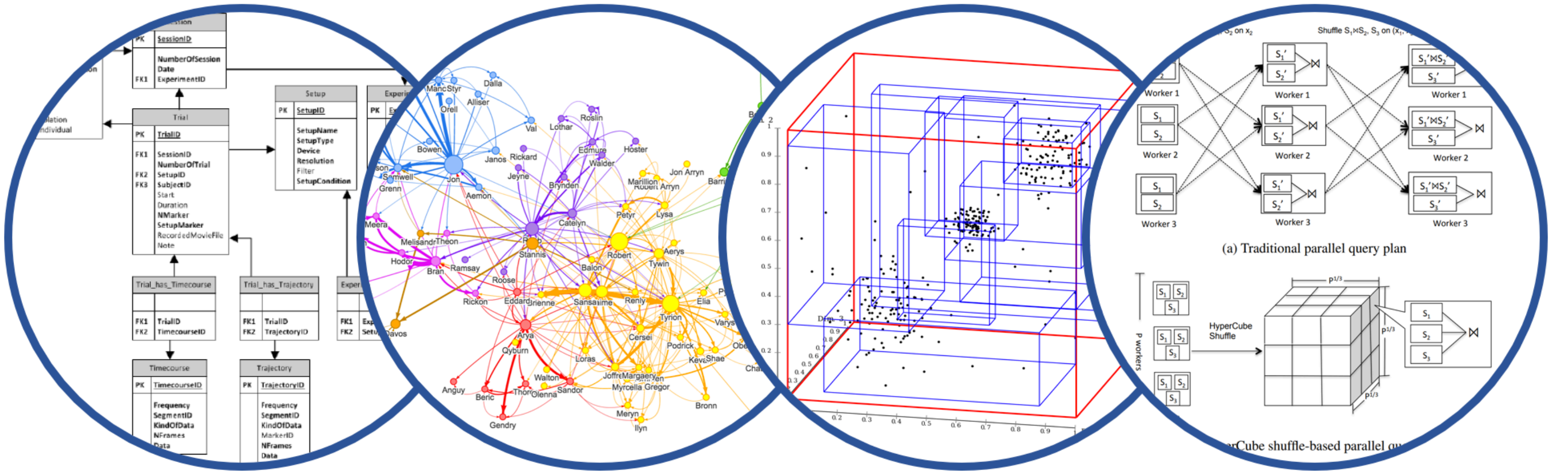


Database System Internals

Intro to Parallel DBMSs

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

Scaling Single Query Response Time

- OLAP: Query response time
  "Online Analytical Processing"
- Entire parallel system answers one query
- Goal is to improve query runtime
- Use case is analysis of massive datasets

Big Data

Volume alone is not an issue

- Relational databases *do* parallelize easily
  - Data partitioning
  - Parallel query processing
- SQL is *embarrassingly parallel*
  - We will learn how to do this!

Big Data

New **workloads** are an issue

- Big volumes, small analytics
  - OLAP queries: join + group-by + aggregate
  - Can be handled by today's RDBMSs
- Big volumes, big analytics
  - More complex Machine Learning
  - E.g. click prediction, topic modeling, SVM, k-means
  - Active area of research

Building Our Parallel DBMS

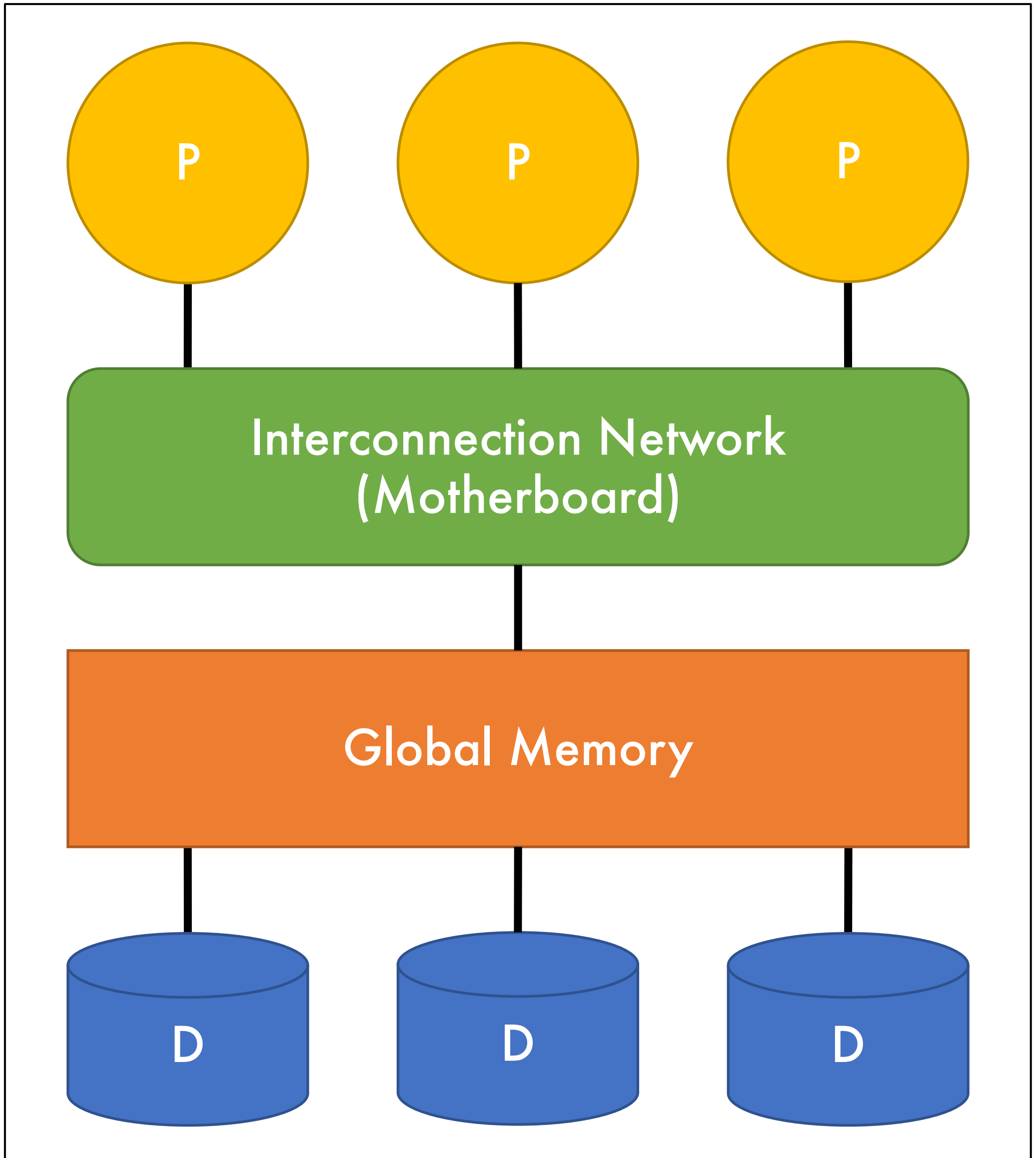
Data model? Relational

Scaleup goal? OLAP

Building Our Parallel DBMS

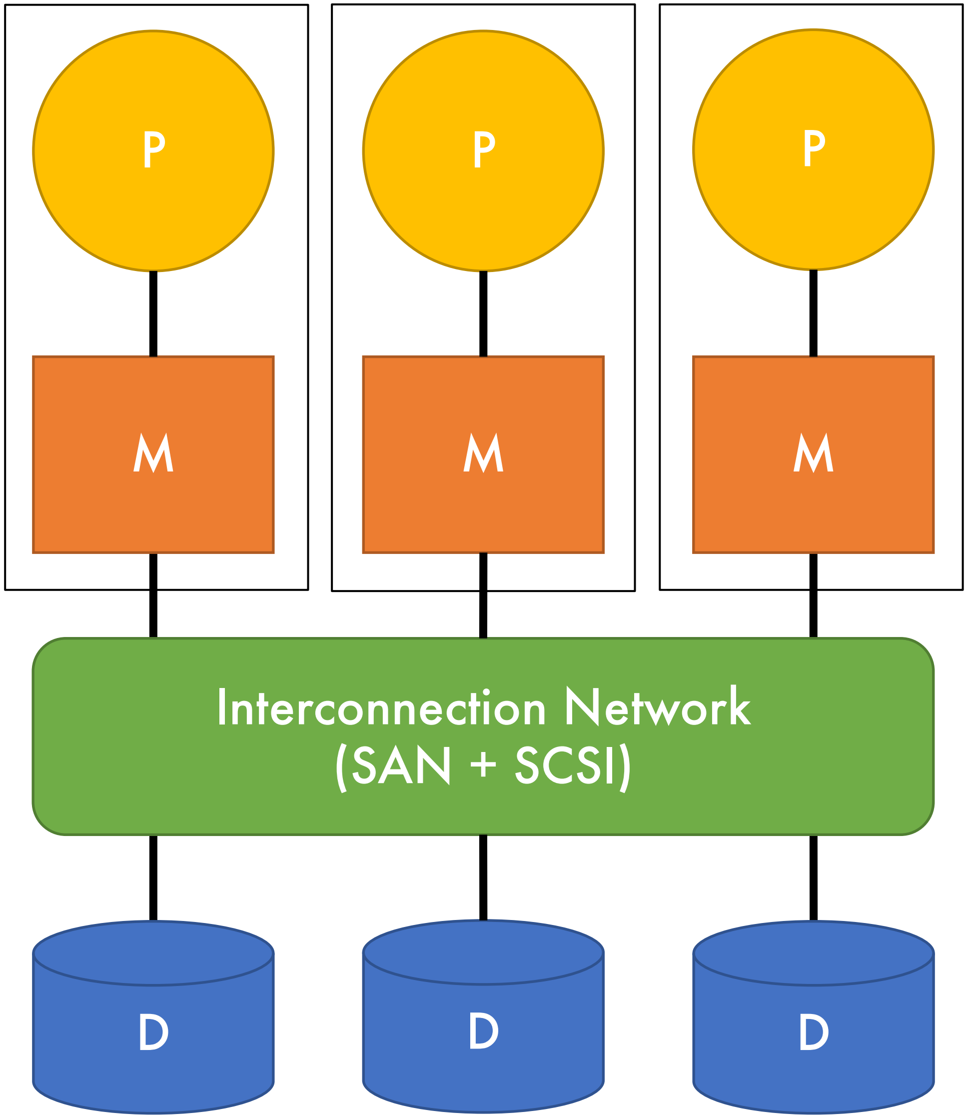
Data model?	Relational
Scaleup goal?	OLAP
Architecture?	

Shared-Memory Architecture



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

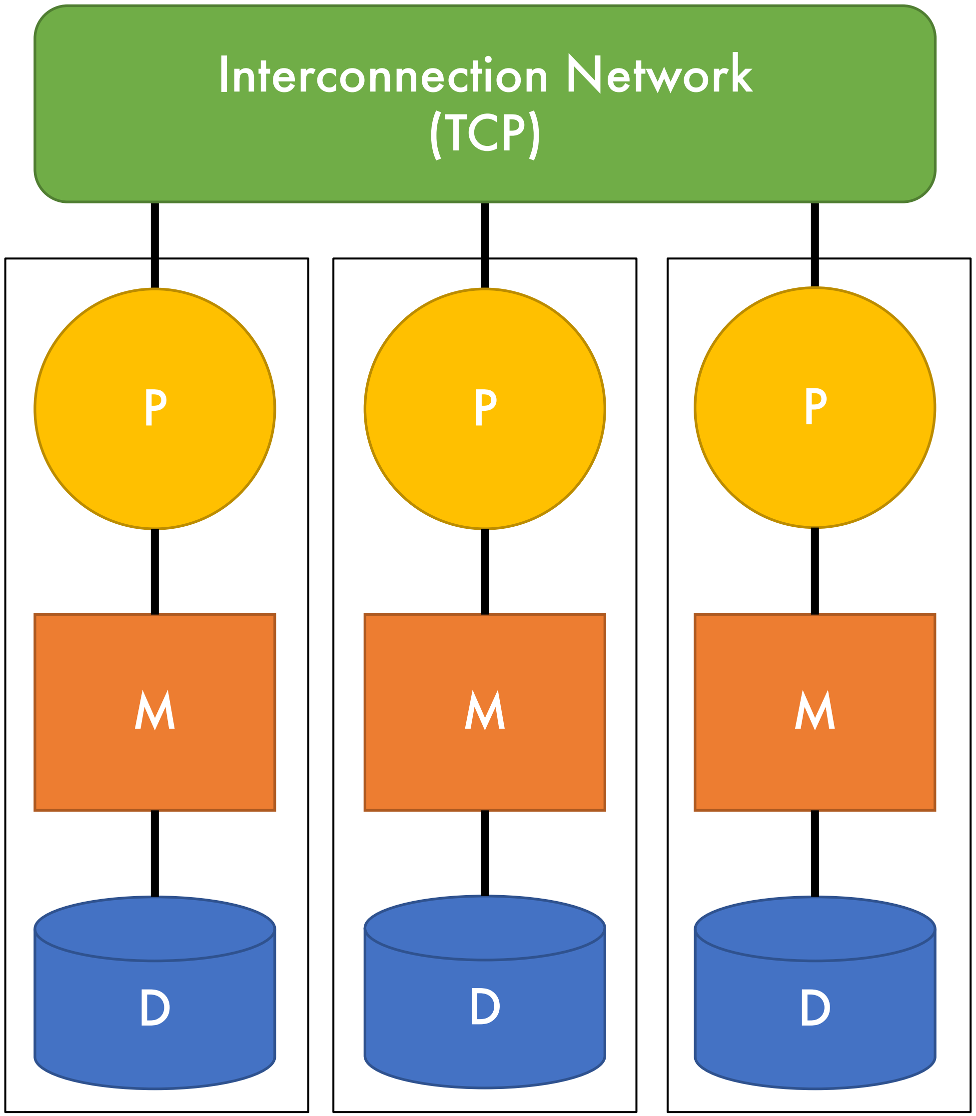
Shared-Disk Architecture



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

ORACLE DATABASE

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

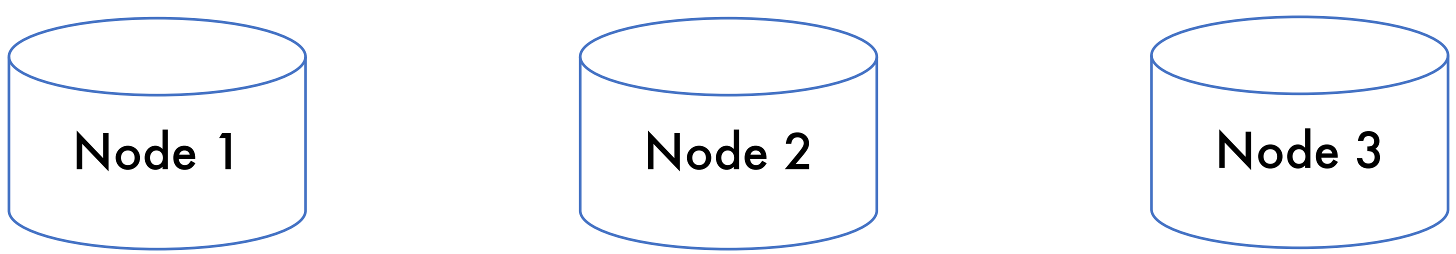
Building Our Parallel DBMS

Data model?	Relational
Scaleup goal?	OLAP
Architecture?	Shared-Nothing

Shared-Nothing Execution Basics

- Multiple DBMS instances (= processes) also called "nodes" execute on machines in a cluster
  - One node plays role of the coordinator
  - Other nodes play role of workers

- Workers execute queries
  - Typically **all workers execute the same plan**
  - Workers can execute multiple queries at the same time

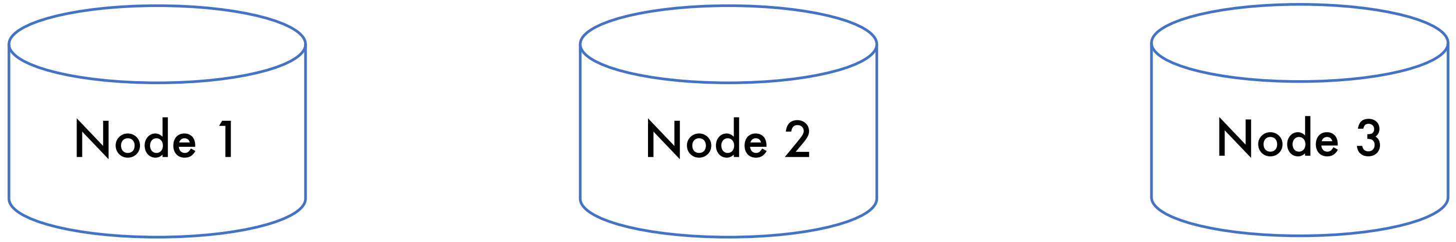


Shared-Nothing Database

We will assume a system that consists of multiple commodity machines on a common network

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

The answer will influence our execution techniques

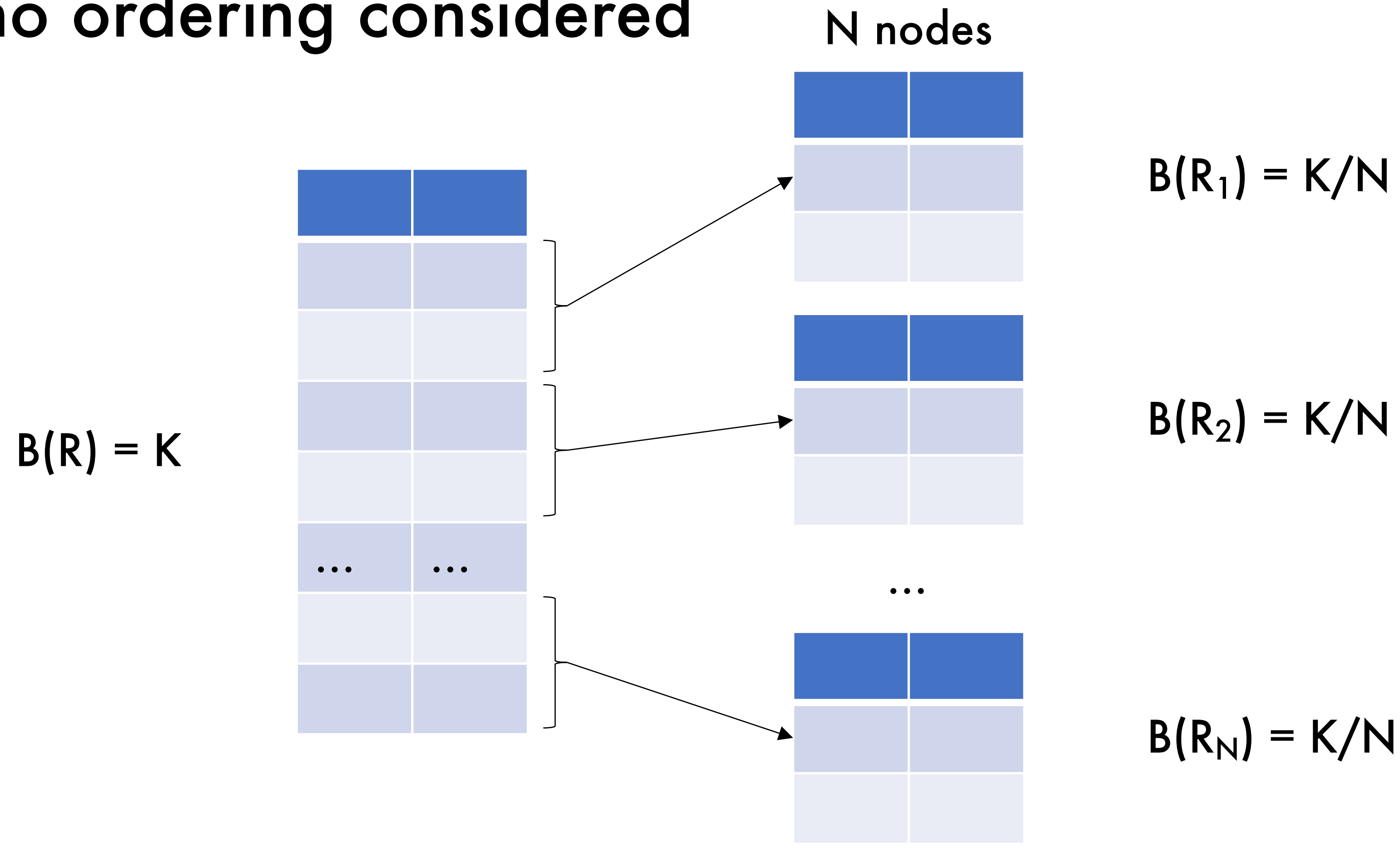


Option 1: Unpartitioned Table

- Entire table on just one node in the system
- Will bottleneck any query we need to run in parallel
- We choose partitioning scheme to divide rows among machines

Option 2: Block Partitioning

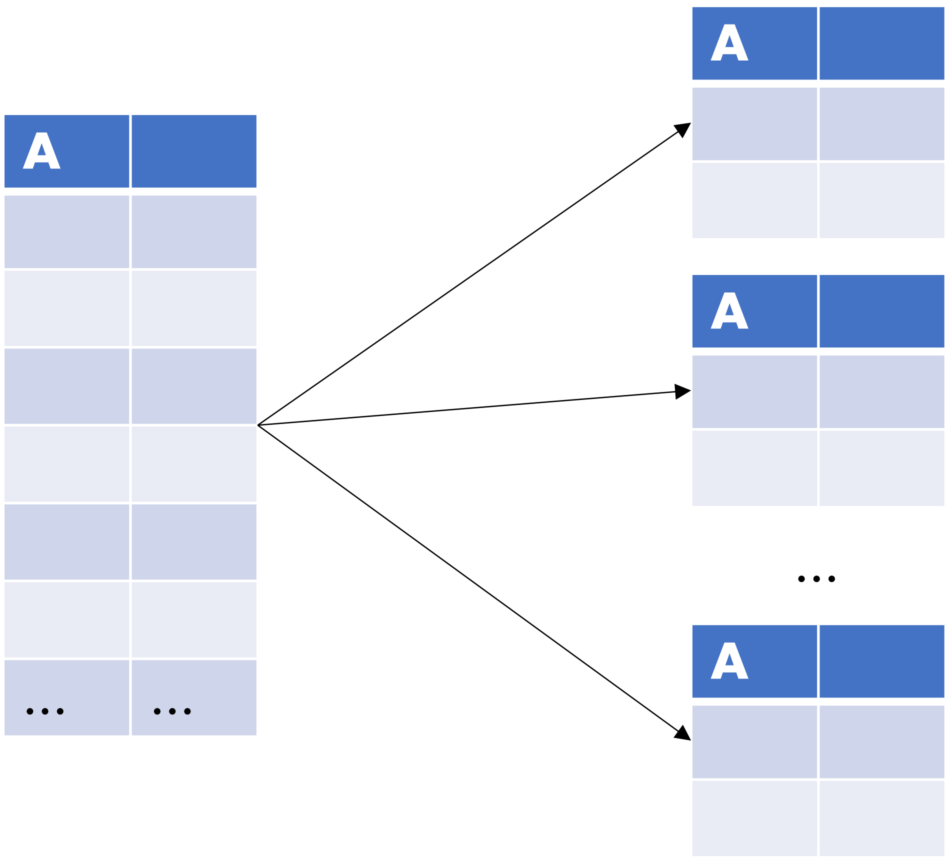
Tuples are horizontally (row) partitioned by raw size with no ordering considered



Option 3: Range Partitioning

Node contains tuples in chosen attribute ranges

N nodes



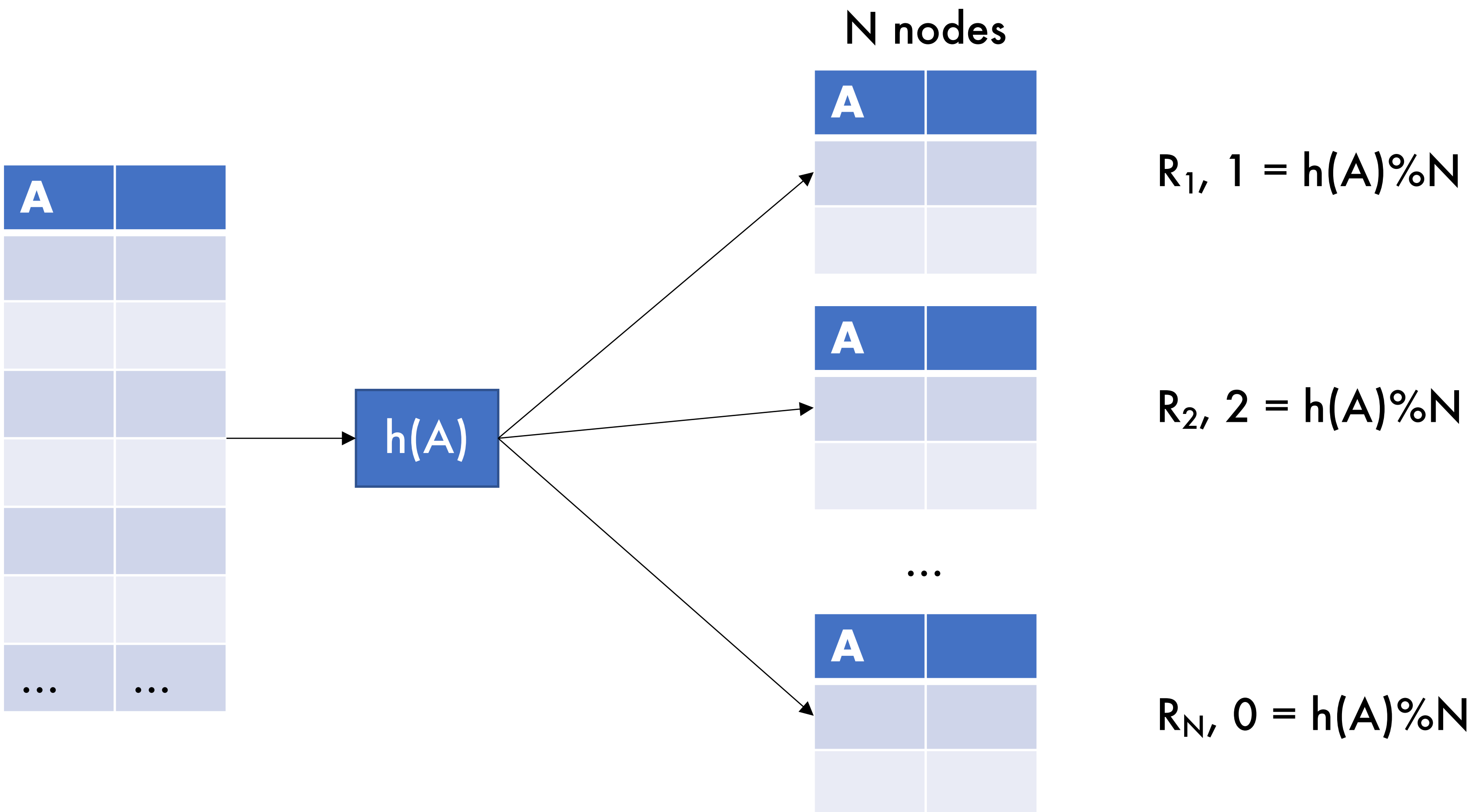
R_1, $-\text{inf} < A <= v_1$

R_2, $v_1 < A <= v_2$

R_N, $v_N < A < \text{inf}$

Option 4: Hash Partitioning

Node contains tuples with chosen attribute hashes



Skew: 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

Parallel Selection

Assume:
R is block partitioned

```
SELECT *
  FROM R
 WHERE A = 2
```

A	…
1	…
2	…

Node 1

A	…
2	…
3	…

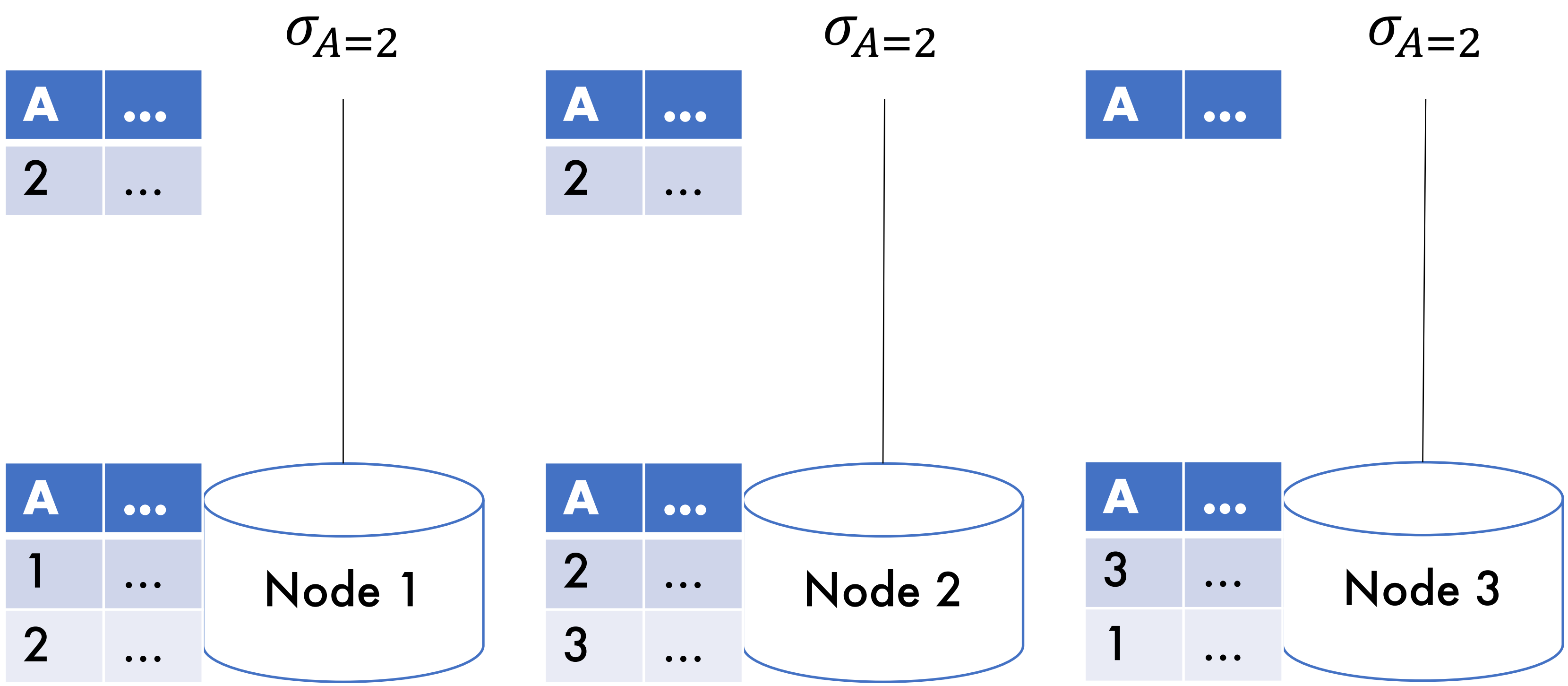
Node 2

A	…
3	…
1	…

Node 3

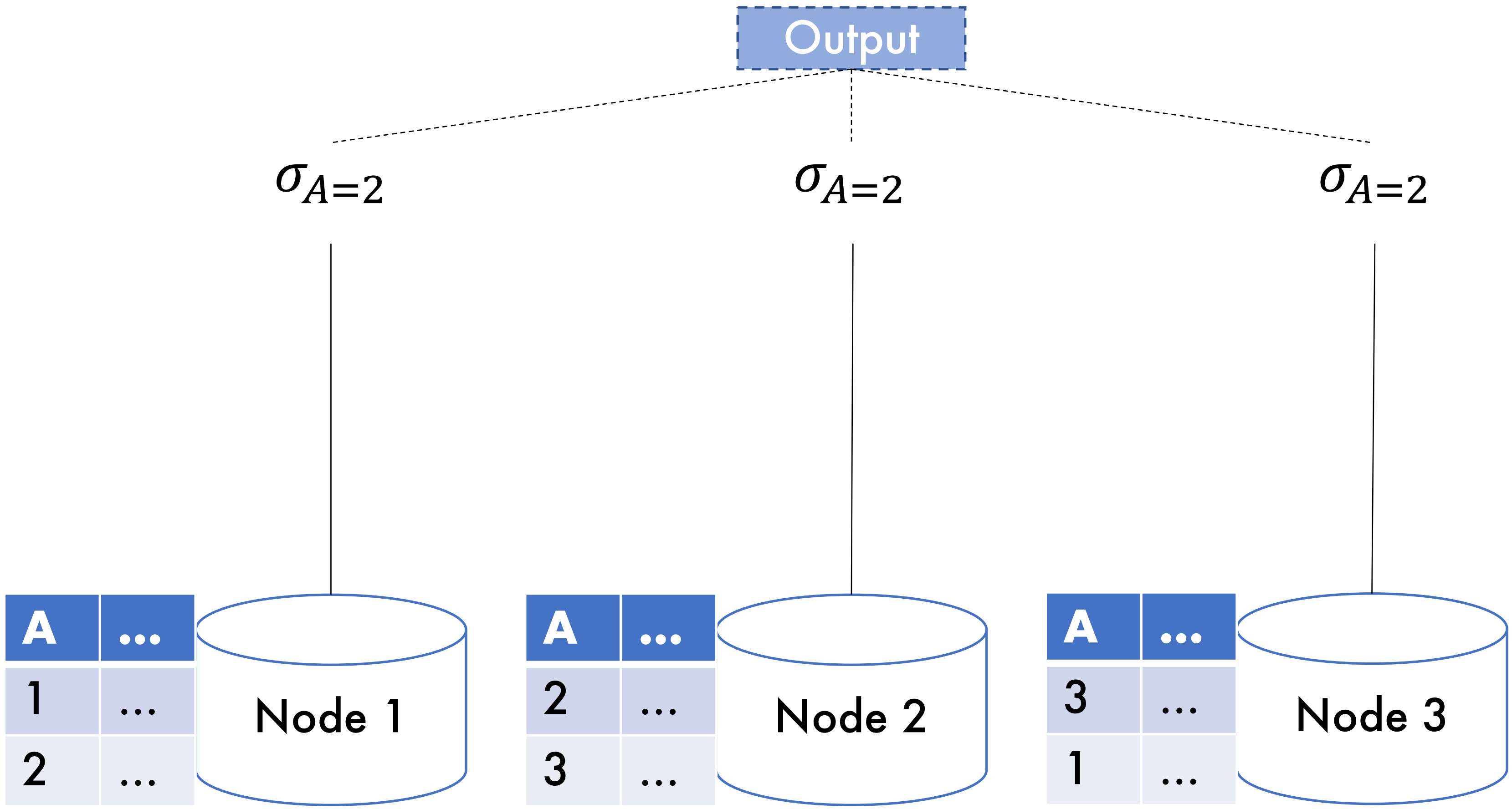
Parallel Selection

```
SELECT *
  FROM R
 WHERE A = 2
```



Implicit Union

Parallel query plans implicitly union at the end



Parallel Selection

Compute $\sigma_{A=v}(R)$, or $\sigma_{v1<A<v2}(R)$

- On a conventional database: cost = **B(R)**

Q: What is the cost on each node for a database with N nodes ?

A:

Parallel Selection

Compute $\sigma_{A=v}(R)$, or $\sigma_{v1<A<v2}(R)$

- On a conventional database: cost = **B(R)**

Q: What is the cost on each node for a database with N nodes ?

A: B(R) / N block reads on each node

Partitioned Aggregation

Assume:
R is block partitioned

```
SELECT R.A, sum(…)
  FROM R
 GROUP BY R.A
```

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

A	…
1	…
2	…

Node 1

A	…
2	…
3	…

Node 2

A	…
3	…
1	…

Node 3

Partitioned Aggregation

Assume:
R is block partitioned

```
SELECT R.A, sum(…)
  FROM R
 GROUP BY R.A
```

A	…
1	…
1	…

$\gamma_{R.A}$

A	…
2	…
2	…

$\gamma_{R.A}$

A	…
3	…
3	…

$\gamma_{R.A}$

A	…
1	…
2	…

Node 1

A	…
2	…
3	…

Node 2

A	…
3	…
1	…

Node 3

Partitioned Aggregation

1. Hash shuffle tuples

Assume:
R is block partitioned

```
SELECT R.A, sum(…)
  FROM R
 GROUP BY R.A
```

A	…
1	…
1	…

$\gamma_{R.A}$

A	…
2	…
2	…

$\gamma_{R.A}$

A	…
3	…
3	…

$\gamma_{R.A}$

A	…
1	…
2	…

Node 1

A	…
2	…
3	…

Node 2

A	…
3	…
1	…

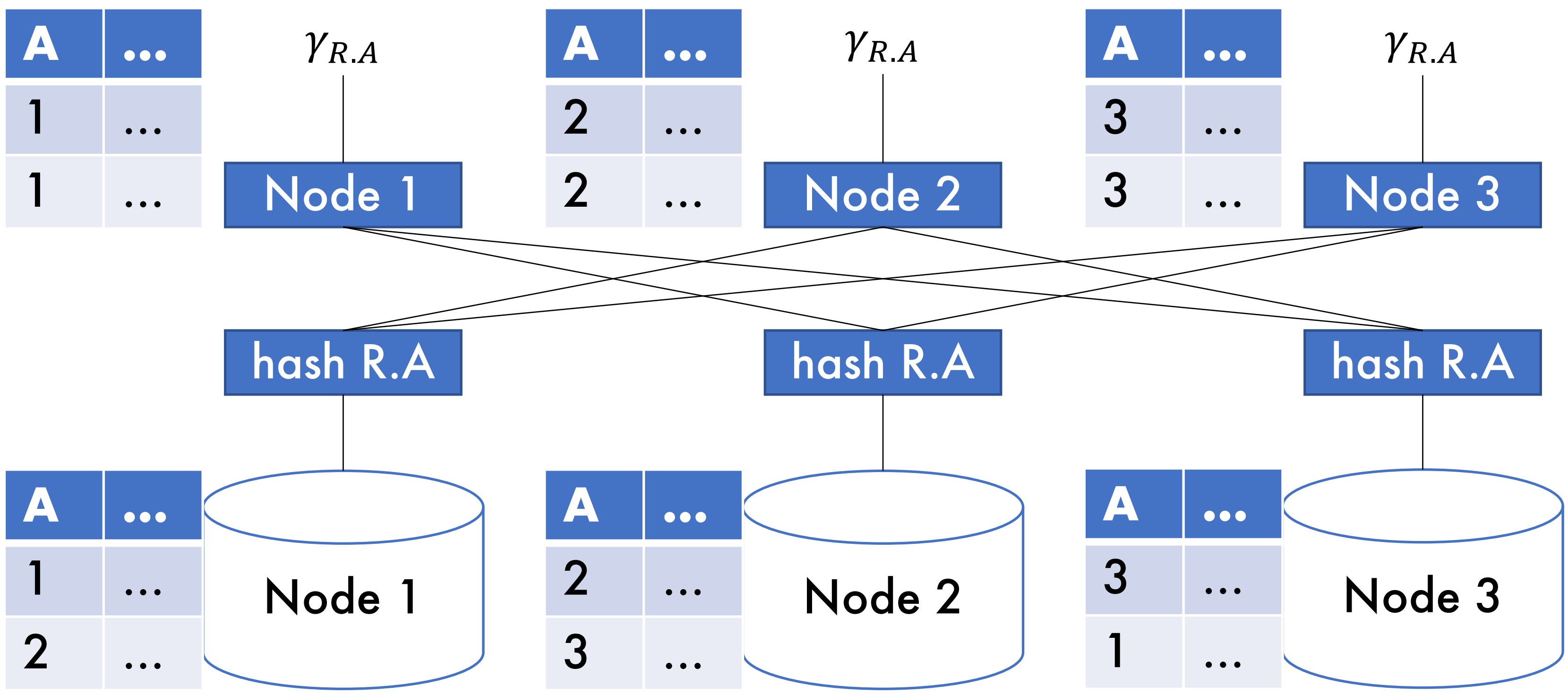
Node 3

Partitioned Aggregation

1. Hash shuffle tuples

Assume:
R is block partitioned

```
SELECT R.A, sum(…)
  FROM R
 GROUP BY R.A
```

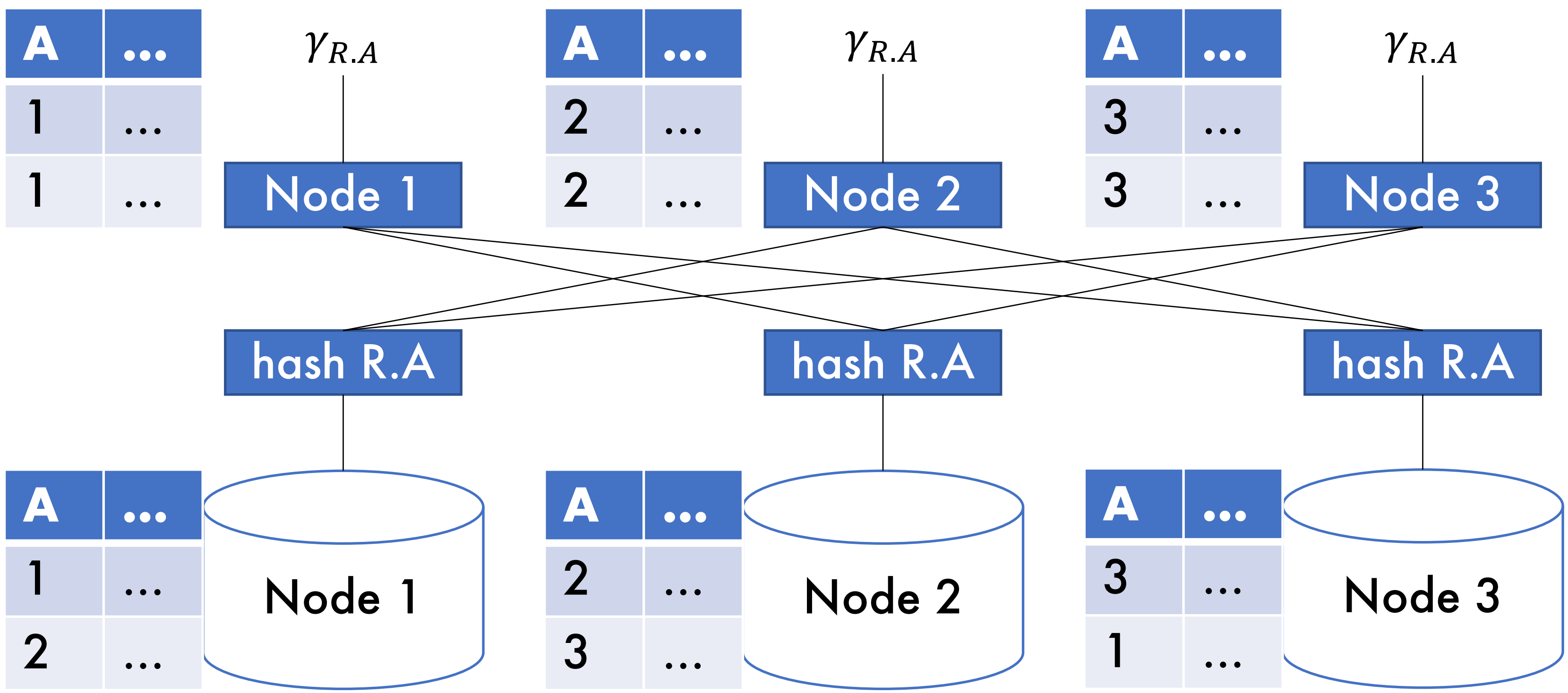


Partitioned Aggregation

1. Hash shuffle tuples
2. Local aggregation

Assume:
R is block partitioned

```
SELECT R.A,sum(…)
  FROM R
 GROUP BY R.A
```



Partition Aggregation: Summary

```
Select A, sum(B) from R group by A
```

- **Case 1**: R is partitioned on A
  - Do the group-by locally; done.

Partition Aggregation: Summary

```
Select A, sum(B) from R group by A
```

- **Case 1**: R is partitioned on A
  - Do the group-by locally; done.

- **Case 2**: R is partitioned on something else
  - Naïve: reshuffle on A, then do as in case 1

Partition Aggregation: Summary

```
Select A, sum(B) from R group by A
```

- **Case 1**: R is partitioned on A
  - Do the group-by locally; done.
- **Case 2**: R is partitioned on something else
  - Naïve: reshuffle on A, then do as in case 1
  - Better: do a *local* group-by-sum (reduces size), then reshuffle on A and do a second group-by

Partition Aggregation: Summary

```
Select A, sum(B) from R group by A
```

- **Case 1**: R is partitioned on A
  - Do the group-by locally; done.
- **Case 2**: R is partitioned on something else
  - Naïve: reshuffle on A, then do as in case 1
  - Better: do a *local* group-by-sum (reduces size), then reshuffle on A and do a second group-by

$$\gamma_{A,sum(B)}(R_1 \cup R_2 \cup \cdots \cup R_N)$$
$$= \gamma_{A,sum(B)}(\gamma_{A,sum(B)}(R_1) \cup \cdots \cup \gamma_{A,sum(B)}(R_N))$$

Partition Aggregation: Summary

```
Select A, sum(B) from R group by A
```

- **Case 1**: R is partitioned on A
  - Do the group-by locally; done.

- **Case 2**: R is partitioned on something else
  - Naïve: reshuffle on A, then do as in case 1
  - Better: do a *local* group-by-sum (reduces size), then reshuffle on A and do a second group-by

"Combiners" in MapReduce

$$\gamma_{A,sum(B)}(R_1 \cup R_2 \cup \cdots \cup R_N)$$
$$= \gamma_{A,sum(B)}(\gamma_{A,sum(B)}(R_1) \cup \cdots \cup \gamma_{A,sum(B)}(R_N))$$

Basic Parallel GroupBy

Can we do partial aggregate before reshuffle?

- Sum?
- Count?
- Avg?
- Max?
- Median?

Basic Parallel GroupBy

Can we do partial aggregate before reshuffle?

- Sum?
- Count?
- Avg?
- Max?
- Median?

Distributive	Algebraic	Holistic
$sum(a_1+a_2+\ldots+a_9)=$ $sum(sum(a_1+a_2+a_3)+$ $sum(a_4+a_5+a_6)+$ $sum(a_7+a_8+a_9))$	avg(B) = sum(B)/count(B)	median(B)

Basic Parallel GroupBy

Can we do partial aggregate before reshuffle?

- Sum?
- Count?
- Avg?
- Max?
- Median?

Distributive	Algebraic	Holistic
$sum(a_1+a_2+\ldots+a_9)=$ $sum(sum(a_1+a_2+a_3)+$ $sum(a_4+a_5+a_6)+$ $sum(a_7+a_8+a_9))$	avg(B) = sum(B)/count(B)	median(B)

Yes for Distributive

Yes for Algebraic (just compute two aggregates)

Hash-Partitioned Parallel Join

$R(A,B) \bowtie_{B=C} S(C,D)$

Hash-Partitioned Parallel Join

$R(A,B) \bowtie_{B=C} S(C,D)$

- Step 1: reshuffle R on B; reshuffle S on C

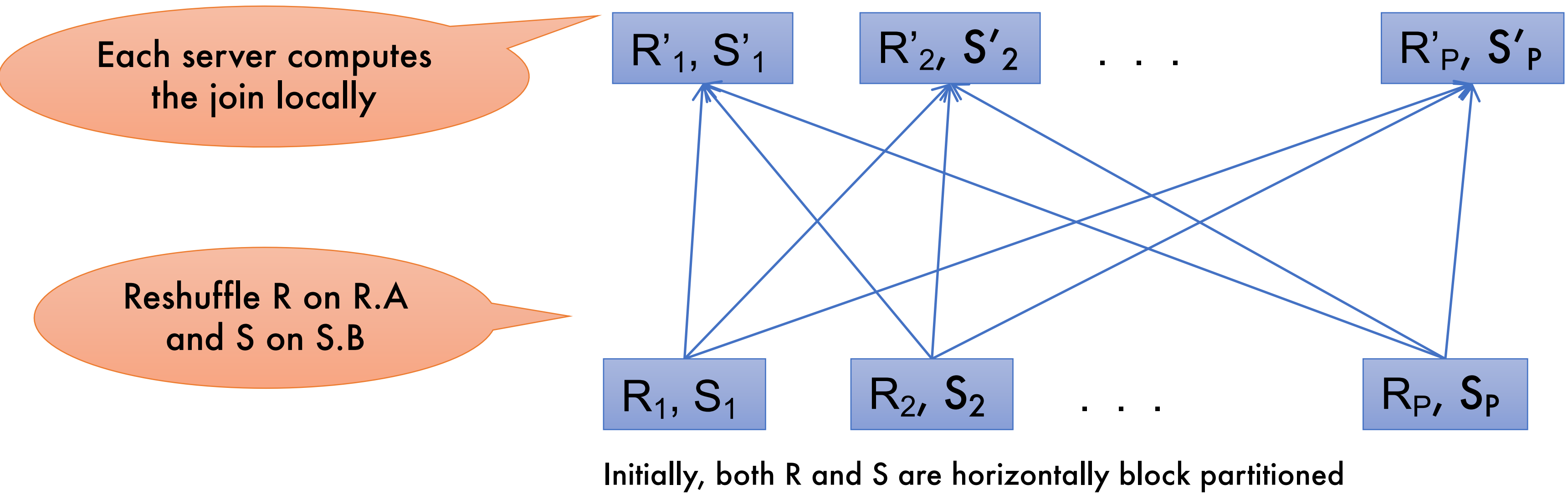
Hash-Partitioned Parallel Join

$R(A,B) \bowtie_{B=C} S(C,D)$

- Step 1: reshuffle R on B; reshuffle S on C
- Step 2: join locally each fragment $R_i \bowtie S_i$

Hash-Partitioned Parallel Join

$R(A,B) \bowtie_{B=C} S(C,D)$



Hash-Partitioned Parallel Join: Recap

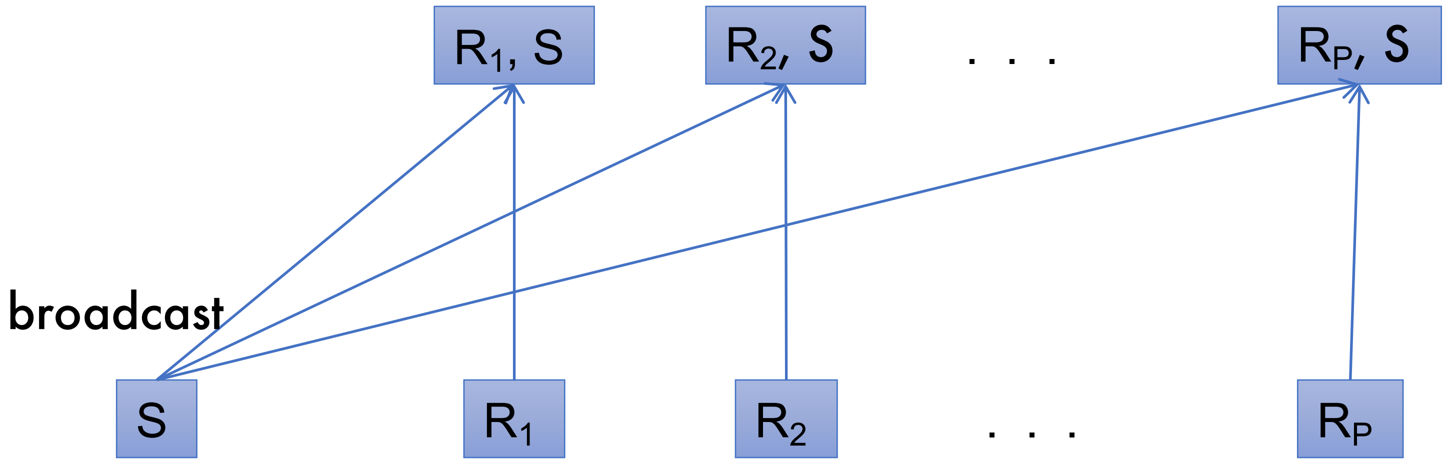
- Step 1
  - Every server holding any chunk of R partitions its chunk using a hash function h(t.A) mod P
  - Every server holding any chunk of S partitions its chunk using a hash function h(t.B) mod P
- Step 2:
  - Each server computes the join of its local fragment of R with its local fragment of S

Optimization for Small Relations

When joining R and S

- If |R| >> |S|
  - Leave R where it is
  - Replicate entire S relation across nodes

- Also called a small join or a broadcast join



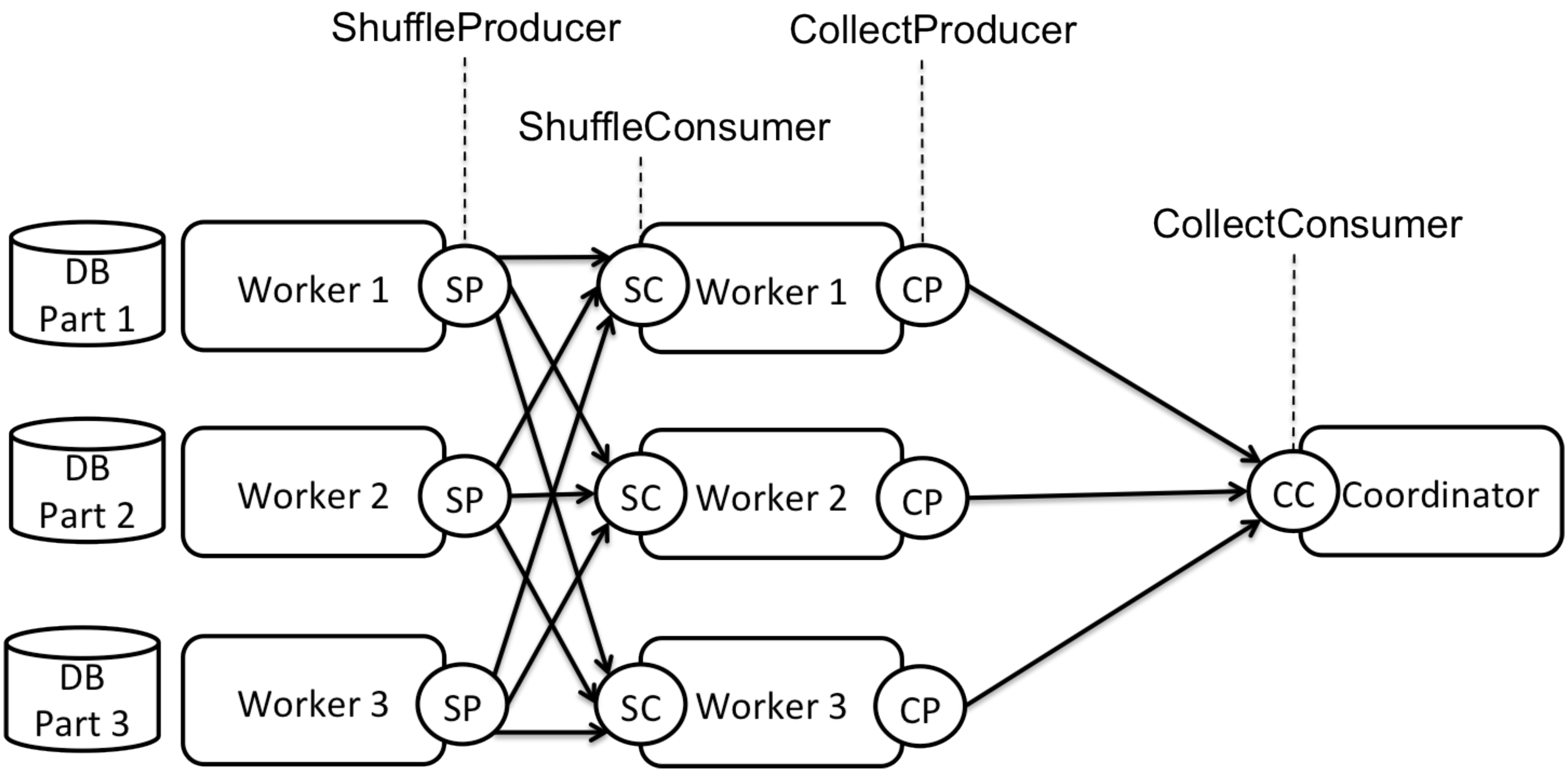
Parallel Query Evaluation

New operator: Shuffle

- Serves to re-shuffle data between processes
  - Handles data routing, buffering, and flow control

- Two parts: ShuffleProducer and ShuffleConsumer
- Producer:
  - Pulls data from child operator and sends to n consumers
  - Producer acts as driver for operators below it in query plan
- Consumer:
  - Buffers input data from n producers and makes it available to operator through getNext() interface

Parallel Query Execution

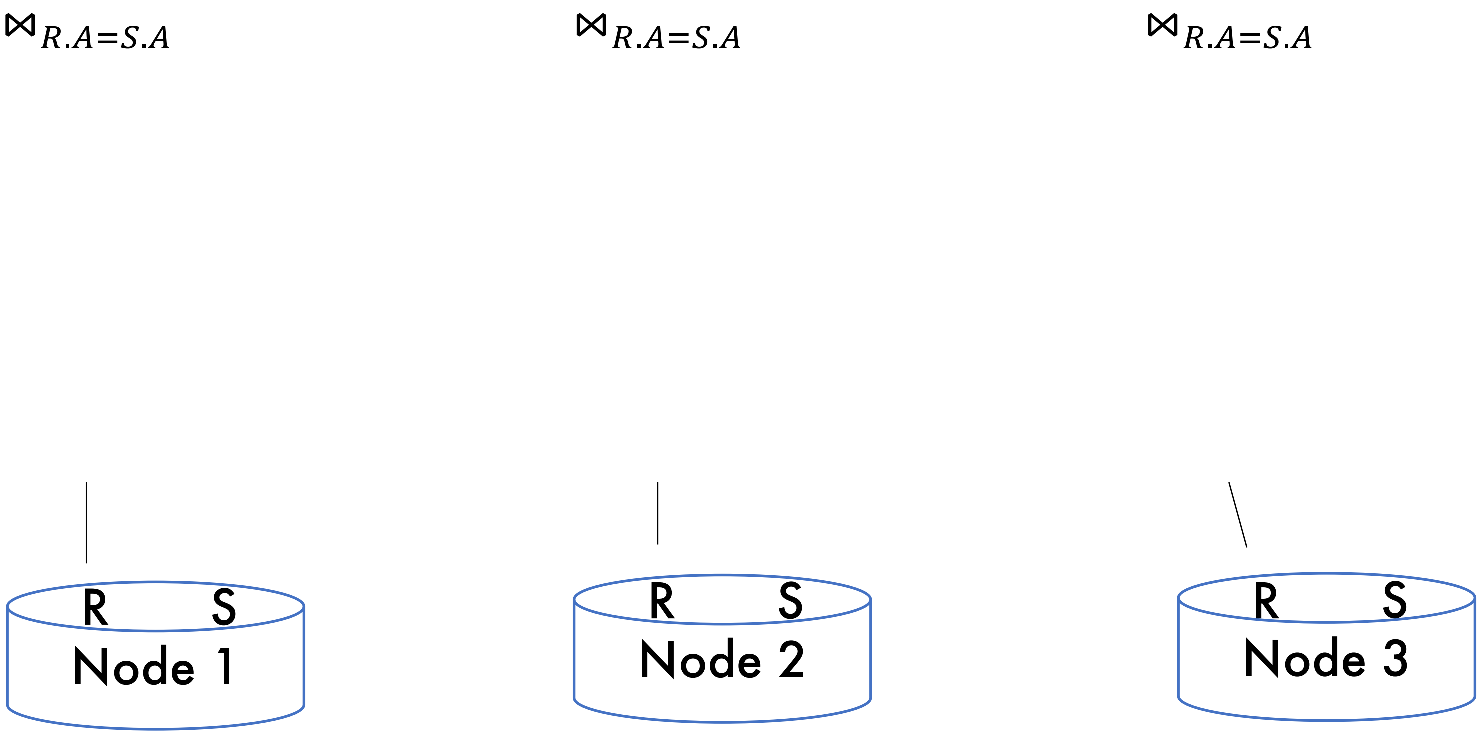


Partitioned Hash Equijoin Algorithm

1. Hash shuffle tuples on join attributes

Assume:
R and S are block partitioned

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

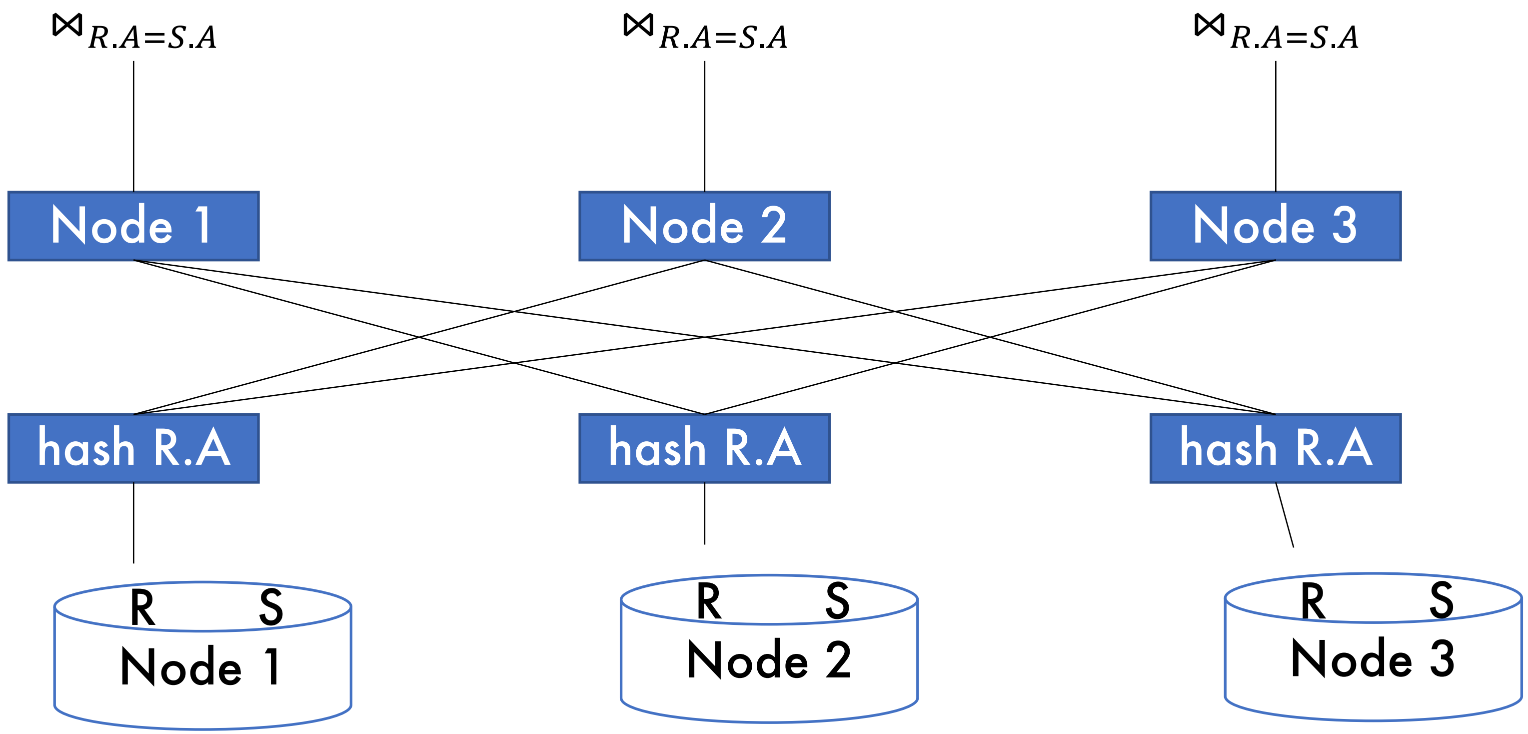


Partitioned Hash Equijoin Algorithm

1. Hash shuffle tuples on join attributes

Assume:
R and S are block partitioned

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

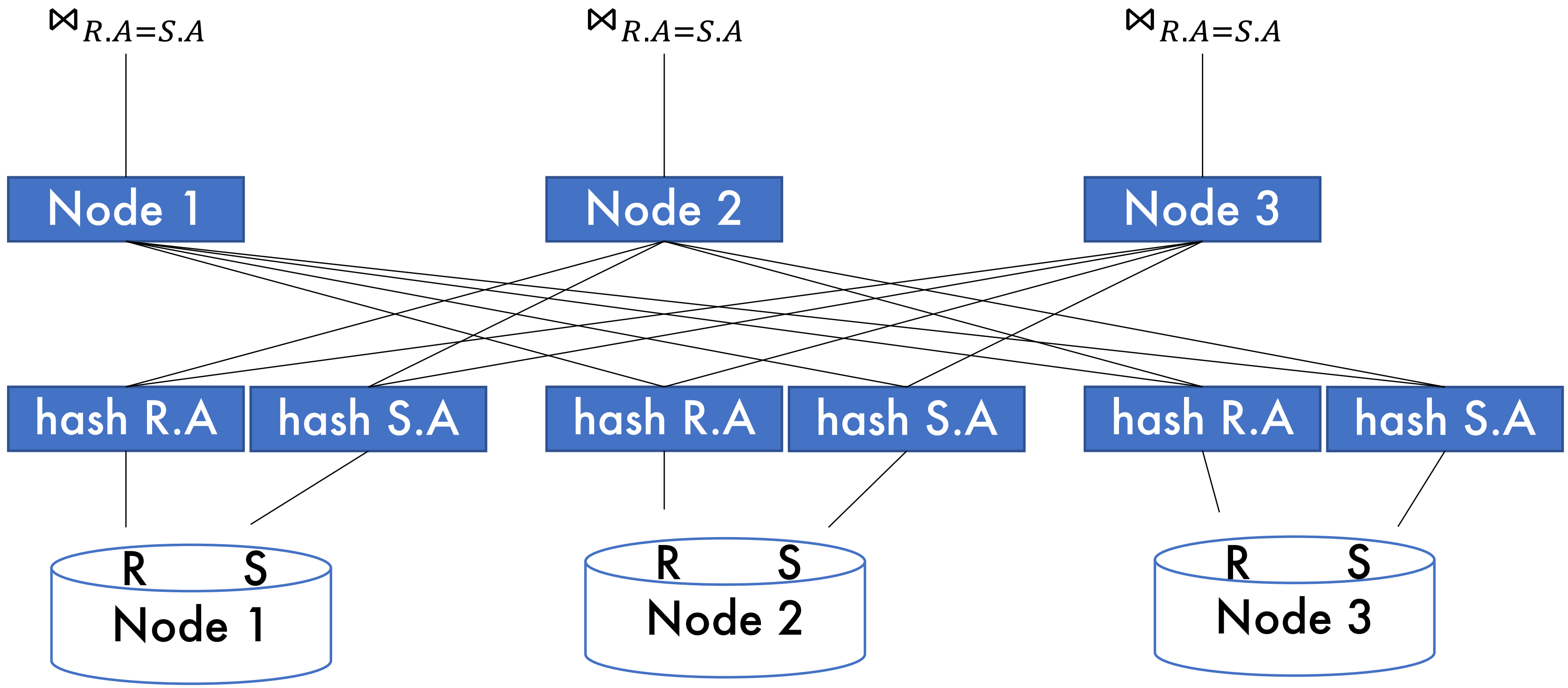


Partitioned Hash Equijoin Algorithm

1. Hash shuffle tuples on join attributes

Assume:
R and S are block partitioned

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

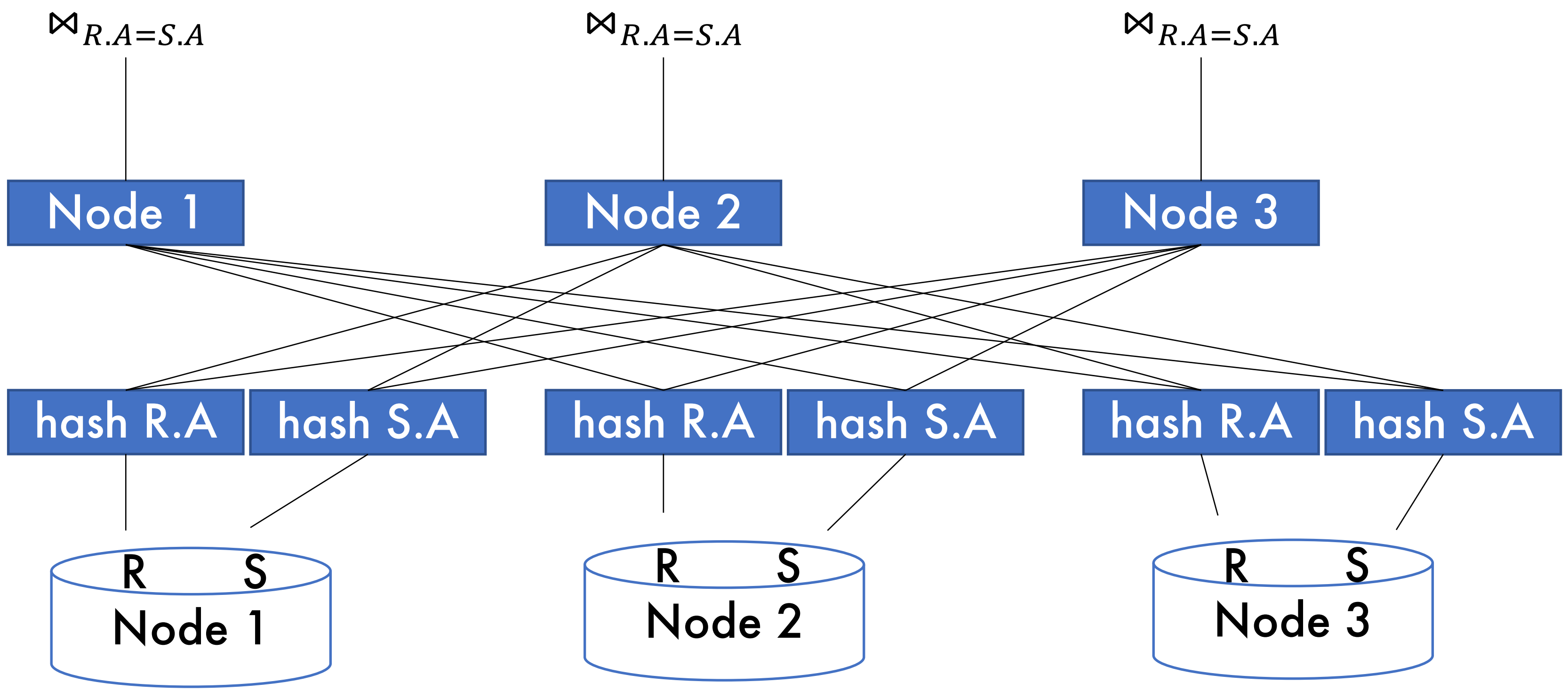


Partitioned Hash Equijoin Algorithm

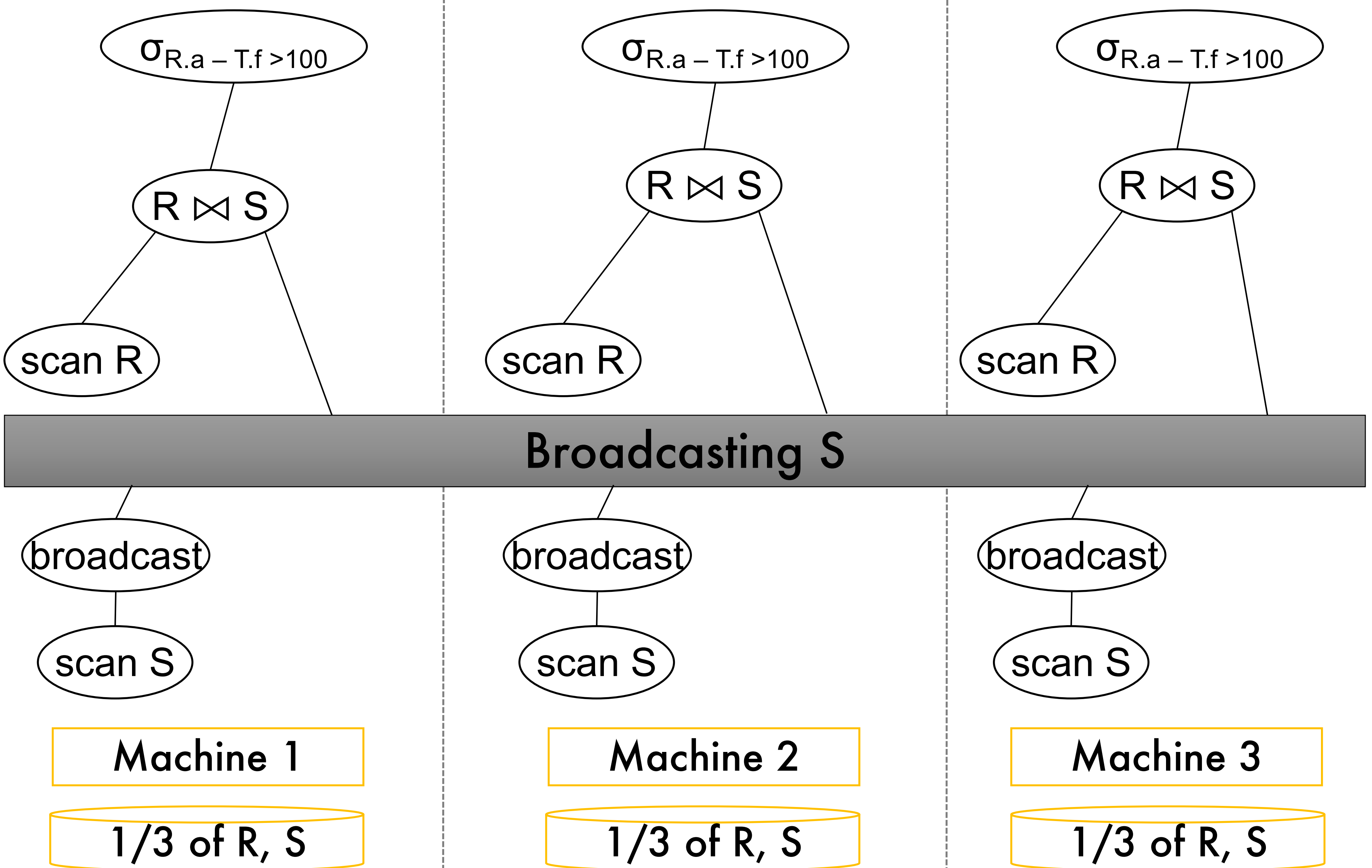
1. Hash shuffle tuples on join attributes
2. Local join

Assume:
R and S are block partitioned

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

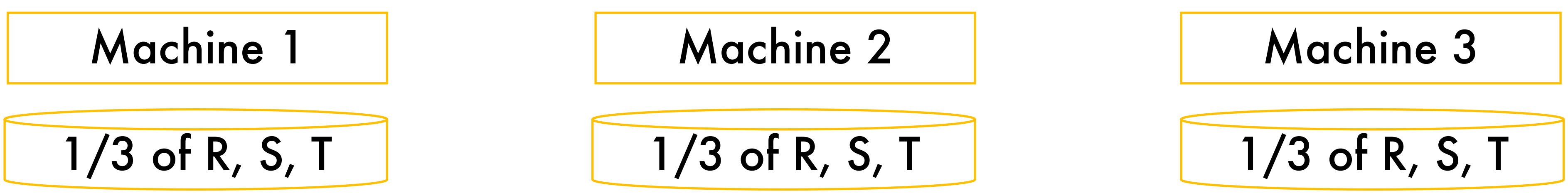


Broadcast Join Example

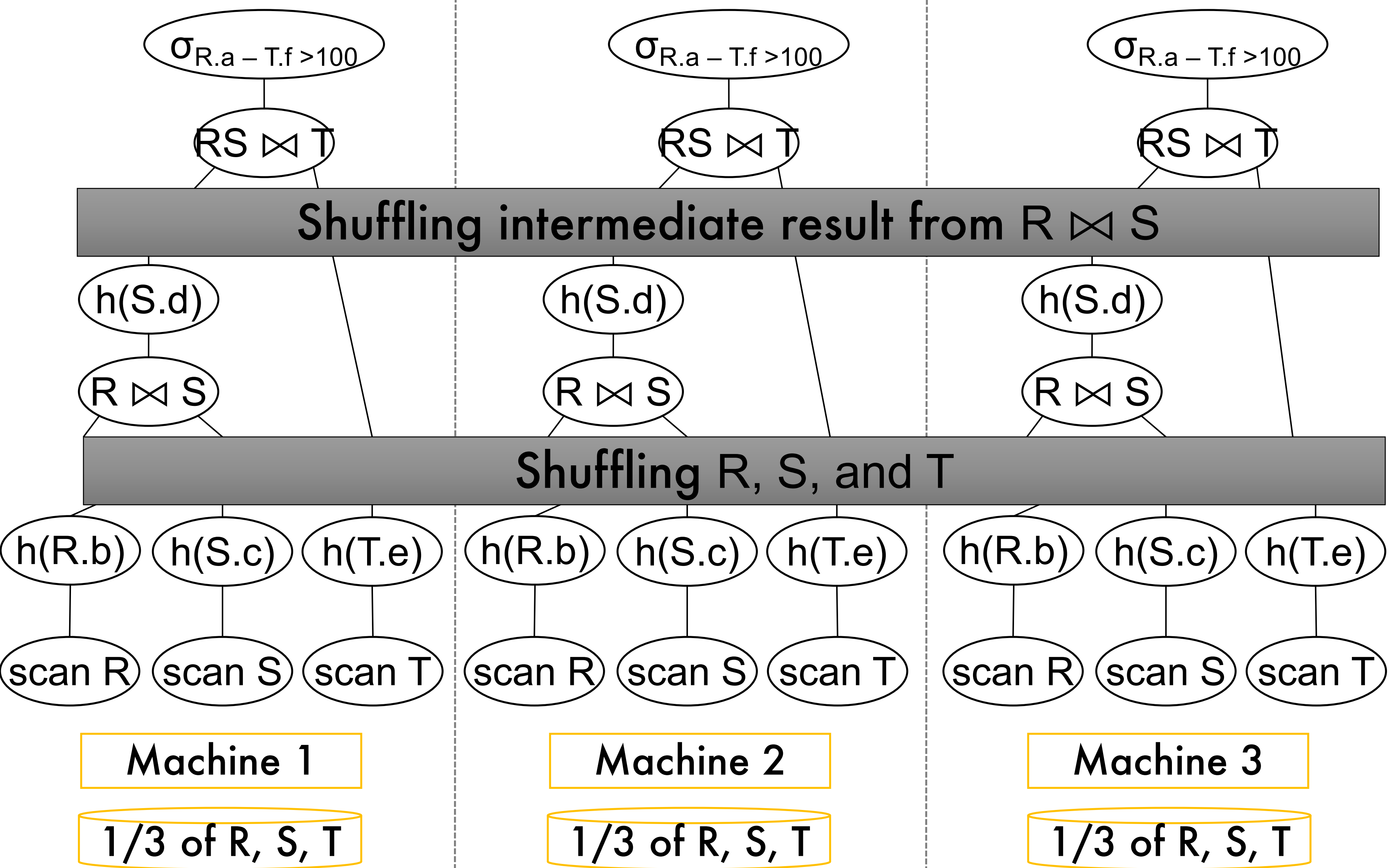


Example

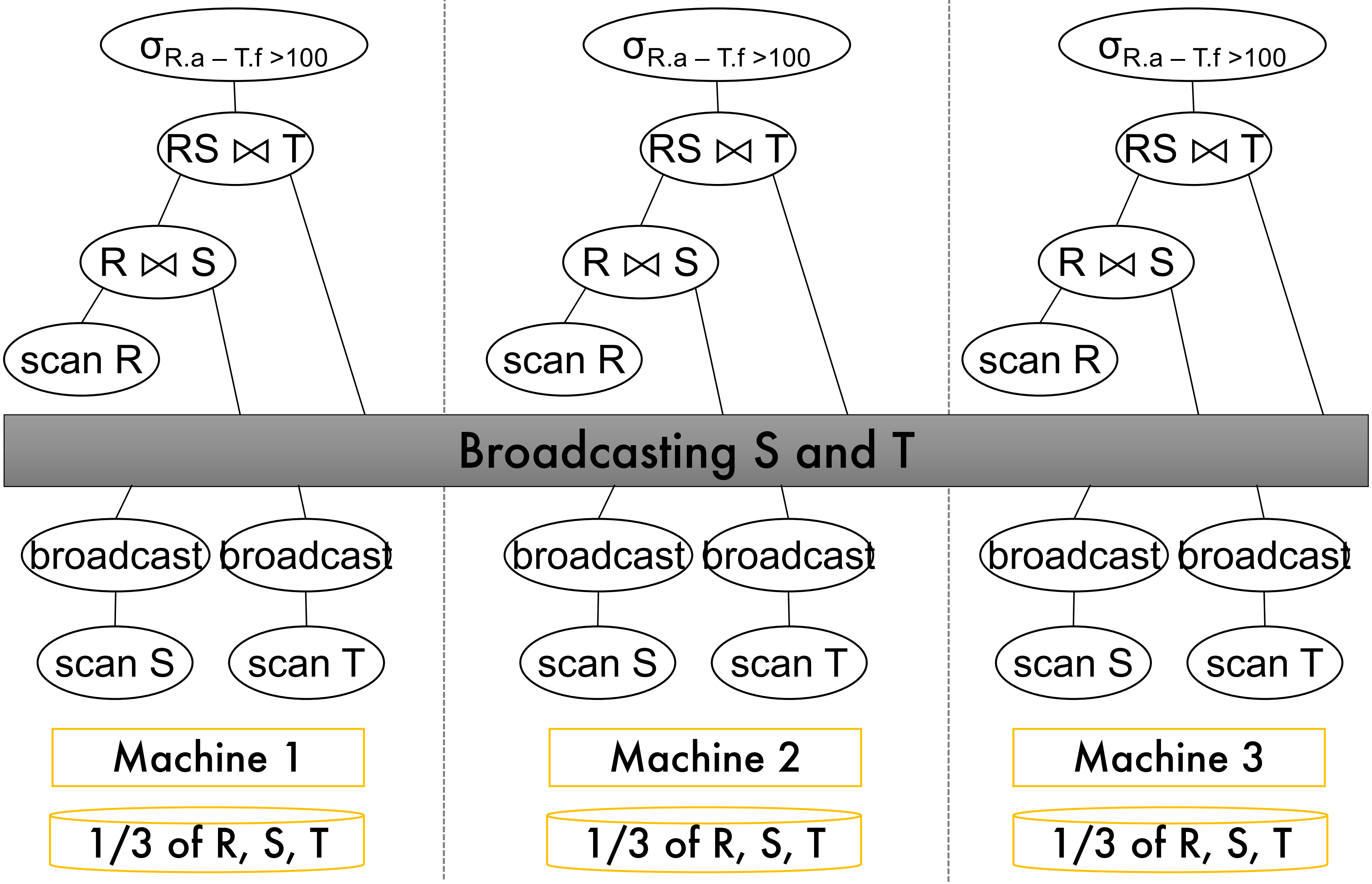
```
SELECT *
FROM R, S, T
WHERE R.b = S.c AND S.d = T.e AND (R.a - T.f) > 100
```



Example with Two Joins



Example with Two Joins



Speedup and Scaleup

- Consider:
  - Query: $\gamma_{A,sum(C)}(R)$
  - Runtime: dominated by reading chunks from disk
- Speedup: If we double the number of nodes P, what is the new running time?
- Scaleup: If we double both P and the size of R, what is the new running time?

Speedup and Scaleup

- Consider:
  - Query: $\gamma_{A,sum(C)}(R)$
  - Runtime: dominated by reading chunks from disk
- Speedup: If we double the number of nodes P, what is the new running time?
  - **Half** (each server holds ½ as many chunks)
- Scaleup: If we double both P and the size of R, what is the new running time?

Speedup and Scaleup

- Consider:
  - Query: $\gamma_{A,sum(C)}(R)$
  - Runtime: dominated by reading chunks from disk
- Speedup: If we double the number of nodes P, what is the new running time?
  - **Half** (each server holds ½ as many chunks)
- Scaleup: If we double both P and the size of R, what is the new running time?
  - **Same** (each server holds the same # of chunks)

Summary

- With one new operator, we've made SimpleDB an OLAP-ready parallel DBMS!

- Next lecture:
  - Skew handling
  - Algorithm refinements

Justin Biebers Re-visited

Skew:

- Some partitions get more input tuples than others
  - Range-partition instead of hash
  - Some values are very popular: "heavy hitters"
  - Selection before join with different selectivities

- Some partitions generate more output tuples than others

Some Skew Handling Techniques

If using range partition:

- Ensure each range gets same number of tuples
- E.g.: {1, 1, 1, 2, 3, 4, 5, 6 } → [1,2] and [3,6]
- Eq-depth v.s. eq-width histograms

Some Skew Handling Techniques

Create more partitions than nodes

- And be smart about scheduling the partitions
  - E.g. One node ONLY does Justin Biebers
- Note: MapReduce uses this technique

Some Skew Handling Techniques

- Broadcast join: if the join attribute of R is heavily skewed, then broadcast S

- If S is also large, then use "skew-join":
  - Join the heavy hitters in R by broadcasting a fragment of S
  - Join the light hitters of R using hash-partition with the rest of S
  - (next slide)

Some Skew Handling Techniques

Use subset-replicate (a.k.a. "skewedJoin")

- Given $R \bowtie_{A=B} S$
- Given a heavy hitter value R.A = 'v'
  (i.e. 'v' occurs very many times in R)
- Partition R tuples with value 'v' across all nodes
  e.g. block-partition, or hash on other attributes
- Replicate S tuples with value 'v' to all nodes
- R = the build relation
- S = the probe relation