Introduction to Data Management
CSE 414

Unit 4: RDBMS Internals

Logical and Physical Plans

Query Execution

Query Optimization

(3 lectures)

Introduction to Data Management CSE 414

Lecture 15: Introduction to Query Evaluation

Announcements

- WQ5 (datalog) due tomorrow

- HW4 (datalog) due tomorrow

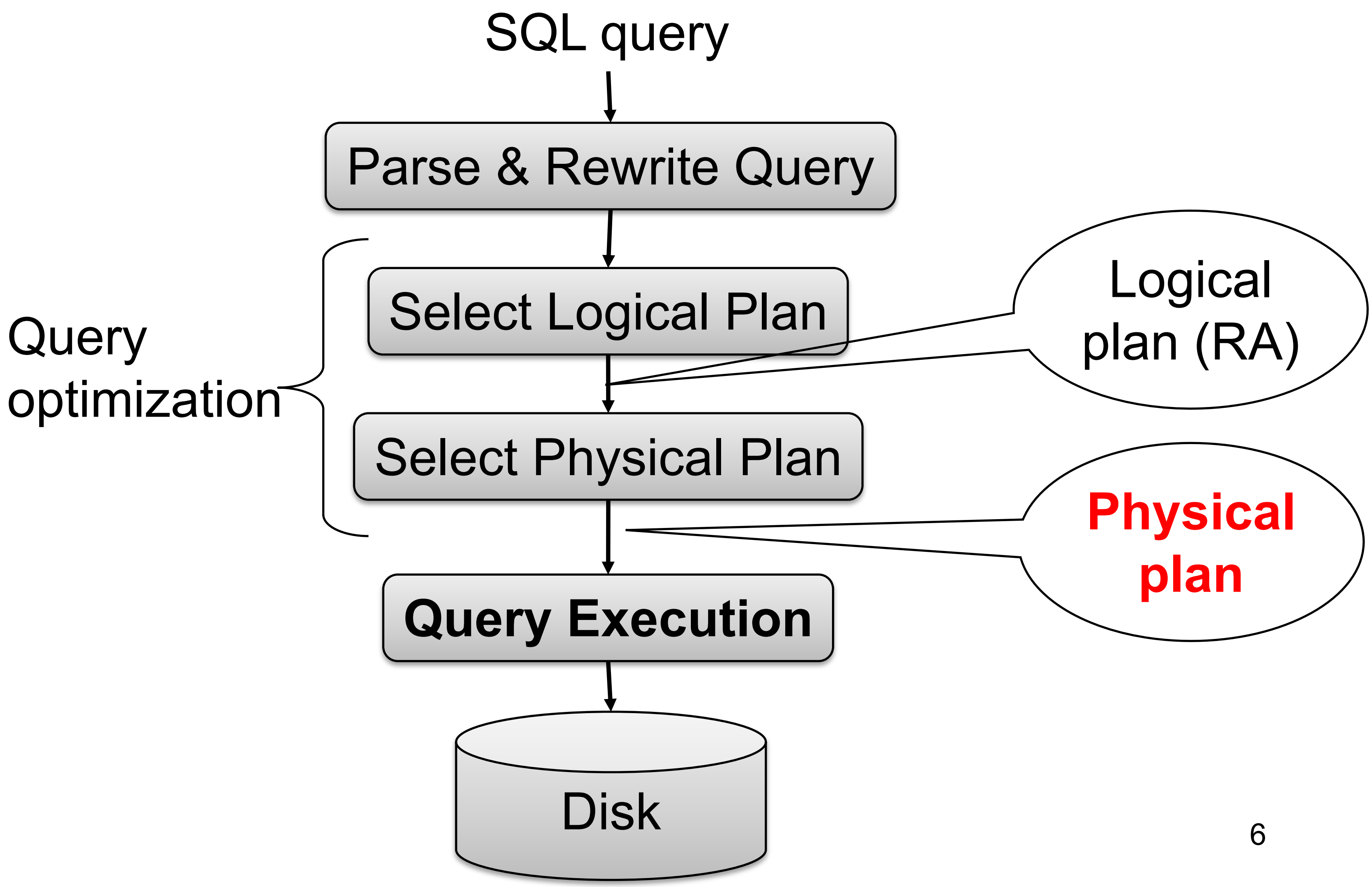
- Midterm review session this evening
  - 5:30pm, CSE 2nd Floor Breakout

Class Overview

- Unit 1: Intro
- Unit 2: Relational Data Models and Query Languages
- Unit 3: Non-relational data
- Unit 4: RDMBS internals and query optimization
- Unit 5: Parallel query processing
- Unit 6: DBMS usability, conceptual design
- Unit 7: Transactions
- Unit 8: Advanced topics (time permitting)

From Logical RA Plans to Physical Plans

Query Evaluation Steps Review



How plans is laid out vs how the plan is actual run out

Logical vs Physical Plans

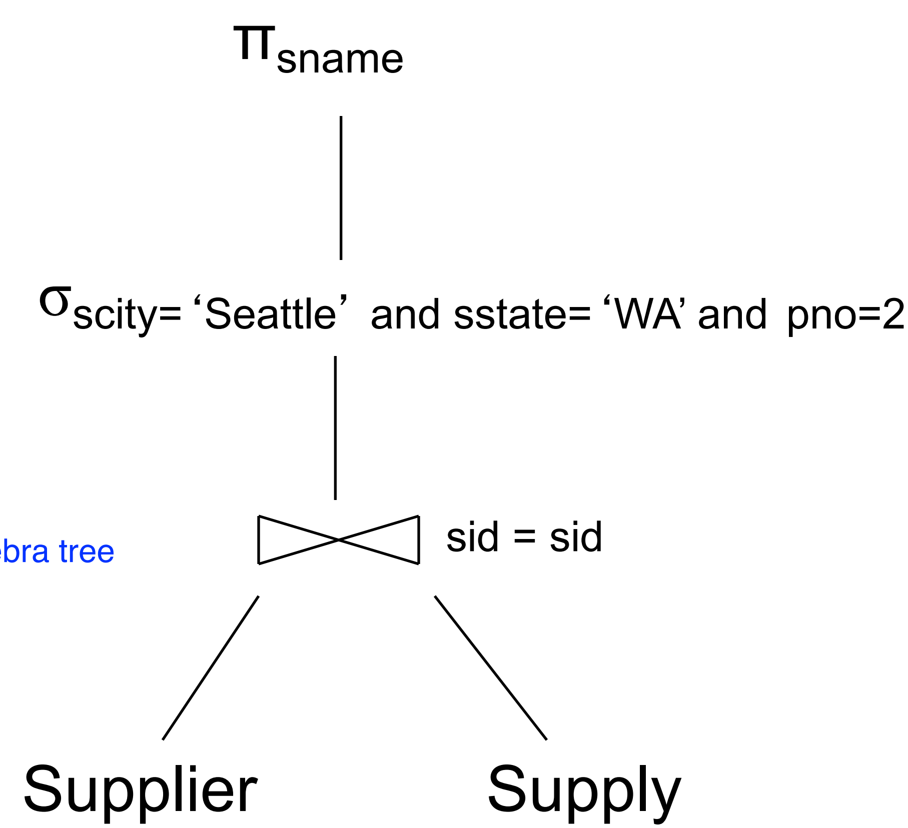
- Logical plans:
  - Created by the parser from the input SQL text
  - Expressed as a relational algebra tree
  - Each SQL query has many possible logical plans

- Physical plans:
  - Goal is to choose an efficient implementation for each operator in the RA tree
  - Each logical plan has many possible physical plans

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

Review: Relational Algebra

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



we already know that there exists multiple valid relational algebra tree and thus there are multiple logical plans!

Relational algebra expression is also called the “logical query plan”

Logical Plan v.s. Physical Plan

- Logical Plan = a Relational Algebra tree
- Physical Plan = a Logical Plan plus annotation of each operator with an algorithm

Text

Query Optimization and Execution

- Query optimizer:
  - Choose a good logical plan
  - Refine it to a good physical plan
  - Sometimes these steps are intertwined
- Query execution usually this takes far longer, which is why it makes sense to invest time into optimizing the query
  - Execute the physical plan

Query Execution

Physical Operators

Relational algebra operators:

- Selection, projection, join, union, difference
- Group-by, distinct, sort

Physical operators:

- For each operators above, several possible algorithms
- Main memory algorithms, or disk-based algorithms

where are the
operations done?

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

Main Memory Algorithms

Logical operator:

Supplier $\bowtie_{sid=sid}$ Supply

Propose three physical operators for the join, assuming the tables are in main memory:

1.
2.
3.

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

Main Memory Algorithms

Logical operator:

Supplier $\bowtie_{sid=sid}$ Supply   each option here has a different expected runtime

Propose three physical operators for the join, assuming the tables are in main memory:

1. Nested Loop Join $O(n^2)$ cartesian product?
2. Merge join $O(n \log n)$ Merge and Hash joins do not work well with nonequality constraints
3. Hash join $O(n) \ldots O(n^2)$

balance between how much memory is required and runtime … as usual

BRIEF Review of Hash Tables

A (naïve) hash function:

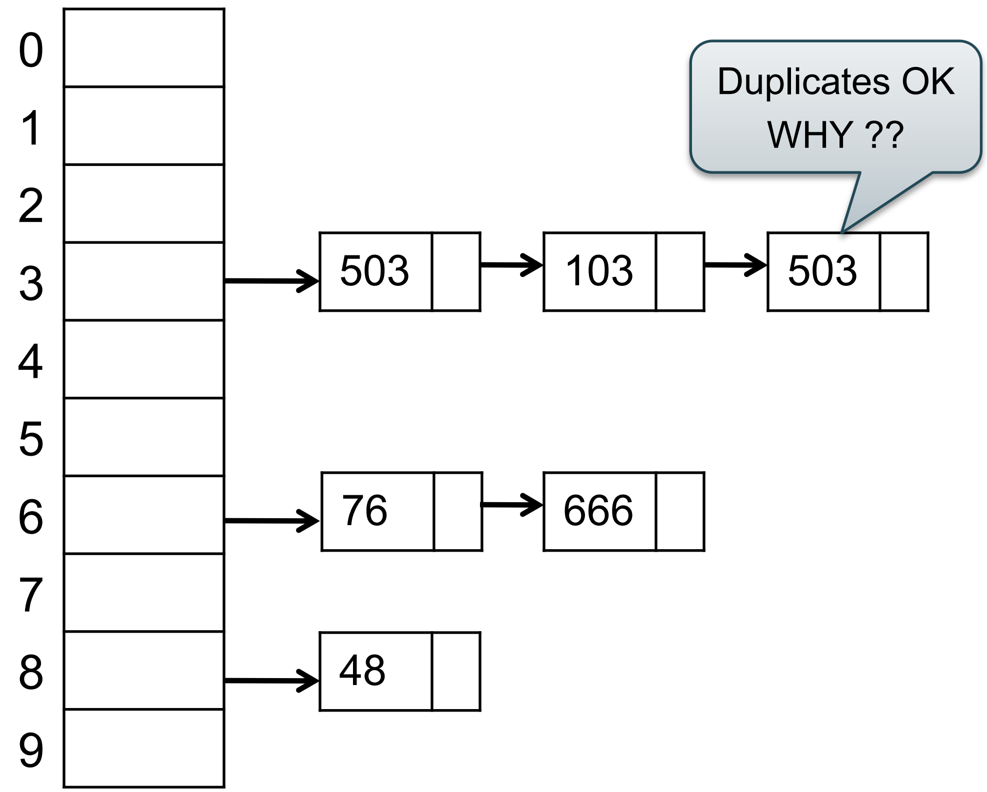
$h(x) = x \bmod 10$

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

Query Execution

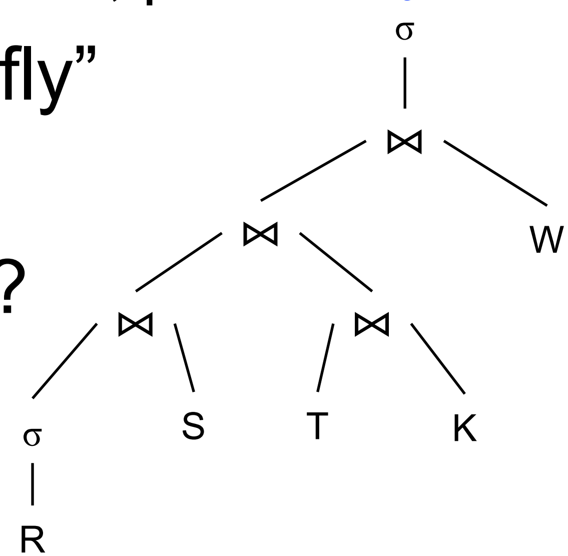
- Join R ⋈ S: e.g. using hash-join:
  - Nested-loop: forall x in R forall y in S do …
  - Hash–join: build a hash table on S, probe R

loop over table R, but now we can quickly access values of S because they have been hashed, assuming good hashing

- Selection: σ(R): e.g. “on-the-fly”

on the fly?

- But what about a larger plan?
  - Each operator implements the Iterator Interface



Tree + implementation choices = physical plan

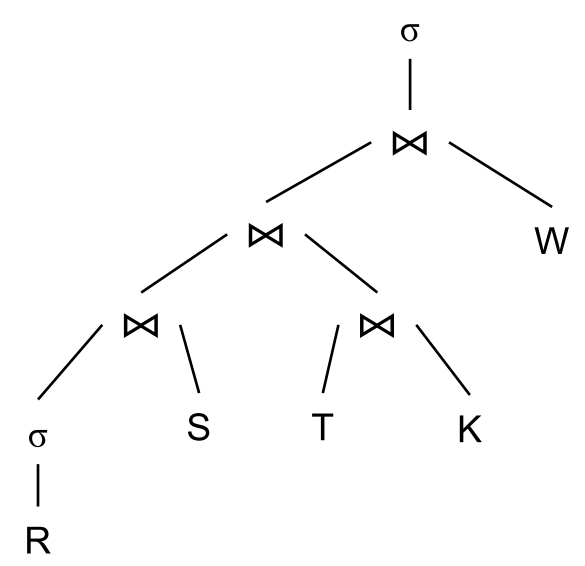
Implementing Query Operators with the Iterator Interface

in query plan
Each operator implements three methods:

sets up/starts operator
- open()

iterator; requests next input value from this operator
(returns a single tuple)
- next()

shuts down operator
- close()



Implementing Query Operators with the Iterator Interface

Example "on the fly" selection operator

```
interface Operator {










}
```

Implementing Query Operators with the Iterator Interface

Example "on the fly" selection operator

```
interface Operator {

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








}
```

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 ();


}
```

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 child) {
    this.p = p; this.child = child;
  }
}
```

what condition are we checkin
operators below it

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 ();
}
```

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

loops over until it runs out of tuples to check or it finds a match, which it returns

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 child) {
    this.p = p; this.child = child;
  }
  Tuple next () {
    boolean found = false;
    Tuple r = null;
    while (!found) {
       r = child.next();
       if (r == null) break;
       found = p(in);
    }
    return r;
  }
  void close () { child.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);
```

Final operator at top of tree; works to optimize query plan?

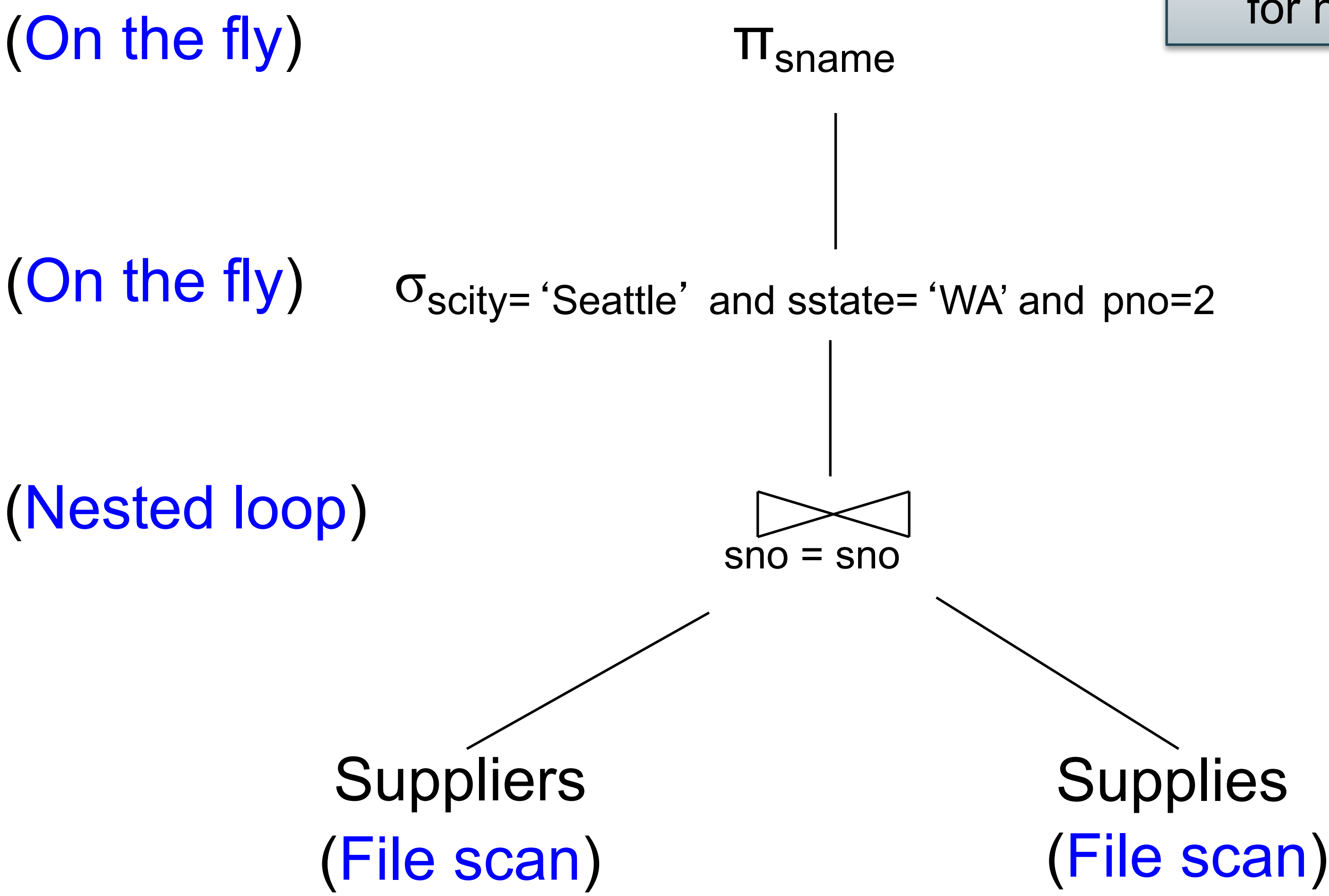
```
q.open();
while (true) {
  Tuple t = q.next();
  if (t == null) break;
  else printOnScreen(t);
}
q.close();
```

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

Pipelining

PHYSICAL PLAN

Discuss: open/next/close for nested loop join

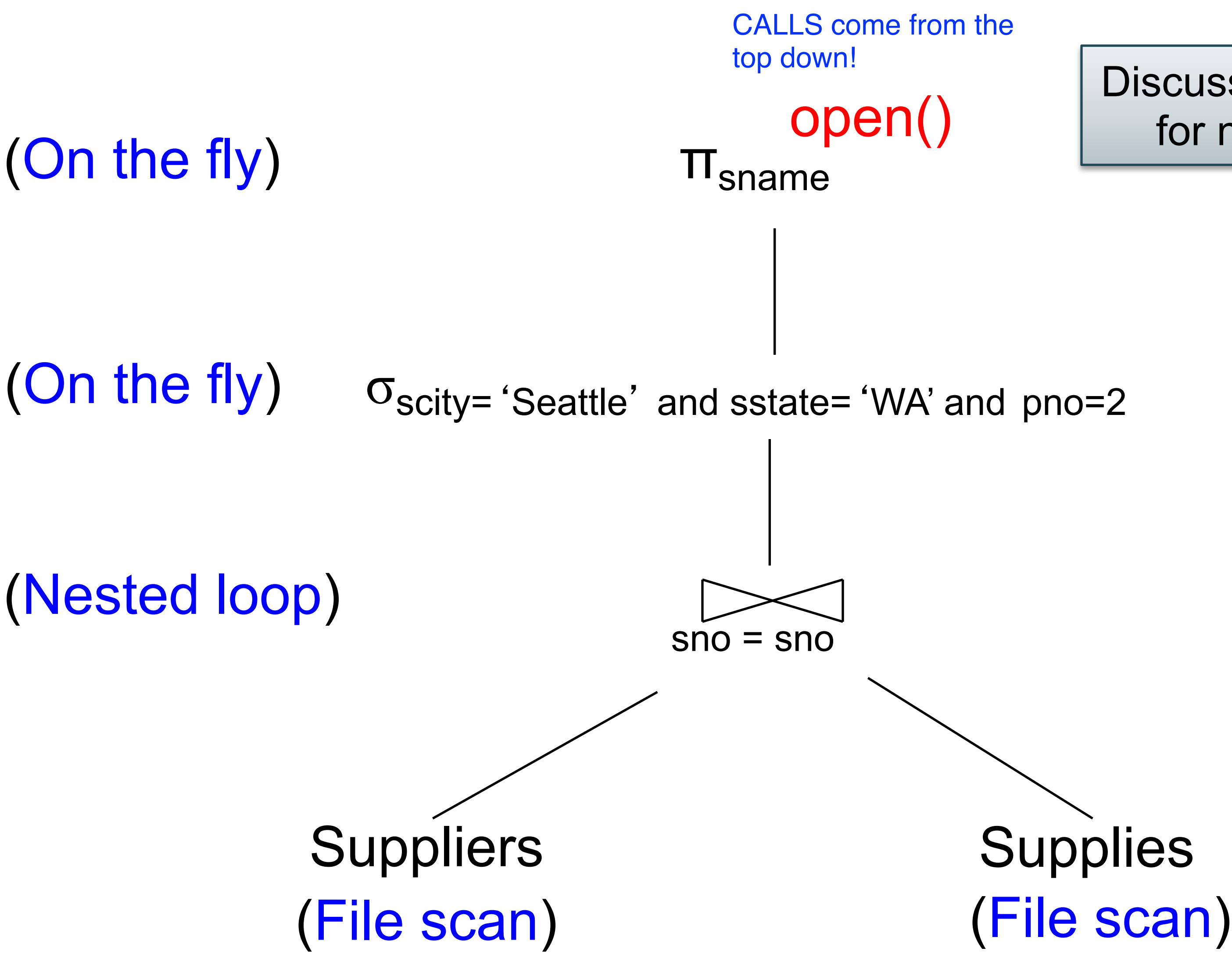


`Supplier(sid, sname, scity, sstate)`

`Supply(sid, pno, quantity)`

Pipelining

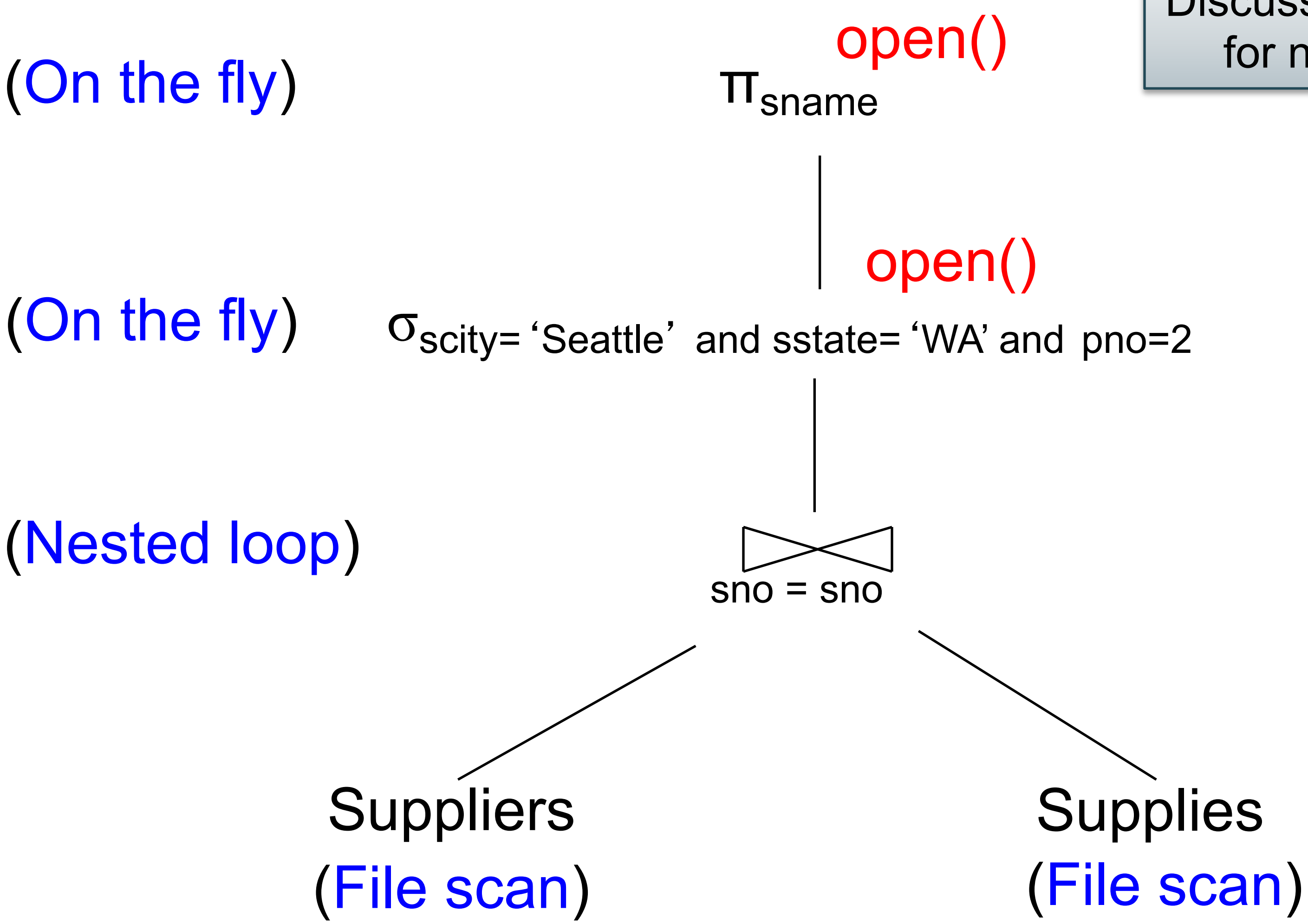
Discuss: open/next/close for nested loop join



Pipelining

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

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

(On the fly) π_{sname} open()

(On the fly) $\sigma_{scity=\text{'Seattle'}\ \text{and}\ sstate=\text{'WA'}\ \text{and}\ pno=2}$ open()

(Nested loop) $\bowtie_{sno = sno}$ open()

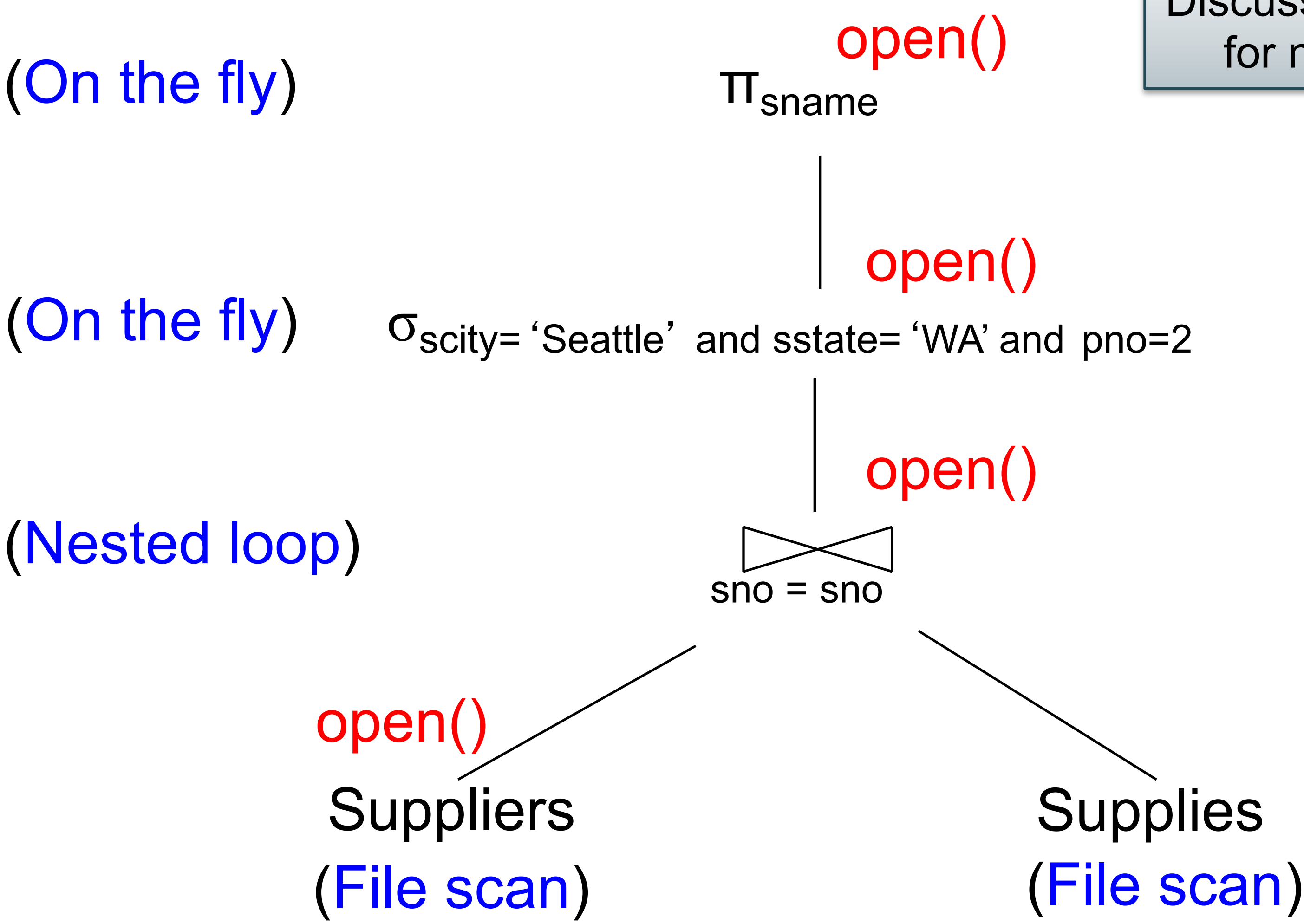
Suppliers (File scan)

Supplies (File scan)

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

Pipelining

Discuss: open/next/close for nested loop join

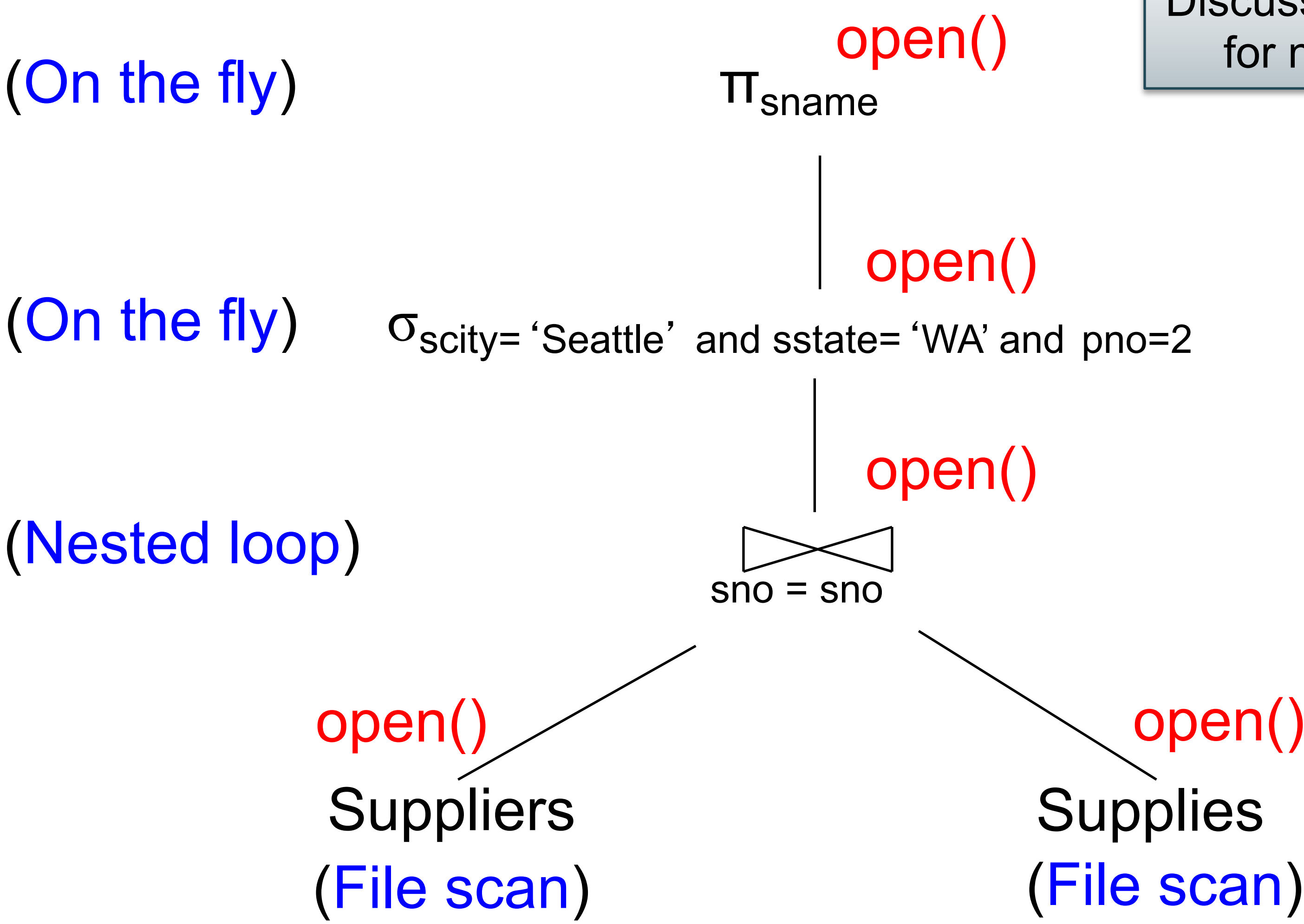


Supplier(sid, sname, scity, sstate)

Supply(sid, pno, quantity)

Pipelining

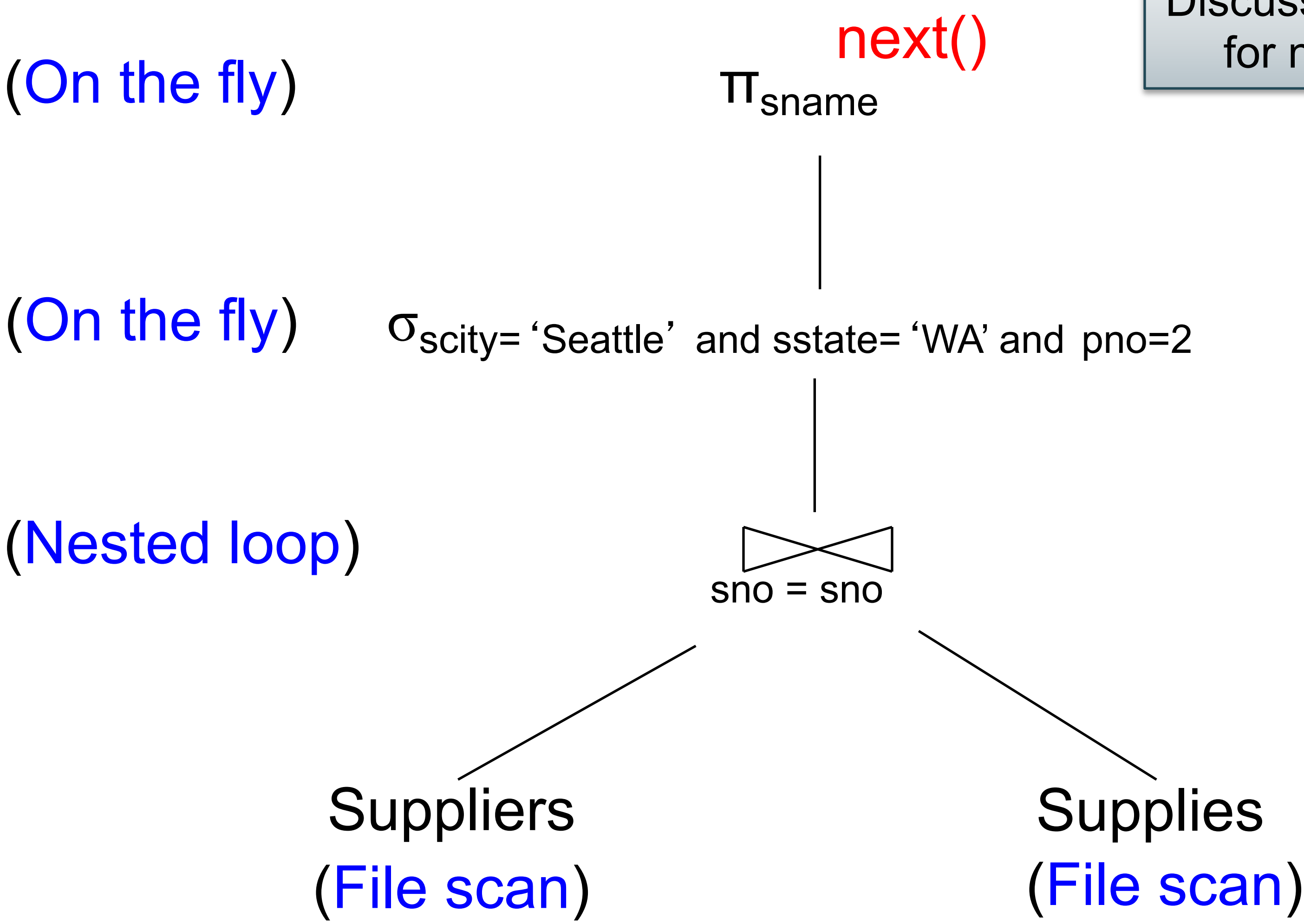
Discuss: open/next/close for nested loop join



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

Pipelining

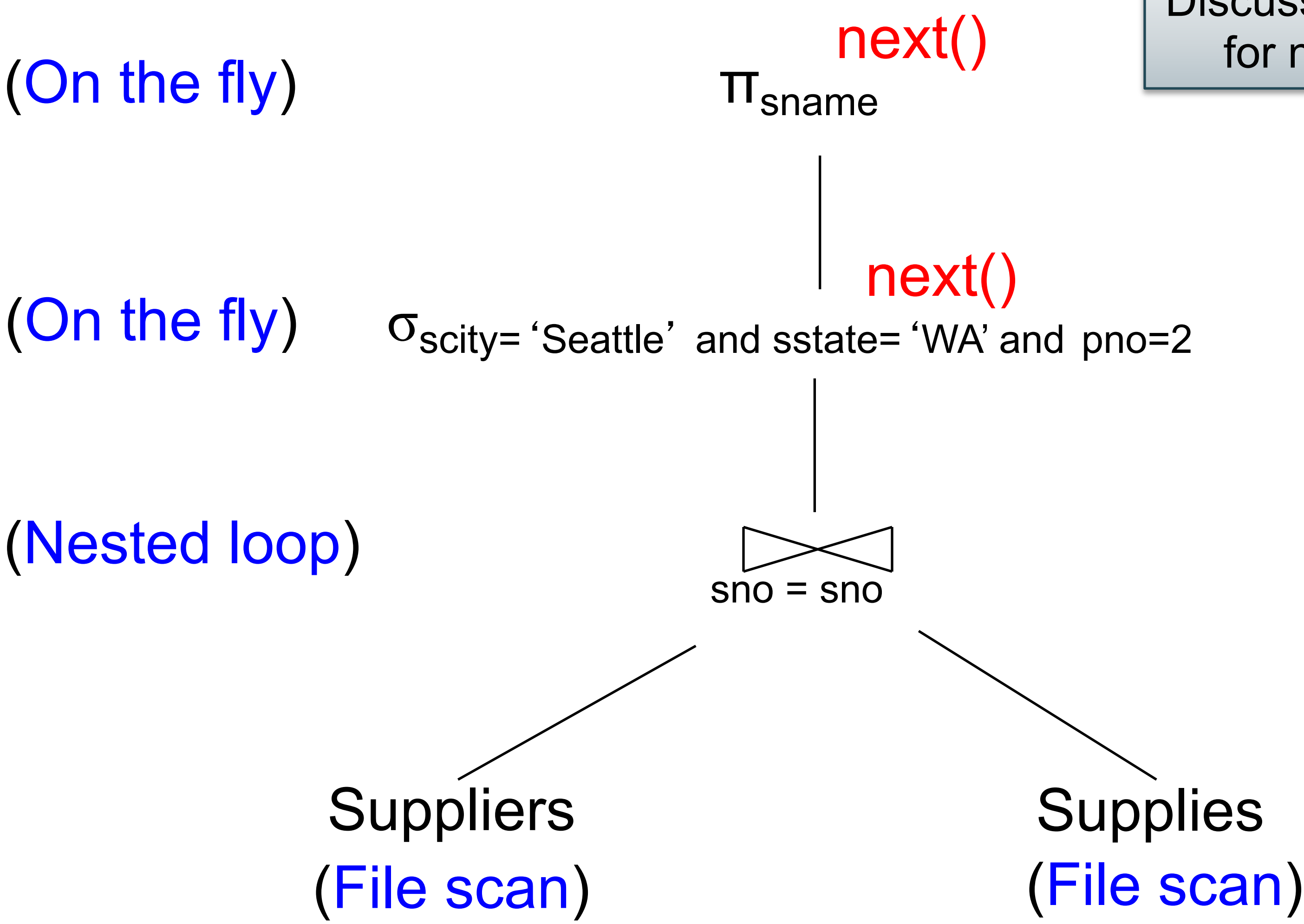
Discuss: open/next/close for nested loop join



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

Pipelining

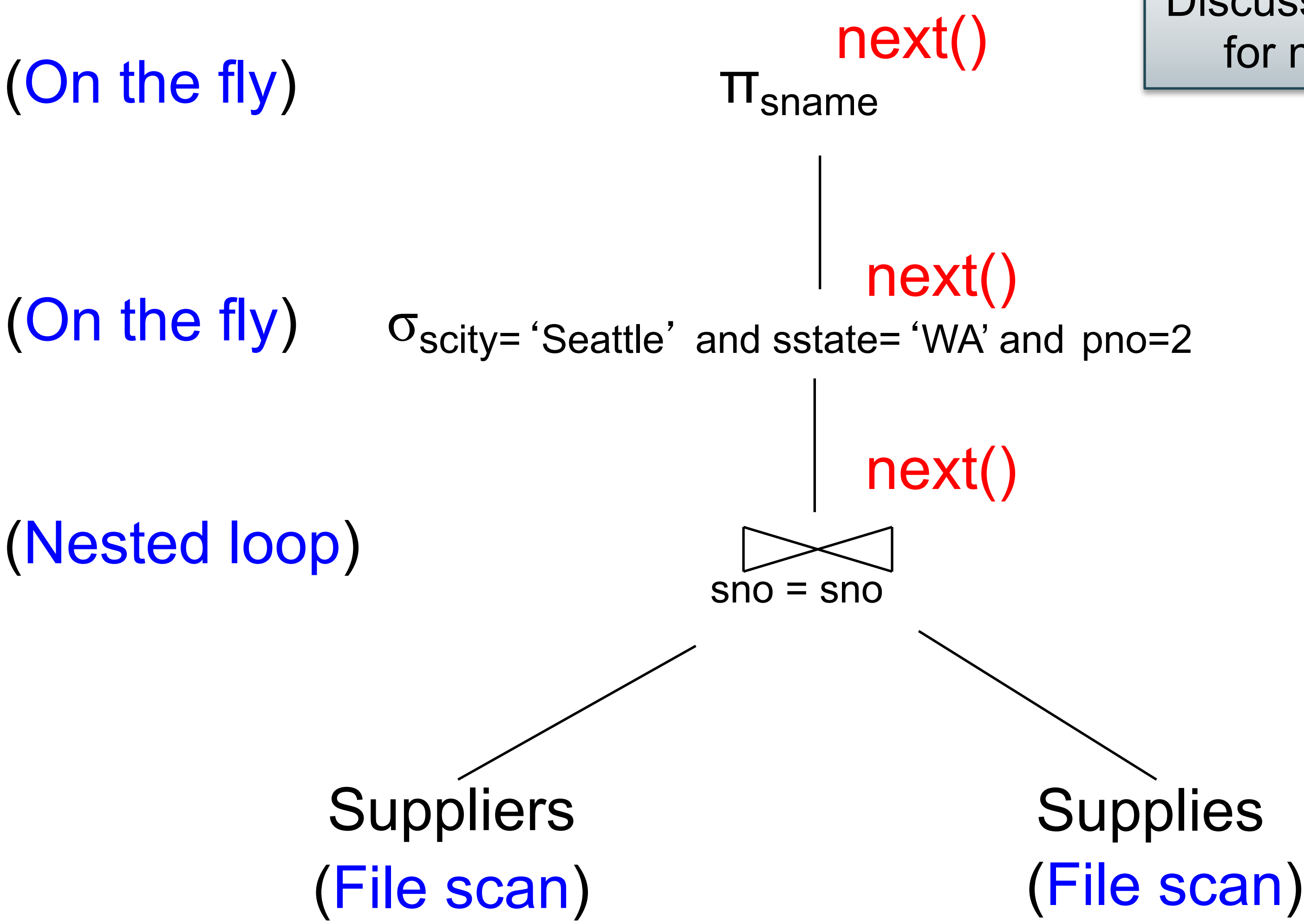
Discuss: open/next/close for nested loop join



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

Pipelining

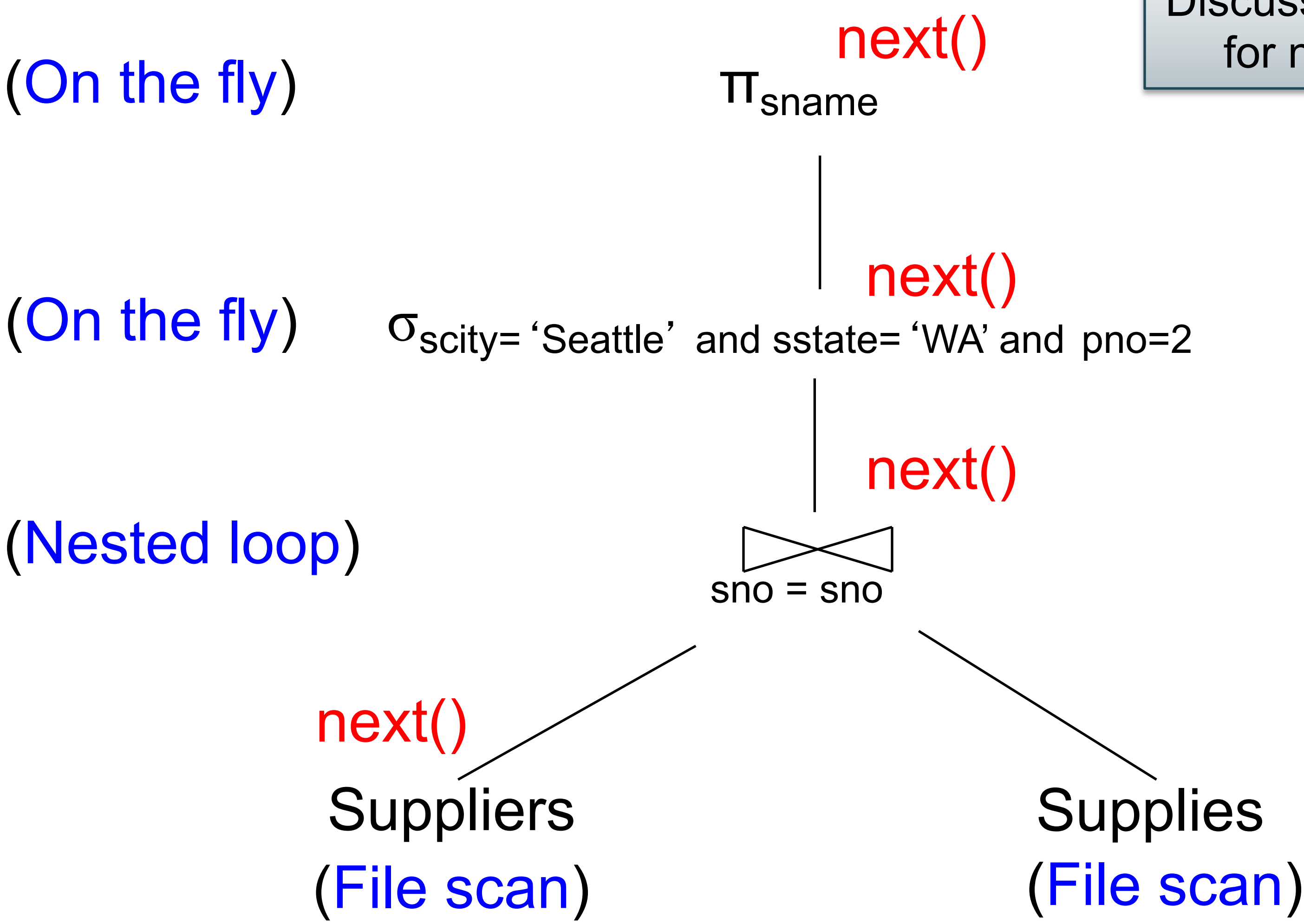
Discuss: open/next/close for nested loop join



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

Pipelining

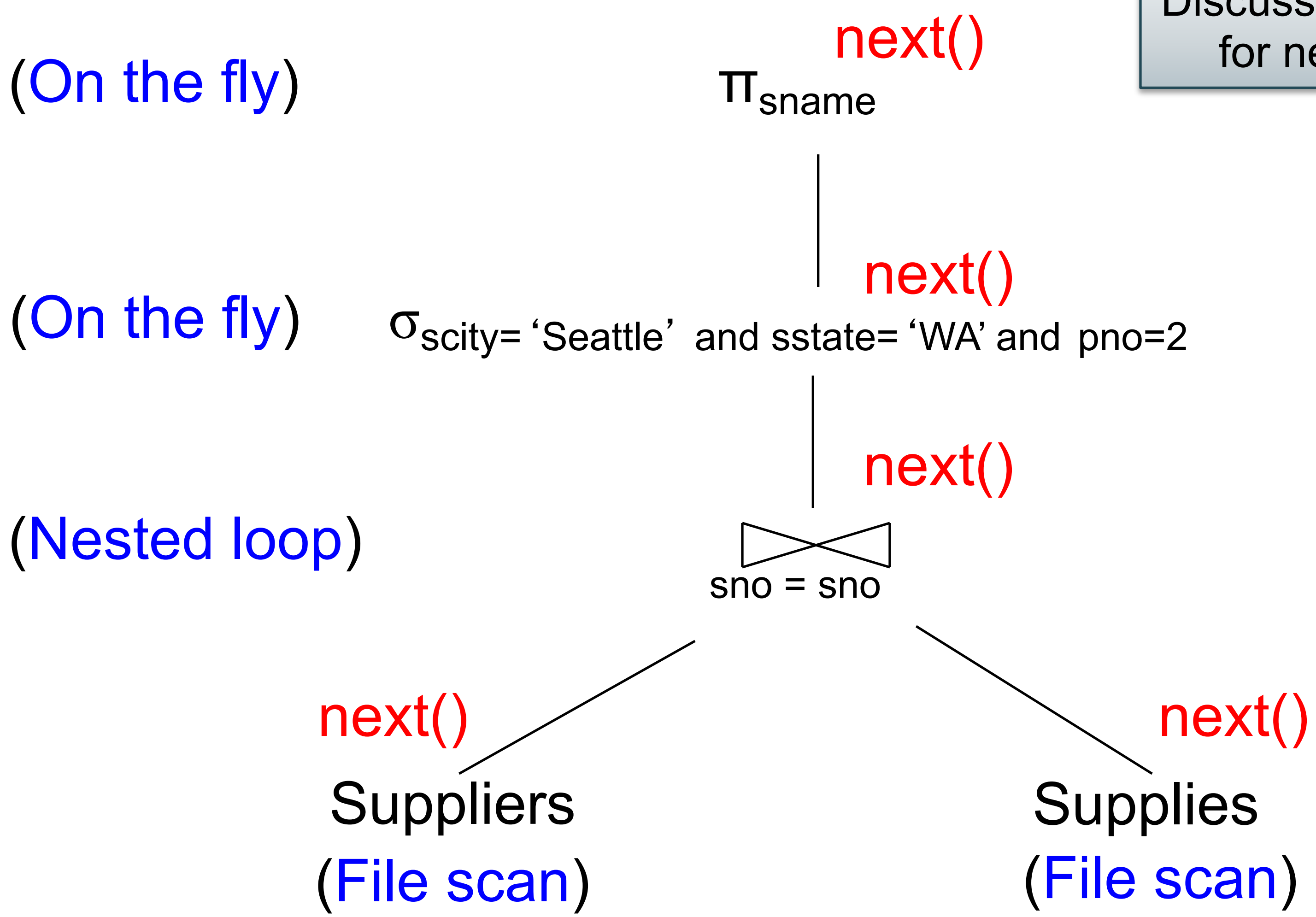
Discuss: open/next/close for nested loop join



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

Pipelining

Discuss: open/next/close for nested loop join

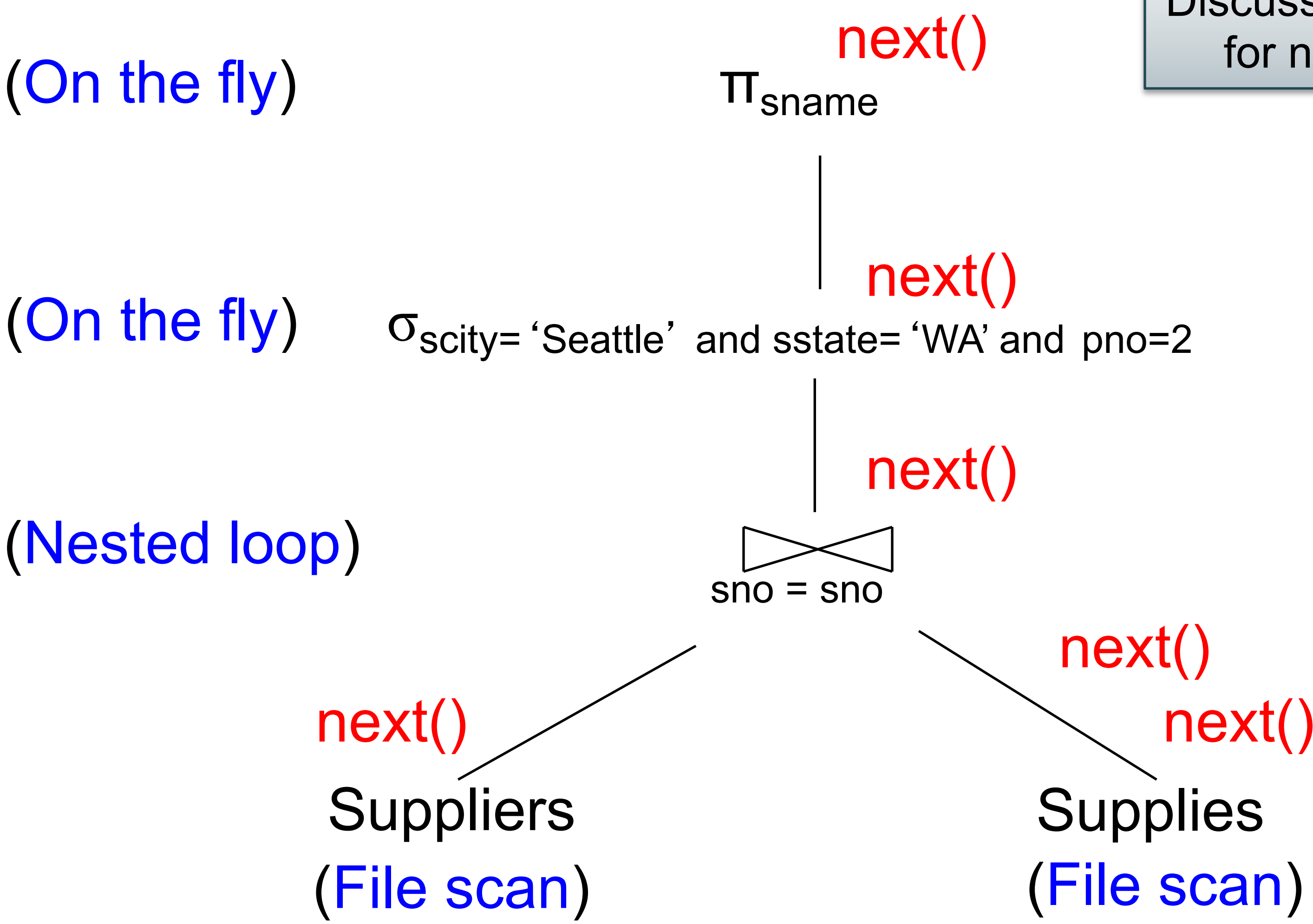


Supplier(sid, sname, scity, sstate)

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

(On the fly) π_{sname}

(On the fly) $\sigma_{scity=\text{'Seattle'} \text{ and } sstate=\text{'WA'} \text{ and } pno=2}$

(Hash Join) $\bowtie_{sno = sno}$

Does this change how search is carried out?
YES!

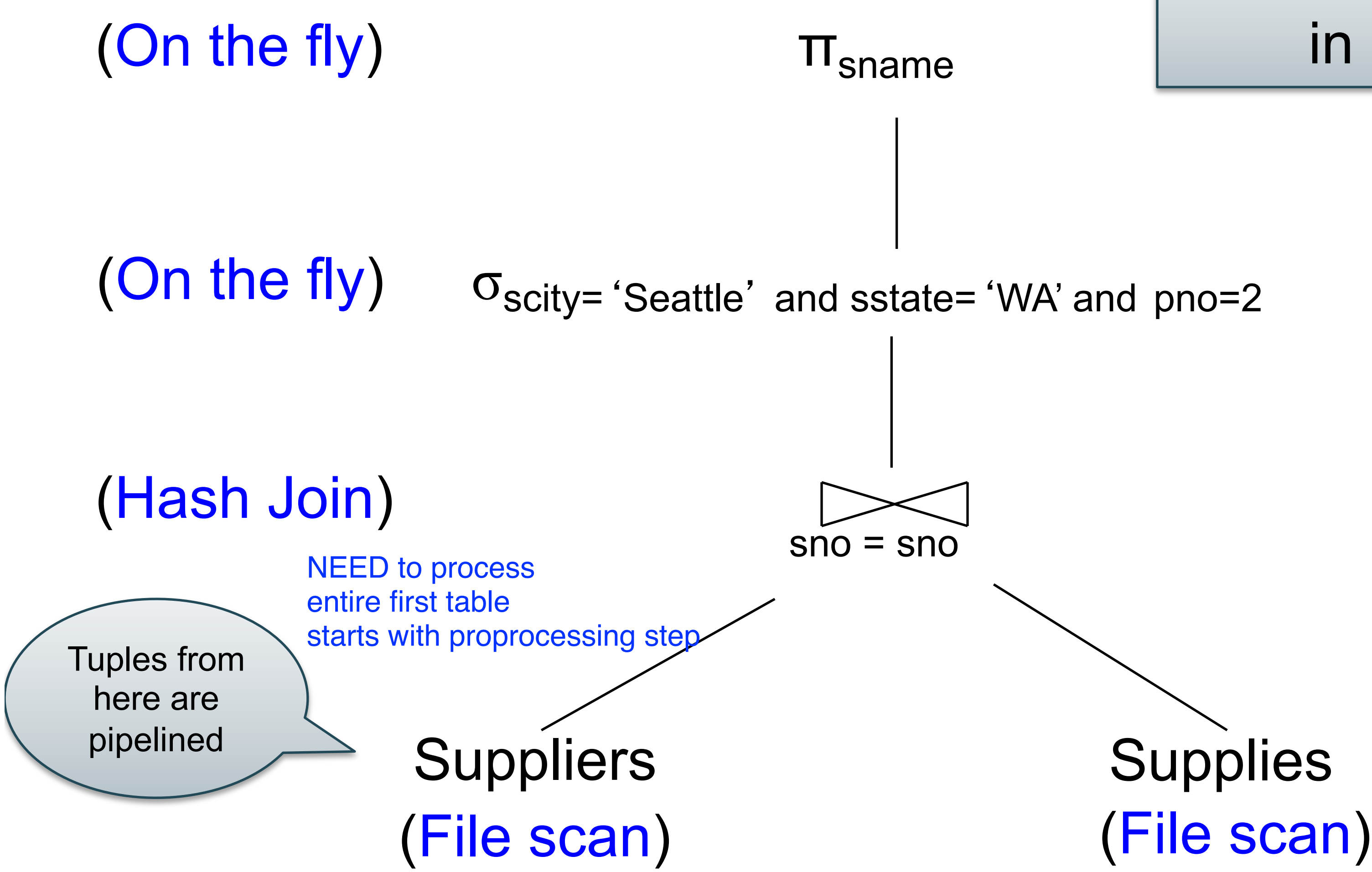
Suppliers (File scan)

Supplies (File scan)

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

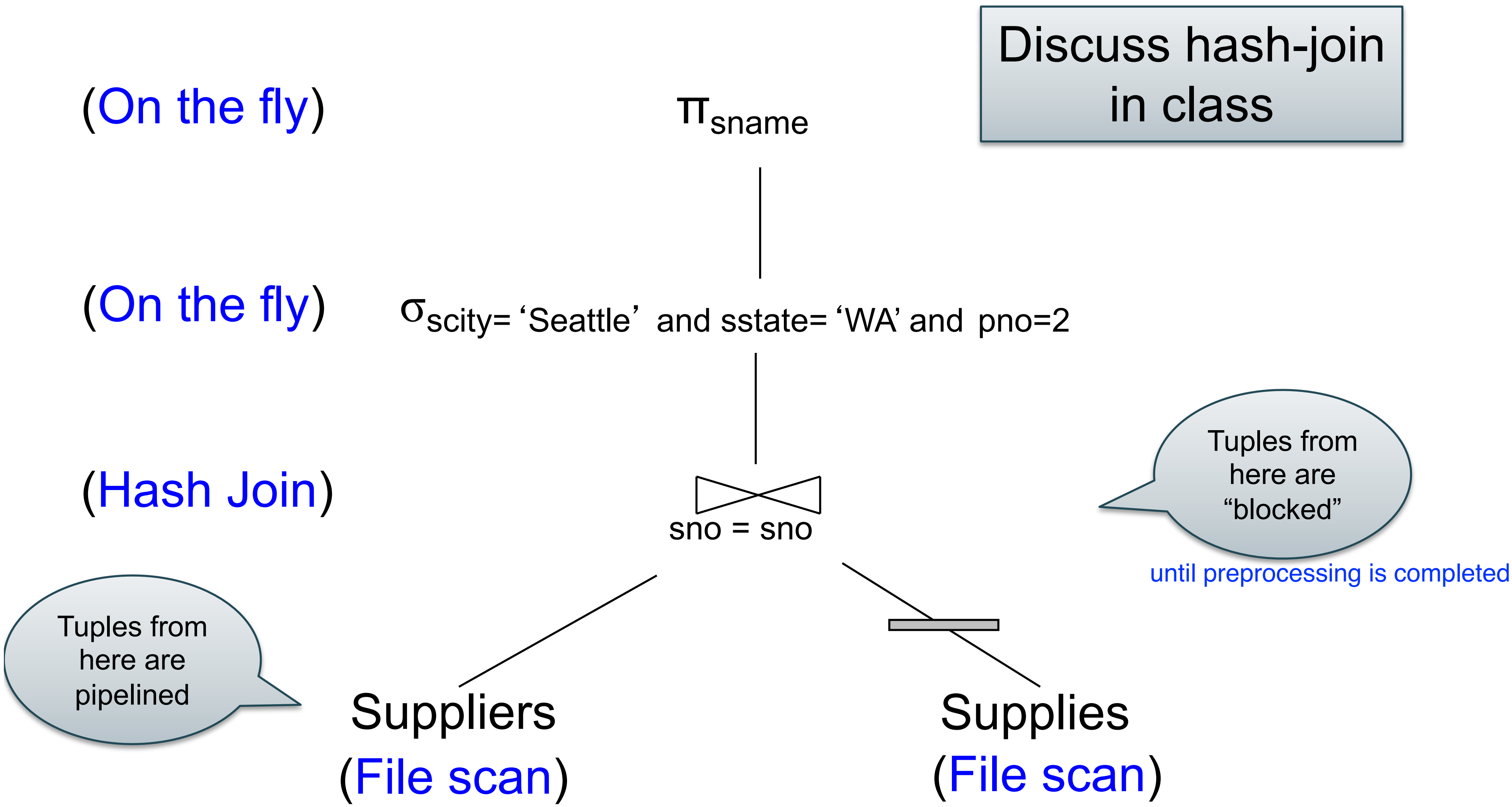
Pipelining

Discuss hash-join in class



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

Pipelining



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

Discussion on Physical Plan

ALL THESE HAVE TO BE DECIDED AS PART OF FORMING THE PHYSICAL QUERY PLAN

More components of a physical plan:

general filescan or using indexes?

- **Access path selection** for each relation
  – Scan the relation or use an index (next lecture)
- **Implementation choice** for each operator
  – Nested loop join, hash join, etc.
- **Scheduling decisions** for operators
  – Pipelined execution or intermediate materialization

BLOCKING