Advanced Topics in Data Management

Lecture 1

Welcome

- Welcome to the Special Topics Course: Advanced Topics in Data Management
- Goal: drill deep into modern database engines, new and old techniques, explore extensions.

Staff

- Instructor: Dan Suciu

- TA: Remy Wang

Course Organization

- Lectures: Thursday, 6:30-9:20
  - Guest lecture followed by regular lecture
- Project:
  - Open ended, e.g. extend some query optimizer
- Paper reviews:
  - Short reviews, maybe short programs
  - Please submit *before* the lecture

Evaluation

- Project 50%
- Paper reviews 30%
- Class participation 20%

Communication

- Ed – everybody is subscribed
- Class mailing list – very low traffic
- Website

Lectures

1. Introduction, Review of Query Processing
2. Query Processing (continued)
3. Rebecca Taft (Cockroach Labs)
   Tutorial on Egg (Max Wilsey)
4. Nico Bruno and César A. Galindo-Legaria (Microsoft):
   The Cascades Framework
   Tutorial on optimizing tensor expressions (Remy Wang)
5. Sergey Melnik (Google): Big Query
6. Ippokratis Pandis (Amazon): Redshift
7. Guest lecturer: Doug Brown (Teradata)
8. Guest lecturer: Jiaqi Yan (Snowflake)
9. Guest lecturer: Martin Bravenboer (RelationalAI)
10. Project presentations

Please attend the lectures!

What you (we?) will learn

- Consolidated knowledge of query optimization and execution

- Understand the choices made by various state-of-the art commercial systems

- Explore possible extensions of optimizers; e.g. to tensor algebra

Tools

- Please have your favorite, state-of-the-art database system on your laptop!
  - Postgres, SQL Server -- yes
  - (access to) Redshift, Snowflake -- yes
  - Sqlite -- no

- The project webpage has links to some open source optimizers

Prerequisits

- If you took csep544, you should be fine in this class
- If you haven’t, then you should think hard if you want to take this class

Today’s Outline

- Overview of SQL processing and optimization

Relational Data Model

Relational Data Model

- A Database is a collection of relations
- A Relation is a set of tuples
  - Also called Table
- A Tuple t is an element of $\mathbf{Dom_1 \times \ldots \times Dom_n}$

Discussion

- Order of records is immaterial
- Sets semantics or bag semantics
- Attribute domains are primitive types:
  First Normal Form (1NF)

Schema

- Relation schema: describes column heads

- Database schema: set of all relation schemas

Instance

- Relation instance: concrete table content
- Database instance: set of relation instances

Relational Query Language

- Set-at-a-time:
  - Query inputs and outputs are relations

- Two variants of the query language:
  - SQL: declarative
  - Relational algebra: specifies order of operations

Discussion

- **Physical Data Independence**:
  - *No* physical spec of the data

- Declarative query language:
  - Say *what* we want
  - Don't say *how* to get it

- Query optimization: what → how

SQL

SQL

- Standard query language
- Introduced late 70’s, now it ballooned
- We briefly review “core SQL” (whatever that means)

Structured Query Language: SQL

- Data definition language: DDL
  - Statements to create, modify tables and views
  - CREATE TABLE …,
    CREATE VIEW …,
    ALTER TABLE…

Structured Query Language: SQL

- Data definition language: DDL
  - Statements to create, modify tables and views
  - CREATE TABLE …,
    CREATE VIEW …,
    ALTER TABLE…
- Data manipulation language: DML
  - Statements to issue queries, insert, delete data
  - SELECT-FROM-WHERE…,
    INSERT…,
    UPDATE…,
    DELETE…

Our focus

SQL Query

Basic form: (plus many many more bells and whistles)

```
SELECT  <attributes>
FROM    <one or more relations>
WHERE   <conditions>
```

```
Supplier(sno,sname,scity,sstate)
Supply(sno,pno,qty,price)
Part(pno,pname,psize,pcolor)
```

Quick Review of SQL

```
Supplier(sno,sname,scity,sstate)
Supply(sno,pno,qty,price)
Part(pno,pname,psize,pcolor)
```

Quick Review of SQL

Retrieve all parts under $100,
and the cities in Washington that supply them:

```
Supplier(sno,sname,scity,sstate)
Supply(sno,pno,qty,price)
Part(pno,pname,psize,pcolor)
```

Quick Review of SQL

Retrieve all parts under $100,
and the cities in Washington that supply them:

```
SELECT DISTINCT  z.pno, z.pname, x.scity
FROM      Supplier x, Supply y, Part z
WHERE     x.sno = y.sno
       and y.pno = z.pno
       and x.sstate = ‘WA’
       and y.price < 100
```

```
Supplier(sno,sname,scity,sstate)
Supply(sno,pno,qty,price)
Part(pno,pname,psize,pcolor)
```

Terminology

- Selection/filter: return a subset of the rows:
  - SELECT * FROM Supplier
    WHERE scity = ’Seattle’
- Projection: return subset of the columns:
  - SELECT DISTINCT scity FROM Supplier;
- Join: refers to combining two or more tables
  - SELECT * FROM Supplier, Supply, Part …

`Supplier(sno,sname,scity,sstate)`
`Supply(sno,pno,qty,price)`
`Part(pno,pname,psize,pcolor)`

Self-Joins

Find the Parts numbers available both from suppliers in Seattle, and suppliers in Portland

```
Supplier(sno,sname,scity,sstate)
Supply(sno,pno,qty,price)
Part(pno,pname,psize,pcolor)
```

Self-Joins

Find the Parts numbers available both from suppliers in Seattle, and suppliers in Portland

```
SELECT DISTINCT  y1.pno
FROM      Supplier x1, Supplier x2, Supply y1, Supply y2
WHERE     x1.scity = 'Seattle'
        and x1.sno = y1.sno
        and x2.scity = 'Portland'
        and x2.sno = y2.sno
        and y1.pno = y2.pno
```

Supplier(sno,sname,scity,sstate)
Supply(sno,pno,qty,price)
Part(pno,pname,psize,pcolor)

Self-Joins

Find the Parts numbers available both from suppliers in Seattle, and suppliers in Portland

Need TWO Suppliers
and TWO Supplies

```
SELECT DISTINCT  y1.pno
FROM      Supplier x1, Supplier x2, Supply y1, Supply y2
WHERE     x1.scity = ‘Seattle’
          and x1.sno = y1.sno
          and x2.scity = ‘Portland’
          and x2.sno = y2.sno
          and y1.pno = y2.pno
```

```
Supplier(sno,sname,scity,sstate)
Supply(sno,pno,qty,price)
Part(pno,pname,psize,pcolor)
```

Self-Joins

Find the Parts numbers available both from suppliers in Seattle, and suppliers in Portland

Need TWO Suppliers
and TWO Supplies

```
SELECT DISTINCT  y1.pno
FROM      Supplier x1, Supplier x2, Supply y1, Supply y2
WHERE     x1.scity = 'Seattle'
          and x1.sno = y1.sno
          and x2.scity = 'Portland'
          and x2.sno = y2.sno
          and y1.pno = y2.pno
```

one in Seattle
the other in Portland

```
Supplier(sno,sname,scity,sstate)
Supply(sno,pno,qty,price)
Part(pno,pname,psize,pcolor)
```

Self-Joins

Find the Parts numbers available both from suppliers in Seattle, and suppliers in Portland

```
SELECT DISTINCT  y1.pno
FROM     Supplier x1, Supplier x2, Supply y1, Supply y2
WHERE    x1.scity = 'Seattle'
         and x1.sno = y1.sno
         and x2.scity = 'Portland'
         and x2.sno = y2.sno
         and y1.pno = y2.pno
```

Need TWO Suppliers
and TWO Supplies

one in Seattle
the other in Portland

the SAME part

Nested-Loop Semantics of SQL

```
SELECT [DISTINCT] a1, a2, …, ak
FROM    R1 AS x1, R2 AS x2, …, Rn AS xn
WHERE  Conditions
```

Nested-Loop Semantics of SQL

```
SELECT [DISTINCT] a_1, a_2, …, a_k
FROM    R_1 AS x_1, R_2 AS x_2, …, R_n AS x_n
WHERE  Conditions
```

```
Answer = {}
for x_1 in R_1 do
    for x_2 in R_2 do
        …..
            for x_n in R_n do
                if Conditions
                    then Answer = Answer ∪ {(a_1,…,a_k)}
return Answer
```

Nested-Loop Semantics of SQL

```
SELECT [DISTINCT] $a_1, a_2, \ldots, a_k$
FROM    $R_1$ AS $x_1$, $R_2$ AS $x_2$, …, $R_n$ AS $x_n$
WHERE  Conditions
```

This SEMANTICS!
It is NOT how the
engine computes
the query!

```
Answer = {}
for $x_1$ in $R_1$ do
    for $x_2$ in $R_2$ do
        …..
            for $x_n$ in $R_n$ do
                if Conditions
                    then Answer = Answer $\cup$ $\{(a_1,\ldots,a_k)\}$
return Answer
```

Query Evaluation

Query Evaluation

- Convert SQL into a query plan
- Optimize the query plan
- Execute each operator of the query plan

Relational algebra (subset)

- Selection $\sigma_{condition}$
- Projection $\Pi_{\mathrm{attributes}}$
- Join $\bowtie_{condition}$
- Duplicate elimination δ

Supplier(sno,sname,scity,sstate)
Supply(sno,pno,qty,price)
Part(pno,pname,psize,pcolor)

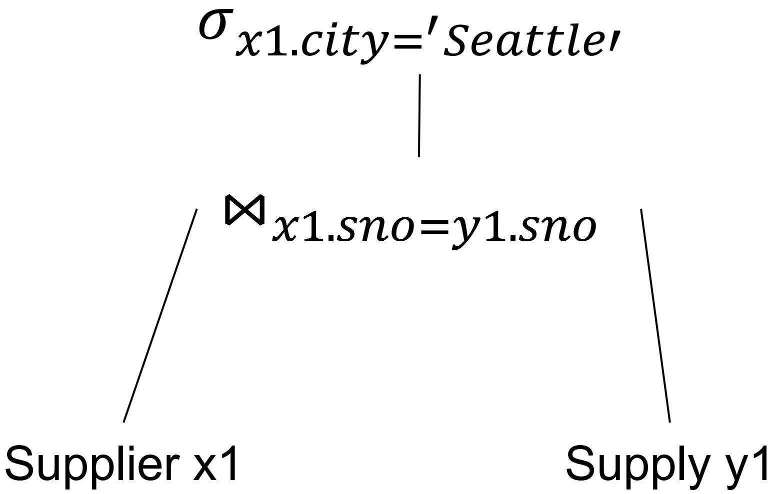
Convert SQL to Query Plan

```
SELECT DISTINCT  y1.pno
FROM      Supplier x1, Supplier x2, Supply y1, Supply y2
WHERE    x1.scity = ‘Seattle’
               and x1.sno = y1.sno
               and x2.scity = ‘Portland’
               and x2.sno = y2.sno
               and y1.pno = y2.pno
```

Supplier(sno,sname,scity,sstate)
Supply(sno,pno,qty,price)
Part(pno,pname,psize,pcolor)

Convert SQL to Query Plan

```
SELECT DISTINCT  y1.pno
FROM      Supplier x1, Supplier x2, Supply y1, Supply y2
WHERE     x1.scity = ‘Seattle’
              and x1.sno = y1.sno
              and x2.scity = ‘Portland’
              and x2.sno = y2.sno
              and y1.pno = y2.pno
```



$\sigma_{x1.city='Seattle'}$

$\bowtie_{x1.sno=y1.sno}$

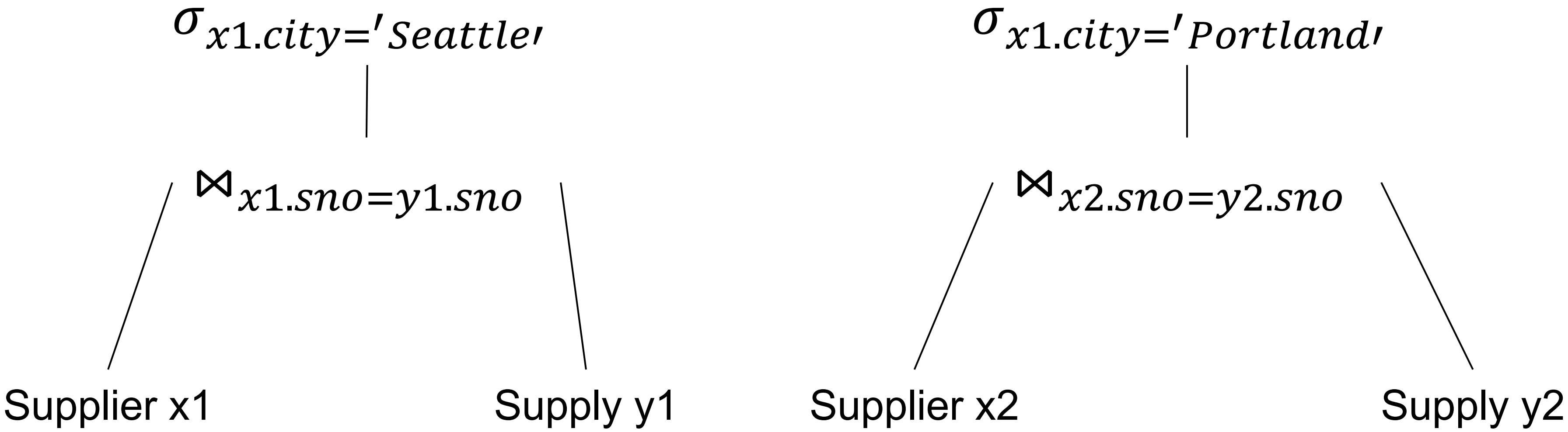
Supplier x1

Supply y1

```
Supplier(sno,sname,scity,sstate)
Supply(sno,pno,qty,price)
Part(pno,pname,psize,pcolor)
```

Convert SQL to Query Plan

```
SELECT DISTINCT  y1.pno
FROM     Supplier x1, Supplier x2, Supply y1, Supply y2
WHERE    x1.scity = 'Seattle'
           and x1.sno = y1.sno
           and x2.scity = 'Portland'
           and x2.sno = y2.sno
           and y1.pno = y2.pno
```



$\sigma_{x1.city='Seattle'}$

$\bowtie_{x1.sno=y1.sno}$

Supplier x1

Supply y1

$\sigma_{x1.city='Portland'}$

$\bowtie_{x2.sno=y2.sno}$

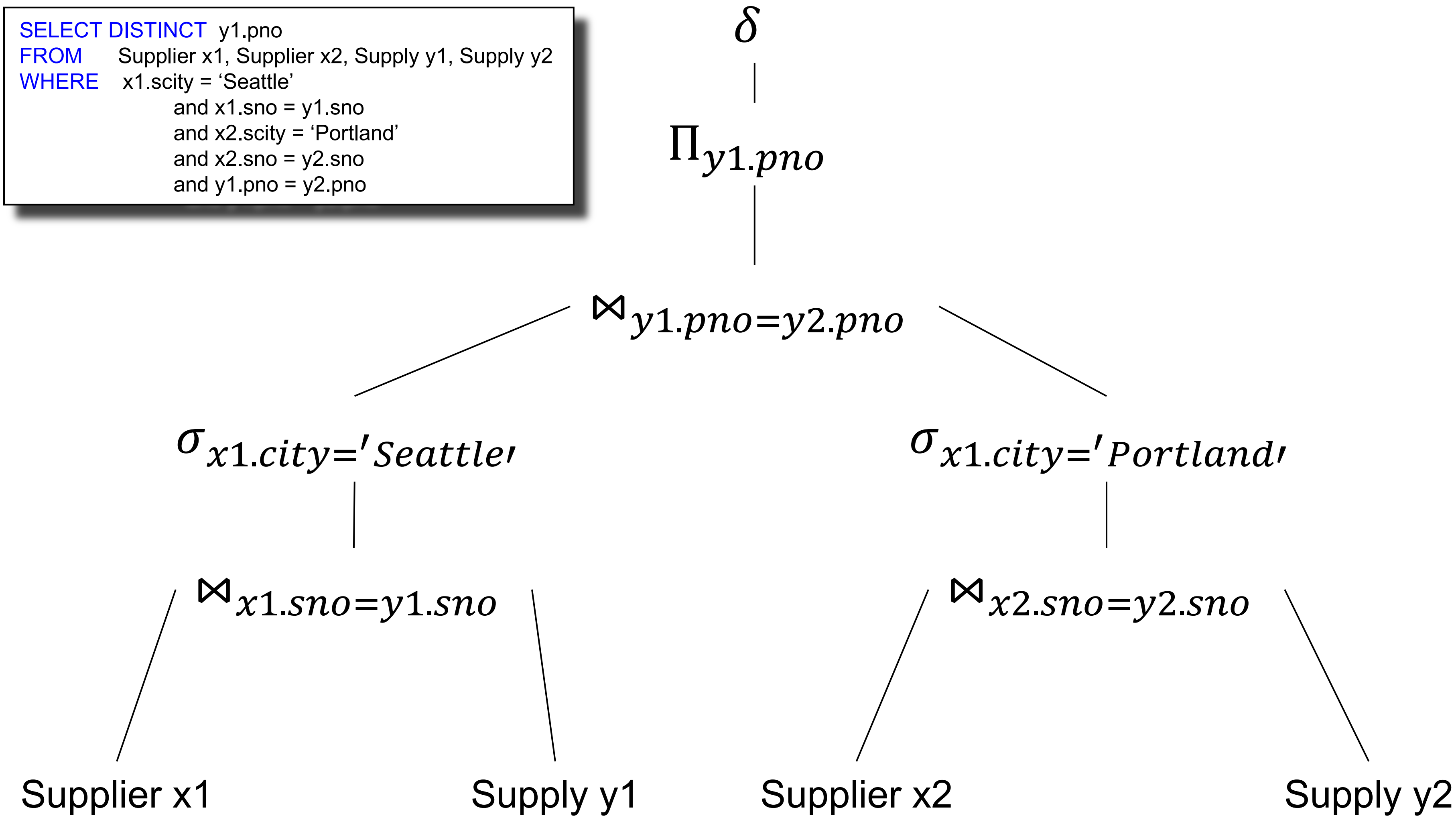
Supplier x2

Supply y2

Supplier(sno,sname,scity,sstate)
Supply(sno,pno,qty,price)
Part(pno,pname,psize,pcolor)

Convert SQL to Query Plan

```
SELECT DISTINCT  y1.pno
FROM      Supplier x1, Supplier x2, Supply y1, Supply y2
WHERE    x1.scity = 'Seattle'
                and x1.sno = y1.sno
                and x2.scity = 'Portland'
                and x2.sno = y2.sno
                and y1.pno = y2.pno
```



δ

$\Pi_{y1.pno}$

$\bowtie_{y1.pno=y2.pno}$

$\sigma_{x1.city='Seattle'}$

$\sigma_{x1.city='Portland'}$

$\bowtie_{x1.sno=y1.sno}$

$\bowtie_{x2.sno=y2.sno}$

Supplier x1

Supply y1

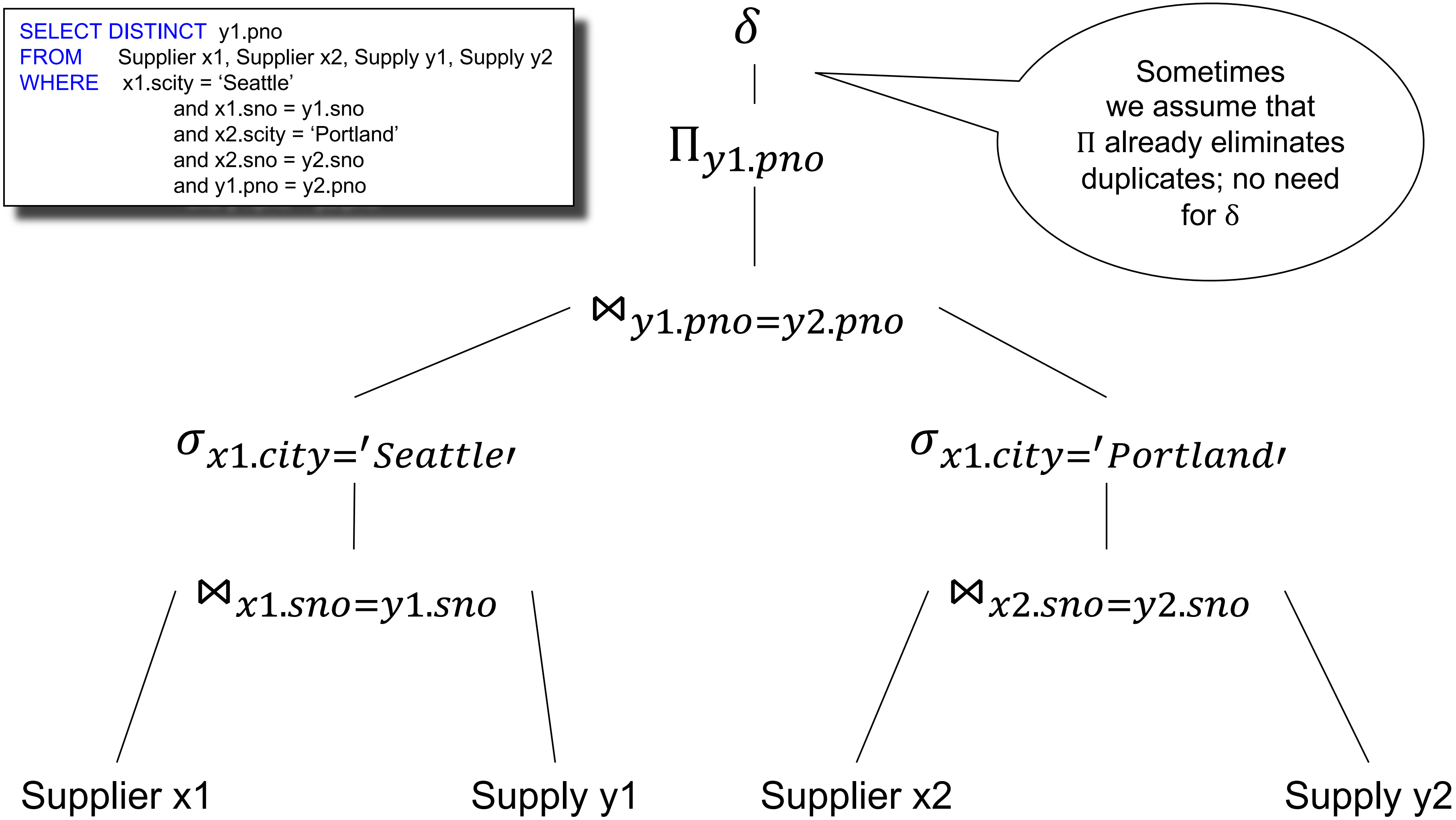
Supplier x2

Supply y2

```
Supplier(sno,sname,scity,sstate)
Supply(sno,pno,qty,price)
Part(pno,pname,psize,pcolor)
```

Convert SQL to Query Plan

```
SELECT DISTINCT  y1.pno
FROM      Supplier x1, Supplier x2, Supply y1, Supply y2
WHERE     x1.scity = 'Seattle'
            and x1.sno = y1.sno
            and x2.scity = 'Portland'
            and x2.sno = y2.sno
            and y1.pno = y2.pno
```



δ

$\Pi_{y1.pno}$

$\bowtie_{y1.pno=y2.pno}$

$\sigma_{x1.city='Seattle'}$

$\sigma_{x1.city='Portland'}$

$\bowtie_{x1.sno=y1.sno}$

$\bowtie_{x2.sno=y2.sno}$

Supplier x1

Supply y1

Supplier x2

Supply y2

Sometimes we assume that Π already eliminates duplicates; no need for δ

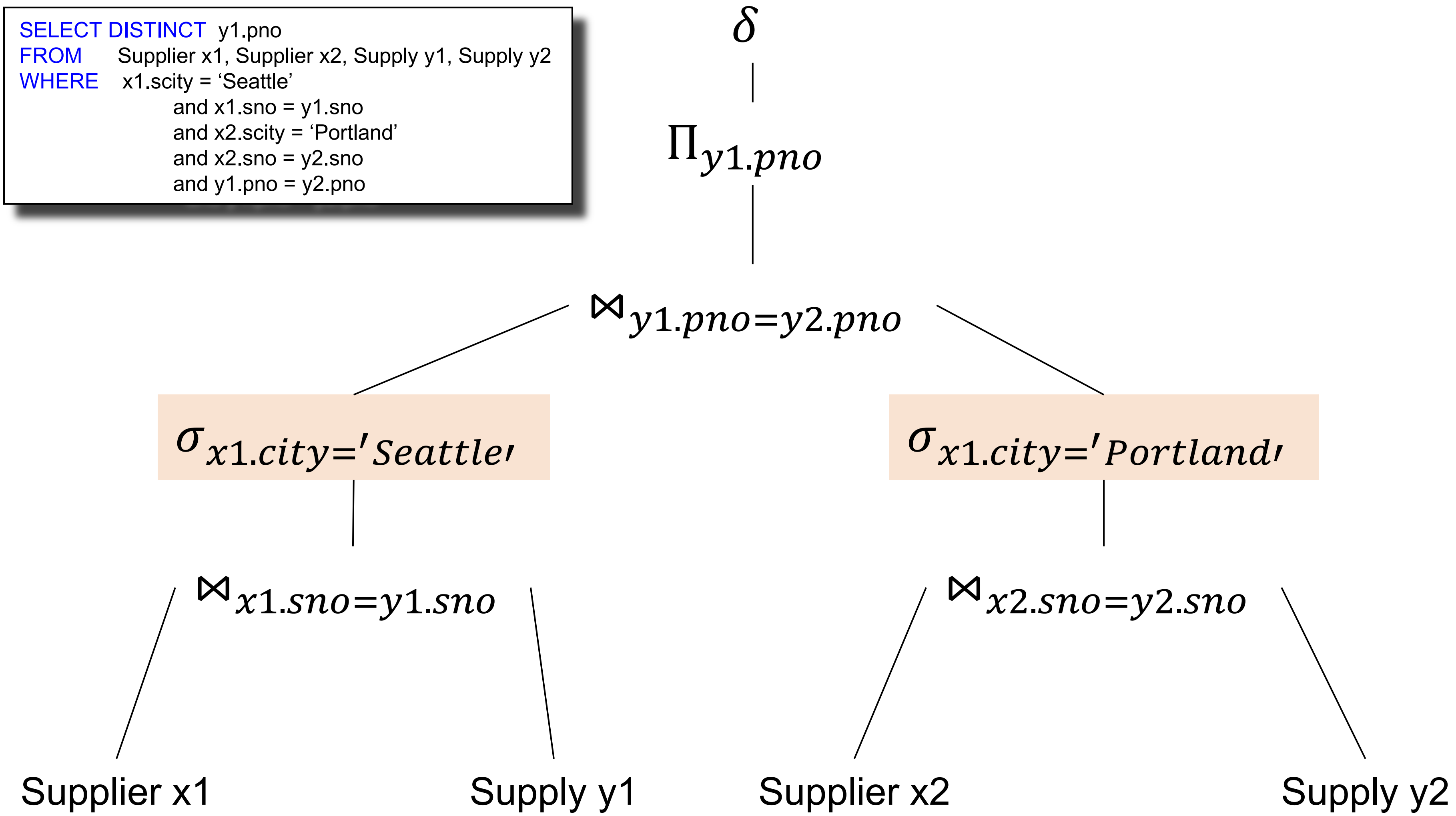
Optimize the Query Plan

- Heuristics:
  - Push selections down
  - Pull projections up
- Cost based:
  - Join reordering: dynamic programming
  - Rule based

```
Supplier(sno,sname,scity,sstate)
Supply(sno,pno,qty,price)
Part(pno,pname,psize,pcolor)
```

Push Selections Down

```
SELECT DISTINCT  y1.pno
FROM      Supplier x1, Supplier x2, Supply y1, Supply y2
WHERE    x1.scity = 'Seattle'
              and x1.sno = y1.sno
              and x2.scity = 'Portland'
              and x2.sno = y2.sno
              and y1.pno = y2.pno
```



δ

$\Pi_{y1.pno}$

$\bowtie_{y1.pno=y2.pno}$

$\sigma_{x1.city='Seattle'}$

$\sigma_{x1.city='Portland'}$

$\bowtie_{x1.sno=y1.sno}$

$\bowtie_{x2.sno=y2.sno}$

Supplier x1

Supply y1

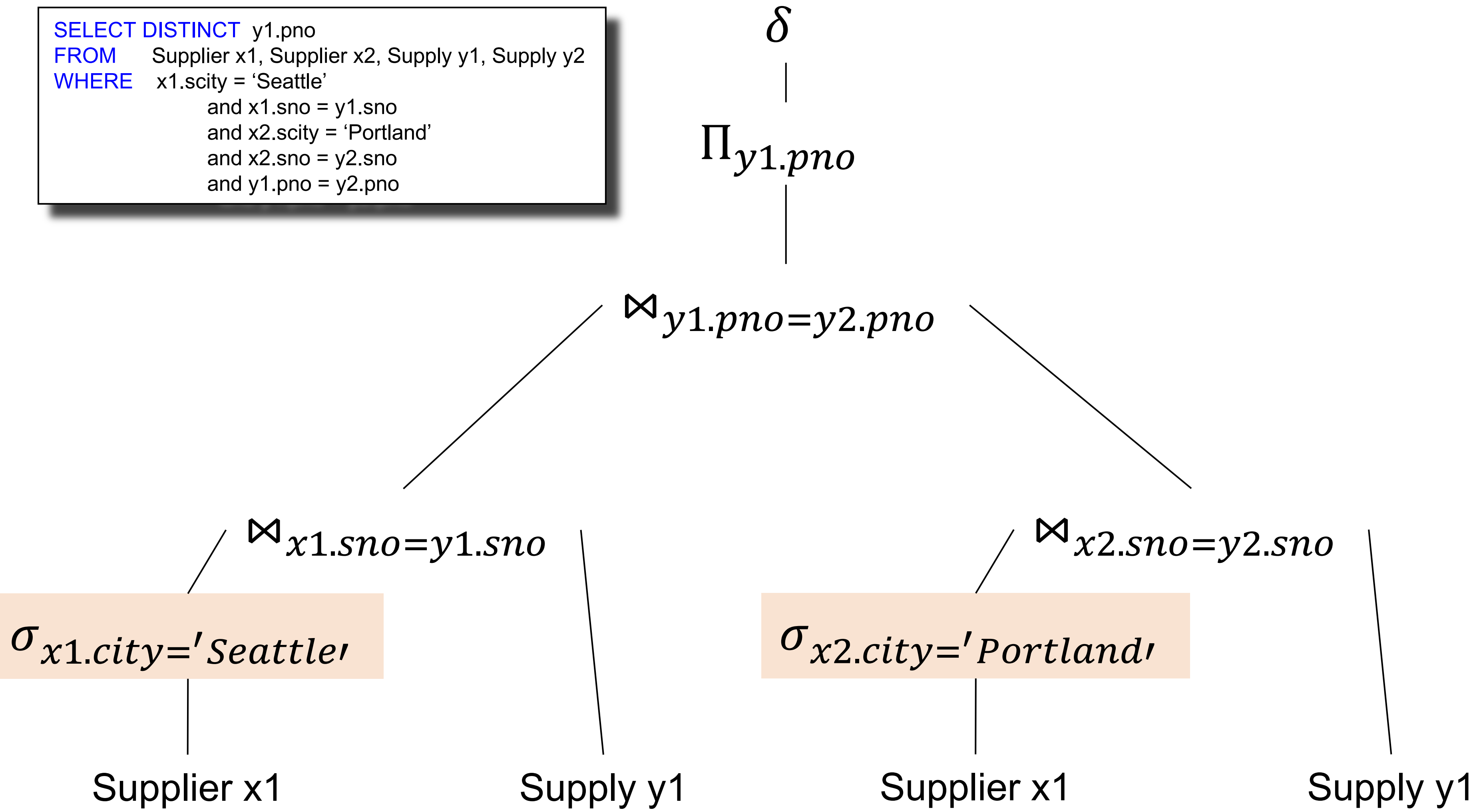
Supplier x2

Supply y2

`Supplier(sno,sname,scity,sstate)`
`Supply(sno,pno,qty,price)`
`Part(pno,pname,psize,pcolor)`

Push Selections Down

```
SELECT DISTINCT  y1.pno
FROM      Supplier x1, Supplier x2, Supply y1, Supply y2
WHERE    x1.scity = 'Seattle'
              and x1.sno = y1.sno
              and x2.scity = 'Portland'
              and x2.sno = y2.sno
              and y1.pno = y2.pno
```



δ

$\Pi_{y1.pno}$

$\bowtie_{y1.pno=y2.pno}$

$\bowtie_{x1.sno=y1.sno}$

$\bowtie_{x2.sno=y2.sno}$

$\sigma_{x1.city='Seattle'}$

$\sigma_{x2.city='Portland'}$

Supplier x1

Supply y1

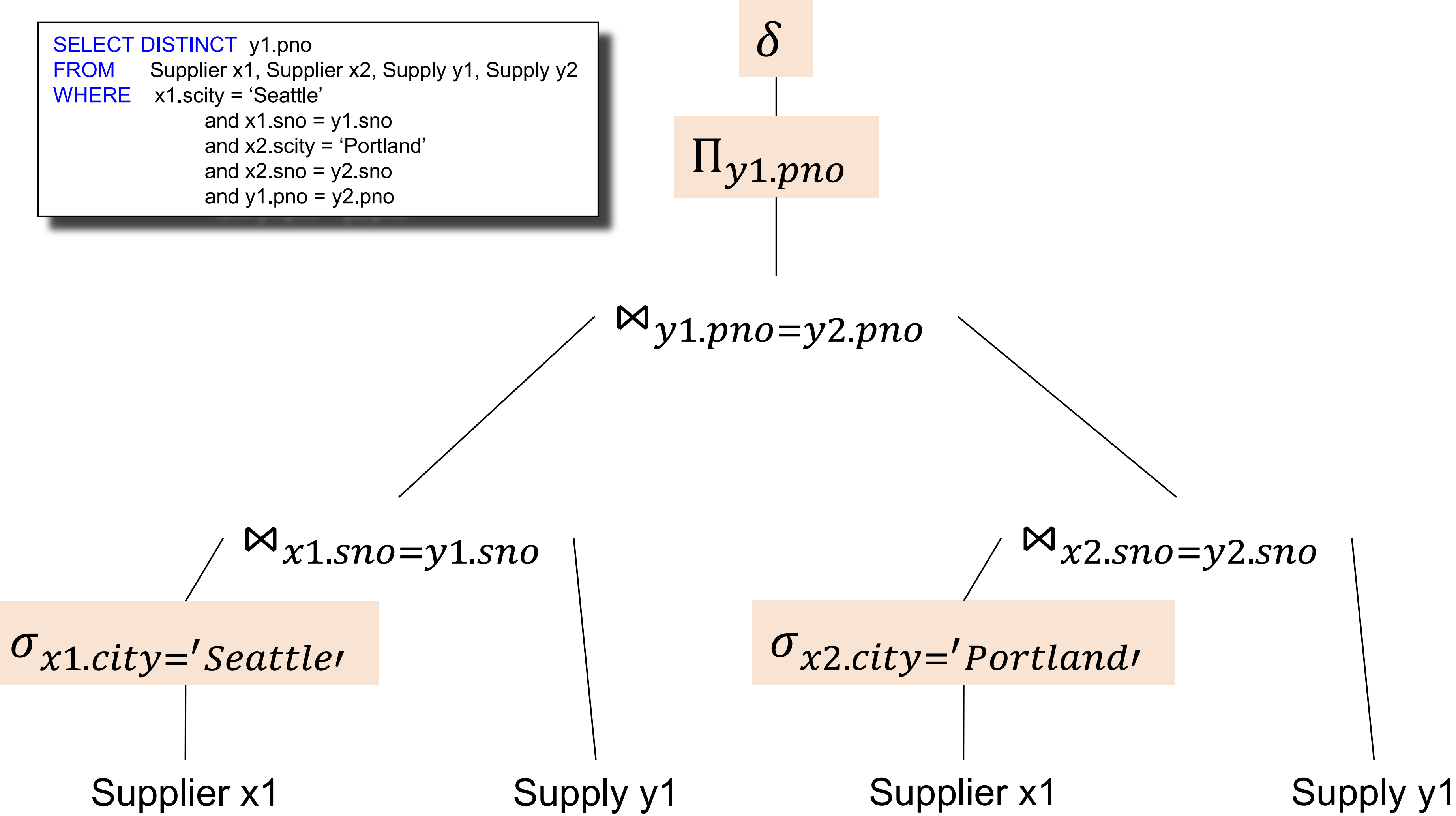
Supplier x1

Supply y1

```
Supplier(sno,sname,scity,sstate)
Supply(sno,pno,qty,price)
Part(pno,pname,psize,pcolor)
```

…and Pull Projections Up

```
SELECT DISTINCT  y1.pno
FROM      Supplier x1, Supplier x2, Supply y1, Supply y2
WHERE     x1.scity = 'Seattle'
             and x1.sno = y1.sno
             and x2.scity = 'Portland'
             and x2.sno = y2.sno
             and y1.pno = y2.pno
```



δ

$\Pi_{y1.pno}$

$\bowtie_{y1.pno=y2.pno}$

$\bowtie_{x1.sno=y1.sno}$

$\bowtie_{x2.sno=y2.sno}$

$\sigma_{x1.city='Seattle'}$

$\sigma_{x2.city='Portland'}$

Supplier x1

Supply y1

Supplier x1

Supply y1

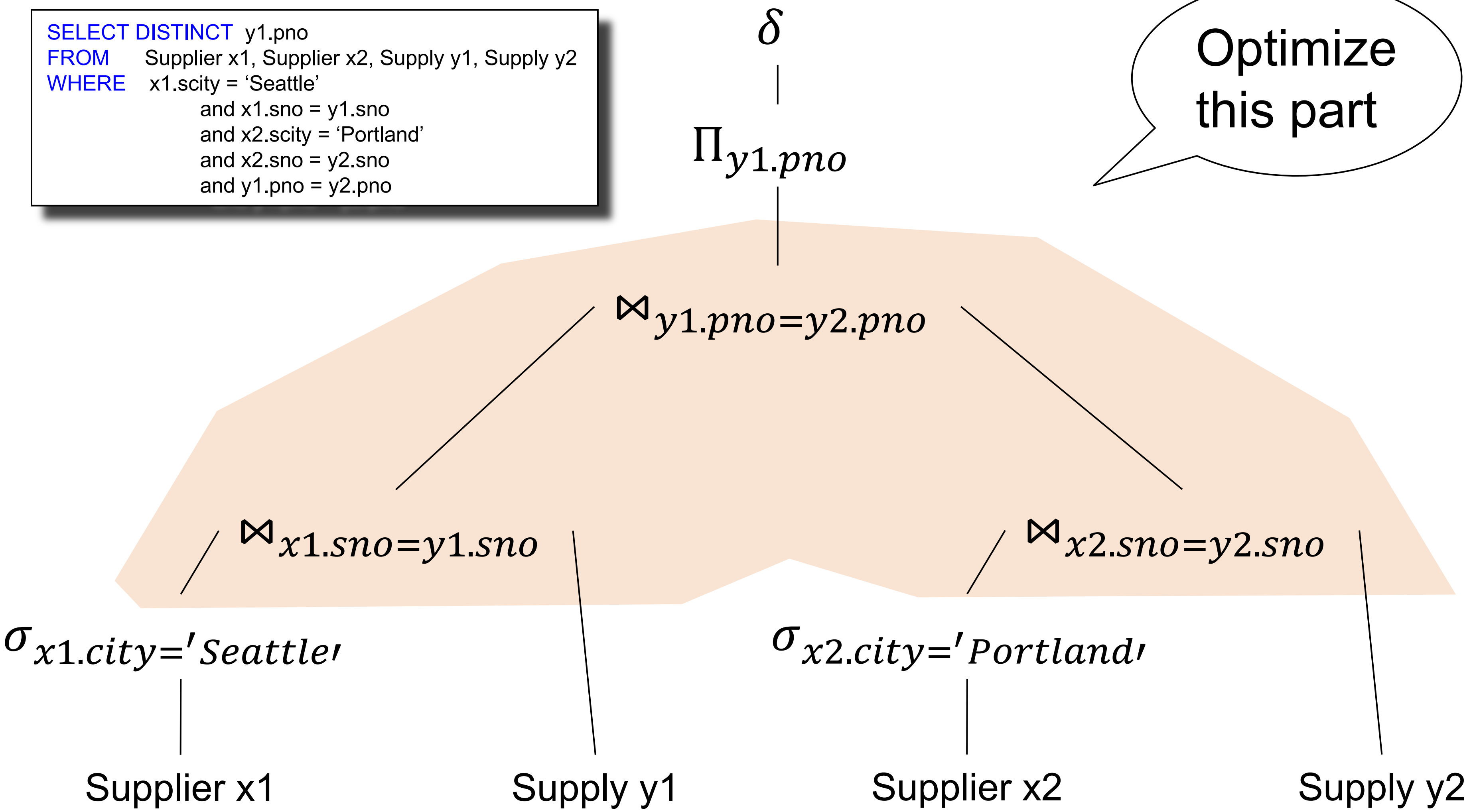
Optimize the Query Plan

- Heuristics:
  - Push selections down
  - Pull projections up
- Cost based:
  - Join reordering: dynamic programming
  - Rule based

Supplier(sno,sname,scity,sstate)
Supply(sno,pno,qty,price)
Part(pno,pname,psize,pcolor)

Join Reordering

```
SELECT DISTINCT  y1.pno
FROM      Supplier x1, Supplier x2, Supply y1, Supply y2
WHERE     x1.scity = 'Seattle'
              and x1.sno = y1.sno
              and x2.scity = 'Portland'
              and x2.sno = y2.sno
              and y1.pno = y2.pno
```



δ

$\Pi_{y1.pno}$

$\bowtie_{y1.pno=y2.pno}$

$\bowtie_{x1.sno=y1.sno}$

$\bowtie_{x2.sno=y2.sno}$

$\sigma_{x1.city='Seattle'}$

$\sigma_{x2.city='Portland'}$

Supplier x1

Supply y1

Supplier x2

Supply y2

Optimize this part

Joins Reordering

- It’s the bread and butter of query optimizers

- Performed using dynamic programming, a.k.a. Selinger’s algorithm

- Before we see this, let’s examine how joins are evaluated

Supplier(sid, sname, scity, sstate)
Supply(sid, pno, quantity)

Join Evaluation Algorithms

Logical operator:

Supplier $\bowtie_{sid=sid}$ Supply

Three algorithms:

1. Nested Loops
2. Hash-join
3. Merge-join

Supplier(<u>sid</u>, sname, scity, sstate)
Supply(<u>sid, pno</u>, quantity)

1. Nested Loop Join

Logical operator:

Supplier $\bowtie_{sid=sid}$ Supply

```
for x in Supplier do
    for y in Supply do
        if x.sid = y.sid
            then output(x,y)
```

Runtime $O(n^2)$

Supplier(sid, sname, scity, sstate)
Supply(sid, pno, quantity)

2. Hash Join

Logical operator:

Supply $\bowtie_{sid=sid}$ Supplier

Build phase

```
for x in Supplier do
    insert(x.sid, x)
```

Supplier(sid, sname, scity, sstate)
Supply(sid, pno, quantity)

2. Hash Join

Logical operator:

Supply $\bowtie_{sid=sid}$ Supplier

Build phase

```
for x in Supplier do
      insert(x.sid, x)

for y in Supply do
      x = find(y.sid);
      output(x,y);
```

Probe phase

Supplier(<u>sid</u>, sname, scity, sstate)
Supply(<u>sid, pno</u>, quantity)

2. Hash Join

Logical operator:

Supply $\bowtie_{sid=sid}$ Supplier

Build phase

```
for x in Supplier do
        insert(x.sid, x)
```

Probe phase

```
for y in Supply do
        x = find(y.sid);
        output(x,y);
```

Runtime O(n)

Supplier(sid, sname, scity, sstate)
Supply(sid, pno, quantity)

2. Hash Join

Change join order

Logical operator:

Supplier $\bowtie_{sid=sid}$ Supply

```
for y in Supply do
    insert(y.sid, y)

for x in Supplier do ??
```

Supplier(sid, sname, scity, sstate)
Supply(sid, pno, quantity)

2. Hash Join

Change join order

Logical operator:

Supplier $\bowtie_{sid=sid}$ Supply

```
for y in Supply do
      insert(y.sid, y)

for x in Supplier do
    for y in find(x.sid) do
        output(x,y);
```

Supplier(sid, sname, scity, sstate)
Supply(sid, pno, quantity)

2. Hash Join

Change join order

Logical operator:

Supplier $\bowtie_{sid=sid}$ Supply

```
for y in Supply do
      insert(y.sid, y)

for x in Supplier do
    for y in find(x.sid) do
        output(x,y);
```

Runtime can be $O(n^2)$ because Supply.sid is not a key and there may be many duplicates

Supplier(sid, sname, scity, sstate)
Supply(sid, pno, quantity)

3. Merge Join

Logical operator:

Supplier $\bowtie_{sid=sid}$ Supply

```
Sort(Supplier); Sort(Supply);
x = Supplier.first();
y = Supply.first();
```

Supplier(sid, sname, scity, sstate)
Supply(sid, pno, quantity)

3. Merge Join

Logical operator:

Supplier $\bowtie_{sid=sid}$ Supply

```
Sort(Supplier); Sort(Supply);
x = Supplier.first();
y = Supply.first();
while y != NULL do
  case:
    x.sid < y.sid: ???
    x.sid = y.sid: ???
    x.sid > y.sid: ???
```

Supplier(sid, sname, scity, sstate)
Supply(sid, pno, quantity)

3. Merge Join

Logical operator:

Supplier $\bowtie_{sid=sid}$ Supply

```
Sort(Supplier); Sort(Supply);
x = Supplier.first();
y = Supply.first();
while y != NULL do
  case:
    x.sid < y.sid: x = x.next()
    x.sid = y.sid: ???
    x.sid > y.sid: ???
```

Supplier(sid, sname, scity, sstate)
Supply(sid, pno, quantity)

3. Merge Join

Logical operator:

Supplier $\bowtie_{sid=sid}$ Supply

```
Sort(Supplier); Sort(Supply);
x = Supplier.first();
y = Supply.first();
while y != NULL do
  case:
    x.sid < y.sid: x = x.next()
    x.sid = y.sid: output(x,y); y = y.next();
    x.sid > y.sid: ???
```

Supplier(sid, sname, scity, sstate)
Supply(sid, pno, quantity)

3. Merge Join

Logical operator:

Supplier $\bowtie_{sid=sid}$ Supply

```
Sort(Supplier); Sort(Supply);
x = Supplier.first();
y = Supply.first();
while y != NULL do
  case:
    x.sid < y.sid: x = x.next()
    x.sid = y.sid: output(x,y); y = y.next();
    x.sid > y.sid: y = y.next();
```

Supplier(sid, sname, scity, sstate)
Supply(sid, pno, quantity)

3. Merge Join

Logical operator:

Supplier $\bowtie_{sid=sid}$ Supply

```
Sort(Supplier); Sort(Supply);
x = Supplier.first();
y = Supply.first();
while y != NULL do
  case:
    x.sid < y.sid: x = x.next()
    x.sid = y.sid: output(x,y); y = y.next();
    x.sid > y.sid: y = y.next();
```

Runtime O(n log(n))
(because sorting…)

Discussion

- Joins = most studied relational operator
- Variations:
  - Blocking (materialize) v.s. pipelining
  - Main memory join v.s. external memory
  - Single server v.s. distributed

Join Ordering

Optimize the Query Plan

- Heuristics:
  - Push selections down
  - Pull projections up

- Cost based:
  - Join reordering: dynamic programming
  - Rule based

Join Reordering

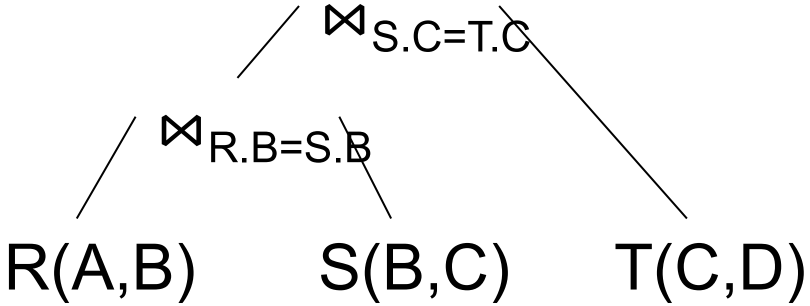
- Dynamic programming
- Introduced by Selinger, “System R”, 79
- Also called Selinger’s algorithm
- Originally restricted to:
  - Left-deep plans
  - No cartesian products

Cartesian Products

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

Cartesian Products

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



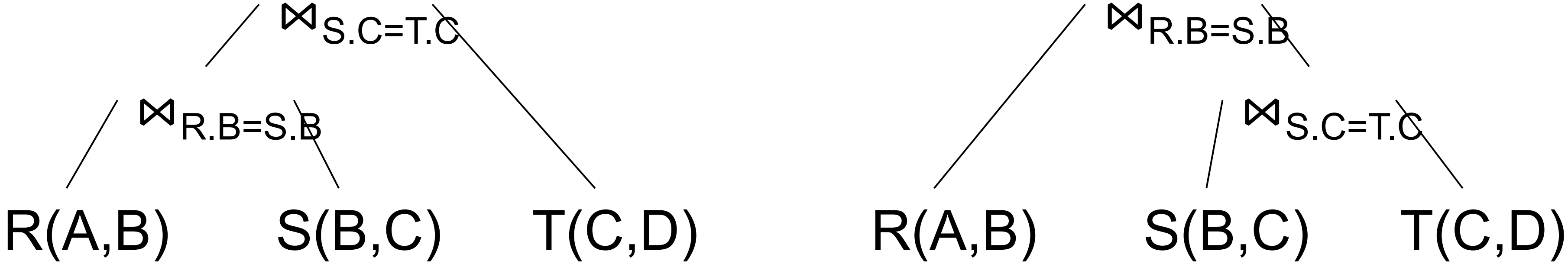
$\bowtie_{S.C=T.C}$

$\bowtie_{R.B=S.B}$

R(A,B) S(B,C) T(C,D)

Cartesian Products

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



$\bowtie_{S.C=T.C}$

$\bowtie_{R.B=S.B}$

R(A,B) S(B,C) T(C,D)

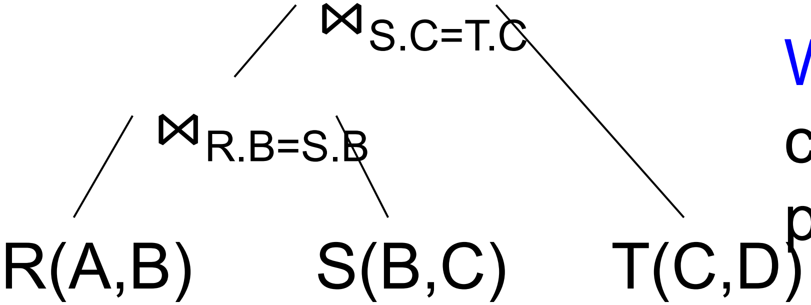
$\bowtie_{R.B=S.B}$

$\bowtie_{S.C=T.C}$

R(A,B) S(B,C) T(C,D)

Cartesian Products

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

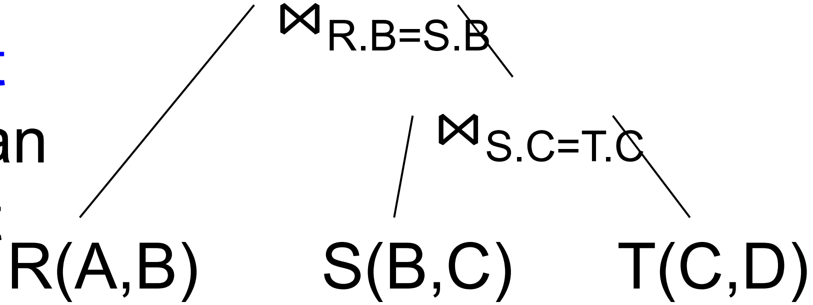


$\bowtie_{S.C=T.C}$

$\bowtie_{R.B=S.B}$

$R(A,B)$ $S(B,C)$ $T(C,D)$

Without cartesian product



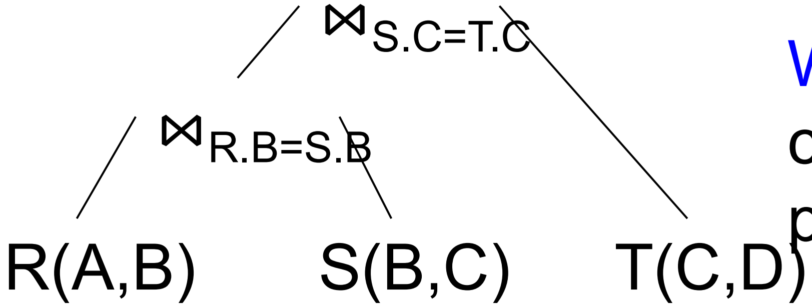
$\bowtie_{R.B=S.B}$

$\bowtie_{S.C=T.C}$

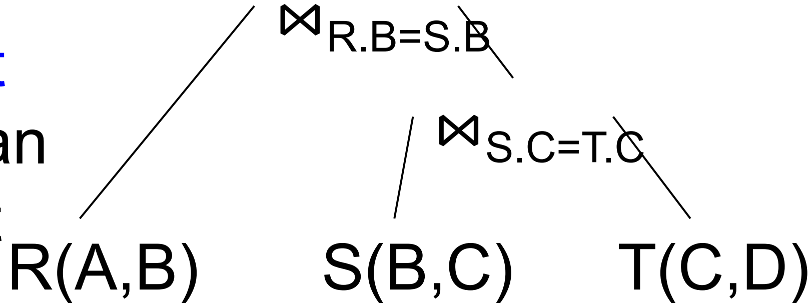
$R(A,B)$ $S(B,C)$ $T(C,D)$

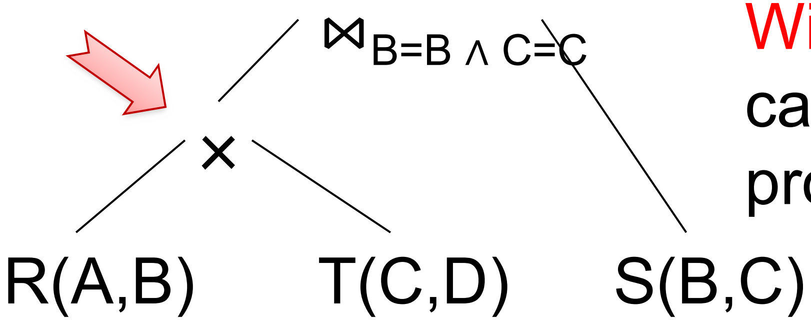
Cartesian Products

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



Without cartesian product

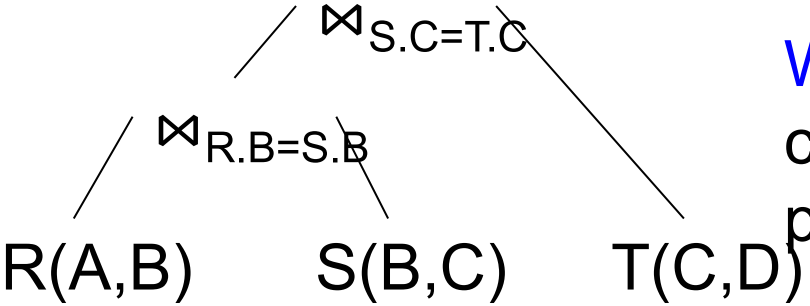




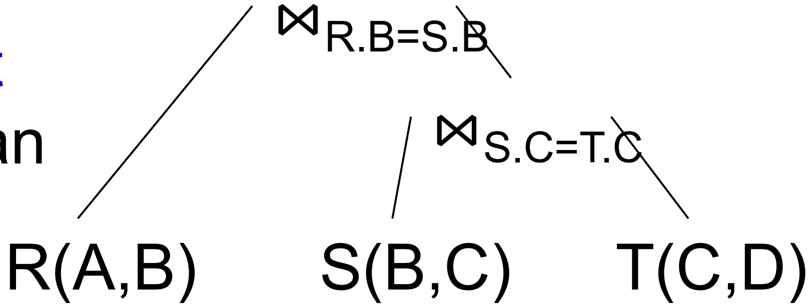
With cartesian product

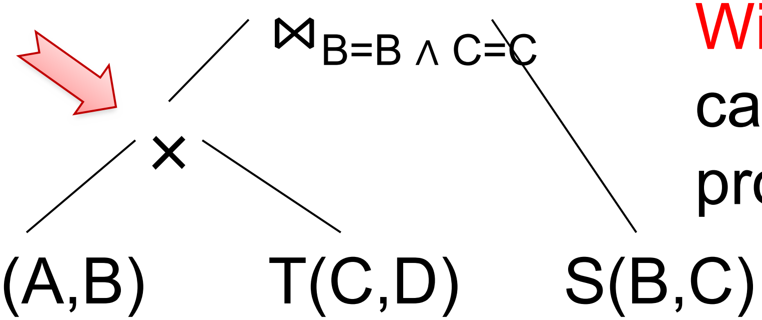
Cartesian Products

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



Without cartesian product



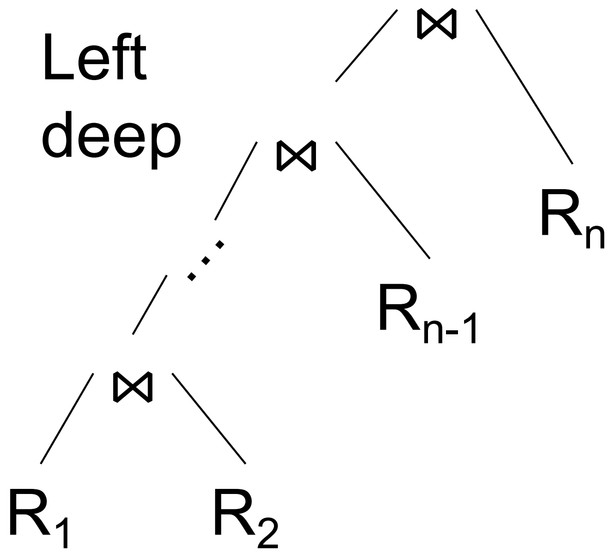


With cartesian product

When could this plan be better?

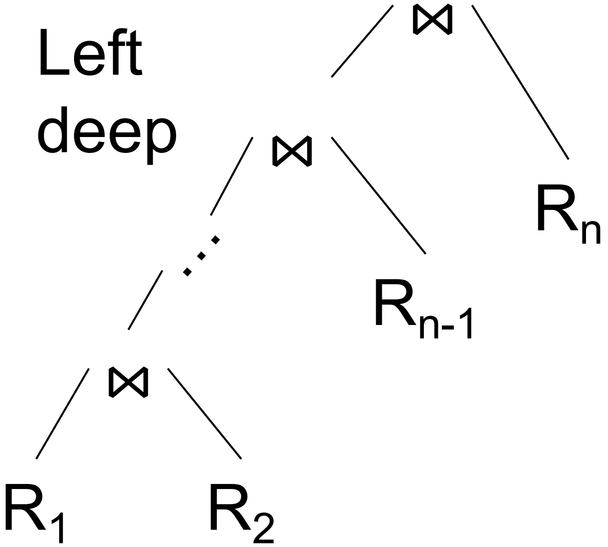
Shapes of Join Trees

Left
deep



Shapes of Join Trees

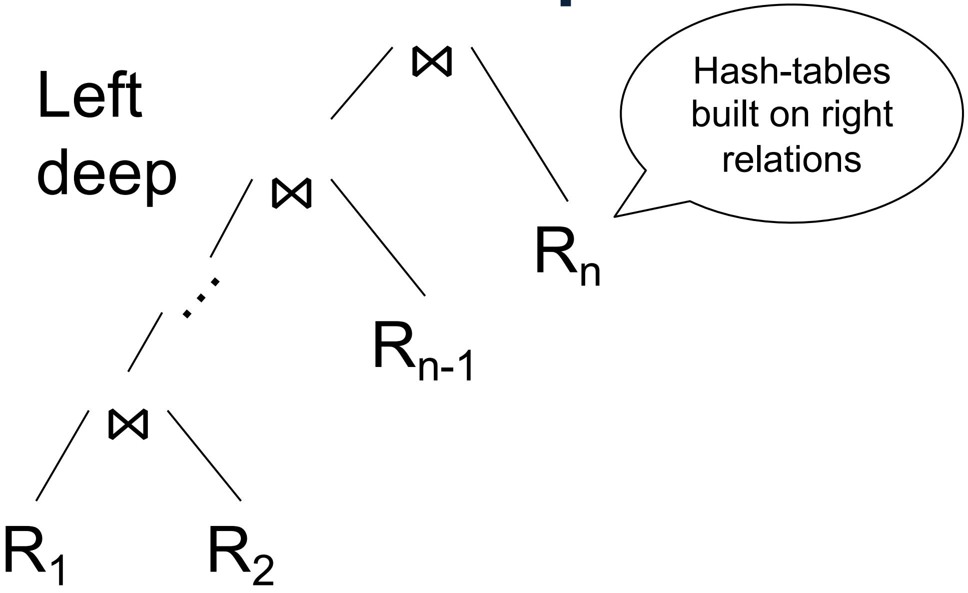
Left
deep



Hash-tables built on right relations

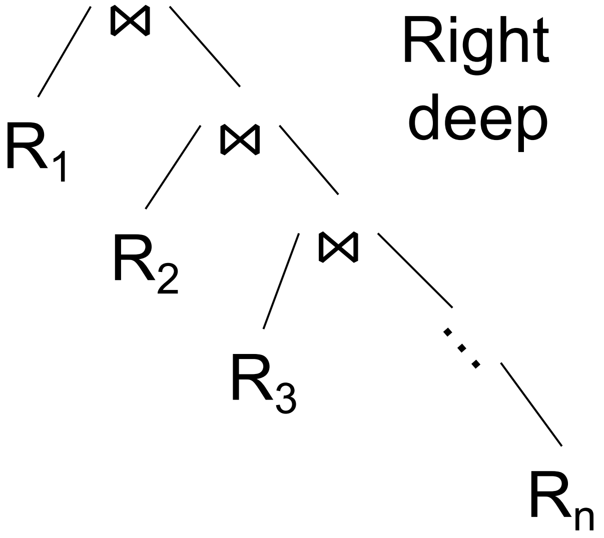
Shapes of Join Trees

Left deep



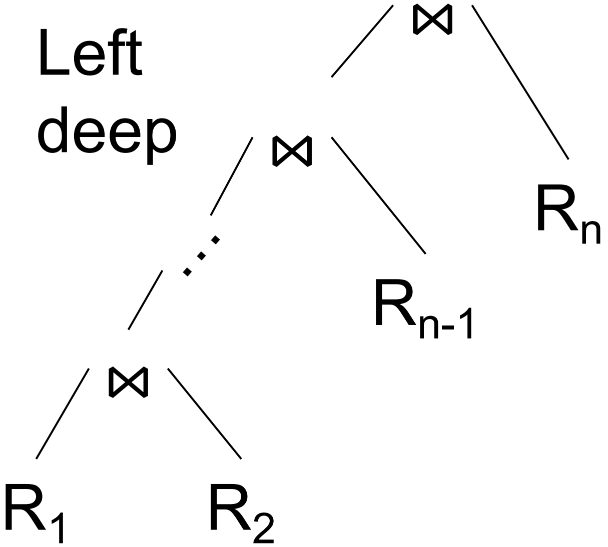
Hash-tables built on right relations

Right deep



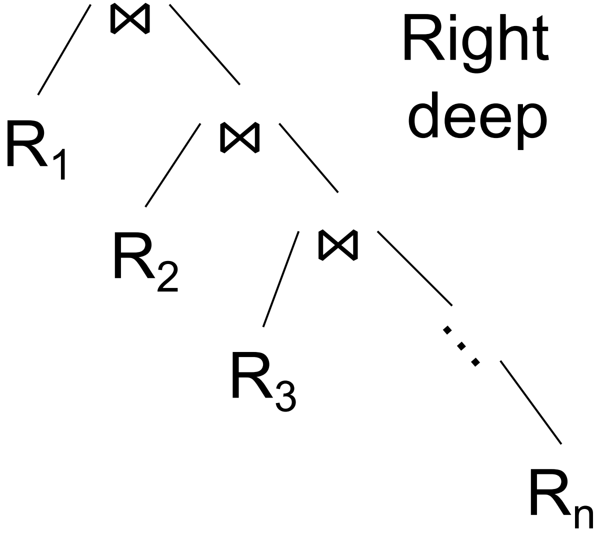
Shapes of Join Trees

Left deep

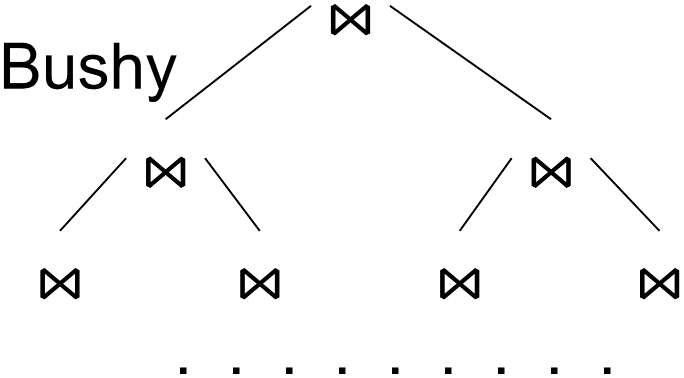


Hash-tables built on right relations

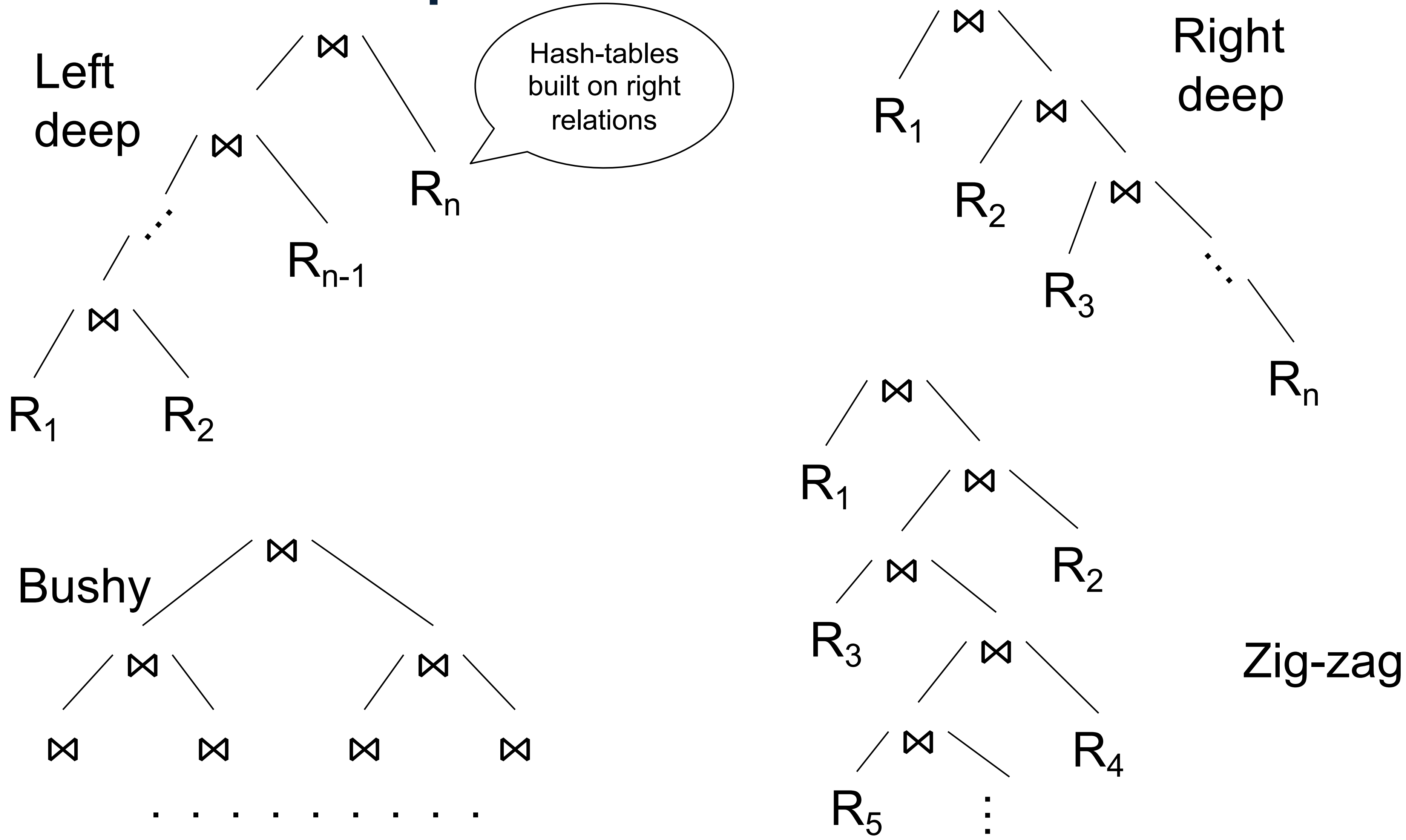
Right deep



Bushy



Shapes of Join Trees



Dynamic Programming

- Join order: a misnomer, since we are not just ordering, but we compute a tree
- Main idea: compute optimal join order for every subset of relations
- With or without cartesian products
  With or without restricting tree shapes

Dynamic Programming

- Let m = number of relations to join
- For s = 1, m do:
  - For each subset S of of size s do:
    - Split S into [relation R] + [set of s-1 relations S’]
    - Lookup Cost(S’)
    - Cost(S) := $\min_{\text{splits}}$ (Cost(S’) + cost-of($R \bowtie S'$))
    - Memorize (S, Cost(S))
- Return Cost(All-relations)

Discussion

- Dynamic programming: exponential in # of relations; works for up to 10-20 rels
- Variations:
  - “Interesting orders” for merge-join
  - With or without cartesian product
  - Left-, right-, bushy-, zig-zag plans
  - Outerjoins? Anti-semijoins?

NULLs in SQL

NULLs in SQL

- A NULL value means missing, or unknown, or undefined, or inapplicable

NULLs in WHERE Clause

A WHERE clause contains a predicate:

- Expr1 op Expr2

How do we compute the predicate when values are NULL?

Example

where price < 100 and (pcolor='red' or psize=2)

SQL Has Three-Valued Logic

- False=0, Unknown=0.5, True=1

SQL Has Three-Valued Logic

- False=0, Unknown=0.5, True=1

- A = B, A < B, …: **Unknown**, if either A or B is NULL
  AND, OR, NOT: **min**, **max**, and 1- …

SQL Has Three-Valued Logic

- False=0, Unknown=0.5, True=1

- A = B, A < B, …: **Unknown**, if either A or B is NULL
  AND, OR, NOT: **min**, **max**, and 1- …
- Return only tuples whose condition is **True**

SQL Has Three-Valued Logic

- False=0, Unknown=0.5, True=1

- A = B, A < B, …: **Unknown**, if either A or B is NULL
  AND, OR, NOT: **min**, **max**, and 1- …
- Return only tuples whose condition is **True**

- E.g. price < 100: can be False, Unkown, or True

SQL Has Three-Valued Logic

- False=0, Unknown=0.5, True=1

- A = B, A < B, …: **Unknown**, if either A or B is NULL
  AND, OR, NOT: **min**, **max**, and 1- …
- Return only tuples whose condition is **True**

- E.g. price < 100: can be False, Unkown, or True

- What about (price < 100) and (pcolor = ‘red’)?

SQL Has Three-Valued Logic

```
select *
from Part
where price < 100
 and (psize=2 or pcolor='red')
```

pno	pname	price	psize	pcolor
1	iPad	500	13	blue
2	Scooter	99	NULL	NULL
3	Charger	NULL	NULL	red
4	iPad	50	2	NULL

(in class: discuss which tuples are returned)

SQL Has Three-Valued Logic

Problem:
does *not* return
all records!

```
select *
from Part
where (price <= 100) or (price > 100)
```

SQL Has Three-Valued Logic

Problem:
does *not* return
all records!

```
select *
from Part
where (price <= 100) or (price > 100)
```

```
select *
from Part
where (price <= 100) or (price > 100) or isNull(price)
```

Now it does

Discussion

NULLs and their 3-valued logic are a major headache for query optimizers:

- (A and not(A)) ≠ True
- Aggregates need special cases
- Outerjoins are not commutative, etc

Aggregates in SQL

```
Supplier(sno,sname,scity,sstate)
Supply(sno,pno,qty,price)
Part(pno,pname,psize,pcolor)
```

Aggregates

```
SELECT count(*)
FROM      Part
```

```
Supplier(sno,sname,scity,sstate)
Supply(sno,pno,qty,price)
Part(pno,pname,psize,pcolor)
```

Aggregates

```
SELECT count(*)
FROM   Part
```

For each city, compute the average size of parts supplied from that city.

```
Supplier(sno,sname,scity,sstate)
Supply(sno,pno,qty,price)
Part(pno,pname,psize,pcolor)
```

Aggregates

```
SELECT count(*)
FROM      Part
```

```
SELECT x.scity, avg(psize)
FROM     Supplier x, Supply y, Part z
WHERE    x.sno = y.sno  and y.pno = z.pno
GROUP BY x.scity
```

For each city, compute the average size of parts supplied from that city.

```
Supplier(sno,sname,scity,sstate)
Supply(sno,pno,qty,price)
Part(pno,pname,psize,pcolor)
```

Aggregates

```
SELECT count(*)
FROM   Part
```

```
SELECT x.scity, avg(psize)
FROM   Supplier x, Supply y, Part z
WHERE  x.sno = y.sno  and y.pno = z.pno
GROUP BY x.scity
```

For each city, compute the average size of parts supplied from that city.

…but only for cities that supply > 200 parts

```
Supplier(sno,sname,scity,sstate)
Supply(sno,pno,qty,price)
Part(pno,pname,psize,pcolor)
```

Aggregates

```
SELECT count(*)
FROM     Part
```

```
SELECT x.scity, avg(psize)
FROM    Supplier x, Supply y, Part z
WHERE   x.sno = y.sno  and y.pno = z.pno
GROUP BY x.scity
```

For each city, compute the average size of parts supplied from that city.

```
SELECT x.scity, avg(psize)
FROM    Supplier x, Supply y, Part z
WHERE   x.sno = y.sno  and y.pno = z.pno
GROUP BY x.scity
HAVING count(*) > 200
```

…but only for cities that supply > 200 parts

Aggregates

- Semantics:
  - FROM-WHERE (nested-loop semantics)
  - Group answers by GROUP BY attrs
  - Apply HAVING predicates on groups
  - Apply SELECT aggregates on groups

- Aggregate functions:
  - count, sum, min, max, avg

Relational Algebra

- Group-by:

$$\gamma_{attrbutes, agg(A_1)\rightarrow B_1, agg(A_2)\rightarrow B_2, \ldots}$$

Rule-based Optimization

- Collection of rewrite rules:

  $E_1 = E_1'$
  $E_2 = E_2'$
  …
- Given a query plan P, apply rules repeatedly, to generate equivalent plans:

  $P = P_1 = P_2 = P_3 = \ldots$

- Return the plan with lowest cost

Examples of rules

$$(R \bowtie S) \bowtie T = R \bowtie (S \bowtie T)$$

$$\gamma(R \bowtie S) = \gamma(R \bowtie \gamma(S))$$

$$\gamma(R \cup S) = \gamma(\gamma(R) \cup \gamma(S))$$

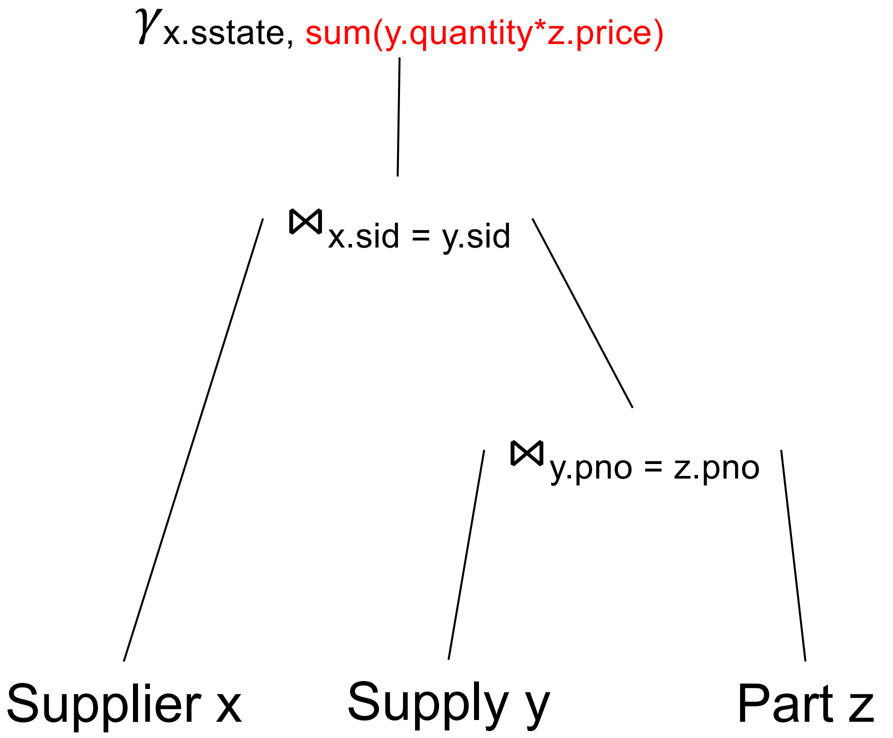
```
Supplier(sid, sname, scity, sstate)
Supply(sid, pno, quantity)
Part(pno, pname, pprice)
```

Aggregate Push-down

```
SELECT x.sstate, sum(y.quanity*z.price)
FROM Supplier x, Supply y, Part z
WHERE x.sid = y.sid and y.pno = z.pno
GROUP BY x.sstate
```

Aggregate Push-down

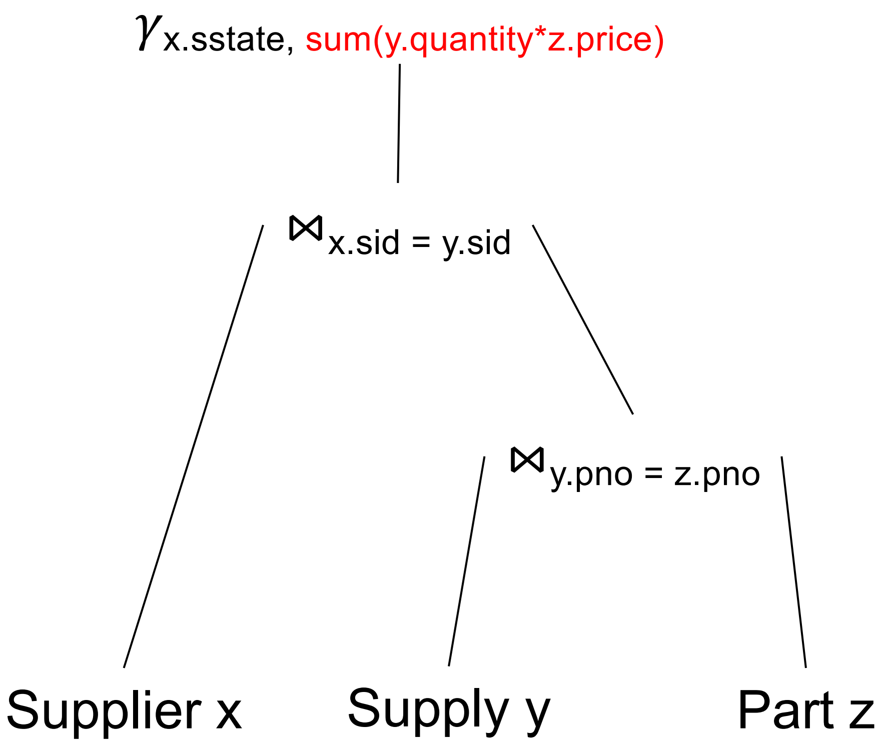
Supplier(sid, sname, scity, sstate)
Supply(sid, pno, quantity)
Part(pno, pname, pprice)



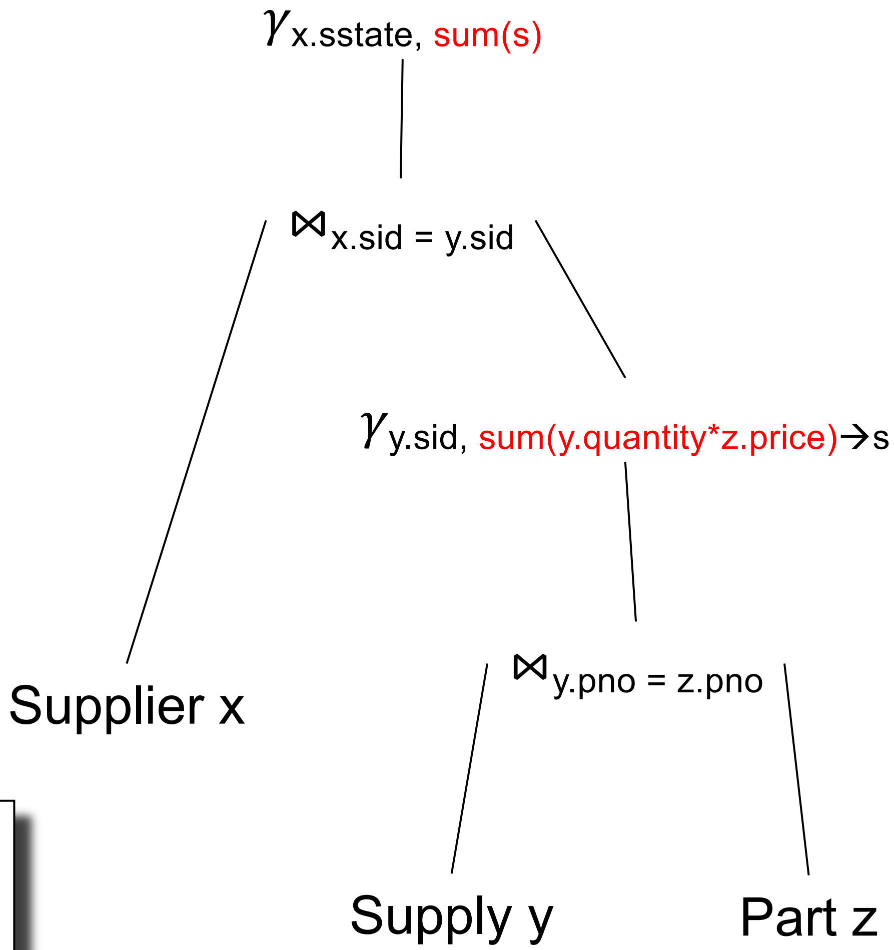
```sql
SELECT x.sstate, sum(y.quanity*z.price)
FROM Supplier x, Supply y, Part z
WHERE x.sid = y.sid and y.pno = z.pno
GROUP BY x.sstate
```

Aggregate Push-down

Supplier(sid, sname, scity, sstate)
Supply(sid, pno, quantity)
Part(pno, pname, pprice)



```sql
SELECT x.sstate, sum(y.quanity*z.price)
FROM Supplier x, Supply y, Part z
WHERE x.sid = y.sid and y.pno = z.pno
GROUP BY x.sstate
```



Discussion

- Rule-based optimizer introduced by Graefe in the Volcano system, at Wisconsin
- Later refined by Graefe into the CASCADES framework → SQL Server
- Most modern systems use rule-based optimizers
- EGG = open-source equality saturation system

Outer Joins

`Supplier(sno,sname,scity,sstate)`
`Supply(sno,pno,qty,price)`

Outer joins

Compute the number of products sold by each supplier

```sql
SELECT x.sno, x.sname, count(*)
FROM   Supplier x, Supply y
WHERE  x.sno = y.sno
GROUP BY x.sno, x.sname
```

Problem: suppliers with 0 products are not included.

`Supplier(sno,sname,scity,sstate)`
`Supply(sno,pno,qty,price)`

Outer joins

Compute the number of products sold by each supplier

```
SELECT x.sno, x.sname, count(y.sno)
FROM  Supplier x LEFT OUTER JOIN Supply y
  ON  x.sno = y.sno
GROUP BY x.sno, x.sname
```

Now they are included

Left Outer Join (Details)

from R left outer join S on C1 where C2

1. Compute cross product R×S
2. Filter on C1
3. Add all R records without a match
4. Filter on C2

Joins

- **Inner join** = includes only matching tuples (i.e. regular join)
- **Left outer join** = includes everything from the left
- **Right outer join** = includes everything from the right
- **Full outer join** = includes everything

Relational Algebra

- Left outer join: ⟕
- Right outer join: ⟖
- Full outer join: ⟗

Supplier(sid, sname, scity, sstate)
Supply(sid, pno, quantity)

Hash-based Left Outer Join

Supplier $⟕_{sid=sid}$ Supply

```
for x in Supplier do
     insert(x.sid, x)

for y in Supply do
     x = find(y.sid);
     y.found = true
     output(x,y);

for x in Supplier do
     if not x.found
        then output(x,NULL)
```

Discussion

- Left outer join:
  - Very useful for one-to-many relationships
  - Eg each Supplier has 0 or more Supply
  - Eg each Student takes 0 or more Courses
- Right outer join, full outer join: rarely used
- Major pain for optimization

Subqueries in SQL

Subqueries

- Subquery in SELECT:
  - Must return single value
- Subquery in FROM
  - Like a temporary relation
  - Alternative: use the WITH clause
- Subquery in WHERE or in HAVING
  - Can express sophisticated queries

```
Supplier(sno,sname,scity,sstate)
Supply(sno,pno,qty,price)
Part(pno,pname,psize,pcolor)
```

Subquery in SELECT

Compute the number of products sold by each supplier

```
SELECT x.sno, x.sname,
     (SELECT count(*)
      FROM Supply y
      WHERE x.sno = y.sno)
FROM  Supplier x
```

```
Supplier(sno,sname,scity,sstate)
Supply(sno,pno,qty,price)
Part(pno,pname,psize,pcolor)
```

Subquery in FROM

Better: use the WITH statement!

```
Supplier(sno,sname,scity,sstate)
Supply(sno,pno,qty,price)
Part(pno,pname,psize,pcolor)
```

Subquery in FROM

Better: use the WITH statement!

Find the supplier who supplies the maximum number of parts

```
Supplier(sno,sname,scity,sstate)
Supply(sno,pno,qty,price)
Part(pno,pname,psize,pcolor)
```

Subquery in FROM

Better: use the WITH statement!

Find the supplier who supplies the maximum number of parts

```
WITH Cnt AS (SELECT x.sno, x.sname, count(*) as c
             FROM Supplier x, Supply y
             WHERE x.sno = y.sno
             GROUP BY x.sno)
```

For each supplier, compute how many parts they supply

Supplier(sno,sname,scity,sstate)
Supply(sno,pno,qty,price)
Part(pno,pname,psize,pcolor)

Subquery in FROM

Better: use the WITH statement!

Find the supplier who supplies the maximum number of parts

```
WITH Cnt AS (SELECT x.sno, x.sname, count(*) as c
             FROM Supplier x, Supply y
             WHERE x.sno = y.sno
             GROUP BY x.sno),
     Mx AS (SELECT max(c) as m
            FROM Cnt)
```

Find the maximum

```
Supplier(sno,sname,scity,sstate)
Supply(sno,pno,qty,price)
Part(pno,pname,psize,pcolor)
```

Subquery in FROM

Better: use the WITH statement!

Find the supplier who supplies the maximum number of parts

```
WITH Cnt AS (SELECT x.sno, x.sname, count(*) as c
             FROM Supplier x, Supply y
             WHERE x.sno = y.sno
             GROUP BY x.sno),
     Mx AS (SELECT max(c) as m
            FROM Cnt)
SELECT z.sno, z.sname, m.m
FROM  Cnt z, Mx m
WHERE z.c = m.m;
```

Find the “witness”,
i.e. the supplier that
supplies the maximum
number of parts; **argmax**

```
Supplier(sno,sname,scity,sstate)
Supply(sno,pno,qty,price)
Part(pno,pname,psize,pcolor)
```

Subquery in WHERE

Find suppliers that supply some ‘blue’ parts

```
Supplier(sno,sname,scity,sstate)
Supply(sno,pno,qty,price)
Part(pno,pname,psize,pcolor)
```

Subquery in WHERE

Find suppliers that supply some ‘blue’ parts

```
SELECT x.sno
FROM Supplier x
WHERE exists (SELECT * FROM Supply y, Part z
              WHERE x.sno=y.sno
                and y.pno=z.pno
                and z.pcolor = ‘blue’);
```

```
Supplier(sno,sname,scity,sstate)
Supply(sno,pno,qty,price)
Part(pno,pname,psize,pcolor)
```

Subquery in WHERE

Find suppliers that supply *only* ‘red’ parts

```
Supplier(sno,sname,scity,sstate)
Supply(sno,pno,qty,price)
Part(pno,pname,psize,pcolor)
```

Subquery in WHERE

Find suppliers that supply *only* ‘red’ parts

Find the *other* suppliers

```
Supplier(sno,sname,scity,sstate)
Supply(sno,pno,qty,price)
Part(pno,pname,psize,pcolor)
```

Subquery in WHERE

Find suppliers that supply *only* ‘red’ parts

Find the *other* suppliers

```
SELECT x.sno
FROM Supplier x
WHERE exists (SELECT * FROM Supply y, Part z
              WHERE x.sno=y.sno
                and y.pno=z.pno
                and z.pcolor != ‘red’);
```

```
Supplier(sno,sname,scity,sstate)
Supply(sno,pno,qty,price)
Part(pno,pname,psize,pcolor)
```

Subquery in WHERE

Find suppliers that supply *only* ‘red’ parts

Find the *other* suppliers

```
SELECT x.sno
FROM Supplier x
WHERE exists (SELECT * FROM Supply y, Part z
              WHERE x.sno=y.sno
                and y.pno=z.pno
                and z.pcolor != ‘red’);
```

```
SELECT x.sno
FROM Supplier x
WHERE not exists (SELECT * FROM Supply y, Part z
                  WHERE x.sno=y.sno
                    and y.pno=z.pno
                    and z.pcolor != ‘red’);
```

Negate to get the right ones

Relational Algebra

- Semijoin: R ⋉ S
  - Subset of R that joins with S
  - $R ⋉ S = \Pi_{Attrs(R)}(R \bowtie S)$

- Anti-semijoin: R⊳S
  - Subset of R that does not join with S
  - $R ⊳ S = R - (R ⋉ S)$

```
Supplier(sno,sname,scity,sstate)
Supply(sno,pno,qty,price)
Part(pno,pname,psize,pcolor)
```

Semi-Join

Find suppliers that supply some ‘blue’ parts

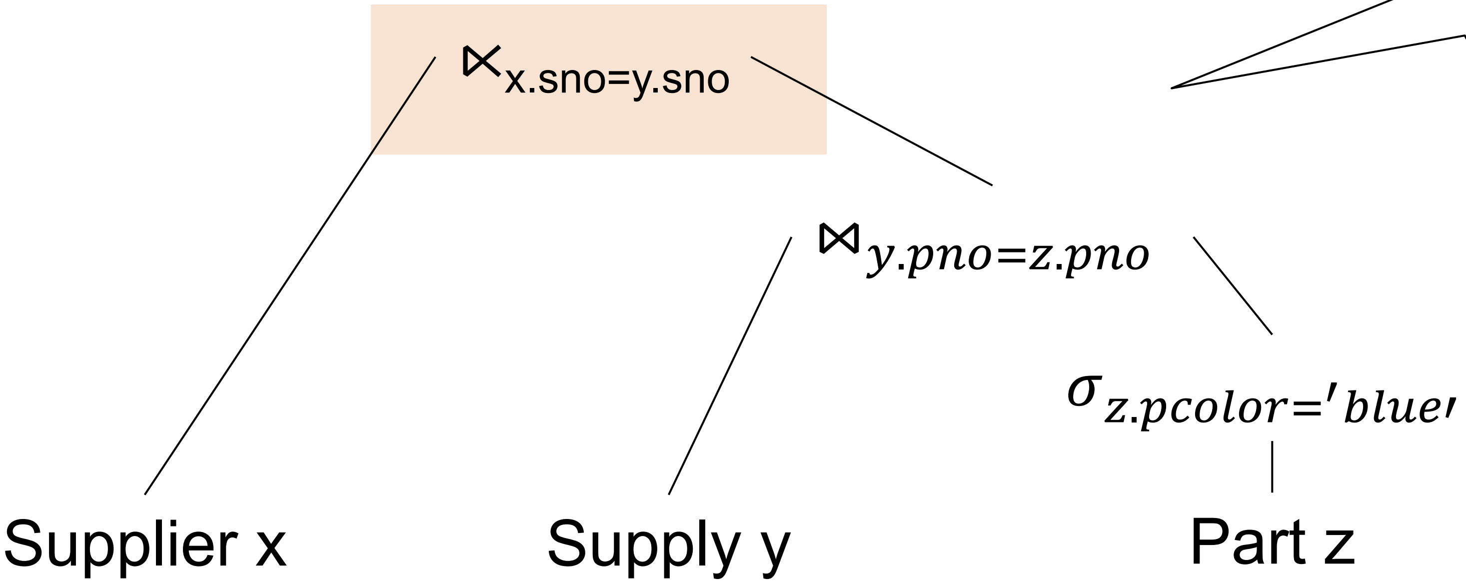
```
SELECT x.sno
FROM Supplier x
WHERE exists (SELECT * FROM Supply y, Part z
              WHERE x.sno=y.sno
                and y.pno=z.pno
                and z.pcolor = ‘blue’);
```

```
Supplier(sno,sname,scity,sstate)
Supply(sno,pno,qty,price)
Part(pno,pname,psize,pcolor)
```

Semi-Join

Find suppliers that supply some ‘blue’ parts

```sql
SELECT x.sno
FROM Supplier x
WHERE exists (SELECT * FROM Supply y, Part z
              WHERE x.sno=y.sno
                and y.pno=z.pno
                and z.pcolor = ‘blue’);
```



$\ltimes_{\text{x.sno=y.sno}}$

$\bowtie_{y.pno=z.pno}$

$\sigma_{z.pcolor='blue'}$

Supplier x

Supply y

Part z

Semi-join does not introduce duplicates

Supplier(sno,sname,scity,sstate)
Supply(sno,pno,qty,price)
Part(pno,pname,psize,pcolor)

Anti-semi-Join

Find suppliers that supply *only* ‘red’ parts

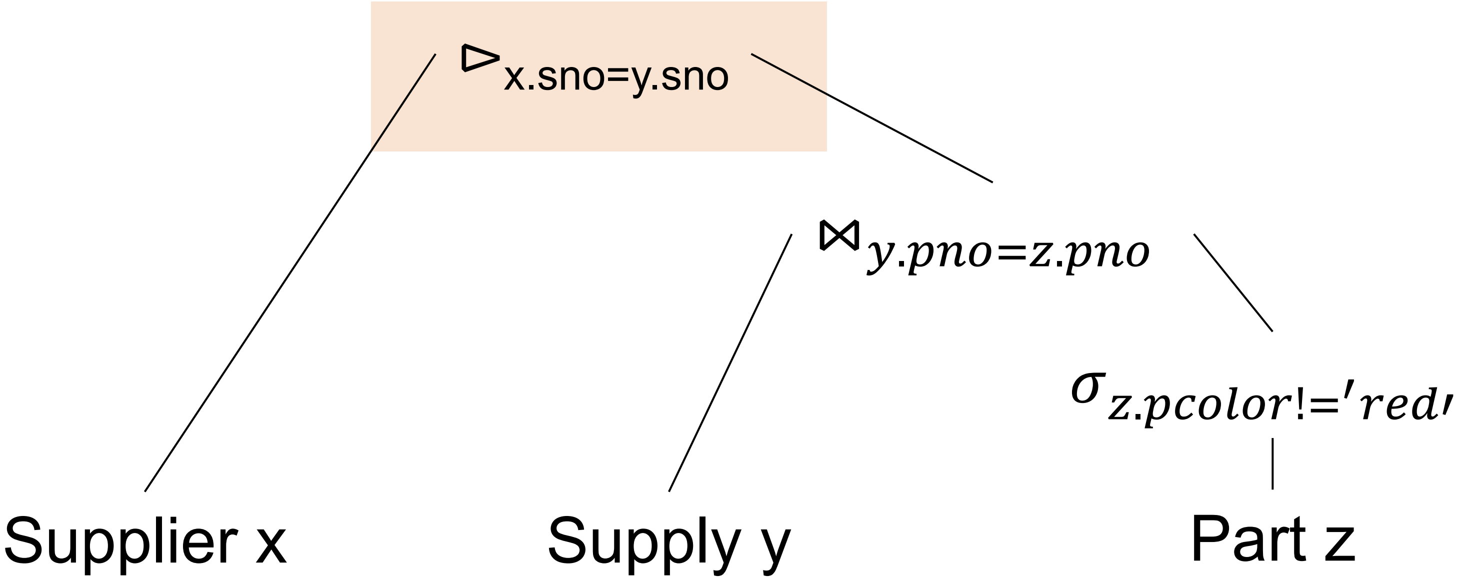
```
SELECT x.sno
FROM Supplier x
WHERE not exists (SELECT * FROM Supply y, Part z
                  WHERE x.sno=y.sno
                    and y.pno=z.pno
                    and z.pcolor != ‘red’);
```

`Supplier(sno,sname,scity,sstate)`
`Supply(sno,pno,qty,price)`
`Part(pno,pname,psize,pcolor)`

Anti-semi-Join

Find suppliers that supply *only* ‘red’ parts

```
SELECT x.sno
FROM Supplier x
WHERE not exists (SELECT * FROM Supply y, Part z
                  WHERE x.sno=y.sno
                    and y.pno=z.pno
                    and z.pcolor != ‘red’);
```



$\triangleright_{x.sno=y.sno}$

$\bowtie_{y.pno=z.pno}$

$\sigma_{z.pcolor!='red'}$

Supplier x

Supply y

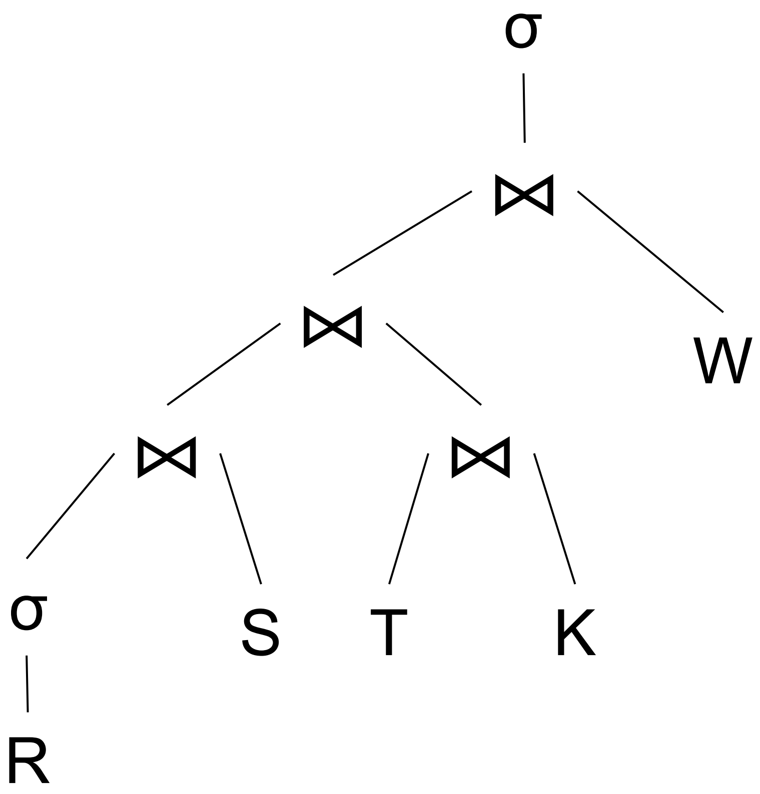
Part z

Discussion

- RA does not have variables
  - Exception: “dependent” join allows variables, but needs to be removed
- Query unnesting: rewriting a query with subqueries into a query without subqueries
- Some systems fail to unnest complicated queries: nested loop join

Operator Interface

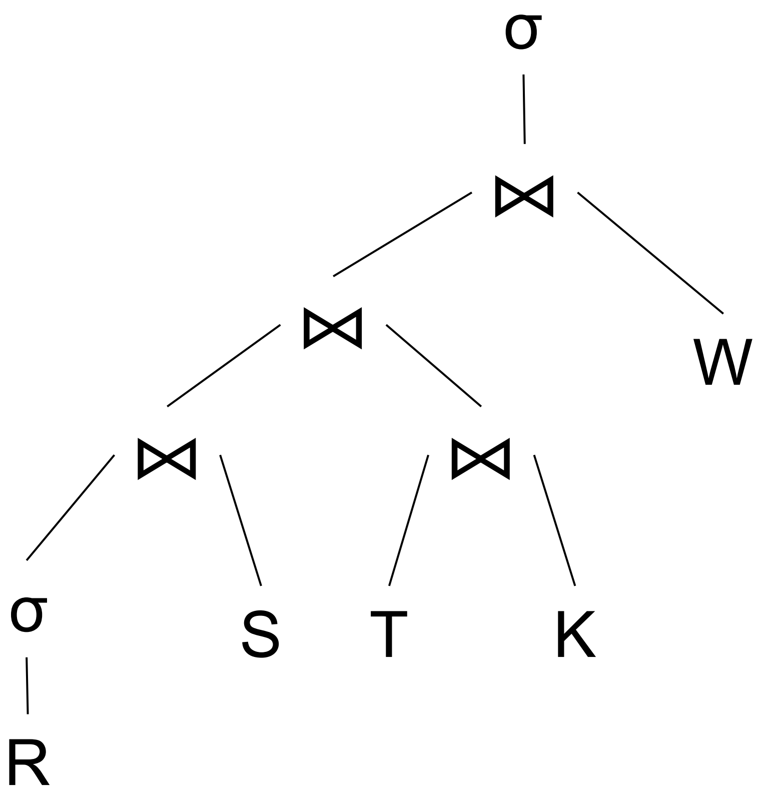
How Do We Combine Them?



How Do We Combine Them?

Option 1:
materialize intermediate results

Option 2:
Pipeline tuples btw. ops

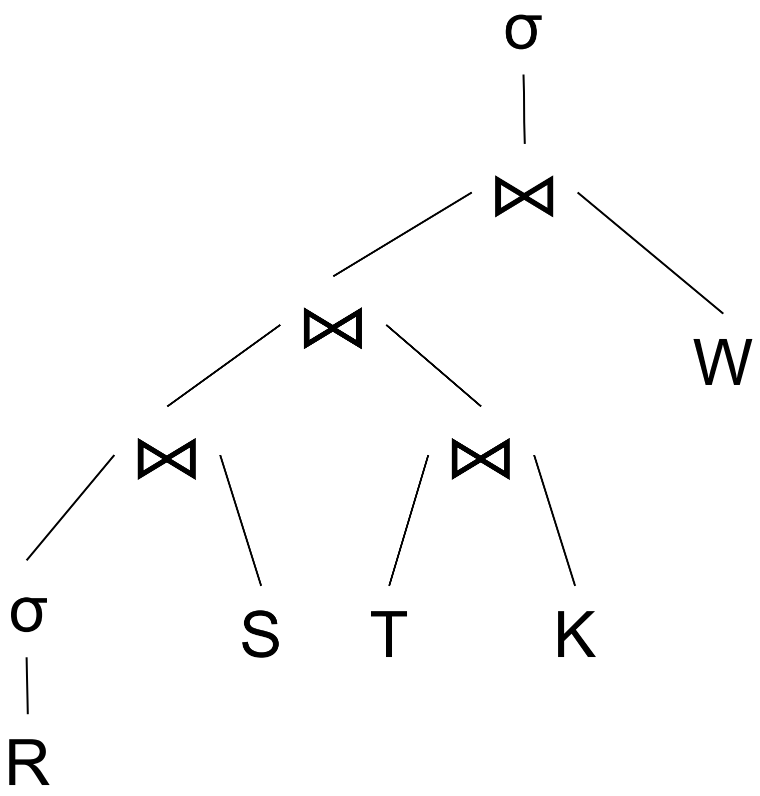


How Do We Combine Them?

Option 1:
materialize intermediate results

Option 2:
Pipeline tuples btw. ops

Implementation:
Iterator Interface



Operator Interface

Volcano model:

- open(), next(), close()
- Pull model
- Volcano optimizer: G. Graefe's (Wisconsin) → SQL Server
- Supported by most DBMS today
- Will discuss next

Operator Interface

Volcano model:

- open(), next(), close()
- Pull model
- Volcano optimizer: G. Graefe’s (Wisconsin) → SQL Server
- Supported by most DBMS today

Data-driven model:

- open(),produce(), consume(),close()
- Push model
- Introduced by Thomas Neumann in Hyper (at TU Munich), later acquired by Tableau

Discussion

- Most systems adopt the Volcano-model, a.k.a. the iterator interface
- Vectorized processing = iterator interface that processes a *block* of tuples (vector?) instead of one tuple
- Compiled model = compile to machine code *and* use the push model