CSE544
Data Management

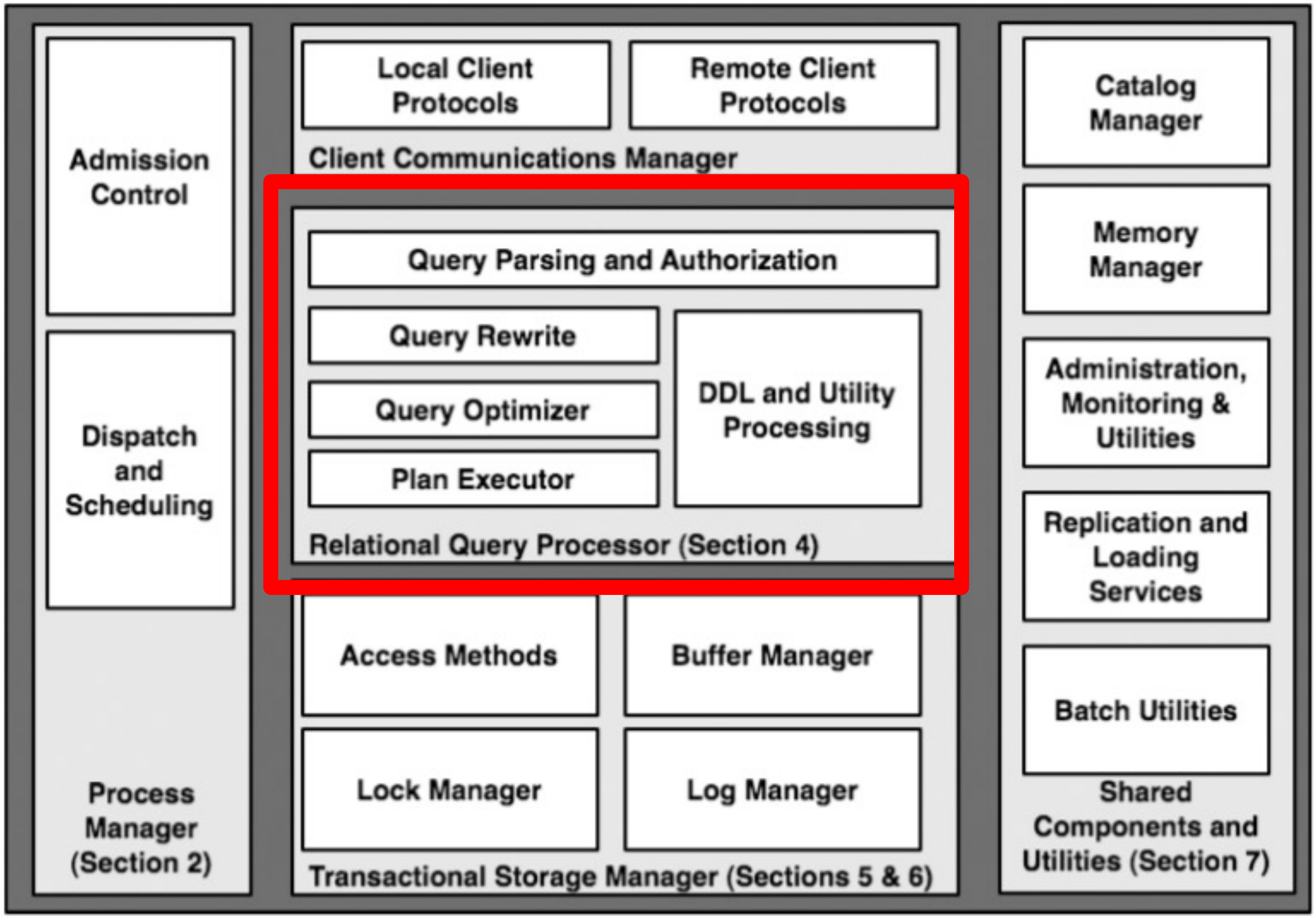
Lectures 7

Query Execution – Part I

Announcements

- HW1 was due on Friday night.
- HW2 is posted, due on Tuesday, 4/27
- Review 4 is due on Wednesday

Query Optimization



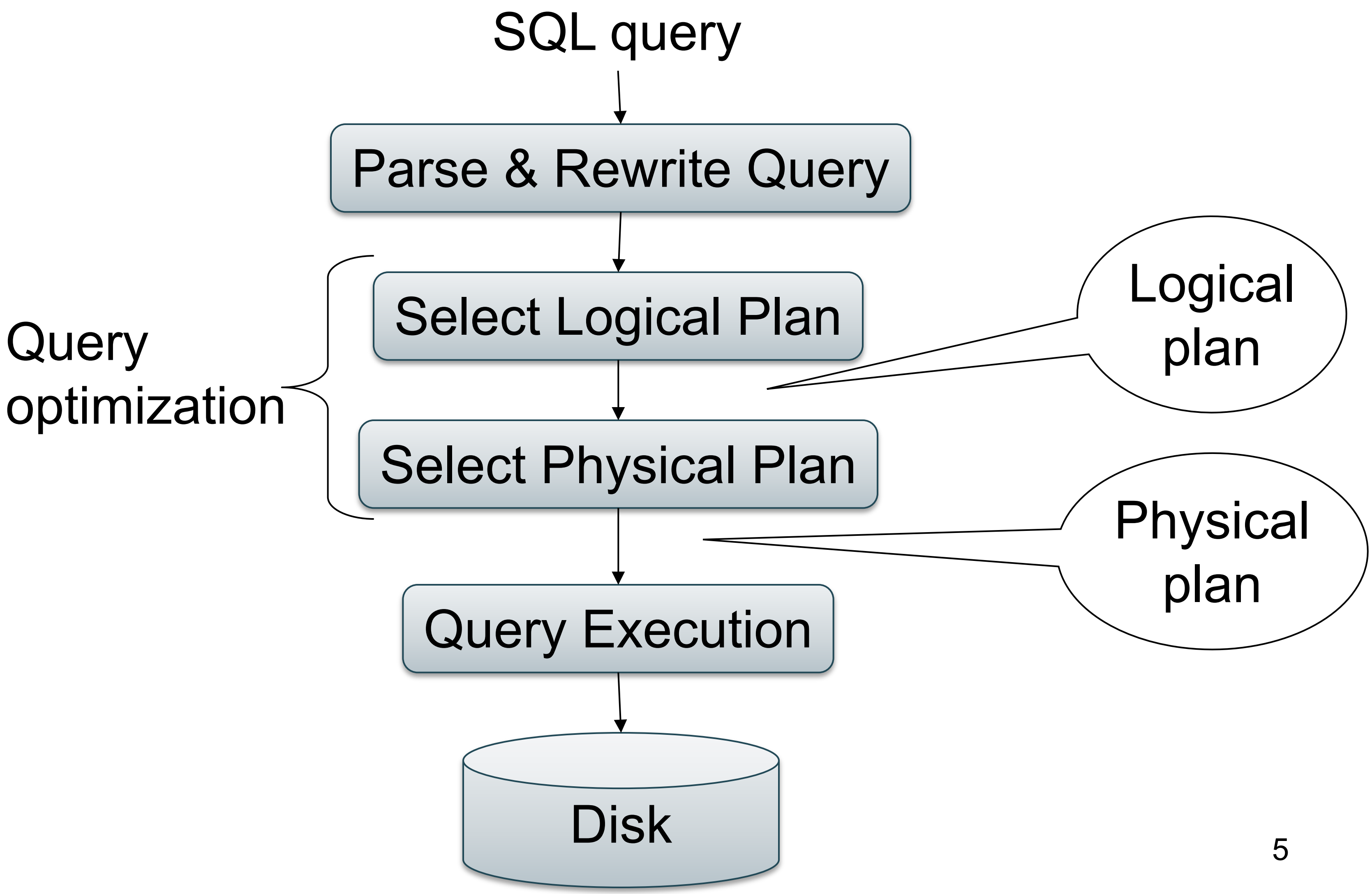
Outline

- Steps involved in processing a query
- Main Memory Operators
- Query execution
- External Memory Operators

Today

Wednesday

Lifecycle of a Query



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

Simple Example

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

Simple Example

View: Suppliers in Seattle

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

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

Added to the schema

NearbySupp(sno, sname)

Simple Example

View: Suppliers in Seattle

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

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

NearbySupp(sno, sname)

Simple Example

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

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

Lifecycle of a Query (1)

- **Step 0: admission control**
  - User connects to the db with username, password
  - User sends query in text format
- **Step 1: Query parsing**
  - Parses query into an internal format
  - Performs various checks using catalog: Correctness, authorization, integrity constraints
- **Step 2: Query rewrite**
  - View rewriting, flattening, decorrelation, etc.

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

NearbySupp(sno, sname)

View Rewriting, Flattening

Original query:

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

View rewriting
 = view inlining
 = view expansion
Flattening
 = unnesting

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

NearbySupp(sno, sname)

View Rewriting, Flattening

Original query:

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

View rewriting
 = view inlining
 = view expansion
Flattening
 = unnesting

Rewritten query:

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

Lifecycle of a Query (2)

- **Step 3: Query optimization**
  - Find an efficient query plan for the query
  - We will spend two lectures on this topic

- A **query plan** is
  - **Logical query plan**: a relational algebra tree
  - **Physical query plan**: add specific algorithms

Five Basic Relational Operators

- Selection: $\sigma_{condition}(S)$
- Projection: $\pi_{list\text{-}of\text{-}attributes}(S)$
- Union ($\cup$)
- Set difference ($-$),
- Cross-product/cartesian product ($\times$),
  Join: $R \bowtie_{\theta} S = \sigma_{\theta}(R \times S)$

Other operators: semi-join, anti-semijoin

Extended Operators of Relational Algebra

- Duplicate elimination (δ)
  - Bag to set
  - Special case of γ
- Group-by/aggregate (γ)
  - Example: $\gamma_{\text{pcolor, max(psize)}\rightarrow\text{m, avg(psize)}\rightarrow\text{s}}(\text{Part})$
  - Min, max, sum, average, count
- Sort operator (τ)

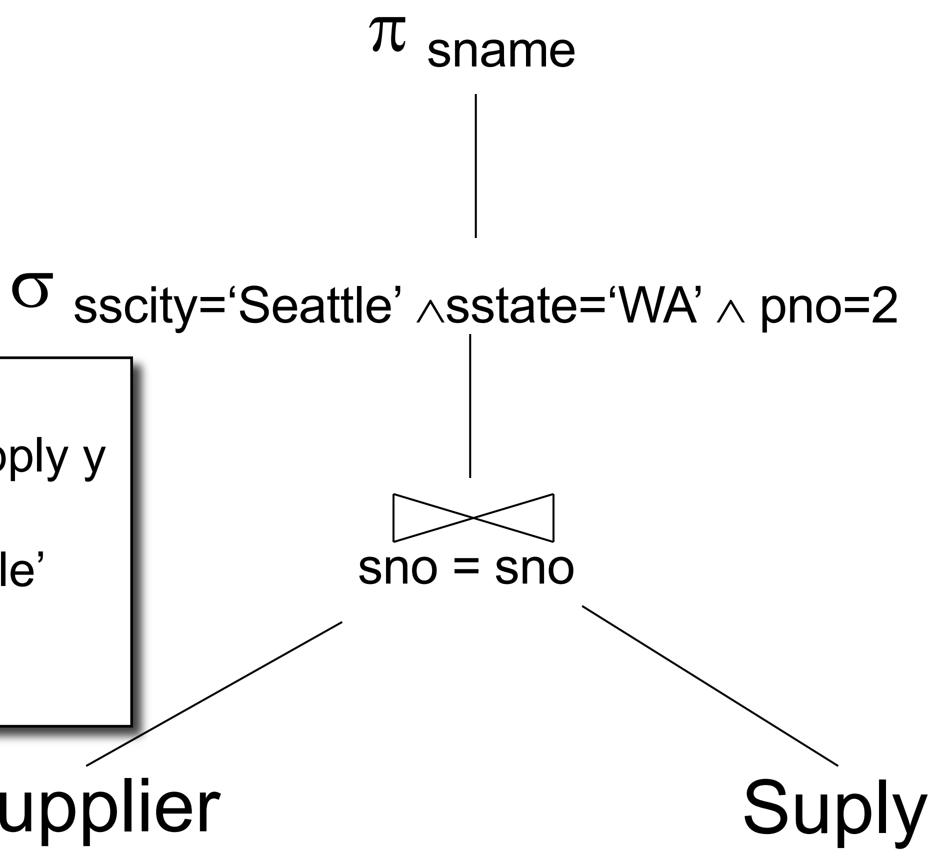
Supplier(sno,sname,scity,sstate)
Supply(sno,pno,price)
Part(pno,pname,psize,pcolor)

Logical Query Plan

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

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

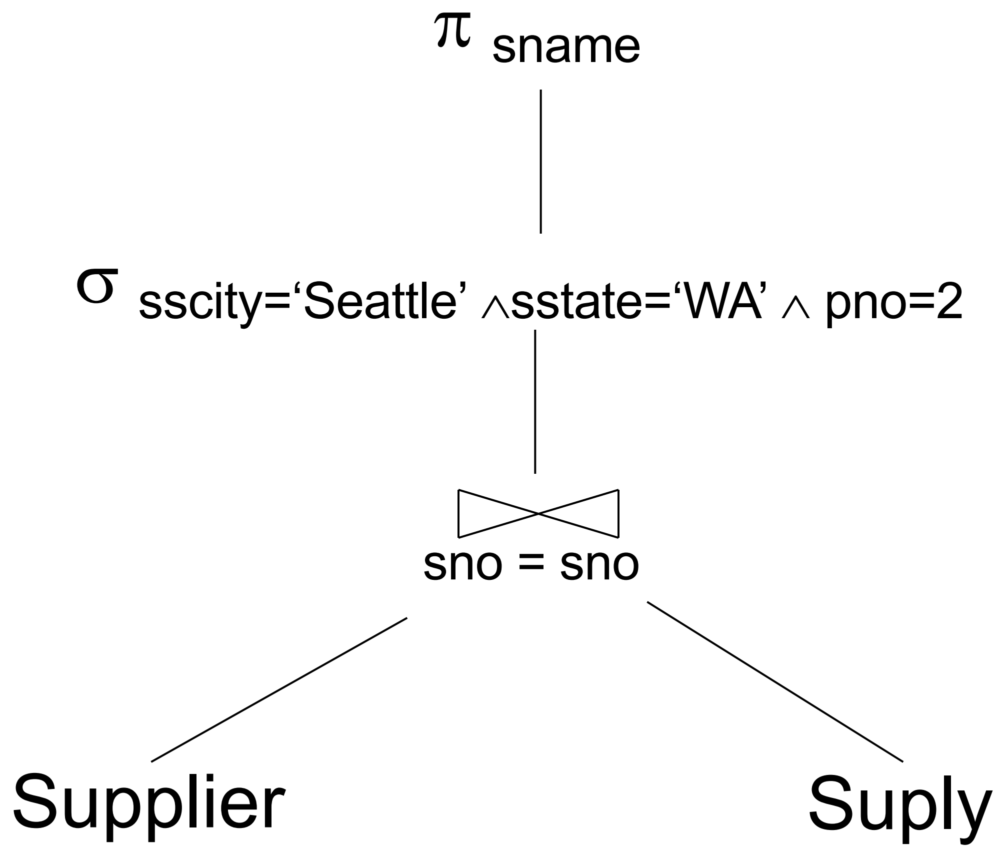
Logical Query Plan




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

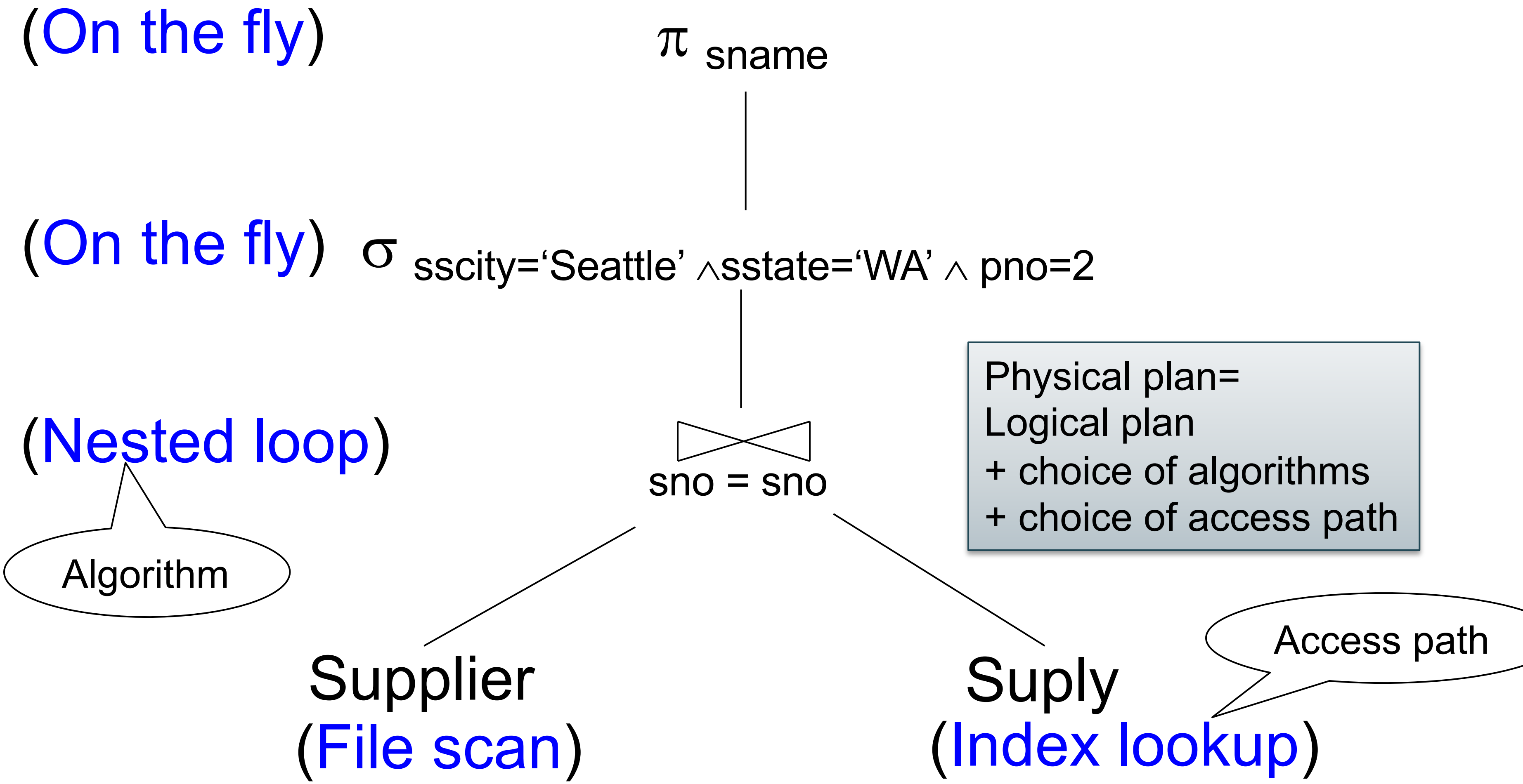
Supplier(sno,sname,scity,sstate)
Supply(sno,pno,price)
Part(pno,pname,psize,pcolor)

Logical Query Plan



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

Physical Query Plan



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

Decorrelation

Find all suppliers in ‘WA’ that supply *only* parts under $100

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

Decorrelation

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

Find all suppliers in 'WA' that supply *only* parts under $100

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

Decorrelation

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

Correlation !

Problem: RA consists of "set-at-at-time" operators.

Cannot express correlated subqueries

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

Decorrelation

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

De-Correlation

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

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

Decorrelation

Un-nesting

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

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

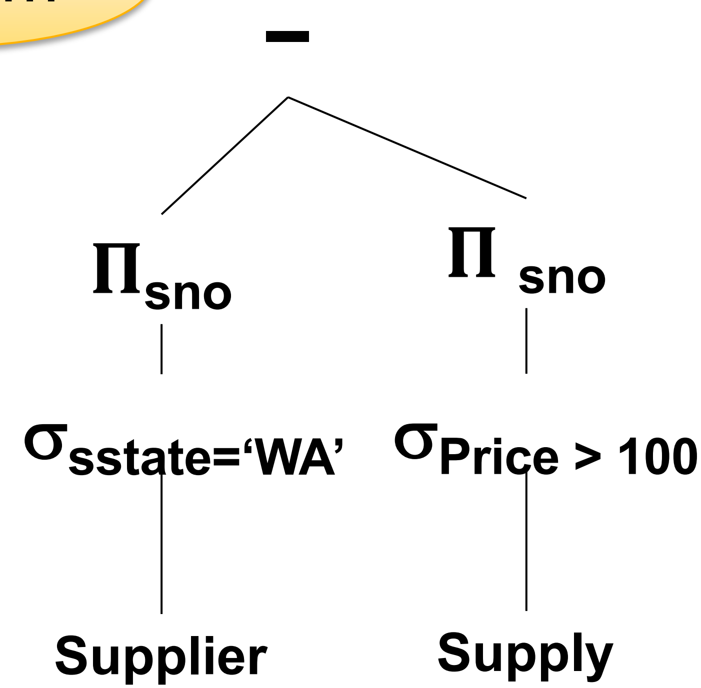
EXCEPT = set difference

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

Decorrelation

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

Finally…



Final Step in Query Processing

- **Step 4: Query execution**
  - Choice of algorithm
  - How to pass data between operators, e.g. materialized, or pipelined

Outline

- Steps involved in processing a query
- Main Memory Operators
- Query execution
- External Memory Operators

Physical Operators

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

```
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?

$O(n^2)$

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

2. Hash Join

Logical operator:

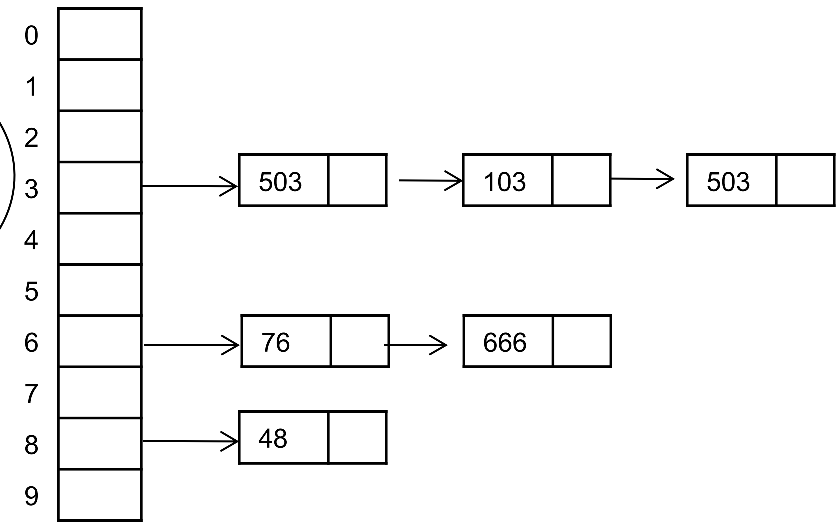
Supply $\bowtie_{sid=sid}$ Supplier

Build phase

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

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

Probe phase



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

2. Hash Join

Logical operator:

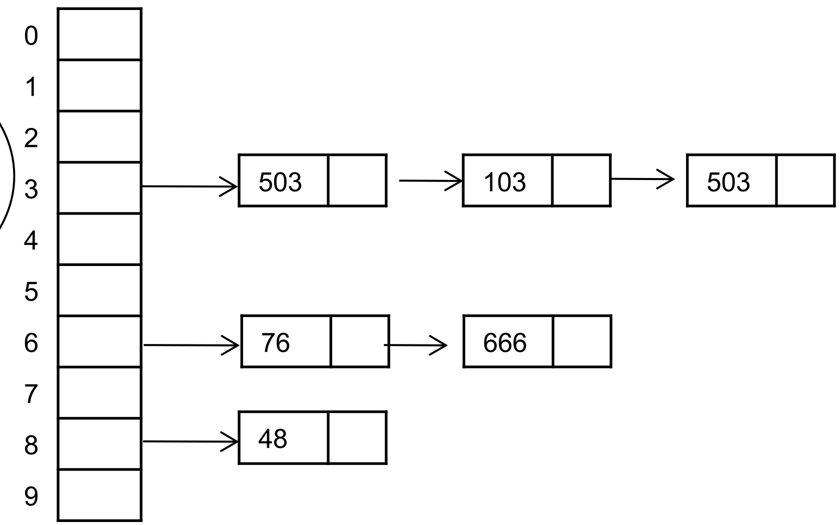
Supply $\bowtie_{sid=sid}$ Supplier

Build phase

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

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

Probe phase



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

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

2. Hash Join

Logical operator:

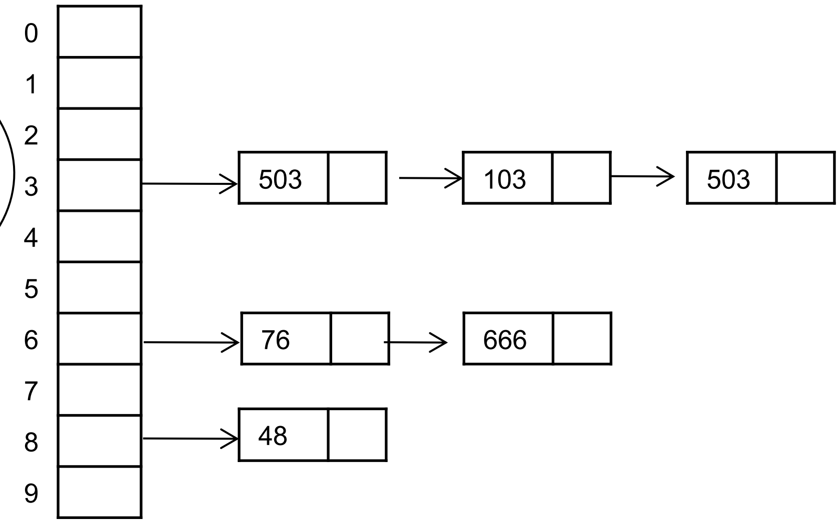
Supply $\bowtie_{sid=sid}$ Supplier

Build phase

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

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

Probe phase



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

O(n)

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

2. Hash Join

Outer table

Inner table

Logical operator:

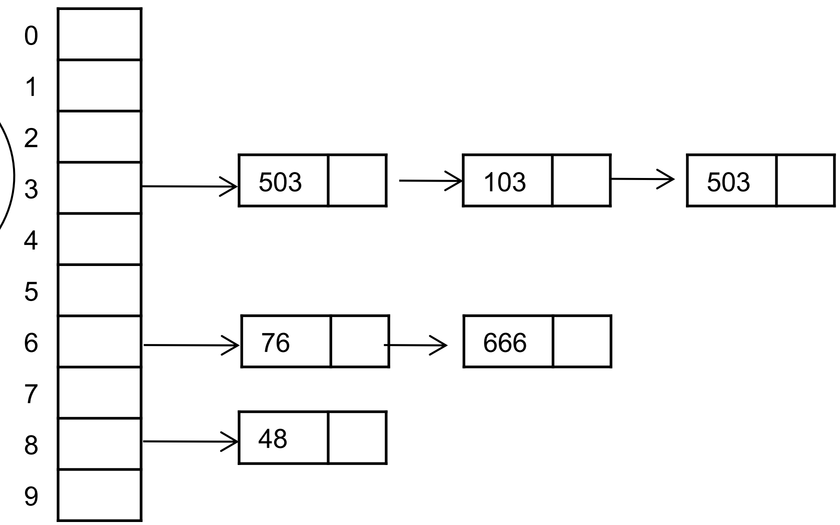
Supply $\bowtie_{sid=sid}$ Supplier

Build phase

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

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

Probe phase



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

Duplicate sid's

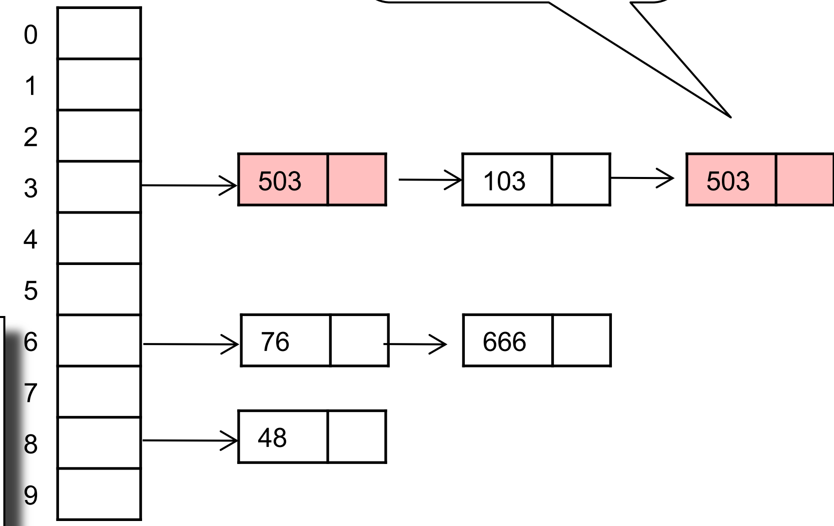
Logical operator:

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

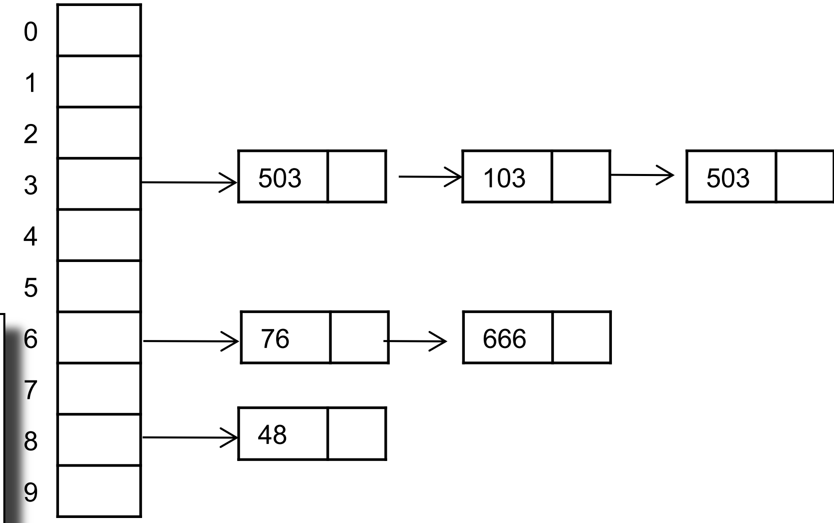
Logical operator:

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);
```



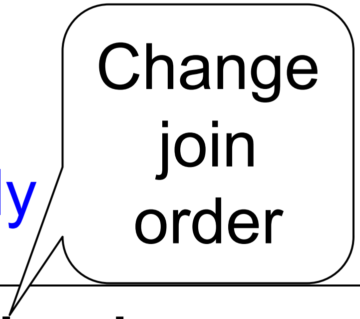
If |R|=|S|=n,
what is the runtime?

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

2. Hash Join

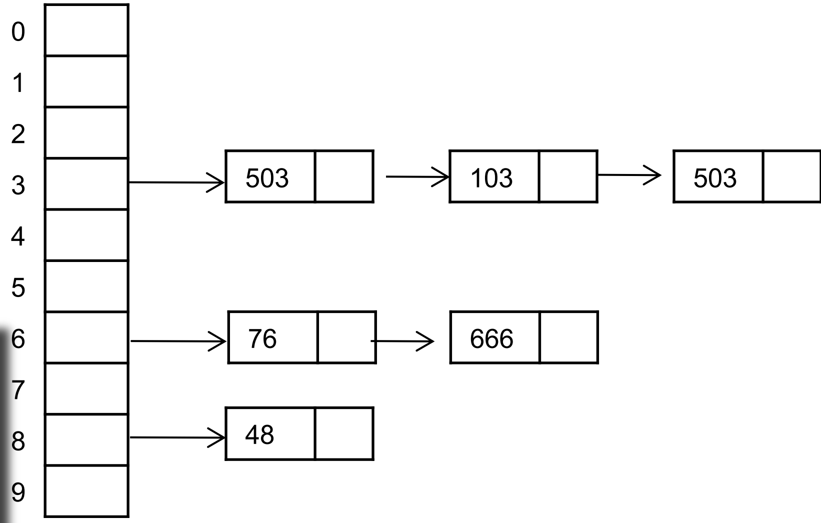
Logical operator:

Supplier $\bowtie_{sid=sid}$ Supply




```
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))

Sumamry of Main Memory Algorithms

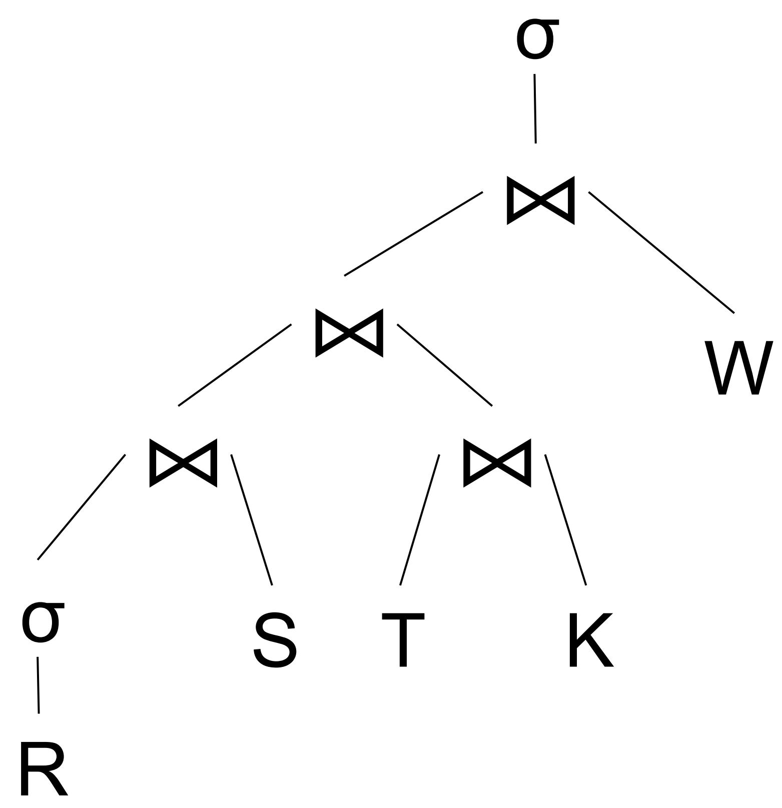
- Join ⋈:
  - Nested loop join
  - Hash join
  - Merge join
- Selection σ
  - "on-the-fly"
  - Index-based selection (next lecture)
- Group by γ
  - Hash–based
  - Merge-based

Briefly discuss in class

Outline

- Steps involved in processing a query
- Main Memory Operators
- Query execution
- External Memory Operators

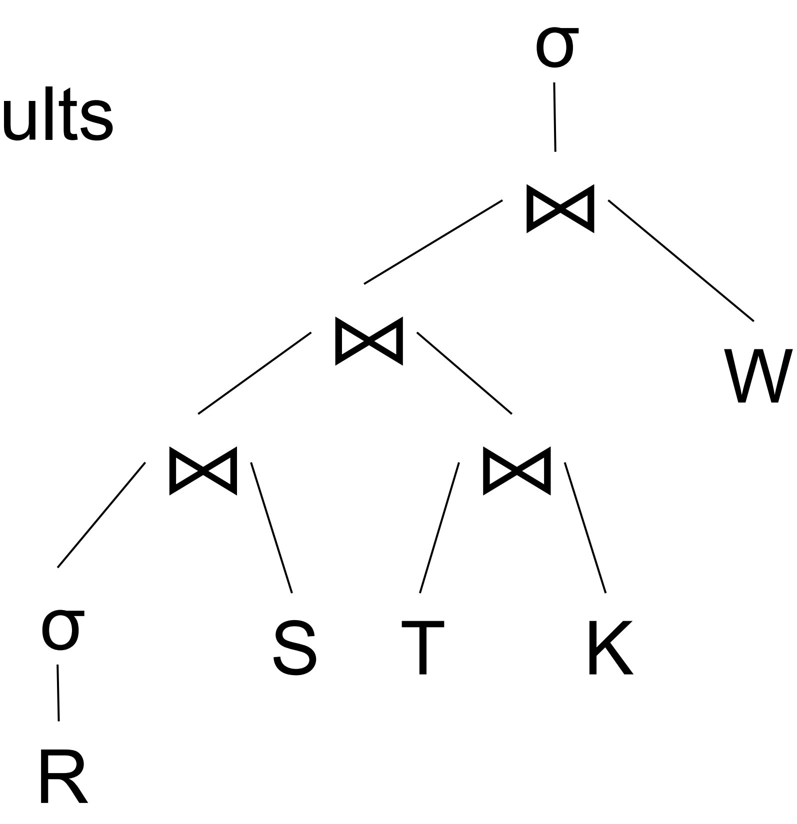
How Do We Combine Them?



How Do We Combine Them?

Option 1:
materialize intermediate results

Option 2:
Pipeline tuples btw. ops

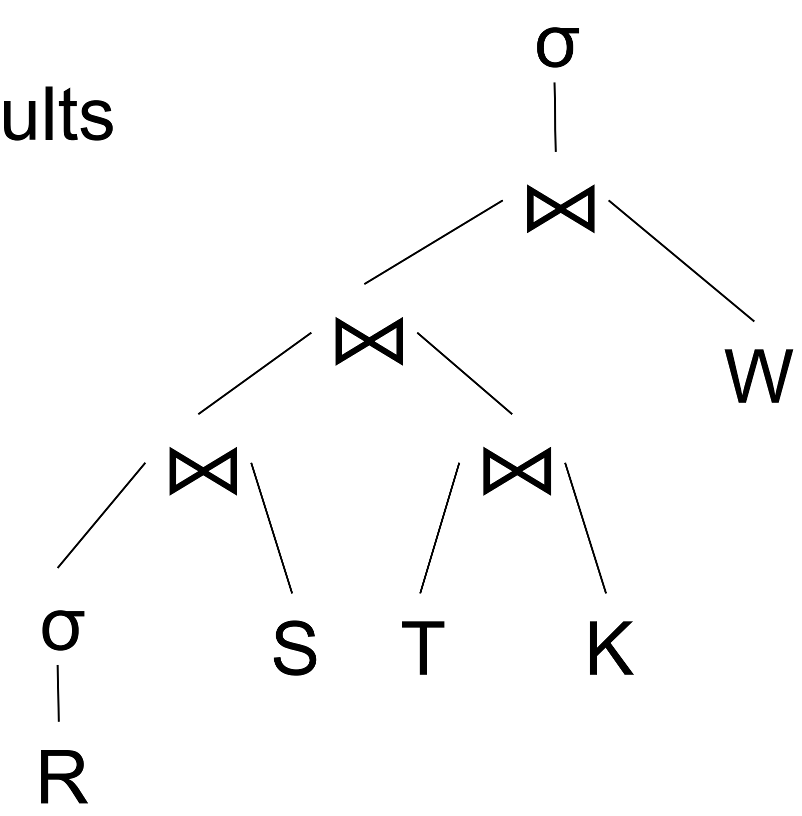


How Do We Combine Them?

Option 1:
materialize intermediate results

Option 2:
Pipeline tuples btw. ops

Implementation:
Iterator Interface



Operator Interface

Volcano model:

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

Operator Interface

Volcano model:

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

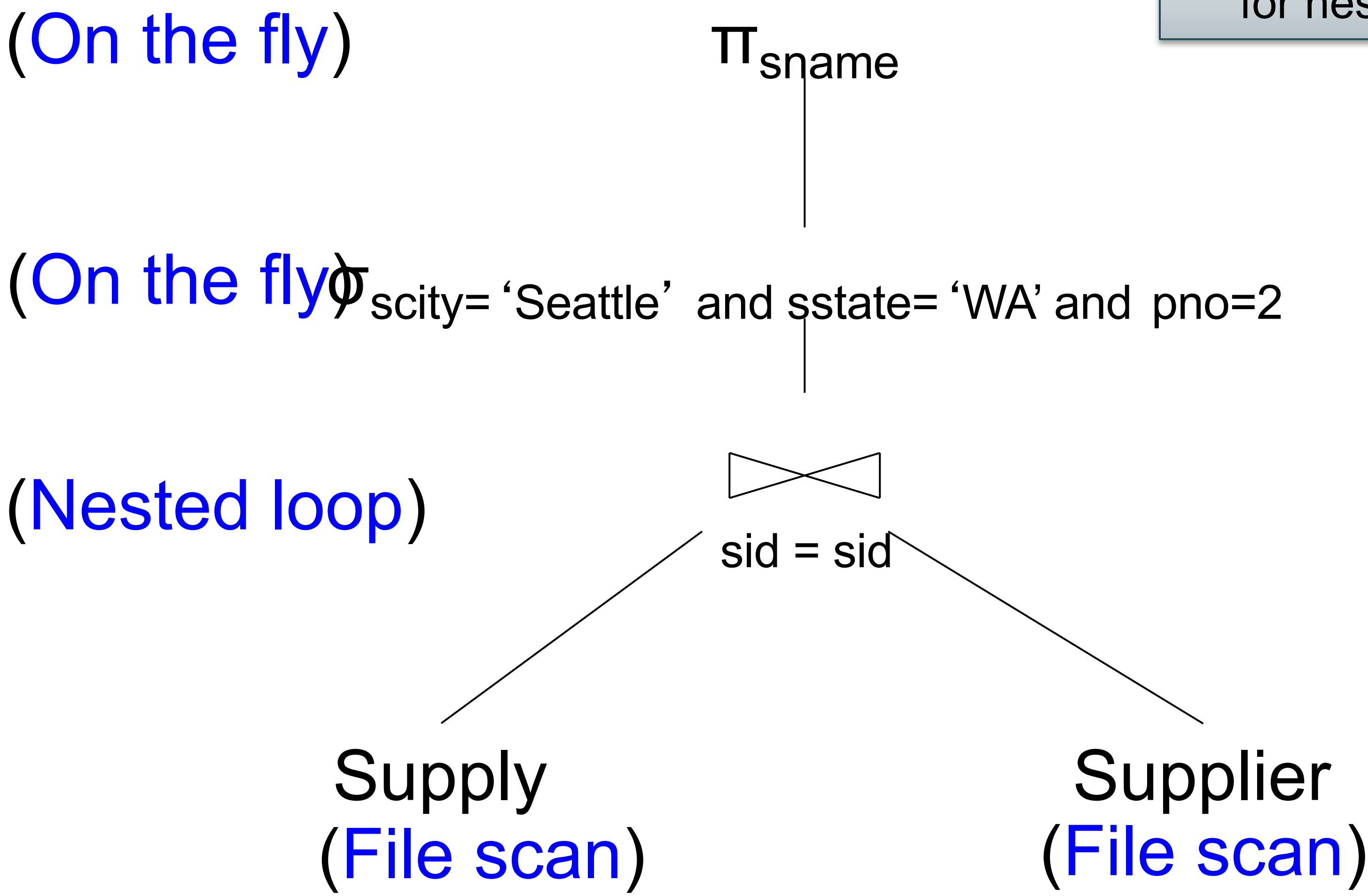
Data-driven model:

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

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

Pipelining

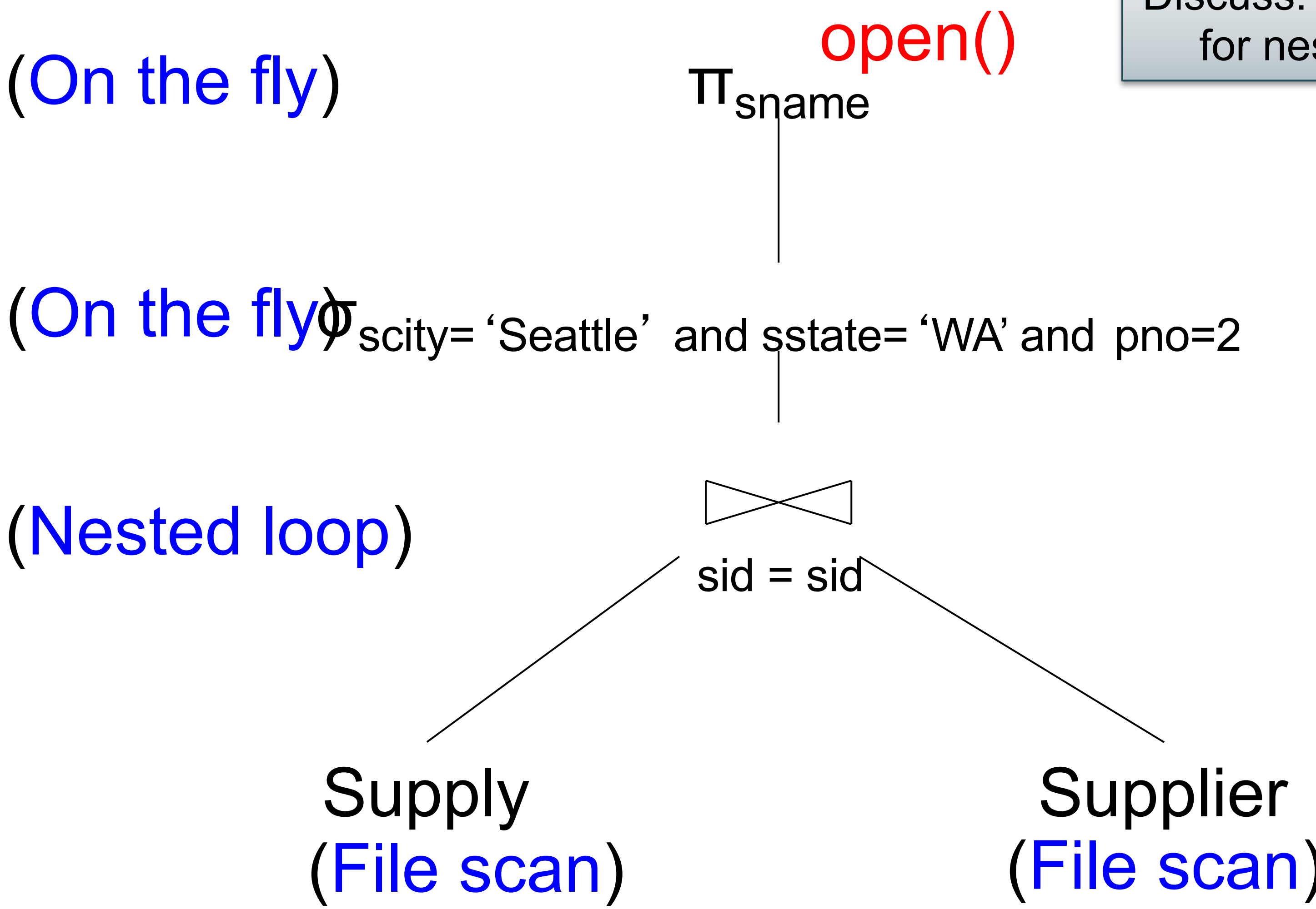
Discuss: open/next/close for nested loop join



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

Pipelining

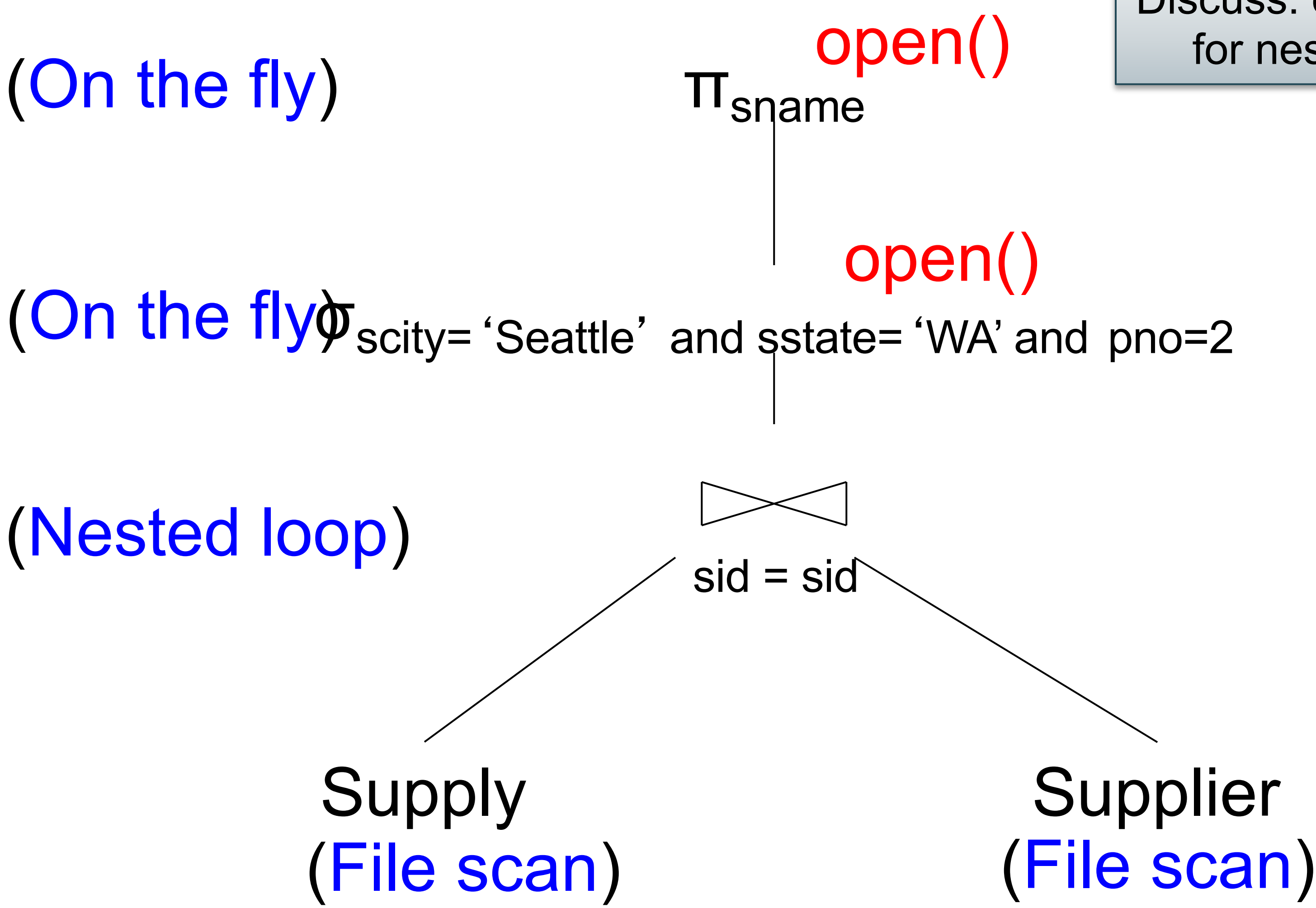
Discuss: open/next/close for nested loop join



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

Pipelining

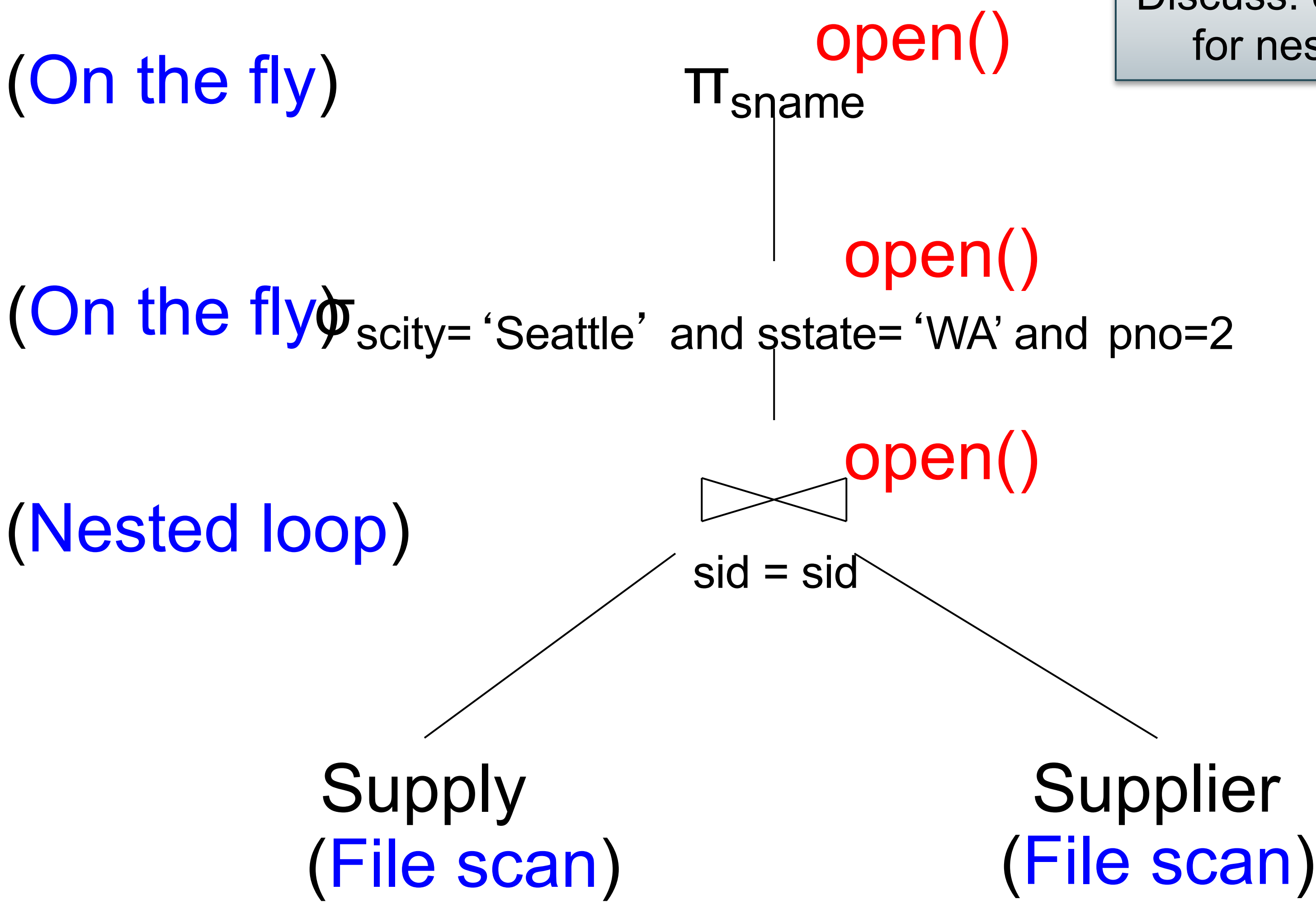
Discuss: open/next/close for nested loop join



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

Pipelining

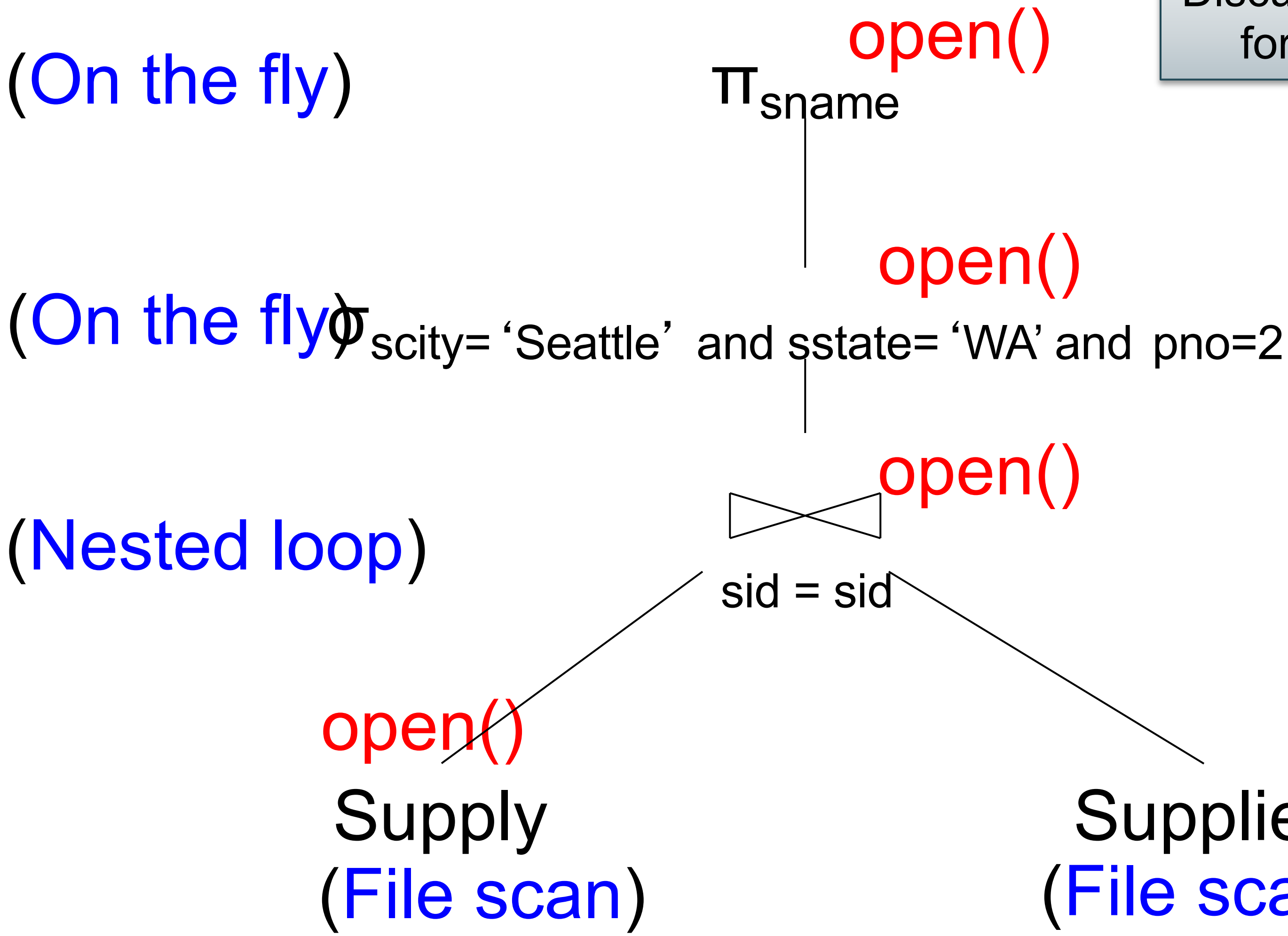
Discuss: open/next/close for nested loop join



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

Pipelining

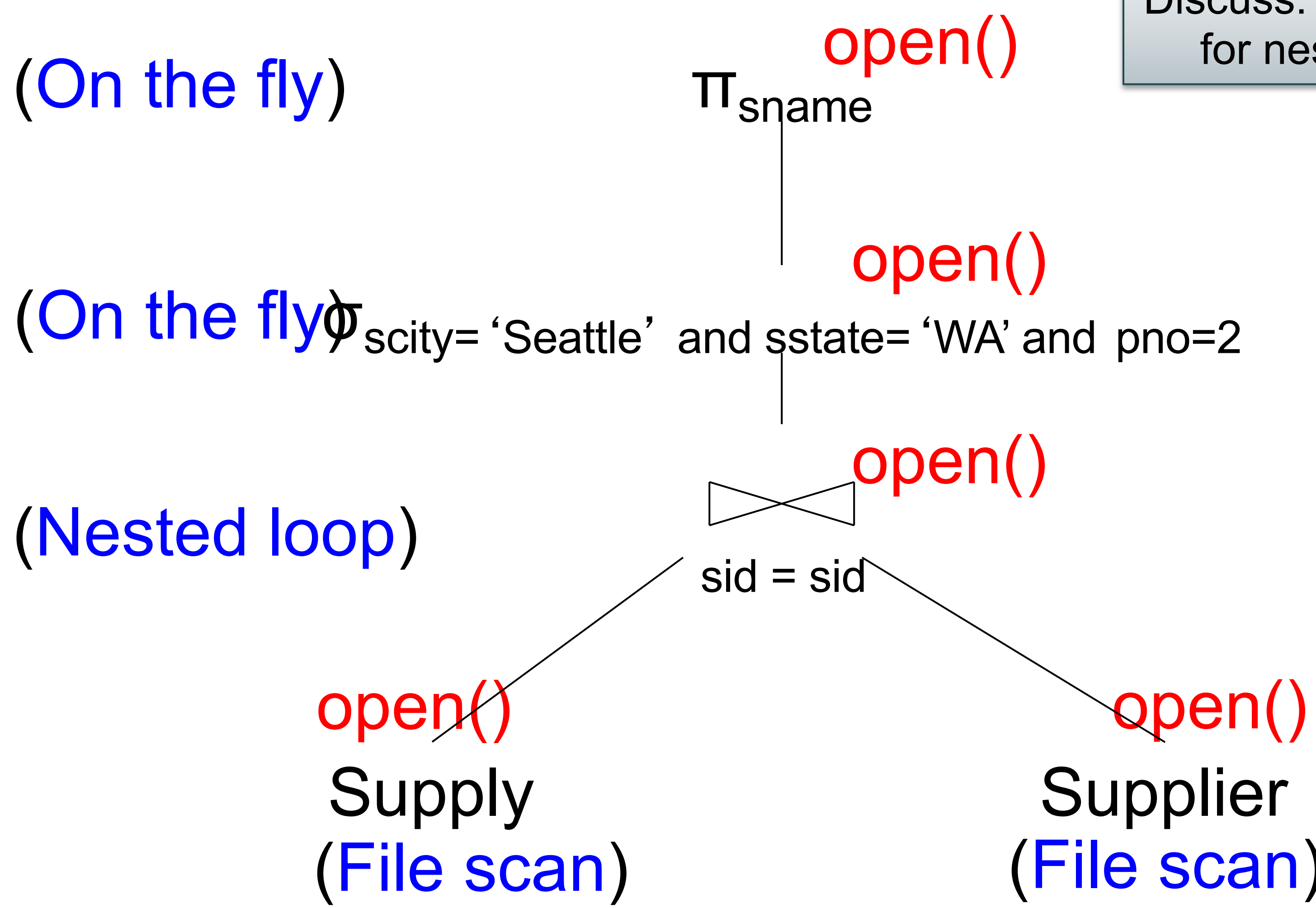
Discuss: open/next/close for nested loop join



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

Pipelining

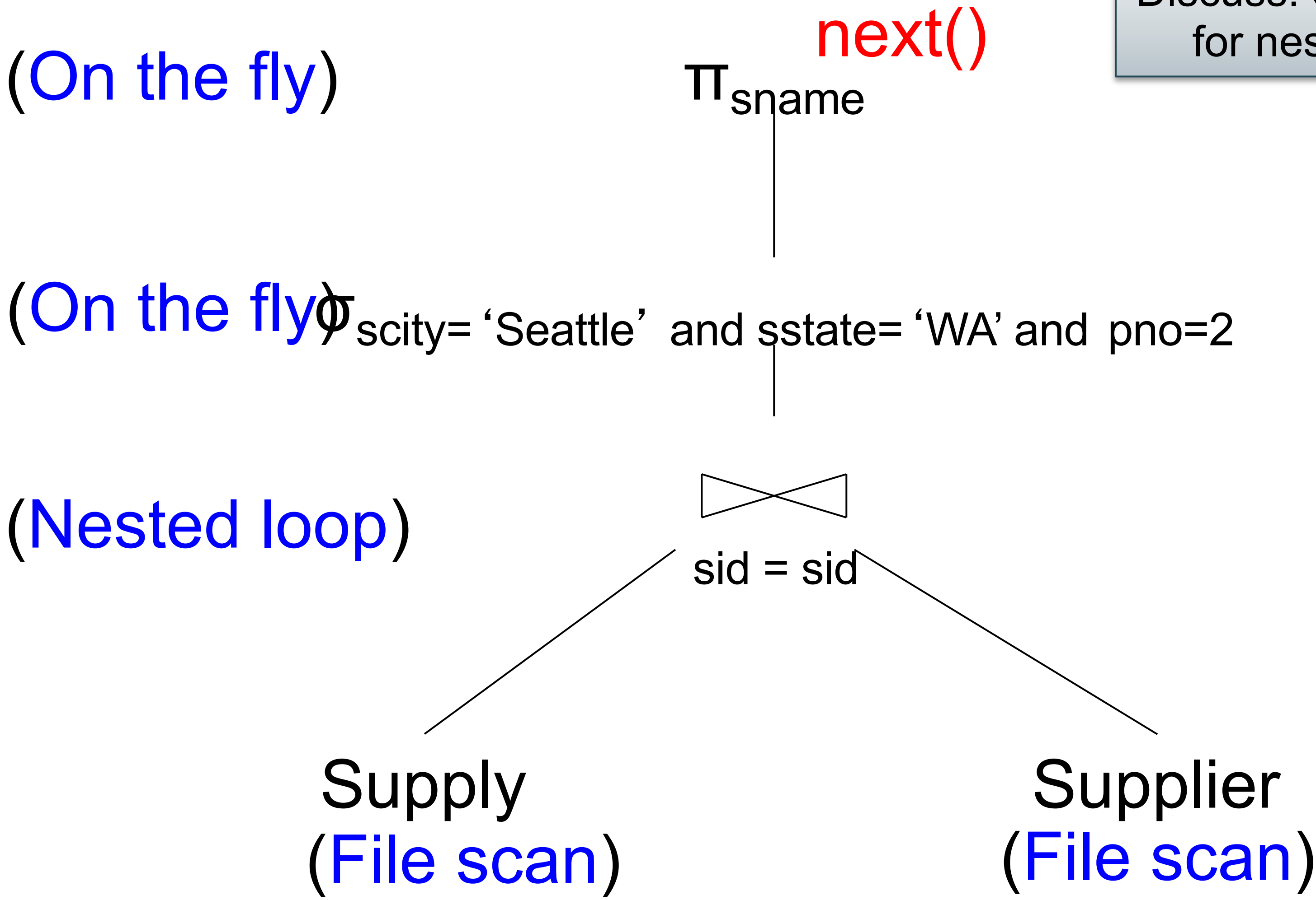
Discuss: open/next/close for nested loop join



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

Pipelining

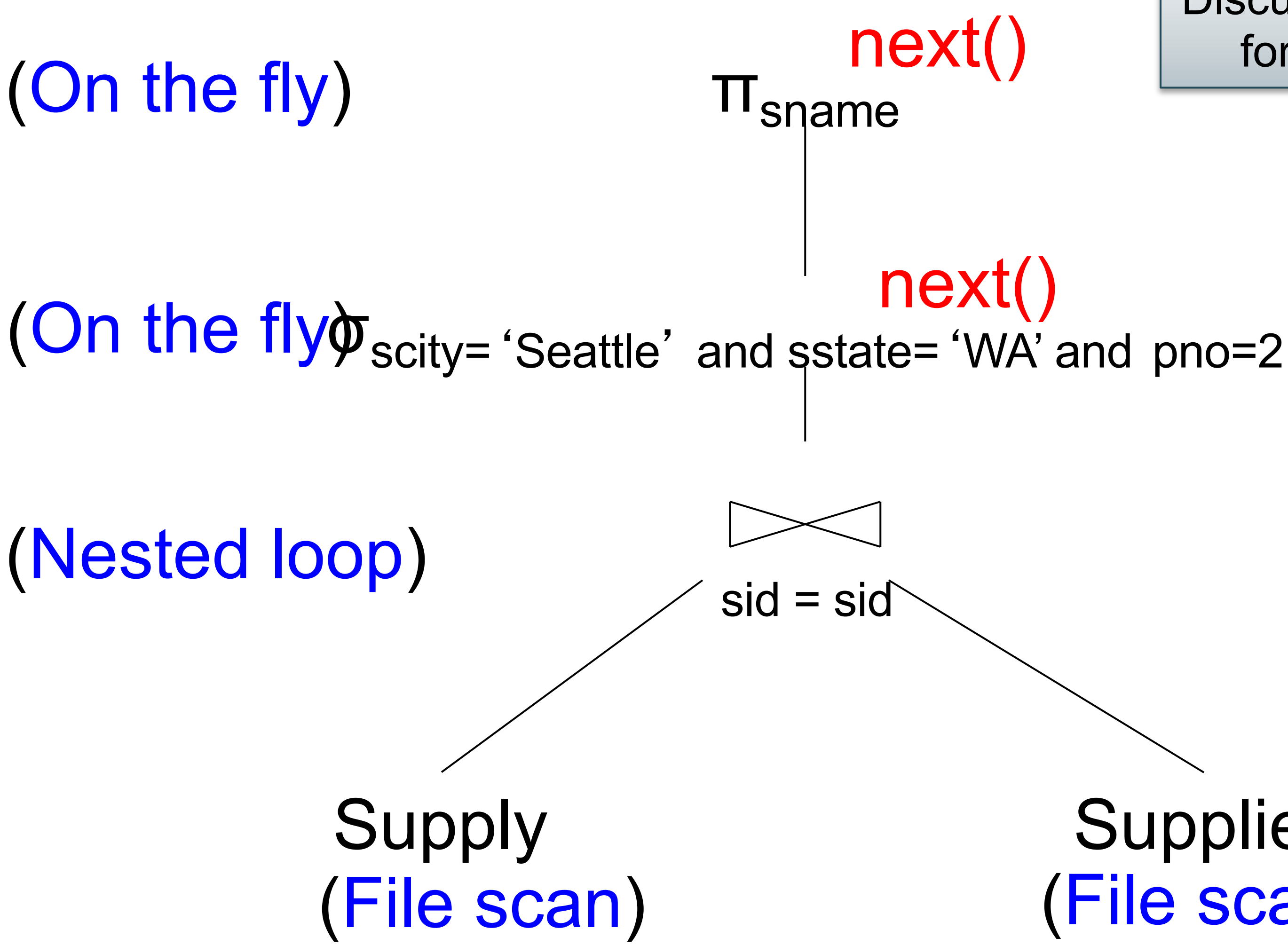
Discuss: open/next/close for nested loop join



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

Pipelining

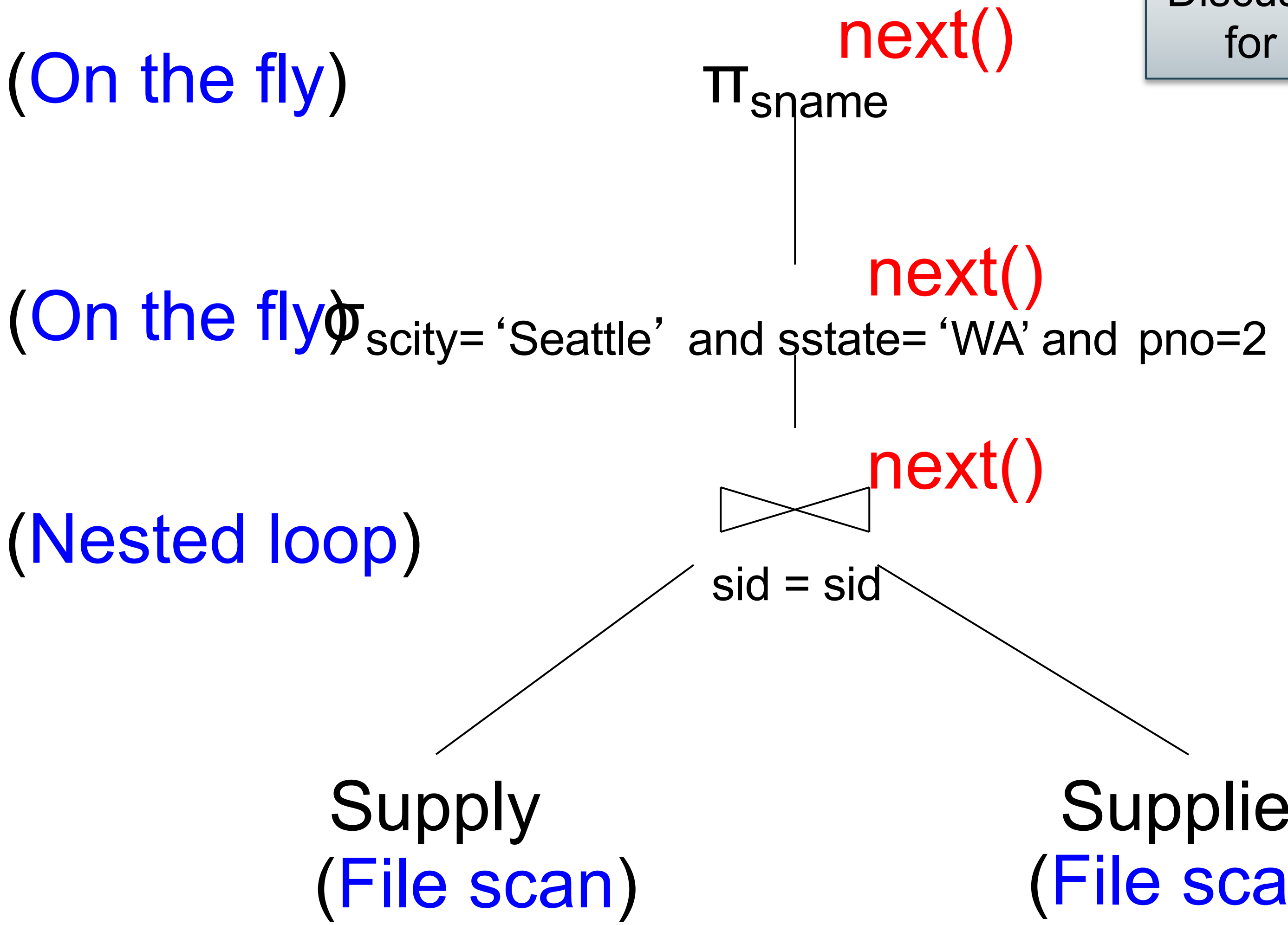
Discuss: open/next/close for nested loop join



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

Pipelining

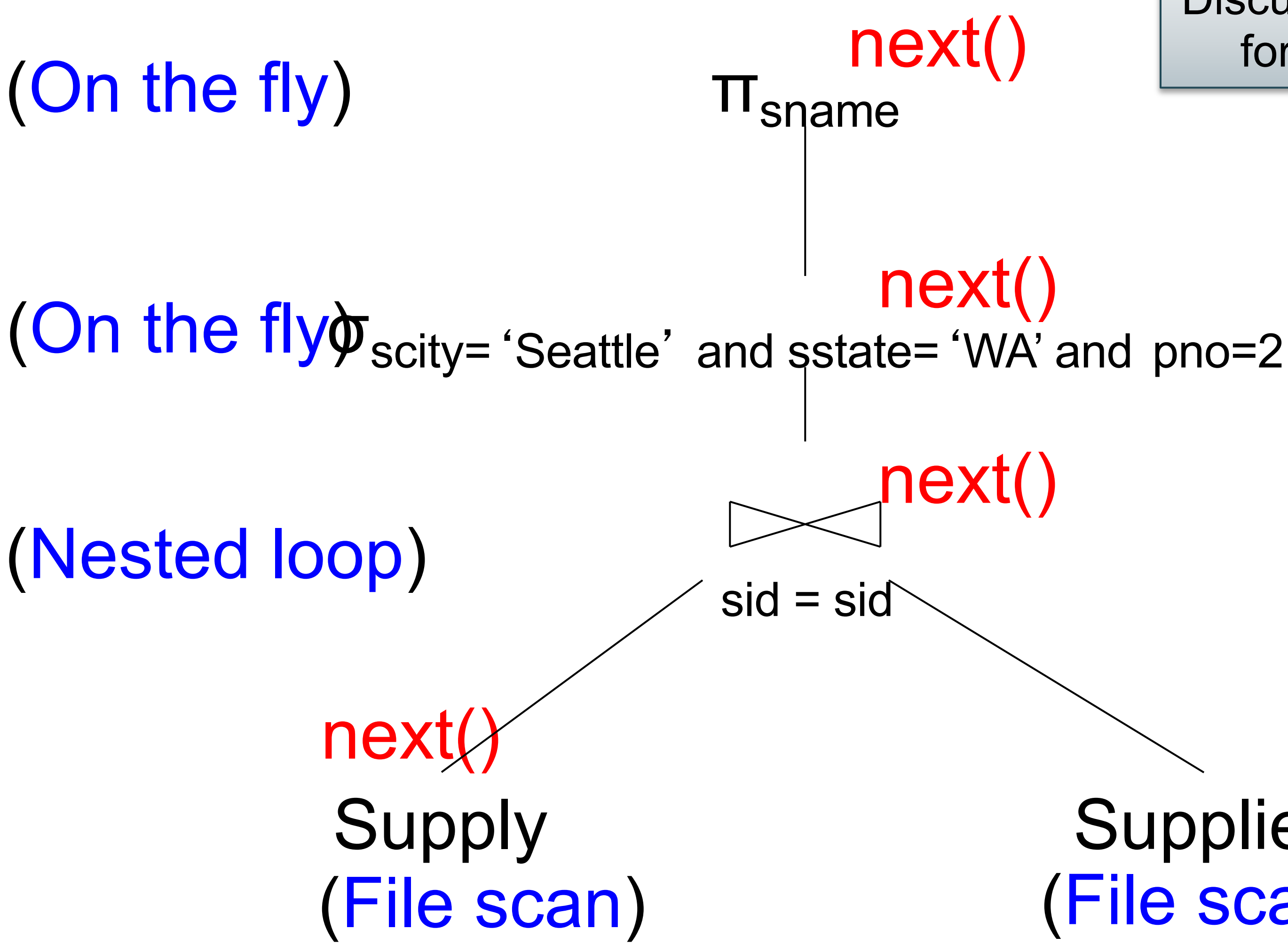
Discuss: open/next/close for nested loop join



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

Pipelining

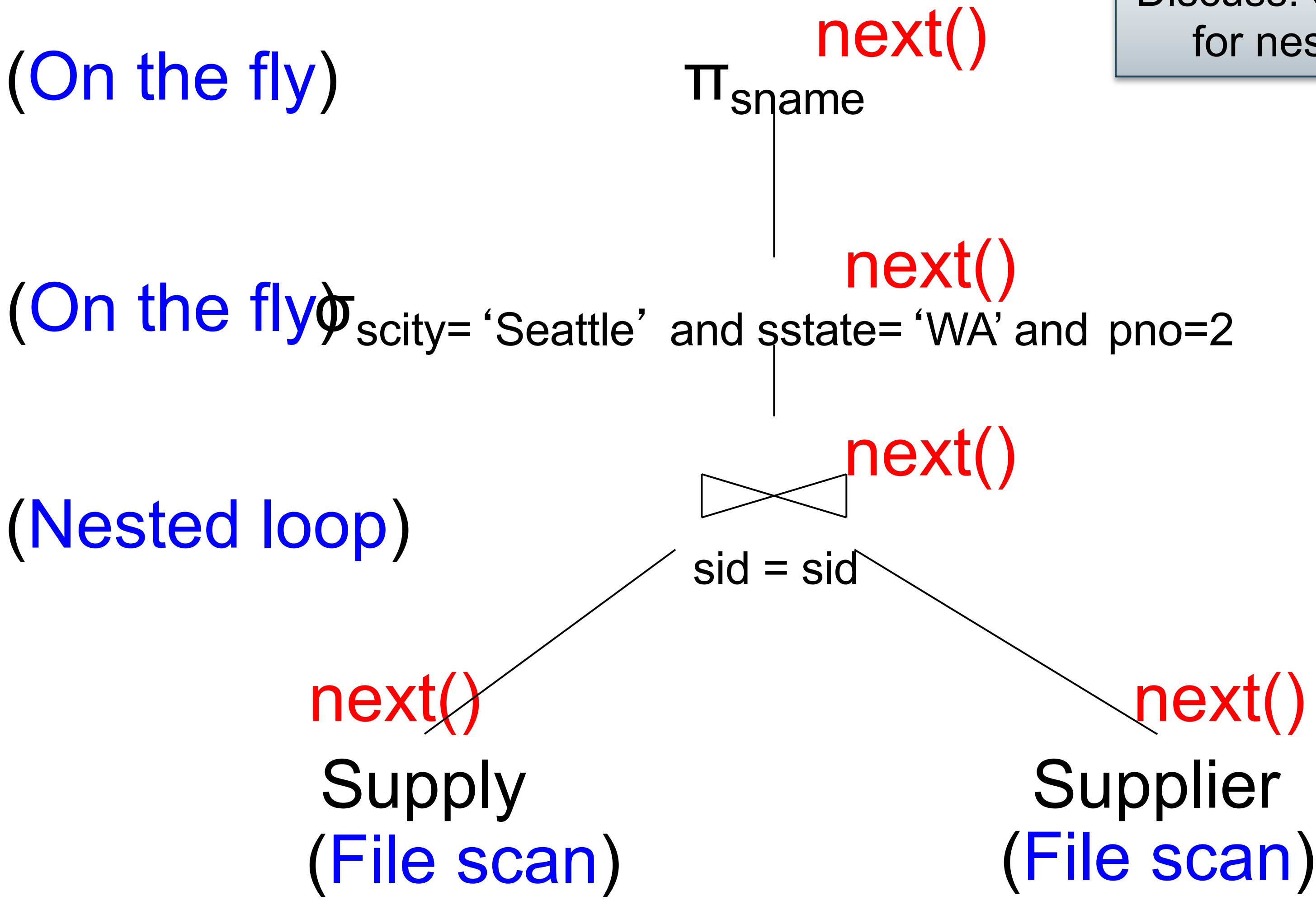
Discuss: open/next/close for nested loop join



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

Pipelining

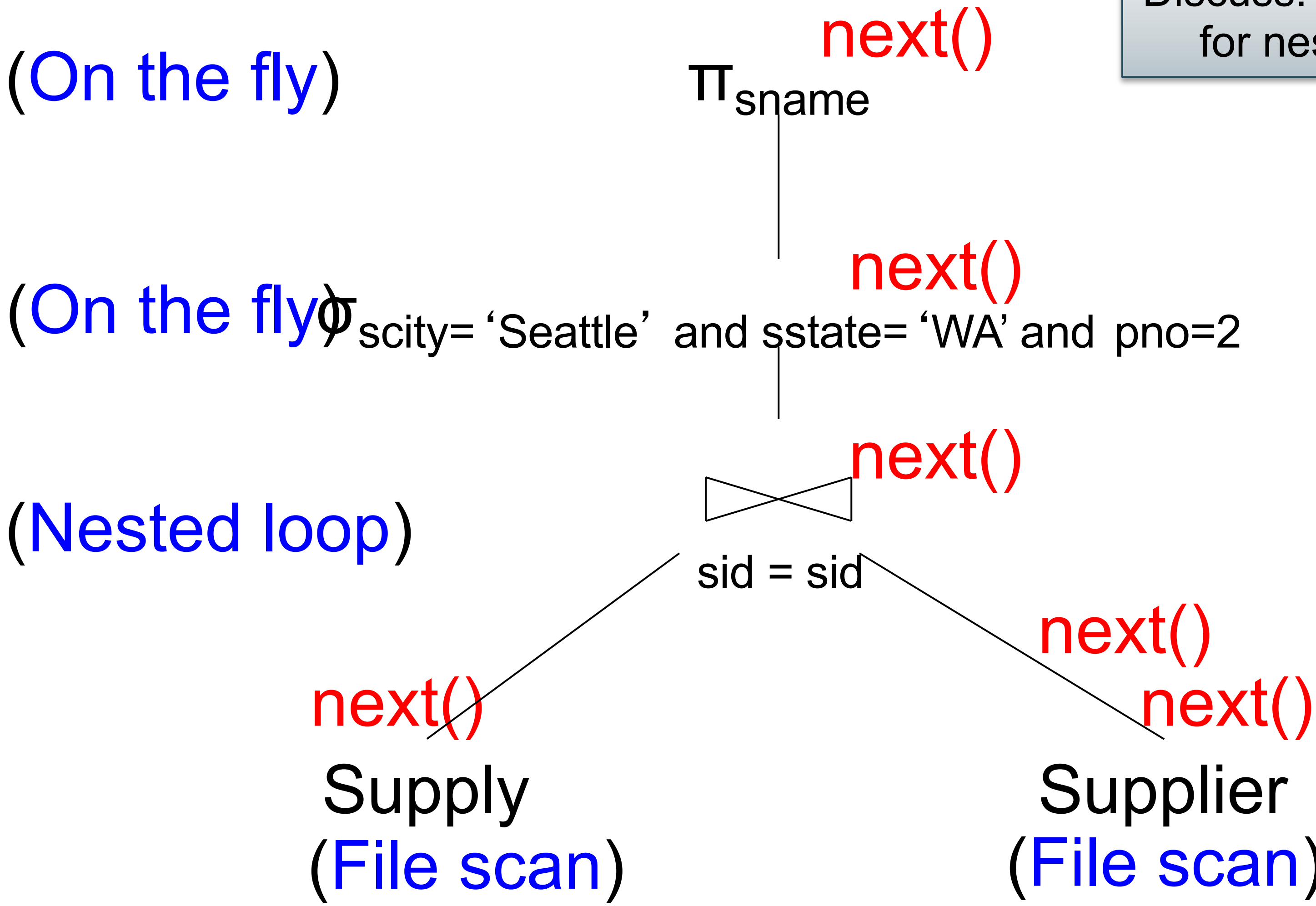
Discuss: open/next/close for nested loop join



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

Pipelining

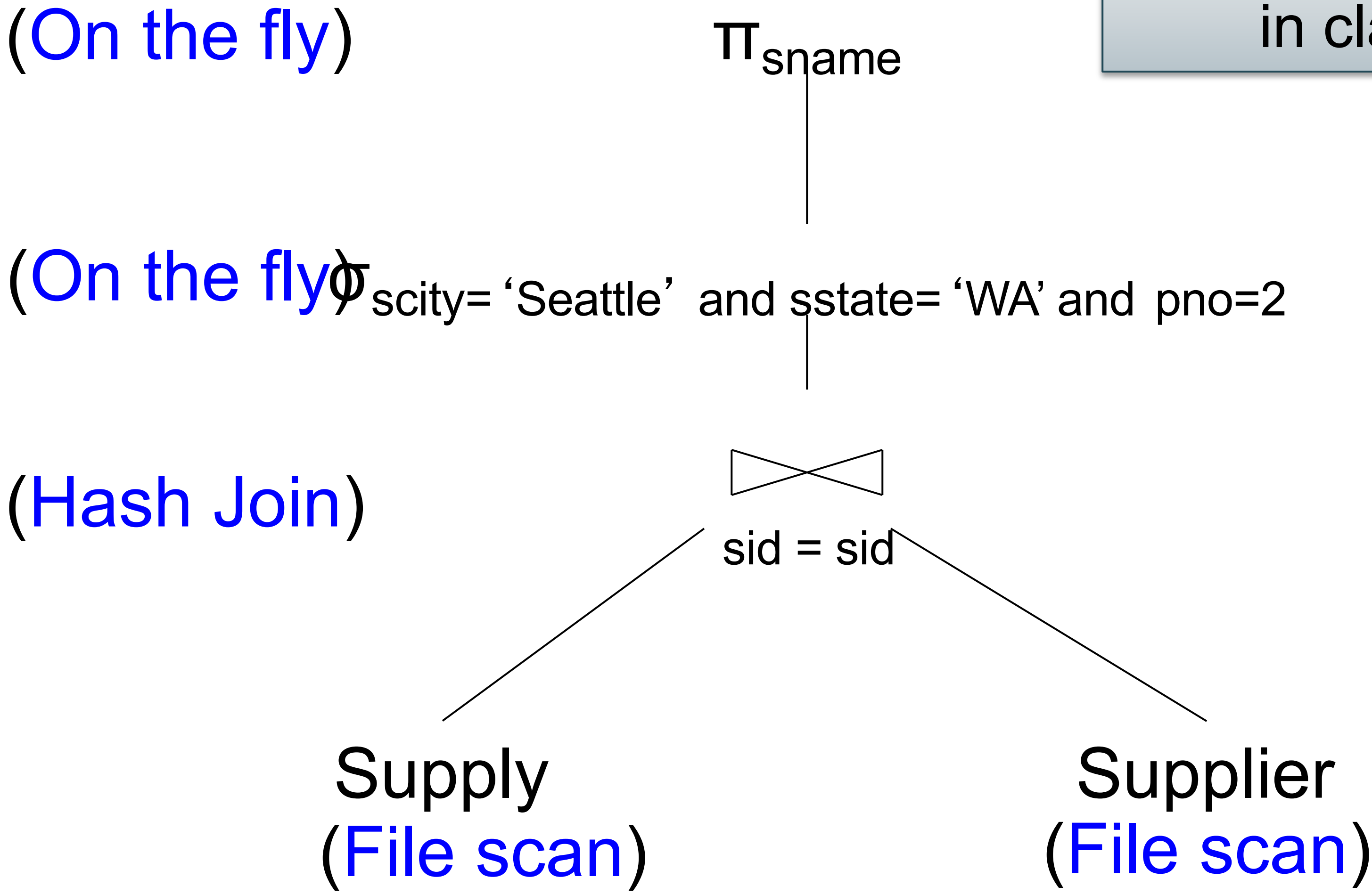
Discuss: open/next/close for nested loop join



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

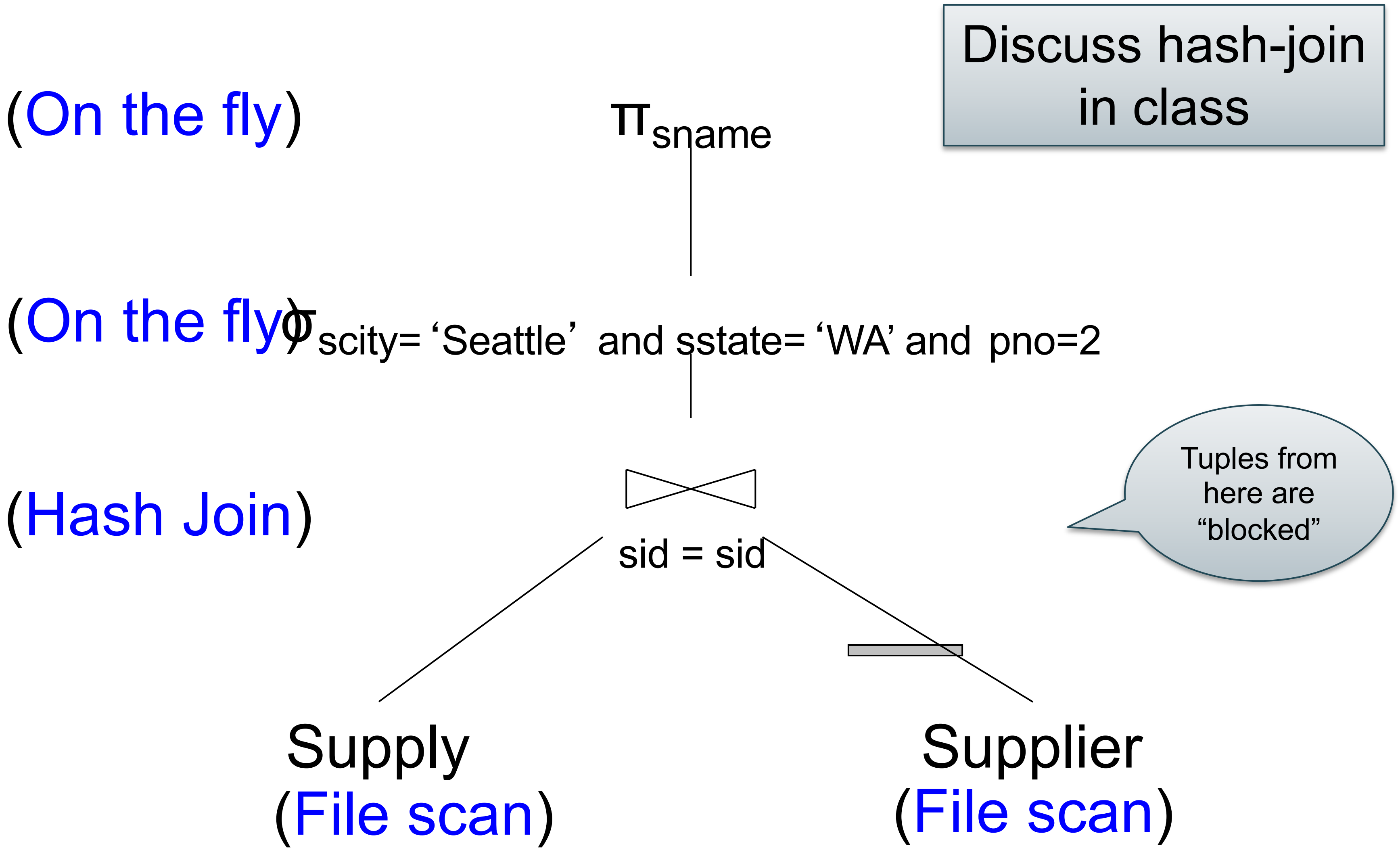
Pipelining

Discuss hash-join in class



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

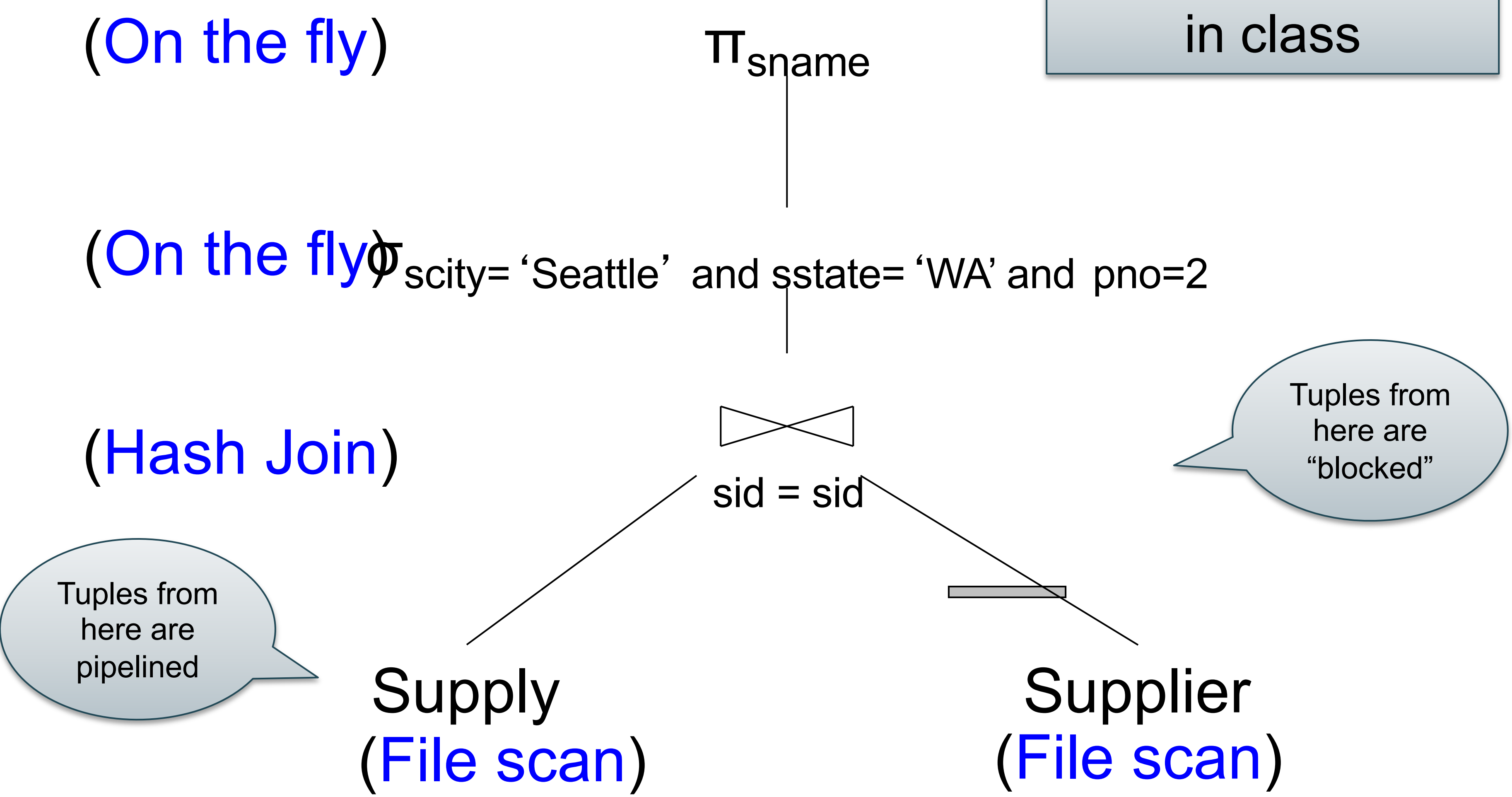
Pipelining



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

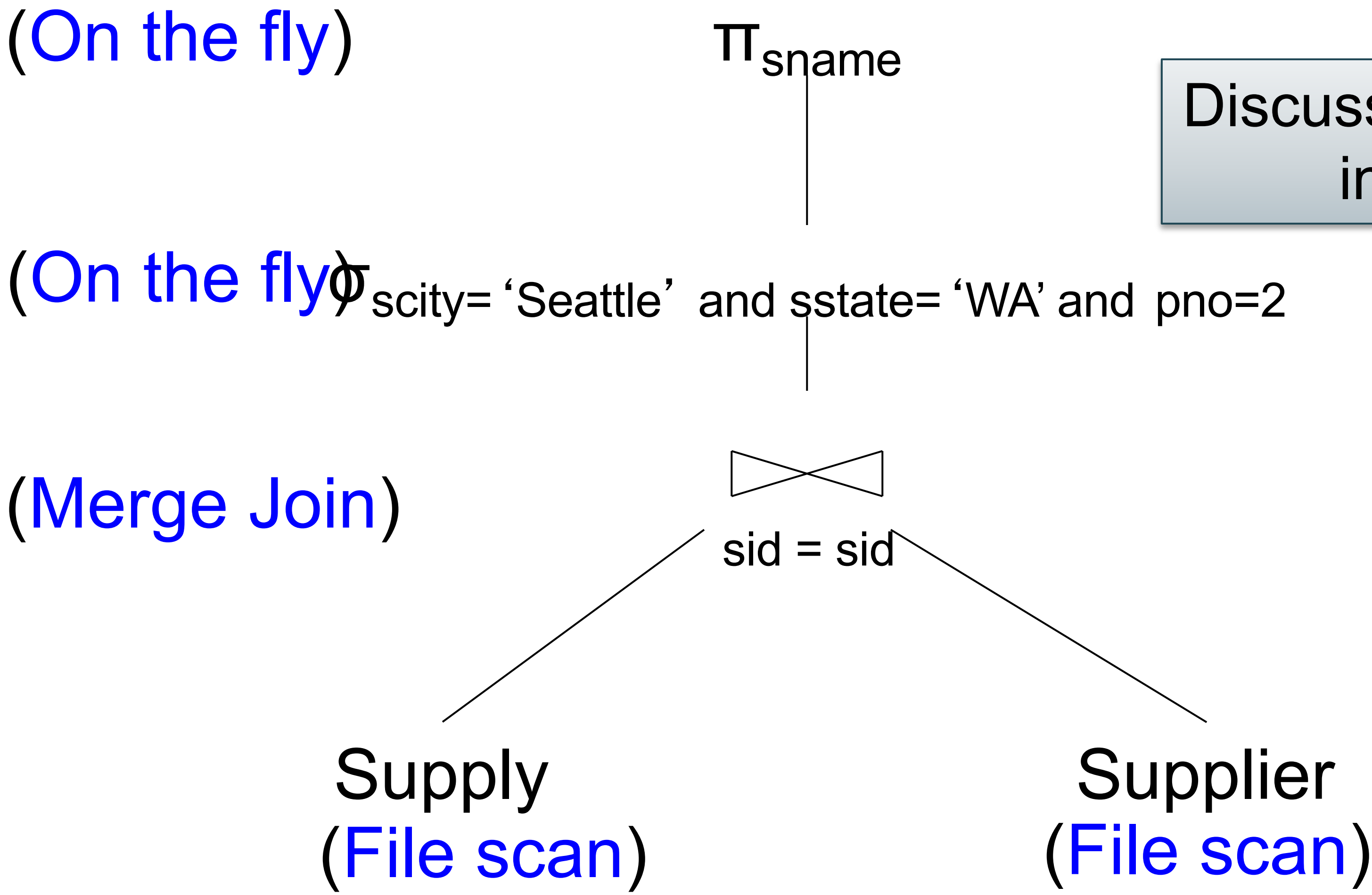
Pipelining

Discuss hash-join in class



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

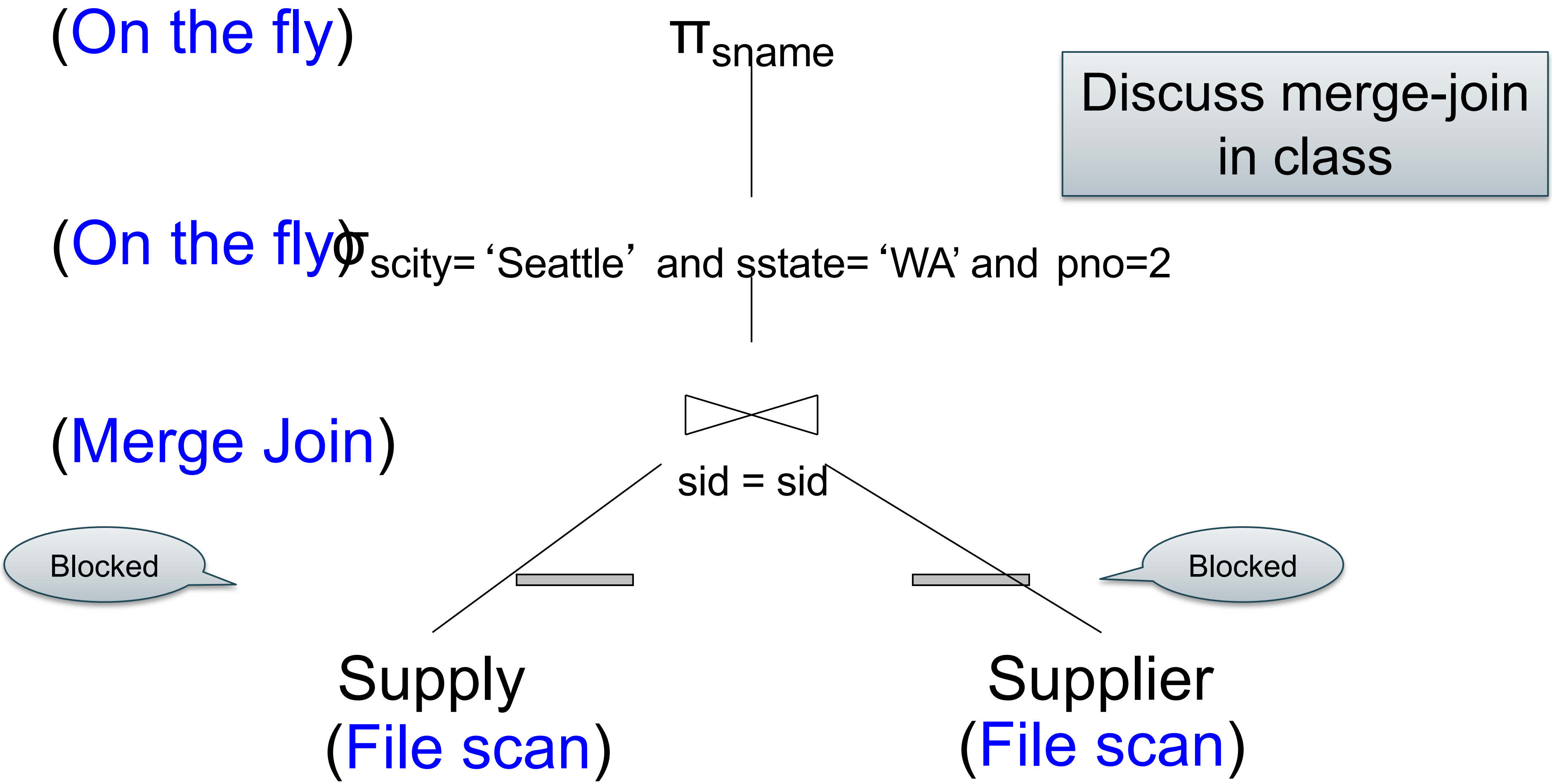
Blocked Execution



Discuss merge-join in class

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

Blocked Execution



Pipeline v.s. Blocking

- Pipeline
  - A tuple moves 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

Iterator Model
A.k.a. Volcano-style execution

```
interface Operator {







}
```

Iterator Model
A.k.a. Volcano-style execution

```
interface Operator {

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

}
```

Iterator Model
A.k.a. Volcano-style execution

```
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 ();




}
```

Iterator Model
A.k.a. Volcano-style execution

```
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 ();
}
```

Iterator Model
A.k.a. Volcano-style execution

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 ();
}
```

Iterator Model
A.k.a. Volcano-style execution

```
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();
  }
}
```

Iterator Model
A.k.a. Volcano-style execution

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 ();
}
```

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




  }
}
```

Iterator Model
A.k.a. Volcano-style execution

```
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);
    }

  }
}
```

Iterator Model
A.k.a. Volcano-style execution

```
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;
  }

}
```

Iterator Model
A.k.a. Volcano-style execution

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 ();
}
```

```
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(); }
}
```

Iterator Model
A.k.a. Volcano-style execution

```
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();
```

Summary

- Three join operators: loop, hash, merge
- Many variations:
  - ”double pipelined hash join” – what is this?
- Tuple flow: materialize, pipeline
- Interface:
  - Volcano (pull): next()
  - Data-driven (push): produce(), consume()