CSE 344

FEBRUARY 12TH – RDBMS INTERNALS

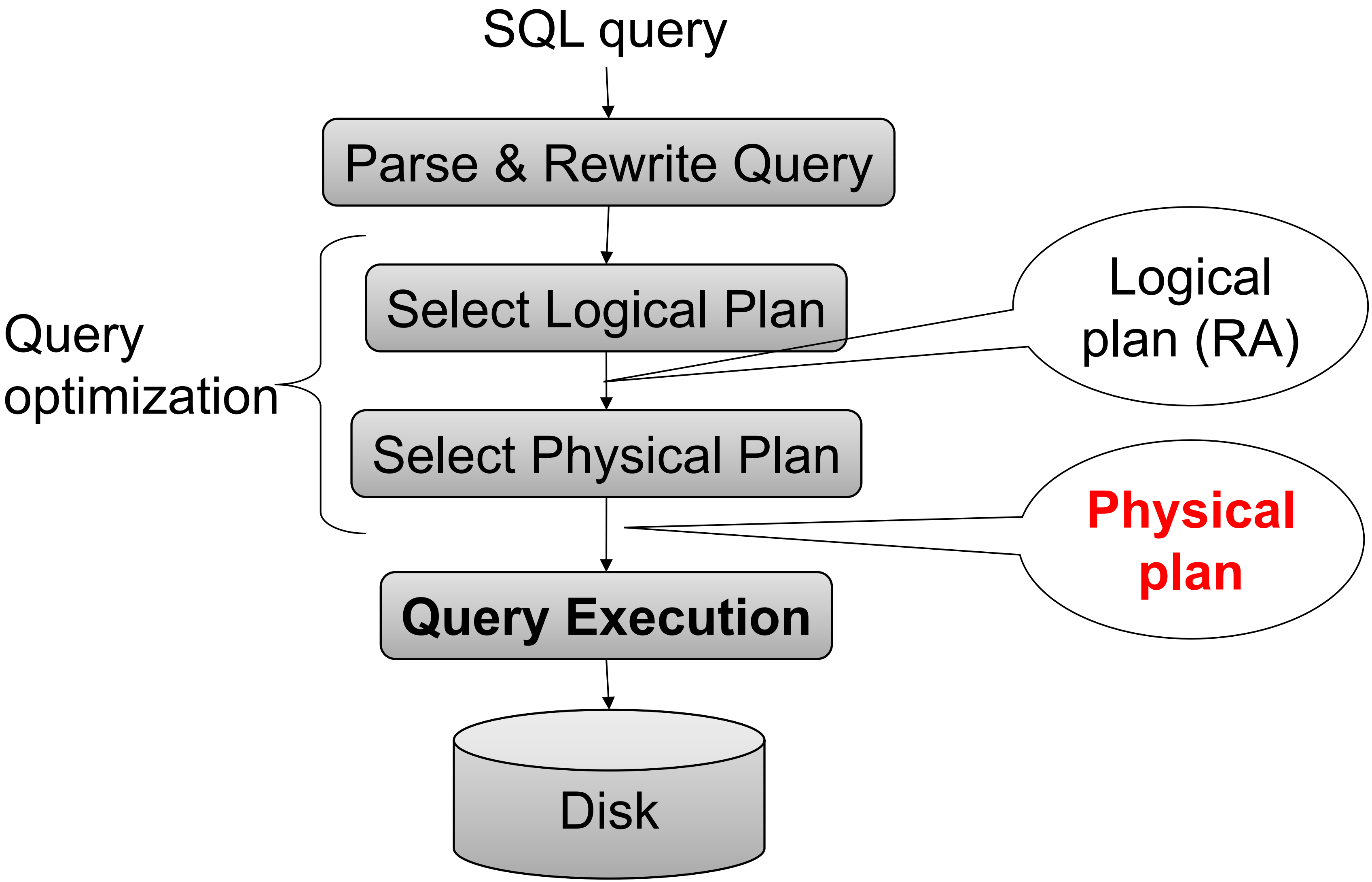
ADMINISTRIVIA

- HW5 out tonight
- OQ5 out Wednesday
- Both due February 21 (11:30 & 11:00)
- Exam grades on canvas by Wednesday
- Handed back in section on Thursday

TODAY

- **Back to RDBMS**
  - ”Query plans” and DBMS planning
  - Management between SQL and execution
  - Optimization techniques
  - Indexing and data arrangement

QUERY EVALUATION STEPS



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

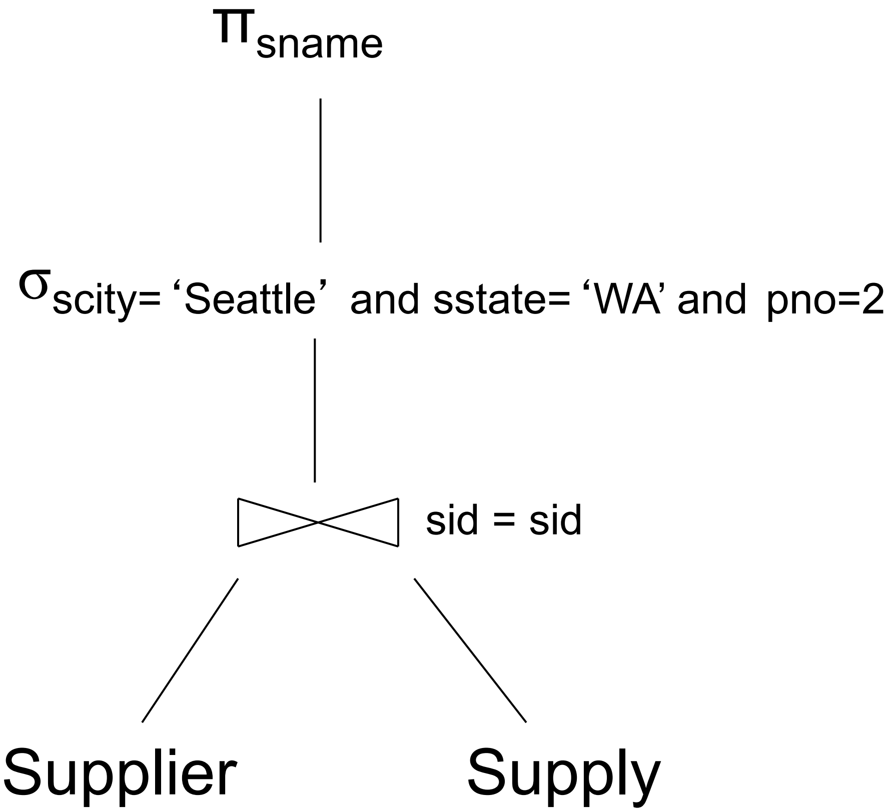
Physical plans:

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

REVIEW: RELATIONAL ALGEBRA

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

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



π_{sname}

$\sigma_{scity=\text{‘Seattle’ and sstate=‘WA’ and pno=2}}$

$\bowtie$ sid = sid

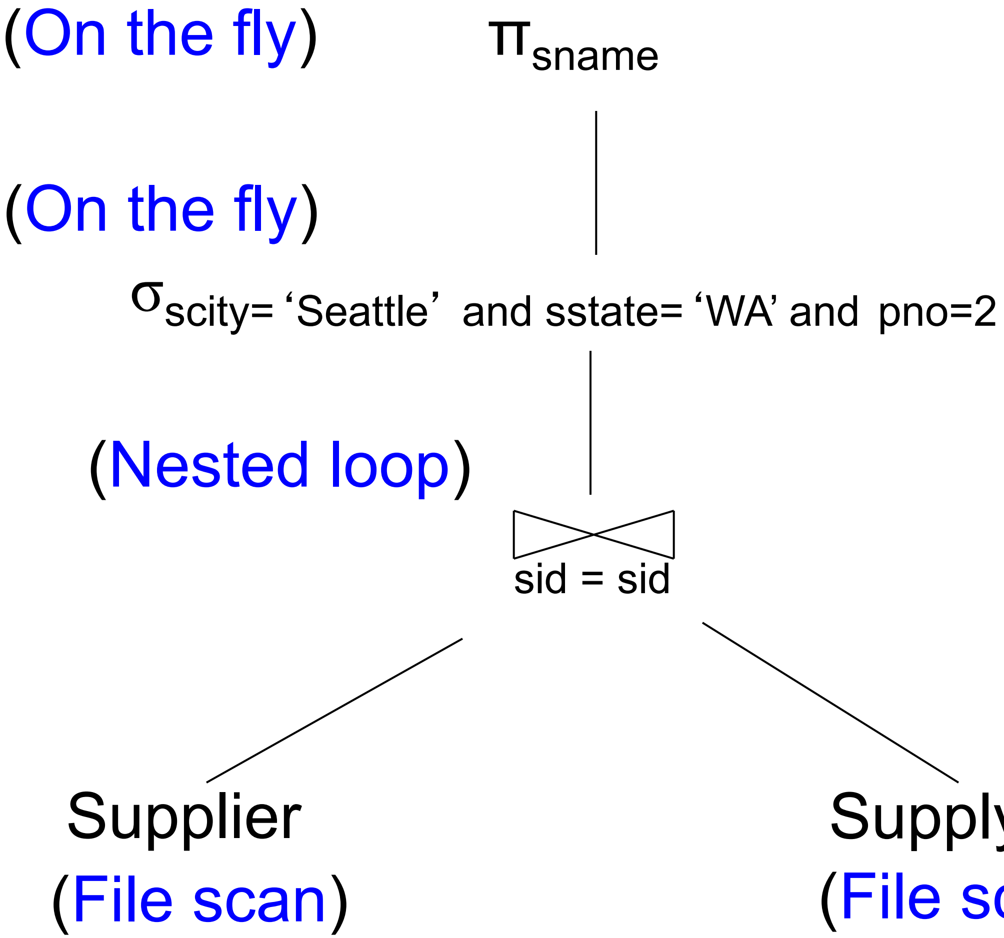
Supplier

Supply

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

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

PHYSICAL QUERY PLAN 1



(On the fly) π_{sname}

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

(Nested loop) $\bowtie_{sid = sid}$

Supplier (File scan)

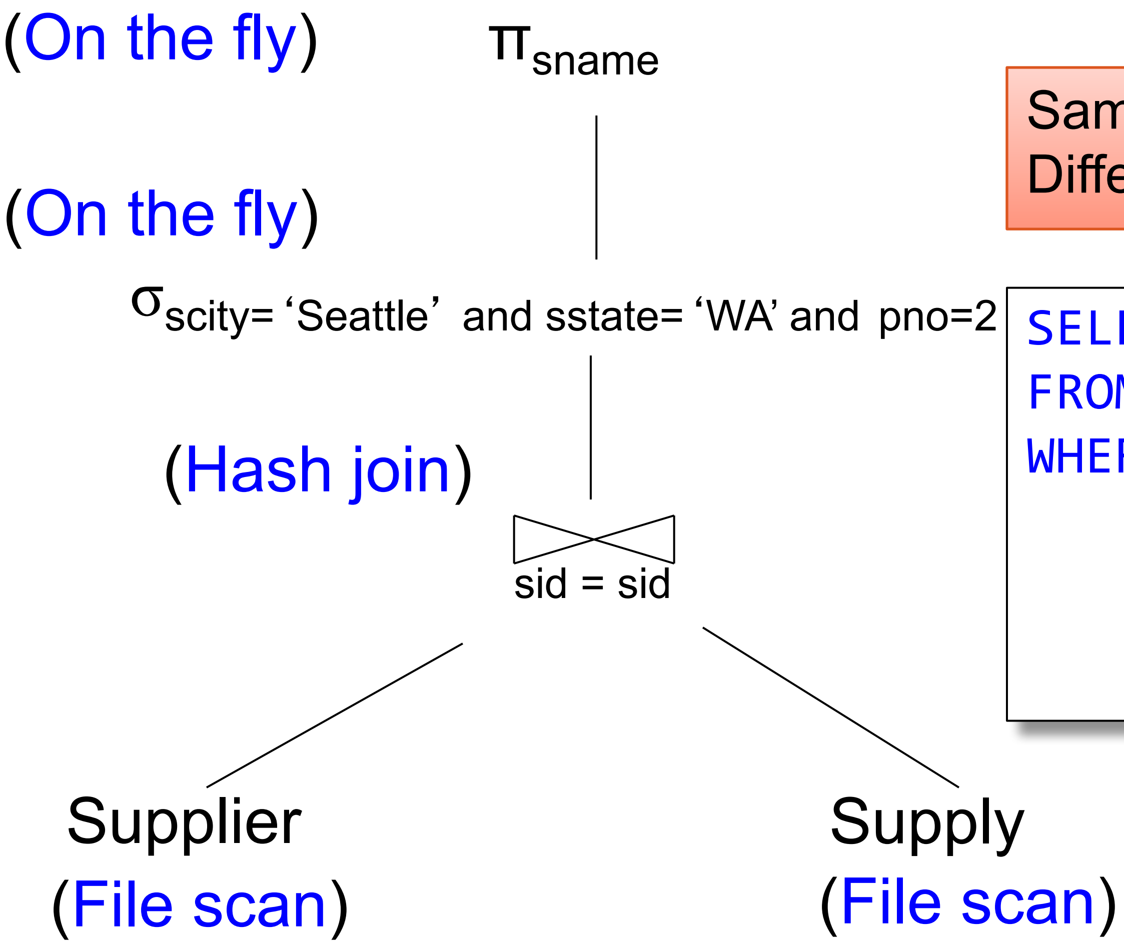
Supply (File scan)

A physical query plan is a logical query plan annotated with physical implementation details

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

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

PHYSICAL QUERY PLAN 2



(On the fly) π_{sname}

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

(Hash join) $\bowtie_{sid = sid}$

Supplier (File scan)

Supply (File scan)

Same logical query plan
Different physical plan

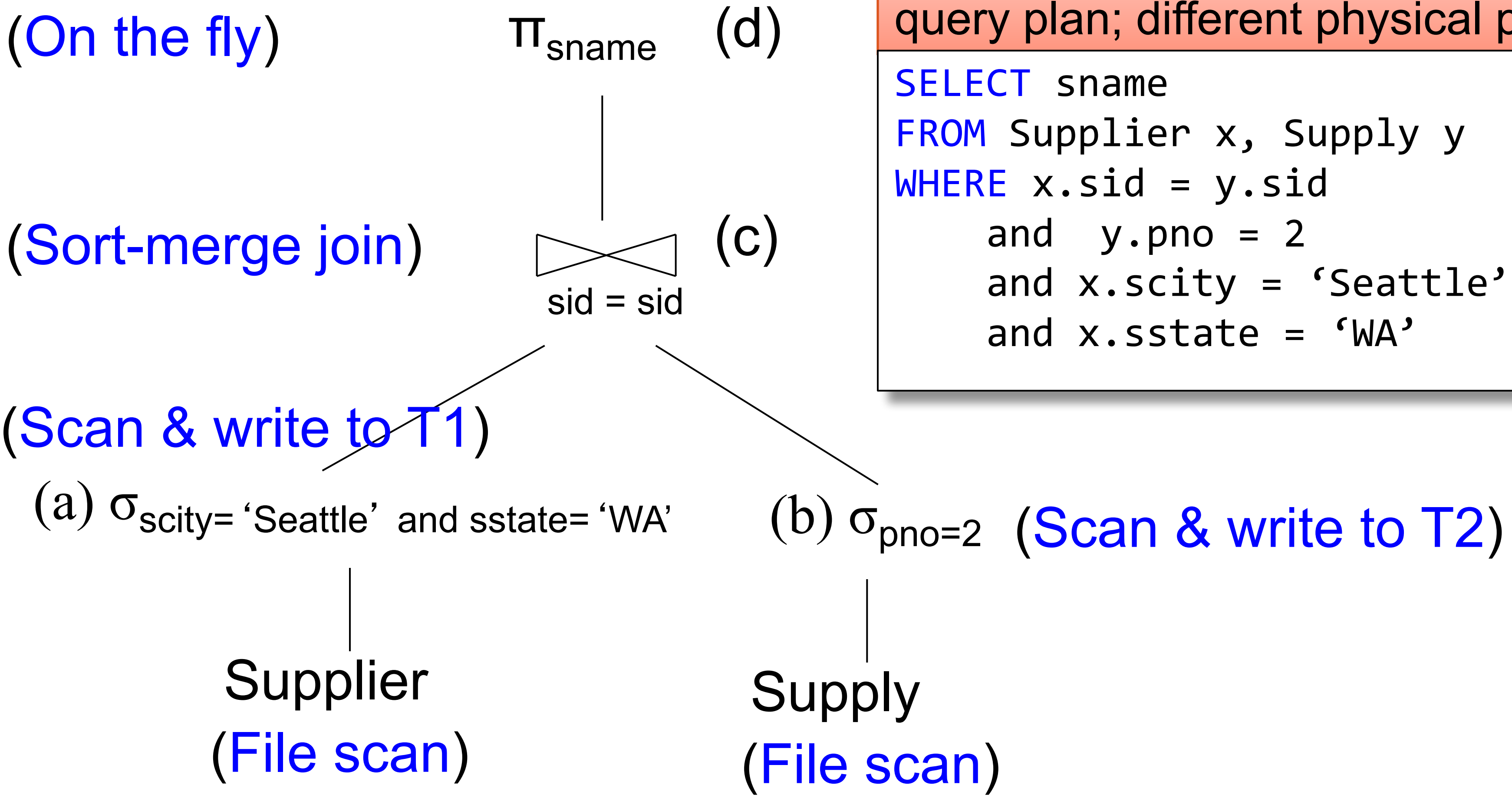
```
SELECT sname
FROM Supplier x, Supply y
WHERE x.sid = y.sid
    and  y.pno = 2
    and x.scity = 'Seattle'
    and x.sstate = 'WA'
```

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

PHYSICAL QUERY PLAN 3

Different but equivalent logical query plan; different physical plan

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



(On the fly) π_{sname} (d)

(Sort-merge join) $\bowtie_{sid = sid}$ (c)

(Scan & write to T1) (a) $\sigma_{scity=\text{‘Seattle’ and sstate=‘WA’}}$

(b) $\sigma_{pno=2}$ (Scan & write to T2)

Supplier (File scan)

Supply (File scan)

QUERY OPTIMIZATION PROBLEM

For each SQL query… many logical plans

For each logical plan… many physical plans

Next: we will discuss physical operators;
how exactly are query executed?

PHYSICAL OPERATORS

Each of the logical operators may have one or more implementations = physical operators

Will discuss several basic physical operators, with a focus on join

```
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

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

1. **Nested Loop Join** O(??)
2. **Merge join** O(??)
3. **Hash join** O(??)

`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.	**Nested Loop Join**	**$O(n^2)$**
2.	**Merge join**	**$O(n \log n)$**
3.	**Hash join**	**$O(n) \ldots O(n^2)$**

BRIEF REVIEW OF HASH TABLES

A (naïve) hash function:

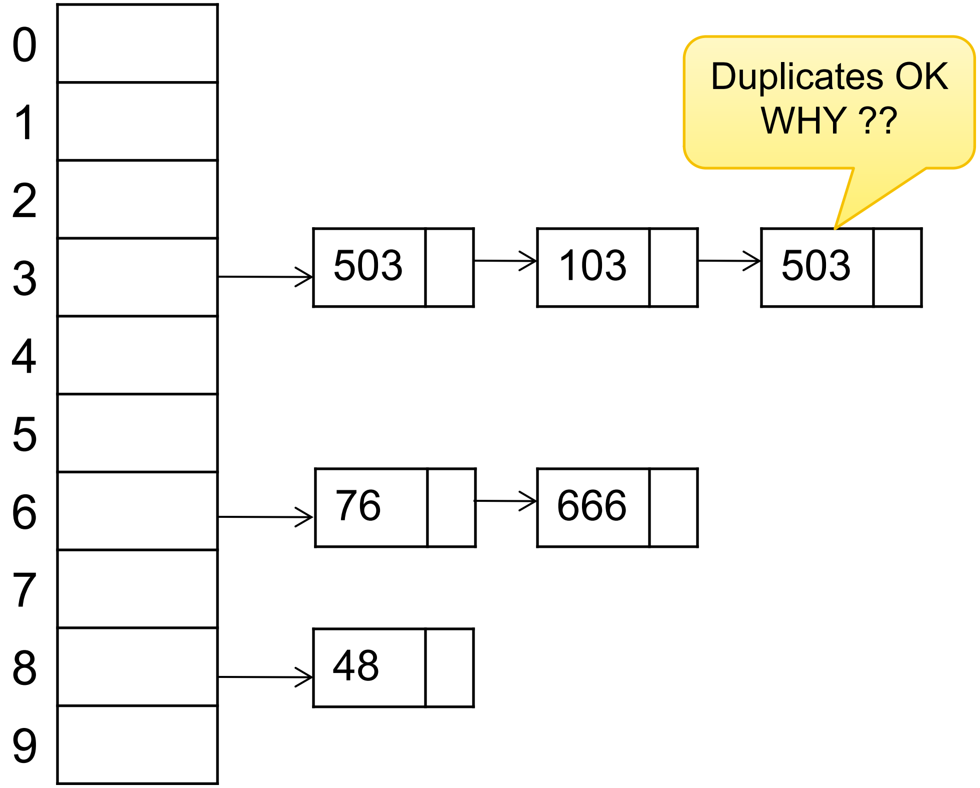
h(x) = x mod 10

Operations:

find(103) = ??

insert(488) = ??

Separate chaining:



Duplicates OK
WHY ??

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

ITERATOR INTERFACE

Each operator implements three methods:

open()

next()

close()

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 ();
}
```

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;
  }










}
```

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 () {








  }

}
```

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);
    }

  }
}
```

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;
  }

}
```

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(); }
}
```

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

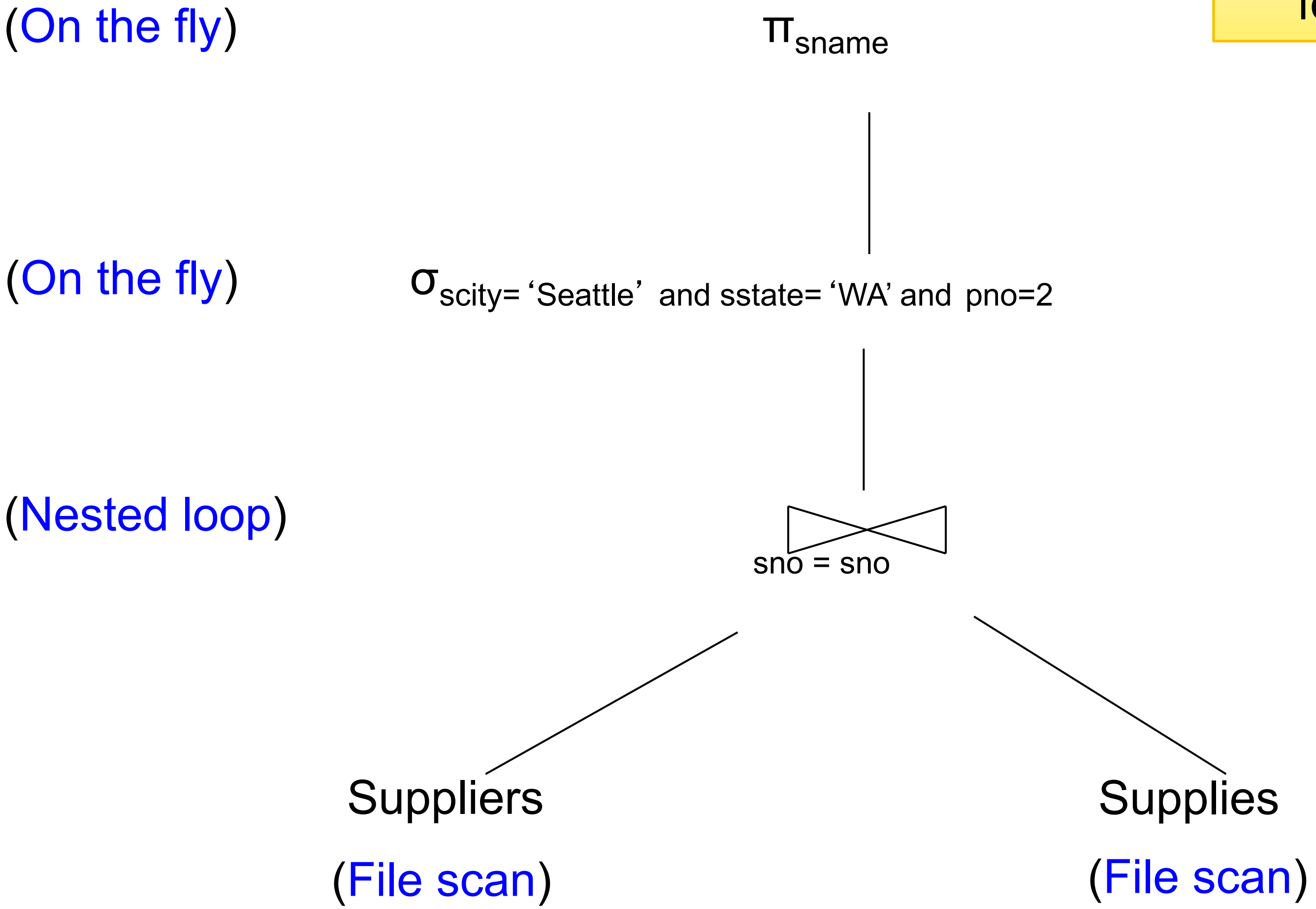
```
Operator q = parse("SELECT ...");
q = optimize(q);

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) π_{sname}

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

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

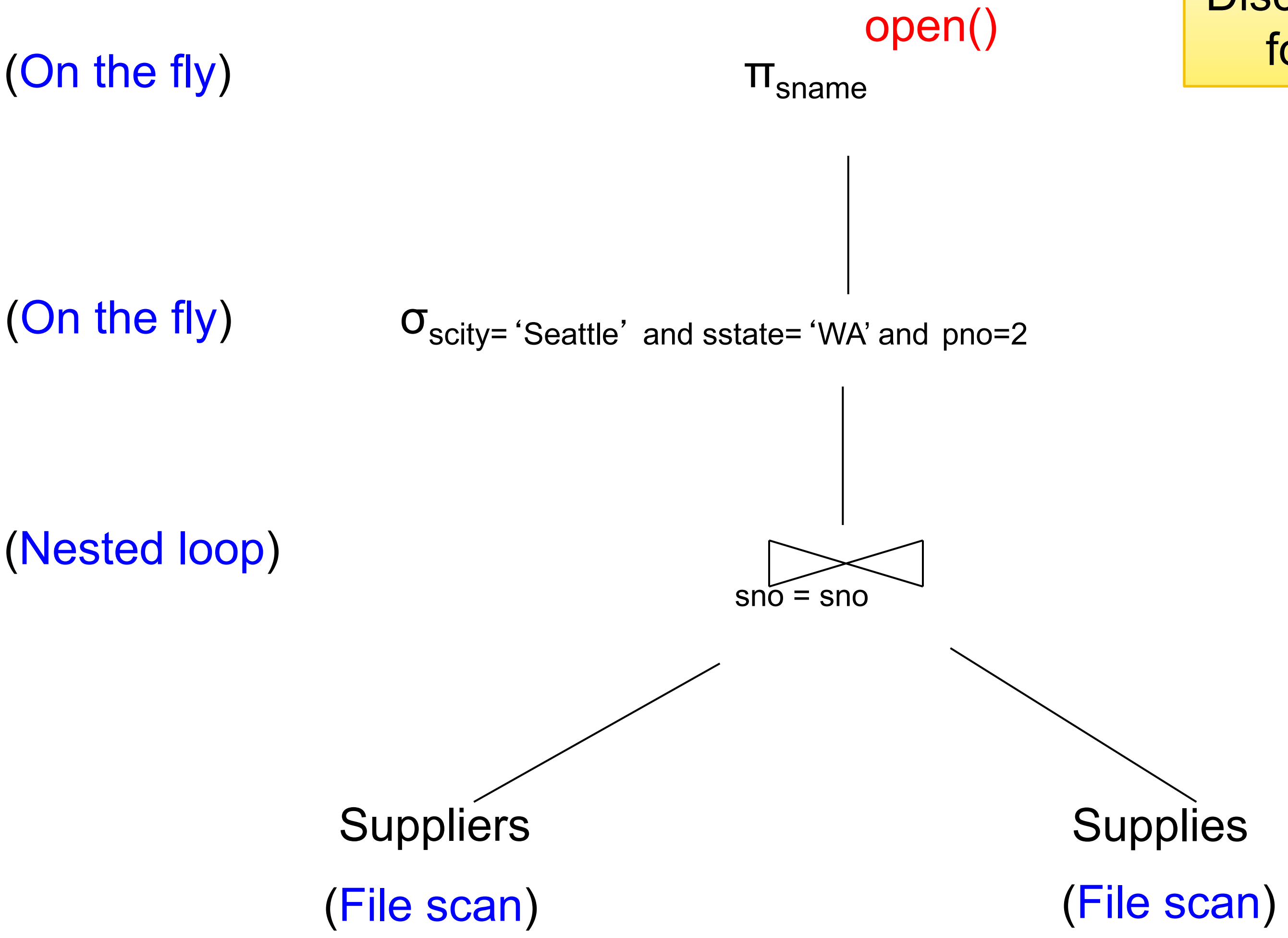
Suppliers (File scan)

Supplies (File scan)

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

PIPELINING

Discuss: open/next/close for nested loop join



open()

(On the fly) π_{sname}

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

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

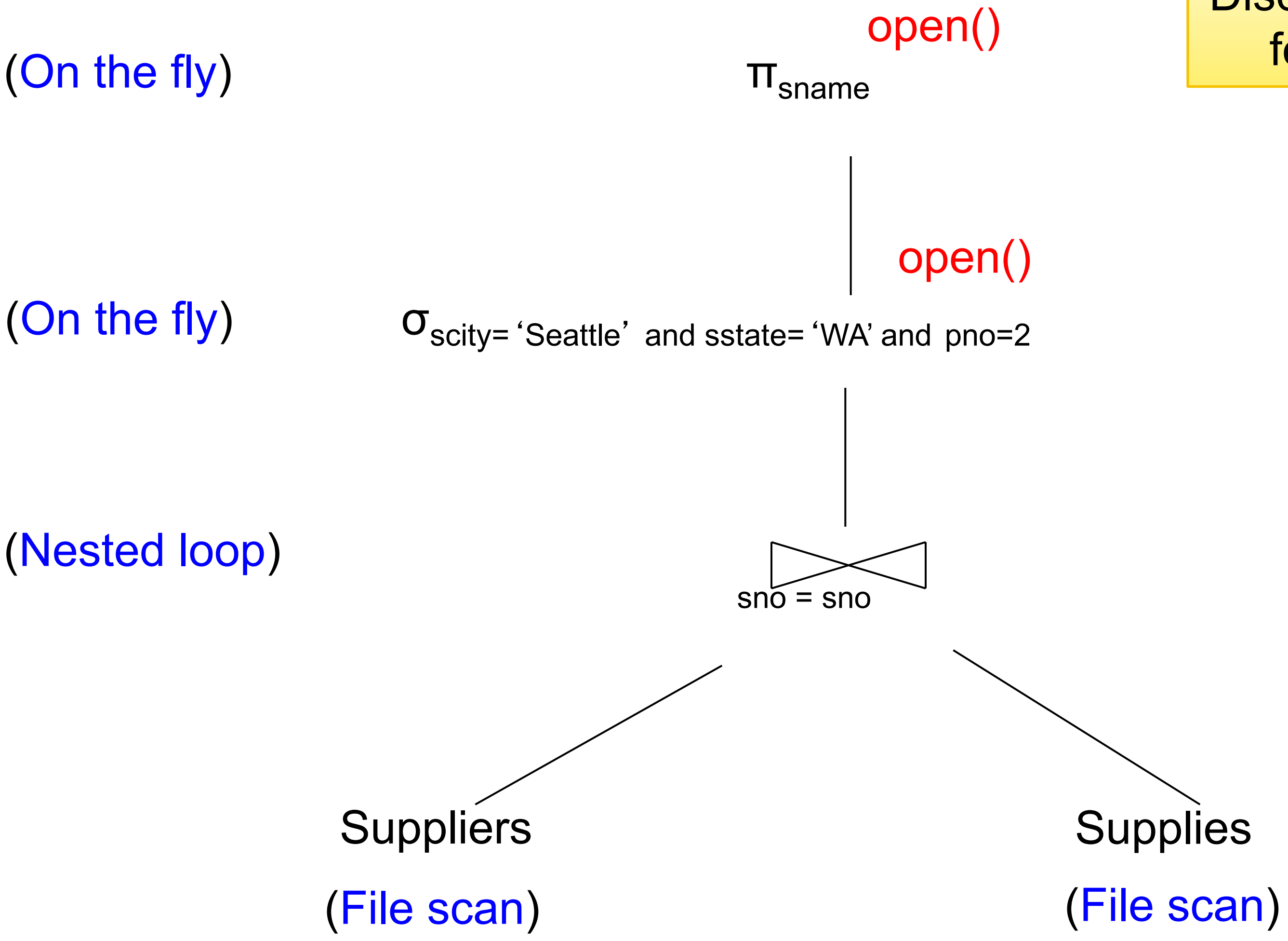
Suppliers (File scan)

Supplies (File scan)

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

PIPELINING

Discuss: open/next/close for nested loop join



(On the fly)

open()

π_{sname}

open()

(On the fly)

$\sigma_{scity=\text{'Seattle'} \text{ and } sstate=\text{'WA'} \text{ and } pno=2}$

(Nested loop)

$\bowtie_{sno = sno}$

Suppliers

(File scan)

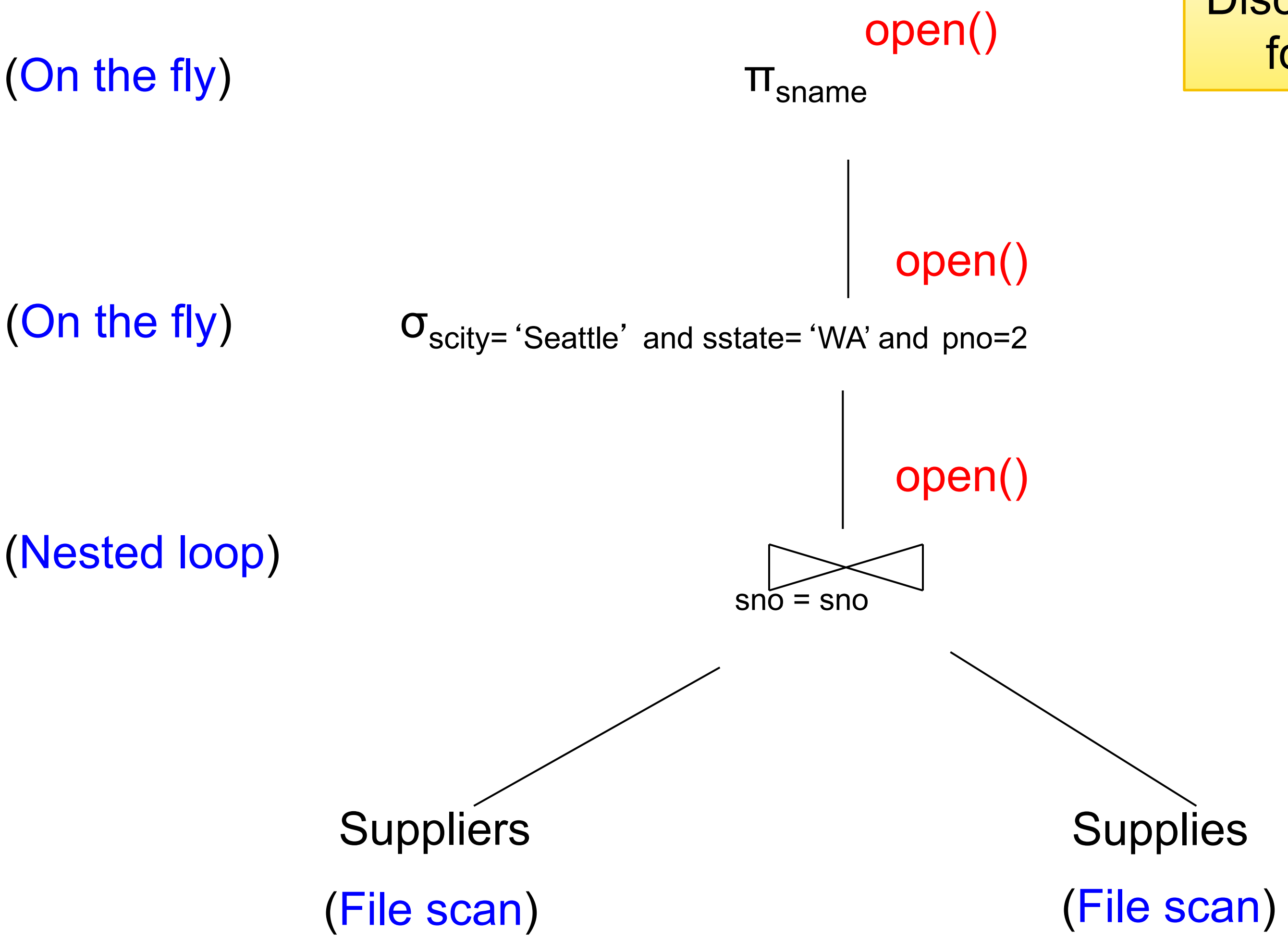
Supplies

(File scan)

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

PIPELINING

Discuss: open/next/close for nested loop join



open()

(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) $\bowtie_{sno = sno}$

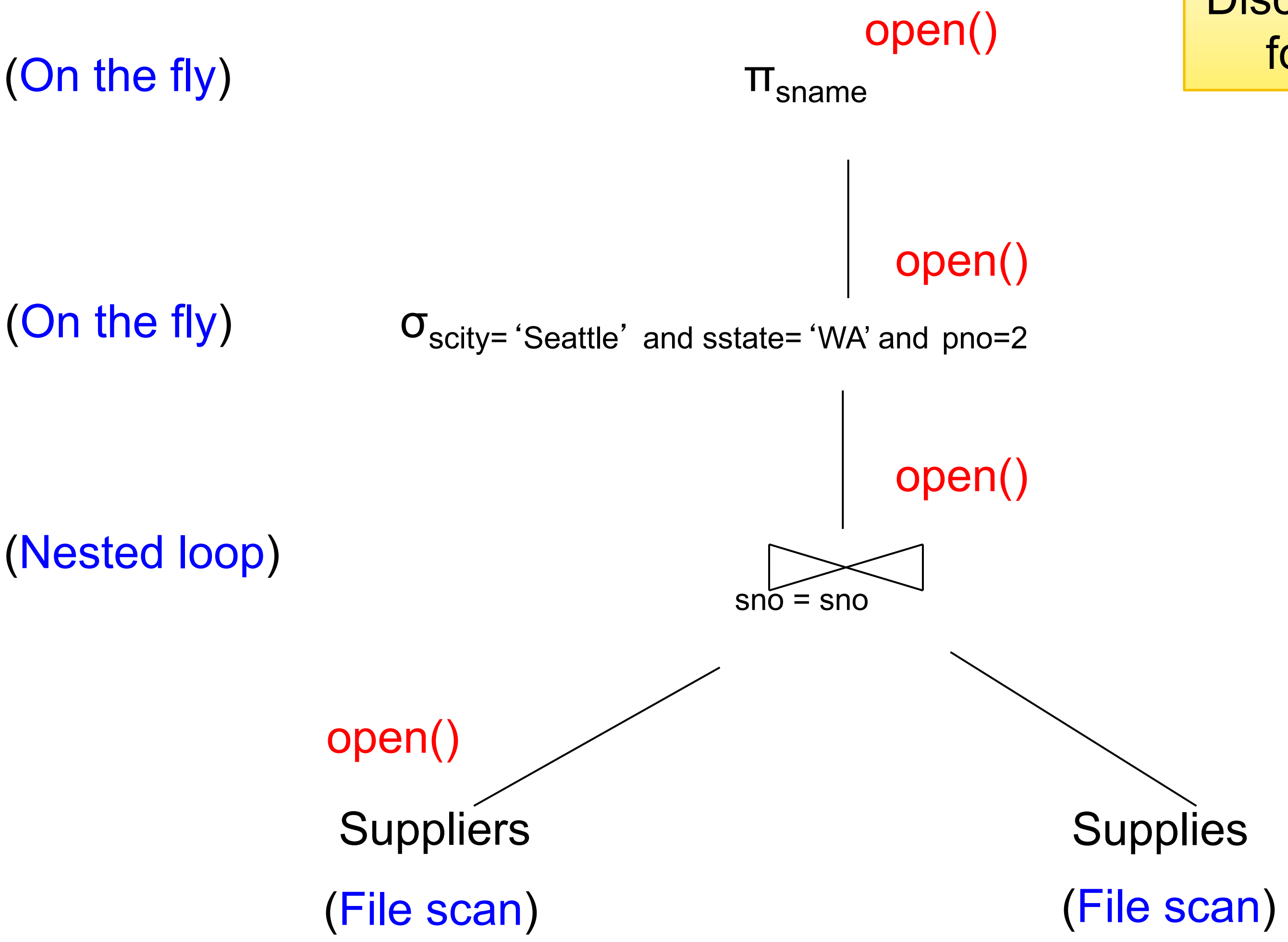
Suppliers (File scan)

Supplies (File scan)

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

PIPELINING

Discuss: open/next/close for nested loop join



open()

(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()

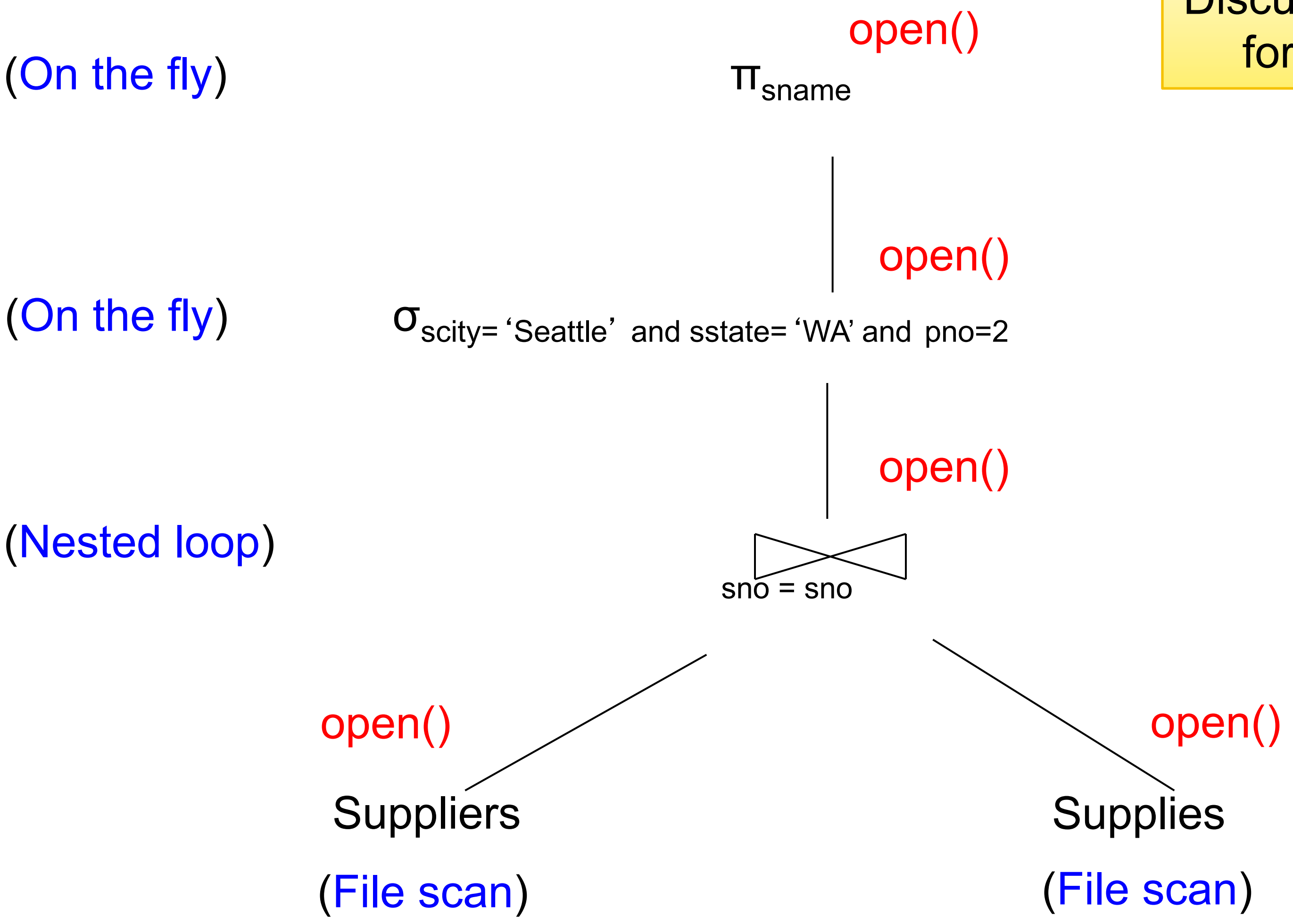
Suppliers (File scan)

Supplies (File scan)

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

PIPELINING

Discuss: open/next/close for nested loop join



open()

(On the fly) π_{sname}

open()

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

open()

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

open()

Suppliers

(File scan)

open()

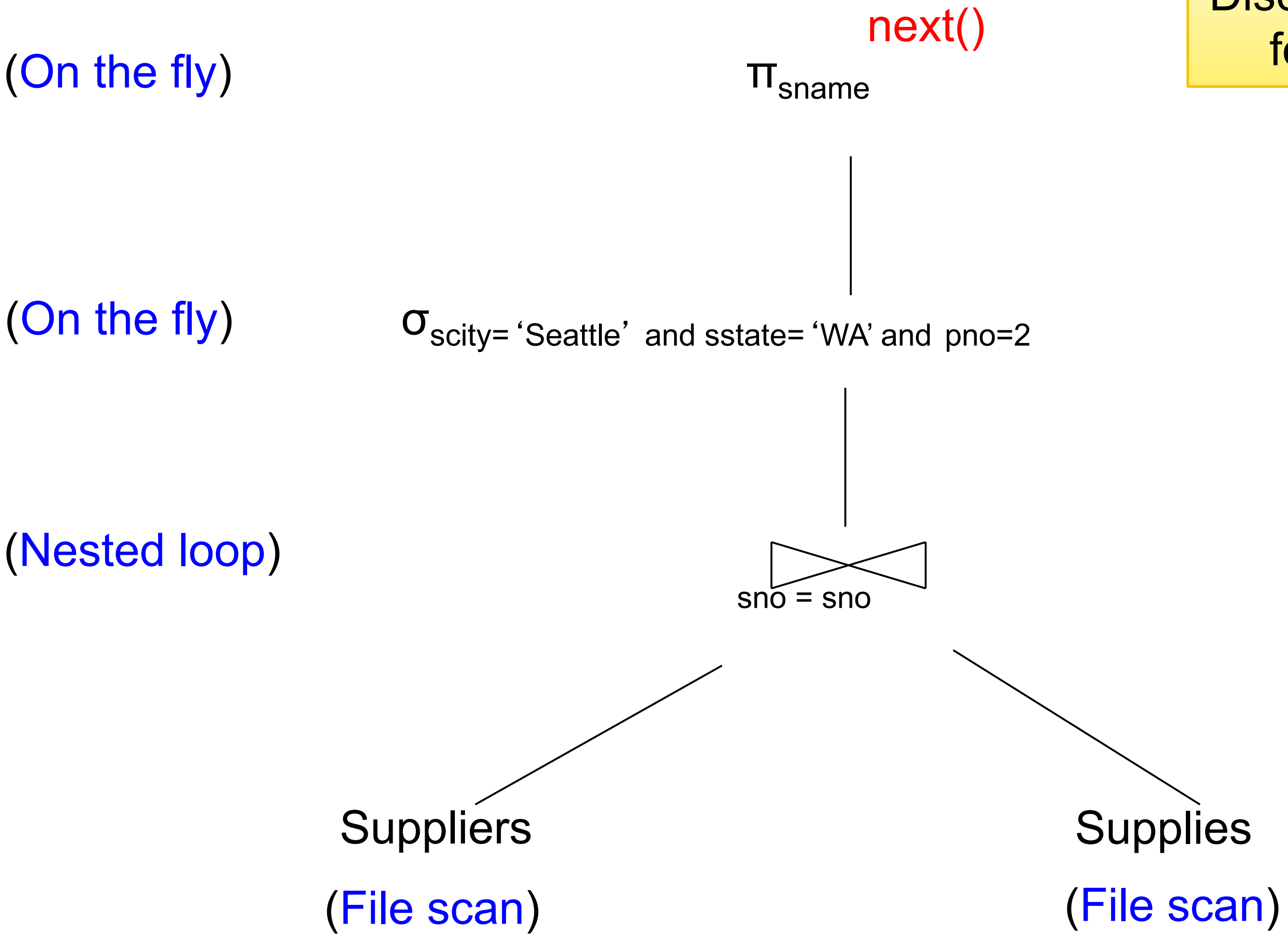
Supplies

(File scan)

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

PIPELINING

Discuss: open/next/close for nested loop join



next()

(On the fly) π_{sname}

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

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

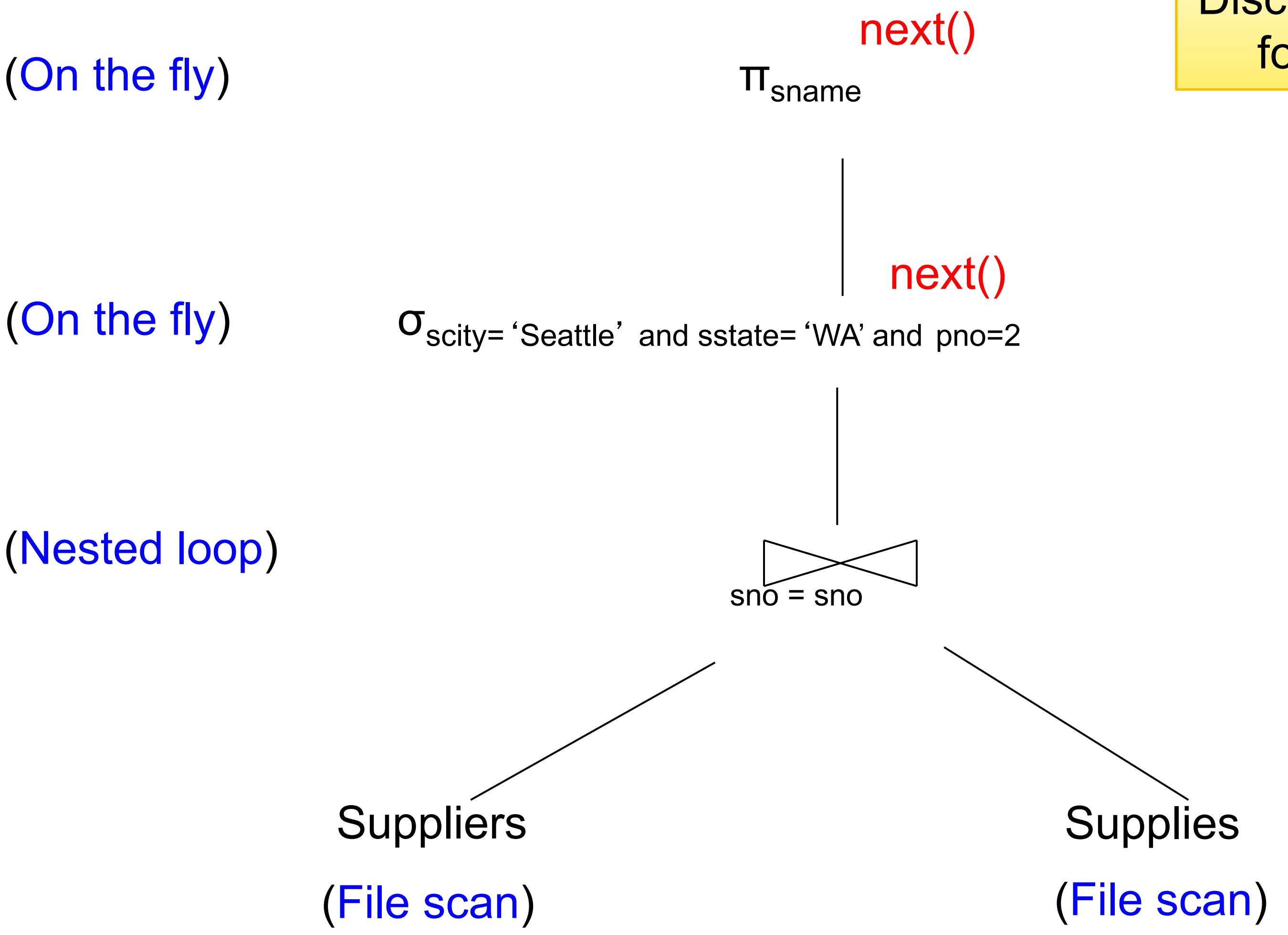
Suppliers (File scan)

Supplies (File scan)

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

PIPELINING

Discuss: open/next/close for nested loop join



(On the fly)

next()

π_{sname}

(On the fly)

next()

$\sigma_{scity=\text{'Seattle'} \text{ and } sstate=\text{'WA'} \text{ and } pno=2}$

(Nested loop)

$\bowtie_{sno = sno}$

Suppliers

(File scan)

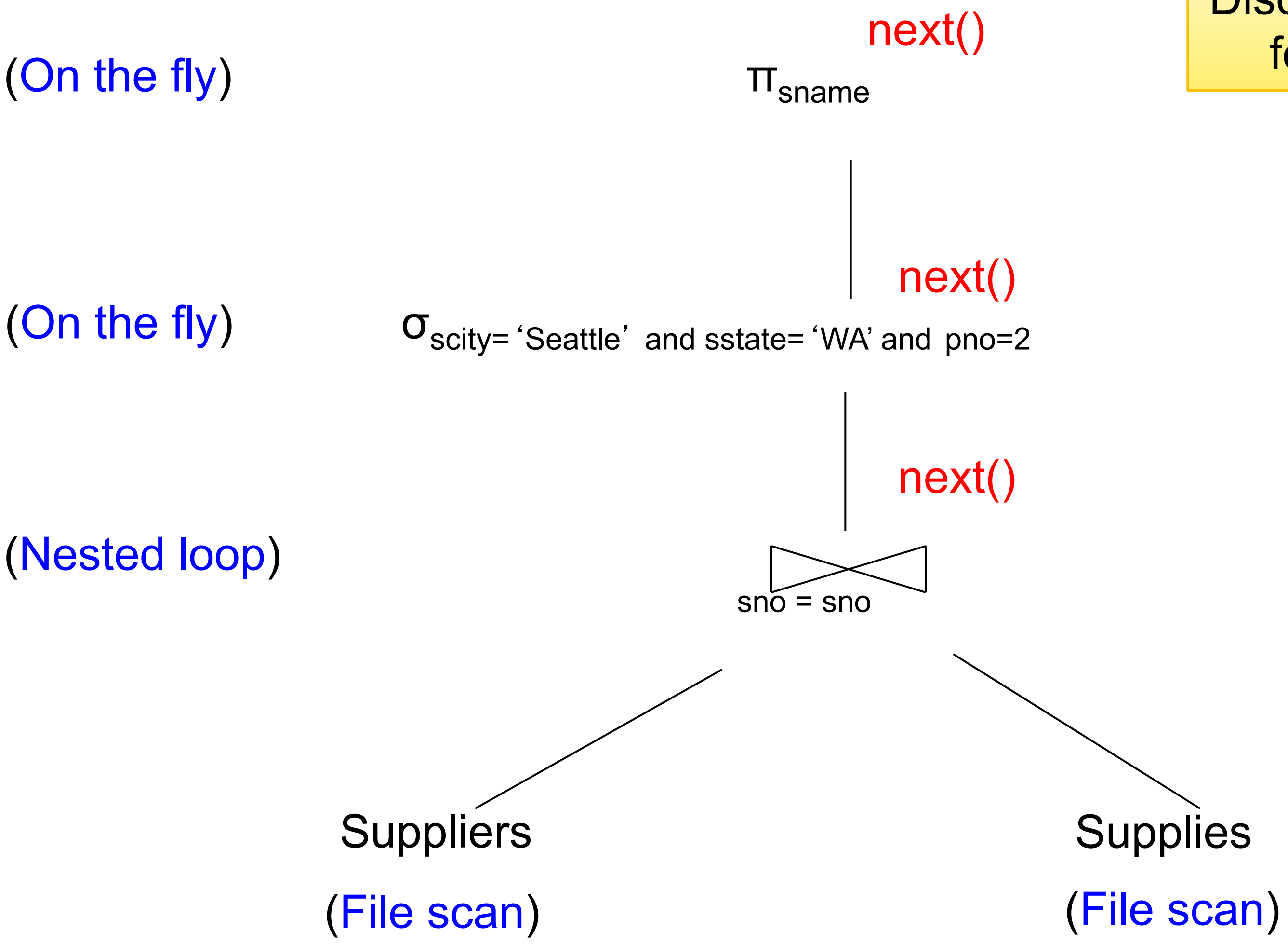
Supplies

(File scan)

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

PIPELINING

Discuss: open/next/close for nested loop join



next()

(On the fly) π_{sname}

next()

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

next()

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

Suppliers

(File scan)

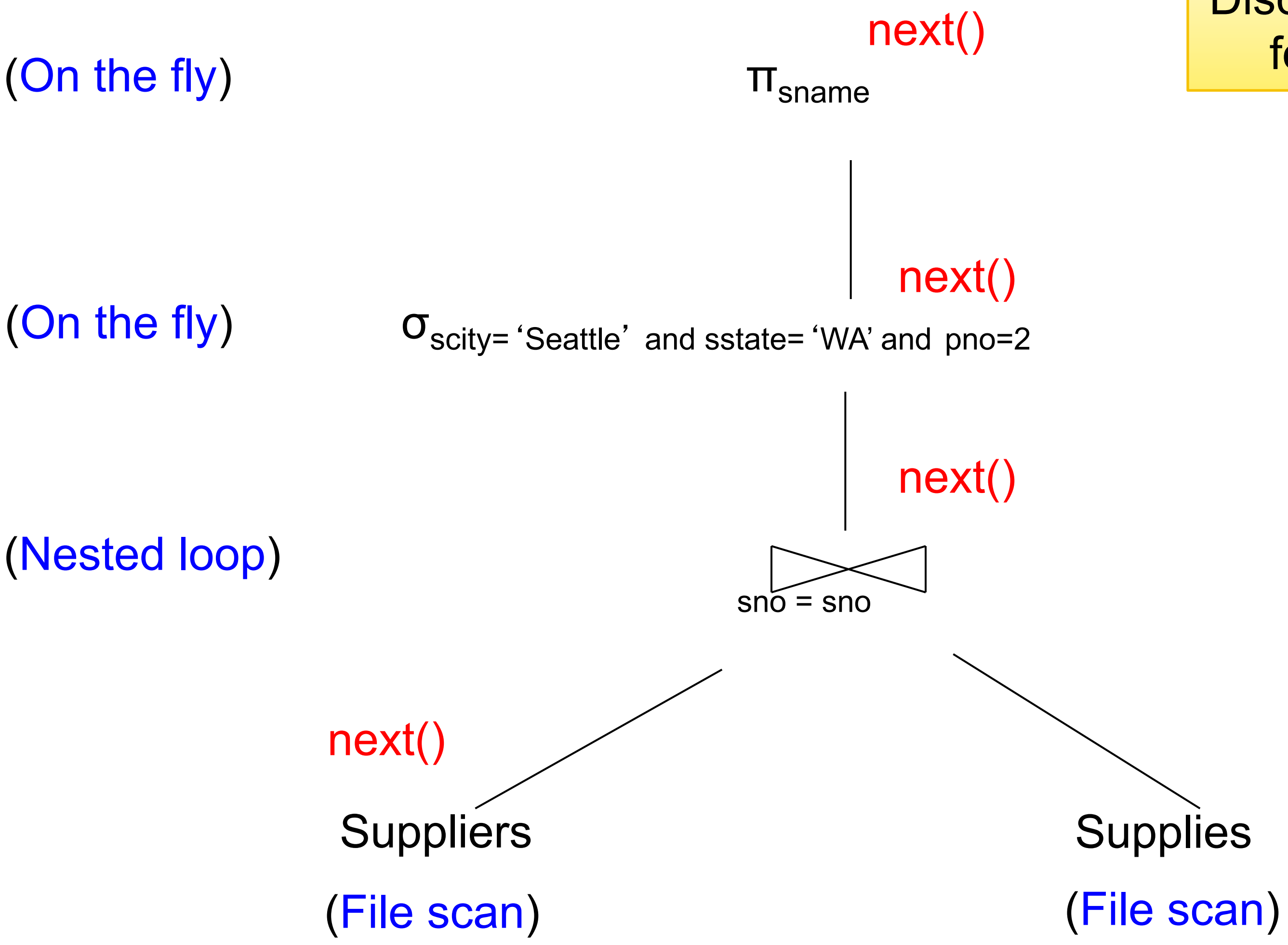
Supplies

(File scan)

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

PIPELINING

Discuss: open/next/close for nested loop join



(On the fly)

next()

π_{sname}

(On the fly)

next()

$\sigma_{scity='Seattle' \text{ and } sstate='WA' \text{ and } pno=2}$

(Nested loop)

next()

sno = sno

next()

Suppliers

(File scan)

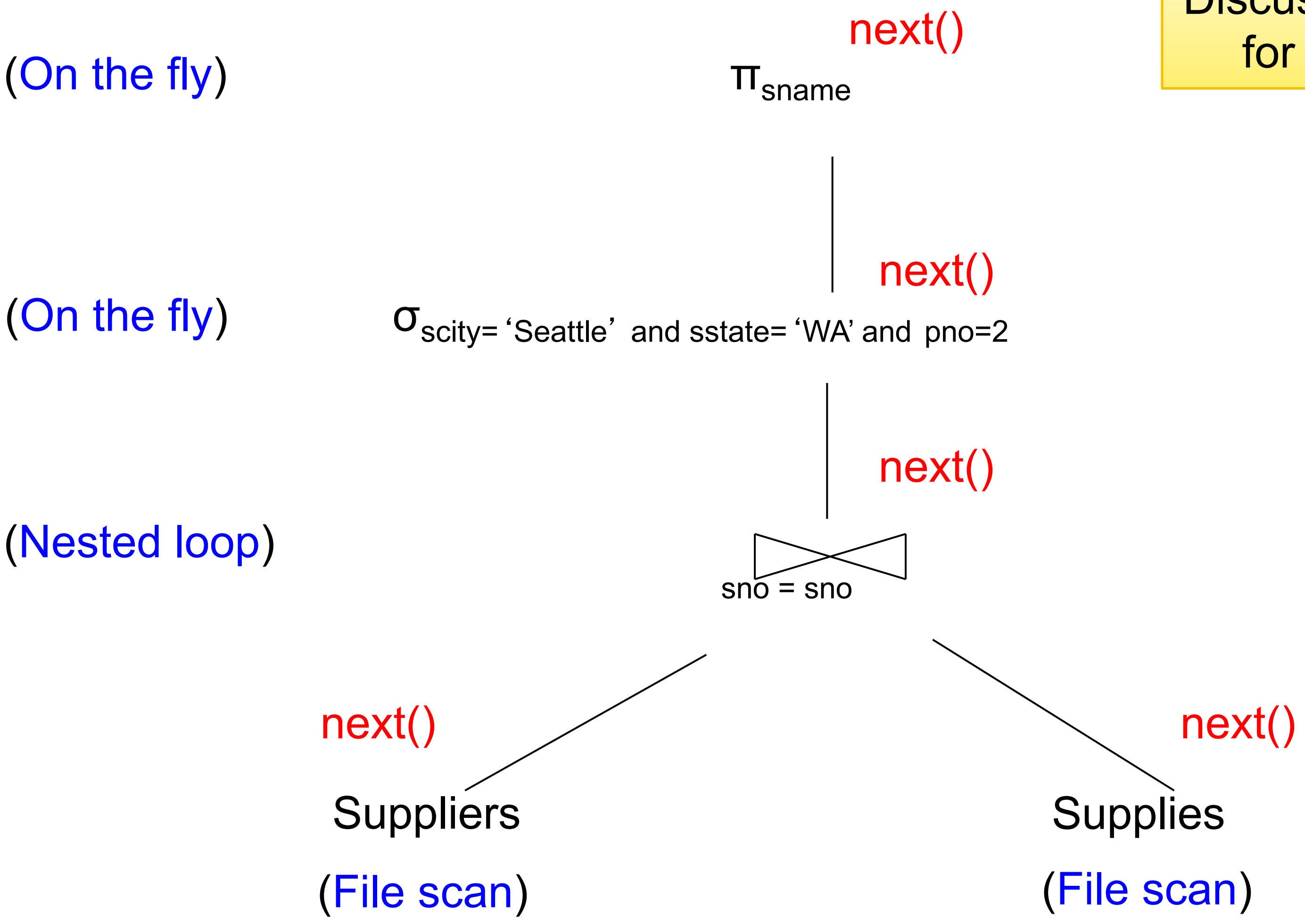
Supplies

(File scan)

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

PIPELINING

Discuss: open/next/close for nested loop join



(On the fly)

next()

π_{sname}

(On the fly)

next()

$\sigma_{scity='Seattle' \text{ and } sstate='WA' \text{ and } pno=2}$

next()

(Nested loop)

$\bowtie_{sno = sno}$

next()

next()

Suppliers

(File scan)

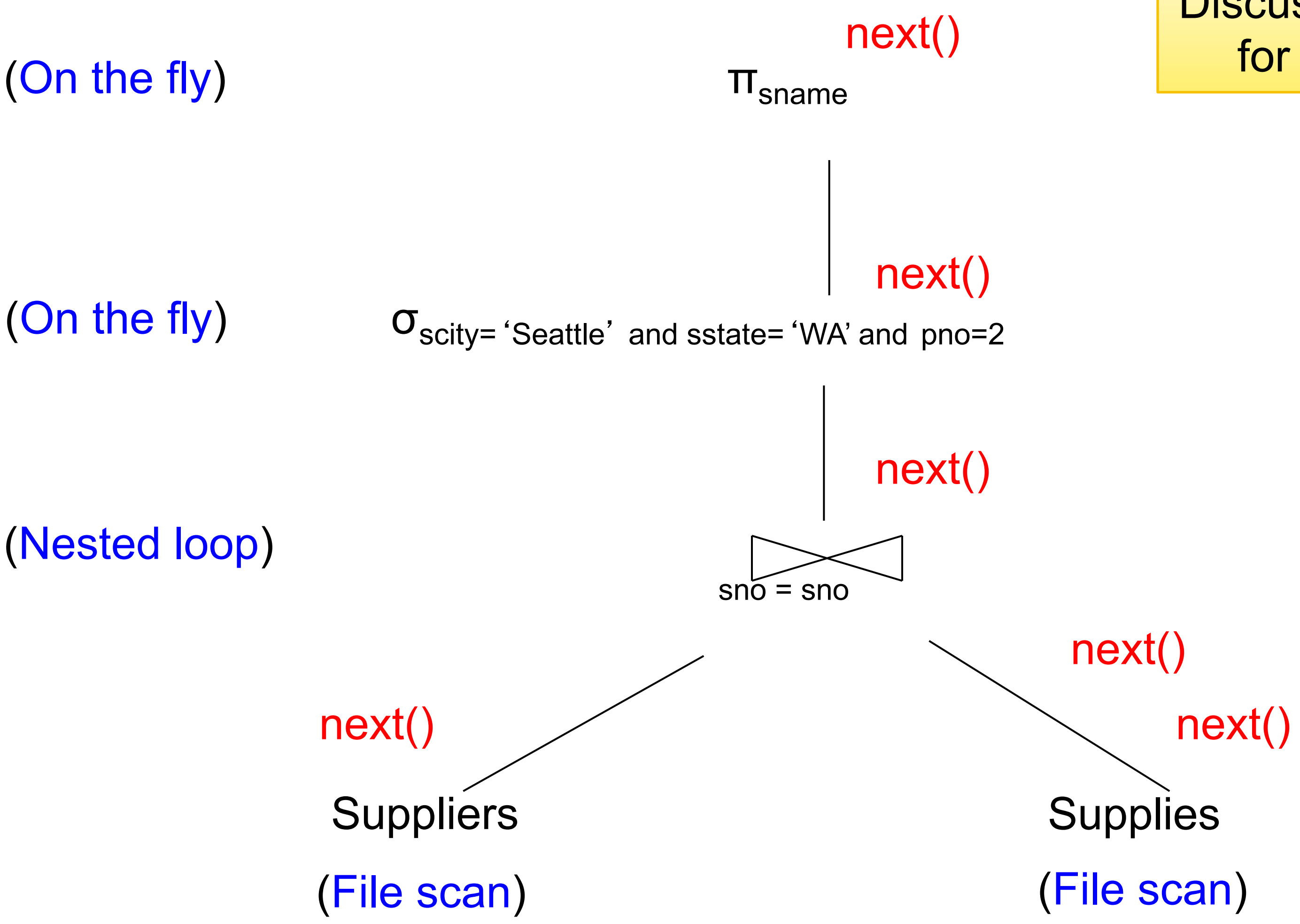
Supplies

(File scan)

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

PIPELINING

Discuss: open/next/close for nested loop join



next()

(On the fly) π_{sname}

next()

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

next()

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

next()

next()

next()

Suppliers

(File scan)

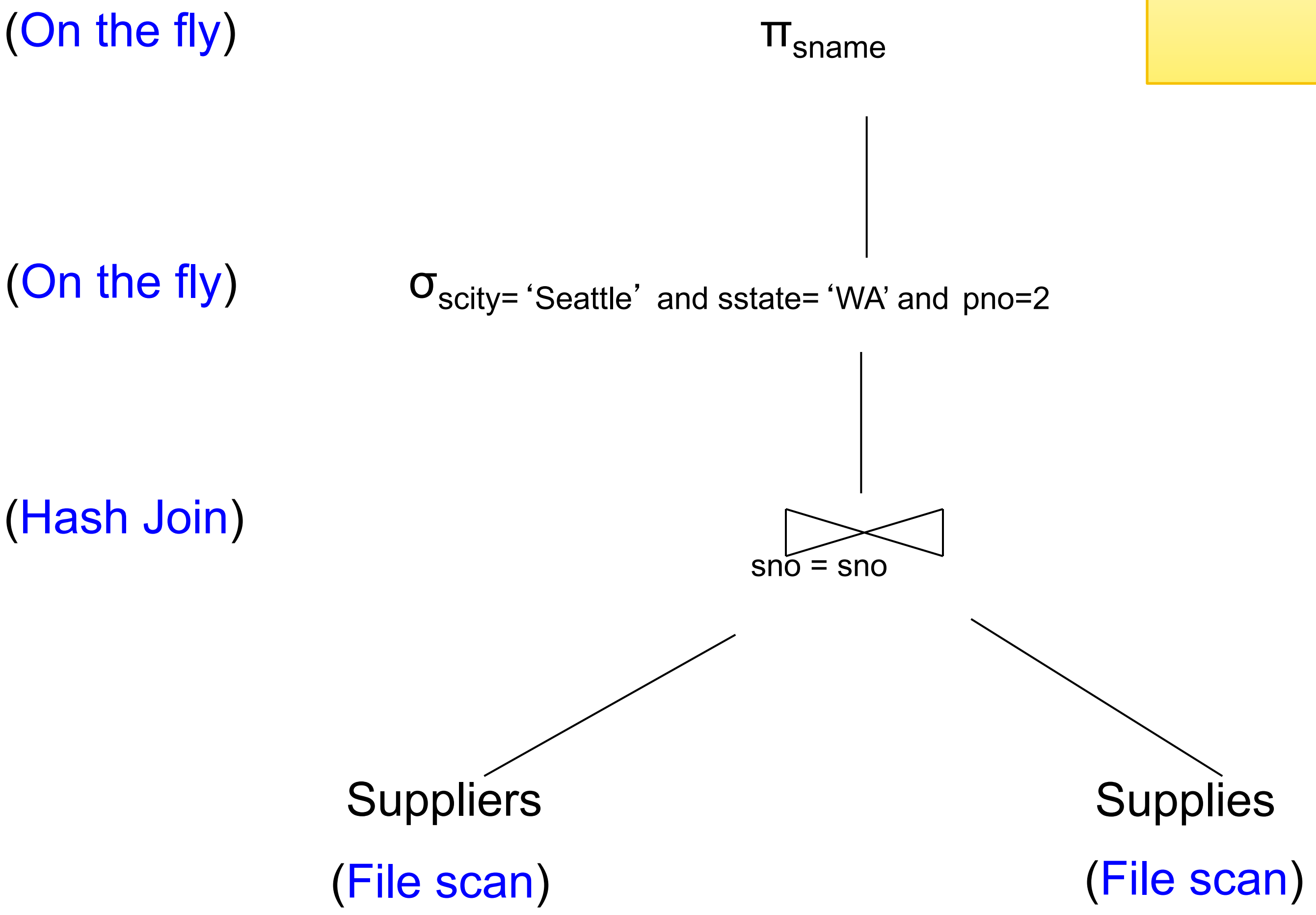
Supplies

(File scan)

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

PIPELINING

Discuss hash-join
in class



(On the fly)

π_{sname}

(On the fly)

$\sigma_{scity='Seattle' \text{ and } sstate='WA' \text{ and } pno=2}$

(Hash Join)

$\bowtie_{sno = sno}$

Suppliers

(File scan)

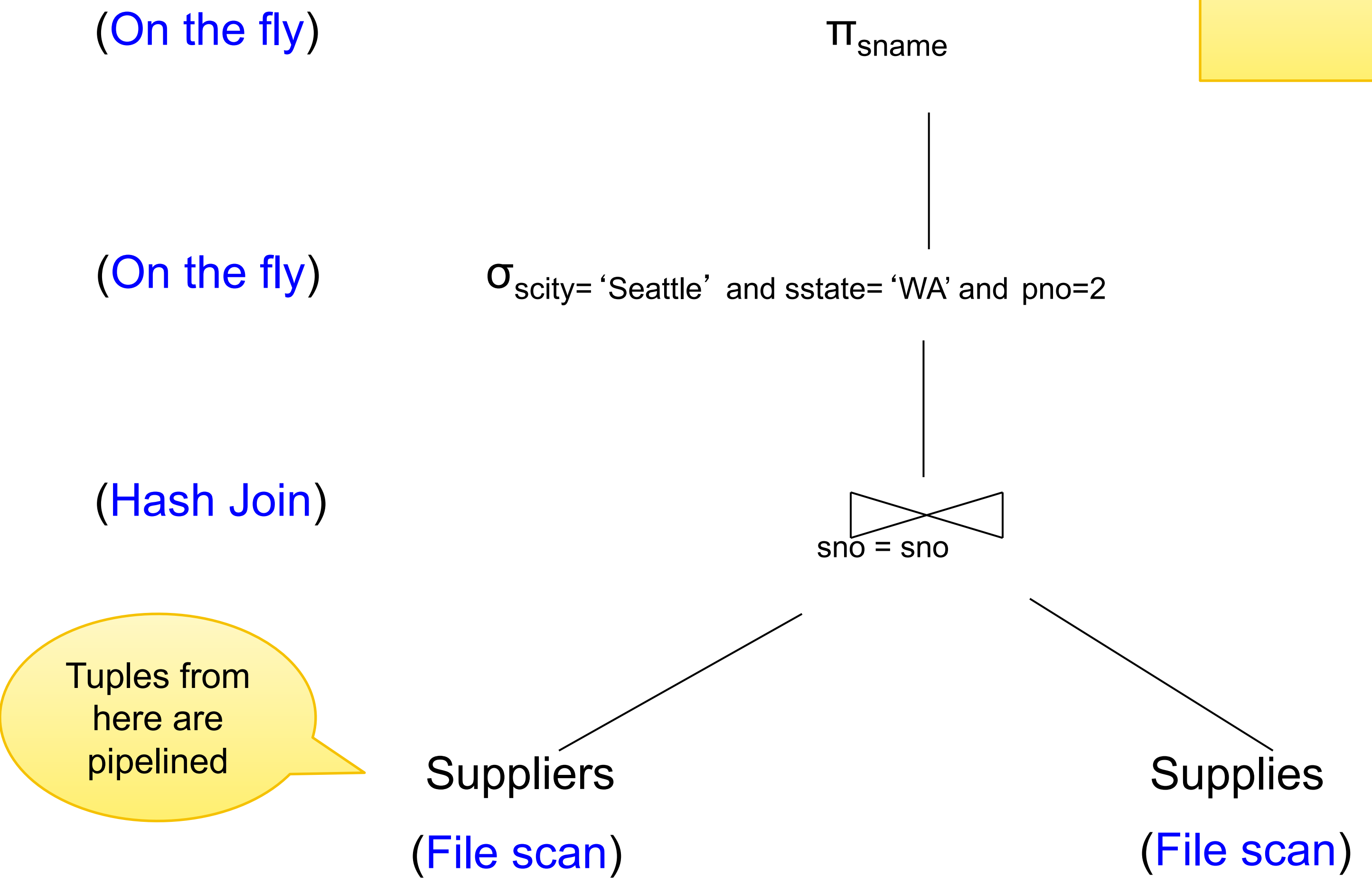
Supplies

(File scan)

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) $\bowtie_{sno = sno}$

Tuples from here are pipelined

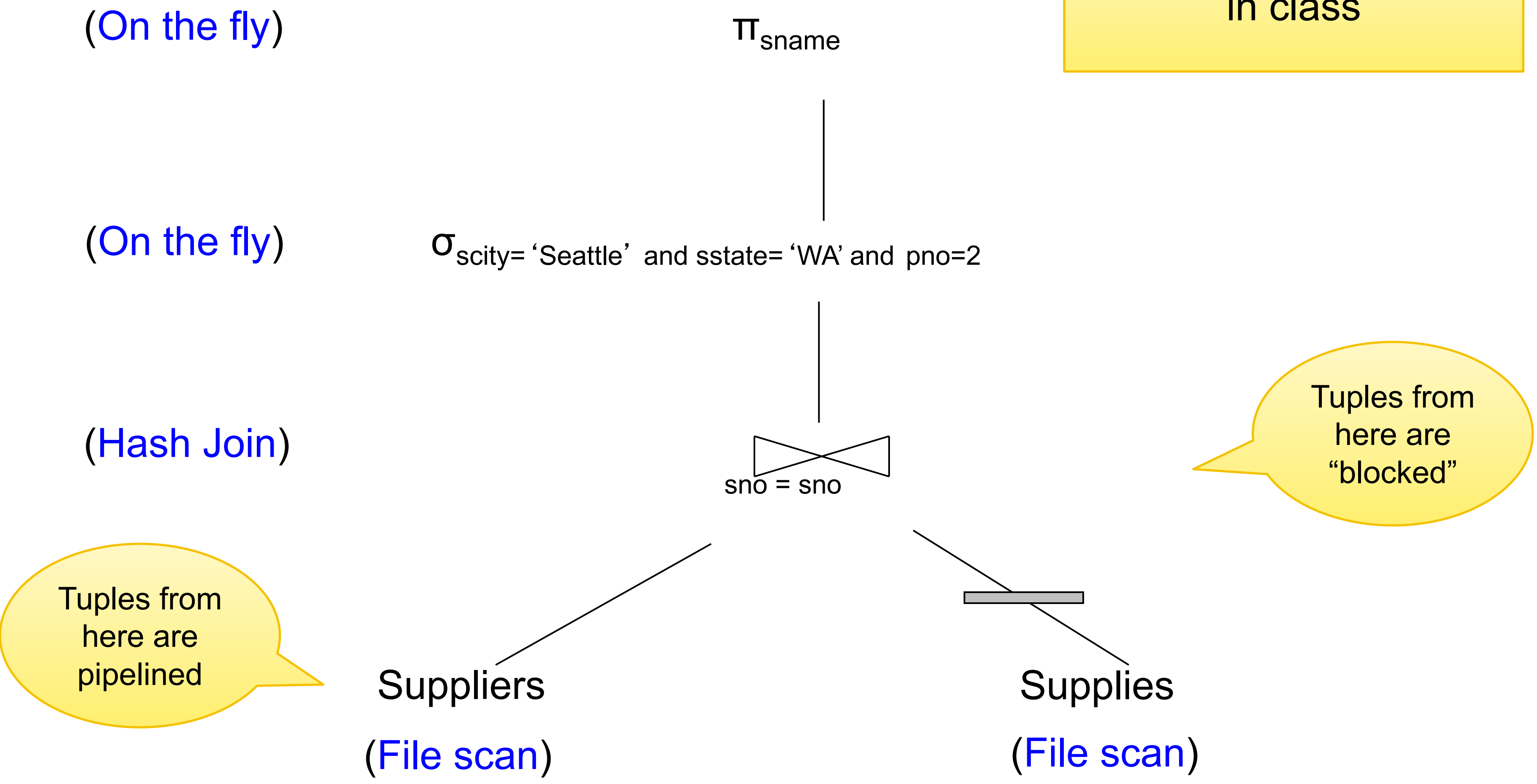
Suppliers (File scan)

Supplies (File scan)

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

PIPELINING

Discuss hash-join
in class



(On the fly) π_{sname}

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

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

Suppliers (File scan)

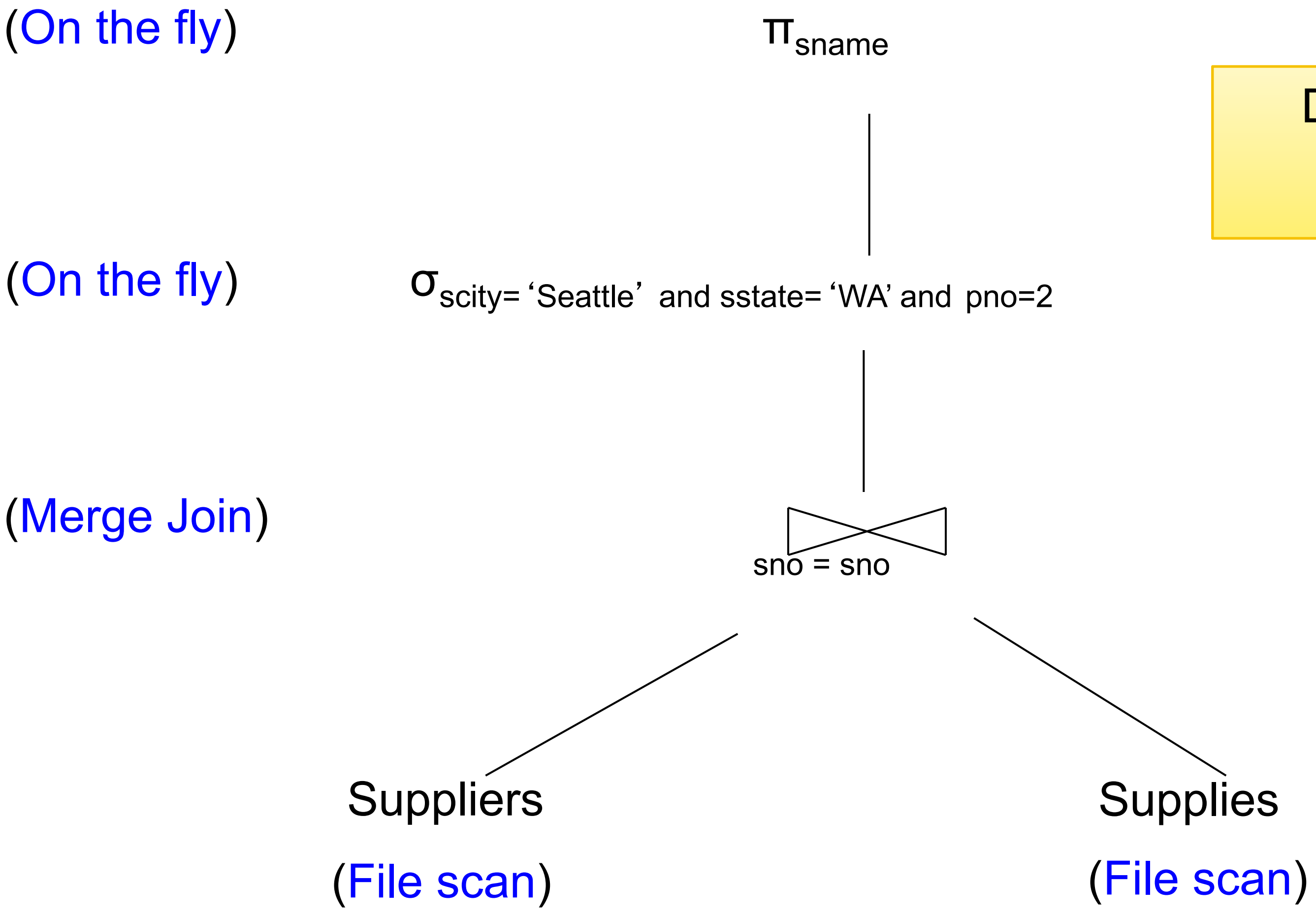
Supplies (File scan)

Tuples from here are pipelined

Tuples from here are “blocked”

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

BLOCKED EXECUTION



(On the fly) π_{sname}

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

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

Suppliers (File scan)

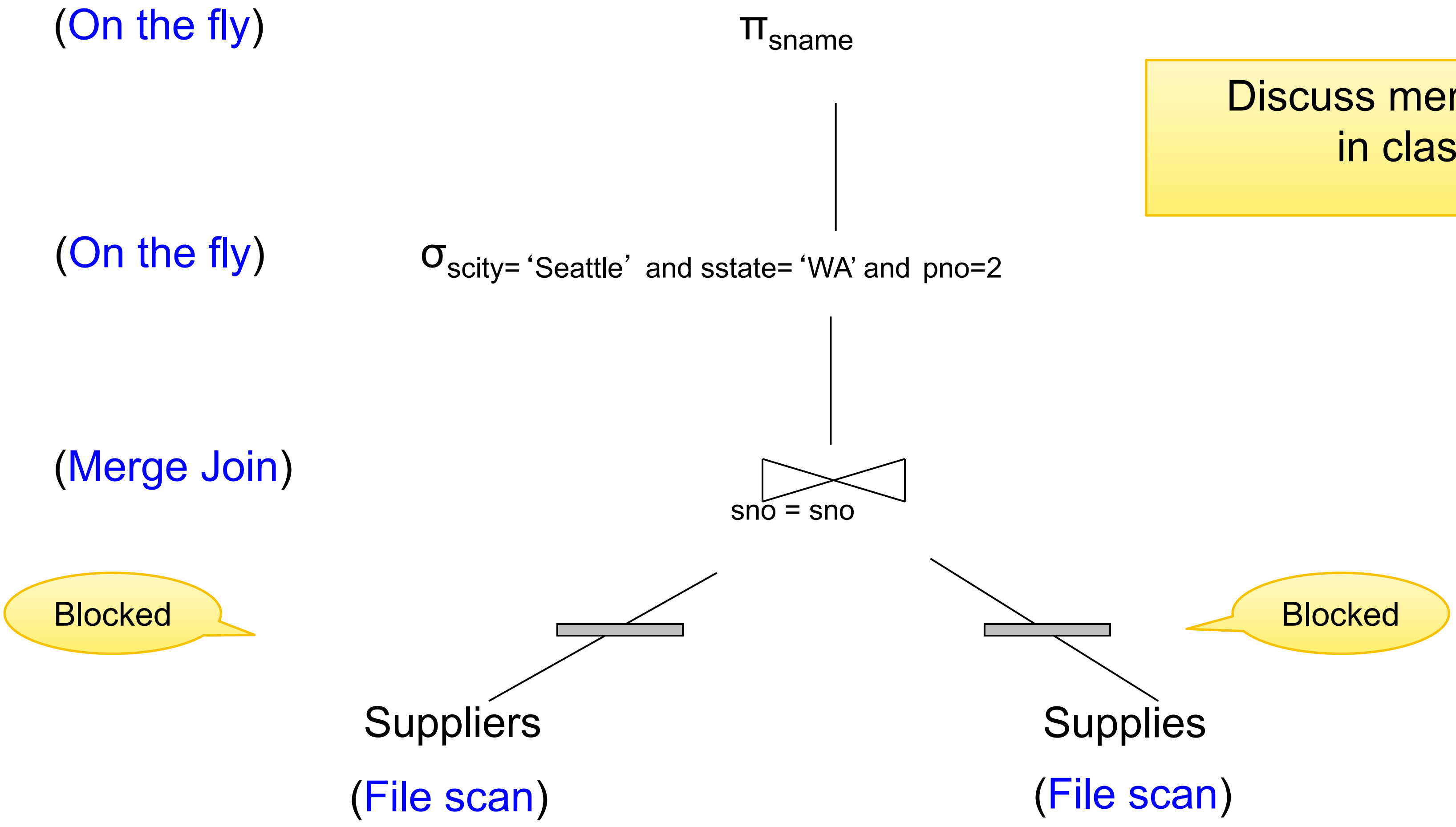
Supplies (File scan)

Discuss merge-join in class

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

BLOCKED EXECUTION

Discuss merge-join in class


(On the fly)

π_{sname}

(On the fly)

$\sigma_{scity=\text{'Seattle'} \text{ and } sstate=\text{'WA'} \text{ and } pno=2}$

(Merge Join)

$\bowtie_{sno = sno}$

Blocked

Blocked

Suppliers

(File scan)

Supplies

(File scan)

PIPELINED EXECUTION

Tuples generated by an operator are immediately sent to the parent

Benefits:

- No operator synchronization issues
- No need to buffer tuples between operators
- Saves cost of writing intermediate data to disk
- Saves cost of reading intermediate data from disk

This approach is used whenever possible

QUERY EXECUTION BOTTOM LINE

SQL query transformed into physical plan

- **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

Pipelined execution of physical plan

RECALL: PHYSICAL DATA INDEPENDENCE

Applications are insulated from changes in physical storage details

SQL and relational algebra facilitate physical data independence

- Both languages input and output relations
- Can choose different implementations for operators

QUERY PERFORMANCE

My database application is too slow… why?

One of the queries is very slow… why?

To understand performance, we need to understand:

- How is data organized on disk
- How to estimate query costs

- In this course we will focus on **disk-based** DBMSs

DATA STORAGE

Student

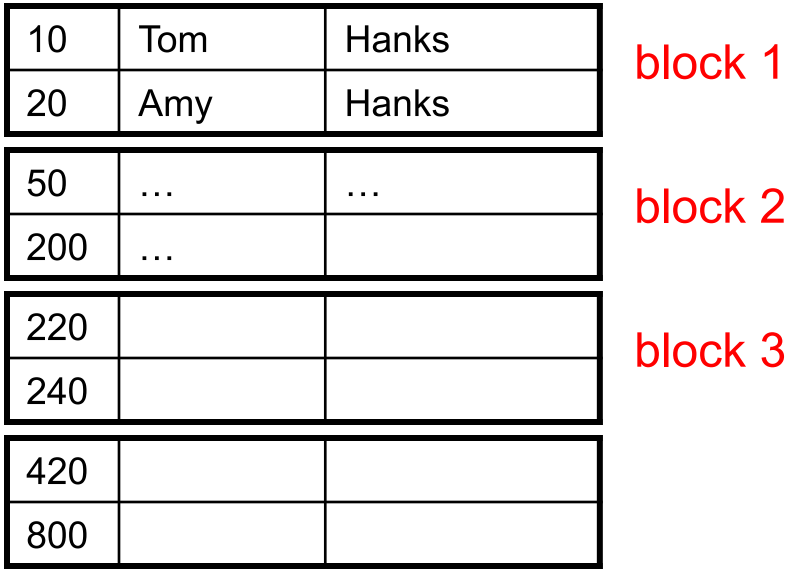
ID	fName	lName
10	Tom	Hanks
20	Amy	Hanks
…		

DBMSs store data in files

Most common organization is row-wise storage

On disk, a file is split into blocks

Each block contains a set of tuples



10	Tom	Hanks	block 1
20	Amy	Hanks	
50	…	…	block 2
200	…		
220			block 3
240			
420			
800			

In the example, we have 4 blocks with 2 tuples each

Student

ID	fName	lName
10	Tom	Hanks
20	Amy	Hanks
…		

DATA FILE TYPES

The data file can be one of:

Heap file

- Unsorted

Sequential file

- Sorted according to some attribute(s) called *key*

Student

ID	fName	lName
10	Tom	Hanks
20	Amy	Hanks
…		

DATA FILE TYPES

The data file can be one of:

Heap file

- Unsorted

Sequential file

- Sorted according to some attribute(s) called *key*

Note: *key* here means something different from primary key:
it just means that we order the file according to that attribute.
In our example we ordered by **ID**. Might as well order by **fName,**
if that seems a better idea for the applications running on
our database.

INDEX

An additional file, that allows fast access to records in the data file given a search key

INDEX

An additional file, that allows fast access to records in the data file given a search key

The index contains (key, value) pairs:

- The key = an attribute value (e.g., student ID or name)
- The value = a pointer to the record

INDEX

An additional file, that allows fast access to records in the data file given a search key

The index contains (key, value) pairs:

- The key = an attribute value (e.g., student ID or name)
- The value = a pointer to the record

Could have many indexes for one table

Key = means here search key

KEYS IN INDEXING

Different keys:

Primary key – uniquely identifies a tuple

Key of the sequential file – how the data file is sorted, if at all

Index key – how the index is organized

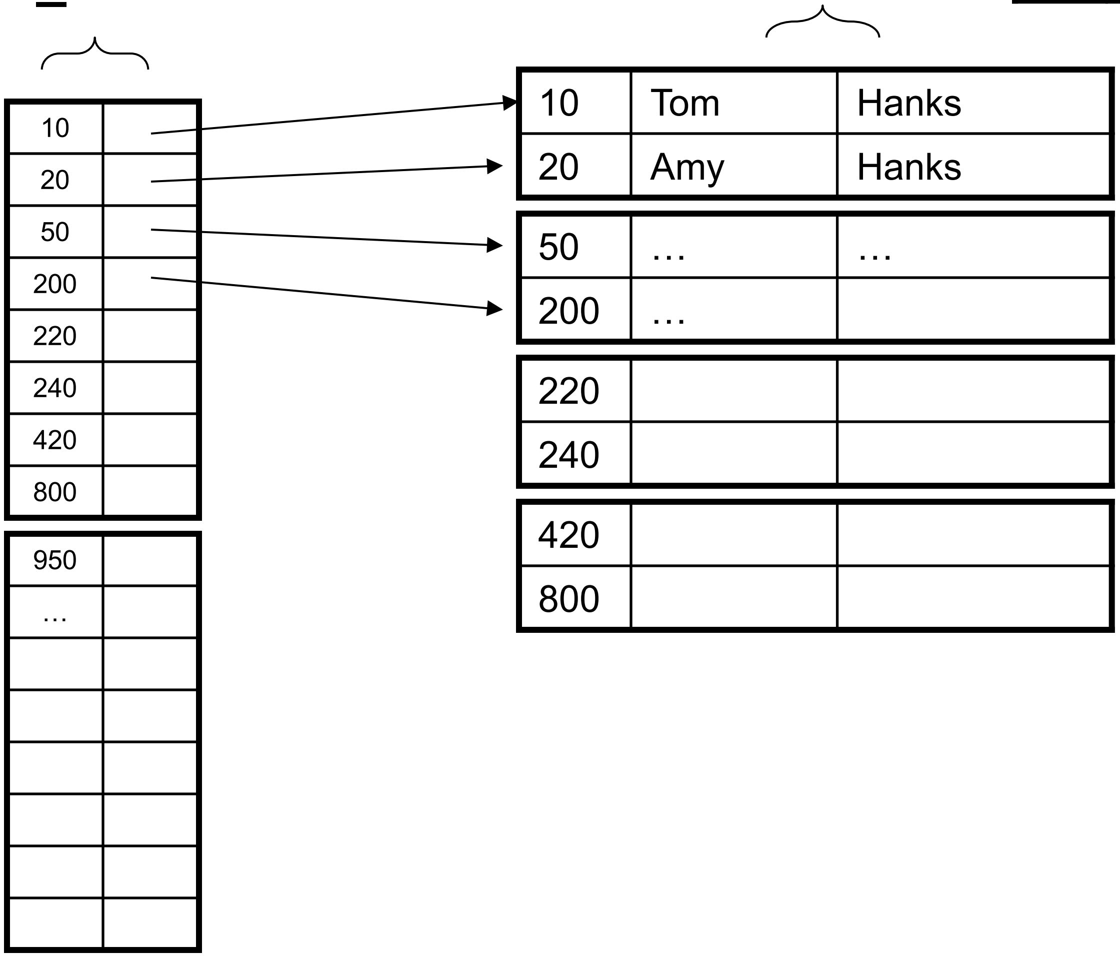
EXAMPLE 1: INDEX ON ID

Student

ID	fName	lName
10	Tom	Hanks
20	Amy	Hanks
…		

Index **Student_ID** on **Student.ID**

Data File **Student**



10	
20	
50	
200	
220	
240	
420	
800	
950	
…	

10	Tom	Hanks
20	Amy	Hanks
50	…	…
200	…	
220		
240		
420		
800		

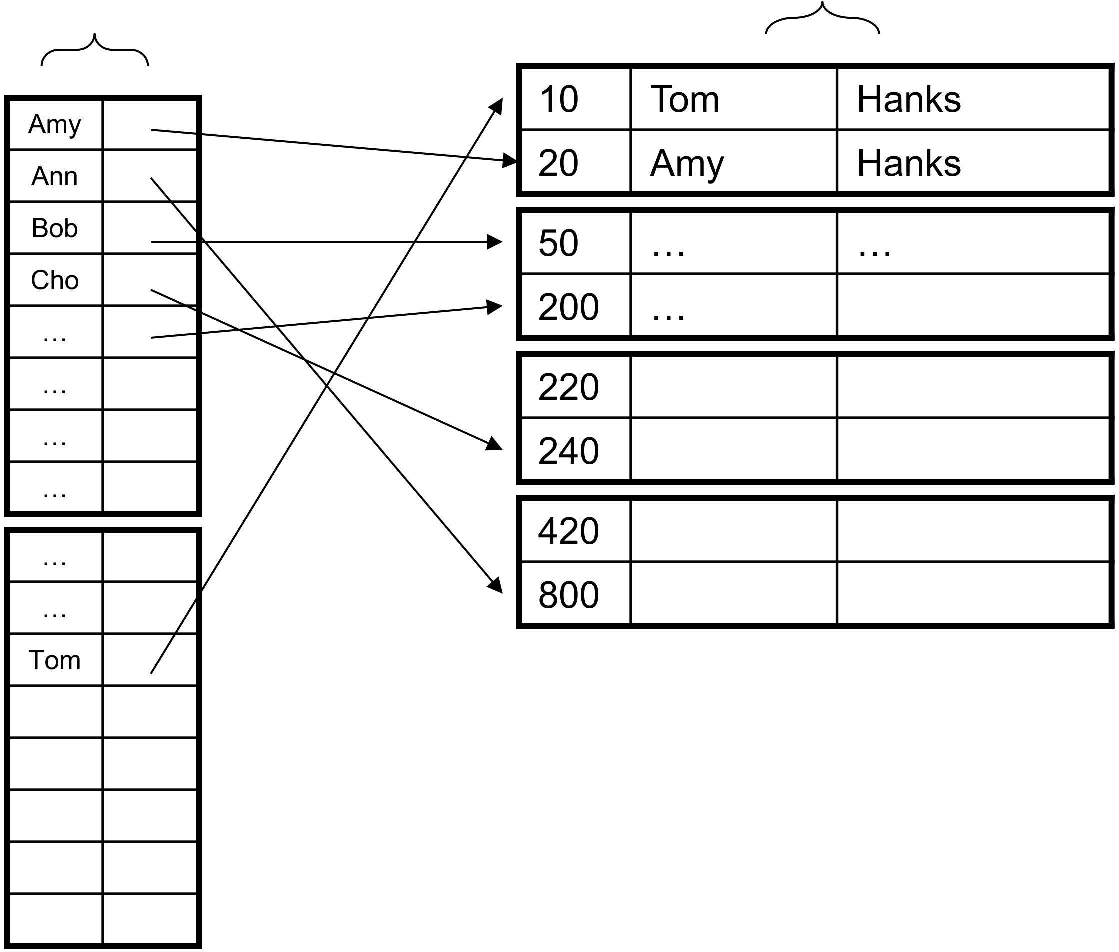
EXAMPLE 2: INDEX ON FNAME

Student

ID	fName	lName
10	Tom	Hanks
20	Amy	Hanks
…		

Index **Student_fName**
on **Student.fName**

Data File **Student**



Amy	
Ann	
Bob	
Cho	
…	
…	
…	
…	

…	
…	
Tom	

10	Tom	Hanks
20	Amy	Hanks
50	…	…
200	…	
220		
240		
420		
800		

INDEX ORGANIZATION

We need a way to represent indexes after loading into memory so that they can be used

Several ways to do this:

Hash table

B+ trees – most popular

- They are search trees, but they are not binary instead have higher fanout
- Will discuss them briefly next

Specialized indexes: bit maps, R-trees, inverted index

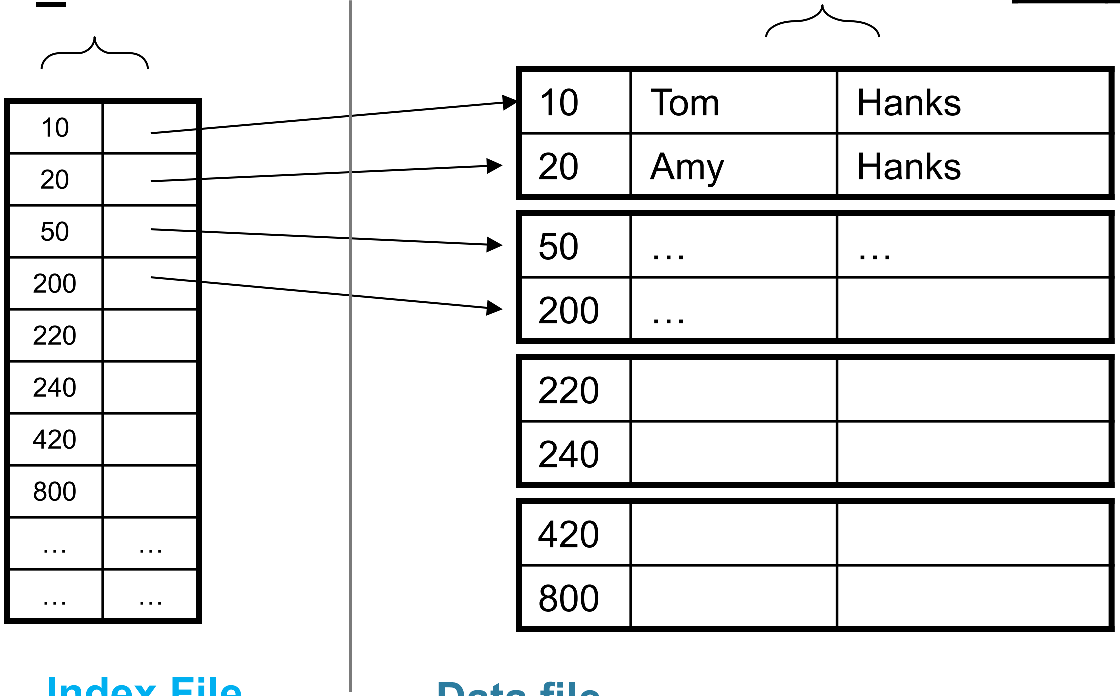
HASH TABLE EXAMPLE

Student

ID	fName	lName
10	Tom	Hanks
20	Amy	Hanks
…		

Index **Student_ID** on **Student.ID**

Data File **Student**



10	
20	
50	
200	
220	
240	
420	
800	
…	…
…	…

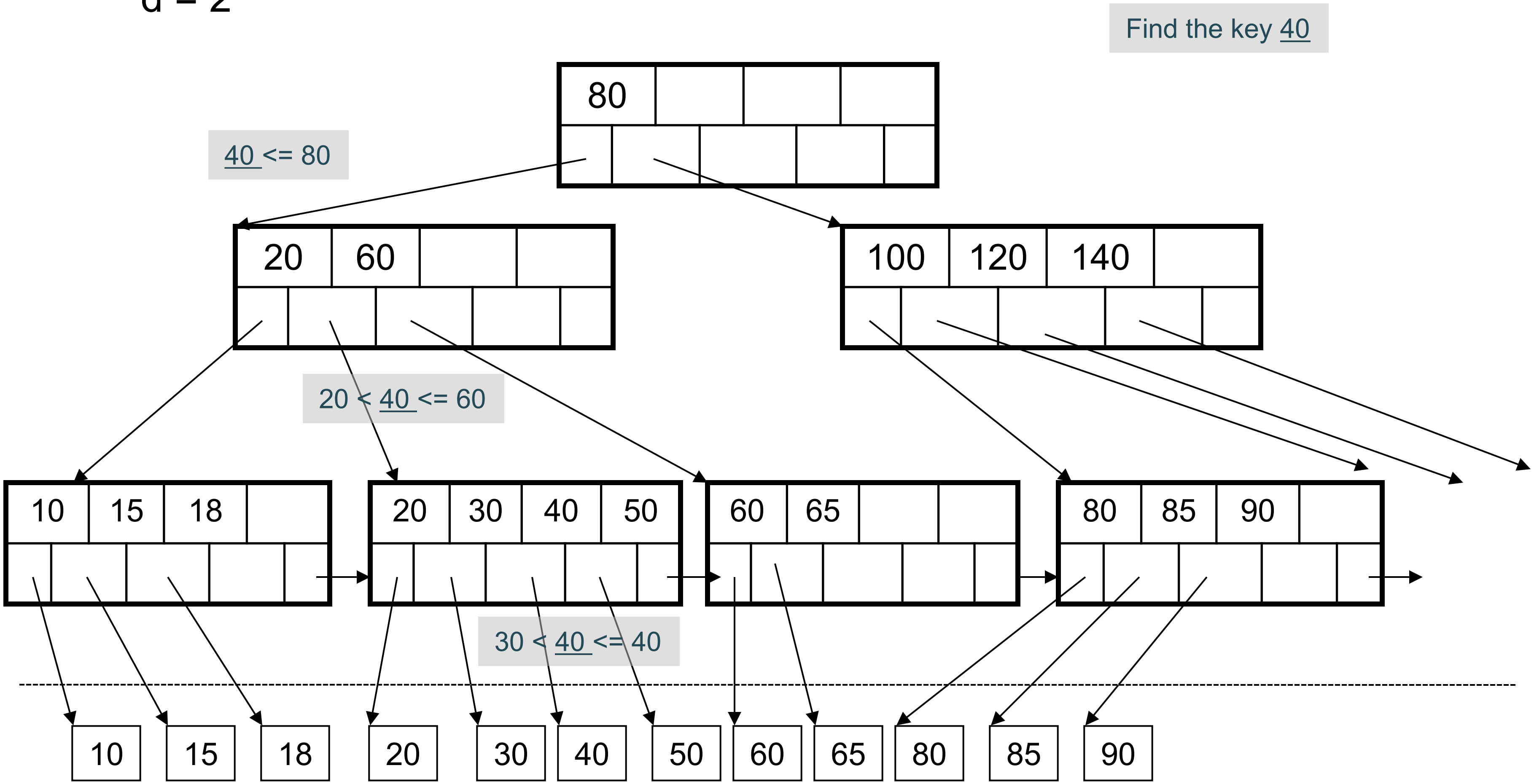
10	Tom	Hanks
20	Amy	Hanks
50	…	…
200	…	
220		
240		
420		
800		

Index File (preferably in memory)

Data file (on disk)

B+ TREE INDEX BY EXAMPLE

d = 2

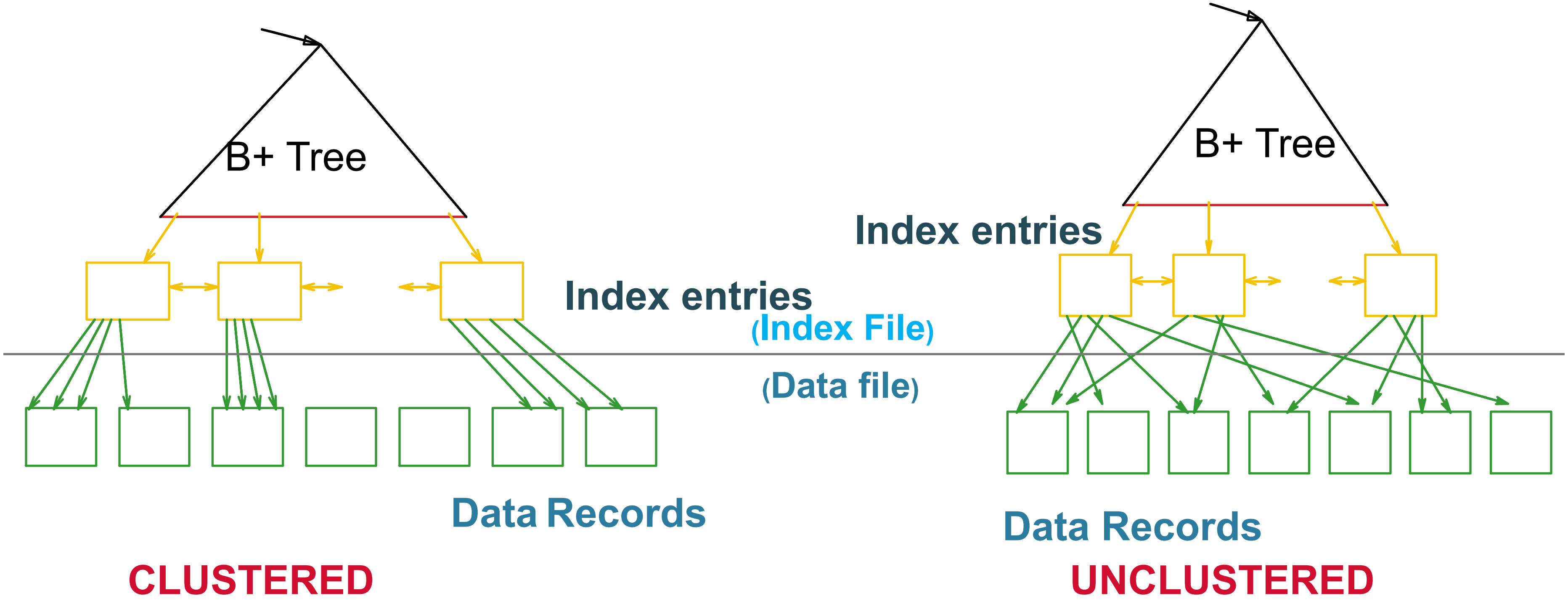

Find the key 40

40 <= 80

20 < 40 <= 60

30 < 40 <= 40

CLUSTERED VS UNCLUSTERED



B+ Tree

Index entries

(Index File)

(Data file)

Data Records

CLUSTERED

B+ Tree

Index entries

Data Records

UNCLUSTERED

Every table can have **only one** clustered and **many** unclustered indexes
Why?

INDEX CLASSIFICATION

Clustered/unclustered

- Clustered = records close in index are close in data
    - Option 1: Data inside data file is sorted on disk
    - Option 2: Store data directly inside the index (no separate files)
- Unclustered = records close in index may be far in data

INDEX CLASSIFICATION

Clustered/unclustered

- Clustered = records close in index are close in data
  - Option 1: Data inside data file is sorted on disk
  - Option 2: Store data directly inside the index (no separate files)
- Unclustered = records close in index may be far in data

Primary/secondary

- Meaning 1:
  - Primary = is over attributes that include the primary key
  - Secondary = otherwise
- Meaning 2: means the same as clustered/unclustered

INDEX CLASSIFICATION

Clustered/unclustered

- Clustered = records close in index are close in data
    - Option 1: Data inside data file is sorted on disk
    - Option 2: Store data directly inside the index (no separate files)
- Unclustered = records close in index may be far in data

Primary/secondary

- Meaning 1:
    - Primary = is over attributes that include the primary key
    - Secondary = otherwise
- Meaning 2: means the same as clustered/unclustered

Organization B+ tree or Hash table