CS 347
Distributed Databases and Transaction Processing
Notes03: Query Processing

Hector Garcia-Molina
Zoltan Gyongyi

Query Processing

- Decomposition
- Localization
- Optimization

Decomposition

- Same as in centralized system

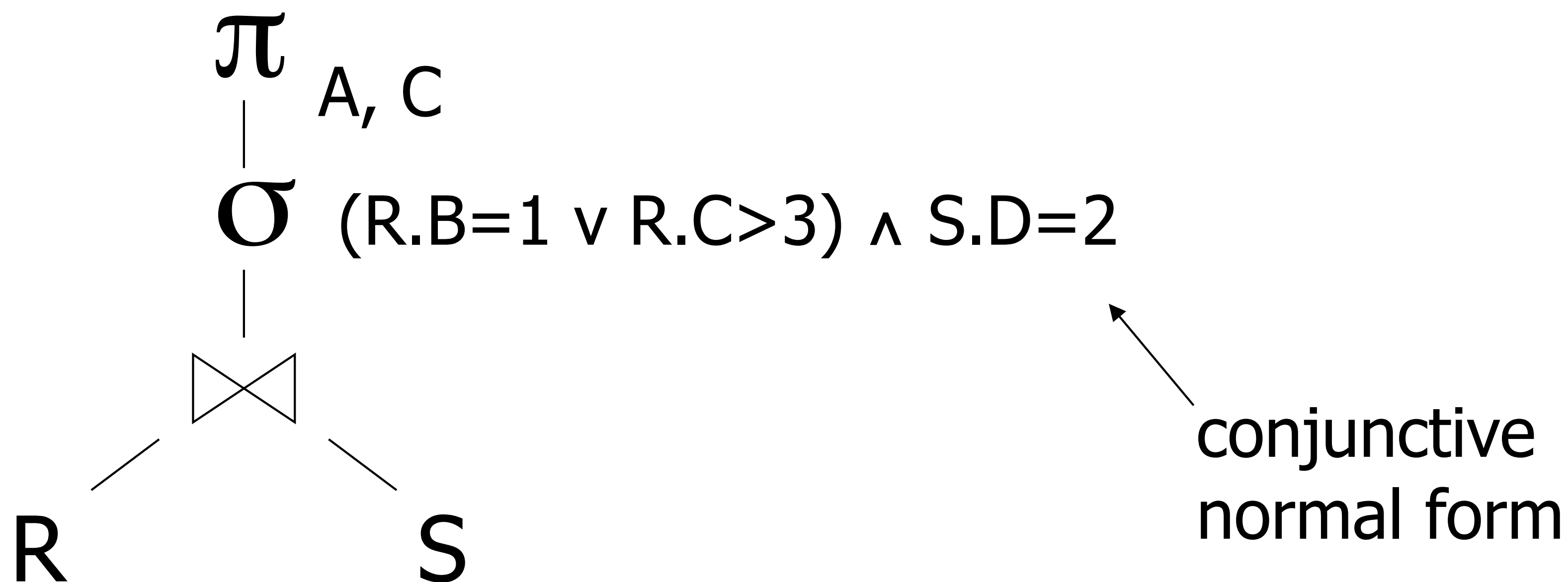
- Normalization
- Eliminating redundancy
- Algebraic rewriting

Normalization

- Convert from general language to a "standard" form (e.g., relational algebra)

Example

select A, C
from R, S
where (R.B=1 and S.D=2) or (R.C>3 and S.D=2)

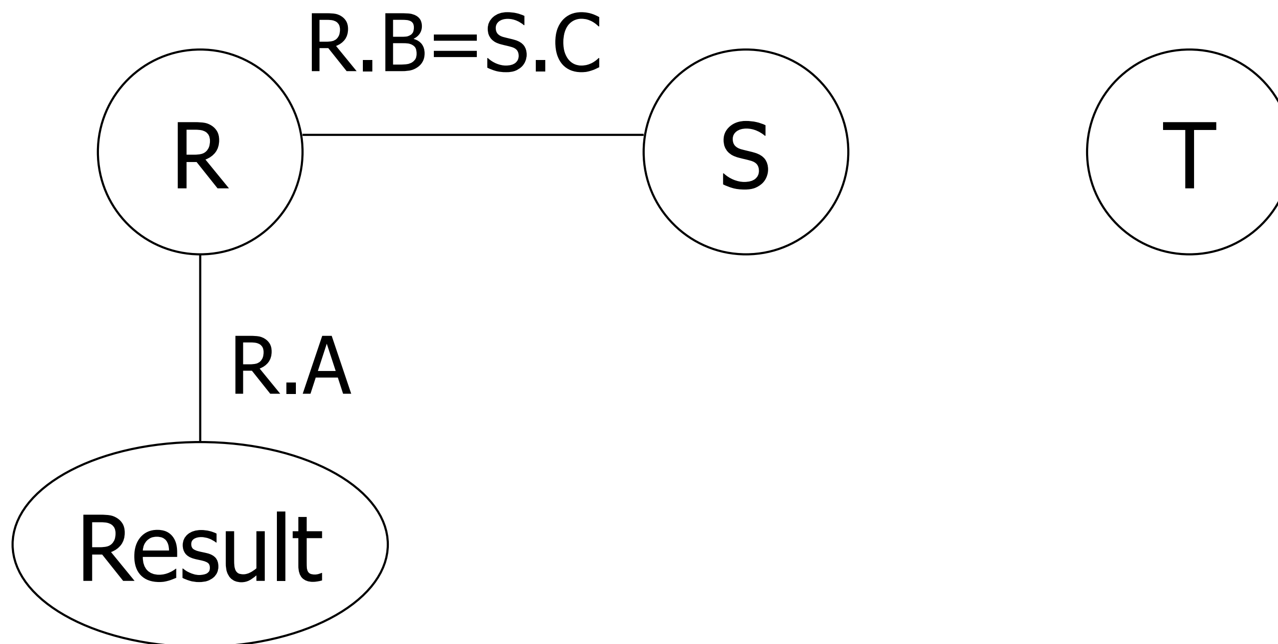


<u>Also</u>: Detect invalid expressions

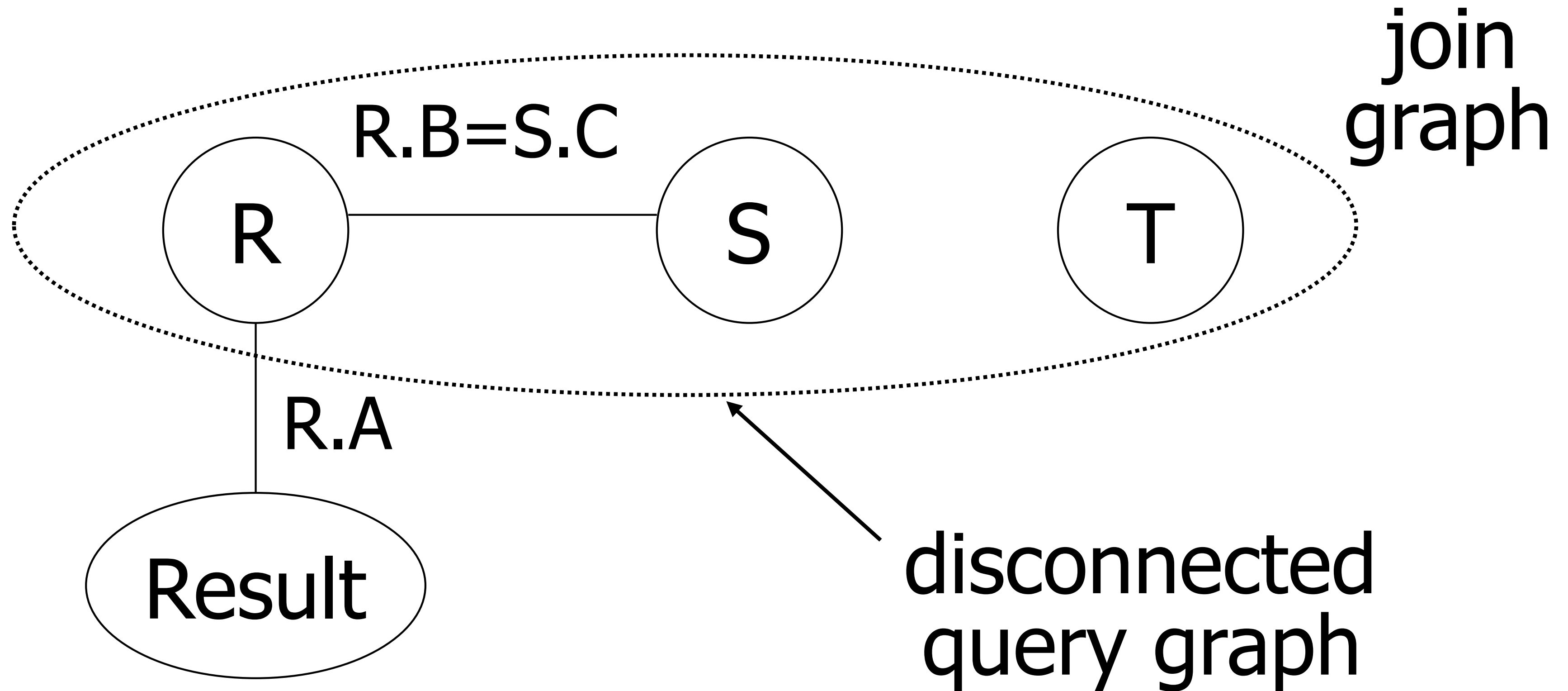
E.g.: select * from R where R.A=3
✖ R does not have "A" attribute

E.g.: select R.A
from R, S, T
where R.B=S.C

Note: Query graph useful for detecting last problem



Note: Query graph useful for detecting last problem



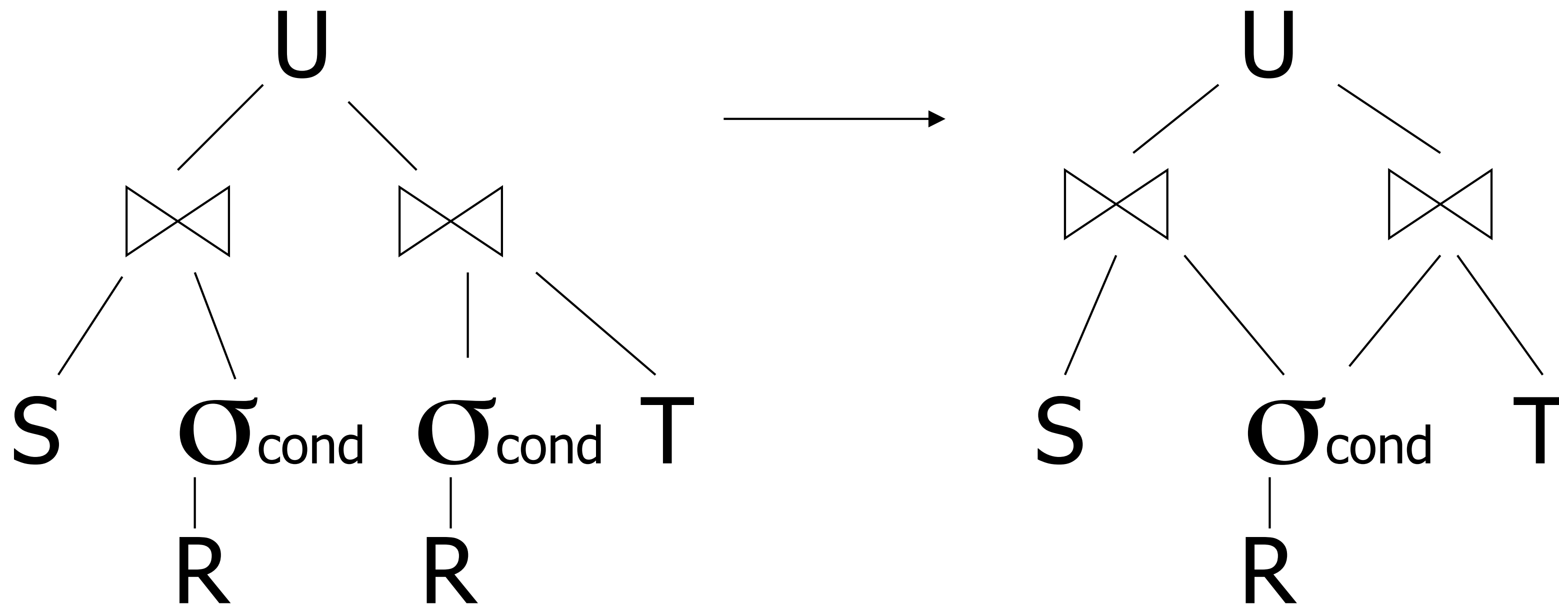
Eliminate redundancy

E.g.: in conditions

$(S.A=1) \wedge (S.A>5) \Rightarrow$ false

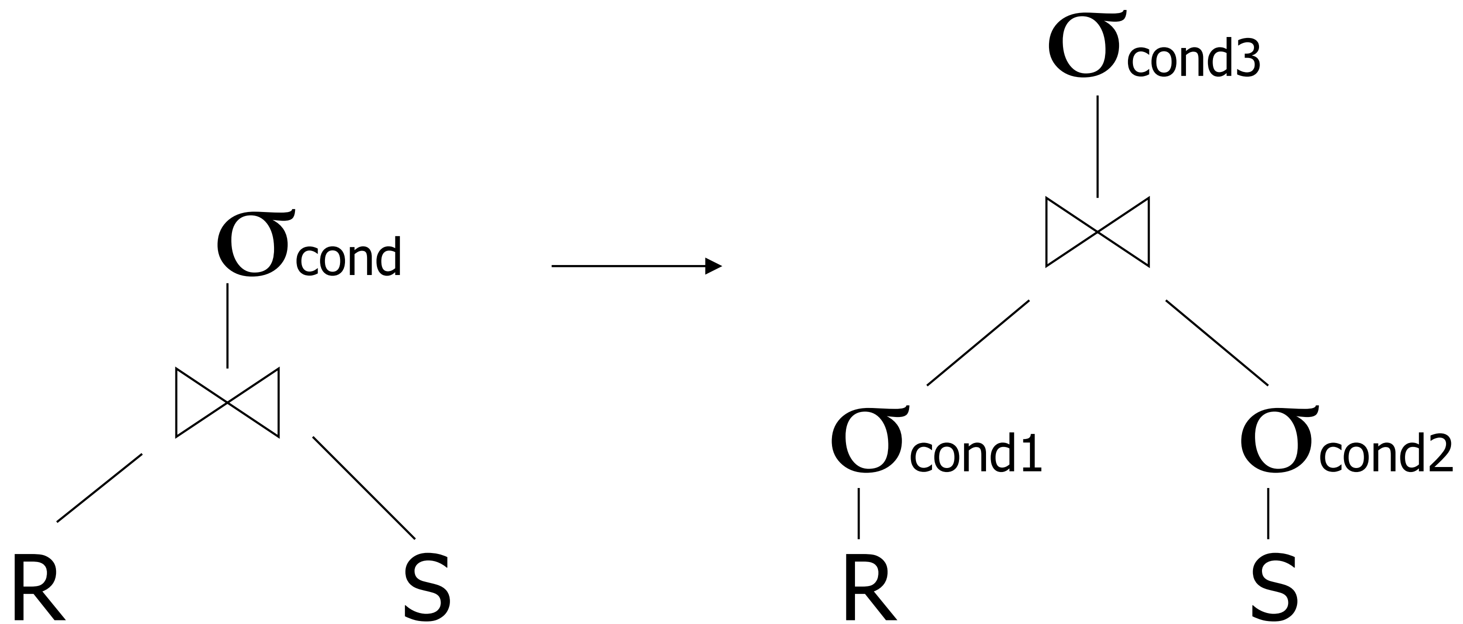
$(S.A<10) \wedge (S.A<5) \Rightarrow S.A<5$

E.g.: common sub-expressions



Algebraic rewriting

E.g.: push conditions down



- After decomposition:
  - One or more algebraic query trees on relations
- Localization:
  - Replace relations by corresponding fragments

Localization steps

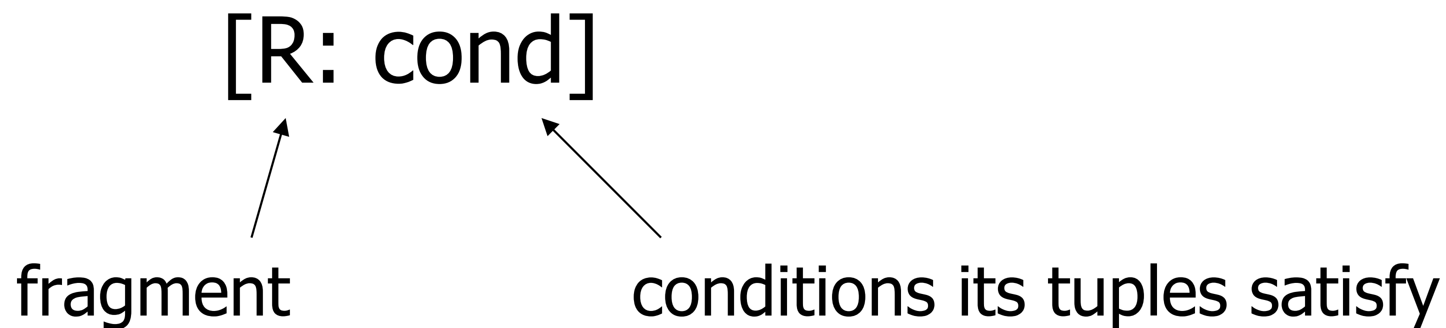
(1) Start with query

(2) Replace relations by fragments

(3) Push $\begin{cases} \cup \text{ up} \\ \pi, \sigma \text{ down} \end{cases}$ (use CS 245 rules)

(4) Simplify – eliminate unnecessary operations

Notation for fragment



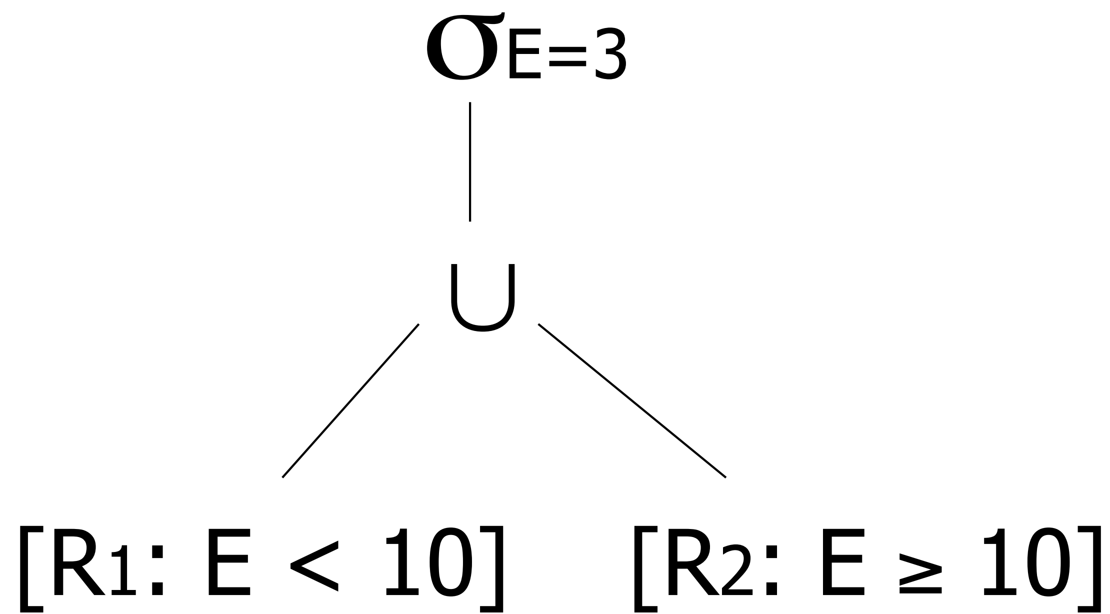
Example A

(1) $\sigma_{E=3}$

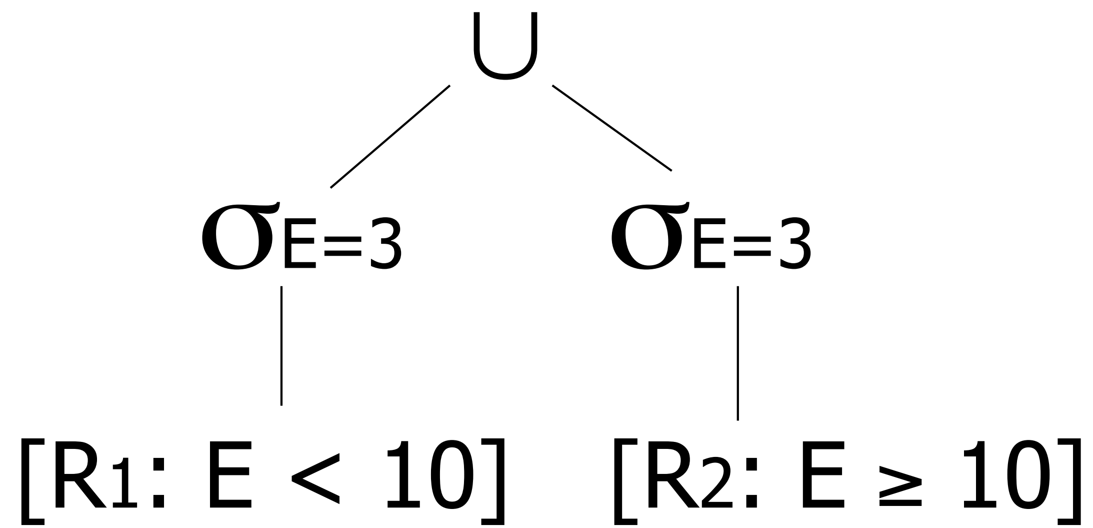
$|$

R

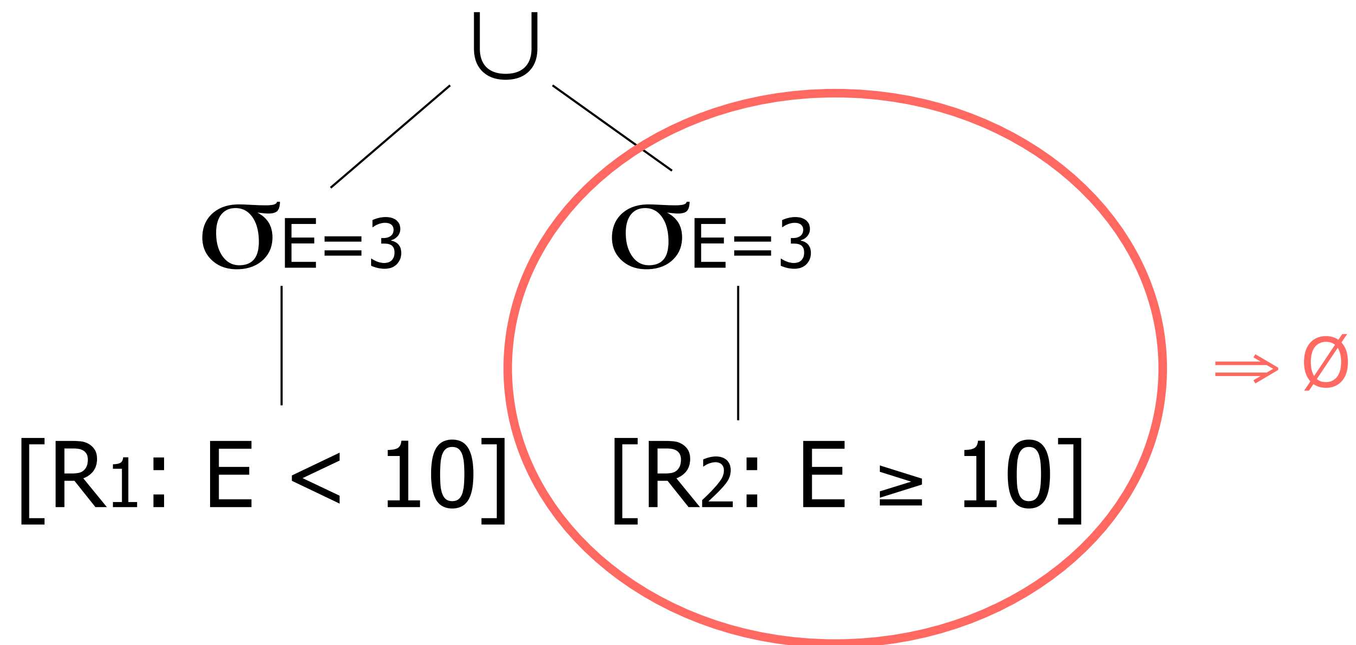
(2)



(3)



(3)



(4)

$$\sigma_{E=3}$$

[R_1: E < 10]

Rule 1

(A) $\sigma_{C1}[R: c2] \Rightarrow \sigma_{C1}[R: c1 \wedge c2]$

(B) $[R: False] \Rightarrow \emptyset$

In example A:

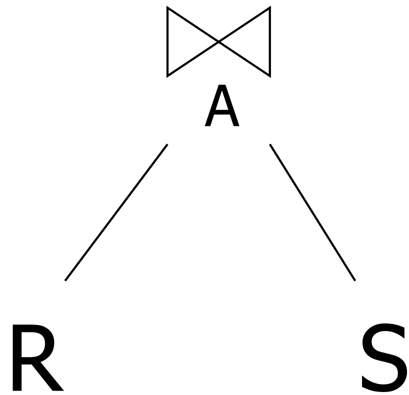
$\sigma_{E=3}[R_2: E \geq 10] \Rightarrow [\sigma_{E=3} R_2: E=3 \wedge E \geq 10]$

$\Rightarrow [\sigma_{E=3} R_2: \text{False}]$

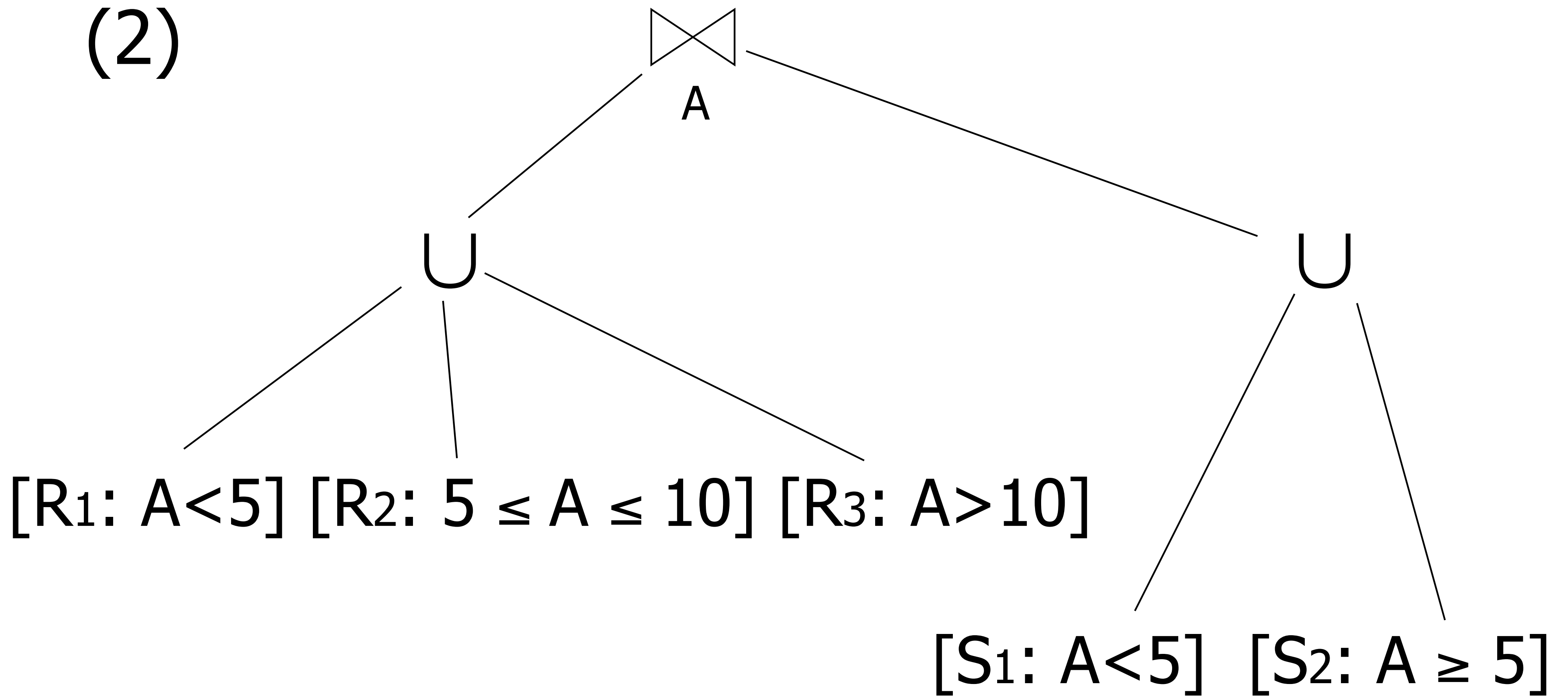
$\Rightarrow \emptyset$

Example B

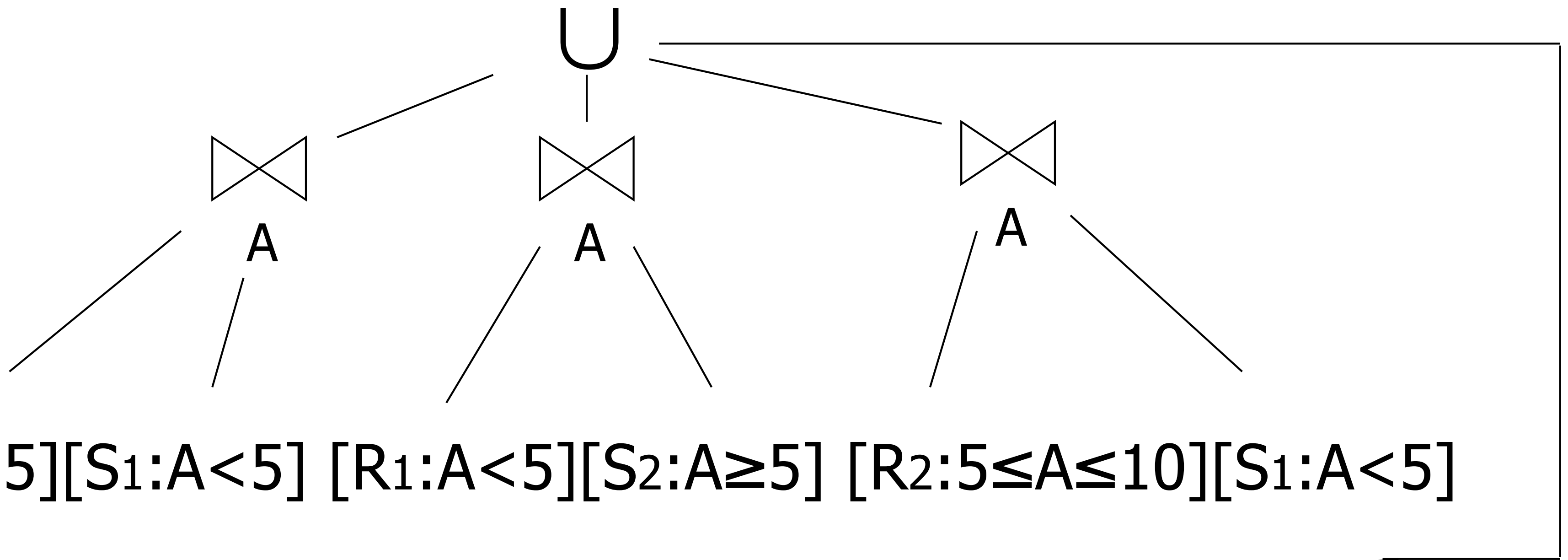
(1)

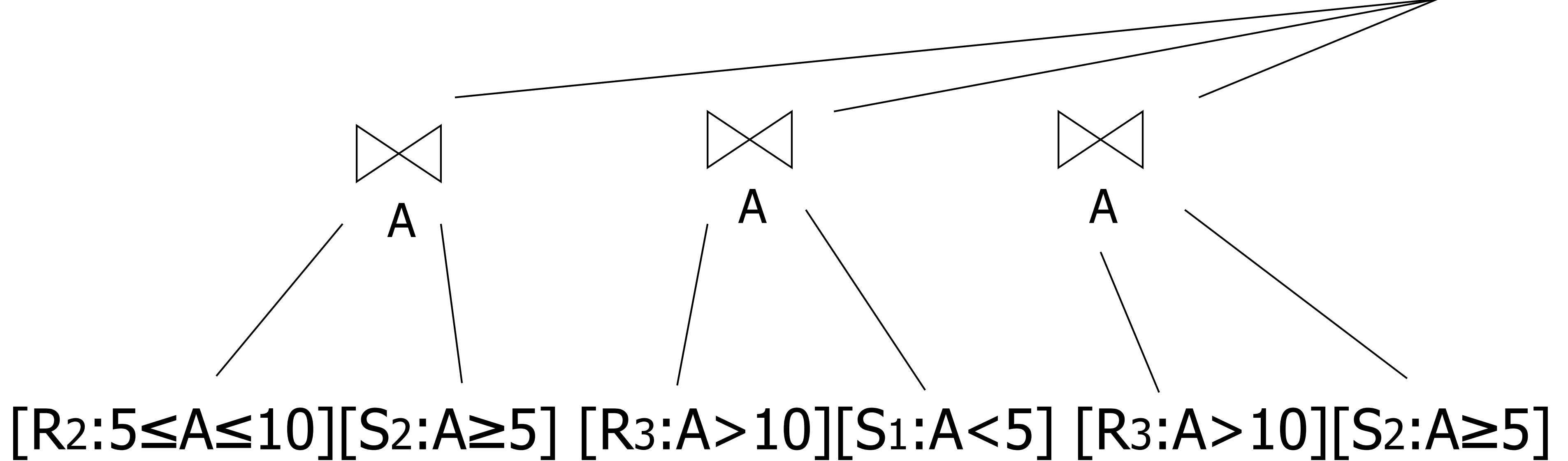


A = common attribute

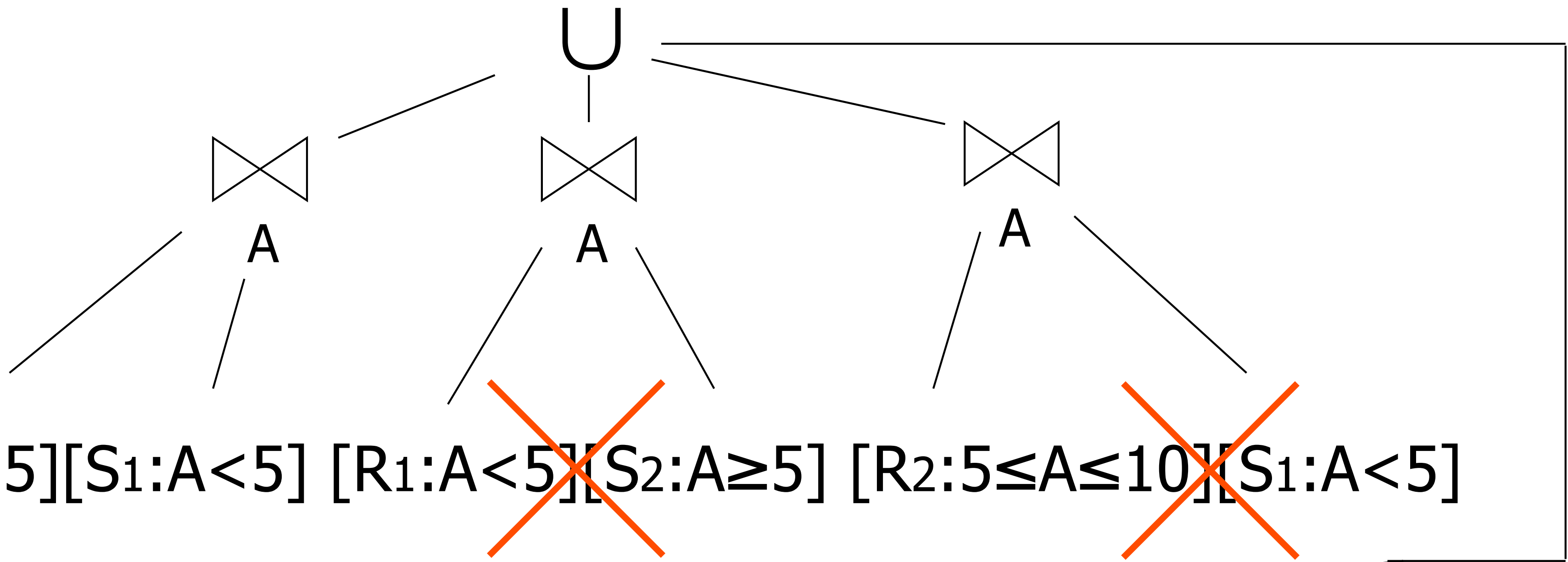

(2)
⋈
A
∪
∪
[R1: A<5] [R2: 5 ≤ A ≤ 10] [R3: A>10]
[S1: A<5] [S2: A ≥ 5]

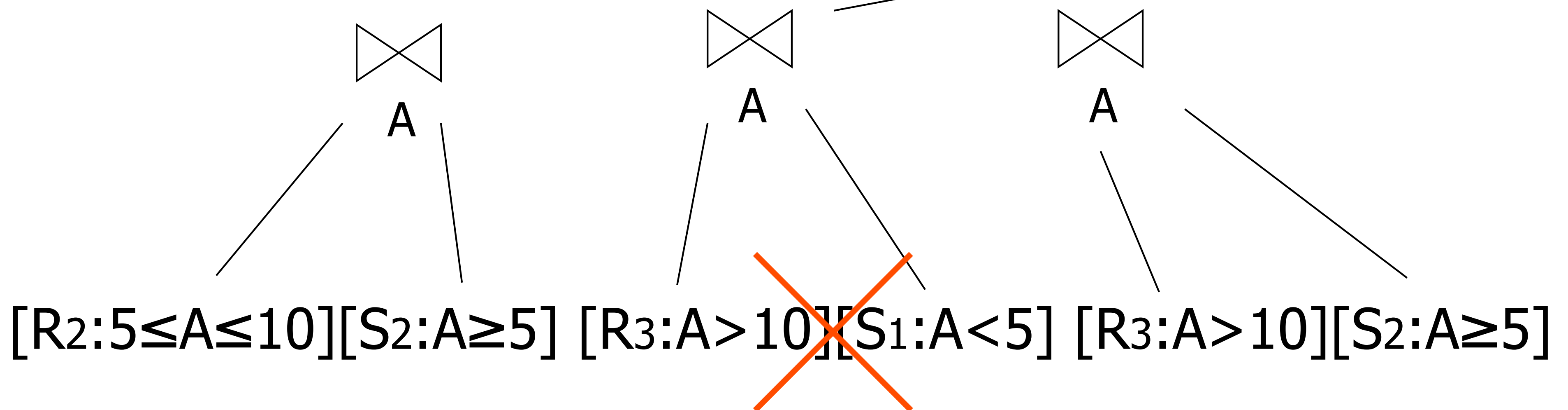
(3)

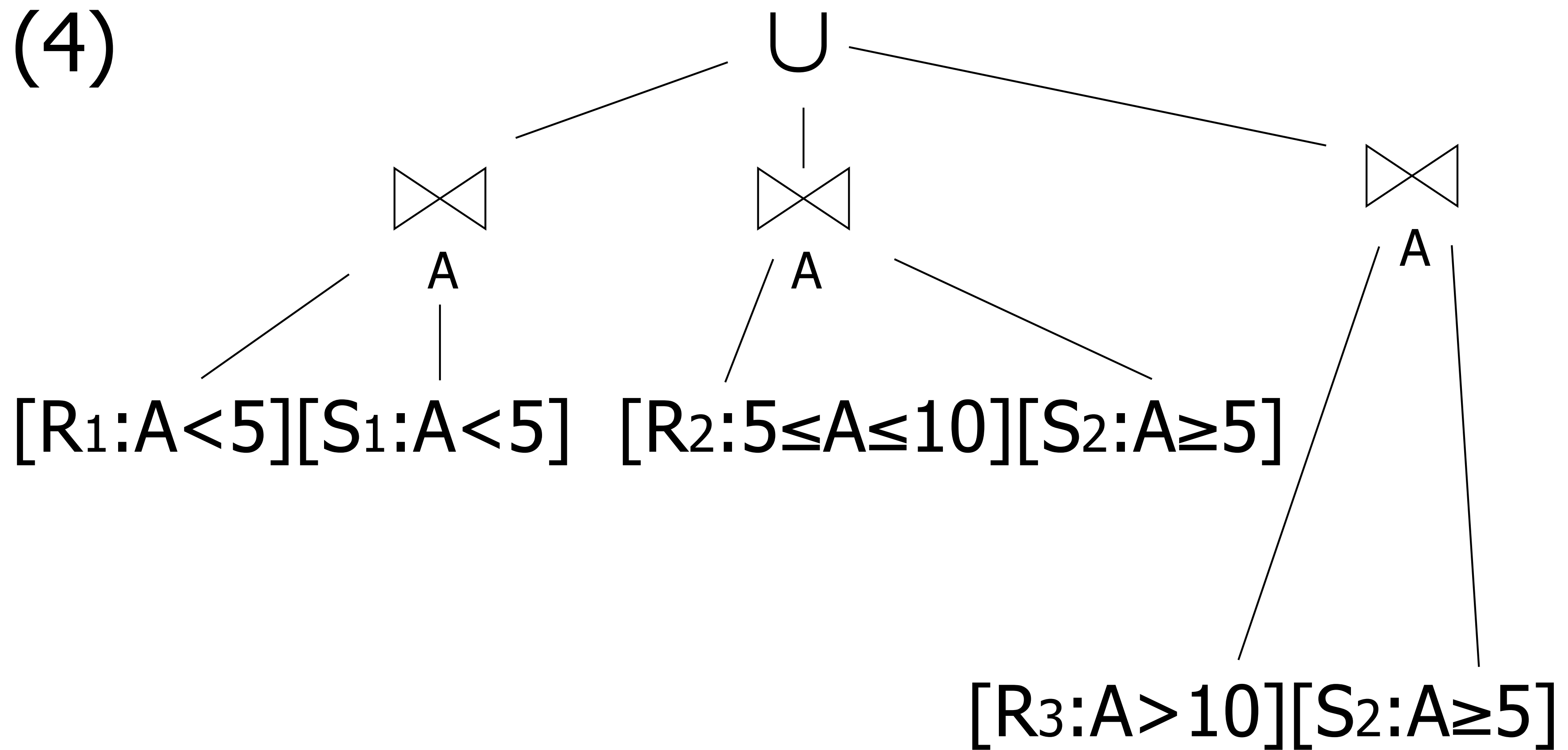




(3)






(4)
∪
A
A
A
[R1:A<5][S1:A<5]
[R2:5≤A≤10][S2:A≥5]
[R3:A>10][S2:A≥5]

Rule 2

$$[R: C_1] \bowtie_A [S: C_2] \Rightarrow$$

$$[R \bowtie_A S: C_1 \wedge C_2 \wedge R.A = S.A]$$

In step 4 of Example B:

$$[R_1: A<5] \bowtie_A [S_2: A \geq 5]$$

$$\Rightarrow [R_1 \bowtie_A S_2: \; R_1.A < 5 \;\wedge\; S_2.A \geq 5 \;\wedge\; R_1.A = S_2.A\,]$$

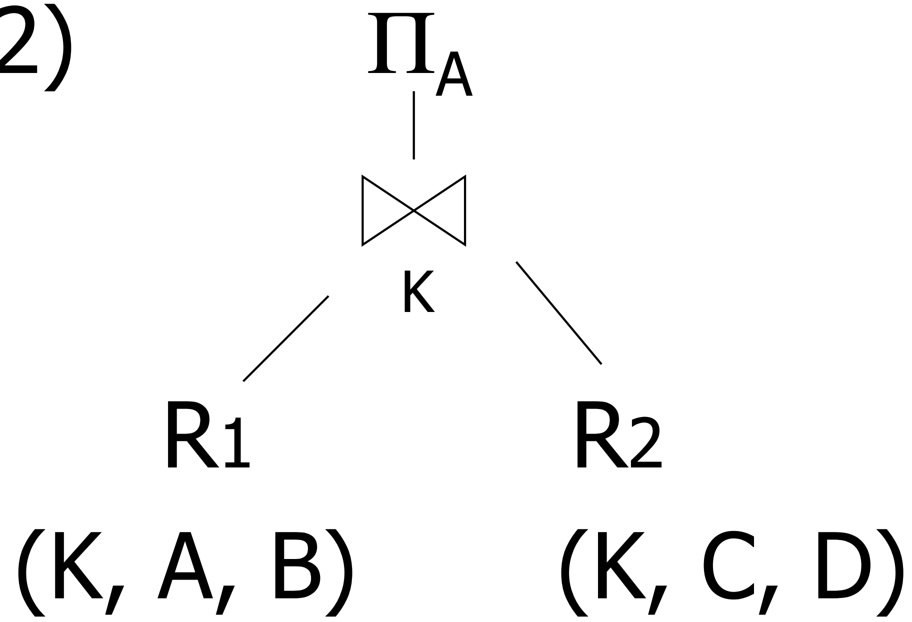
$$\Rightarrow [R_1 \bowtie_A S_2: \text{False}] \;\Rightarrow\; \emptyset$$

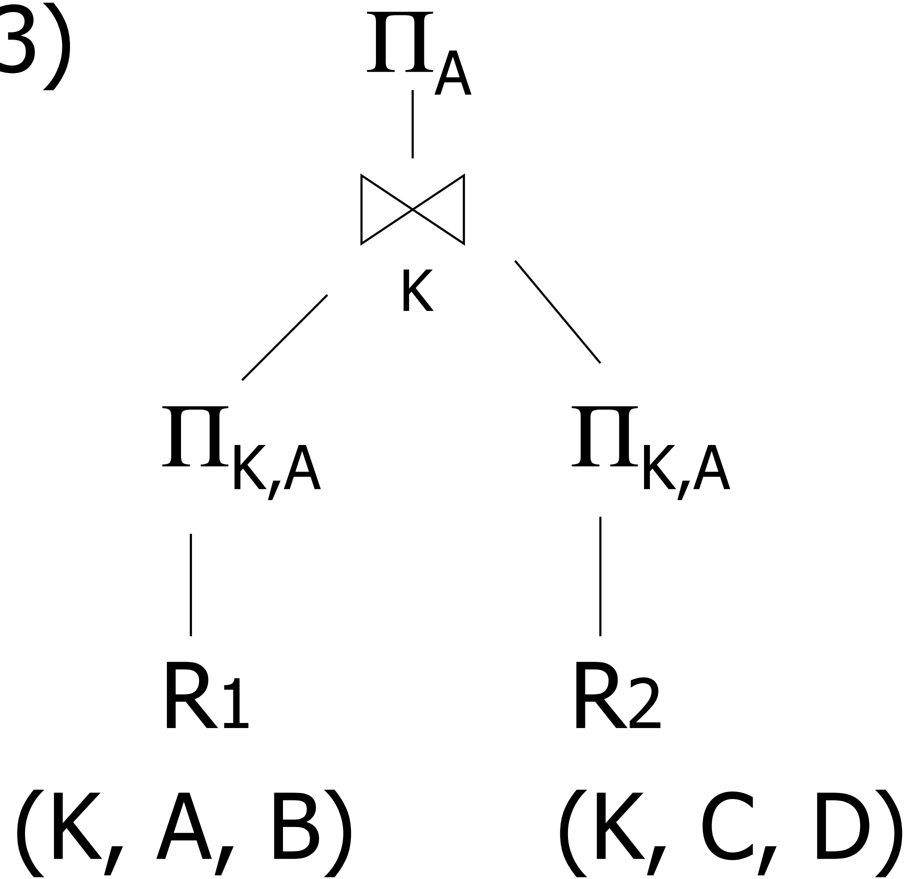
- Localization with vertical fragmentation

Example C

(1) Π_A over R (Π_A — R)

$\left\{ \begin{array}{l} R_1(K, A, B) \\ R_2(K, C, D) \end{array} \right.$


(2)
Π_A
$\bowtie$
K
R_1
R_2
(K, A, B)
(K, C, D)


(3)
Π_A
K
$\Pi_{K,A}$
$\Pi_{K,A}$
R_1
R_2
(K, A, B)
(K, C, D)

(4) Π_A

R_1

(K, A, B)

Rule 3

- Given vertical fragmentation of R:

  $R_i = \Pi_{A_i}(R)$, $A_i \subseteq A$
- Then for any $B \subseteq A$:

  $\Pi_B(R) = \Pi_B [\bowtie_i R_i \mid B \cap A_i \neq \emptyset]$

Summary - Query Processing

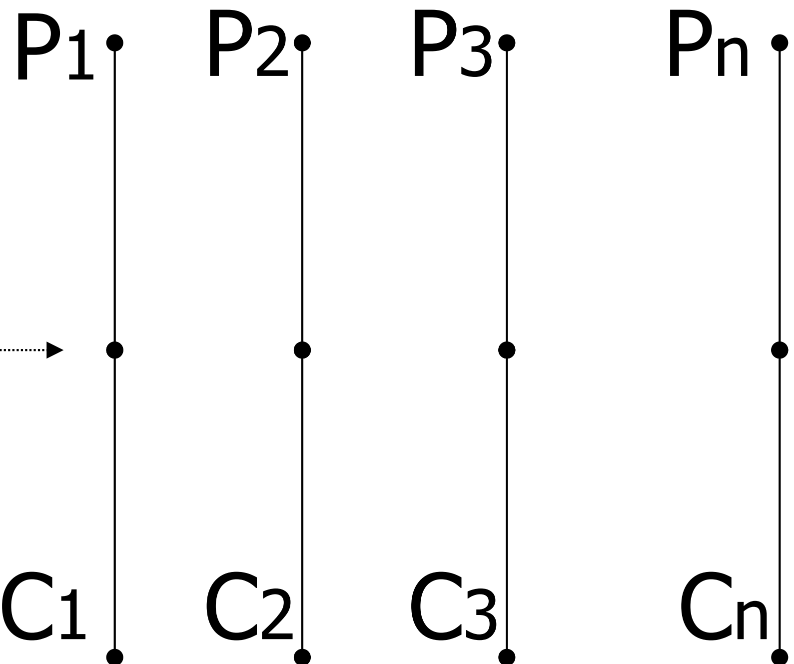
- Decomposition ✓
- Localization ✓
- Optimization
  – Overview
  – Tricks for joins + other operations
  – Strategies for optimization

Optimization Process:

Generate query plans

Estimate size of intermediate results

Estimate cost of plan ($,time,…)



Differences from centralized optimization:

- New strategies for some operations (semi-join, range-partitioning, sort, ...)
- Many ways to assign and schedule processors

Parallel/distributed sort

Input: (a) relation R on single site/disk
(b) R fragmented/partitioned by sort attribute
(c) R fragmented/partitioned by other attribute

Output: (a) sorted R on single site/disk

(b) fragments/partitions sorted

5	...
6	...
10	

F_1

12	...
15	

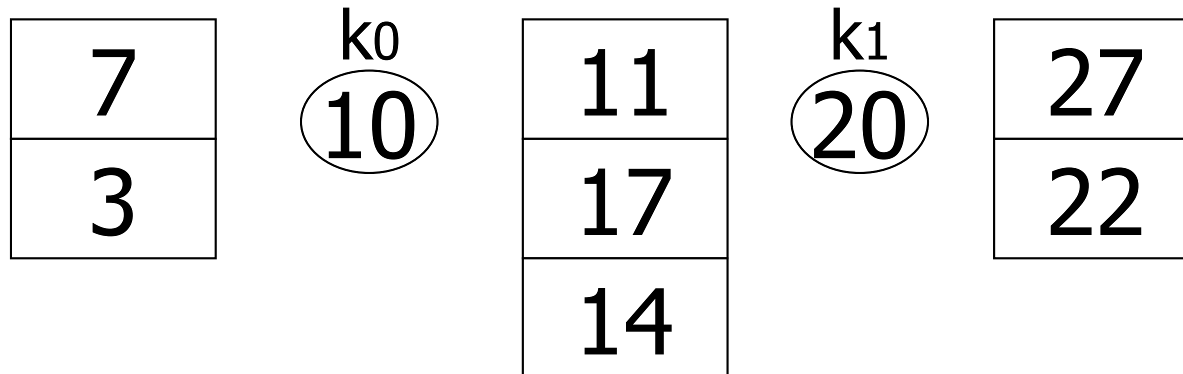
F_2

19	...
20	
21	
50	

F_3

Basic sort

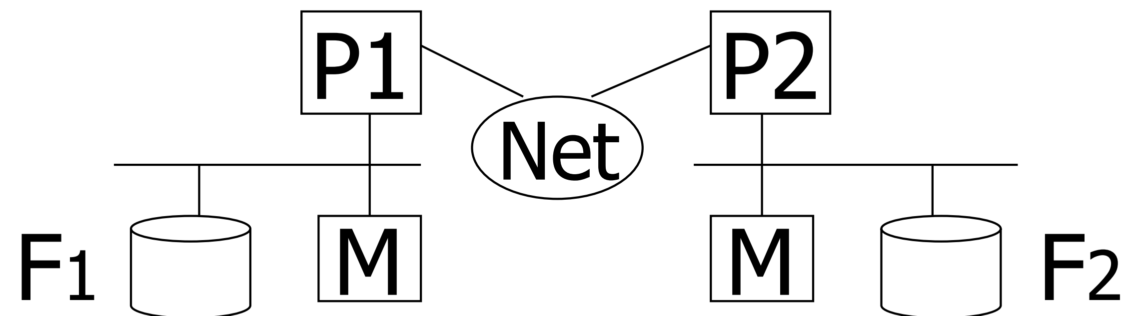
- R(K, …); sort on K
- Fragmented (range partitioned) on K
  Vector: [k_0, k_1, ..., k_n]



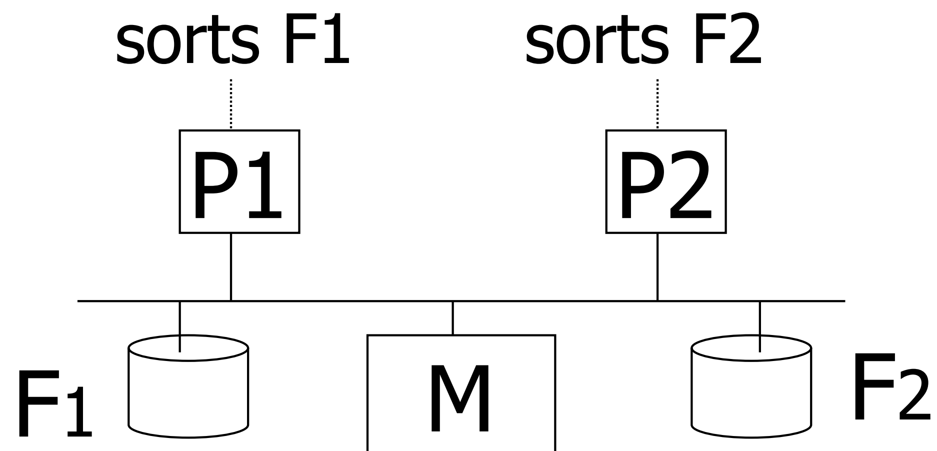
- Algorithm:
  - Sort each fragment independently
  - If necessary, ship results

⇒ Same idea on different architectures:

Shared nothing



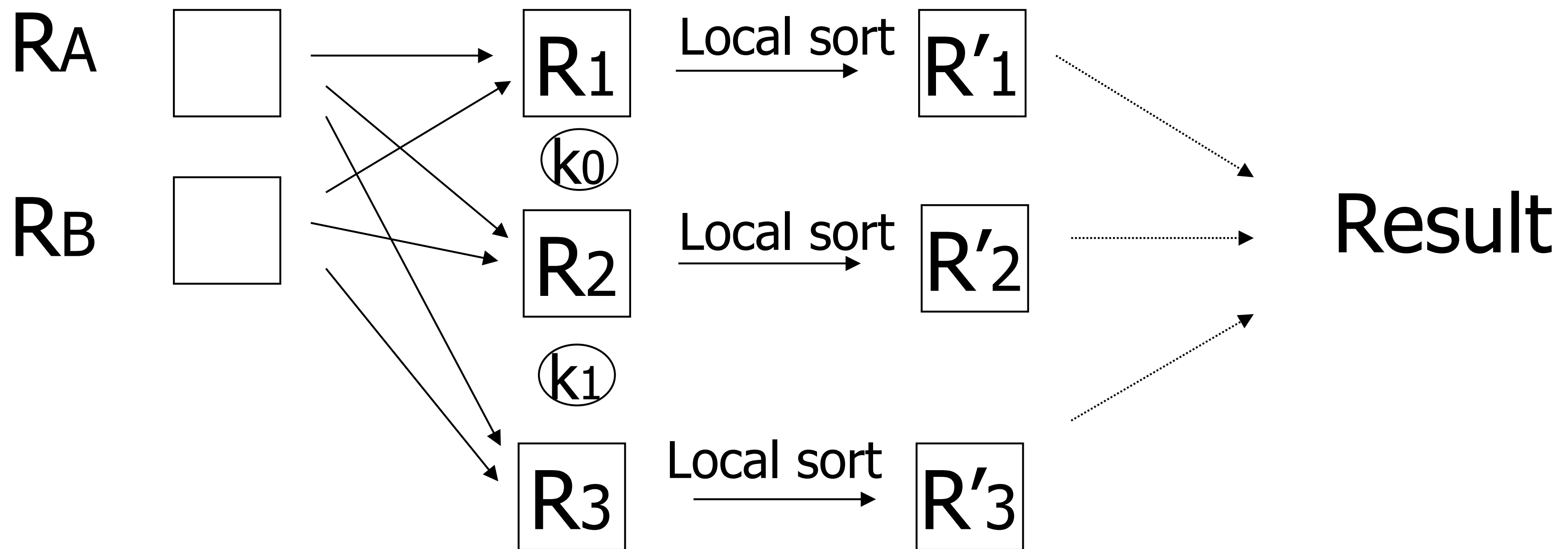
Shared memory



Range partitioning sort

- R(K,...); sort on K
- R located at one or more sites/disks
  not fragmented on K

- Algorithm:
  - (a) Range partition on K
  - (b) Basic sort



Selecting a good partition vector

7	...
52	
11	
14	

R_A

31	...
8	
15	
11	
32	
17	

R_B

10	...
12	
4	

R_C

Example

- Each site sends to coordinator:
  - Min sort key
  - Max sort key
  - Number of tuples
- Coordinator computes vector and distributes to sites

  (also decides # of sites for local sorts)

Sample scenario

Coordinator receives:

R_A: Min=5 Max=10 # = 10 tuples

R_B: Min=7 Max=17 # = 10 tuples

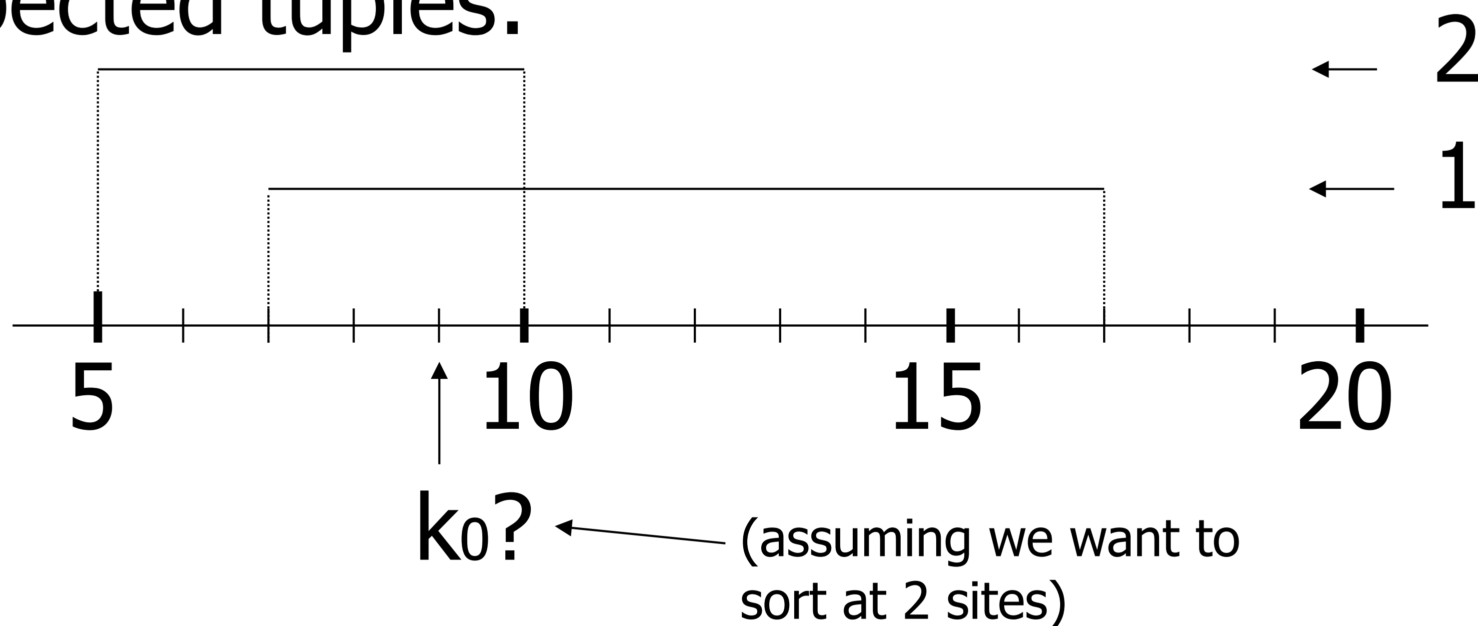
Sample scenario

Coordinator receives:

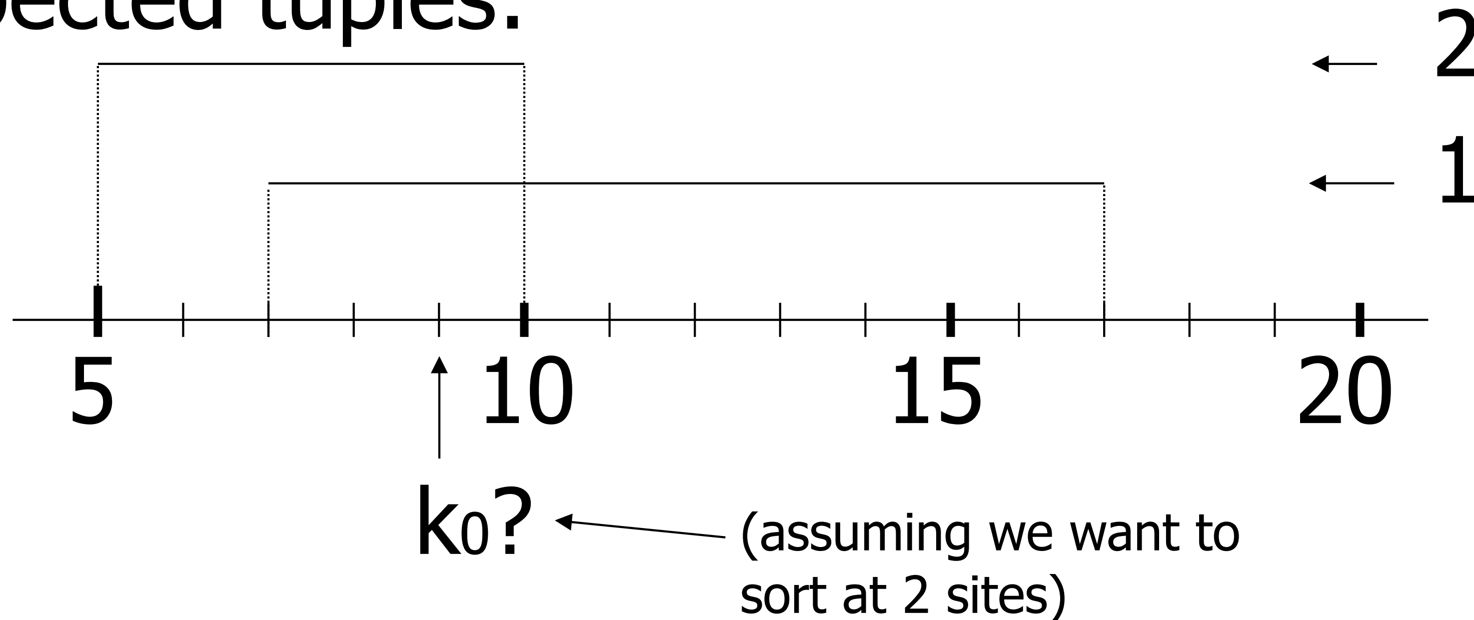
R_A: Min=5 Max=10 # = 10 tuples

R_B: Min=7 Max=17 # = 10 tuples

Expected tuples:



Expected tuples:



$$\text{Expected tuples with key} < k_0 = \frac{\text{Total tuples}}{2}$$

$2(k_0 - 5) + (k_0 - 7) = 10$

$3\ k_0 = 10 + 10 + 7 = 27$

$k_0 = 9$

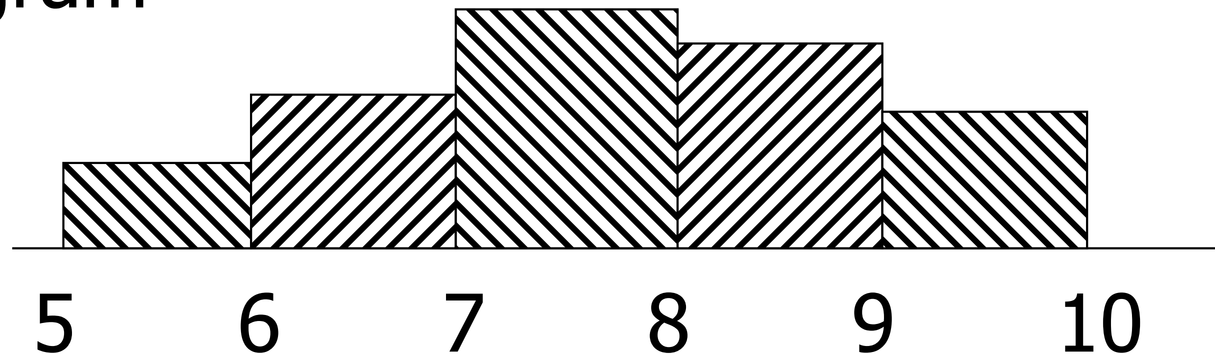
Variations

- Send more info to coordinator
  - Partition vector for local site

E.g., R_A: 3 3 3 ← # tuples

5 6 8 10 ← local vector

  - Histogram



☛ More than one round

E.g.: (1) Sites send range and # tuples

(2) Coordinator returns “preliminary” vector V_0

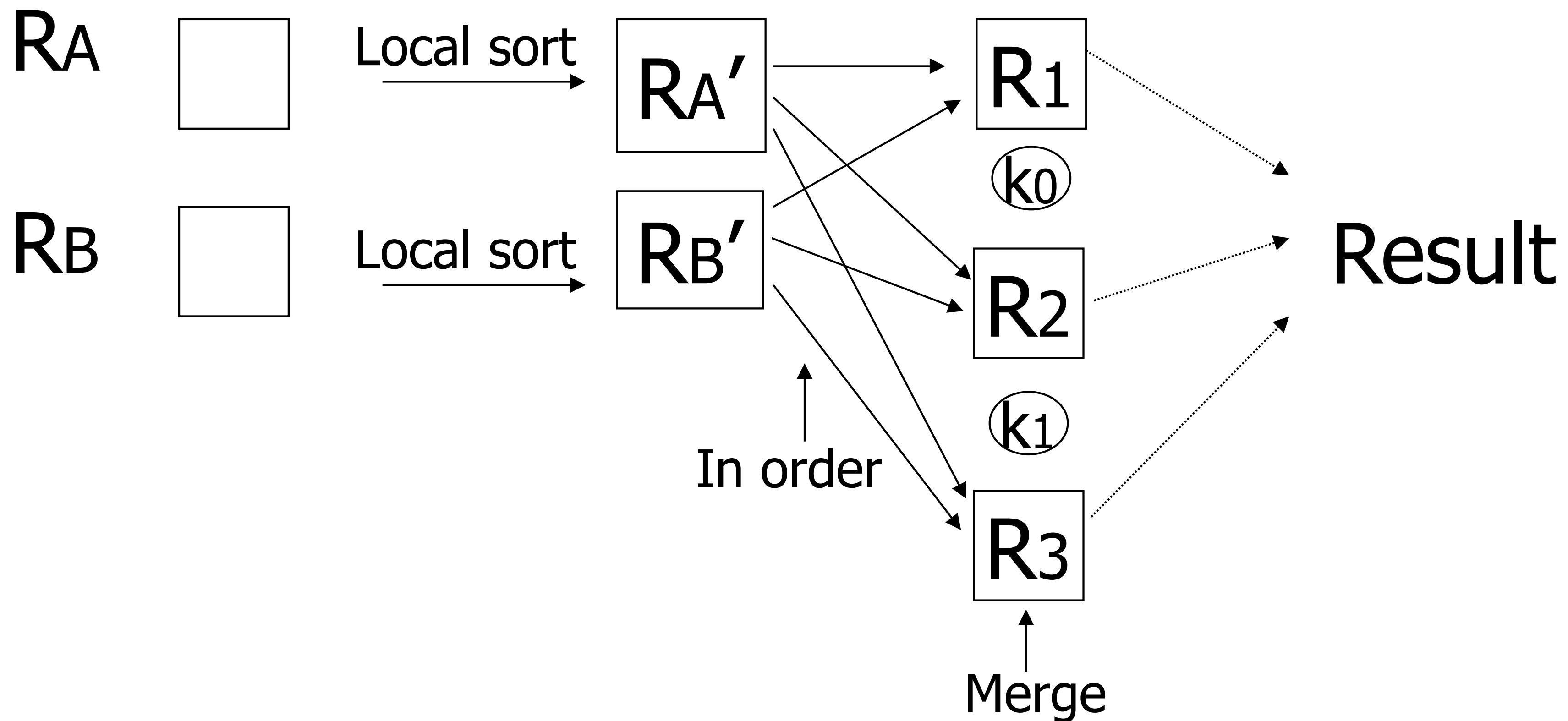
(3) Sites tell coordinator how many tuples in each V_0 range

(4) Coordinator computes final vector V_f

- Can you come up with a distributed algorithm? (no coordinator)

Parallel external sort-merge

- Same as range-partitioning sort, except sort first

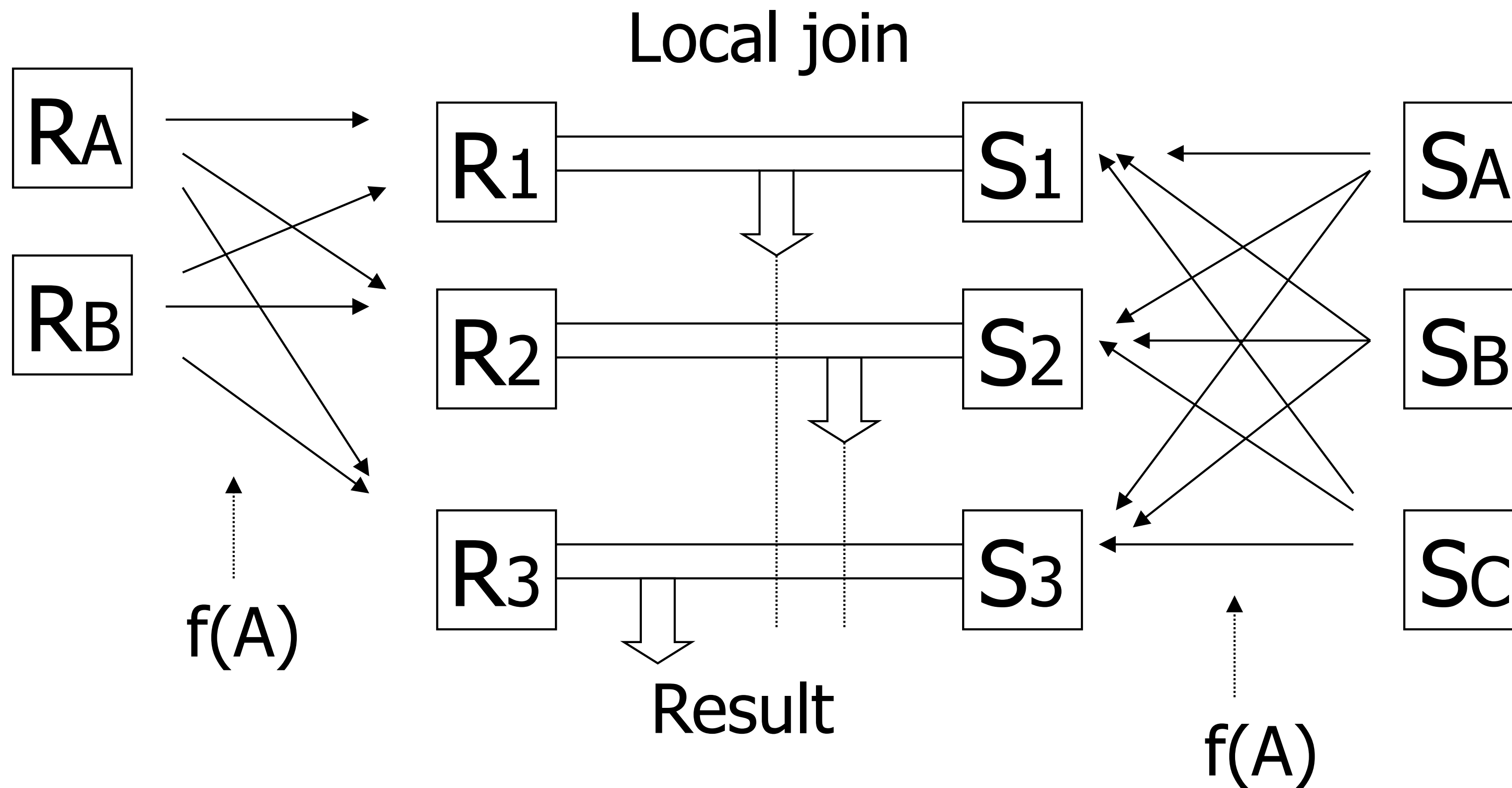


Parallel/distributed join

Input: Relations R, S
May or may not be partitioned

Output: $R \bowtie S$
Result at one or more sites

Partitioned join (equi-join)

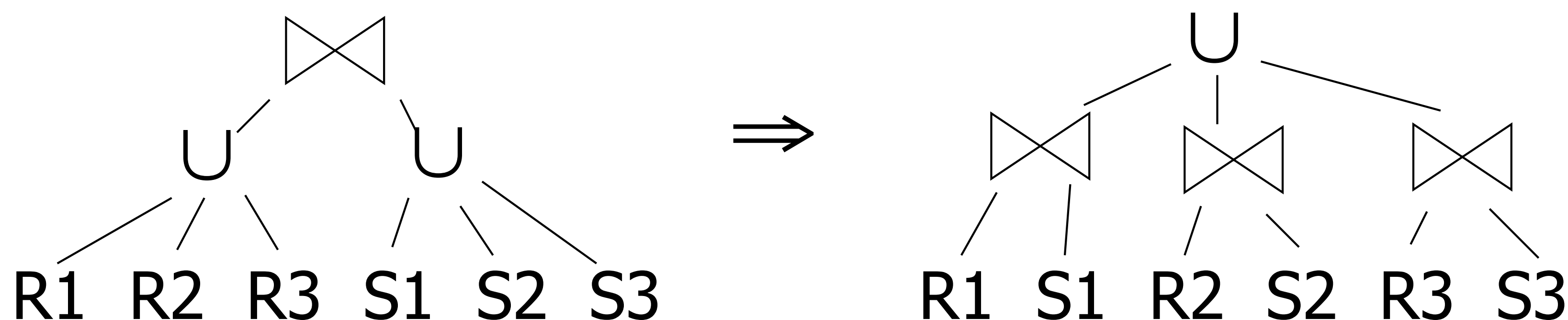


Notes

- Same partition function f is used for both R and S (applied to join attribute A)
- f can be range or hash partitioning
- Local join can be of any type (use any CS 245 optimization)
- Various scheduling options, e.g.,
  - (a) partition R; partition S; join
  - (b) partition R; build local hash table for R; partition S and hash-join

More notes

- We already know why partition-join works:



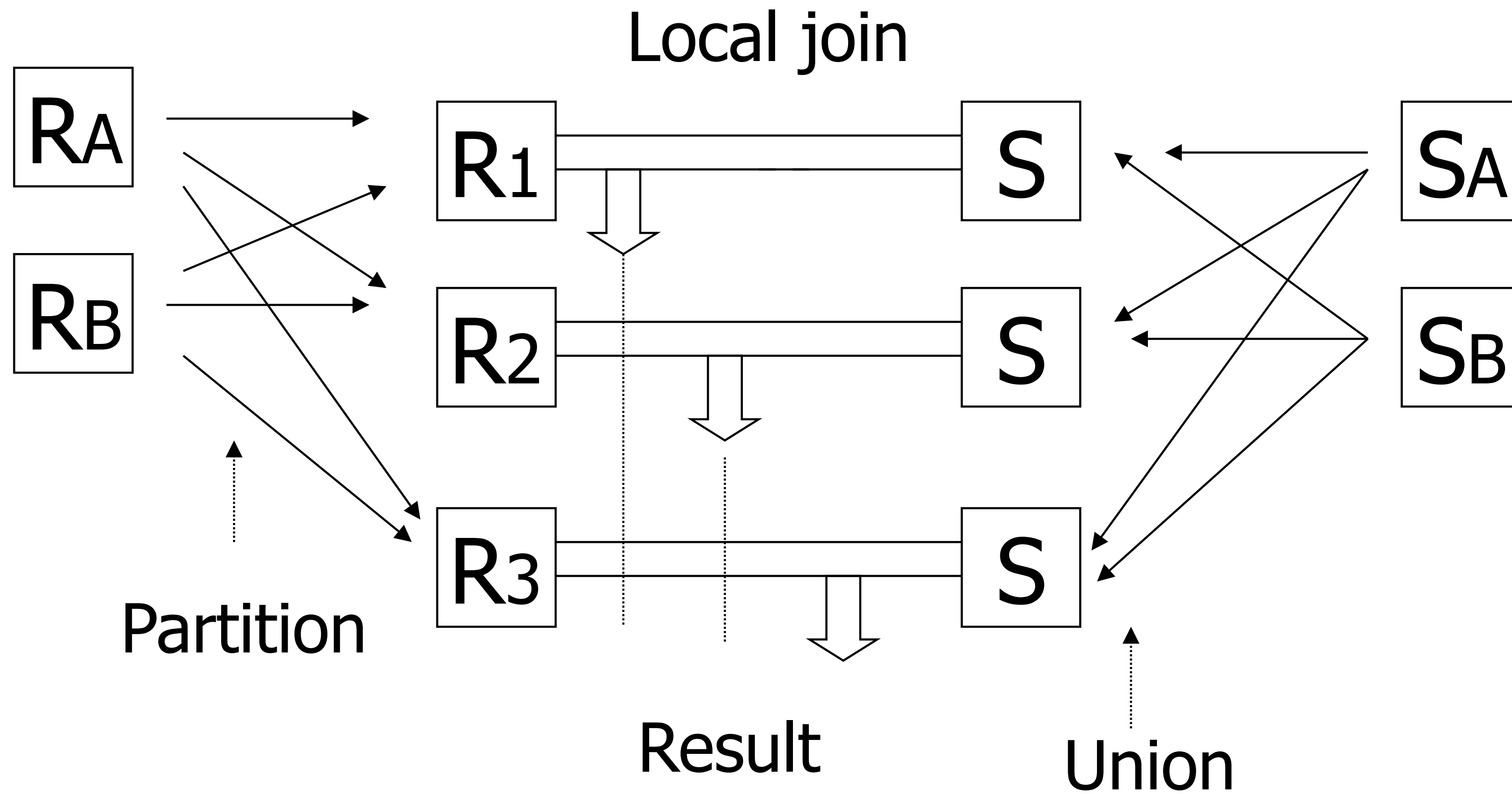
- Useful to give this type of join a name, because we may want to partition data to make partition-join possible (especially in a parallel DB system)

Even more notes

- Selecting a good partition function f is very important:
  - Number of fragments
  - Hash function
  - Partition vector

- Good partition vector
  - Goal: $| R_i | + | S_i |$ the same
  - Can use coordinator to select

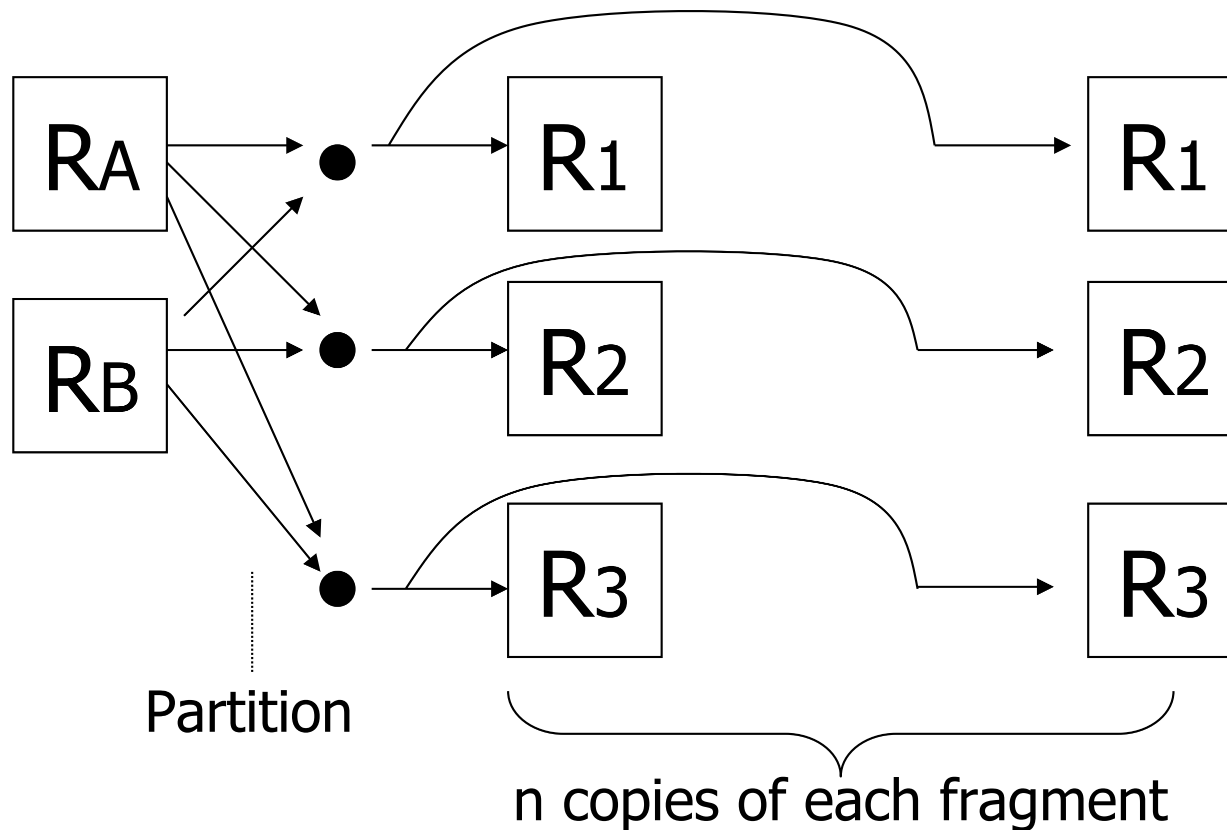
Asymmetric fragment + replicate join



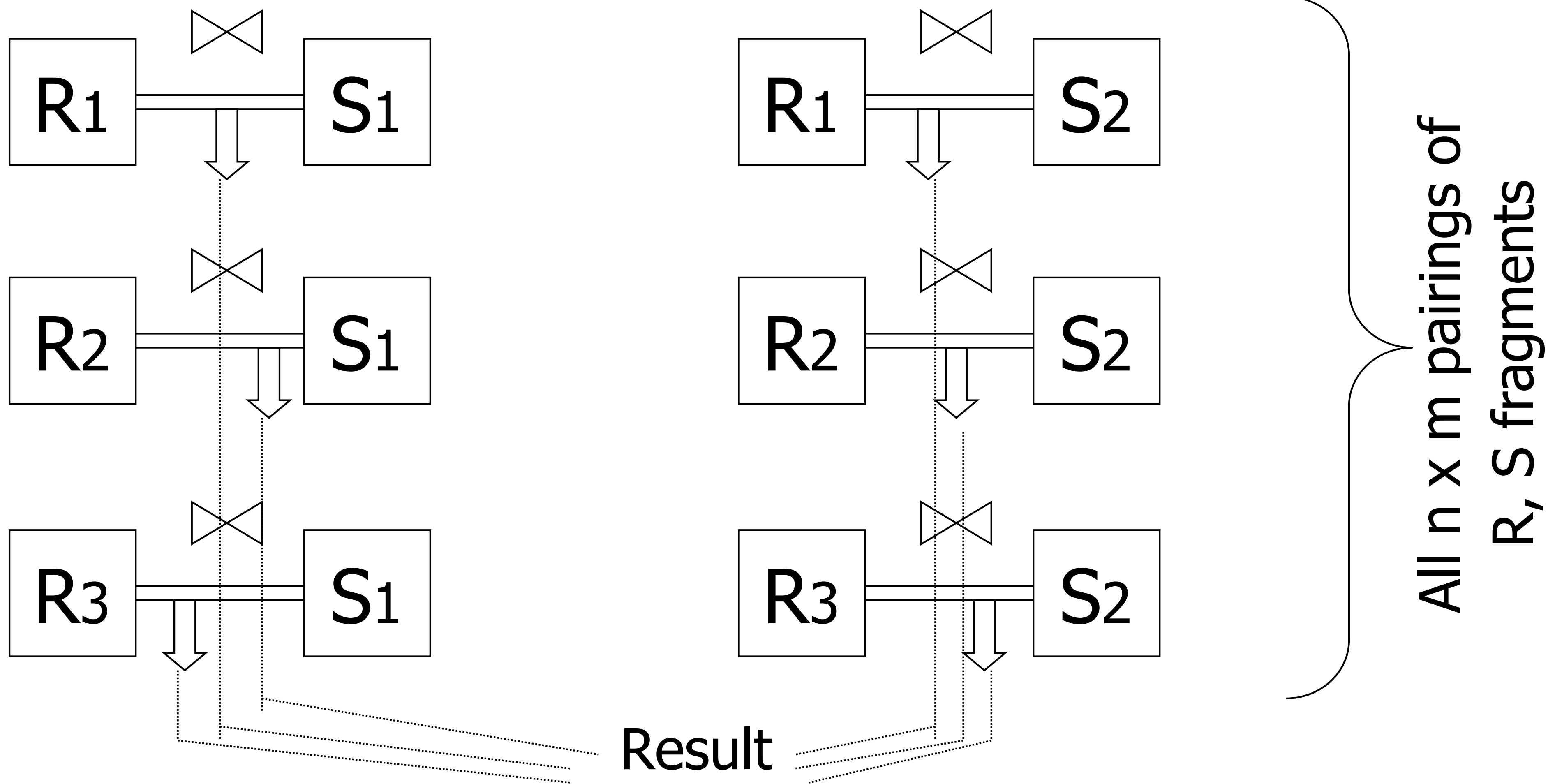
Notes:

- Can use any partition function f for R (even round robin)
- Can do any join, not just equi-join
  e.g.: $R \bowtie_{R.A < S.B} S$

General fragment and replicate join



☛ S is partitioned in similar fashion



Notes

- Asymmetric F+R join is special case of general F+R
- Asymmetric F+R may be good if S small
- Works for non-equi-joins

Semi-join

- Goal: reduce communication traffic
- $R \bowtie_A S \Rightarrow (R \ltimes_A S) \bowtie_A S$ or

  $R \bowtie_A (S \ltimes_A R)$ or

  $(R \ltimes_A S) \bowtie_A (S \ltimes_A R)$

Example: R $\bowtie$ S

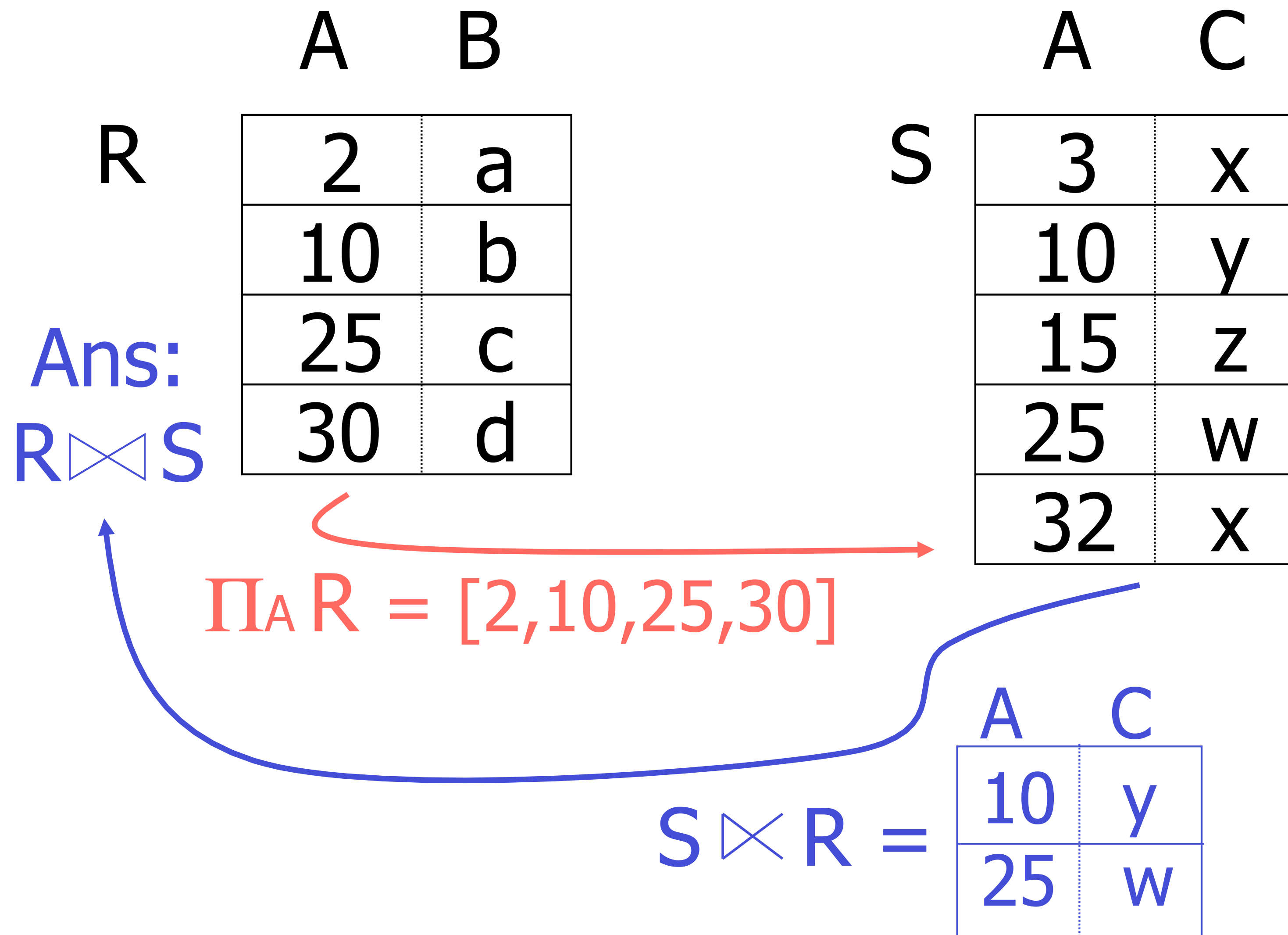
R

A	B
2	a
10	b
25	c
30	d

S

A	C
3	x
10	y
15	z
25	w
32	x

Example: R ⋈ S



R

A	B
2	a
10	b
25	c
30	d

S

A	C
3	x
10	y
15	z
25	w
32	x

Computing transmitted data in example

- with semi-join $R \bowtie (S \ltimes R)$:

$T = 4\ |A| + 2\ |A + C| + \text{result}$

- with join $R \bowtie S$:

$T = 4\ |A + B| + \text{result}$

better if |B| is large

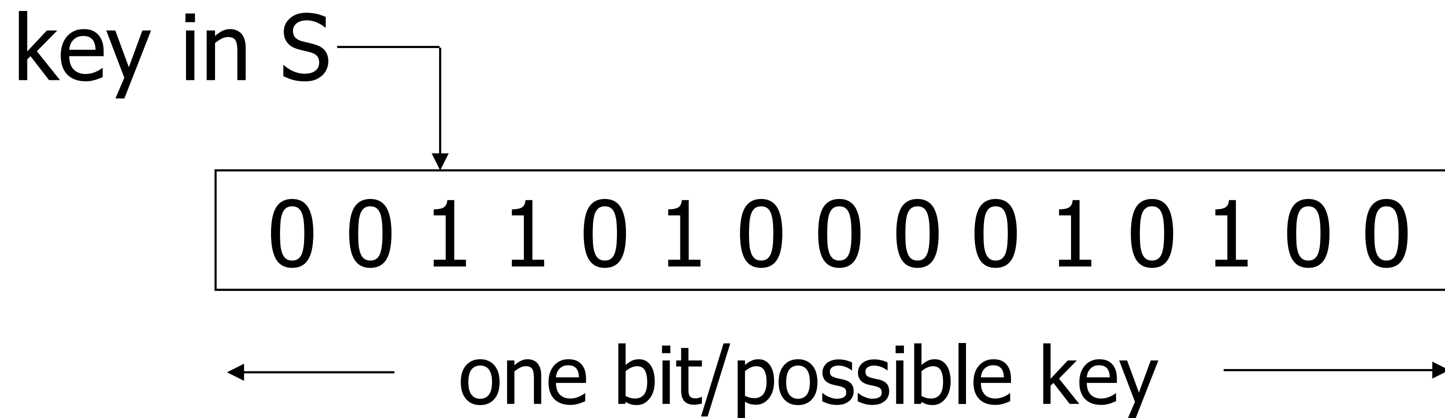
☛ In general:

- Say R is smaller relation
- $(R \ltimes_A S) \bowtie_A S$ better than $R \bowtie_A S$ if

  $\text{size}(\Pi_A S) + \text{size}(R \ltimes_A S) < \text{size}(R)$

- Similar comparisons for other semi-joins
- Remember: only taking into account transmission cost

- Trick:
  Encode $\Pi_A S$ (or $\Pi_A R$) as a <u>bit vector</u>



Three way joins with semi-joins

Goal: R $\bowtie$ S $\bowtie$ T

Three way joins with semi-joins

Goal: $R \bowtie S \bowtie T$

Option 1: $R' \bowtie S' \bowtie T$

where $R' = R \ltimes S$; $S' = S \ltimes T$

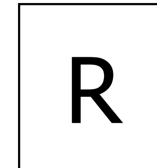
Option 2: $R'' \bowtie S' \bowtie T$

where $R'' = R \ltimes S'$; $S' = S \ltimes T$

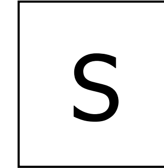
- Many options
- Number of semi-join options is exponential in # of relations in join

Privacy-preserving join

- Site 1 has R(A,B)
- Site 2 has S(A,C)



site 1



site 2

- Want to compute R $\bowtie$ S
- Site 1 should NOT discover any S info not in the join
- Site 2 should NOT discover any R info not in the join

Semi-join does not work

- If site 1 sends Π_A R to site 2, site 2 learns all keys of R

R A	B
a1	b1
a2	b2
a3	b3
a4	b4

site 1

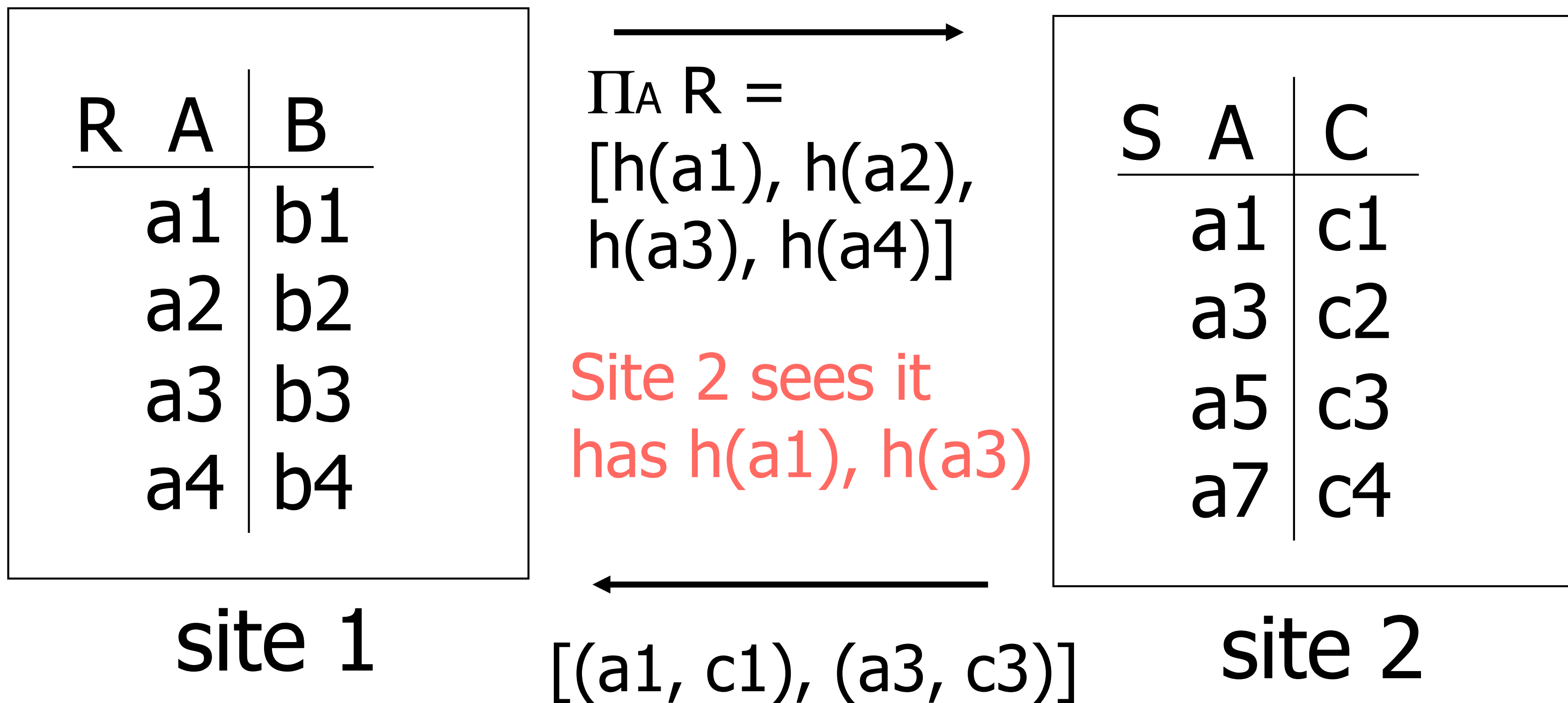
Π_A R = [a1, a2, a3, a4] →

S A	C
a1	c1
a3	c2
a5	c3
a7	c4

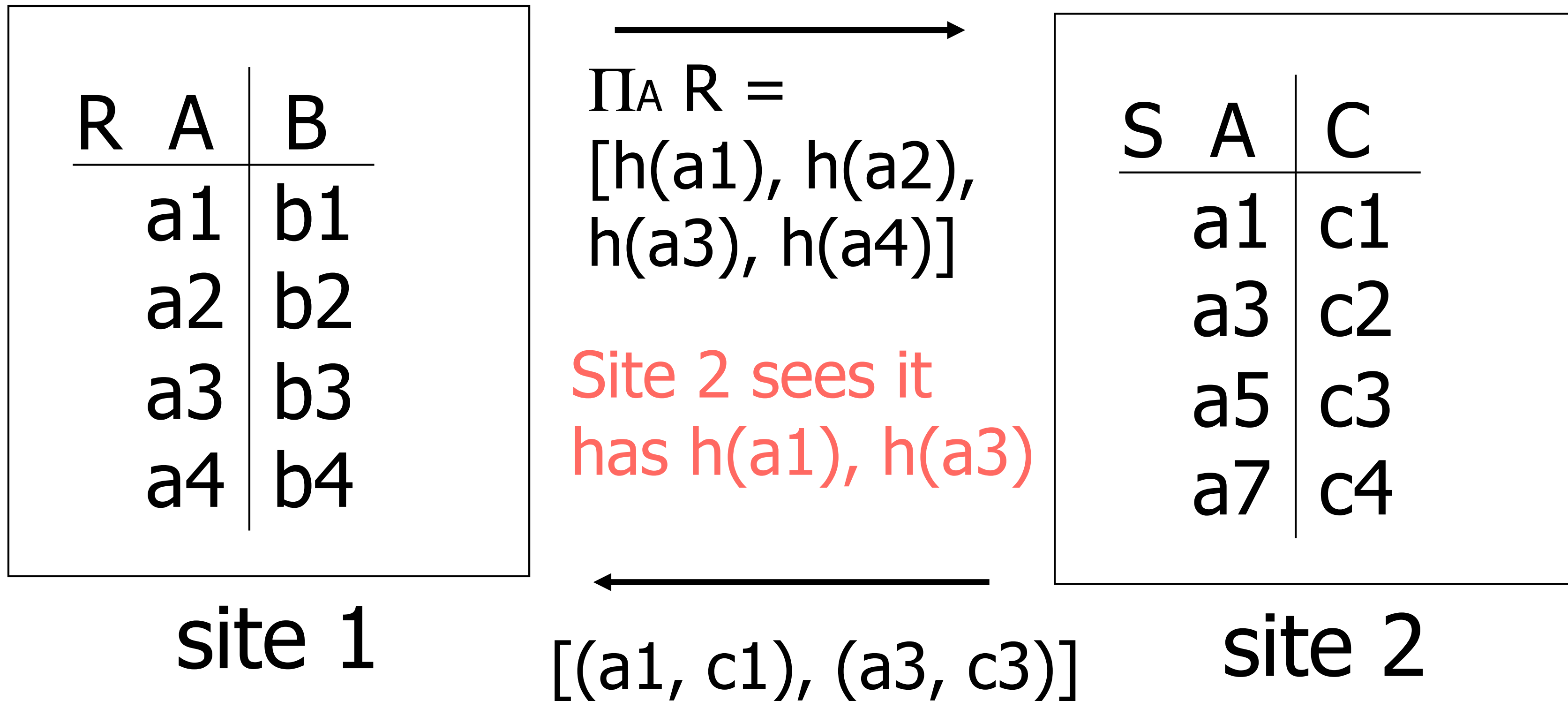
site 2

Fix: send hashed keys

- Site 1 hashes each value of A before sending
- Site 2 hashes (same function) its own A values to see what tuples match



What is problem?



- Dictionary attack
  - Site 2 takes all keys a1, a2, a3, ... and checks if h(a1), h(a2), h(a3), ... matches what site 1 sent
  - Site 1 will never notice

Adversary model

- Honest but curious
  - Observes the rules of the interaction (e.g., sending incorrect keys not possible)
  - May analyze received data in hope of gaining additional information

One solution

[Agrawal+03: Information Sharing across Private Databases]

- Use commutative encryption function
  - $E_i(x)$ = x encryption using site i's private key
  - $E_1(E_2(x)) = E_2(E_1(X))$
  - Shorthand for example:
    $E_1(x)$ is $\overline{x}$
    $E_2(x)$ is $\underline{x}$
    $E_1(E_2(x))$ is $\underline{\overline{x}}$

Solution

R

A	B
a1	b1
a2	b2
a3	b3
a4	b4

site 1

→ ($\overline{a1}$, $\overline{a2}$, $\overline{a3}$, $\overline{a4}$)

← ($\underline{\overline{a1}}$, $\underline{\overline{a2}}$, $\underline{\overline{a3}}$, $\underline{\overline{a4}}$)

← ($\underline{a1}$, $\underline{a3}$, $\underline{a5}$, $\underline{a7}$)

S

A	C
a1	c1
a3	c2
a5	c3
a7	c4

site 2

↓ computes [$\underline{\overline{a1}}$, $\underline{\overline{a3}}$, $\underline{\overline{a5}}$, $\underline{\overline{a7}}$], intersects with [$\underline{\overline{a1}}$, $\underline{\overline{a2}}$, $\underline{\overline{a3}}$, $\underline{\overline{a4}}$]

→ [(a1, b1), (a3, b3)]

- Why does this solution work?
  - Could one site perform dictionary attack without the other "noticing"?

Other parallel operations

- Duplicate elimination
  - Sort first (in parallel)
    then eliminate duplicates in result
  - Partition tuples (range or hash)
    and eliminate locally
- Aggregates
  - Partition by grouping attributes;
    compute aggregate locally

Example

R_a

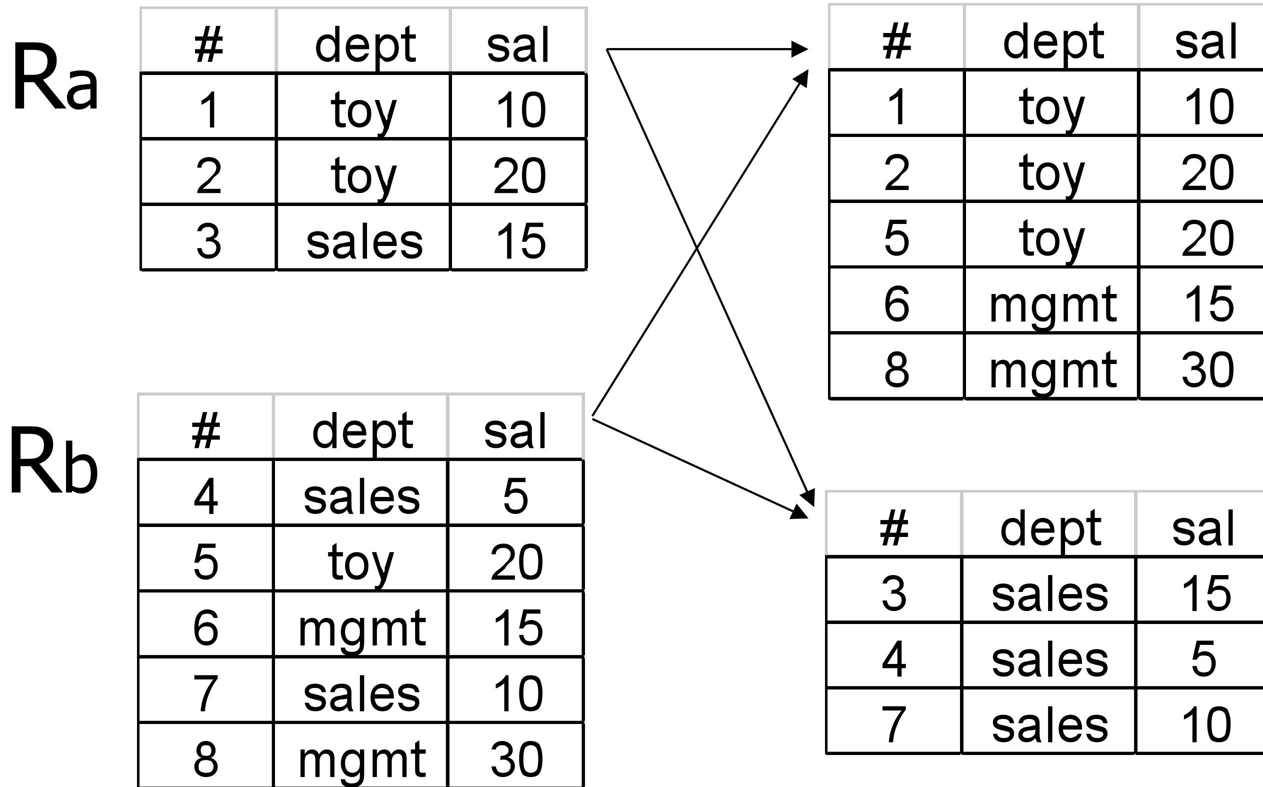
#	dept	sal
1	toy	10
2	toy	20
3	sales	15

R_b

#	dept	sal
4	sales	5
5	toy	20
6	mgmt	15
7	sales	10
8	mgmt	30

- sum(sal) group by dept

Example



R_a

#	dept	sal
1	toy	10
2	toy	20
3	sales	15

R_b

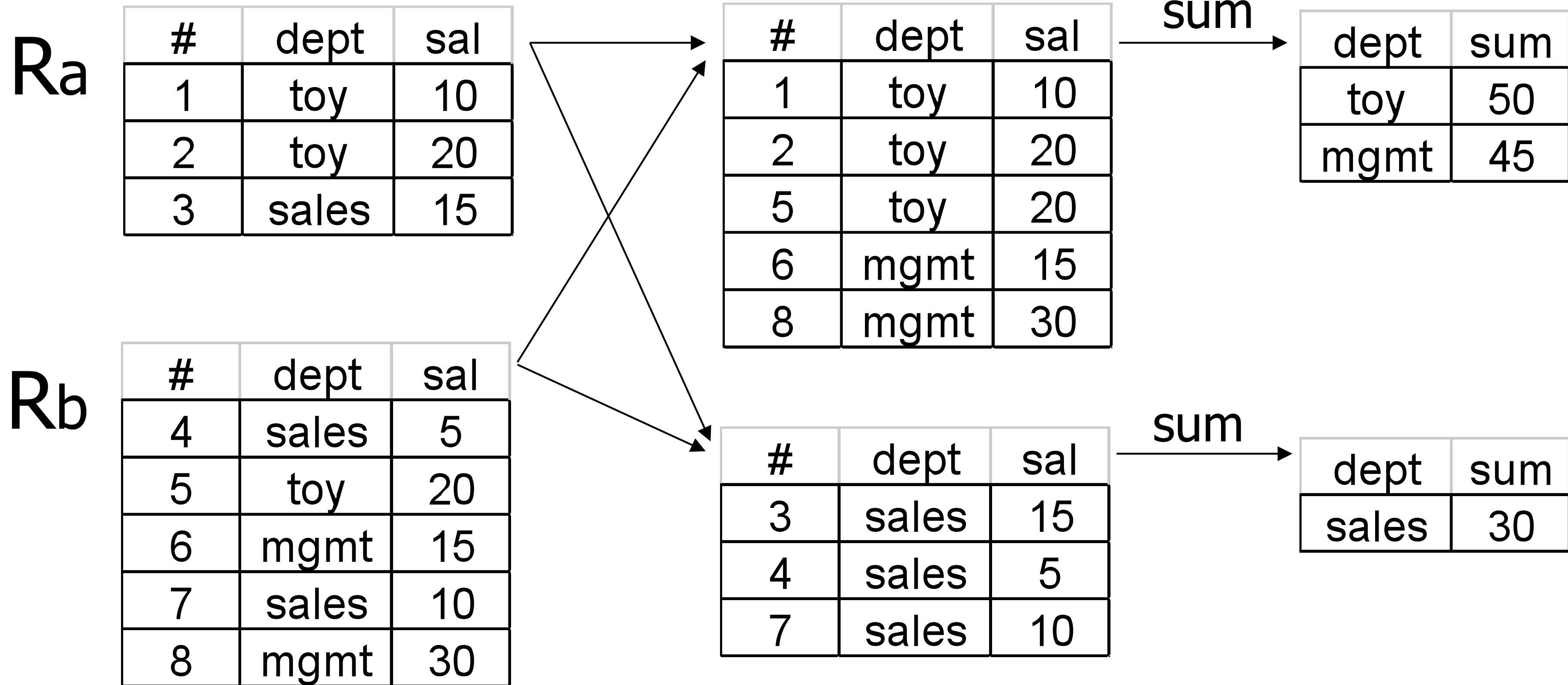
#	dept	sal
4	sales	5
5	toy	20
6	mgmt	15
7	sales	10
8	mgmt	30

#	dept	sal
1	toy	10
2	toy	20
5	toy	20
6	mgmt	15
8	mgmt	30

#	dept	sal
3	sales	15
4	sales	5
7	sales	10

- sum(sal) group by dept

Example



Ra

#	dept	sal
1	toy	10
2	toy	20
3	sales	15

Rb

#	dept	sal
4	sales	5
5	toy	20
6	mgmt	15
7	sales	10
8	mgmt	30

#	dept	sal
1	toy	10
2	toy	20
5	toy	20
6	mgmt	15
8	mgmt	30

dept	sum
toy	50
mgmt	45

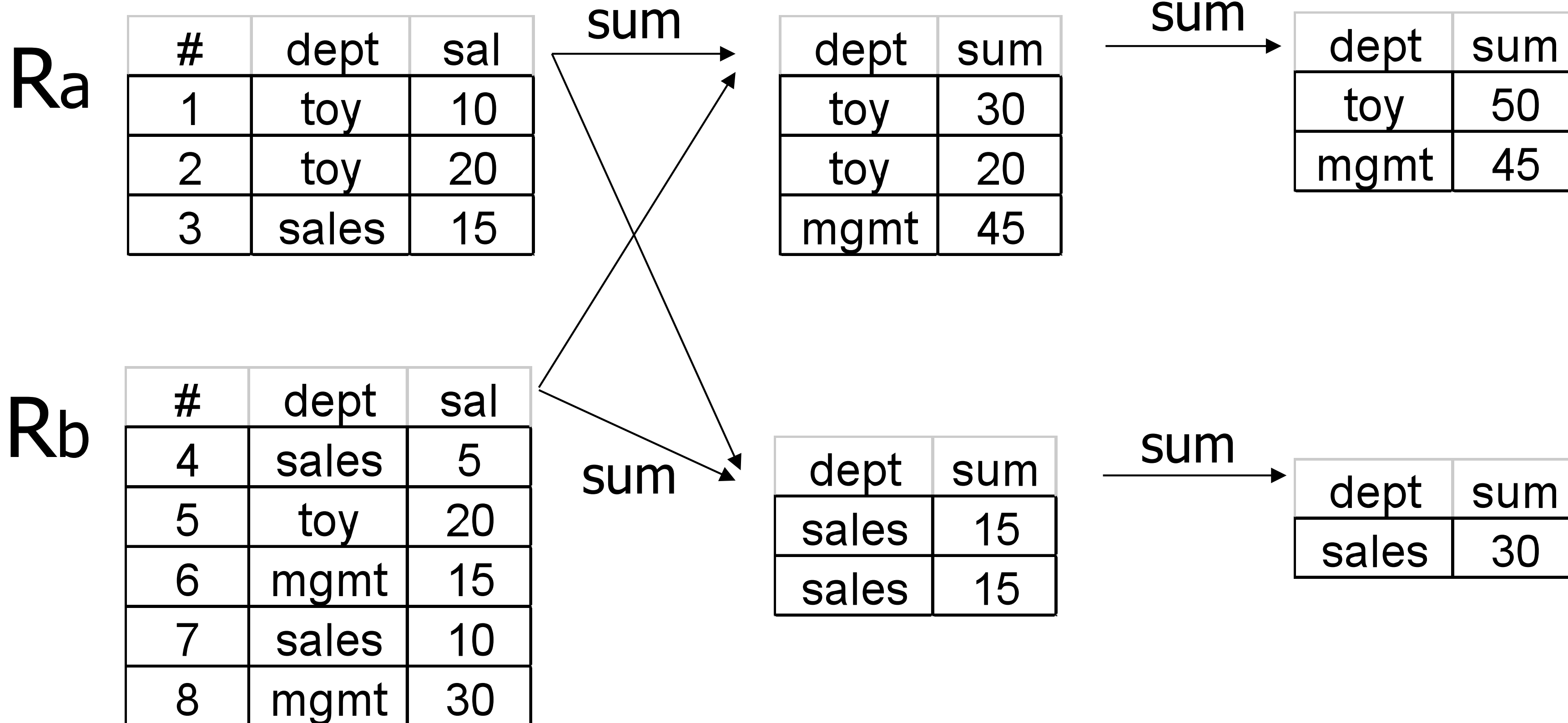
#	dept	sal
3	sales	15
4	sales	5
7	sales	10

dept	sum
sales	30

- sum(sal) group by dept

Example

less data!



Ra

#	dept	sal
1	toy	10
2	toy	20
3	sales	15

Rb

#	dept	sal
4	sales	5
5	toy	20
6	mgmt	15
7	sales	10
8	mgmt	30

dept	sum
toy	30
toy	20
mgmt	45

dept	sum
sales	15
sales	15

dept	sum
toy	50
mgmt	45

dept	sum
sales	30

- sum(sal) group by dept

Enhancements for aggregates

- Perform aggregate during partition to reduce data transmitted
- Does not work for all aggregate functions…

Which ones?

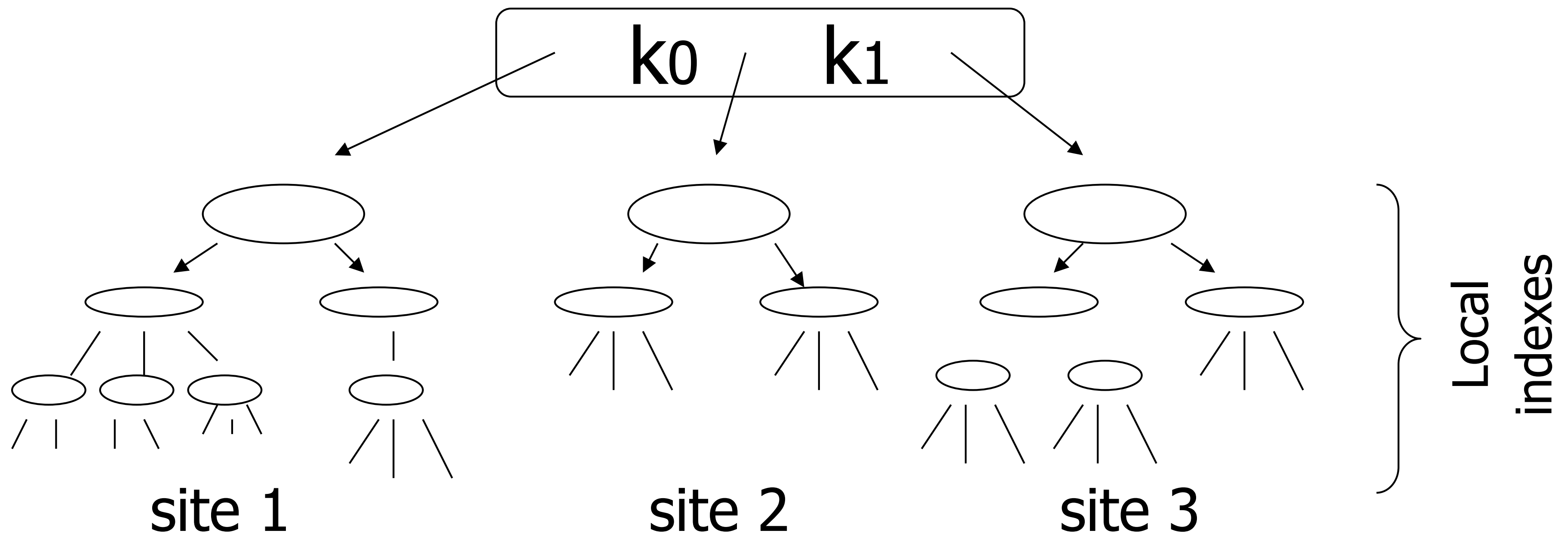
Selection

- Range or hash partition
- Straightforward

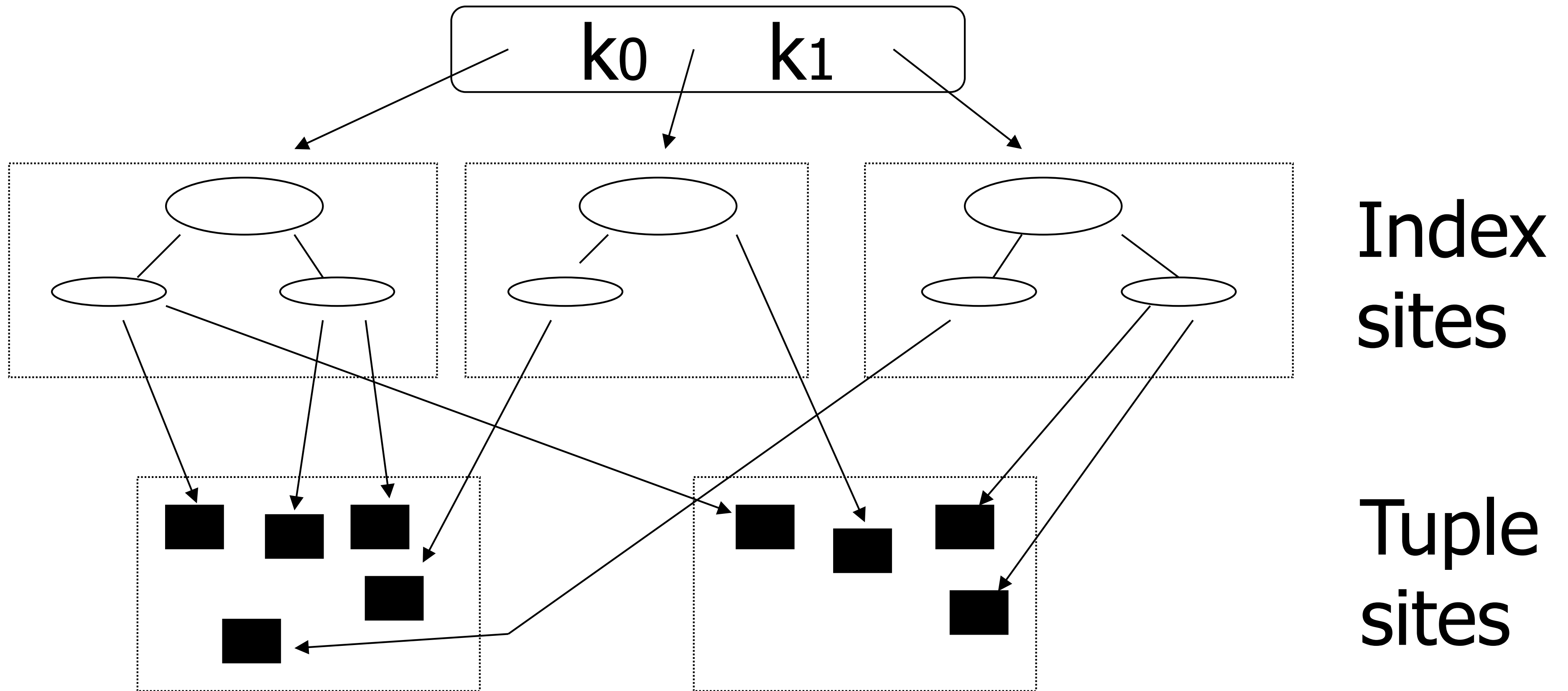
But what about indexes?

Indexing

- Can think of partition vector as root of distributed index:



Index on non-partition attribute

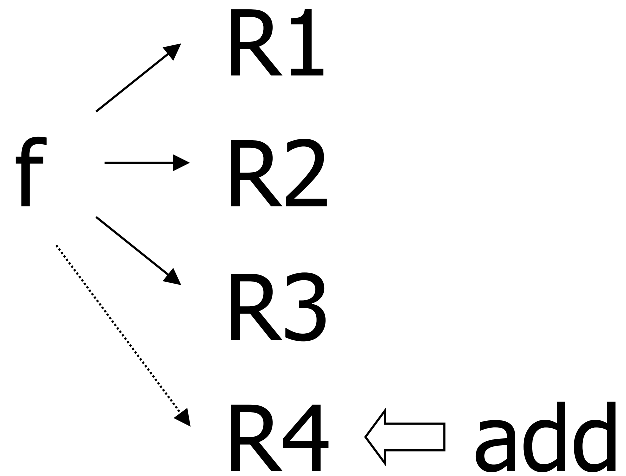


Notes

- If index is not too big, it may be better to keep whole and make copies
- If updates are frequent, can partition update work

How do we handle the split of B-tree pages?

- Extensible or linear hashing



- How do we adapt schemes?
- Where do we store directory, set of participants, …?
- Which one is better for a distributed environment?
- Can we design a hashing scheme with <u>no</u> global knowledge (P2P)?

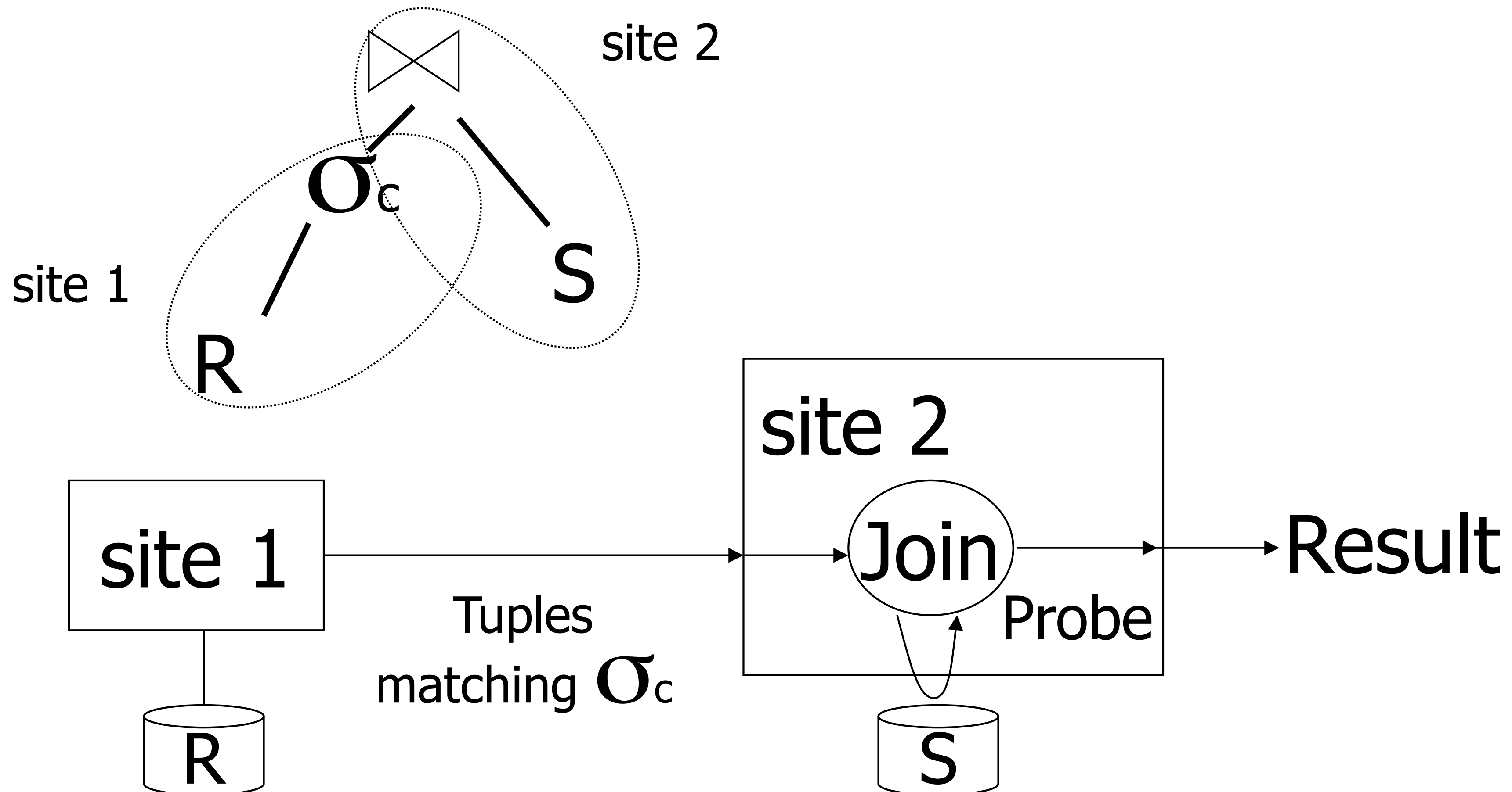
Summary: query processing

- Decomposition and localization ✓
- Optimization
  - Overview ✓
  - Tricks for joins, sort, ... ✓
  - Tricks for inter-operations parallelism
  - Strategies for optimization

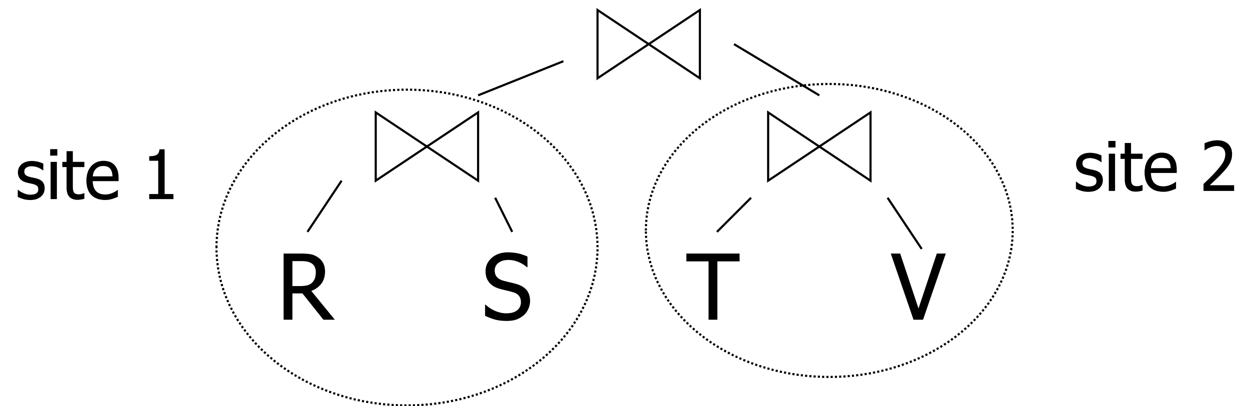
Inter-operation parallelism

- Pipelined
- Independent

Pipelined parallelism



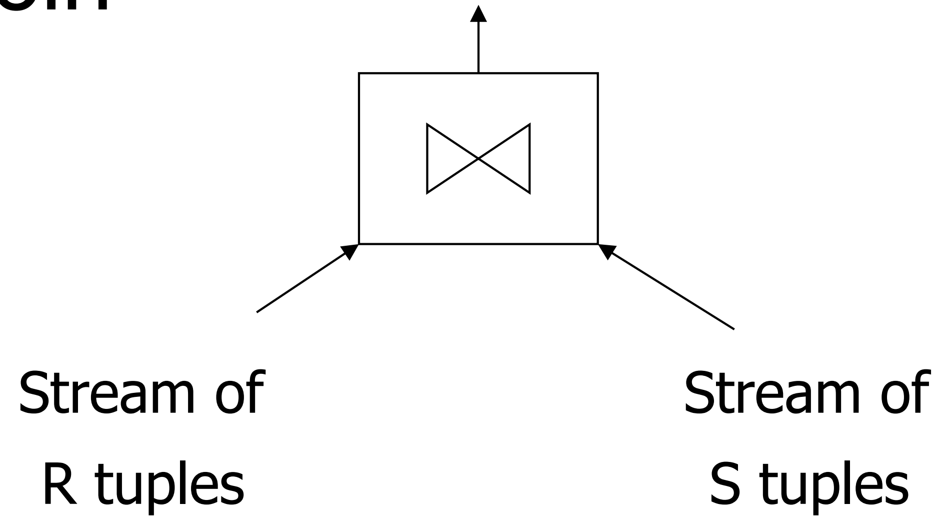
Independent parallelism



(1) temp1 ← R $\bowtie$ S; temp2 ← T $\bowtie$ V

(2) result ← temp1 $\bowtie$ temp2

- Pipelining cannot be used in all cases
  E.g., hash join



Summary

- As we consider query plans for optimization, we must consider various tricks:
  - for individual operations
  - for scheduling operations