Carnegie Mellon University

Intro to Database Systems (15-445/645)

Lecture #22

Intro to Distributed Databases

SPRING 2024 ❯❯ Prof. Jignesh Patel

ADMINISTRIVIA

HW #5 is released. Due Sat April 20, 2024 @ 11:59pm

Project #4 is released. Due Sun April 28, 2024 @ 11:59pm

→ Checkpoint to keep you on track and not wait till the last minute

Final Exam

→ Thu May 2, 2024 @ 05:30pm-08:30pm

Lectures #23 and #24

→ Recorded lectures and will be posted next week

Lecture #26: Guest Speaker from Snowflake

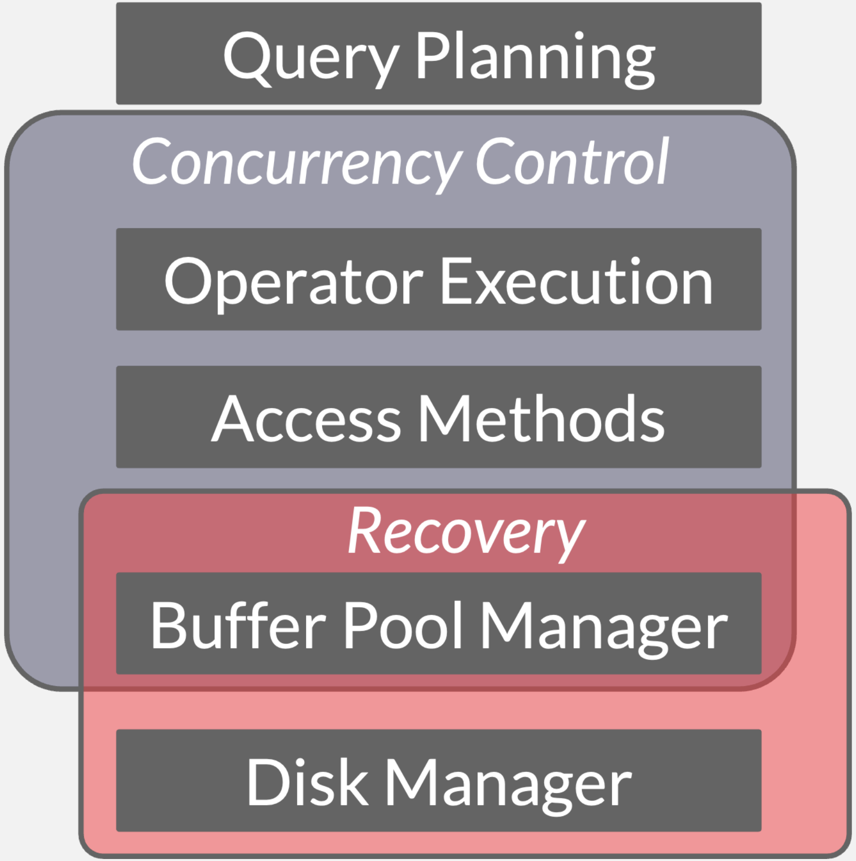
→ Devin Petersohn on "Beyond SQL: Dataframes in the Database"

https://db.cs.cmu.edu/events/spring-2024-beyond-sql-dataframes-in-the-database-devin-petersohn/

COURSE STATUS

Databases are hard.

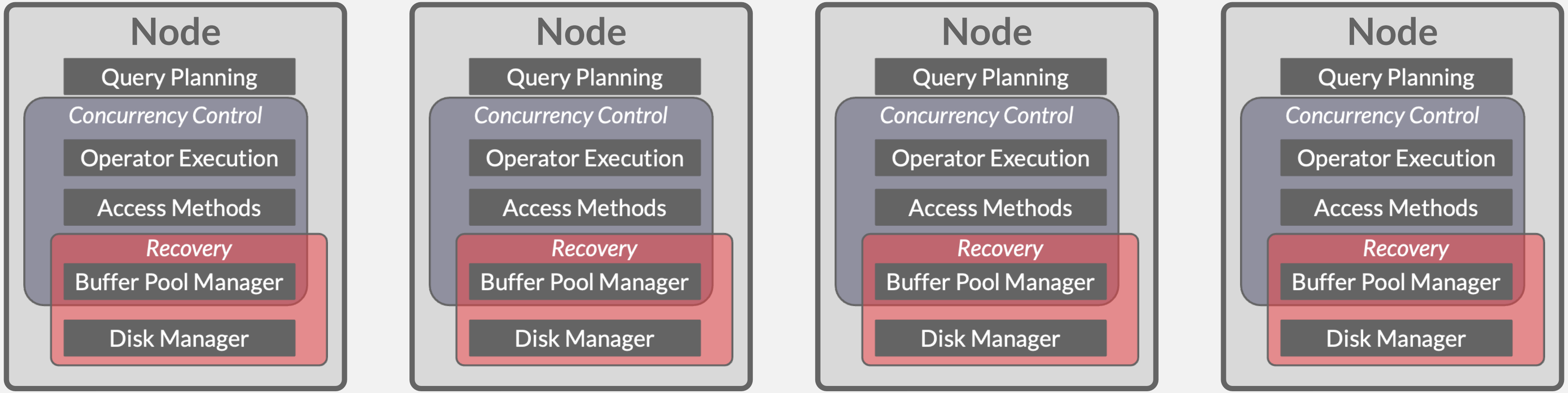
Distributed databases are harder.



COURSE STATUS

Databases are hard.

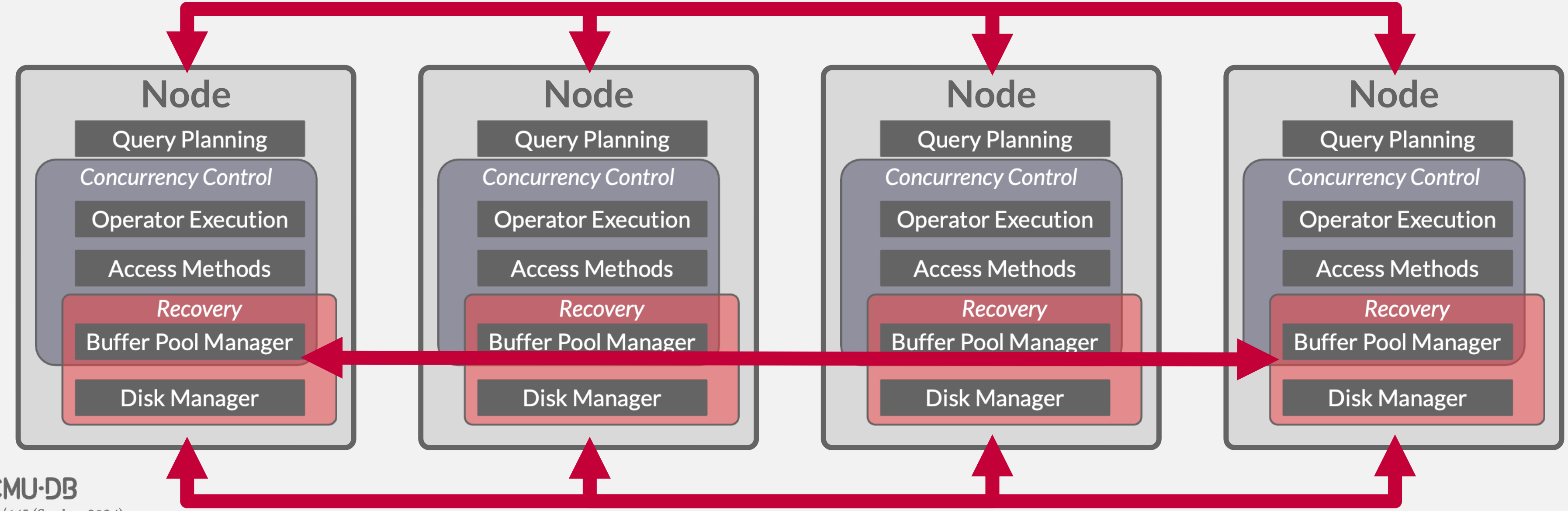
Distributed databases are harder.



COURSE STATUS

Databases are hard.

Distributed databases are harder.



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

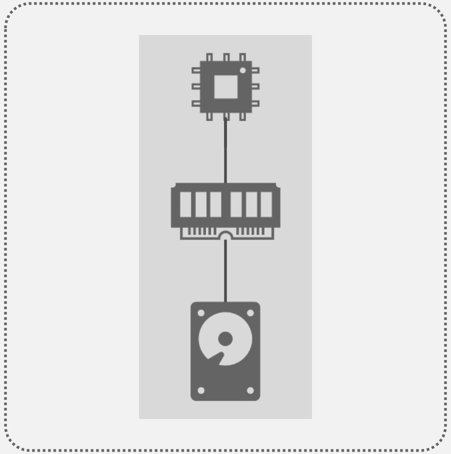
System Architectures
Design Issues
Partitioning Schemes
Distributed Concurrency Control

SYSTEM ARCHITECTURE

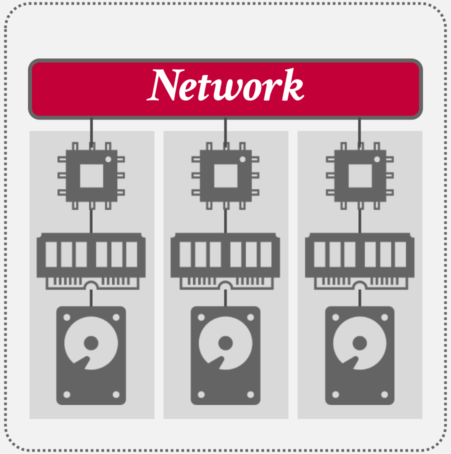
A distributed DBMS's system architecture specifies what shared resources are directly accessible to CPUs.

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

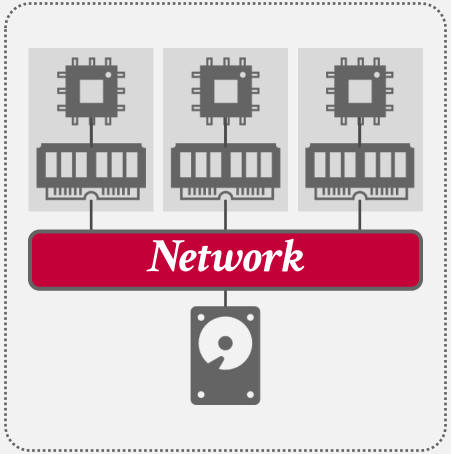
SYSTEM ARCHITECTURE



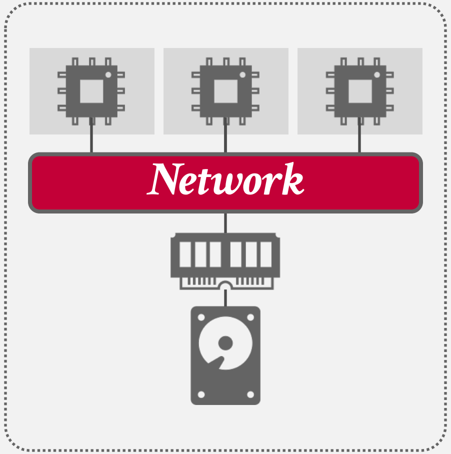
Shared Everything



Shared Nothing



Shared Disk



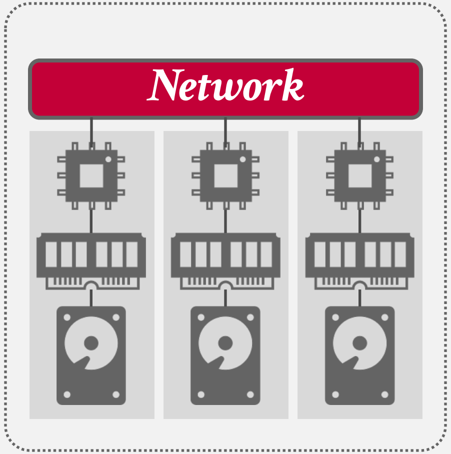
Shared Memory

SHARED NOTHING

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

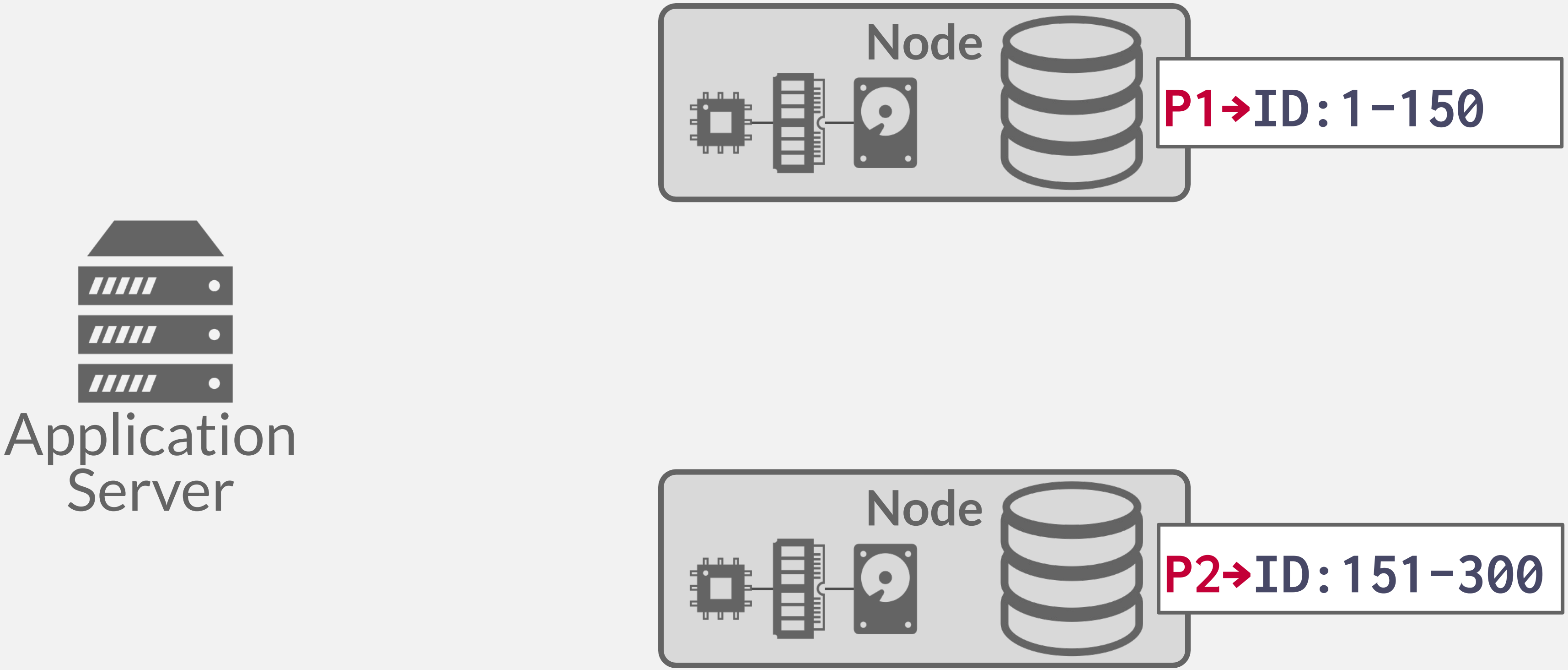
Nodes only communicate with each other via network.

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

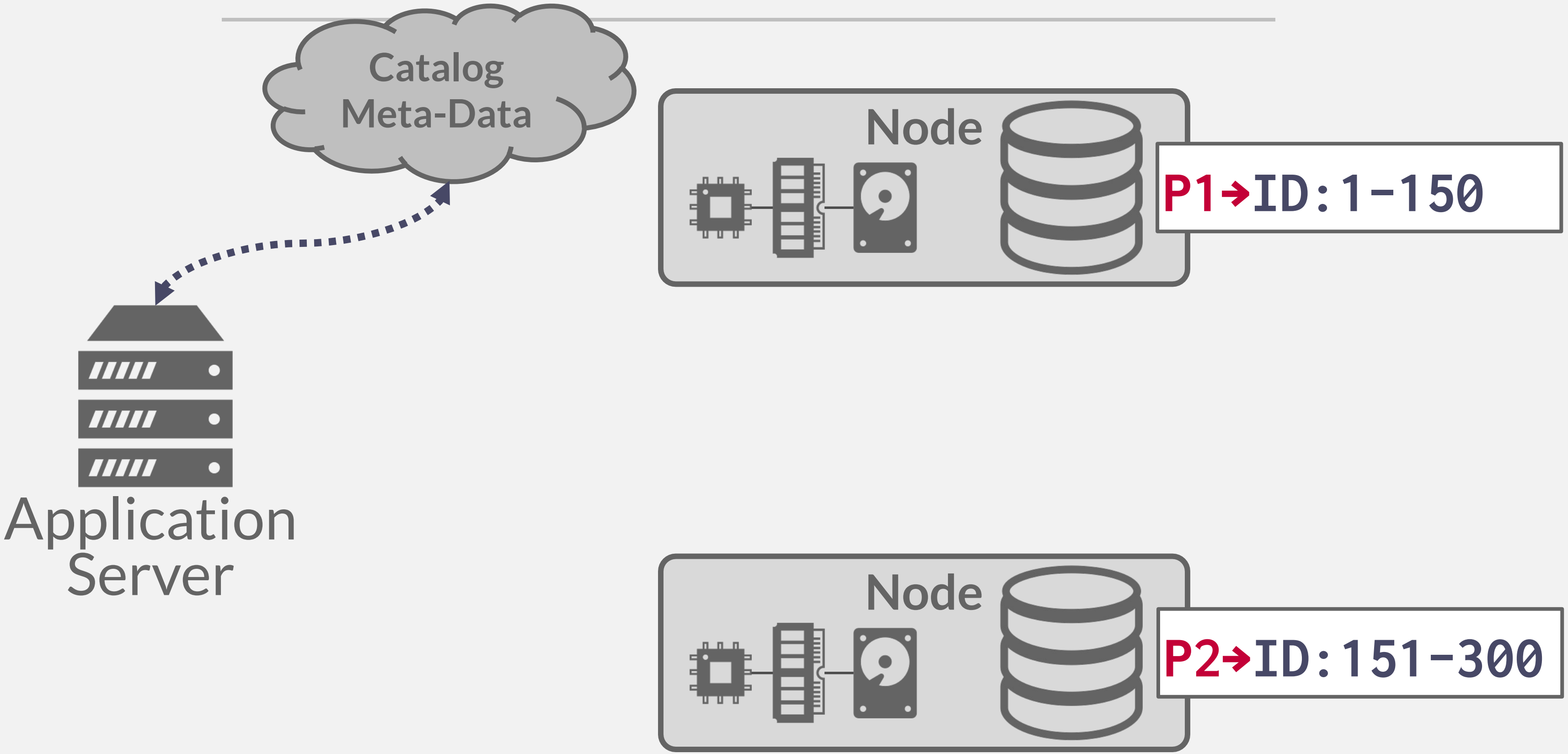


kinetica, SciDB, fauna, ORACLE NOSQL DATABASE, Yellowbrick, MEMCACHED, elastic, TigerBeetle, etcd, Exasol, citusdata, Dgraph, CockroachDB, MongoDB, FOUNDATIONDB, ClickHouse, cassandra, H-Store, VOLTDB, CrateDB, Comdb2, APACHE GEODE, orbit db, APACHE ZooKeeper, Greenplum, AEROSPIKE, SingleStore, VERTICA, YugaByte, TiDB, redis, TERADATA, OCEANBASE, Couchbase

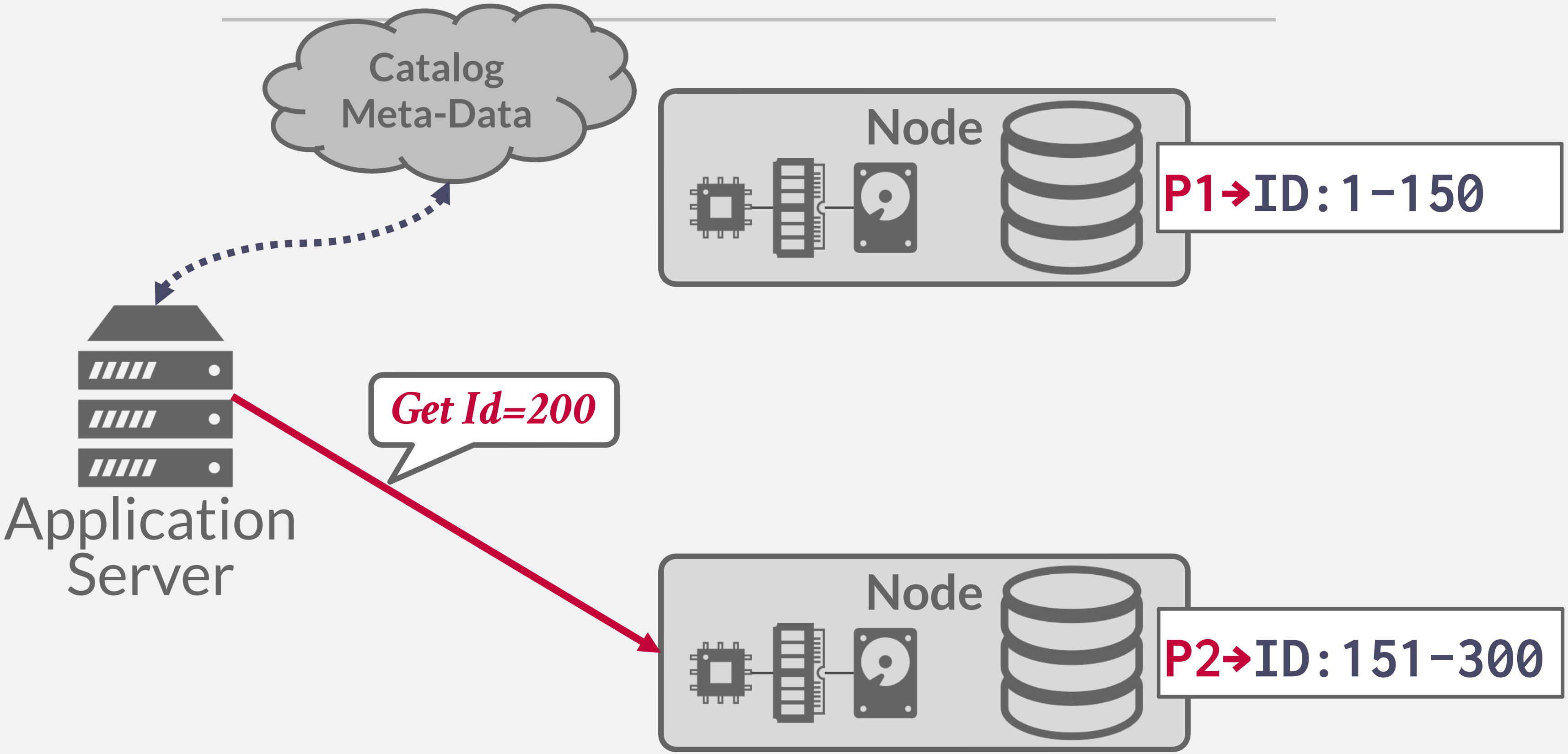
SHARED NOTHING EXAMPLE



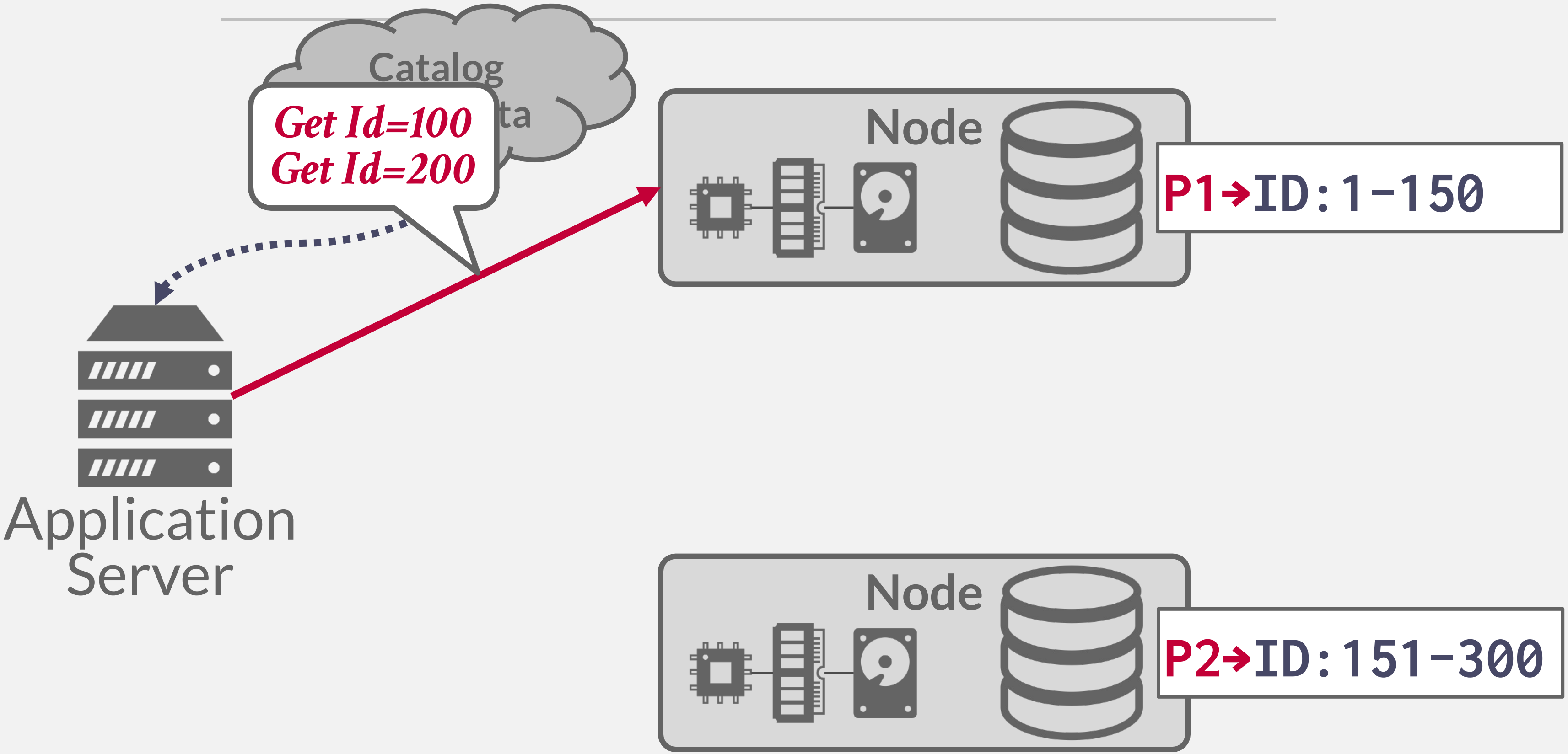
SHARED NOTHING EXAMPLE



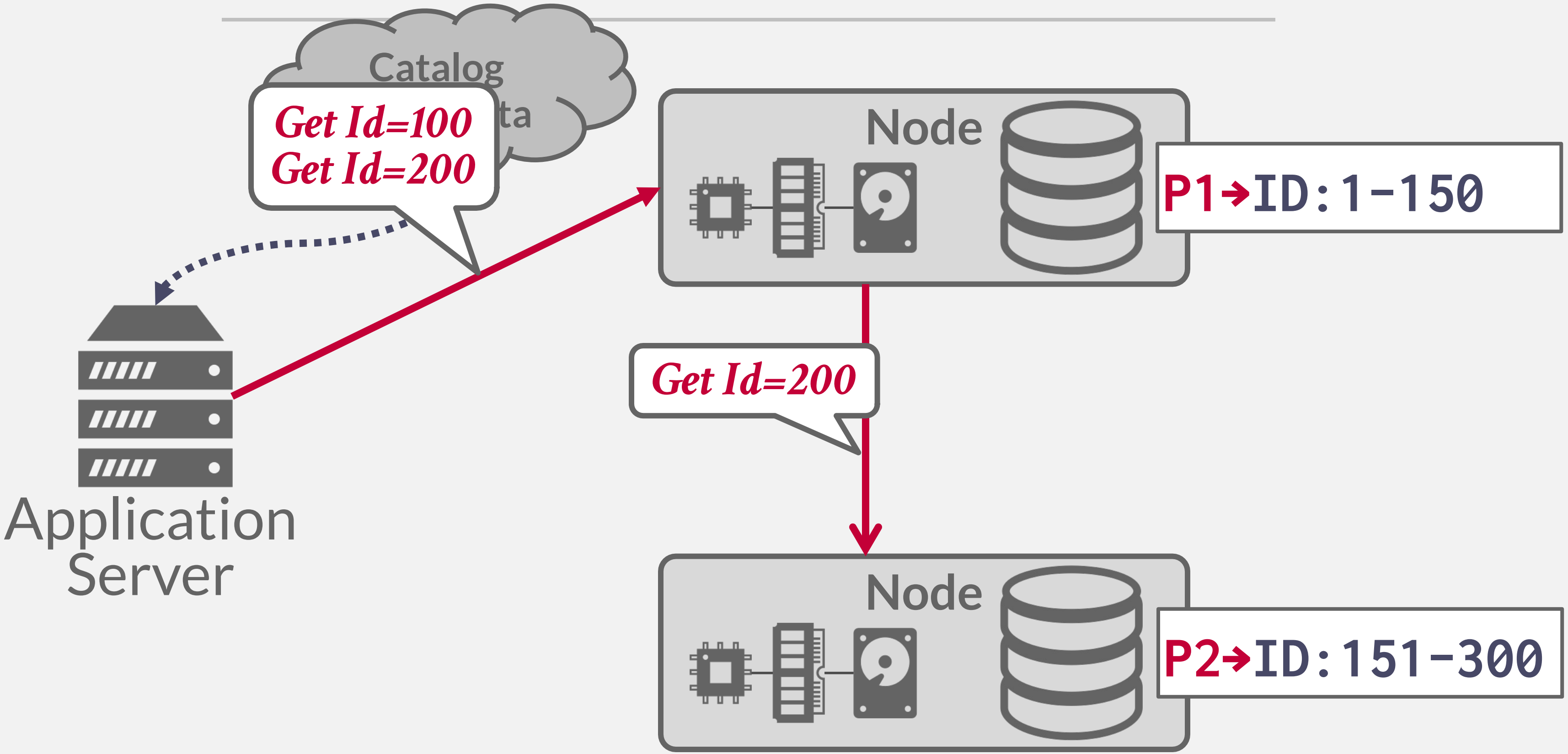
SHARED NOTHING EXAMPLE



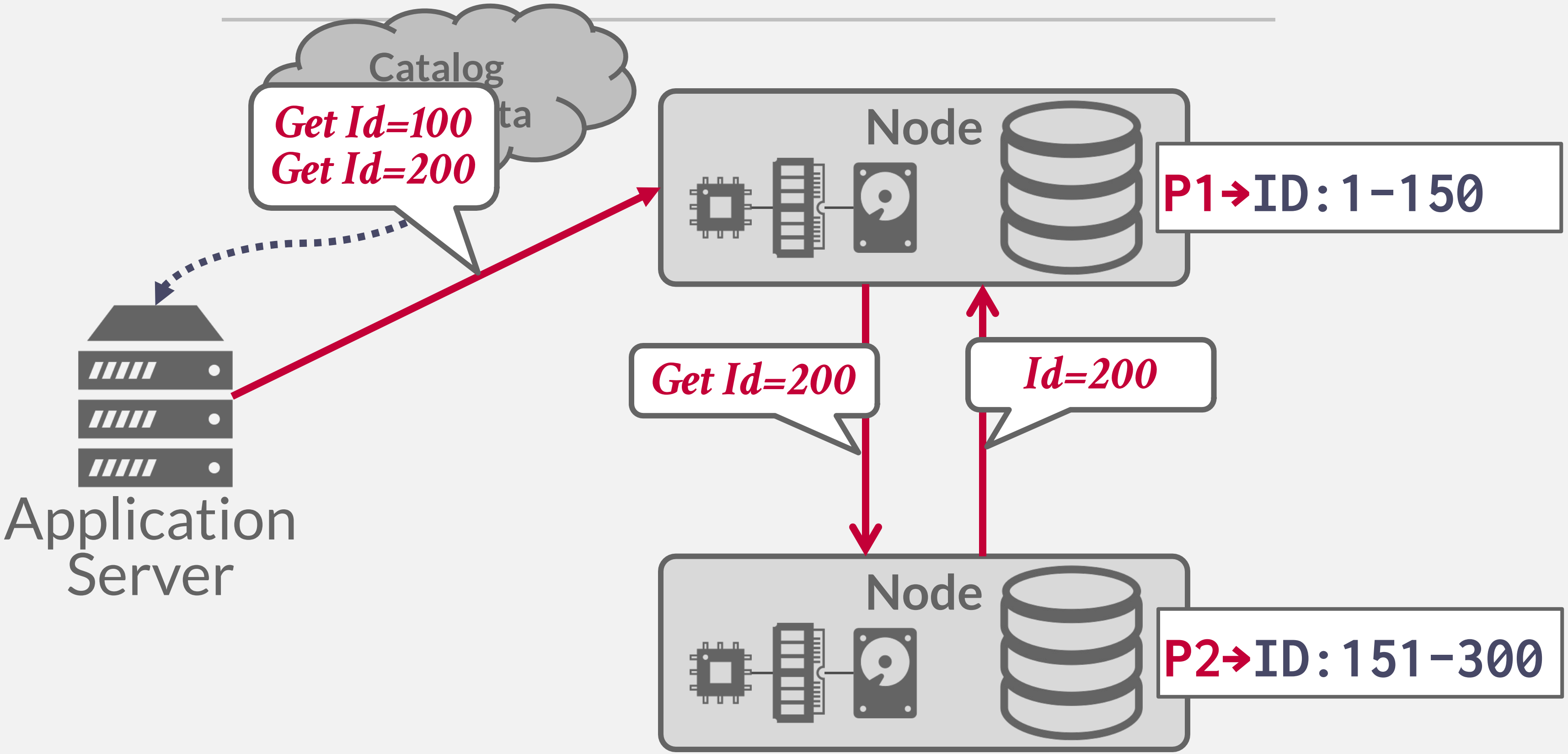
SHARED NOTHING EXAMPLE



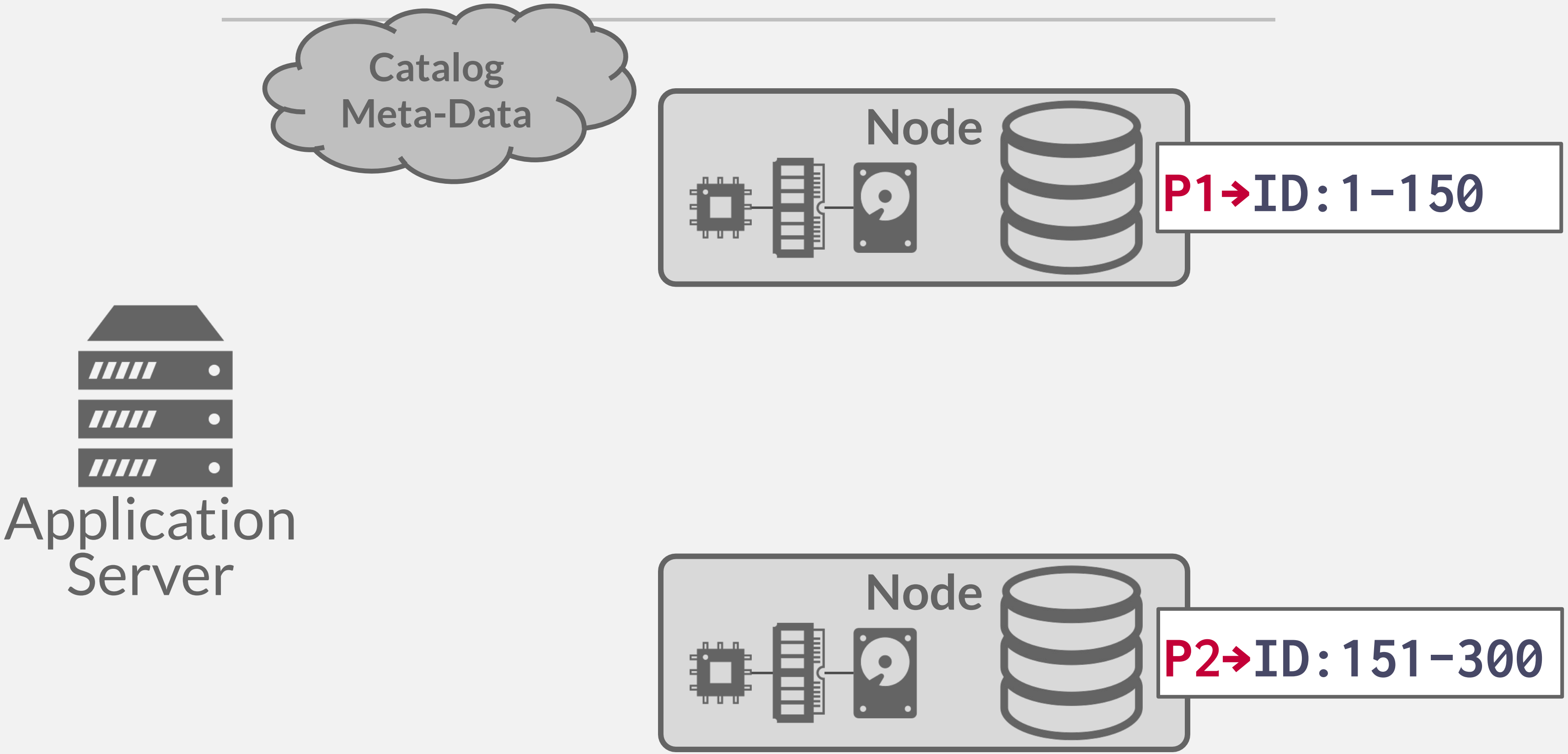
SHARED NOTHING EXAMPLE



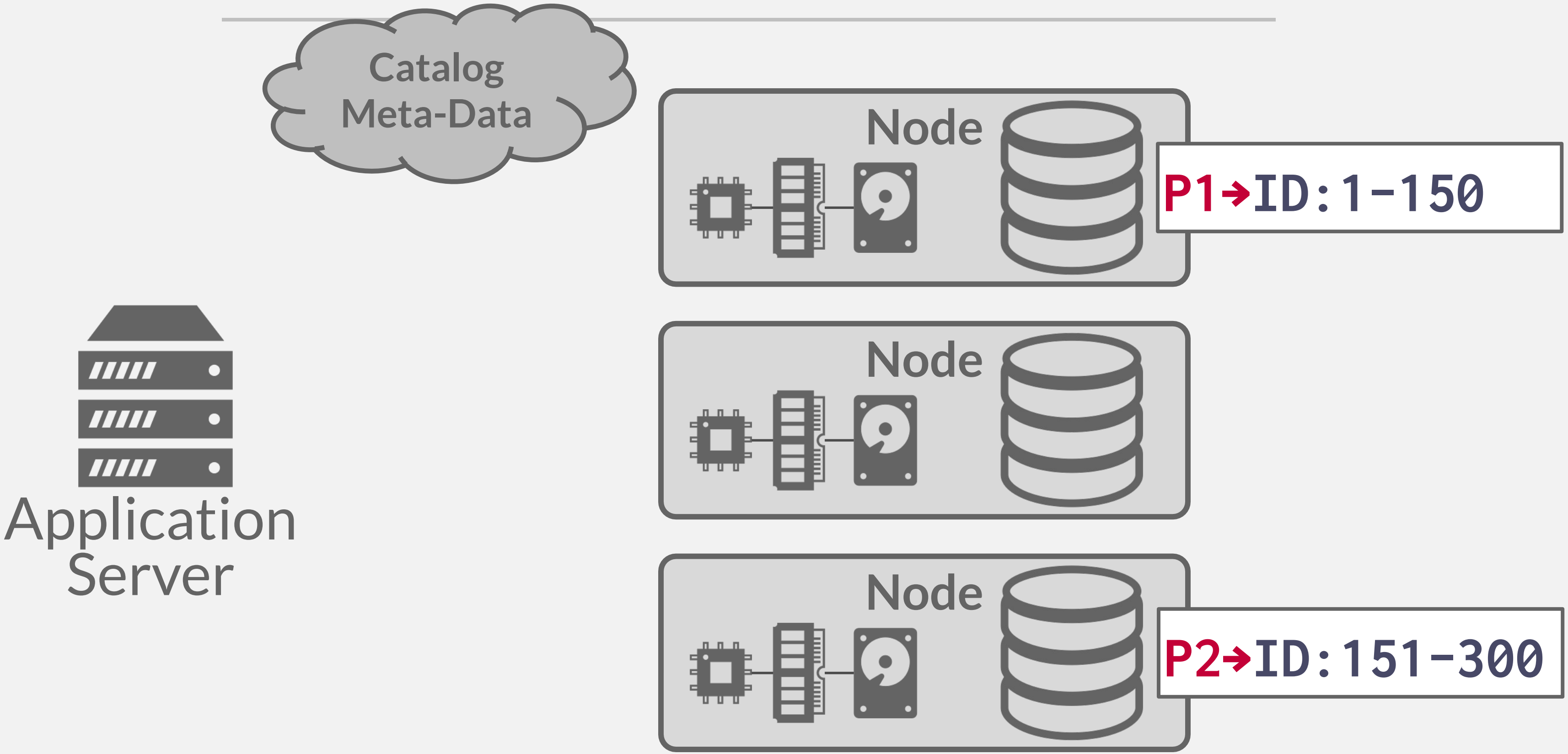
SHARED NOTHING EXAMPLE



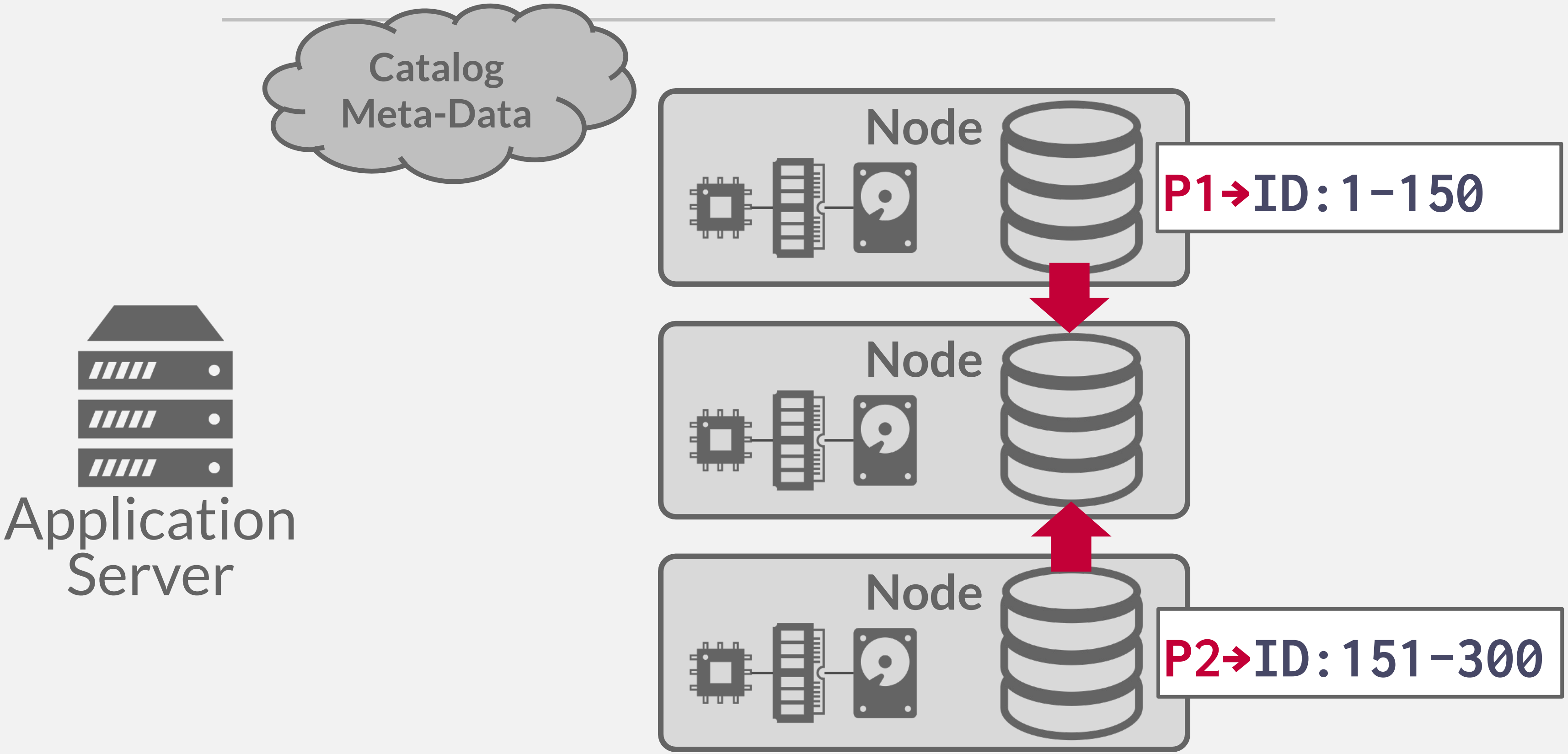
SHARED NOTHING EXAMPLE



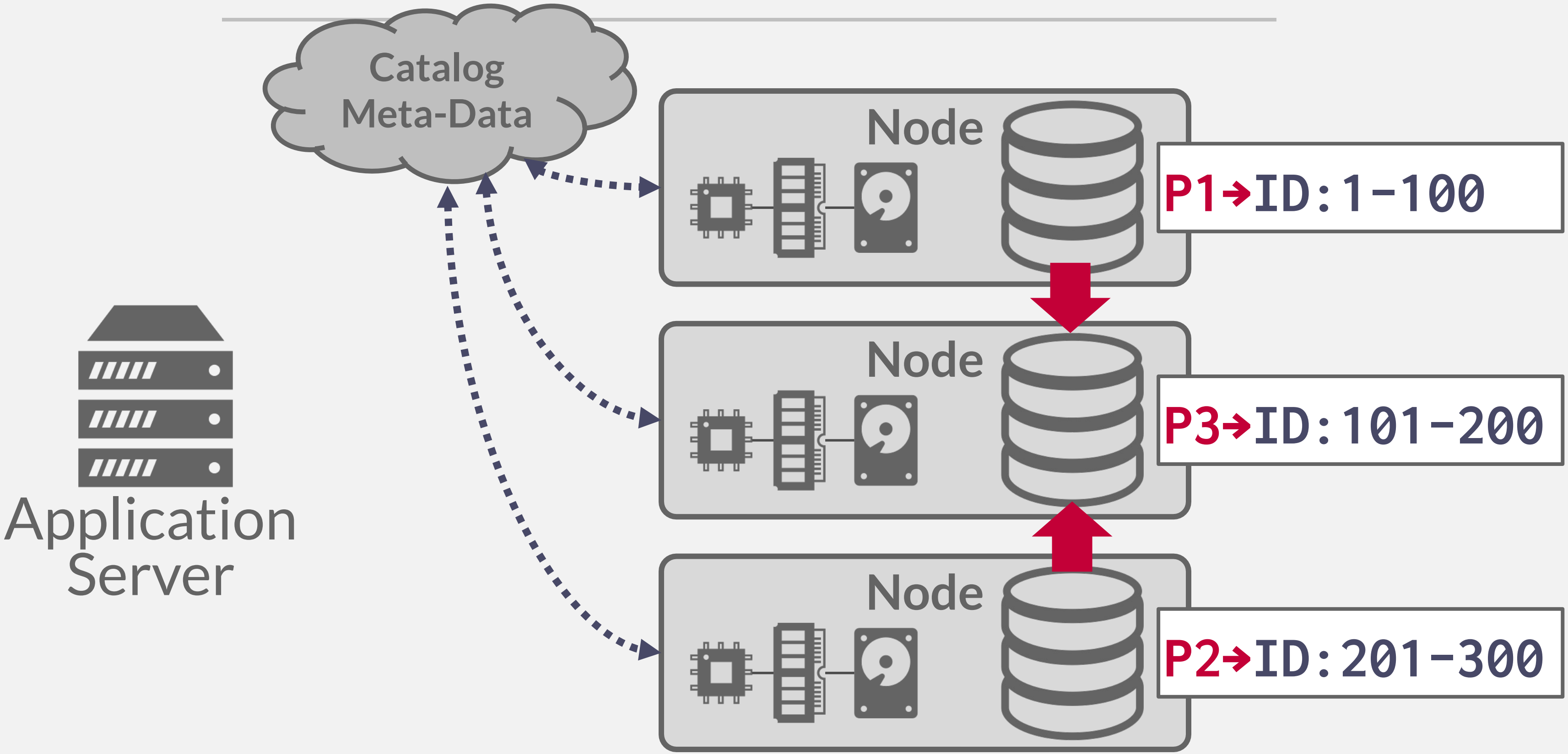
SHARED NOTHING EXAMPLE



SHARED NOTHING EXAMPLE



SHARED NOTHING EXAMPLE



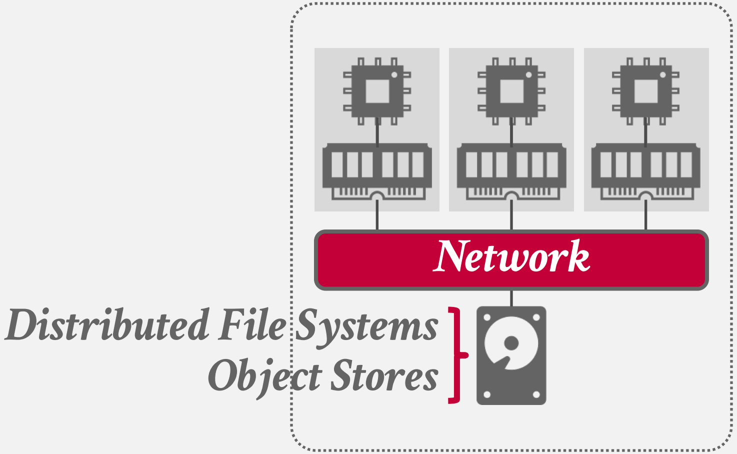
SHARED DISK

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

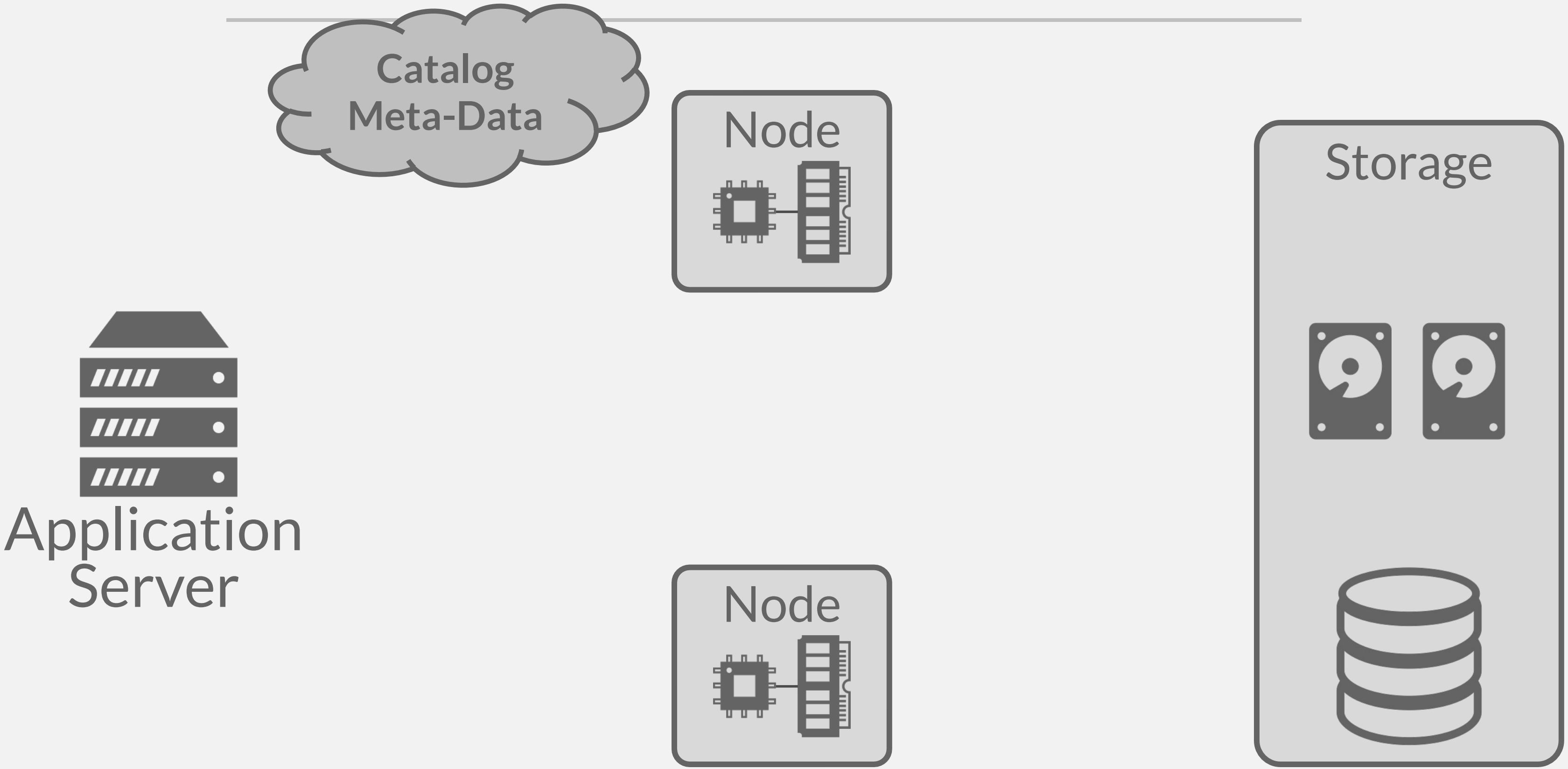
→ Scale execution layer independently from the storage layer.

→ Nodes can still use direct attached storage as a slower/larger cache.

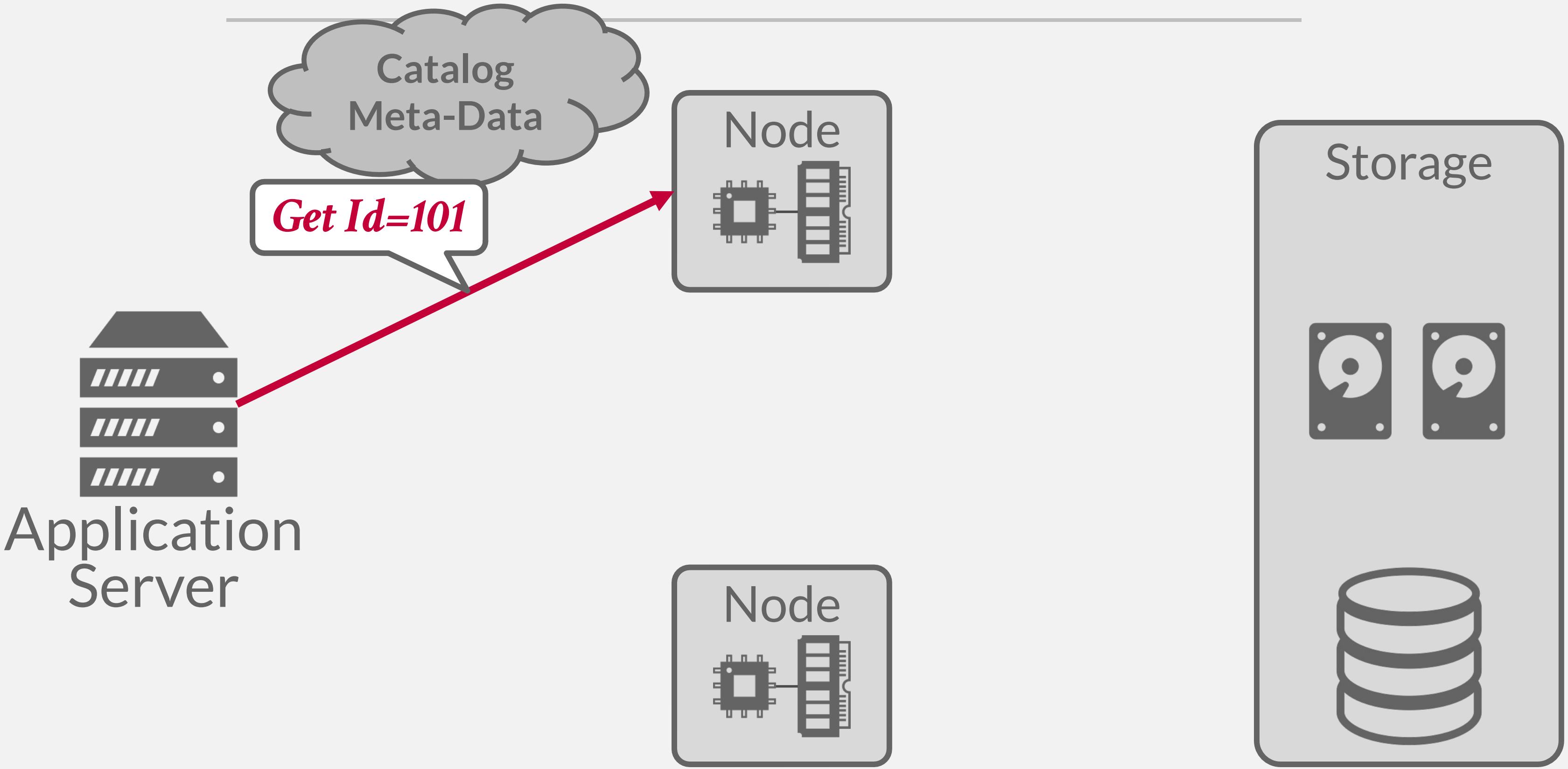
→ This architecture facilitates **data lakes** and **serverless** systems.



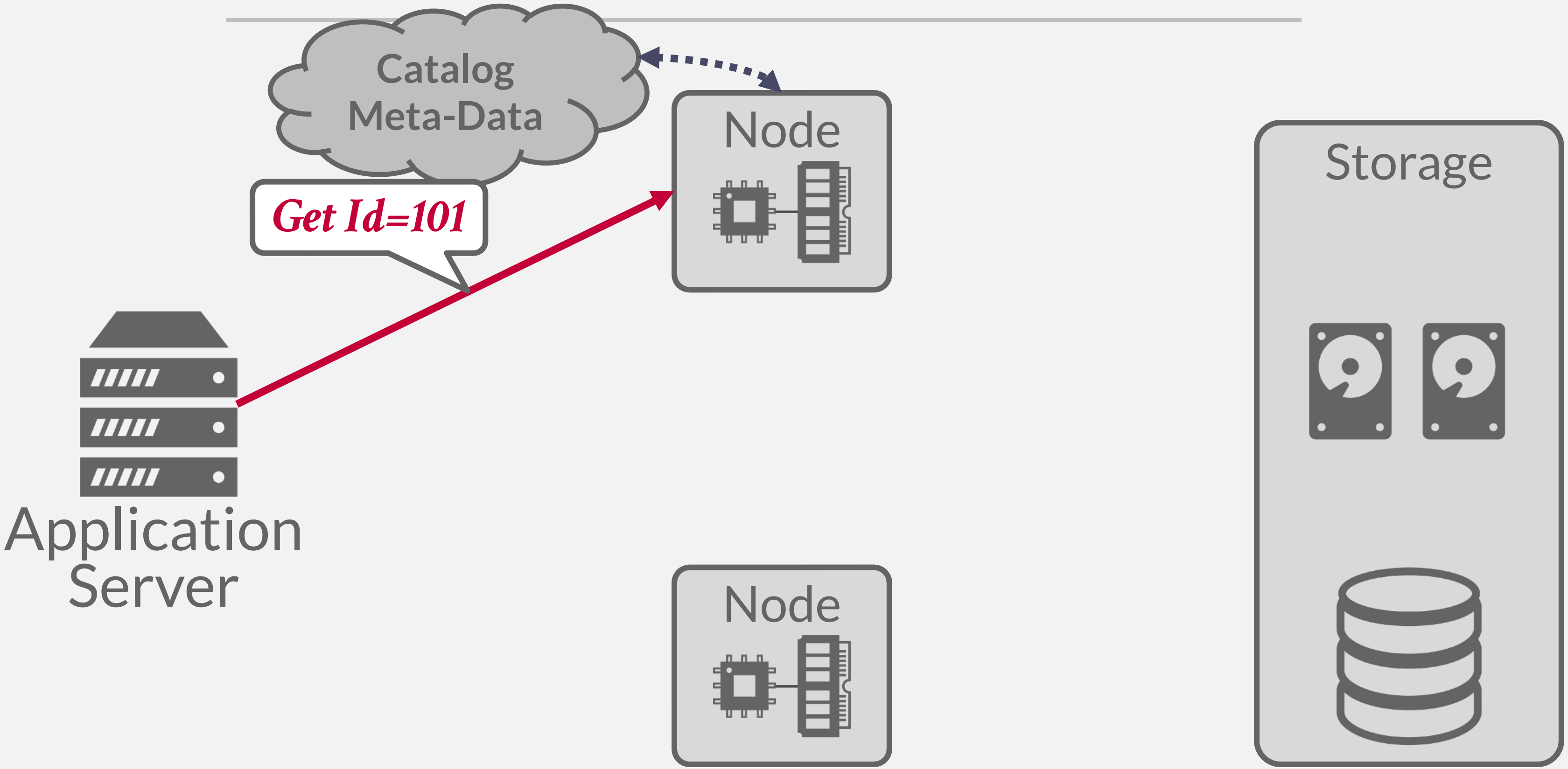
SHARED DISK EXAMPLE



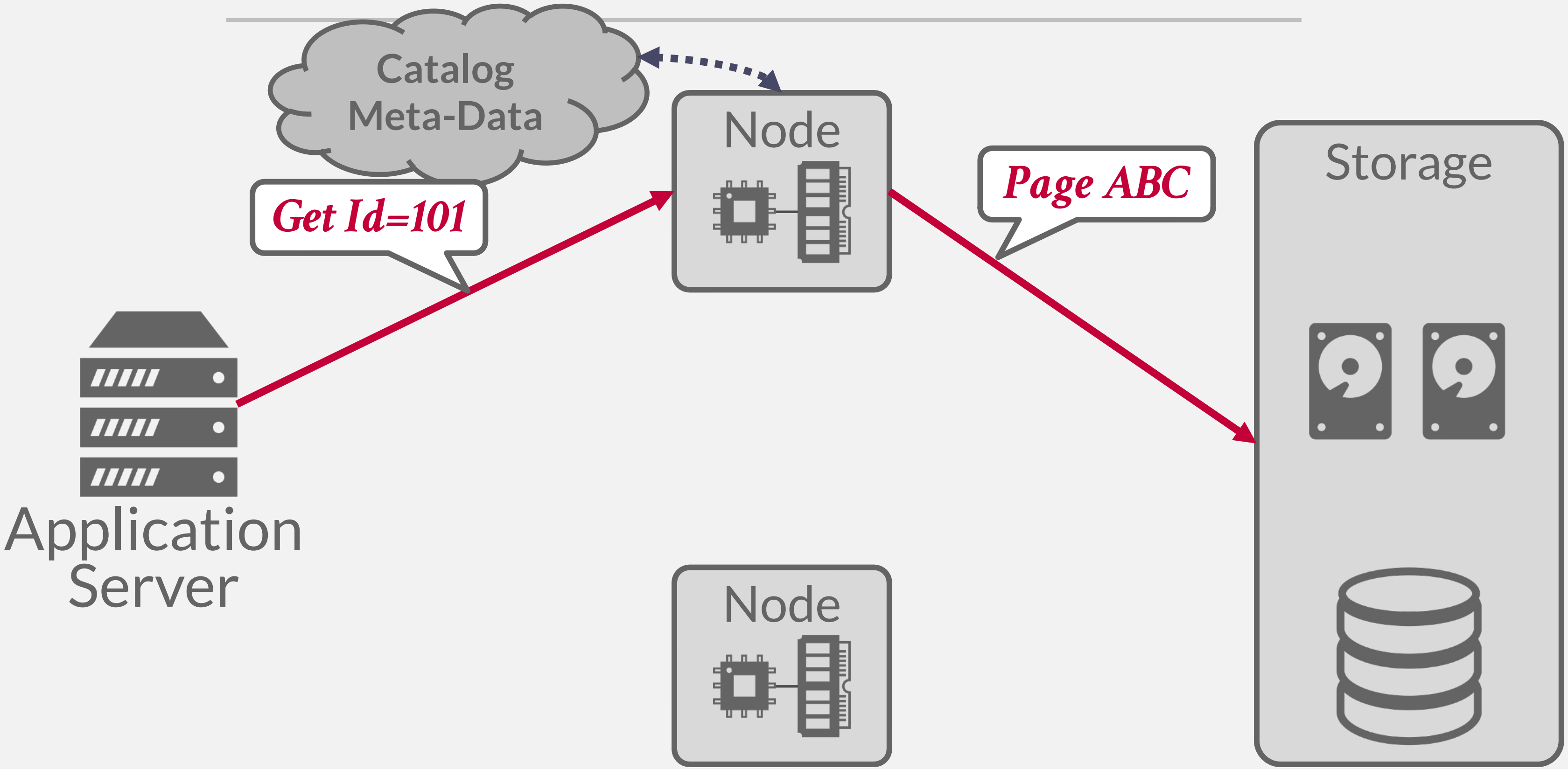
SHARED DISK EXAMPLE



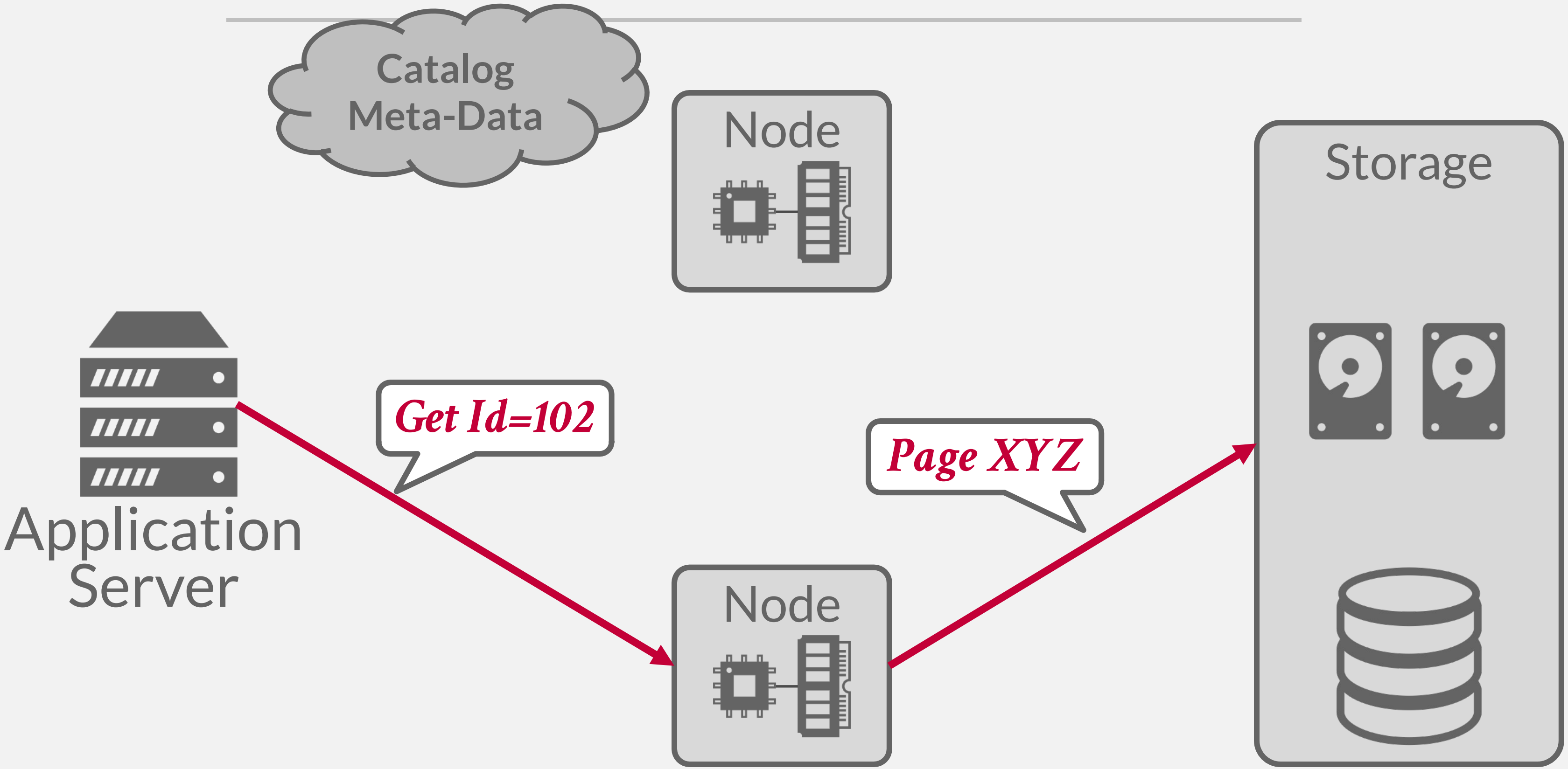
SHARED DISK EXAMPLE



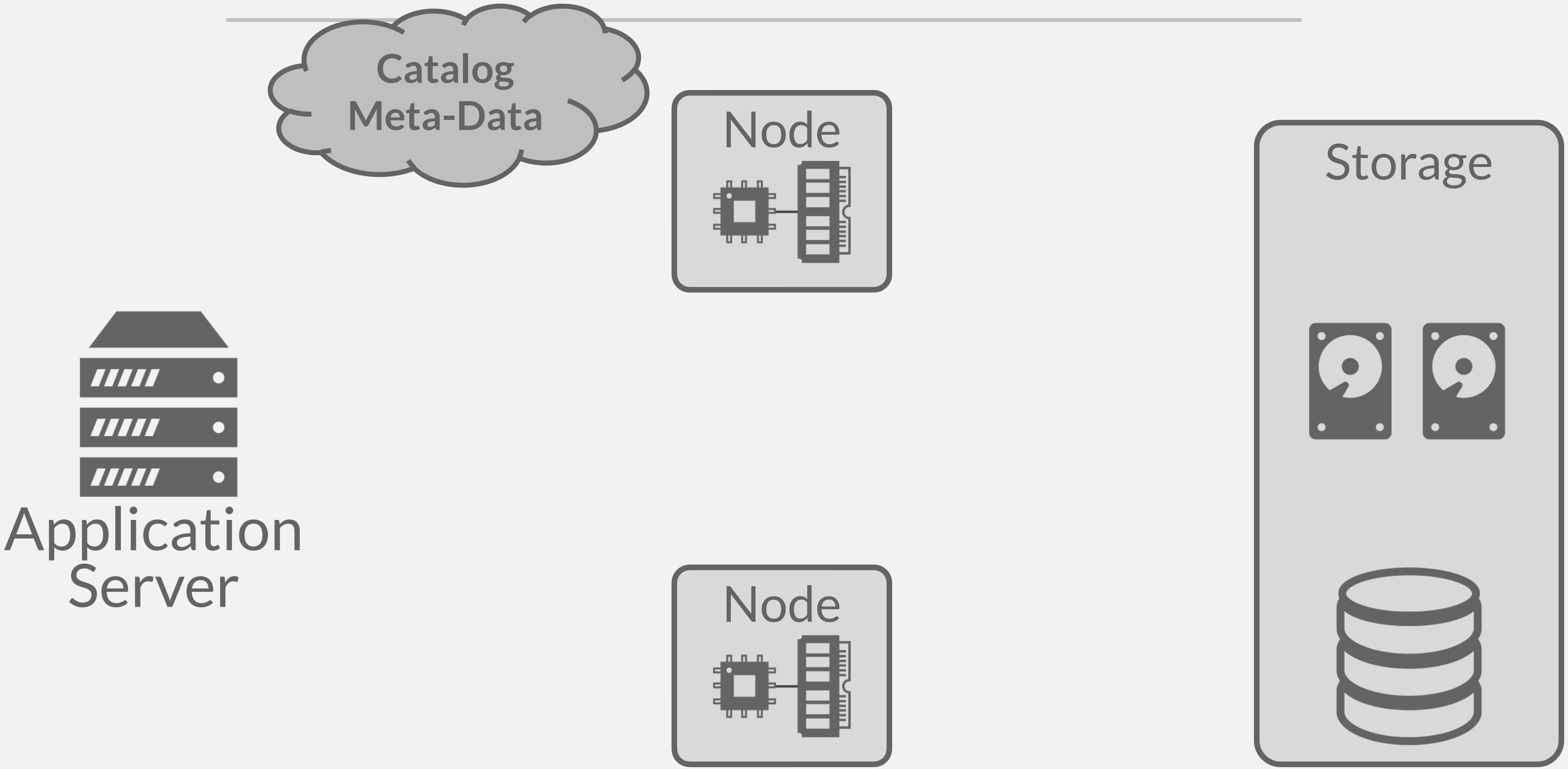
SHARED DISK EXAMPLE



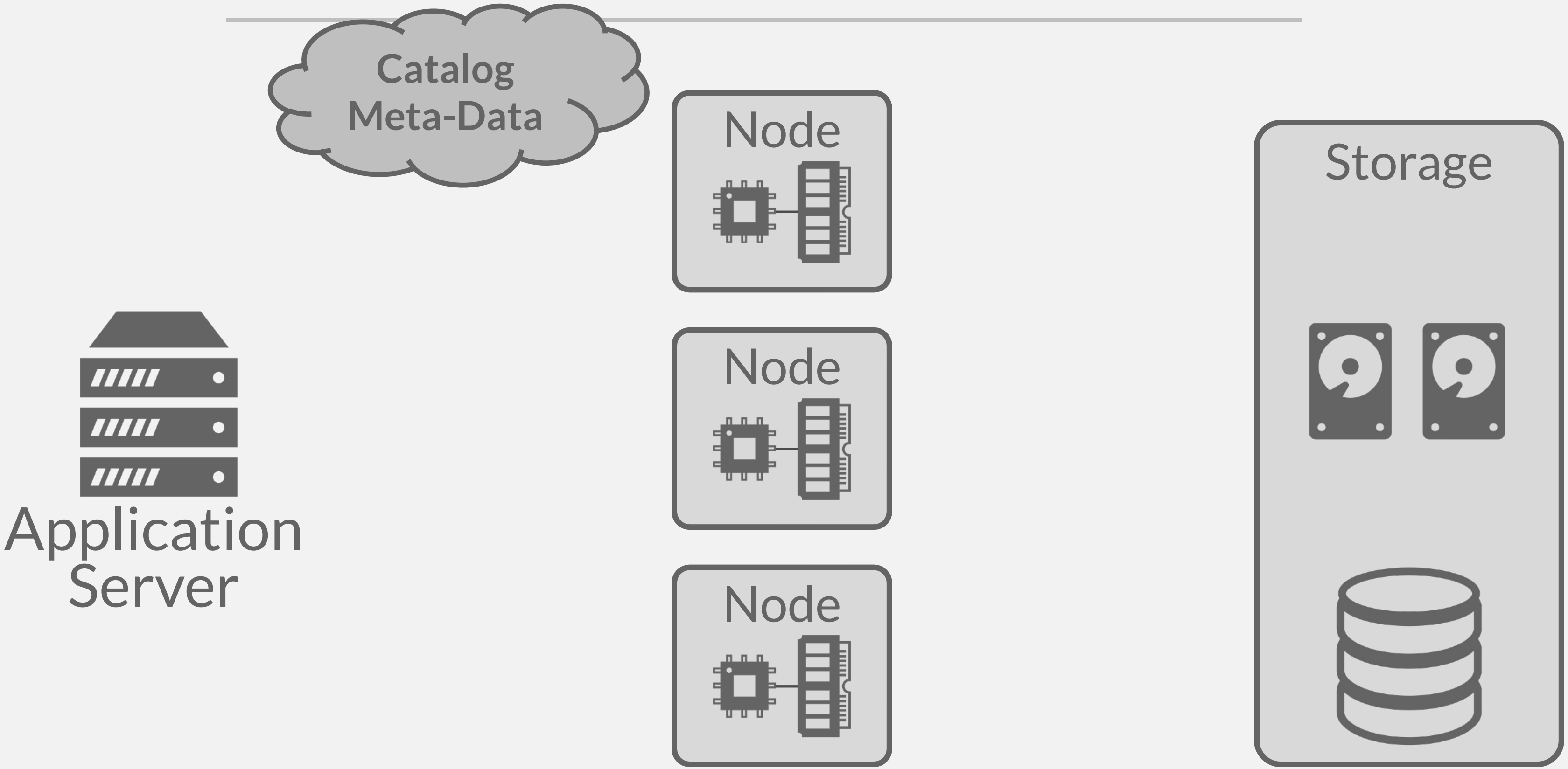
SHARED DISK EXAMPLE



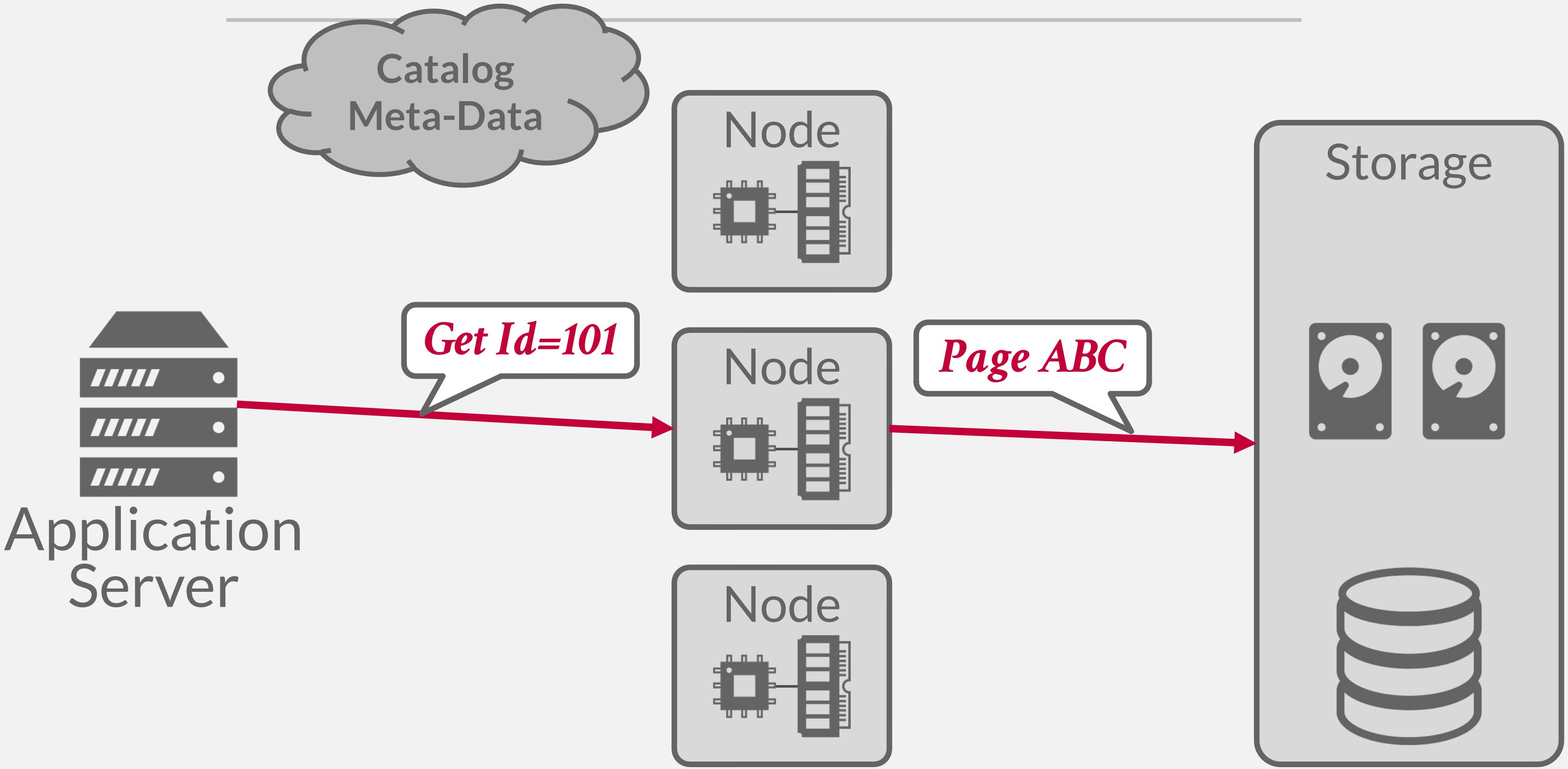
SHARED DISK EXAMPLE



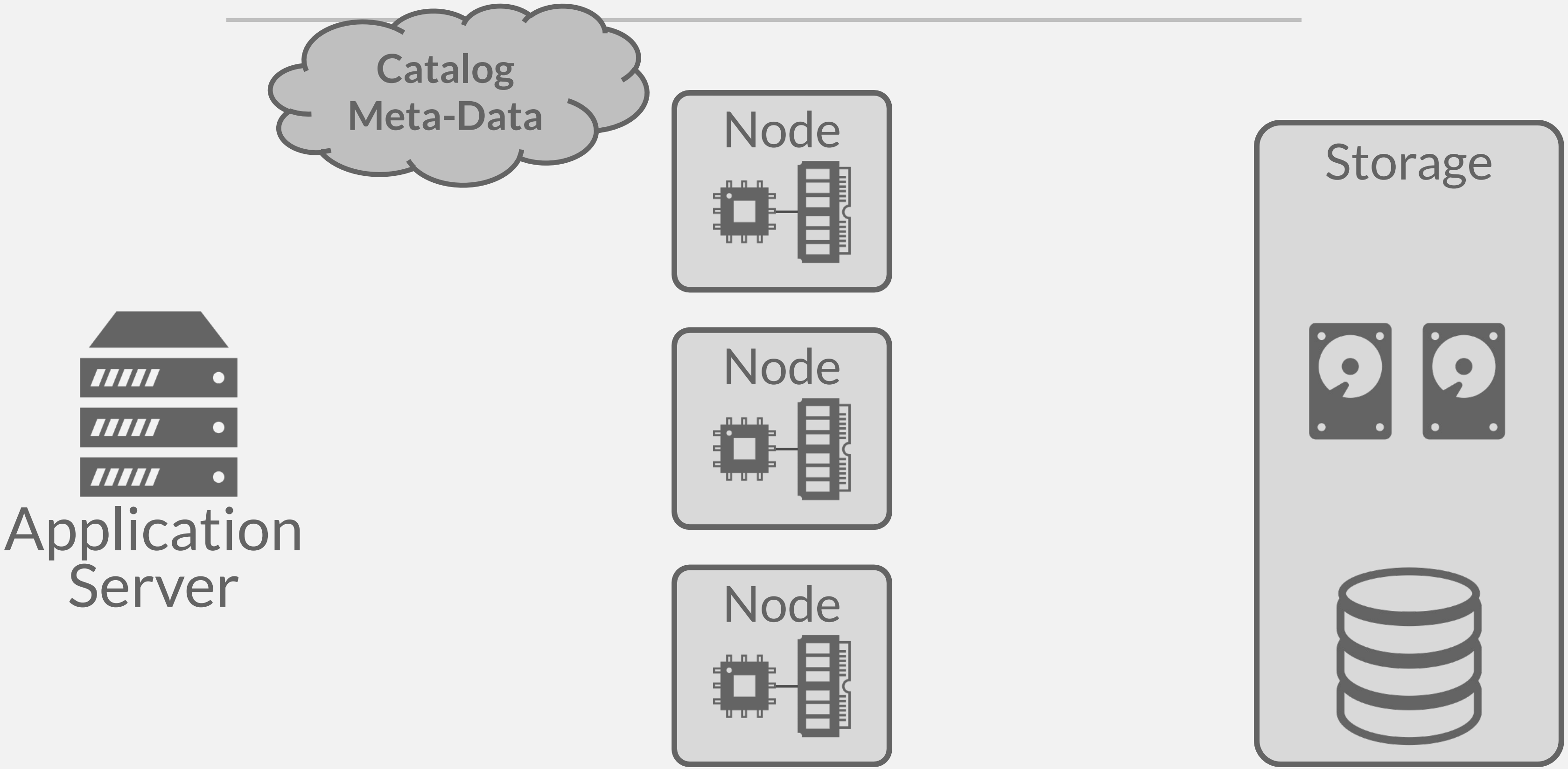
SHARED DISK EXAMPLE



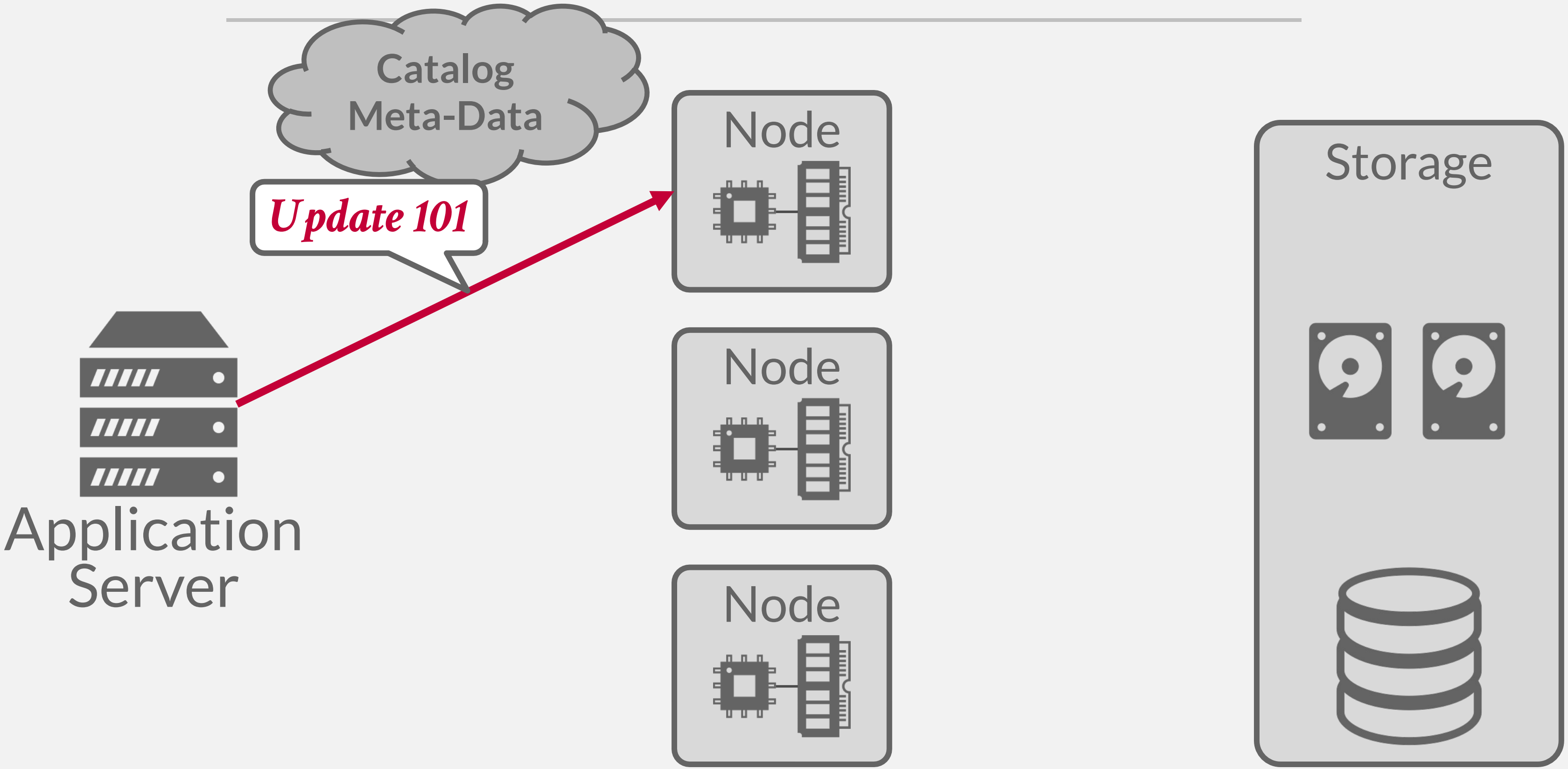
SHARED DISK EXAMPLE



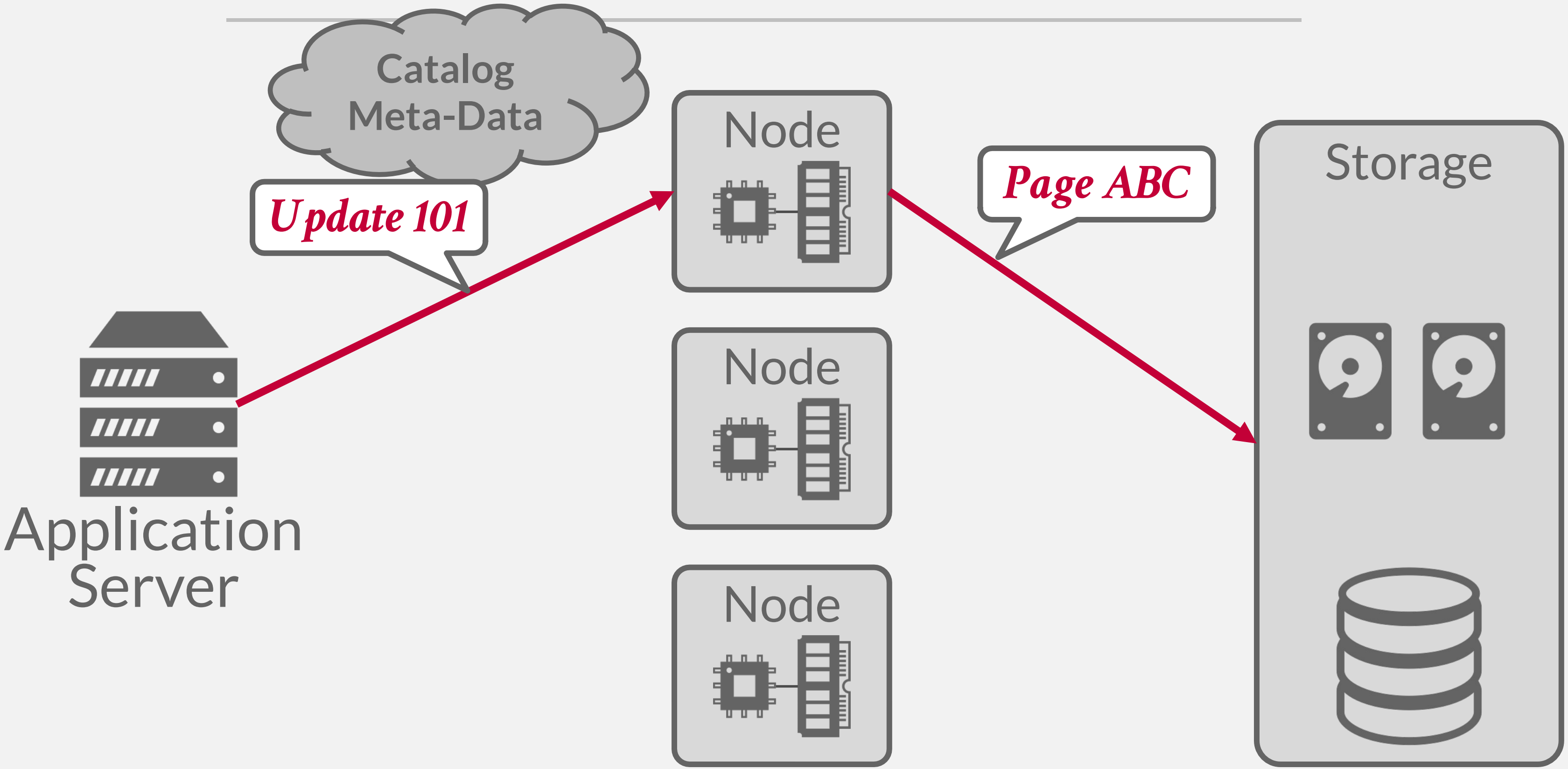
SHARED DISK EXAMPLE



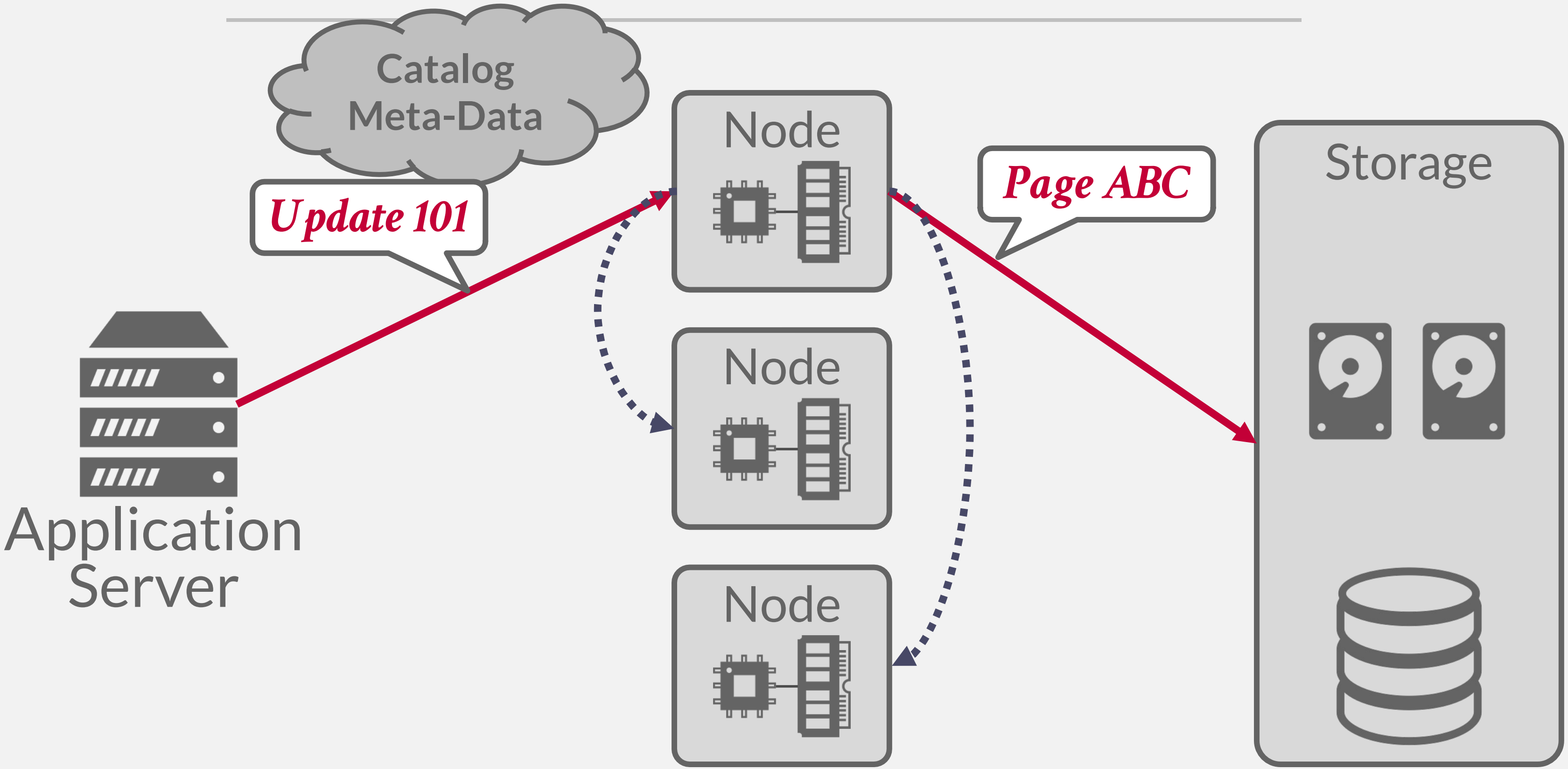
SHARED DISK EXAMPLE



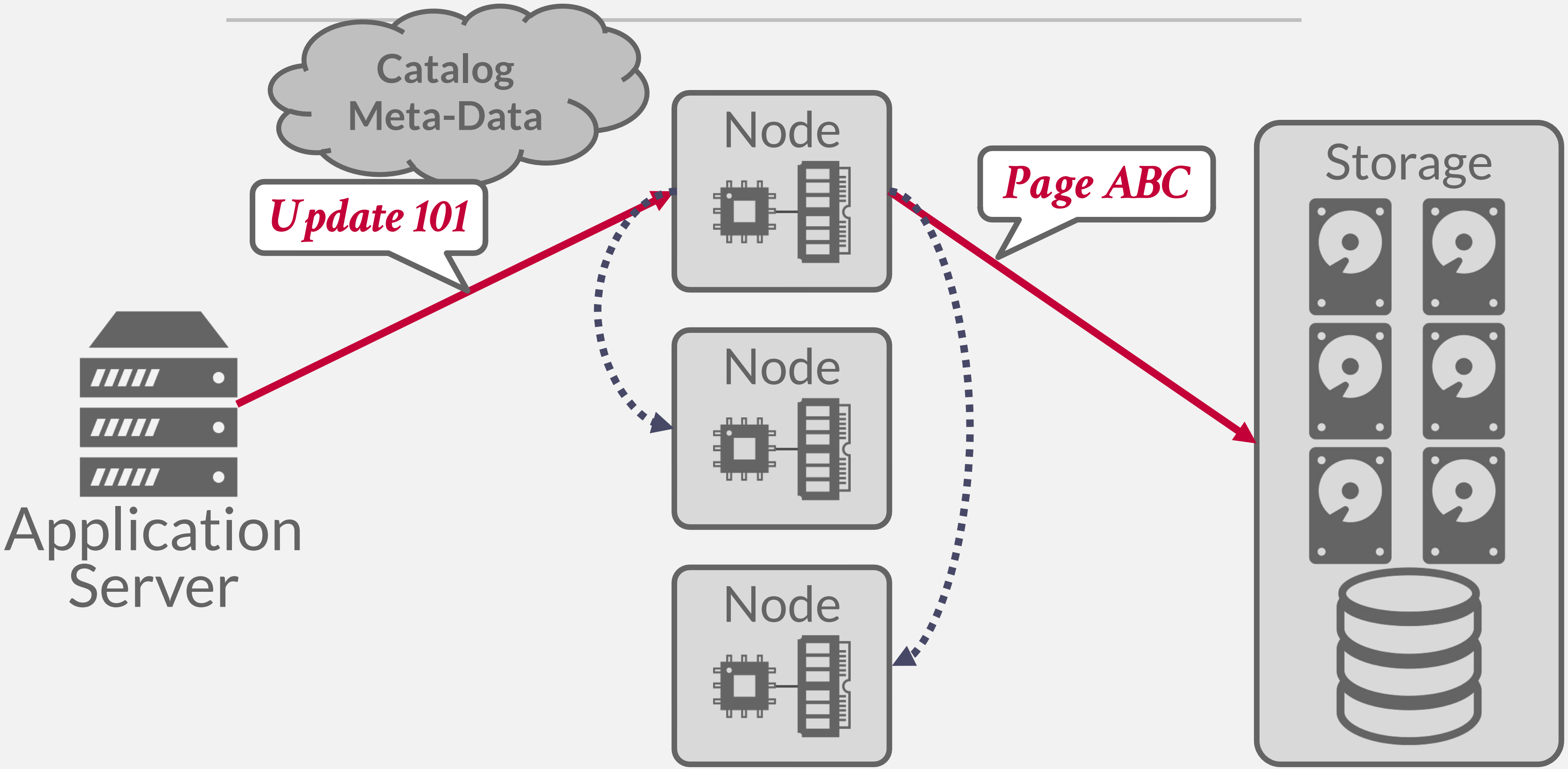
SHARED DISK EXAMPLE



SHARED DISK EXAMPLE



SHARED DISK EXAMPLE



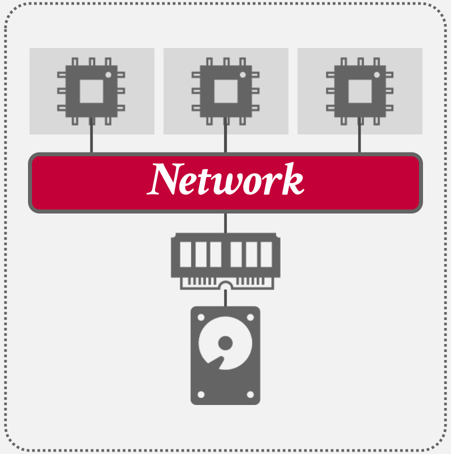
SHARED MEMORY

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

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

→ Can still use local memory / disk for intermediate results.

This looks a lot like shared-everything. Nobody does this.

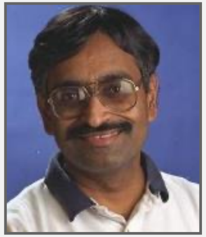


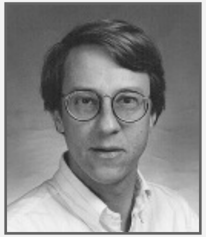
EARLY DISTRIBUTED DATABASE SYSTEMS

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.

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

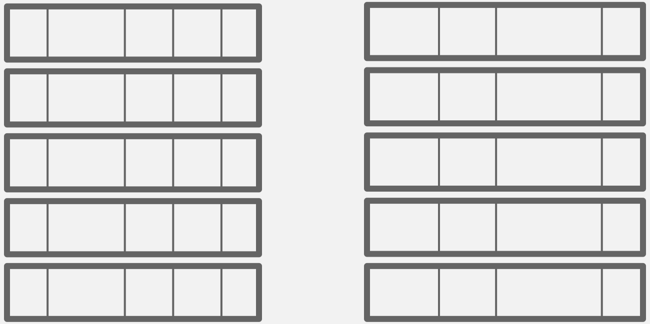
Assign an entire table to a single node.

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

Table1 Table2



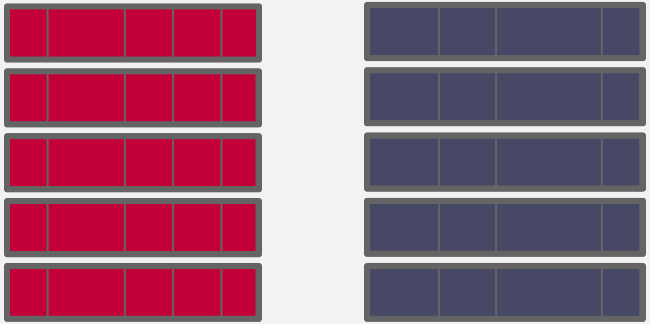
Partitions

Ideal Query:

```
SELECT * FROM table1
```

NAÏVE TABLE PARTITIONING

Table1 Table2

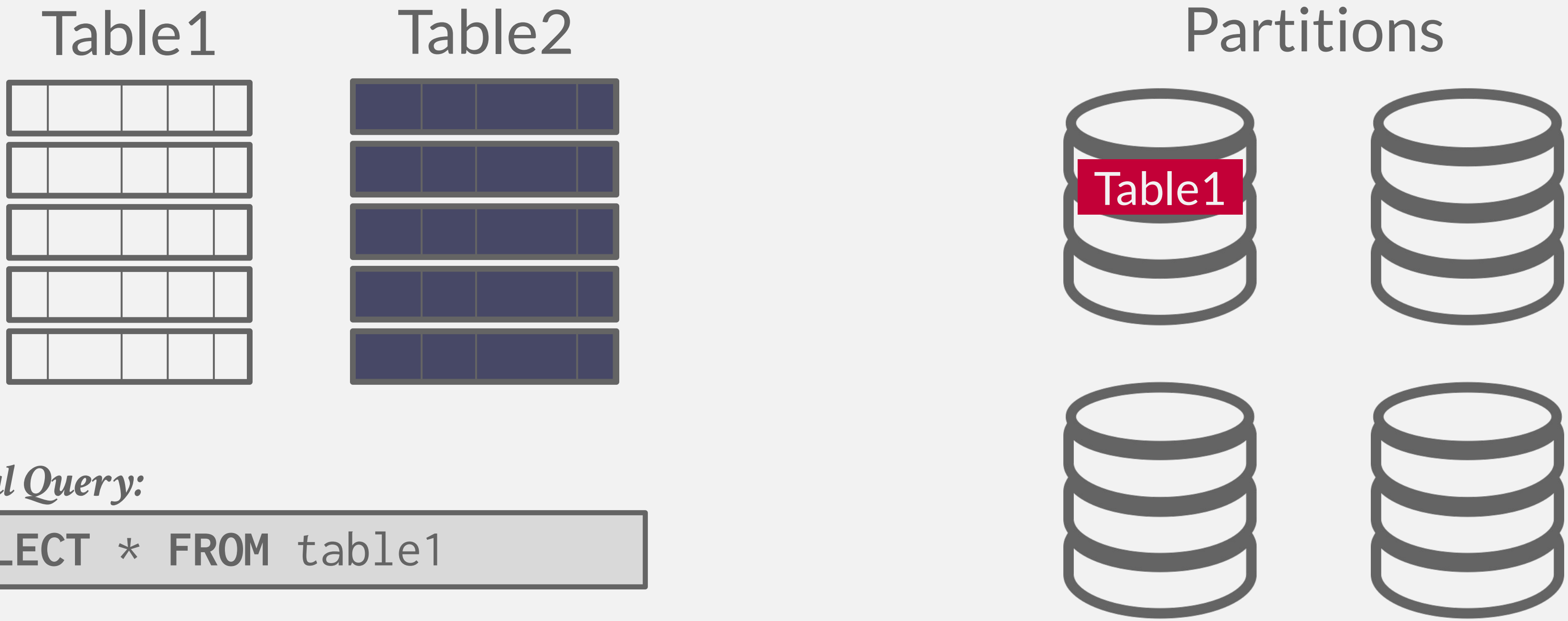


Partitions

Ideal Query:

```
SELECT * FROM table1
```

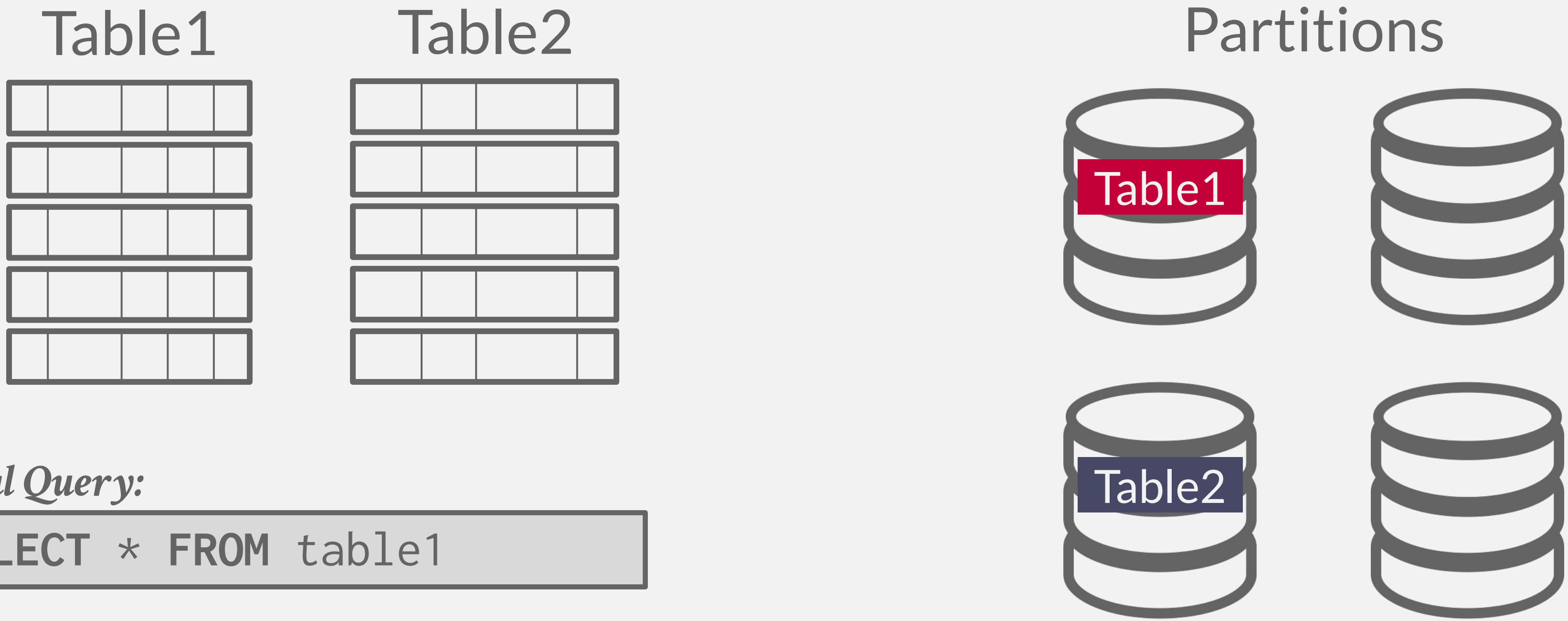
NAÏVE TABLE PARTITIONING



Ideal Query:

```
SELECT * FROM table1
```

NAÏVE TABLE PARTITIONING



Ideal Query:

```
SELECT * FROM table1
```

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
);
```

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

Table

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

Ideal Query:

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

Partitions

HORIZONTAL PARTITIONING

Partitioning Key

Table

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

Ideal Query:

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

Partitions

HORIZONTAL PARTITIONING

Partitioning Key

Table

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 = ?
```

HORIZONTAL PARTITIONING

Partitioning Key

Table

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 = ?
```

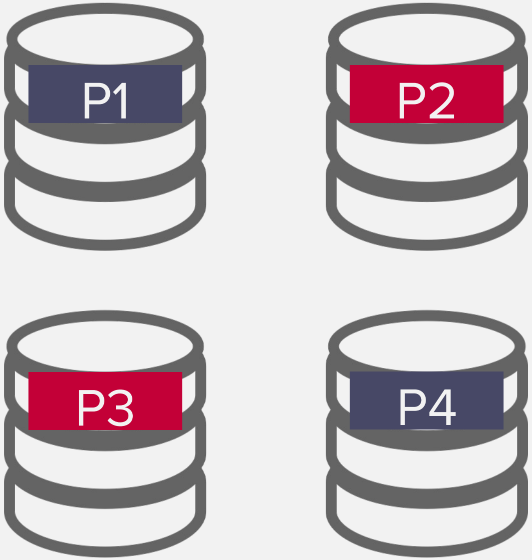
HORIZONTAL PARTITIONING

Partitioning Key

Table

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 = ?
```

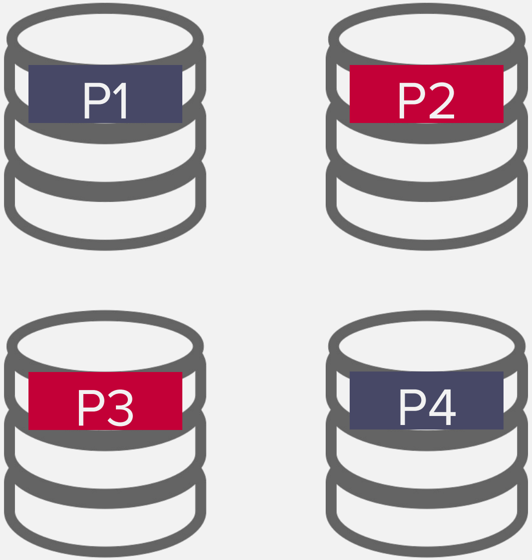
HORIZONTAL PARTITIONING

Partitioning Key

Table

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*

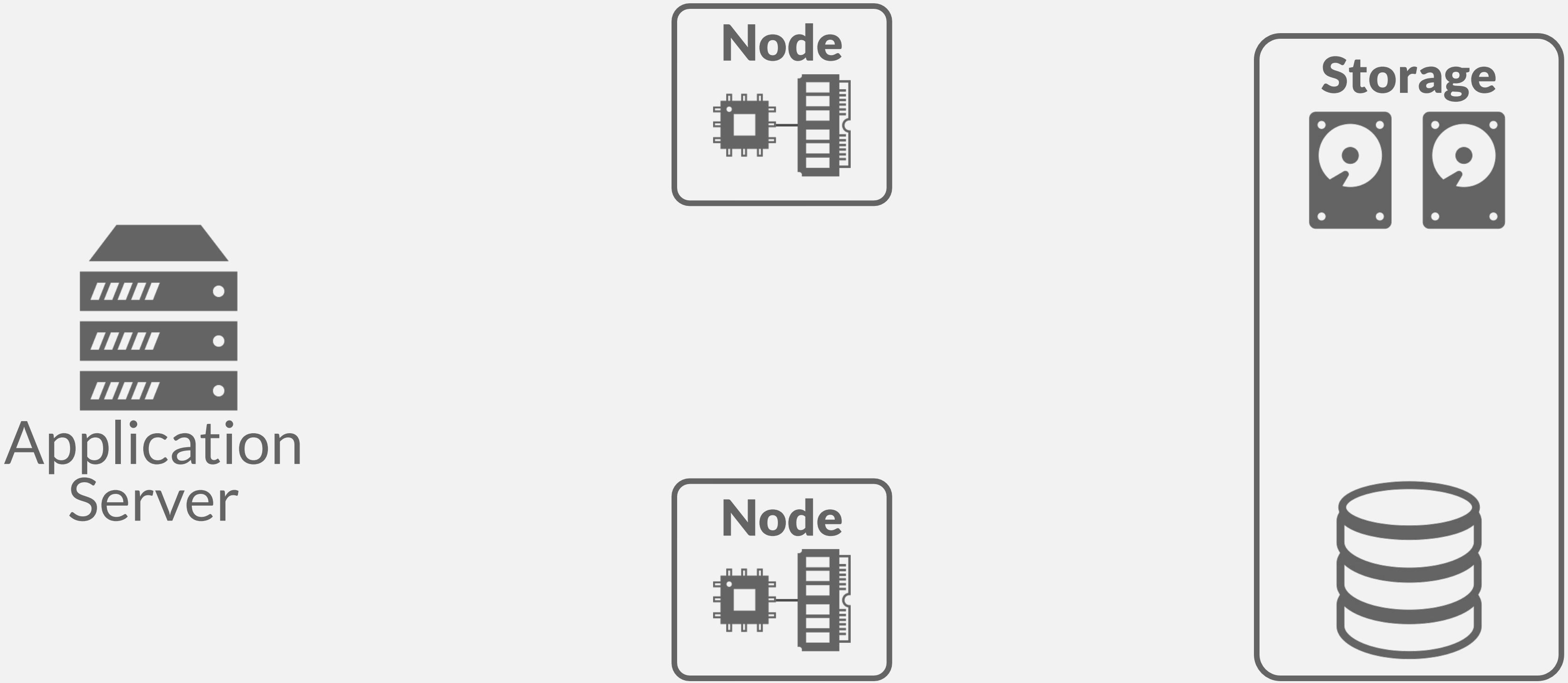
Partitions



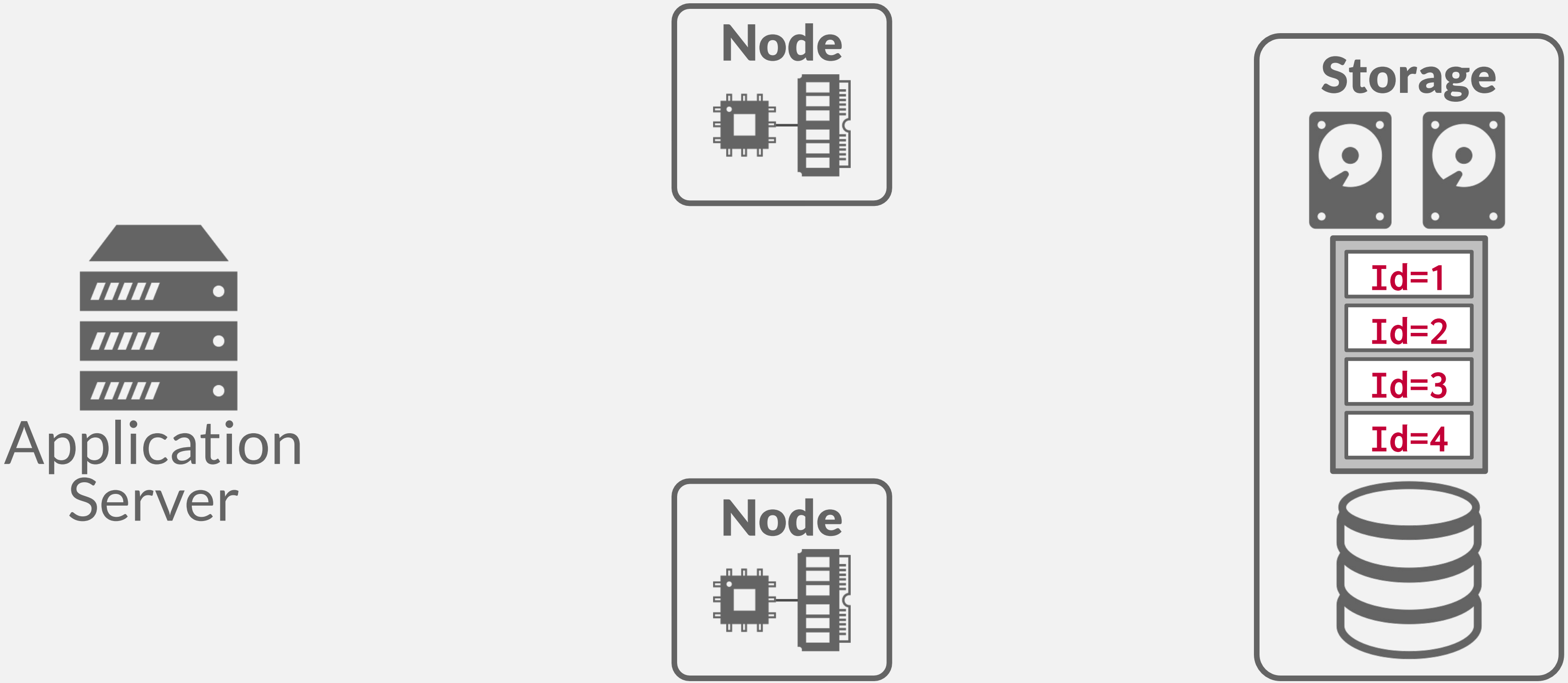
Ideal Query:

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

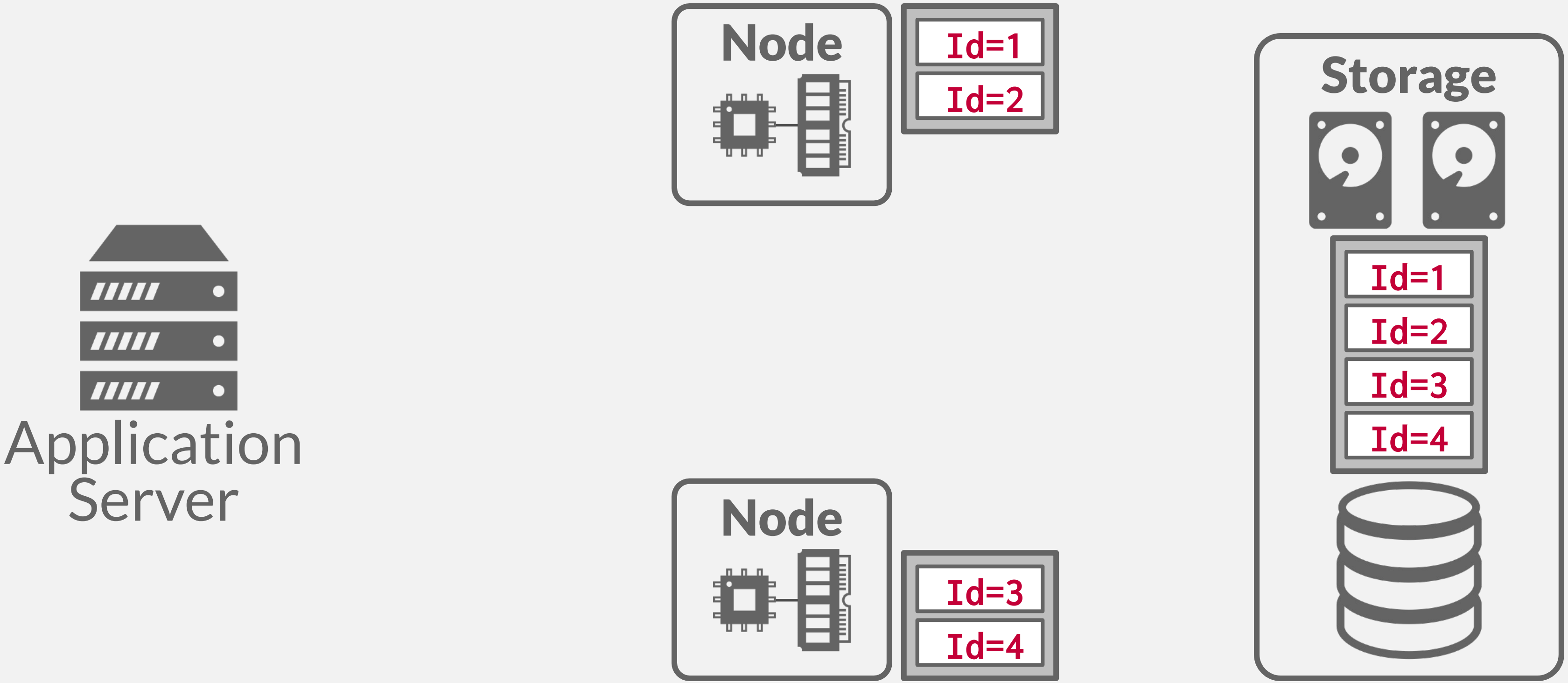
LOGICAL PARTITIONING



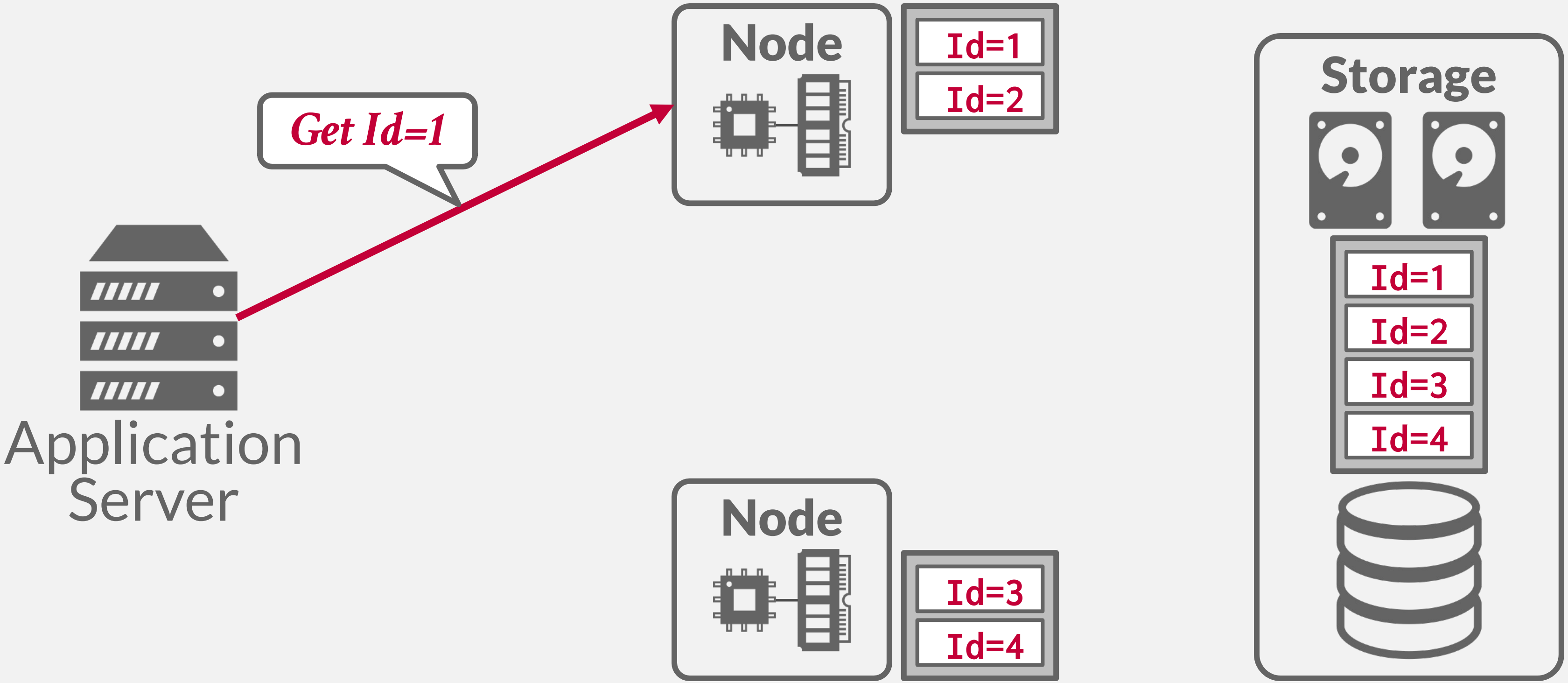
LOGICAL PARTITIONING



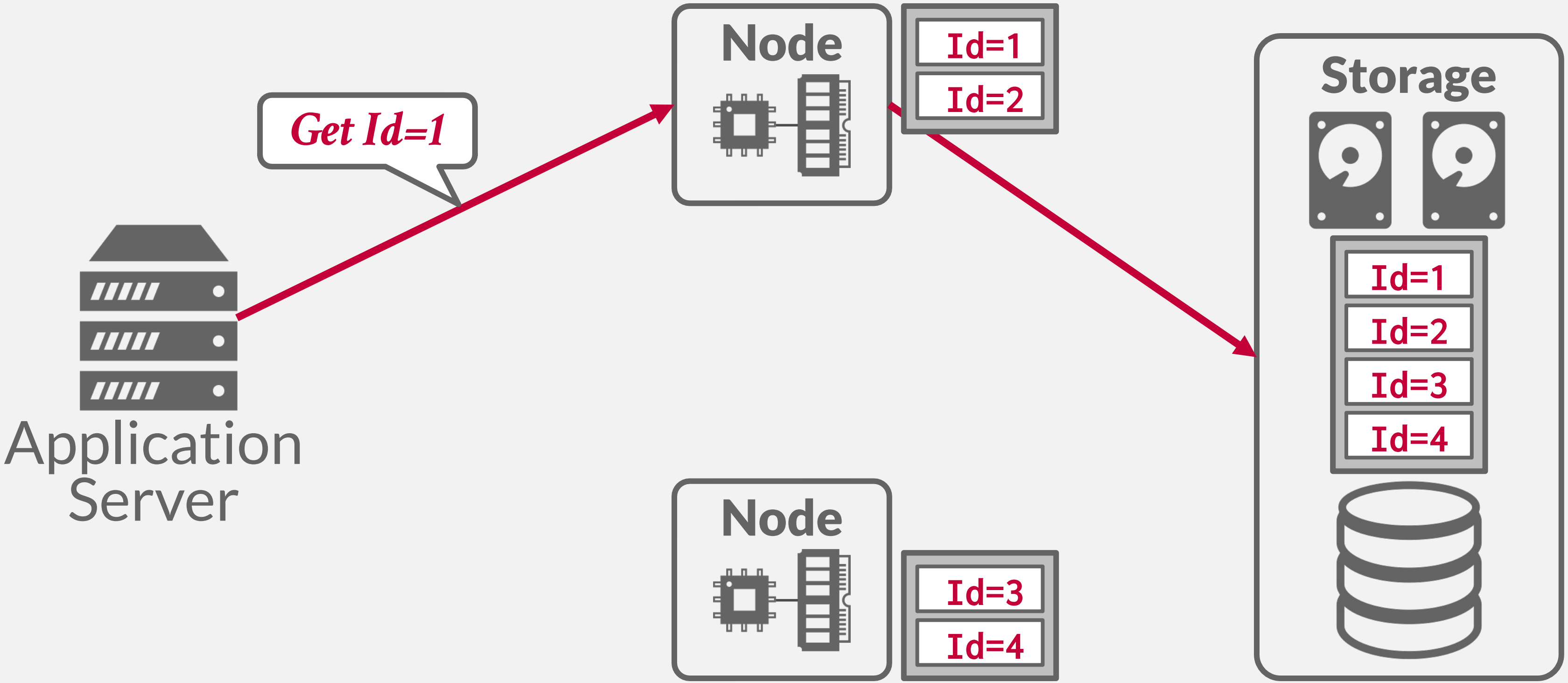
LOGICAL PARTITIONING



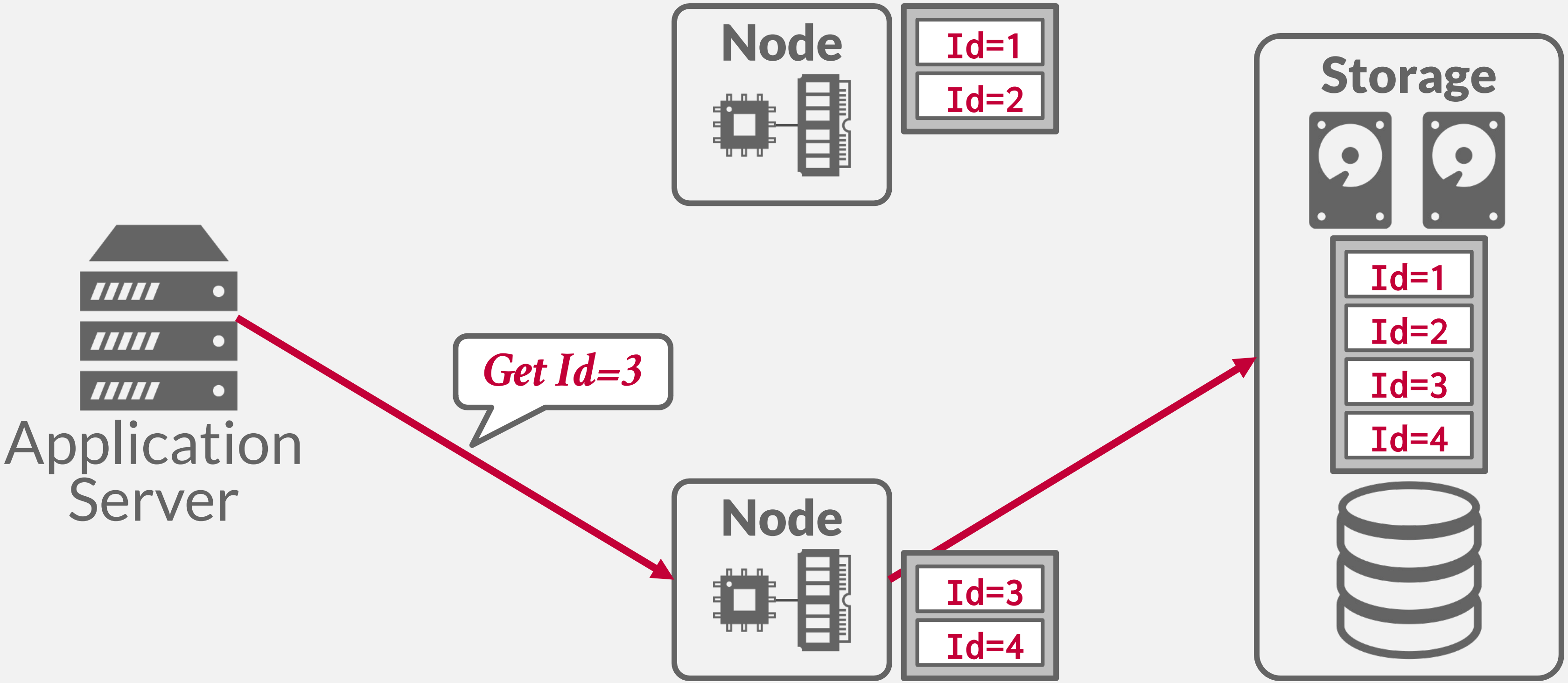
LOGICAL PARTITIONING



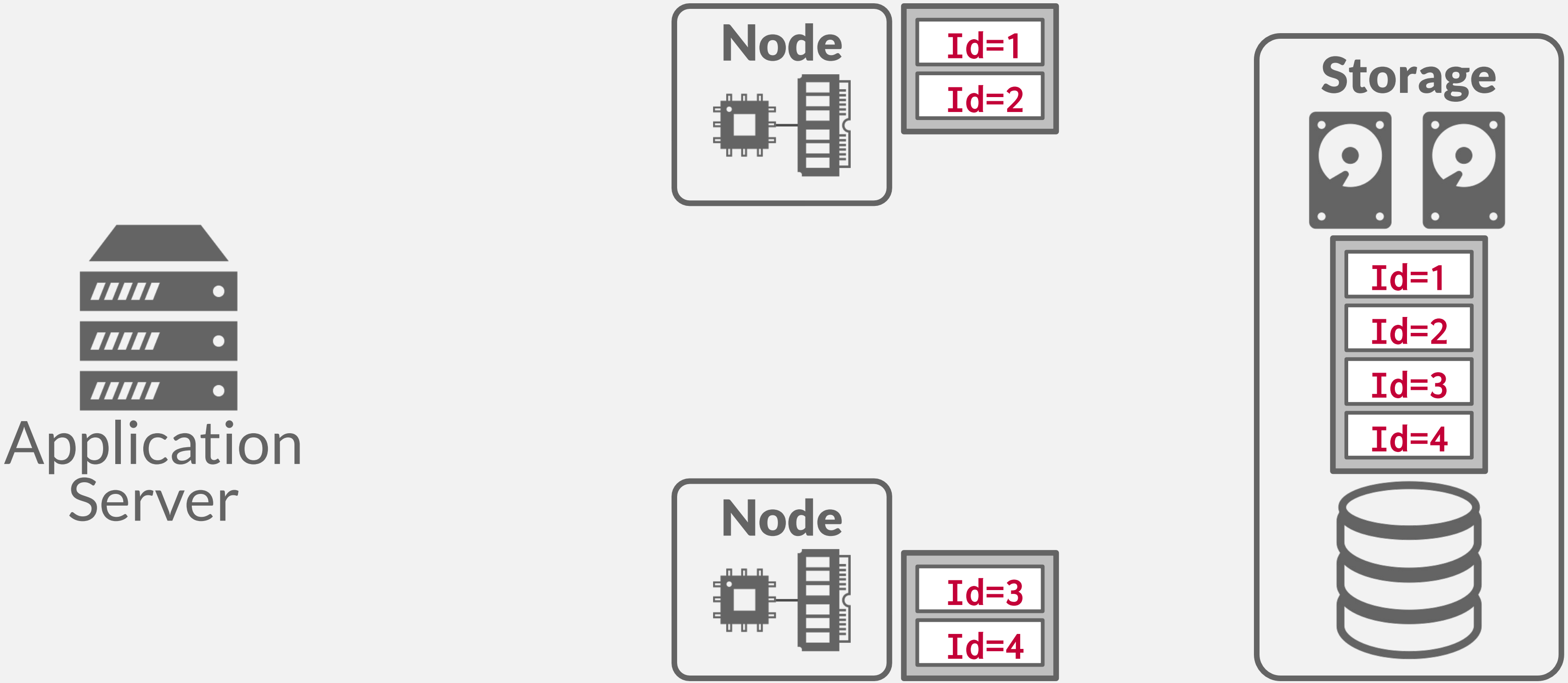
LOGICAL PARTITIONING



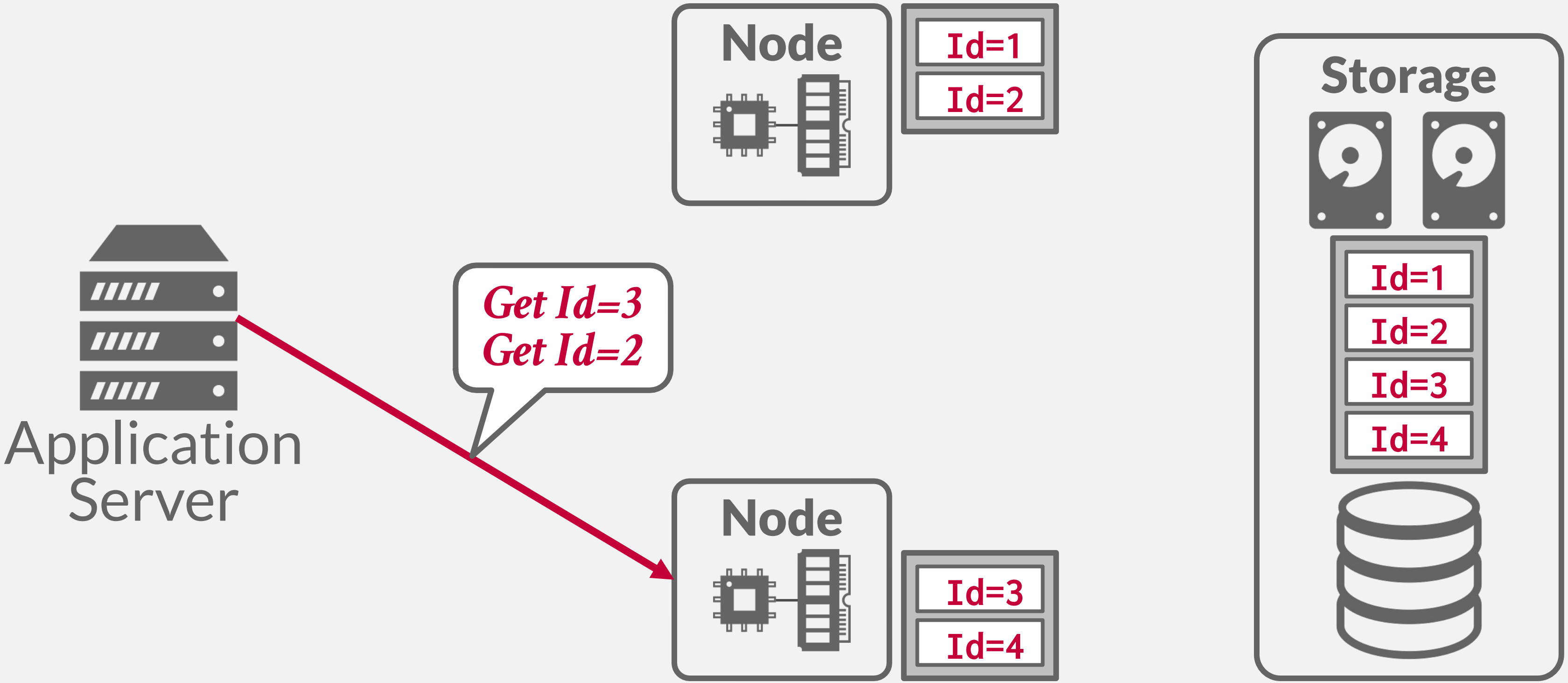
LOGICAL PARTITIONING



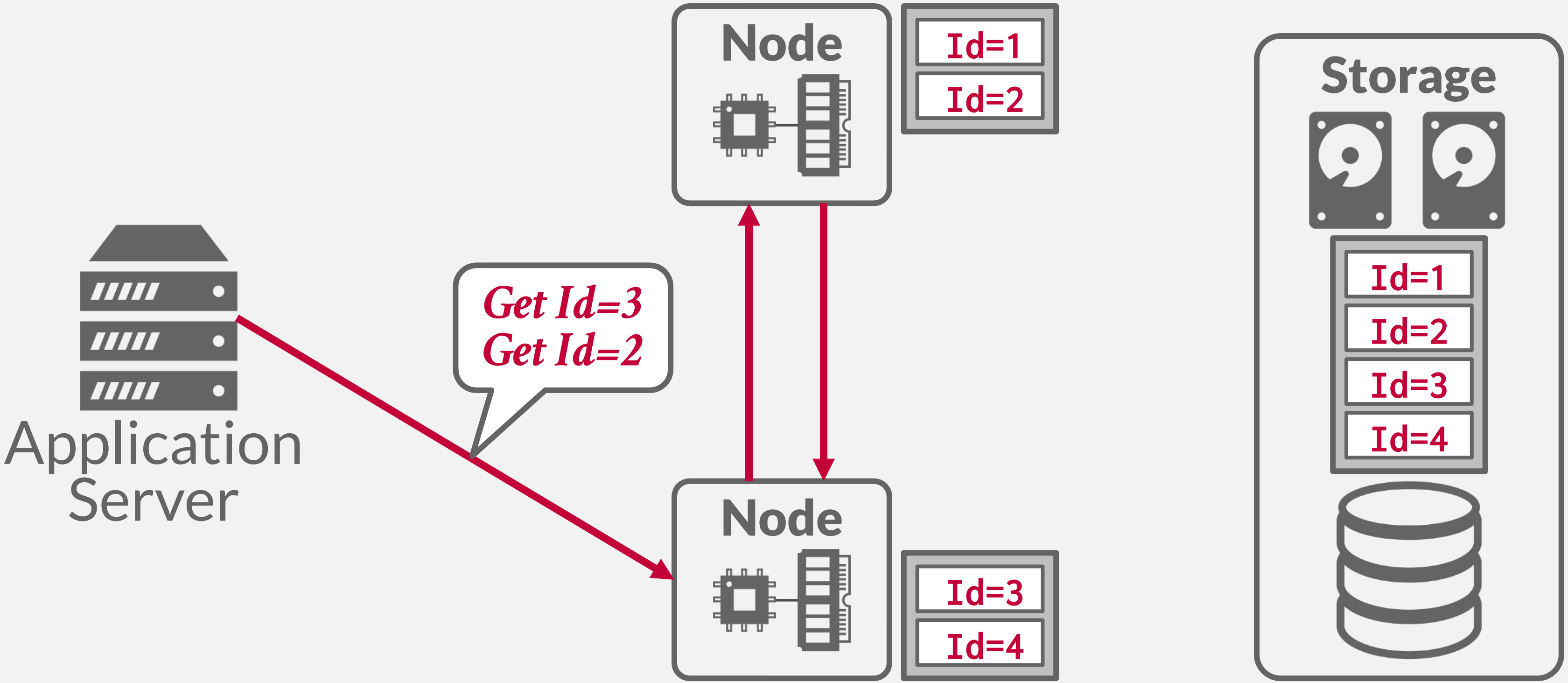
LOGICAL PARTITIONING



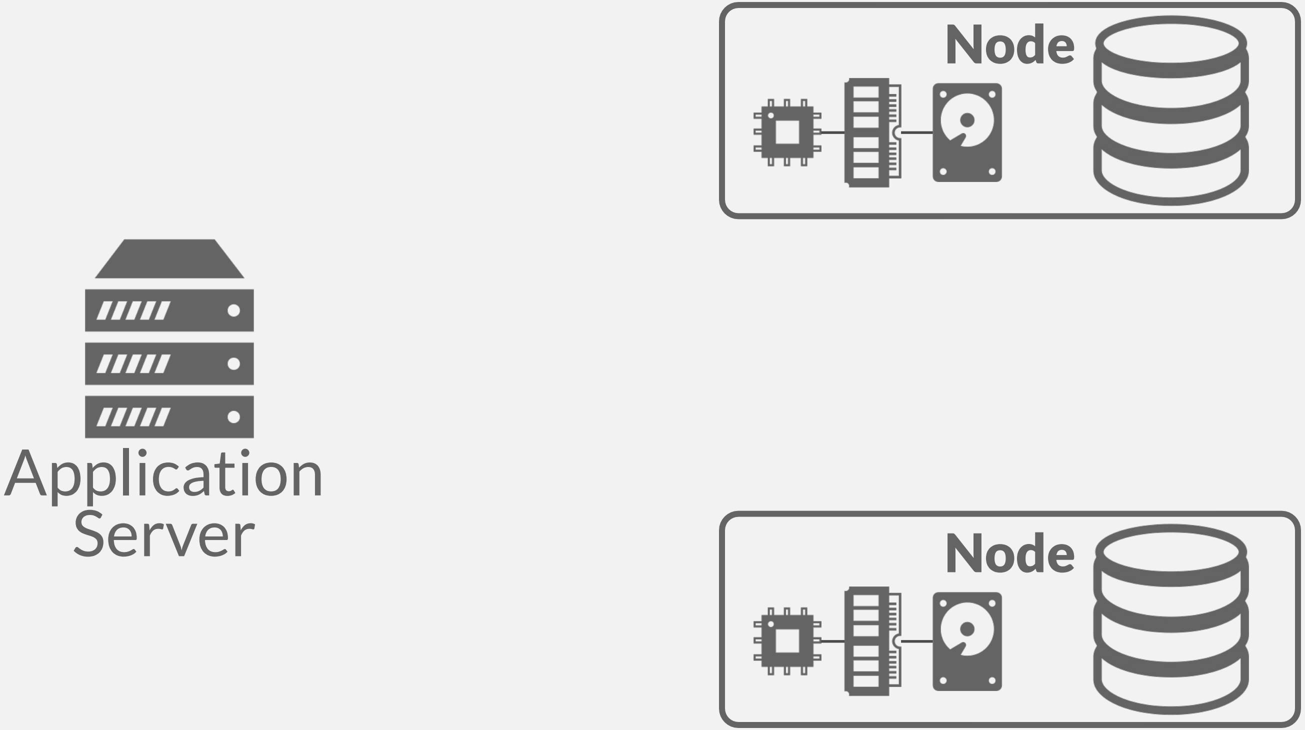
LOGICAL PARTITIONING



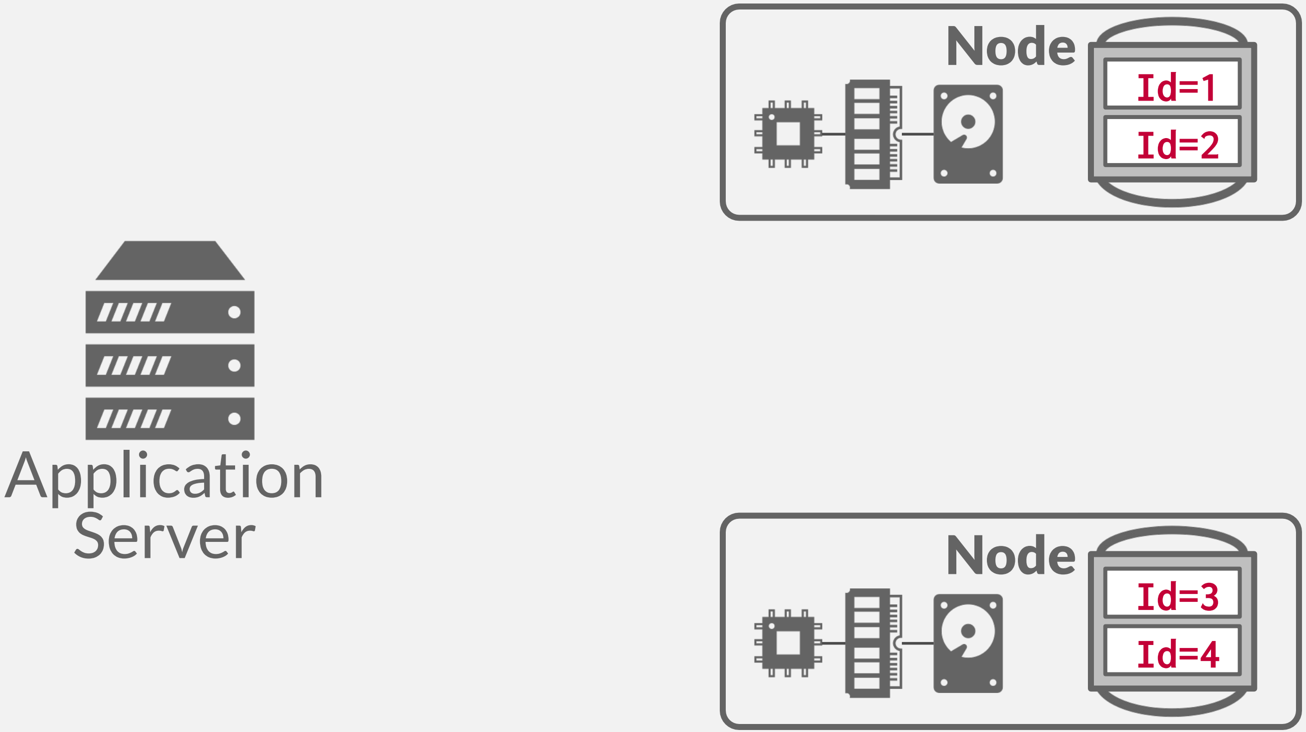
LOGICAL PARTITIONING



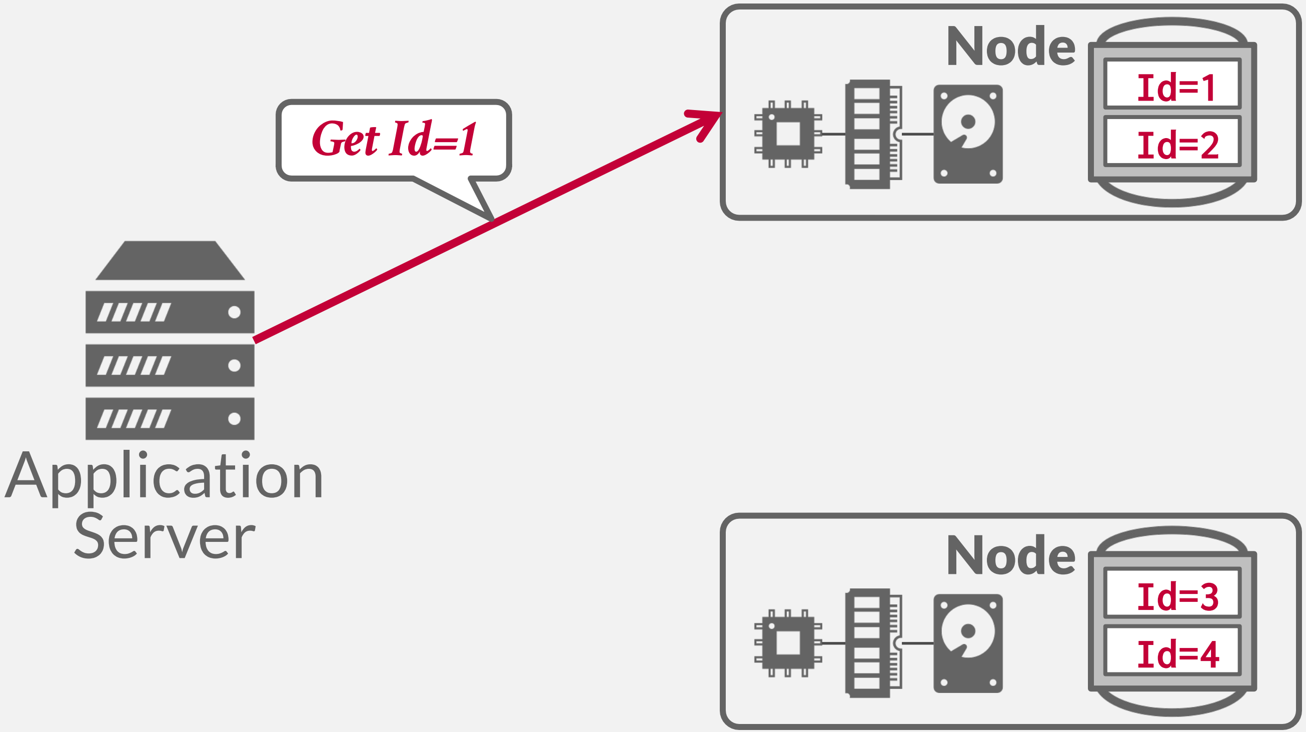
PHYSICAL PARTITIONING



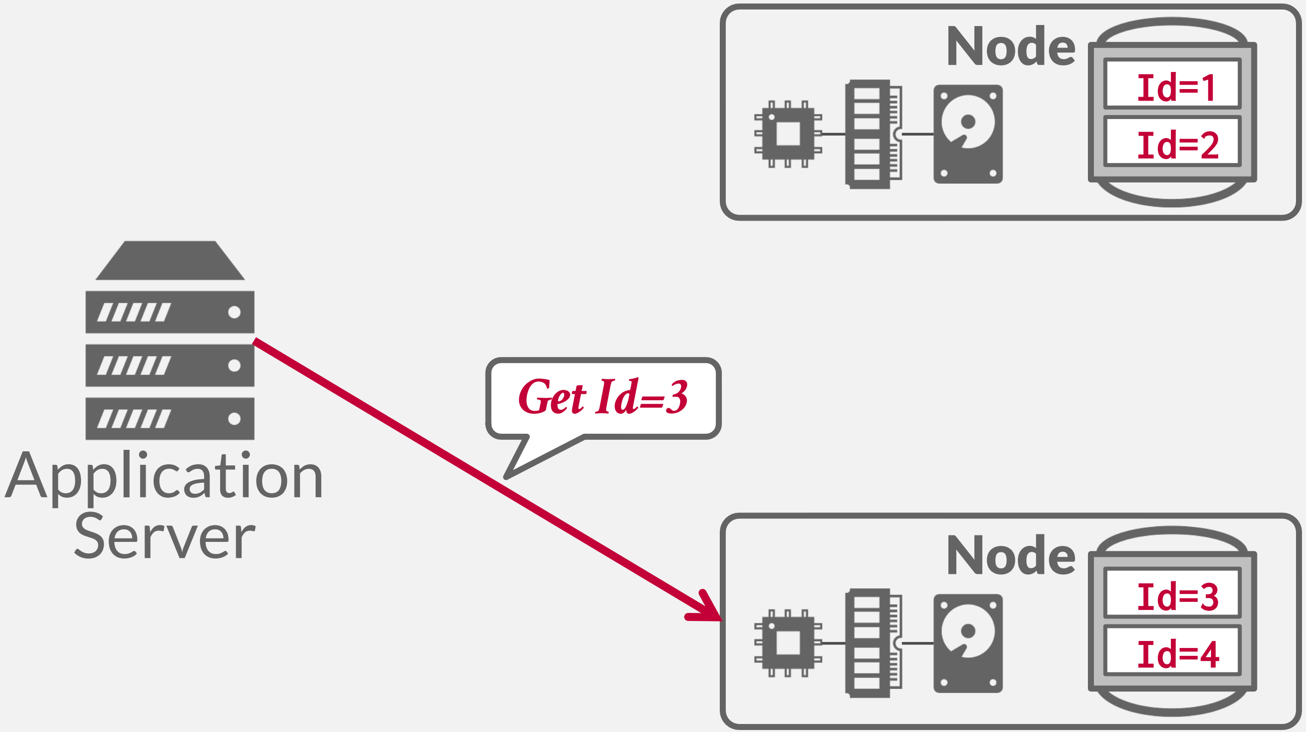
PHYSICAL PARTITIONING



PHYSICAL PARTITIONING



PHYSICAL PARTITIONING



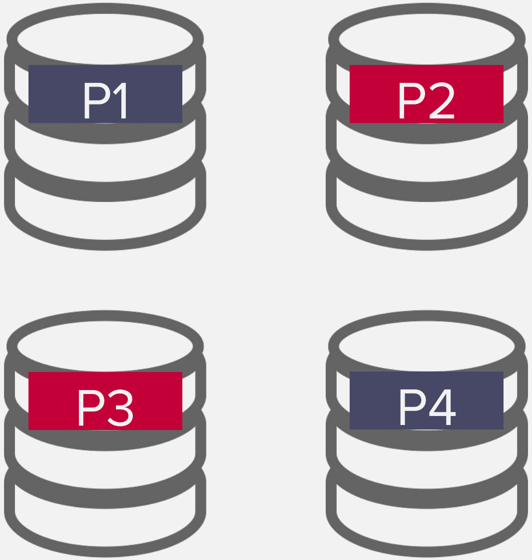
HORIZONTAL PARTITIONING

Partitioning Key

Table

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 = ?
```

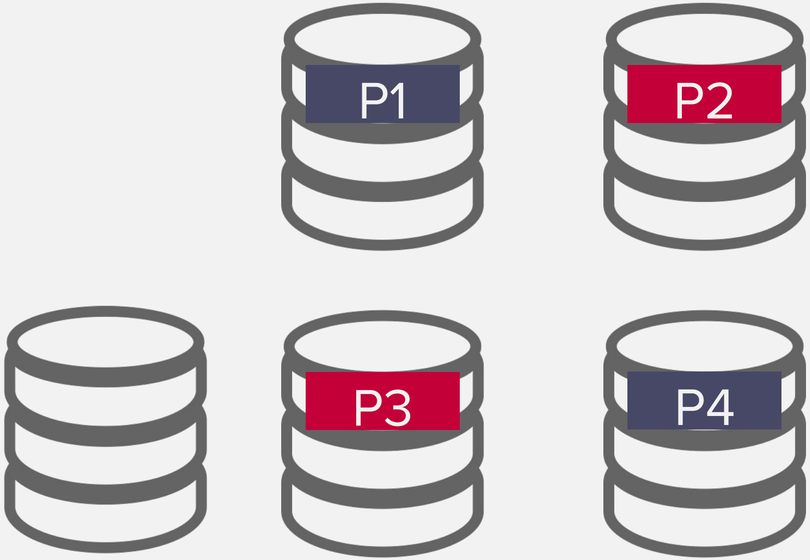
HORIZONTAL PARTITIONING

Partitioning Key

Table

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

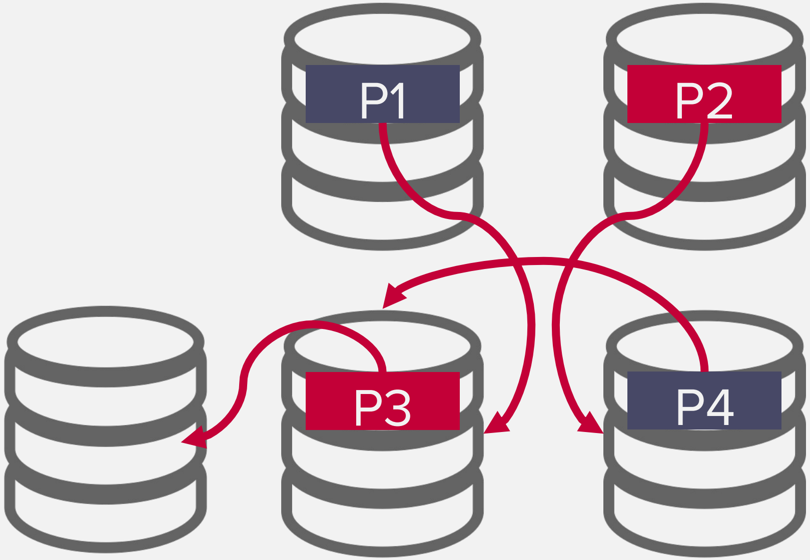
Table

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*

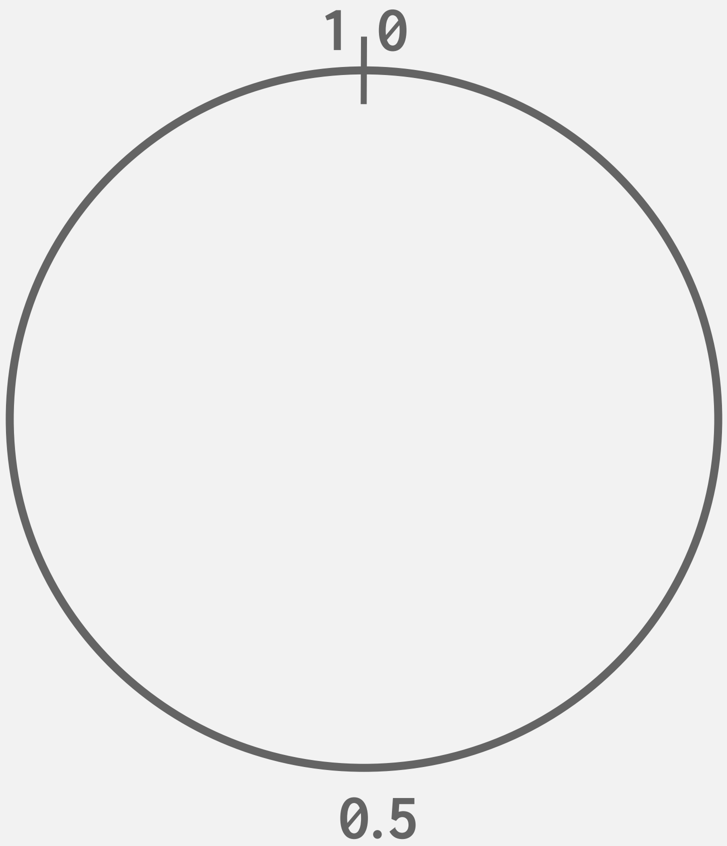
Ideal Query:

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

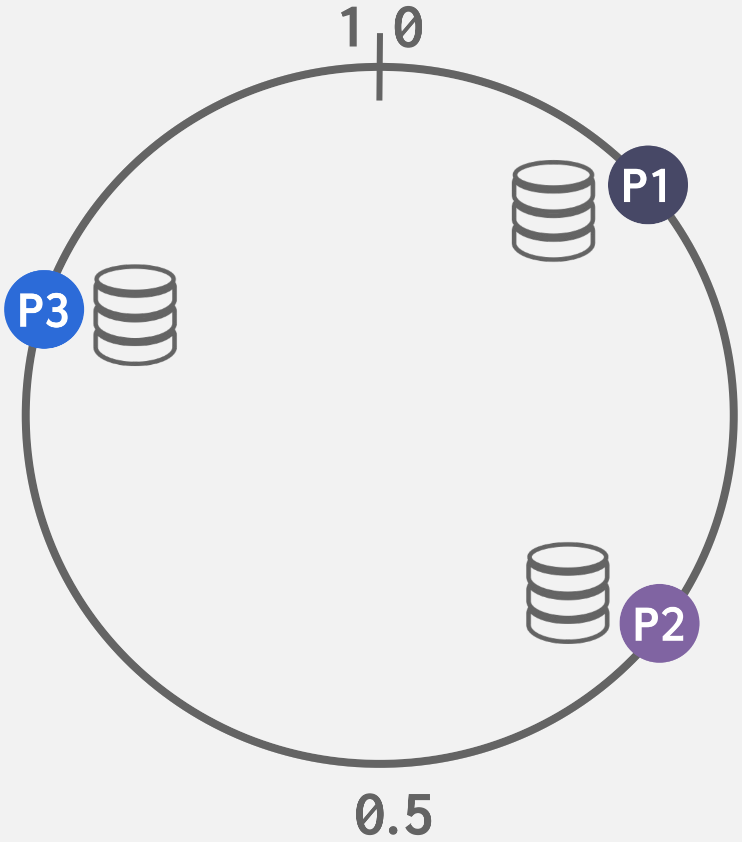
Partitions



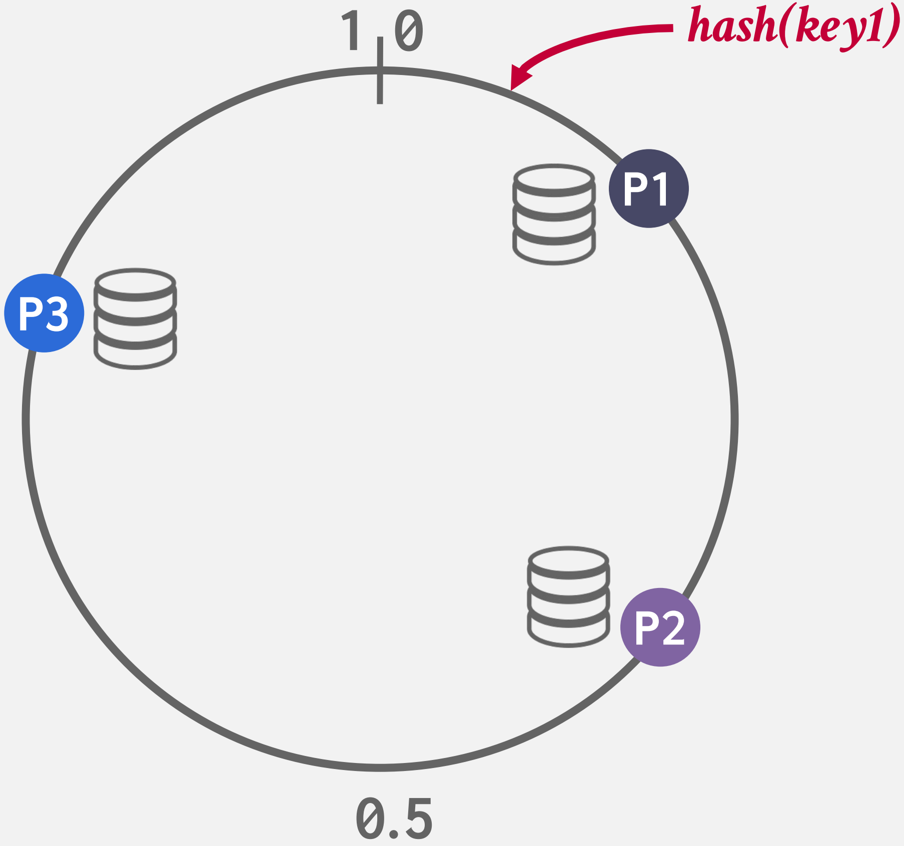
CONSISTENT HASHING



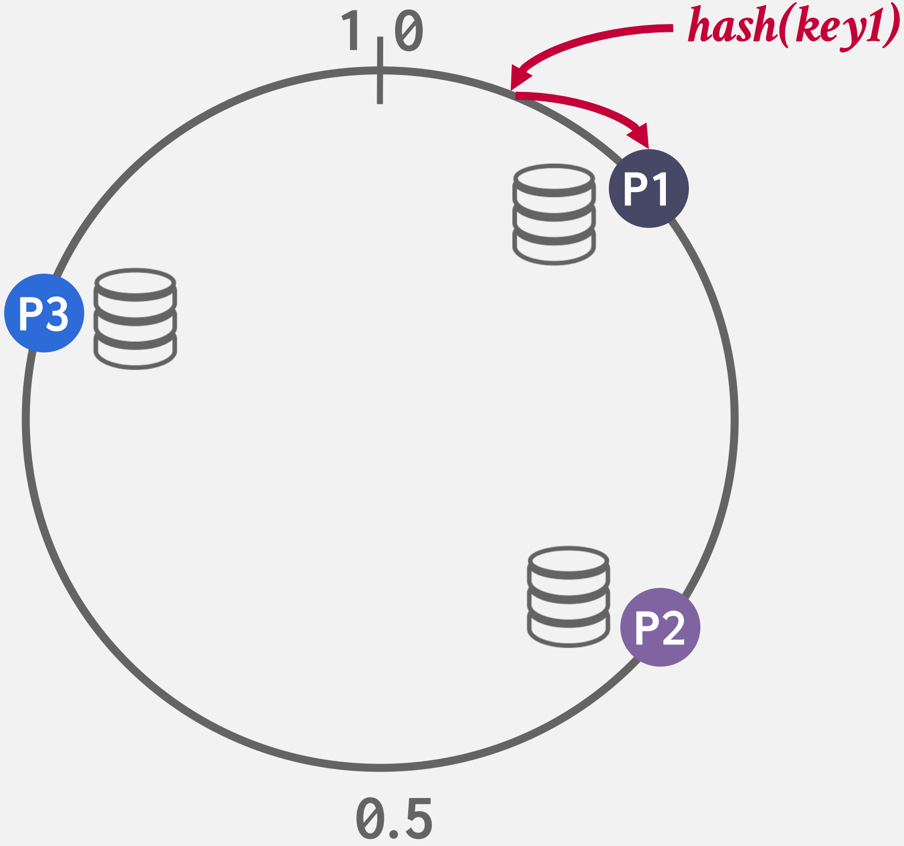
CONSISTENT HASHING



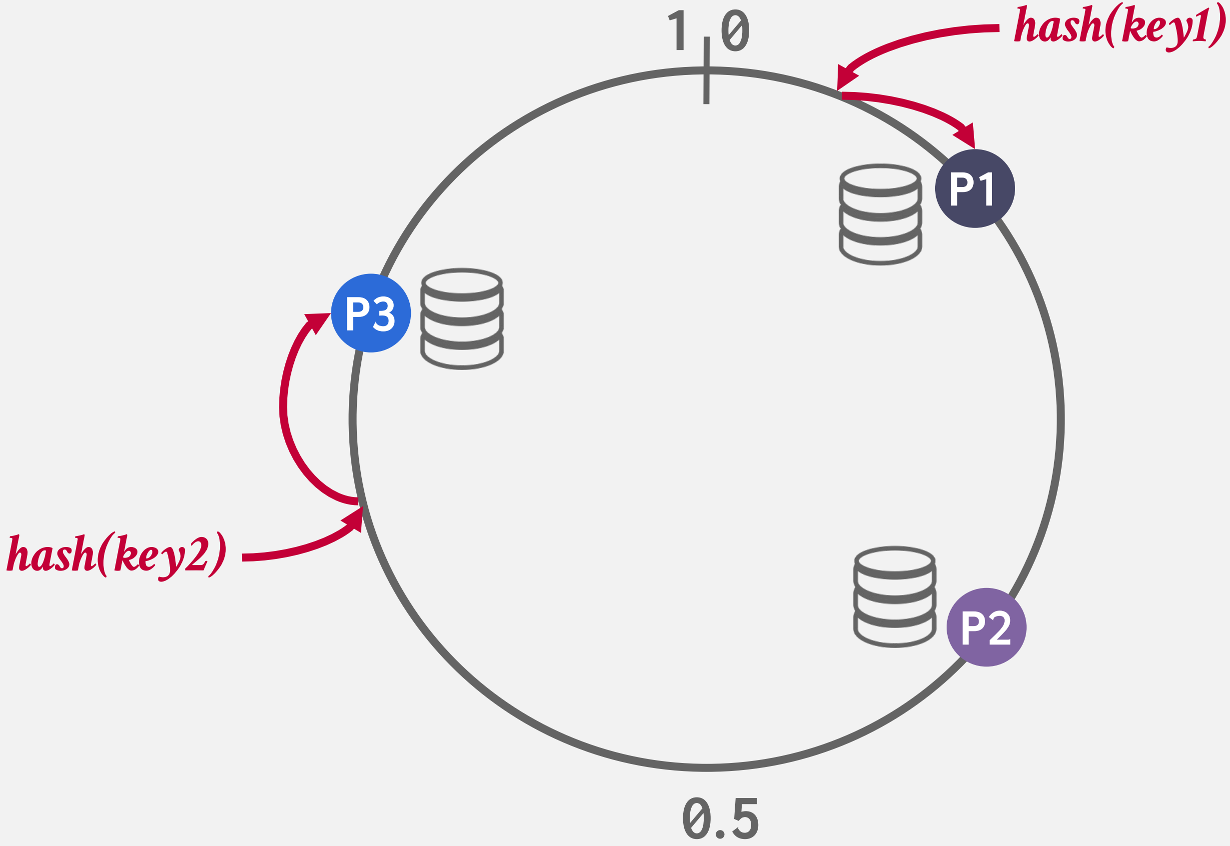
CONSISTENT HASHING



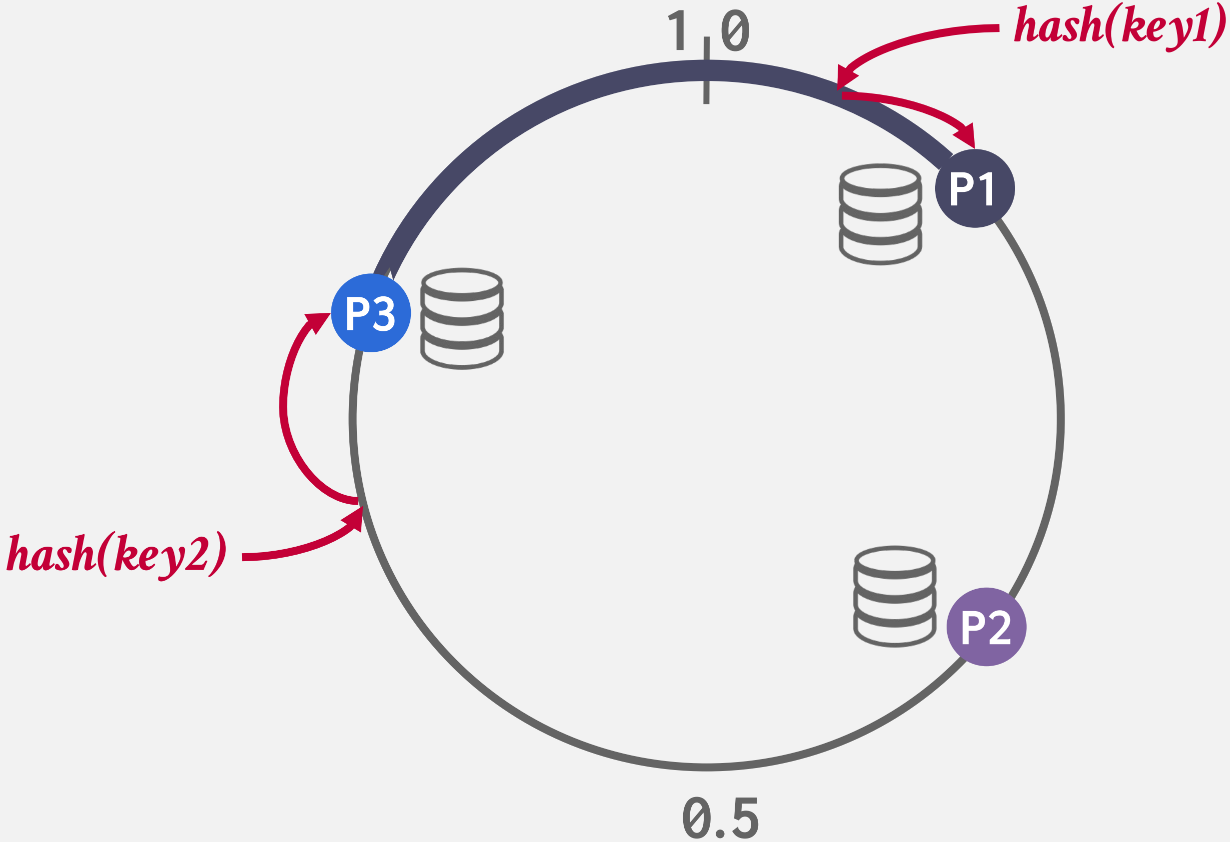
CONSISTENT HASHING



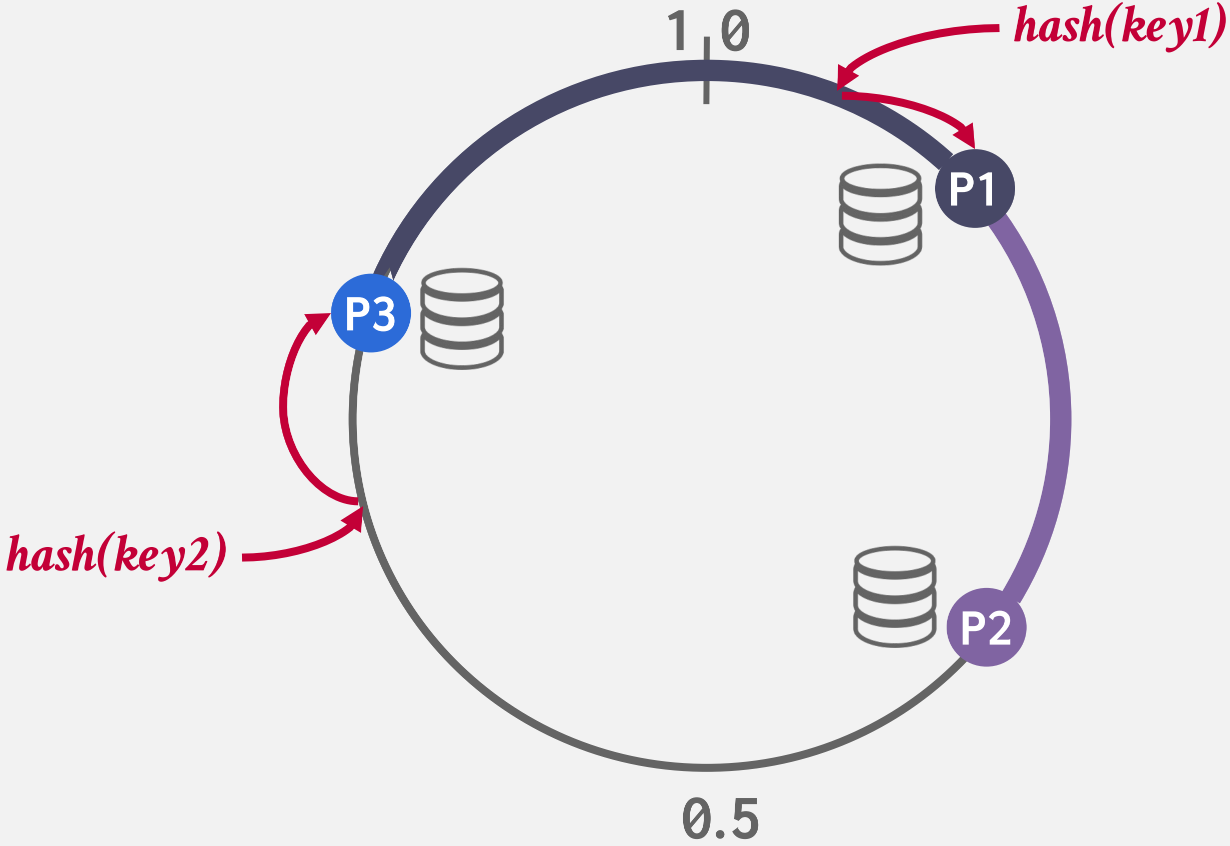
CONSISTENT HASHING



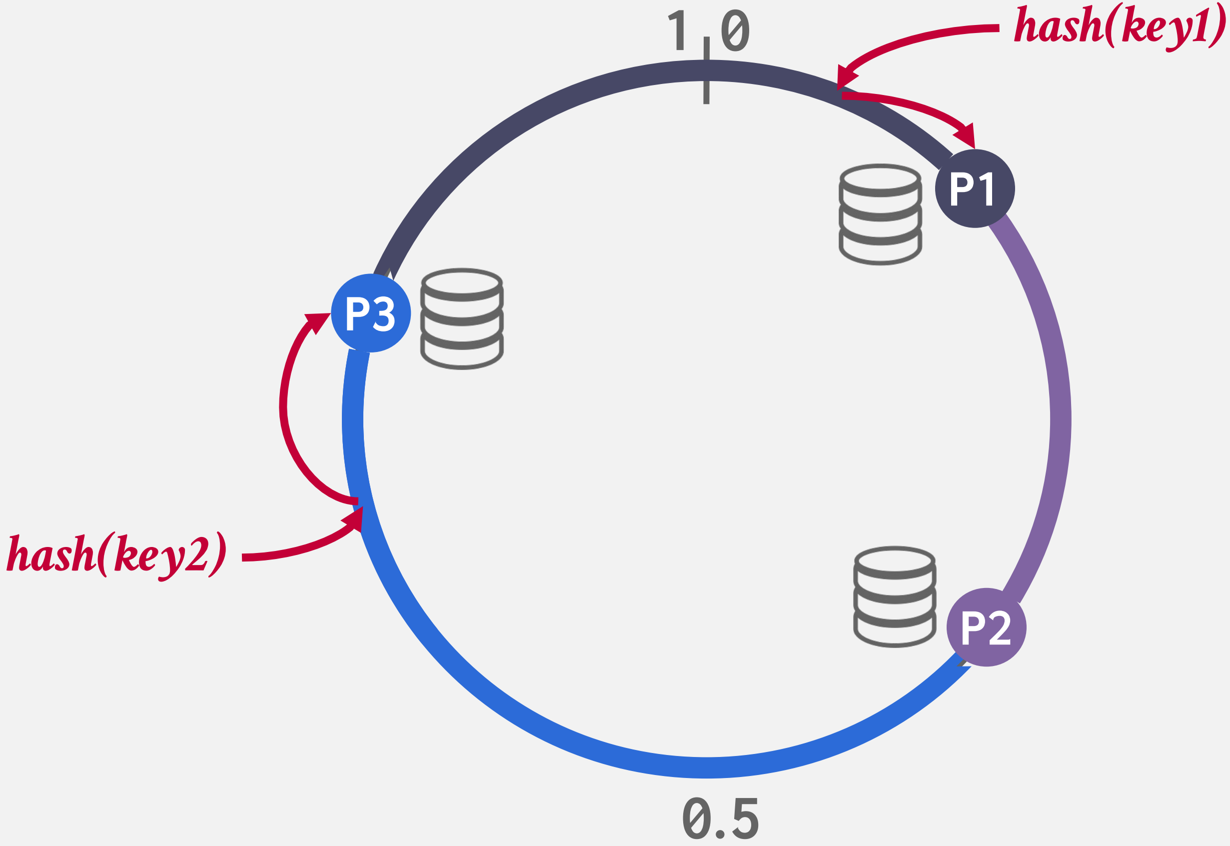
CONSISTENT HASHING



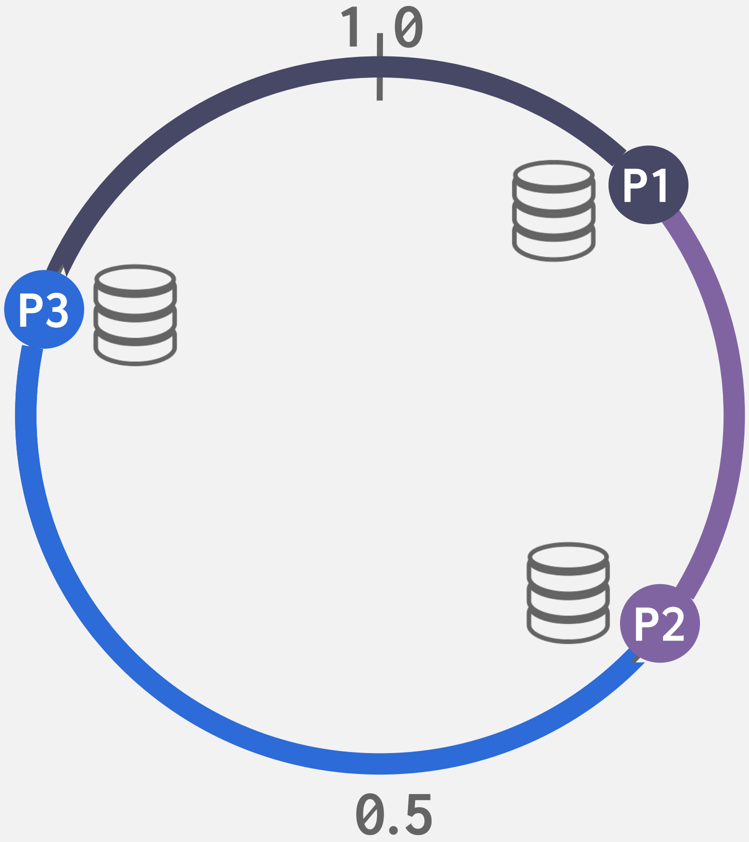
CONSISTENT HASHING



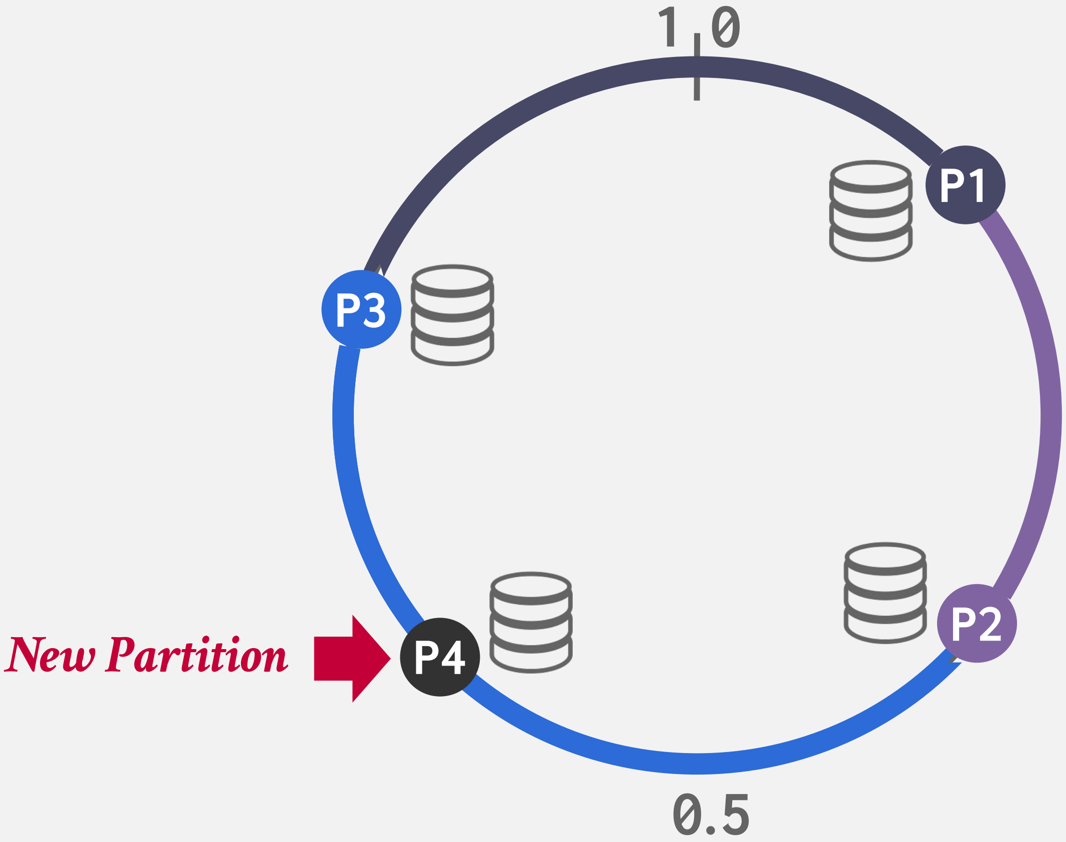
CONSISTENT HASHING



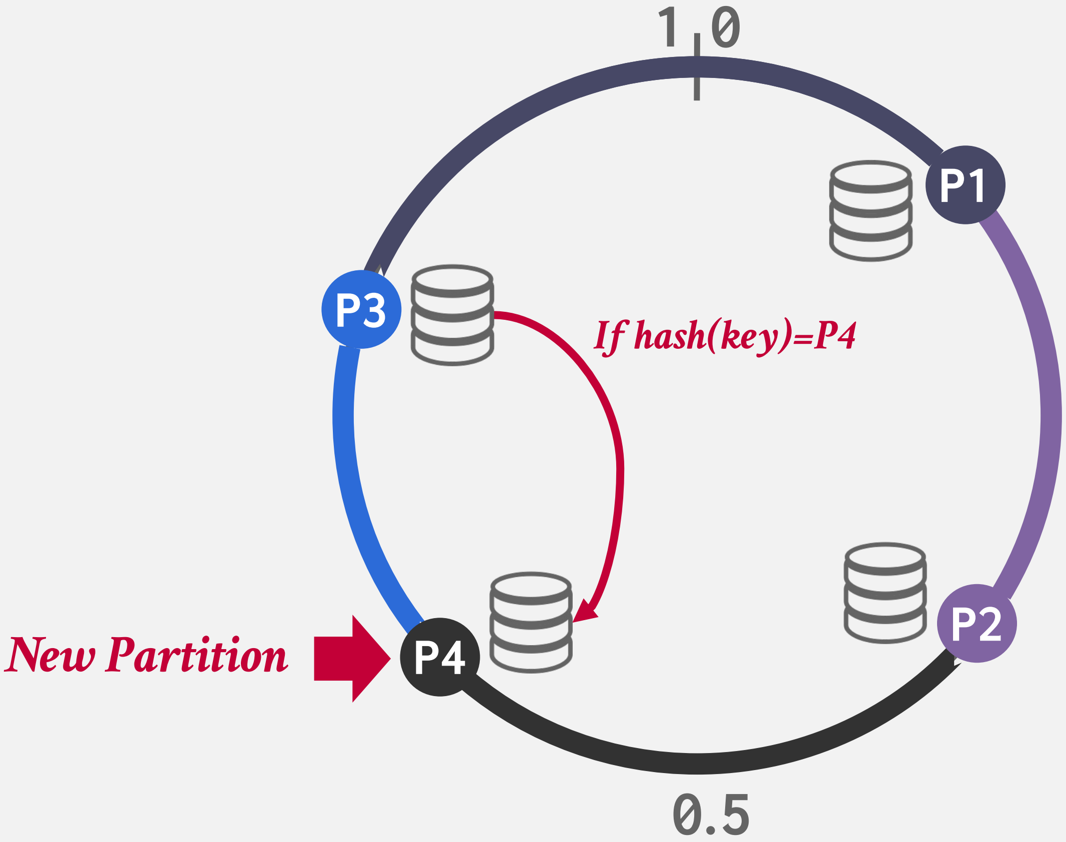
CONSISTENT HASHING



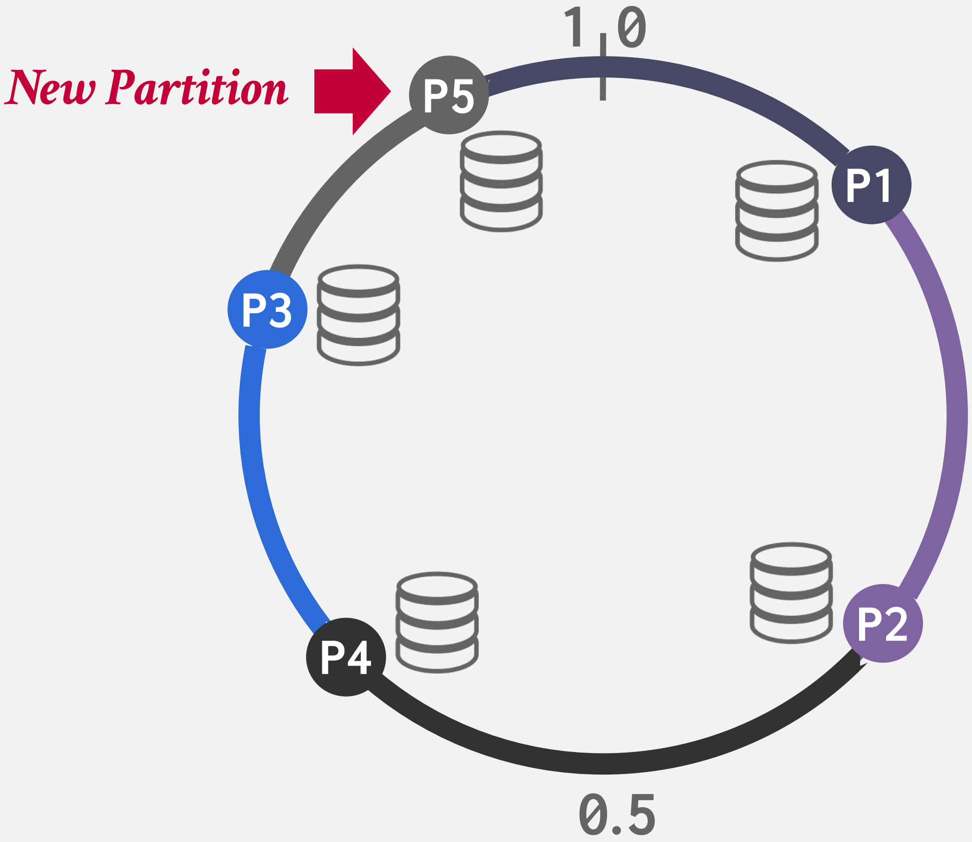
CONSISTENT HASHING



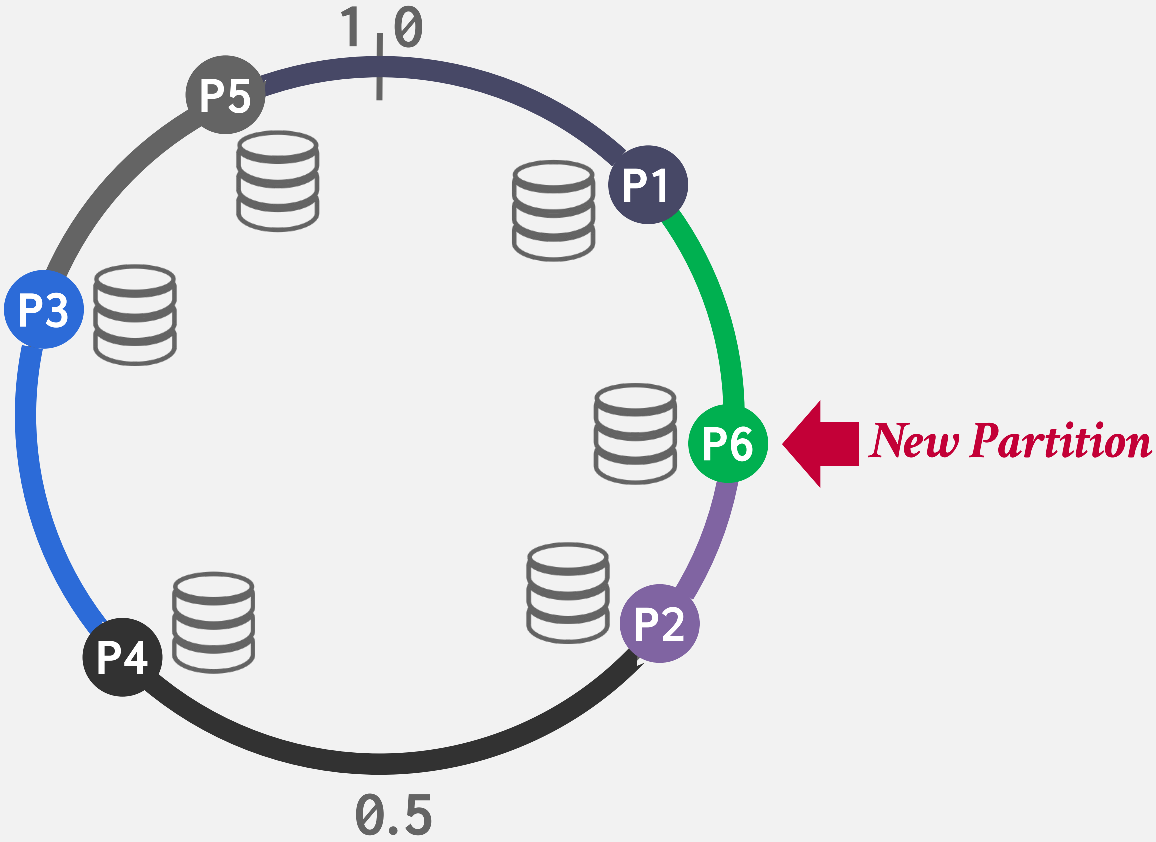
CONSISTENT HASHING



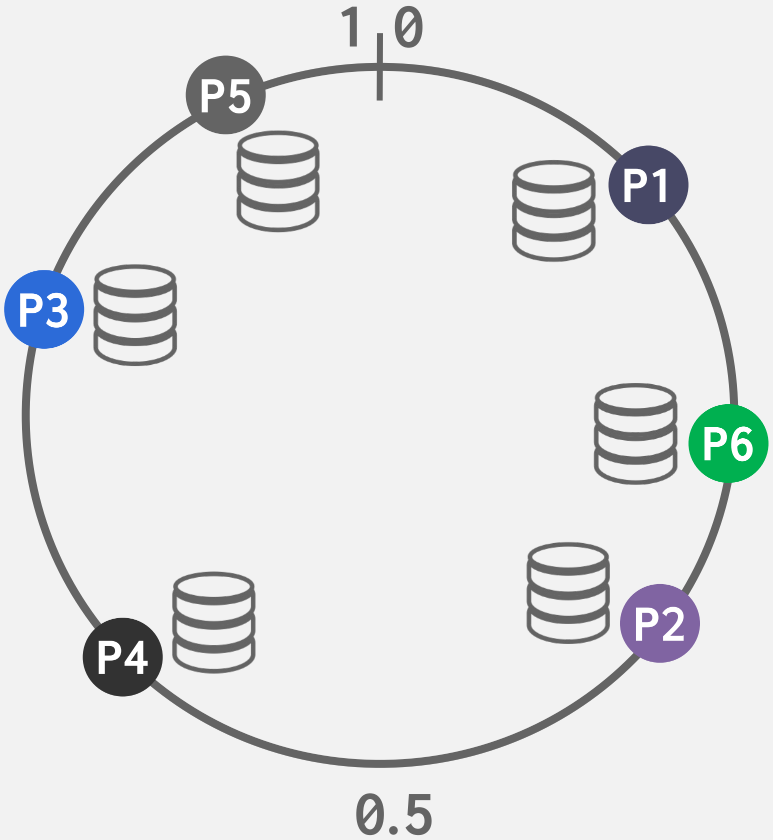
CONSISTENT HASHING



CONSISTENT HASHING

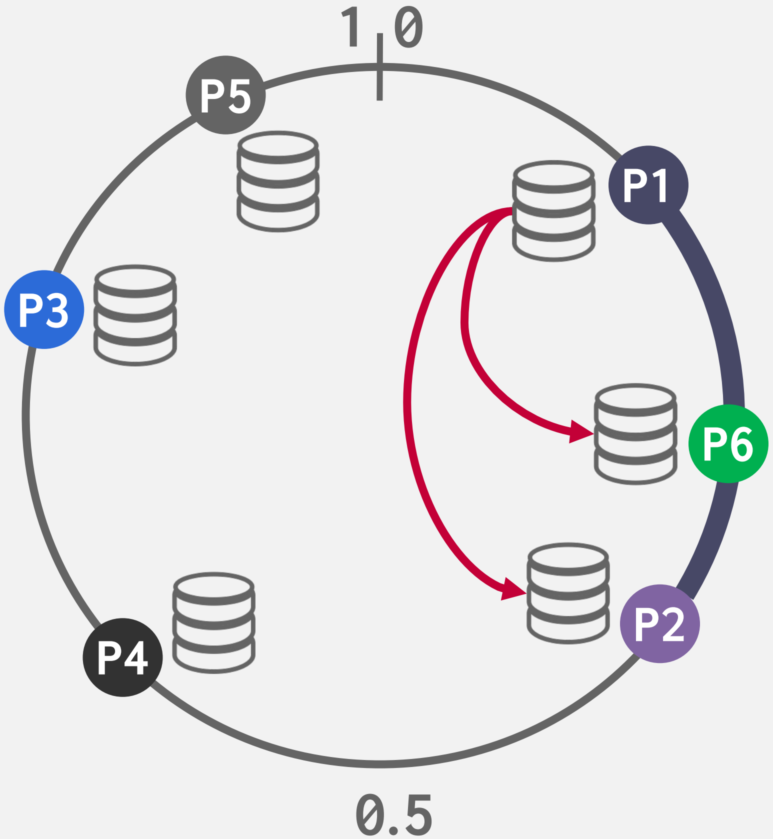


CONSISTENT HASHING



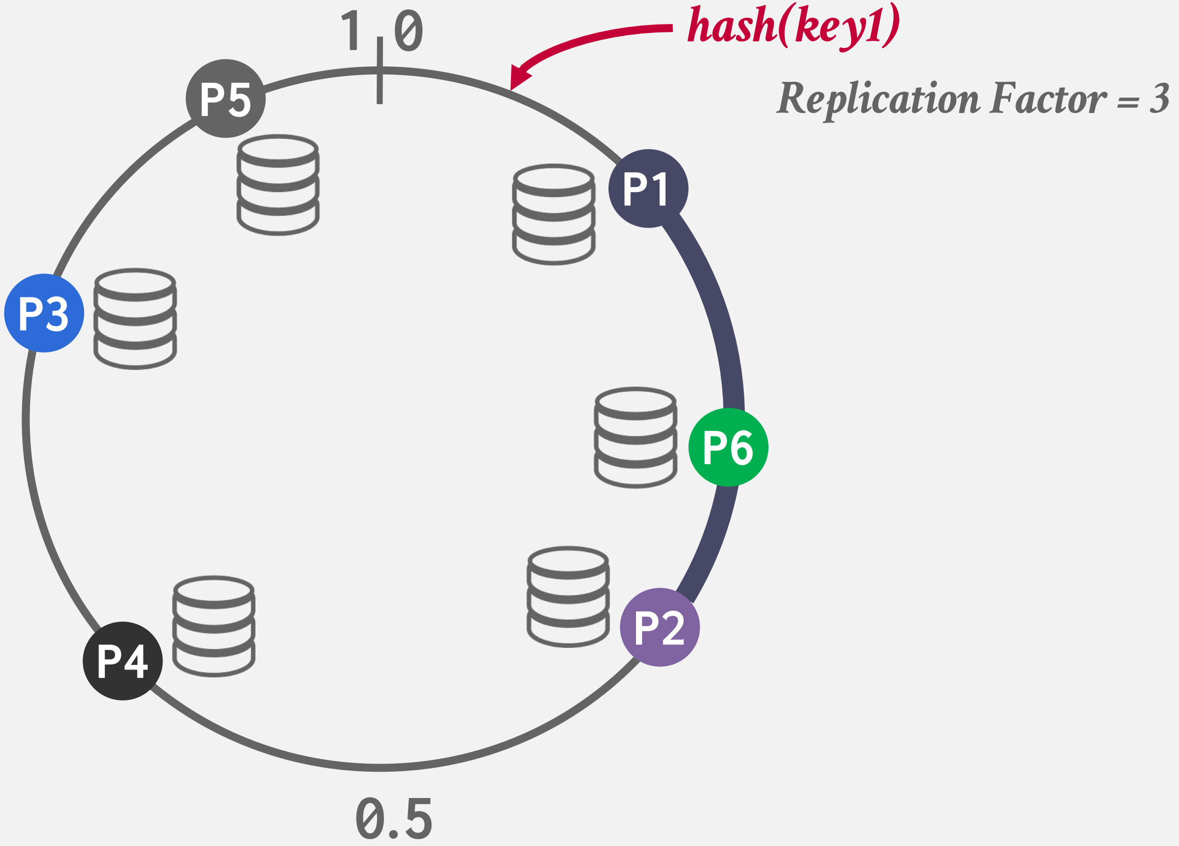
Replication Factor = 3

CONSISTENT HASHING

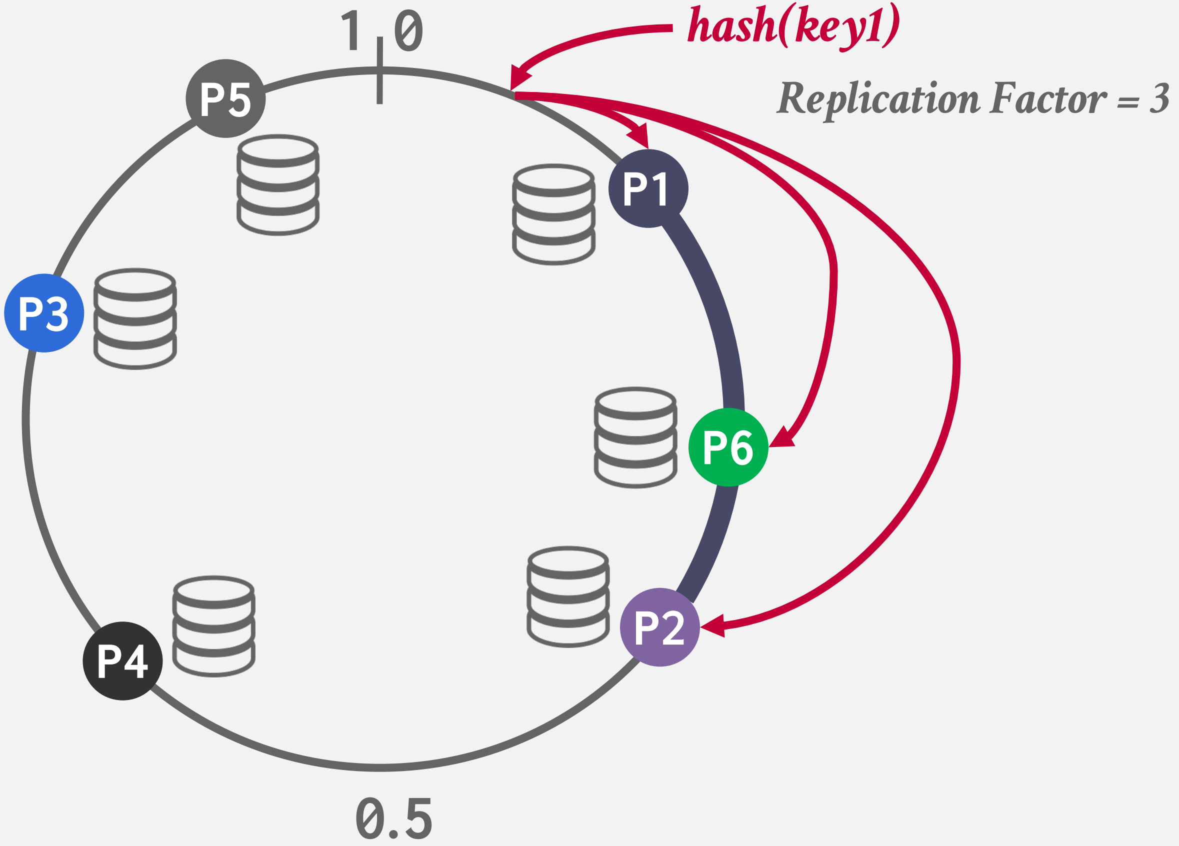


Replication Factor = 3

CONSISTENT HASHING

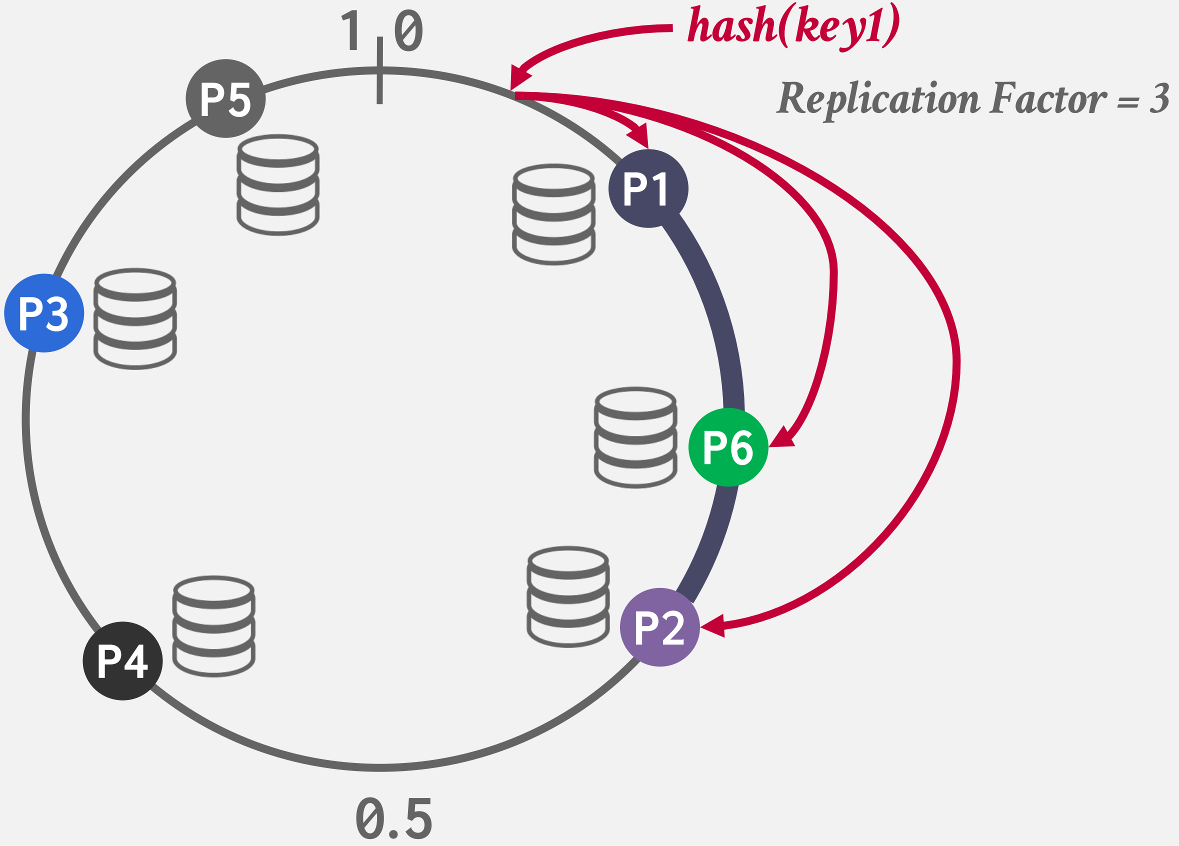


CONSISTENT HASHING



CONSISTENT HASHING

Couchbase
snowflake
AEROSPIKE
MEMCACHED
cassandra
riak
SCYLLA



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.

Most distributed DBMSs use a hybrid approach where they periodically elect some node to be a temporary coordinator.

TP MONITORS

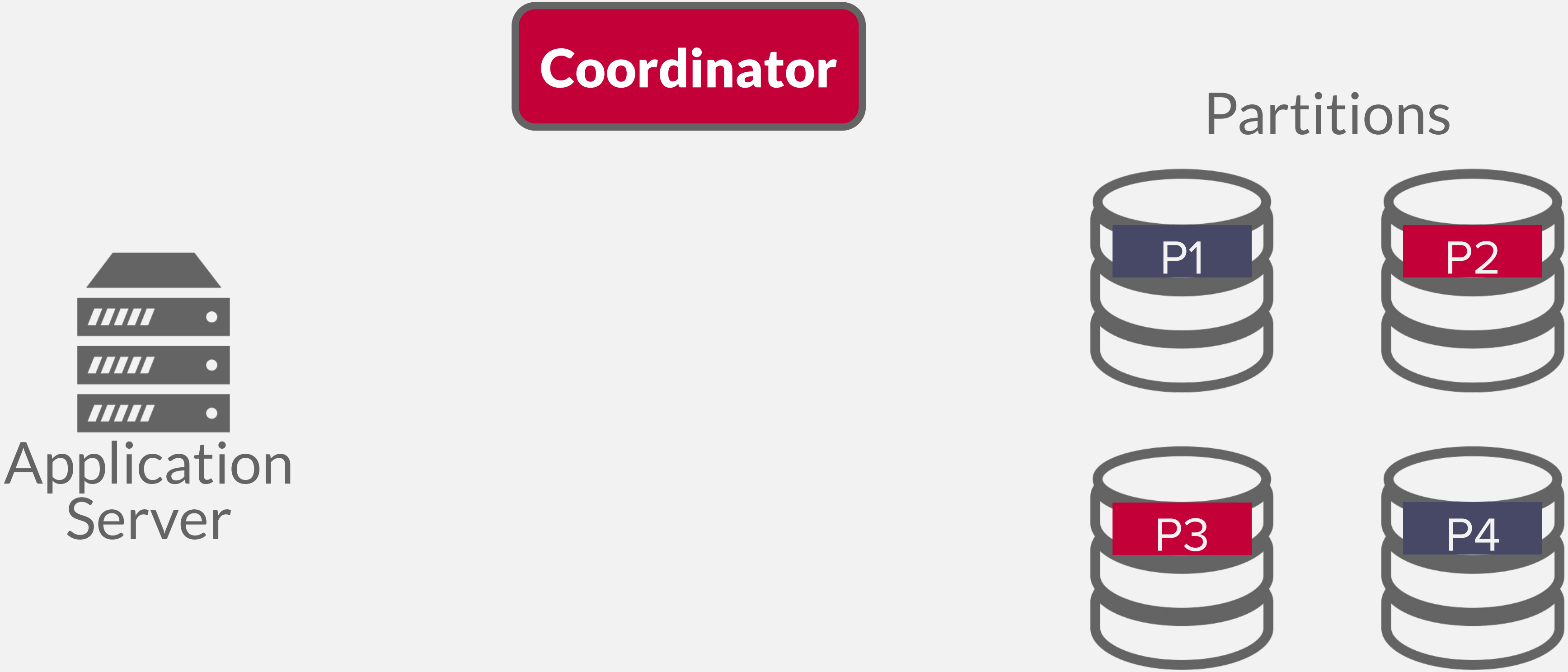
A **TP Monitor** is an example of a centralized coordinator for distributed DBMSs.

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

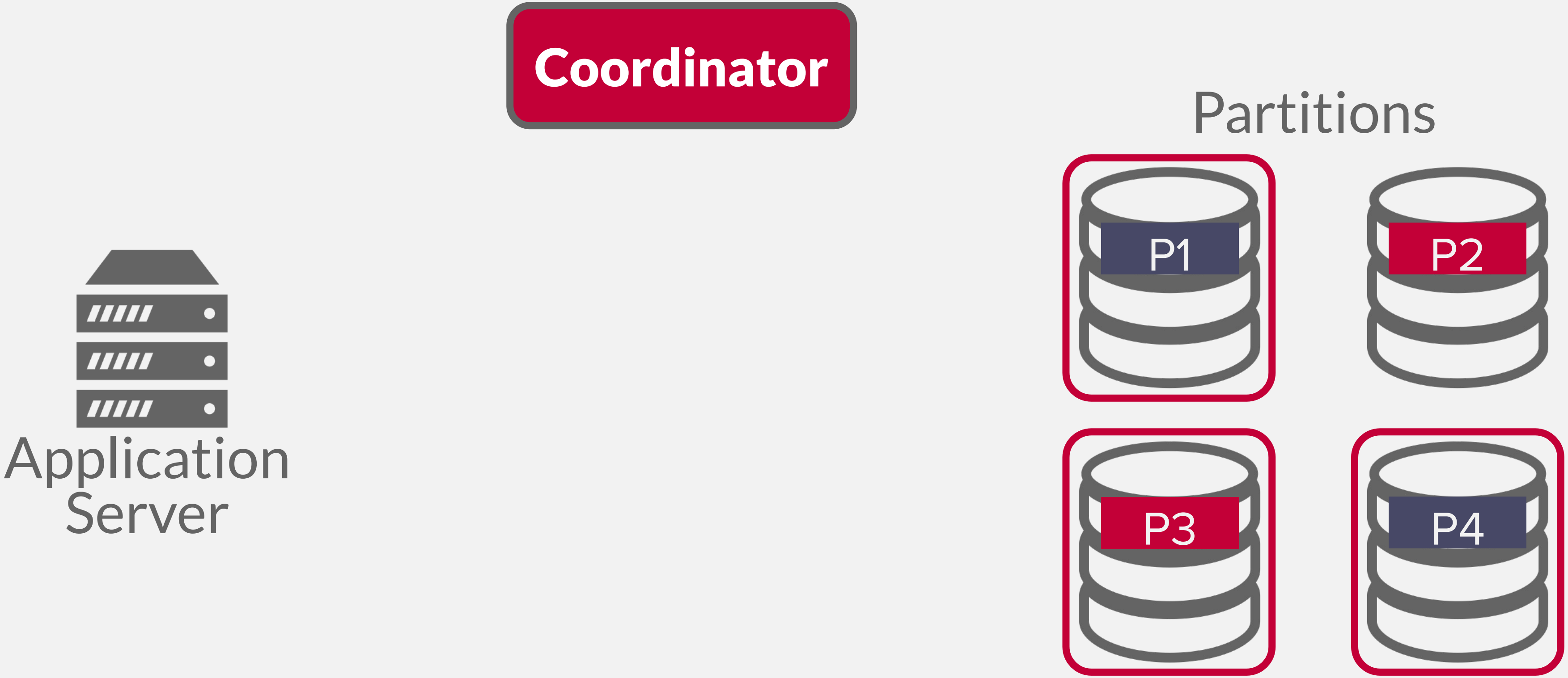
→ Examples: ATMs, Airline Reservations.

Standardized protocol from 1990s: X/Open XA

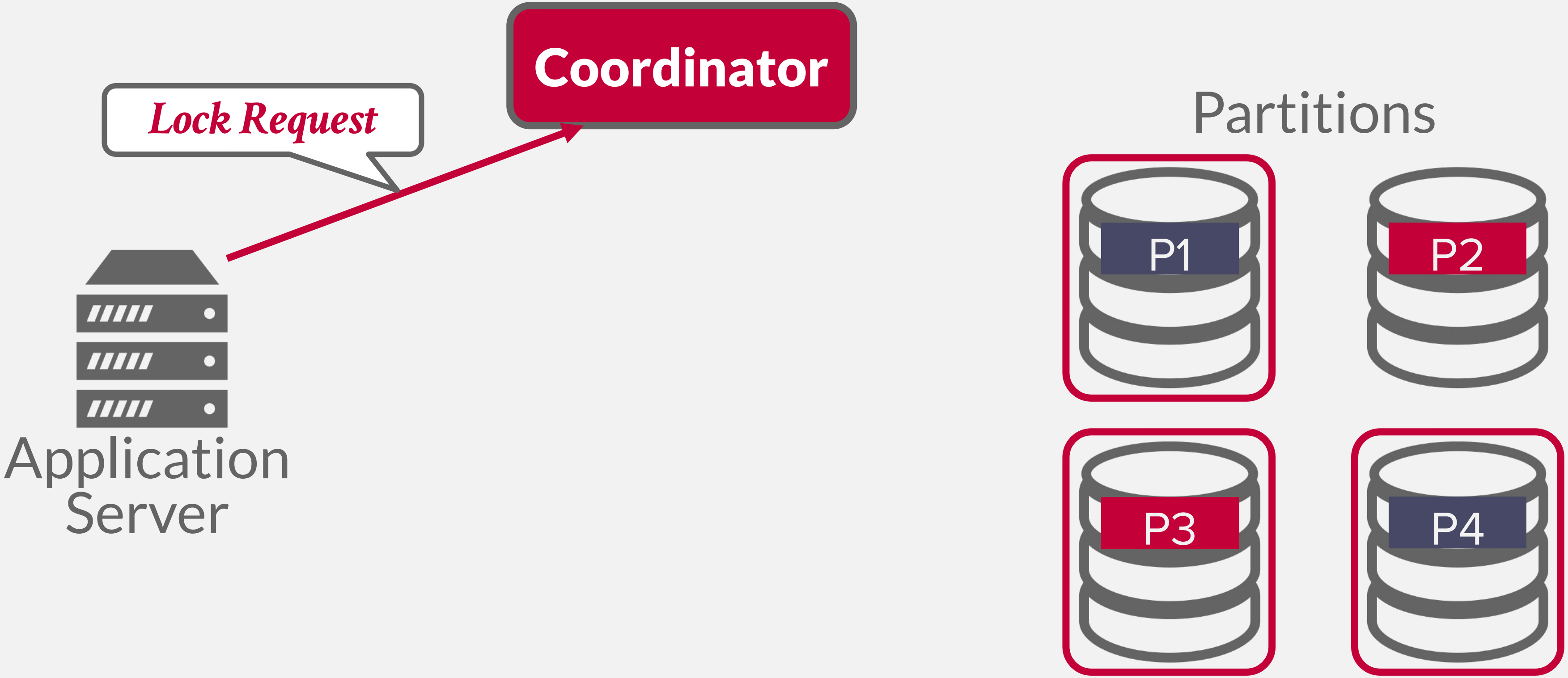
CENTRALIZED COORDINATOR



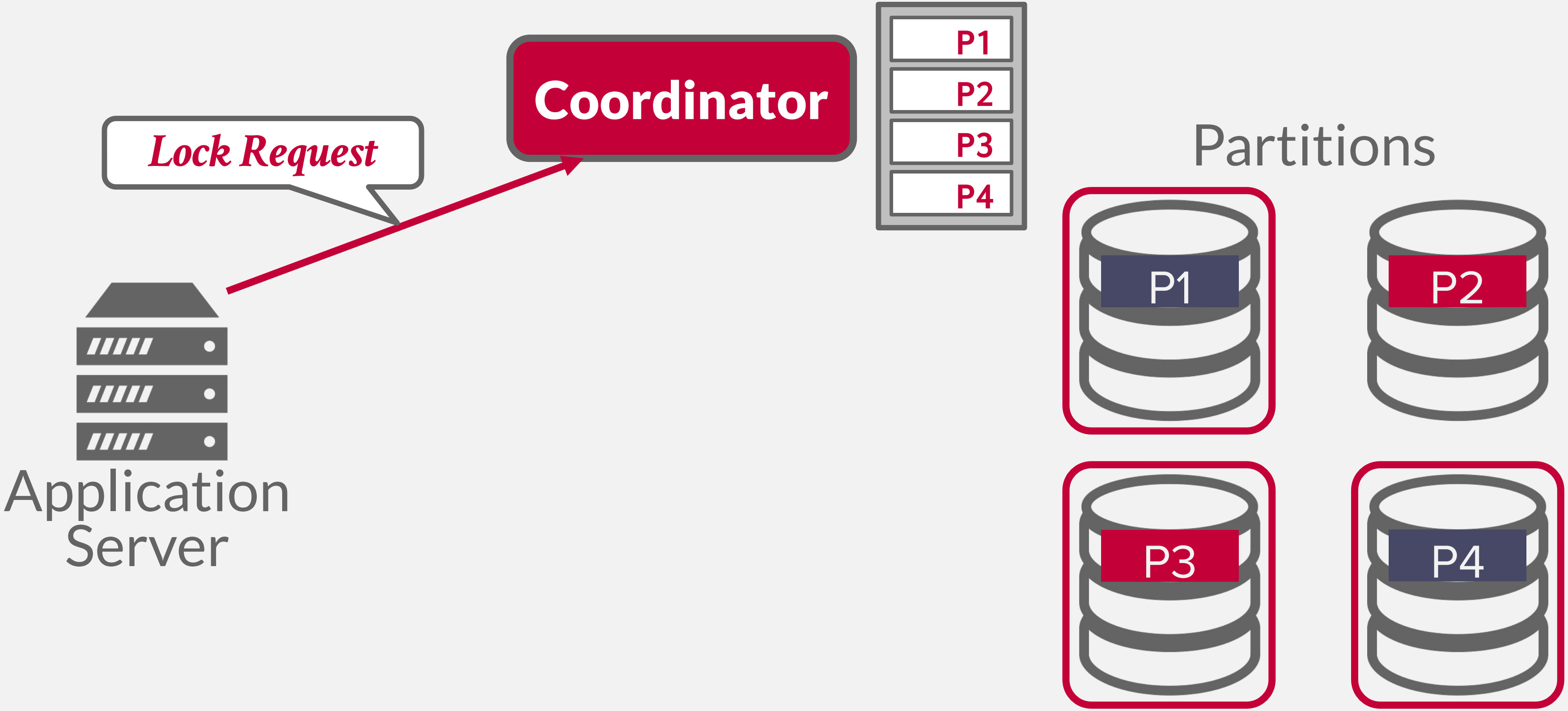
CENTRALIZED COORDINATOR



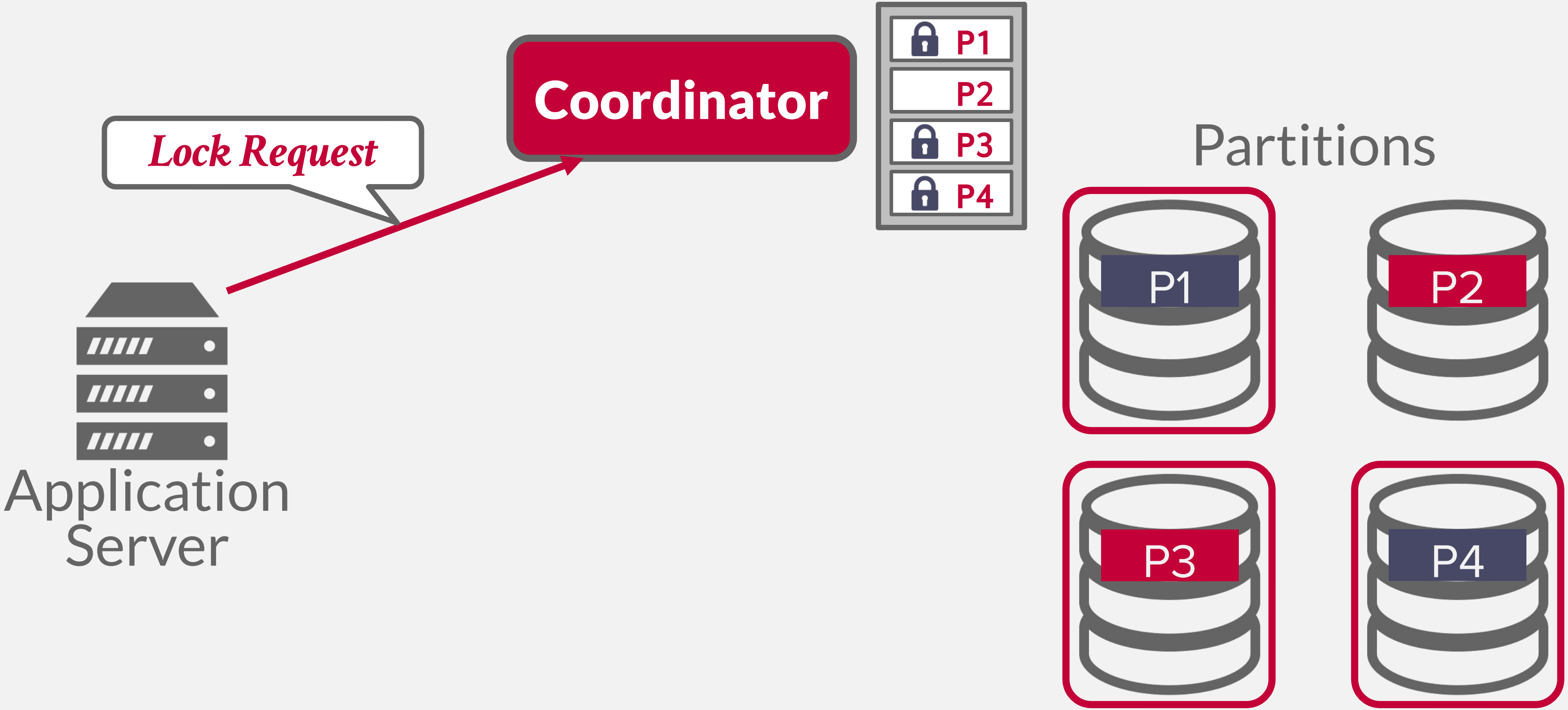
CENTRALIZED COORDINATOR



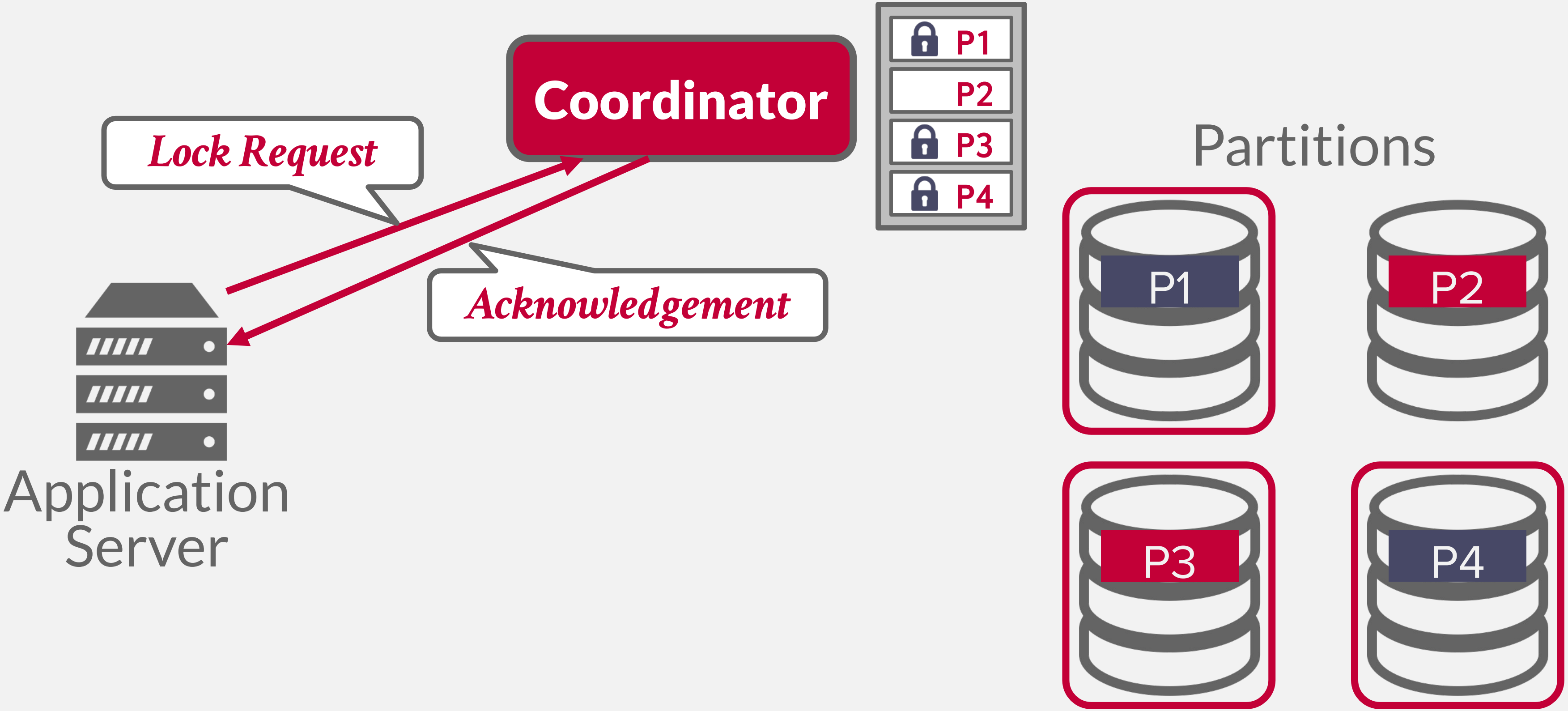
CENTRALIZED COORDINATOR



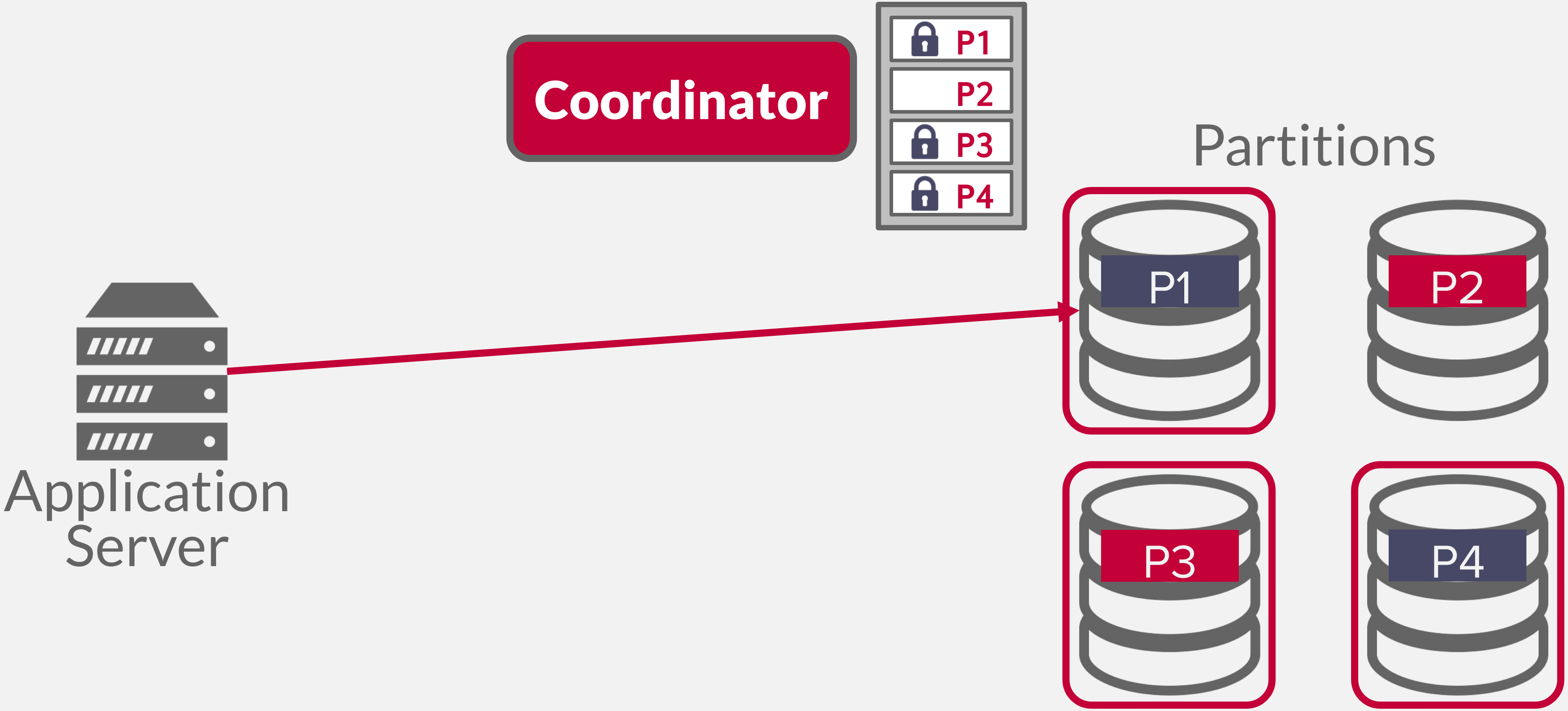
CENTRALIZED COORDINATOR



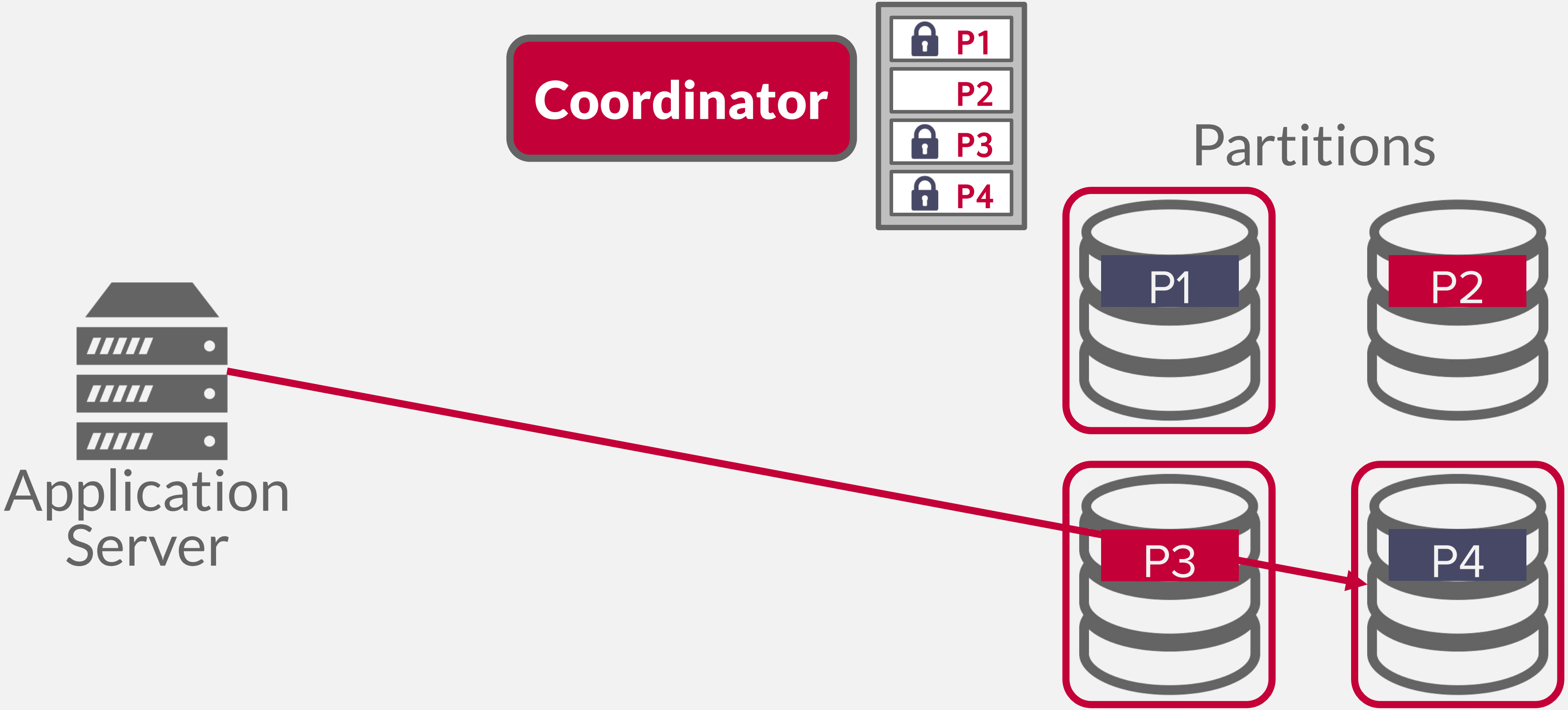
CENTRALIZED COORDINATOR



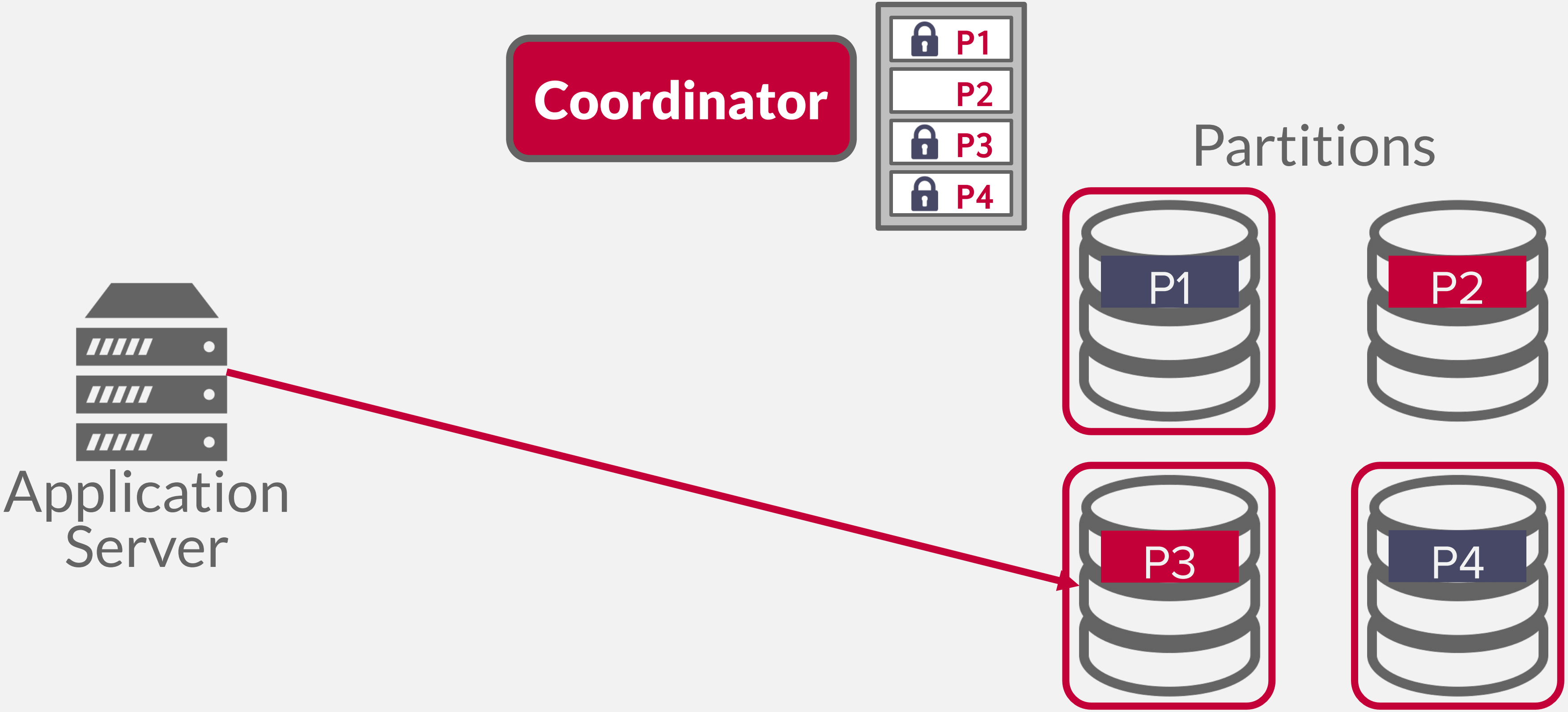
CENTRALIZED COORDINATOR



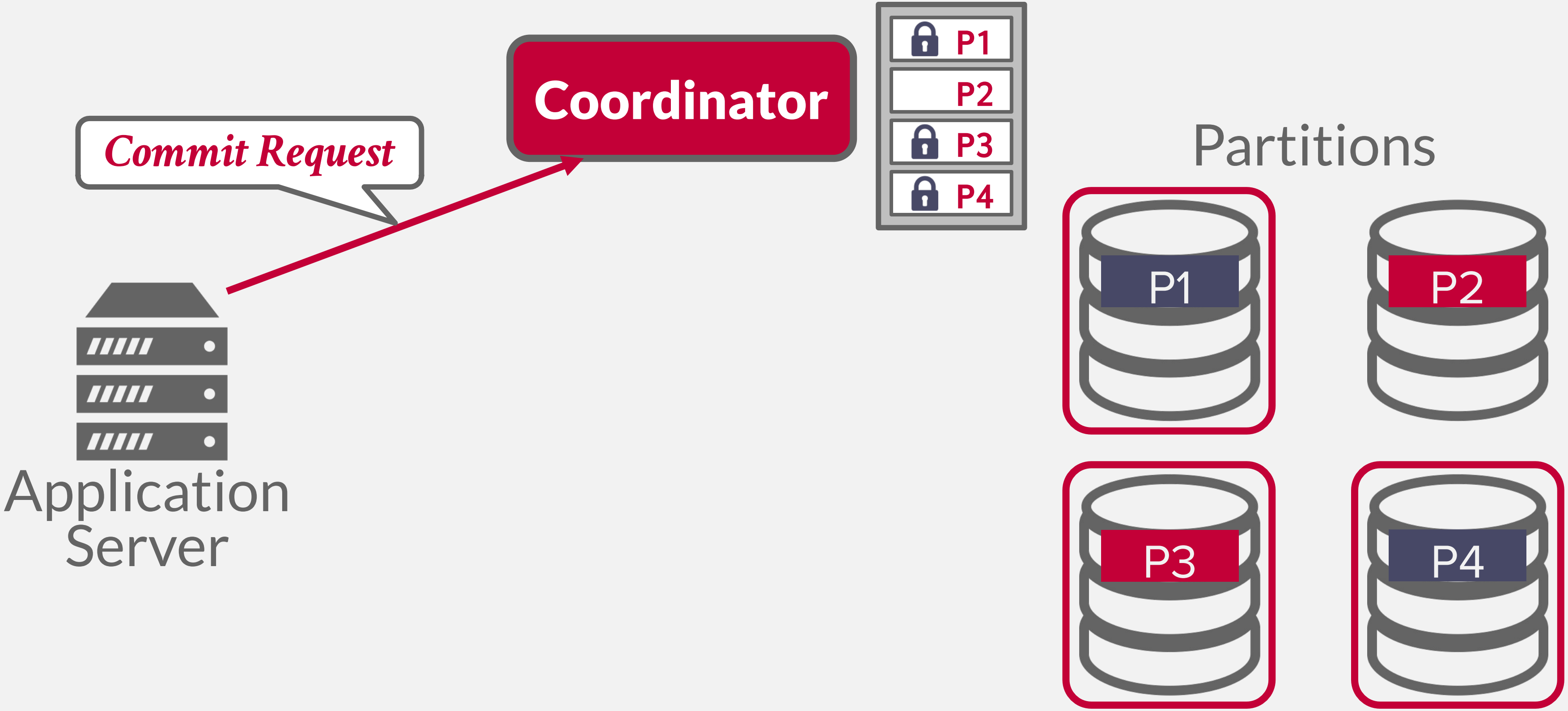
CENTRALIZED COORDINATOR



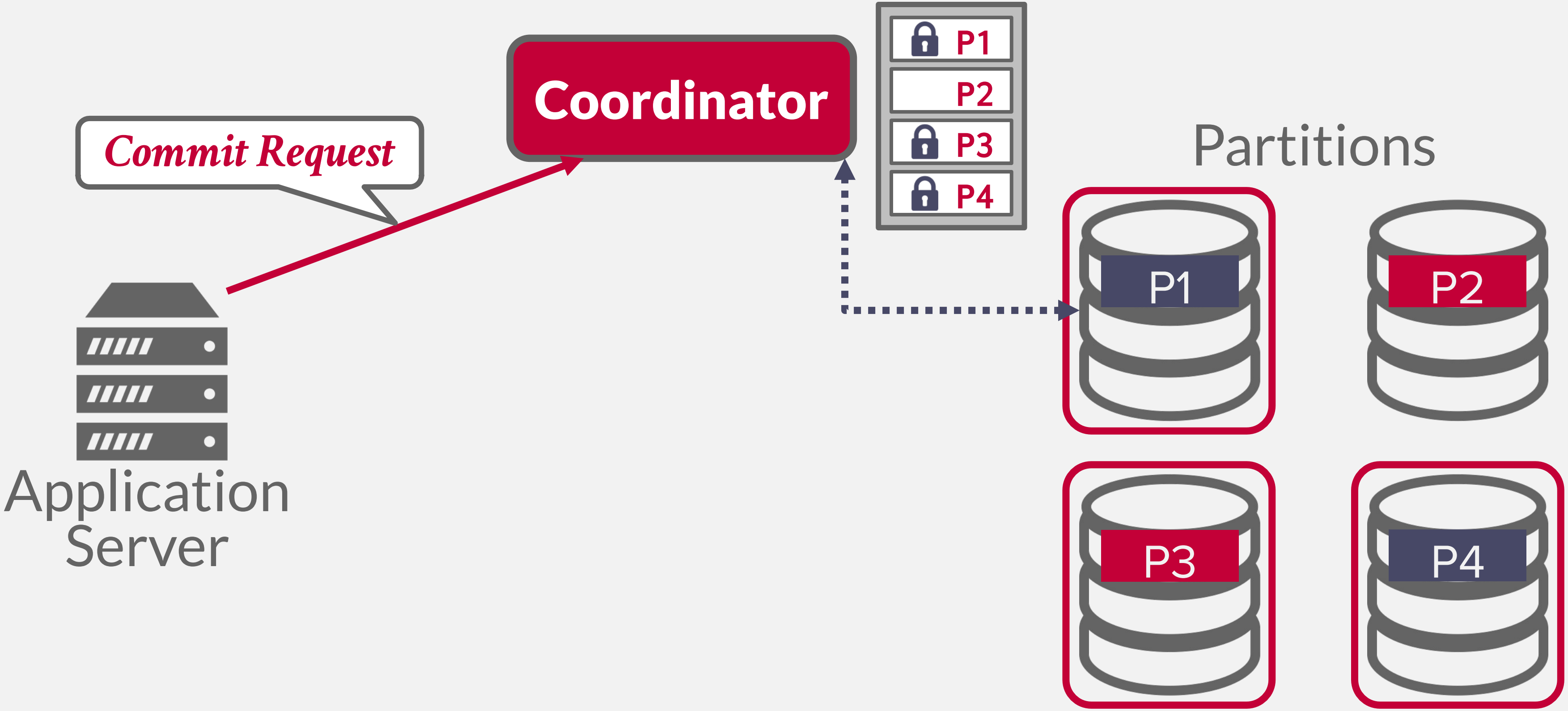
CENTRALIZED COORDINATOR



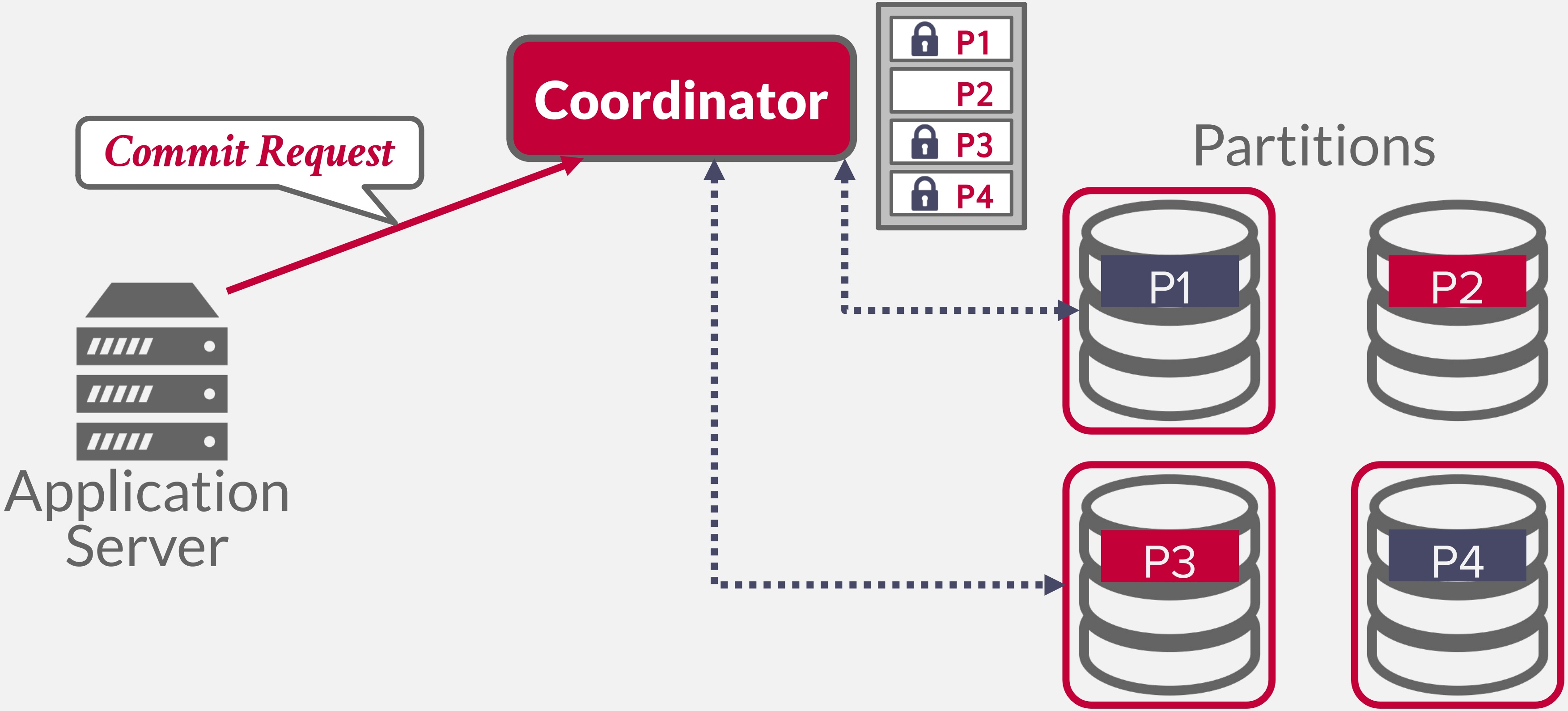
CENTRALIZED COORDINATOR



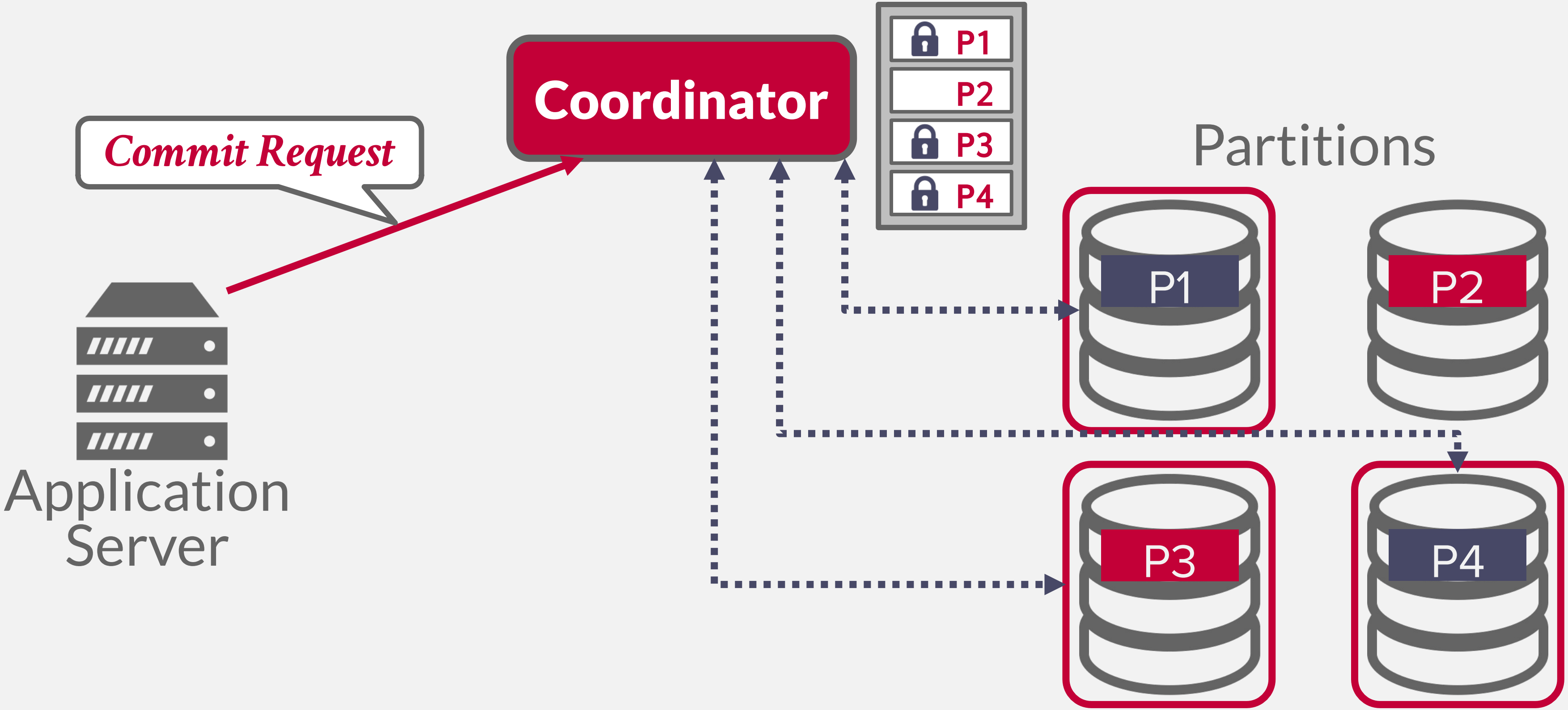
CENTRALIZED COORDINATOR



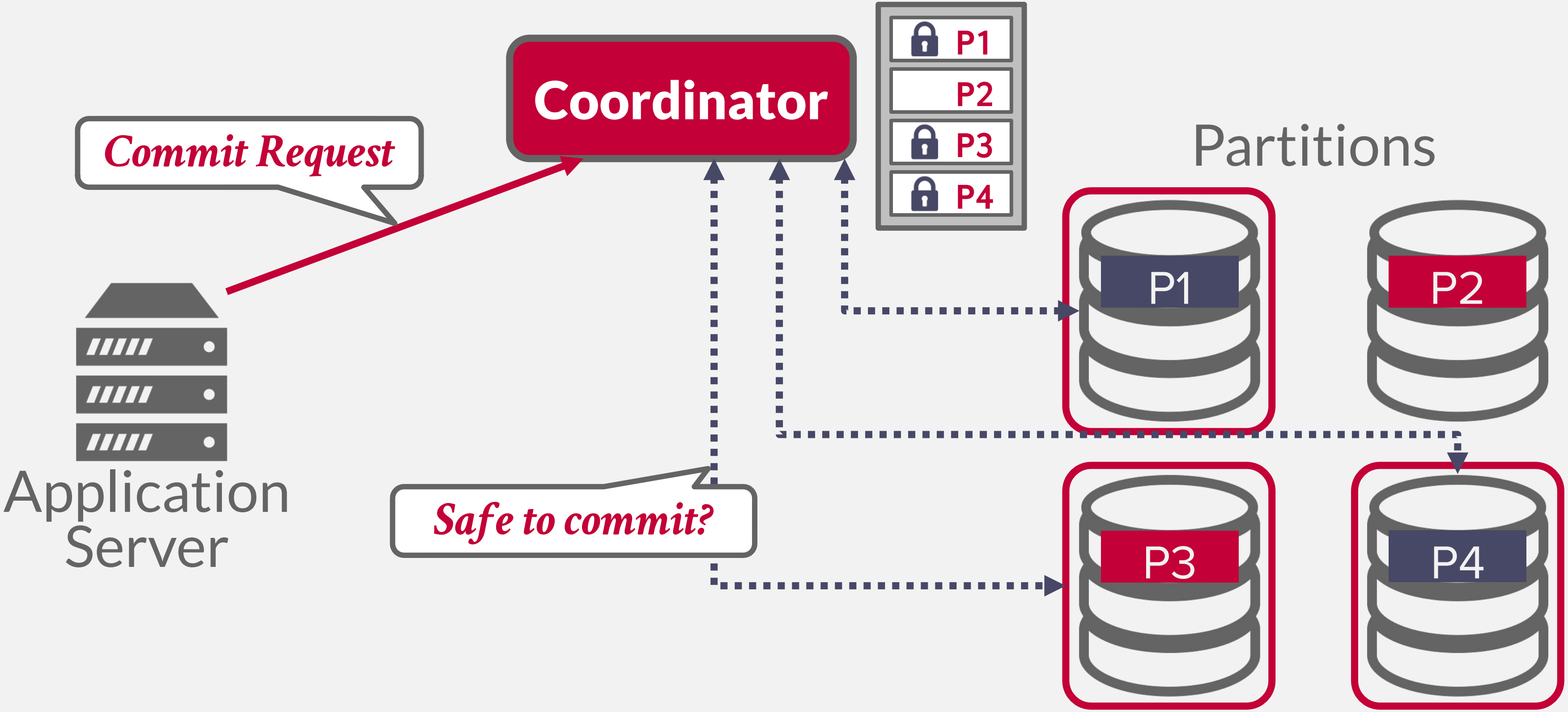
CENTRALIZED COORDINATOR



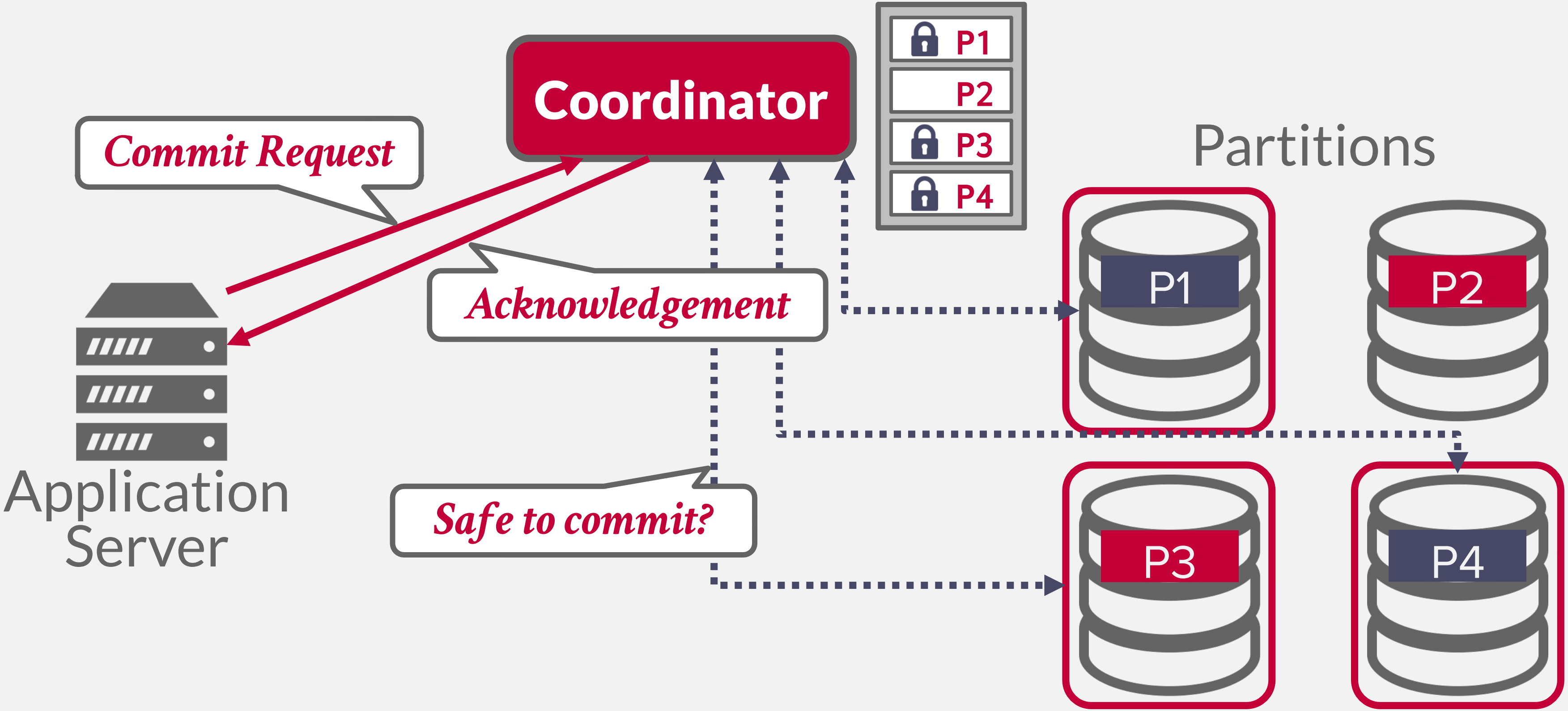
CENTRALIZED COORDINATOR



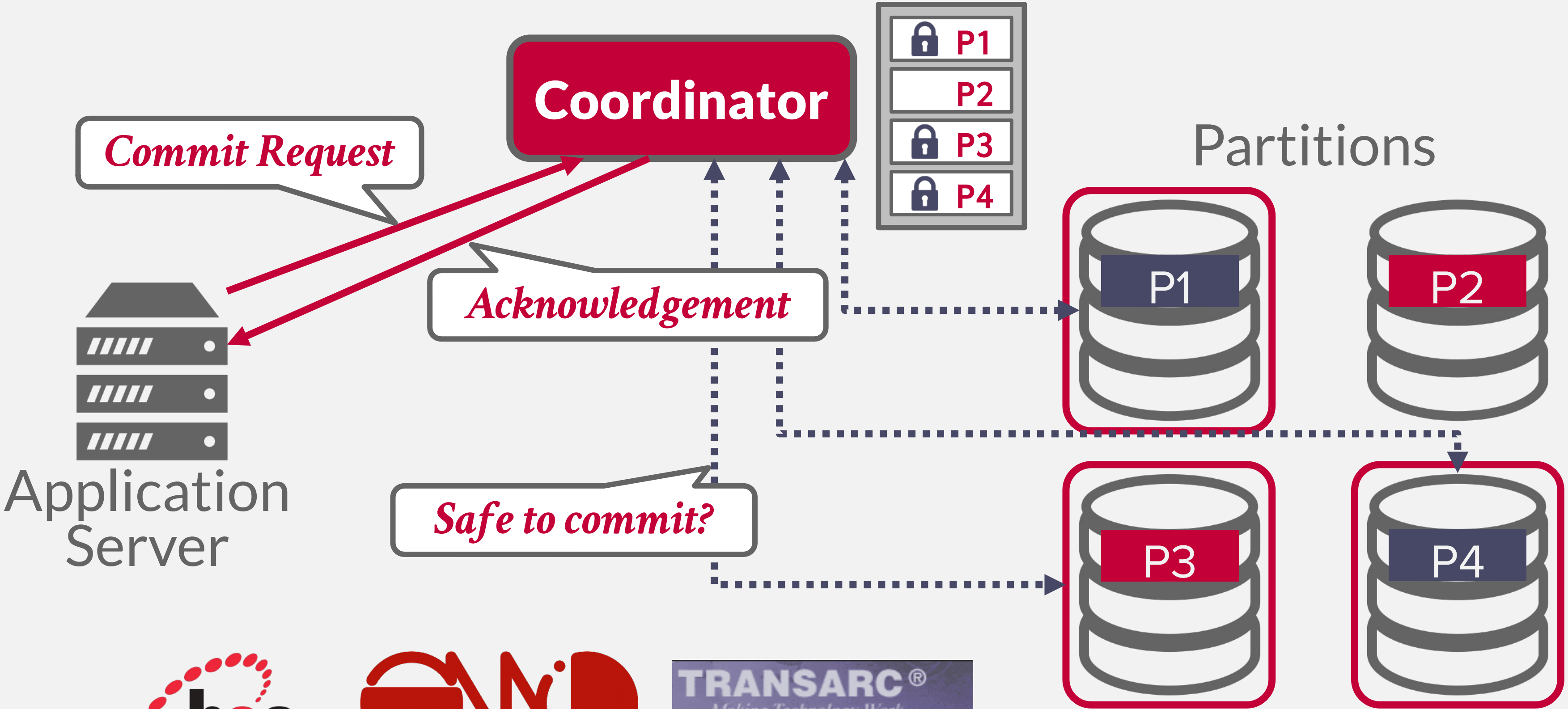
CENTRALIZED COORDINATOR



CENTRALIZED COORDINATOR



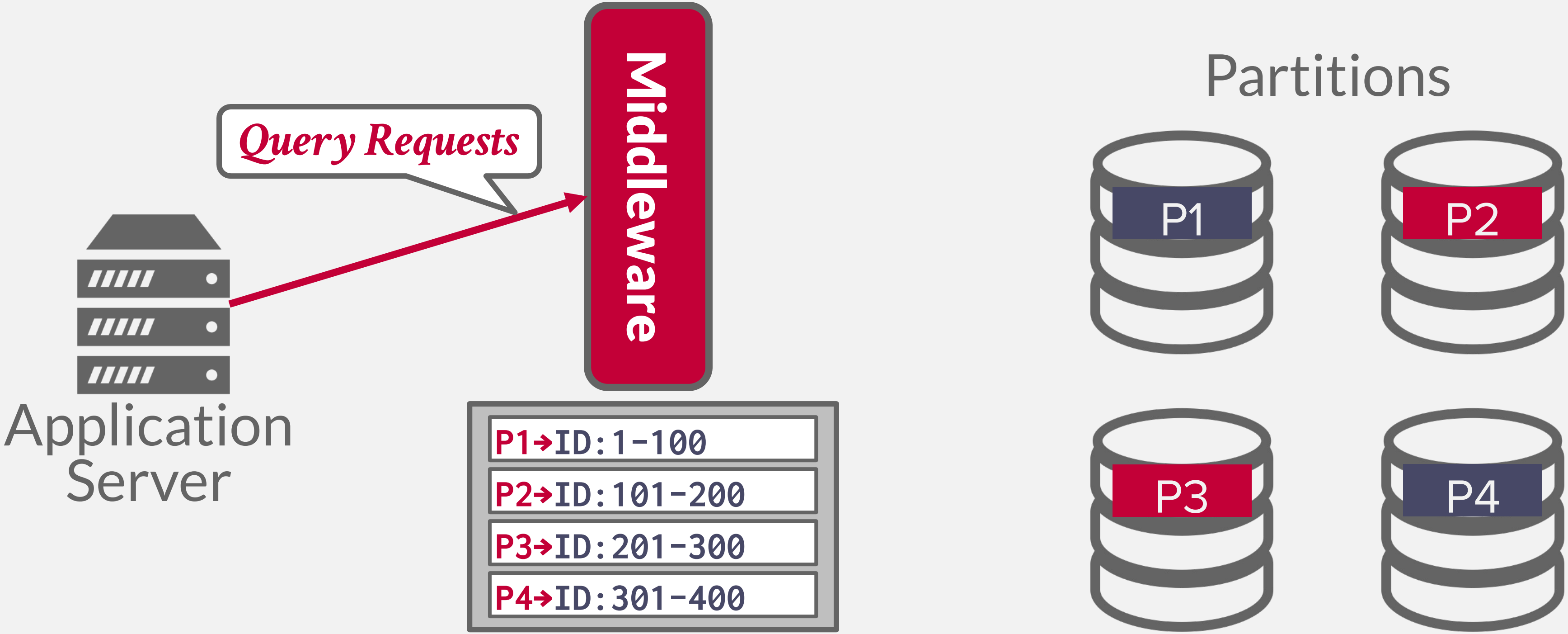
CENTRALIZED COORDINATOR



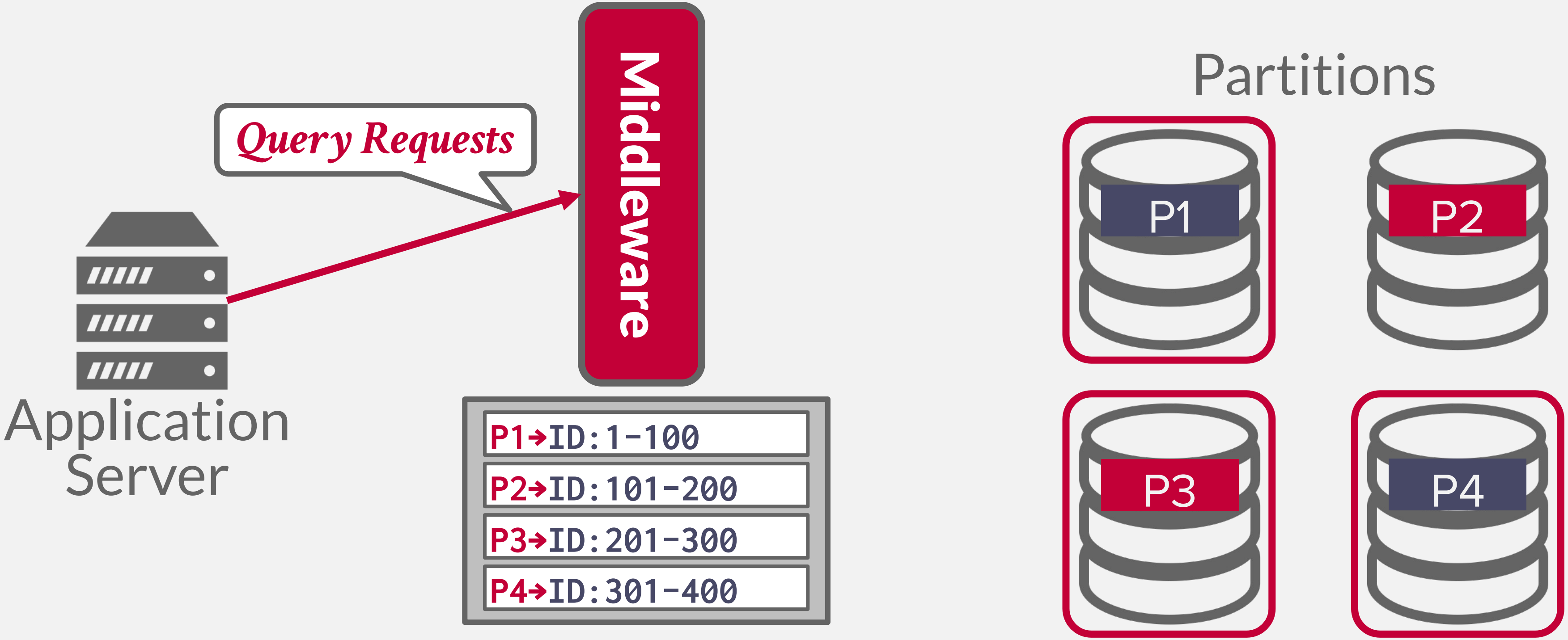
CENTRALIZED COORDINATOR

CENTRALIZED COORDINATOR

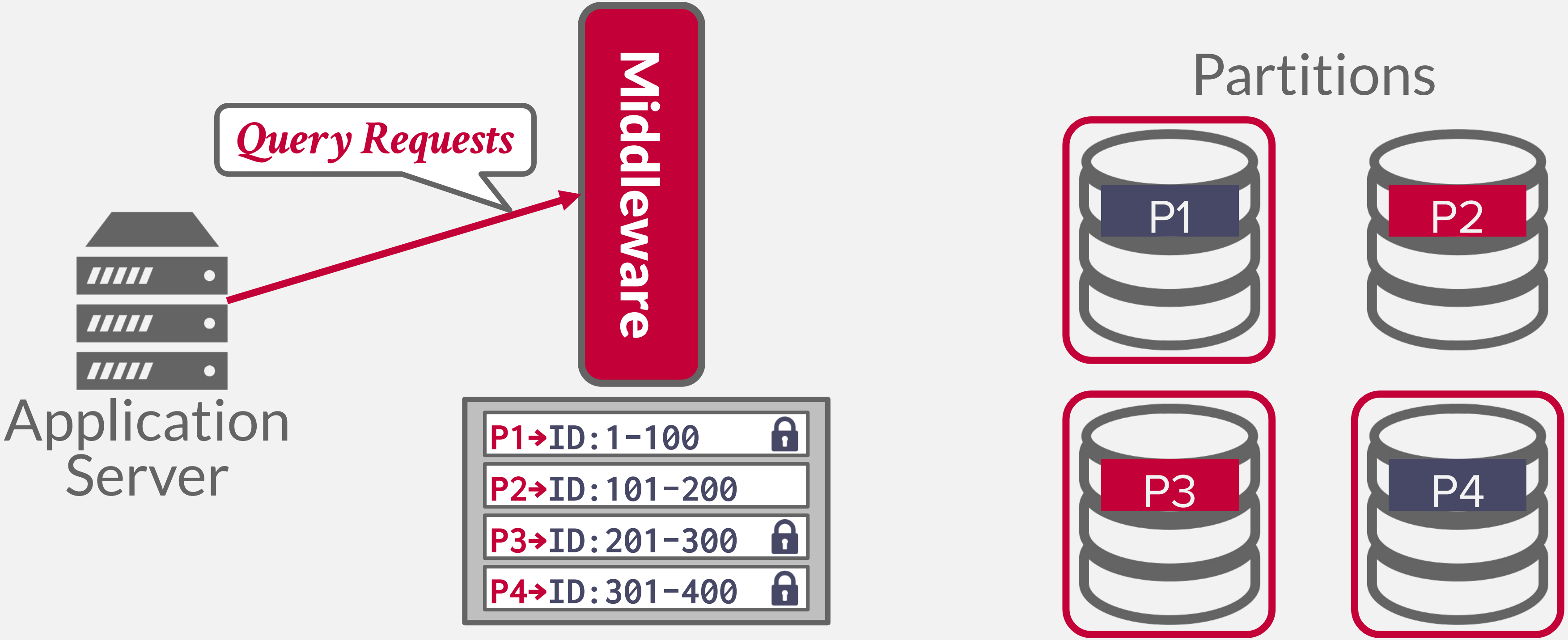
CENTRALIZED COORDINATOR



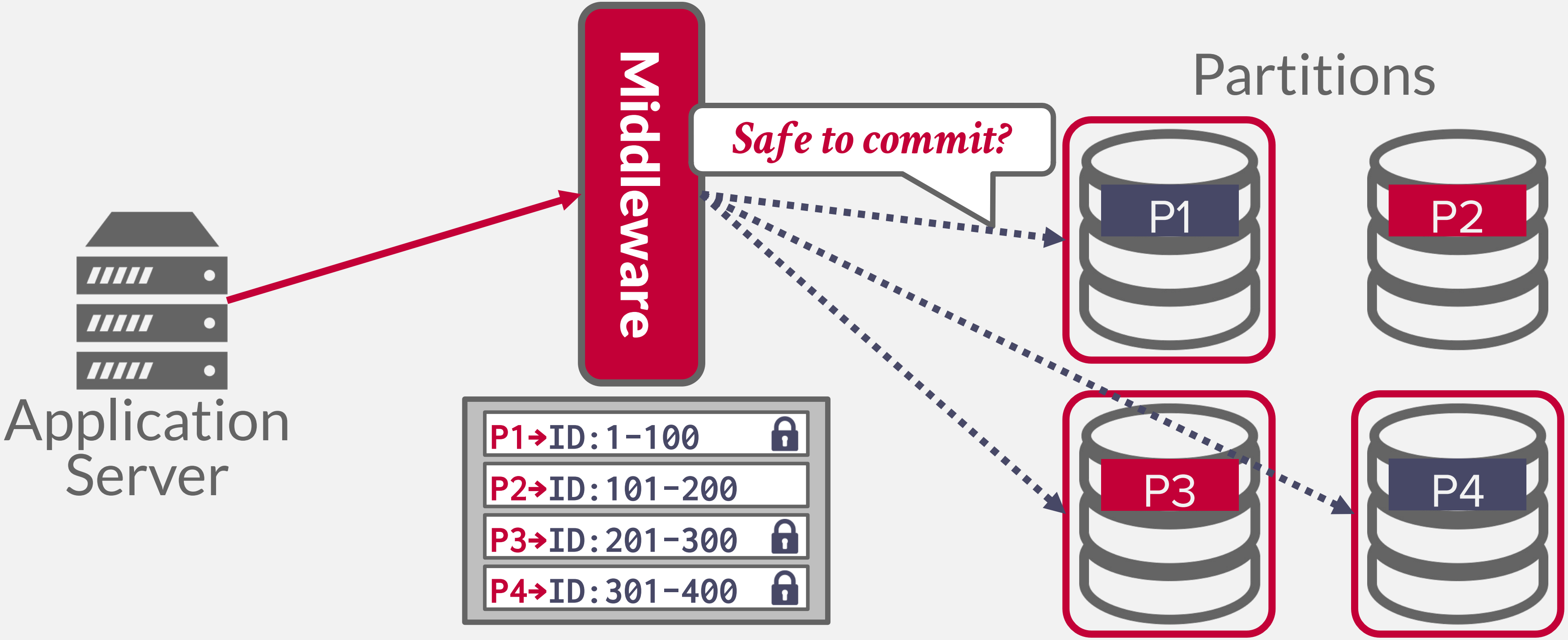
CENTRALIZED COORDINATOR



CENTRALIZED COORDINATOR

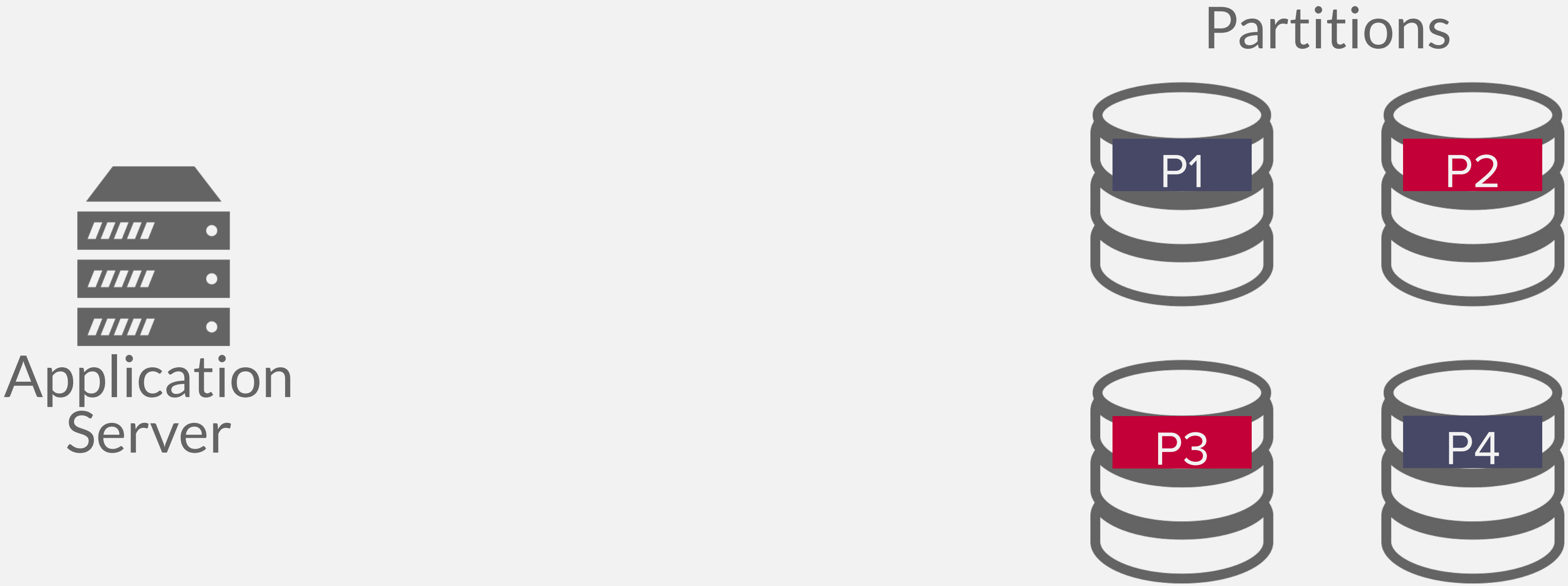


CENTRALIZED COORDINATOR



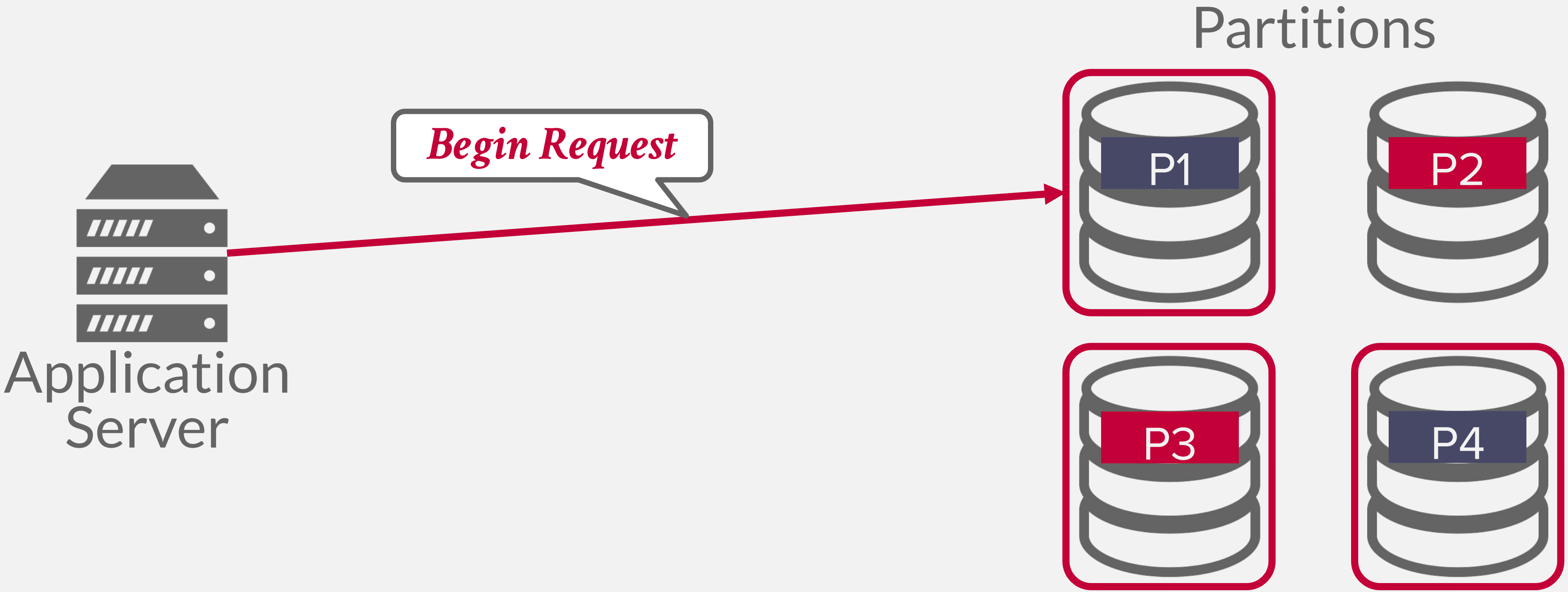
P1→ID:1-100	🔒
P2→ID:101-200	
P3→ID:201-300	🔒
P4→ID:301-400	🔒

DECENTRALIZED COORDINATOR

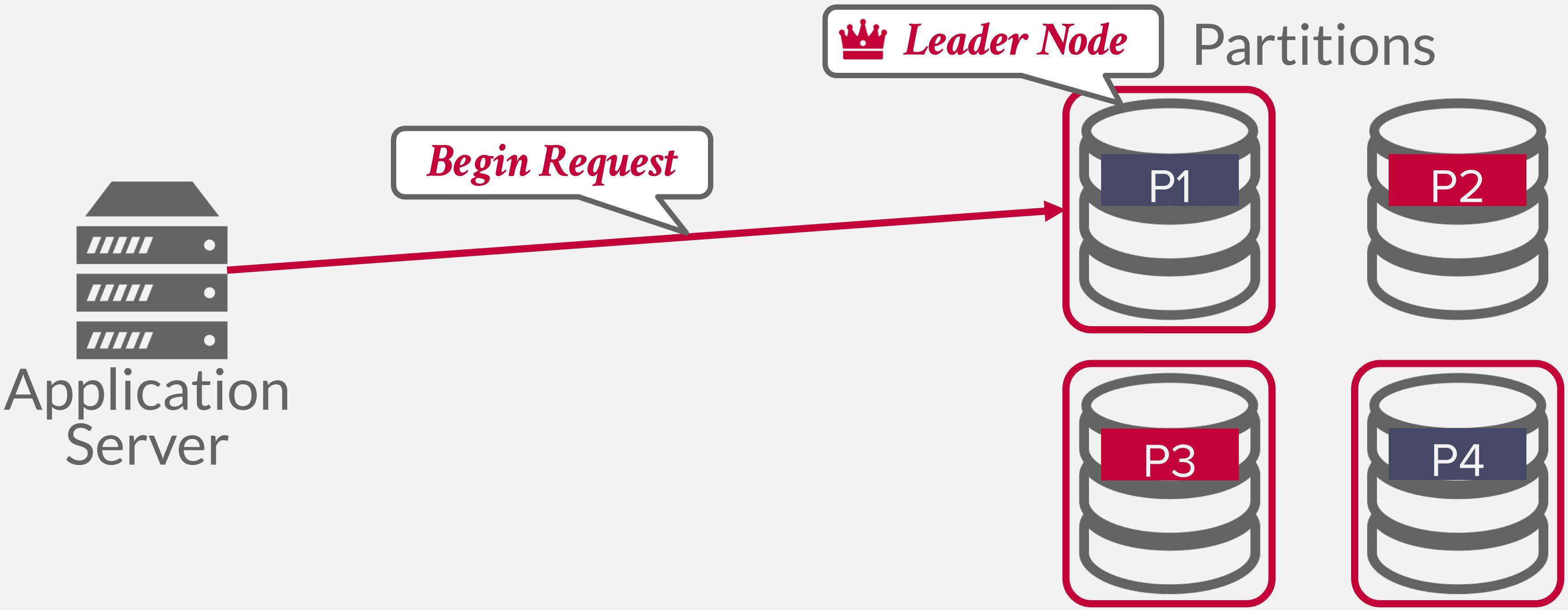


DECENTRALIZED COORDINATOR

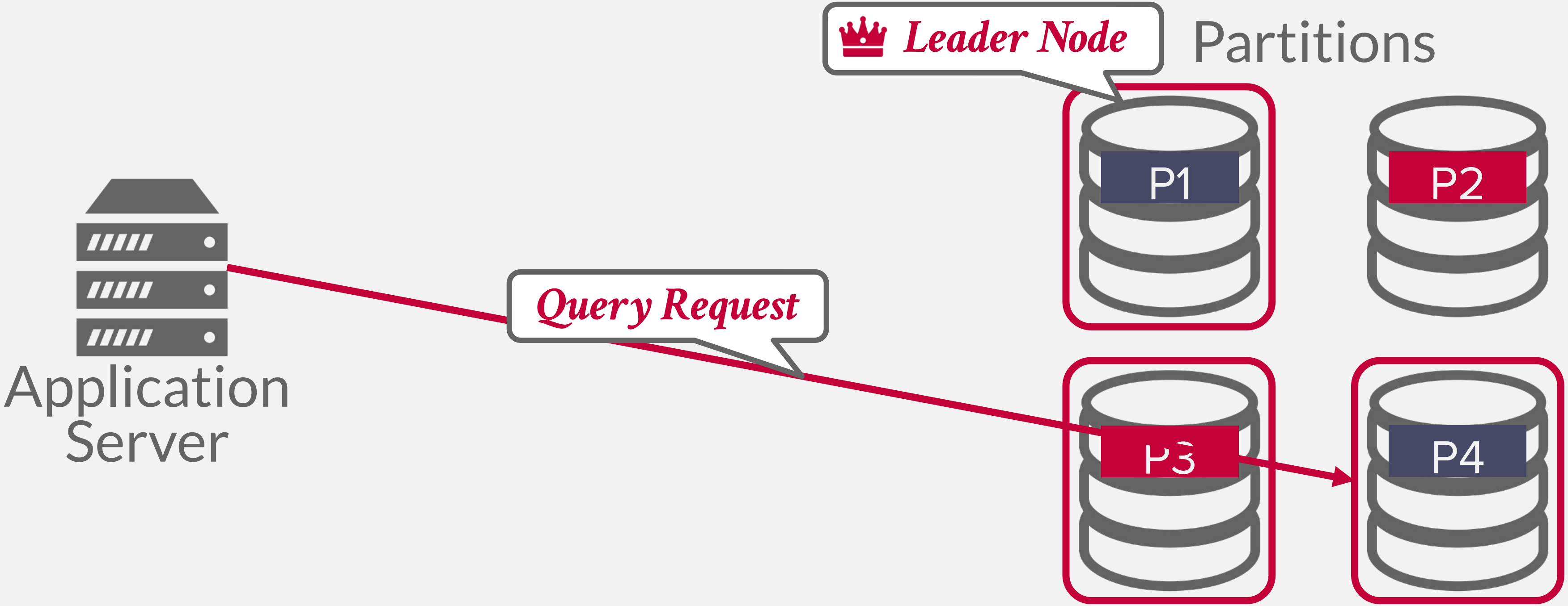
DECENTRALIZED COORDINATOR



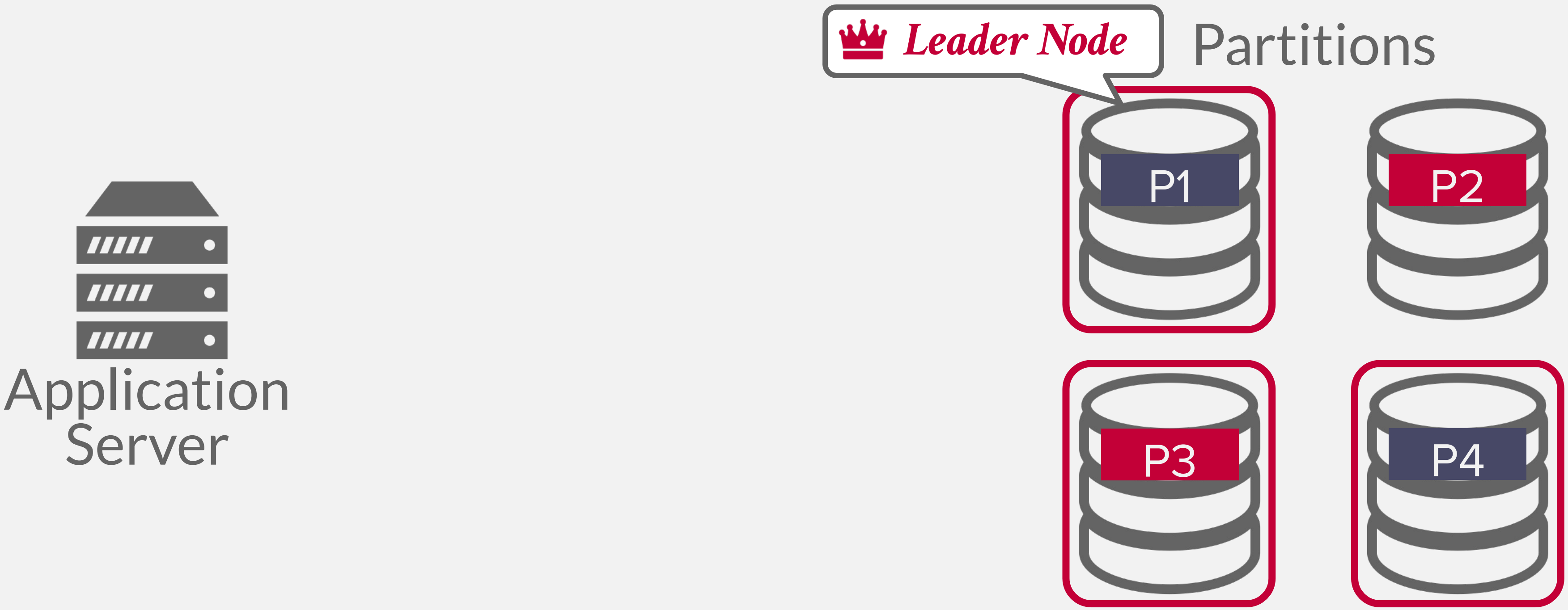
DECENTRALIZED COORDINATOR



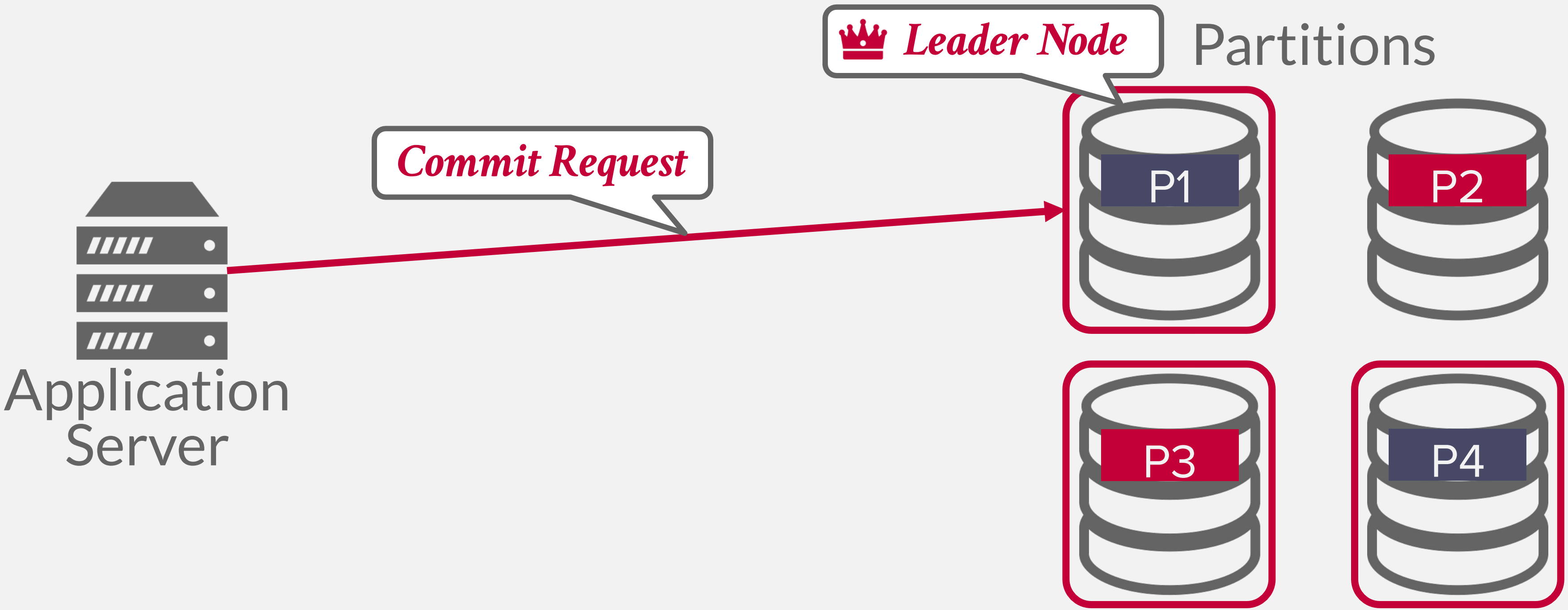
DECENTRALIZED COORDINATOR



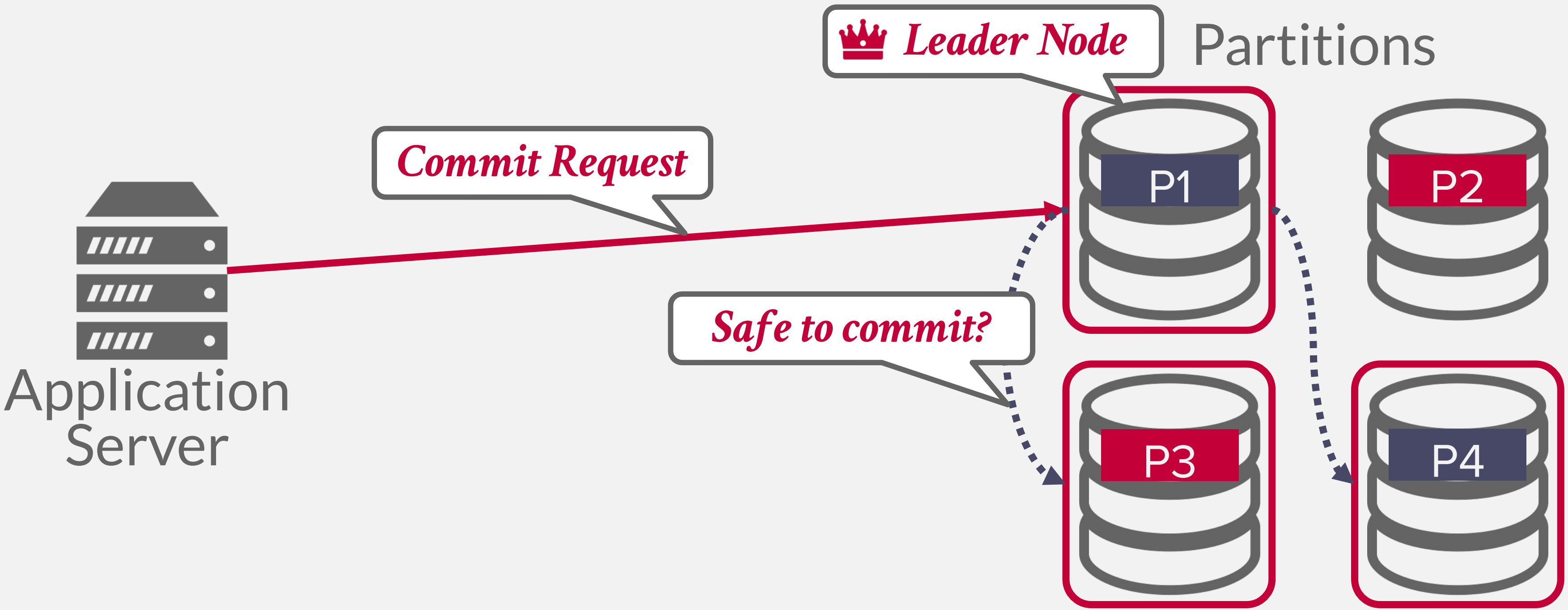
DECENTRALIZED COORDINATOR



DECENTRALIZED COORDINATOR



DECENTRALIZED COORDINATOR



OBSERVATION

We have assumed that the nodes in our distributed systems are running the same DBMS software.

But organizations often run many different DBMSs in their applications.

It would be nice if we could have a single interface for all our data.

FEDERATED DATABASES

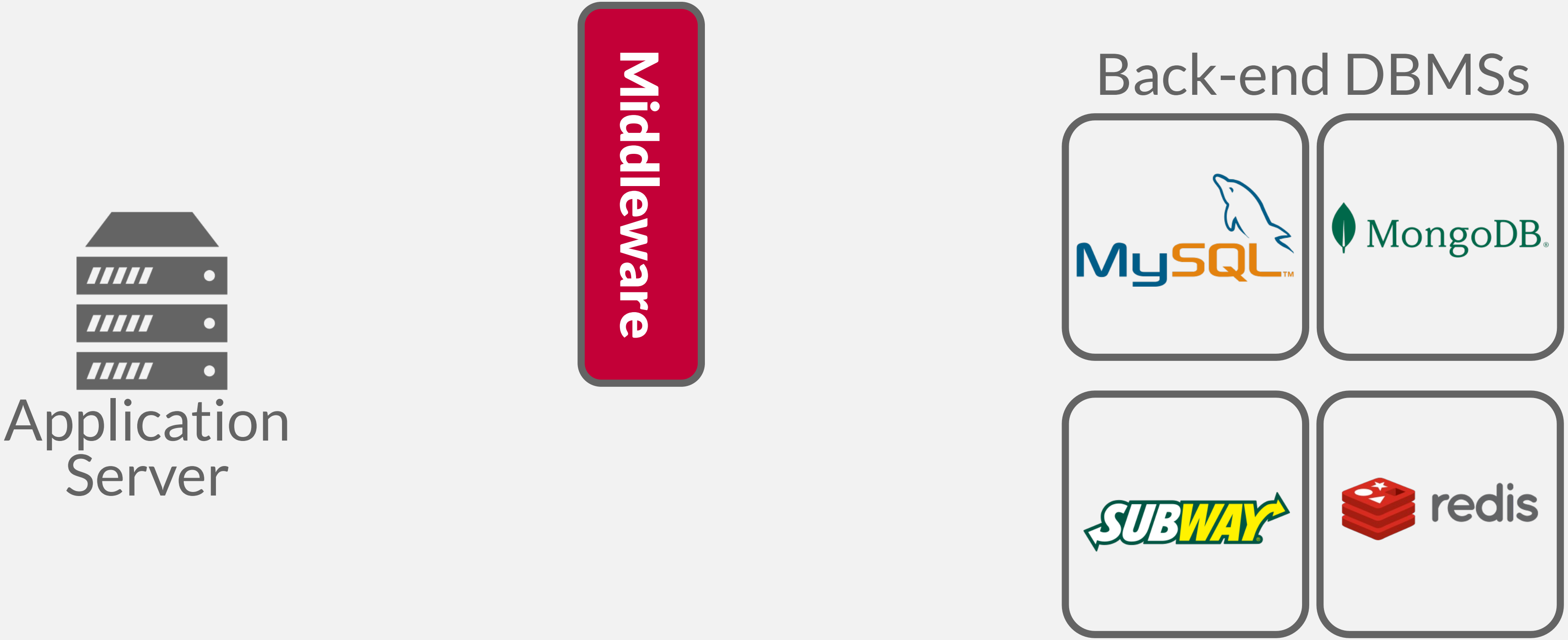
Distributed architecture that connects disparate DBMSs into a single logical system.

→ Expose a single query interface that can access data at any location.

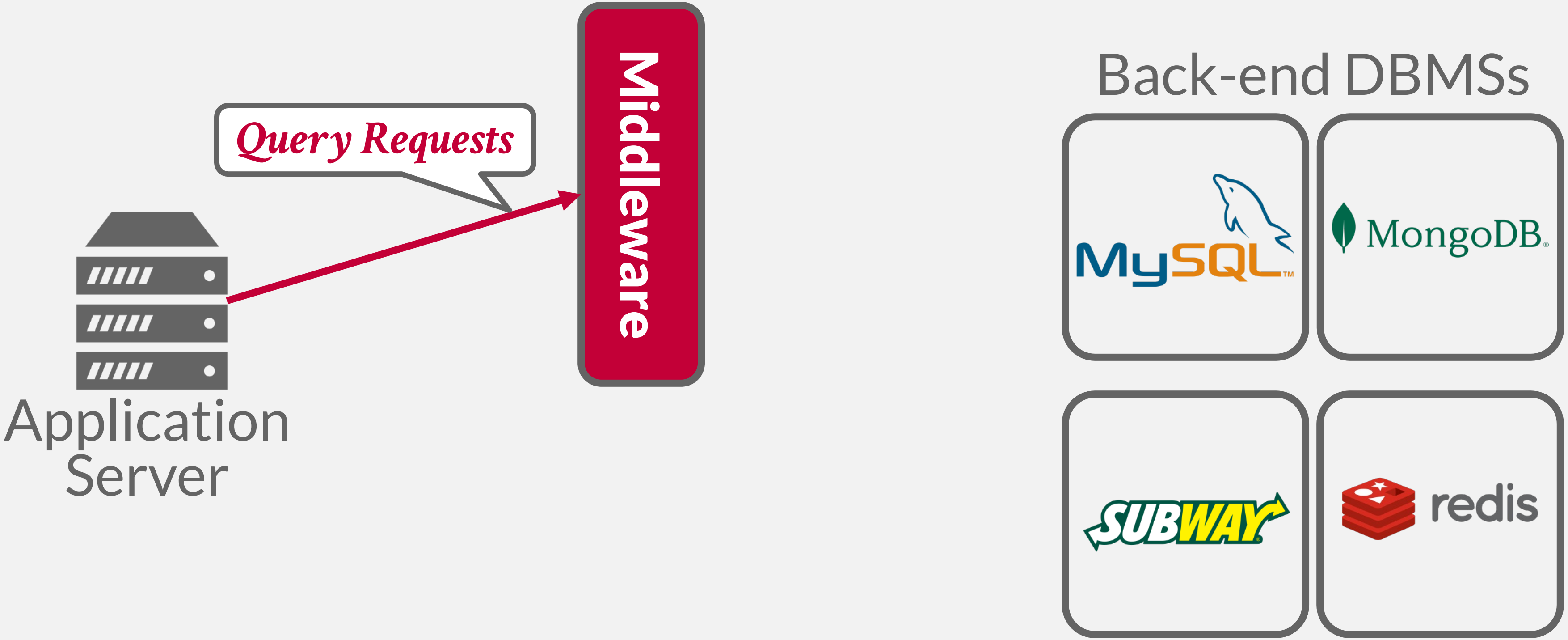
This is hard and nobody does it well

→ Different data models, query languages, limitations.
→ No easy way to optimize queries
→ Lots of data copying (bad).

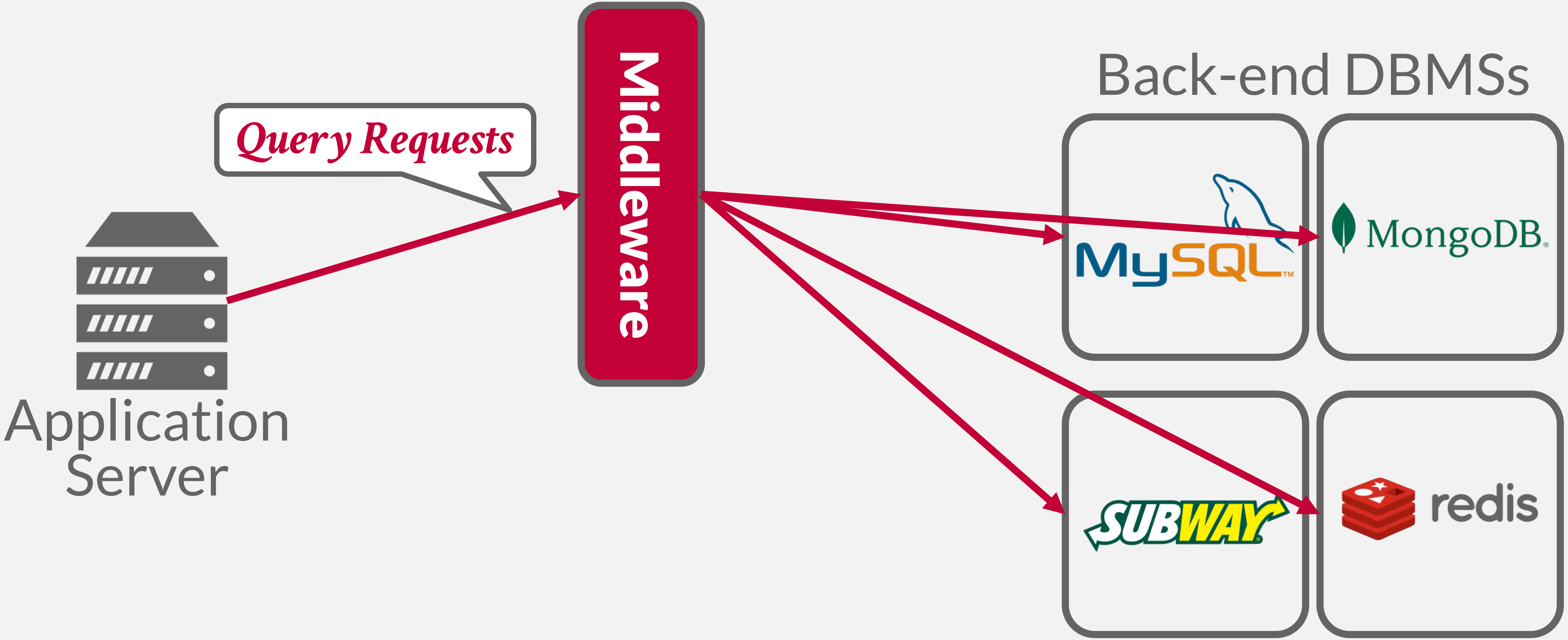
FEDERATED DATABASE EXAMPLE



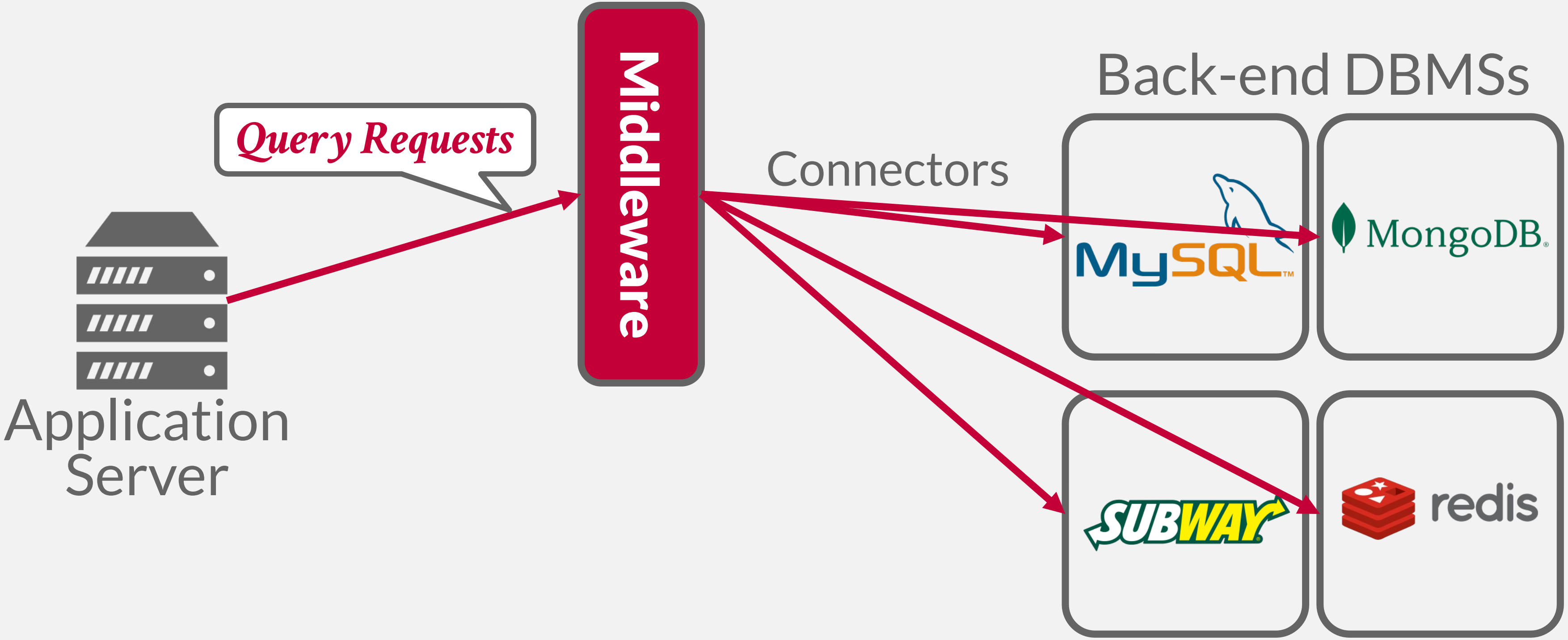
FEDERATED DATABASE EXAMPLE



FEDERATED DATABASE EXAMPLE



FEDERATED DATABASE EXAMPLE



DISTRIBUTED CONCURRENCY CONTROL

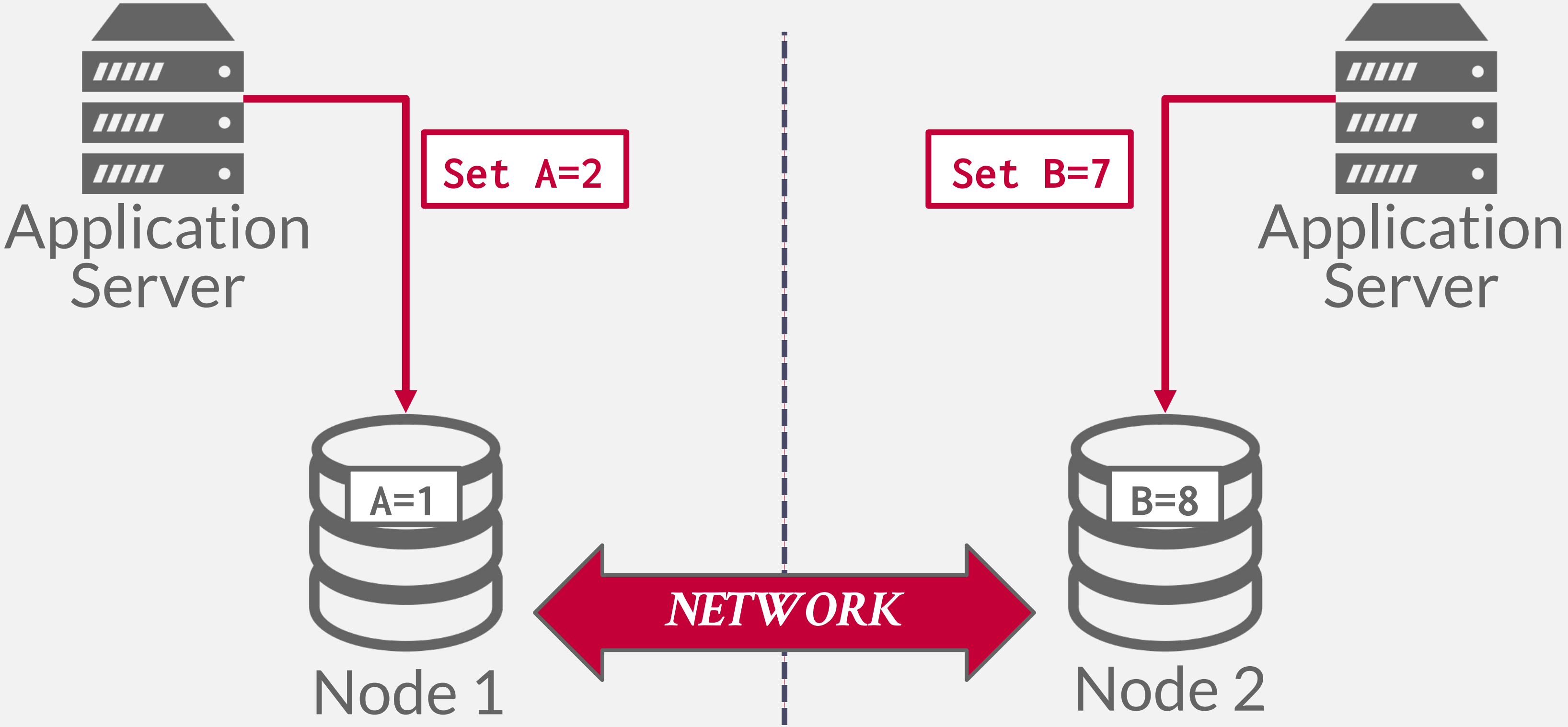
Need to allow multiple txns to execute simultaneously across multiple nodes.

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

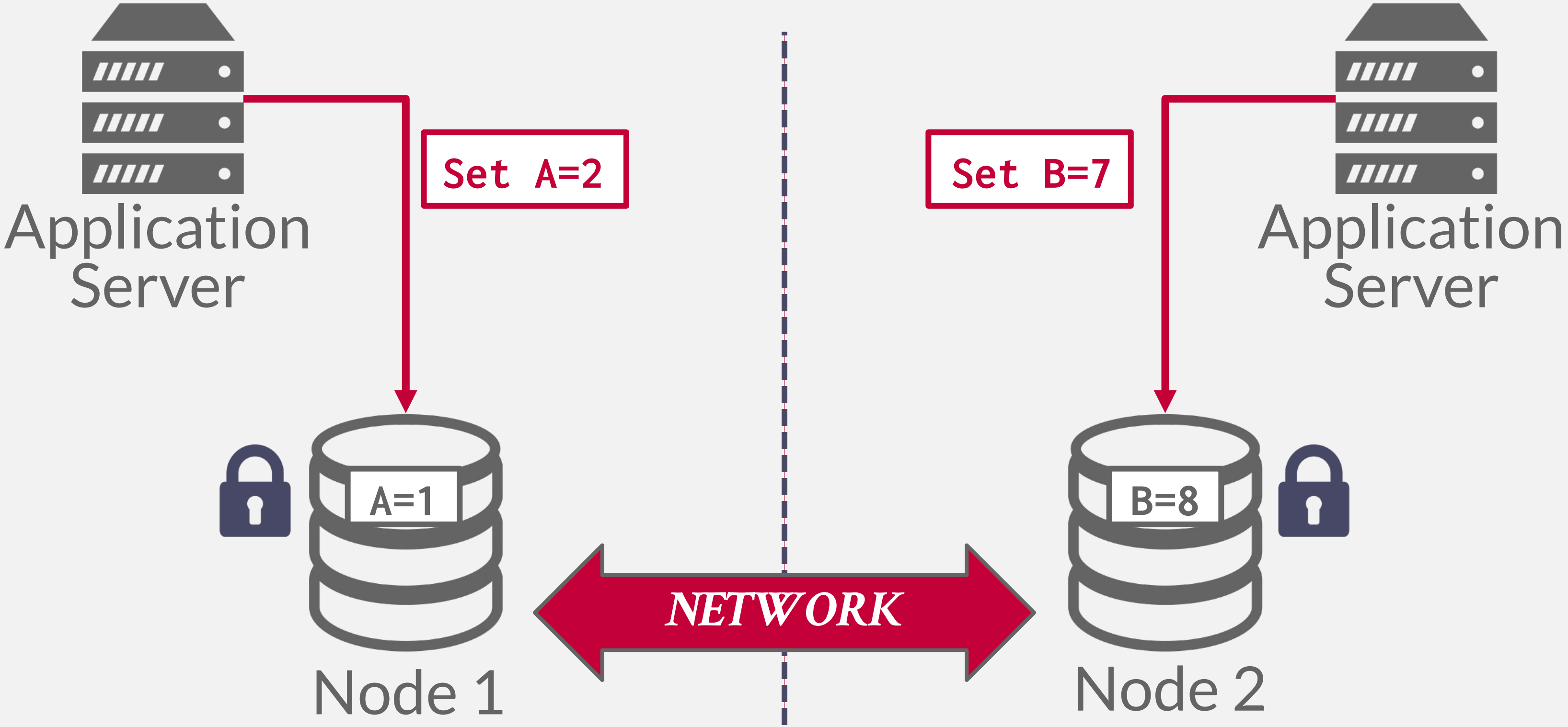
This is harder because of:

→ Replication.
→ Network Communication Overhead.
→ Node Failures (Permanent + Ephemeral).
→ Clock Skew.

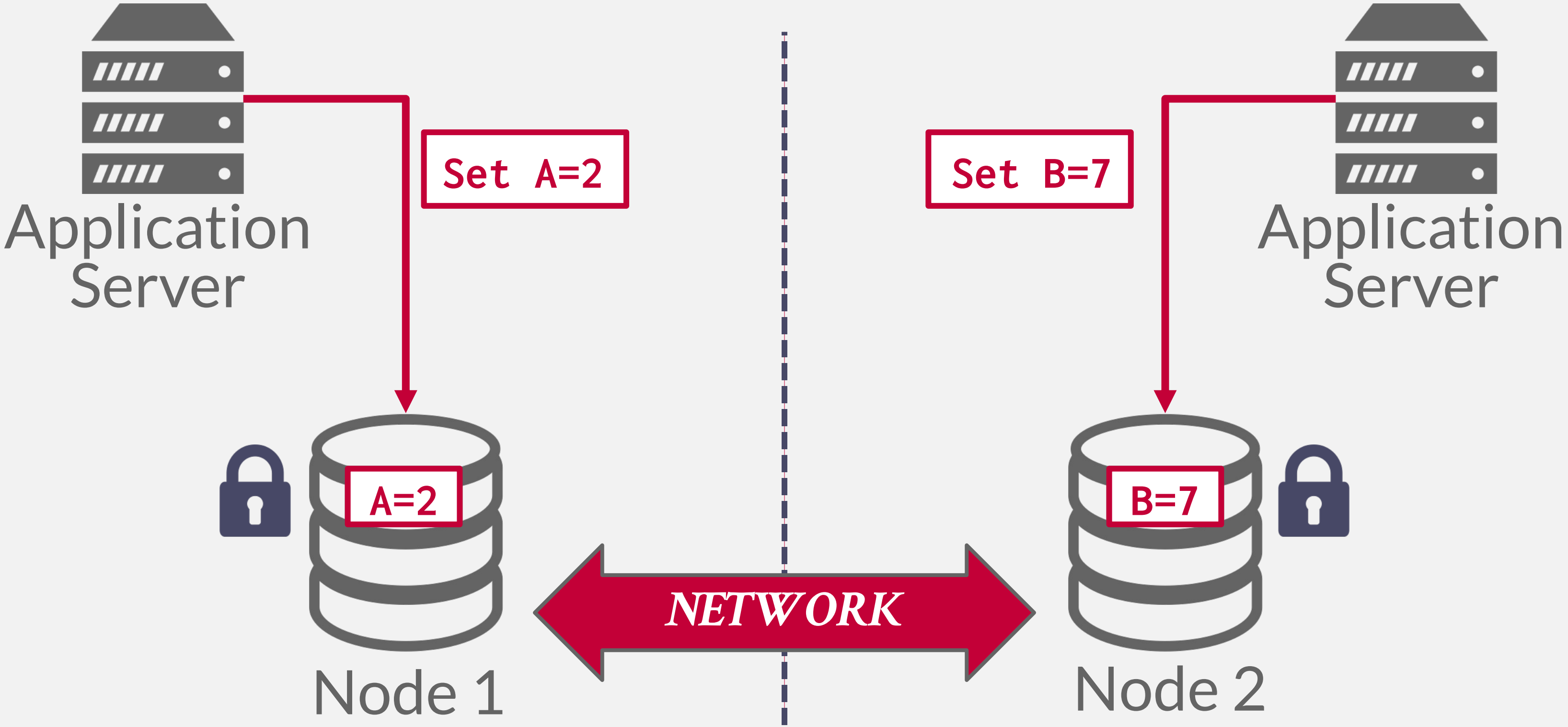
DISTRIBUTED 2PL



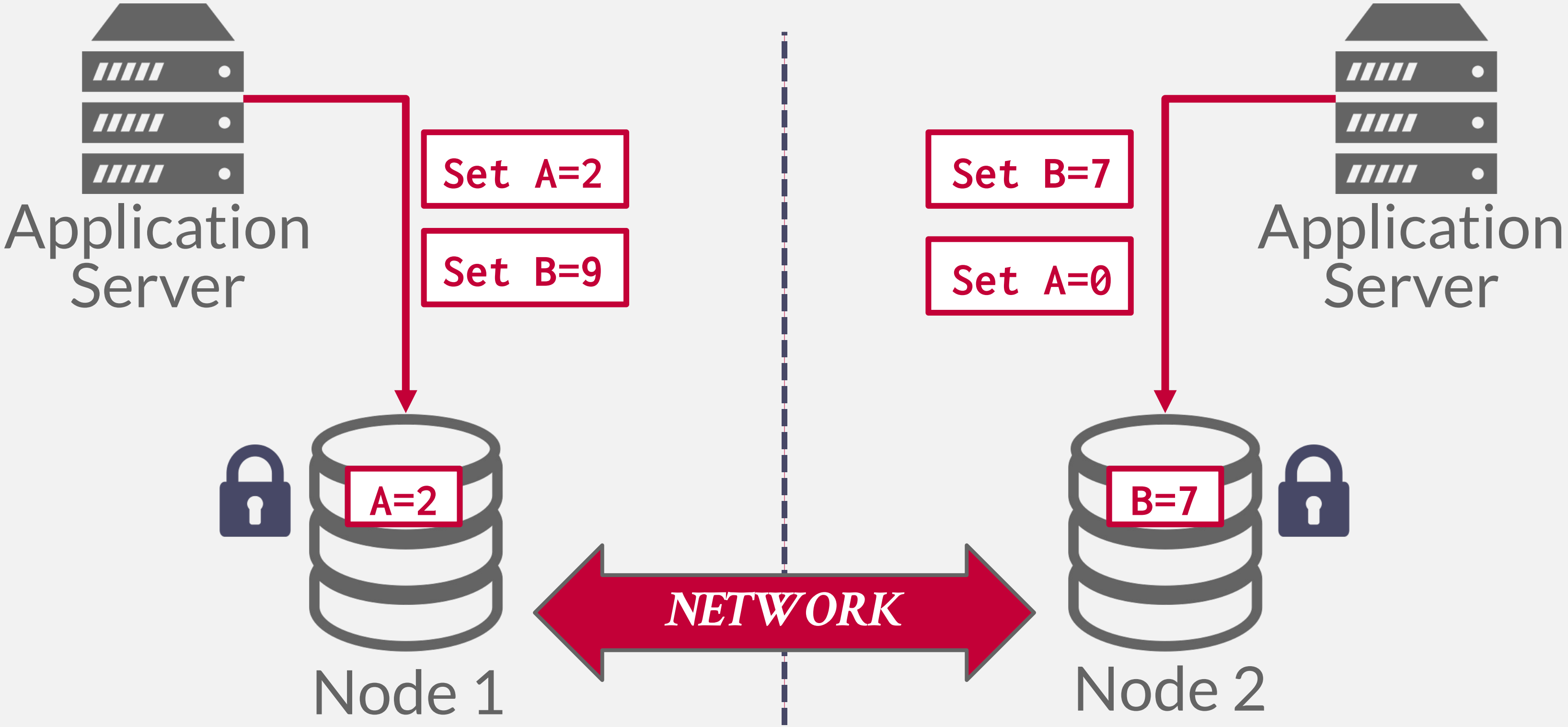
DISTRIBUTED 2PL



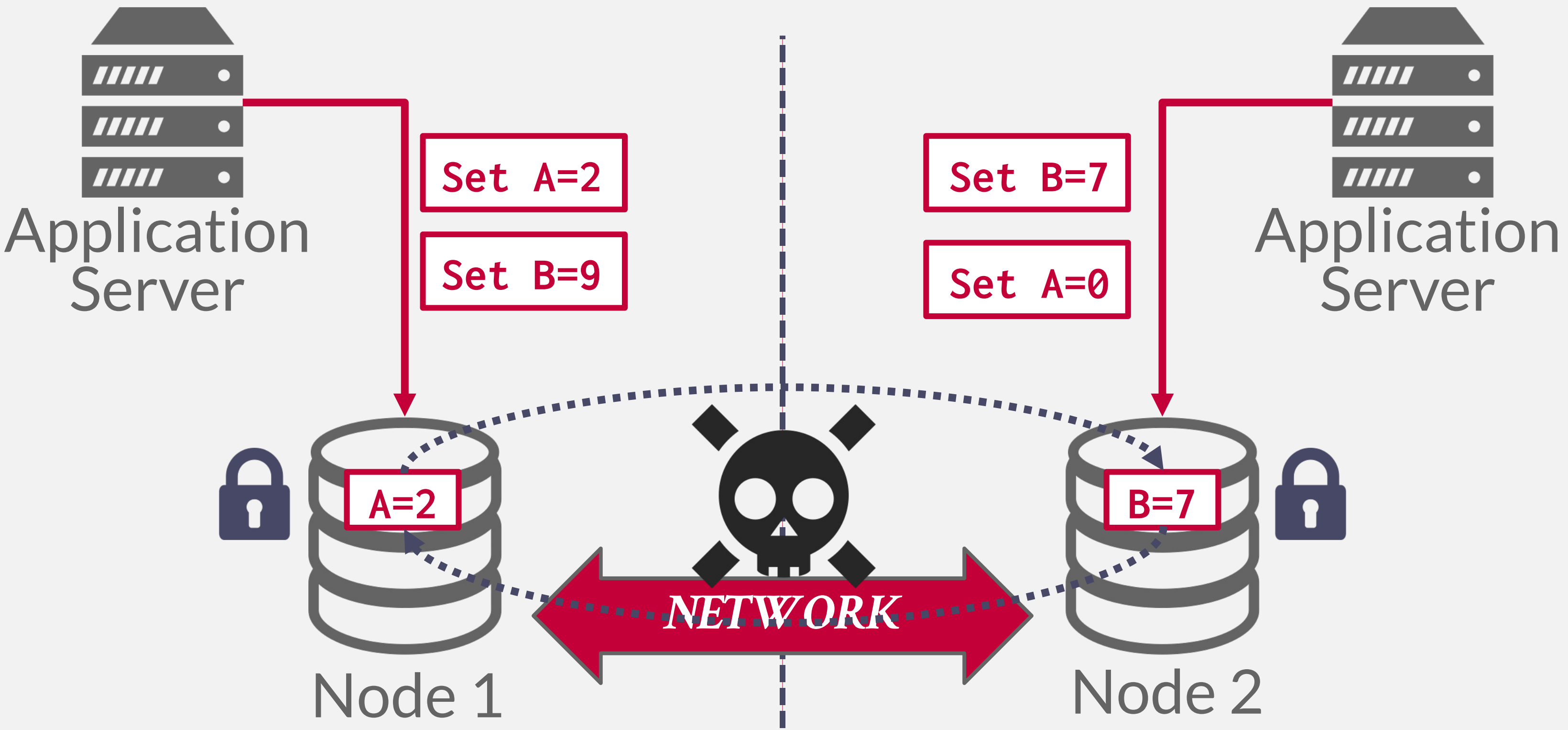
DISTRIBUTED 2PL



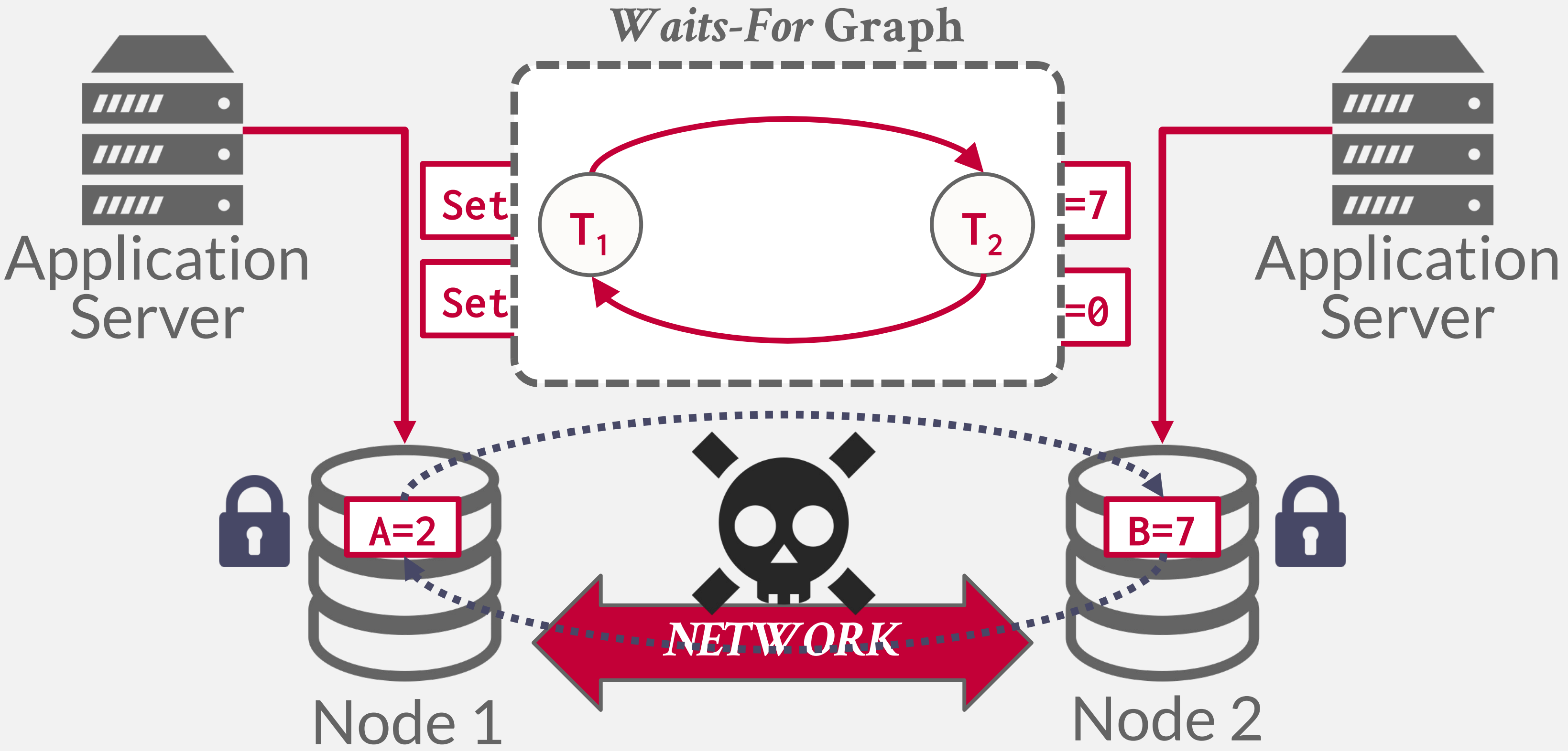
DISTRIBUTED 2PL



DISTRIBUTED 2PL



DISTRIBUTED 2PL



CONCLUSION

We 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