

query processing

Query Processing

- The query processor turns user queries and data modification commands into a query plan - a sequence of operations (or algorithm) on the database from high level queries to low level commands
- **Decisions taken by the query processor**
Which of the algebraically equivalent forms of a query will lead to the most efficient algorithm?
For each algebraic operator what algorithm should we use to run the operator?
How should the operators pass data from one to the other? (eg, main memory buffers, disk buffers)

Query Processing Example

SQL Query

```
SELECT B,D  
FROM R,S  
WHERE R.A = "c" AND S.E = 2 AND R.C=S.C
```

Example

R	A	B	C
	a	1	10
	b	1	20
	c	2	10
	d	2	35
	e	3	45

S	C	D	E
	10	x	2
	20	y	2
	30	z	2
	40	x	1
	50	y	3



B	D
2	x

Query Processing Example

How do we execute the query eventually?

- Scan relations
- Compute Cartesian product
- Select tuples
- Compute projection

One idea!

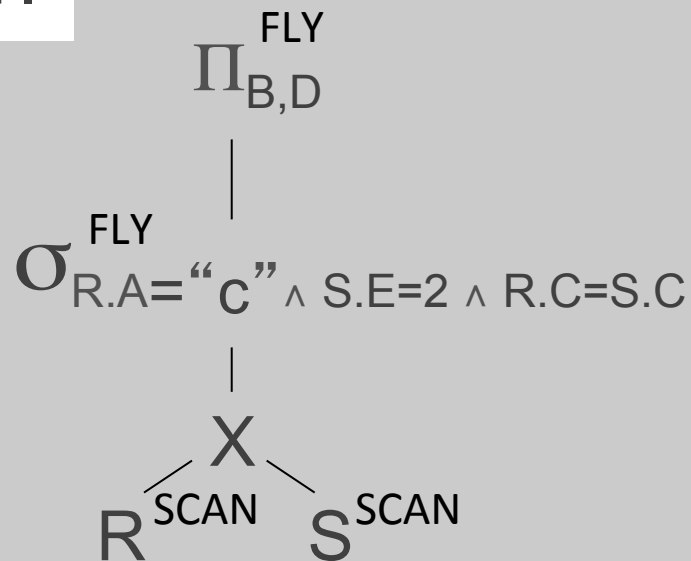
R×S	R.A	R.B	R.C	S.C	S.D	S.E
	a	1	10	10	x	2
	a	1	10	20	y	2
	...	...	...	...	...	...
	c	2	10	10	x	2
	...	...	...	...	...	...

We found one!

Describing execution plans

Using an enhanced version of the relational algebra

PLAN I



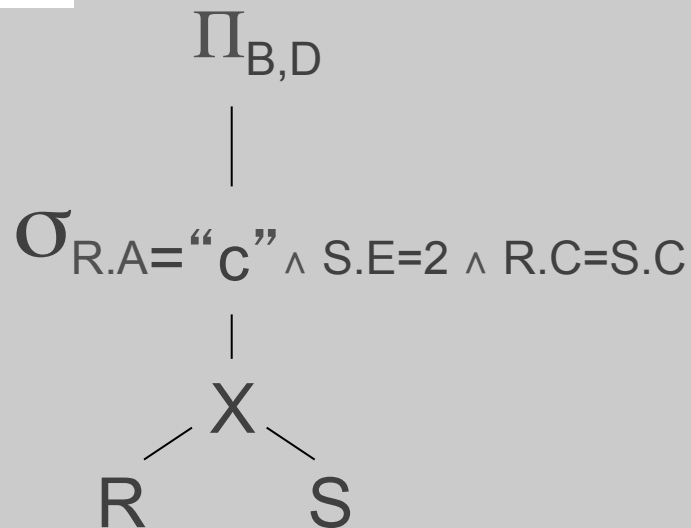
1. Scan R
2. For each tuple r of R scan S
3. For each (r,s), where s in S select and project on the fly

or: $\Pi_{B,D}^{FLY} [\sigma_{R.A="C" \wedge S.E=2 \wedge R.C=S.C}^{FLY} (R^{SCAN} \times S^{SCAN})]$

Describing execution plans

Using an enhanced version of the relational algebra

PLAN I



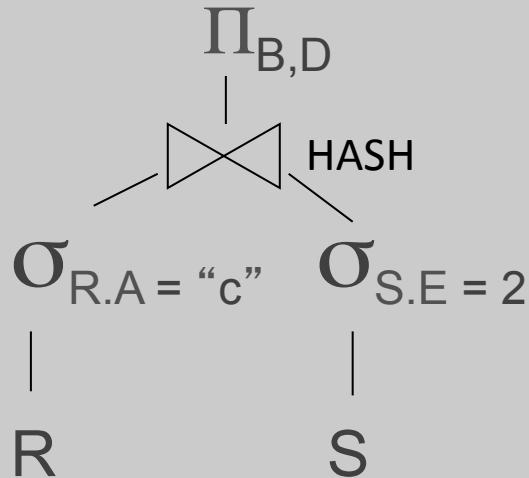
1. Scan R
2. For each tuple r of R scan S
3. For each (r,s) , where s in S
select and project on the fly

FLY & SCAN are the defaults!

Describing execution plans

Using an enhanced version of the relational algebra

PLAN II



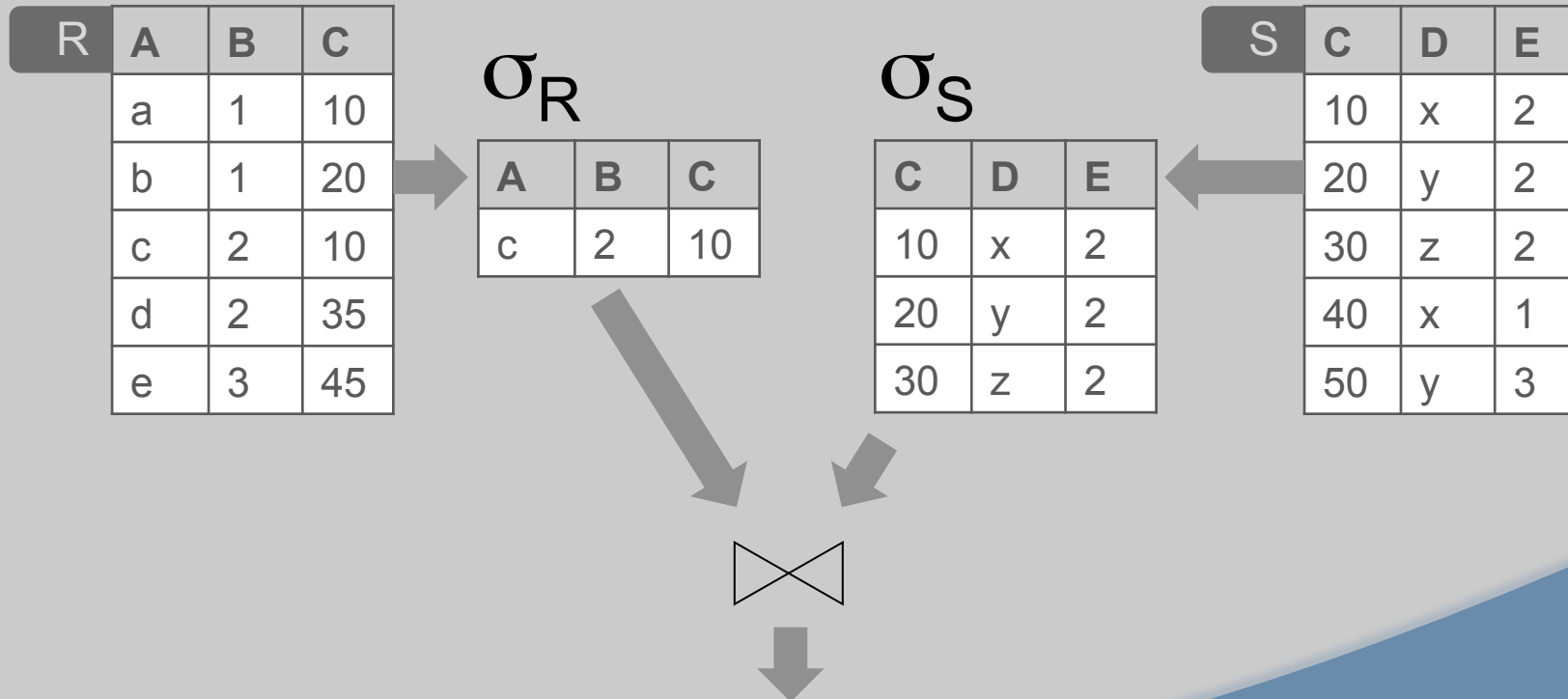
1. Scan R & S
2. Perform on the fly selections
3. Do hash join
4. Project

 : natural join

Describing execution plans

Using an enhanced version of the relational algebra

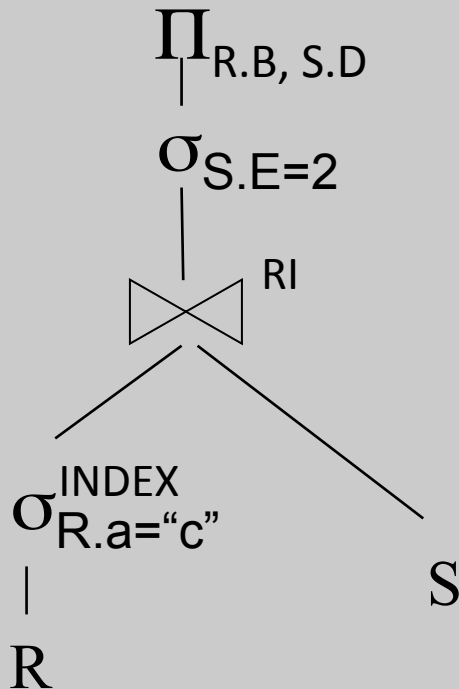
PLAN II



Describing execution plans

Using an enhanced version of the relational algebra

PLAN III



Use R.A and S.C Indexes

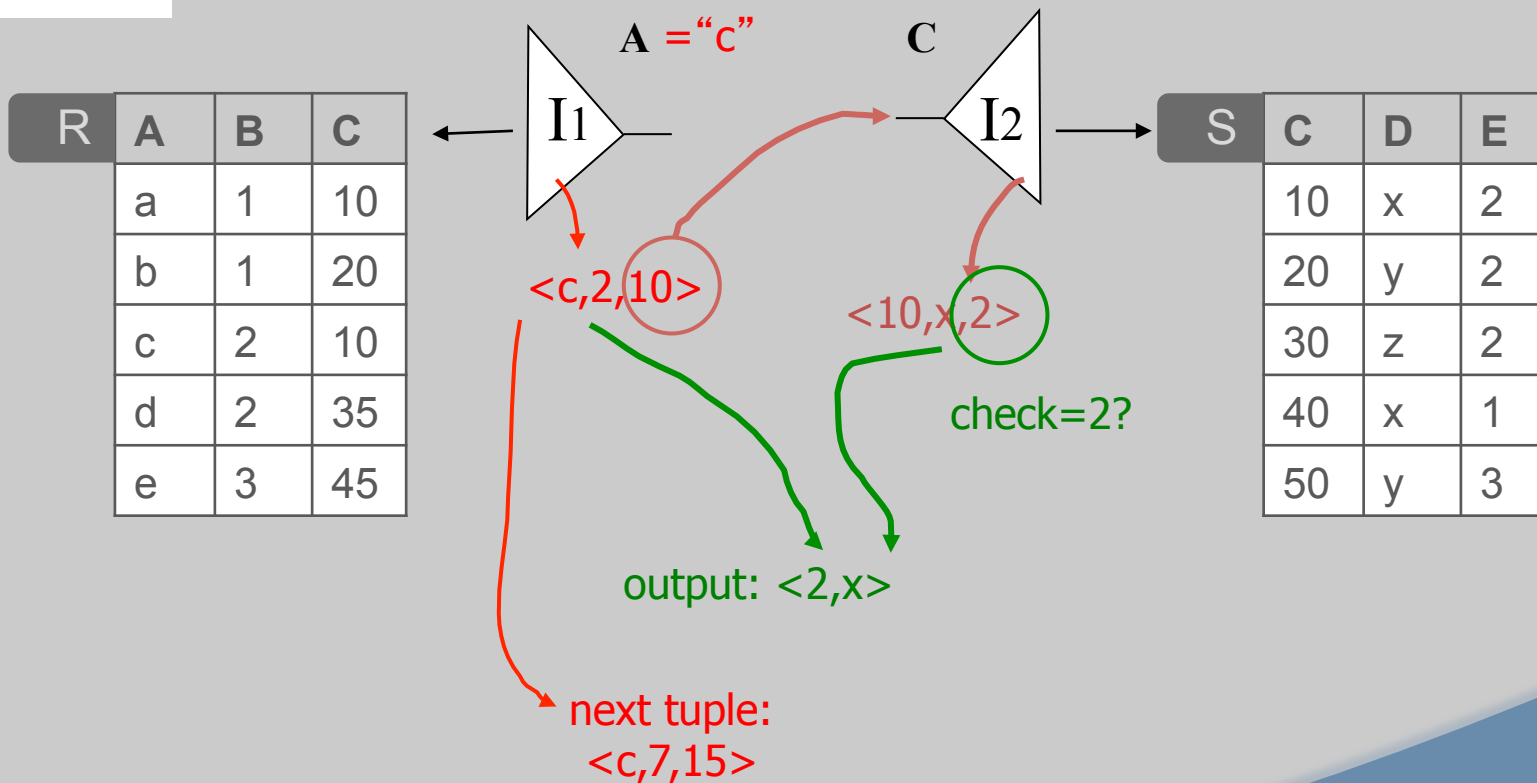
1. Use R.A index to select R tuples with R.A = "c"
2. For each R.C value found, use S.C index to find matching join tuples
3. Eliminate join tuples S.E $\neq$ 2
4. Project B,D attributes

^{RI} : right index natural join

Describing execution plans

Using an enhanced version of the relational algebra

PLAN III



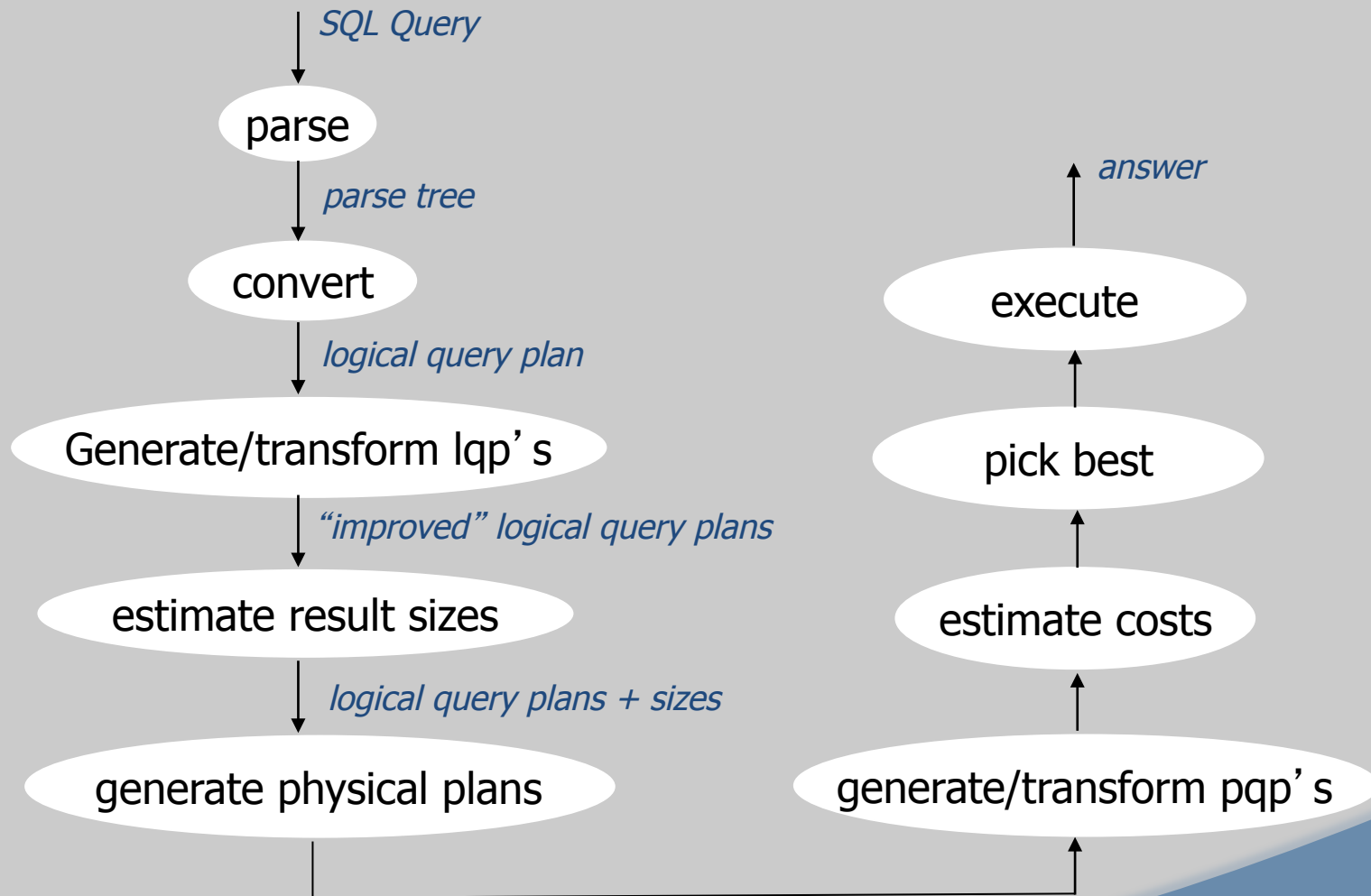
From Query To Optimal Plan

- Complex process
- Algebra-based logical and physical plans
- Transformations
- Evaluation of multiple alternatives

Issues in Query Processing and Optimization

- **Generate Plans**
Employ efficient execution primitives for computing relational algebra operations
Systematically transform expressions to achieve more efficient combinations of operators
- **Estimate Cost of Generated Plans**
Statistics
- **“Smart” Search of the Space of Possible Plans**
Always do the “good” transformations (relational algebra optimization)
Prune the space (e.g., System R)
- **Often the above steps are mixed**

The Journey of a Query



Scope of responsibility of each module is fuzzy

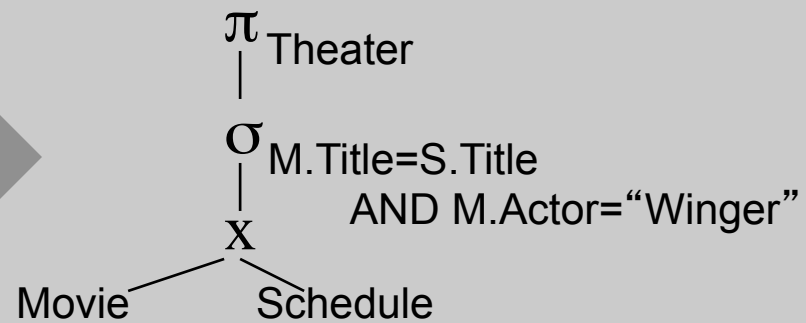
The Journey of a Query

Logical query plan generator

SELECT Theater
FROM Movie M, Schedule S
WHERE
M.Title = S.Title
AND M.Actor="Winger"

SQL query

Parse/
Convert

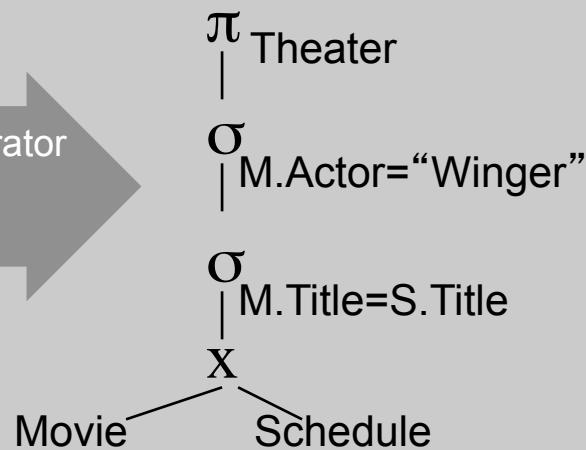


Initial logical query plan

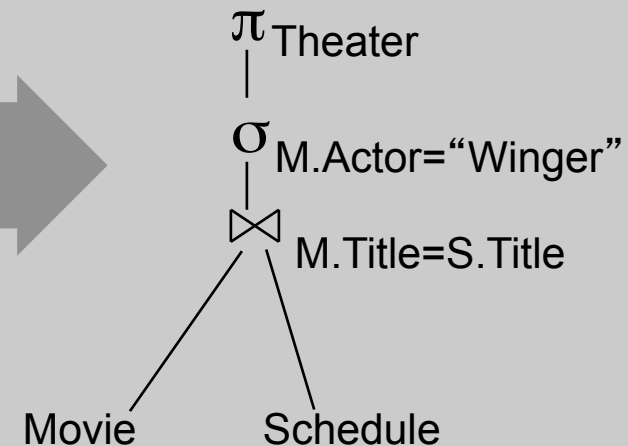
The Journey of a Query

Logical query plan generator

Logical Plan Generator
applies Algebraic
Rewriting



Another logical query plan

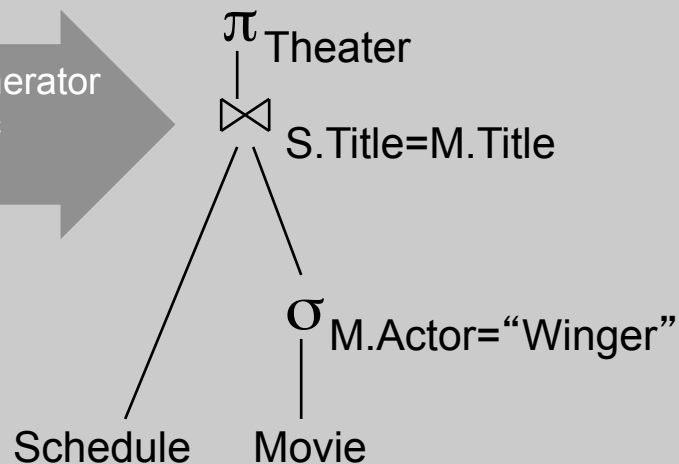


Another logical query plan

The Journey of a Query

Logical query plan generator

Logical Plan Generator
applies Algebraic
Rewriting



Another logical query plan

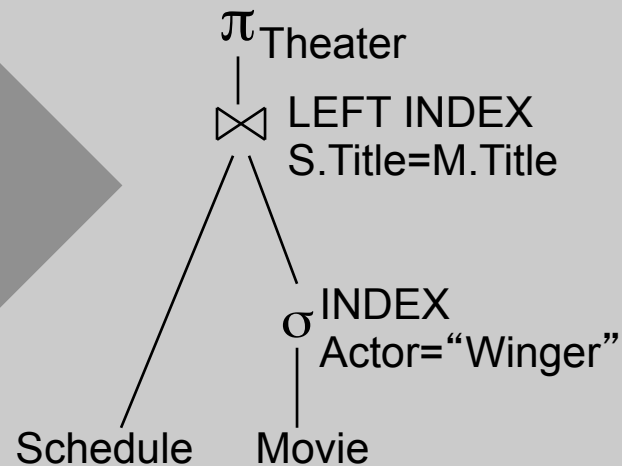
Summary of Logical Plan Generator:

- 4 logical query plans created
- Algebraic rewritings were used for producing the candidate logical query plans
- The last one is the winner (at least, cannot be a big loser)
- In general, multiple logical plans may “win” eventually

The Journey of a Query

Physical query plan generator

Physical Plan
Generator chooses
execution primitives
and data passing



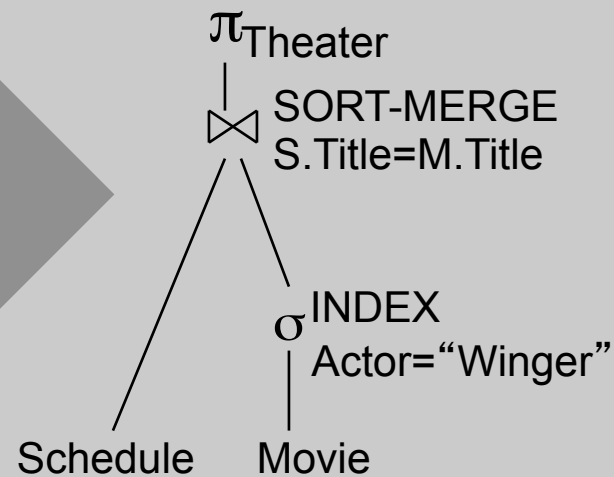
Index on Actor and
Title, unsorted tables,
Table size >> Memory size

1st physical query plan

The Journey of a Query

Physical query plan generator

Physical Plan
Generator chooses
execution primitives
and data passing



Index on Actor, Table
Schedule sorted on Schedule

2nd physical query plan

The Journey of a Query

Physical query plan generator

Summary of
Physical Plan Generator:

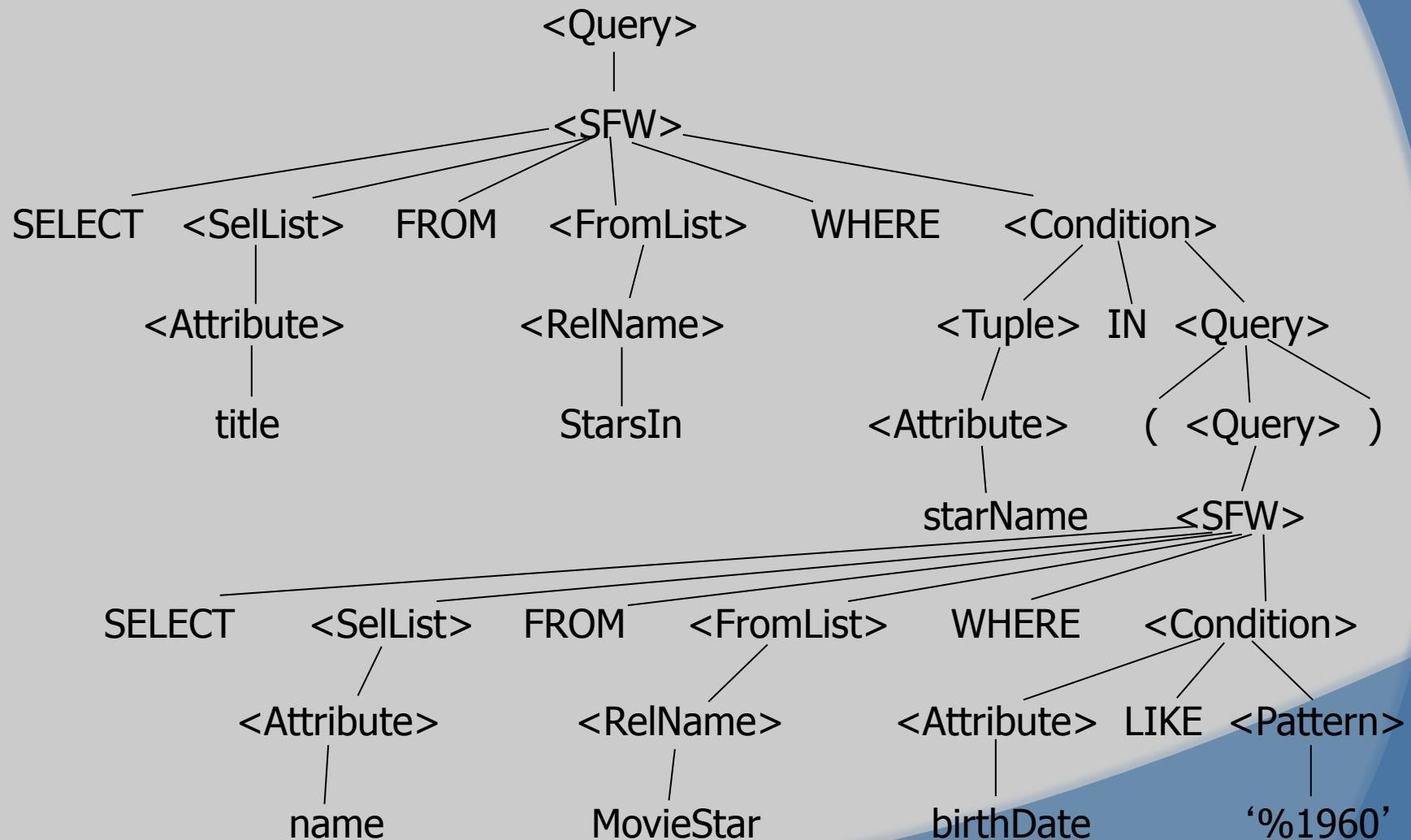
- 2 logical query plans created in this case)
- In general, more than one different logical plans may be generated by choosing different primitives

Example: Nested SQL Query

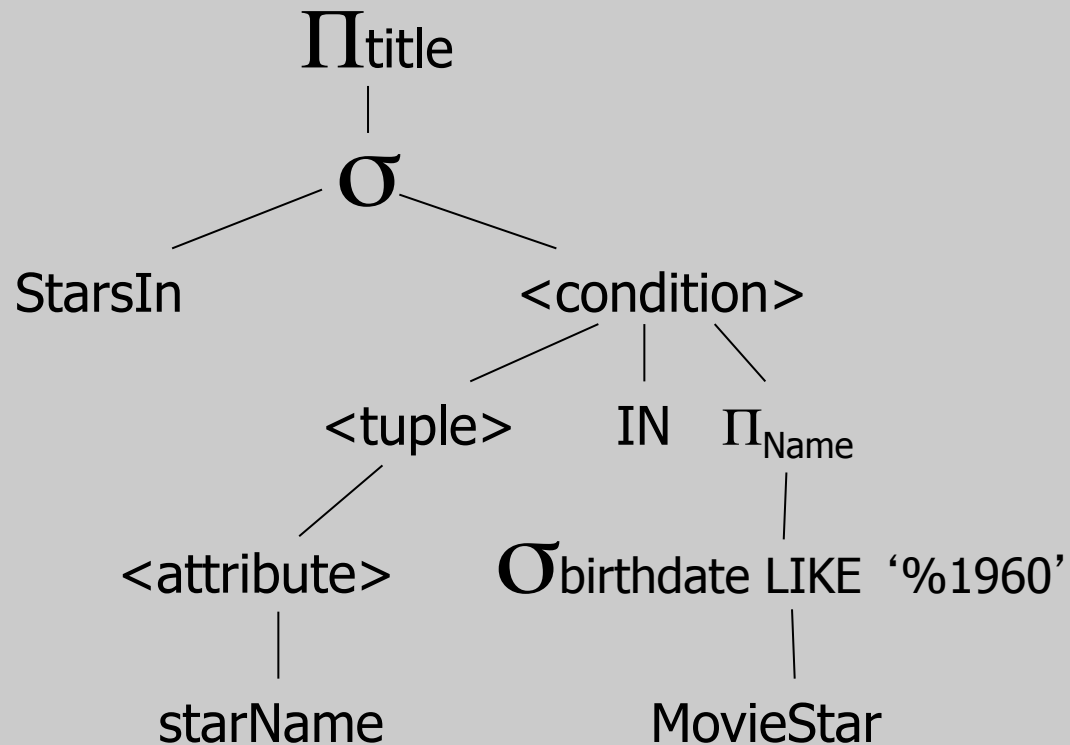
```
SELECT title  
FROM StarsIn  
WHERE starName IN (  
    SELECT name  
    FROM MovieStar  
    WHERE birthdate LIKE '%1960' )
```

(Find the movies with stars born in 1960)

Example: Parse Tree

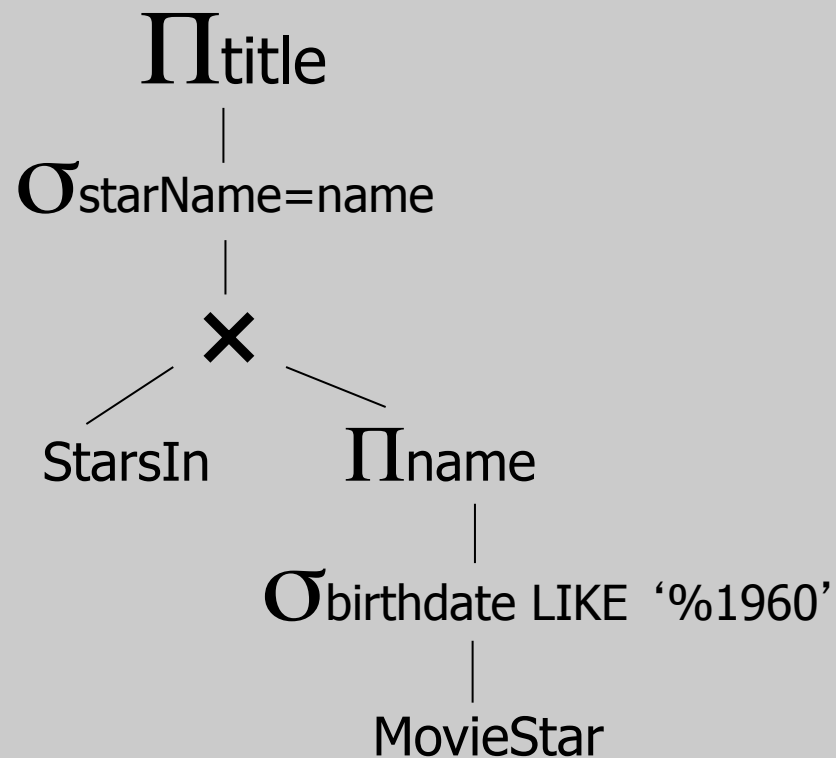


Example: Generating Rel. Algebra



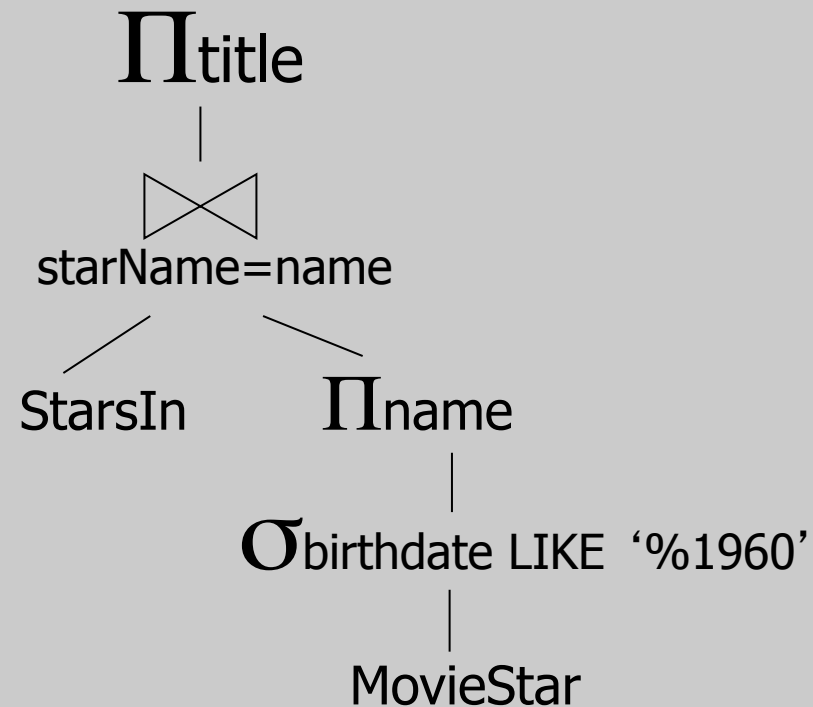
An expression using a two-argument σ , midway between a parse tree and relational algebra

Example: Logical Query Plan



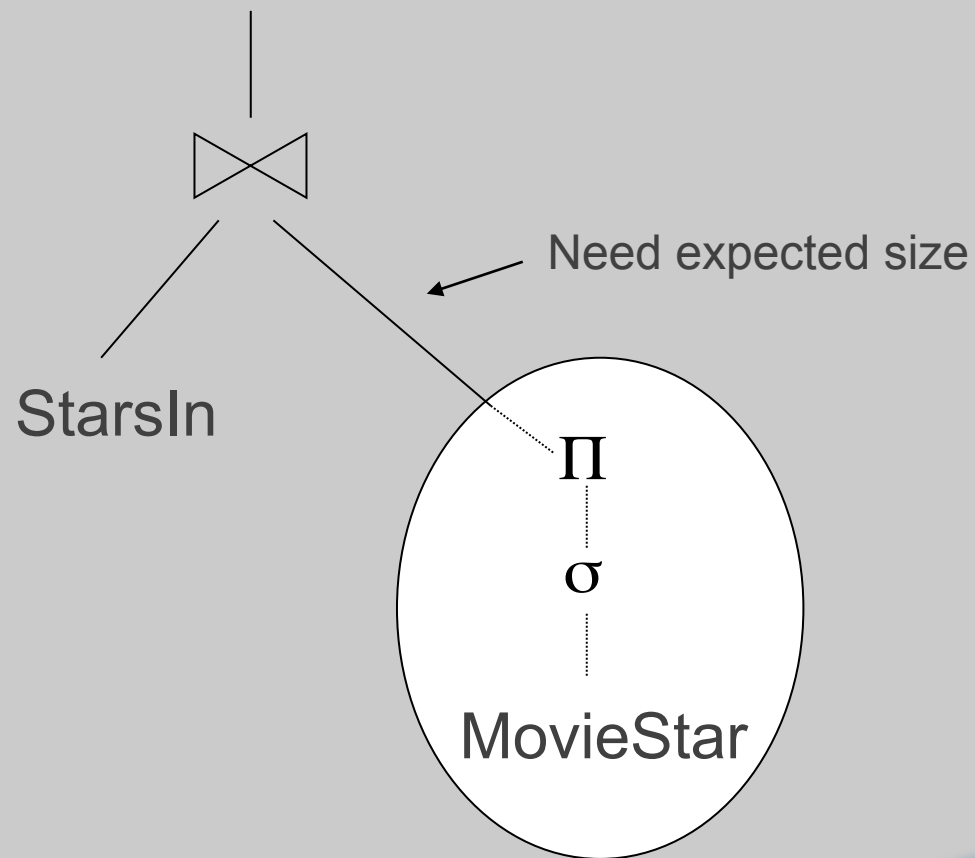
May consider “IN” elimination as a rewriting in the logical plan generator or may consider it a task of the converter

Example: Improved Logical Plan

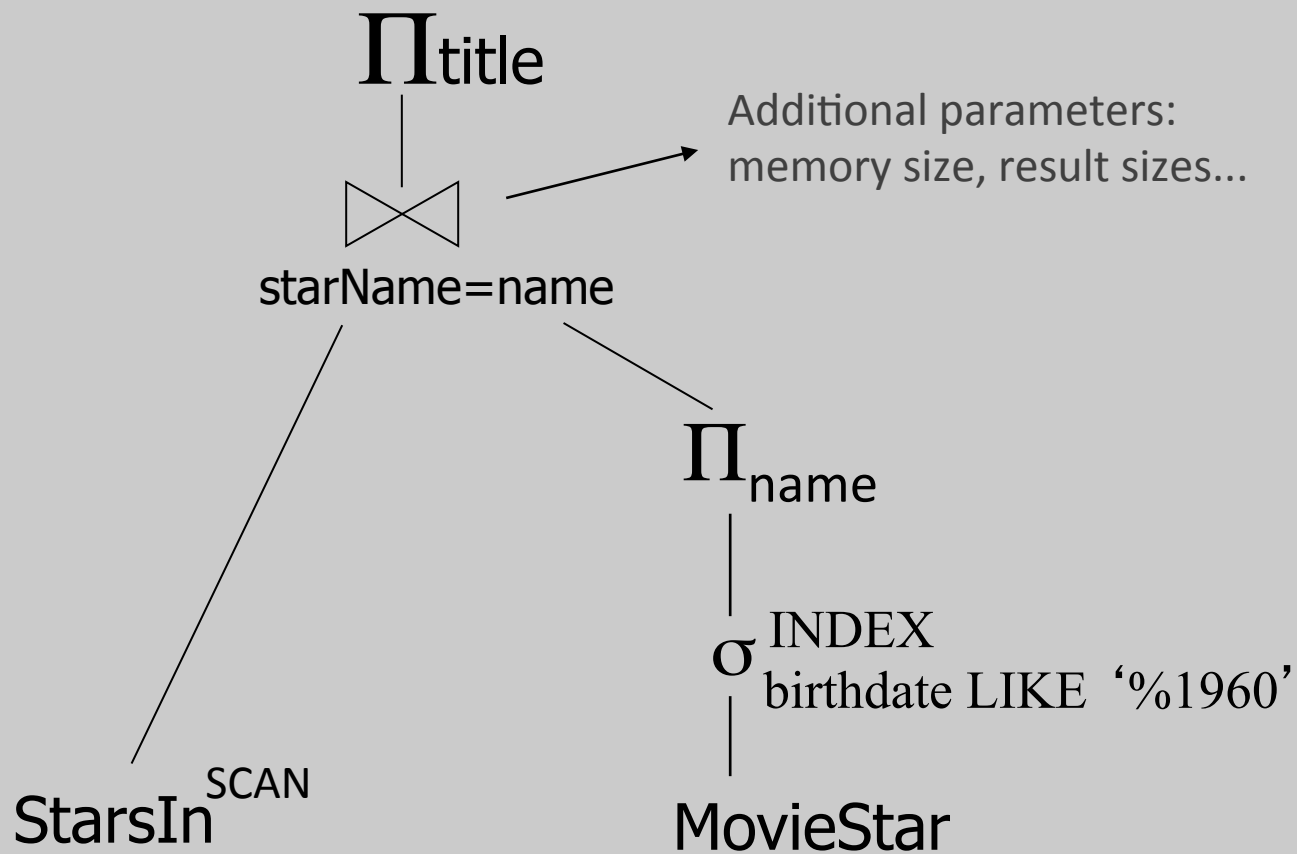


Example: Size estimation

Sizes are important for selecting physical query plans



Example: One Physical Plan



Relational Algebra Optimization

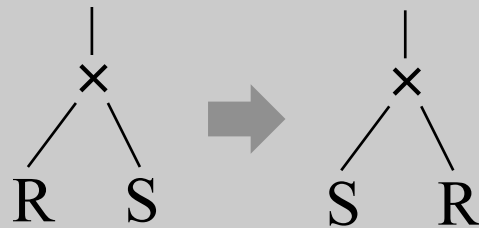
- Transformation rules (preserve equivalence)
- What are good transformations?

Algebraic Rewritings

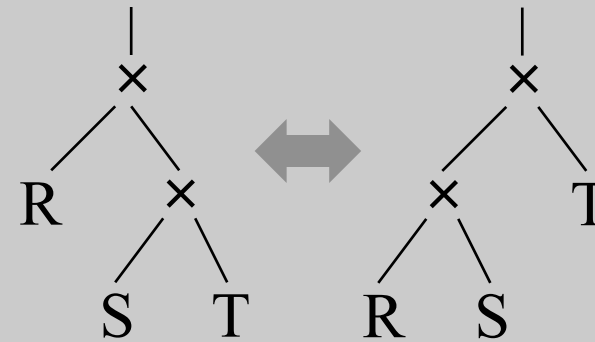
Commutativity & Associativity

Commutativity

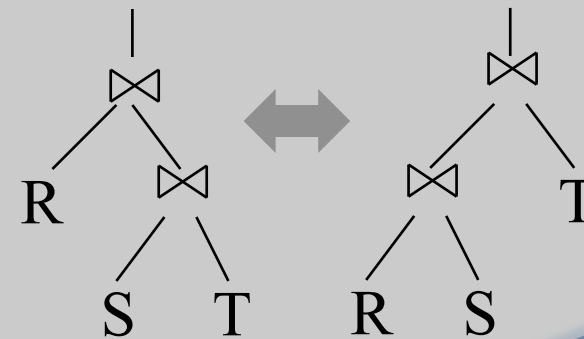
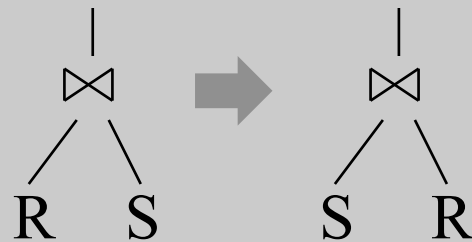
Cartesian
Product



Associativity



Natural
Join

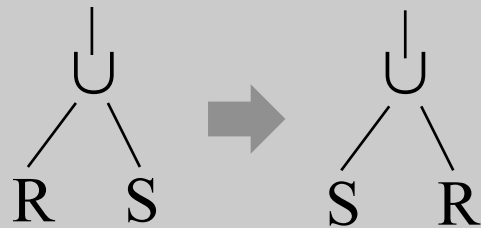


Algebraic Rewritings

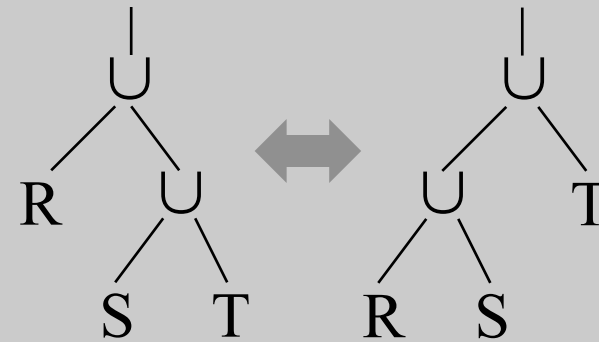
Commutativity & Associativity

Commutativity

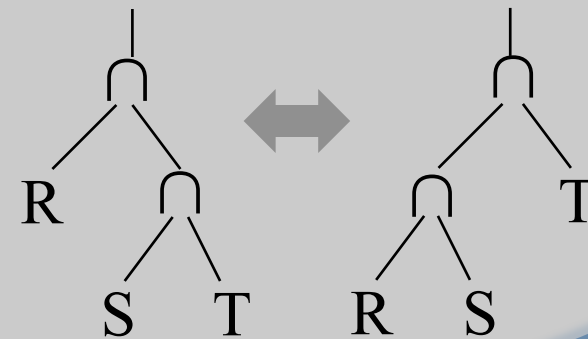
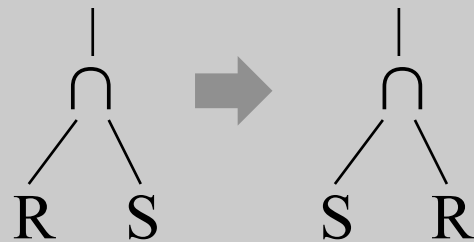
Union



Associativity

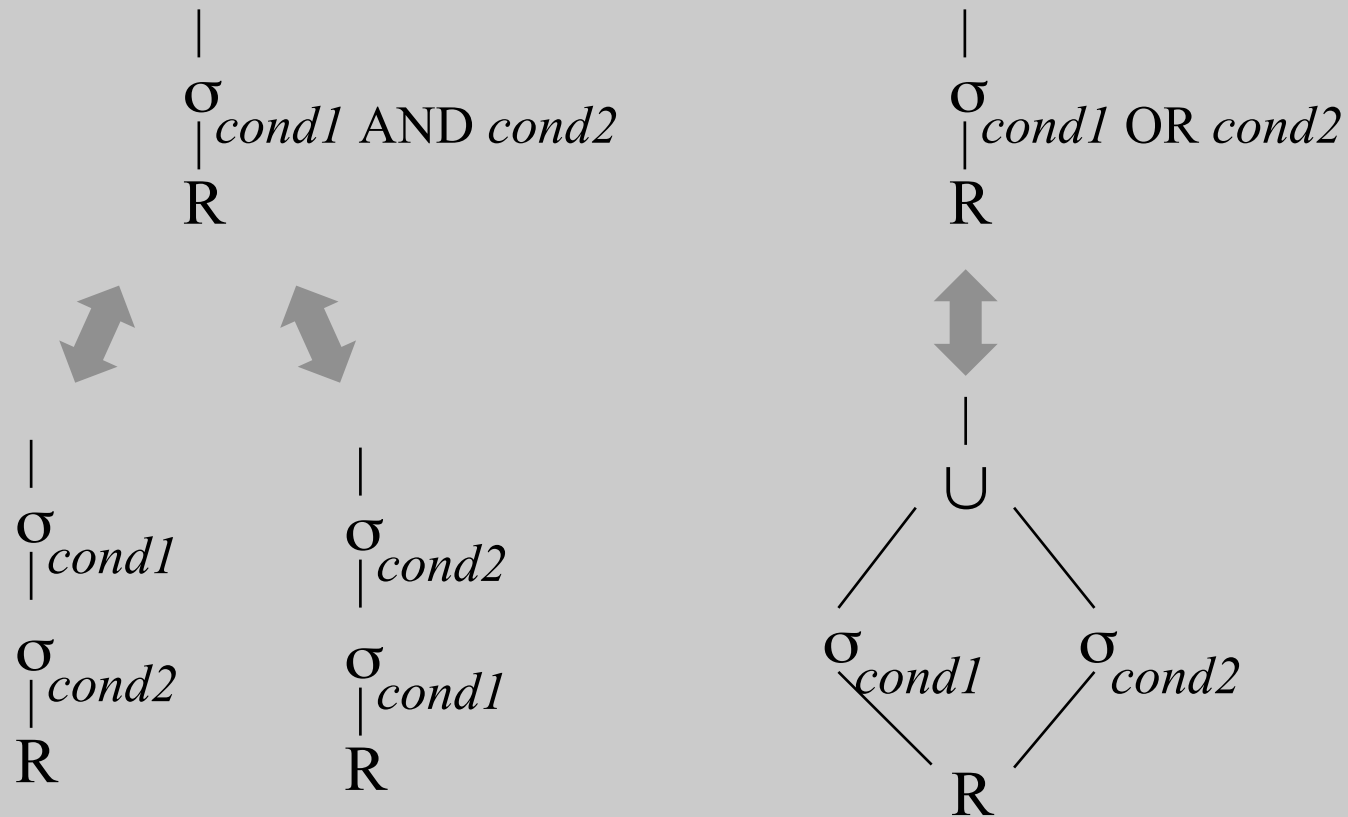


Intersection



Algebraic Rewritings

Decomposition of Logical Connectives



Algebraic Rewritings

Decomposition of Negation

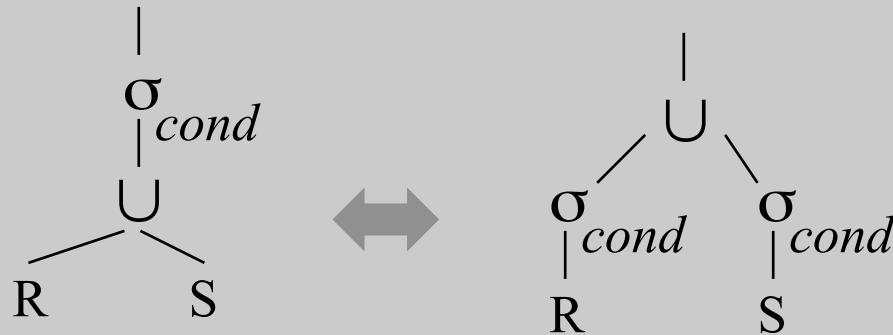
$$\begin{array}{c} | \\ \sigma_{cond1 \text{ AND NOT } cond2} \\ | \\ R \end{array} \quad \longleftrightarrow$$

$$\begin{array}{c} | \\ \sigma_{NOT \ cond2} \\ | \\ R \end{array} \quad \longleftrightarrow$$

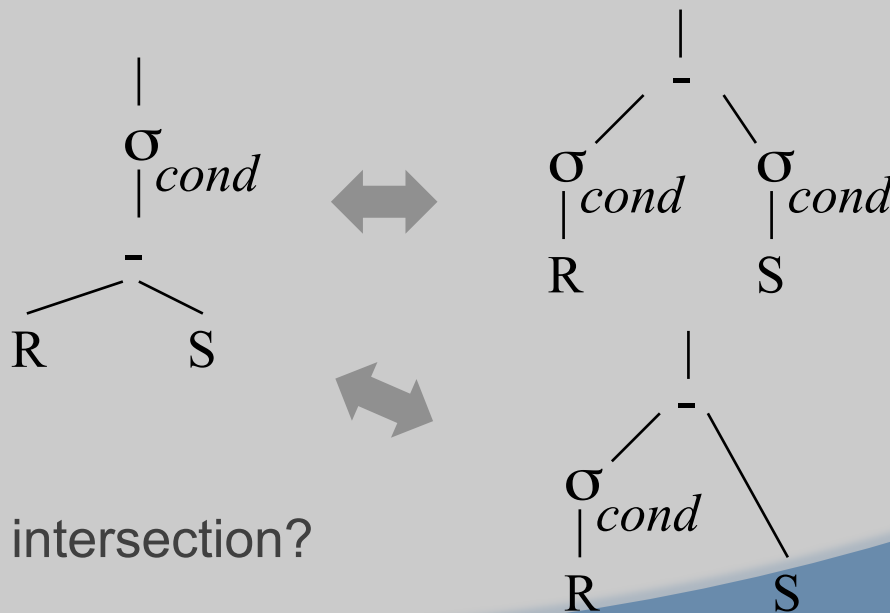
Algebraic Rewritings

Pushing Selection through Union & Difference

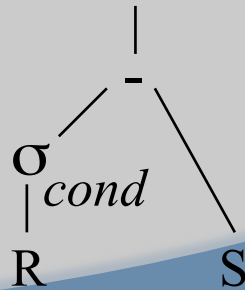
Union



Difference

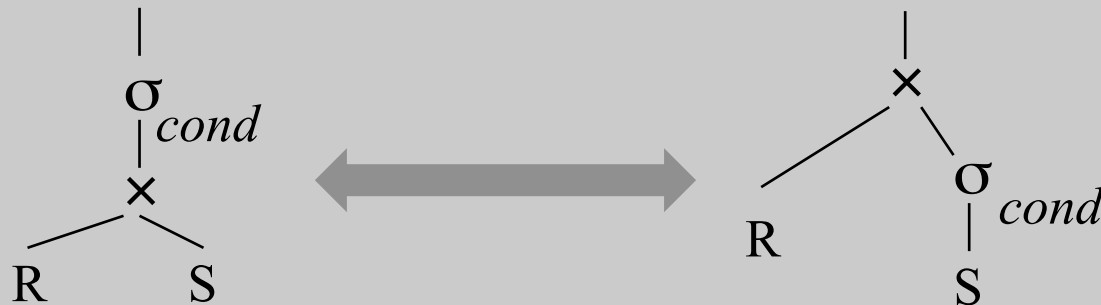


What about intersection?

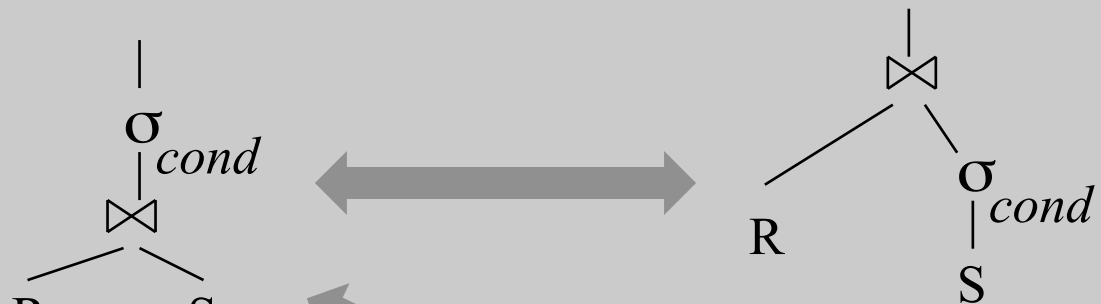


Algebraic Rewritings

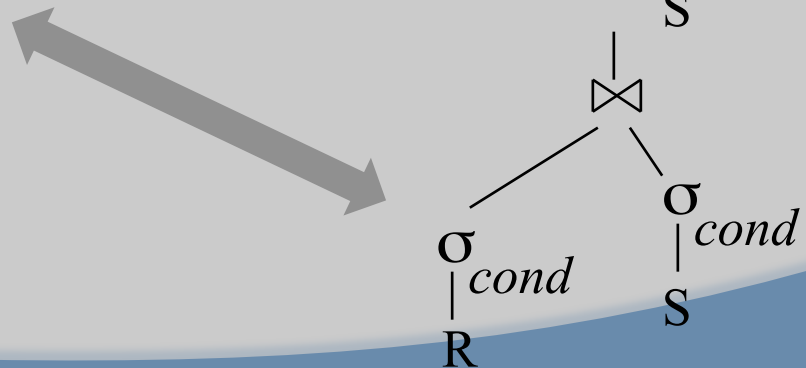
Pushing Selection through Cartesian Product & Join



The right direction requires that $cond$ refers to S attributes only



The right direction requires that $cond$ refers to S attributes only



The right direction requires that all the attributes in $cond$ appear in both R and S

Rules: π, σ combined

Let x = subset of R attributes

z = attributes in predicate P (subset of R attributes)

$$\pi_x[\sigma_p(R)] = \pi_x \{ \sigma_p [\pi_{xz}(\cancel{\pi_x}(R))] \}$$

Algebraic Rewritings

Projection Decomposition

$$\begin{array}{c} | \\ \pi_X \\ | \\ R \end{array} \rightarrow \begin{array}{c} | \\ \pi_X \\ | \\ \pi_{XY} \\ | \\ R \end{array}$$

Derived Rules: $\sigma + \bowtie$ combined

More Rules can be Derived:

$$\sigma_{p \wedge q} (R \bowtie S) = ?$$

where **p** only at **R**, **q** only at **S**

Derived Rules: $\sigma + \bowtie$ combined

More Rules can be Derived:

$$\sigma_{p \wedge q} (R \bowtie S) =$$

$$\sigma_p [\sigma_q (R \bowtie S)] =$$

$$\sigma_p [R \bowtie \sigma_q (S)] =$$

$$[\sigma_p (R)] \bowtie [\sigma_q (S)]$$

where **p** only at **R**, **q** only at **S**

Which are always “good” transformations?

- $\sigma_{p1 \wedge p2}(R) \rightarrow \sigma_{p1}[\sigma_{p2}(R)]$
- $\sigma_p(R \bowtie S) \rightarrow [\sigma_p(R)] \bowtie S$
- $R \bowtie S \rightarrow S \bowtie R$
- $\pi_x[\sigma_p(R)] \rightarrow \pi_x\{\sigma_p[\pi_{xz}(R)]\}$

Bottom Line

- No transformation is always good at the l.q.p level
- Usually good:
 - early selections
 - elimination of cartesian products
 - elimination of redundant subexpressions
- Many transformations lead to “promising” plans
 - Commuting/rearranging joins
 - In practice too “combinatorially explosive” to be handled as rewriting of l.q.p.

Arranging the Join Order: The Wong-Youssefi algorithm (INGRES)

Sample TPC-H Schema

```
Nation(NationKey, NName)
Customer(CustKey, CName, NationKey)
Order(OrderKey, CustKey, Status)
Lineitem(OrderKey, PartKey, Quantity)
Product(SuppKey, PartKey, PName)
Supplier(SuppKey, SName)
```

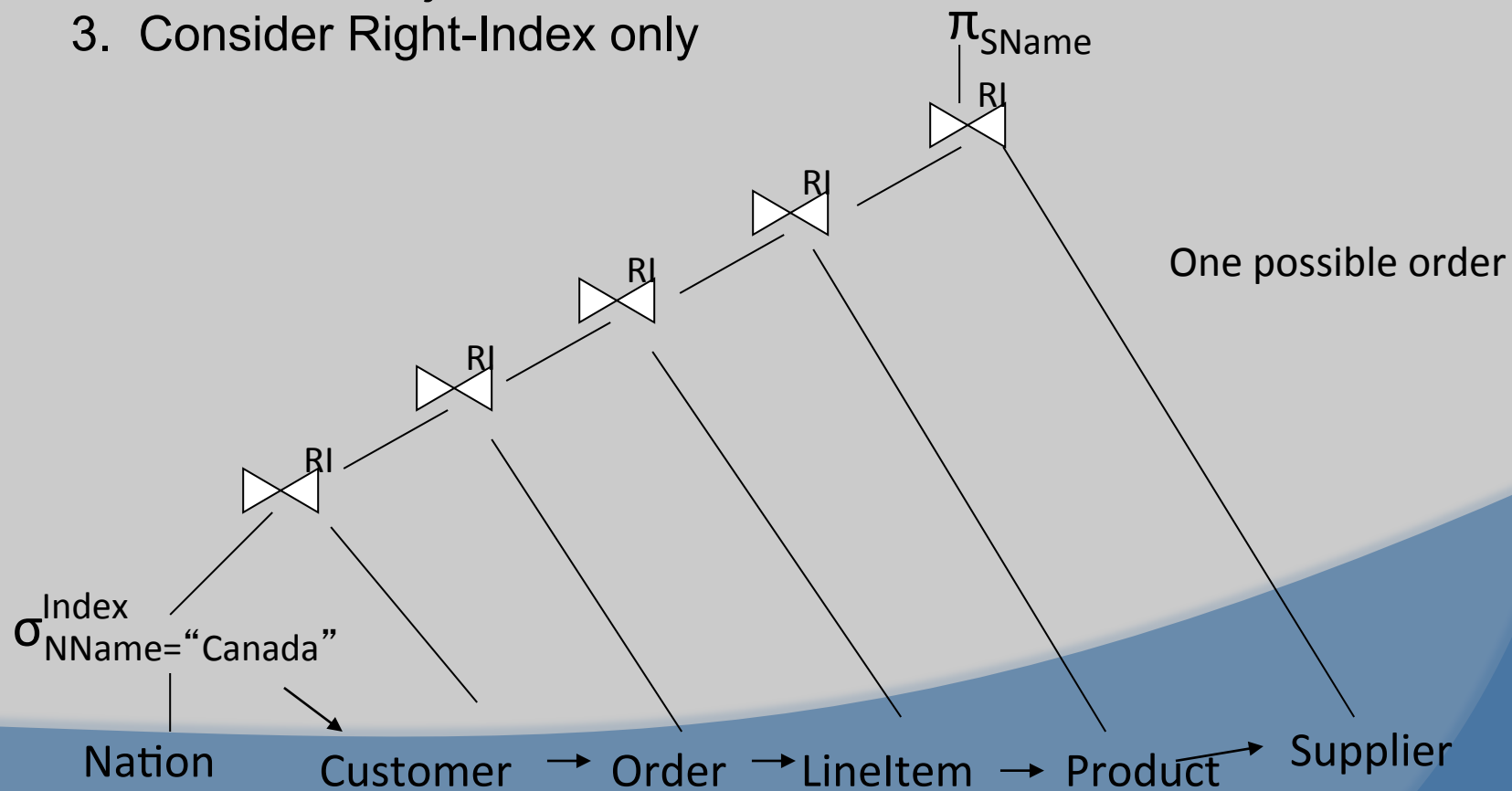
```
SELECT SName
FROM Nation, Customer, Order, Lineitem, Product, Supplier
WHERE Nation.NationKey = Customer.NationKey
      AND Customer.CustKey = Order.CustKey
      AND Order.OrderKey=Lineitem.OrderKey
      AND Lineitem.PartKey= Product.Partkey
      AND Product.Suppkey = Supplier.SuppKey
      AND NName = "Canada"
```

Find the names
of suppliers that
sell a product
that appears in
a line item of an
order made by a
customer who is
in Canada

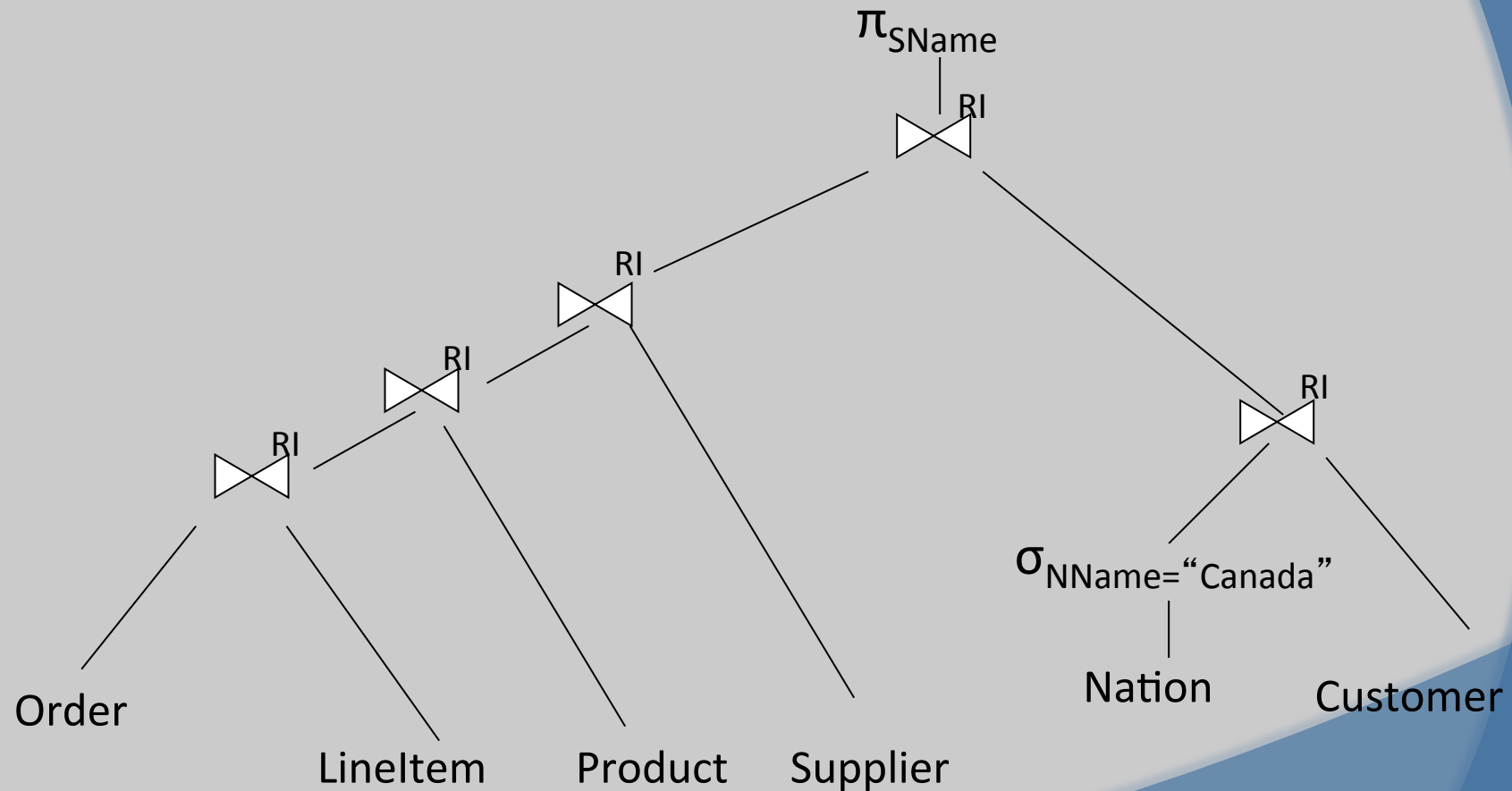
Challenges with Large Natural Join Expressions

For simplicity, assume that in the query

1. All joins are natural
2. whenever two tables of the FROM clause have common attributes we join on them
3. Consider Right-Index only



Multiple Possible Orders

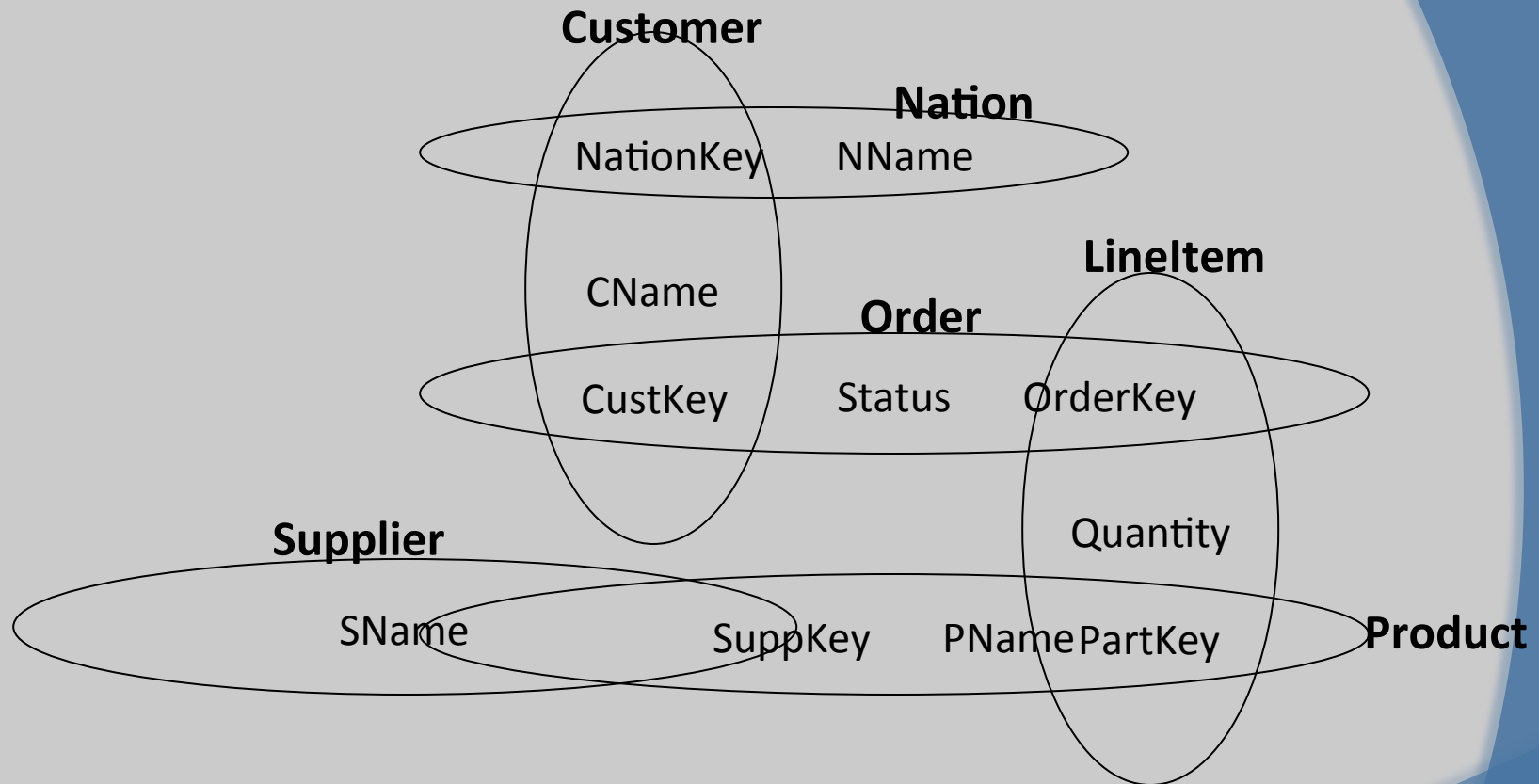


Wong-Yussefi algorithm

assumptions and objectives

- Assumption 1 (weak): Indexes on all join attributes (keys and foreign keys)
- Assumption 2 (strong): At least one selection creates a *small* relation
A join with a small relation results in a small relation
- Objective: Create sequence of index-based joins such that all intermediate results are small

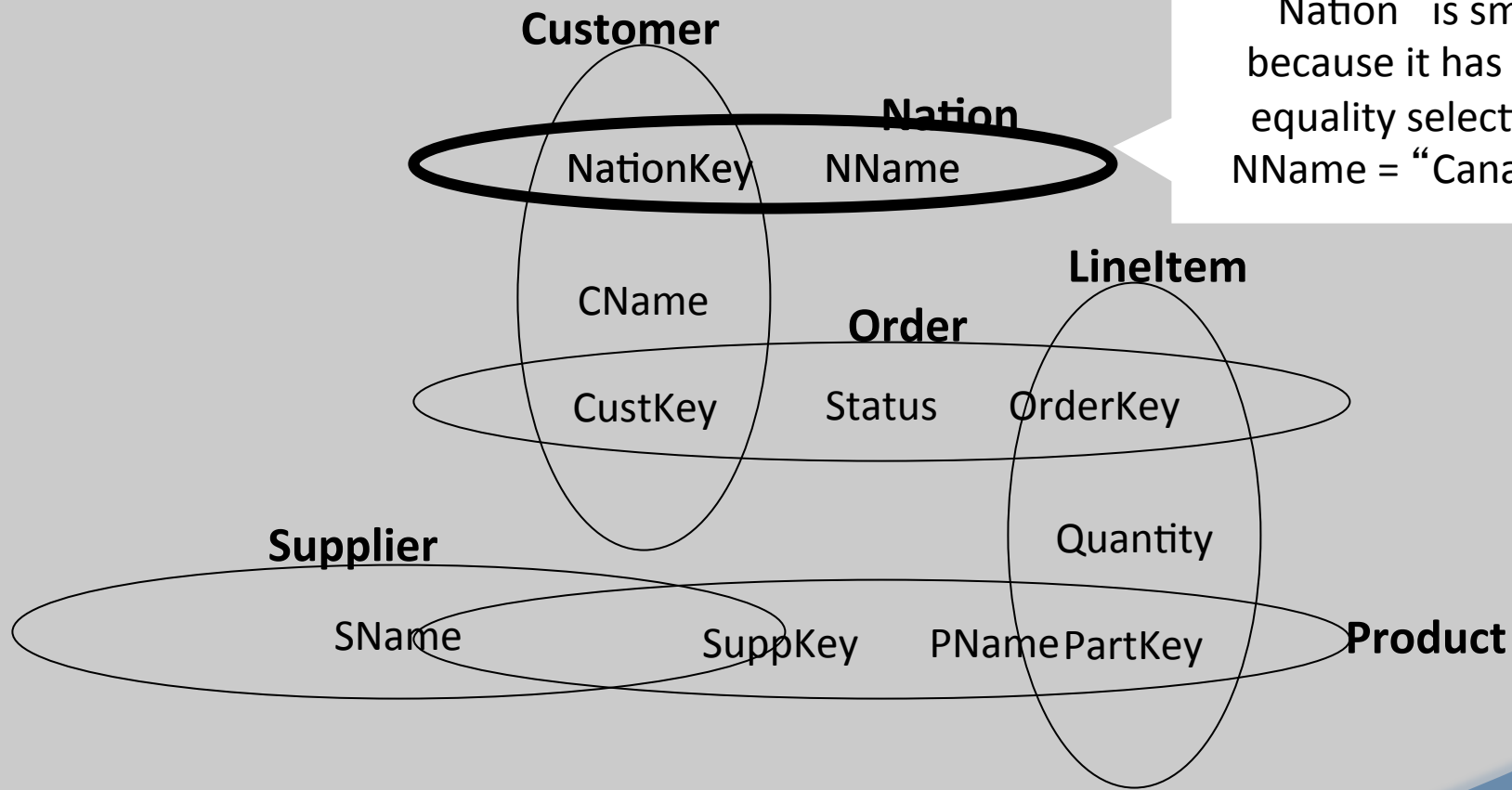
Hypergraphs



- relation hyperedges
 - two hyperedges for same relation are possible
- each node is an attribute
- can extend for non-natural equality joins by merging nodes

Small Relations/Hypergraph Reduction

“Nation” is small
because it has the
equality selection
NName = “Canada”

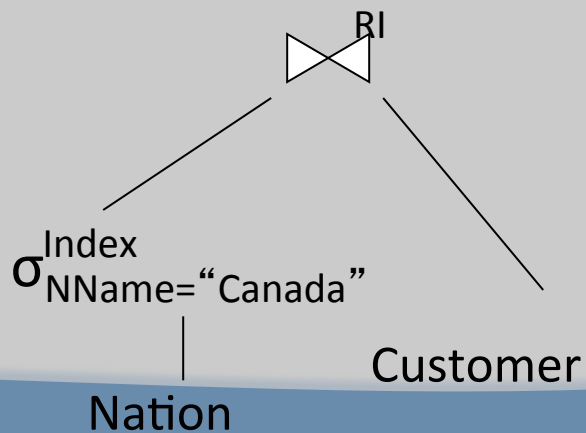
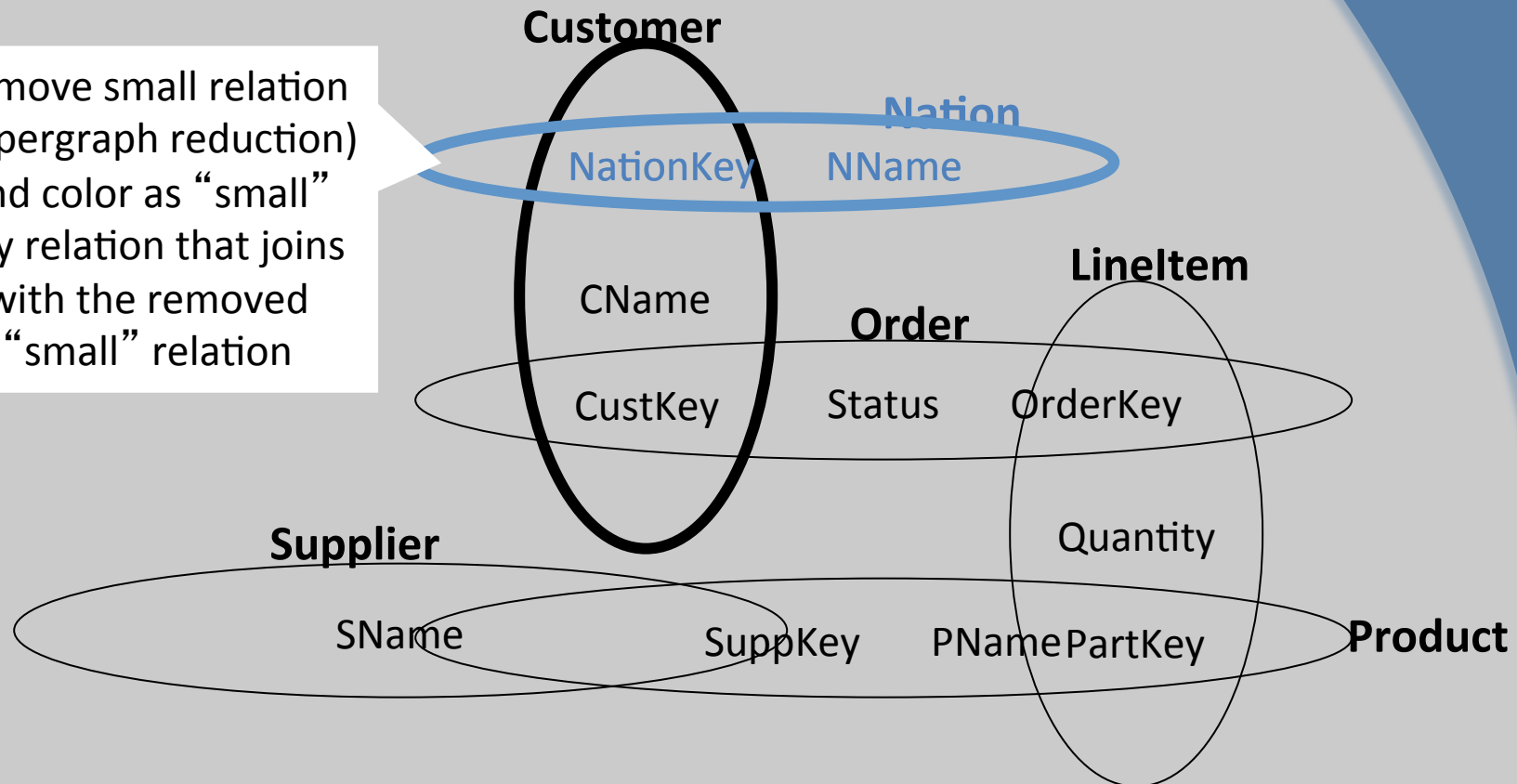


$\sigma_{\text{Index NName}=\text{"Canada"}}$

Nation

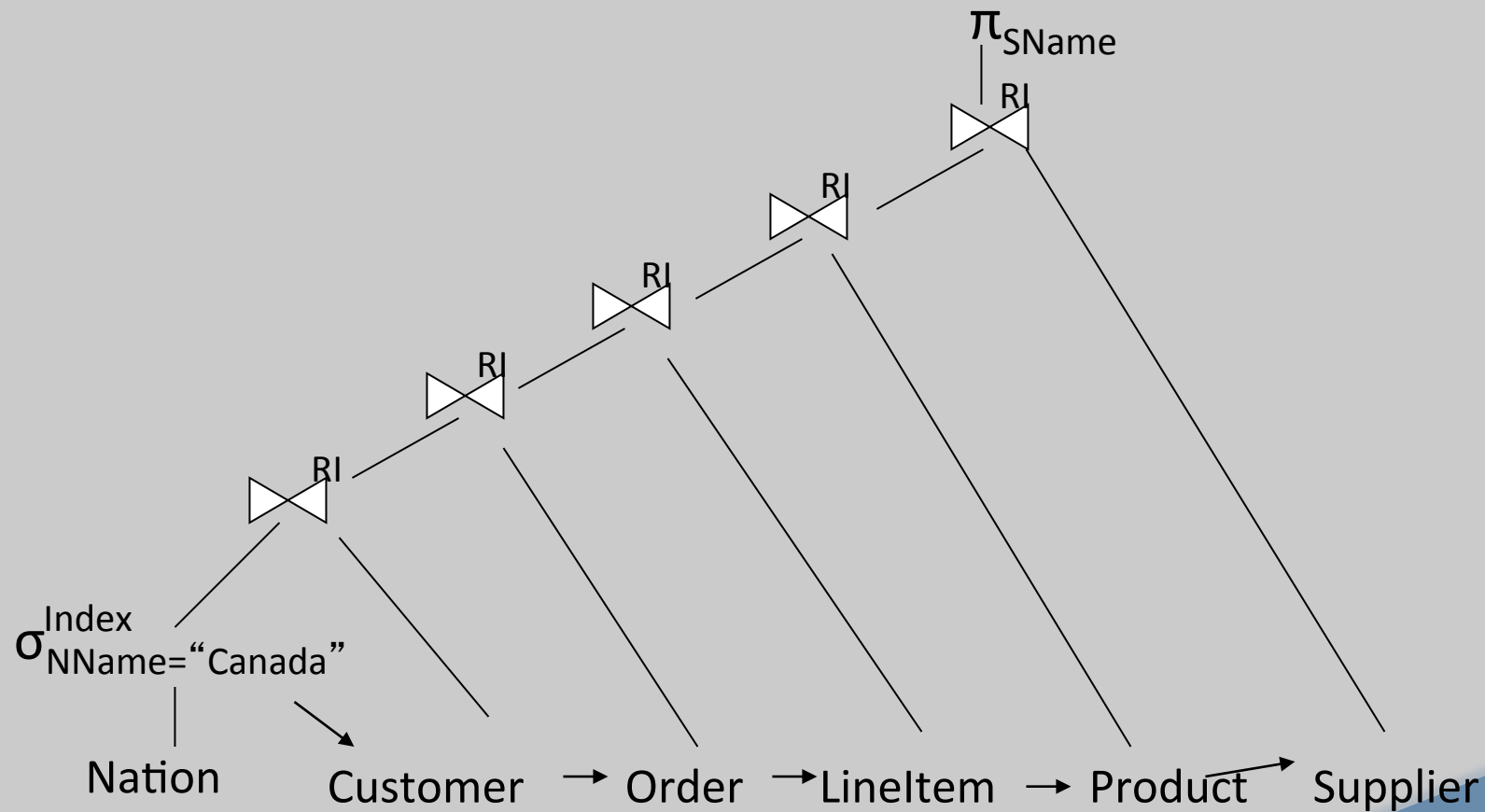
Pick a small relation
(and its conditions)
to start the plan

Remove small relation
(hypergraph reduction)
and color as “small”
any relation that joins
with the removed
“small” relation



Pick a small relation
(and its conditions if
any) and join it with
the small relation
that has been
reduced

After a bunch of steps...

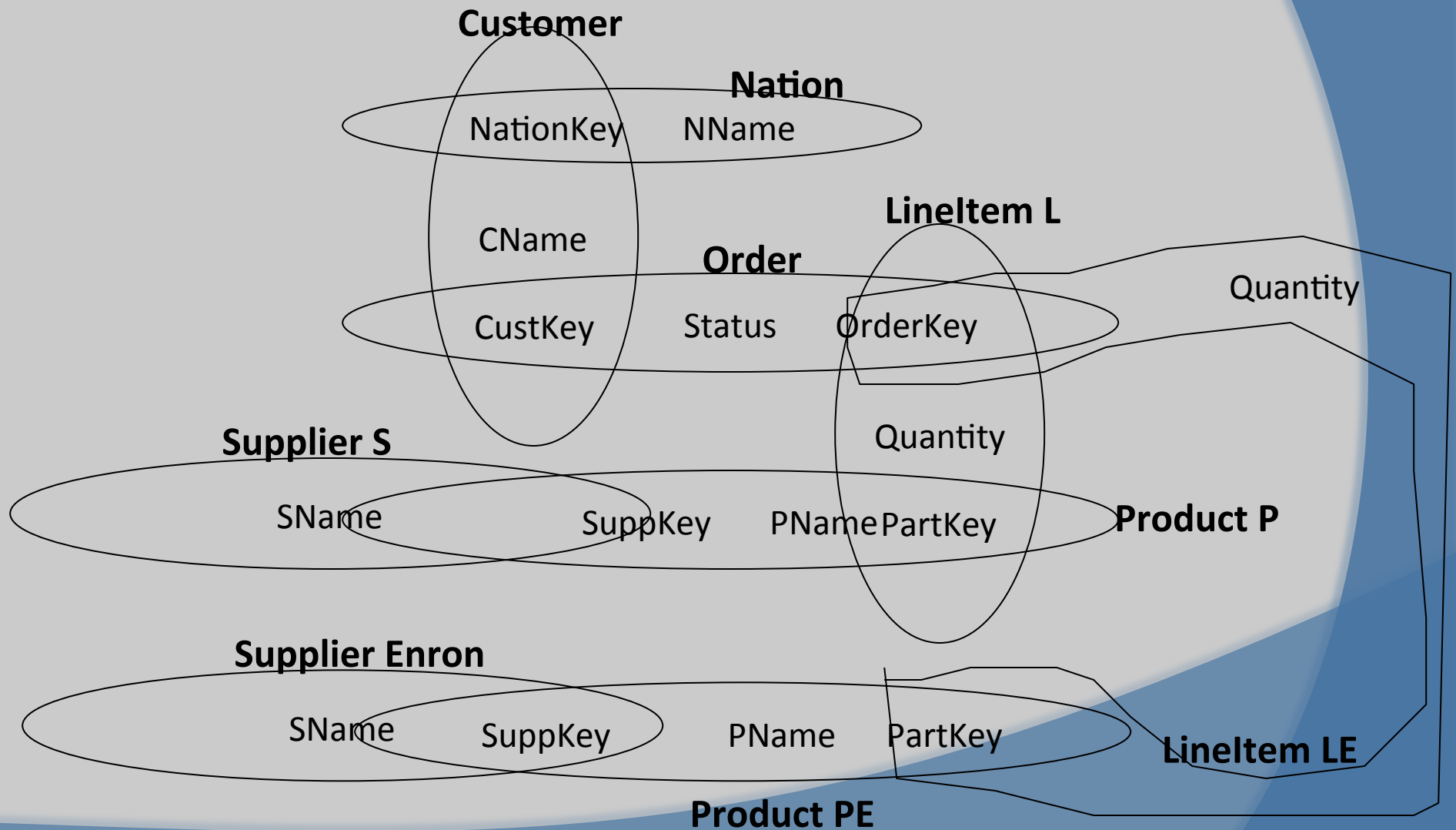


Multiple Instances of Each Relation

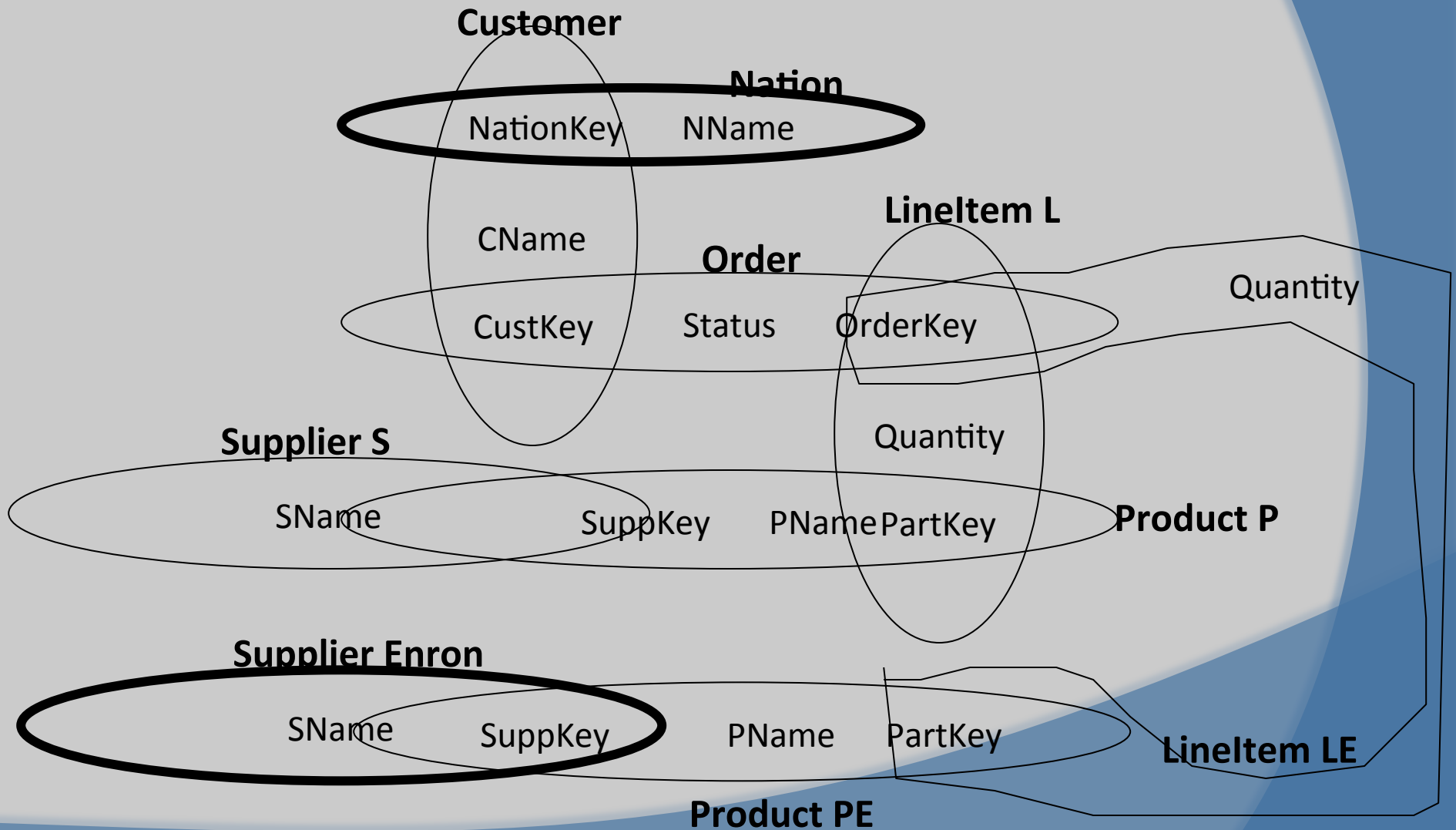
```
SELECT S.SName
FROM Nation, Customer, Order, Lineltem L, Product P, Supplier S,
      Lineltem LE, Product PE, Supplier Enron
WHERE Nation.NationKey = Customer.NationKey
      AND Customer.CustKey = Order.CustKey
      AND Order.OrderKey=L.OrderKey
      AND L.PartKey= P.Partkey
      AND P.Suppkey = S.SuppKey
      AND Order.OrderKey=LE.OrderKey
      AND LE.PartKey= PE.Partkey
      AND PE.Suppkey = Enron.SuppKey
      AND Enron.Sname = "Enron"
      AND NName = "Cayman"
```

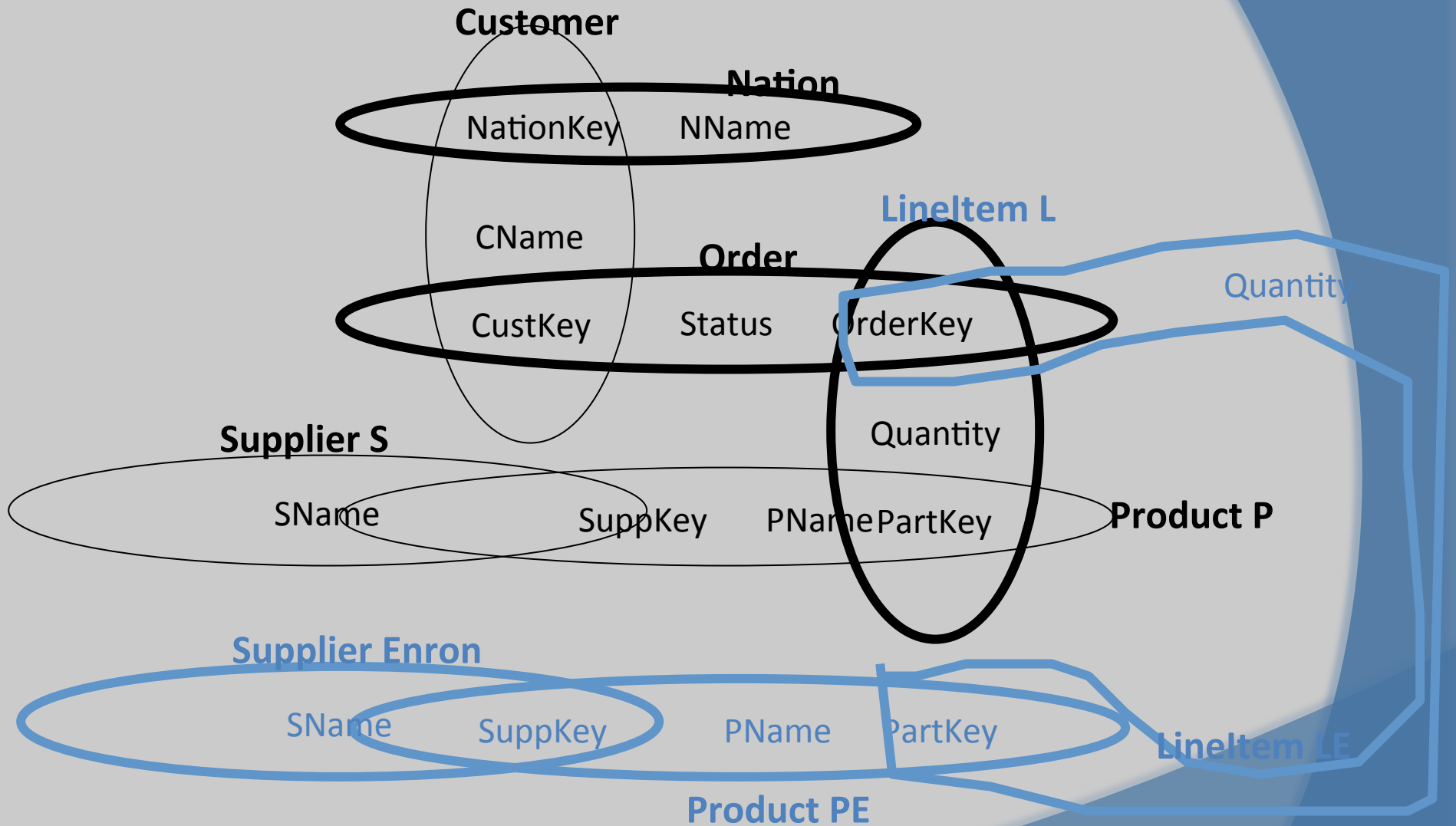
Find the names
of suppliers
whose products
appear in an
order made by a
customer who is
in Cayman
Islands and an
Enron product
appears in the
same order

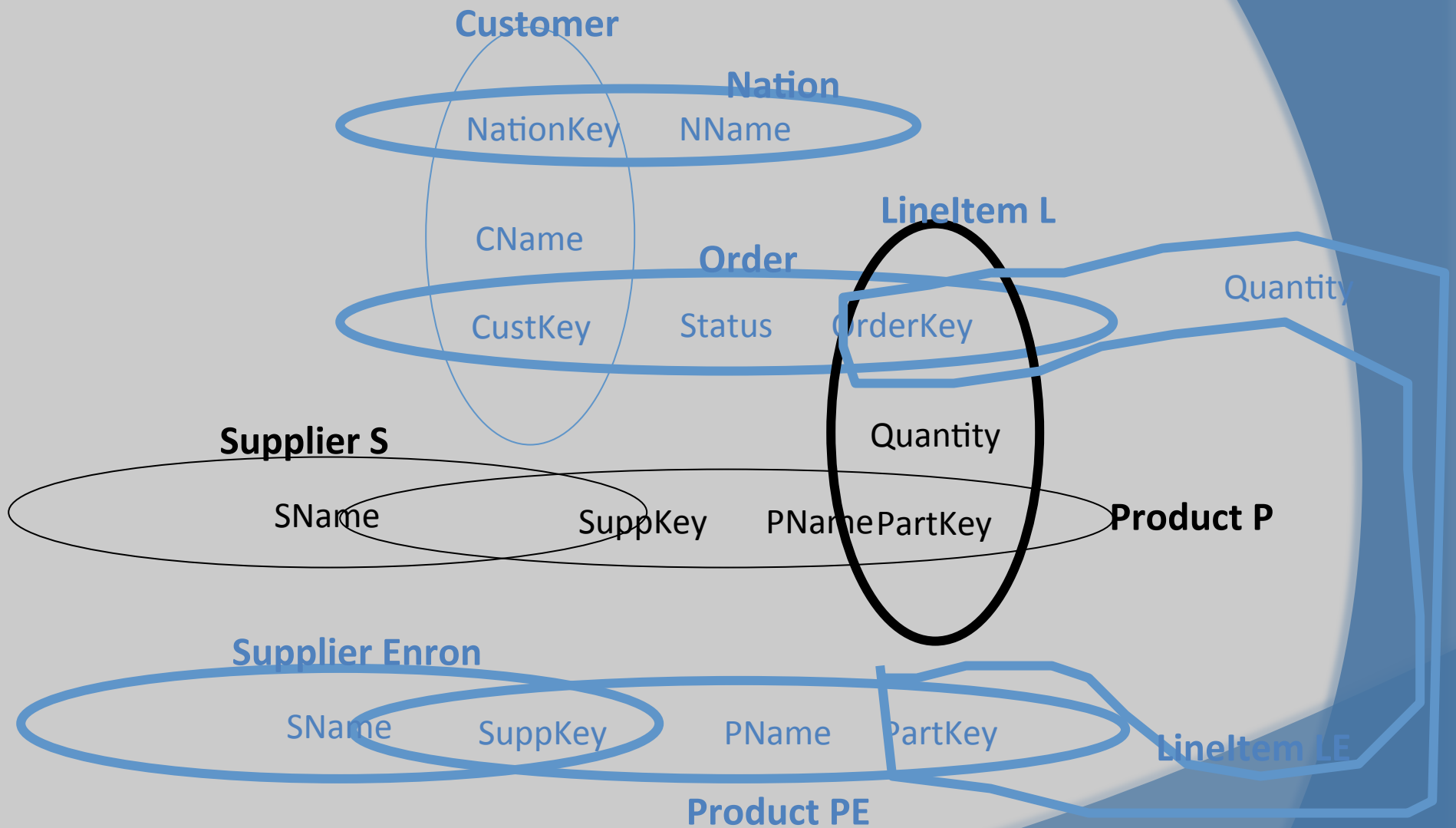
Multiple Instances of Each Relation

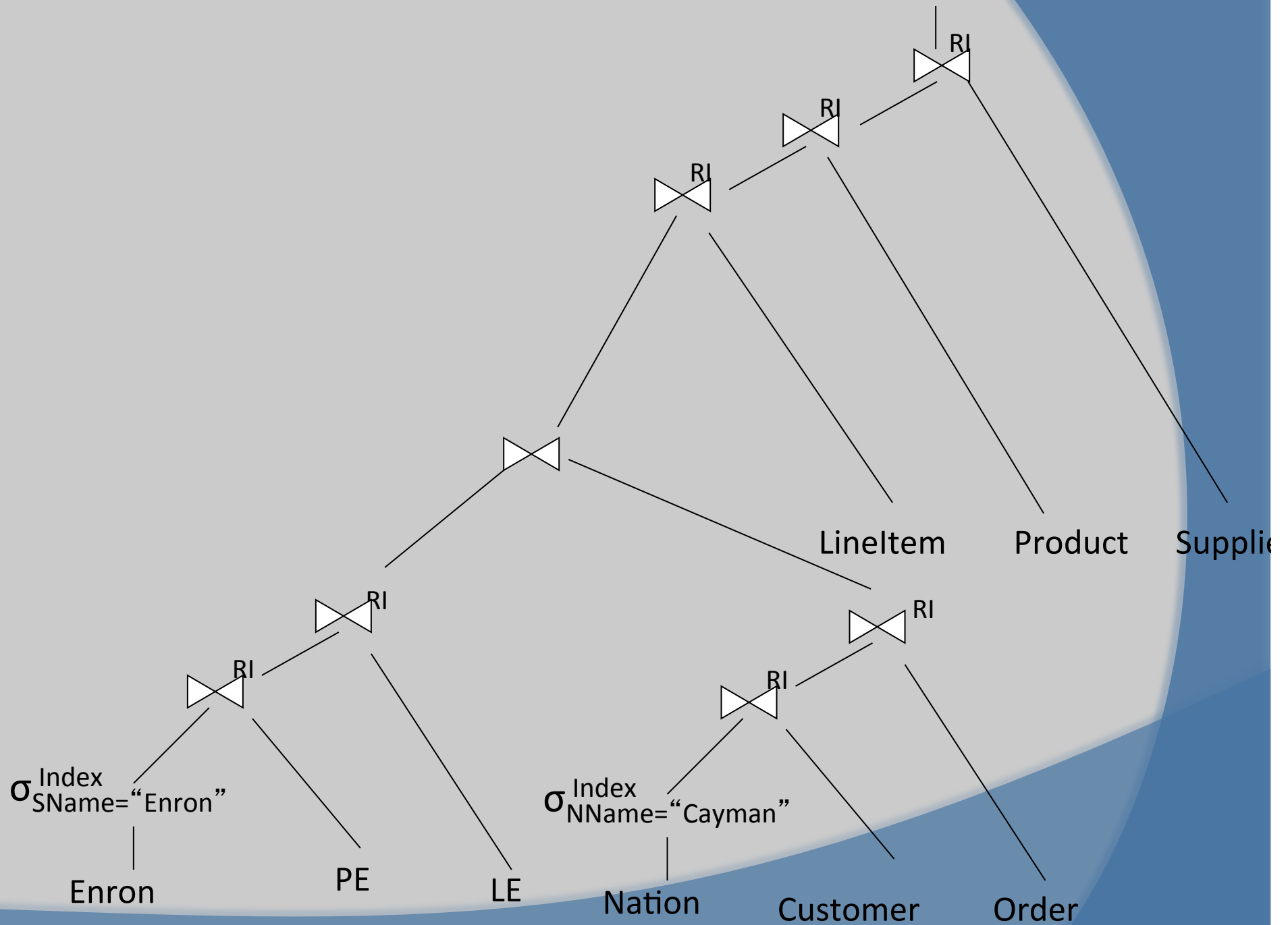


Multiple choices are possible









Algorithms for Relational Algebra Operators

- Three primary techniques

Sorting

Hashing

Indexing

- Three degrees of difficulty

Data small enough to fit in memory

Too large to fit in main memory but small enough to be handled by a “two-pass” algorithm

So large that “two-pass” methods have to be generalized to “multi-pass” methods (quite unlikely nowadays)

Join Implementations

- Nested-loop Join
- Merge Join
- Index Join
- Hash Join

Join Example

$R1 \bowtie R2$ on common attribute C

Nested-Loop Join

- Nested loop join
(conceptually, without taking into account disk block issues)

for each $r \in R1$ do

for each $s \in R2$ do

if $r.C = s.C$ then output r,s pair

Merge Join

- Merge join (*conceptually*)

(1) if R1 and R2 not sorted, sort them

(2) $i \leftarrow 1; j \leftarrow 1;$

While $(i \leq T(R1)) \wedge (j \leq T(R2))$ do

if $R1\{i\}.C = R2\{j\}.C$ then outputTuples

else if $R1\{i\}.C > R2\{j\}.C$ then $j \leftarrow j+1$

else if $R1\{i\}.C < R2\{j\}.C$ then $i \leftarrow i+1$

Merge Join (continued)

Procedure Output-Tuples

While $(R1\{i\}.C = R2\{j\}.C) \wedge (i \leq T(R1))$ do

$jj \leftarrow j$;

 while $(R1\{i\}.C = R2\{jj\}.C) \wedge (jj \leq T(R2))$ do

 [output pair $R1\{i\}$, $R2\{jj\}$;

$jj \leftarrow jj+1$]

$i \leftarrow i+1$]

Merge Join Example

i	R1{i}.C	R2{j}.C	j
1	10	5	1
2	20	20	2
3	20	20	3
4	30	30	4
5	40	30	5
		50	6
		52	7

Index Join

- Index Join (*conceptually*)

For each $r \in R1$ do

[$X \leftarrow \text{index}(R2, C, r.C)$

Assume $R2.C$ index

for each $s \in X$ do

output r,s pair]

Note: $X \leftarrow \text{index}(\text{rel}, \text{attr}, \text{value})$

means that $X = \text{set of rel tuples with attr = value}$

Hash Join

- Hash Join (*conceptually*)
 - (1) Hash R1 tuples into G buckets
 - (2) Hash R2 tuples into H buckets
 - (3) For $i = 0$ to k do
 - match tuples in G_i , H_i buckets

Hash function h , range $0 \rightarrow k$
Buckets for R1: $G_0, G_1, \dots, G_k$
Buckets for R2: $H_0, H_1, \dots, H_k$

Hashing Example: Even/Odd

R1	R2	<i>Buckets</i>	R1	R2
2	5	Even	2 4 8	4 12 8 14
4	4			
3	12	Odd	3 5 9	5 3 13 11
5	3			
8	13			
9	8			
	11			
	14			

Factors affecting performance

- (1) Tuples of relation stored physically together?
- (2) Relations sorted by join attribute?
- (3) Indexes exist?

More details

- The above algorithms are conceptual and do not account for physical implementation details.
- In reality we have to specify exactly how the algorithm will operate in terms of disk/memory accesses.
- Then based on this information we can calculate the cost of each operator in terms of disk accesses.

Disk-oriented Computation Model

- There are M main memory buffers.
Each buffer has the size of a disk block
- The input relation is read one block at a time.
- The cost is the number of blocks read.
- If B consecutive blocks are read the cost is B/d .
- The output buffers are not part of the M buffers mentioned above.
Pipelining allows the output buffers of an operator to be the input of the next one.
We do not count the cost of writing the output.

Disk-oriented Computation Model

Metric: # of I/Os (ignoring writing the result)

Caution: This may not be the best way to compare

- ignoring CPU costs
- ignoring timing
- ignoring double buffering requirements

Notation

- M = # memory blocks available
- $B(R)$ = number of blocks that R occupies
- $T(R)$ = number of tuples of R
- $V(R, [a_1, a_2, \dots, a_n])$ = number of distinct tuples in the projection of R on $a_1, a_2, \dots, a_n$

One-Pass Main Memory Algorithms for Unary Operators

- Assumption: Enough memory to keep the relation
- Projection and selection:
Scan the input relation R and apply operator one tuple at a time
Incremental cost of “on the fly” operators is 0
Cost depends on
 - clustering of R
 - whether the blocks are consecutive
- Duplicate elimination and aggregation
Create one entry for each group and compute the aggregated value of the group
It becomes hard to assume that CPU cost is negligible
 - main memory data structures are needed

One-Pass Nested Loop Join

- Assume $B(R)$ is less than M
- Tuples of R should be stored in an efficient lookup structure
- Algorithm:

```
for each block  $B_r$  of  $R$  do
  store tuples of  $B_r$  in main memory
for each block  $B_s$  of  $S$  do
  for each tuple  $s$  of  $B_s$ 
    join tuples of  $s$  with matching tuples of  $R$ 
```

What is the cost of this algorithm?

Generalization of Nested-Loops

- Works even if $B(R)$ is greater than M

for each chunk of $M-1$ blocks B_r of R do

store tuples of B_r in main memory

for each block B_s of S do

for each tuple s of B_s

join tuples of s with matching tuples of R

What is the cost of this algorithm?

Two-Pass Hash-Based Algorithms

- General Idea: Hash the tuples of the input arguments in such a way that all tuples that must be considered together will have hashed to the same hash value.
If there are M buffers pick $M-1$ as the number of hash buckets
- Example: Duplicate Elimination
Phase 1: Hash each tuple of each input block into one of the $M-1$ bucket/buffers. When a buffer fills save to disk.
Phase 2: For each bucket:
 - load the bucket in main memory,
 - treat the bucket as a small relation and eliminate duplicates
 - save the bucket back to disk.

Catch: Each bucket has to be less than M .

Cost = ?