CSE544
Data Management

Lectures 4-6

Query Execution

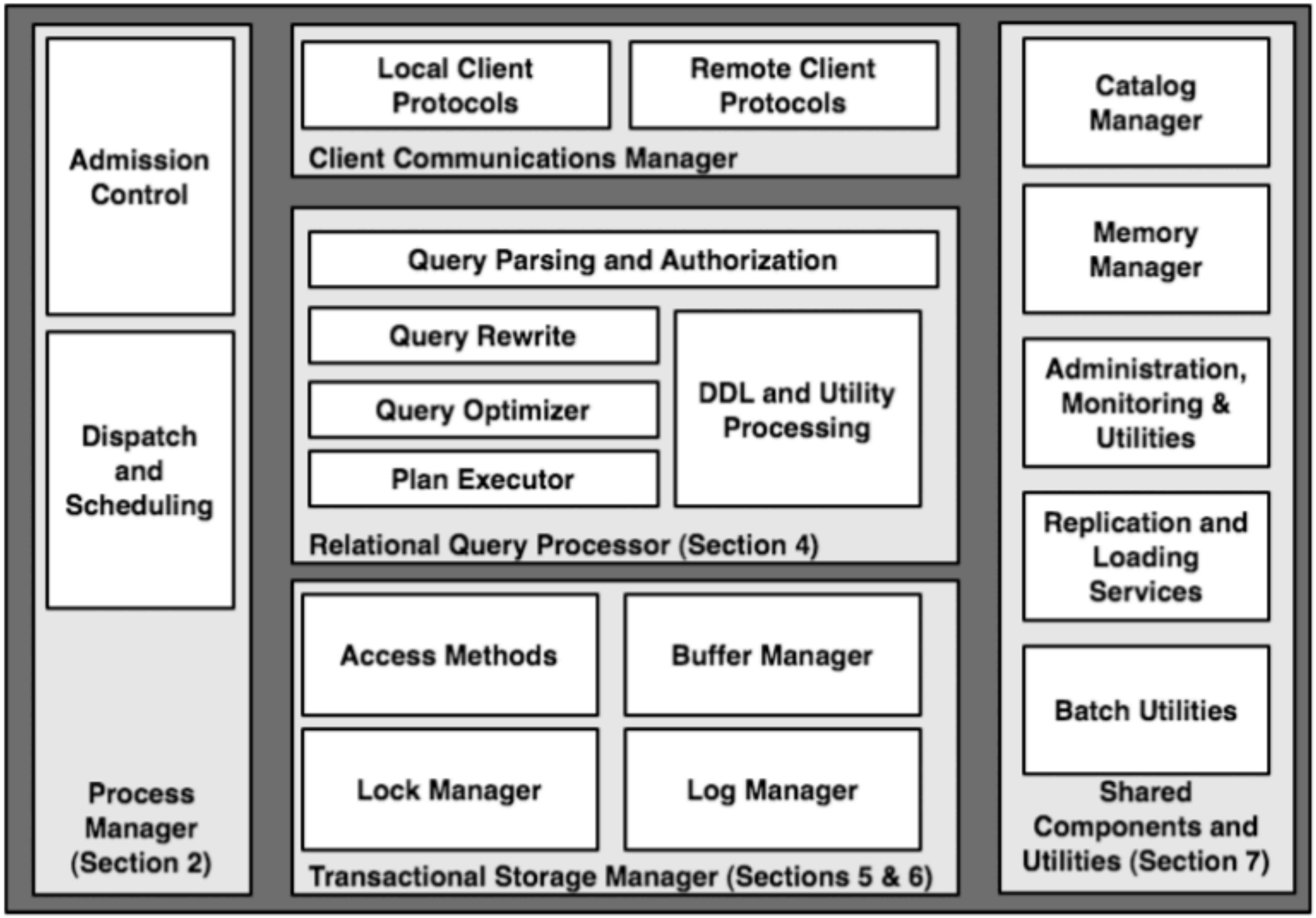
Announcements

- HW1 due on Friday
- No lecture on Monday
- Review 2 due on Wednesday (Ch. 1&2 only)
- Project groups by next Friday (email to me)

Outline for the Next 3 Lectures

- Architecture of a DBMS
- Steps involved in processing a query
- Main Memory Operators
- Storage
- External Memory Operators

Architecture of DBMS

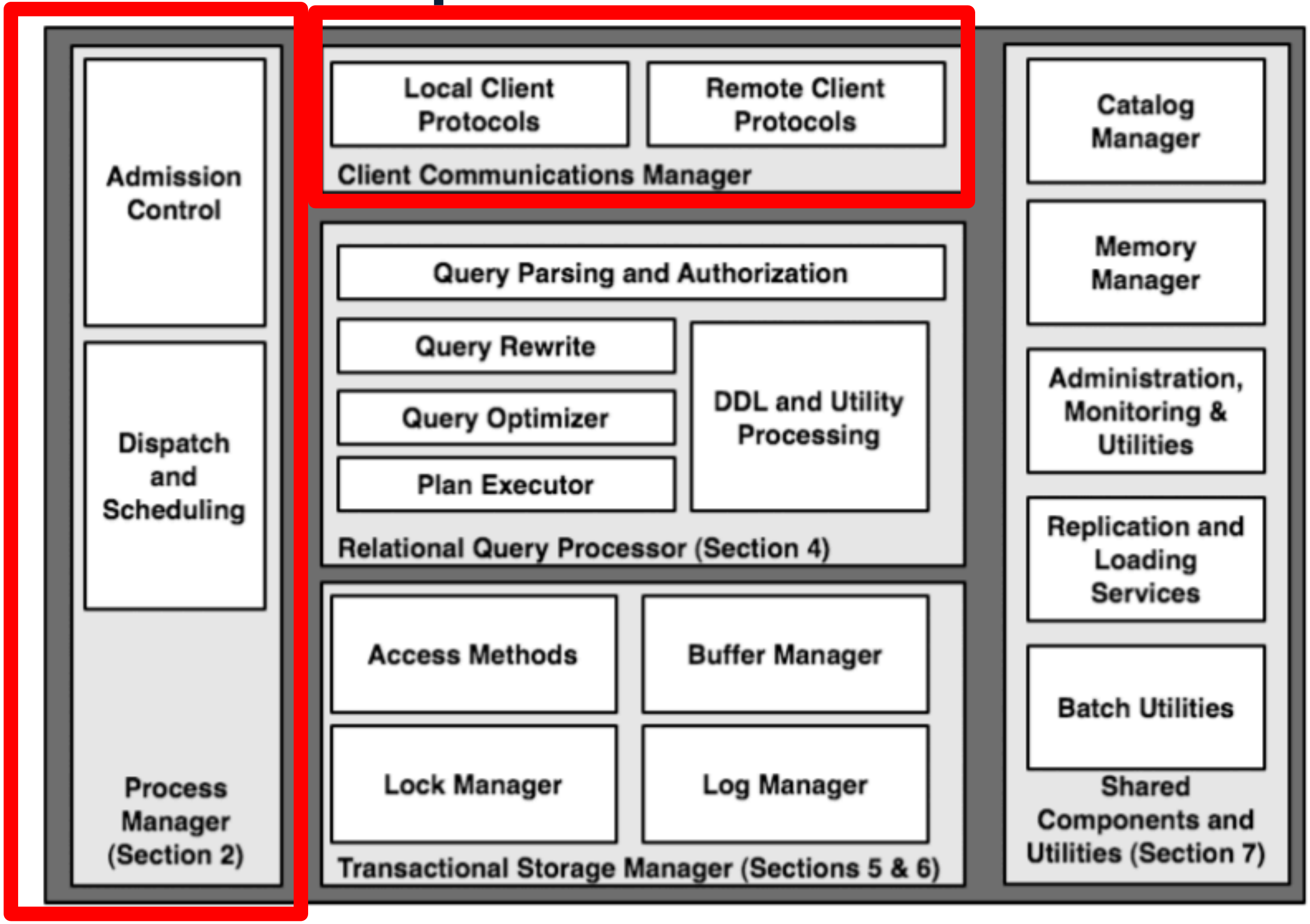


Warning: it will be confusing…

DBMS are monoliths: components cannot be isolated

- Good news:
  - Hole system has rich functionality and is efficient
- Bad news:
  - Hard to discuss components in isolation

Multiple Processes



Why Multiple Processes

- DBMS listens to requests from clients
- Each request = one SQL command
- Handles multiple requests concurrently; multiple processes

Process Models

- Process per DBMS worker
- Thread per DBMS worker
- Process pool

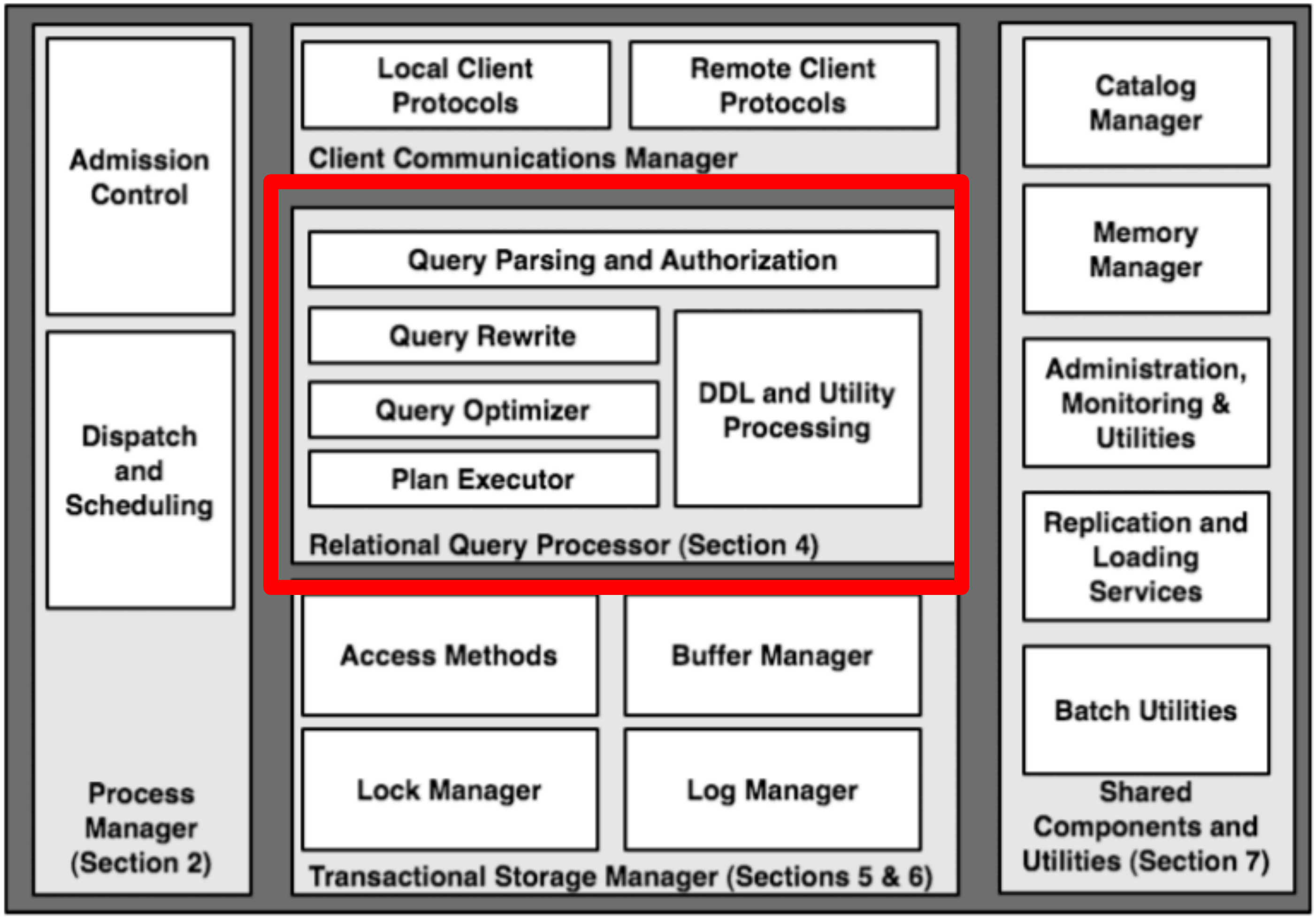
Next week's review:

Discuss pro/cons for each model

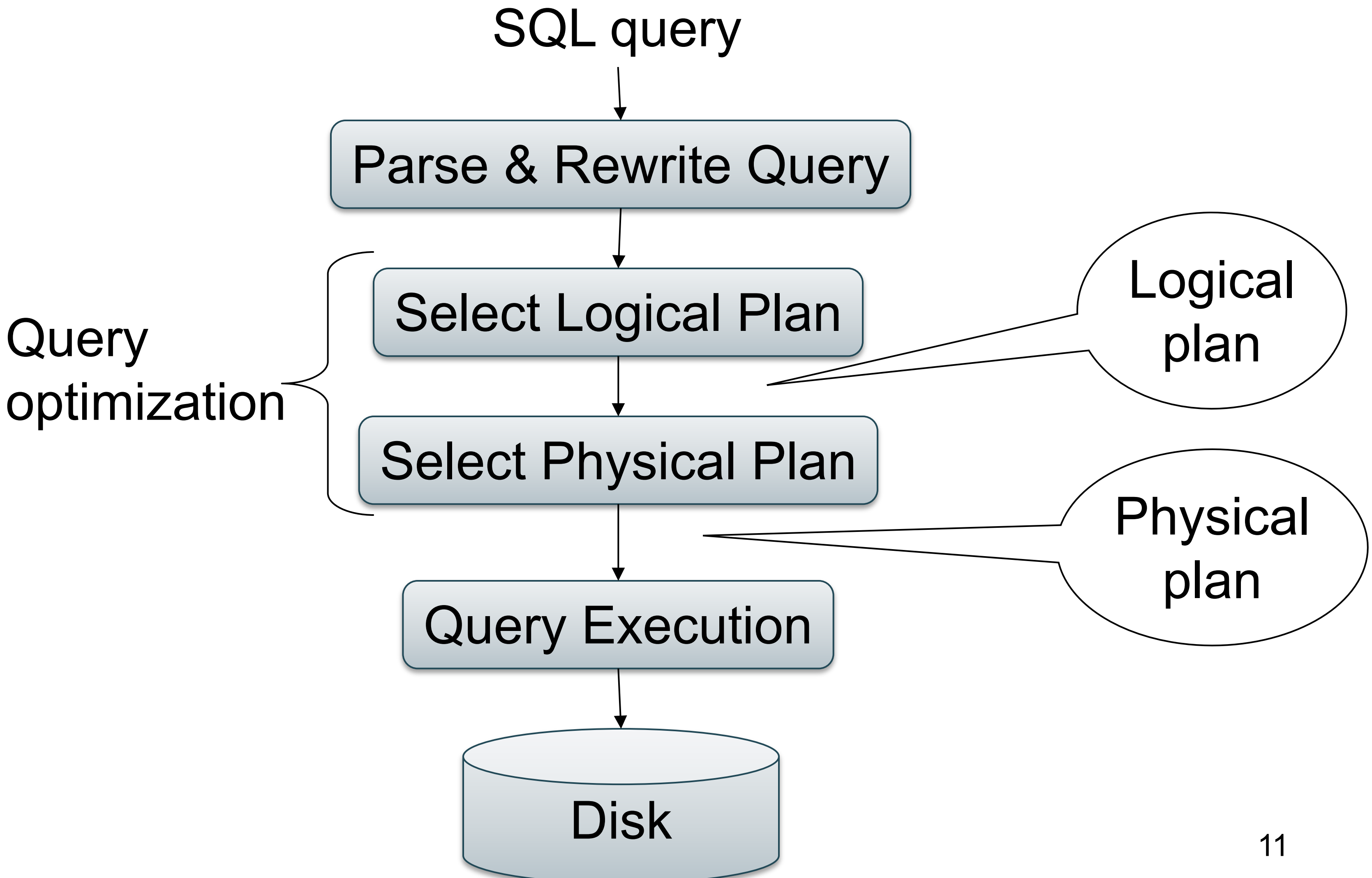
Outline

- Architecture of a DBMS
- Steps involved in processing a query
- Main Memory Operators
- Storage
- External Memory Operators

Query Optimization



Lifecycle of a Query



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

NearbySupp(sno, sname)

Example Database Schema

View: Suppliers in Seattle

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

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

NearbySupp(sno, sname)

Example Query

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

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

Lifecycle of a Query (1)

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

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

NearbySupp(sno, sname)

View Rewriting, Flattening

Original query:

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

View rewriting
 = view inlining
 = view expansion
Flattening
 = unnesting

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

NearbySupp(sno, sname)

View Rewriting, Flattening

Original query:

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

View rewriting
 = view inlining
 = view expansion
Flattening
 = unnesting

Rewritten query:

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

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

Decorrelation

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

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

Decorrelation

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

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

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

Decorrelation

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

Correlation !

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

Decorrelation

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

De-Correlation

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

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

Decorrelation

Un-nesting

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

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

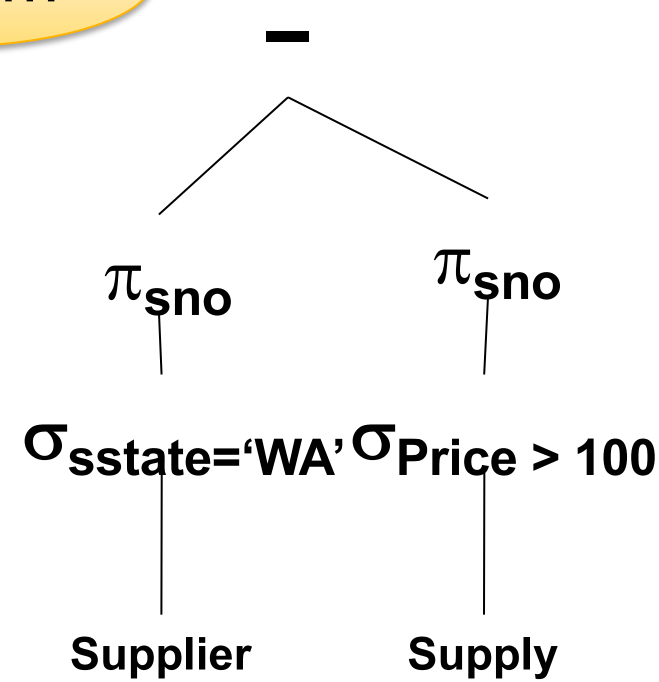
EXCEPT = set difference

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

Decorrelation

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

Finally…



Lifecycle of a Query (2)

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

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

Five Basic Relational Operators

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

Other operators: semi-join, anti-semijoin

Extended Operators of Relational Algebra

- Duplicate elimination (δ)
  - Convert a bag to a set
  - Can be expressed as a group-by γ
- Group-by/aggregate (γ)
  - Example: $\gamma_{\text{pcolor, max(psize)}\rightarrow\text{m, avg(psize)}\rightarrow\text{s}}(\text{Part})$
  - Min, max, sum, average, count
  - Partitions tuples of a relation into "groups"
  - Aggregates can then be applied to groups
- Sort operator (τ)

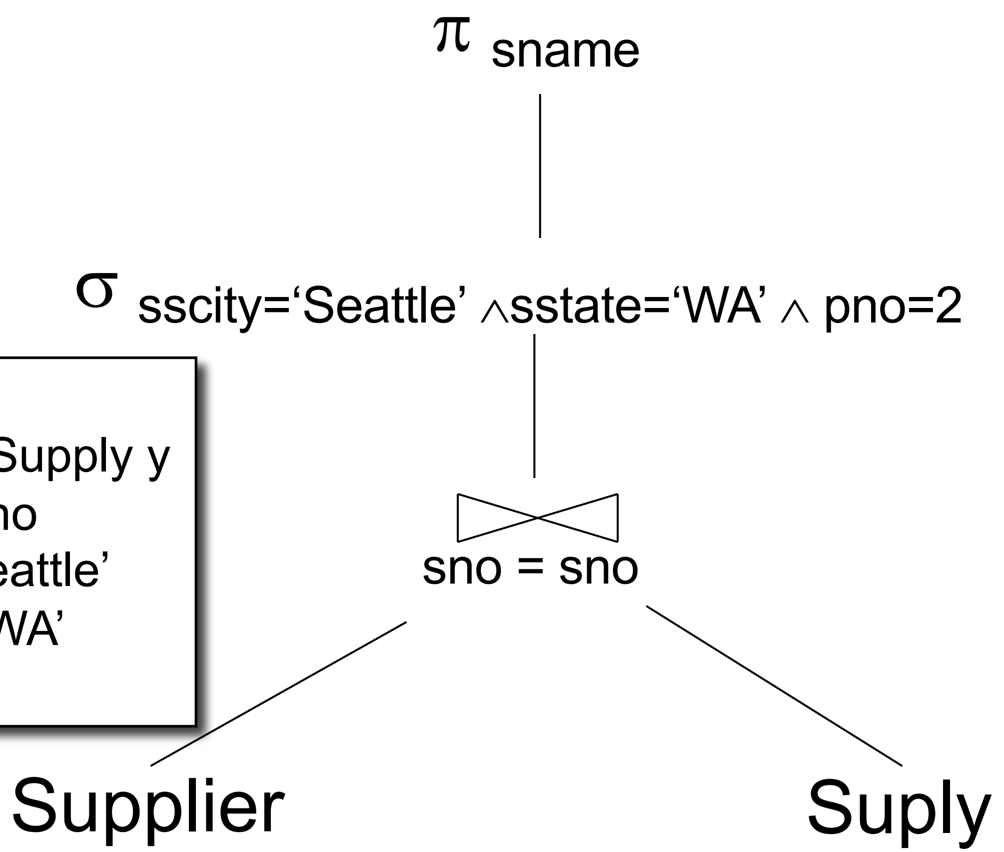
Supplier(sno,sname,scity,sstate)
Part(pno,pname,psize,pcolor)
Supply(sno,pno,price)

Logical Query Plan

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

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

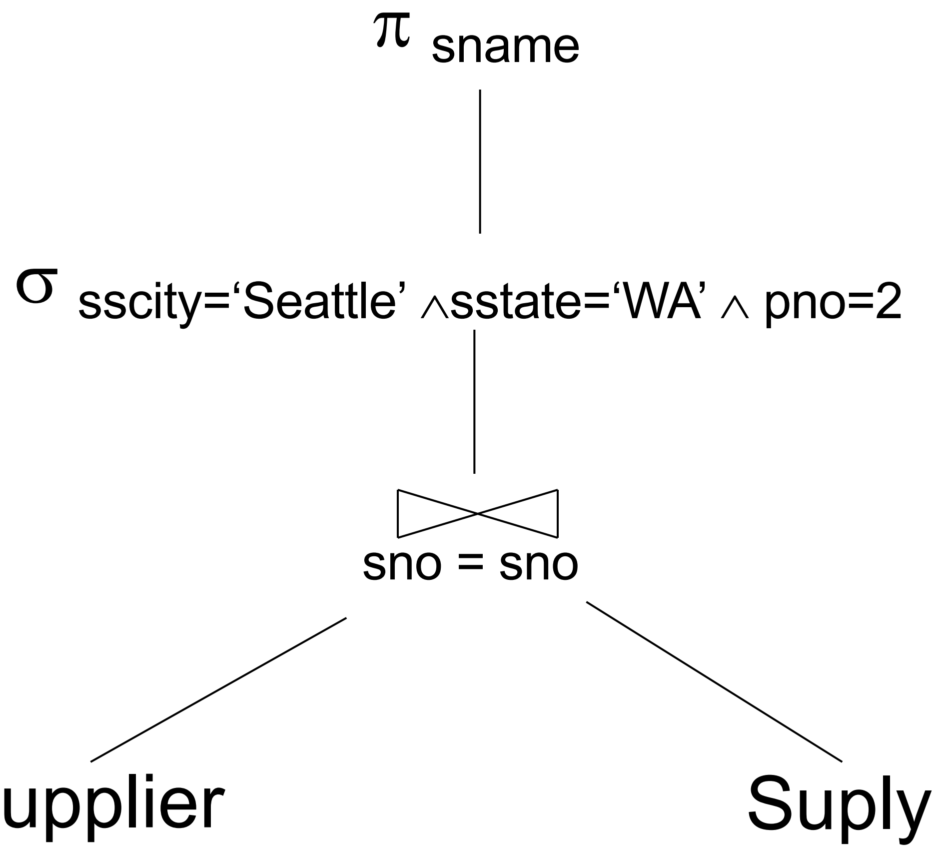
Logical Query Plan




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

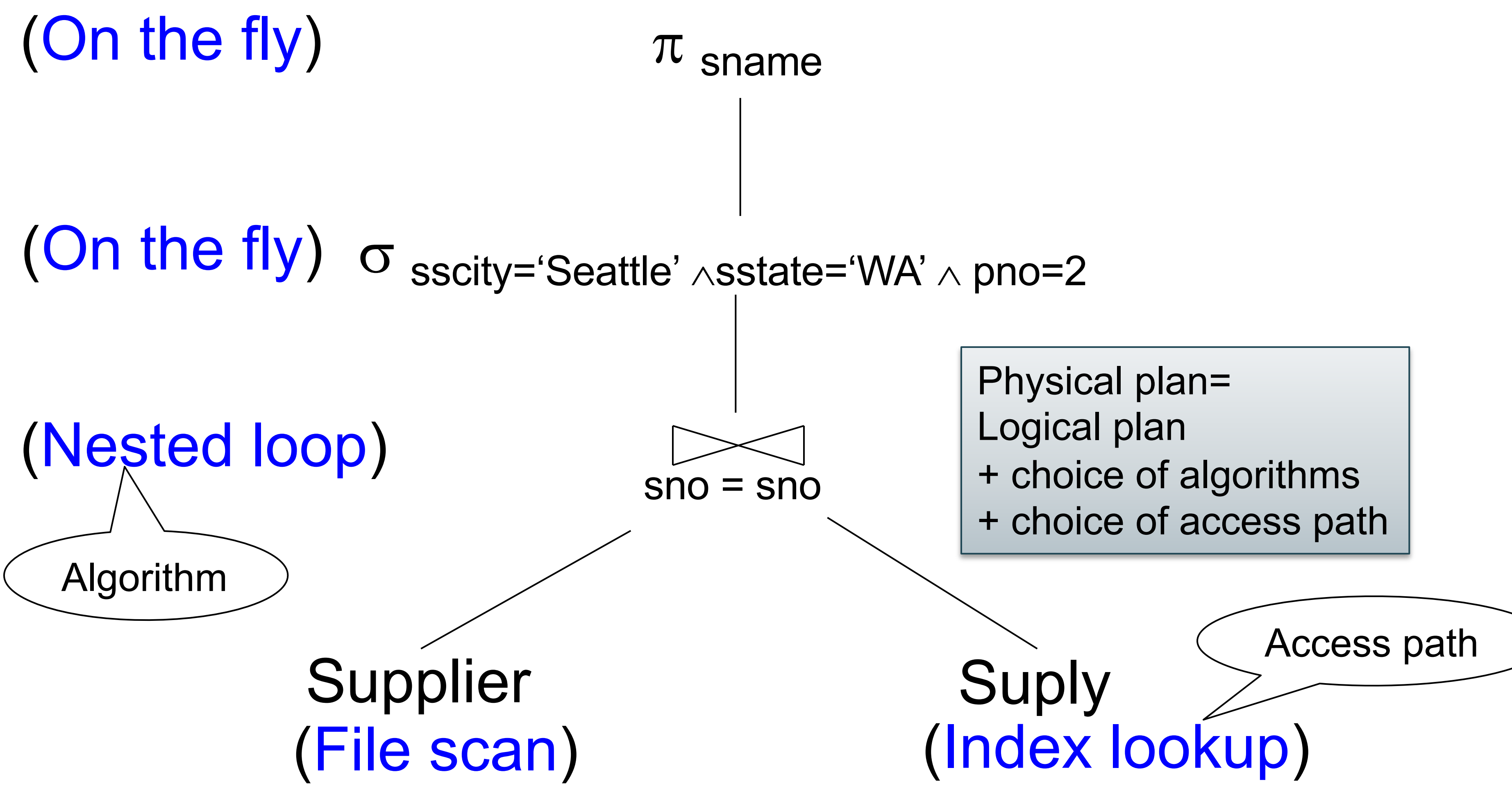
Supplier(sno,sname,scity,sstate)
Part(pno,pname,psize,pcolor)
Supply(sno,pno,price)

Logical Query Plan



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

Physical Query Plan



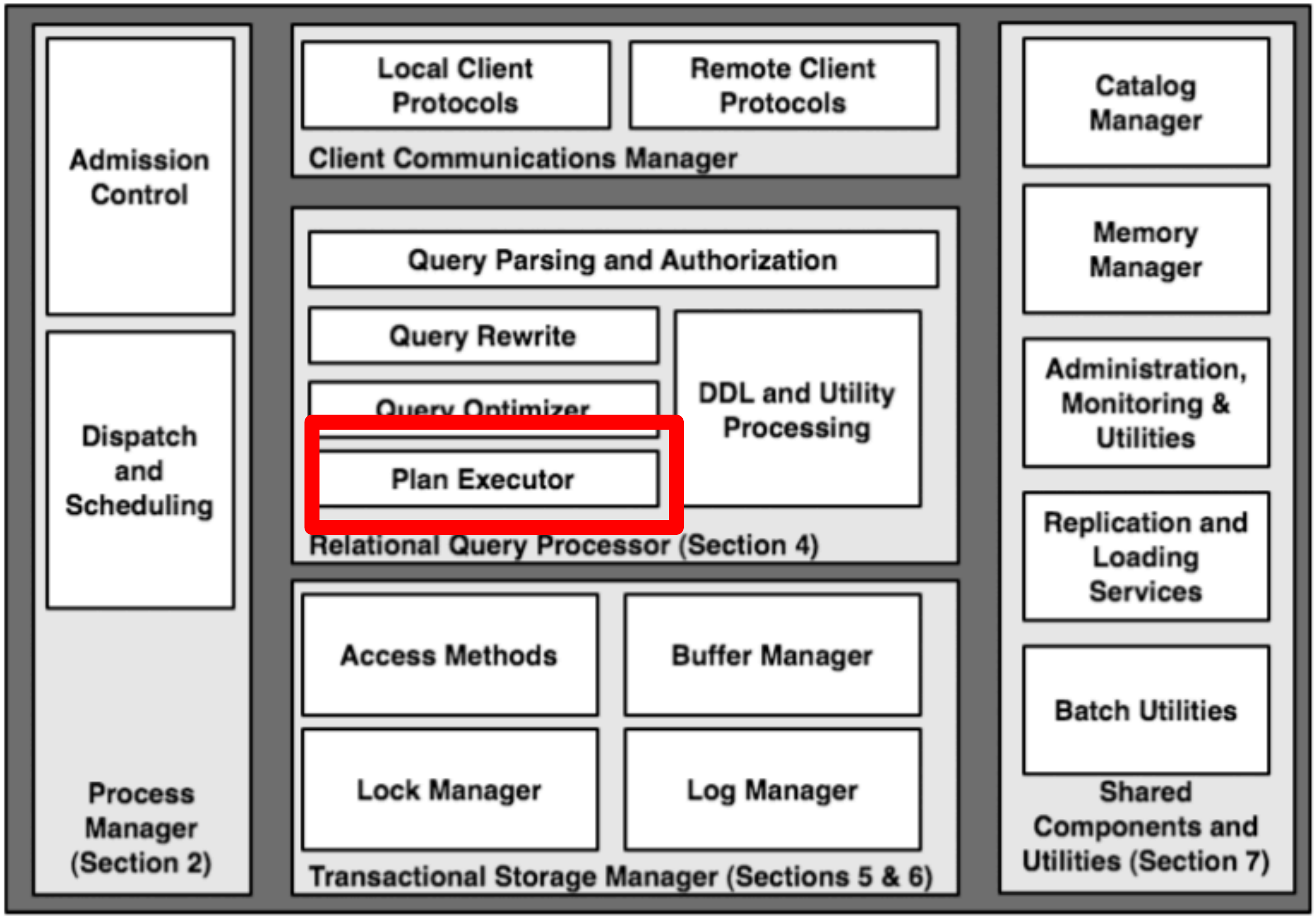
Final Step in Query Processing

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

Outline

- Architecture of a DBMS
- Steps involved in processing a query
- Main Memory Operators
- Storage
- External Memory Operators

Multiple Processes



Physical Operators

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

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

Main Memory Algorithms

Logical operator:

Supplier $\bowtie_{sid=sid}$ Supply

Three algorithms:

1. Nested Loops
2. Hash-join
3. Merge-join

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

1. Nested Loop Join

Logical operator:

Supplier $\bowtie_{sid=sid}$ Supply

```
for x in Supplier do
    for y in Supply do
        if x.sid = y.sid
            then output(x,y)
```

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

1. Nested Loop Join

Logical operator:

Supplier $\bowtie_{sid=sid}$ Supply

```
for x in Supplier do
  for y in Supply do
    if x.sid = y.sid
      then output(x,y)
```

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

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

1. Nested Loop Join

Logical operator:

Supplier $\bowtie_{sid=sid}$ Supply

```
for x in Supplier do
    for y in Supply do
        if x.sid = y.sid
            then output(x,y)
```

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

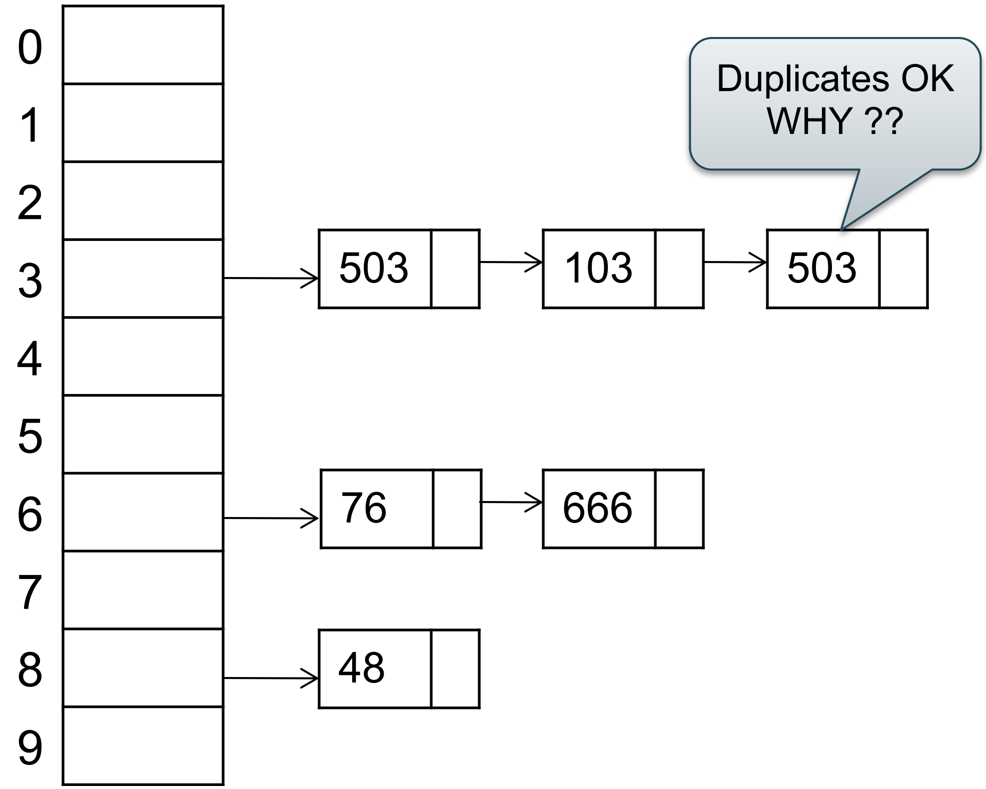
$O(n^2)$

BRIEF Review of Hash Tables

A (naïve) hash function:

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

Separate chaining:

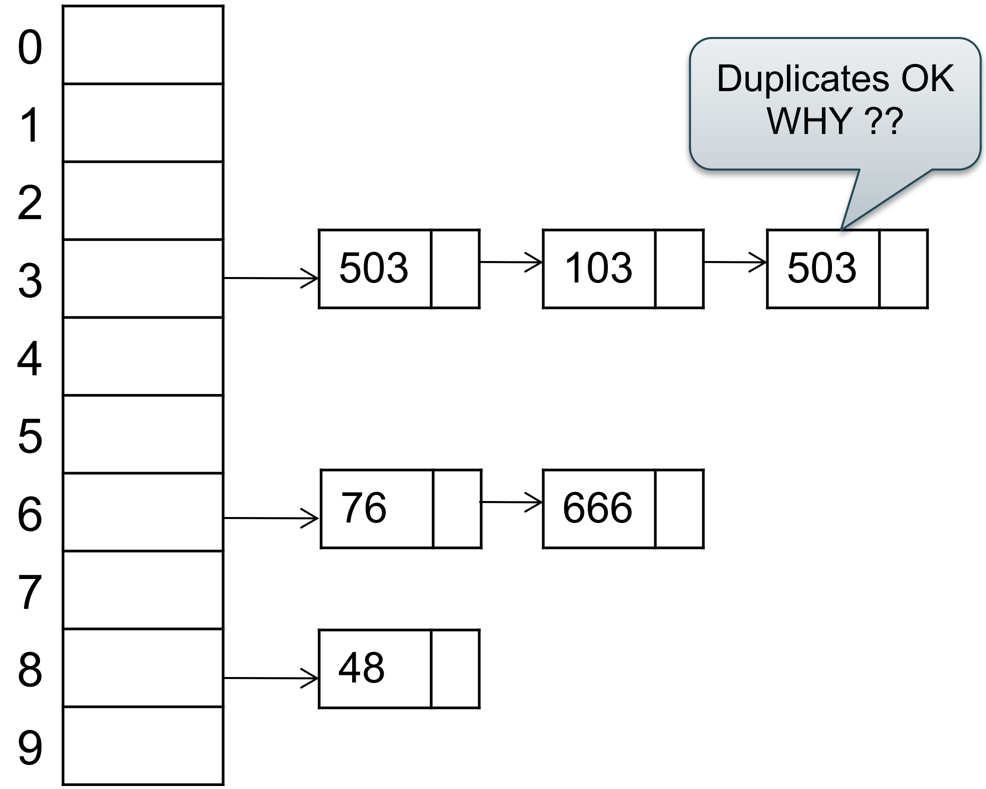


BRIEF Review of Hash Tables

Separate chaining:

A (naïve) hash function:

$h(x) = x \bmod 10$



Operations:

find(103) = ??

BRIEF Review of Hash Tables

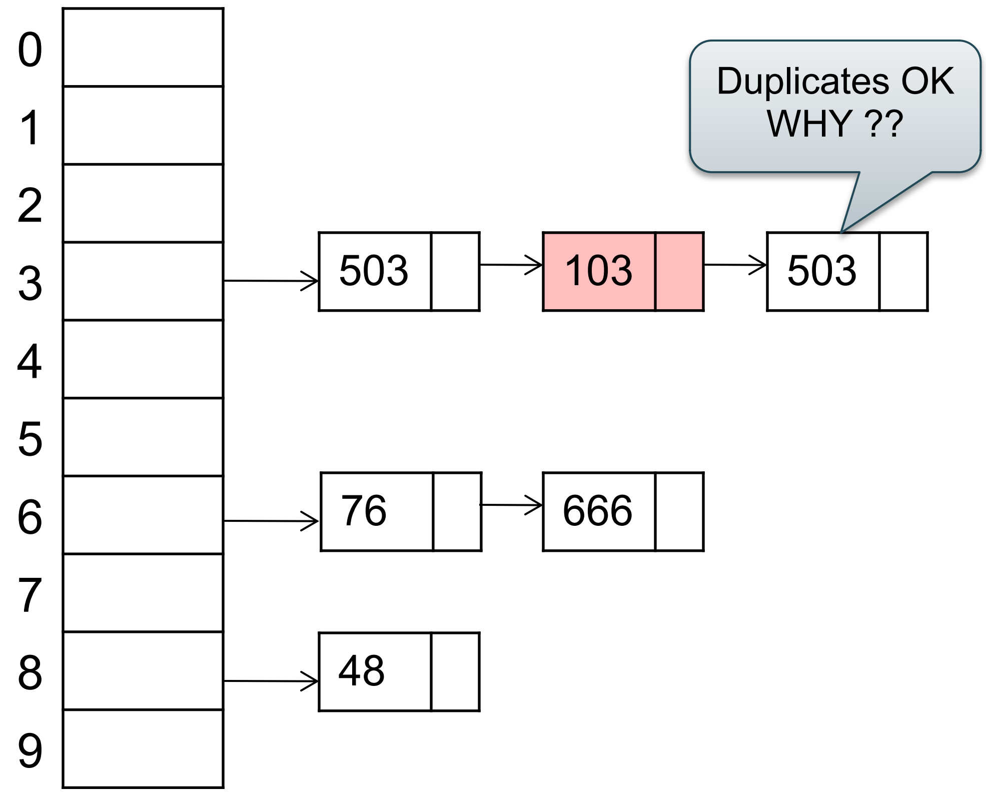
A (naïve) hash function:

$h(x) = x \bmod 10$

Operations:

find(103) = ??

Separate chaining:

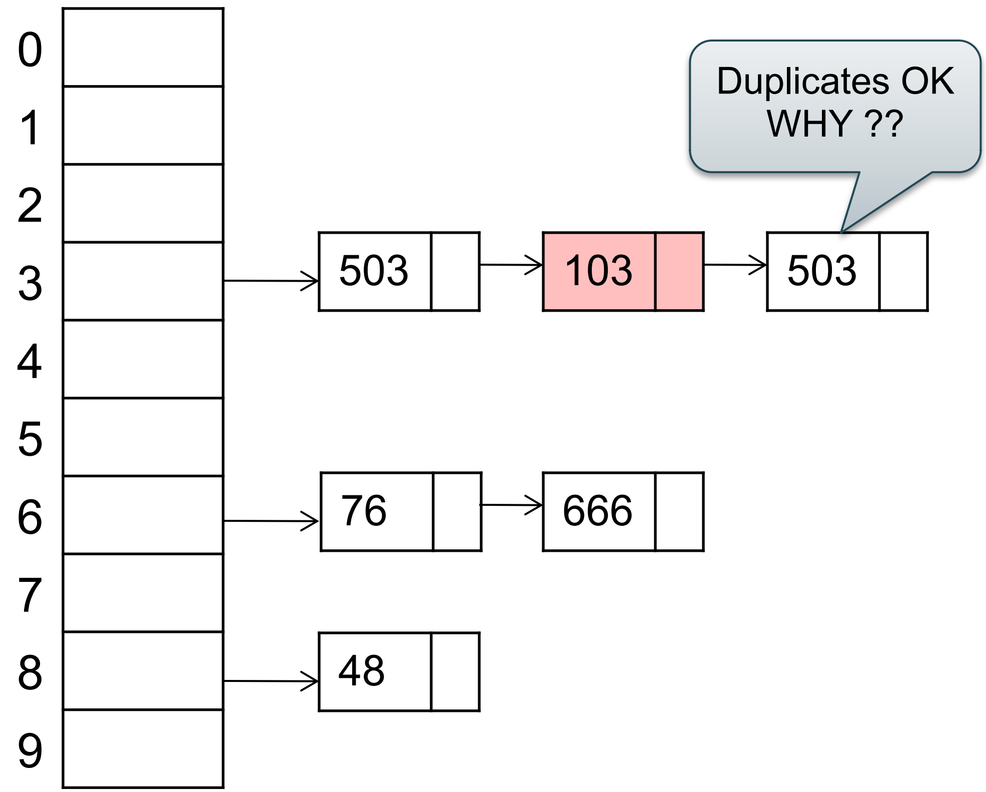


BRIEF Review of Hash Tables

A (naïve) hash function:

h(x) = x mod 10

Separate chaining:



Operations:

find(103) = ??

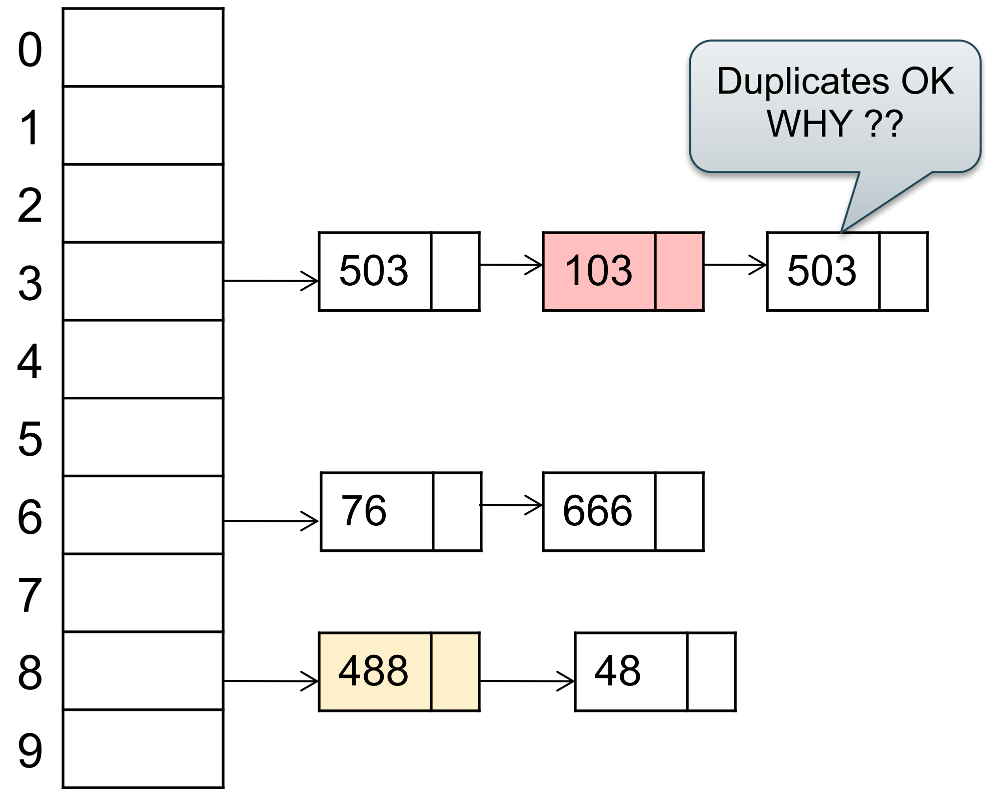
insert(488) = ??

BRIEF Review of Hash Tables

Separate chaining:

A (naïve) hash function:

$h(x) = x \bmod 10$



Operations:

find(103) = ??

insert(488) = ??

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

2. Hash Join

Logical operator:

Supply $\bowtie_{sid=sid}$ Supplier

```
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:

Supply $\bowtie_{sid=sid}$ Supplier

```
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:

Supply $\bowtie_{sid=sid}$ Supplier

```
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?

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?

When |Supply| << |Supplier|

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)
```

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();
```

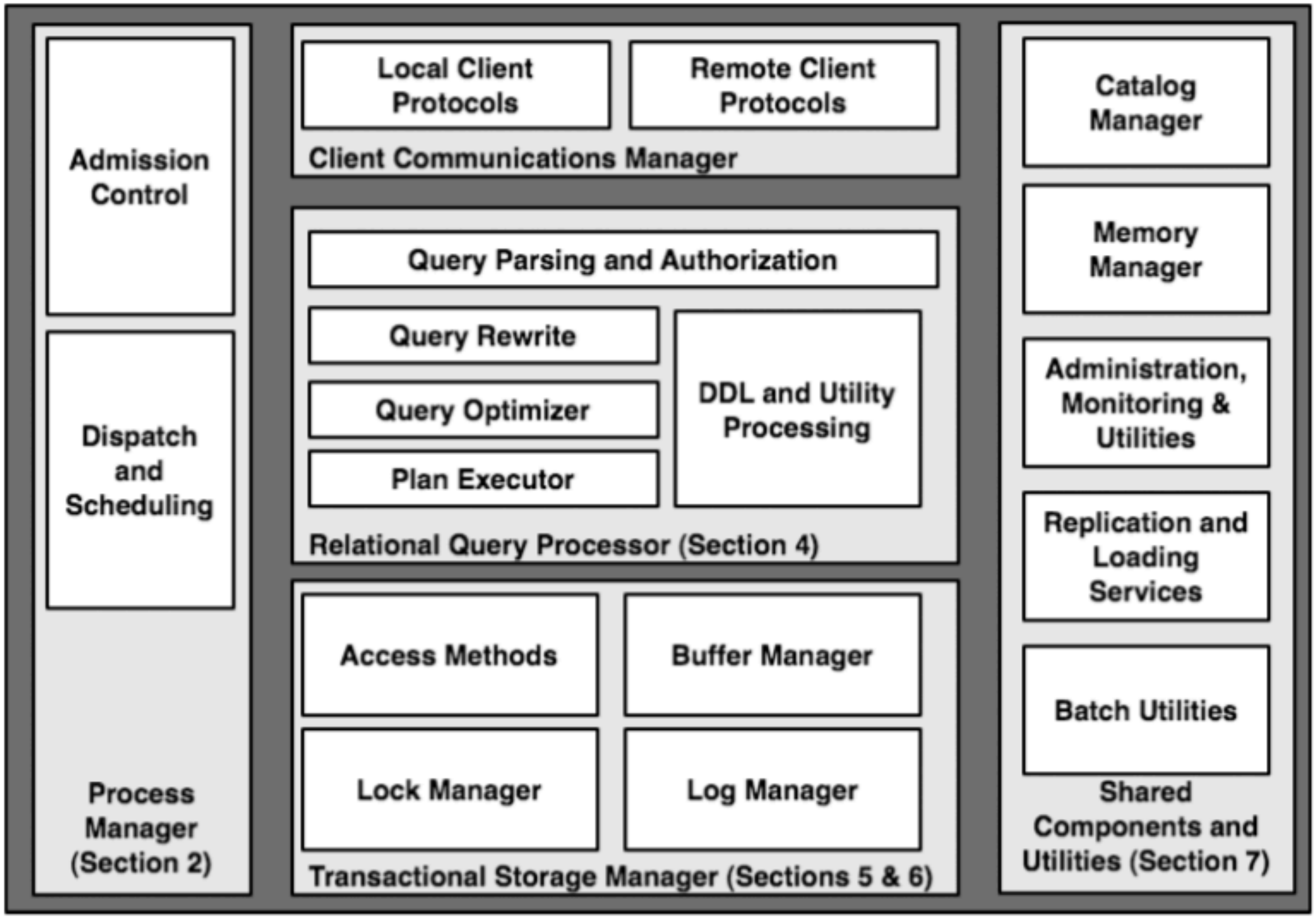
If |R|=|S|=n,
what is the runtime?

O(n log(n))

Announcements

- Project teams due by Friday (email to me)
- HW2 posted, we use Snowflake
  - Consider using Snowflake in your project!
- Architecture paper was due today

Discuss Architecture Paper

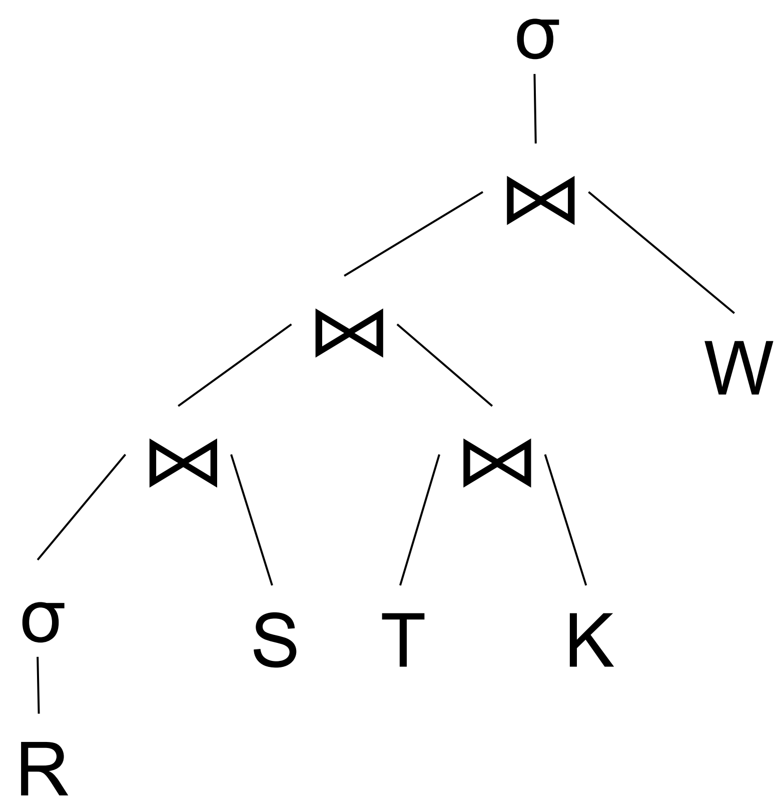


Recap: Main Memory Algorithms

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

Briefly discuss in class

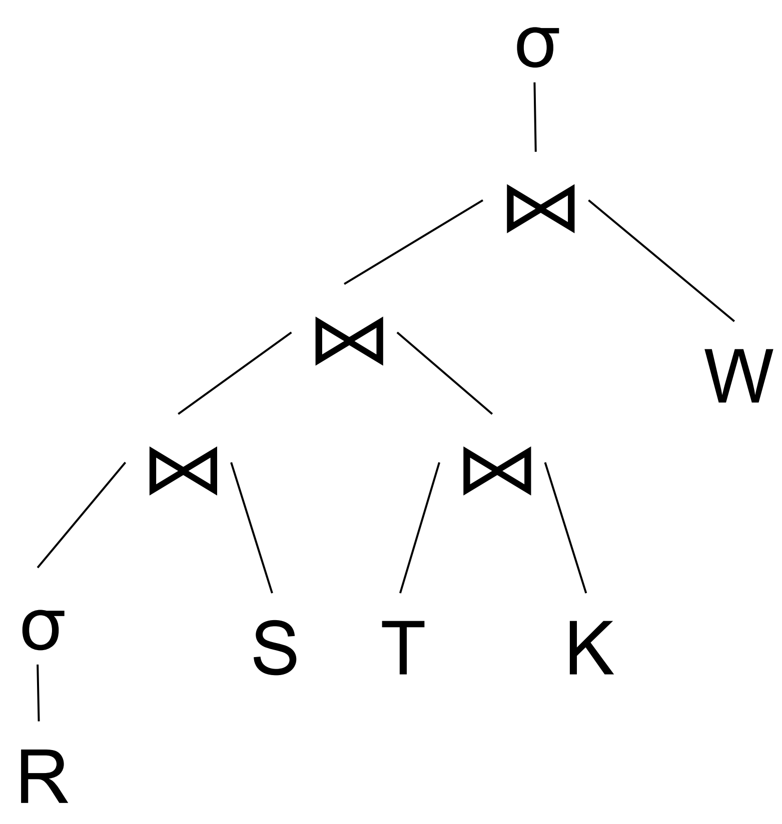
How Do We Combine Them?



How Do We Combine Them?

The *Iterator Interface*

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



Implementing Query Operators with the Iterator Interface

```
interface Operator {





}
```

Implementing Query Operators with the Iterator Interface

```
interface Operator {

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

}
```

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 ();




}
```

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 ();
}
```

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 (...);


  // 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();
  }
}
```

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 () {









  }

}
```

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

```
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

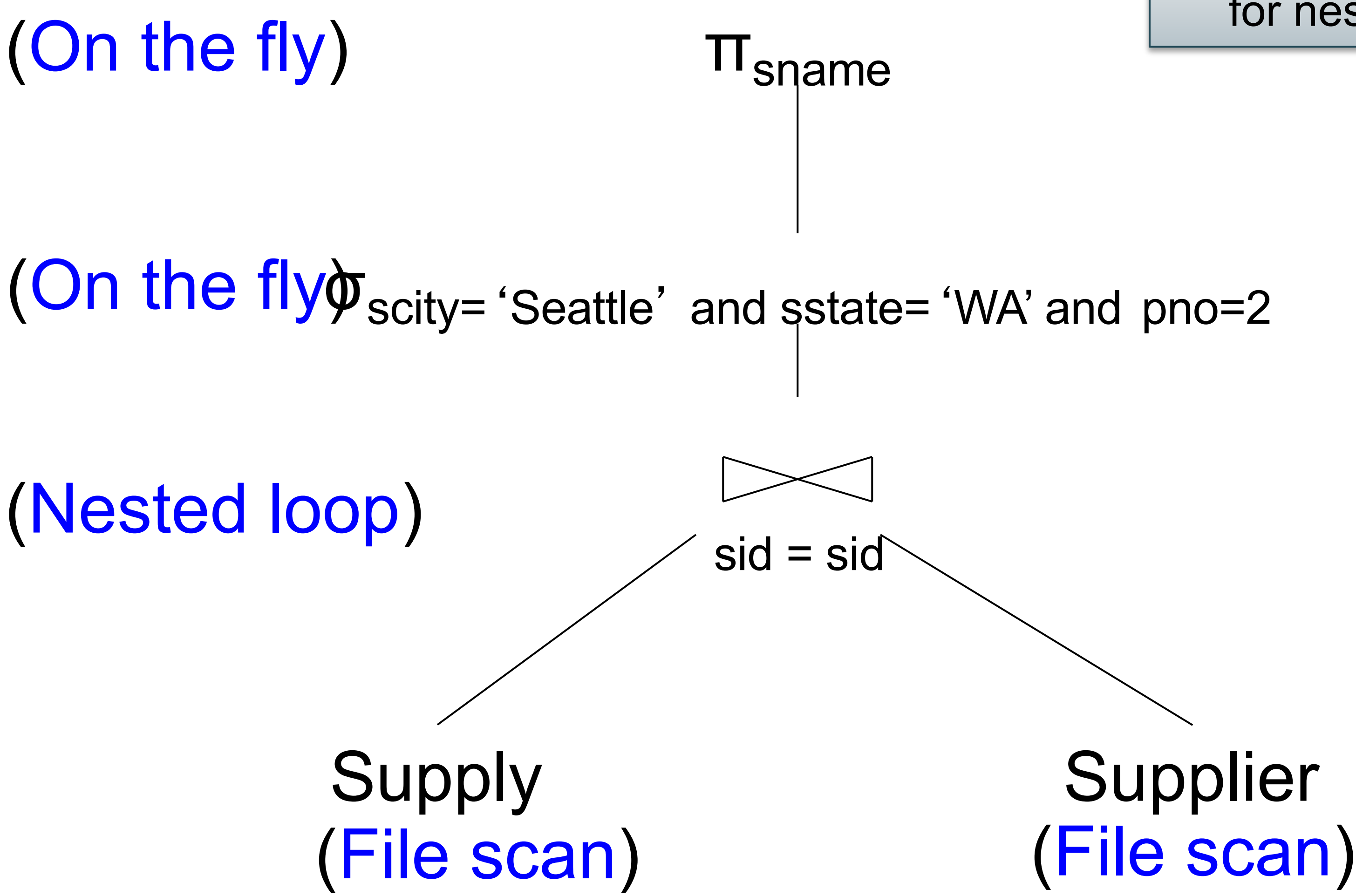
```
Operator q = parse(“SELECT ...”);
q = optimize(q);

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)

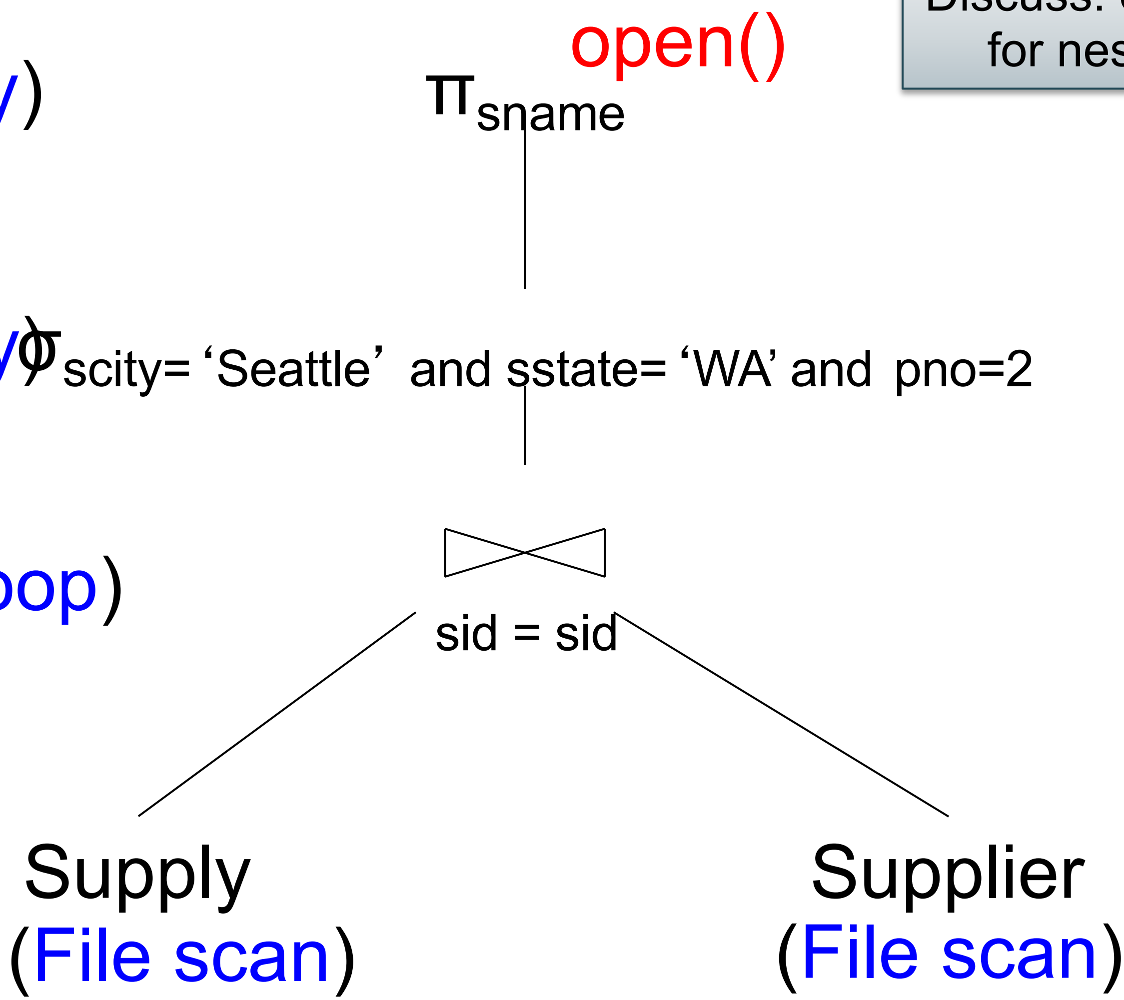
Pipelining

Discuss: open/next/close for nested loop join

(On the fly)

(On the fly)

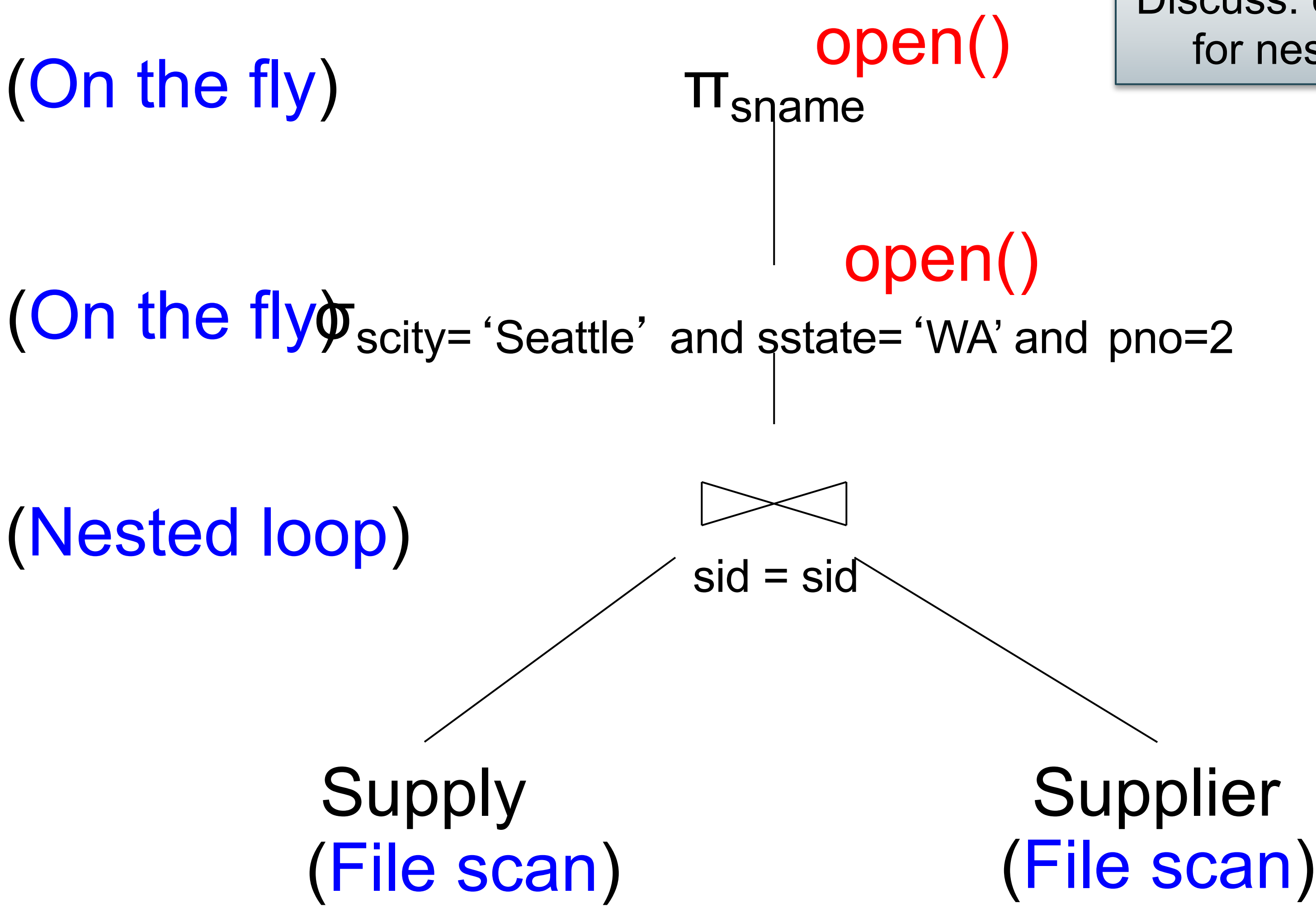
(Nested loop)



```
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

(On the fly) π_{sname} open()

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

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

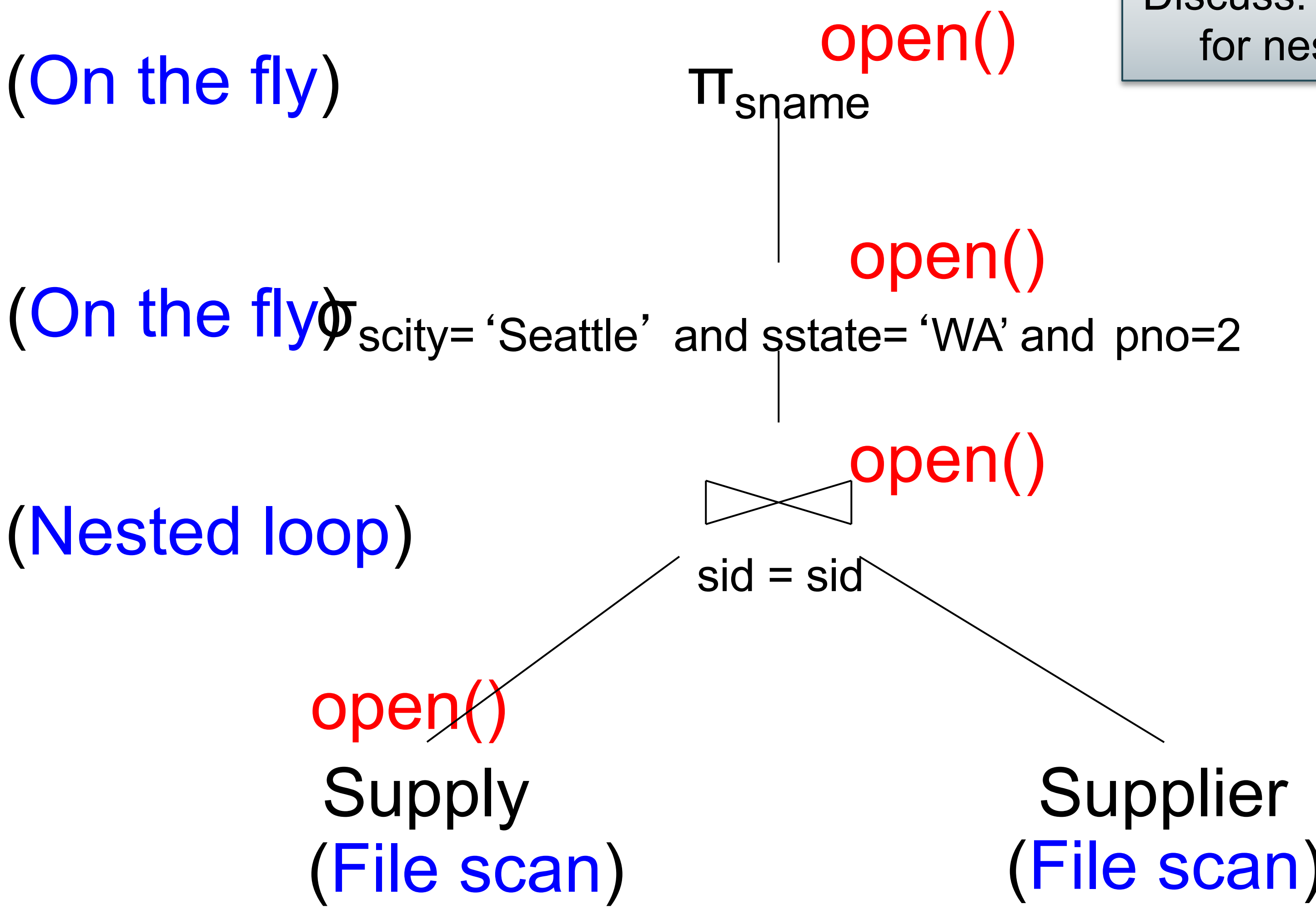
Supply (File scan)

Supplier (File scan)

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

Pipelining

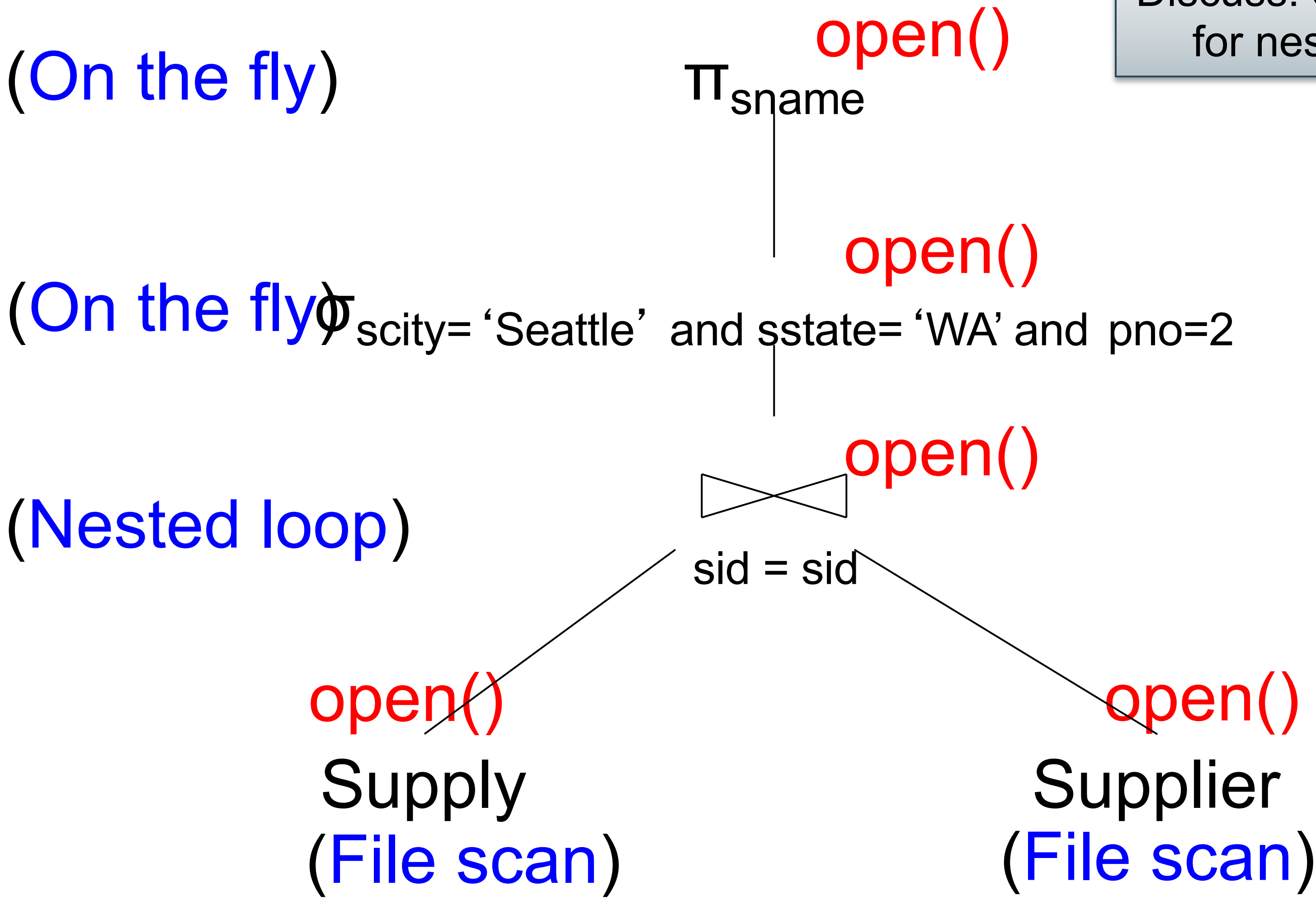
Discuss: open/next/close for nested loop join



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

Pipelining

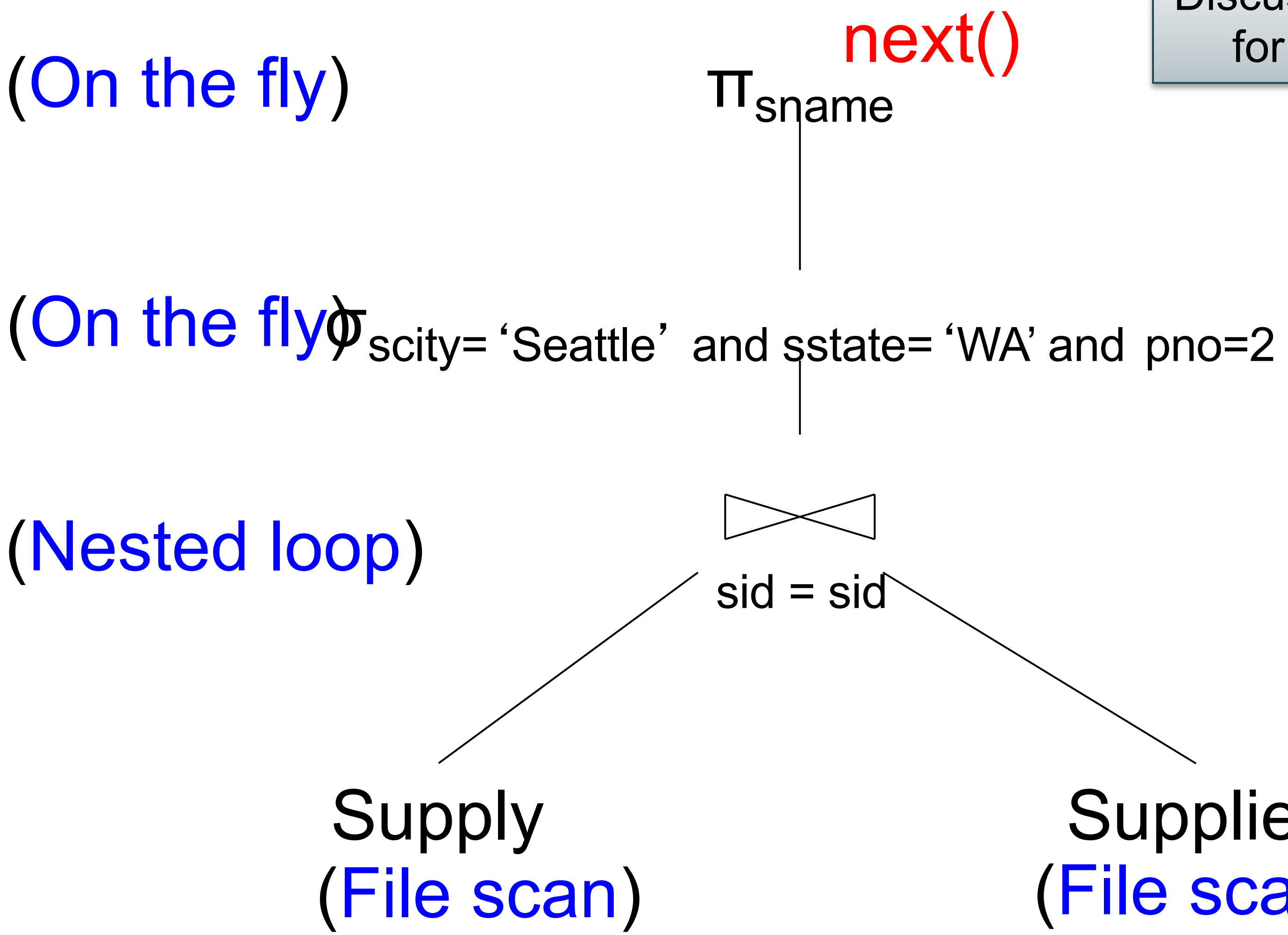
Discuss: open/next/close for nested loop join



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

Pipelining

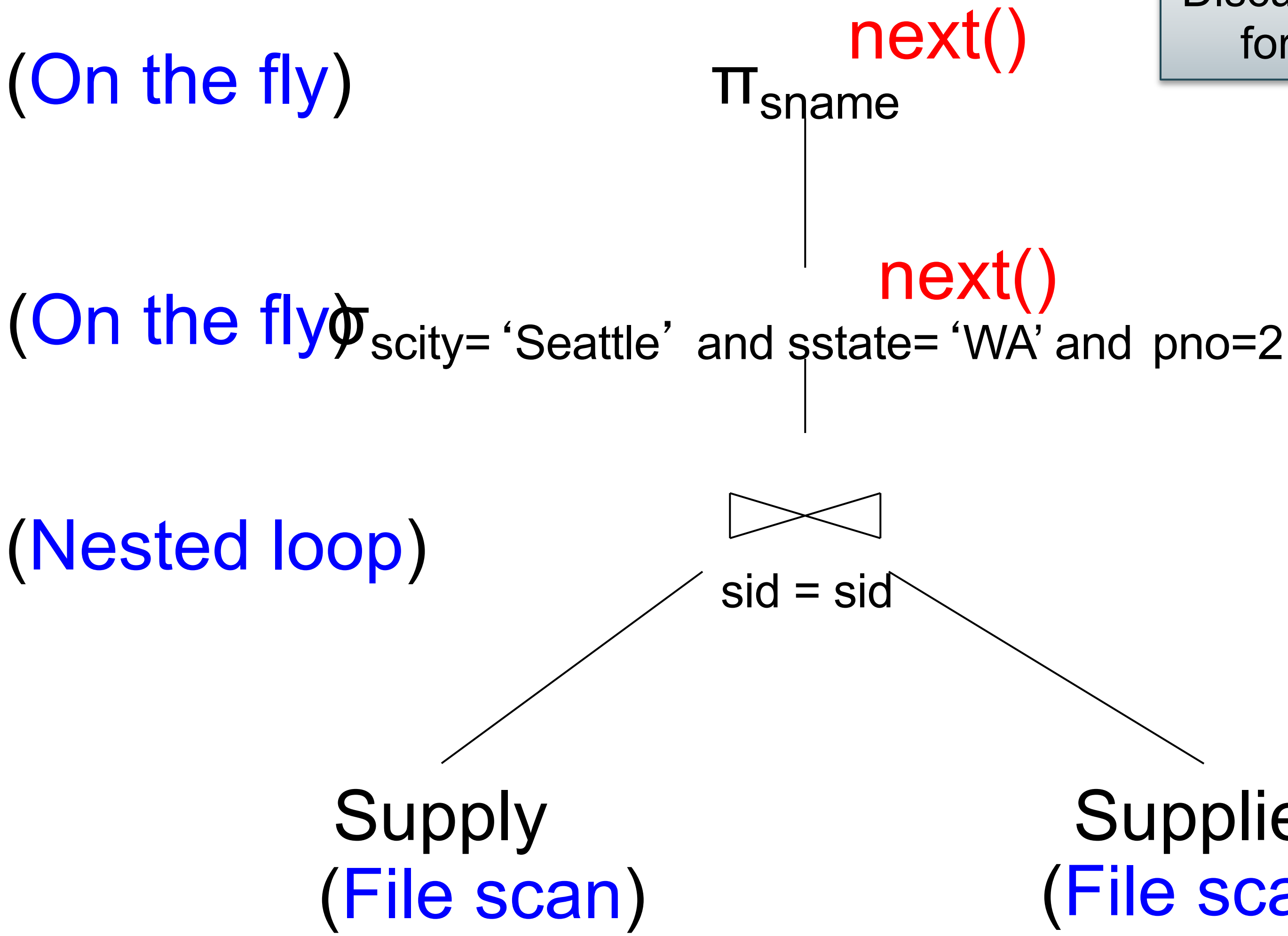
Discuss: open/next/close for nested loop join



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

Pipelining

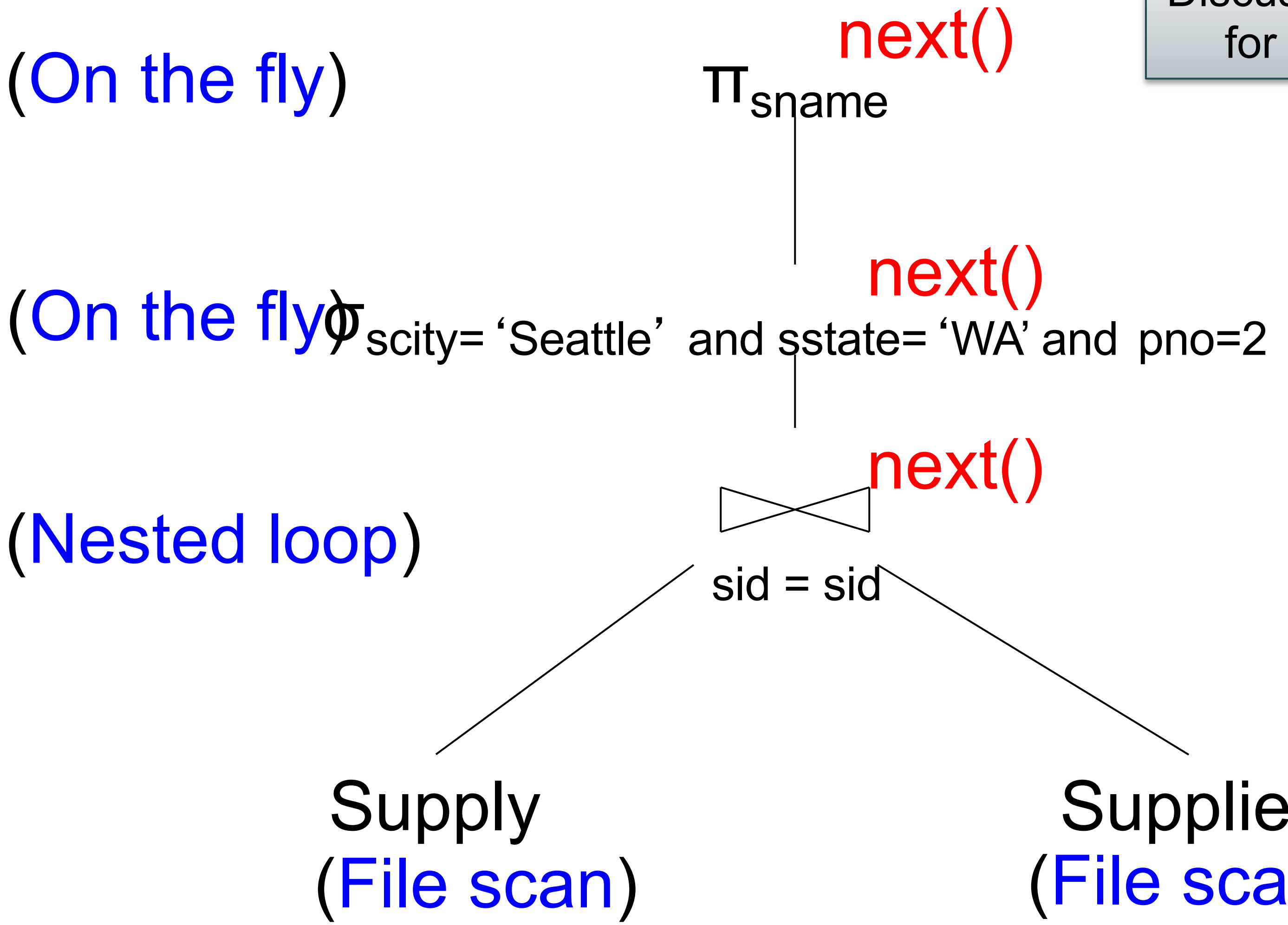
Discuss: open/next/close for nested loop join



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

Pipelining

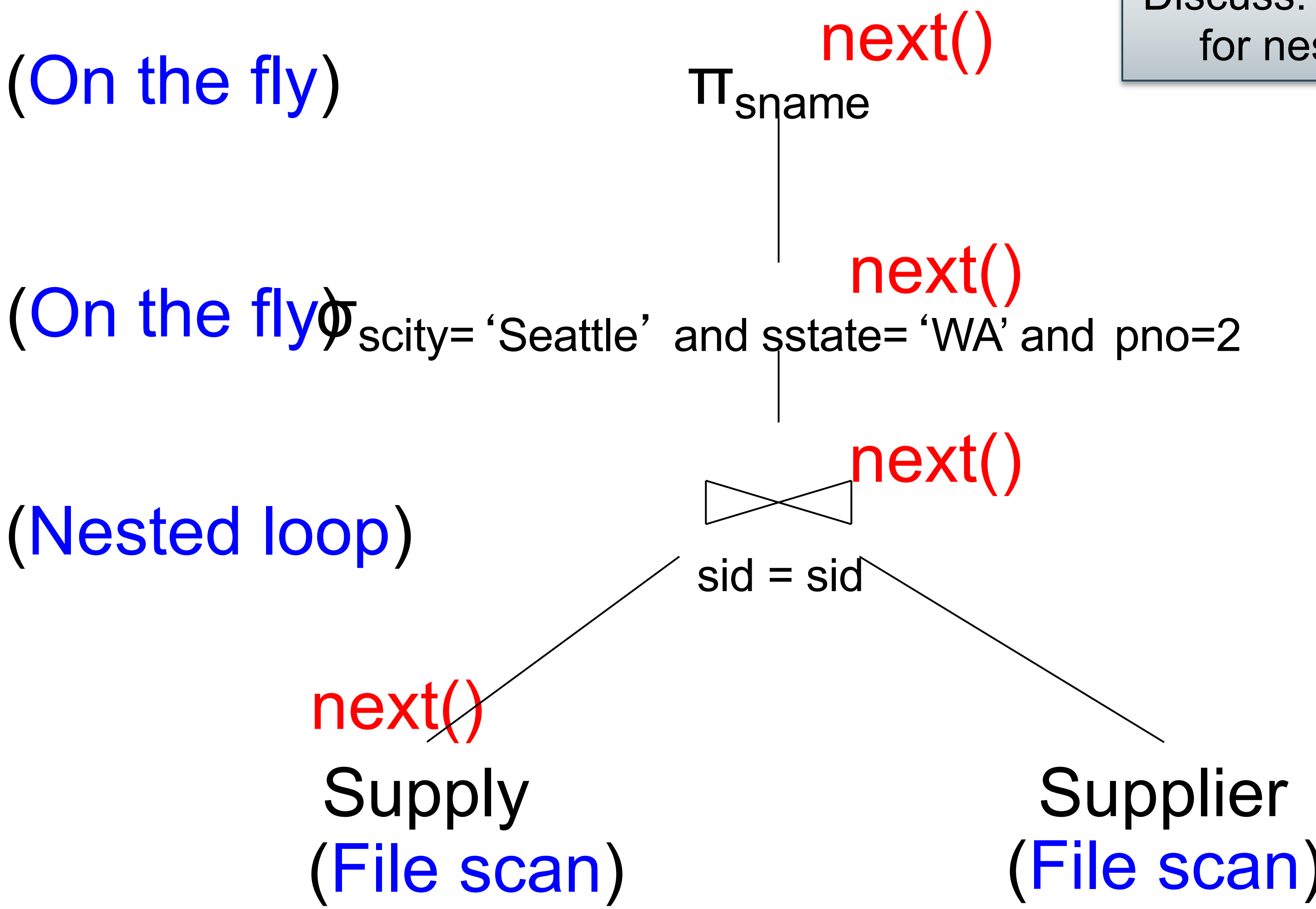
Discuss: open/next/close for nested loop join



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

Pipelining

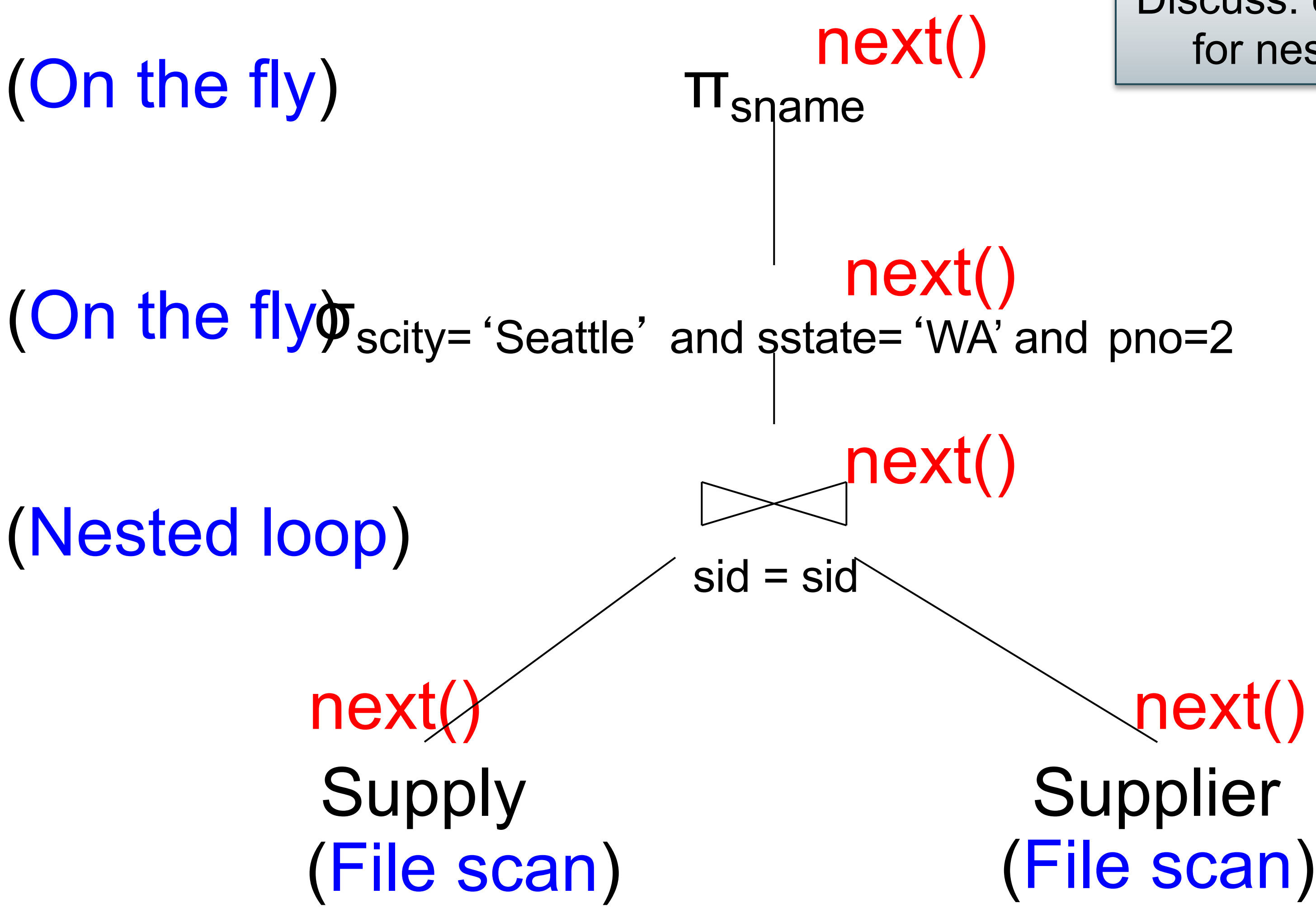
Discuss: open/next/close for nested loop join



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

Pipelining

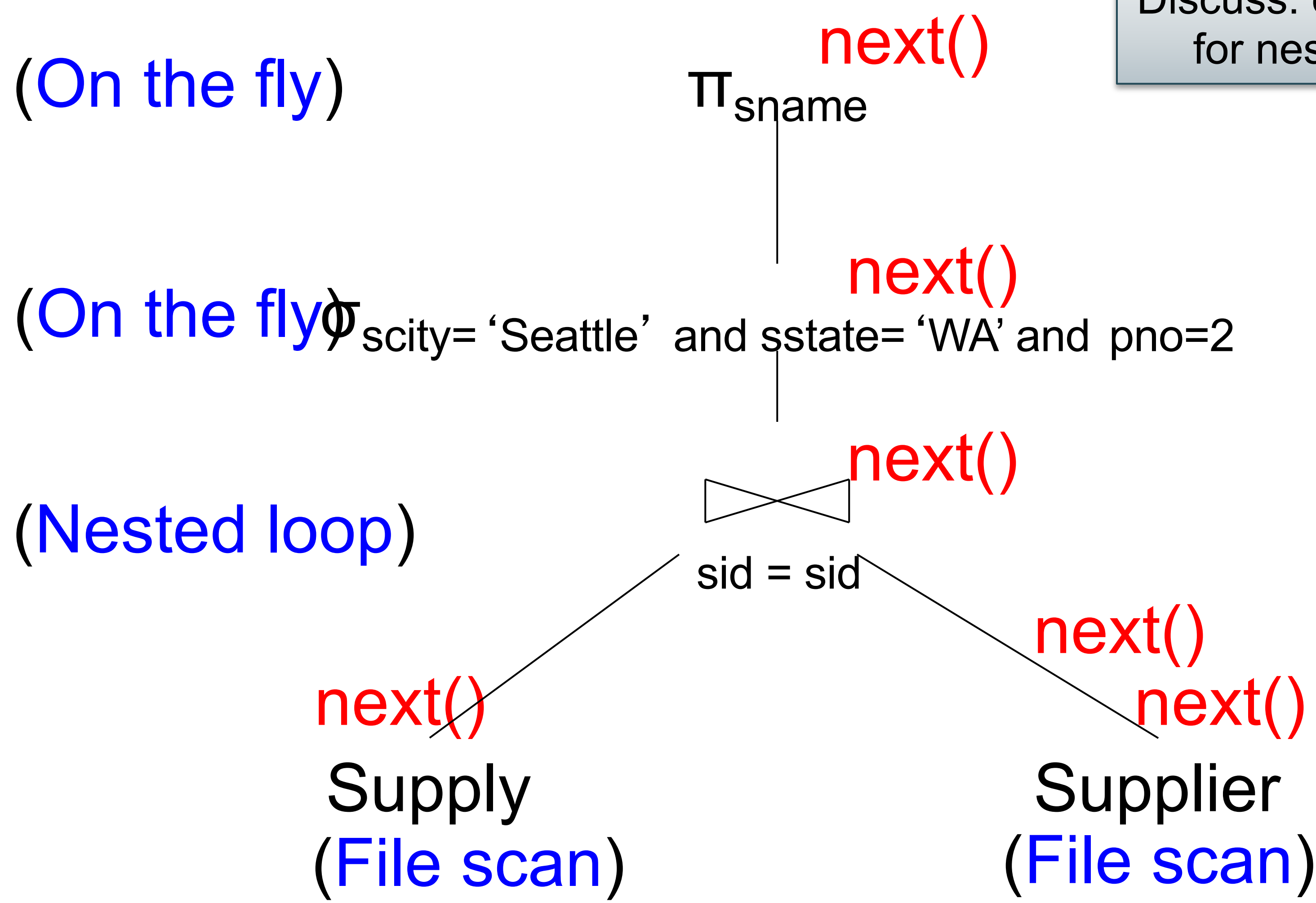
Discuss: open/next/close for nested loop join



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

Pipelining

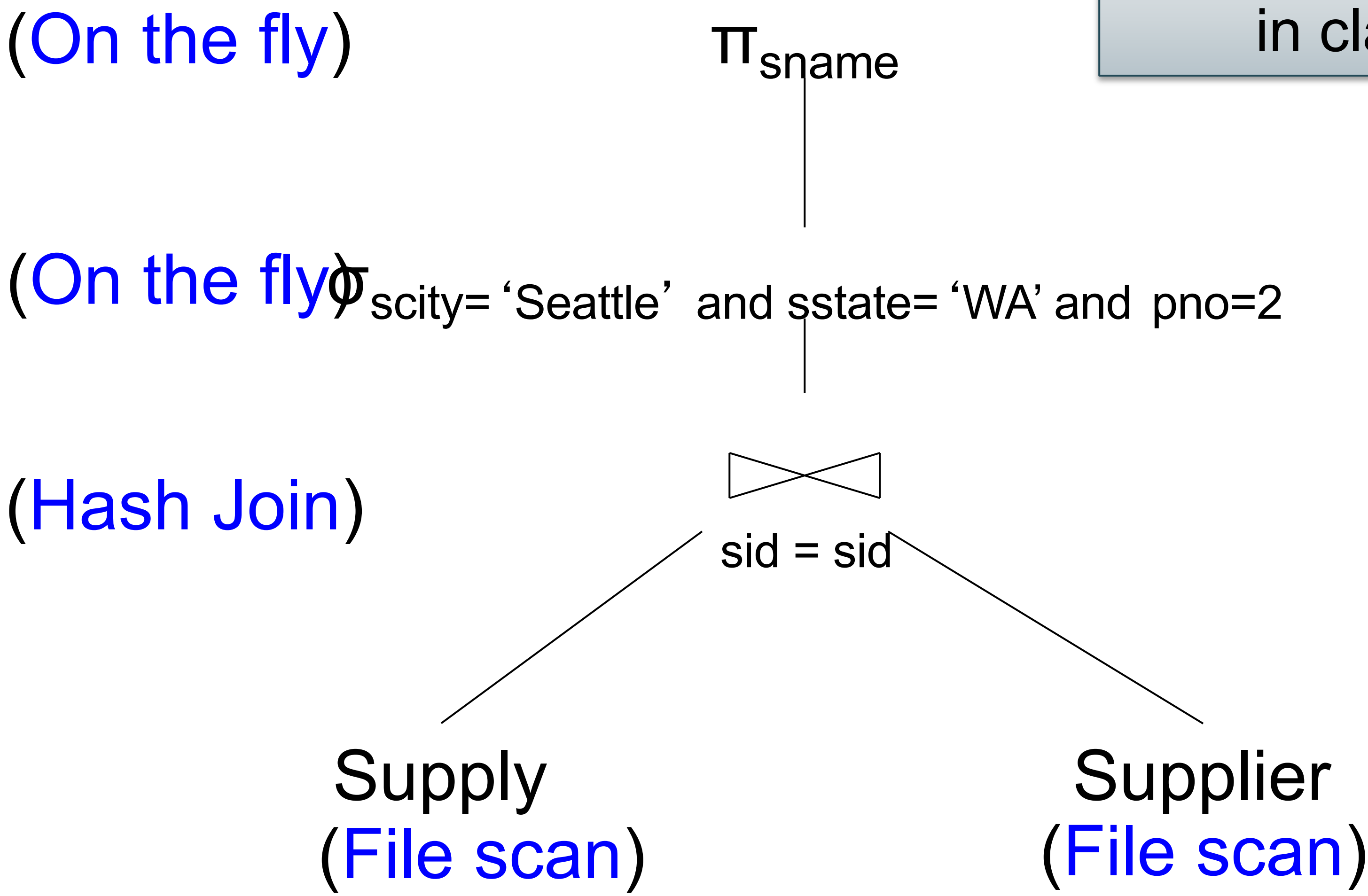
Discuss: open/next/close for nested loop join



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

Pipelining

Discuss hash-join in class



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

Pipelining

Discuss hash-join in class

(On the fly) π_{sname}

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

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

Tuples from here are "blocked"

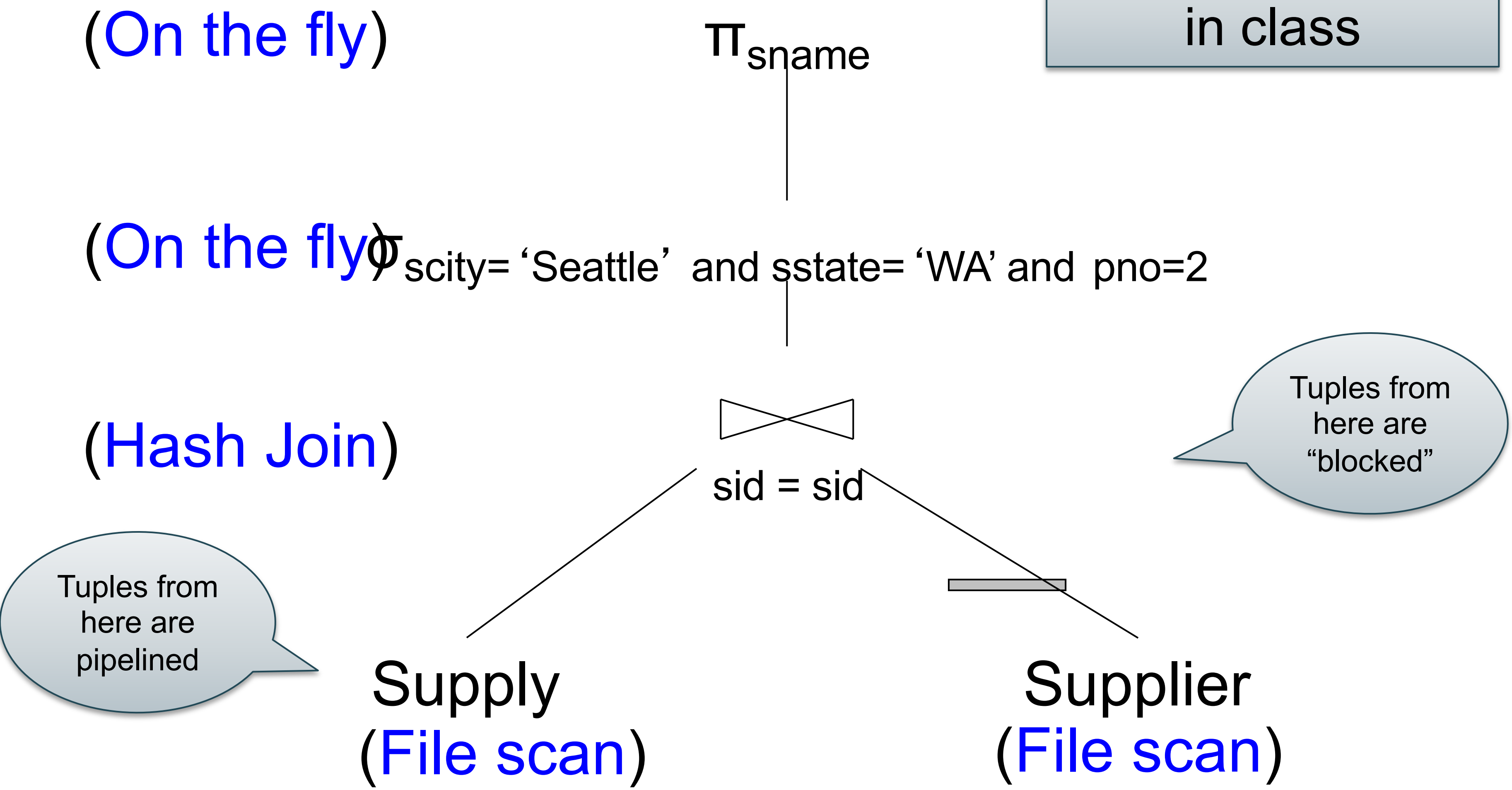
Supply (File scan)

Supplier (File scan)

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

Pipelining

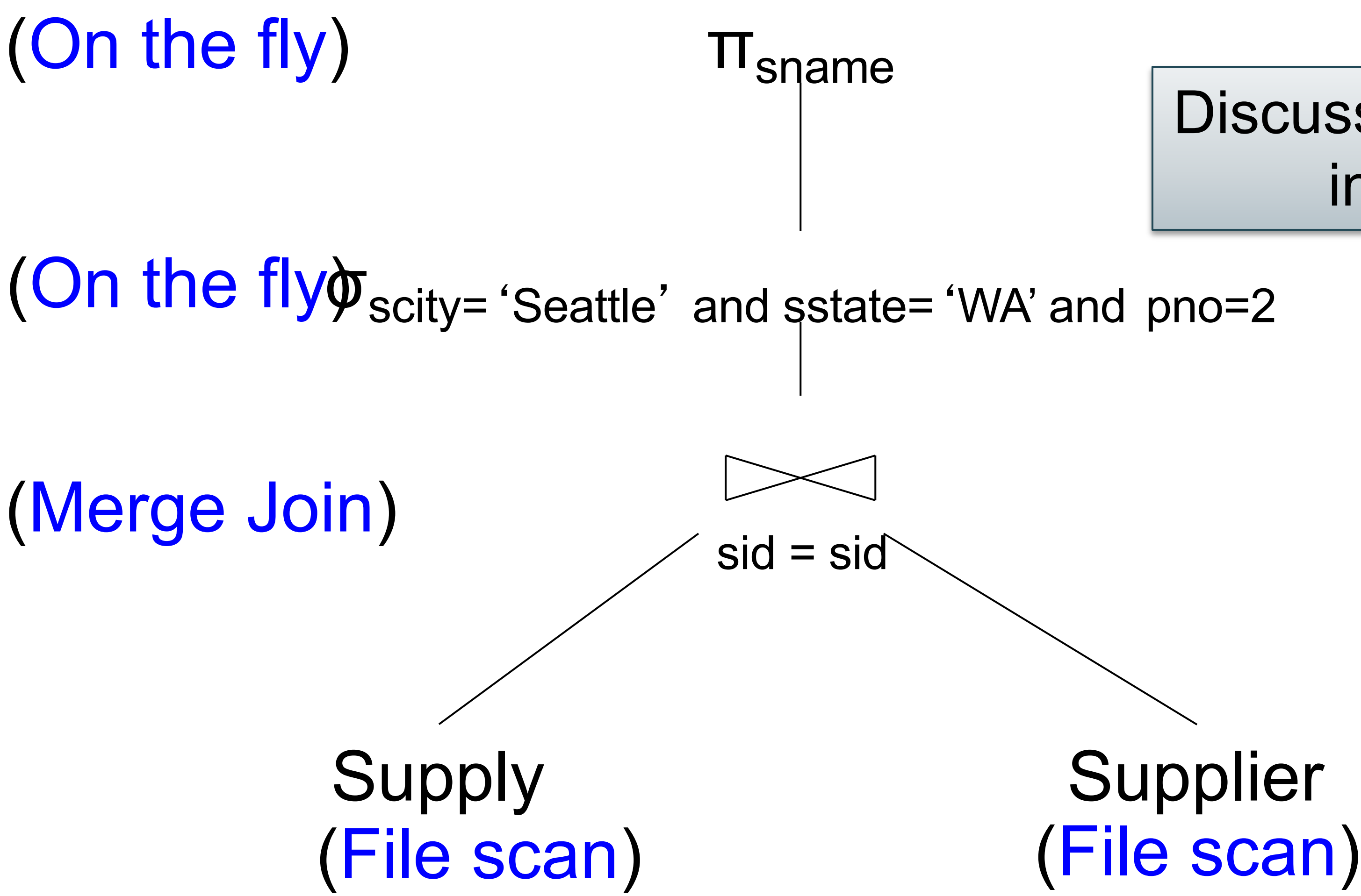
Discuss hash-join in class



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

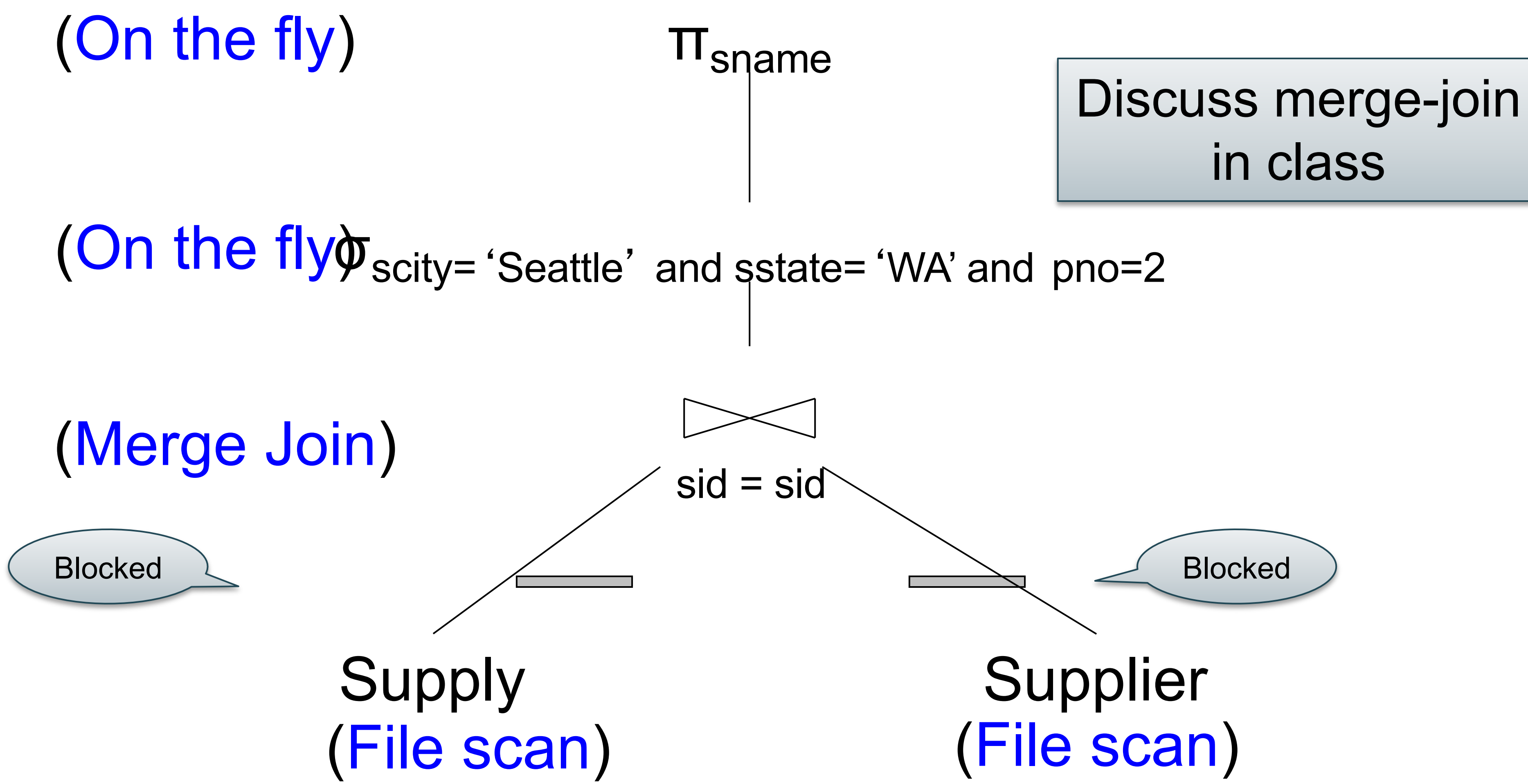
Blocked Execution

Discuss merge-join in class



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

Blocked Execution



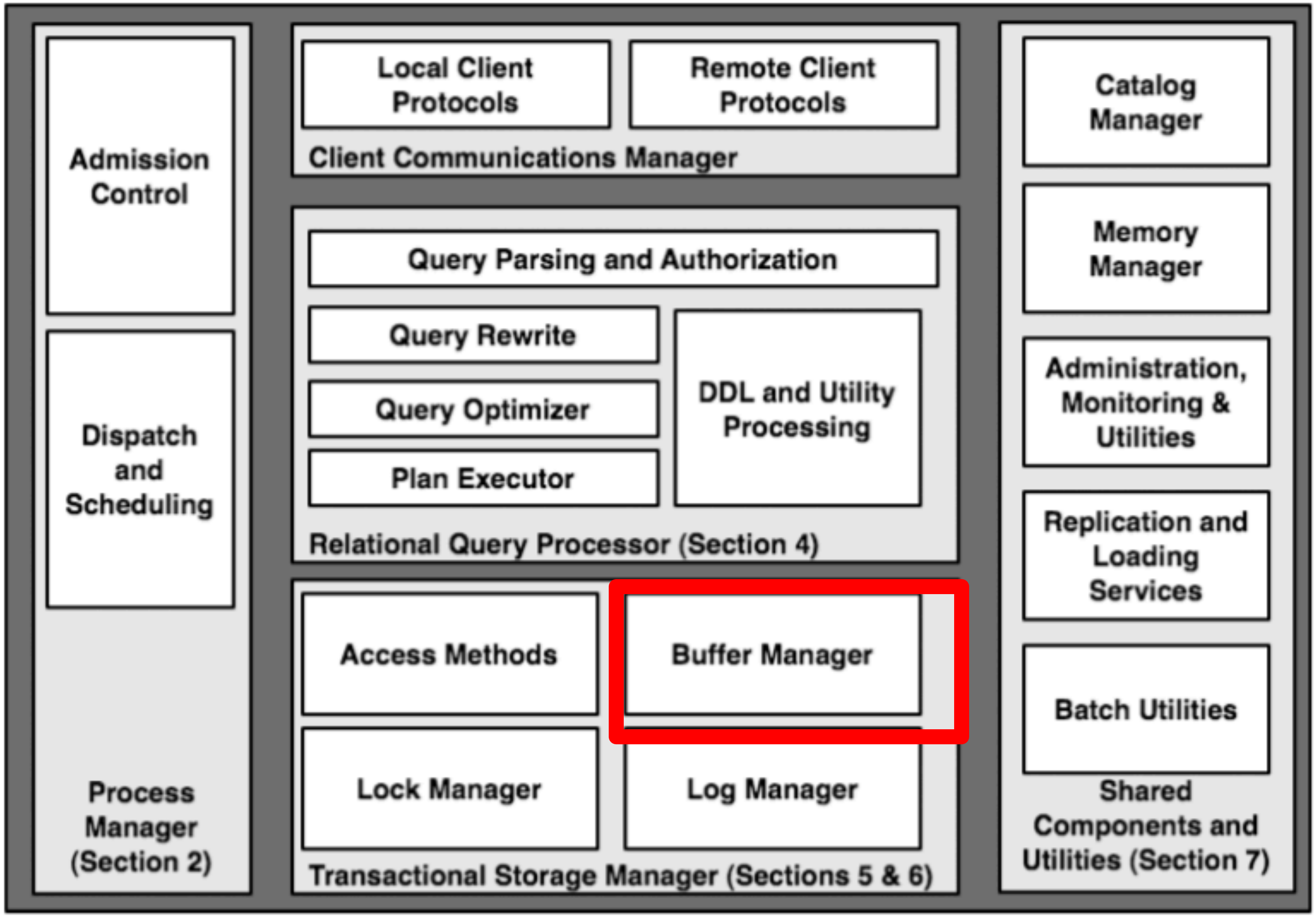
Pipeline v.s. Blocking

- Pipeline
  - A tuple moves all the way through up the query plan
  - Advantages: speed
  - Disadvantage: need all hash tables in memory
- Blocking
  - Compute and store on disk entire subplan
  - Advantage: needs less memory
  - Disadvantage: slower

Outline

- Architecture of a DBMS
- Steps involved in processing a query
- Main Memory Operators
- Storage
- External Memory Operators

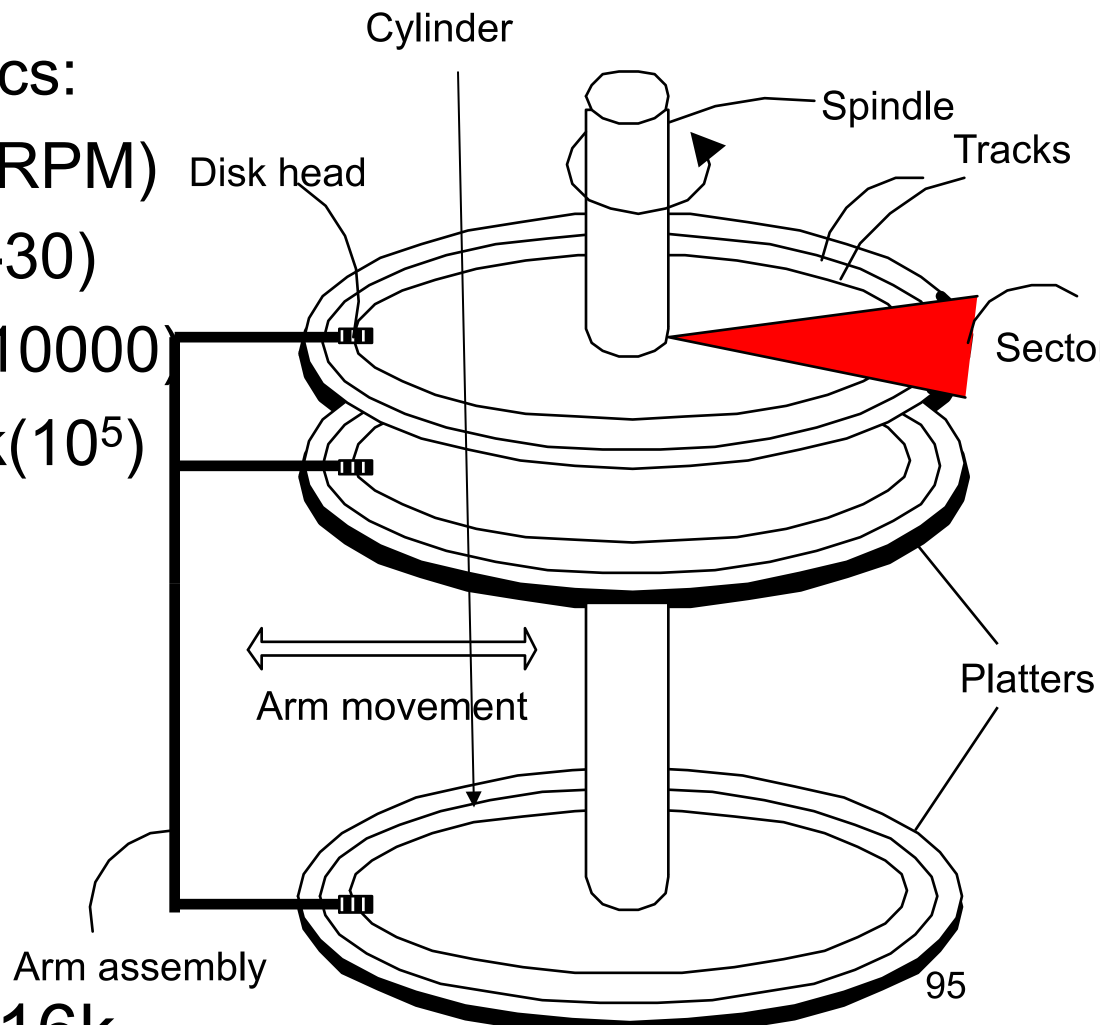
Multiple Processes



The Mechanics of Disk

Mechanical characteristics:

- Rotation speed (5400RPM)
- Number of platters (1-30)
- Number of tracks (<=10000)
- Number of bytes/track(10^5)



Unit of read or write:

disk block

Once in memory:

page

Typically: 4k or 8k or 16k

Data Storage

Student

ID	fName	lName
10	Tom	Hanks
20	Amy	Hanks
…		

- DBMSs store data in **files**
- Most common organization is row-wise storage
- On disk, a file is split into blocks
- Each block contains a set of tuples

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

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

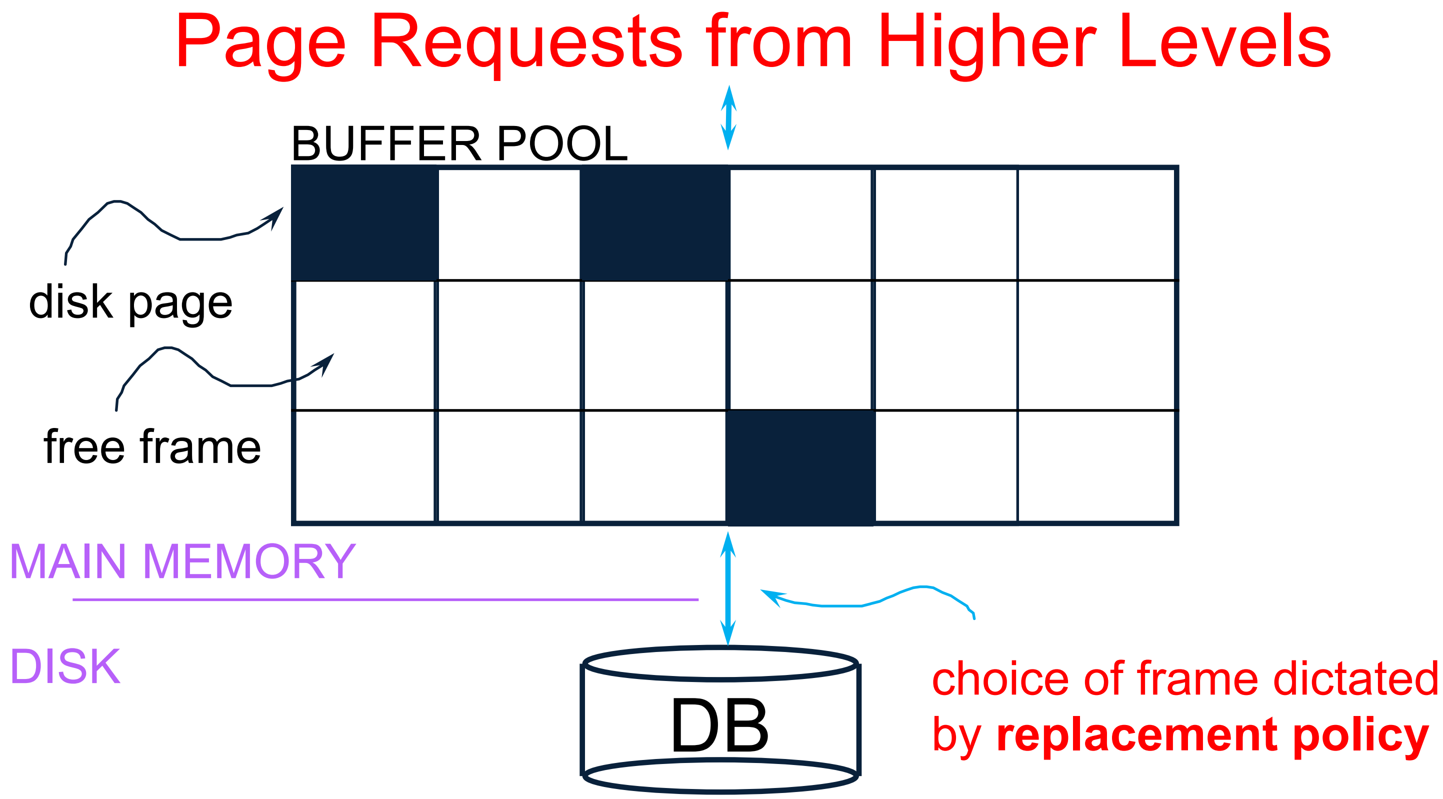
<u>Basic fact</u>: disks always read/write an entire block at a time

Disk Access Characteristics

- Disk latency
  - Time between when command is issued and when data is in memory
  - Equals = seek time + rotational latency
- Seek time = time for the head to reach cylinder
  - 10ms – 40ms
- Rotational latency = time for the sector to rotate
    - Rotation time = 10ms
    - Average latency = 10ms/2
- Transfer time = typically 40MB/s

Basic fact: disks access MUCH slower than main memory

Buffer Manager



- Data must be in RAM for DBMS to operate on it!
- Table of <frame#, pageid> pairs is maintained

Buffer Manager

Needs to decide on page replacement policy

- LRU
- Clock algorithm

Both work well in OS, but not always in DB

Enables the higher levels of the DBMS to assume that the needed data is in main memory.

Arranging Pages on Disk

A disk is organized into blocks (a.k.a. pages)

- blocks on same track, followed by
- blocks on same cylinder, followed by
- blocks on adjacent cylinder

A file should (ideally) consists of sequential blocks on disk, to minimize seek and rotational delay.

For a sequential scan, pre-fetching several pages at a time is a big win!

Storing Records On Disk

- Page format: records inside a page
- Record format: attributes inside a record
- File Organization

Page Format

- 1 page = 1 disk block = fixed size (e.g. 8KB)
- Records:
  - Fixed length
  - Variable length
- Record id = RID
  - Typically RID = (PageID, SlotNumber)

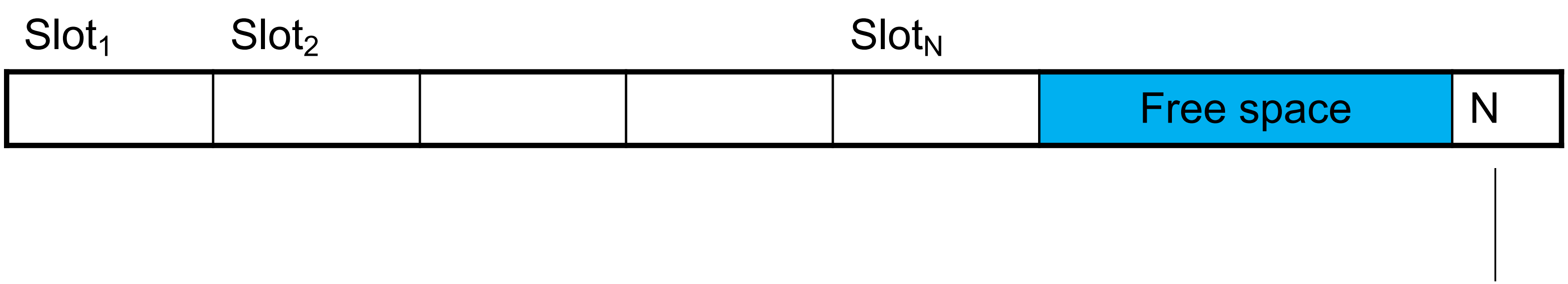
Need RID’s for indexes and for transactions

Page Format Approach 1

Fixed-length records: packed representation

Divide page into **slots**. Each slot can hold one tuple

Record ID (RID) for each tuple is (PageID,SlotNb)



Number of records

How do we insert a new record?

Page Format Approach 1

Fixed-length records: packed representation

Divide page into **slots**. Each slot can hold one tuple

Record ID (RID) for each tuple is (PageID,SlotNb)

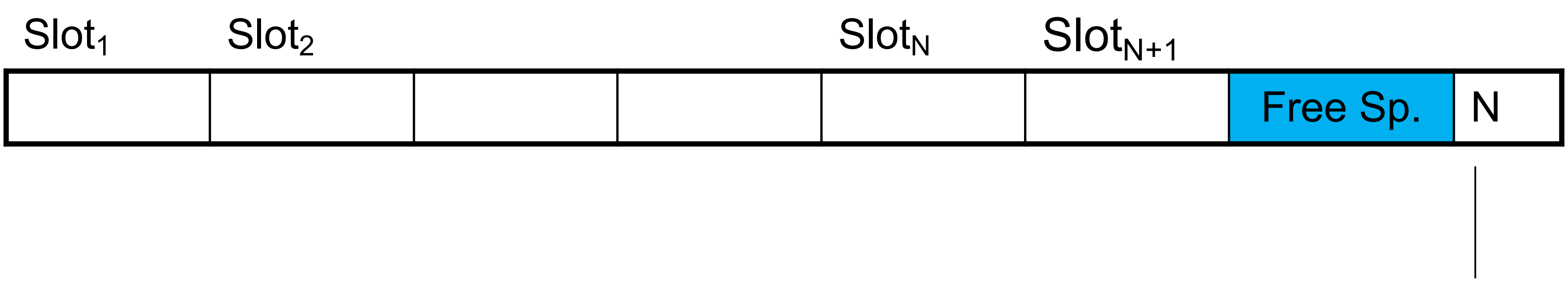
How do we insert a new record?

Page Format Approach 1

Fixed-length records: packed representation

Divide page into **slots**. Each slot can hold one tuple

Record ID (RID) for each tuple is (PageID,SlotNb)

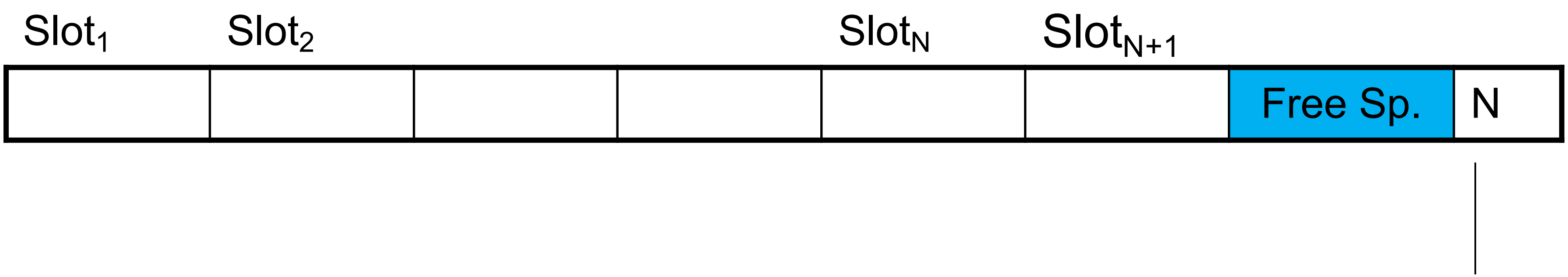


How do we insert a new record?

How do we delete a record?

Page Format Approach 1

Fixed-length records: packed representation
Divide page into **slots**. Each slot can hold one tuple
Record ID (RID) for each tuple is (PageID,SlotNb)



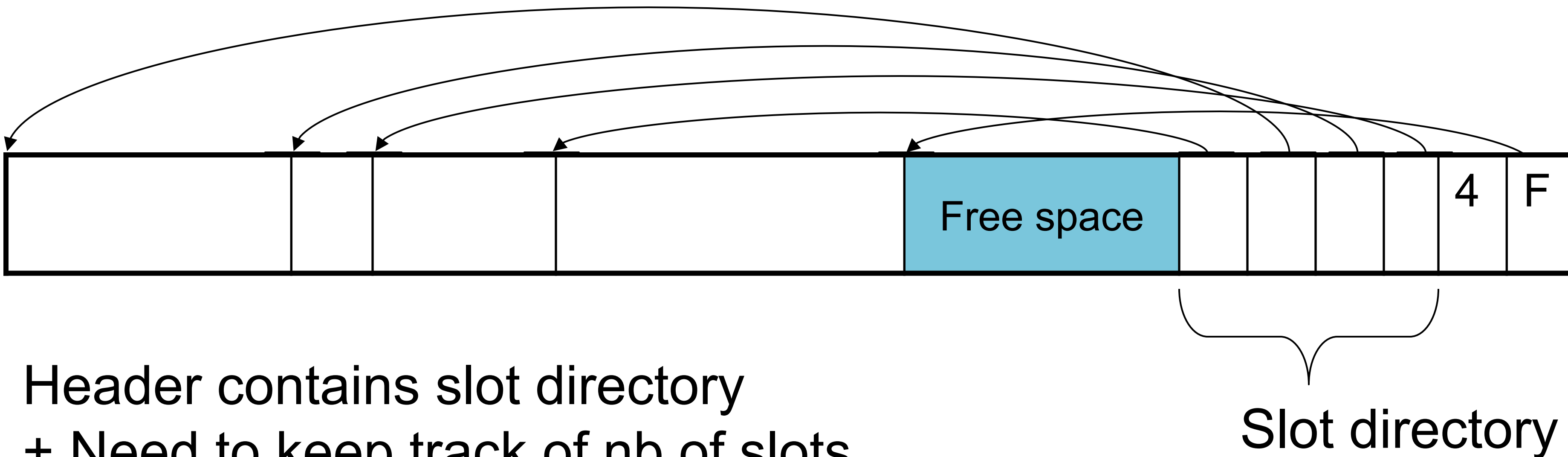
Number of records

How do we insert a new record?

How do we delete a record? Cannot remove record (why?)

How do we handle variable-length records?

Page Format Approach 2



Header contains slot directory
+ Need to keep track of nb of slots
+ Also need to keep track of free space (F)

Can handle variable-length records
Can move tuples inside a page without changing RIDs
RID is (PageID, SlotID) combination

Record Formats

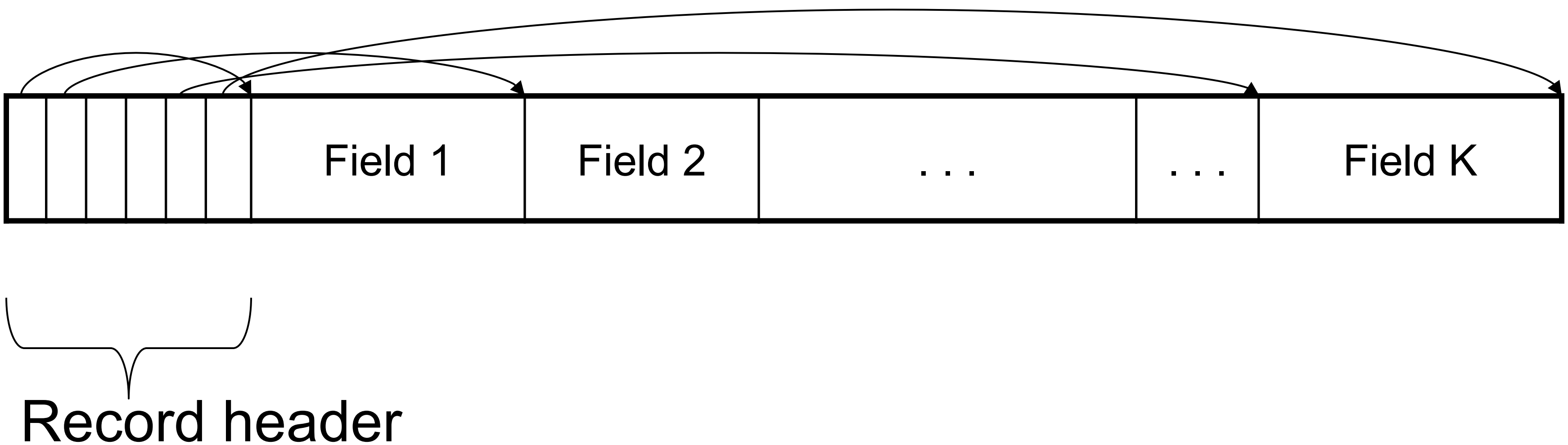
Fixed-length records => Each field has a fixed length (i.e., it has the same length in all the records)

Field 1	Field 2	. . .	. . .	Field K

Information about field lengths and types is in the catalog

Record Formats

Variable length records



Remark: NULLS require no space at all (why ?)

Announcements

- Project teams were due last week
- PAX paper review due on Wednesday
- HW2 Snowflake due on Friday

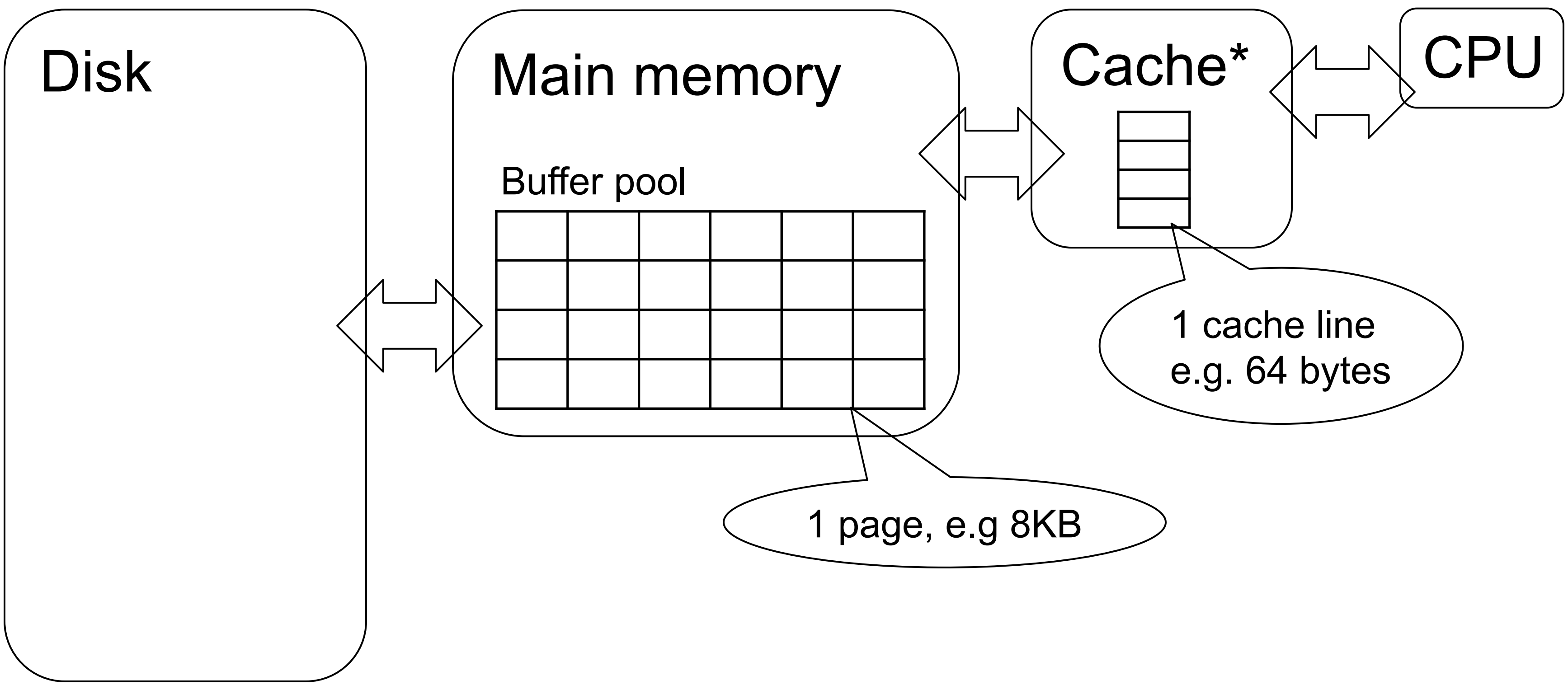
Quick Review

- What is the unit of access for RAM*?
  What is the unit of access for disk?
  Why the difference?
- What is the Buffer Pool?
- Describe how a table is stored on disk

*RAM = Random Access memory = main memory

Notes for the PAX paper

Memory hierarchies:



*aka CPU cache; several! L3, L2, L1 cache

File Organizations

- Heap (random order) files: Suitable when typical access is a file scan retrieving all records.
- Sequential file (sorted): Best if records must be retrieved in some order, or by a `range'
- Index: Data structures to organize records via trees or hashing.

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
- Could have many indexes for one table

Key = means here search key

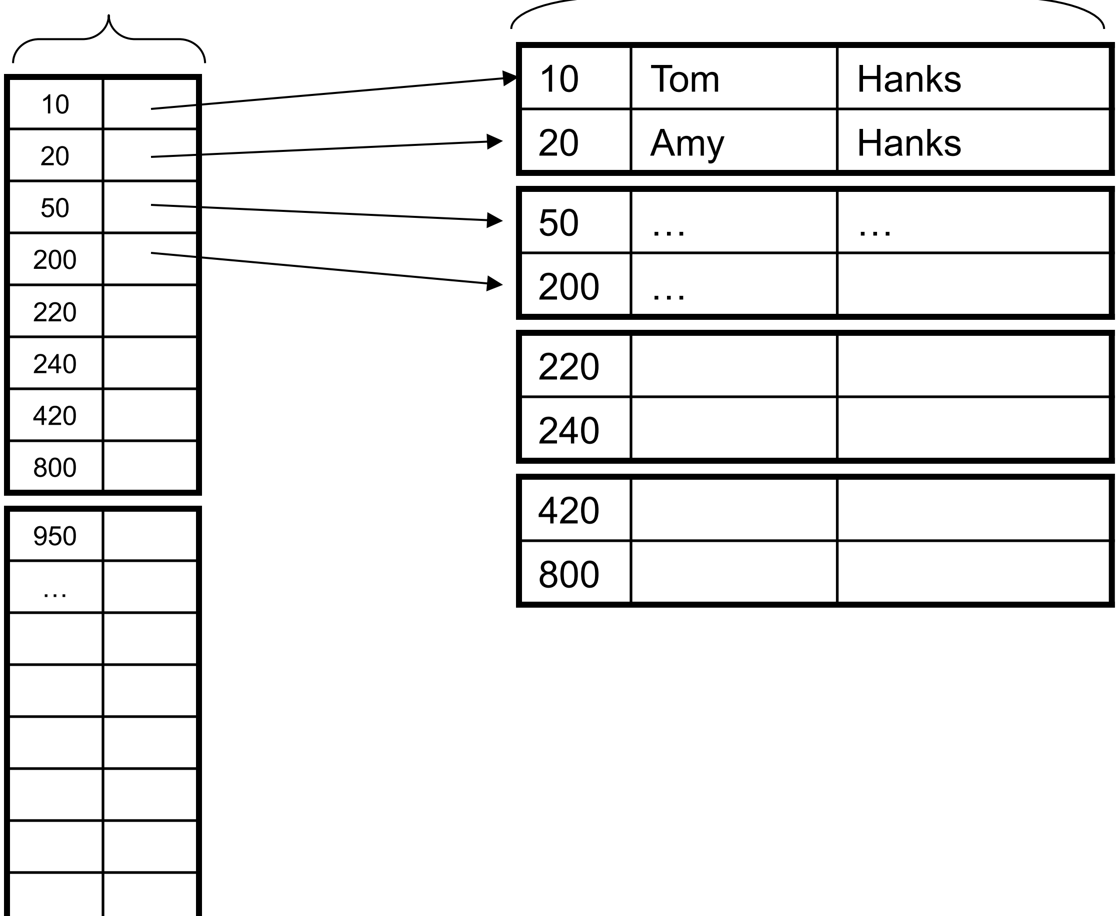
Example 1: Index on ID

Actor

ID	fName	lName
10	Tom	Hanks
20	Amy	Hanks
…		

Index **Actor_ID** on **Actor.ID**

Data File **Actor**



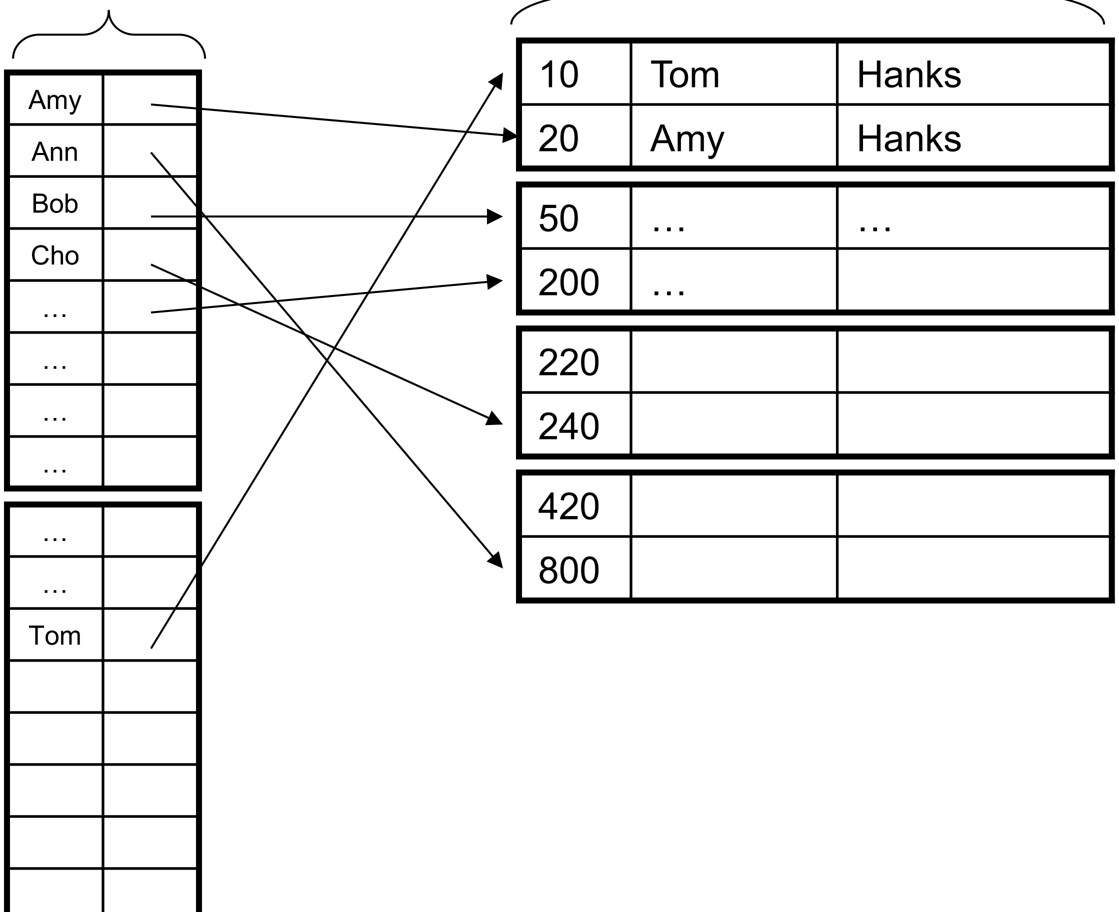
Example 2: Index on fName

Actor

ID	fName	lName
10	Tom	Hanks
20	Amy	Hanks
…		

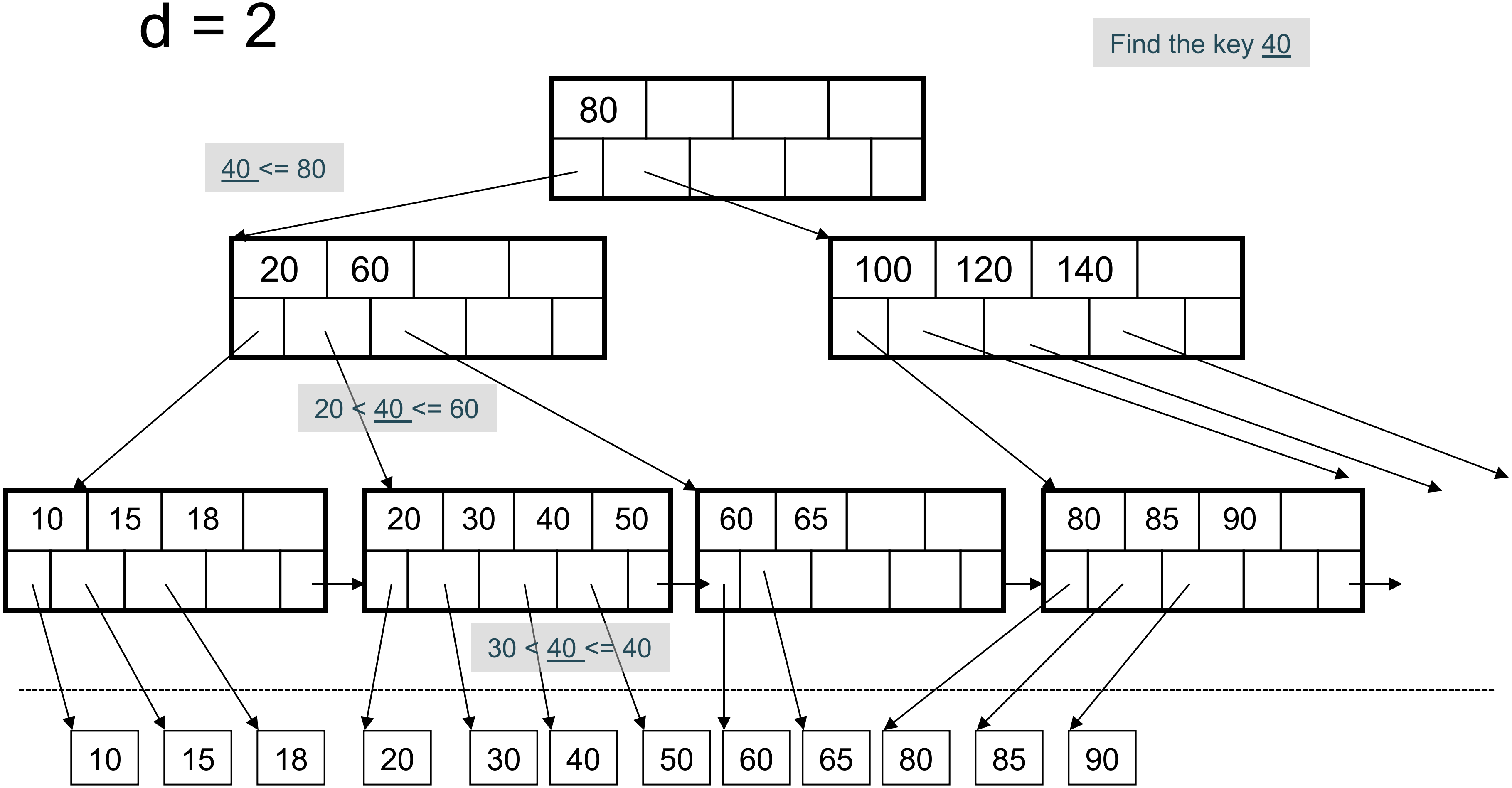
Index **Actor_fName** on **Actor.fName**

Data File **Actor**

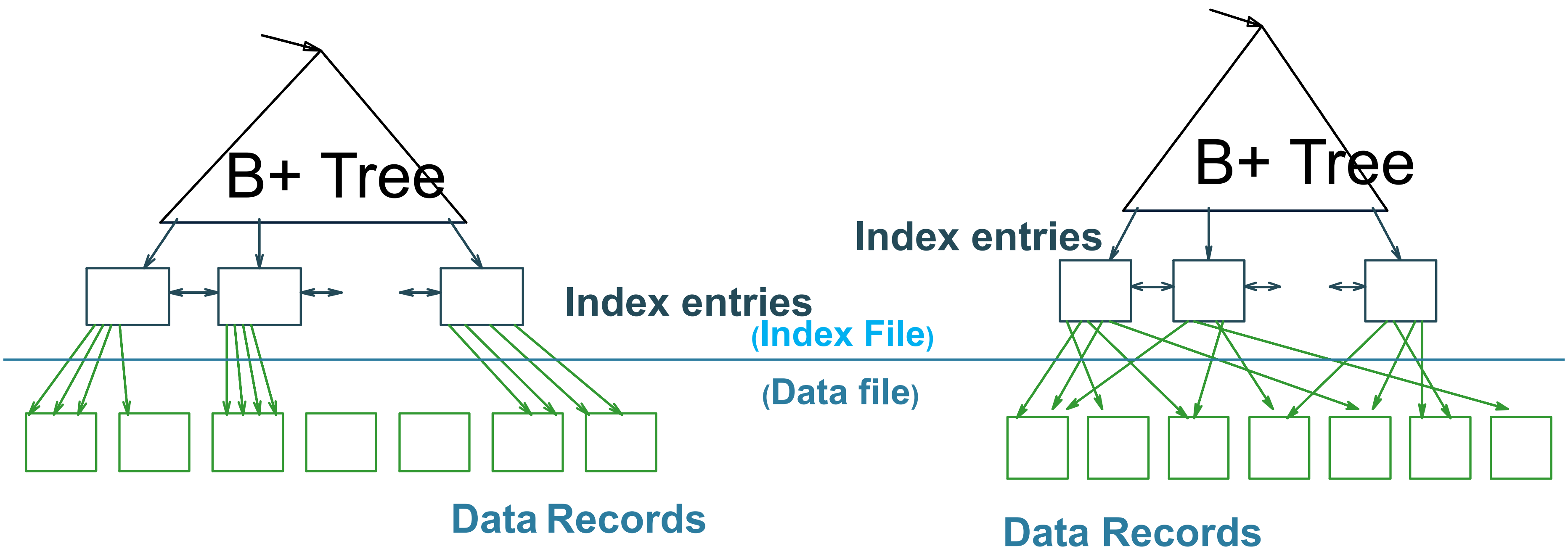


B+ Tree Index by Example

d = 2



Clustered vs Unclustered



CLUSTERED **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

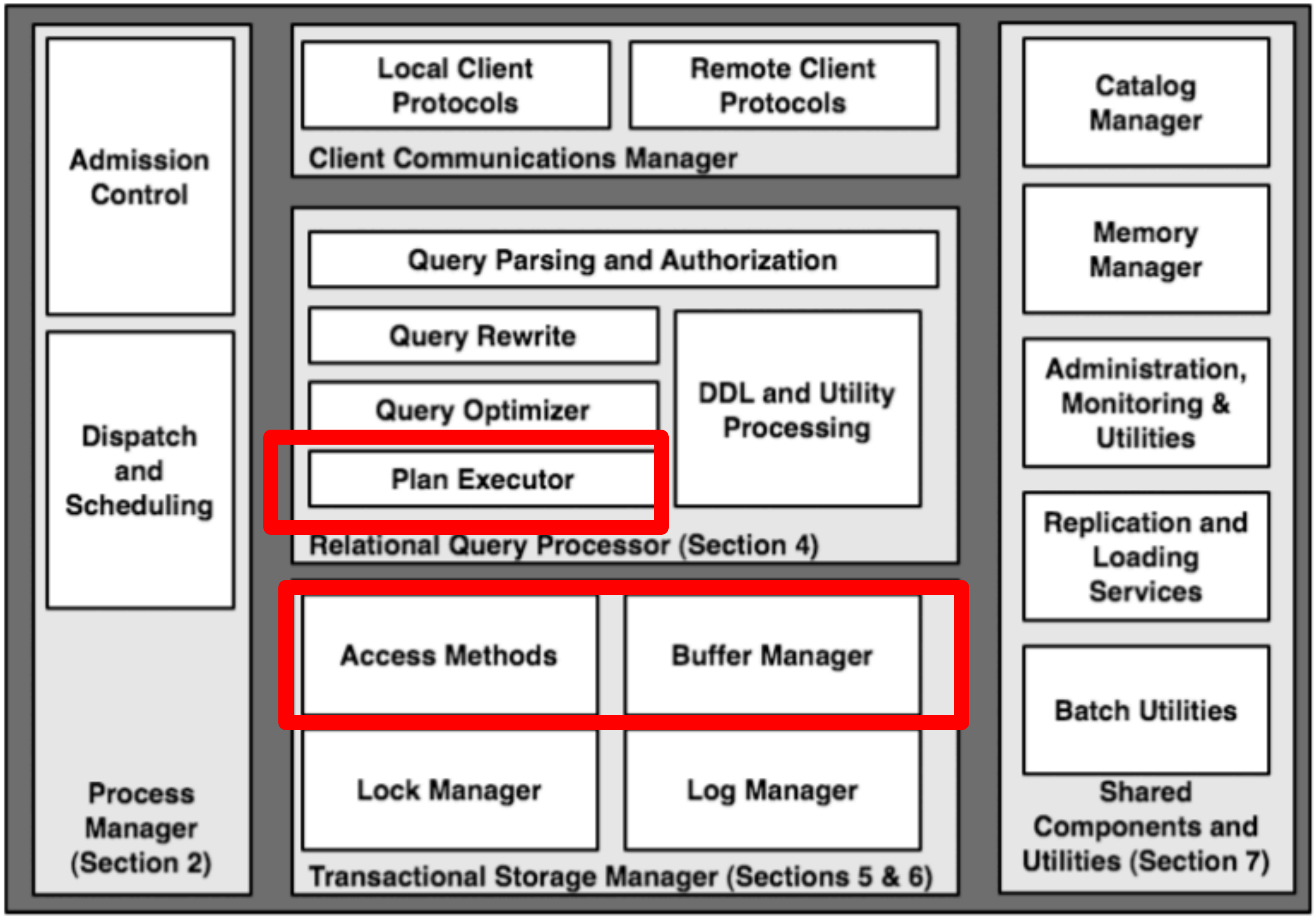
Discussion on Indices

- What they do: speed up disk access
  What they don't: speed up RAM algo.
- They benefit only SELECT queries that have some predicate A=… or A<=…
- They hurt *<u>all</u>* INSERT/UPDATE/DELETE queries (why?)

Outline

- Architecture of a DBMS
- Steps involved in processing a query
- Main Memory Operators
- Storage
- External Memory Operators

Architecture



Cost Parameters

- In database systems the data is on disk
- Parameters:
  - 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
  - M = # pages available in main memory
- Cost = total number of I/Os
- Convention: writing the final result to disk is *not included*

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

Cost Parameters

Supplier(sid,sname,scity,sstate)

Block size = 8KB

- B(Supplier) = 1,000,000 blocks = 8GB
- T(Supplier) = 50,000,000 records ~ 50 / block
- V(Supplier, sid) =
- V(Supplier, sname) =
- V(Supplier, scity) =
- V(Supplier, sstate) =
- M =

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

Cost Parameters

Supplier(sid,sname,scity,sstate)

Block size = 8KB

- B(Supplier) = 1,000,000 blocks = 8GB
- T(Supplier) = 50,000,000 records ~ 50 / block
- V(Supplier, sid) = 50,000,000 why?
- V(Supplier, sname) =
- V(Supplier, scity) =
- V(Supplier, sstate) =
- M =

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

Cost Parameters

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

Block size = 8KB

- B(Supplier) = 1,000,000 blocks = 8GB
- T(Supplier) = 50,000,000 records ~ 50 / block
- V(Supplier, sid) = 50,000,000 why?
- V(Supplier, sname) = 40,000,000 meaning?
- V(Supplier, scity) =
- V(Supplier, sstate) =
- M =

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

Cost Parameters

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

Block size = 8KB

- B(Supplier) = 1,000,000 blocks = 8GB
- T(Supplier) = 50,000,000 records ~ 50 / block
- V(Supplier, sid) = 50,000,000 why?
- V(Supplier, sname) = 40,000,000 meaning?
- V(Supplier, scity) = 860
- V(Supplier, sstate) =
- M =

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

Cost Parameters

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

Block size = 8KB

- B(Supplier) = 1,000,000 blocks = 8GB
- T(Supplier) = 50,000,000 records ~ 50 / block
- V(Supplier, sid) = 50,000,000 why?
- V(Supplier, sname) = 40,000,000 meaning?
- V(Supplier, scity) = 860
- V(Supplier, sstate) = 50 why?
- M =

Cost Parameters

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

Block size = 8KB

• B(Supplier) = 1,000,000 blocks	= 8GB
• T(Supplier) = 50,000,000 records	~ 50 / block
• V(Supplier, sid) = 50,000,000	why?
• V(Supplier, sname) = 40,000,000	meaning?
• V(Supplier, scity) = 860	
• V(Supplier, sstate) = 50	why?
• M = 10,000,000 = 80GB	why so little?

```
SELECT *
FROM R
WHERE R.a = v
```

Index Based Selection

Selection on equality: $\sigma_{a=v}(R)$

V(R, a) = # of distinct values of attribute a

Cost of index-based selection:

- Clustered index on a:
- Unclustered index on a:

```
SELECT *
FROM R
WHERE R.a = v
```

Index Based Selection

Selection on equality: $\sigma_{a=v}(R)$

V(R, a) = # of distinct values of attribute a

Assumptions:

- Values are uniformly distributed
- Ignore the cost of reading the index (why?)

Cost of index-based selection:

- Clustered index on a:
- Unclustered index on a:

```
SELECT *
FROM R
WHERE R.a = v
```

Index Based Selection

Selection on equality: $\sigma_{a=v}(R)$

V(R, a) = # of distinct values of attribute a

Assumptions:

- Values are uniformly distributed
- Ignore the cost of reading the index (why?)

Cost of index-based selection:

- Clustered index on a: cost = B(R) / V(R,a)
- Unclustered index on a: cost = T(R) / V(R,a)

```
SELECT *
FROM R
WHERE R.a = v
```

Index Based Selection

- Example:

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

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

- Table scan (assuming R is clustered)
- Index based selection
  - If index is clustered:
  - If index is unclustered:

```
SELECT *
FROM R
WHERE R.a = v
```

Index Based Selection

- Example:

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

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

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

```
SELECT *
FROM R
WHERE R.a = v
```

Index Based Selection

- Example:

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

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

- Table scan (assuming R is clustered)
  - B(R) = 2,000 I/Os
- Index based selection
  - If index is clustered: B(R)/V(R,a) = 100 I/Os
  - If index is unclustered:

```
SELECT *
FROM R
WHERE R.a = v
```

Index Based Selection

- Example:

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

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

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

SELECT *
FROM R
WHERE R.a = v

Index Based Selection

- Example:

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

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

The 2% rule!

- Table scan (assuming R is clustered)
  - B(R) = 2,000 I/Os
- Index based selection
  - If index is clustered: B(R)/V(R,a) = 100 I/Os
  - If index is unclustered: T(R)/V(R,a) = 5,000 I/Os
- Lesson
  - Don't build unclustered indexes when V(R,a) is small !

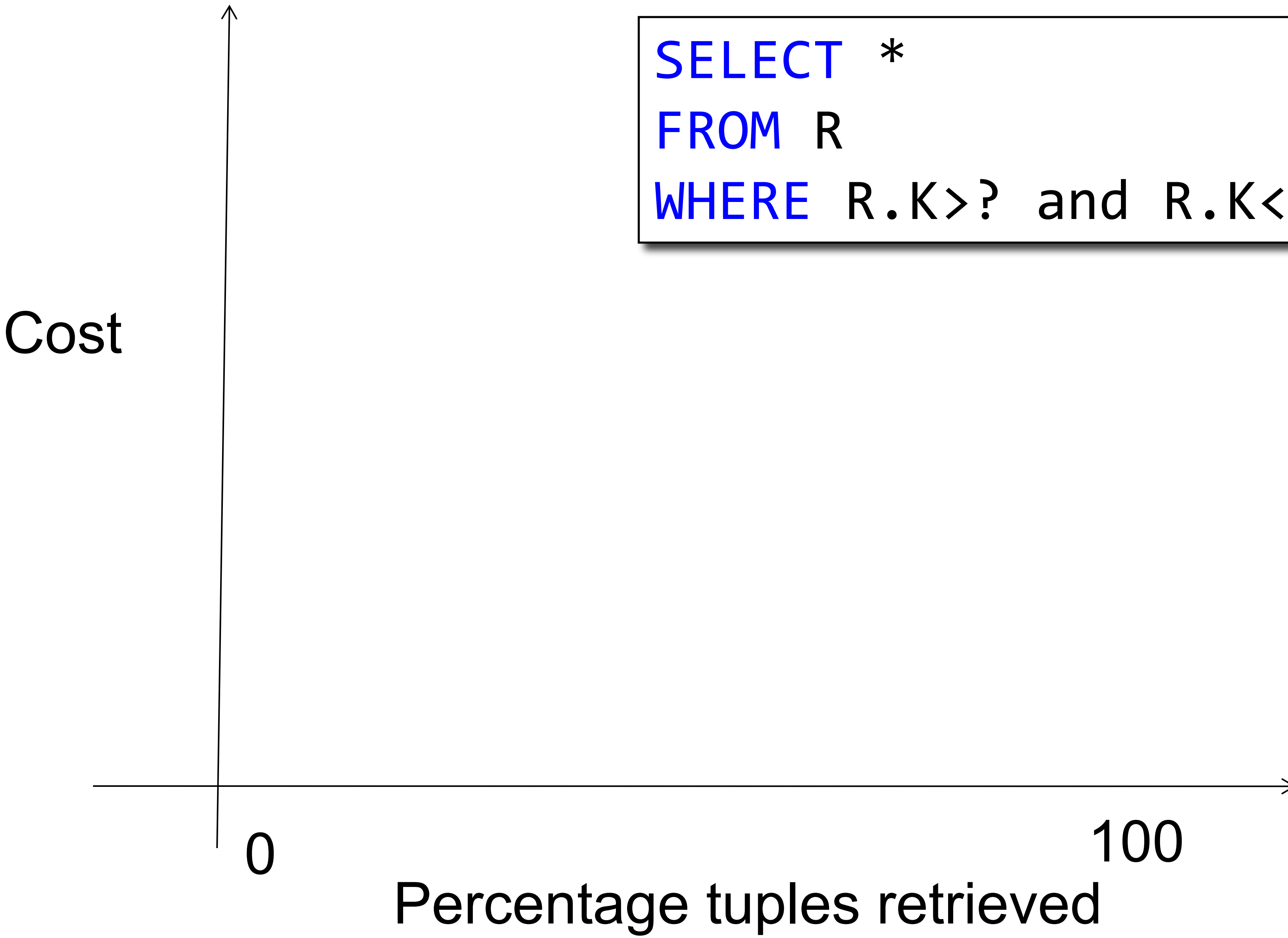
To Cluster or Not

Remember:

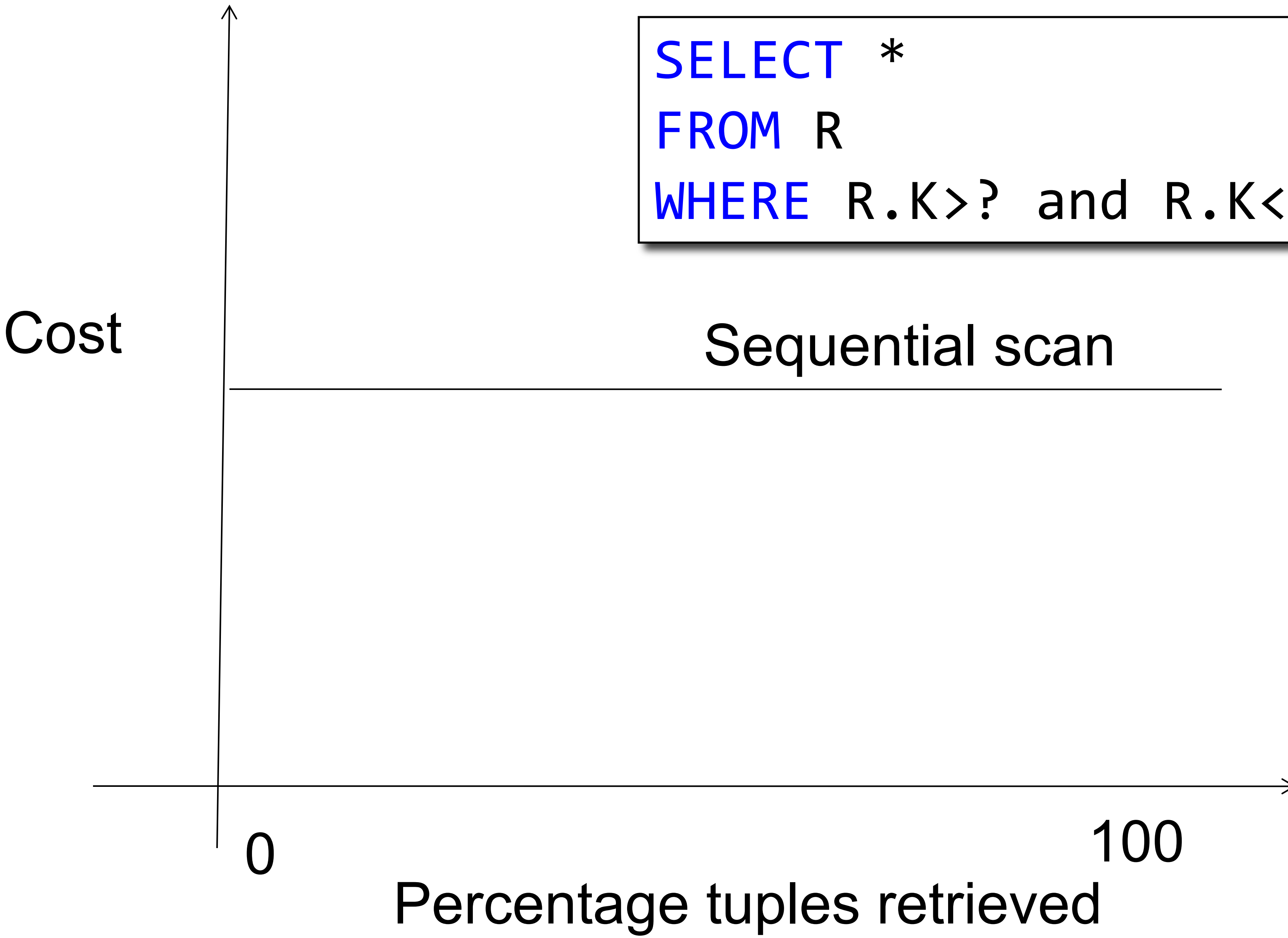
- Rule of thumb:
  Random reading 1-2% of file ≈ sequential scan entire file;

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

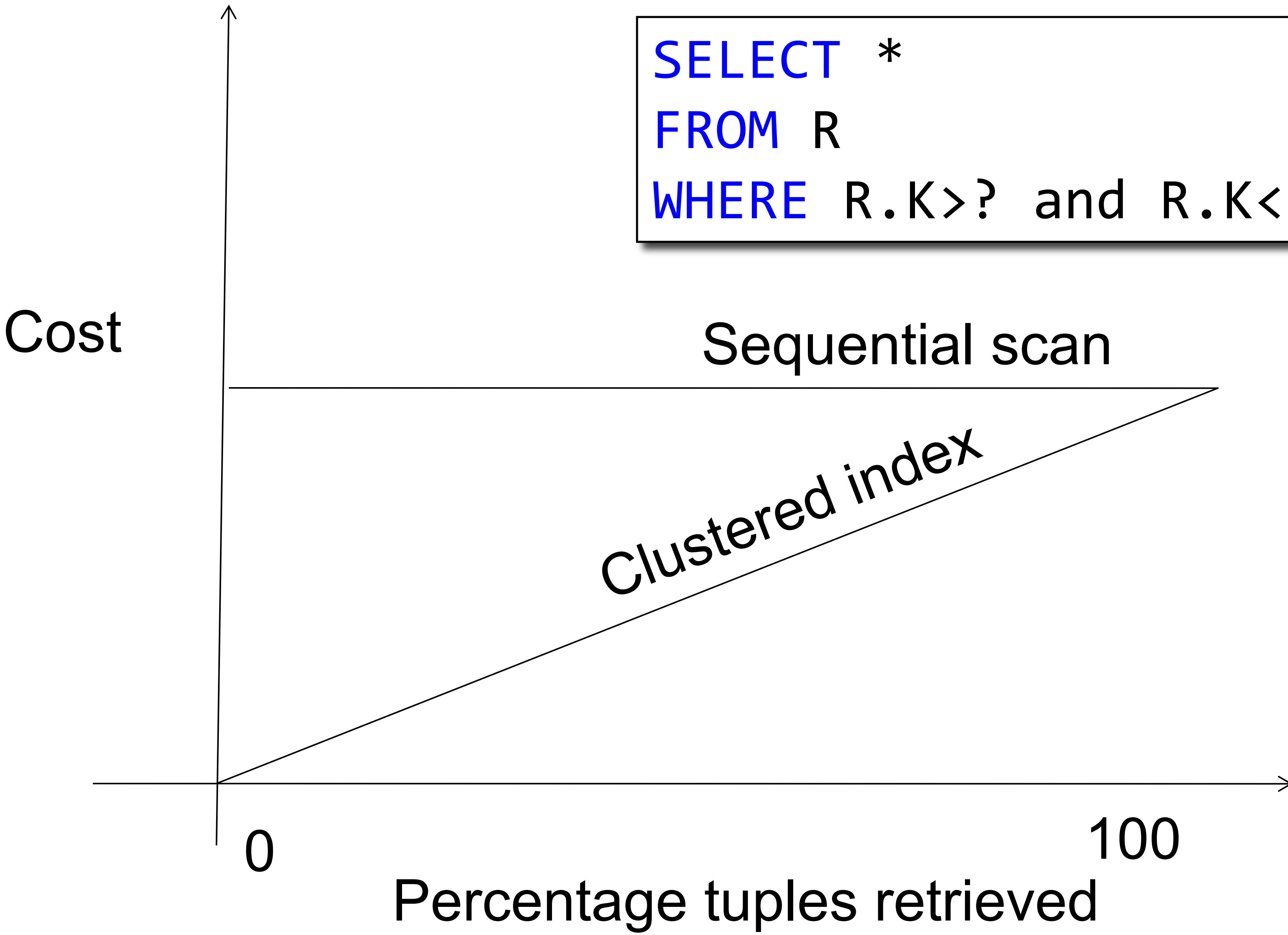
```
SELECT *
FROM R
WHERE R.K>? and R.K<?
```



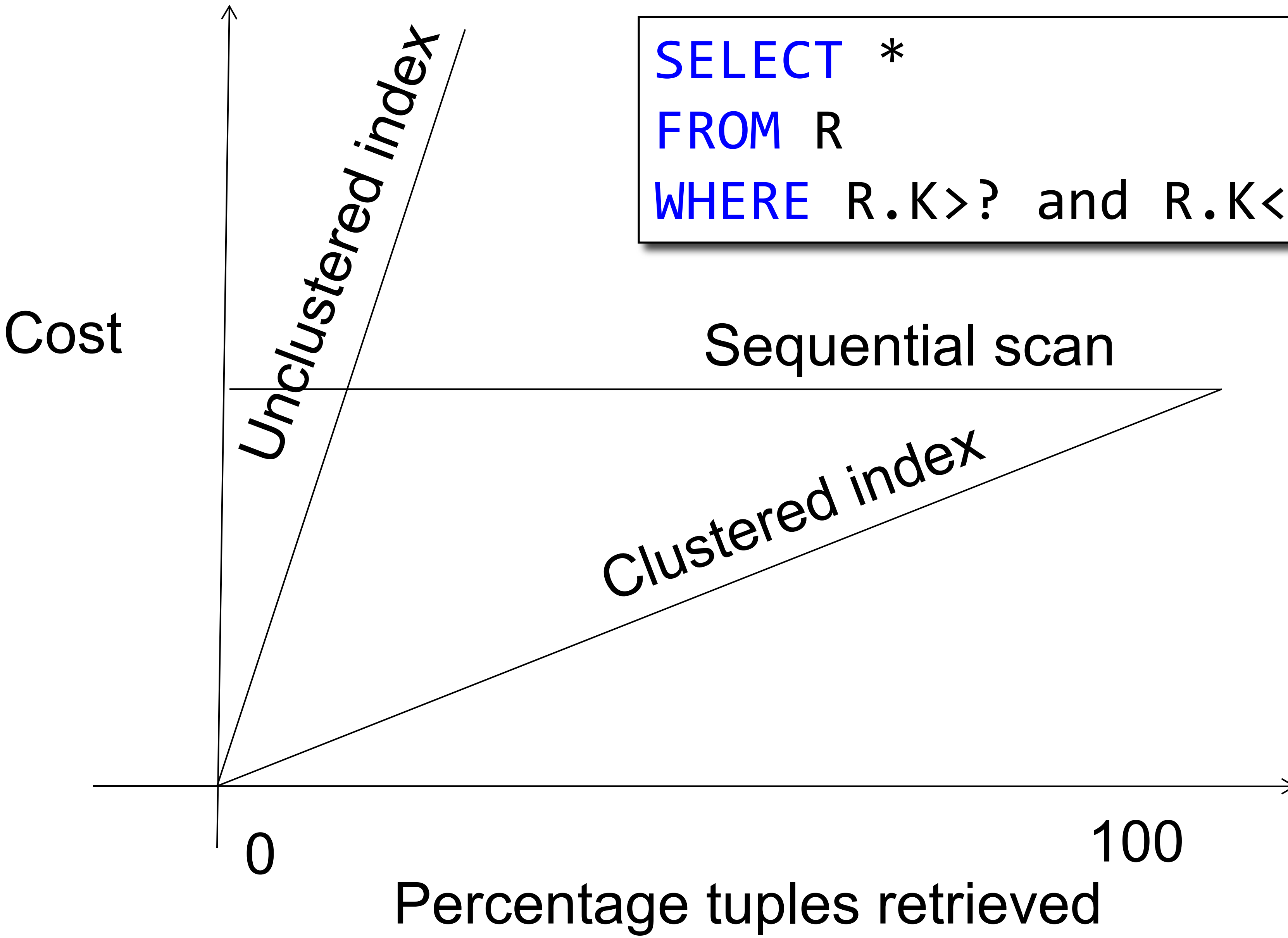
```
SELECT *
FROM R
WHERE R.K>? and R.K<?
```



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



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



External Memory Joins

- Nested loop join
- Index join, a.k.a. index nested loop join
- Partitioned hash-join, a.k.a. grace join
- Merge-join

Nested Loop Joins

- R ⋈ S

- Naïve nested loop joint: T(R) * B(S) I/Os? WHY?
  Of course, can switch order: B(R) * T(S)

- We can be much cleverer by using the available main memory: M

- Assume |R| >> |S|. (Outer relation is bigger than inner relation)

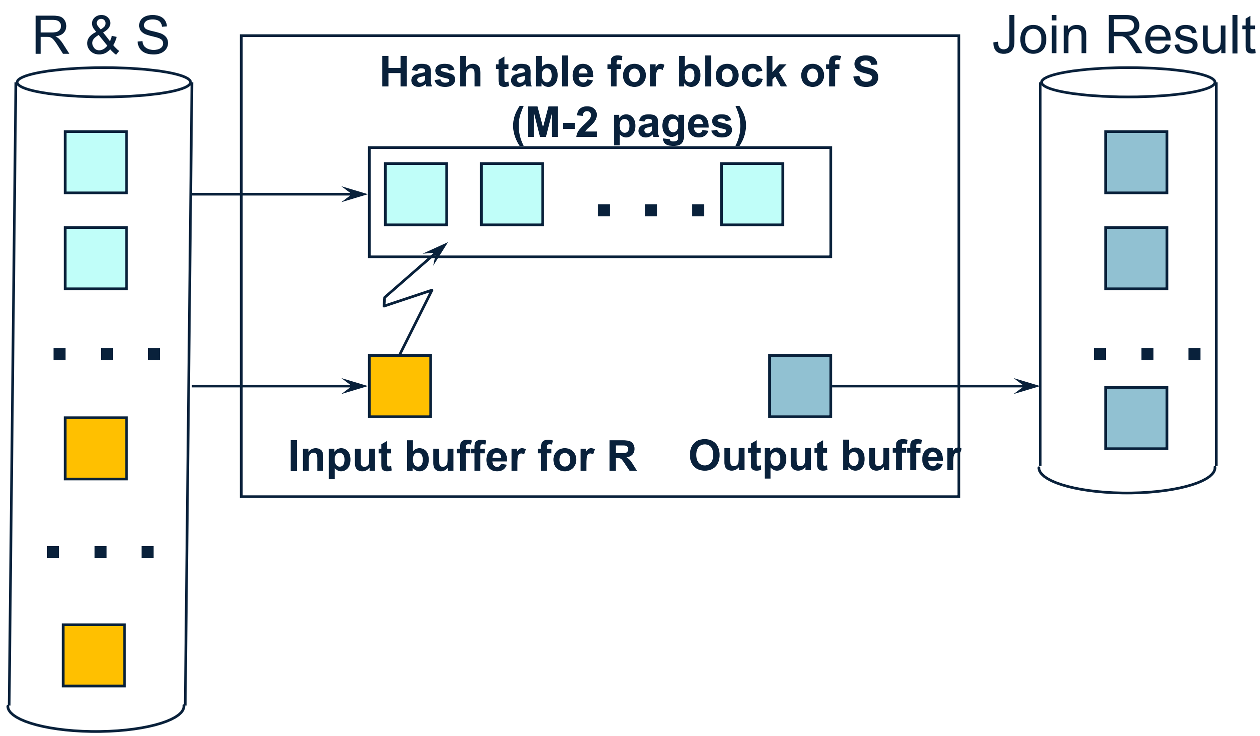
Block Nested Loop Join

- Group of (M-2) pages of S is called a “block”

```
for each (M-2) pages ps of S do
   for each page pr of R do
      for each tuple s in ps
         for each tuple r in pr do
            if r and s join then output(r,s)
```

Main memory hash-join
(M-1)ps ⋈ pr

Block Nested Loop Join



Nested Loop Joins

Cost of block-based nested loop join

- Read S once: B(S)
- Outer loop runs B(S)/(M-2) times, each iteration reads the entire R: B(S)B(R)/(M-2)
- Total cost: B(S) + B(S)B(R)/(M-2)

Nested Loop Joins

Cost of block-based nested loop join

- Read S once: B(S)
- Outer loop runs B(S)/(M-2) times, each iteration reads the entire R: B(S)B(R)/(M-2)
- Total cost: B(S) + B(S)B(R)/(M-2)

Iterate over the smaller relation first!

Index Nested Loop Join

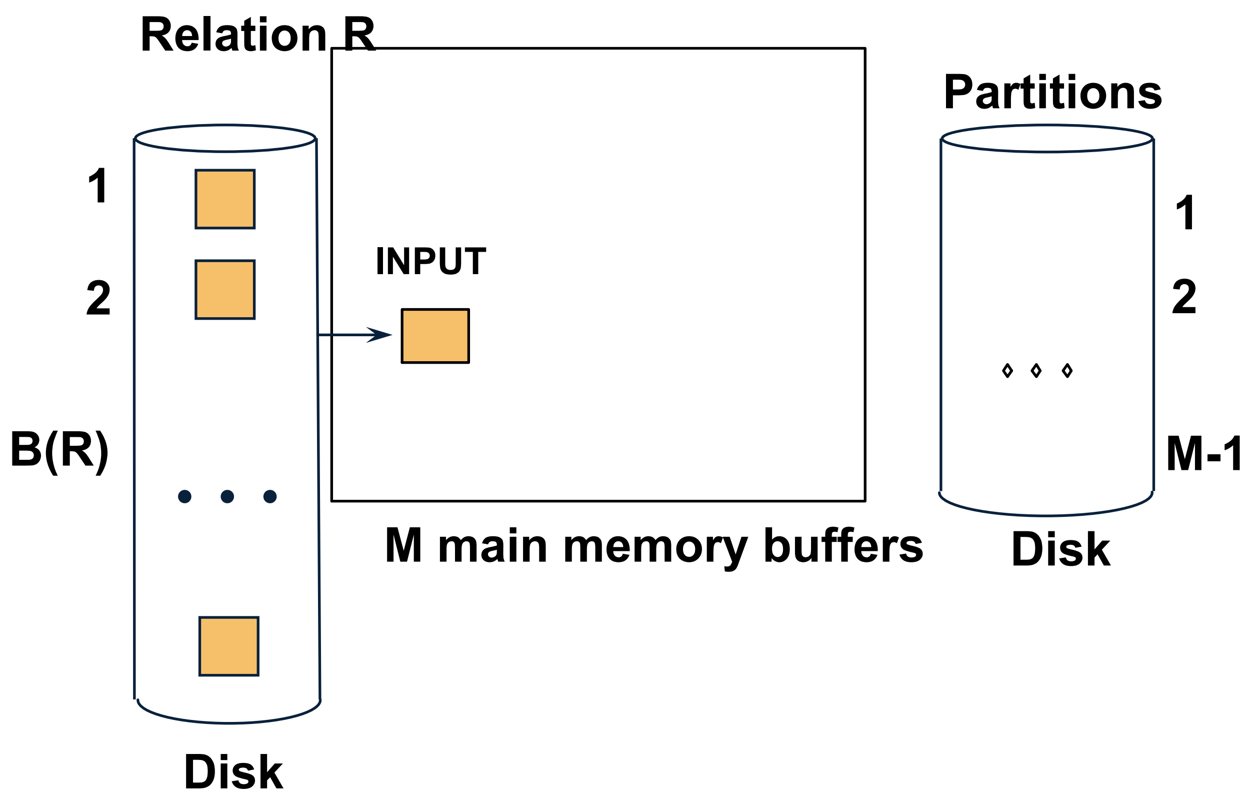
$R \bowtie S$

- 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: $B(R) + T(R)B(S) / V(S,a)$
  - If index on S is unclustered: $B(R) + T(R)T(S) / V(S,a)$

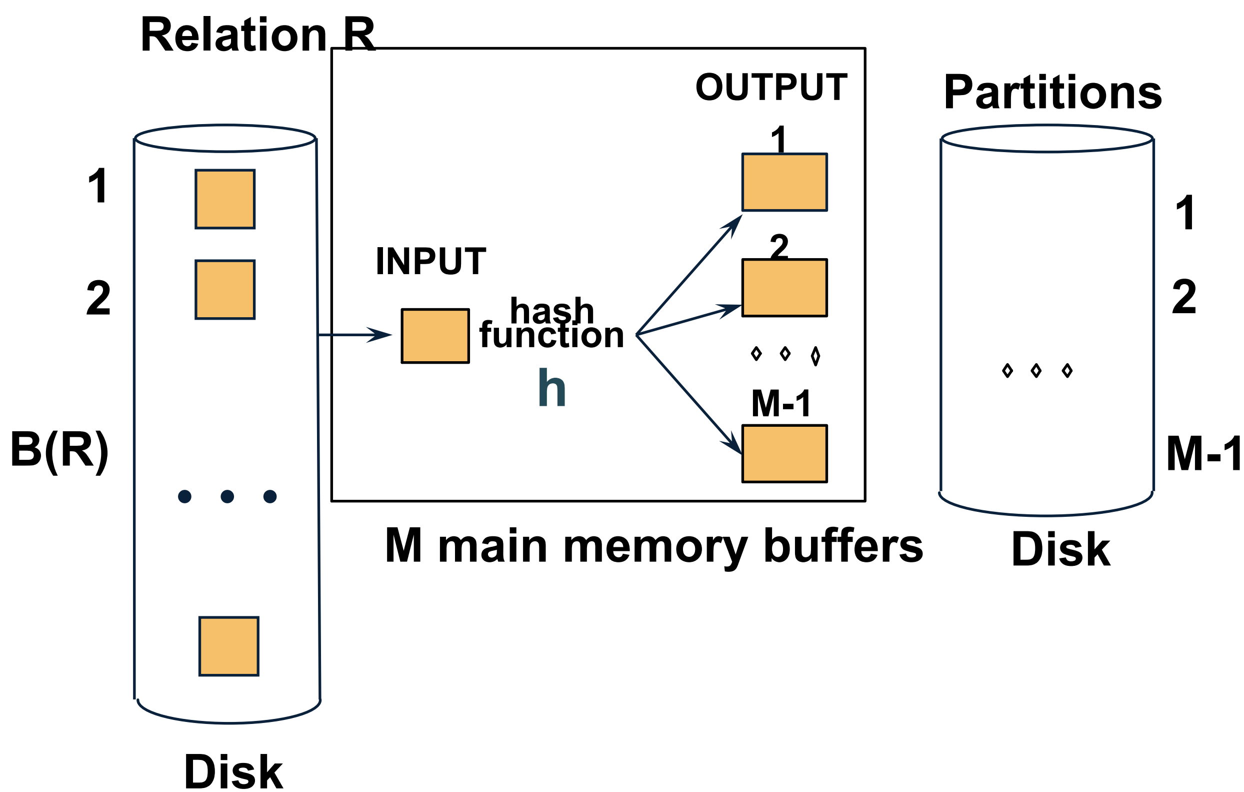
Two Pass Algorithms

- Idea: partition a relation R into buckets, on disk



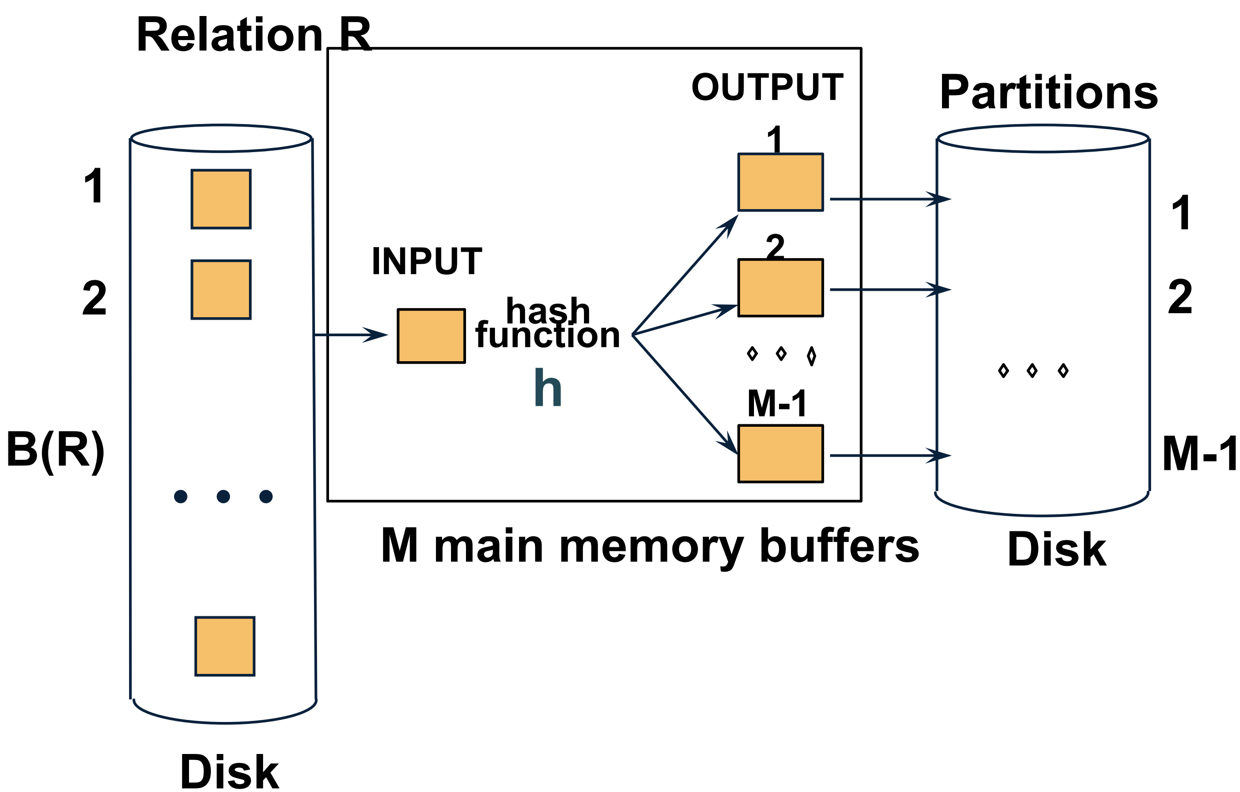
Two Pass Algorithms

- Idea: partition a relation R into buckets, on disk



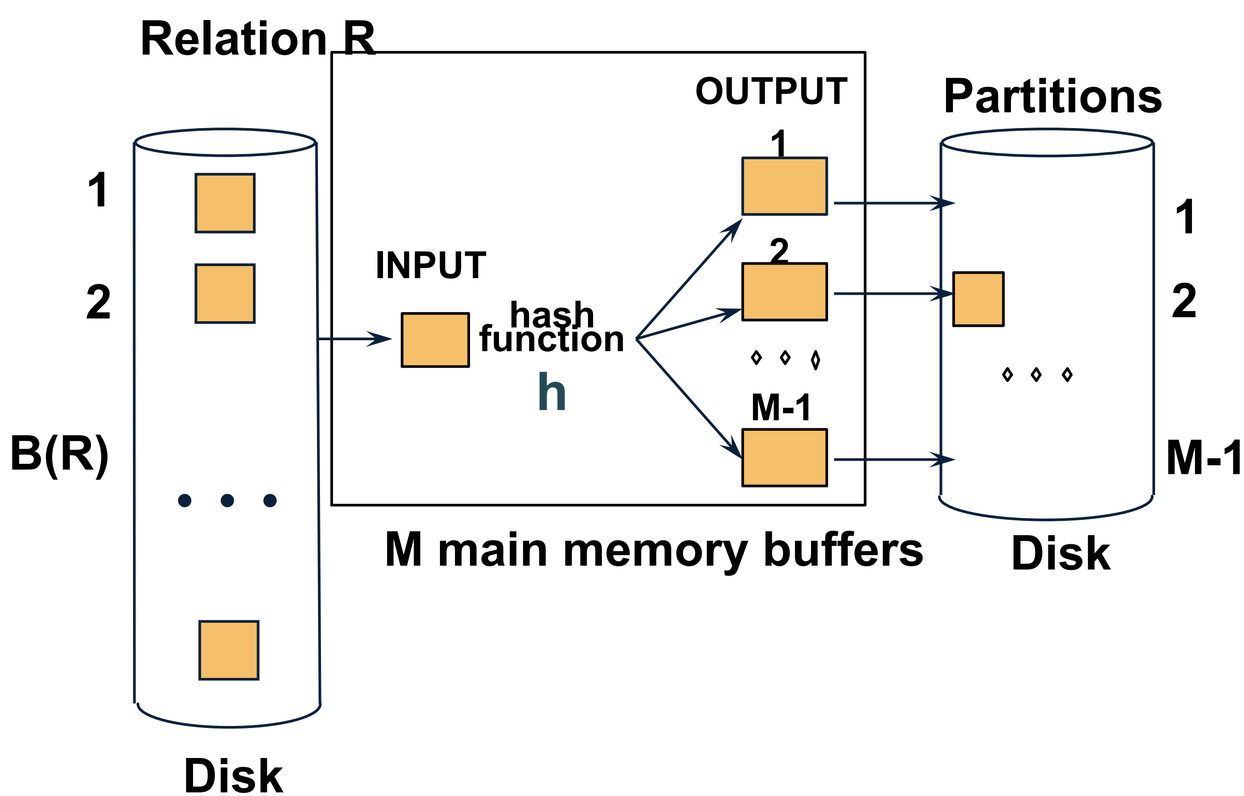
Two Pass Algorithms

- Idea: partition a relation R into buckets, on disk



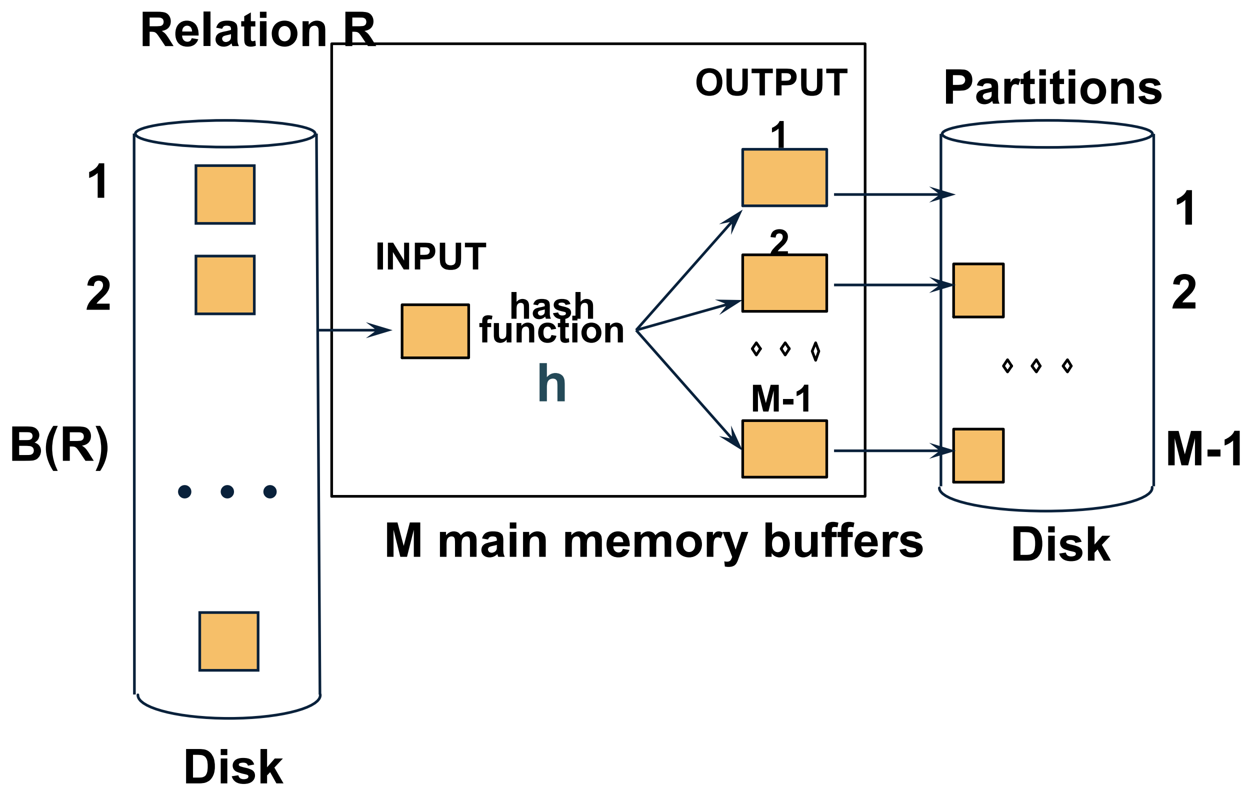
Two Pass Algorithms

- Idea: partition a relation R into buckets, on disk



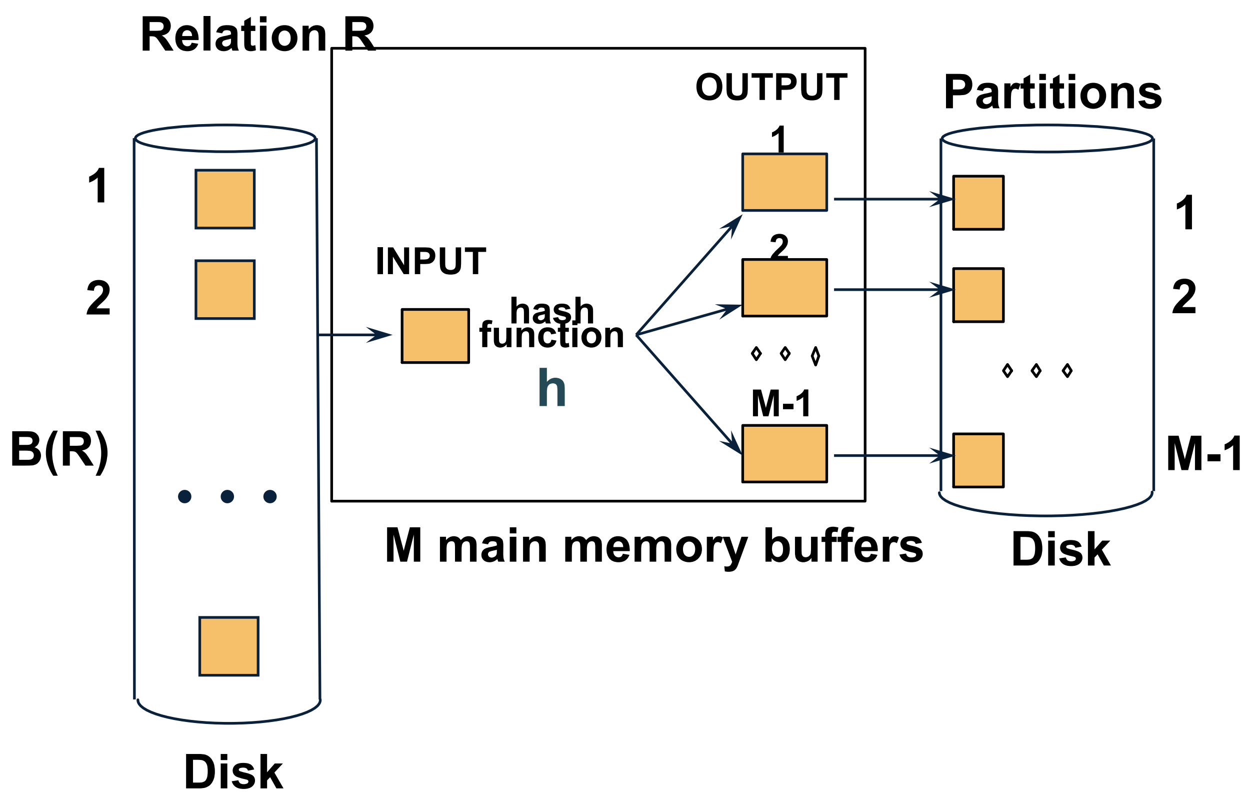
Two Pass Algorithms

- Idea: partition a relation R into buckets, on disk



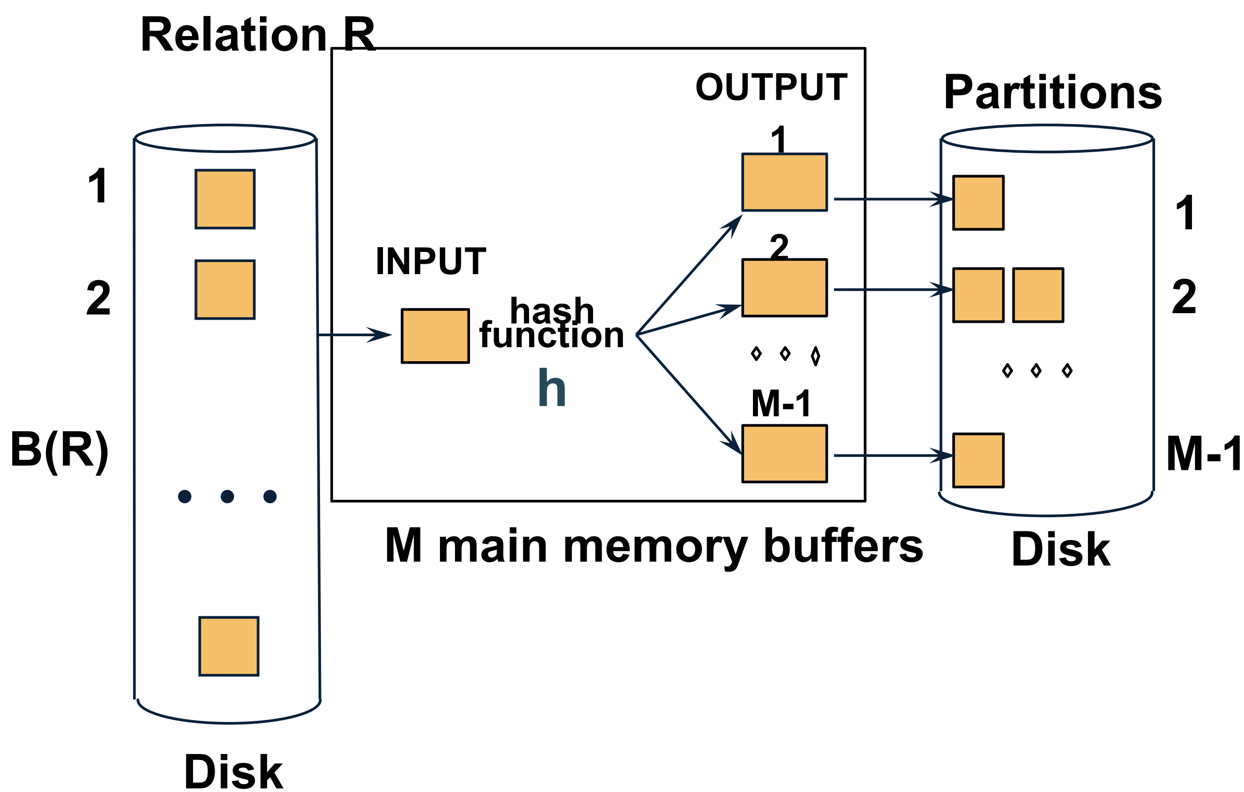
Two Pass Algorithms

- Idea: partition a relation R into buckets, on disk



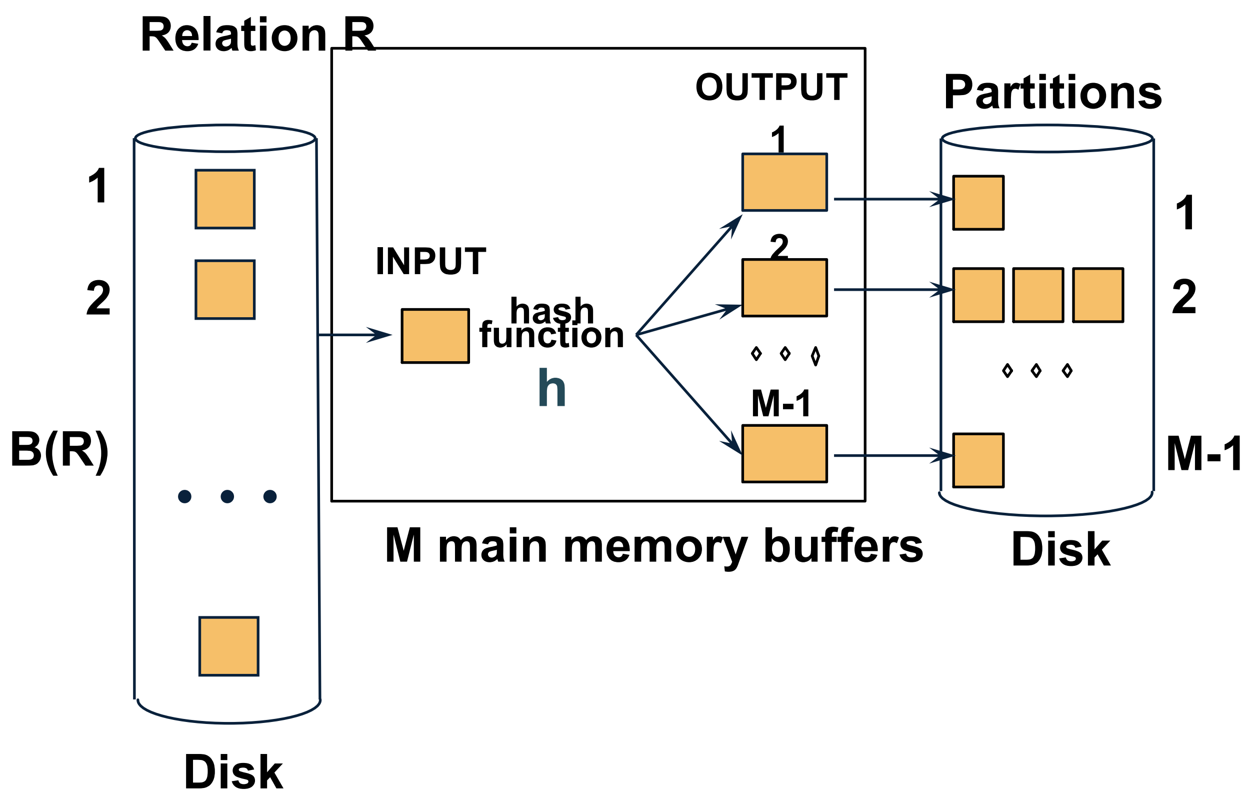
Two Pass Algorithms

- Idea: partition a relation R into buckets, on disk



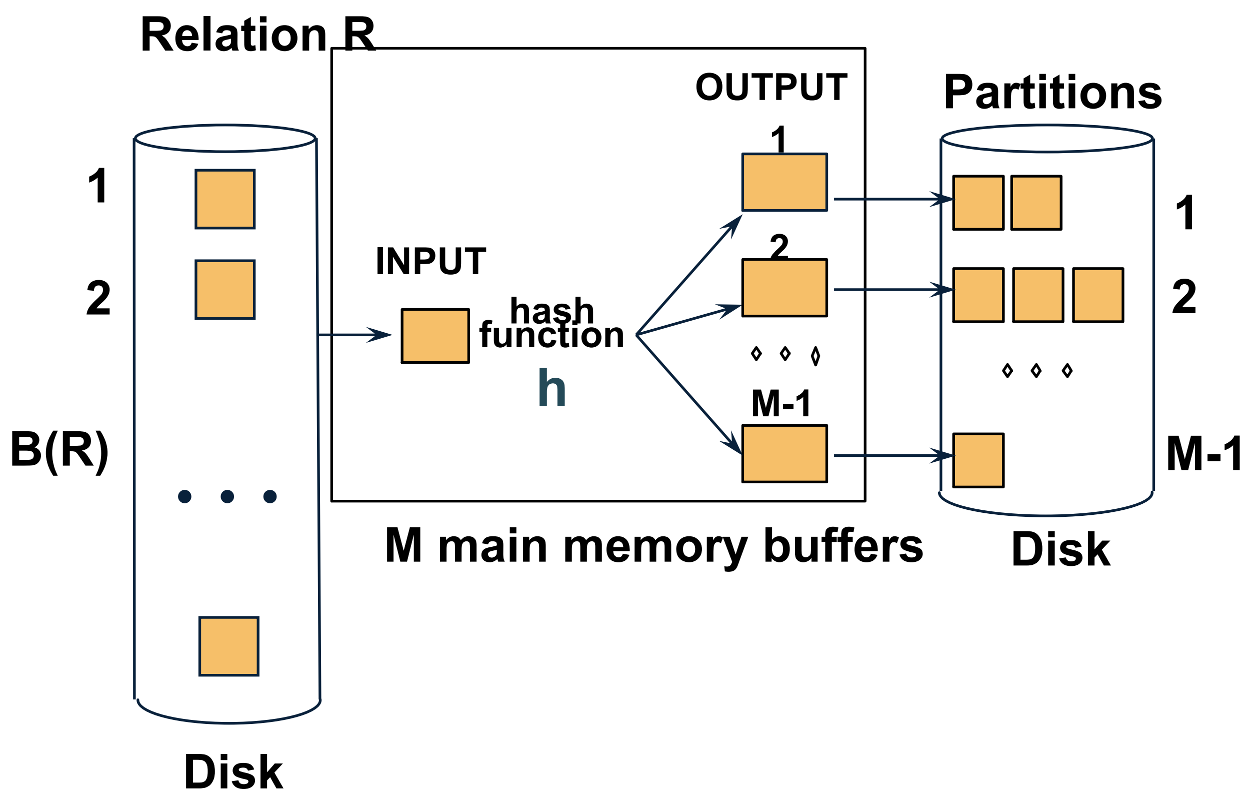
Two Pass Algorithms

- Idea: partition a relation R into buckets, on disk



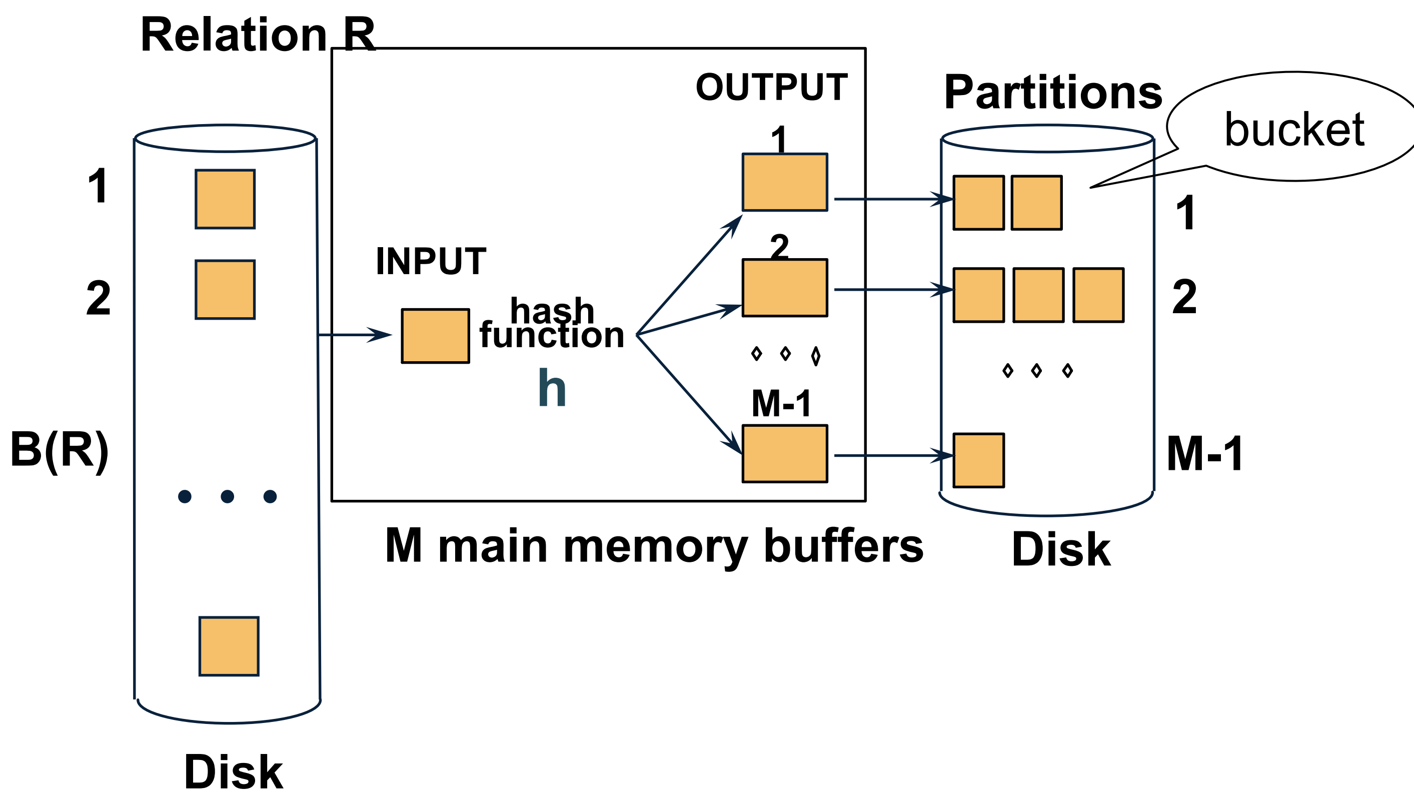
Two Pass Algorithms

- Idea: partition a relation R into buckets, on disk



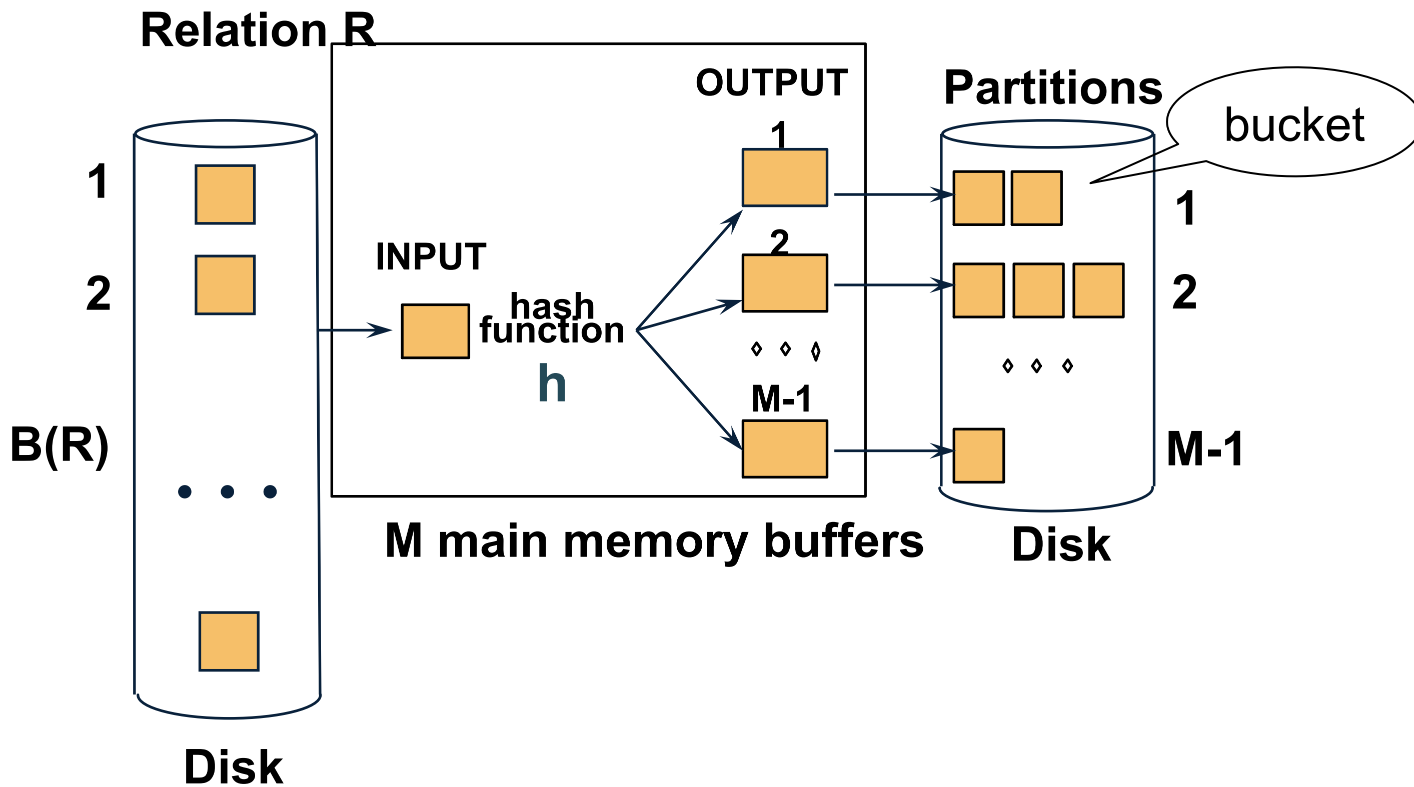
Two Pass Algorithms

- Idea: partition a relation R into buckets, on disk



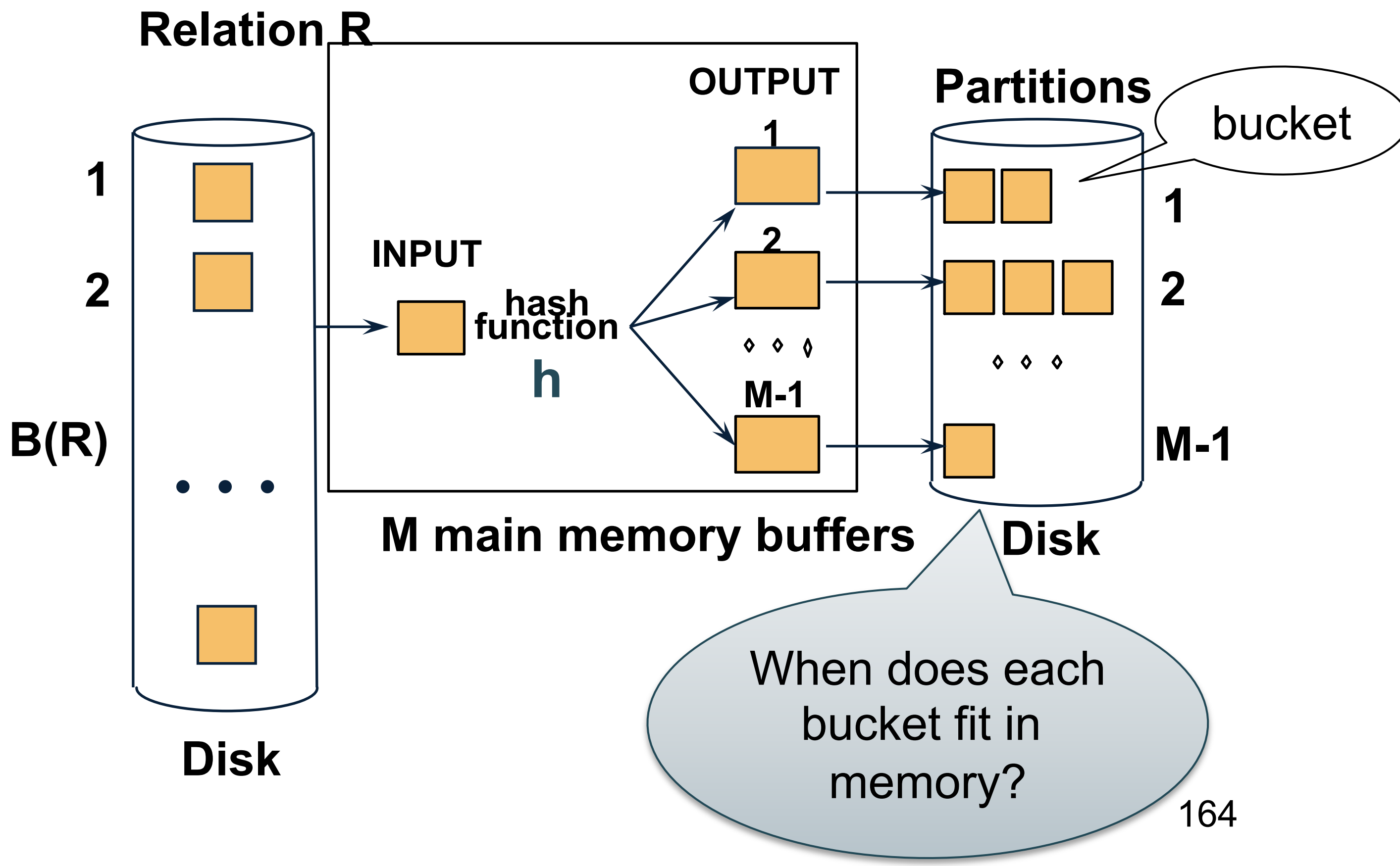
Two Pass Algorithms

- Idea: partition a relation R into buckets, on disk
- Each bucket has size approx. B(R)/M



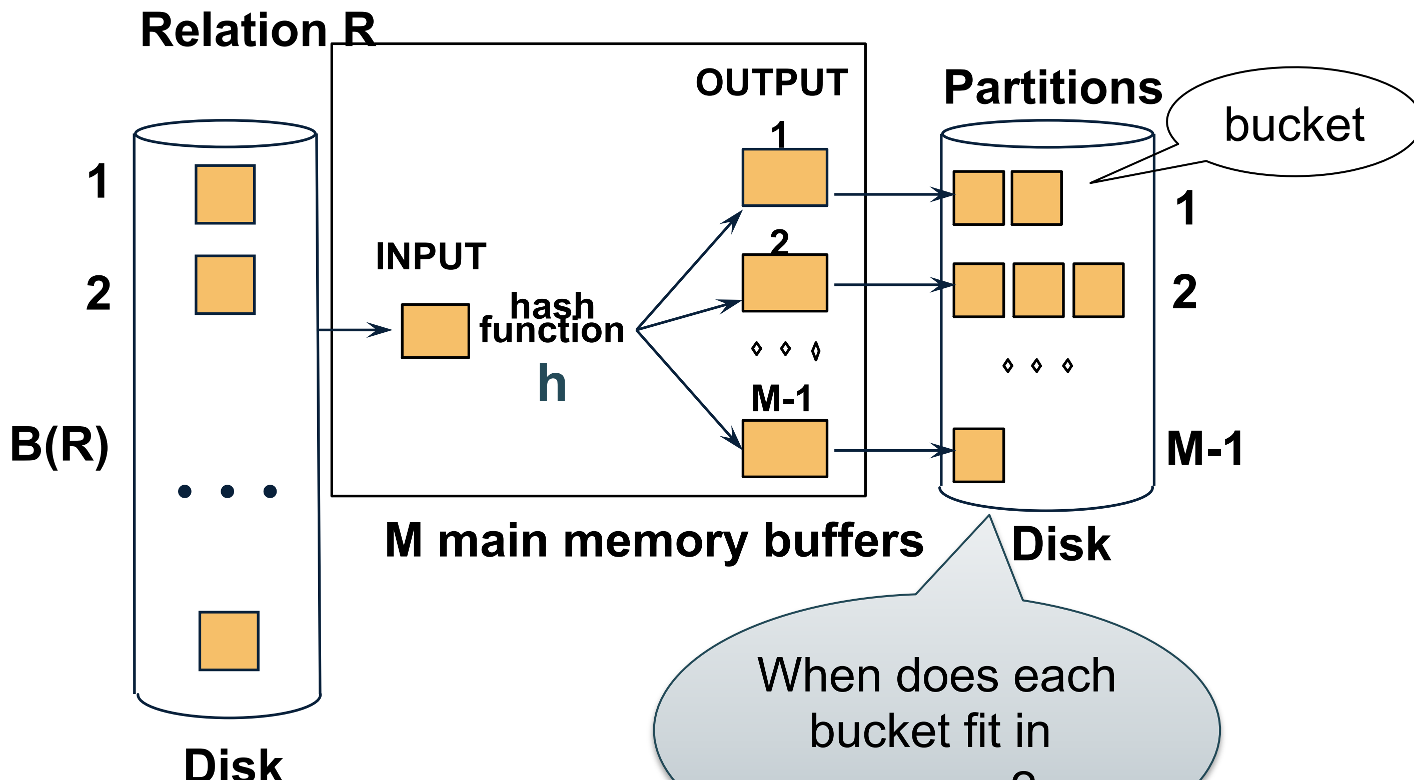
Two Pass Algorithms

- Idea: partition a relation R into buckets, on disk
- Each bucket has size approx. B(R)/M



Two Pass Algorithms

- Idea: partition a relation R into buckets, on disk
- Each bucket has size approx. B(R)/M



When does each bucket fit in memory?

When $B(R)/M \leq M$, i.e. $B(R) \leq M^2$

Partitioned (Grace) Hash Join

$R \bowtie S$

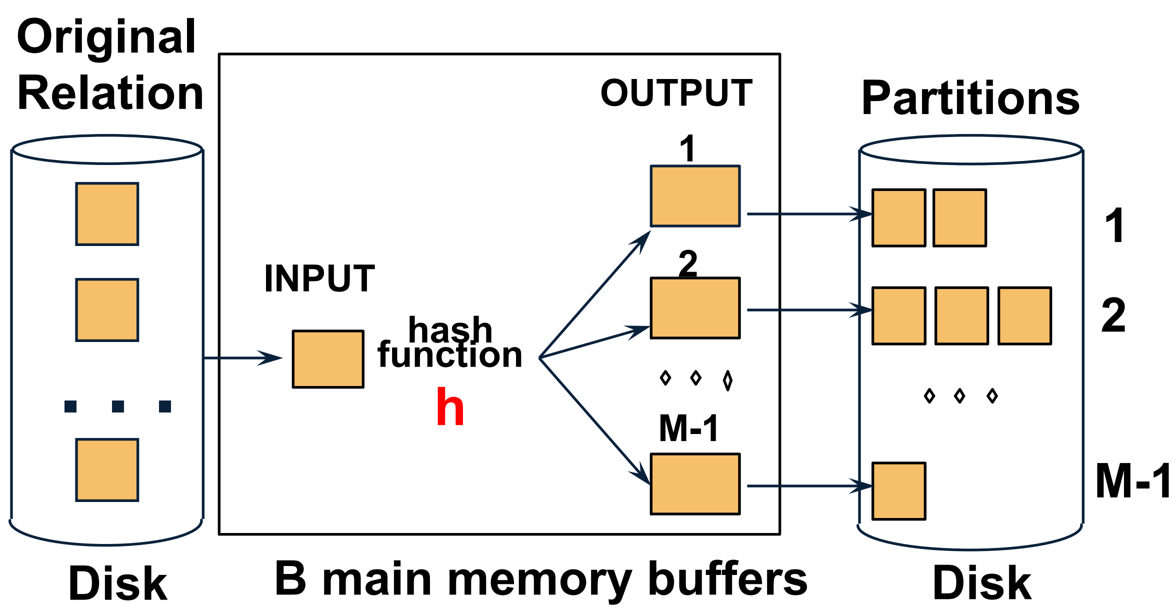
- Step 1:
  – Hash S into M-1 buckets
  – Send all buckets to disk

- Step 2
  – Hash R into M-1 buckets
  – Send all buckets to disk

- Step 3
  – Join every pair of buckets

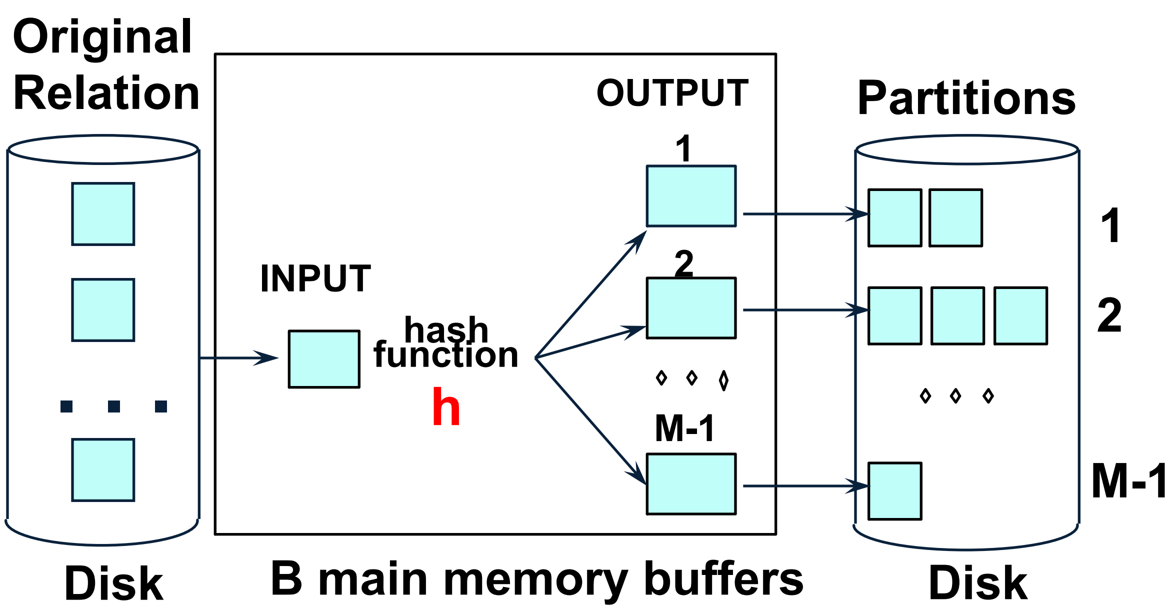
Partition R using hash fn h

R ⋈ S



Partition S using hash fn h

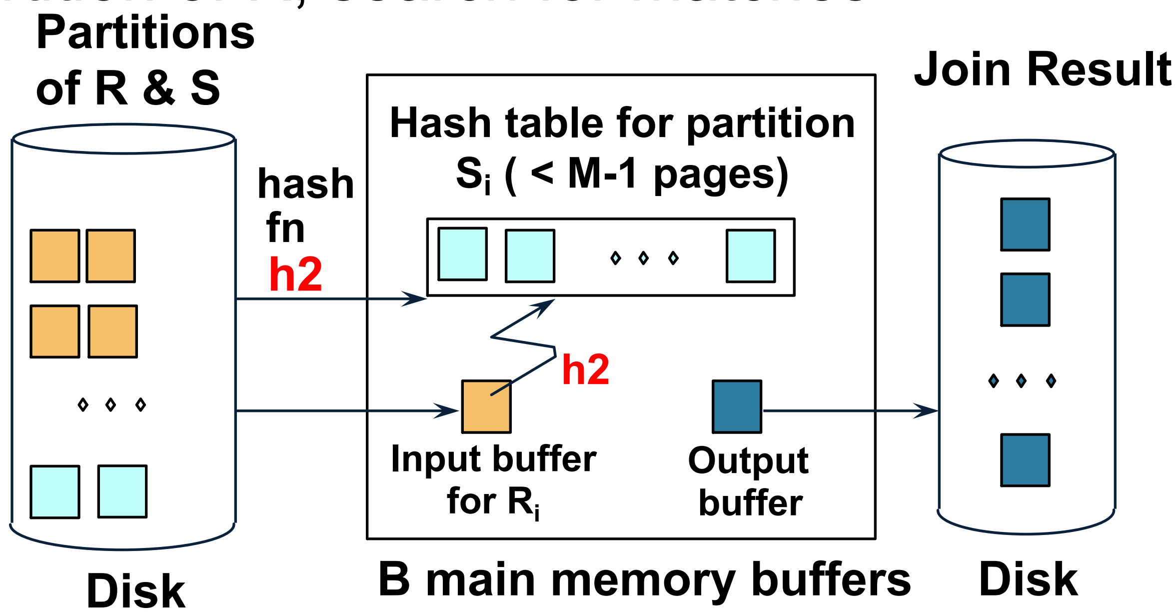
R ⋈ S



Partitioned Hash Join

$R \bowtie S$

- Read in partition of S, hash it using h2 ($\neq$ h)
- Scan same partition of R, search for matches



Partitioned Hash Join

- Cost: 3B(R) + 3B(S)
- Assumption: min(B(R), B(S)) ≤ M^2

Hybrid Hash Join Algorithm

- Assume we have extra memory available

- Partition S into k buckets
  t buckets S_1 , …, S_t stay in memory
  k-t buckets S_{t+1}, …, S_k to disk

- Partition R into k buckets
  - First t buckets join immediately with S
  - Rest k-t buckets go to disk

- Finally, join k-t pairs of buckets:
  (R_{t+1},S_{t+1}), (R_{t+2},S_{t+2}), …, (R_k,S_k)

Hybrid Hash Join Algorithm

How to choose k and t ?

- The first t buckets must fin in M: $t/k * B(S) \leq M$

Hybrid Hash Join Algorithm

How to choose k and t ?

- The first t buckets must fin in M: $t/k * B(S) \leq M$
- Need room for k-t additional pages: $k-t \leq M$

Hybrid Hash Join Algorithm

How to choose k and t ?

- The first t buckets must fin in M: $t/k * B(S) \leq M$
- Need room for k-t additional pages: $k-t \leq M$
- Thus: $t/k * B(S) + k-t \leq M$

Hybrid Hash Join Algorithm

How to choose k and t ?

- The first t buckets must fin in M: $t/k * B(S) \leq M$
- Need room for k-t additional pages: $k-t \leq M$
- Thus: $t/k * B(S) + k-t \leq M$

Assuming $t/k * B(S) \gg k-t$: $t/k = M/B(S)$

Hybrid Hash Join Algorithm

- How many I/Os ?

- Cost of partitioned hash join: 3B(R) + 3B(S)

- Hybrid join saves 2 I/Os for a t/k fraction of buckets
- Hybrid join saves 2t/k(B(R) + B(S)) I/Os

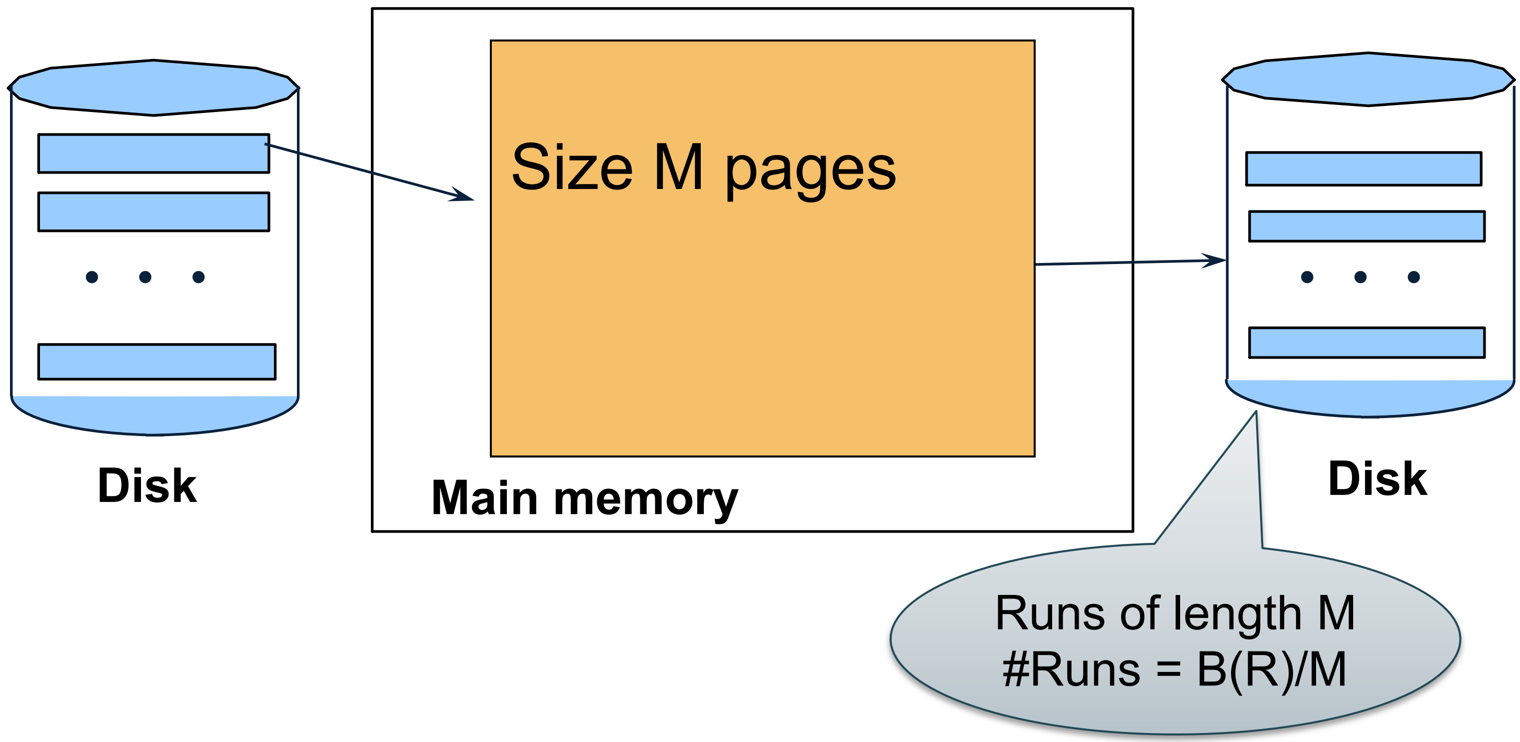
Cost: (3-2t/k)(B(R) + B(S)) = (3-2M/B(S))(B(R) + B(S))

External Sorting

- Problem: Sort a file of size B with memory M
- Where we need this:
  - ORDER BY in SQL queries
  - Several physical operators
  - Bulk loading of B+-tree indexes.
- Will discuss only 2-pass sorting, for when $B \leq M^2$

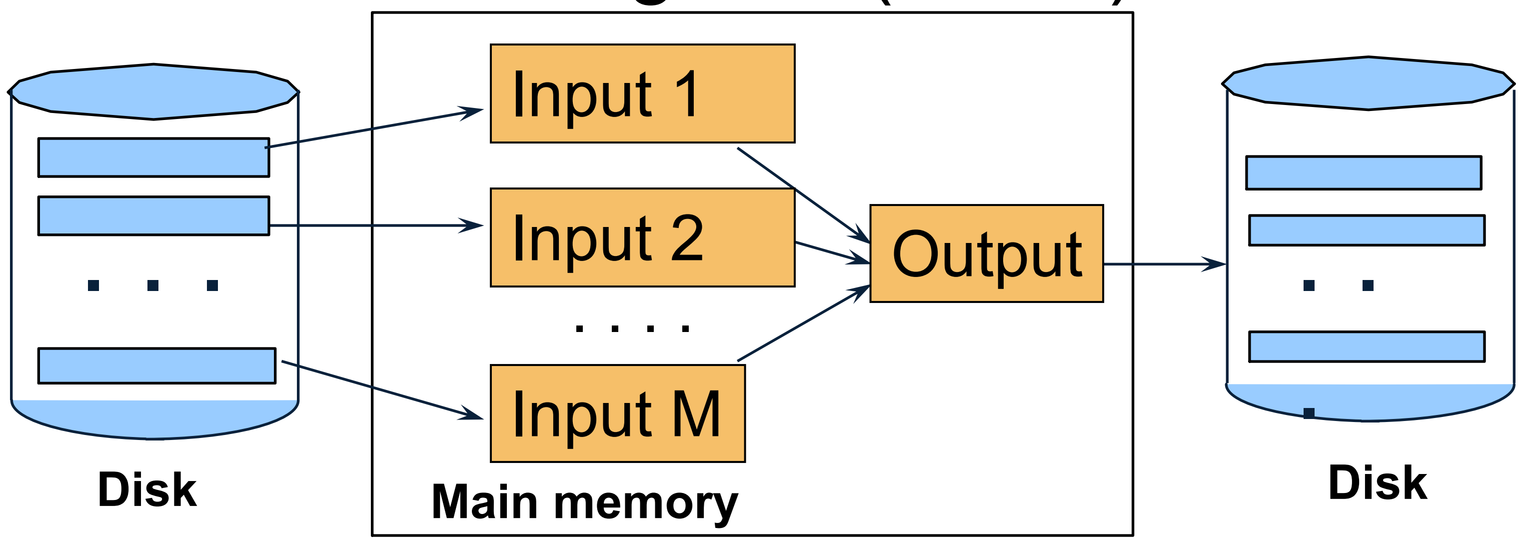
External Merge-Sort: Step 1

- Phase one: load M pages in memory, sort



External Merge-Sort: Step 2

- Merge M – 1 runs into a new run
- Result: runs of length M (M – 1) $\approx M^2$



Assuming $B \leq M^2$, we are done

External Merge-Sort

- Cost:
  – Read+write+read = 3B(R)
  – Assumption: B(R) <= M^2

- Other considerations
  – In general, a lot of optimizations are possible

Two-Pass Algorithms Based on Sorting

Grouping: $\gamma_{a,\ sum(b)}$ (R)

Sort, then compute the sum(b) for each group of a's

- Step 1: sort chunks of size M, write
  - cost 2B(R)
- Step 2: merge M-1 runs, combining groups by addition
  - cost B(R)
- Total cost: 3B(R), Assumption: $B(R) \leq M^2$

Two-Pass Algorithms Based on Sorting

Join R ⋈ S

- Start by creating initial runs of length M, for R and S:
  - Cost: 2B(R)+2B(S)
- Merge (and join) M_1 runs from R, M_2 runs from S:
  - Cost: B(R)+B(S)
- Total cost: 3B(R)+3B(S)
- Assumption:
  - R has M_1=B(R)/M runs, S has M_2=B(S)/M runs
  - $M_1 + M_2 \leq M$
  - Hence: $B(R)+B(S) \leq M^2$

Summary of External Join Algorithms

- Block Nested Loop Join: B(R) + B(R)*B(S)/M
- Index Nested Loop Join: B(R) + T(R)B(S)/V(S,a)
- Hash Join: 3(B(R) + B(S))
  Hybrid Hash Join: (3-2M/B(S))(B(R) + B(S))
- Sort-Merge Join: 3B(R)+3B(S)