Advanced Databases

Spring 2020

Lecture #16:

Distributed Database Systems

Milos Nikolic

THE UNIVERSITY of EDINBURGH

Parallel/Distributed DBMSs

Why Do We Need Parallel/Distributed DBMSs?

- Increased performance
  - Throughput and latency
- Increased availability
- Potentially lower TCO (total cost of ownership)

Database is spread out across multiple resources to improve parallelism

Appears as a single database instance to the application

- SQL query for a single-node DBMS should generate same result on a parallel or distributed DBMS

Parallel vs. Distributed DBMSs

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

System Architecture

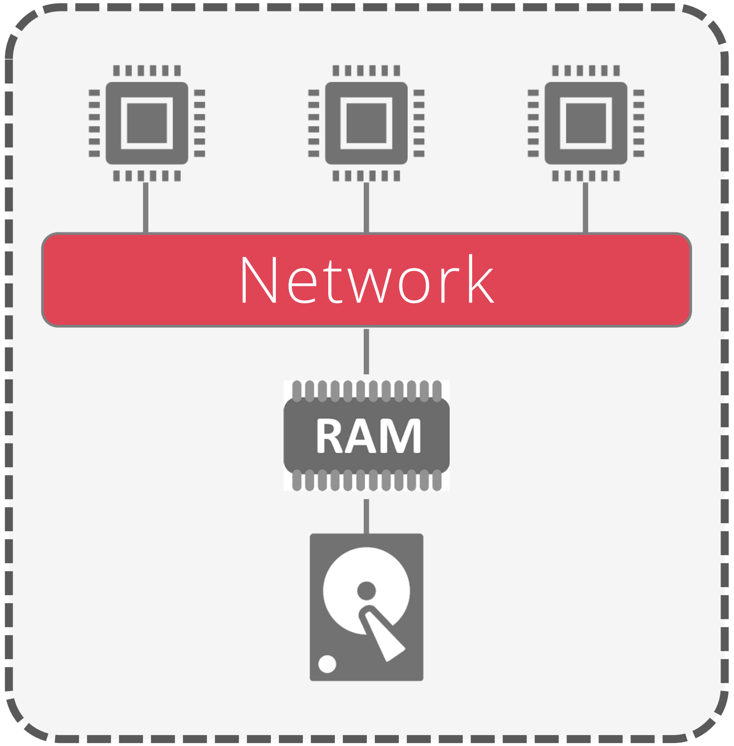
A DBMS's system architecture specifies what shared resources are directly accessible to CPUs

The goal is to parallelize operations across multiple resources

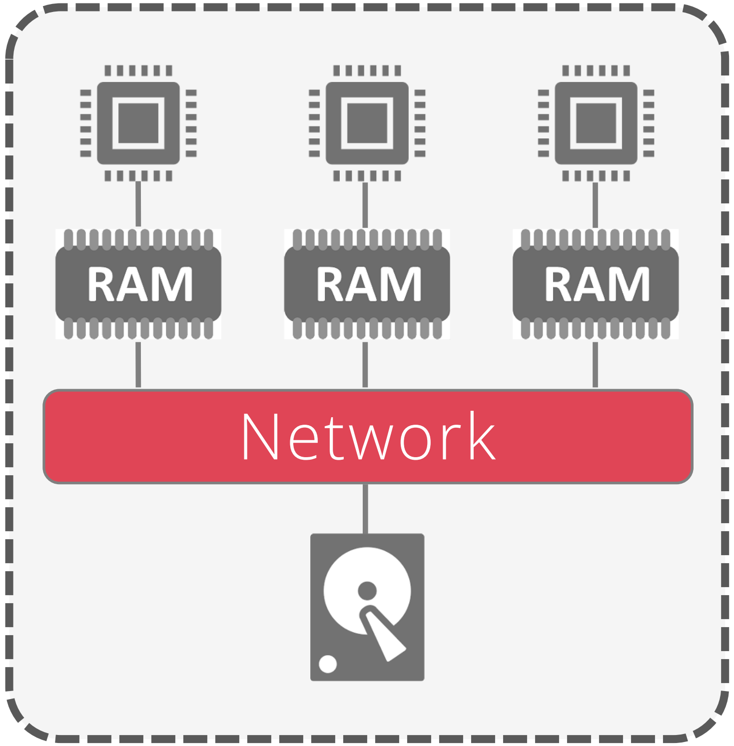
- CPU, memory, network, disk

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

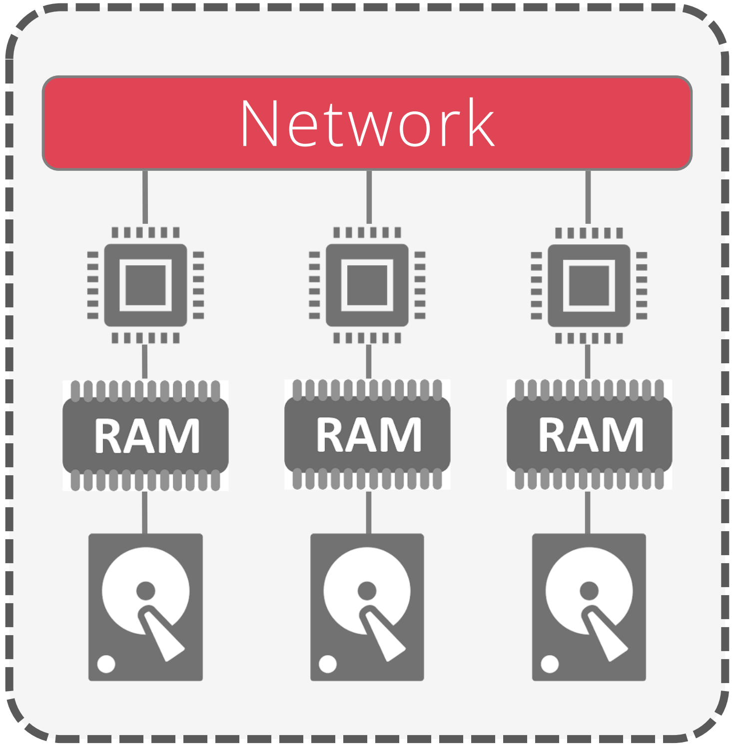
System Architecture



Shared Memory



Shared Disk

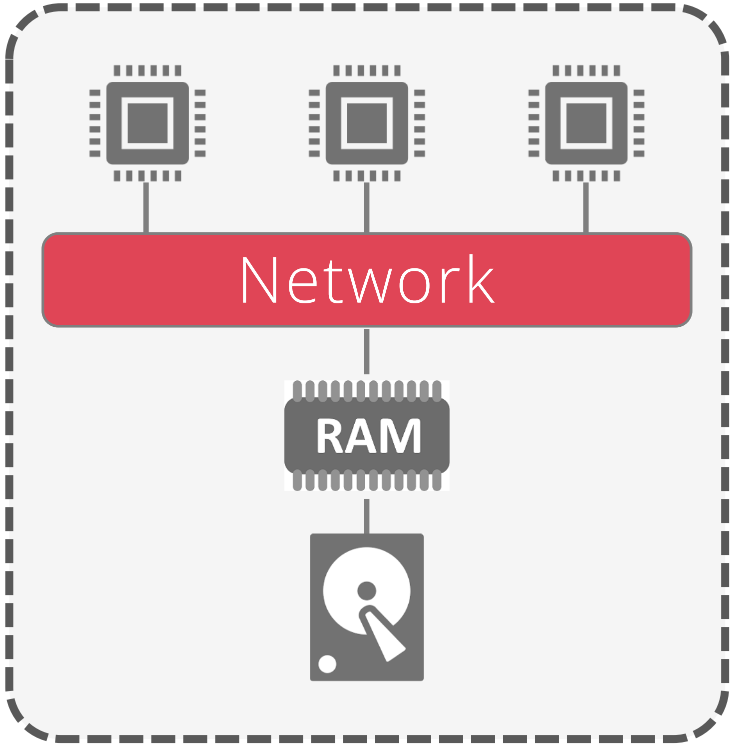


Shared Nothing

Shared Memory

CPUs have access to common memory address space via a fast interconnect

- Efficient to send messages between processors
- Each processor has a global view of all the in-memory data structures
- Each DBMS instance on a processor has to "know" about the other instances

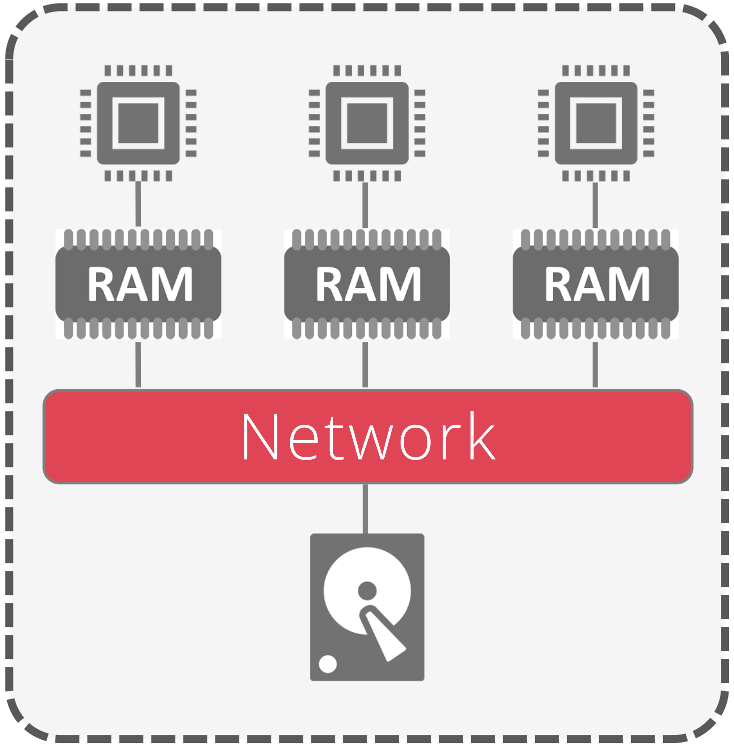


Sometimes called "shared everything"

Shared Disk

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

- Can scale execution layer independently from the storage layer
- Easy consistency since there is a single copy of DB
- Easy fault tolerance

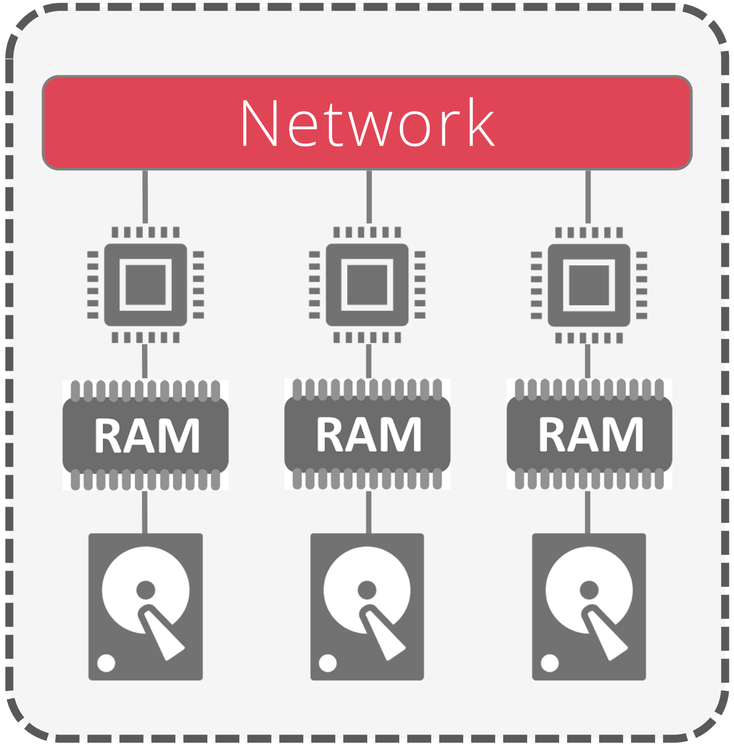


Shared Nothing

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

Nodes only communicate with each other via network

- Easy to increase capacity
- Hard to ensure consistency



Types of Parallelism in DBMSs

Inter-Query: Different queries are executed concurrently

- Increases throughput & reduces latency

Intra-Query: Execute the operations of a single query in parallel

- Decreases latency for long-running queries
- **Inter-operator:** Execute operators of a query in parallel (exploits pipelining)
- **Intra-operator:** Get all CPUs to compute a given operation (scan, sort, join)

Parallel/Distributed DBMS

Advantage

- Data sharing
- Reliability and availability
- Speed up of query processing

Disadvantage

- May increase processing overhead
- Harder to ensure ACID guarantees
- More database design issues

Design Issues

How do we store data across nodes?

How does the application find data?

How to execute queries on distributed data?

- Push query to data
- Pull data to query

How does the DBMS ensure correctness?

This lecture

Data Transparency

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

A SQL query that works on a single node DBMS should work the same on a distributed DBMS

Database Partitioning

Split database across multiple resources:

- Disks, nodes, processors
- Sometimes called "sharding"

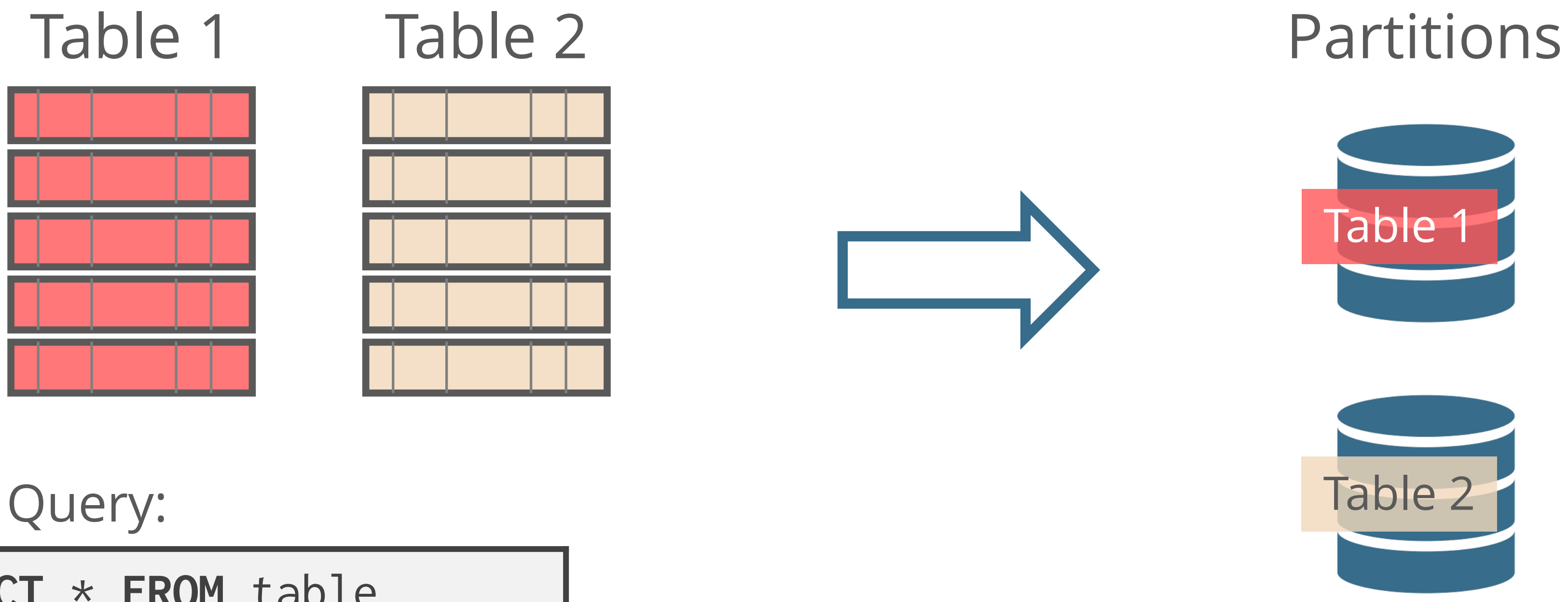
The DBMS executes query fragments on each partition and then combines the results to produce a single answer

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

Naïve Table Partitioning

Each node stores one and only table

Assumes that each node has enough storage space for a table



Ideal Query:

```
SELECT * FROM table
```

Horizontal Partitioning

Split a table's tuples into disjoint subsets

- Choose column(s) that divides the database equally in terms of size, load, or usage
- Each tuple contains all of its columns

Three main approaches:

- Round-robin Partitioning
- Hash Partitioning
- Range Partitioning

HORIZONTAL PARTITIONING

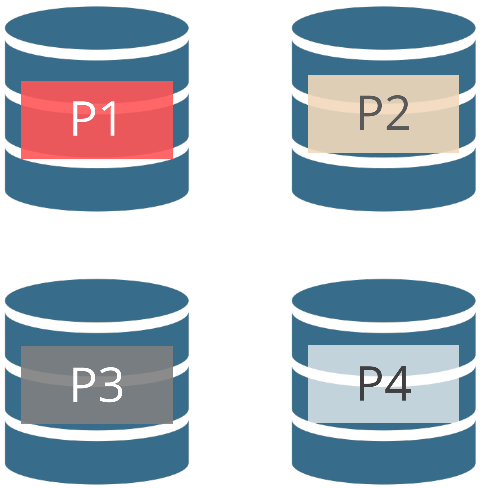
Partition Key

Table

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

hash(a) % 4 = P2
hash(b) % 4 = P4
hash(c) % 4 = P3
hash(d) % 4 = P2
hash(e) % 4 = P1

Partitions



Ideal Query:

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

Vertical Partitioning

Split the columns of tuples into fragments:

- Each fragment contains all of the tuples' values for column(s)

Use fixed length attribute values to ensure that the original tuple can be reconstructed

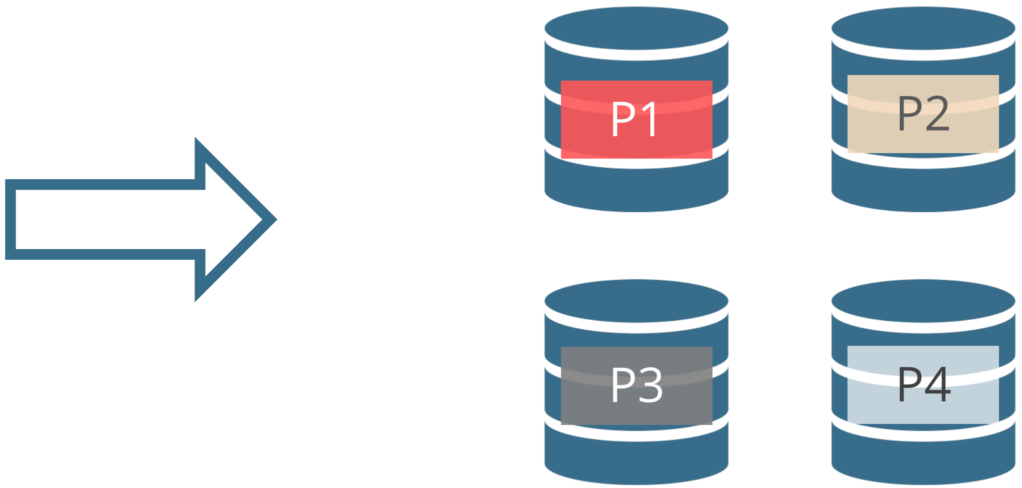
Column fragments can also be horizontally partitioned

Vertical Partitioning

Table

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

Partitions



Ideal Query:

```
SELECT column FROM table
```

REPLICATION

The DBMS can replicate data across nodes to increase availability

Partition Replication: Store a copy of an entire partition in multiple locations

- Master – Slave Replication

Table Replication: Store an entire copy of a table in each partition

- Usually small, read-only tables

The DBMS ensures updates are propagated to all replicas in either case

Replica Configurations

Approach #1: Master-Replica

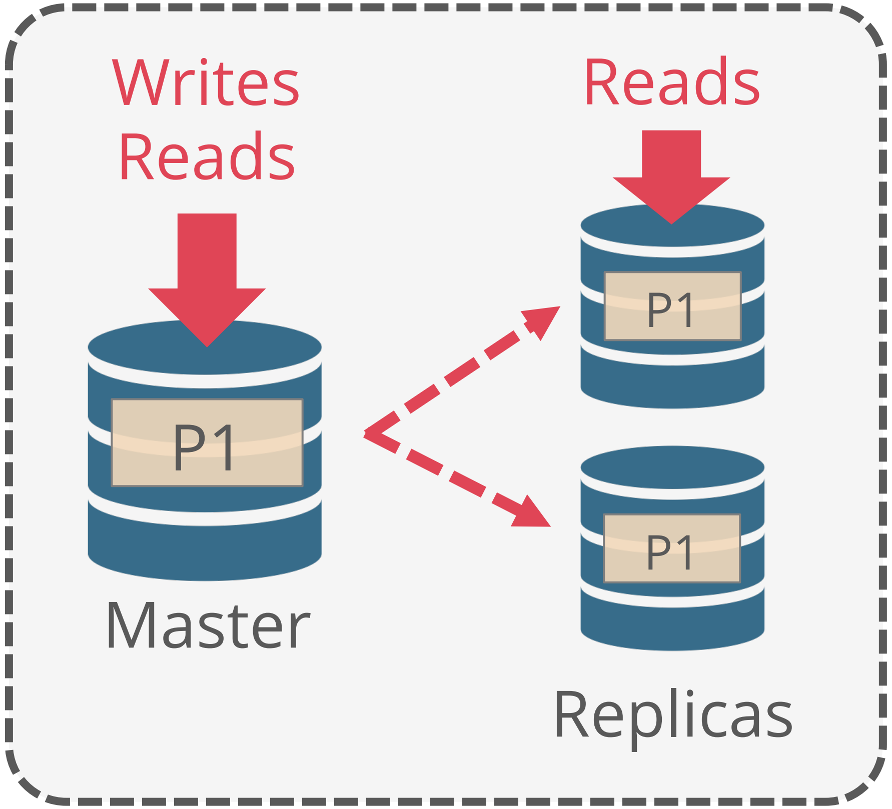
- All updates go to a designated master for each object
- The master then propagates those updates to its replicas
- Read only txns may be allowed to access replicas
- If the master goes down, then hold an election to select a new master

Approach #2: Multi-Master

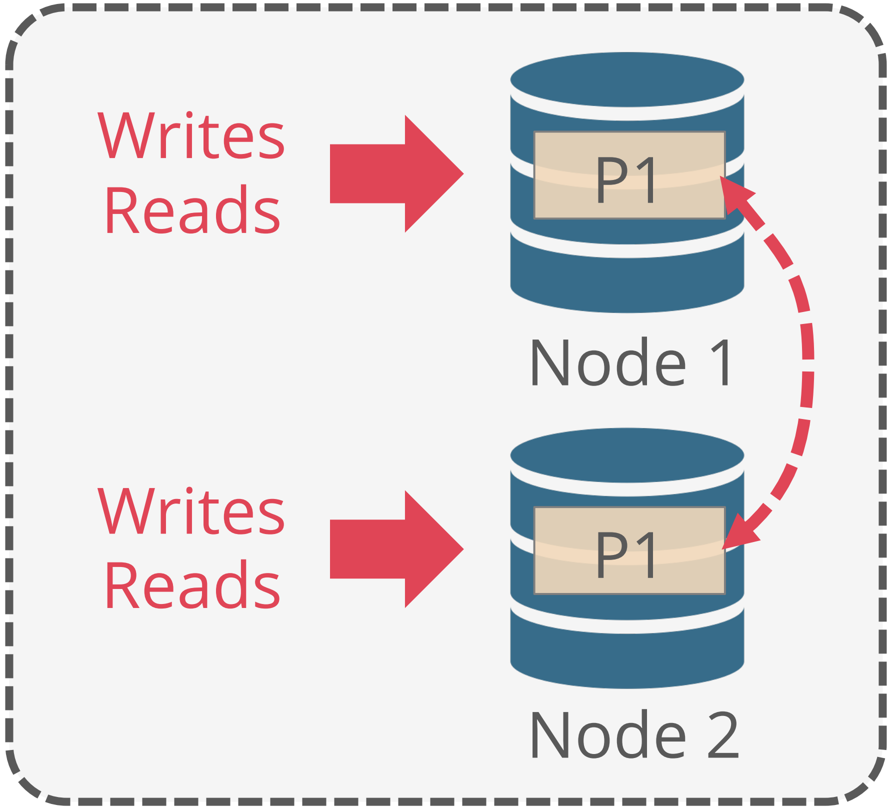
- Txns can update data objects at any replica
- Replicas synchronize with each other

Replica Configurations

Master-Replica



Multi-Master



OLTP vs. OLAP

On-line Transaction Processing (OLTP):

- Short lived read/write txns
- Small footprint
- Repetitive operations

On-line Analytical Processing (OLAP): → This lecture

- Long running, read only queries
- Complex joins
- Exploratory queries

Distributed OLAP

Execute analytical queries that examine large portions of the database

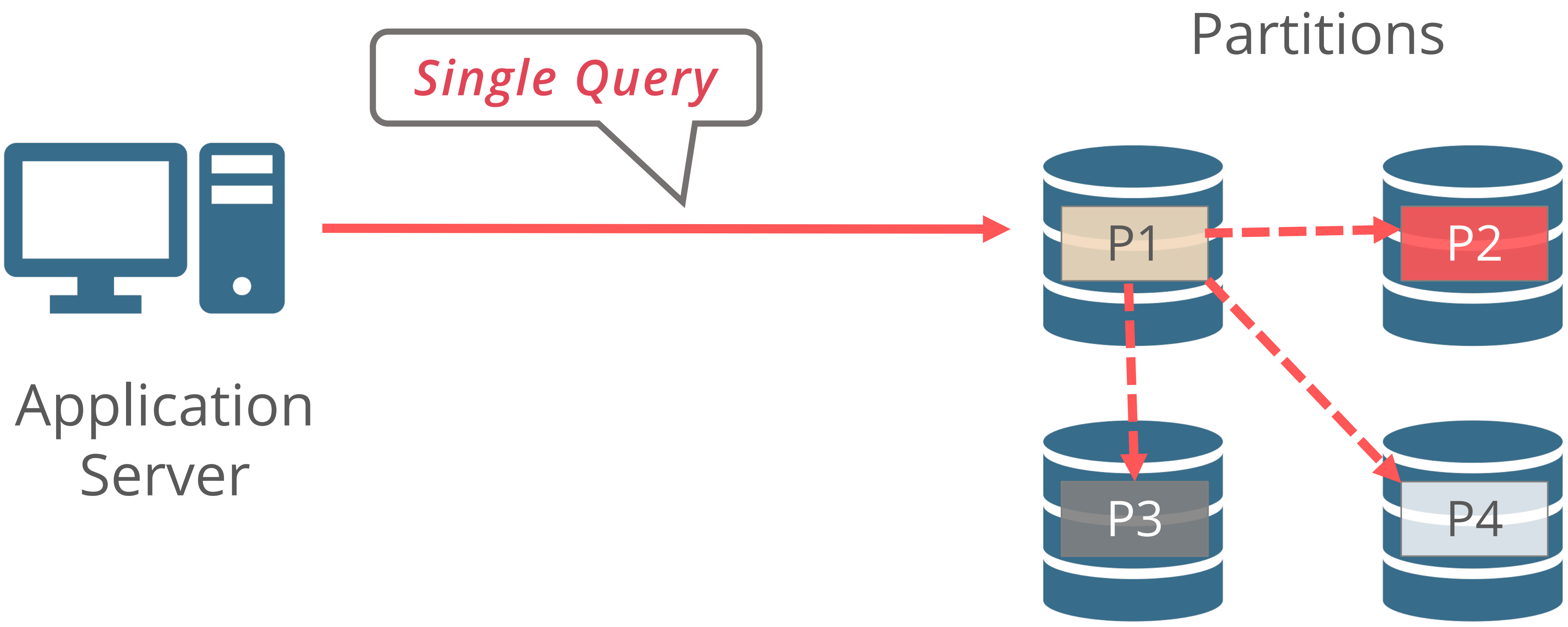
Used for back-end data warehouses

- *Decision support systems:* Applications that serve the management, operations, and planning levels of an organization to help people make decisions about future issues and problems by analysing historical data

Key challenges

- Data movement
- Query planning

PROBLEM SETUP



PUSH VS. PULL

Approach #1: Push Query to Data

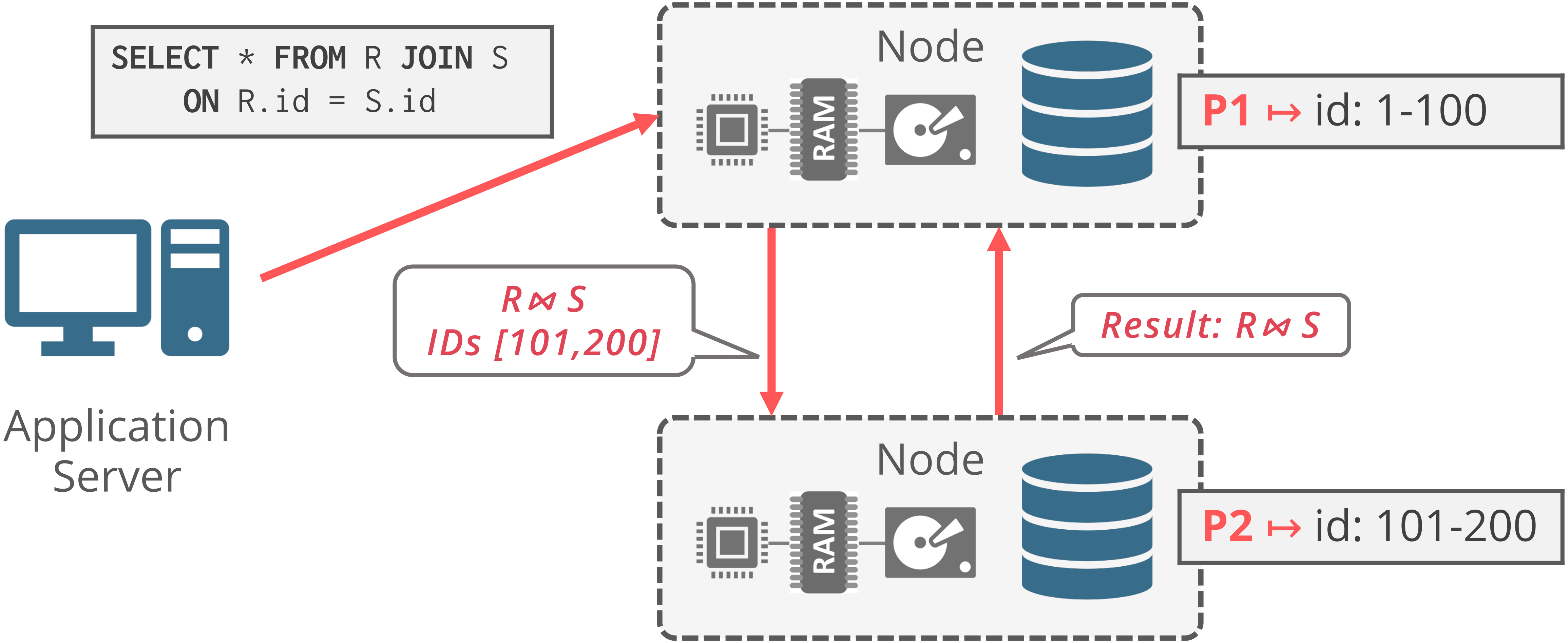
Send the query (or a portion of it) to the node that contains the data

Perform as much filtering and processing as possible where data resides before transmitting over network

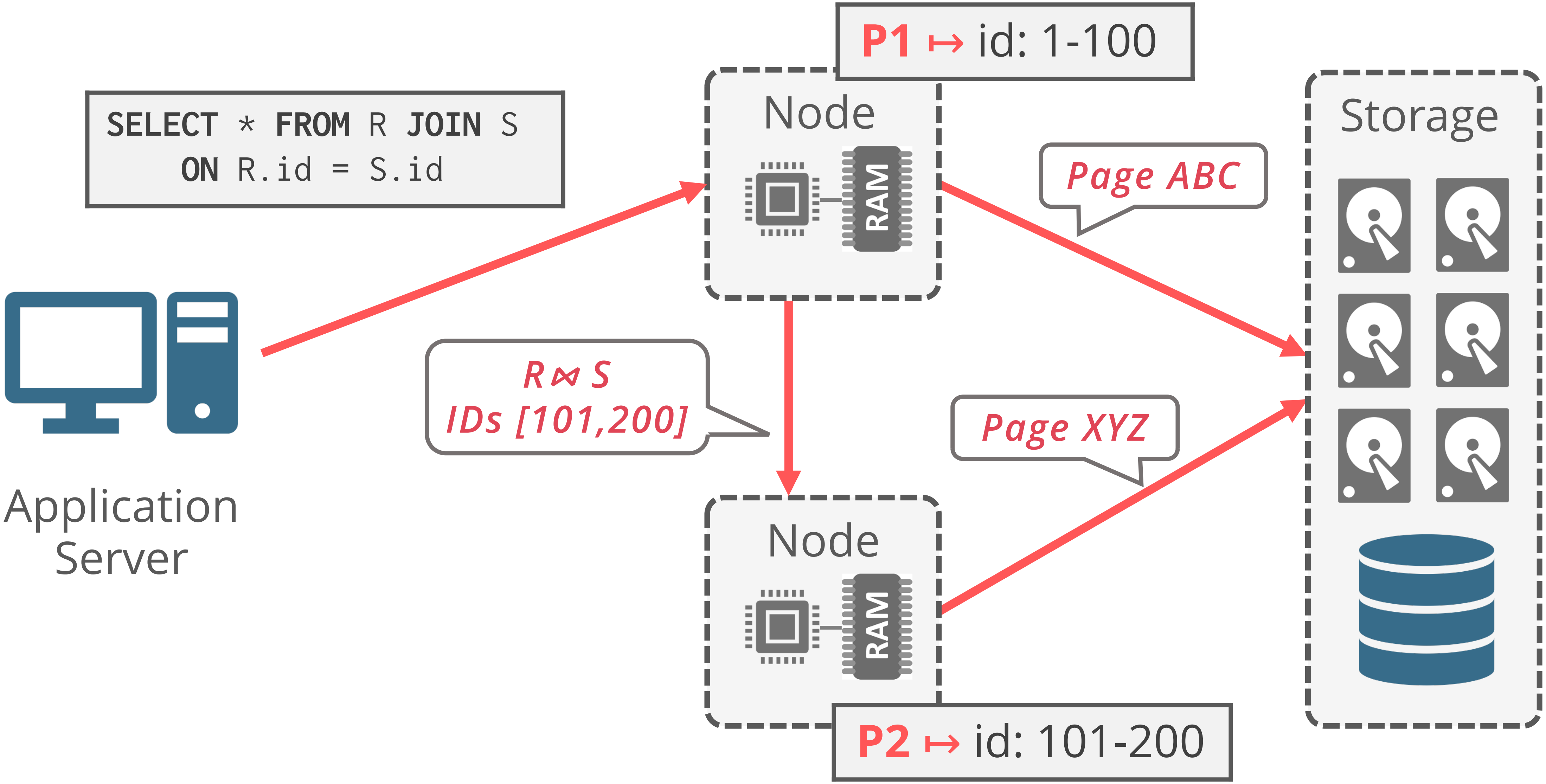
Approach #2: Pull Data to Query

Bring the data to the node that is executing a query that needs it for processing

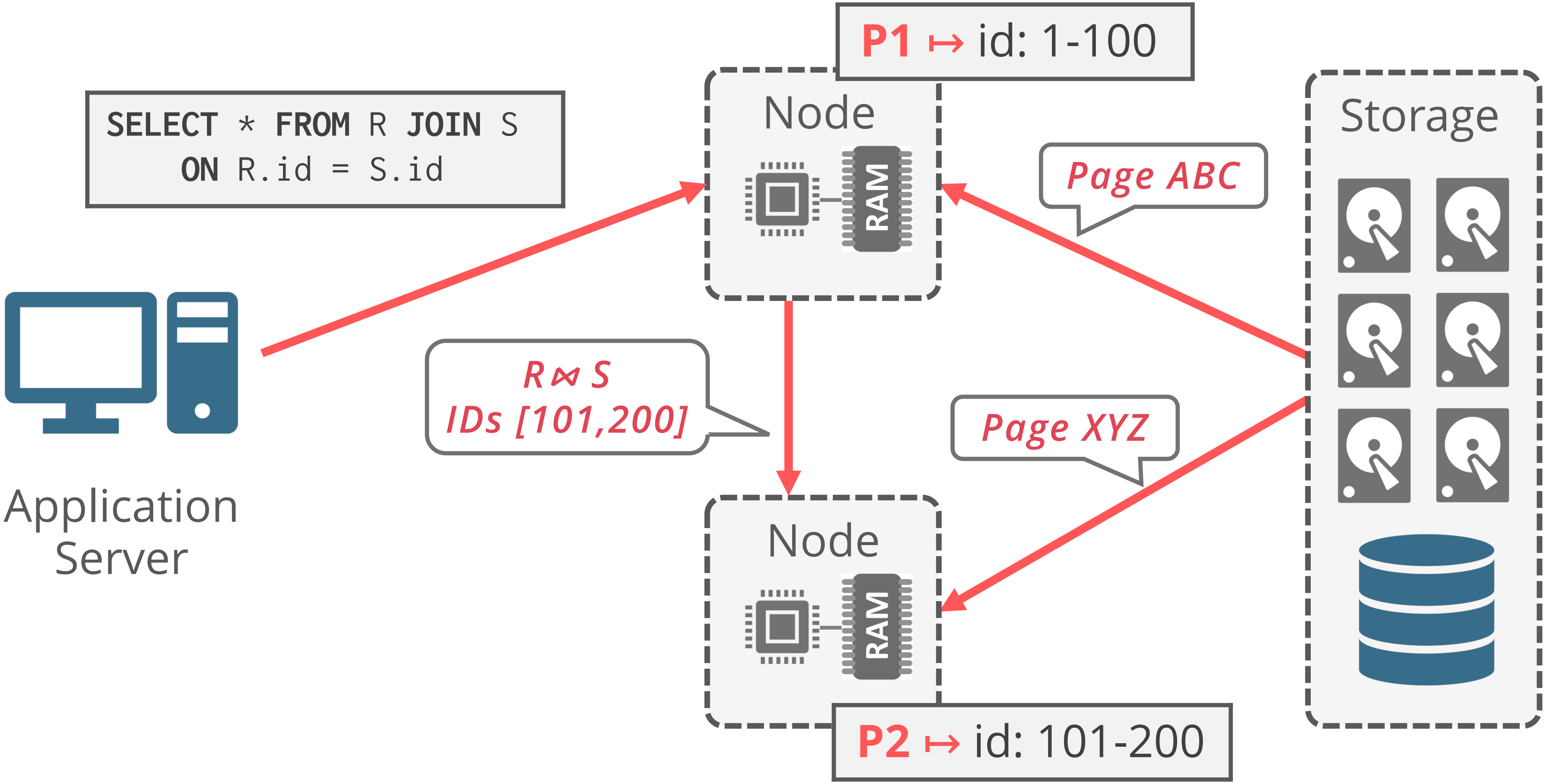
Push Query to Data



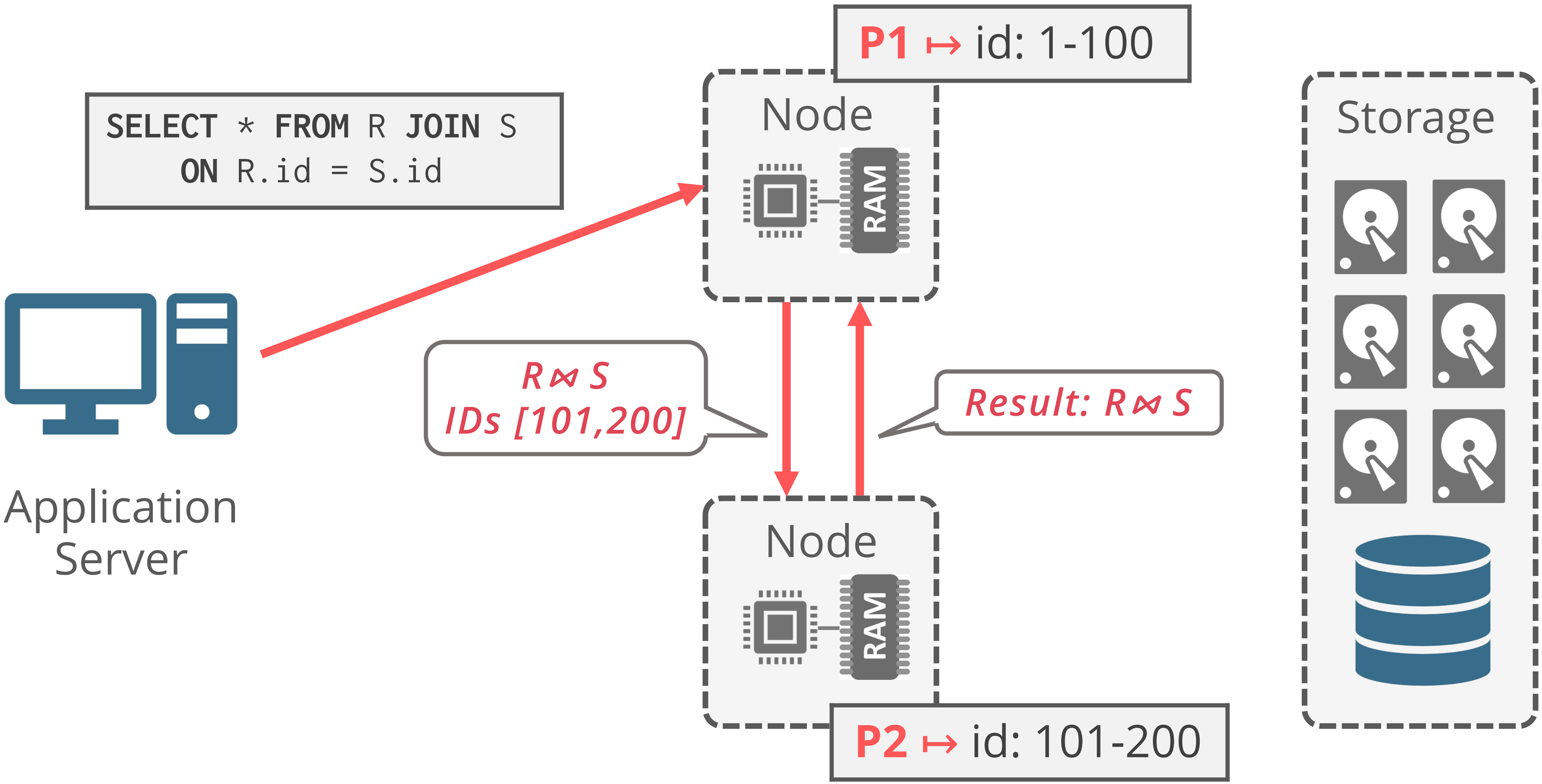
PULL DATA TO QUERY



PULL DATA TO QUERY



PULL DATA TO QUERY



Query Planning

All the optimizations that we talked about before are still applicable in a distributed environment

- Predicate Pushdown
- Early Projections
- Optimal Join Orderings

But now the DBMS must also consider the location of data at each partition when optimizing

Query Plan Fragments

Approach #1: Physical Operators

Generate a single query plan and then break it up into partition specific fragments

Most systems implement this approach

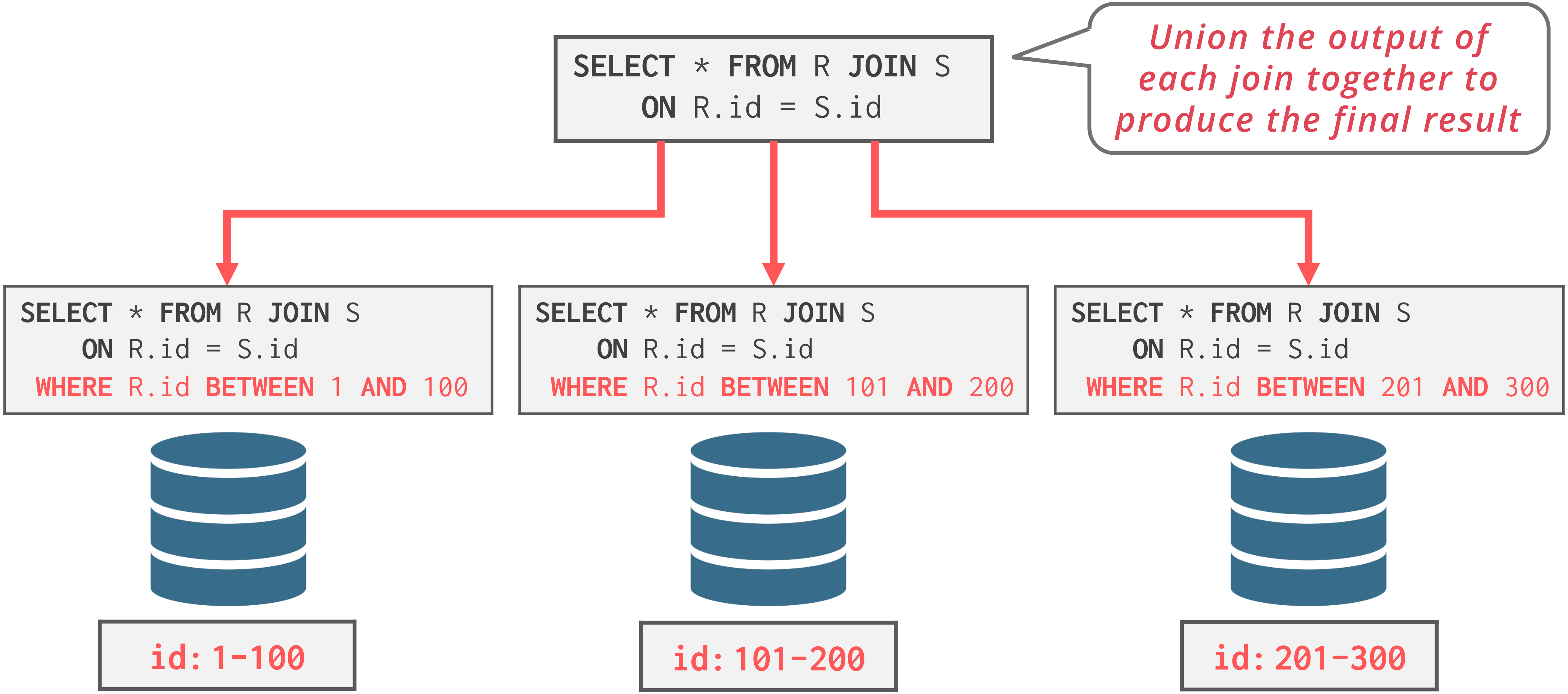
Approach #2: SQL

Rewrite original query into partition specific queries

Allows for local optimization at each node

MemSQL implements this approach

Query Plan Fragments



OBSERVATION

The efficiency of a distributed join depends on the target tables' partitioning schemes

One approach is to put entire tables on a single node and then perform the join

- You lose the parallelism of a distributed DBMS
- Costly data transfer over the network

Distributed Join Algorithms

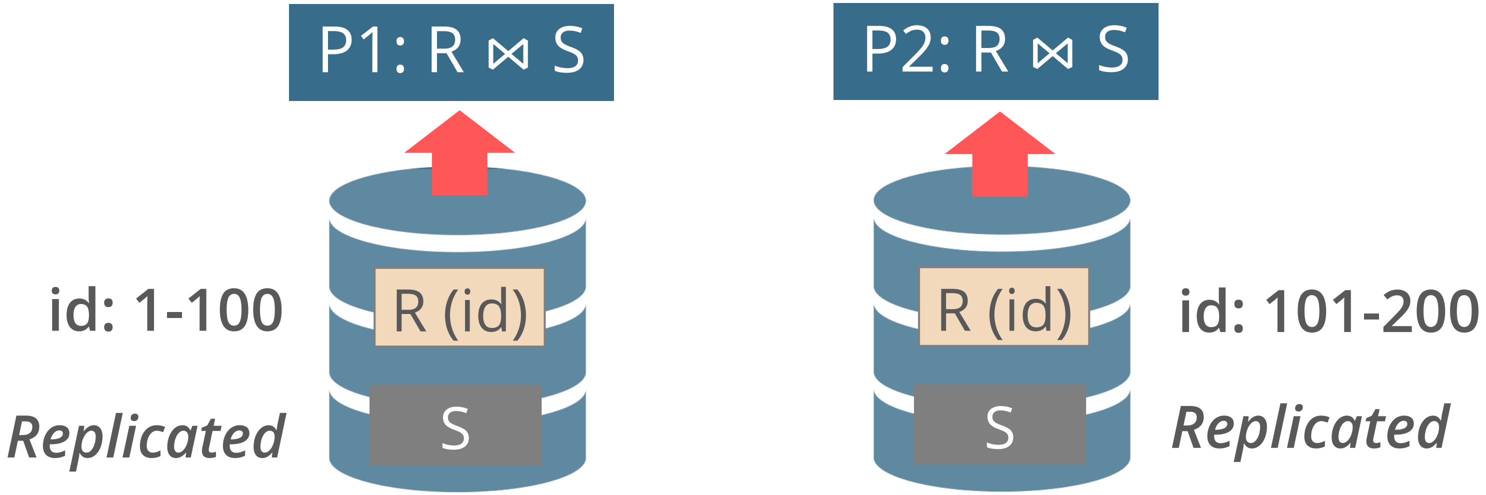
To join tables R and S, the DBMS needs to get the proper tuples on the same node

Once there, it then executes the same join algorithms that we discussed earlier

Scenario #1

One table is replicated at every node.
Each node joins its local data and then sends their results to a coordinating node.

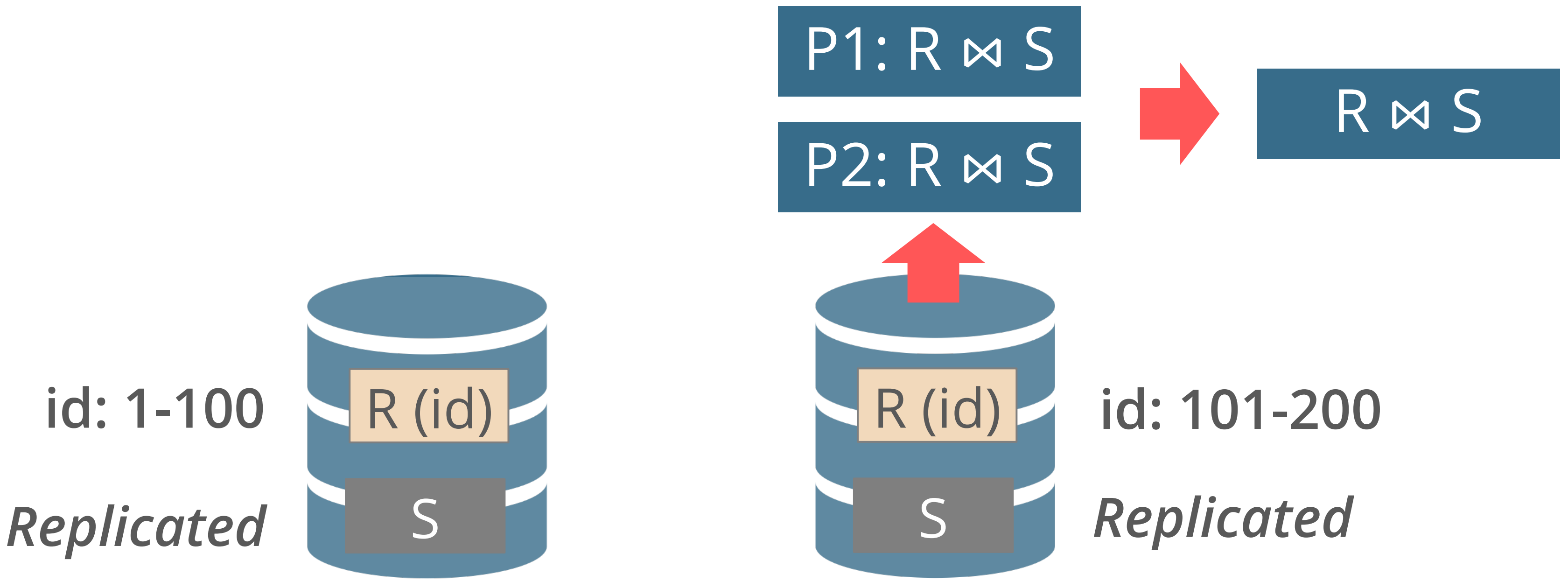
```
SELECT * FROM R JOIN S
    ON R.id = S.id
```



SCENARIO #1

One table is replicated at every node.
Each node joins its local data and then
sends their results to a coordinating node.

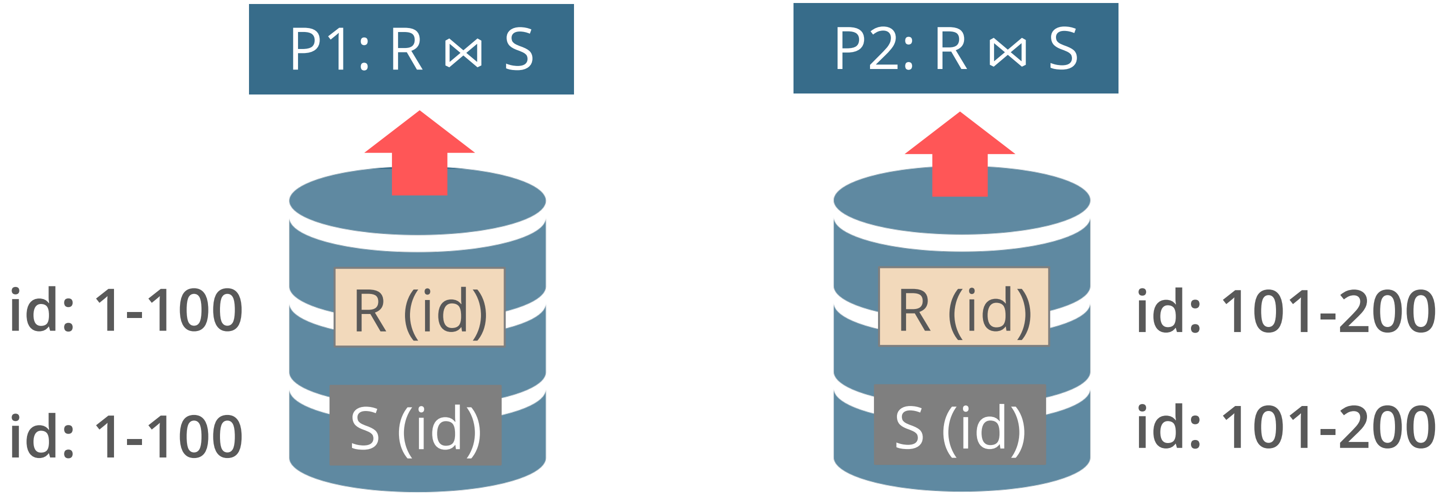
```
SELECT * FROM R JOIN S
    ON R.id = S.id
```



SCENARIO #2

Tables are partitioned on the join attribute. Each node performs the join on local data and then sends to a node for coalescing.

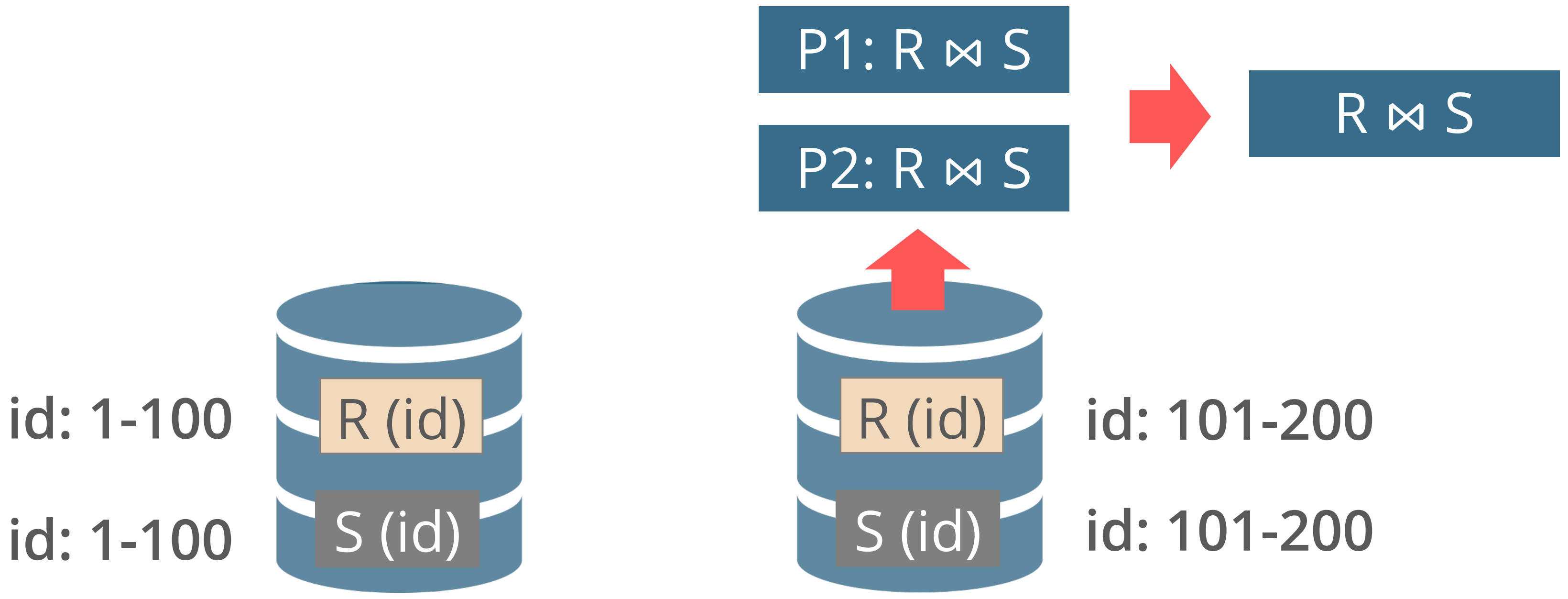
```
SELECT * FROM R JOIN S
    ON R.id = S.id
```



SCENARIO #2

Tables are partitioned on the join attribute. Each node performs the join on local data and then sends to a node for coalescing.

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



SCENARIO #3

Both tables are partitioned on different keys.

If one of the tables is small, then the DBMS **broadcasts** that table to all nodes.

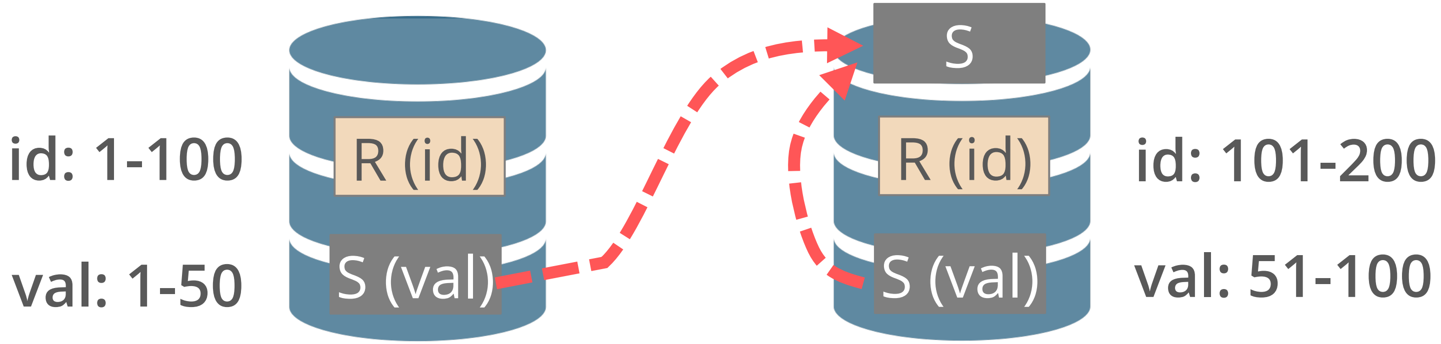
```
SELECT * FROM R JOIN S
    ON R.id = S.id
```

SCENARIO #3

Both tables are partitioned on different keys.

If one of the tables is small, then the DBMS <u>**broadcasts**</u> that table to all nodes.

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

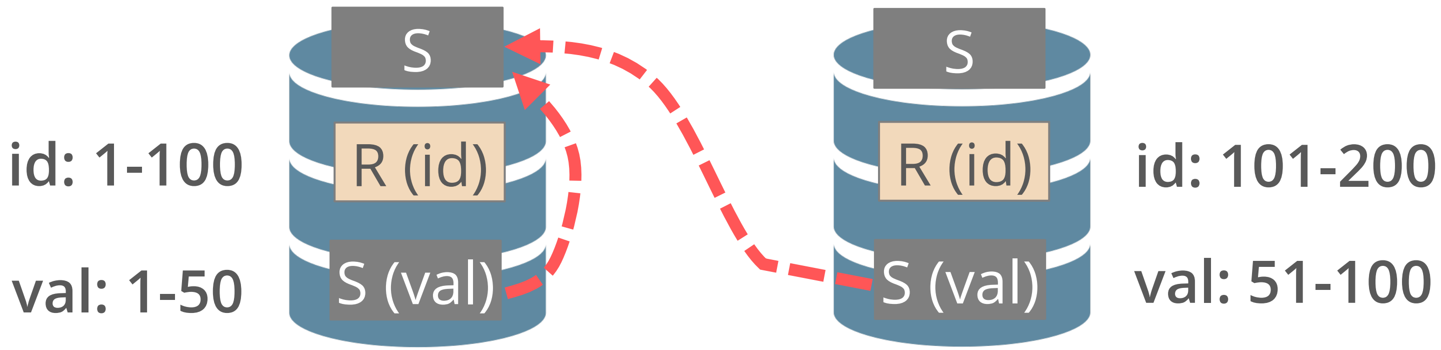


SCENARIO #3

Both tables are partitioned on different keys.

If one of the tables is small, then the DBMS **broadcasts** that table to all nodes.

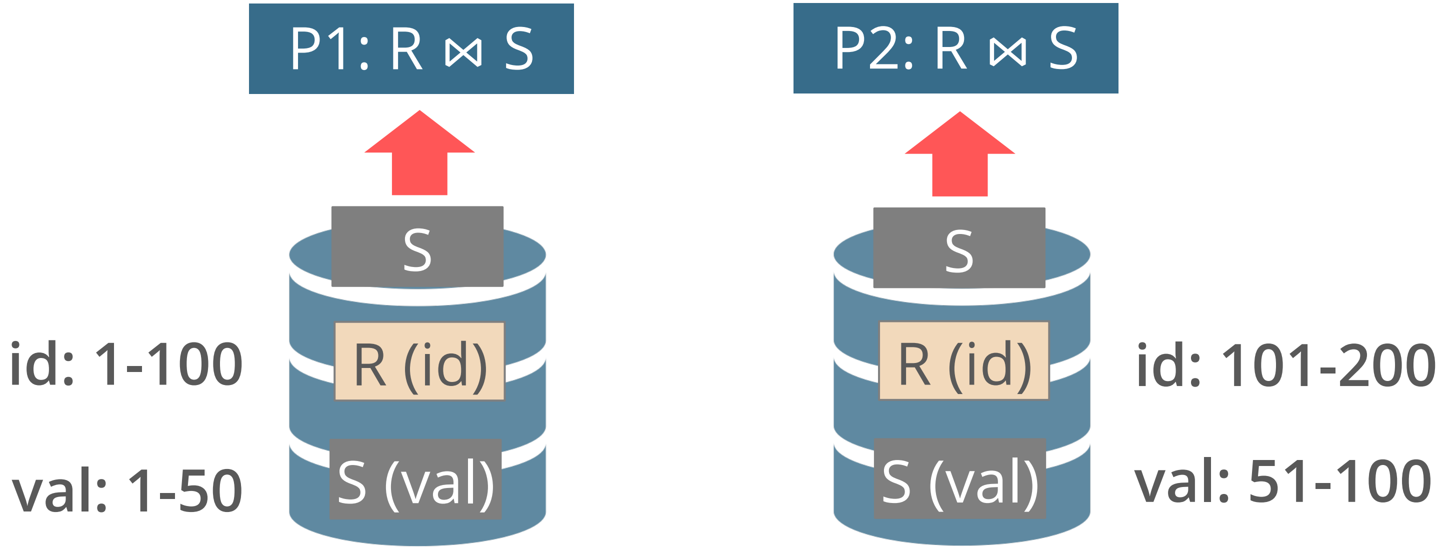
```
SELECT * FROM R JOIN S
    ON R.id = S.id
```



Scenario #3

Both tables are partitioned on different keys. If one of the tables is small, then the DBMS **broadcasts** that table to all nodes.

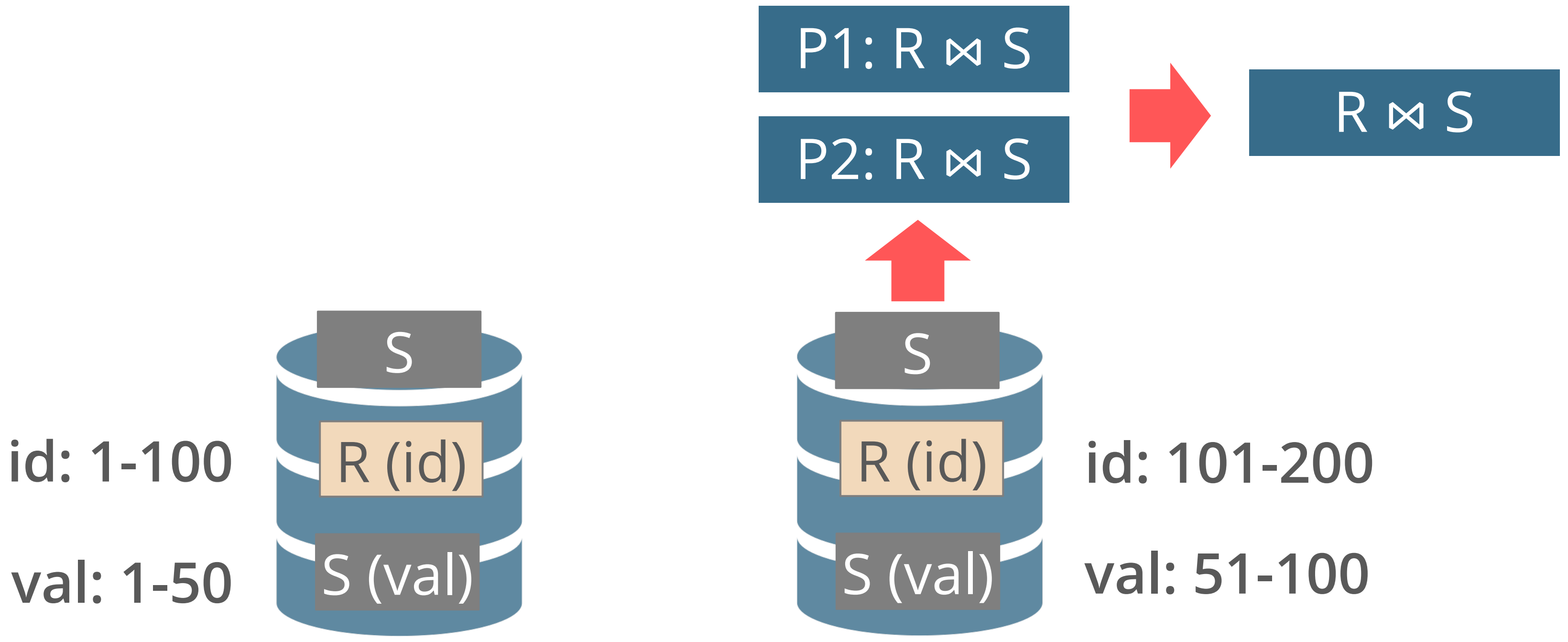
```
SELECT * FROM R JOIN S
    ON R.id = S.id
```



Scenario #3

Both tables are partitioned on different keys. If one of the tables is small, then the DBMS **broadcasts** that table to all nodes.

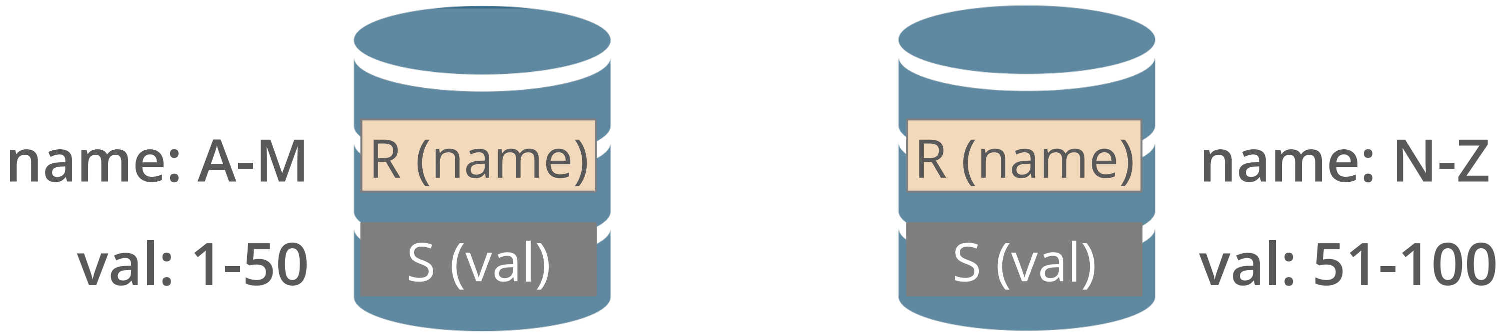
```
SELECT * FROM R JOIN S
    ON R.id = S.id
```



Scenario #4

Both tables are not partitioned on the join key. The DBMS copies the tables by <u>**reshuffling**</u> them across nodes.

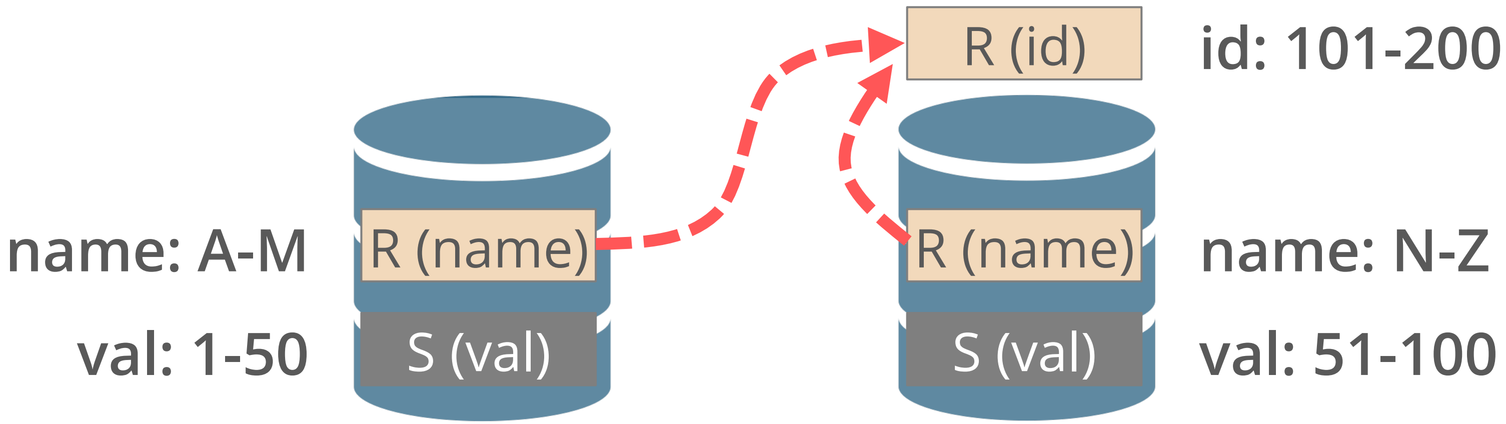
```
SELECT * FROM R JOIN S
    ON R.id = S.id
```



SCENARIO #4

Both tables are not partitioned on the join key. The DBMS copies the tables by **reshuffling** them across nodes.

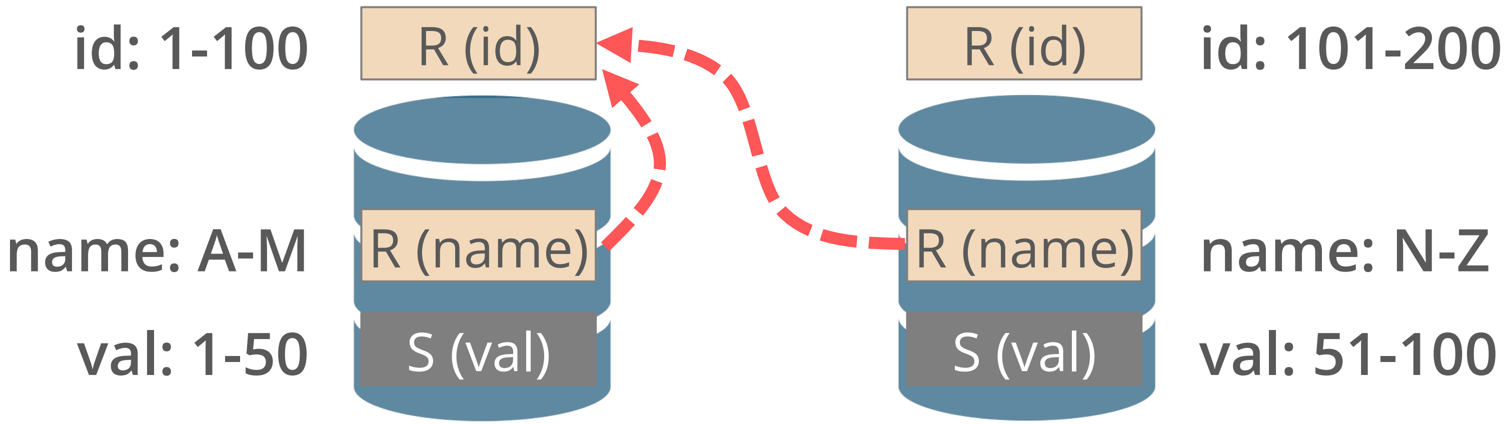
```
SELECT * FROM R JOIN S
  ON R.id = S.id
```



SCENARIO #4

Both tables are not partitioned on the join key. The DBMS copies the tables by **reshuffling** them across nodes.

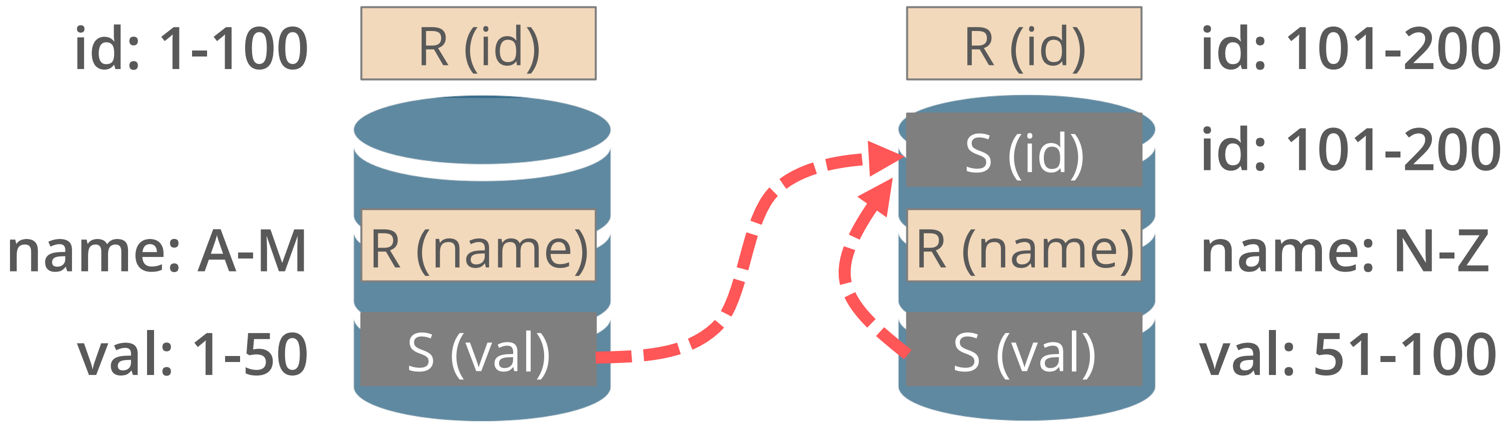
```
SELECT * FROM R JOIN S
    ON R.id = S.id
```



SCENARIO #4

Both tables are not partitioned on the join key. The DBMS copies the tables by **reshuffling** them across nodes.

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

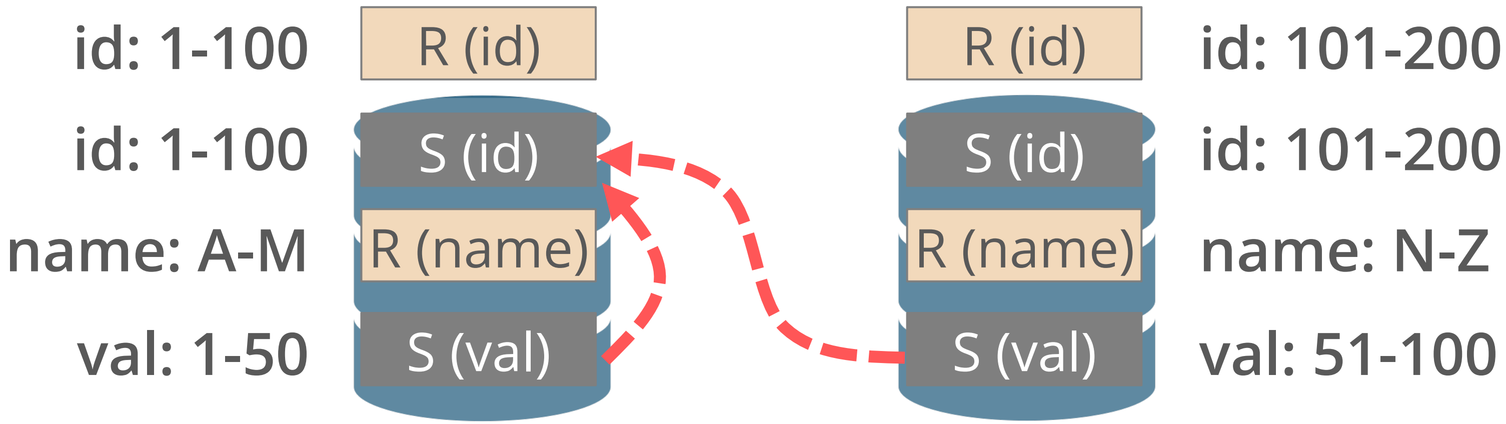


Scenario #4

Both tables are not partitioned on the join key.

The DBMS copies the tables by **reshuffling** them across nodes.

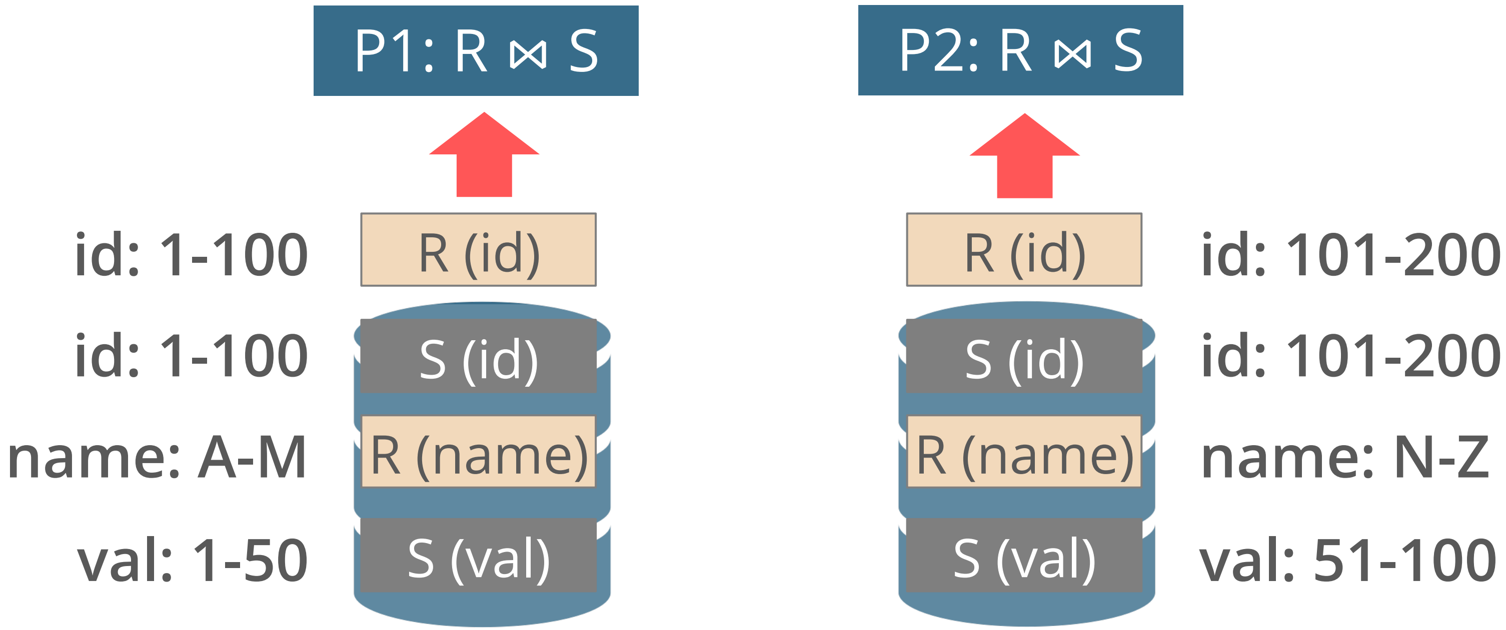
```
SELECT * FROM R JOIN S
    ON R.id = S.id
```



SCENARIO #4

Both tables are not partitioned on the join key. The DBMS copies the tables by **reshuffling** them across nodes.

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

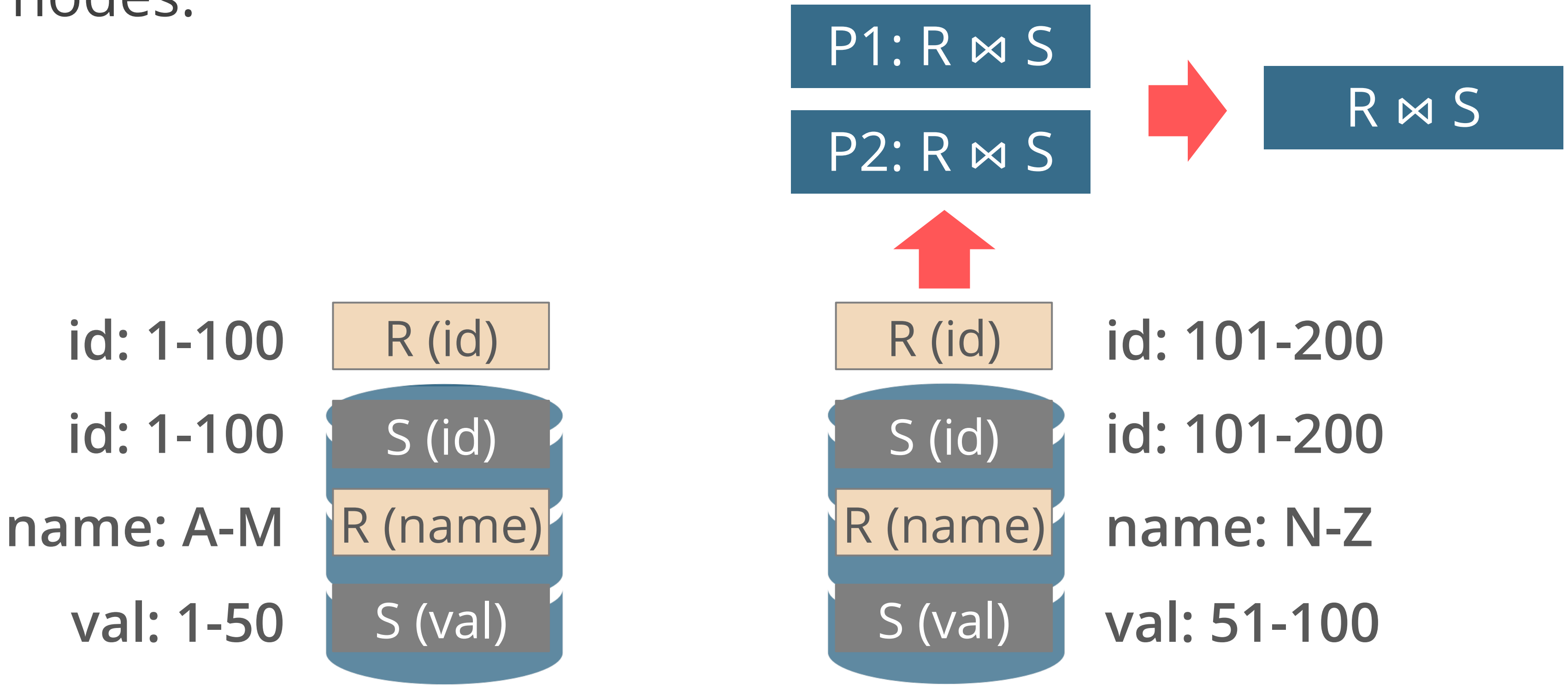


SCENARIO #4

Both tables are not partitioned on the join key.

The DBMS copies the tables by **reshuffling** them across nodes.

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



Conclusion

Efficient distributed OLAP systems are difficult to implement

Everything is harder in a distributed setting

- Concurrency control, query execution, recovery

Distributed transactions access data at one or more partitions

- Require expensive coordination
- Key challenges: consistency and atomicity
- Not enough time to cover in this course