Carnegie Mellon University

Database Systems

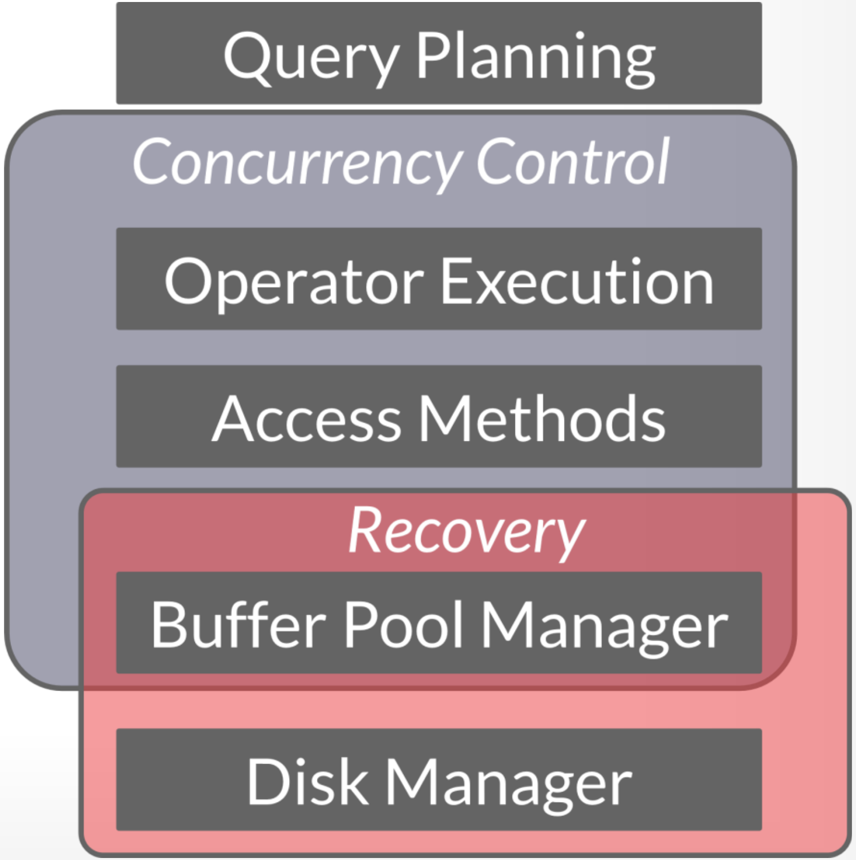
Distributed Databases

15-445/645 FALL 2024 » PROF. ~~A PA~~ WAN SHEN LIM

COURSE STATUS

Databases are hard.

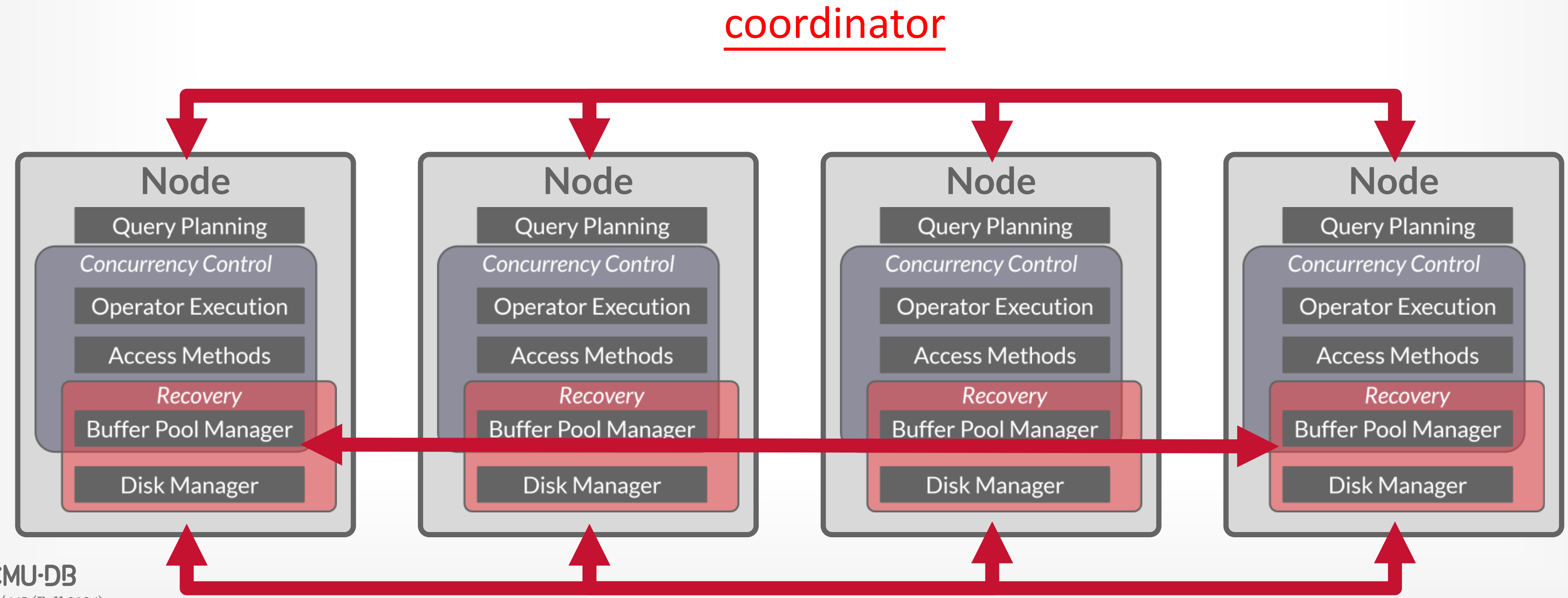
Distributed databases are harder.



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 需要考虑节点之间的数据迁移成本
→ Concurrency Control
→ Logging & Recovery

TODAY'S AGENDA

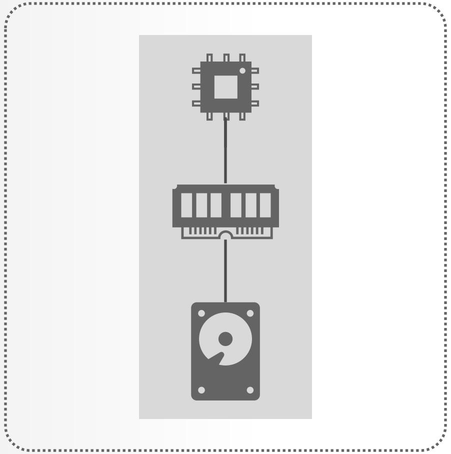
System Architectures
Design Issues
Partitioning Schemes
Distributed Concurrency Control
DB Flash Talk: DataStax

SYSTEM ARCHITECTURE

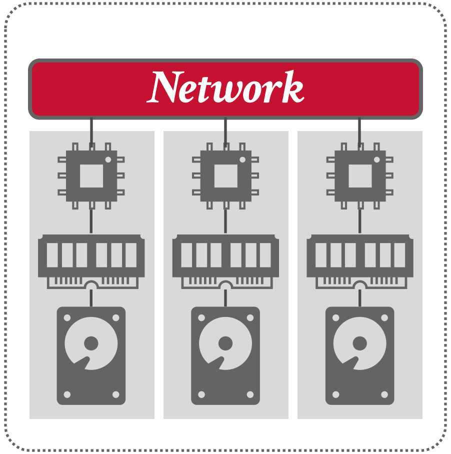
A distributed DBMS's system architecture specifies what shared resources are directly accessible to CPUs.

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

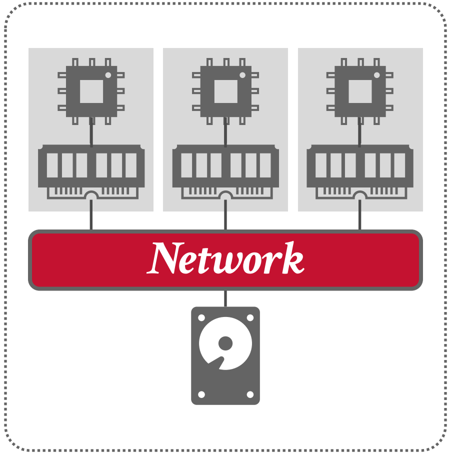
SYSTEM ARCHITECTURE



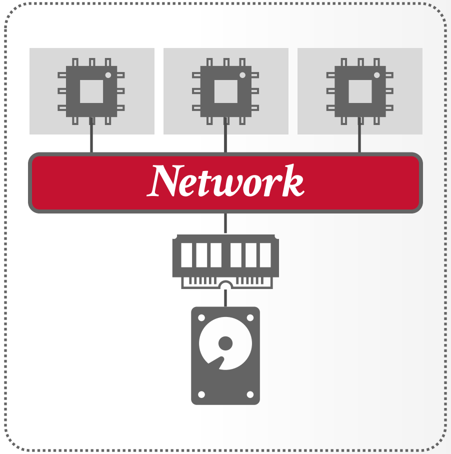
Shared Everything



Shared Nothing



Shared Disk



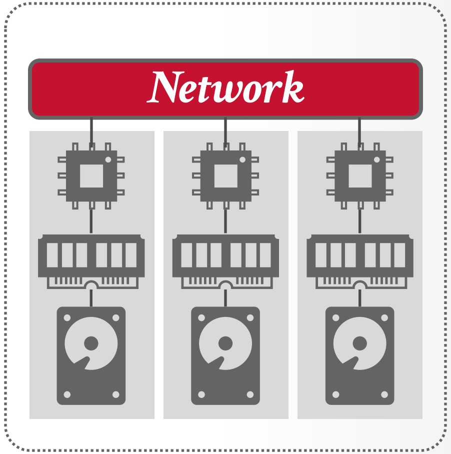
Shared Memory

SHARED NOTHING

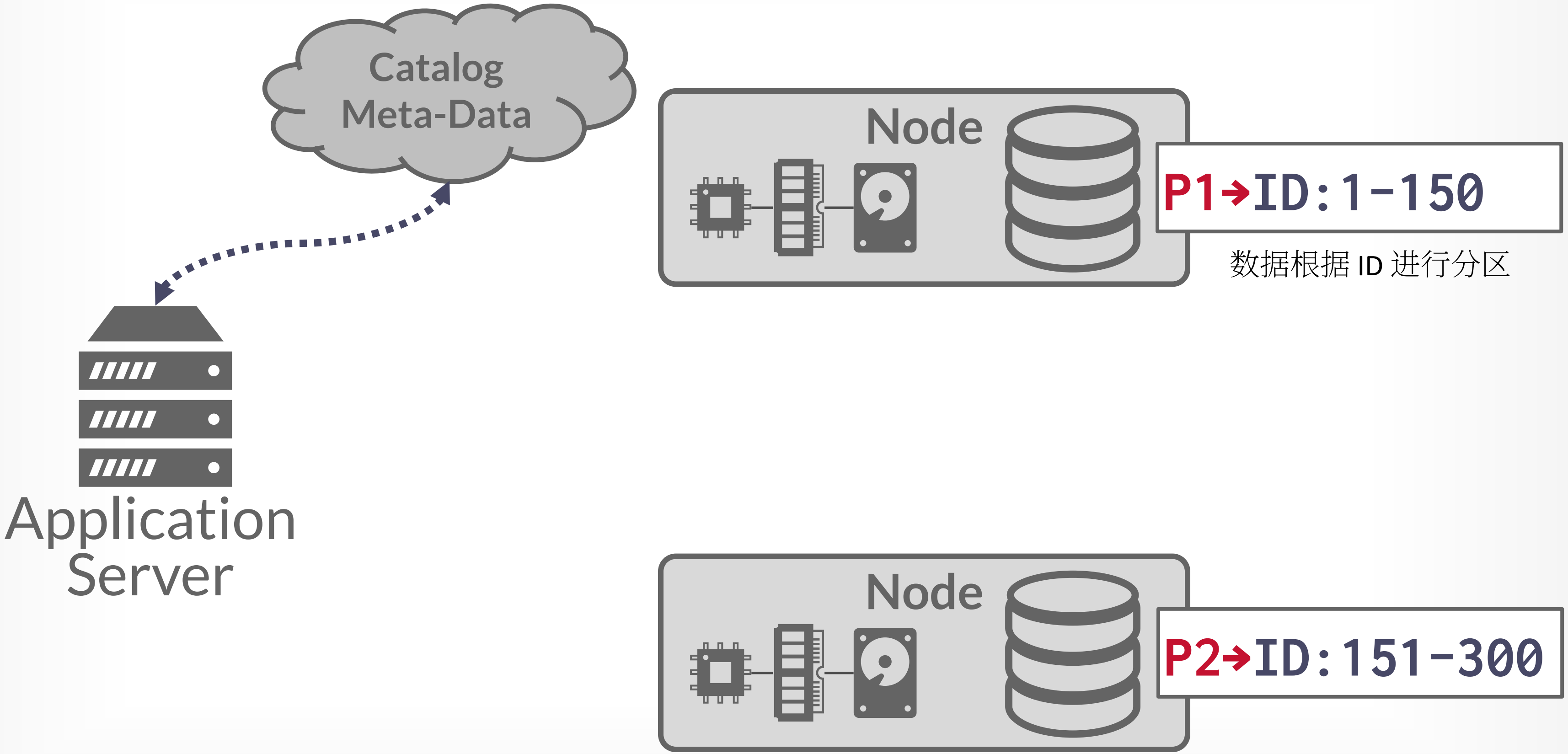
Each DBMS node has its own CPU, memory, and local disk.

Nodes only communicate with each other via network.

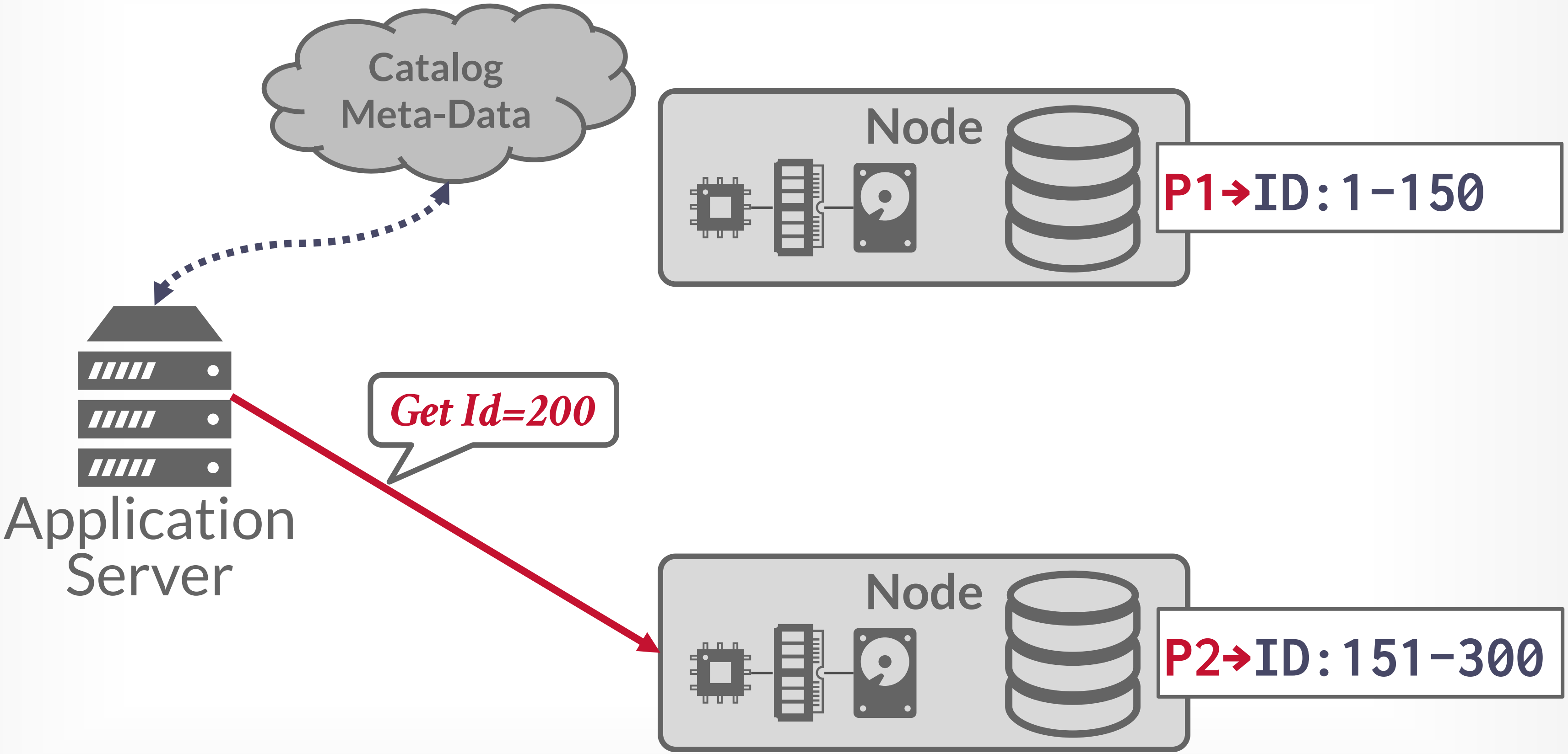
→ Better performance & efficiency.
→ Harder to scale capacity.
→ Harder to ensure consistency.



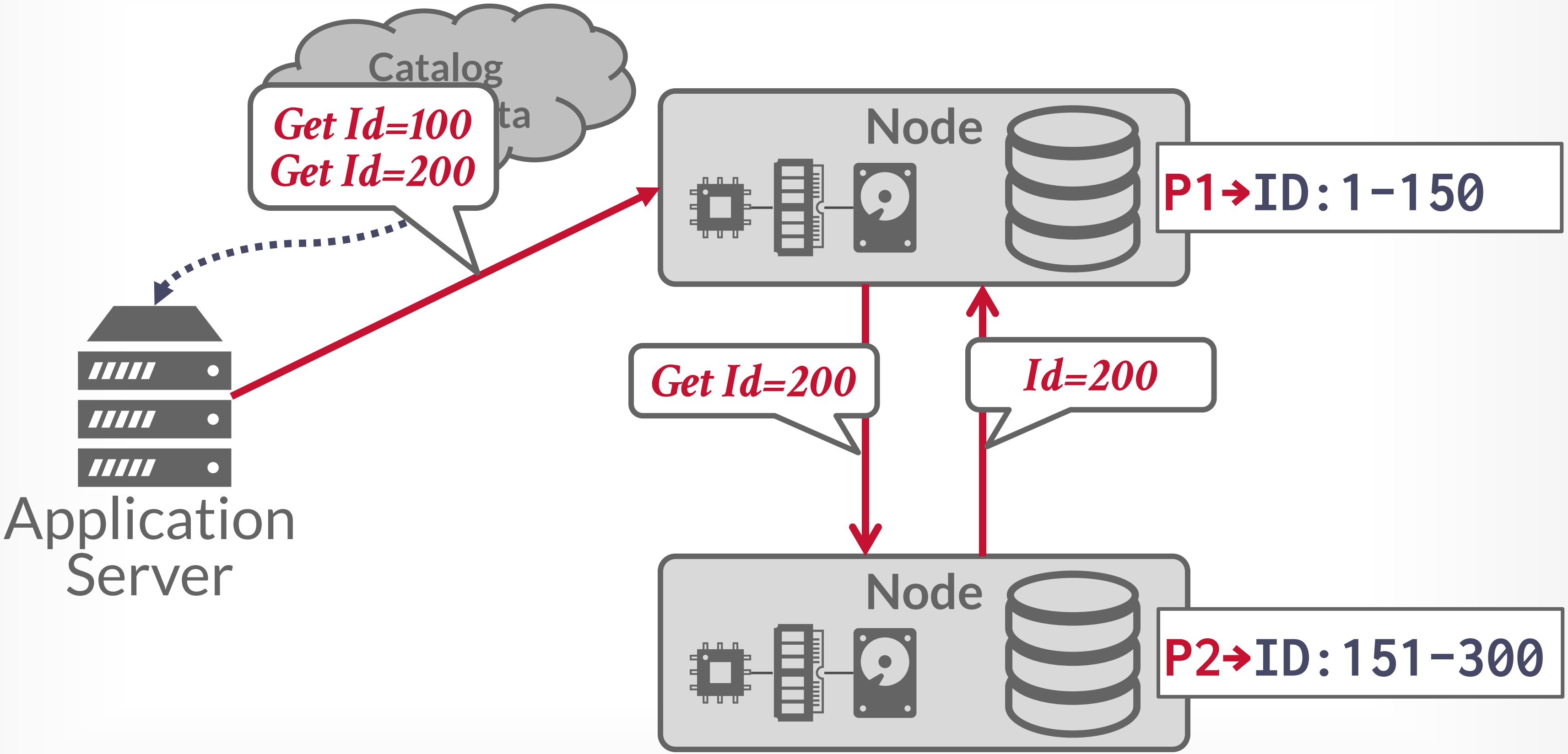
SHARED NOTHING EXAMPLE



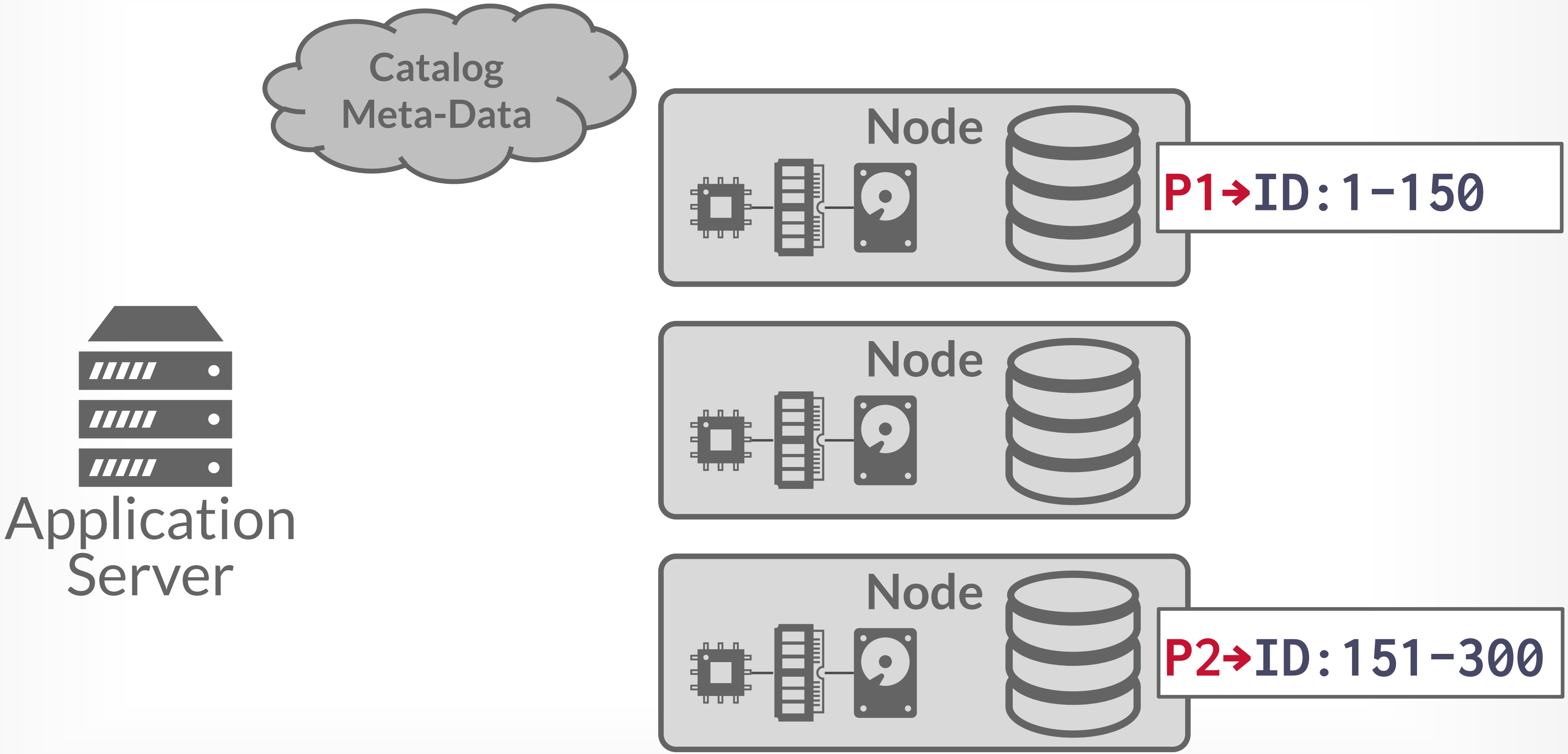
SHARED NOTHING EXAMPLE



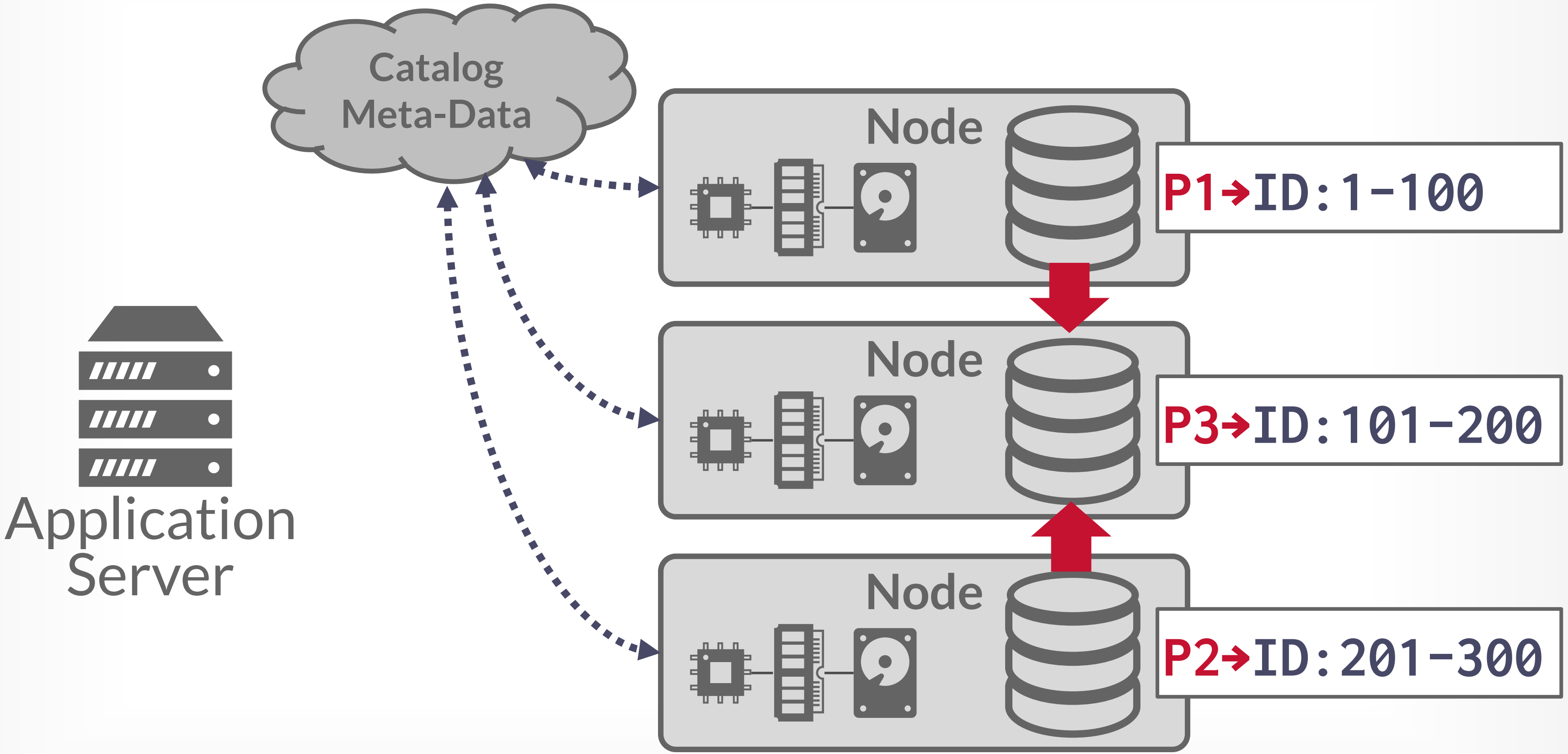
SHARED NOTHING EXAMPLE



SHARED NOTHING EXAMPLE



SHARED NOTHING EXAMPLE



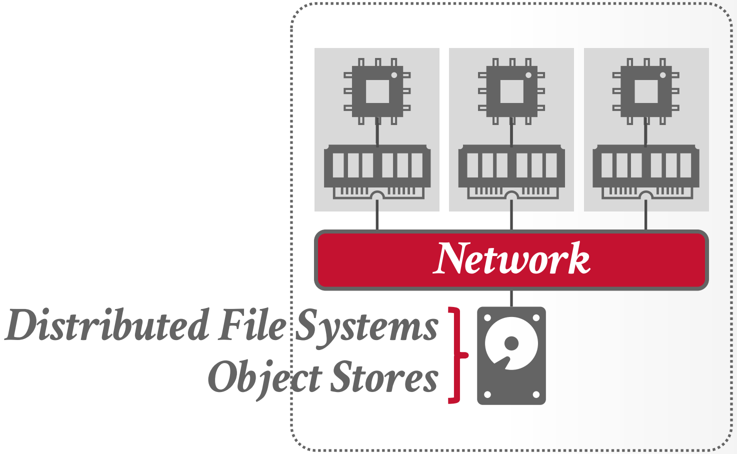
SHARED DISK

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

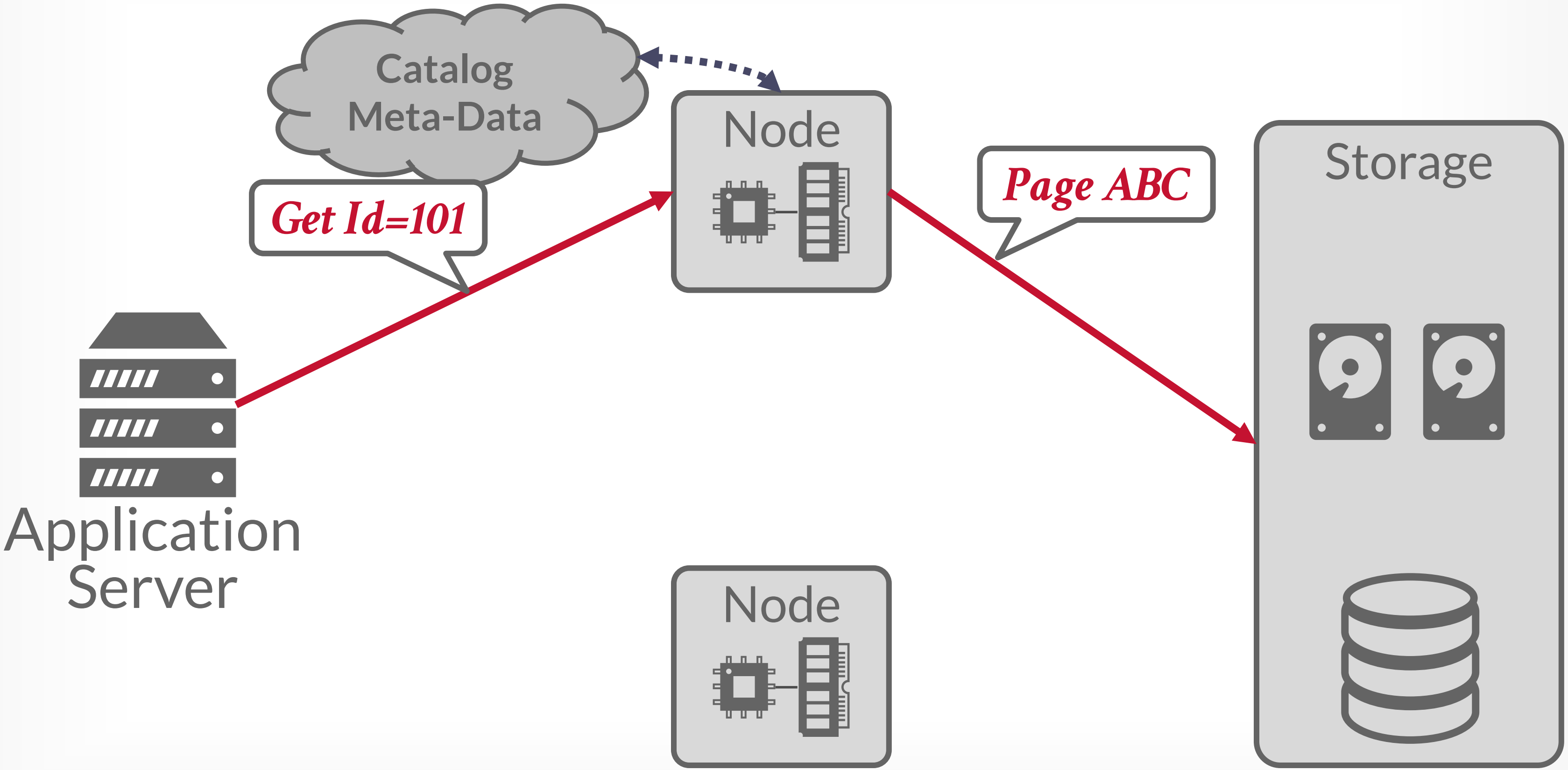
→ Scale execution layer independently from the storage layer.

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

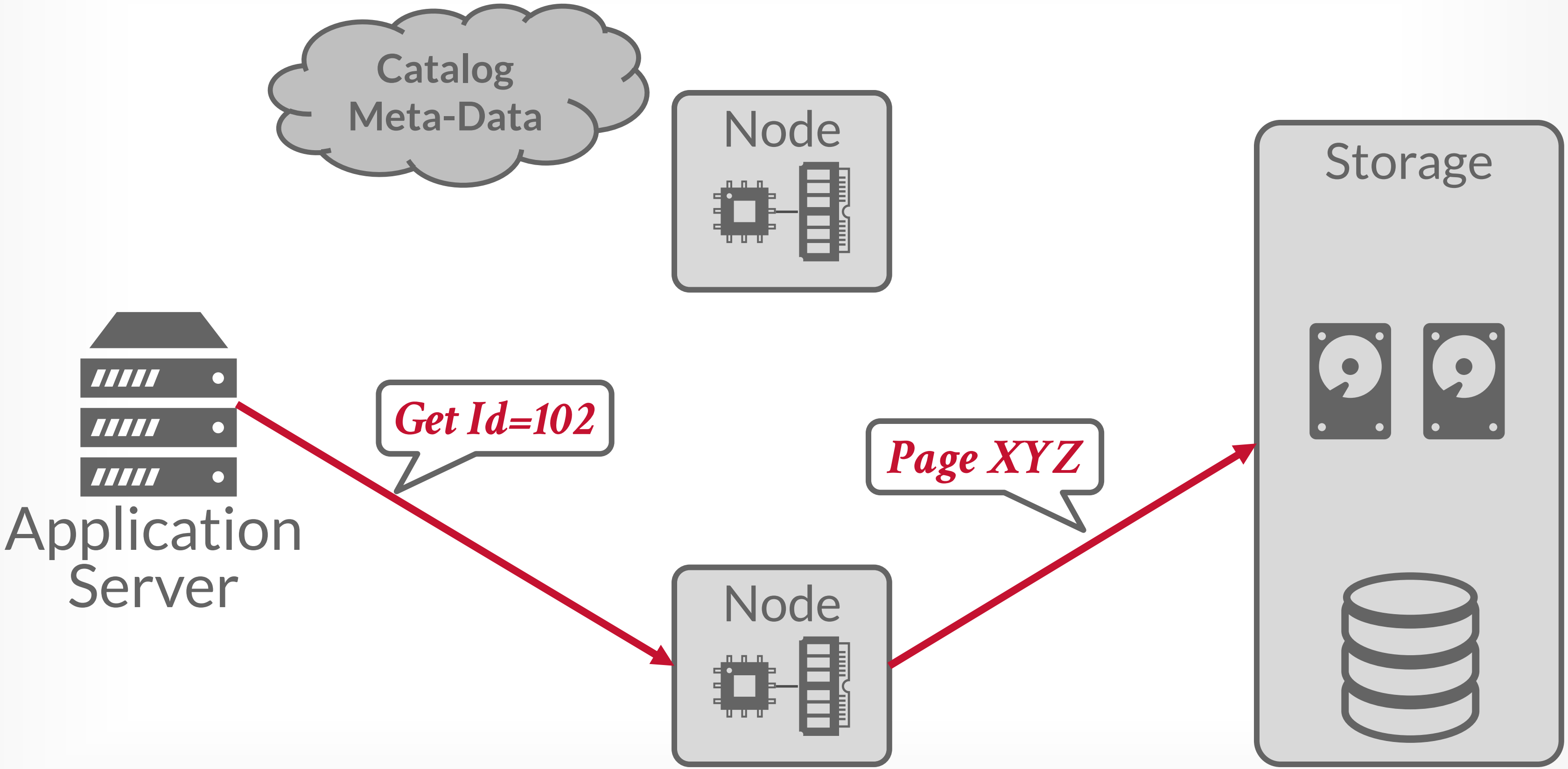
→ This architecture facilitates **data lakes** and **serverless** systems.



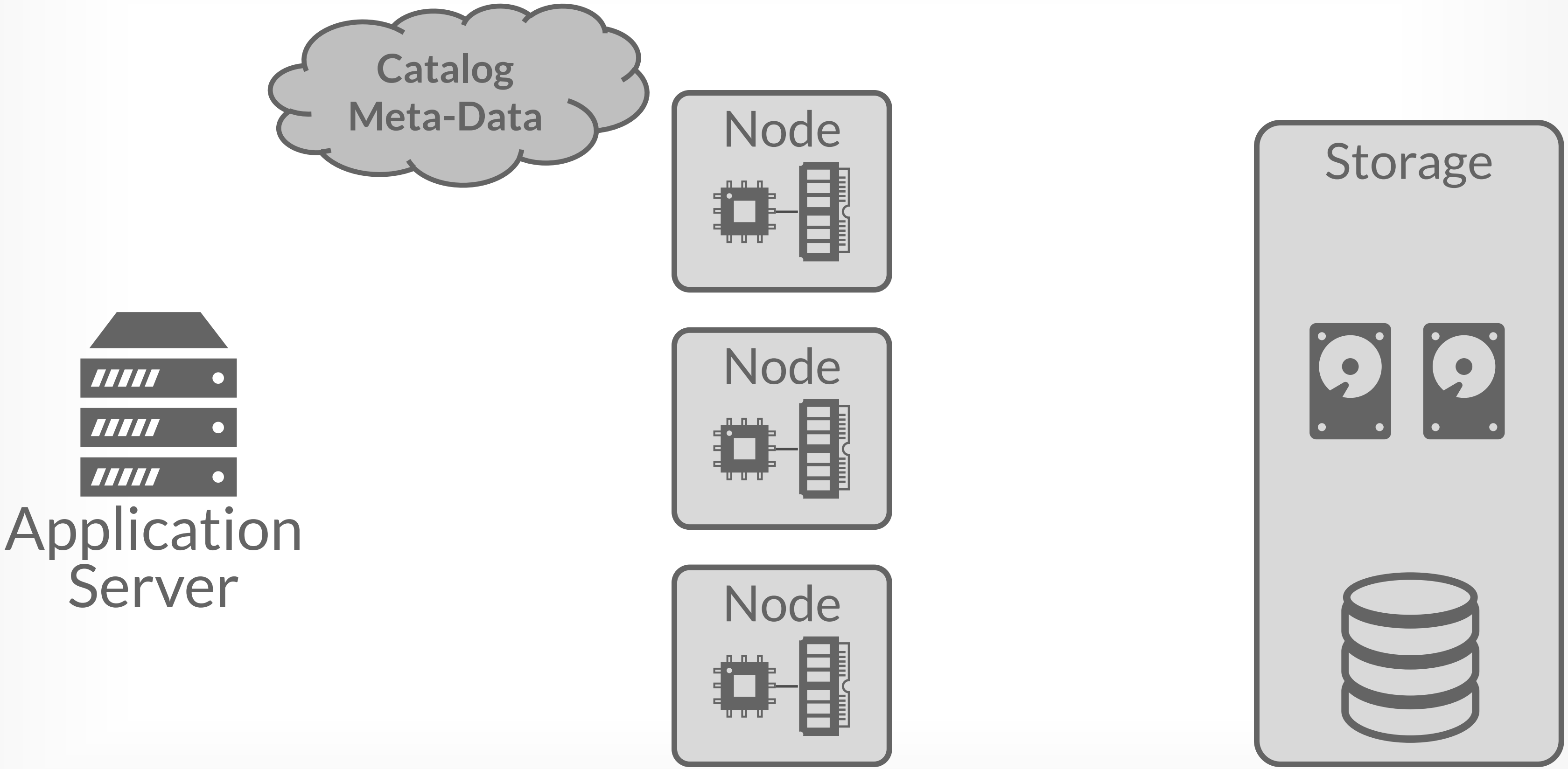
SHARED DISK EXAMPLE



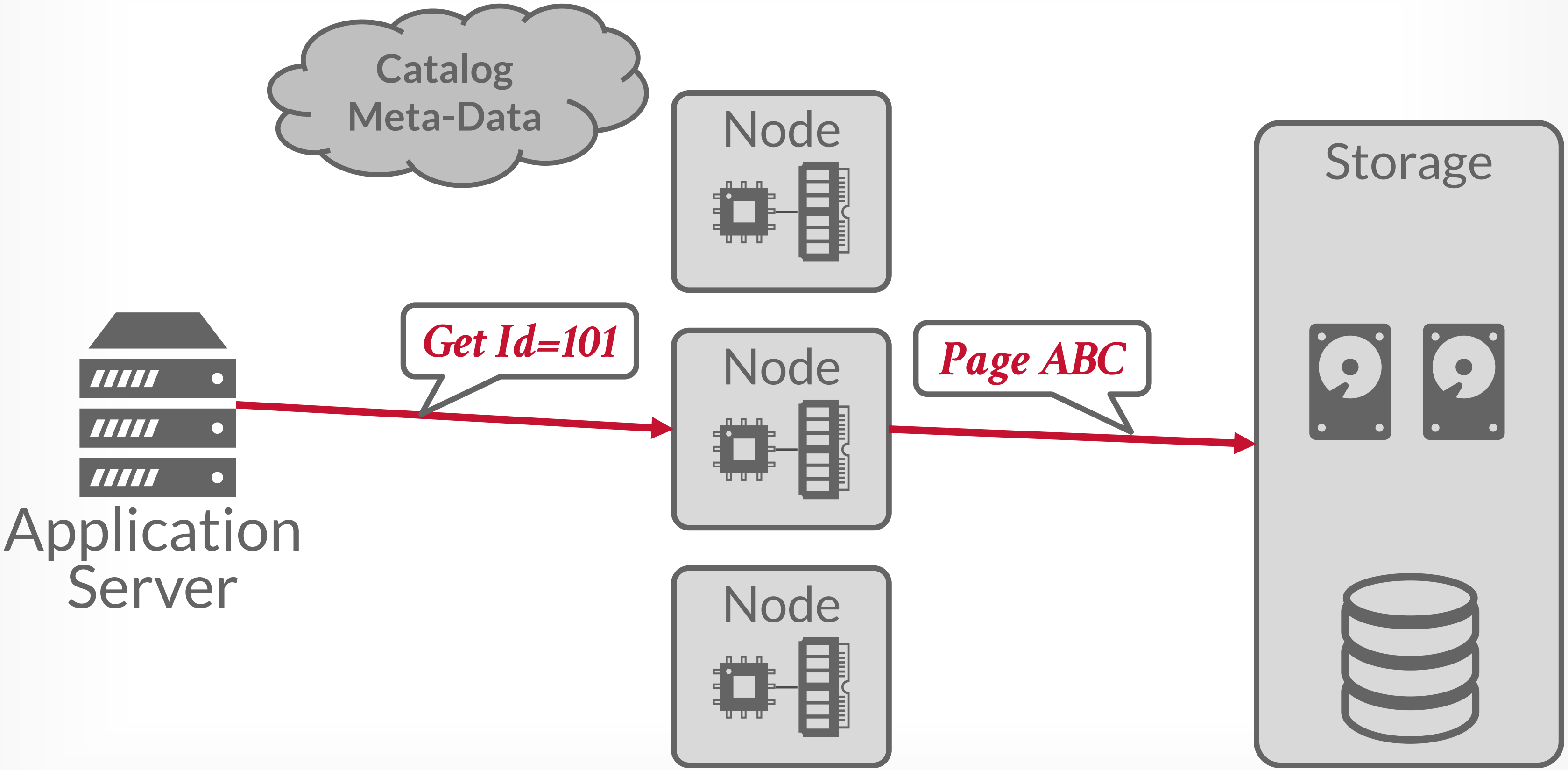
SHARED DISK EXAMPLE



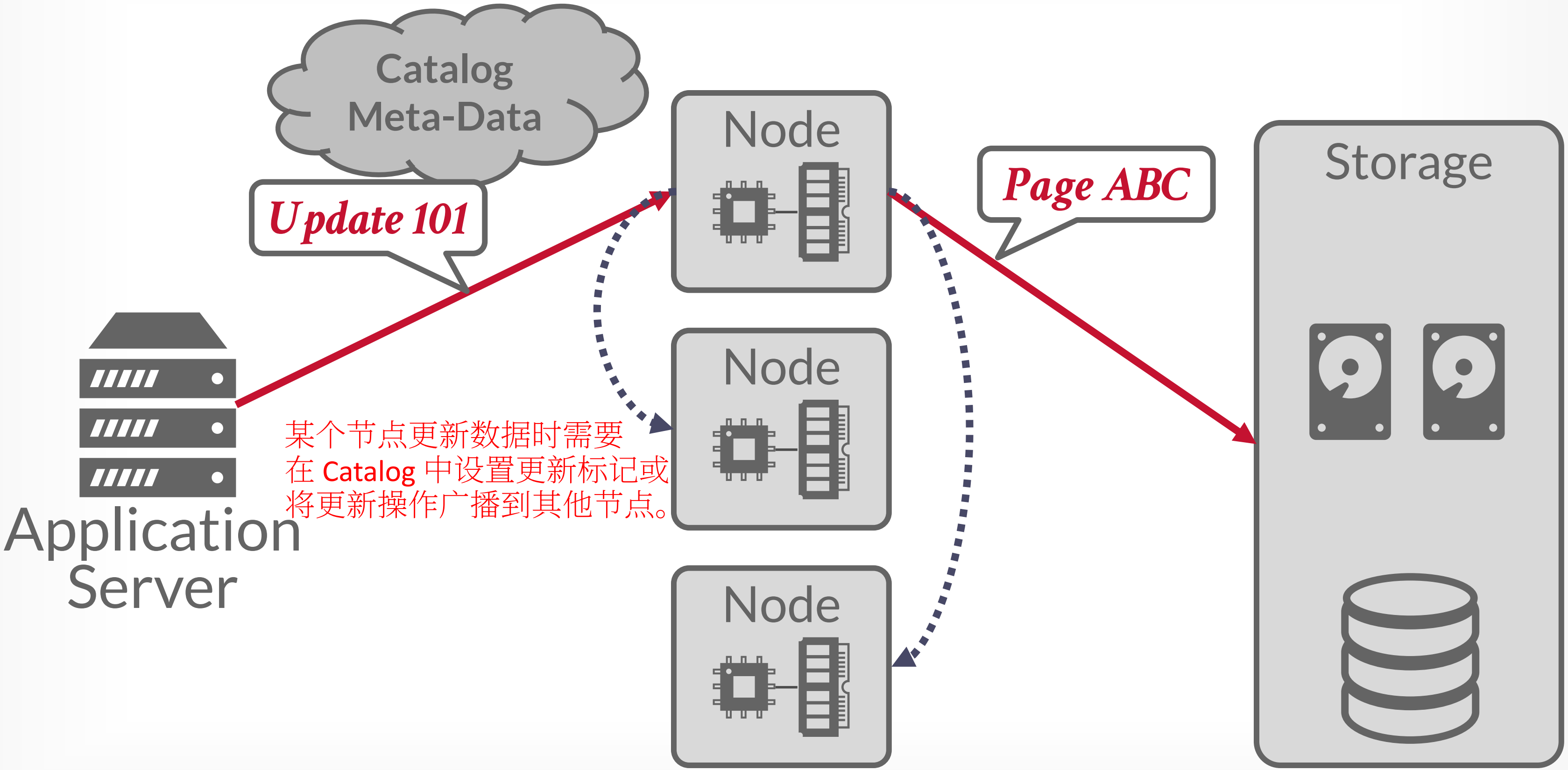
SHARED DISK EXAMPLE



SHARED DISK EXAMPLE

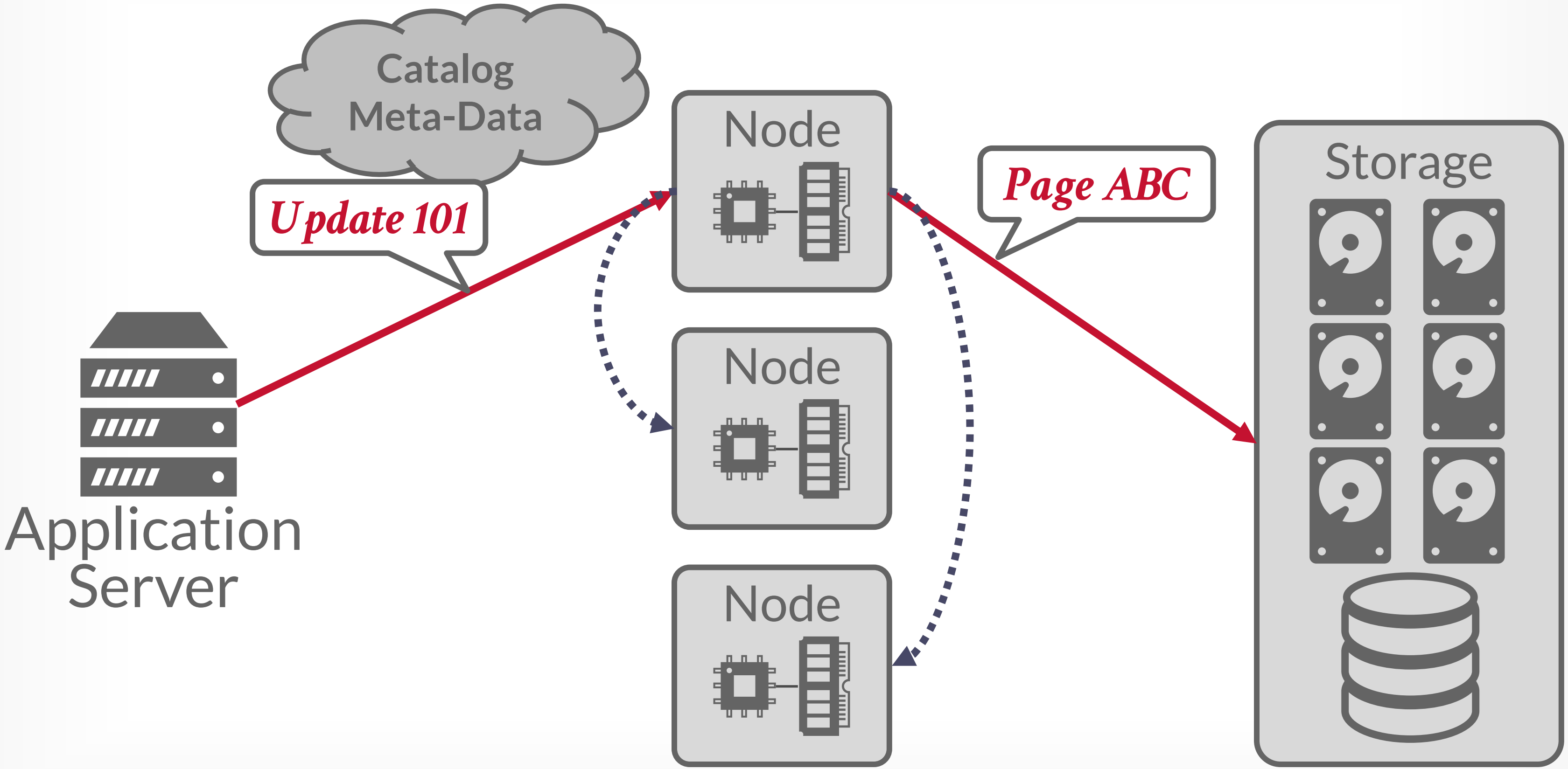


SHARED DISK EXAMPLE



某个节点更新数据时需要
在 Catalog 中设置更新标记或
将更新操作广播到其他节点。

SHARED DISK EXAMPLE



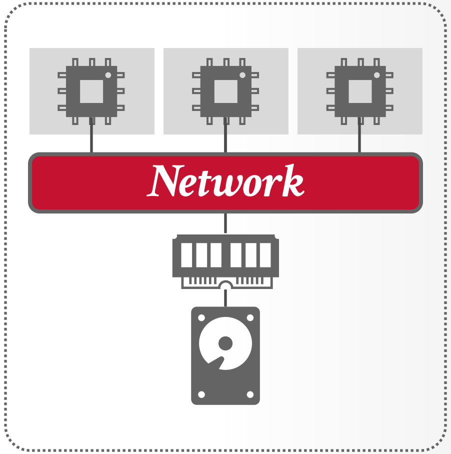
SHARED MEMORY

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.



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?

$\boxed{\text{How does the DBMS ensure correctness?}}$ ***Next Class***

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.
→ Called "sharding" in NoSQL systems.

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

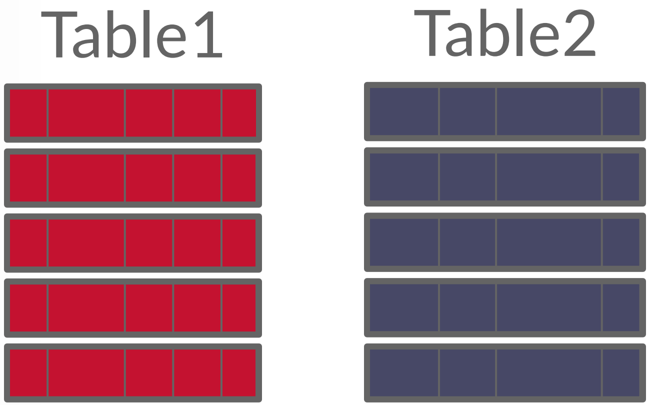
NAÏVE TABLE PARTITIONING

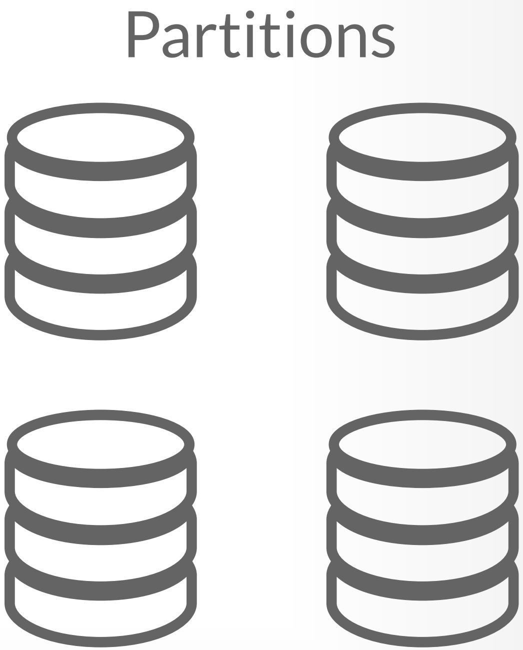
Assign an entire table to a single node.

Assumes that each node has enough storage space for an entire table.

Ideal if queries never join data across tables stored on different nodes and access patterns are uniform.

NAÏVE TABLE PARTITIONING

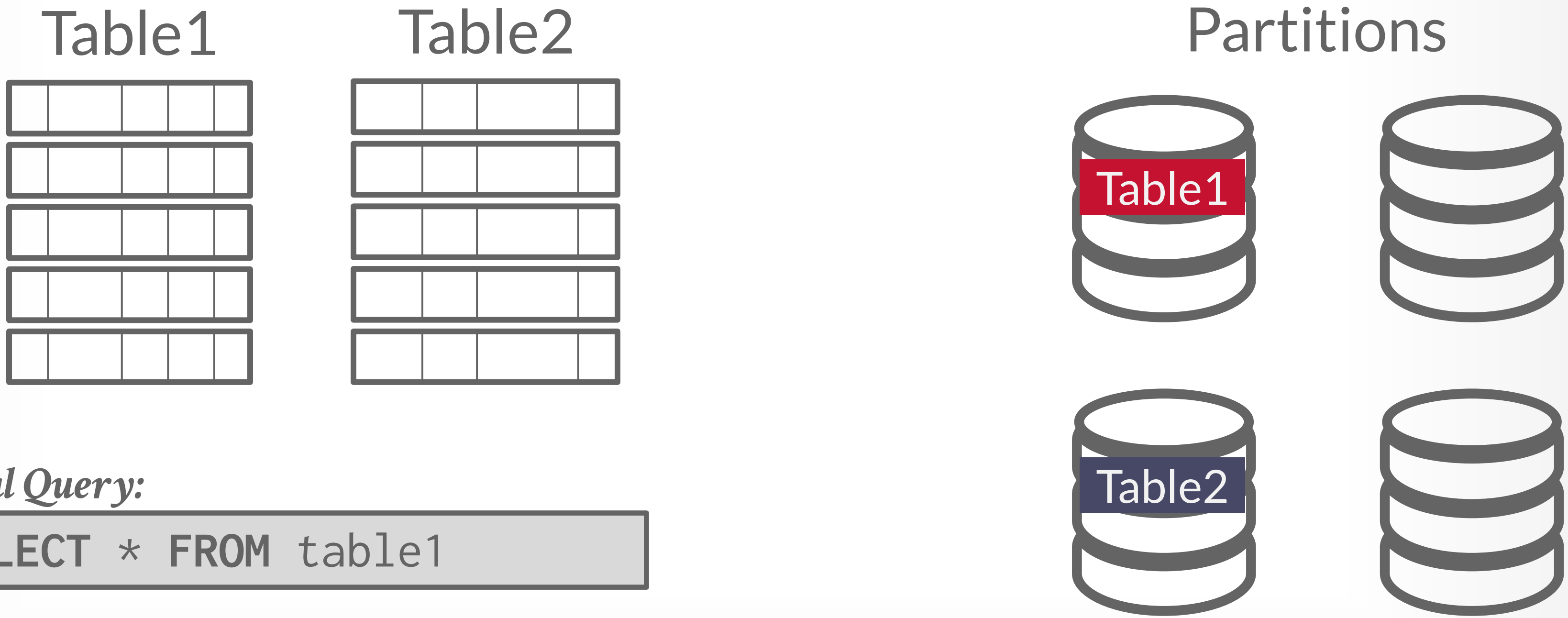




Ideal Query:

```
SELECT * FROM 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
);
```

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 where 子句进行手动分区

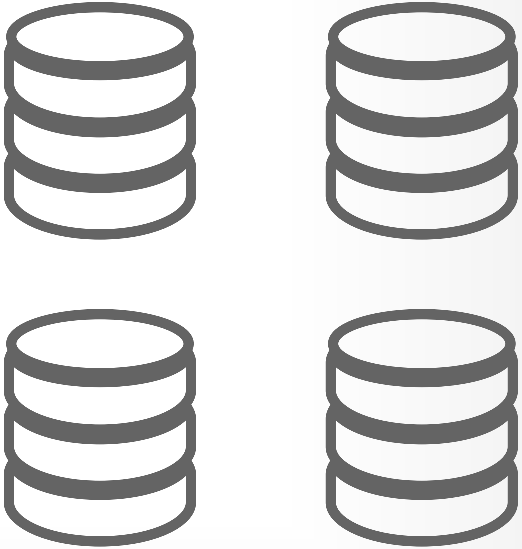
HORIZONTAL PARTITIONING

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 = ?
```

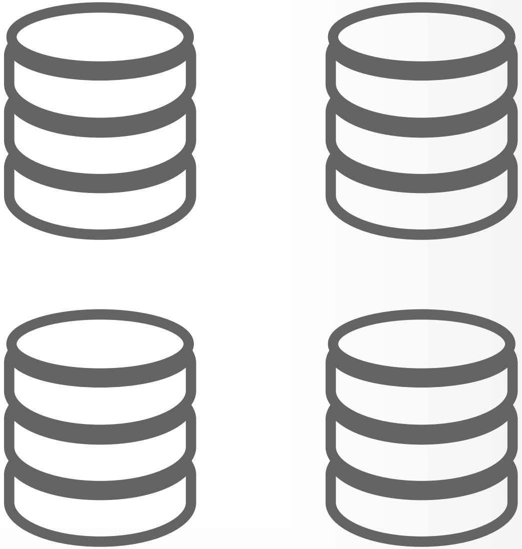
HORIZONTAL PARTITIONING

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 = ?
```

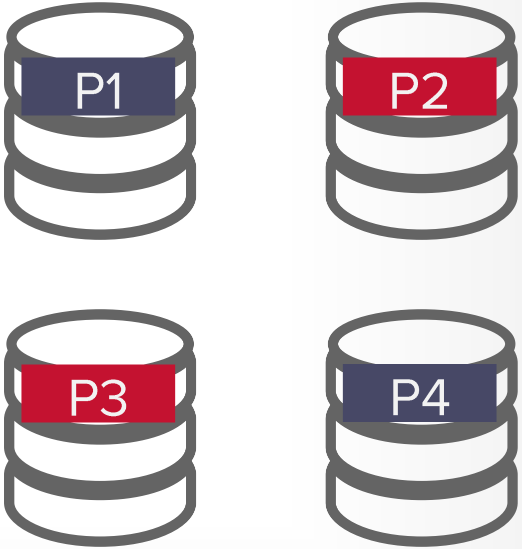
HORIZONTAL PARTITIONING

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

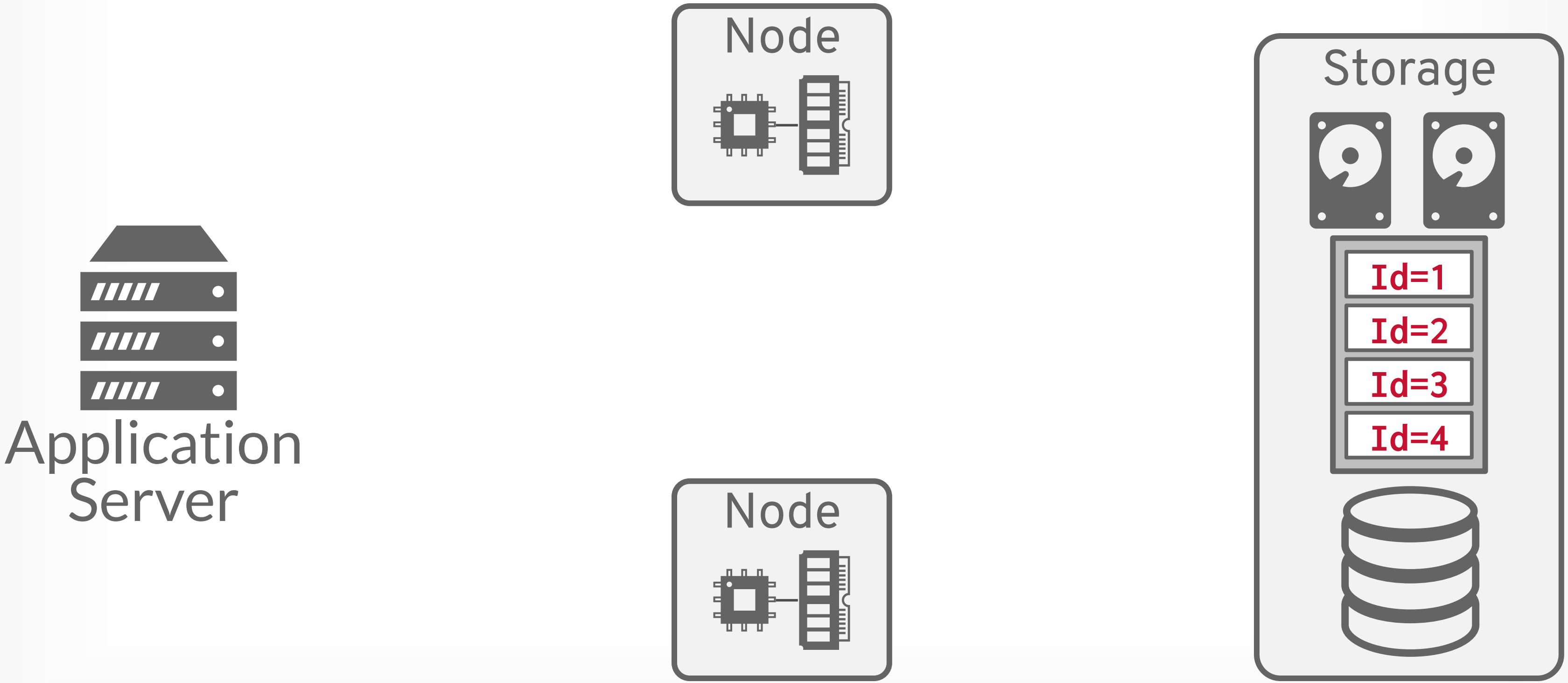


Ideal Query:

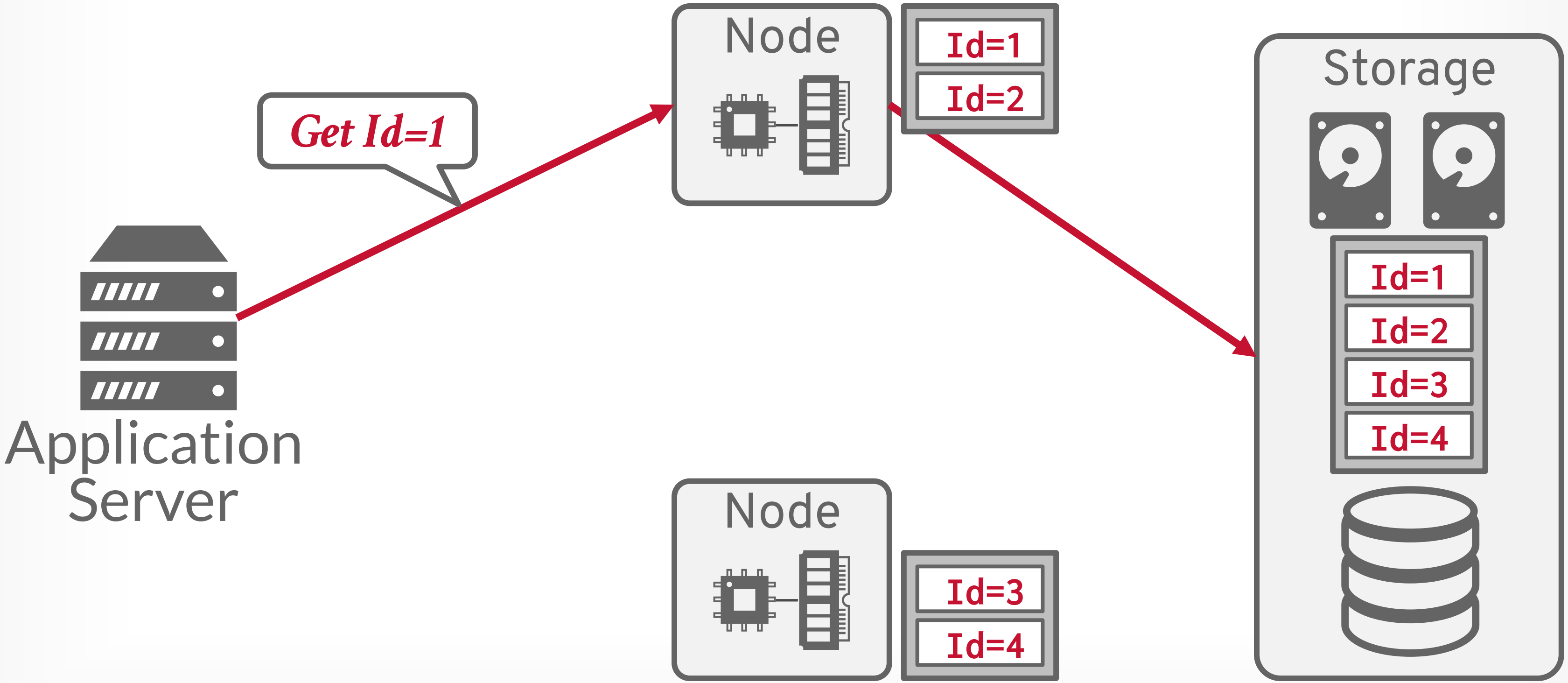
```
SELECT * FROM table
 WHERE partitionKey = ?
```

选择合适的分区键非常重要，
便于均衡地访问某个分区数据。

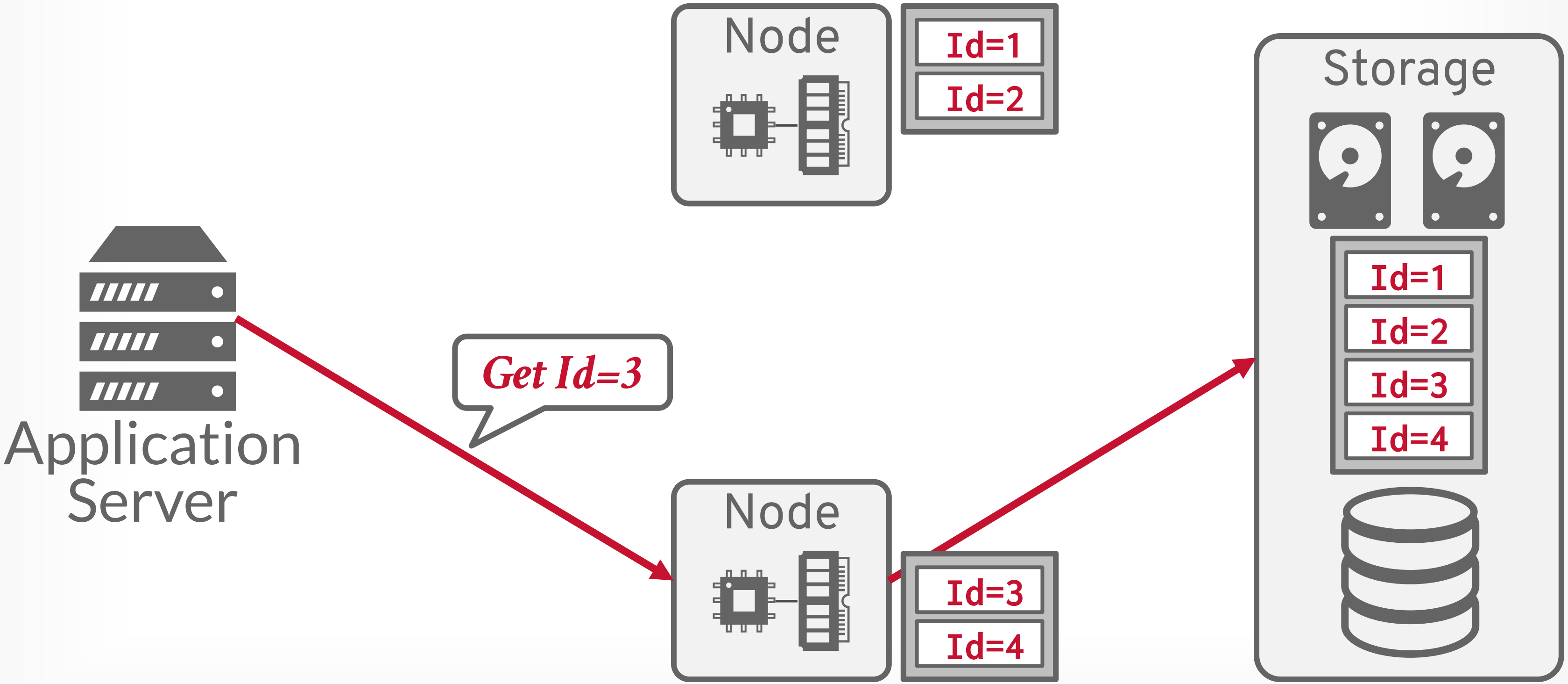
SHARED-DISK PARTITIONING



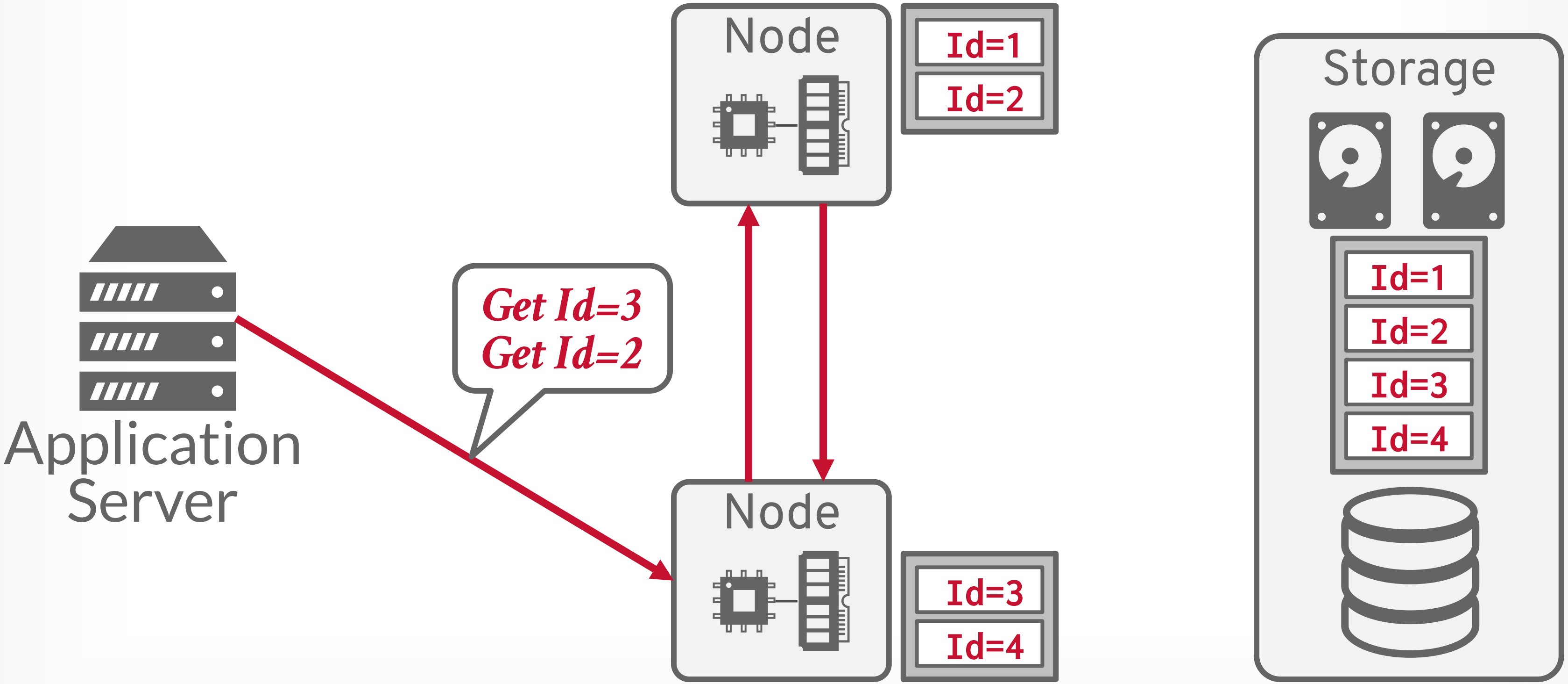
SHARED-DISK PARTITIONING



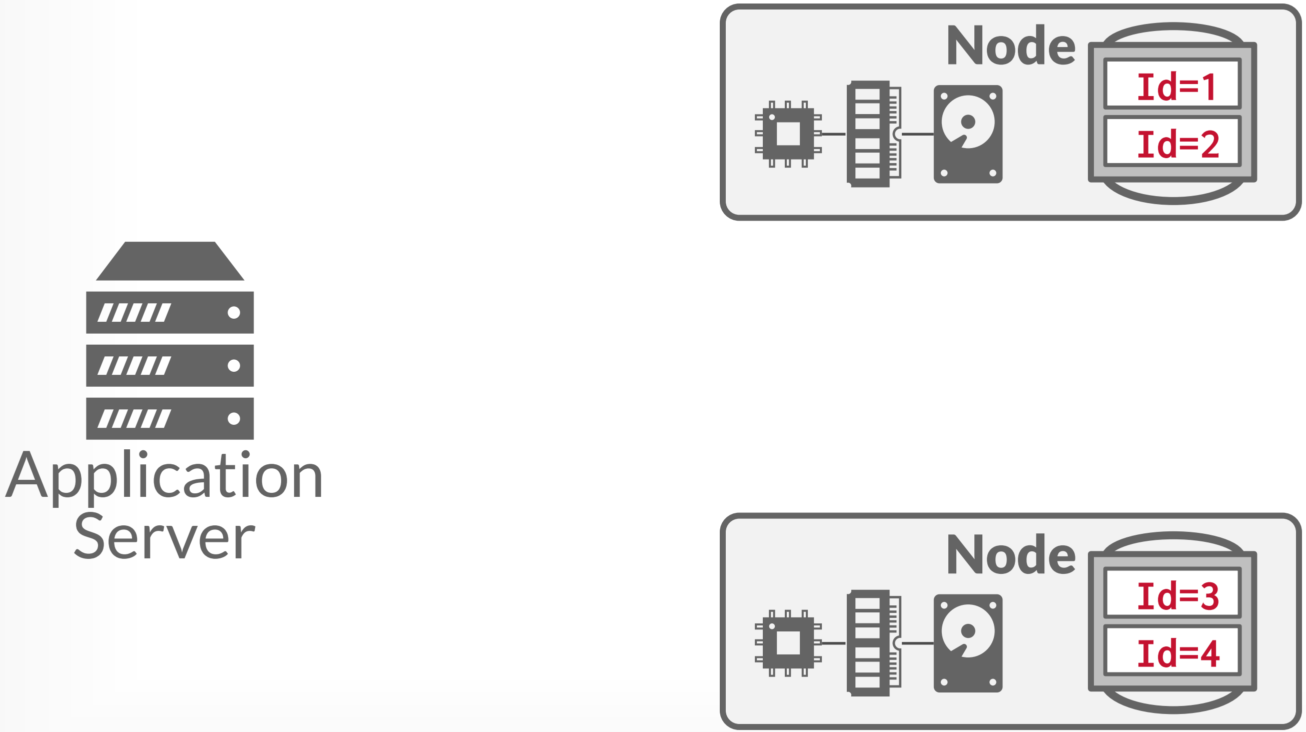
SHARED-DISK PARTITIONING



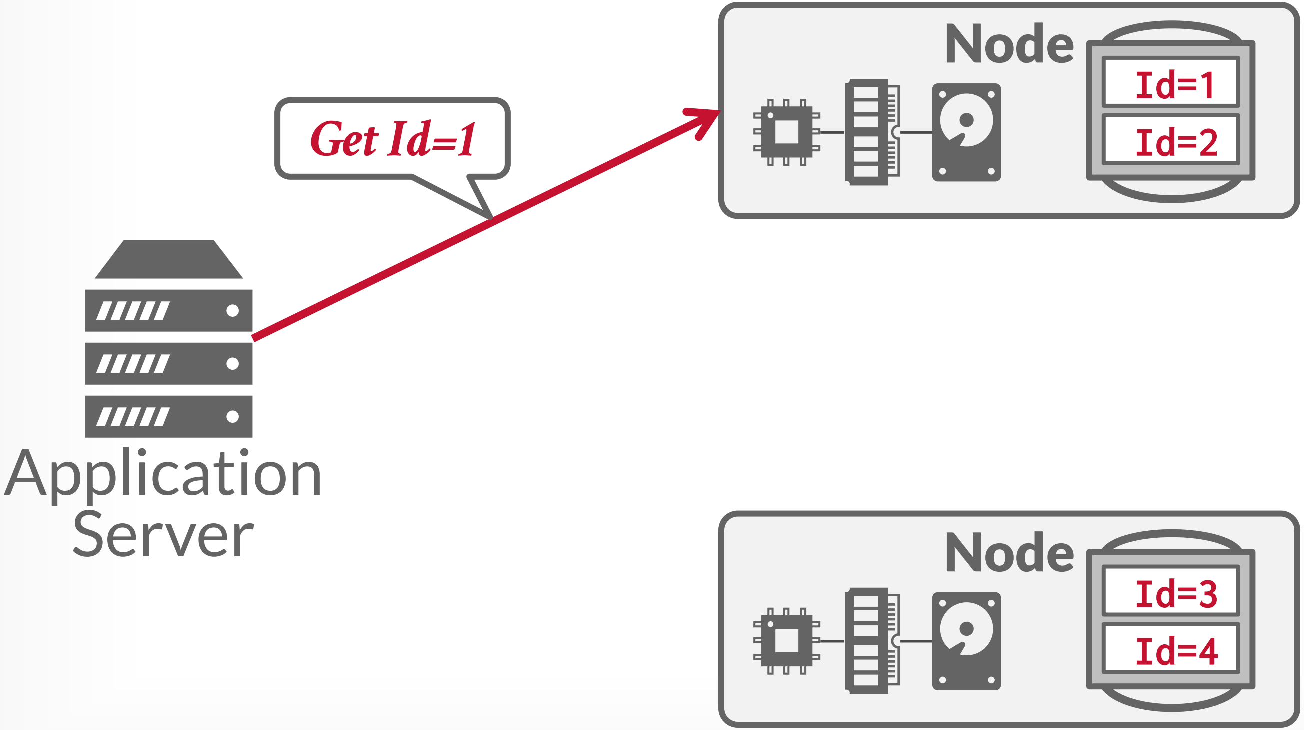
SHARED-DISK PARTITIONING



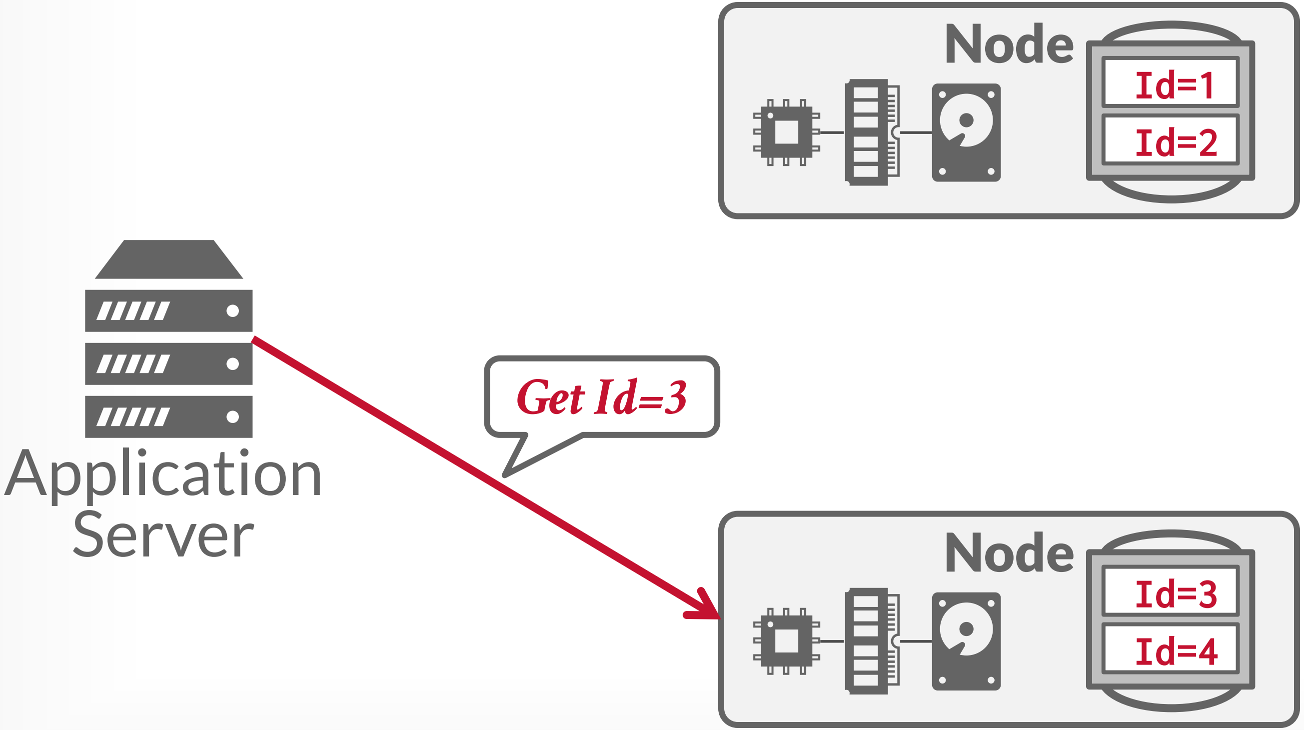
SHARED-NOTHING PARTITIONING



SHARED-NOTHING PARTITIONING



SHARED-NOTHING PARTITIONING



HORIZONTAL PARTITIONING

Partitioning Key

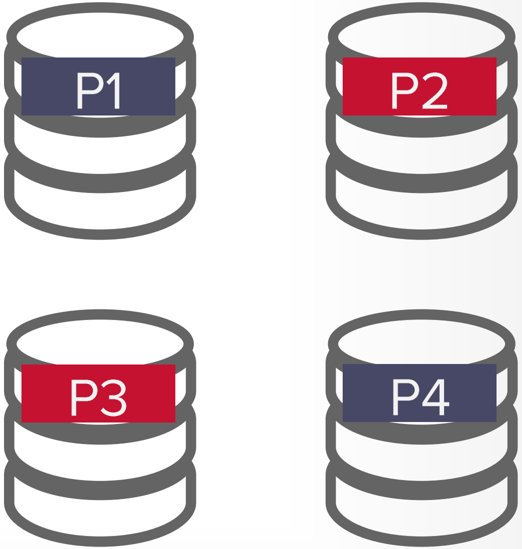
Table

101	a	XXX	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*

Ideal Query:

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

Partitions



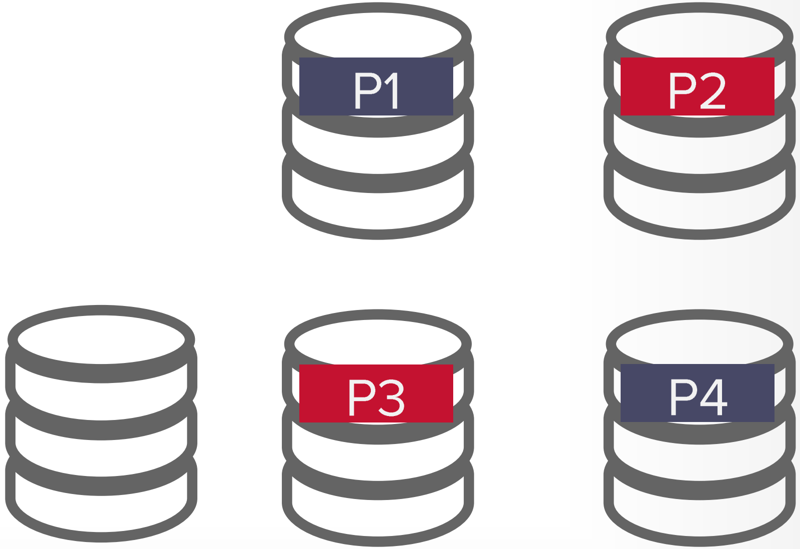
HORIZONTAL PARTITIONING

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
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)%5 = P4

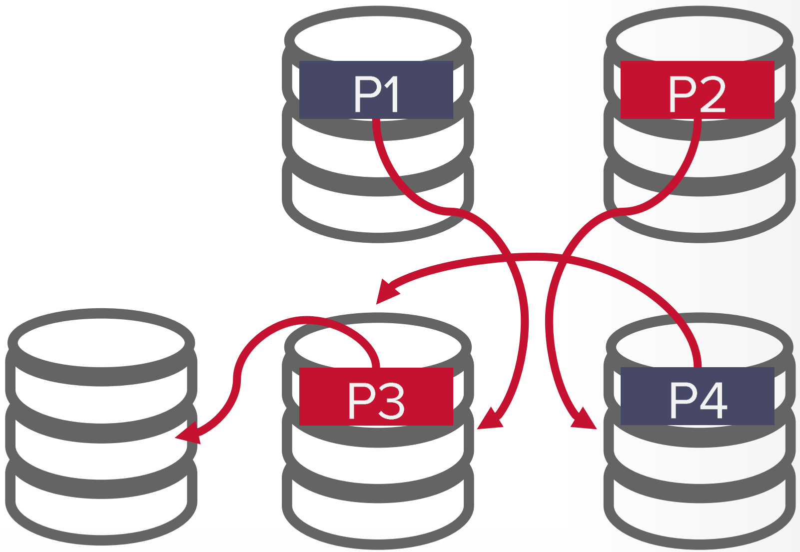
hash(b)%5 = P3

hash(c)%5 = P5

hash(d)%5 = P1

hash(e)%5 = P3

Partitions



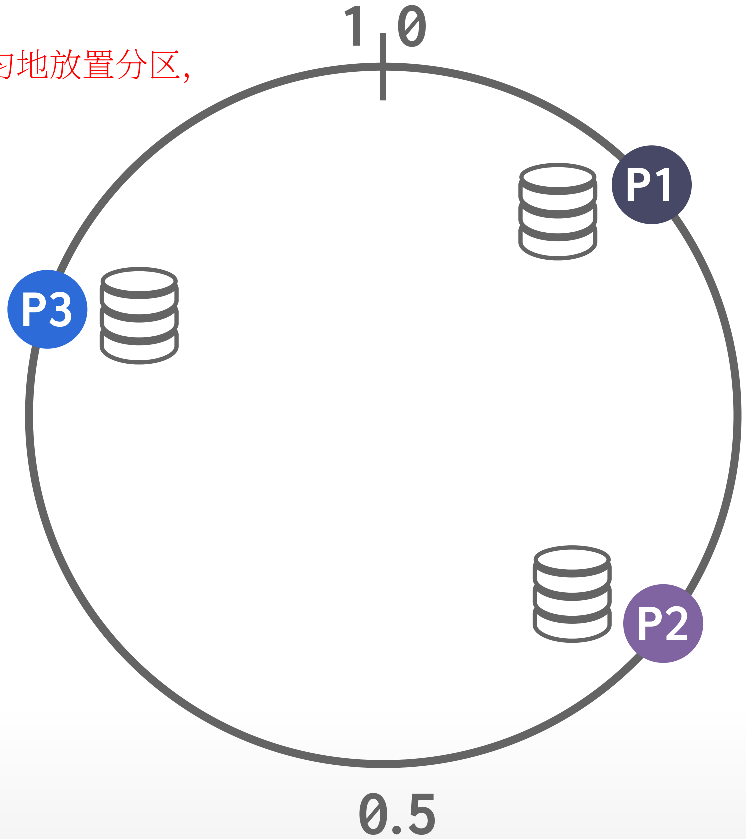
增加节点后数据需要重新分区

Ideal Query:

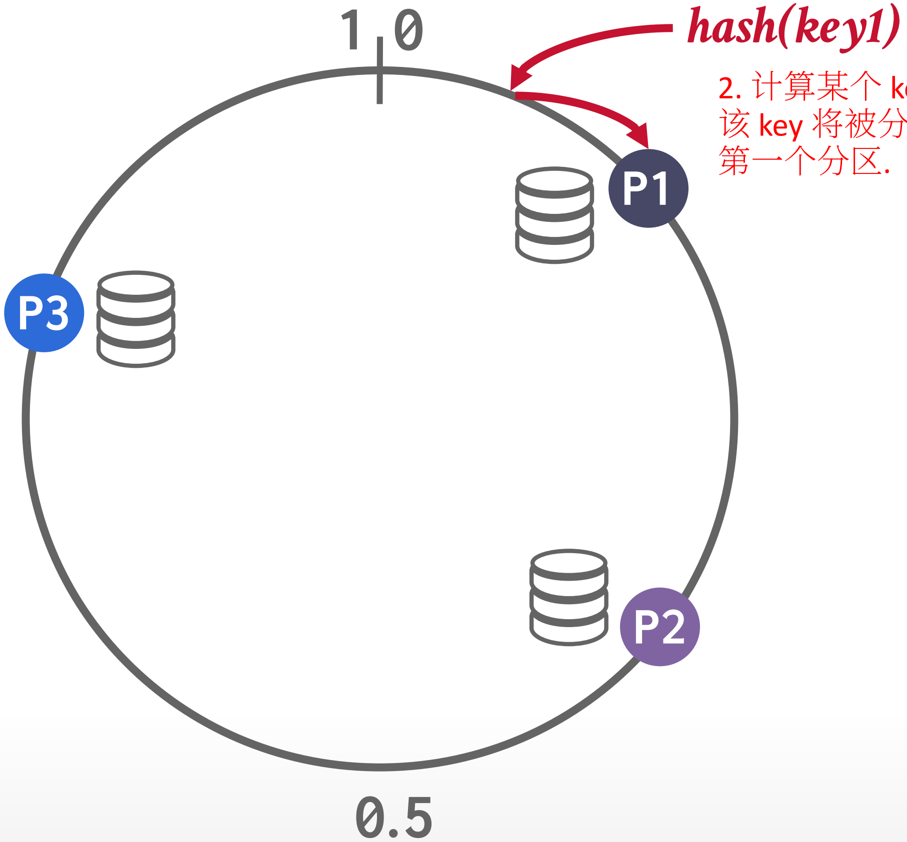
```
SELECT * FROM table
 WHERE partitionKey = ?
```

CONSISTENT HASHING

1. 单位圆上大致均匀地放置分区，
例如 P1, P2 和 P3；

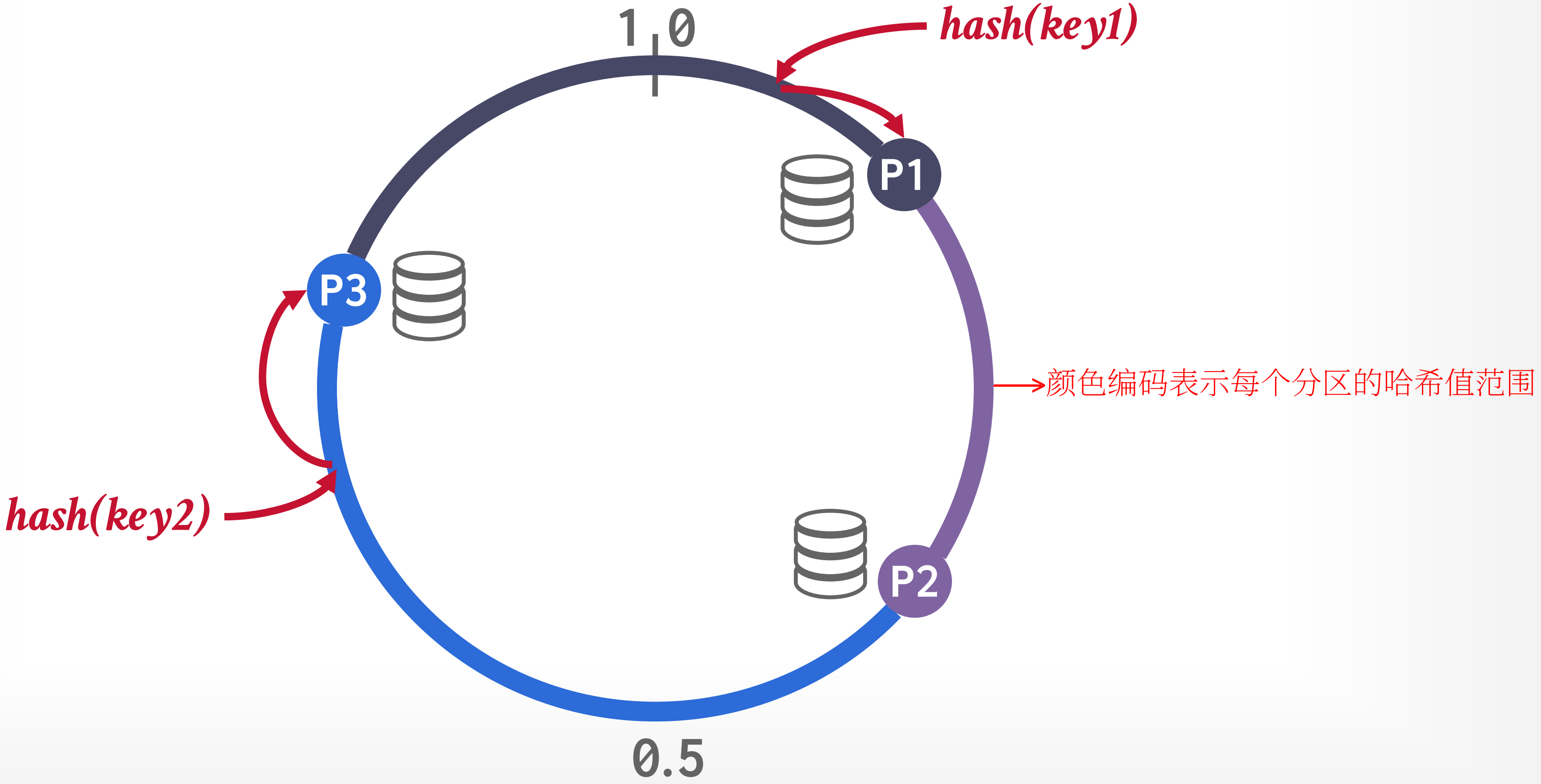


CONSISTENT HASHING

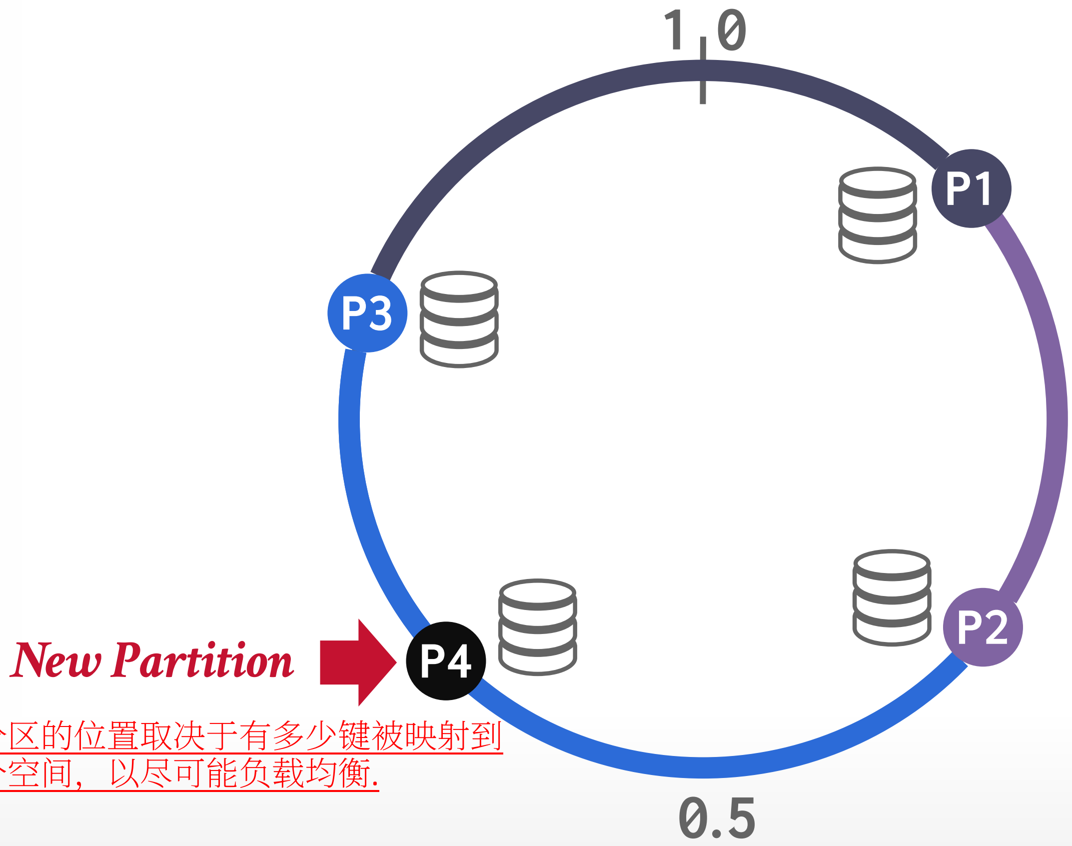


2. 计算某个 key 的哈希值确定分区：
该 key 将被分配到沿顺时针方向到达的第一个分区.

CONSISTENT HASHING

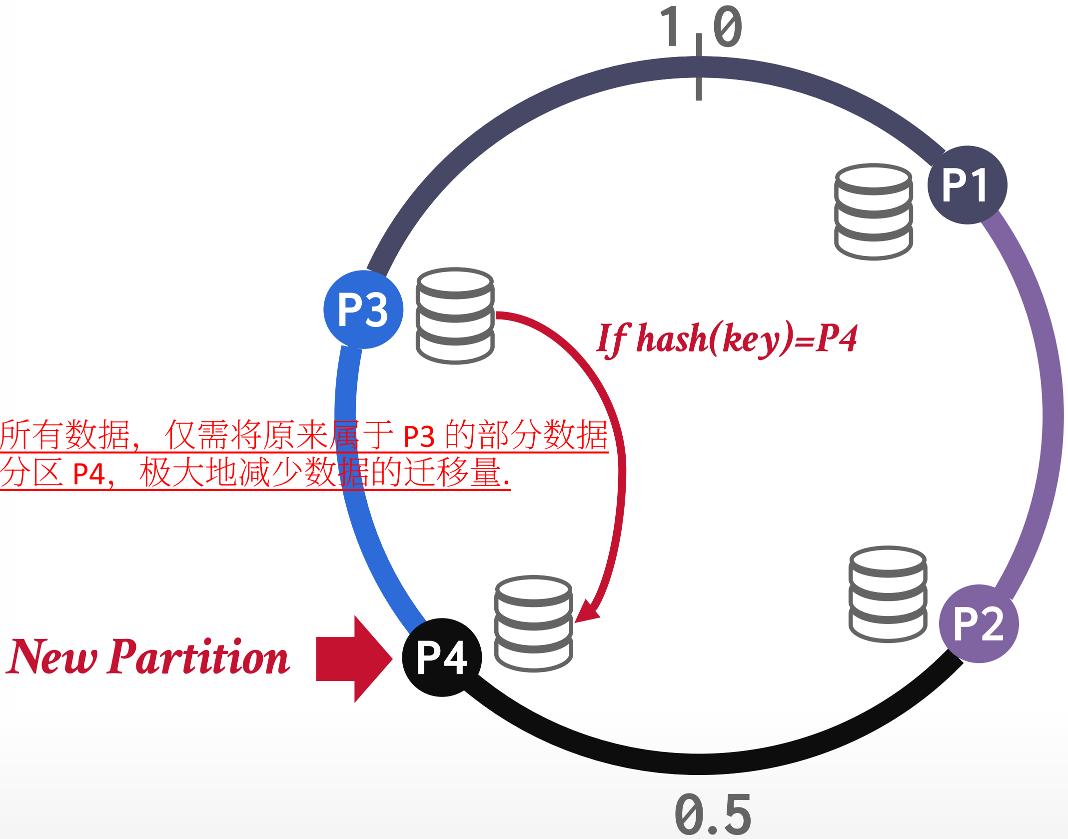


CONSISTENT HASHING



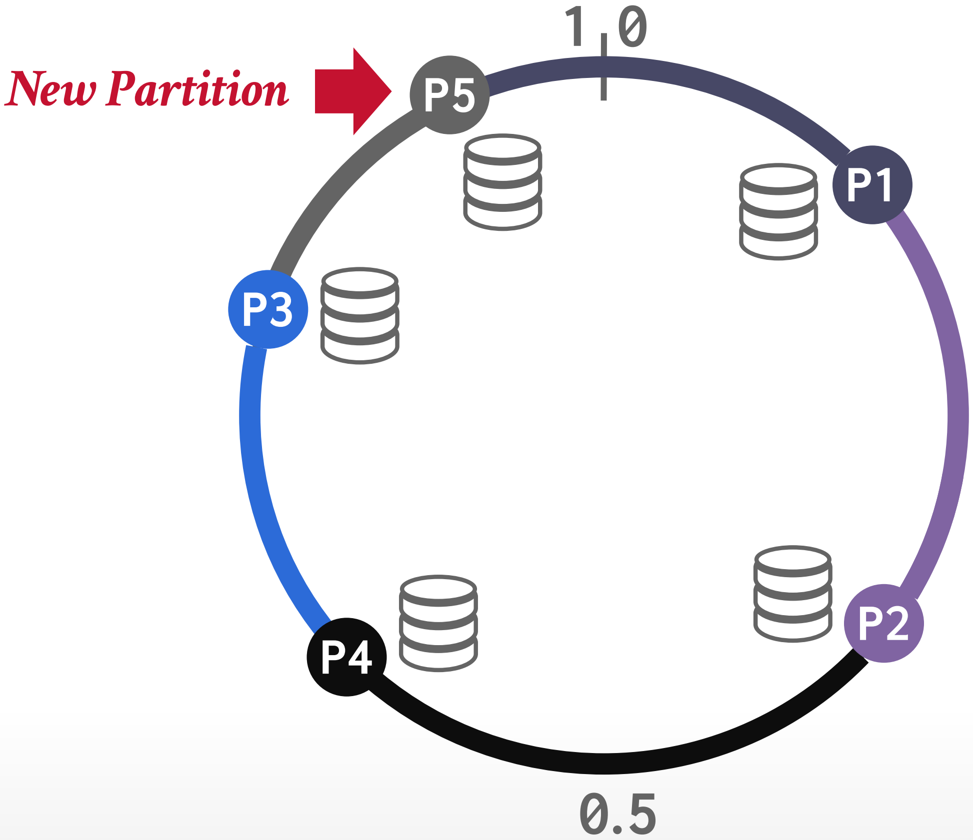
新分区的位置取决于有多少键被映射到那个空间，以尽可能负载均衡.

CONSISTENT HASHING

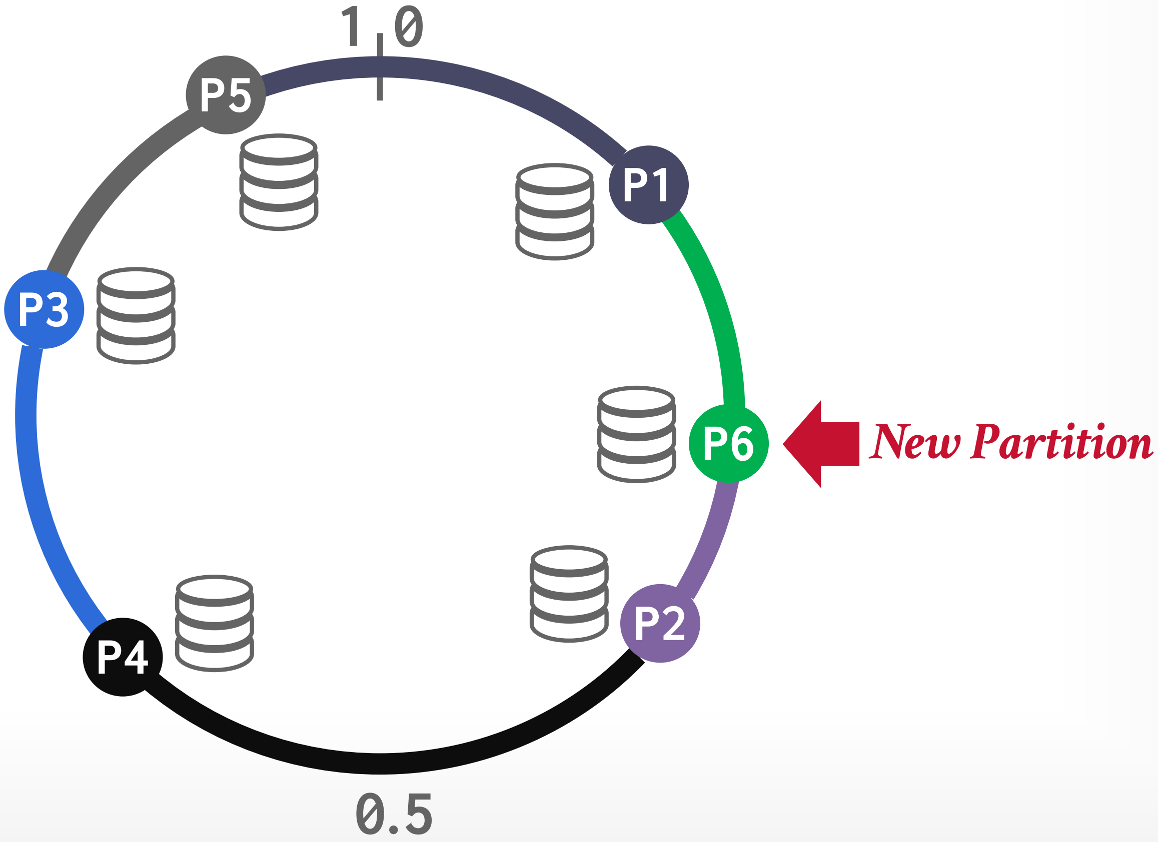


无需移动所有数据，仅需将原来属于 P3 的部分数据移动到新分区 P4，极大地减少数据的迁移量.

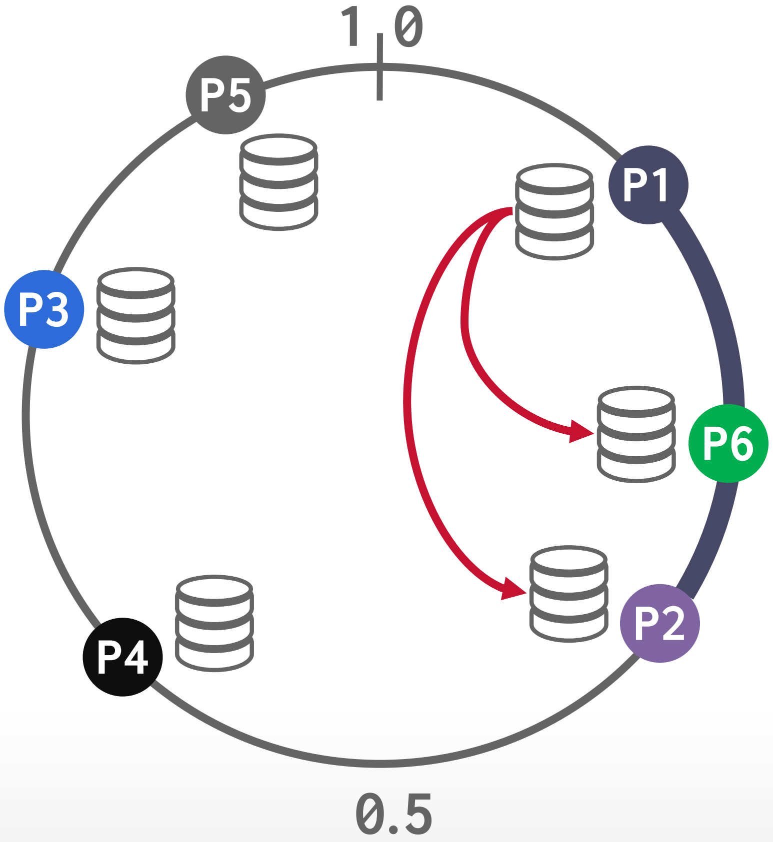
CONSISTENT HASHING



CONSISTENT HASHING

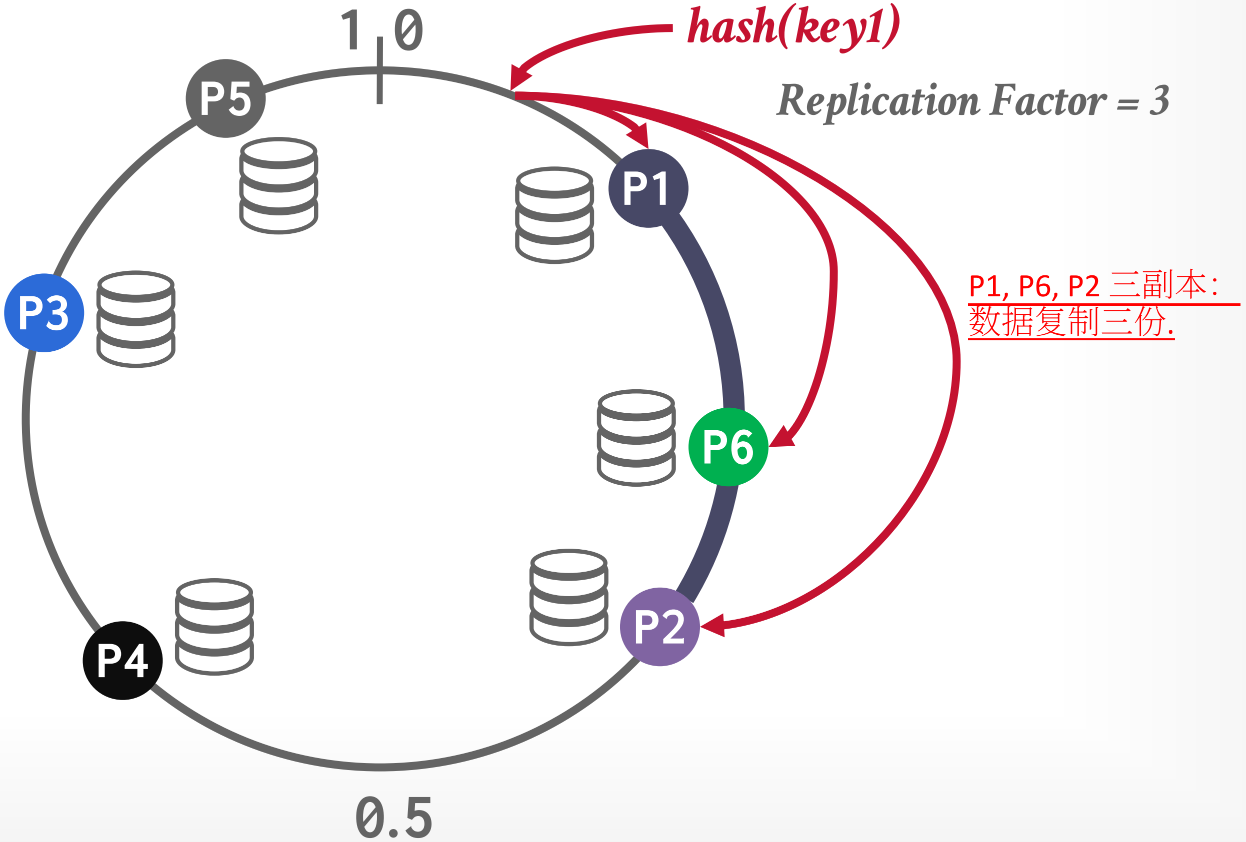


CONSISTENT HASHING



Replication Factor = 3

CONSISTENT HASHING



CONSISTENT HASHING

Couchbase

snowflake

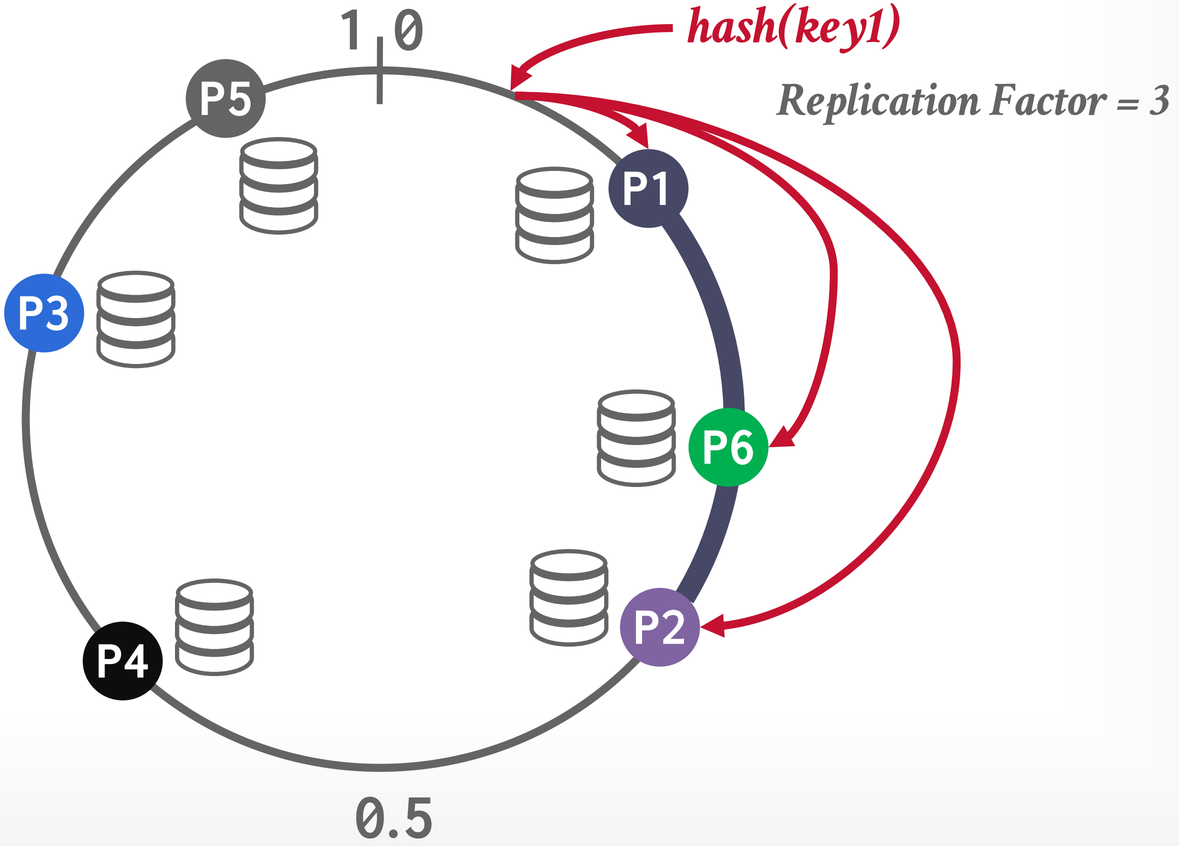
AEROSPIKE

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.

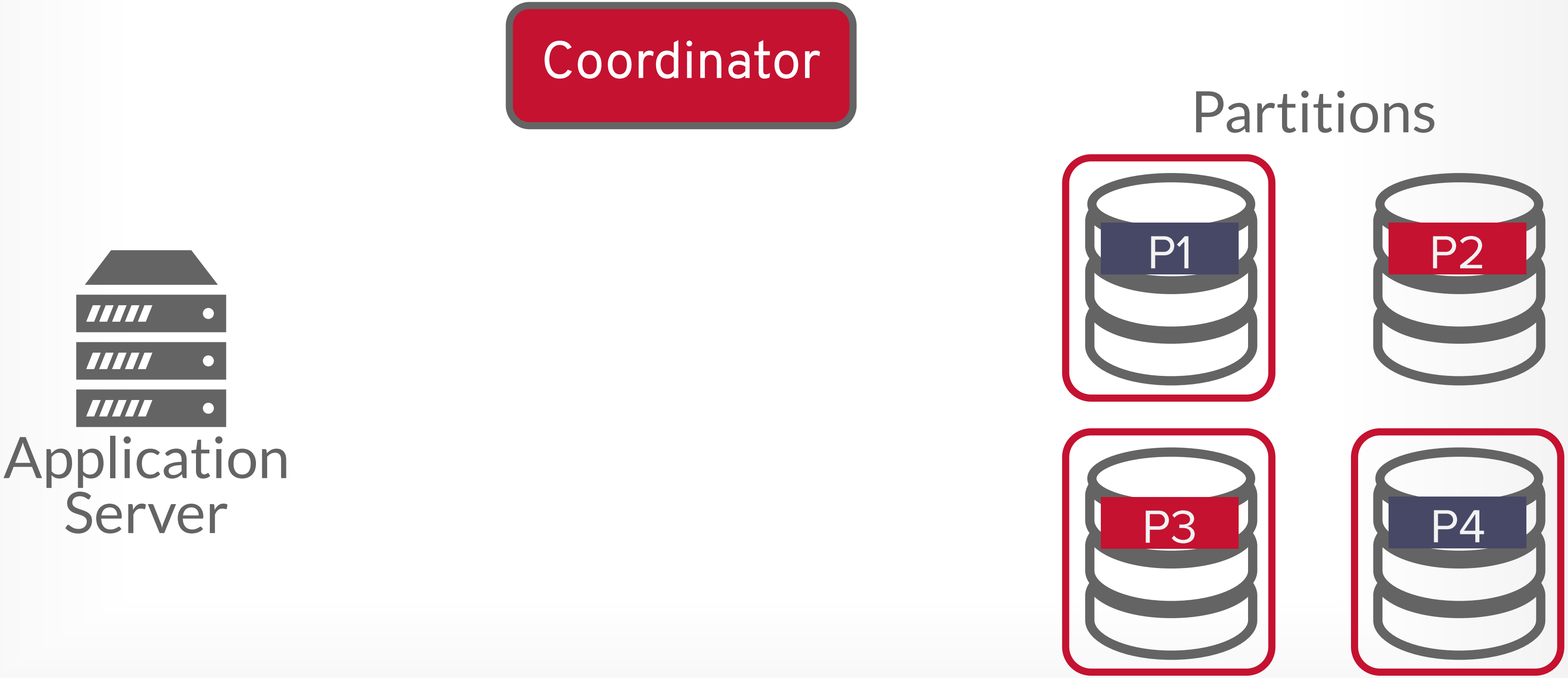
TRANSACTION COORDINATION

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

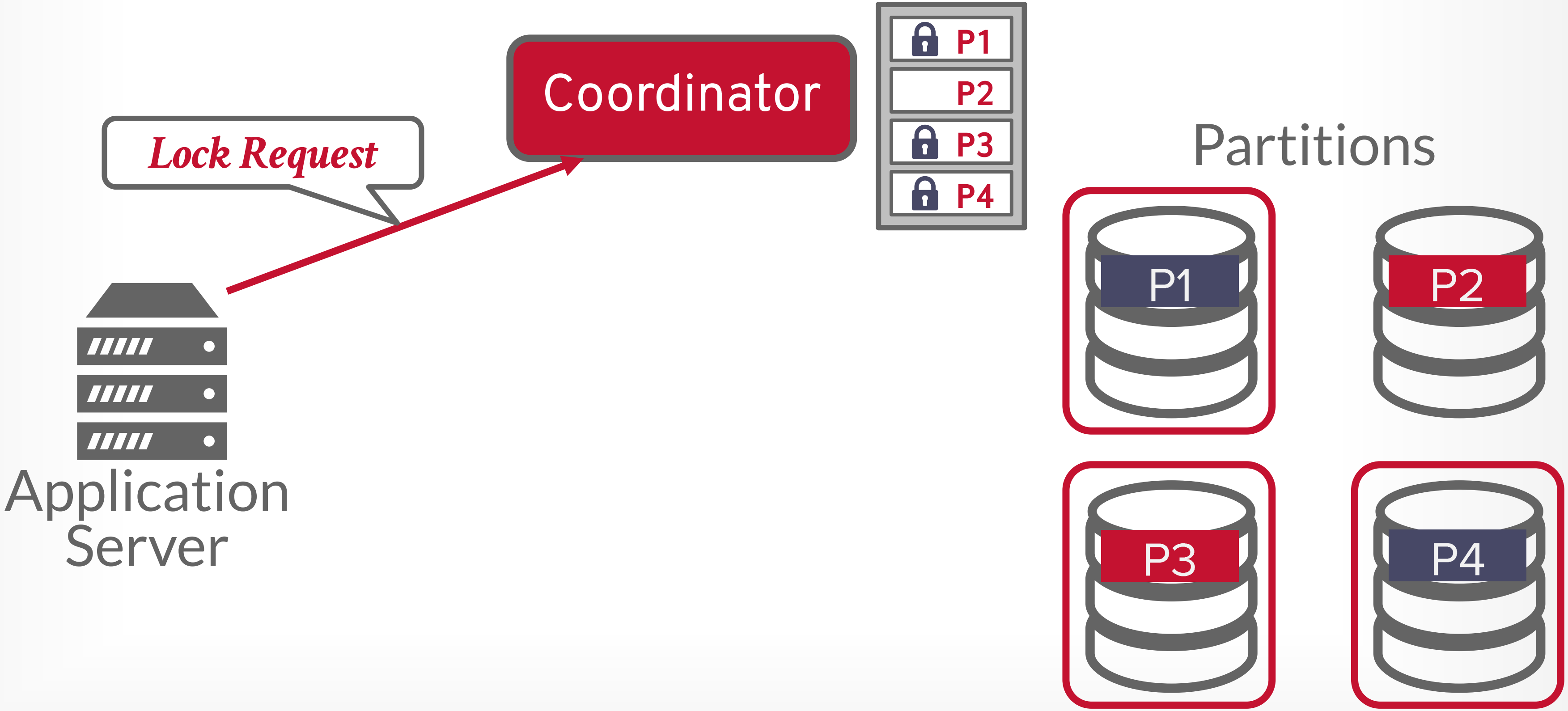
Two different approaches:
→ **Centralized**: Global "traffic cop".
→ **Decentralized**: Nodes organize themselves.

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

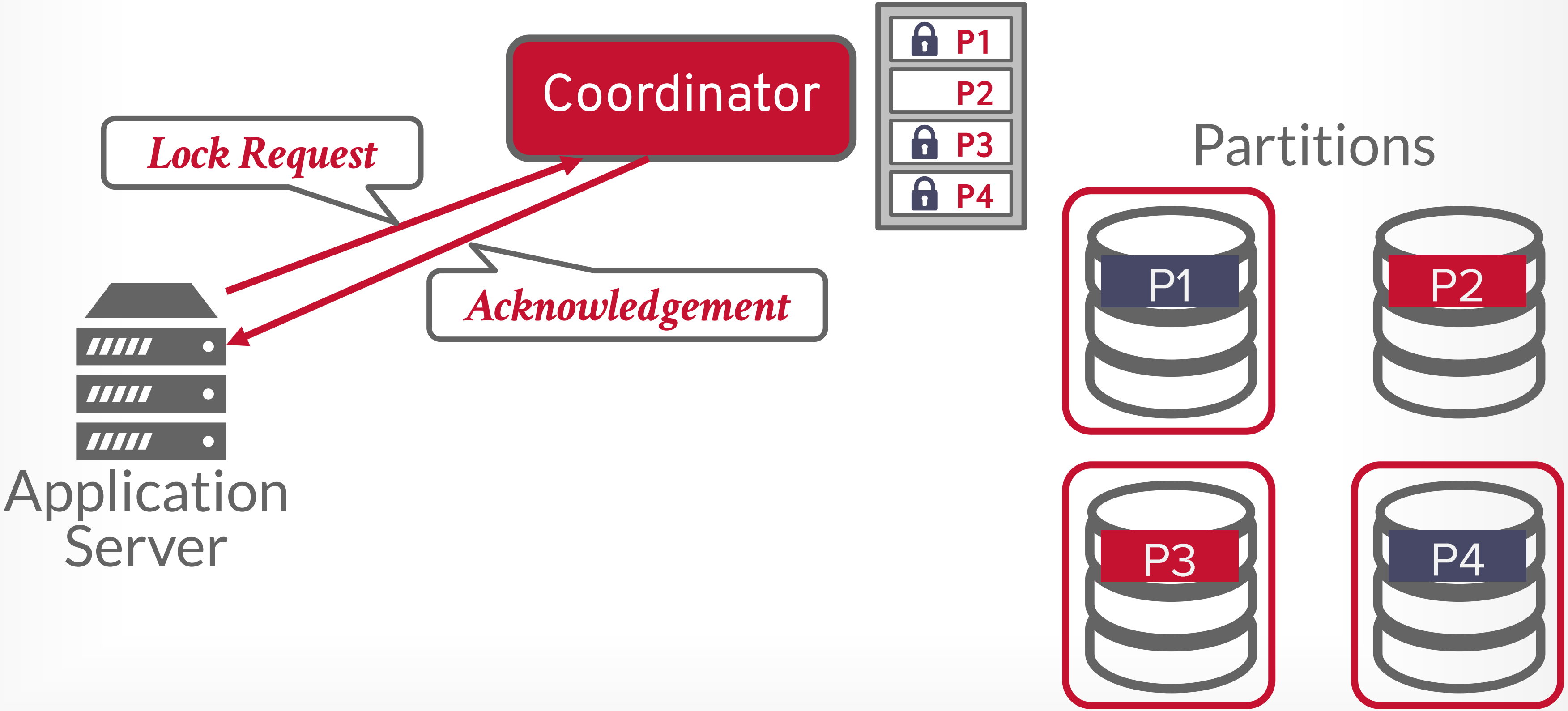
CENTRALIZED COORDINATOR



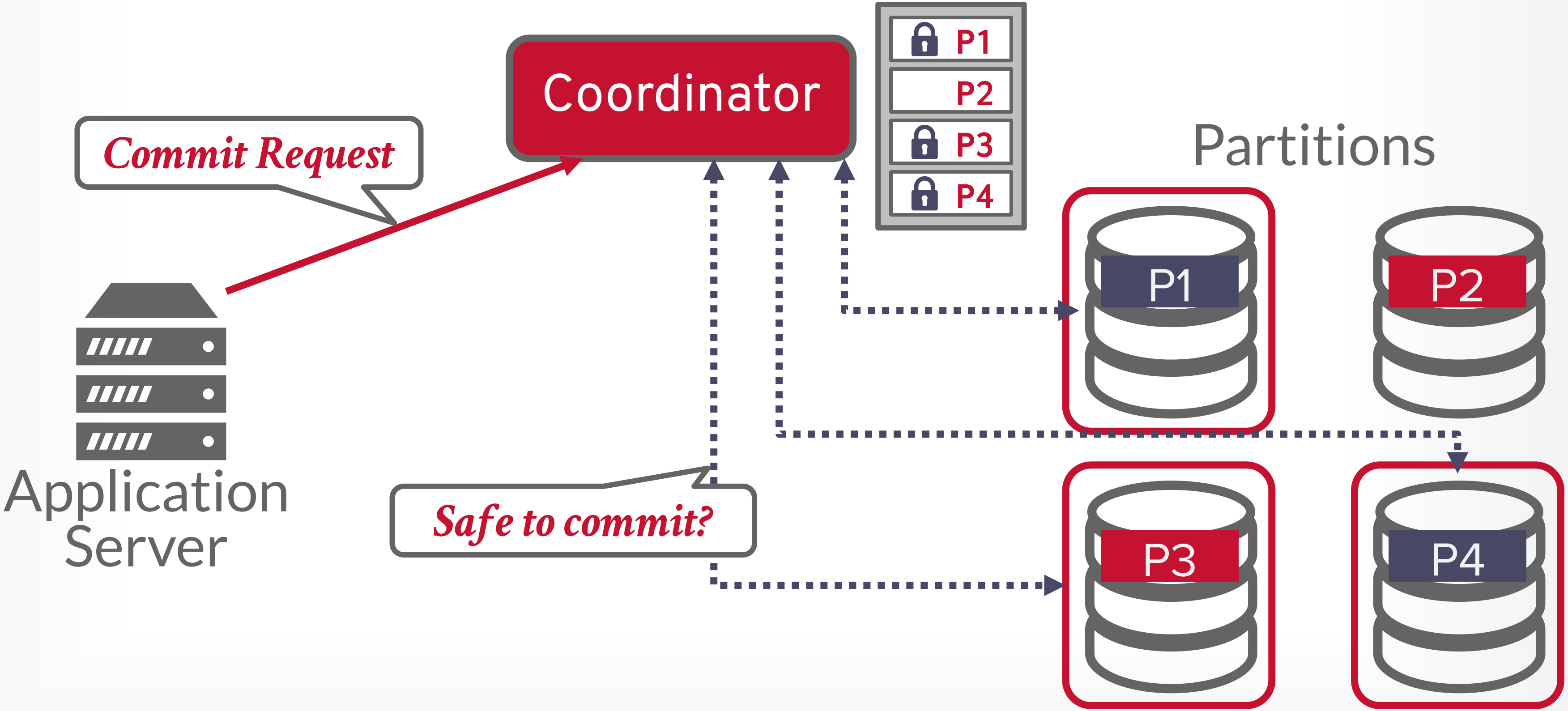
CENTRALIZED COORDINATOR



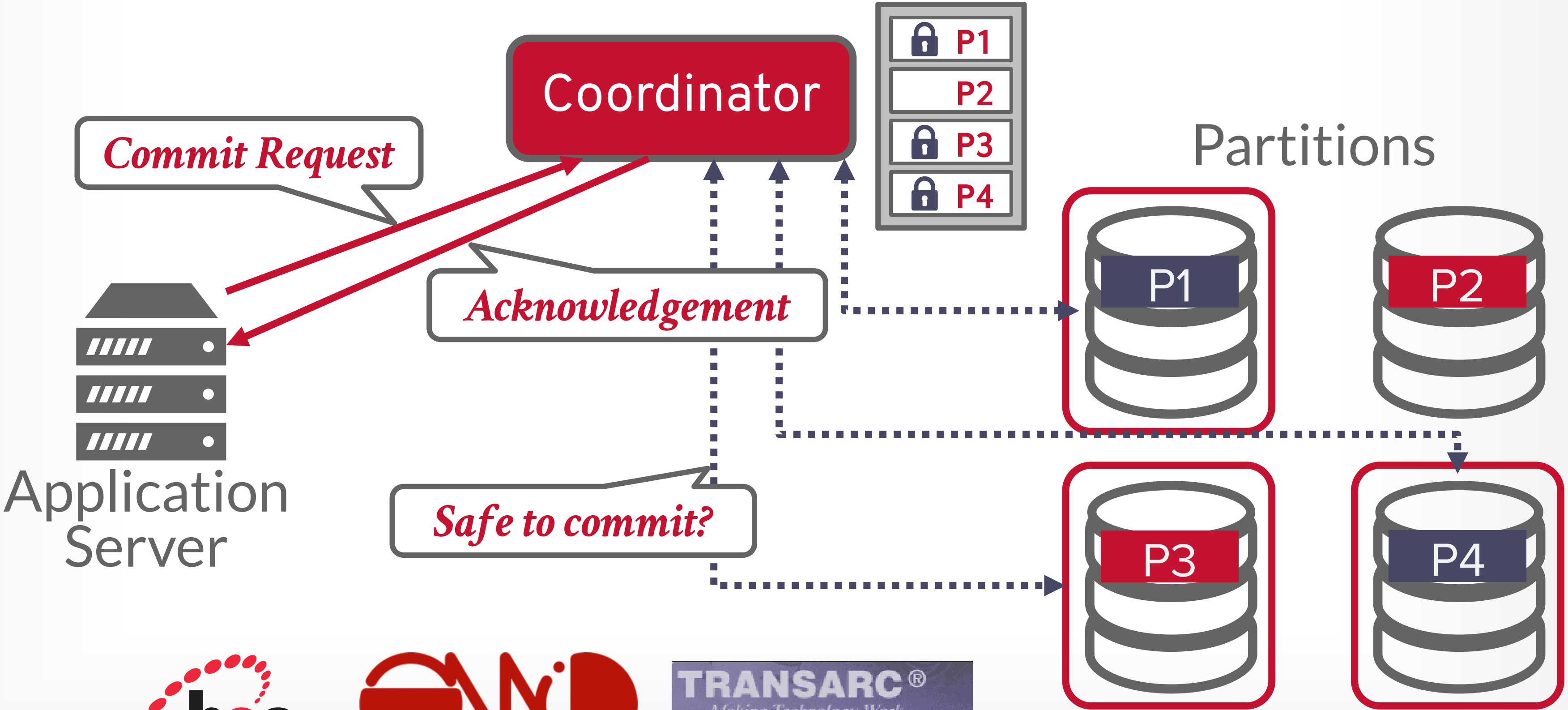
CENTRALIZED COORDINATOR



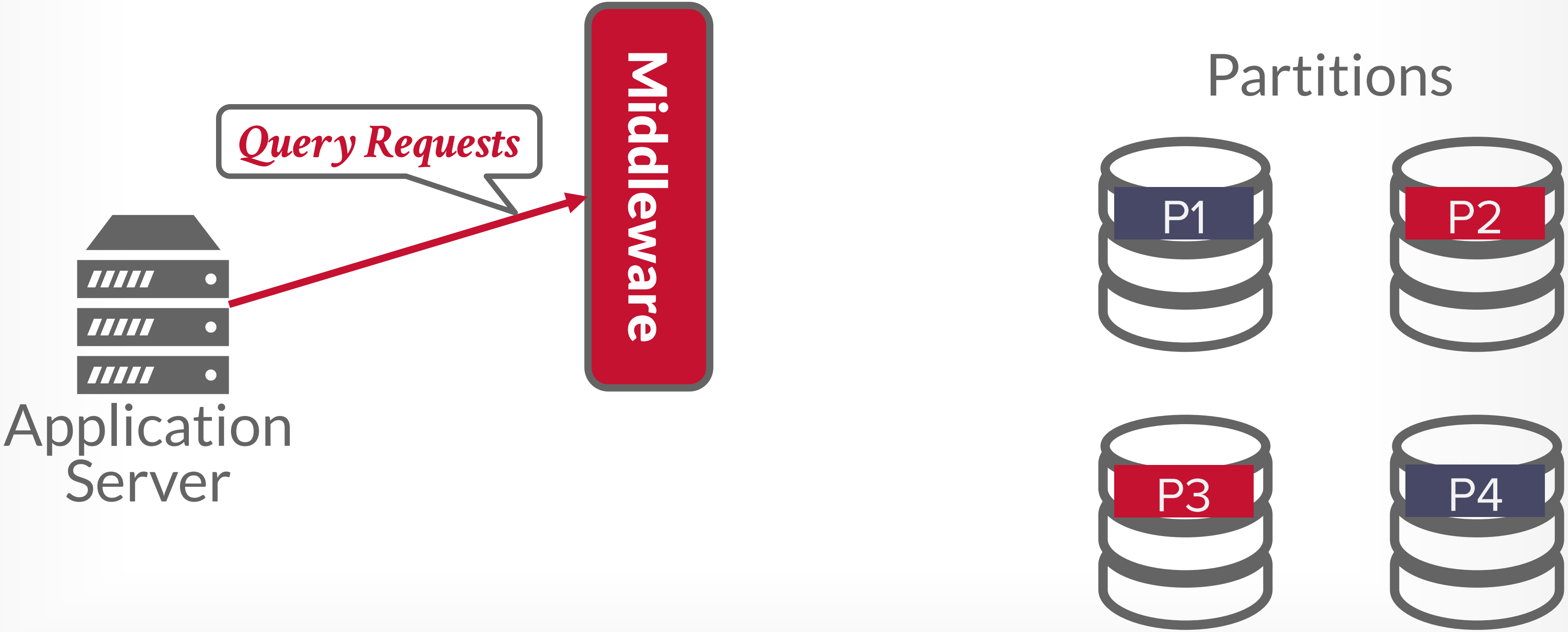
CENTRALIZED COORDINATOR



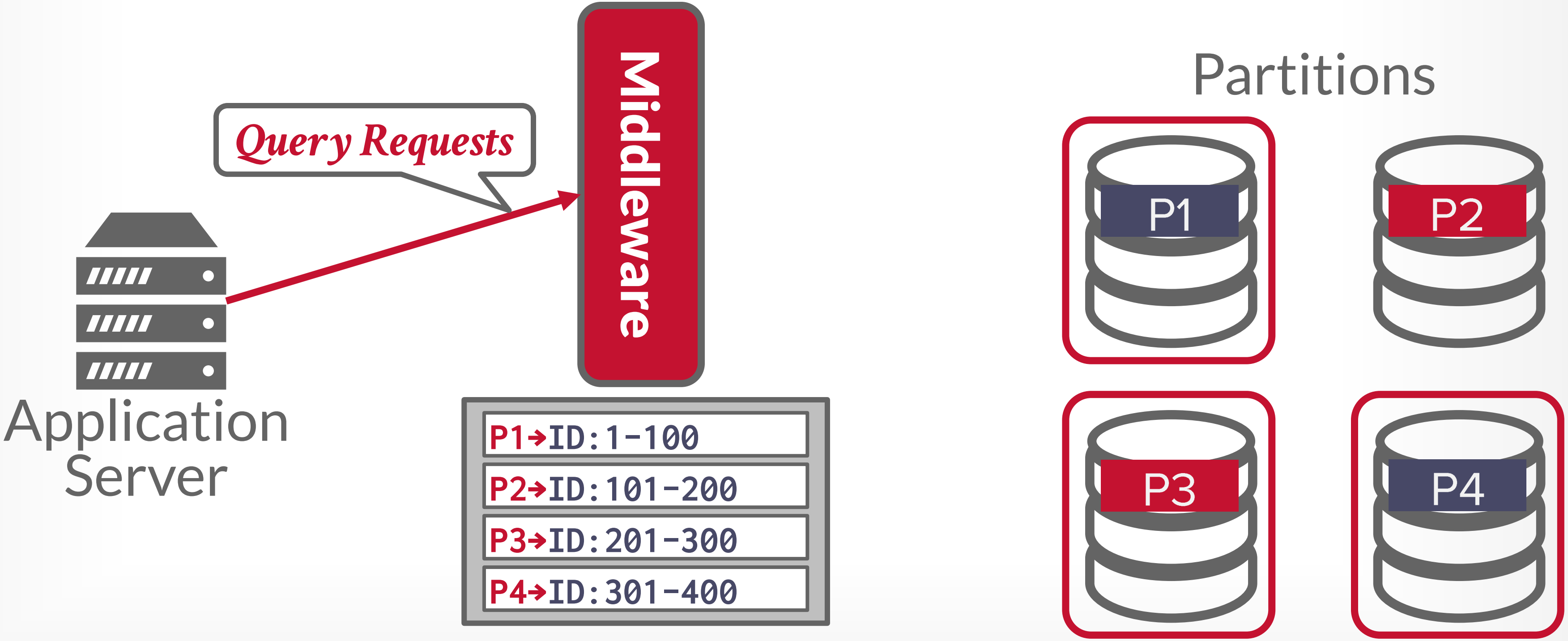
CENTRALIZED COORDINATOR



CENTRALIZED COORDINATOR

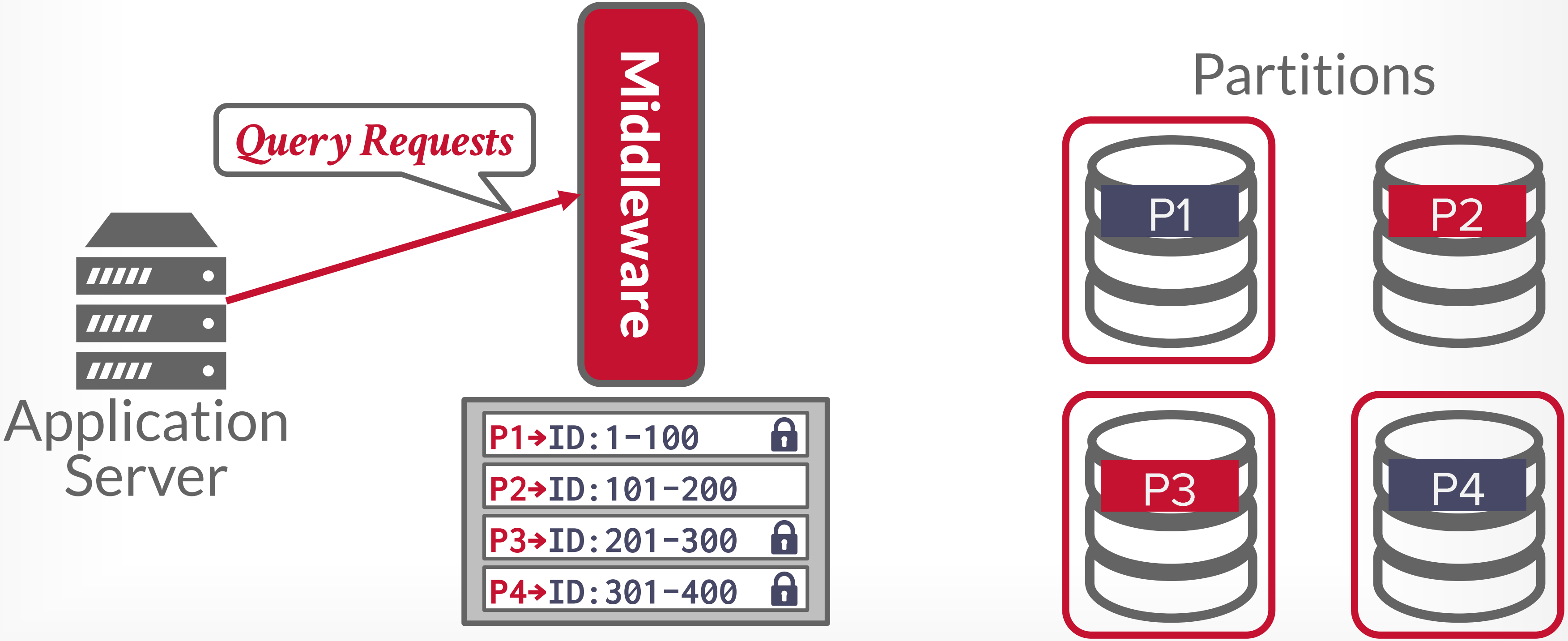


CENTRALIZED COORDINATOR

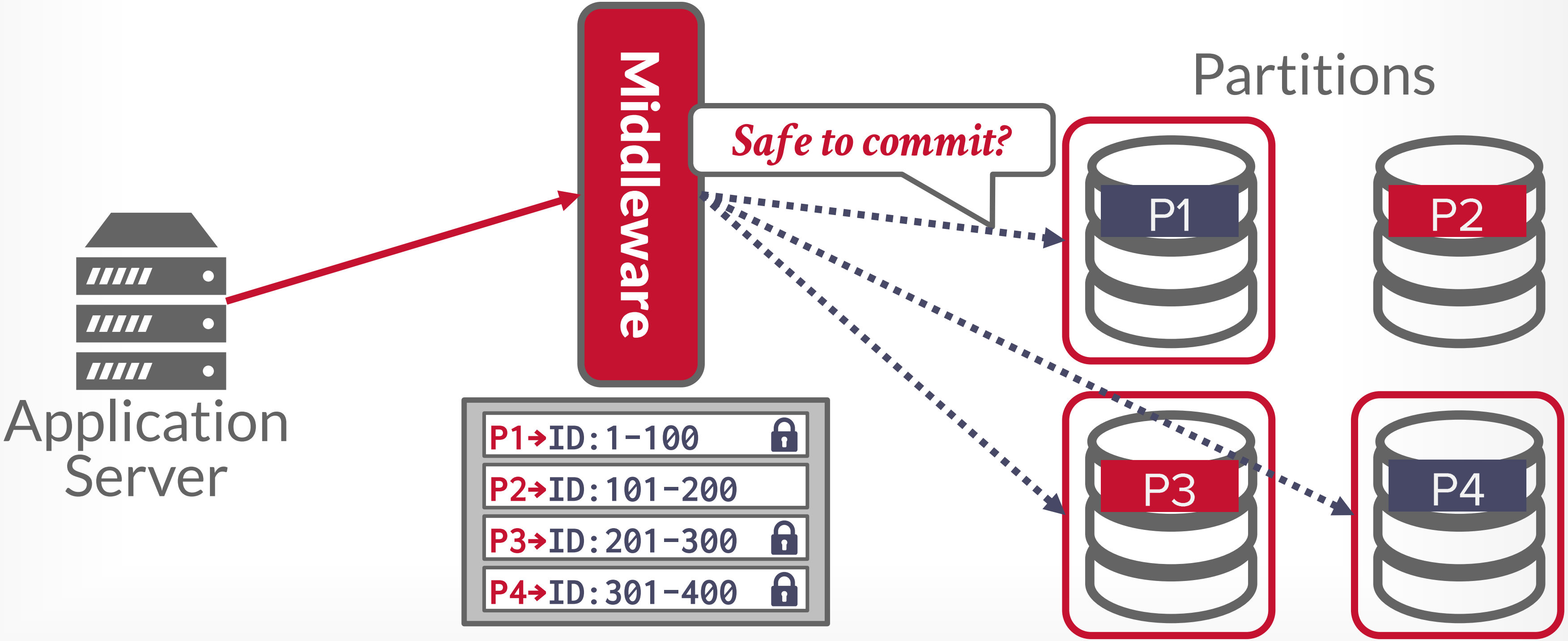


P1→ID:1-100
P2→ID:101-200
P3→ID:201-300
P4→ID:301-400

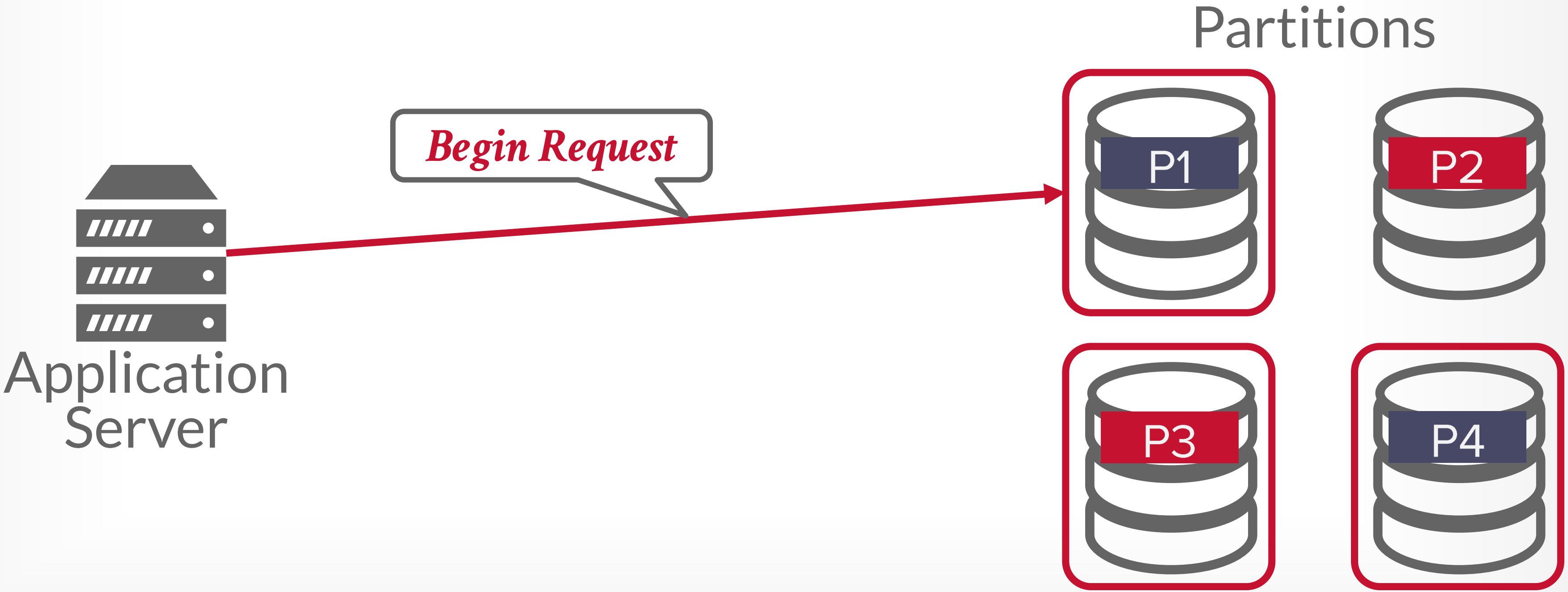
CENTRALIZED COORDINATOR



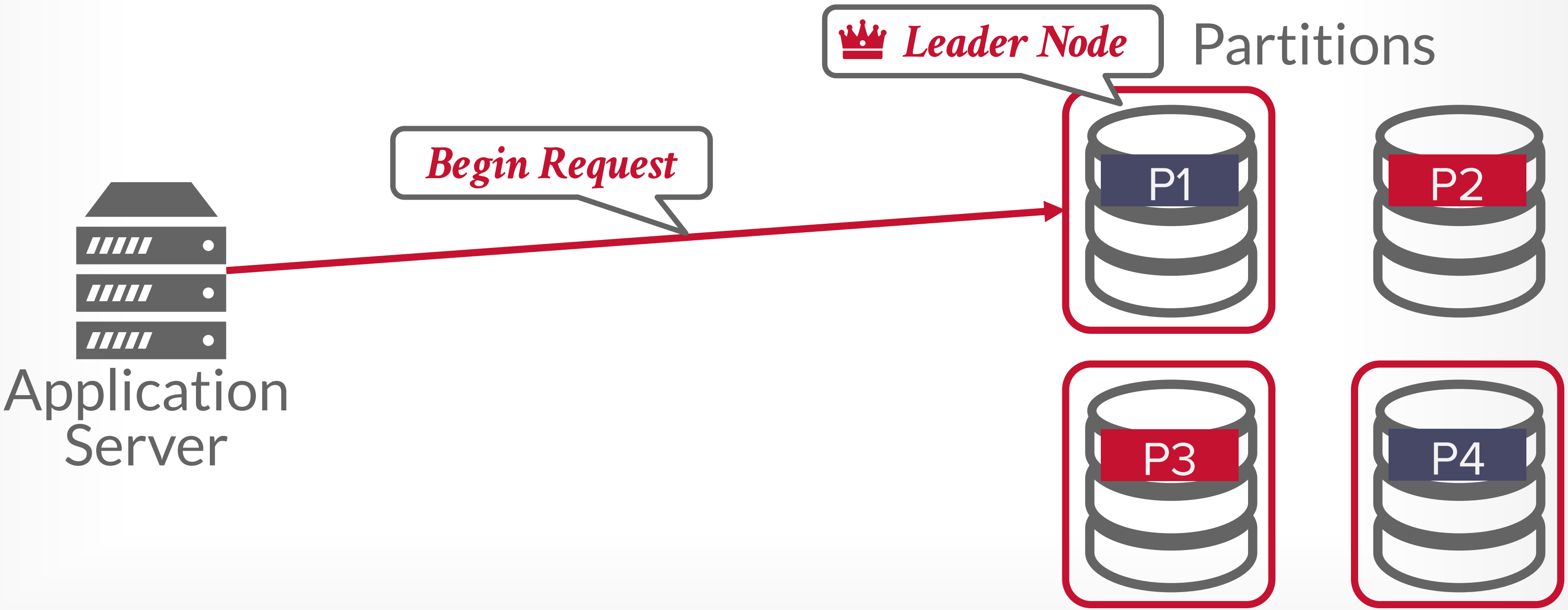
CENTRALIZED COORDINATOR



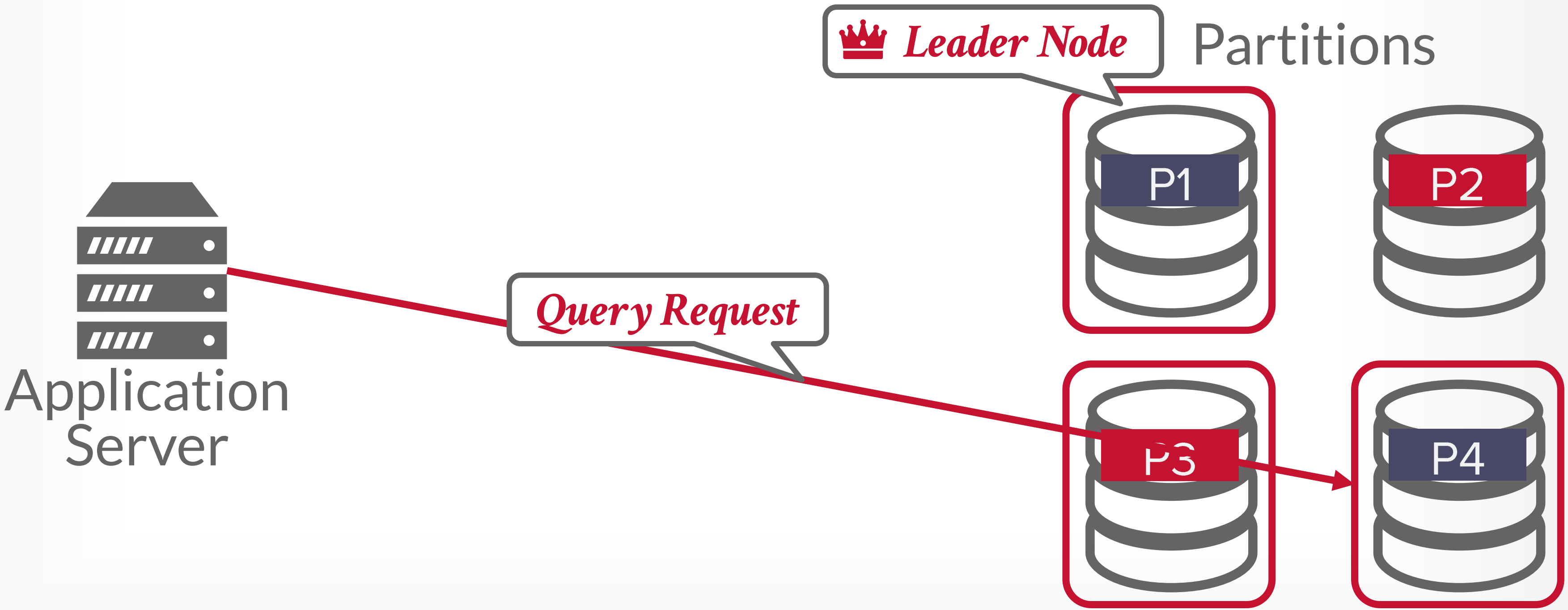
DECENTRALIZED COORDINATOR



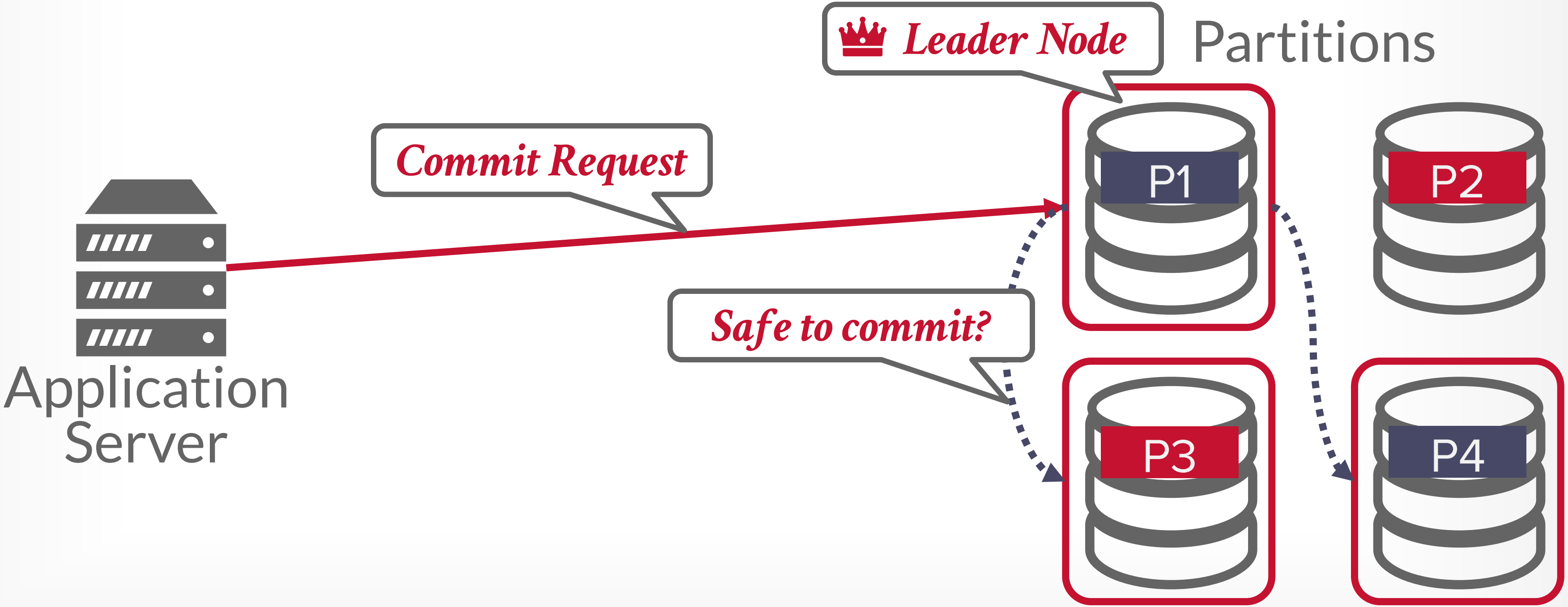
DECENTRALIZED COORDINATOR



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

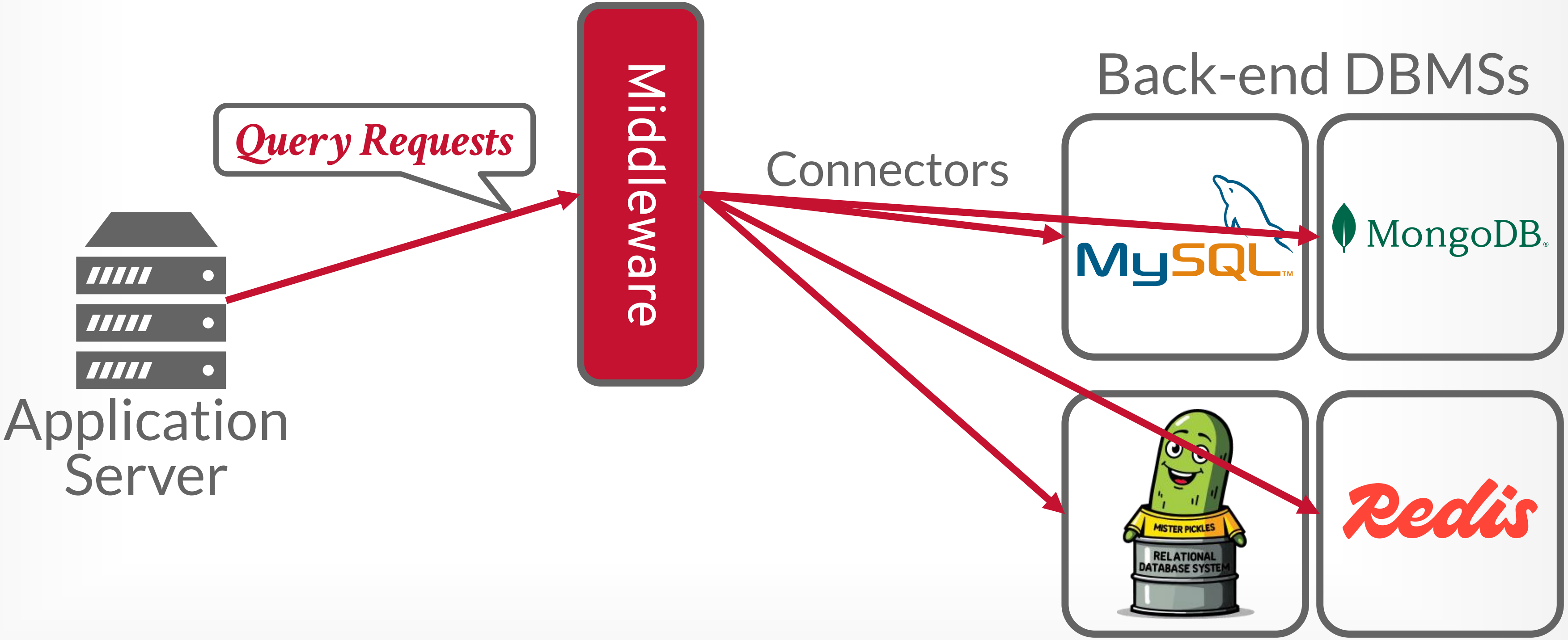
Distributed architecture that connects disparate DBMSs into a single logical system.

→ 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



DISTRIBUTED CONCURRENCY CONTROL

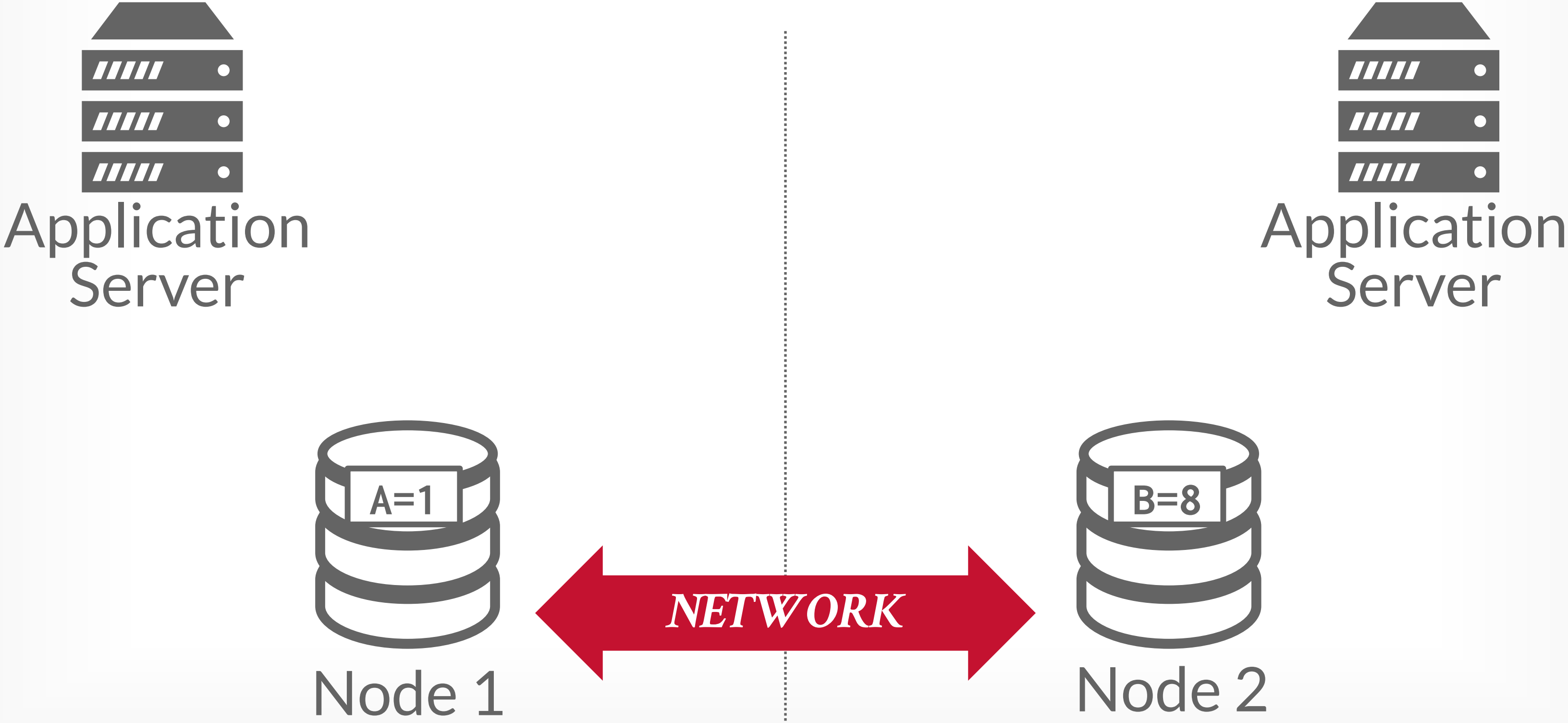
Need to allow multiple txns to execute simultaneously across multiple nodes.

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

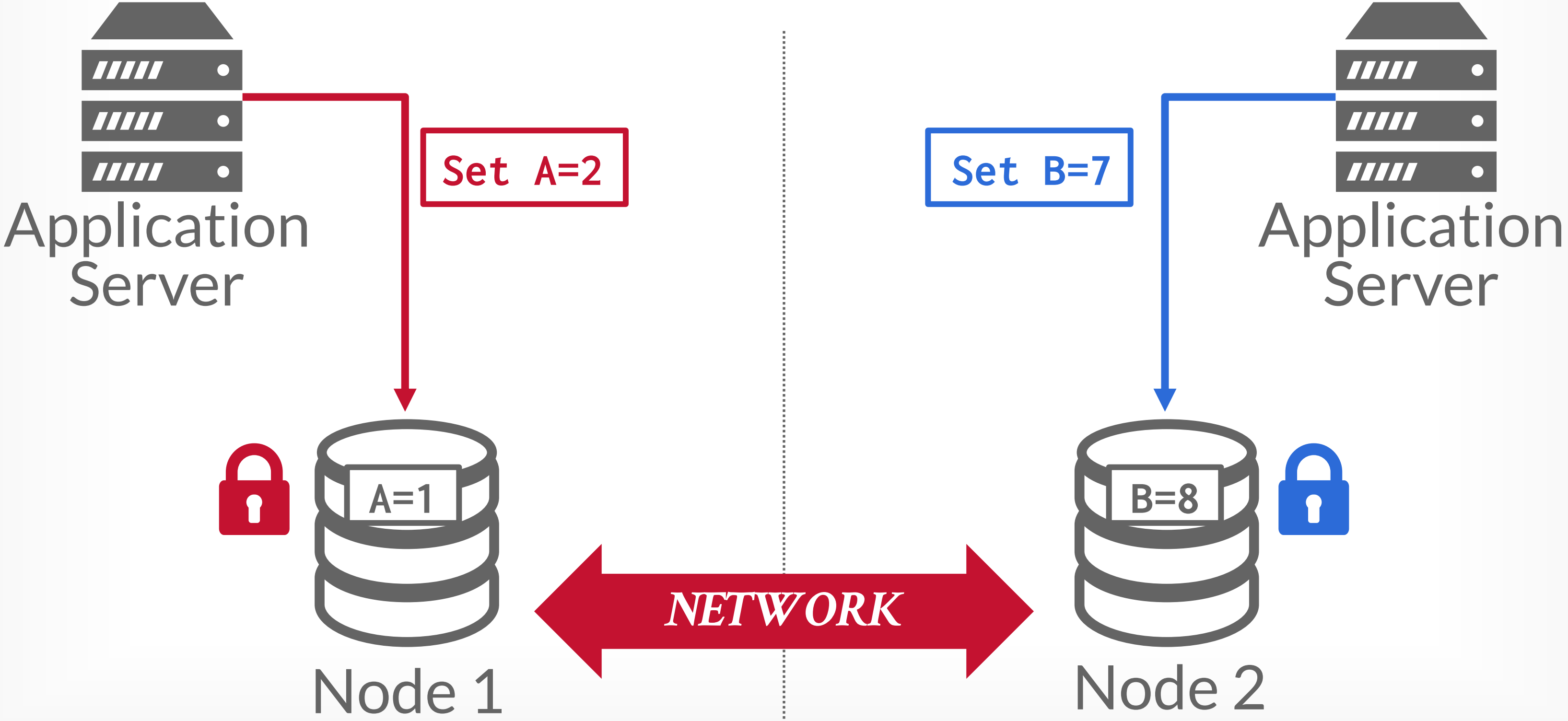
This is harder because of:

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

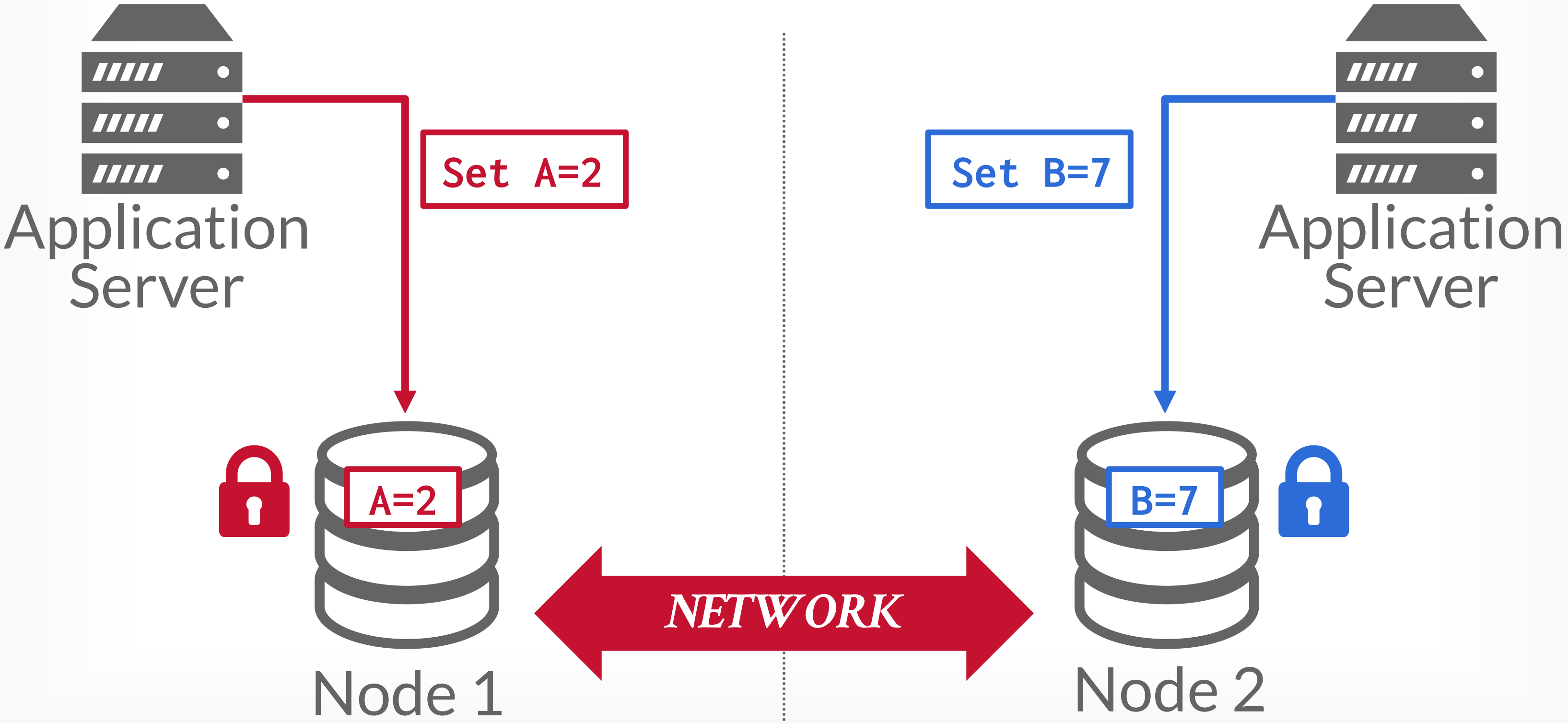
DISTRIBUTED 2PL



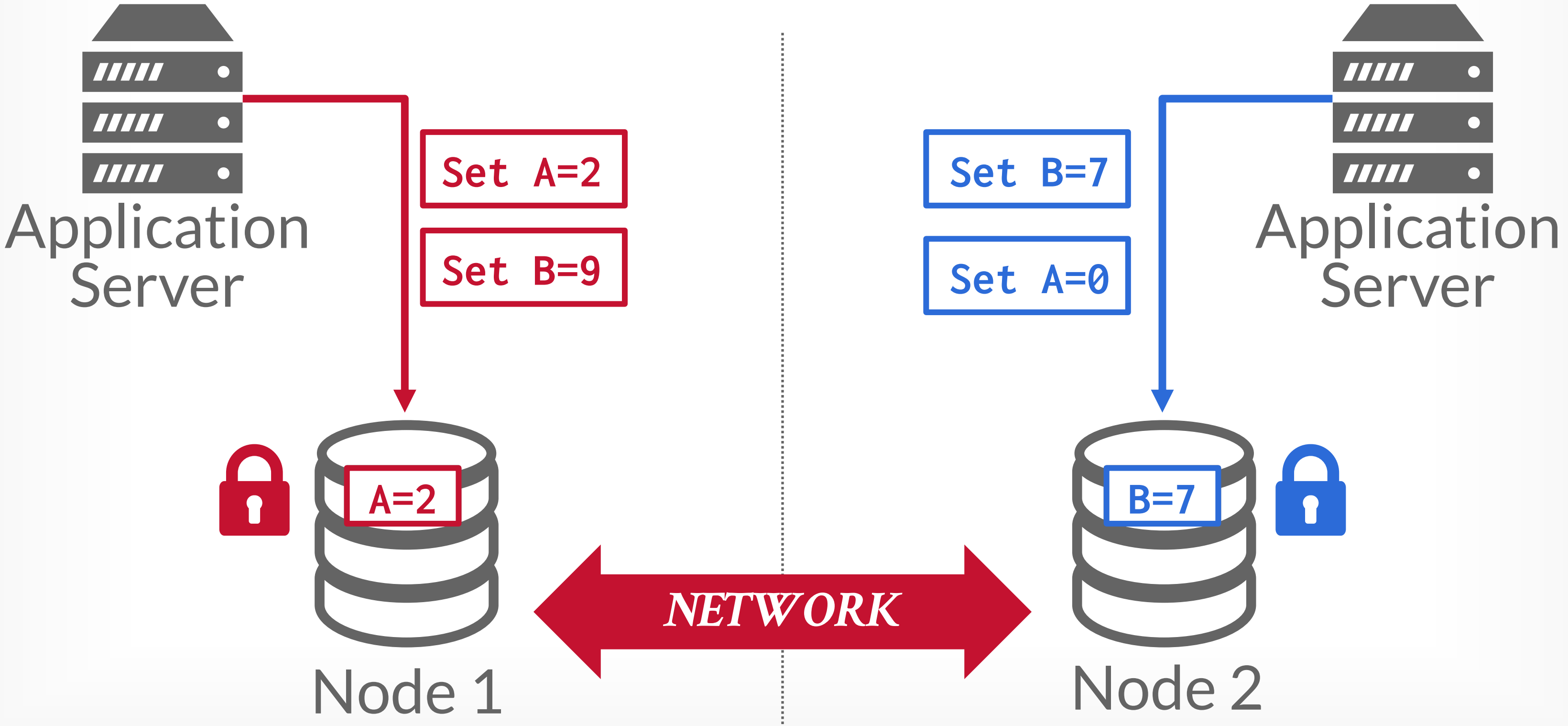
DISTRIBUTED 2PL



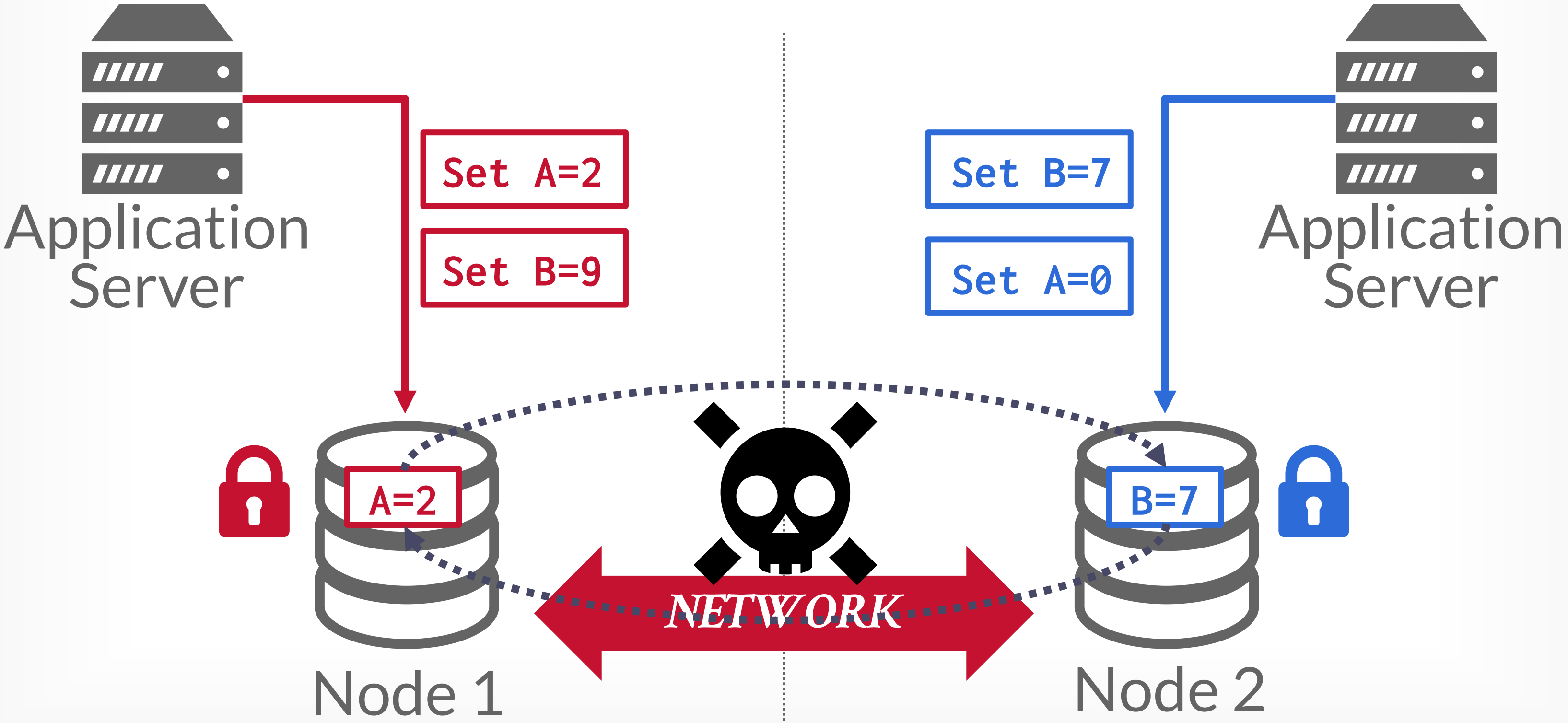
DISTRIBUTED 2PL



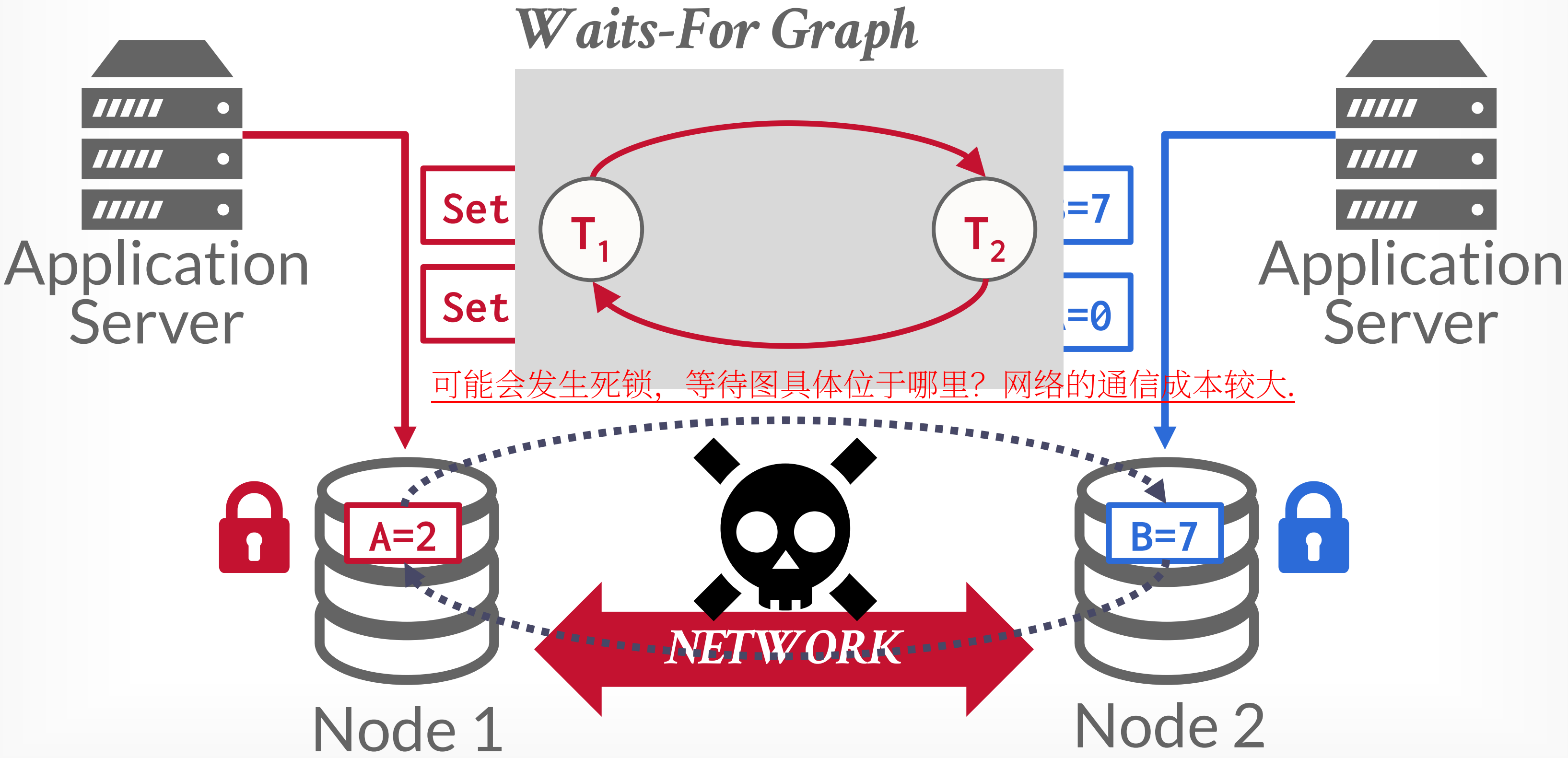
DISTRIBUTED 2PL



DISTRIBUTED 2PL



DISTRIBUTED 2PL



可能会发生死锁，等待图具体位于哪里？网络的通信成本较大.

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