CSE544
Data Management

Lecture 5

Query Execution

Announcements

- HW3 is posted, due on Sunday, 2/23
  - Some Java programming involved

- Project proposal due on Sunday:
  - Submit on gitlab, under /project
  - Only one team member needs to submit
  - See instructions on the Website

Where We Are

- SQL, Relational Model
- Storage manager, buffer pool, indexes
- Today and next week: the query engine

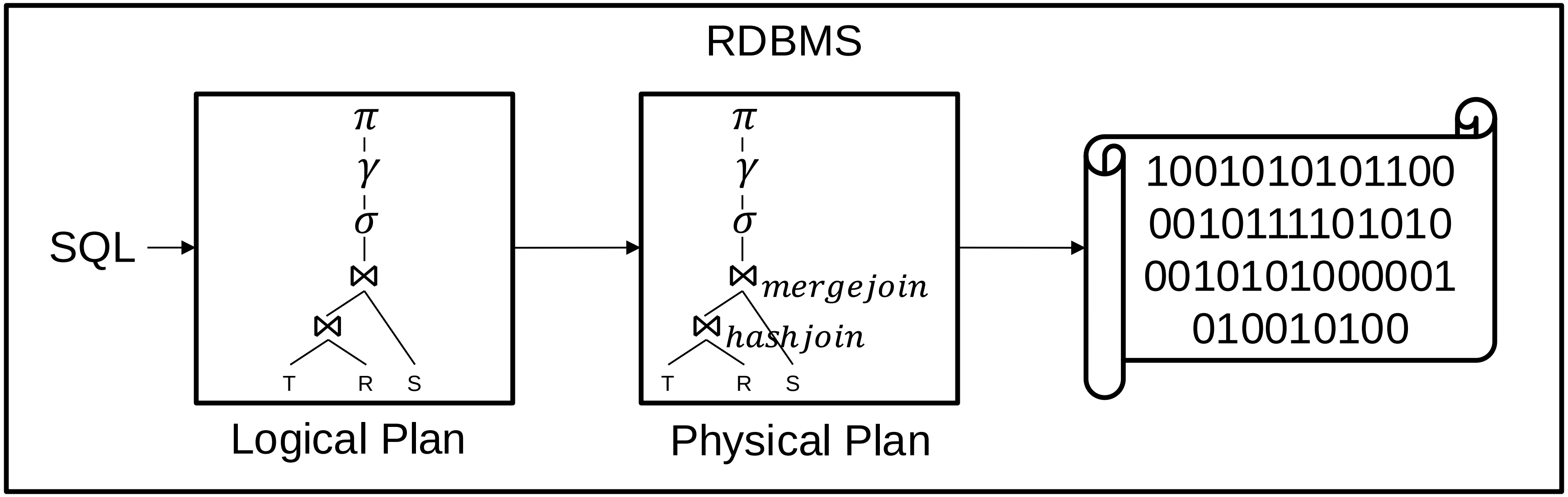
Outline

- Query engine overview
- Relational Algebra
- Physical Operators
- Iterator Model

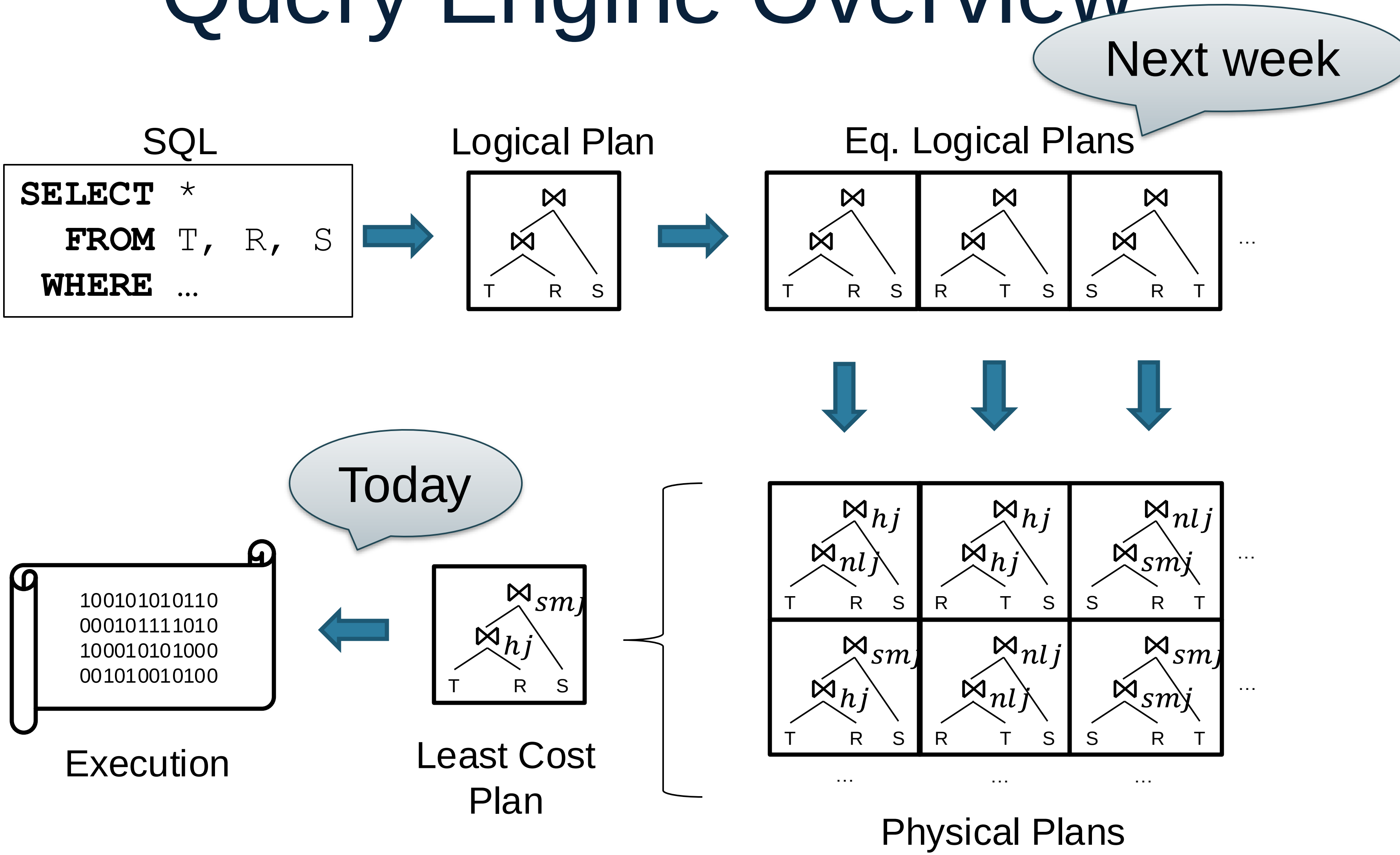
Query Engine Overview

- SQL query is parsed, analyzed
- Converted to Relational Algebra: "Logical Query Plan"
- Query plan is optimized, converted to "Physical Query Plan"
- Physical query plan is executed

Query Engine Overview



Query Engine Overview



Relational Algebra

Relational Algebra

- SQL declarative: we say what we want
- RA: says how to get it

RA: Five Basic Operators

1. Selection $\sigma_{\text{condition}}(S)$
2. Projection $\Pi_{\text{attrs}}(S)$
3. Join $\mathrm{R} \bowtie_{\theta} S = \sigma_{\theta}(\mathrm{R} \times S)$
4. Union $\cup$
5. Set difference $-$

1. Selection

$$\sigma_{\mathrm{condition}}(\mathrm{T})$$

Returns those tuples in T that satisfy the condition:

```
SELECT *
FROM T
WHERE condition;
```

1. Selection

$$\sigma_{\text{condition}}(T)$$

Returns those tuples in T that satisfy the condition:

```
SELECT *
FROM T
WHERE condition;
```

Part

pno	pname	psize
0005	Sneaker	8
0032	Sandal	7
0555	Boot	12
0621	Sandal	5

1. Selection

$$\sigma_{\text{condition}}(\text{T})$$

Returns those tuples in T that satisfy the condition:

```
SELECT *
FROM T
WHERE condition;
```

$$\sigma_{\text{psize} \geq 8}(\text{Part})$$

Part

pno	pname	psize
0005	Sneaker	8
0032	Sandal	7
0555	Boot	12
0621	Sandal	5

1. Selection

$$\sigma_{condition}(T)$$

Returns those tuples in T that satisfy the condition:

```
SELECT *
FROM T
WHERE condition;
```

pno	pname	psize
0005	Sneaker	8
0555	Boot	12

$$\sigma_{psize \geq 8}(Part)$$

Part

pno	pname	psize
0005	Sneaker	8
0032	Sandal	7
0555	Boot	12
0621	Sandal	5

1. Selection

$$\sigma_{condition}(T)$$

Returns those tuples in T that satisfy the condition:

```
SELECT *
FROM T
WHERE condition;
```

$$\sigma_{psize \geq 8 \wedge pname='Sneaker'}(Part)$$

Part

pno	pname	psize
0005	Sneaker	8
0032	Sandal	7
0555	Boot	12
0621	Sandal	5

1. Selection

$\sigma_{\text{condition}}(\text{T})$

Returns those tuples in T that satisfy the condition:

```
SELECT *
FROM T
WHERE condition;
```

pno	pname	psize
0005	Sneaker	8

$\sigma_{\text{psize} \geq 8 \wedge \text{pname} = '\text{Sneaker}'}(\text{Part})$

Part

pno	pname	psize
0005	Sneaker	8
0032	Sandal	7
0555	Boot	12
0621	Sandal	5

$$\Pi_{attrs}(T)$$

2. Projection

Returns all tuples in T keeping only the attributes in the subscript:

```
SELECT attrs
FROM T;
```

2. Projection

$$\Pi_{\text{attrs}}(\text{T})$$

Returns all tuples in T keeping only the attributes in the subscript:

```
SELECT attrs
FROM T;
```

$$\Pi_{\text{pname}}(\text{Part})$$

Part

pno	pname	psize
0005	Sneaker	8
0032	Sandal	7
0555	Boot	12
0621	Sandal	5

2. Projection

$\Pi_{\text{attrs}}(T)$

Set semantics

pname
Sneaker
Sandal
Boot

Returns all tuples in T keeping only the attributes in the subscript:

$\Pi_{\text{pname}}(\text{Part})$

```
SELECT attrs
FROM T;
```

Part

pno	pname	psize
0005	Sneaker	8
0032	Sandal	7
0555	Boot	12
0621	Sandal	5

2. Projection

$\Pi_{attrs}(T)$

Returns all tuples in T keeping only the attributes in the subscript:

```
SELECT attrs
FROM T;
```

Set semantics

pname
Sneaker
Sandal
Boot

Bag semantics

pname
Sneaker
Sandal
Boot
Sandal

$\Pi_{pname}(Part)$

Part

pno	pname	psize
0005	Sneaker	8
0032	Sandal	7
0555	Boot	12
0621	Sandal	5

Discussion

Two semantics for Relational Algebra:

- Bag semantics: what engines implement

- Set semantics: what Codd initially proposed, and what allows us to study equivalence with First Order Logic

3. Join

$S \bowtie_\theta T$

Join S and T using condition θ

```
SELECT *
FROM S,T
WHERE θ;
```

3. Join

$S \bowtie_{\theta} T$

Join S and T using condition θ

```
SELECT *
FROM S,T
WHERE θ;
```

$R \bowtie_{B=C} S$

R

A	B
2	10
5	10
5	20

S

C	D
10	a
10	b
20	b
30	a

3. Join

$S \bowtie_{\theta} T$

Join S and T using condition θ

```
SELECT *
FROM S,T
WHERE θ;
```

$R \bowtie_{B=C} S$

R

A	B
2	10
5	10
5	20

S

C	D
10	a
10	b
20	b
30	a

A	B	C	D
2	10	10	a
2	10	10	b
5	10	10	a
5	10	10	b
5	20	20	b

Variants of Join

- Eq-join: $\mathrm{R} \bowtie_{\mathrm{A=B}} S$
- Theta-join: $\mathrm{R} \bowtie_{\mathrm{A \leq B}} S$
- Cartesian product: $\mathrm{R} \times S$
- Natural Join: $\mathrm{R} \bowtie S$

Natural Join

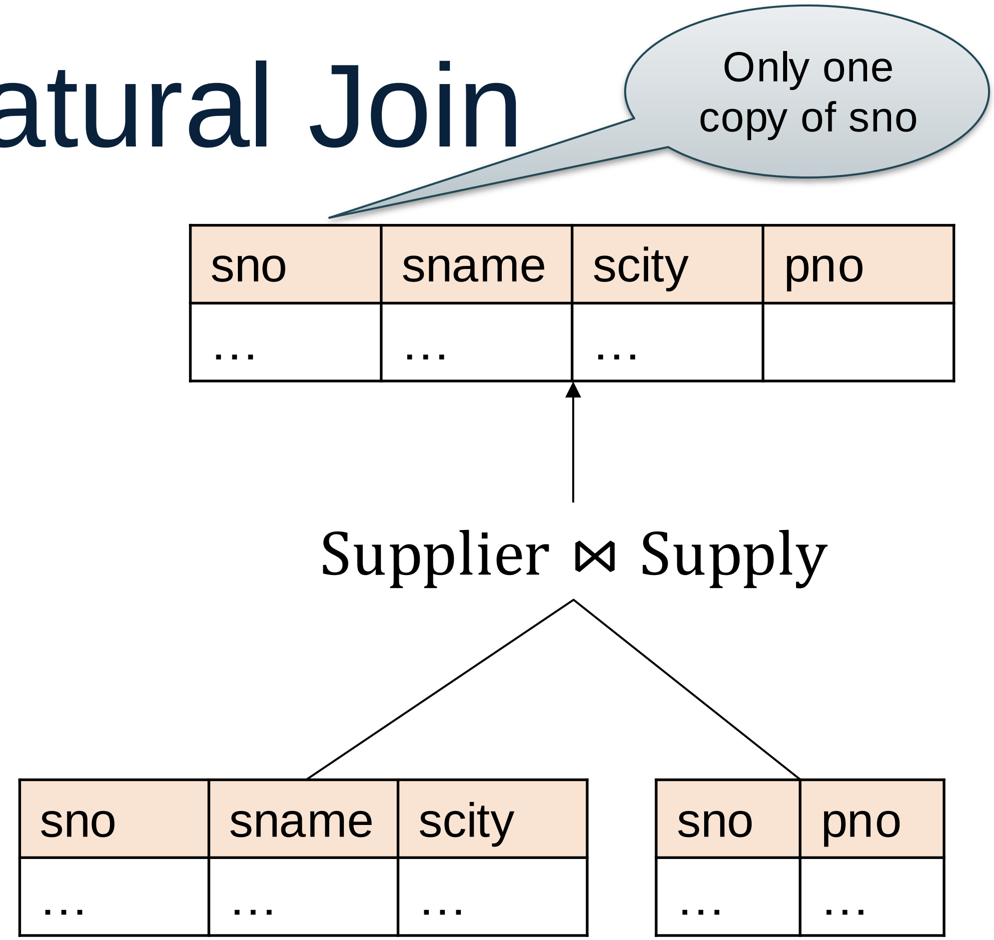
$S \bowtie T$

Join S, T on common attributes, retain only one copy of those attributes

Natural Join

$S \bowtie T$

Join S, T on common attributes, retain only one copy of those attributes



Natural Join

What do these natural joins output?

- $R(A, B) \bowtie S(B, C)$

- $R(A, B) \bowtie S(C, D)$

- $R(A, B) \bowtie S(A, B)$

Natural Join

R

A	B
1	10
2	10
2	20

S

B	C
10	8
10	9
20	8
50	7

What do these natural joins output?

- $R(A,B) \bowtie S(B,C)$
- $R(A,B) \bowtie S(C,D)$
- $R(A,B) \bowtie S(A,B)$

Natural Join

R

A	B
1	10
2	10
2	20

S

B	C
10	8
10	9
20	8
50	7

What do these natural joins output?

- $R(A,B) \bowtie S(B,C)$
  eqjoin on attribute B (5 tuples)
- $R(A,B) \bowtie S(C,D)$
- $R(A,B) \bowtie S(A,B)$

Natural Join

R

A	B
1	10
2	10
2	20

S

B	C
10	8
10	9
20	8
50	7

What do these natural joins output?

- $R(A, B) \bowtie S(B, C)$
  eqjoin on attribute B (5 tuples)

R

A	B
1	10
2	10
2	20

S

C	D
8	u
9	v
8	v
7	w

- $R(A, B) \bowtie S(C, D)$

- $R(A, B) \bowtie S(A, B)$

Natural Join

R

A	B
1	10
2	10
2	20

S

B	C
10	8
10	9
20	8
50	7

What do these natural joins output?

- $R(A, B) \bowtie S(B, C)$
  eqjoin on attribute B (5 tuples)

R

A	B
1	10
2	10
2	20

S

C	D
8	u
9	v
8	v
7	w

- $R(A, B) \bowtie S(C, D)$
  cross product (12 tuples)

- $R(A, B) \bowtie S(A, B)$

Natural Join

What do these natural joins output?

- $R(A,B) \bowtie S(B,C)$
  eqjoin on attribute B (5 tuples)

R

A	B
1	10
2	10
2	20

S

B	C
10	8
10	9
20	8
50	7

- $R(A,B) \bowtie S(C,D)$
  cross product (12 tuples)

R

A	B
1	10
2	10
2	20

S

C	D
8	u
9	v
8	v
7	w

- $R(A,B) \bowtie S(A,B)$

R

A	B
1	10
2	10
2	20

S

A	B
1	10
2	20

Natural Join

What do these natural joins output?

- $R(A, B) \bowtie S(B, C)$
  eqjoin on attribute B (5 tuples)

R

A	B
1	10
2	10
2	20

S

B	C
10	8
10	9
20	8
50	7

- $R(A, B) \bowtie S(C, D)$
  cross product (12 tuples)

R

A	B
1	10
2	10
2	20

S

C	D
8	u
9	v
8	v
7	w

- $R(A, B) \bowtie S(A, B)$
  intersection (2 tuples)

R

A	B
1	10
2	10
2	20

S

A	B
1	10
2	20

Natural Join

What do these natural joins output?

- $R(A, B) \bowtie S(B, C)$
  eqjoin on attribute B (5 tuples)

R

A	B
1	10
2	10
2	20

S

B	C
10	8
10	9
20	8
50	7

- $R(A, B) \bowtie S(C, D)$
  cross product (12 tuples)

R

A	B
1	10
2	10
2	20

S

C	D
8	u
9	v
8	v
7	w

- $R(A, B) \bowtie S(A, B)$
  intersection (2 tuples)

R

A	B
1	10
2	10
2	20

S

A	B
1	10
2	20

Intersection is a special case of join!

4-5. Union and Difference

$S \cup T$

Union of S and T

```
S UNION T;
```

SQL

4-5. Union and Difference

$S \cup T$

Union of S and T

```
S UNION T;
```

SQL

$S - T$

Set difference of S and T

```
S EXCEPT T;
```

The Five Basic Relational Operators

1. Selection $\sigma_{\mathrm{condition}}(S)$
2. Projection $\Pi_{\mathrm{attrs}}(S)$
3. Join $\mathrm{R} \bowtie_{\theta} \mathrm{S} = \sigma_{\theta}(\mathrm{R} \times \mathrm{S})$
4. Union $\cup$
5. Set difference $-$

Which are monotone?

The Five Basic Relational Operators

1. Selection $\sigma_{\text{condition}}(S)$
2. Projection $\Pi_{\text{attrs}}(S)$
3. Join $\mathrm{R} \bowtie_{\theta} \mathrm{S} = \sigma_{\theta}(\mathrm{R} \times S)$
4. Union $\cup$
5. Set difference $-$

Which are monotone?

Set difference is the only non-monotone

Extended Operators of Relational Algebra

- Duplicate elimination $\delta(R)$
- Group-by/aggregate $\gamma_{A,B,sum(C)}(R)$
- Sort operator $\tau(R)$

Query Plans

Query Plans

- SQL is translated into an RA expression
- The expression is usually shown as a tree, and called a query plan*

*aka Query Execution Plan

Query Plans

- SQL is translated into an RA expression

- The expression is usually shown as a tree, and called a query plan*

$$\gamma_{B,sum(E)}(\sigma_{A=5}(R) \bowtie_{C=D} S)$$

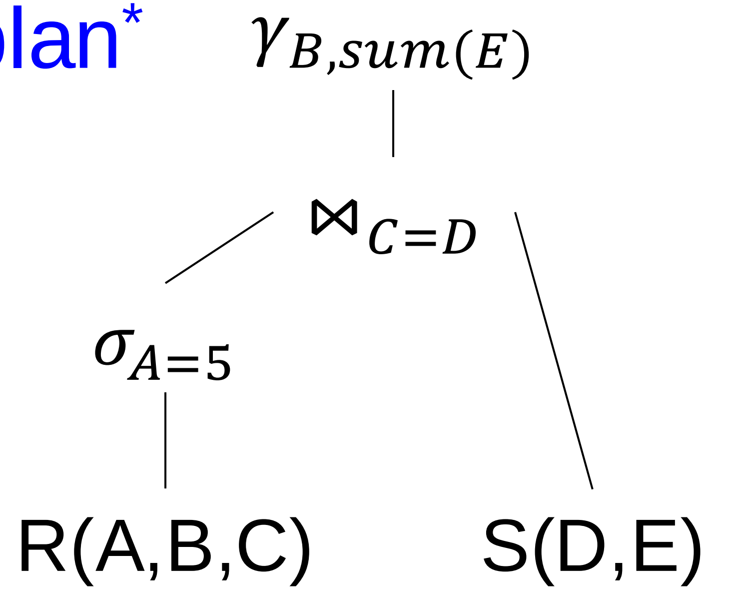
*aka Query Execution Plan

Query Plans

- SQL is translated into an RA expression

- The expression is usually shown as a tree, and called a query plan*

$$\gamma_{B,sum(E)}(\sigma_{A=5}(R) \bowtie_{C=D} S)$$



*aka Query Execution Plan

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

Query Plan Example

```
SELECT scity,avg(psize)as a
FROM Supplier x,
     Supply y,
     Part z
WHERE x.sno=y.sno
  and y.pno=z.pno
  and z.pcolor='red'
GROUP BY scity
HAVING count(*)>=100
```

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

Query Plan Example

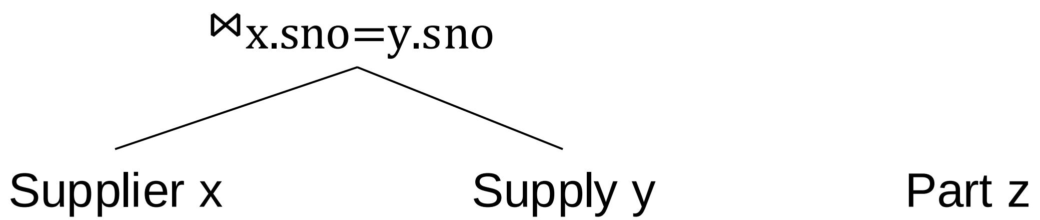
```
SELECT scity,avg(psize)as a
FROM Supplier x,
     Supply y,
     Part z
WHERE x.sno=y.sno
  and y.pno=z.pno
  and z.pcolor='red'
GROUP BY scity
HAVING count(*)>=100
```

Supplier x   Supply y   Part z

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

Query Plan Example

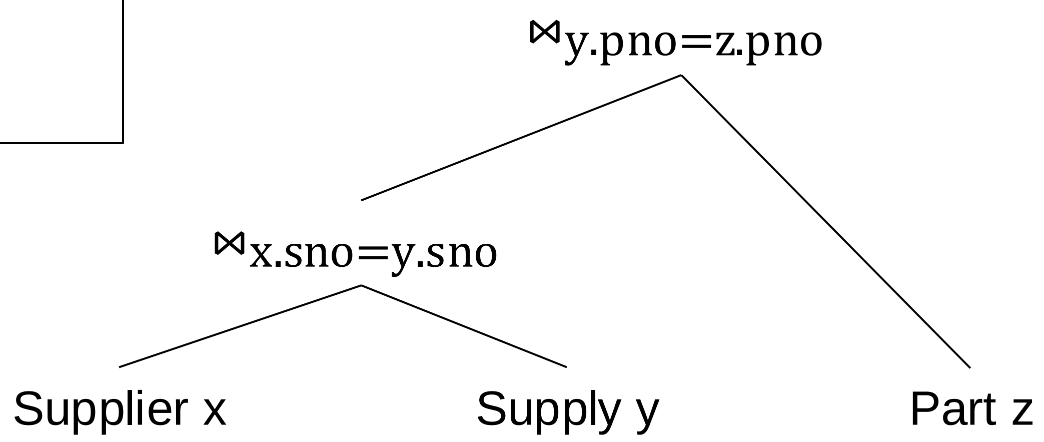
```
SELECT scity,avg(psize)as a
FROM Supplier x,
     Supply y,
     Part z
WHERE x.sno=y.sno
  and y.pno=z.pno
  and z.pcolor='red'
GROUP BY scity
HAVING count(*)>=100
```



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

Query Plan Example

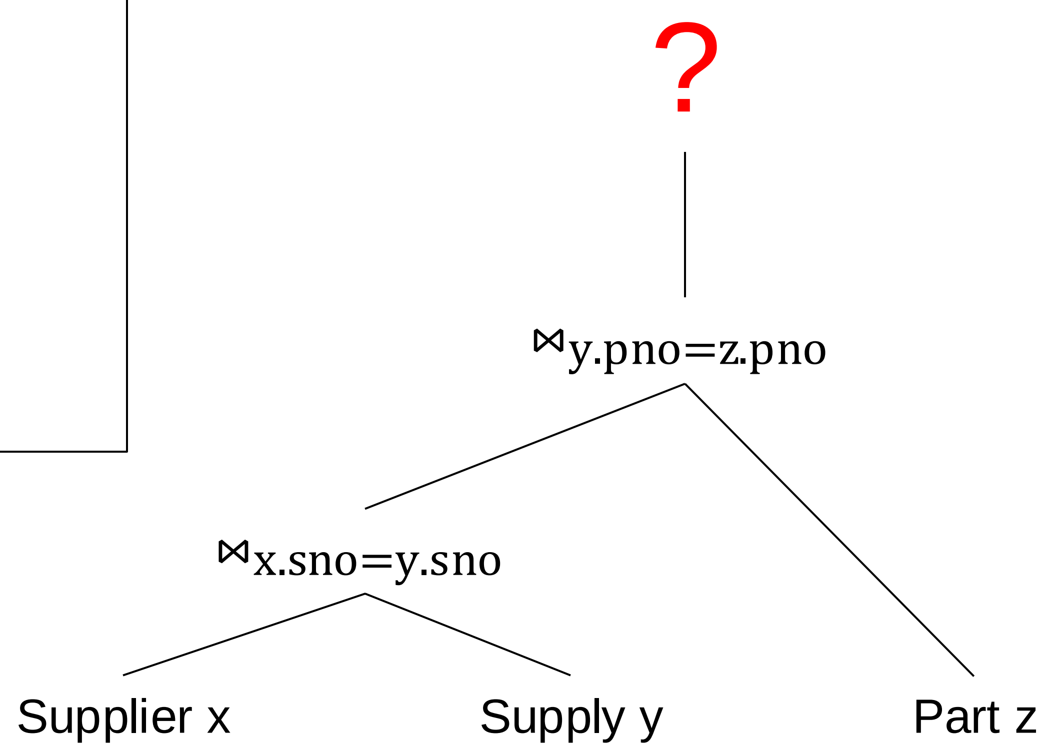
```
SELECT scity,avg(psize)as a
FROM Supplier x,
     Supply y,
     Part z
WHERE x.sno=y.sno
  and y.pno=z.pno
  and z.pcolor='red'
GROUP BY scity
HAVING count(*)>=100
```



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

Query Plan Example

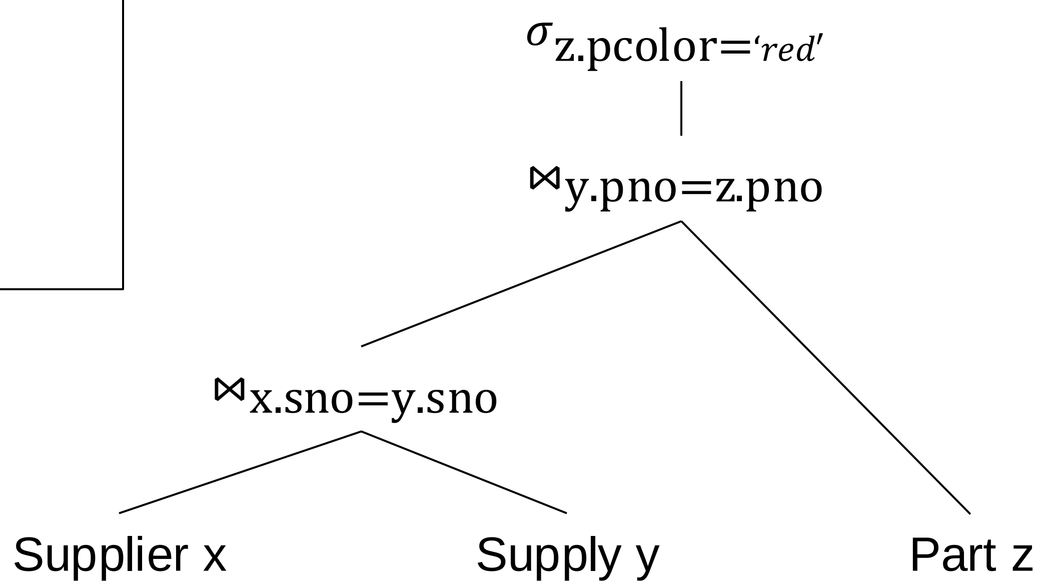
```
SELECT scity,avg(psize)as a
FROM Supplier x,
     Supply y,
     Part z
WHERE x.sno=y.sno
  and y.pno=z.pno
  and z.pcolor='red'
GROUP BY scity
HAVING count(*)>=100
```



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

Query Plan Example

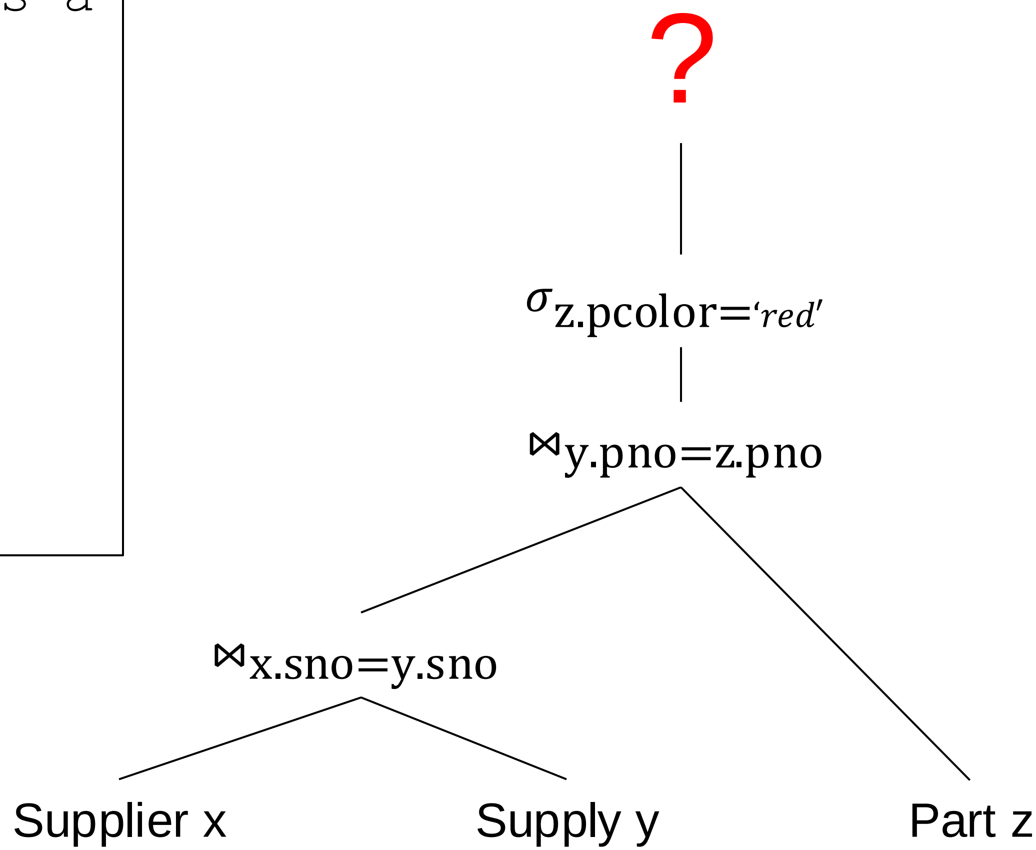
```
SELECT scity,avg(psize)as a
FROM Supplier x,
     Supply y,
     Part z
WHERE x.sno=y.sno
  and y.pno=z.pno
  and z.pcolor='red'
GROUP BY scity
HAVING count(*)>=100
```



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

Query Plan Example

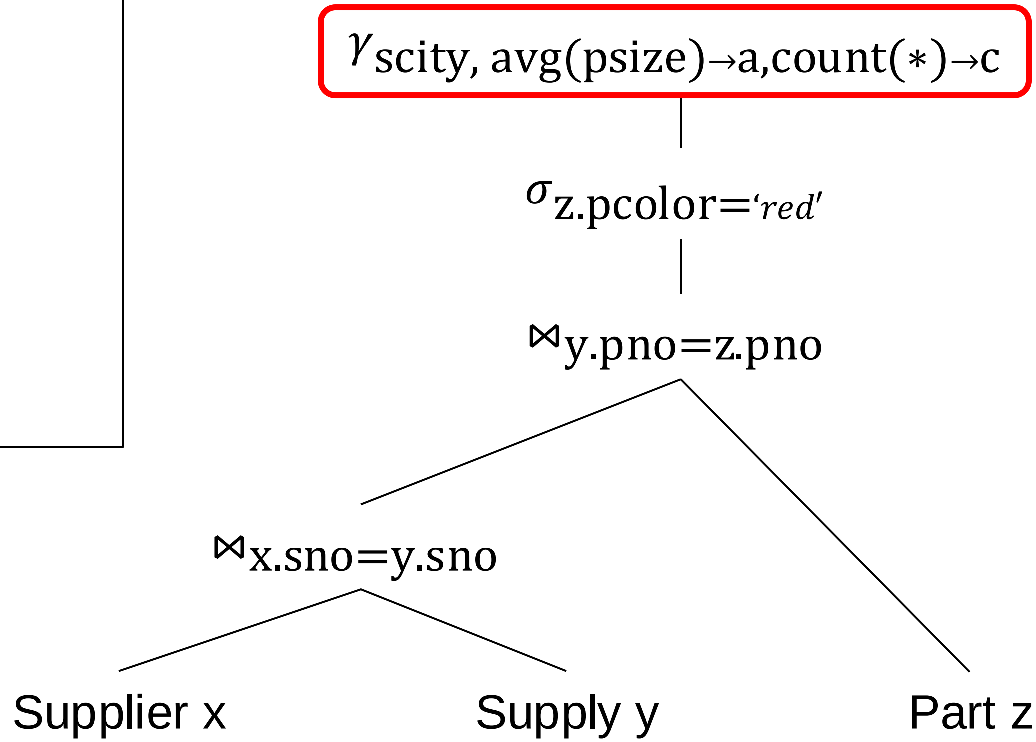
```
SELECT scity,avg(psize)as a
FROM Supplier x,
     Supply y,
     Part z
WHERE x.sno=y.sno
  and y.pno=z.pno
  and z.pcolor='red'
GROUP BY scity
HAVING count(*)>=100
```



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

Query Plan Example

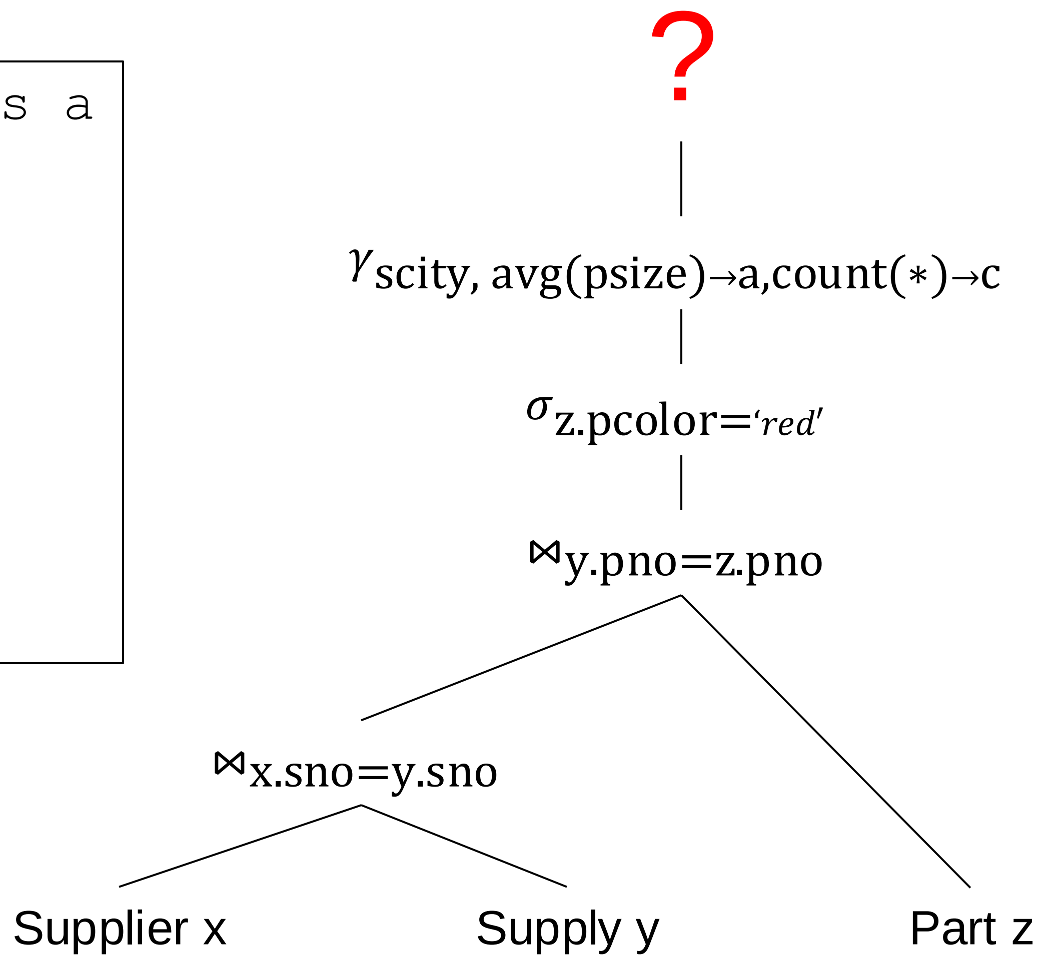
```
SELECT scity, avg(psize) as a
FROM Supplier x,
     Supply y,
     Part z
WHERE x.sno=y.sno
  and y.pno=z.pno
  and z.pcolor='red'
GROUP BY scity
HAVING count(*)>=100
```



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

Query Plan Example

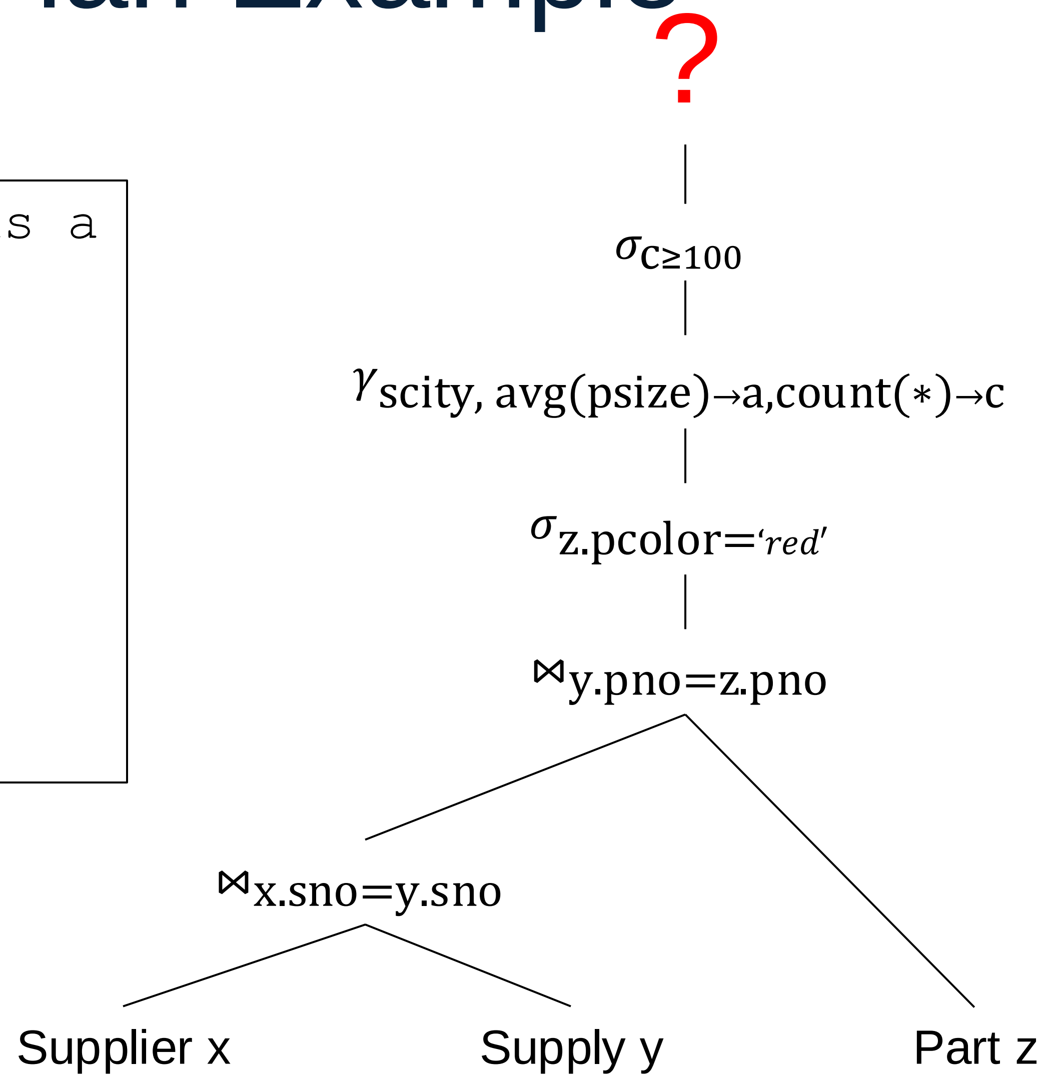
```
SELECT scity,avg(psize)as a
FROM Supplier x,
     Supply y,
     Part z
WHERE x.sno=y.sno
  and y.pno=z.pno
  and z.pcolor='red'
GROUP BY scity
HAVING count(*)>=100
```



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

Query Plan Example

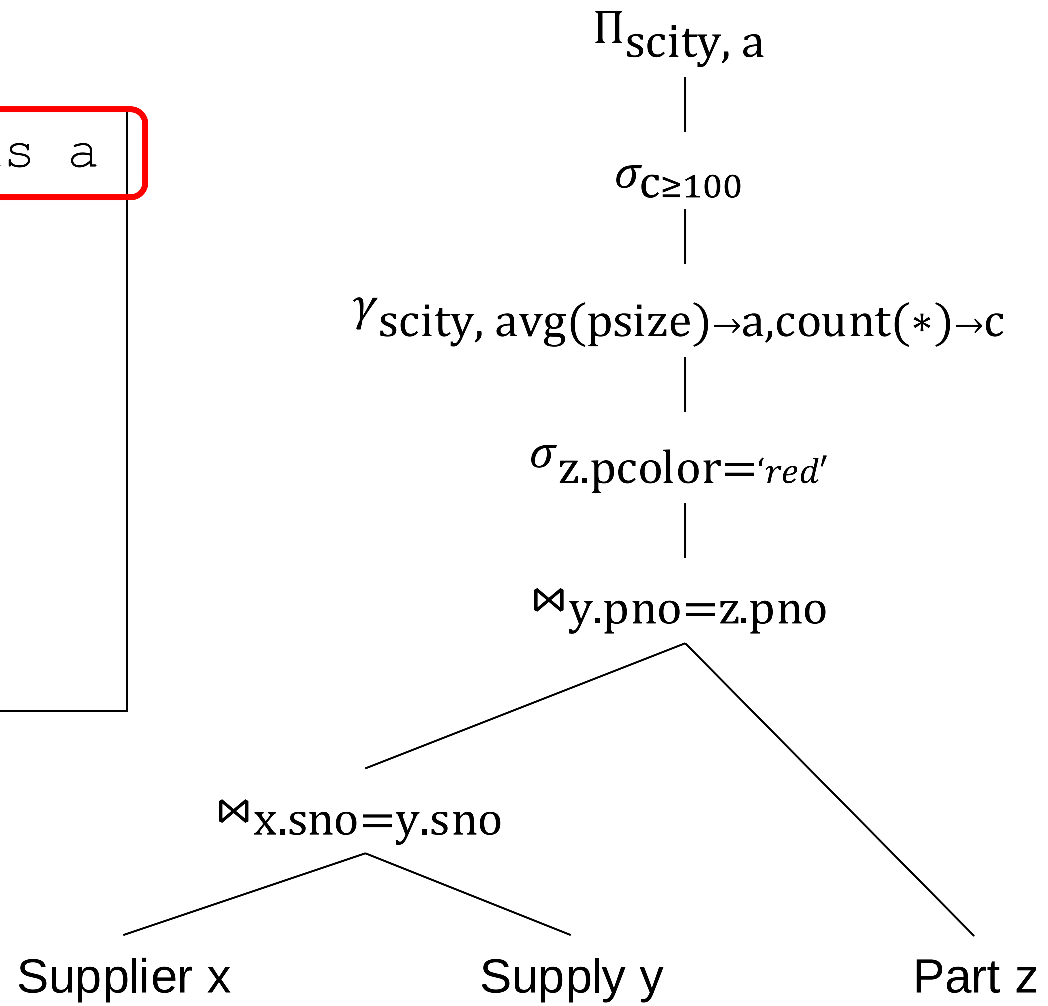
```
SELECT scity,avg(psize)as a
FROM Supplier x,
     Supply y,
     Part z
WHERE x.sno=y.sno
  and y.pno=z.pno
  and z.pcolor='red'
GROUP BY scity
HAVING count(*)>=100
```



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

Query Plan Example

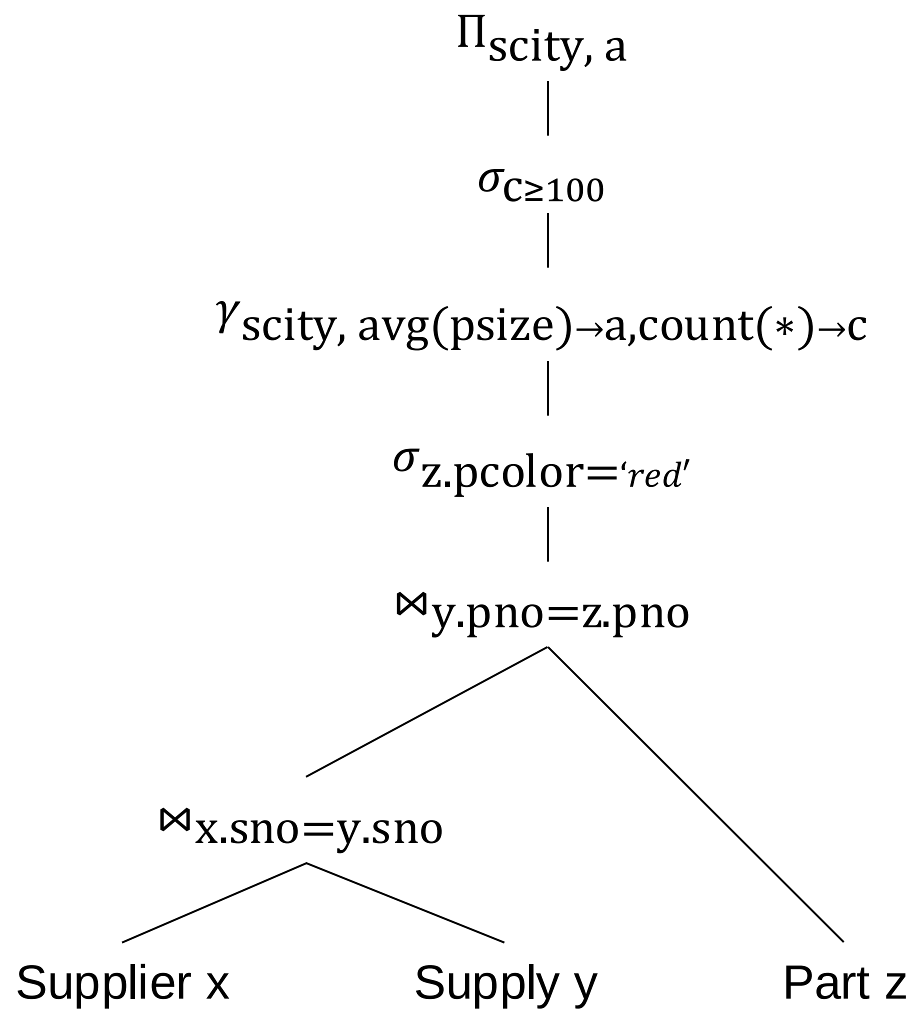
```
SELECT scity,avg(psize)as a
FROM Supplier x,
     Supply y,
     Part z
WHERE x.sno=y.sno
  and y.pno=z.pno
  and z.pcolor='red'
GROUP BY scity
HAVING count(*)>=100
```



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

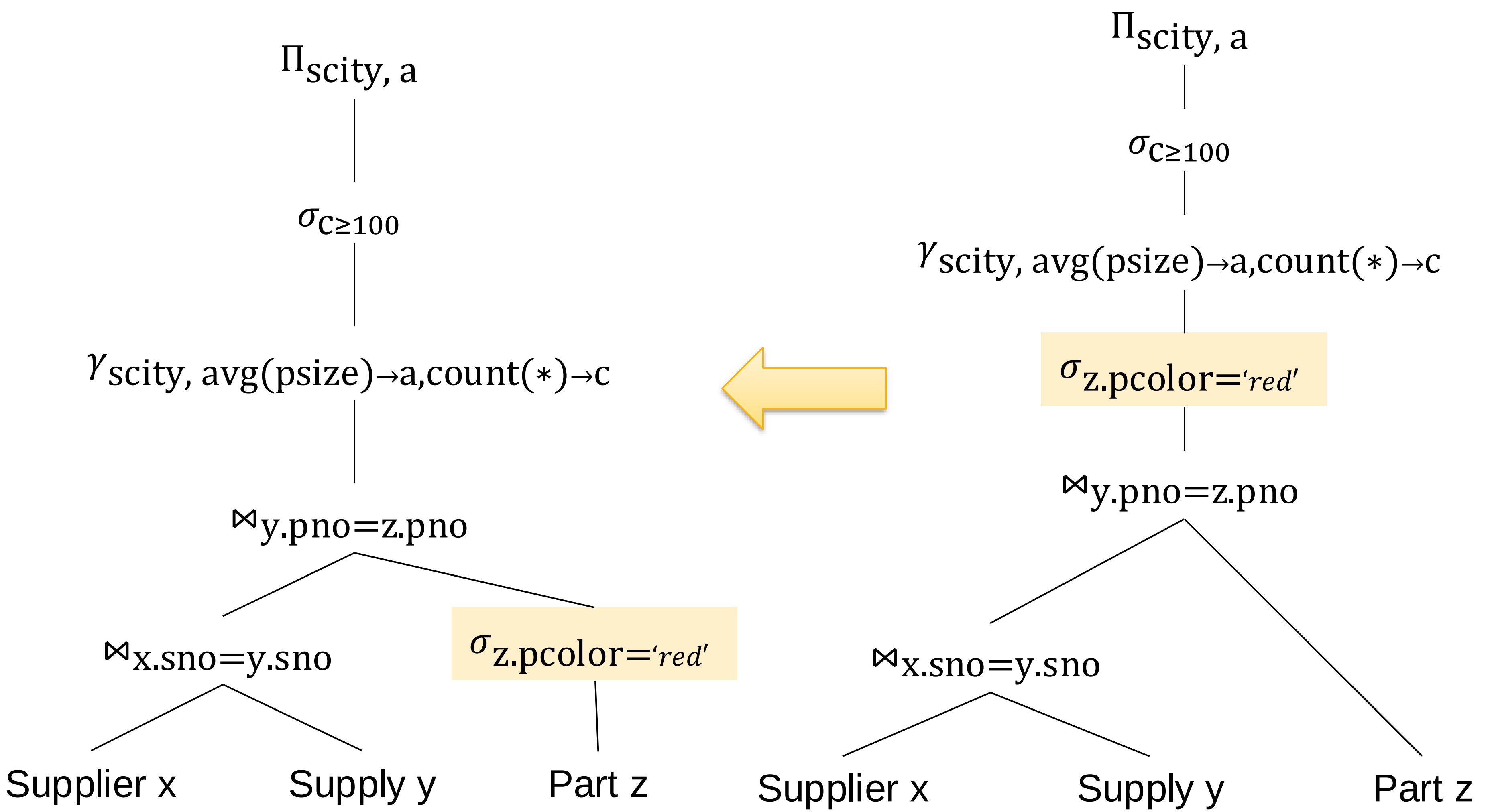
Logical Optimization

Find a better plan!



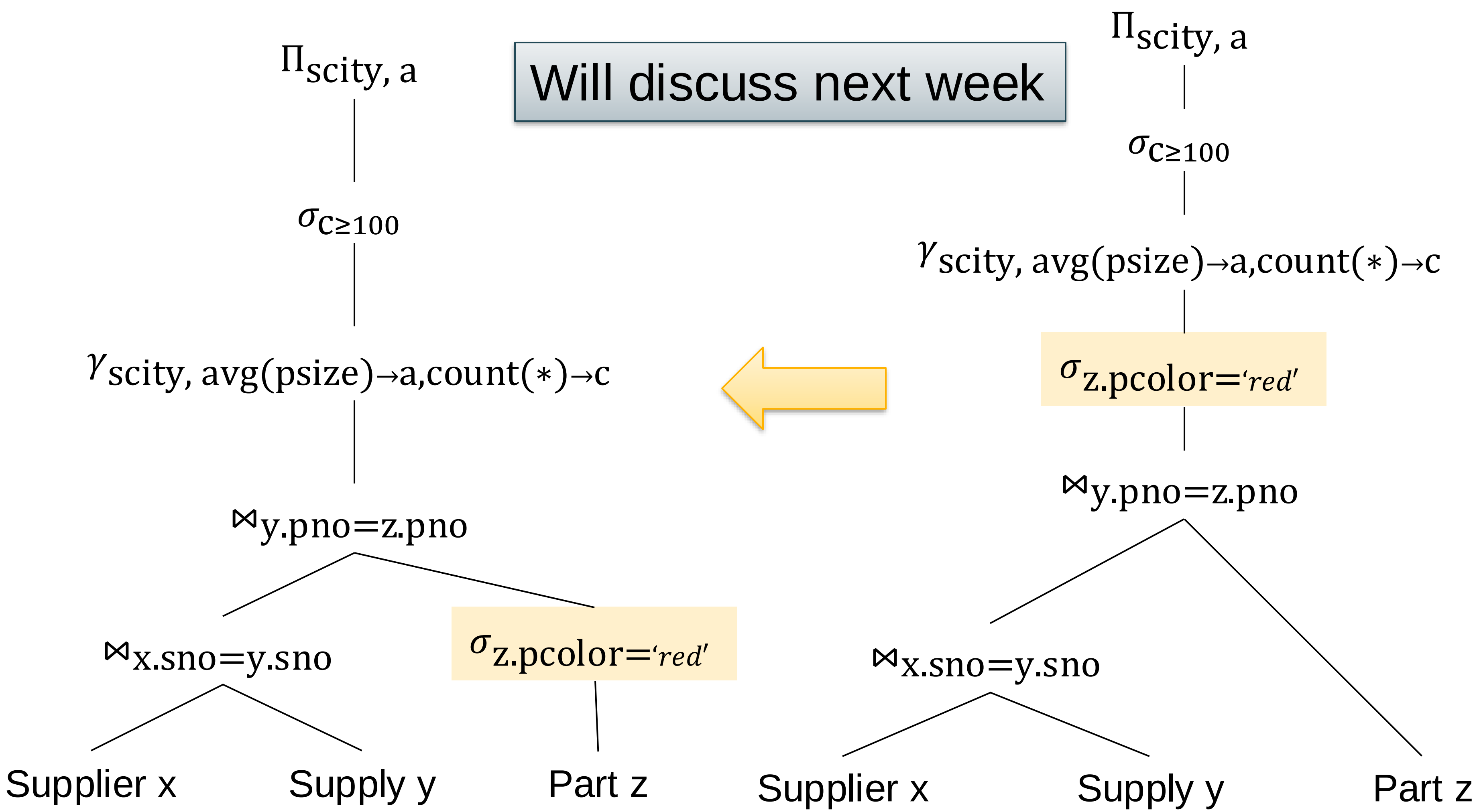
```
Supplier(sno,sname,scity)
Supply(sno,pno)
Part(pno,pname,psize,pcolor)
```

Logical Optimization



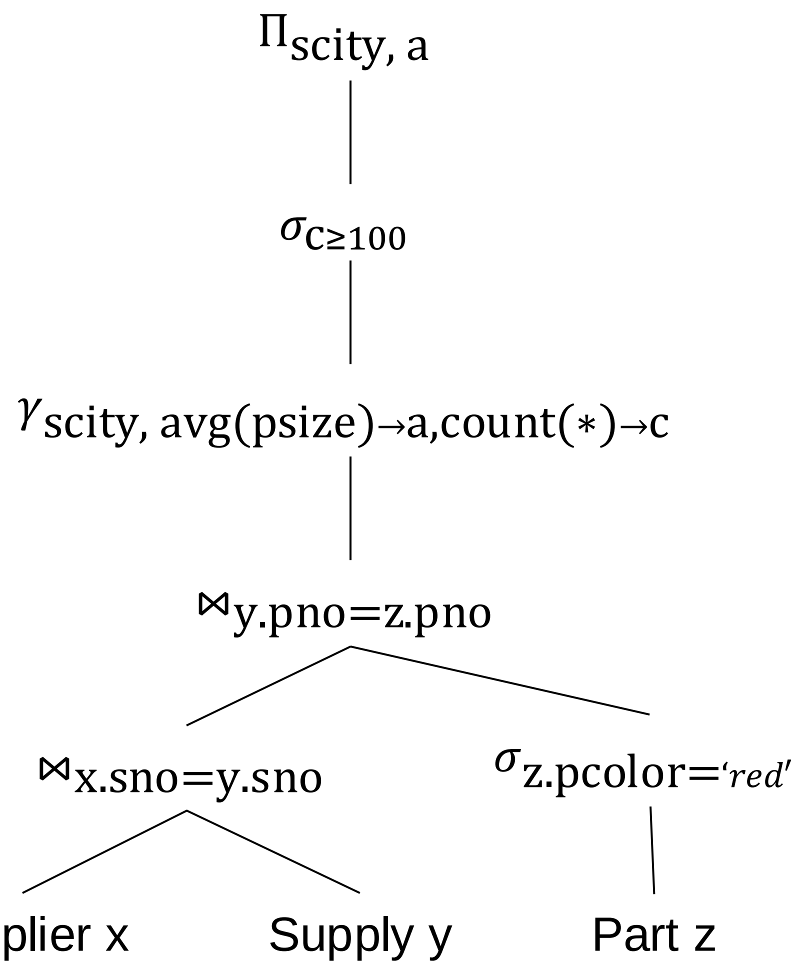
Supplier(<u>sno</u>,sname,scity)
Supply(sno,pno)
Part(<u>pno</u>,pname,psize,pcolor)

Logical Optimization



Supplier(<u>sno</u>,sname,scity)
Supply(sno,pno)
Part(<u>pno</u>,pname,psize,pcolor)

Physical Operators

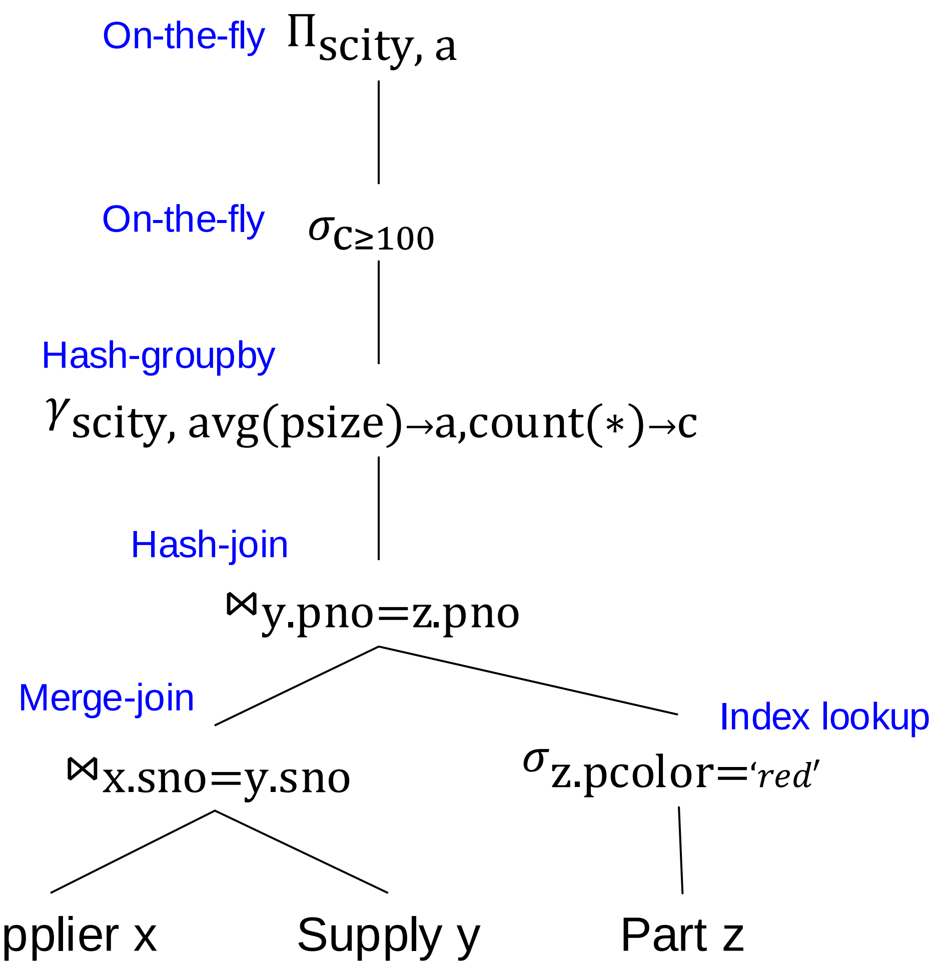


The plan tells
us the order of ops

Need to decide
on what algorithm
to use for each op

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

Physical Operators

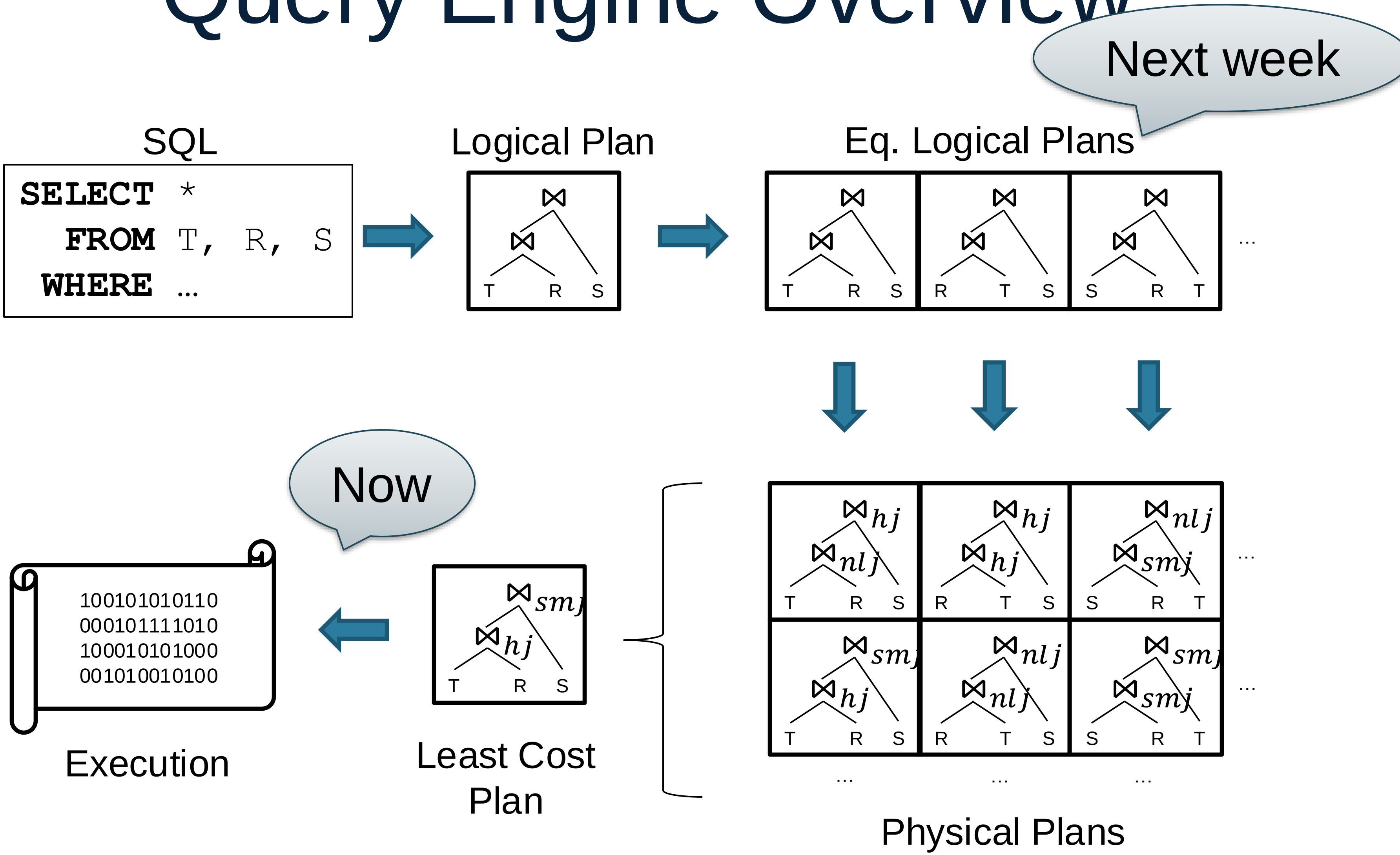


The plan tells
us the order of ops

Need to decide
on what algorithm
to use for each op

Physical Operators:
next

Query Engine Overview



Physical Operators

Physical Operators

- For each operator, several algorithms
- Main memory or external memory

```
Supplier(sno, sname, scity, sstate)
Supply(sno, pno, quantity)
```

Join Algorithms

Logical operator:

$$\text{Supplier} \bowtie_{\text{sno}=\text{sno}} \text{Supply}$$

Three algorithms:

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

```
Supplier(sno, sname, scity, sstate)
Supply(sno, pno, quantity)
```

1. Nested Loop Join

Logical operator:

$$\text{Supplier} \bowtie_{\text{sno}=\text{sno}} \text{Supply}$$

```
Supplier(sno, sname, scity, sstate)
Supply(sno, pno, quantity)
```

1. Nested Loop Join

Logical operator:

$$\text{Supplier} \bowtie_{\text{sno}=\text{sno}} \text{Supply}$$

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

```
Supplier(sno, sname, scity, sstate)
Supply(sno, pno, quantity)
```

1. Nested Loop Join

Logical operator:

$$\text{Supplier} \bowtie_{\text{sno}=\text{sno}} \text{Supply}$$

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

If |R|=|S|=n,
what is the runtime?

```
Supplier(sno, sname, scity, sstate)
Supply(sno, pno, quantity)
```

1. Nested Loop Join

Logical operator:

$$\text{Supplier} \bowtie_{\text{sno}=\text{sno}} \text{Supply}$$

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

If |R|=|S|=n,
what is the runtime?

$O(n^2)$

1. Nested Loop Join

When the data is on disk, the main cost is due to the number of I/Os (block read)

- Block nested loop join
- Index join

B(R) := number of disk blocks used by R

```
Supplier(sno, sname, scity, sstate)
Supply(sno, pno, quantity)
```

1a. Block Nested Loop Join

Nested Loop is very bad:

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

Number of I/Os:

B(Supplier) + |Supplier| * B(Supply)

```
Supplier(sno, sname, scity, sstate)
Supply(sno, pno, quantity)
```

1a. Block Nested Loop Join

M frames are available in the buffer pool

for each (M-2) blocks of Supplier **do**

Read M-2 blocks: $Supplier_{mem}$

```
Supplier(sno, sname, scity, sstate)
Supply(sno, pno, quantity)
```

1a. Block Nested Loop Join

M frames are available in the buffer pool

for each (M-2) blocks of Supplier **do**
  Read M-2 blocks: Supplier_{mem}
  for each block of Supply **do**
    Read 1 block: Supply_1

```
Supplier(sno, sname, scity, sstate)
Supply(sno, pno, quantity)
```

1a. Block Nested Loop Join

M frames are available in the buffer pool

for each (M-2) blocks of Supplier **do**
  Read M-2 blocks: Supplier_{mem}
  for each block of Supply **do**
    Read 1 block: Supply_1
    Output $\text{Supplier}_{mem} \bowtie \text{Supply}_1$

```
Supplier(sno, sname, scity, sstate)
Supply(sno, pno, quantity)
```

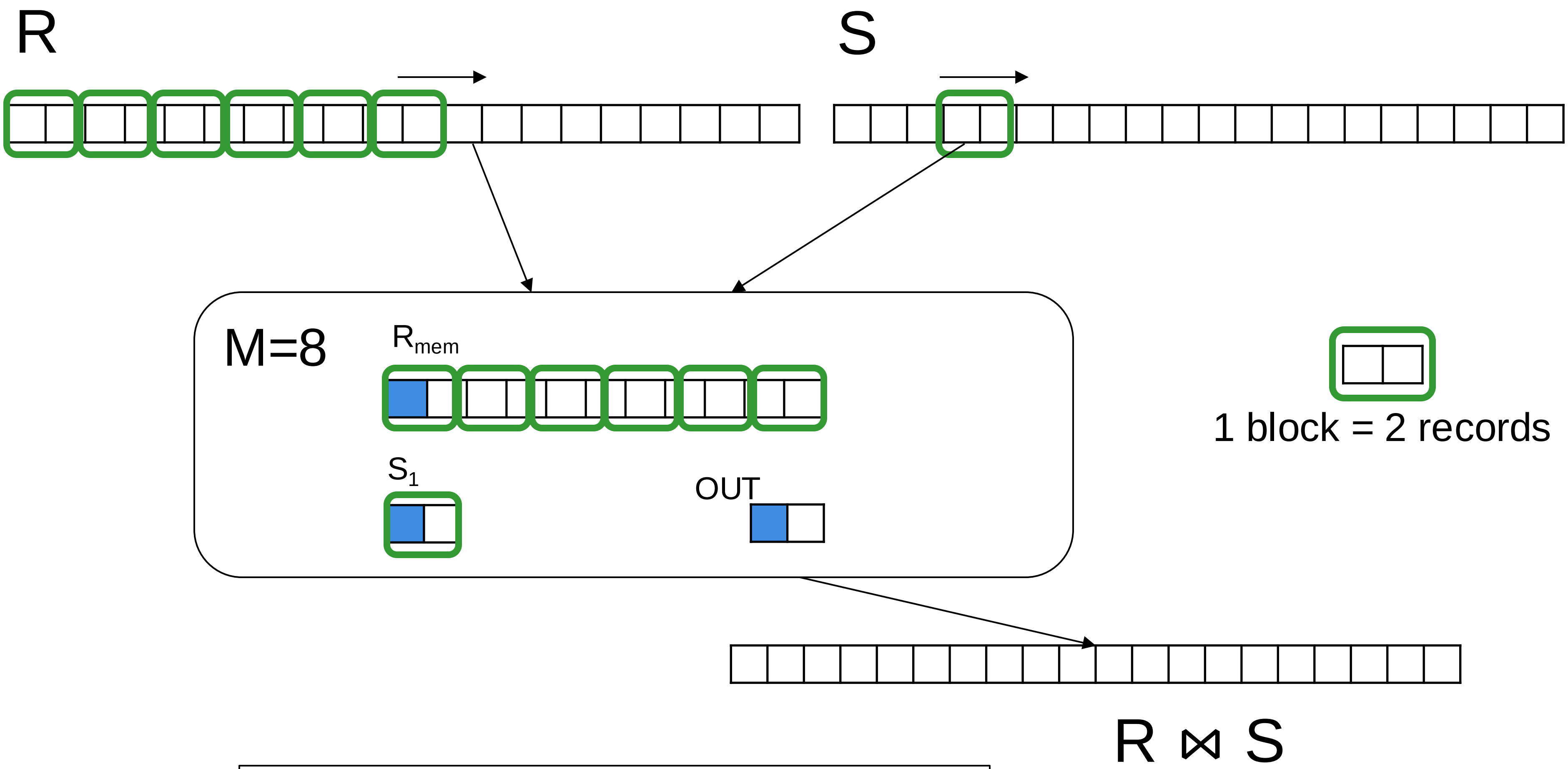
1a. Block Nested Loop Join

M frames are available in the buffer pool

for each (M-2) blocks of Supplier **do**
    Read M-2 blocks: Supplier_{mem}
    for each block of Supply **do**
        Read 1 block: Supply_1
        Output $\text{Supplier}_{mem} \bowtie \text{Supply}_1$

#I/O = B(Supplier)+B(Supplier)*B(Supply)/(M-2)

1a. Block Nested Loop Join



#I/O = B(R) + B(R)*B(S)/(M-2)

```
Supplier(sno, sname, scity, sstate)
Supply(sno, pno, quantity)
```

1b. Index Join

Logical operator:

$$\text{Supplier} \bowtie_{sno=sno} \text{Supply}$$

```
for x in Supplier do
  for y in SupplyIndex(x.sno) do
    output(x,y)
```

How many blocks do we read*?

*assume O(1) index lookup (why??)

`Supplier(sno, sname, scity, sstate)`
`Supply(sno, pno, quantity)`

1b. Index Join

Logical operator:

$$\text{Supplier} \bowtie_{\text{sno}=\text{sno}} \text{Supply}$$

```
for x in Supplier do
  for y in SupplyIndex(x.sno) do
    output(x,y)
```

How many blocks do we read*?

O(|Supplier|)

*assume O(1) index lookup (why??)

Discussion

- While $R \bowtie S = S \bowtie R$,
  an algorithm for $R \bowtie S$ may be quite different from one for $S \bowtie R$
- Terminology for $R \bowtie S$:
  - R = the outer table
  - S = the inner table
- We want the outer table to be small

```
Supplier(sno, sname, scity, sstate)
Supply(sno, pno, quantity)
```

2. Hash Join

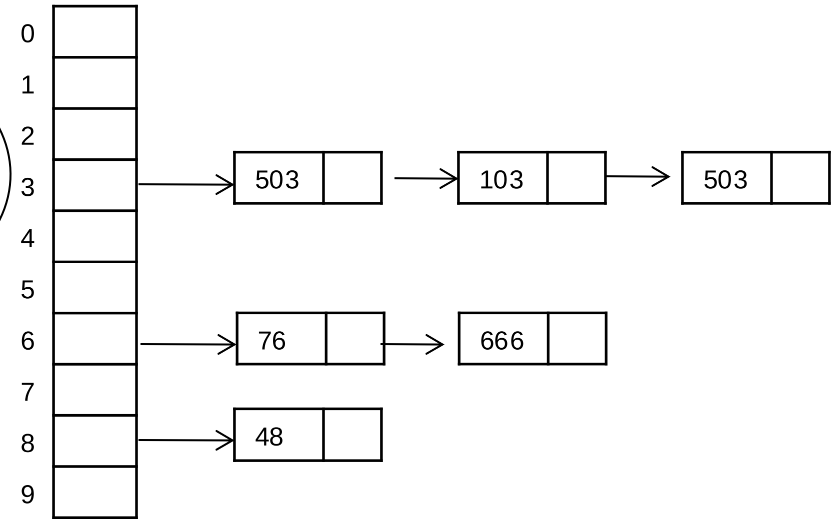
Logical operator:

$$\text{Supplier} \bowtie_{\text{sno}=\text{sno}} \text{Supply}$$

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

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

2. Hash Join



Logical operator:

$\text{Supplier} \bowtie_{\text{sno}=\text{sno}} \text{Supply}$

Build phase

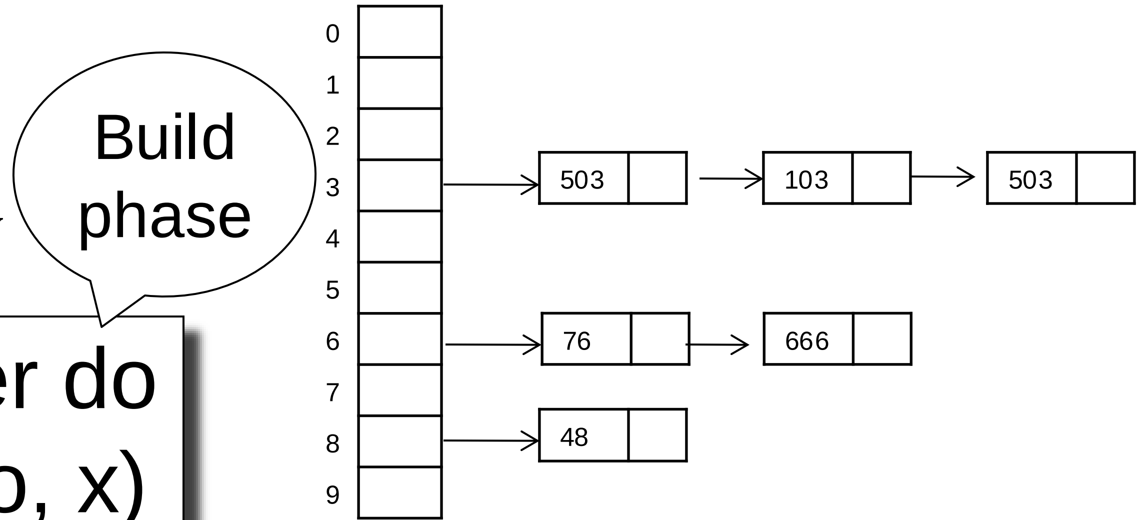
```
for x in Supplier do
      insert(x.sno, x)
```

```
Supplier(sno, sname, scity, sstate)
Supply(sno, pno, quantity)
```

2. Hash Join

Logical operator:

$$\text{Supplier} \bowtie_{sno=sno} \text{Supply}$$




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

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

```
Supplier(sno, sname, scity, sstate)
Supply(sno, pno, quantity)
```

2. Hash Join

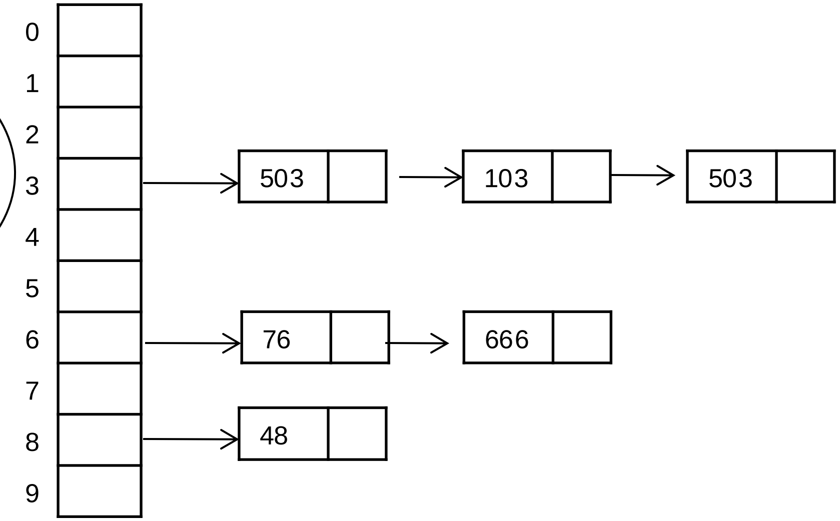
Logical operator:

$\text{Supplier} \bowtie_{sno=sno} \text{Supply}$

Build phase

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

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

Probe phase



```
Supplier(sno, sname, scity, sstate)
Supply(sno, pno, quantity)
```

2. Hash Join

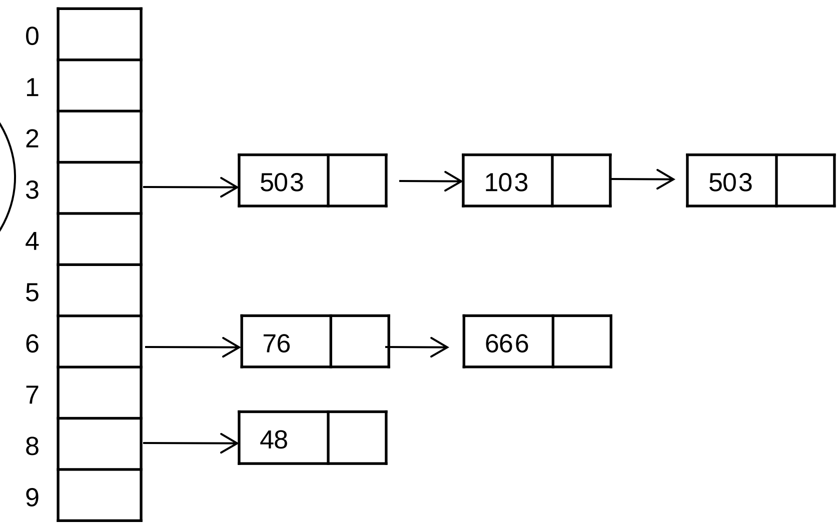
Logical operator:

$$\text{Supplier} \bowtie_{sno=sno} \text{Supply}$$

Build phase

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

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

Probe phase



If |R|=|S|=n,
what is the runtime?

```
Supplier(sno, sname, scity, sstate)
Supply(sno, pno, quantity)
```

2. Hash Join

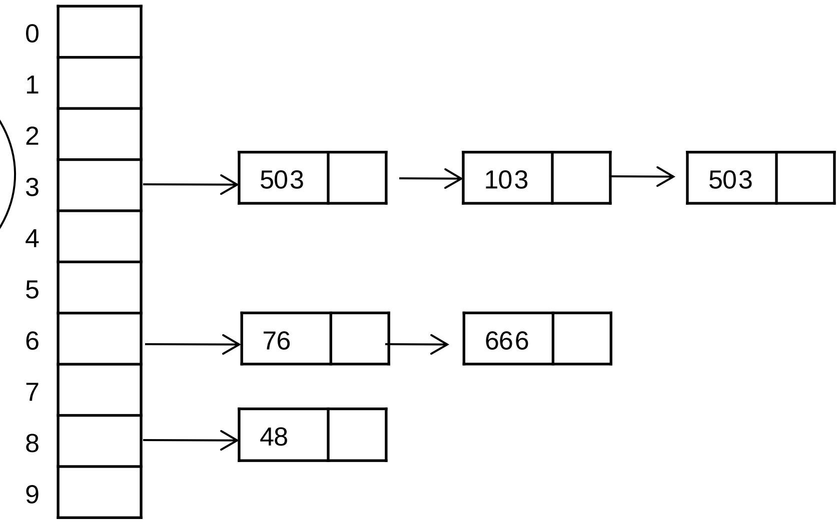
Logical operator:

$\text{Supplier} \bowtie_{\text{sno}=\text{sno}} \text{Supply}$

Build phase

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

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

Probe phase



If |R|=|S|=n,
what is the runtime?

O(n)

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

2. Hash Join

Changed join order

Logical operator:

$$\text{Supply} \bowtie_{\text{sno}=\text{sno}} \text{Supplier}$$

```
Supplier(sno, sname, scity, sstate)
Supply(sno, pno, quantity)
```

2. Hash Join

Changed join order

Logical operator:

$$\text{Supply} \bowtie_{\text{sno}=\text{sno}} \text{Supplier}$$

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

for x in Supplier do
   ???
```

```
Supplier(sno, sname, scity, sstate)
Supply(sno, pno, quantity)
```

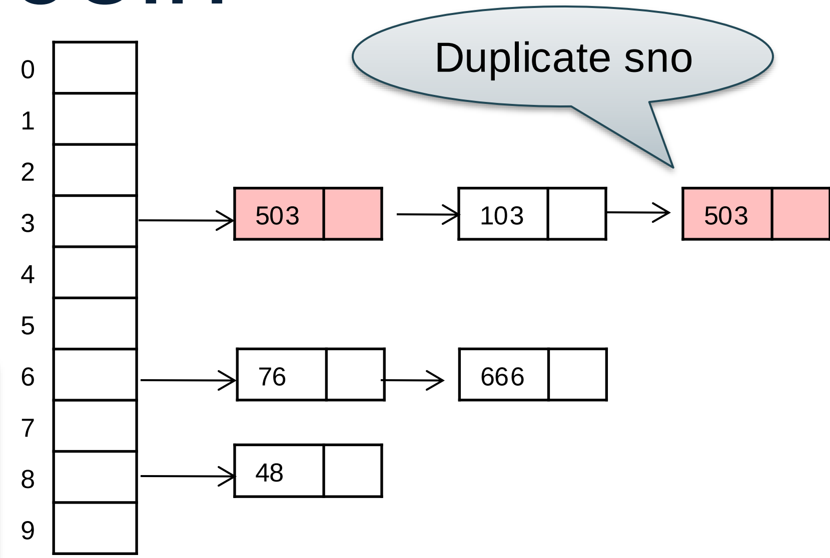
2. Hash Join

Logical operator:

$$\text{Supply} \bowtie_{sno=sno} \text{Supplier}$$

Changed join order

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

for x in Supplier do
  ???
```



```
Supplier(sno, sname, scity, sstate)
Supply(sno, pno, quantity)
```

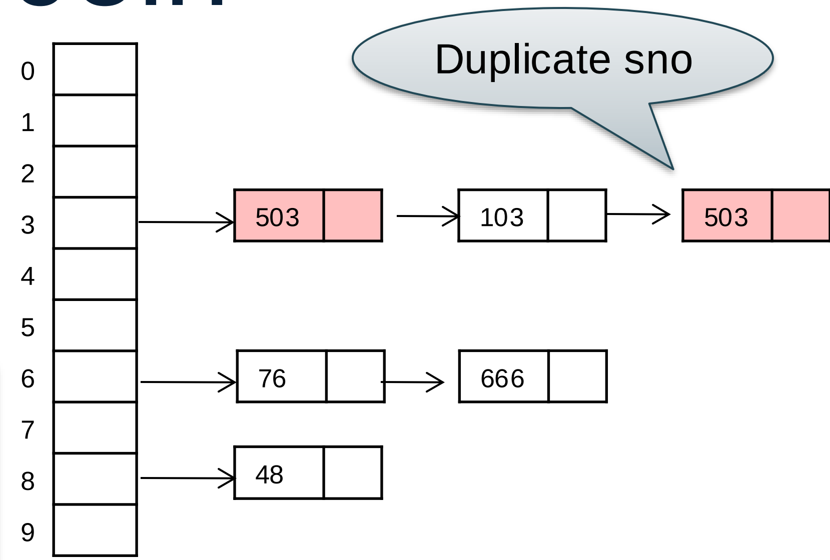
2. Hash Join

Changed join order

Logical operator:

$$\text{Supply} \bowtie_{sno=sno} \text{Supplier}$$

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

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



```
Supplier(sno, sname, scity, sstate)
Supply(sno, pno, quantity)
```

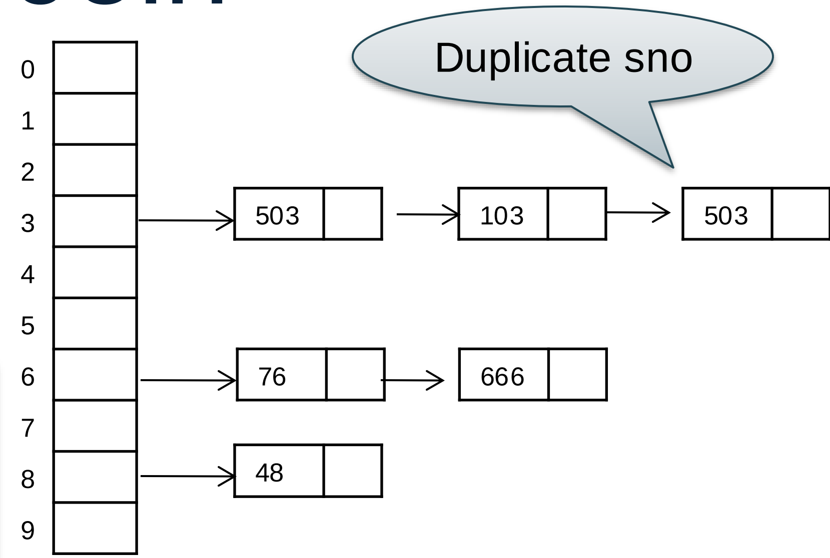
2. Hash Join

Changed join order

Logical operator:

$$\text{Supply} \bowtie_{sno=sno} \text{Supplier}$$

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

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



If |R|=|S|=n,
what is the runtime?

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

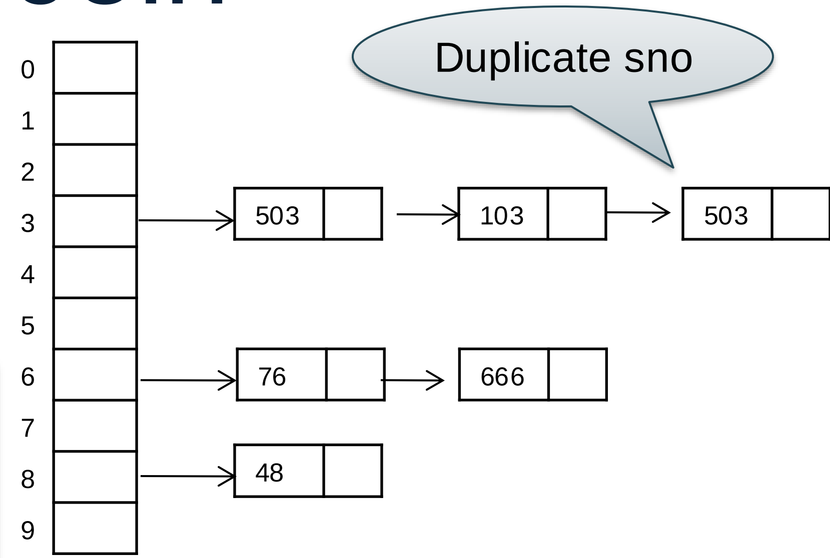
2. Hash Join

Logical operator:

$\text{Supply} \bowtie_{\text{sno=sno}} \text{Supplier}$

Changed join order

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

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



If |R|=|S|=n,
what is the runtime?

O(n)
But can be $O(n^2)$

Discussion

- Hash join is most commonly used
- More complicated for disk-resident data:
  - Partitioned hash-join, Hybrid join
- Convention:
  - Build on outer table, probe on inner table
  - But some people use opposite convention

```
Supplier(sno, sname, scity, sstate)
Supply(sno, pno, quantity)
```

3. Merge Join

Logical operator:

$\text{Supplier} \bowtie_{\text{sno}=\text{sno}} \text{Supply}$

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

```
Supplier(sno, sname, scity, sstate)
Supply(sno, pno, quantity)
```

3. Merge Join

Logical operator:

$$\text{Supplier} \bowtie_{sno=sno} \text{Supply}$$

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

```
Supplier(sno, sname, scity, sstate)
Supply(sno, pno, quantity)
```

3. Merge Join

Logical operator:

$\text{Supplier} \bowtie_{sno=sno} \text{Supply}$

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

```
Supplier(sno, sname, scity, sstate)
Supply(sno, pno, quantity)
```

3. Merge Join

Logical operator:

$$\text{Supplier} \bowtie_{\text{sno}=\text{sno}} \text{Supply}$$

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

```
Supplier(sno, sname, scity, sstate)
Supply(sno, pno, quantity)
```

3. Merge Join

Logical operator:

$\text{Supplier} \bowtie_{\text{sno}=\text{sno}} \text{Supply}$

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

```
Supplier(sno, sname, scity, sstate)
Supply(sno, pno, quantity)
```

3. Merge Join

Logical operator:

$\text{Supplier} \bowtie_{sno=sno} \text{Supply}$

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

If |R|=|S|=n,
what is the runtime?

```
Supplier(sno, sname, scity, sstate)
Supply(sno, pno, quantity)
```

3. Merge Join

Logical operator:

$\text{Supplier} \bowtie_{sno=sno} \text{Supply}$

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

If |R|=|S|=n, what is the runtime?

O(n log(n))

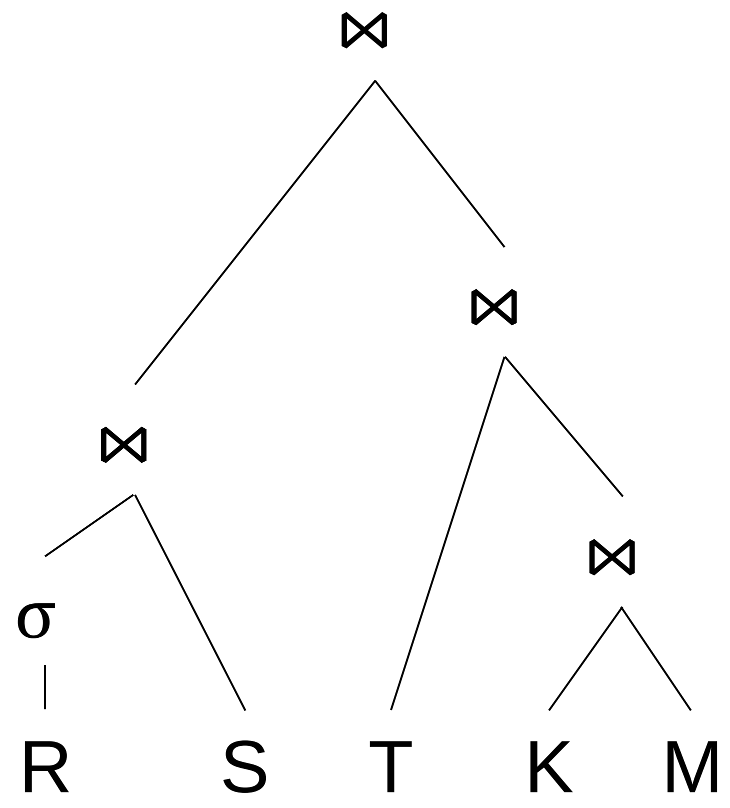
Sumamry of Main Memory Algorithms

- Join ⋈:
  - Nested loop join
  - Hash join
  - Merge join
- Selection σ
  - “on-the-fly”
  - Index-based selection
- Group by γ
  - Hash–based
  - Merge-based

Briefly discuss in class

Iterator Model

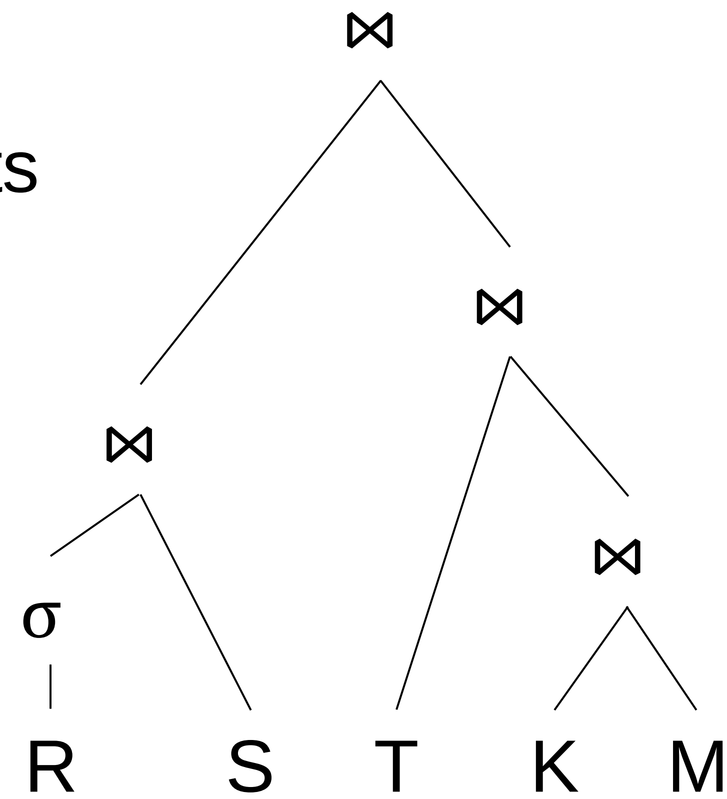
How Do We Combine Them?



How Do We Combine Them?

Option 1:
materialize intermediate results

Option 2:
Pipeline tuples btw. ops

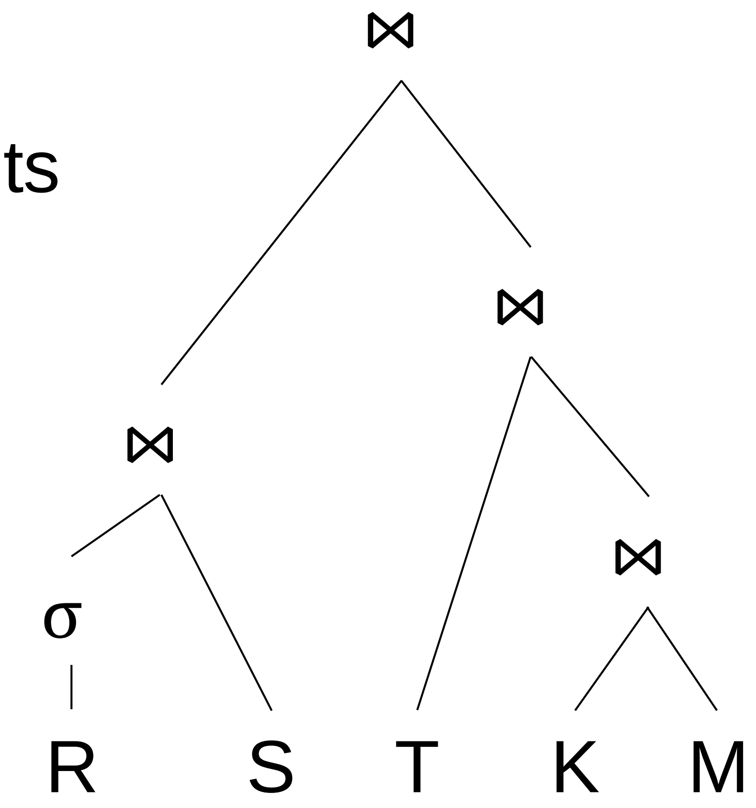


How Do We Combine Them?

Option 1:
materialize intermediate results

Option 2:
Pipeline tuples btw. ops

Implementation:
Operator Interface



Operator Interface

Volcano model:

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

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
- "How to architect…"

Volcano Model

Open()

- Calls open() on the children
- Creates any local data structures

Next()

- May call next() repeatedly on children
- Returns exactly 1 tuple, or EOF

Close()

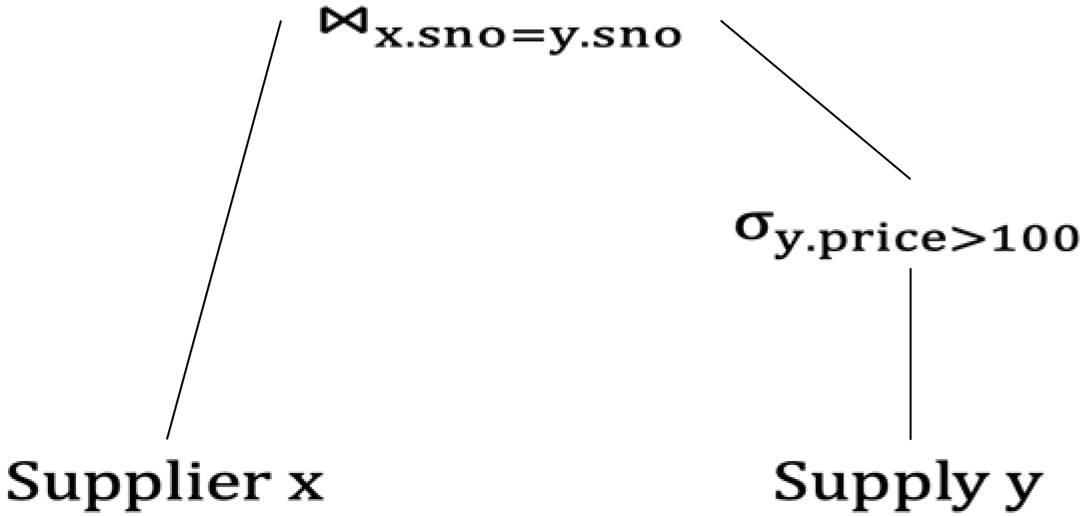
- Free any local memory

Volcano Example

”Normal” hash-join

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

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

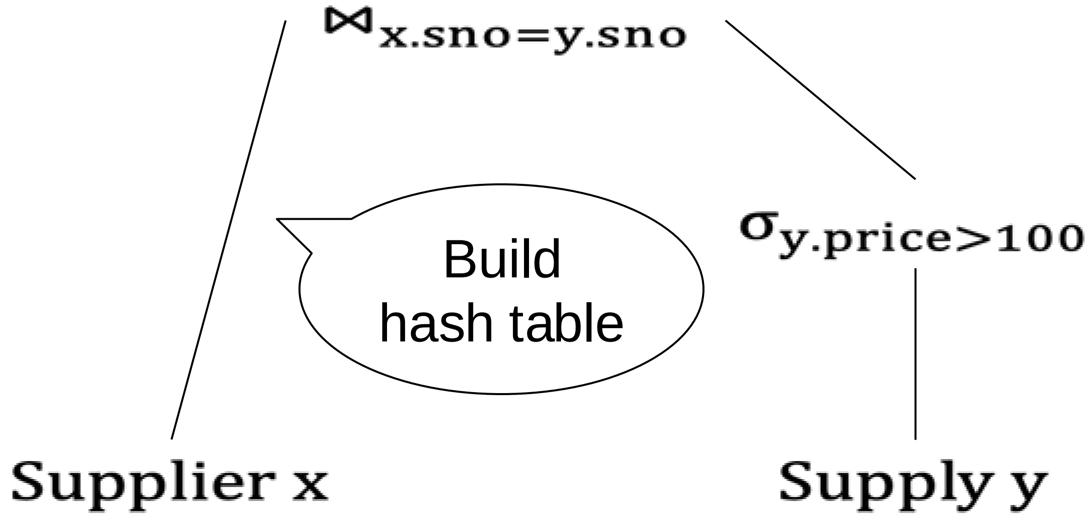
Pipelining changes the order significantly



Volcano Example

”Normal” hash-join

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

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

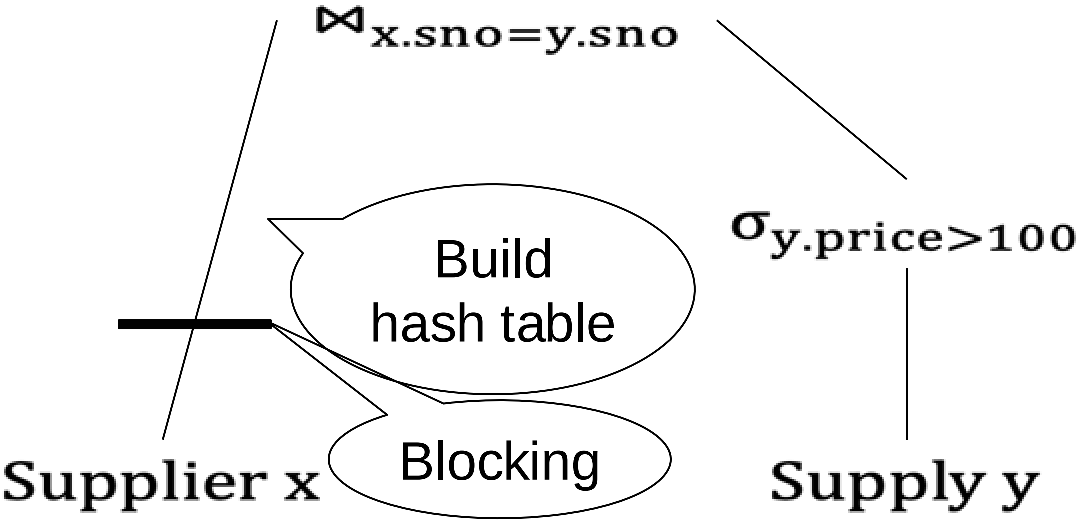


Pipelining changes the order significantly

Volcano Example

”Normal” hash-join

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

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



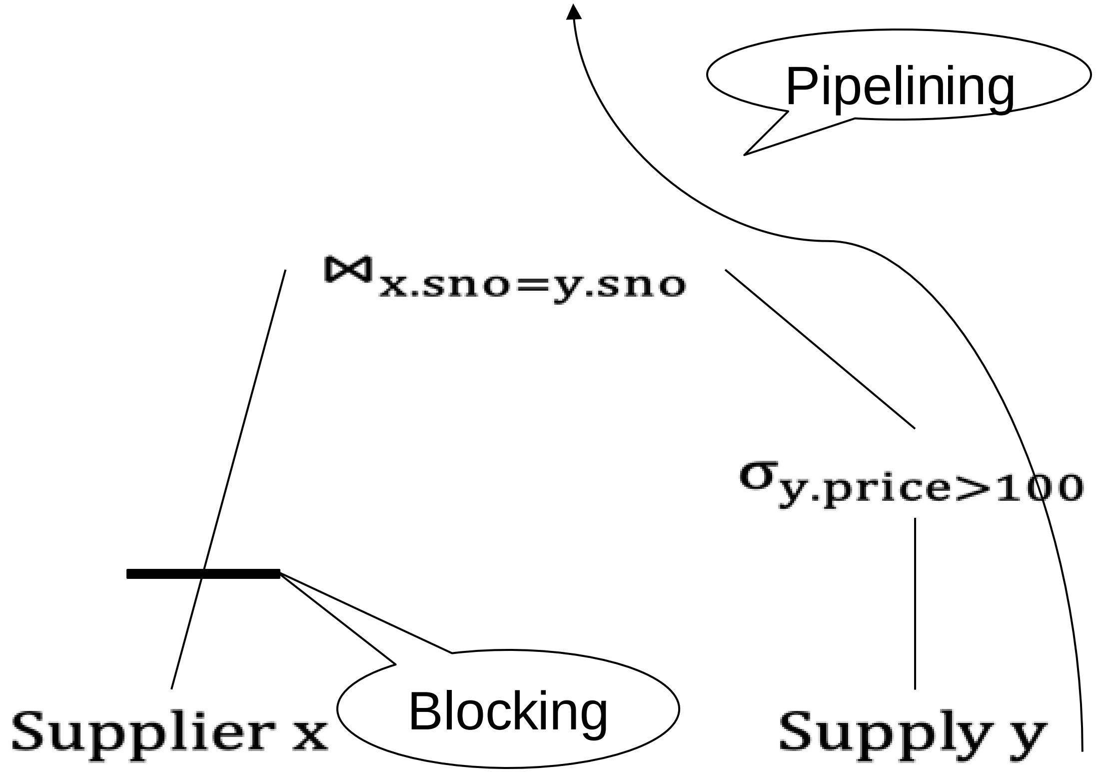
Pipelining changes the order significantly

Volcano Example

”Normal” hash-join

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

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

Pipelining changes the order significantly

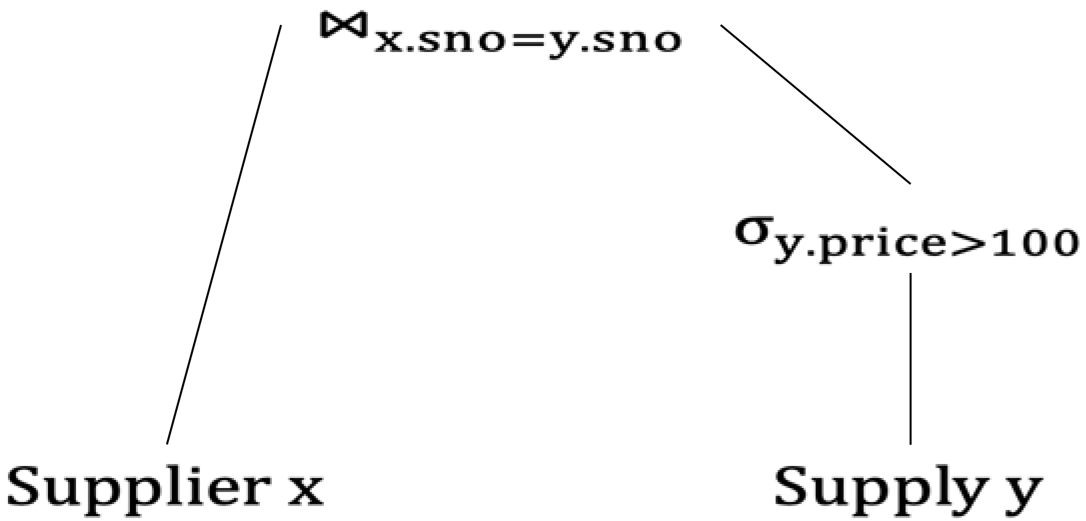


Volcano Example

Details

”Normal” hash-join

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

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



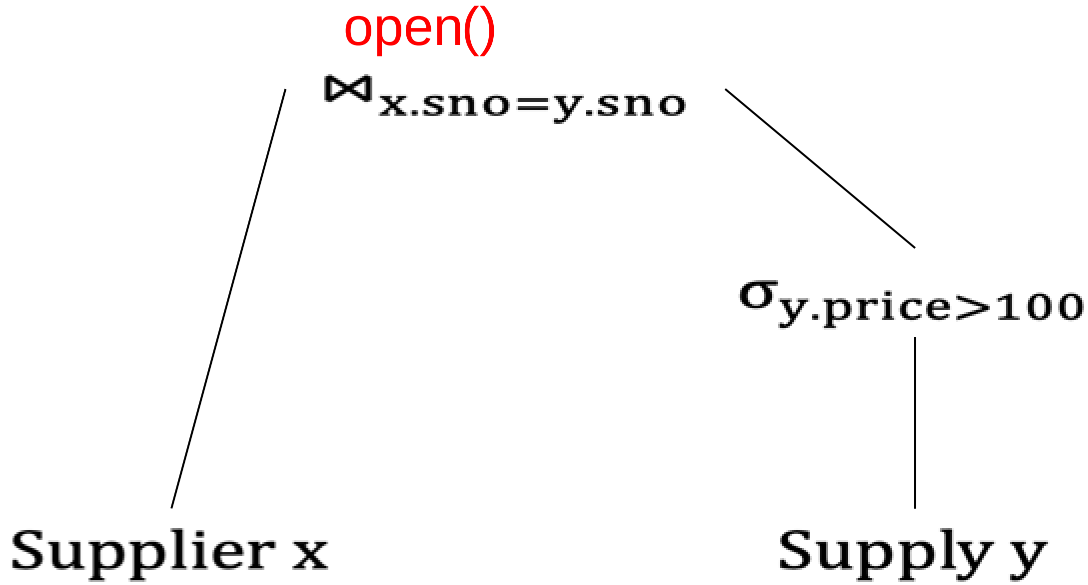
Pipelining changes
the order significantly

Volcano Example

”Normal” hash-join

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

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

Pipelining changes
the order significantly

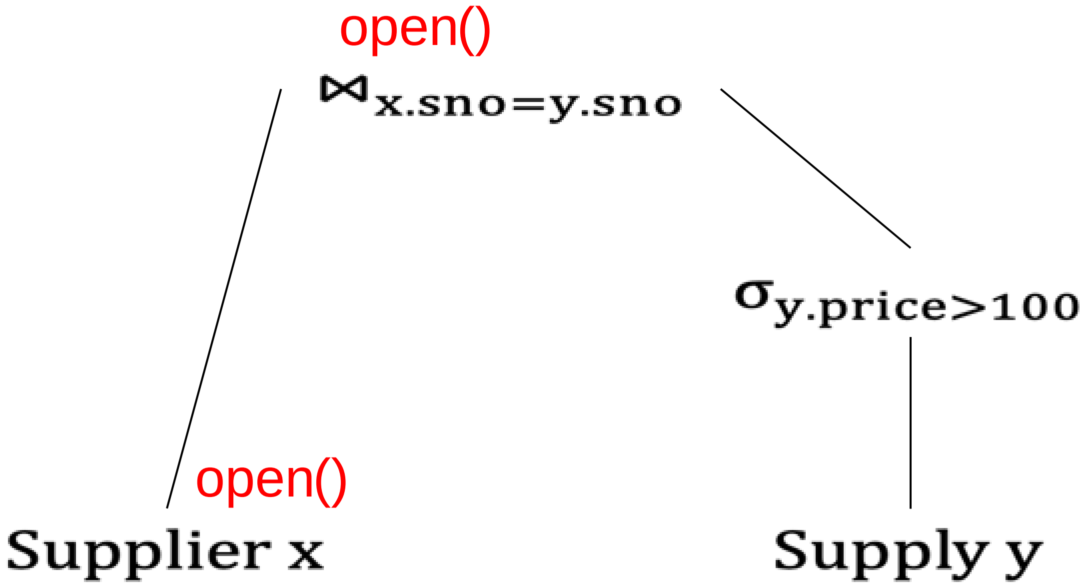


Volcano Example

”Normal” hash-join

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

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

Pipelining changes the order significantly

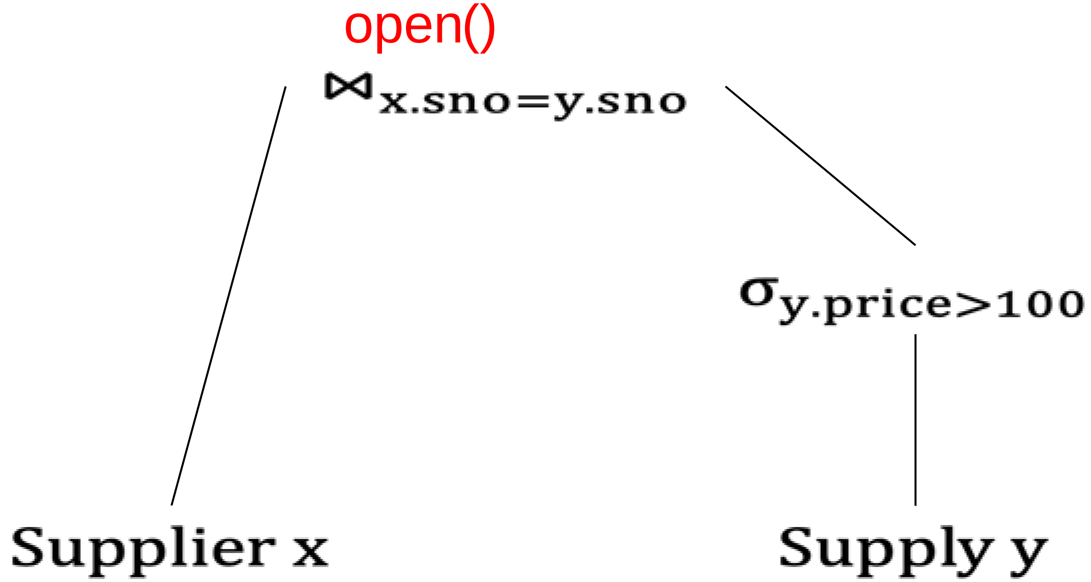


Volcano Example

"Normal" hash-join

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

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

Pipelining changes the order significantly

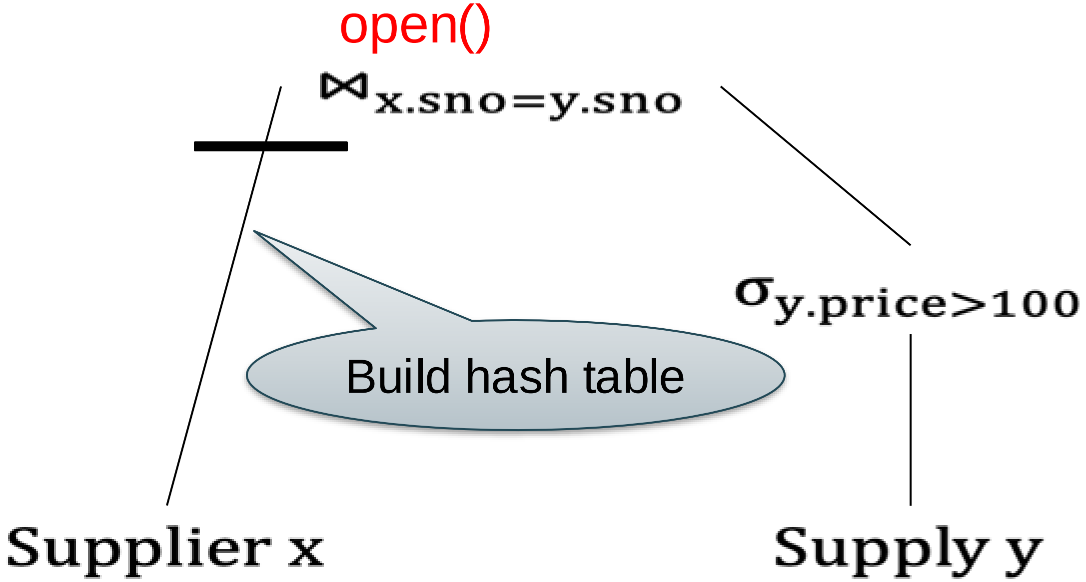


Volcano Example

"Normal" hash-join

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

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

Pipelining changes the order significantly

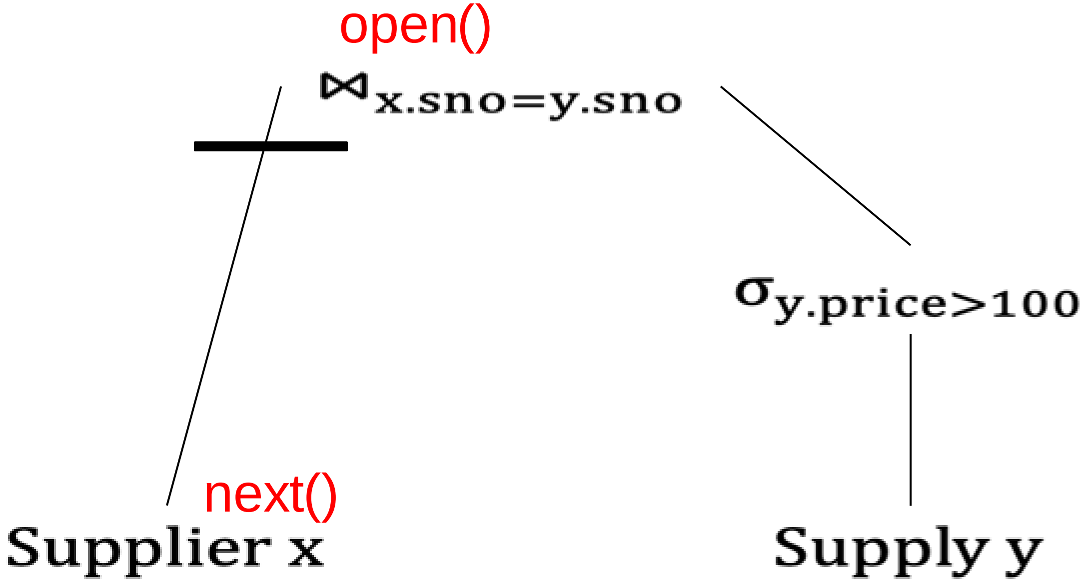


Volcano Example

"Normal" hash-join

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

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

Pipelining changes the order significantly

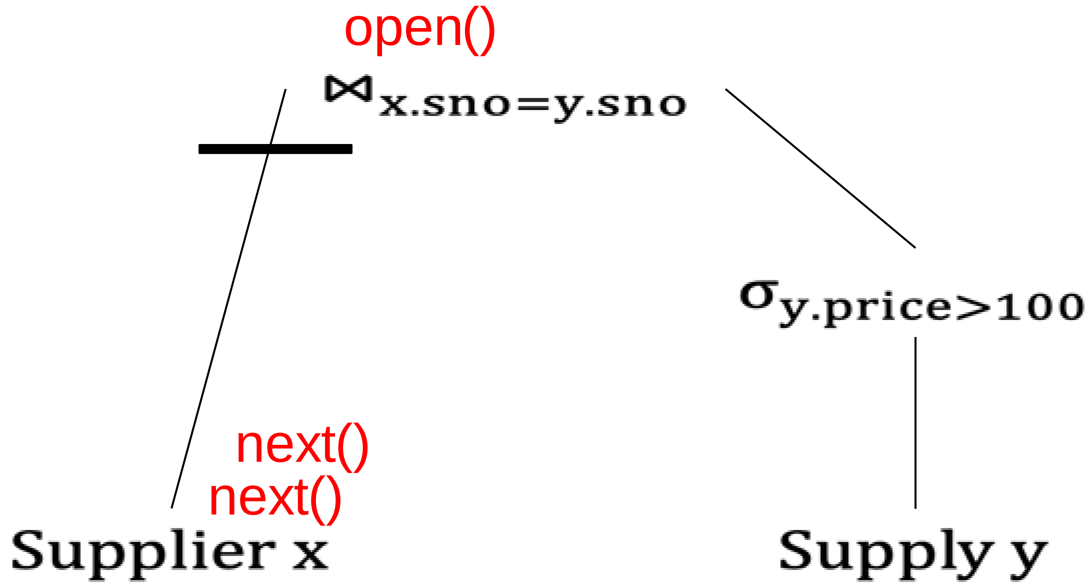


Volcano Example

"Normal" hash-join

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

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

Pipelining changes the order significantly

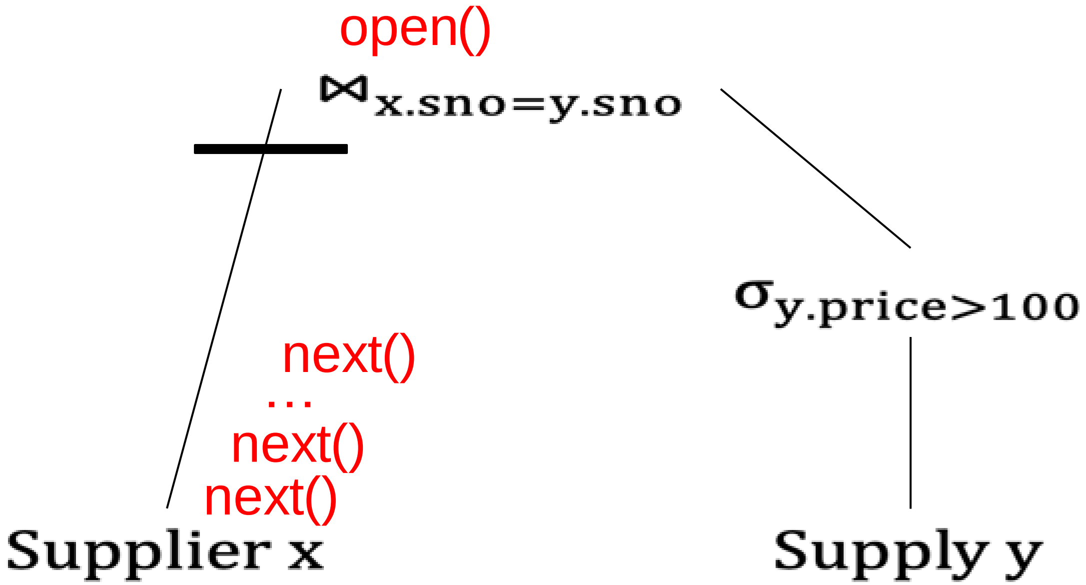


Volcano Example

"Normal" hash-join

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

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

Pipelining changes the order significantly

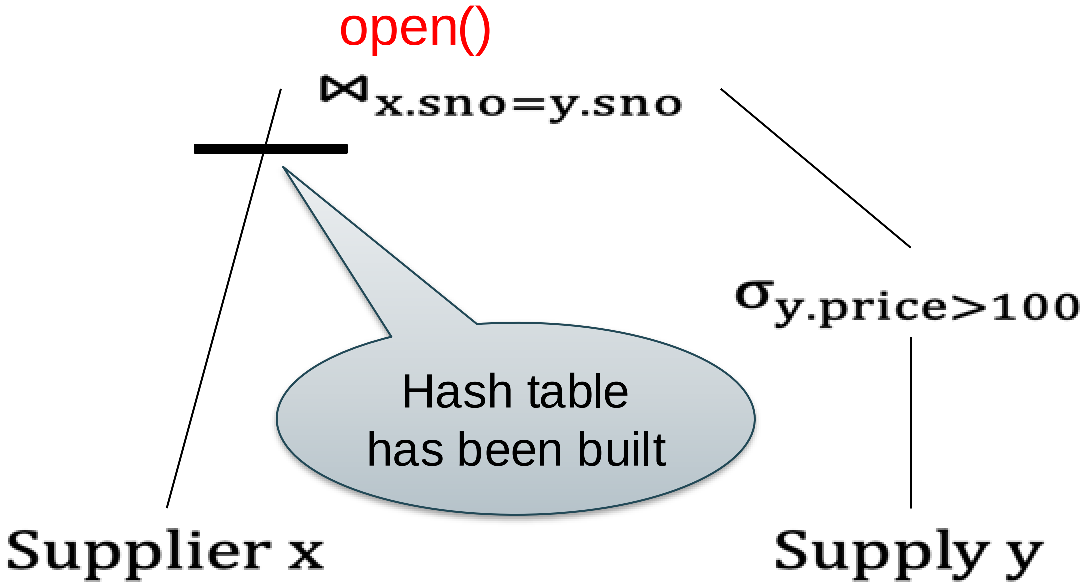


Volcano Example

"Normal" hash-join

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

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

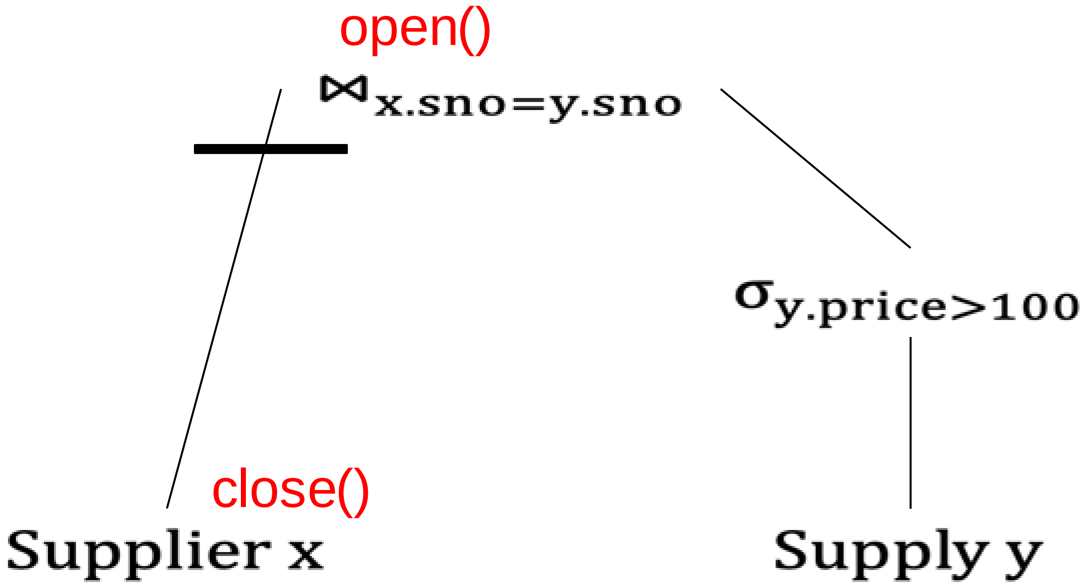
Pipelining changes the order significantly



Volcano Example

"Normal" hash-join

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

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



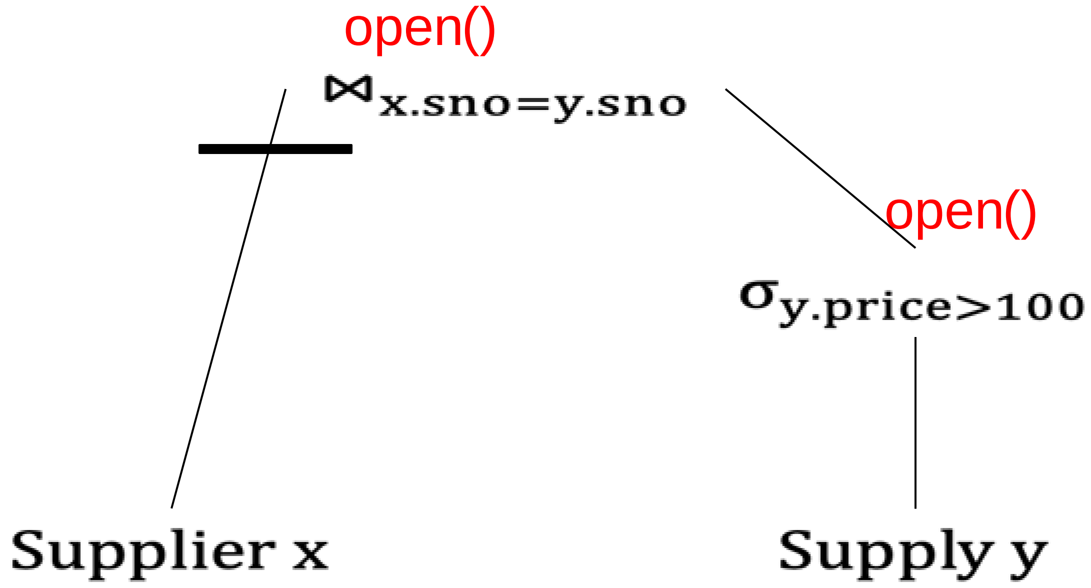
Pipelining changes the order significantly

Volcano Example

”Normal” hash-join

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

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

Pipelining changes the order significantly

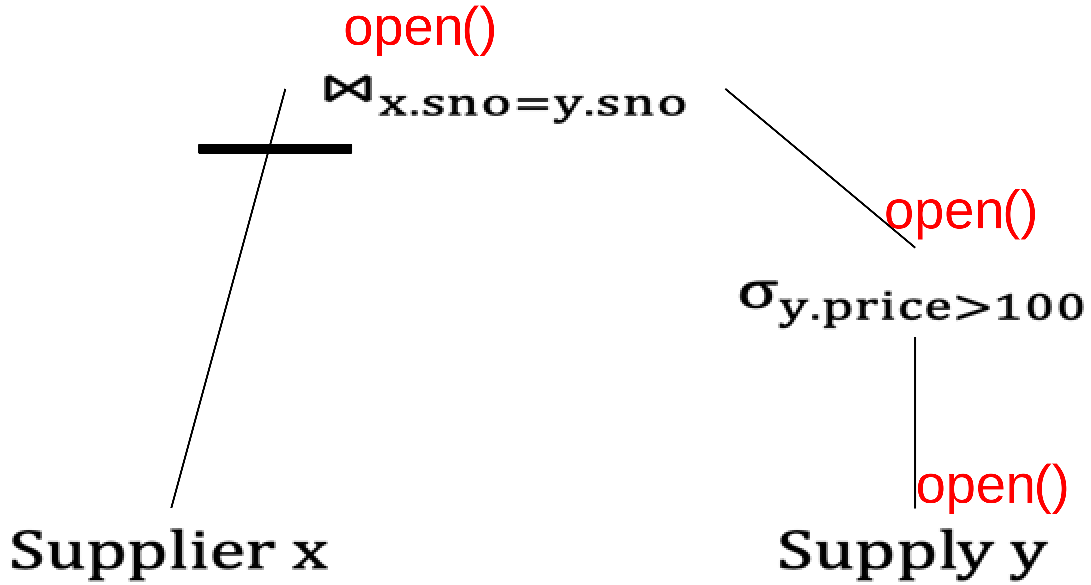


Volcano Example

"Normal" hash-join

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

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

Pipelining changes the order significantly

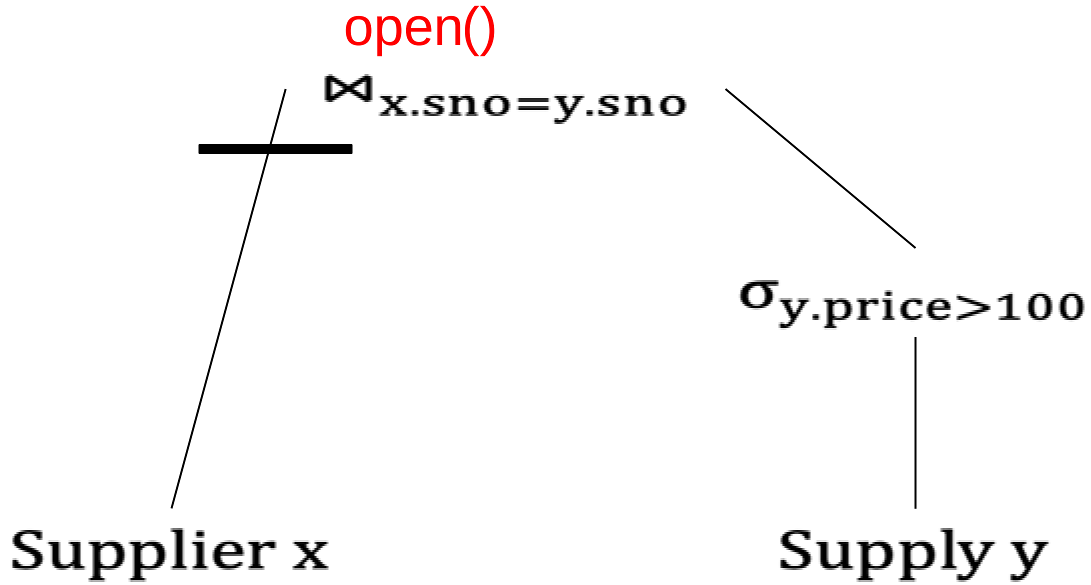


Volcano Example

”Normal” hash-join

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

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

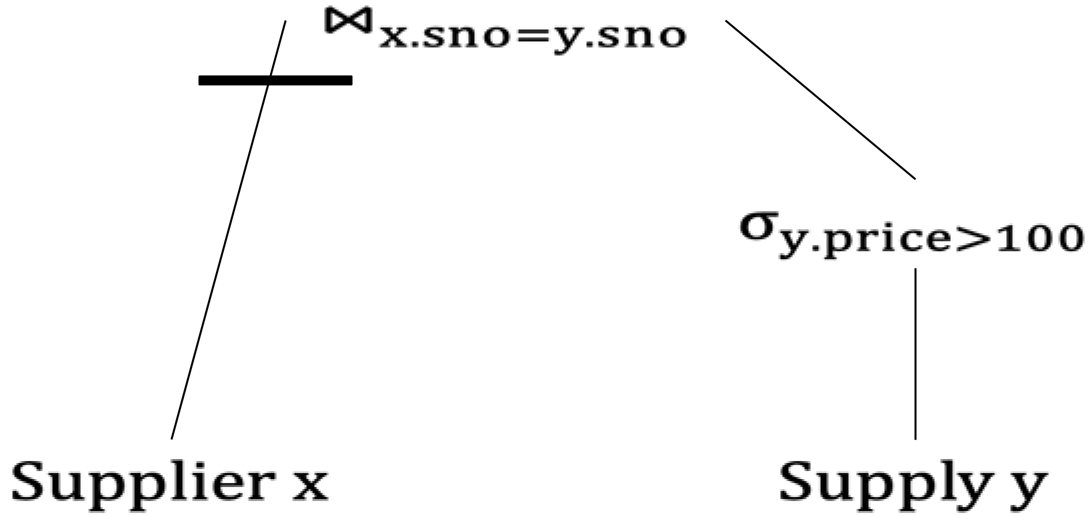
Pipelining changes the order significantly



Volcano Example

”Normal” hash-join

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

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



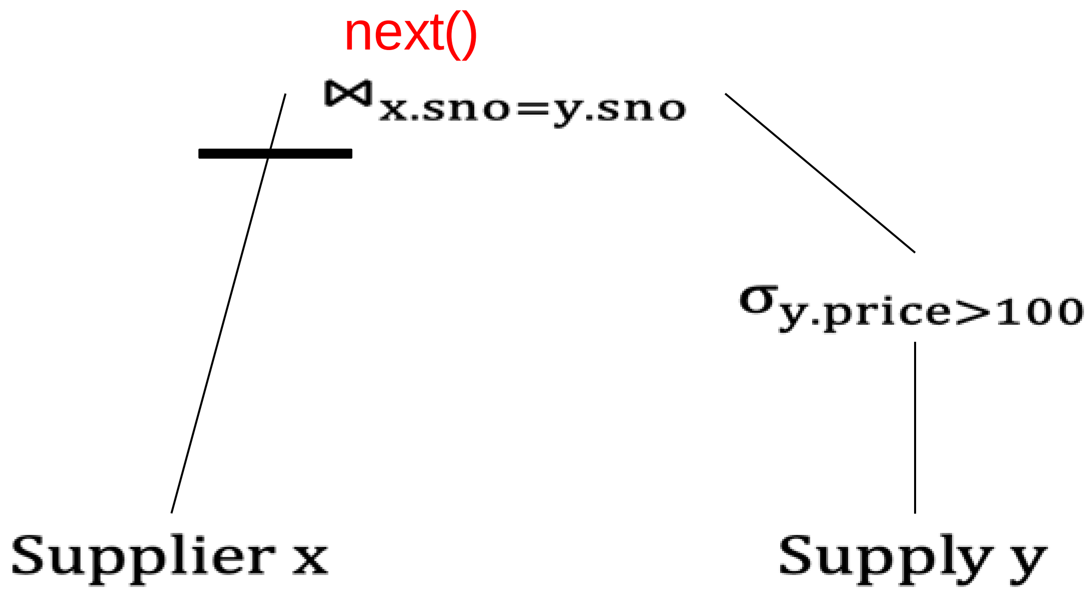
Pipelining changes
the order significantly

Volcano Example

"Normal" hash-join

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

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

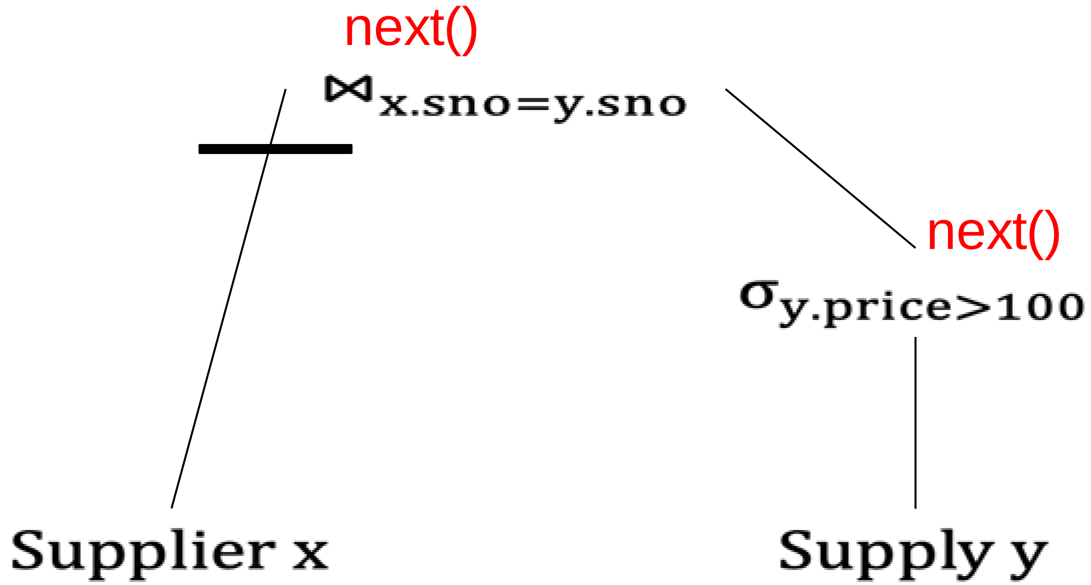
Pipelining changes the order significantly



Volcano Example

”Normal” hash-join

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

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



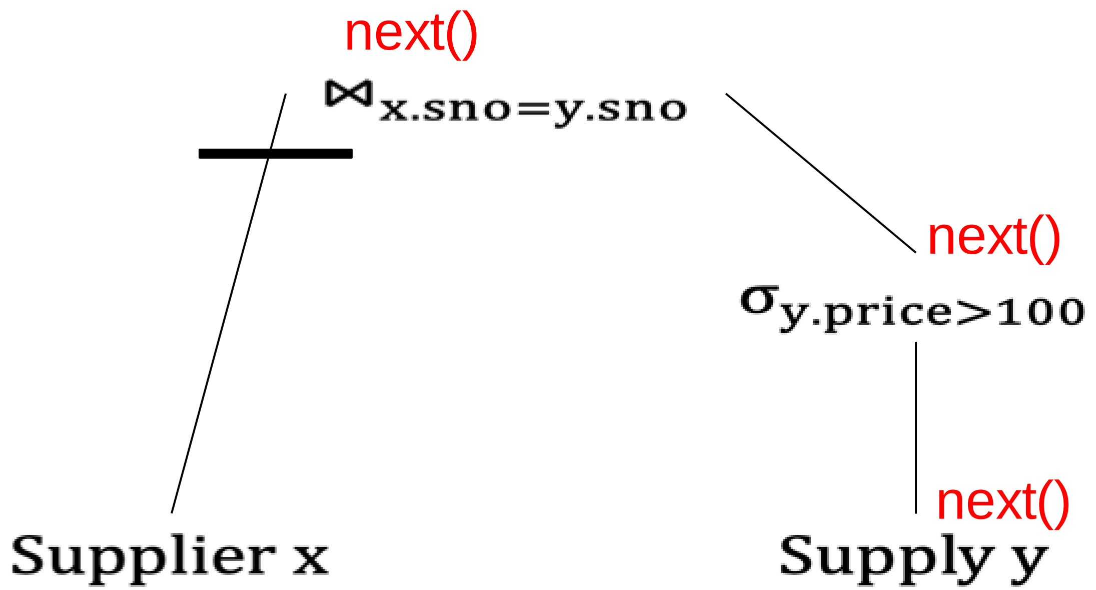
Pipelining changes the order significantly

Volcano Example

”Normal” hash-join

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

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

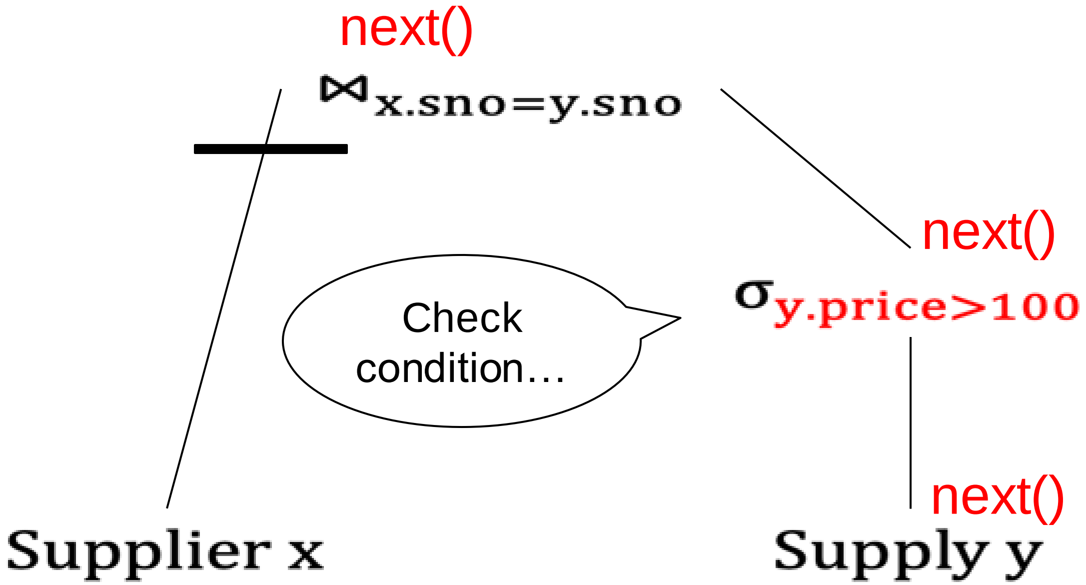
Pipelining changes
the order significantly



Volcano Example

”Normal” hash-join

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

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



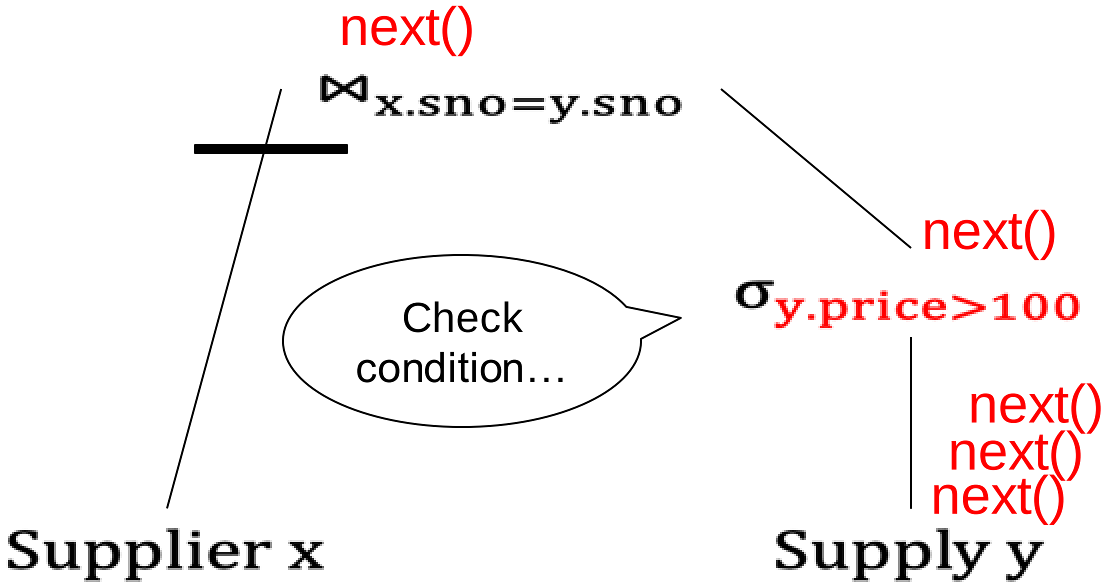
Pipelining changes the order significantly

Volcano Example

”Normal” hash-join

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

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

Pipelining changes the order significantly

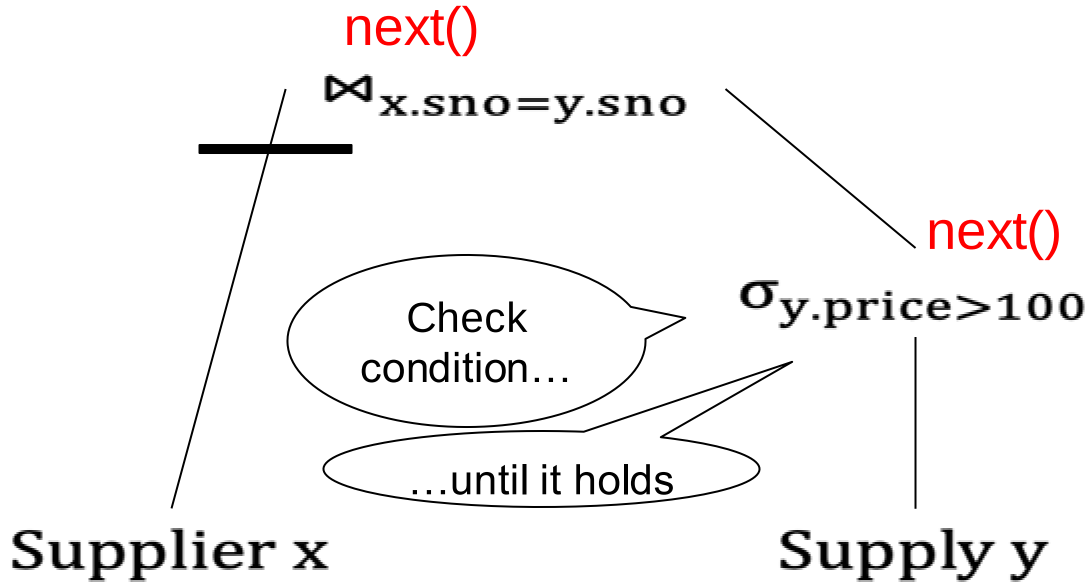


Volcano Example

"Normal" hash-join

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

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

Pipelining changes the order significantly

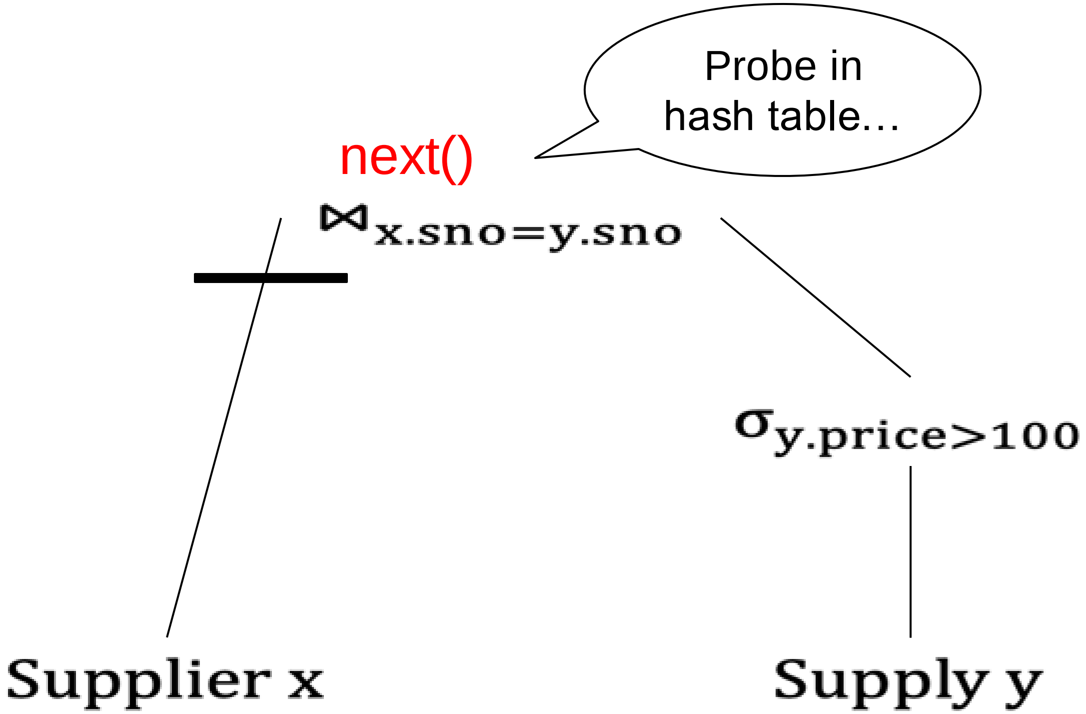


Volcano Example

"Normal" hash-join

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

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

Pipelining changes
the order significantly

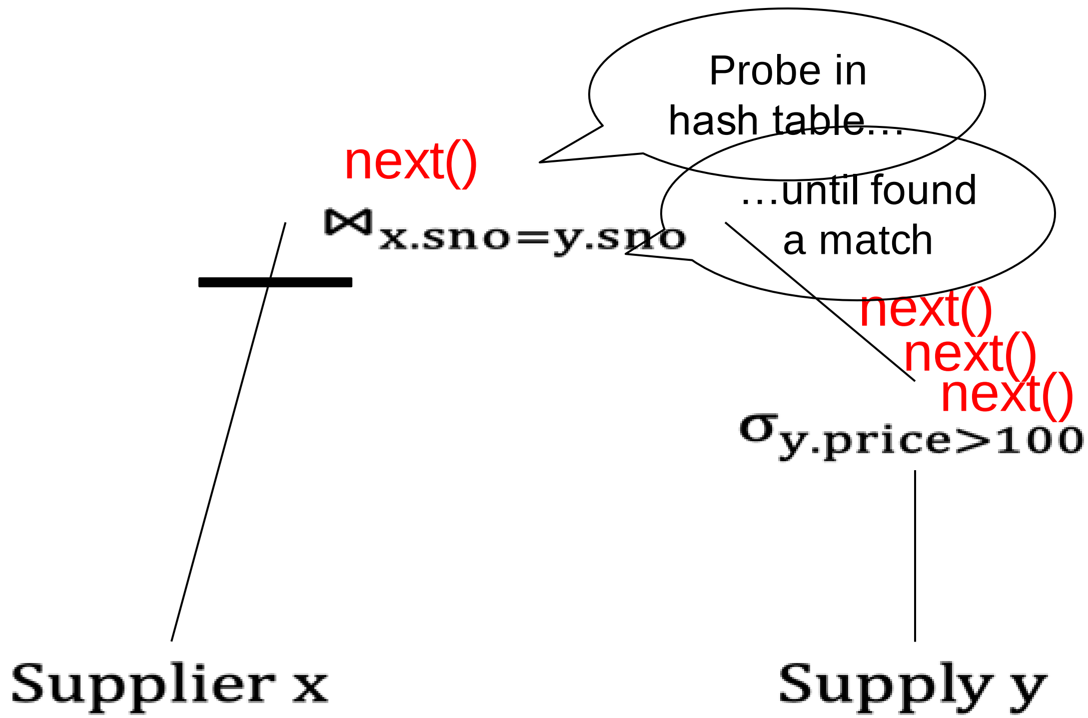


Volcano Example

"Normal" hash-join

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

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

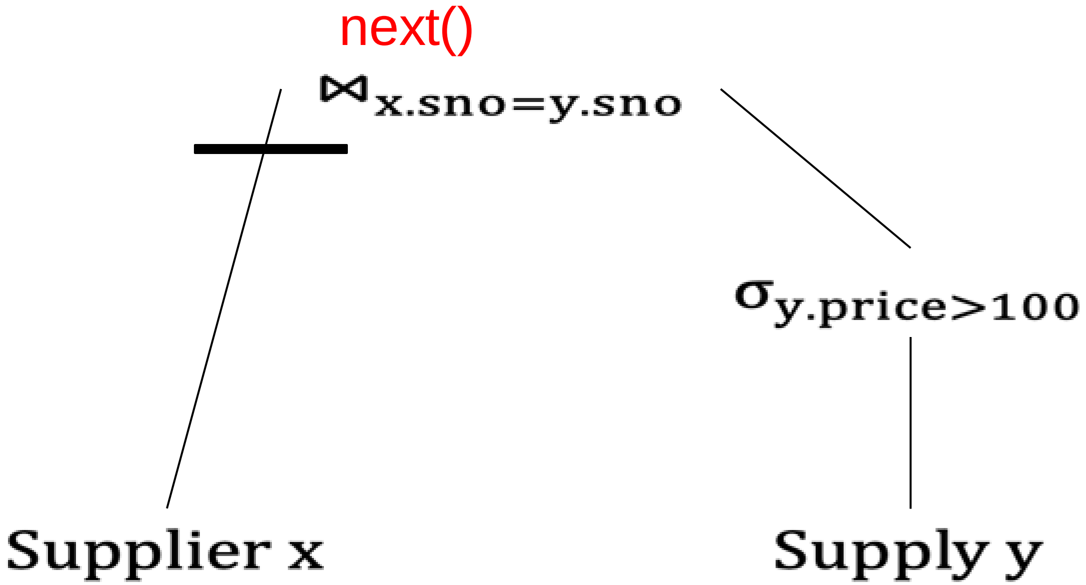
Pipelining changes the order significantly



Volcano Example

"Normal" hash-join

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

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



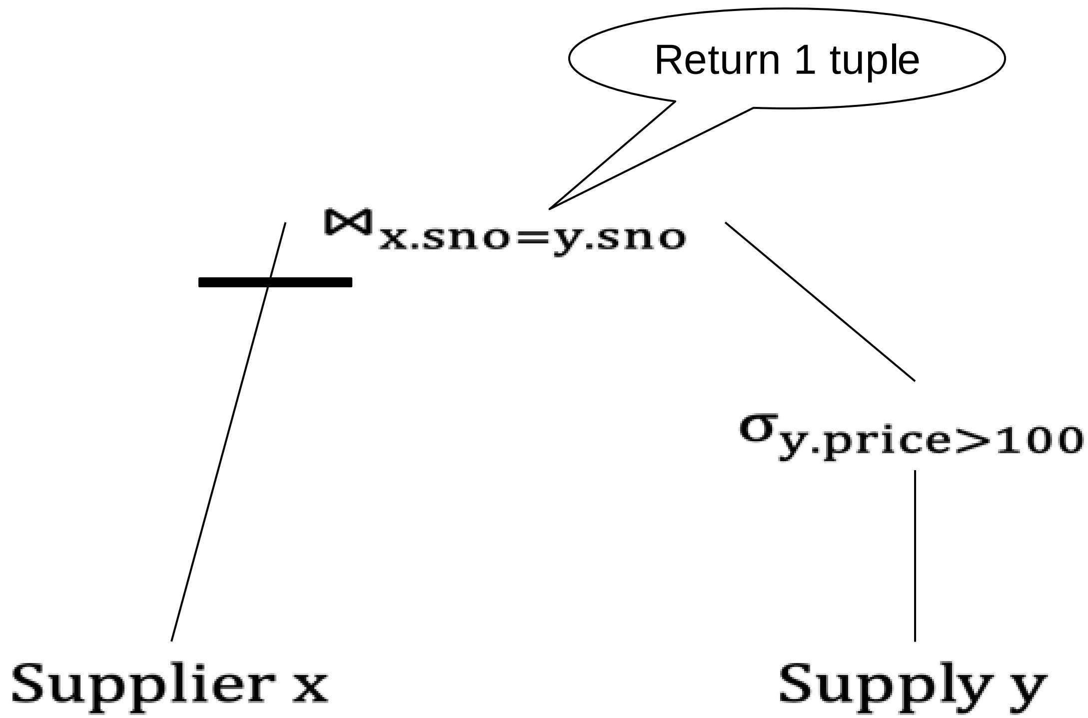
Pipelining changes the order significantly

Volcano Example

”Normal” hash-join

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

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

Pipelining changes the order significantly

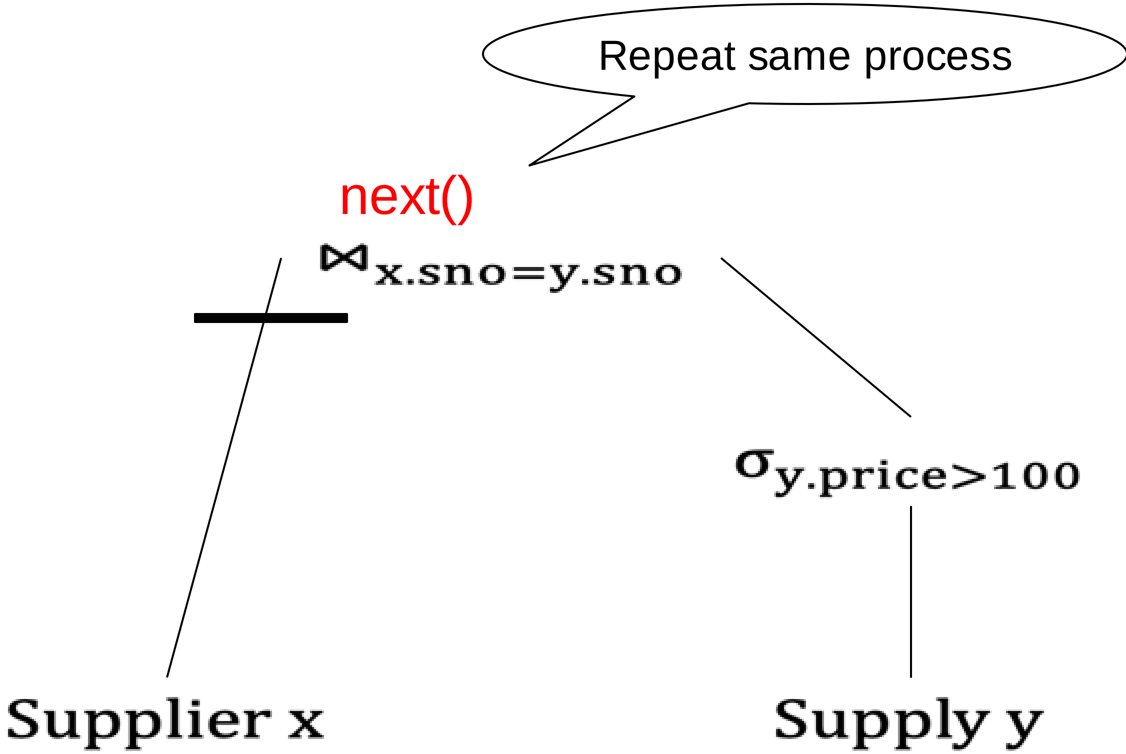


Volcano Example

”Normal” hash-join

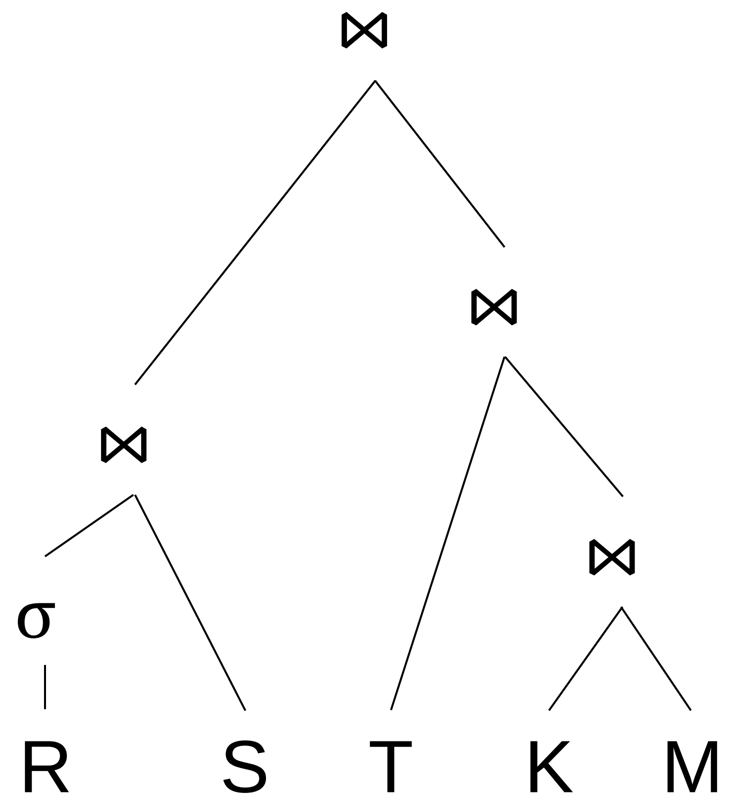
```
for x in Supplier do
      insert(x.sno, x)

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

Pipelining changes
the order significantly



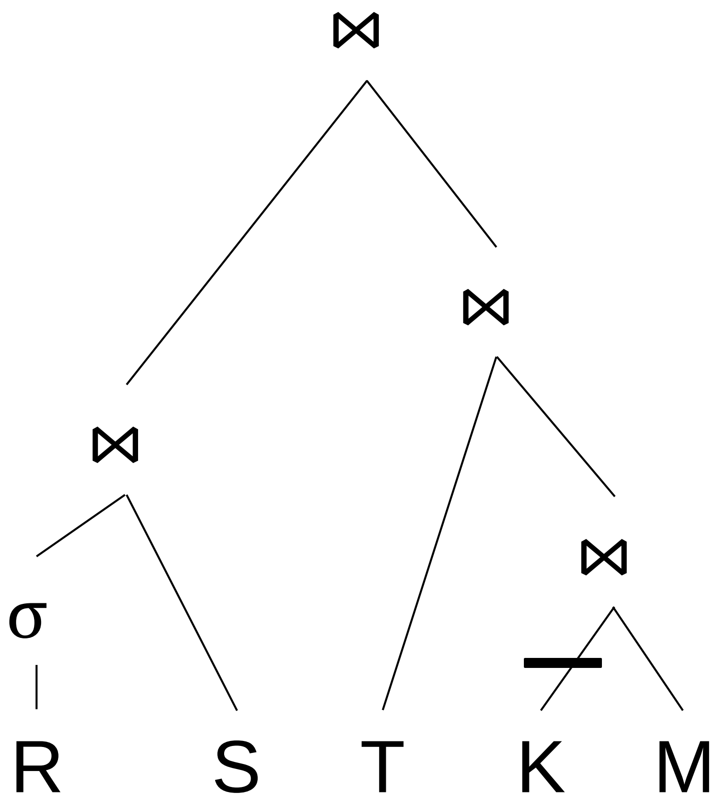
Volcano Model

Usually: block the outer relation, pipeline the inner relation



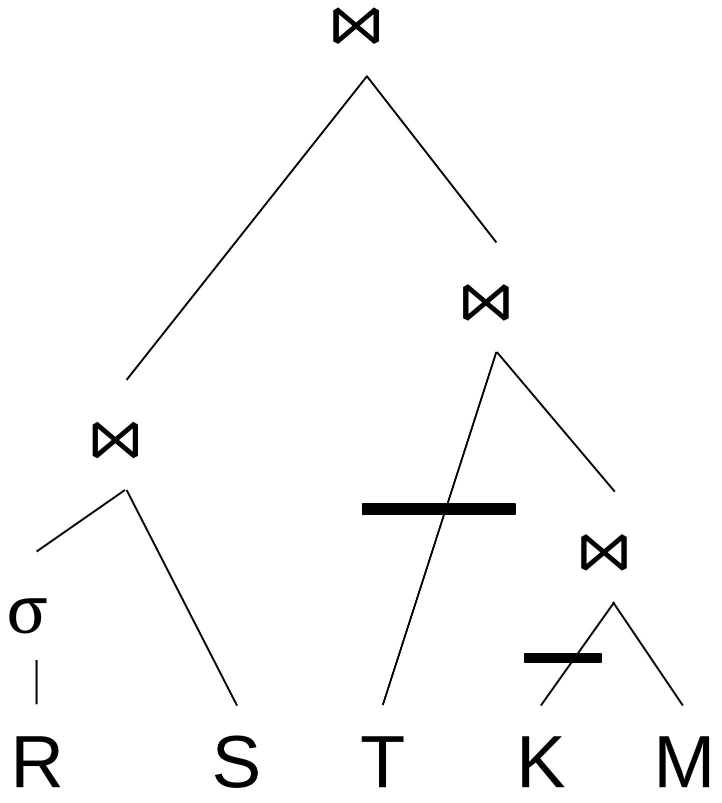
Volcano Model

Usually: block the outer relation,
pipeline the inner relation



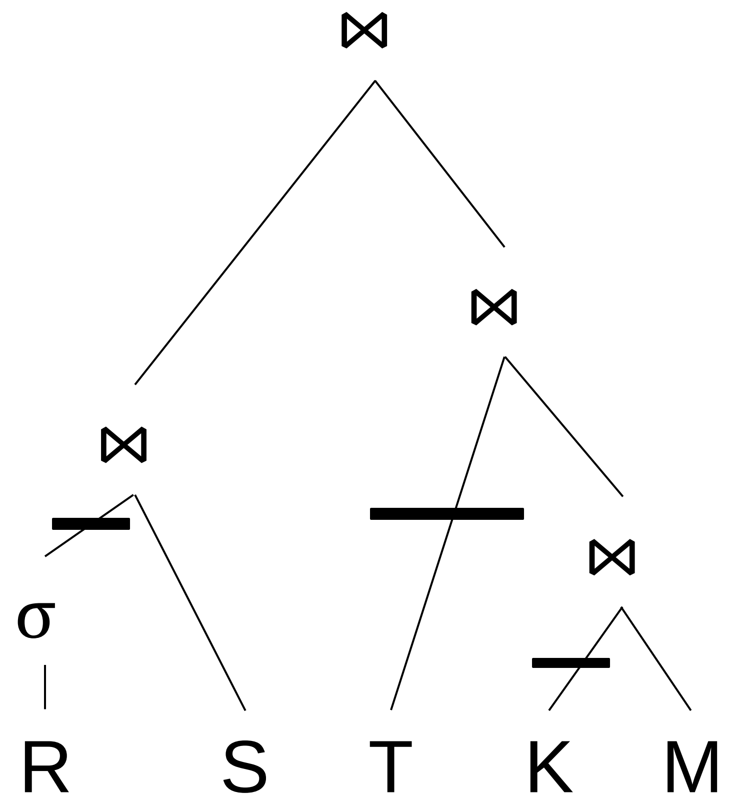
Volcano Model

Usually: block the outer relation, pipeline the inner relation



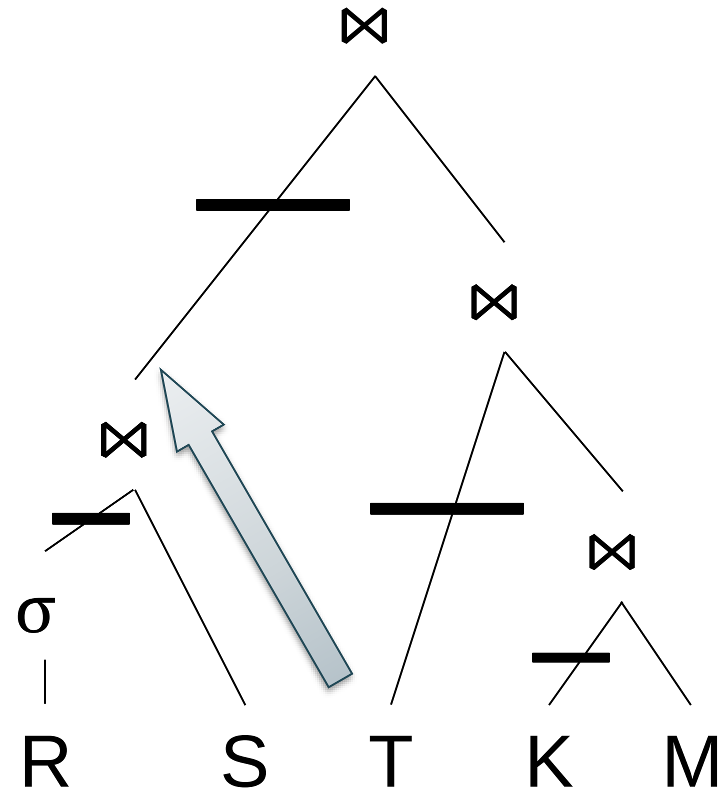
Volcano Model

Usually: block the outer relation, pipeline the inner relation



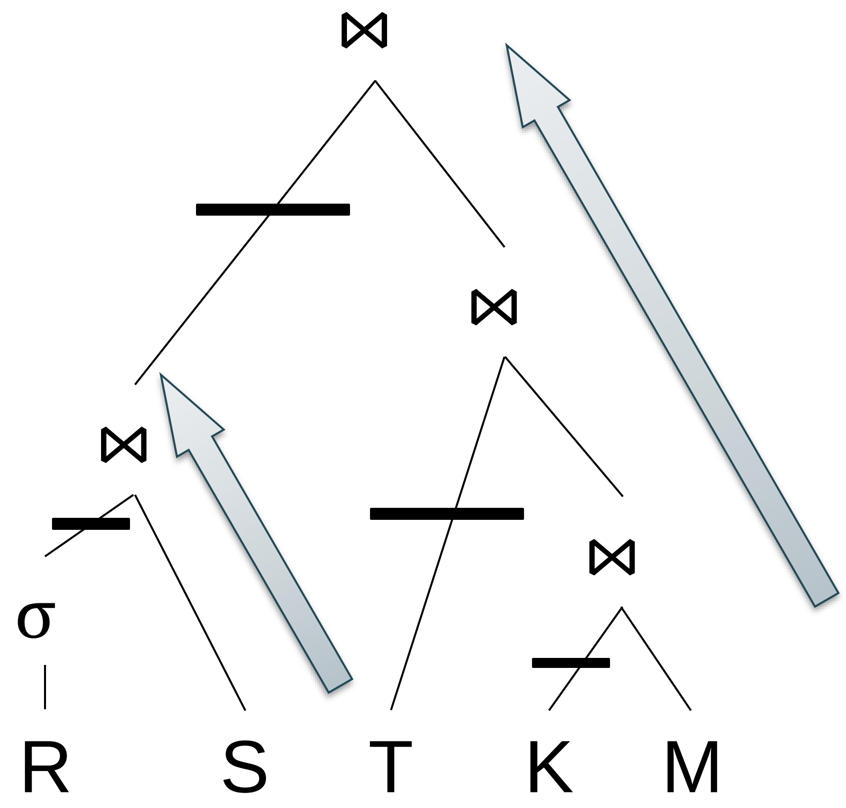
Volcano Model

Usually: block the outer relation, pipeline the inner relation



Volcano Model

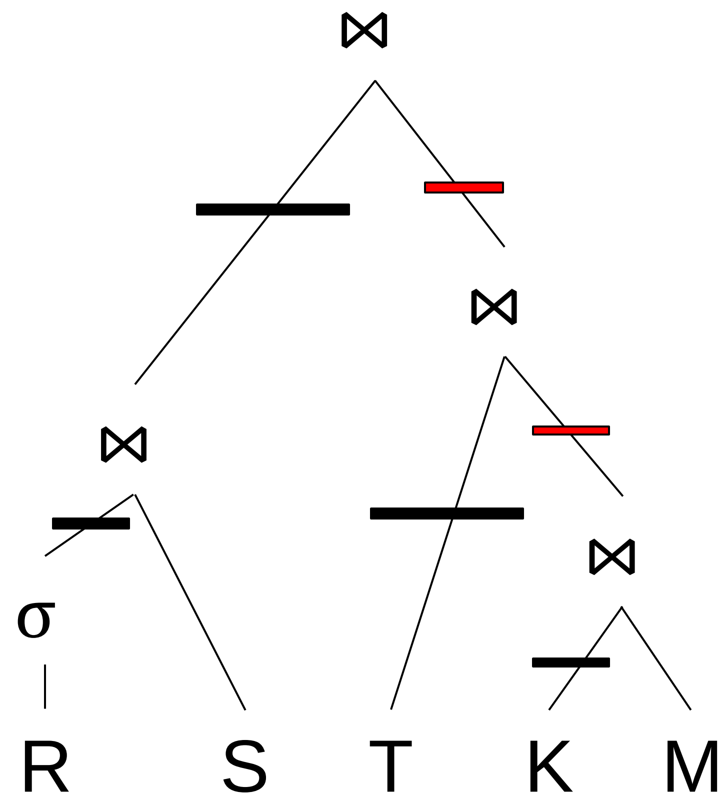
Usually: block the outer relation, pipeline the inner relation



Volcano Model

Usually: block the outer relation, pipeline the inner relation

But may also block everything



Volcano Model

```
interface Operator {
  // initializes operator state
  // and sets parameters
  void open (...);

  // calls next() on its inputs
  // processes an input tuple
  // produces output tuple(s)
  // returns null when done
  Tuple next ();

  // cleans up (if any)
  void close ();
}
```

Volcano Model

```
interface Operator {
  // initializes operator state
  // and sets parameters
  void open (...);

  // calls next() on its inputs
  // processes an input tuple
  // produces output tuple(s)
  // returns null when done
  Tuple next ();

  // cleans up (if any)
  void close ();
}
```

Example selection operator

```
class Select implements Operator {...
  void open (Predicate p,
             Operator c) {
    this.p = p; this.c = c; c.open();
    }
```

Volcano Model

```
interface Operator {
  // initializes operator state
  // and sets parameters
  void open (...);

  // calls next() on its inputs
  // processes an input tuple
  // produces output tuple(s)
  // returns null when done
  Tuple next ();

  // cleans up (if any)
  void close ();
}
```

Example selection operator

```
class Select implements Operator {...
  void open (Predicate p,
             Operator c) {
    this.p = p; this.c = c; c.open();
    }
  Tuple next () {
    boolean found = false;
    Tuple r = null;
    while (!found) {
       r = c.next();
       if (r == null) break;
       found = p(r);
    }
    return r;
  }
```

Volcano Model

```
interface Operator {
  // initializes operator state
  // and sets parameters
  void open (...);

  // calls next() on its inputs
  // processes an input tuple
  // produces output tuple(s)
  // returns null when done
  Tuple next ();

  // cleans up (if any)
  void close ();
}
```

Example selection operator

```
class Select implements Operator {...
  void open (Predicate p,
             Operator c) {
    this.p = p; this.c = c; c.open();
    }
  Tuple next () {
    boolean found = false;
    Tuple r = null;
    while (!found) {
       r = c.next();
       if (r == null) break;
       found = p(r);
    }
    return r;
  }
  void close () { c.close(); }
}
```

Volcano Model

Query plan execution

```
Operator q = parse("SELECT ..."); # sql -> root of an op tree
q = optimize(q); # op tree -> optimized op tree
```

Volcano Model

Query plan execution

```
Operator q = parse(“SELECT ...”); # sql -> root of an op tree
q = optimize(q); # op tree -> optimized op tree

q.open();
```

Volcano Model

Query plan execution

```
Operator q = parse("SELECT ..."); # sql -> root of an op tree
q = optimize(q); # op tree -> optimized op tree

q.open();
while (true) {
  Tuple t = q.next();
  if (t == null) break; # end of results
  else printOnScreen(t); # output tuple
}
```

Volcano Model

Query plan execution

```
Operator q = parse(“SELECT ...”); # sql -> root of an op tree
q = optimize(q); # op tree -> optimized op tree

q.open();
while (true) {
  Tuple t = q.next();
  if (t == null) break; # end of results
  else printOnScreen(t); # output tuple
}
q.close();
```

Pipeline v.s. Blocking

- Pipeline
  - Tuples move all the way through up the query plan
  - Advantages: speed
  - Disadvantage: need all hash tables in memory
- Blocking
  - Compute and store on disk entire subplan
  - Advantage: needs less memory
  - Disadvantage: slower

Today’s Paper

How to Architect a Query Compiler

Query execution: interpret v.s. compile

- What are the pros/cons?
- What was the traditional approach?
- What changed recently?

Data-Driven Model

Each operator exports four methods:

- Open()
- Produce()
- Consume()
- Close()

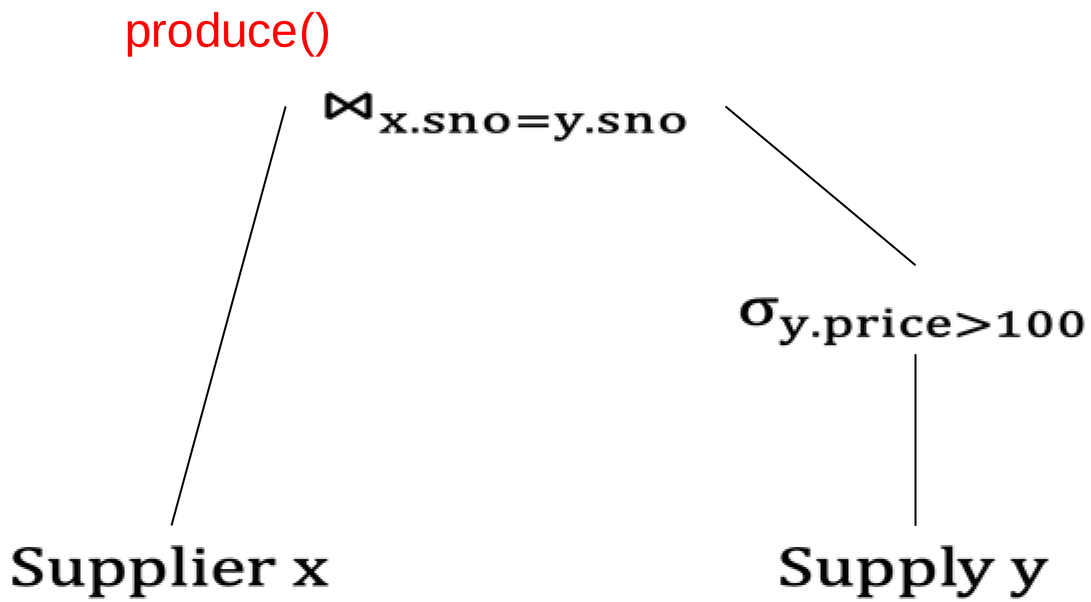
called **once** by parent

called **repeatedly** by children

Example

"Normal" hash-join

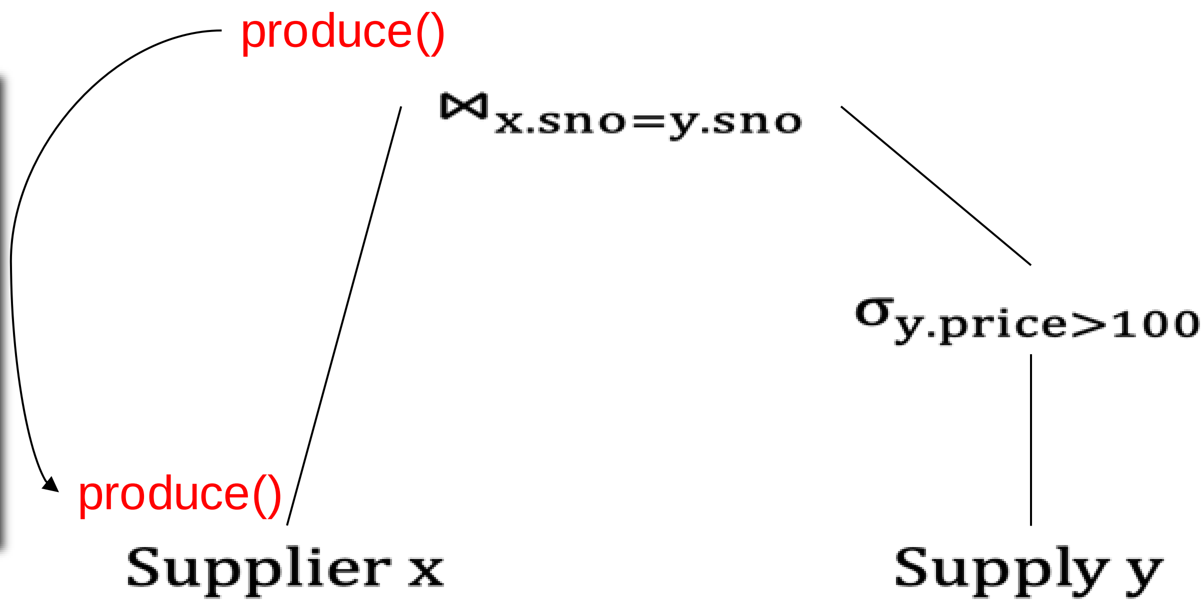
```
for x in Supplier do
      insert(x.sno, x)

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



Example

"Normal" hash-join

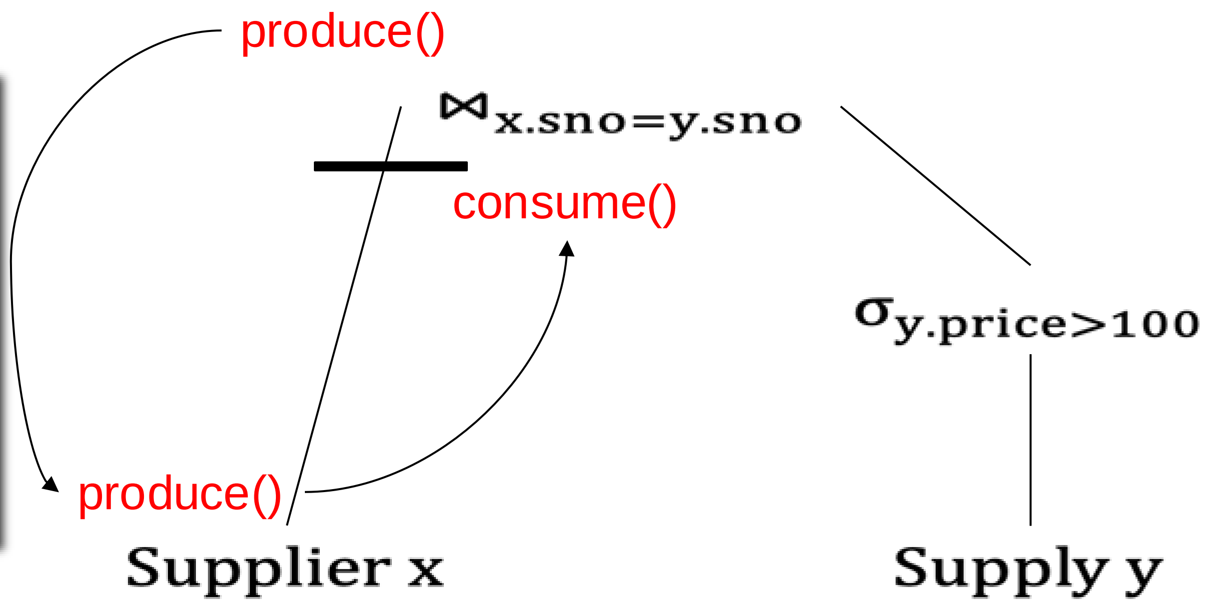
```
for x in Supplier do
      insert(x.sno, x)

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



Example

"Normal" hash-join

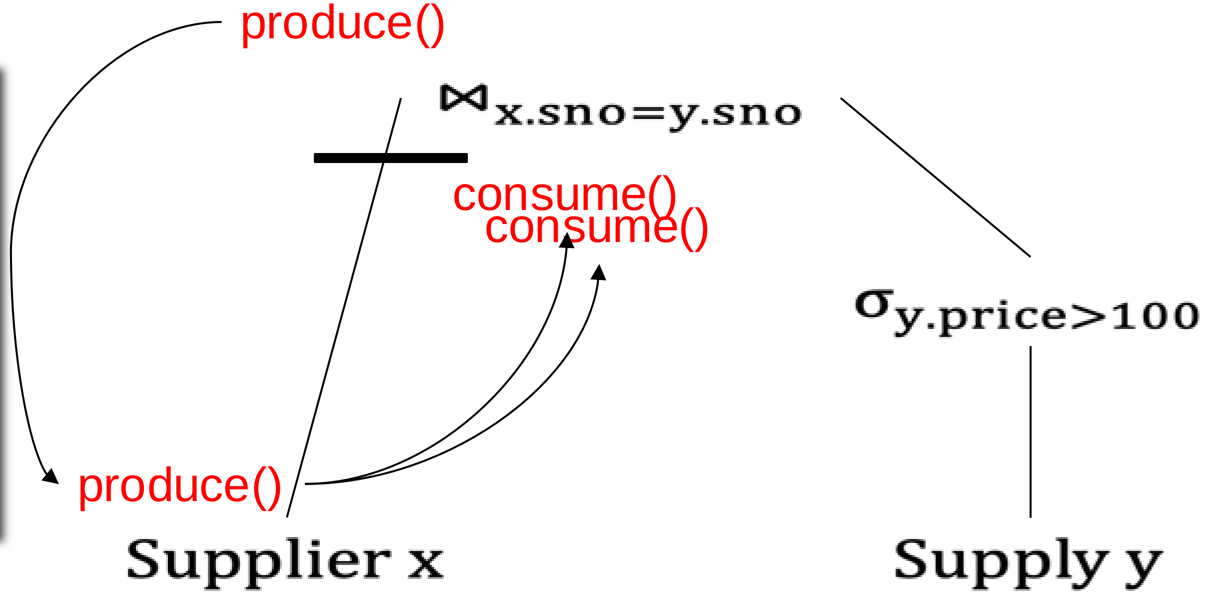
```
for x in Supplier do
     insert(x.sno, x)

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



Example

"Normal" hash-join

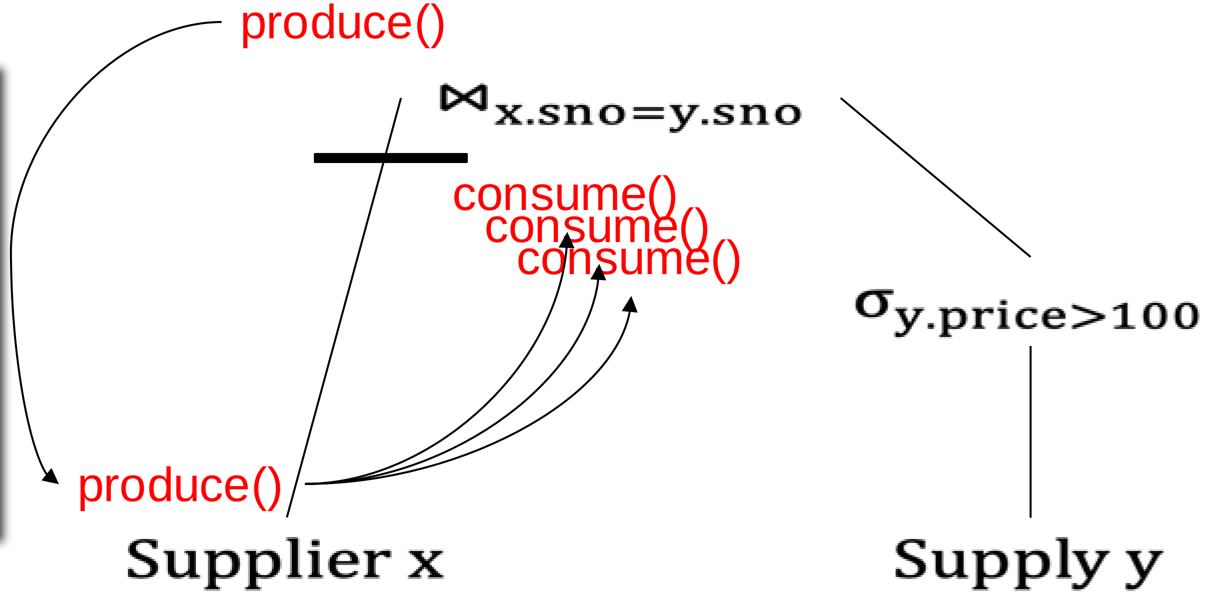
```
for x in Supplier do
      insert(x.sno, x)

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



Example

"Normal" hash-join

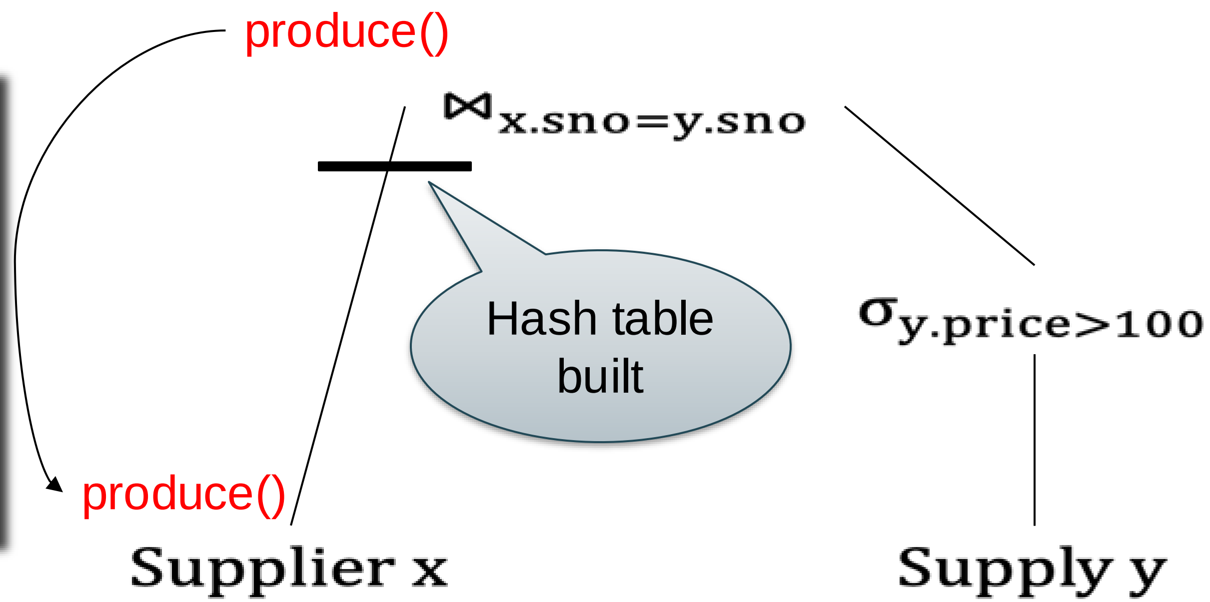
```
for x in Supplier do
      insert(x.sno, x)

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



Example

"Normal" hash-join

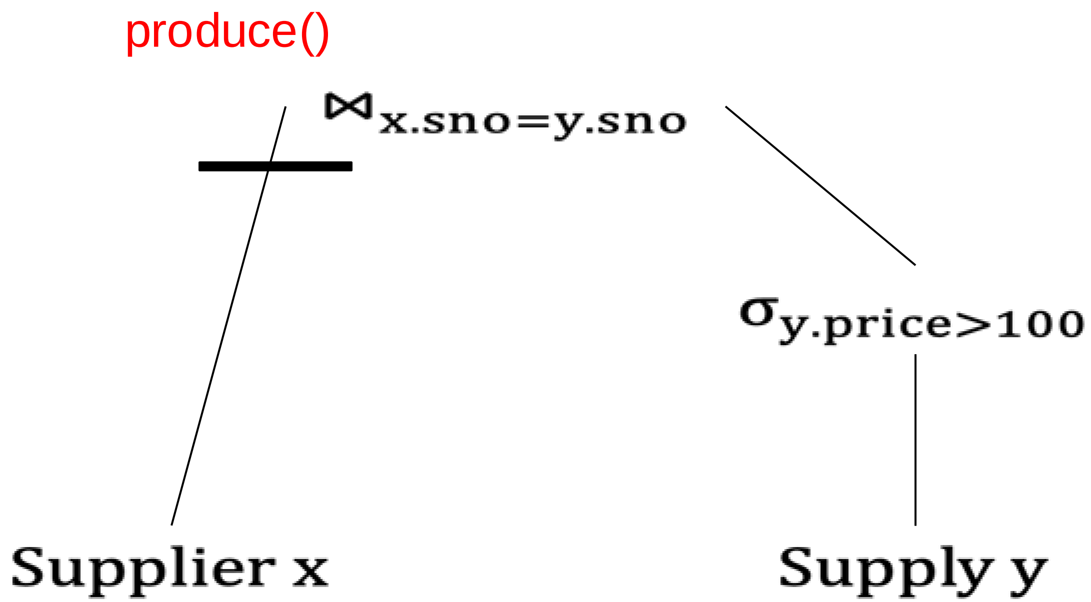
```
for x in Supplier do
      insert(x.sno, x)

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



Example

”Normal” hash-join

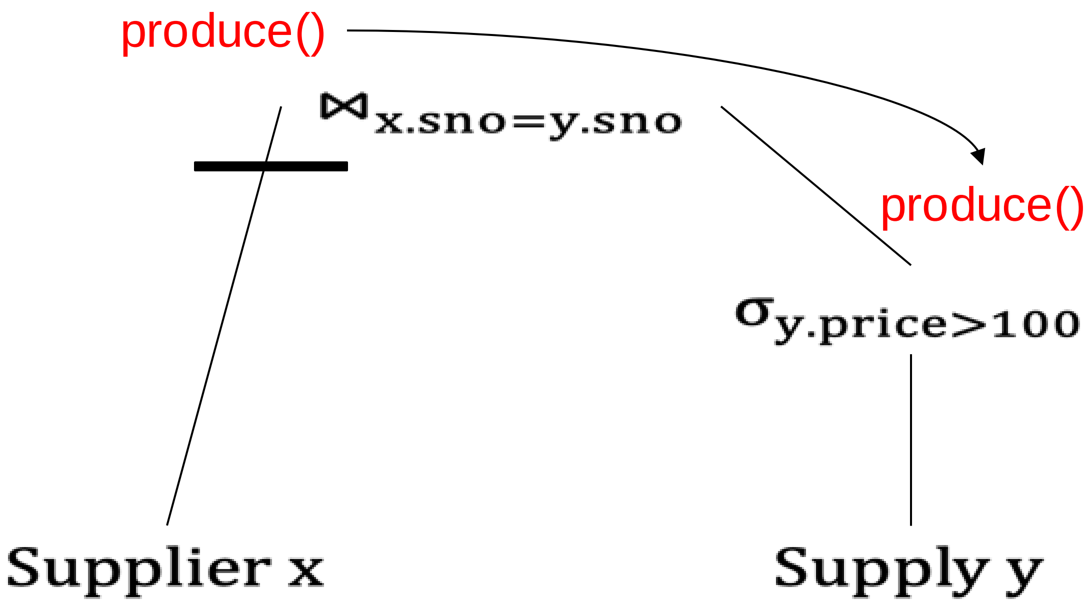
```
for x in Supplier do
      insert(x.sno, x)

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



Example

"Normal" hash-join

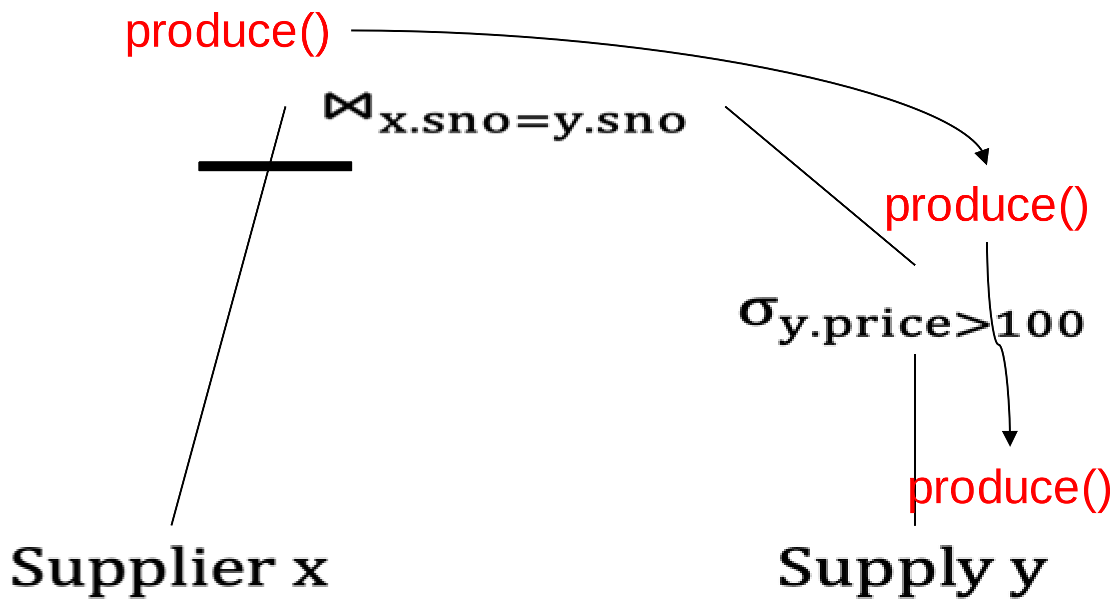
```
for x in Supplier do
      insert(x.sno, x)

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



Example

"Normal" hash-join

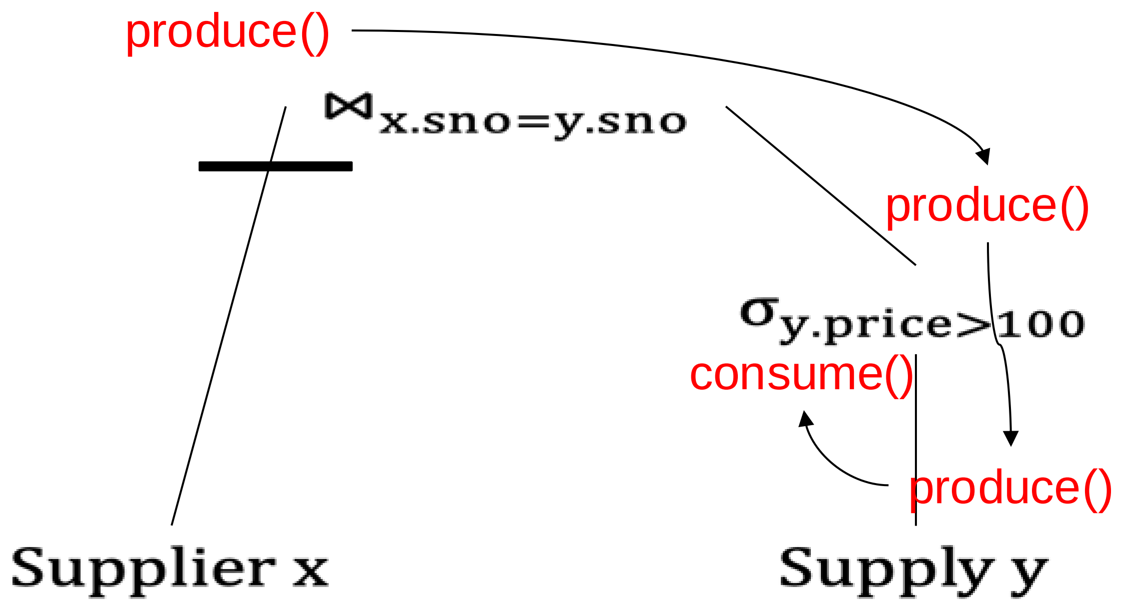
```
for x in Supplier do
      insert(x.sno, x)

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



Example

”Normal” hash-join

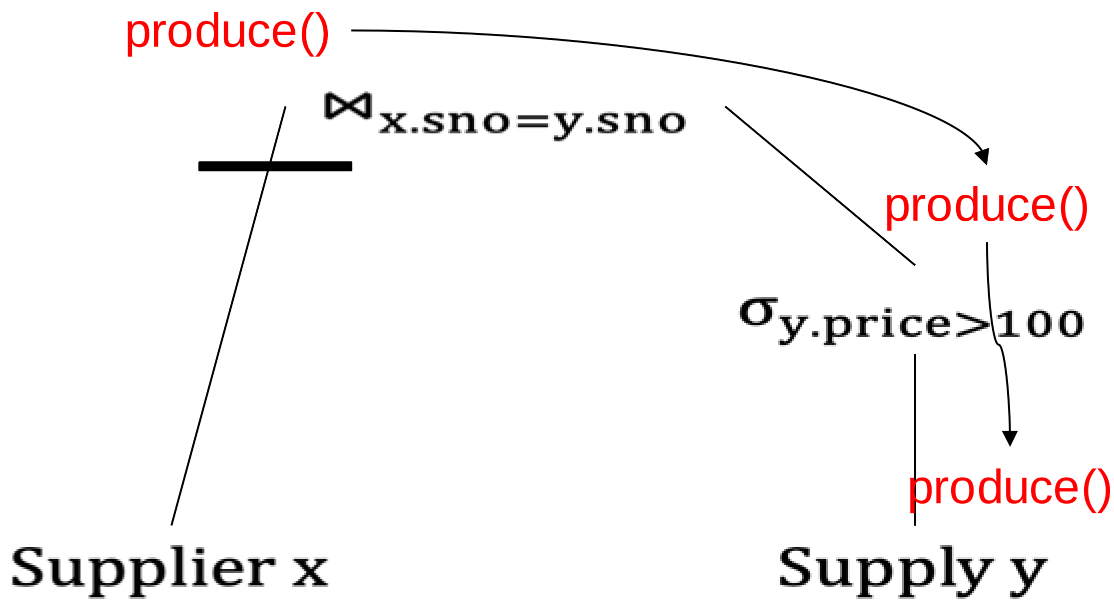
```
for x in Supplier do
     insert(x.sno, x)

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



Example

”Normal” hash-join

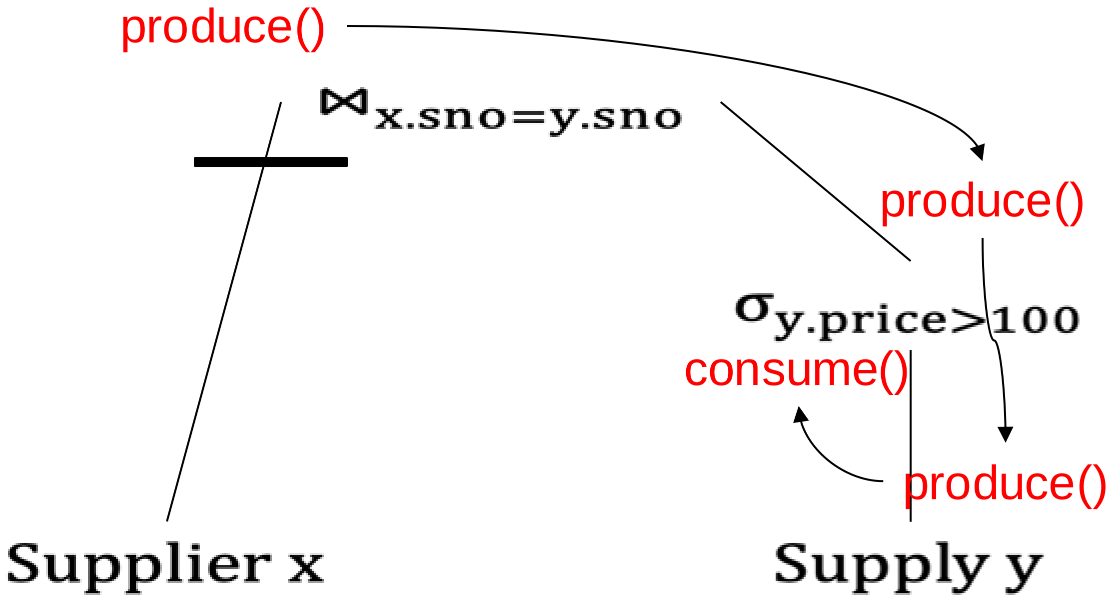
```
for x in Supplier do
      insert(x.sno, x)

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



Example

"Normal" hash-join

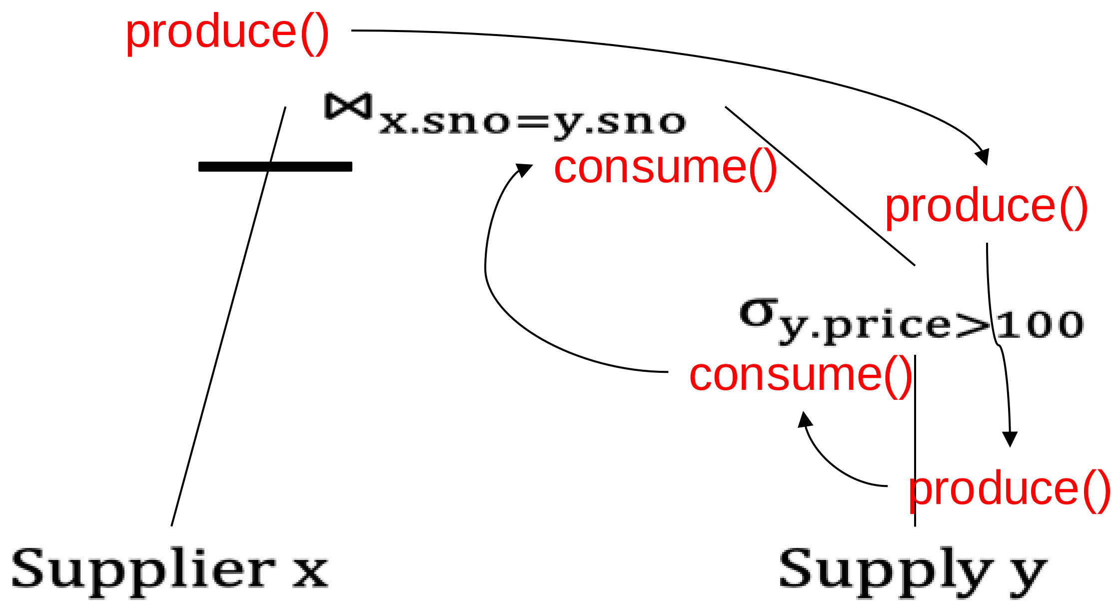
```
for x in Supplier do
      insert(x.sno, x)

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



Example

”Normal” hash-join

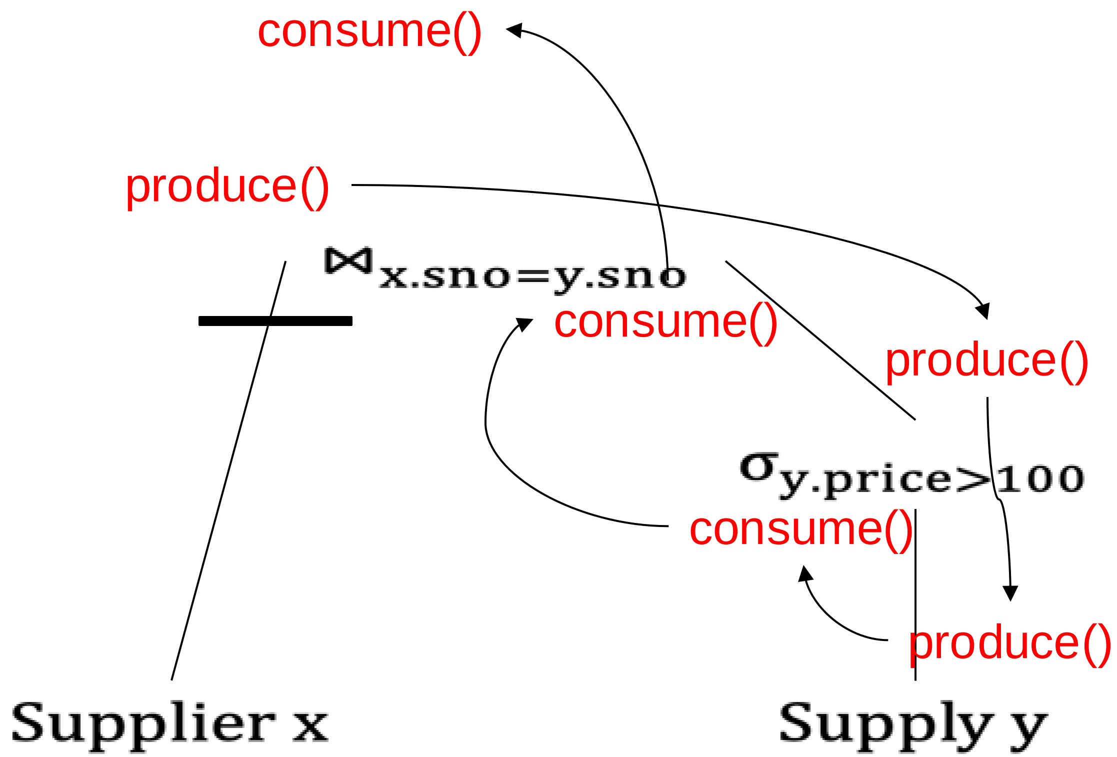
```
for x in Supplier do
     insert(x.sno, x)

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



Example

"Normal" hash-join

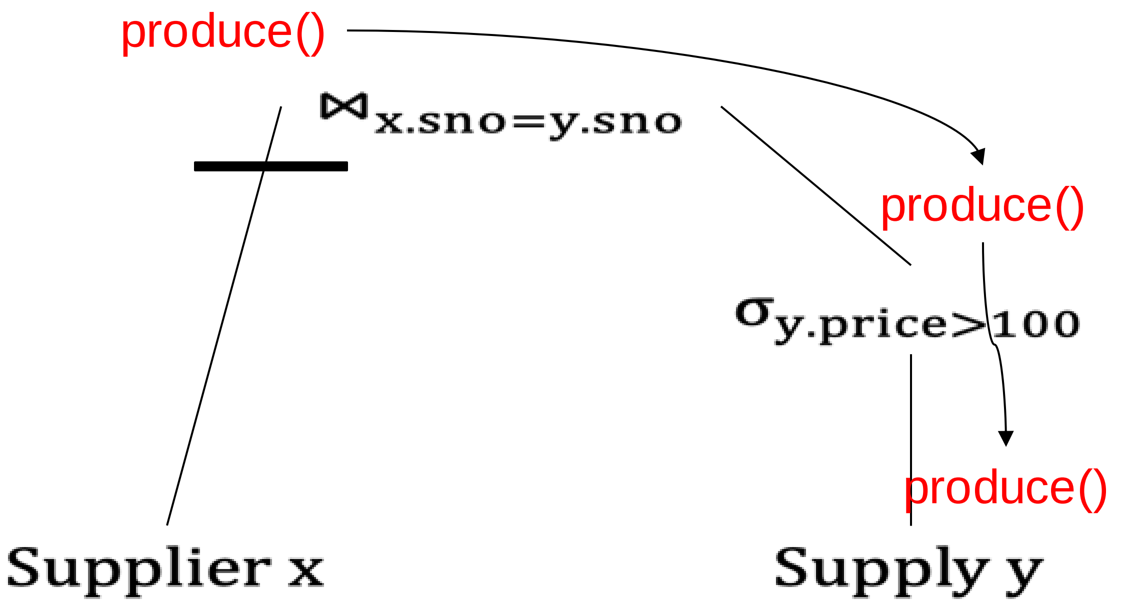
```
for x in Supplier do
      insert(x.sno, x)

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



Example

"Normal" hash-join

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

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

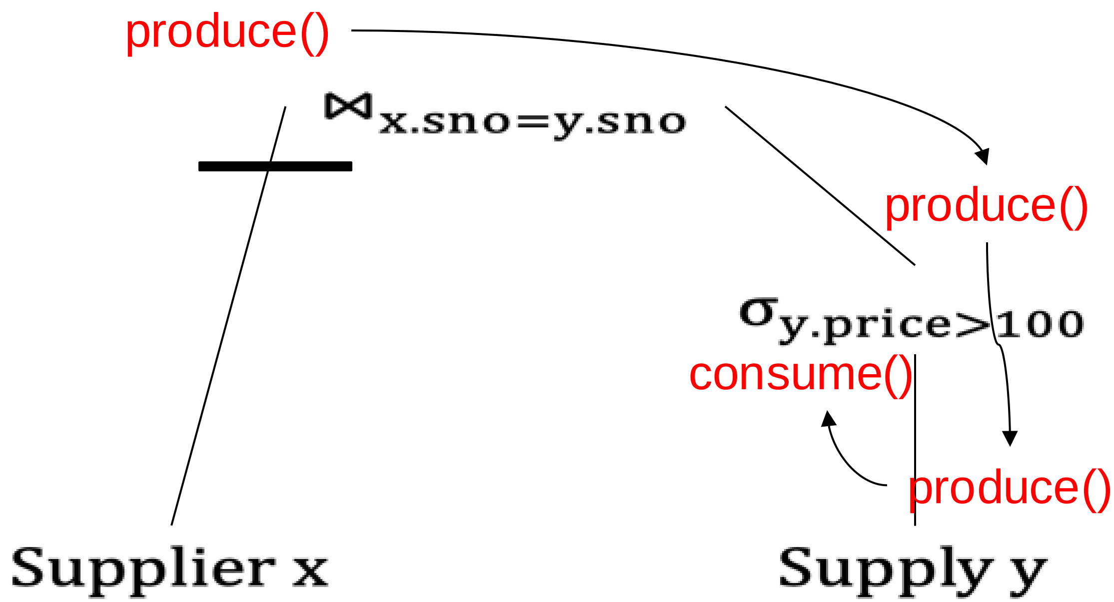


Example

And so on…

"Normal" hash-join

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

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



Volcano v.s. Data Driven

$\sigma_{B=7}$

$\sigma_{A=5}$

R

Volcano v.s. Data Driven

repeat /* $\sigma_B.next()$ */

output x
until x.isNull()

$\sigma_{B=7}$

$\sigma_{A=5}$

R

Volcano v.s. Data Driven

```
repeat /* σ_B.next() */
  repeat /* σ_A.next() *//

  until x.isNull() or x.B=7
  output x
until x.isNull()
```

$\sigma_{B=7}$

$\sigma_{A=5}$

R

Volcano v.s. Data Driven

repeat /* $\sigma_B.next()$ */
 repeat /* $\sigma_A.next()$ *//
  repeat /* $R.next()$ *//

  until x.isNull() or x.A=5
 until x.isNull() or x.B=7
 output x
until x.isNull()

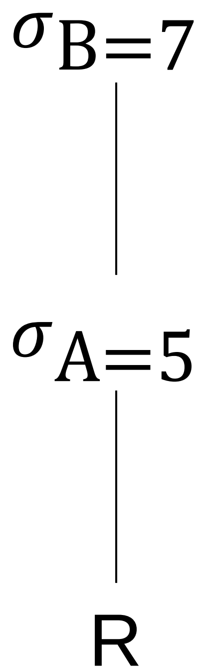
$\sigma_{B=7}$

$\sigma_{A=5}$

R

Volcano v.s. Data Driven

```
repeat /* σB.next() */
 repeat /* σA.next() *//
  repeat /* R.next() *//
      x=R.next()
  until x.isNull() or x.A=5
 until x.isNull() or x.B=7
 output x
until x.isNull()
```



Volcano v.s. Data Driven

```
repeat /* σ_B.next() */
 repeat /* σ_A.next() *//
  repeat /* R.next() *//
      x=R.next()
  until x.isNull() or x.A=5
 until x.isNull() or x.B=7
 output x
until x.isNull()
```

$\sigma_{B=7}$

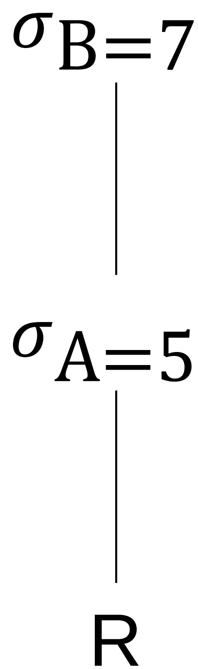
$\sigma_{A=5}$

R

```
for x in R    // R.produce()
```

Volcano v.s. Data Driven

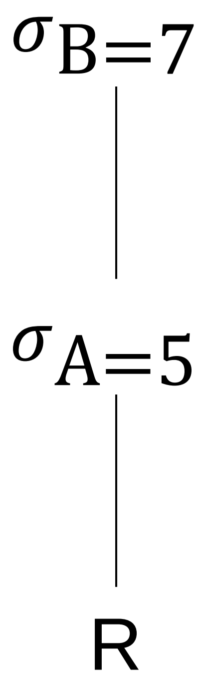
```
repeat /* σ_B.next() */
 repeat /* σ_A.next() *//
  repeat /* R.next() *//
     x=R.next()
  until x.isNull() or x.A=5
 until x.isNull() or x.B=7
 output x
until x.isNull()
```




```
for x in R      // R.produce()
  if x.A=5      // σ_A.consume()
```

Volcano v.s. Data Driven

repeat /* $\sigma_B.next()$ */
 repeat /* $\sigma_A.next()$ *//
  repeat /* $R.next()$ *//
   x=R.next()
  until x.isNull() or x.A=5
 until x.isNull() or x.B=7
 output x
until x.isNull()



for x in R // $R.produce()$
 if x.A=5 // $\sigma_A.consume()$
  if x.B=7 // $\sigma_B.consume()$

Volcano v.s. Data Driven

repeat /* $\sigma_B.next()$ */
  repeat /* $\sigma_A.next()$ *//
    repeat /* $R.next()$ *//
      x=R.next()
    until x.isNull() or x.A=5
  until x.isNull() or x.B=7
  output x
until x.isNull()

$\sigma_{B=7}$

$\sigma_{A=5}$

R

for x in R // $R.produce()$
  if x.A=5 // $\sigma_A.consume()$
    if x.B=7 // $\sigma_B.consume()$
      output x

Volcano v.s. Data Driven

```
repeat /* σ_B.next() */
  repeat /* σ_A.next() *//
    repeat /* R.next() *//
        x=R.next()
    until x.isNull() or x.A=5
  until x.isNull() or x.B=7
  output x
until x.isNull()
```

$\sigma_{B=7}$

$\sigma_{A=5}$

R

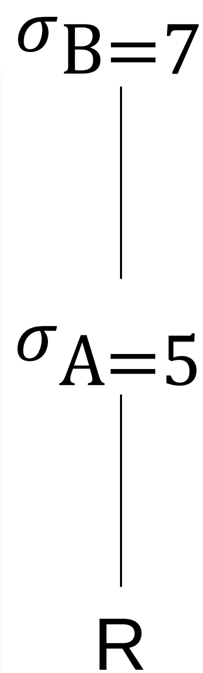
```
for x in R          // R.produce()
  if x.A=5          // σ_A.consume()
    if x.B=7        // σ_B.consume()
      output x
```

Three iterations.

isNull tested repeatedly

Volcano v.s. Data Driven

repeat /* $\sigma_B.next()$ */
 repeat /* $\sigma_A.next()$ *//
  repeat /* $R.next()$ *//
    x=R.next()
  until x.isNull() or x.A=5
 until x.isNull() or x.B=7
 output x
until x.isNull()



for x in R   // $R.produce()$
 if x.A=5   // $\sigma_A.consume()$
  if x.B=7   // $\sigma_B.consume()$
   output x

Three iterations.
isNull tested repeatedly

Simpler code. Better instruction cache locality

Call-back

- For any non-commutative operator like hash-join, consume() must treat differently calls from left and right child

- Paper's solution (next)

```
class HashJoin(left: Op, right: Op)
  (lkey: KeyFun)(rkey: KeyFun) extends Op {
  val hm = new HashMultiMap()
  var isLeft = true
  var parent  = null
  def open() = {                    // Step 1
    left.parent = this; right.parent = this
    left.open; right.open
  }
  def produce() = {
    isLeft = true;  left.produce() // Step 2
    isLeft = false; right.produce()// Step 4
  }
  def consume(rec: Record) = {
    if (isLeft)          // Step 3
      hm += (lkey(rec), rec)
    else                 // Step 5
      for (lr <- hm(rkey(rec)))
        parent.consume(merge(lr,rec))
  }
}
```

(a) (b)

Figure 5: Hash join implementation in (a) Data-centric (b) Data-centric with callbacks model (LB2)

```
class HashJoin(left: Op, right: Op)
  (lkey: KeyFun)(rkey: KeyFun) extends Op {
  val hm = new HashMultiMap()
  var isLeft = true
  var parent  = null
  def open() = {                   // Step 1
    left.parent = this; right.parent = this
    left.open; right.open
  }
  def produce() = {
    isLeft = true;  left.produce() // Step 2
    isLeft = false; right.produce()// Step 4
  }
  def consume(rec: Record) = {
    if (isLeft)          // Step 3
      hm += (lkey(rec), rec)
    else                 // Step 5
      for (lr <- hm(rkey(rec))
        parent.consume(merge(lr,rec))
  }
}
```

(a)

(b)

Figure 5: Hash join implementation in (a) Data-centric (b) Data-centric with callbacks model (LB2)

```
class HashJoin(left: Op, right: Op)
  (lkey: KeyFun)(rkey: KeyFun) extends Op {
  val hm = new HashMultiMap()
  var isLeft = true
  var parent  = null
  def open() = {                      // Step 1
    left.parent = this; right.parent = this
    left.open; right.open
  }
  def produce() = {
    isLeft = true;  left.produce() // Step 2
    isLeft = false; right.produce()// Step 4
  }
  def consume(rec: Record) = {
    if (isLeft)          // Step 3
      hm += (lkey(rec), rec)
    else                 // Step 5
      for (lr <- hm(rkey(rec))
        parent.consume(merge(lr,rec))
  }
}
```

(a) (b)

Figure 5: Hash join implementation in (a) Data-centric (b) Data-centric with callbacks model (LB2)

```
class HashJoin(left: Op, right: Op)
  (lkey: KeyFun)(rkey: KeyFun) extends Op {
  val hm = new HashMultiMap()
  var isLeft = true
  var parent  = null
  def open() = {                    // Step 1
    left.parent = this; right.parent = this
    left.open; right.open
  }
  def produce() = {
    isLeft = true;  left.produce() // Step 2
    isLeft = false; right.produce()// Step 4
  }
  def consume(rec: Record) = {
    if (isLeft)           // Step 3
      hm += (lkey(rec), rec)
    else                  // Step 5
      for (lr <- hm(rkey(rec))
        parent.consume(merge(lr,rec))
  }
}
```

(a) (b)

Figure 5: Hash join implementation in (a) Data-centric (b) Data-centric with callbacks model (LB2)

```
class HashJoin(left: Op, right: Op)
  (lkey: KeyFun)(rkey: KeyFun) extends Op {
  val hm = new HashMultiMap()
  var isLeft = true
  var parent  = null
  def open() = {                   // Step 1
    left.parent = this; right.parent = this
    left.open; right.open
  }
  def produce() = {
    isLeft = true;  left.produce() // Step 2
    isLeft = false; right.produce()// Step 4
  }
  def consume(rec: Record) = {
    if (isLeft)           // Step 3
      hm += (lkey(rec), rec)
    else                  // Step 5
      for (lr <- hm(rkey(rec)))
        parent.consume(merge(lr,rec))
  }
}
```

(a) (b)

Figure 5: Hash join implementation in (a) Data-centric (b) Data-centric with callbacks model (LB2)

```
class HashJoin(left: Op, right: Op)
  (lkey: KeyFun)(rkey: KeyFun) extends Op {
  val hm = new HashMultiMap()
  var isLeft = true
  var parent  = null
  def open() = {                   // Step 1
    left.parent = this; right.parent = this
    left.open; right.open
  }
  def produce() = {
    isLeft = true;  left.produce() // Step 2
    isLeft = false; right.produce()// Step 4
  }
  def consume(rec: Record) = {
    if (isLeft)          // Step 3
      hm += (lkey(rec), rec)
    else                 // Step 5
      for (lr <- hm(rkey(rec))
        parent.consume(merge(lr,rec))
  }
}
```

(a)

```
class HashJoin(left: Op, right: Op)
  (lkey: KeyFun)(rkey: KeyFun) extends Op {

  // refactored open, produce, consume
  // into single method exec
  def exec(cb: Record => Unit) = {
    val hm = new HashMultiMap()
    left.exec { rec =>  // Step 1
      hm += (lkey(rec), rec)
    }
    right.exec { rec => // Step 2
      for (lr <- hm(rkey(rec))
        cb(merge(lr,rec))
    }
  }
}
```

(b)

Figure 5: Hash join implementation in (a) Data-centric (b) Data-centric with callbacks model (LB2)

```
class HashJoin(left: Op, right: Op)
  (lkey: KeyFun)(rkey: KeyFun) extends Op {
  val hm = new HashMultiMap()
  var isLeft = true
  var parent  = null
  def open() = {                  // Step 1
    left.parent = this; right.parent = this
    left.open; right.open
  }
  def produce() = {
    isLeft = true;  left.produce() // Step 2
    isLeft = false; right.produce()// Step 4
  }
  def consume(rec: Record) = {
    if (isLeft)          // Step 3
      hm += (lkey(rec), rec)
    else                 // Step 5
      for (lr <- hm(rkey(rec))
        parent.consume(merge(lr,rec))
  }
}
```

(a)

```
class HashJoin(left: Op, right: Op)
  (lkey: KeyFun)(rkey: KeyFun) extends Op {

  // refactored open, produce, consume
  // into single method exec
  def exec(cb: Record => Unit) = {
    val hm = new HashMultiMap()
    left.exec { rec =>  // Step 1
      hm += (lkey(rec), rec)
    }
    right.exec { rec => // Step 2
      for (lr <- hm(rkey(rec))
        cb(merge(lr,rec))
    }
  }
}
```

(b)

Figure 5: Hash join implementation in (a) Data-centric (b) Data-centric with callbacks model (LB2)

```
class HashJoin(left: Op, right: Op)
  (lkey: KeyFun)(rkey: KeyFun) extends Op {
  val hm = new HashMultiMap()
  var isLeft = true
  var parent  = null
  def open() = {                    // Step 1
    left.parent = this; right.parent = this
    left.open; right.open
  }
  def produce() = {
    isLeft = true;  left.produce() // Step 2
    isLeft = false; right.produce()// Step 4
  }
  def consume(rec: Record) = {
    if (isLeft)           // Step 3
      hm += (lkey(rec), rec)
    else                  // Step 5
      for (lr <- hm(rkey(rec))
        parent.consume(merge(lr,rec))
  }
}
```

(a)

```
class HashJoin(left: Op, right: Op)
  (lkey: KeyFun)(rkey: KeyFun) extends Op {

  // refactored open, produce, consume
  // into single method exec
  def exec(cb: Record => Unit) = {
    val hm = new HashMultiMap()
    left.exec { rec =>  // Step 1
      hm += (lkey(rec), rec)
    }
    right.exec { rec => // Step 2
      for (lr <- hm(rkey(rec))
        cb(merge(lr,rec))
    }
  }
}
```

(b)

Figure 5: Hash join implementation in (a) Data-centric (b) Data-centric with callbacks model (LB2)

```scala
class HashJoin(left: Op, right: Op)
  (lkey: KeyFun)(rkey: KeyFun) extends Op {
  val hm = new HashMultiMap()
  var isLeft = true
  var parent  = null
  def open() = {                    // Step 1
    left.parent = this; right.parent = this
    left.open; right.open
  }
  def produce() = {
    isLeft = true;  left.produce() // Step 2
    isLeft = false; right.produce()// Step 4
  }
  def consume(rec: Record) = {
    if (isLeft)           // Step 3
      hm += (lkey(rec), rec)
    else                  // Step 5
      for (lr <- hm(rkey(rec)))
        parent.consume(merge(lr,rec))
  }
}
```

(a)

```scala
class HashJoin(left: Op, right: Op)
  (lkey: KeyFun)(rkey: KeyFun) extends Op {

  // refactored open, produce, consume
  // into single method exec
  def exec(cb: Record => Unit) = {
    val hm = new HashMultiMap()
    left.exec { rec =>  // Step 1
      hm += (lkey(rec), rec)
    }
    right.exec { rec => // Step 2
      for (lr <- hm(rkey(rec)))
        cb(merge(lr,rec))
    }
  }
}
```

(b)

Figure 5: Hash join implementation in (a) Data-centric (b) Data-centric with callbacks model (LB2)

```
class HashJoin(left: Op, right: Op)
  (lkey: KeyFun)(rkey: KeyFun) extends Op {
  val hm = new HashMultiMap()
  var isLeft = true
  var parent  = null
  def open() = {                    // Step 1
    left.parent = this; right.parent = this
    left.open; right.open
  }
  def produce() = {
    isLeft = true;  left.produce() // Step 2
    isLeft = false; right.produce()// Step 4
  }
  def consume(rec: Record) = {
    if (isLeft)            // Step 3
      hm += (lkey(rec), rec)
    else                   // Step 5
      for (lr <- hm(rkey(rec))
        parent.consume(merge(lr,rec))
  }
}
```

(a)

```
class HashJoin(left: Op, right: Op)
  (lkey: KeyFun)(rkey: KeyFun) extends Op {

  // refactored open, produce, consume
  // into single method exec
  def exec(cb: Record => Unit) = {
    val hm = new HashMultiMap()
    left.exec { rec =>  // Step 1
      hm += (lkey(rec), rec)
    }
    right.exec { rec => // Step 2
      for (lr <- hm(rkey(rec))
        cb(merge(lr,rec))
    }
  }
}
```

(b)

Figure 5: Hash join implementation in (a) Data-centric (b) Data-centric with callbacks model (LB2)

Final Discussion

80s and 90s: cost dominated by I/O

- Interpretation simpler than compilation
- Volcano iterator model also simple

Final Discussion

80s and 90s: cost dominated by I/O

- Interpretation simpler than compilation
- Volcano iterator model also simple

Today: data in memory, cost dominated by CPU

- Compilation has significant advantage

Final Discussion

80s and 90s: cost dominated by I/O

- Interpretation simpler than compilation
- Volcano iterator model also simple

Today: data in memory, cost dominated by CPU

- Compilation has significant advantage
- Alternative: vectorized processing
  - Next() returns batch of 1000 tuples
  - Interpreter as good as compiled code