DATA516/CSED516 Scalable Data Systems and Algorithms

Lecture 5

Parallel Query Execution (cont.) + Graphs

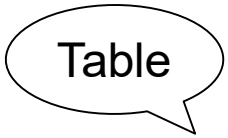
Announcements

- HW2 (Spark) will be released today
  - Due November 21st
  - Pull upstream for new assignment
  - Spark intro video on Canvas

- Project Milestone Due November 24th
- Thanksgiving week poll

Distributed Query Processing Algorithms

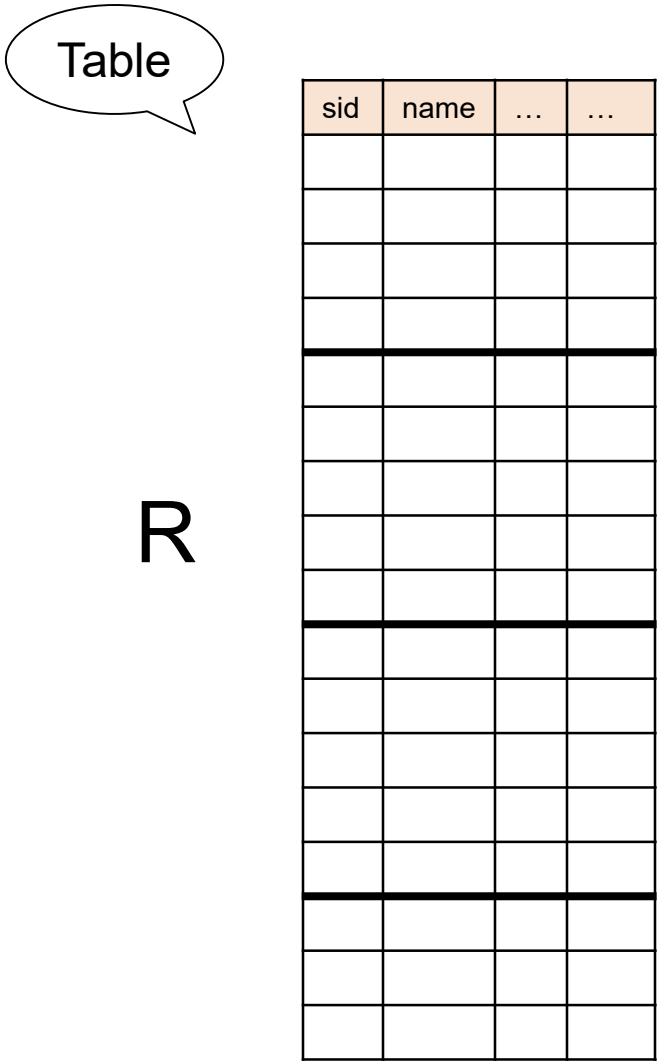
Horizontal Data Partitioning



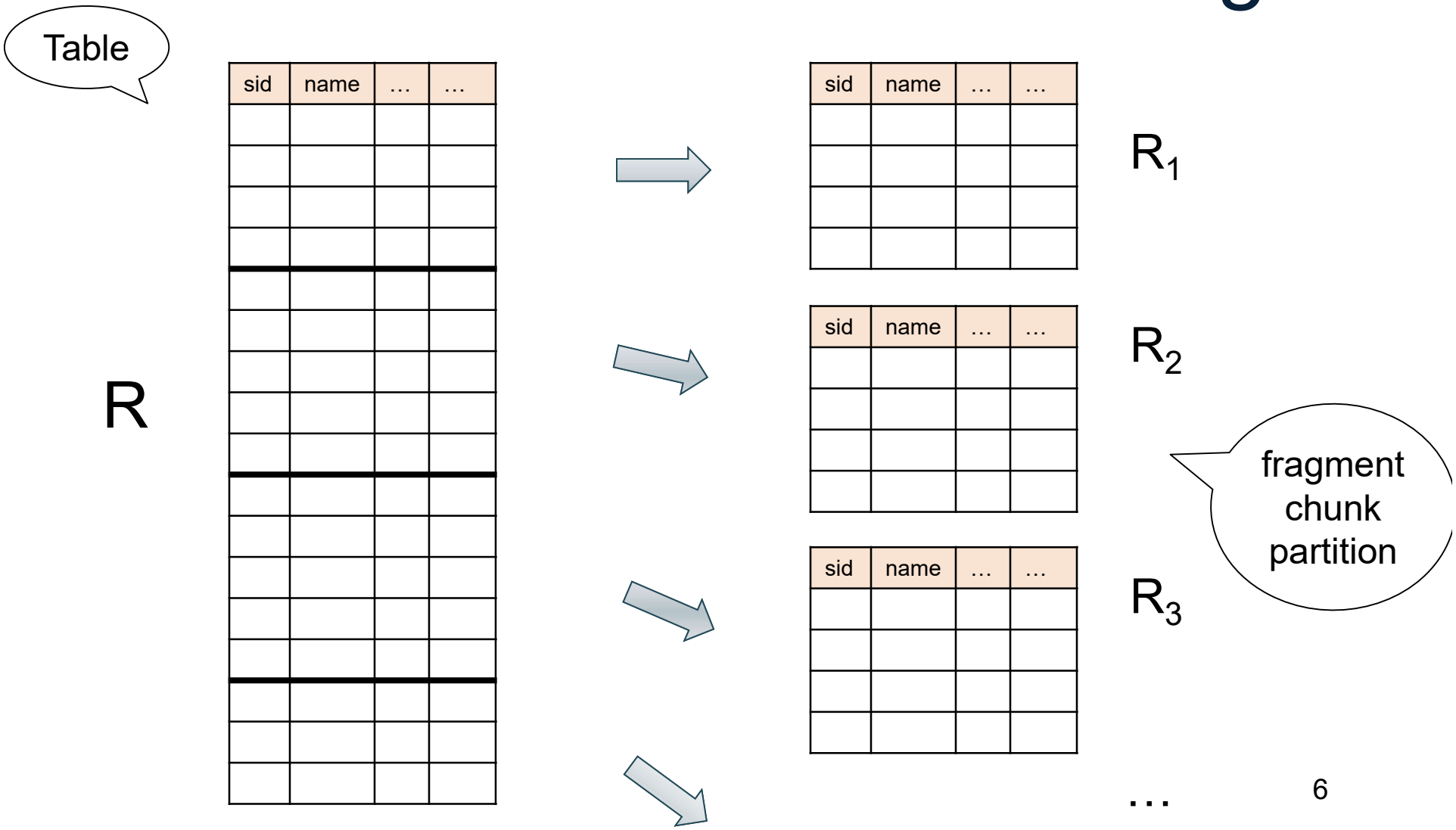
R

sid	name	…	…

Horizontal Data Partitioning



Horizontal Data Partitioning



Horizontal Data Partitioning

- Block Partition, a.k.a. Round Robin:
  - Partition tuples arbitrarily s.t. size(R_1)≈ … ≈ size(R_P)
- Hash partitioned on attribute A:
  - Tuple t goes to chunk i, where i = h(t.A) mod 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 ɣ

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: (on A)

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: (on A)

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:
  - Point query: only one server needs to scan
  - Range query: all servers need to scan
- Range partitioned: (on A)

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:
  - Point query: only one server needs to scan
  - Range query: all servers need to scan
- Range partitioned: (on A)
  - Only some servers need to scan

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
- 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(K, A, B, C)

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

Discuss in class how to compute in each case:

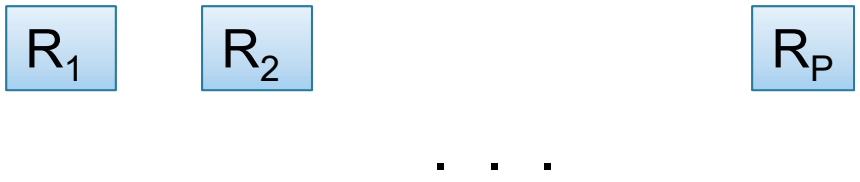
- R is hash-partitioned on A
  - Each server i computes locally $\gamma_{A,sum(C)}(R_i)$
- R is block-partitioned or hash-partitioned on K
  - Need to reshuffle data on A first (next slide)
  - Then compute locally $\gamma_{A,sum(C)}(R_i)$

Basic Parallel GroupBy

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

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

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

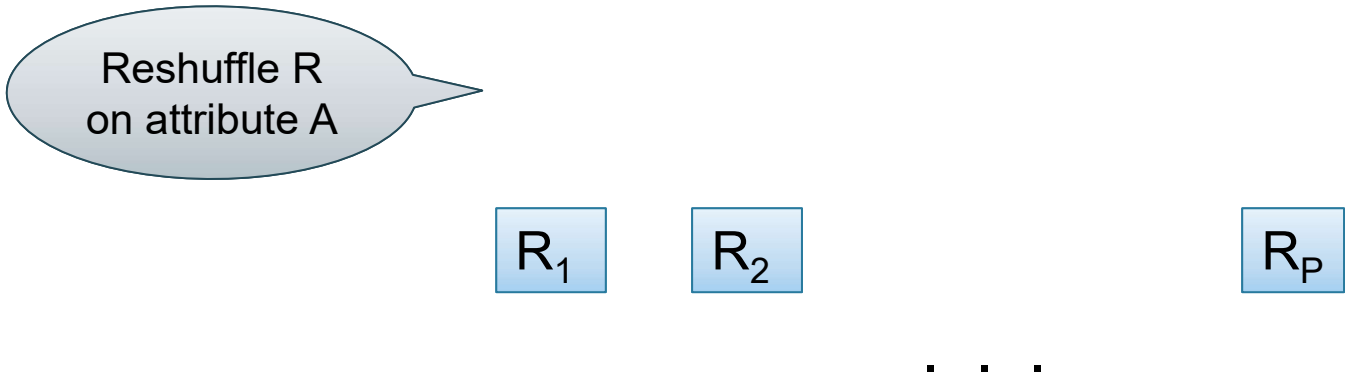


Basic Parallel GroupBy

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

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

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

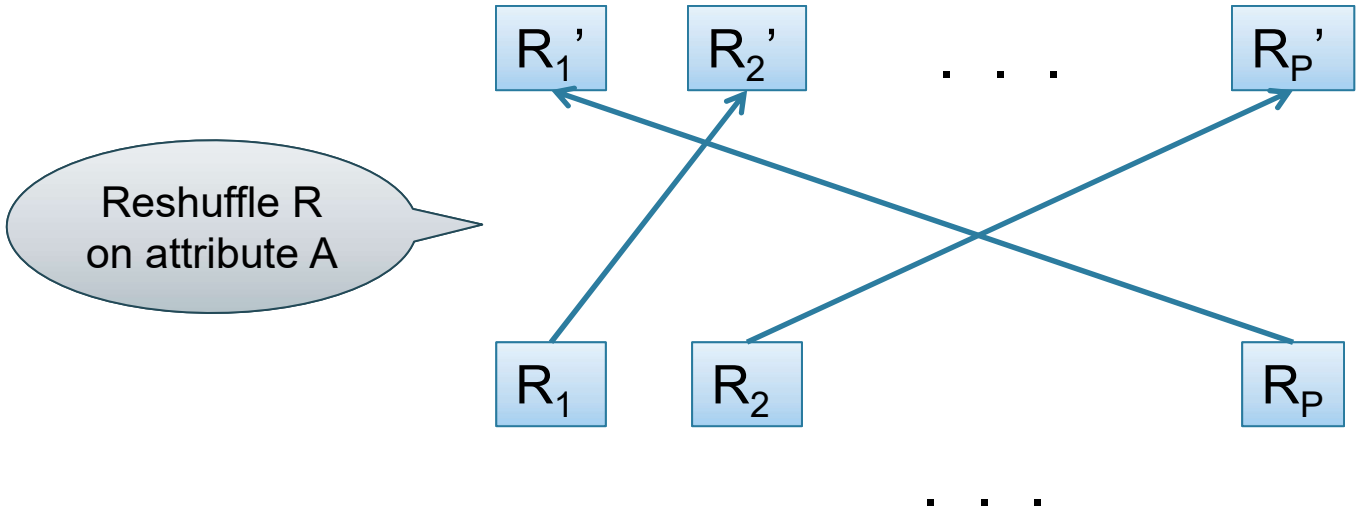


Basic Parallel GroupBy

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

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

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

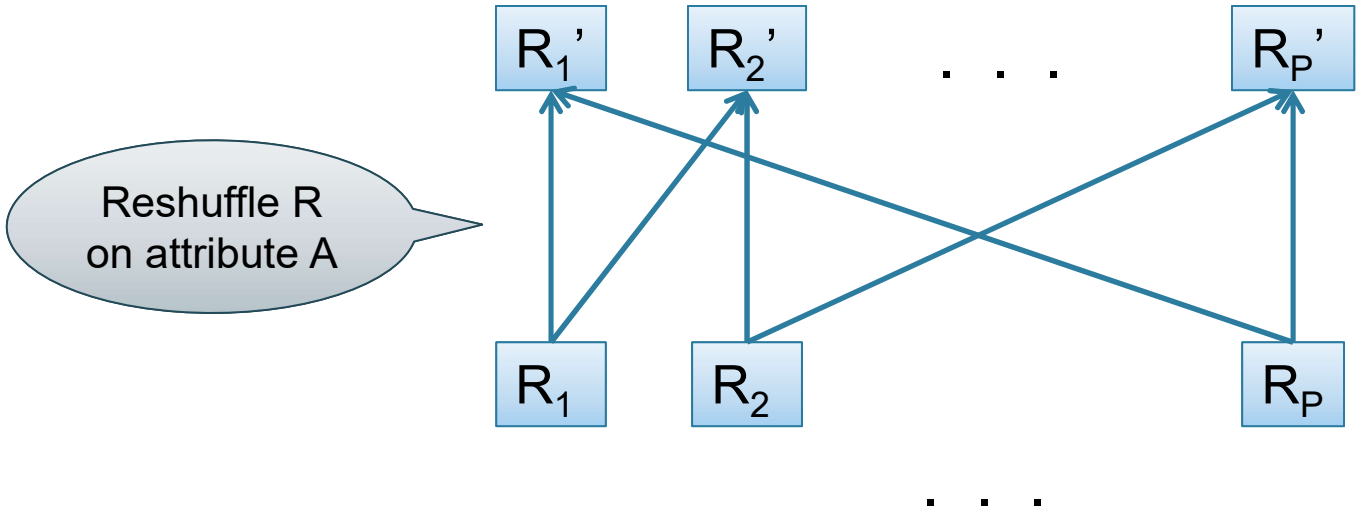


Basic Parallel GroupBy

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

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

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

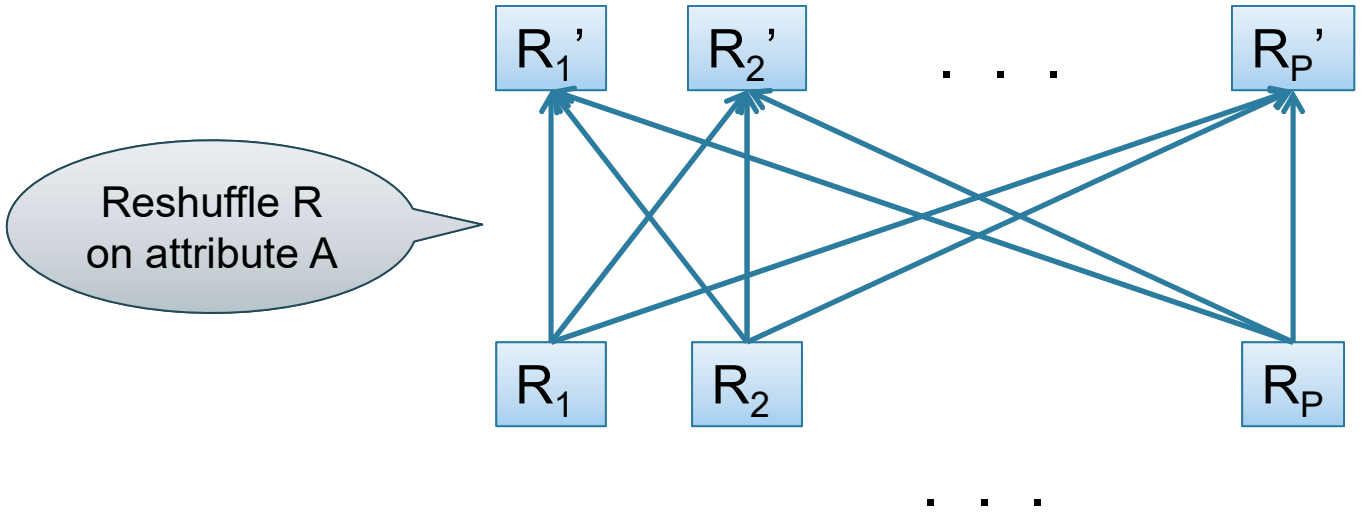


Basic Parallel GroupBy

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

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

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

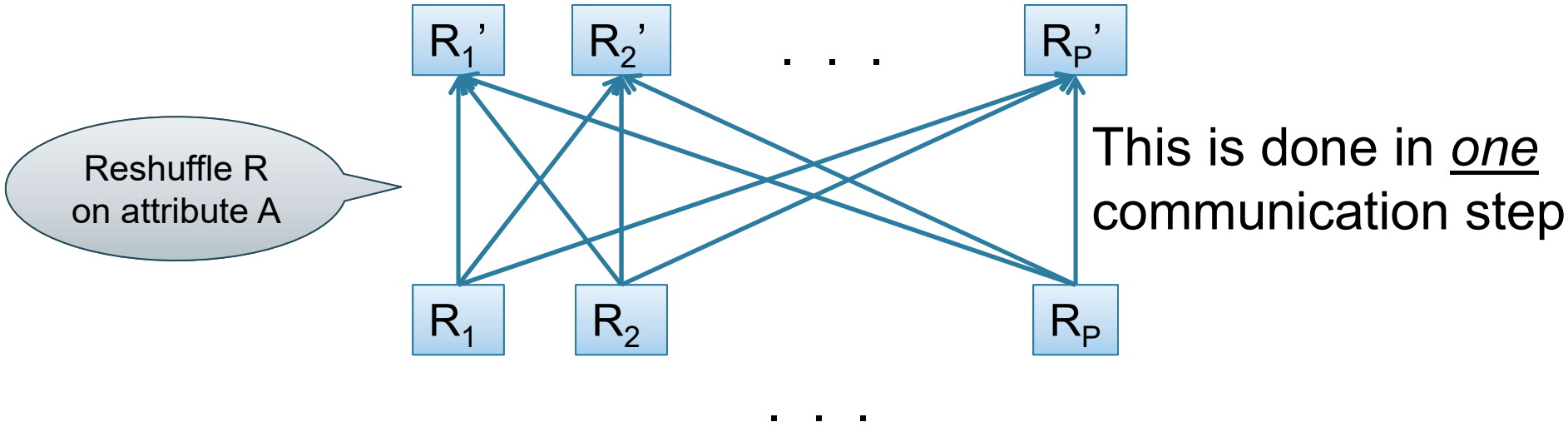


Basic Parallel GroupBy

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

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

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



Reshuffling

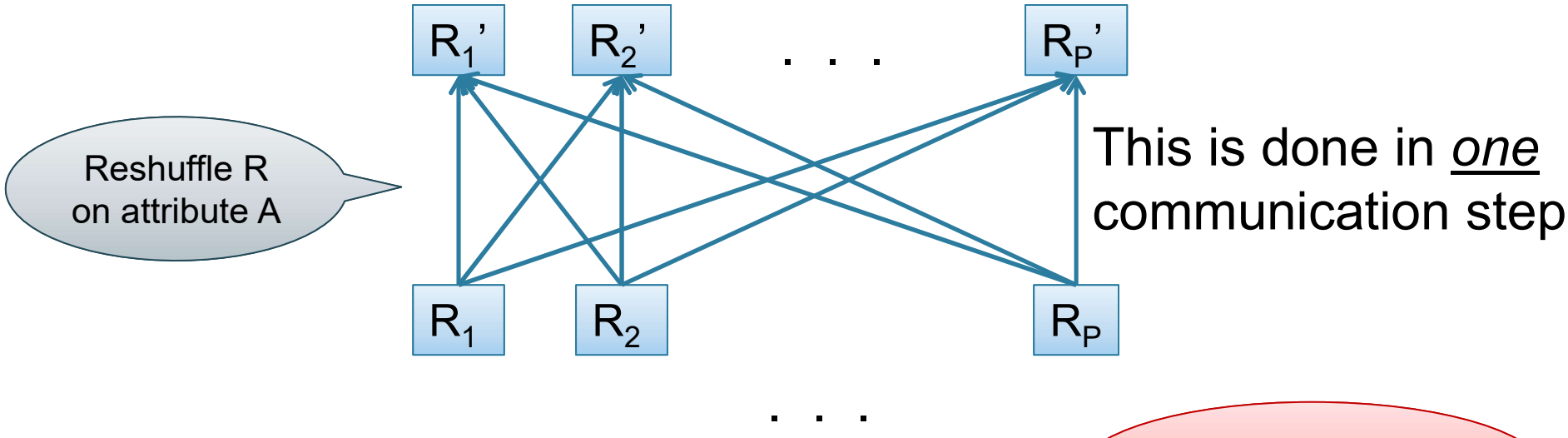
- Nodes send data over the network
- Many-many communications possible
- Throughput:
  - Better than disk
  - Worse than main memory

Basic Parallel GroupBy

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

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

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



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

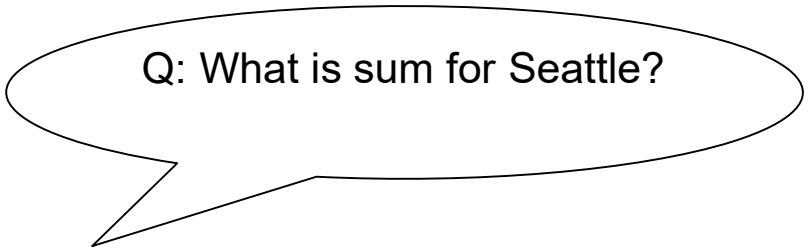
```
SELECT city, sum(quant)
FROM R
GROUP BY city
```

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
```

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

Q: What is sum for Seattle?
A: 106

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

GroupBy/Union Commutativity

	city	…	qant
	Seattle		**10**
	LA		20
	Seattle		**30**
	NY		40

Sum here = 40

Q: What is sum for Seattle?
A: 106

	city	…	qant
	LA		22
	NY		33
	LA		44
	Austin		55

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

	city	…	qant
	Seattle		**66**
	LA		77
	NY		88
	LA		99

Sum here = 66

GroupBy/Union Commutativity

	city	…	qant
	Seattle		**10**
	LA		20
	Seattle		**30**
	NY		40

Sum here = 40

Q: What is sum for Seattle?
A: 106

	city	…	qant
	LA		22
	NY		33
	LA		44
	Austin		55

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

	city	…	qant
	Seattle		**66**
	LA		77
	NY		88
	LA		99

Sum here = 66

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

GroupBy/Union Commutativity

	city	…	qant
	Seattle		**10**
	LA		20
	Seattle		**30**
	NY		40

Sum here = 40

Q: What is sum for Seattle?
A: 106

	city	…	qant
	LA		22
	NY		33
	LA		44
	Austin		55

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

	city	…	qant
	Seattle		**66**
	LA		77
	NY		88
	LA		99

Sum here = 66

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

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

Basic Parallel GroupBy

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

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:

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

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(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

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?

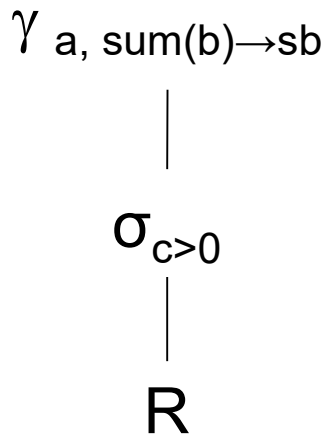
Distributive	Algebraic	Holistic
$\text{sum}(a_1+a_2+\ldots+a_9)=$ $\text{sum}(\text{sum}(a_1+a_2+a_3)+$ $\text{sum}(a_4+a_5+a_6)+$ $\text{sum}(a_7+a_8+a_9))$	avg(B) = sum(B)/count(B)	median(B)

Example Query with Group By

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

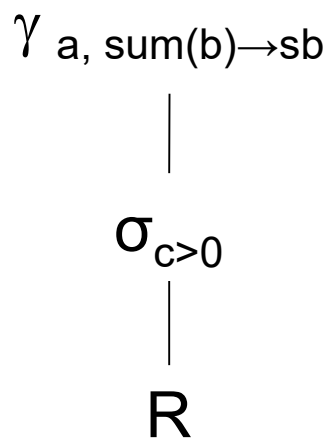
Example Query with Group By

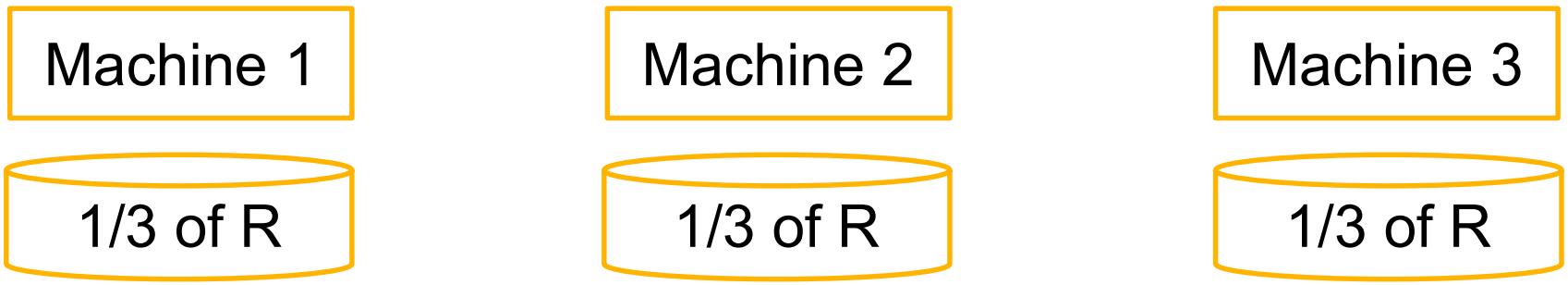
```
SELECT a, sum(b) as sb
FROM R WHERE c > 0
GROUP BY a
```



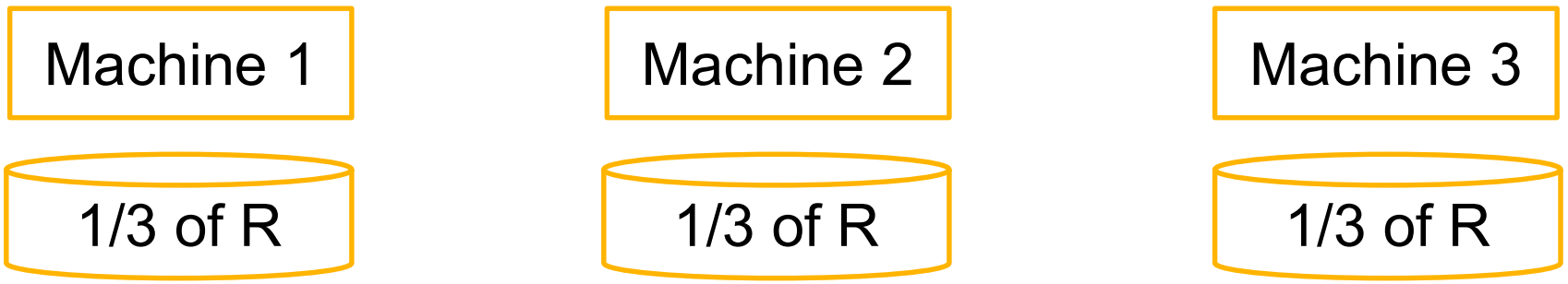
Example Query with Group By

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

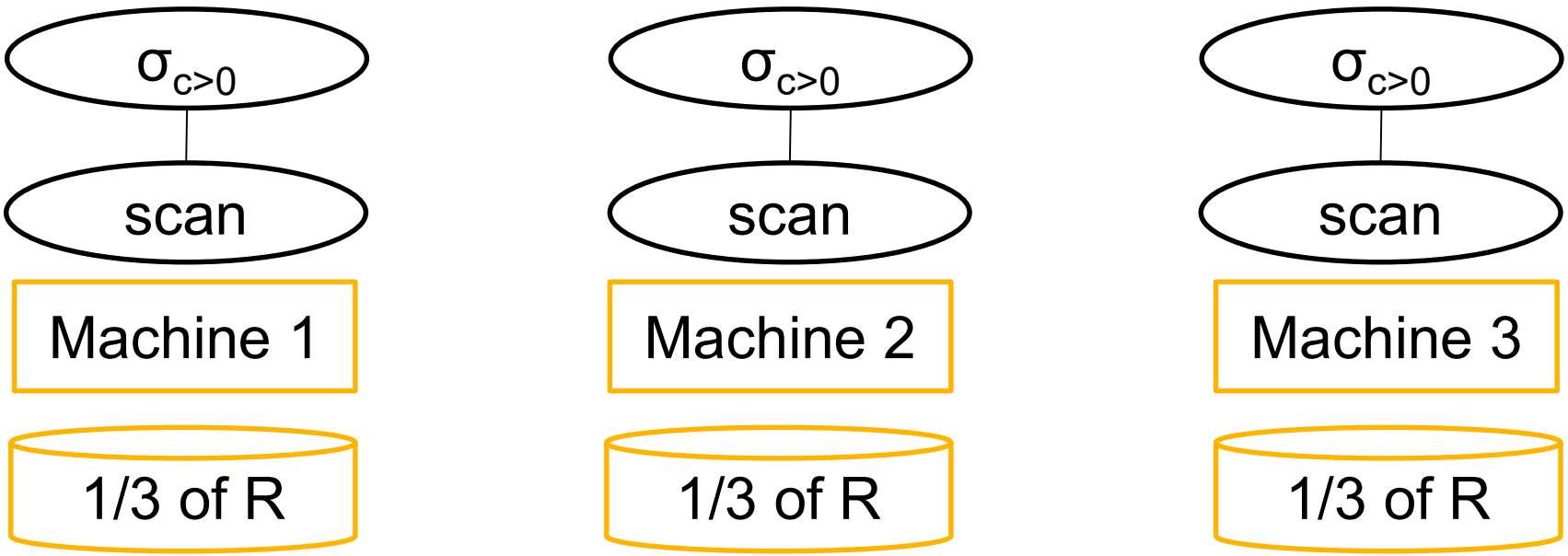




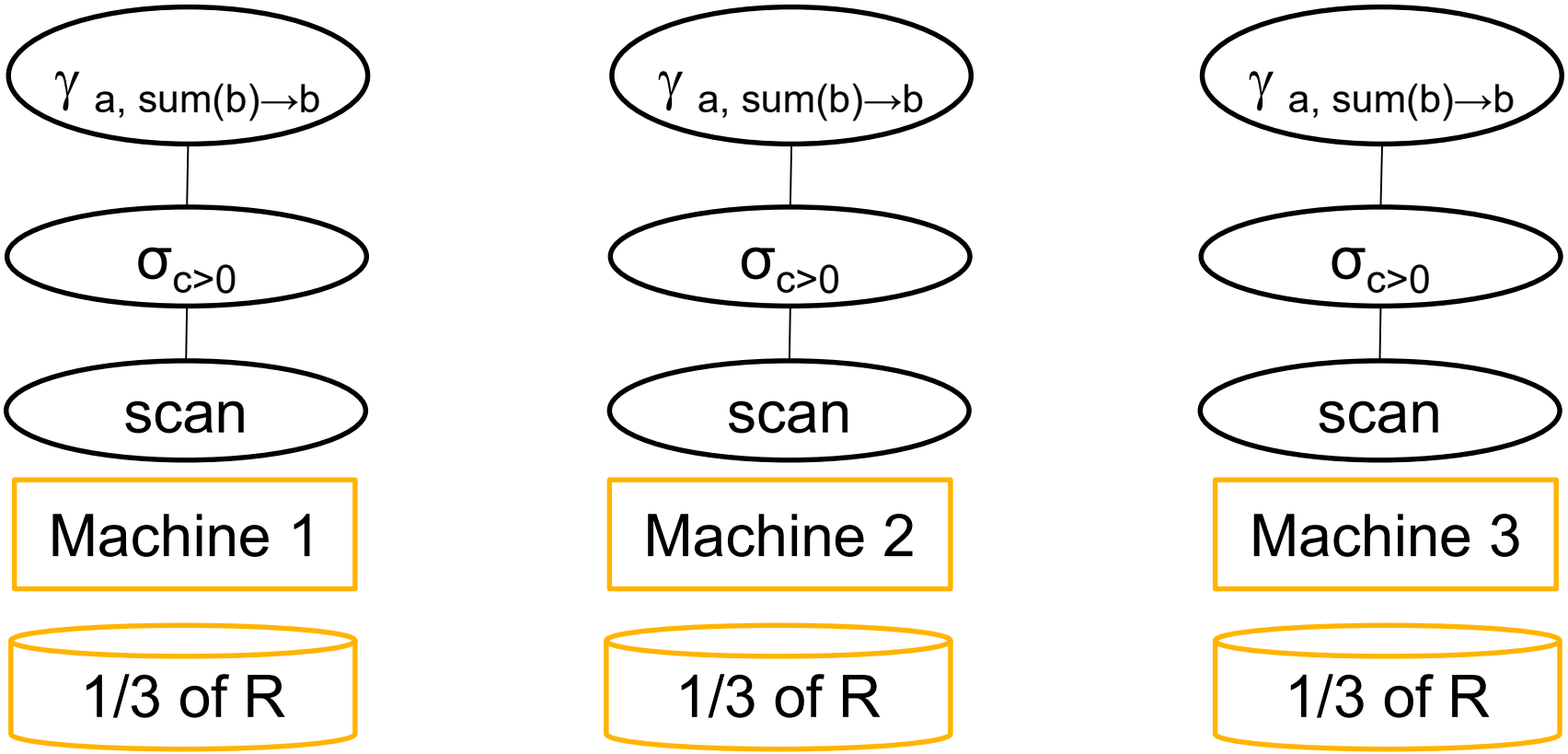
```
SELECT a, sum(b) as sb    FROM R   WHERE c > 0 GROUP BY a
```



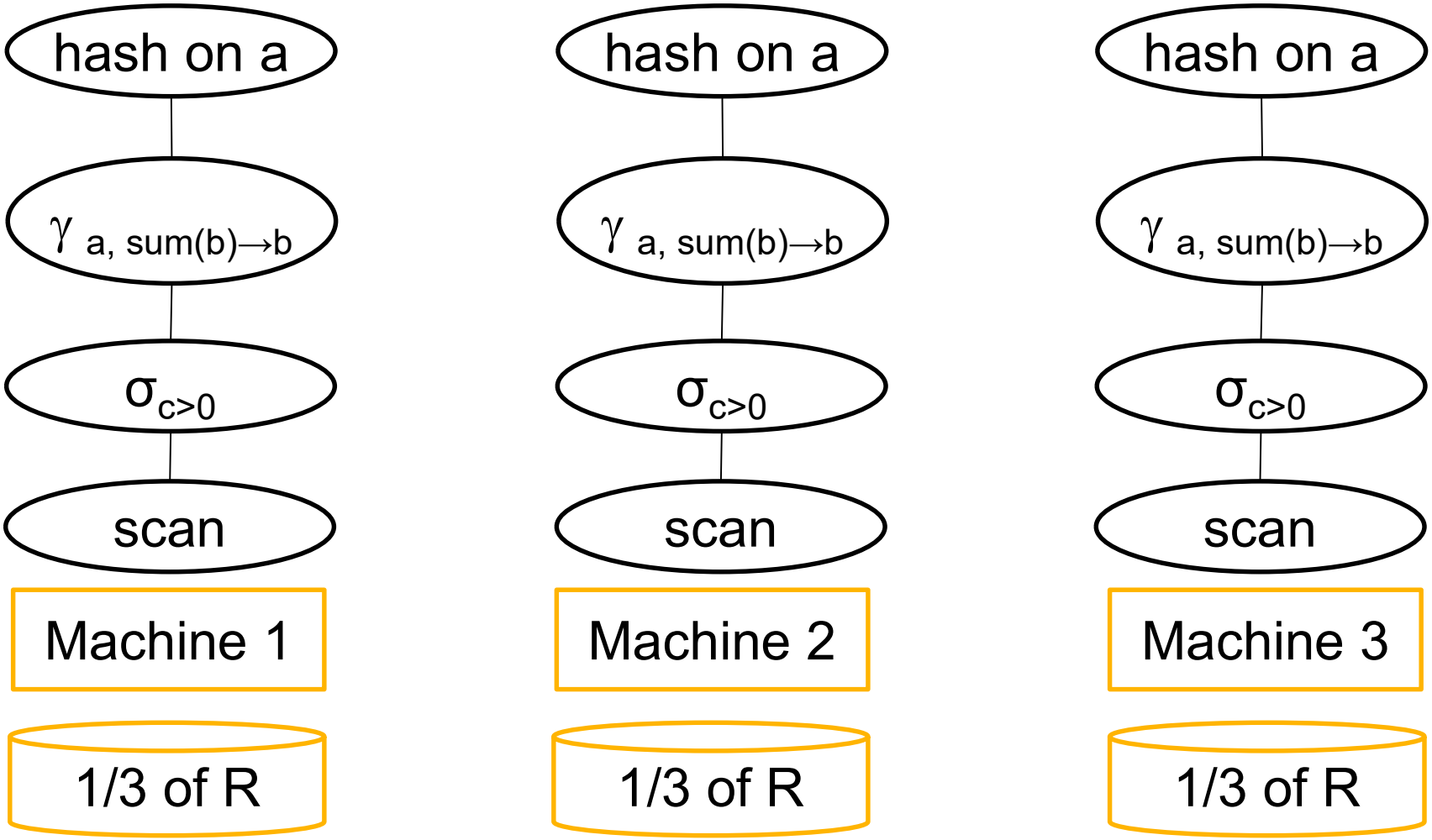
```
SELECT a, sum(b) as sb    FROM R   WHERE c > 0 GROUP BY a
```

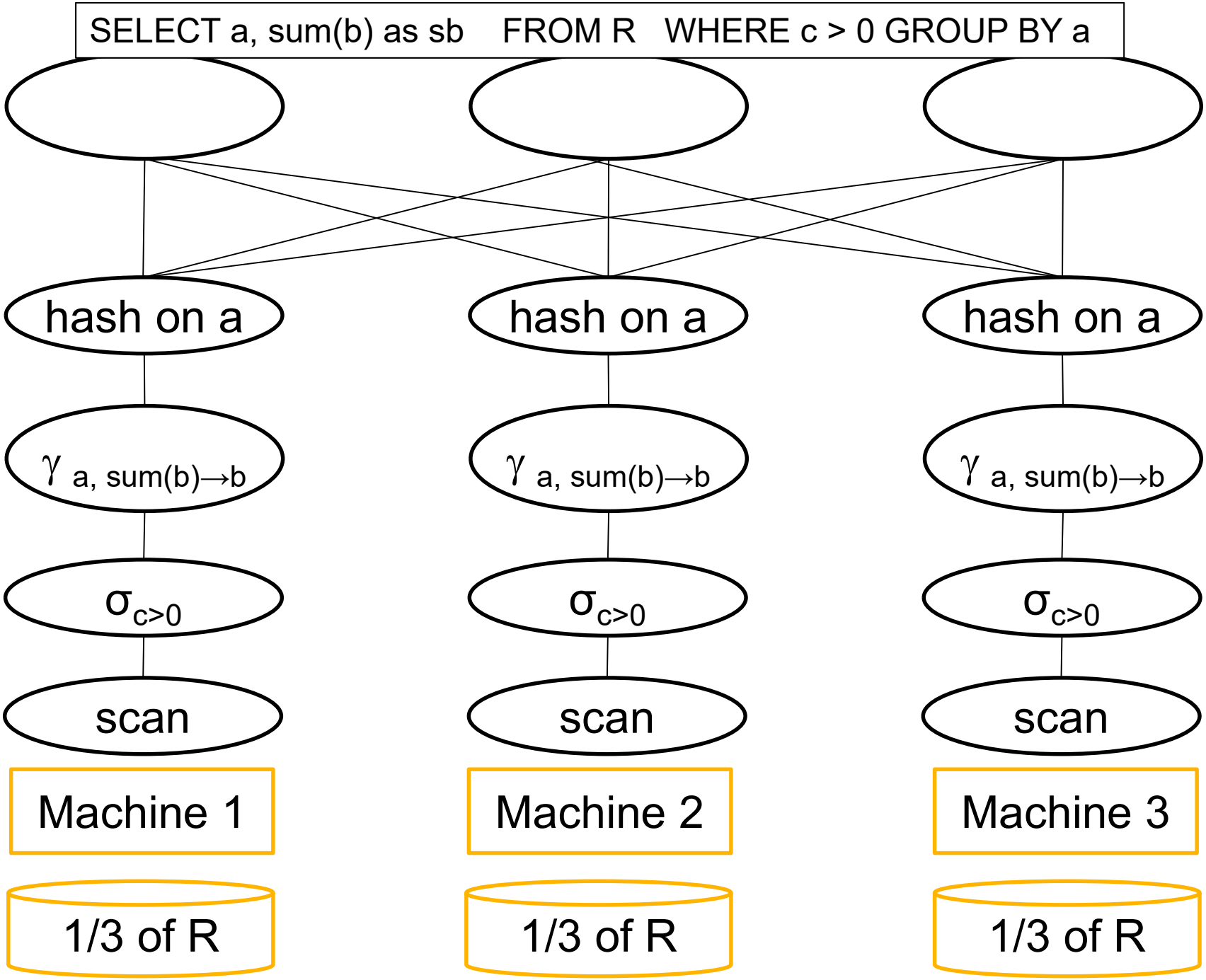


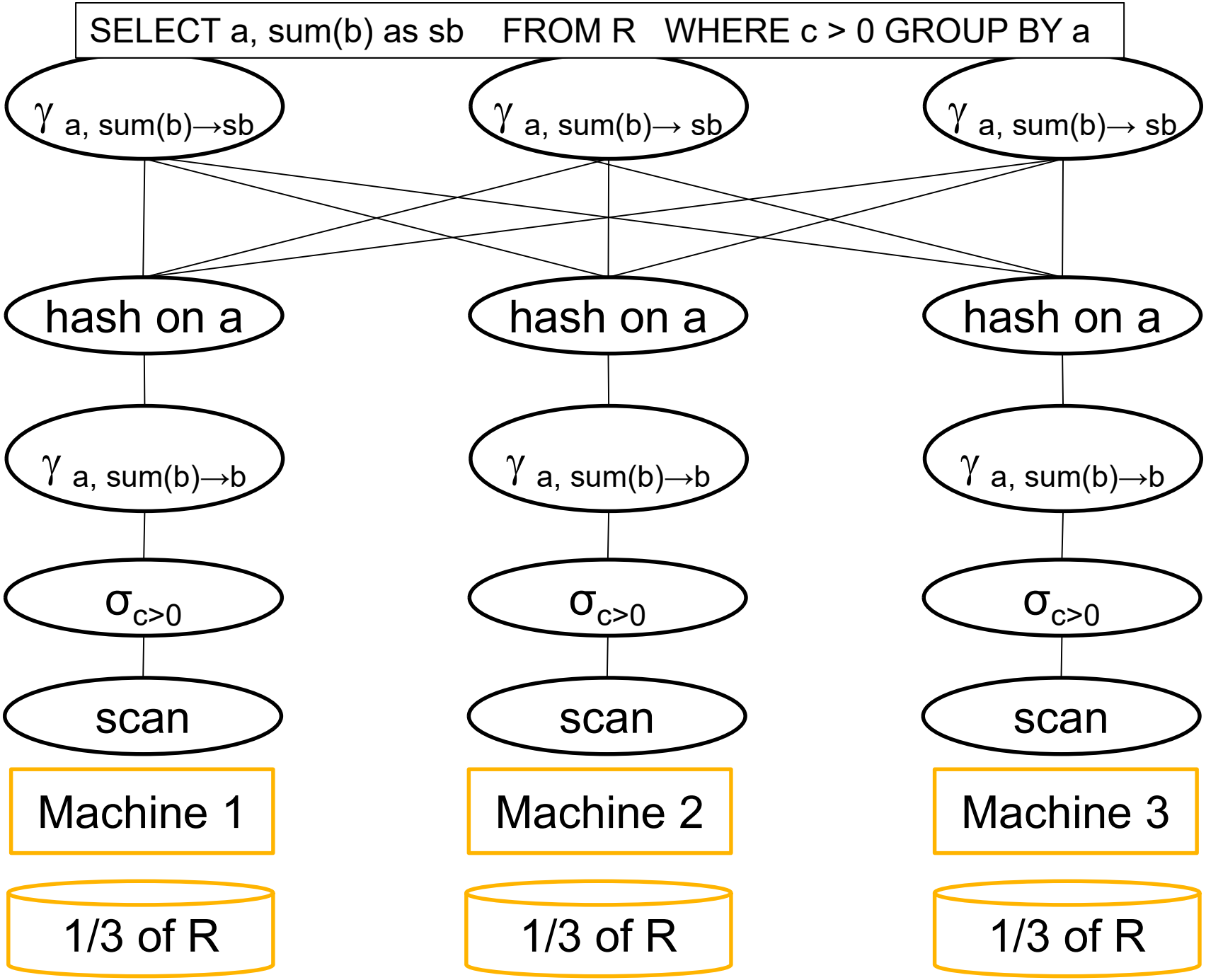
SELECT a, sum(b) as sb FROM R WHERE c > 0 GROUP BY a



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
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


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

Outline

- Distributed Joins
- Skew
- Parallel Query Processing Wrap-up
- Graph

Parallel/Distributed Join

Three “algorithms”:

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

Distributed Hash-Join

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

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

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

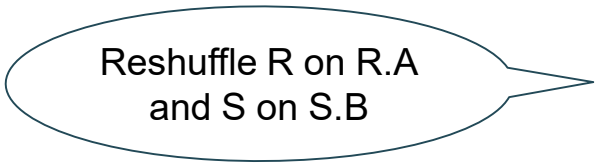
Initially, R and S are block partitioned.
Notice: they may be stored in DFS (recall MapReduce)

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

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

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

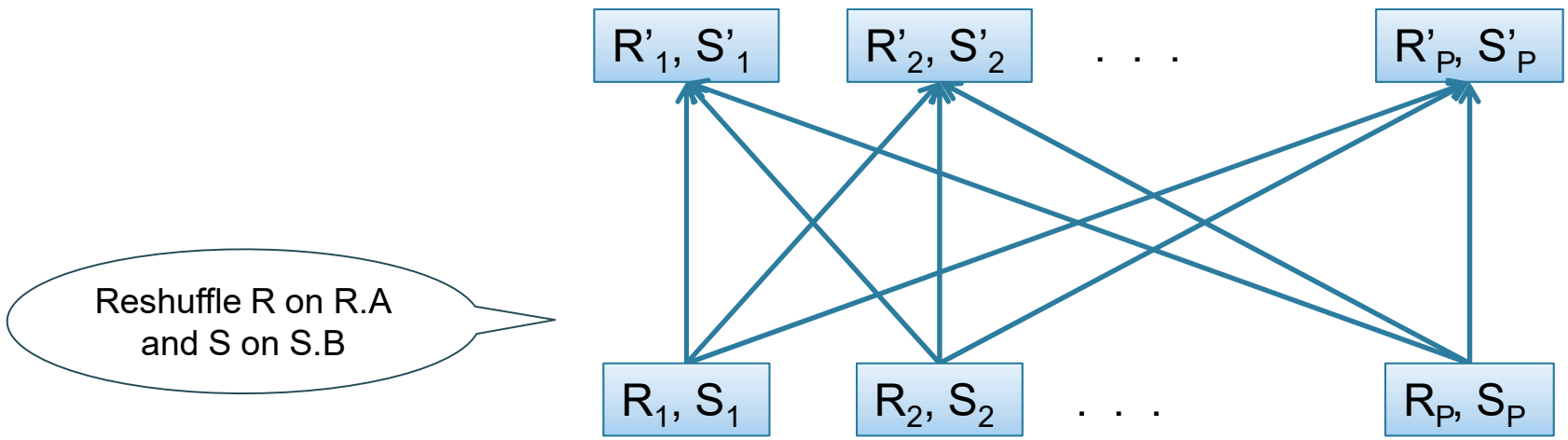
Initially, R and S are block partitioned.
Notice: they may be stored in DFS (recall MapReduce)

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



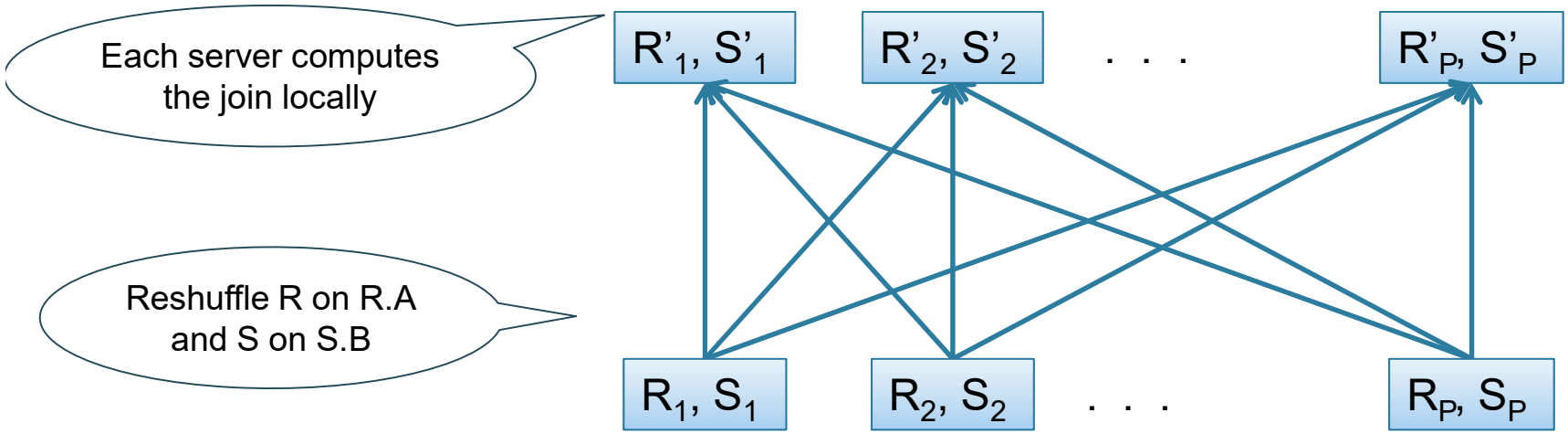
Initially, R and S are block partitioned.
Notice: they may be stored in DFS (recall MapReduce)

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

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

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
A.k.a. “Small Join”

Broadcast Join

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

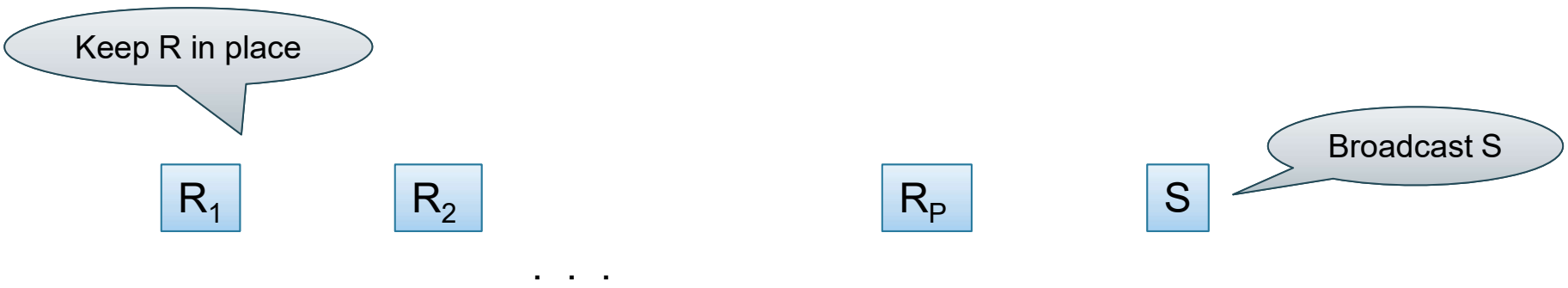
- Called a small join or a broadcast join

Query: R ⋈ S

Broadcast Join

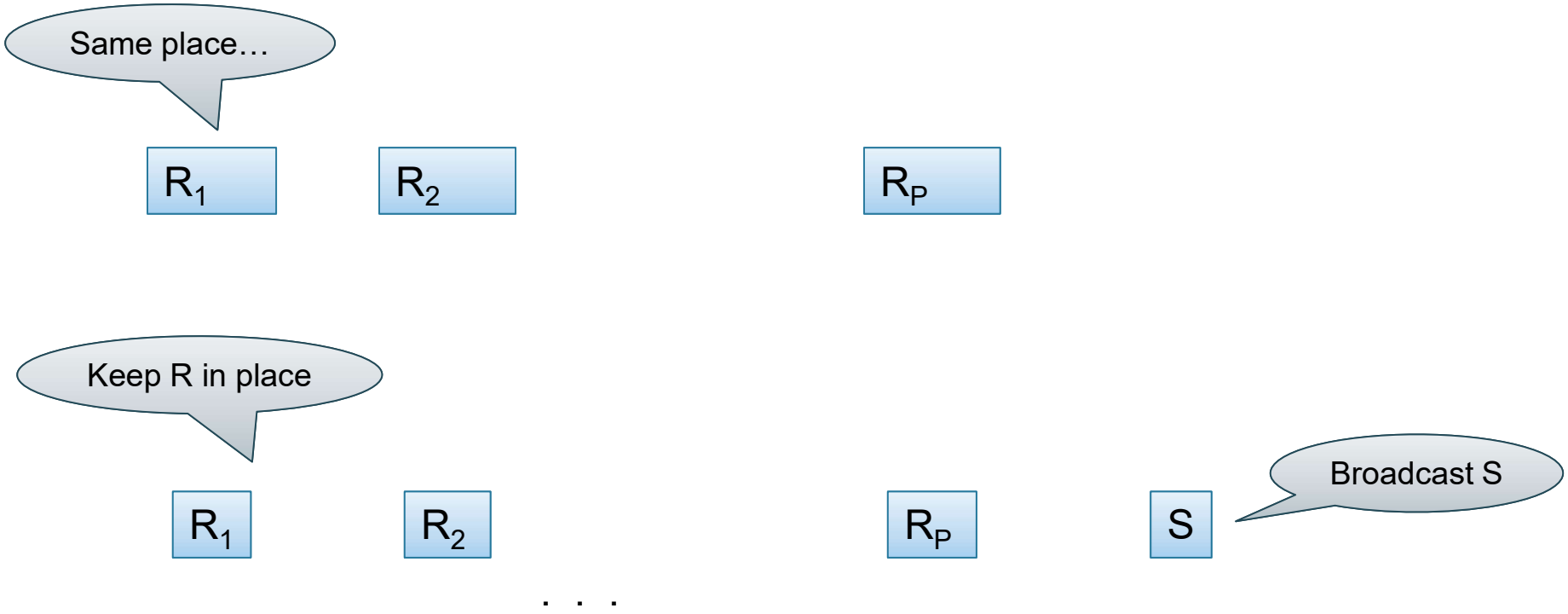
Query: R ⋈ S

Broadcast Join



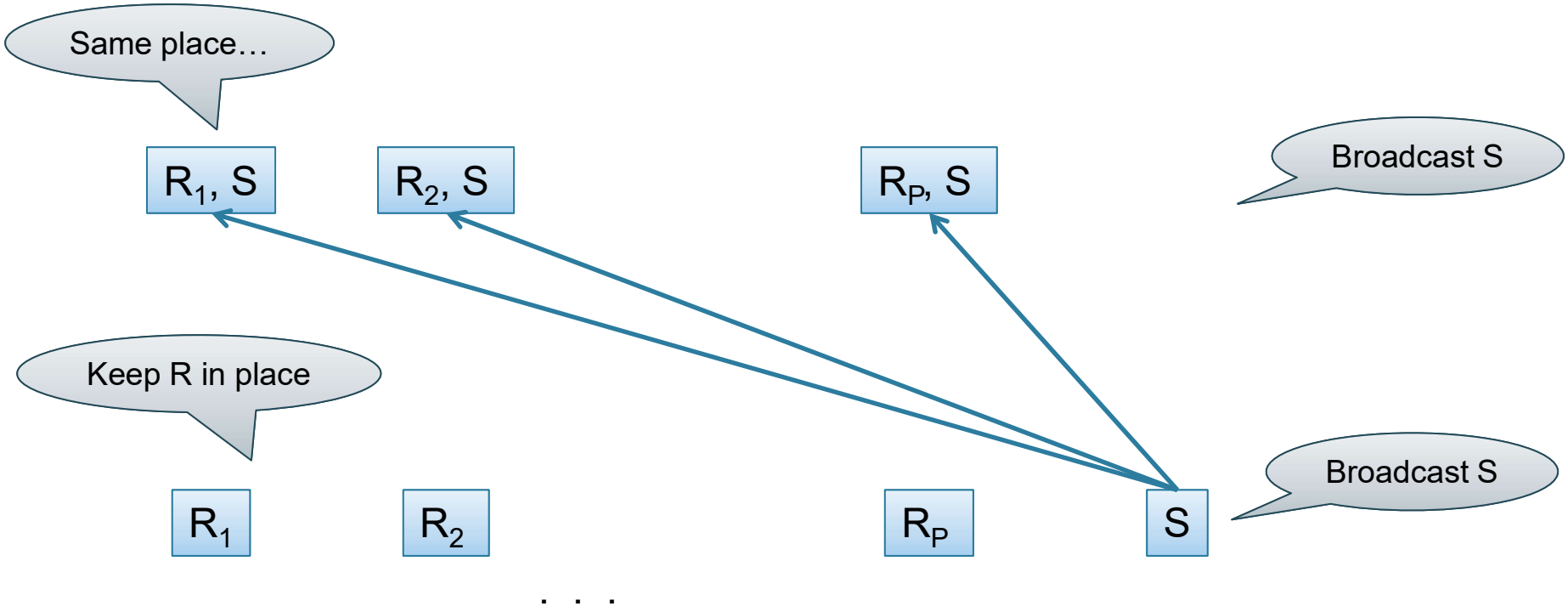
Query: R ⋈ S

Broadcast Join



Query: R ⋈ S

Broadcast Join

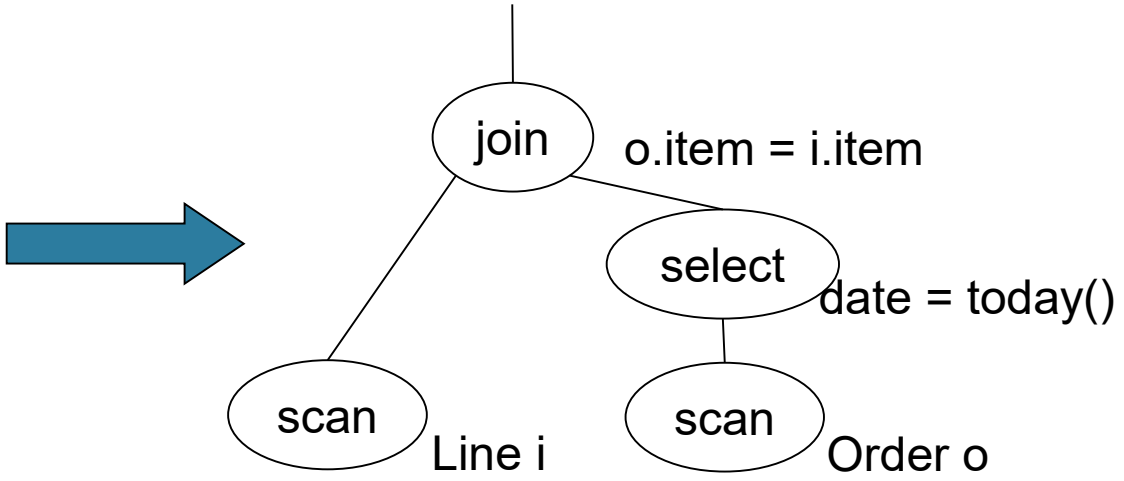


Order(oid, item, date), Line(item, …)

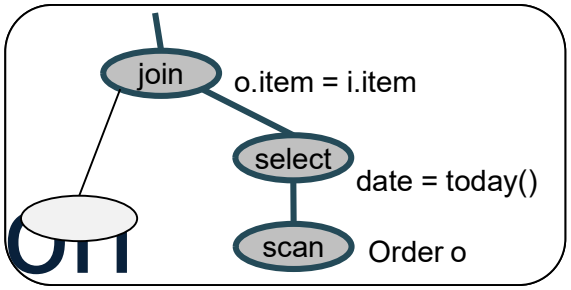
Example Query Execution

Find all orders from today, along with the items ordered

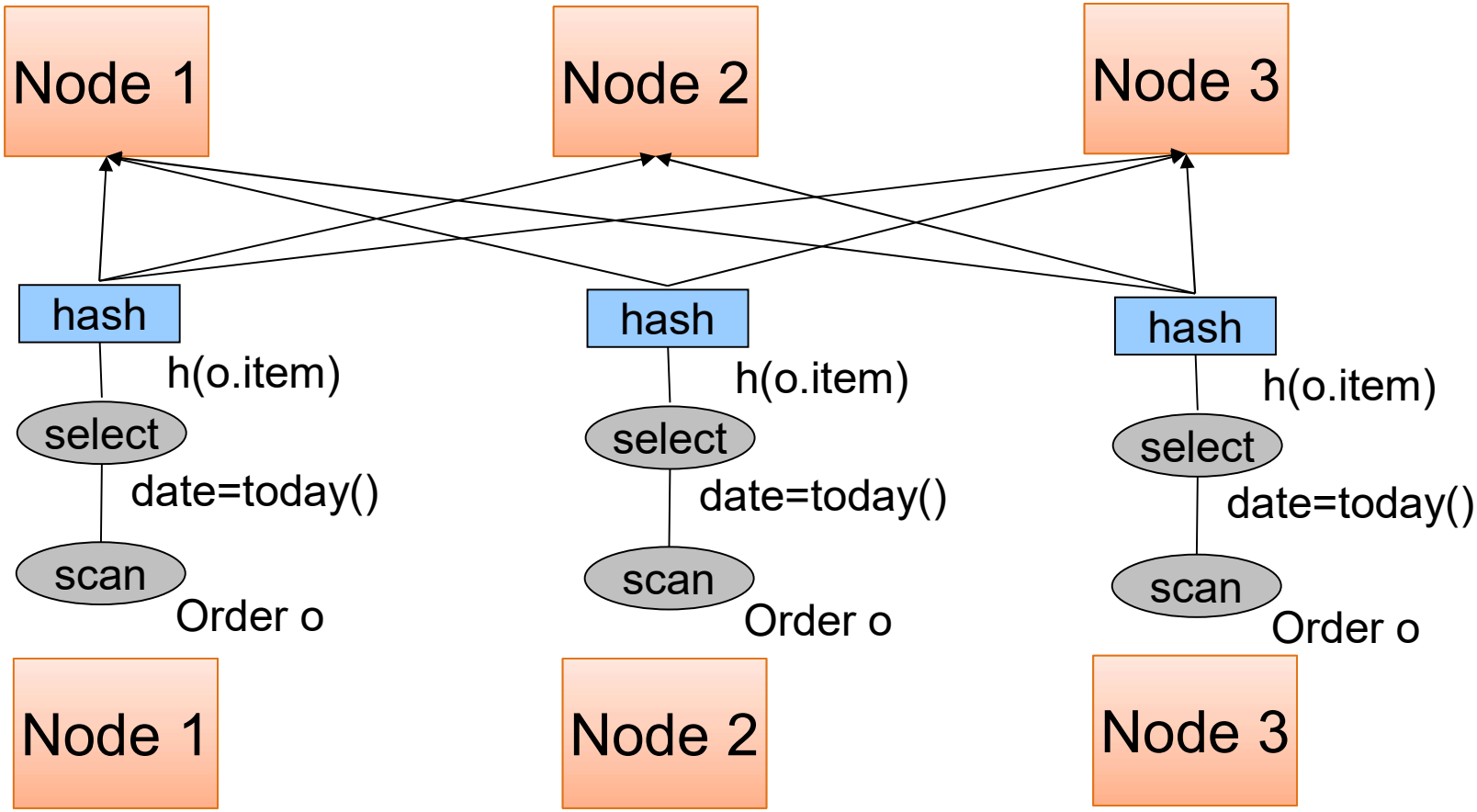
```
SELECT *
FROM Order o, Line i
WHERE o.item = i.item
      AND o.date = today()
```



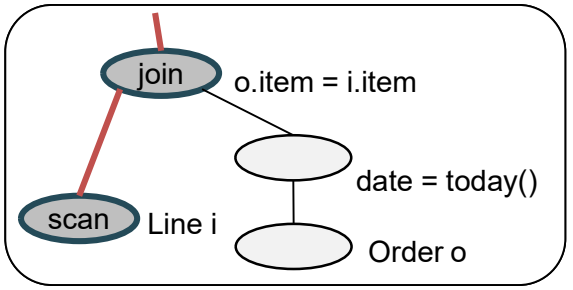
Order(oid, item, date), Line(item, …)



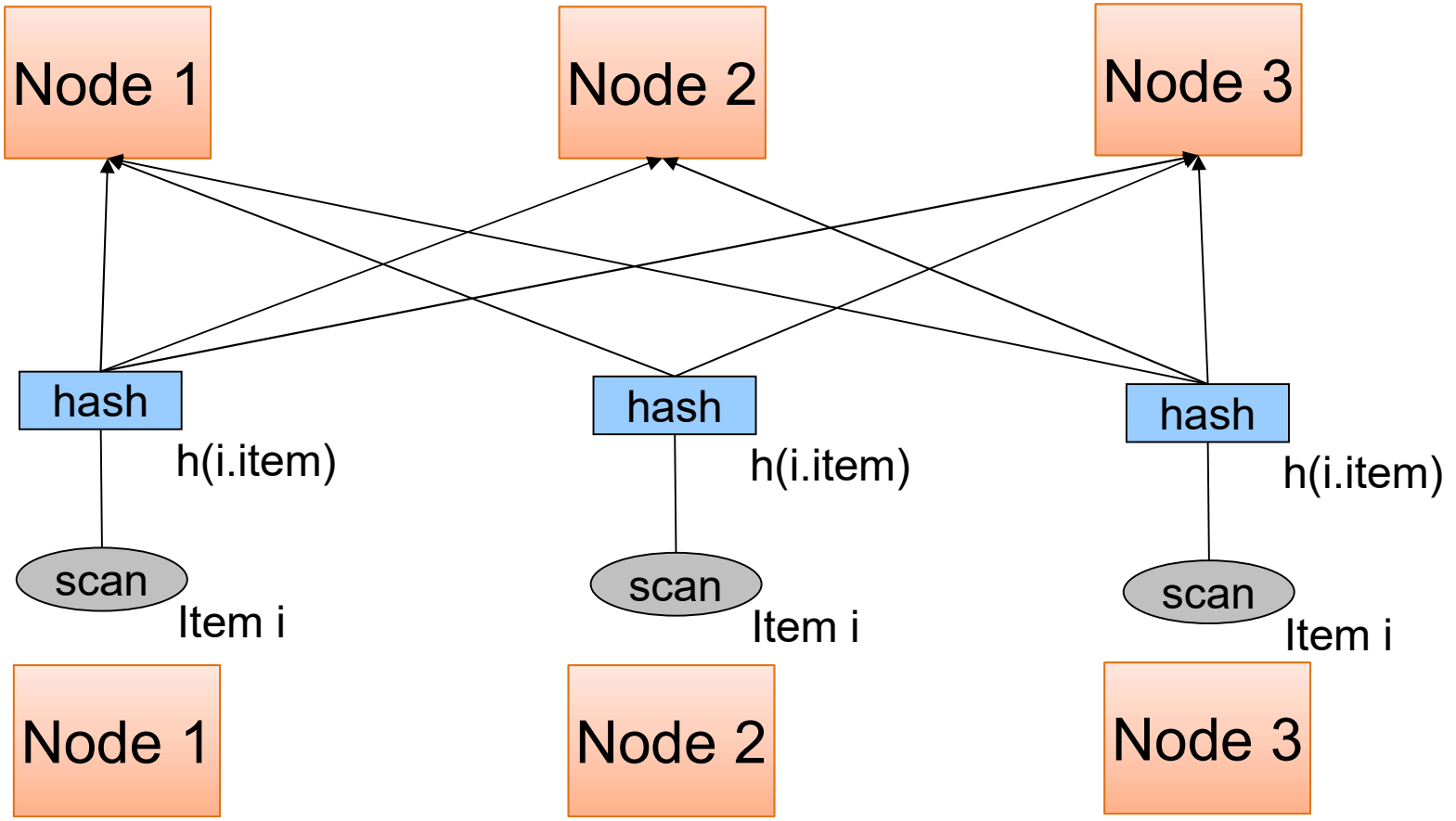
Query Execution



Order(oid, item, date), Line(item, …)

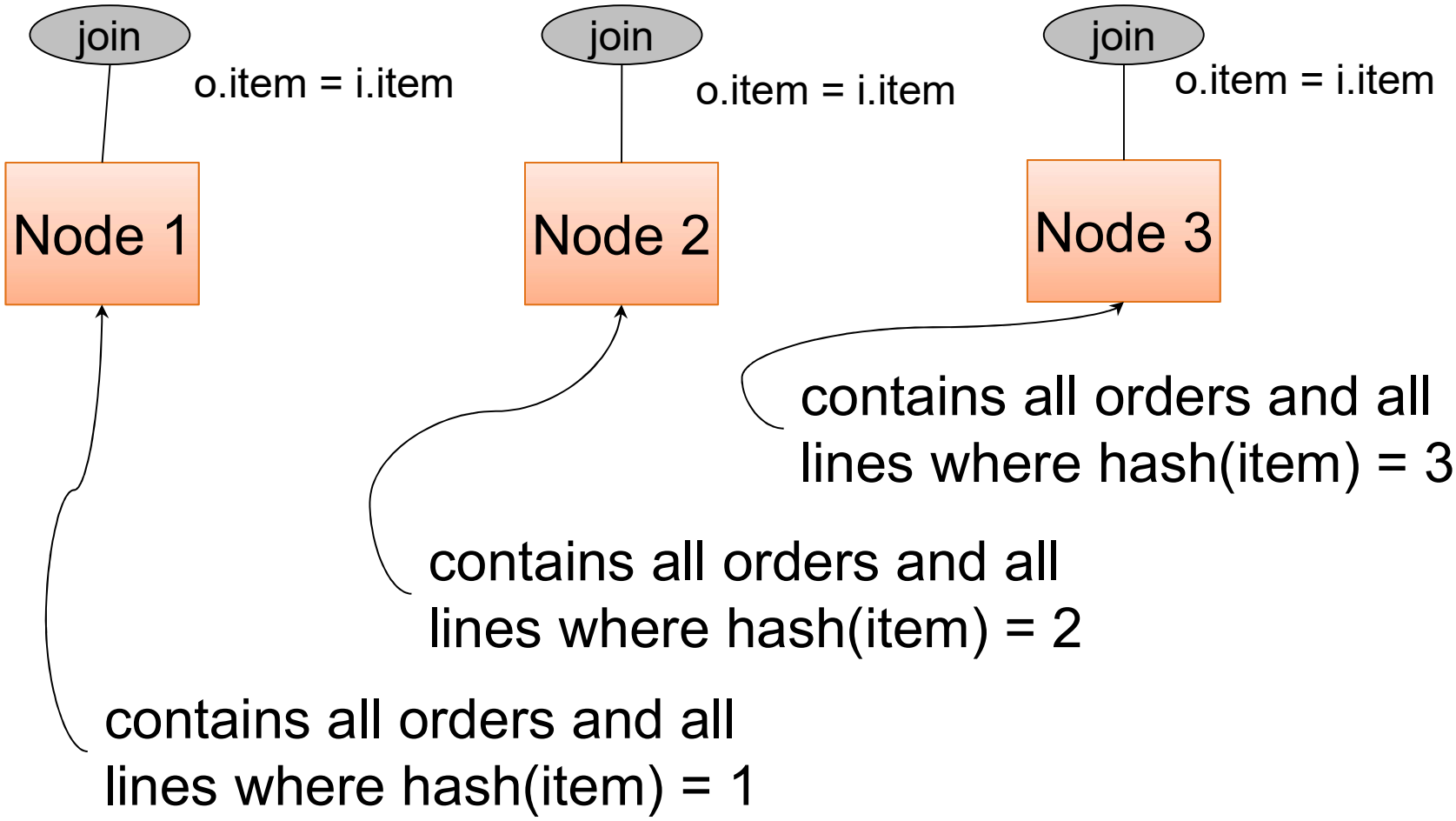


Query Executi



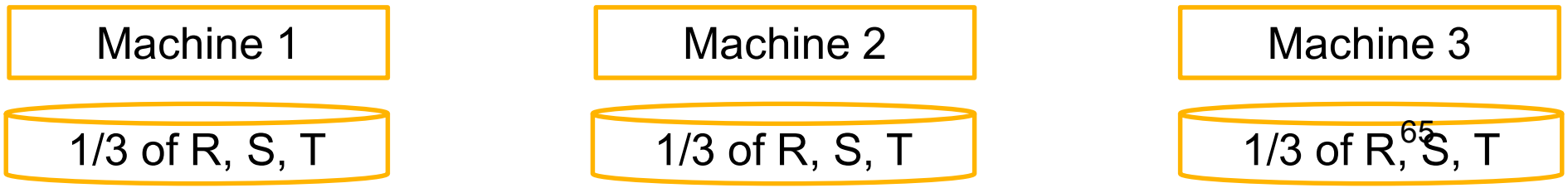
Order(oid, item, date), Line(item, …)

Query Execution

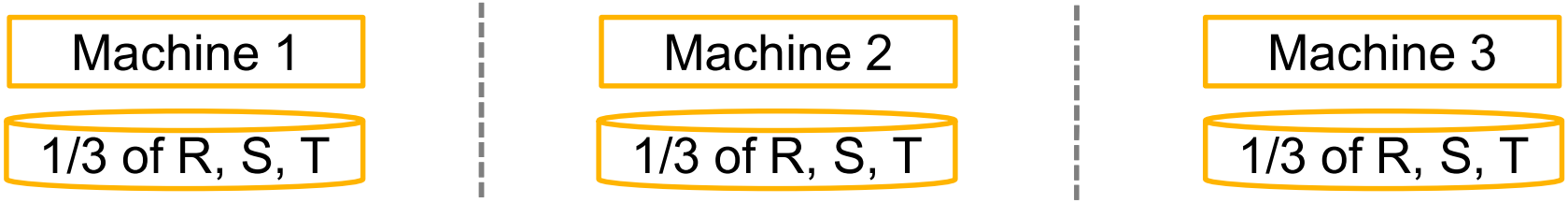


Example 2

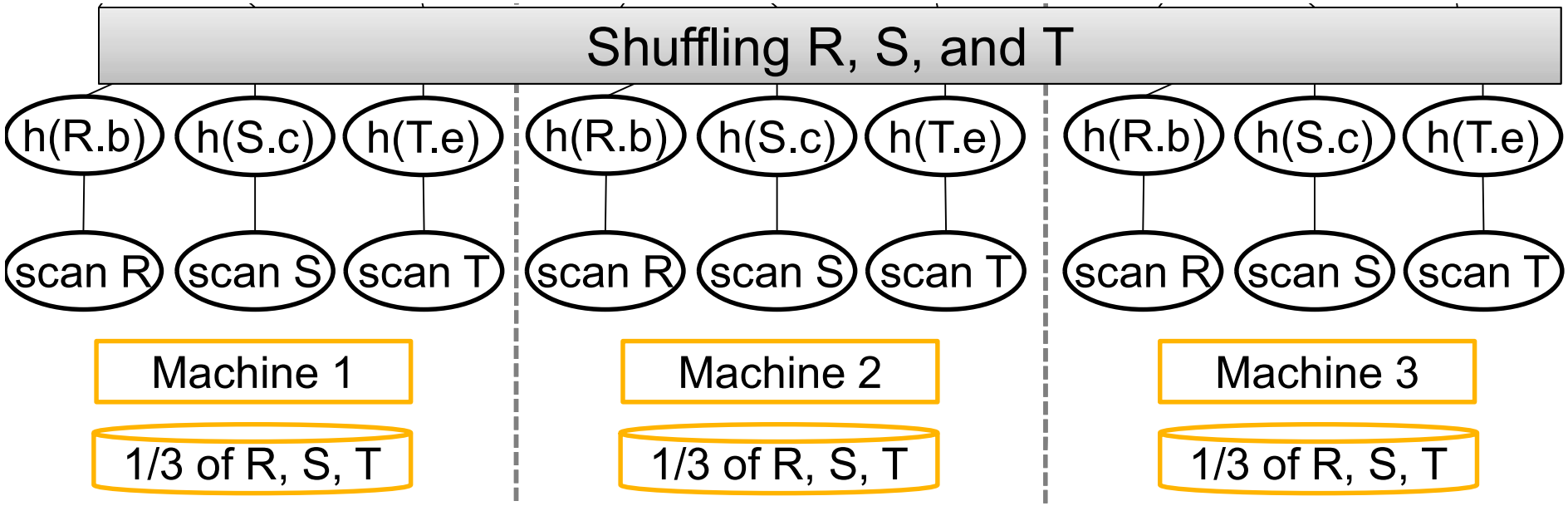
```
SELECT *
FROM R, S, T
WHERE R.b = S.c AND S.d = T.e AND (R.a - T.f) > 100
```



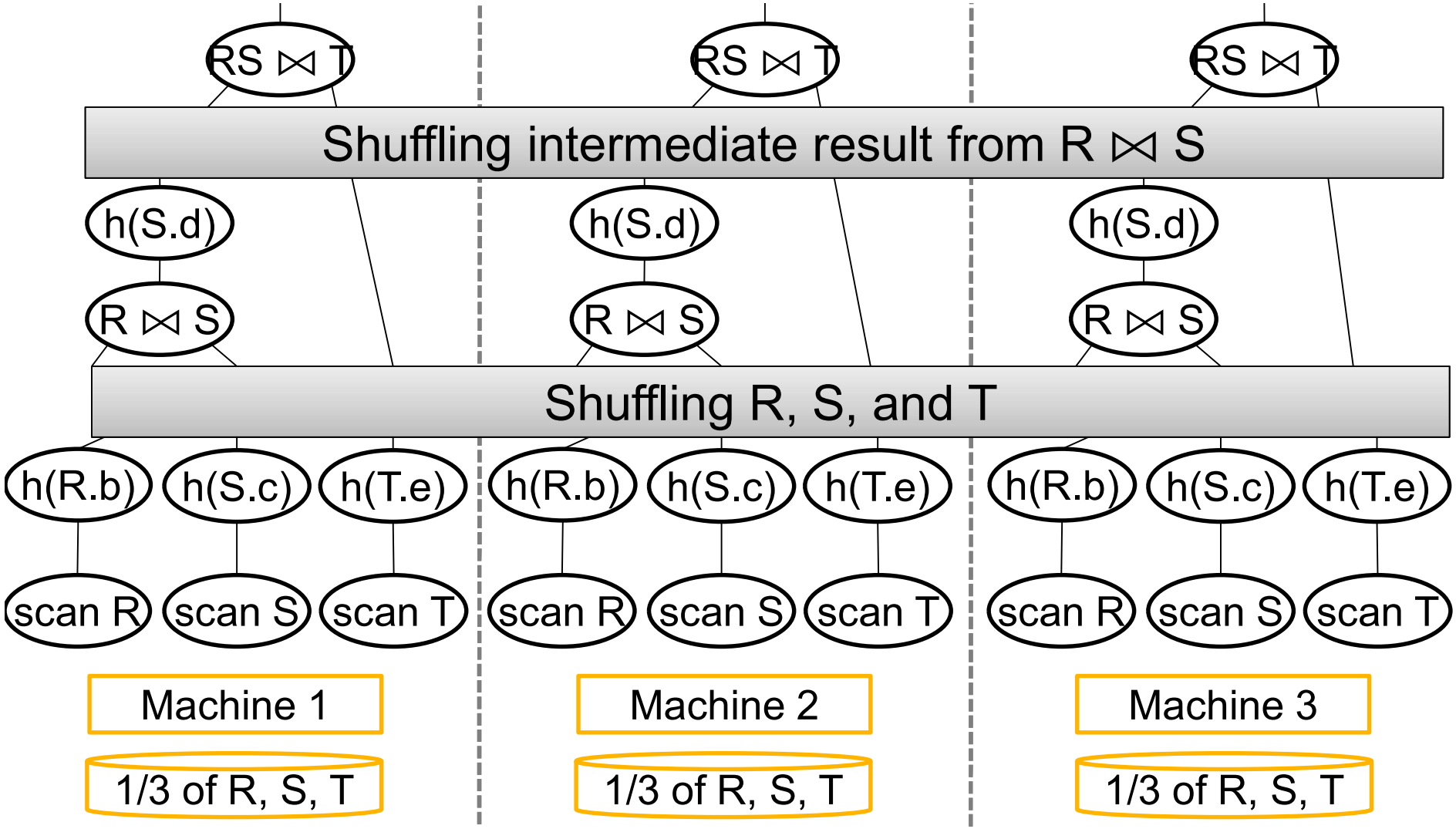
… WHERE R.b = S.c AND S.d = T.e AND (R.a - T.f) > 100



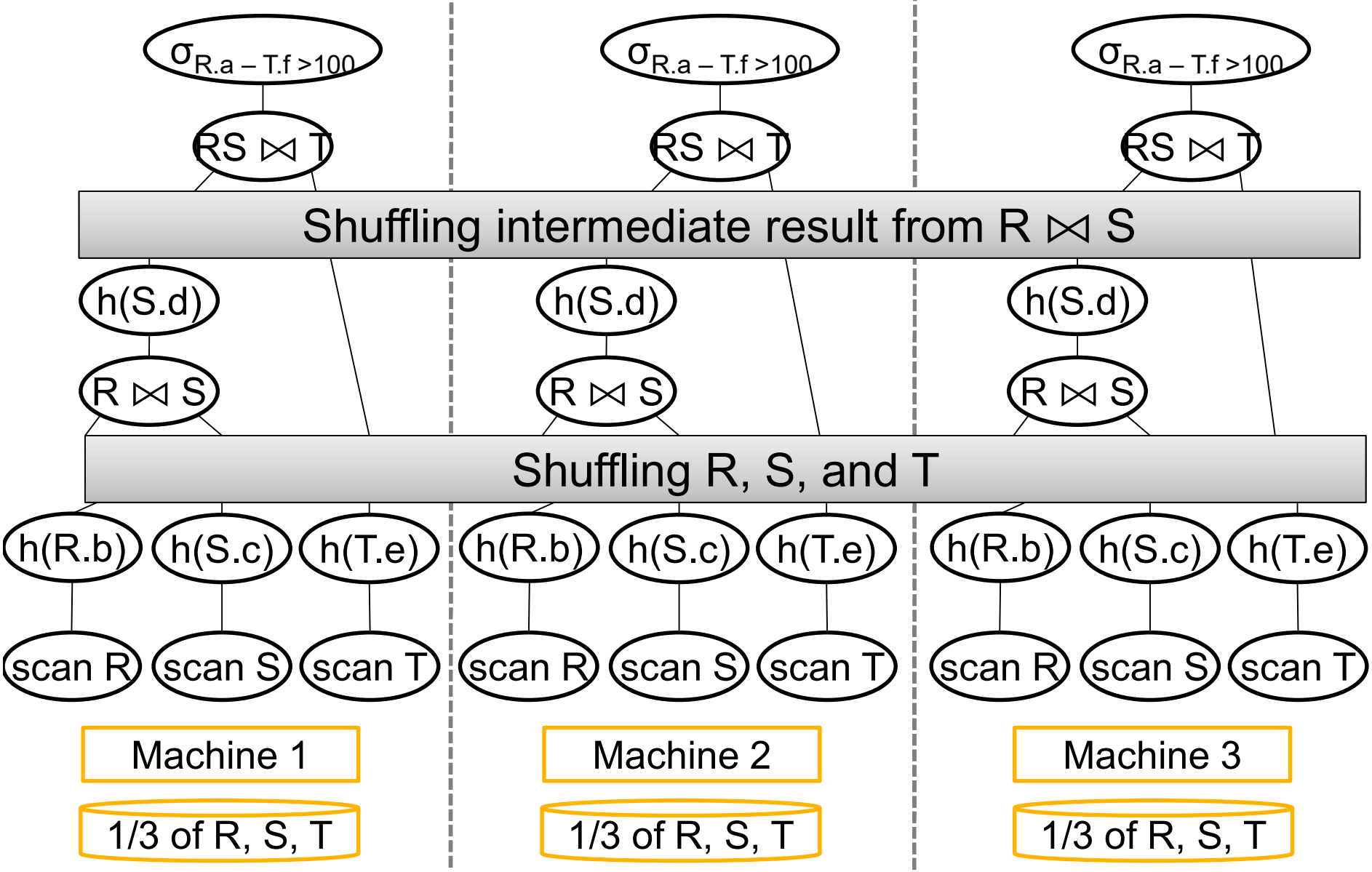
… WHERE R.b = S.c AND S.d = T.e AND (R.a - T.f) > 100



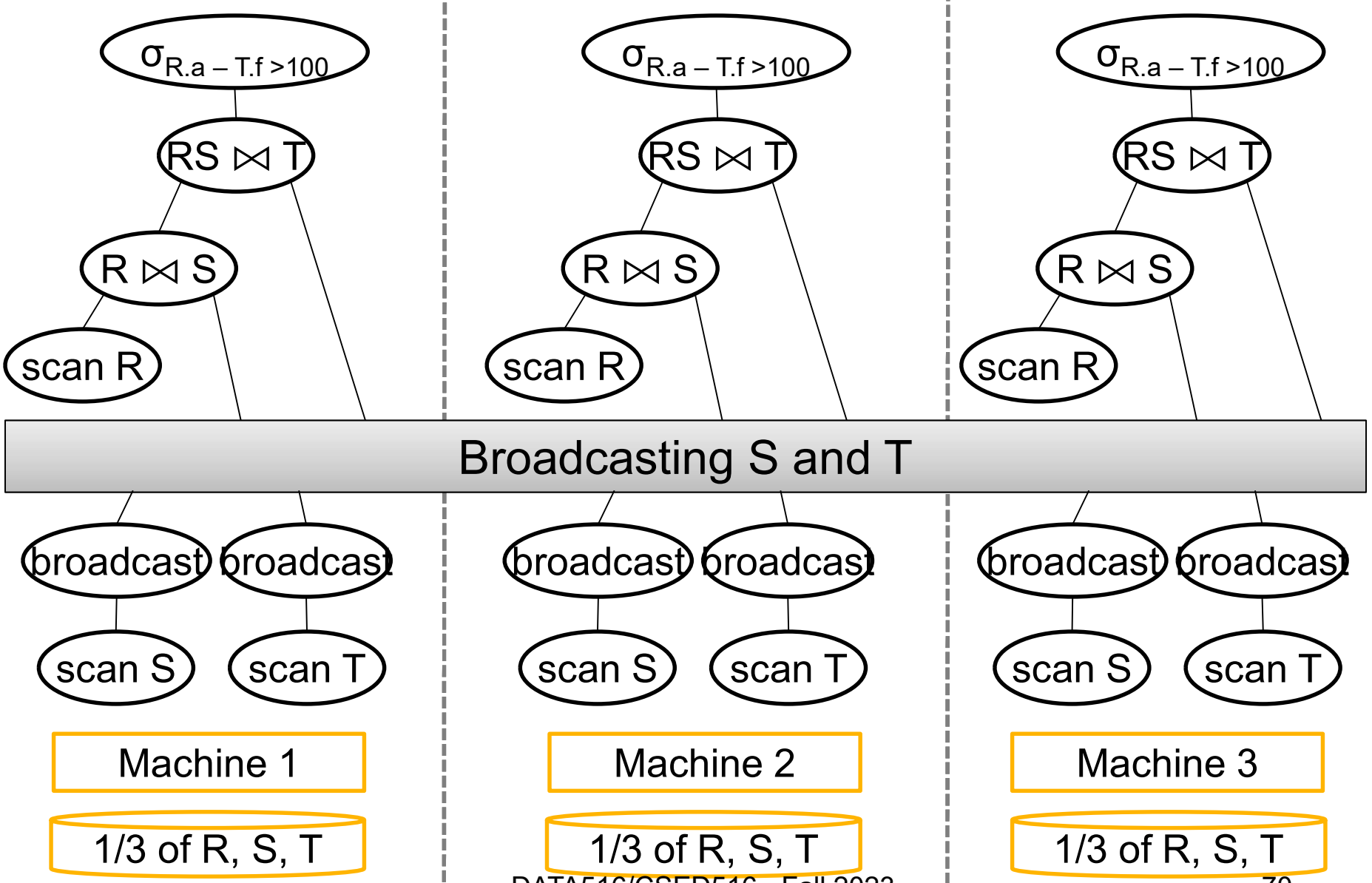
… WHERE R.b = S.c AND S.d = T.e AND (R.a - T.f) > 100



… 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



Discussion

- Hash-join:
  - Both relations are partitioned (good)
  - May have skew (bad)

Discussion

- Hash-join:
  - Both relations are partitioned (good)
  - May have skew (bad)
- Broadcast join
  - One relation must be broadcast (bad)
  - No worry about skew (good)

Discussion

- 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

Outline

- Distributed Joins
- Skew
- Parallel Query Processing Wrap-up
- Graph

Skew

Skew

- Skew means that one server runs much longer than the other servers

- Reasons:
  - Computation skew
  - Data skew

Computation Skew

- All workers receive the same amount of input data, but some need to run much longer than others
- E.g. perform some image processing whose runtimes depends on the image
- Solution: use virtual servers

Virtual Servers

Main idea:

- If we send the data uniformly to the P servers, and one of them is stuck with the complicated image, then we have skew
- Solution: Pretend we have many "virtual" servers

Virtual Servers

Large number P_v of "virtual servers"

- Design algorithm for P_v virtual servers
- Scale down to $P << P_v$ physical servers, by simulating them round-robin

E.g. MapReduce: P=workers, P_v=map tasks

Data Skew

- We fail to distribute the data uniformly to the servers
- Question: why can this happen?

Data Skew

- We fail to distribute the data uniformly to the servers
- Question: why can this happen?
- Answer:
  - Range partition may have many more tuples in one bucket than another
  - Hash partition may suffer from heavy hitters

Data Skew

Assume we hash-partition data items

- All records with same partition key are sent to the same server
- Heavy hitter

Analyzing Heavy Hitters

- How many times can an item occur before it is called a “heavy hitter”?

- The answer requires a deep analysis of what a good hash function can do.

Problem Statement

Given: N distinct data items $v_1, \ldots, v_N$

- We hash-partition them to P nodes
- How uniform is the partition?

Discussion

There is no deterministic "good" hash function:

- For any hash function h,
  there exists distinct data items v_1, …, v_N
  s.t. all are mapped to the same value:
  $h(v_1)=h(v_2)=$ …$h(v_N)$

Discussion

There is no deterministic "good" hash function:

- For any hash function h,
  there exists distinct data items v_1, …, v_N
  s.t. all are mapped to the same value:
  $h(v_1)=h(v_2)= \ldots h(v_N)$

A "good" hash function is random function:

- Intuition: every day we choose another h, we want the partition to be uniform in *expectation*

Discussion

- Many distributed query processors do not handle data skew well
- (Project idea: how does your favorite engine handle skewed data?)
- In practice, you may need to partition skewed data manually

Outline

- Distributed Joins
- Skew
- Parallel Query Processing Wrap-up
- Graph

Parallel Query Processing Wrap-up

Recap: Parallel Architectures:

1.

2.

3.

Recap: Parallel Architectures:

1. Shared Memory
2. Shared Disk
3. Shared Nothing – aka distributed

Recap: Motivation

- Discuss when to use distributed data processing v.s. single server

Recap: Explain these terms

- Speedup v.s. Scaleup

Recap: Horizontal Data Partitioning

Describe three strategies:

1.

2.

3.

Recap: Horizontal Data Partitioning

Describe three strategies:

1. Block partition

2. Hash partition

3. Range partition

Recap: Distributed Join

Describe/discuss these algorithms:

1. Parallel Hash Join

2. Broadcast join, a.k.a. small join

Conclusion

- Distributed data processing:
  - Spread the data to fit in main memory
  - Take advantage of parallelism
- "SQL is embarrassingly parallel"
  - Relational algebra: easy to parallelize
  - Hash-based algorithm suffer from skew

Outline

- Distributed Joins
- Skew
- Parallel Query Processing Wrap-up
- Graph

Graphs

Graph Processing Motivation

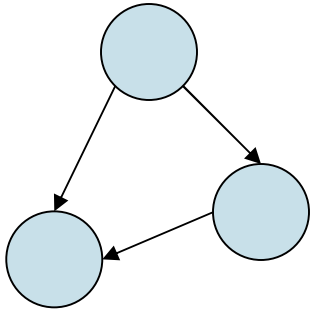
- Many apps need to do analytics on graphs
  - Web graph
  - Social networks
  - Transportation routes
  - Citation graphs
  - Disease propagation graphs
  - …
- A graph: G(V,E)
  - V: Vertices in the graph
  - E: Edges between the vertices
  - Large graph means many edges, not many gigabytes

Graph Analysis

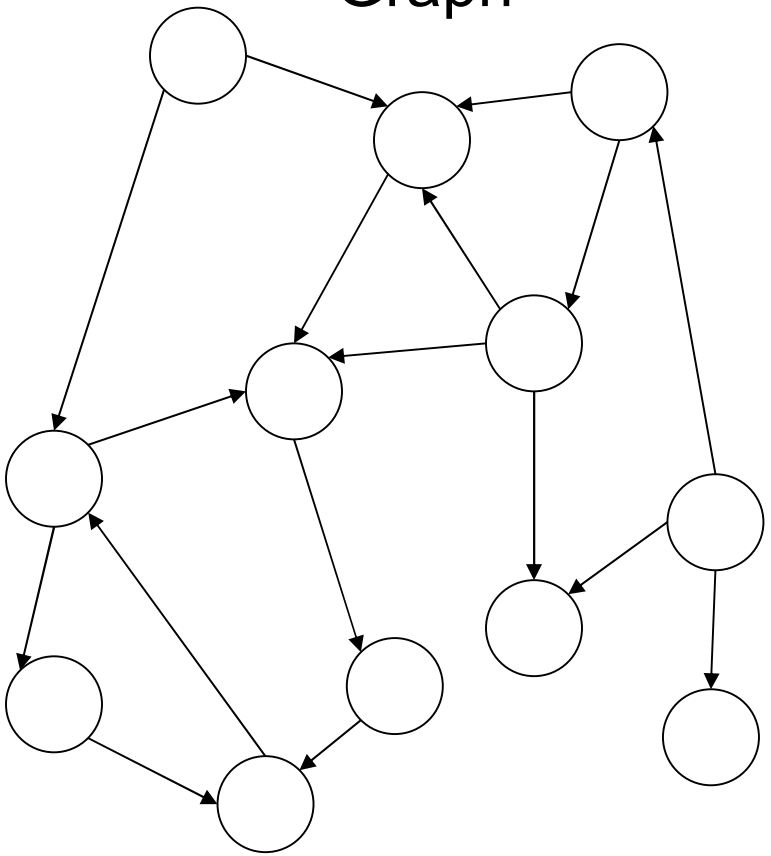
- Graph analytics has several unique properties
  - One large object: the graph
  - Difficult to partition and process in parallel
  - Iterative processing
    - Little work per vertex at each iteration
    - Many iterations & significant amount of communication
- Example applications
  - Shortest path
  - Clustering
  - Page rank and variants
  - Triangles and other structure
  - …

Example 1: Pattern Matching

Pattern

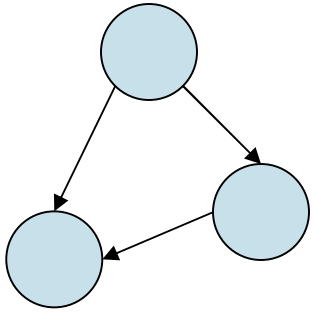


Graph

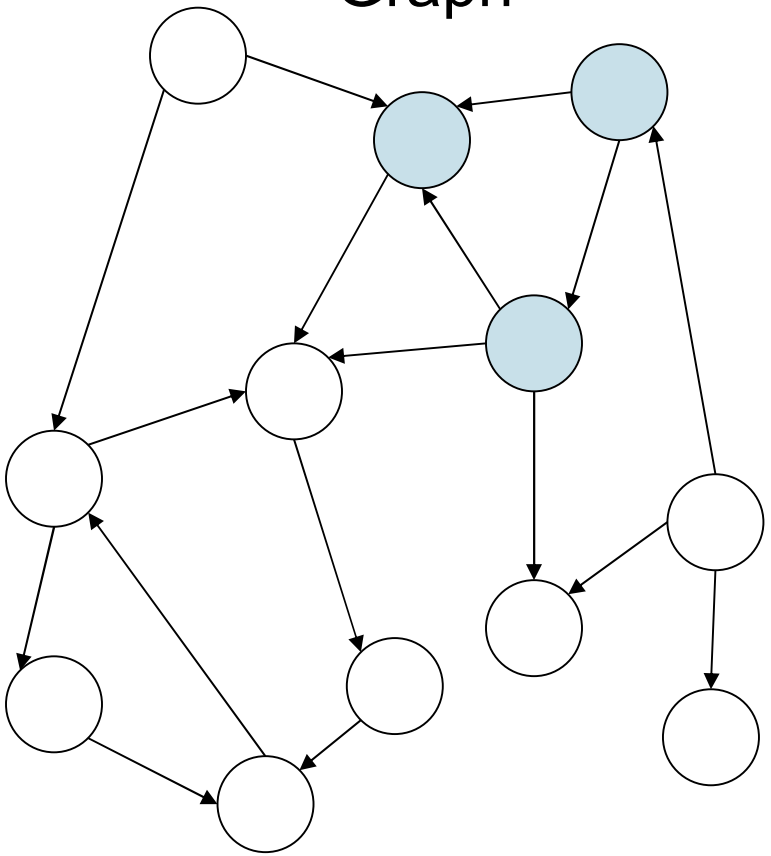


Example 1: Pattern Matching

Pattern

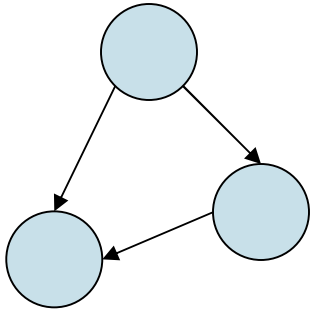


Graph

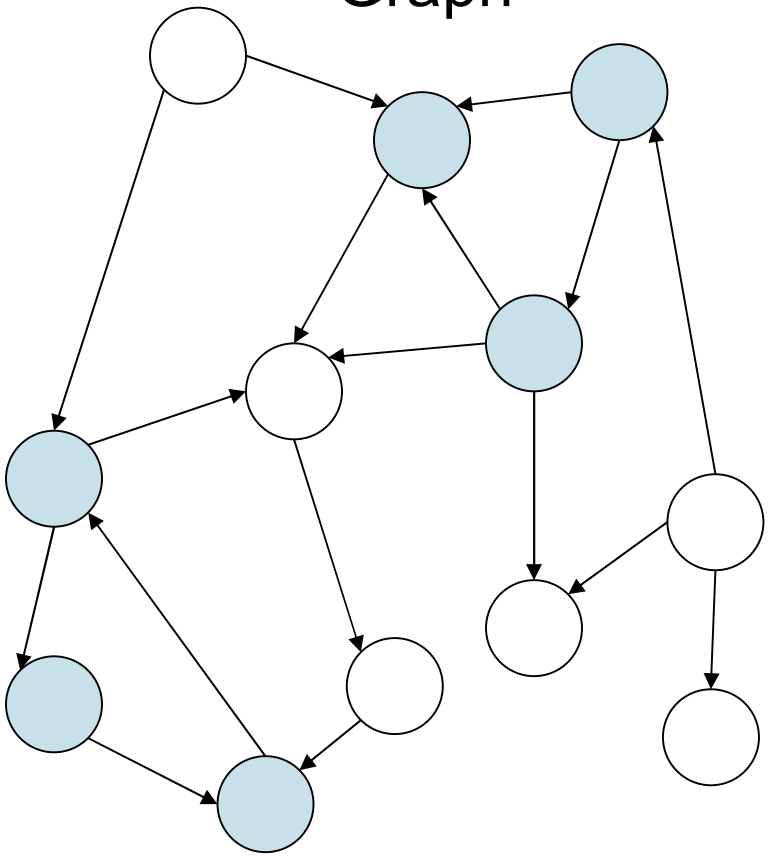


Example 1: Pattern Matching

Pattern

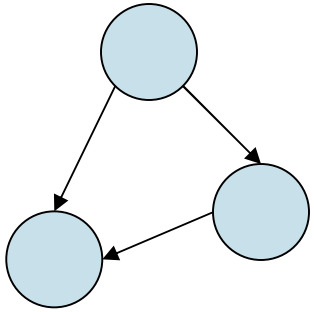


Graph

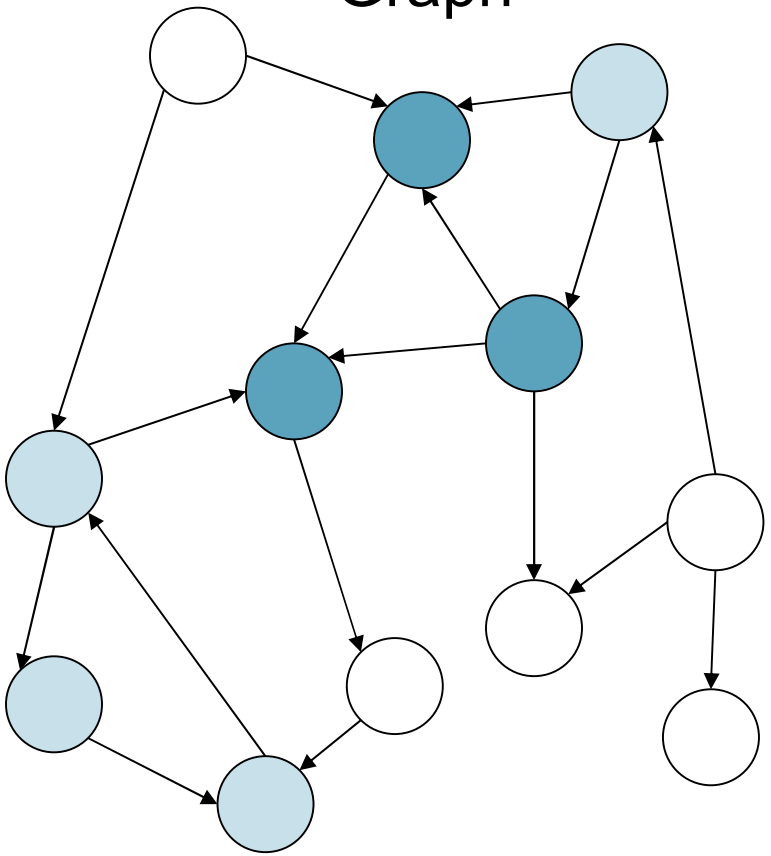


Example 1: Pattern Matching

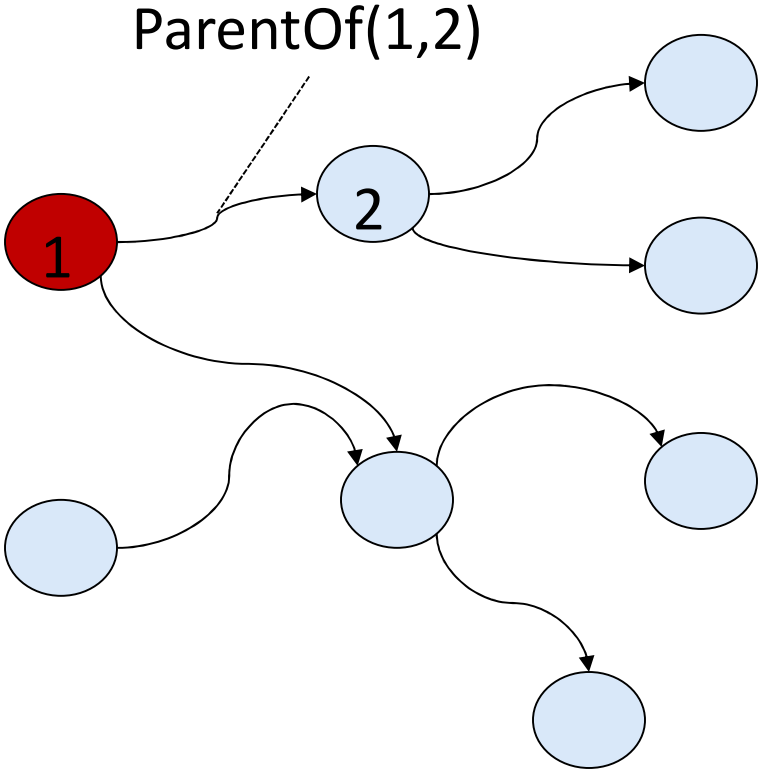
Pattern



Graph



Example 2: Descendants



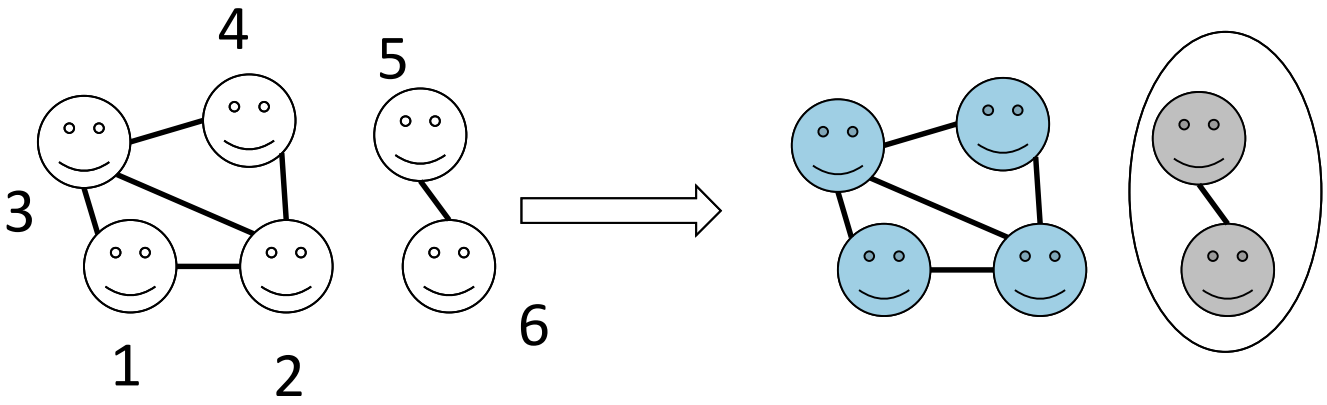
Find all descendants

of the red node

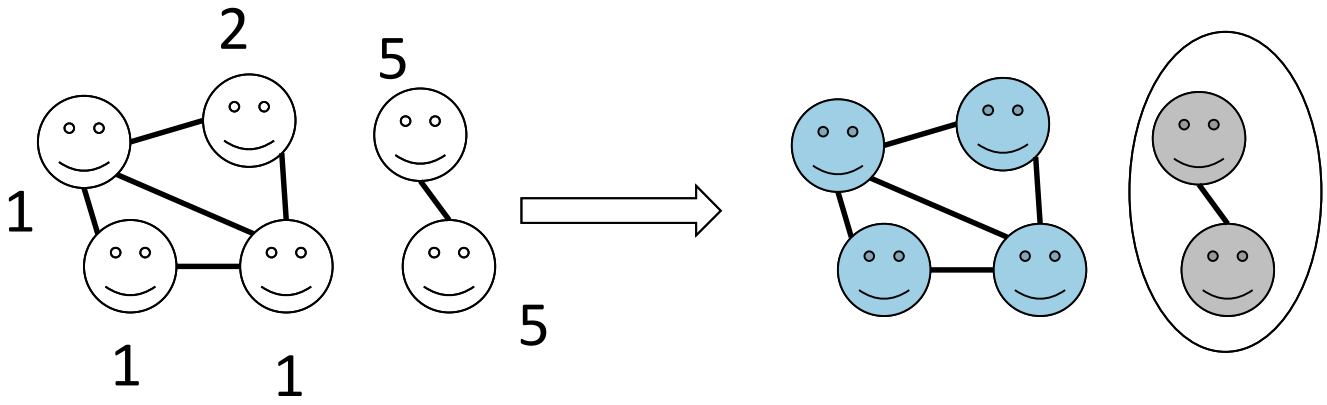
Recursively follow the ParentOf links

until no new descendants are found

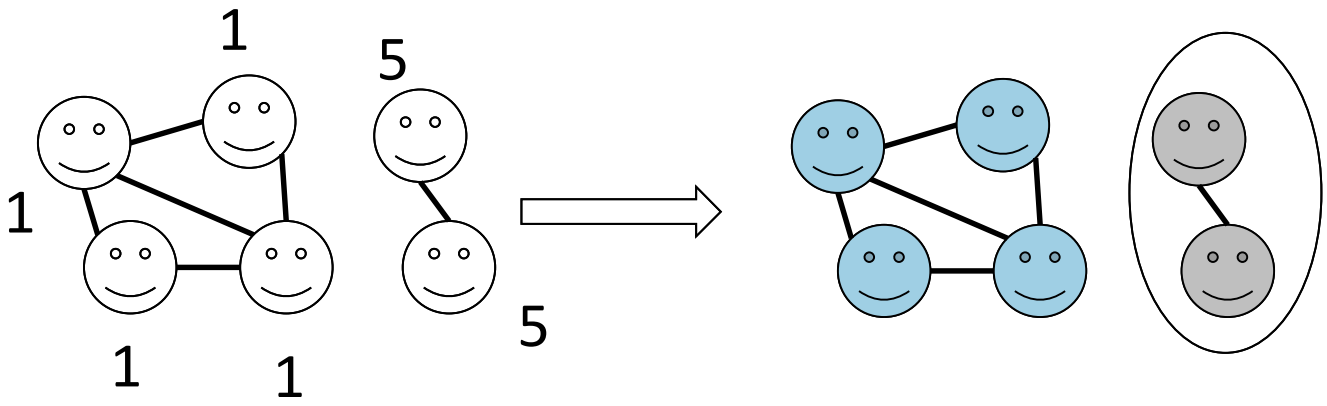
Example 3: Connected Components



Example 3: Connected Components

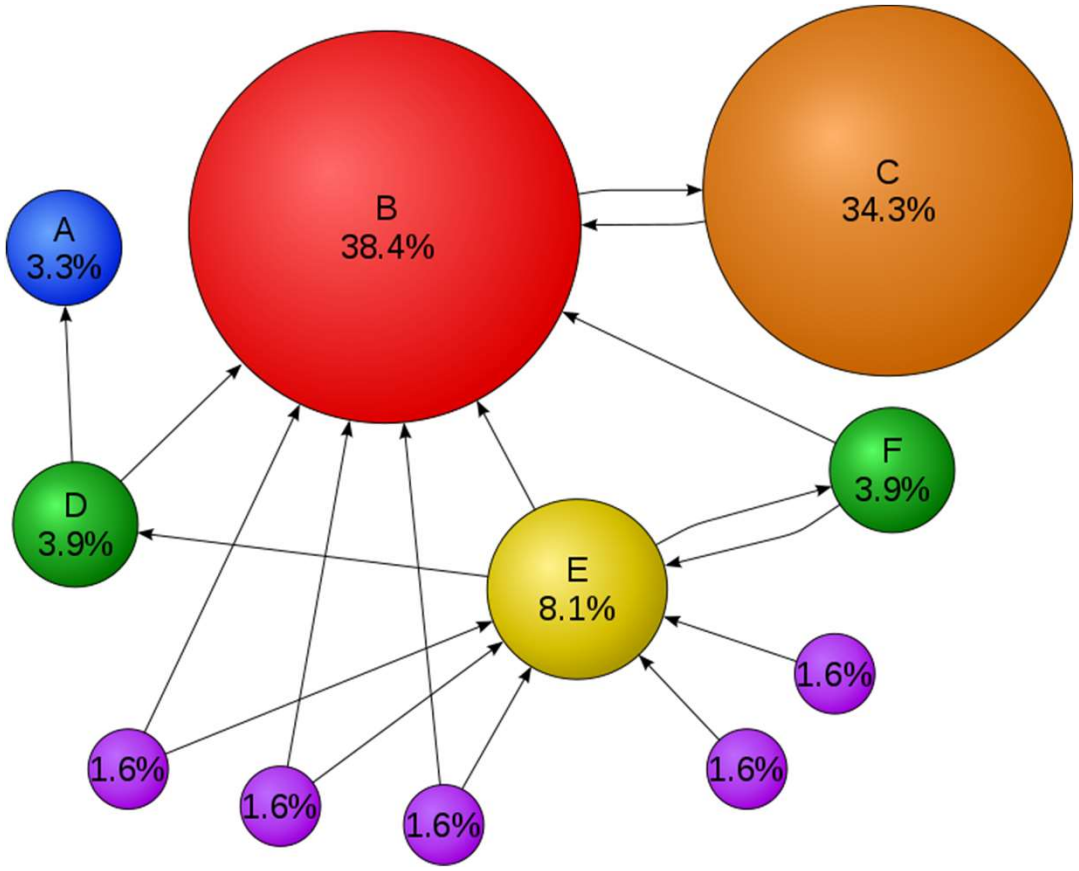


Example 3: Connected Components



Example 4: PageRank

The PageRank algorithm outputs a probability distribution used to represent the likelihood that a person randomly clicking on links will arrive at any particular page.



Iterate until convergence

$$PR(p_i,t+1) = (1-d)/N + d \sum_{pj \in M(pi)} PR(p_j,t)/L(p_j)$$

Where

- PR(x): Page rank of x
- L(x): # Outgoing links from x
- M(x): Incoming pages to x

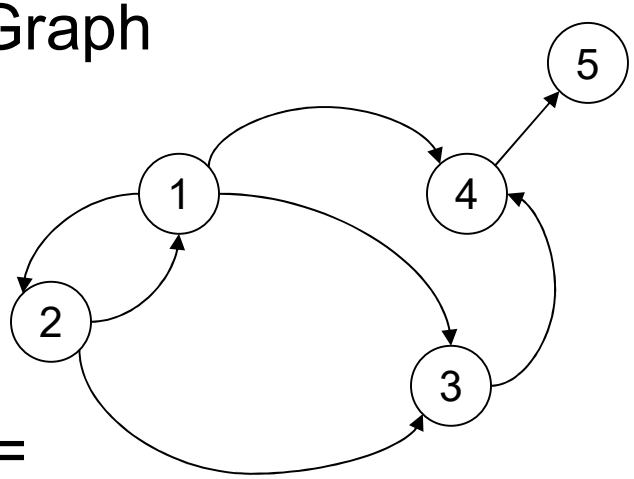
From Wikipedia: https://en.wikipedia.org/wiki/PageRank

How to Model Graph Analytics

- Option 1: Relational Model
  - Relation Edges(v1,v2)
  - Optionally can also have a relation Vertices(v)
  - Relational queries
- Option 2: Graph Model
  - The graph is a first-class citizen
  - Vertex-based API
  - Pattern-based and/or traversal-based queries
- Option 3: Linear Algebra
  - Graph as a matrix

Processing Graphs in SQL

Graph

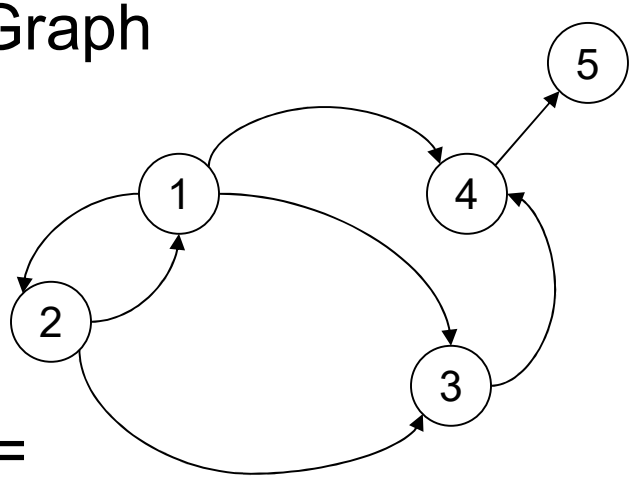


R=

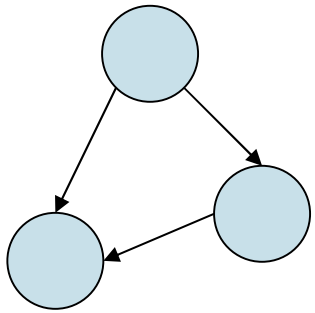
src	dst
1	2
2	1
2	3
1	4
3	4
4	5
1	3

Processing Graphs in SQL

Graph



Pattern Matching

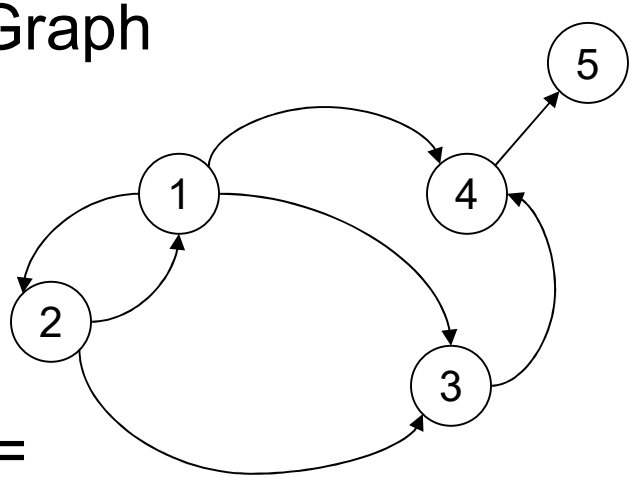


R=

src	dst
1	2
2	1
2	3
1	4
3	4
4	5
1	3

Processing Graphs in SQL

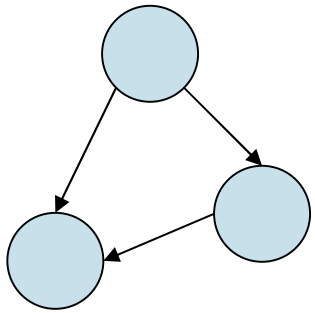
Graph



R=

src	dst
1	2
2	1
2	3
1	4
3	4
4	5
1	3

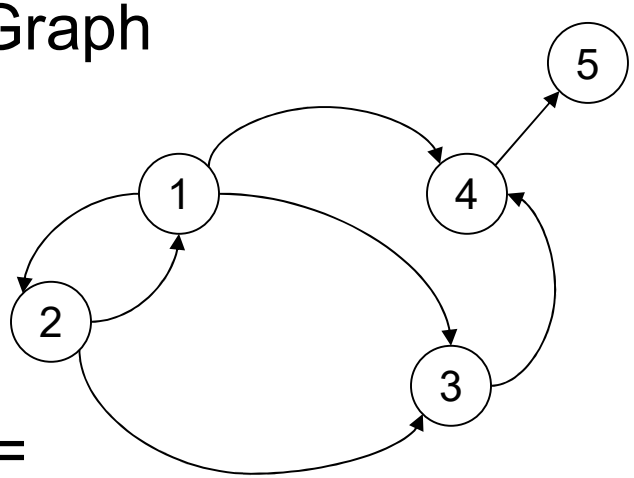
Pattern Matching



```
SELECT …
FROM …
WHERE …
```

Processing Graphs in SQL

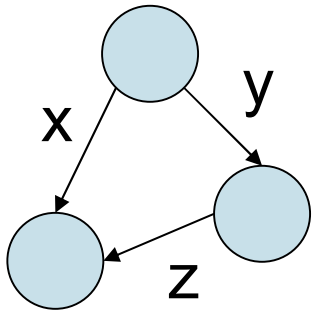
Graph



R=

src	dst
1	2
2	1
2	3
1	4
3	4
4	5
1	3

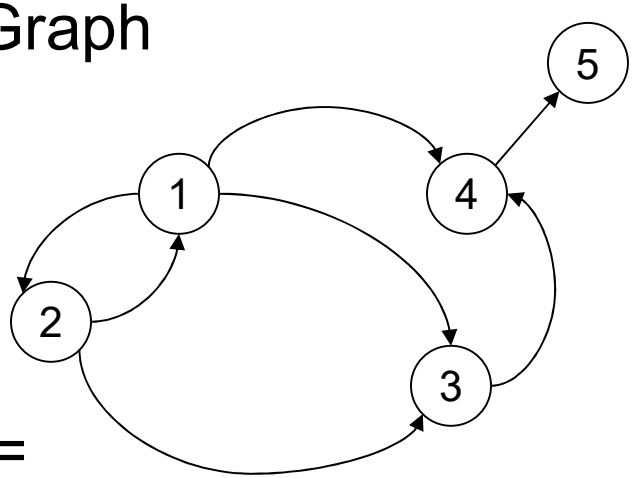
Pattern Matching




```
SELECT …
FROM …
WHERE …
```

Processing Graphs in SQL

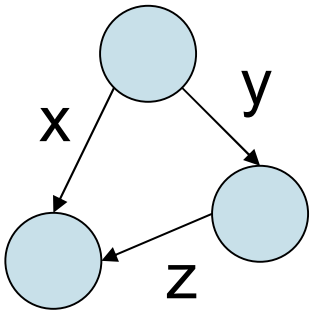
Graph



R=

src	dst
1	2
2	1
2	3
1	4
3	4
4	5
1	3

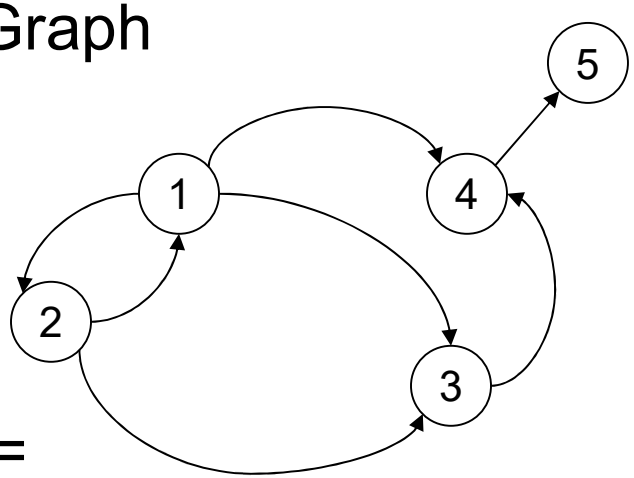
Pattern Matching




```
SELECT
FROM R x, R y, R z
WHERE
```

Processing Graphs in SQL

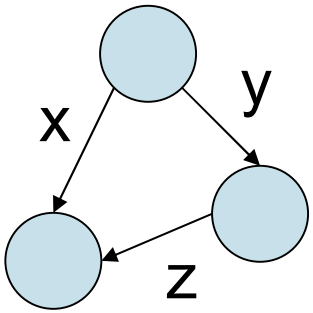
Graph



R=

src	dst
1	2
2	1
2	3
1	4
3	4
4	5
1	3

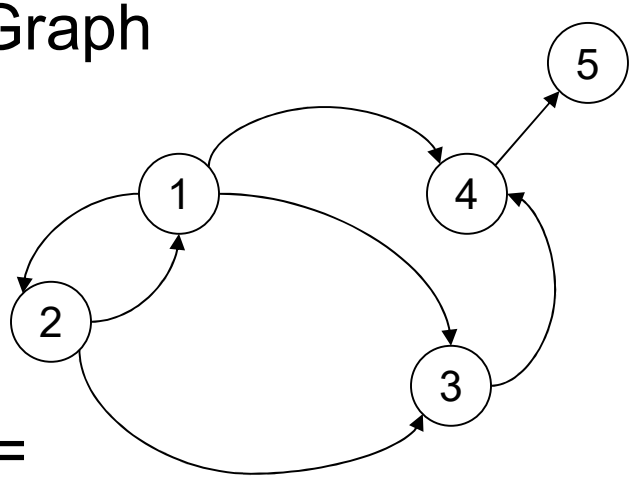
Pattern Matching




```
SELECT x.src, y.dst, z.dst
FROM R x, R y, R z
WHERE
```

Processing Graphs in SQL

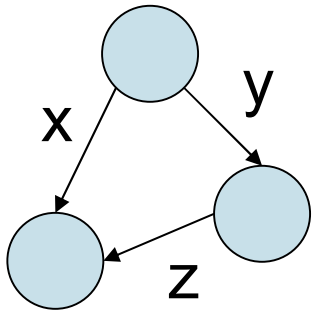
Graph



R=

src	dst
1	2
2	1
2	3
1	4
3	4
4	5
1	3

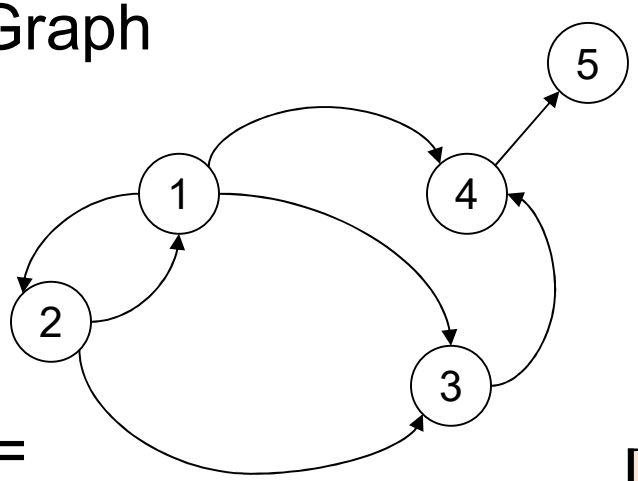
Pattern Matching



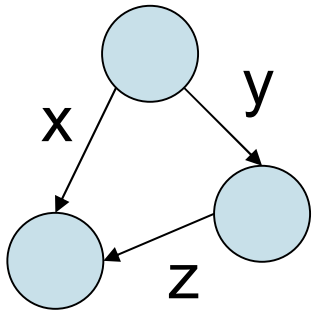

```
SELECT x.src, y.dst, z.dst
FROM R x, R y, R z
WHERE x.src = y.src
      and x.dst = z.dst
      and y.dst = z.src
```

Processing Graphs in SQL

Graph



Pattern Matching



R=

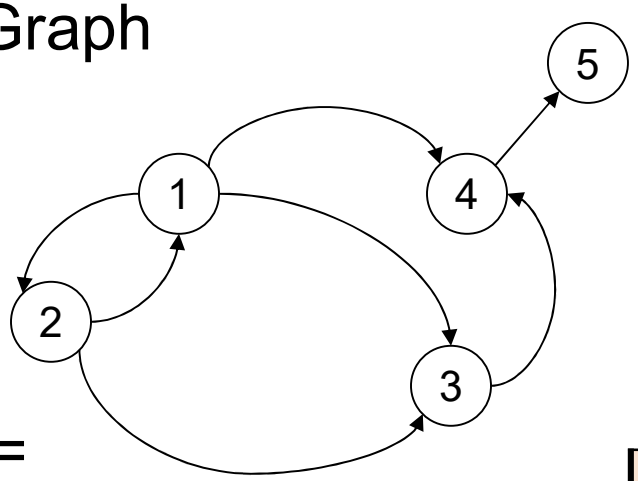
src	dst
1	2
2	1
2	3
1	4
3	4
4	5
1	3

x.src	y.dst	z.dst
1	2	3
1	3	4
2	1	3

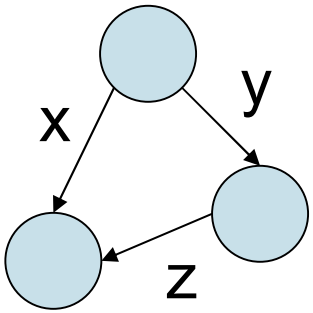
```
SELECT x.src, y.dst, z.dst
FROM R x, R y, R z
WHERE x.src = y.src
      and x.dst = z.dst
      and y.dst = z.src
```

Processing Graphs in SQL

Graph



Pattern Matching



R=

src	dst
1	2
2	1
2	3
1	4
3	4
4	5
1	3

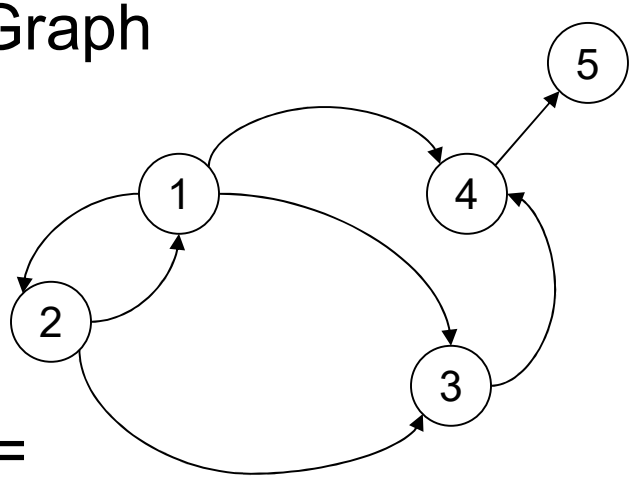
x.src	y.dst	z.dst
1	2	3
1	3	4
2	1	3

```
SELECT x.src, y.dst, z.dst
FROM R x, R y, R z
WHERE x.src = y.src
      and x.dst = z.dst
      and y.dst = z.src
```

A pattern with n edges becomes an n-way selfjoin

Processing Graphs in SQL

Graph



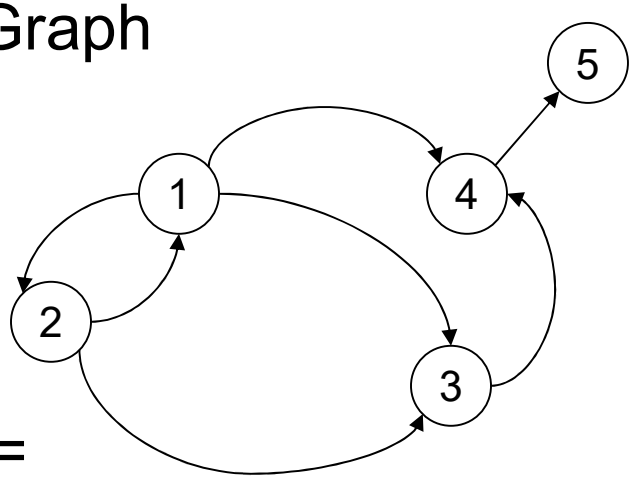
Find Descendants of node 2

R=

src	dst
1	2
2	1
2	3
1	4
3	4
4	5
1	3

Processing Graphs in SQL

Graph



R=

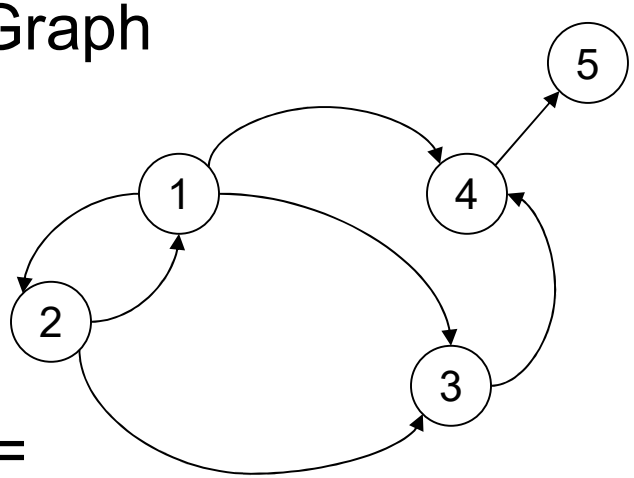
src	dst
1	2
2	1
2	3
1	4
3	4
4	5
1	3

Find Descendants of node 2

Find children:

Processing Graphs in SQL

Graph



R=

src	dst
1	2
2	1
2	3
1	4
3	4
4	5
1	3

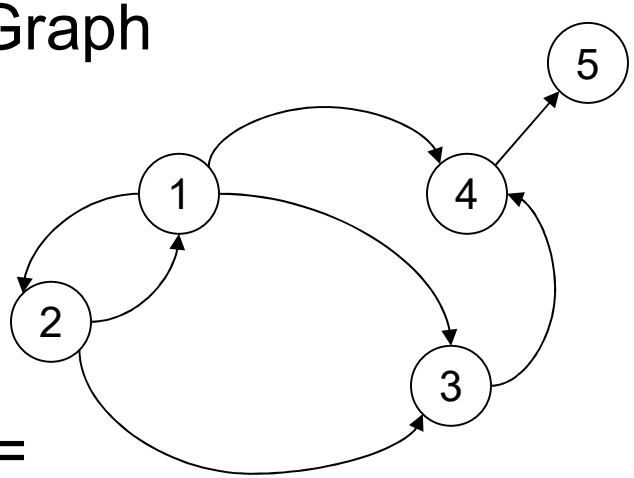
Find Descendants of node 2

Find children:

```
SELECT x.dst as d
FROM R x
WHERE x.src = 2
```

Processing Graphs in SQL

Graph



R=

src	dst
1	2
2	1
2	3
1	4
3	4
4	5
1	3

d
1
3

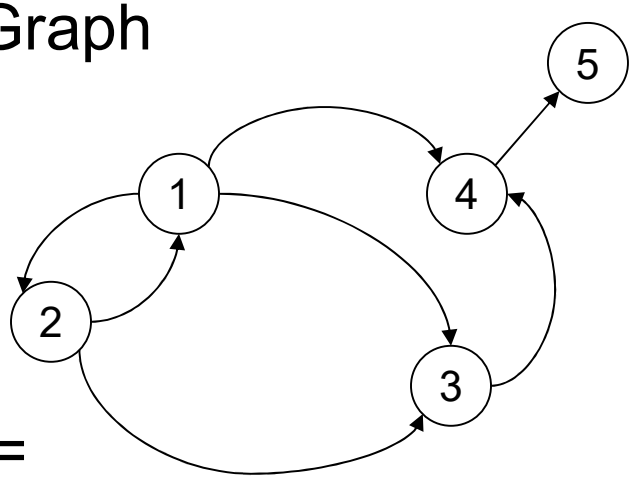
Find Descendants of node 2

Find children:

```
SELECT x.dst as d
FROM R x
WHERE x.src = 2
```

Processing Graphs in SQL

Graph



R=

src	dst
1	2
2	1
2	3
1	4
3	4
4	5
1	3

Find Descendants of node 2

Find children:

d
1
3

```
SELECT x.dst as d
FROM R x
WHERE x.src = 2
```

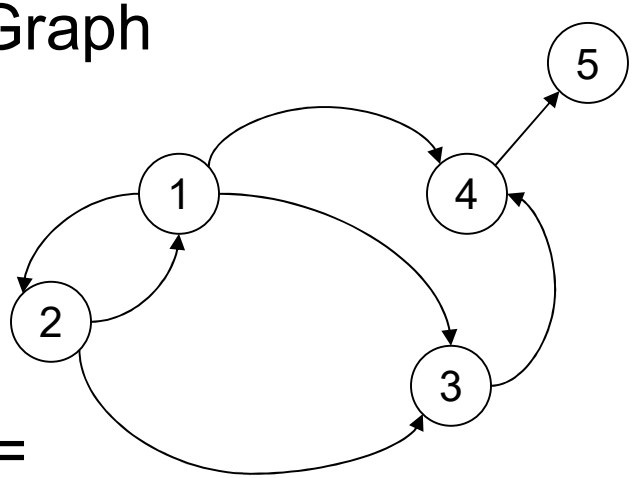
…and their children

2
4

```
UNION
SELECT DISTINCT y.dst as d
FROM R x, R y
WHERE x.src = 2 and x.dst = y.srt
```

Processing Graphs in SQL

Graph



R=

src	dst
1	2
2	1
2	3
1	4
3	4
4	5
1	3

Find Descendants of node 2

Find children:

d
1
3

```
SELECT x.dst as d
FROM R x
WHERE x.src = 2
```

…and their children

2
4

```
UNION
SELECT DISTINCT y.dst as d
FROM R x, R y
WHERE x.src = 2 and x.dst = y.srt
```

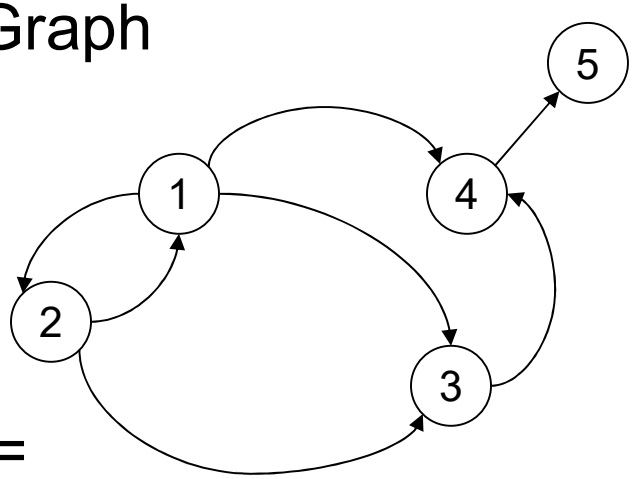
5

…

…and their children…

Processing Graphs in SQL

Graph



R=

src	dst
1	2
2	1
2	3
1	4
3	4
4	5
1	3

Cannot compute in SQL

d
1
3

2
4

5

…

Find Descendants of node 2

Find children:

```
SELECT x.dst as d
FROM R x
WHERE x.src = 2
```

…and their children

```
UNION
SELECT DISTINCT y.dst as d
FROM R x, R y
WHERE x.src = 2 and x.dst = y.srt
```

…and their children…

Discussion

- Graph processing often requires recursion:
  - Descendants, connected components, etc
- SQL *does* support recursion using WITH and CTE (Common Table Expression)
  - Lots of restrictions
- Origin of recursion in SQL: datalog

Datalog

- Designed in the 80's: simple, concise, elegant, very popular in research
- All techniques for recursive relational queries were developed for datalog
- But: no standard, no reference implementation

Outline

- Datalog rules
- Recursion
- Semantics

Next time: extensions, semi-naïve algo.

Actor(id, fname, lname)
Casts(pid, mid)
Movie(id, name, year)

← Schema

Datalog: Facts and Rules

Actor(id, fname, lname)
Casts(pid, mid)
Movie(id, name, year)

Datalog: Facts and Rules

Facts = tuples in the database

Rules = queries

Actor(id, fname, lname)
Casts(pid, mid)
Movie(id, name, year)

Datalog: Facts and Rules

Facts = tuples in the database

Rules = queries

```
Actor(344759,‘Douglas’, ‘Fowley’).
Casts(344759, 29851).
Casts(355713, 29000).
Movie(7909, ‘A Night in Armour’, 1910).
Movie(29000, ‘Arizona’, 1940).
Movie(29445, ‘Ave Maria’, 1940).
```

Actor(id, fname, lname)
Casts(pid, mid)
Movie(id, name, year)

Datalog: Facts and Rules

Facts = tuples in the database

```
Actor(344759,‘Douglas’, ‘Fowley’).
Casts(344759, 29851).
Casts(355713, 29000).
Movie(7909, ‘A Night in Armour’, 1910).
Movie(29000, ‘Arizona’, 1940).
Movie(29445, ‘Ave Maria’, 1940).
```

Rules = queries

```
Q1(y) :-  Movie(x,y,z), z=‘1940’.
```

Actor(id, fname, lname)
Casts(pid, mid)
Movie(id, name, year)

Datalog: Facts and Rules

Facts = tuples in the database

Actor(344759,'Douglas', 'Fowley').
Casts(344759, 29851).
Casts(355713, 29000).
Movie(7909, 'A Night in Armour', 1910).
Movie(29000, 'Arizona', 1940).
Movie(29445, 'Ave Maria', 1940).

Rules = queries

Q1(y) :- Movie(x,y,z), z='1940'.

Find Movies made in 1940

Actor(id, fname, lname)
Casts(pid, mid)
Movie(id, name, year)

Datalog: Facts and Rules

Facts = tuples in the database

Actor(344759,'Douglas', 'Fowley').
Casts(344759, 29851).
Casts(355713, 29000).
Movie(7909, 'A Night in Armour', 1910).
Movie(29000, 'Arizona', 1940).
Movie(29445, 'Ave Maria', 1940).

Rules = queries

Q1(y) :- Movie(x,y,z), z='1940'.

Q2(f, l) :- Actor(z,f,l), Casts(z,x),
Movie(x,y,'1940').

Actor(id, fname, lname)
Casts(pid, mid)
Movie(id, name, year)

Datalog: Facts and Rules

Facts = tuples in the database

Actor(344759,'Douglas', 'Fowley').
Casts(344759, 29851).
Casts(355713, 29000).
Movie(7909, 'A Night in Armour', 1910).
Movie(29000, 'Arizona', 1940).
Movie(29445, 'Ave Maria', 1940).

Rules = queries

Q1(y) :- Movie(x,y,z), z='1940'.

Q2(f, l) :- Actor(z,f,l), Casts(z,x),
Movie(x,y,'1940').

Find Actors who acted in Movies made in 1940

Actor(id, fname, lname)
Casts(pid, mid)
Movie(id, name, year)

Datalog: Facts and Rules

Facts = tuples in the database

Actor(344759,'Douglas', 'Fowley').
Casts(344759, 29851).
Casts(355713, 29000).
Movie(7909, 'A Night in Armour', 1910).
Movie(29000, 'Arizona', 1940).
Movie(29445, 'Ave Maria', 1940).

Rules = queries

Q1(y) :- Movie(x,y,z), z='1940'.

Q2(f, l) :- Actor(z,f,l), Casts(z,x),
Movie(x,y,'1940').

Q3(f,l) :- Actor(z,f,l), Casts(z,x1), Movie(x1,y1,1910),
Casts(z,x2), Movie(x2,y2,1940)

Actor(id, fname, lname)
Casts(pid, mid)
Movie(id, name, year)

Datalog: Facts and Rules

Facts = tuples in the database

Actor(344759,'Douglas', 'Fowley').
Casts(344759, 29851).
Casts(355713, 29000).
Movie(7909, 'A Night in Armour', 1910).
Movie(29000, 'Arizona', 1940).
Movie(29445, 'Ave Maria', 1940).

Rules = queries

Q1(y) :- Movie(x,y,z), z='1940'.

Q2(f, l) :- Actor(z,f,l), Casts(z,x),
Movie(x,y,'1940').

Q3(f,l) :- Actor(z,f,l), Casts(z,x1), Movie(x1,y1,1910),
Casts(z,x2), Movie(x2,y2,1940)

Find Actors who acted in a Movie in 1940 and in one in 1910

Actor(id, fname, lname)
Casts(pid, mid)
Movie(id, name, year)

Datalog: Facts and Rules

Facts = tuples in the database

Actor(344759,'Douglas', 'Fowley').
Casts(344759, 29851).
Casts(355713, 29000).
Movie(7909, 'A Night in Armour', 1910).
Movie(29000, 'Arizona', 1940).
Movie(29445, 'Ave Maria', 1940).

Rules = queries

Q1(y) :- Movie(x,y,z), z='1940'.

Q2(f, l) :- Actor(z,f,l), Casts(z,x),
Movie(x,y,'1940').

Q3(f,l) :- Actor(z,f,l), Casts(z,x1), Movie(x1,y1,1910),
Casts(z,x2), Movie(x2,y2,1940)

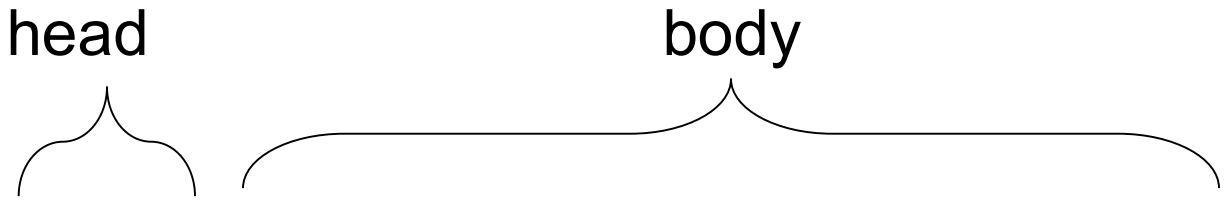
Extensional Database Predicates = EDB = Actor, Casts, Movie

Intensional Database Predicates = IDB = Q1, Q2, Q3

Anatomy of a Rule

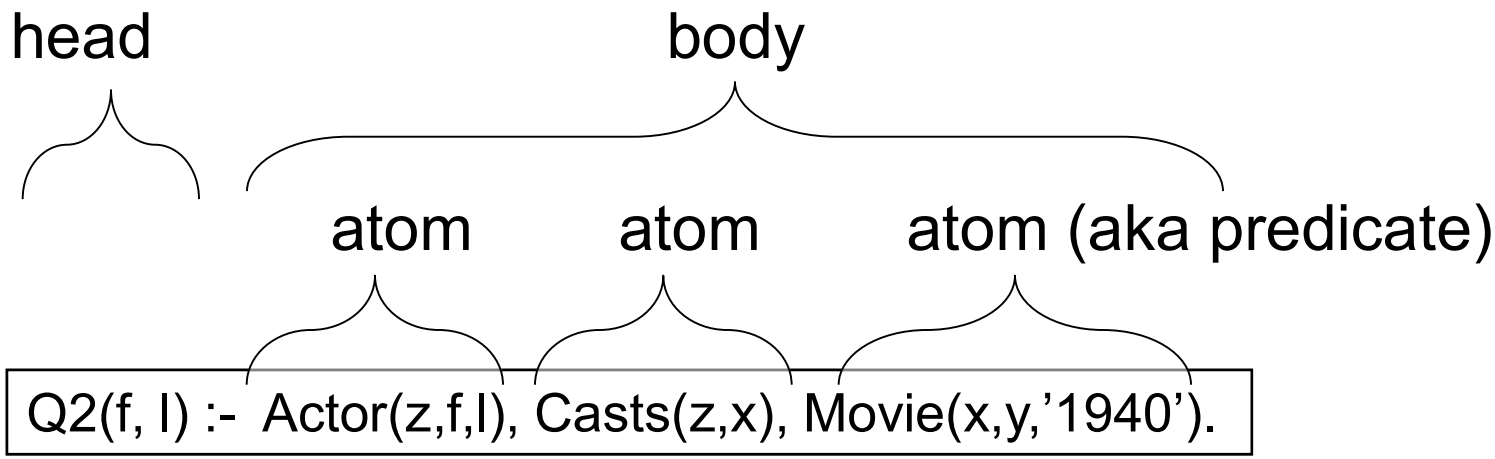
```
Q2(f, l) :-  Actor(z,f,l), Casts(z,x), Movie(x,y,'1940').
```

Anatomy of a Rule

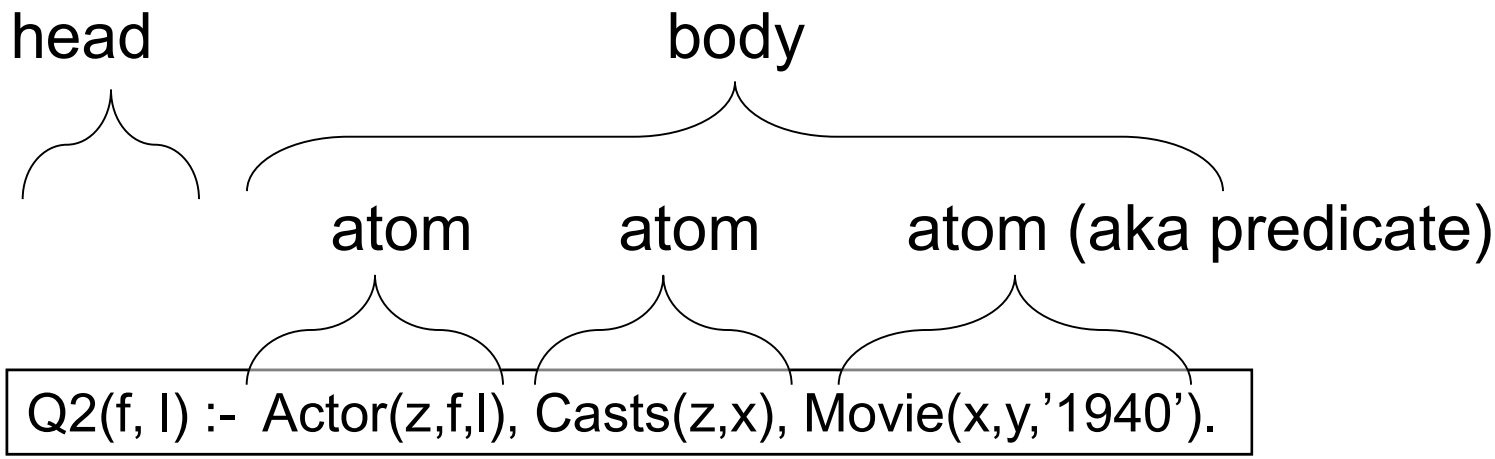


Q2(f, l) :- Actor(z,f,l), Casts(z,x), Movie(x,y,'1940').

Anatomy of a Rule



Anatomy of a Rule



f, l = head variables

x,y,z = existential variables

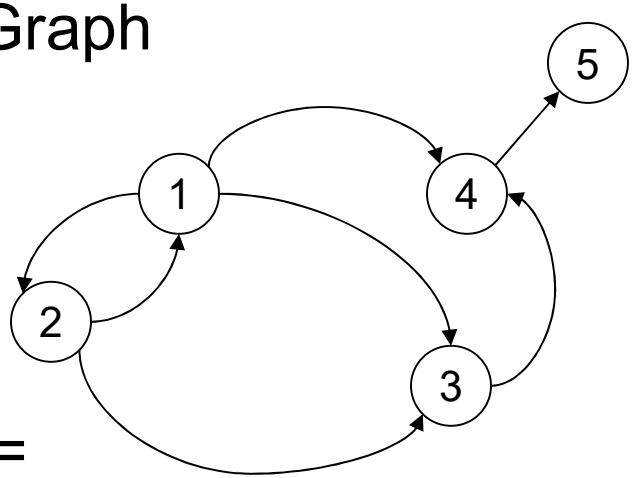
Outline

- Datalog rules
- Recursion
- Semantics

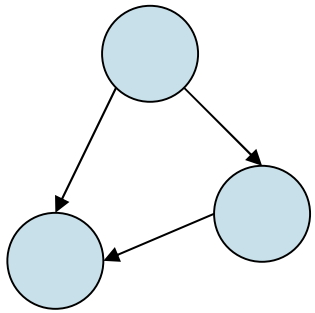
Next time: extensions, semi-naïve algo.

Processing Graphs in Datalog

Graph



Pattern Matching

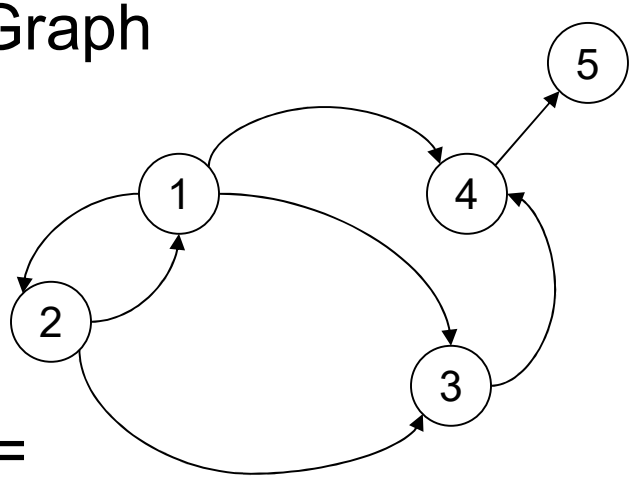


R=

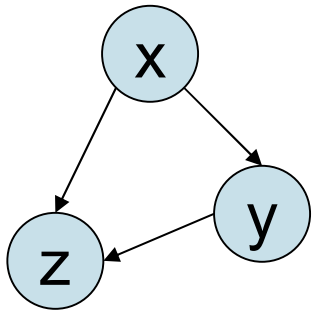
src	dst
1	2
2	1
2	3
1	4
3	4
4	5
1	3

Processing Graphs in Datalog

Graph



Pattern Matching



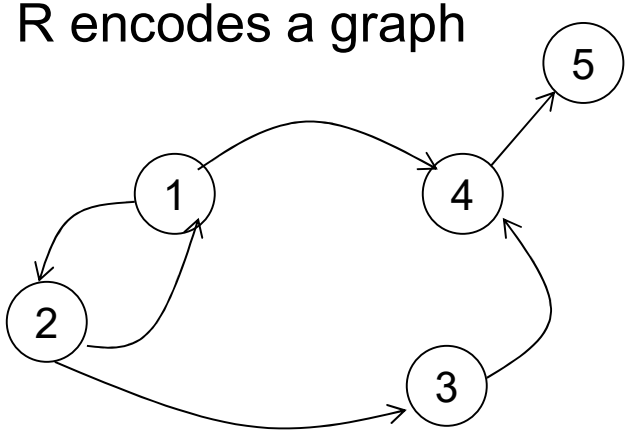
R=

src	dst
1	2
2	1
2	3
1	4
3	4
4	5
1	3

Answer(x,y,z) :- R(x,y), R(x,z), R(y,z)

Example

R encodes a graph



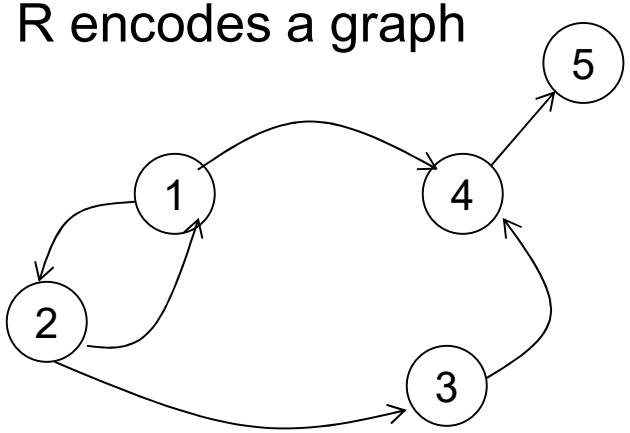
Descendants of node 2

R=

1	2
2	1
2	3
1	4
3	4
4	5

Example

R encodes a graph



R=

1	2
2	1
2	3
1	4
3	4
4	5

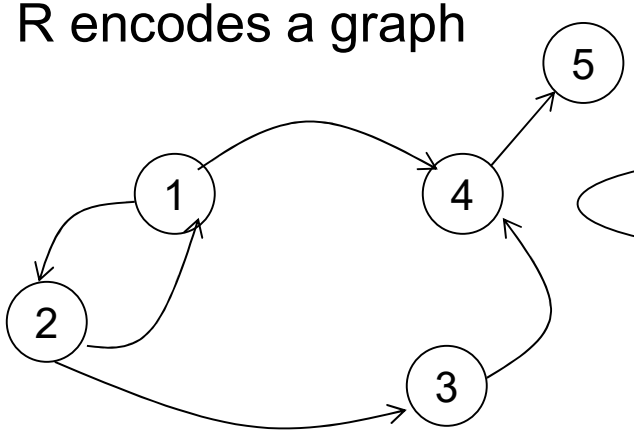
Descendants of node 2

D(x) :- R(2, x)

D(y) :- D(x), R(x,y)

Example

R encodes a graph



Descendants of node 2

Recursive rule

D(x) :- R(2, x)

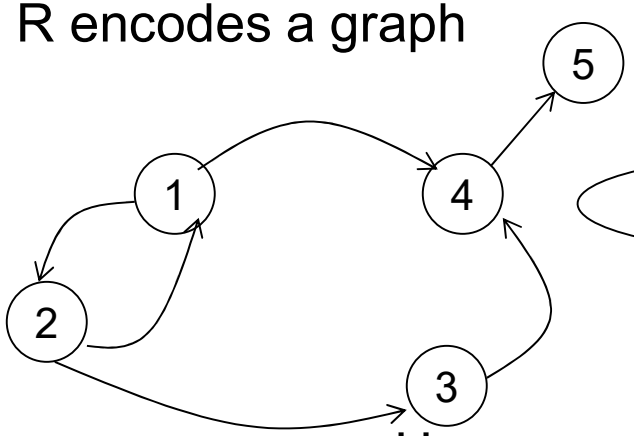
D(y) :- D(x), R(x,y)

R=

1	2
2	1
2	3
1	4
3	4
4	5

Example

R encodes a graph



Descendants of node 2

Recursive rule

D(x) :- R(2, x)

D(y) :- D(x), R(x,y)

How recursion works in datalog:

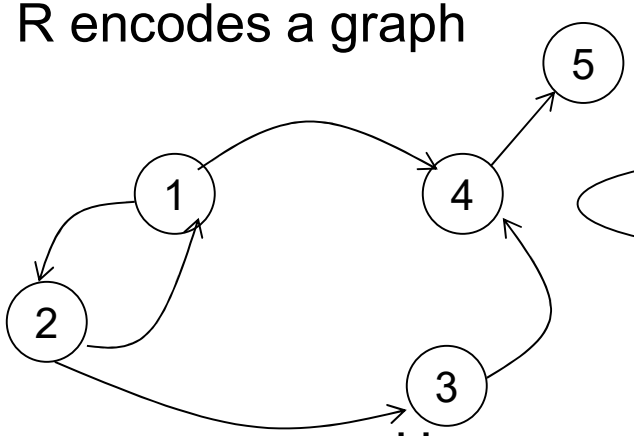
Initially D = empty

R=

1	2
2	1
2	3
1	4
3	4
4	5

Example

R encodes a graph



Descendants of node 2

Recursive rule

D(x) :- R(2, x)
D(y) :- D(x), R(x,y)

R=

1	2
2	1
2	3
1	4
3	4
4	5

How recursion works in datalog:

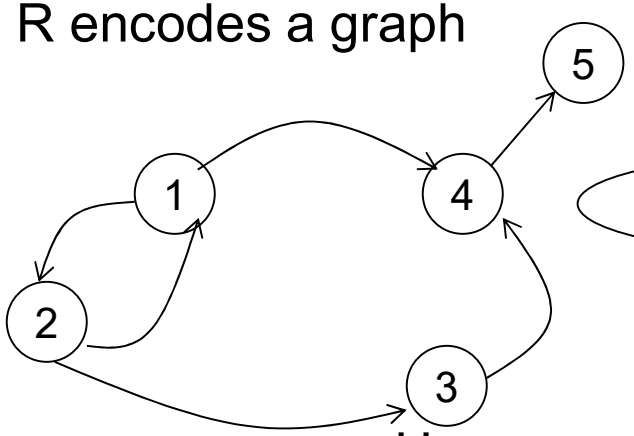
Initially D = empty

- Compute both rules:

D(x) :- R(2, x)
D(y) :- D(x), R(x,y)

Example

R encodes a graph



Descendants of node 2

Recursive rule

D(x) :- R(2, x)
D(y) :- D(x), R(x,y)

R=

1	2
2	1
2	3
1	4
3	4
4	5

How recursion works in datalog:

Initially D = empty

- Compute both rules:
  …now D = {1,3}

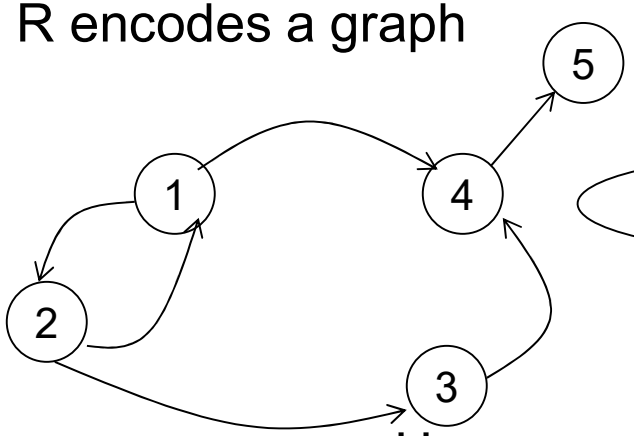
{1,3}

{ }

D(x) :- R(2, x)
D(y) :- D(x), R(x,y)

Example

R encodes a graph



Descendants of node 2

Recursive rule

D(x) :- R(2, x)
D(y) :- D(x), R(x,y)

R=

1	2
2	1
2	3
1	4
3	4
4	5

How recursion works in datalog:

Initially D = empty

- Compute both rules:
  …now D = {1,3}
- Compute both rules:

{1,3}

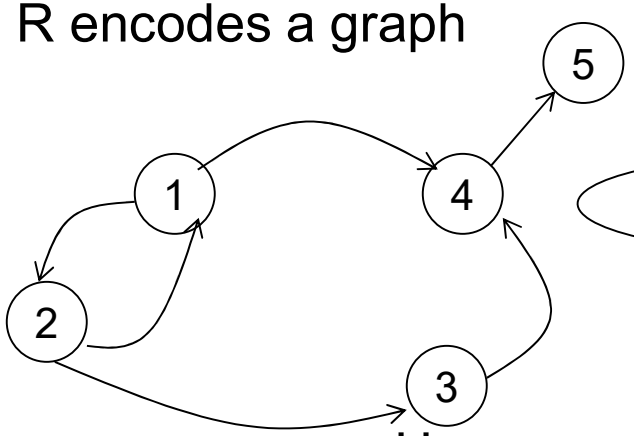
{ }

D(x) :- R(2, x)
D(y) :- D(x), R(x,y)

D(x) :- R(2, x)
D(y) :- D(x), R(x,y)

Example

R encodes a graph



Descendants of node 2

Recursive rule

D(x) :- R(2, x)
D(y) :- D(x), R(x,y)

R=

1	2
2	1
2	3
1	4
3	4
4	5

How recursion works in datalog:

Initially D = empty

- Compute both rules:
  …now D = {1,3}
- Compute both rules:
  …now D = {1,3,2,4}

{1,3}

{ }

D(x) :- R(2, x)
D(y) :- D(x), R(x,y)

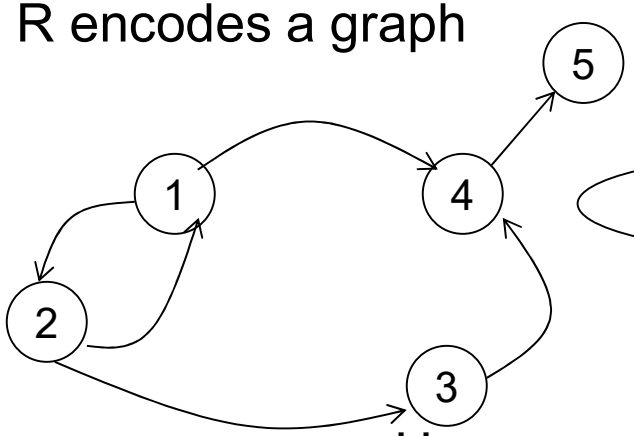
{1,3}

{2,4 }

D(x) :- R(2, x)
D(y) :- D(x), R(x,y)

Example

R encodes a graph



Descendants of node 2

Recursive rule

D(x) :- R(2, x)
D(y) :- D(x), R(x,y)

R=

1	2
2	1
2	3
1	4
3	4
4	5

How recursion works in datalog:

Initially D = empty

- Compute both rules:
  …now D = {1,3}
- Compute both rules:
  …now D = {1,3,2,4}
- Compute both rules:

{1,3}

{ }

D(x) :- R(2, x)
D(y) :- D(x), R(x,y)

{1,3}

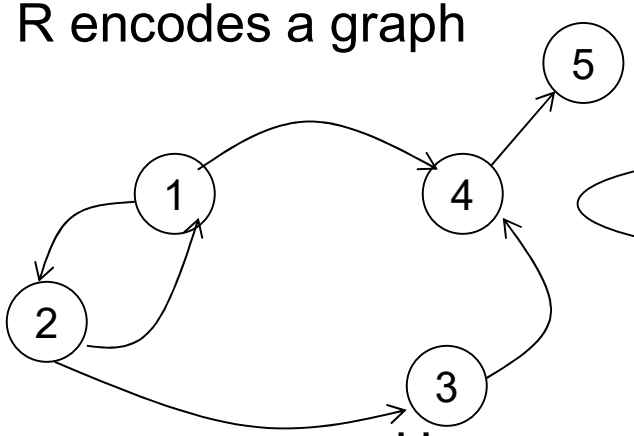
{2,4 }

D(x) :- R(2, x)
D(y) :- D(x), R(x,y)

D(x) :- R(2, x)
D(y) :- D(x), R(x,y)

Example

R encodes a graph



Descendants of node 2

Recursive rule

D(x) :- R(2, x)
D(y) :- D(x), R(x,y)

R=

1	2
2	1
2	3
1	4
3	4
4	5

How recursion works in datalog:

Initially D = empty

- Compute both rules:
  …now D = {1,3}
- Compute both rules:
  …now D = {1,3,2,4}
- Compute both rules:
  …now D = {1,3,2,4,5}

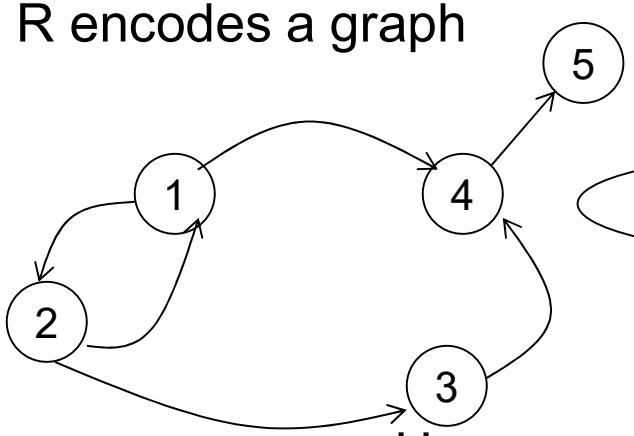
{1,3}
{ }
D(x) :- R(2, x)
D(y) :- D(x), R(x,y)

{1,3}
{2,4 }
D(x) :- R(2, x)
D(y) :- D(x), R(x,y)

{1,3}
{2,4,1,3,5 }
D(x) :- R(2, x)
D(y) :- D(x), R(x,y)

Example

R encodes a graph



Descendants of node 2

Recursive rule

D(x) :- R(2, x)
D(y) :- D(x), R(x,y)

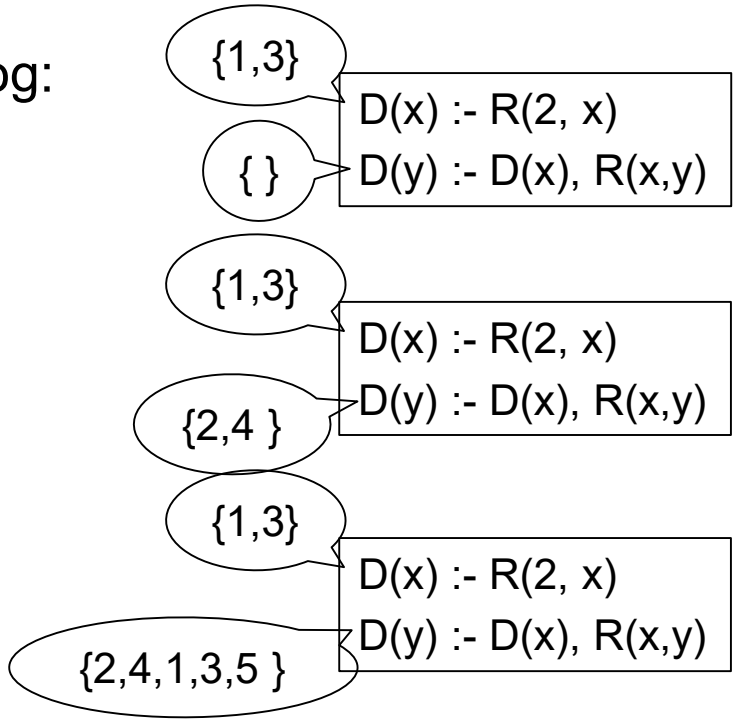
R=

1	2
2	1
2	3
1	4
3	4
4	5

How recursion works in datalog:

Initially D = empty

- Compute both rules:
  …now D = {1,3}
- Compute both rules:
  …now D = {1,3,2,4}
- Compute both rules:
  …now D = {1,3,2,4,5}
- Compute both rules:
  …nothing new. STOP



Outline

- Datalog rules
- Recursion
- Semantics

Next time: extensions, semi-naïve algo.

Datalog program

- A datalog program = several rules
- Rules may be recursive
- Set semantics only

Naïve Evaluation Algorithm

- Every rule → SPJ* query

*SPJ = select-project-join
+USPJ = union-select-project-join

Naïve Evaluation Algorithm

- Every rule → SPJ* query

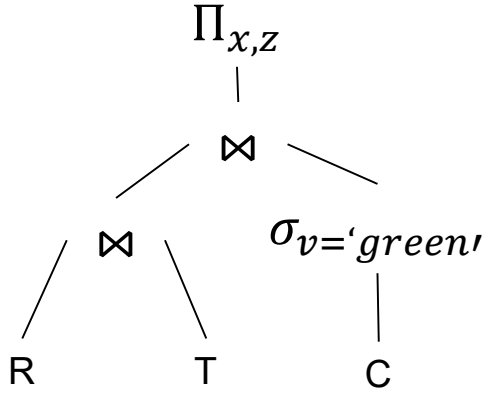
T(x,z) :- R(x,y), T(y,z), C(y,'green')

*SPJ = select-project-join
+USPJ = union-select-project-join

Naïve Evaluation Algorithm

- Every rule → SPJ* query

T(x,z) :- R(x,y), T(y,z), C(y,'green')

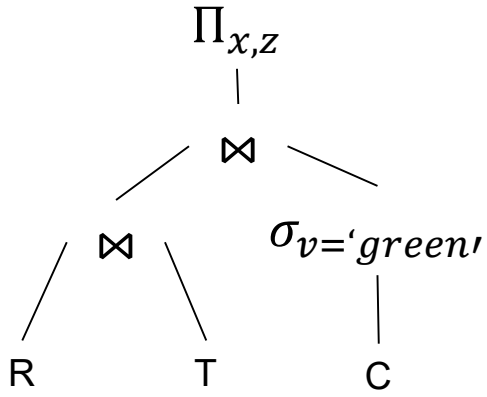


*SPJ = select-project-join
+USPJ = union-select-project-join

Naïve Evaluation Algorithm

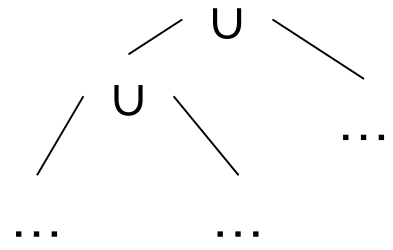
- Every rule → SPJ* query

T(x,z) :- R(x,y), T(y,z), C(y,'green')



- Multiple rules same head → USPJ+

T(x,y) :- …
T(x,y) :- …
…

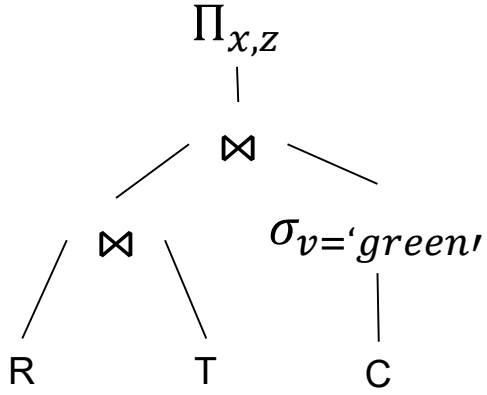


*SPJ = select-project-join
+USPJ = union-select-project-join

Naïve Evaluation Algorithm

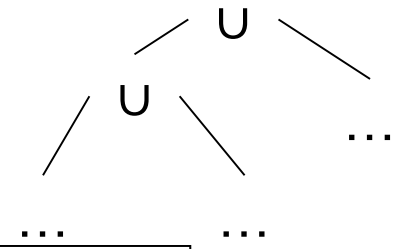
- Every rule → SPJ* query

T(x,z) :- R(x,y), T(y,z), C(y,'green')



- Multiple rules same head → USPJ+

T(x,y) :- …
T(x,y) :- …
…



- Naïve Algorithm:

$IDBs := \emptyset$
repeat $IDBs := USPJs$
until no more change

*SPJ = select-project-join
+USPJ = union-select-project-join

Naïve Evaluation Algorithm

```
D(x) :- R(2,x)
D(y) :- D(x),R(x,y)
```

Naïve Evaluation Algorithm

D(x) :- R(2,x)

D(y) :- D(x),R(x,y)

$$\Pi_{R.dst}(\sigma_{R.src=2}(R))$$

Naïve Evaluation Algorithm

D(x) :- R(2,x)

D(y) :- D(x),R(x,y)

$$\Pi_{R.dst}(\sigma_{R.src=2}(R)) \cup \Pi_{R.dst}(D \bowtie_{D.node=R.src} R);$$

Naïve Evaluation Algorithm

D(x) :- R(2,x)

D(y) :- D(x),R(x,y)

$$\Pi_{R.dst}(\sigma_{R.src=2}(R)) \cup \Pi_{R.dst}(D \bowtie_{D.node=R.src} R);$$

Naïve Evaluation Algorithm

D(x) :- R(2,x)

D(y) :- D(x),R(x,y)

$D := \emptyset$;

repeat

$D := \Pi_{R.dst}(\sigma_{R.src=2}(R)) \cup \Pi_{R.dst}(D \bowtie_{D.node=R.src} R)$;

until [no more change]

Naïve Evaluation Algorithm

The Naïve Evaluation Algorithm:

- Always terminates
- Always terminates in a number of steps that is polynomial in the size of the database