Introduction to Data Management
CSE 344

Unit 4: RDBMS Internals

Logical and Physical Plans

Query Execution

Query Optimization

(4 lectures)

Introduction to Data Management
CSE 344

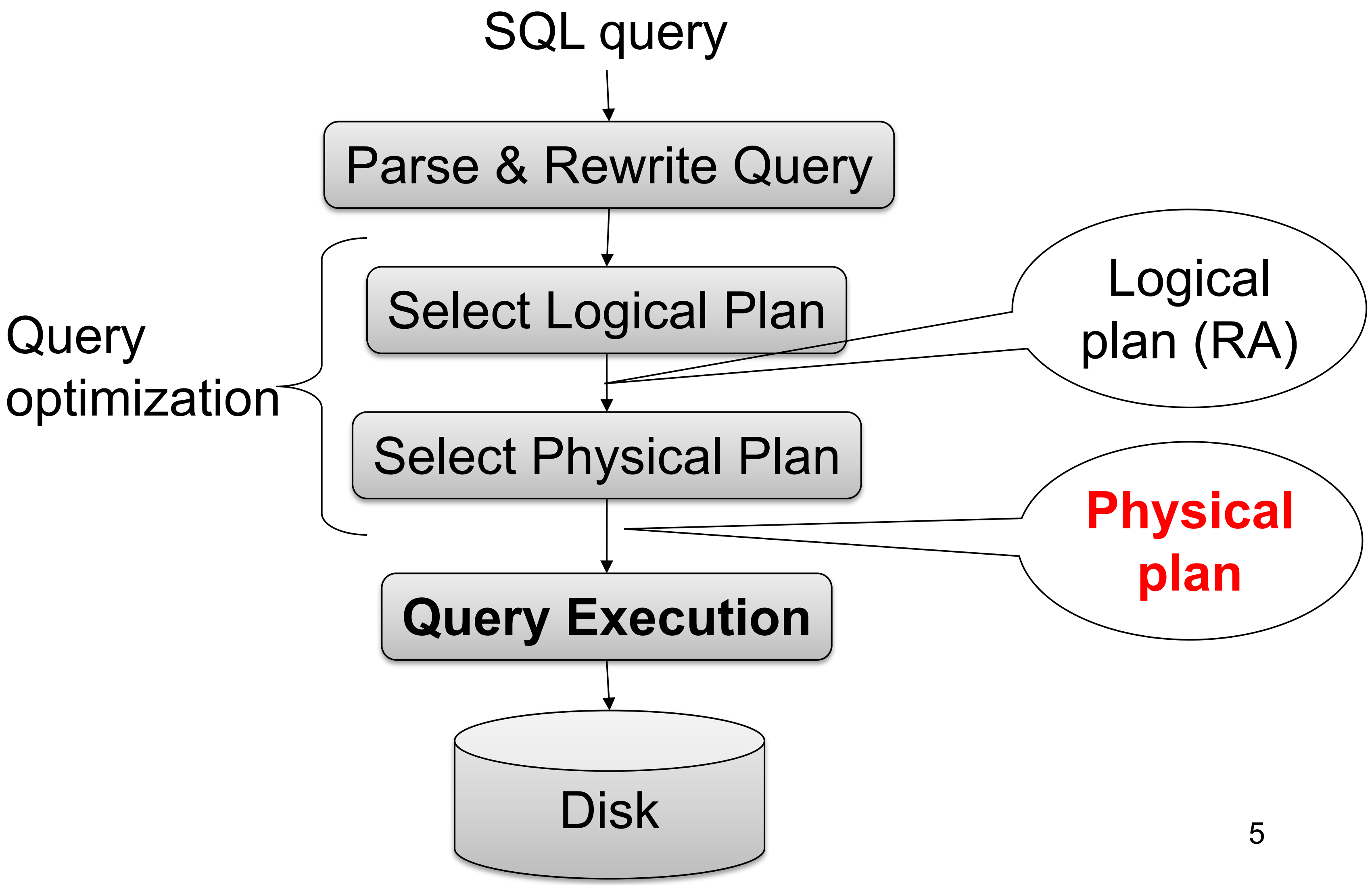
Lecture 15: Introduction to Query Evaluation

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



Relational Algebra Operators

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

RA

Extended RA

Physical Operators

- For each operators above, several possible algorithms
- Main memory or external memory algorithms
- Examples:
  - Main memory hash join
  - External memory merge join
  - External memory partitioned hash join
  - Sort-based group by
  - Etc, etc

$\bowtie$

γ

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:

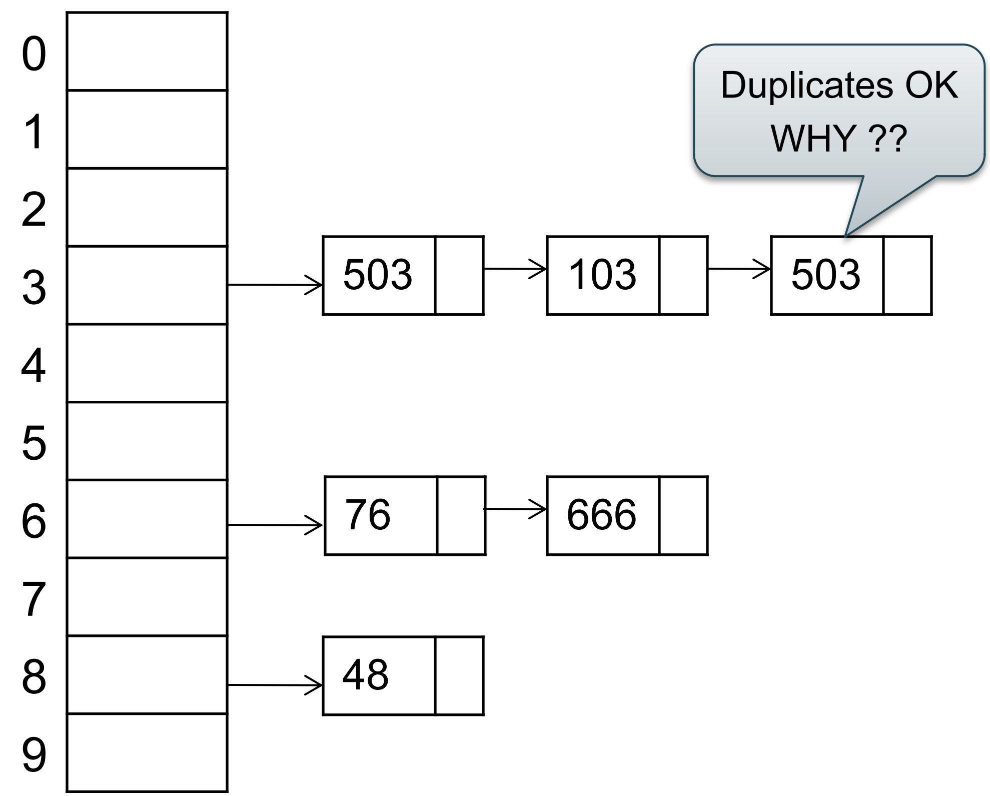
h(x) = x mod 10

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

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

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

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


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

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

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?

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)
```

2. Hash Join

Why would we change the order?

Logical operator:

Supplier $\bowtie_{sid=sid}$ Supply

Change jo

When |Supply| << |Supplier|

```
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))

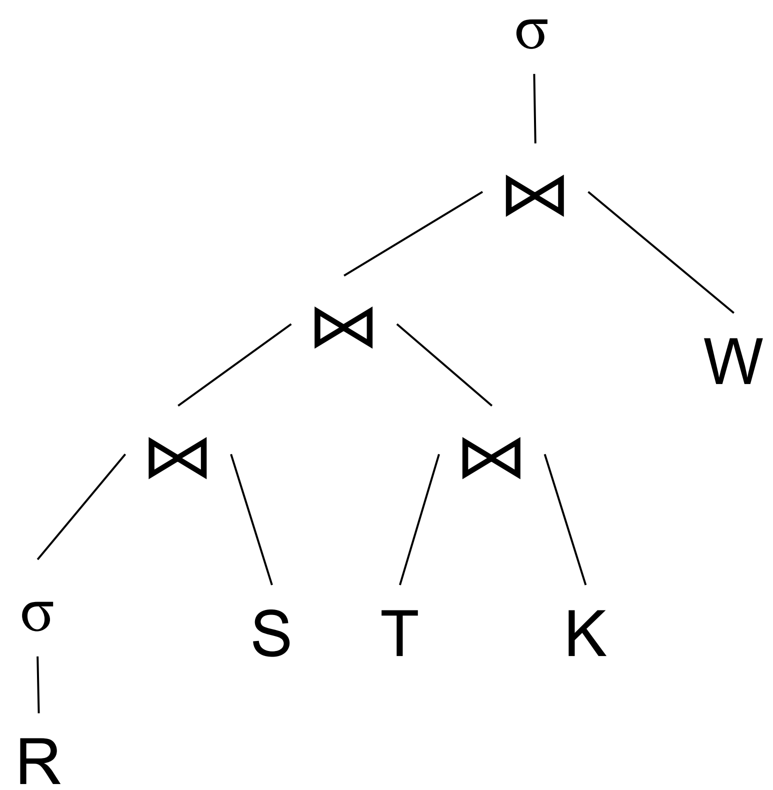
Main Memory Algorithms

- 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

Discuss in class

Discuss in class

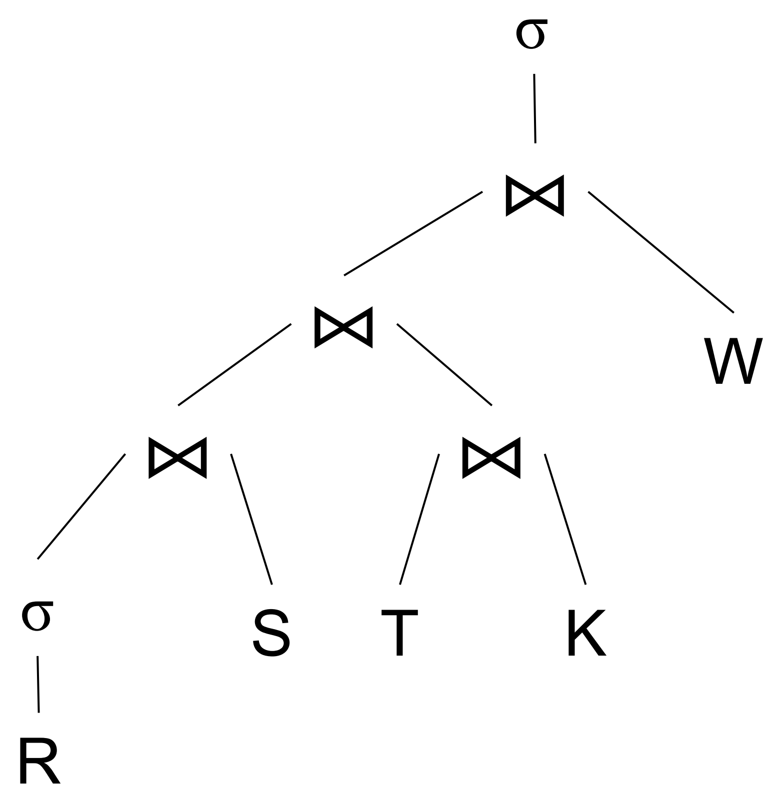
How Do We Combine Them?



How Do We Combine Them?

The *Iterator Interface*

- open()
- next()
- 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 c) {
    this.p = p; this.c = c; c.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 ();


  // 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 () {




  }
}
```

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

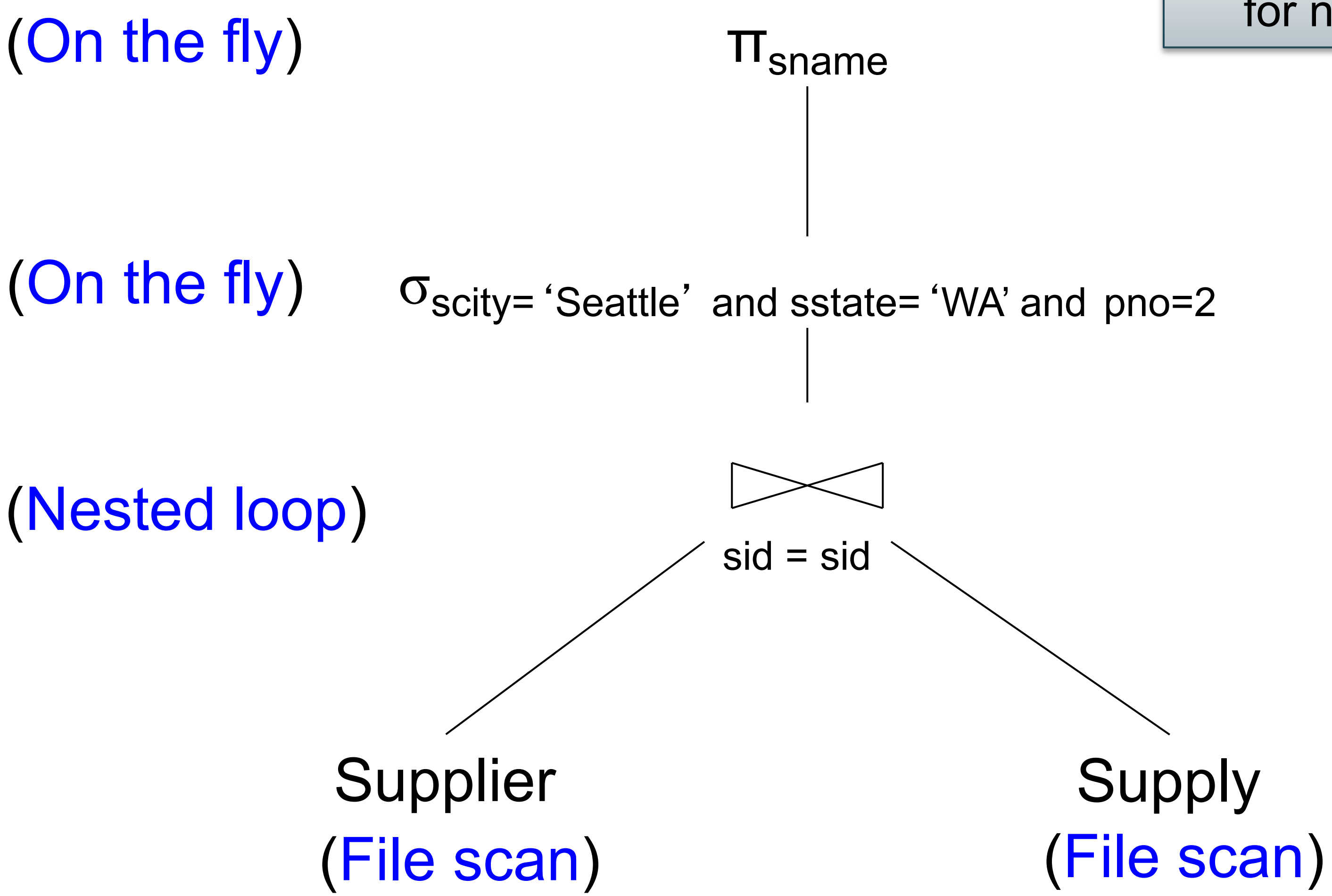
```
Operator q = parse("SELECT ...");
q = optimize(q);

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

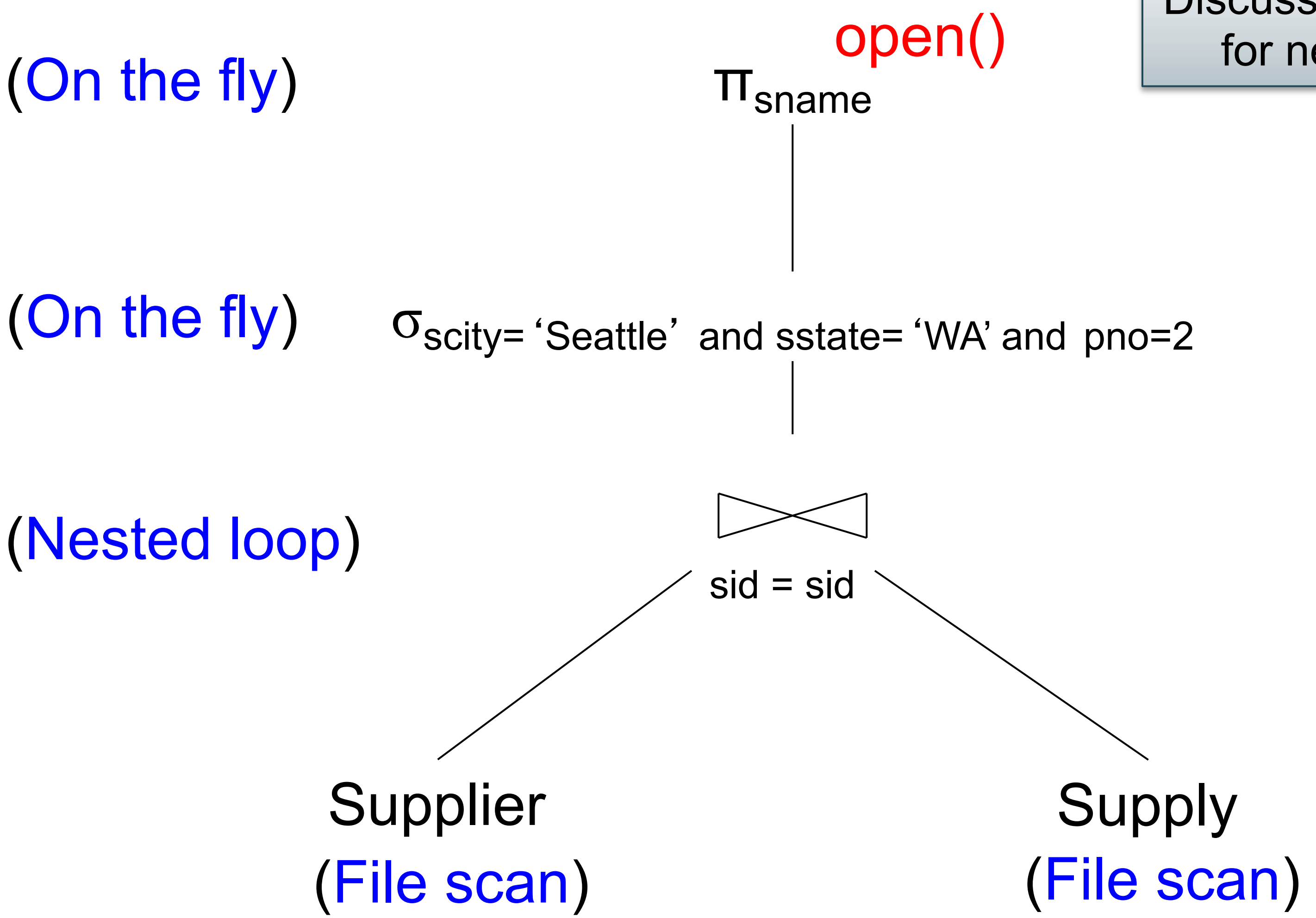
Discuss: open/next/close for nested loop join



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

Pipelining

Discuss: open/next/close for nested loop join

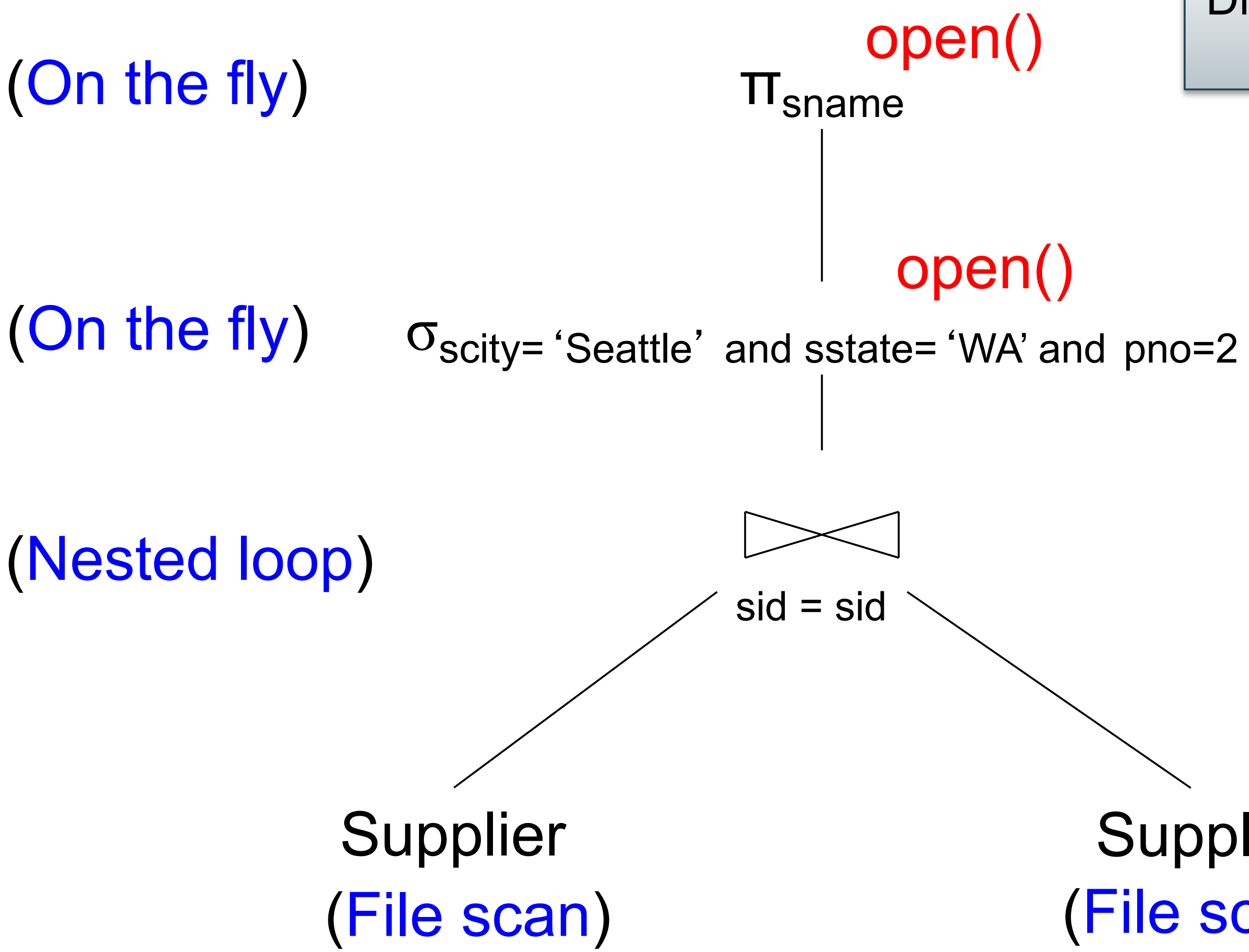


Supplier(sid, sname, scity, sstate)

Supply(sid, pno, quantity)

Pipelining

Discuss: open/next/close for nested loop join

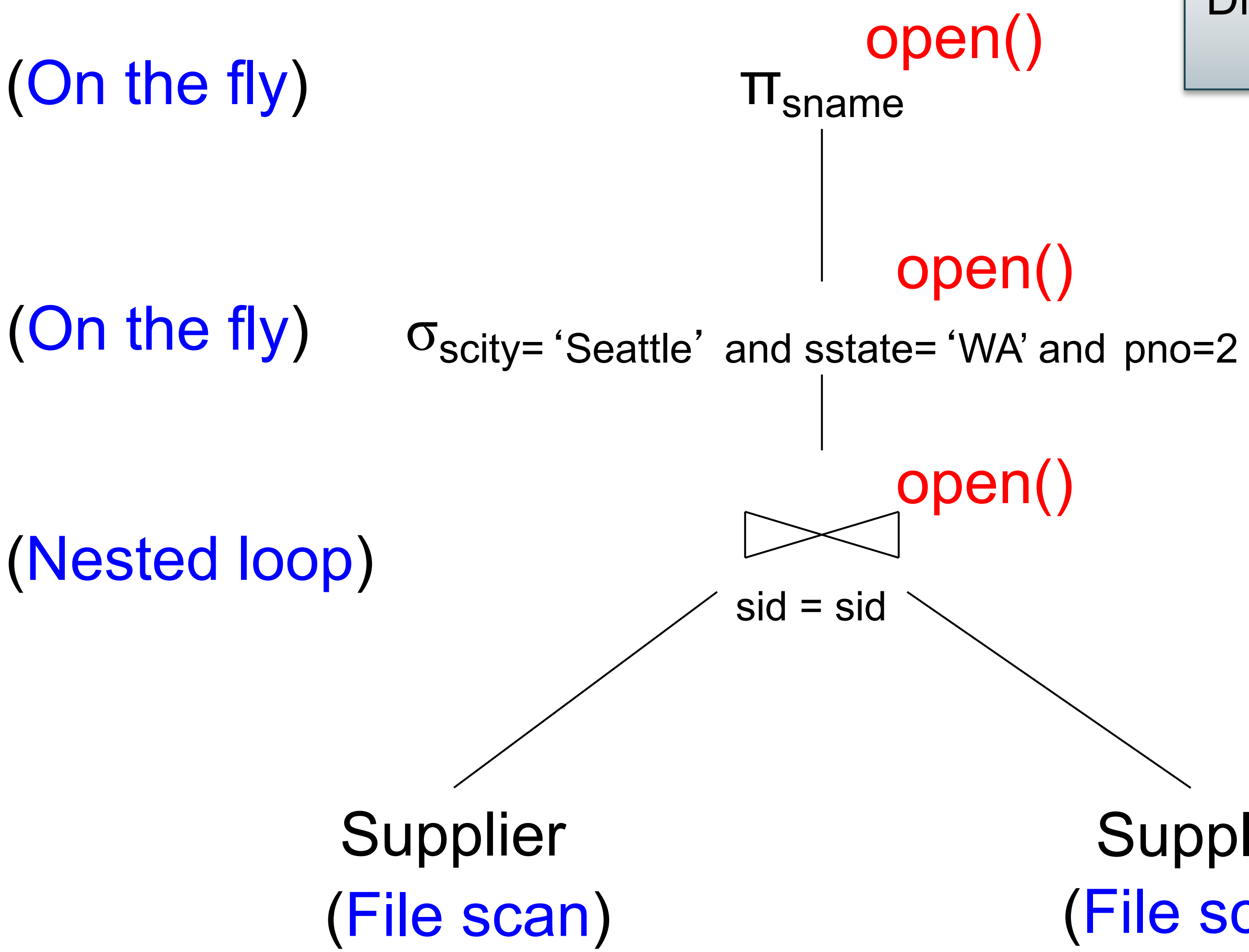


`Supplier(sid, sname, scity, sstate)`

`Supply(sid, pno, quantity)`

Pipelining

Discuss: open/next/close for nested loop join

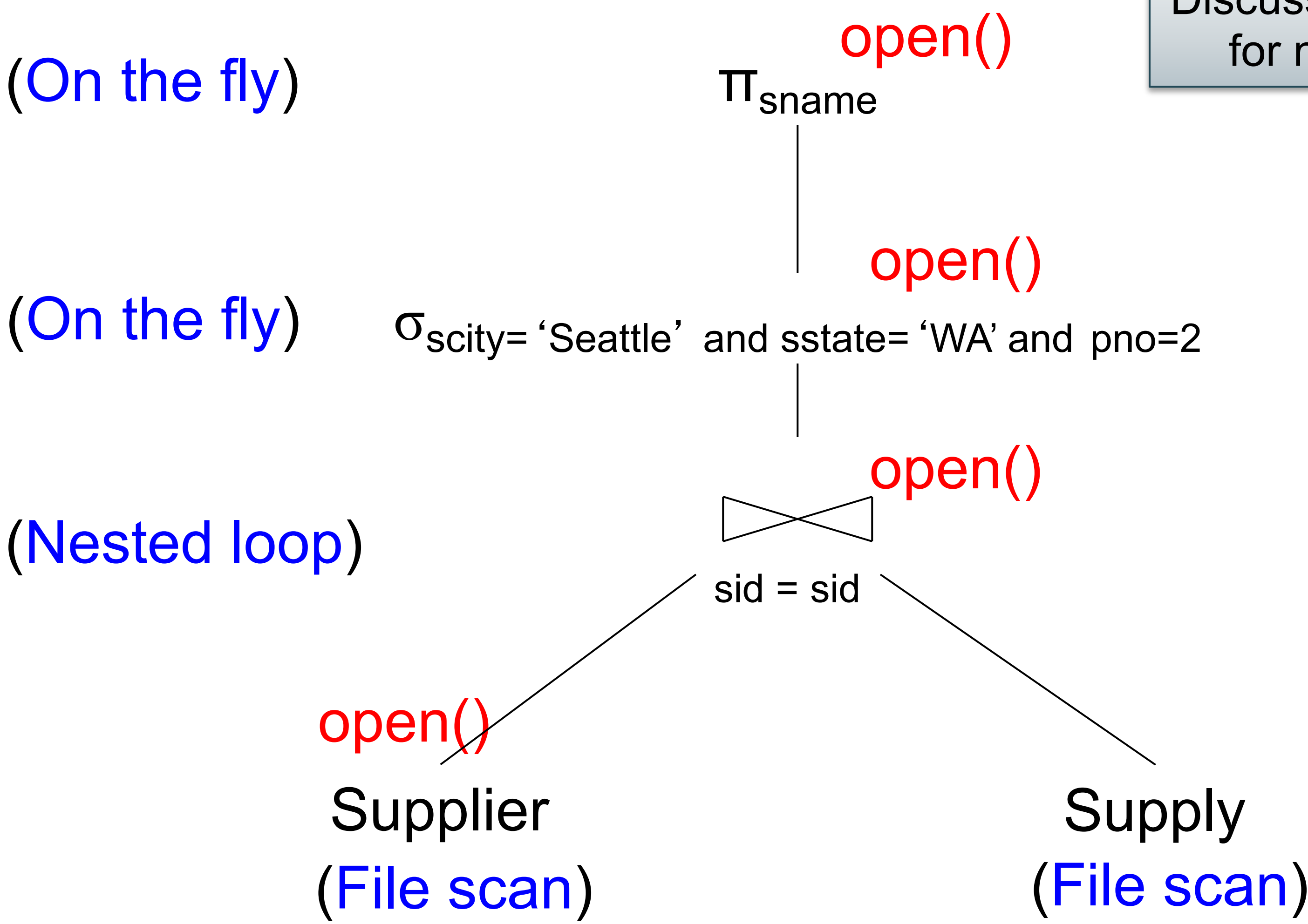


Supplier(sid, sname, scity, sstate)

Supply(sid, pno, quantity)

Pipelining

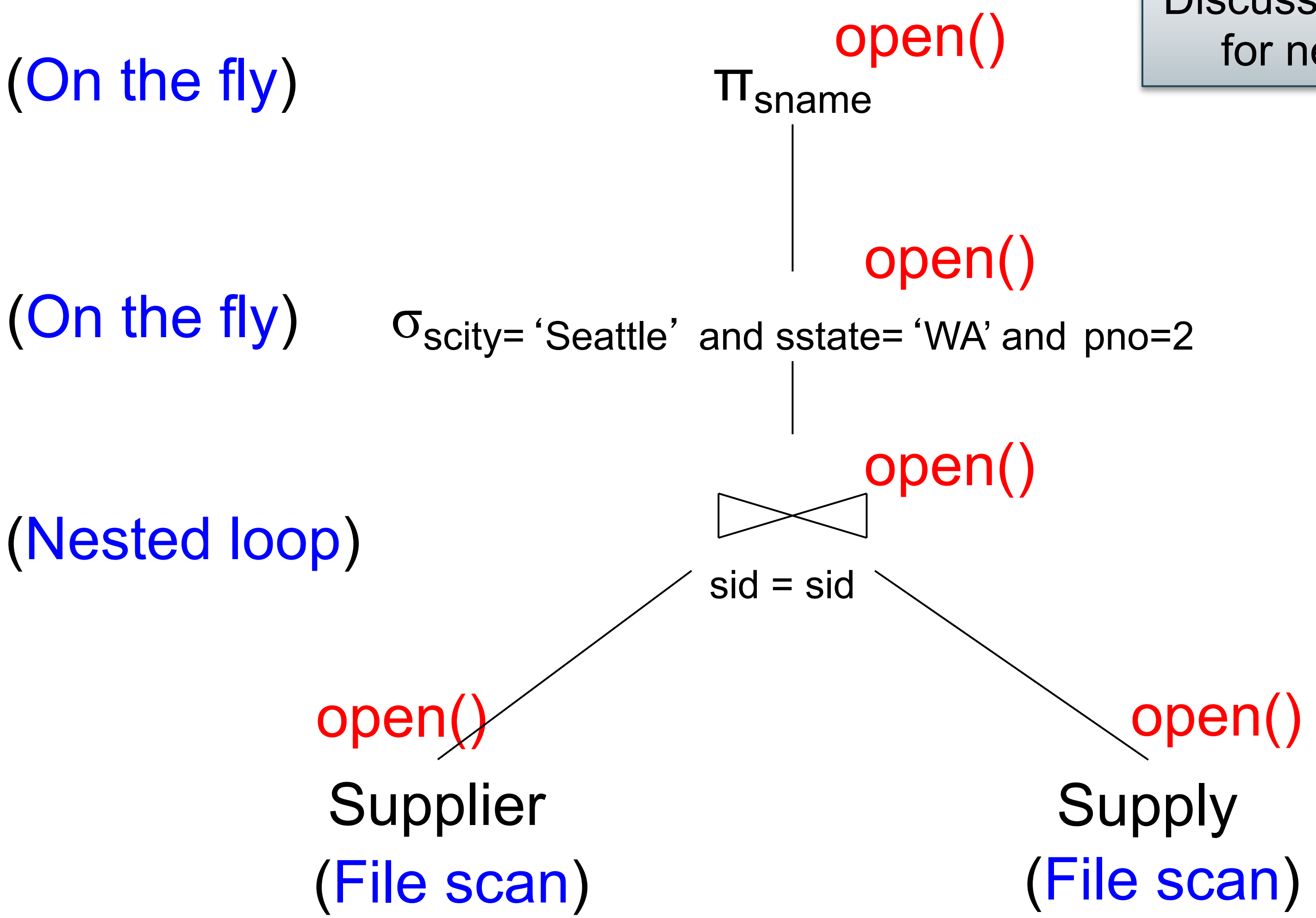
Discuss: open/next/close for nested loop join



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

Pipelining

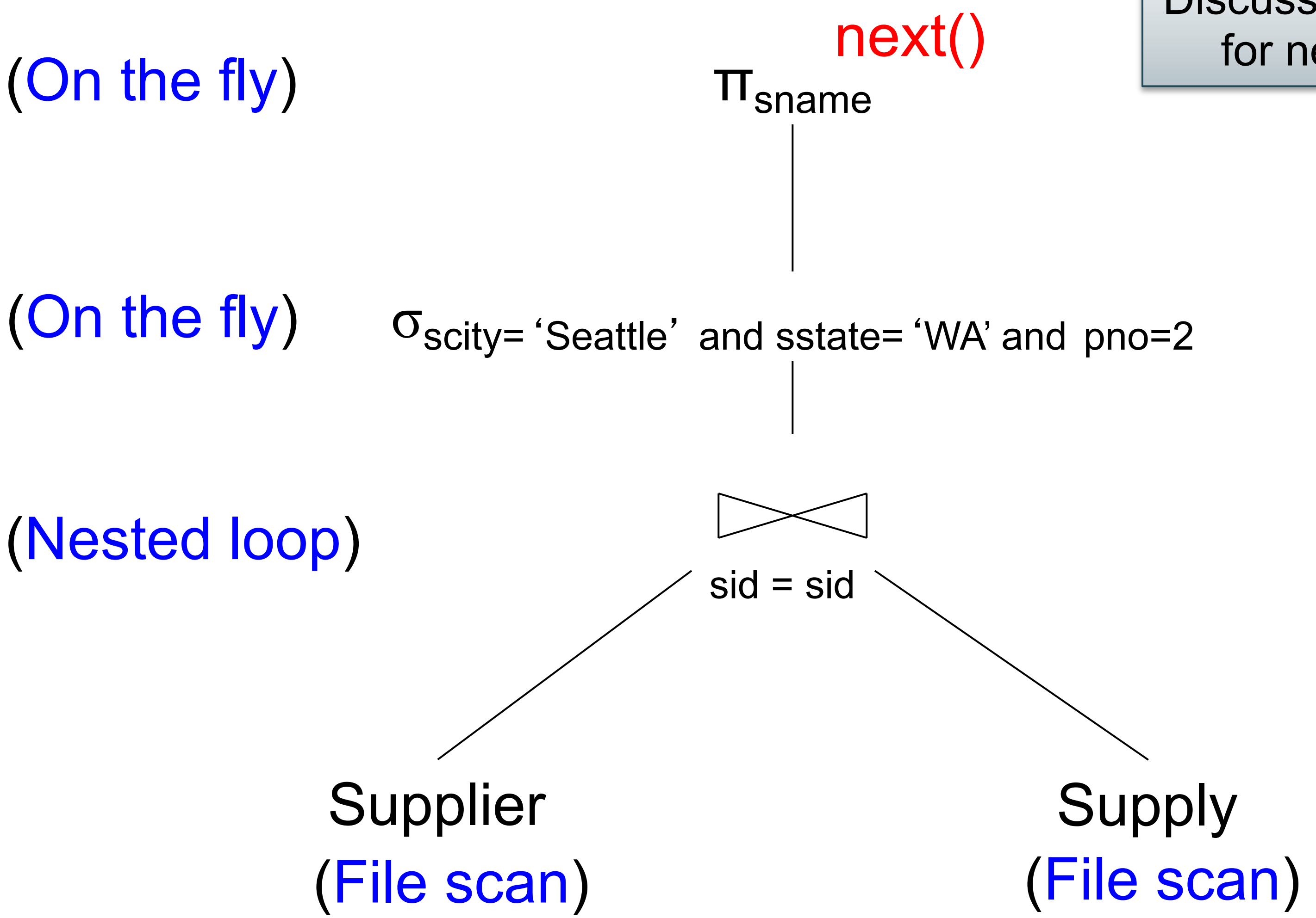
Discuss: open/next/close for nested loop join



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

Pipelining

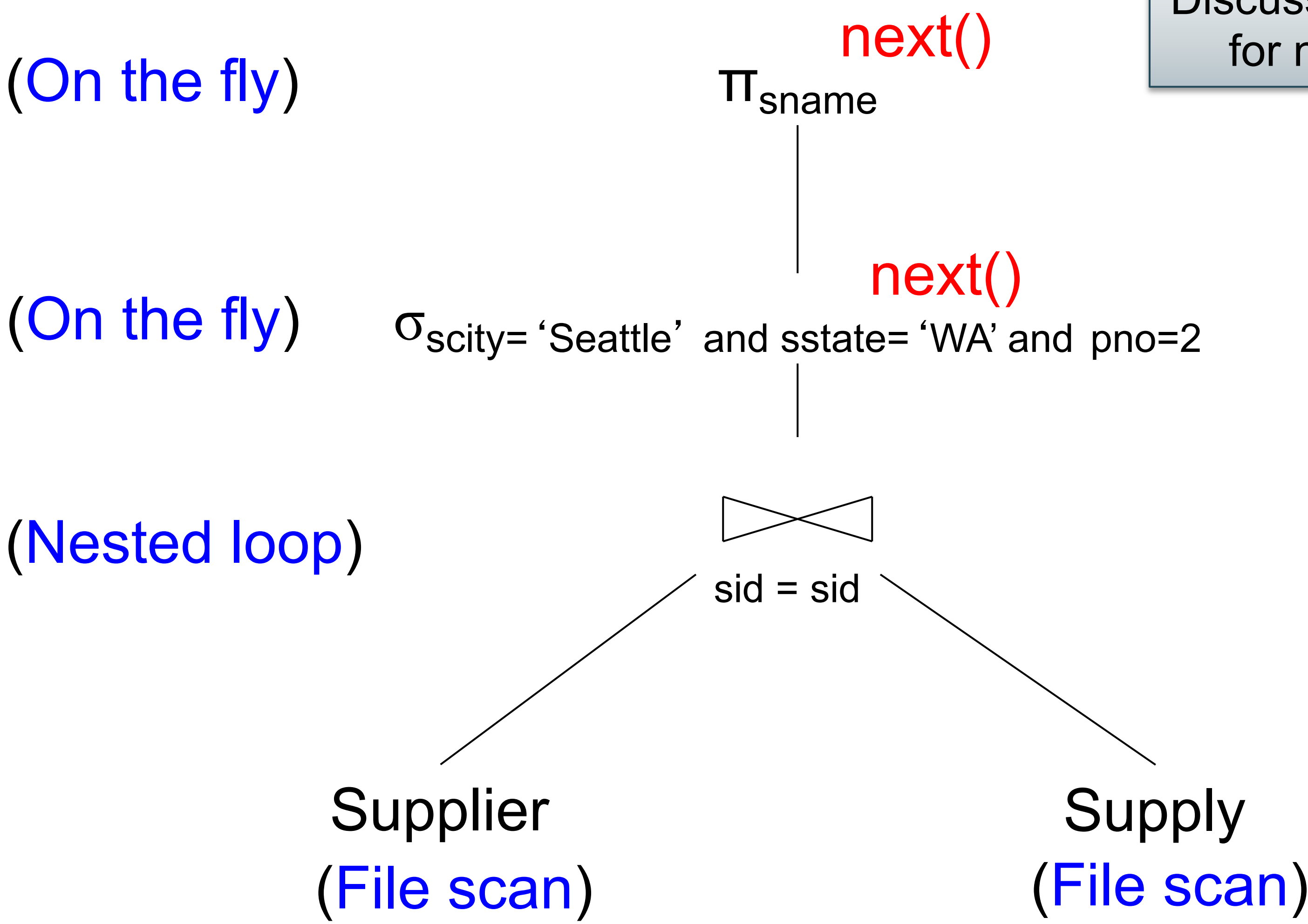
Discuss: open/next/close for nested loop join



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

Pipelining

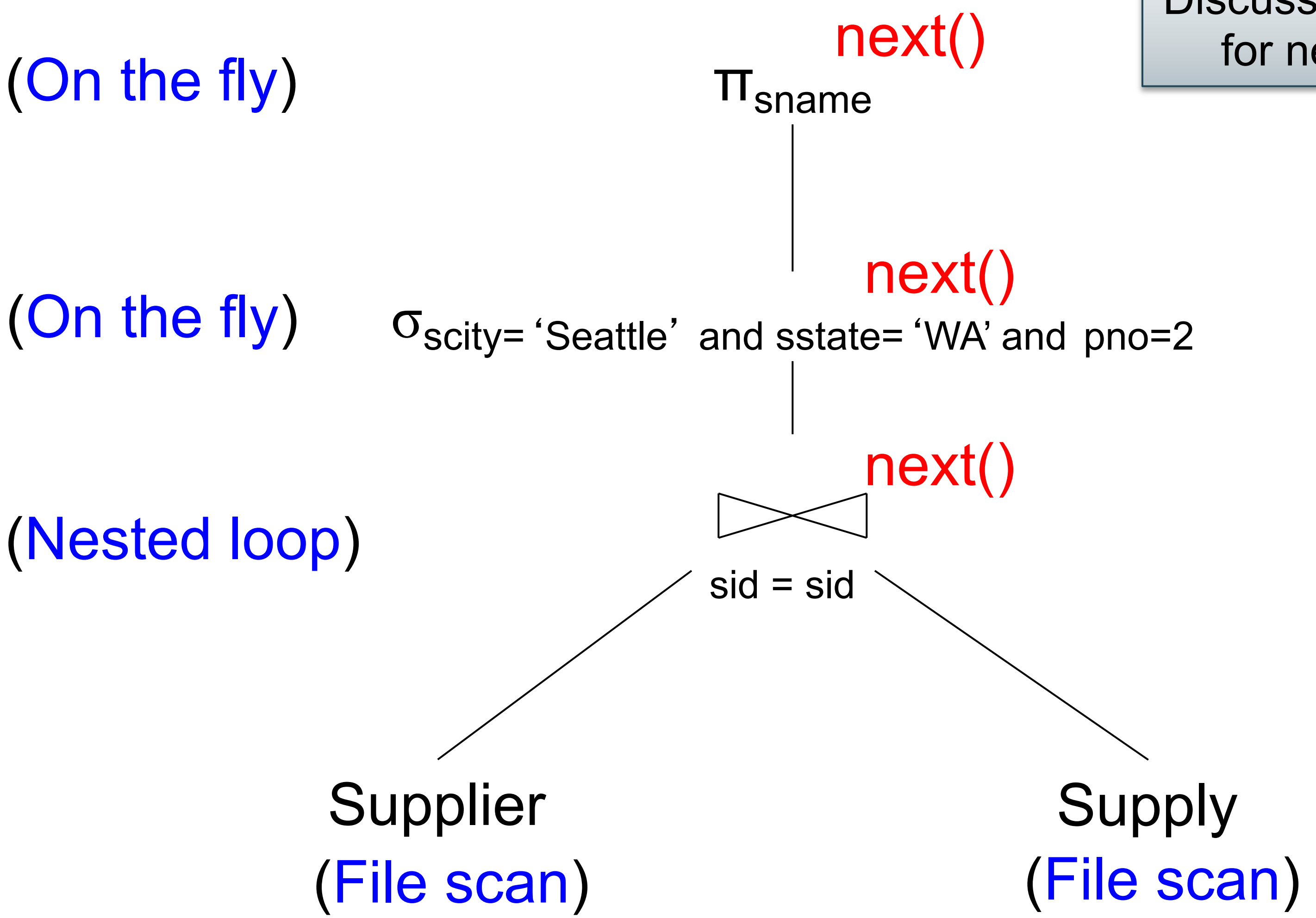
Discuss: open/next/close for nested loop join



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

Pipelining

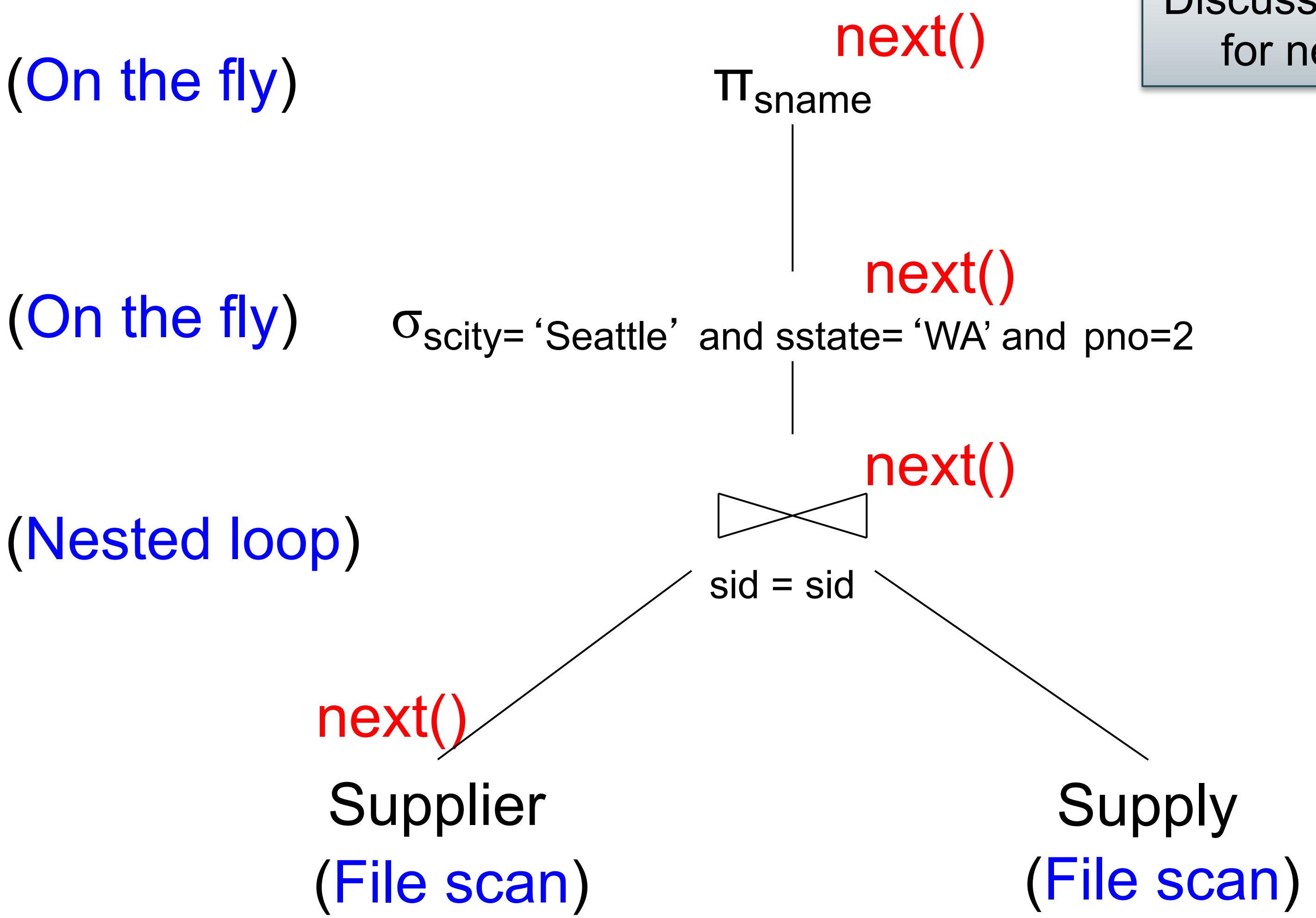
Discuss: open/next/close for nested loop join



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

Pipelining

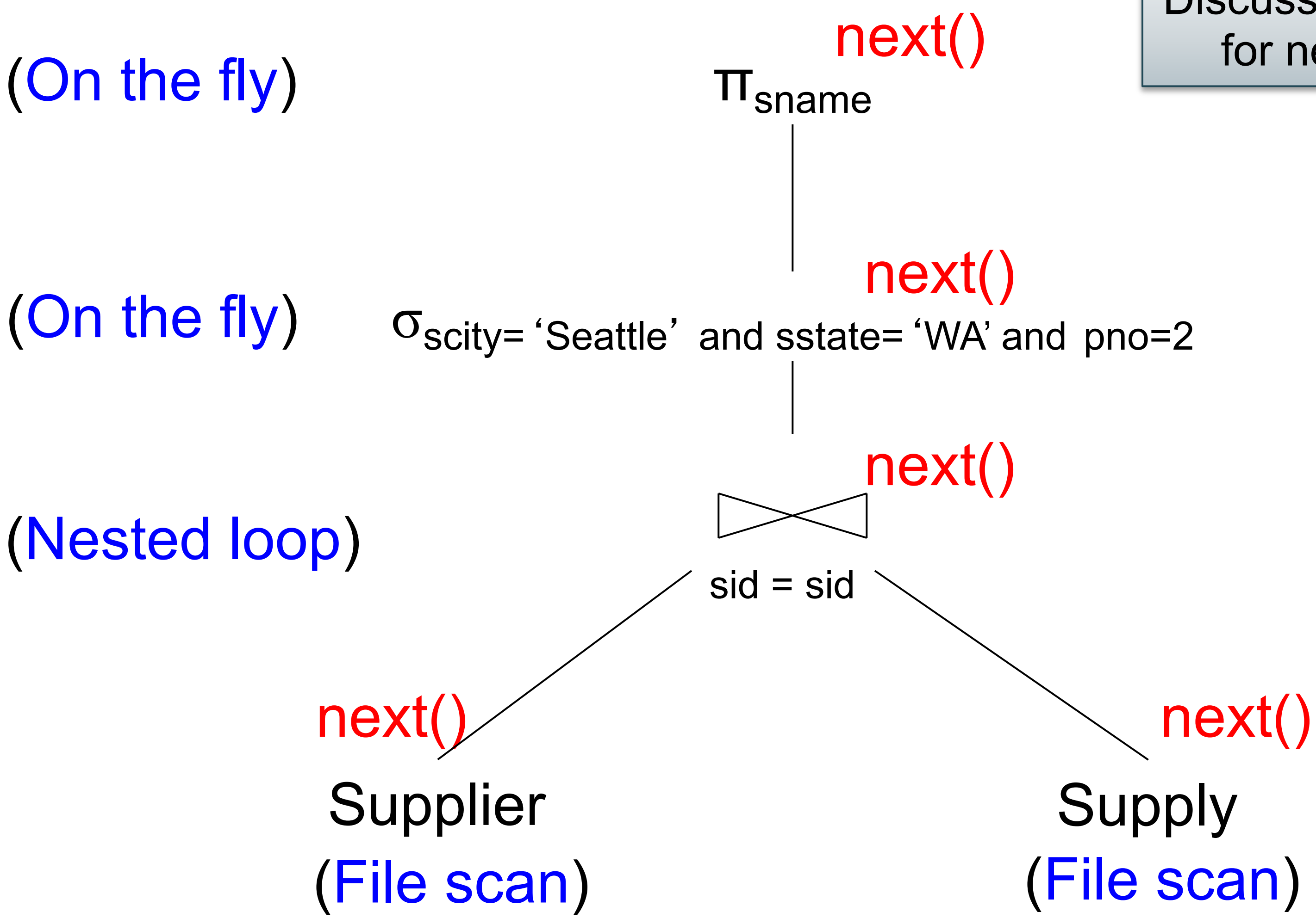
Discuss: open/next/close for nested loop join



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

Pipelining

Discuss: open/next/close for nested loop join

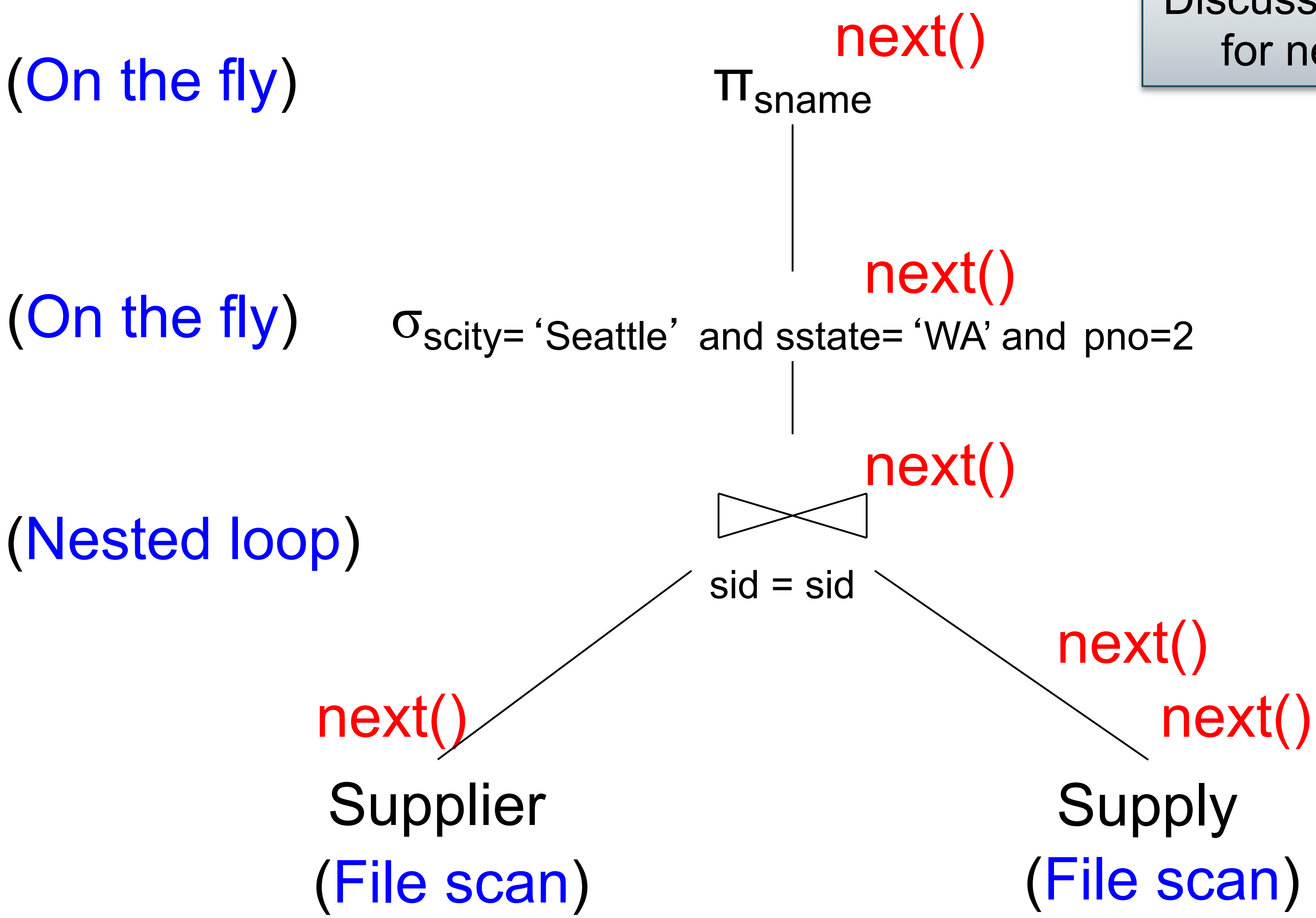


Supplier(sid, sname, scity, sstate)

Supply(sid, pno, quantity)

Pipelining

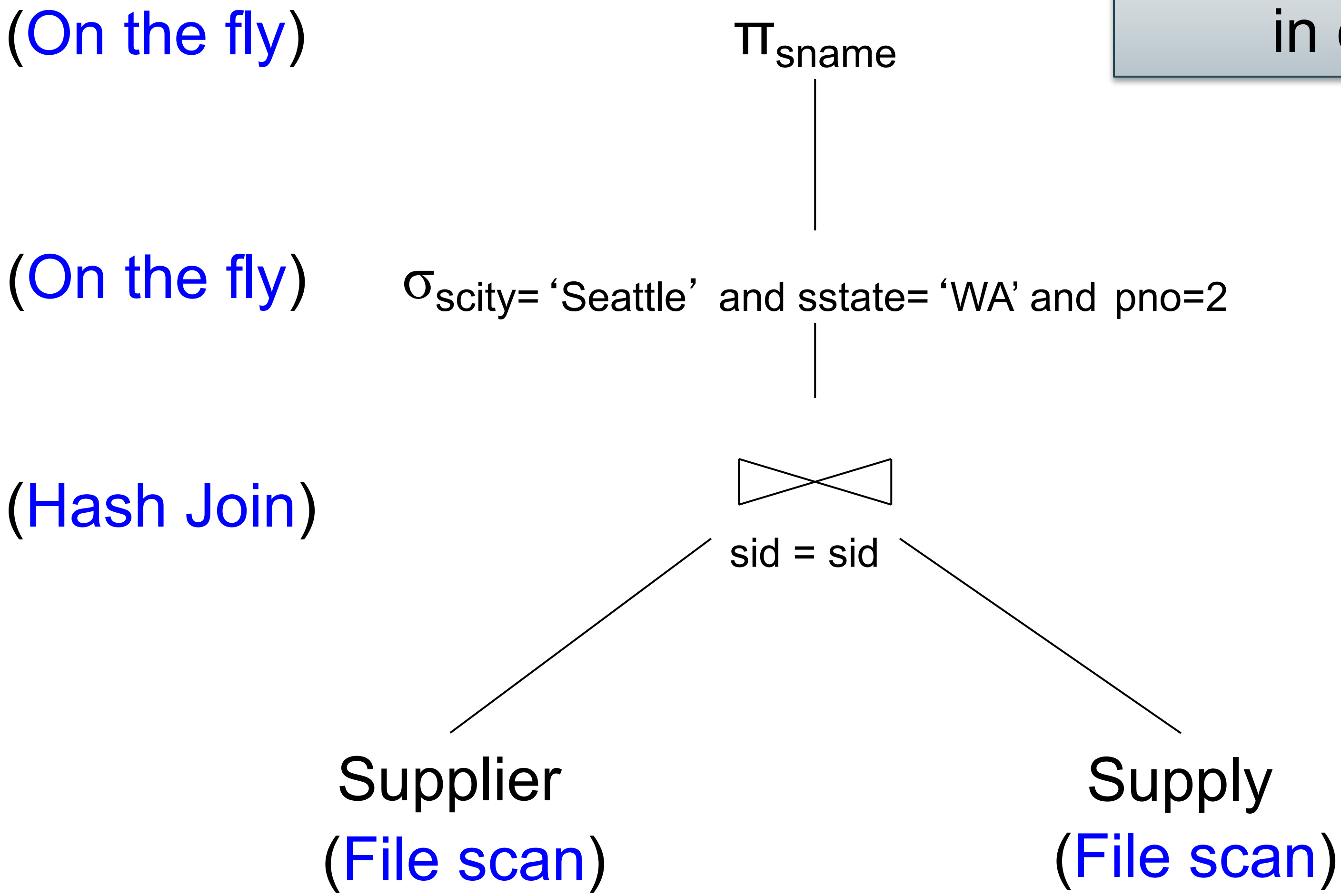
Discuss: open/next/close for nested loop join



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

Pipelining

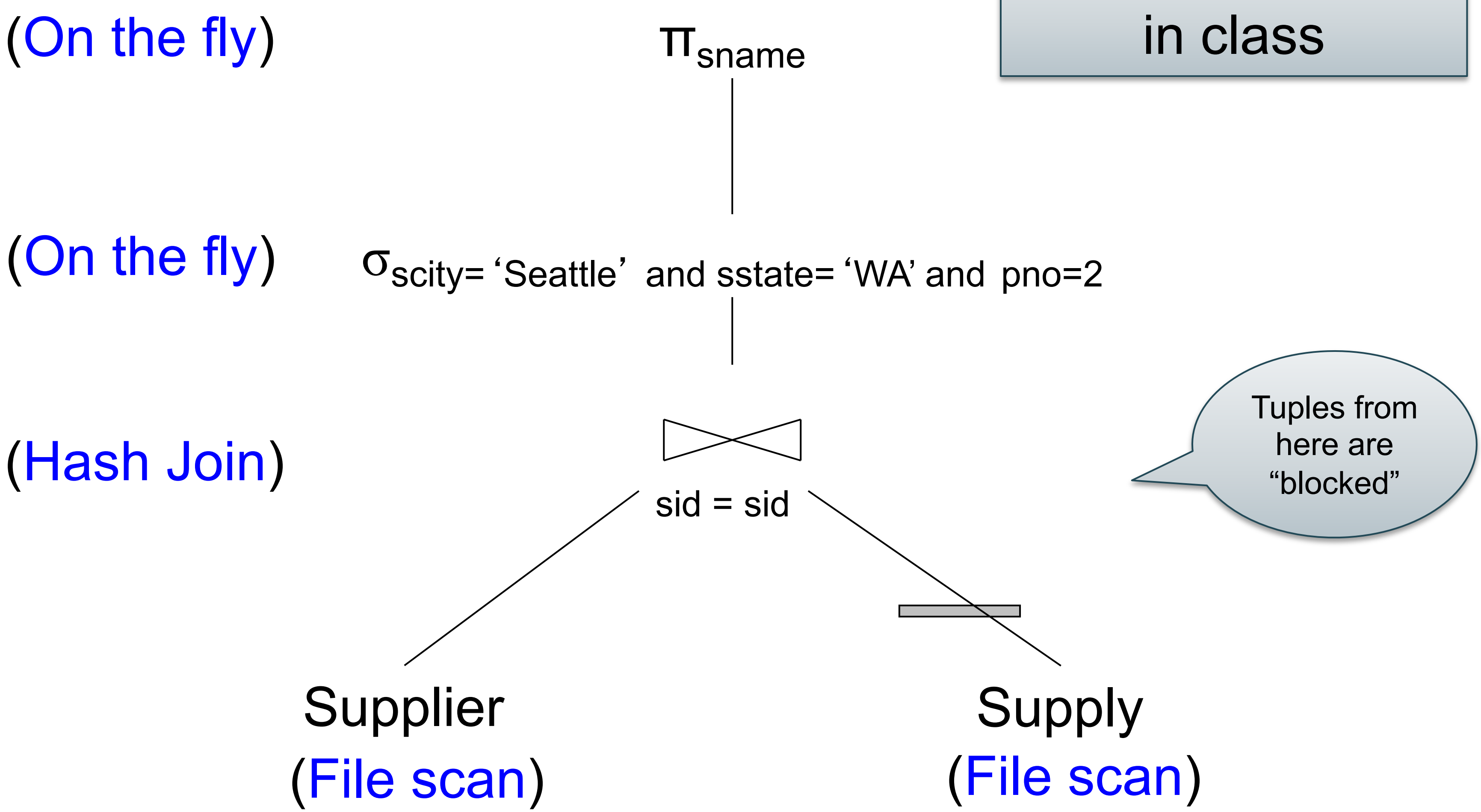
Discuss hash-join in class



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

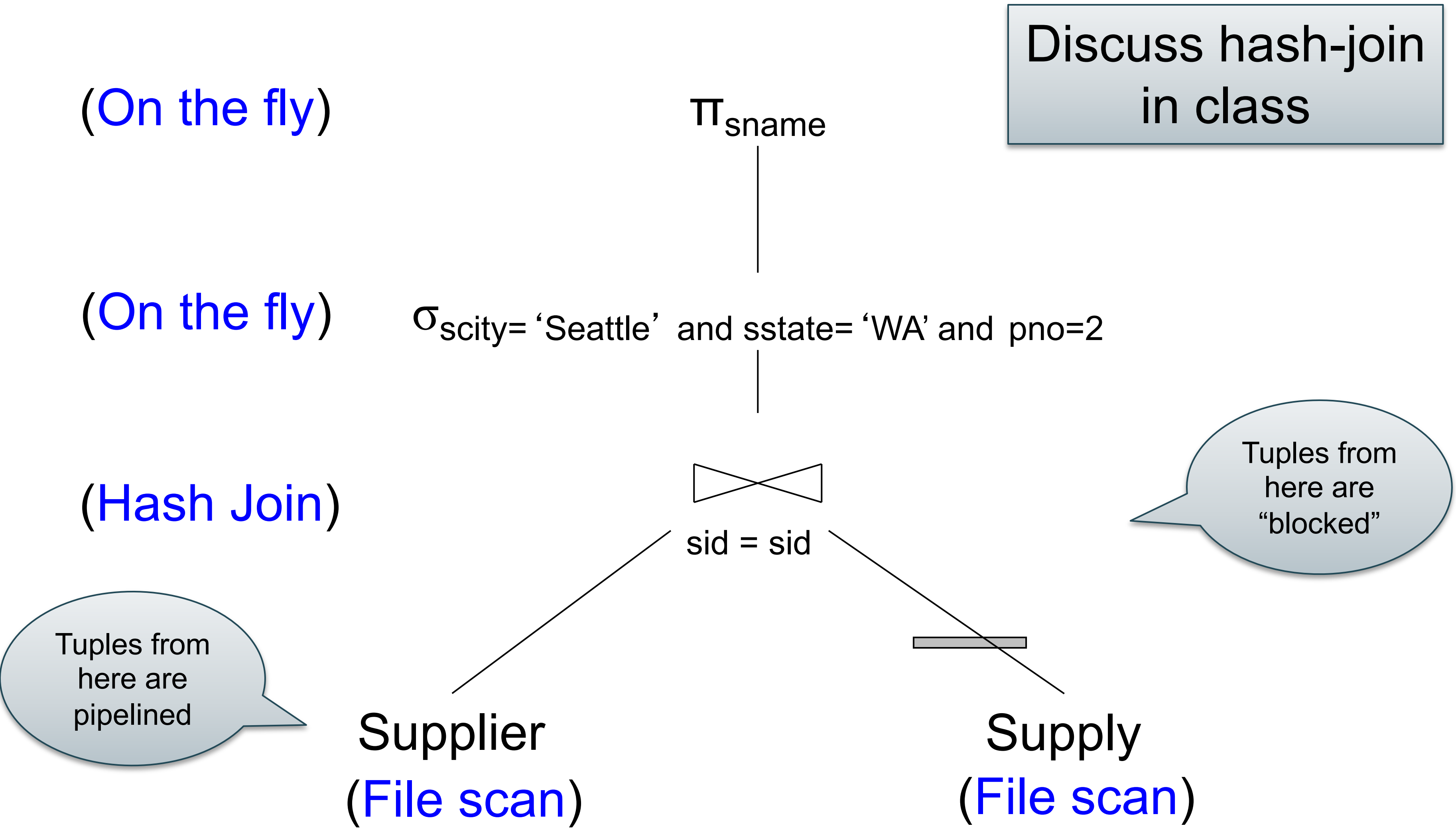
Pipelining

Discuss hash-join in class



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

Pipelining



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

Blocked Execution

(On the fly) π_{sname}

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

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

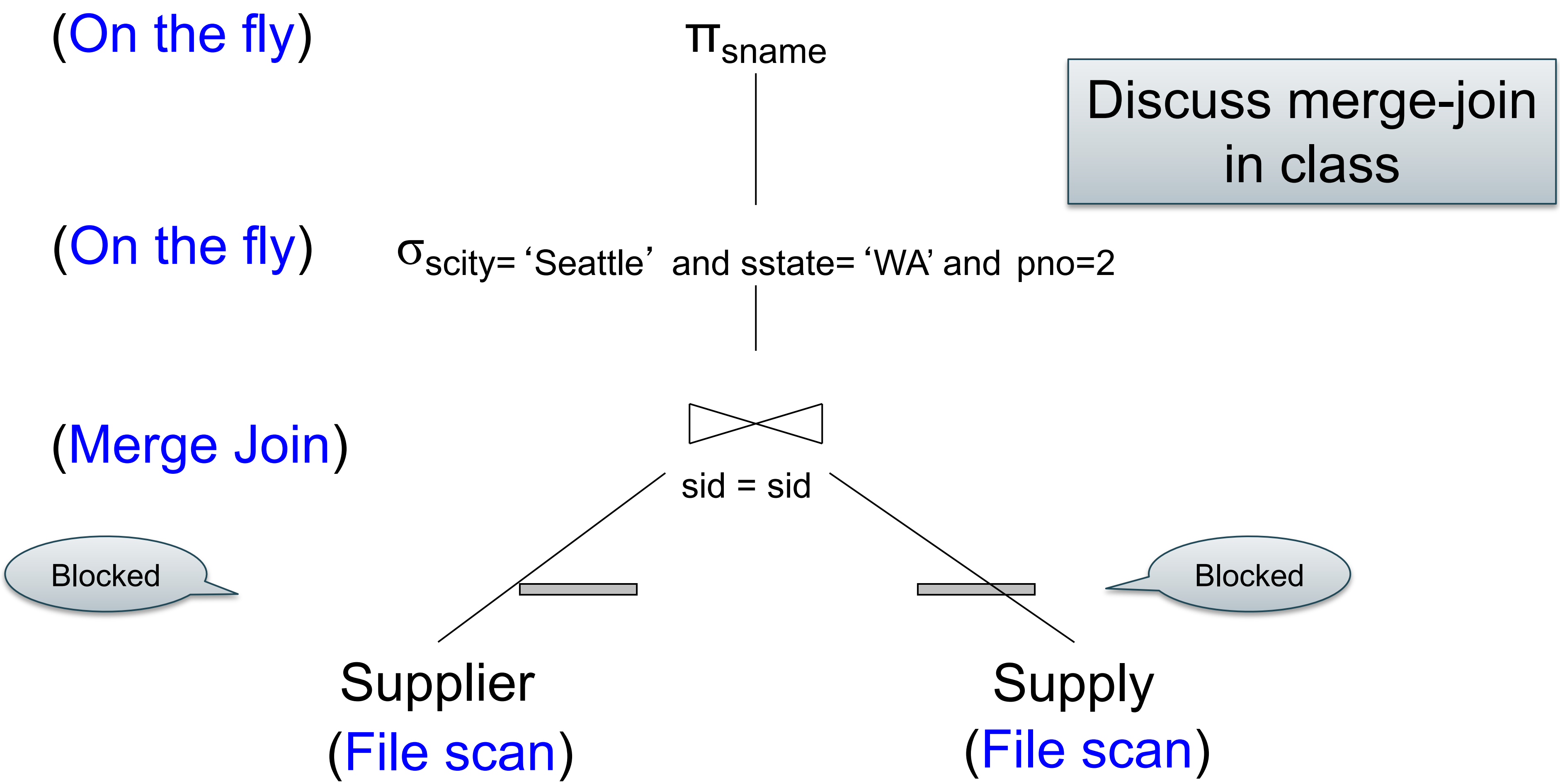
Supplier (File scan)

Supply (File scan)

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 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

Introduction to Database Systems
CSE 344

Lecture 16:

Basics of Data Storage and Indexes

Query Performance

To understand query performance and query optimization we need to understand:

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

We focus on **disk-based** DBMSs

Hard Disk

- Disks are mechanical devices
- A *block* = unit of read/write
- Once in main memory we call it a *page*
- Read only at the rotation speed
- Consequence: sequential scan faster than random
  - Fast: read blocks 1,2,3,4,5,…
  - Slow: read blocks 2342, 11, 321,9, …
- Rule of thumb:
  - Random read 1-2% of file ≈ sequential scan entire file;
  - 1-2% decreases over time, because of increased density

Student

ID	fName	lName
10	Tom	Hanks
20	Amy	Hanks
…		

Data Storage

- 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*

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:
  - Key = an attribute value (e.g., student ID or name)
  - Value = a pointer to the record OR the record itself

Index

- An **additional** file, that allows fast access to records in the data file given a search key
- The index contains (key, value) pairs:
  - Key = an attribute value (e.g., student ID or name)
  - Value = a pointer to the record OR the record itself
- Could have many indexes for one table

Key = means here search key

This  Is Not A Key

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

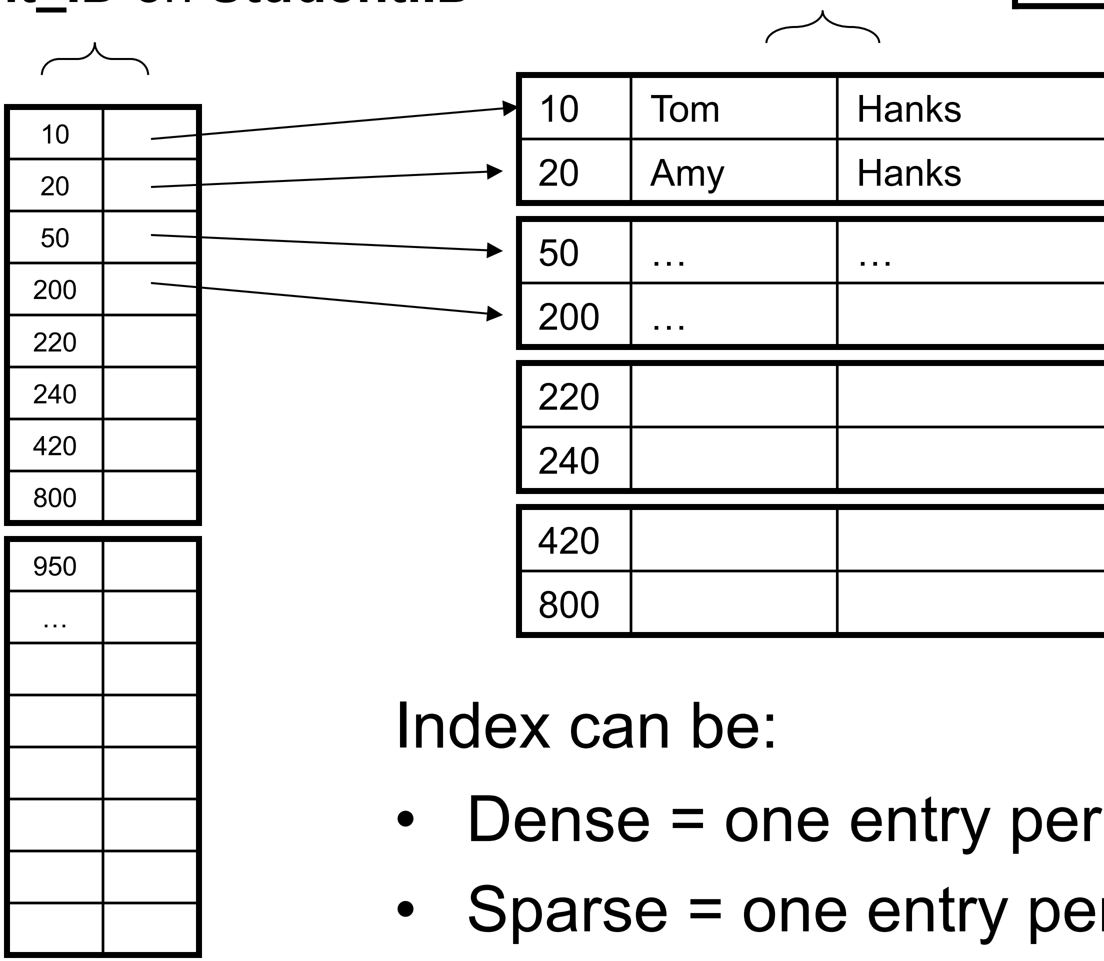
Example 1: Index on ID

Student

ID	fName	lName
10	Tom	Hanks
20	Amy	Hanks
…		

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

Data File **Student**



Index can be:

- Dense = one entry per record
- Sparse = one entry per block

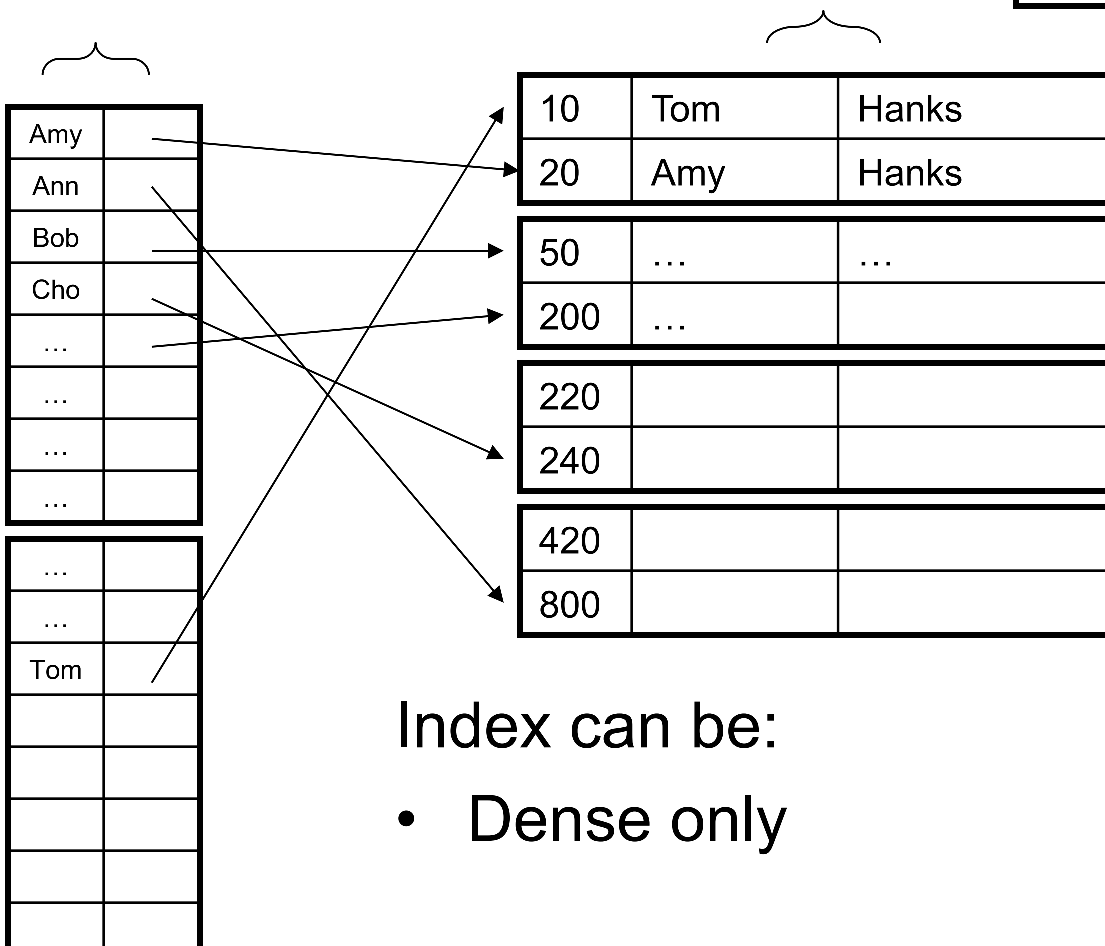
Student

ID	fName	lName
10	Tom	Hanks
20	Amy	Hanks
…		

Example 2: Index on fName

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

Data File **Student**



Index can be:

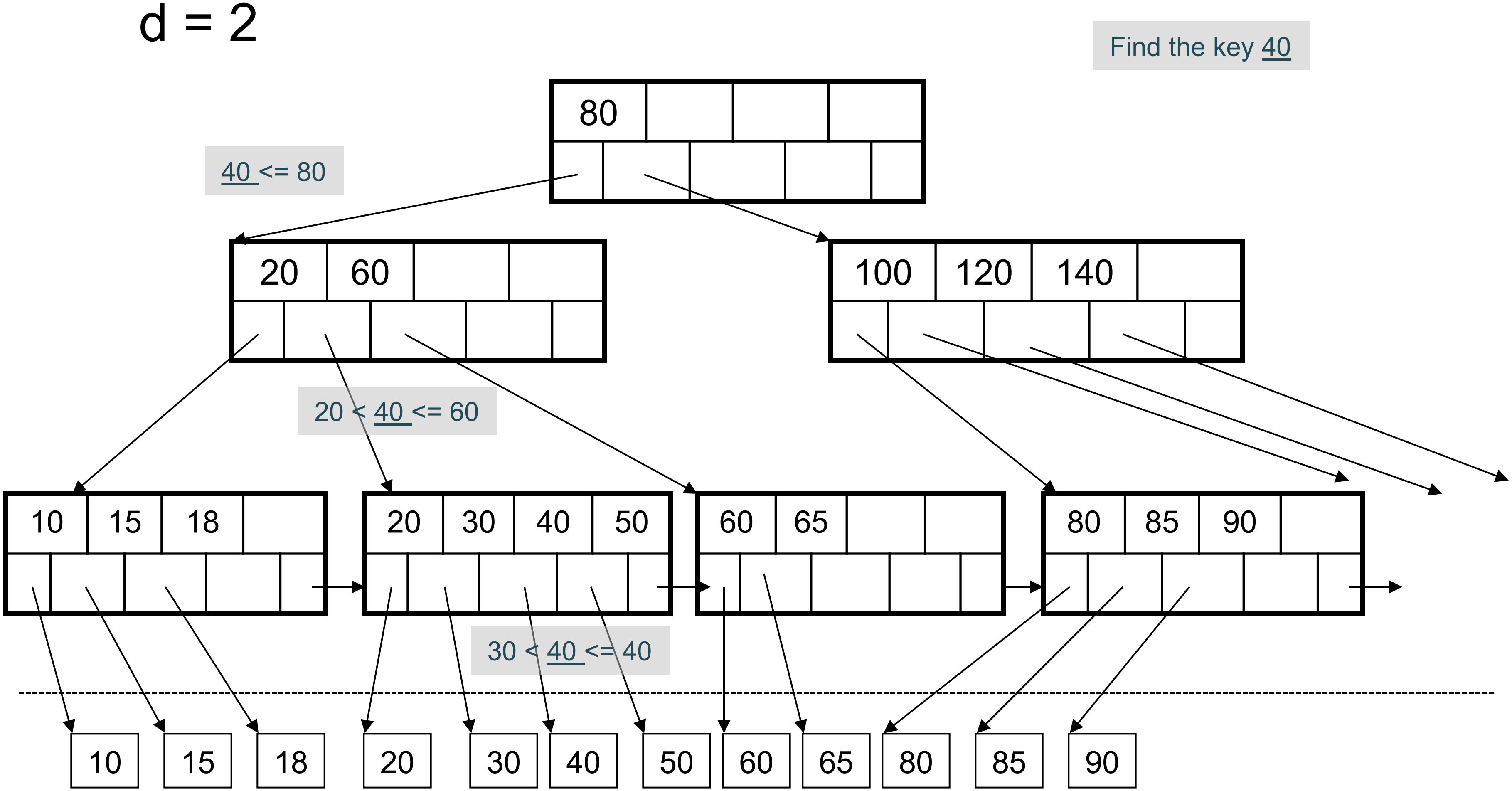
- Dense only

Index Organization

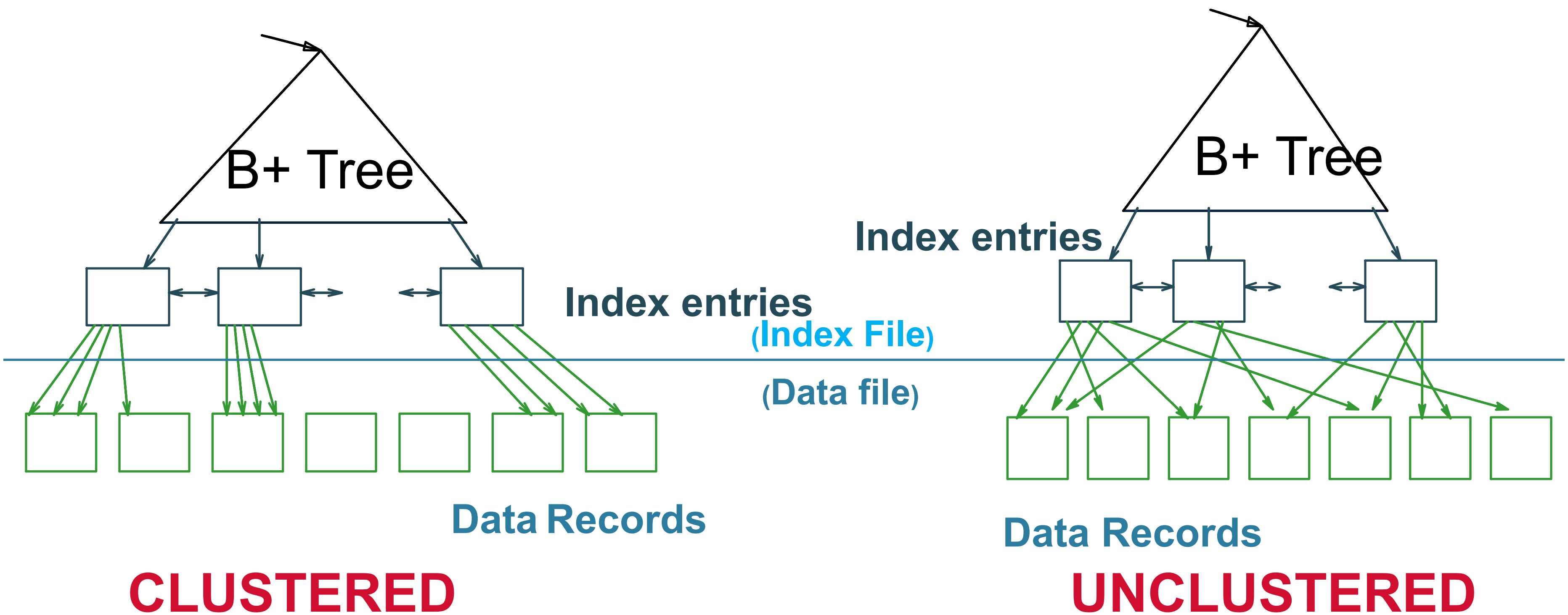
- Hash table
- B+ trees – most common
  - They are search trees, but they are not binary instead have higher fan-out
  - Will discuss them briefly next
- Specialized indexes: bit maps, R-trees, inverted index; won’t discuss

B+ Tree Index by Example

d = 2



Clustered vs 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

Summary So Far

- Index = a file that enables direct access to records in another data file
  - B+ tree / Hash table
  - Clustered/unclustered
- Data resides on disk
  - Organized in blocks
  - Sequential reads are efficient
  - Random access less efficient
  - Random read 1-2% of data worse than sequential

Main Memory Algorithms

- Selection σ
  - "on-the-fly"
  - Index-based selection
- Join $\bowtie$:
  - Nested loop join
  - Hash join
  - Merge join
  - Index join

Student(ID, fname, lname)
Takes(studentID, courseID)

Selection

```
SELECT *
FROM  Takes y
WHERE y.courseID = 300
```

σ_{300}

|

Takes

Logical plan

Student(<u>ID</u>, fname, lname)

Takes(studentID, courseID)

Selection

```
SELECT *
FROM  Takes y
WHERE y.courseID = 300
```

$$\sigma_{300}$$

|

Takes

Logical plan

Physical plan

On-the-fly σ_{300}

|

Takes

On-the-fly selection

Student(ID, fname, lname)
Takes(studentID, courseID)

Selection

```
SELECT *
FROM  Takes y
WHERE y.courseID = 300
```

σ_{300}
|
Takes

Logical plan

On-the-fly selection

On-the-fly σ_{300}
|
Takes

Physical plan

for y **in** Takes
  if courseID = 300 **then**
    output y

Student(<u>ID</u>, fname, lname)
Takes(studentID, courseID)

Selection

```
SELECT *
FROM  Takes y
WHERE y.courseID = 300
```

σ_{300}
|
Takes

Logical plan

Physical plan

for y **in** Takes
  if courseID = 300 **then**
    output y

On-the-fly selection

On-the-fly σ_{300}
|
Takes

Index-based selection:
Takes_courseID =
index on Takes.courseID

Index-based σ_{300}
|
Takes

Student(ID, fname, lname)
Takes(studentID, courseID)

Selection

```
SELECT *
FROM  Takes y
WHERE y.courseID = 300
```

σ_{300}
|
Takes

Logical plan

Physical plan

On-the-fly selection

On-the-fly σ_{300}
|
Takes

```
for y in Takes
    if courseID = 300 then
        output y
```

Index-based selection:

Takes_courseID =
index on Takes.courseID

Index-based σ_{300}
|
Takes

```
rid_list = Takes_courseID.get(300)
for r in rid_list
    y = Student.getRecord(rid)
    output y
```

Student(<u>ID</u>, fname, lname)
Takes(studentID, courseID)

Selection

```
SELECT *
FROM  Takes y
WHERE y.courseID = 300
```

σ_{300}
|
Takes

Logical plan

Physical plan

On-the-fly selection

On-the-fly σ_{300}
|
Takes

```
for y in Takes
    if courseID = 300 then
        output y
```

Index-based selection:

Takes_courseID =
index on Takes.courseID

Index-based σ_{300}
|
Takes

```
rid_list = Takes_courseID.get(300)
for r in rid_list
    y = Student.getRecord(rid)
    output y
```

$\bowtie$ of σ_{300} (over Takes_courseID) and Takes

SQL Server represents index-based selection as a join

Student(ID, fname, lname)
Takes(studentID, courseID)

Discussion

Can the optimizer use the index Takes_courseID to answer these queries?

```
SELECT *
FROM  Takes y
WHERE y.courseID = 300 or y.courseID = 444
```

```
(SELECT *
 FROM  Takes y
 WHERE y.courseID = 300)
      UNION ALL
(SELECT *
 FROM Takes Y
 WHERE y.courseID = 444)
```

Student(ID, fname, lname)
Takes(studentID, courseID)

Discussion

Can the optimizer use the index Takes_courseID to answer these queries?

```
SELECT *
FROM  Takes y
WHERE y.courseID = 300 or y.courseID = 444
```

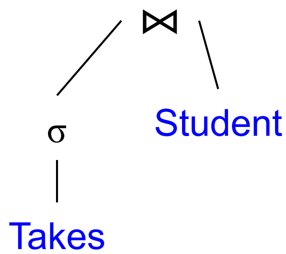
Probably not

```
(SELECT *
 FROM  Takes y
 WHERE y.courseID = 300)
      UNION ALL
(SELECT *
 FROM Takes Y
 WHERE y.courseID = 444)
```

Yes

Recall HW3!!

Student(<u>ID</u>, fname, lname)
Takes(studentID, courseID)




```
SELECT *
FROM  Student x, Takes y
WHERE x.ID=y.studentID AND y.courseID = 300
```

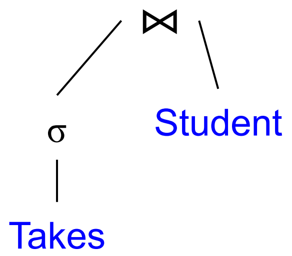
Join

Nested Loop Join:

```
for y in Takes
  if courseID = 300 then
    for x in Student
        if x.ID=y.studentID
          output *
```

Student(<u>ID</u>, fname, lname)

Takes(studentID, courseID)




```
SELECT *
FROM  Student x, Takes y
WHERE x.ID=y.studentID AND y.courseID = 300
```

Join

Nested Loop Join:

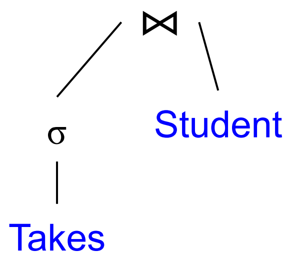
```
for y in Takes
    if courseID = 300 then
      for x in Student
            if x.ID=y.studentID
              output *
```

Index Join:

assume the database has these indexes

- **Takes_courseID** = on Takes.courseID
- **Student_ID** = on Student.ID

Student(ID, fname, lname)
Takes(studentID, courseID)




```
SELECT *
FROM  Student x, Takes y
WHERE x.ID=y.studentID AND y.courseID = 300
```

Join

Nested Loop Join:

```
for y in Takes
    if courseID = 300 then
      for x in Student
            if x.ID=y.studentID
              output *
```

Index Join:

```
for y' in Takes_courseID.get(300)
   y = Takes.getRecord(y')
```

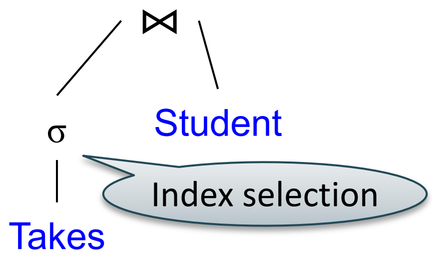
assume the database has these indexes

- **Takes_courseID** = on Takes.courseID
- **Student_ID** = on Student.ID

Student(<u>ID</u>, fname, lname)
Takes(studentID, courseID)

```
SELECT *
FROM  Student x, Takes y
WHERE x.ID=y.studentID AND y.courseID = 300
```

Join



Nested Loop Join:

```
for y in Takes
   if courseID = 300 then
     for x in Student
          if x.ID=y.studentID
            output *
```

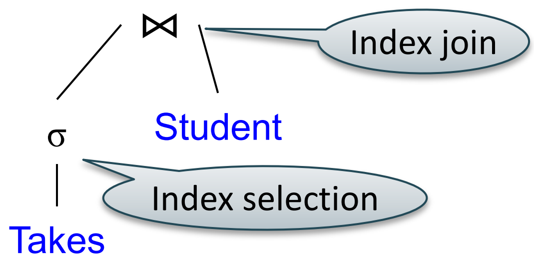
Index Join:

```
for y' in Takes_courseID.get(300)
  y = Takes.getRecord(y')
```

assume the database has these indexes

- **Takes_courseID** = on Takes.courseID
- **Student_ID** = on Student.ID

Student(ID, fname, lname)
Takes(studentID, courseID)




```
SELECT *
FROM  Student x, Takes y
WHERE x.ID=y.studentID AND y.courseID = 300
```

Join

Nested Loop Join:

```
for y in Takes
    if courseID = 300 then
      for x in Student
            if x.ID=y.studentID
              output *
```

Index Join:

assume the database has these indexes

- **Takes_courseID** = on Takes.courseID
- **Student_ID** = on Student.ID

```
for y’ in Takes_courseID.get(300)
  y = Takes.getRecord(y’)
  x’ = Student_ID .get(y.studentID)
  x =  Student.getRecord(x’)
  output *
```

Getting Practical: Creating Indexes in SQL

```
CREATE  TABLE    V(M int,   N text,    P int);
```

```
CREATE  INDEX V1 ON V(N)
```

Getting Practical: Creating Indexes in SQL

```
CREATE  TABLE   V(M int,  N text,   P int);
```

```
CREATE  INDEX V1 ON V(N)
```

```
CREATE  INDEX V2 ON V(P, M)
```

Getting Practical: Creating Indexes in SQL

```
CREATE  TABLE   V(M int,  N text,   P int);
```

```
CREATE  INDEX V1 ON V(N)
```

```
CREATE  INDEX V2 ON V(P, M)
```

What does this mean?

Getting Practical: Creating Indexes in SQL

```
CREATE TABLE V(M int, N text, P int);
```

```
CREATE INDEX V1 ON V(N)
```

```
CREATE INDEX V2 ON V(P, M)
```

What does this mean?

yes

```
select *
from V
where P=55 and M=77
```

Getting Practical: Creating Indexes in SQL

```
CREATE  TABLE   V(M int,   N text,   P int);
```

```
CREATE  INDEX V1 ON V(N)
```

```
CREATE  INDEX V2 ON V(P, M)
```

What does this mean?

yes

```
select *
from V
where P=55 and M=77
```

```
select *
from V
where P=55
```

Getting Practical: Creating Indexes in SQL

```
CREATE TABLE V(M int, N text, P int);
```

```
CREATE INDEX V1 ON V(N)
```

```
CREATE INDEX V2 ON V(P, M)
```

What does this mean?

yes

```
select *
from V
where P=55 and M=77
```

```
select *
from V
where P=55
```

yes

Getting Practical: Creating Indexes in SQL

```
CREATE TABLE V(M int, N text, P int);
```

```
CREATE INDEX V1 ON V(N)
```

```
CREATE INDEX V2 ON V(P, M)
```

What does this mean?

```
select *
from V
where P=55 and M=77
```

yes

```
select *
from V
where P=55
```

yes

```
select *
from V
where M=77
```

Getting Practical: Creating Indexes in SQL

```
CREATE  TABLE   V(M int,  N text,   P int);
```

```
CREATE  INDEX V1 ON V(N)
```

```
CREATE  INDEX V2 ON V(P, M)
```

What does this mean?

```
select *
from V
where P=55 and M=77
```

yes

```
select *
from V
where P=55
```

yes

```
select *
from V
where M=77
```

no

Getting Practical: Creating Indexes in SQL

```
CREATE  TABLE   V(M int,   N text,    P int);
```

```
CREATE  INDEX V1 ON V(N)
```

```
CREATE  INDEX V2 ON V(P, M)
```

What does this mean?

```
CREATE  INDEX V3 ON V(M, N)
```

```
CREATE UNIQUE INDEX V4 ON V(N)
```

```
CREATE CLUSTERED INDEX V5 ON V(N)
```

Not supported in SQLite

```
select *
from V
where P=55 and M=77
```

yes

```
select *
from V
where P=55
```

yes

```
select *
from V
where M=77
```

no

Student

ID	fName	lName
10	Tom	Hanks
20	Amy	Hanks
…		

Which Indexes?

- How many indexes could we create?
- Which indexes should we create?

Student

ID	fName	lName
10	Tom	Hanks
20	Amy	Hanks
…		

Which Indexes?

- How many indexes could we create?
- Which indexes should we create?

This is called the *Index Selection Problem*

(not to be confused with the *index selection* operator!)

The Index Selection Problem 1

```
V(M, N, P);
```

Your workload is this

100000 queries:

```
SELECT *
FROM V
WHERE N=?
```

100 queries:

```
SELECT *
FROM V
WHERE P=?
```

The Index Selection Problem 1

V(M, N, P);

Your workload is this

100000 queries:

```
SELECT *
FROM V
WHERE N=?
```

100 queries:

```
SELECT *
FROM V
WHERE P=?
```

What indexes ?

The Index Selection Problem 1

V(M, N, P);

Your workload is this

100000 queries:

```
SELECT *
FROM V
WHERE N=?
```

100 queries:

```
SELECT *
FROM V
WHERE P=?
```

A: V(N) and V(P) (hash tables or B-trees)

The Index Selection Problem 2

```
V(M, N, P);
```

Your workload is this

100000 queries:

```
SELECT *
FROM V
WHERE N>? and N<?
```

100 queries:

```
SELECT *
FROM V
WHERE P=?
```

100000 queries:

```
INSERT INTO V
VALUES (?, ?, ?)
```

What indexes ?

The Index Selection Problem 2

V(M, N, P);

Your workload is this

100000 queries:

```
SELECT *
FROM V
WHERE N>? and N<?
```

100 queries:

```
SELECT *
FROM V
WHERE P=?
```

100000 queries:

```
INSERT INTO V
VALUES (?, ?, ?)
```

A: definitely V(N) (must B-tree); unsure about V(P)

The Index Selection Problem 3

```
V(M, N, P);
```

Your workload is this

100000 queries:

```
SELECT *
FROM V
WHERE N=?
```

1000000 queries:

```
SELECT *
FROM V
WHERE N=? and P>?
```

100000 queries:

```
INSERT INTO V
VALUES (?, ?, ?)
```

What indexes ?

The Index Selection Problem 3

V(M, N, P);

Your workload is this

100000 queries:

```
SELECT *
FROM V
WHERE N=?
```

1000000 queries:

```
SELECT *
FROM V
WHERE N=? and P>?
```

100000 queries:

```
INSERT INTO V
VALUES (?, ?, ?)
```

A: V(N, P)

How does this index differ from:

1. Two indexes V(N) and V(P)?
2. An index V(P, N)?

The Index Selection Problem 4

```
V(M, N, P);
```

Your workload is this

1000 queries:

```
SELECT *
FROM V
WHERE N>? and N<?
```

100000 queries:

```
SELECT *
FROM V
WHERE P>? and P<?
```

What indexes ?

The Index Selection Problem 4

V(M, N, P);

Your workload is this

1000 queries:

```
SELECT *
FROM V
WHERE N>? and N<?
```

100000 queries:

```
SELECT *
FROM V
WHERE P>? and P<?
```

A: V(N) secondary, V(P) primary index

Two typical kinds of queries

```
SELECT *
FROM Movie
WHERE year = ?
```

- Point queries
- Hash- or B^+-tree index
- Clustered or not

```
SELECT *
FROM Movie
WHERE year >= ? AND
             year <= ?
```

- Range queries
- B^+-tree index
- Clustered

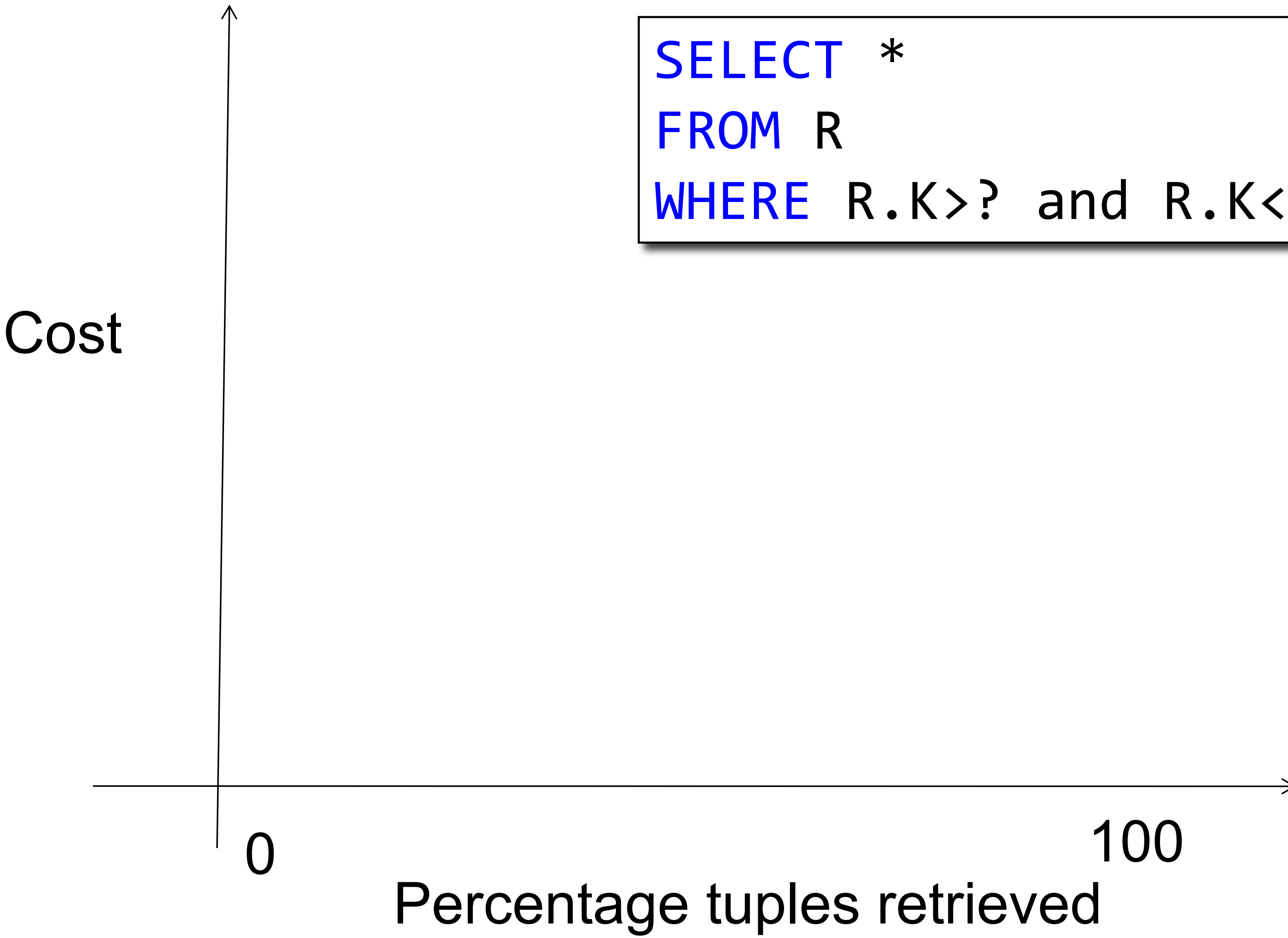
To Cluster or Not

Remember:

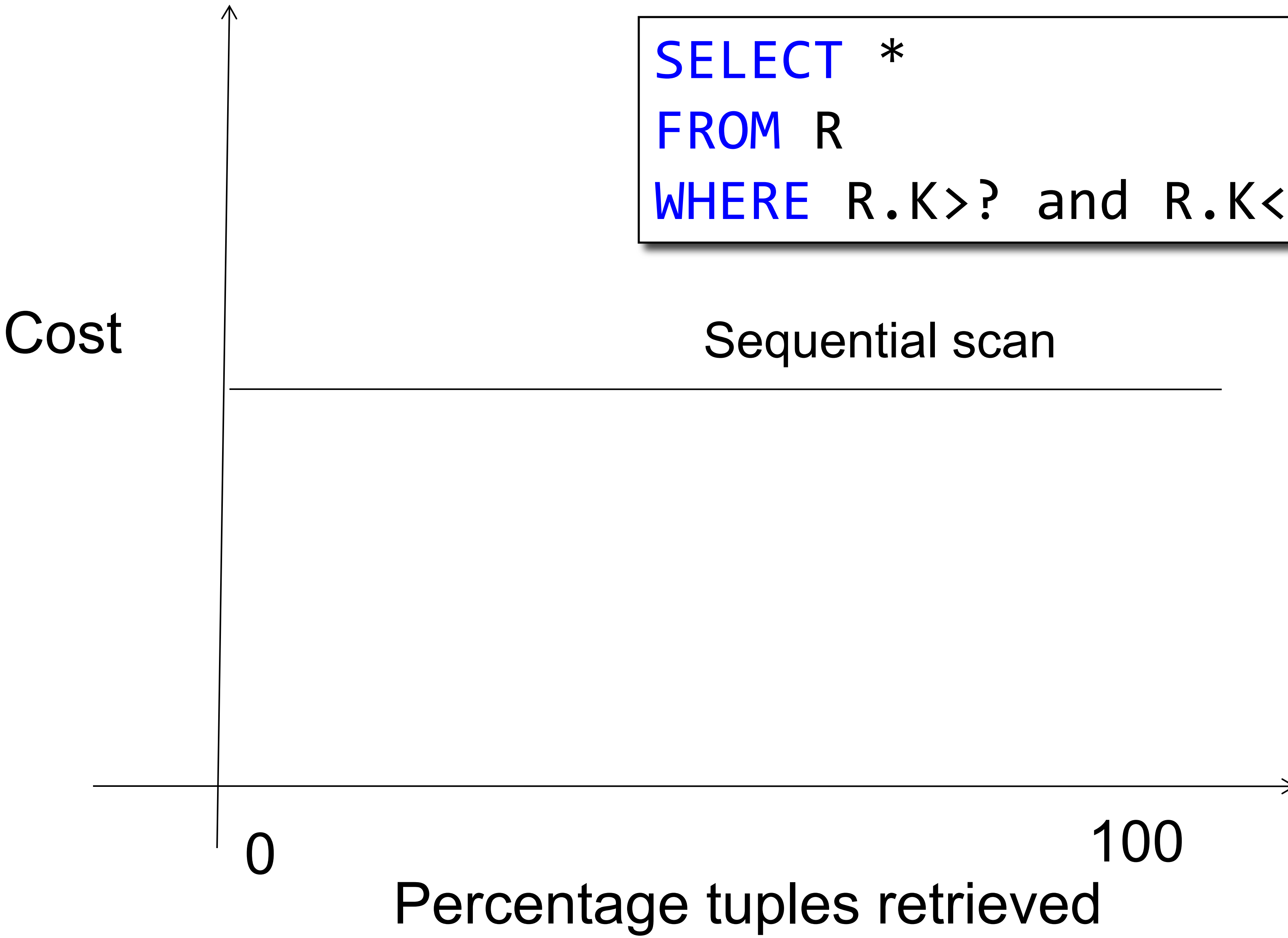
- Rule of thumb:
  Random reading 1-2% of file ≈ sequential scan entire file;

Range queries benefit mostly from clustering because they may read more than 1-2%

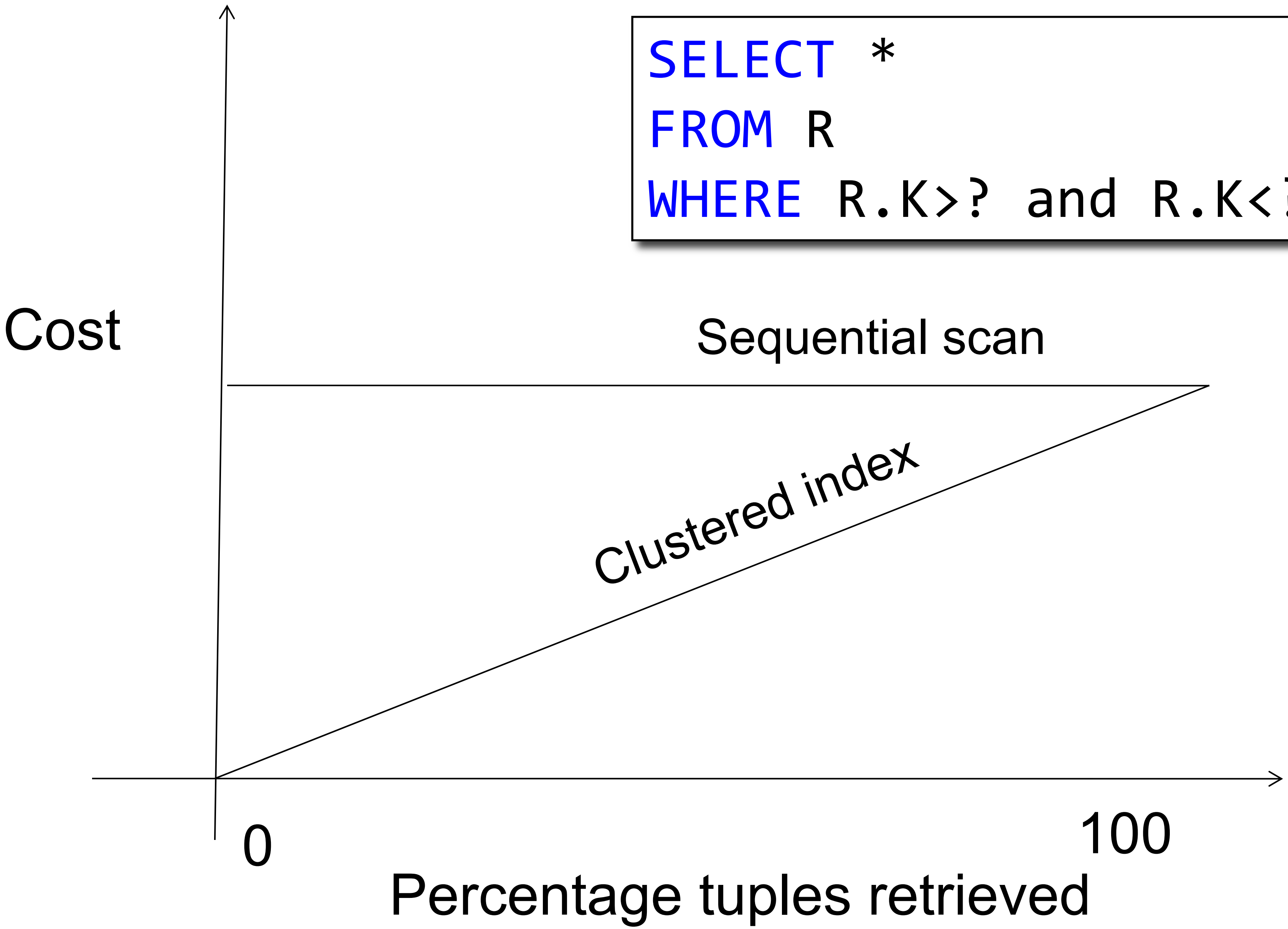
```sql
SELECT *
FROM R
WHERE R.K>? and R.K<?
```



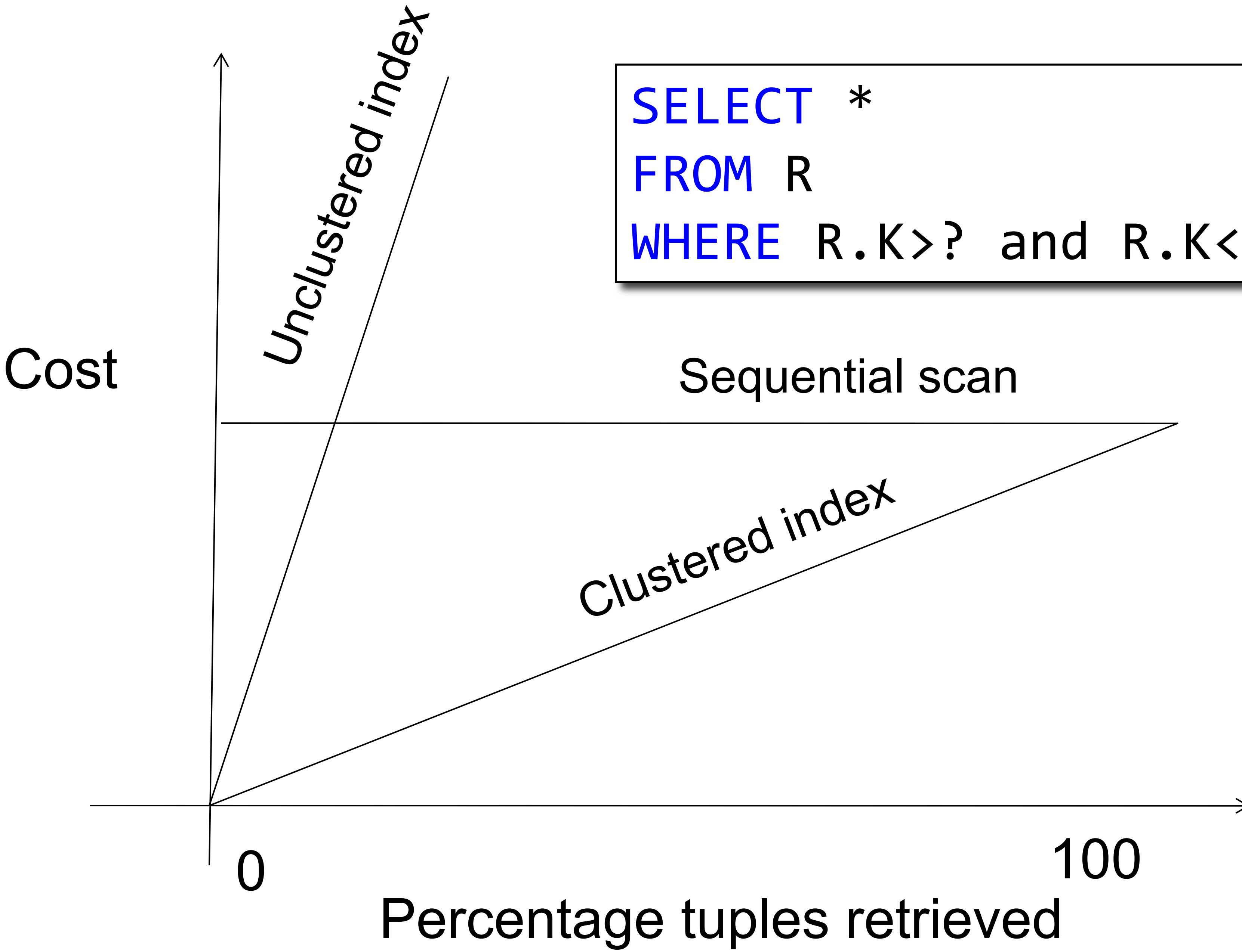
```
SELECT *
FROM R
WHERE R.K>? and R.K<?
```



```
SELECT *
FROM R
WHERE R.K>? and R.K<?
```



```
SELECT *
FROM R
WHERE R.K>? and R.K<?
```



Summary of Physical Plan

More components of a physical plan:

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

Introduction to Database Systems
CSE 344

Lecture 17:
Basics of Query Optimization and Query Cost Estimation

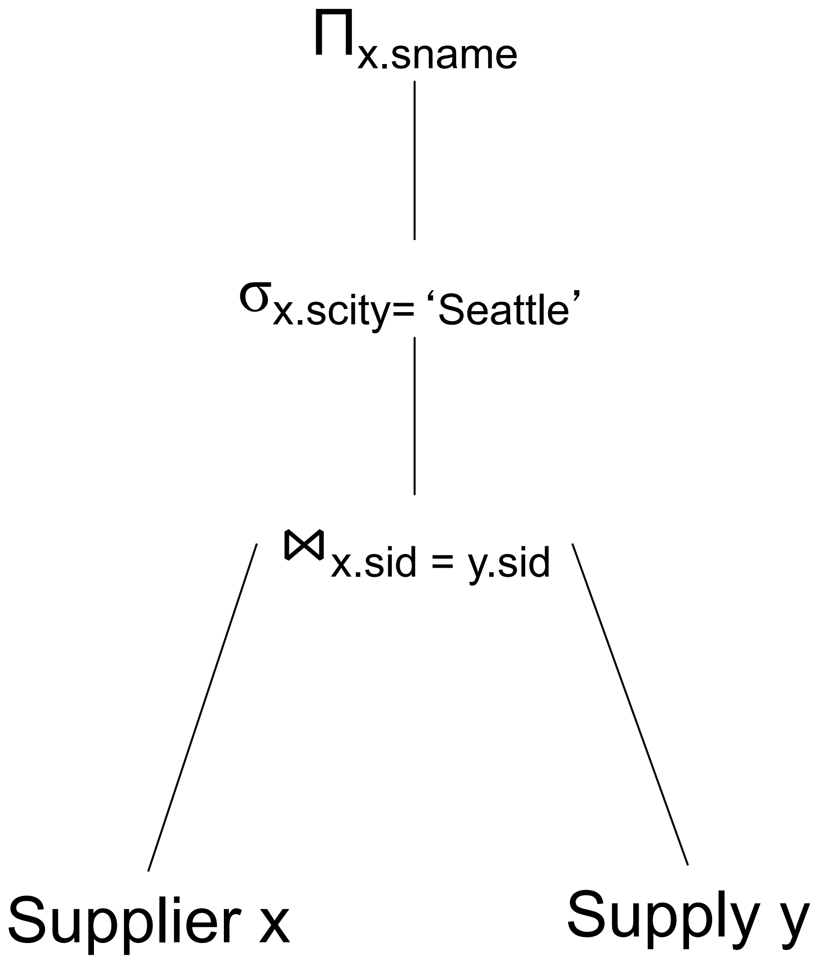
Query Optimization

- Main idea: replace a query plan with another one that is equivalent, but cheaper
- Algebraic identities of the relational algebra
- Will discuss:
  1. Pushing selections down
  2. Join reorder

Supplier(sid, sname, scity, sstate)

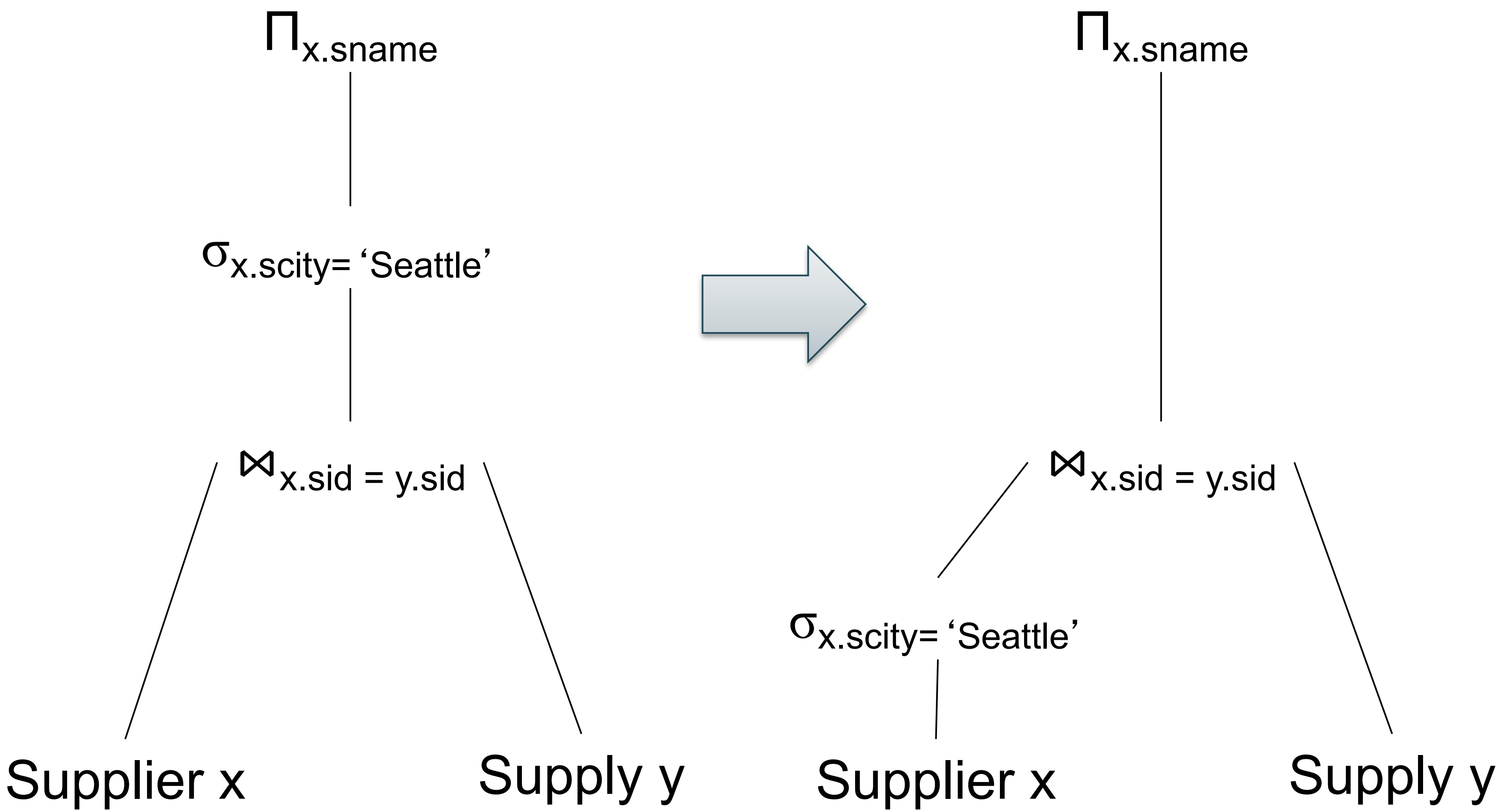
Supply(sid, pno, quantity)

Push Selections Down



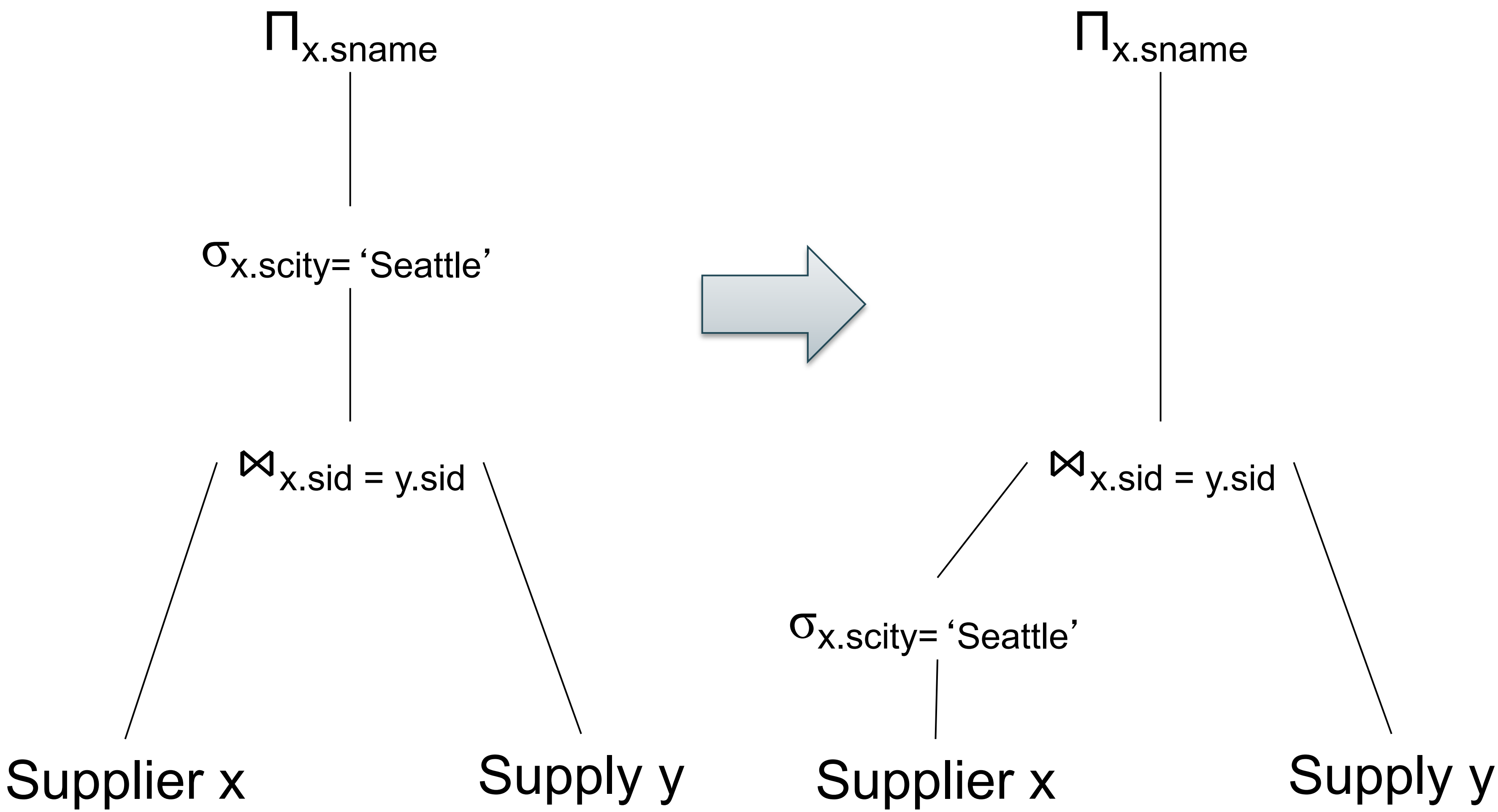
`Supplier(sid, sname, scity, sstate)`
`Supply(sid, pno, quantity)`

Push Selections Down



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

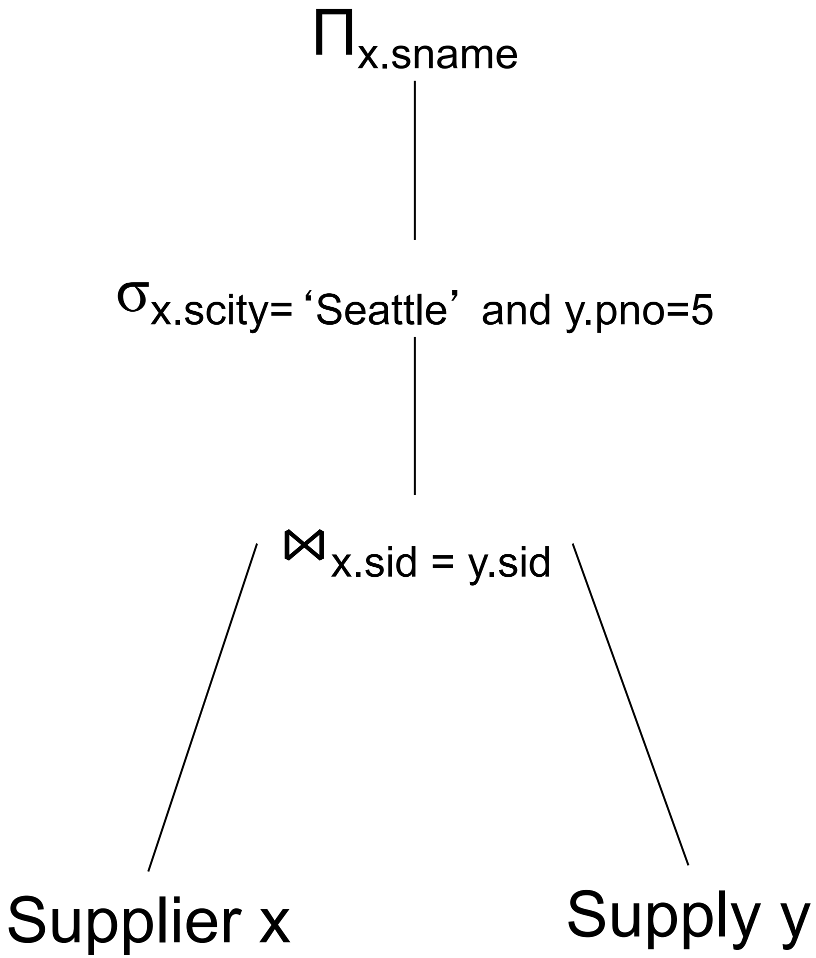
Push Selections Down



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

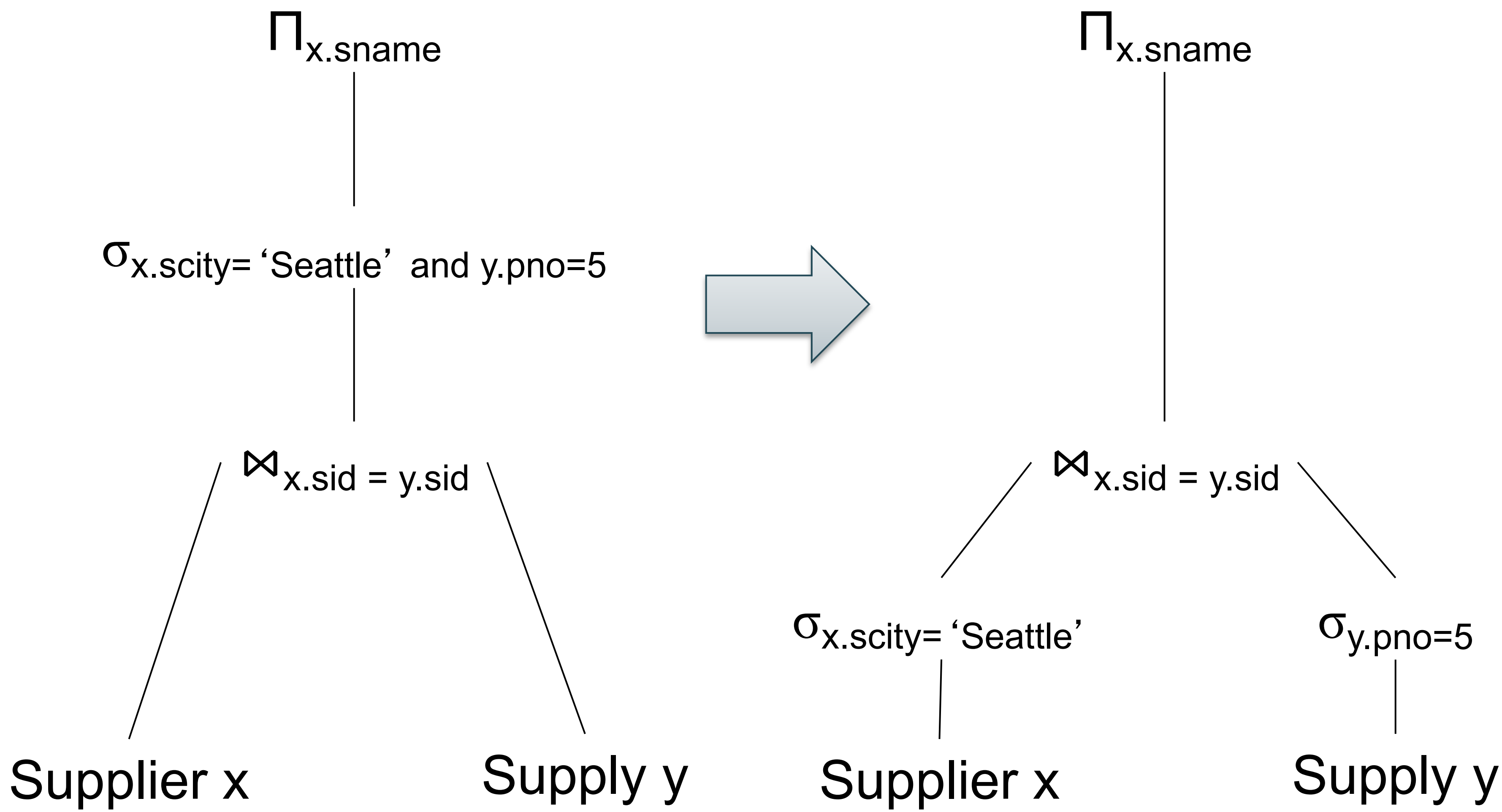
```
Supplier(sid, sname, scity, sstate)
Supply(sid, pno, quantity)
```

Push Selections Down



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

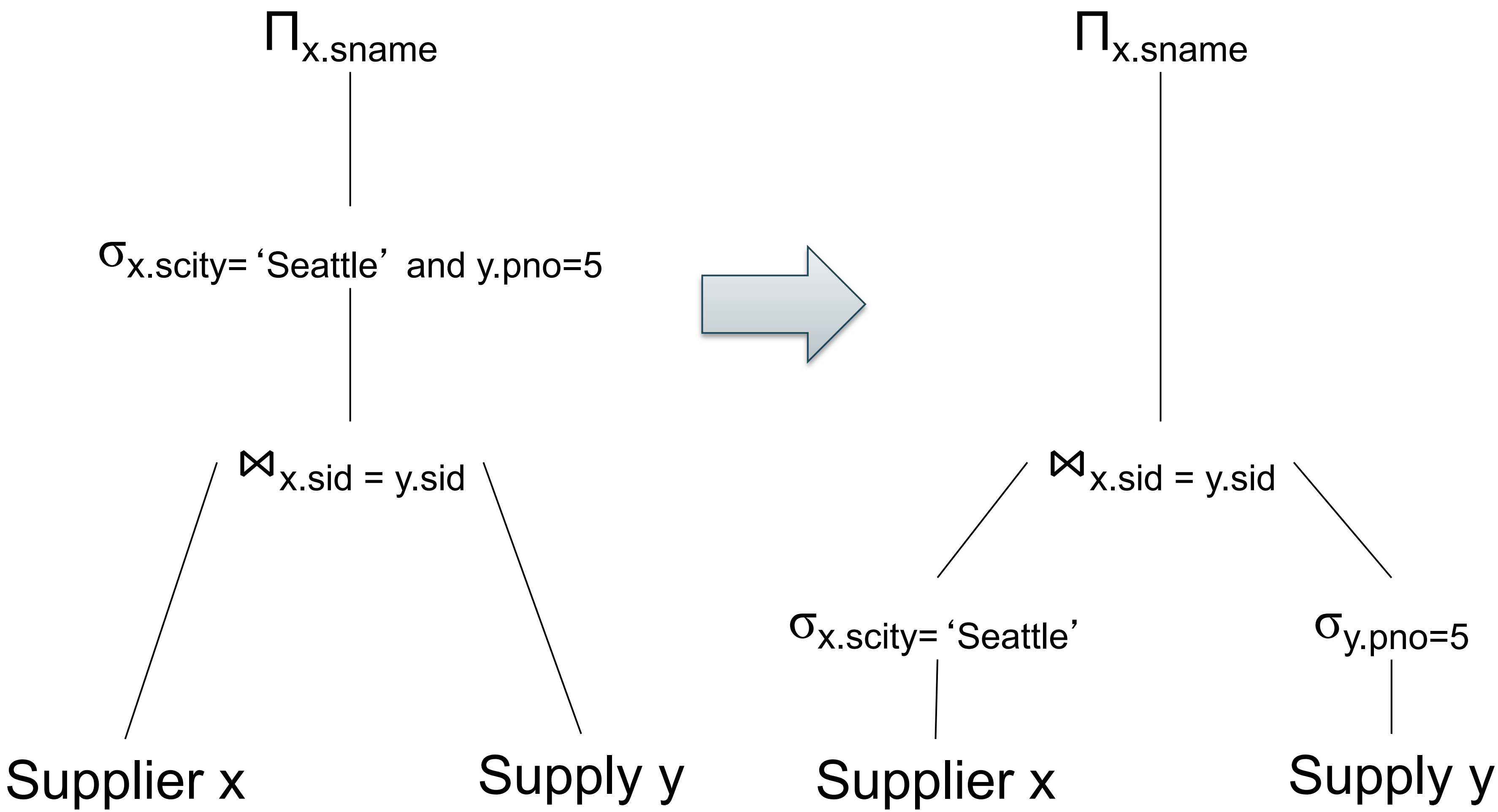
Push Selections Down



Supplier(sid, sname, scity, sstate)

Supply(sid, pno, quantity)

Push Selections Down



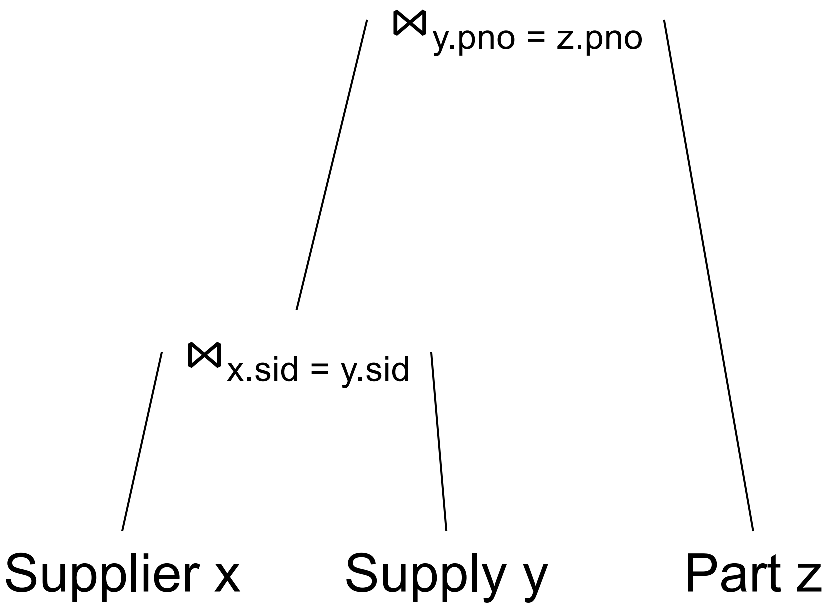
$$\sigma_{C1 \text{ and } C2}(R \bowtie S) = \sigma_{C1}(\sigma_{C2}(R \bowtie S)) = \sigma_{C1}(R \bowtie \sigma_{C2}(S)) = \sigma_{C1}(R) \bowtie \sigma_{C2}(S)$$

Supplier(sid, sname, scity, sstate)

Supply(sid, pno, quantity)

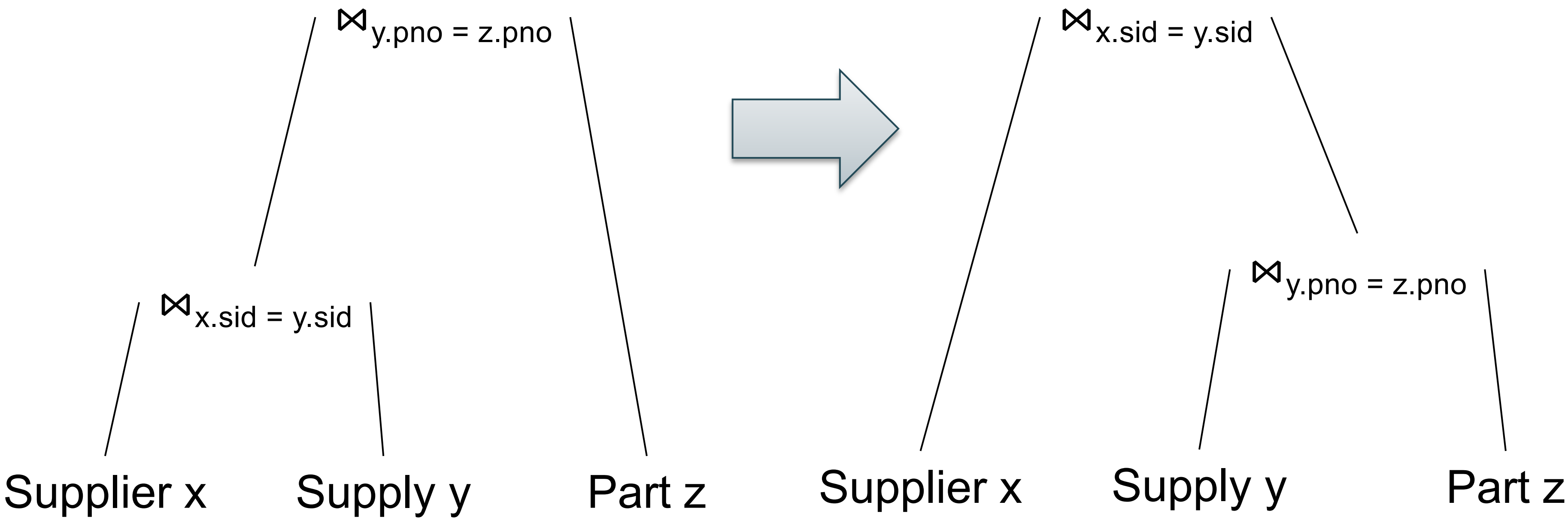
Part(pno, pname, pprice)

Join Reorder



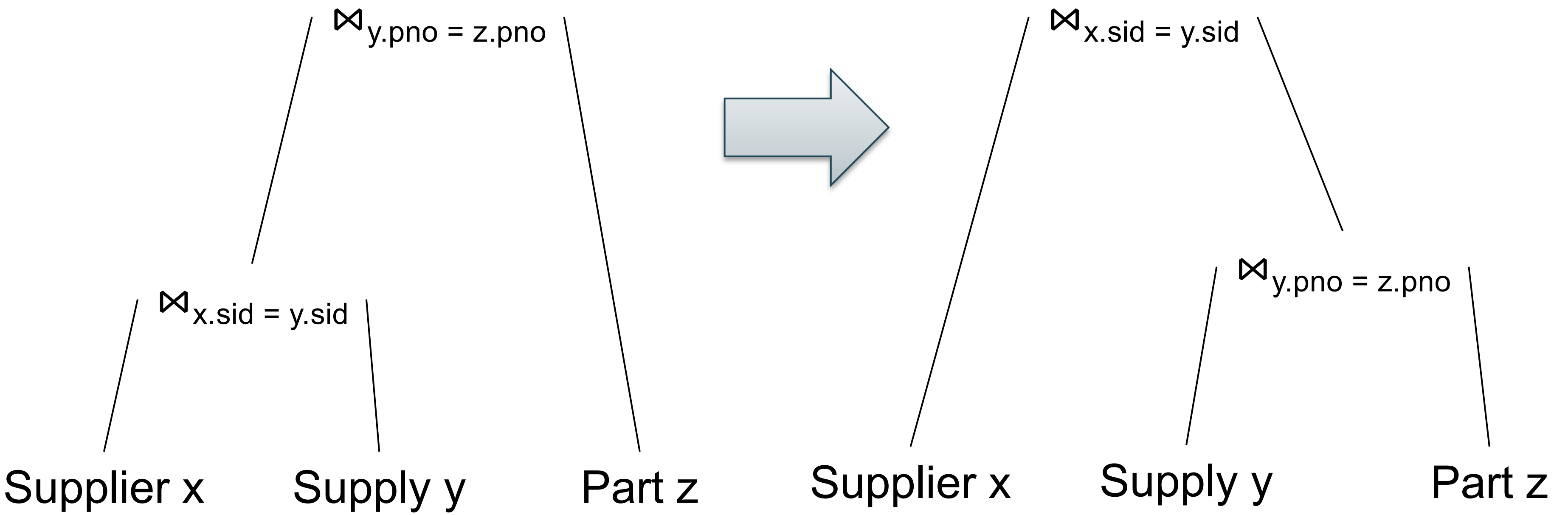
Join Reorder

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



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

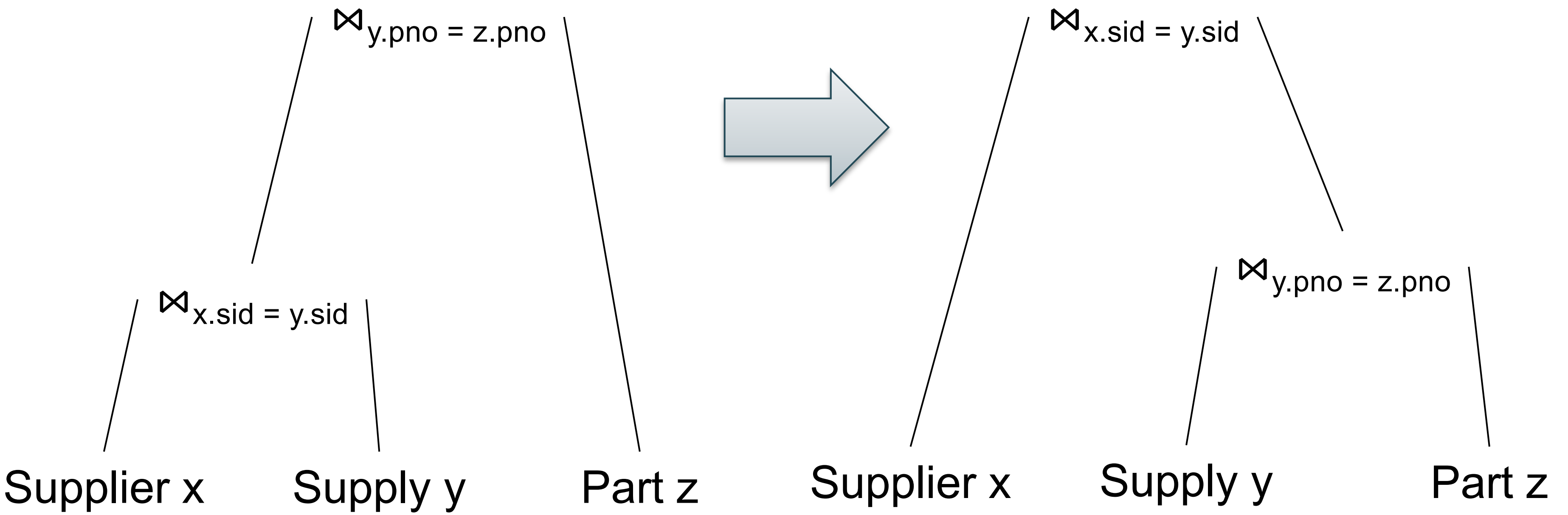
Join Reorder



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

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

Join Reorder



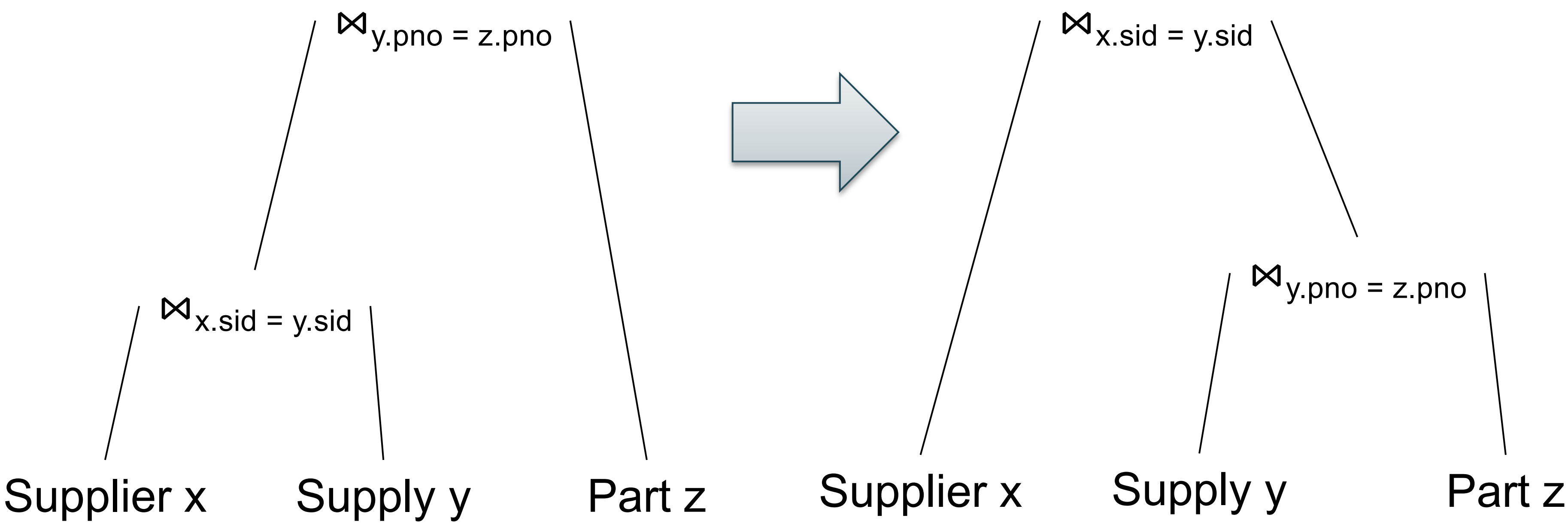
$(R \bowtie S) \bowtie T = R \bowtie (S \bowtie T)$

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

Join Reorder

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

When is one plan better than the other?

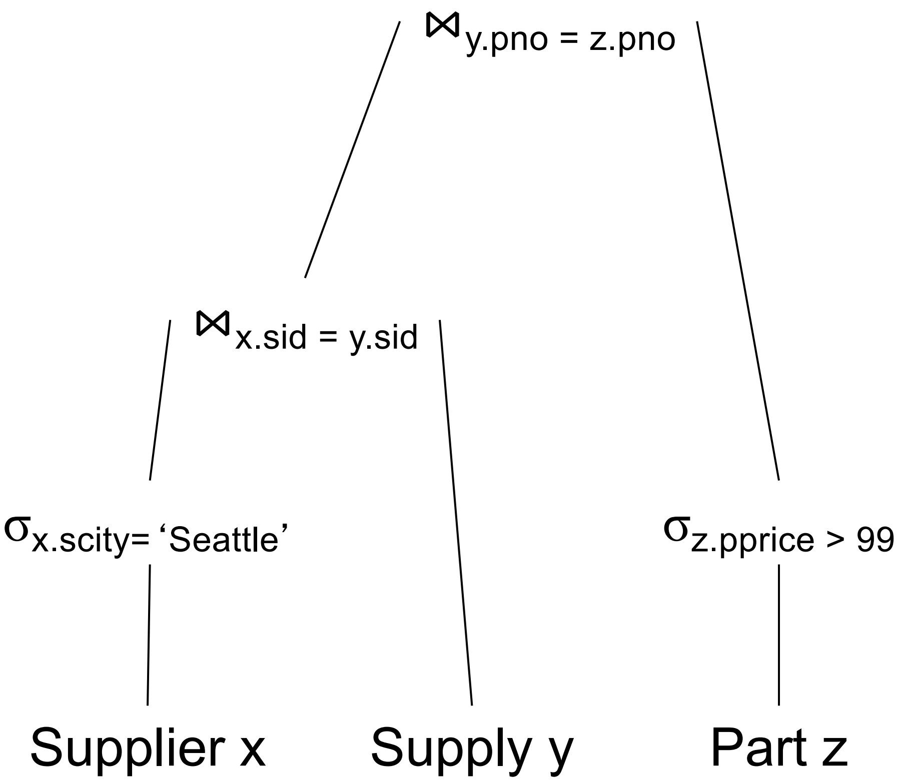


$$(R \bowtie S) \bowtie T = R \bowtie (S \bowtie T)$$

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

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

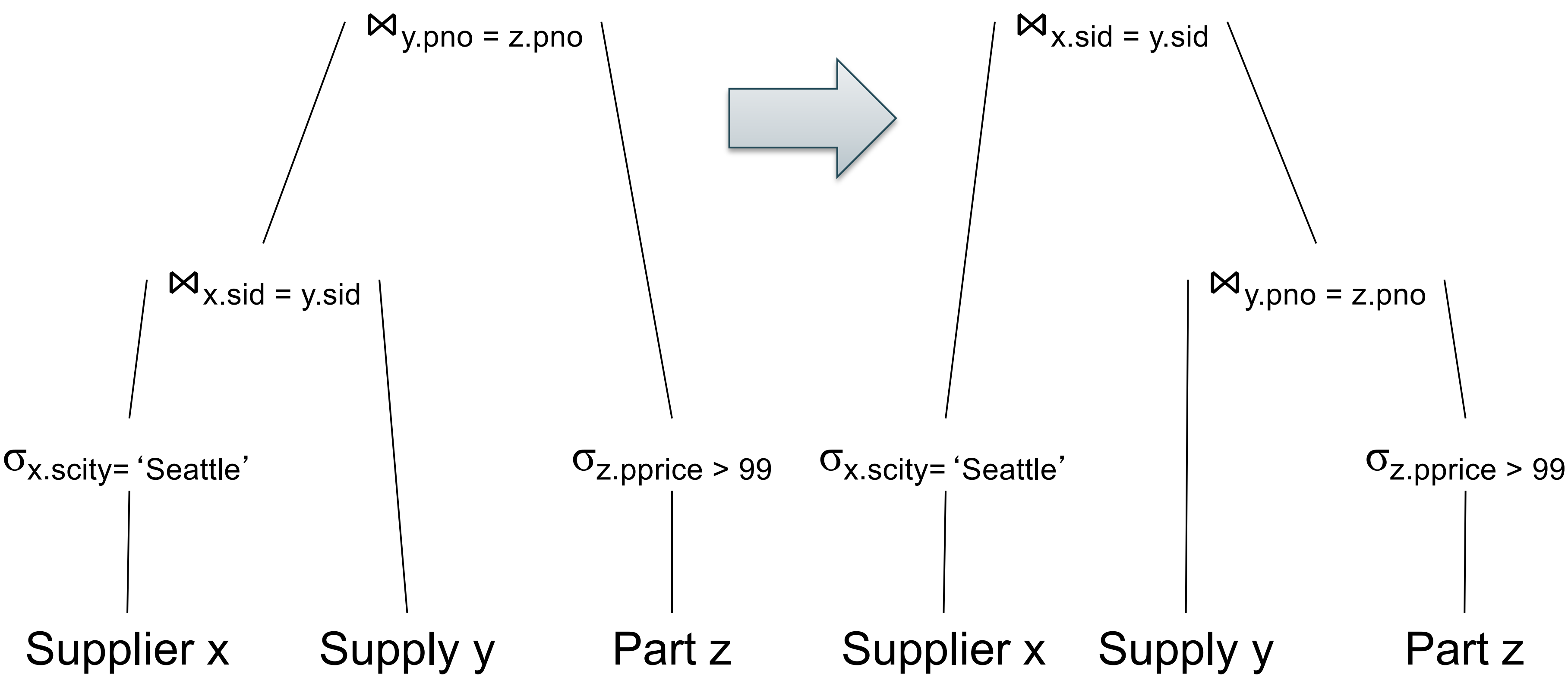
Join Reorder



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

Join Reorder

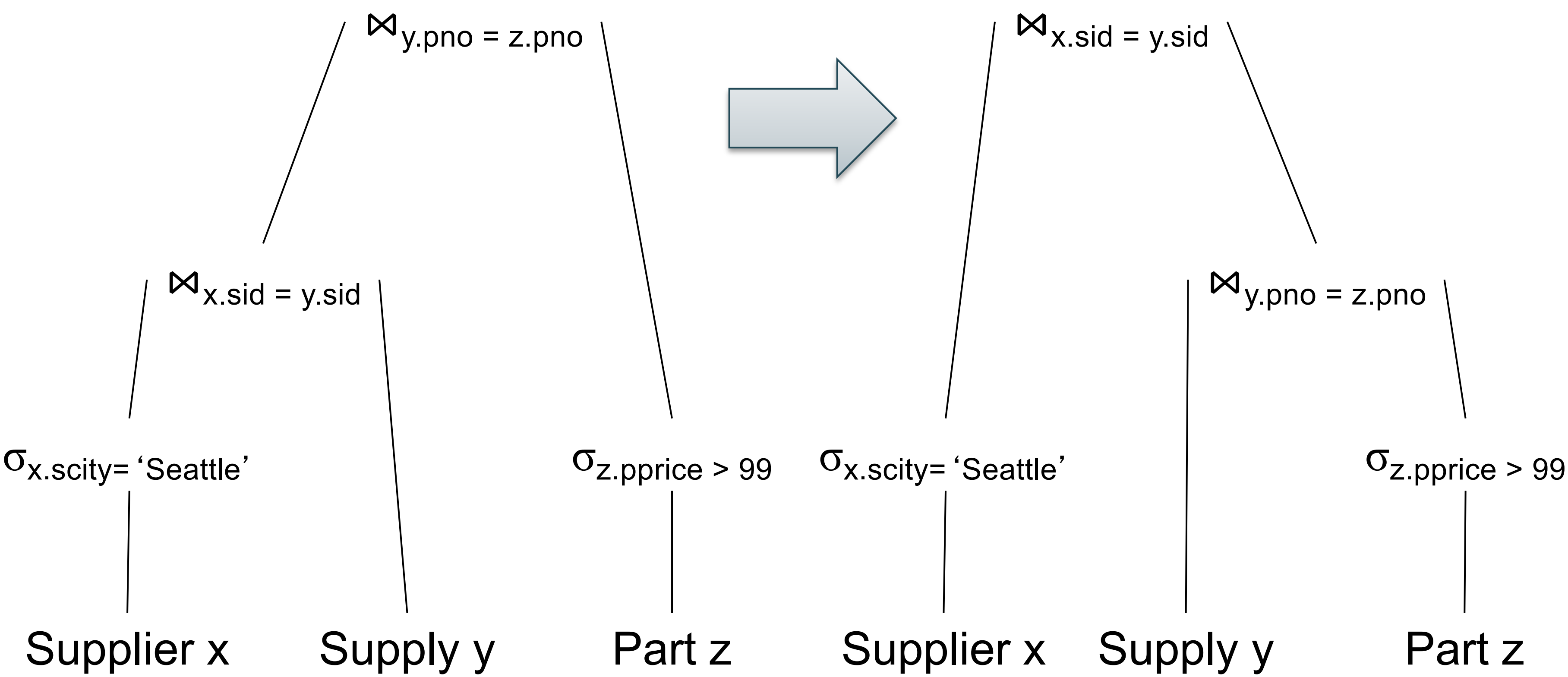
When is one plan better than the other?



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)

Size and Cost Estimation

Given statistics on the base tables:

- **B(R)** = # of blocks (i.e., pages) for relation R
- **T(R)** = # of tuples in relation R
- **V(R, A)** = # of distinct values of attribute A

Size estimation: estimate the size of a logical subplan

Cost estimation: estimate the cost of a physical subplan

Size Estimation

Problem: estimate the size of a query plan: |P|

We consider plans with selections and joins

Worst case sizes:

- Size of a selection: $|\sigma_C(R)| \leq |R|$
- Size of a join: $|R \bowtie S| \leq |R| * |S|$

Estimate ≈ f*worst-case

where f in (0,1) is called *selectivity factor*

R(A,B)
S(C,D)

Estimating Size of a Selection

Assumption 1: uniform distribution of values

- $|\sigma_{A=v}(R)| \approx |T(R)| / V(R,A)$
- Selectivity factor: $f_{A=v} = 1/V(R,A)$

Assumption 2: independence of attributes

- Selectivity factor: $f_{A=v \text{ and } B=w} = f_{A=v} * f_{B=w}$
- $|\sigma_{A=v \text{ and } B=w}(R)| \approx |T(R)| / (V(R,A) * V(R,B))$

R(A,B)
S(C,D)

Estimating Size of a Join

Assumption 3: Inclusion assumption
if $V(R,B) \leq V(S,C)$ then $\Pi_B(R) \subseteq \Pi_C(S)$

- $|R \bowtie S| \approx |R| * |S| / V(S,C)$

In general:

- $|R \bowtie S| \approx |R| * |S| / \max(V(R,B),V(S,C))$

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

Example

```
SELECT *
FROM Supplier x, Supply y
WHERE x.sid = y.sid
```

T(Supplier) = 100,000

T(Supply) = 3,000,000
V(Supply,sid) = 60,000
V(Supply,pno) = 25,000

T(Part) = 50,000

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

Example

```
SELECT *
FROM Supplier x, Supply y
WHERE x.sid = y.sid
```

T(Supplier) = 100,000

T(Supply) = 3,000,000
V(Supply,sid) = 60,000
V(Supply,pno) = 25,000

T(Part) = 50,000

|Q| = T(Supplier)*T(Supply) / max(V(Supplier,sid), V(Supply,sid))

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

Example

```
SELECT *
FROM Supplier x, Supply y
WHERE x.sid = y.sid
```

T(Supplier) = 100,000

T(Supply) = 3,000,000
V(Supply,sid) = 60,000
V(Supply,pno) = 25,000

T(Part) = 50,000

|Q| = T(Supplier)*T(Supply) / max(V(Supplier,sid), V(Supply,sid))
= 100,000 * 3,000,000 / 100,000

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

Example

```
SELECT *
FROM Supplier x, Supply y
WHERE x.sid = y.sid
```

T(Supplier) = 100,000

T(Supply) = 3,000,000
V(Supply,sid) = 60,000
V(Supply,pno) = 25,000

T(Part) = 50,000

|Q| = T(Supplier)*T(Supply) / max(V(Supplier,sid), V(Supply,sid))
= 100,000 * 3,000,000 / 100,000
= 3,000,000

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

Example

```
SELECT *
FROM Supplier x, Supply y
WHERE x.sid = y.sid
```

T(Supplier) = 100,000

T(Supply) = 3,000,000
V(Supply,sid) = 60,000
V(Supply,pno) = 25,000

T(Part) = 50,000

|Q| = T(Supplier)*T(Supply) / max(V(Supplier,sid), V(Supply,sid))
= 100,000 * 3,000,000 / 100,000
= 3,000,000

This is obvious!! Why?

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

Example

```
SELECT *
FROM Supplier x, Supply y, Part z
WHERE x.sid = y.sid and y.pno = z.pno
    and x.scity = 'Seattle'
    and x.sstate ='WA'
    and z.price = 30
```

T(Supplier) = 100,000
V(Supplier,city) = 2000
V(Supplier,state) = 50

T(Supply) = 3,000,000
V(Supply,sid) = 60,000
V(Supply,pno) = 25,000

T(Part) = 50,000
V(Part,price) = 5000

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

Example

```
SELECT *
FROM Supplier x, Supply y, Part z
WHERE x.sid = y.sid and y.pno = z.pno
    and x.scity = 'Seattle'
    and x.sstate ='WA'
    and z.price = 30
```

T(Supplier) = 100,000
V(Supplier,city) = 2000
V(Supplier,state) = 50

T(Supply) = 3,000,000
V(Supply,sid) = 60,000
V(Supply,pno) = 25,000

T(Part) = 50,000
V(Part,price) = 5000

Q = T(Supply) / V(Supplier,city)*V(Supplier,state),V(Part,price)

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

Example

SELECT *
FROM Supplier x, Supply y, Part z
WHERE x.sid = y.sid and y.pno = z.pno
and x.scity = 'Seattle'
and x.sstate ='WA'
and z.price = 30

T(Supplier) = 100,000
V(Supplier,city) = 2000
V(Supplier,state) = 50

T(Supply) = 3,000,000
V(Supply,sid) = 60,000
V(Supply,pno) = 25,000

T(Part) = 50,000
V(Part,price) = 5000

Q = T(Supply) / V(Supplier,city)*V(Supplier,state),V(Part,price)
= 3,000,000 / (2000 * 50 * 5000) < 1

Optimization

- The optimizer considers several plans
- For each plan, it estimates costs
- Then chooses the cheapest plan

Cost estimation: we will consider only the I/O cost.

I/O Cost of Physical Operators

Cost Parameters

Given statistics on the base tables:

- **B(R)** = # of blocks (i.e., pages) for relation R
- **T(R)** = # of tuples in relation R
- **V(R, A)** = # of distinct values of attribute A

I/O Cost of Selection

- Sequential scan for relation R costs **B(R)**
- Index-based selection
  - Estimate selectivity factor **f**
  - Clustered index: f***B(R)**
  - Unclustered index f***T(R)**

Note: we ignore I/O cost for index pages

Example

$B(R) = 2000$
$T(R) = 100{,}000$
$V(R, A) = 20$

cost of $\sigma_{A=v}(R) = ?$

- Table scan:
- Index based selection:

Example

B(R) = 2000
T(R) = 100,000
V(R, A) = 20

cost of $\sigma_{A=v}(R)$ = ?

- Table scan: B(R) = 2,000 I/Os
- Index based selection:

Example

B(R) = 2000
T(R) = 100,000
V(R, A) = 20

cost of $\sigma_{A=v}(R)$ = ?

- Table scan: B(R) = 2,000 I/Os
- Index based selection:
  - If index is unclustered: T(R) * 1/V(R,A) = 5,000 I/Os

Example

$B(R) = 2000$
$T(R) = 100{,}000$
$V(R, A) = 20$

cost of $\sigma_{A=v}(R)$ = ?

- Table scan: $B(R)$ = 2,000 I/Os
- Index based selection:
  - If index is unclustered: $T(R) * 1/V(R,A)$ = 5,000 I/Os
  - If index is clustered: $B(R) * 1/V(R,A)$ = 100 I/Os

Example

B(R) = 2000
T(R) = 100,000
V(R, A) = 20

cost of $\sigma_{A=v}(R)$ = ?

- Table scan: B(R) = 2,000 I/Os
- Index based selection:
  - If index is unclustered: T(R) * 1/V(R,A) = 5,000 I/Os
  - If index is clustered: B(R) * 1/V(R,A) = 100 I/Os

Lesson: Don't build unclustered indexes when V(R,A) is small !

NOT COVERED

CSE 414, Spring 2019:

- We will not cover the I/O cost of a join

- **Skip slides until “Cost of a query plan”**
- Study the size estimate of the logical plan.

I/O Cost of a Join

- Nested loop join
- Hash join
- Sort-merge join
- Index-join

Read: sections 15.2, 15.3, 15.6

Join Example

Patient(pid, name, address)

Insurance(pid, provider, policy_nb)

Patient $\bowtie$ Insurance

Two tuples per page

Patient

1	'Bob'	'Seattle'
2	'Ela'	'Everett'

3	'Jill'	'Kent'
4	'Joe'	'Seattle'

Insurance

2	'Blue'	123
4	'Prem'	432

4	'Prem'	343
3	'GrpH'	554

Nested Loop Joins

- Tuple-based nested loop R ⋈ S
- R is the outer relation, S is the inner relation

```
for each tuple t1 in R do
   for each tuple t2 in S do
      if t1 and t2 join then output (t1,t2)
```

What is the Cost?

Nested Loop Joins

- Tuple-based nested loop R ⋈ S
- R is the outer relation, S is the inner relation

```
for each tuple t1 in R do
   for each tuple t2 in S do
      if t1 and t2 join then output (t1,t2)
```

What is the Cost?

- Cost: $B(R) + T(R)\,B(S)$
- Multiple-pass since S is read many times

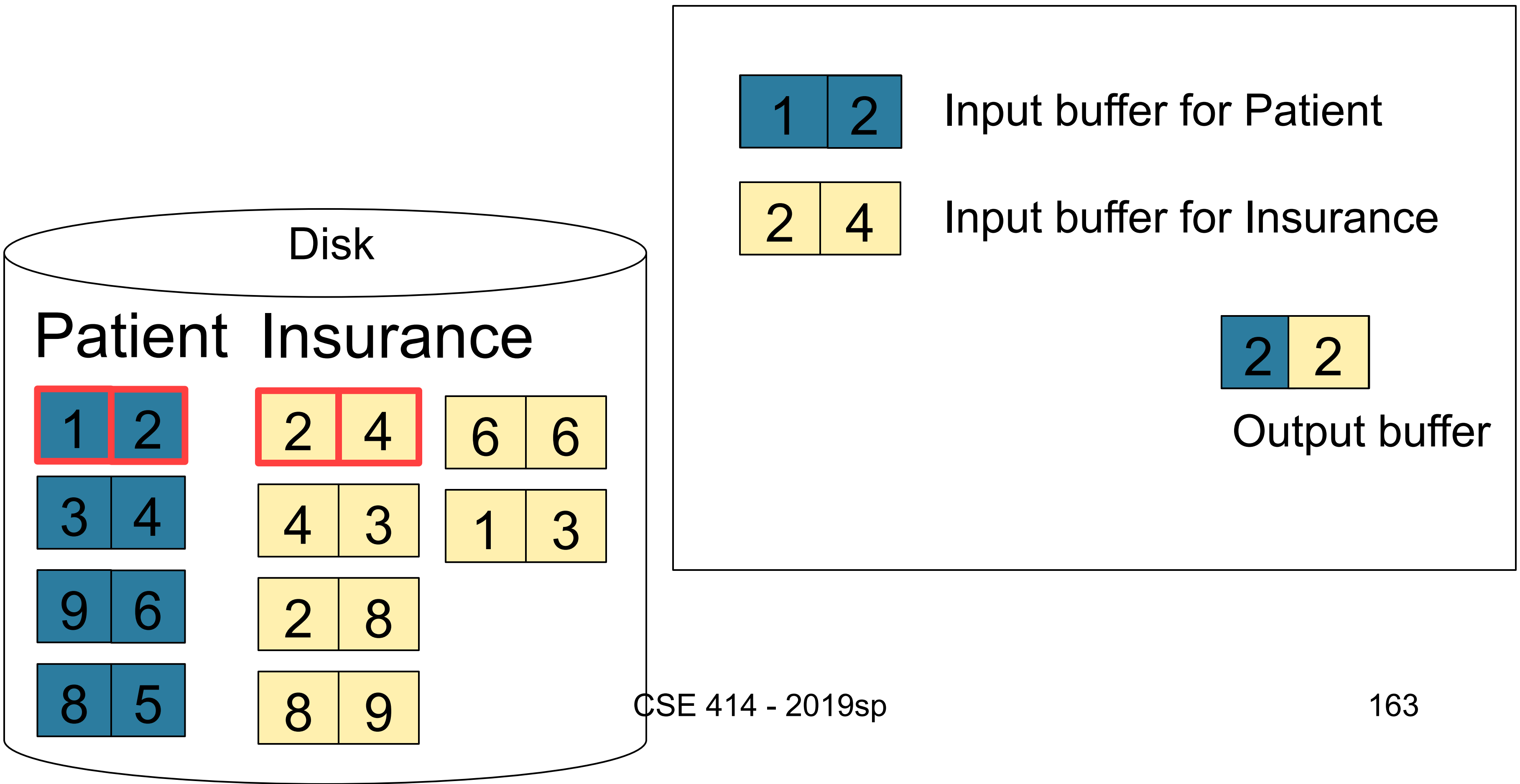
Page-at-a-time Refinement

```
for each page of tuples r in R do
   for each page of tuples s in S do
         for all pairs of tuples t1 in r, t2 in s
               if t1 and t2 join then output (t1,t2)
```

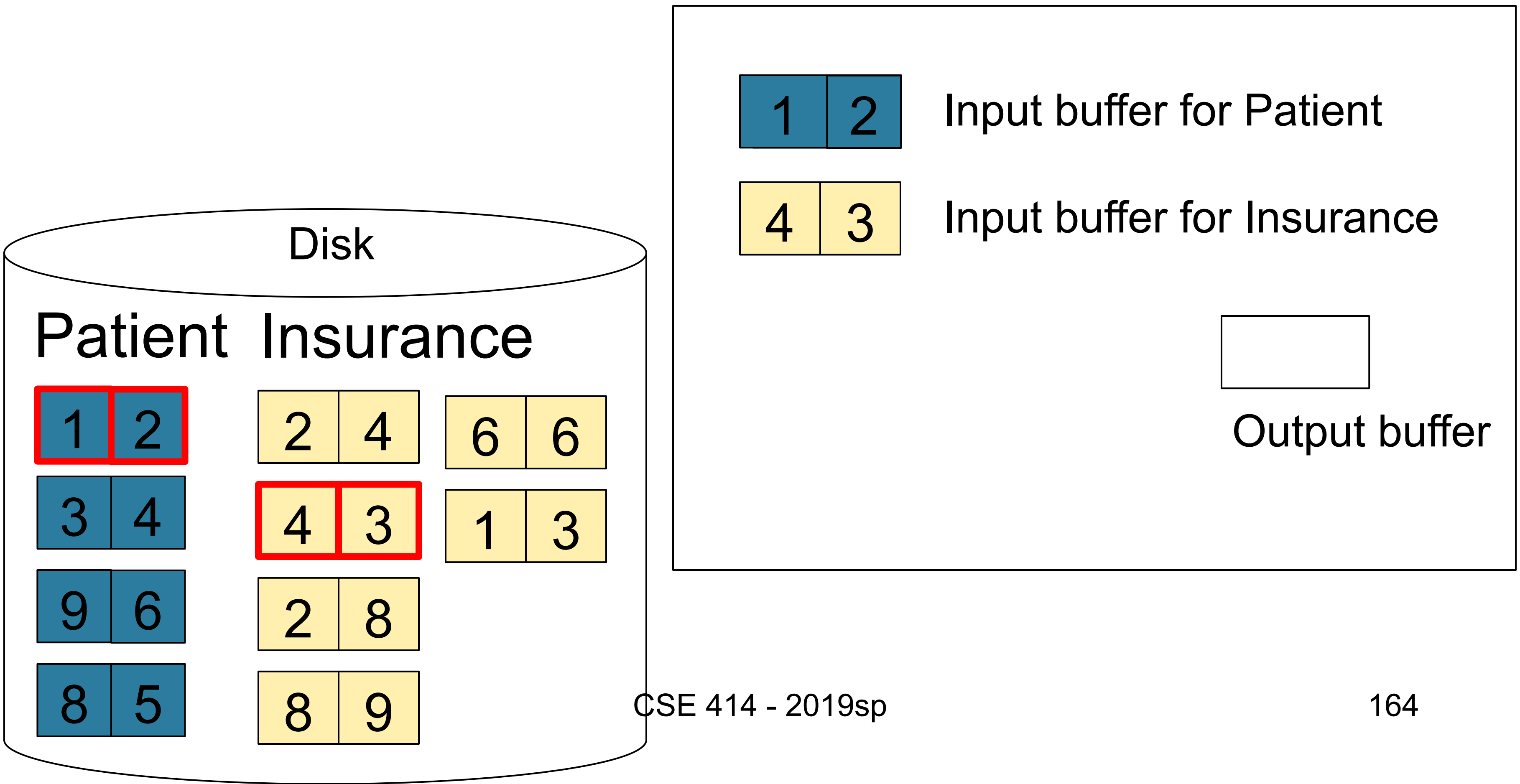
- Cost: B(R) + B(R)B(S)

What is the Cost?

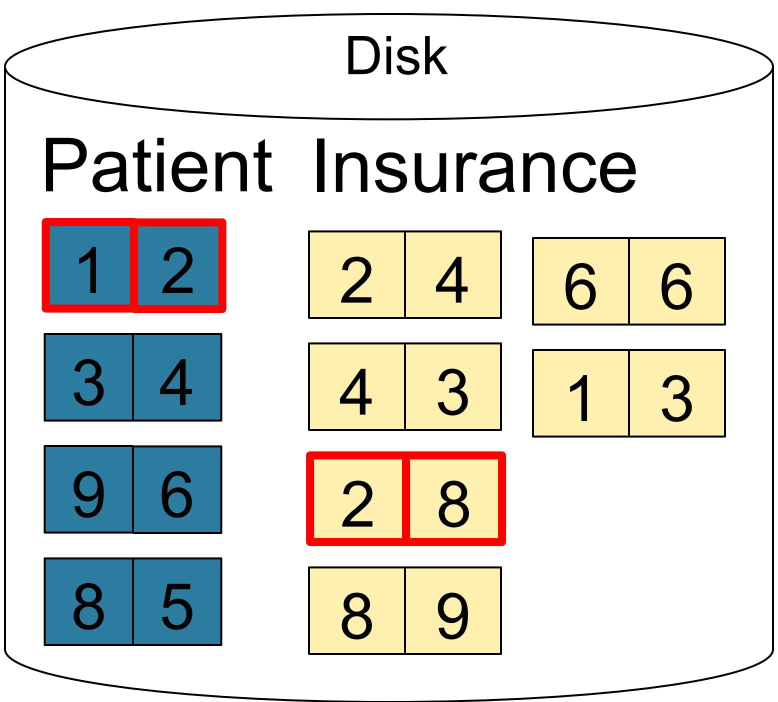
Page-at-a-time Refinement

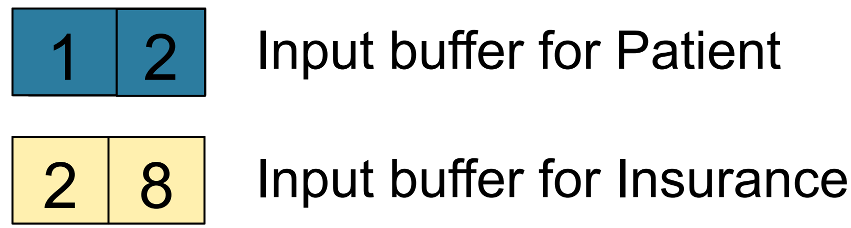


Page-at-a-time Refinement

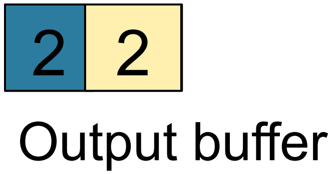


Page-at-a-time Refinement





Keep going until read all of Insurance



Then repeat for next page of Patient… until end of Patient

Cost: B(R) + B(R)B(S)

Block-Nested-Loop Refinement

```
for each group of M-1 pages r in R do
   for each page of tuples s in S do
           for all pairs of tuples t1 in r, t2 in s
                   if t1 and t2 join then output (t1,t2)
```

- Cost: $B(R) + B(R)B(S)/(M-1)$

What is the Cost?

Hash Join

Hash join: R ⋈ S

- Scan R, build buckets in main memory
- Then scan S and join
- Cost: B(R) + B(S)
- Which relation to build the hash table on?

Hash Join

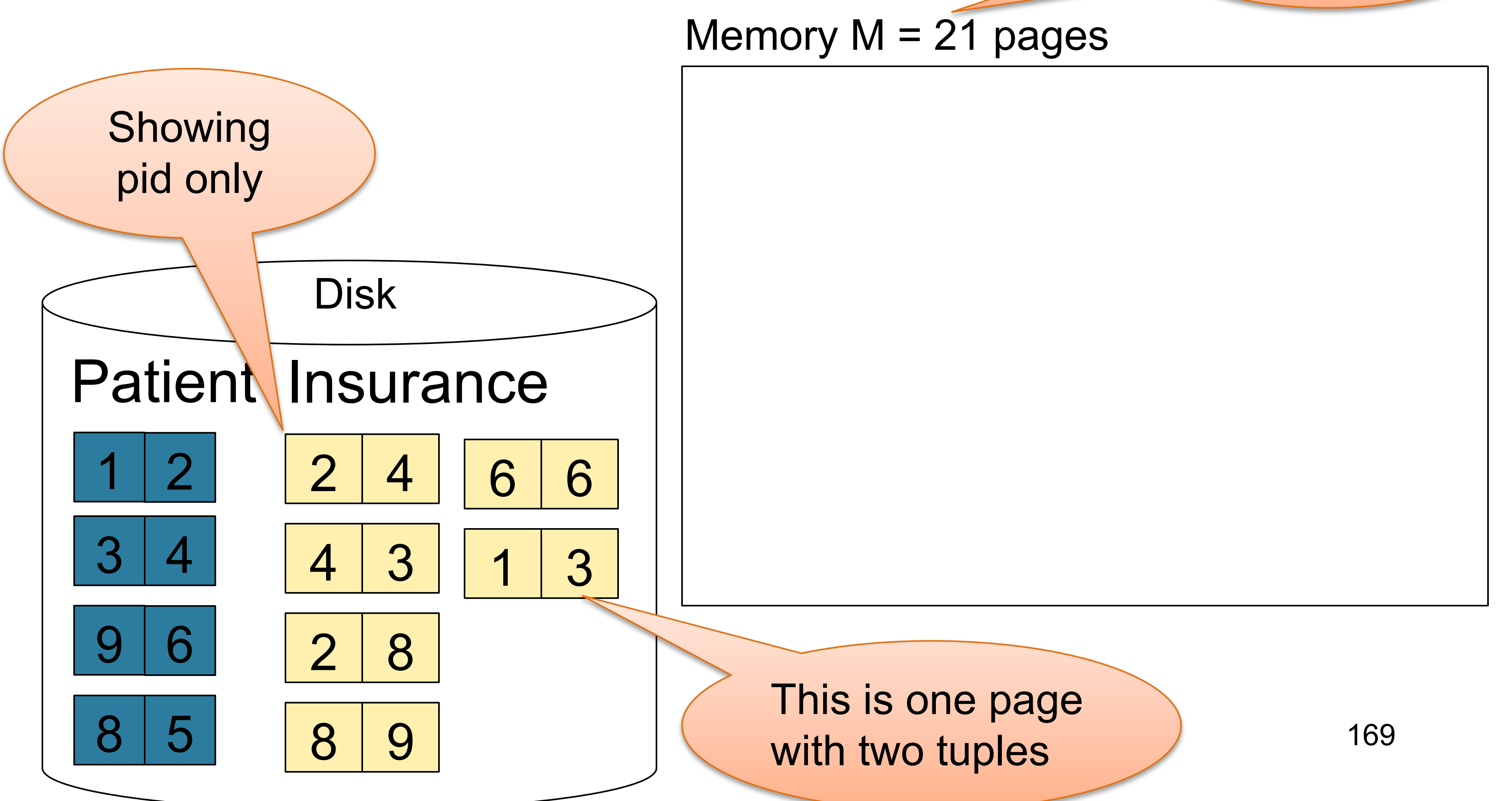
Hash join: $R \bowtie S$

- Scan R, build buckets in main memory
- Then scan S and join
- Cost: B(R) + B(S)
- Which relation to build the hash table on?

- One-pass algorithm when B(R) ≤ M
  - M = number of memory pages available

Hash Join Example

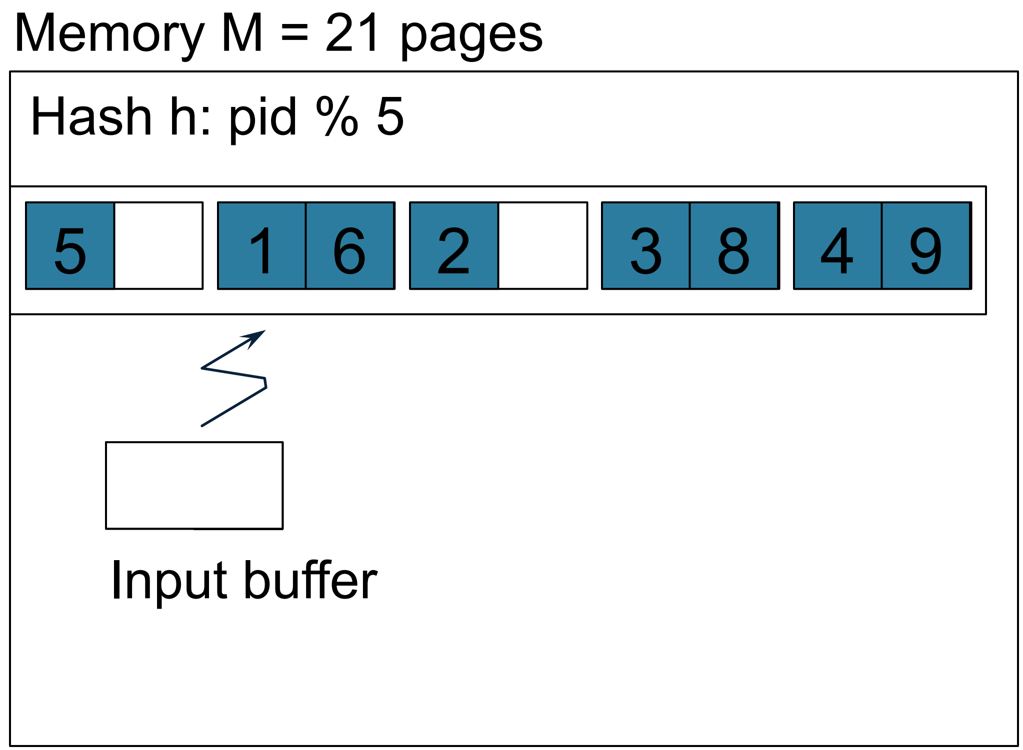
Patient ⋈ Insurance

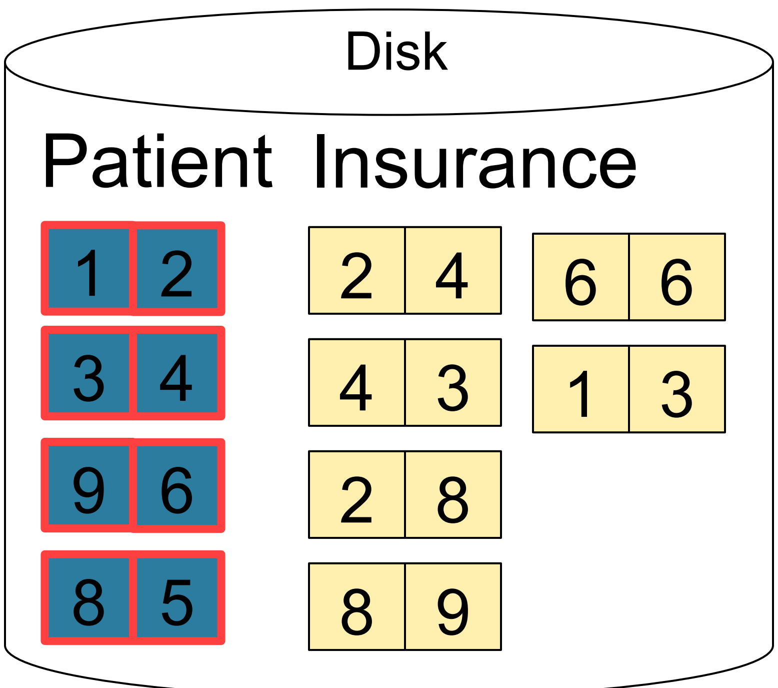


Hash Join Example

Step 1: Scan Patient and build hash table in memory

Can be done in method open()



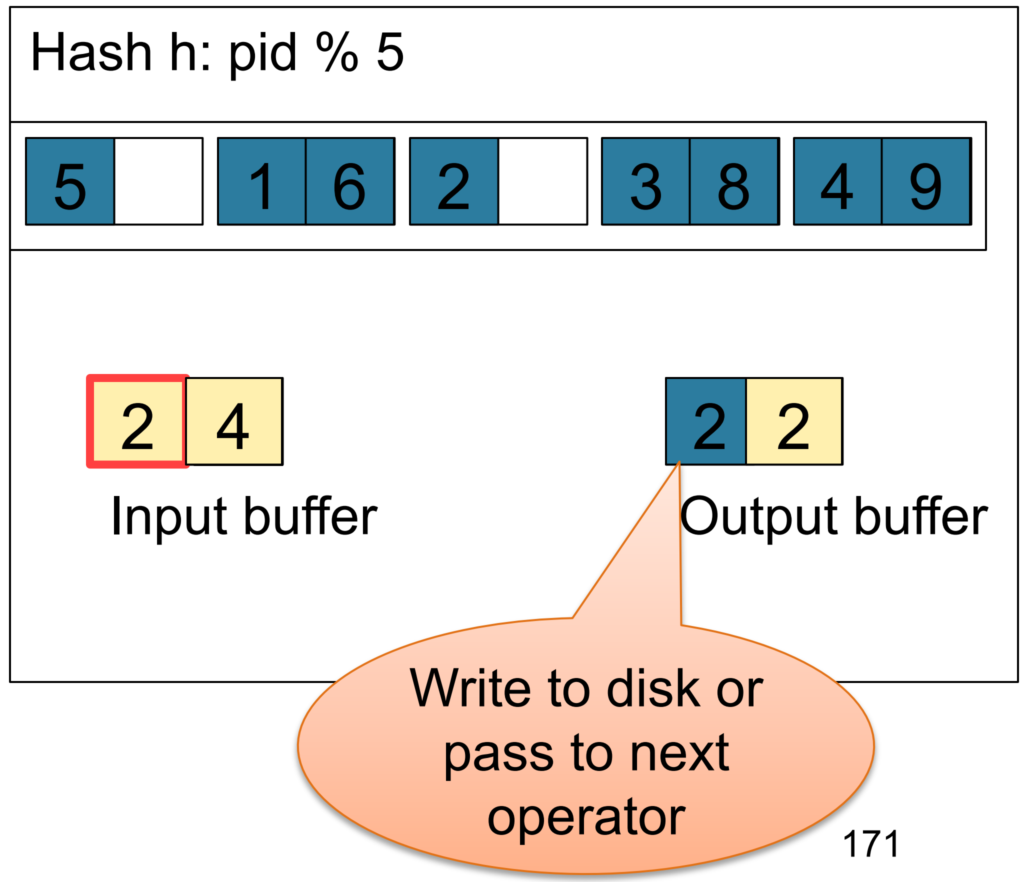


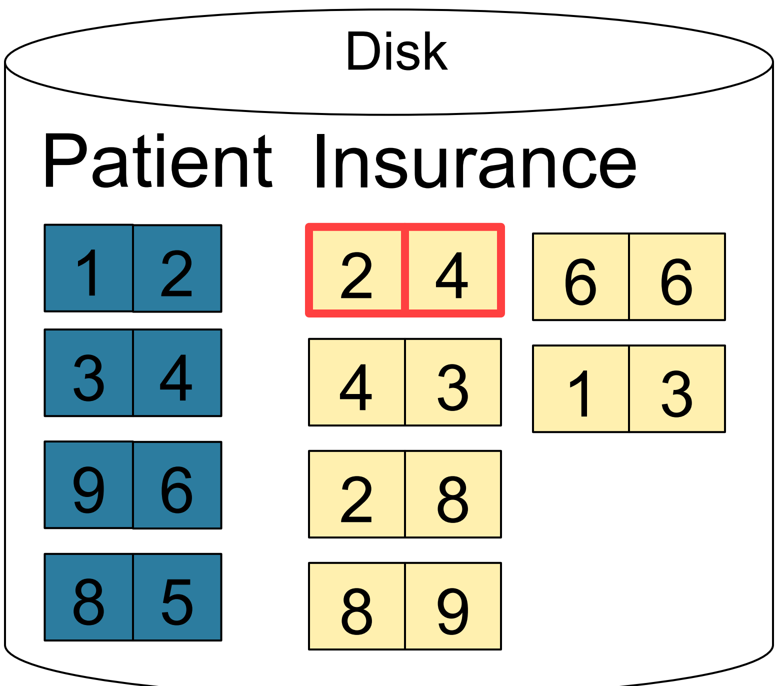
Hash Join Example

Step 2: Scan Insurance and probe into hash table

Done during
calls to next()

Memory M = 21 pages

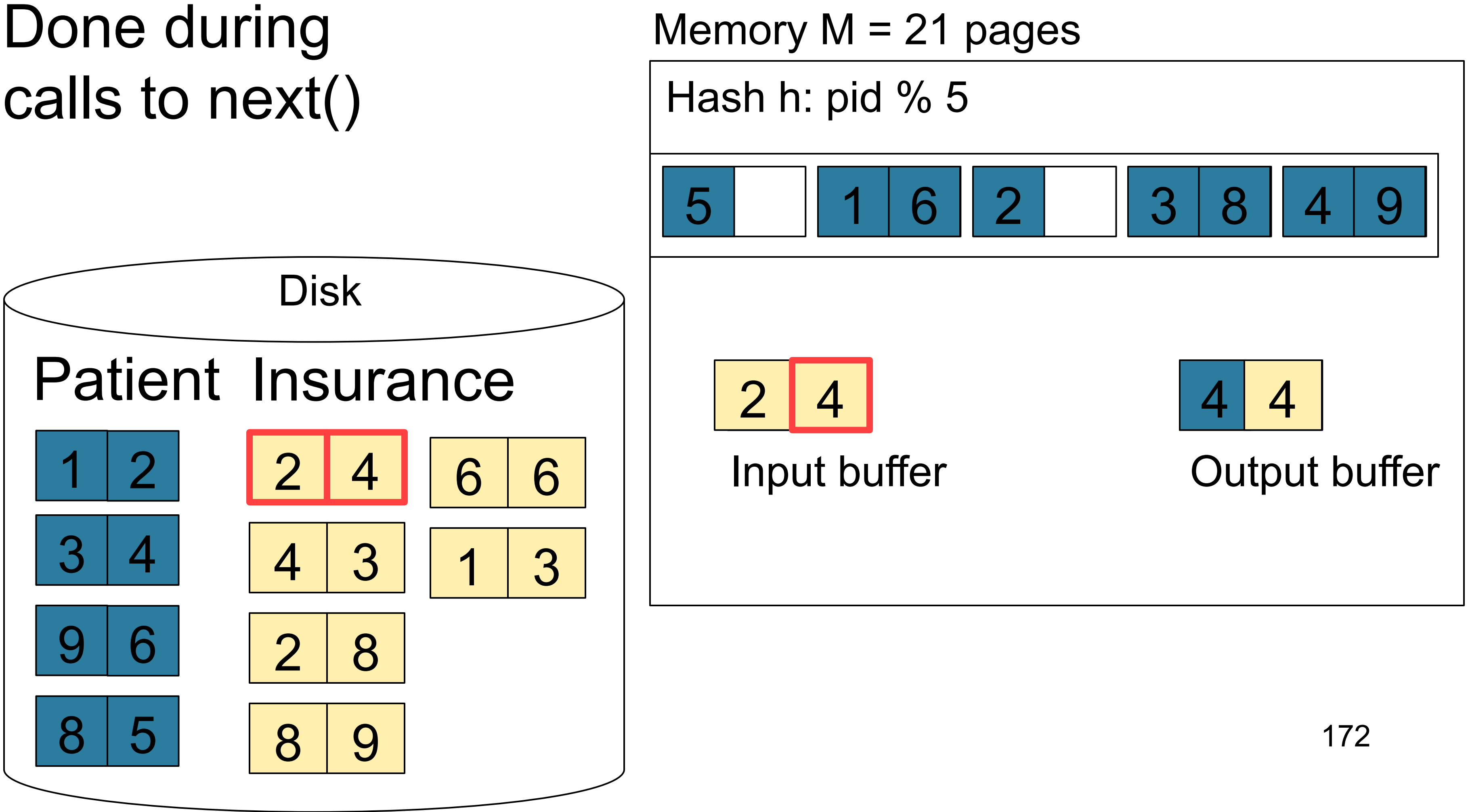




Hash Join Example

Step 2: Scan Insurance and probe into hash table

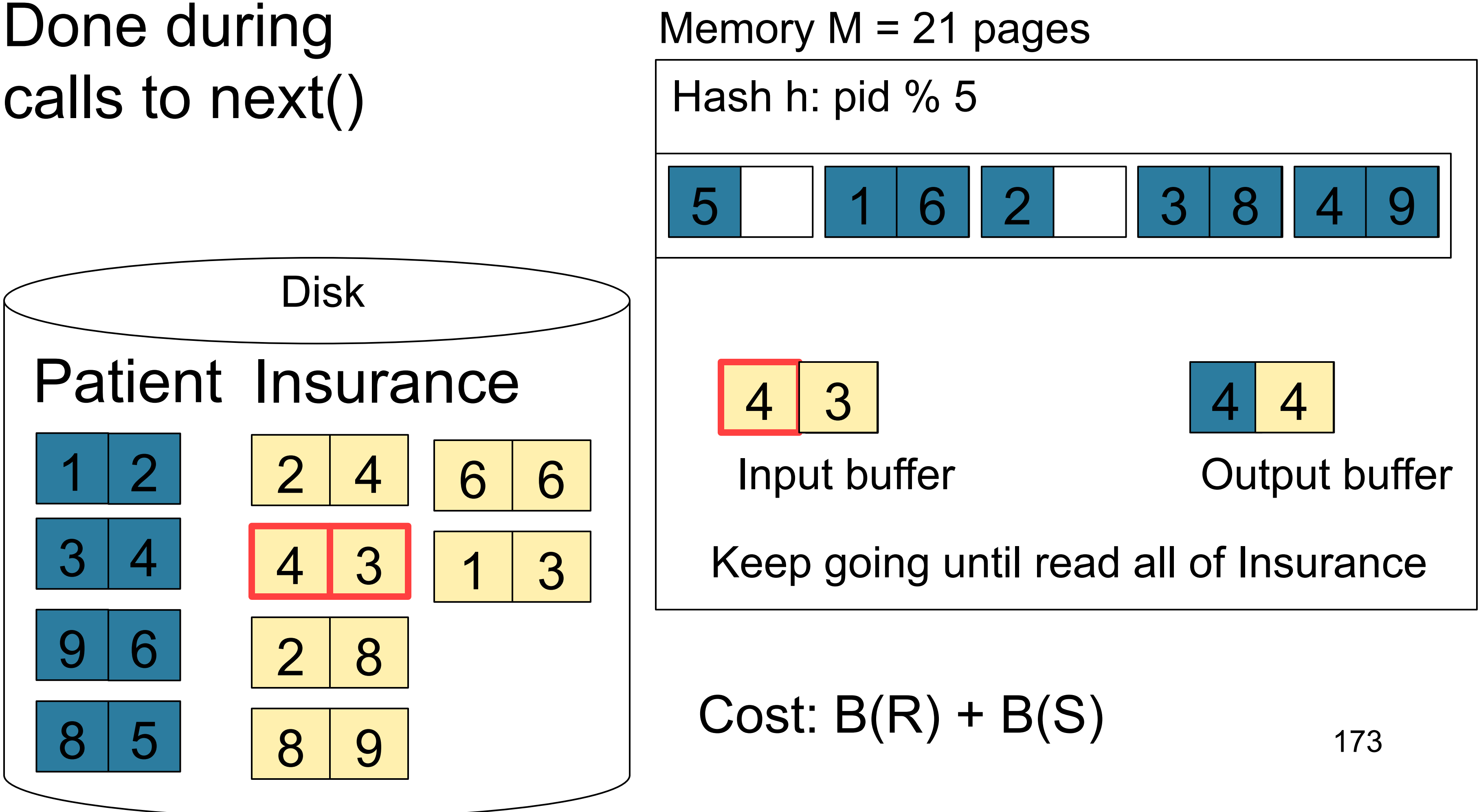
Done during calls to next()



Hash Join Example

Step 2: Scan Insurance and probe into hash table

Done during calls to next()



Cost: B(R) + B(S)

Sort-Merge Join

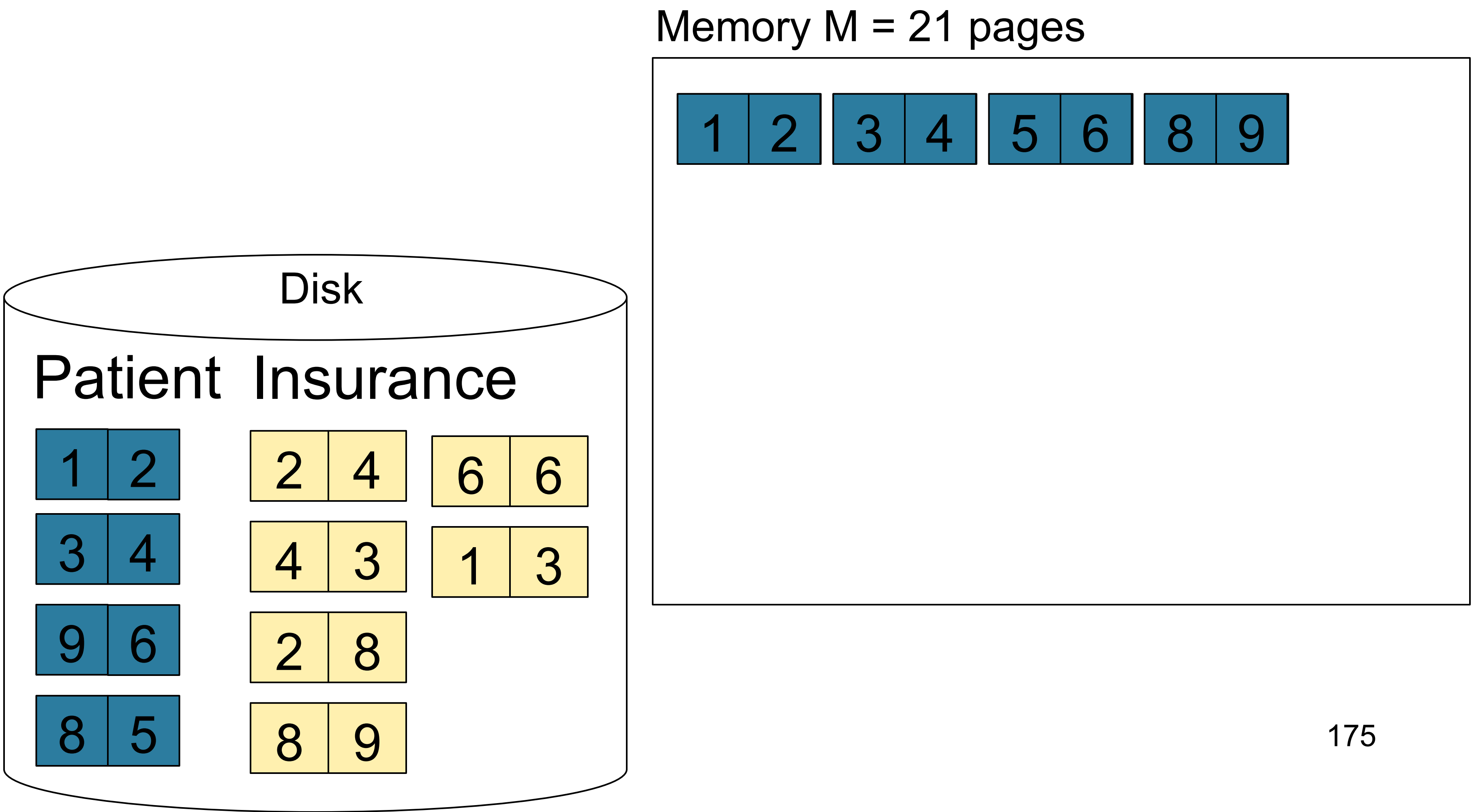
Sort-merge join: $R \bowtie S$

- Scan R and sort in main memory
- Scan S and sort in main memory
- Merge R and S

- Cost: B(R) + B(S)
- One pass algorithm when B(S) + B(R) <= M
- Typically, this is NOT a one pass algorithm

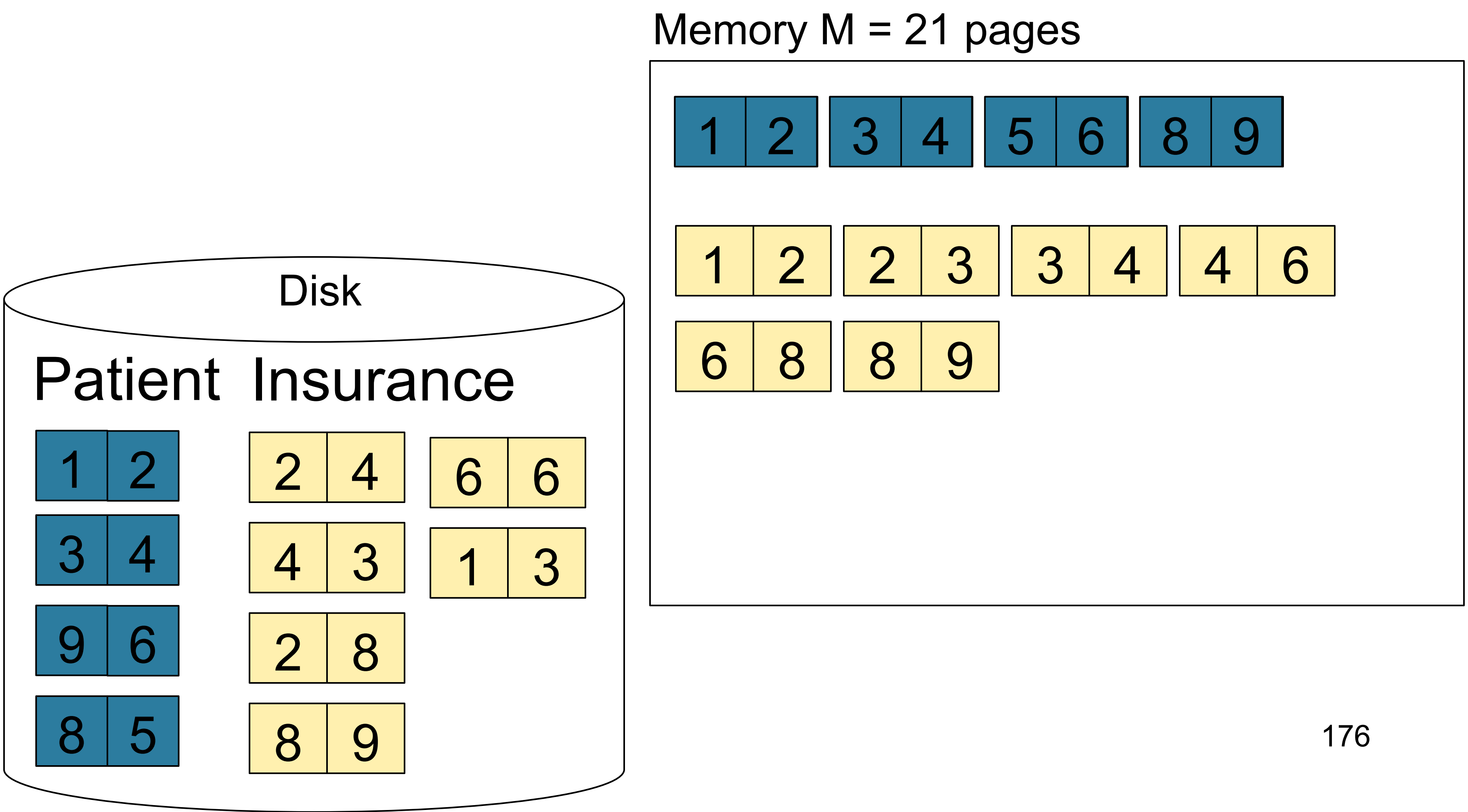
Sort-Merge Join Example

Step 1: Scan Patient and sort in memory



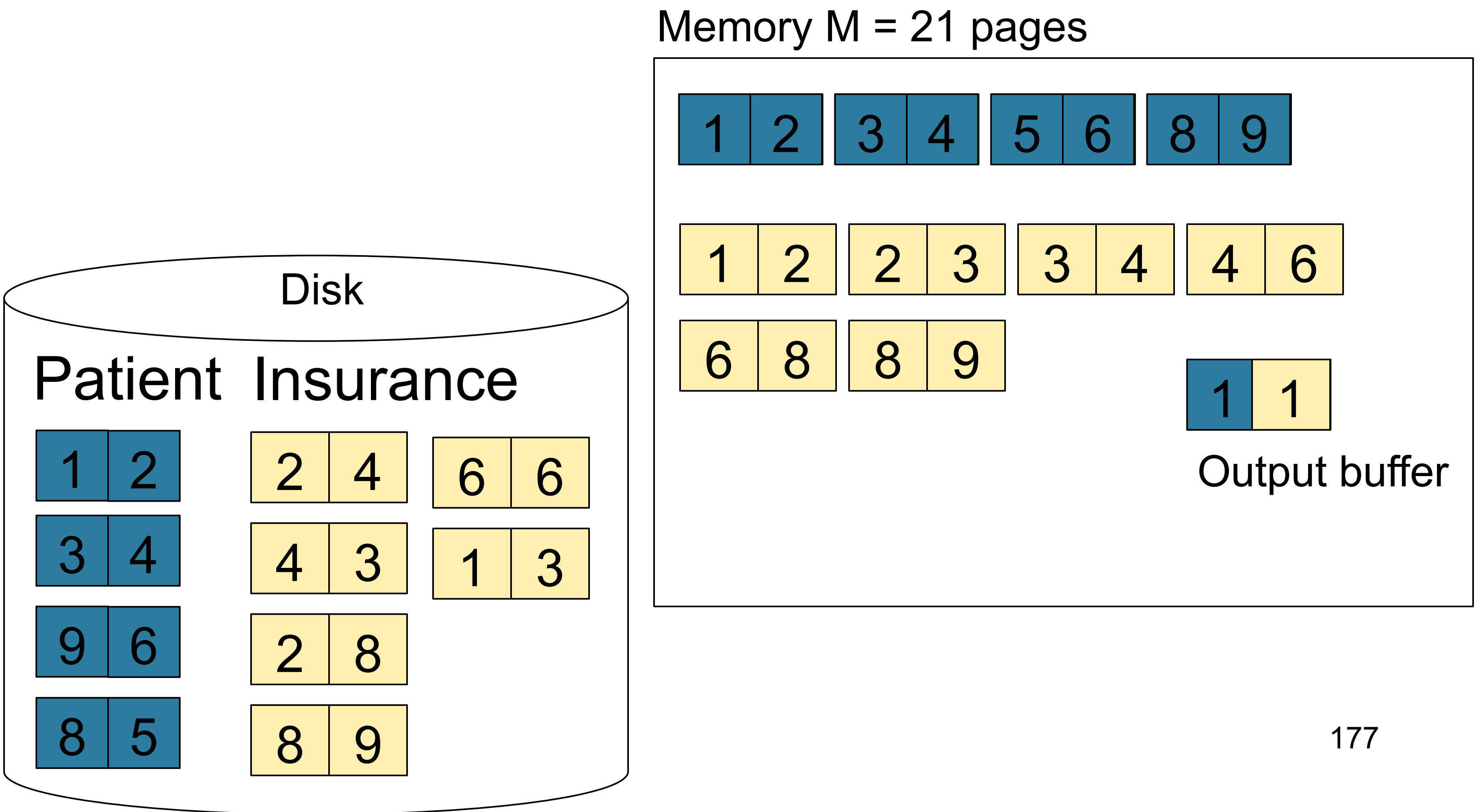
Sort-Merge Join Example

Step 2: Scan Insurance and sort in memory



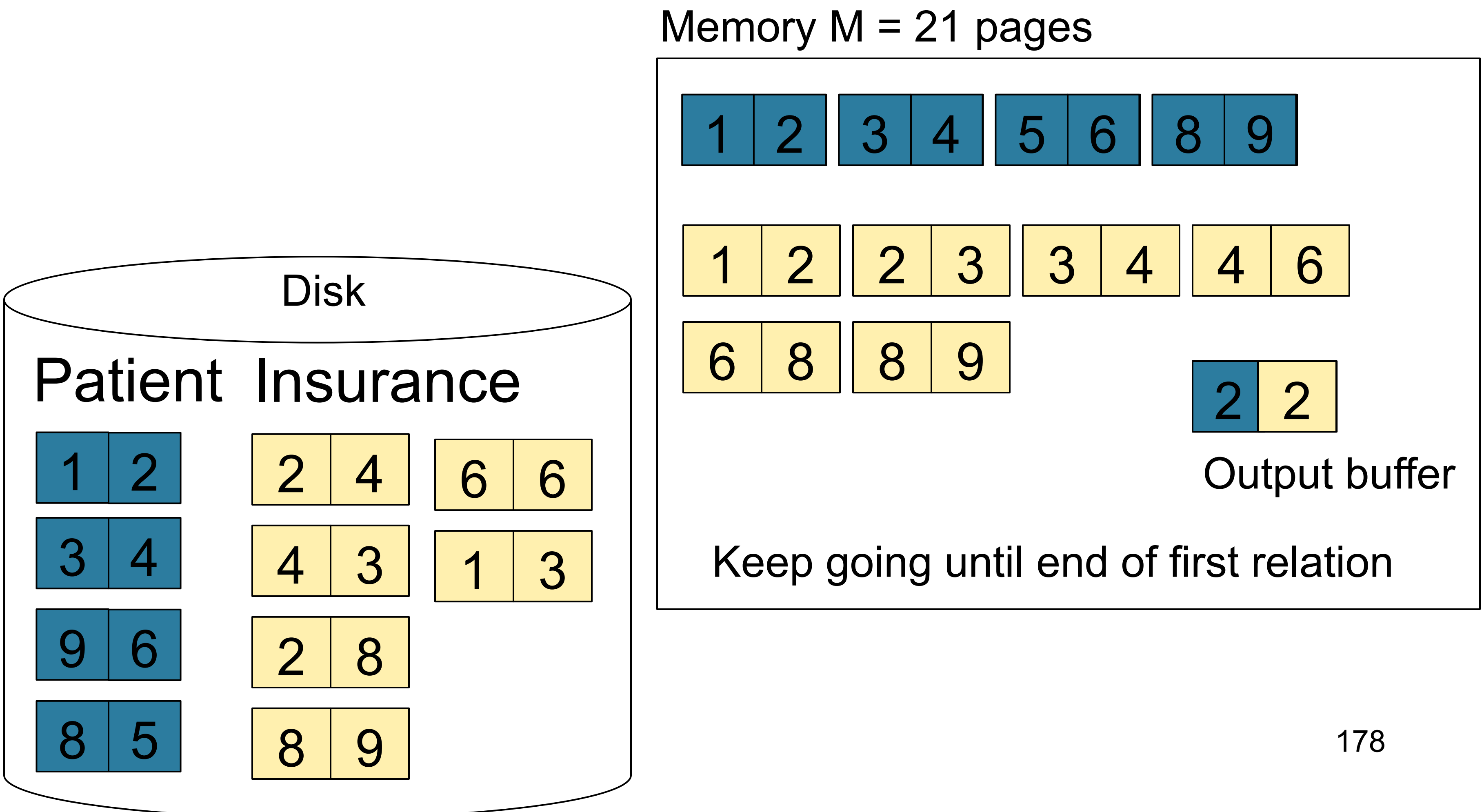
Sort-Merge Join Example

Step 3: Merge Patient and Insurance



Sort-Merge Join Example

Step 3: Merge Patient and Insurance



Index Join

R ⋈ S

- Assume S has an index on the join attribute
- Iterate over R, for each tuple fetch corresponding tuple(s) from S

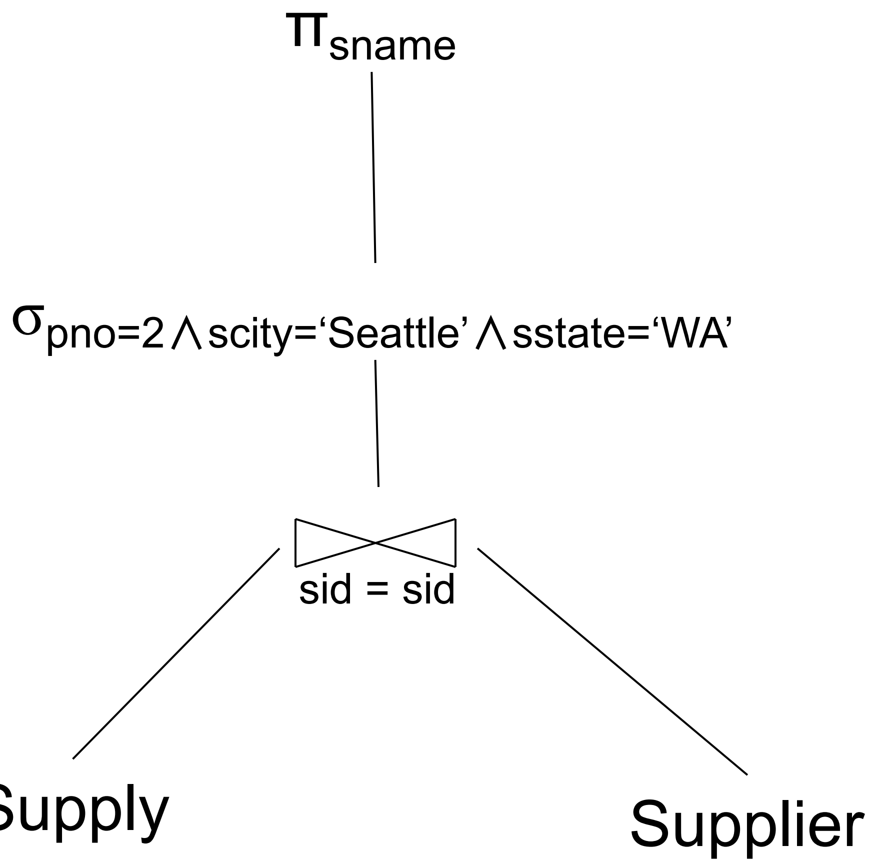
- Cost:
  - If index on S is clustered:
    B(R) + T(R) * (B(S) * 1/V(S,a))
  - If index on S is unclustered:
    B(R) + T(R) * (T(S) * 1/V(S,a))

Cost of Query Plans Example

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

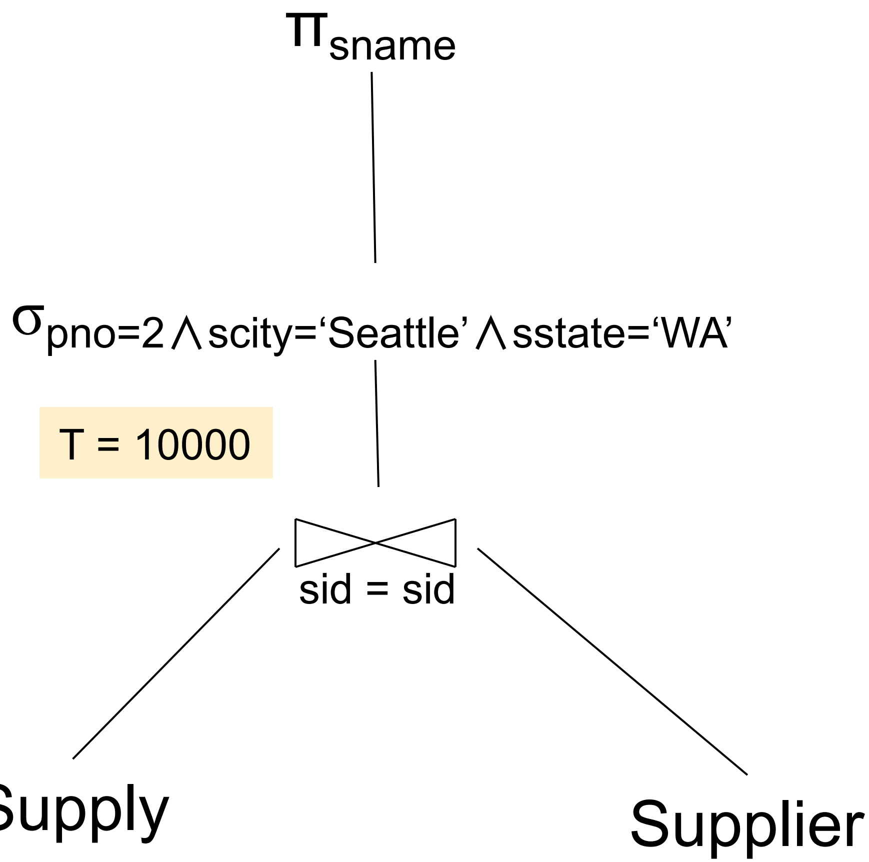
T(Supplier) = 1000
B(Supplier) = 100
V(Supplier, scity) = 20
V(Supplier, state) = 10

M=11

Supplier(<u>sid</u>, sname, scity, sstate)

Supply(<u>sid, pno</u>, 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

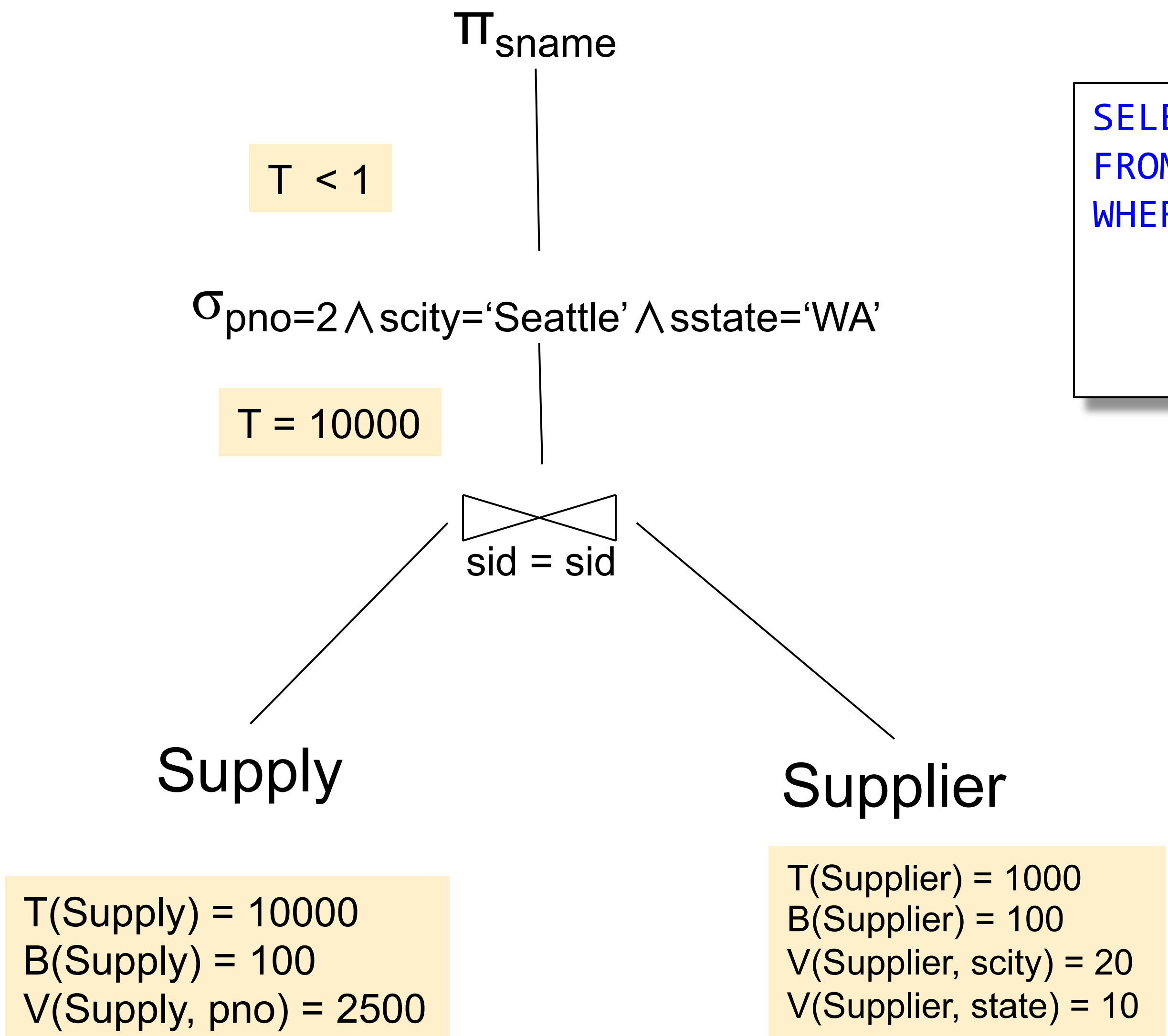
T(Supplier) = 1000
B(Supplier) = 100
V(Supplier, scity) = 20
V(Supplier, state) = 10

M=11

Supplier(sid, sname, scity, sstate)

Supply(sid, pno, quantity)

Logical Query Plan 1

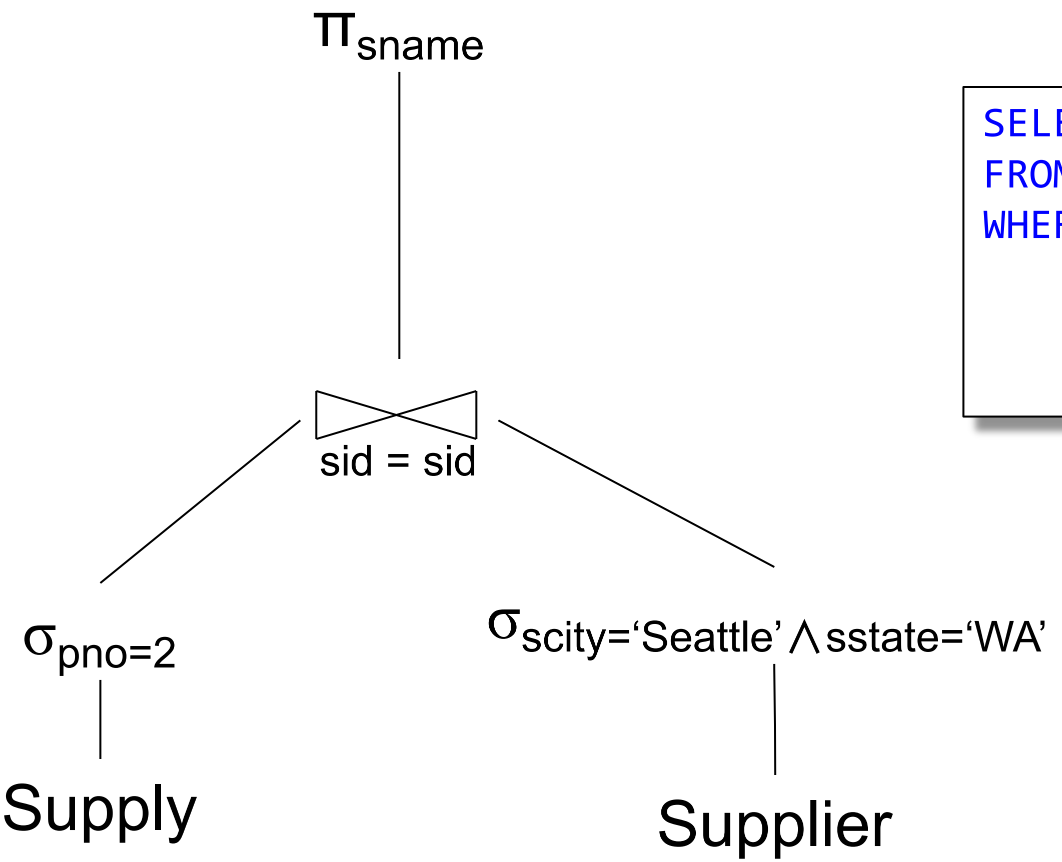



```
SELECT sname
FROM Supplier x, Supply y
WHERE x.sid = y.sid
    and y.pno = 2
    and x.scity = ‘Seattle’
    and x.sstate = ‘WA’
```

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’
```

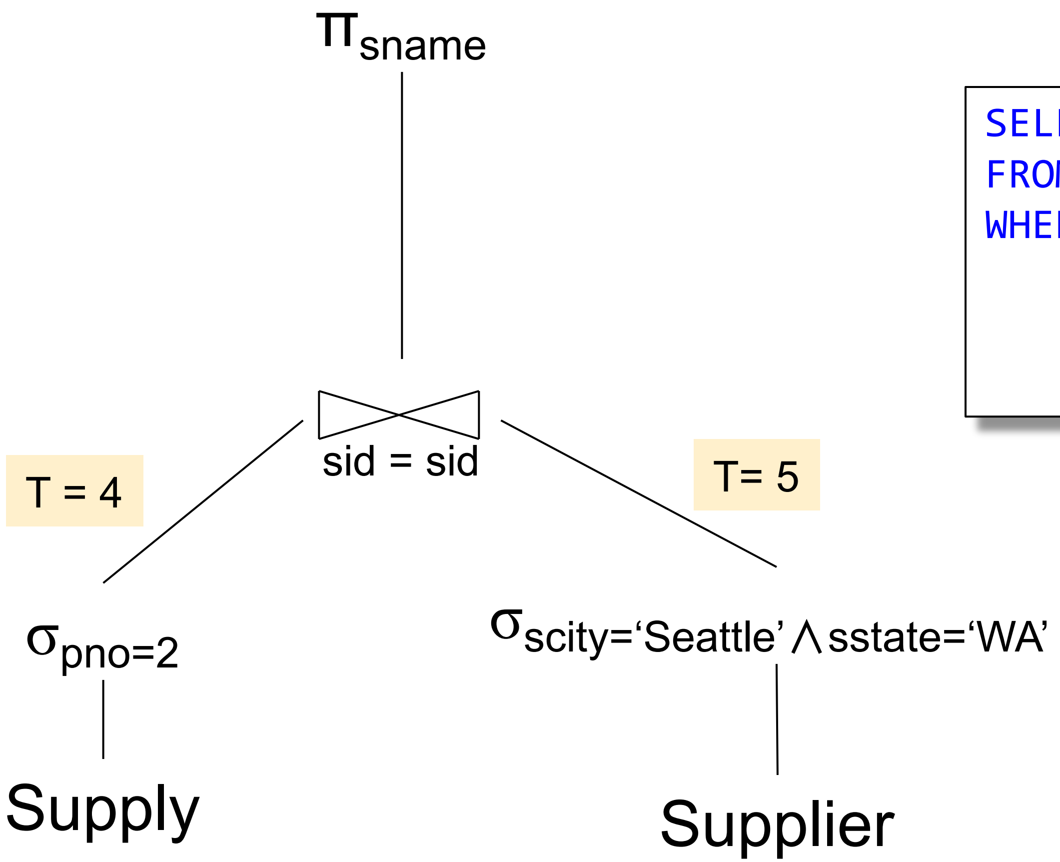
T(Supply) = 10000
B(Supply) = 100
V(Supply, pno) = 2500

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

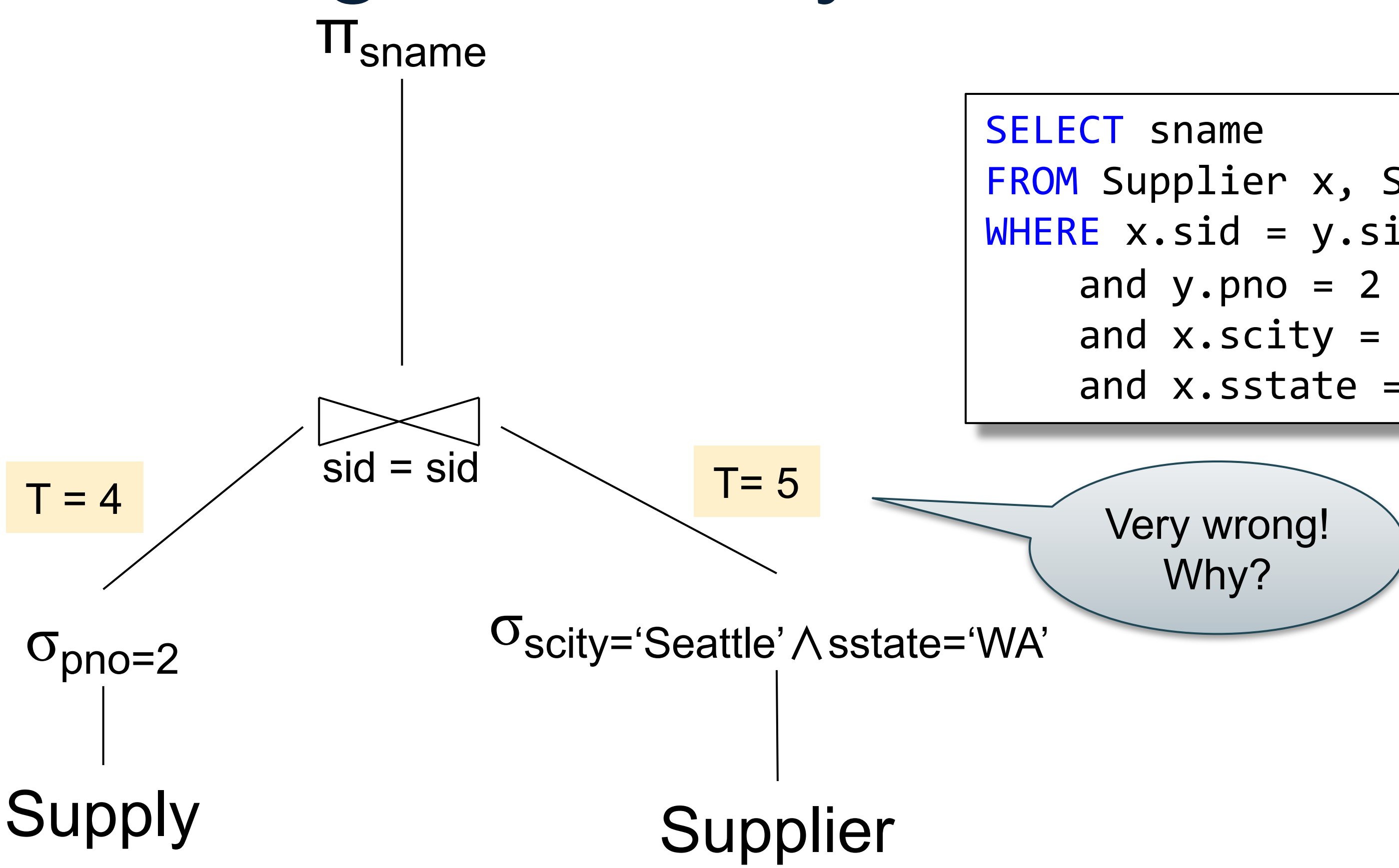
T(Supplier) = 1000
B(Supplier) = 100
V(Supplier, scity) = 20
V(Supplier, state) = 10

M=11

Supplier(<u>sid</u>, sname, scity, sstate)

Supply(<u>sid, pno</u>, 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

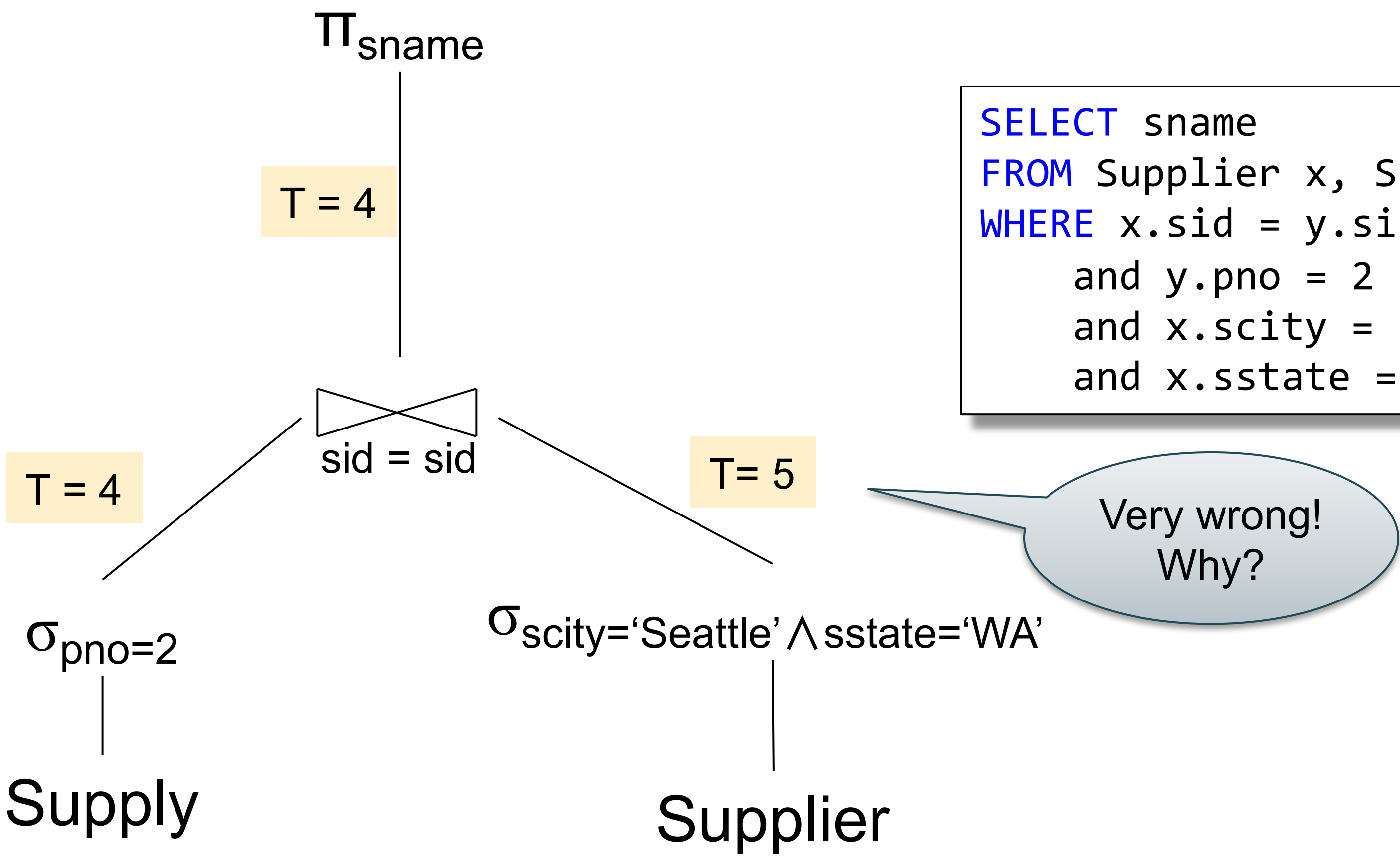
M=11

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’
```



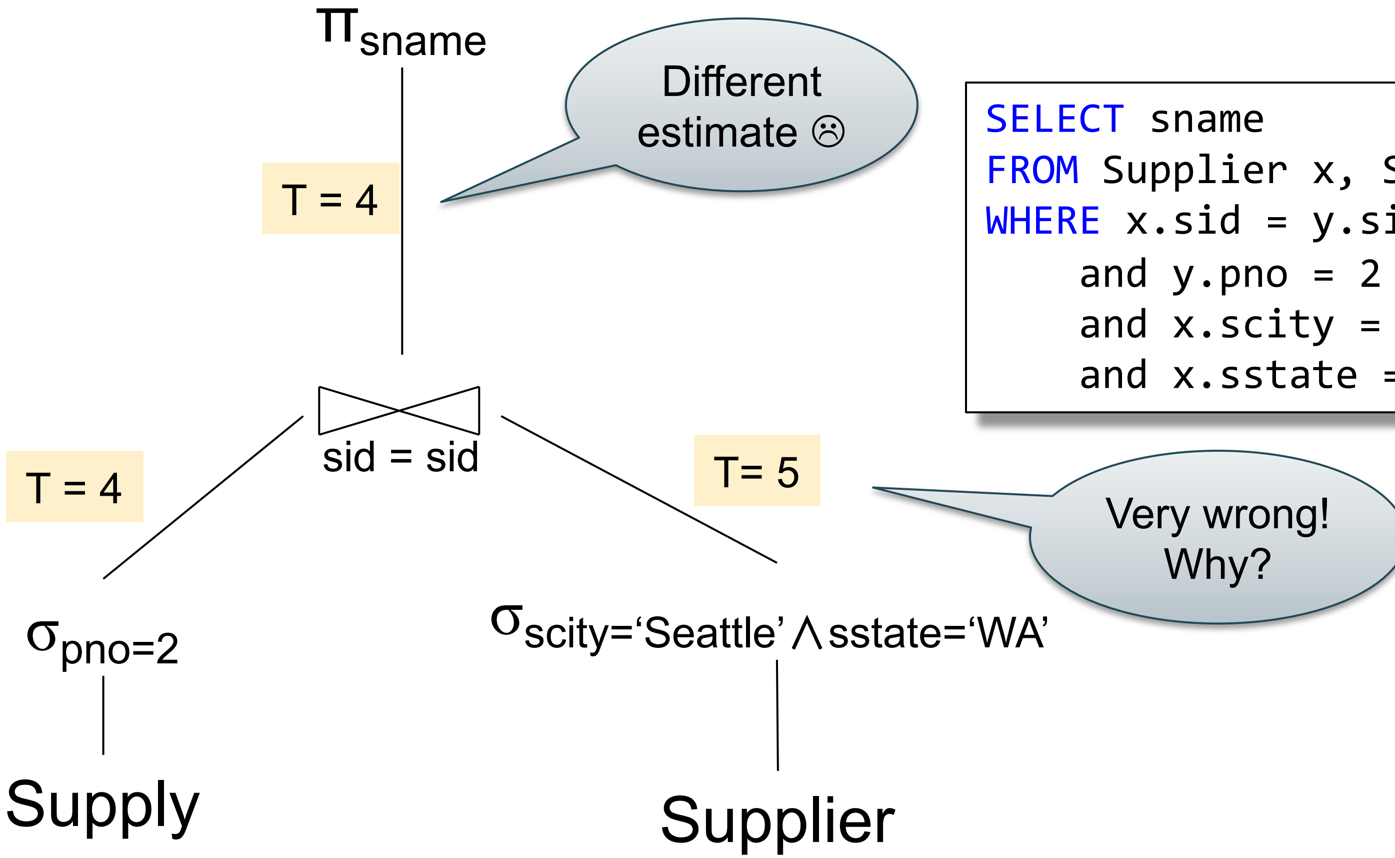
T(Supply) = 10000
B(Supply) = 100
V(Supply, pno) = 2500

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'
```

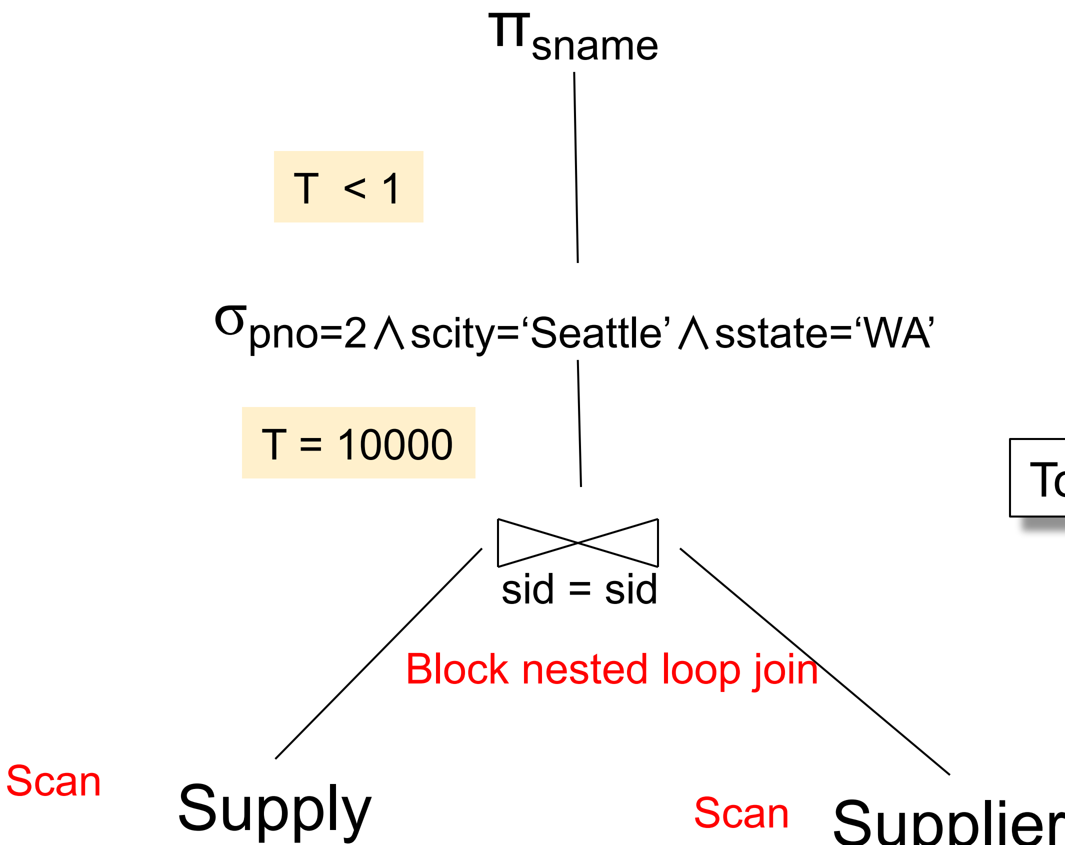
T(Supply) = 10000
B(Supply) = 100
V(Supply, pno) = 2500

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)

Physical Plan 1



Total cost:

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

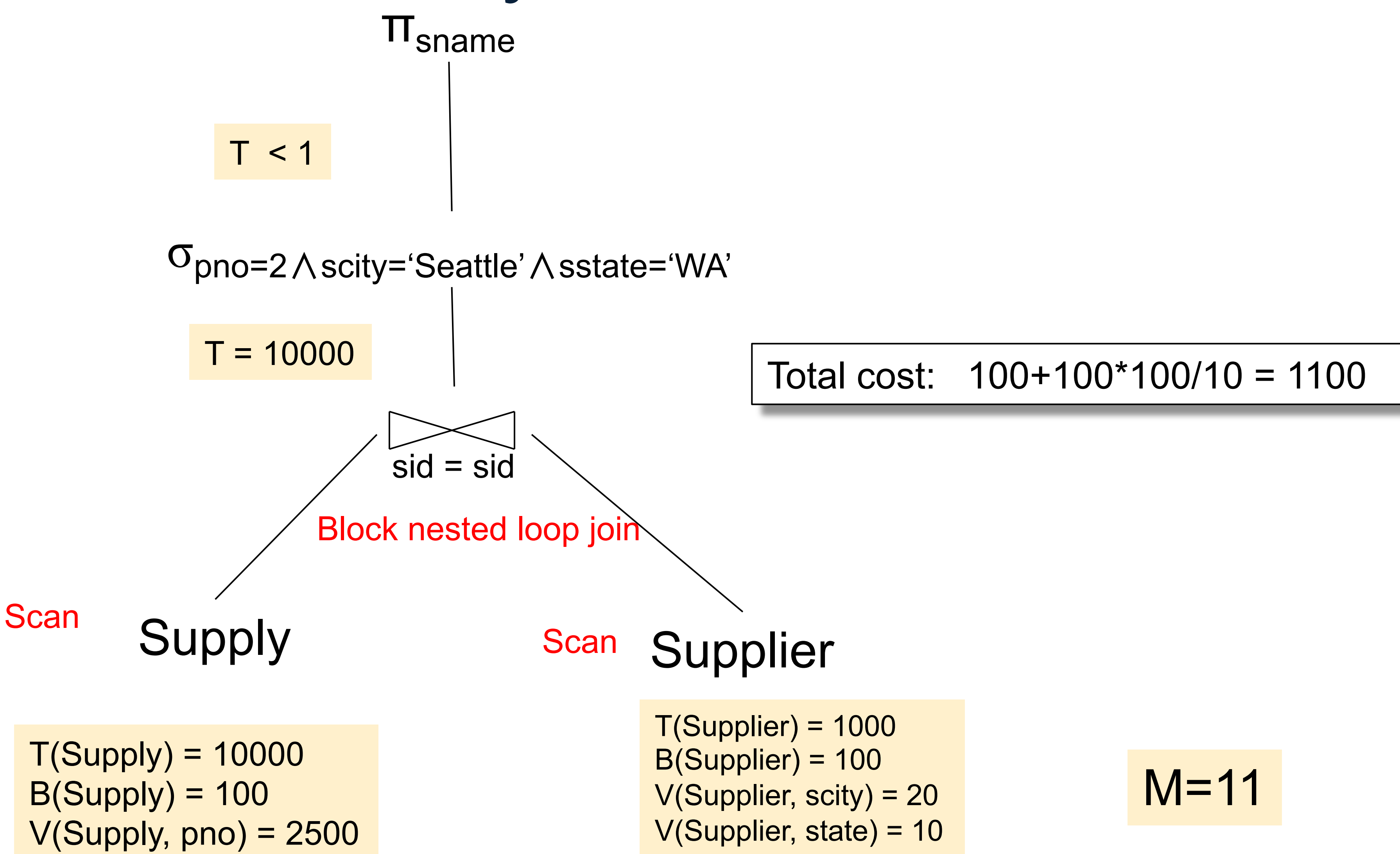
T(Supplier) = 1000
B(Supplier) = 100
V(Supplier, scity) = 20
V(Supplier, state) = 10

M=11

Supplier(sid, sname, scity, sstate)

Supply(sid, pno, quantity)

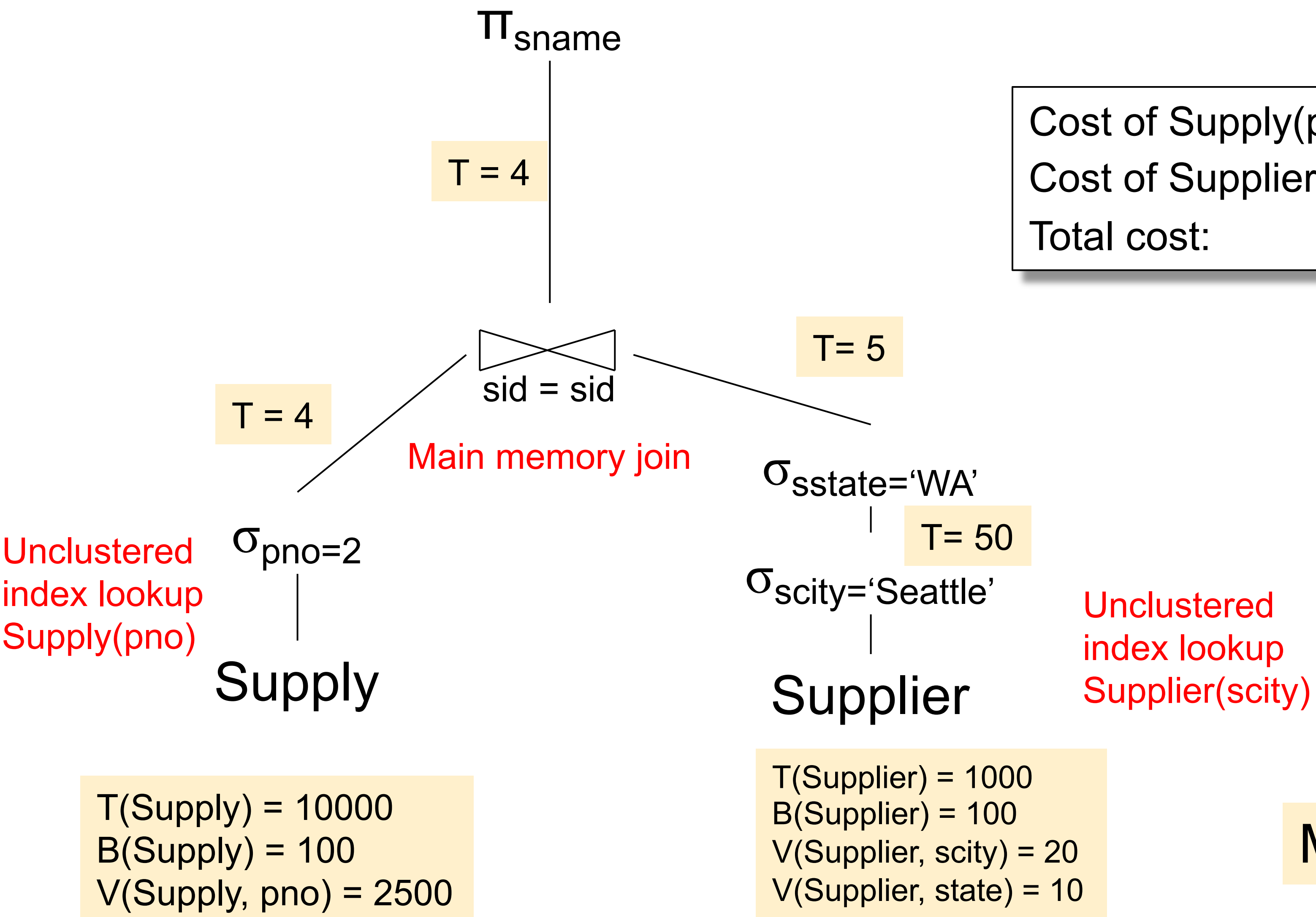
Physical Plan 1



Supplier(<u>sid</u>, sname, scity, sstate)

Supply(<u>sid, pno</u>, quantity)

Physical Plan 2



Cost of Supply(pno) =

Cost of Supplier(scity) =

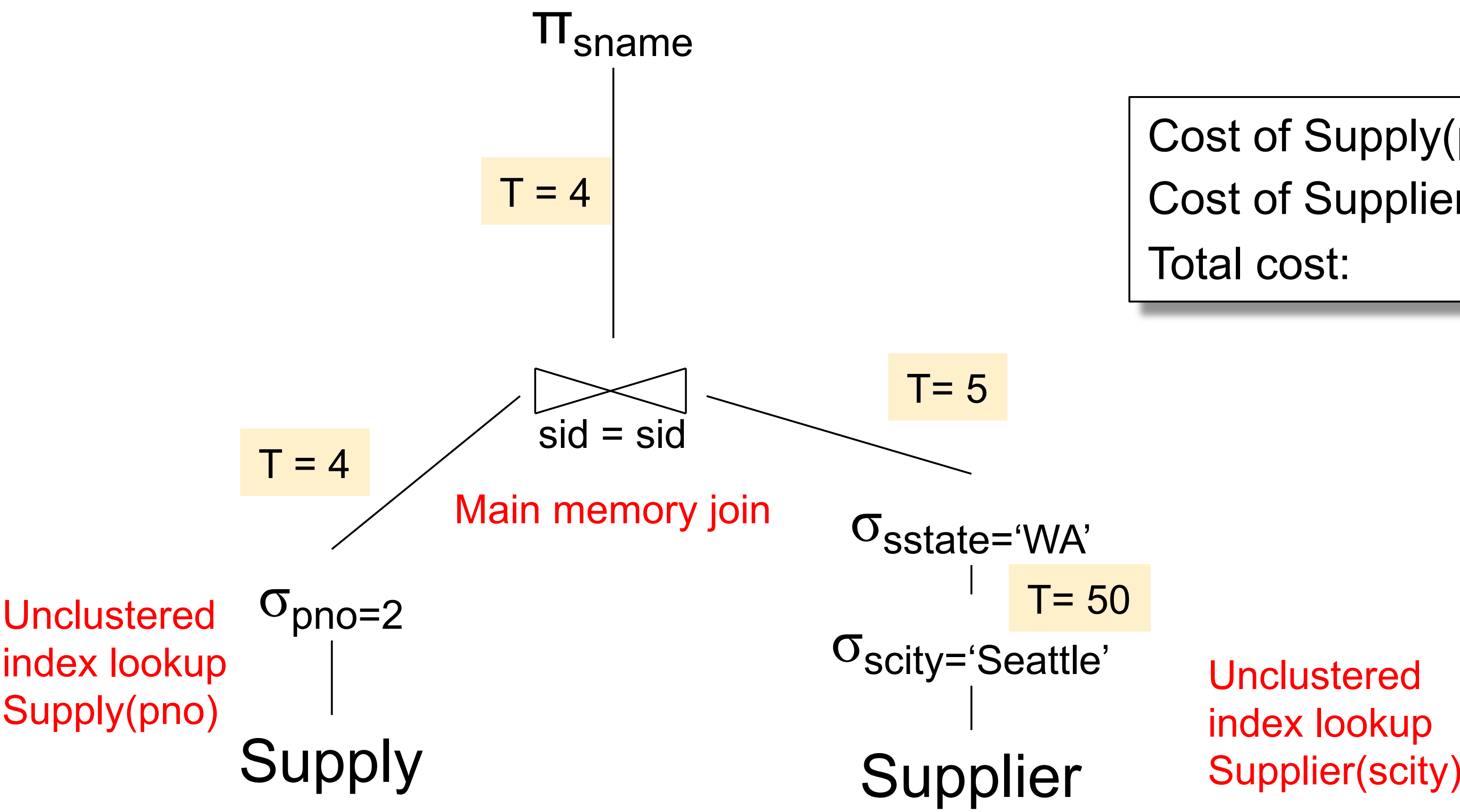
Total cost:

M=11

Supplier(sid, sname, scity, sstate)

Supply(sid, pno, quantity)

Physical Plan 2



Cost of Supply(pno) = 4

Cost of Supplier(scity) =

Total cost:

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

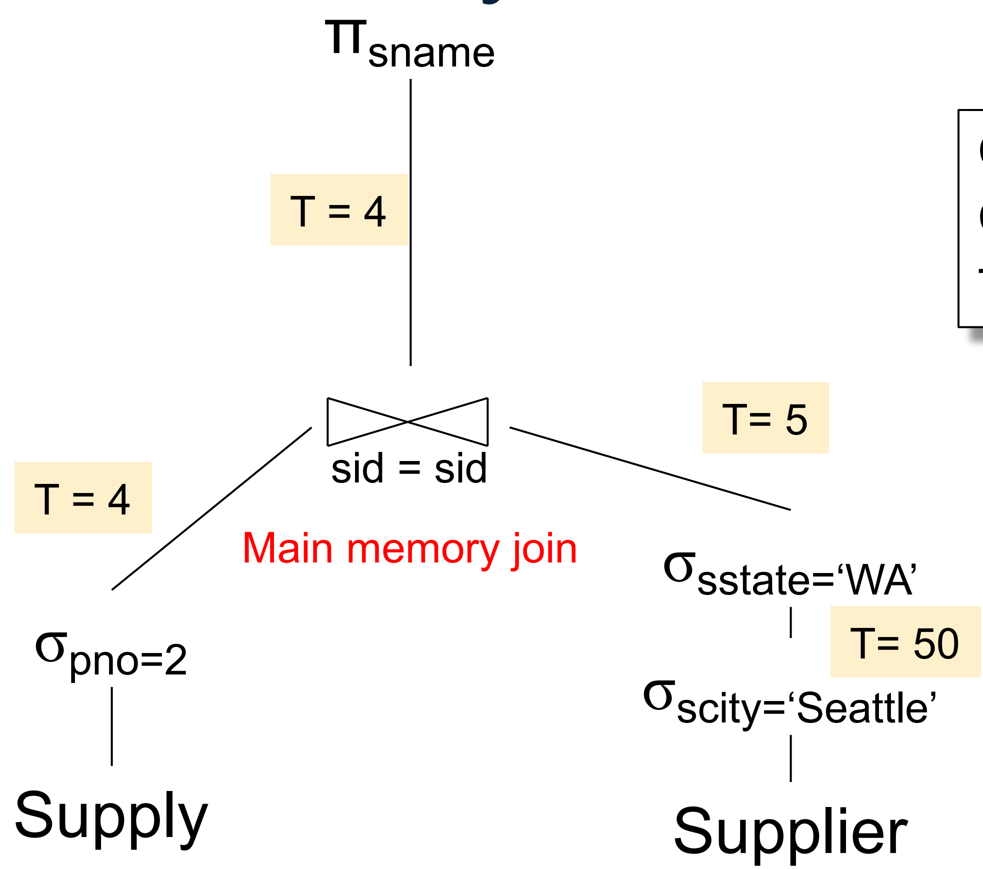
T(Supplier) = 1000
B(Supplier) = 100
V(Supplier, scity) = 20
V(Supplier, state) = 10

M=11

Supplier(sid, sname, scity, sstate)

Supply(sid, pno, quantity)

Physical Plan 2



Unclustered index lookup Supply(pno)

Unclustered index lookup Supplier(scity)

Cost of Supply(pno) = 4
Cost of Supplier(scity) = 50
Total cost: 54

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

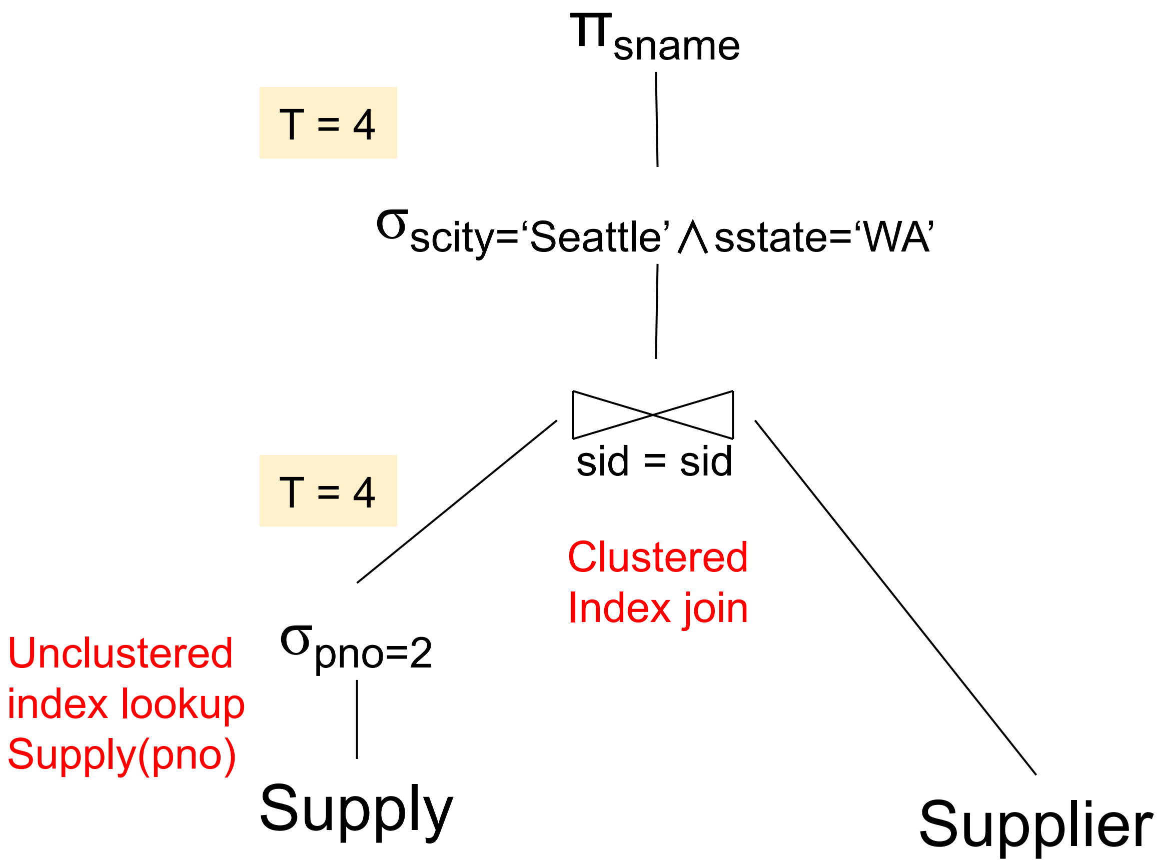
T(Supplier) = 1000
B(Supplier) = 100
V(Supplier, scity) = 20
V(Supplier, state) = 10

M=11

Supplier(sid, sname, scity, sstate)

Supply(sid, pno, quantity)

Physical Plan 3



Cost of Supply(pno) =

Cost of Index join =

Total cost:

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

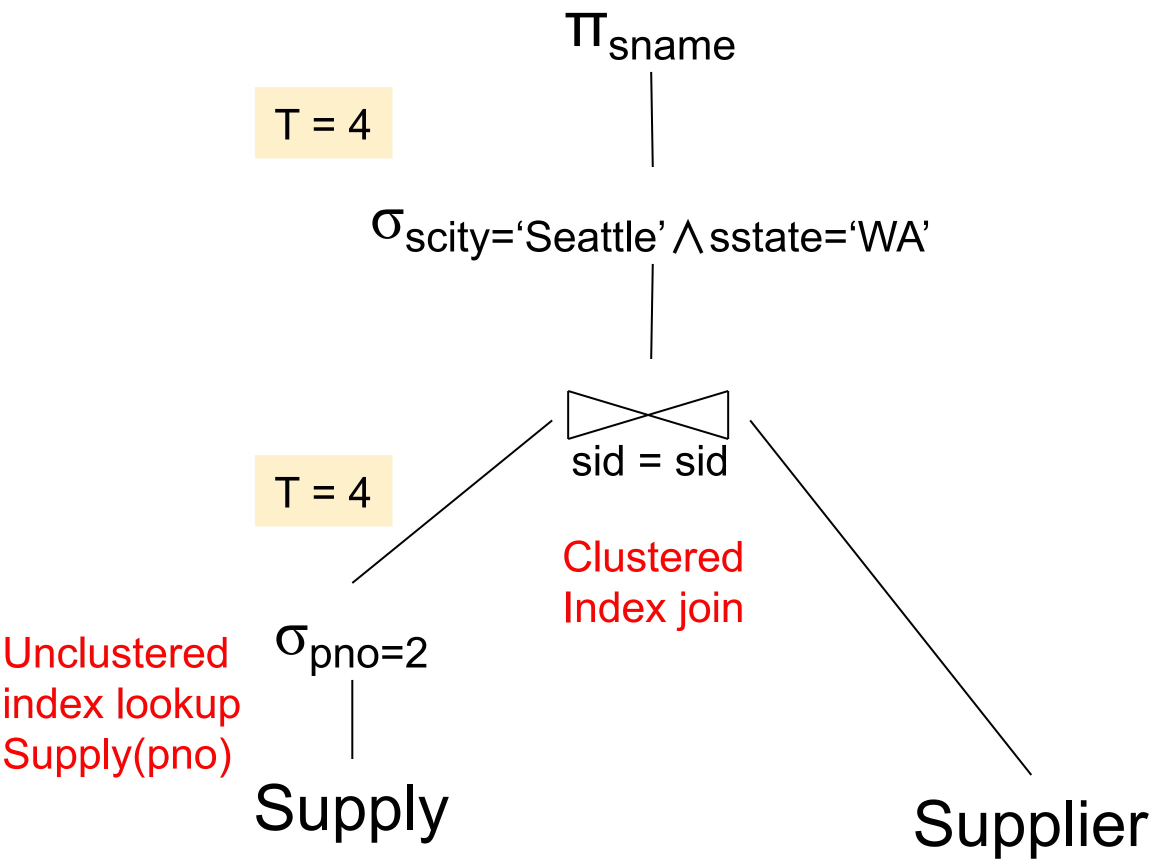
T(Supplier) = 1000
B(Supplier) = 100
V(Supplier, scity) = 20
V(Supplier, state) = 10

M=11

Supplier(sid, sname, scity, sstate)

Supply(sid, pno, quantity)

Physical Plan 3



Cost of Supply(pno) = 4

Cost of Index join =

Total cost:

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

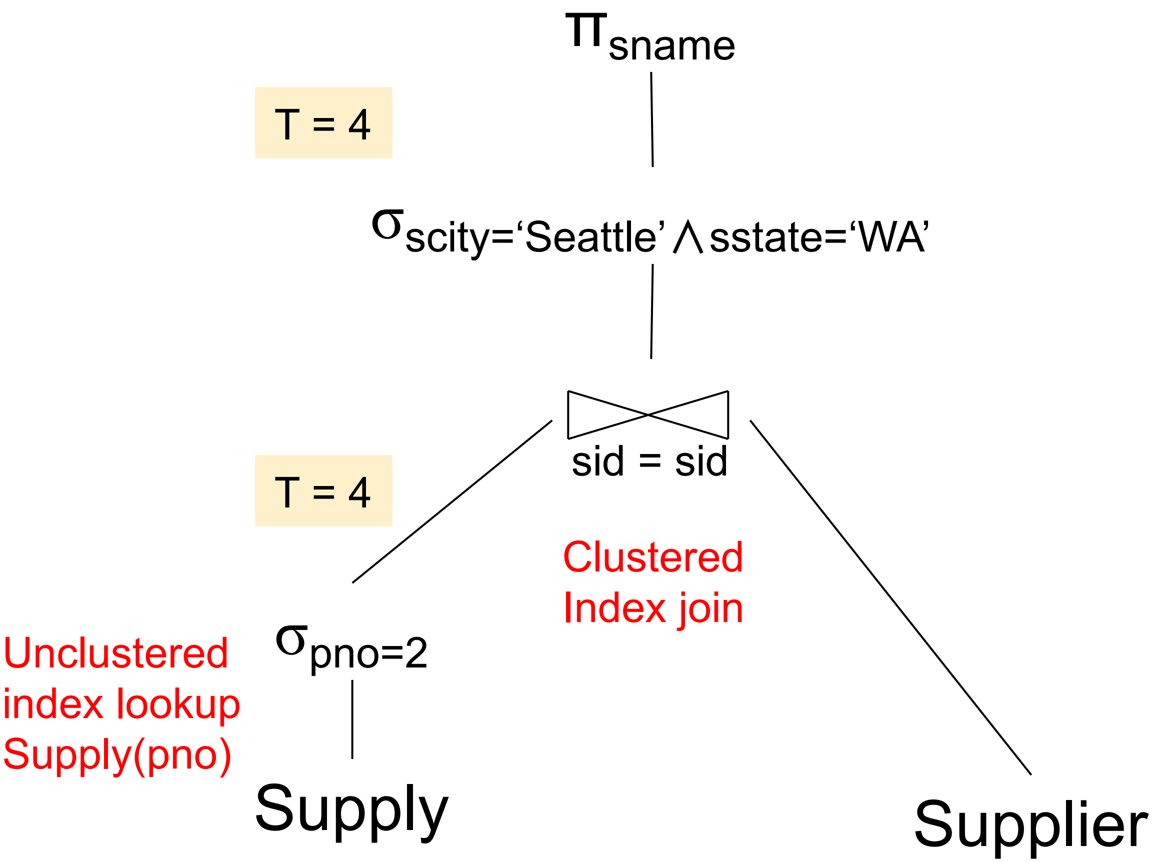
T(Supplier) = 1000
B(Supplier) = 100
V(Supplier, scity) = 20
V(Supplier, state) = 10

M=11

Supplier(sid, sname, scity, sstate)

Supply(sid, pno, quantity)

Physical Plan 3



Cost of Supply(pno) = 4

Cost of Index join = 4

Total cost: 8

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

Query Optimizer Summary

- Input: A logical query plan
- Output: A good physical query plan
- Basic query optimization algorithm
  - Enumerate alternative plans (logical and physical)
  - Compute estimated cost of each plan
  - Choose plan with lowest cost

- This is called cost-based optimization