

Fundamentals of Database Systems

Unit 5: RDBMS Internals
Logical and Physical Plans
Query Execution
Query Optimization

(3 lectures)

Fundamentals of Database Systems

Lecture 14: Introduction to Query Evaluation

Class Overview

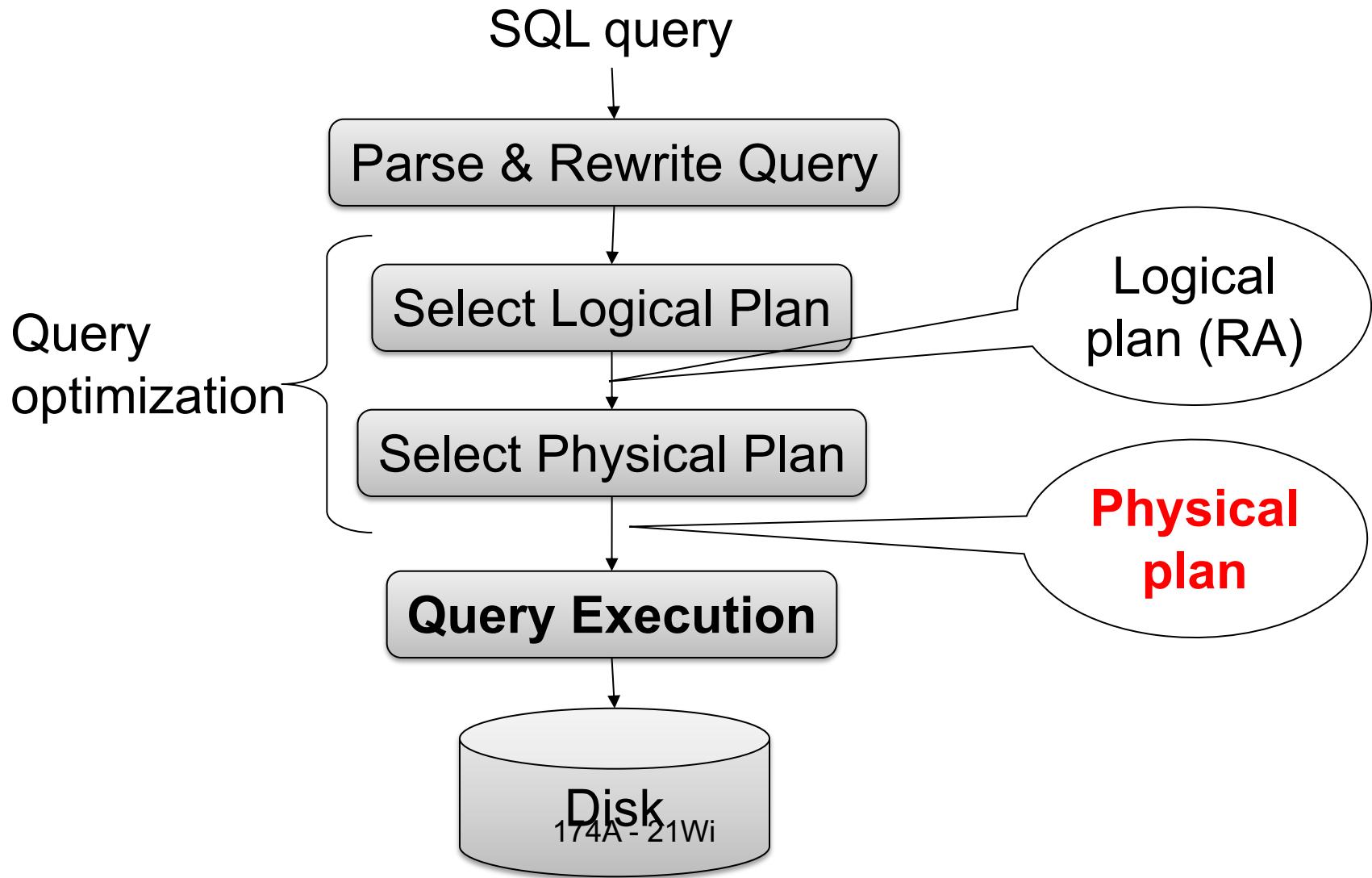
- Unit 1: Intro
- Unit 2: Relational Data Models and Query Languages
- Unit 3: Non-relational data
- Unit 4: DBMS usability, conceptual design
- Unit 5: RDMBS internals and query optimization
- Unit 6: Transactions

From Logical RA Plans to Physical Plans

Logistics

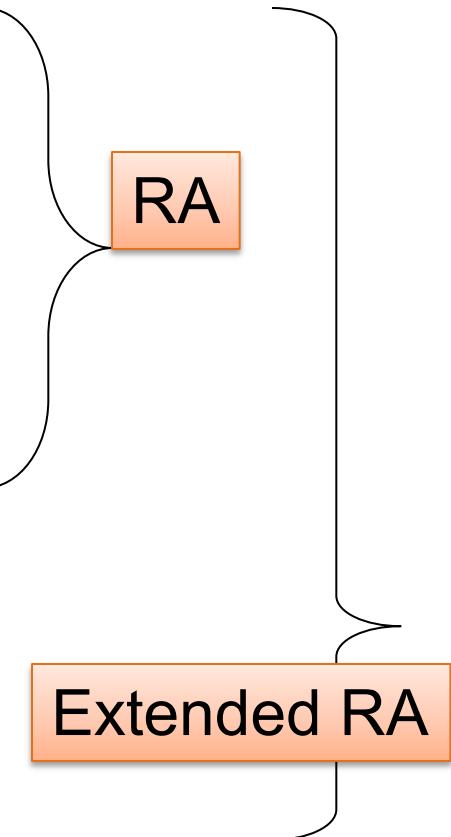
HW4 due is postponed to Thursday.
Come to OH if you have problems.

Query Evaluation Steps Review



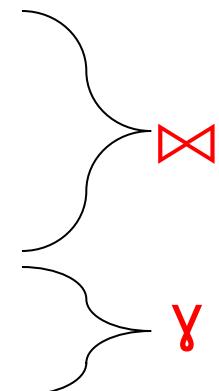
Relational Algebra Operators

- Union \cup , intersection \cap , difference $-$
- Selection σ
- Projection π
- Cartesian product \times , join \bowtie
- (Rename ρ)
- Duplicate elimination δ
- Grouping and aggregation γ
- Sorting τ



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



Supplier(sid, sname, scity, sstate)

Supply(sid, pno, quantity)

Main Memory Algorithms

Logical operator:

Supplier $\bowtie_{\text{sid}=\text{sid}}$ 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 ⚡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 $\bowtie_{\text{sid}=\text{sid}}$ Supply

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

1. Nested Loop Join
2. Merge join
3. Hash join

$O(n^2)$
 $O(n \log n)$ ↘ ↗
 $O(n) \dots O(n^2)$

$$D \rightarrow d \quad |D| > |d|$$

BRIEF Review of Hash Tables

Separate chaining:

A (naïve) hash function:

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

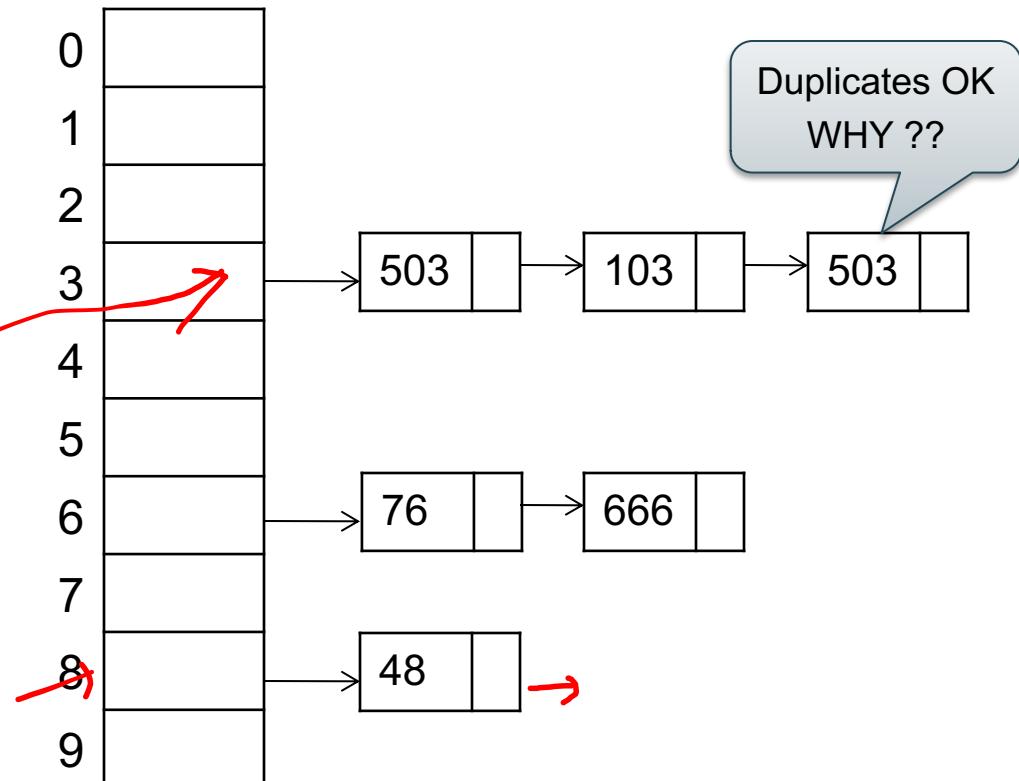
$$h(143) = 3$$

Operations:

$$\underline{\text{find}(103) = ??}$$

$$\underline{\text{insert}(488) = ??}$$

$$h(488)$$



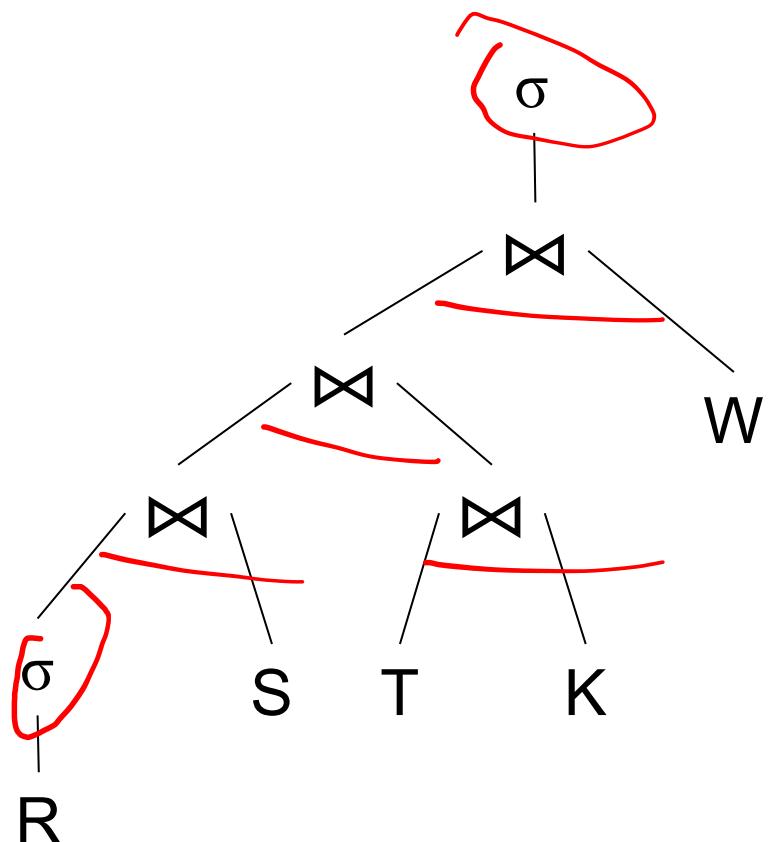
BRIEF Review of Hash Tables

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

Recap of Main Memory Algorithms

- Join \bowtie :
 - Nested loop join
 - Hash join
 - Merge join
- Selection σ
 - “on-the-fly”
 - Index-based selection (next lecture)
- Group by γ
 - Hash-based
 - Merge-based

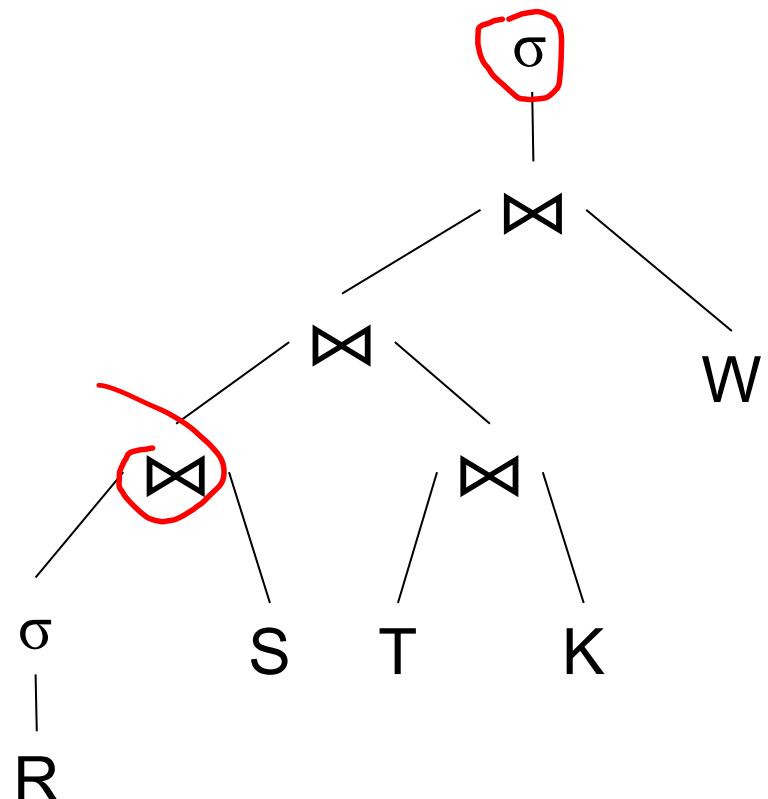
How Do We Combine Them?



How Do We Combine Them?

The Iterator Interface

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



Implementing Query Operators with the Iterator Interface

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

Example “on the fly” selection operator

```
interface Operator {  
  
    // initializes operator state  
    // and sets parameters  
    void open (...);
```

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

174A - 21Wi

}

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

Example “on the fly” selection operator

```
interface Operator {  
  
    // initializes operator state  
    // and sets parameters  
    void open (...);  
  
    // calls next() on its inputs  
    // processes an input tuple  
    // produces output tuple(s)  
    // returns null when done  
    Tuple next ();  
  
    // cleans up (if any)  
    void close ();  
}
```

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

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 () {  
        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

Example “on the fly” selection operator

```
interface Operator {  
  
    // initializes operator state  
    // and sets parameters  
    void open (...);  
  
    // calls next() on its inputs  
    // processes an input tuple  
    // produces output tuple(s)  
    // returns null when done  
    Tuple next ();  
  
    // cleans up (if any)  
    void close ();  
}
```

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

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

(On the fly)

Π_{sname}

(On the fly)

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

(Nested loop)

$sid = sid$

Supplier
(File scan)

Supply
(File scan)

Discuss: open/next/close
for nested loop join



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

`Supply(sid, pno, quantity)`

Pipelining

(On the fly)

Π_{sname} **open()**

(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)

Discuss: open/next/close
for nested loop join

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

`Supply(sid, pno, quantity)`

Pipelining

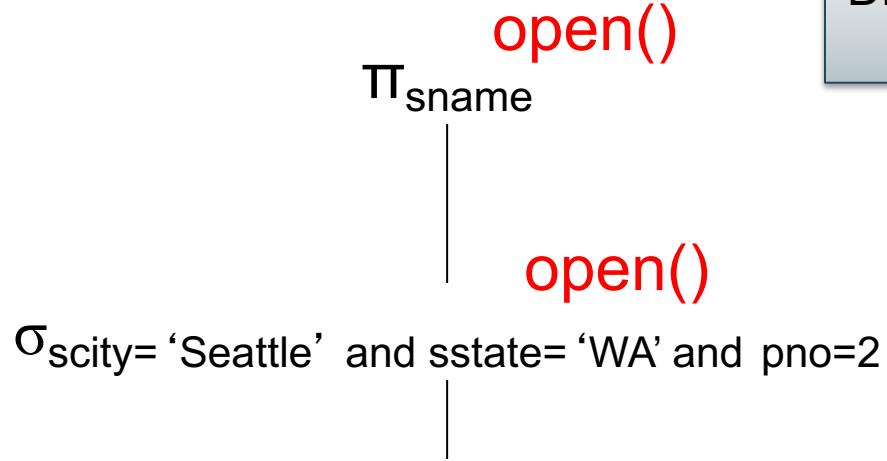
(On the fly)

(On the fly)

(Nested loop)

Supplier
(File scan)

Supply
(File scan)



Discuss: open/next/close
for nested loop join

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

`Supply(sid, pno, quantity)`

Pipelining

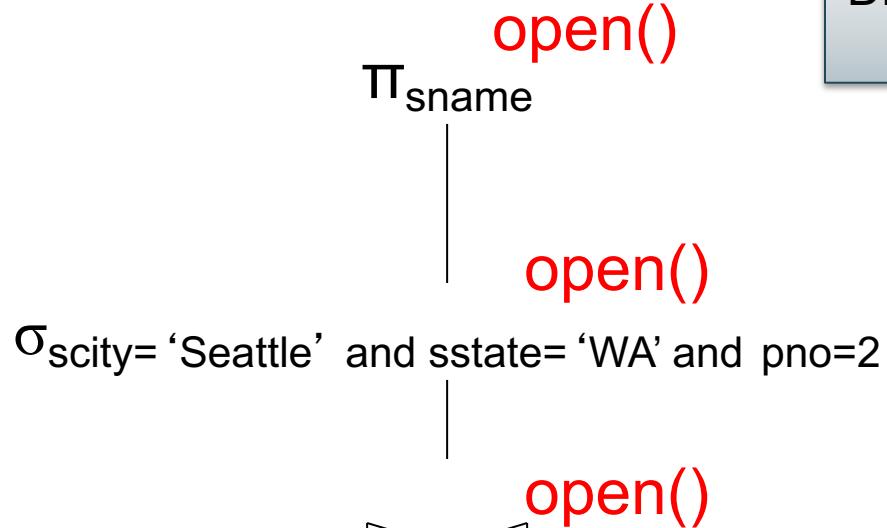
(On the fly)

(On the fly)

(Nested loop)

Supplier
(File scan)

Supply
(File scan)



Discuss: open/next/close
for nested loop join

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

`Supply(sid, pno, quantity)`

Pipelining

(On the fly)

(On the fly)

(Nested loop)

`Supplier
(File scan)`

`Supply
(File scan)`

Π_{sname}

`open()`

`open()`

`open()`

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

$sid = sid$

Discuss: open/next/close
for nested loop join

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

`Supply(sid, pno, quantity)`

Pipelining

(On the fly)

(On the fly)

(Nested loop)

`Supplier
(File scan)`

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

Π_{sname}

`open()`

`open()`

$sid = sid$

Discuss: open/next/close
for nested loop join

`Supply
(File scan)`

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

`Supply(sid, pno, quantity)`

Pipelining

(On the fly)

Π_{sname}
next()

(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)

Discuss: open/next/close
for nested loop join

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

`Supply(sid, pno, quantity)`

Pipelining

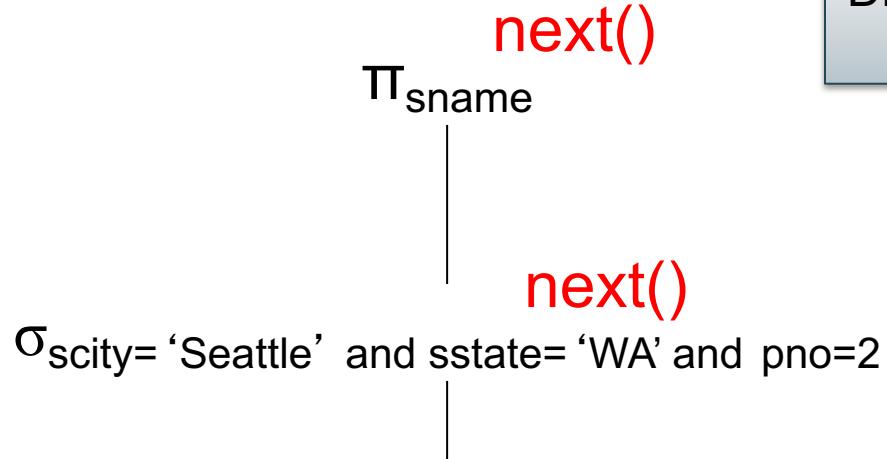
(On the fly)

(On the fly)

(Nested loop)

Supplier
(File scan)

Supply
(File scan)



Discuss: open/next/close
for nested loop join

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

`Supply(sid, pno, quantity)`

Pipelining

(On the fly)

(On the fly)

(Nested loop)

Supplier
(File scan)

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

π_{sname}

next()

next()

sid = sid

Discuss: open/next/close
for nested loop join

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

`Supply(sid, pno, quantity)`

Pipelining

(On the fly)

(On the fly)

(Nested loop)

next()

Supplier
(File scan)

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

Π_{sname}

next()

next()

next()

\bowtie
sid = sid

Supply
(File scan)

Discuss: open/next/close
for nested loop join

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

`Supply(sid, pno, quantity)`

Pipelining

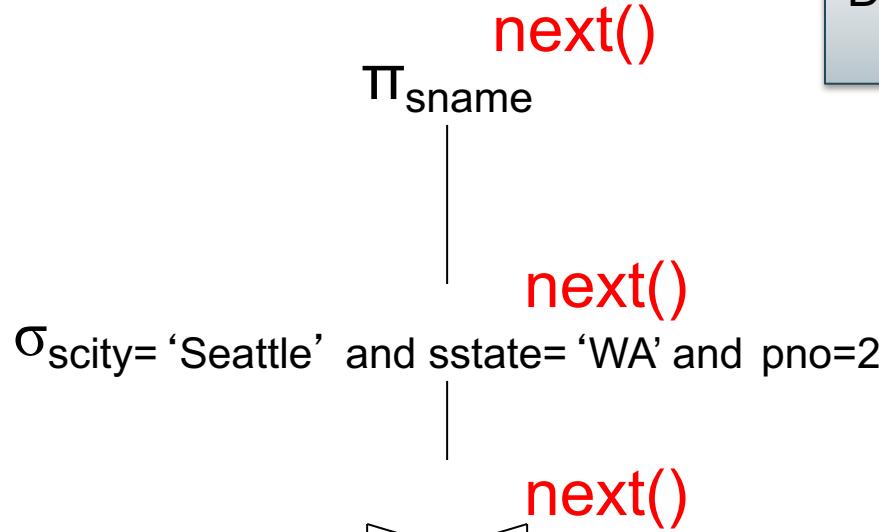
(On the fly)

(On the fly)

(Nested loop)

Supplier
(File scan)

Supply
(File scan)



Discuss: open/next/close
for nested loop join

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

`Supply(sid, pno, quantity)`

Pipelining

(On the fly)

(On the fly)

(Nested loop)

next()

Supplier
(File scan)

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

π_{sname}

next()

next()

next()

sid = sid

next()
next()

Supply
(File scan)

Discuss: open/next/close
for nested loop join

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

`Supply(sid, pno, quantity)`

Pipelining

(On the fly)

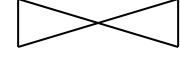
Π_{sname}

Discuss hash-join
in class

(On the fly)

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

(Hash Join)

 sid = sid

Supplier
(File scan)

Supply
(File scan)

Supplier(sid, sname, scity, sstate)

Supply(sid, pno, quantity)

Pipelining

(On the fly)

Π_{sname}

Discuss hash-join
in class

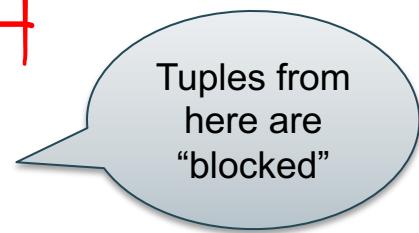
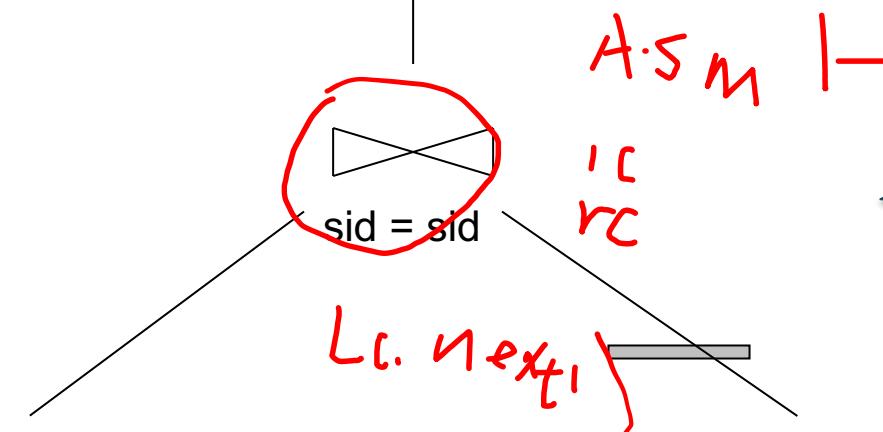
(On the fly)

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

(Hash Join)

Supplier
(File scan)

Supply
(File scan)



Supplier(sid, sname, scity, sstate)

Supply(sid, pno, quantity)

Pipelining

(On the fly)

Π_{sname}

Discuss hash-join
in class

(On the fly)

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

(Hash Join)

Tuples from here are pipelined

\bowtie
 $sid = sid$

Tuples from here are “blocked”

Supplier
(File scan)

Supply
(File scan)

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

`Supply(sid, pno, quantity)`

Blocked Execution

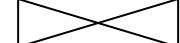
(On the fly)

Π_{sname}

(On the fly)

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

(Merge Join)


sid = sid

Supplier
(File scan)

Supply
(File scan)

Discuss merge-join
in class

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

`Supply(sid, pno, quantity)`

Blocked Execution

(On the fly)

Π_{sname}

Discuss merge-join
in class

(On the fly)

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

(Merge Join)

$sid = sid$

E

Blocked

Blocked

Supplier
(File scan)

Supply
(File scan)

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

More components of a 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

Fundamentals of Database Systems

Lecture 13: Basics of Data Storage and Indexes

Query Performance

To understand query 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**

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
 - Good: read blocks 1,2,3,4,5,...
 - Bad: read blocks 2342, 11, 321,9, ...
- Rule of thumb: *10%*
 - Random read 1-2% of file \approx sequential scan entire file;
 - 1-2% decreases over time, because of increased density



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

Student		
ID	fName	IName
10	Tom	Hanks
20	Amy	Hanks
...		

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

columns

ID	fName	IName
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



This is not a pipe.

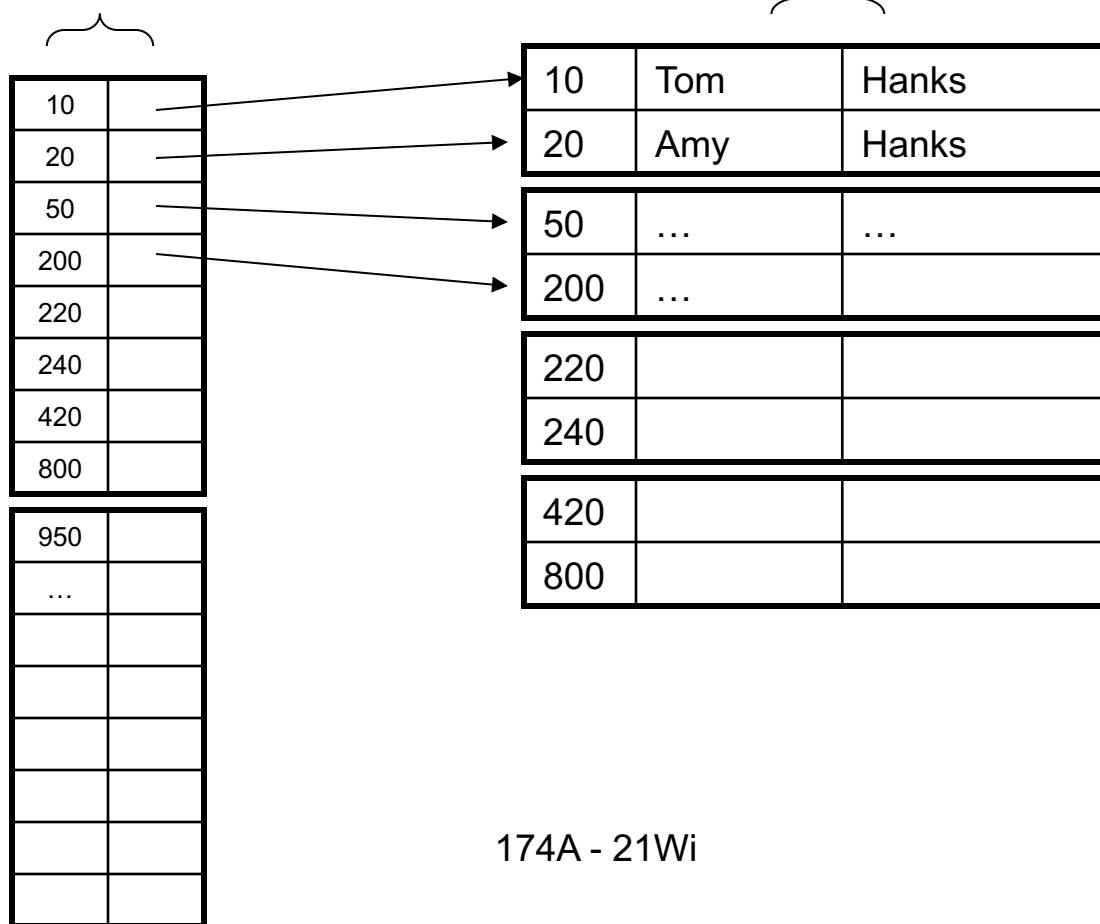
174A - 21Wi



Example 1: Index on ID

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

Data File **Student**



Example 2:

Index on fName

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

Amy	
Ann	
Bob	
Cho	
...	
...	
...	
...	

...	
...	
Tom	

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

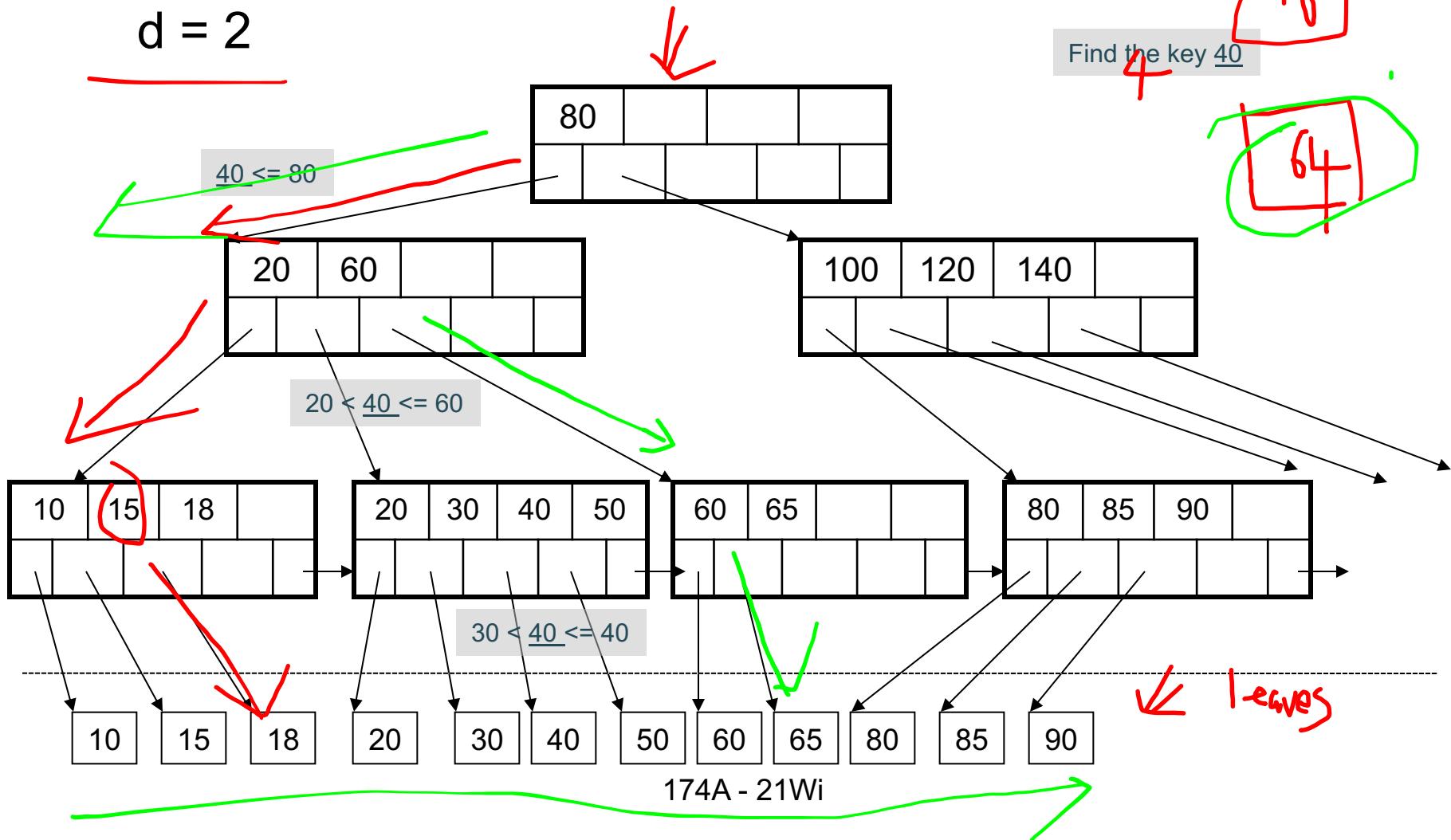
Data File Student

Index Organization

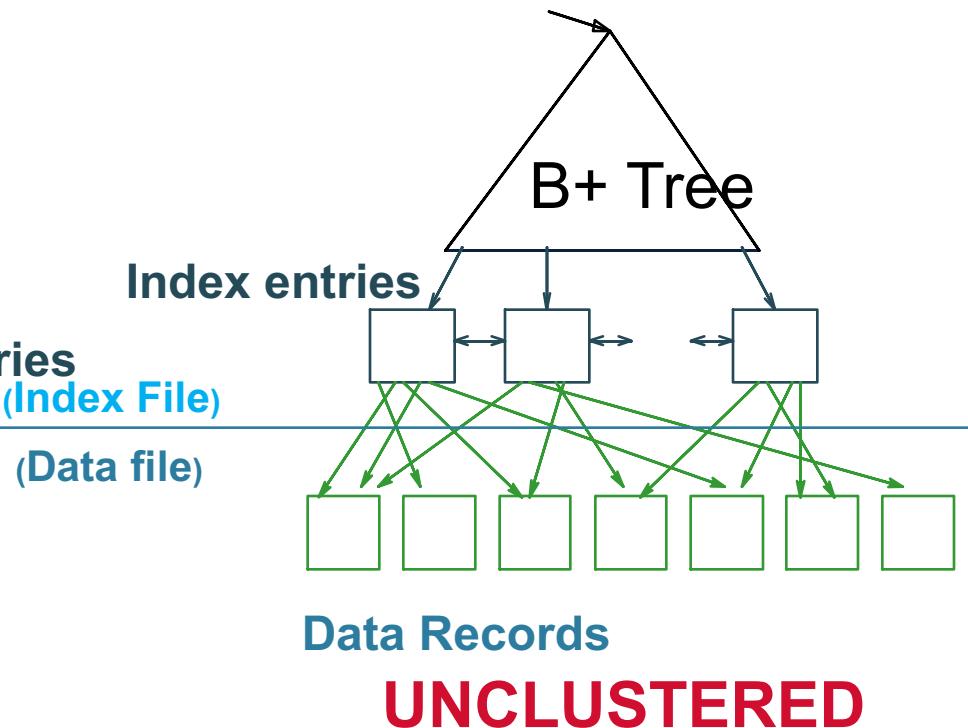
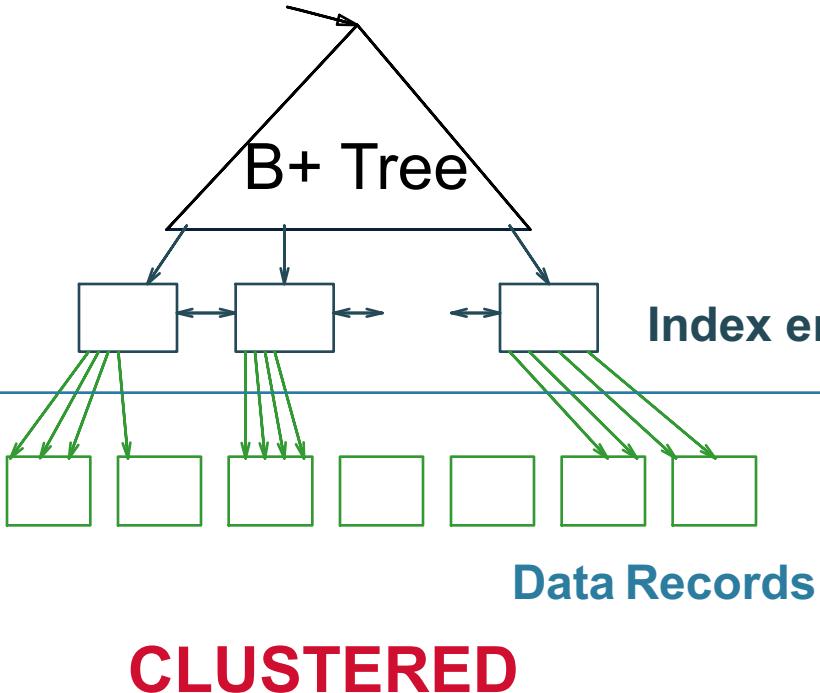
- Hash table
- B+ trees – most common
 - They are search trees, but they are not binary instead have higher fan-out
 - Will discuss them briefly next
- Specialized indexes: bit maps, R-trees, inverted index

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

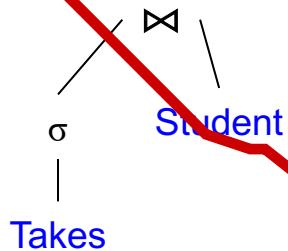
- **Clustered/unclustered**
 - Clustered = records close in index are close in data
 - Option 1: Data inside data file is sorted on disk
 - Option 2: Store data directly inside the index (no separate files)
 - Unclustered = records close in index may be far in data
- **Primary/secondary**
 - Meaning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

END

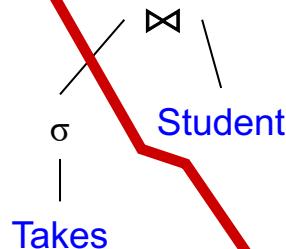
`Student(ID, fname, lname)`
`Takes(studentID, courseID)`



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

Example

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

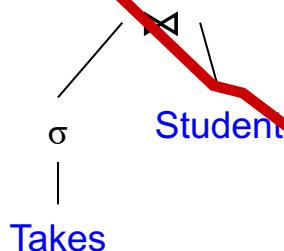


```
SELECT *
FROM Student x, Takes y
WHERE x.ID=y.studentID AND y.courseID > 300
```

Example

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

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



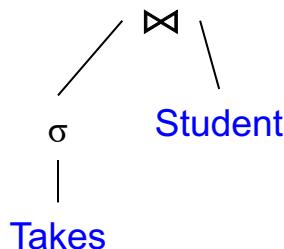
```
SELECT *
FROM Student x, Takes y
WHERE x.ID=y.studentID AND y.courseID > 300
```

Example

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

- Assume the database has indexes on these attributes:
- **Takes_courseID** = index on `Takes.courseID`
 - **Student_ID** = index on `Student.ID`

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



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

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

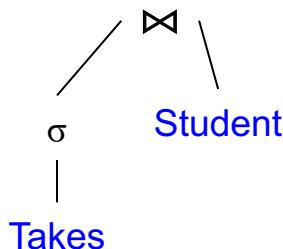
Assume the database has indexes on these attributes:

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

Index selection

```
for y' in Takes_courseID where y'.courseID > 300
```

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



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

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

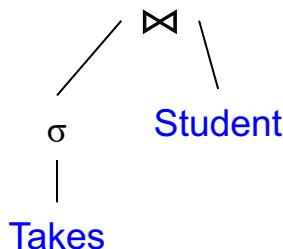
Assume the database has indexes on these attributes:

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

Index selection

```
for y' in Takes_courseID where y'.courseID > 300
  y = fetch the Takes record pointed to by y'
```

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



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

Assume the database has indexes on these attributes:

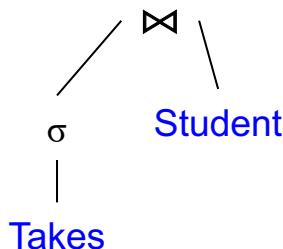
- **Takes_courseID** = index on Takes.courseID
- **Student_ID** = index on Student.ID

Index selection

Index join

```
for y' in Takes_courseID where y'.courseID > 300
  y = fetch the Takes record pointed to by y'
  for x' in Student_ID where x'.ID = y.studentID
    x = fetch the Student record pointed to by x'
```

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



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

Assume the database has indexes on these attributes:

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

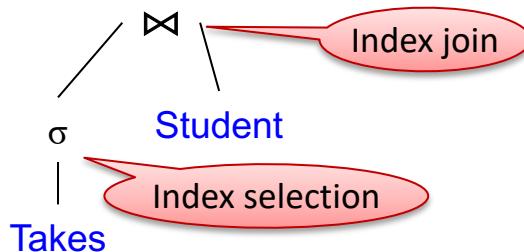
Index selection

Index join

```
for y' in Takes_courseID where y'.courseID > 300
  y = fetch the Takes record pointed to by y'
  for x' in Student_ID where x'.ID = y.studentID
    x = fetch the Student record pointed to by x'
    output *
```

Student(ID, fname, lname)

Takes(studentID, courseID)



SELECT *

FROM Student x, Takes y

WHERE x.ID=y.studentID AND y.courseID > 300

Example

for y in Takes

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

Assume the database has indexes on these attributes:

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

Index selection

for y' in Takes_courseID where y'.courseID > 300

y = fetch the Takes record pointed to by y'

for x' in Student_ID where x'.ID = y.studentID

x = fetch the Student record pointed to by x'

output *

Index join

Getting Practical: Creating Indexes in SQL

```
CREATE TABLE V(M int, N varchar(20), P int);
```

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

Getting Practical: Creating Indexes in SQL

```
CREATE TABLE V(M int, N varchar(20), 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 varchar(20), 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 varchar(20), P int);
```

yes

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

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

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

What does this mean?

Getting Practical: Creating Indexes in SQL

```
CREATE TABLE V(M int, N varchar(20), P int);
```

yes

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

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

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

What does this mean?

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

Getting Practical: Creating Indexes in SQL

```
CREATE TABLE V(M int, N varchar(20), P int);
```

yes

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

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

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

What does this mean?

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

yes

Getting Practical: Creating Indexes in SQL

```
CREATE TABLE V(M int, N varchar(20), P int);
```

yes

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

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

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

What does this mean?

```
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 varchar(20), P int);
```

yes

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

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

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

What does this mean?

```
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 varchar(20), P int);
```

yes

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

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

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

What does this mean?

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

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

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

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

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

yes

no

Not supported
in SQLite

Which Indexes?

ID	fName	IName
10	Tom	Hanks
20	Amy	Hanks
...		

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

Which Indexes?

ID	fName	IName
10	Tom	Hanks
20	Amy	Hanks
...		

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

In general this is a very hard problem

Index Selection: Which Search Key

- Make some attribute K a search key if the WHERE clause contains:
 - An exact match on K
 - A range predicate on K
 - A join on K

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: 1000000 queries: 100000 queries:

SELECT *
FROM V
WHERE N=?

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

INSERT INTO V
VALUES (?, ?, ?)

What indexes ?

The Index Selection Problem 3

$V(M, N, P);$

Your workload is this

100000 queries: 1000000 queries: 100000 queries:

SELECT *
FROM V
WHERE N=?

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

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
- What data structure should be used for index?

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

- Range queries
- What data structure should be used for index?

Basic Index Selection Guidelines

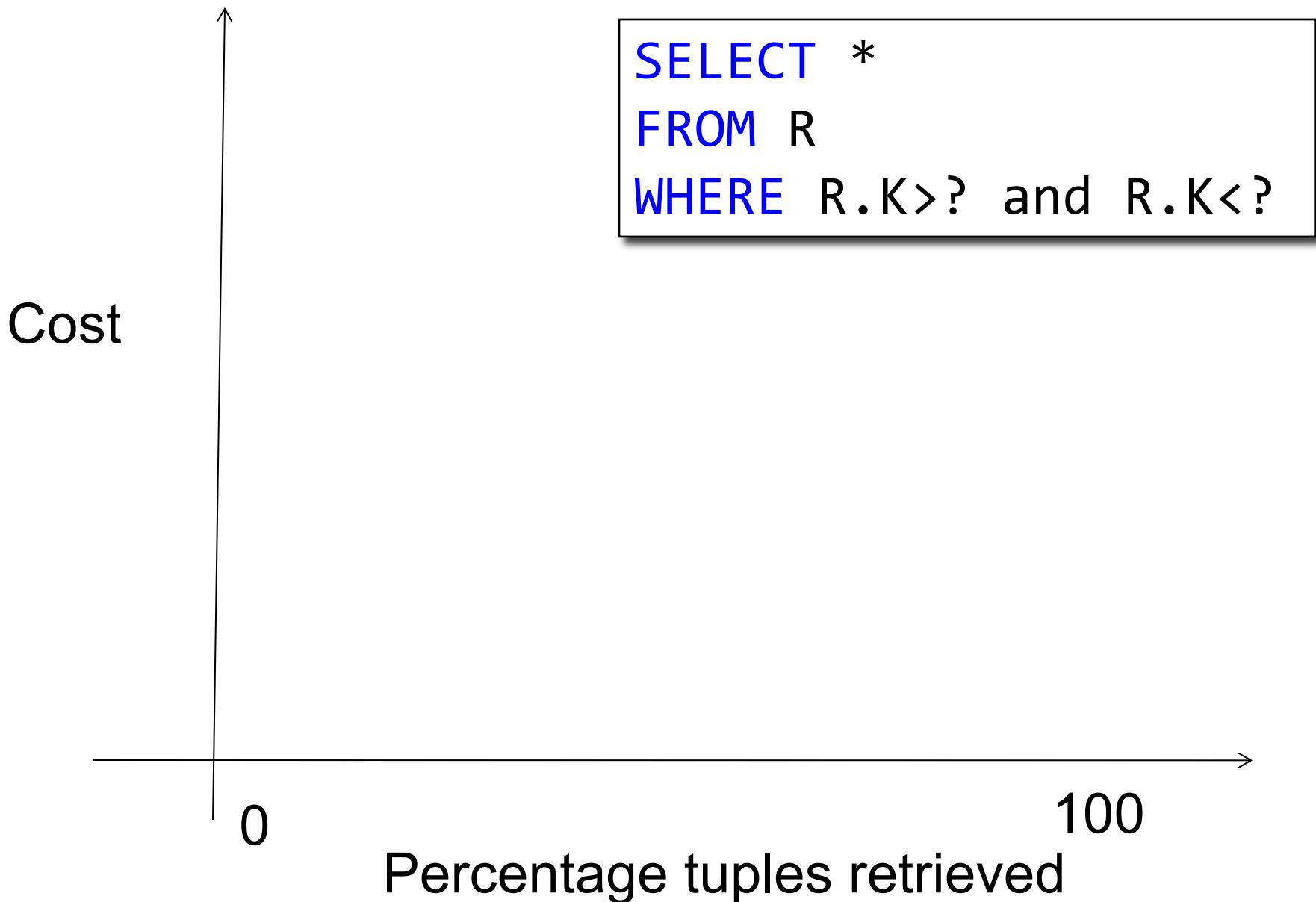
- Consider queries in workload in order of importance
- Consider relations accessed by query
 - No point indexing other relations
- Look at WHERE clause for possible search key
- Try to choose indexes that speed-up multiple queries

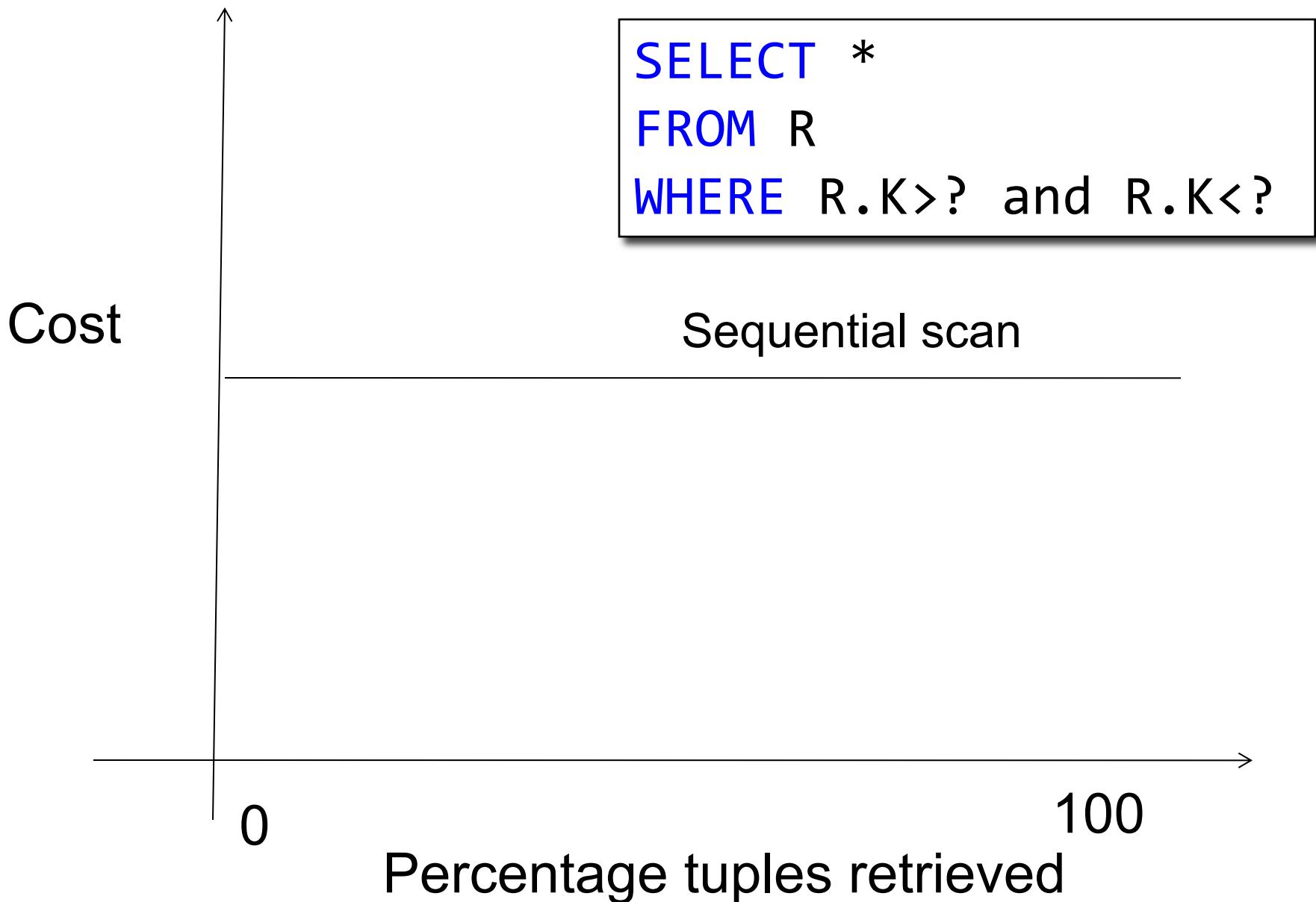
To Cluster or Not

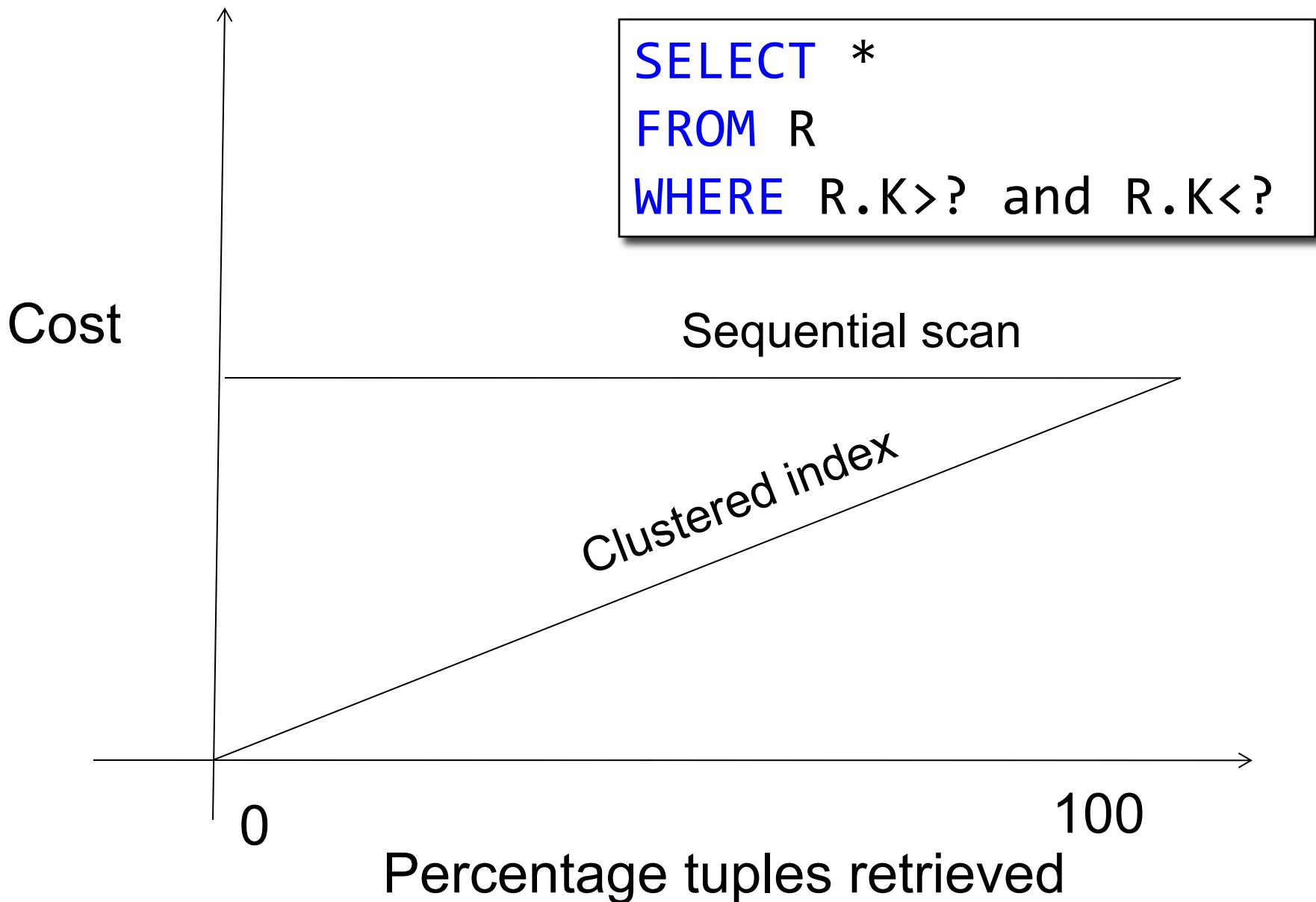
Remember:

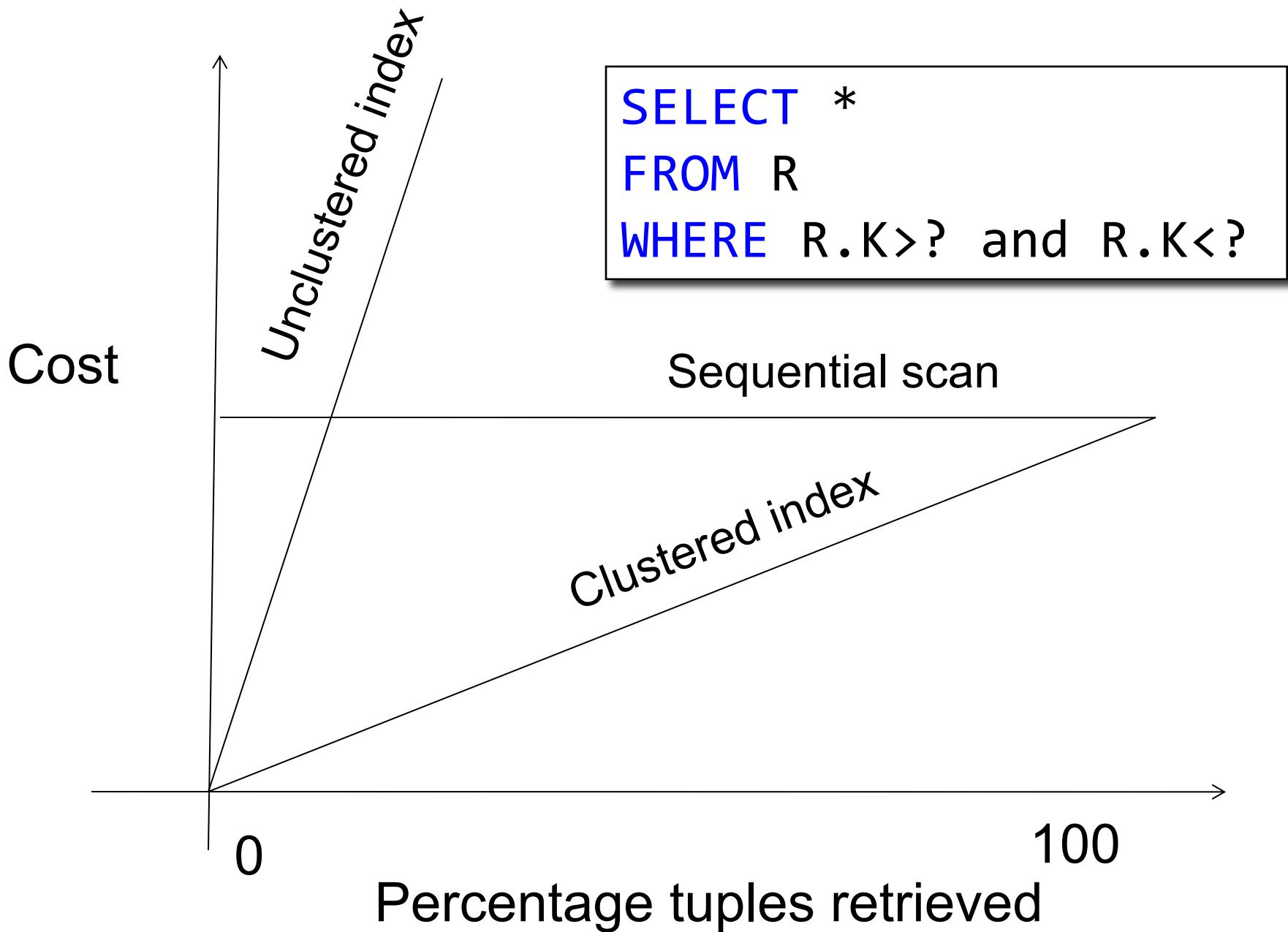
- **Rule of thumb:**
Random reading 1-2% of file \approx sequential scan entire file;

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









Introduction to Database Systems

CSE 344

Lecture 17:
Basics of Query Optimization and
Query Cost Estimation

Cost Estimation

- The optimizer considers several plans, estimates their costs, and chooses the cheapest
- This lecture: cost estimation for relational operators
- The cost is always dominated by the cost of reading from, or writing to disk

Cost of Reading Data From Disk

Cost Parameters

- Cost = I/O + CPU + Network BW
 - We will focus on I/O in this class
- Parameters (a.k.a. statistics):
 - $B(R)$ = # of blocks (i.e., pages) for relation R
 - $T(R)$ = # of tuples in relation R
 - $V(R, a)$ = # of distinct values of attribute a

Cost Parameters

- Cost = I/O + CPU + Network BW
 - We will focus on I/O in this class
- Parameters (a.k.a. statistics):
 - $B(R)$ = # of blocks (i.e., pages) for relation R
 - $T(R)$ = # of tuples in relation R
 - $V(R, a)$ = # of distinct values of attribute a

When a is a key, $V(R,a) = T(R)$

When a is not a key, $V(R,a)$ can be anything $\leq T(R)$

Cost Parameters

- Cost = I/O + CPU + Network BW
 - We will focus on I/O in this class
- Parameters (a.k.a. statistics):
 - $B(R)$ = # of blocks (i.e., pages) for relation R
 - $T(R)$ = # of tuples in relation R
 - $V(R, a)$ = # of distinct values of attribute a

When a is a key, $V(R,a) = T(R)$

When a is not a key, $V(R,a)$ can be anything $\leq T(R)$

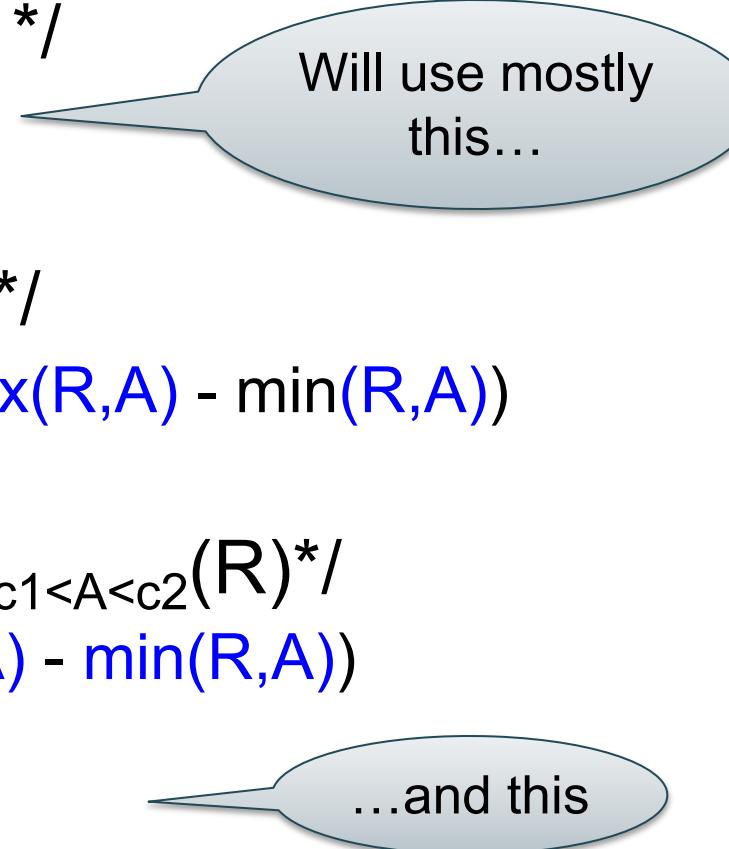
- DBMS collects **statistics** about base tables
must infer them for intermediate results

Size Estimation

Main principle:

- Size of the output = some *fraction* of the size of the input
- The *fraction* is called the *selectivity factor*

Selectivity Factors for Conditions

- $A = c$ /* $\sigma_{A=c}(R)$ */
 - Selectivity $f = 1/V(R,A)$
 - $A < c$ /* $\sigma_{A < c}(R)$ */
 - Selectivity $f = (c - \min(R, A)) / (\max(R, A) - \min(R, A))$
 - $c_1 < A < c_2$ /* $\sigma_{c_1 < A < c_2}(R)$ */
 - Selectivity $f = (c_2 - c_1) / (\max(R, A) - \min(R, A))$
 - $\text{Cond1} \wedge \text{Cond2} \wedge \text{Cond3} \wedge \dots$
 - Selectivity = $f_1 * f_2 * f_3 * \dots$ (^{174A-21Wi} assumes independence)
- 

Cost of Reading Data From Disk

- Sequential scan for relation R costs **B(R)**
- Index-based selection
 - Estimate selectivity factor f (see previous slide)
 - Clustered index: $f^* \mathbf{B(R)}$
 - Unclustered index $f^* \mathbf{T(R)}$

Note: we ignore I/O cost for index pages

Index Based Selection

- Example:

$B(R) = 2000$
 $T(R) = 100,000$
 $V(R, a) = 20$
- Table scan:
- Index based selection:

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

Index Based Selection

- Example:

$$\begin{aligned}B(R) &= 2000 \\T(R) &= 100,000 \\V(R, a) &= 20\end{aligned}$$

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

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

Index Based Selection

- Example:

$$\begin{aligned}B(R) &= 2000 \\T(R) &= 100,000 \\V(R, a) &= 20\end{aligned}$$

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

- Table scan: $B(R) = 2,000$ I/Os
- Index based selection:
 - If index is clustered:
 - If index is unclustered:

Index Based Selection

- Example:

$B(R) = 2000$
$T(R) = 100,000$
$V(R, a) = 20$
- Table scan: $B(R) = 2,000$ I/Os
- Index based selection:
 - If index is clustered: $B(R) * 1/V(R,a) = 100$ I/Os
 - If index is unclustered:

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

Index Based Selection

- Example:

$B(R) = 2000$
$T(R) = 100,000$
$V(R, a) = 20$
- Table scan: $B(R) = 2,000$ I/Os
- Index based selection:
 - If index is clustered: $B(R) * 1/V(R,a) = 100$ I/Os
 - If index is unclustered: $T(R) * 1/V(R,a) = 5,000$ I/Os

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

Index Based Selection

- Example:

$B(R) = 2000$
$T(R) = 100,000$
$V(R, a) = 20$
- Table scan: $B(R) = 2,000$ I/Os
- Index based selection:
 - If index is clustered: $B(R) * 1/V(R,a) = 100$ I/Os
 - If index is unclustered: $T(R) * 1/V(R,a) = 5,000$ I/Os

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

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

Cost of Executing Operators (Focus on Joins)

Outline

- **Join operator algorithms**
 - One-pass algorithms (Sec. 15.2 and 15.3)
 - Index-based algorithms (Sec 15.6)
- Note about readings:
 - In class, we discuss only algorithms for joins
 - Other operators are easier: read the book

Join Algorithms

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

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 $R \bowtie S$
- R is the outer relation, S is the inner relation

```
for each tuple  $t_1$  in R do  
  for each tuple  $t_2$  in S do  
    if  $t_1$  and  $t_2$  join then output  $(t_1, t_2)$ 
```

What is the Cost?

Nested Loop Joins

- Tuple-based nested loop $R \bowtie S$
- R is the outer relation, S is the inner relation

```
for each tuple  $t_1$  in R do  
  for each tuple  $t_2$  in S do  
    if  $t_1$  and  $t_2$  join then output  $(t_1, t_2)$ 
```

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

What is the Cost?

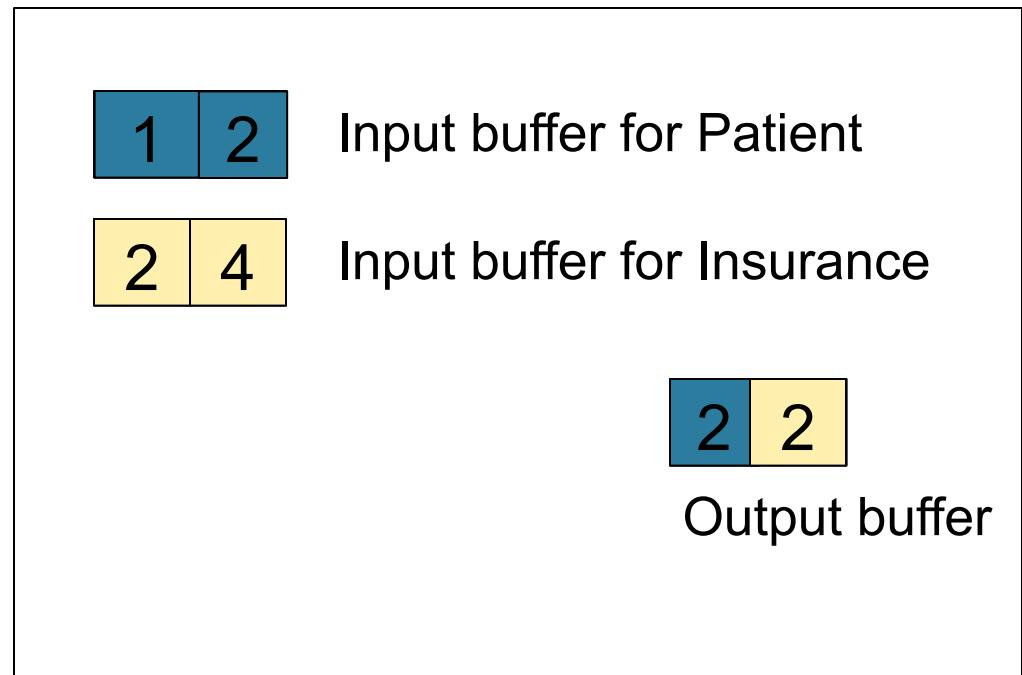
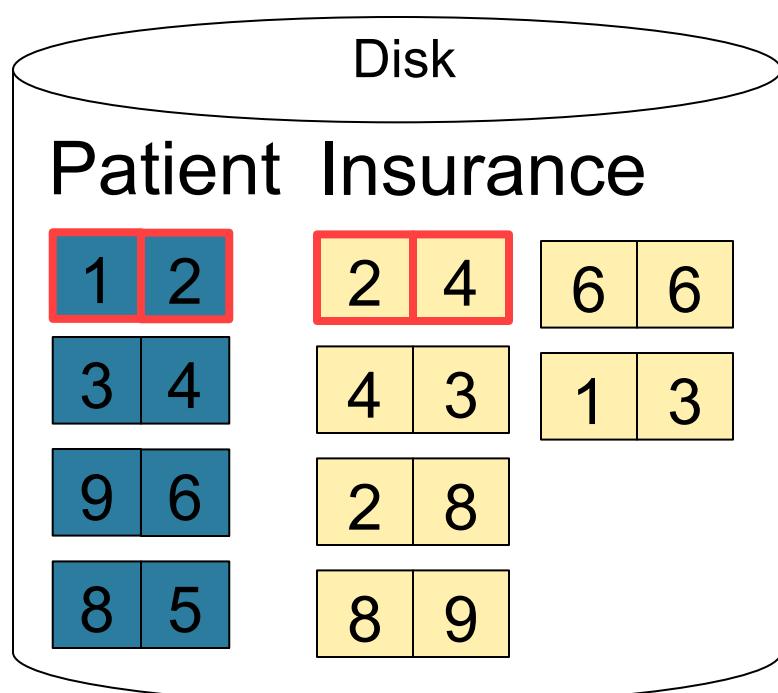
Page-at-a-time Refinement

```
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)
```

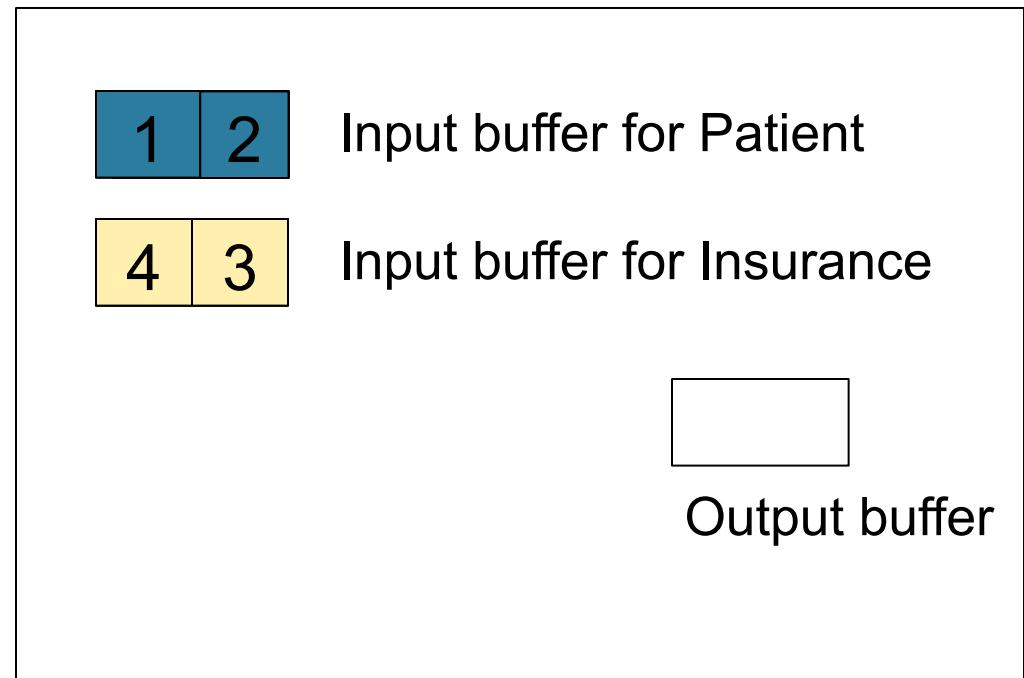
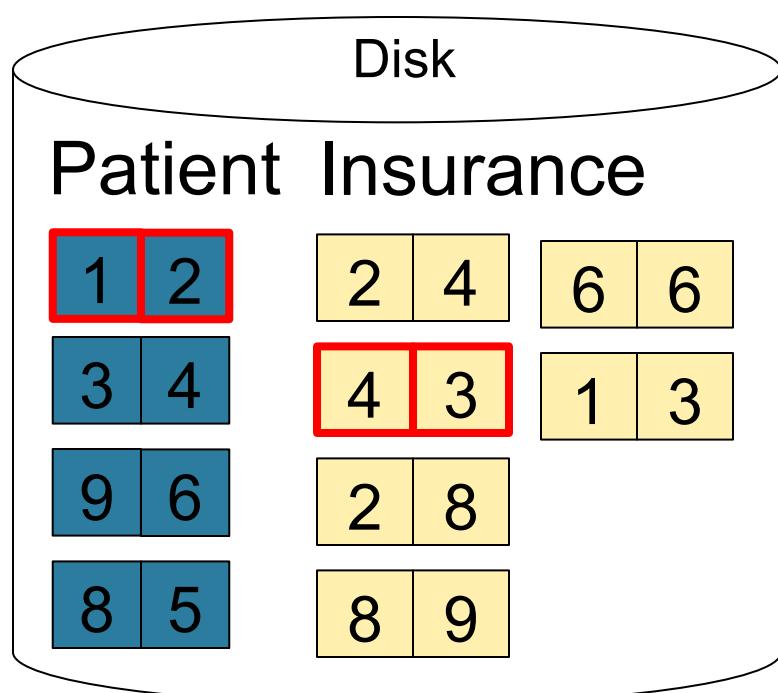
- Cost: $B(R) + B(R)B(S)$

What is the Cost?

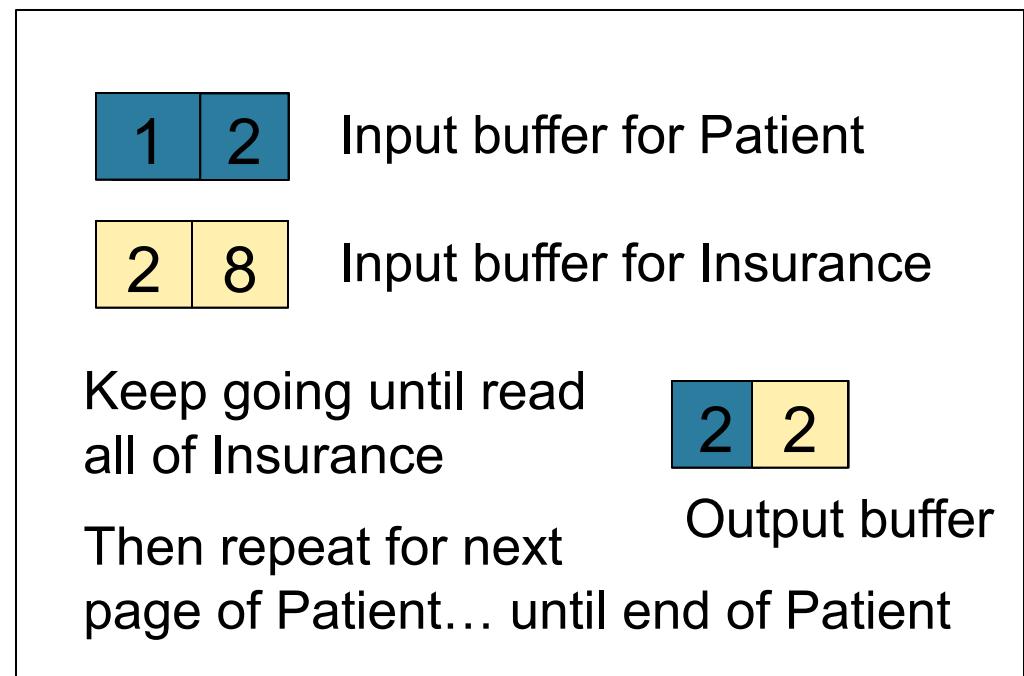
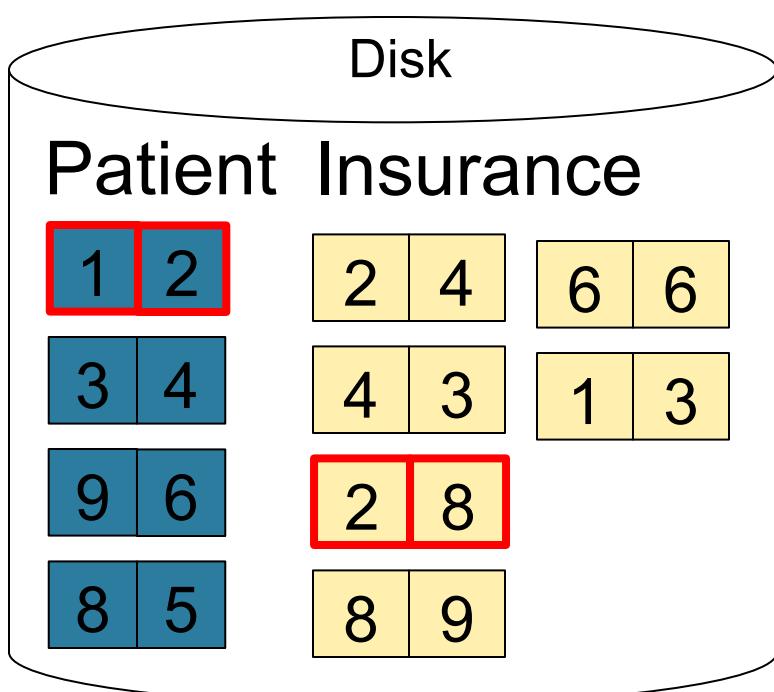
Page-at-a-time Refinement



Page-at-a-time Refinement



Page-at-a-time Refinement



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: $R \bowtie S$

- 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?

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 $B(R) \leq M$
 - M = number of memory pages available

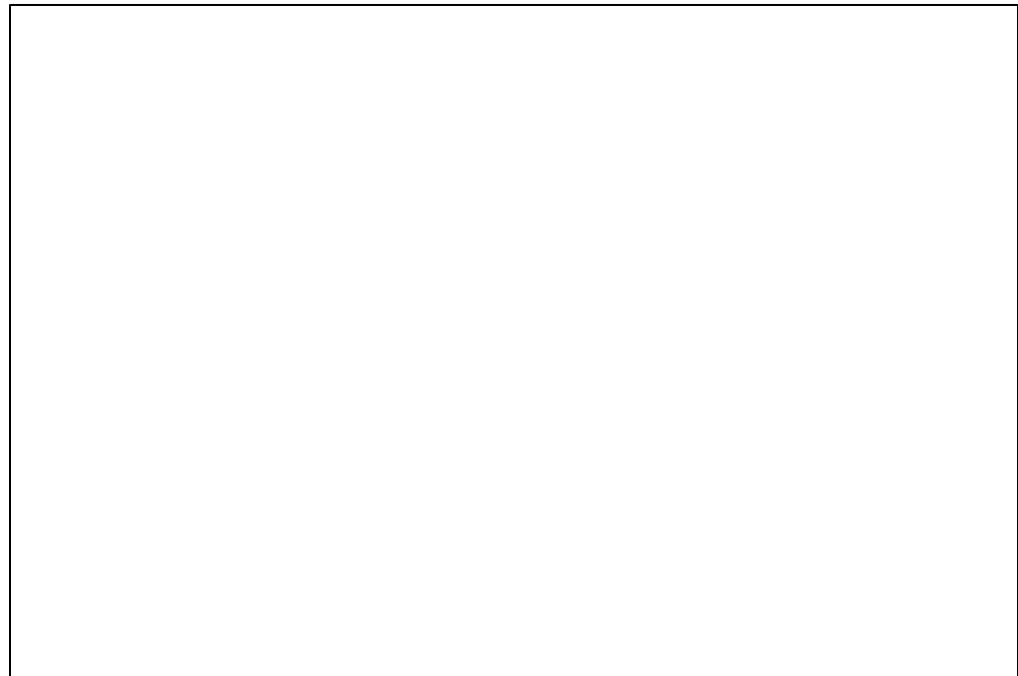
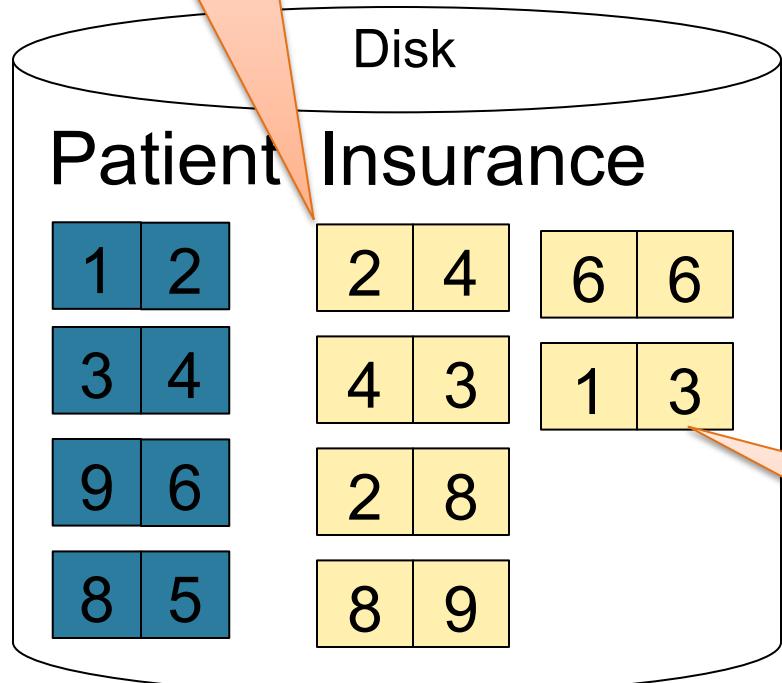
Hash Join Example

Patient \bowtie Insurance

Some large-enough #

Memory M = 21 pages

Showing pid only

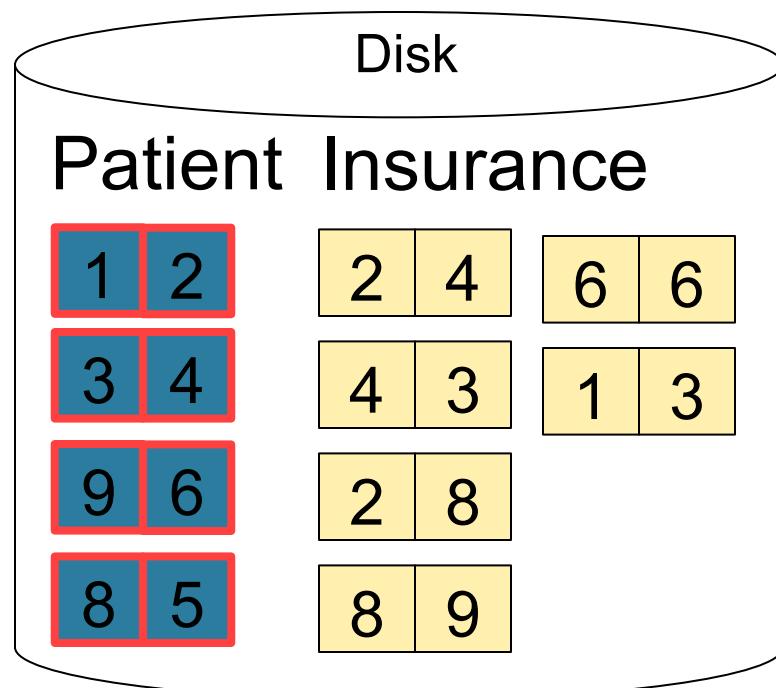


This is one page
with two tuples
174A - 21Wi

Hash Join Example

Step 1: Scan Patient and **build** hash table in memory

Can be done in
method open()



Memory M = 21 pages

Hash h: pid % 5

5		1	6	2		3	8	4	9
---	--	---	---	---	--	---	---	---	---

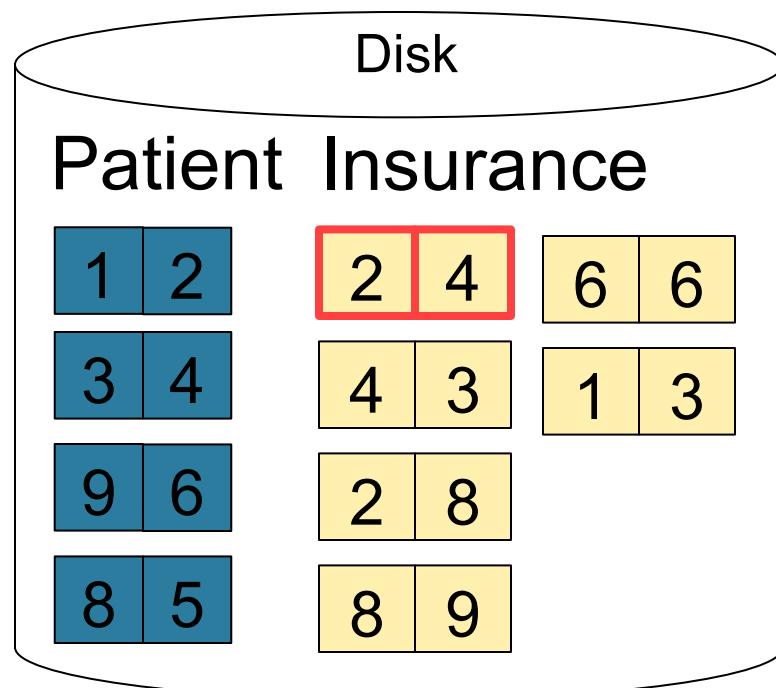


Input buffer

Hash Join Example

Step 2: Scan Insurance and **probe** into hash table

Done during
calls to next()



Memory M = 21 pages

Hash h: pid % 5

5		1	6	2		3	8	4	9
---	--	---	---	---	--	---	---	---	---

2	4
---	---

Input buffer

2	2
---	---

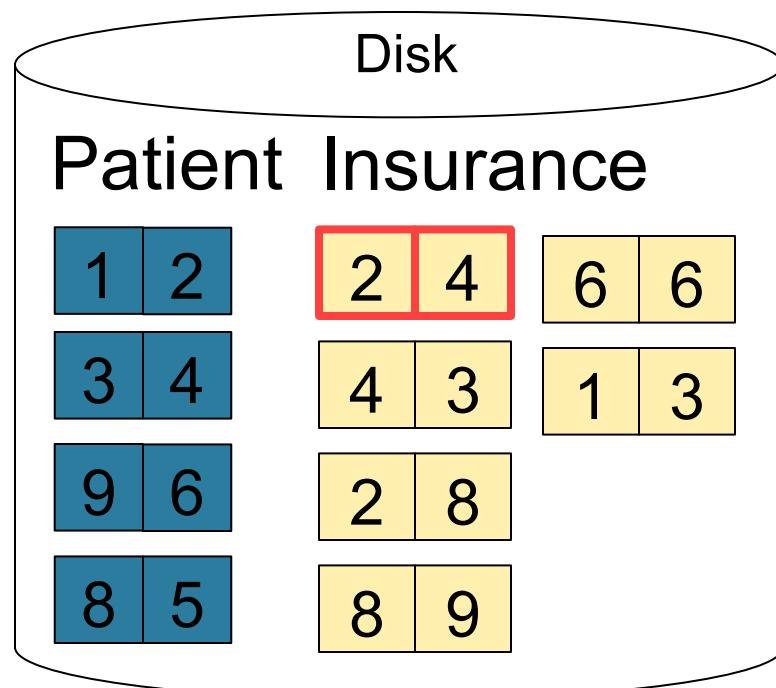
Output buffer

Write to disk or
pass to next
operator

Hash Join Example

Step 2: Scan Insurance and **probe** into hash table

Done during
calls to next()



Memory M = 21 pages

Hash h: pid % 5

5		1	6	2		3	8	4	9
---	--	---	---	---	--	---	---	---	---

2	4
---	---

Input buffer

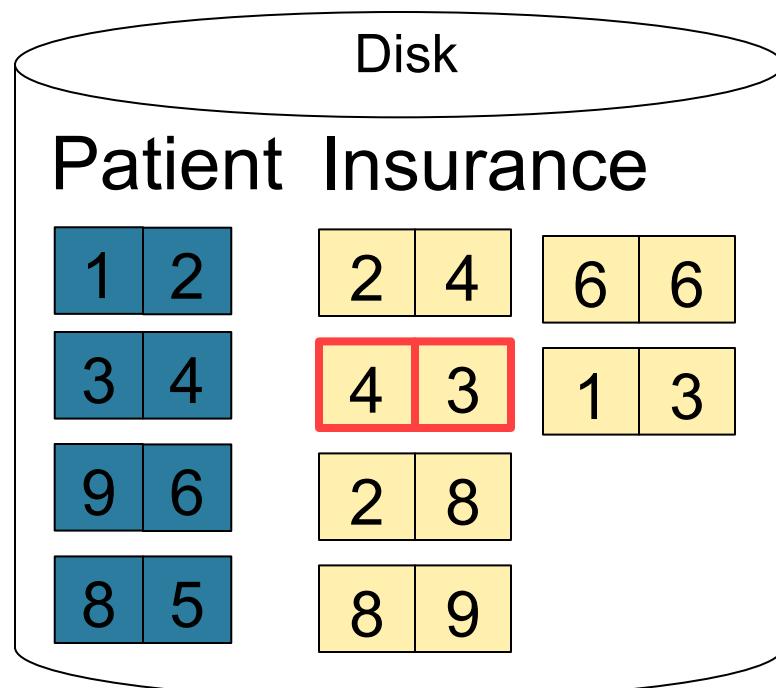
4	4
---	---

Output buffer

Hash Join Example

Step 2: Scan Insurance and **probe** into hash table

Done during
calls to next()



Memory M = 21 pages

Hash h: pid % 5

5		1	6	2		3	8	4	9
---	--	---	---	---	--	---	---	---	---

4	3
---	---

Input buffer

4	4
---	---

Output buffer

Keep going until read all of Insurance

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

174A - 21Wi

Sort-Merge Join

Sort-merge join: $R \bowtie S$

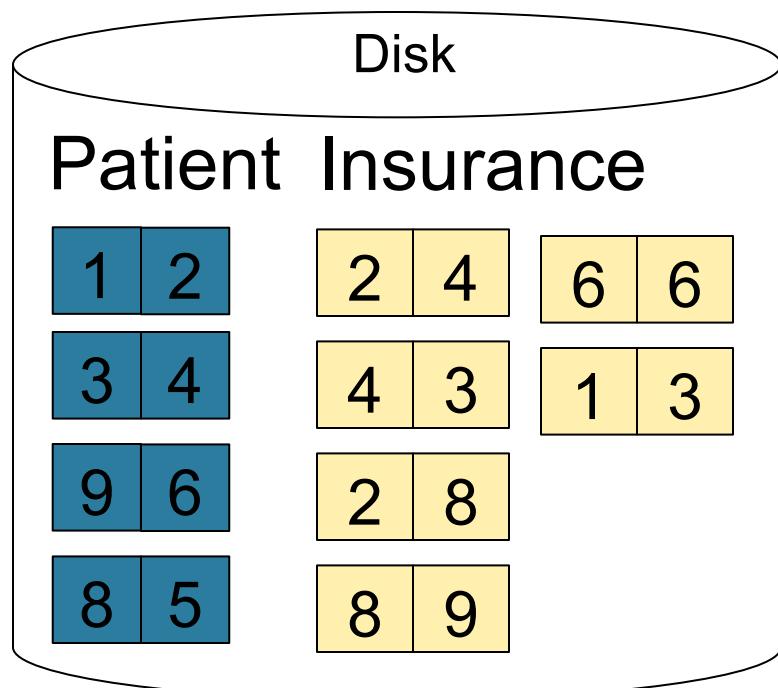
- Scan R and sort in main memory
 - Scan S and sort in main memory
 - Merge R and S
-
- Cost: $B(R) + B(S)$
 - One pass algorithm when $B(S) + B(R) \leq M$
 - Typically, this is NOT a one pass algorithm

Sort-Merge Join Example

Step 1: Scan Patient and **sort** in memory

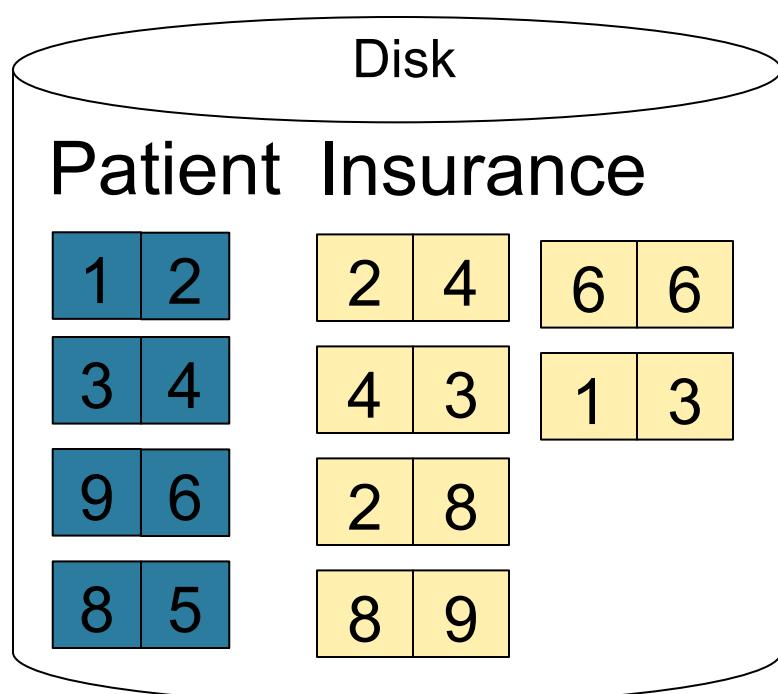
Memory M = 21 pages

1	2	3	4	5	6	8	9
---	---	---	---	---	---	---	---



Sort-Merge Join Example

Step 2: Scan Insurance and **sort** in memory

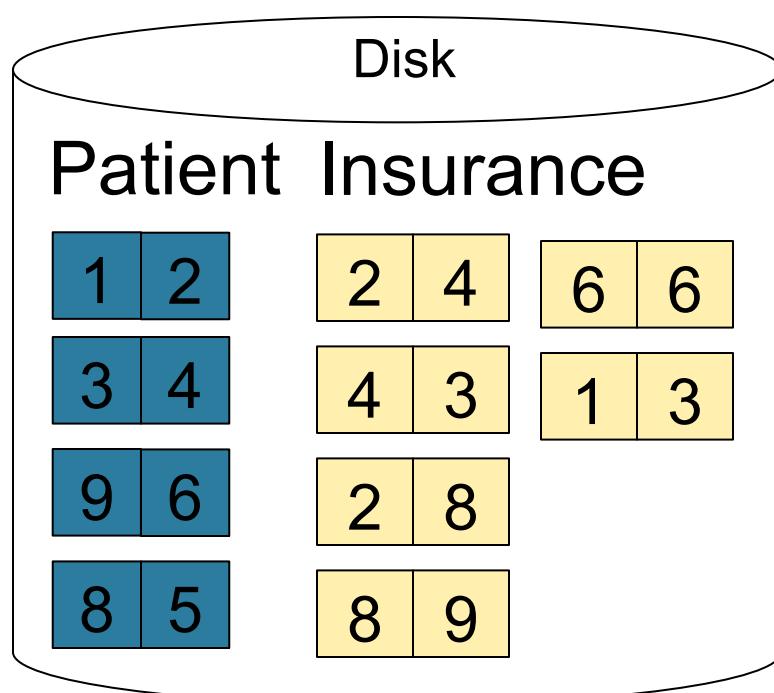


Memory M = 21 pages

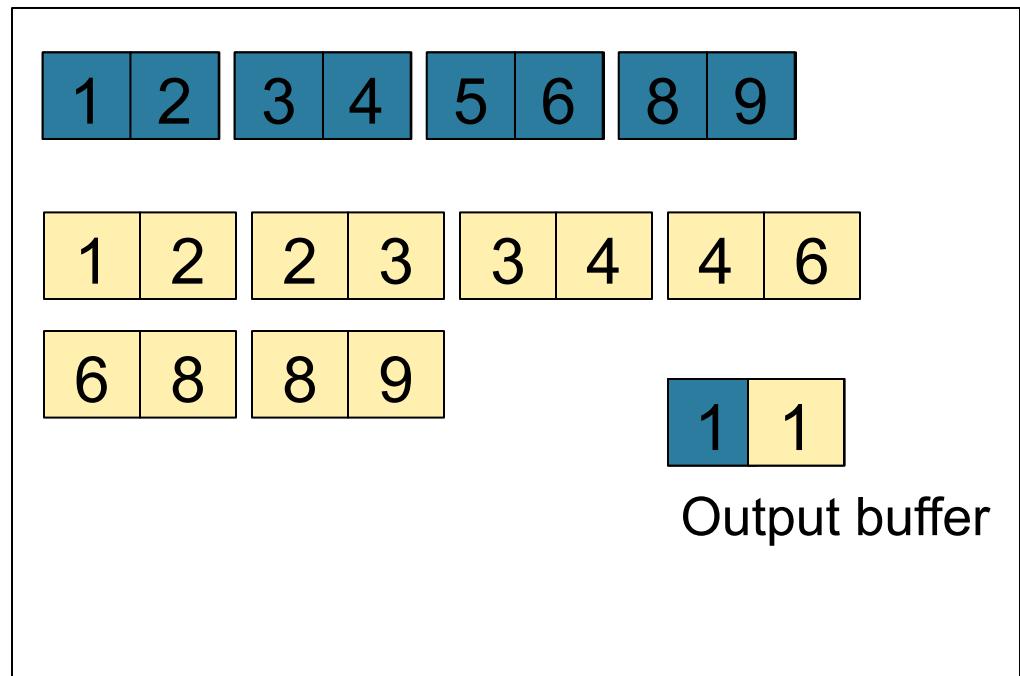
1	2	3	4	5	6	8	9
1	2	2	3	3	4	4	6
6		8		9			

Sort-Merge Join Example

Step 3: Merge Patient and Insurance

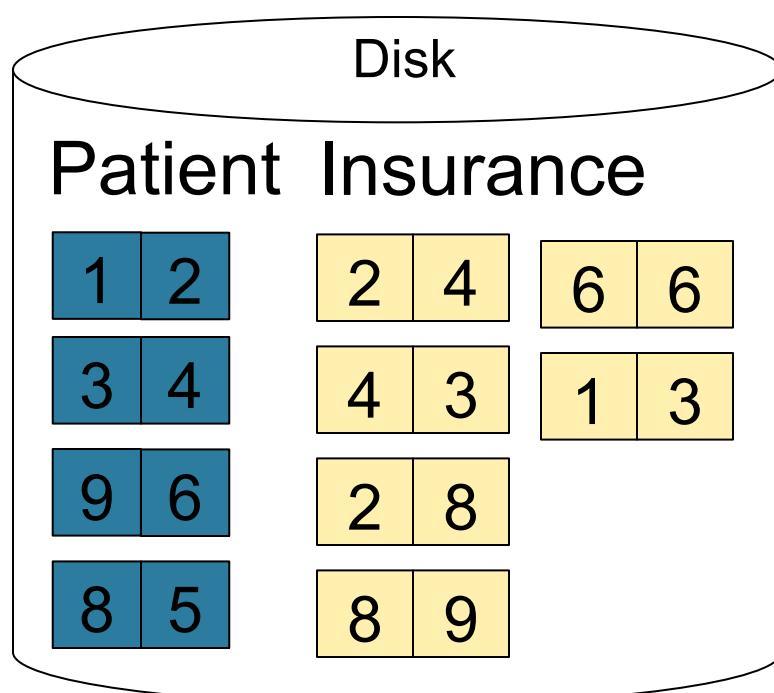


Memory M = 21 pages

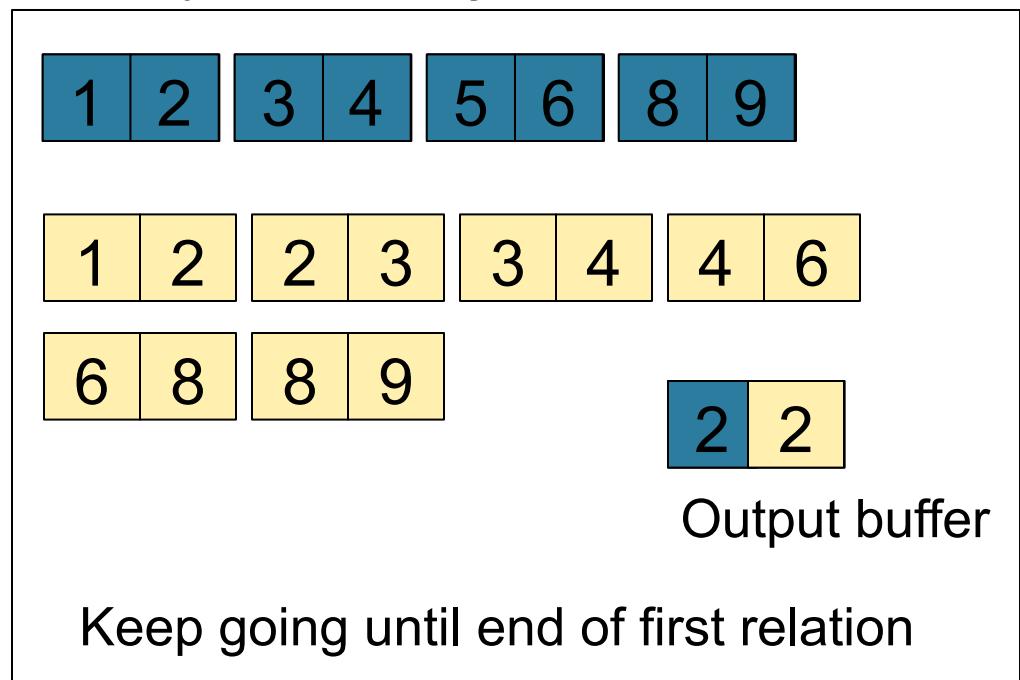


Sort-Merge Join Example

Step 3: Merge Patient and Insurance



Memory M = 21 pages



Index Join

$R \bowtie S$

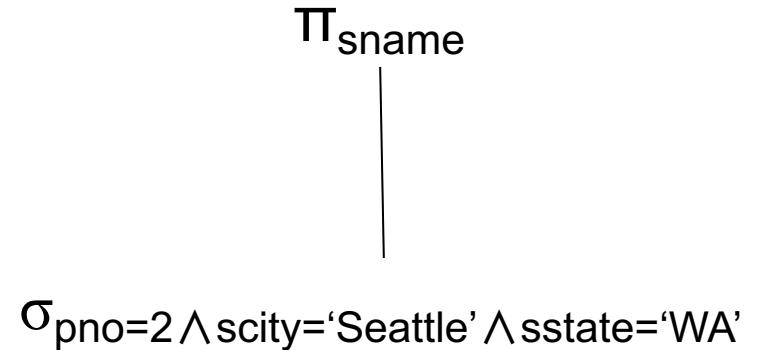
- Assume S has an index on the join attribute
- Iterate over R, for each tuple fetch corresponding tuple(s) from S
- Cost:
 - If index on S is clustered:
 $B(R) + T(R) * (B(S) * 1/V(S,a))$
 - If index on S is unclustered:
 $B(R) + T(R) * (T(S) * 1/V(S,a))$

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'
```

Supply

Supplier

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

174A
T(Supplier) = 1000
B(Supplier) = 100
V(Supplier, scity) = 20
V(Supplier, state) = 10

M=11

Supplier(sid, sname, scity, sstate)

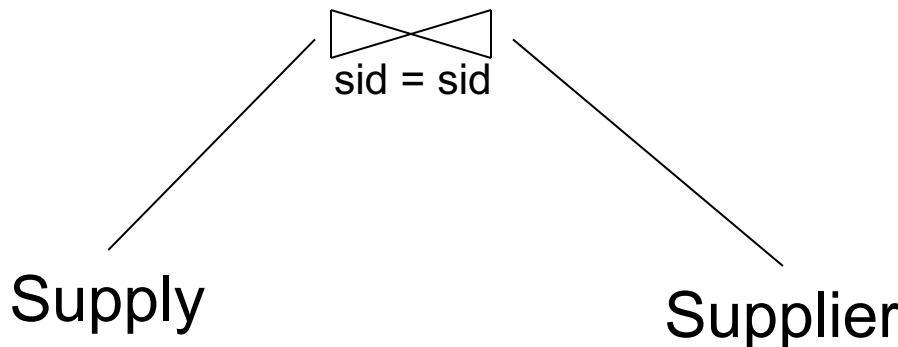
Supply(sid, pno, quantity)

Logical Query Plan 1

Π_{sname}

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

$T = 10000$



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

$T(\text{Supply}) = 10000$
 $B(\text{Supply}) = 100$
 $V(\text{Supply}, \text{pno}) = 2500$

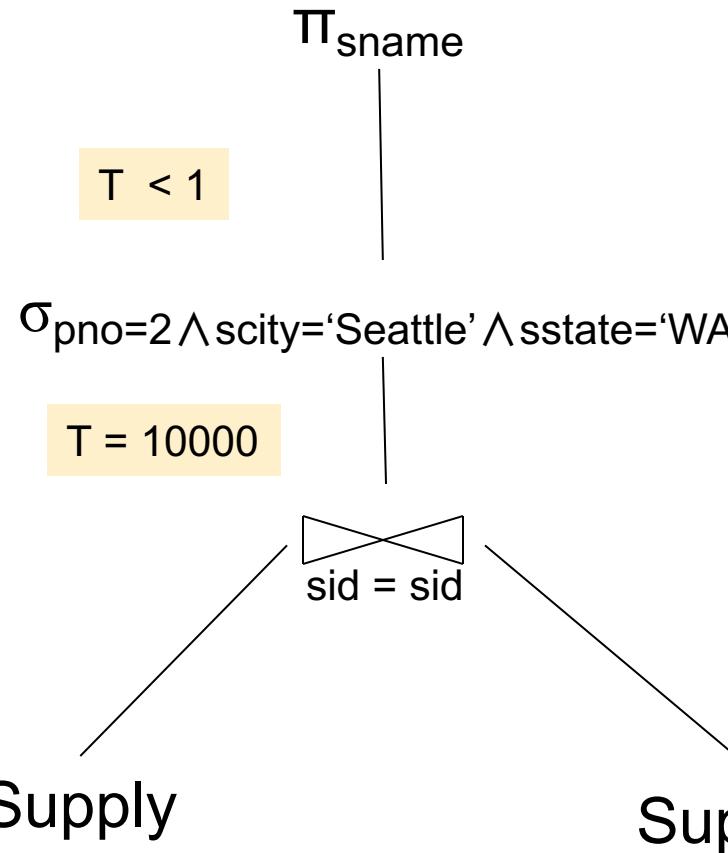
$T(\text{Supplier}) = 1000$
 $B(\text{Supplier}) = 100$
 $V(\text{Supplier}, \text{scity}) = 20$
174A
 $V(\text{Supplier}, \text{state}) = 10$

M=11

Supplier(sid, sname, scity, sstate)

Supply(sid, pno, quantity)

Logical Query Plan 1



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

$T(\text{Supply}) = 10000$
 $B(\text{Supply}) = 100$
 $V(\text{Supply}, \text{pno}) = 2500$

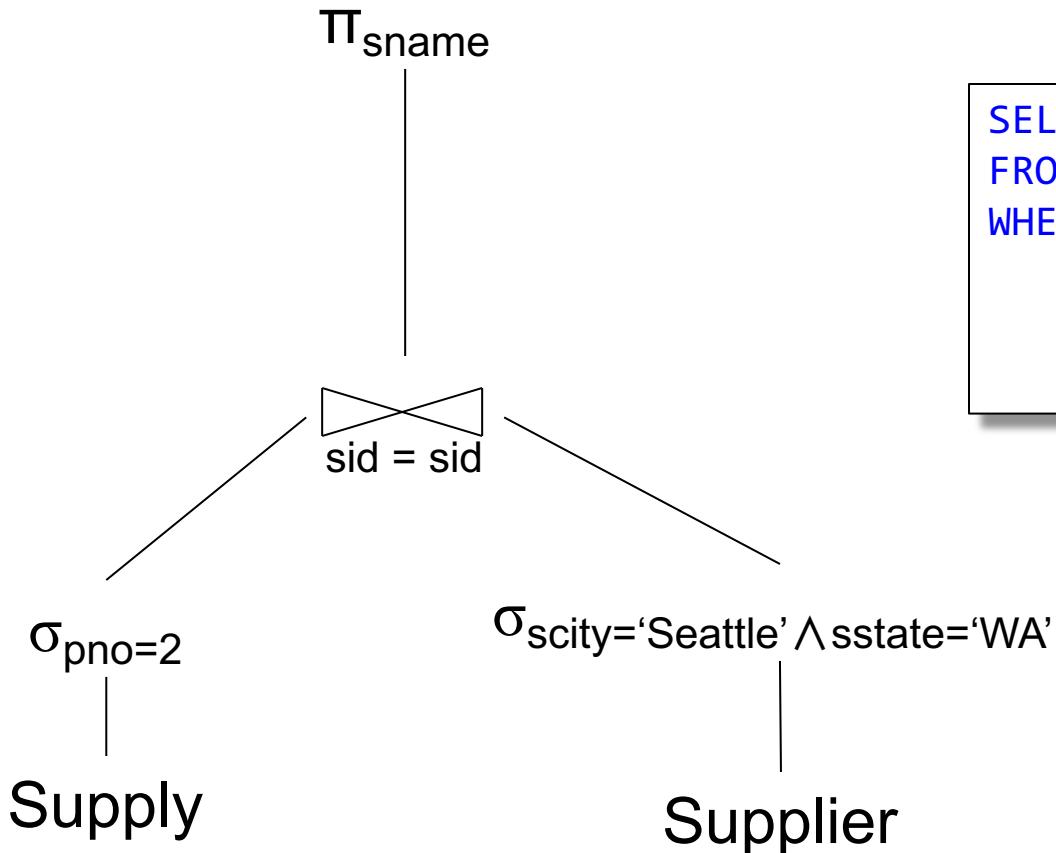
$T(\text{Supplier}) = 1000$
 $B(\text{Supplier}) = 100$
 $V(\text{Supplier}, \text{scity}) = 20$
 $V(\text{Supplier}, \text{sstate}) = 10$

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

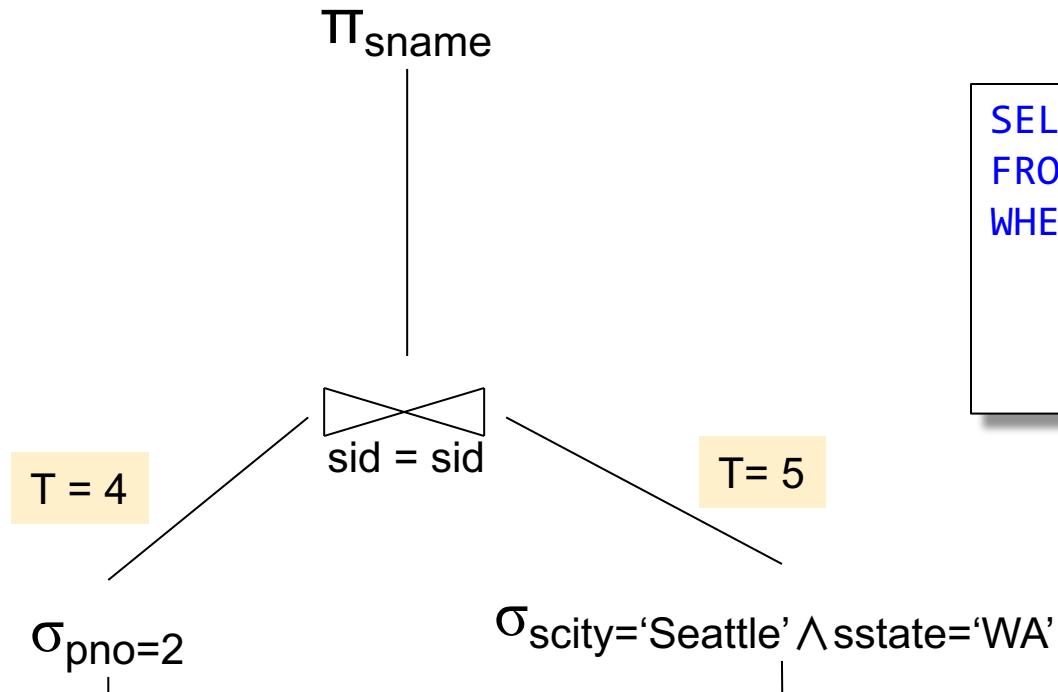
T(Supplier) = 1000
B(Supplier) = 100
V(Supplier, scity) = 20
174A-2TW
V(Supplier, state) = 10

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

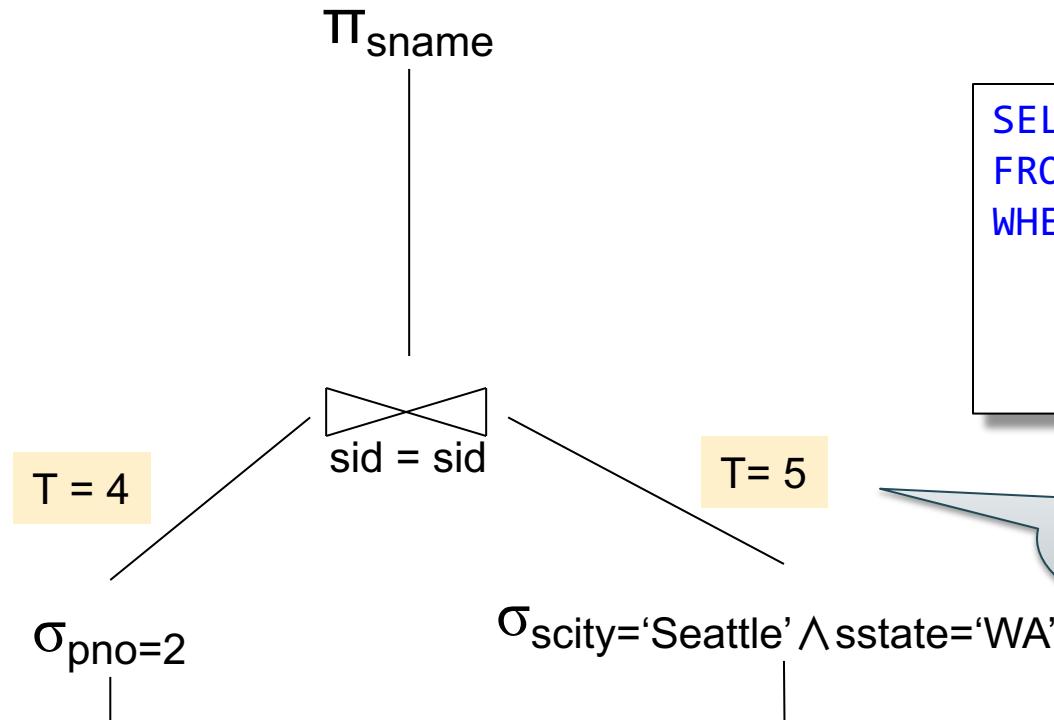
T(Supplier) = 1000
B(Supplier) = 100
V(Supplier, scity) = 20
174A-2TW
V(Supplier, state) = 10

M=11

Supplier(sid, sname, scity, sstate)

Supply(sid, pno, quantity)

Logical Query Plan 2



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

Very wrong!
Why?

$T(\text{Supply}) = 10000$
 $B(\text{Supply}) = 100$
 $V(\text{Supply}, \text{pno}) = 2500$

$T(\text{Supplier}) = 1000$
 $B(\text{Supplier}) = 100$
 $V(\text{Supplier}, \text{scity}) = 20$
 $V(\text{Supplier}, \text{state}) = 10$

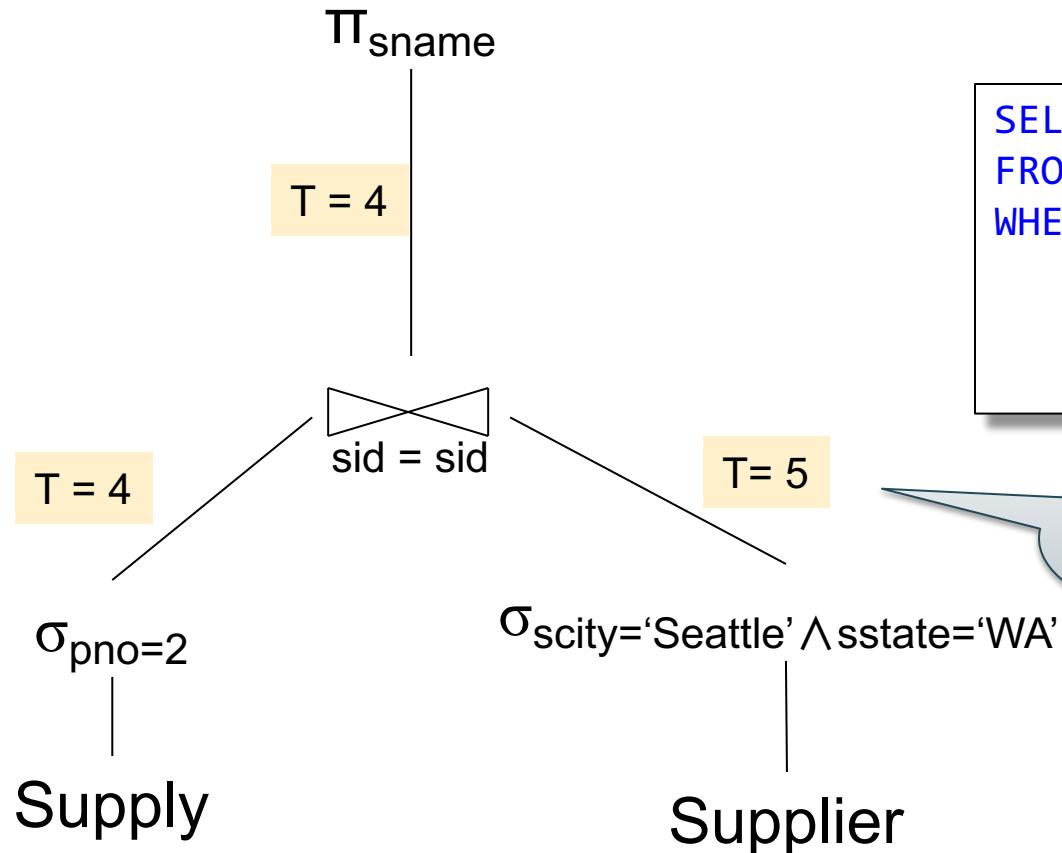
174A-2TW

M=11

Supplier(sid, sname, scity, sstate)

Supply(sid, pno, quantity)

Logical Query Plan 2



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

Very wrong!
Why?

$T(\text{Supply}) = 10000$
 $B(\text{Supply}) = 100$
 $V(\text{Supply}, \text{pno}) = 2500$

$T(\text{Supplier}) = 1000$
 $B(\text{Supplier}) = 100$
 $V(\text{Supplier}, \text{scity}) = 20$
 $V(\text{Supplier}, \text{sstate}) = 10$

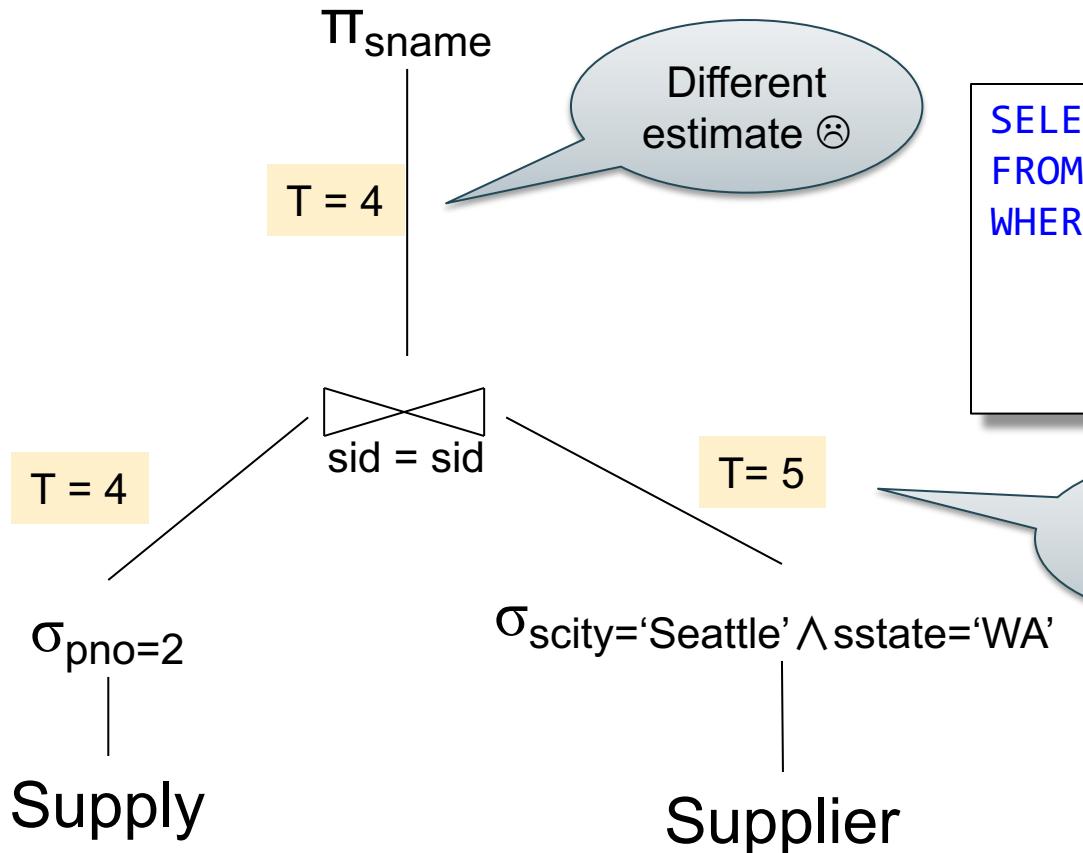
174A-2TW

M=11

Supplier(sid, sname, scity, sstate)

Supply(sid, pno, quantity)

Logical Query Plan 2



$T(\text{Supply}) = 10000$
 $B(\text{Supply}) = 100$
 $V(\text{Supply}, pno) = 2500$

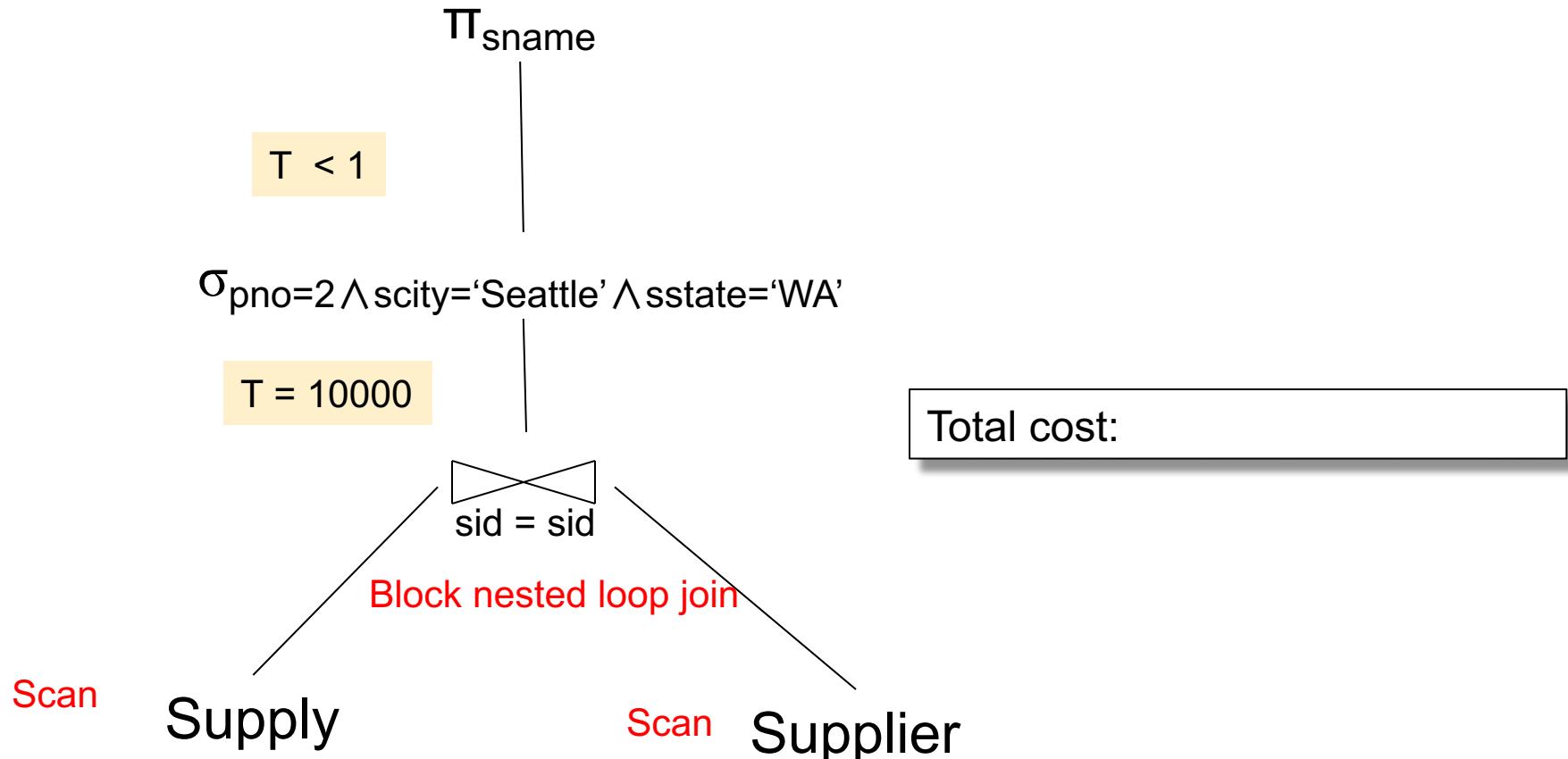
$T(\text{Supplier}) = 1000$
 $B(\text{Supplier}) = 100$
 $V(\text{Supplier}, scity) = 20$
174A
 $V(\text{Supplier}, state) = 10$

M=11

Supplier(sid, sname, scity, sstate)

Supply(sid, pno, quantity)

Physical Plan 1



$T(\text{Supply}) = 10000$
 $B(\text{Supply}) = 100$
 $V(\text{Supply}, \text{pno}) = 2500$

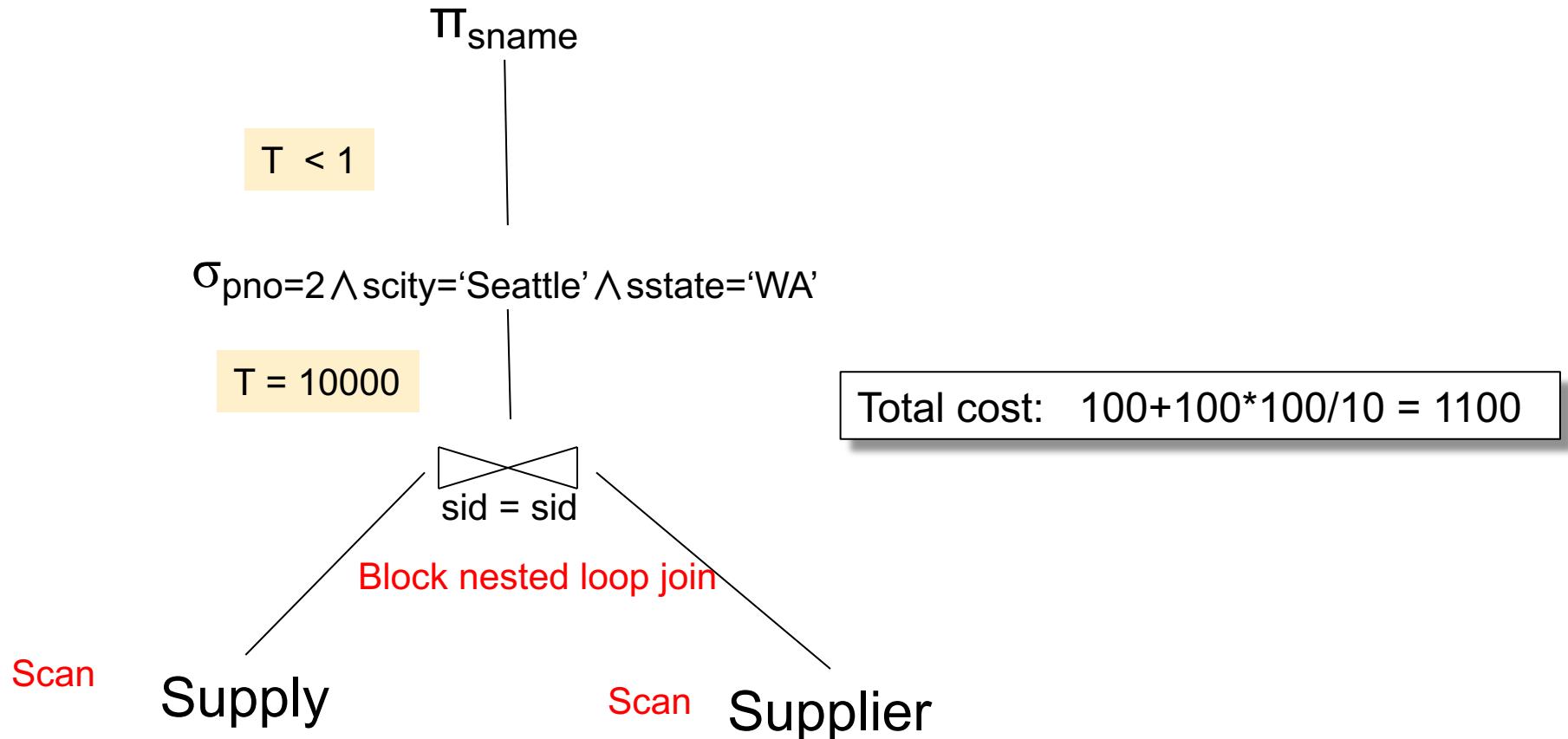
$T(\text{Supplier}) = 1000$
 $B(\text{Supplier}) = 100$
 $V(\text{Supplier}, \text{scity}) = 20$
174A
 $V(\text{Supplier}, \text{state}) = 10$

M=11

Supplier(sid, sname, scity, sstate)

Supply(sid, pno, quantity)

Physical Plan 1



$T(\text{Supply}) = 10000$
 $B(\text{Supply}) = 100$
 $V(\text{Supply}, \text{pno}) = 2500$

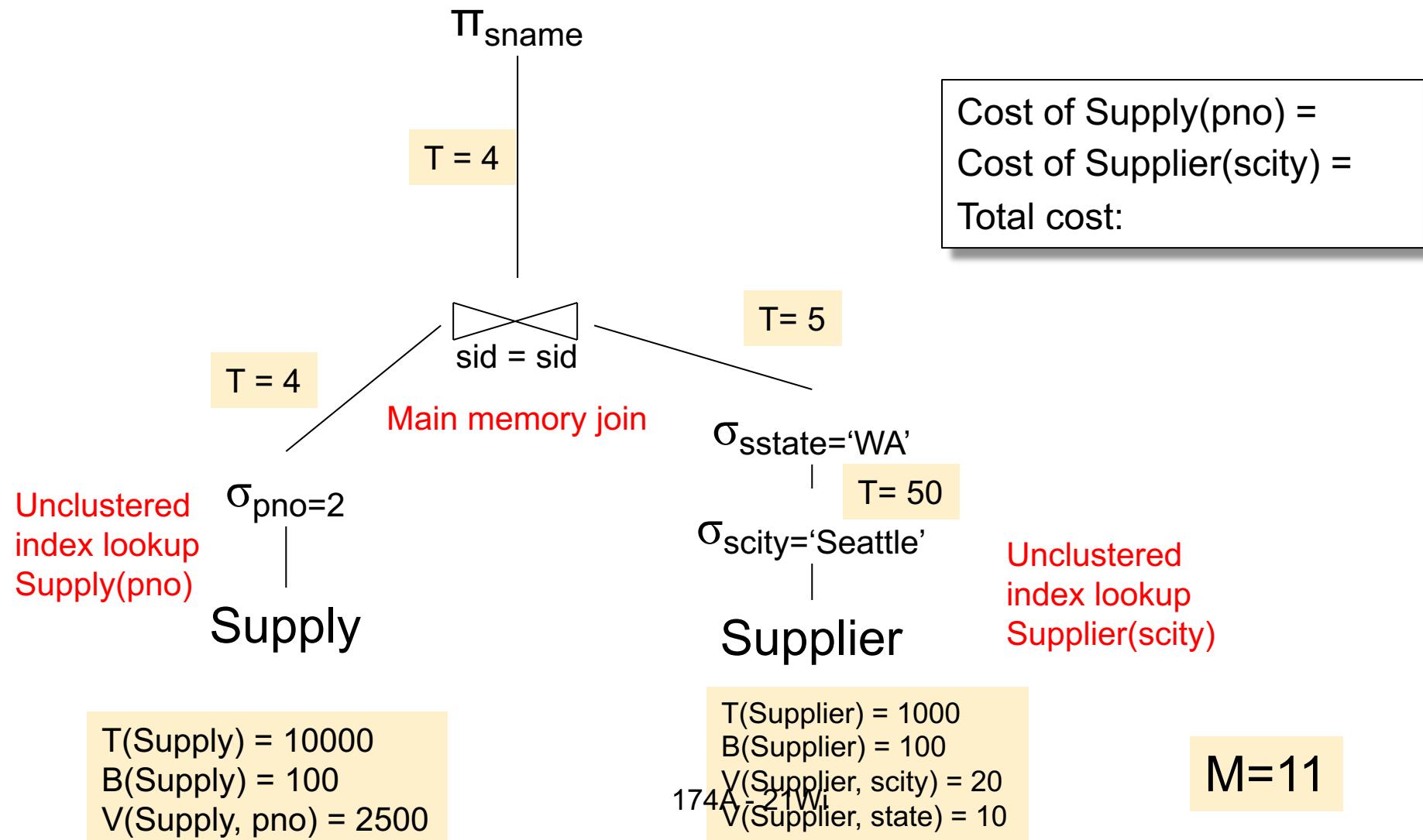
$T(\text{Supplier}) = 100$
 $B(\text{Supplier}) = 100$
 $V(\text{Supplier}, \text{scity}) = 20$
 $V(\text{Supplier}, \text{sstate}) = 10$

M=11

Supplier(sid, sname, scity, sstate)

Supply(sid, pno, quantity)

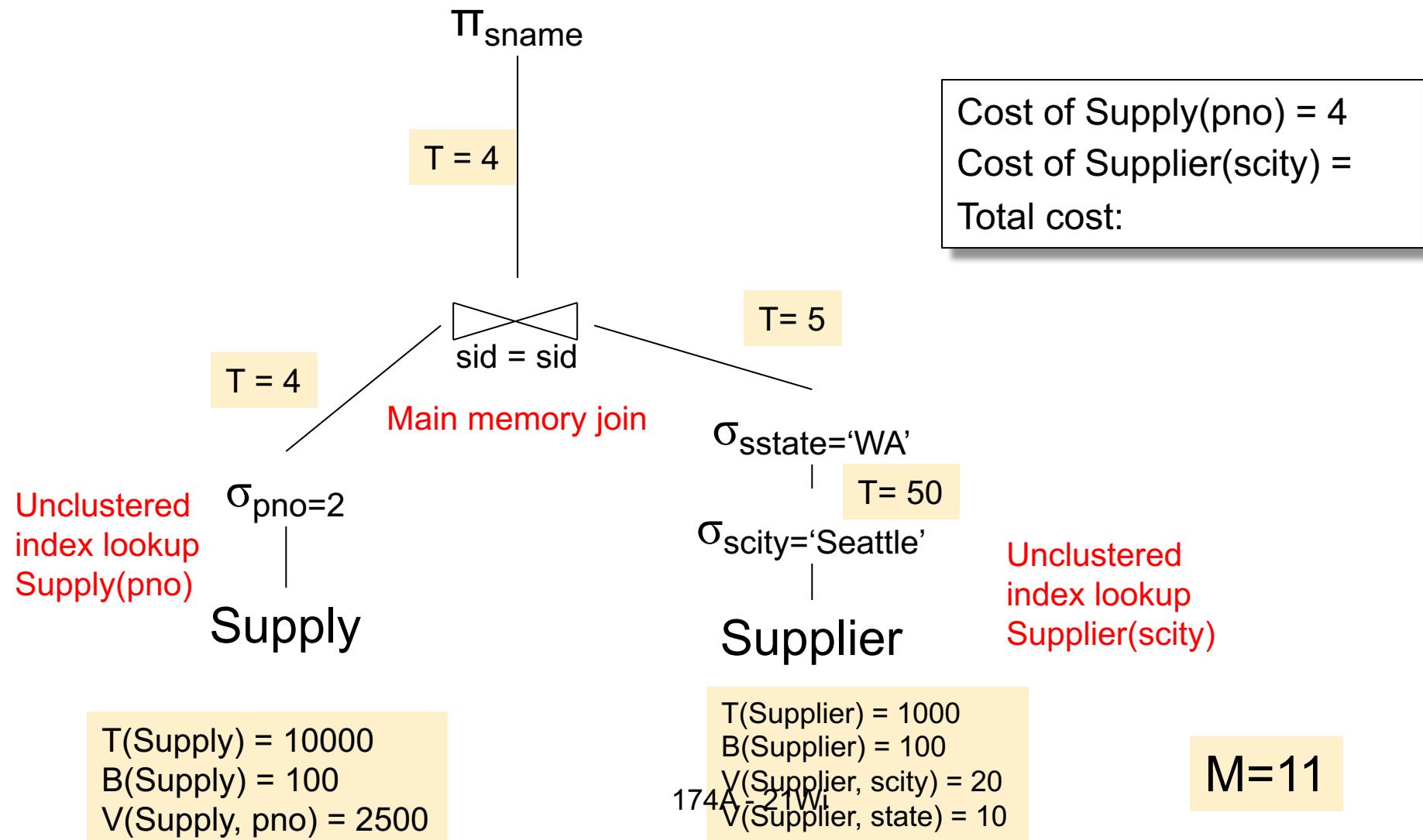
Physical Plan 2



Supplier(sid, sname, scity, sstate)

Supply(sid, pno, quantity)

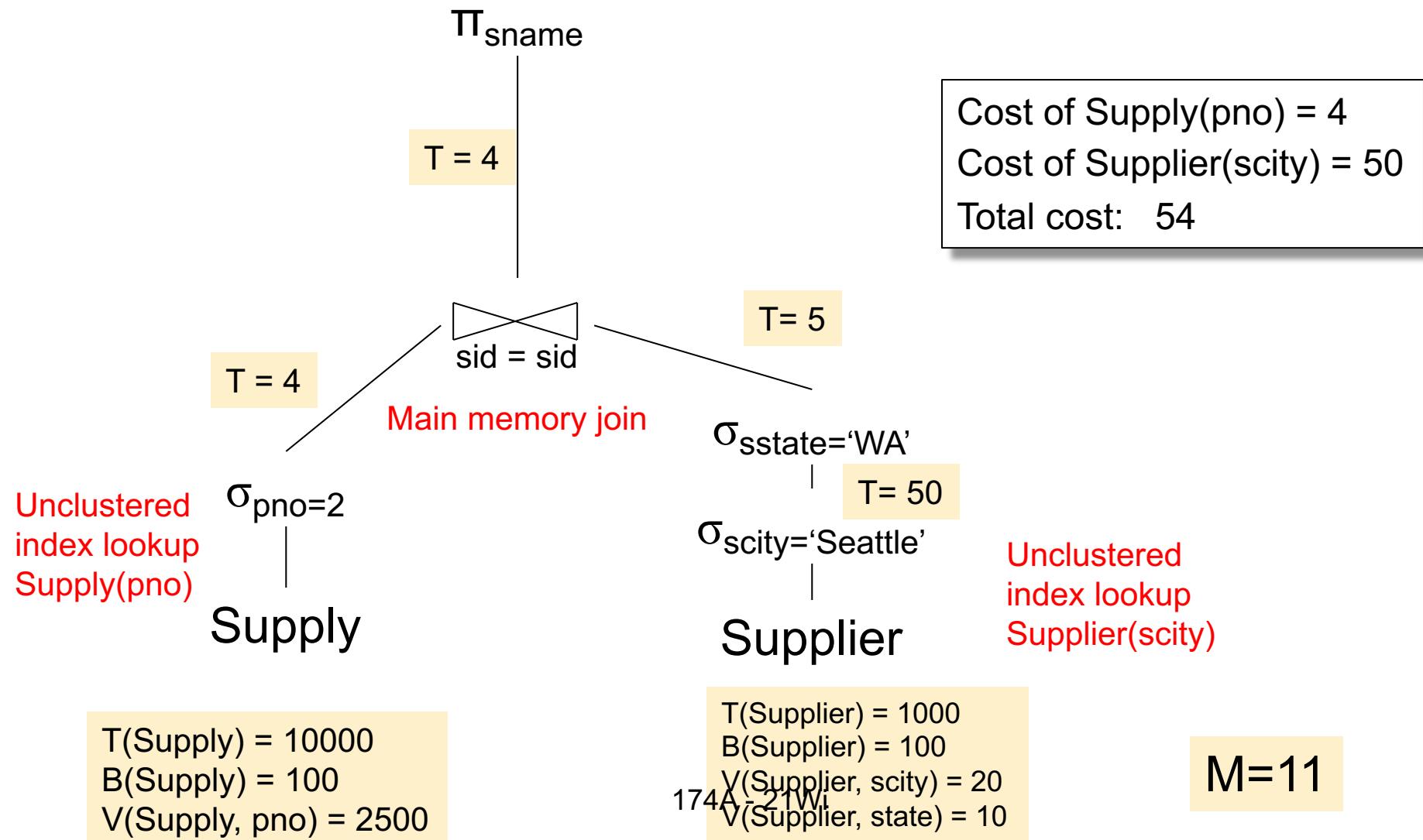
Physical Plan 2



Supplier(sid, sname, scity, sstate)

Supply(sid, pno, quantity)

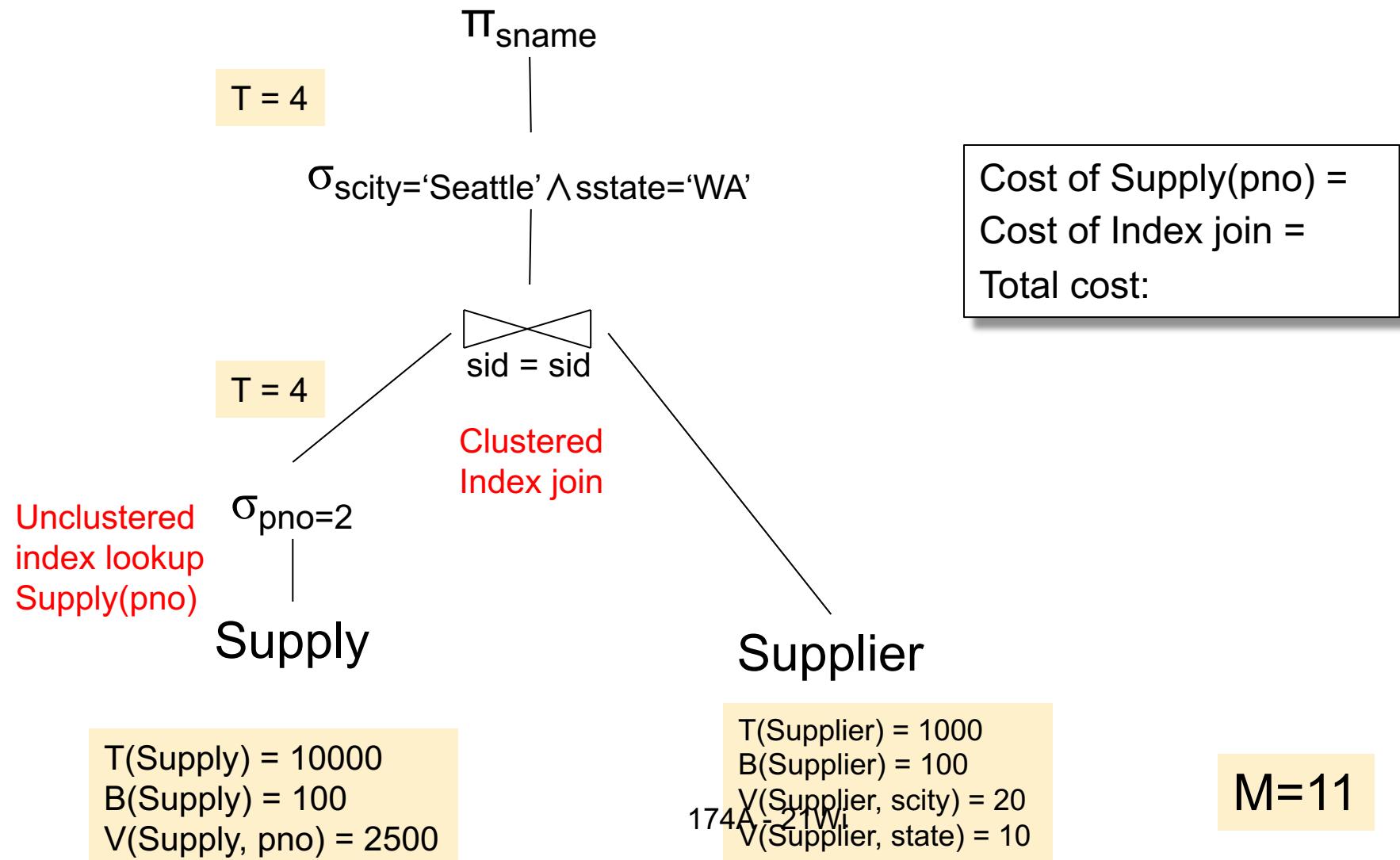
Physical Plan 2



Supplier(sid, sname, scity, sstate)

Supply(sid, pno, quantity)

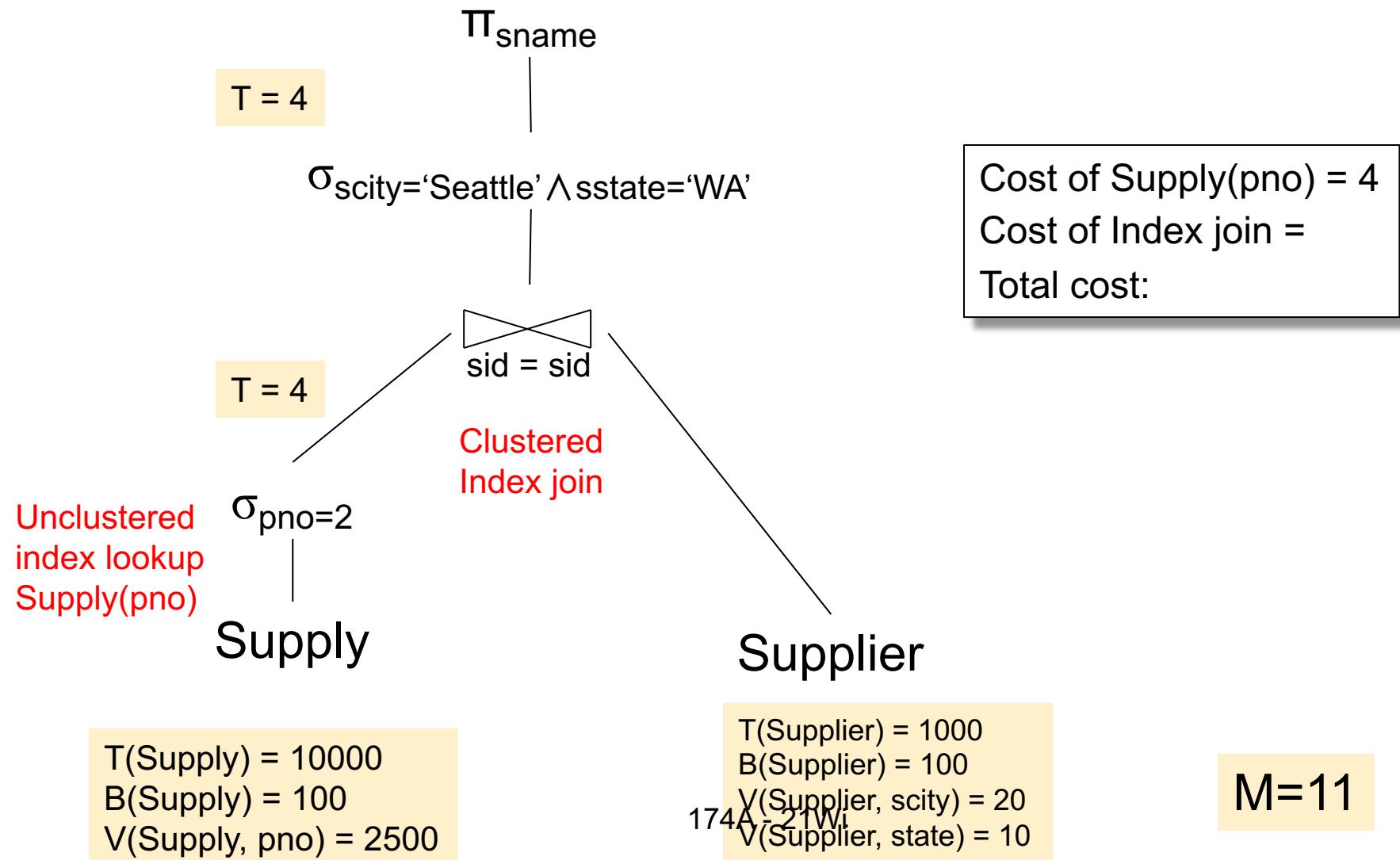
Physical Plan 3



Supplier(sid, sname, scity, sstate)

Supply(sid, pno, quantity)

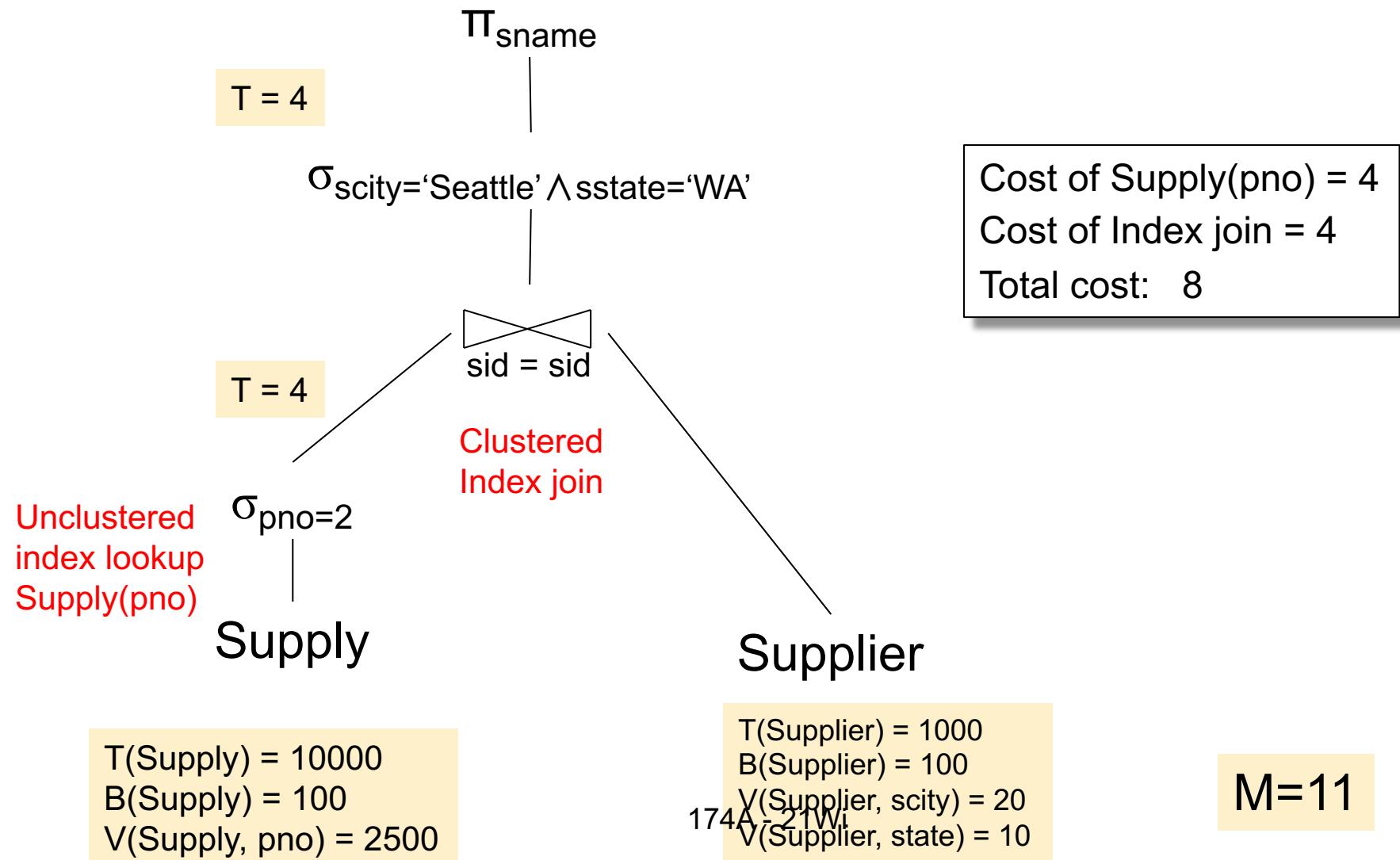
Physical Plan 3



Supplier(sid, sname, scity, sstate)

Supply(sid, pno, quantity)

Physical Plan 3



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