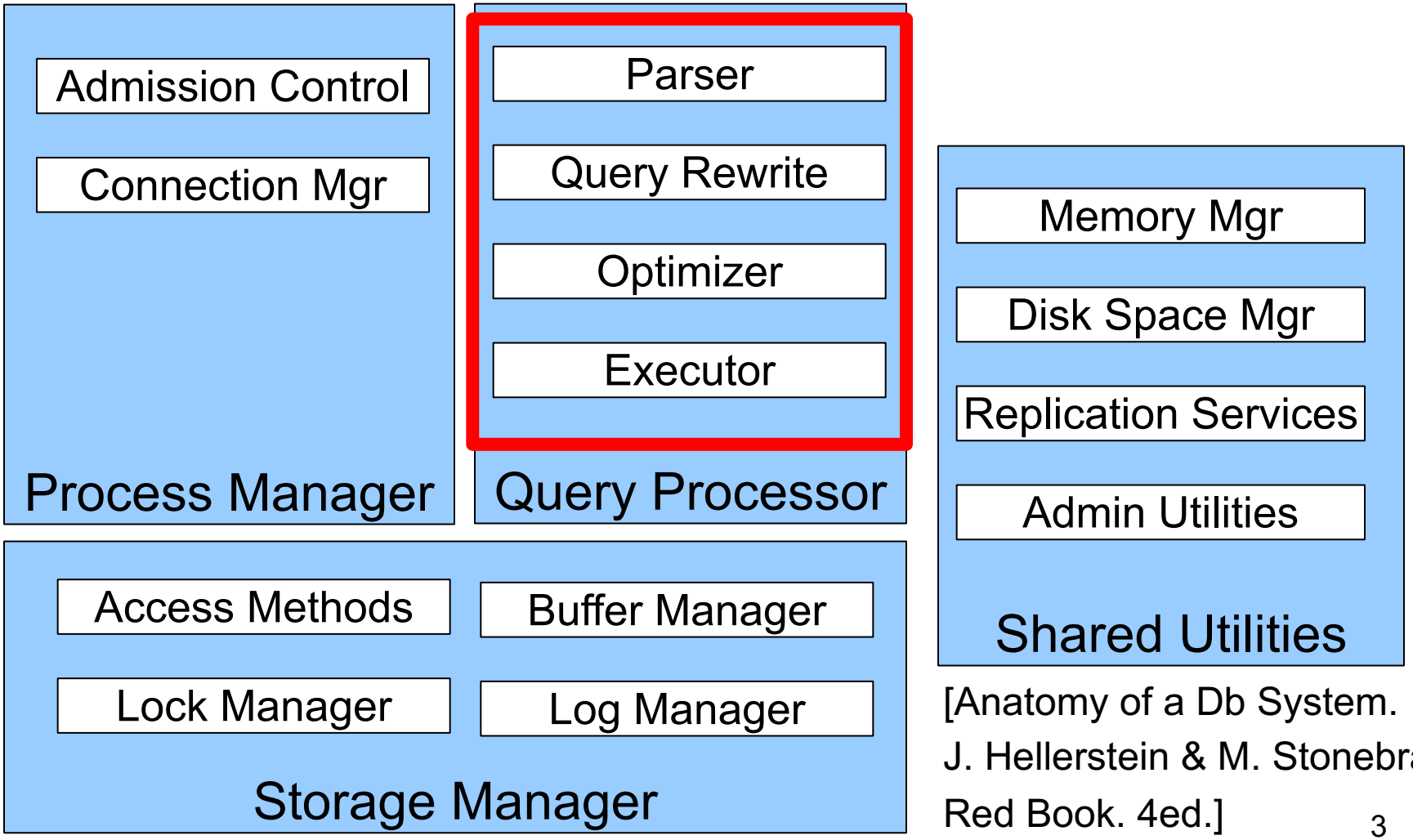
DATA516/CSED516
Scalable Data Systems and Algorithms

Lecture 2 Query Execution and Optimization

Announcements

- Paper reviews are due before the beginning of the class
- HW1 is due next Tuesday
- Please start thinking about the project

DBMS Architecture



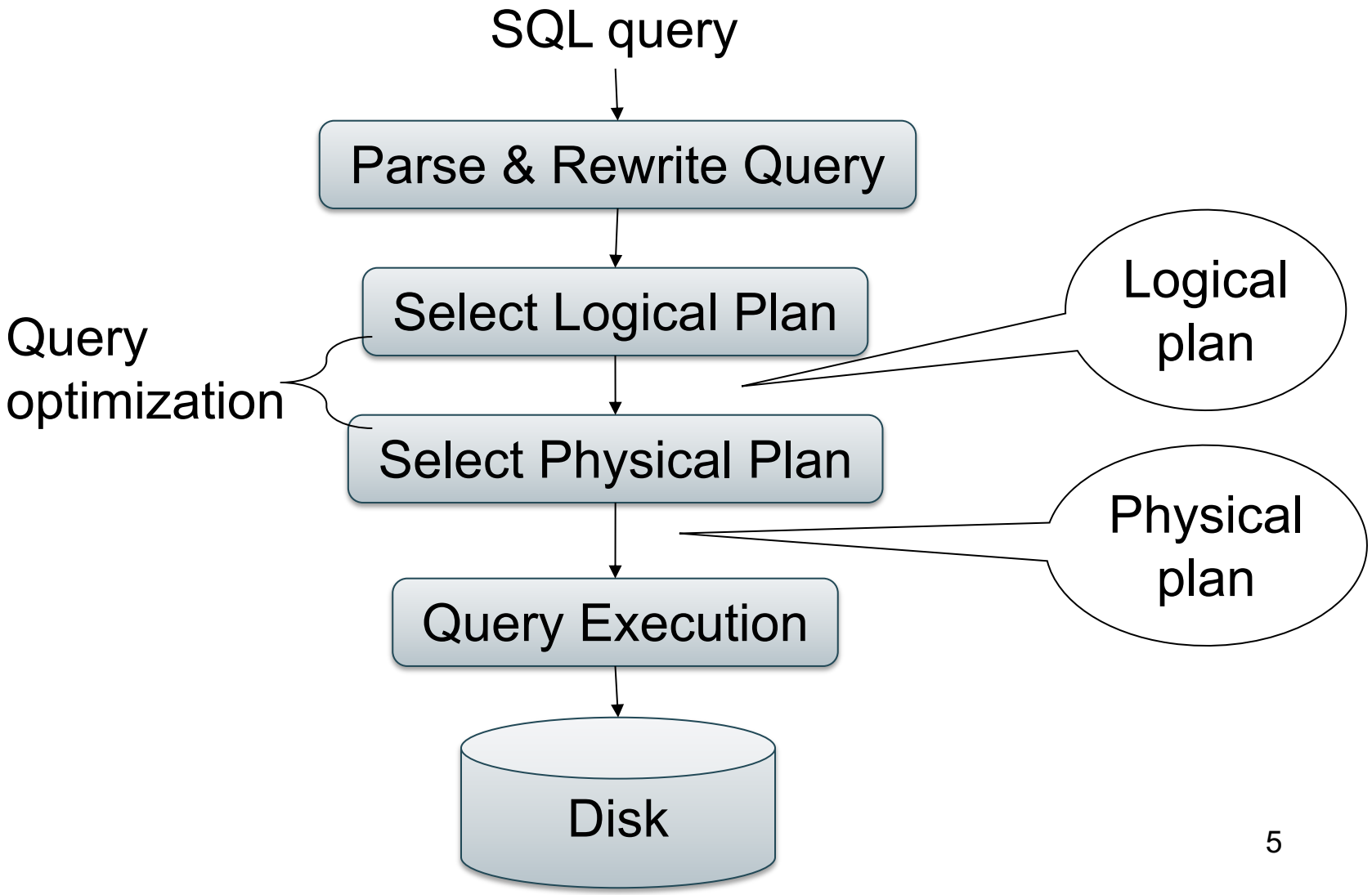
[Anatomy of a Db System. J. Hellerstein & M. Stonebraker. Red Book. 4ed.]

Warning: it will be confusing…

DBMS are monoliths: all components must work together and cannot be isolated

- Good news:
  - Hole system has rich functionality and is efficient
- Bad news:
  - Hard to discuss components in isolation
  - Impossible to use components in isolation

Lifecycle of a Query



Example Database Schema

Supplier(sno,sname,scity,sstate)

Part(pno,pname,psize,pcolor)

Supply(sno,pno,price)

View: Suppliers in Seattle

```
CREATE VIEW NearbySupp AS
SELECT sno, sname
FROM Supplier
WHERE scity='Seattle' AND sstate='WA'
```

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

Example Query

- Find the names of all suppliers in Seattle who supply part number 2

```
SELECT sno, sname FROM NearbySupp
WHERE sno IN (     SELECT sno
                   FROM Supply
                   WHERE pno = 2 )
```

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

View Rewriting, Flattening

Original query:

```
SELECT sno, sname
FROM NearbySupp
WHERE sno IN (SELECT sno
              FROM Supply
              WHERE pno = 2 )
```

View rewriting
= view inlining
= view expansion

Flattening
= unnesting

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

View Rewriting, Flattening

Original query:

```
SELECT sno, sname
FROM NearbySupp
WHERE sno IN (SELECT sno
              FROM Supply
              WHERE pno = 2 )
```

Rewritten query:

```
SELECT DISTINCT S.sno, S.sname
FROM Supplier S, Supply U
WHERE S.scity='Seattle' AND S.sstate='WA'
AND S.sno = U.sno
AND U.pno = 2;
```

View rewriting
= view inlining
= view expansion

Flattening
= unnesting

Reasoning about sets/bags

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

Decorrelation

```
SELECT  Q.sno
FROM Supplier Q
WHERE  Q.sstate = 'WA'
   and not exists
      (SELECT *
       FROM Supply P
       WHERE P.sno = Q.sno
             and P.price > 100)
```

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

Decorrelation

```
SELECT  Q.sno
FROM Supplier Q
WHERE  Q.sstate = 'WA'
   and not exists
      (SELECT *
       FROM Supply P
       WHERE P.sno = Q.sno
             and P.price > 100)
```

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

Decorrelation

```
SELECT  Q.sno
FROM Supplier Q
WHERE  Q.sstate = 'WA'
   and not exists
      (SELECT *
       FROM Supply P
       WHERE P.sno = Q.sno
             and P.price > 100)
```

De-Correlation

```
SELECT  Q.sno
FROM Supplier Q
WHERE  Q.sstate = 'WA'
   and Q.sno not in
      (SELECT P.sno
       FROM Supply P
       WHERE P.price > 100)
```

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

Decorrelation

Un-nesting

```
(SELECT  Q.sno
 FROM Supplier Q
 WHERE  Q.sstate = 'WA')
   EXCEPT
 (SELECT P.sno
  FROM Supply P
  WHERE P.price > 100)
```

EXCEPT = set difference

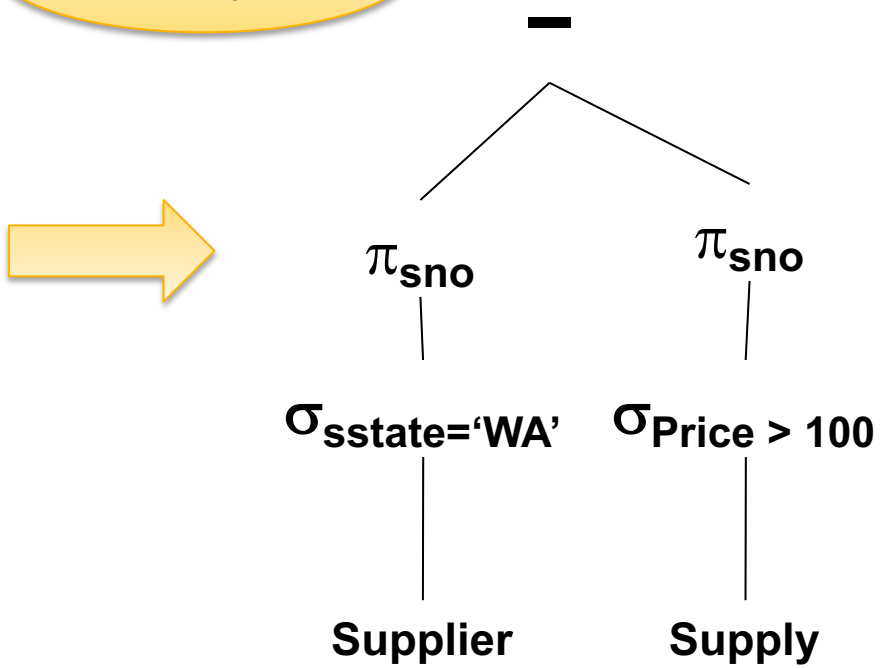
```
SELECT  Q.sno
FROM Supplier Q
WHERE  Q.sstate = 'WA'
   and Q.sno not in
      (SELECT P.sno
       FROM Supply P
       WHERE P.price > 100)
```

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

Decorrelation

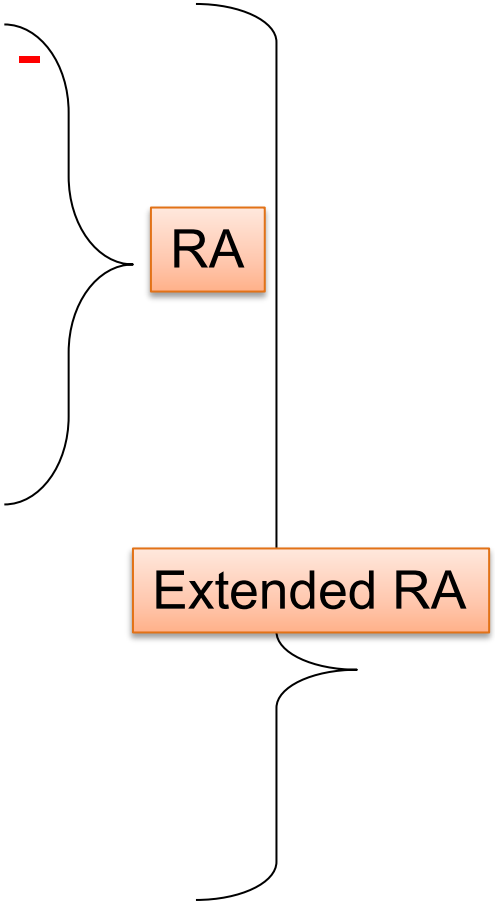
```
(SELECT  Q.sno
 FROM Supplier Q
 WHERE  Q.sstate = 'WA')
   EXCEPT
 (SELECT P.sno
  FROM Supply P
  WHERE P.price > 100)
```

Finally…

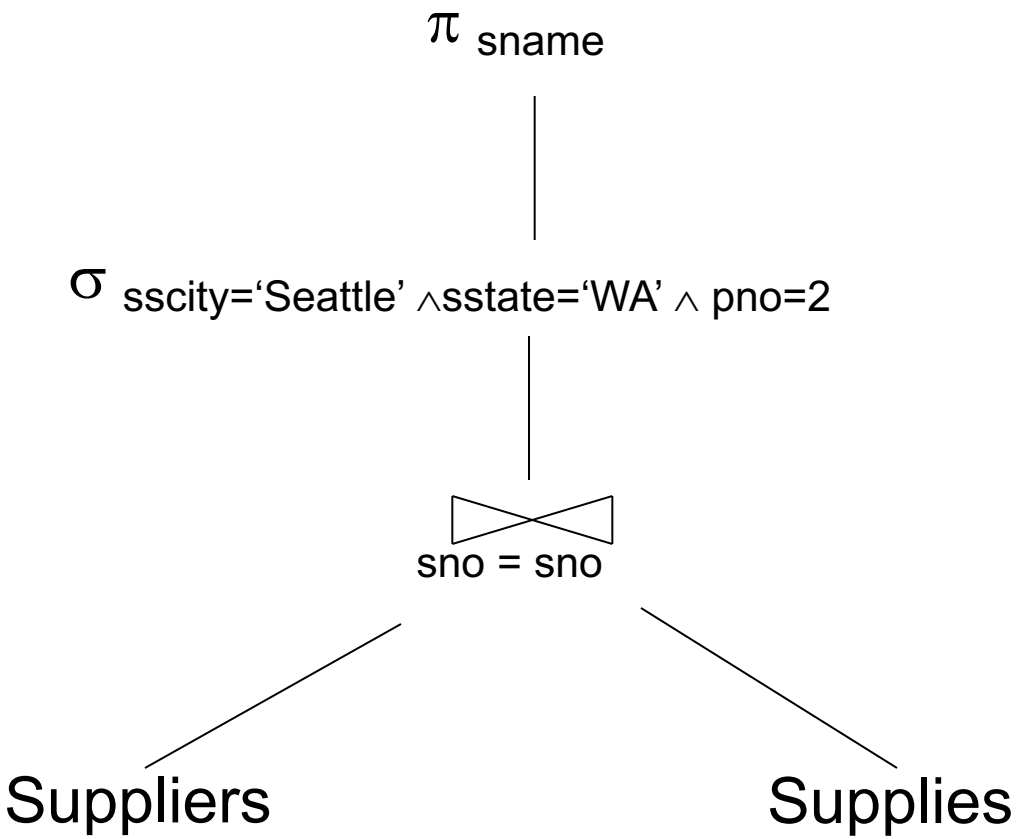


Relational Algebra Operators

- Union ∪, ~~intersection ∩~~, difference -
- Selection σ
- Projection π
- Cartesian product ×, join ⋈
- (Rename ρ)
- Duplicate elimination δ
- Grouping and aggregation γ
- Sorting τ



Logical Query Plan

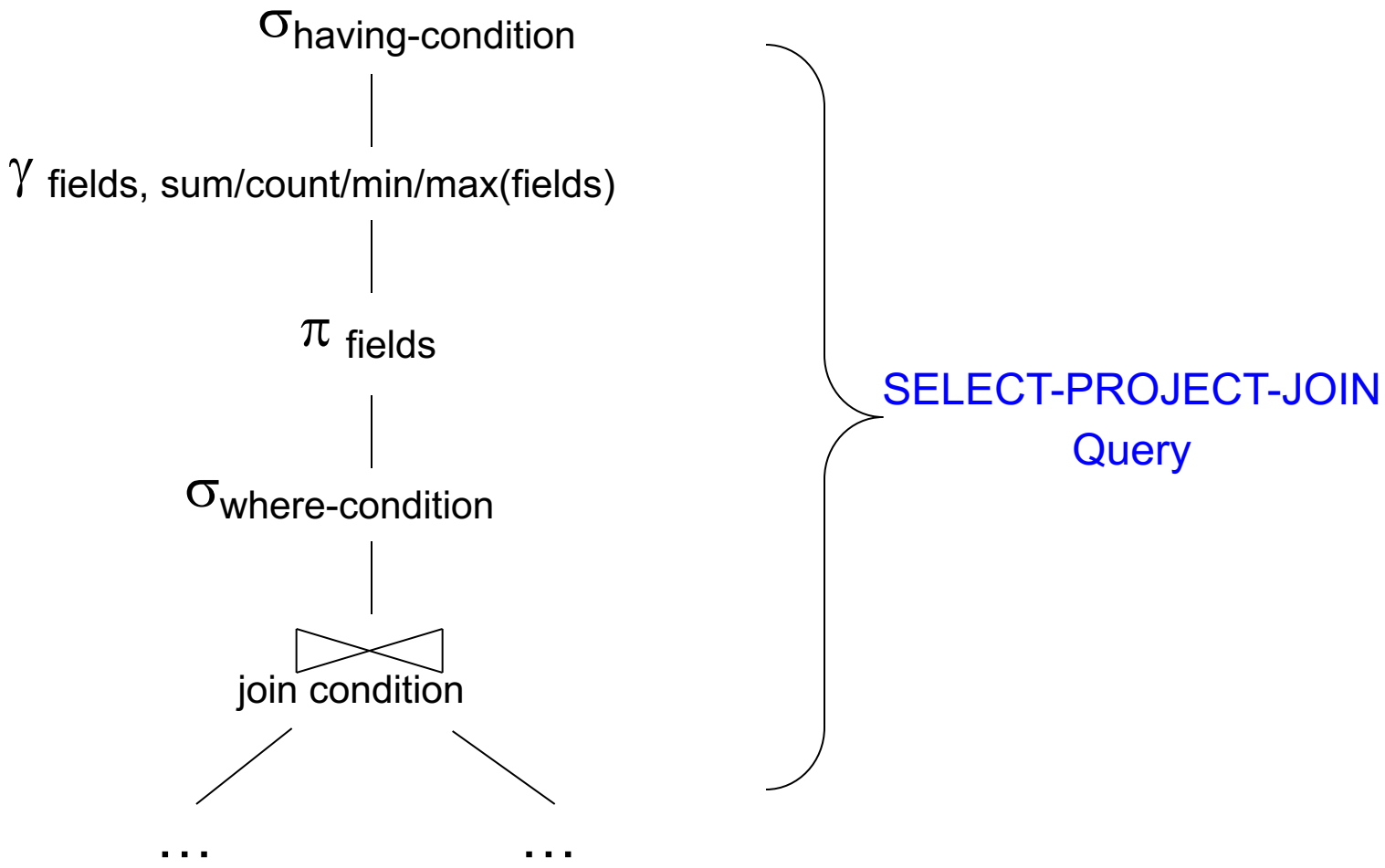


Query Block

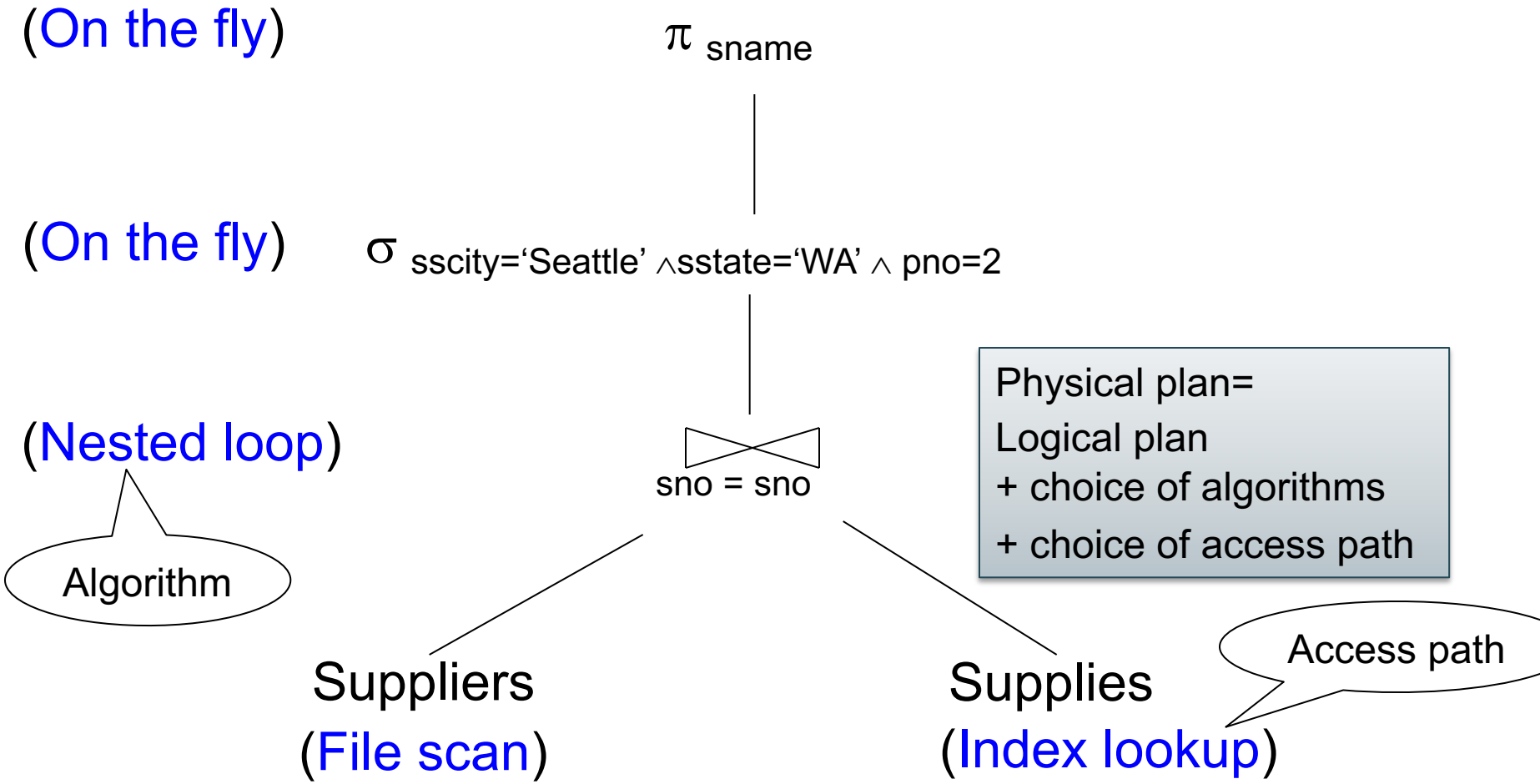
- Most optimizers operate on individual query blocks

- A query block is an SQL query with **no nesting**
  - **Exactly one**
    - SELECT clause
    - FROM clause
  - **At most one**
    - WHERE clause
    - GROUP BY clause
    - HAVING clause

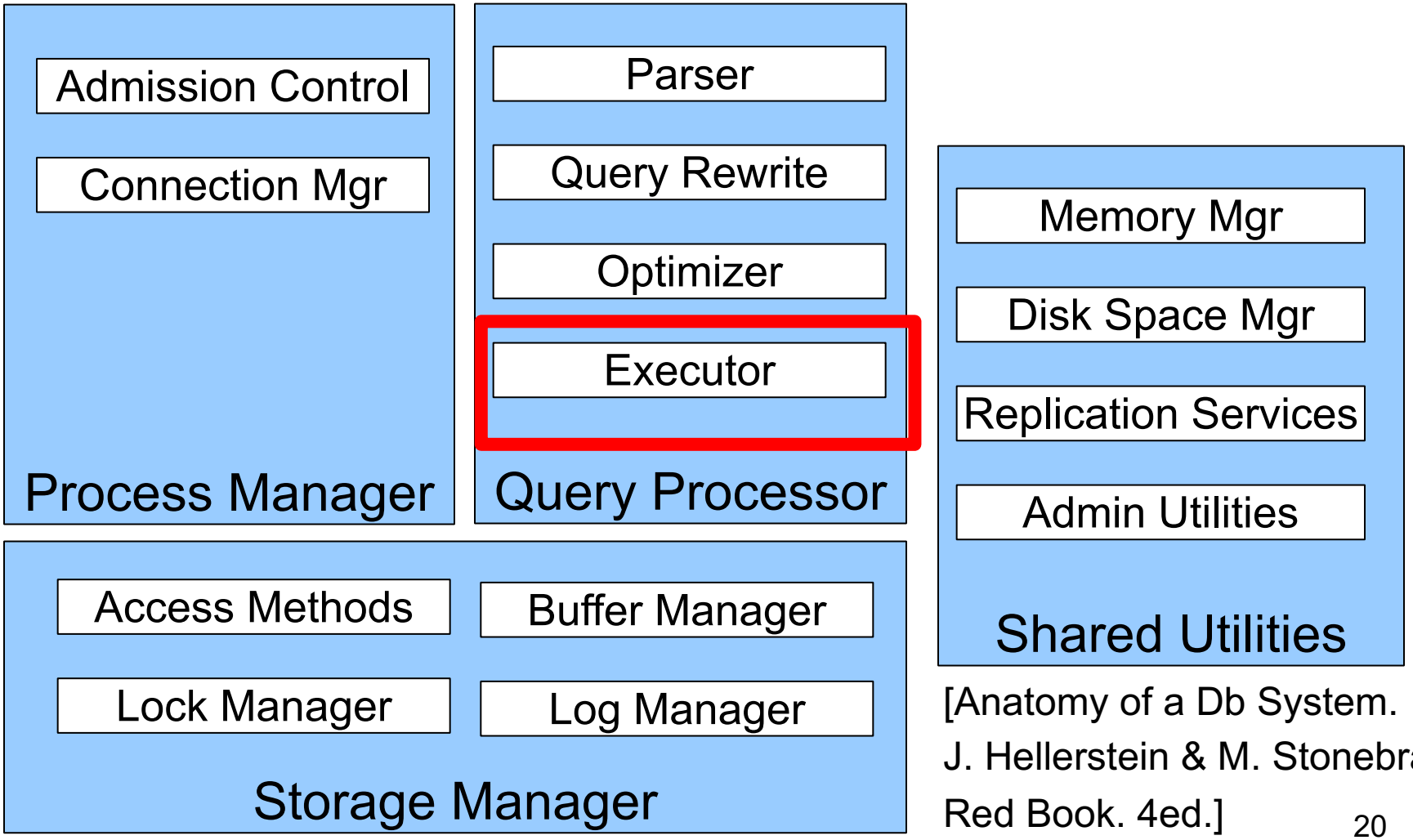
Query Plan For A Block



Physical Query Plan

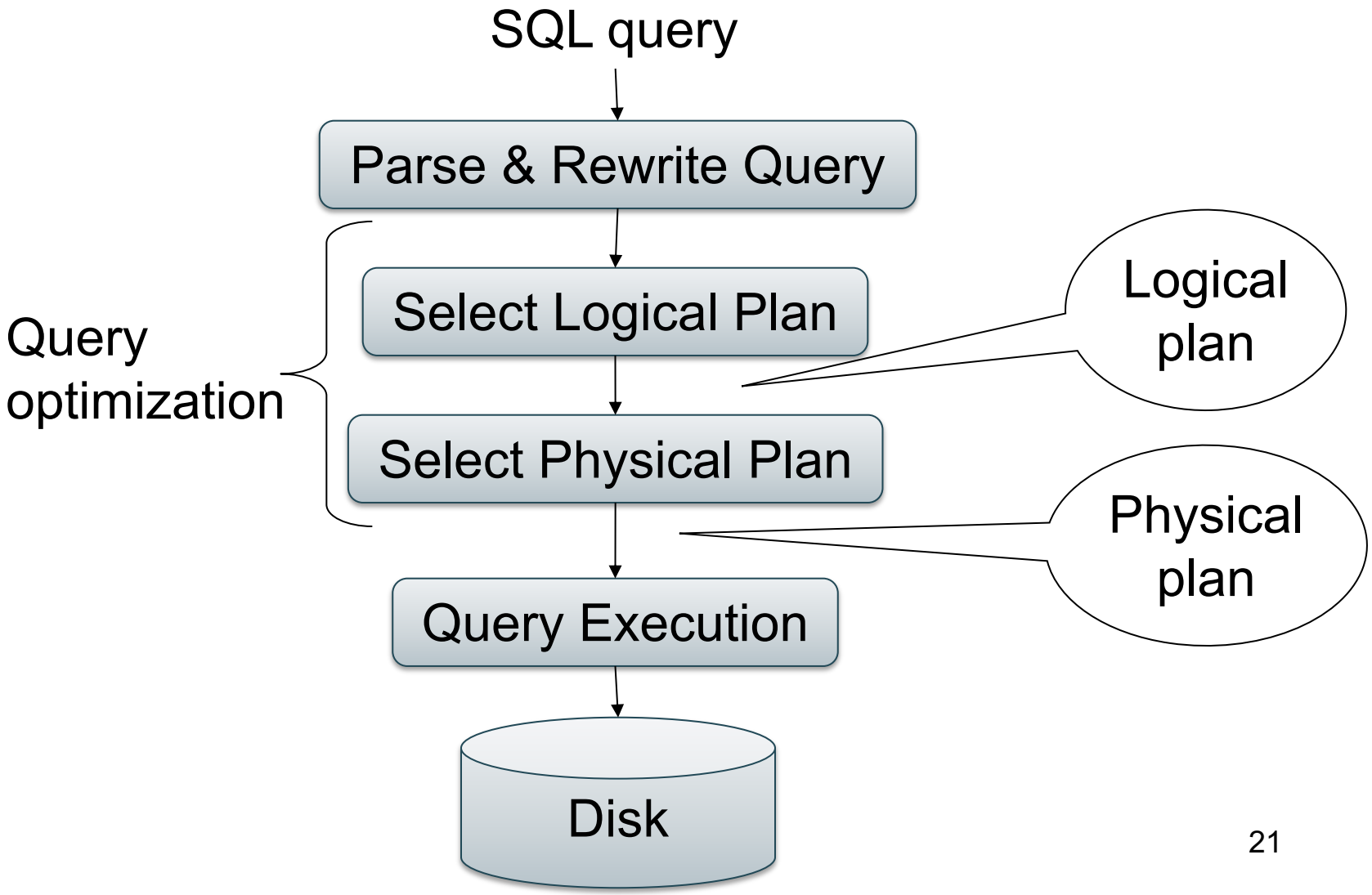


DBMS Architecture



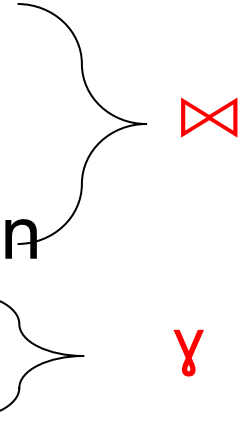
[Anatomy of a Db System. J. Hellerstein & M. Stonebraker. Red Book. 4ed.]

Lifecycle of a Query



Physical Operators

- For each operator, several algorithms
- Main memory or external memory
- Examples:
  - Main memory hash join
  - External memory merge join
  - External memory partitioned hash join
  - Sort-based group by
  - Hash-based group by



Supplier(sid, sname, scity, sstate)

Supply(sid, pno, quantity)

Main Memory Algorithms

Logical operator:

Supplier $\bowtie_{sid=sid}$ Supply

Three algorithms:

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

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

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

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

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

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

$O(n^2)$

BRIEF Review of Hash Tables

A (naïve) hash function:

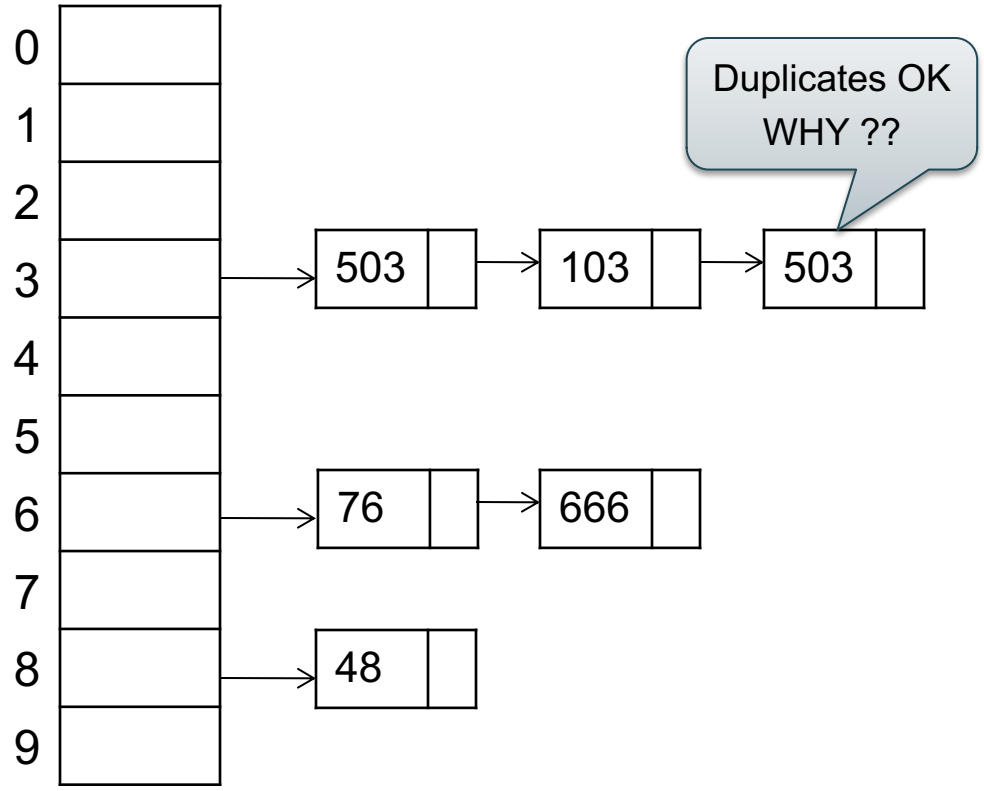
$$h(x) = x \bmod 10$$

Operations:

find(103) = ??

insert(488) = ??

Separate chaining:



BRIEF Review of Hash Tables

- insert(k, v) = inserts a key k with value v
- Many values for one key
  - Hence, duplicate k's are OK
- find(k) = returns the ***list*** of all values v associated to the key k

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

2. Hash Join

Logical operator:

Supplier $\bowtie_{sid=sid}$ Supply

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:

Supplier $\bowtie_{sid=sid}$ Supply

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);
```

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

2. Hash Join

Logical operator:

Supplier $\bowtie_{sid=sid}$ Supply

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);
```

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

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

2. Hash Join

Logical operator:

Supplier $\bowtie_{sid=sid}$ Supply

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);
```

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

O(n)

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

2. Hash Join

Logical operator:

Supplier $\bowtie_{sid=sid}$ Supply

Change join order

```
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

Logical operator:

Supplier $\bowtie_{sid=sid}$ Supply

Change join order

```
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

Logical operator:

Supplier $\bowtie_{sid=sid}$ Supply

Change join order

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

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

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

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

2. Hash Join

Logical operator:

Supplier $\bowtie_{sid=sid}$ Supply

Change join order

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

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

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

O(n)

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

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

2. Hash Join

Why would we change the order?

Logical operator:

Supplier $\bowtie_{sid=sid}$ Supply

Change join order

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

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

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

O(n)

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

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

2. Hash Join

Why would we change the order?

When |Supply| << |Supplier|

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);
```

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

O(n)

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

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();
```

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

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();
```

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

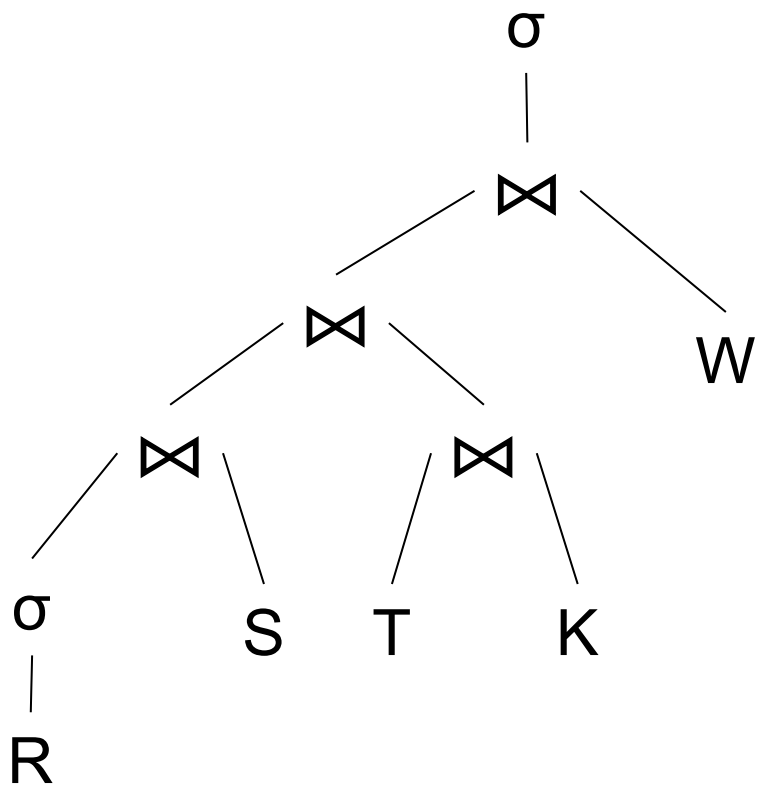
O(n log(n))

(because sorting…)

Main Memory Algorithms

- Join ⋈:
  - Nested loop join
  - Hash join
  - Merge join
- Selection σ
  - "on-the-fly"
- Group by γ
  - Hash–based
  - Merge-based

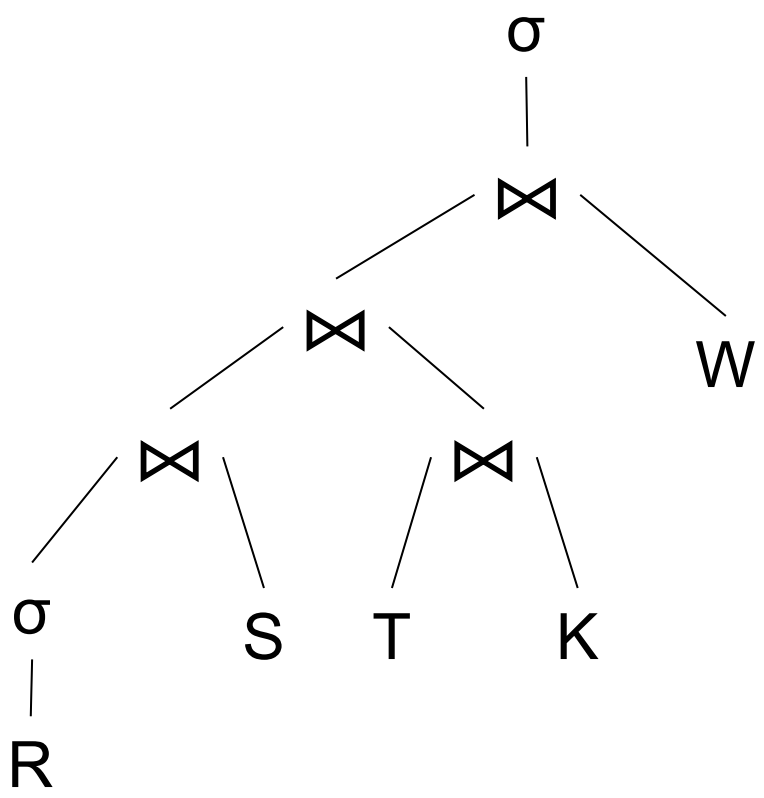
How Do We Combine Them?



How Do We Combine Them?

The *Iterator Interface*

- open()
- next()
- close()



Implementing Query Operators with the Iterator Interface

```
interface Operator {









}
```

Implementing Query Operators with the Iterator Interface

```
interface Operator {

  // initializes operator state
  // and sets parameters
  void open (...);









}
```

Implementing Query Operators with the Iterator Interface

```
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 ();




}
```

Implementing Query Operators with the Iterator Interface

```
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 ();
}
```

Implementing Query Operators with the Iterator Interface

Example “on the fly” selection operator

```
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 ();
}
```

Implementing Query Operators with the Iterator Interface

```
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 “on the fly” selection operator

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

}
```

Implementing Query Operators with the Iterator Interface

```
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 “on the fly” selection operator

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

    }

}
```

Implementing Query Operators with the Iterator Interface

```
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 “on the fly” 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);
    }

  }

}
```

Implementing Query Operators with the Iterator Interface

```
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 “on the fly” 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;
  }

}
```

Implementing Query Operators with the Iterator Interface

```
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 “on the fly” 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(); }
}
```

Implementing Query Operators with the Iterator Interface

```
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 ();
}
```

Query plan execution

```
Operator q = parse("SELECT ...");
q = optimize(q);

q.open();
while (true) {
  Tuple t = q.next();
  if (t == null) break;
  else printOnScreen(t);
}
q.close();
```

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

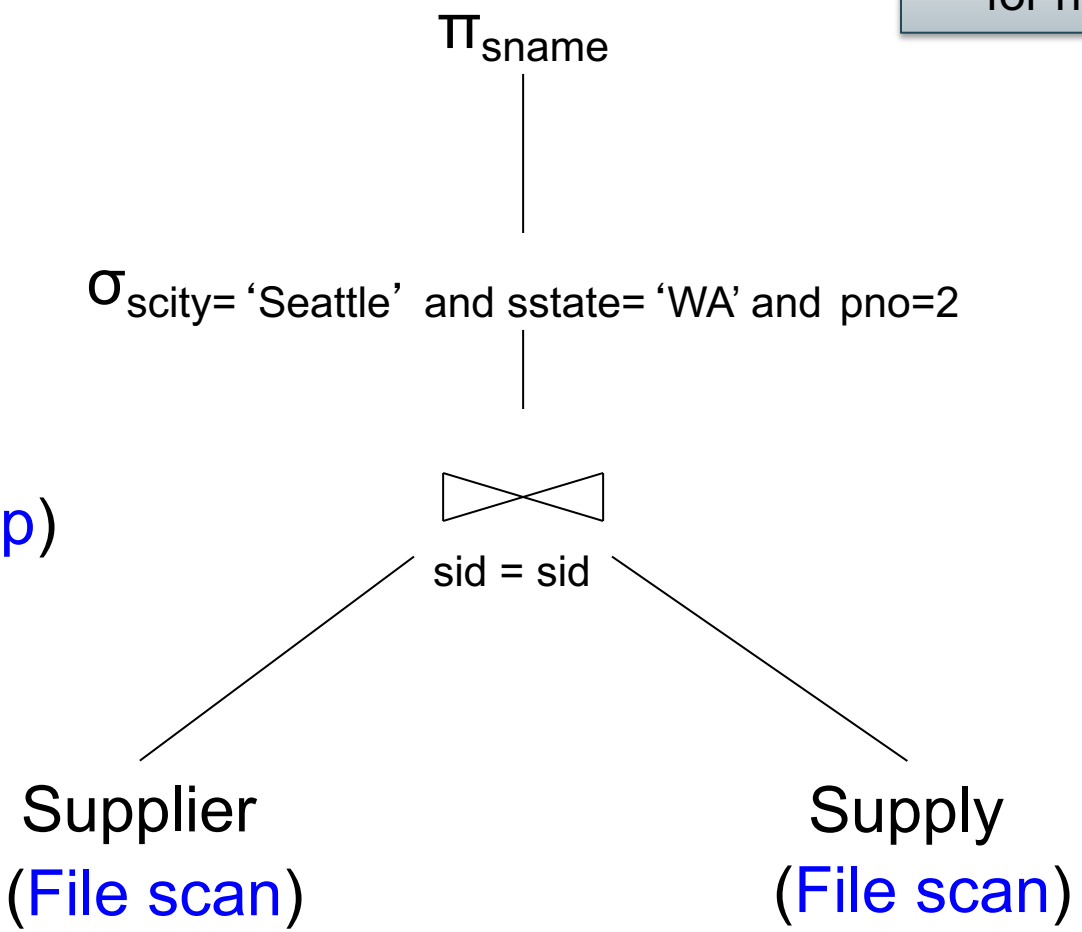
Pipelining

Discuss: open/next/close for nested loop join

(On the fly)

(On the fly)

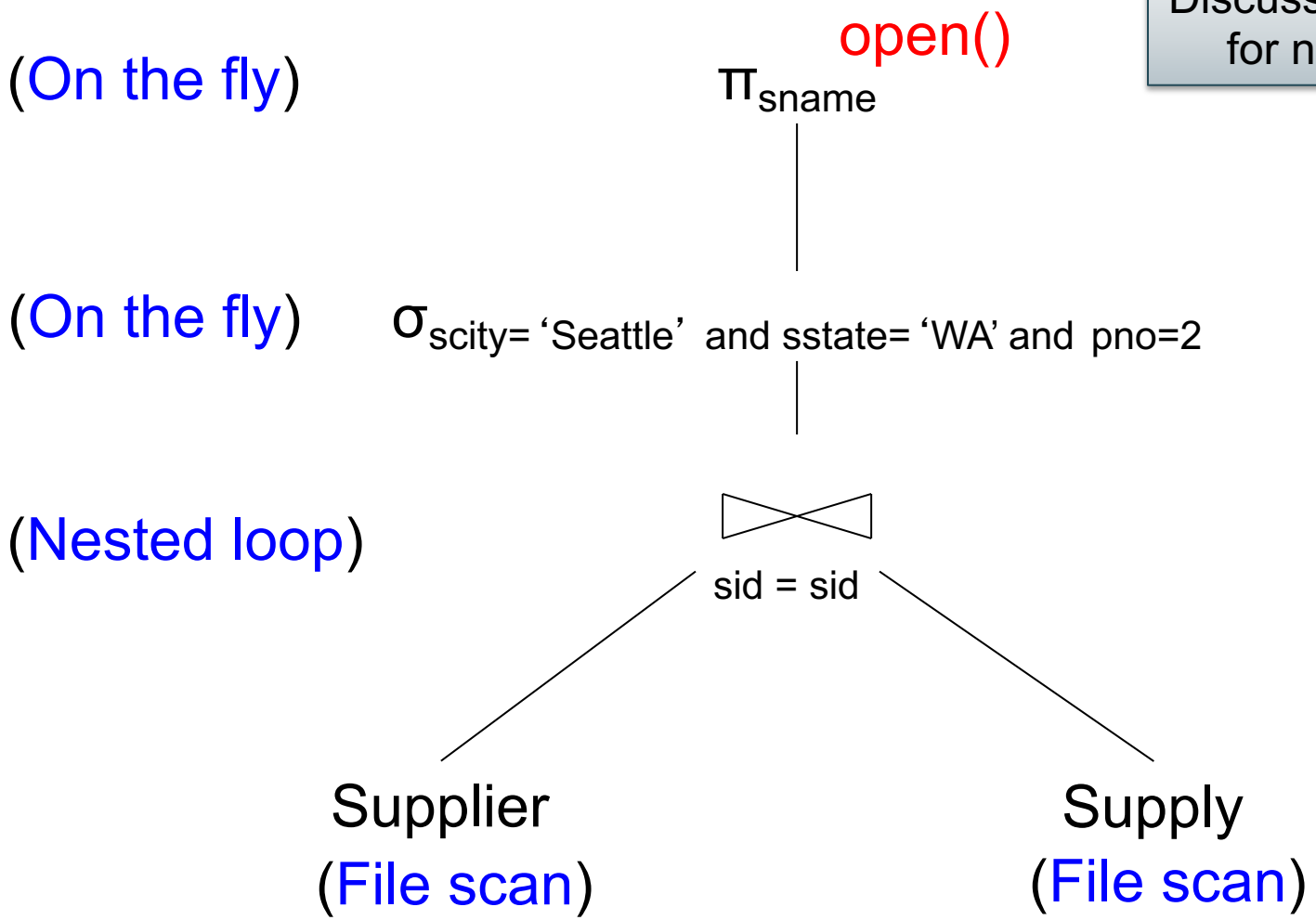
(Nested loop)



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

Pipelining

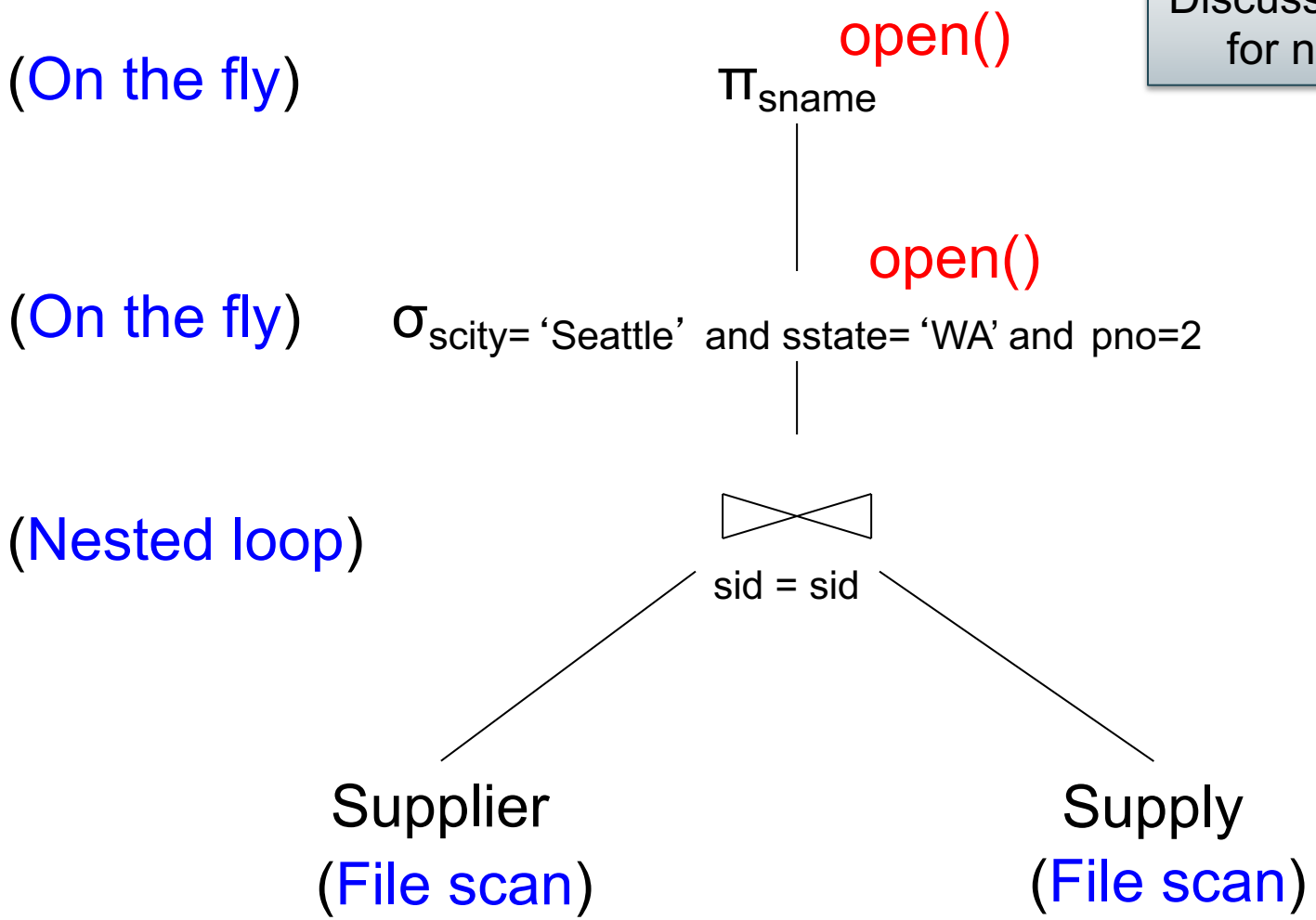
Discuss: open/next/close for nested loop join



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

Pipelining

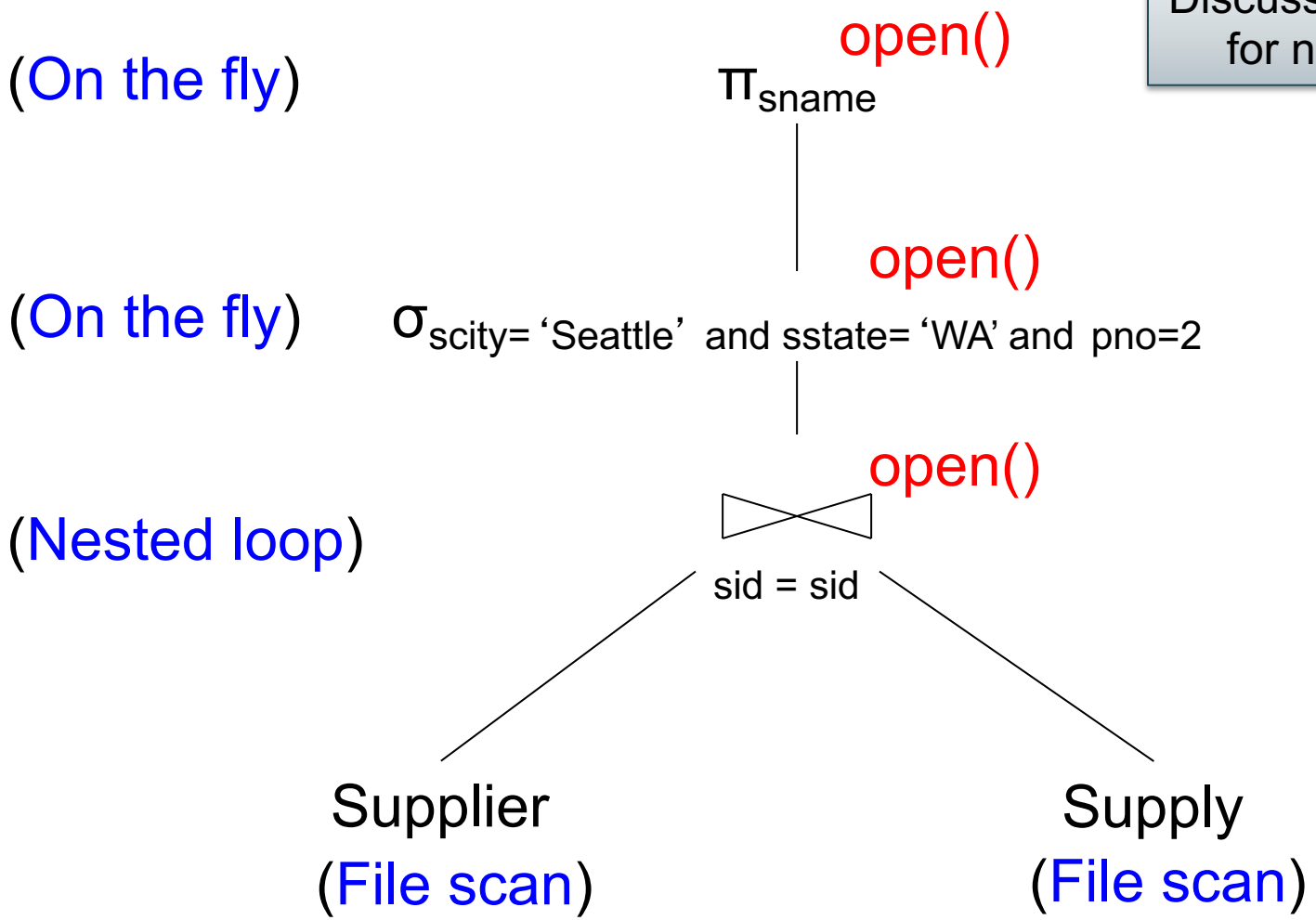
Discuss: open/next/close for nested loop join



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

Pipelining

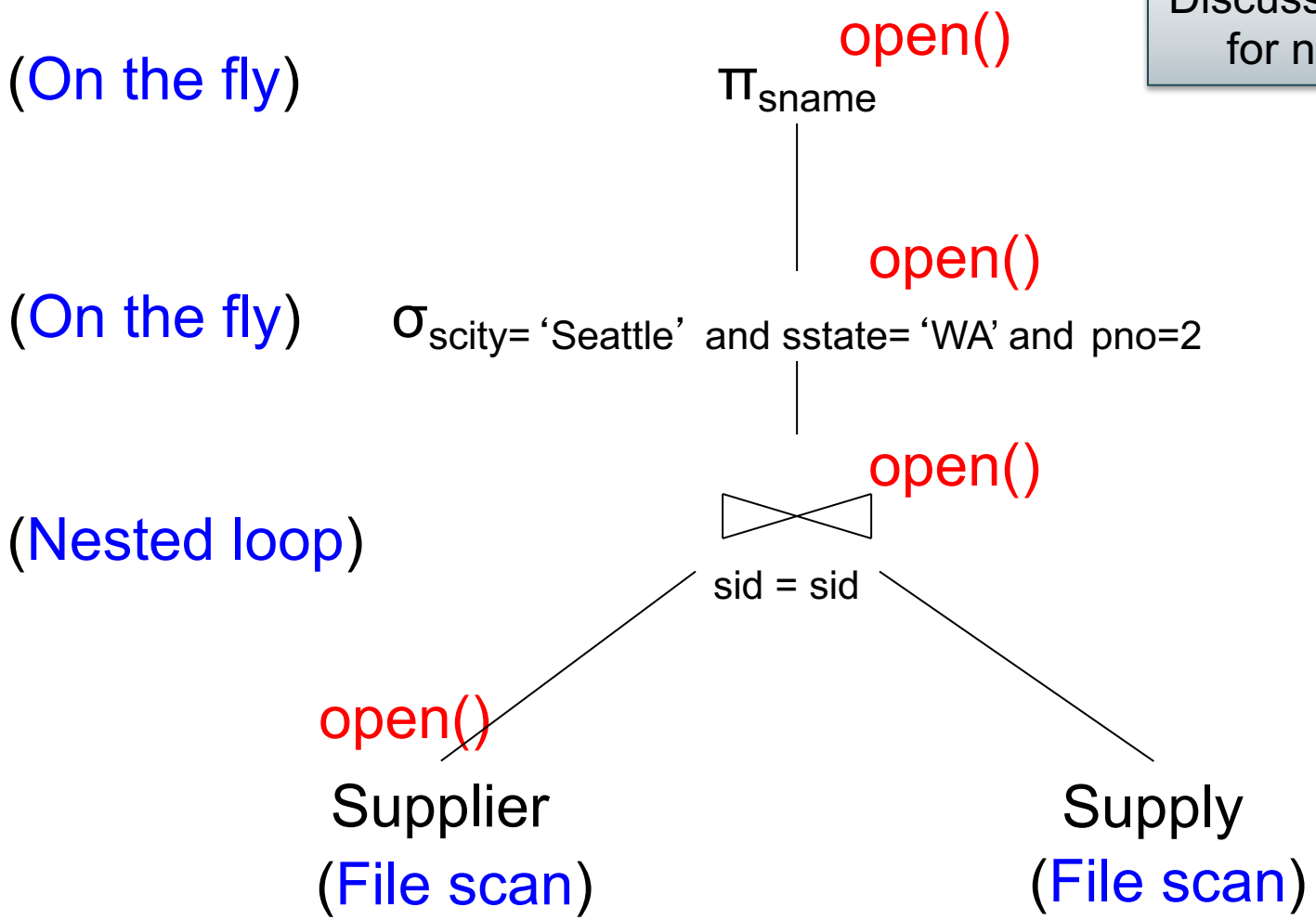
Discuss: open/next/close for nested loop join



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

Pipelining

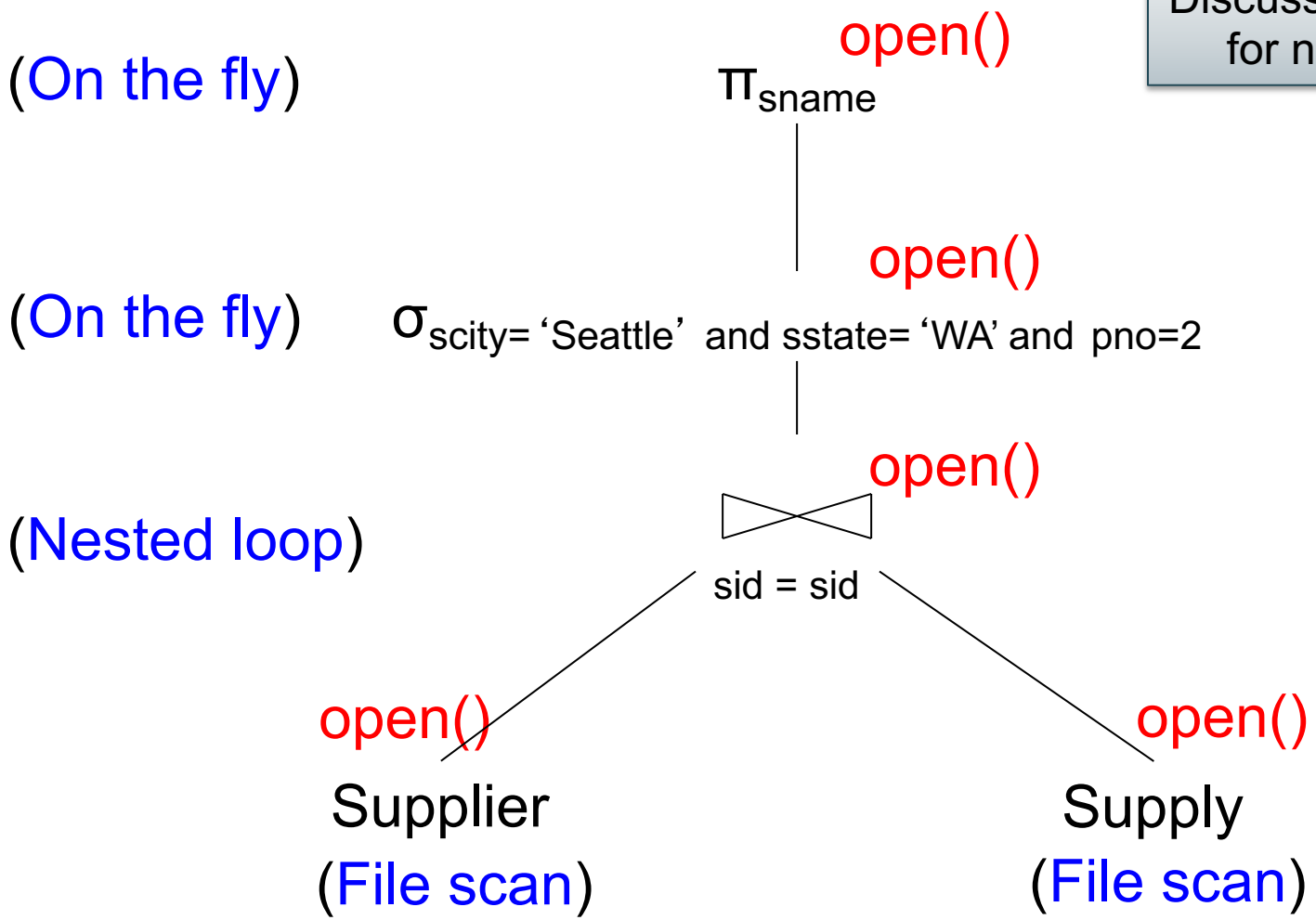
Discuss: open/next/close for nested loop join



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

Pipelining

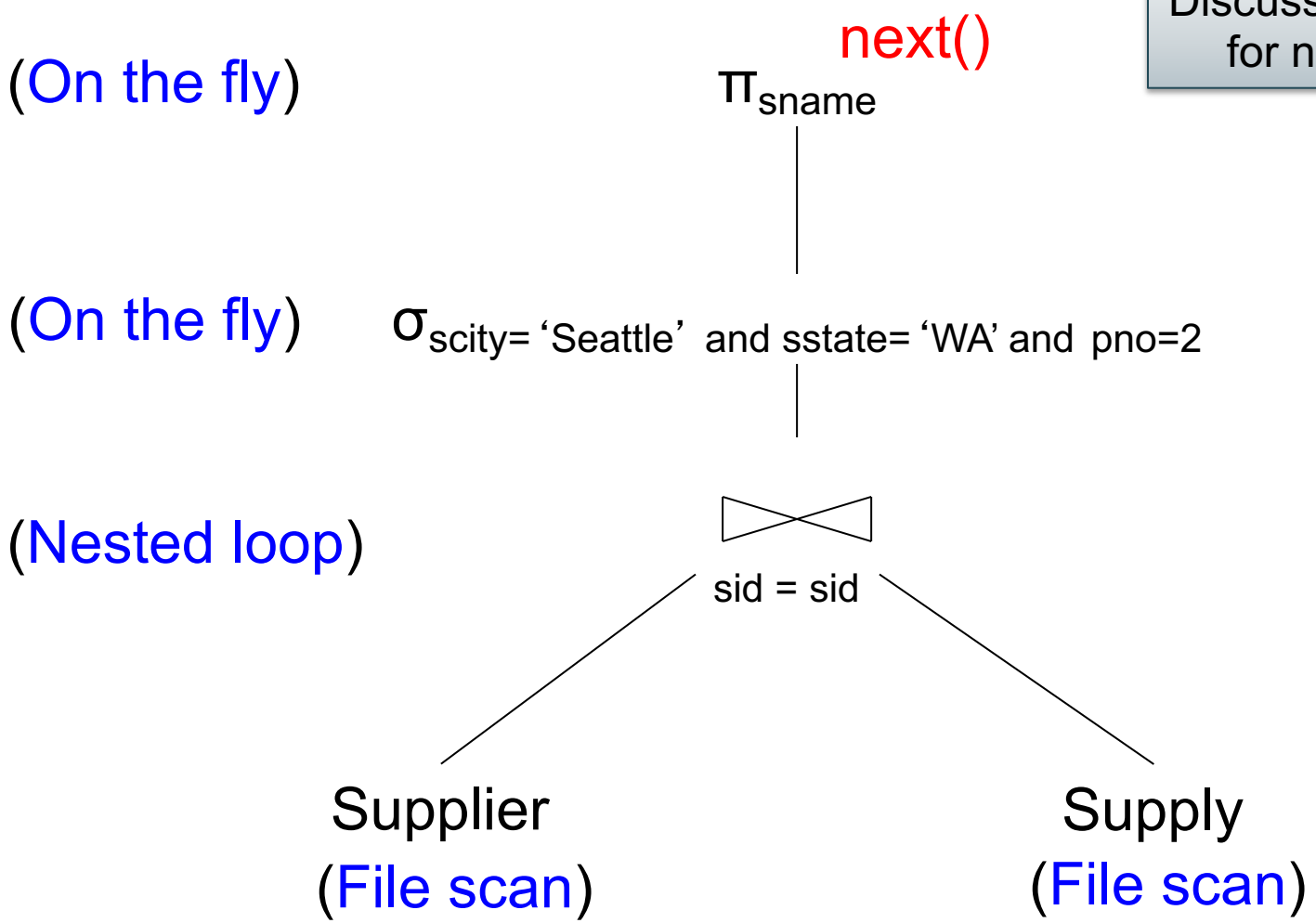
Discuss: open/next/close for nested loop join



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

Pipelining

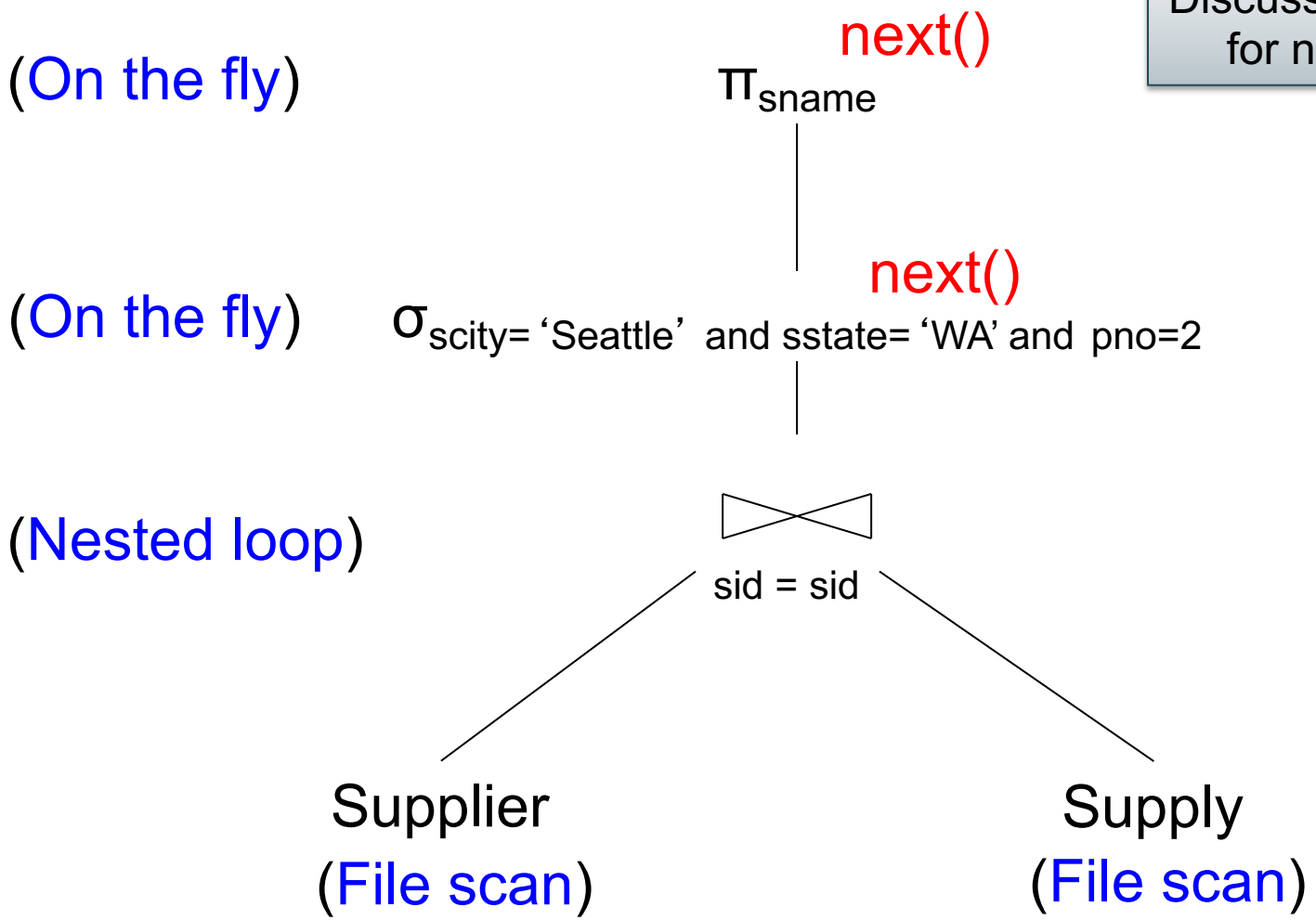
Discuss: open/next/close for nested loop join



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

Pipelining

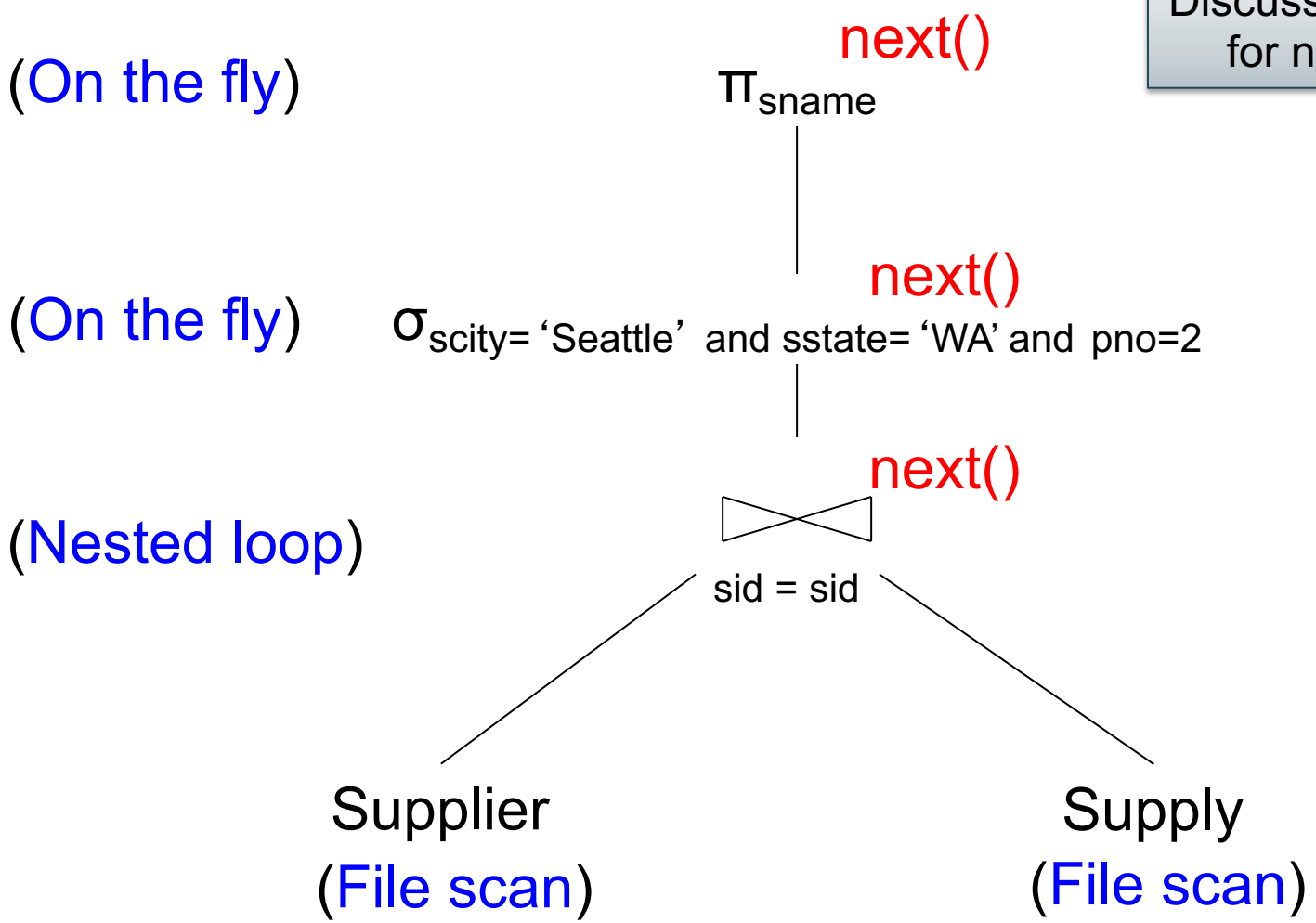
Discuss: open/next/close for nested loop join



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

Pipelining

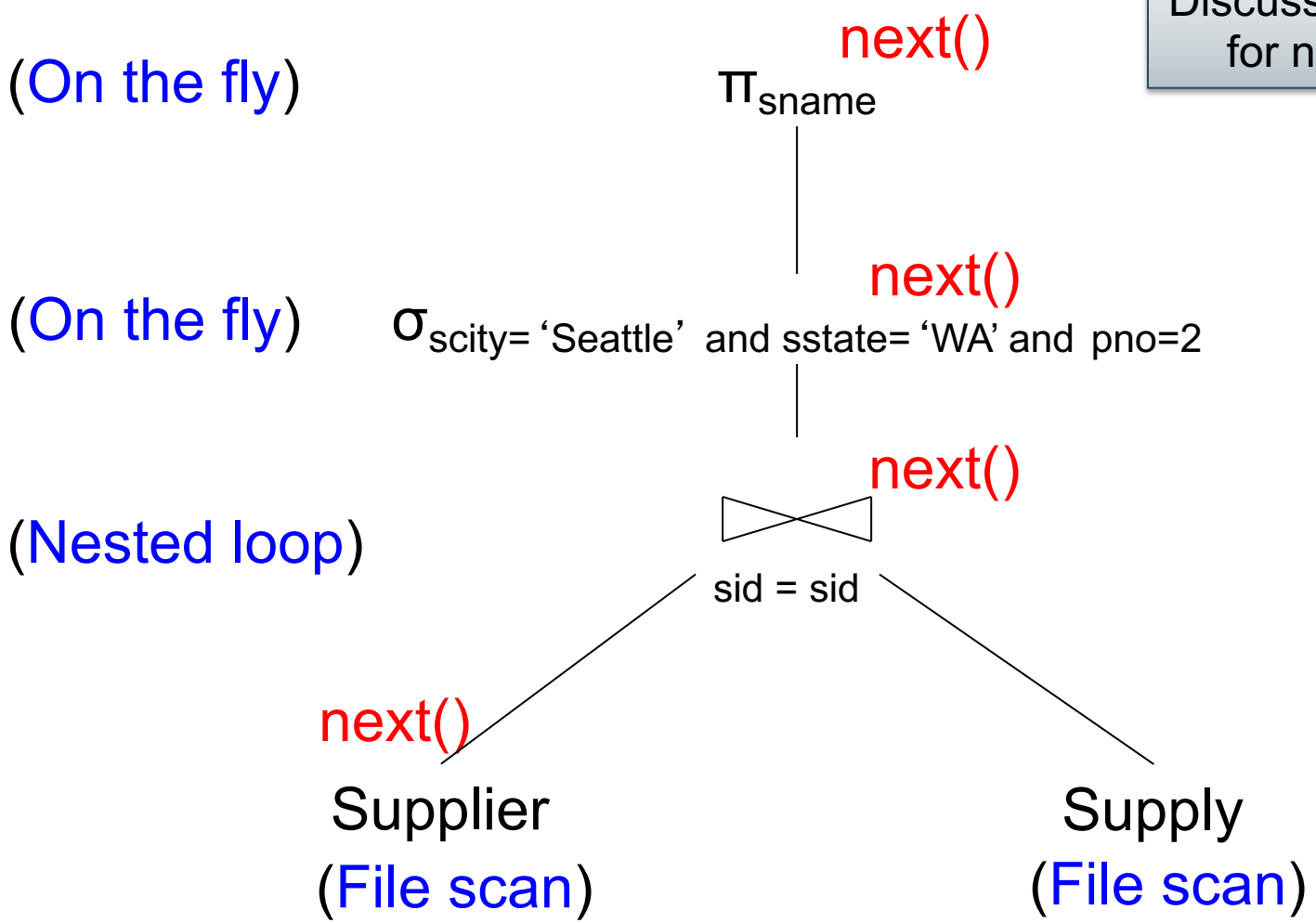
Discuss: open/next/close for nested loop join



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

Pipelining

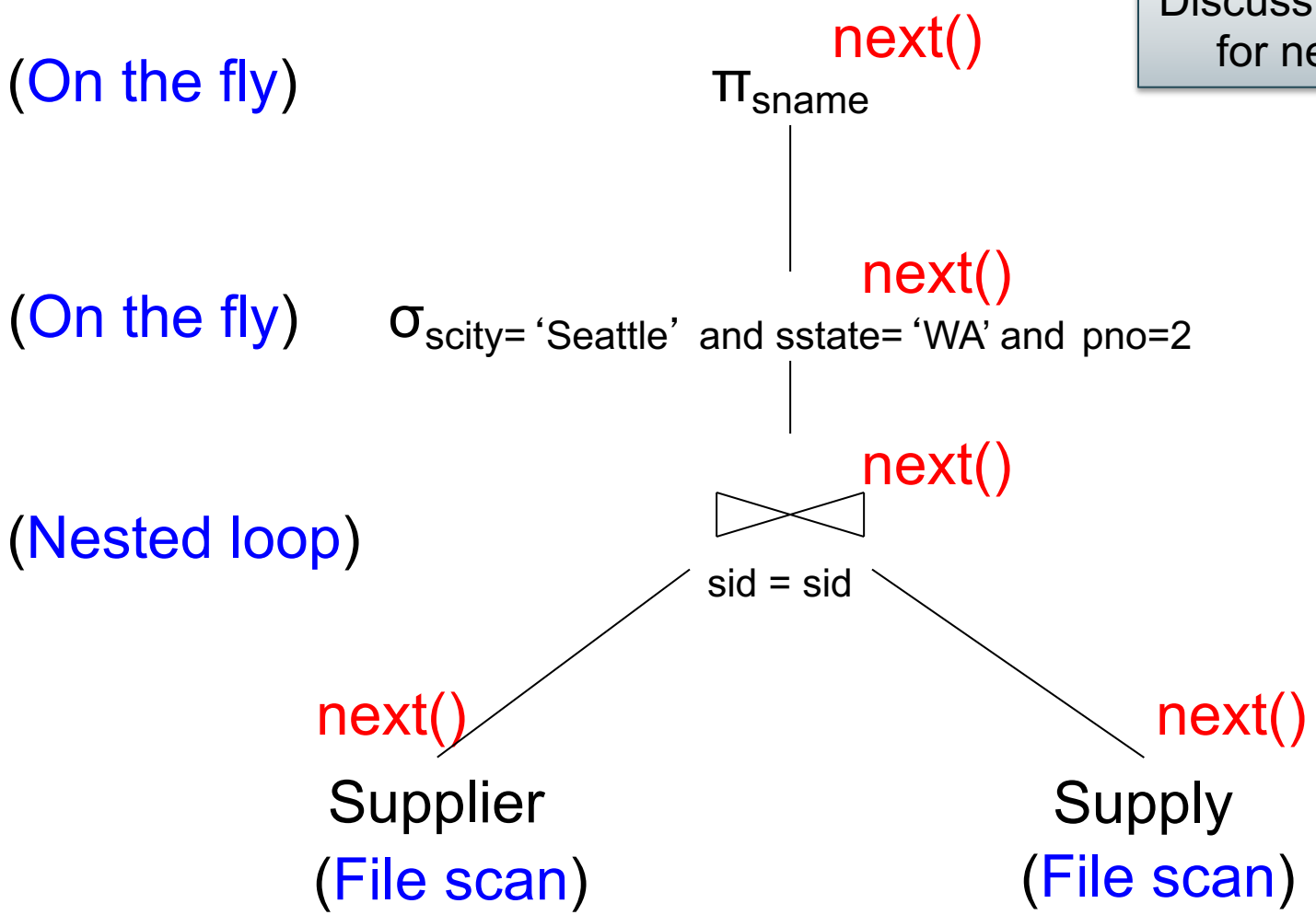
Discuss: open/next/close for nested loop join



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

Pipelining

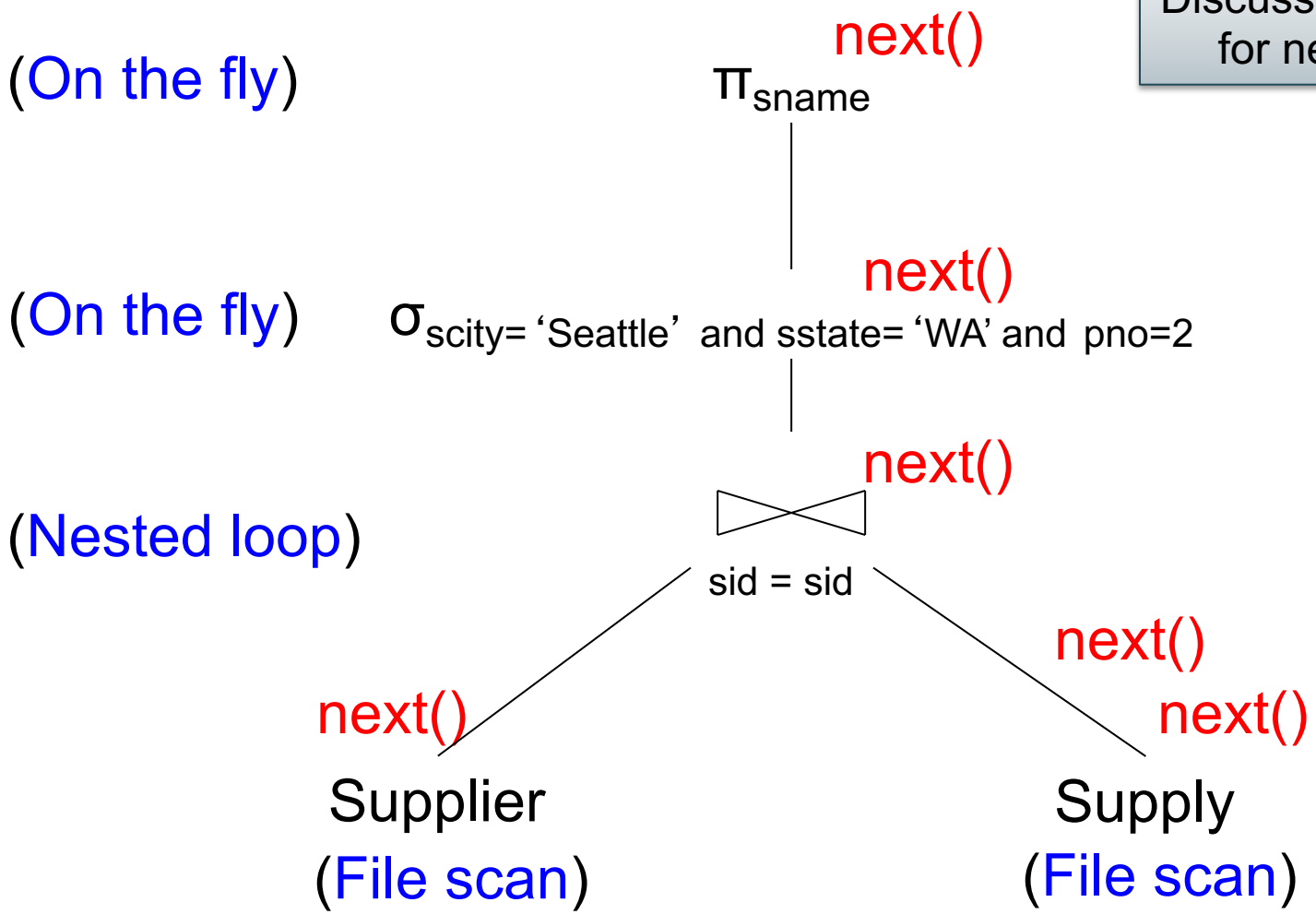
Discuss: open/next/close for nested loop join



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

Pipelining

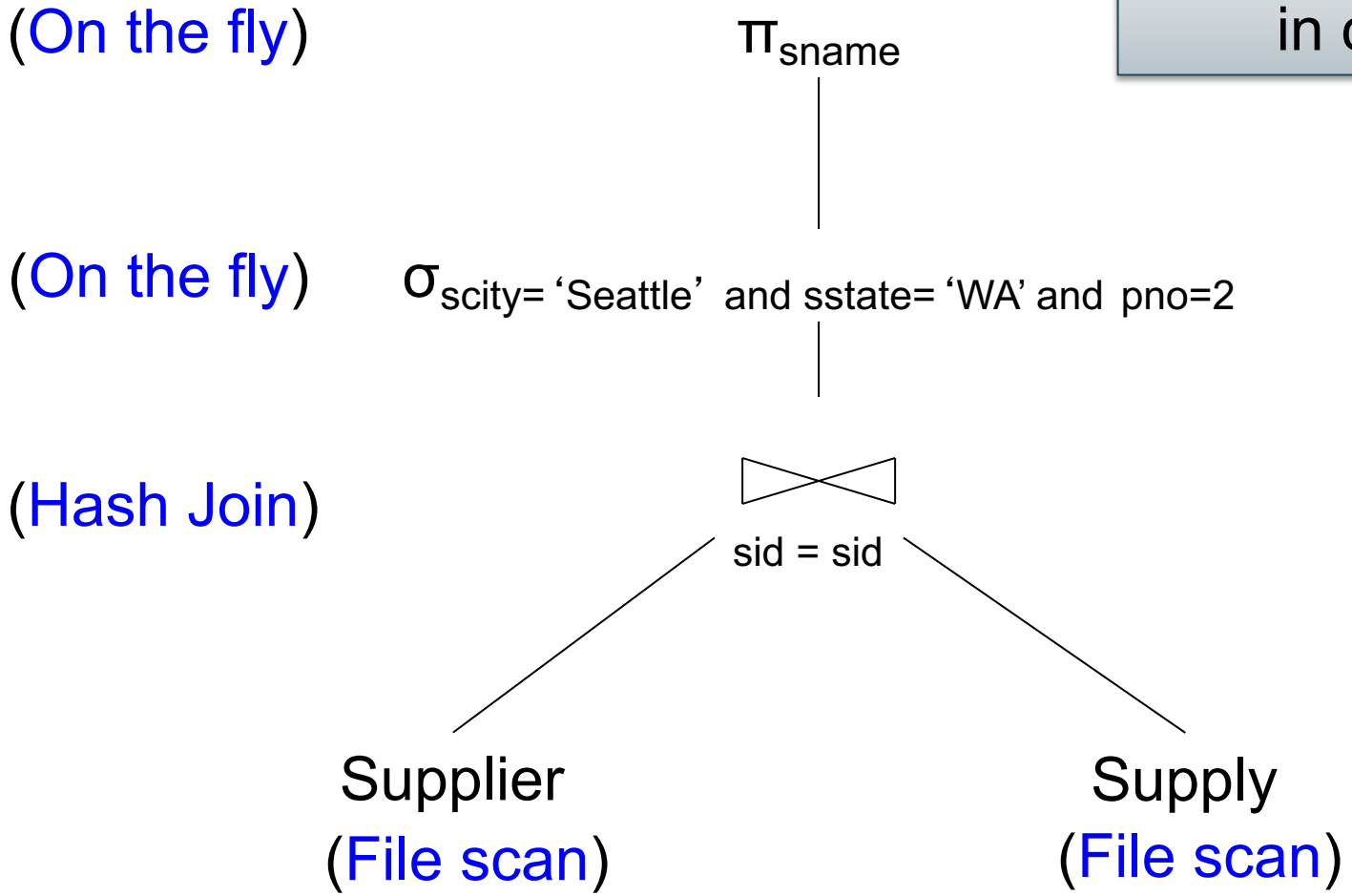
Discuss: open/next/close for nested loop join



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

Pipelining

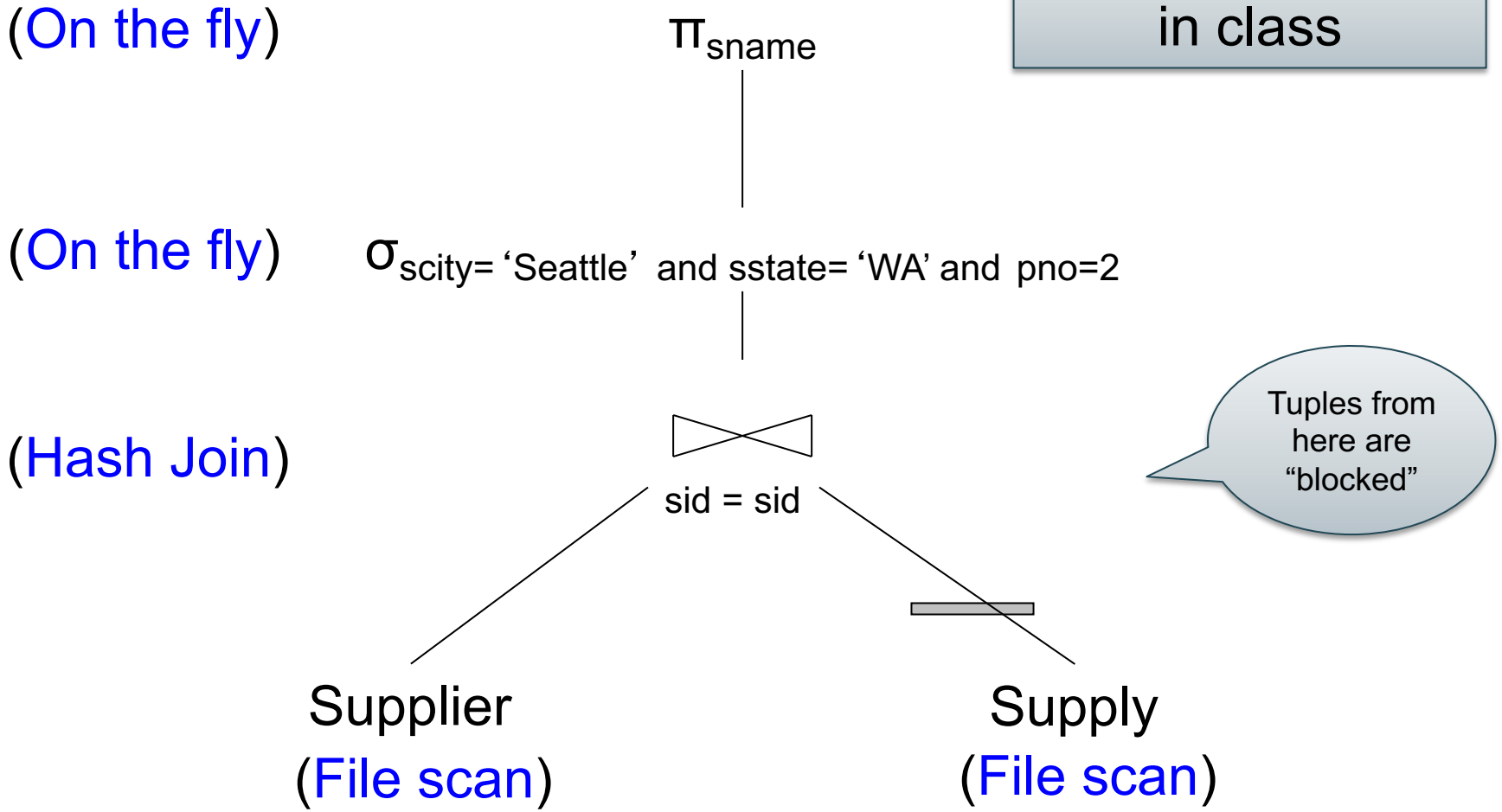
Discuss hash-join in class



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

Pipelining

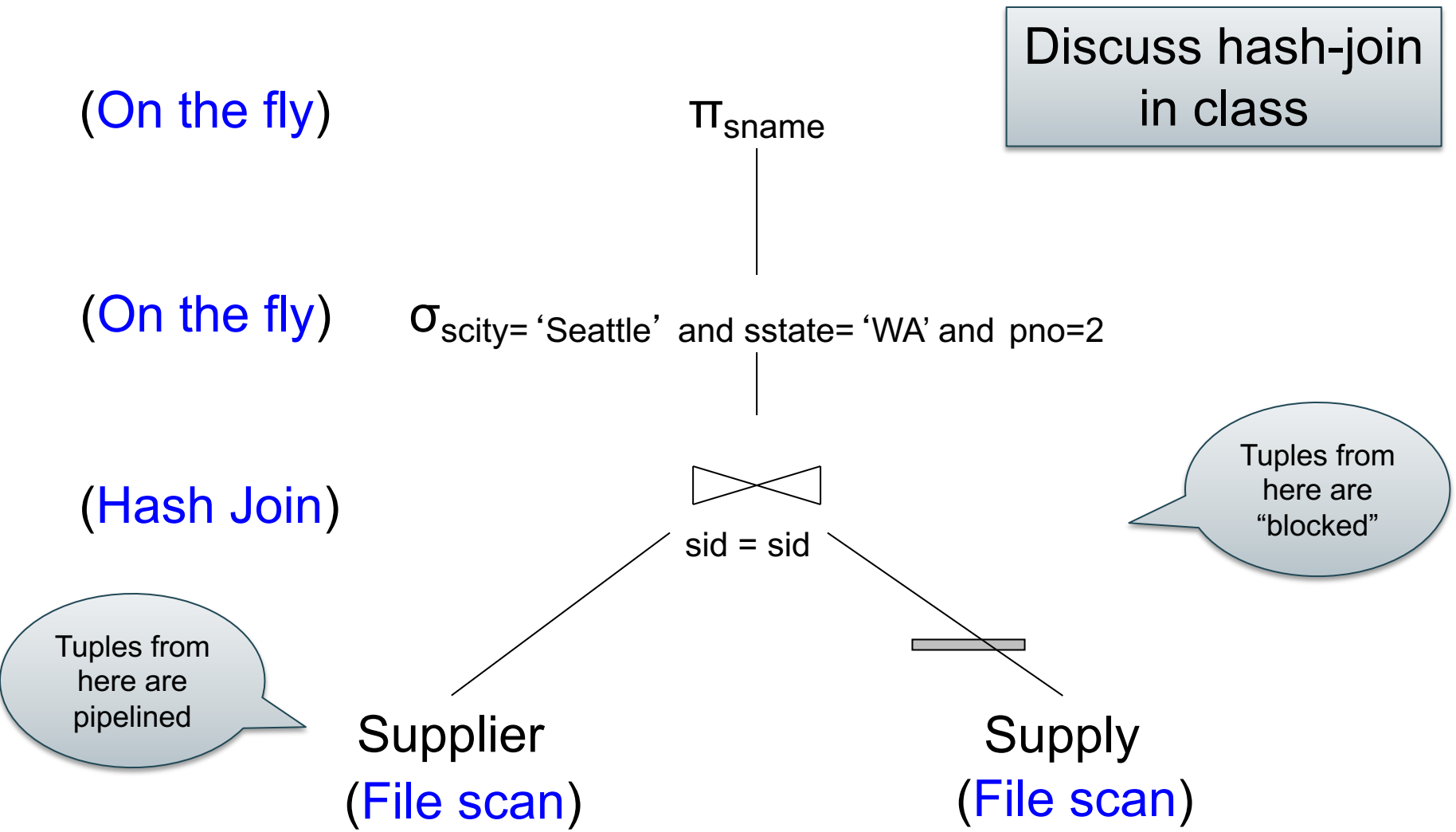
Discuss hash-join in class



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

Pipelining

Discuss hash-join in class



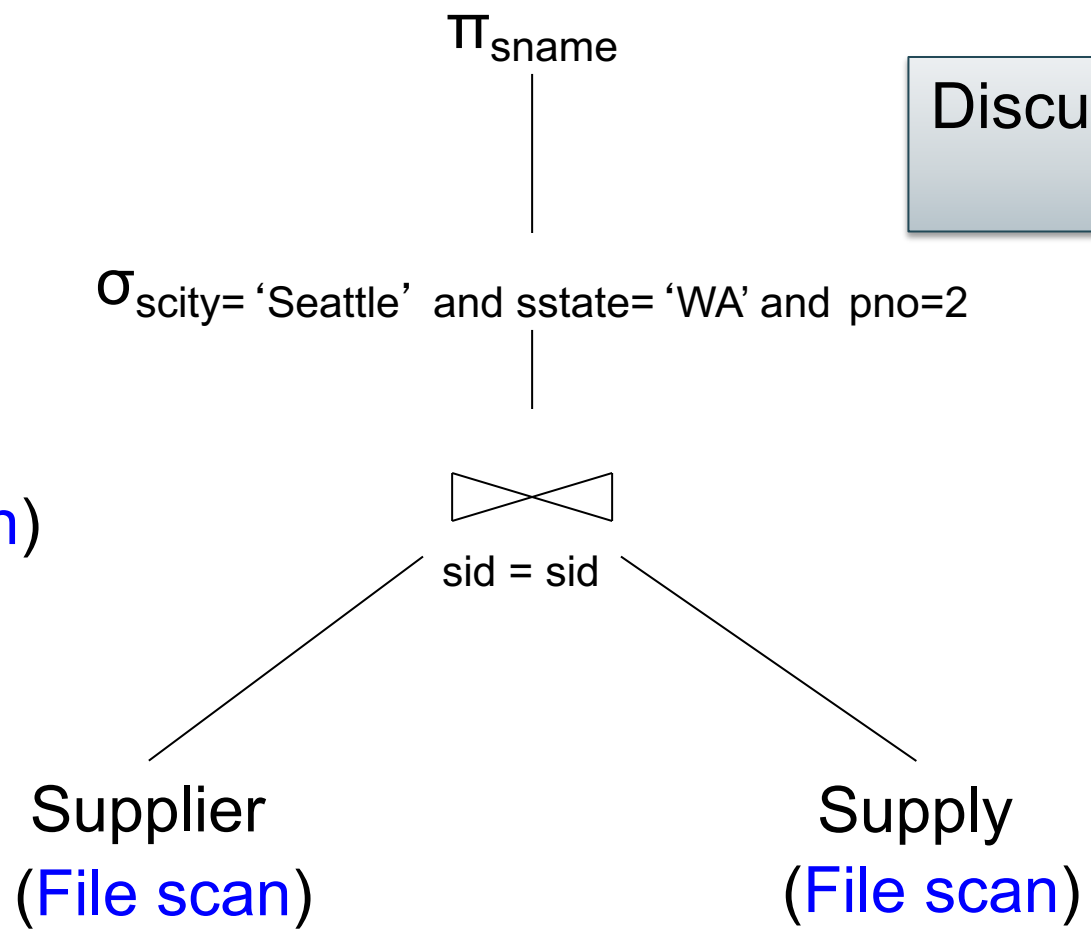
Supplier(sid, sname, scity, sstate)
Supply(sid, pno, quantity)

Blocked Execution

(On the fly)

(On the fly)

(Merge Join)



Discuss merge-join in class

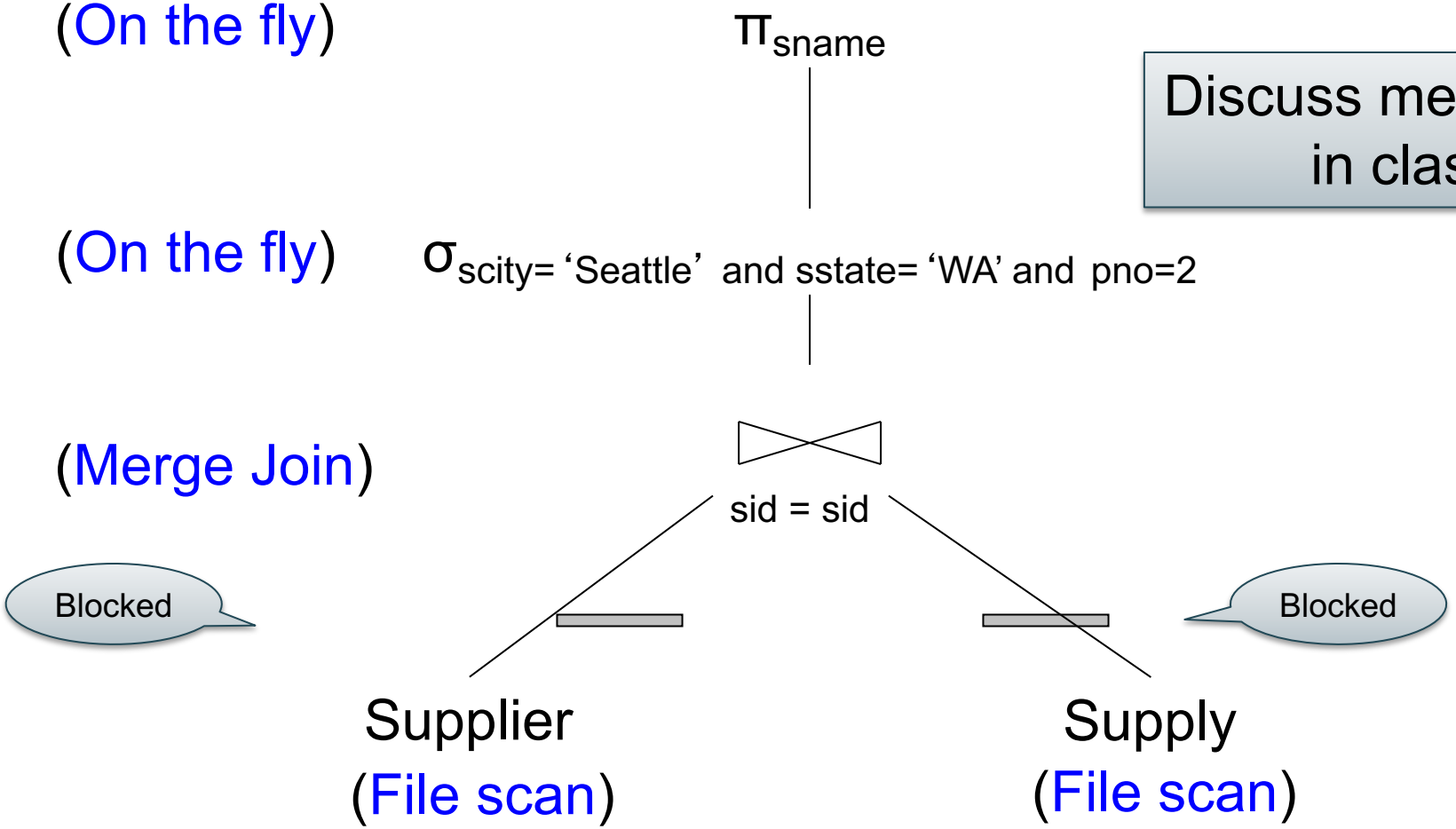
Supplier(sid, sname, scity, sstate)
Supply(sid, pno, quantity)

Blocked Execution

(On the fly)

(On the fly)

(Merge Join)

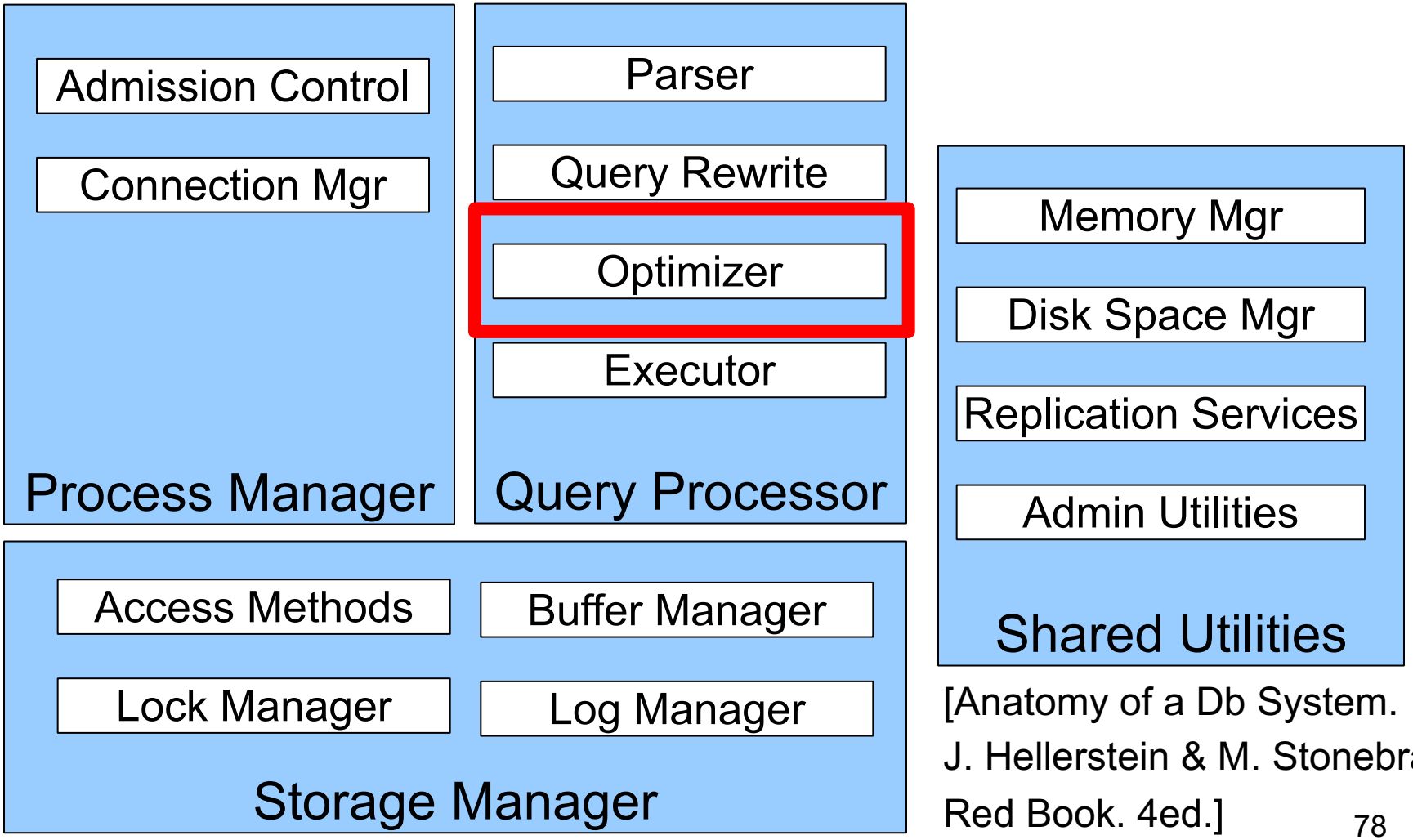


Discuss merge-join in class

Pipeline v.s. Blocking

- Pipeline
  - A tuple moves all the way through up the query plan
  - Advantages: speed
  - Disadvantage: need all hash at the same time in memory
- Blocking
  - The entire result of the subplan is computed (and stored to disk) before the first tuple is sent up the plan
  - Advantage: saves memory
  - Disadvantage: slower

DBMS Architecture



[Anatomy of a Db System. J. Hellerstein & M. Stonebraker. Red Book. 4ed.]

Query Optimizer

- SQL → Relational Algebra Plan (RA)
- RA → Optimized RA
- Algebraic identities of RA; examples:
  1. Pushing selections down
  2. Join reorder
  3. Pushing aggregates down

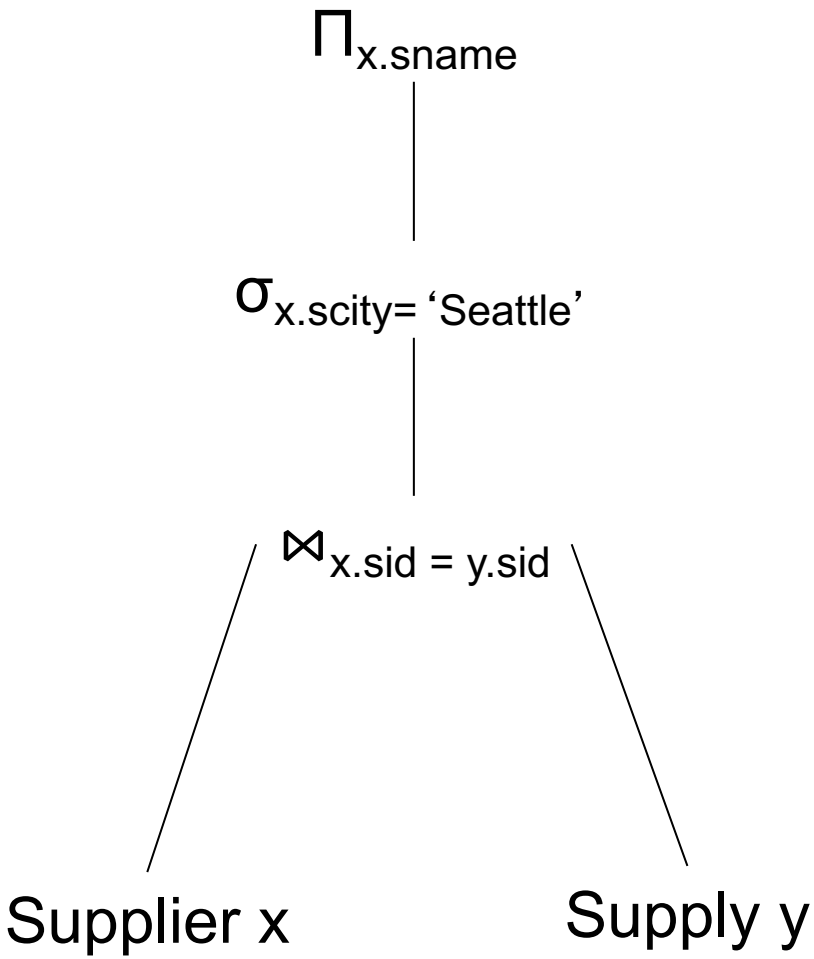
Supplier(sid, sname, scity, sstate)
Supply(sid, pno, quantity)

Example Optimization

```
SELECT x.name
FROM.   Supplier x, Supply y
WHERE x.sid = y.sid
        and x.scity = 'Seattle'
```

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

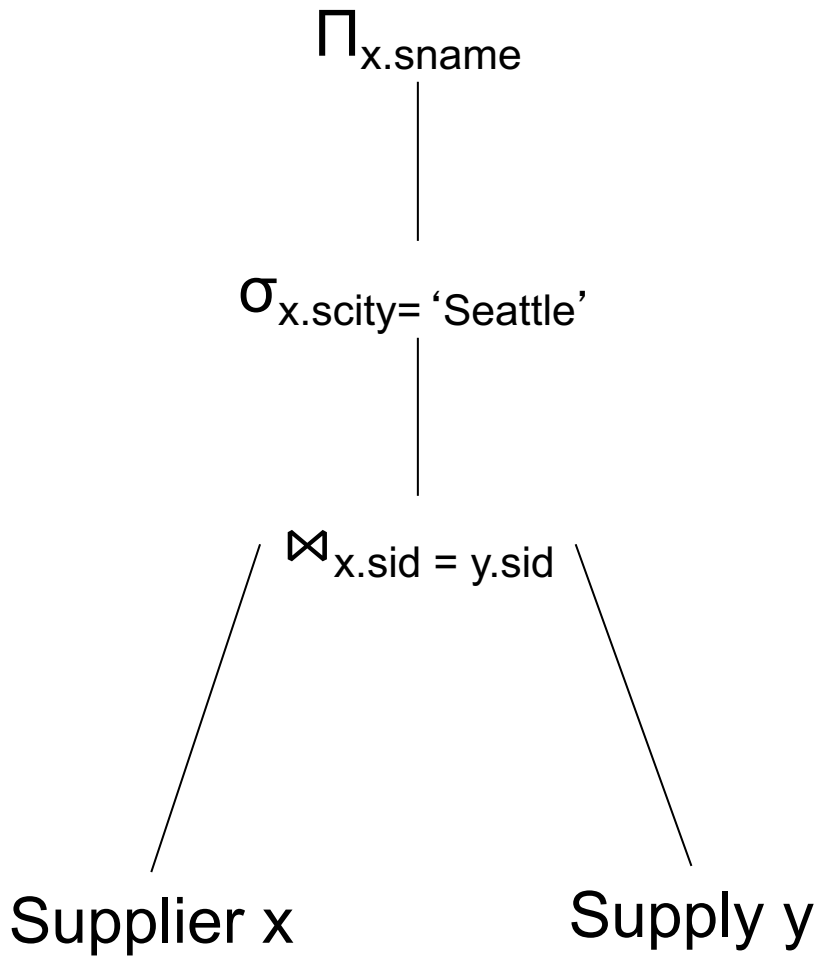
Example Optimization




```
SELECT x.name
FROM.   Supplier x, Supply y
WHERE x.sid = y.sid
        and x.scity = 'Seattle'
```

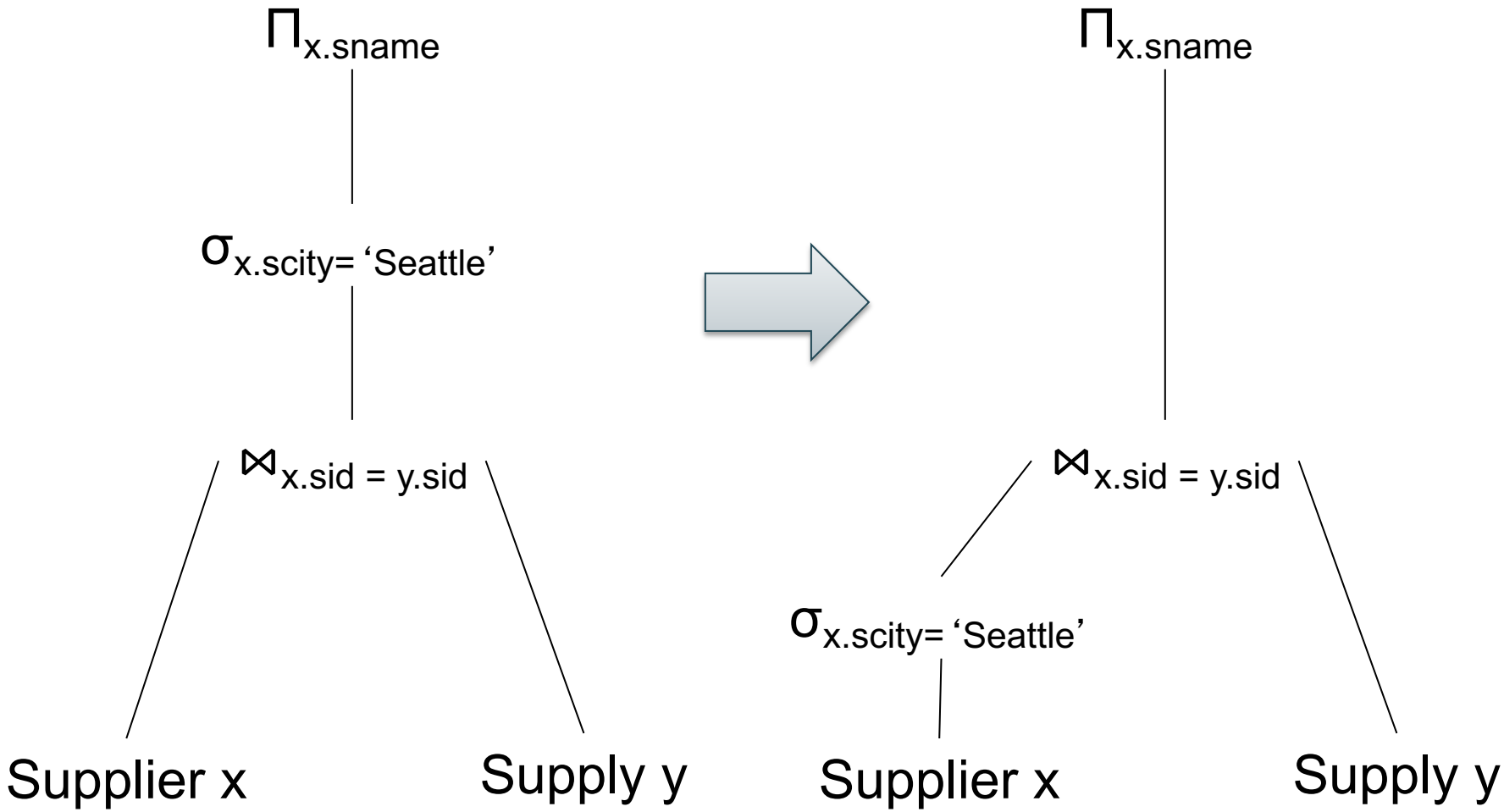
Supplier(sid, sname, scity, sstate)
Supply(sid, pno, quantity)

Push Selections Down



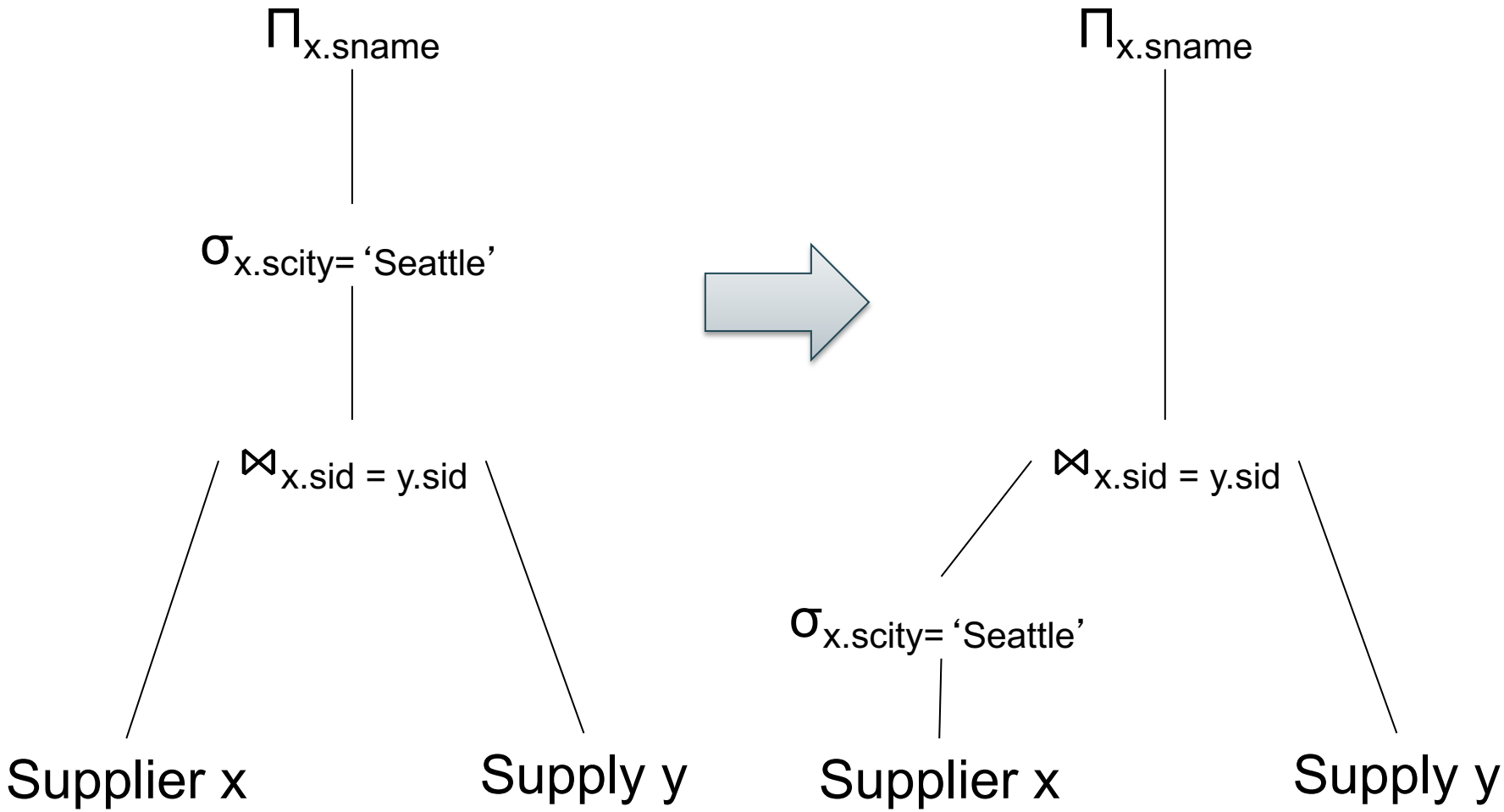
Supplier(sid, sname, scity, sstate)
Supply(sid, pno, quantity)

Push Selections Down



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

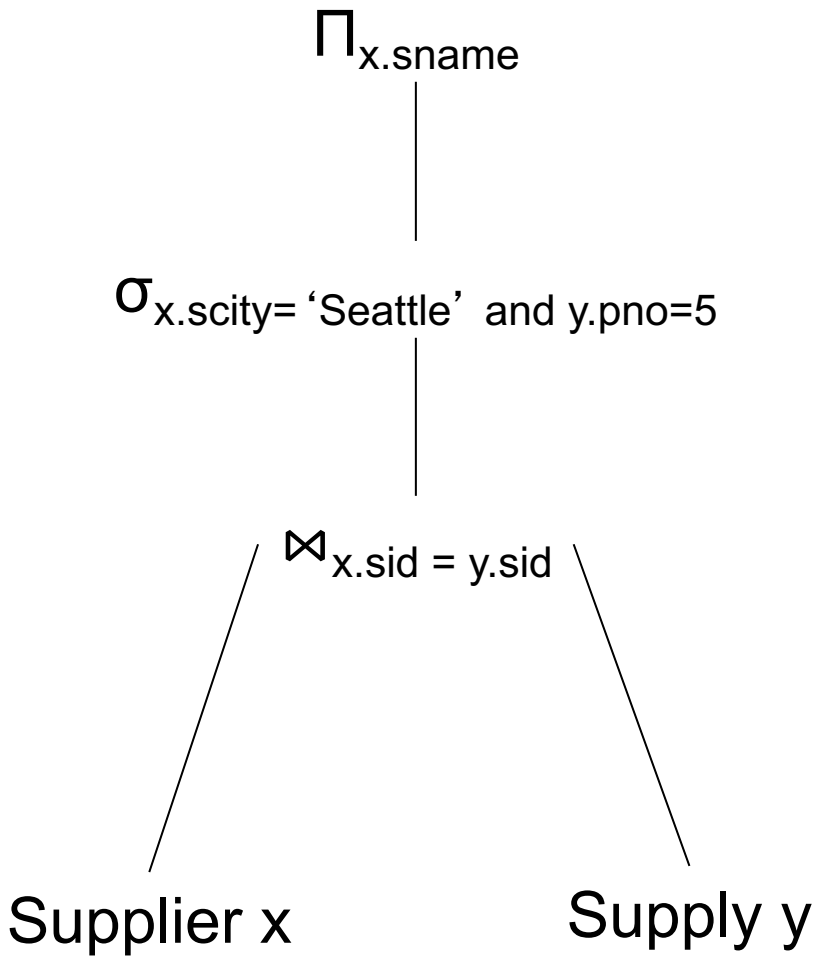
Push Selections Down



$\sigma_C(R \bowtie S) = \sigma_C(R) \bowtie S$ when C refers only to R

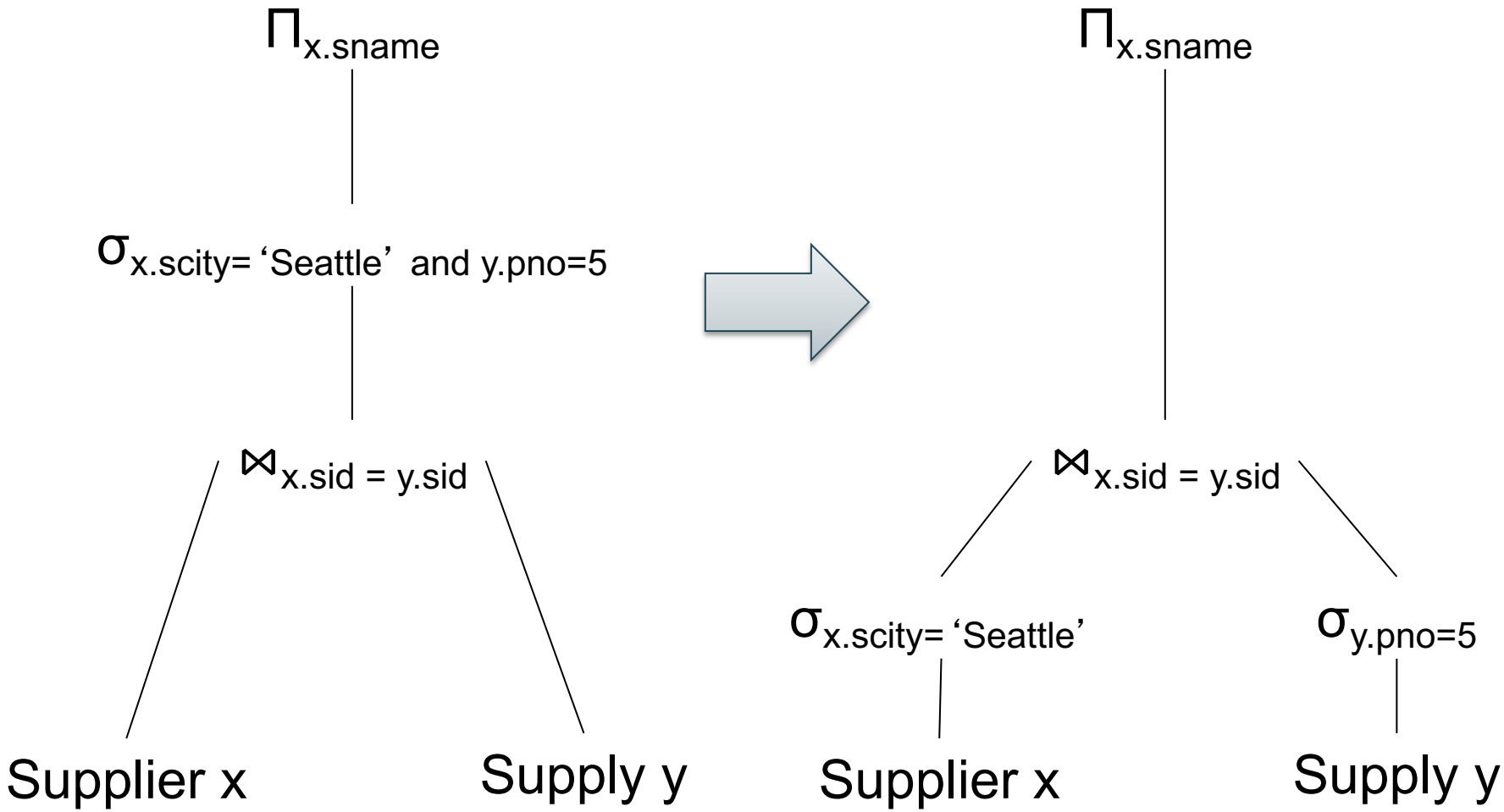
Supplier(sid, sname, scity, sstate)
Supply(sid, pno, quantity)

Push Selections Down



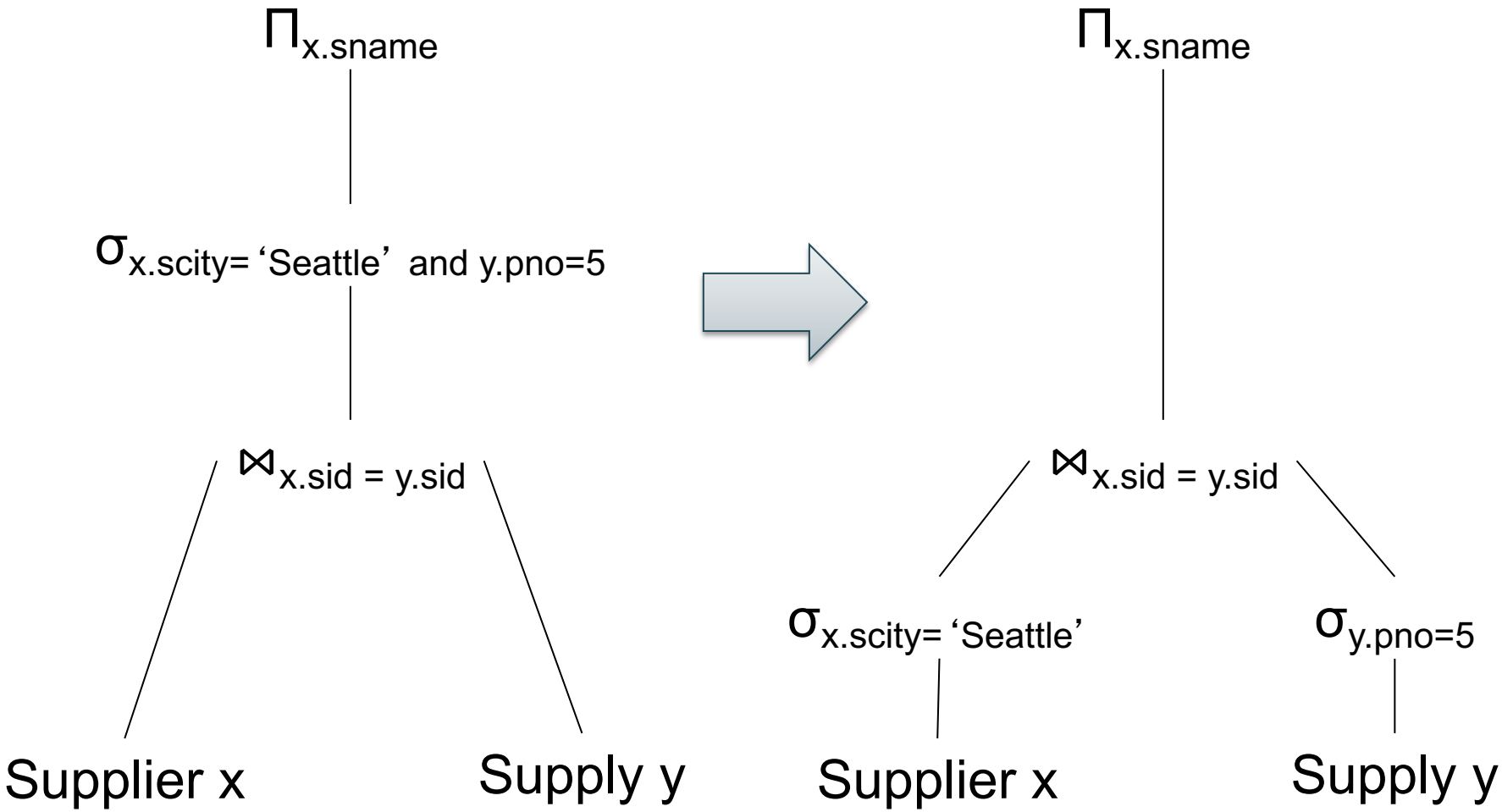
Supplier(sid, sname, scity, sstate)
Supply(sid, pno, quantity)

Push Selections Down



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

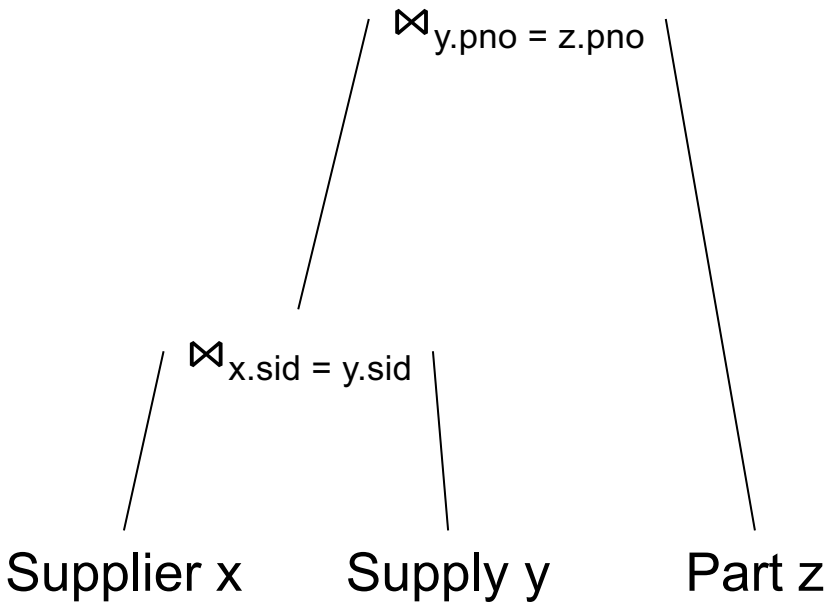
Push Selections Down



$$\sigma_{C1 \text{ and } C2}(R \bowtie S) = \sigma_{C1}(\sigma_{C2}(R \bowtie S)) = \sigma_{C1}(R \bowtie \sigma_{C2}(S)) = \sigma_{C1}(R) \bowtie \sigma_{C2}(S)$$

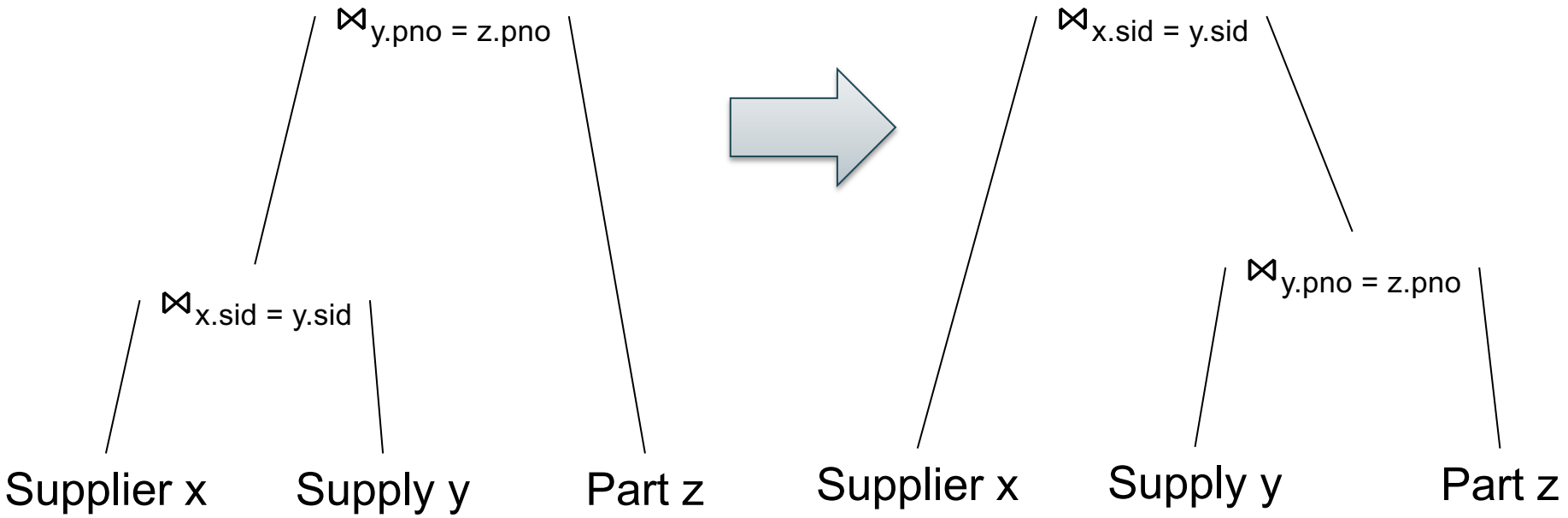
Supplier(sid, sname, scity, sstate)
Supply(sid, pno, quantity)
Part(pno, pname, pprice)

Join Reorder



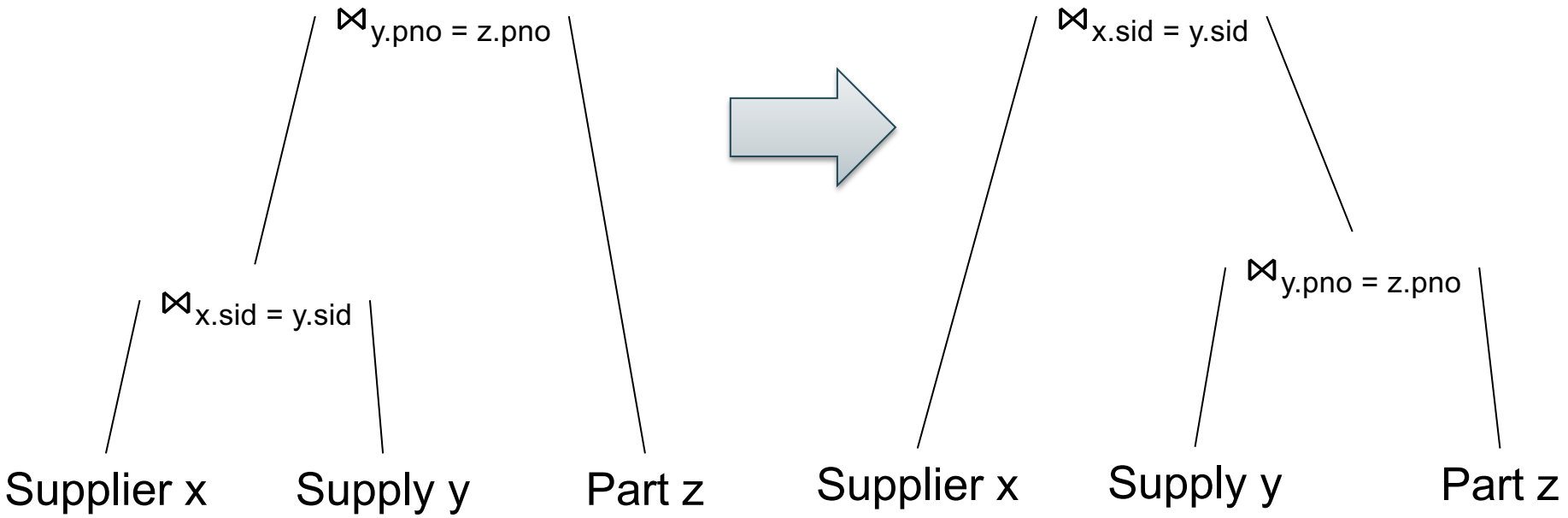
Supplier(sid, sname, scity, sstate)
Supply(sid, pno, quantity)
Part(pno, pname, pprice)

Join Reorder



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

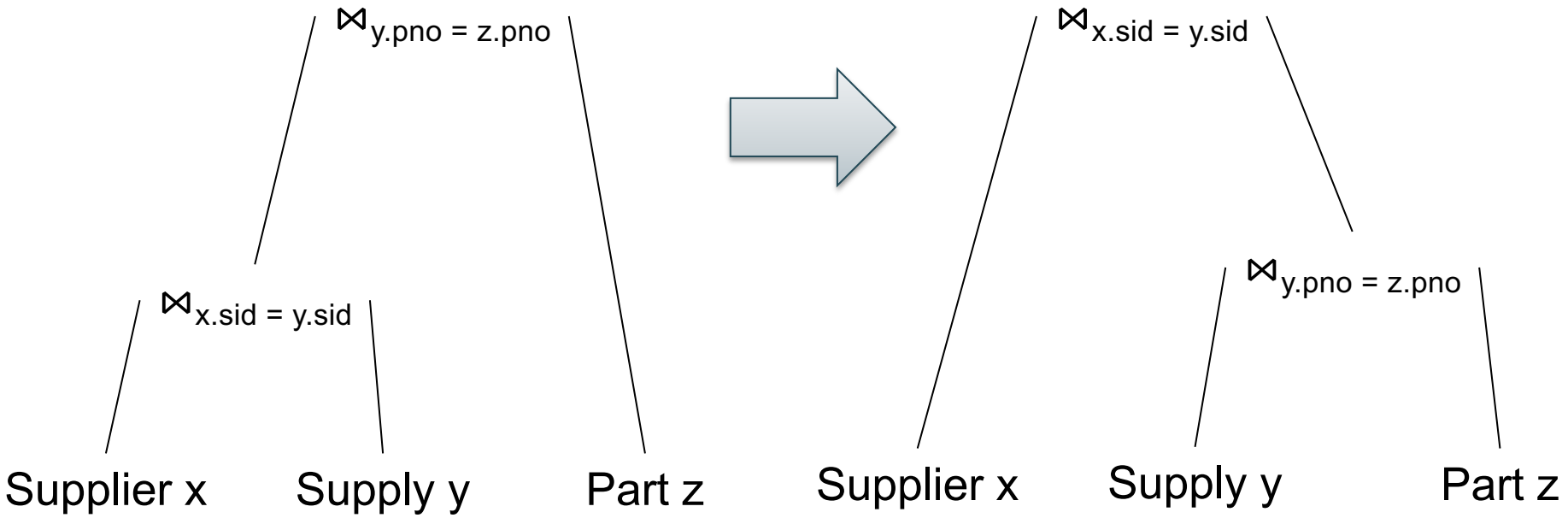
Join Reorder



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

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

Join Reorder



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

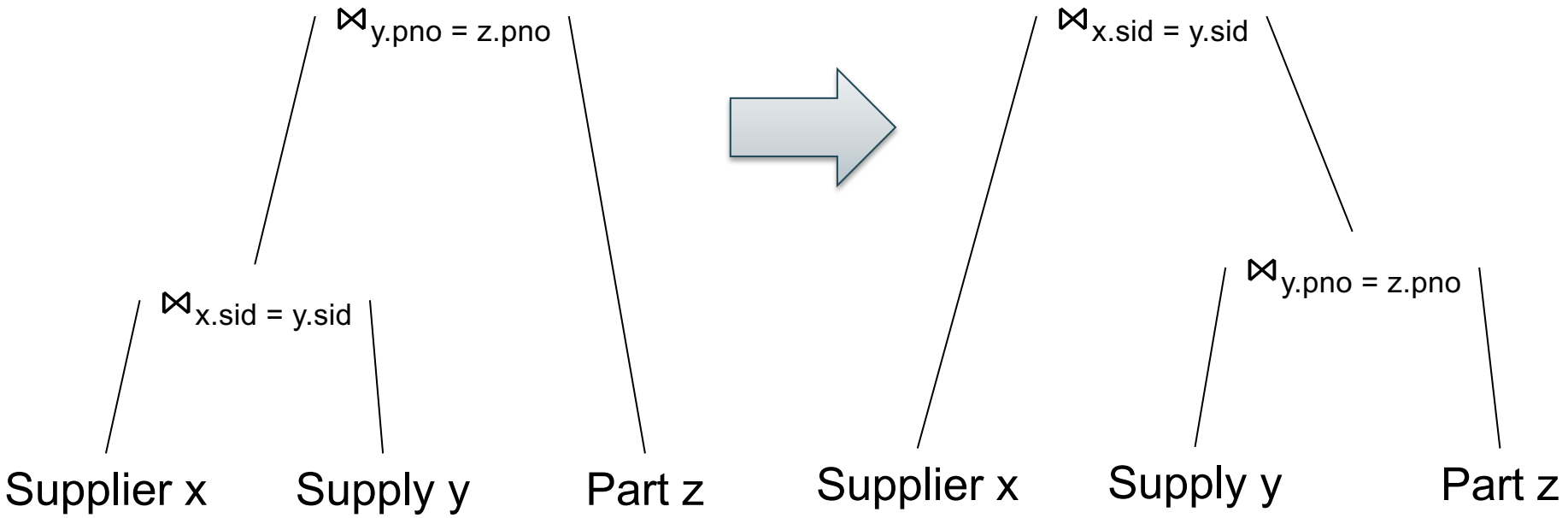
Also:

$$R \bowtie S = S \bowtie R$$

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

Join Reorder

When is one plan better than the other?



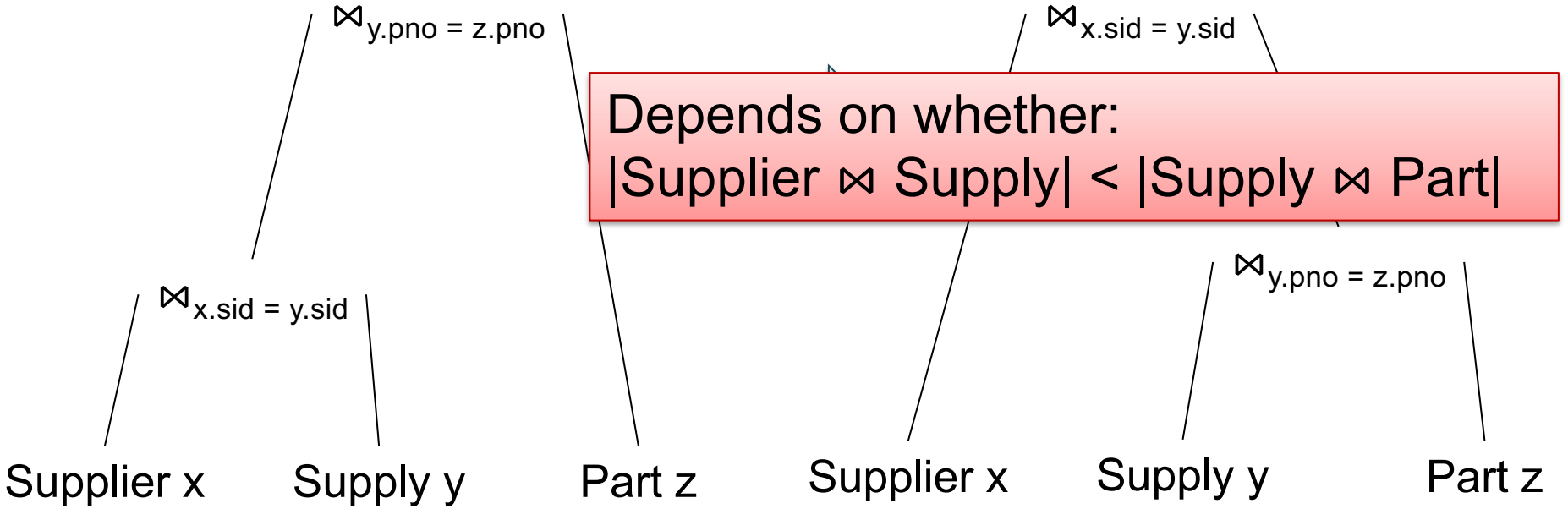
$(R \bowtie S) \bowtie T = R \bowtie (S \bowtie T)$

Also: $R \bowtie S = S \bowtie R$

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

Join Reorder

When is one plan better than the other?

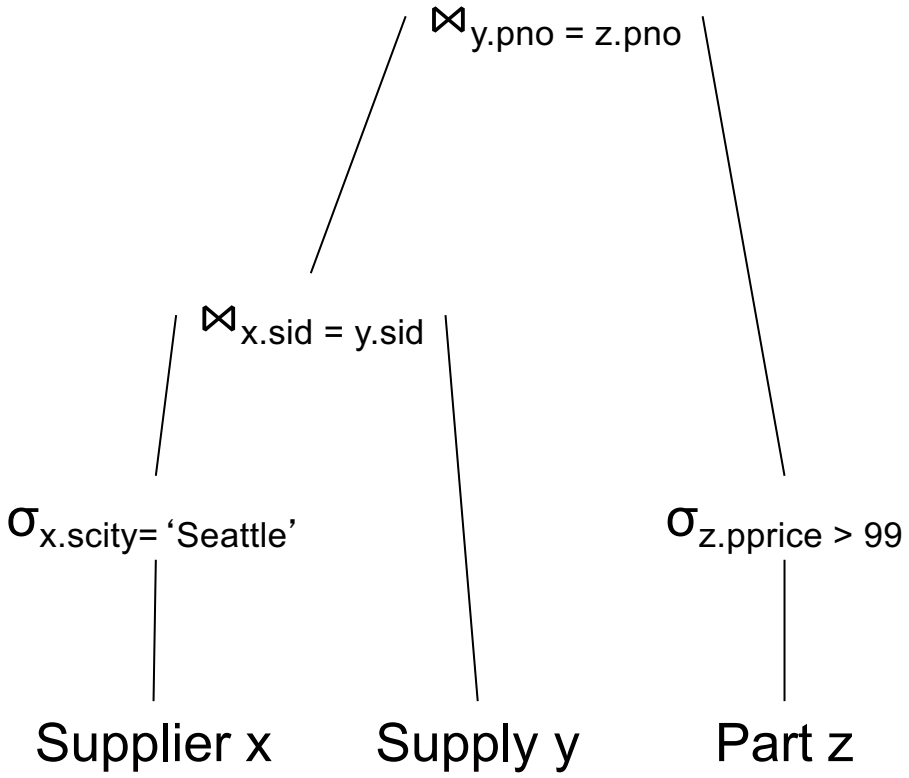


$(R \bowtie S) \bowtie T = R \bowtie (S \bowtie T)$

Also: $R \bowtie S = S \bowtie R$

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

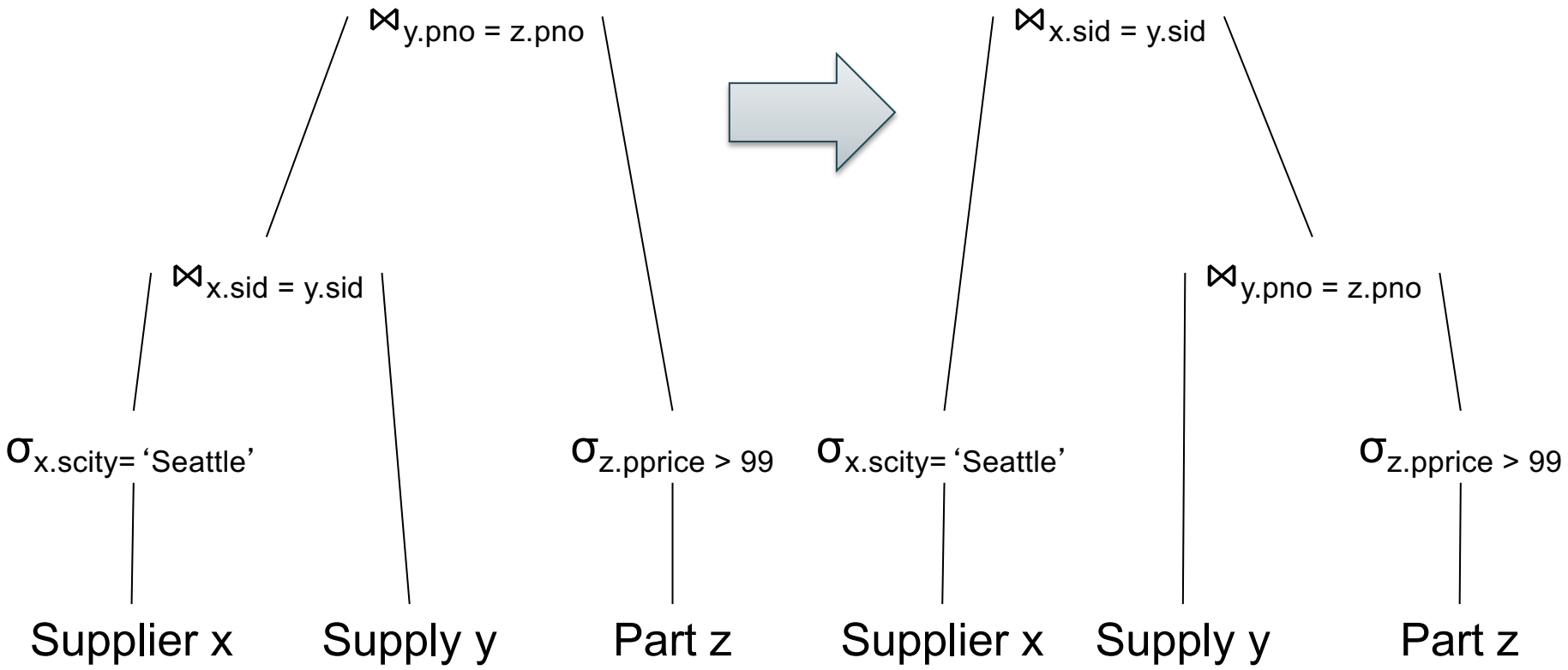
Join Reorder



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

Join Reorder

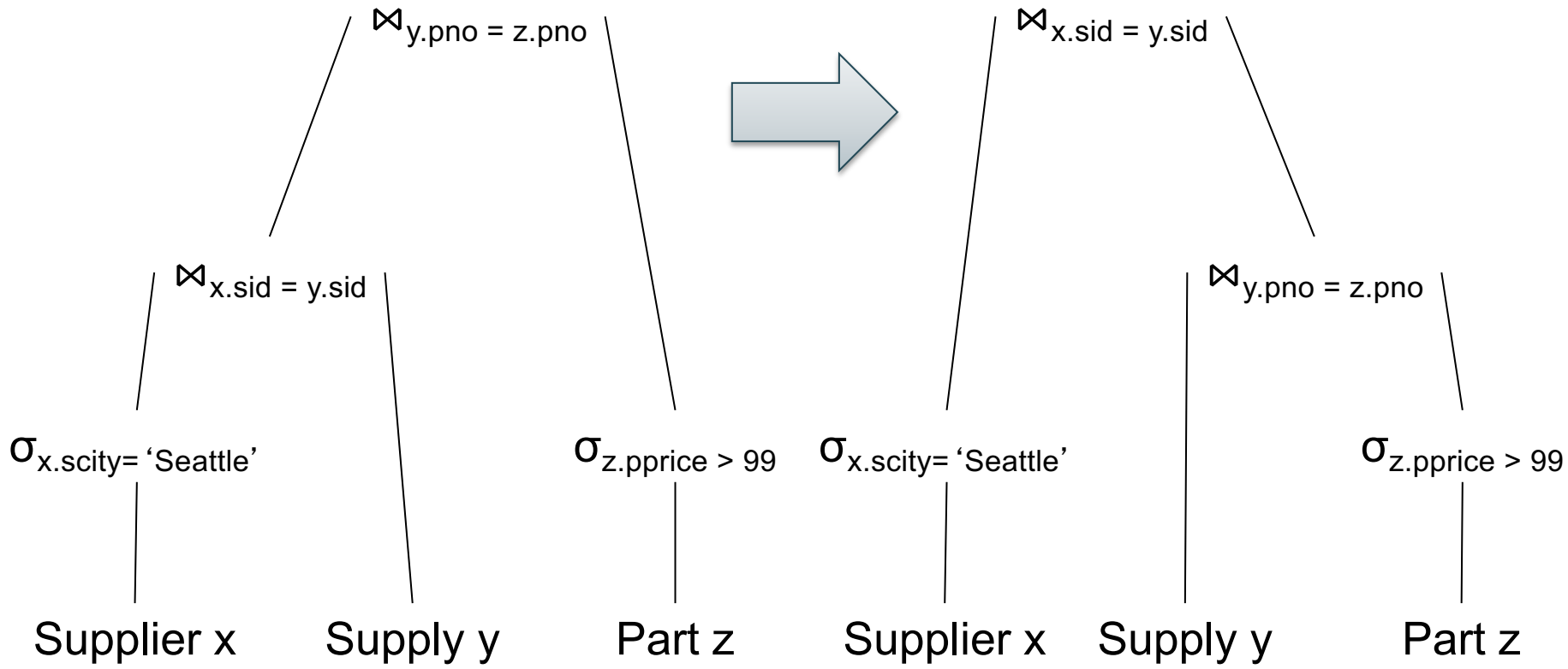
When is one plan better than the other?



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

Join Reorder

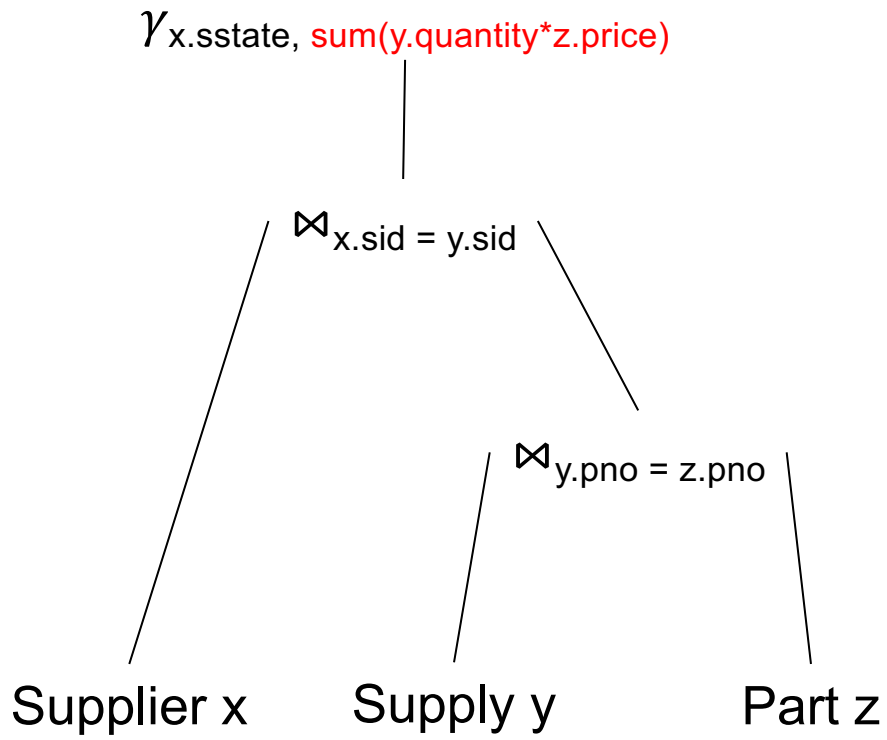
When is one plan better than the other?



Lesson: need sizes of $\sigma_{x.scity= \text{'Seattle'}}$(Supplier), $\sigma_{z.pprice > 99}$(Part)

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

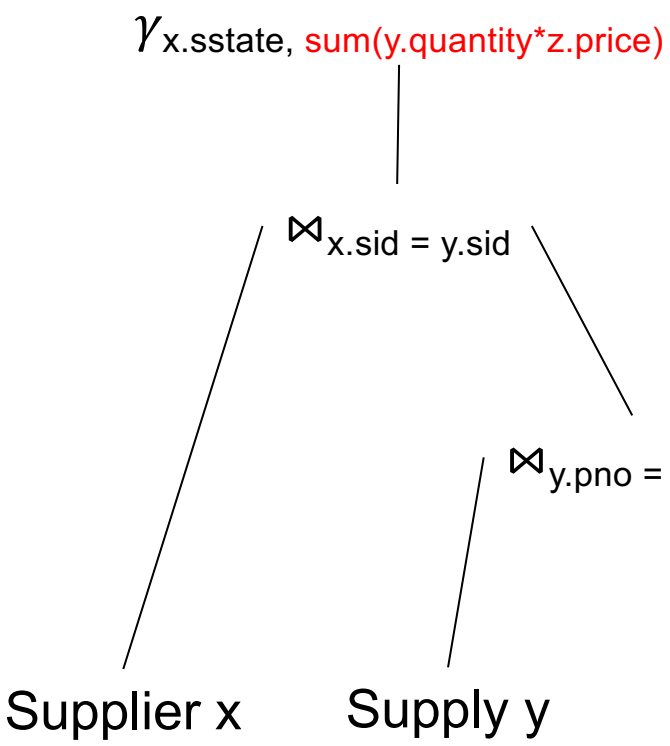
Aggregate Push-down

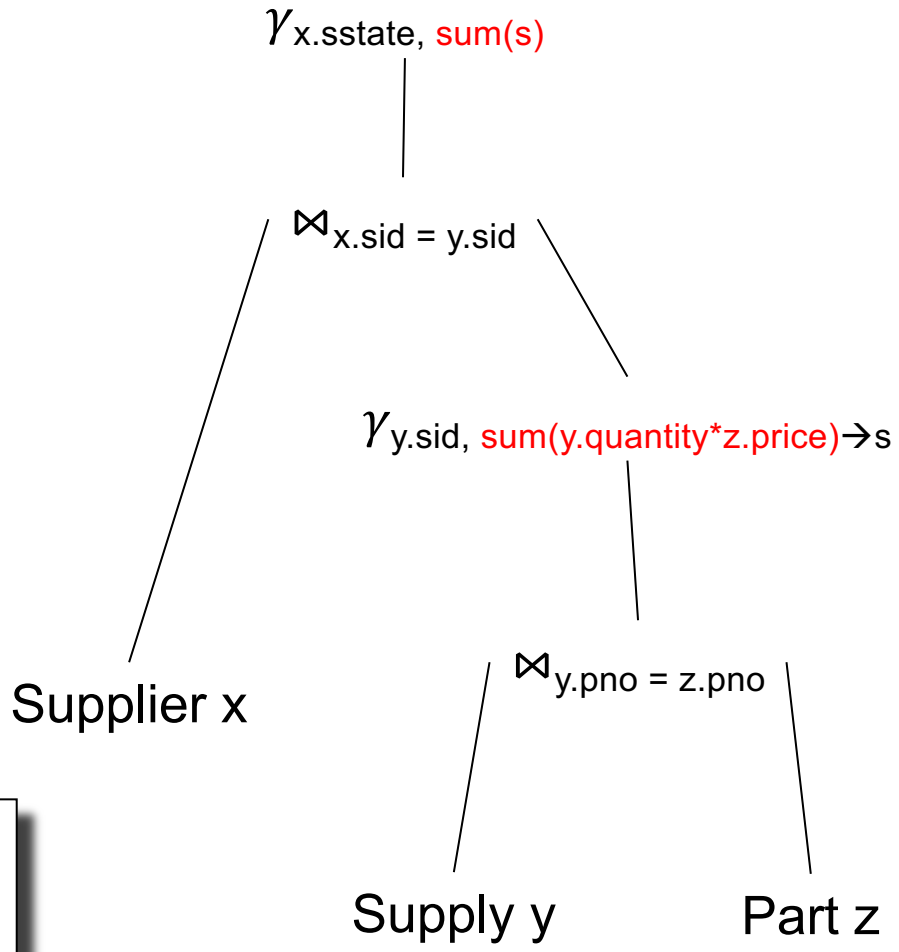



```
SELECT x.sstate, sum(y.quanity*z.price)
FROM Supplier x, Supply y, Part z
WHERE x.sid = y.sid and y.pno = z.pno
GROUP BY x.sstate
```

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

Aggregate Push-down

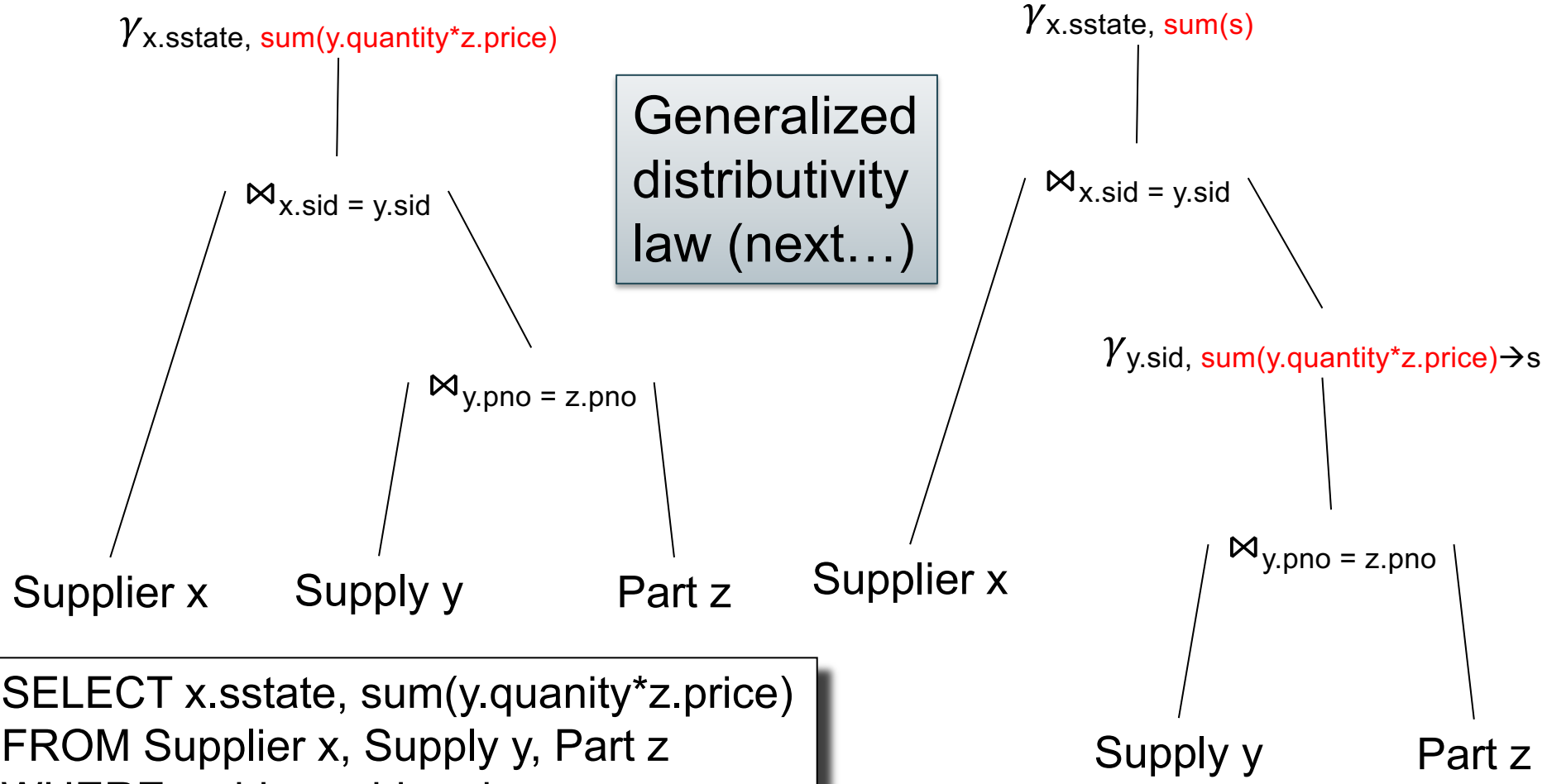





```
SELECT x.sstate, sum(y.quanity*z.price)
FROM Supplier x, Supply y, Part z
WHERE x.sid = y.sid and y.pno = z.pno
GROUP BY x.sstate
```

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

- Is an instance of the *generalized distributivity law* (next)
- Motivation: try this in postgres

```
select count(*) from author;
```

Answer: 2652053

Time: 58.974 ms

Aggregate Push-Down

- Is an instance of the *generalized distributivity law* (next)
- Motivation: try this in postgres

```
select count(*) from author;
```

Answer: 2652053

Time: 58.974 ms

```
select count(*) from publication;
```

Answer: 5120896

Time: 62.465 ms

Aggregate Push-Down

- Is an instance of the *generalized distributivity law* (next)
- Motivation: try this in postgres

```
select count(*) from author;
```

Answer: 2652053

Time: 58.974 ms

```
select count(*) from publication;
```

Answer: 5120896

Time: 62.465 ms

```
select count(*) from author, publication;
```

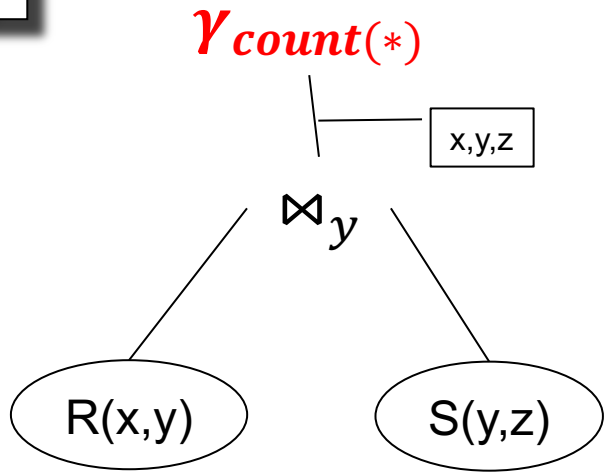
Timeout!!!

Generalized Distributivity Law

```
SELECT count(*) from R, S where R.y=S.y
```

Generalized Distributivity Law

SELECT count(*) from R, S where R.y=S.y



Generalized Distributivity Law

SELECT count(*) from R, S where R.y=S.y

R:

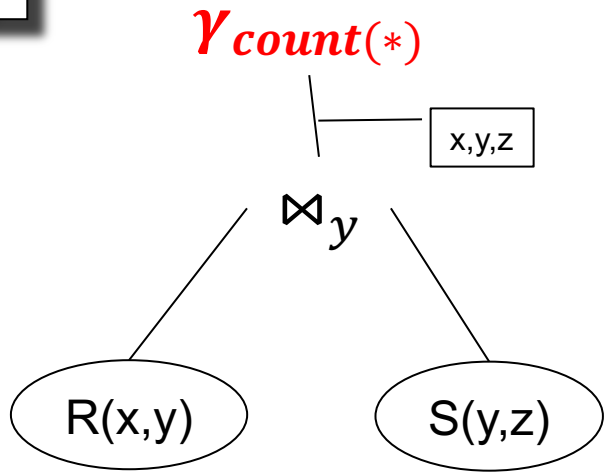
x	y
a	b
c	b
d	f
g	h

S:

y	z
b	g
b	k
h	m

Answer = 5

Runtime = $O(N^2)$



Generalized Distributivity Law

SELECT count(*) from R, S where R.y=S.y

R:

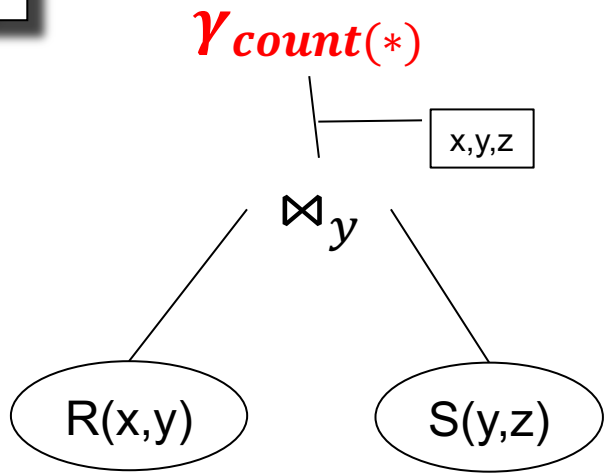
x	y
a	b
c	b
d	f
g	h

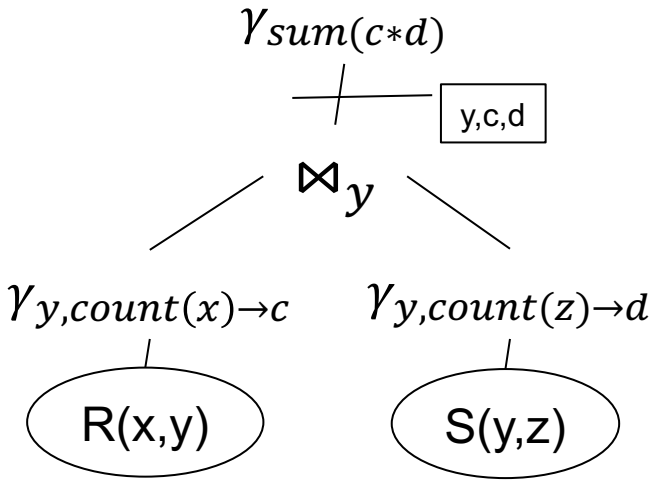
S:

y	z
b	g
b	k
h	m

Answer = 5

Runtime = $O(N^2)$





Generalized Distributivity Law

SELECT count(*) from R, S where R.y=S.y

R:

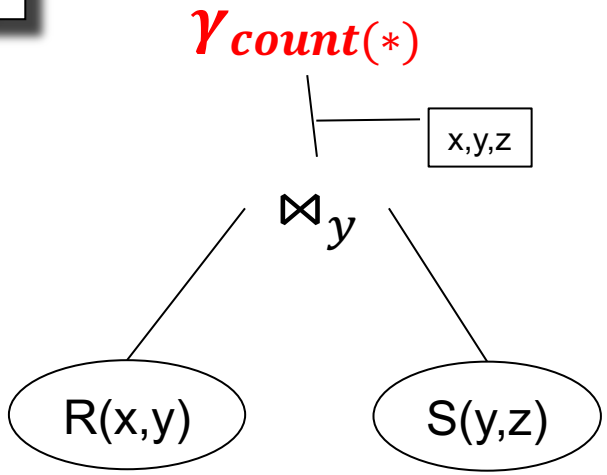
x	y
a	b
c	b
d	f
g	h

S:

y	z
b	g
b	k
h	m

Answer = 5

Runtime = $O(N^2)$



A:

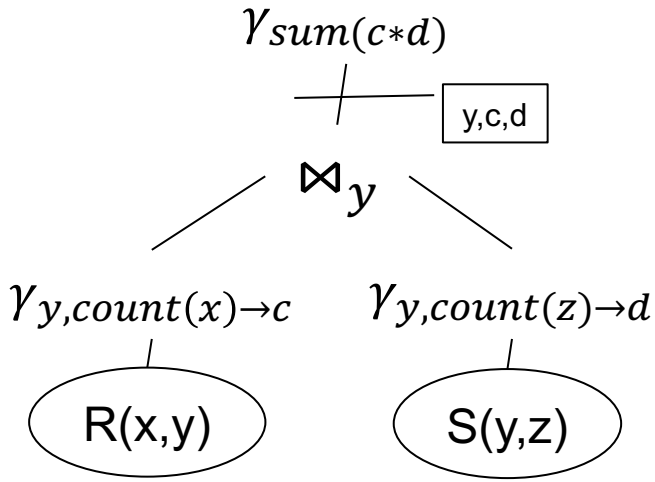
y	c
b	2
f	1
h	1

B:

y	c
b	2
h	1

A⋈B

y	c	d
b	2	2
h	1	1



Generalized Distributivity Law

SELECT count(*) from R, S where R.y=S.y

R:

x	y
a	b
c	b
d	f
g	h

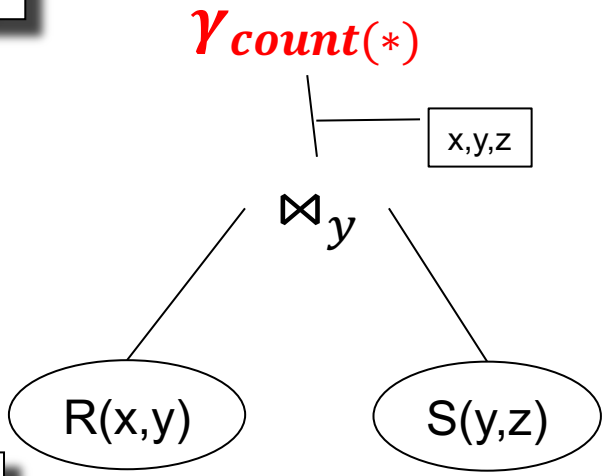
S:

y	z
b	g
b	k
h	m

Answer = 5

Runtime = $O(N^2)$

Runtime = $O(N)$



A=select y,count(*) as c from R group by y

B=select y,count(*) as d from S group by y

select sum(c*d) from A, B where A.y=B.y

A:

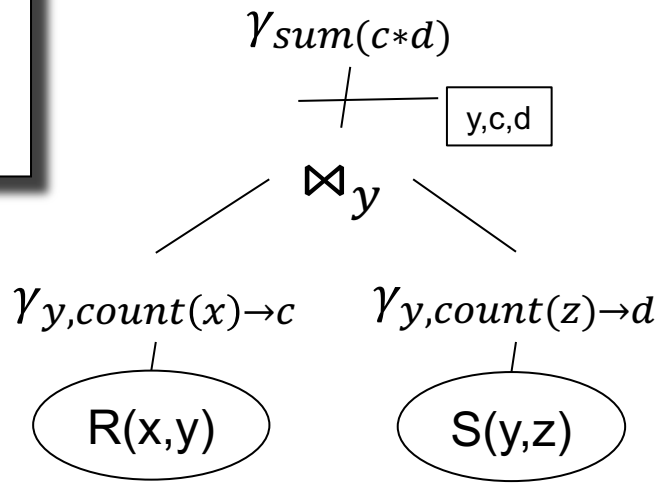
y	c
b	2
f	1
h	1

B:

y	c
b	2
h	1

A⋈B

y	c	d
b	2	2
h	1	1



Cardinality Estimation

Problem: given statistics on base tables and a query, estimate size of the answer

Very difficult, because:

- Need to do it very fast
- Need to use very little memory

Statistics on Base Data

Statistics on base tables

- Number of tuples (cardinality) T(R)
- Number of physical pages B(R)
- Indexes, number of keys in the index V(R,a)
- Histogram on single attribute (1d)
- Histogram on two attributes (2d)

Computed periodically, often using sampling

Assumptions

- Uniformity
- Independence
- Containment of values
- Preservation of values

Selectivity of a Predicate

Selection: size decreases by *selectivity factor* θ

$$T(\sigma_{pred}(R)) = T(R) * \theta_{pred}$$

Uniformity assumption

Selectivity Factors

- A = c /* $\sigma_{A=c}(R)$ */
  - Selectivity = 1/V(R,A)

Selectivity Factors

- A = c /* $\sigma_{A=c}(R)$ */
  - Selectivity = 1/V(R,A)
- c1 < A < c2 /* $\sigma_{c1<A<c2}(R)$*/
  - Selectivity = (c2 – c1)/(max(R,A) - min(R,A))

Selectivity Factors

- A = c /* $\sigma_{A=c}(R)$ */
  - Selectivity = 1/V(R,A)

- c1 < A < c2 /* $\sigma_{c1<A<c2}(R)$*/
  - Selectivity = (c2 – c1)/(max(R,A) - min(R,A))

Multiple predicates:

- A = c and B = d /* $\sigma_{A=c \text{ and } B=d}(R)$ */

Selectivity Factors

- A = c /* $\sigma_{A=c}(R)$ */
  - Selectivity = 1/V(R,A)

- c1 < A < c2 /* $\sigma_{c1<A<c2}(R)$*/
  - Selectivity = (c2 – c1)/(max(R,A) - min(R,A))

Multiple predicates: *independence assumption*

- A = c and B = d /* $\sigma_{A=c \text{ and } B=d}(R)$ */
  - Selectivity = 1/V(R,A) * 1/V(R,B)

Estimating Result Sizes

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

- Take product of cardinalities of R and S
- Apply this selectivity factor:
  1/ (MAX(V(R,A), V(S,B))
- Why? Will explain next...

Assumptions

Containment of values: if V(R,A) ≤ V(S,B), then the set of A values of R is included in the set of B values of S

- Note: this indeed holds when A is a foreign key in R, and B is a key in S

Selectivity of $R \bowtie_{A=B} S$

Assume $V(R,A) \leq V(S,B)$

- Each tuple t in R joins with $T(S)/V(S,B)$ tuples in S

- Hence $T(R \bowtie_{A=B} S) = T(R)\ T(S) / V(S,B)$

In general:

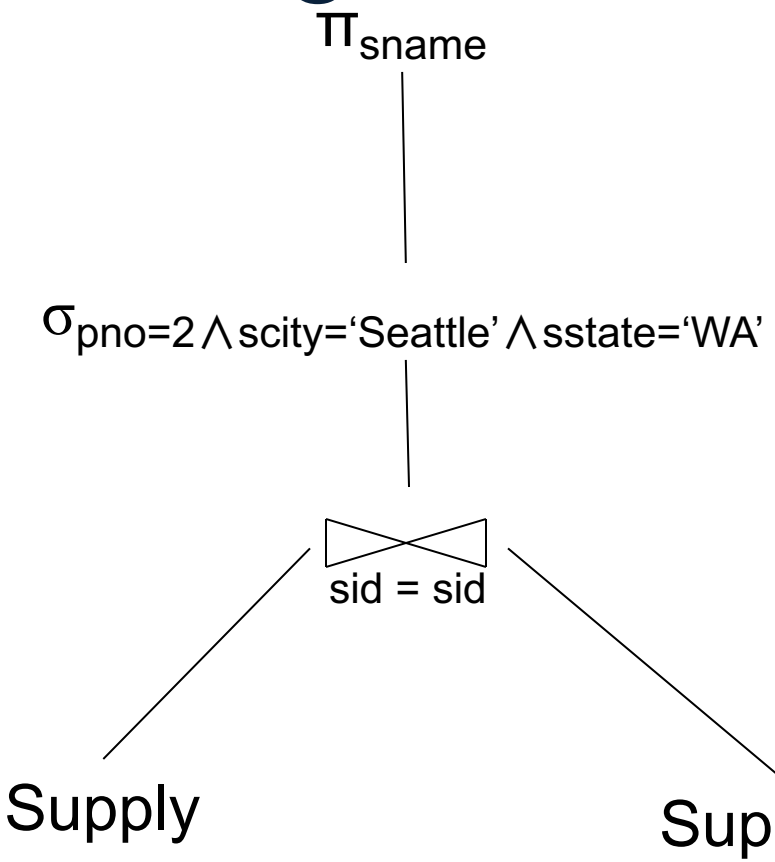
$T(R \bowtie_{A=B} S) = T(R)\ T(S) / \max(V(R,A),V(S,B))$

Computing the Cost of a Plan

- Estimate cardinality in a bottom-up fashion
  - Cardinality is the size of a relation (nb of tuples)
  - Compute size of *all* intermediate relations in plan
- Estimate cost by using the estimated cardinalities
- Extensive example next...

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

Logical Query Plan 1




```
SELECT sname
FROM Supplier x, Supply y
WHERE x.sid = y.sid
    and y.pno = 2
    and x.scity = 'Seattle'
    and x.sstate = 'WA'
```

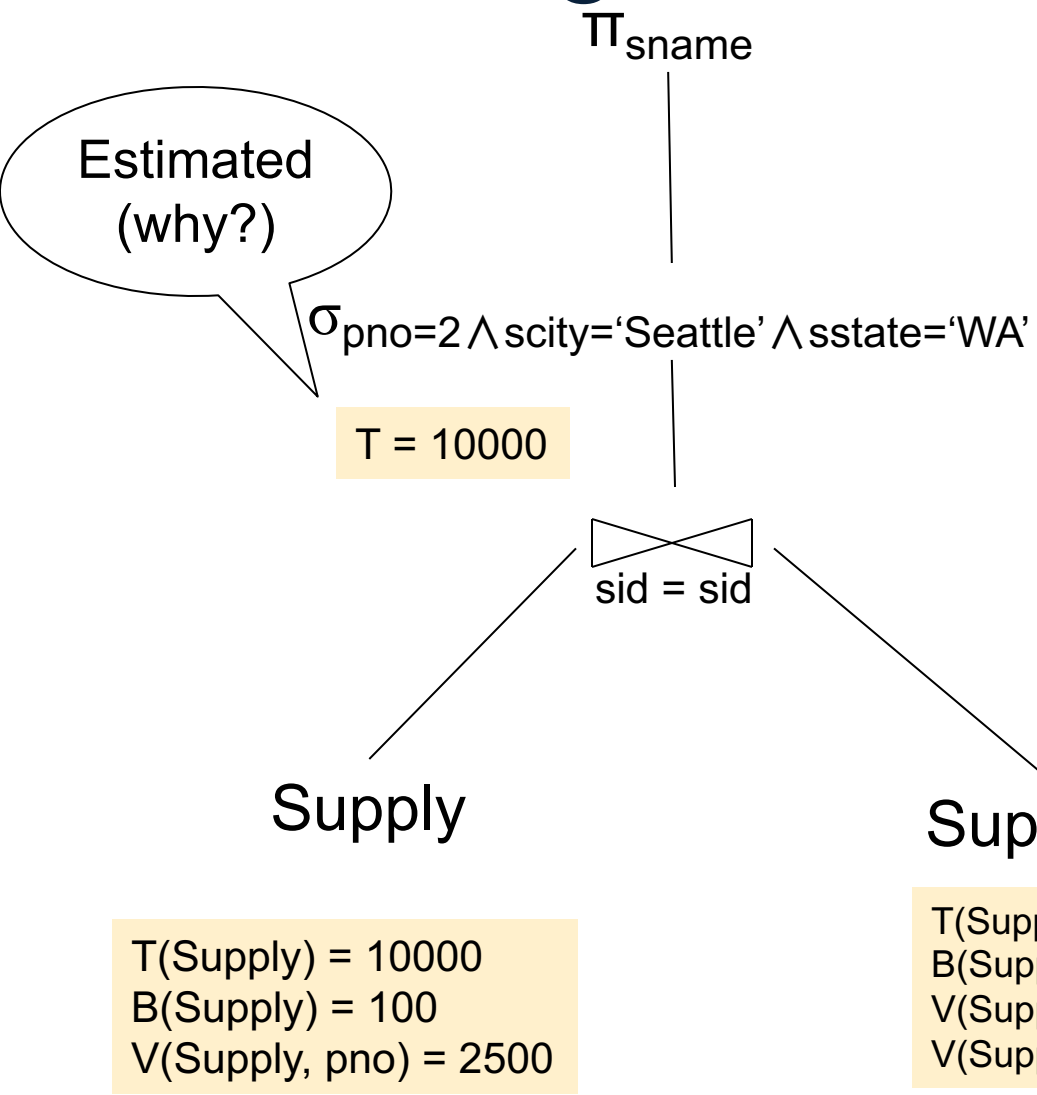
T(Supply) = 10000
B(Supply) = 100
V(Supply, pno) = 2500

T(Supplier) = 1000
B(Supplier) = 100
V(Supplier, scity) = 20
V(Supplier, state) = 10

M=11

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

Logical Query Plan 1




```
SELECT sname
FROM Supplier x, Supply y
WHERE x.sid = y.sid
    and y.pno = 2
    and x.scity = 'Seattle'
    and x.sstate = 'WA'
```

T(Supply) = 10000
B(Supply) = 100
V(Supply, pno) = 2500

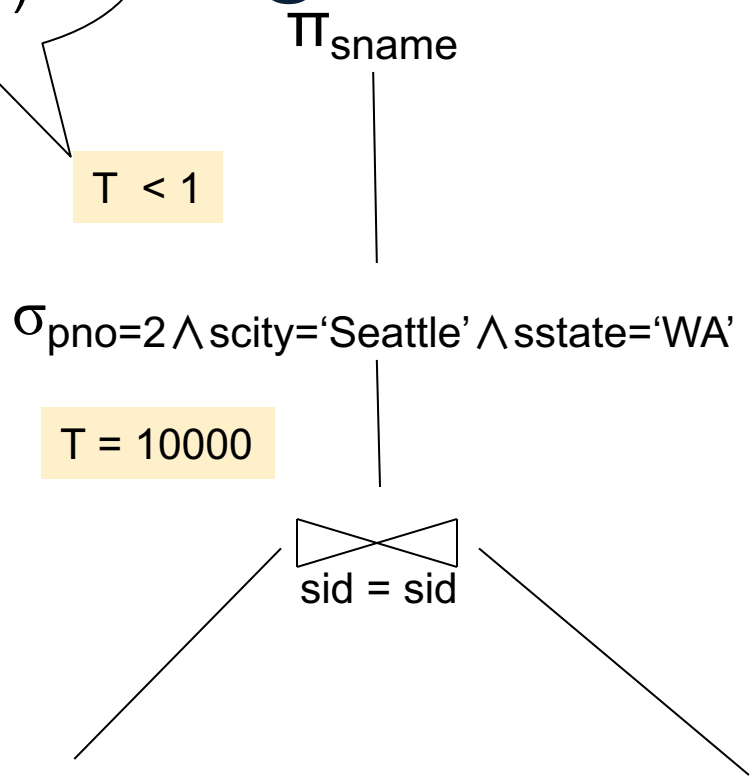
T(Supplier) = 1000
B(Supplier) = 100
V(Supplier, scity) = 20
V(Supplier, state) = 10

M=11

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

Logical Query Plan 1

Estimated (why?)




```
SELECT sname
FROM Supplier x, Supply y
WHERE x.sid = y.sid
    and y.pno = 2
    and x.scity = 'Seattle'
    and x.sstate = 'WA'
```

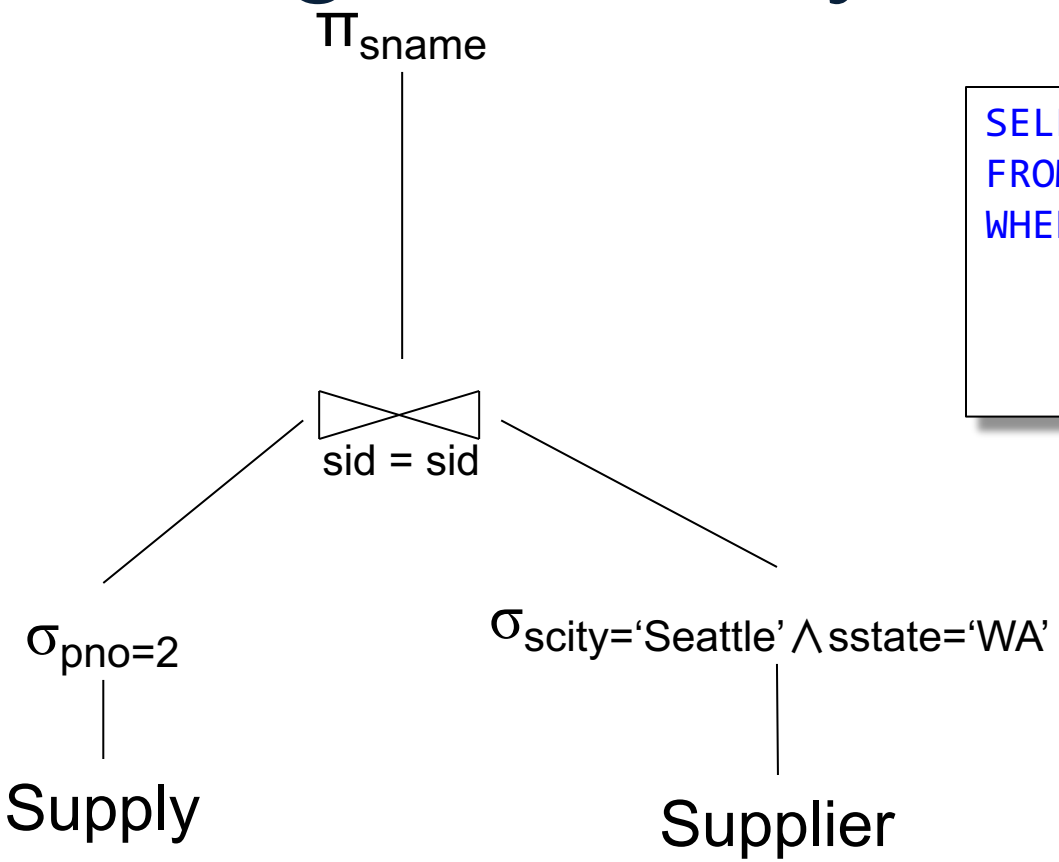
T(Supply) = 10000
B(Supply) = 100
V(Supply, pno) = 2500

T(Supplier) = 1000
B(Supplier) = 100
V(Supplier, scity) = 20
V(Supplier, state) = 10

M=11

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

Logical Query Plan 2




```
SELECT sname
FROM Supplier x, Supply y
WHERE x.sid = y.sid
    and y.pno = 2
    and x.scity = 'Seattle'
    and x.sstate = 'WA'
```

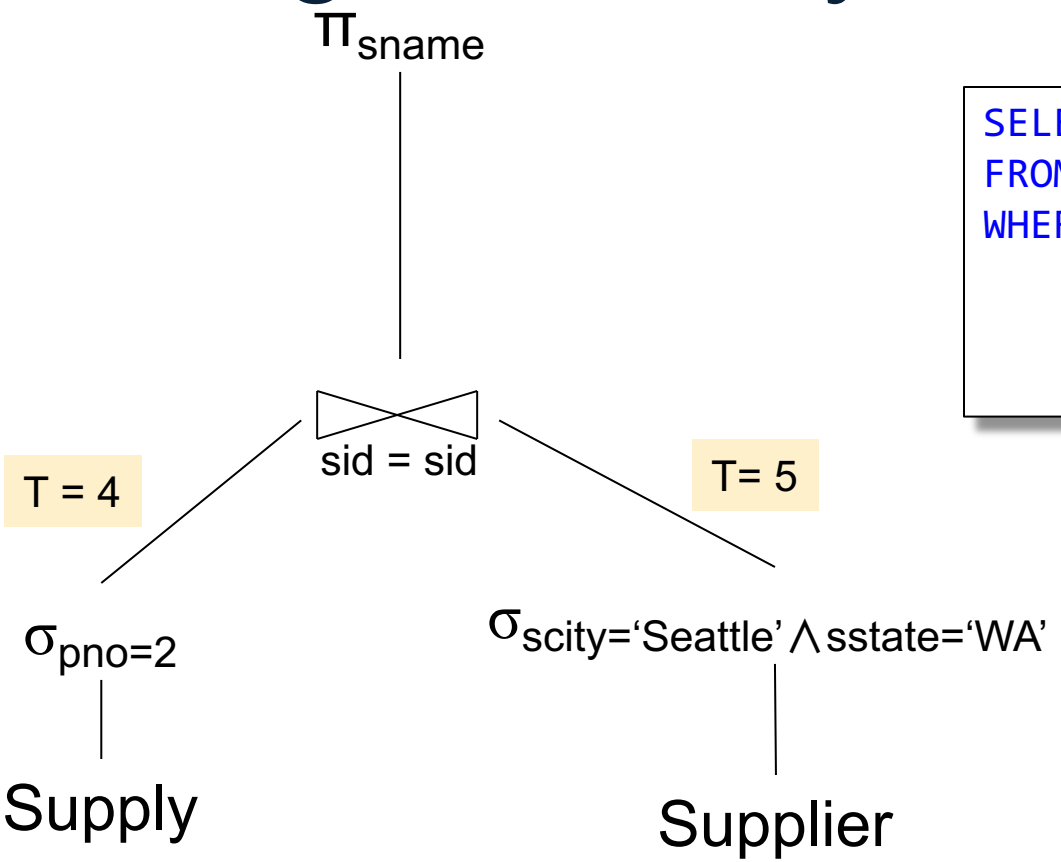
T(Supply) = 10000
B(Supply) = 100
V(Supply, pno) = 2500

T(Supplier) = 1000
B(Supplier) = 100
V(Supplier, scity) = 20
V(Supplier, state) = 10

M=11

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

Logical Query Plan 2




```
SELECT sname
FROM Supplier x, Supply y
WHERE x.sid = y.sid
    and y.pno = 2
    and x.scity = 'Seattle'
    and x.sstate = 'WA'
```

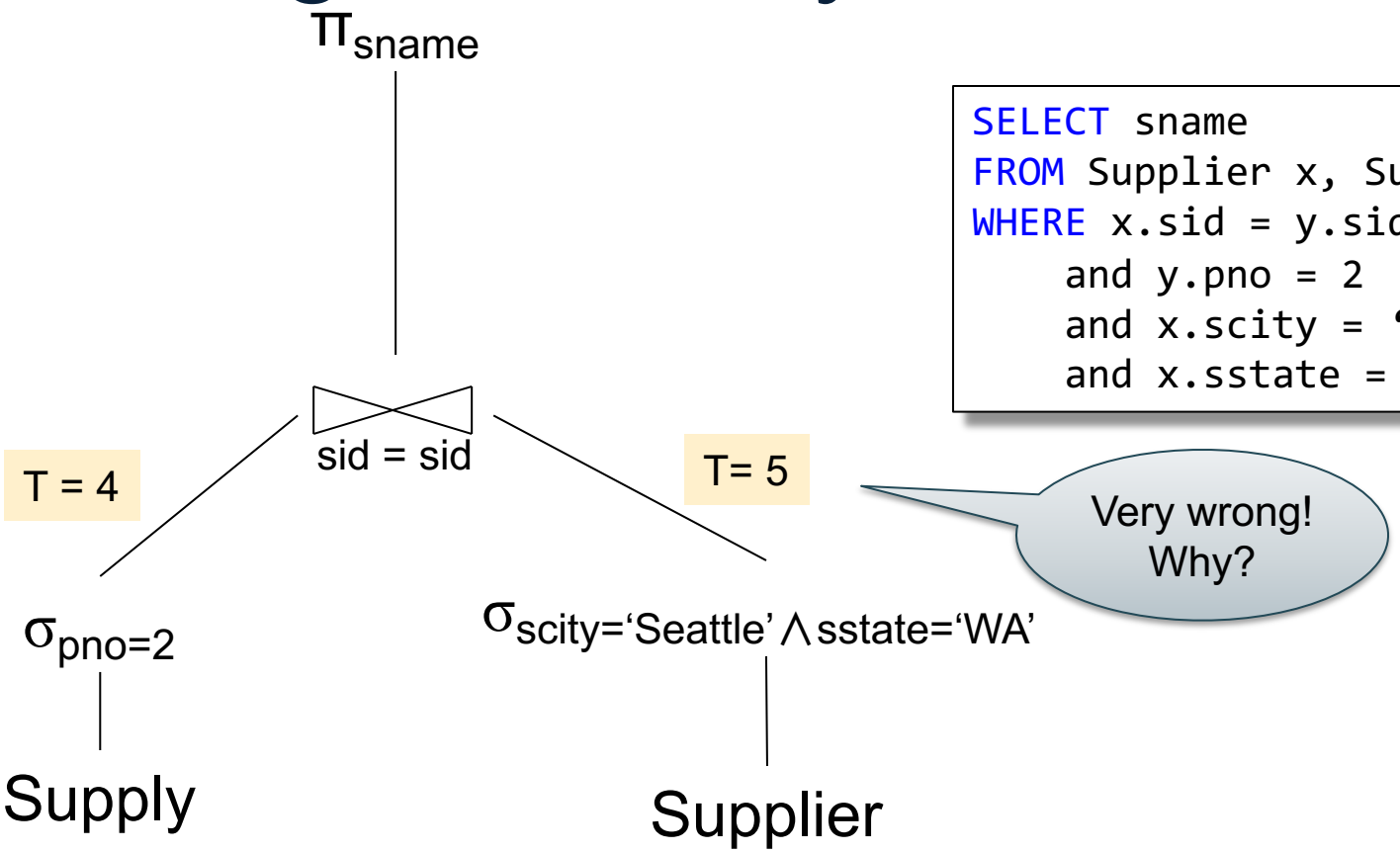
T(Supply) = 10000
B(Supply) = 100
V(Supply, pno) = 2500

T(Supplier) = 1000
B(Supplier) = 100
V(Supplier, scity) = 20
V(Supplier, state) = 10

M=11

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

Logical Query Plan 2




```
SELECT sname
FROM Supplier x, Supply y
WHERE x.sid = y.sid
    and y.pno = 2
    and x.scity = 'Seattle'
    and x.sstate = 'WA'
```

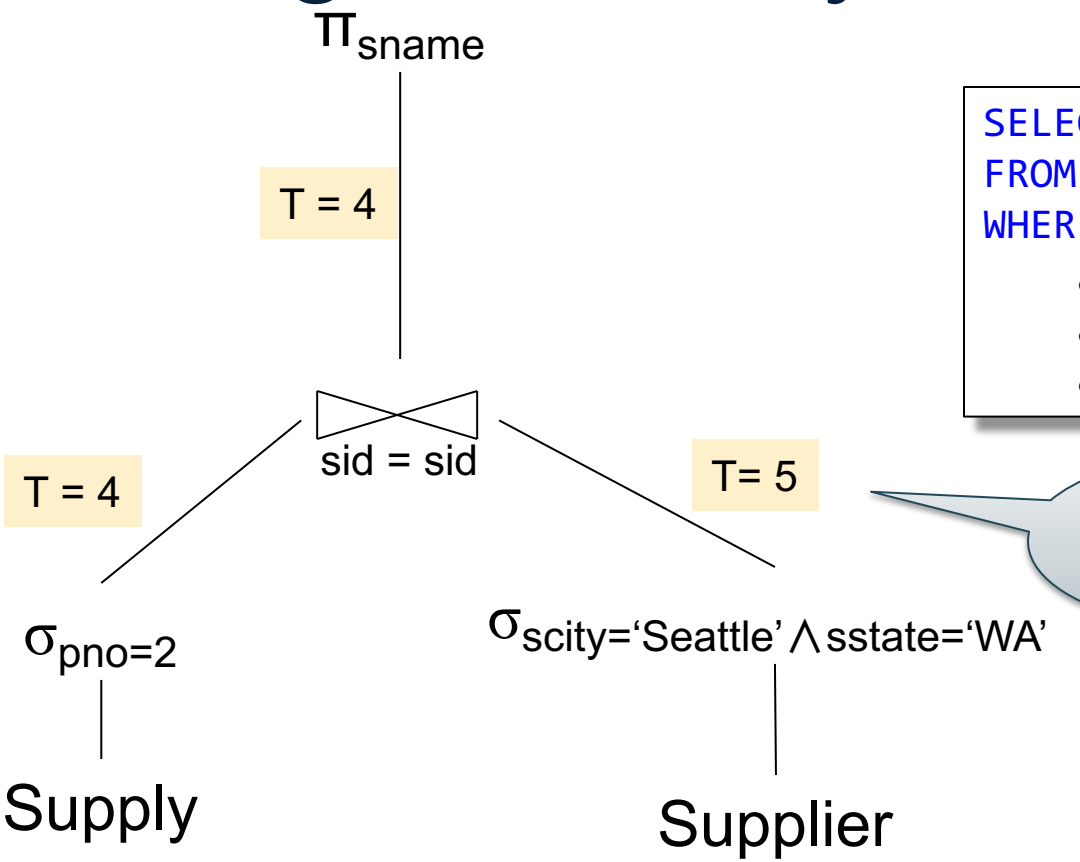
T(Supply) = 10000
B(Supply) = 100
V(Supply, pno) = 2500

T(Supplier) = 1000
B(Supplier) = 100
V(Supplier, scity) = 20
V(Supplier, state) = 10

M=11

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

Logical Query Plan 2




```
SELECT sname
FROM Supplier x, Supply y
WHERE x.sid = y.sid
    and y.pno = 2
    and x.scity = 'Seattle'
    and x.sstate = 'WA'
```

Very wrong!
Why?

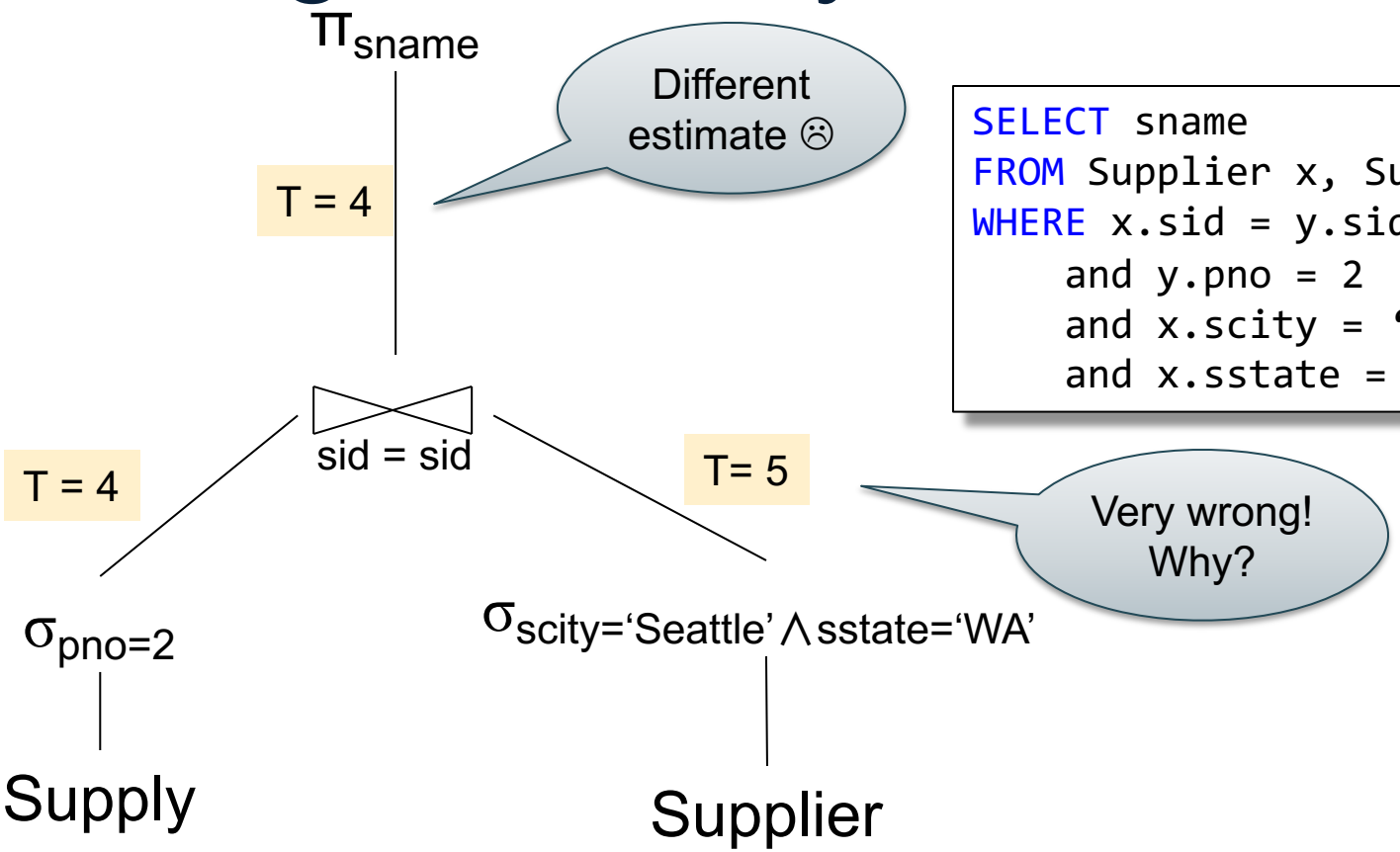
T(Supply) = 10000
B(Supply) = 100
V(Supply, pno) = 2500

T(Supplier) = 1000
B(Supplier) = 100
V(Supplier, scity) = 20
V(Supplier, state) = 10

M=11

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

Logical Query Plan 2




```
SELECT sname
FROM Supplier x, Supply y
WHERE x.sid = y.sid
    and y.pno = 2
    and x.scity = 'Seattle'
    and x.sstate = 'WA'
```

T(Supply) = 10000
B(Supply) = 100
V(Supply, pno) = 2500

T(Supplier) = 1000
B(Supplier) = 100
V(Supplier, scity) = 20
V(Supplier, state) = 10

M=11

Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter A. Boncz, Alfons Kemper, Thomas Neumann: How Good Are Query Optimizers, Really?, VLDB 2015

Joins (0 to 6)

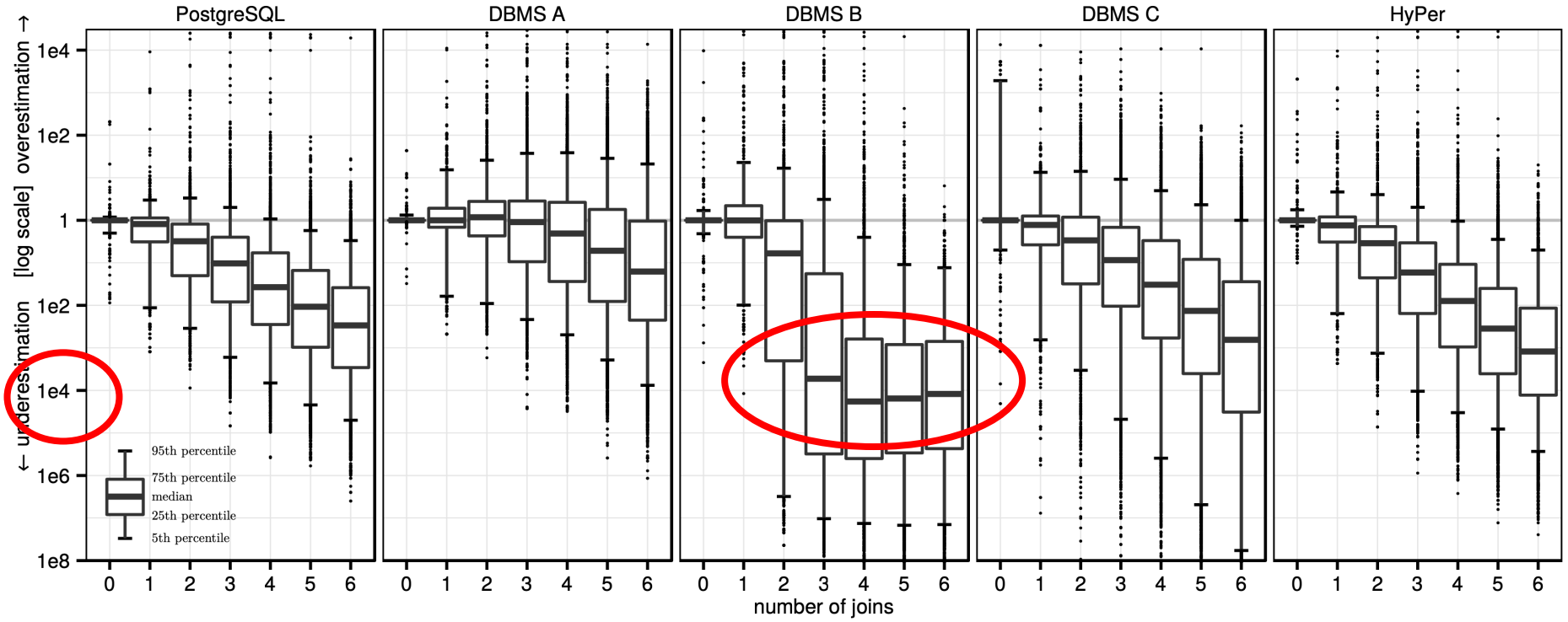


Figure 3: Quality of cardinality estimates for multi-join queries in comparison with the true cardinalities. Each boxplot summarizes the error distribution of all subexpressions with a particular size (over all queries in the workload)

Discussion

Physical data independence recap:

- Users think of the data as a *relations*, SQL is declarative = *what*
- Real data is stored in physical format (on disk, or distributed, etc), real queries are answered by algorithms=*how*
- Query optimizer maps *what* to *how*

Parallel Query Processing

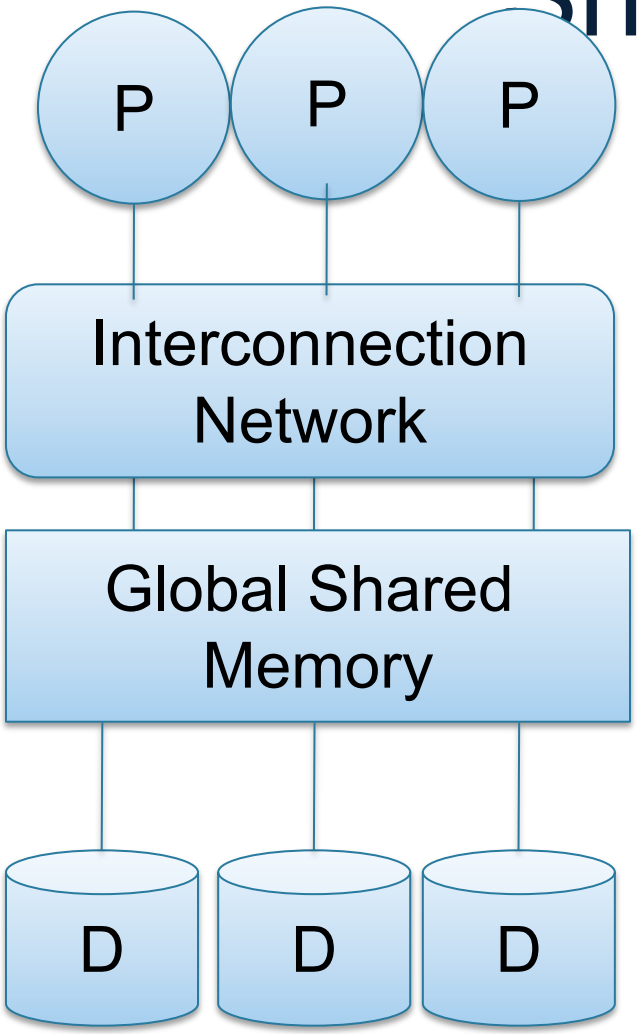
Parallel Query Processing

- Clusters:
  - More servers → more likely to fit data in main memory
  - More servers → more computing power
  - Clusters are now cheaply available in the cloud
  - A.k.a. *distributed* query procesing
- Multicores: the end of Moore's law

Architectures for Parallel Databases

- Shared memory
- Shared disk
- Shared nothing

Shared Memory

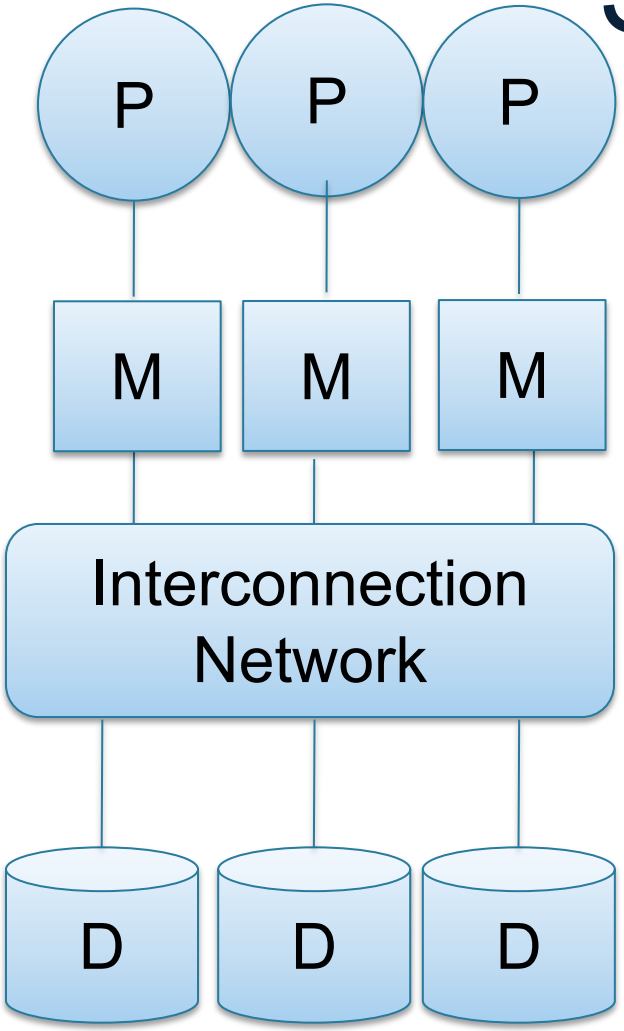


- SMP = symmetric multiprocessor
- Nodes share RAM and disk
- 10x … 100x processors

- Example: SQL Server runs on a single machine and can leverage many threads to speed up a query

- Easy to use and program
- Expensive to scale

Shared Disk



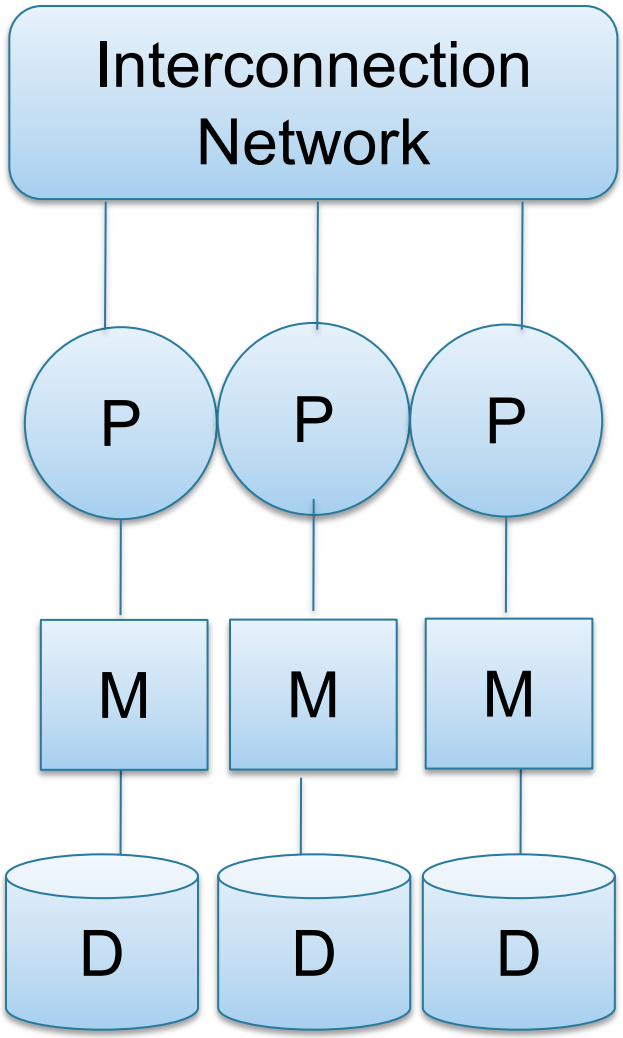
- All nodes access same disks
- 10x processors

- Example: Oracle

- No more memory contention

- Harder to program
- Still hard to scale

Shared Nothing



- Cluster of commodity machines
- Called "clusters" or "blade servers"
- Each machine: own memory&disk
- Up to x1000-x10000 nodes
- Example: redshift, spark, etc, etc

Because all machines today have many cores and many disks, shared-nothing systems typically run many "nodes" on a single physical machine.

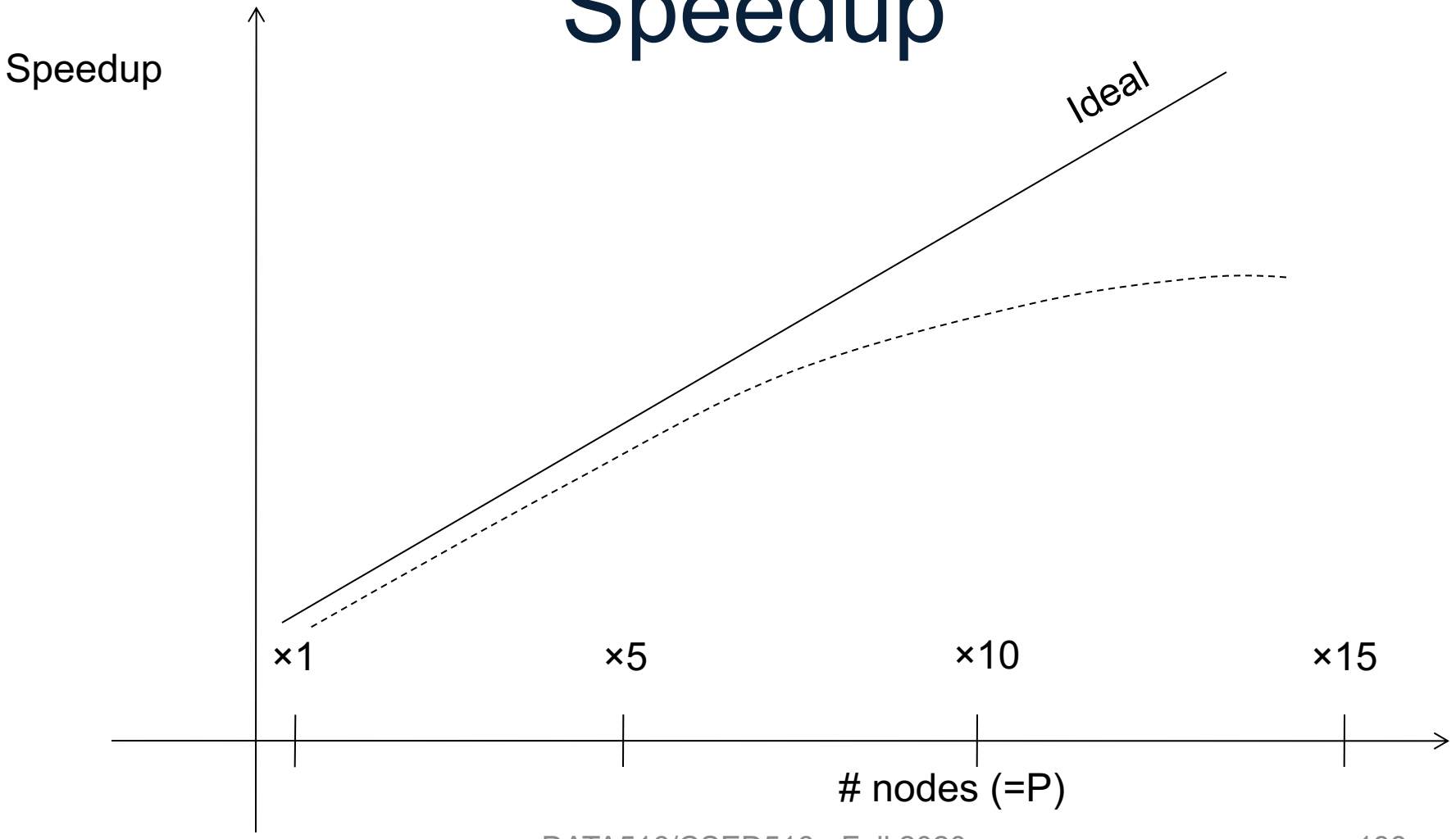
- Easy to maintain and scale
- Most difficult to administer and tune.

Performance Metrics

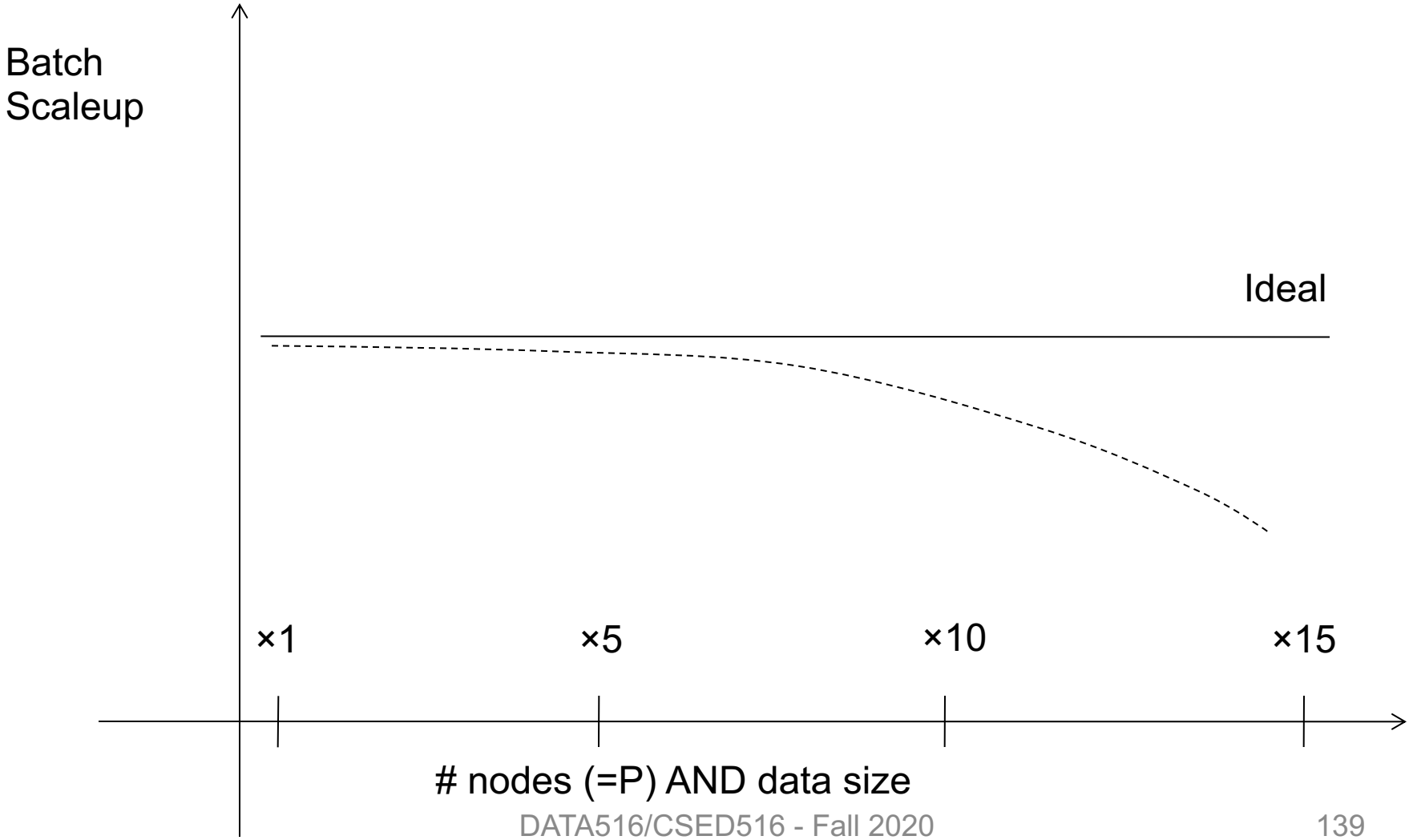
Nodes = processors = computers

- Speedup:
  - More nodes, same data ➔ higher speed
- Scaleup:
  - More nodes, more data ➔ same speed

Linear v.s. Non-linear Speedup



Linear v.s. Non-linear Scaleup



Why Sub-linear?

- Startup cost
  - Cost of starting an operation on many nodes
- Interference
  - Contention for resources between nodes
- Skew
  - Slowest node becomes the bottleneck

Horizontal Data Partitioning

- Distribute the n data on the p servers, such that each server only needs to process n/p data items.

- Called *horizontal data partitioning*

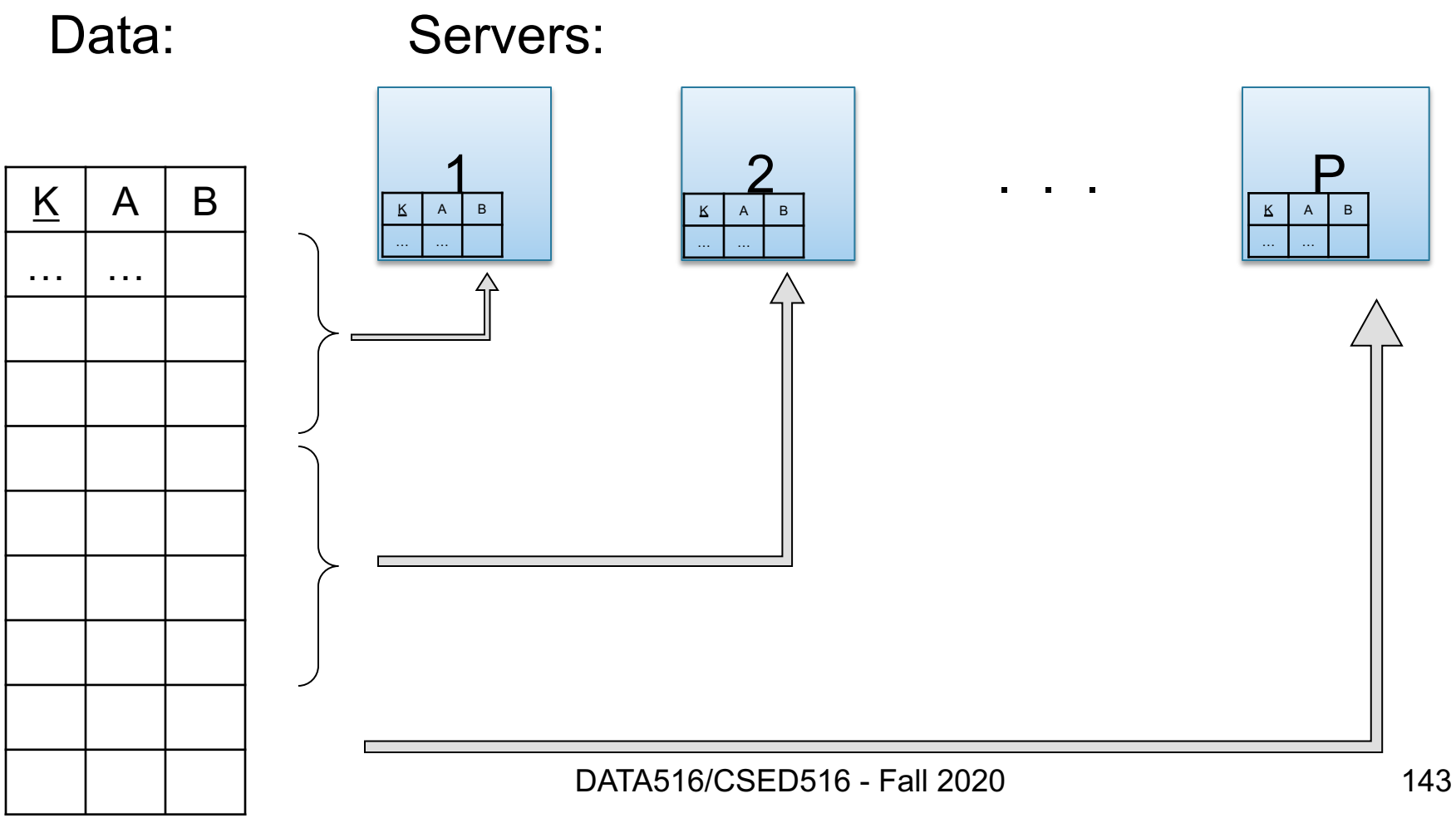
Horizontal Data Partitioning

Data:

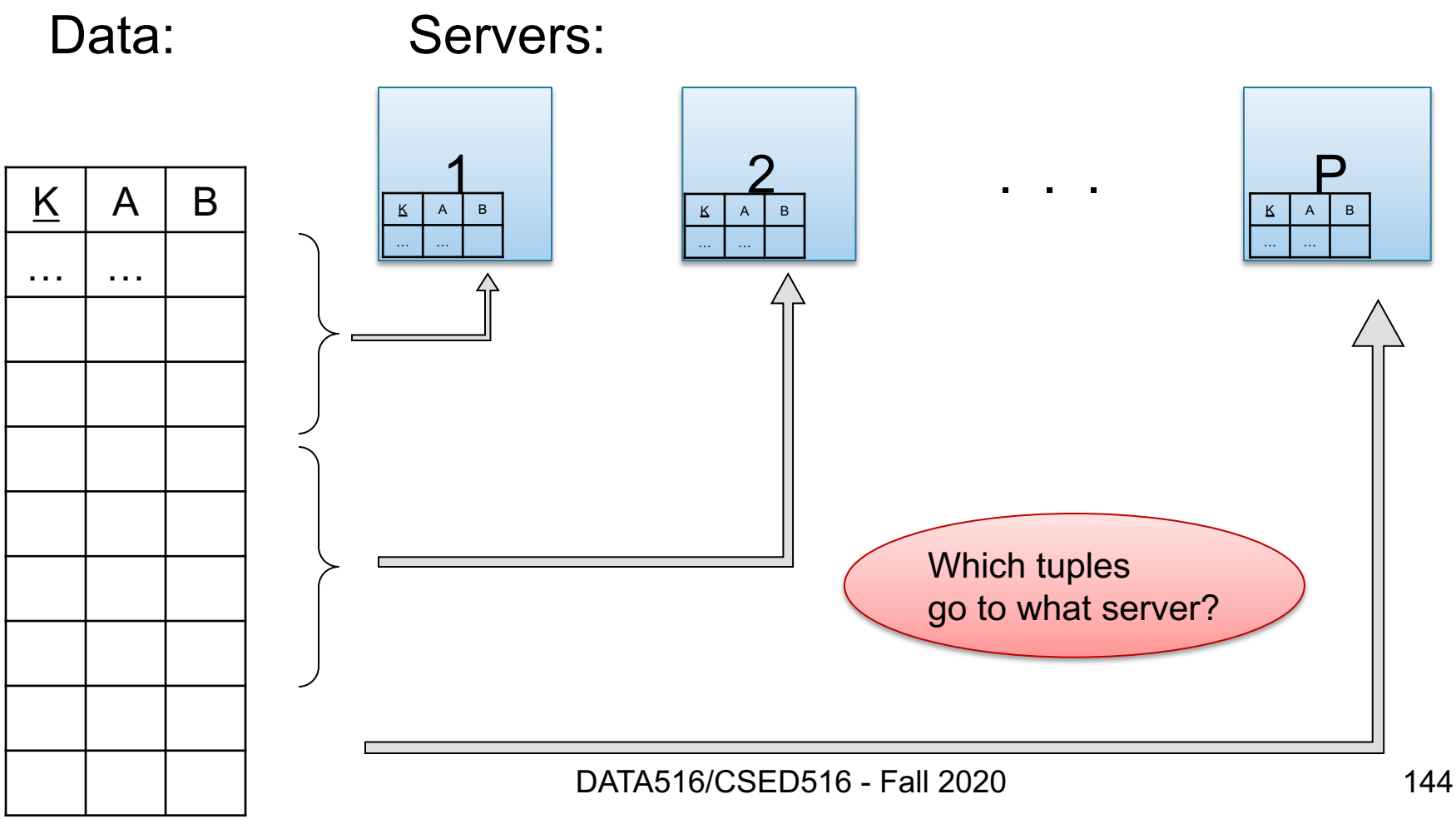
K	A	B
…	…	

Servers:

Horizontal Data Partitioning



Horizontal Data Partitioning



Horizontal Data Partitioning

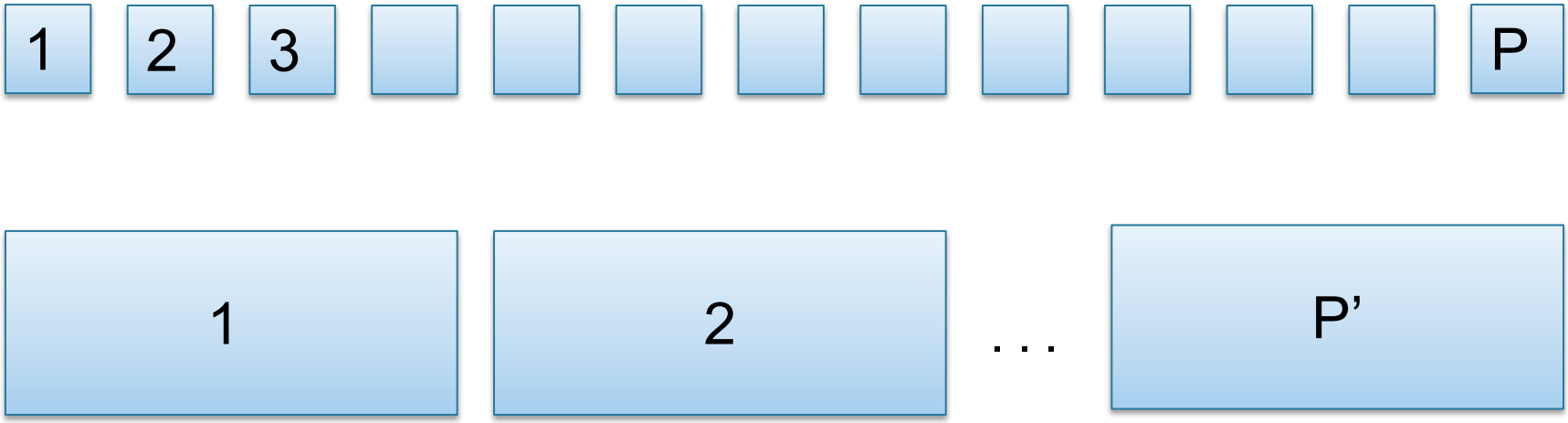
- Block Partition, a.k.a. Round Robin:
  - Partition tuples arbitrarily s.t. size(R_1)≈ … ≈ size(R_P)
- Hash partitioned on attribute A:
  - Tuple t goes to chunk i, where i = h(t.A) mod P + 1
- Range partitioned on attribute A:
  - Partition the range of A into $-\infty = v_0 < v_1 < \ldots < v_P = \infty$
  - Tuple t goes to chunk i, if $v_{i-1} < t.A < v_i$

Brent's Theorem

Suppose we can solve a problem in time T using P servers:

Brent's Theorem

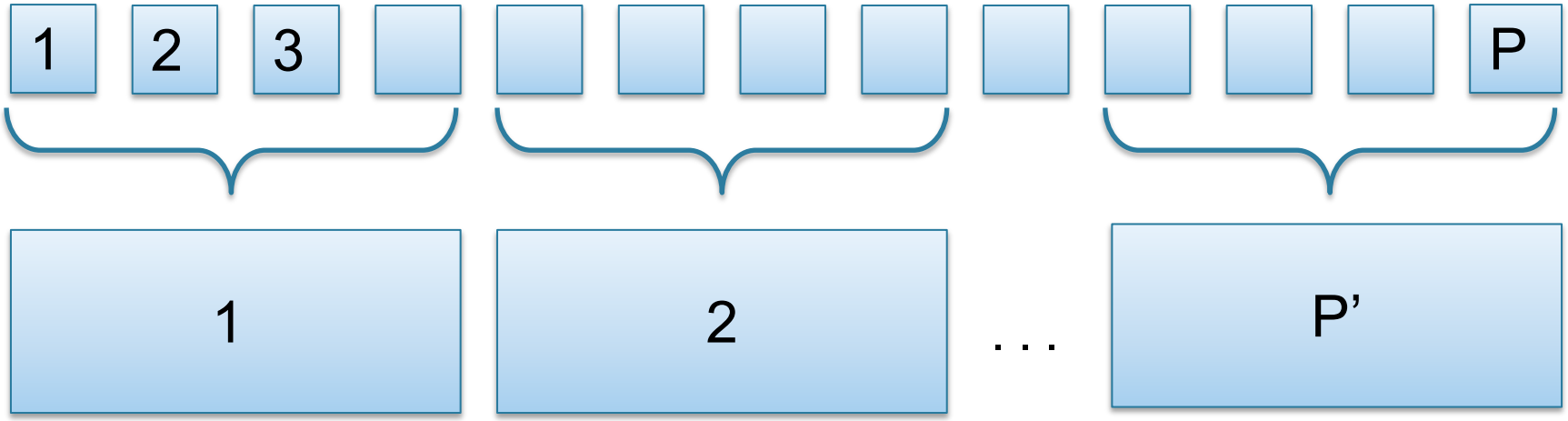
Suppose we can solve a problem in time T using P servers:



Then we can solve the same problem in time T*P/P' using P' < P servers

Brent's Theorem

Suppose we can solve a problem in time T using P servers:



Then we can solve the same problem in time T*P/P' using P' < P servers

Next Lecture

- MapReduce+Spark
  - Big consumers of Brent's theorem
- Algorithm for parallel Query Processing