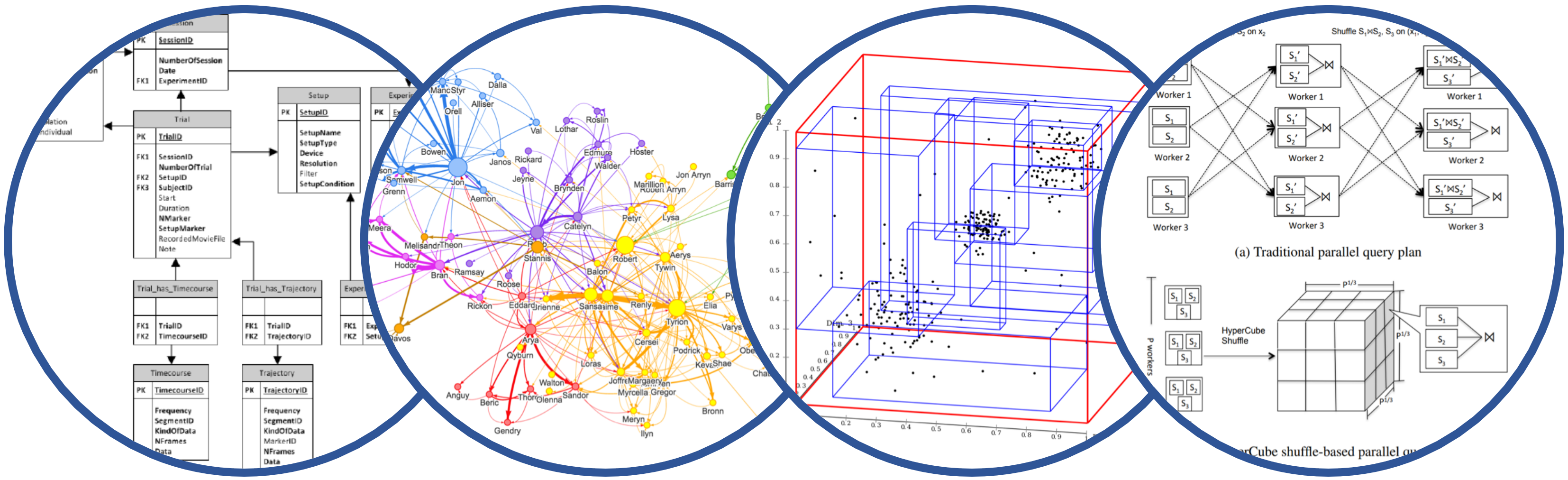


Introduction to Data Management

Parallel Processing

Paul G. Allen School of Computer Science and Engineering
University of Washington, Seattle

Course Context

- Phase 1: Core RDBMS (midterm topics)
  - SQL and RA
  - Logical and Physical Database Design
  - Transactions
- Interlude: Misc. RDBMS Topics
  - Distributed Relational Databases
  - Spark query language
  - Datalog query language
- Phase 2: NoSQL and Streams

We Need More Power

- Humans have a tendency to tackle problems that are too big to compute
  - Breaking the enigma code (WWII)
    - Using automation (the bombe)
  - Computing rocket trajectories (Space Race)
    - Using programming languages (FORTRAN)
  - Now: Data driven applications
    - Protein folding
    - Internet of things
    - Financial forecasting
    - Weather prediction
    - Social media platforms
    - …

More Data, More Problems

- The rates at which we generate and use information have **outpaced the capabilities of a single computer**
- Problems:
  - Need more speed
  - Need more scale

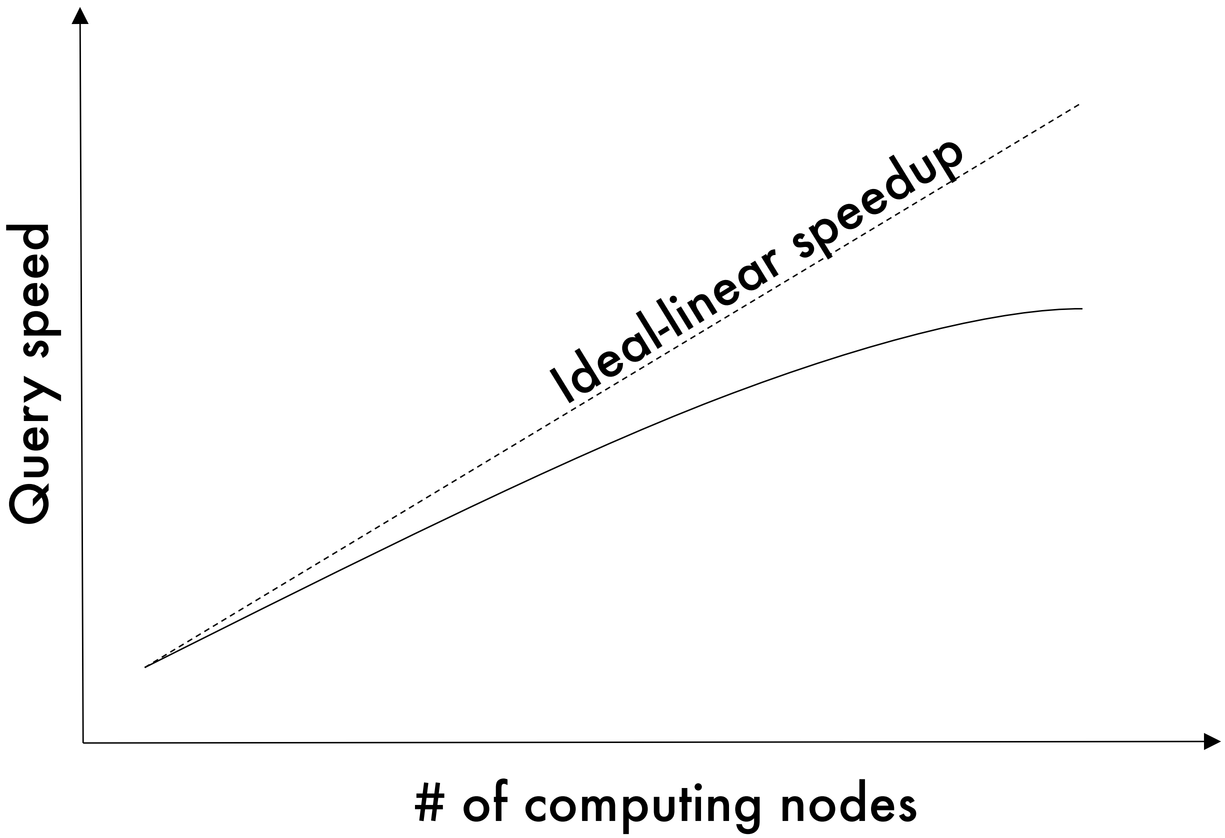
Parallel Computation

- Solution: Add more computing nodes
  - Multiple nodes → Parallel data management
- Most all computers have **multiple cores**
- Distributed architecture is easily available on **cloud services**

Speed Up

Speed up:

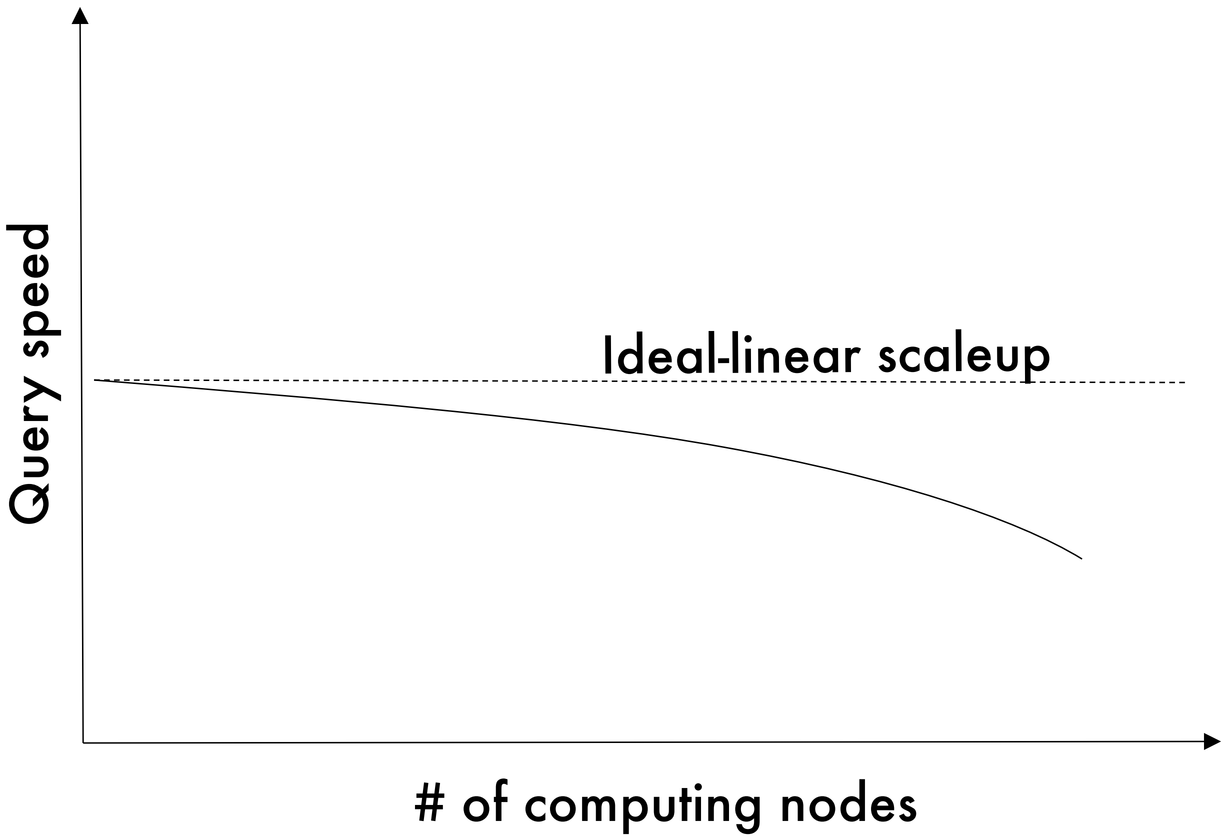
same data, more nodes → higher speed



Scale Up

Scale up:

more data, more nodes → same speed



Sublinear Expected Performance

- Parallel computing is not a magic bullet
- Common reasons for sublinear performance:
  - **Overhead cost**
    - Starting and coordinating operations on many nodes
  - **Interference/Contention**
    - Shared resources are not perfectly split
  - **Skew**
    - Process is only as fast as the slowest node

Implementations for Database Parallelism

- **Architecture Parallelism**
  - Shared Memory
  - Shared Disk
  - Shared Nothing*
- **Query Parallelism**
  - Inter-Query Parallelism
  - Intra-Query Parallelism
    - Inter-Operator Parallelism
    - Intra-Operator Parallelism*

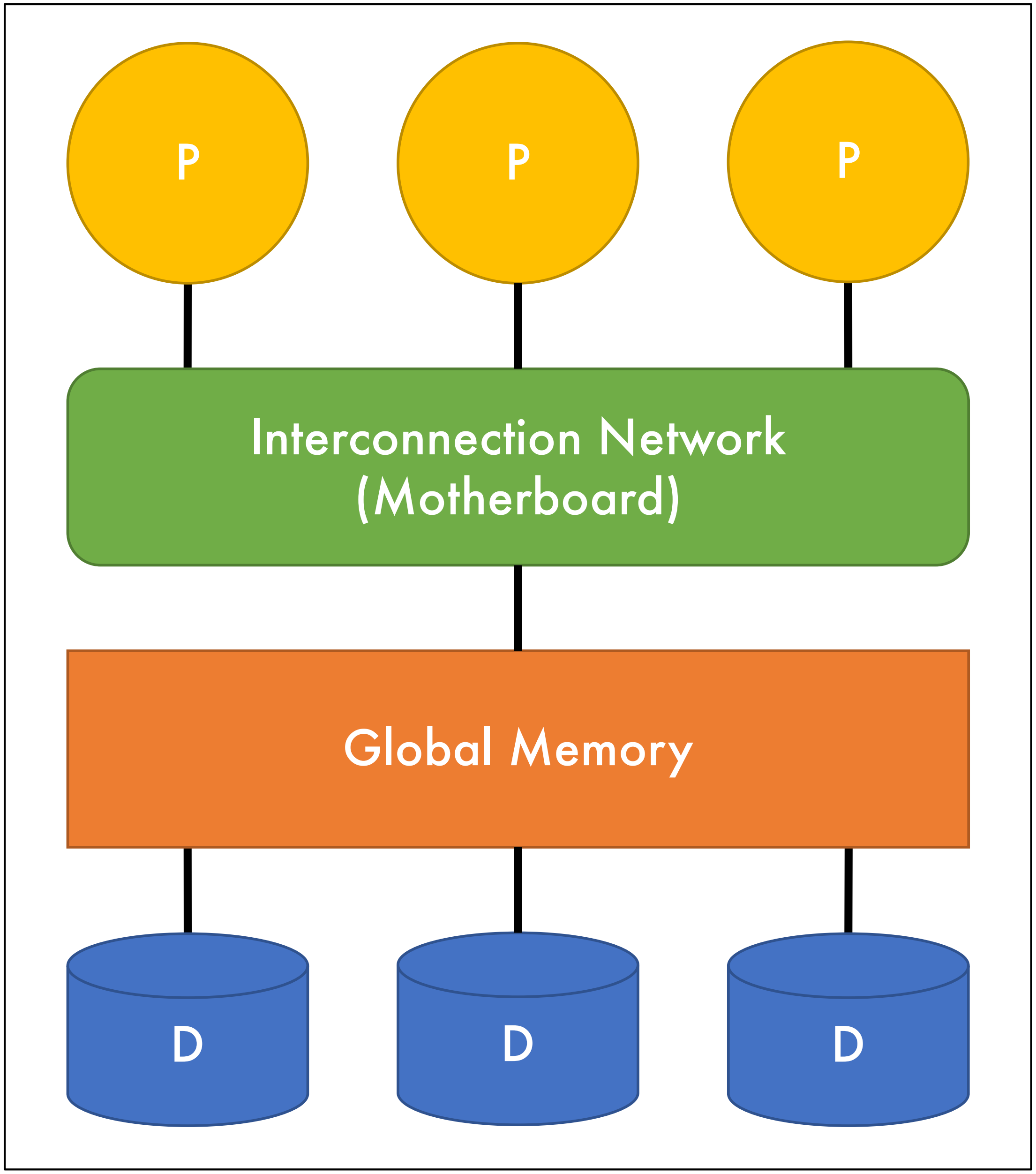
Hardware considerations

Software considerations

Implementations for Database Parallelism

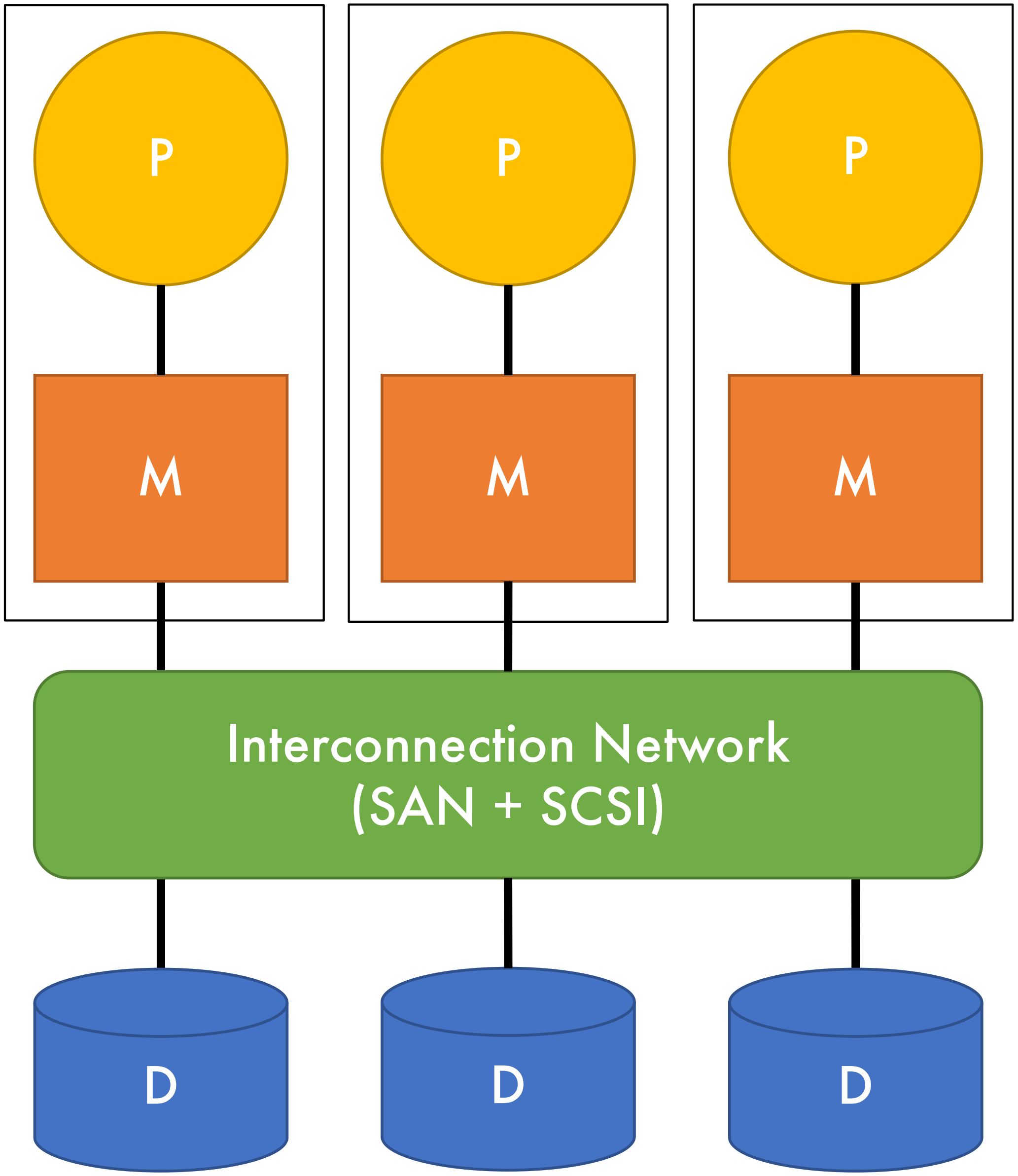
- Architecture Parallelism
  - **Shared Memory**
  - **Shared Disk**
  - **Shared Nothing** *
- Query Parallelism
  - Inter-Query Parallelism
  - Intra-Query Parallelism
    - Inter-Operator Parallelism
    - Intra-Operator Parallelism *

Shared-Memory Architecture



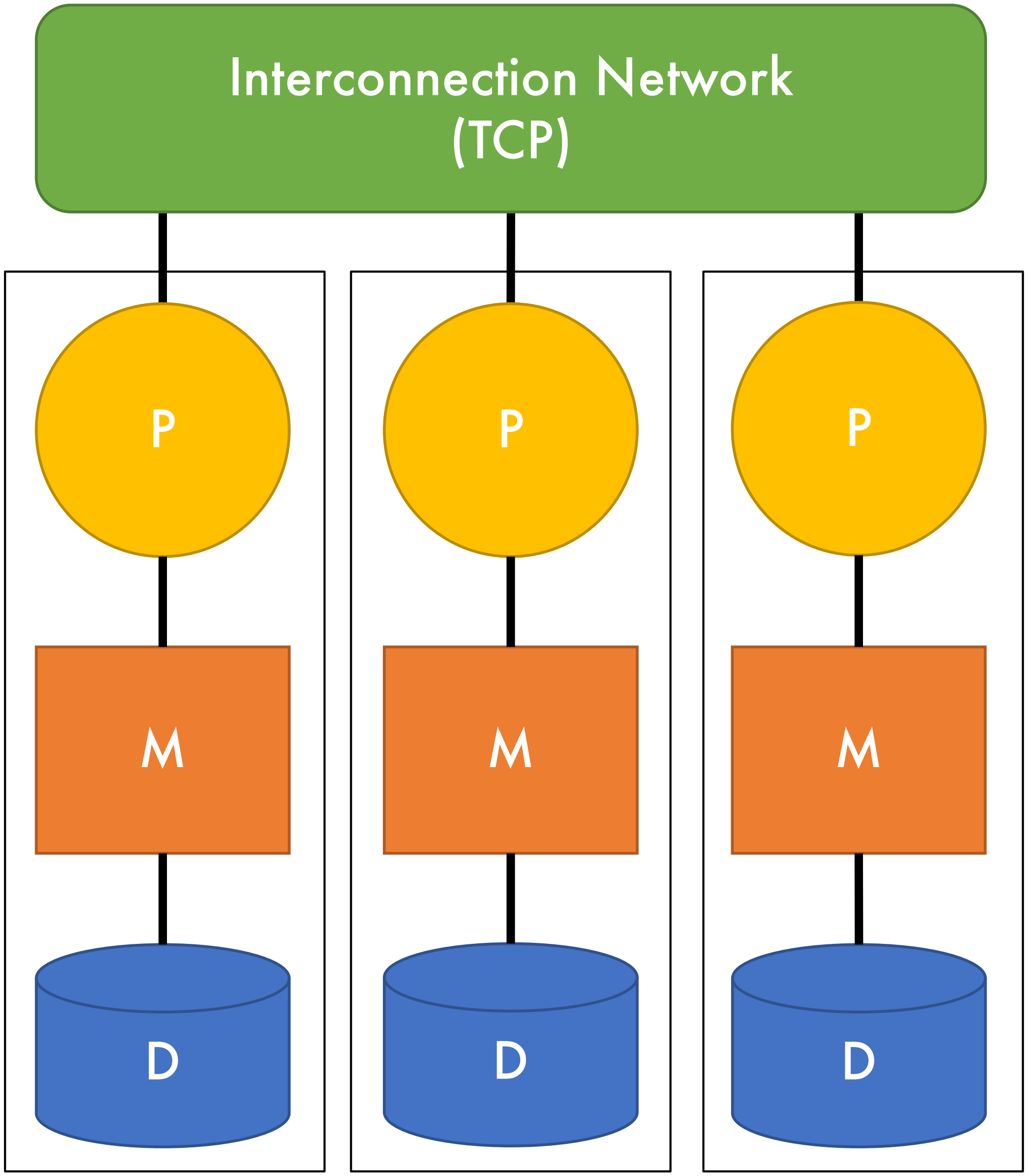
- Shared main memory and disks
- Your laptop or desktop uses this architecture
- Expensive to scale
- Easiest to implement on

Shared-Disk Architecture



- Only shared disks
- No contention for memory and high availability
- Typically 1-10 machines

Shared-Nothing Architecture*

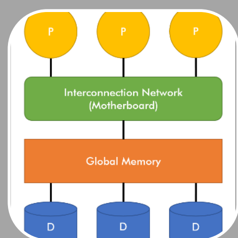


- Uses cheap, commodity hardware
- No contention for memory and high availability
- Theoretically can **scale infinitely**
- Hardest to implement on

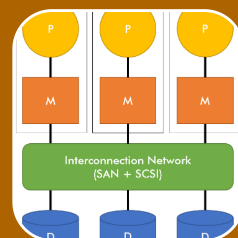
teradata.
APACHE Spark
MySQL Cluster

Architecture Tradeoffs

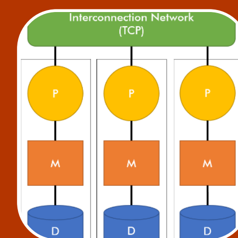
Main tradeoff is administration difficulty vs ability to scale



Shared-Memory Architecture



Shared-Disk Architecture



Shared-Nothing Architecture*

Easy to work on

Easy to scale

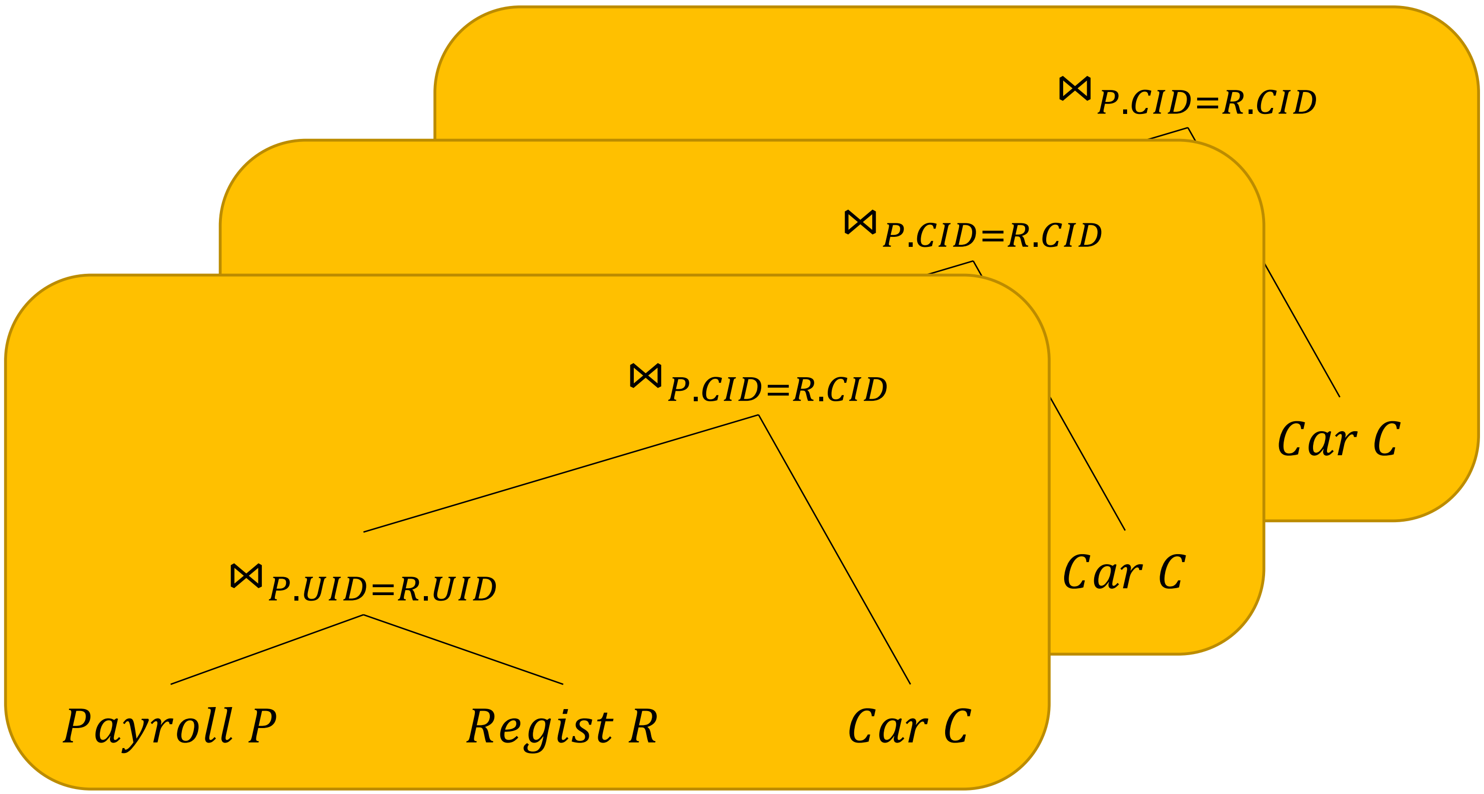
If you can't scale, your product dies, and everyone loses their job

Implementations for Database Parallelism

- Architecture Parallelism
  - Shared Memory
  - Shared Disk
  - Shared Nothing*
- Query Parallelism
  - **Inter-Query Parallelism**
  - Intra-Query Parallelism
    - **Inter-Operator Parallelism**
    - **Intra-Operator Parallelism***

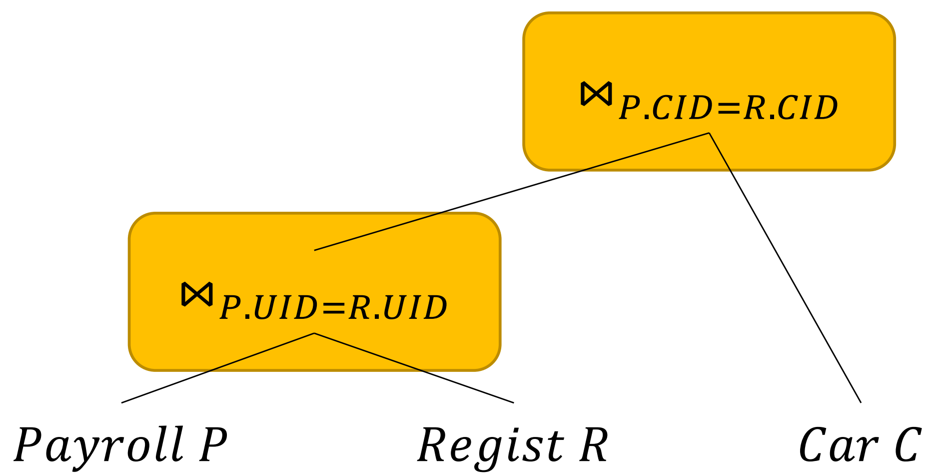
Inter-Query Parallelism

- Each transaction is processed on a separate node
- Scales very well for **lots of simple transactions**



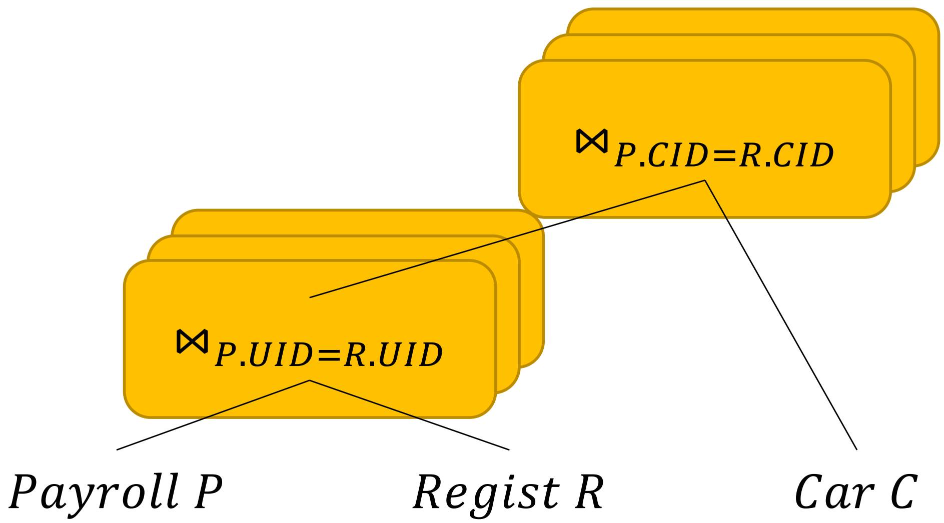
Inter-Operator Parallelism

- Each operator is processed on a separate node
- Scales very well for **complex analytical queries**



Intra-Operator Parallelism*

- Each operator is processed by multiple nodes
- Scales well in general



Shared-Nothing, Intra-Operator Database

From here, we will assume a system that consists of multiple commodity machines on a common network where nodes may carry out specified relational operations.

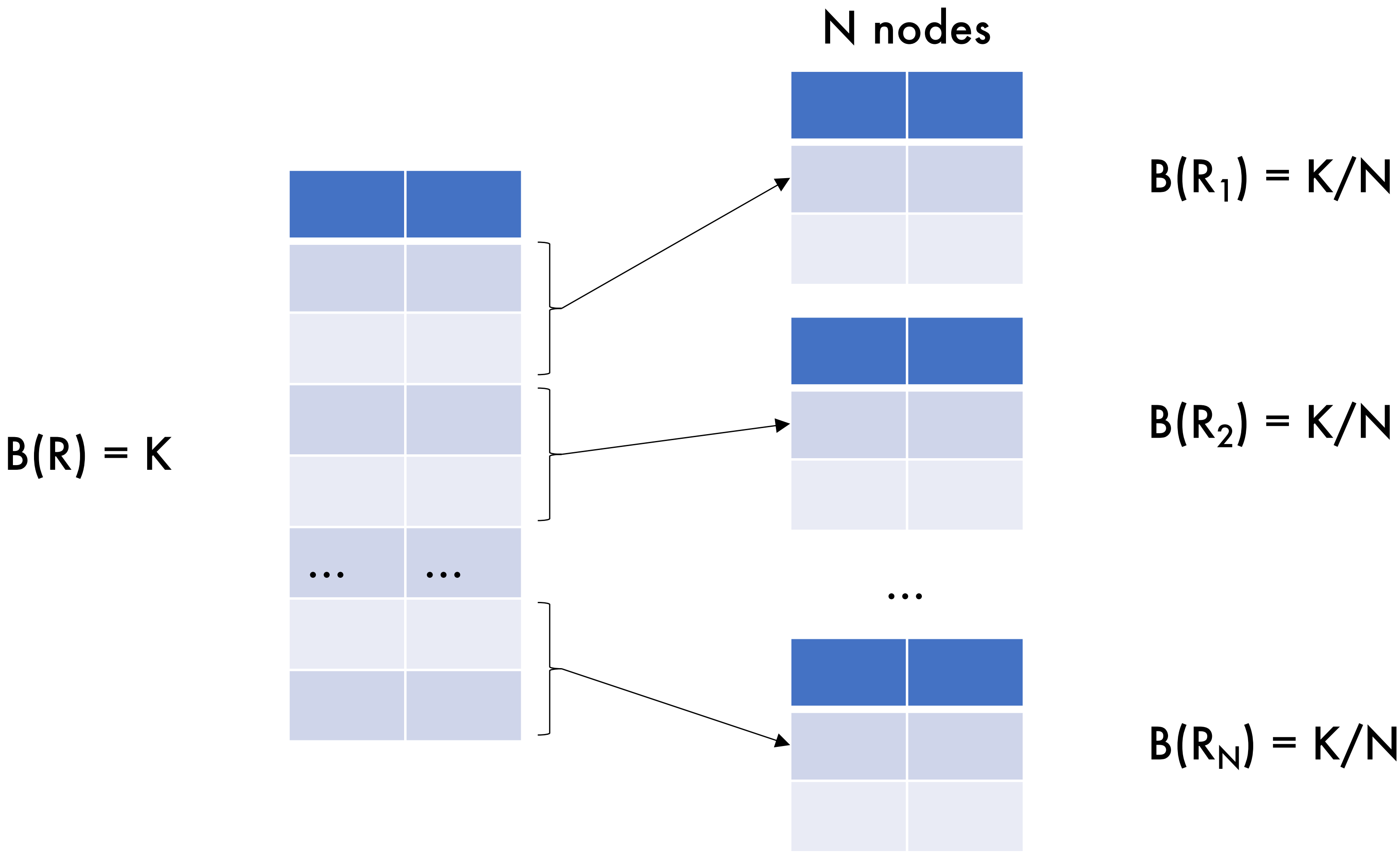
New problem: **Where does the data go?**

Unpartitioned Table

- Simplest choice if data can fit on a single node
- Might result in being a bottleneck

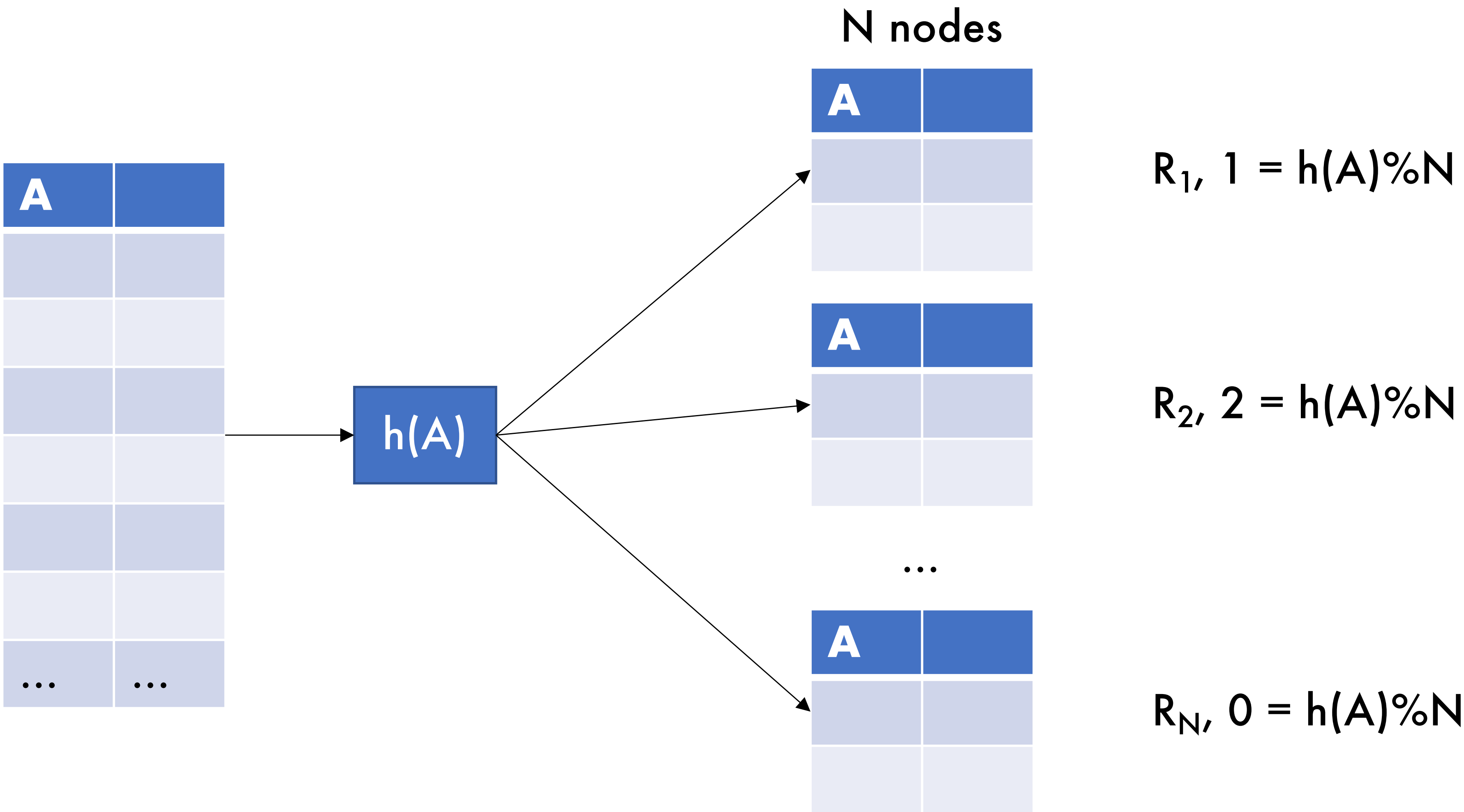
Block Partitioning

Tuples are horizontally partitioned by raw size



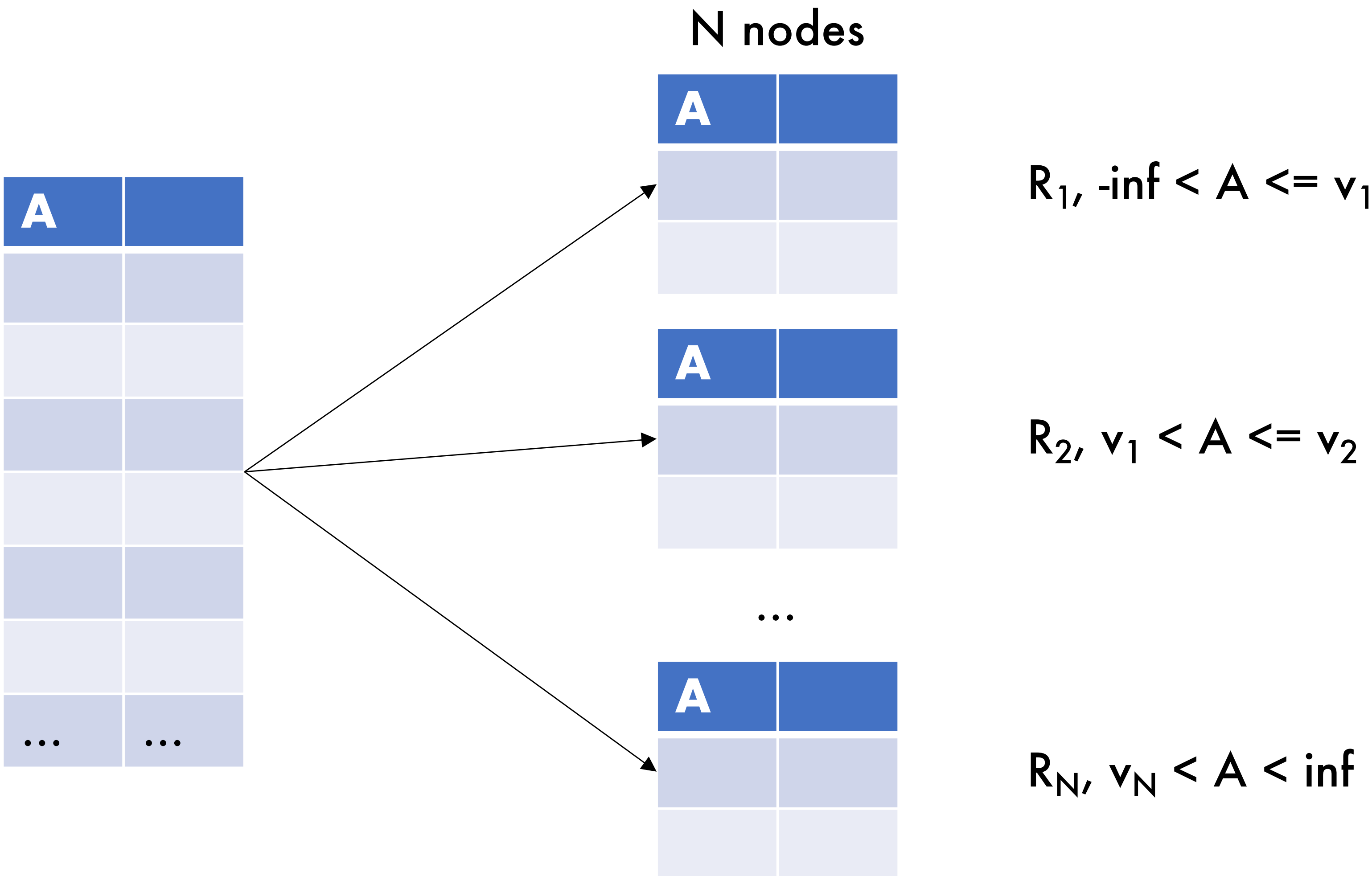
Hash Partitioning

Node contains tuples with chosen attribute hashes



Range Partitioning

Node contains tuples in chosen attribute ranges



The Justin Bieber Effect

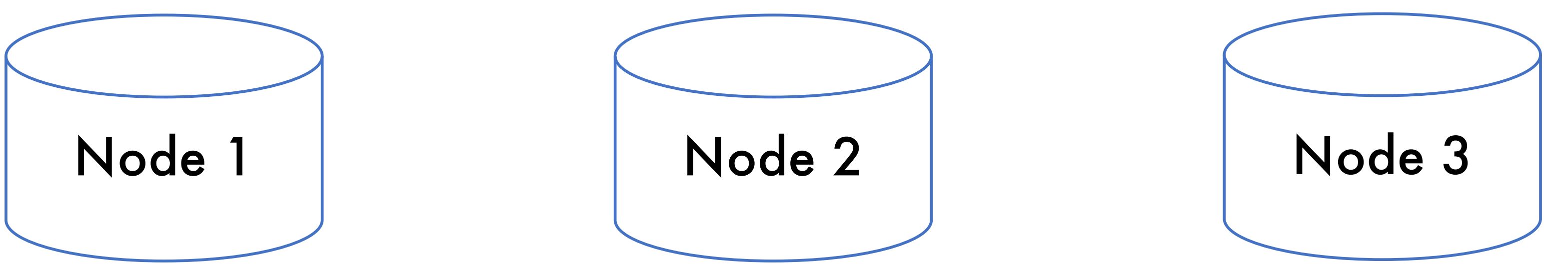
- Hashing data to nodes is very good when the attribute chosen better approximates a uniform distribution
- Keep in mind: Certain nodes will become **bottlenecks** if a **poorly chosen attribute is hashed**

Partitioned Aggregation

1. Hash shuffle tuples
2. Local aggregation

Assume:
R is block partitioned

```
SELECT *
  FROM R
 GROUP BY R.A
```



Partitioned Aggregation

1. Hash shuffle tuples
2. Local aggregation

Assume:
R is block partitioned

```
SELECT *
  FROM R
 GROUP BY R.A
```

A	…
1	…
2	…

Node 1

A	…
2	…
3	…

Node 2

A	…
3	…
1	…

Node 3

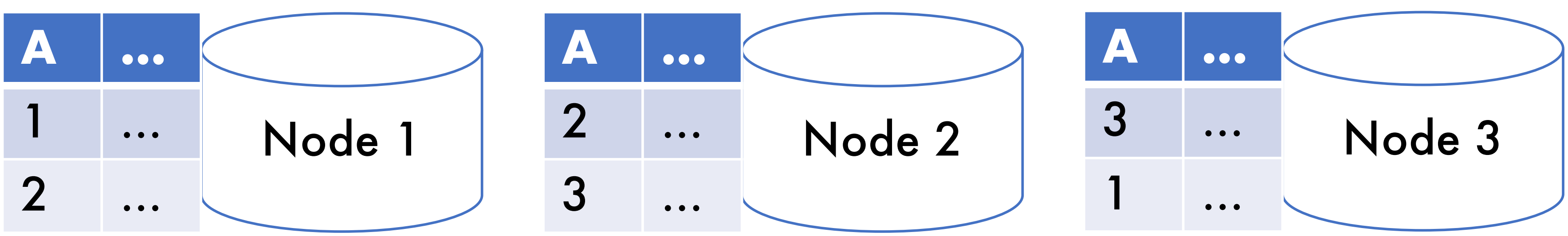
Partitioned Aggregation

1. Hash shuffle tuples
2. Local aggregation

Assume:
R is block partitioned

```
SELECT *
  FROM R
 GROUP BY R.A
```

$\gamma_{R.A}$ $\gamma_{R.A}$ $\gamma_{R.A}$

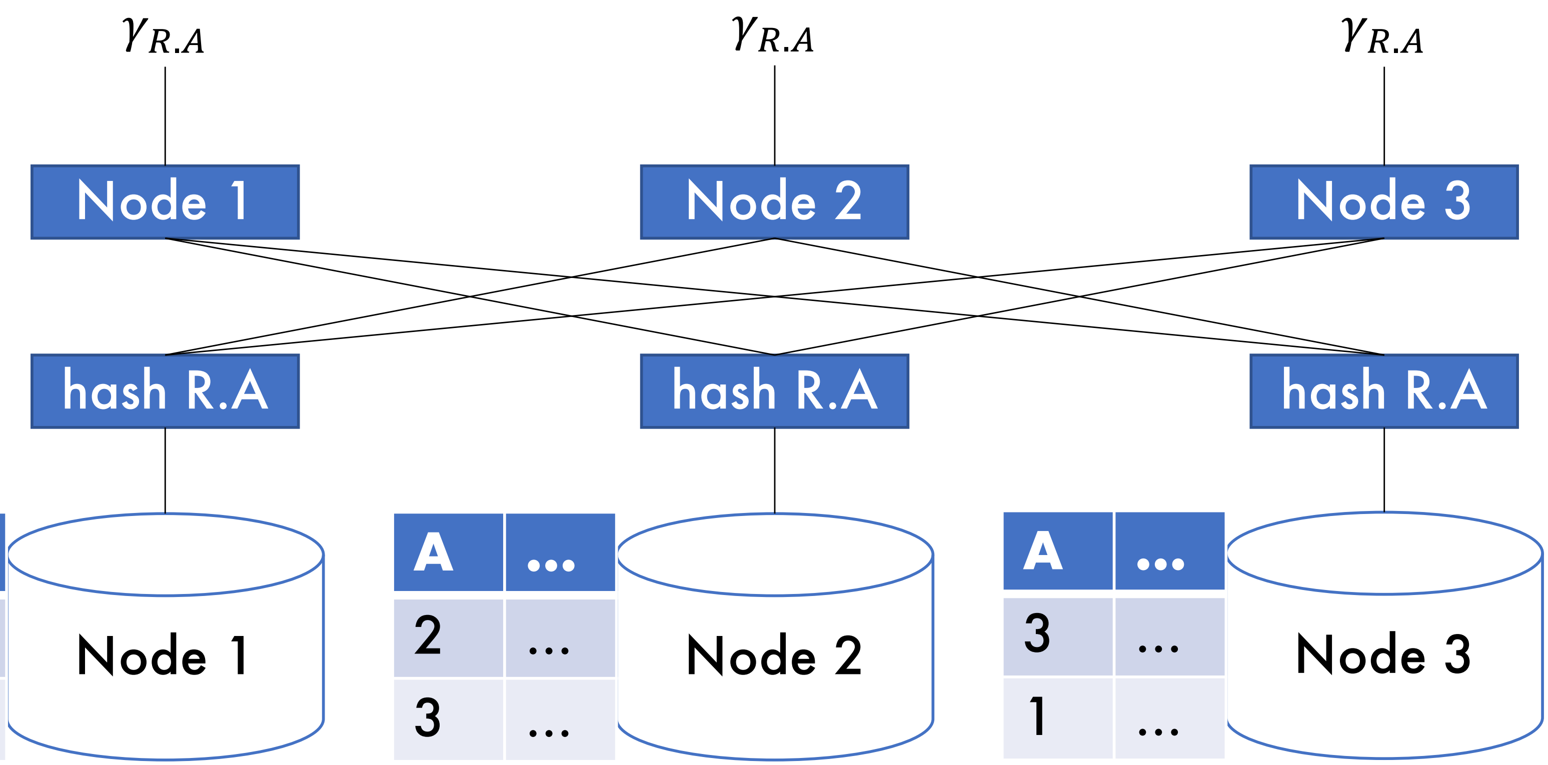


Partitioned Aggregation

1. Hash shuffle tuples
2. Local aggregation

Assume:
R is block partitioned

```
SELECT *
  FROM R
 GROUP BY R.A
```



A	...
1	...
2	...

A	...
2	...
3	...

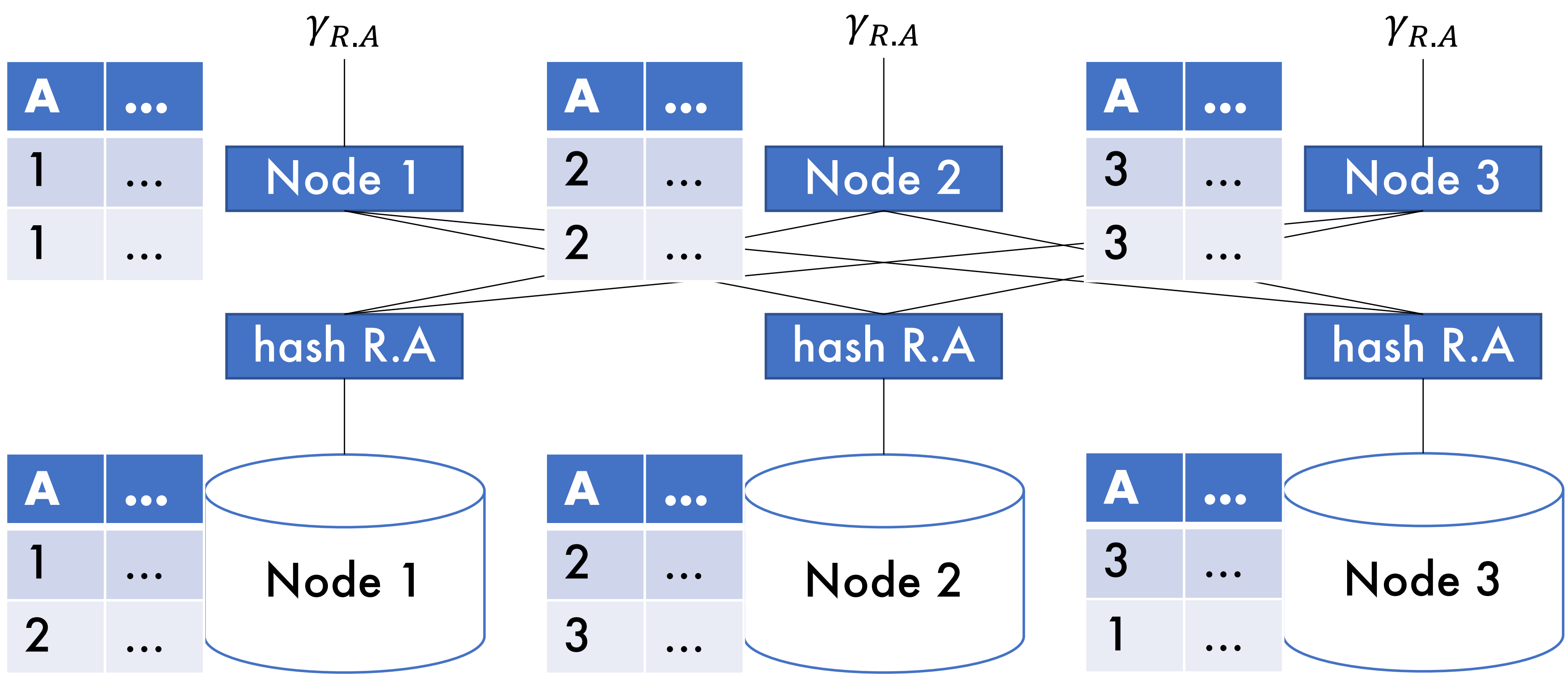
A	...
3	...
1	...

Partitioned Aggregation

1. Hash shuffle tuples
2. Local aggregation

Assume:
R is block partitioned

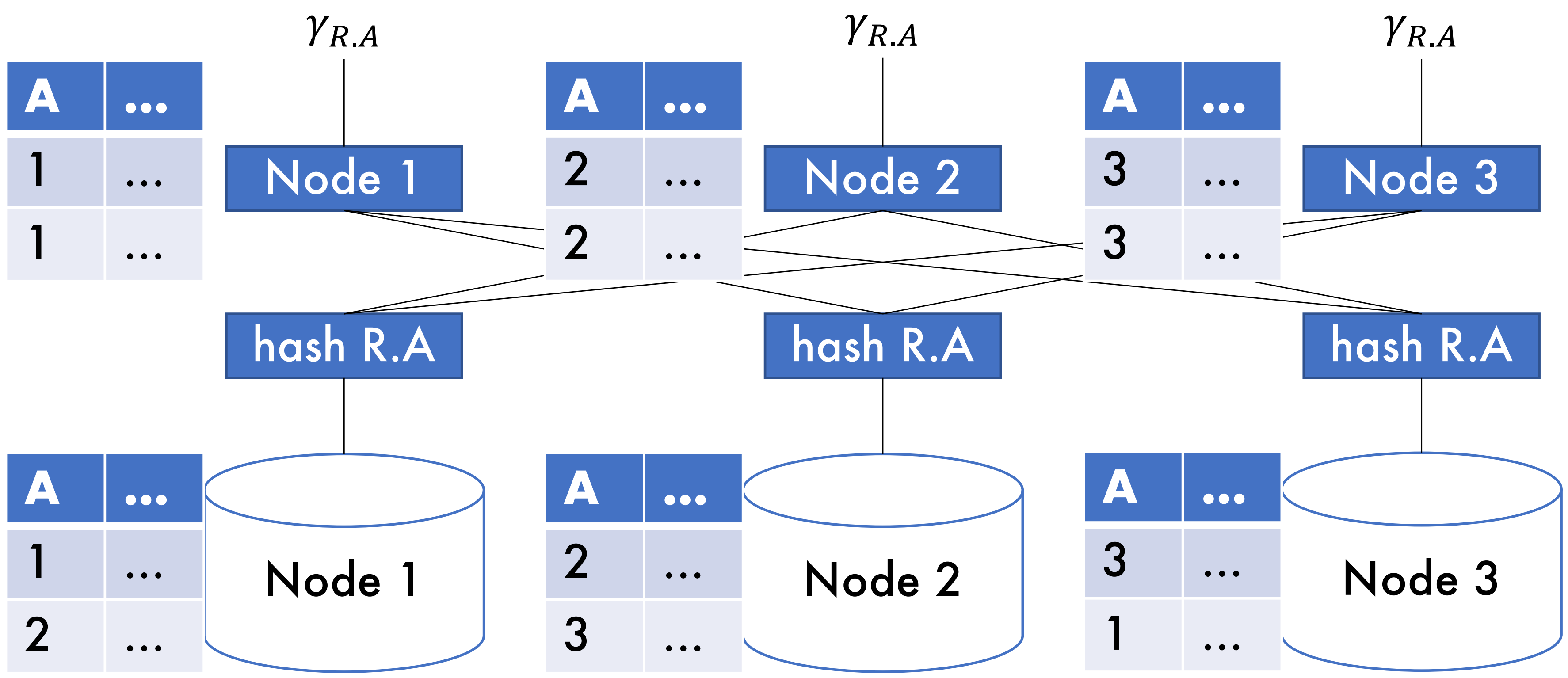
```
SELECT *
  FROM R
 GROUP BY R.A
```



Partitioned Aggregation

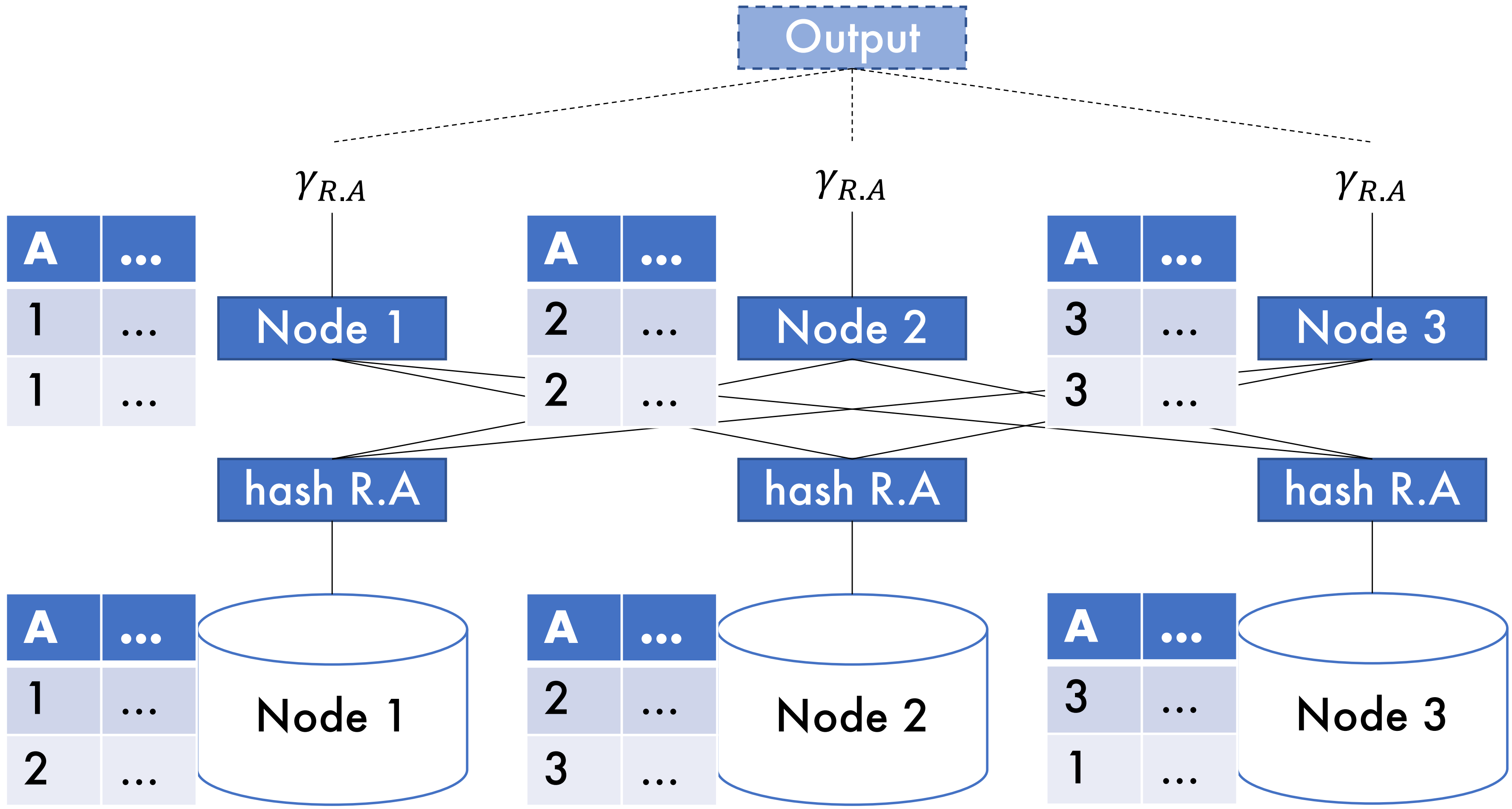
1. Hash shuffle tuples
2. Local aggregation

Would I need to shuffle if R was hash or range partitioned?



Implicit Union

Parallel query plans implicitly union at the end

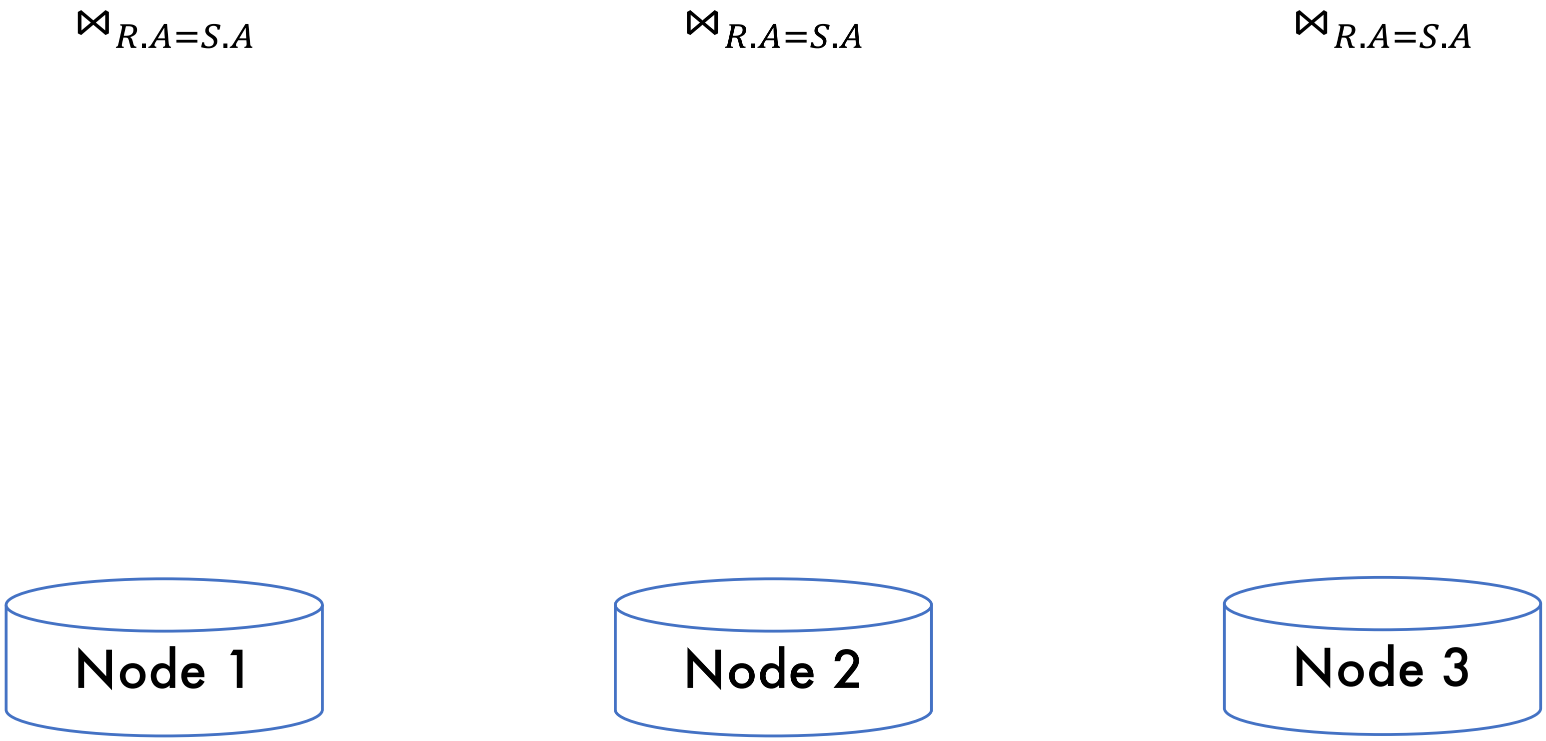


Partitioned Hash Equijoin Algorithm

1. Hash shuffle tuples on join attributes
2. Local join

Assume:
R and S are block partitioned

```
SELECT *
  FROM R, S
 WHERE R.A = S.A
```

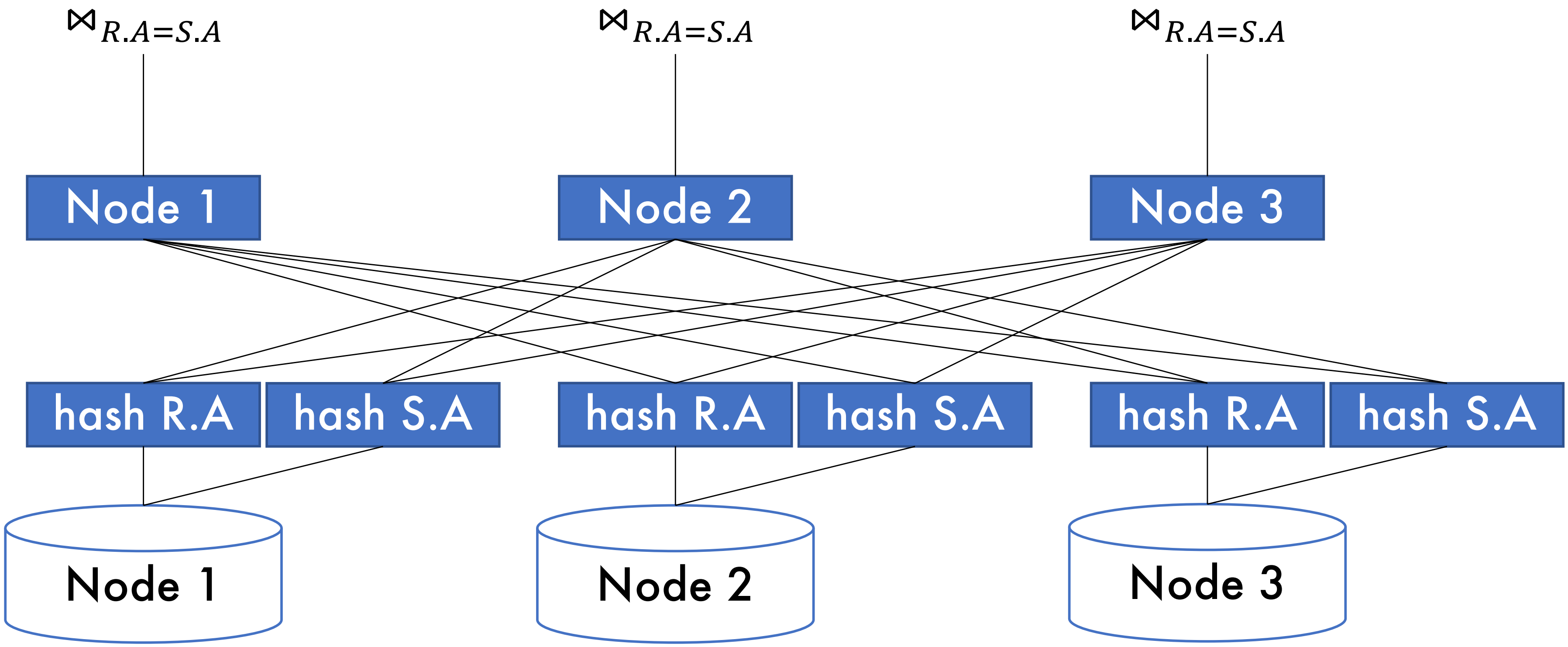


Partitioned Hash Equijoin Algorithm

1. Hash shuffle tuples on join attributes
2. Local join

Assume:
R and S are block partitioned

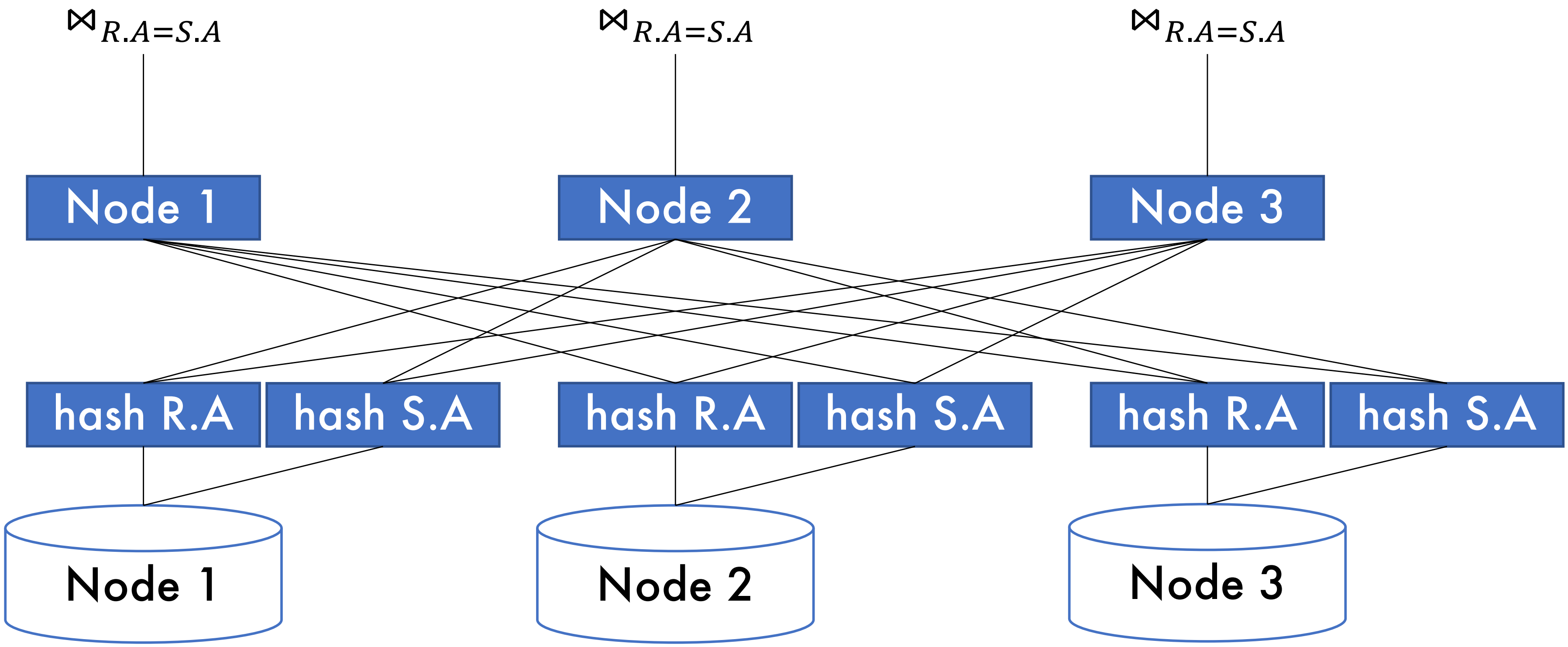
```
SELECT *
  FROM R, S
 WHERE R.A = S.A
```



Partitioned Hash Equijoin Algorithm

1. Hash shuffle tuples on join attributes
2. Local join

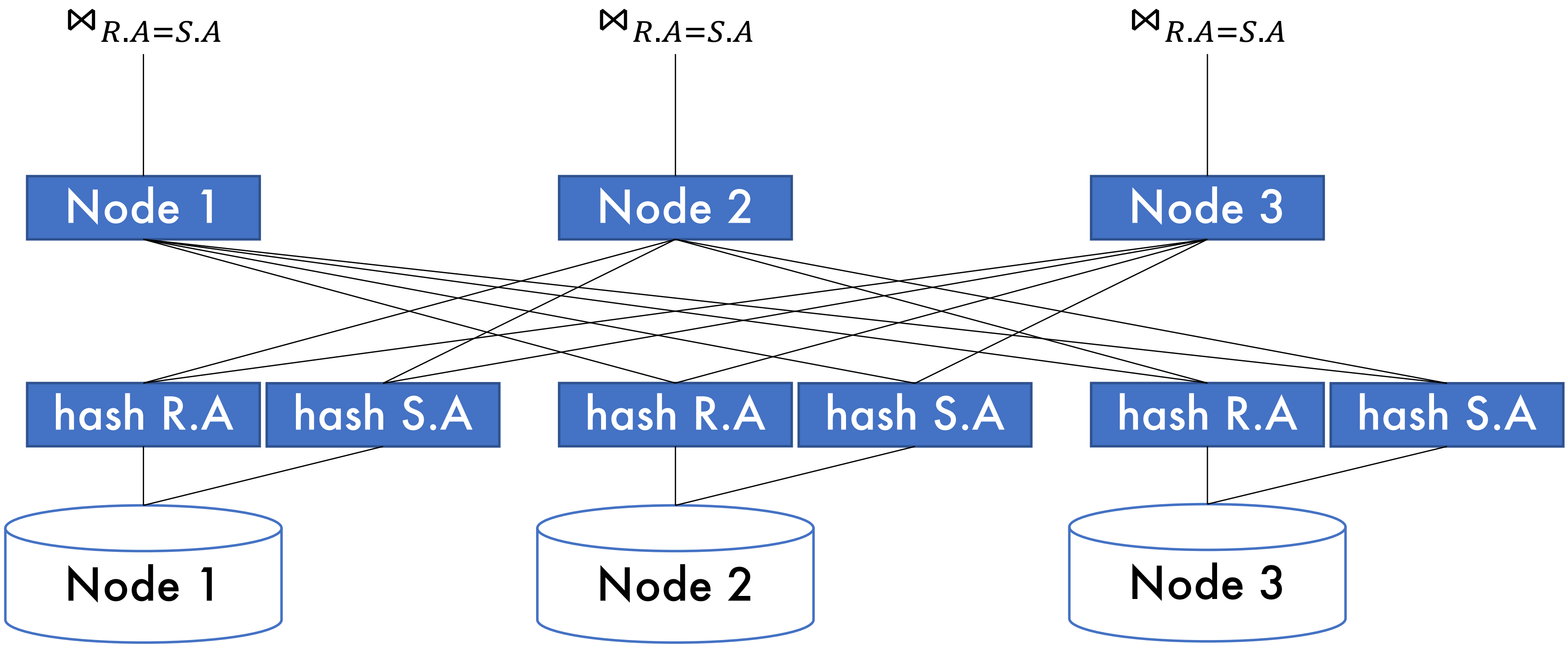
If S was **hash** partitioned on A (on the same hash function) would I need to shuffle S? R?



Partitioned Hash Equijoin Algorithm

1. Hash shuffle tuples on join attributes
2. Local join

If S was **range** partitioned on A would I need to shuffle S? R?

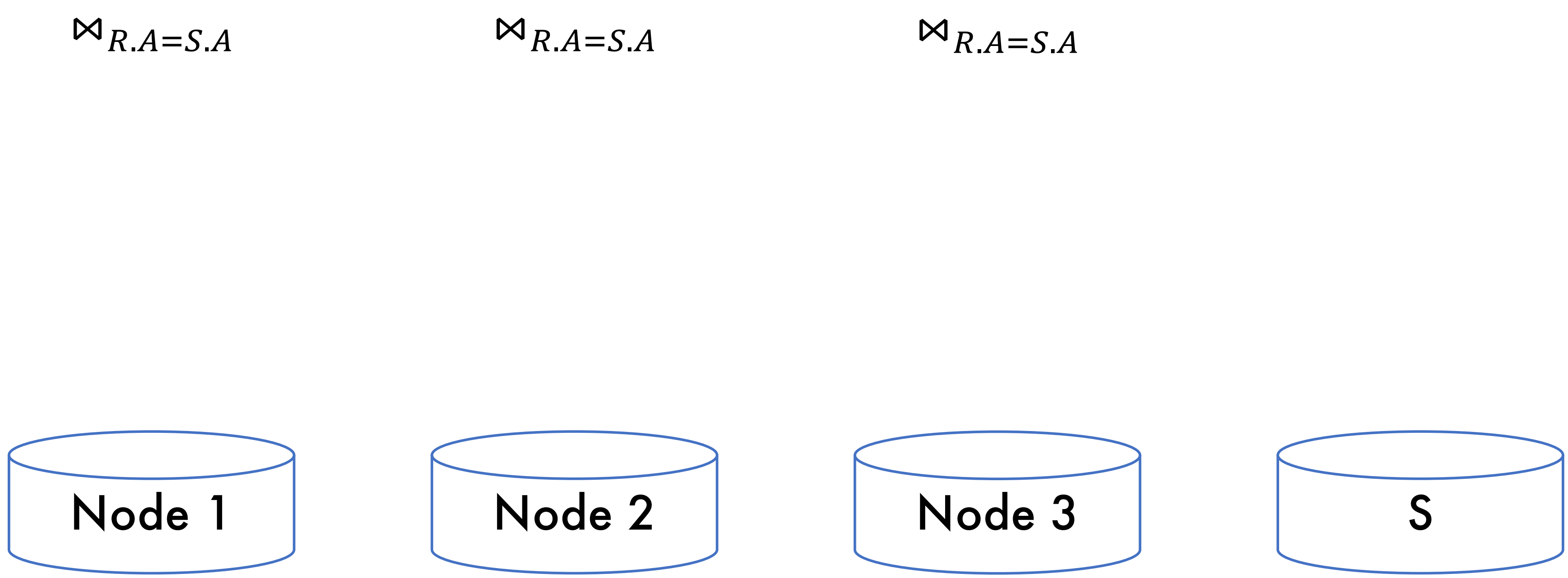


Broadcast Join

1. Broadcast unpartitioned table
2. Local join

Assume:
S is unpartitioned

```
SELECT *
  FROM R, S
 WHERE R.A = S.A
```

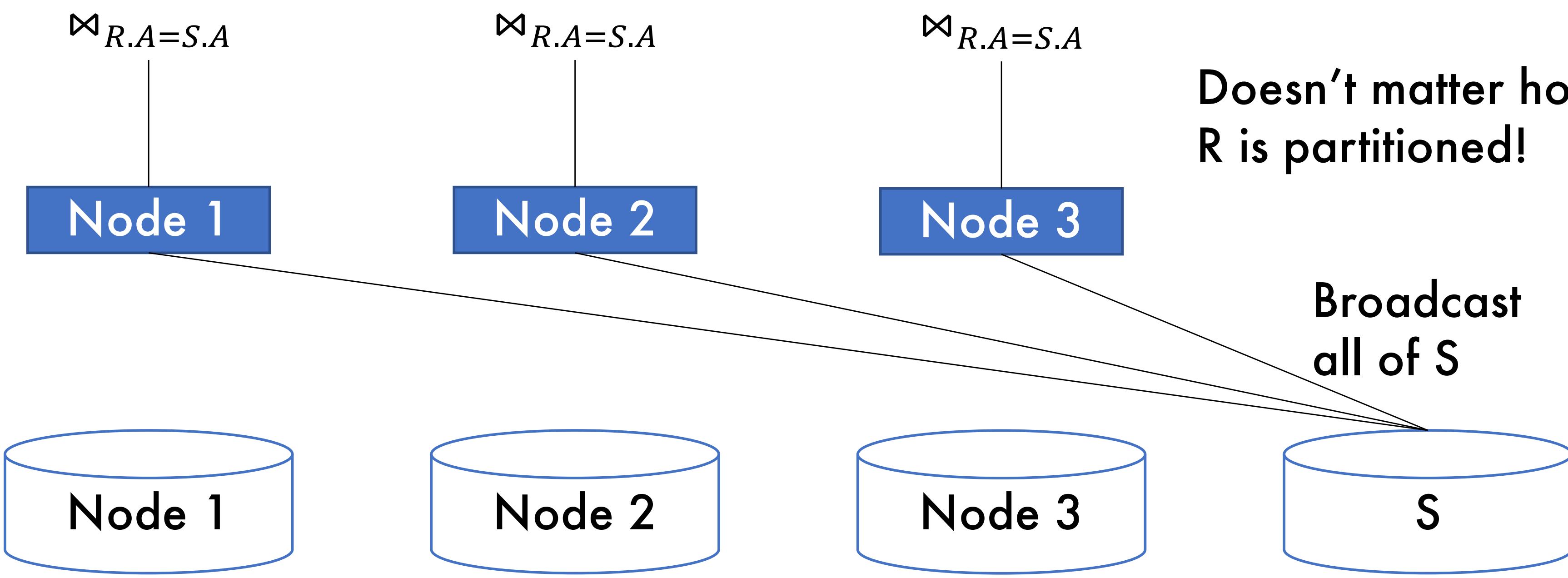


Broadcast Join

1. Broadcast unpartitioned table
2. Local join

Assume:
S is unpartitioned

```
SELECT *
  FROM R, S
 WHERE R.A = S.A
```



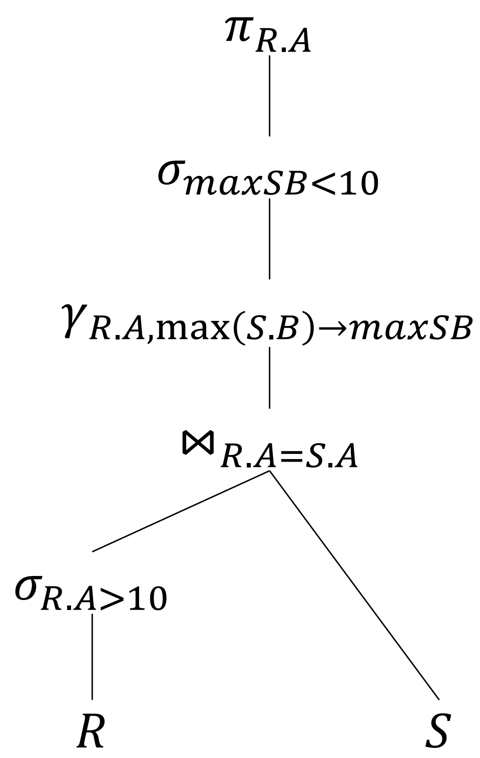
Doesn't matter how R is partitioned!

Broadcast all of S

Parallel Query Plan Example

All queries can be parallelized!

```
SELECT R.A
  FROM R, S
 WHERE R.A = S.A AND R.A > 10
 GROUP BY R.A
HAVING MAX(S.B) < 10
```



Parallel Query Plan Example

Assume:
R is block partitioned
S is hash partitioned on A

$$\pi_{R.A}$$

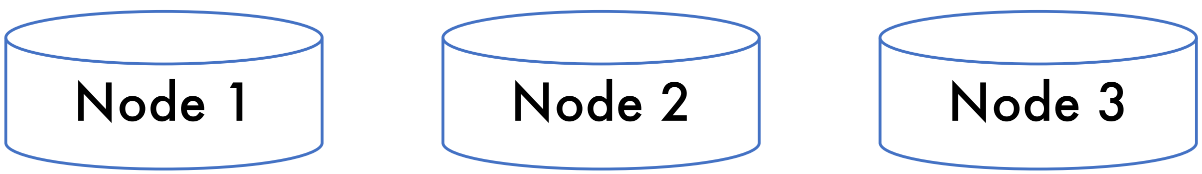
$$\sigma_{maxSB<10}$$

$$\gamma_{R.A,\max(S.B)\rightarrow maxSB}$$

$$\bowtie_{R.A=S.A}$$

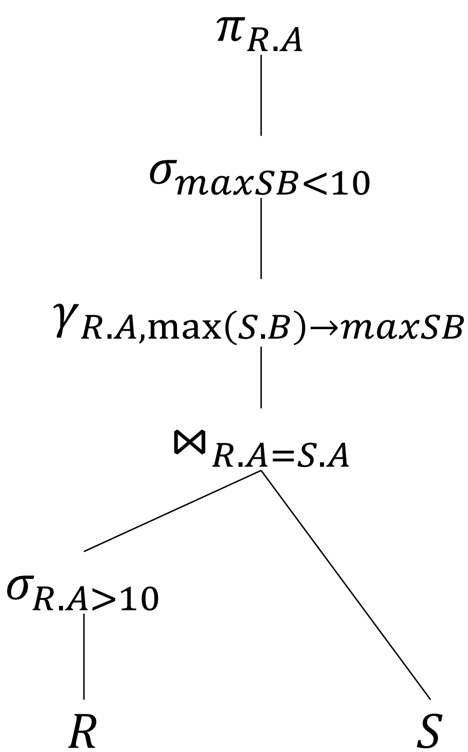
$$\sigma_{R.A>10}$$

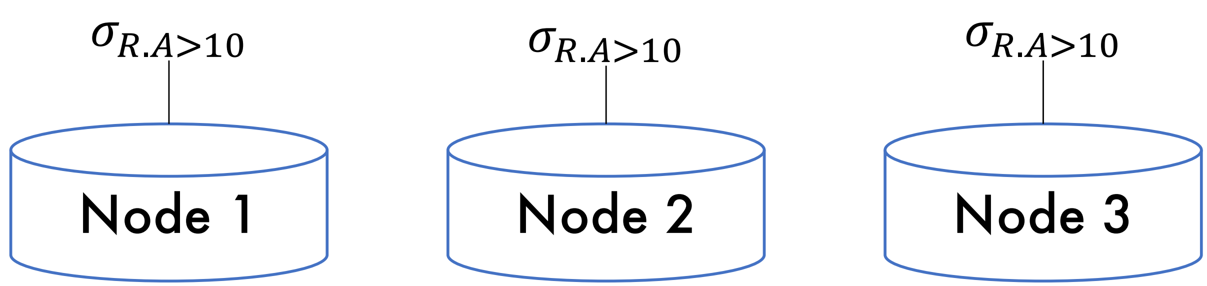
$$R \qquad\qquad S$$



Parallel Query Plan Example

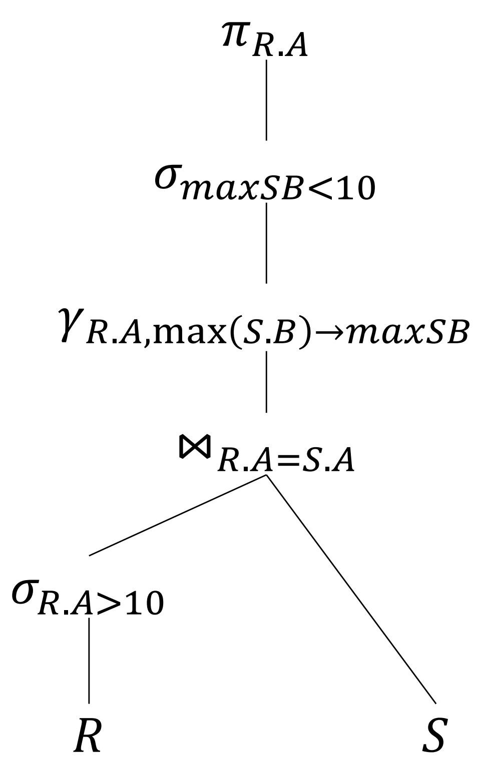
Assume:
R is block partitioned
S is hash partitioned on A

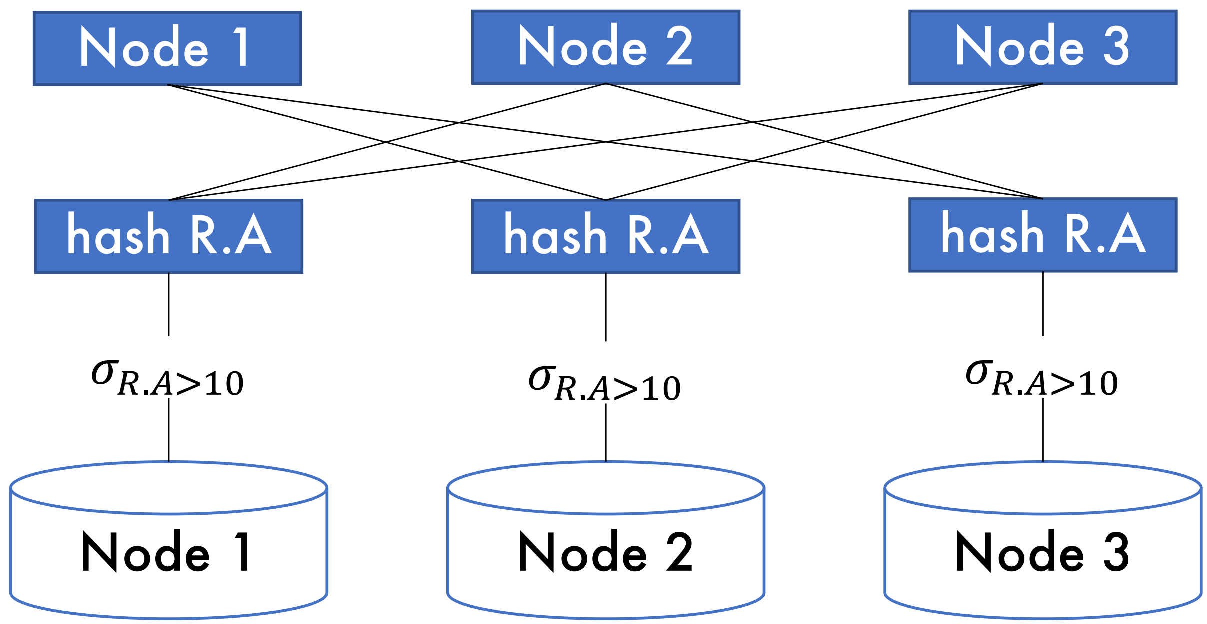




Parallel Query Plan Example

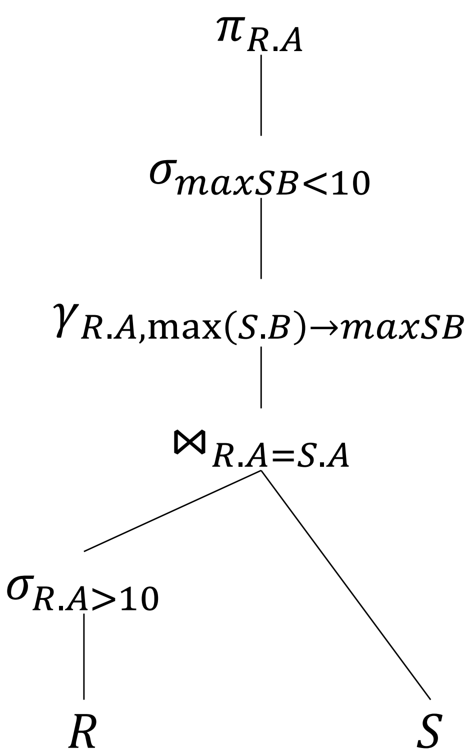
Assume:
R is block partitioned
S is hash partitioned on A

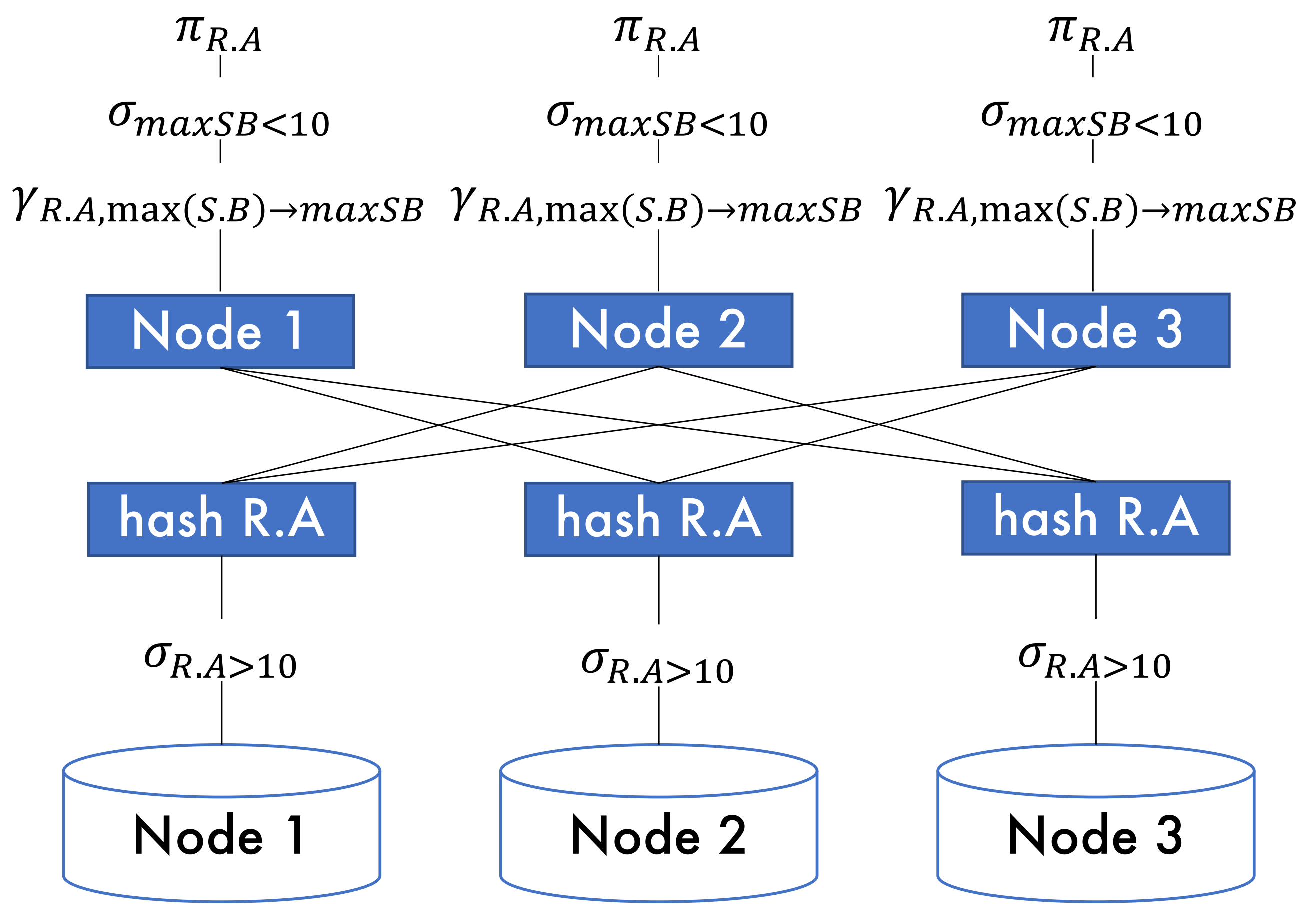




Parallel Query Plan Example

Assume:
R is block partitioned
S is hash partitioned on A





Next Time

- Programming with the Java Spark API