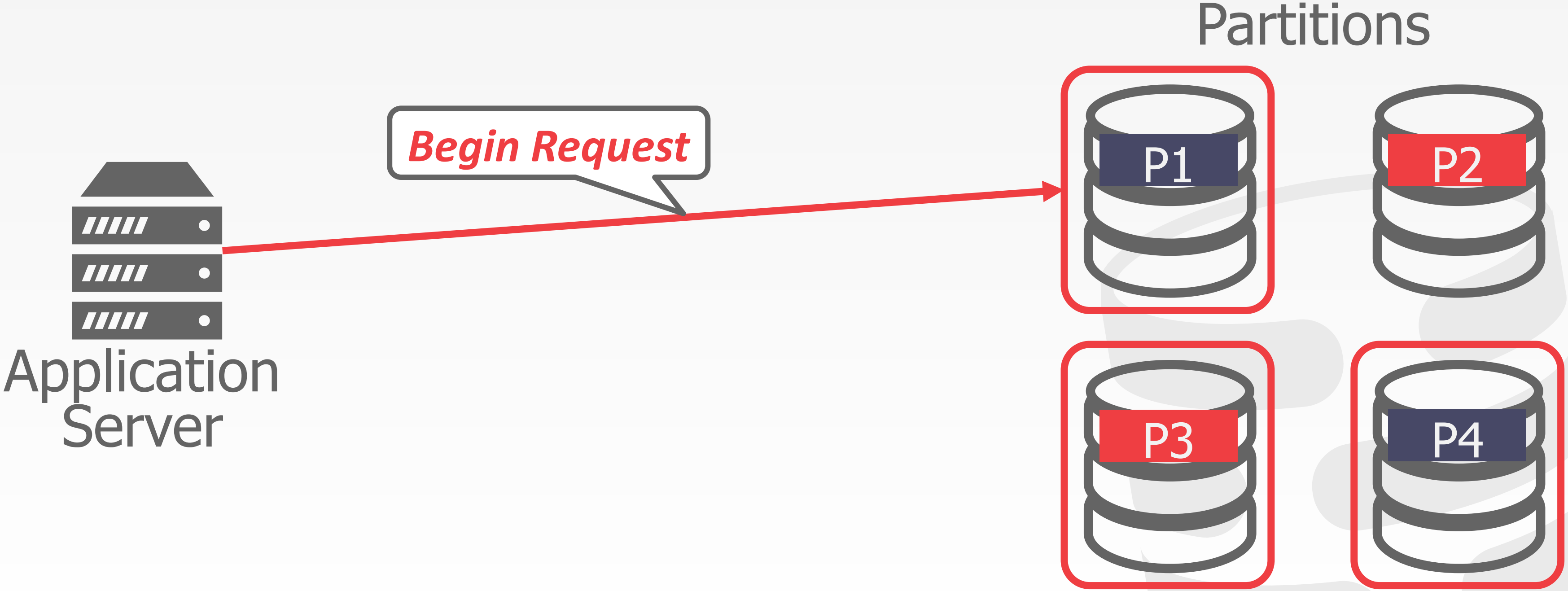
DECENTRALIZED COORDINATOR



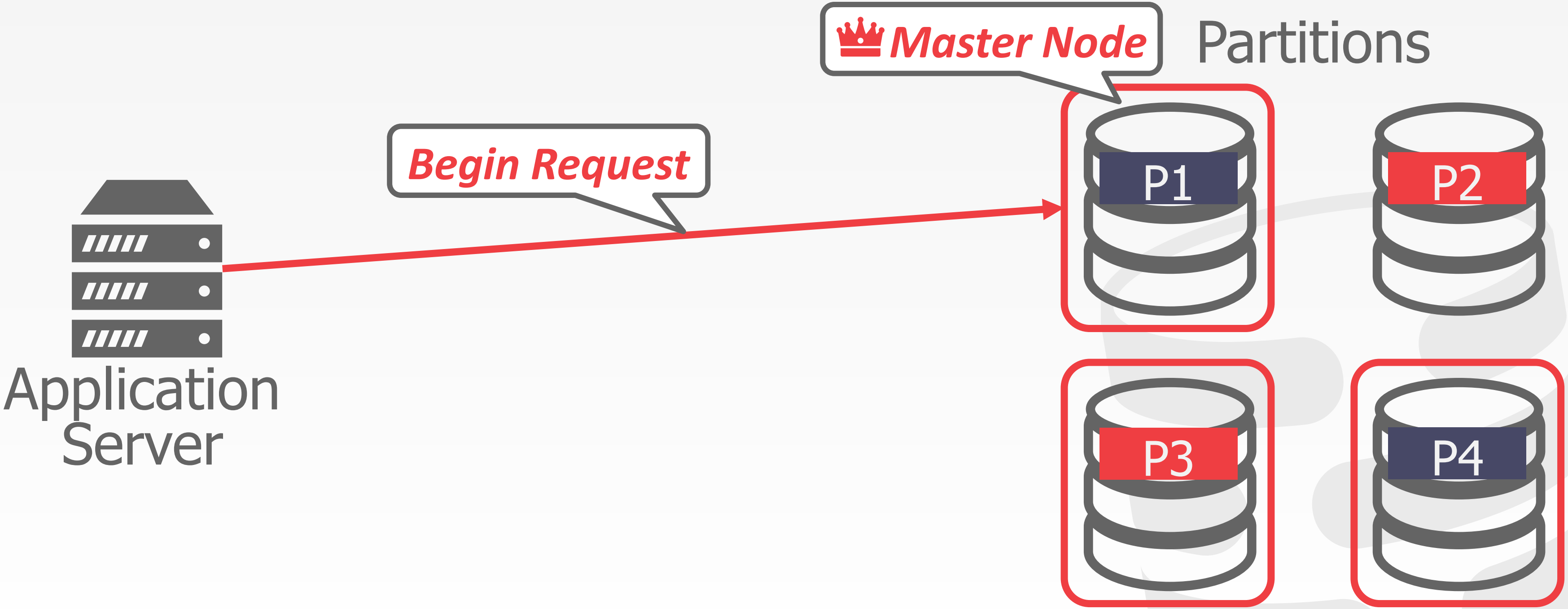
DECENTRALIZED COORDINATOR



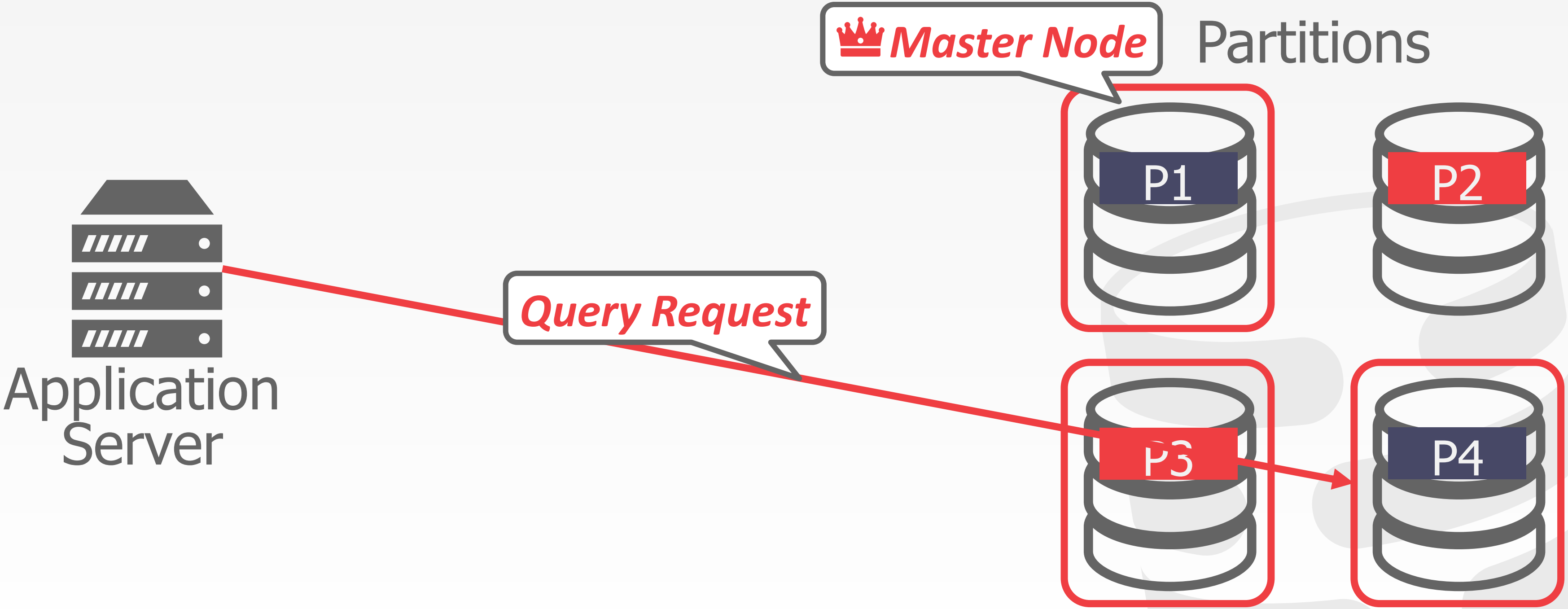
DECENTRALIZED COORDINATOR



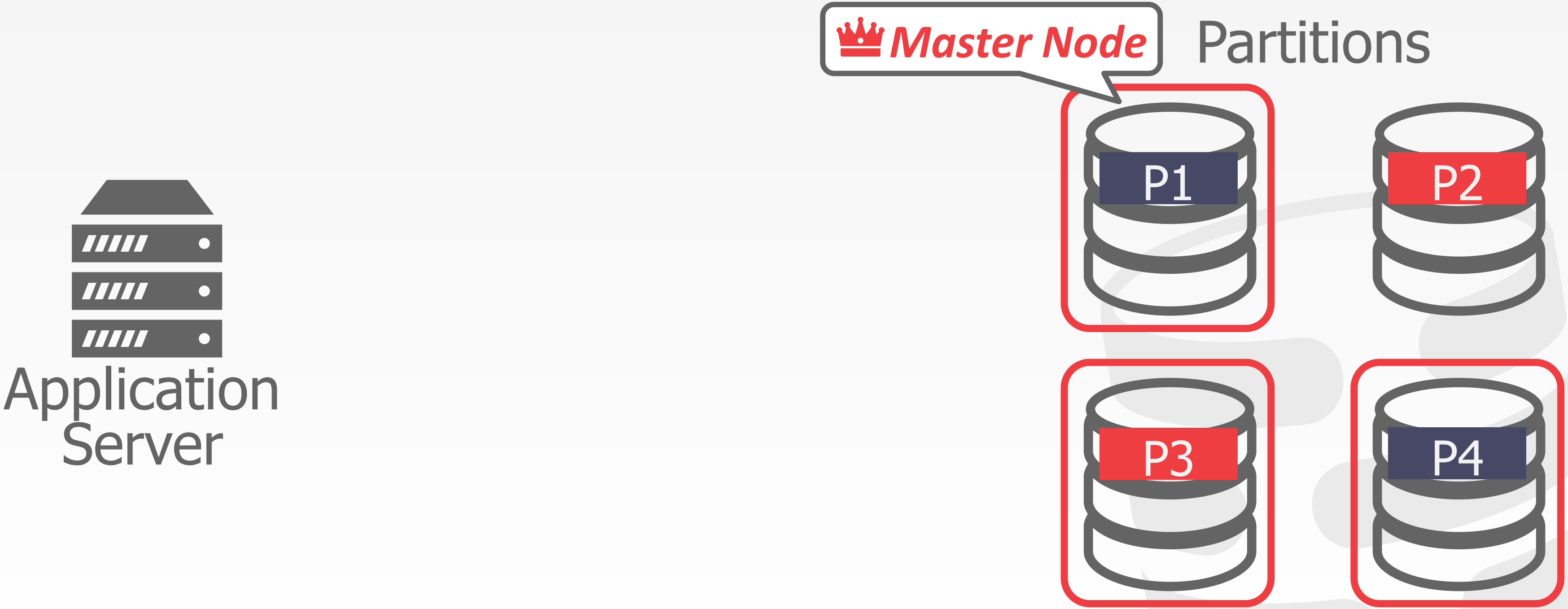
DECENTRALIZED COORDINATOR



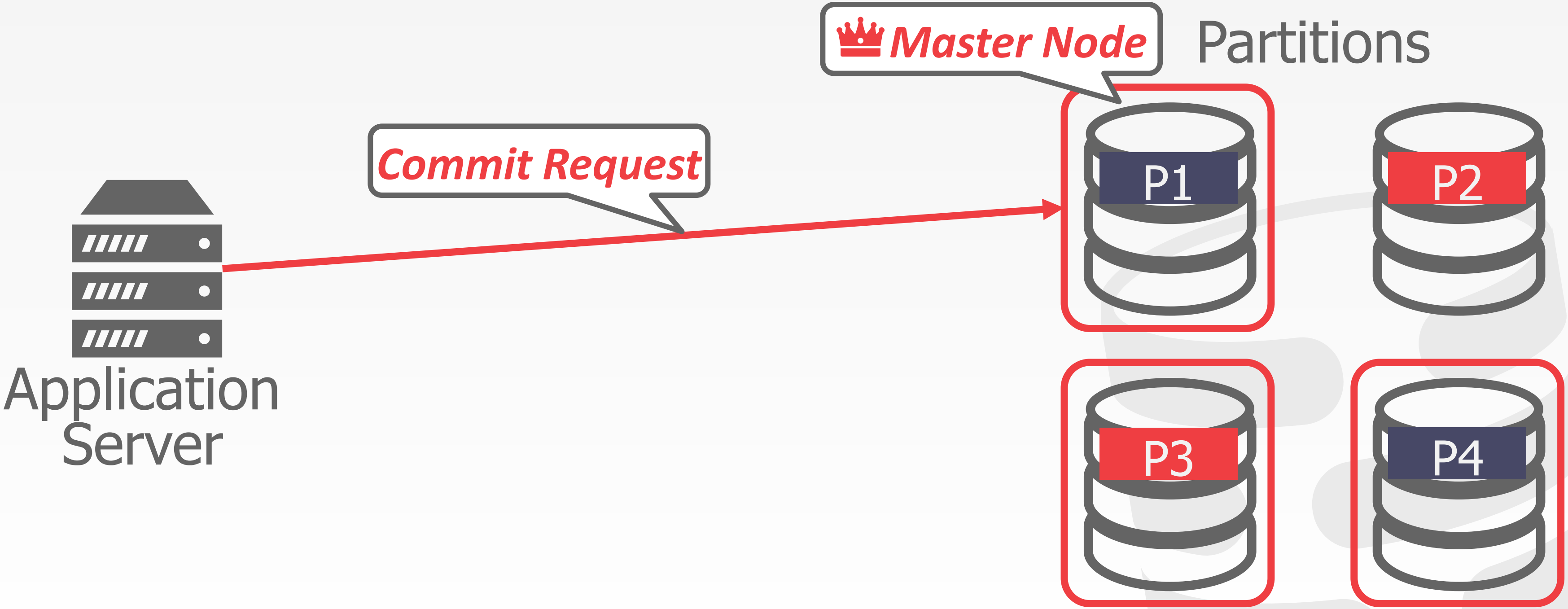
DECENTRALIZED COORDINATOR



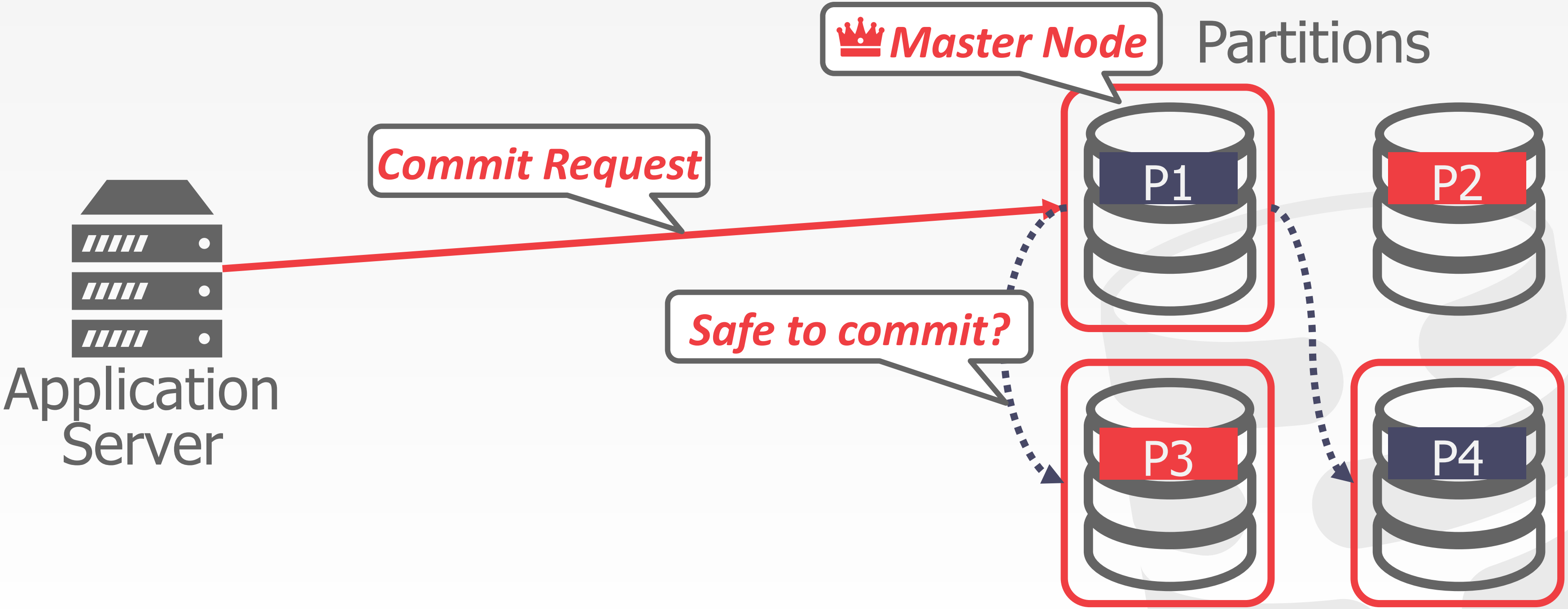
DECENTRALIZED COORDINATOR



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:

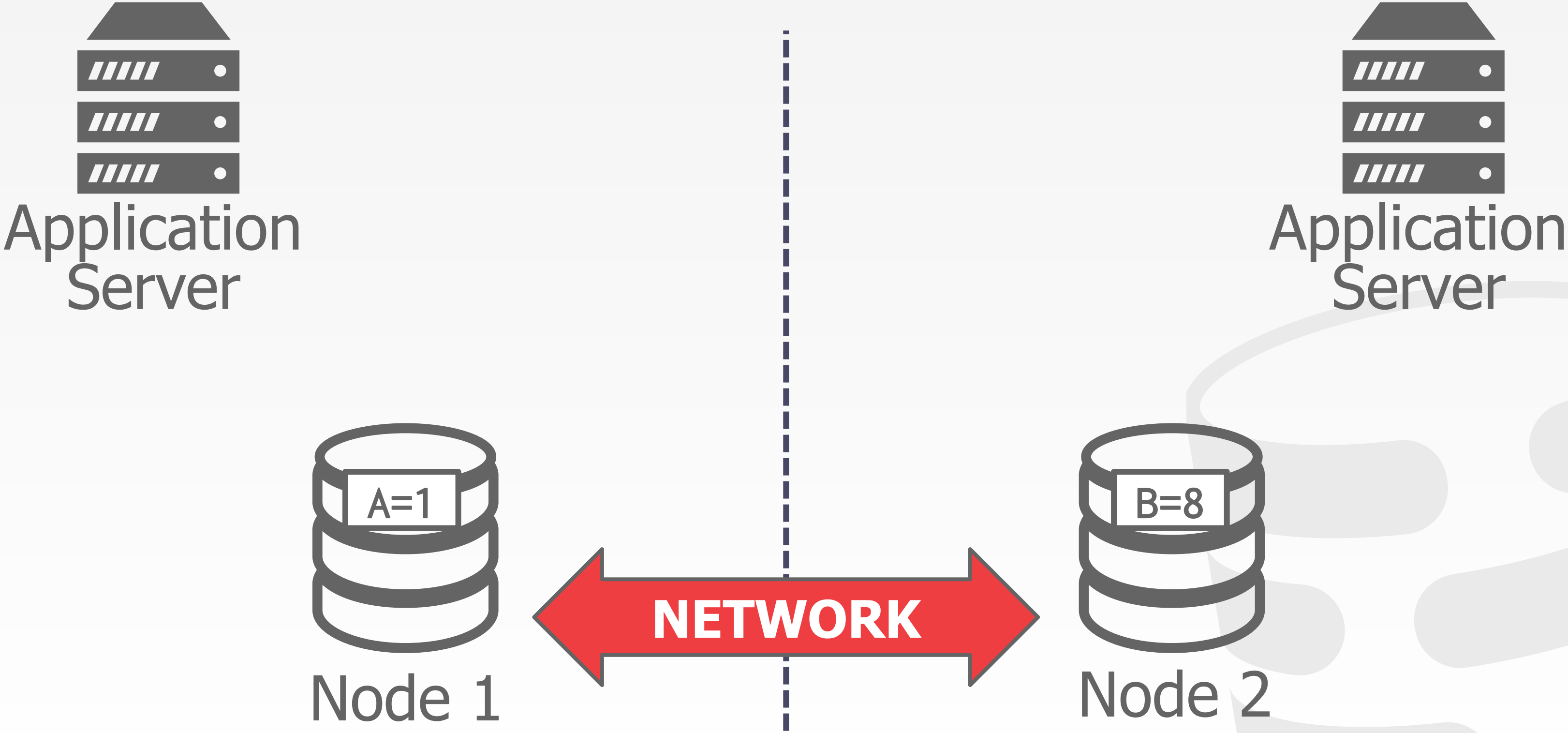
→ Replication.

→ Network Communication Overhead.

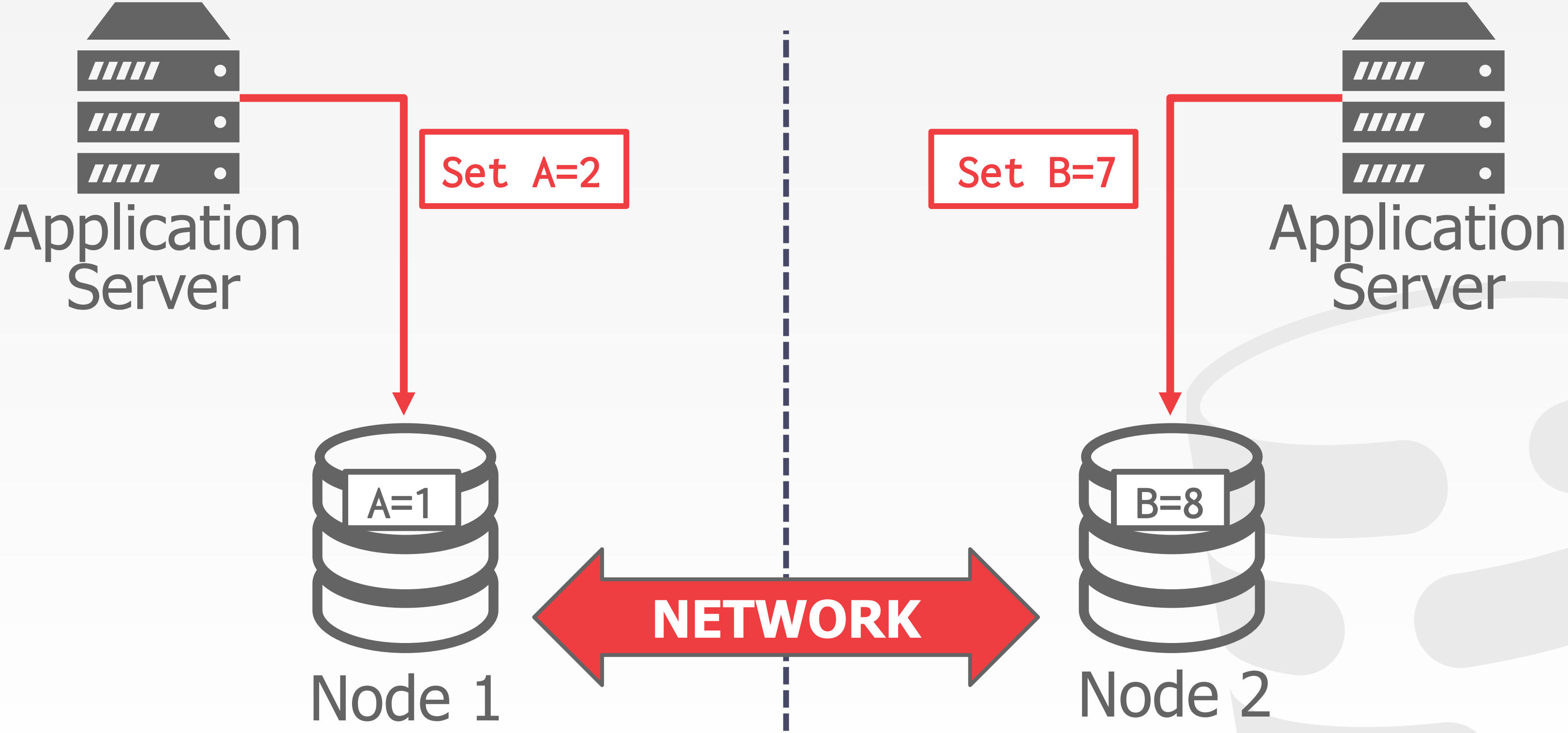
→ Node Failures.

→ Clock Skew.

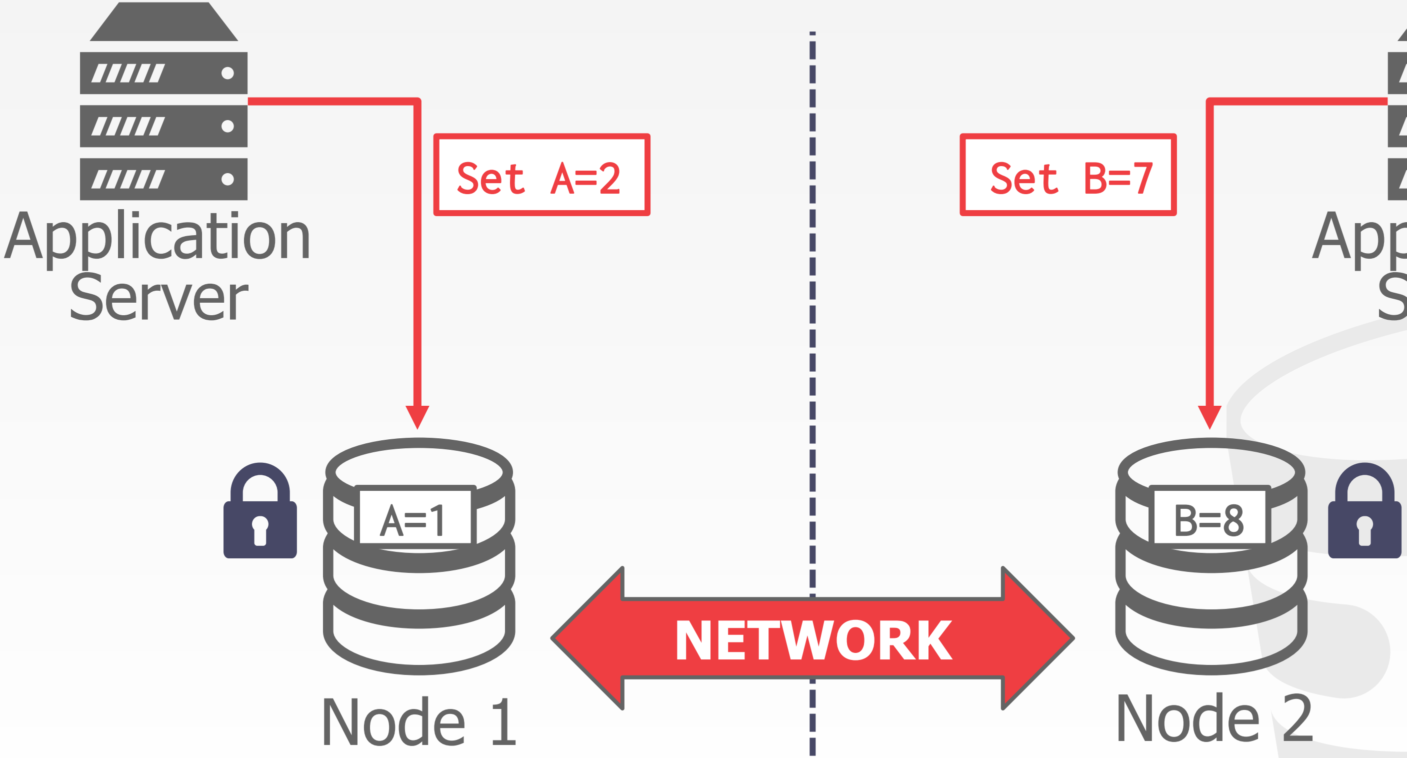
DISTRIBUTED 2PL



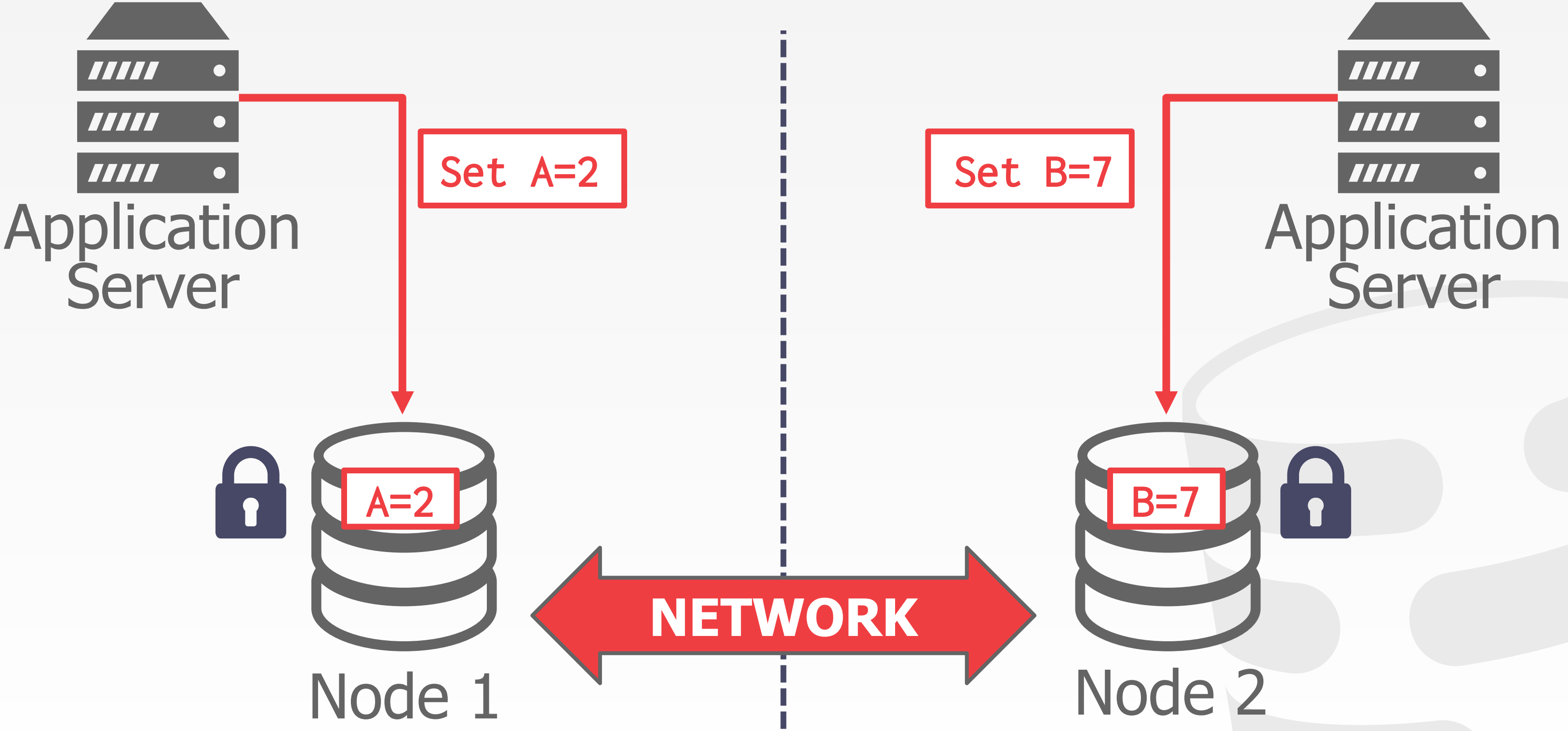
DISTRIBUTED 2PL



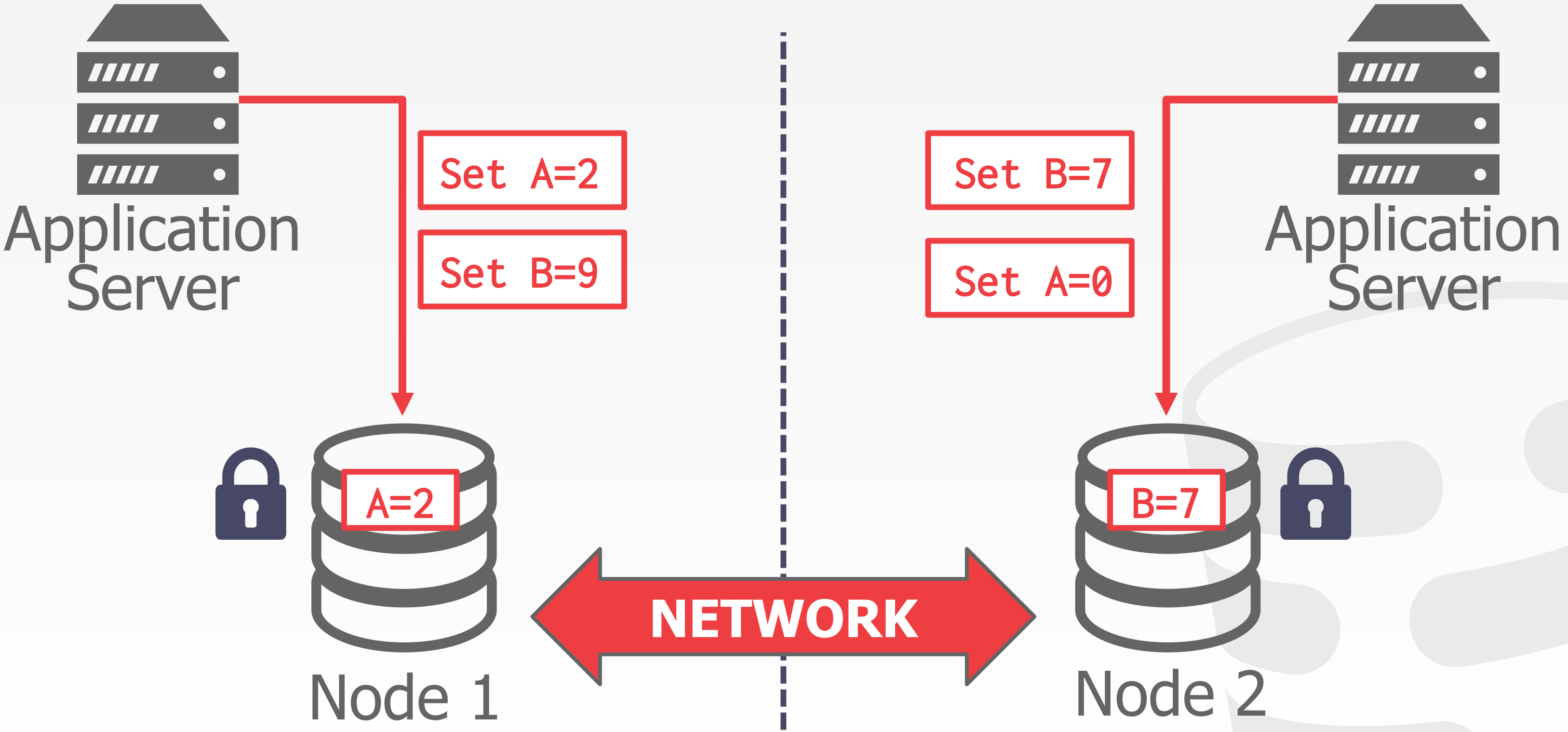
DISTRIBUTED 2PL



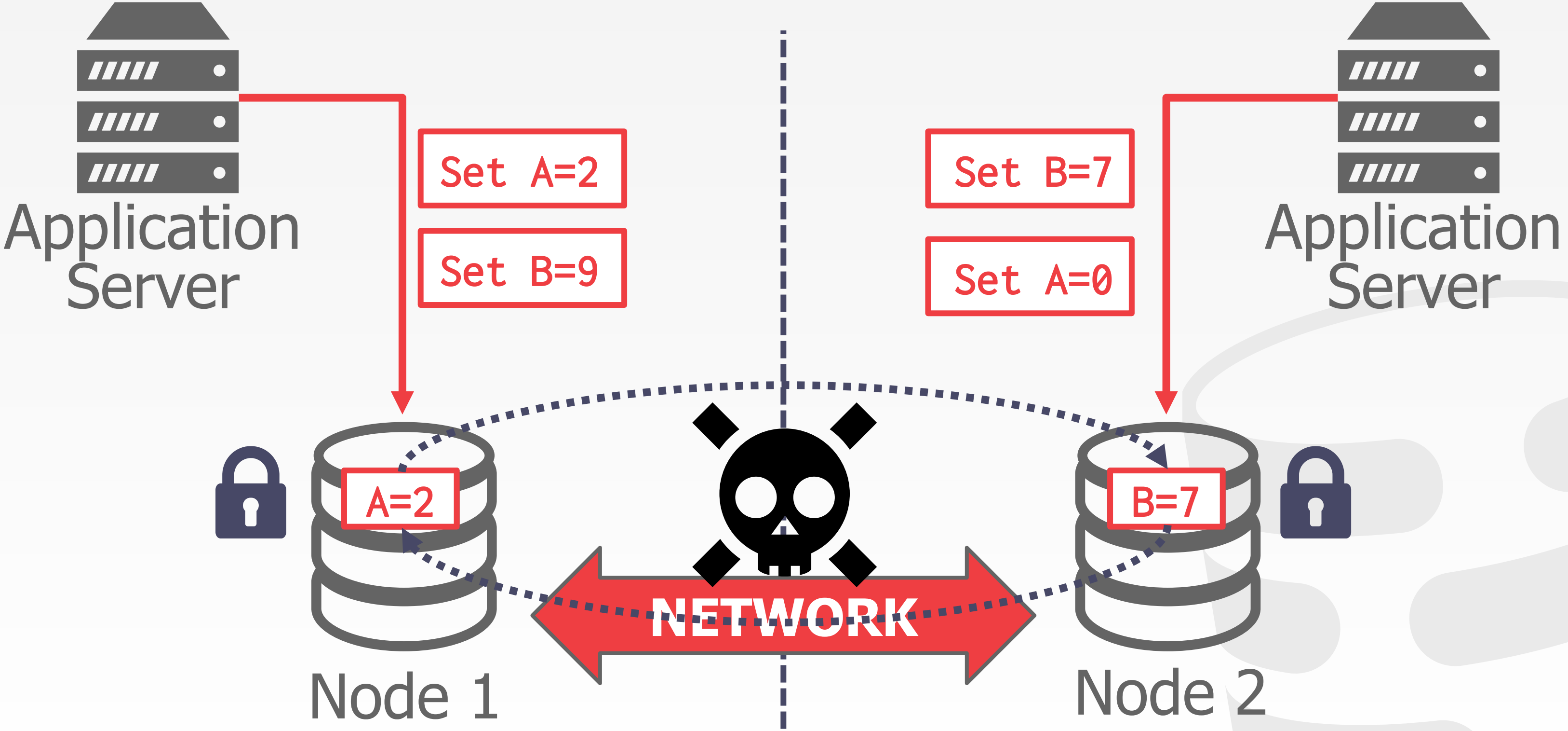
DISTRIBUTED 2PL



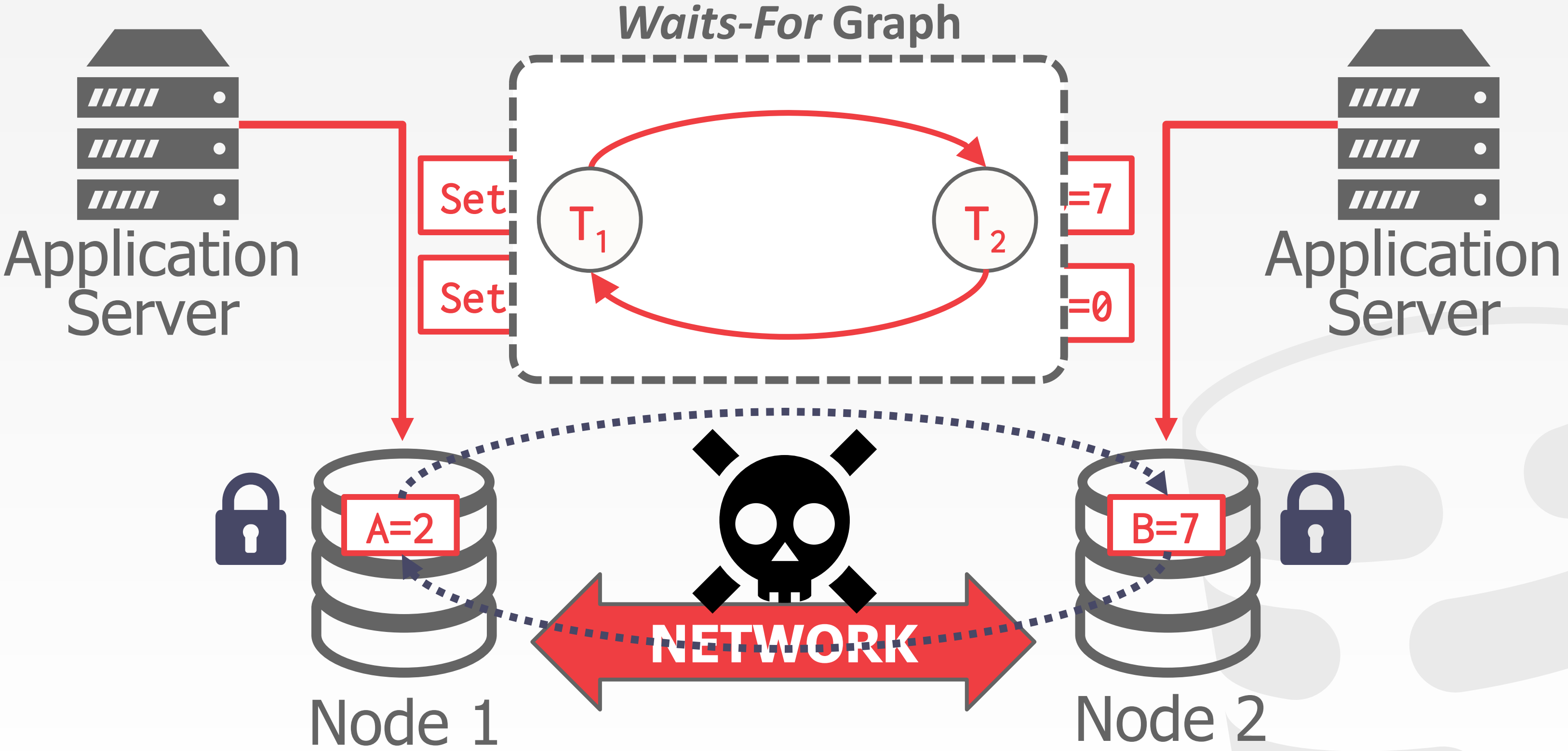
DISTRIBUTED 2PL



DISTRIBUTED 2PL



DISTRIBUTED 2PL



CONCLUSION

I have barely scratched the surface on distributed database systems…

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

NEXT CLASS

Distributed OLTP Systems

Replication

CAP Theorem

Real-World Examples