Carnegie Mellon University

DATABASE SYSTEMS

Distributed Databases I

LECTURE #23 » 15-445/645 FALL 2025 » PROF. ANDY PAVLO

ADMINISTRIVIA

Project #4 is due Sunday Dec 7th @ 11:59pm
→ Recitation Slides + Video (**@300**)

Homework #6 is due Sunday Dec 7th @ 11:59pm

Final Exam is on Thursday Dec 11th @ 1:00pm
→ Do not make travel plans before this date!

This course is recruiting TAs for the next semester
→ Apply at: https://www.ugrad.cs.cmu.edu/ta/S26/

UPCOMING DATABASE TALKS

XTDB (DB Seminar)
→ Monday Nov 24th @ 12:00pm
→ Zoom

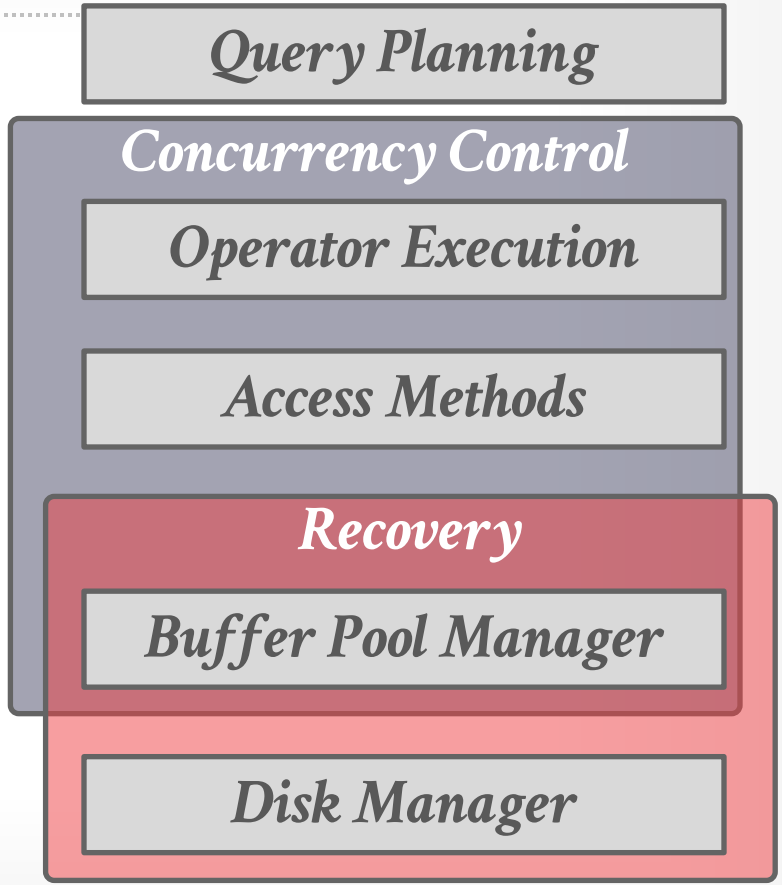
Apache Polaris (DB Seminar)
→ Monday Dec 1st @ 12:00pm
→ Zoom

Apache Fluss (DB Seminar)
→ Monday Dec 7th @ 12:00pm
→ Zoom

COURSE OUTLINE

Databases are hard.

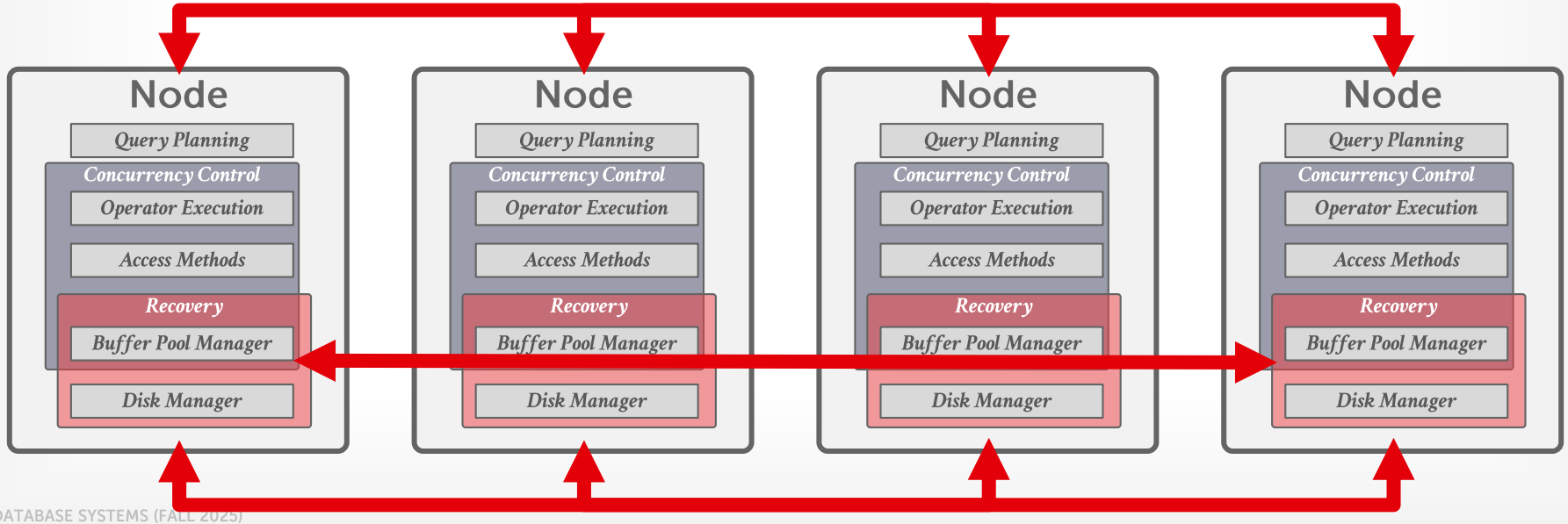
Distributed databases are harder.



COURSE OUTLINE

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

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 do we divide the database across resources?

How does the DBMS ensure correctness?

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 do we divide the database across resources?

How does the DBMS ensure correctness?

TODAY'S AGENDA

System Architectures

Partitioning Schemes

Replication

Distributed Concurrency Control

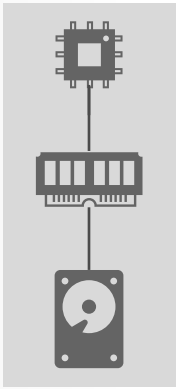
SYSTEM ARCHITECTURE

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.

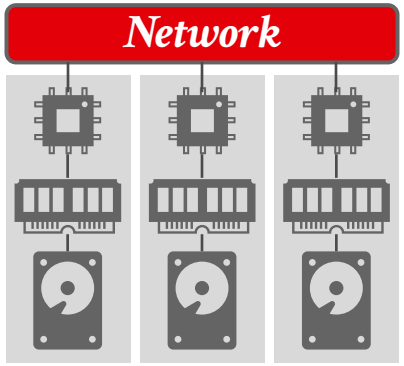
SYSTEM ARCHITECTURE

Shared-Everything



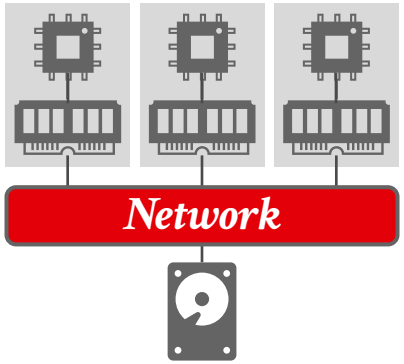
Most Common

Shared-Nothing



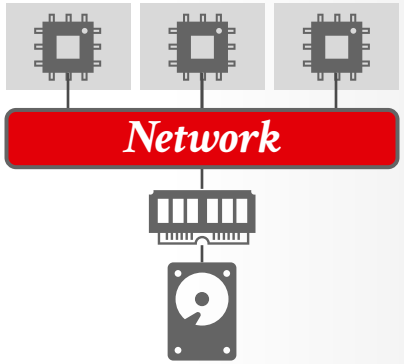
Common (Decreasing)

Shared-Disk



Common (Increasing)

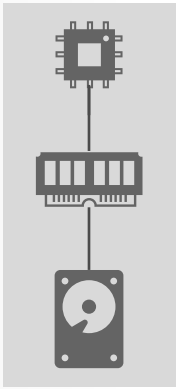
Shared-Memory



Non-Existent?

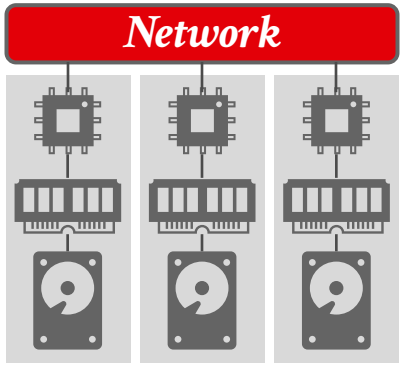
SYSTEM ARCHITECTURE

Shared-Everything



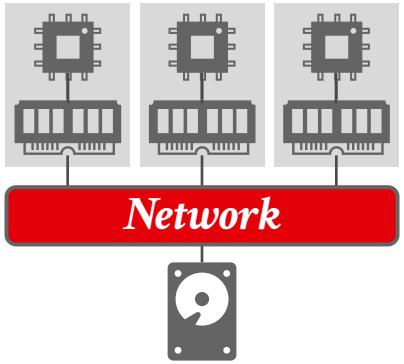
Most Common

Shared-Nothing



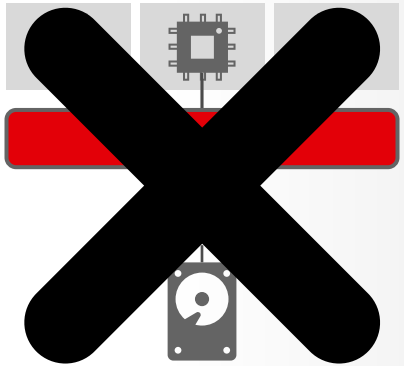
Common (Decreasing)

Shared-Disk



Common (Increasing)

Shared-Memory



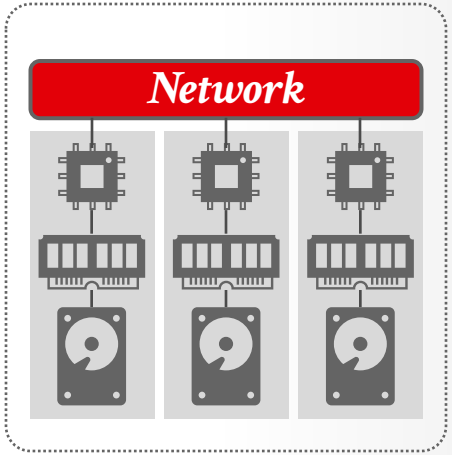
Non-Existent?

SHARED-NOTHING

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

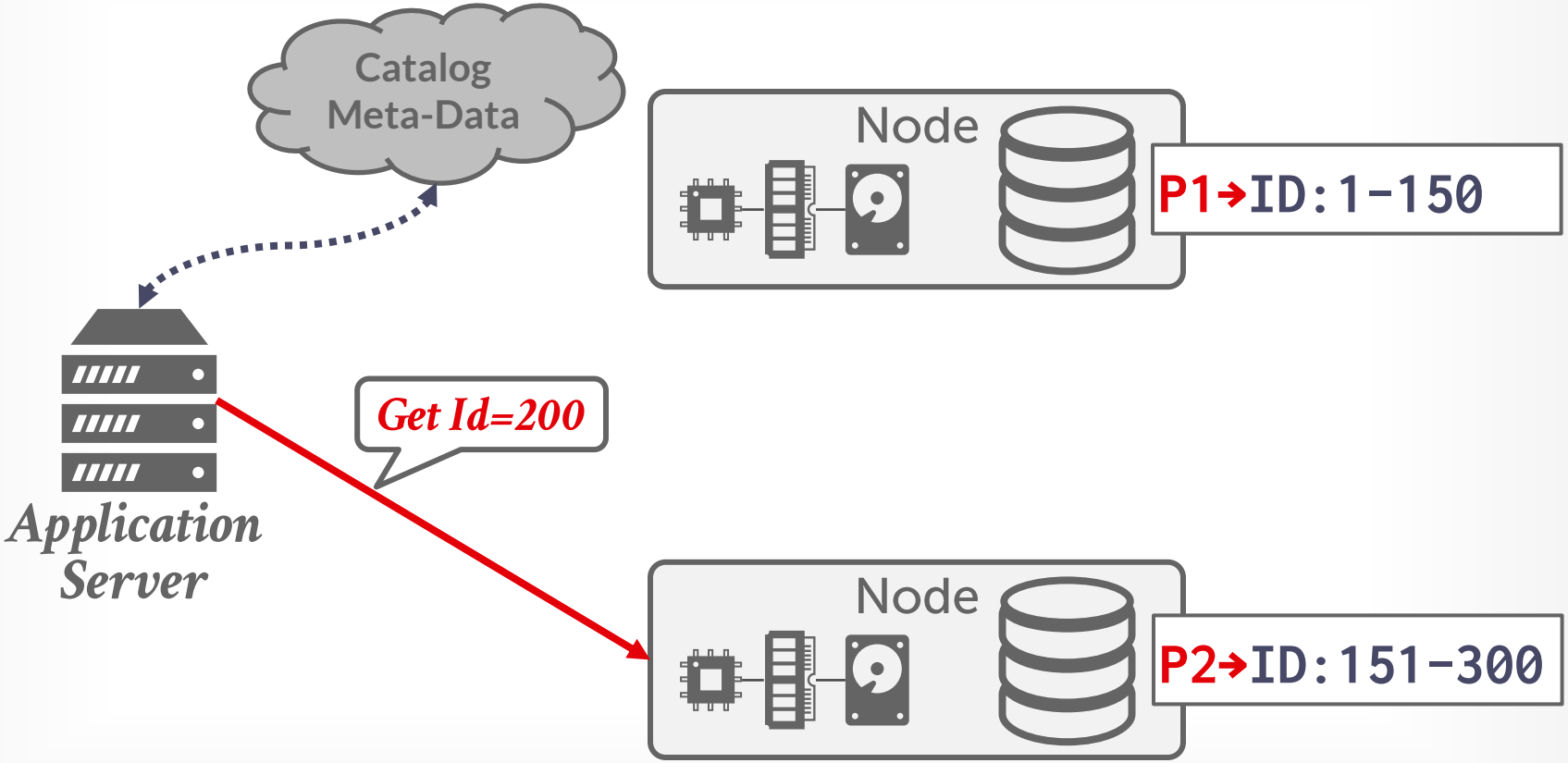
Nodes only communicate with each other via network.

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

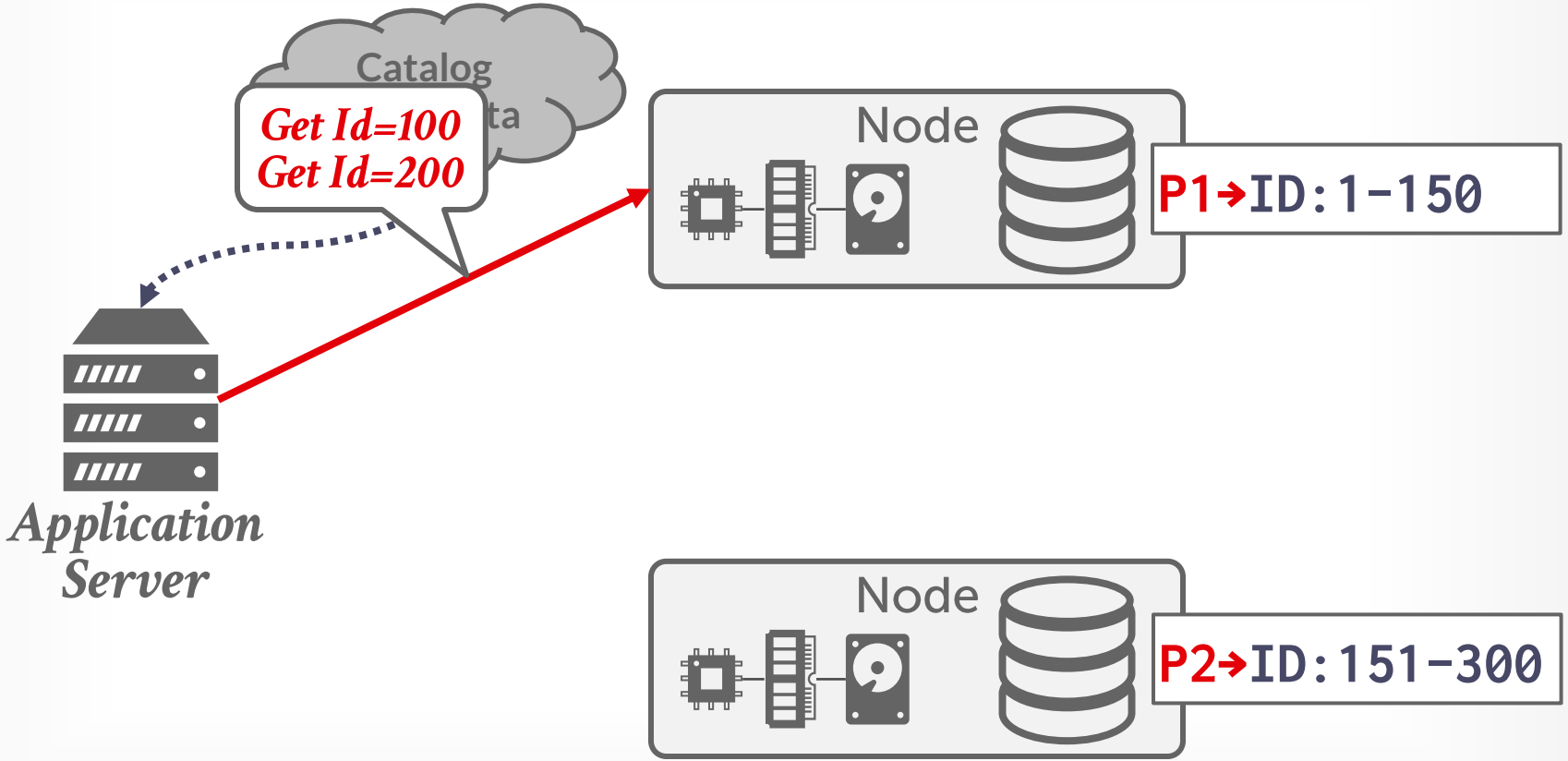


kinetica TigerBeetle ORACLE NOSQL DATABASE Yellowbrick MEMCACHED elastic etcd
SciDB fauna Dgraph CockroachDB Exasol MongoDB. citusdata
FOUNDATIONDB ClickHouse cassandra H-Store VOLTDB CrateDB
Comdb2 APACHE GEODE orbit db APACHE ZooKeeper Greenplum AEROSPIKE SingleStore VERTICA
RonDB TiDB redis TERADATA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

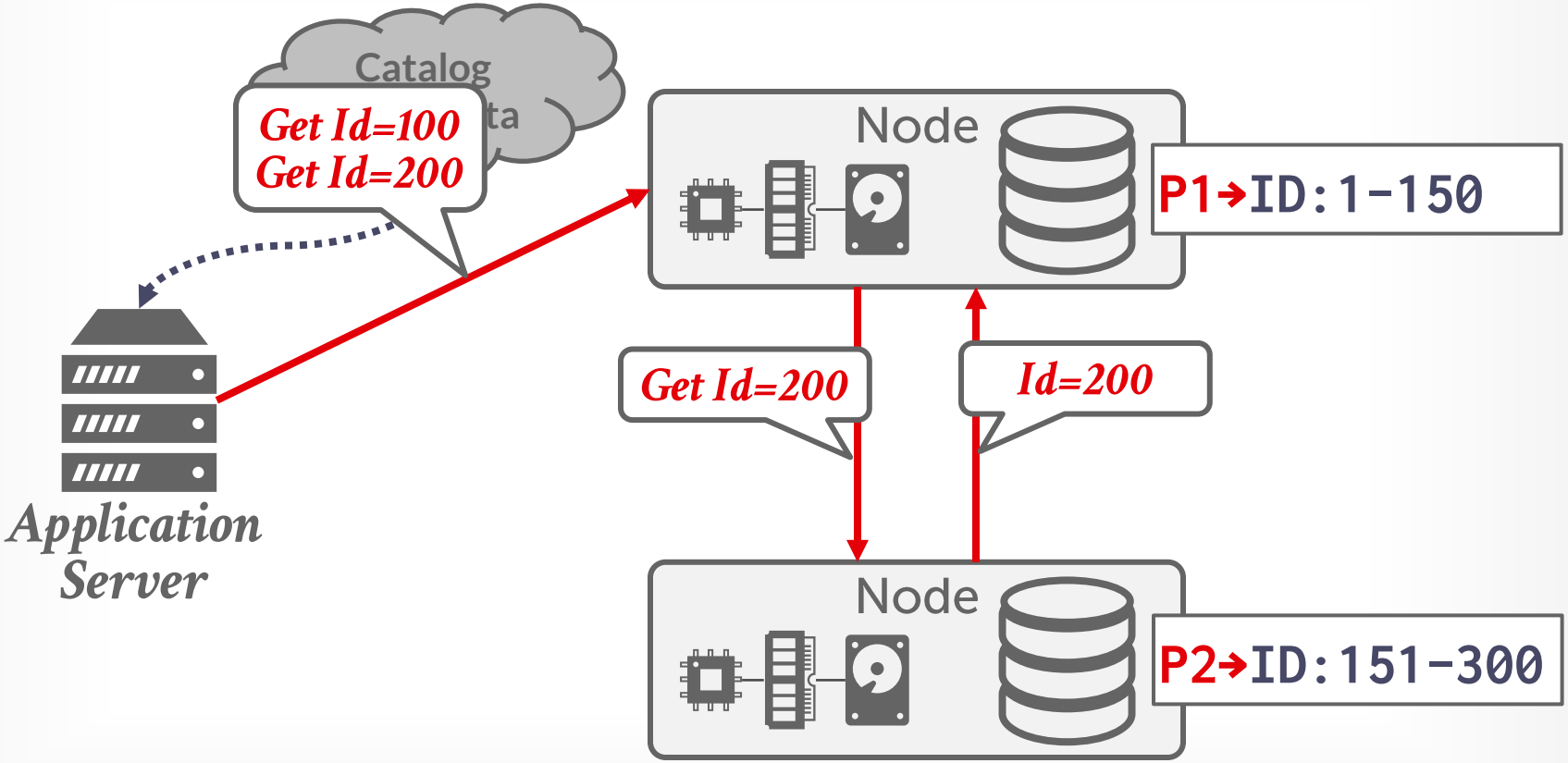
SHARED-NOTHING EXAMPLE



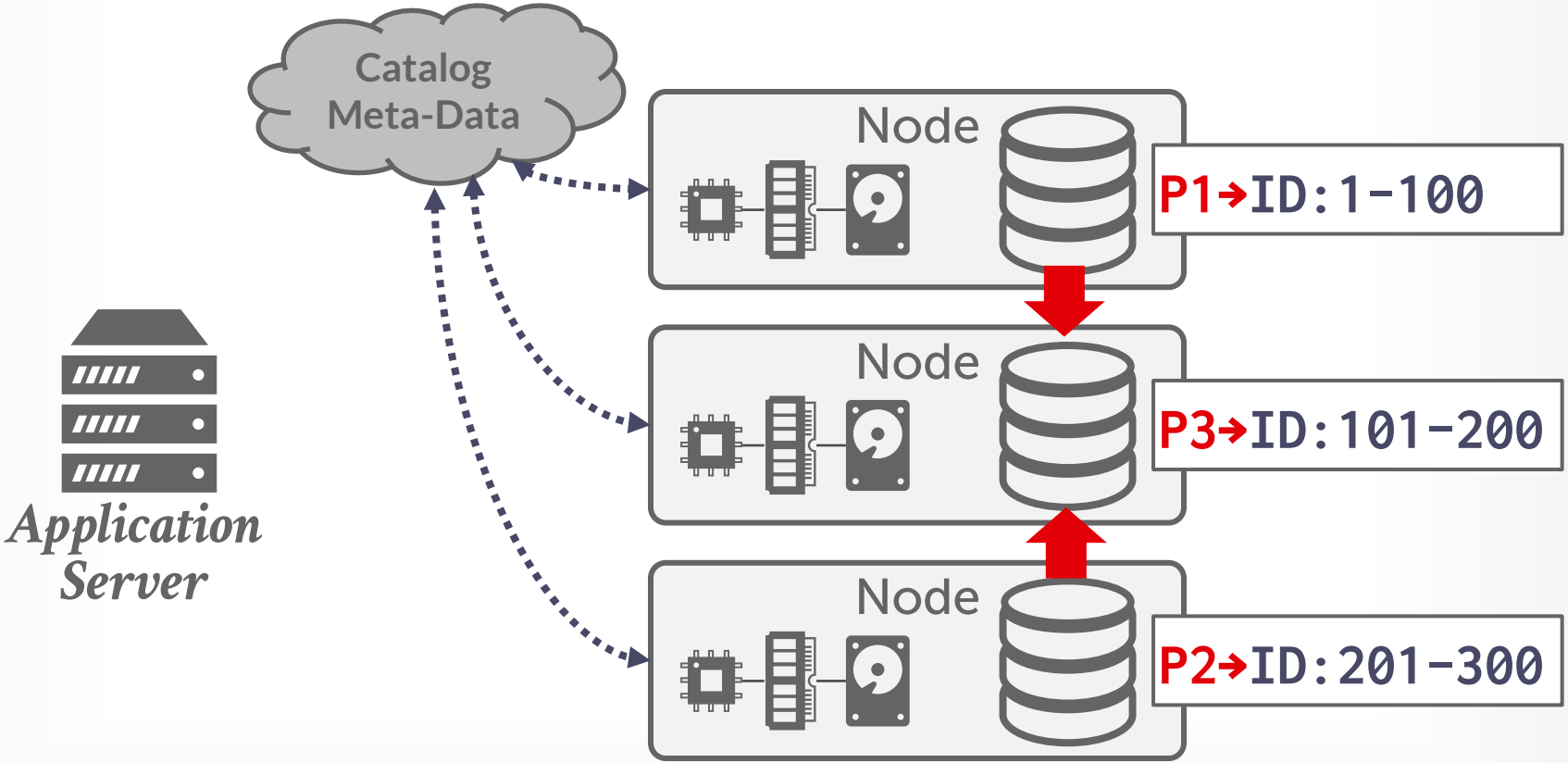
SHARED-NOTHING EXAMPLE



SHARED-NOTHING EXAMPLE



SHARED-NOTHING EXAMPLE



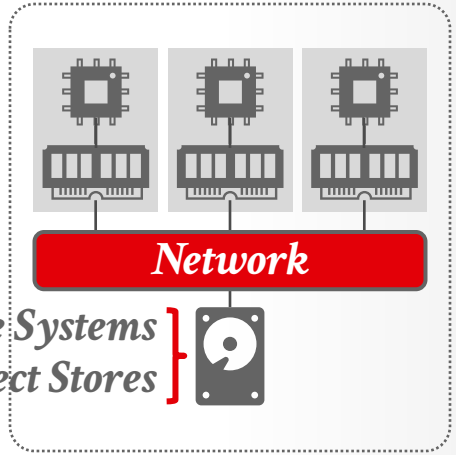
SHARED-DISK

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

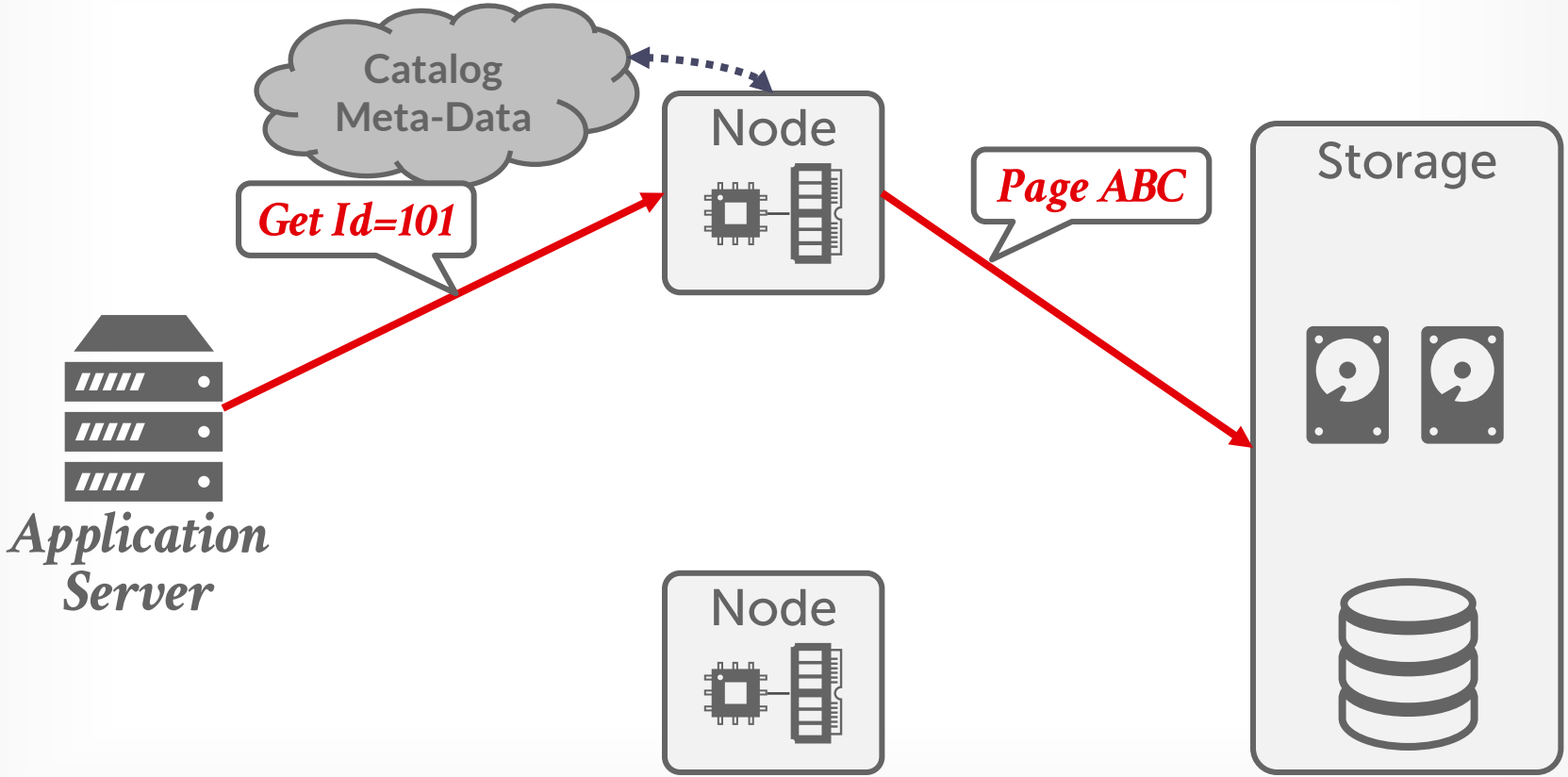
→ Scale execution layer independently from the storage layer.

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

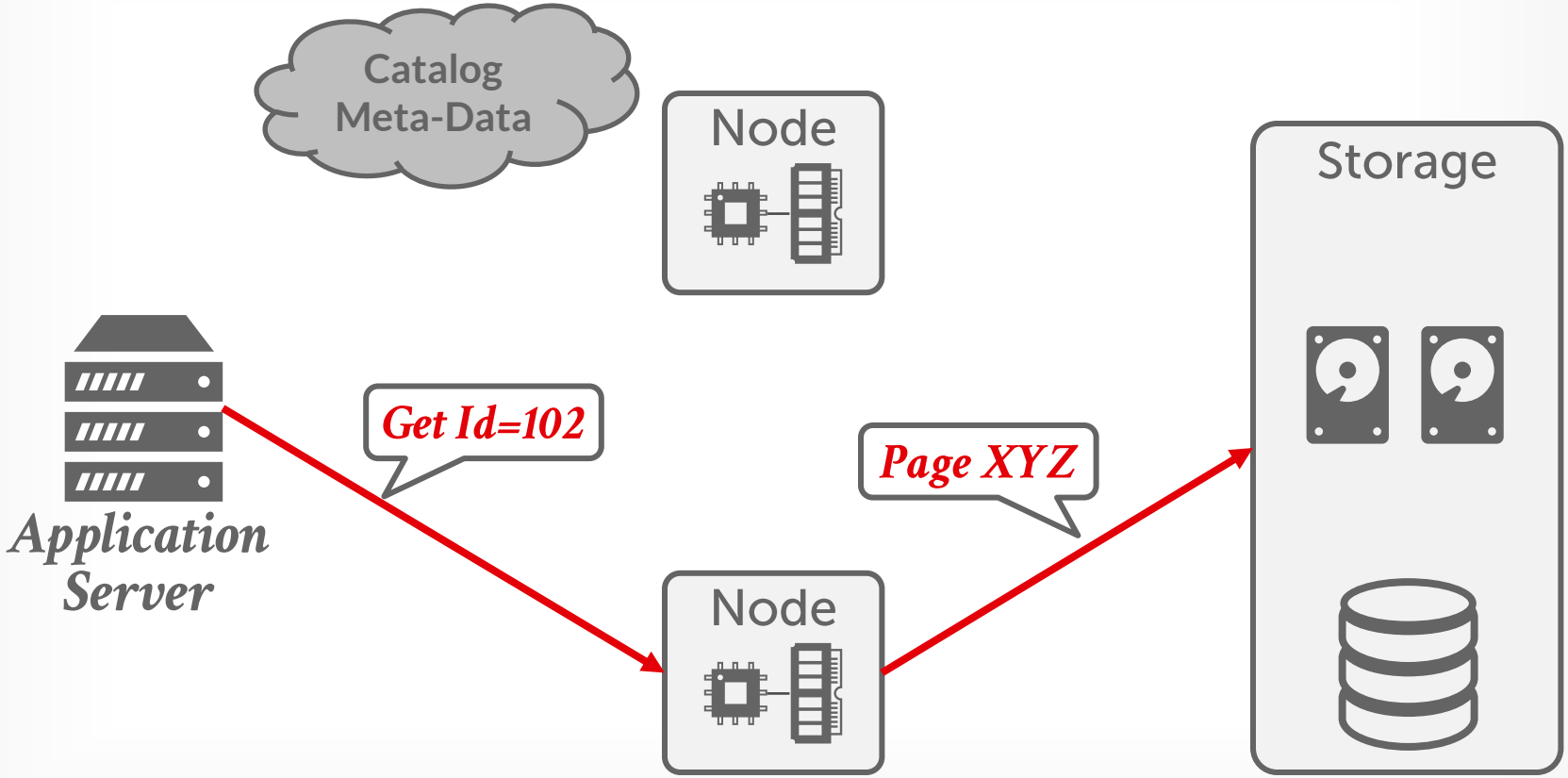
→ This architecture facilitates **data lakes** and **serverless** systems.



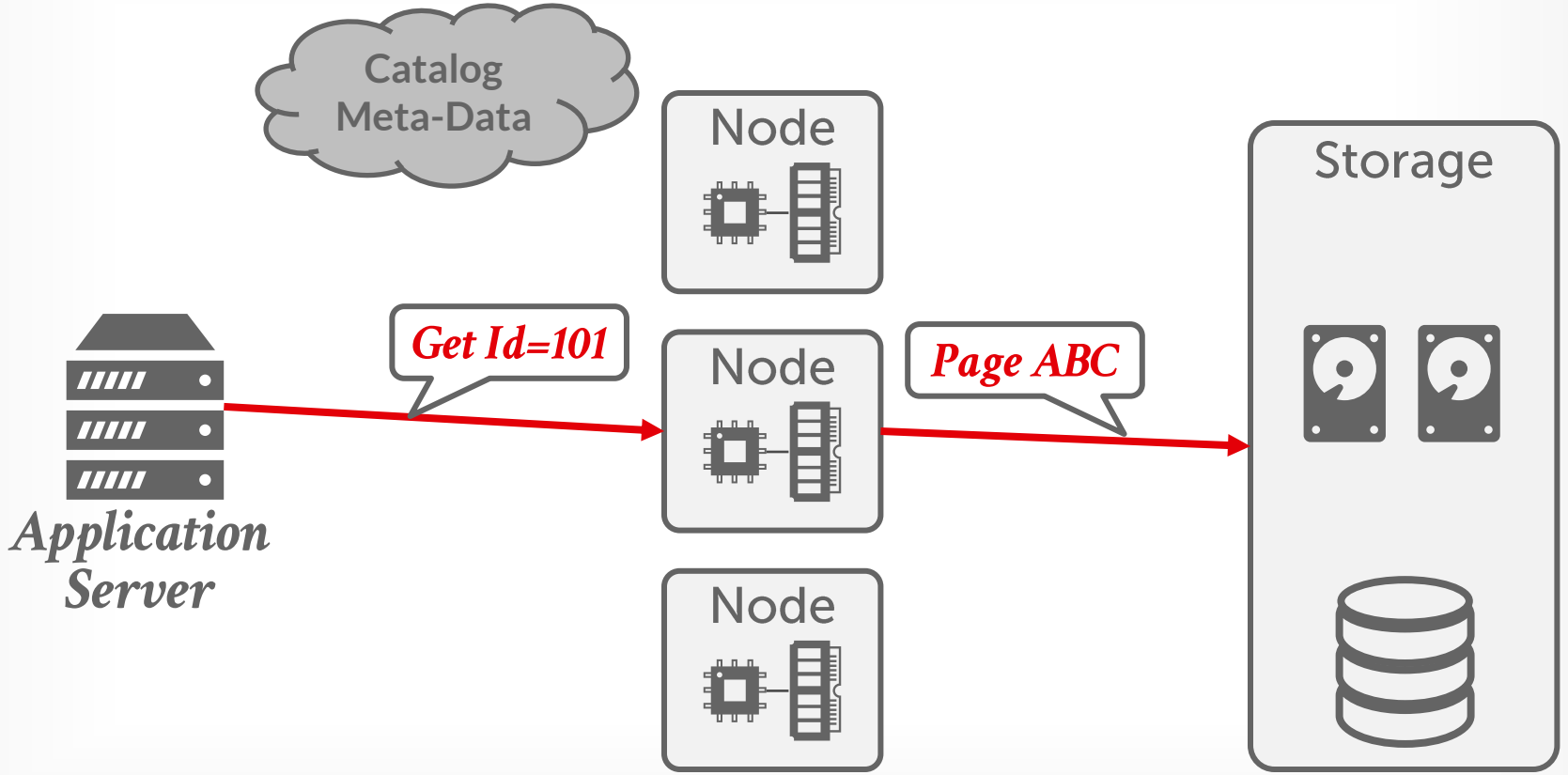
SHARED-DISK EXAMPLE



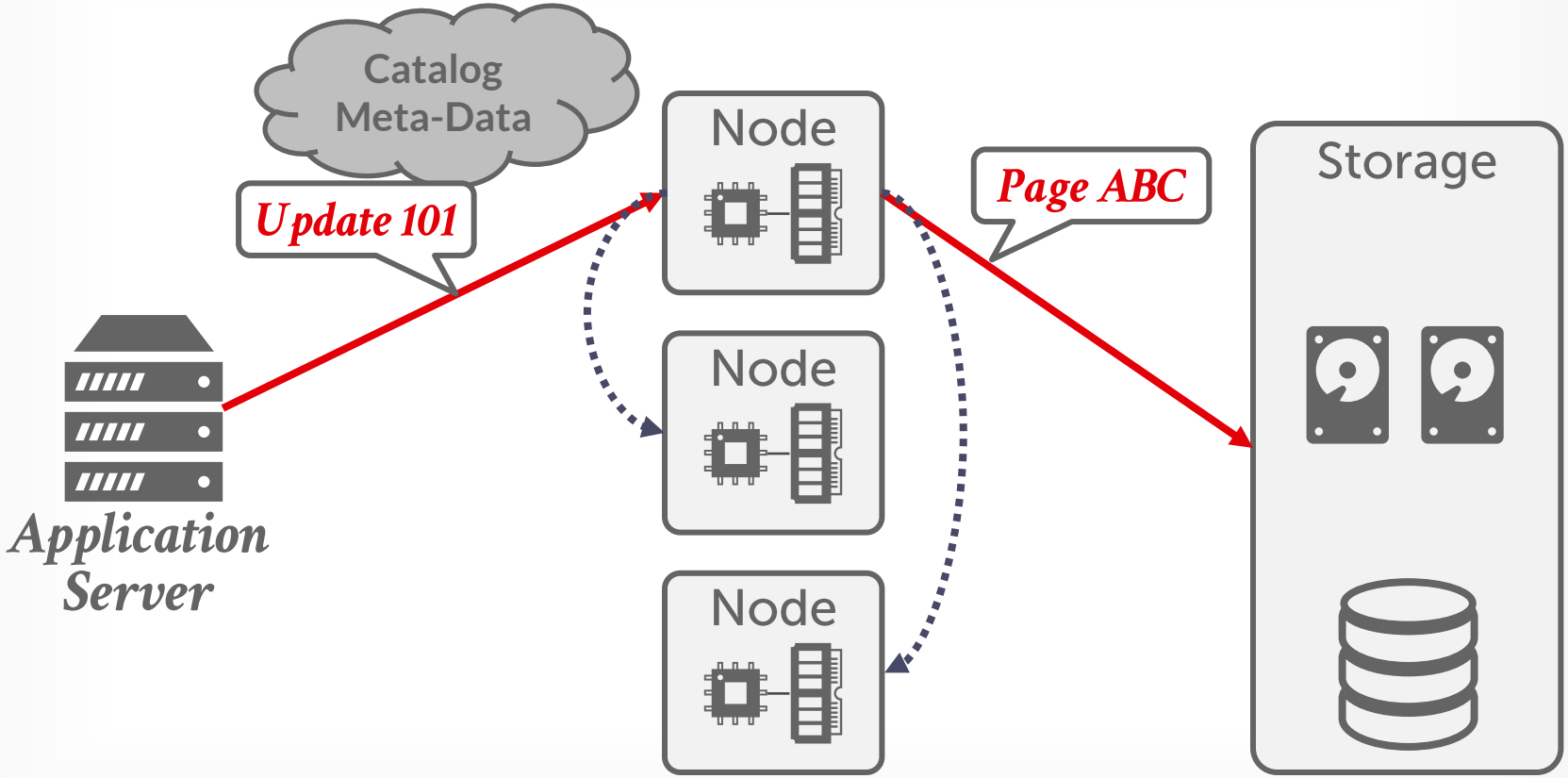
SHARED-DISK EXAMPLE



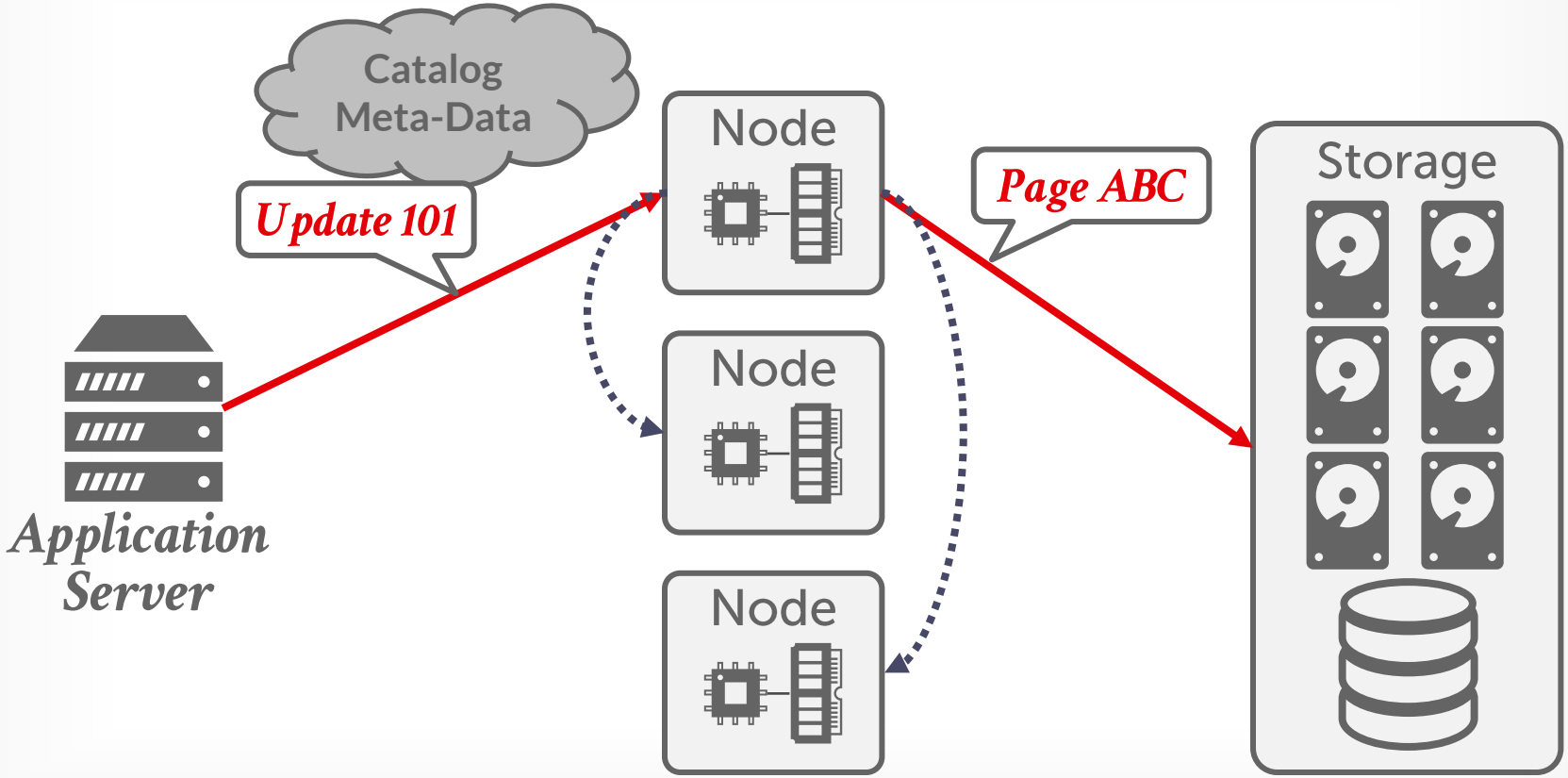
SHARED-DISK EXAMPLE



SHARED-DISK EXAMPLE



SHARED-DISK EXAMPLE



DISTRIBUTED SYSTEM ARCHITECTURE

A distributed DBMS's system architecture specifies the location of the database's data files. This affects how nodes coordinate with each other and where they retrieve/store objects in the database.

Two approaches (not mutually exclusive):

→ **Push Query to Data**

→ **Pull Data to Query**

PUSH VS. PULL

Approach #1: Push Query to Data

→ Send the query (or portion of it) to node that contains the data.
→ Perform as much filtering and processing as possible where data resides before transmitting over network.

Approach #2: Pull Data to Query

→ Bring the data to the node that is executing a query that needs it for processing.
→ This is necessary when there is no compute resources available where database files are located.

Push vs. Pull

Approach

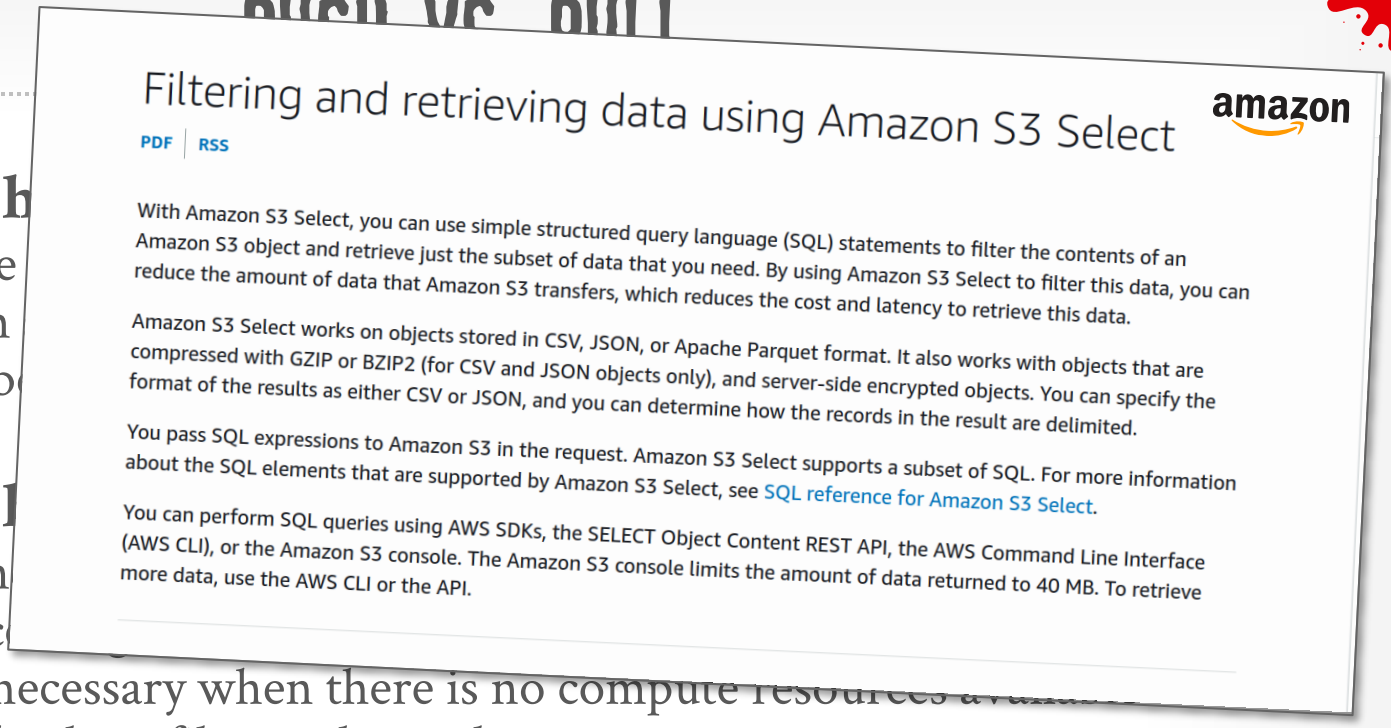
→ Send the

→ Perform
resides b

Approach

→ Bring th
for proc

→ This is necessary when there is no compute resources a
where database files are located.



Filtering and retrieving data using Amazon S3 Select

amazon

PDF | RSS

With Amazon S3 Select, you can use simple structured query language (SQL) statements to filter the contents of an Amazon S3 object and retrieve just the subset of data that you need. By using Amazon S3 Select to filter this data, you can reduce the amount of data that Amazon S3 transfers, which reduces the cost and latency to retrieve this data.

Amazon S3 Select works on objects stored in CSV, JSON, or Apache Parquet format. It also works with objects that are compressed with GZIP or BZIP2 (for CSV and JSON objects only), and server-side encrypted objects. You can specify the format of the results as either CSV or JSON, and you can determine how the records in the result are delimited.

You pass SQL expressions to Amazon S3 in the request. Amazon S3 Select supports a subset of SQL. For more information about the SQL elements that are supported by Amazon S3 Select, see SQL reference for Amazon S3 Select.

You can perform SQL queries using AWS SDKs, the SELECT Object Content REST API, the AWS Command Line Interface (AWS CLI), or the Amazon S3 console. The Amazon S3 console limits the amount of data returned to 40 MB. To retrieve more data, use the AWS CLI or the API.

Approach

Filtering and retrieving data using Amazon S3 Select

amazon

PDF | RSS

With Amazon S3 Select, you

uery language (SQL) statements to filter the contents of an
at you need. By using Amazon S3 Select to filter this data, you can
ich reduces the cost and latency to retrieve this data.

r Apache Parquet format. It also works with objects that are
only), and server-side encrypted objects. You can specify the
etermine how the records in the result are delimited.

azon S3 Select supports a subset of SQL. For more information
Select, see SQL reference for Amazon S3 Select.

bject Content REST API, the AWS Command Line Interface
le limits the amount of data returned to 40 MB. To retrieve

ute resources a

Query Blob Contents

Microsoft

Feedback

Article • 07/20/2021 • 10 minutes to read • 3 contributors

The `Query Blob Contents` API applies a simple Structured Query Language (SQL) statement on a blob's contents and returns only the queried subset of the data. You can also call `Query Blob Contents` to query the contents of a version or snapshot.

Request

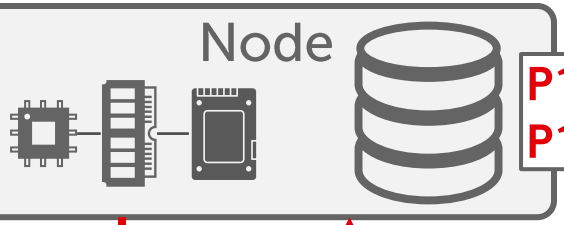
The `Query Blob Contents` request may be constructed as follows. HTTPS is recommended. Replace *myaccount* with the name of your storage account:

POST Method Request URI	HTTP Version
`https://myaccount.blob.core.windows.net/mycontainer/myblob?comp=query`	HTTP/1.0
`https://myaccount.blob.core.windows.net/mycontainer/myblob?comp=query&snapshot=<DateTime>`	HTTP/1.1
`https://myaccount.blob.core.windows.net/mycontainer/myblob?comp=query&versionid=<DateTime>`	

PUSH QUERY TO DATA

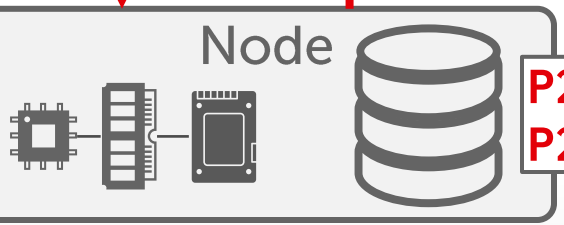
```
SELECT * FROM R JOIN S
  ON R.id = S.id
```

Application Server

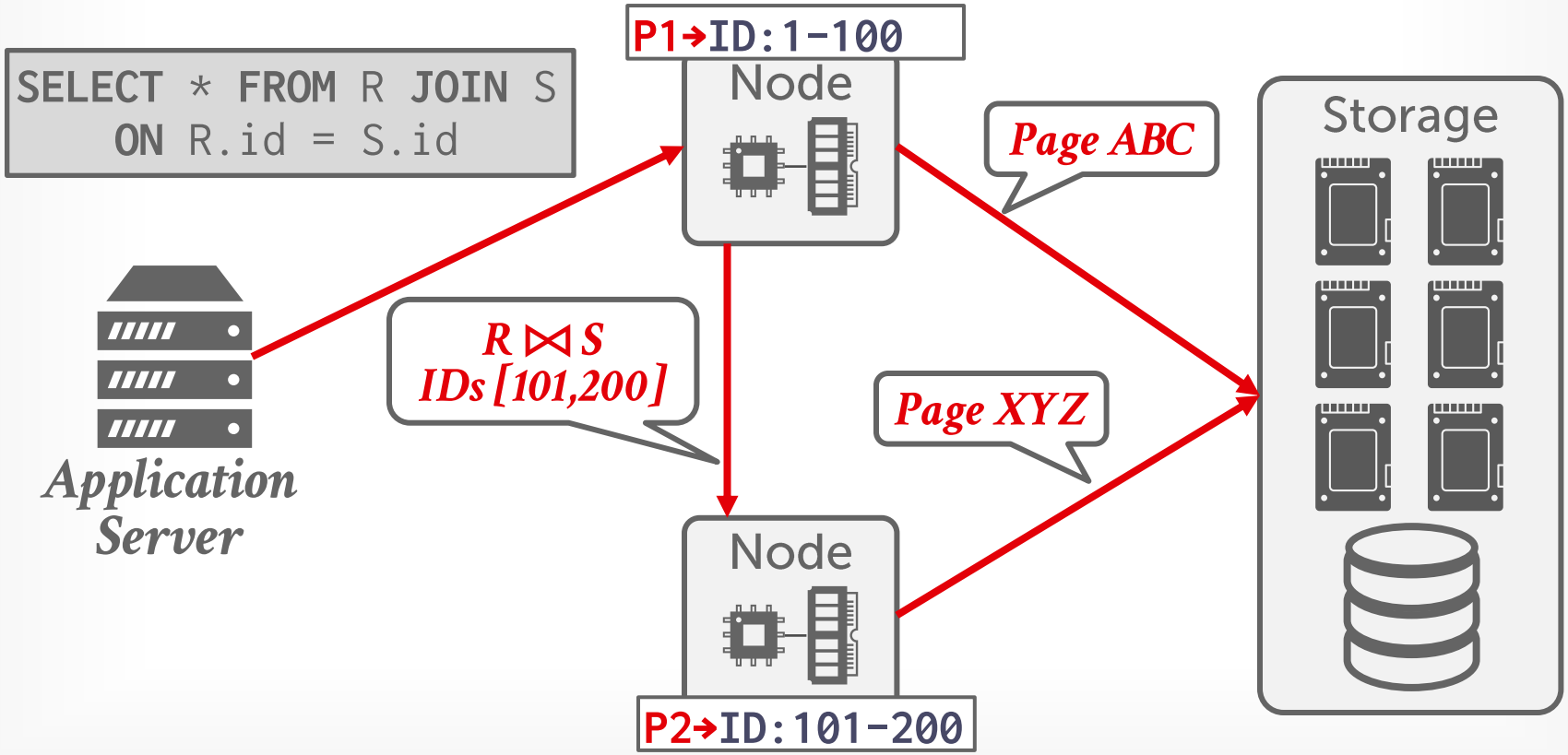


R ⋈ S
IDs [101,200]

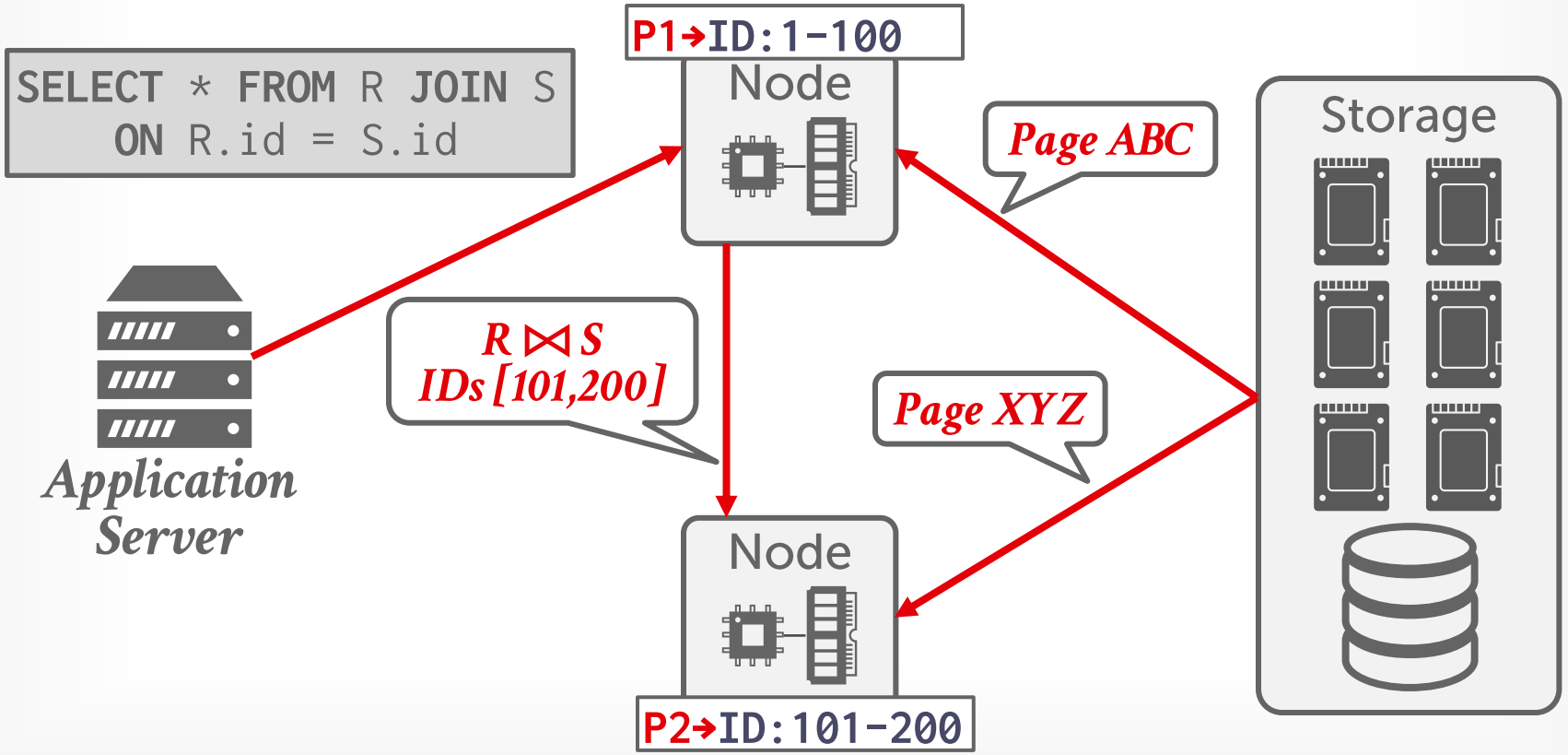
Result: R ⋈ S



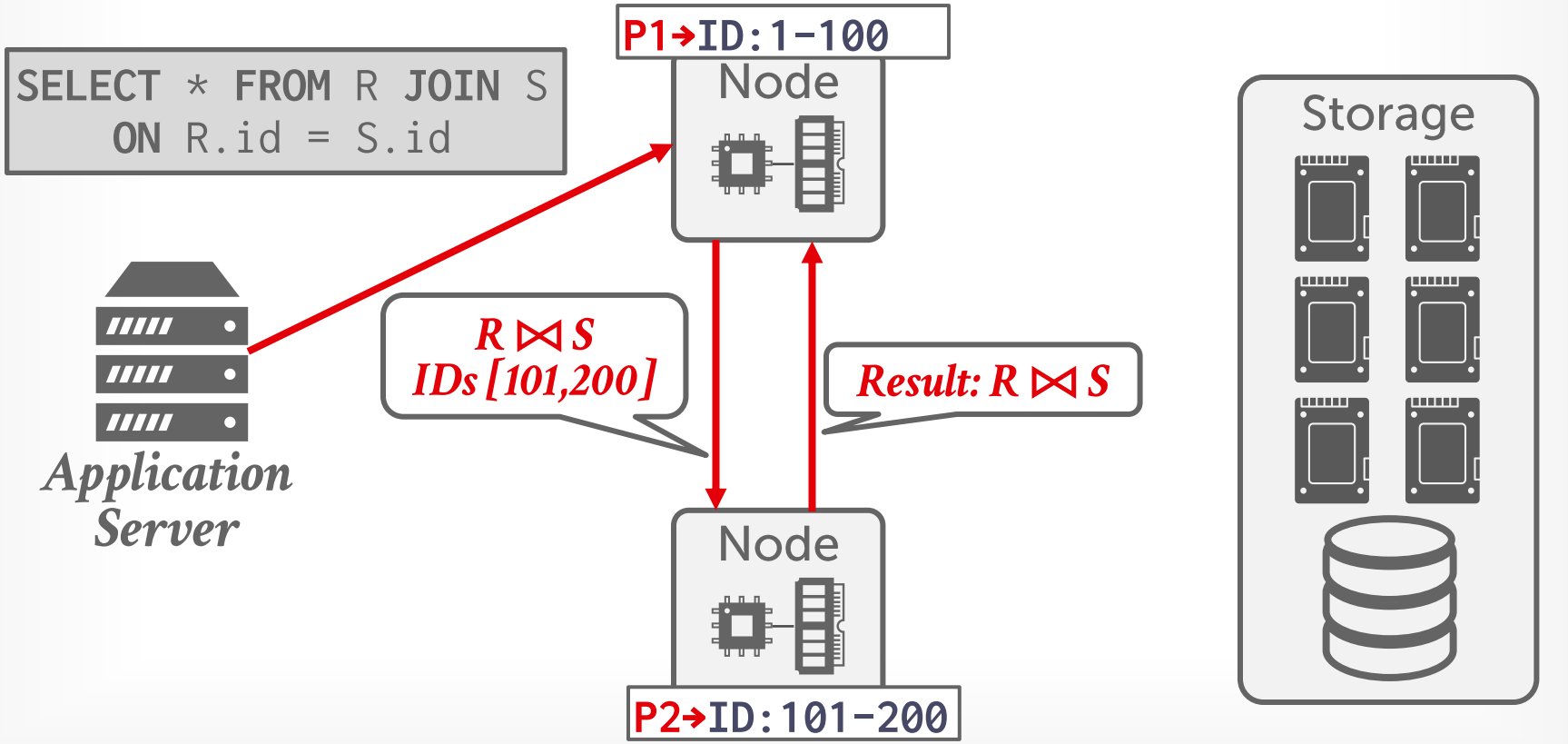
PULL DATA TO QUERY



PULL DATA TO QUERY



PULL DATA TO QUERY



DATABASE PARTITIONING

Split database across multiple resources:

→ Disks, nodes, processors.
→ Called "sharding" in NoSQL systems.
→ Fine-Grained vs. Coarse-grained

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

Table1

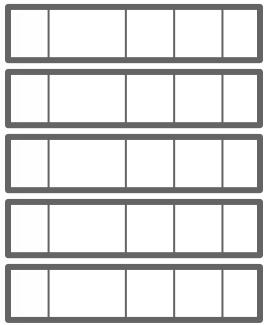
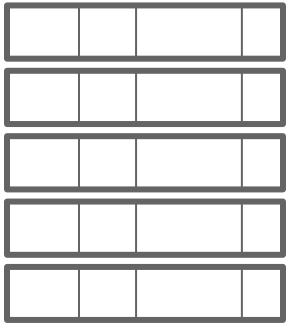
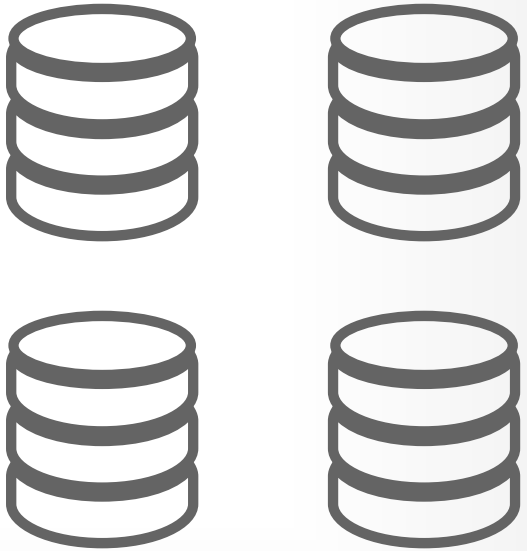


Table2



Partitions



Ideal Query:

```
SELECT * FROM table1
```

NAÏVE TABLE PARTITIONING

Table1

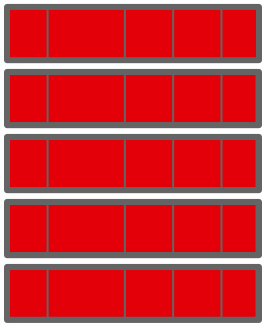
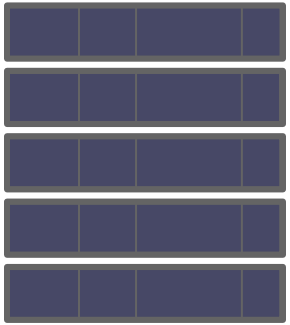
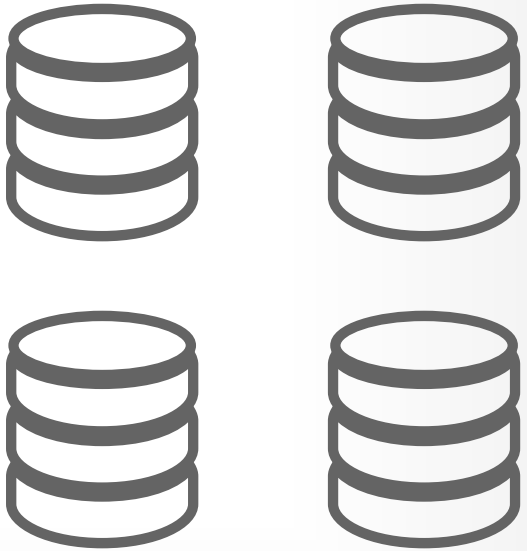


Table2



Partitions



Ideal Query:

```
SELECT * FROM table1
```

NAÏVE TABLE PARTITIONING

Table1

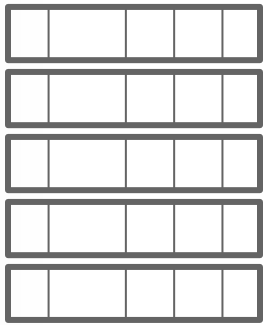
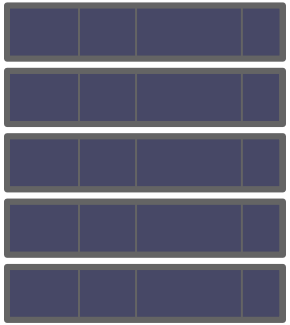
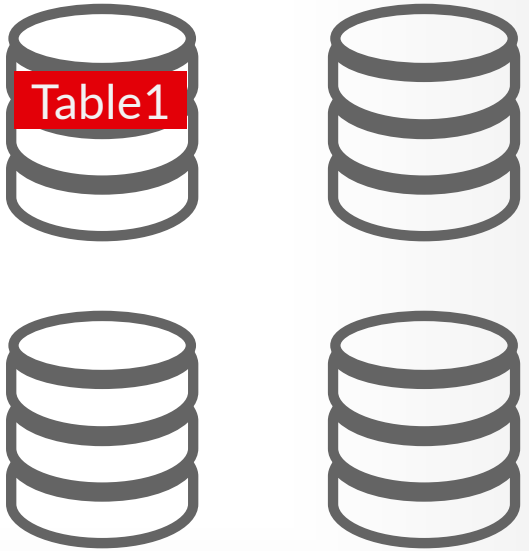


Table2



Partitions



Ideal Query:

```
SELECT * FROM table1
```

NAÏVE TABLE PARTITIONING

Table1

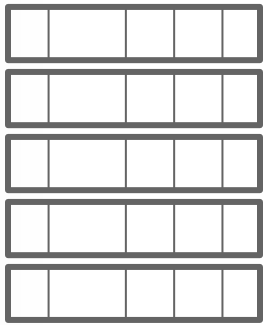
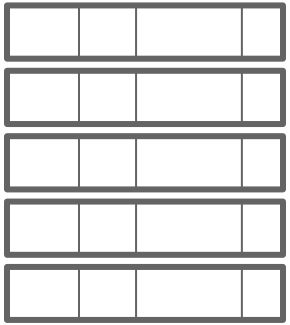
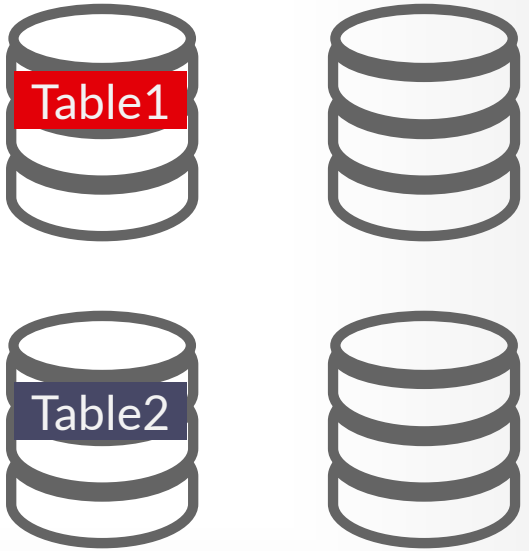


Table2



Partitions



Ideal Query:

```
SELECT * FROM table1
```

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
→ Round Robin

HORIZONTAL PARTITIONING

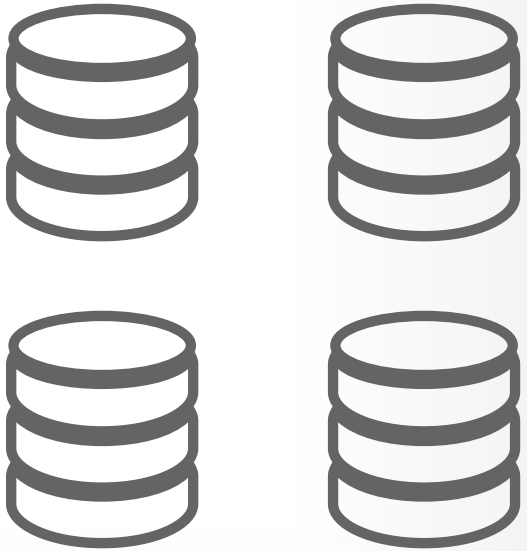
Table

101	a	XXX	2025-11-29
102	b	XXY	2025-11-28
103	c	XYZ	2025-11-29
104	d	XYX	2025-11-27
105	e	XYY	2025-11-29

Ideal Query:

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

Partitions



HORIZONTAL PARTITIONING

Partitioning Key

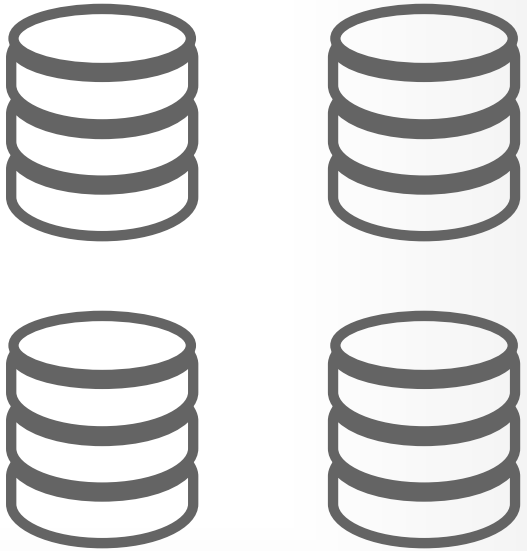
Table

101	a	XXX	2025-11-29	*hash(a)%4 = P2*
102	b	XXY	2025-11-28	*hash(b)%4 = P4*
103	c	XYZ	2025-11-29	*hash(c)%4 = P3*
104	d	XYX	2025-11-27	*hash(d)%4 = P2*
105	e	XYY	2025-11-29	*hash(e)%4 = P1*

Ideal Query:

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

Partitions



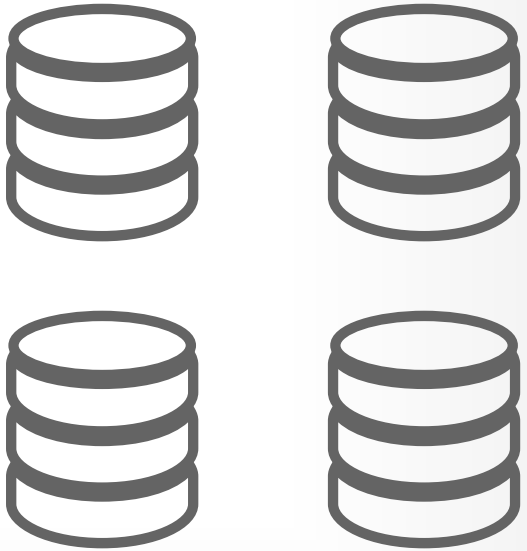
HORIZONTAL PARTITIONING

Partitioning Key

Table

101	a	XXX	2025-11-29	*hash(a)%4 = P2*
102	b	XXY	2025-11-28	*hash(b)%4 = P4*
103	c	XYZ	2025-11-29	*hash(c)%4 = P3*
104	d	XYX	2025-11-27	*hash(d)%4 = P2*
105	e	XYY	2025-11-29	*hash(e)%4 = P1*

Partitions



Ideal Query:

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

HORIZONTAL PARTITIONING

Partitioning Key

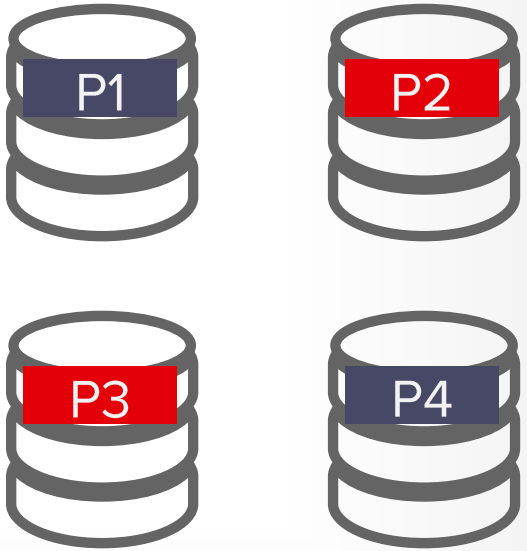
Table

101	a	XXX	2025-11-29	*hash(a)%4 = P2*
102	b	XXY	2025-11-28	*hash(b)%4 = P4*
103	c	XYZ	2025-11-29	*hash(c)%4 = P3*
104	d	XYX	2025-11-27	*hash(d)%4 = P2*
105	e	XYY	2025-11-29	*hash(e)%4 = P1*

Ideal Query:

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

Partitions



HORIZONTAL PARTITIONING

Partitioning Key

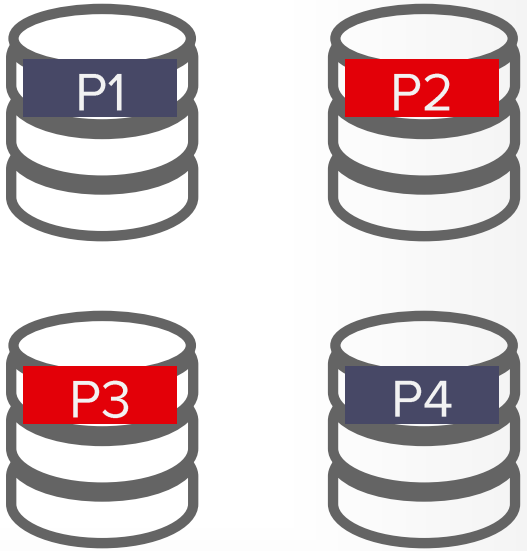
Table

101	a	XXX	2025-11-29	*hash(a)%4 = P2*
102	b	XXY	2025-11-28	*hash(b)%4 = P4*
103	c	XYZ	2025-11-29	*hash(c)%4 = P3*
104	d	XYX	2025-11-27	*hash(d)%4 = P2*
105	e	XYY	2025-11-29	*hash(e)%4 = P1*

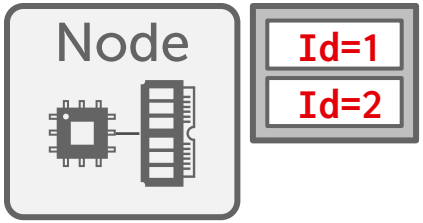
Ideal Query:

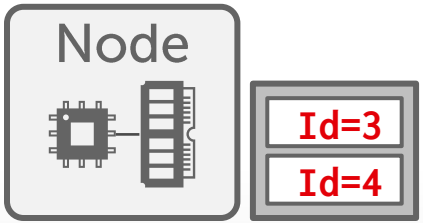
```
SELECT * FROM table
 WHERE partitionKey = ?
```

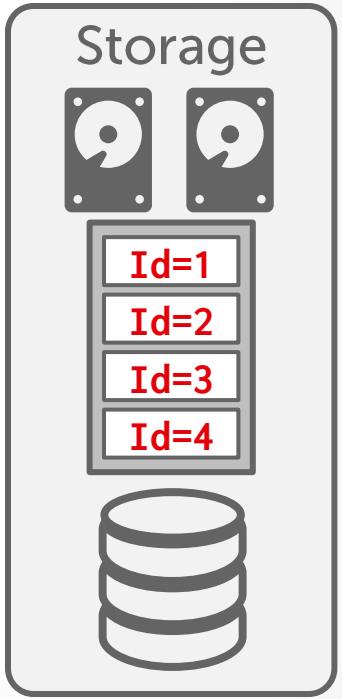
Partitions



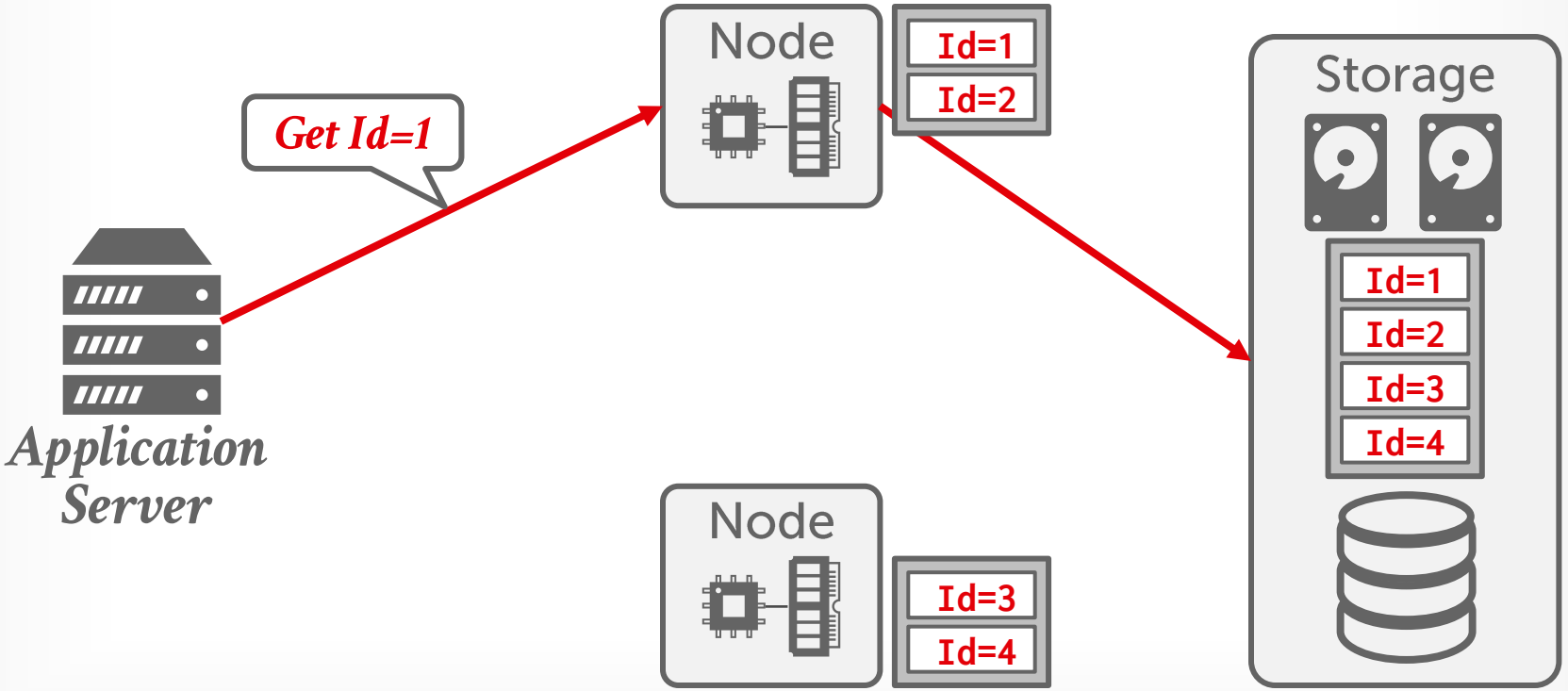
SHARED-DISK PARTITIONING



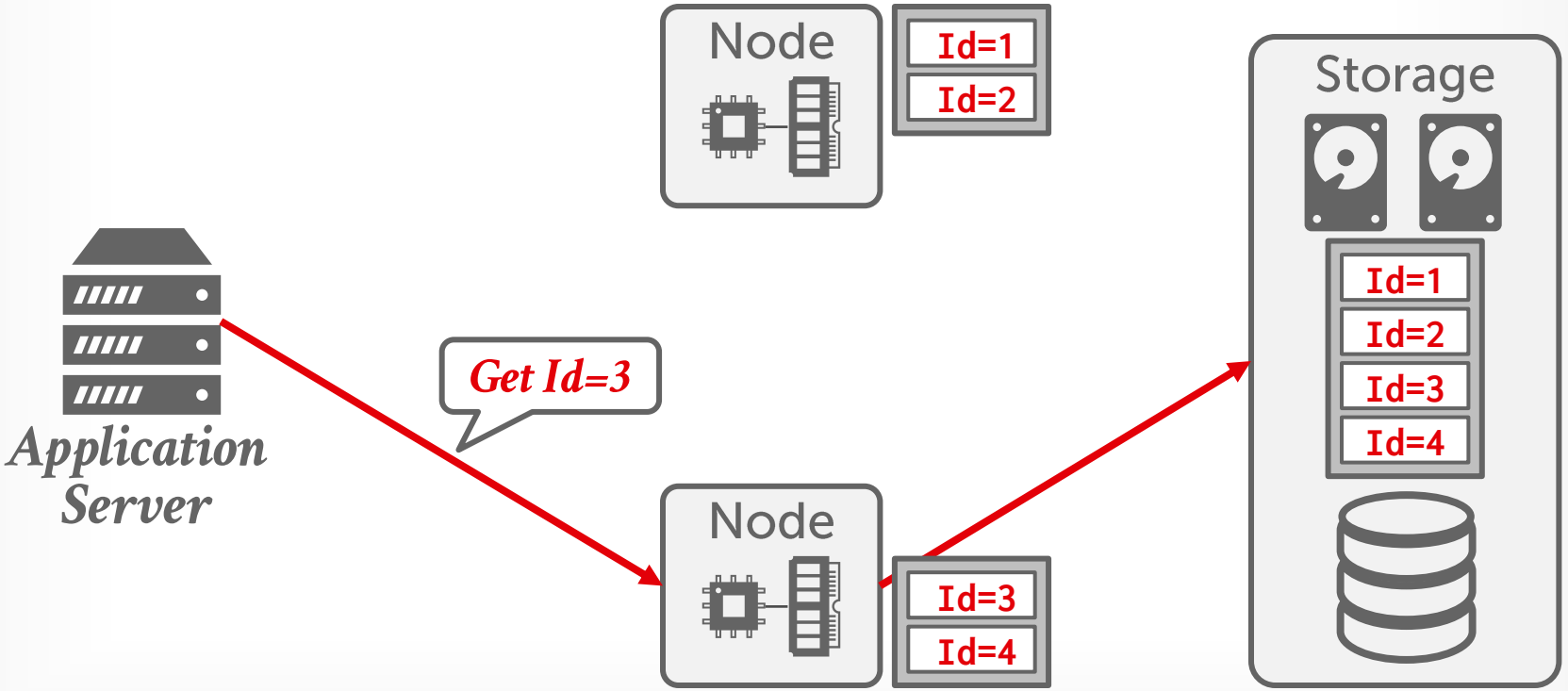




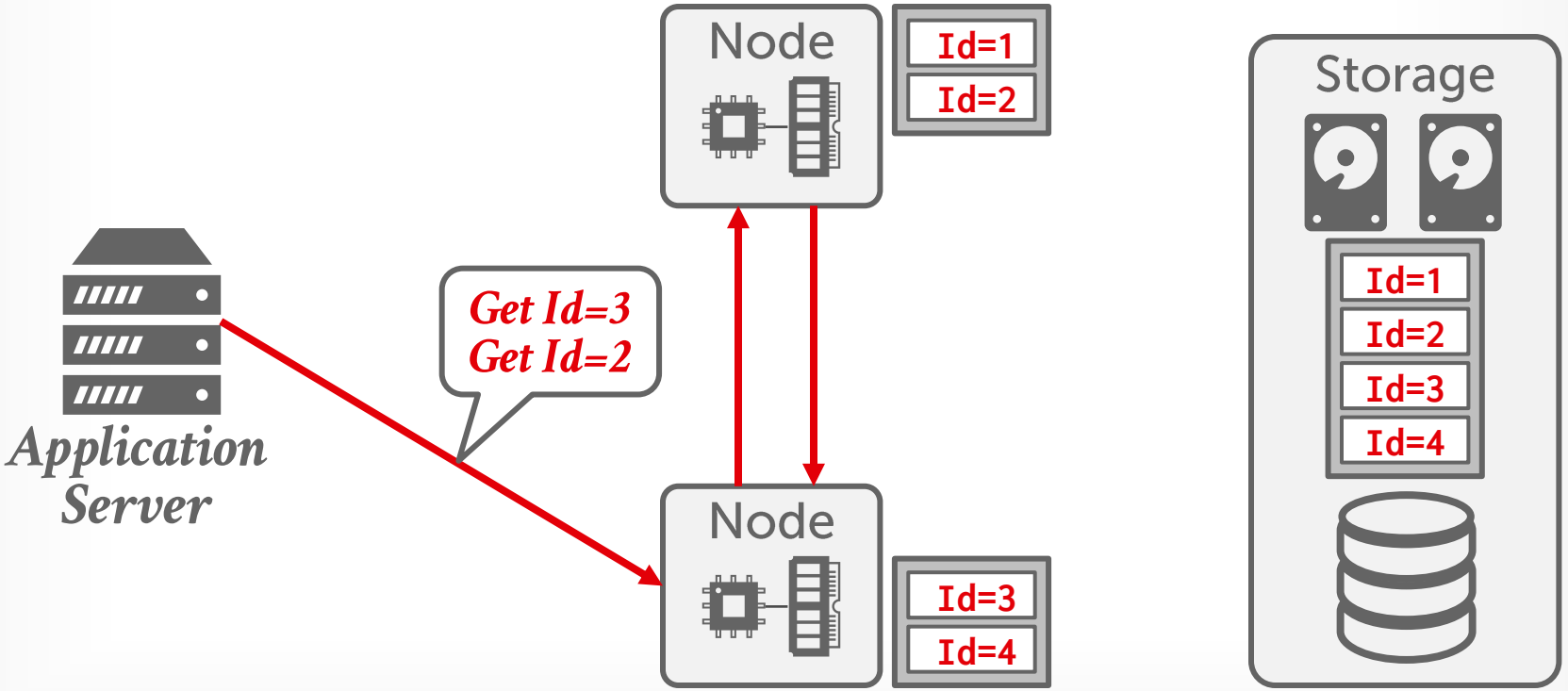
SHARED-DISK PARTITIONING



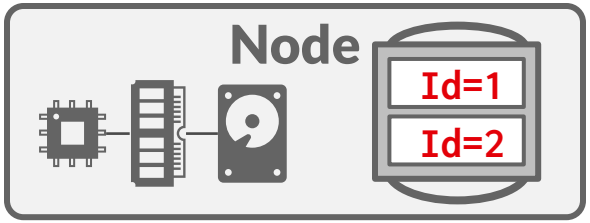
SHARED-DISK PARTITIONING

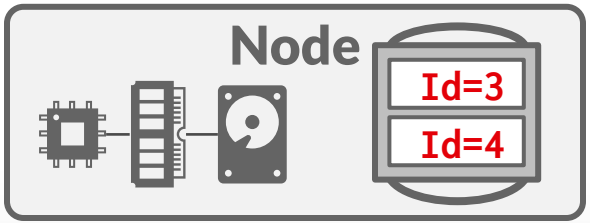


SHARED-DISK PARTITIONING

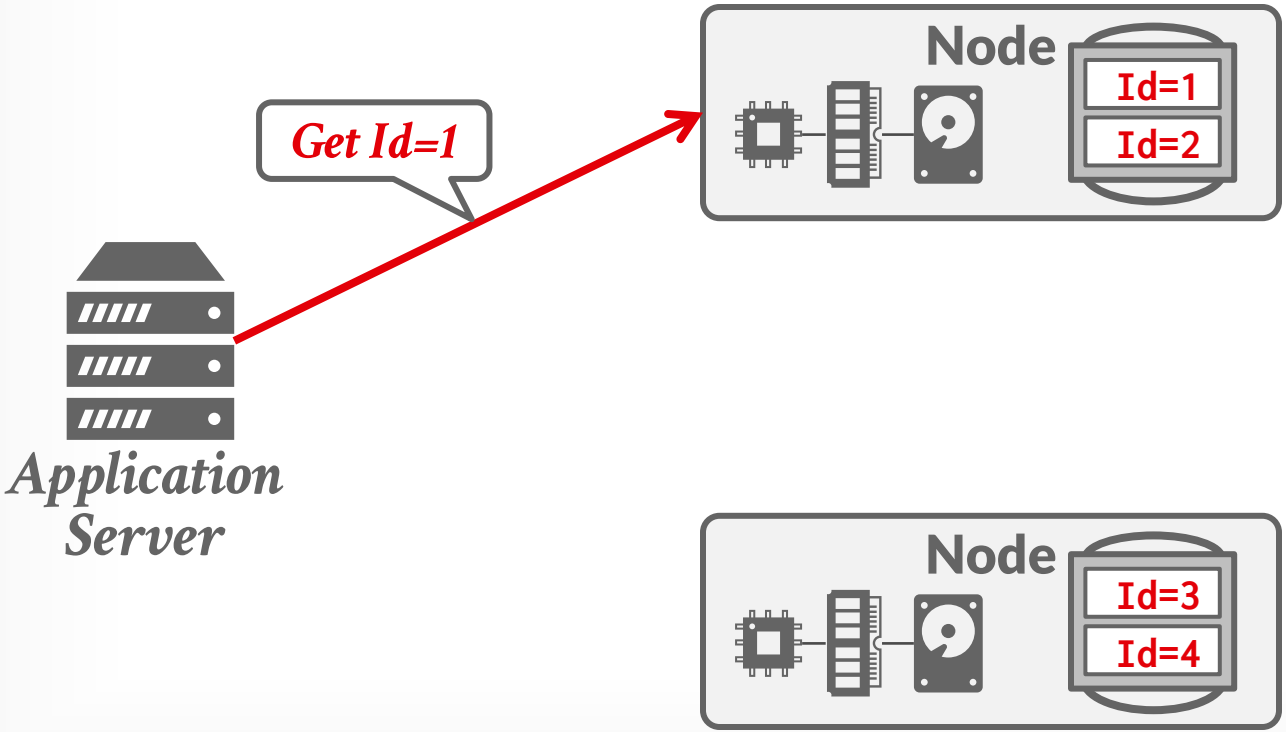


SHARED-NOTHING PARTITIONING

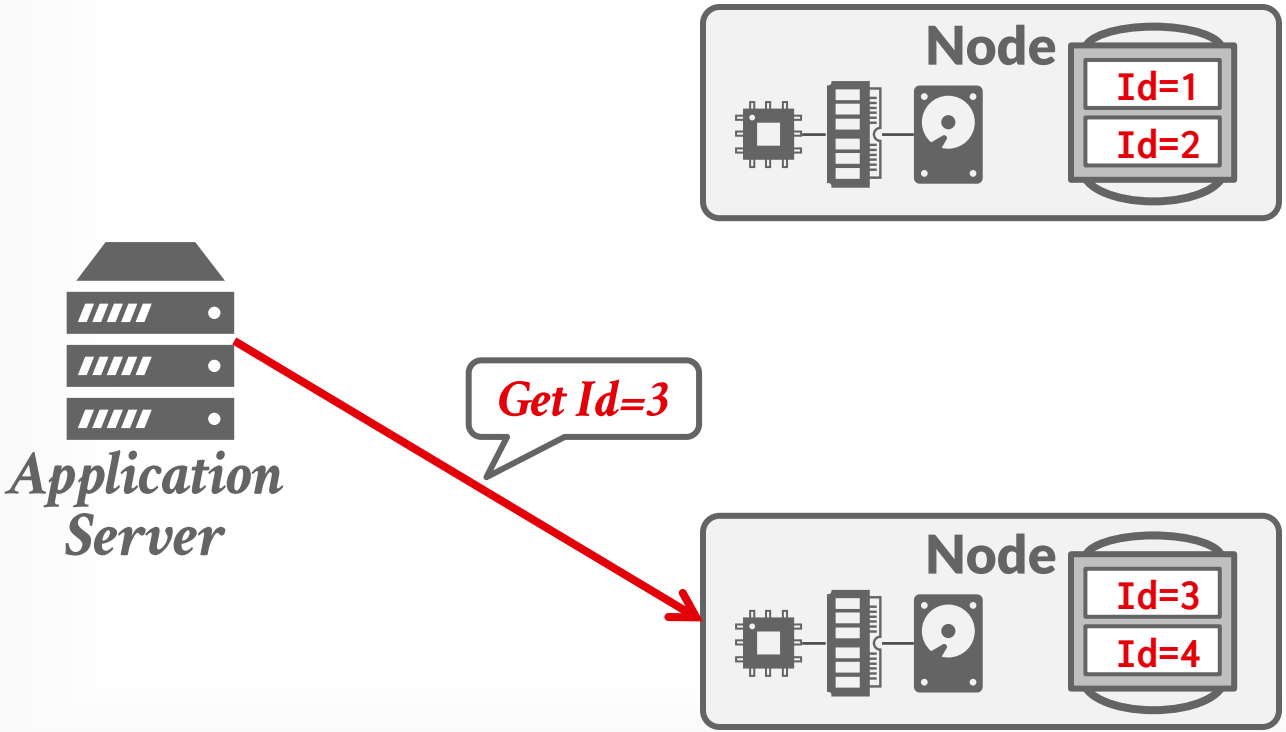




SHARED-NOTHING PARTITIONING



SHARED-NOTHING PARTITIONING



HORIZONTAL PARTITIONING

Partitioning Key

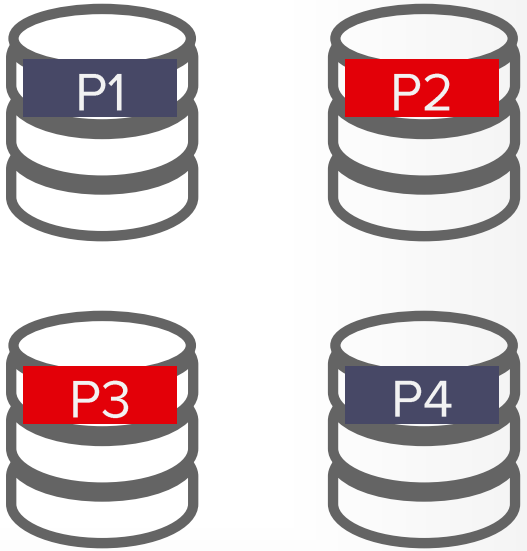
Table

101	a	XXX	2025-11-29	*hash(a)%4 = P2*
102	b	XXY	2025-11-28	*hash(b)%4 = P4*
103	c	XYZ	2025-11-29	*hash(c)%4 = P3*
104	d	XYX	2025-11-27	*hash(d)%4 = P2*
105	e	XYY	2025-11-29	*hash(e)%4 = P1*

Ideal Query:

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

Partitions



HORIZONTAL PARTITIONING

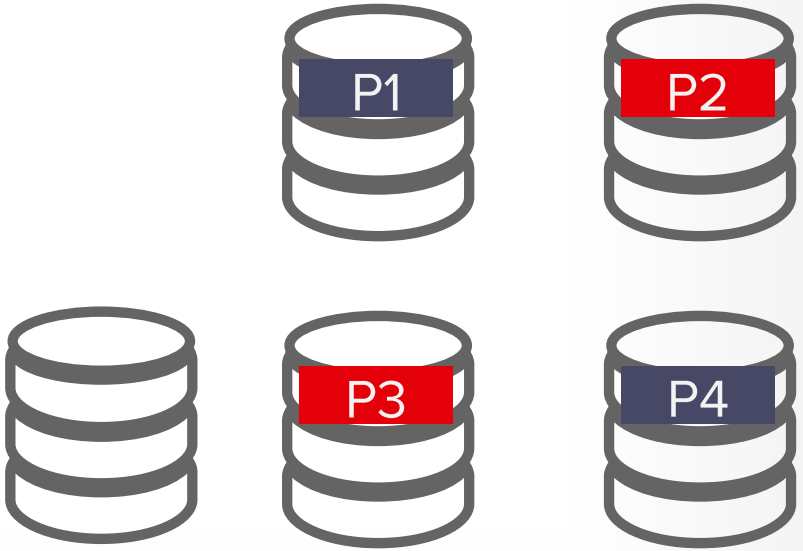
Partitioning Key

Table

101	a	XXX	2025-11-29
102	b	XXY	2025-11-28
103	c	XYZ	2025-11-29
104	d	XYX	2025-11-27
105	e	XYY	2025-11-29

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

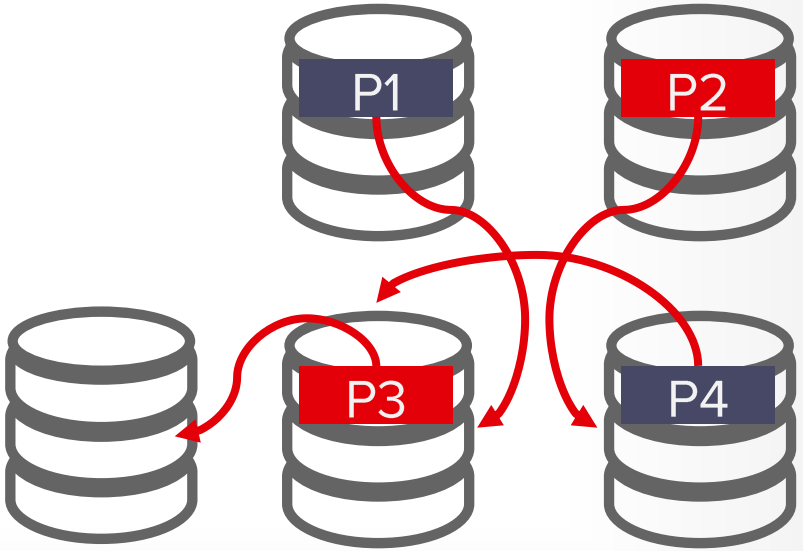
Table

101	a	XXX	2025-11-29	*hash(a)%5 = P4*
102	b	XXY	2025-11-28	*hash(b)%5 = P3*
103	c	XYZ	2025-11-29	*hash(c)%5 = P5*
104	d	XYX	2025-11-27	*hash(d)%5 = P1*
105	e	XYY	2025-11-29	*hash(e)%5 = P3*

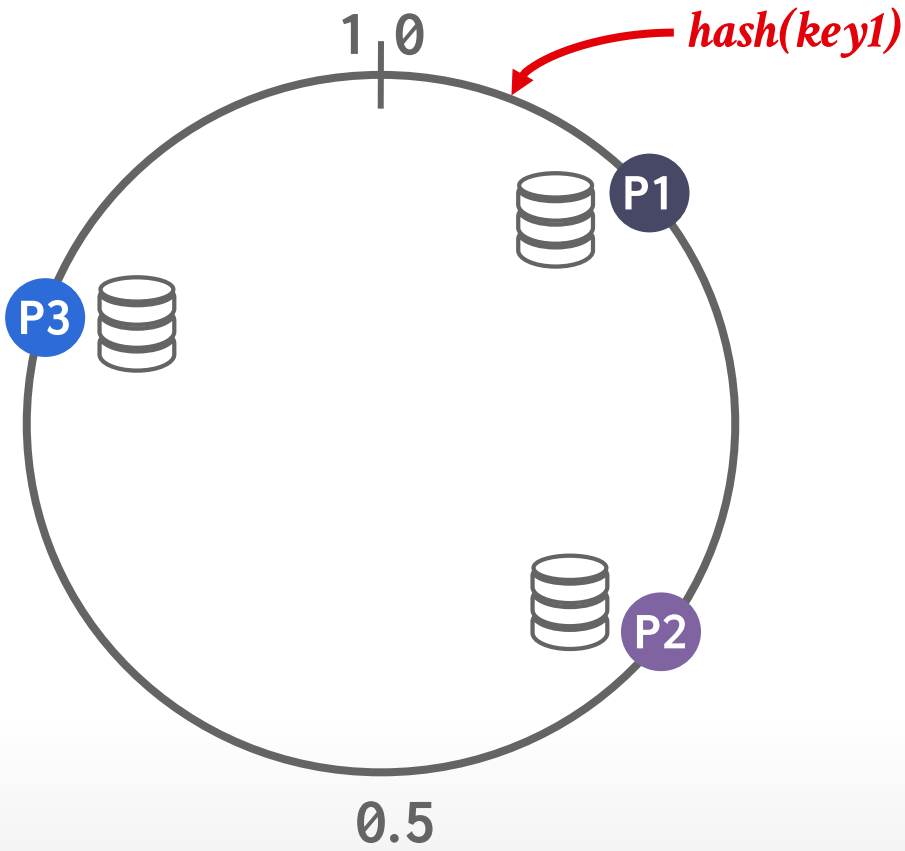
Ideal Query:

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

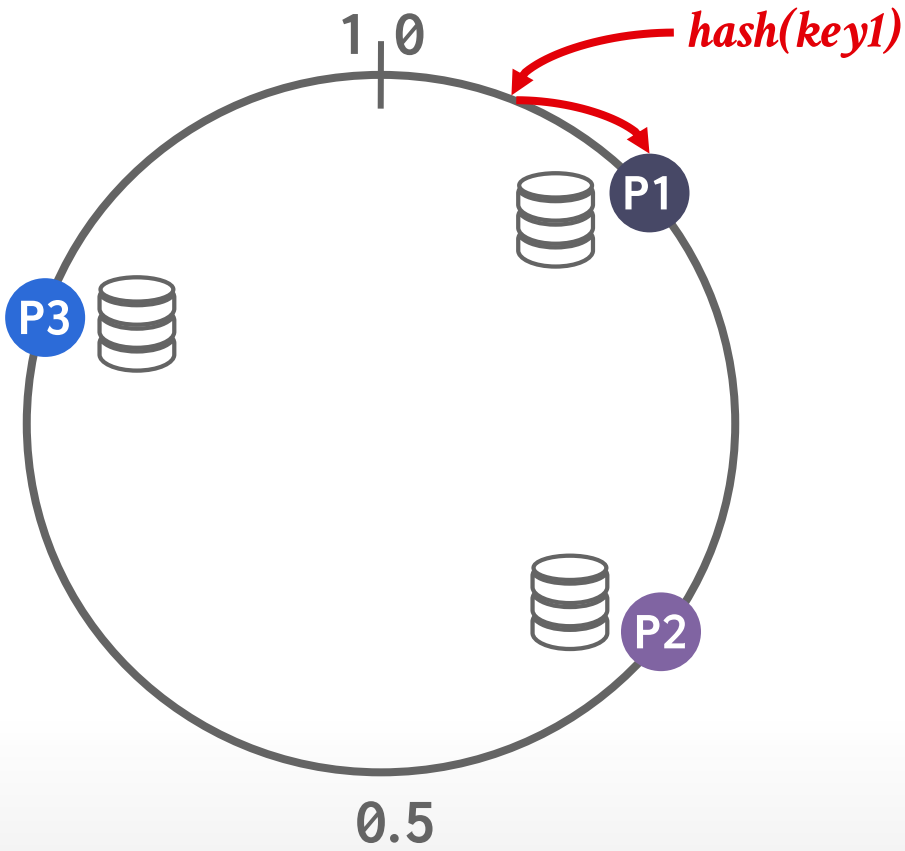
Partitions



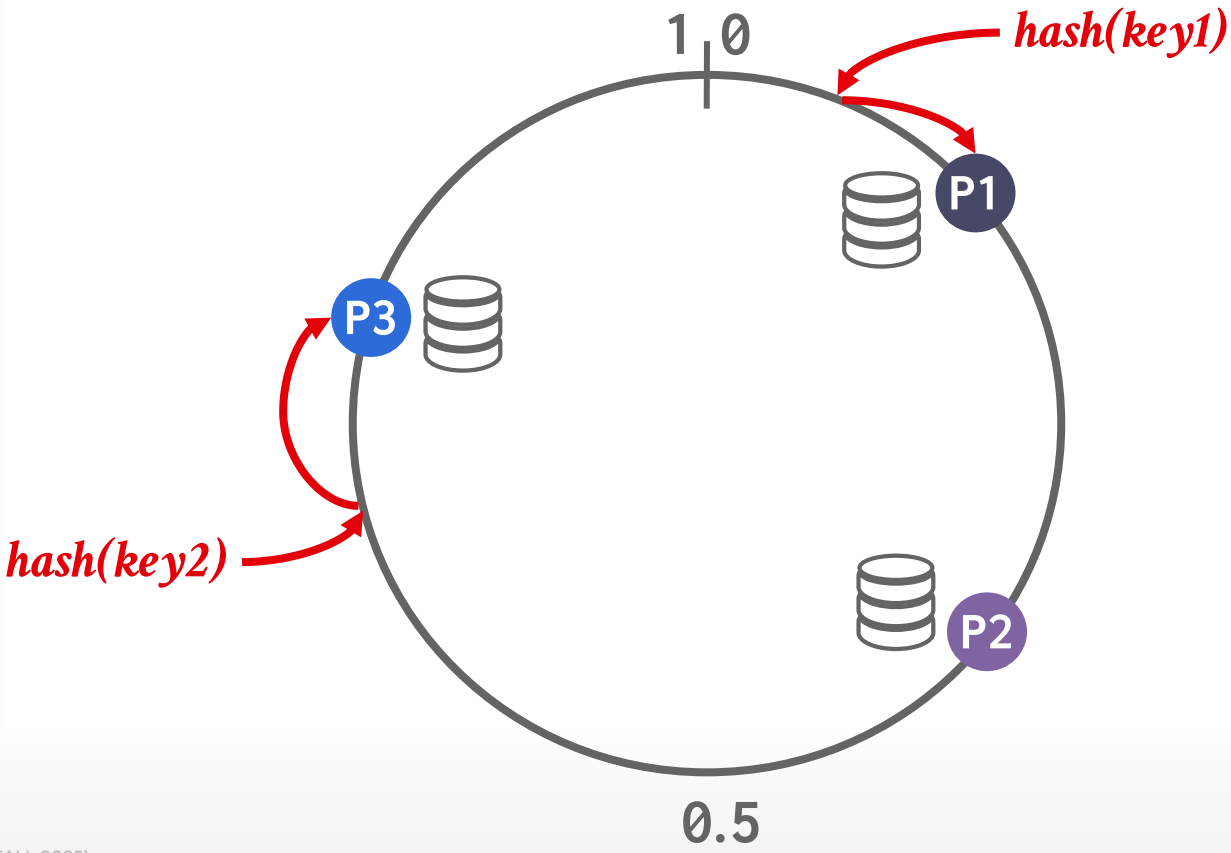
CONSISTENT HASHING



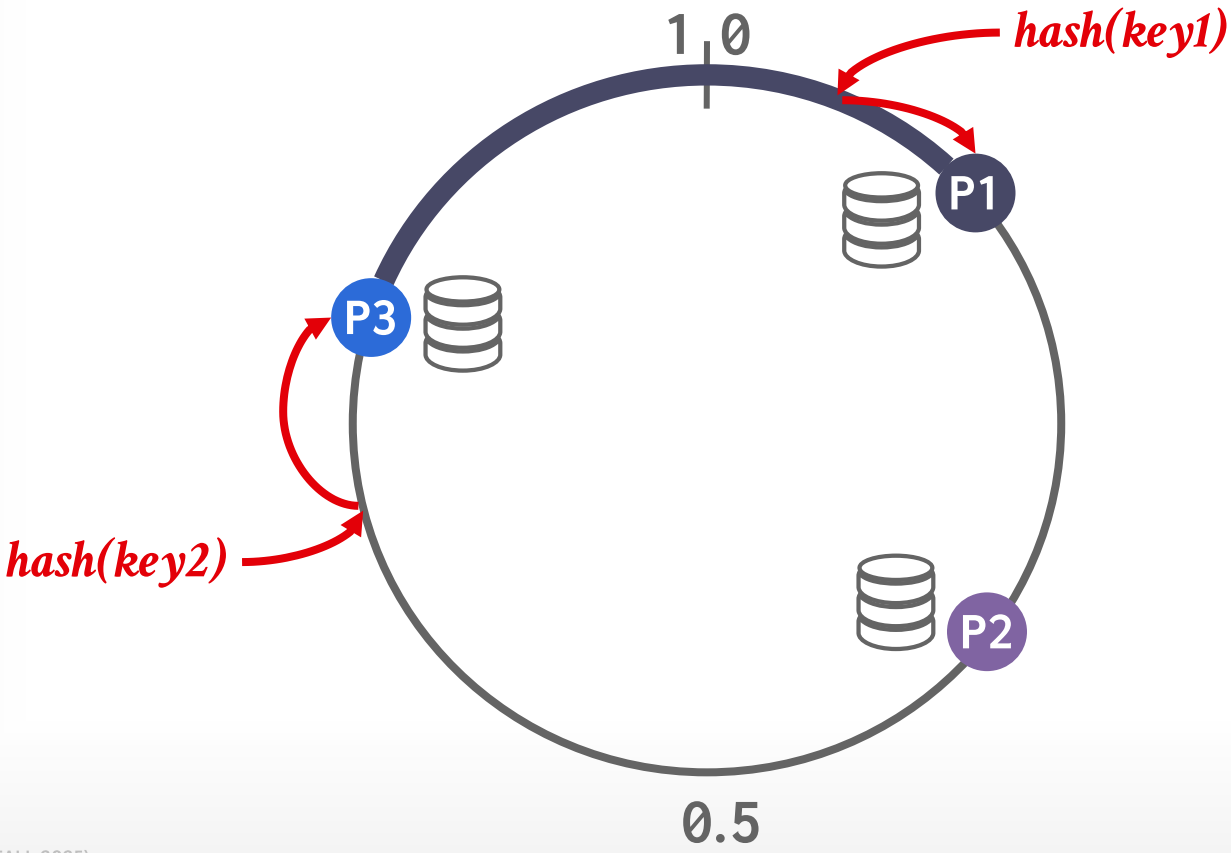
CONSISTENT HASHING



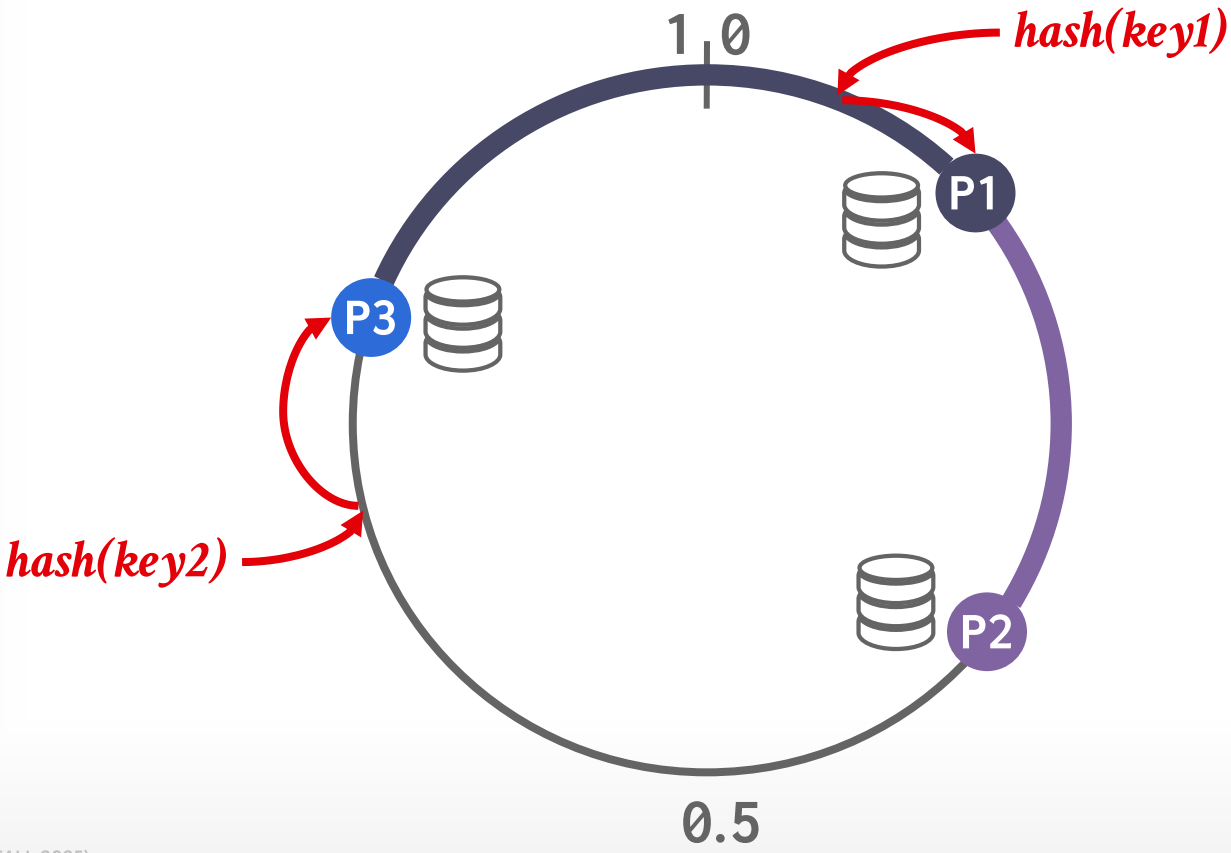
CONSISTENT HASHING



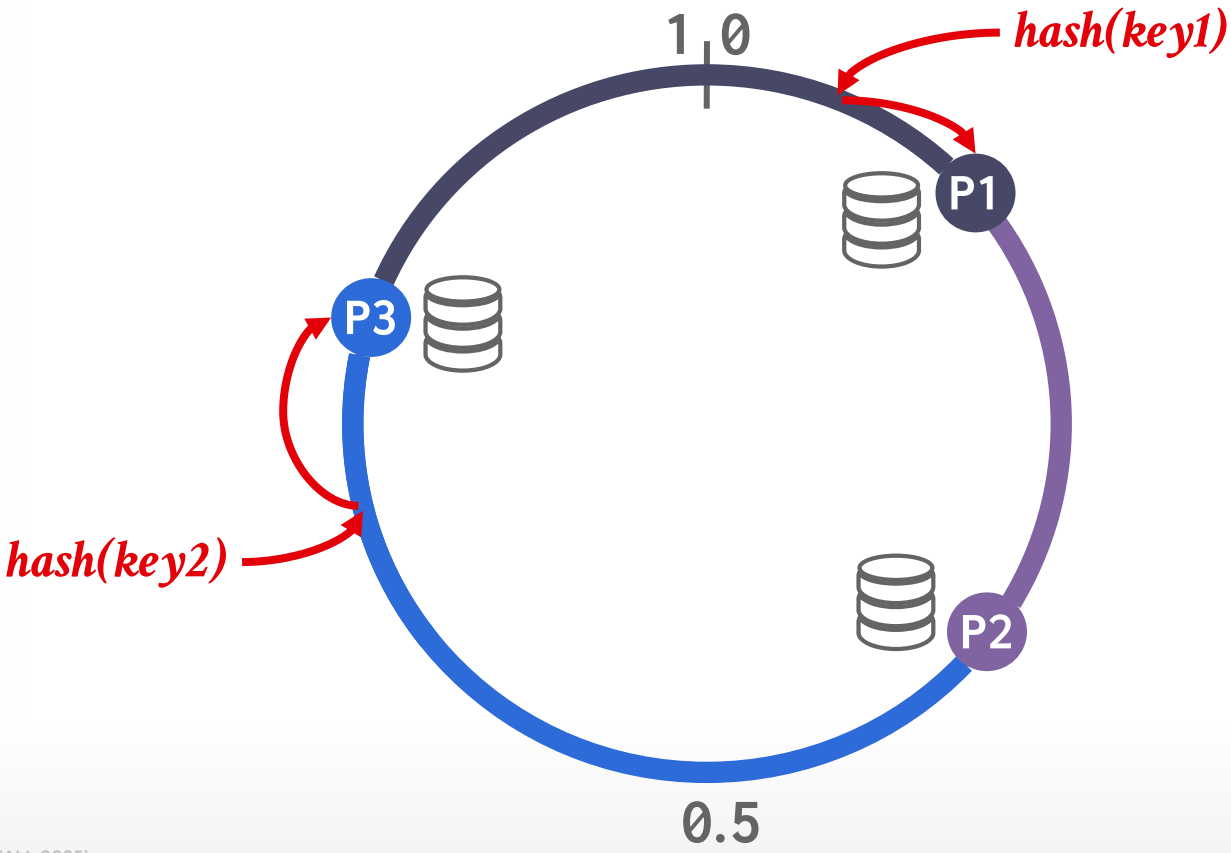
CONSISTENT HASHING



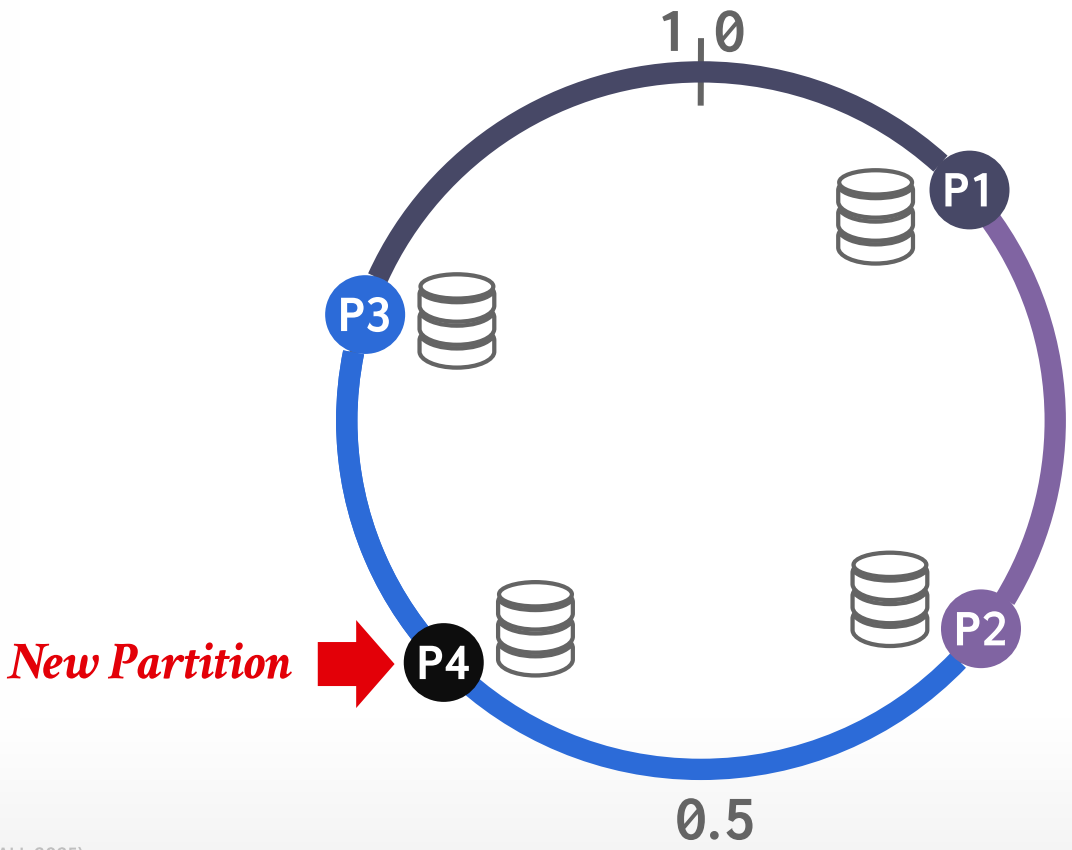
CONSISTENT HASHING



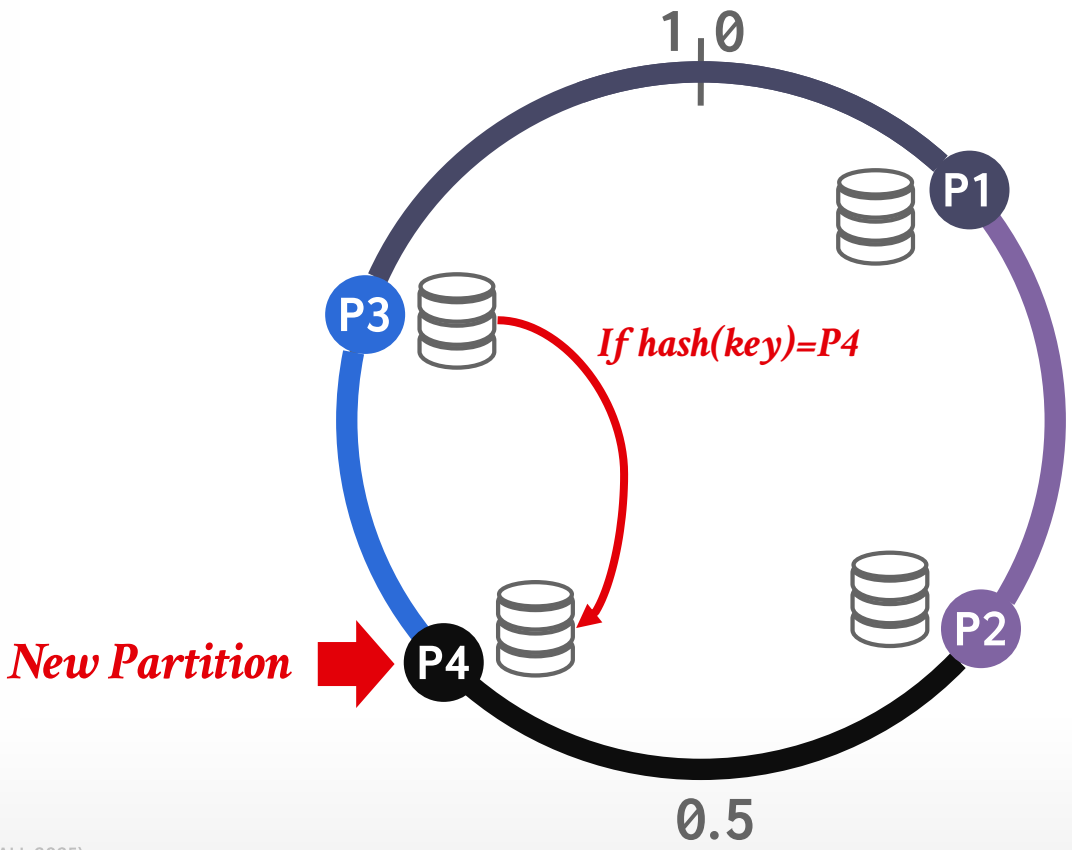
CONSISTENT HASHING



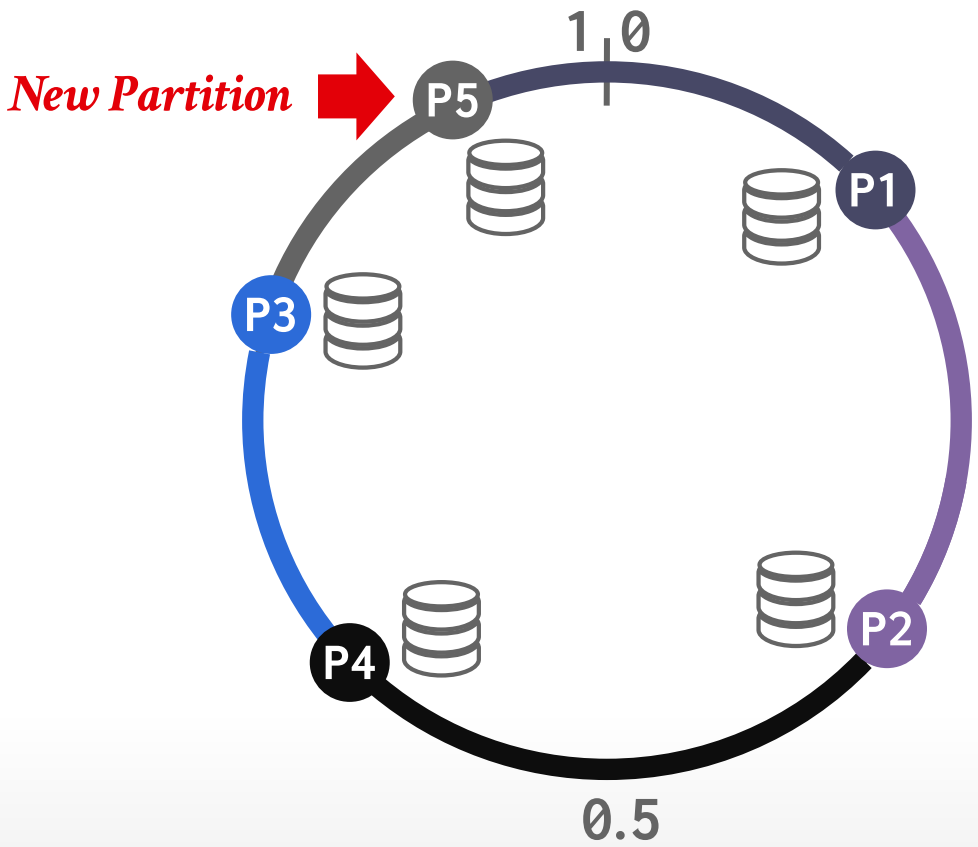
CONSISTENT HASHING



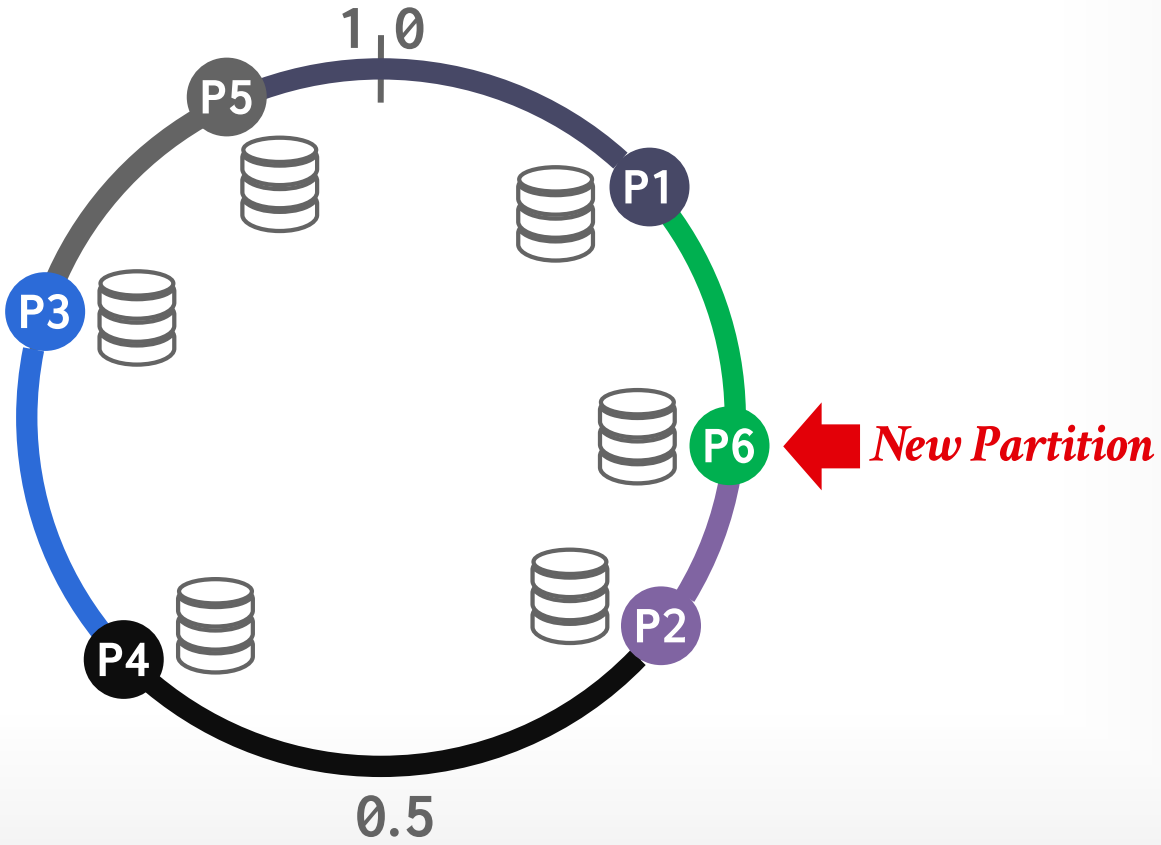
CONSISTENT HASHING



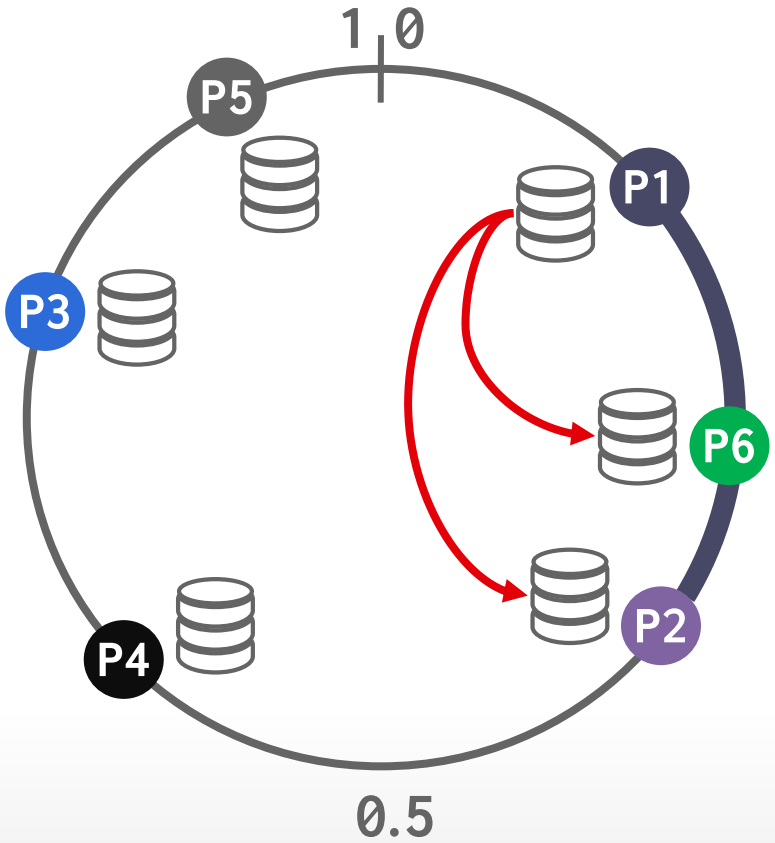
CONSISTENT HASHING



CONSISTENT HASHING



CONSISTENT HASHING



Replication Factor = 3

CONSISTENT HASHING

Couchbase

snowflake

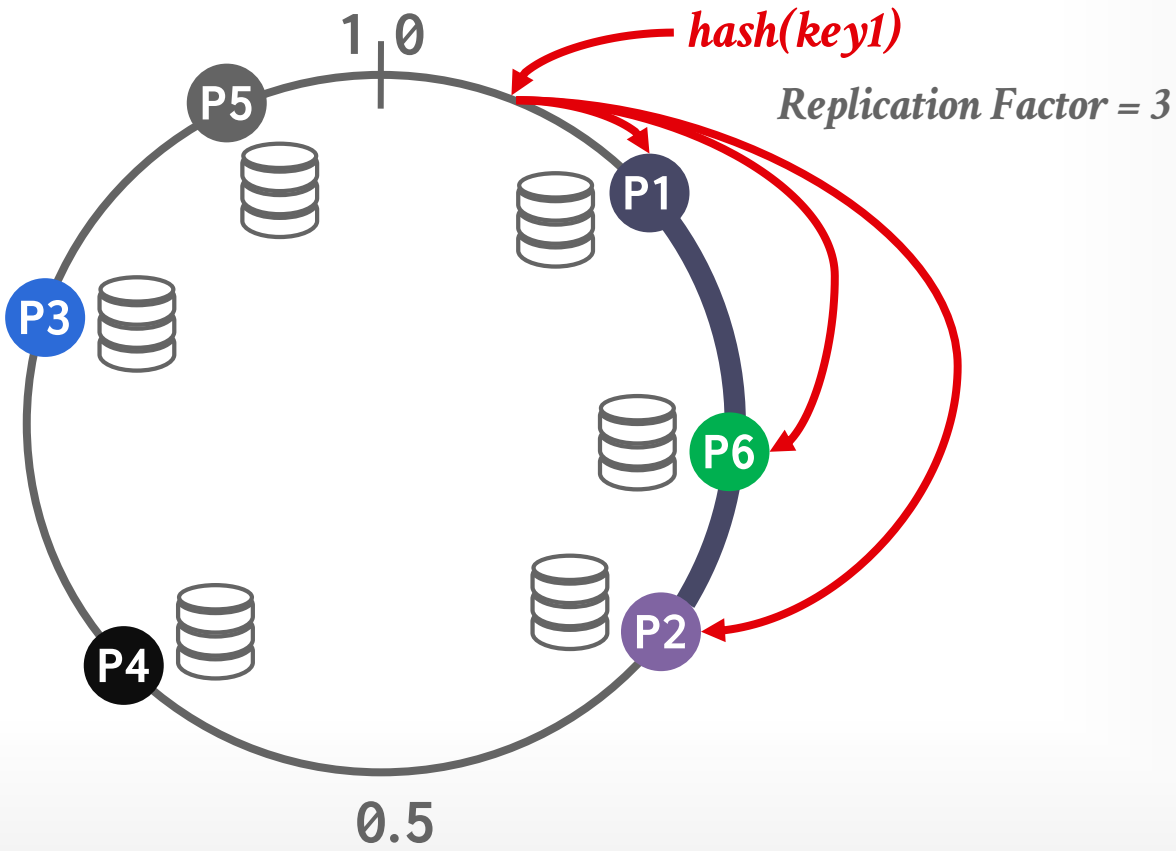
AEROSPIKE

MEMCACHED

cassandra

riak

SCYLLA.

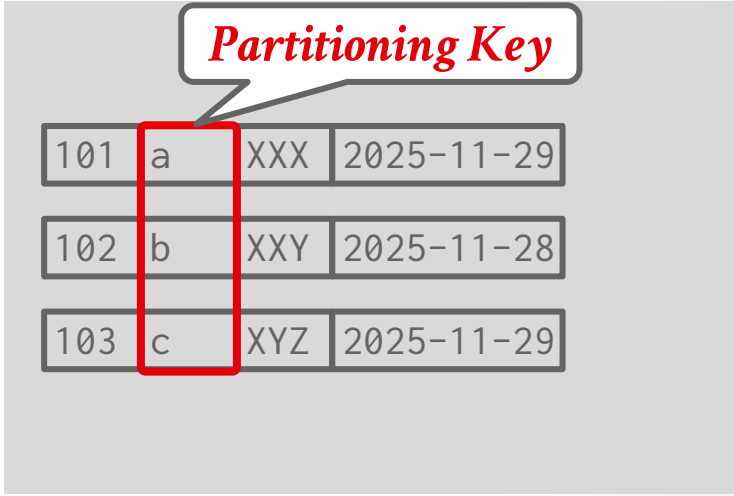


RENDEZVOUS HASHING

For each key, generate a hash value per partition by concatenating the partition's identifier to hashed key.

Assign key to a partition based on the hash value with the highest weight.

Consistent Hashing is a specialized form of Rendezvous Hashing.



Yellowbrick apache Ignite druid kafka

RENDEZVOUS HASHING

For each key, generate a hash value per partition by concatenating the partition's identifier to hashed key.

Assign key to a partition based on the hash value with the highest weight.

Consistent Hashing is a specialized form of Rendezvous Hashing.

hash(a + node1) = 100

hash(a + node2) = 90

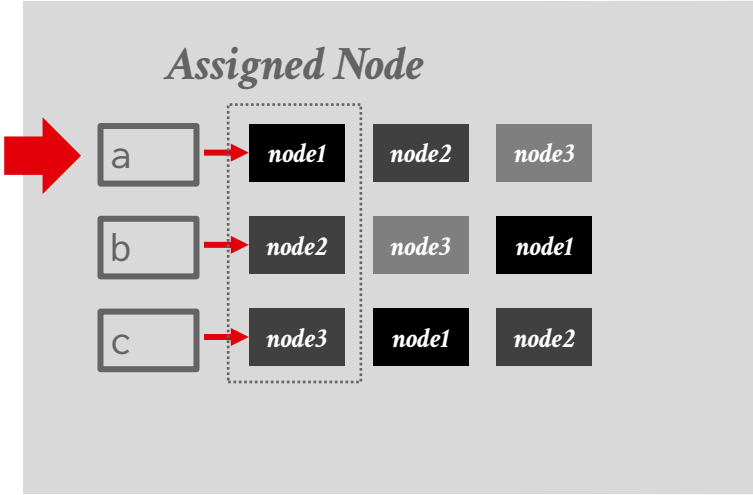
hash(a + node3) = 80

RENDEZVOUS HASHING

For each key, generate a hash value per partition by concatenating the partition's identifier to hashed key.

Assign key to a partition based on the hash value with the highest weight.

Consistent Hashing is a specialized form of Rendezvous Hashing.

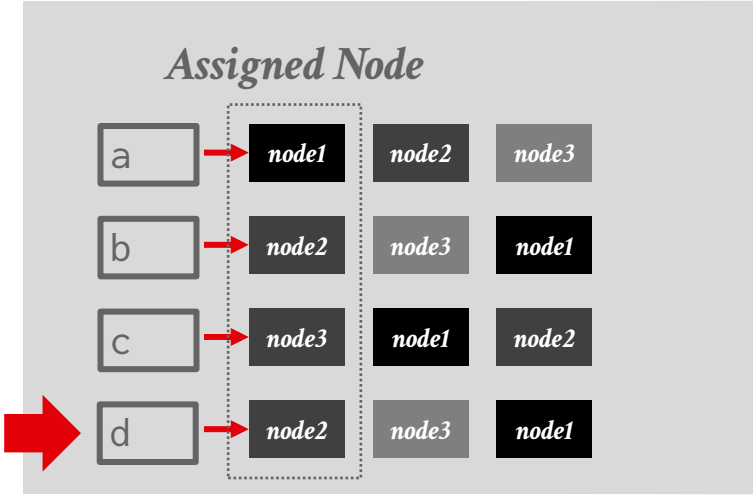


RENDEZVOUS HASHING

For each key, generate a hash value per partition by concatenating the partition's identifier to hashed key.

Assign key to a partition based on the hash value with the highest weight.

Consistent Hashing is a specialized form of Rendezvous Hashing.



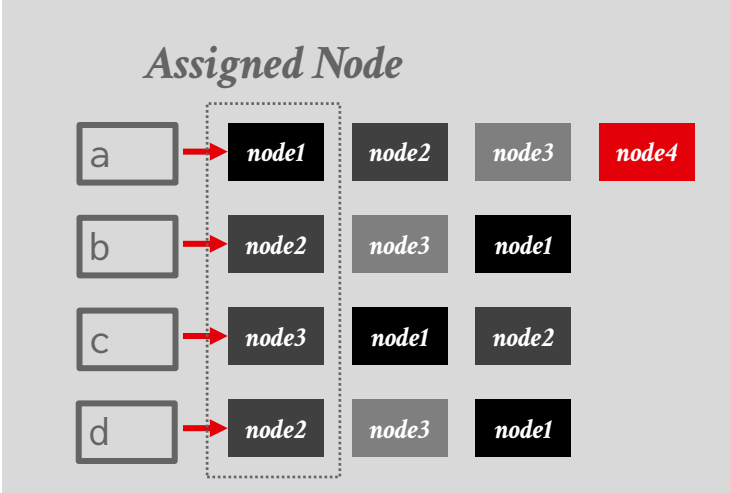
Yellowbrick apache Ignite druid kafka

RENDEZVOUS HASHING

For each key, generate a hash value per partition by concatenating the partition's identifier to hashed key.

Assign key to a partition based on the hash value with the highest weight.

Consistent Hashing is a specialized form of Rendezvous Hashing.

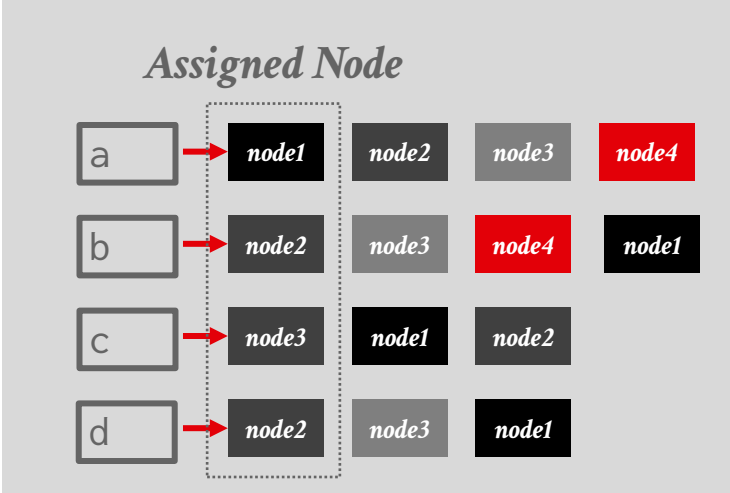


RENDEZVOUS HASHING

For each key, generate a hash value per partition by concatenating the partition's identifier to hashed key.

Assign key to a partition based on the hash value with the highest weight.

Consistent Hashing is a specialized form of Rendezvous Hashing.



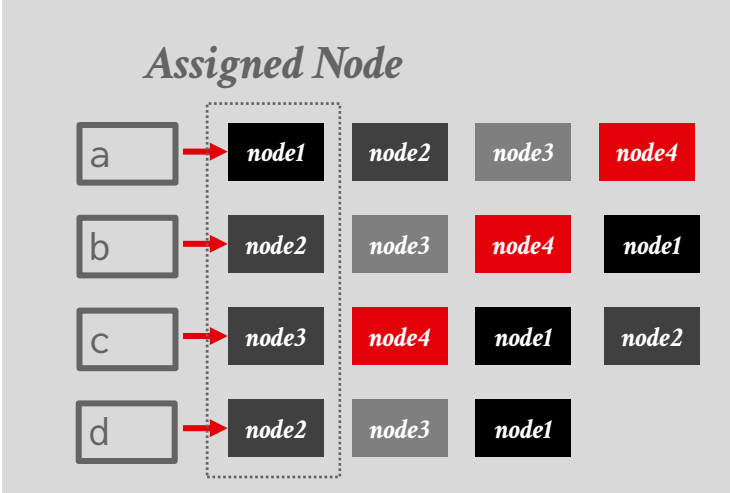
Yellowbrick apache Ignite druid kafka

RENDEZVOUS HASHING

For each key, generate a hash value per partition by concatenating the partition's identifier to hashed key.

Assign key to a partition based on the hash value with the highest weight.

Consistent Hashing is a specialized form of Rendezvous Hashing.

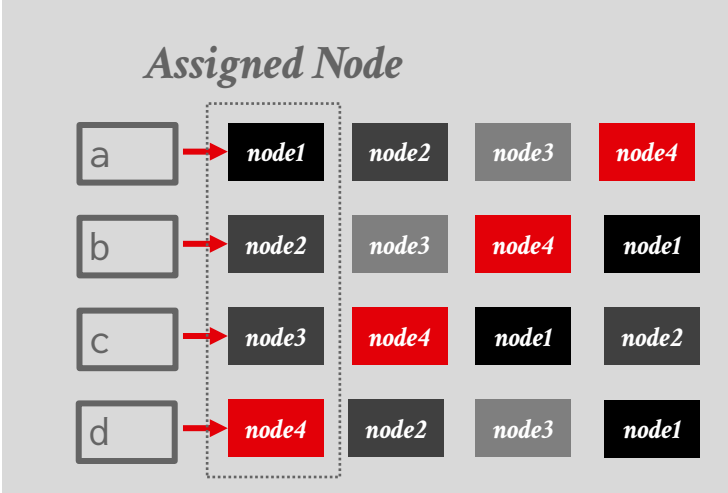


RENDEZVOUS HASHING

For each key, generate a hash value per partition by concatenating the partition's identifier to hashed key.

Assign key to a partition based on the hash value with the highest weight.

Consistent Hashing is a specialized form of Rendezvous Hashing.



REPLICATION

The DBMS can replicate a database across redundant nodes to increase availability.

→ Partitioned vs. Non-Partitioned
→ Shared-Nothing vs. Shared-Disk

Design Decisions:

→ Replica Configuration
→ Propagation Scheme
→ Propagation Timing
→ Update Method

REPLICA CONFIGURATIONS

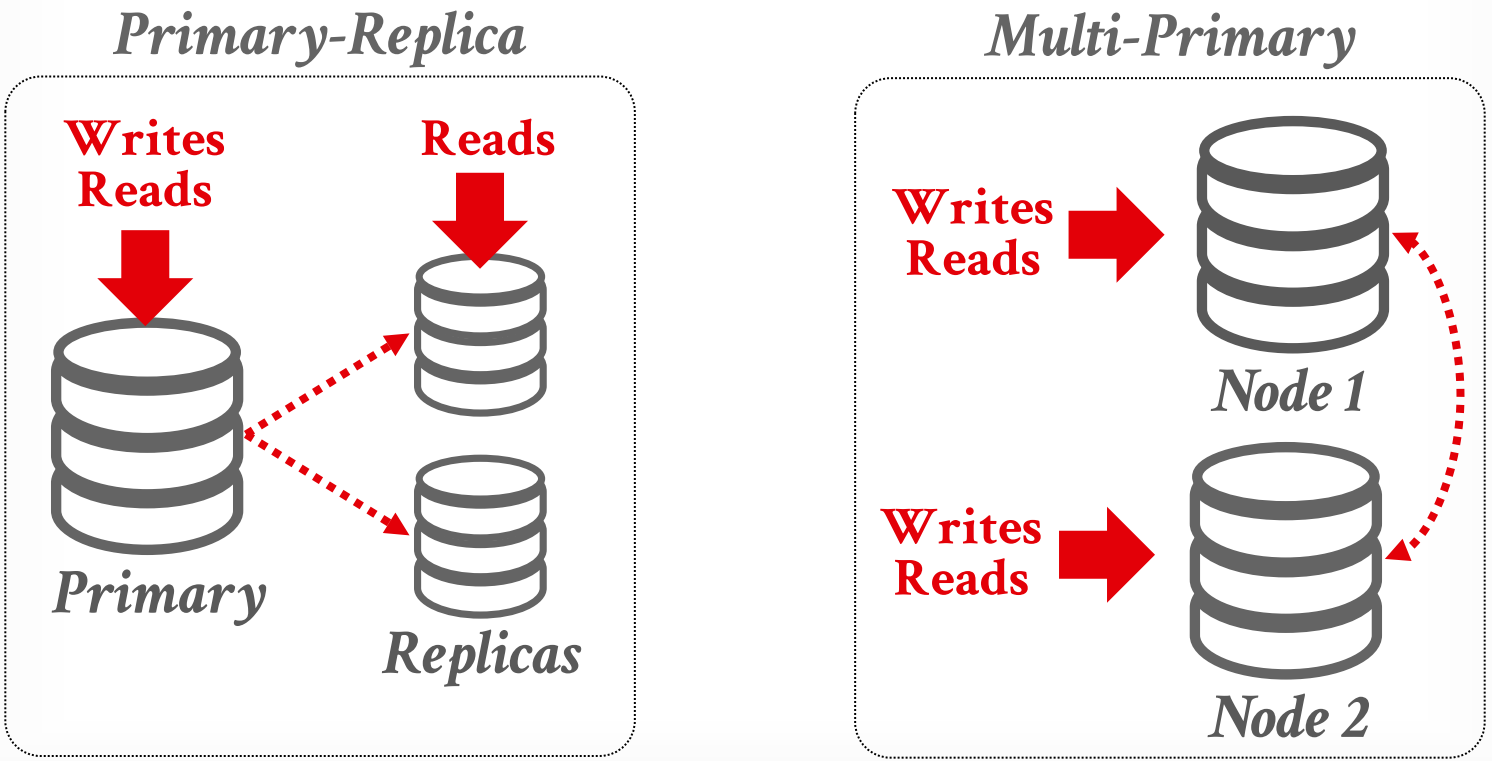
Approach #1: Primary-Replica

→ All updates go to a designated primary for each object.
→ The primary propagates updates to its replicas by shipping logs.
→ Read-only txns may be allowed to access replicas.
→ If the primary goes down, then hold an election to select a new primary.

Approach #2: Multi-Primary

→ Txns can update data objects at any replica.
→ Replicas must synchronize with each other using an atomic commit protocol.

REPLICA CONFIGURATIONS



K-SAFETY

***K*-safety is a threshold for determining the fault tolerance of the replicated database.**

The value *K* represents the number of replicas per data object that must always be available.

If the number of replicas goes below this threshold, then the DBMS halts execution and takes itself offline.

PROPAGATION SCHEME

When a txn commits on a replicated database, the DBMS decides whether it must wait for that txn's changes to propagate to other nodes before it can send the acknowledgement to application.

Propagation levels:

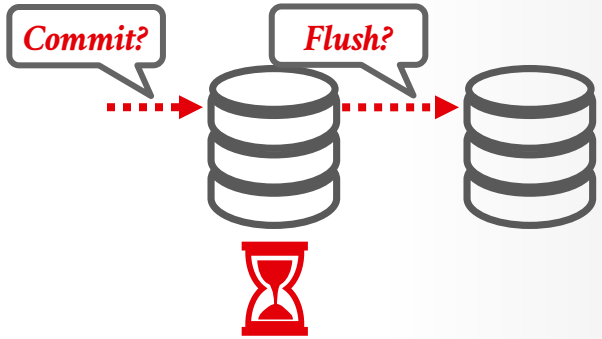
→ Synchronous (*Strong Consistency*)

→ Asynchronous (*Eventual Consistency*)

PROPAGATION SCHEME

Approach #1: Synchronous

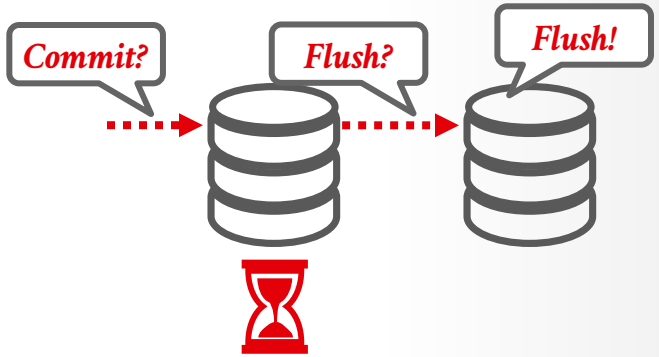
→ The primary sends updates to replicas and then waits for them to acknowledge that they fully applied (i.e., logged) the changes.



PROPAGATION SCHEME

Approach #1: Synchronous

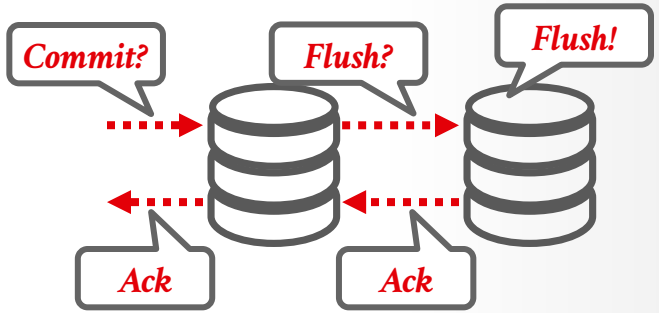
→ The primary sends updates to replicas and then waits for them to acknowledge that they fully applied (i.e., logged) the changes.



PROPAGATION SCHEME

Approach #1: Synchronous

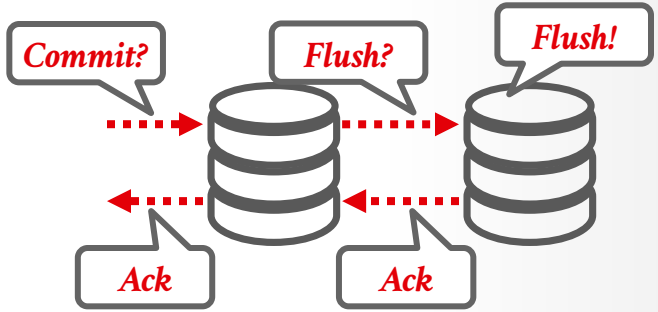
→ The primary sends updates to replicas and then waits for them to acknowledge that they fully applied (i.e., logged) the changes.



PROPAGATION SCHEME

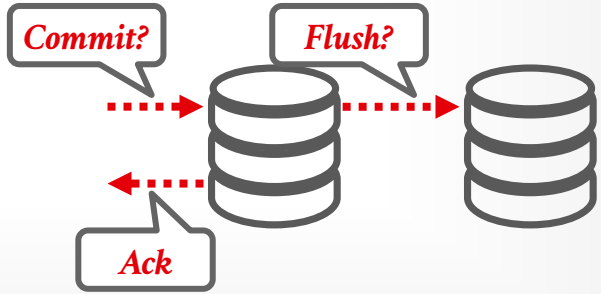
Approach #1: Synchronous

→ The primary sends updates to replicas and then waits for them to acknowledge that they fully applied (i.e., logged) the changes.



Approach #2: Asynchronous

→ The primary immediately returns the acknowledgement to the client without waiting for replicas to apply the changes.



PROPAGATION TIMING

Approach #1: Continuous

→ The DBMS sends log messages immediately as it generates them.
→ Also need to send a commit/abort message.

Approach #2: On Commit

→ The DBMS only sends the log messages for a txn to the replicas once the txn is commits.
→ Do not waste time sending log records for aborted txns.

UPDATE METHOD

Approach #1: Active-Active

→ A txn executes at each replica independently.
→ Need to check at the end whether the txn ends up with the same result at each replica.

Approach #2: Active-Passive

→ Each txn executes at a single location and propagates the changes to the replica.
→ Can either do physical or logical replication.
→ Not the same as Primary-Replica vs. Multi-Primary

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.

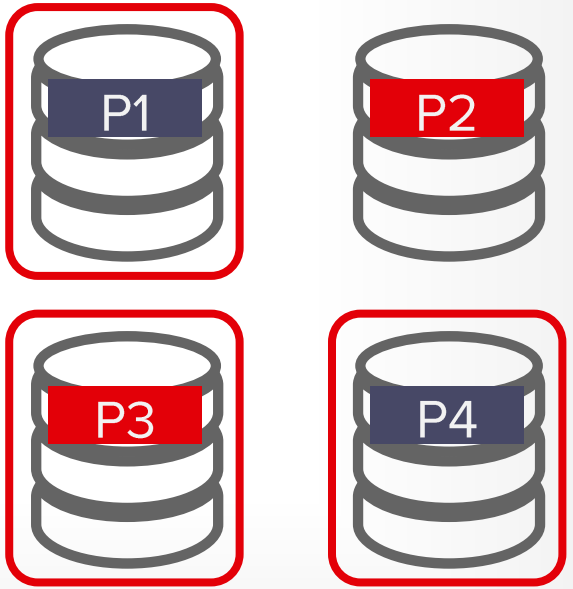
Most distributed DBMSs use a hybrid approach where they periodically elect some node to be a temporary coordinator.

CENTRALIZED COORDINATOR

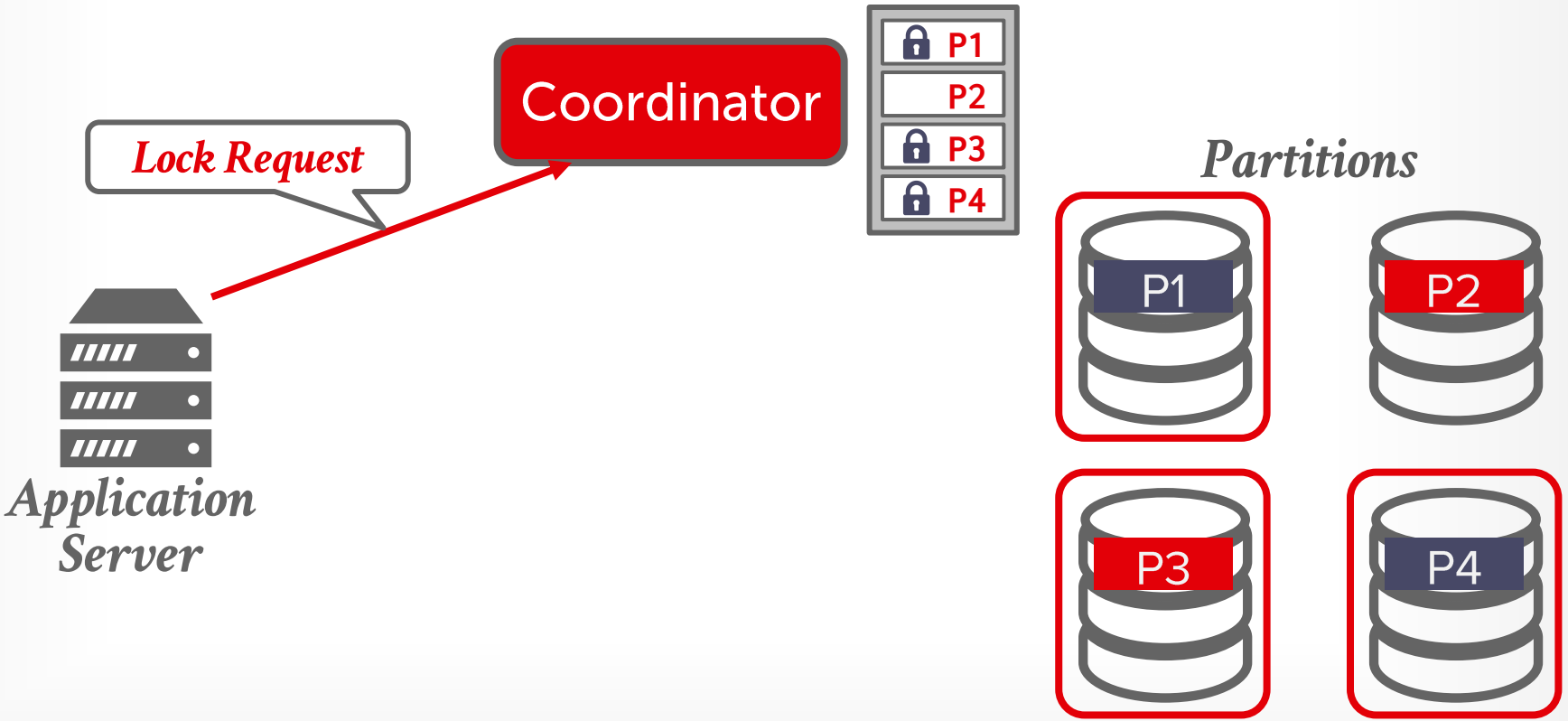
Coordinator

Application Server

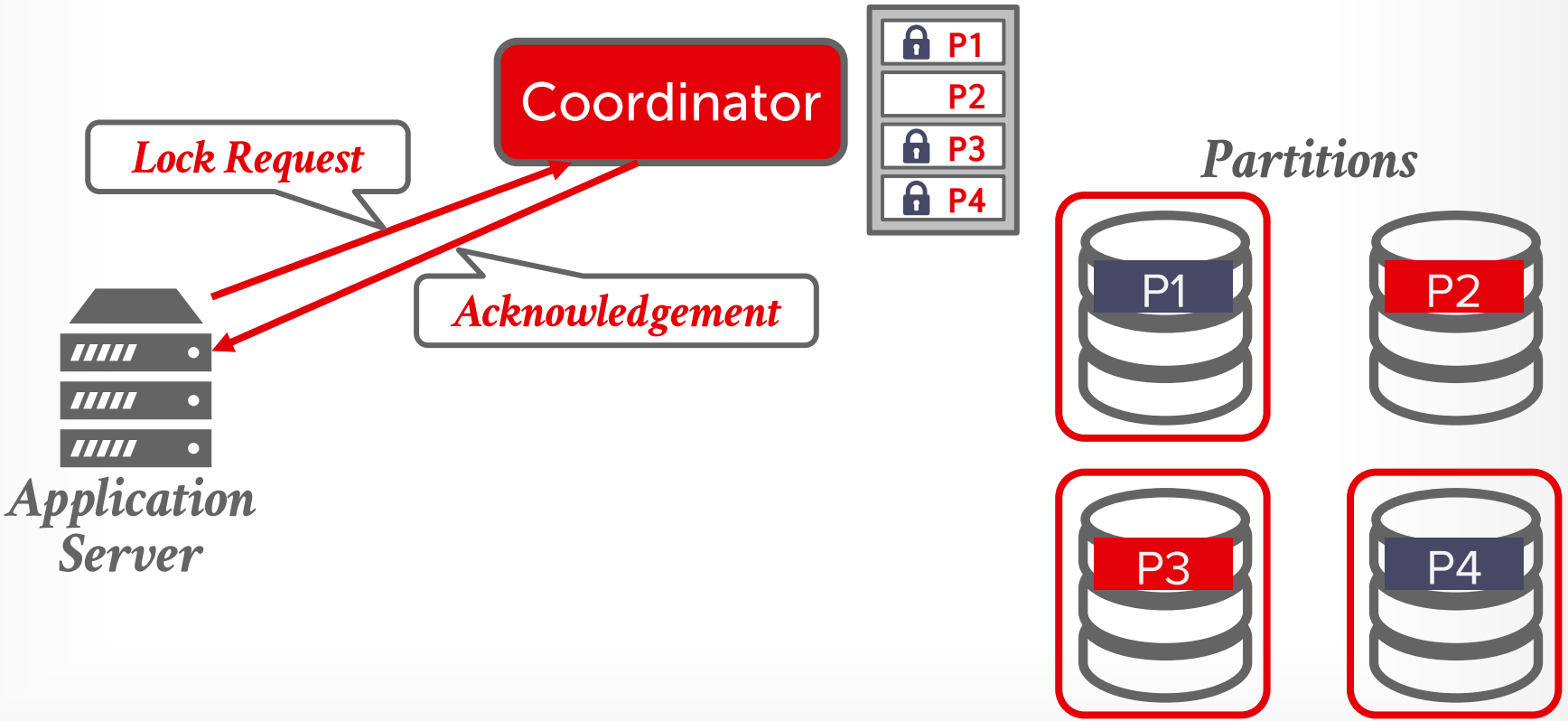
Partitions



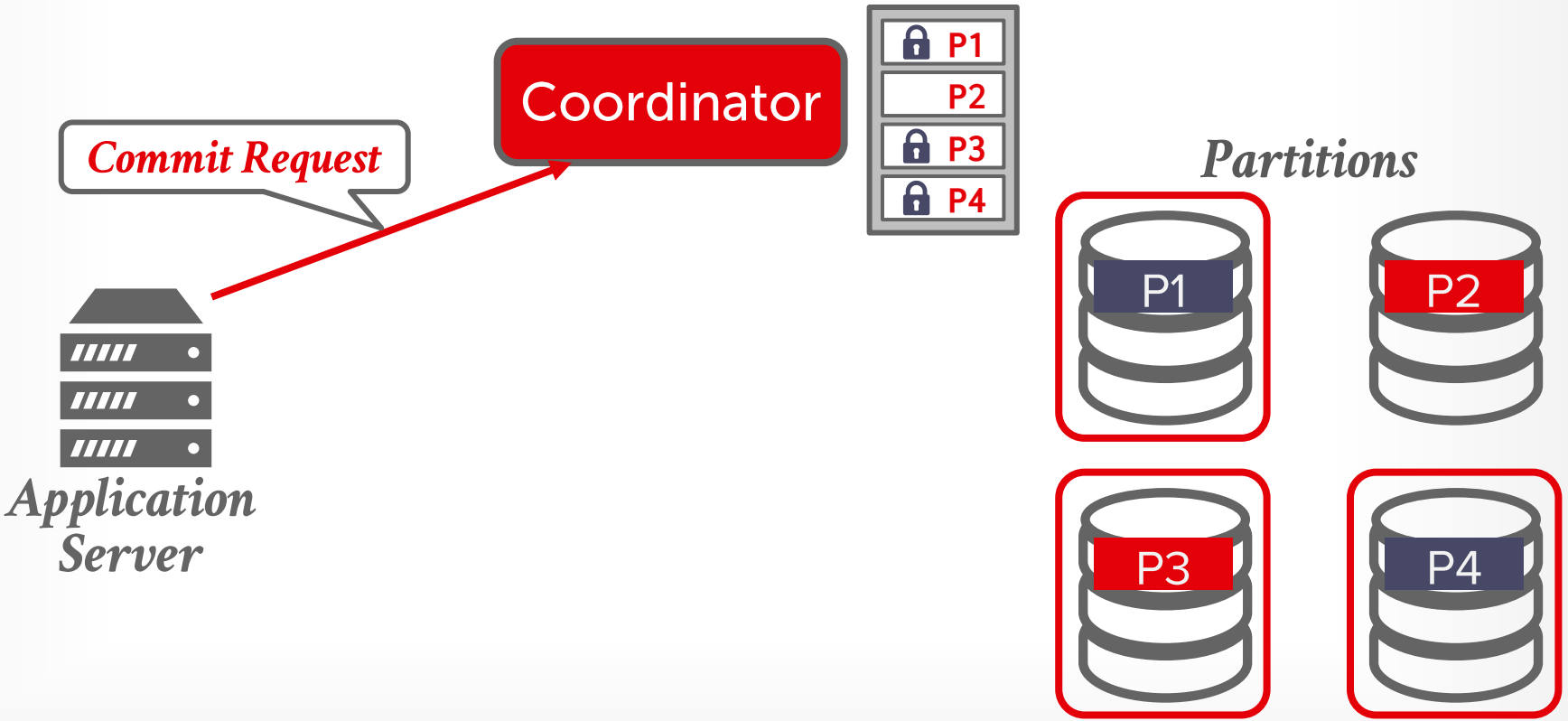
CENTRALIZED COORDINATOR



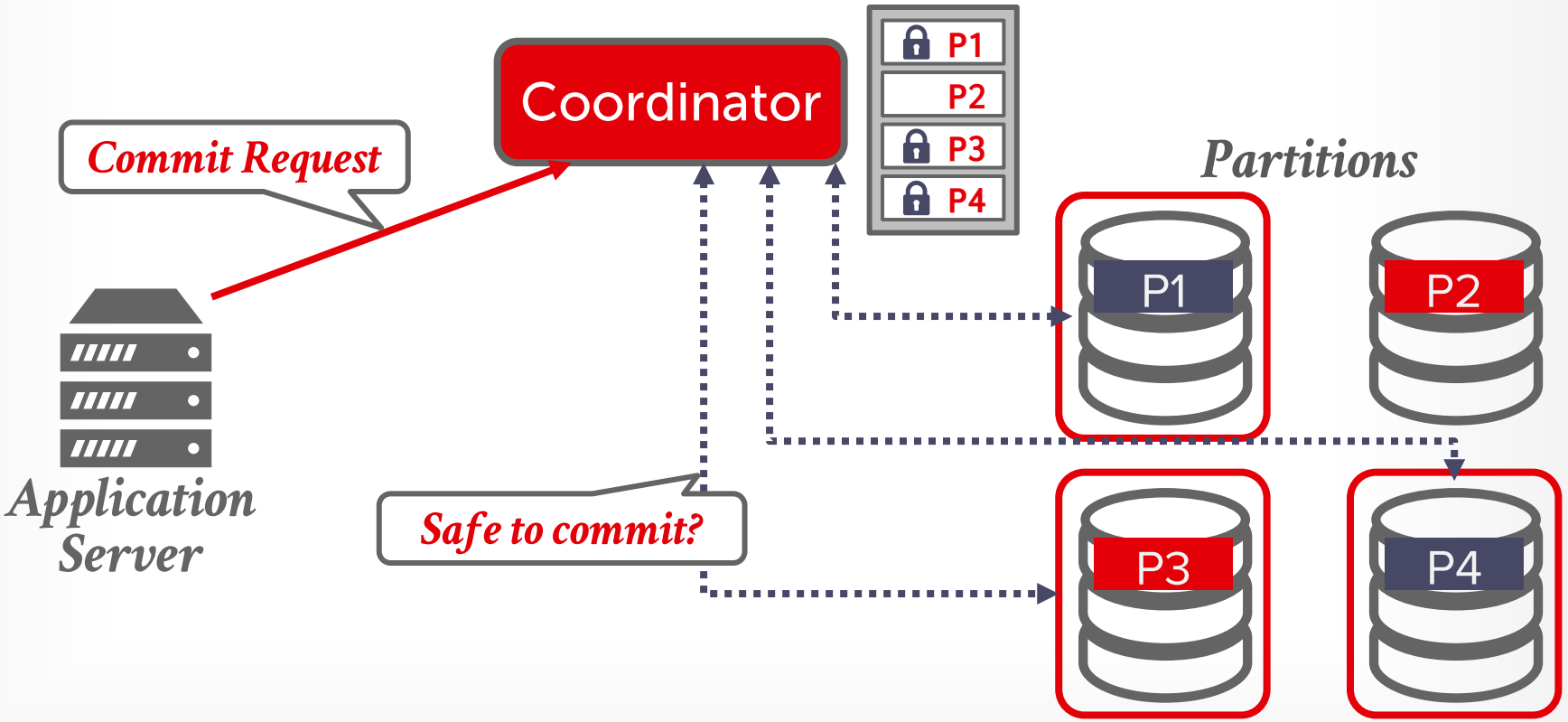
CENTRALIZED COORDINATOR



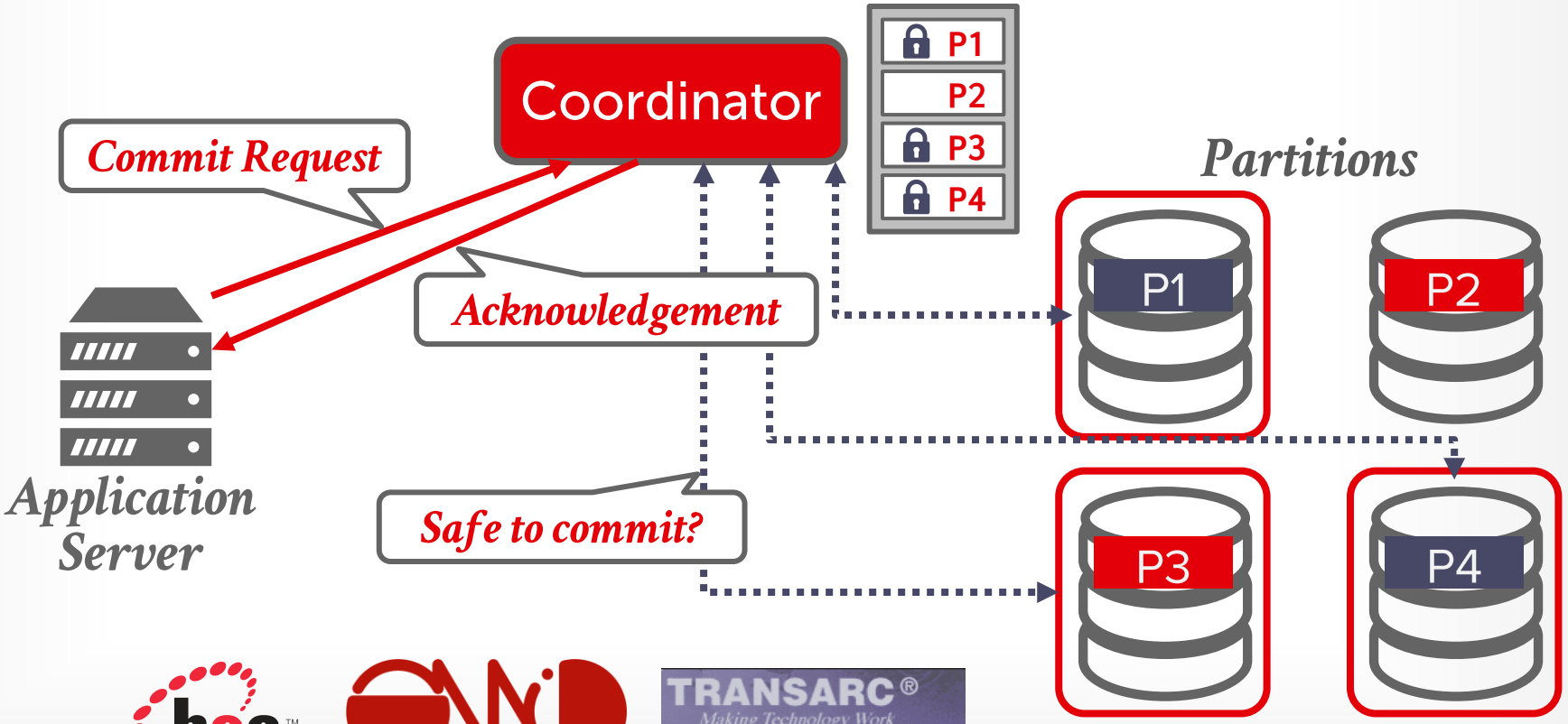
CENTRALIZED COORDINATOR



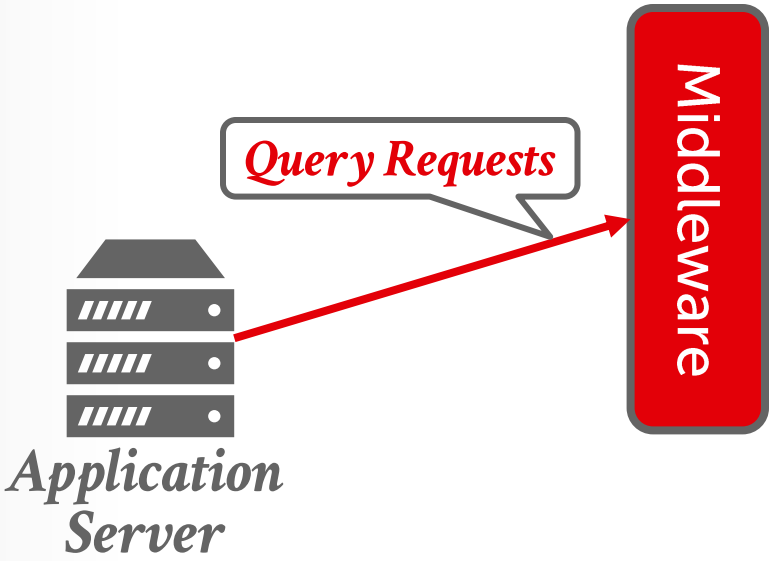
CENTRALIZED COORDINATOR

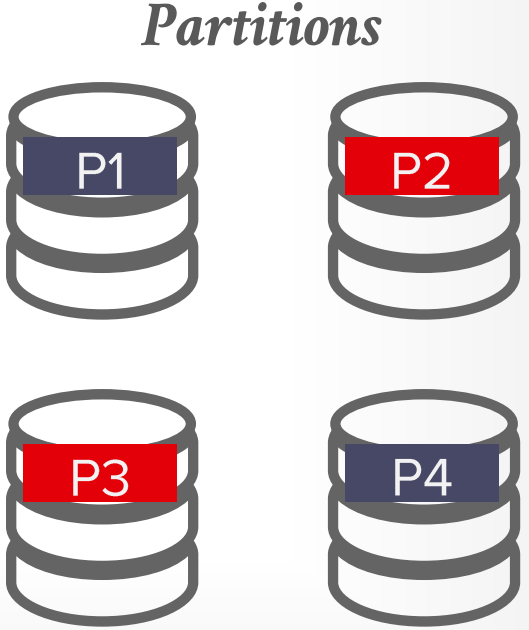


CENTRALIZED COORDINATOR

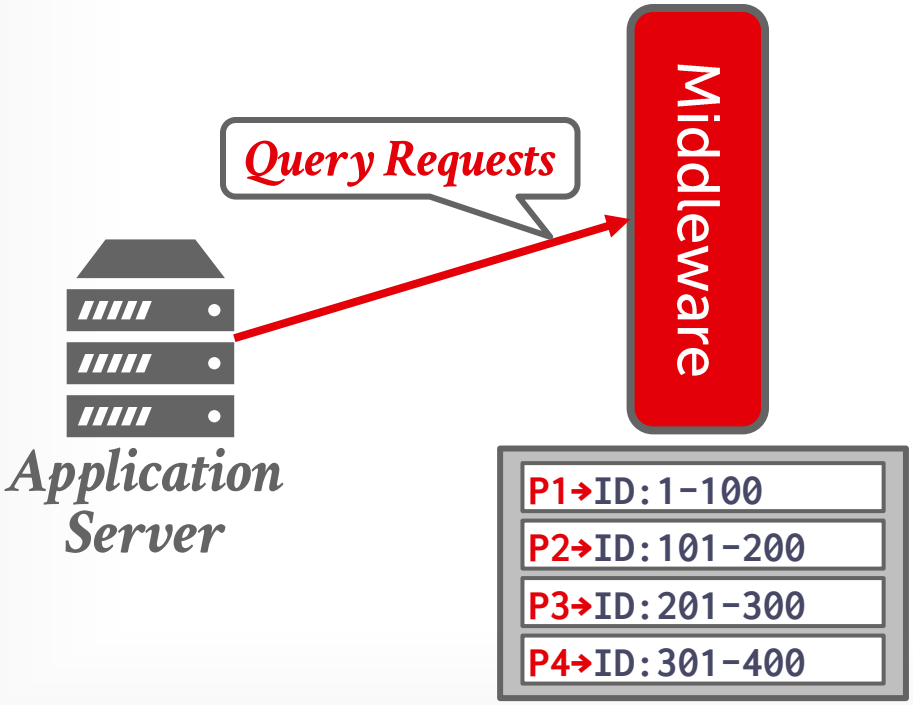


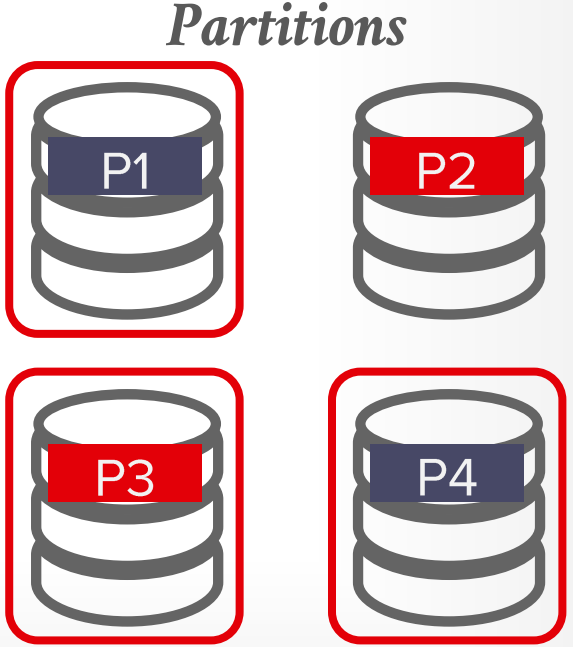
CENTRALIZED COORDINATOR



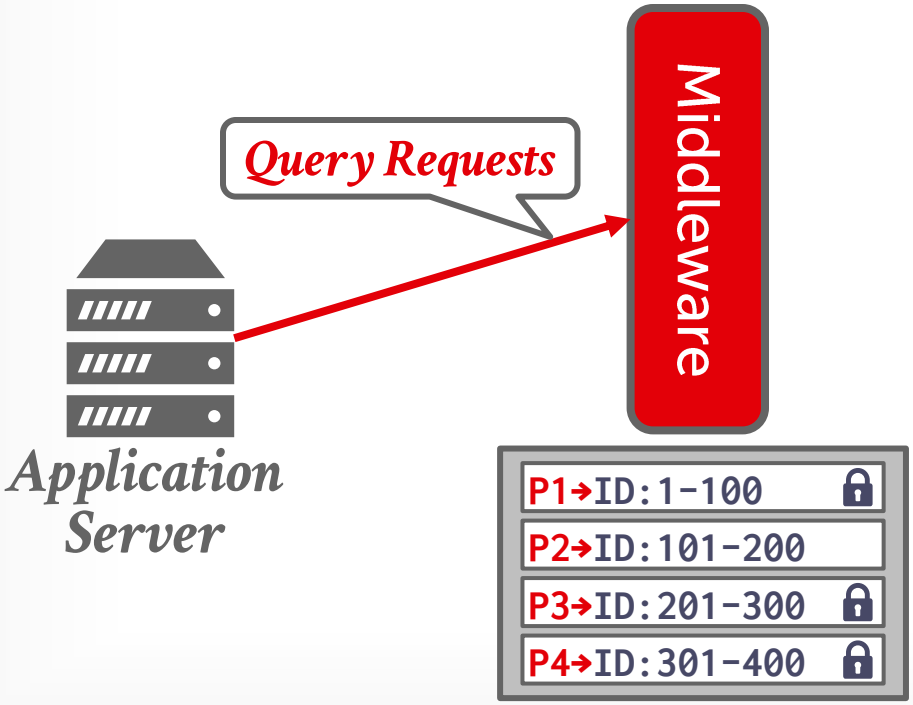


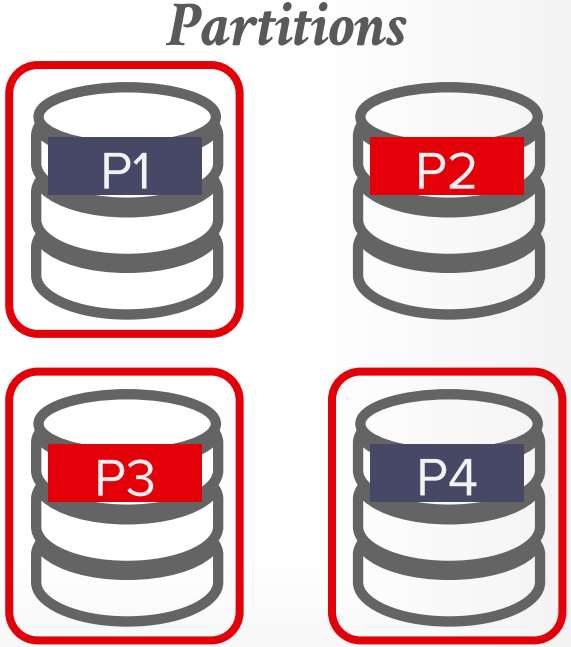
CENTRALIZED COORDINATOR



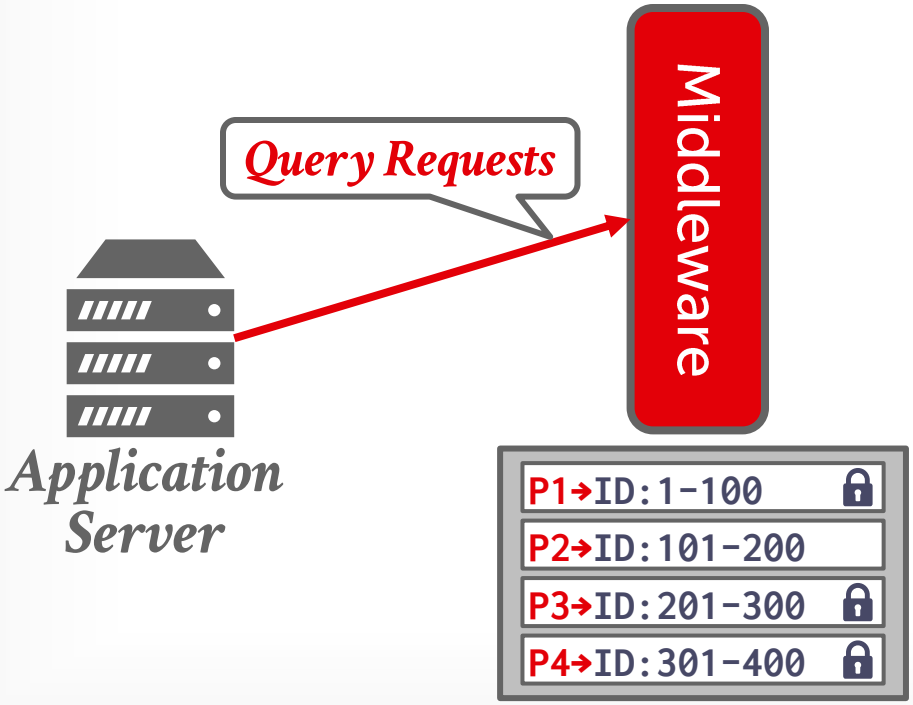


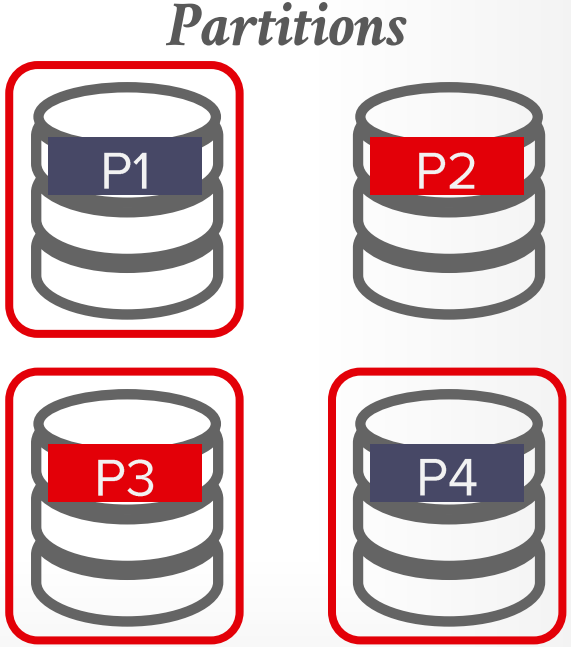
CENTRALIZED COORDINATOR



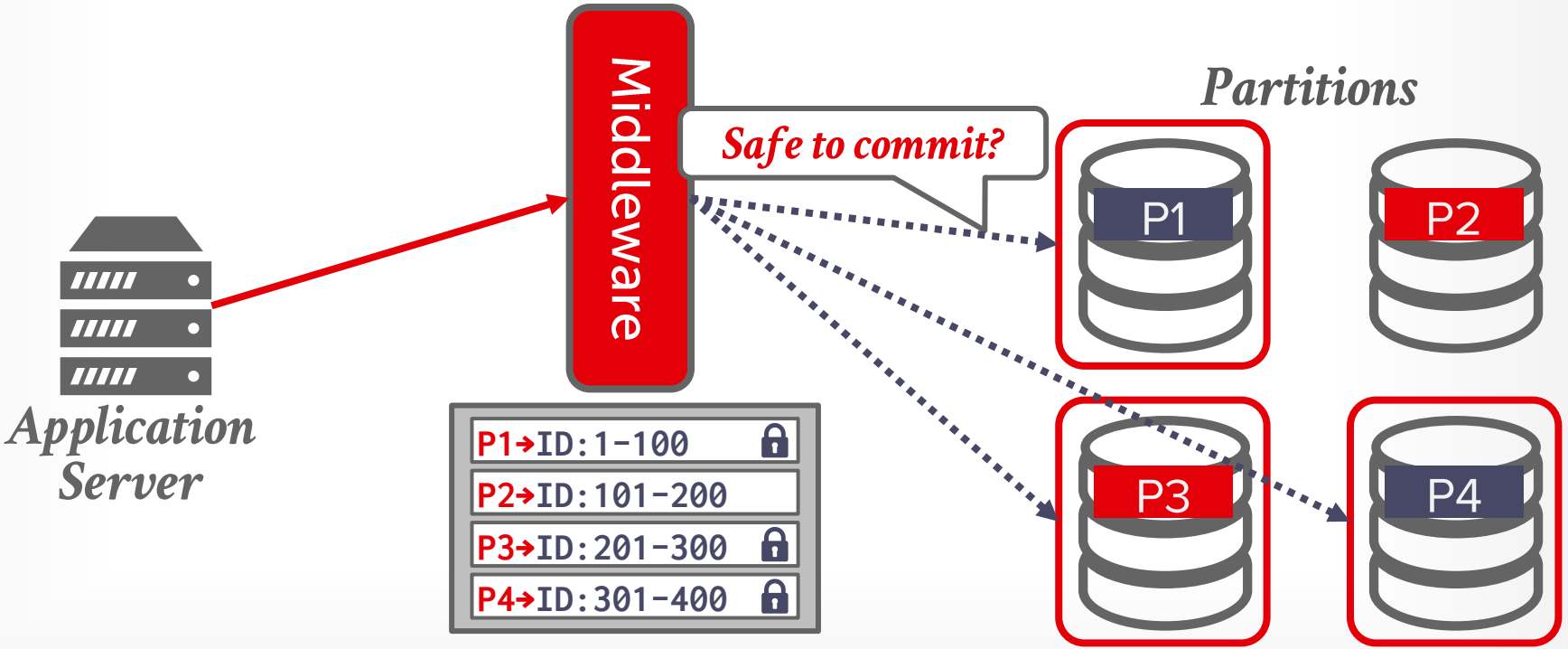


CENTRALIZED COORDINATOR





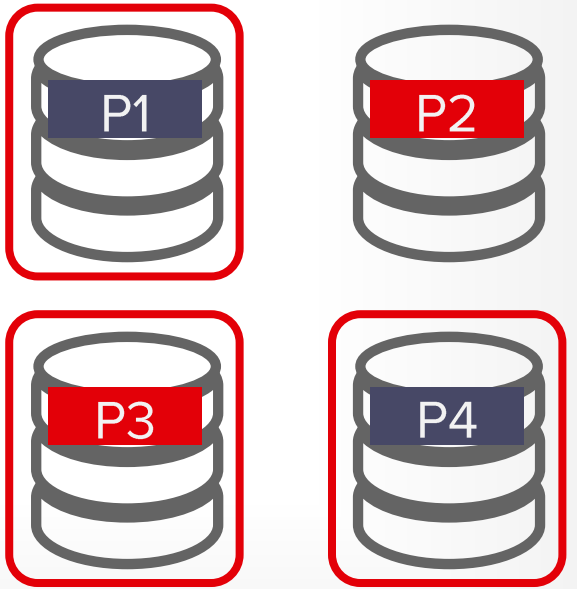
CENTRALIZED COORDINATOR



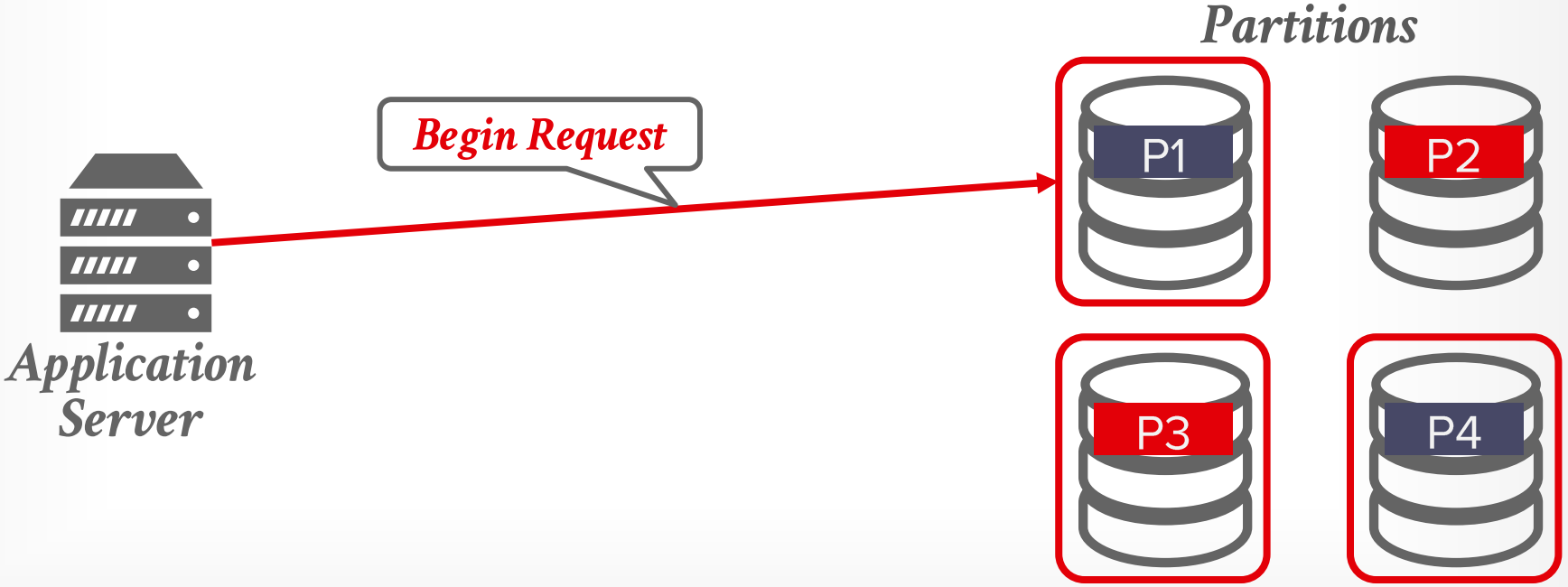
DECENTRALIZED COORDINATOR

Application Server

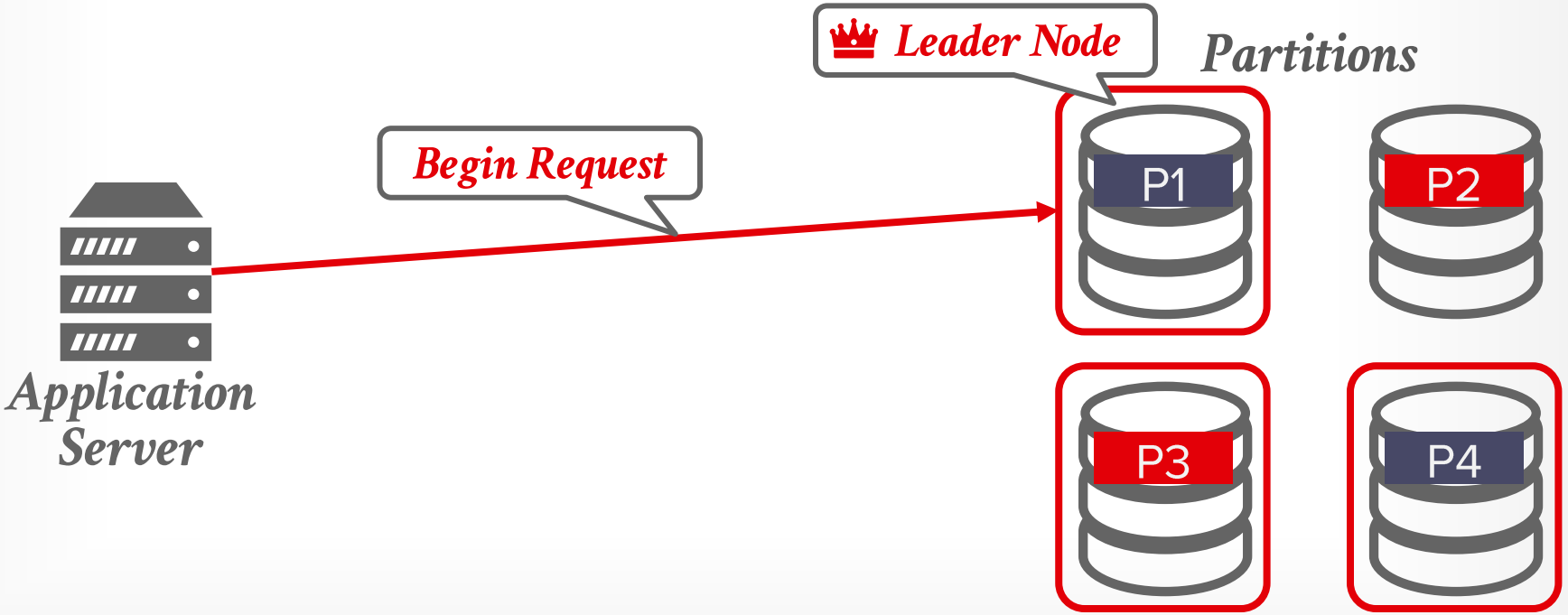
Partitions



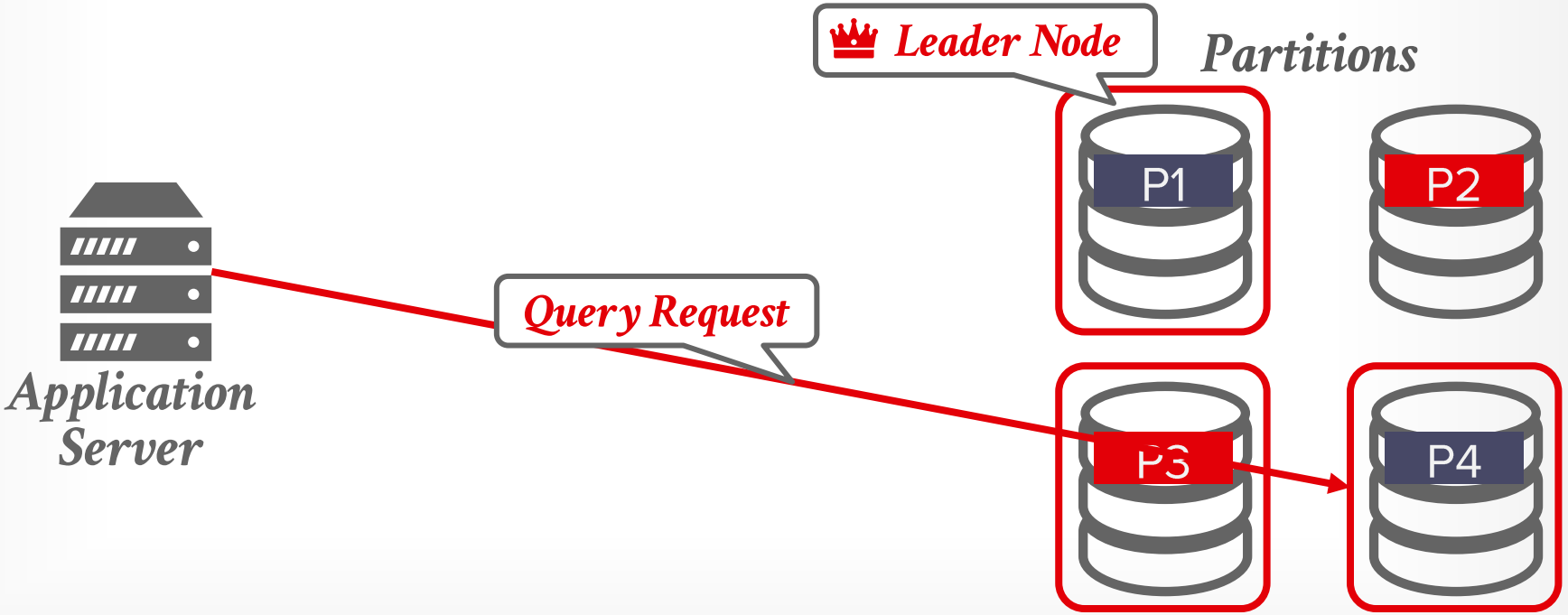
DECENTRALIZED COORDINATOR



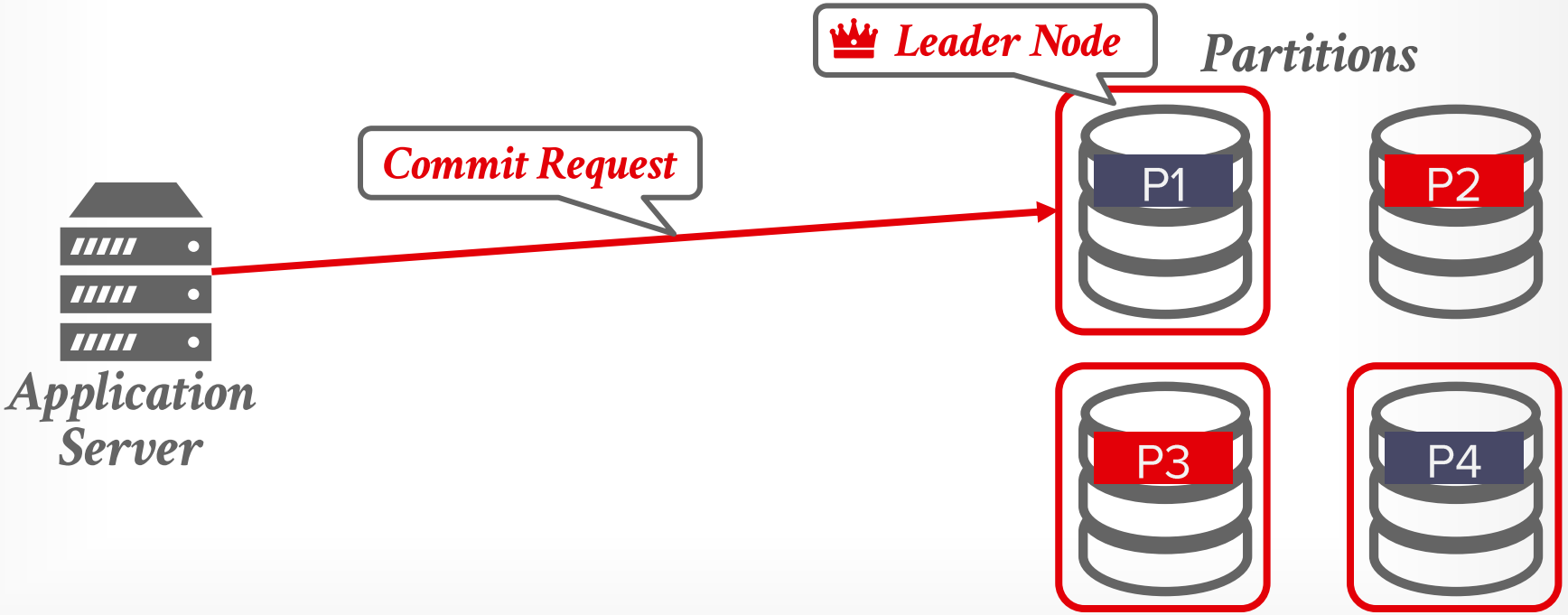
DECENTRALIZED COORDINATOR



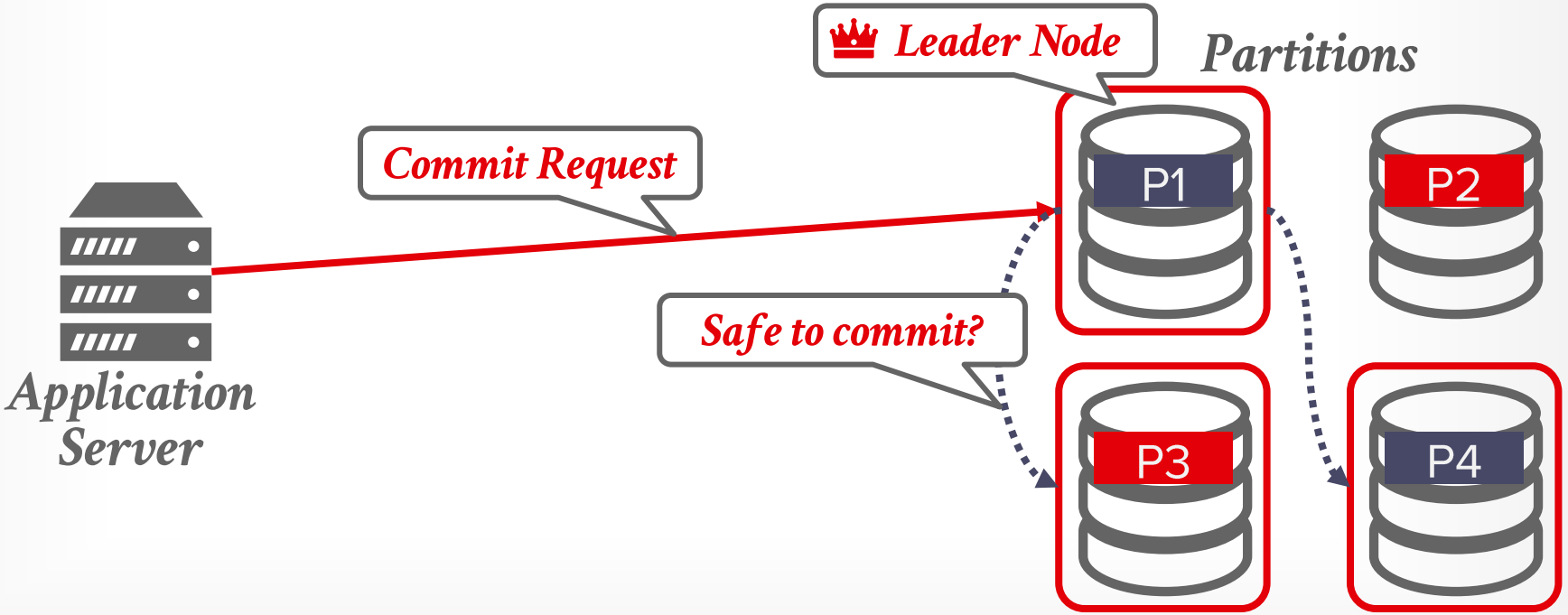
DECENTRALIZED COORDINATOR



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

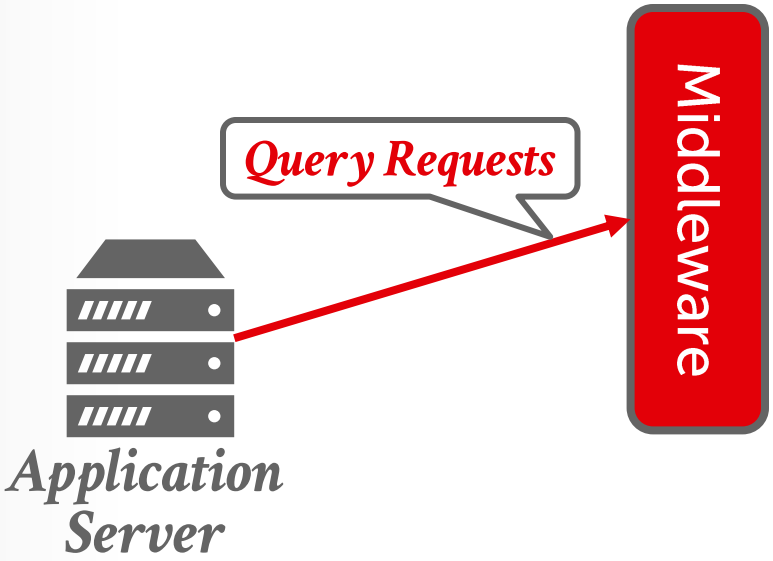
Distributed architecture that connects disparate DBMSs into a single logical system.

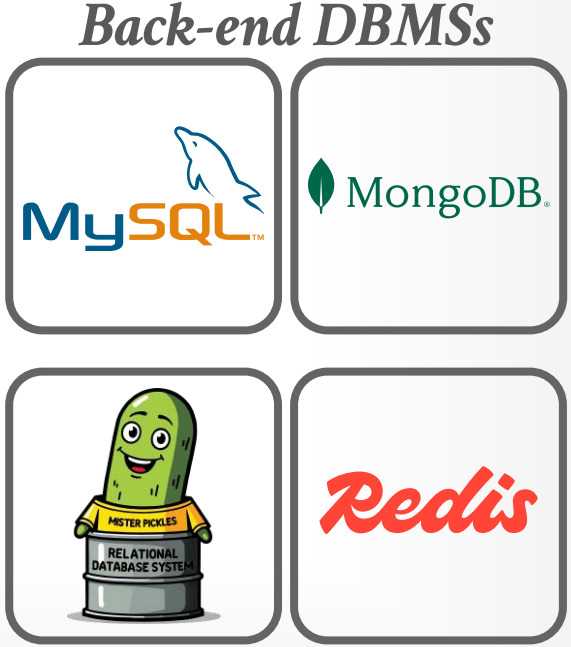
→ Expose a single query interface that can access data at any location.

This is hard and nobody does it well

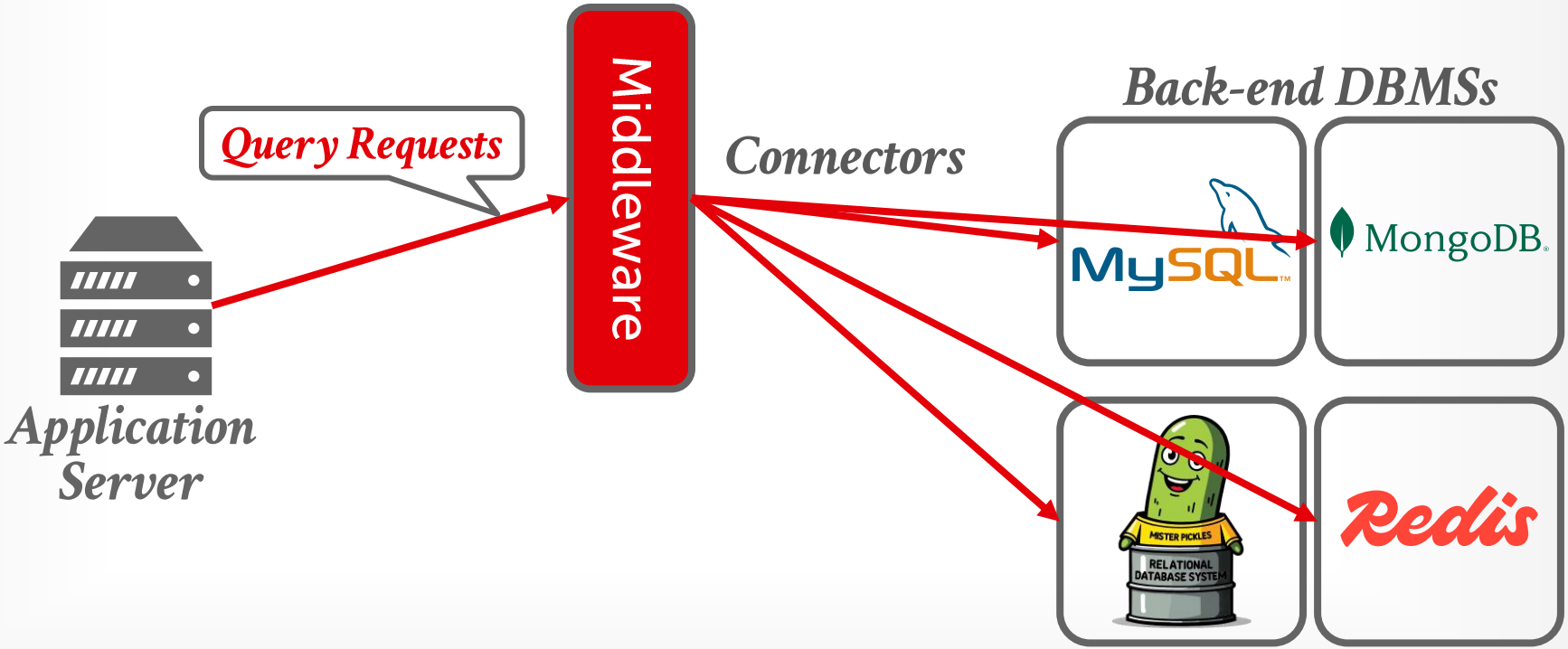
→ Different data models, query languages, limitations.
→ No easy way to optimize queries
→ Lots of data copying (bad).

FEDERATED DATABASE EXAMPLE





FEDERATED DATABASE EXAMPLE



DISTRIBUTED CONCURRENCY CONTROL

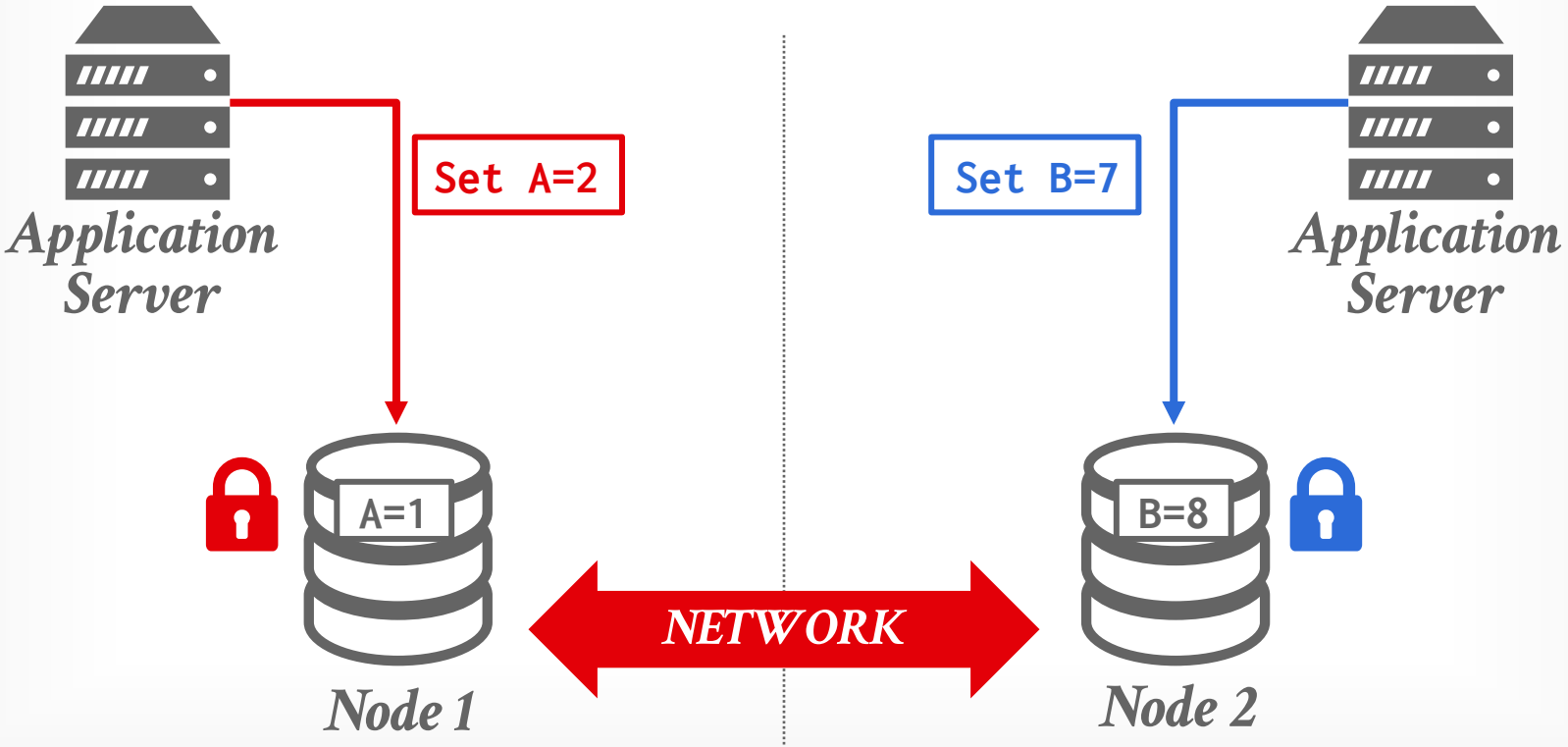
Need to allow multiple txns to execute simultaneously across multiple nodes.

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

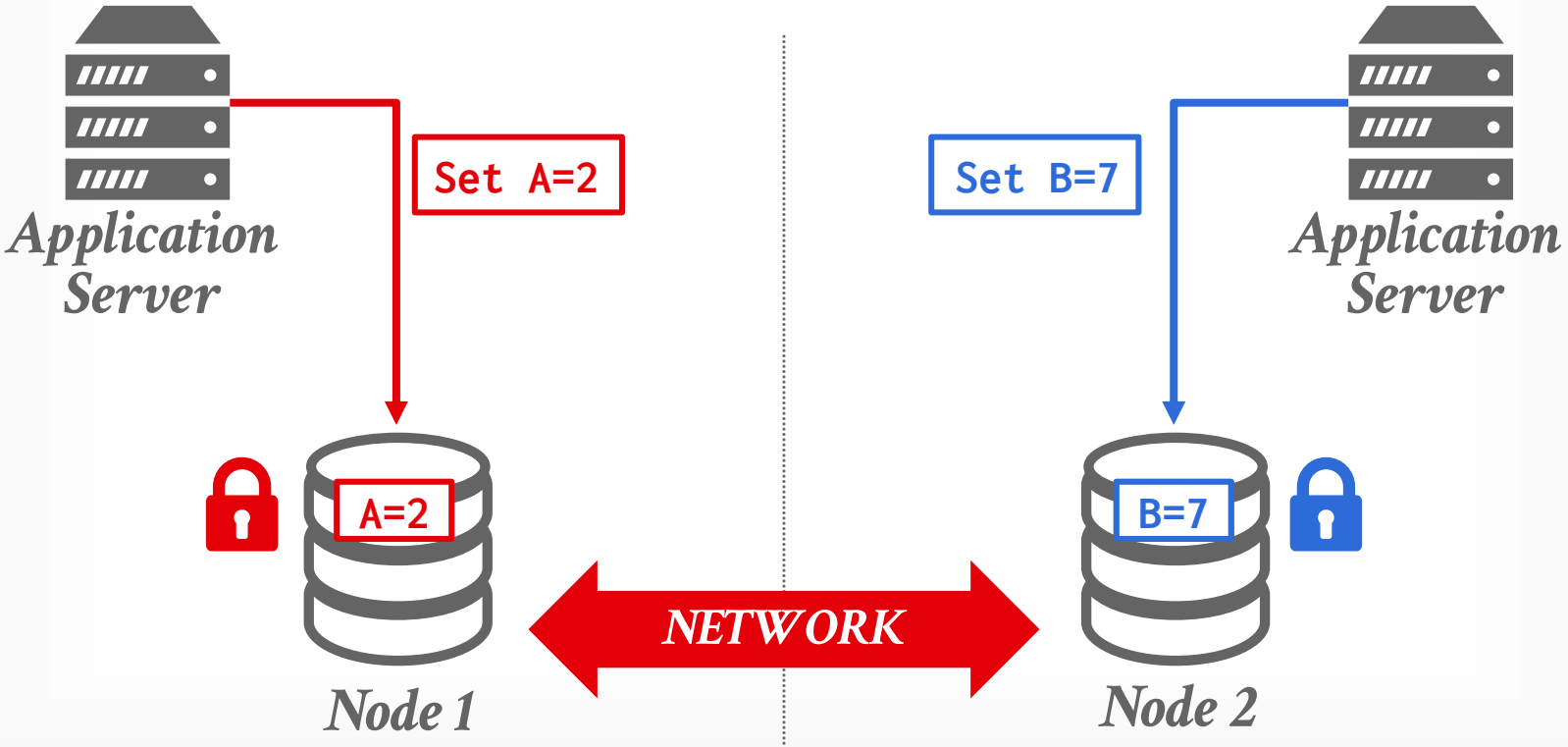
This is harder because of:

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

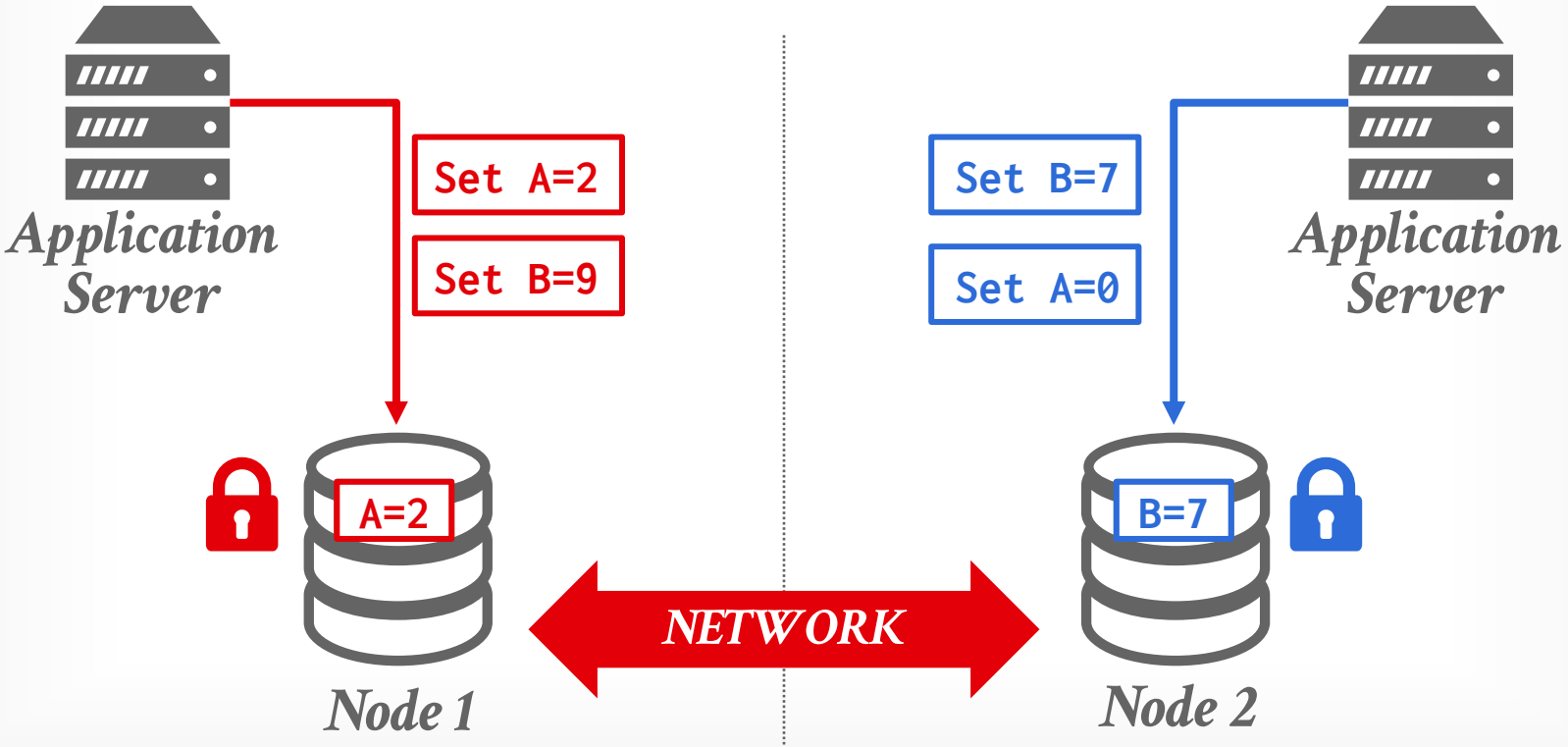
DISTRIBUTED 2PL



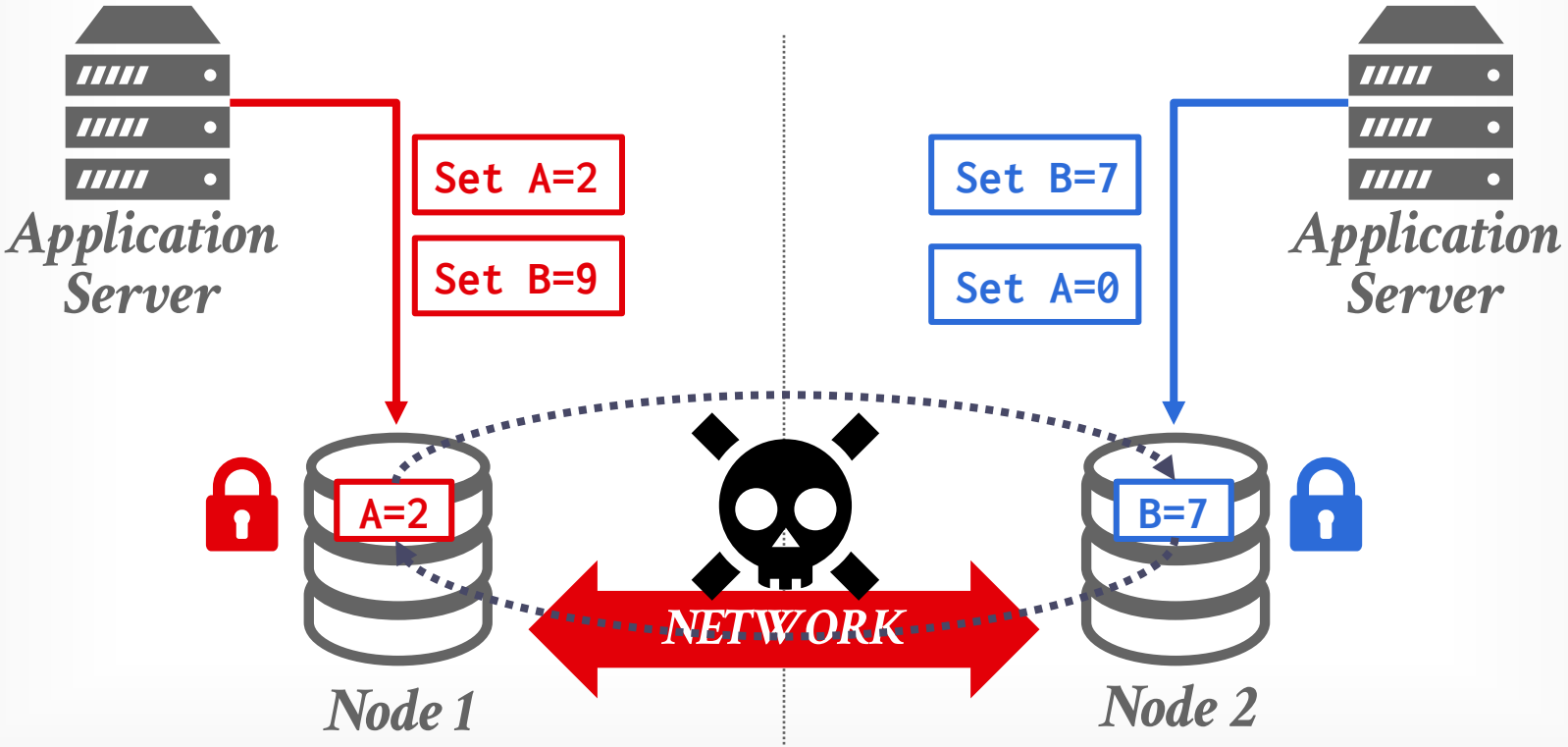
DISTRIBUTED 2PL



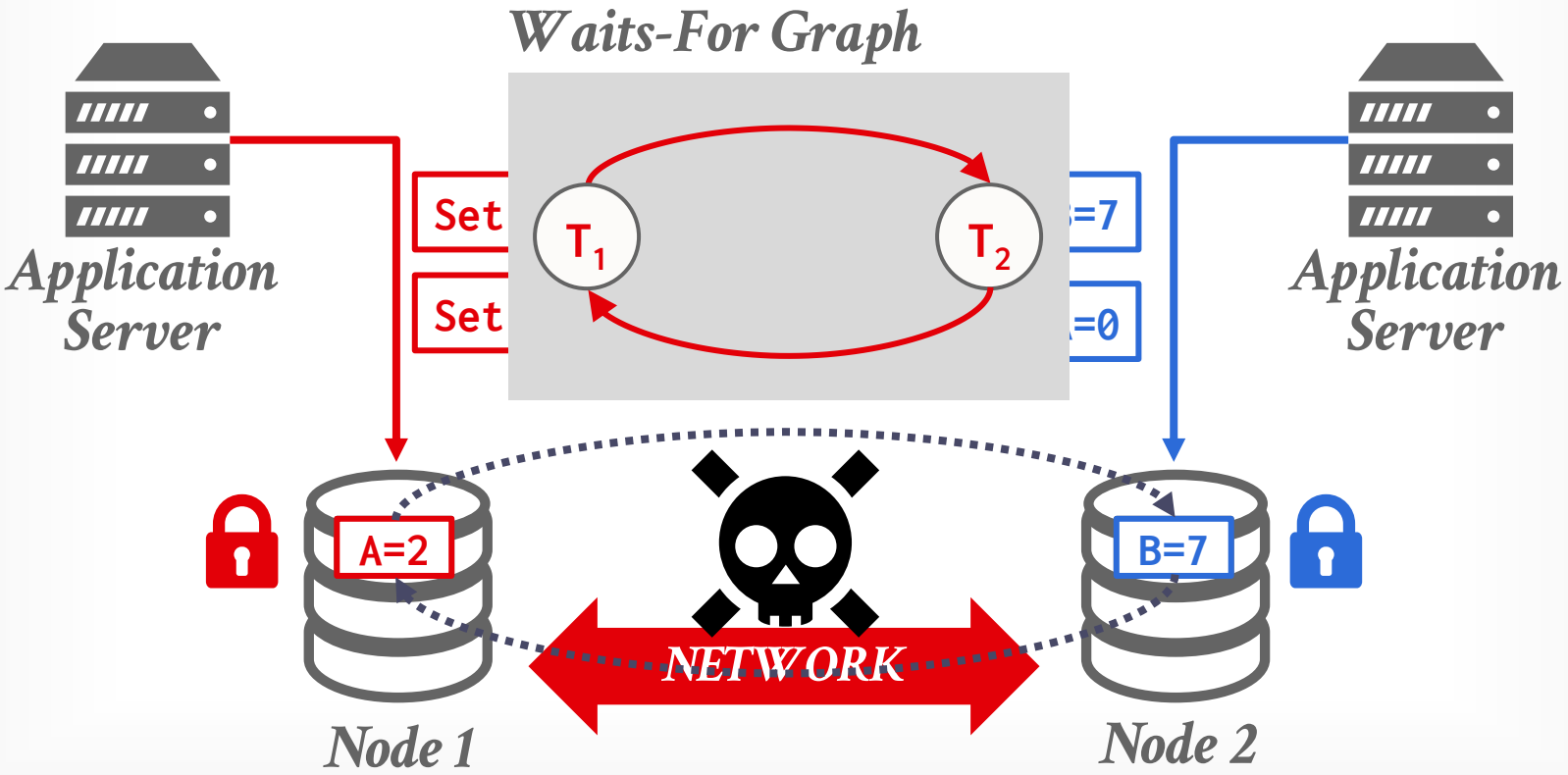
DISTRIBUTED 2PL



DISTRIBUTED 2PL



DISTRIBUTED 2PL



CONCLUSION

We have barely scratched the surface on distributed database systems…

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

NEXT CLASS

Distributed OLTP Systems

Distributed OLAP Systems