CSE544
Data Management

Lectures 15

Parallel Query Processing

Announcements

- Project proposals due on Friday
- HW4 Datalog due next Tuesday
- Tim Kraska talks next Monday, 9-10am
  - The talk is recommended (not mandatory)
  - I will post the zoom link on Ed

Outline

- Brief discussion of the LSM paper

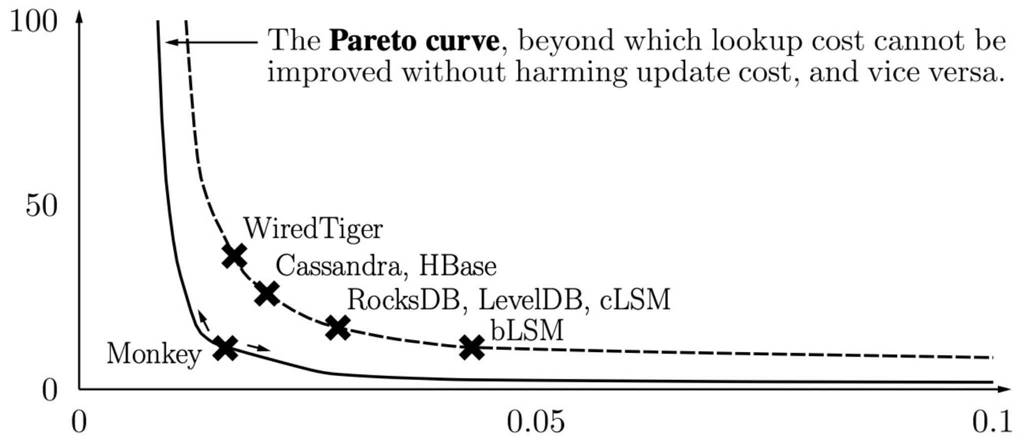
- Parallel Query Processing – Basics

- Parallel Query Processing – Systems
  – Next lecture (Wednesday)

LSM Trees – Review

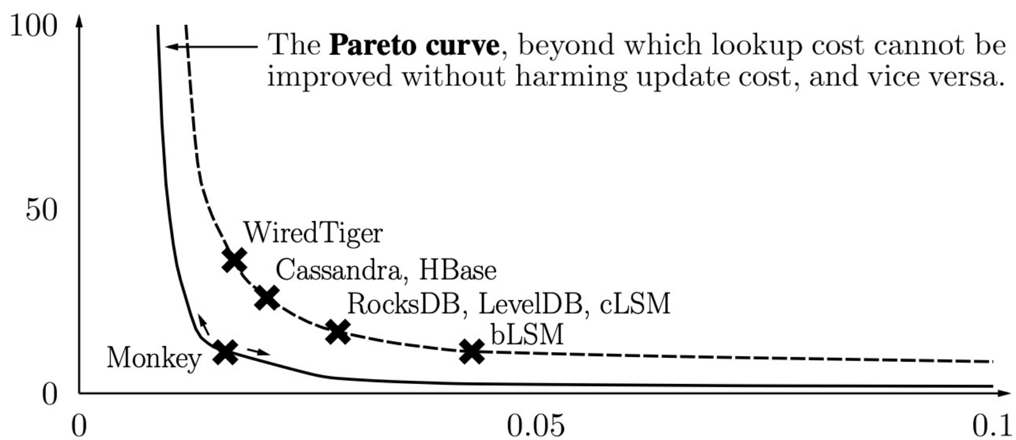
- What is the problem that LSM trees are addressing? And what is their principle?

- What does this graph represent?



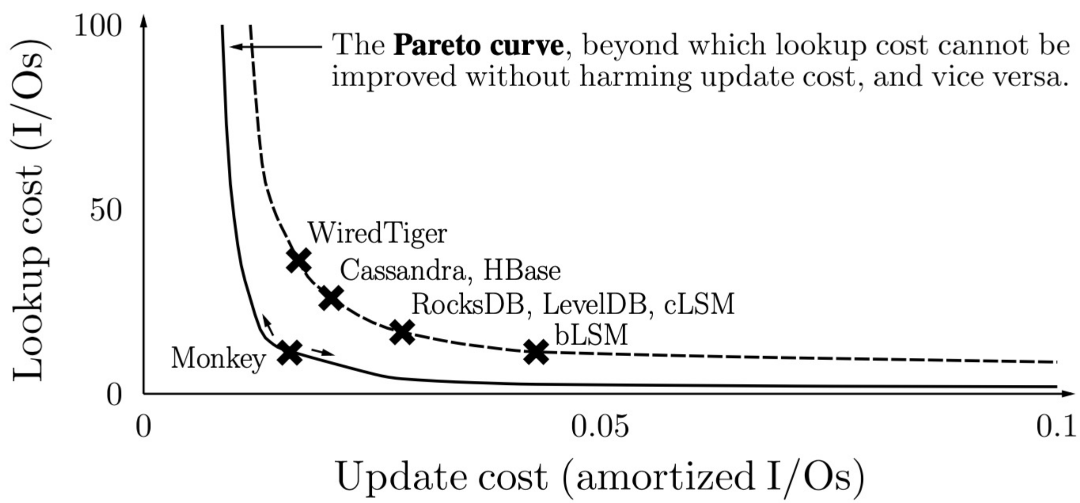
LSM Trees – Review

- What is the problem that LSM trees are addressing? And what is their principle?
  - High throughput updates, compact data representation
  - Principle: buffer updates in main memory, batch-merge
- What does this graph represent?



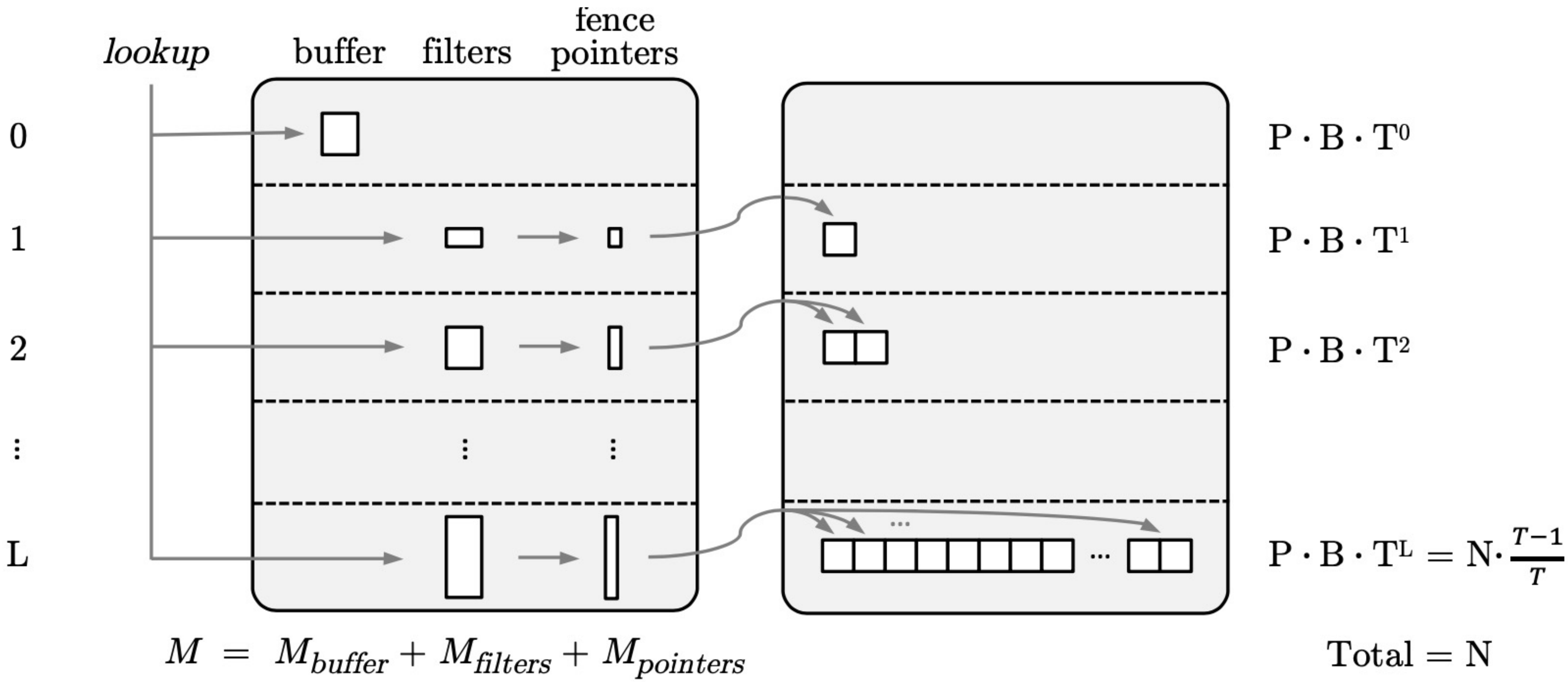
LSM Trees – Review

- What is the problem that LSM trees are addressing? And what is their principle?
  - High throughput updates, compact data representation
  - Principle: buffer updates in main memory, batch-merge
- What does this graph represent?



LSM Trees – Review

- Describe a lookup is an LSM tree



LSM Trees – Review

- What is the False FPR formula in a Bloom filter?
- How do current systems design the sizes M of the Bloom filters?
- Paper says it’s a bad idea. Why?

LSM Trees – Review

- What is the False FPR formula in a Bloom filter?
  - $FPR = e^{-\frac{M}{N}\ln^2 2}$ where
  - M = # bits in the Bloom filter, N = # data entries
- How do current systems design the sizes M of the Bloom filters?

- Paper says it's a bad idea. Why?

LSM Trees – Review

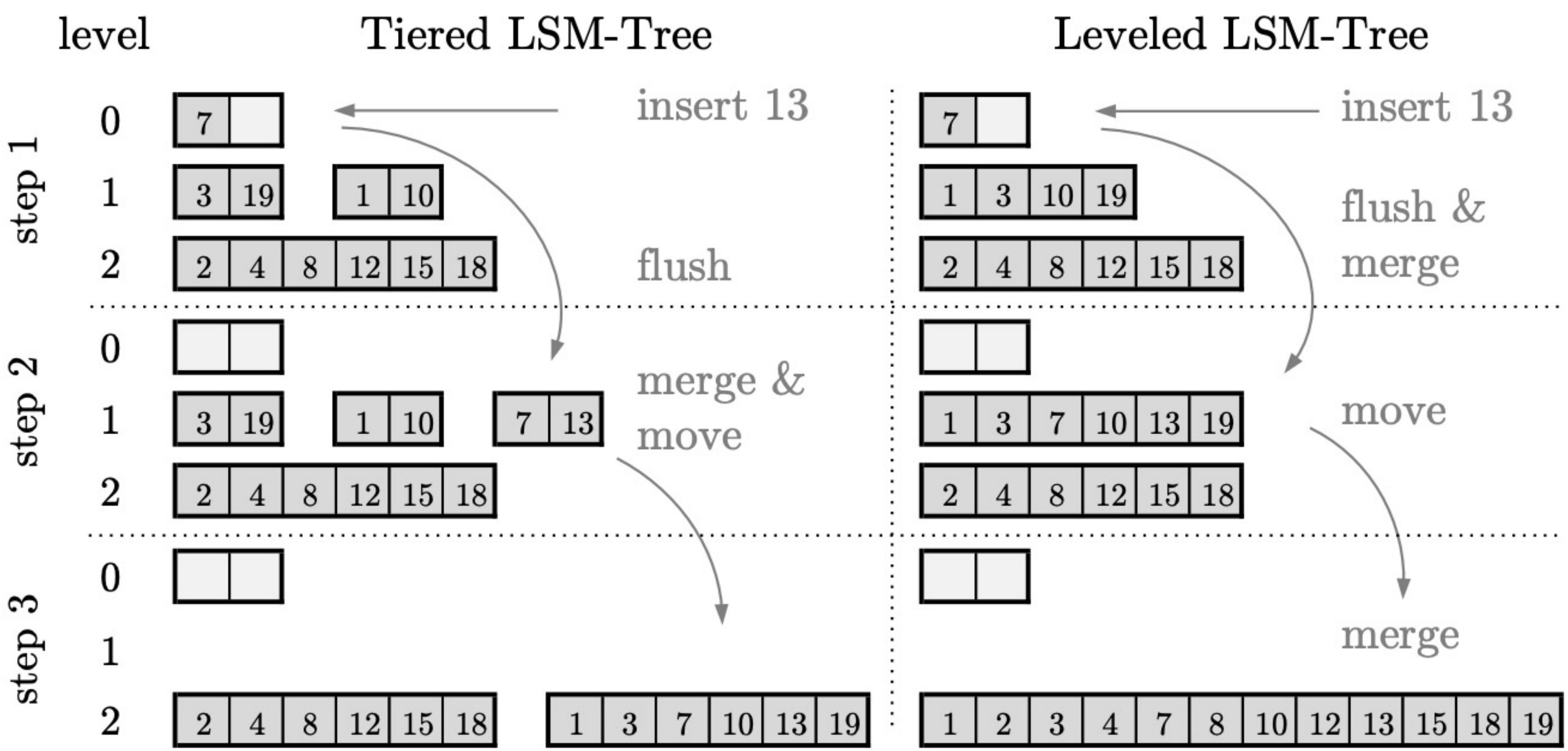
- What is the False FPR formula in a Bloom filter?
  - $FPR = e^{-\frac{M}{N}\ln^2 2}$ where
  - M = # bits in the Bloom filter, N = # data entries
- How do current systems design the sizes M of the Bloom filters?
  - Ensure same FPR at all levels
  - M grows exponentially by factor T
- Paper says it's a bad idea. Why?

LSM Trees – Review

- What is the False FPR formula in a Bloom filter?
  - $FPR = e^{-\frac{M}{N}\ln^2 2}$ where
  - M = # bits in the Bloom filter, N = # data entries
- How do current systems design the sizes M of the Bloom filters?
  - Ensure same FPR at all levels
  - M grows exponentially by factor T
- Paper says it's a bad idea. Why?
  - Every Bloom filter saves only one I/O at each level.
  - Last Bloom filter is larger then other, but same benefit

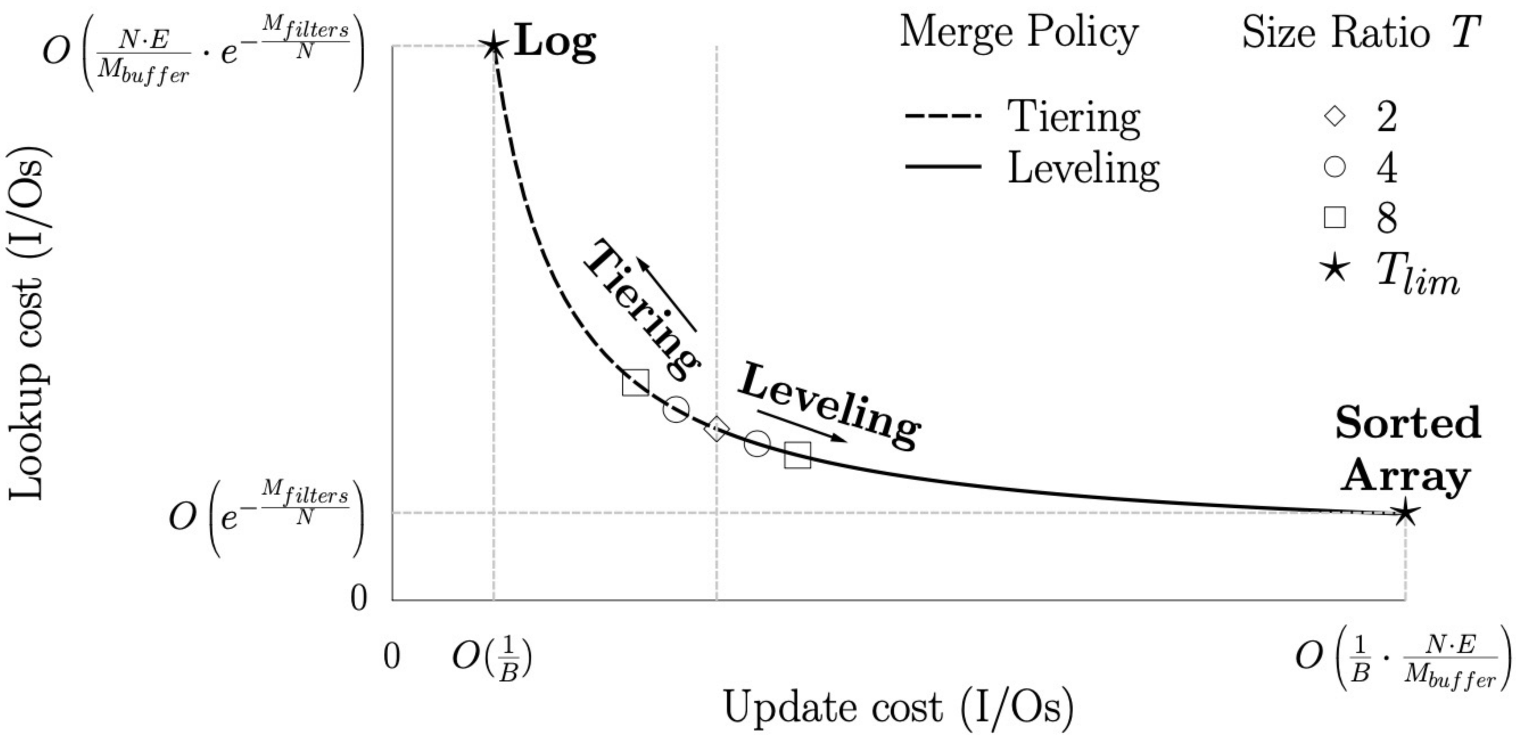
LSM Trees – Review

- Describe the two merge strategies



LSM Trees – Review

- What is the takeaway of the design space of LSM Trees?



Outline

- Brief discussion of the LSM paper
- Parallel Query Processing – Basics
- Parallel Query Processing – Systems
  - Next lecture (Wednesday)

Distributed/Parallel Query Processing

Parallel DBs since the 80s

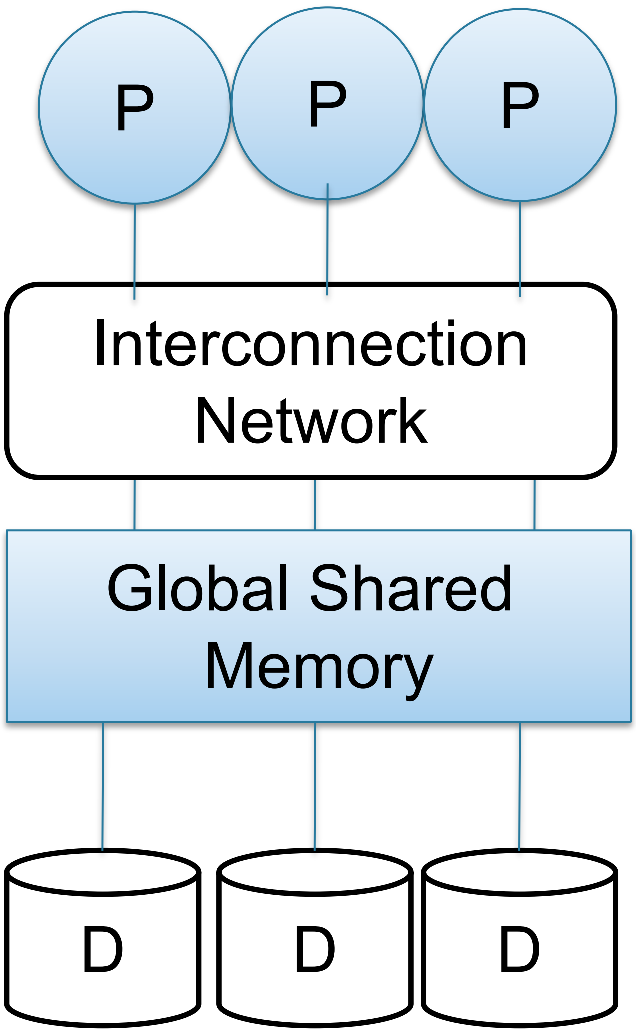
New, strong technology pulls:

- Multi-core
- Cloud computing

Architectures for Parallel Databases

- Shared memory
- Shared disk
- Shared nothing

Shared Memory

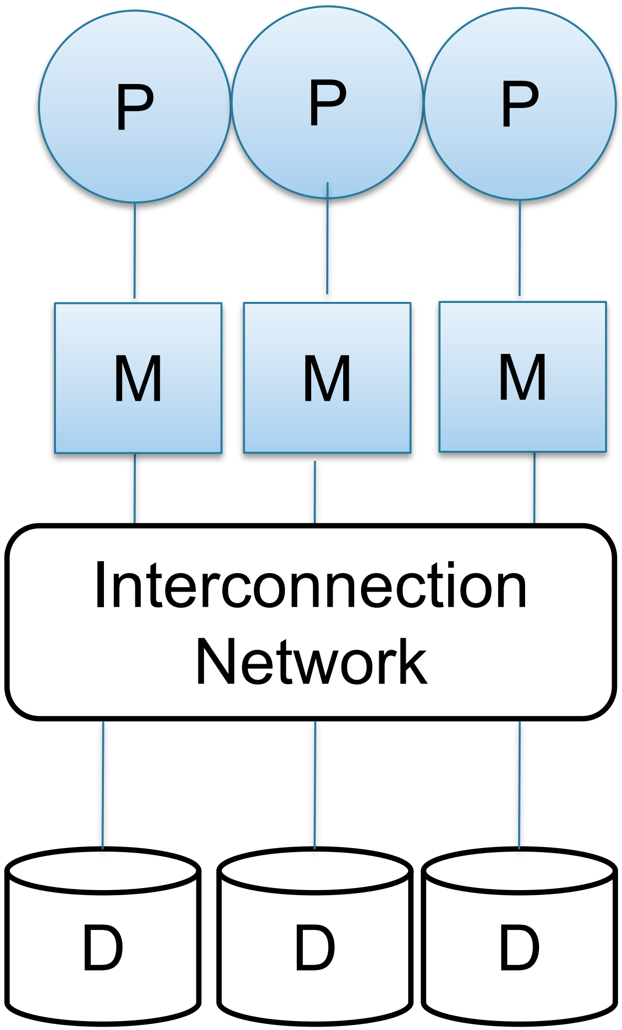


- SMP = symmetric multiprocessor
- Nodes share RAM and disk
- 10x … 100x processors

- Example: SQL Server runs on a single machine and can leverage many threads to speed up a query

- Easy to use and program
- Expensive to scale

Shared Disk



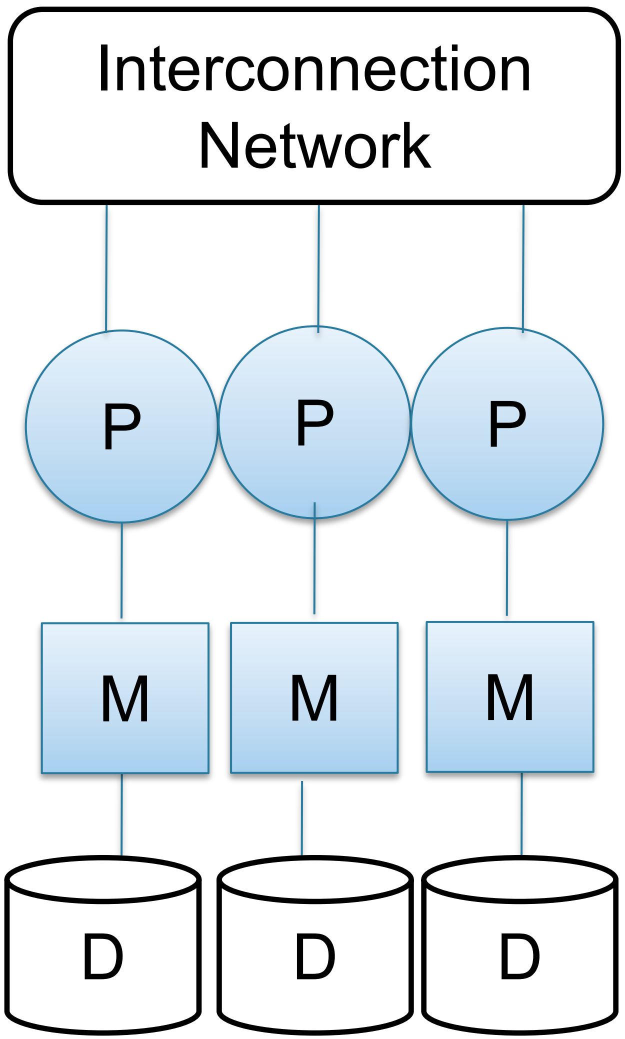
- All nodes access same disks
- 10x processors

- Example: Oracle

- No more memory contention

- Harder to program
- Still hard to scale

Shared Nothing



- Cluster of commodity machines
- Called "clusters" or "blade servers”
- Each machine: own memory&disk
- Up to x1000-x10000 nodes
- Example: redshift, spark, snowflake

Because all machines today have many cores and many disks, shared-nothing systems typically run many "nodes” on a single physical machine.

- Easy to maintain and scale
- Most difficult to administer and tune.

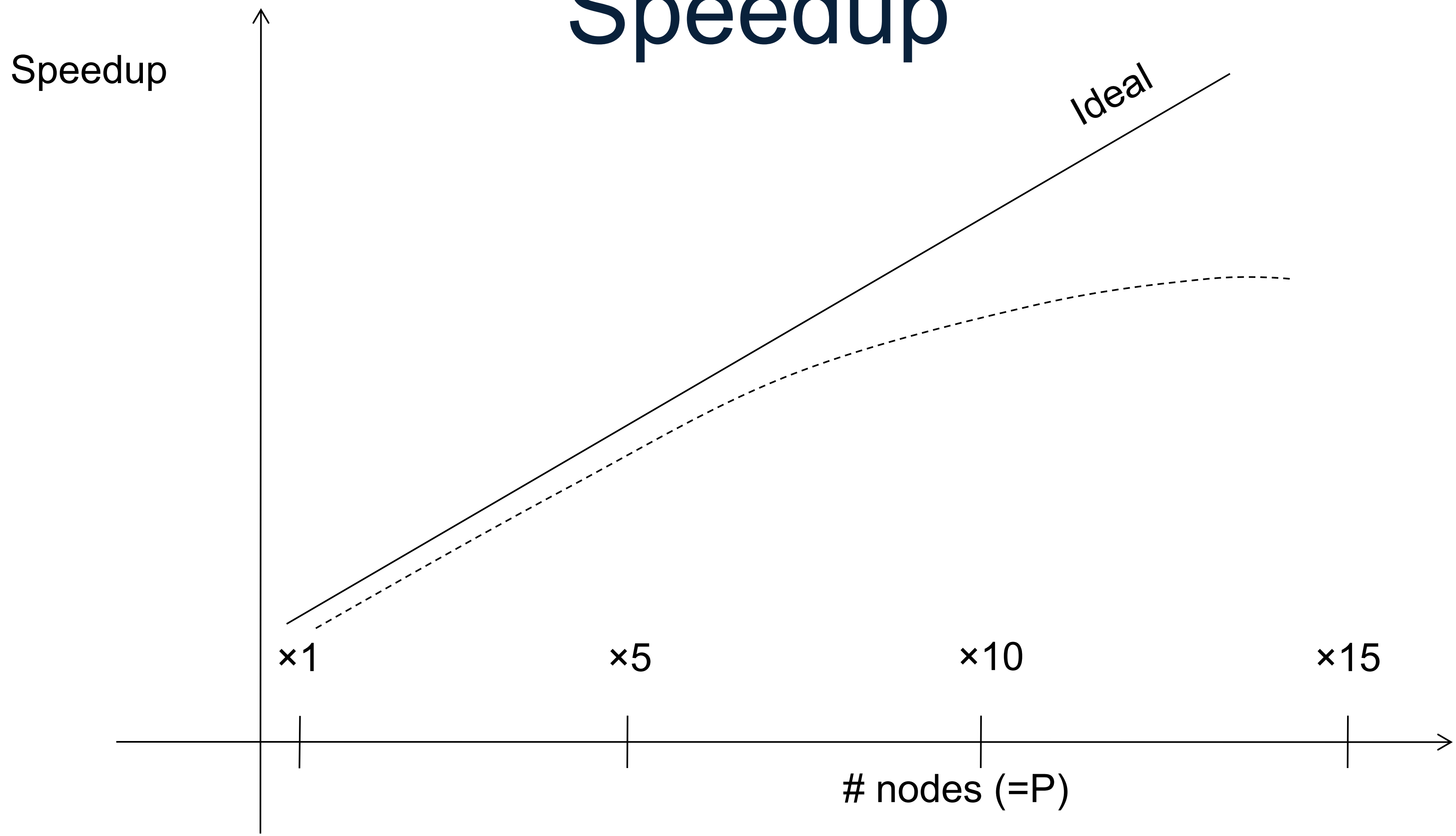
Performance Metrics

Nodes = processors = computers

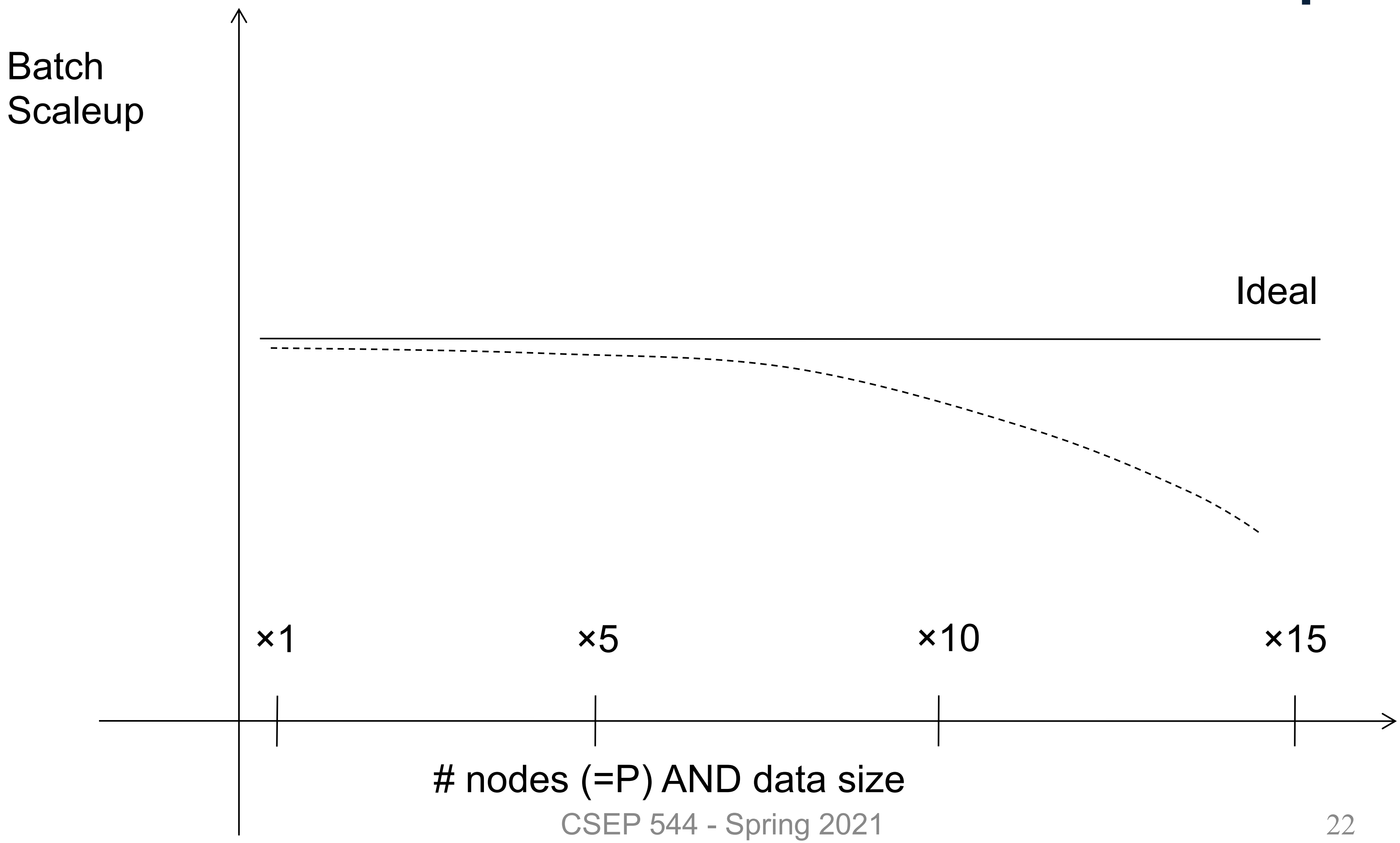
- Speedup:
  - More nodes, same data ➔ higher speed
- Scaleup:
  - More nodes, more data ➔ same speed

Warning: sometimes *Scaleup* is used to mean *Speedup*

Linear v.s. Non-linear Speedup



Linear v.s. Non-linear Scaleup



Why Sub-linear?

- Startup cost
  - Cost of starting an operation on many nodes
- Interference
  - Contention for resources between nodes
- Skew
  - Slowest node becomes the bottleneck

Distributed Query Processing Algorithms

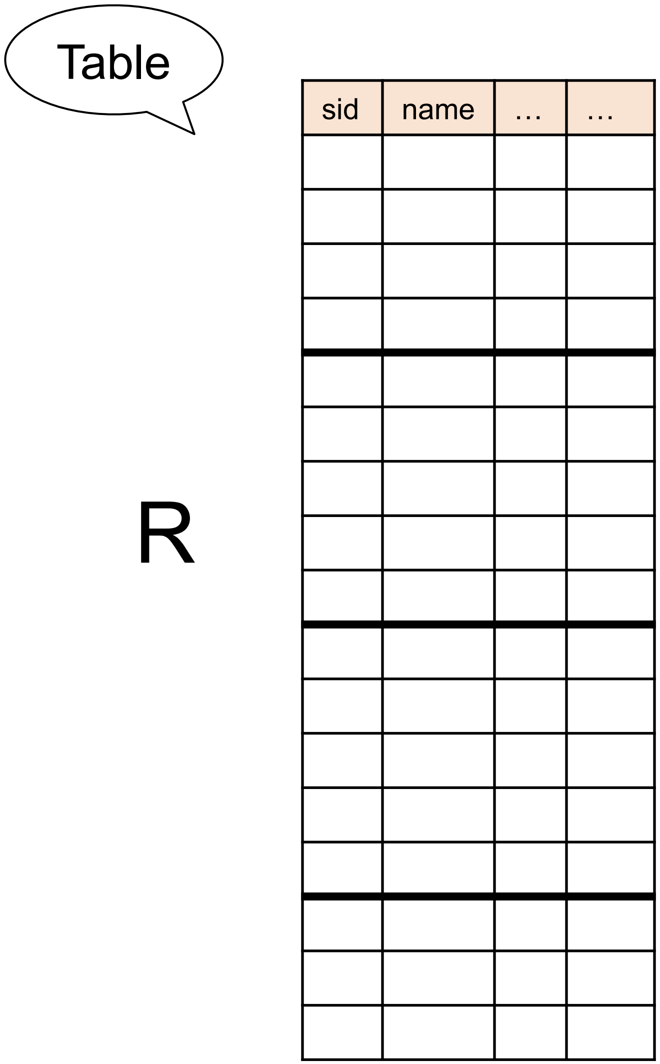
Horizontal Data Partitioning

Table

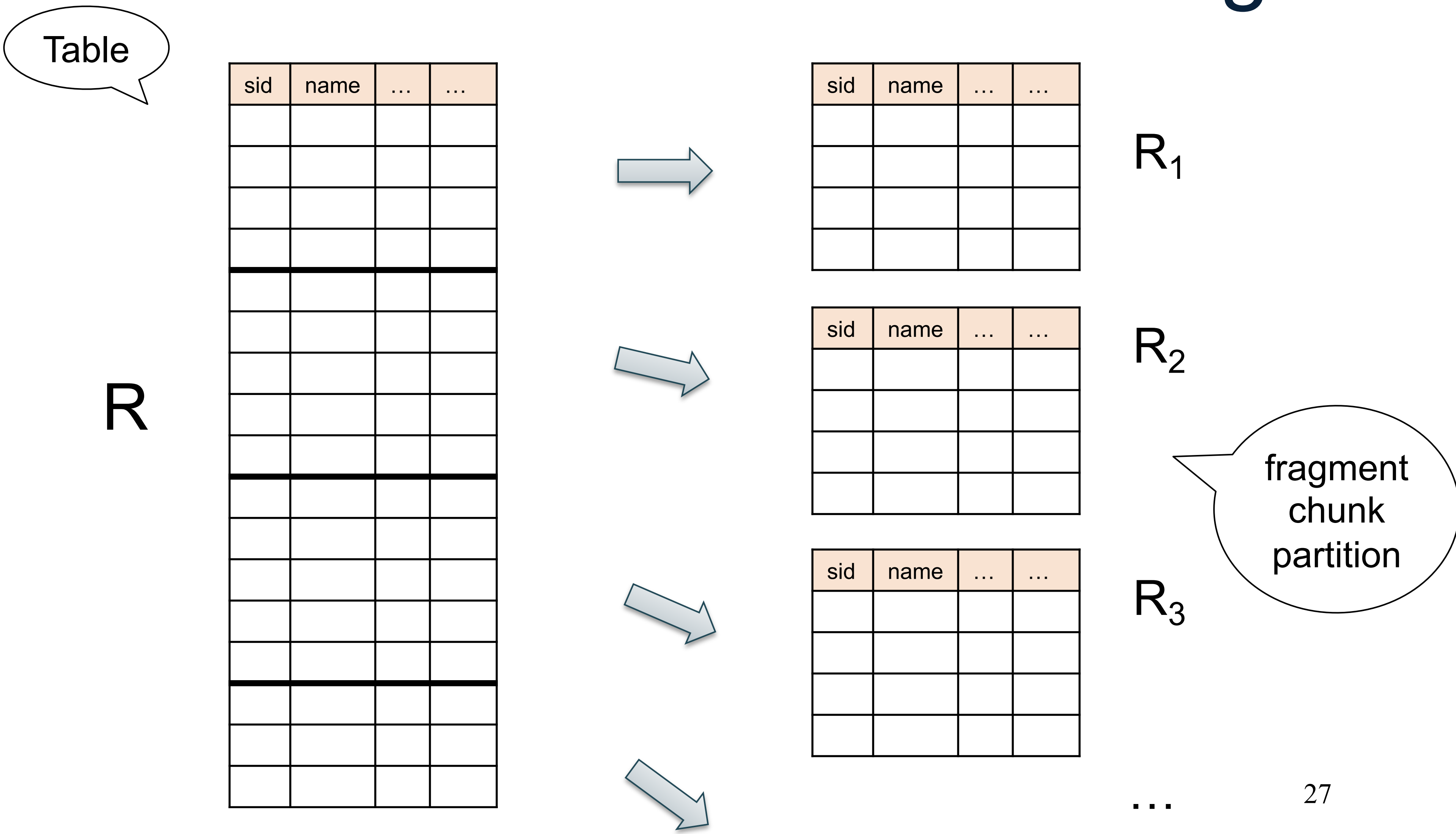
R

sid	name	…	…

Horizontal Data Partitioning



Horizontal Data Partitioning



Horizontal Data Partitioning

- Block Partition, a.k.a. Round Robin:
  - Partition tuples arbitrarily s.t. $\text{size}(R_1) \approx \ldots \approx \text{size}(R_P)$
- Hash partitioned on attribute A:
  - Tuple t goes to chunk i, where $i = h(t.A) \bmod P + 1$
- Range partitioned on attribute A:
  - Partition the range of A into $-\infty = v_0 < v_1 < \ldots < v_P = \infty$
  - Tuple t goes to chunk i, if $v_{i-1} < t.A < v_i$

Notations

p = number of servers (nodes) that hold the chunks

When a relation R is distributed to p servers,
we draw the picture like this:

Here R_1 is the fragment of R stored on server 1, etc

$$R = R_1 \cup R_2 \cup \cdots \cup R_P$$

Uniform Load and Skew

- $|R| = N$ tuples, then $|R_1| + |R_2| + \ldots + |R_p| = N$
- We say the load is uniform when:

  $$|R_1| \approx |R_2| \approx \ldots \approx |R_p| \approx N/p$$

- Skew means that some load is much larger:

  $$\max_i |R_i| >> N/p$$

We design algorithms for uniform load, discuss skew later

Parallel Algorithm

- Selection σ
- Join ⋈
- Group by γ

Parallel Selection

Data: R(K, A, B, C)

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

- Block partitioned:
- Hash partitioned:
- Range partitioned:

Parallel Selection

Data: R(K, A, B, C)

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

- Block partitioned:
  - All servers need to scan
- Hash partitioned:

- Range partitioned:

Parallel Selection

Data: R(<u>K</u>, A, B, C)

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

- Block partitioned:
  - All servers need to scan
- Hash partitioned:
  - Point query: only one server needs to scan
  - Range query: all servers need to scan
- Range partitioned:

Parallel Selection

Data: R(<u>K</u>, A, B, C)

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

- Block partitioned:
  - All servers need to scan
- Hash partitioned:
  - Point query: only one server needs to scan
  - Range query: all servers need to scan
- Range partitioned:
  - Only some servers need to scan

Parallel GroupBy

Data: R(<u>K</u>, A, B, C)

Query: $\gamma_{A,sum(C)}(R)$

Discuss in class how to compute in each case:

- R is hash-partitioned on A
- R is block-partitioned or hash-partitioned on K

Parallel GroupBy

Data: R(K, A, B, C)

Query: $\gamma_{A,sum(C)}(R)$

Discuss in class how to compute in each case:

- R is hash-partitioned on A
  - Each server i computes locally $\gamma_{A,sum(C)}(R_i)$
- R is block-partitioned or hash-partitioned on K

Parallel GroupBy

Data: R(<u>K</u>, A, B, C)

Query: $\gamma_{A,sum(C)}(R)$

Discuss in class how to compute in each case:

- R is hash-partitioned on A
  - Each server i computes locally $\gamma_{A,sum(C)}(R_i)$
- R is block-partitioned or hash-partitioned on K
  - Need to reshuffle data on A first (next slide)
  - Then compute locally $\gamma_{A,sum(C)}(R_i)$

Basic Parallel GroupBy

Data: R(K, A, B, C)

Query: $\gamma_{A,sum(C)}(R)$

- R is block-partitioned or hash-partitioned on K

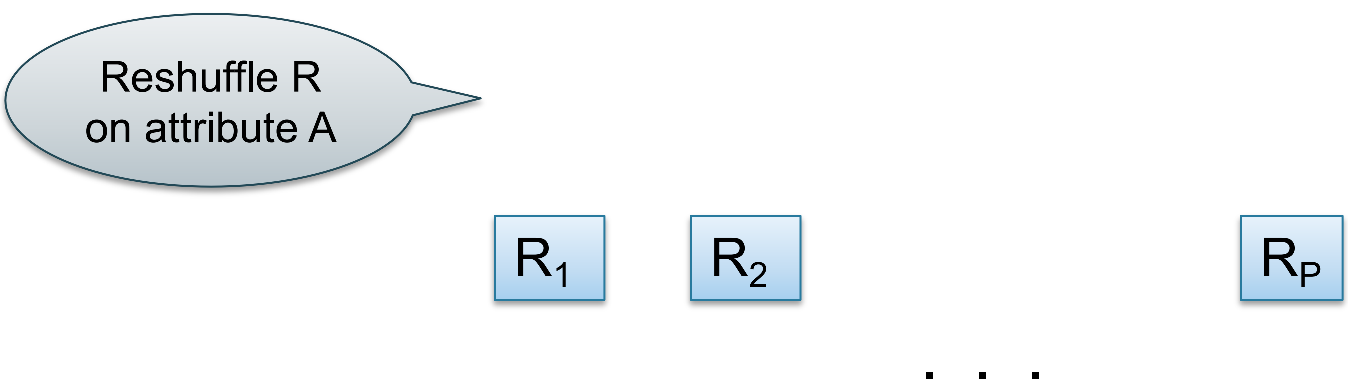
R_1 R_2 . . . R_P

Basic Parallel GroupBy

Data: R(K, A, B, C)

Query: $\gamma_{A,sum(C)}(R)$

- R is block-partitioned or hash-partitioned on K

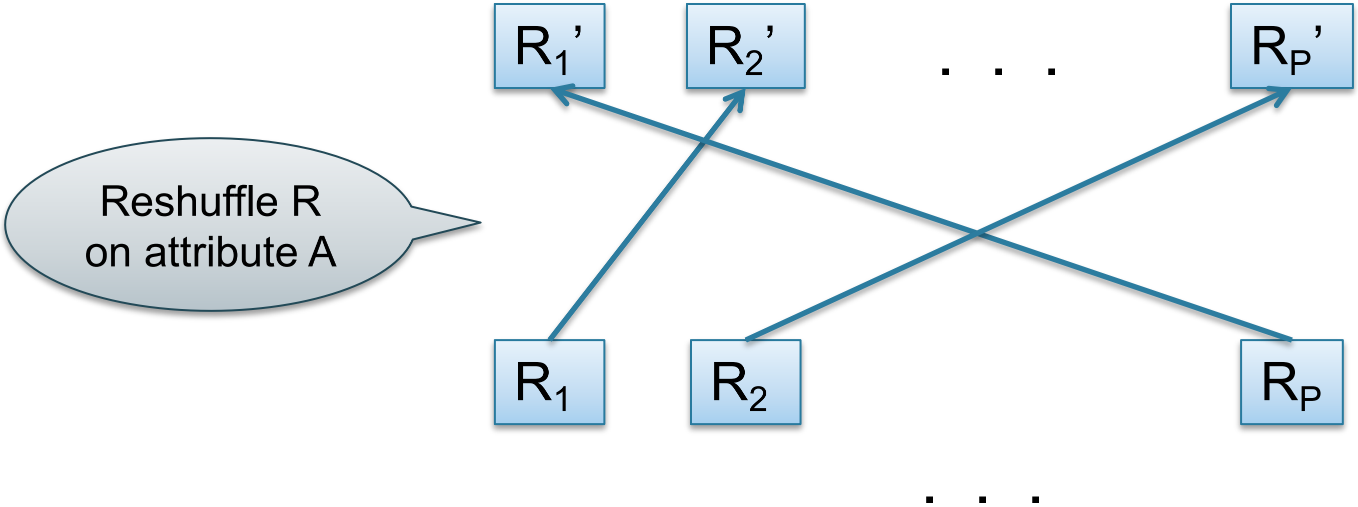


Basic Parallel GroupBy

Data: R(K, A, B, C)

Query: $\gamma_{A,sum(C)}(R)$

- R is block-partitioned or hash-partitioned on K

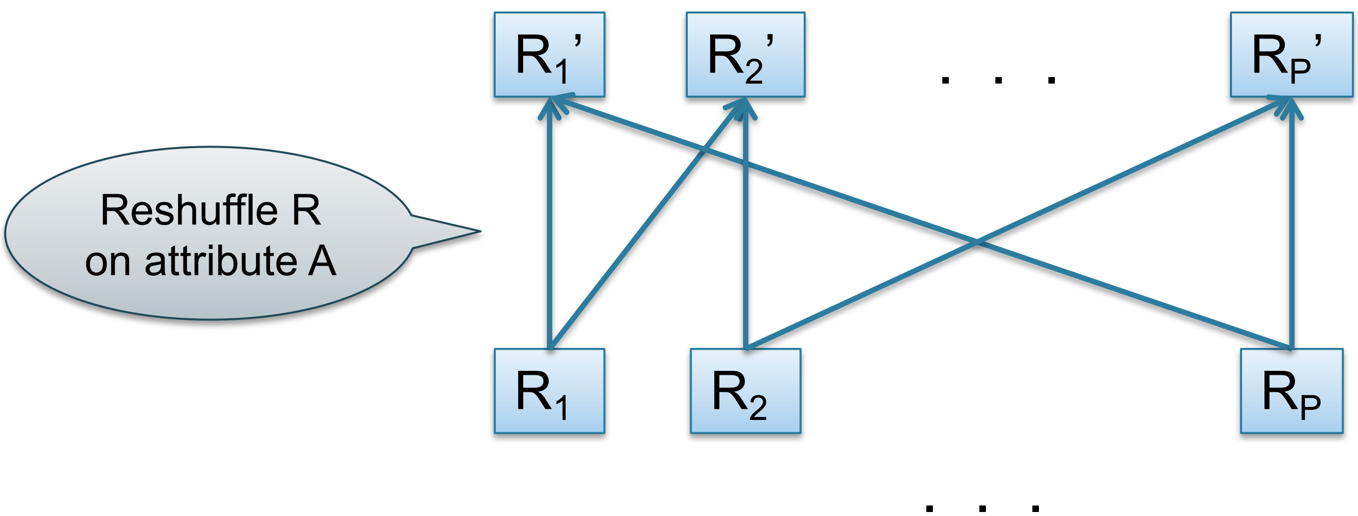


Basic Parallel GroupBy

Data: R(K, A, B, C)

Query: $\gamma_{A,sum(C)}(R)$

- R is block-partitioned or hash-partitioned on K

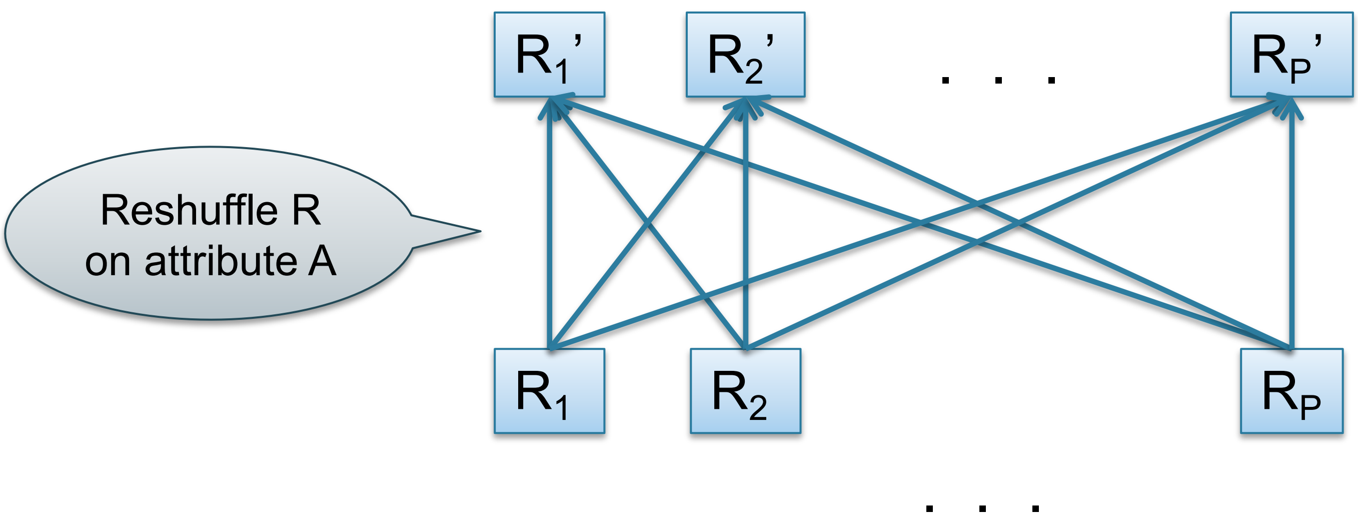


Basic Parallel GroupBy

Data: R(K, A, B, C)

Query: $\gamma_{A,sum(C)}(R)$

- R is block-partitioned or hash-partitioned on K

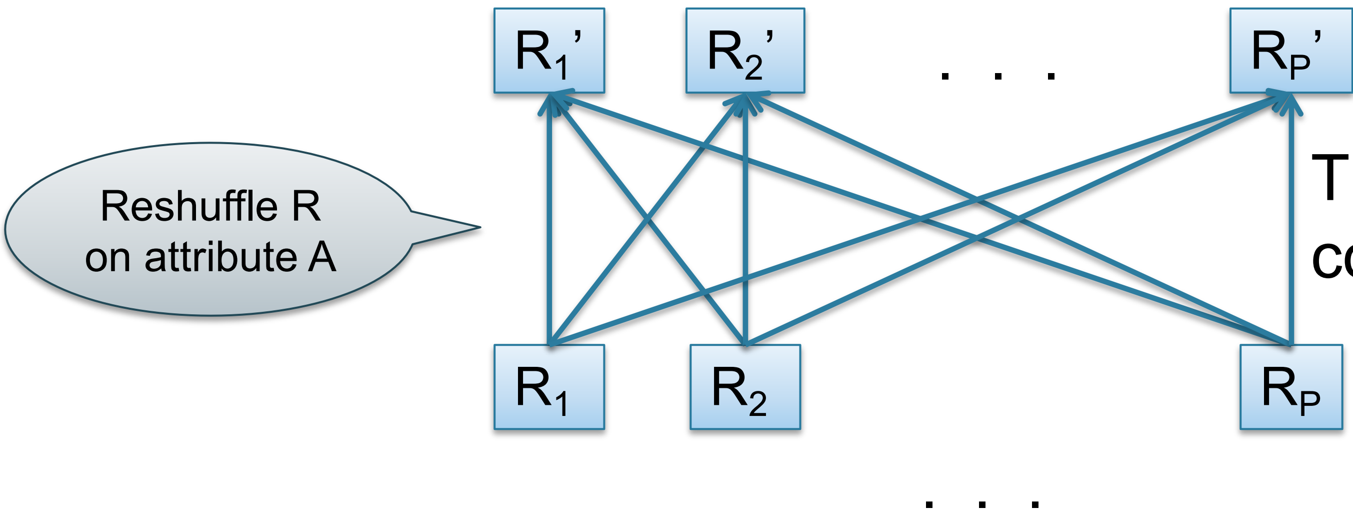


Basic Parallel GroupBy

Data: R(<u>K</u>, A, B, C)

Query: $\gamma_{A,sum(C)}(R)$

- R is block-partitioned or hash-partitioned on K



This is done in *one* communication step

Reshuffling

- Nodes send data over the network
- Many-many communications possible
- Throughput:
  - Better than disk
  - Worse than main memory

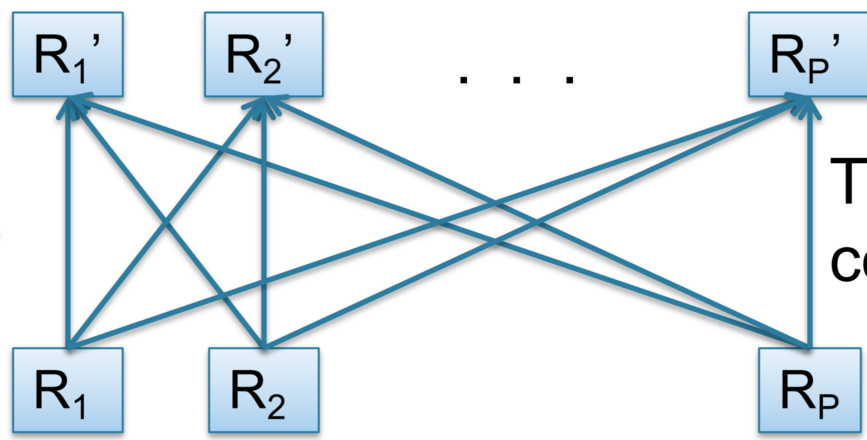
Basic Parallel GroupBy

Data: R(<u>K</u>, A, B, C)

Query: $\gamma_{A,sum(C)}(R)$

- R is block-partitioned or hash-partitioned on K

Reshuffle R on attribute A



This is done in *one* communication step

. . .

Can you think of an optimization?

GroupBy/Union Commutativity

	city	…	qant
	Seattle		10
	LA		20
	Seattle		30
	NY		40

	city	…	qant
	LA		22
	NY		33
	LA		44
	Austin		55

	city	…	qant
	Seattle		66
	LA		77
	NY		88
	LA		99

```
SELECT city, sum(quant)
FROM R
GROUP BY city
```

GroupBy/Union Commutativity

	city	…	qant
	Seattle		10
	LA		20
	Seattle		30
	NY		40

	city	…	qant
	LA		22
	NY		33
	LA		44
	Austin		55

	city	…	qant
	Seattle		66
	LA		77
	NY		88
	LA		99

```
SELECT city, sum(quant)
FROM R
GROUP BY city
```

$$\gamma_{city,sum(q)}(R_1 \cup R_2 \cup R_3) =$$

GroupBy/Union Commutativity

	city	…	qant
	Seattle		10
	LA		20
	Seattle		30
	NY		40

	city	…	qant
	LA		22
	NY		33
	LA		44
	Austin		55

	city	…	qant
	Seattle		66
	LA		77
	NY		88
	LA		99

```
SELECT city, sum(quant)
FROM R
GROUP BY city
```

$$\gamma_{city,sum(q)}(R_1 \cup R_2 \cup R_3) =$$

$$= \gamma_{city,sum(q)}\left(\gamma_{city,sum(q)}(R_1) \cup \gamma_{city,sum(q)}(R_2) \cup \gamma_{city,sum(q)}(R_3)\right)$$

Basic Parallel GroupBy

Data: R(K, A, B, C)
Query: $\gamma_{A,sum(C)}(R)$

Basic Parallel GroupBy

Data: R(<u>K</u>, A, B, C)
Query: $\gamma_{A,sum(C)}(R)$

Step 0: [Optimization] each server i computes local group-by:

$$T_i = \gamma_{A,sum(C)}(R_i)$$

Basic Parallel GroupBy

Data: R(K, A, B, C)
Query: $\gamma_{A,sum(C)}(R)$

Step 0: [Optimization] each server i computes local group-by:

$$T_i = \gamma_{A,sum(C)}(R_i)$$

Step 1: partitions tuples in T_i using hash function h(A):

$$T_{i,1}, T_{i,2}, \ldots, T_{i,p}$$

then send fragment $T_{i,j}$ to server j

Basic Parallel GroupBy

Data: R(<u>K</u>, A, B, C)
Query: $\gamma_{A,sum(C)}(R)$

Step 0: [Optimization] each server i computes local group-by:
$T_i = \gamma_{A,sum(C)}(R_i)$

Step 1: partitions tuples in T_i using hash function h(A):
$T_{i,1}, T_{i,2}, \ldots, T_{i,p}$
then send fragment $T_{i,j}$ to server j

Step 2: receive fragments, union them, then group-by
$R_j' = T_{1,j} \cup \ldots \cup T_{p,j}$
$Answer_j = \gamma_{A, sum(C)} (R_j')$

Pushing Aggregates Past Union

Which other rules can we push past union?

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

Pushing Aggregates Past Union

Which other rules can we push past union?

- 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)

Example Query with Group By

```
SELECT a, sum(b) as sb
FROM R WHERE c > 0
GROUP BY a
```

Example Query with Group By

```
SELECT a, sum(b) as sb
FROM R WHERE c > 0
GROUP BY a
```

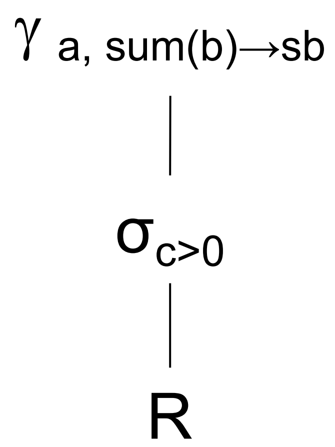
$$\gamma_{a,\ sum(b) \rightarrow sb}$$

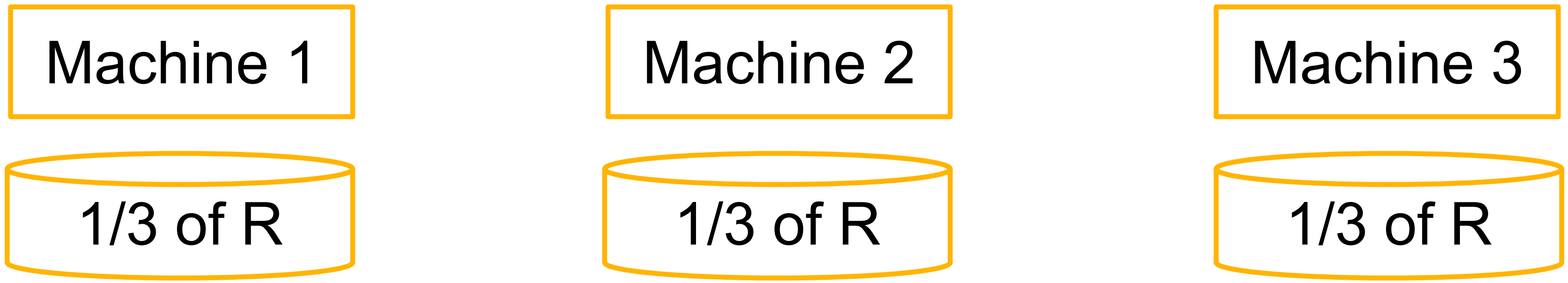
$$\sigma_{c>0}$$

$$R$$

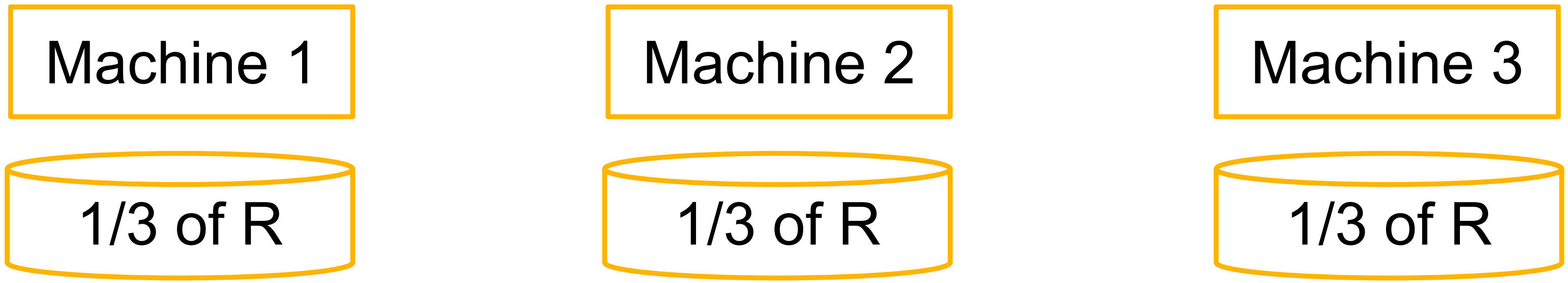
Example Query with Group By

```
SELECT a, sum(b) as sb
FROM R WHERE c > 0
GROUP BY a
```

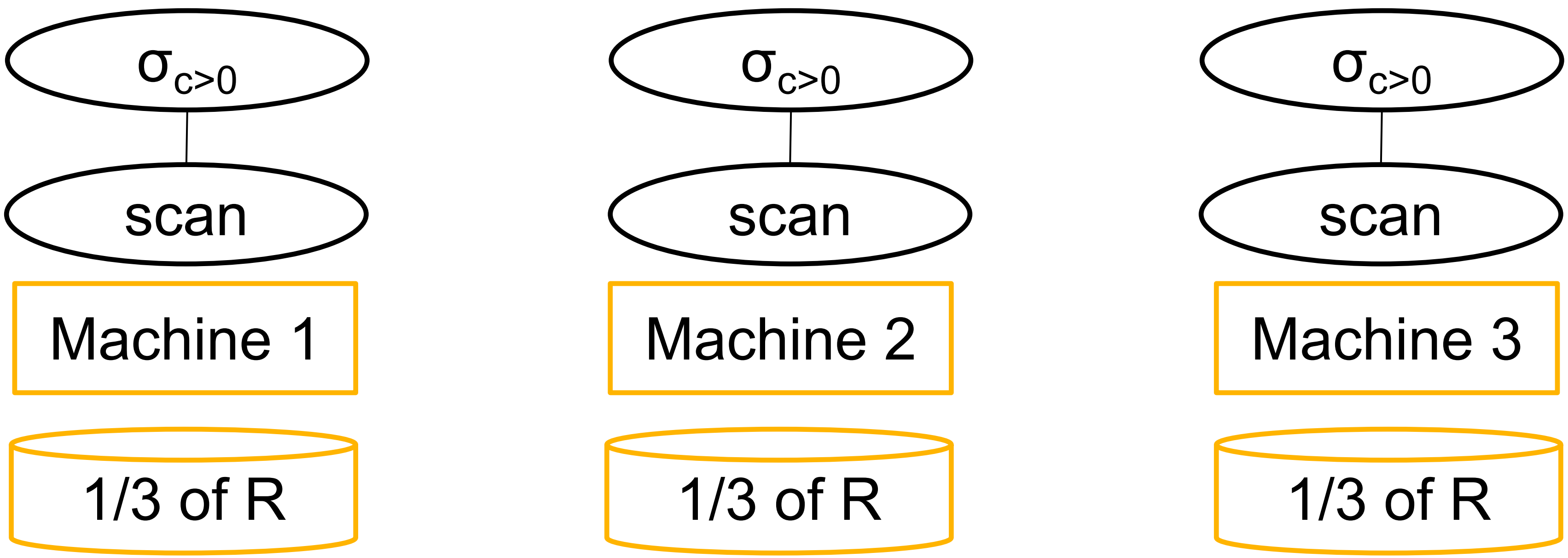




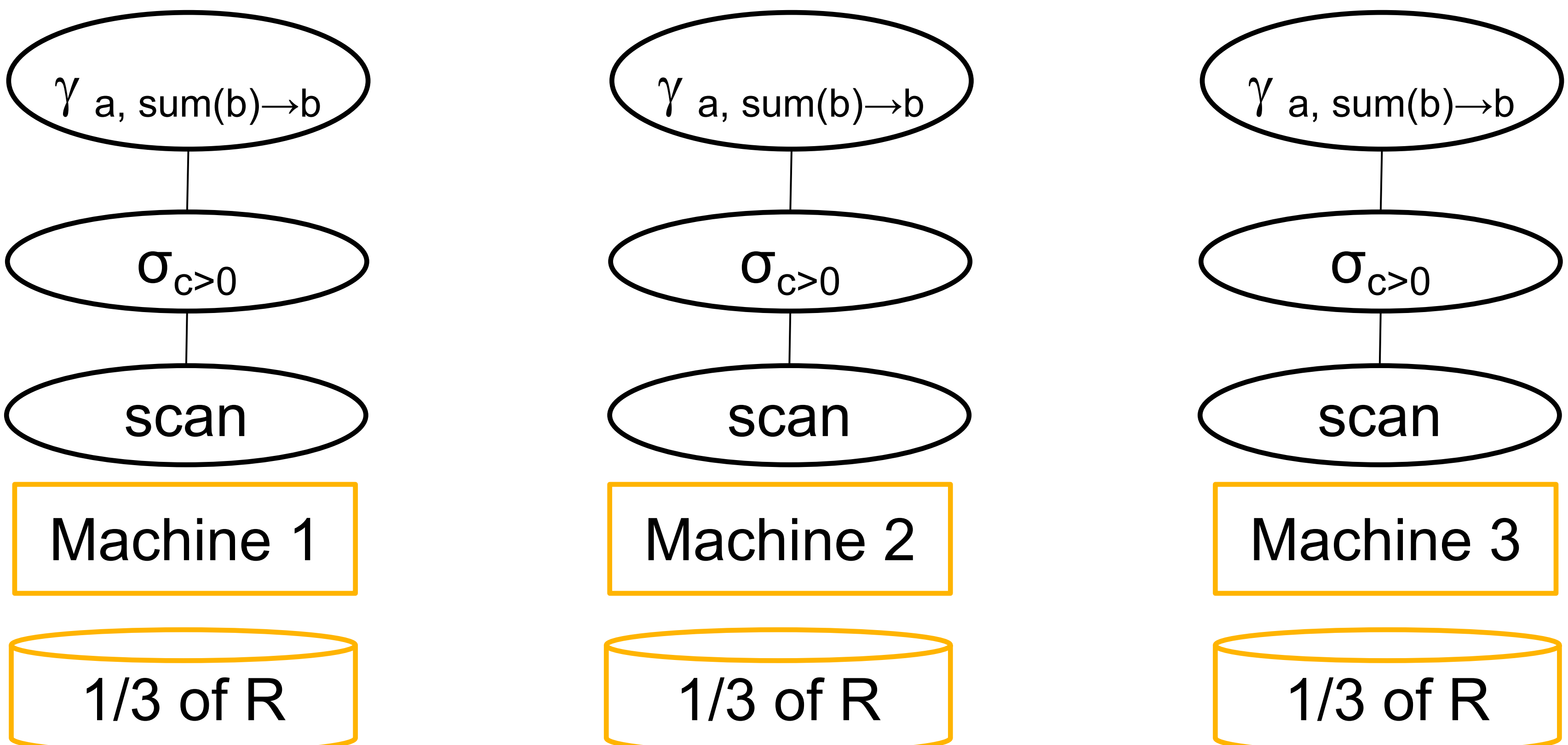
SELECT a, sum(b) as sb FROM R WHERE c > 0 GROUP BY a



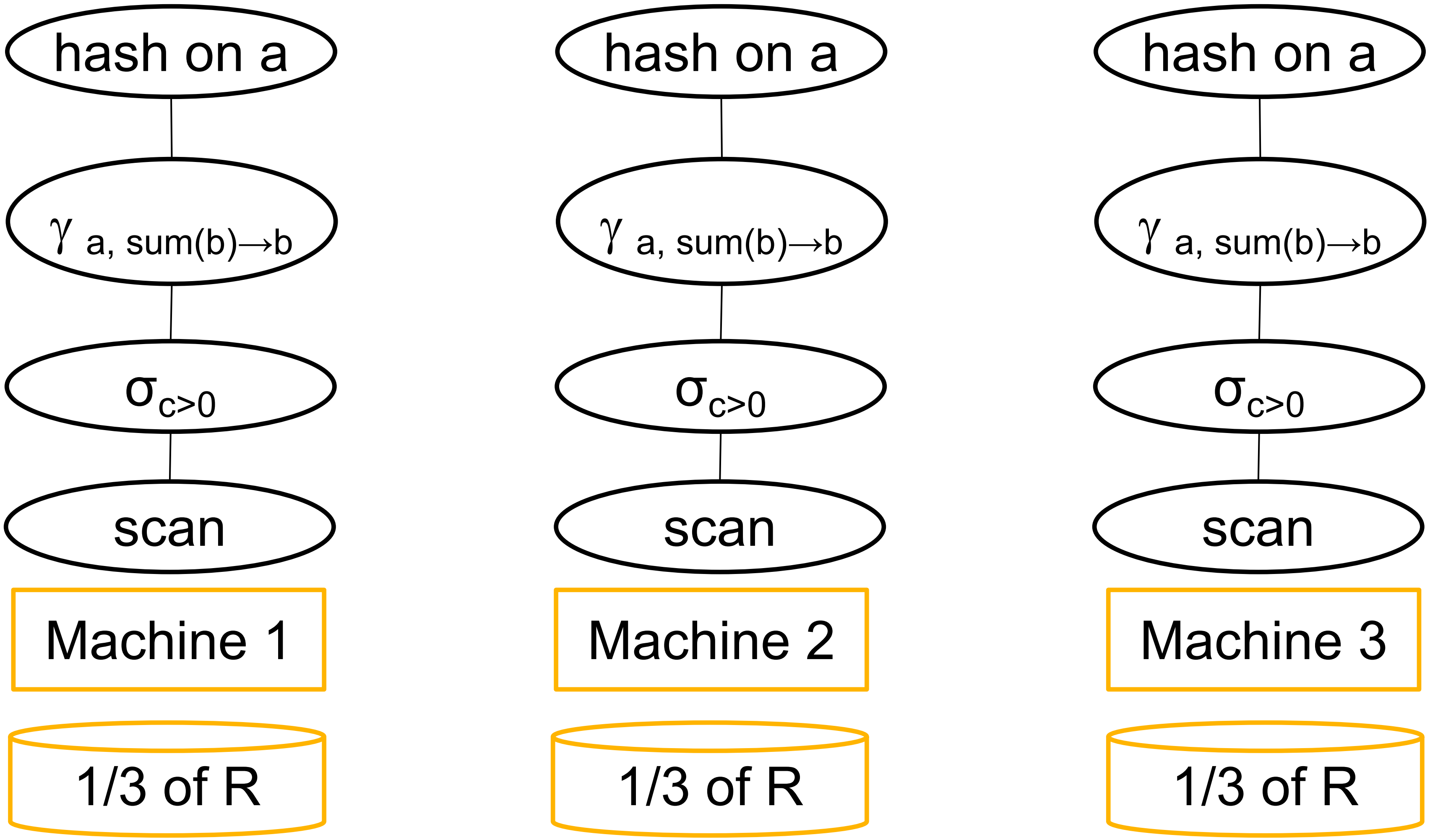
SELECT a, sum(b) as sb FROM R WHERE c > 0 GROUP BY a



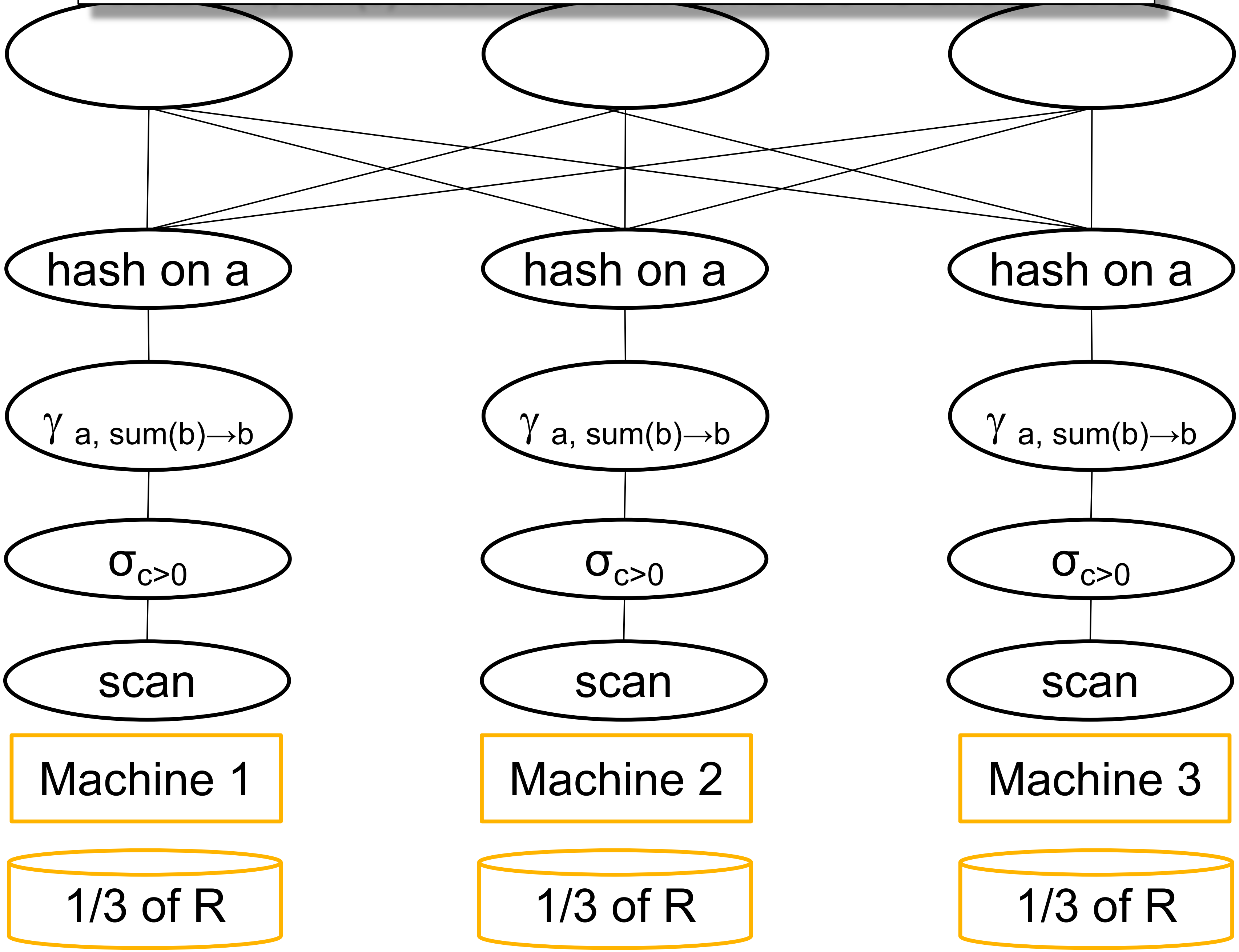
SELECT a, sum(b) as sb FROM R WHERE c > 0 GROUP BY a

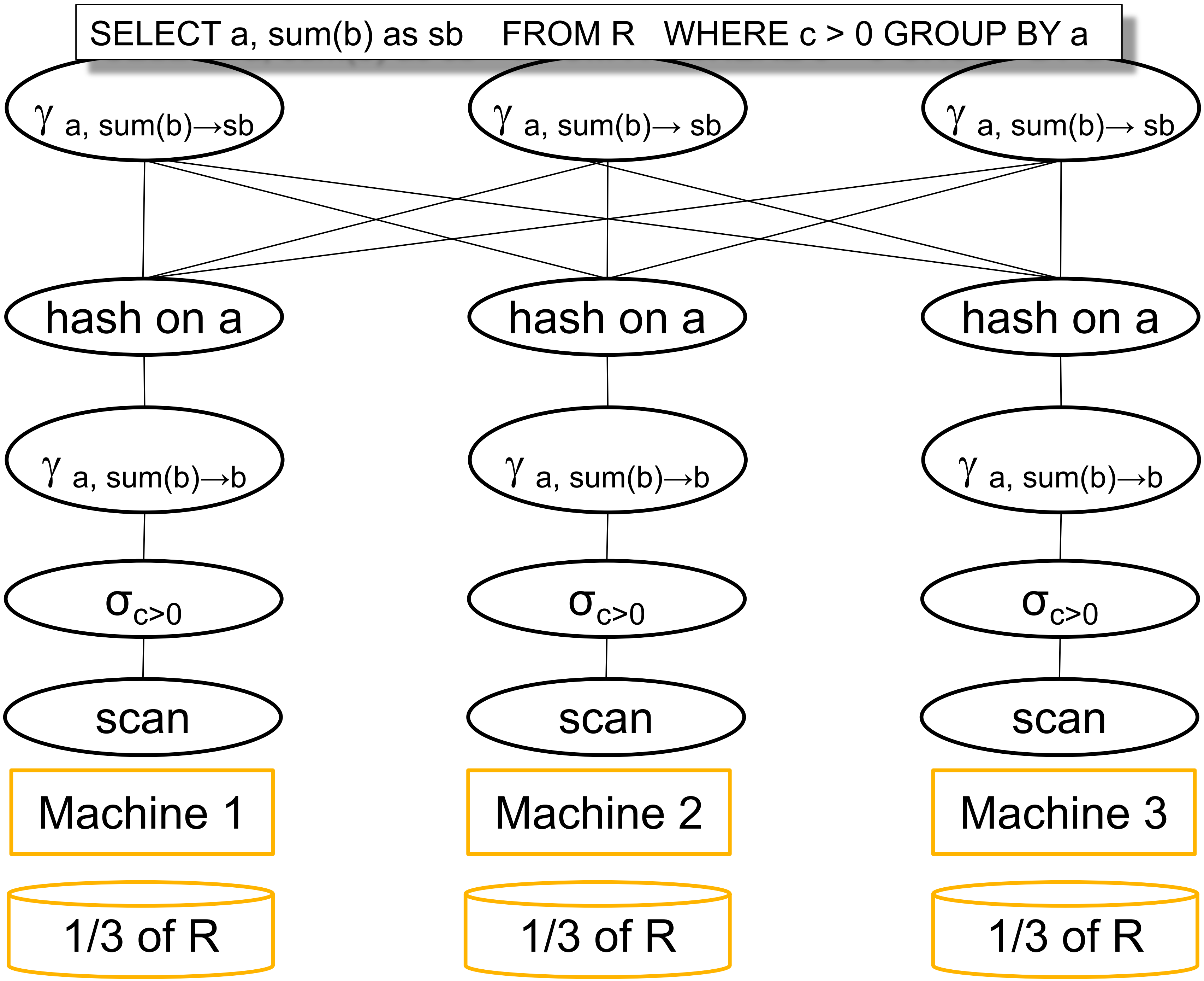


SELECT a, sum(b) as sb FROM R WHERE c > 0 GROUP BY a



SELECT a, sum(b) as sb FROM R WHERE c > 0 GROUP BY a





SELECT a, sum(b) as sb FROM R WHERE c > 0 GROUP BY a
γ a, sum(b)→sb
γ a, sum(b)→ sb
γ a, sum(b)→ sb
hash on a
hash on a
hash on a
γ a, sum(b)→b
γ a, sum(b)→b
γ a, sum(b)→b
σc>0
σc>0
σc>0
scan
scan
scan
Machine 1
Machine 2
Machine 3
1/3 of R
1/3 of R
1/3 of R

Speedup and Scaleup

Consider the query $\gamma_{A,sum(C)}(R)$
Assume the local runtime for group-by is linear O(|R|)

If we double number of nodes P, what is the runtime?

If we double both P and size of R, what is the runtime?

Speedup and Scaleup

Consider the query $\gamma_{A,sum(C)}(R)$
Assume the local runtime for group-by is linear O(|R|)

If we double number of nodes P, what is the runtime?

- Half (chunk sizes become ½)

If we double both P and size of R, what is the runtime?

- Same (chunk sizes remain the same)

Speedup and Scaleup

Consider the query $\gamma_{A,sum(C)}(R)$
Assume the local runtime for group-by is linear O(|R|)

If we double number of nodes P, what is the runtime?

- Half (chunk sizes become ½)

If we double both P and size of R, what is the runtime?

- Same (chunk sizes remain the same)

But only if the data is without skew!

Parallel/Distributed Join

Three “algorithms”:

- Hash-partitioned
- Broadcast
- Combined: “skew-join” or other names

Hash Join: $R \bowtie_{A=B} S$

Data: $R(A, C)$, $S(B, D)$

Query: $R \bowtie_{A=B} S$

R_1, S_1 R_2, S_2 . . . R_P, S_P

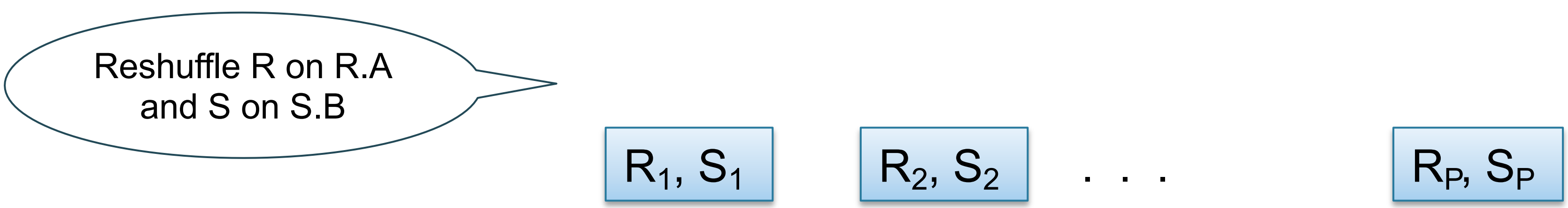
Initially, R and S are block partitioned.
Notice: they may be stored in DFS (recall MapReduce)

Some servers hold R-chunks, some hold S-chunks, some hold both

Hash Join: $R \bowtie_{A=B} S$

Data: R(A, C), S(B, D)

Query: $R \bowtie_{A=B} S$



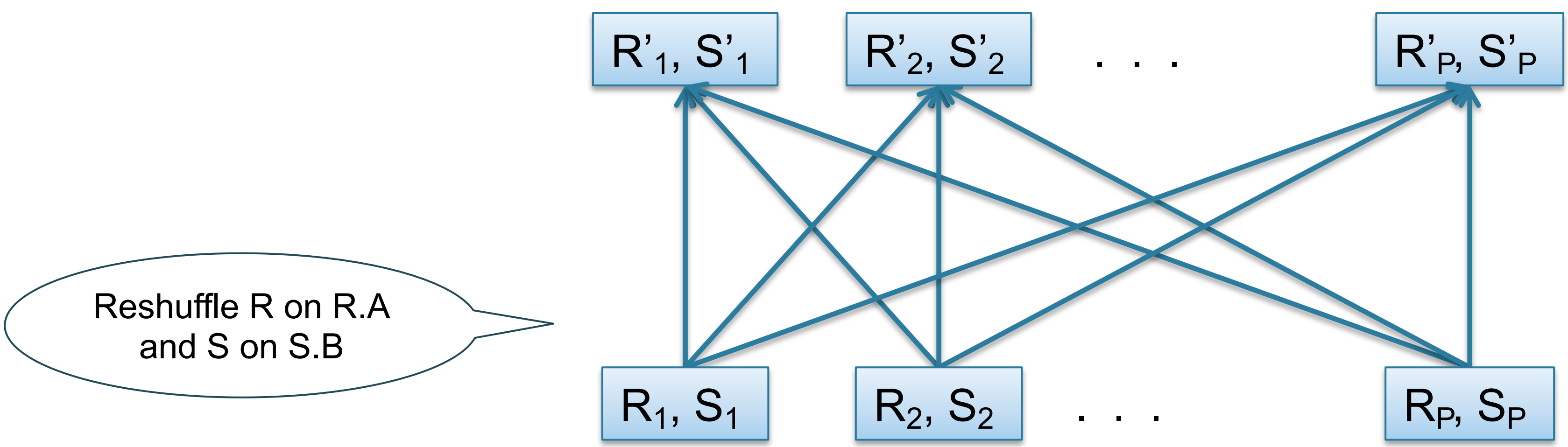
Initially, R and S are block partitioned.
Notice: they may be stored in DFS (recall MapReduce)

Some servers hold R-chunks, some hold S-chunks, some hold both

Hash Join: R ⋈$_{A=B}$ S

Data: R(A, C), S(B, D)

Query: R ⋈$_{A=B}$ S



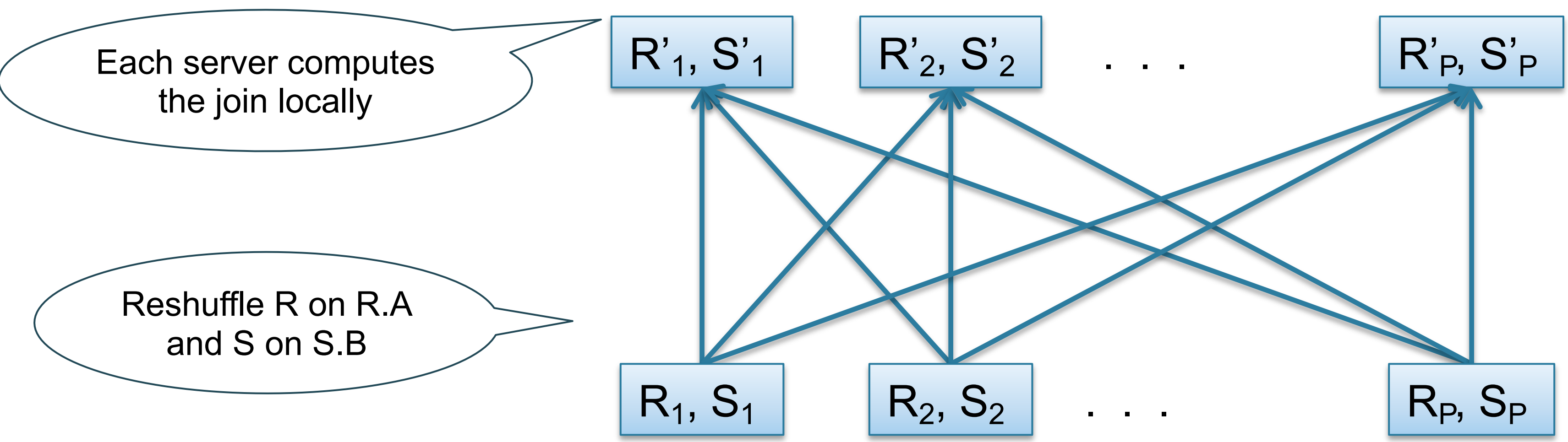
Initially, R and S are block partitioned.
Notice: they may be stored in DFS (recall MapReduce)

Some servers hold R-chunks, some hold S-chunks, some hold both

Hash Join: $R \bowtie_{A=B} S$

Data: R(A, C), S(B, D)

Query: $R \bowtie_{A=B} S$



Initially, R and S are block partitioned.
Notice: they may be stored in DFS (recall MapReduce)

Some servers hold R-chunks, some hold S-chunks, some hold both

Hash Join: $R \bowtie_{A=B} S$

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

Broadcast Join

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

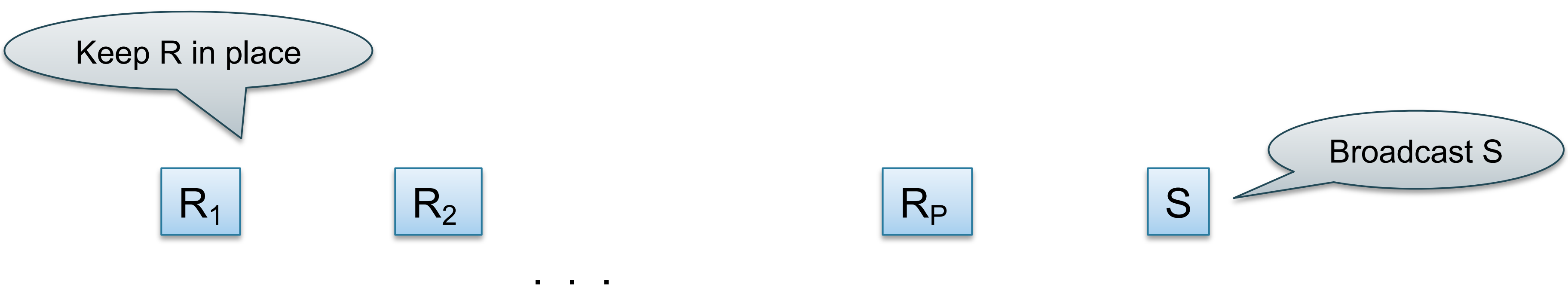
- Also called a small join or a broadcast join

Query: $R \bowtie S$

Broadcast Join

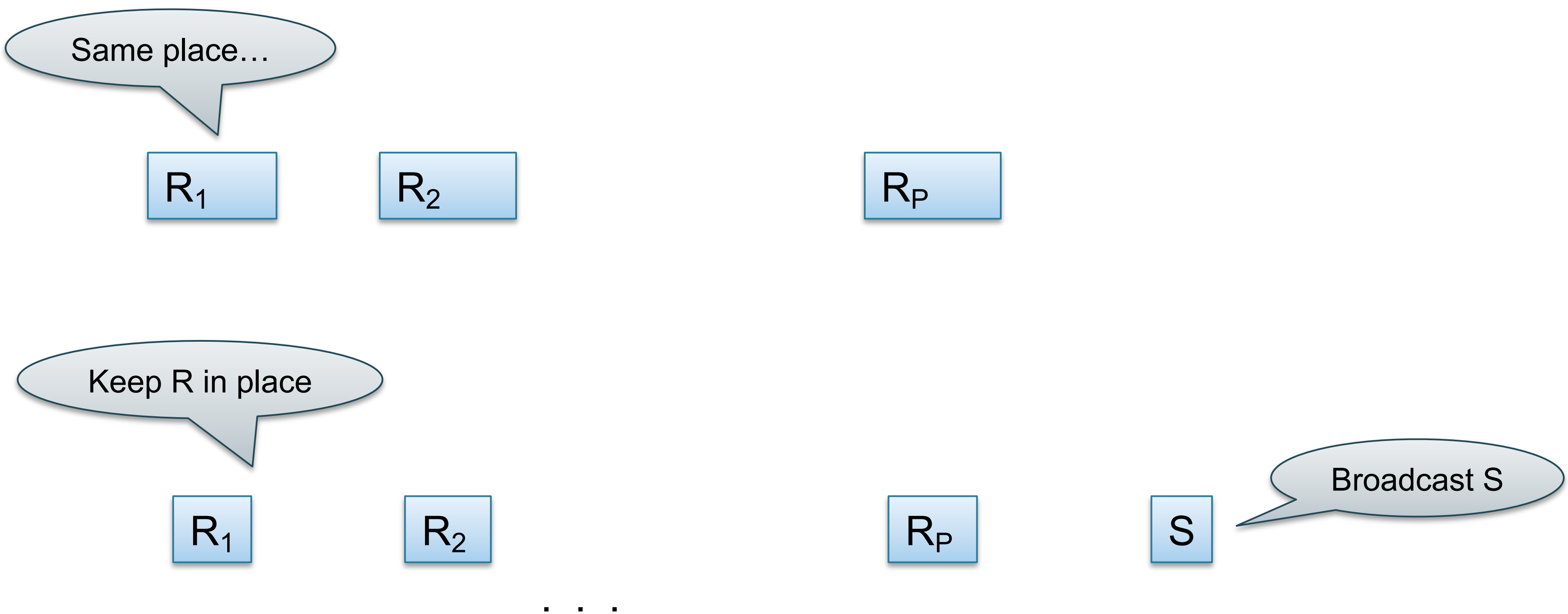
Query: $R \bowtie S$

Broadcast Join



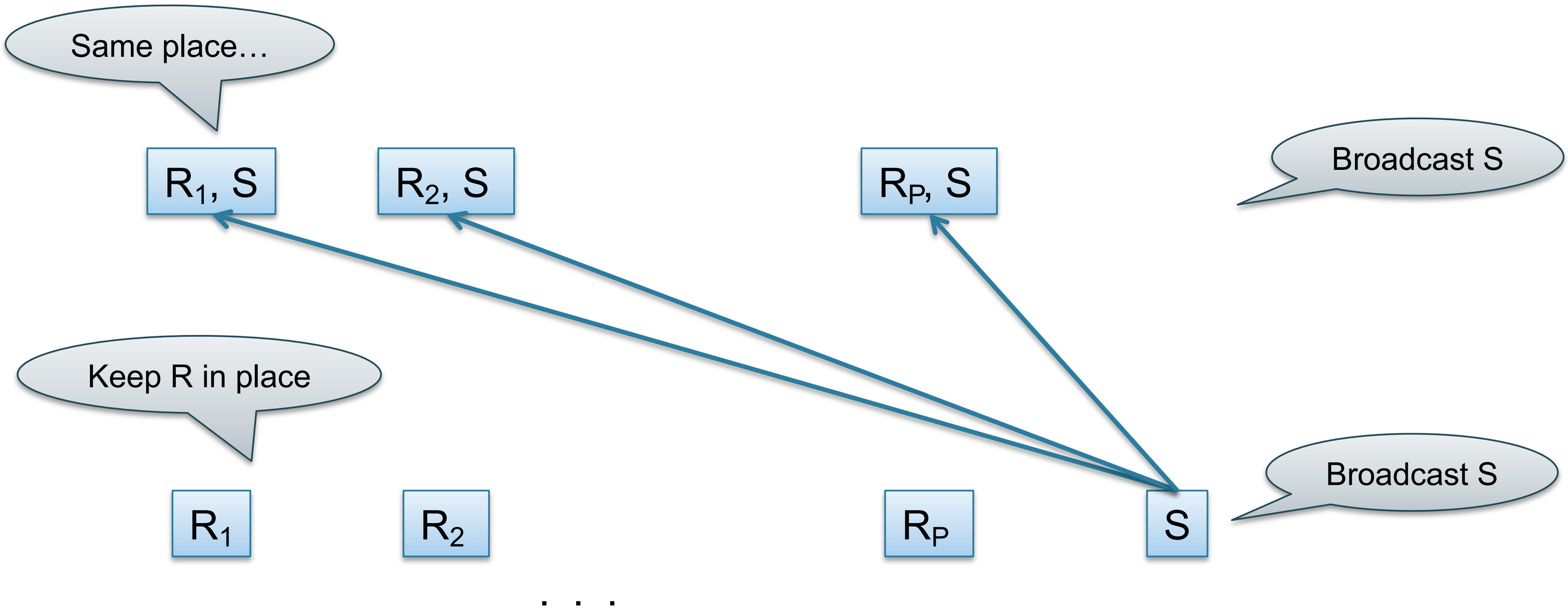
Query: $R \bowtie S$

Broadcast Join



Query: $R \bowtie S$

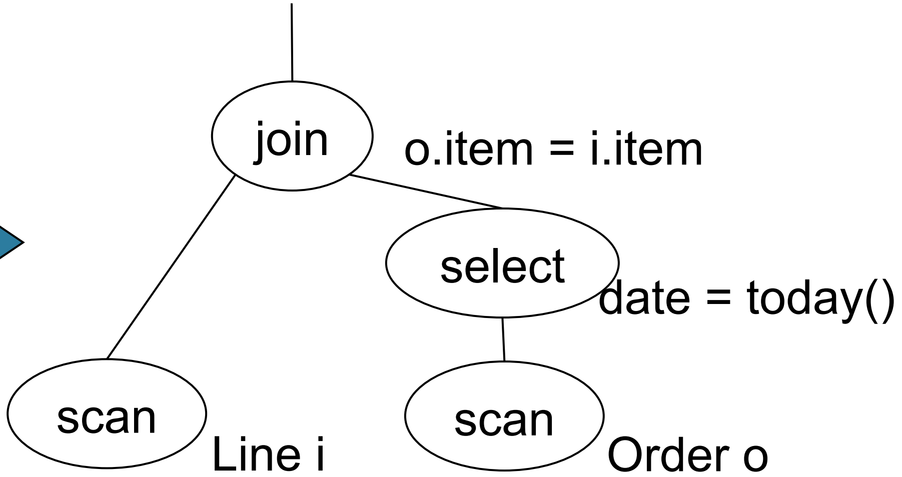
Broadcast Join



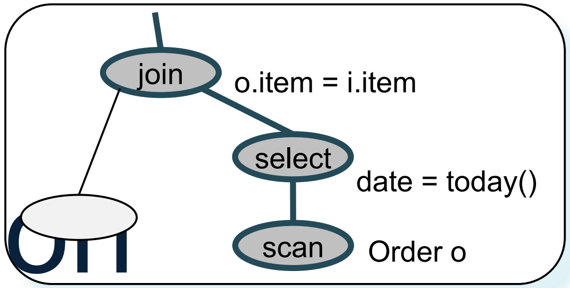
Example Query Execution

Find all orders from today, along with the items ordered

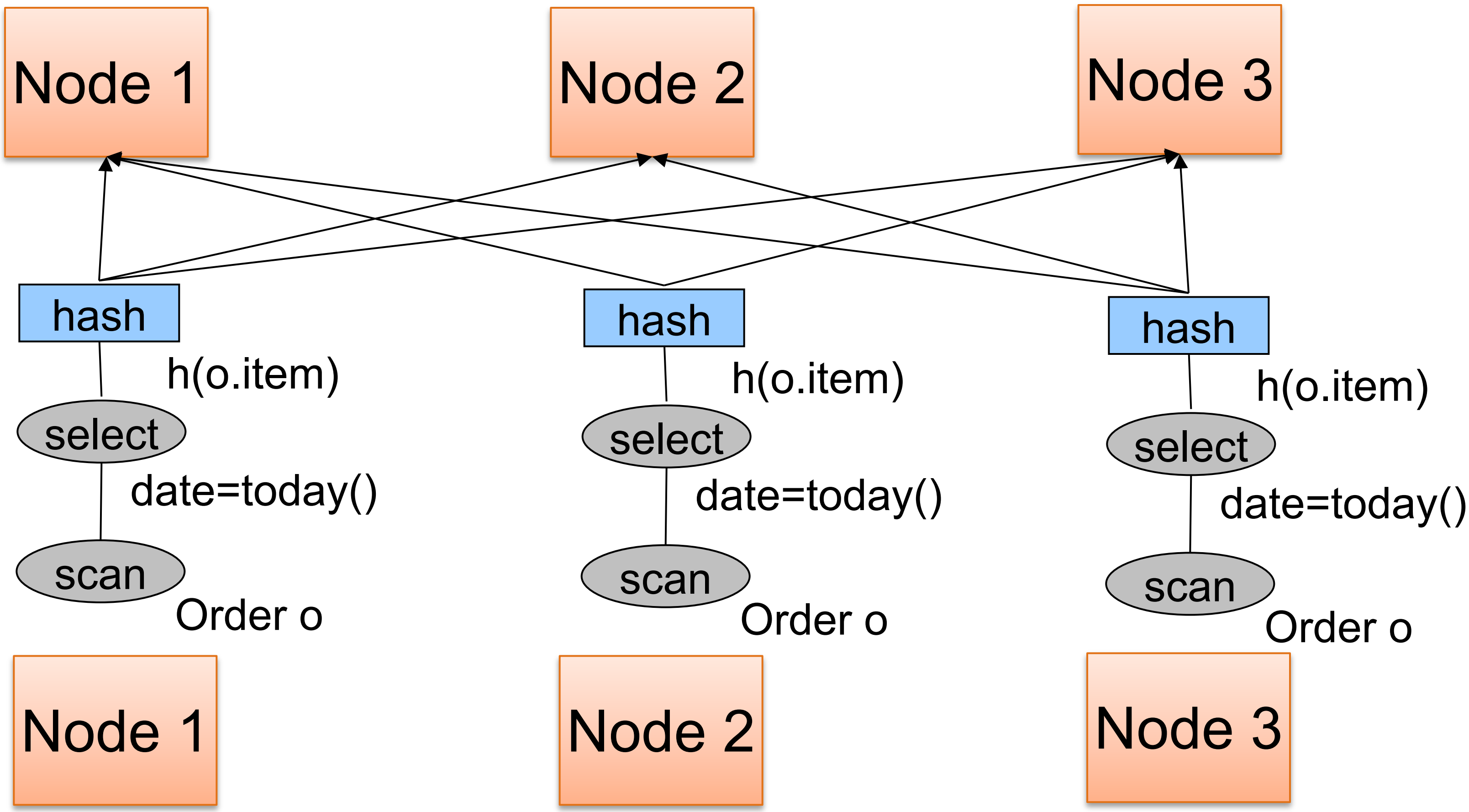
```
SELECT *
FROM Order o, Line i
WHERE o.item = i.item
    AND o.date = today()
```



Order(oid, item, date), Line(item, …)

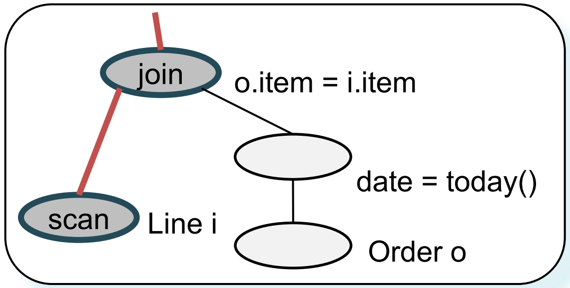


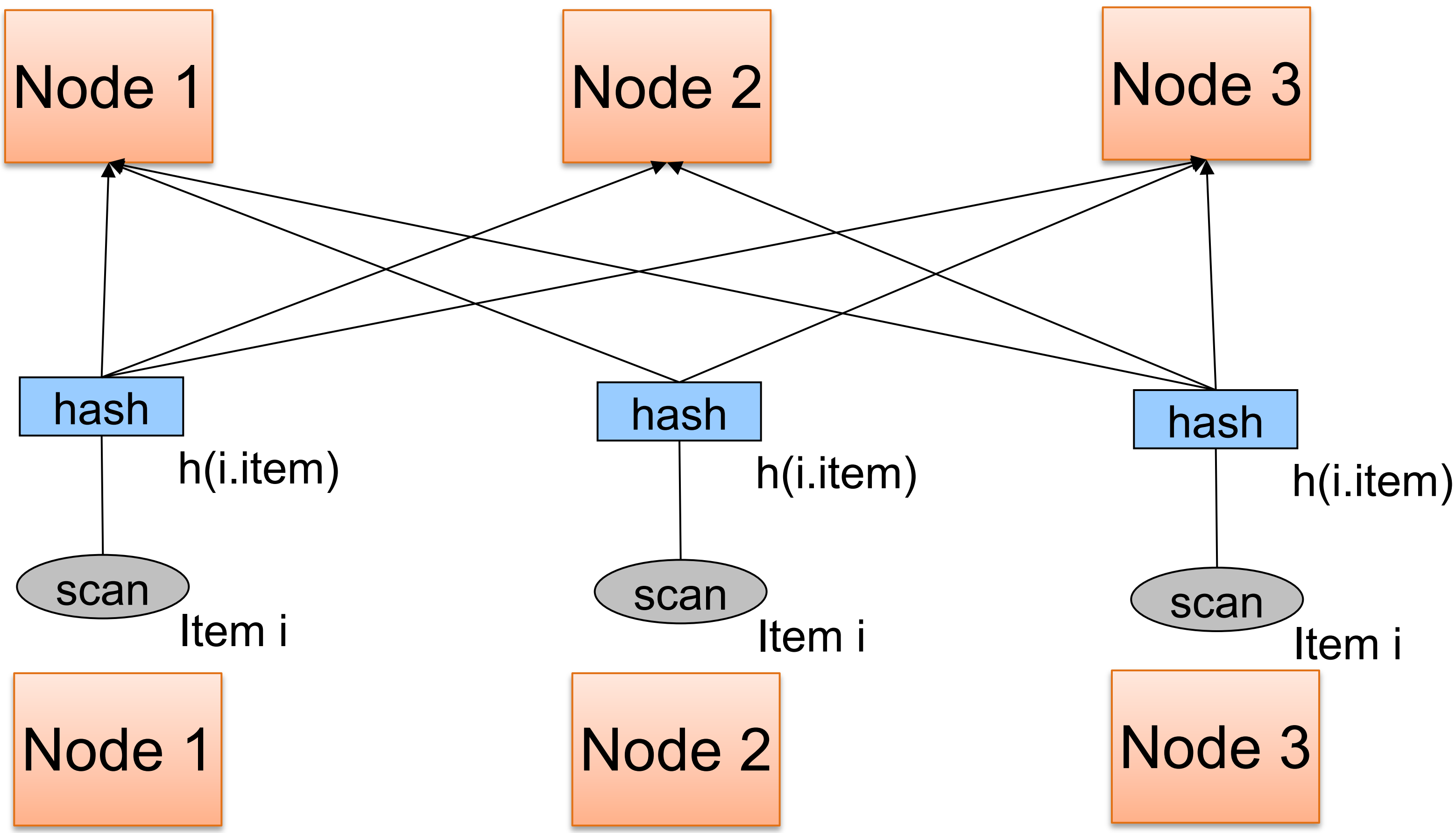
Query Execution



Order(oid, item, date), Line(item, …)

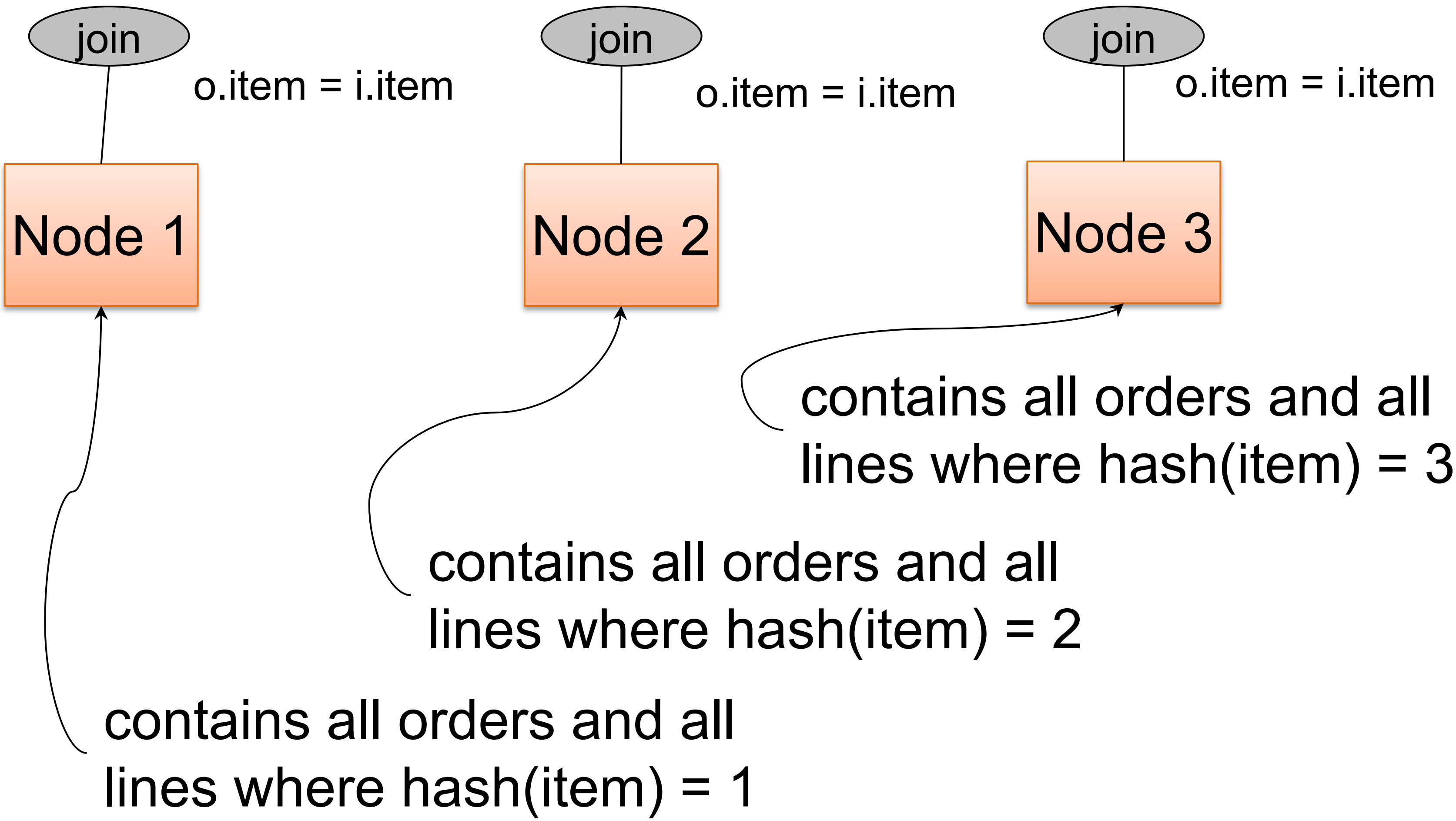
Query Executi





Order(oid, item, date), Line(item, …)

Query Execution

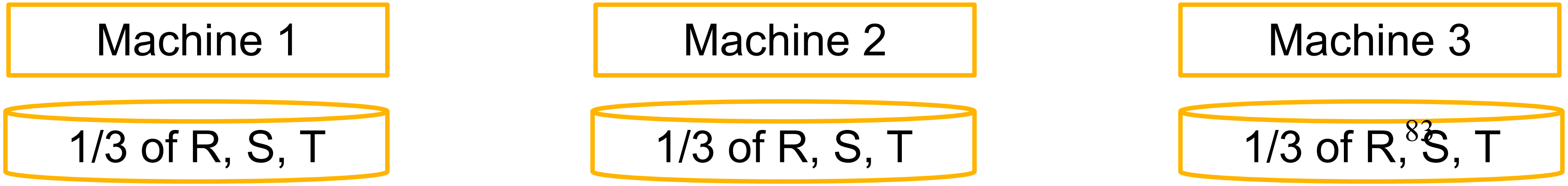


Example 2

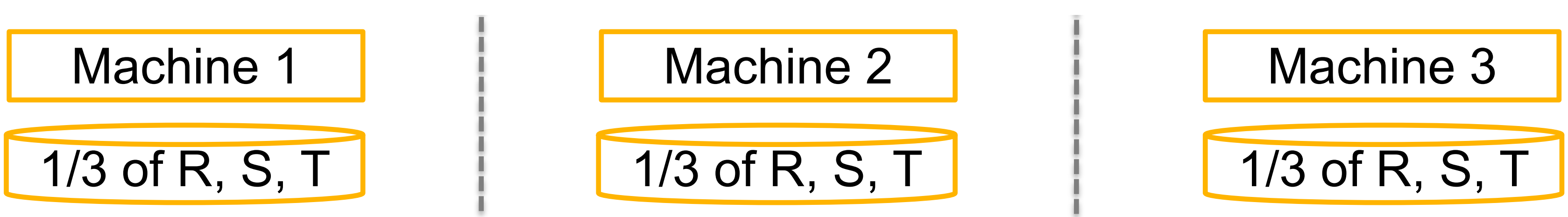
SELECT *

FROM R, S, T

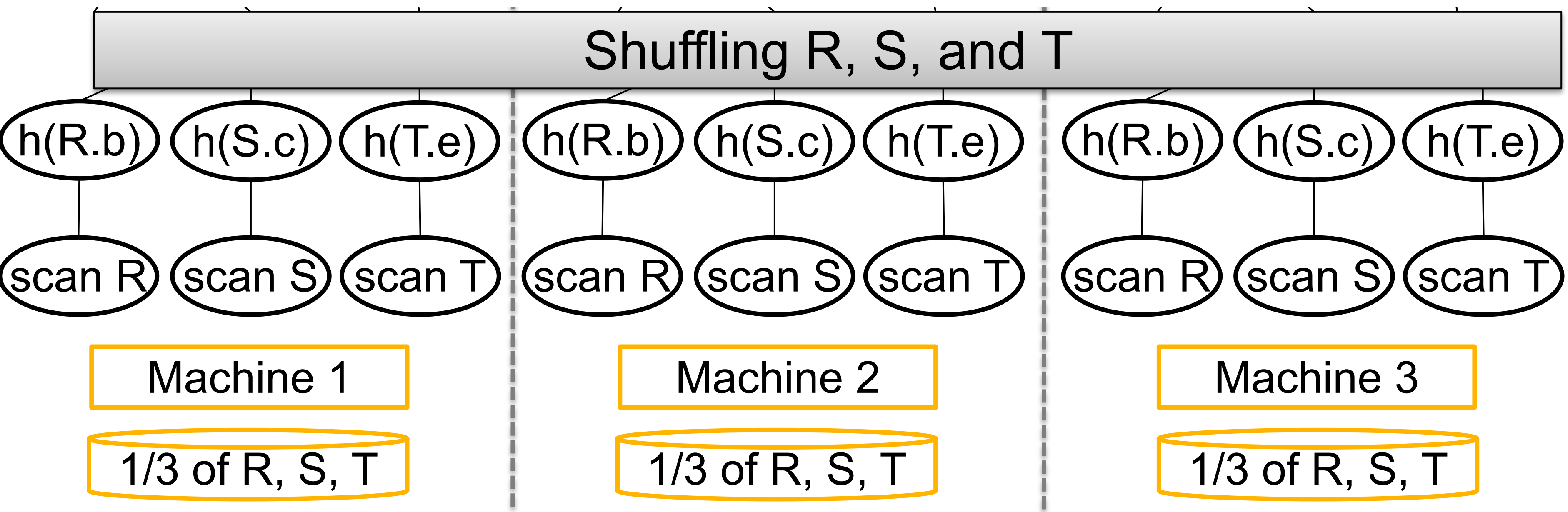
WHERE R.b = S.c AND S.d = T.e AND (R.a - T.f) > 100



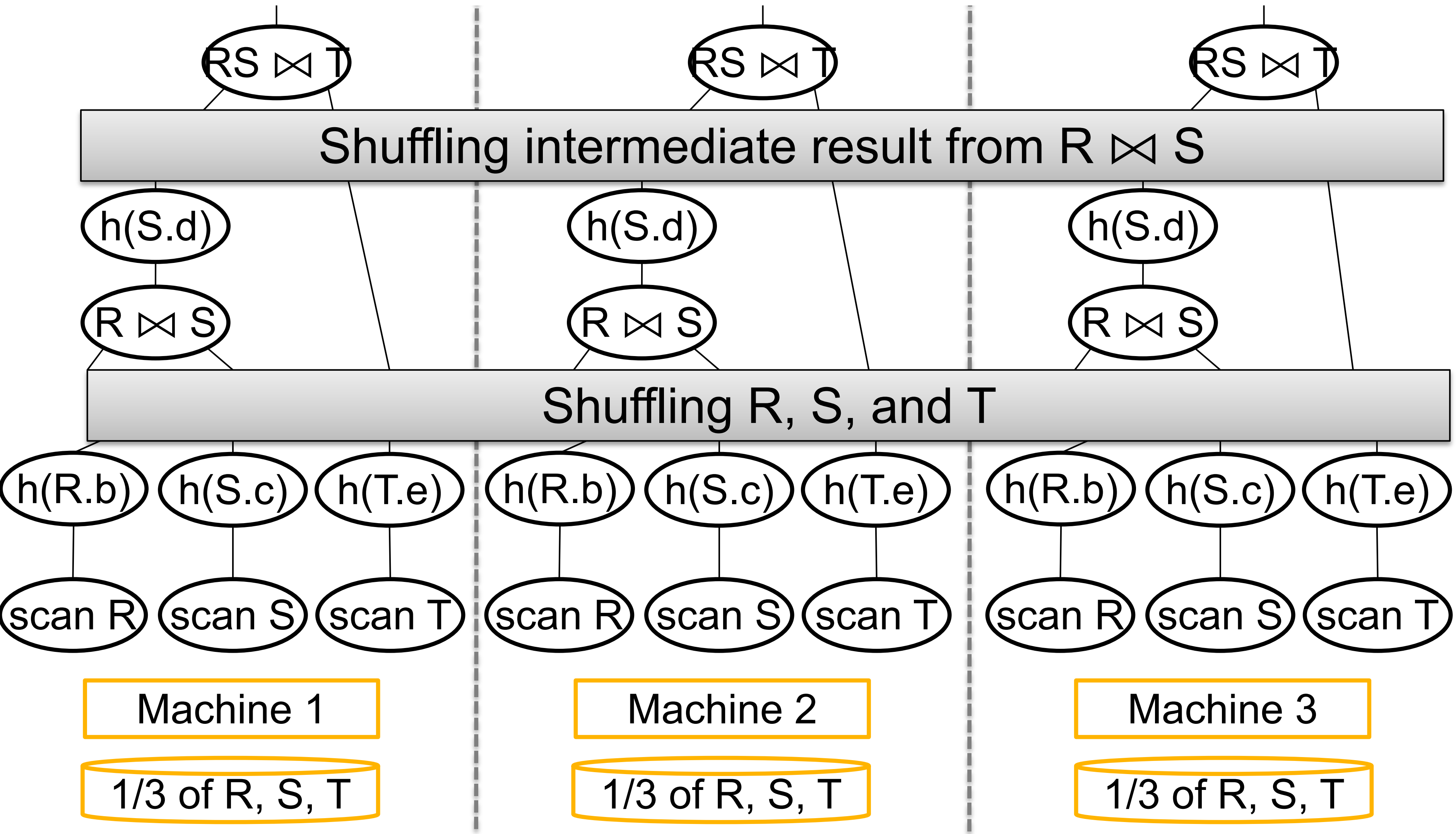
… WHERE R.b = S.c AND S.d = T.e AND (R.a - T.f) > 100



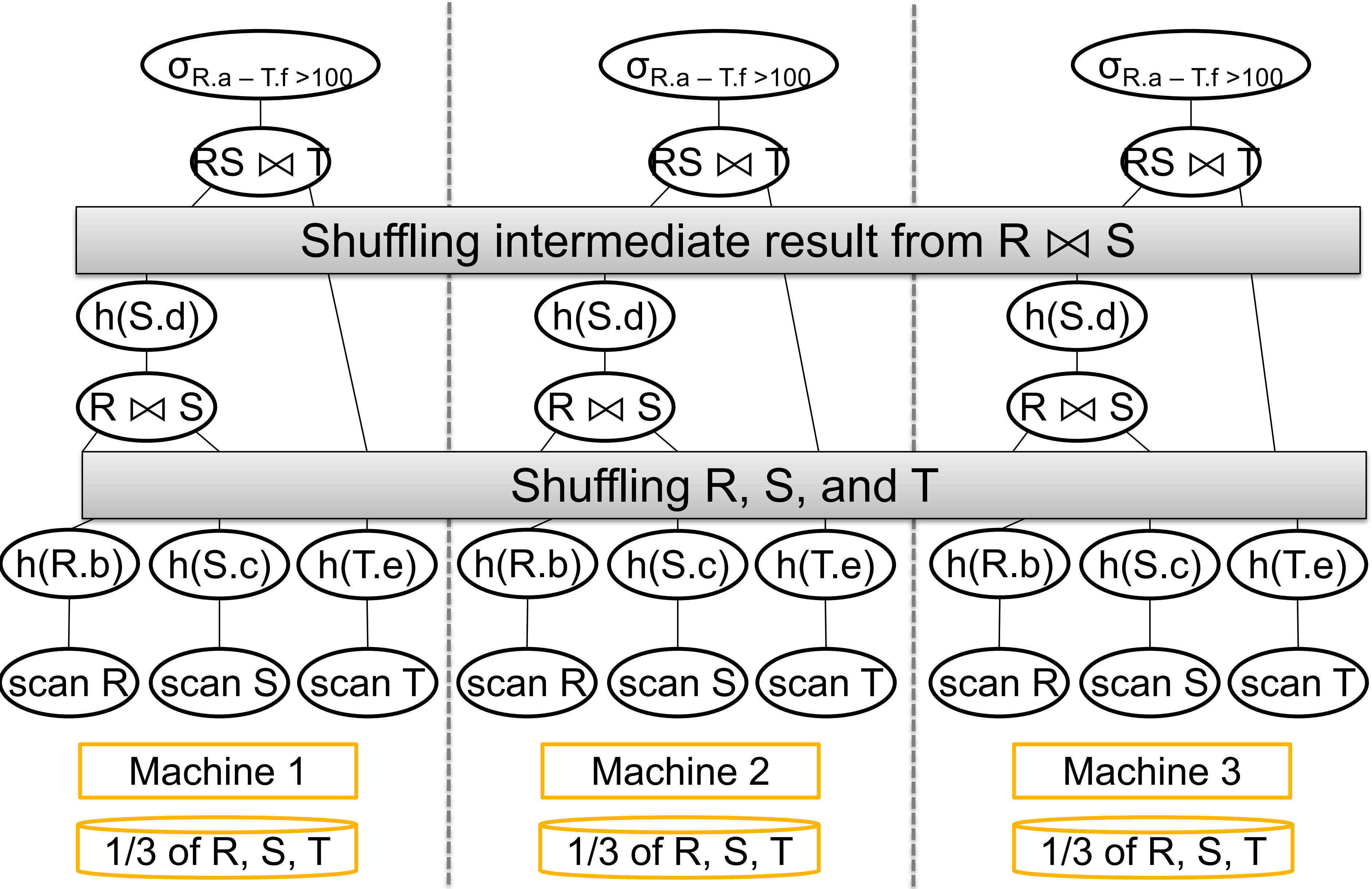
… WHERE R.b = S.c AND S.d = T.e AND (R.a - T.f) > 100



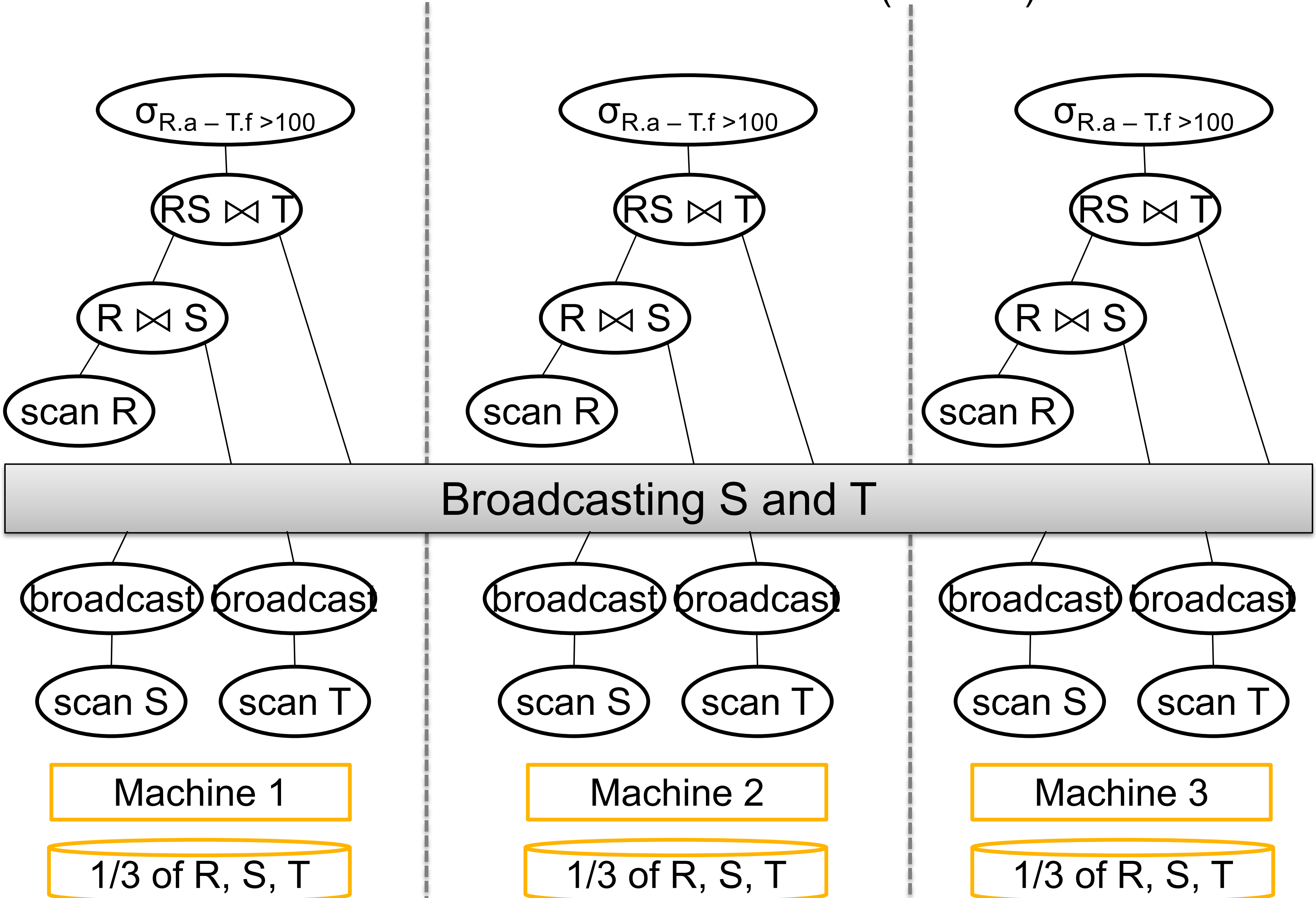
… WHERE R.b = S.c AND S.d = T.e AND (R.a - T.f) > 100



… WHERE R.b = S.c AND S.d = T.e AND (R.a - T.f) > 100



… WHERE R.b = S.c AND S.d = T.e AND (R.a - T.f) > 100



Skew-Join

- Hash-join:
  - Both relations are partitioned (good)
  - May have skew (bad)
- Broadcast join
  - One relation must be broadcast (bad)
  - No worry about skew (good)
- Skew join (has other names):
  - Combine both: in class