Intro to Database Systems (15-445/645)

21 Introduction to Distributed Databases

Carnegie Mellon University

FALL 2022

Andy Pavlo

ADMINISTRIVIA

Project #4 is due **Sun Dec 11th @ 11:59pm**
→ Zoom Q&A Session **TONIGHT @ 8:00pm**

Homework #5 is due **Sun Dec 4th @ 11:59pm**

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

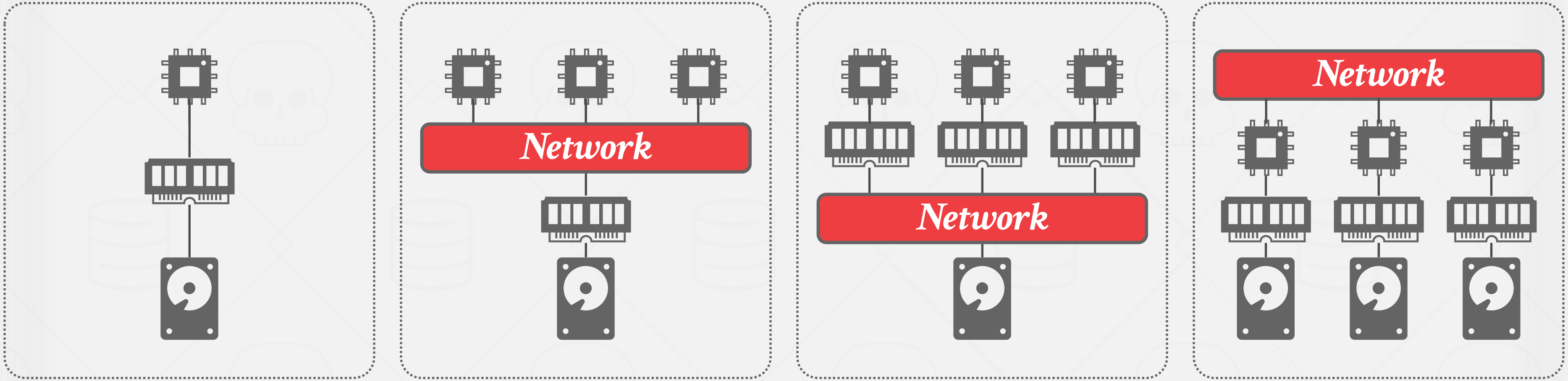
System Architectures
Design Issues
Partitioning Schemes
Distributed Concurrency Control

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



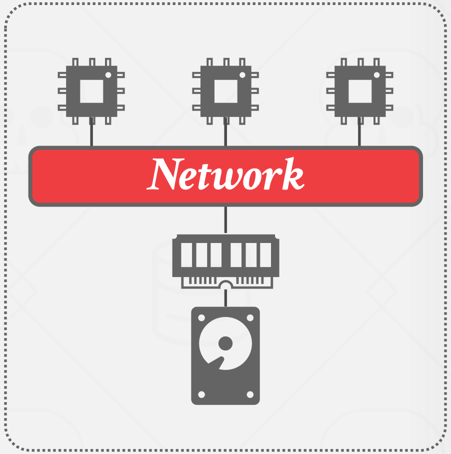
Shared Everything | Shared Memory | Shared Disk | Shared Nothing

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 must "know" about the other instances.

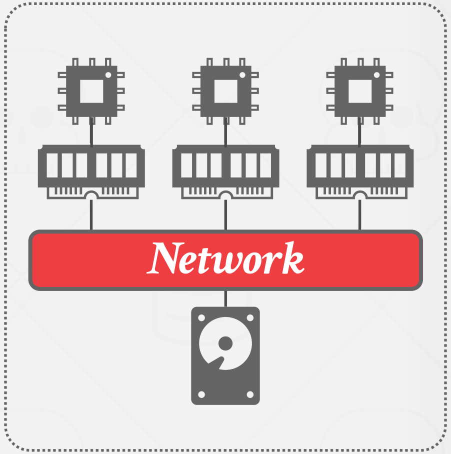


SHARED DISK

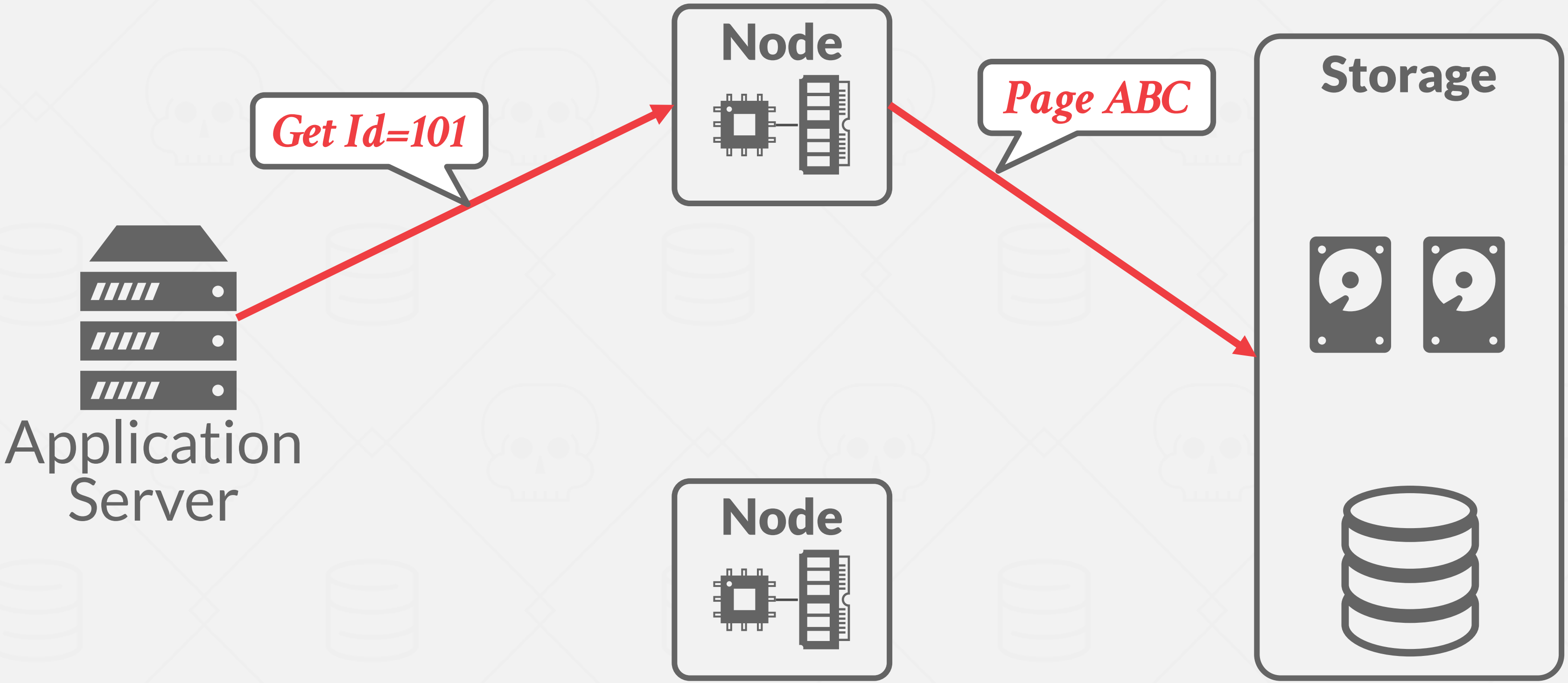
All CPUs can access a single logical disk directly via an interconnect, but each have their own private memories.

→ Can scale execution layer independently from the storage layer.

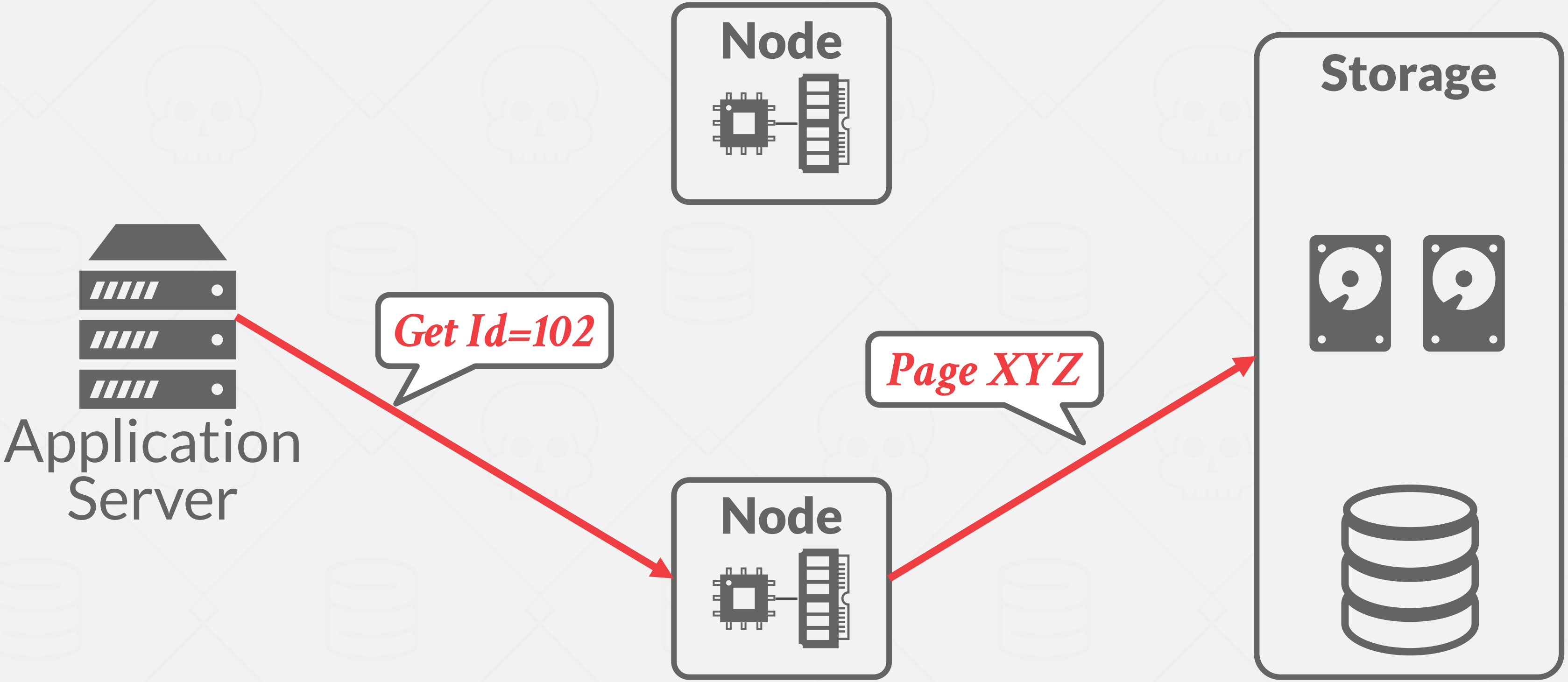
→ Must send messages between CPUs to learn about their current state.



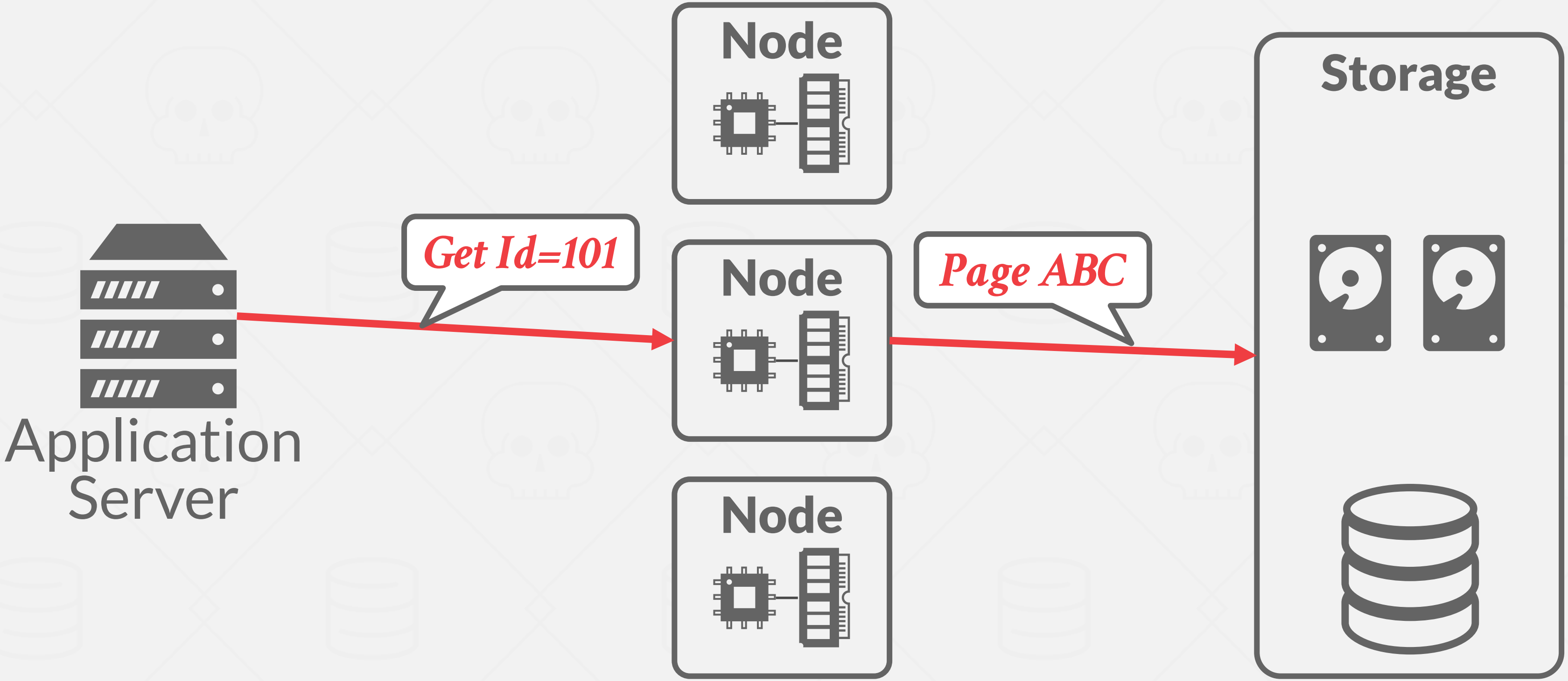
SHARED DISK EXAMPLE



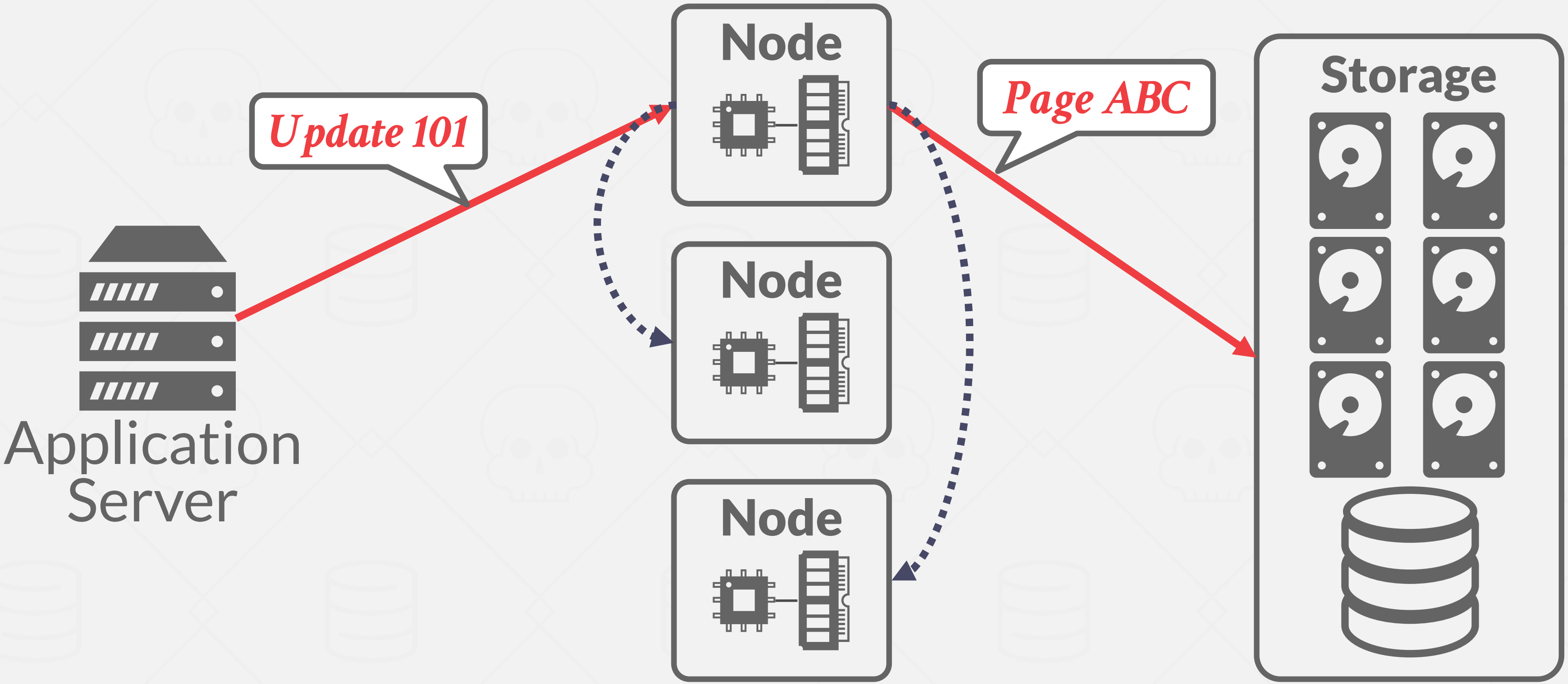
SHARED DISK EXAMPLE



SHARED DISK EXAMPLE



SHARED DISK EXAMPLE



SHARED NOTHING

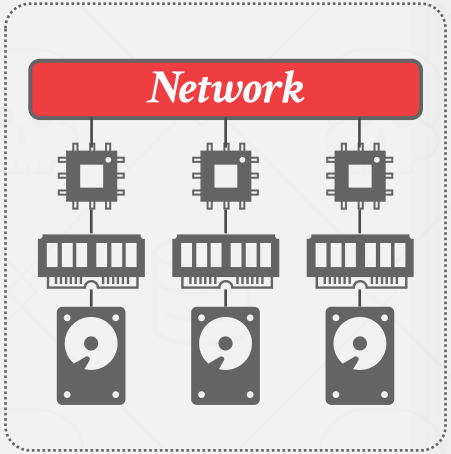
Each DBMS instance has its own CPU, memory, and local disk.

Nodes only communicate with each other via network.

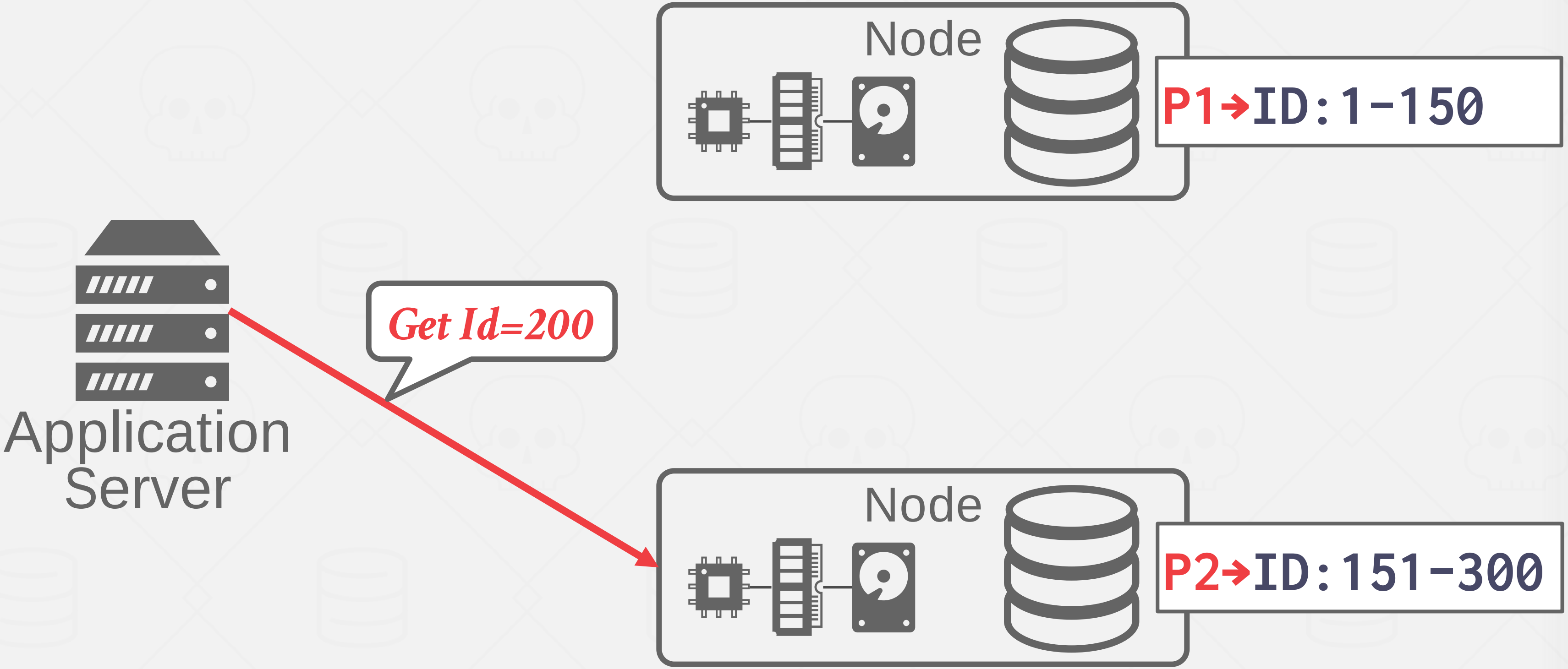
→ Harder to scale capacity.

→ Harder to ensure consistency.

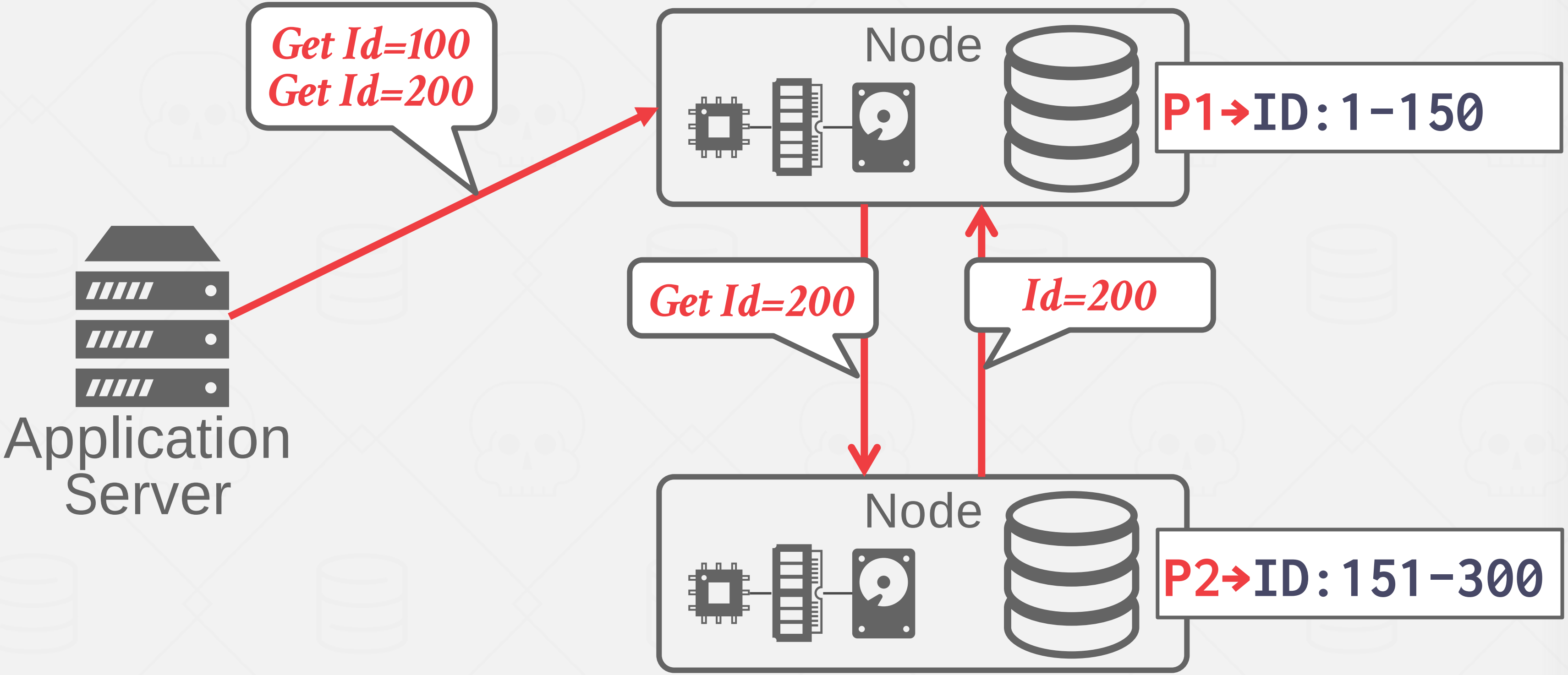
→ Better performance & efficiency.



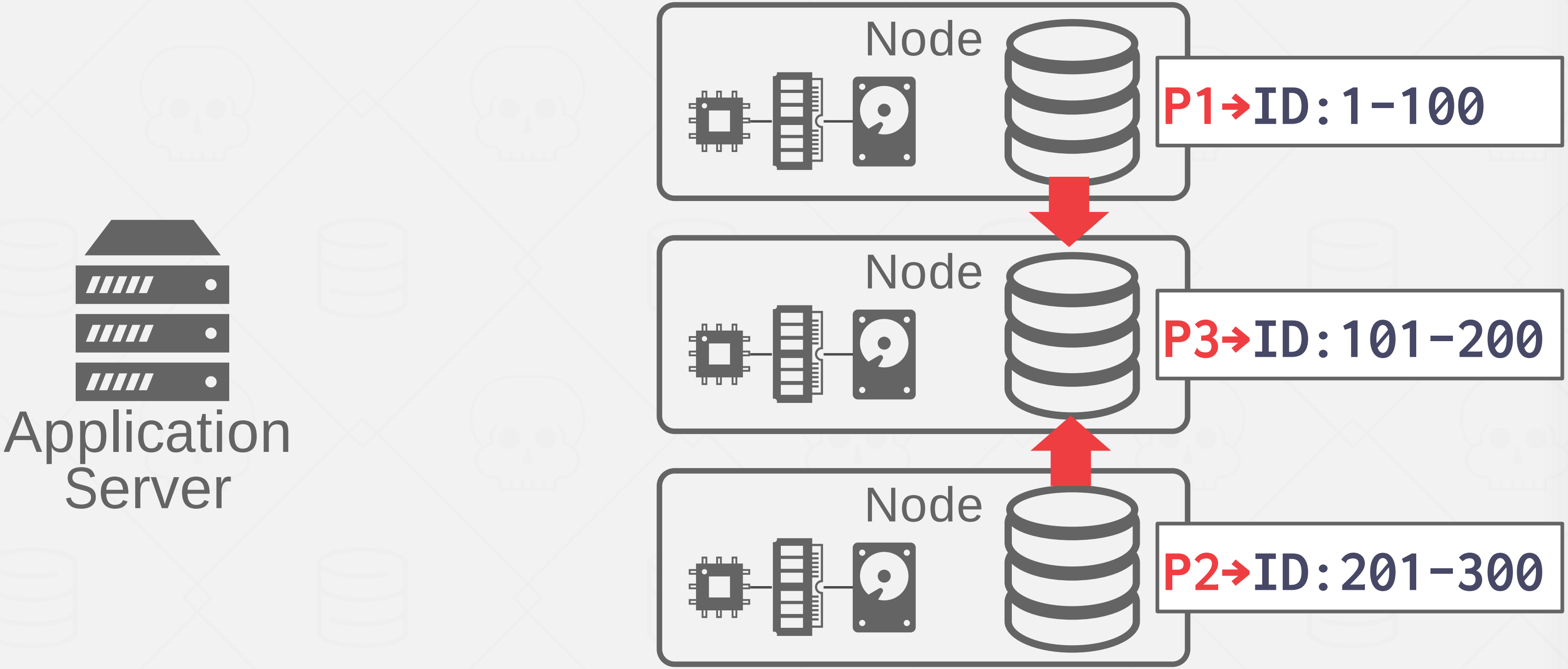
SHARED NOTHING EXAMPLE



SHARED NOTHING EXAMPLE

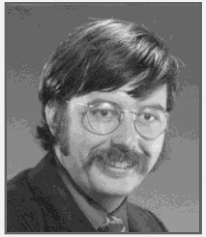


SHARED NOTHING EXAMPLE

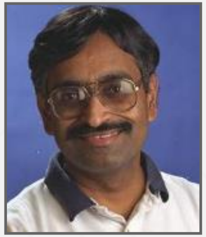


EARLY DISTRIBUTED DATABASE SYSTEMS

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?

Where does the application send queries?

How to execute queries on distributed data?

→ Push query to data.
→ Pull data to query.

How does the DBMS ensure correctness? ***Next Class***

How do we divide the database across resources?

HOMOGENOUS VS. HETEROGENOUS

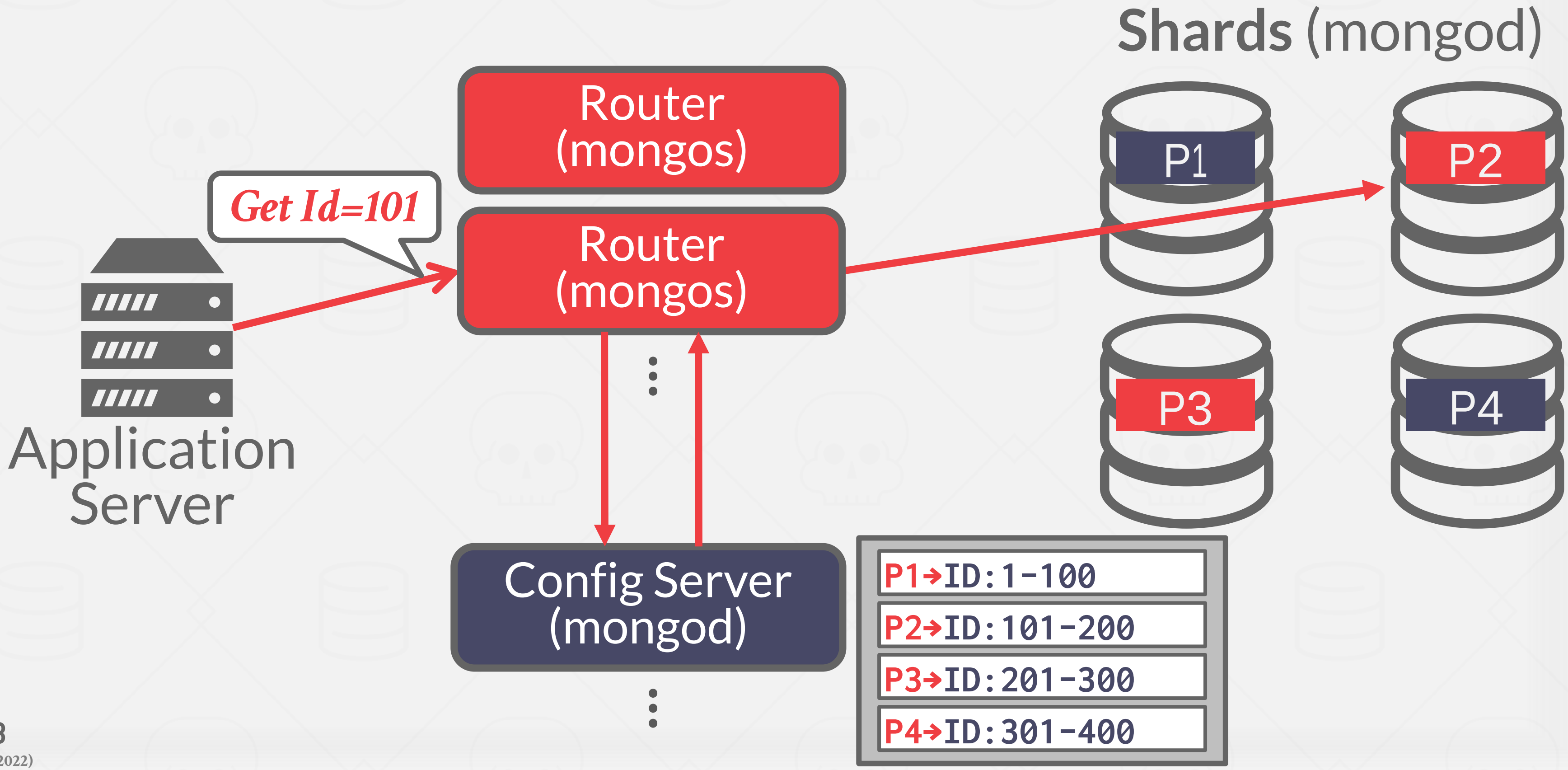
Approach #1: Homogenous Nodes

→ 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



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).

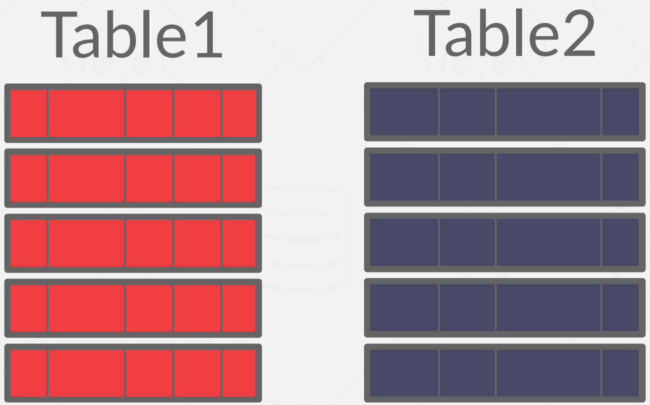
NAÏVE TABLE PARTITIONING

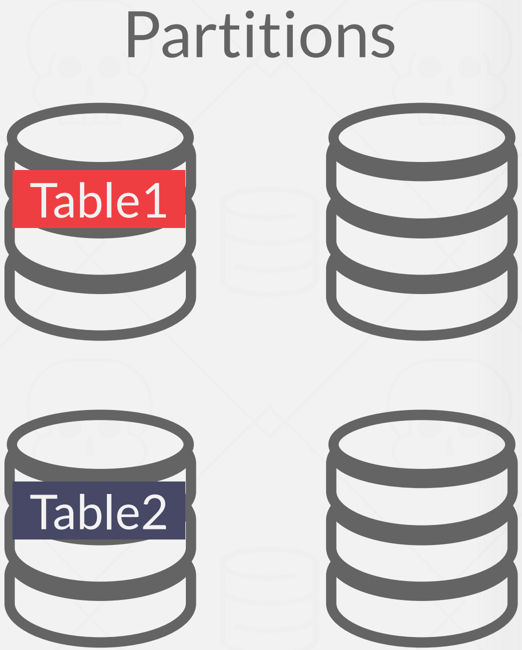
Assign an entire table to a single node.

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
```

VERTICAL PARTITIONING

Split a table's attributes into separate partitions.

Must store tuple information to reconstruct the original record.

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

Tuple#1	attr1	attr2	attr3	attr4
Tuple#2	attr1	attr2	attr3	attr4
Tuple#3	attr1	attr2	attr3	attr4
Tuple#4	attr1	attr2	attr3	attr4

VERTICAL PARTITIONING

Split a table's attributes into separate partitions.

Must store tuple information to reconstruct the original record.

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

Partition #1

Tuple#1	attr1	attr2	attr3
Tuple#2	attr1	attr2	attr3
Tuple#3	attr1	attr2	attr3
Tuple#4	attr1	attr2	attr3

Partition #2

Tuple#1	attr4
Tuple#2	attr4
Tuple#3	attr4
Tuple#4	attr4

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

Partitioning Key

Table1

101	a	XXX	2022-11-29
102	b	XXY	2022-11-28
103	c	XYZ	2022-11-29
104	d	XYX	2022-11-27
105	e	XYY	2022-11-29

hash(a)%4 = P2

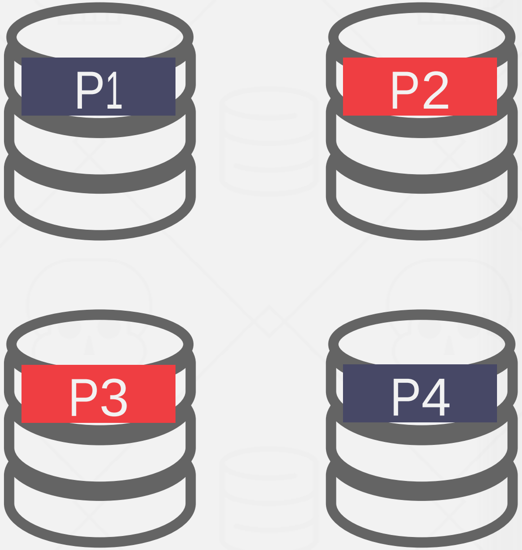
hash(b)%4 = P4

hash(c)%4 = P3

hash(d)%4 = P2

hash(e)%4 = P1

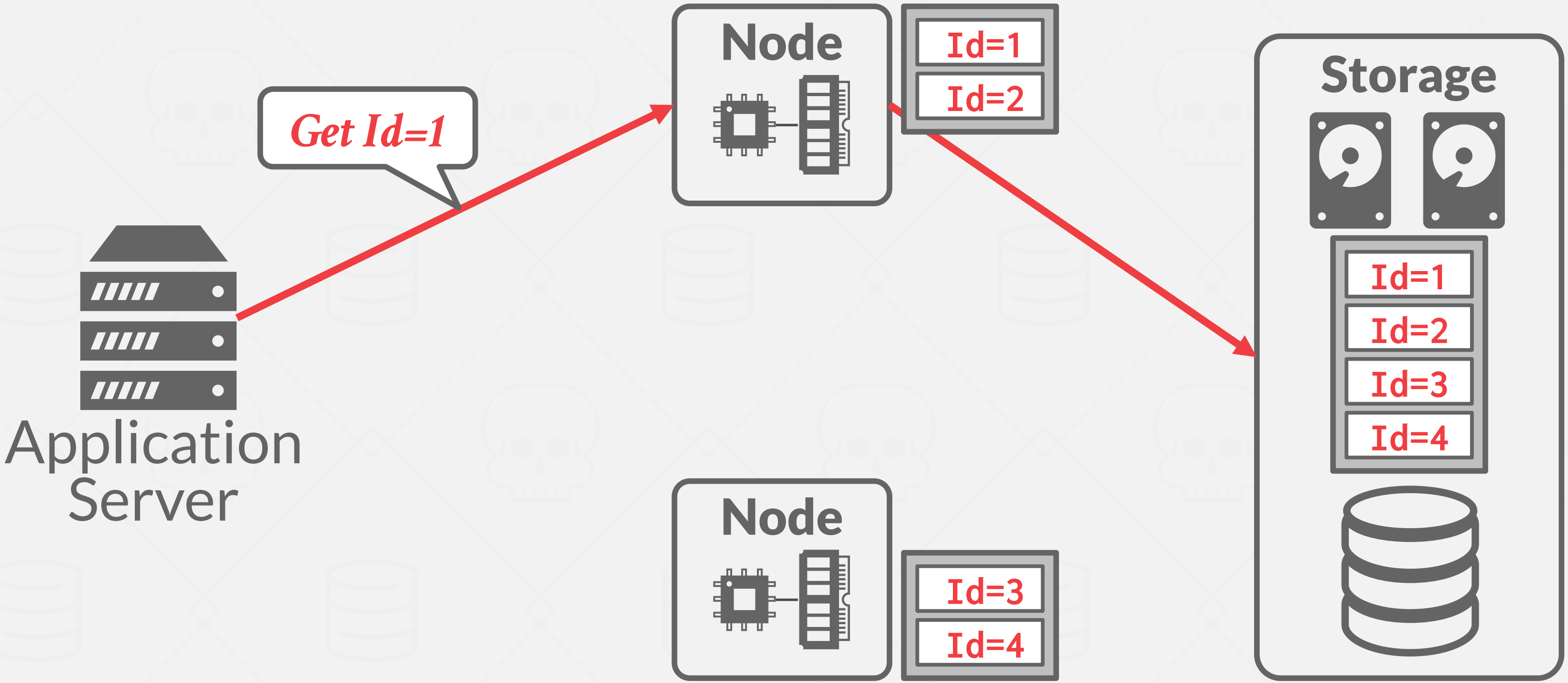
Partitions



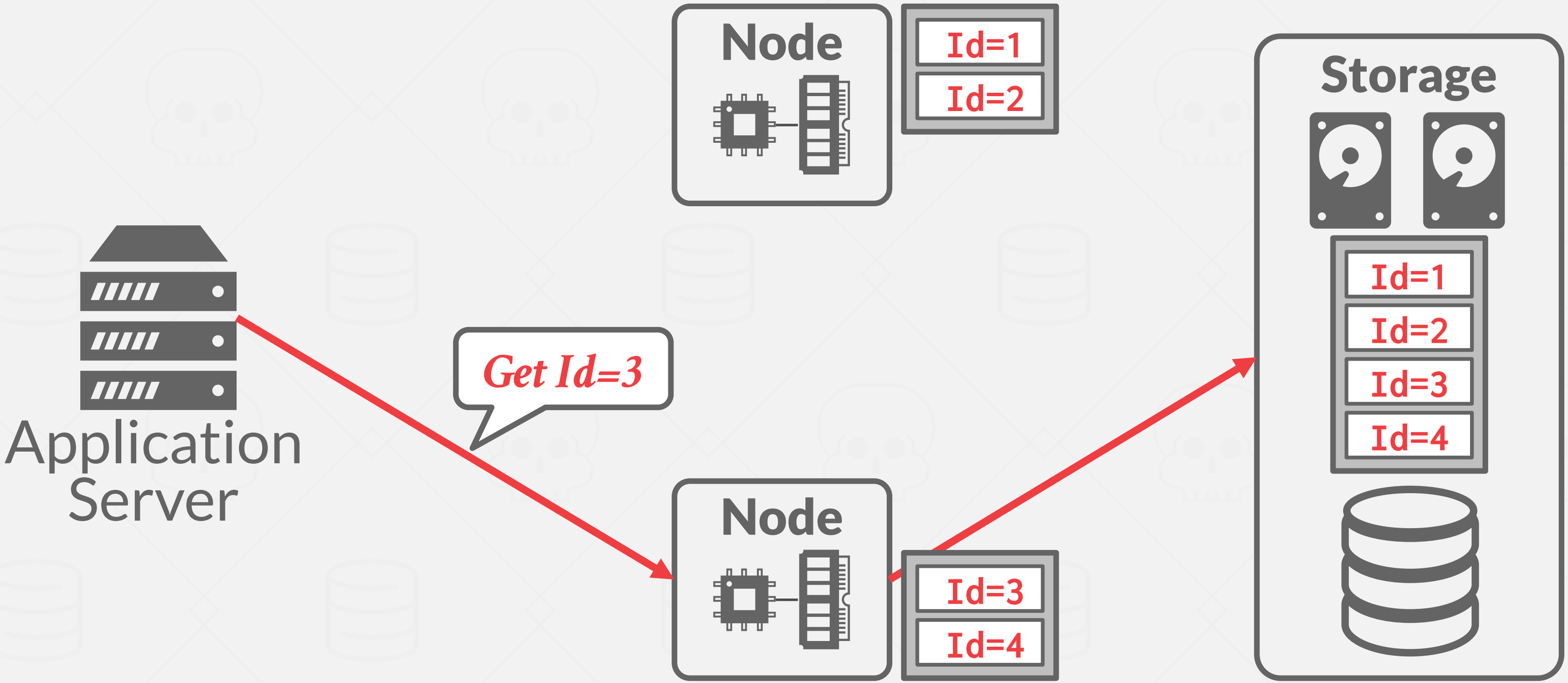
Ideal Query:

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

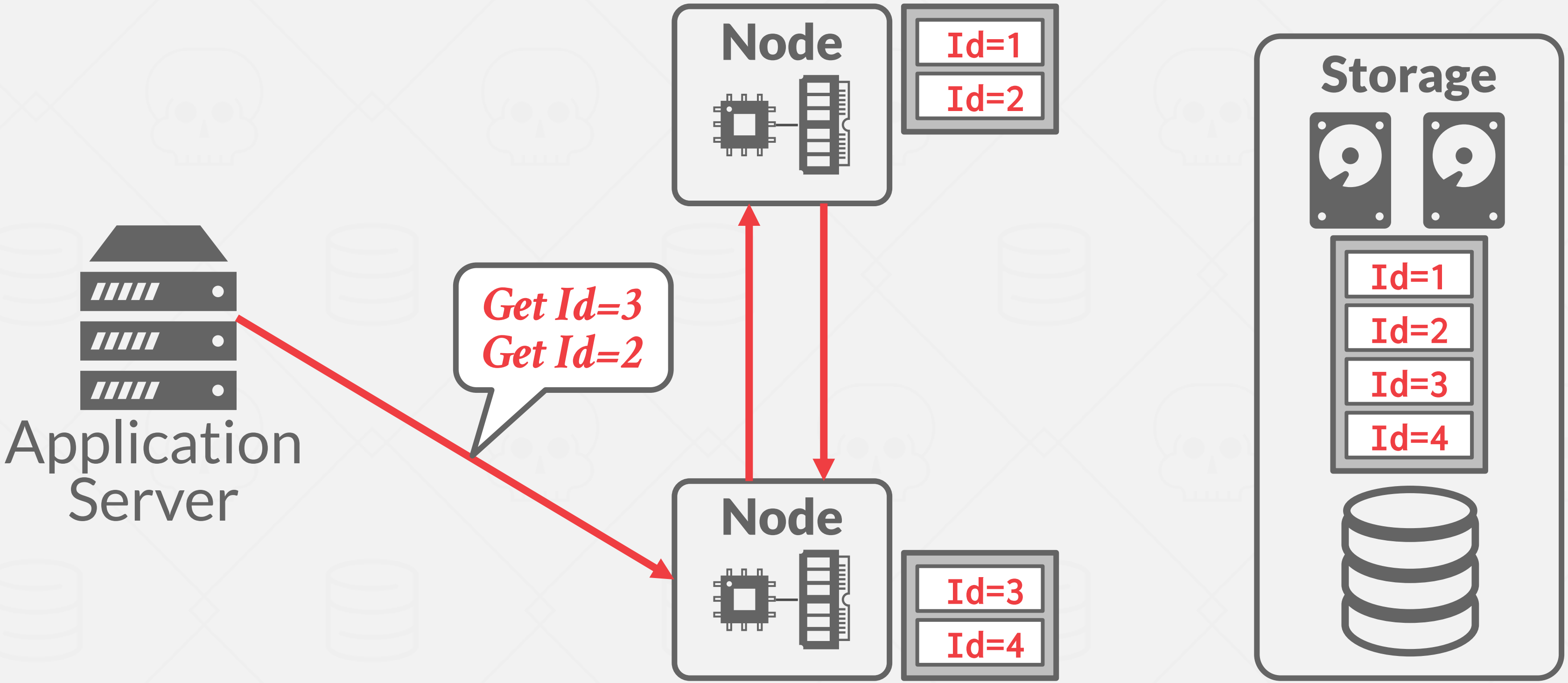
LOGICAL PARTITIONING



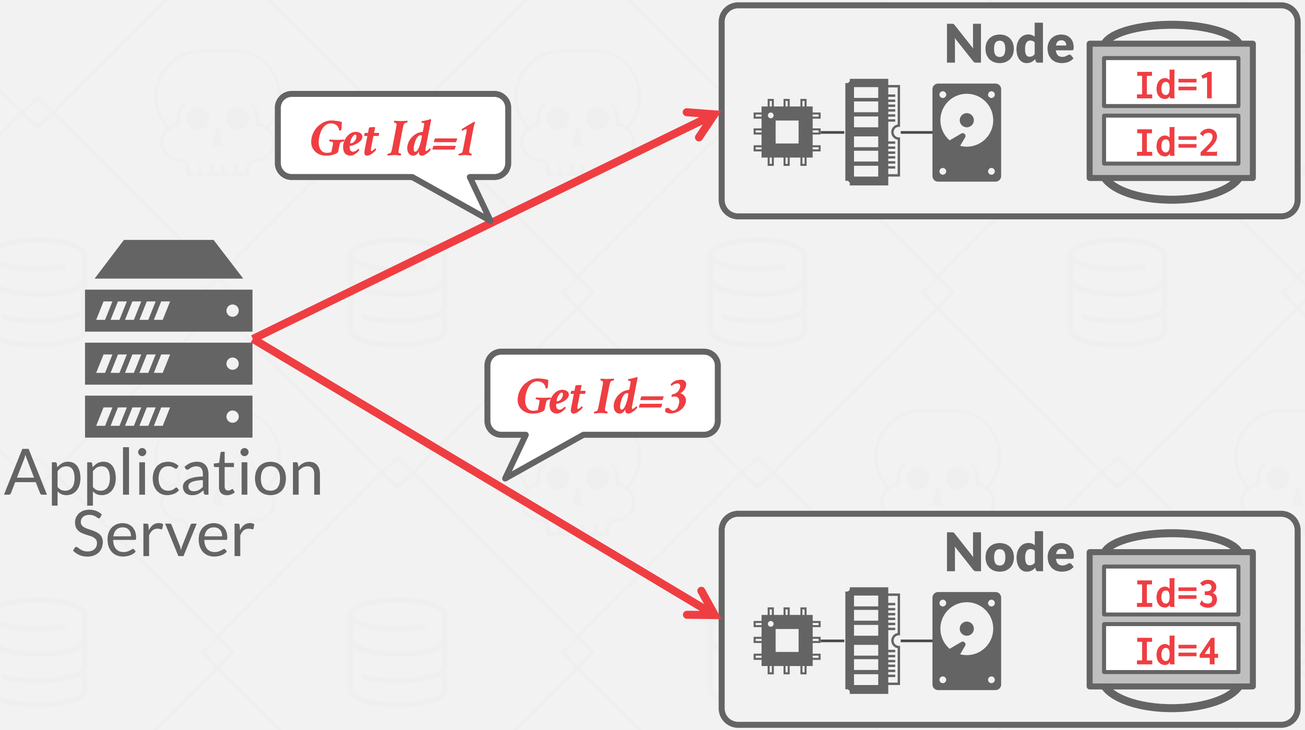
LOGICAL PARTITIONING



LOGICAL PARTITIONING



PHYSICAL PARTITIONING



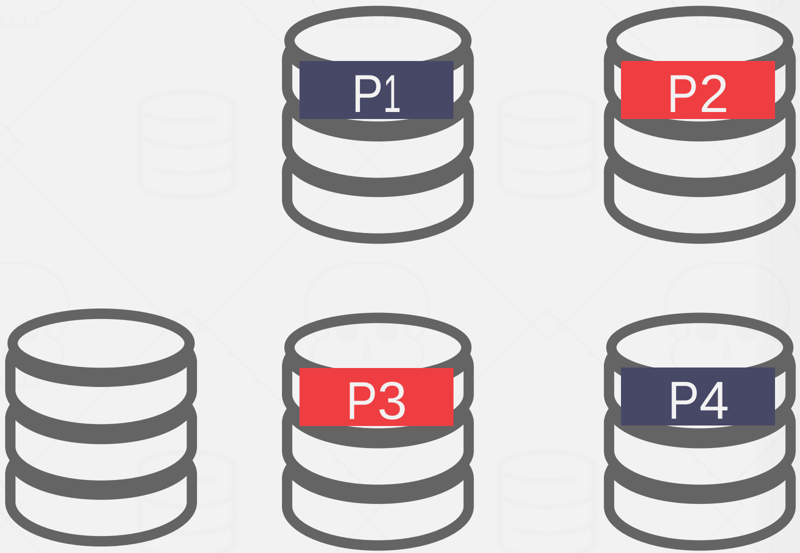
HORIZONTAL PARTITIONING

Partitioning Key

Table1

101	a	XXX	2022-11-29	*hash(a)%4 = P2*
102	b	XXY	2022-11-28	*hash(b)%4 = P4*
103	c	XYZ	2022-11-29	*hash(c)%4 = P3*
104	d	XYX	2022-11-27	*hash(d)%4 = P2*
105	e	XYY	2022-11-29	*hash(e)%4 = P1*

Partitions



Ideal Query:

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

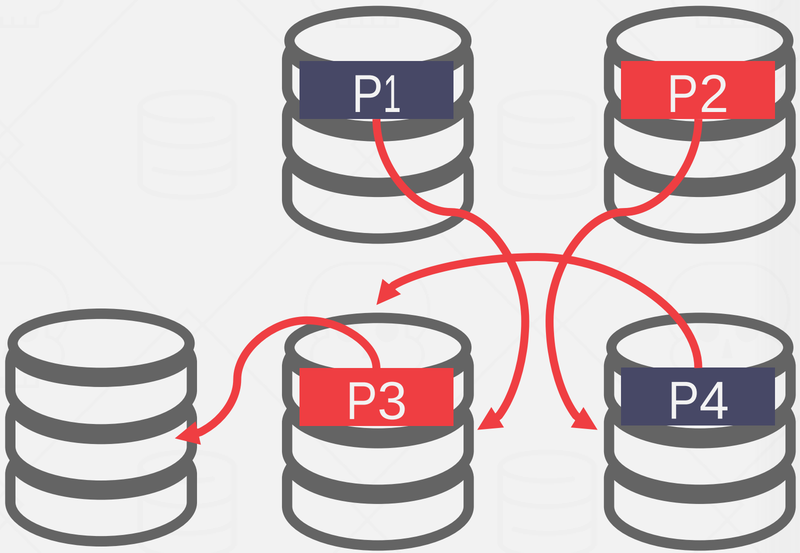
HORIZONTAL PARTITIONING

Partitioning Key

Table1

101	a	XXX	2022-11-29	*hash(a)%5 = P4*
102	b	XXY	2022-11-28	*hash(b)%5 = P3*
103	c	XYZ	2022-11-29	*hash(c)%5 = P5*
104	d	XYX	2022-11-27	*hash(d)%5 = P1*
105	e	XYY	2022-11-29	*hash(e)%5 = P3*

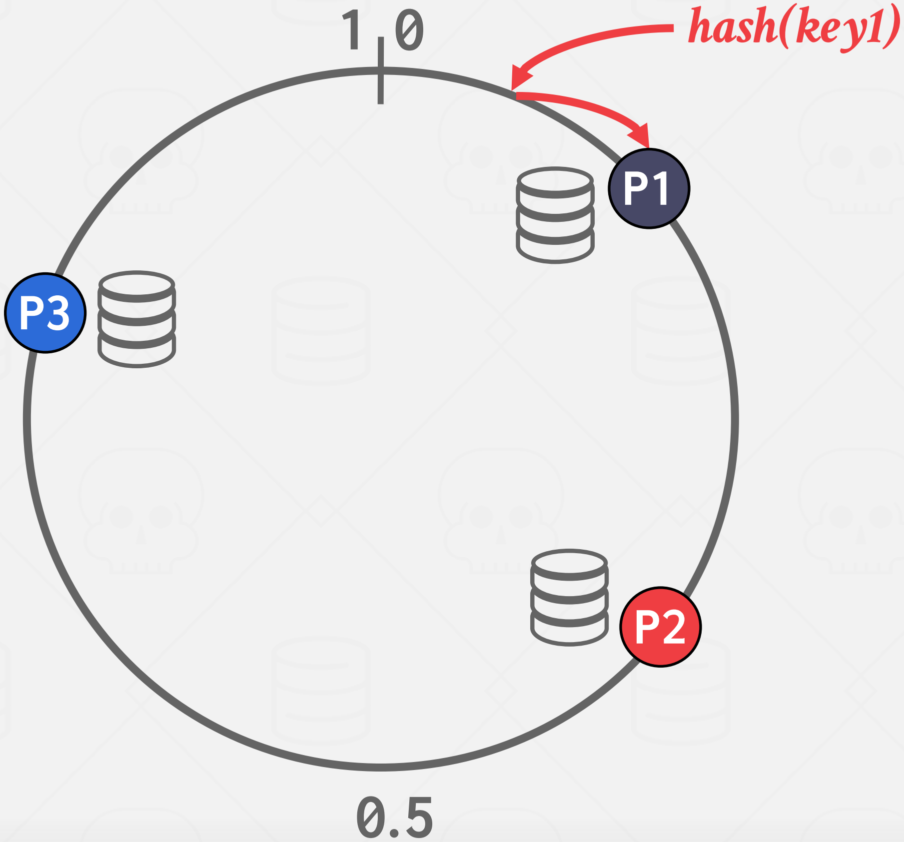
Partitions



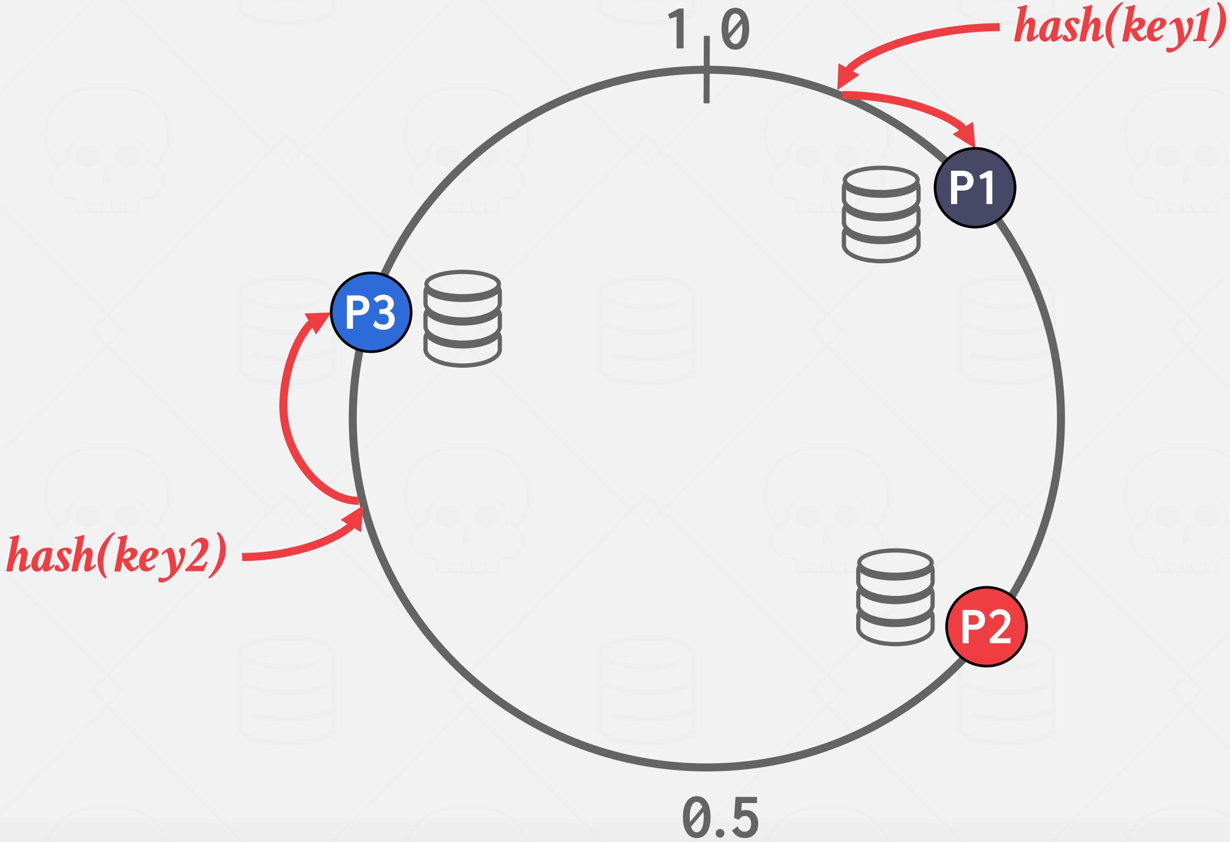
Ideal Query:

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

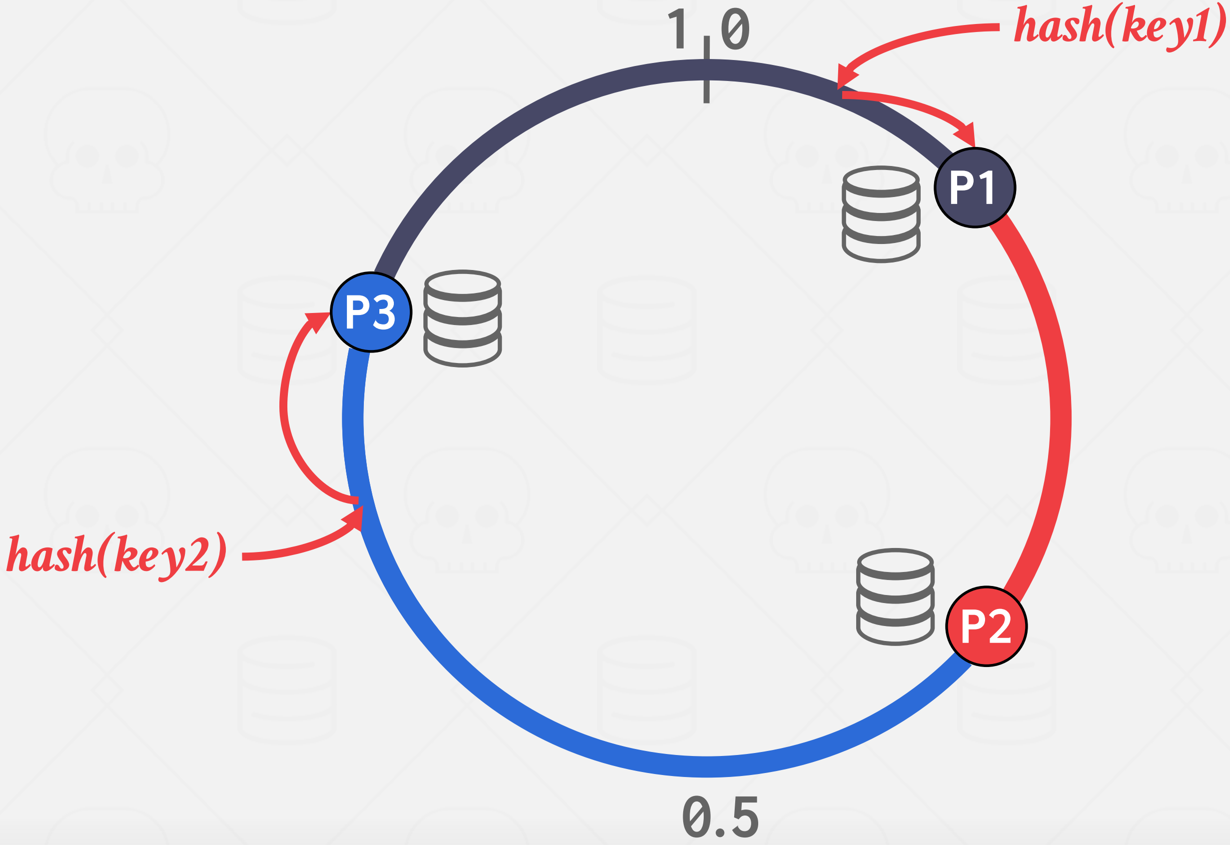
CONSISTENT HASHING



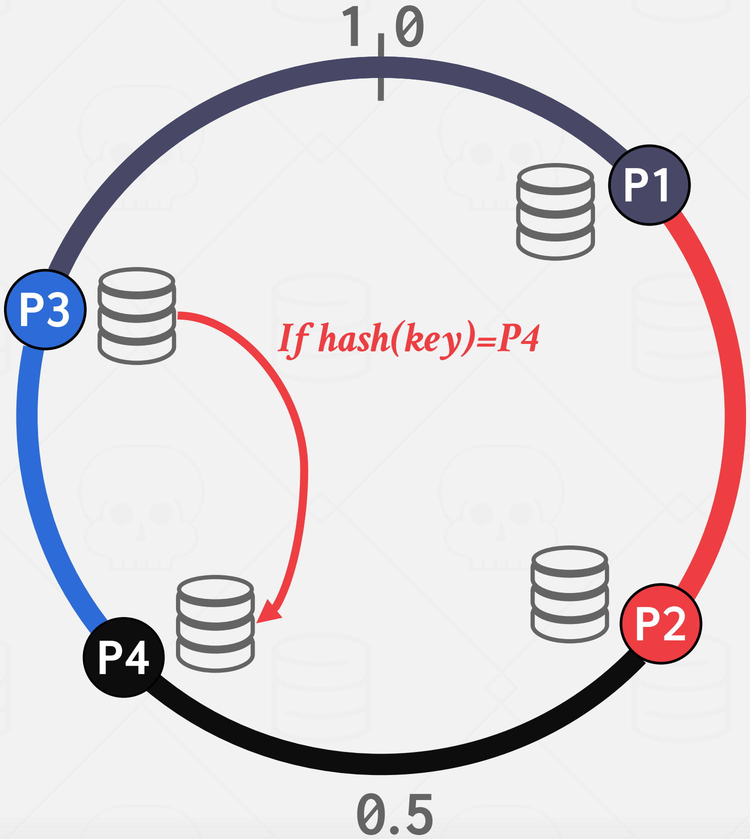
CONSISTENT HASHING



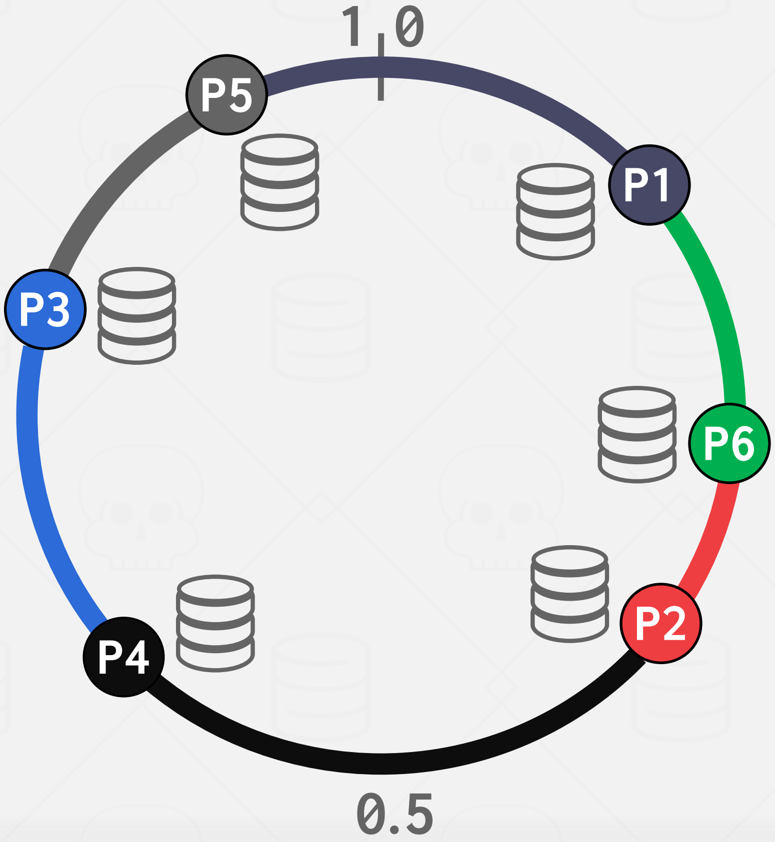
CONSISTENT HASHING



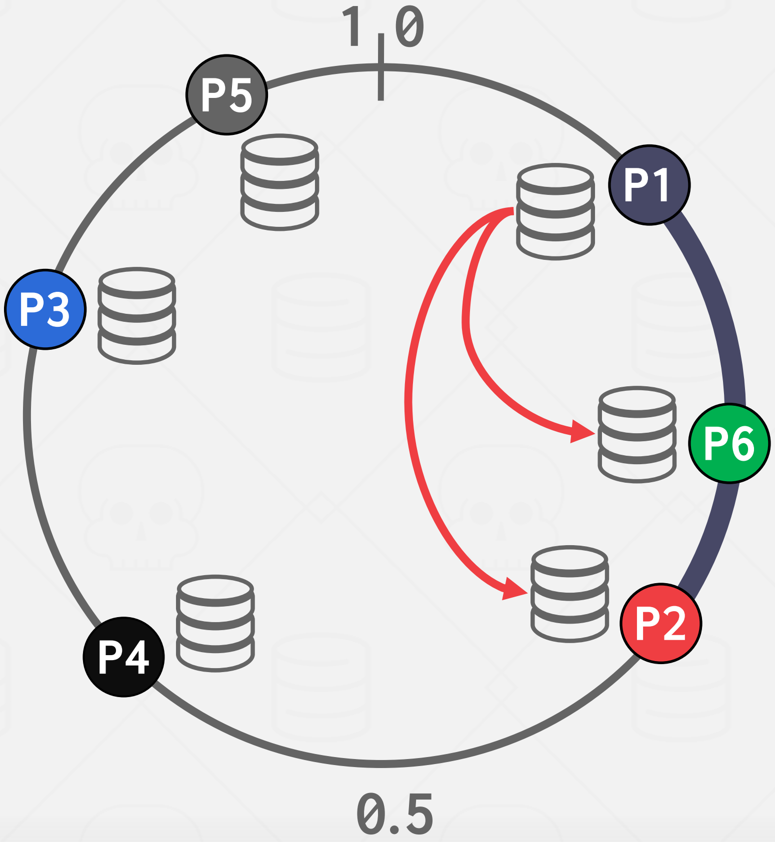
CONSISTENT HASHING



CONSISTENT HASHING

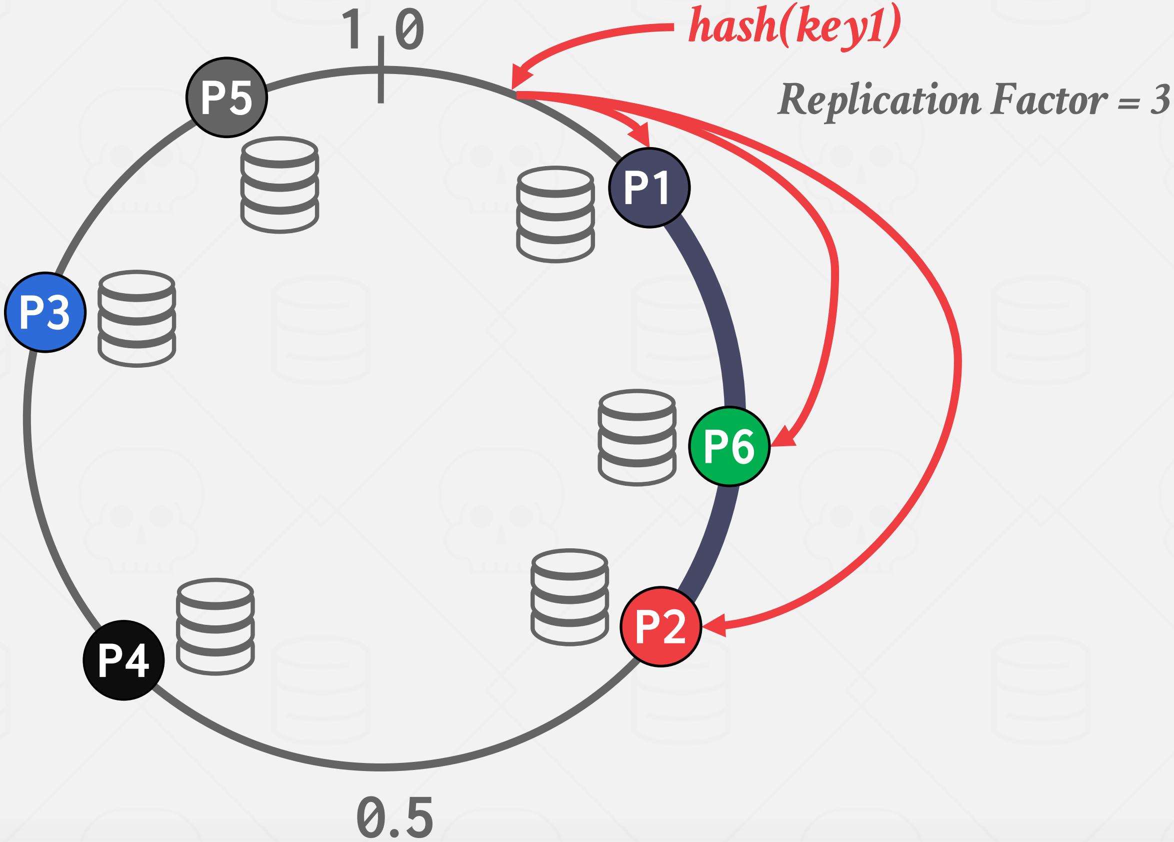


CONSISTENT HASHING

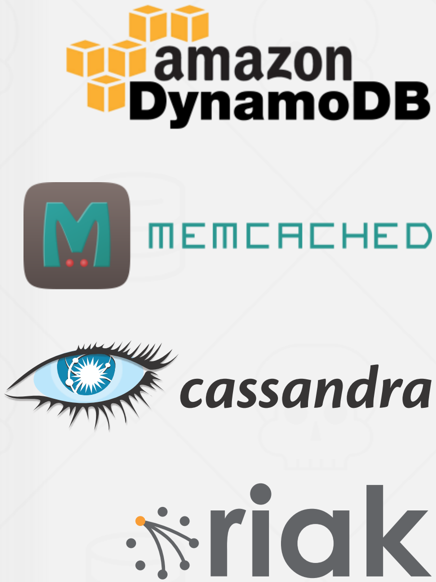


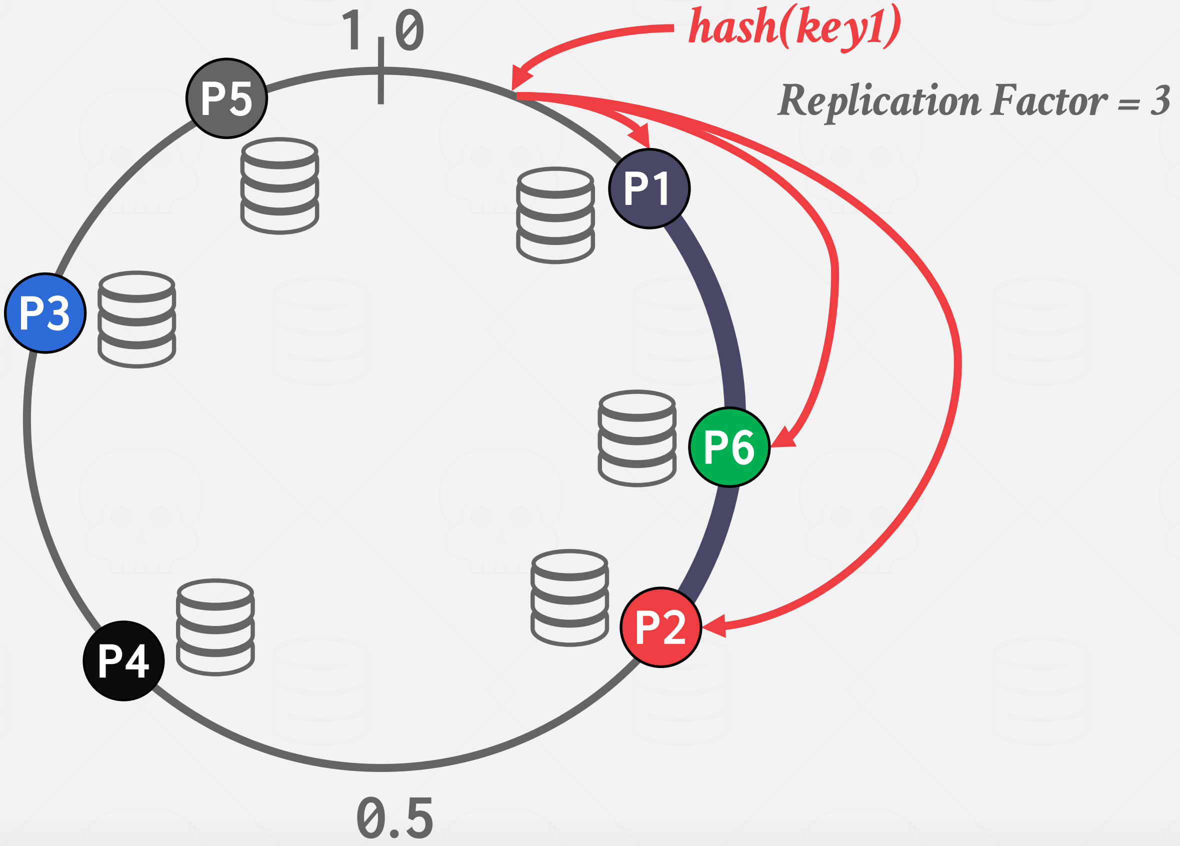
Replication Factor = 3

CONSISTENT HASHING



CONSISTENT HASHING





SINGLE-NODE VS. DISTRIBUTED

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

→ The DBMS may not need check 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

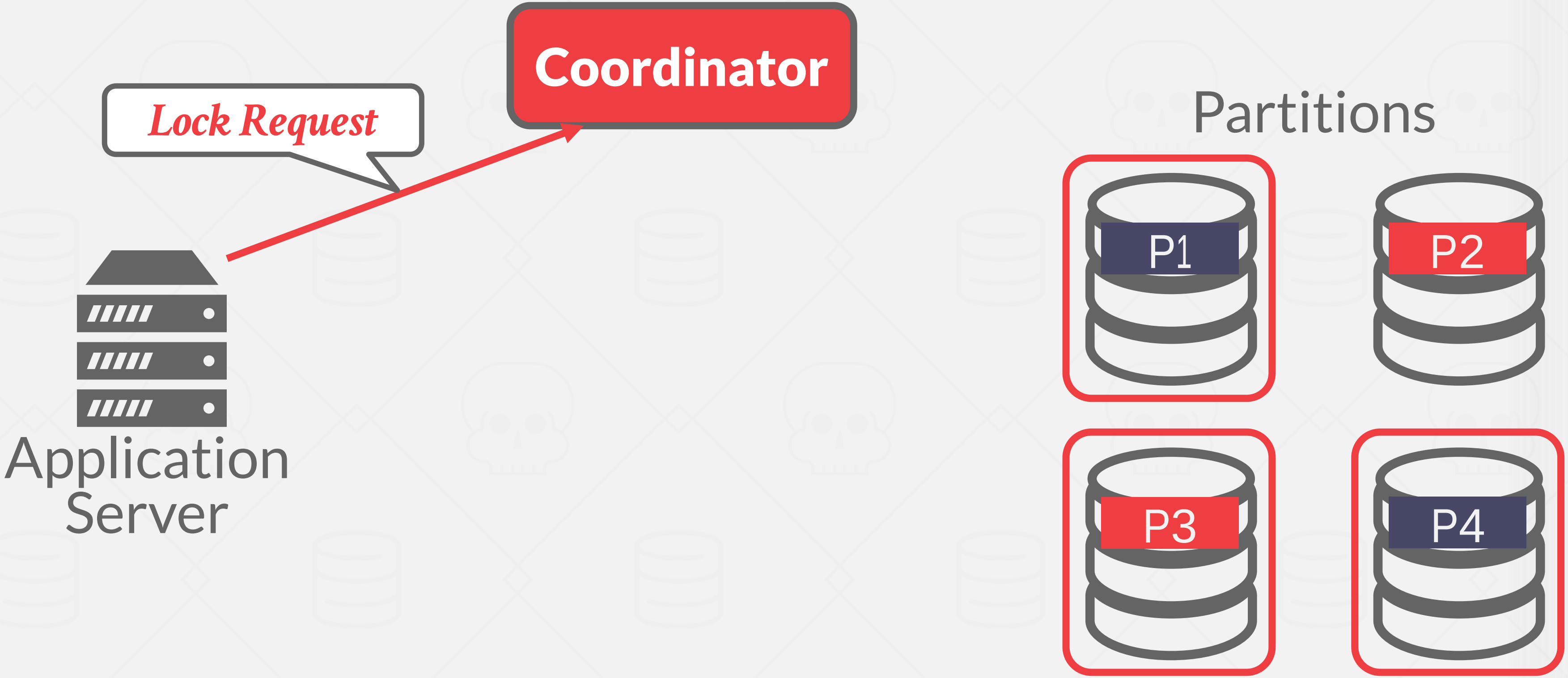
A **TP Monitor** is an example of a centralized coordinator for distributed DBMSs.

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

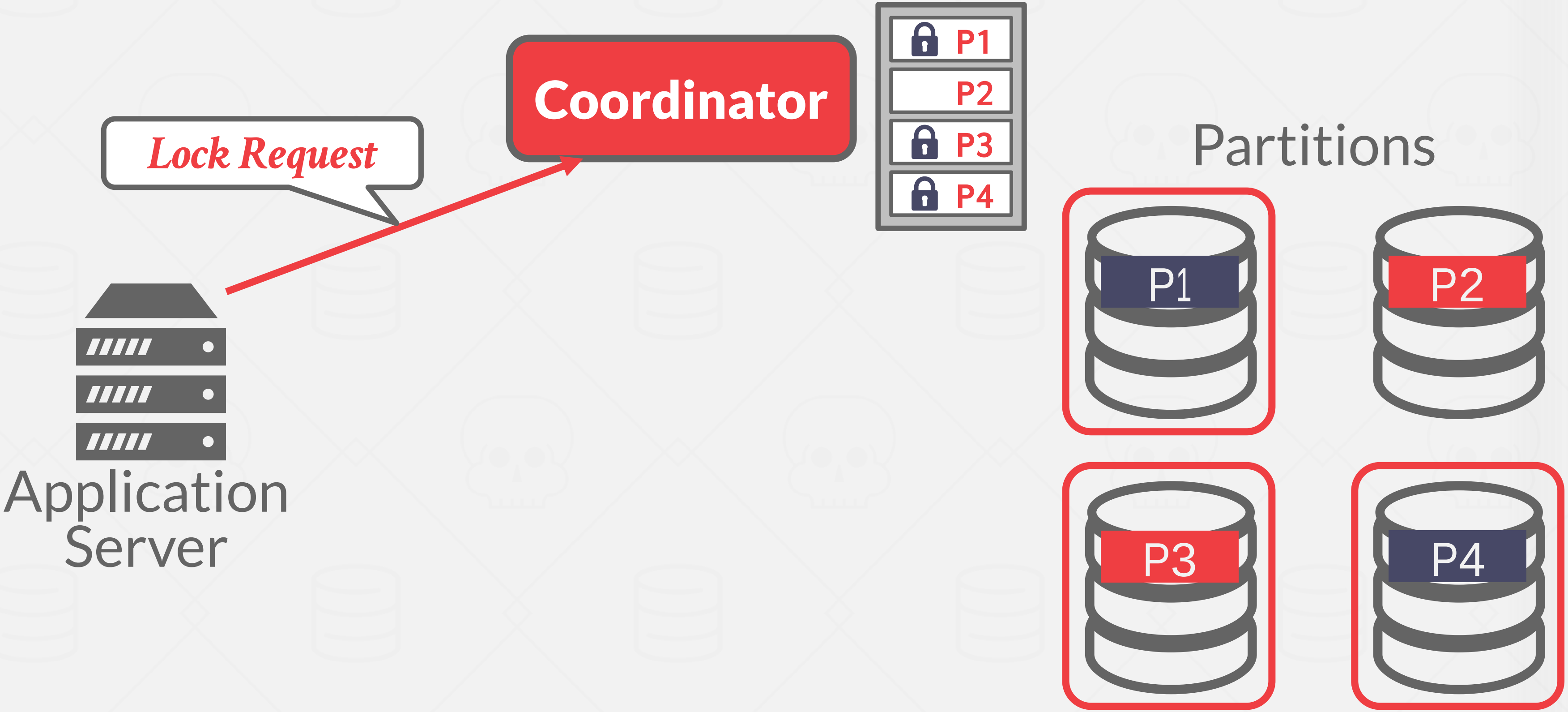
→ Examples: ATMs, Airline Reservations.

Standardized protocol from 1990s: X/Open XA

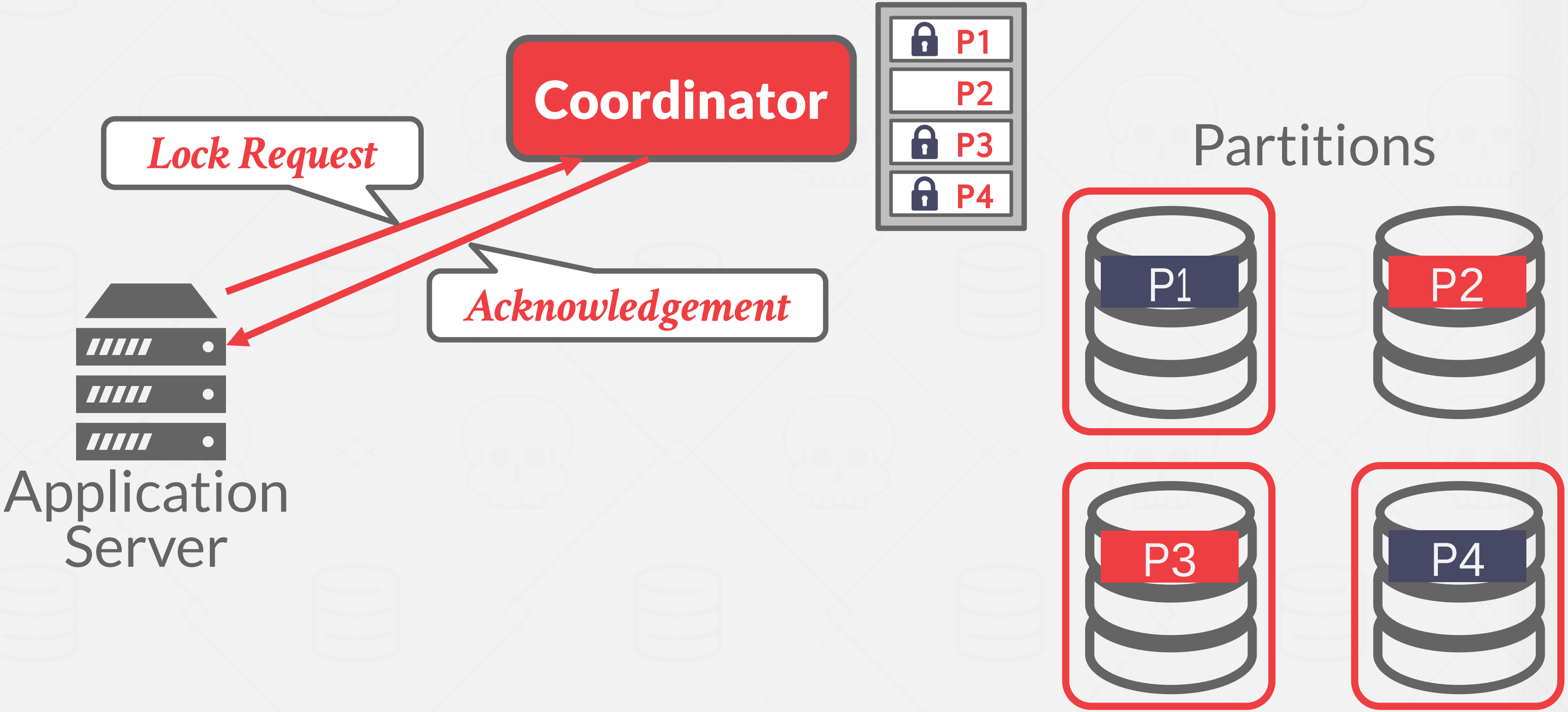
CENTRALIZED COORDINATOR



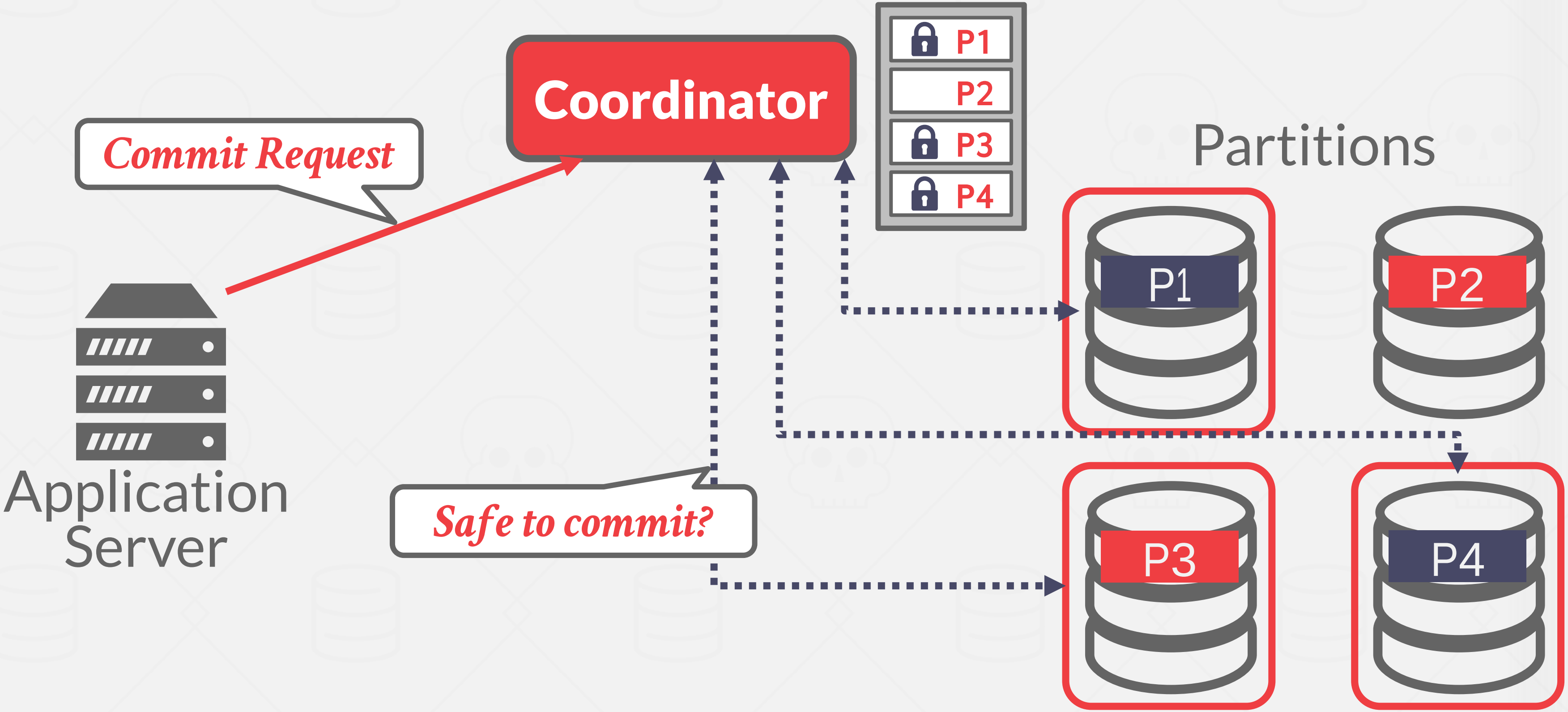
CENTRALIZED COORDINATOR



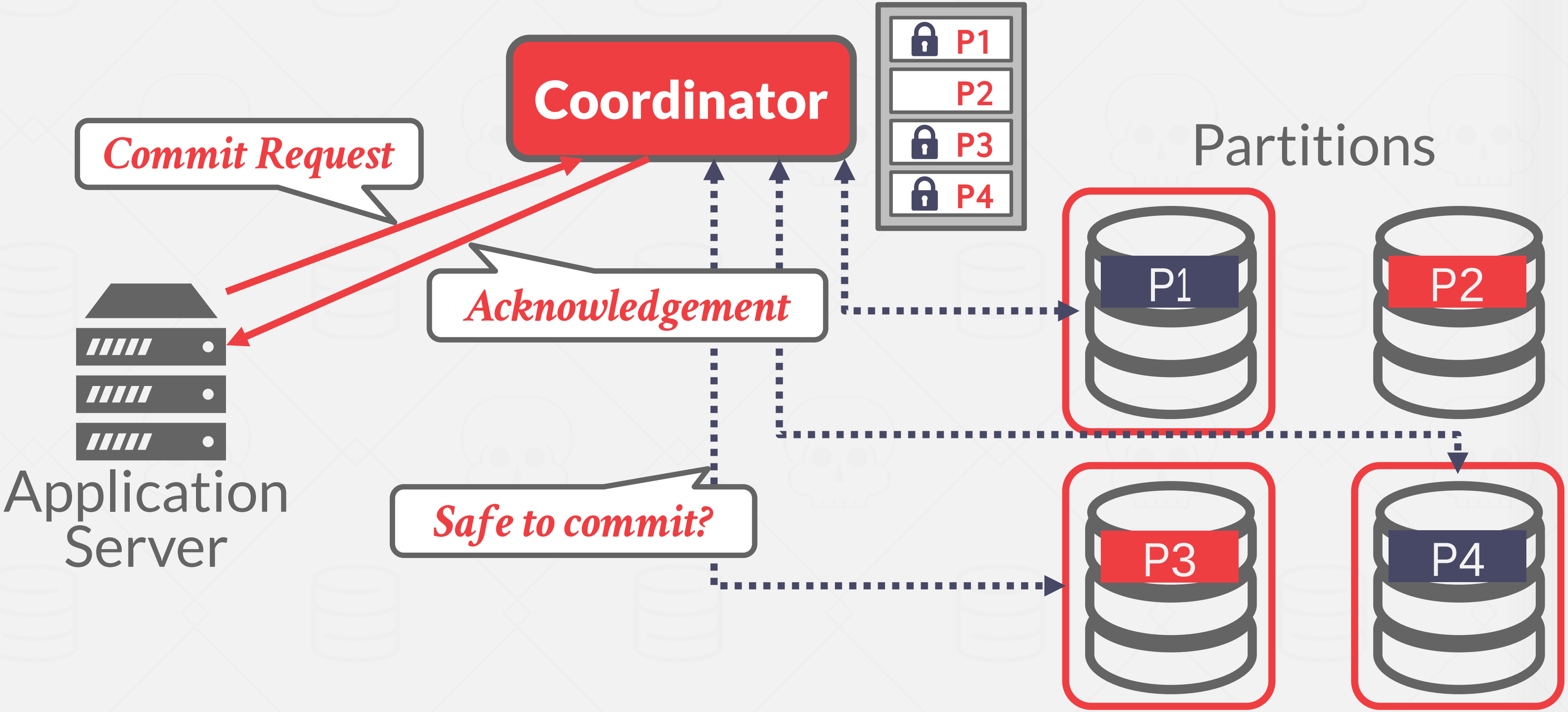
CENTRALIZED COORDINATOR



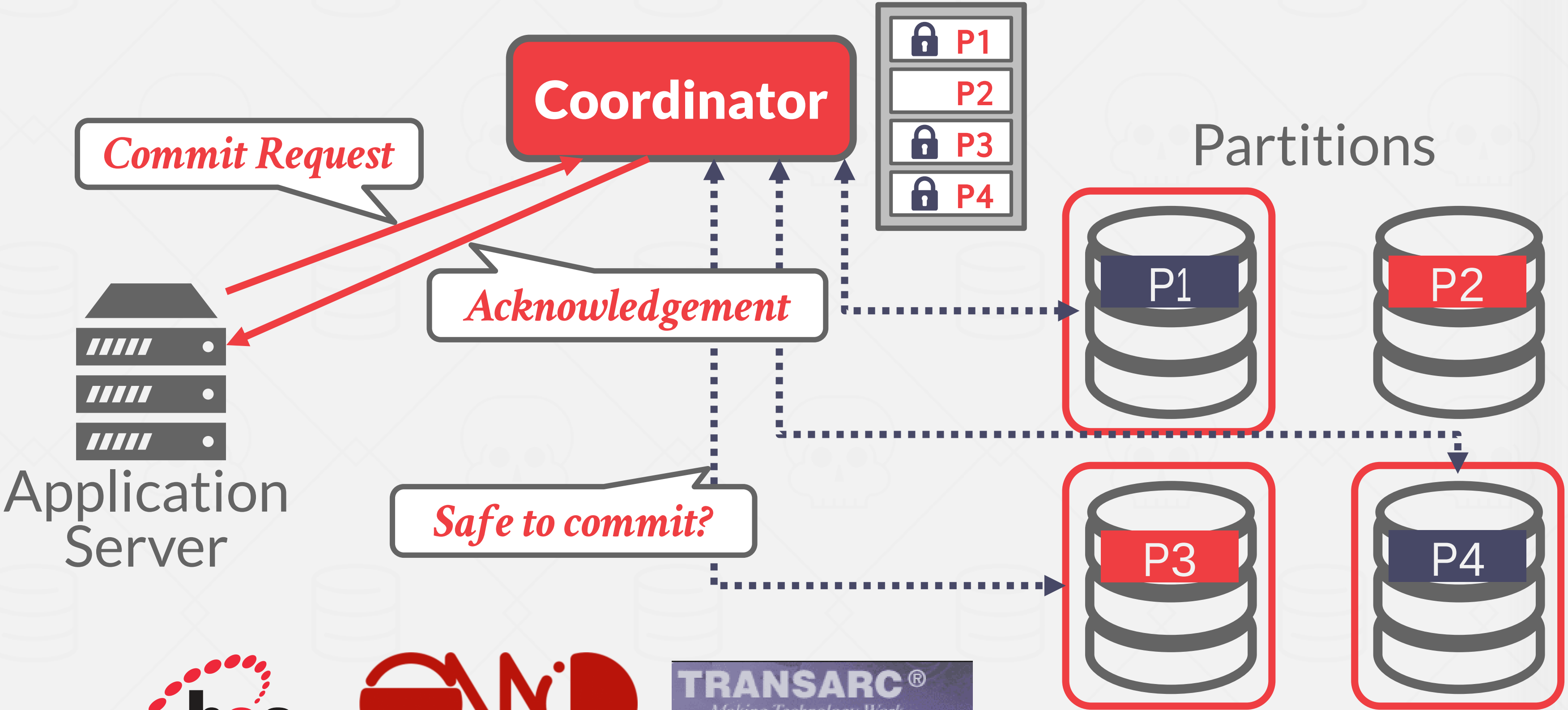
CENTRALIZED COORDINATOR



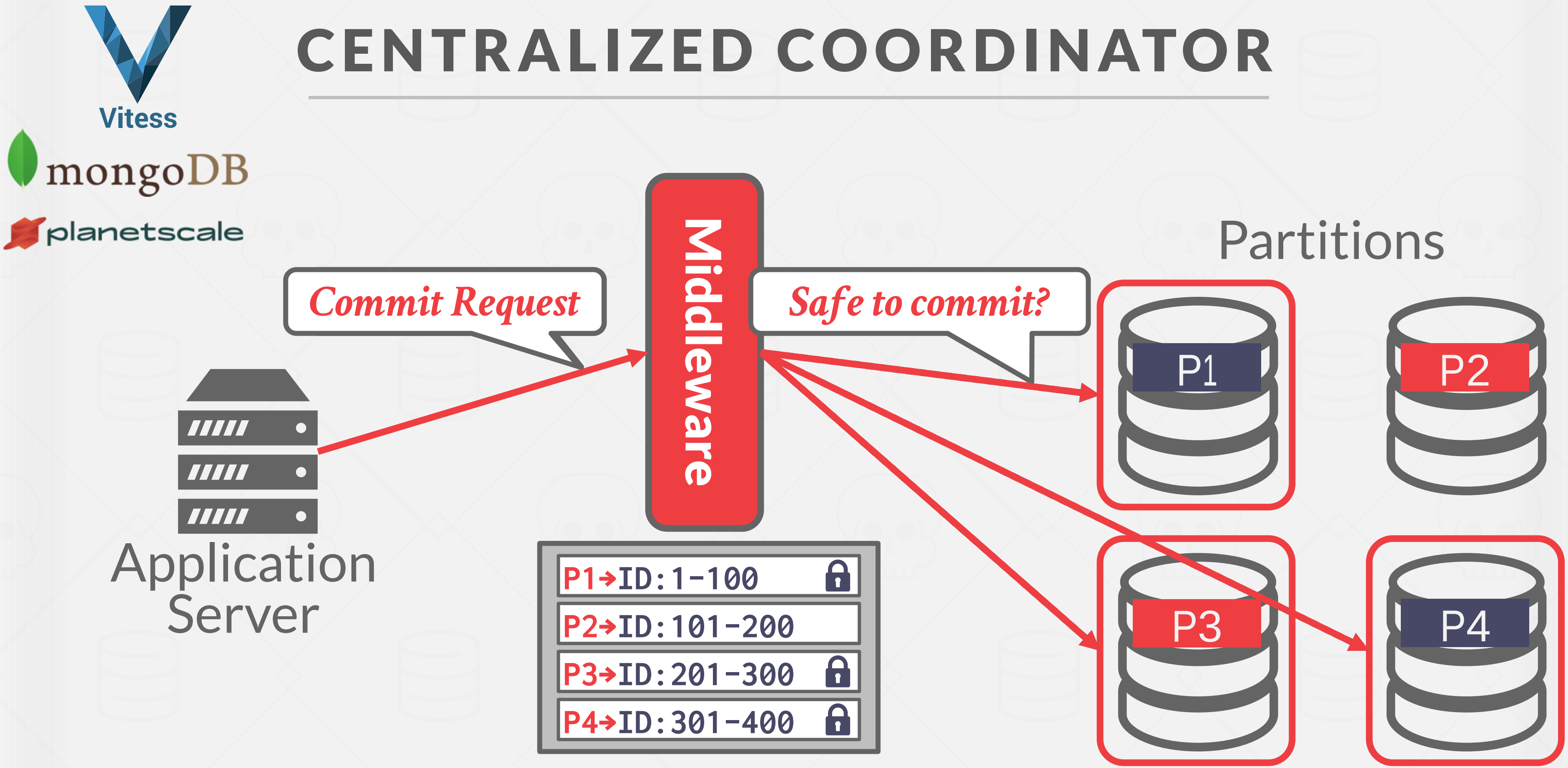
CENTRALIZED COORDINATOR



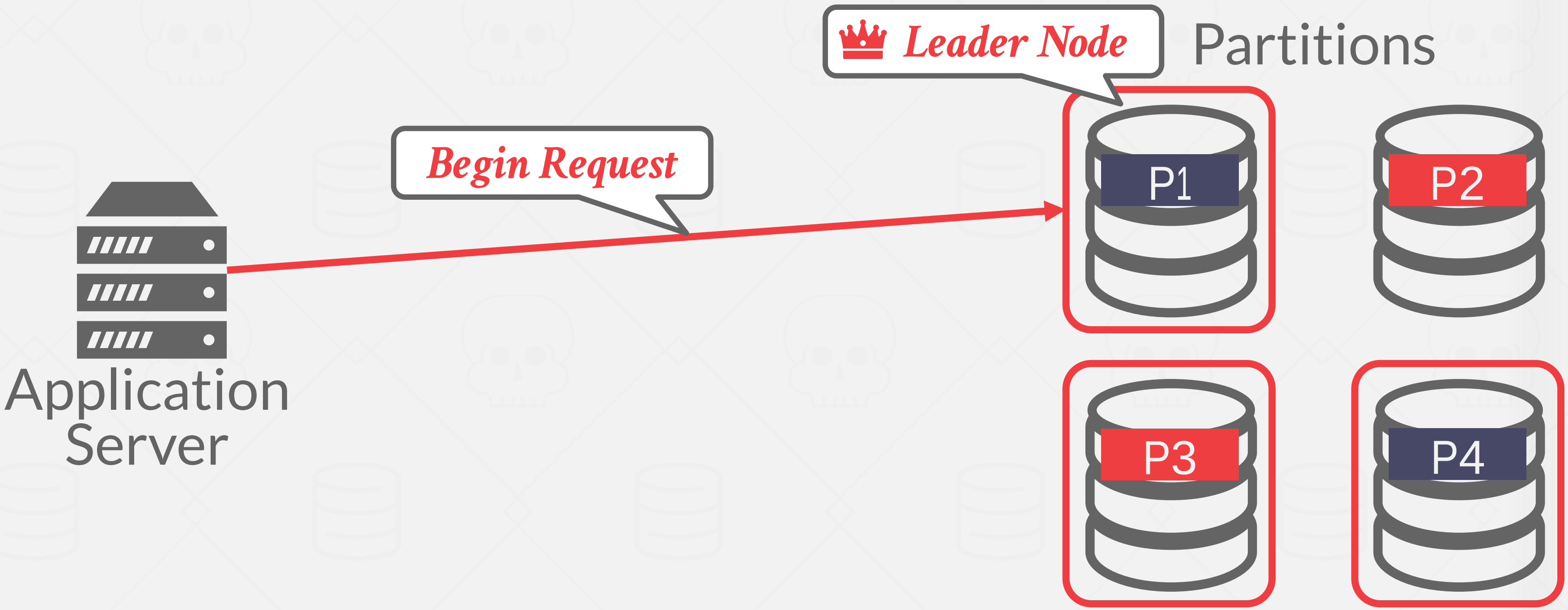
CENTRALIZED COORDINATOR



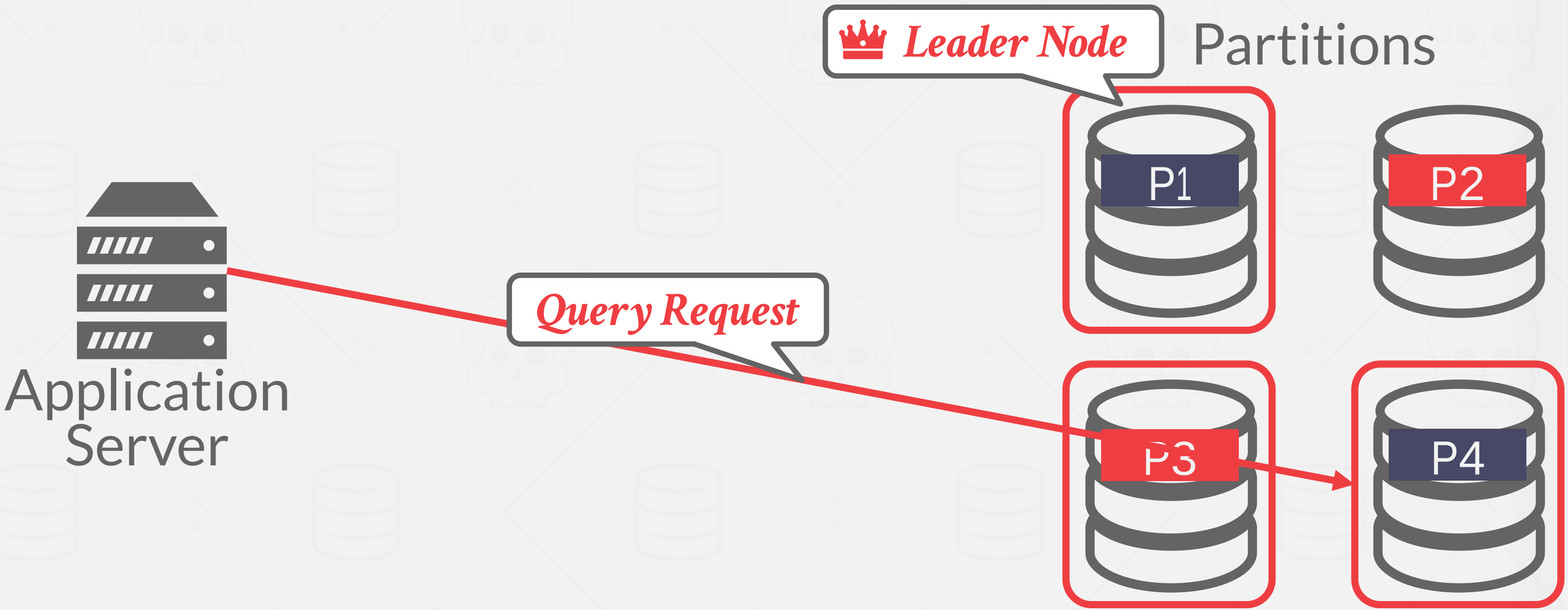
CENTRALIZED COORDINATOR



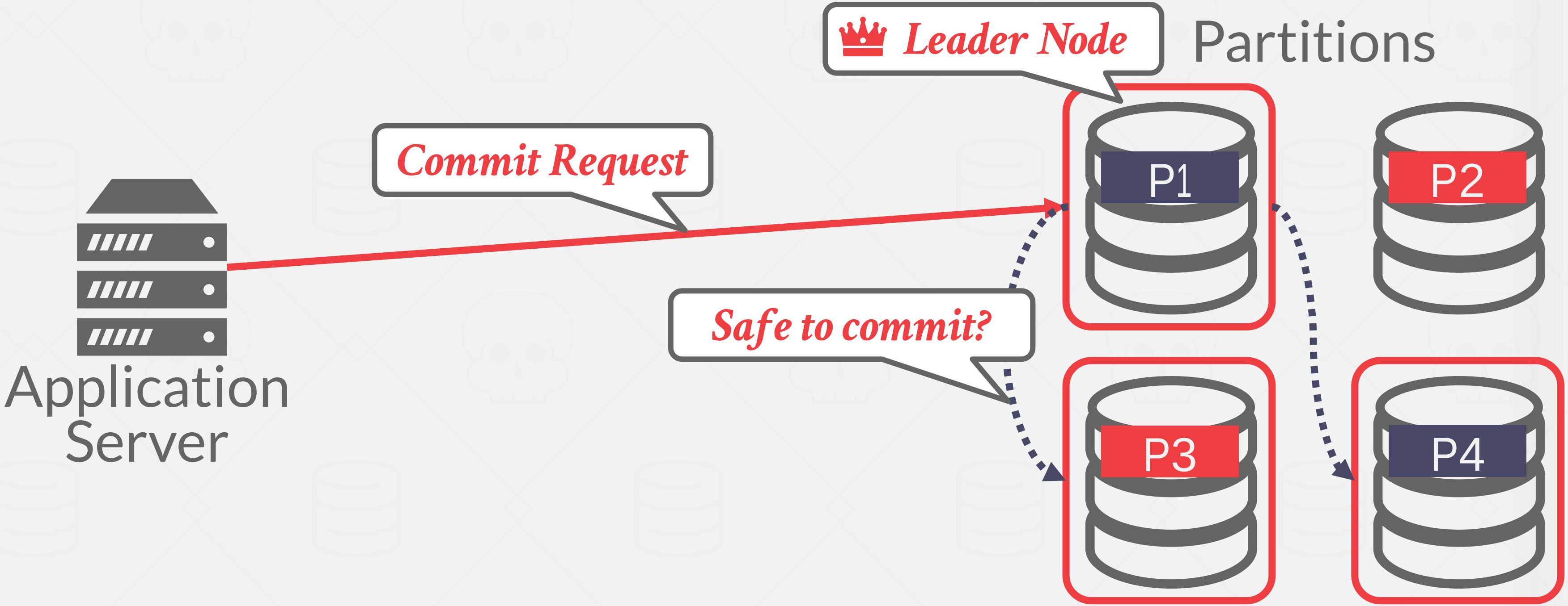
DECENTRALIZED COORDINATOR



DECENTRALIZED COORDINATOR



DECENTRALIZED COORDINATOR



DISTRIBUTED CONCURRENCY CONTROL

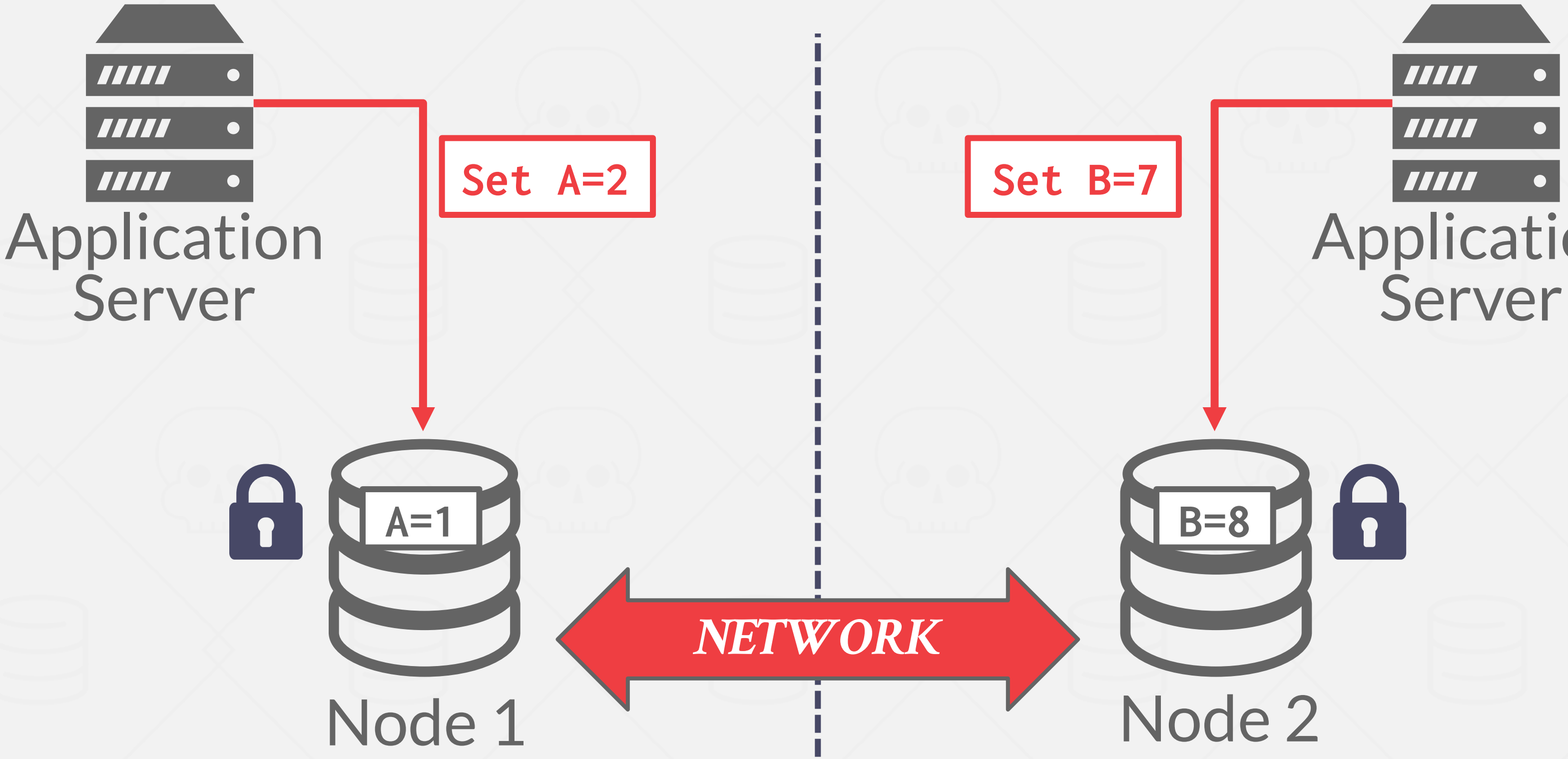
Need to allow multiple txns to execute simultaneously across multiple nodes.

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

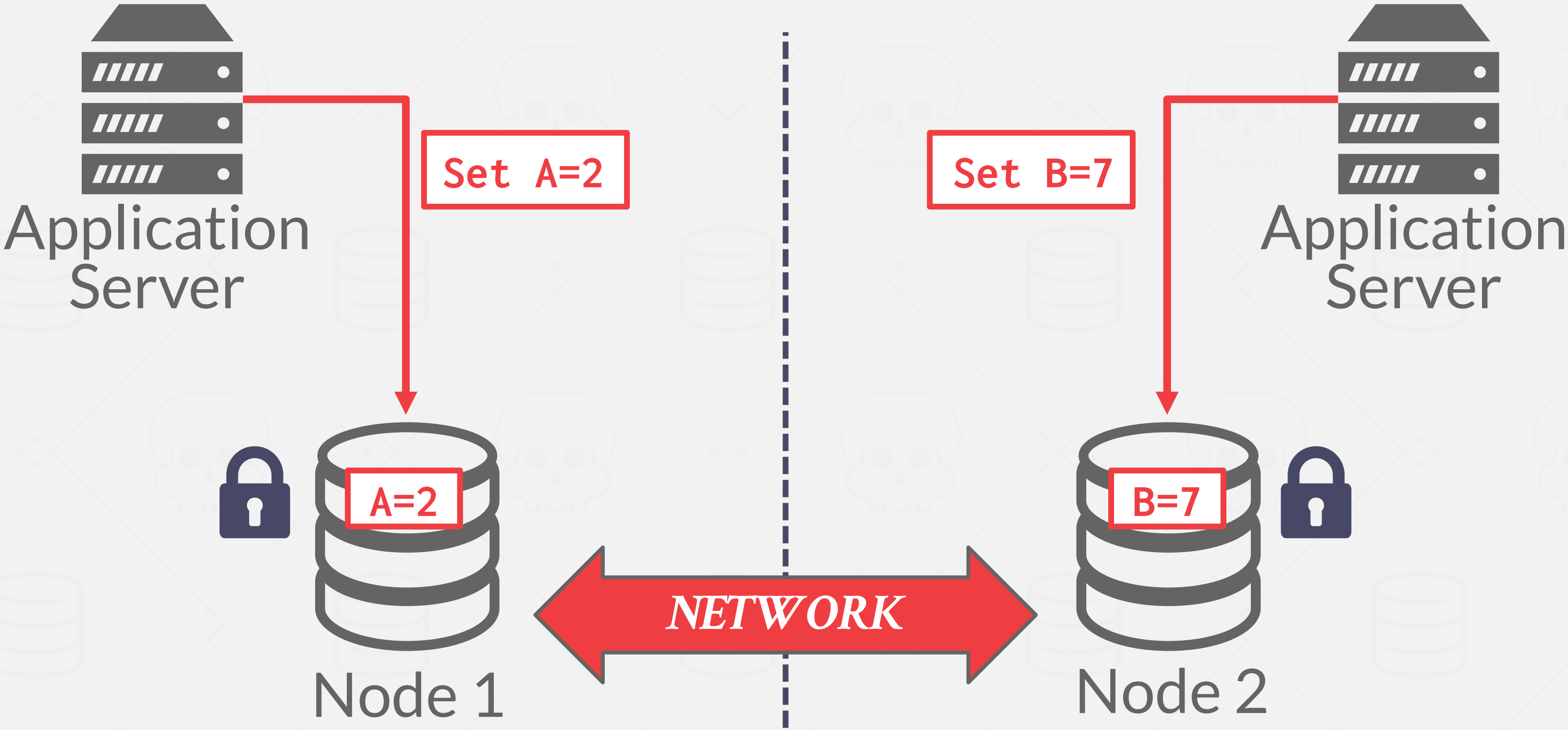
This is harder because of:

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

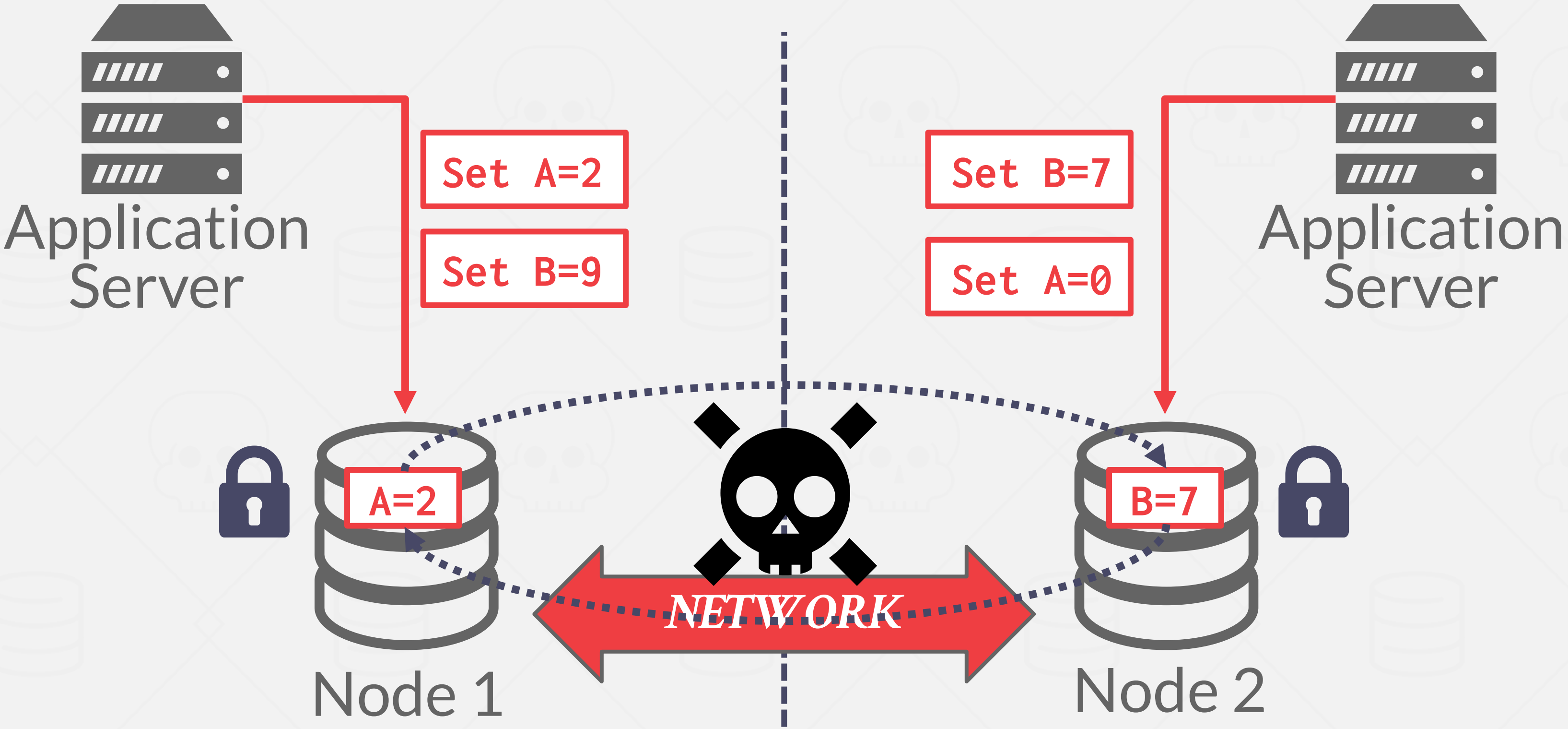
DISTRIBUTED 2PL



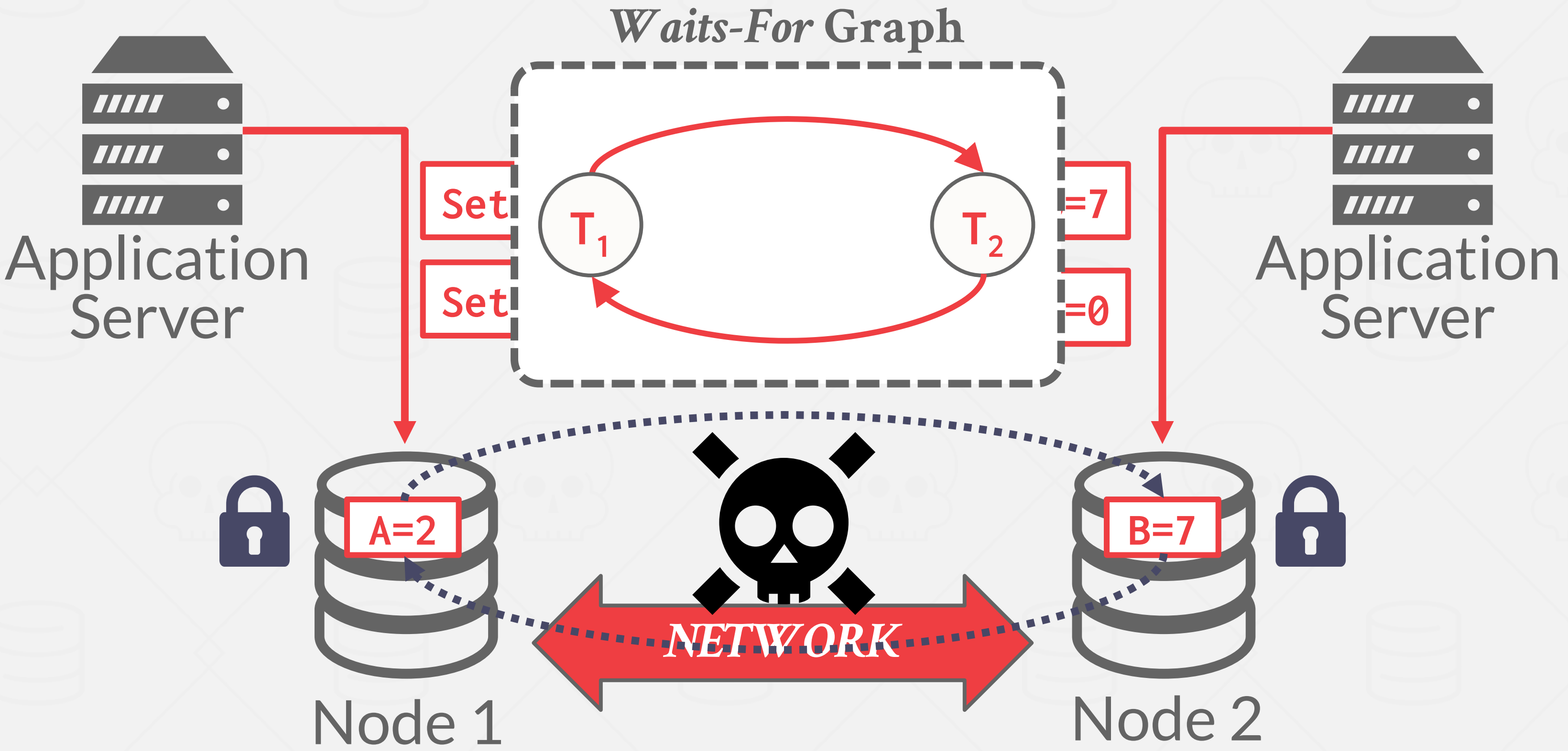
DISTRIBUTED 2PL



DISTRIBUTED 2PL



DISTRIBUTED 2PL



CONCLUSION

I have barely scratched the surface on distributed database systems…

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

PROJECT #4 – CONCURRENCY CONTROL

You will add support for concurrent transactions using two-phase locking in BusTub!

→ Deadlock Detection
→ Hierarchical Locking (Table, Tuple)
→ Multiple Isolation Levels
→ Aborts/Rollbacks

You do not need to worry about logging txns to disk.

Prompt: *A dramatic and vibrant painting of a giant eye in the clouds looking down on a field of grazing sheep with padlocks as their heads.*

https://15445.courses.cs.cmu.edu/fall2022/project4/

PROJECT #3 – TASKS

Lock Manager

→ Maintain internal lock table and queues.
→ Track the growing/shrinking phases of txns.
→ Notify waiting txns when their locks are available.

Deadlock Detector:

→ Build the waits-for graph and deterministically identify what txn to kill off to break deadlocks

Execution Engine

→ Modify Project #3 executors to support txn requests.

PROJECT #3 - LEADERBOARD

We have designed the Terrier benchmark to measure who has the fastest BusTub implementation!

Tasks:
→ UpdateExecutor
→ Predicate Pushdown

THINGS TO NOTE

Do **not** change any file other than the ones that you submit to Gradescope.

Make sure you pull in the latest changes from the BusTub main branch.

Post your questions on Piazza or come to TA office hours.

Compare against our solution in your browser!

PLAGIARISM WARNING

Your project implementation must be your own work.

→ You may **not** copy source code from other groups or the web.

→ Do **not** publish your implementation on Github.

Plagiarism will **not** be tolerated.
See CMU's Policy on Academic Integrity for additional information.

NEXT CLASS

Distributed OLTP Systems
Replication
CAP Theorem
Real-World Examples