



THE UNIVERSITY
of EDINBURGH

Advanced Database Systems

Spring 2026

Lecture #26:

Parallel Query Processing

R&G: Chapter 22

RECAP: PARALLEL / DISTRIBUTED DBMSs

Why do we need parallel / distributed DBMSs?

- Increased performance (throughput and latency)

- Increased availability

Database is spread out across multiple resources to improve parallelism

Appears as a single database instance to the application

- SQL query on a single-node DBMS must generate same result on a parallel or dist. DBMS

- Due to principle of **data independence**

RECAP: 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

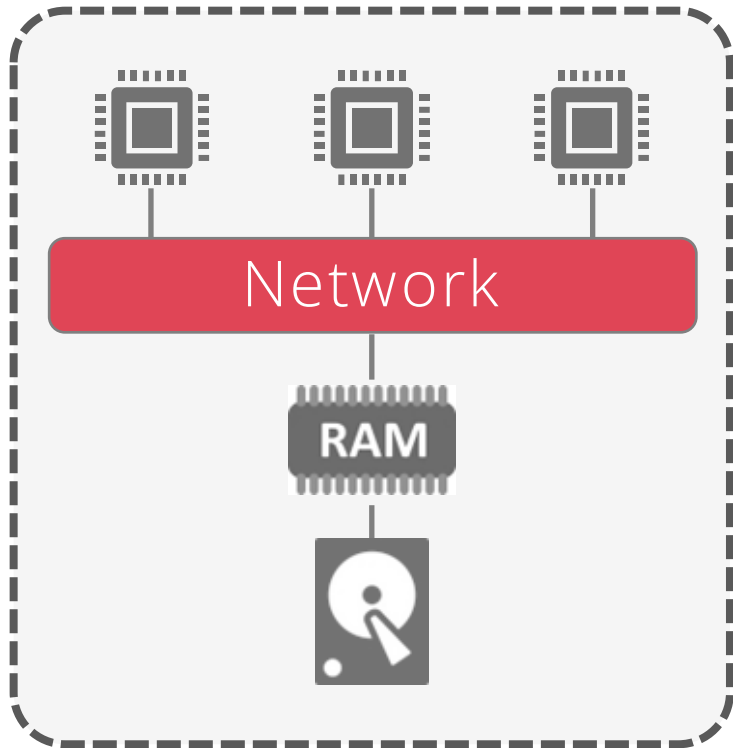
A DBMS's architecture specifies what shared resources are directly accessible to CPUs

The goal is to parallelize operations across multiple resources

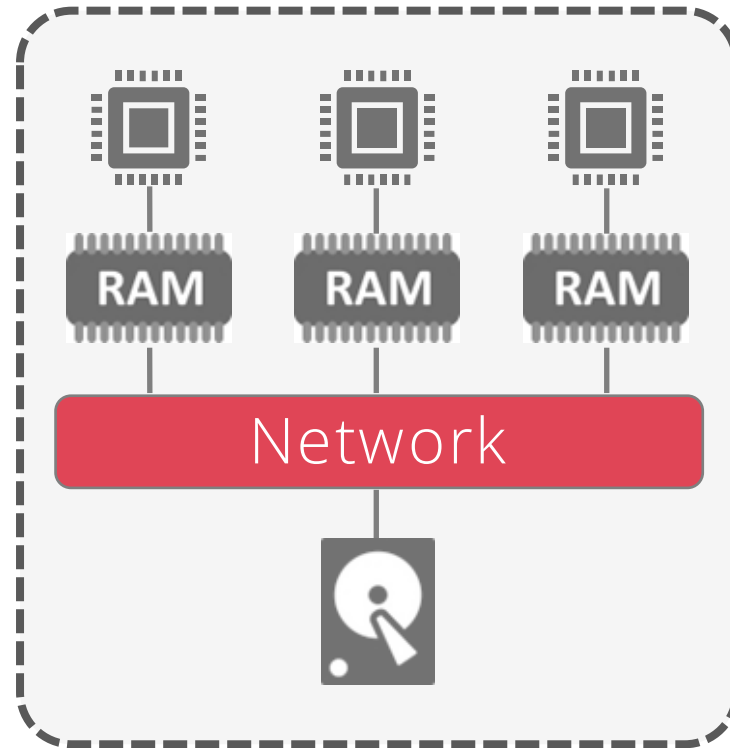
CPU, memory, network, disk

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

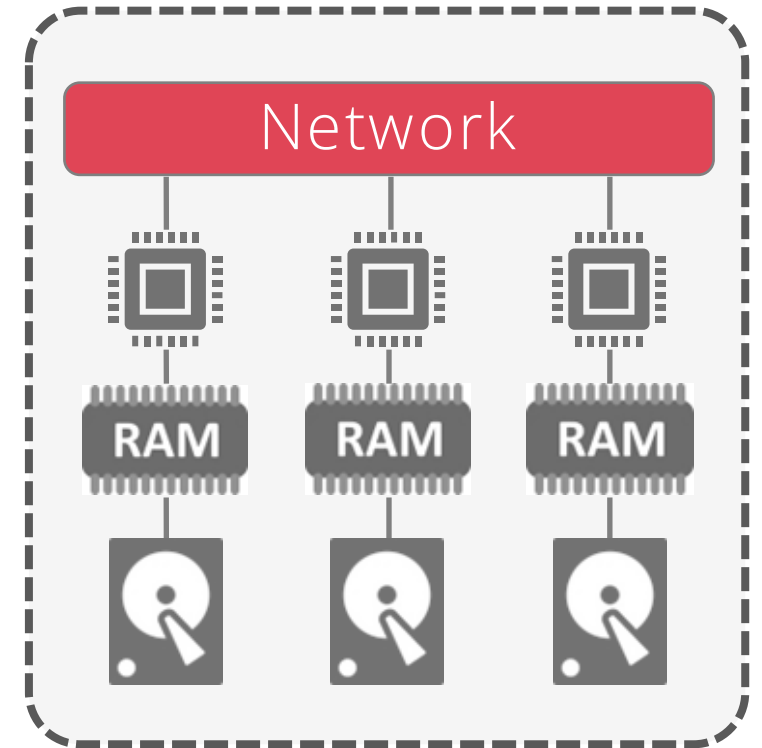
SYSTEM ARCHITECTURE



Shared Memory



Shared Disk



Shared Nothing

SHARED MEMORY

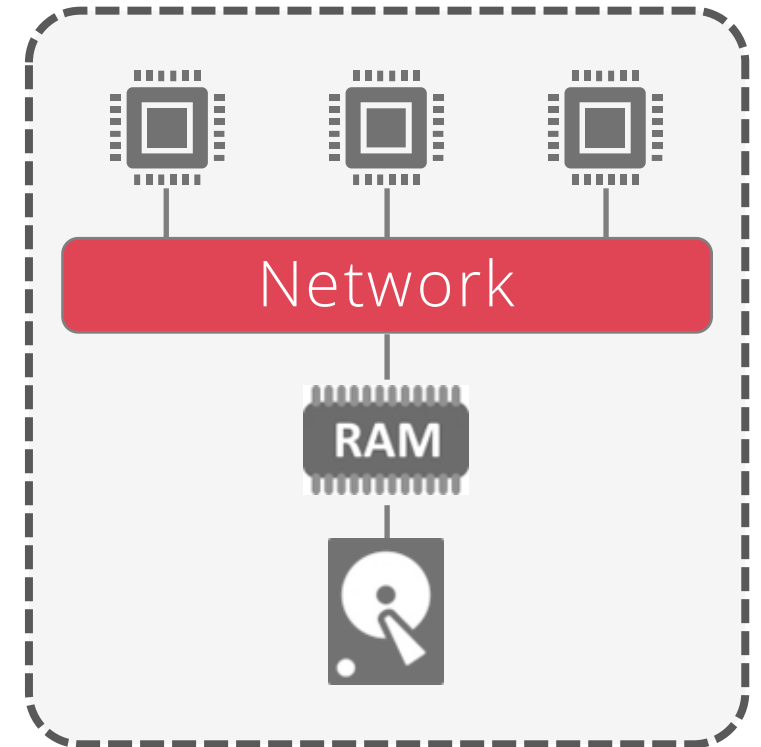
CPU's have access to common memory address space via a fast interconnect

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

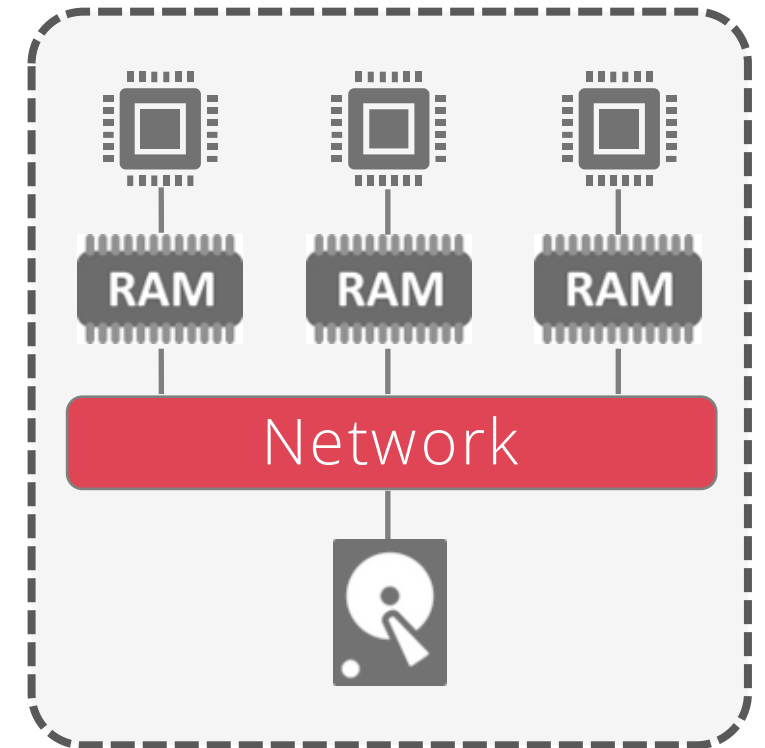
All CPUs can access a single logical disk directly via an interconnect but each CPU has its own private memory

Can scale execution layer independently from the storage layer

Easy consistency since there is a single copy of DB

Easy fault tolerance

The disk becomes a bottleneck with many CPUs



SHARED NOTHING

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

Typically instances run on commodity hardware

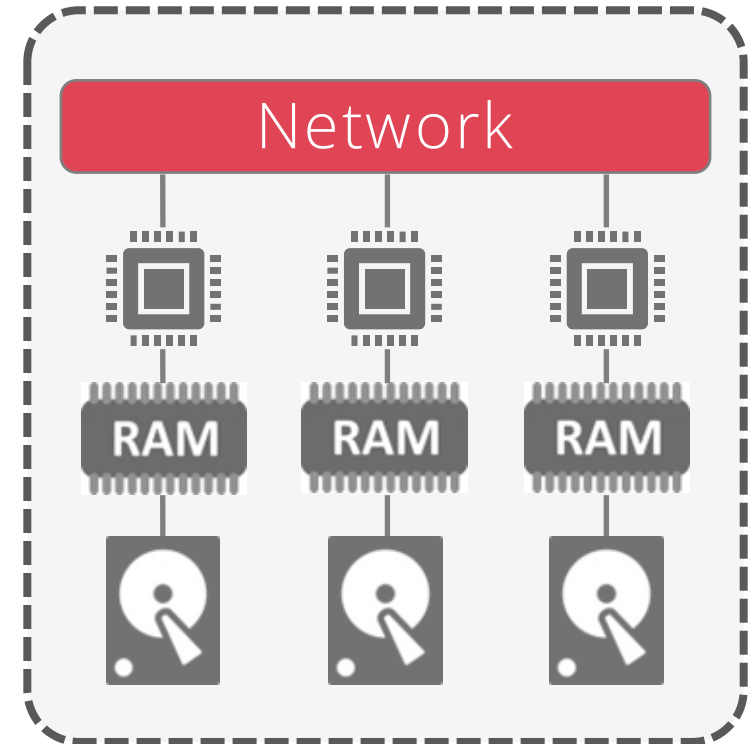
Nodes only communicate with each other via network

Easy to increase capacity

Just keep putting nodes on the network!

Hard to ensure consistency

Nodes need to communicate over the network



TYPES OF PARALLELISM IN DBMSs

Inter-Query: Different queries are executed concurrently

Increases throughput & reduces latency

Does require parallel-aware concurrency control

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

Decreases latency for long-running queries

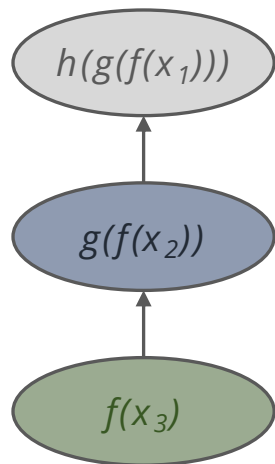
Inter-operator: Execute operators of a query in parallel (exploits pipelining)

Intra-operator: Get all CPUs to compute a given operation (scan, sort, join)

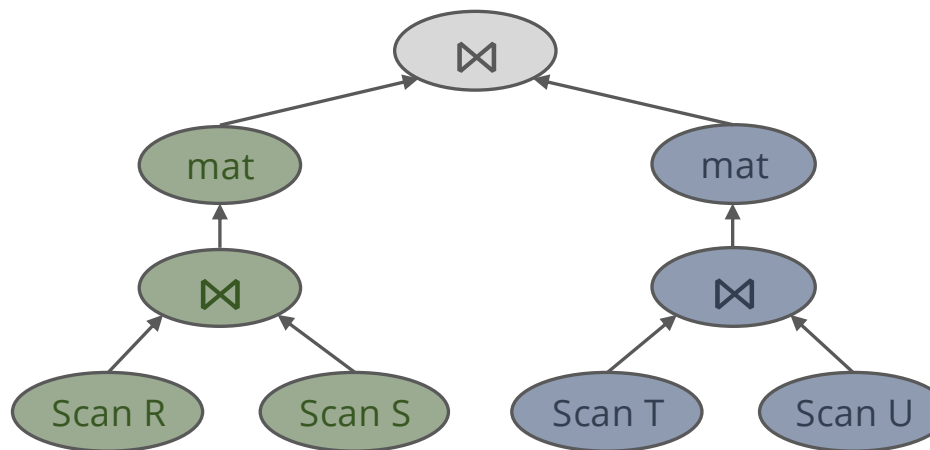
INTRA-QUERY – INTER-OPERATOR

Intra-query (within a single query)

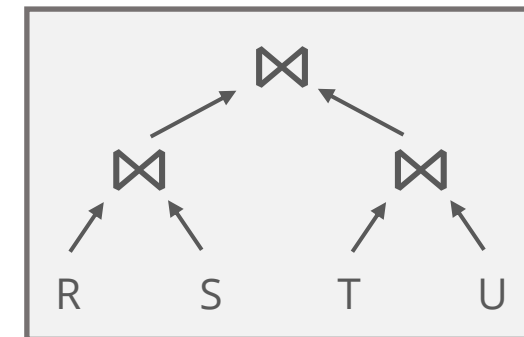
Inter-operator (between operators)



Pipeline Parallelism



Bushy (Tree) Parallelism

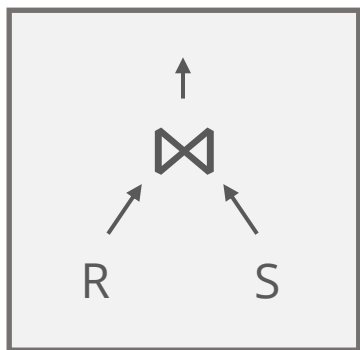


Logical Plan

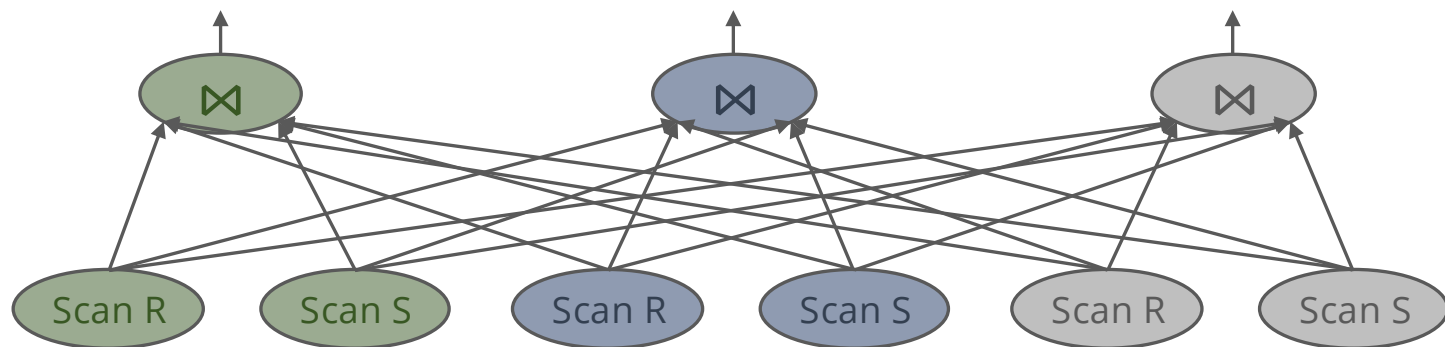
INTRA-QUERY – INTRA-OPERATOR

Intra-query (within a single query)

Intra-operator (within a single operator)



Logical Plan



Partition Parallelism

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

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

ROUND-ROBIN PARTITIONING

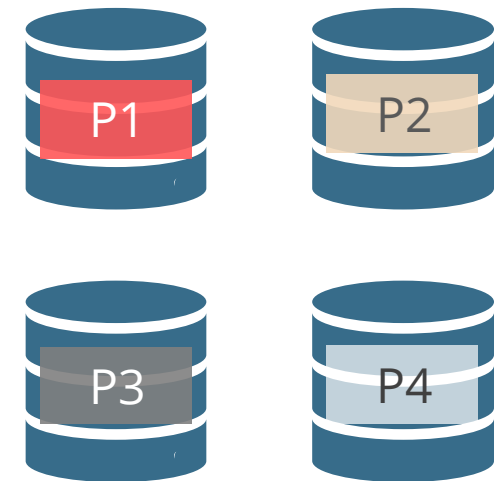
Distribute tuples to partitions in a round-robin fashion

Good for spreading load

Table

101	a	XXY	2019-11-29	P1
102	b	XYX	2019-11-28	P2
103	c	YXX	2019-11-29	P3
104	d	XYY	2019-11-27	P4
105	e	YXY	2019-11-29	P1

Partitions



HASH 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

$\text{hash}(a) \% 4 = P2$

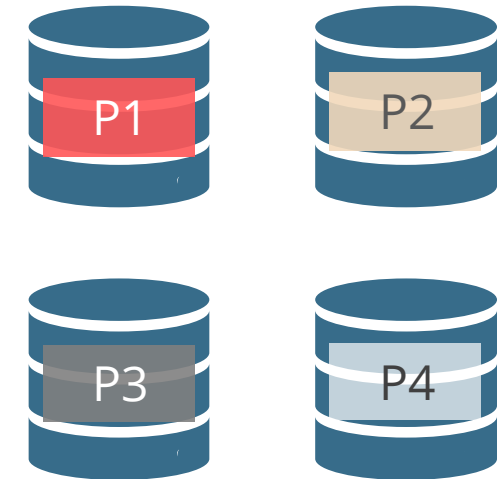
$\text{hash}(b) \% 4 = P4$

$\text{hash}(c) \% 4 = P3$

$\text{hash}(d) \% 4 = P2$

$\text{hash}(e) \% 4 = P1$

Partitions



Ideal Query:

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

Also good for equijoins, group-by

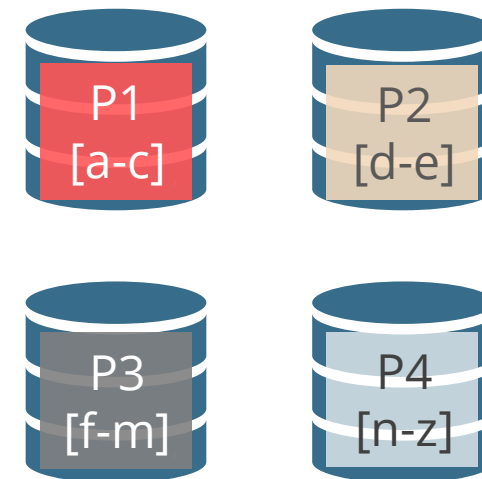
RANGE PARTITIONING

Partition Key

Table

101	a	XXY	2019-11-29	P1
102	b	XYX	2019-11-28	P1
103	c	YXX	2019-11-29	P1
104	d	XYY	2019-11-27	P2
105	e	YXY	2019-11-29	P2

Partitions



Ideal Query:

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

Also good for equijoins, group-by, range queries

REPLICATION

The DBMS can replicate data across nodes to increase availability

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

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

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

INTRA-OPERATOR PARALLELISM

PARALLEL SCANS

Scan in parallel, merge (concat) output

Ex: Sequential scan of 100TB at 0.5 GB/sec takes ~200,000 sec = ~2.31 days

But 100-way parallel scan takes only 2,000 sec = 33 minutes

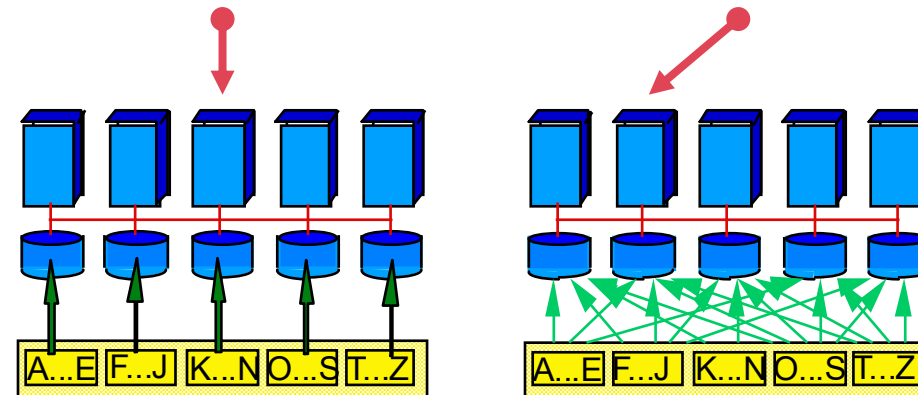
σ_p : skip entire sites that have no tuples satisfying p

Possible with range or hash partitioning

LOOKUP BY KEY

Data partitioned on function of key?

Great! Route lookup only to relevant node



Range

Hash

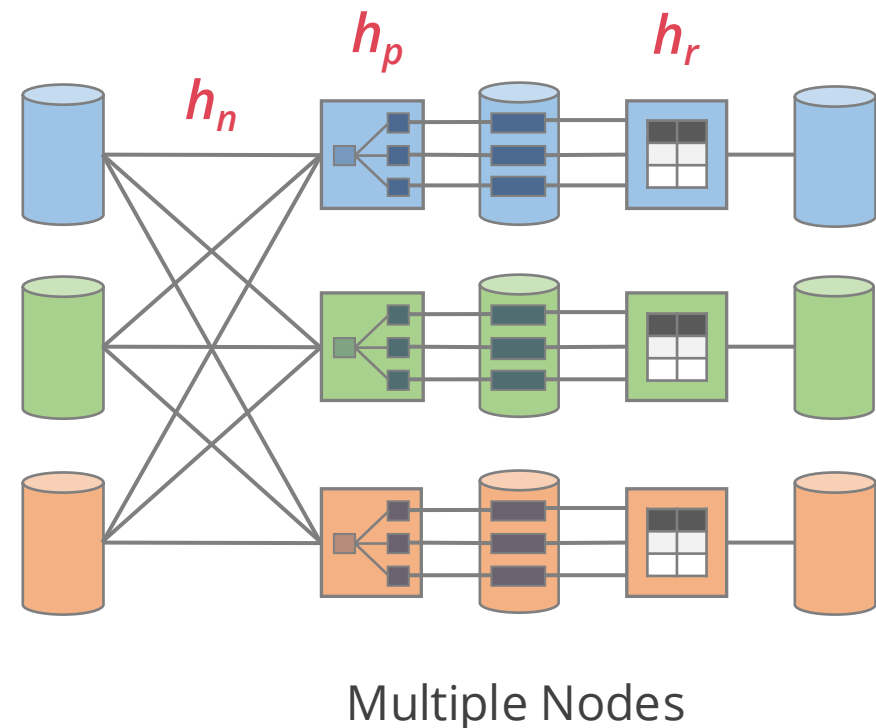
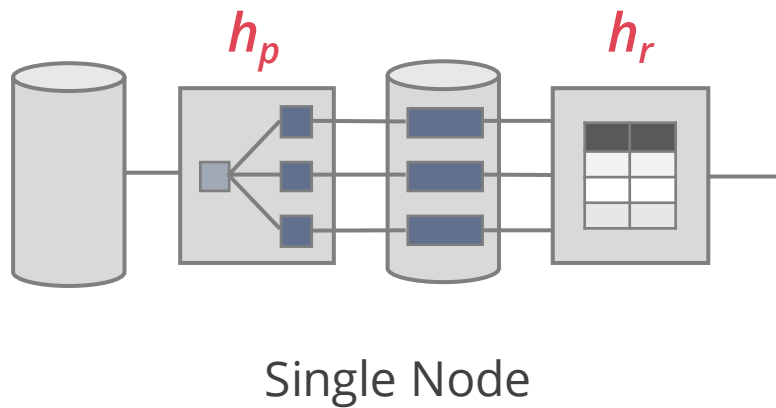
Otherwise

Must broadcast lookup request to **all** nodes

PARALLEL HASHING

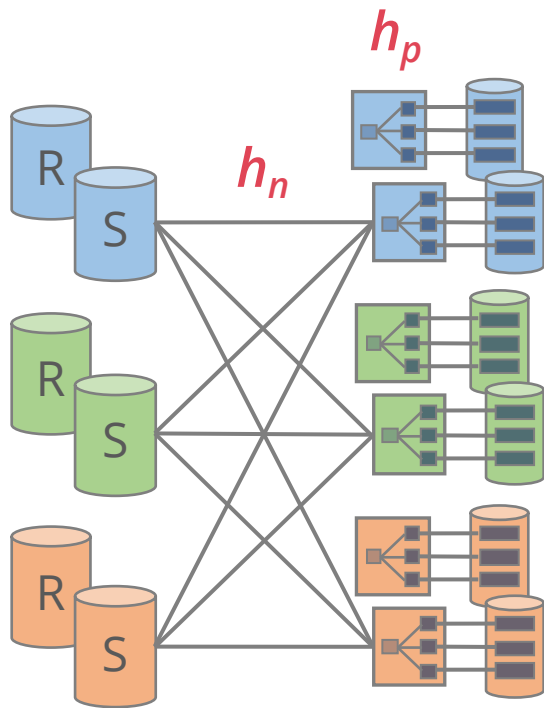
Use a hash function h_n to partition the data over all the nodes (hash partitioning), then run external hashing on each node independently

Similar to recursive partitioning

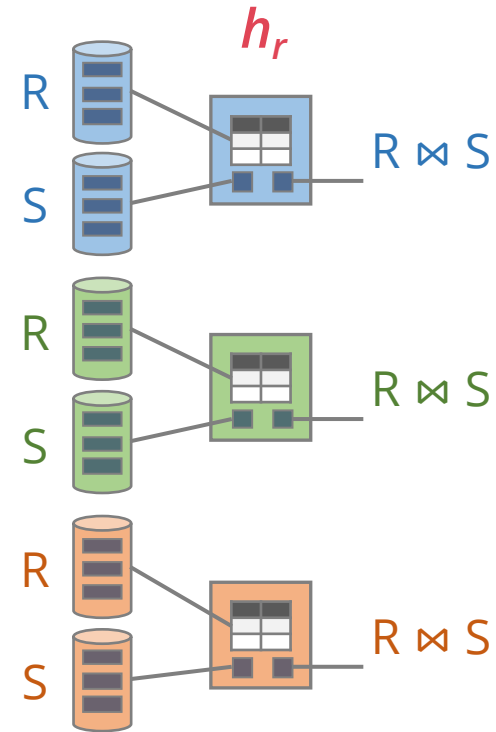


PARALLEL HASH JOIN

Hash partition both relations on the join key, then perform a normal hash join on each node independently



Phase 1: Partition R and S across different nodes

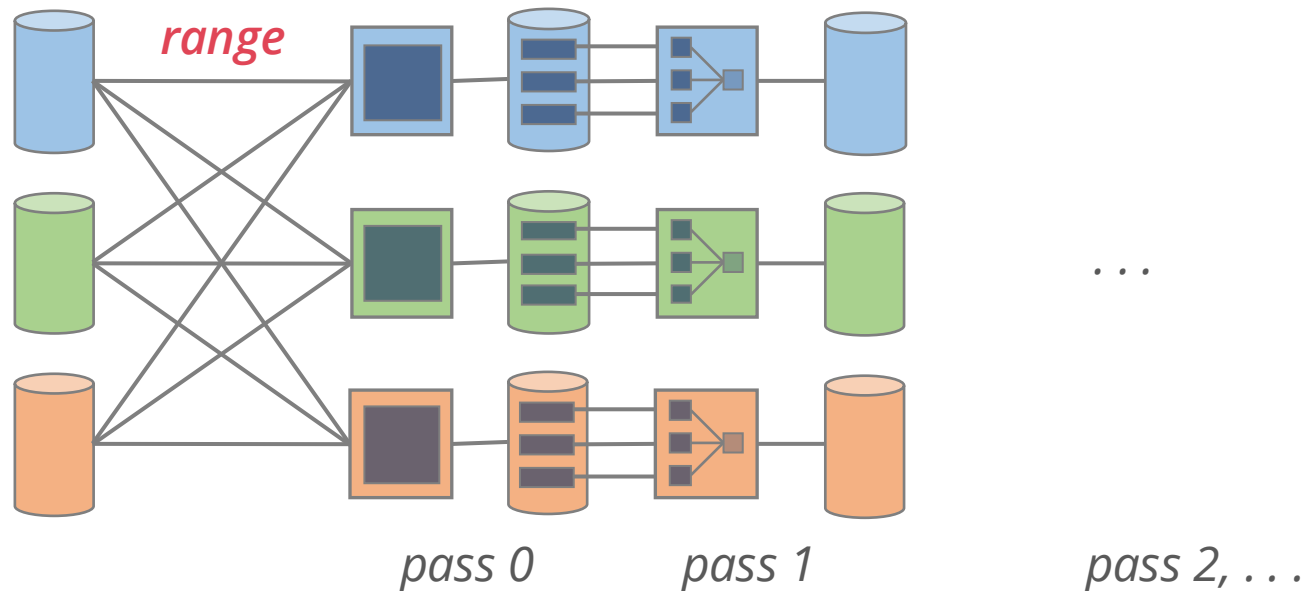


Phase 2: Within each node, perform local (grace) hash joins

PARALLEL SORTING

Partition the data over machines with **range partitioning**

Perform external sorting on each machine independently
(each machine holds a different range of data)

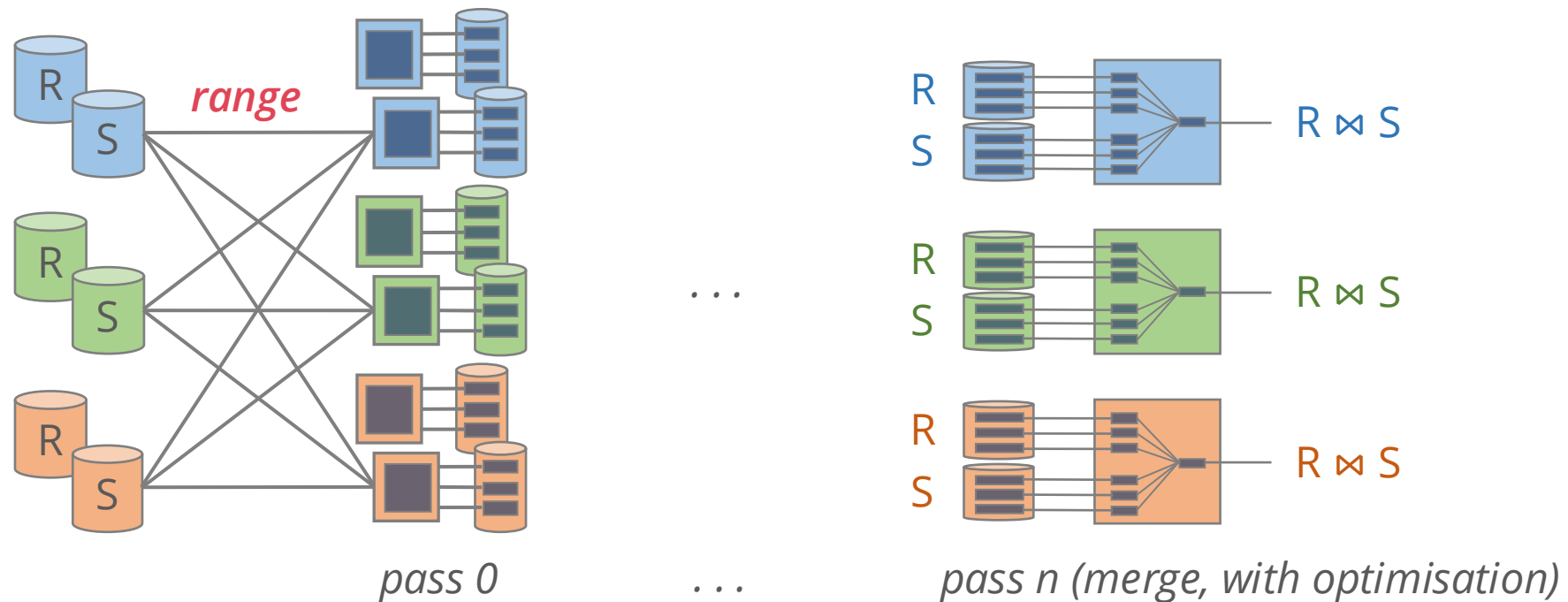


PARALLEL SORT MERGE JOIN

Range partition both relations on the join key over machines

Use the same ranges for both relations

Perform sort merge join on each machine independently



OBSERVATION

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

Naïve approach puts entire tables on a single node, then performs the join

- You lose the parallelism of a distributed DBMS

- Costly data transfer over the network

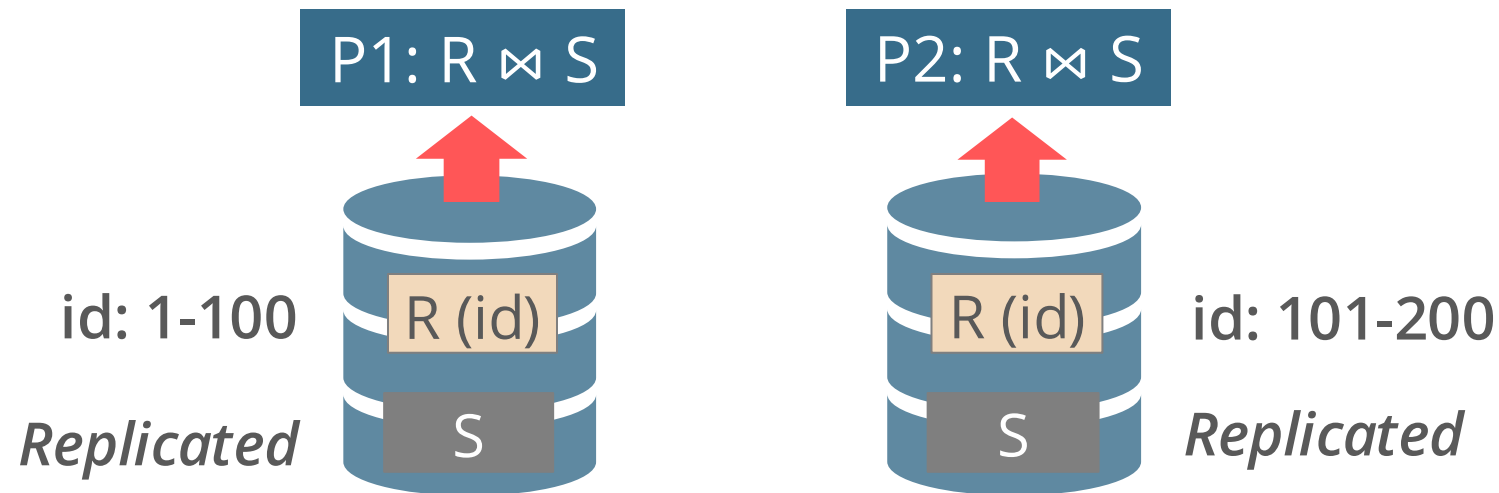
To join **R** and **S**, the DBMS needs to get matching tuples on the same node

- 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

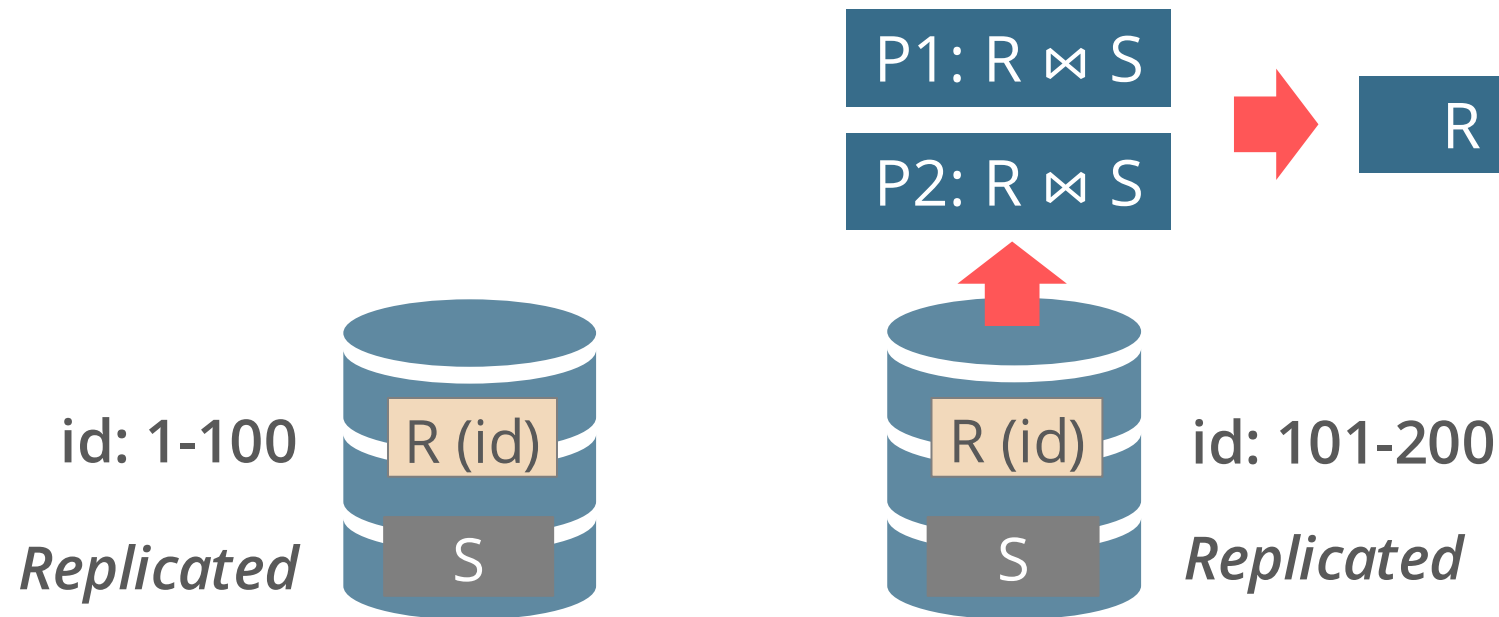
```
SELECT * FROM R JOIN S  
ON R.id = S.id
```



SCENARIO #1

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

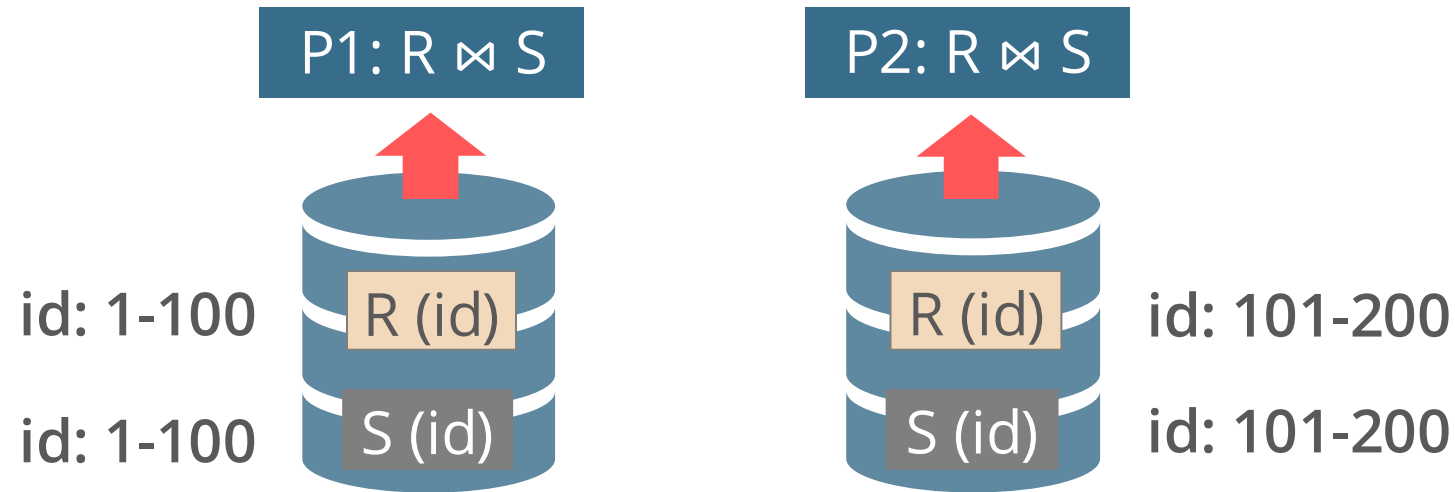
```
SELECT * FROM R JOIN S  
ON R.id = S.id
```



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

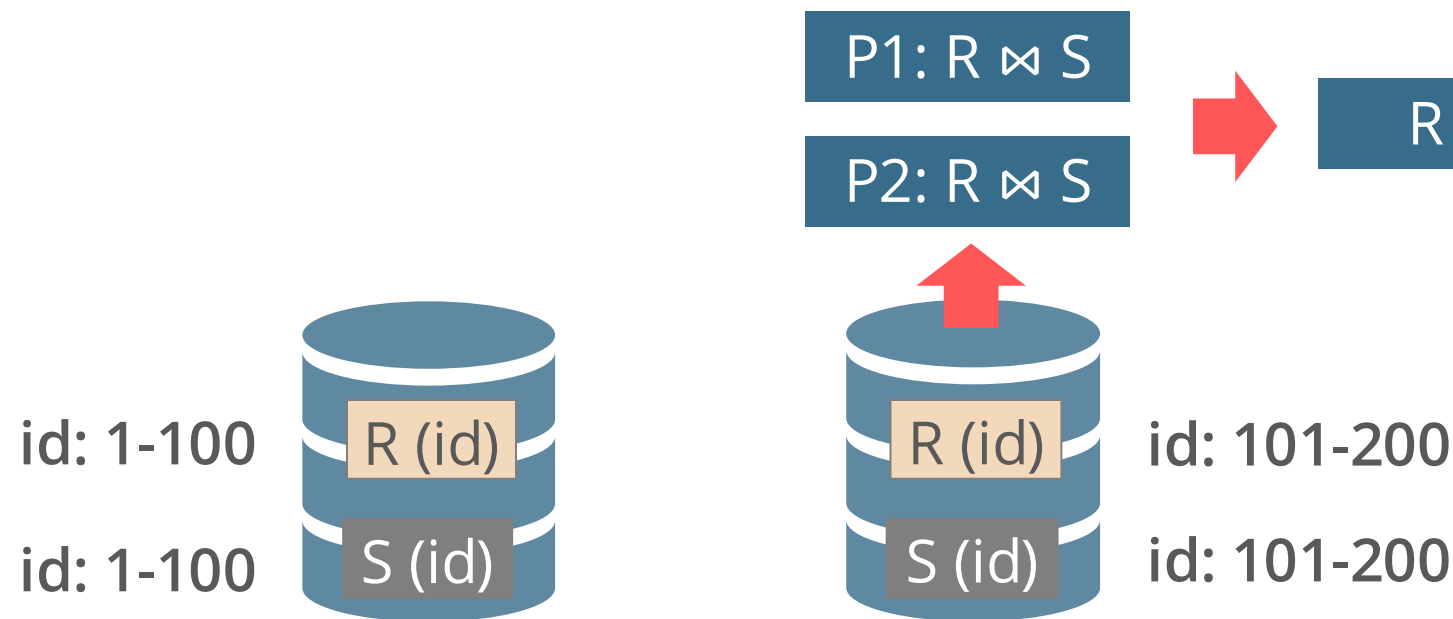
```
SELECT * FROM R JOIN S  
ON R.id = S.id
```



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

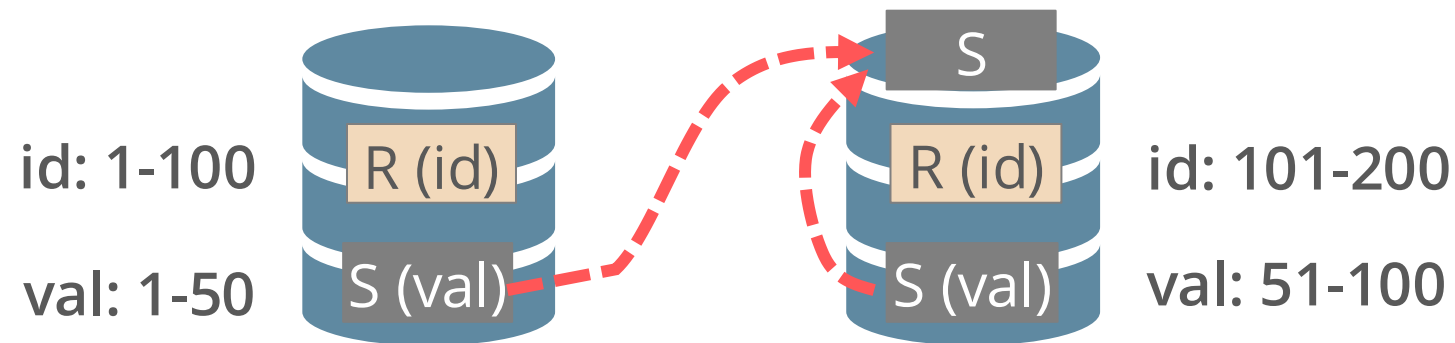
```
SELECT * FROM R JOIN S  
ON R.id = S.id
```



SCENARIO #3

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

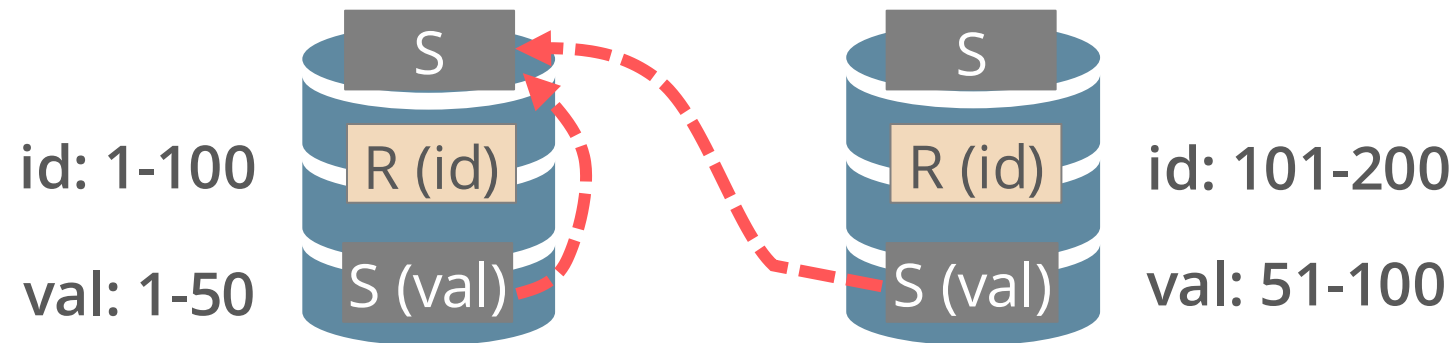
```
SELECT * FROM R JOIN S  
ON R.id = S.id
```



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

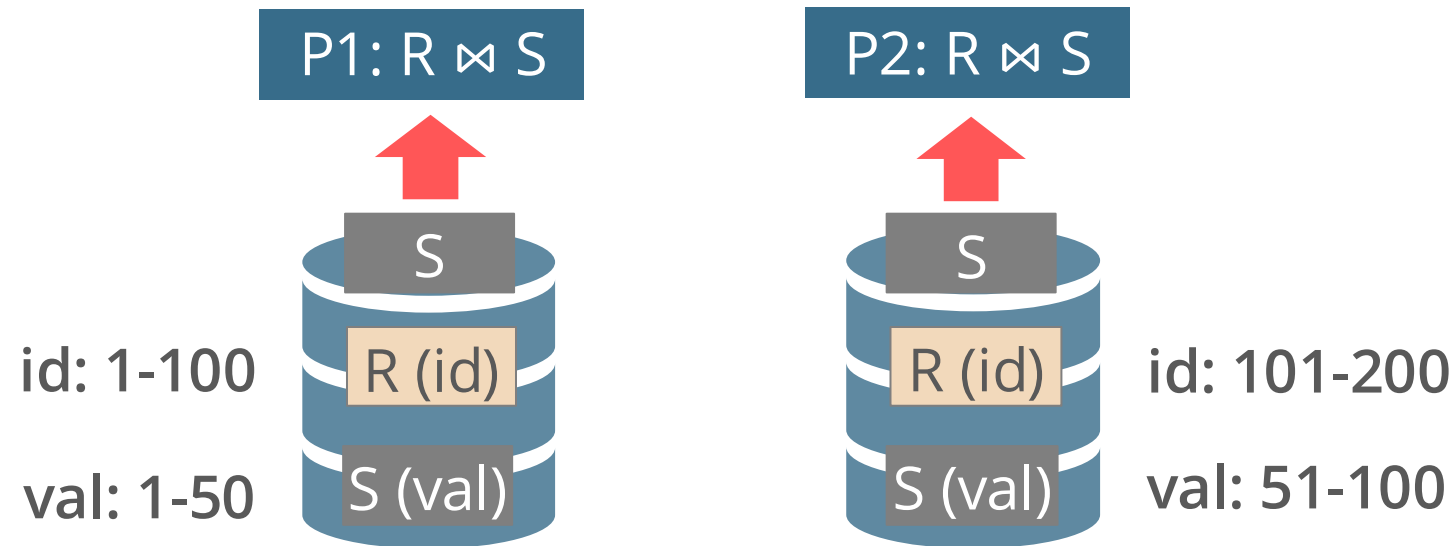
```
SELECT * FROM R JOIN S  
ON R.id = S.id
```



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

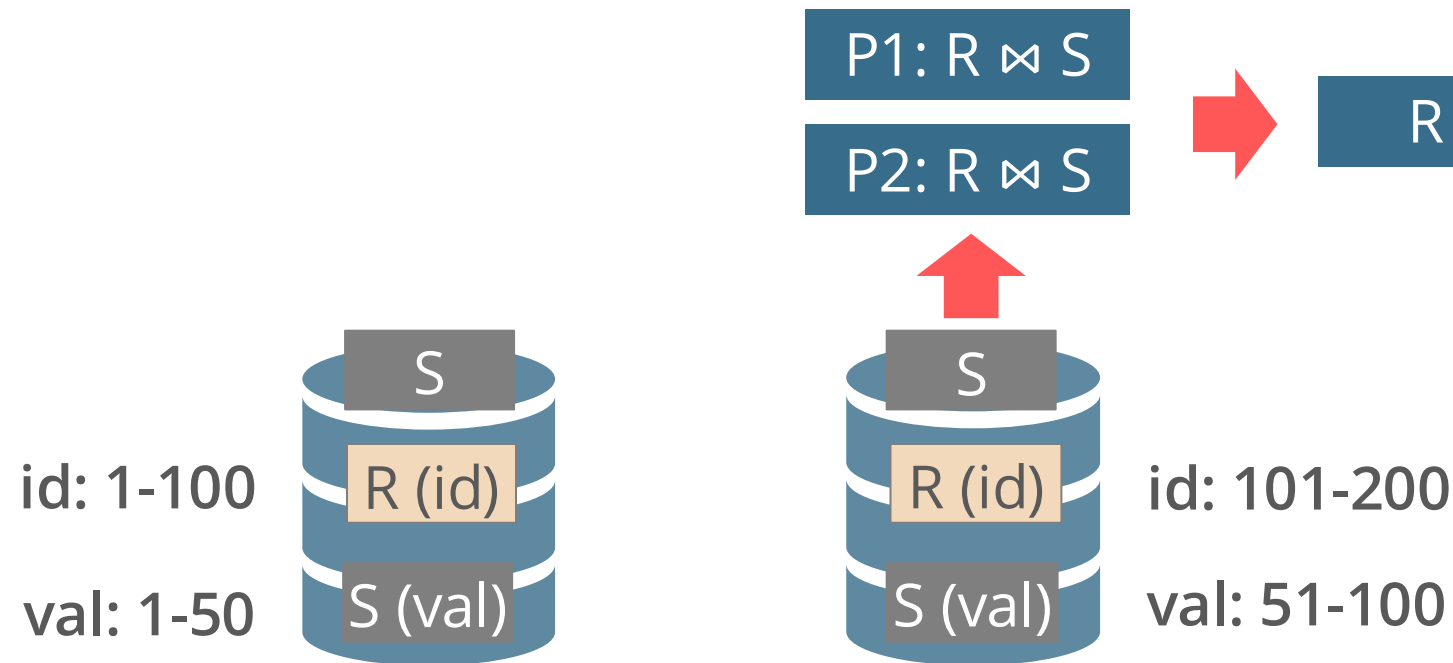
```
SELECT * FROM R JOIN S  
ON R.id = S.id
```



SCENARIO #3

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

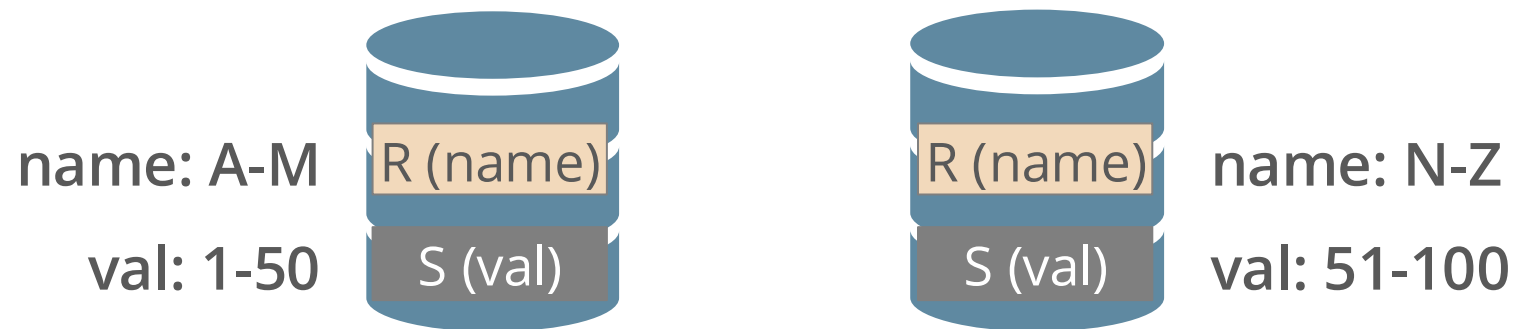
```
SELECT * FROM R JOIN S
ON R.id = S.id
```



SCENARIO #4

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

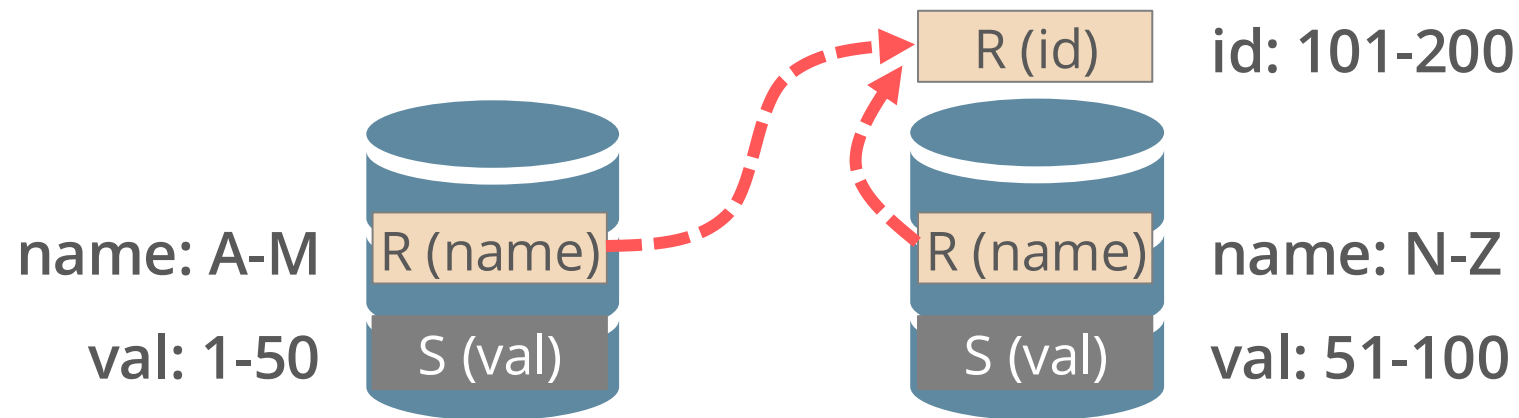
```
SELECT * FROM R JOIN S  
ON R.id = S.id
```



SCENARIO #4

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

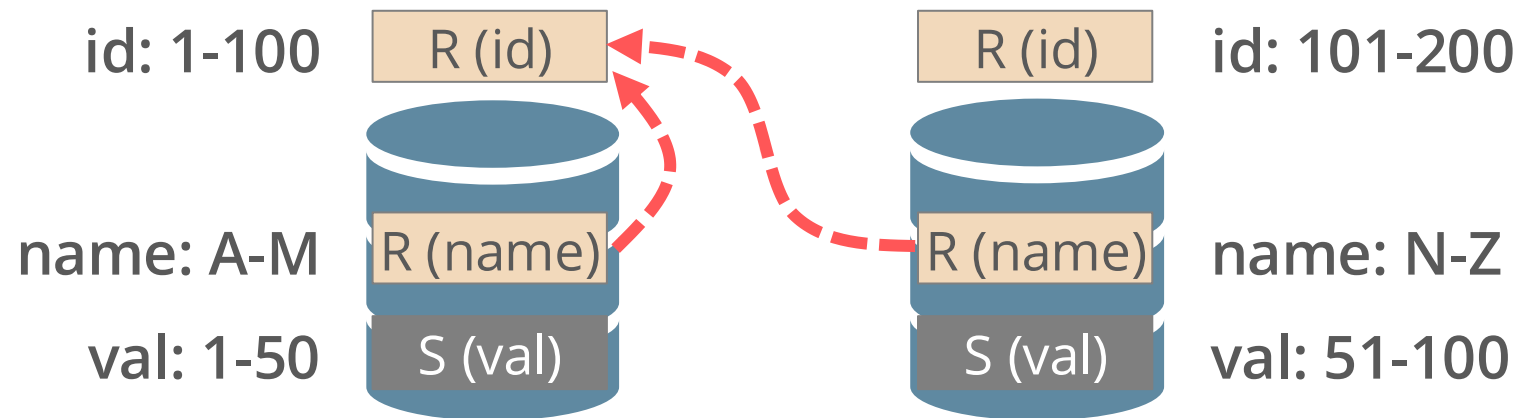
```
SELECT * FROM R JOIN S  
ON R.id = S.id
```



SCENARIO #4

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

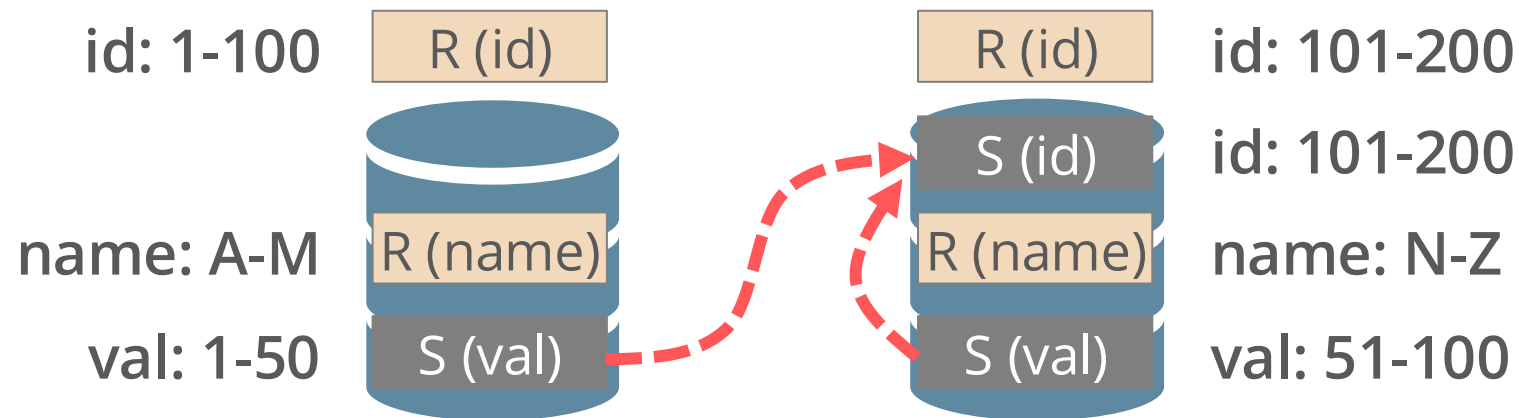
```
SELECT * FROM R JOIN S  
ON R.id = S.id
```



SCENARIO #4

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

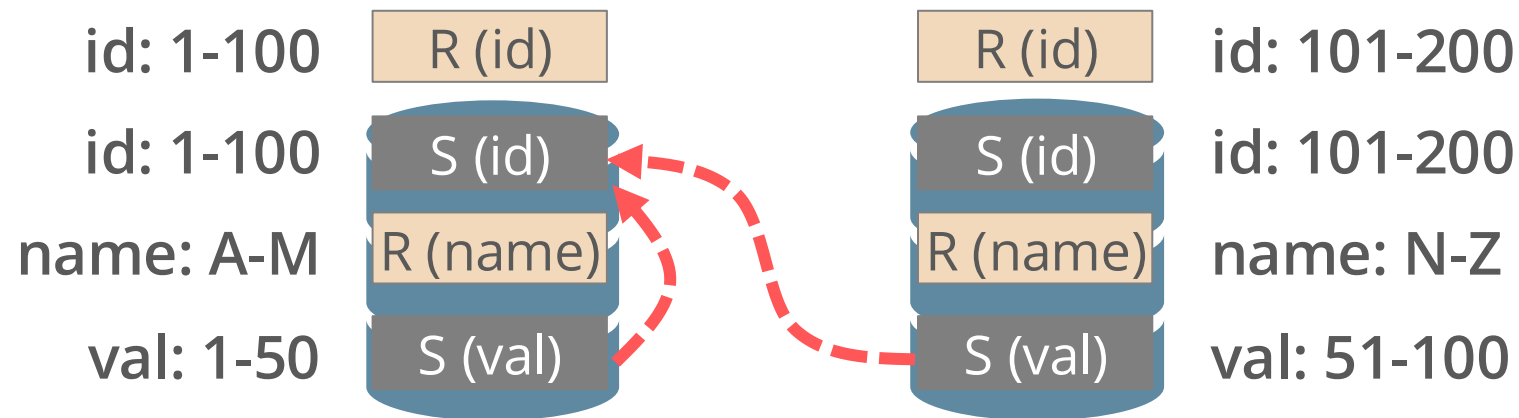
```
SELECT * FROM R JOIN S
ON R.id = S.id
```



SCENARIO #4

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

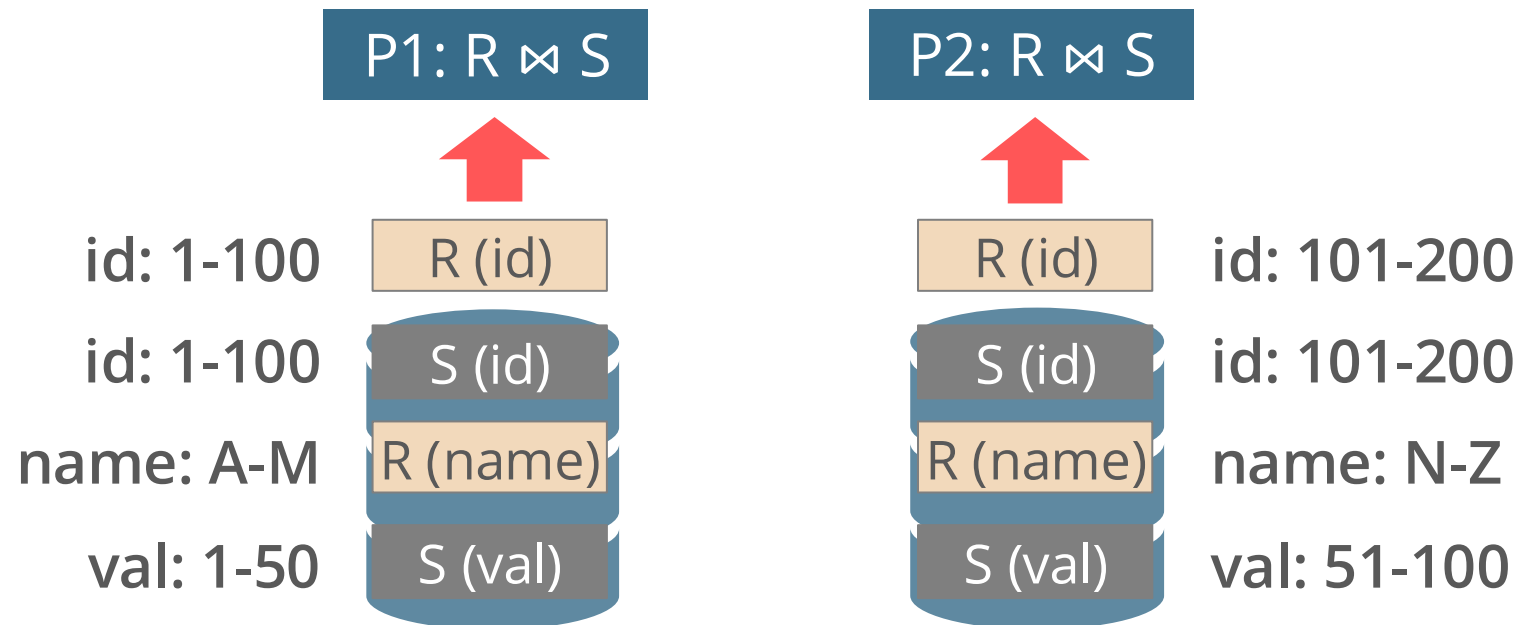
```
SELECT * FROM R JOIN S
ON R.id = S.id
```



SCENARIO #4

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

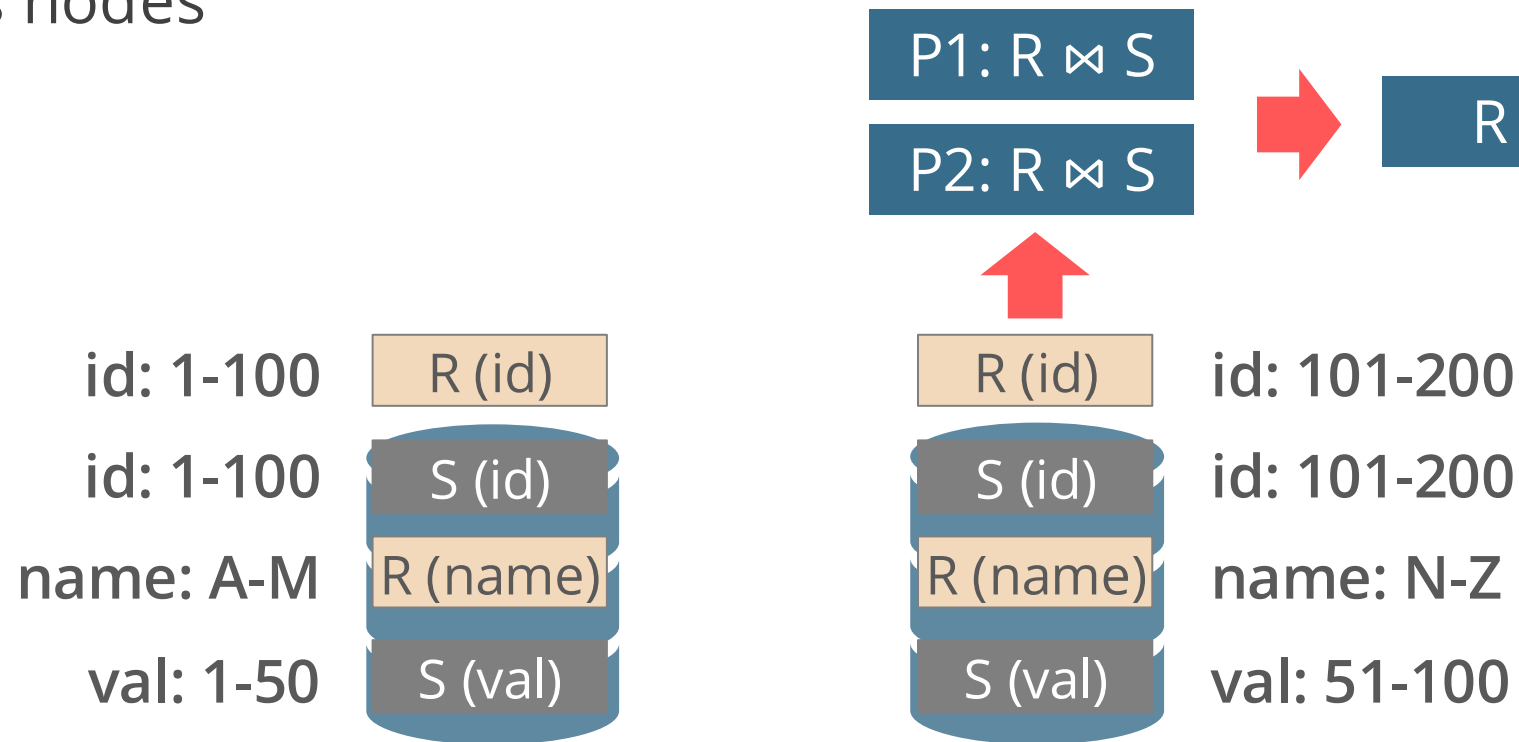
```
SELECT * FROM R JOIN S
ON R.id = S.id
```



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
```



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 PLANNING – CONT.

Query optimisation needs to consider **network cost**

Either in terms of time or total amount of data sent among nodes

Less important is the number of I/Os on a given node

Nodes may have to receive data from other nodes to start processing data

If a table is sorted on only a single machine for example

Since we have multiple nodes to use, we now care about **bottlenecks**

Uneven number of tuples on each node causes the total time spent doing operations (scanning, sorting, etc.) to be the maximum time spent of each individual node

E.g., : Node 1 takes 500ms and Node 2 takes 300ms, then overall parallel query takes 500ms

SUMMARY

Parallelism natural to query processing

Intra-op, inter-op, & Inter-query parallelism all possible

Shared nothing vs. Shared memory vs. Shared disk

Shared memory: easiest SW, costliest HW, doesn't scale indefinitely

Shared nothing: cheap, scales well, harder to implement

Shared disk: a middle ground

Most DB operations can be done partition-parallel

Sort, hash, sort-merge join, hash-join...

Everything is harder in a parallel/distributed setting

Query execution, concurrency control, recovery