# CSE544
# Data Management

Lectures 6-8

Query Execution

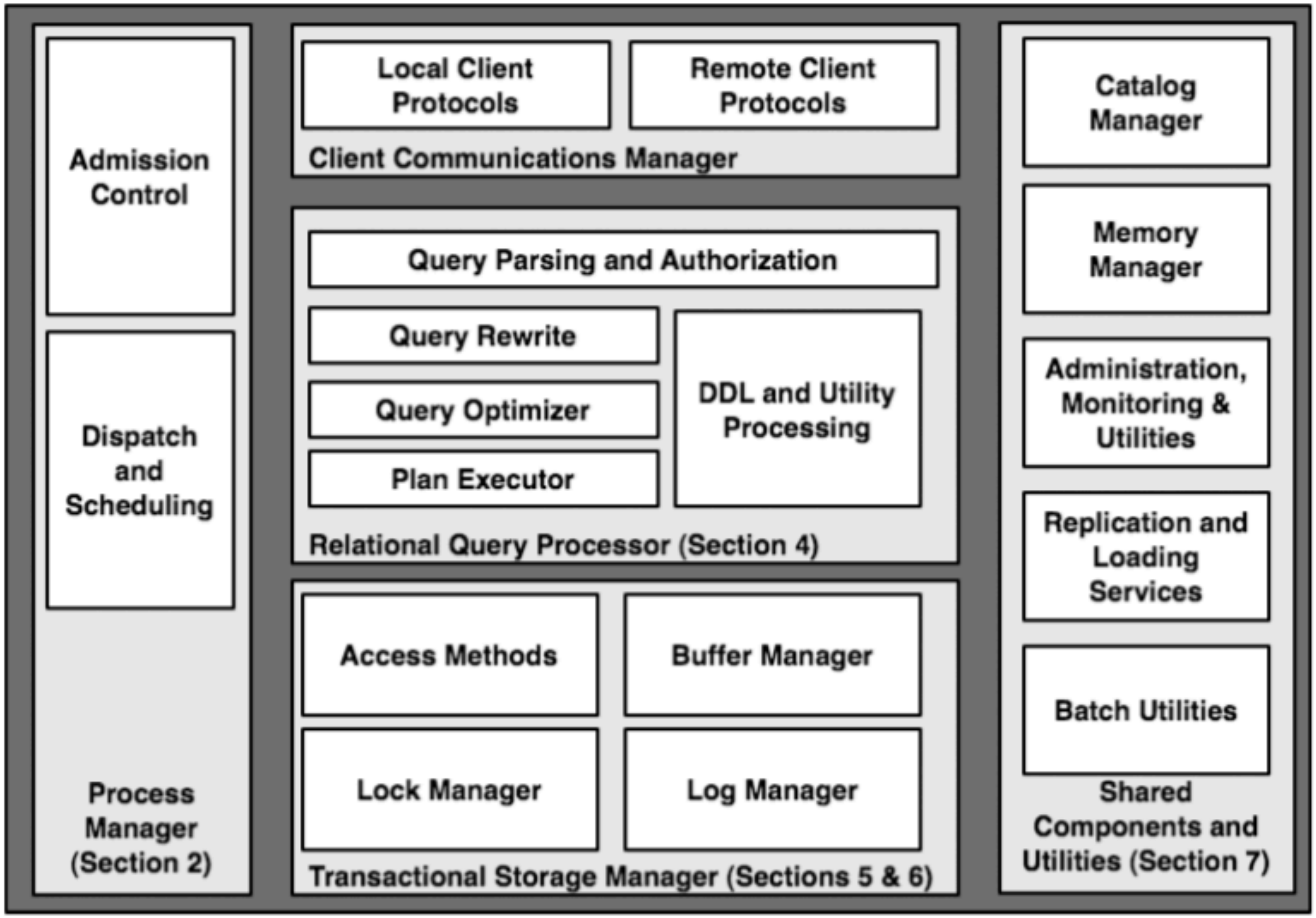
# Announcements

- Review 2 due on Wednesday (Ch. 1&2 only)
- Friday: both HW2 *and* project proposals due
- Next Friday: will meet with each team to discuss the project proposals

# Outline

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

# Architecture of DBMS

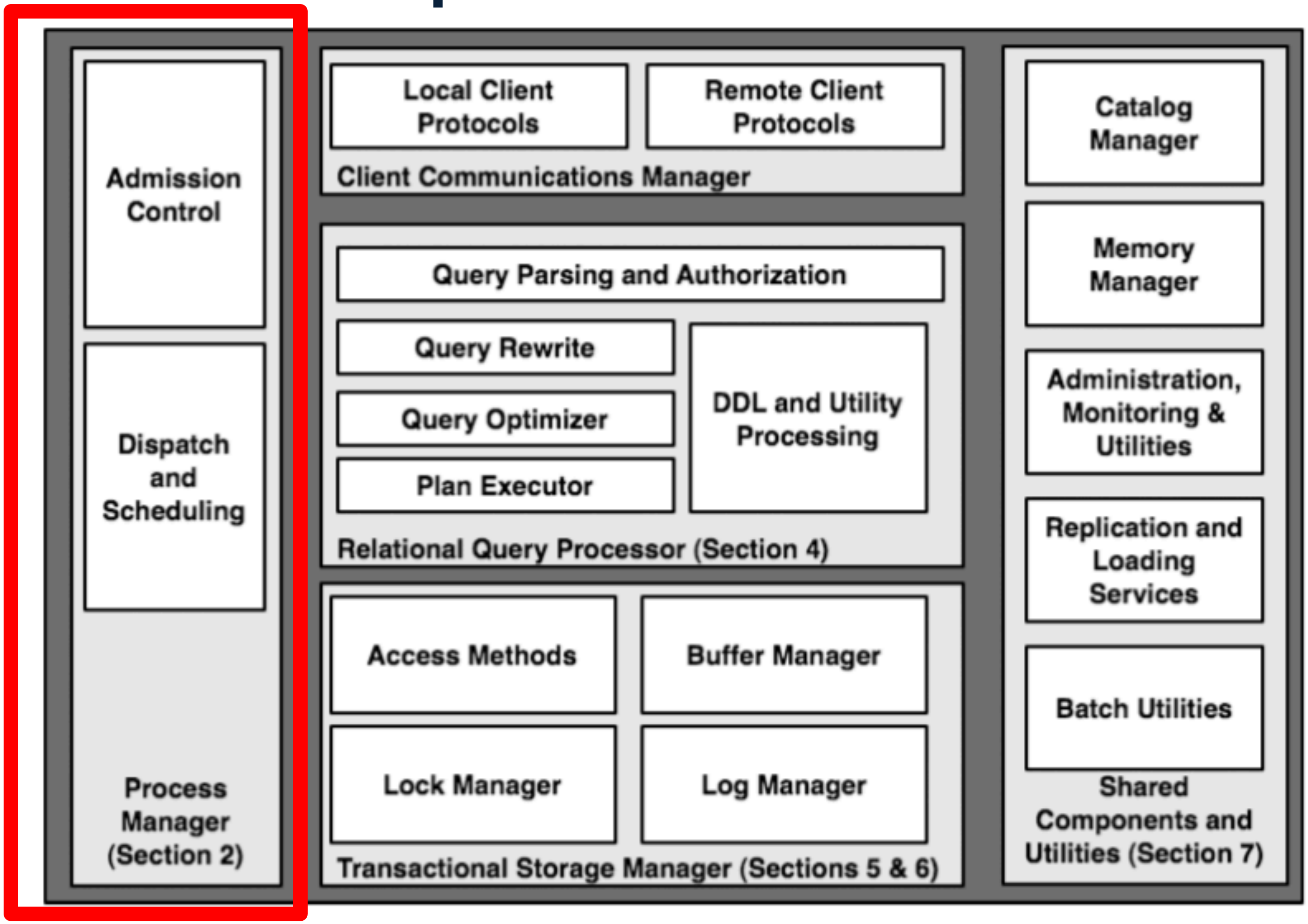


# Warning: it will be confusing…

DBMS are monoliths: all components must work together and cannot be isolated

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

# Multiple Processes



# Why Multiple Processes

- DBMS listens to requests from clients
- Each request = one SQL command
- Need to handle multiple requests concurrently,
  hence, multiple processes

# Process Models

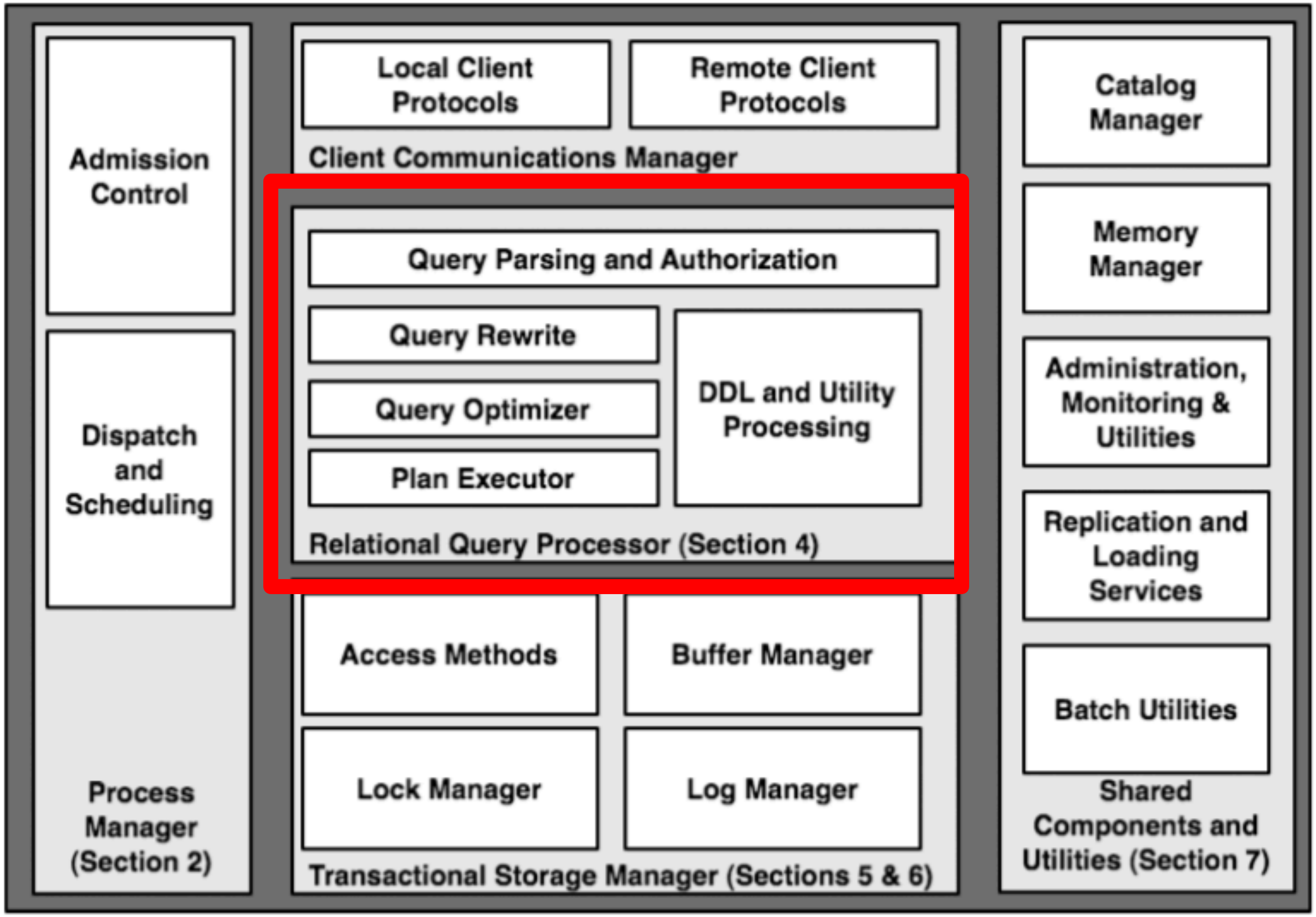
- Process per DBMS worker
- Thread per DBMS worker
- Process pool

Discuss pro/cons for each model

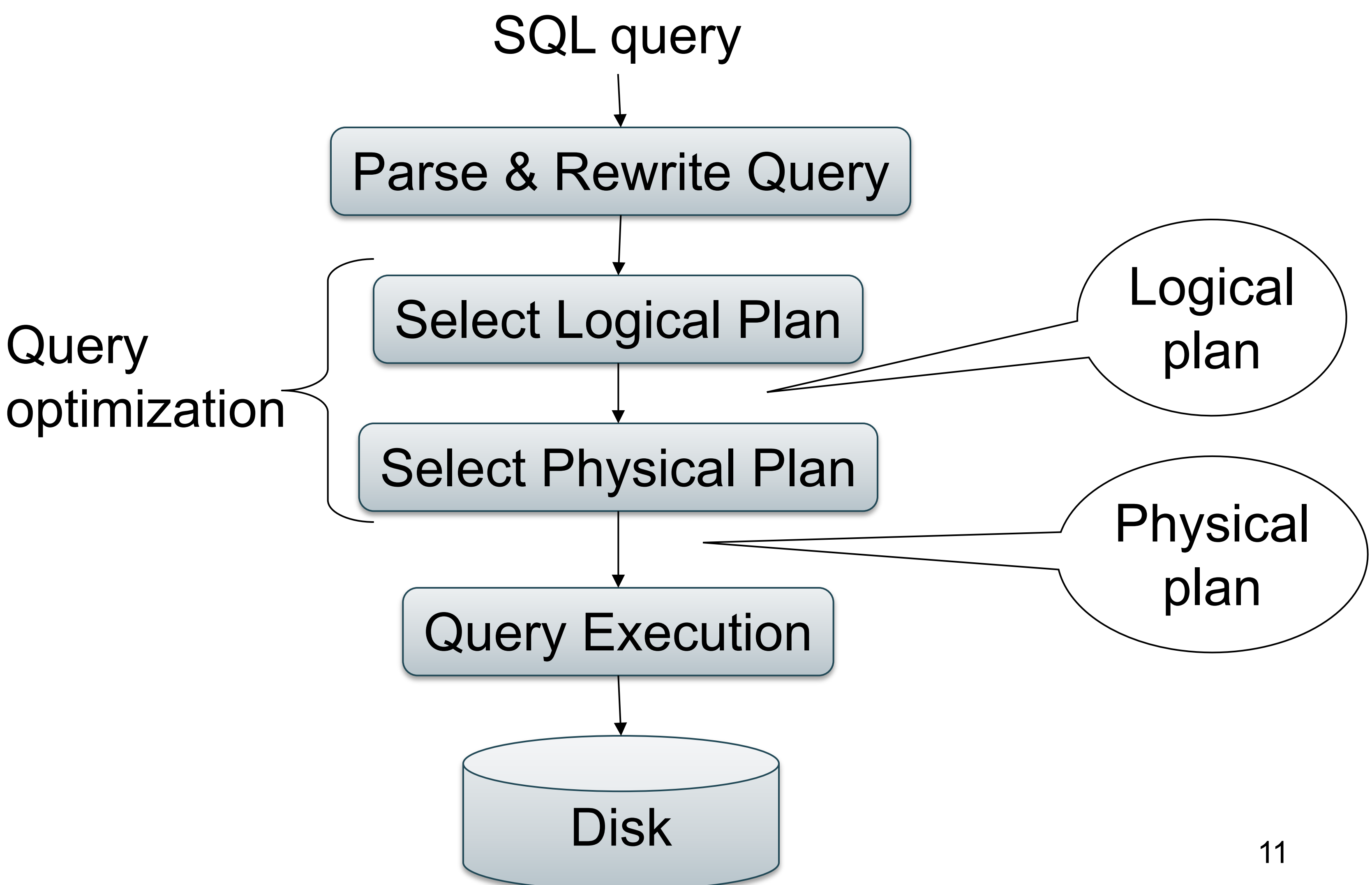
# Outline

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

# Query Optimization



# Lifecycle of a Query



# Example Database Schema

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

View: Suppliers in Seattle

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

# Example Query

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

```
SELECT sname FROM NearbySupp
WHERE sno IN ( SELECT sno
               FROM Supplies
               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.

# View Rewriting, Flattening

Original query:

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

View rewriting
 = view inlining
 = view expansion
Flattening
 = unnesting

# View Rewriting, Flattening

## Original query:

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

View rewriting
 = view inlining
 = view expansion
Flattening
 = unnesting

## Rewritten query:

```
SELECT S.sname
FROM Supplier S, Supplies 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

```
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)
```

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)
```

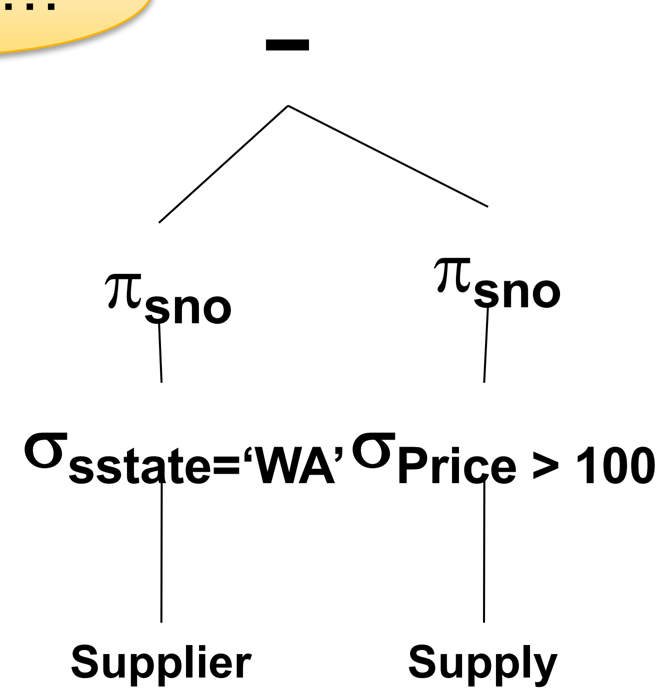
EXCEPT = set difference

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 executing the query
  - We will spend two lectures on this topic

- A **query plan** is
  - **Logical query plan**: an extended relational algebra tree
  - **Physical query plan**: with additional annotations at each node

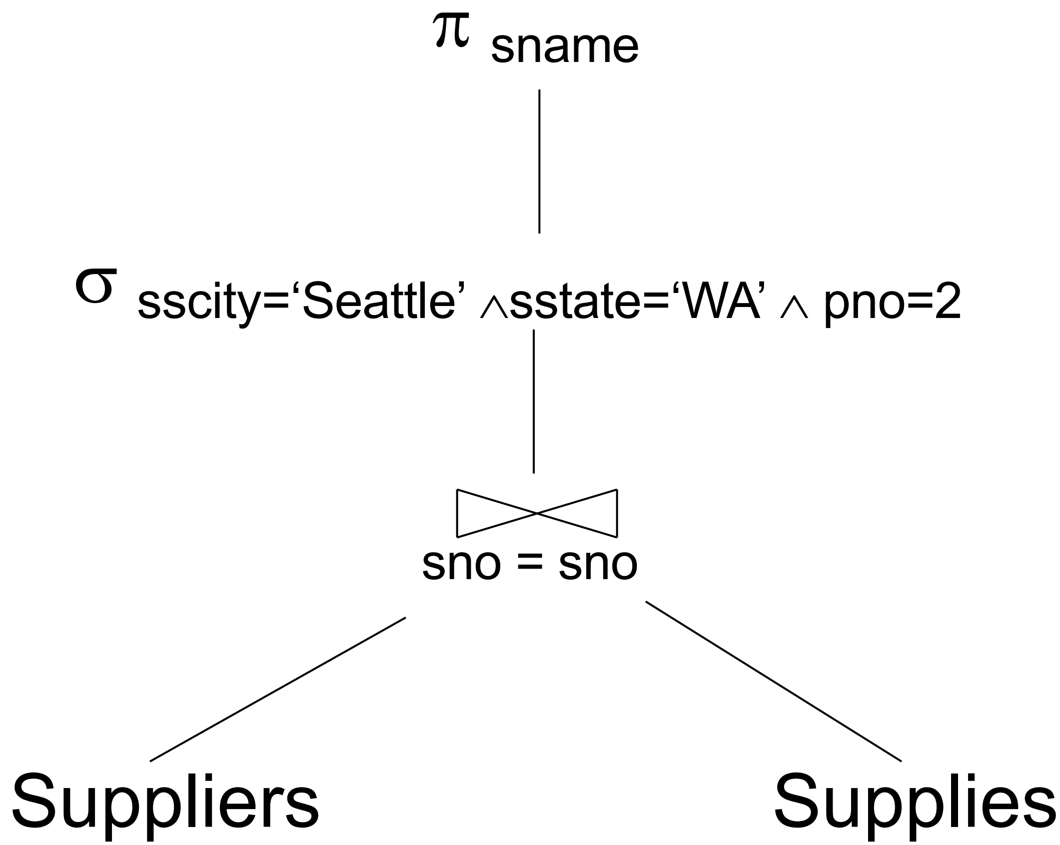
# Relational Algebra Operators

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

RA

Extended RA

# Logical Query Plan



# Query Block

- Most optimizers operate on individual query blocks

- A query block is an SQL query with **no nesting**
  - **Exactly one**
    - SELECT clause
    - FROM clause
  - **At most one**
    - WHERE clause
    - GROUP BY clause
    - HAVING clause

# Query Plan For A Block

$\sigma_{\text{having-condition}}$

$\gamma$ fields, sum/count/min/max(fields)

$\pi$ fields

$\sigma_{\text{where-condition}}$

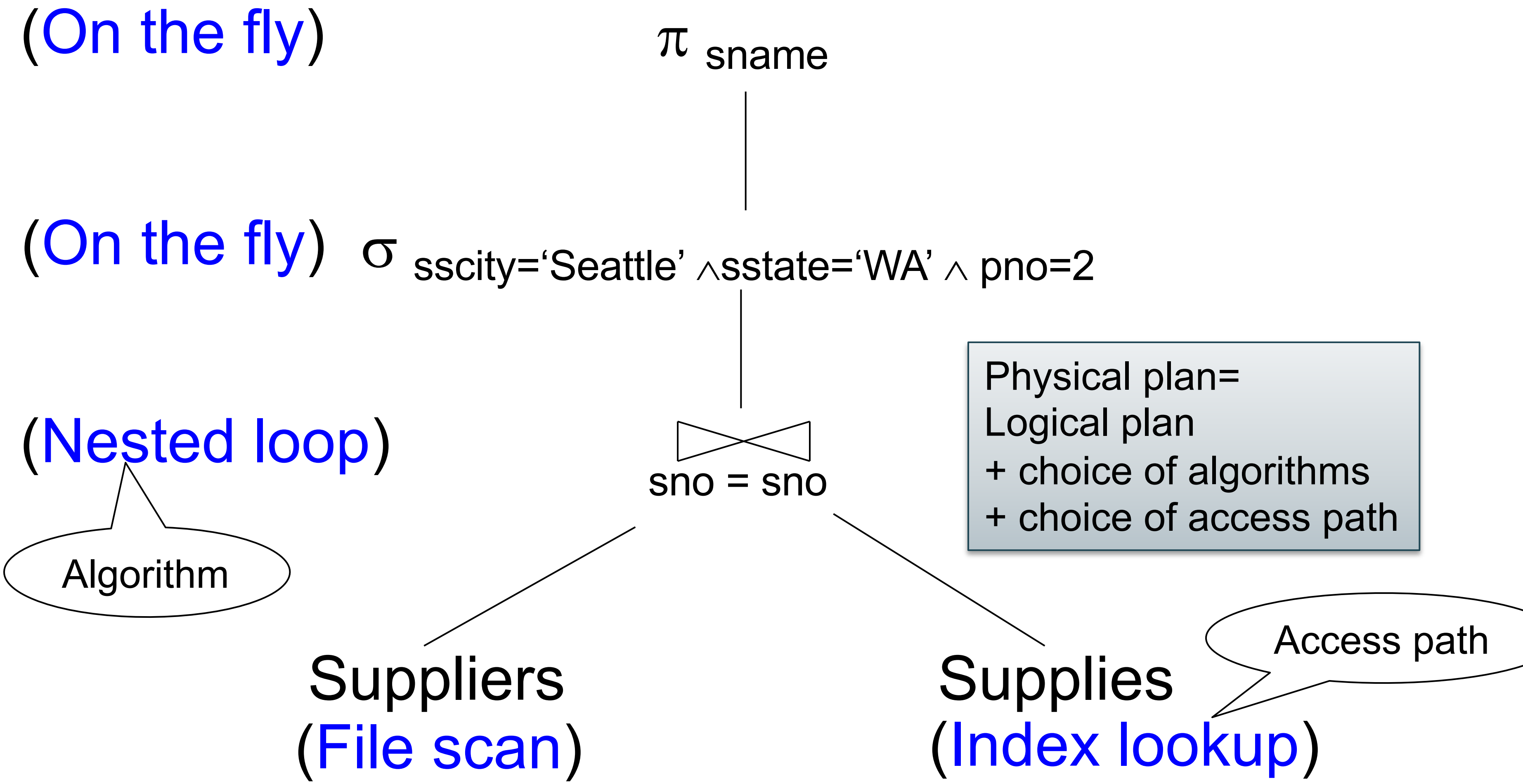
$\bowtie$

join condition

… …

SELECT-PROJECT-JOIN Query

# Physical Query Plan



# Final Step in Query Processing

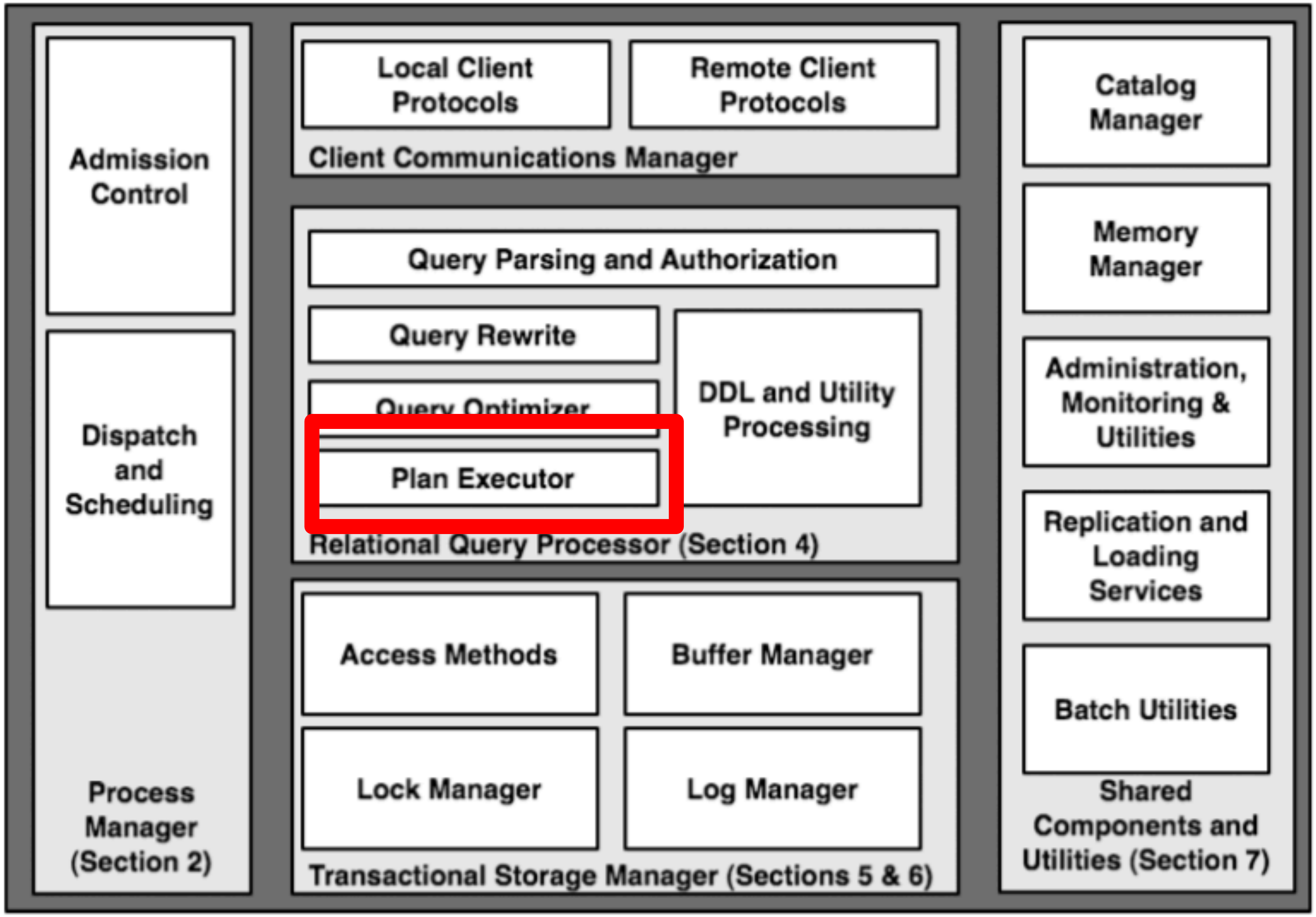
- **Step 4: Query execution**
  - How to synchronize operators
  - How to pass data between operators

- Standard approach:
  - Iterator interface and
  - Pipelined execution or
  - Intermediate result materialization

# 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
- Examples:
  - Main memory hash join
  - External memory merge join
  - External memory partitioned hash join
  - Sort-based group by
  - Hash-based group by

$\bowtie$

$\gamma$

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

# 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:

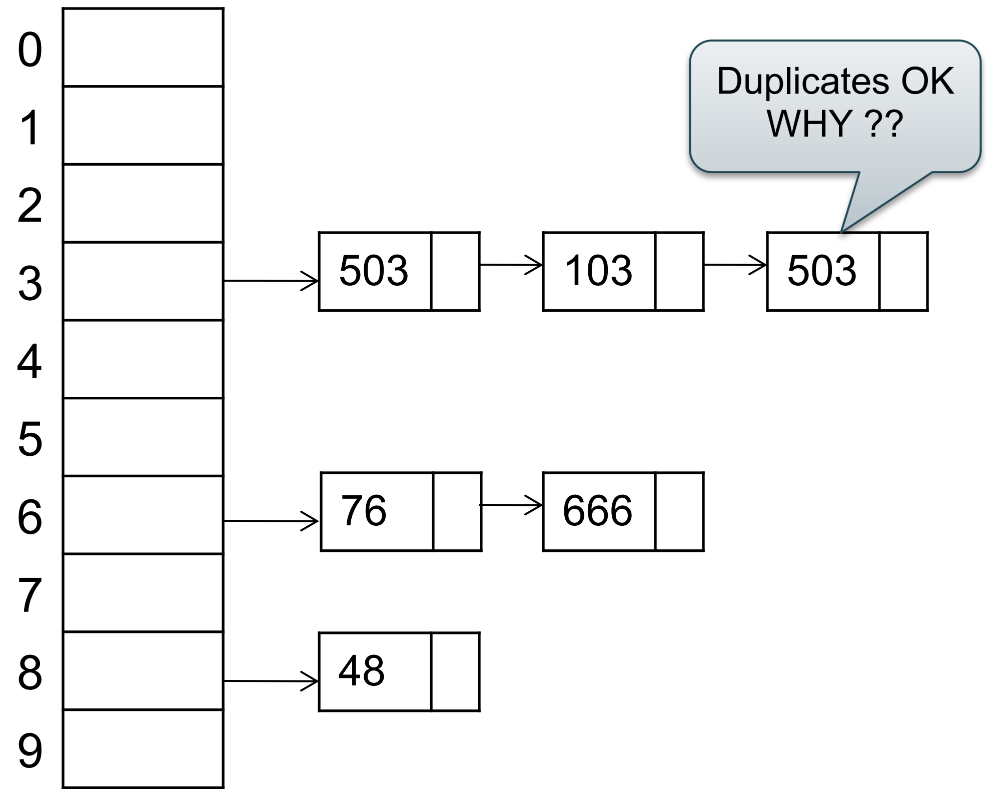
$$h(x) = x \bmod 10$$

Operations:

find(103) = ??

insert(488) = ??

Separate chaining:



# BRIEF Review of Hash Tables

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

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

# 2. Hash Join

Logical operator:

Supplier $\bowtie_{sid=sid}$ Supply

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

for y in Supply do
      x = find(y.sid);
      output(x,y);
```

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

# 2. Hash Join

Logical operator:

Supplier $\bowtie_{sid=sid}$ Supply

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

for y in Supply do
    x = find(y.sid);
    output(x,y);
```

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

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

# 2. Hash Join

Logical operator:

Supplier $\bowtie_{sid=sid}$ Supply

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

for y in Supply do
    x = find(y.sid);
    output(x,y);
```

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

O(n)

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

# 2. Hash Join

Logical operator:

Supplier $\bowtie_{sid=sid}$ Supply

Change join order

```
for y in Supply do
    insert(y.sid, y)

for x in Supplier do
    ????
```

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

# 2. Hash Join

Logical operator:

Supplier $\bowtie_{sid=sid}$ Supply

Change join order

```
for y in Supply do
     insert(y.sid, y)

for x in Supplier do
    for y in find(x.sid) do
        output(x,y);
```

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

# 2. Hash Join

Logical operator:

Supplier $\bowtie_{sid=sid}$ Supply

Change join order

```
for y in Supply do
      insert(y.sid, y)

for x in Supplier do
    for y in find(x.sid) do
        output(x,y);
```

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

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

# 2. Hash Join

Logical operator:

Supplier $\bowtie_{sid=sid}$ Supply

Change join order

```
for y in Supply do
    insert(y.sid, y)

for x in Supplier do
    for y in find(x.sid) do
        output(x,y);
```

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

O(n)
But can be O($n^2$) why?

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

# 2. Hash Join

Why would we change the order?

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?

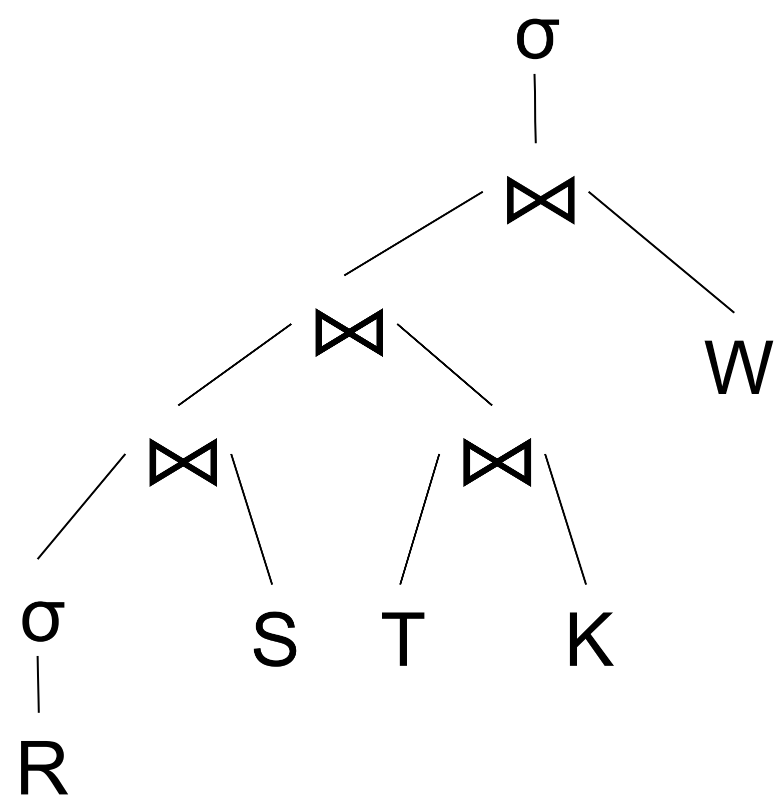
O(n log(n))

# Main Memory Algorithms

- Join $\bowtie$:
  - Nested loop join
  - Hash join
  - Merge join
- Selection $\sigma$
  - "on-the-fly"
  - Index-based selection (next lecture)
- Group by $\gamma$
  - Hash–based
  - Merge-based

Discuss in class

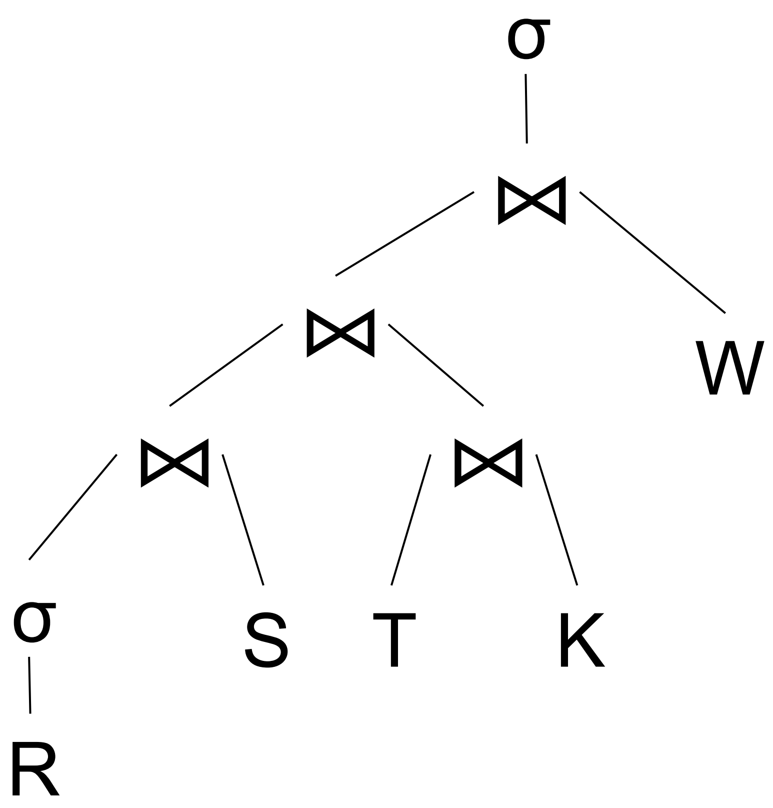
# How Do We Combine Them?



# 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

```
interface Operator {

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


  // calls next() on its inputs
  // processes an input tuple
  // produces output tuple(s)
  // returns null when done
  Tuple next ();


  // cleans up (if any)
  void close ();
}
```

Example "on the fly" selection operator

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









}
```

# Implementing Query Operators with the Iterator Interface

```
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

```
interface Operator {

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


  // calls next() on its inputs
  // processes an input tuple
  // produces output tuple(s)
  // returns null when done
  Tuple next ();


  // cleans up (if any)
  void close ();
}
```

Example “on the fly” selection operator

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

}
```

# Implementing Query Operators with the Iterator Interface

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

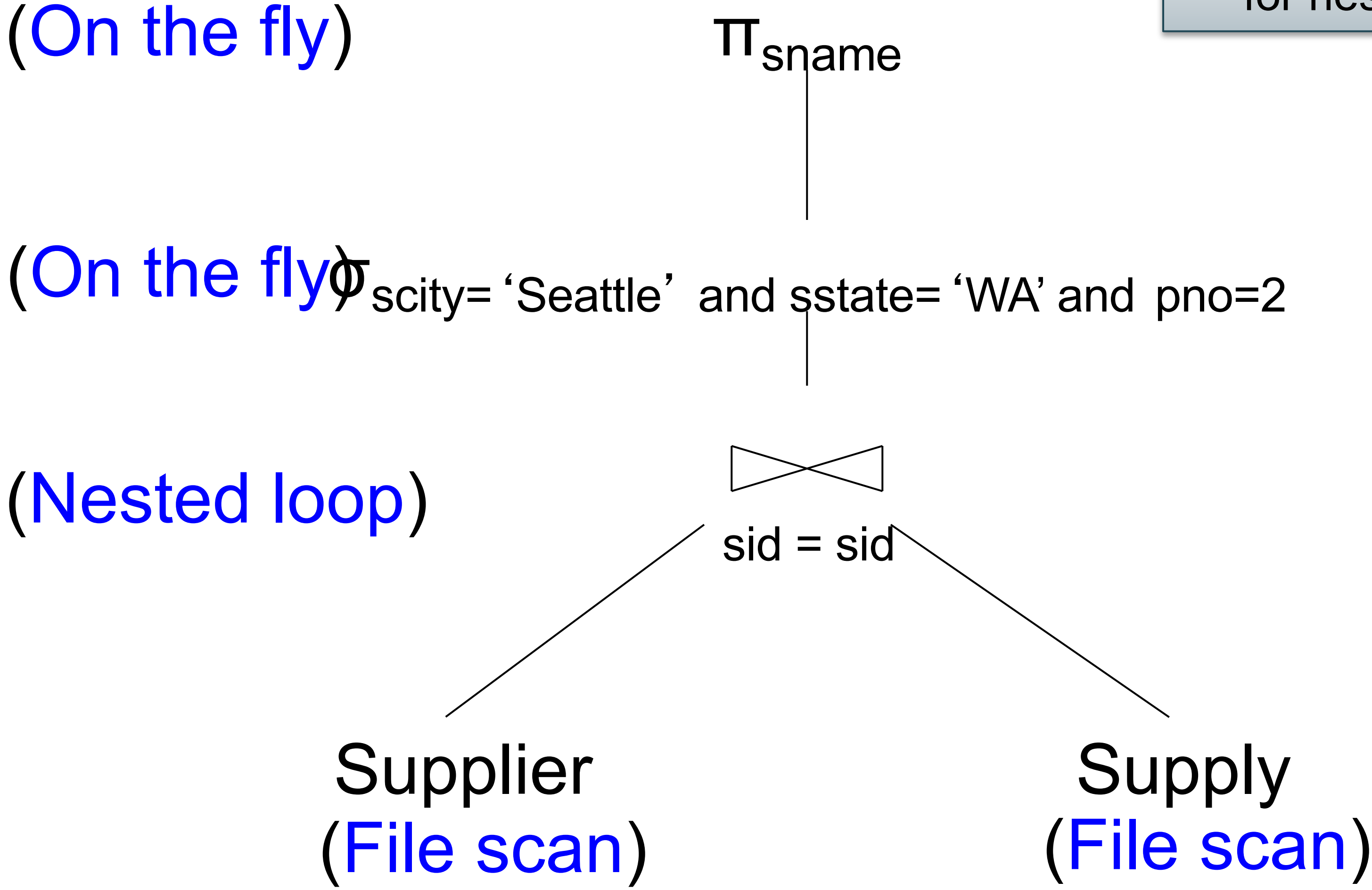
```
Operator q = parse("SELECT ...");
q = optimize(q);

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

Supplier(<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

open()

(On the fly) $\pi_{sname}$

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

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

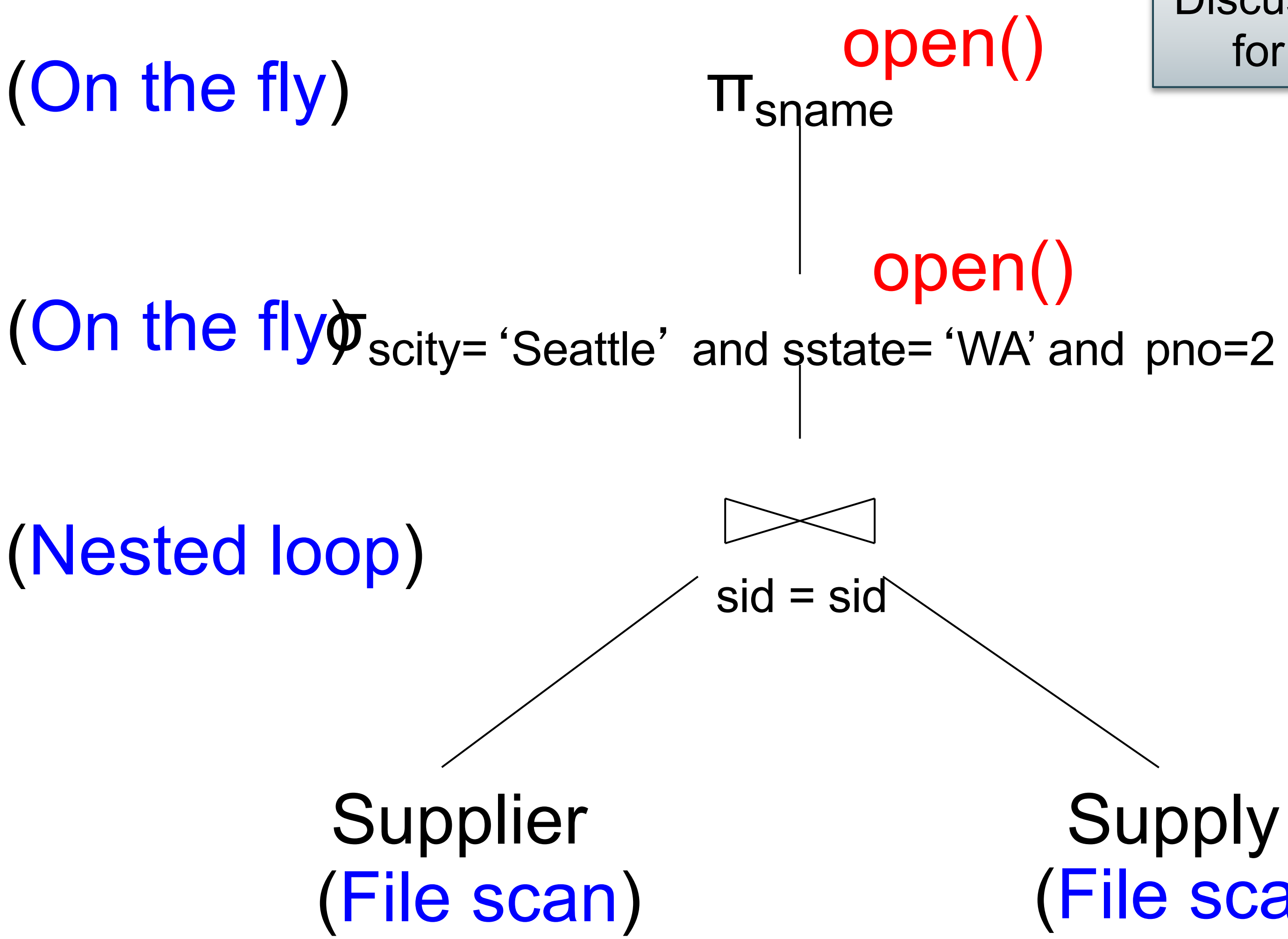
Supplier (File scan)

Supply (File scan)

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

# Pipelining

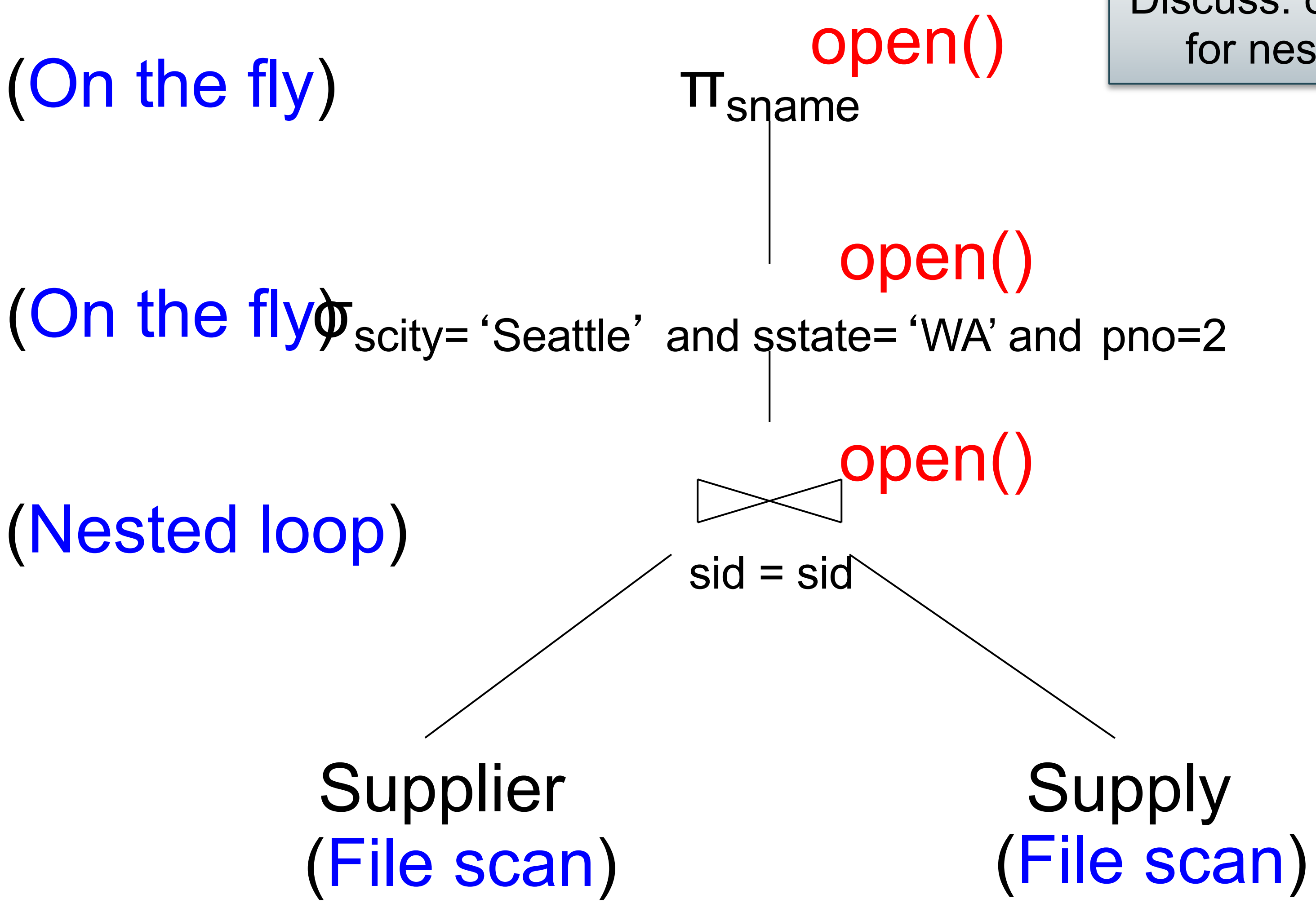
Discuss: open/next/close for nested loop join



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

# Pipelining

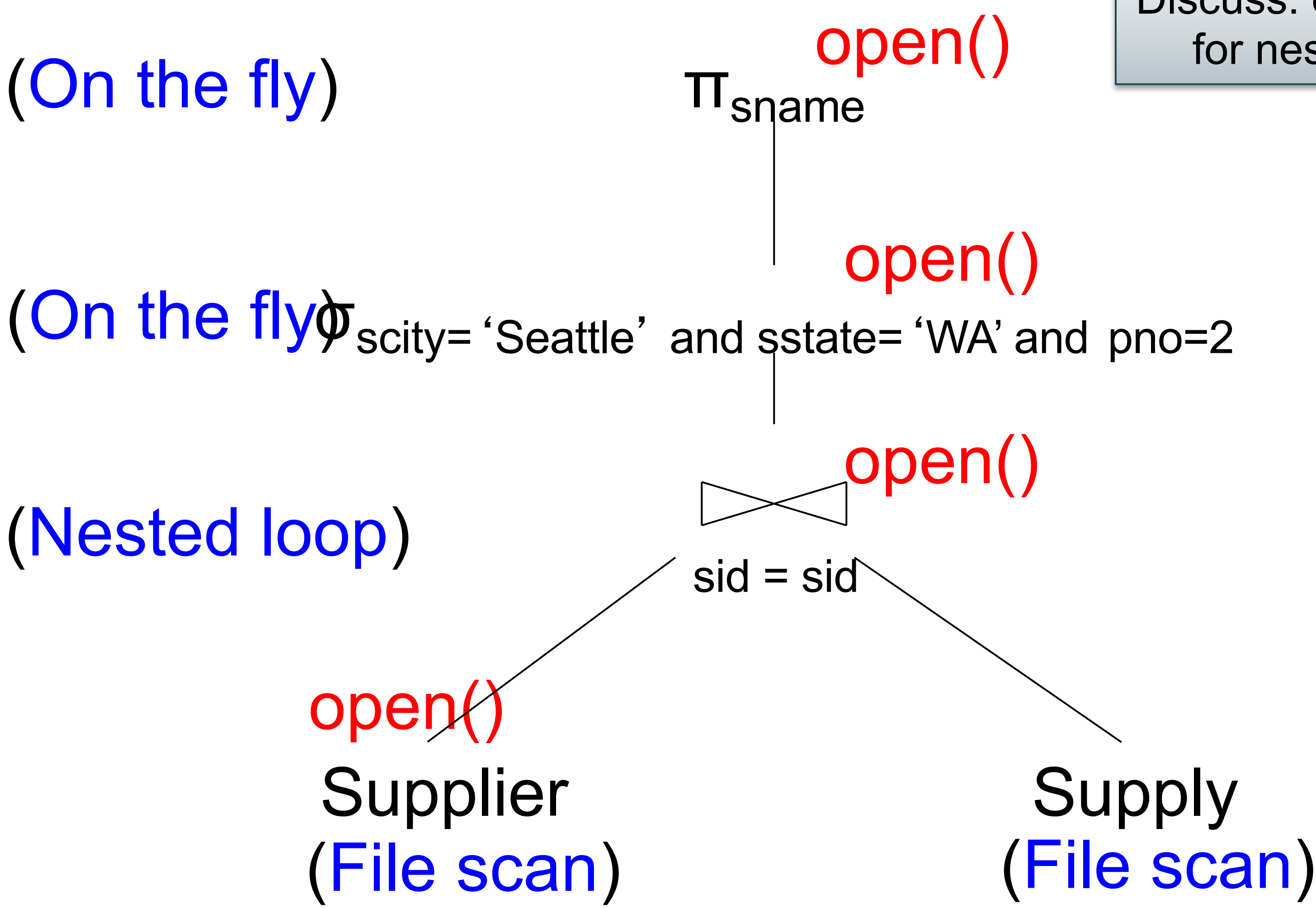
Discuss: open/next/close for nested loop join



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

# Pipelining

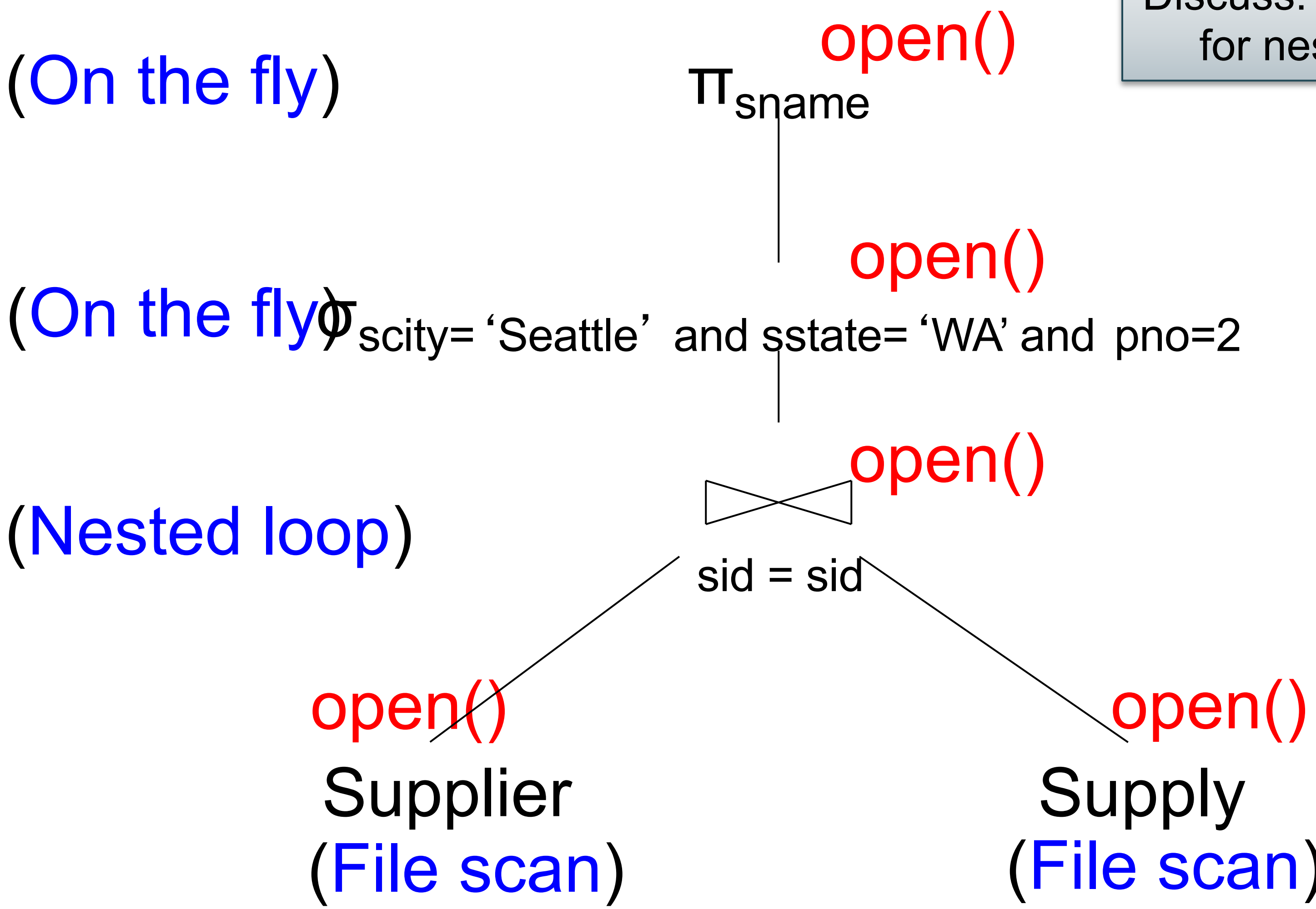
Discuss: open/next/close for nested loop join



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

# Pipelining

Discuss: open/next/close for nested loop join



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

# Pipelining

Discuss: open/next/close for nested loop join

next()

(On the fly) $\pi_{sname}$

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

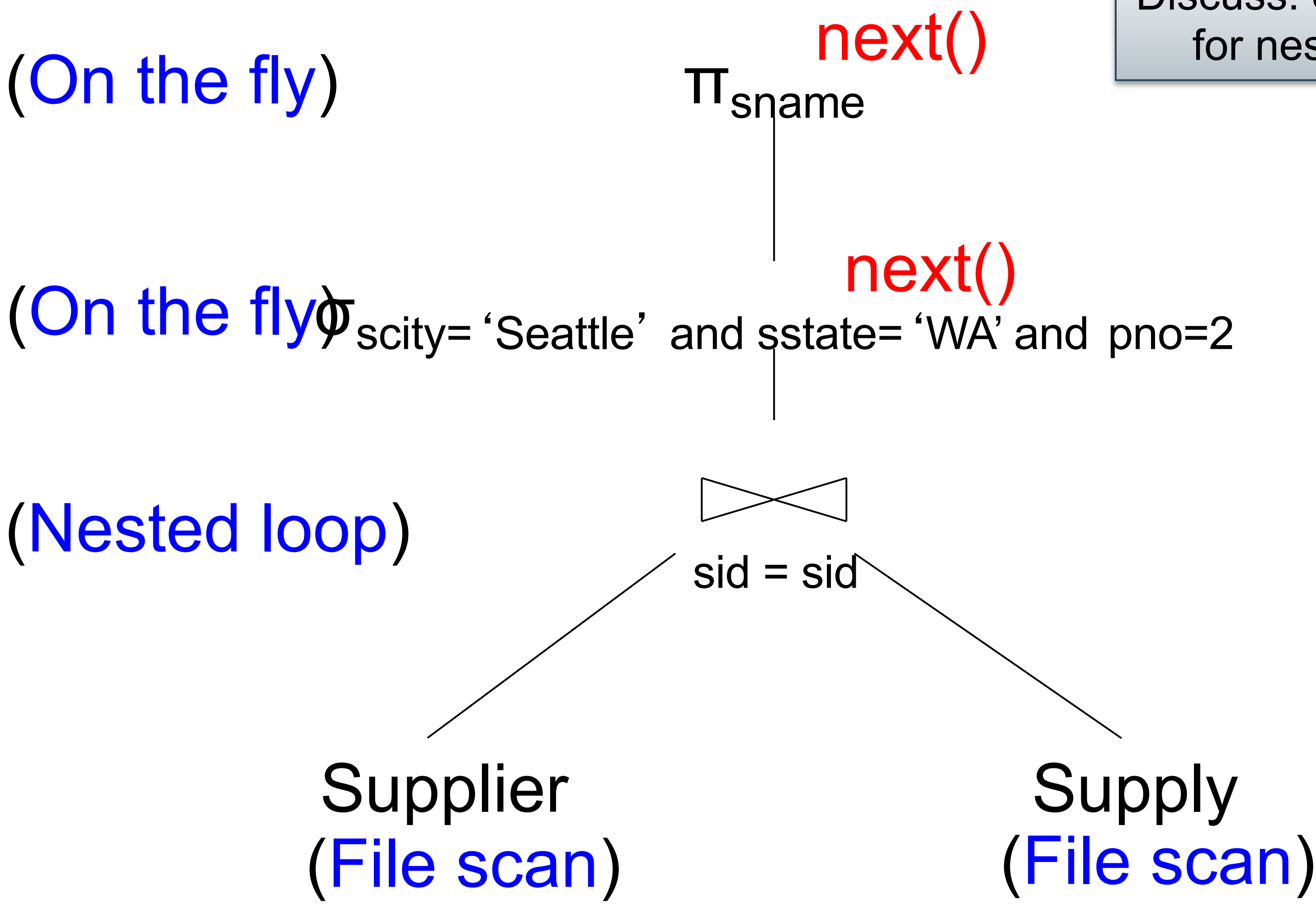
(Nested loop) $\bowtie_{sid = sid}$

Supplier (File scan) | Supply (File scan)

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

# Pipelining

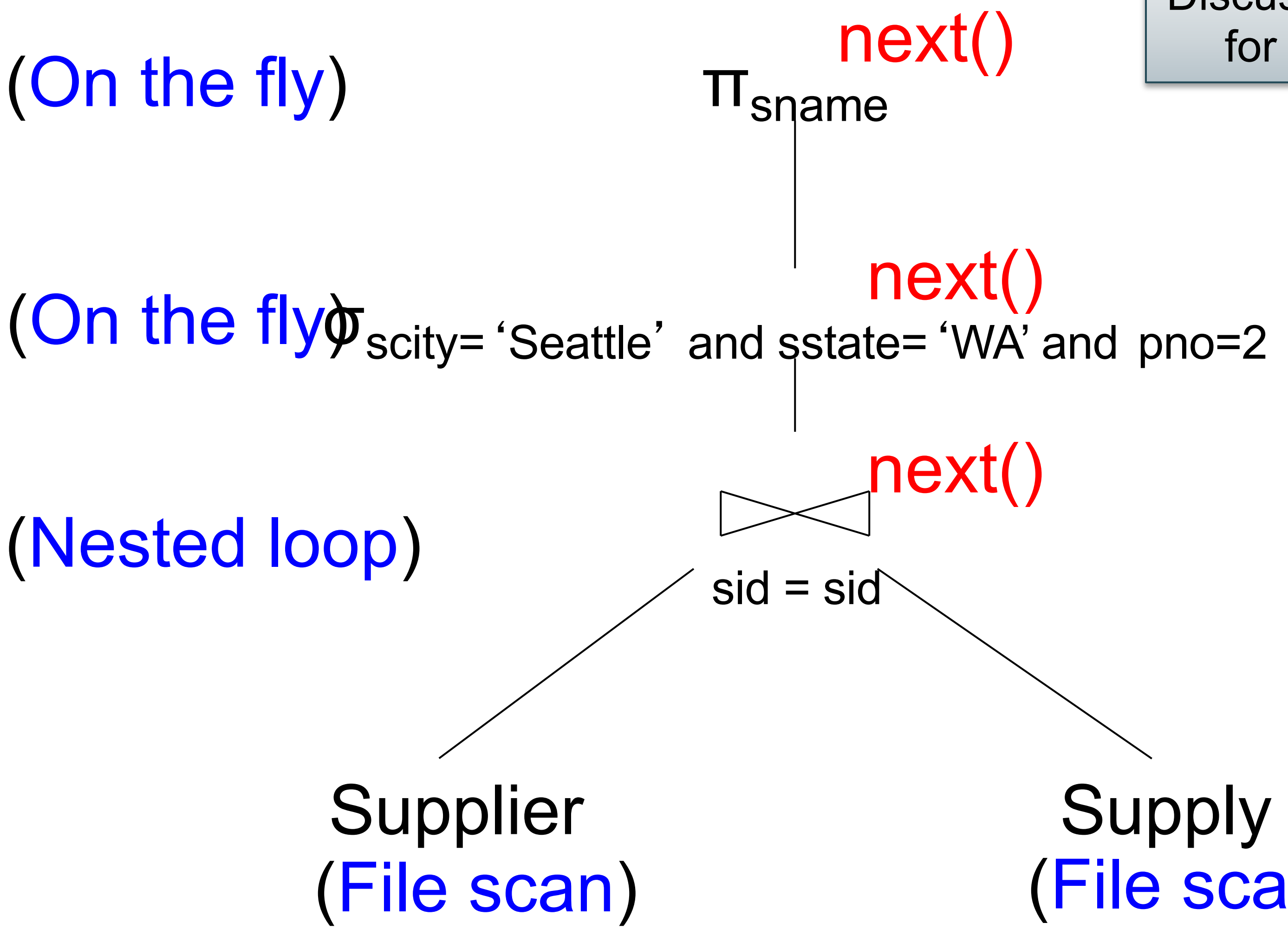
Discuss: open/next/close for nested loop join



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

# Pipelining

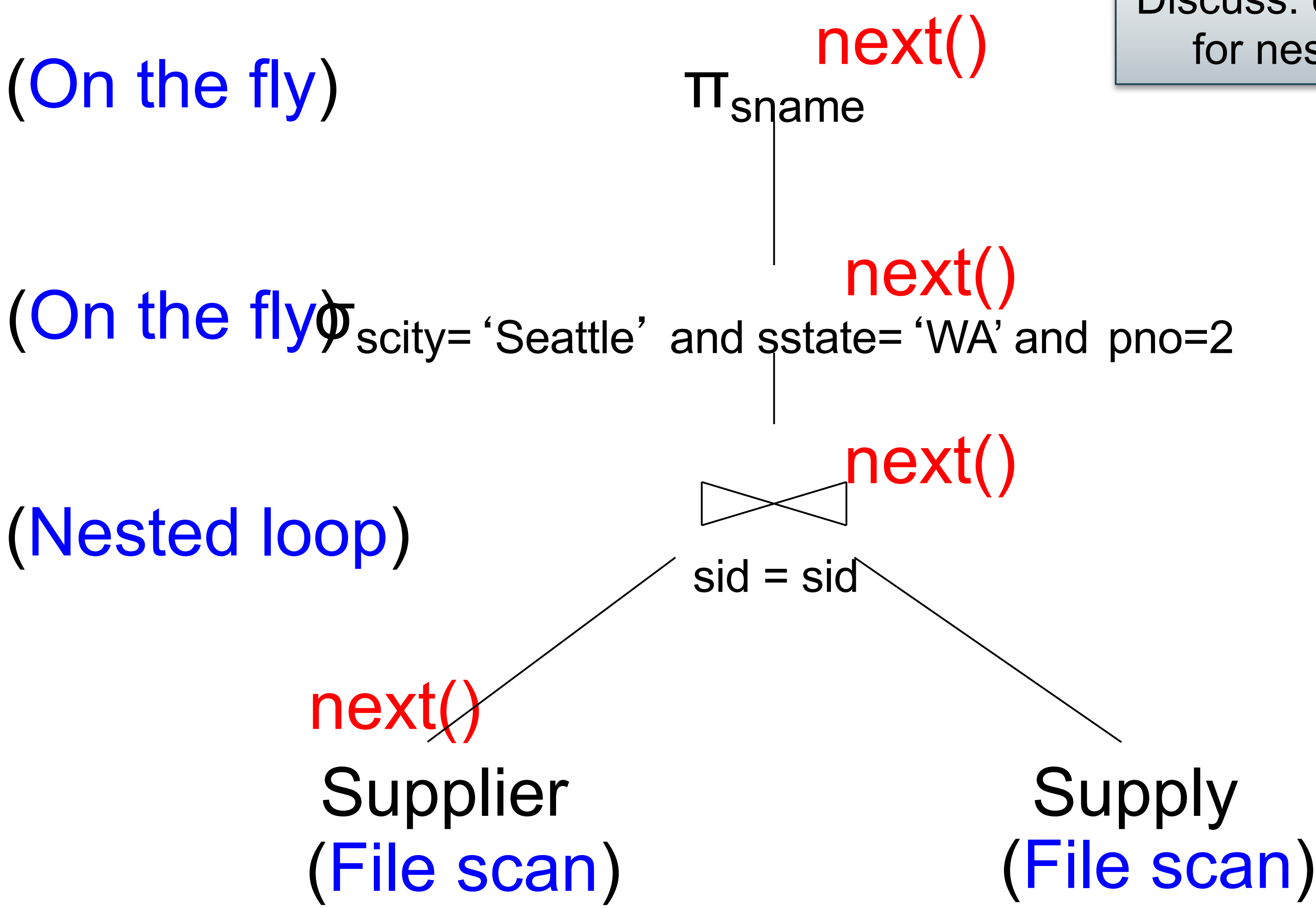
Discuss: open/next/close for nested loop join



```
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: open/next/close for nested loop join

(On the fly) $\pi_{sname}$ next()

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

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

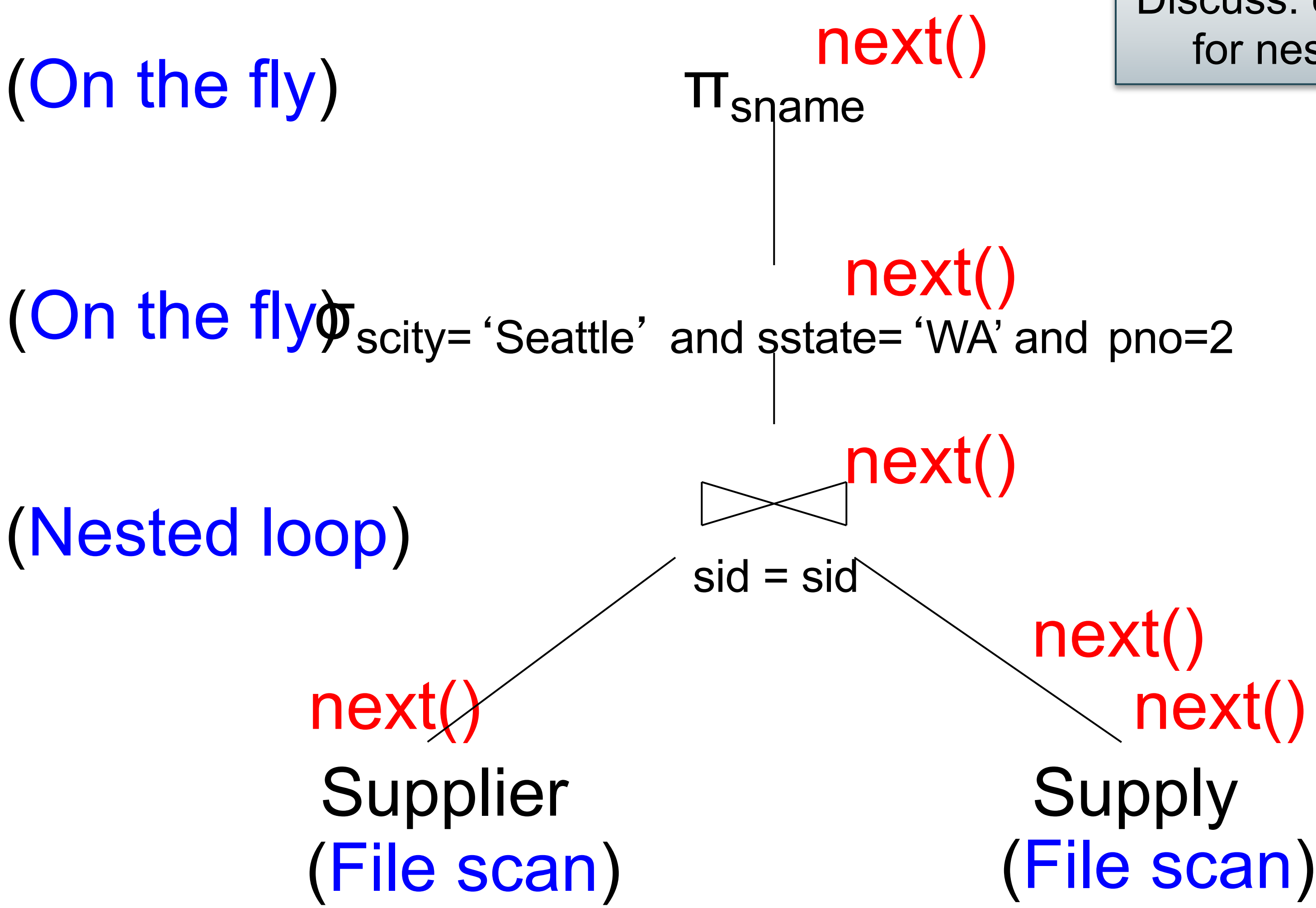
next() Supplier (File scan)

next() Supply (File scan)

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

# Pipelining

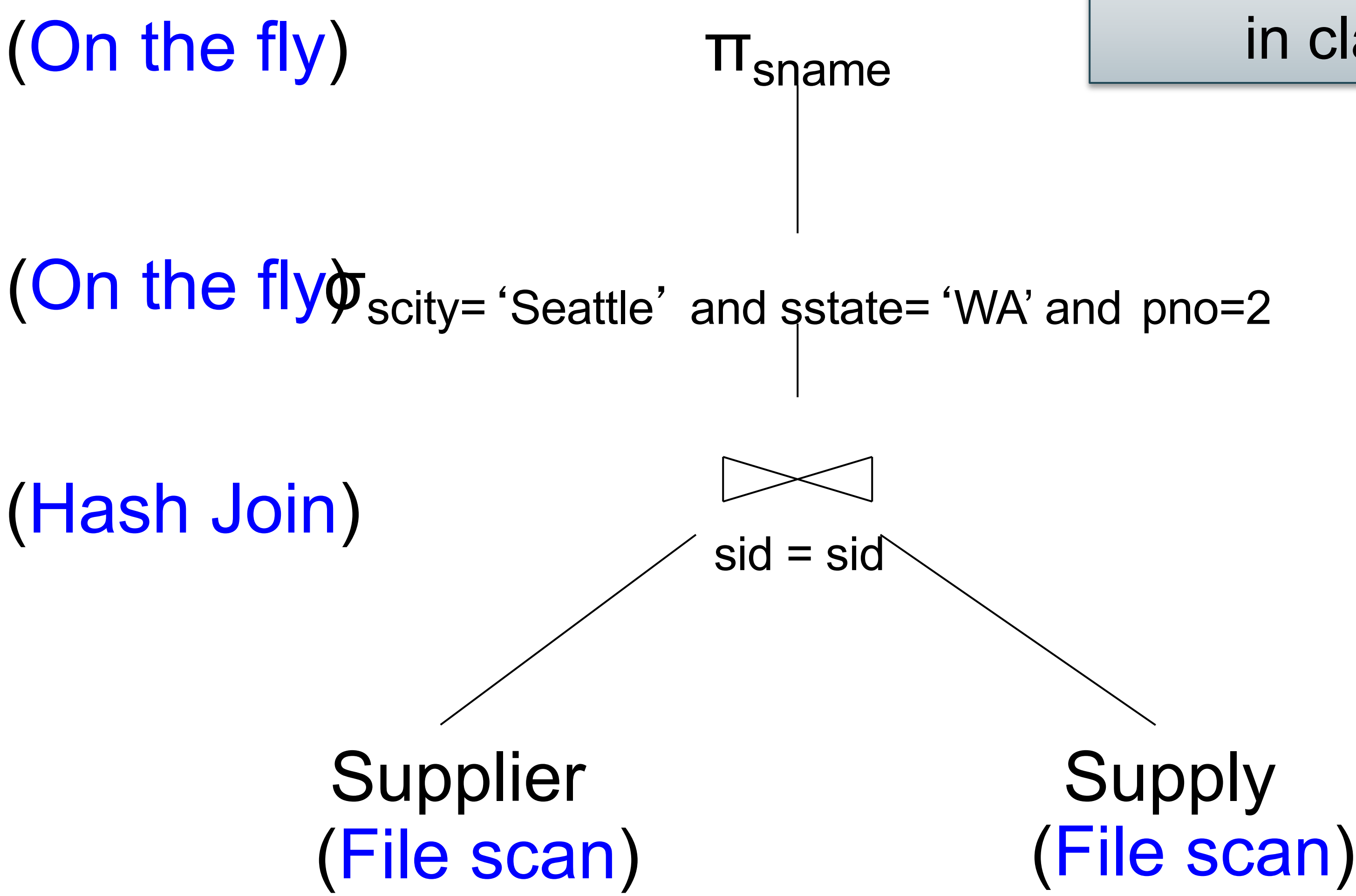
Discuss: open/next/close for nested loop join



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

# Pipelining

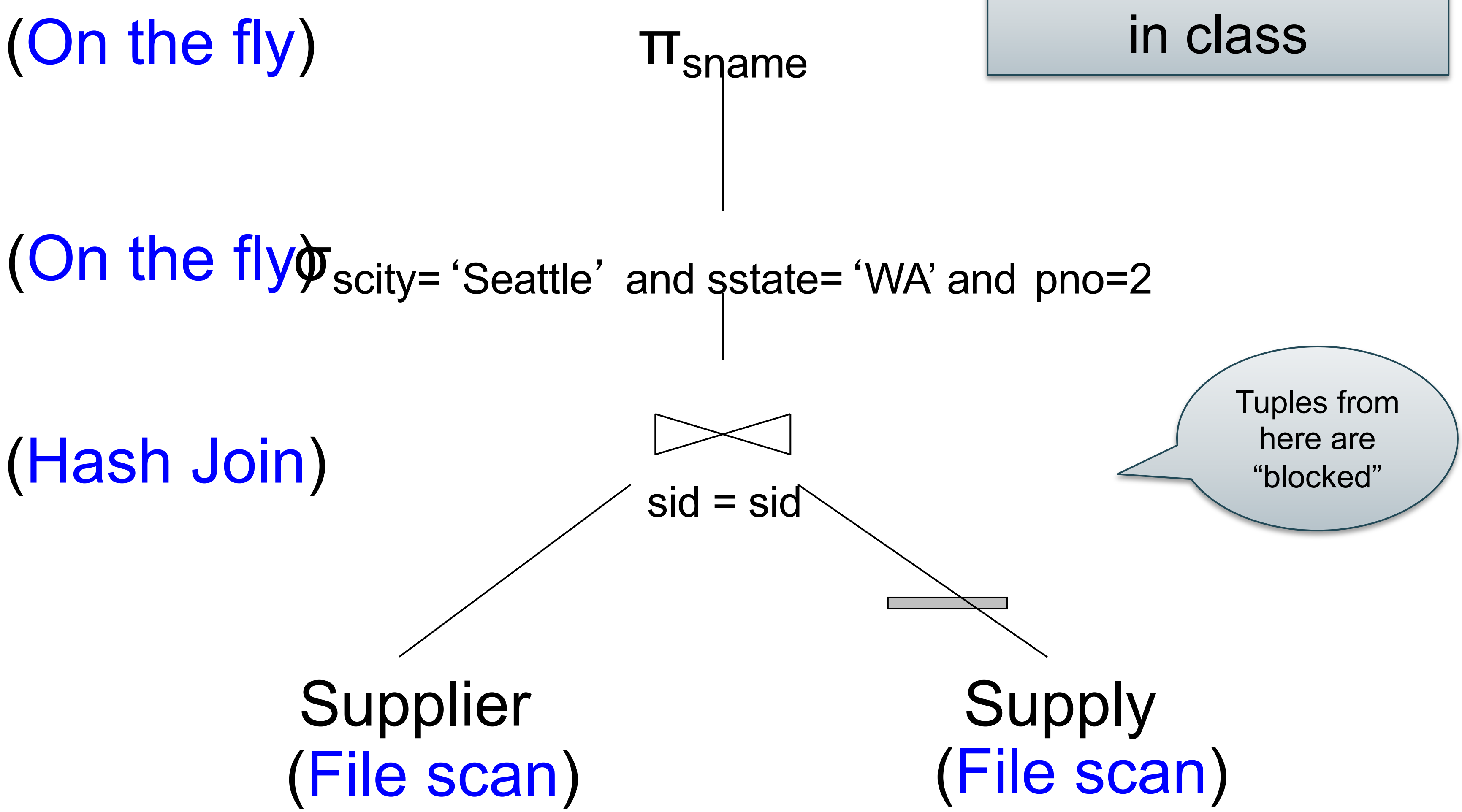
Discuss hash-join in class



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

# Pipelining

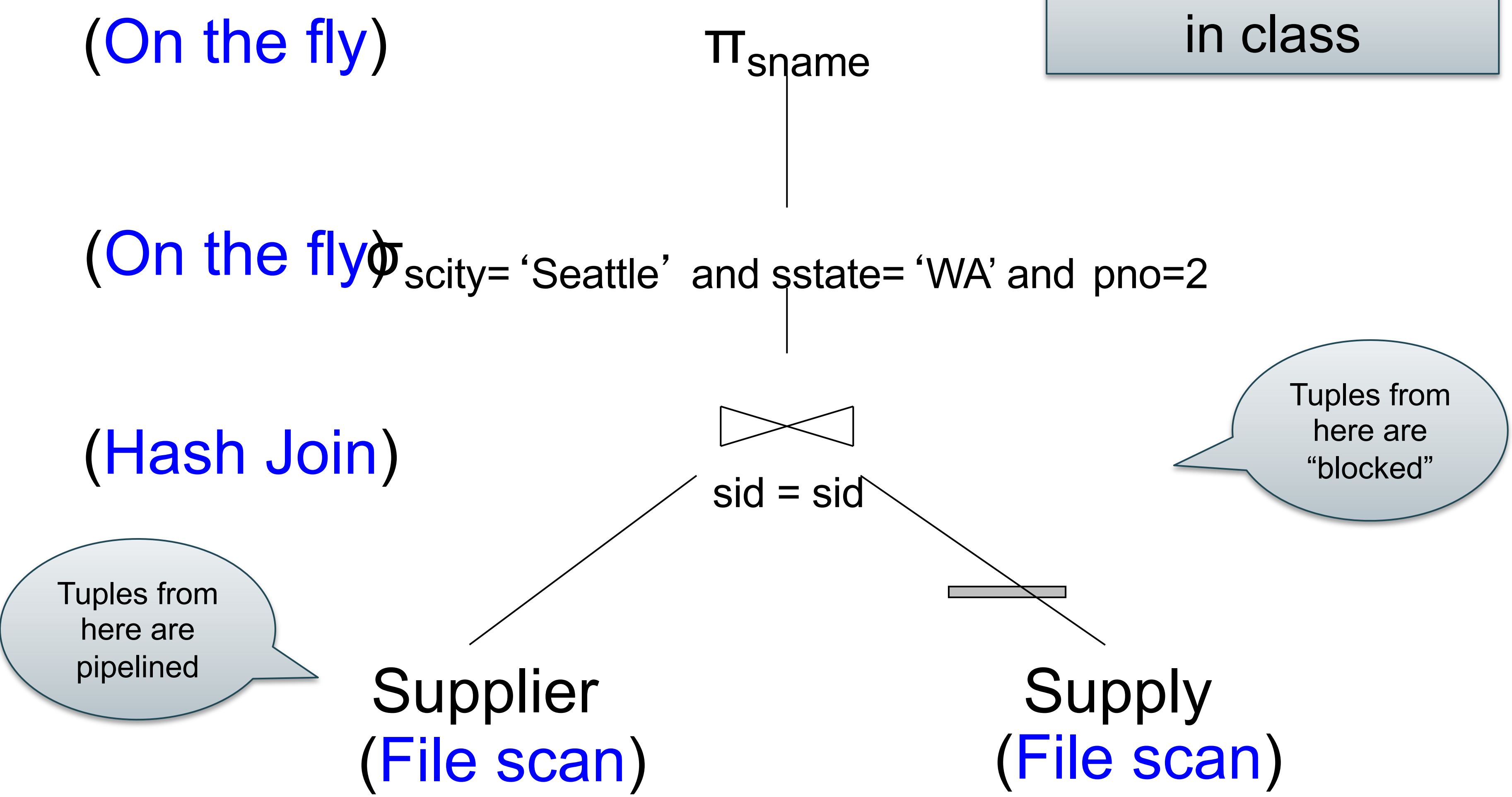
Discuss hash-join in class



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

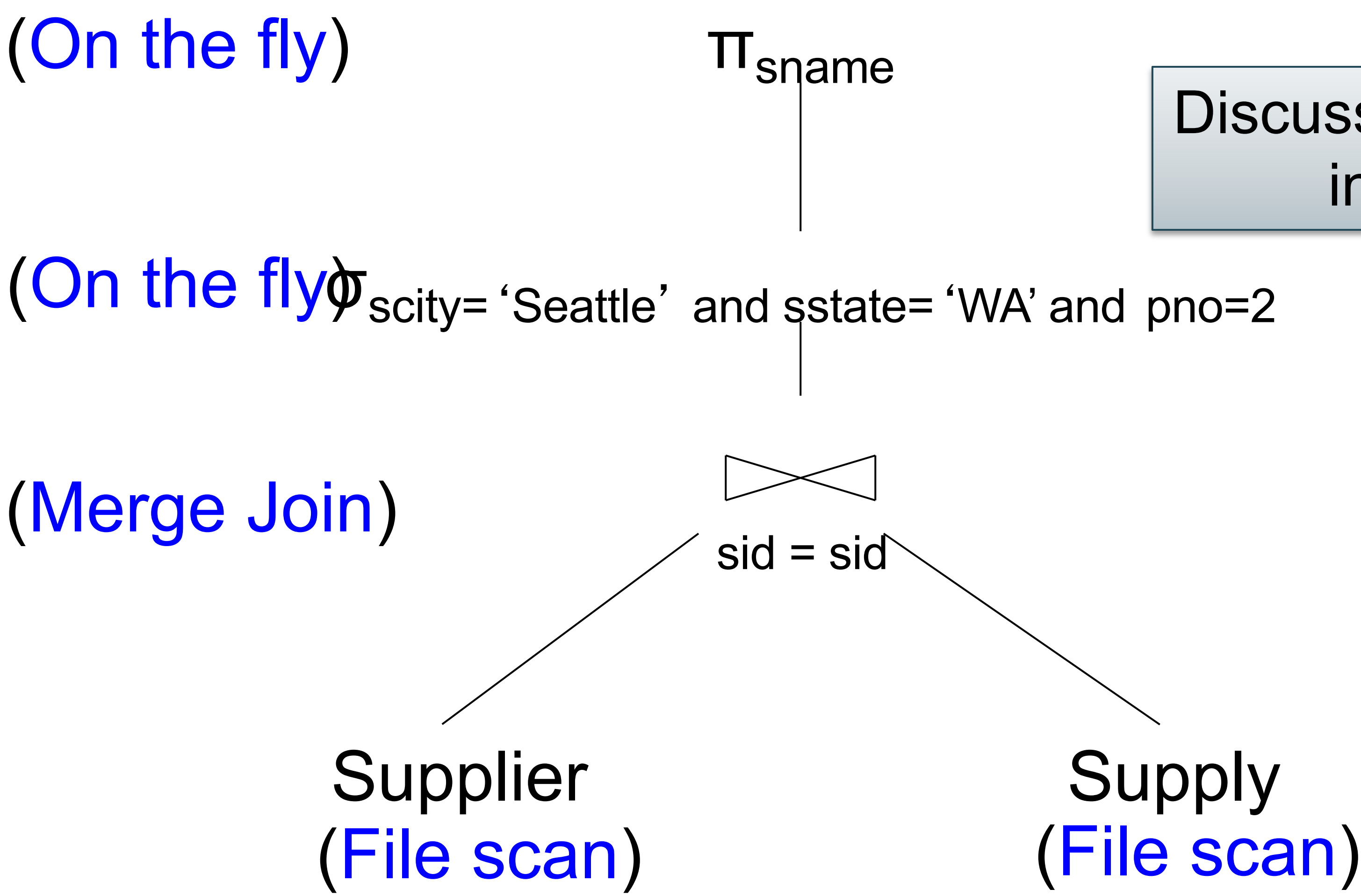
# Pipelining

Discuss hash-join in class



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

# Blocked Execution

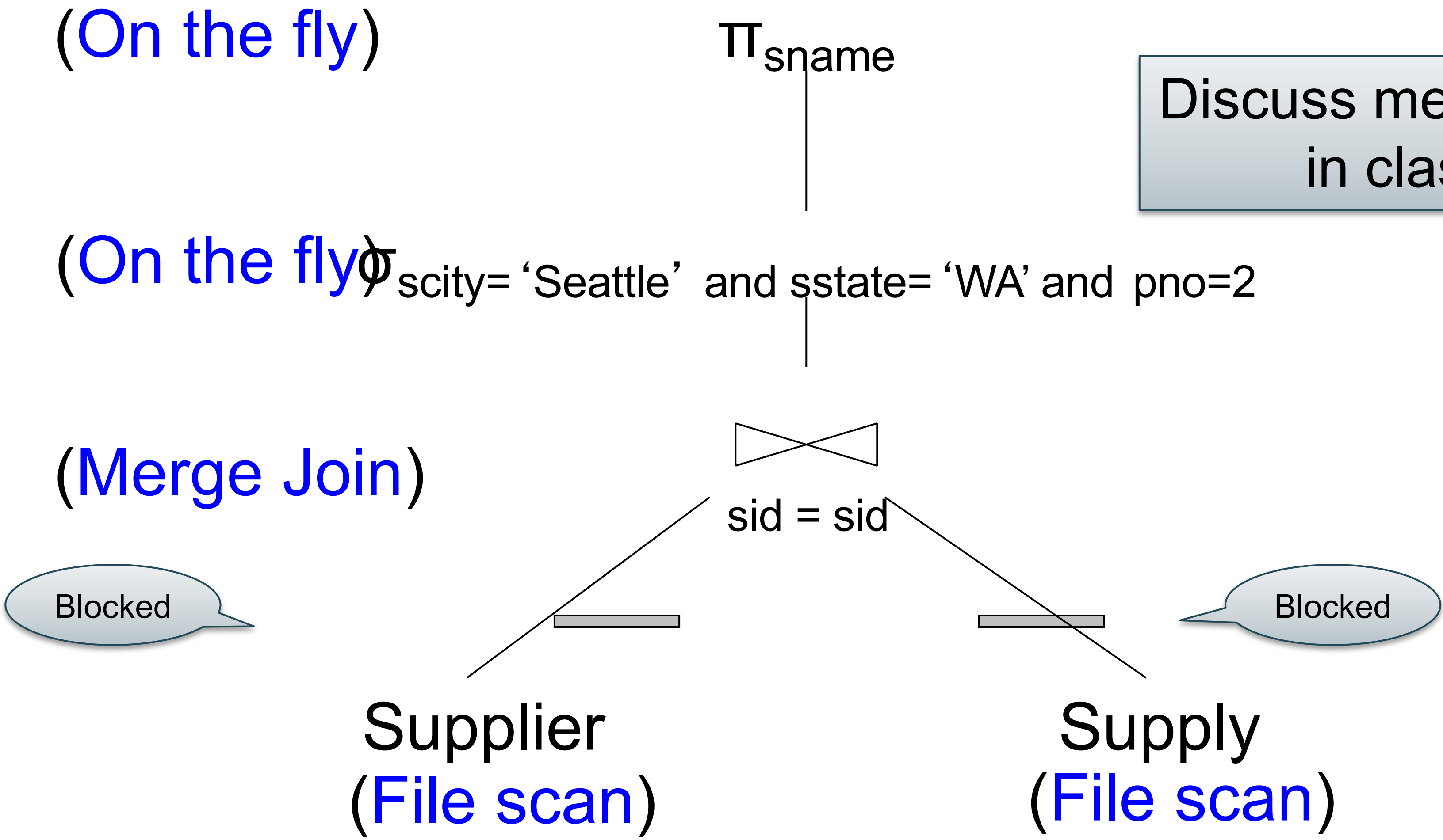



Discuss merge-join in class

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

# Blocked Execution

Discuss merge-join in class



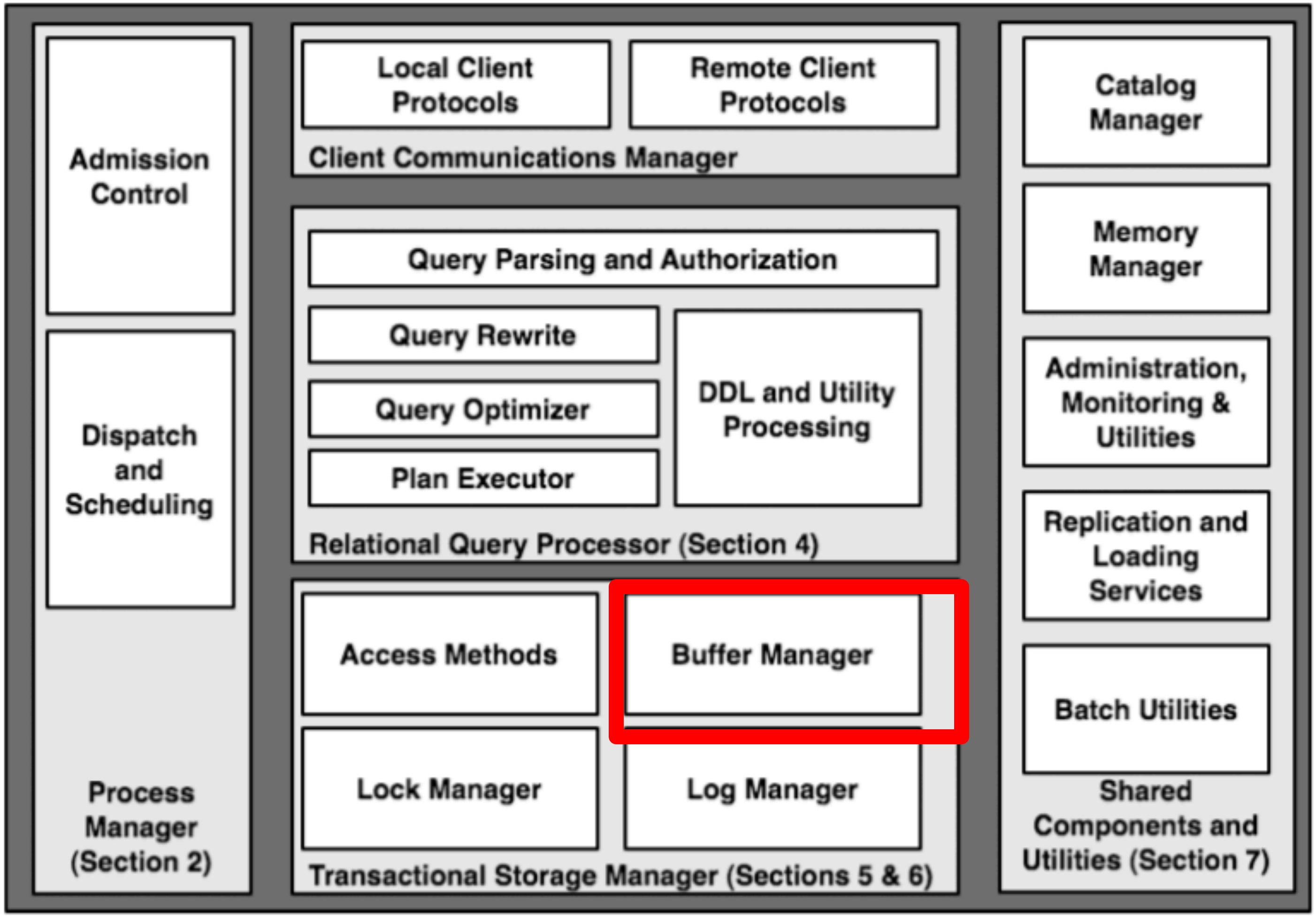
# Pipeline v.s. Blocking

- 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

# Outline

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

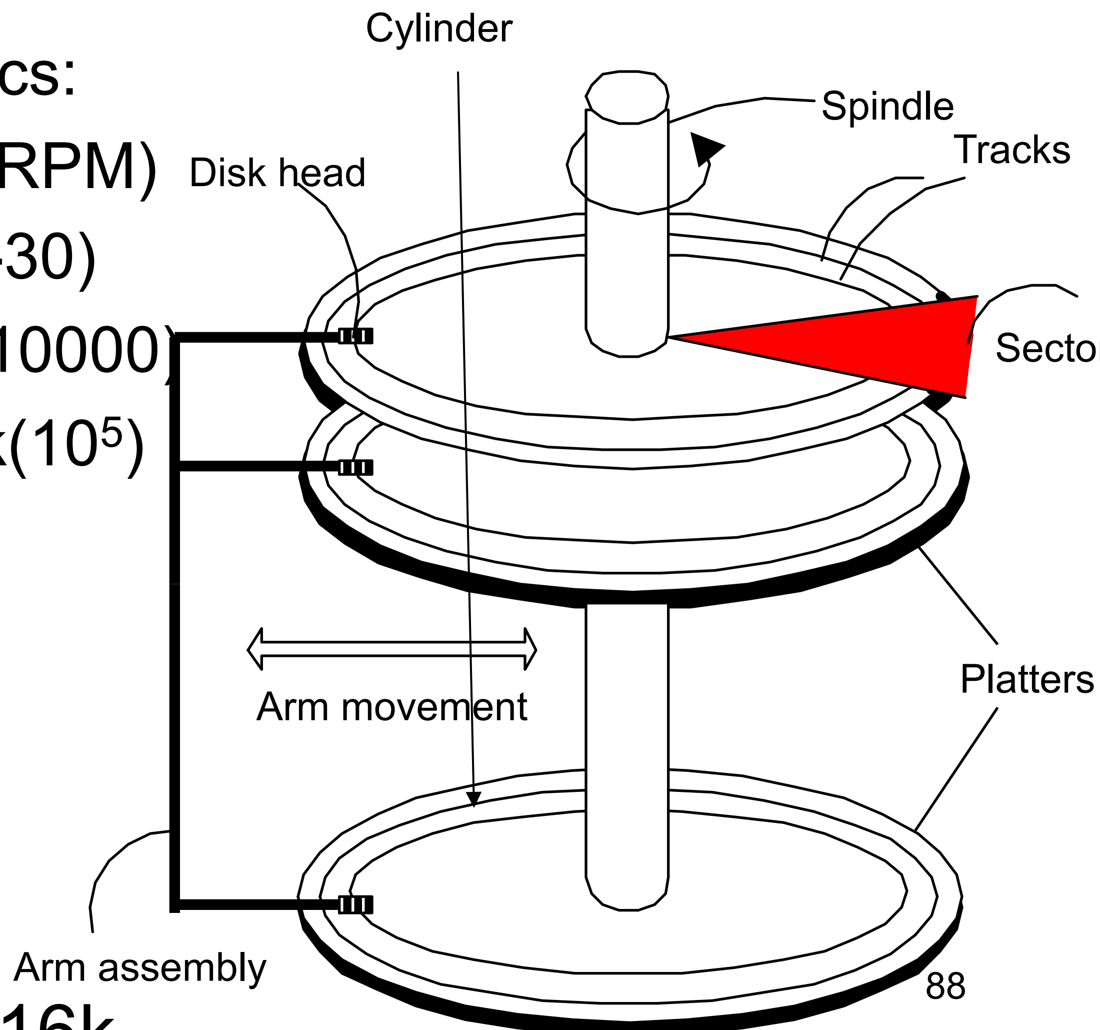
# Multiple Processes



# 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 | | | |

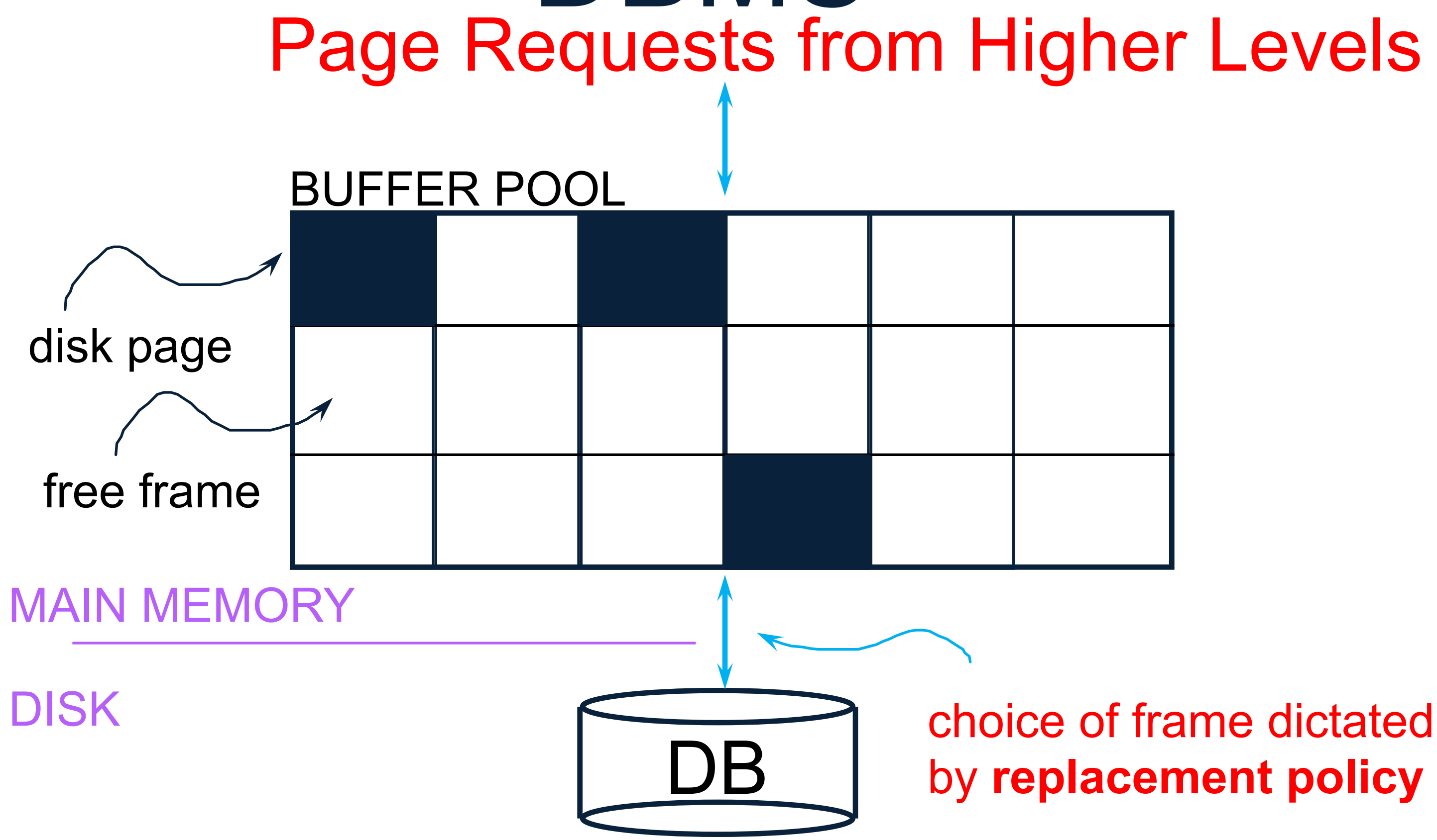
In the example, we have 4 blocks with 2 tuples each

# 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 factoid**: disks always read/write an entire block at a time

# Buffer Management in a DBMS




- 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!

# Issues

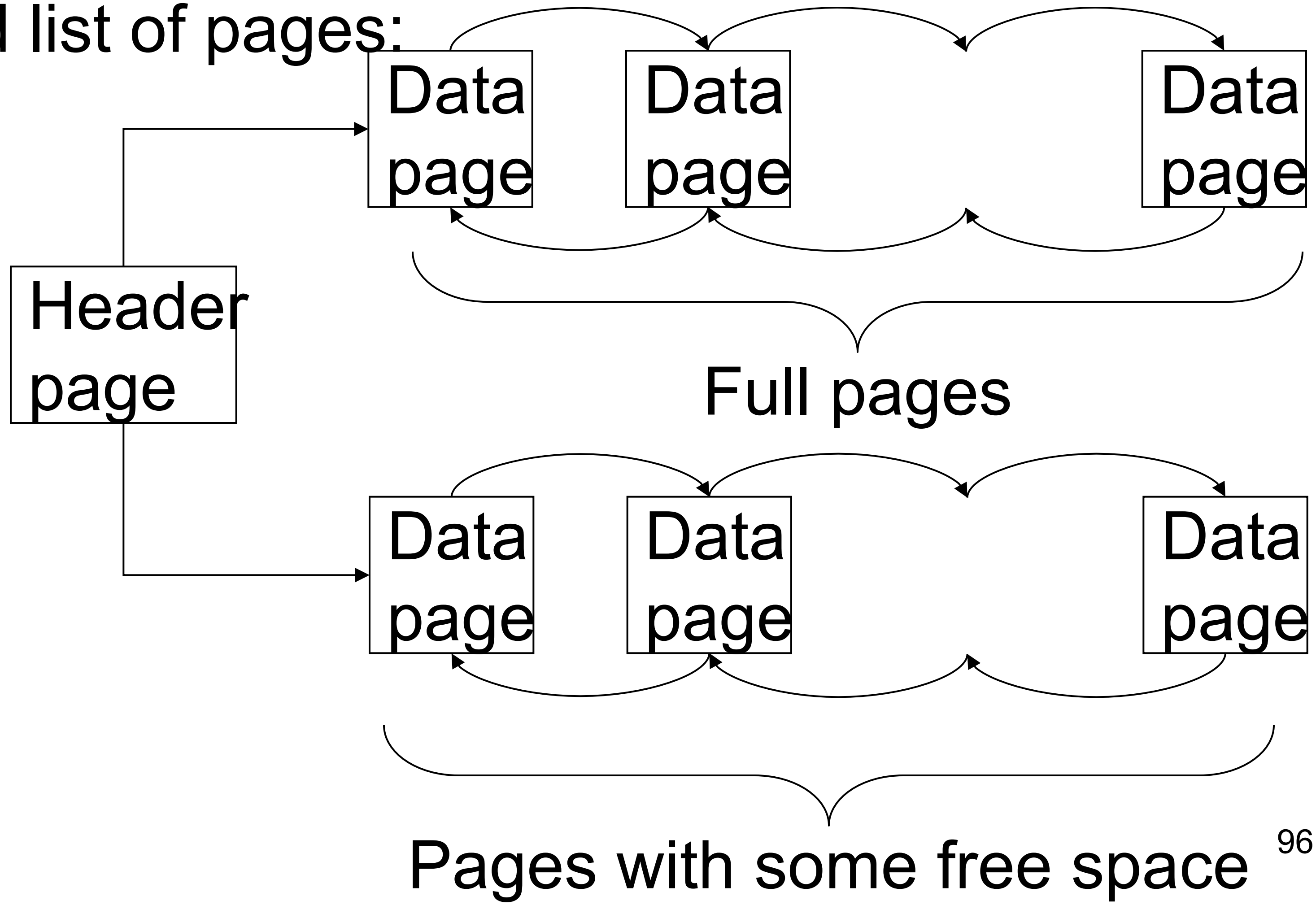
- Managing free blocks
- File Organization
- Represent the records inside a page
- Represent attributes inside the records

# Managing Free Blocks

- Linked list of free blocks
- Directory of pages
- Bit map

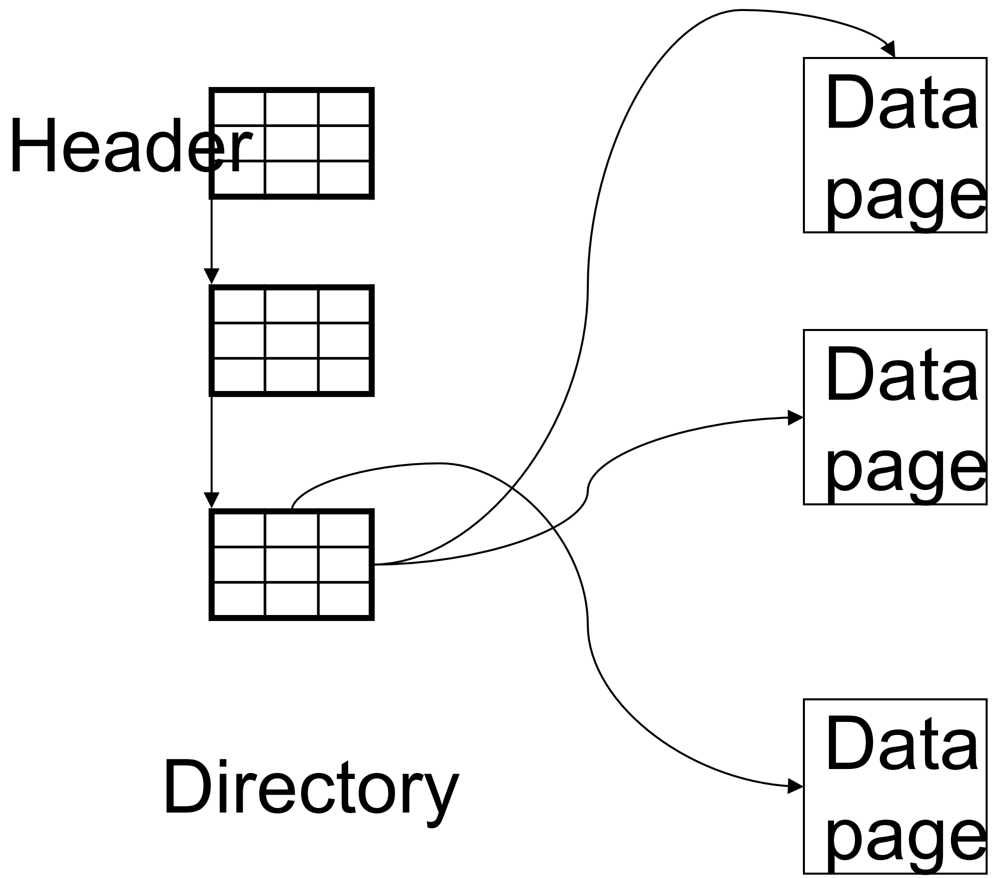
# File Organization

Linked list of pages:



# File Organization

Better: directory of pages



# File Organization

- Bit map: store compactly the free/full status of each page

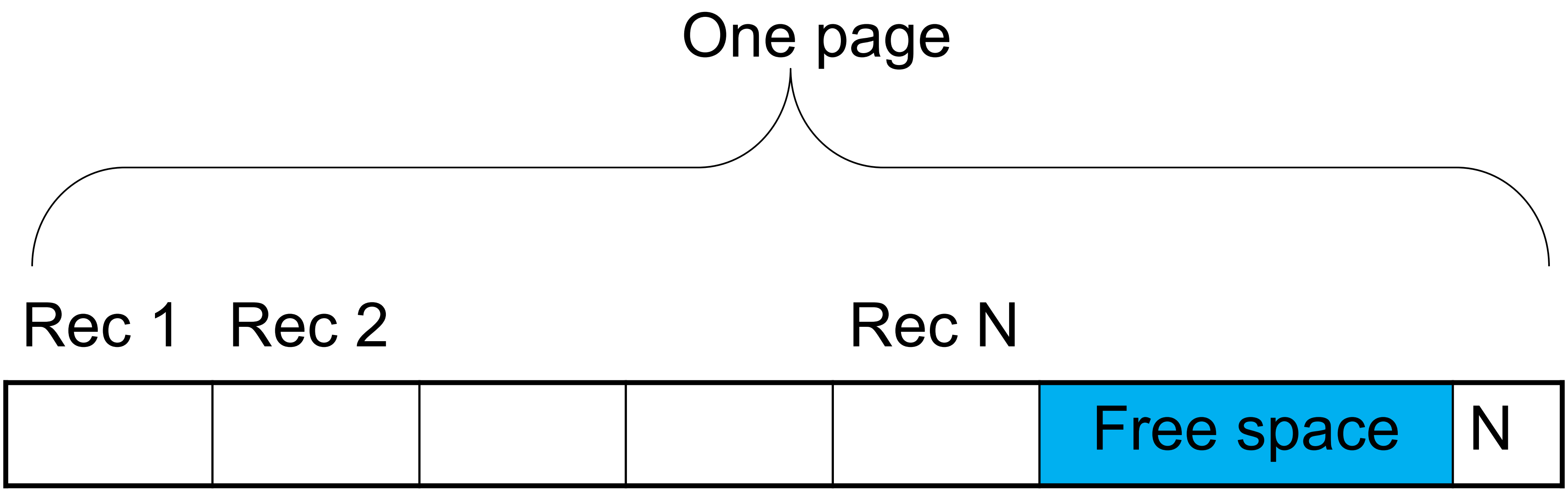
# Records into a Page

Issues to consider

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

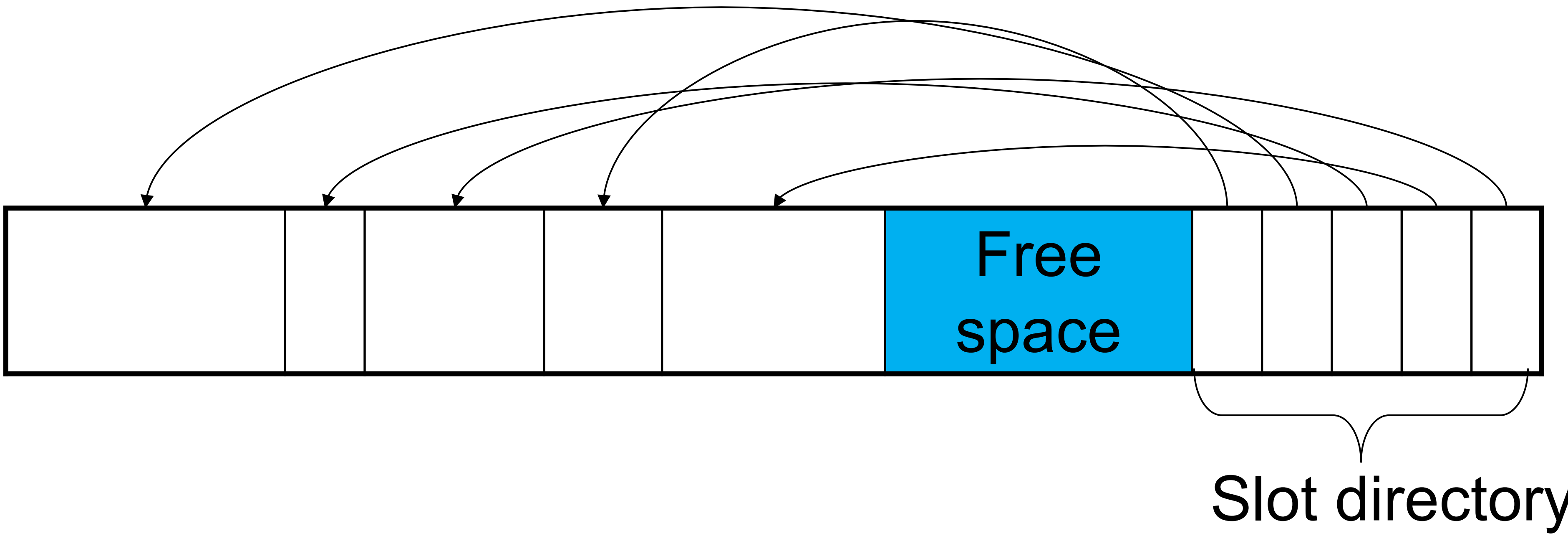
# Records into a Page

Fixed-length records: packed representation




Problems ?

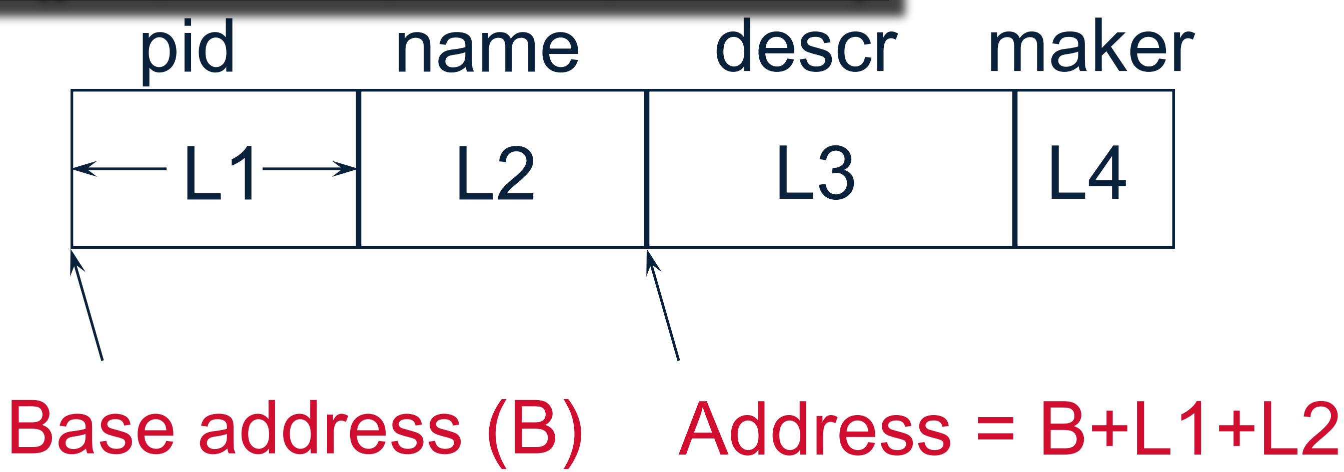
# Records into a Page




Variable-length records

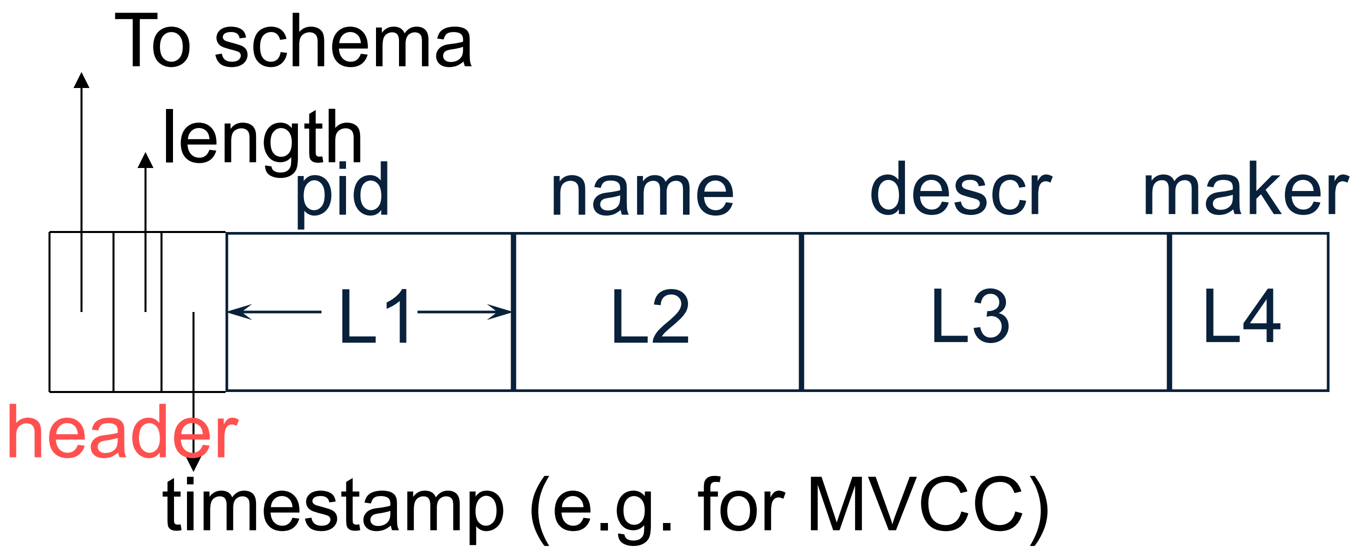
# Record Formats: Fixed Length

Product(pid, name, descr, maker)




- Information about field types same for all records in a file; stored in *system catalogs.*
- Finding *i'th* field requires scan of record.
- Note the importance of schema information!

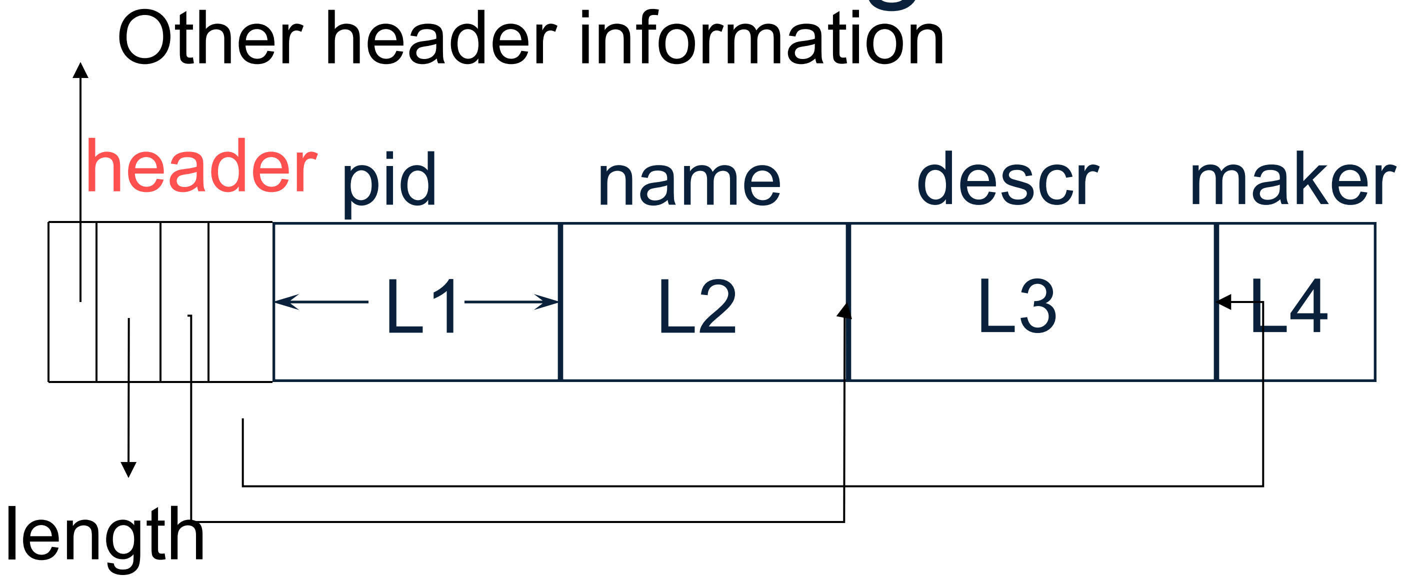
# Record Header




Need the header because:

- The schema may change
  for a while new+old may coexist
- Records from different relations may coexist

# Variable Length Records




Place the fixed fields first: F1

Then the variable length fields: F2, F3, F4

Null values take 2 bytes only

Sometimes they take 0 bytes (when at the end)

# BLOB

- Binary large objects
- Supported by modern database systems
- E.g. images, sounds, etc.
- Storage: attempt to cluster blocks together

CLOB = character large object

- Supports only restricted operations

# 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'
- Indexe: 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

# Index

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

Key = means here search key

# This  Is Not A Key

Different keys:

- Primary key – uniquely identifies a tuple
- Key of the sequential file – how the data file is sorted, if at all
- Index key – how the index is organized



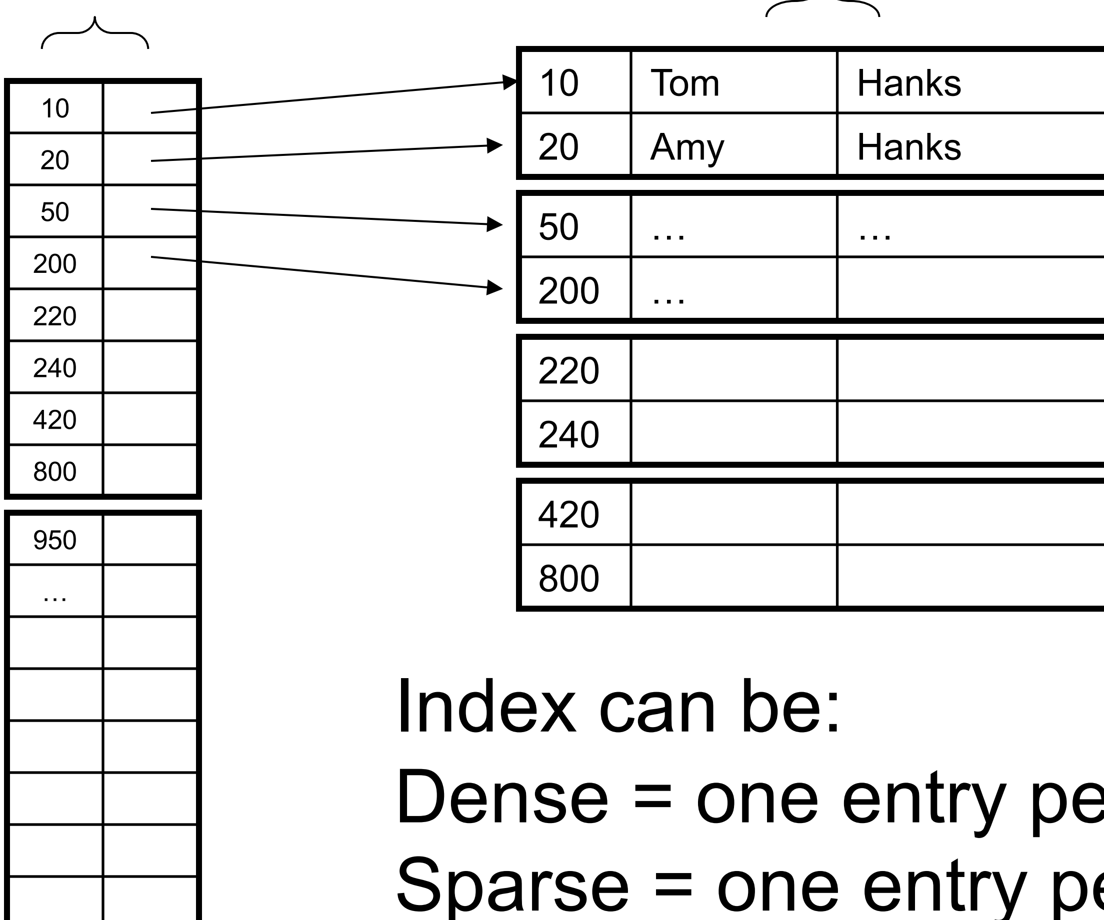

# Example 1: Index on ID

**Student**

| ID | fName | lName |
|---|---|---|
| 10 | Tom | Hanks |
| 20 | Amy | Hanks |
| … | | |

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

Data File **Student**




Index can be:

Dense = one entry per record

Sparse = one entry per block

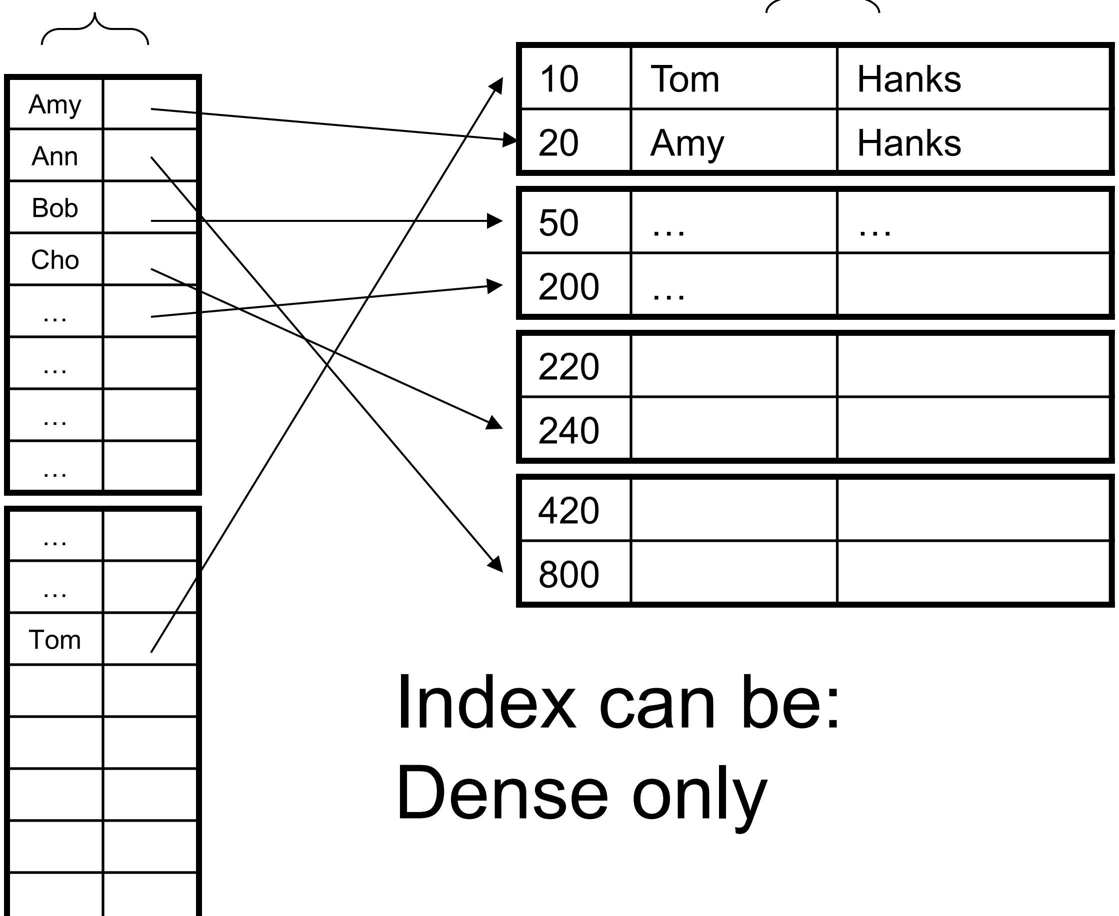
**Student**

| ID | fName | lName |
|---|---|---|
| 10 | Tom | Hanks |
| 20 | Amy | Hanks |
| … | | |

# Example 2: Index on fName

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

Data File **Student**




Index can be:
Dense only

# Index Organization

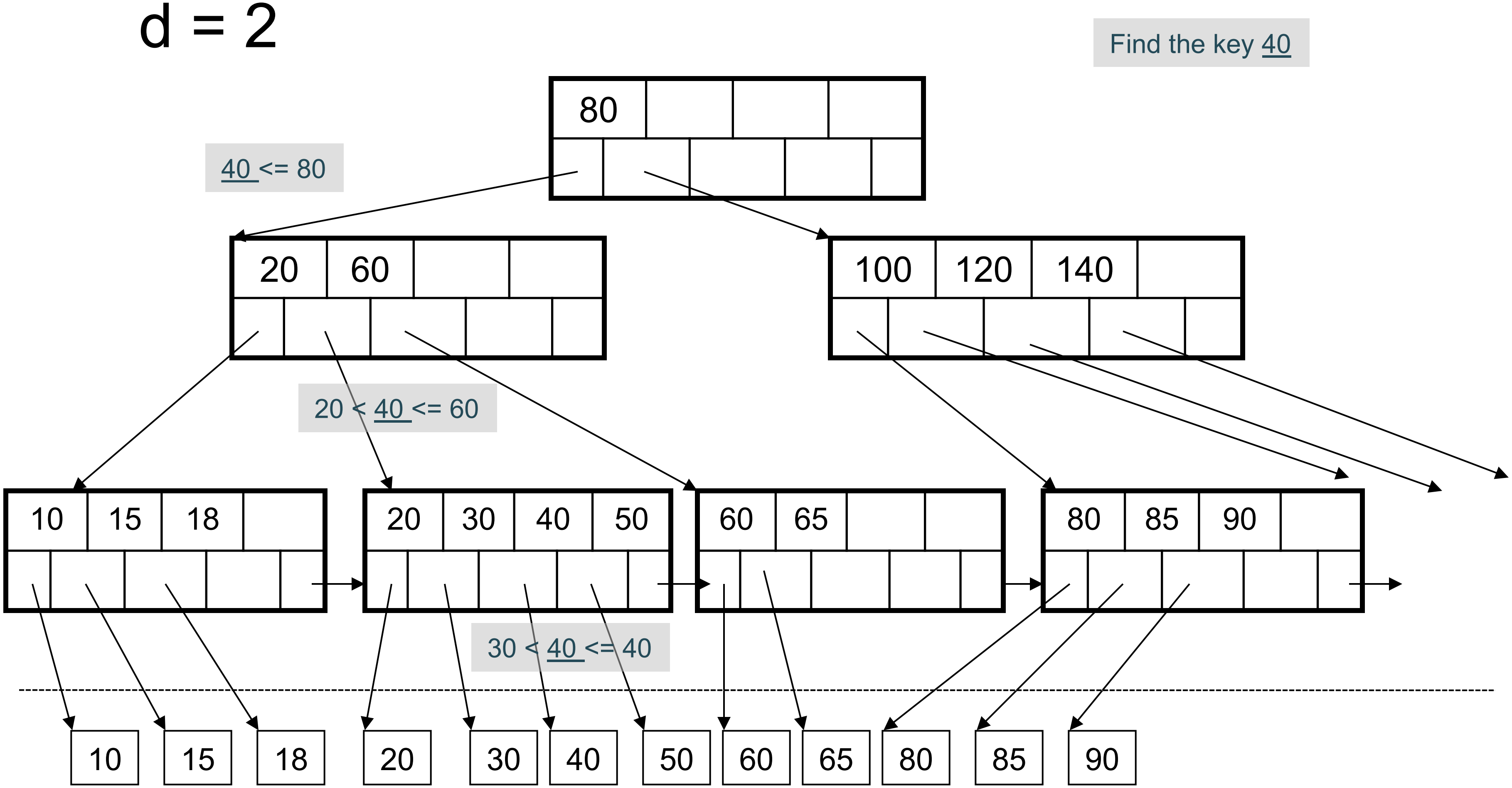
- Hash table

- B+ trees – most common
  - They are search trees, but they are not binary instead have higher fan-out
  - Will discuss them briefly next

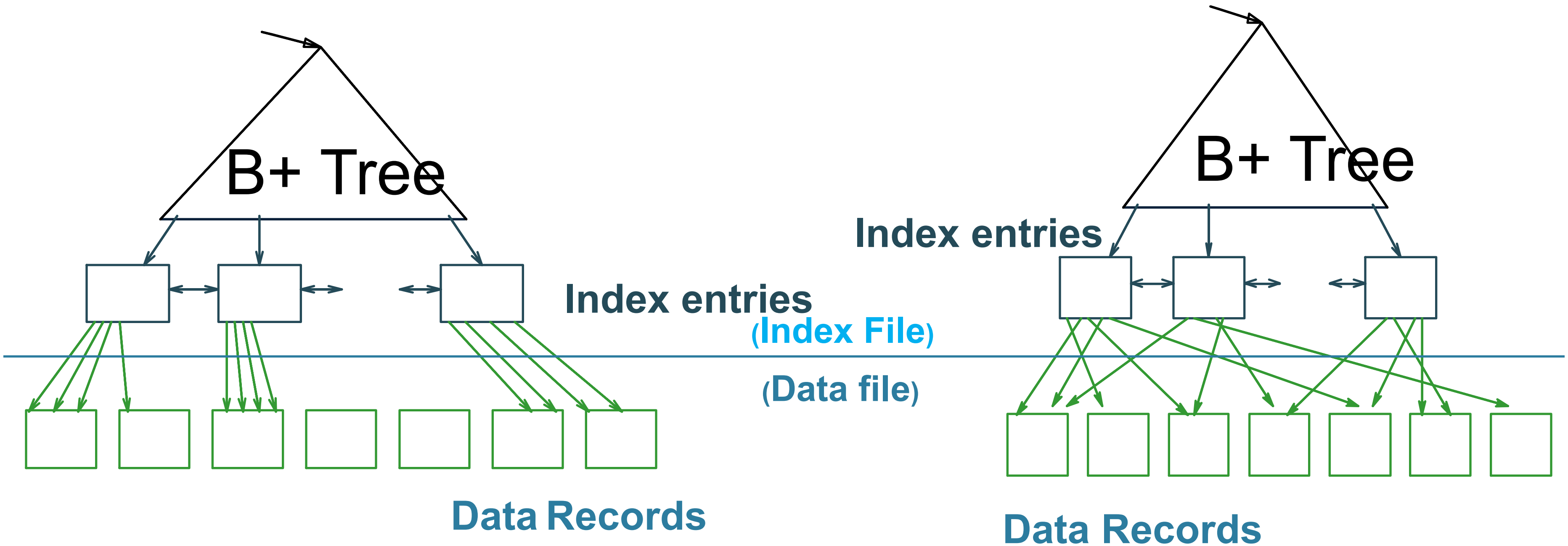
- Specialized indexes: bit maps, R-trees, inverted index; won't discuss

# B+ Tree Index by Example

d = 2



# Clustered vs Unclustered




**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

# Getting Practical: Creating Indexes in SQL

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

# Getting Practical: Creating Indexes in SQL

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

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

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

# Getting Practical: Creating Indexes in SQL

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

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

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

What does this mean?

# Getting Practical: Creating Indexes in SQL

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

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

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

What does this mean?

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

# Getting Practical: Creating Indexes in SQL

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

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

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

What does this mean?

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

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

# Getting Practical: Creating Indexes in SQL

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

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

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

What does this mean?

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

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

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

# Getting Practical: Creating Indexes in SQL

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

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

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

What does this mean?

yes

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

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

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

# Getting Practical: Creating Indexes in SQL

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

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

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

What does this mean?

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

yes

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

yes

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

# Getting Practical: Creating Indexes in SQL

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

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

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

What does this mean?

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

yes

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

yes

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

no

# Getting Practical: Creating Indexes in SQL

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

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

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

What does this mean?

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

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

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

Postgres: CLUSTER V4

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

yes

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

yes

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

no

# Which Indexes?

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

# Which Indexes?

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

**This is called the *Index Selection Problem***

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

# 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=?
```

# 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 ?

# 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)

# 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 ?

# 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)

# 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 ?

# Index Selection Problem 3

V(M, N, P);

Your workload is this

100000 queries:

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

1000000 queries:

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

100000 queries:

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

A: V(N, P)

How does this index differ from:

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

# 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 ?

# Index Selection Problem 4

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

Your workload is this

1000 queries:

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

100000 queries:

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

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

# Two typical kinds of queries

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

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

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

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

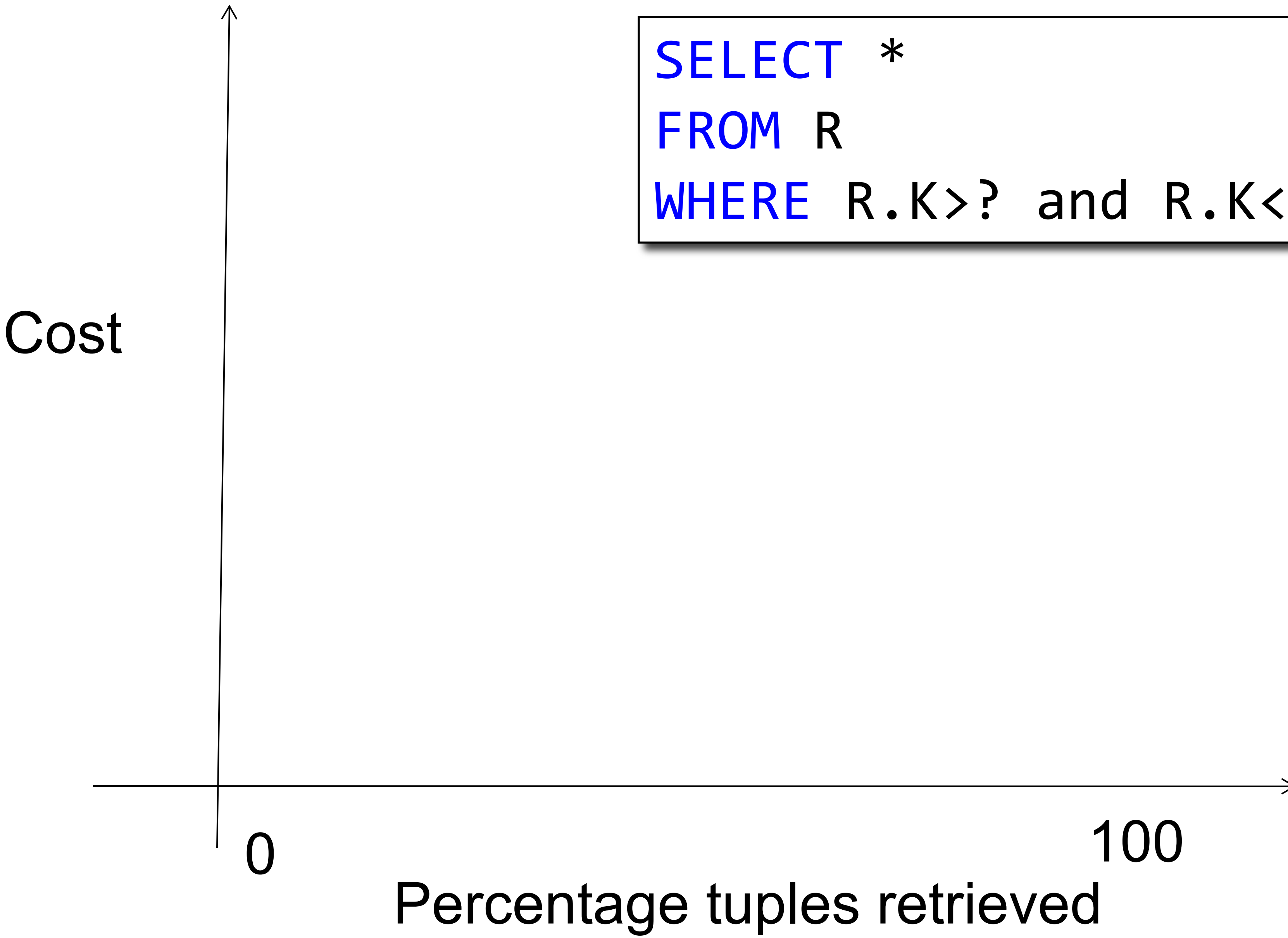
# To Cluster or Not

Remember:

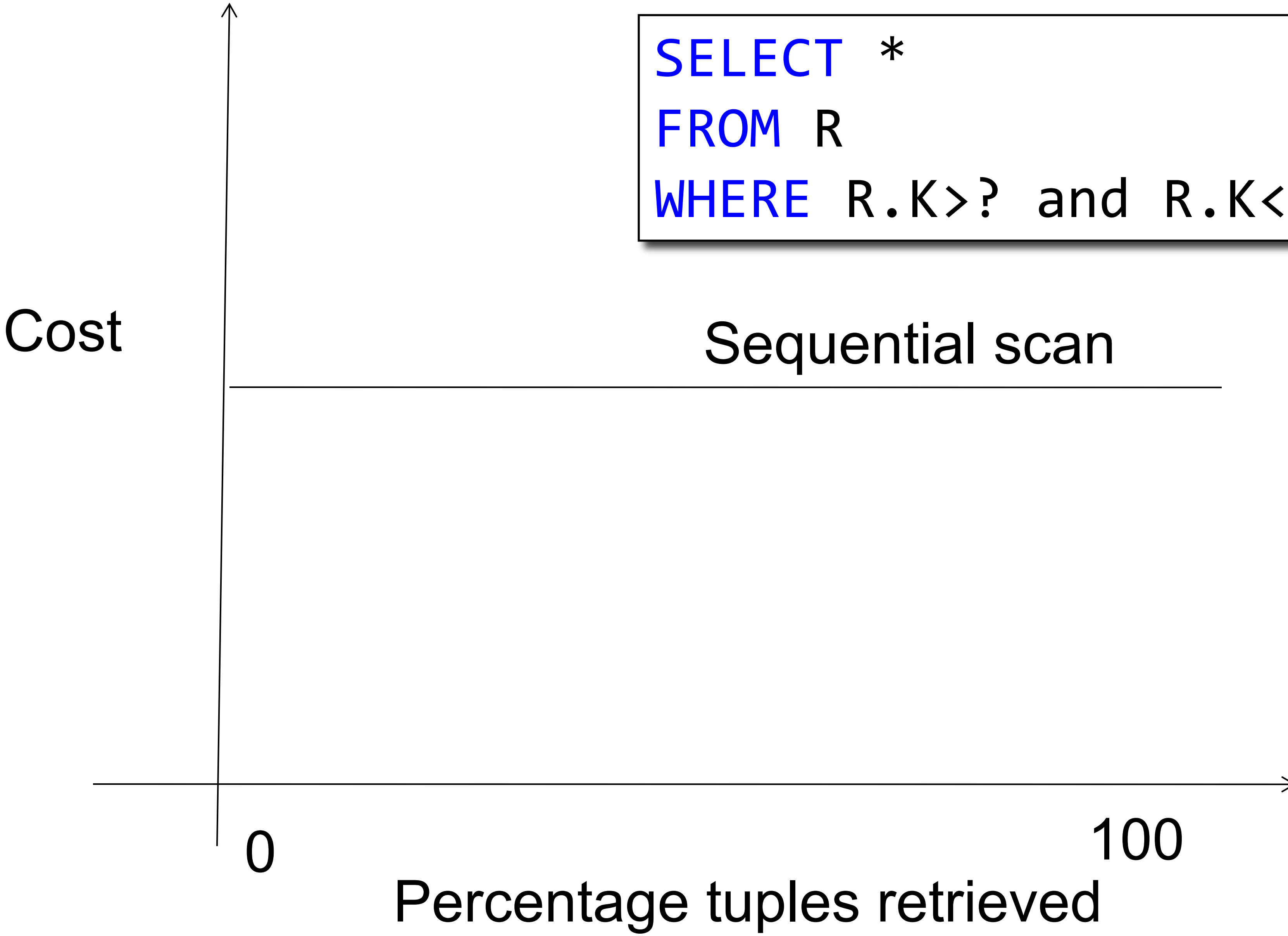
- Rule of thumb:
  Random reading 1-2% of file ≈ sequential scan entire file;

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

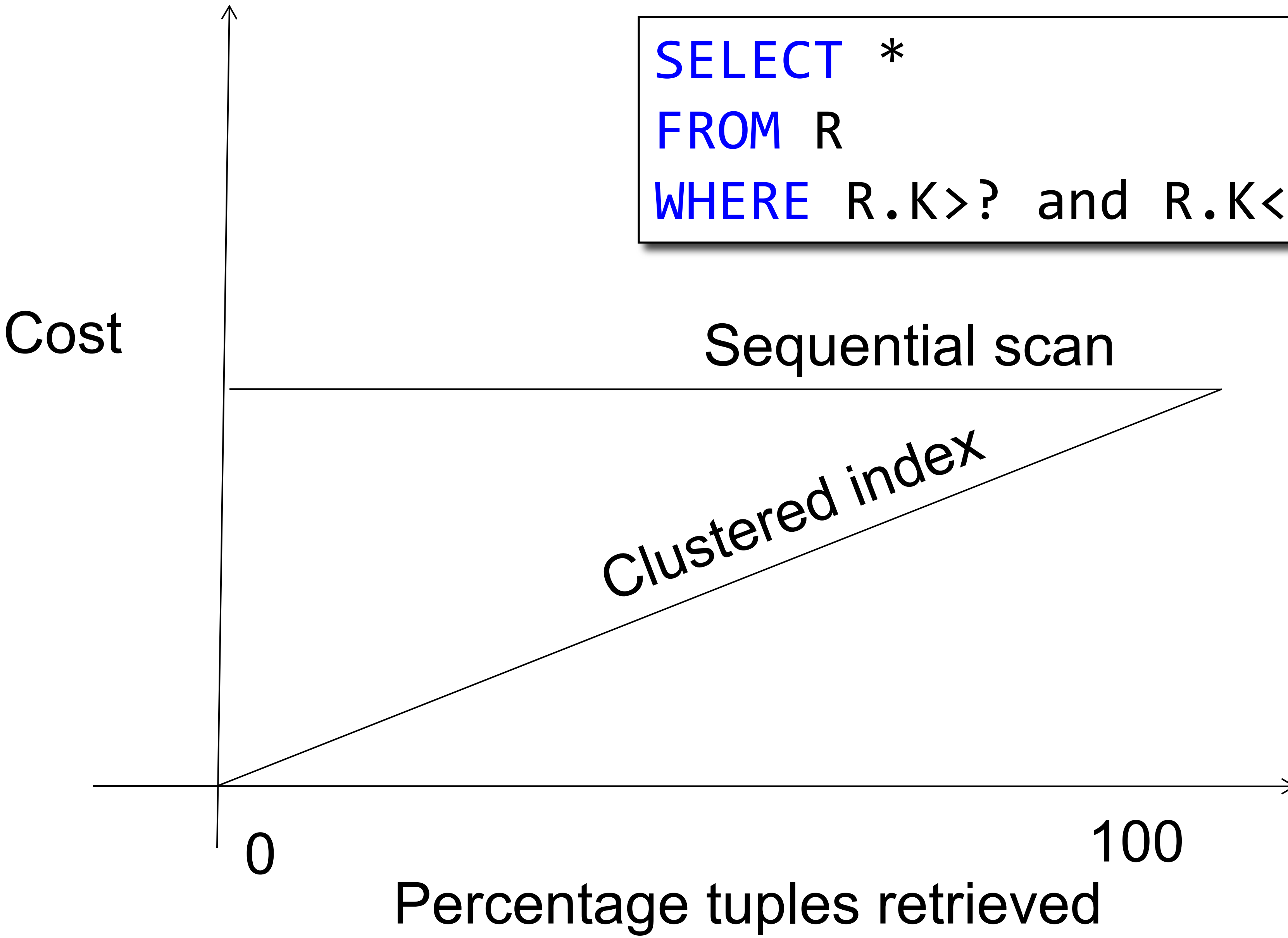
```sql
SELECT *
FROM R
WHERE R.K>? and R.K<?
```



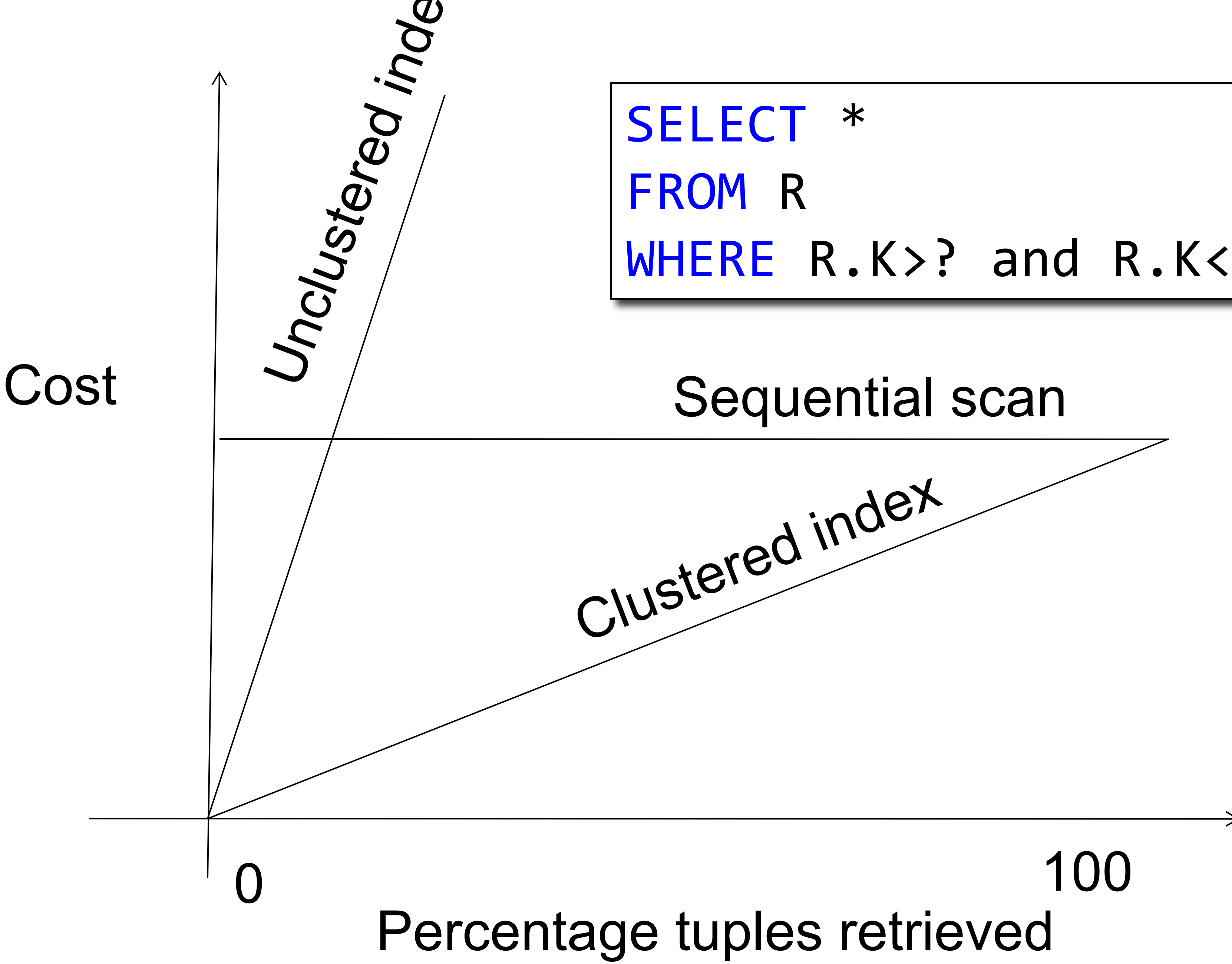
```
SELECT *
FROM R
WHERE R.K>? and R.K<?
```



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



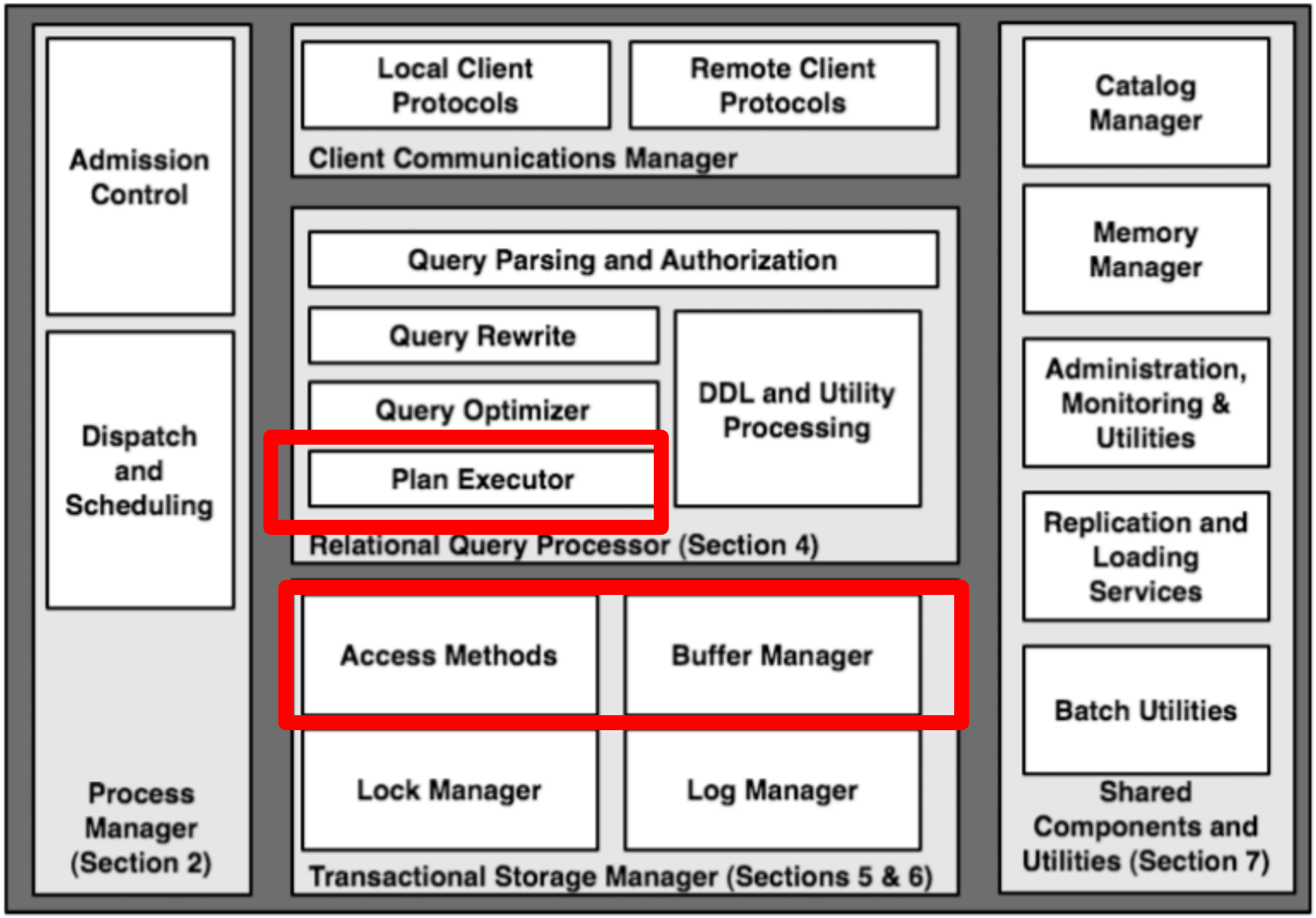
```
SELECT *
FROM R
WHERE R.K>? and R.K<?
```



# 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 =

# 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 =

```
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 = 10,000,000 = 80GB | why so little? |

# 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:

# 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:

# 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)

# 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:

# 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:

# 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:

# 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

# 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 !

# External Memory Joins

Recall standard main memory algorithms:

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

Review in class

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

# One Pass Hash Join

Hash join: $R \bowtie S$

- Scan R, build buckets in main memory
- Then scan S, probe hash table to join

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

- One pass algorithm when $B(R) <= M$

# Nested Loop Joins

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

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

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

# 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
            if r and s join then output (r,s)
```

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

# Nested Loop Joins

- We can be much more clever

- How would you compute the join in the following cases ? What is the cost ?

  – $B(R) = 1000, B(S) = 2, M = 4$

  – $B(R) = 1000, B(S) = 3, M = 4$

  – $B(R) = 1000, B(S) = 6, M = 4$

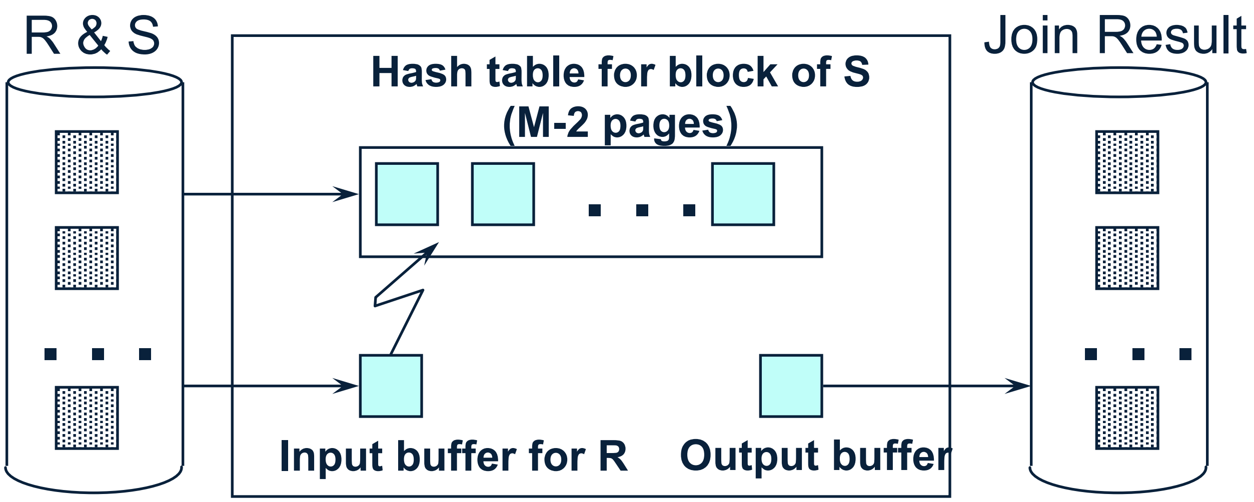
# Nested Loop Joins

- 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

# Nested Loop Joins



# 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

$R \bowtie S$
or
$S \bowtie R$
?

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

$R \bowtie S$
or
$S \bowtie R$
?

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!

# 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:

# 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)

# Sort-Merge Join

Sort-merge join: R ⋈ S

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

- Cost: B(R) + B(S)
- One pass algorithm when B(S) + B(R) <= M

# Grouping

$\gamma_{\text{department, sum(quantity)}}$ (Product)

In class: describe a one-pass algorithms.

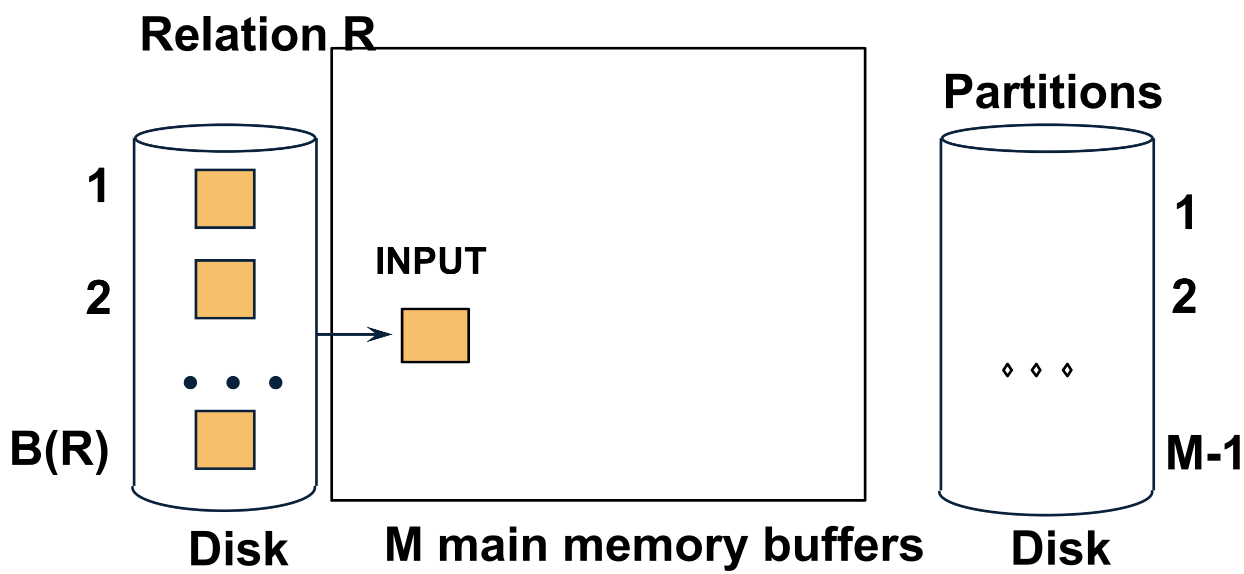
Cost=?

# Two-Pass Algorithms

- When data is larger than main memory, need two or more passes
- Two key techniques
  - Hashing
  - Sorting

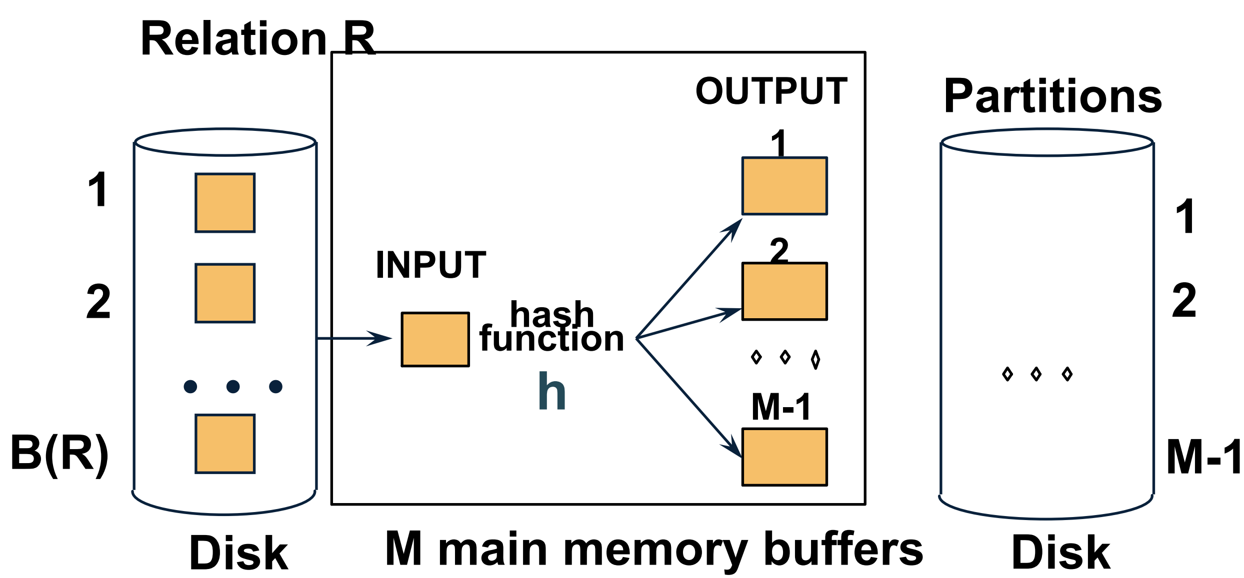
# Two Pass Algorithms Based on Hashing

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



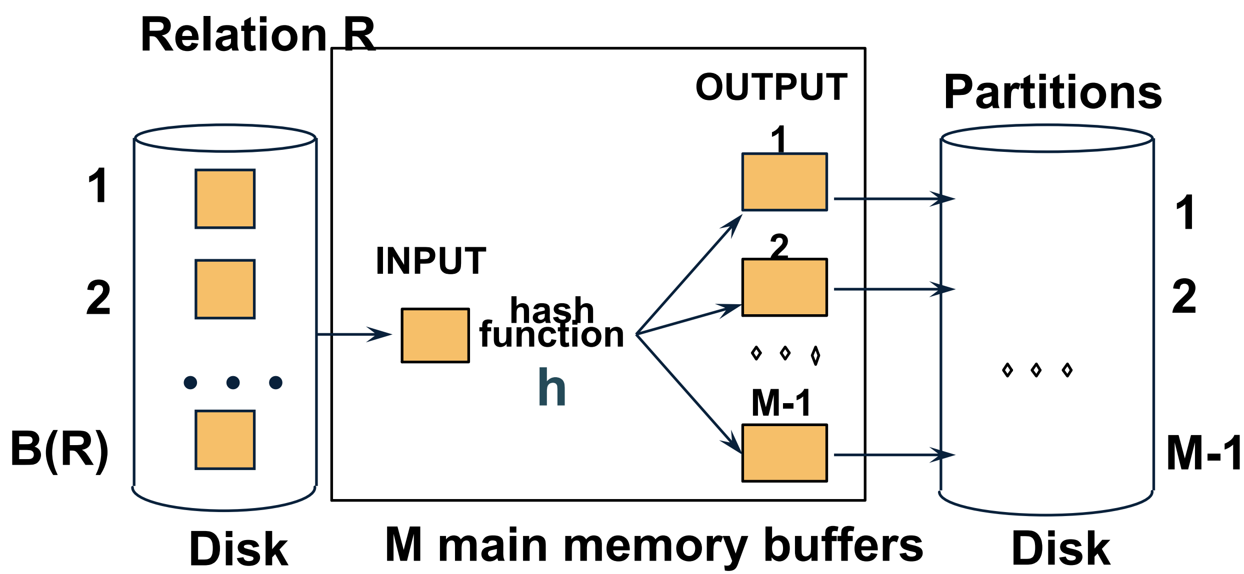
# Two Pass Algorithms Based on Hashing

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



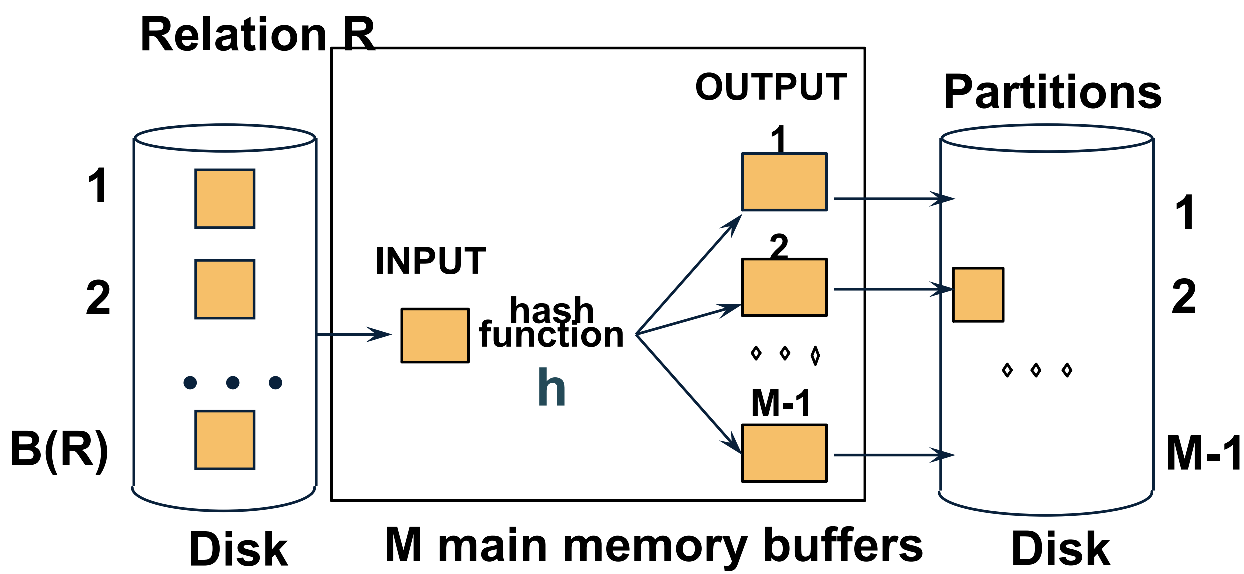
# Two Pass Algorithms Based on Hashing

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



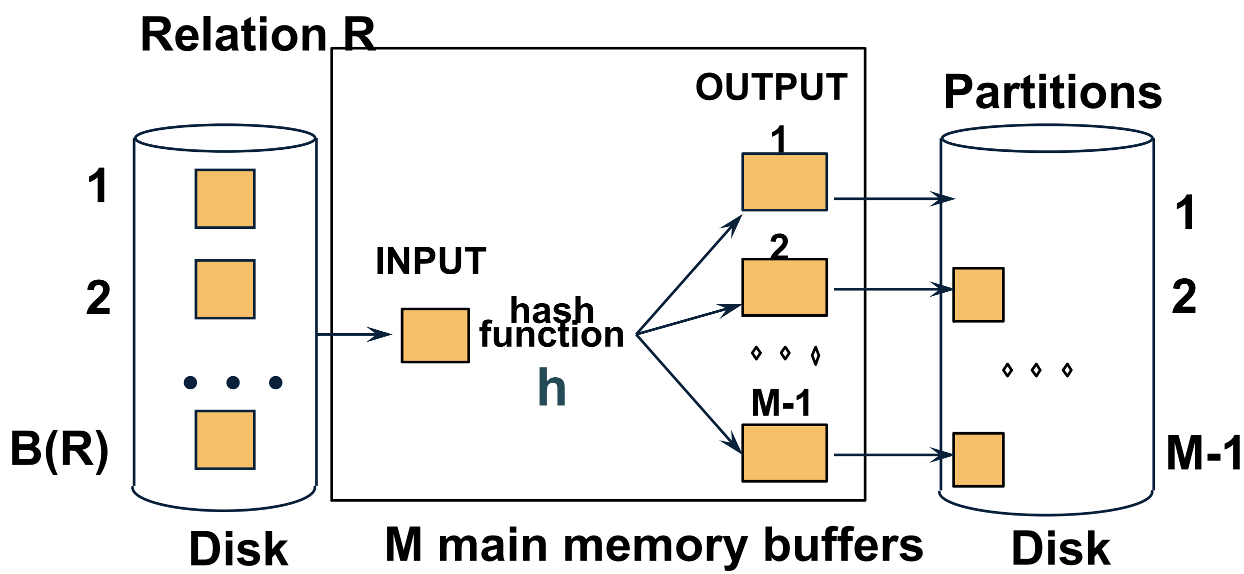
# Two Pass Algorithms Based on Hashing

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



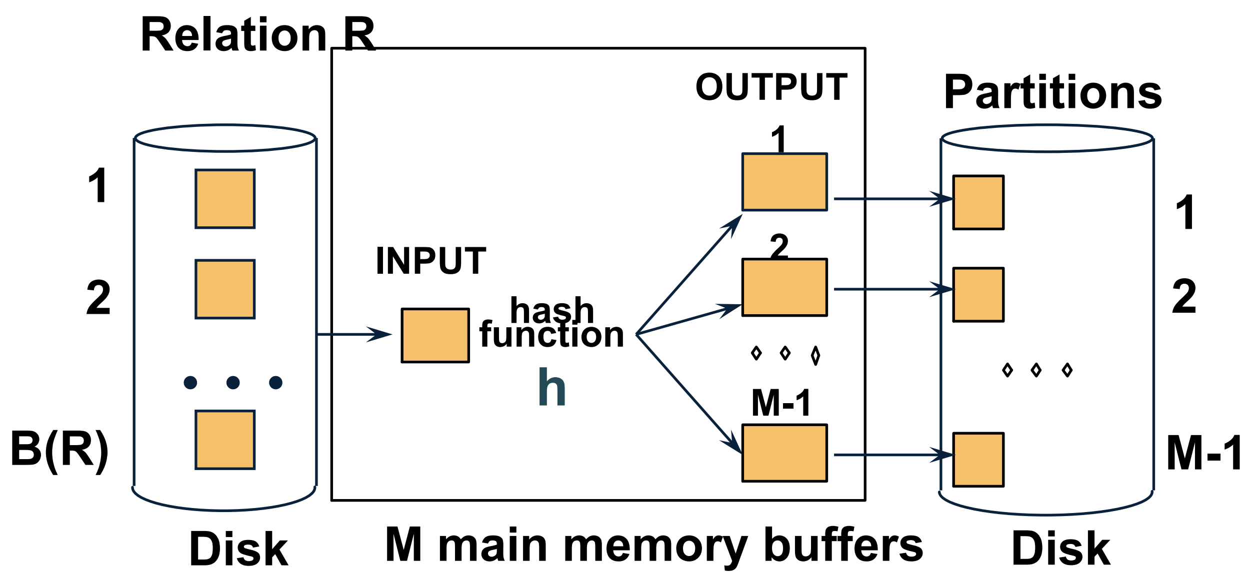
# Two Pass Algorithms Based on Hashing

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



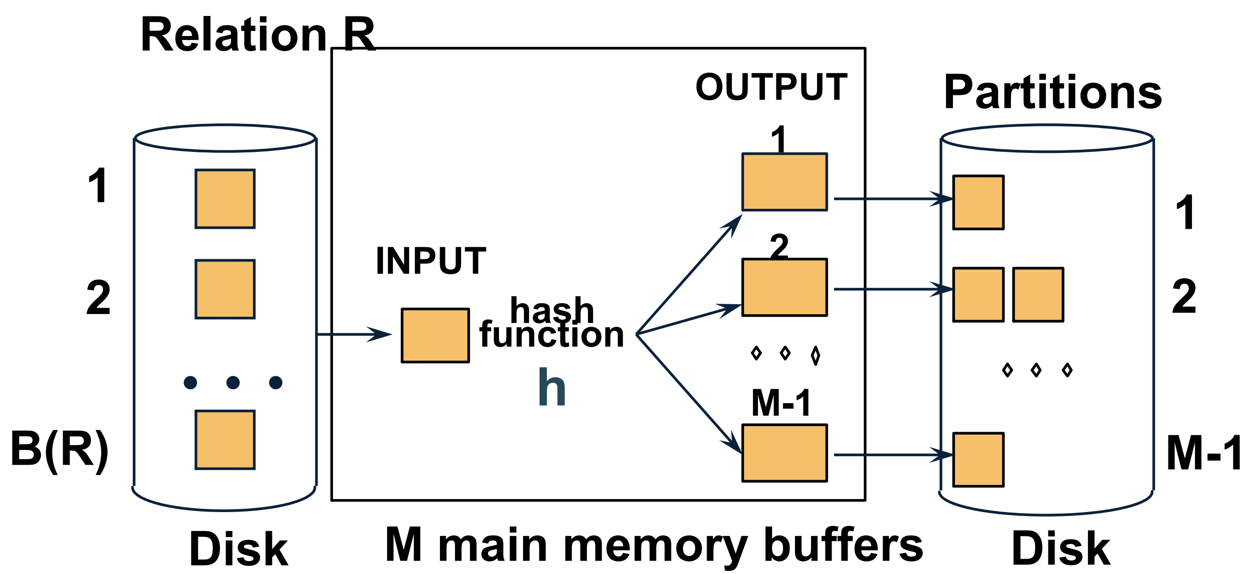
# Two Pass Algorithms Based on Hashing

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



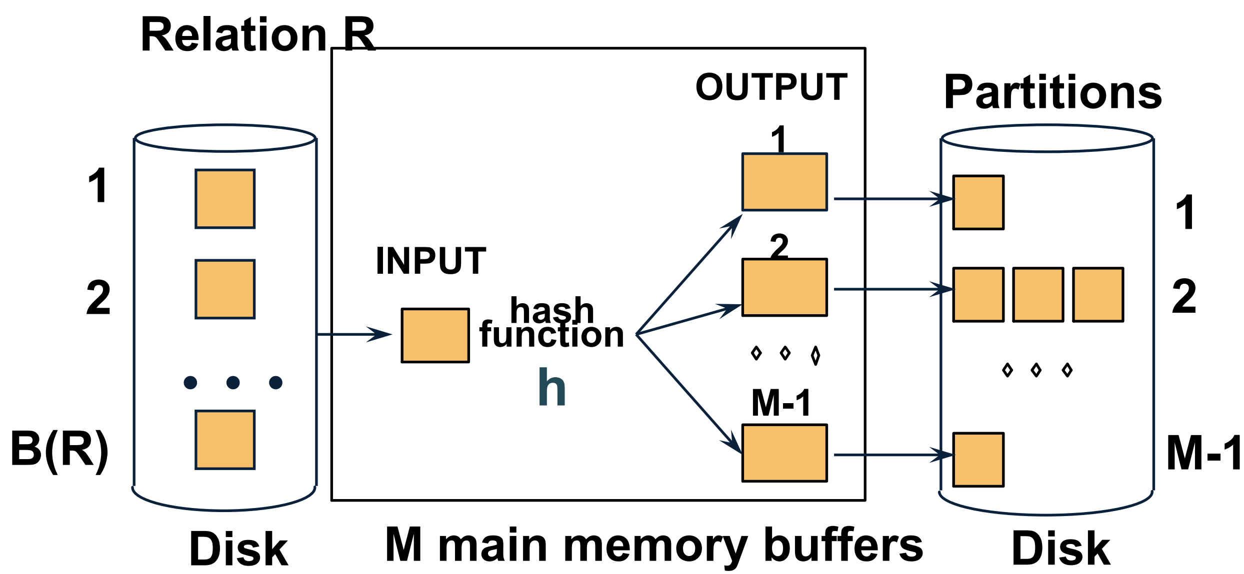
# Two Pass Algorithms Based on Hashing

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



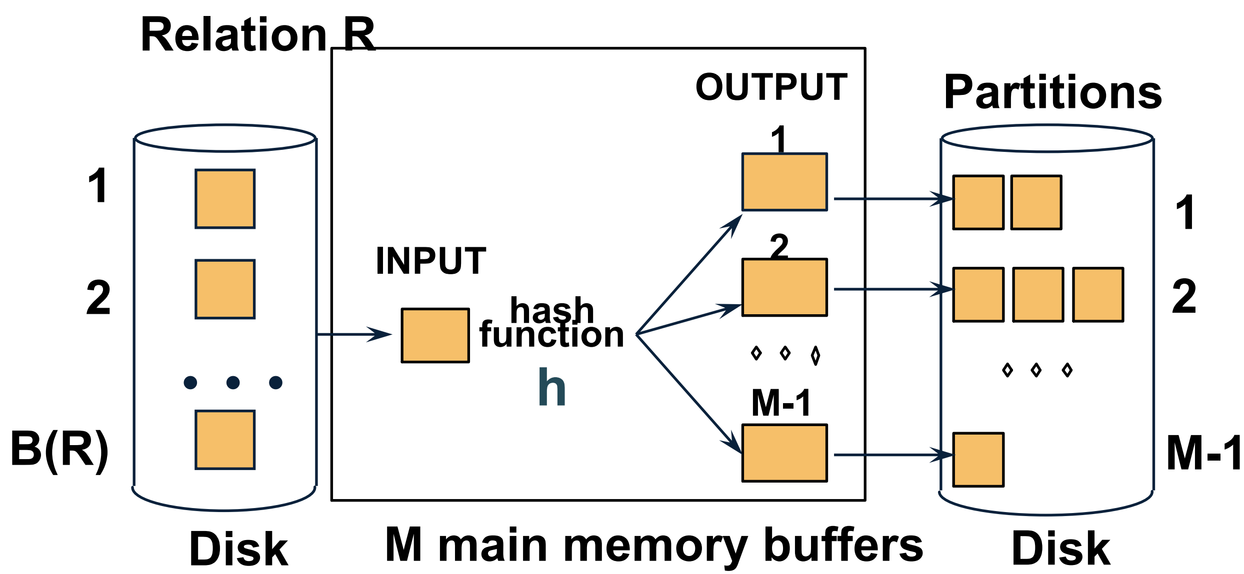
# Two Pass Algorithms Based on Hashing

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



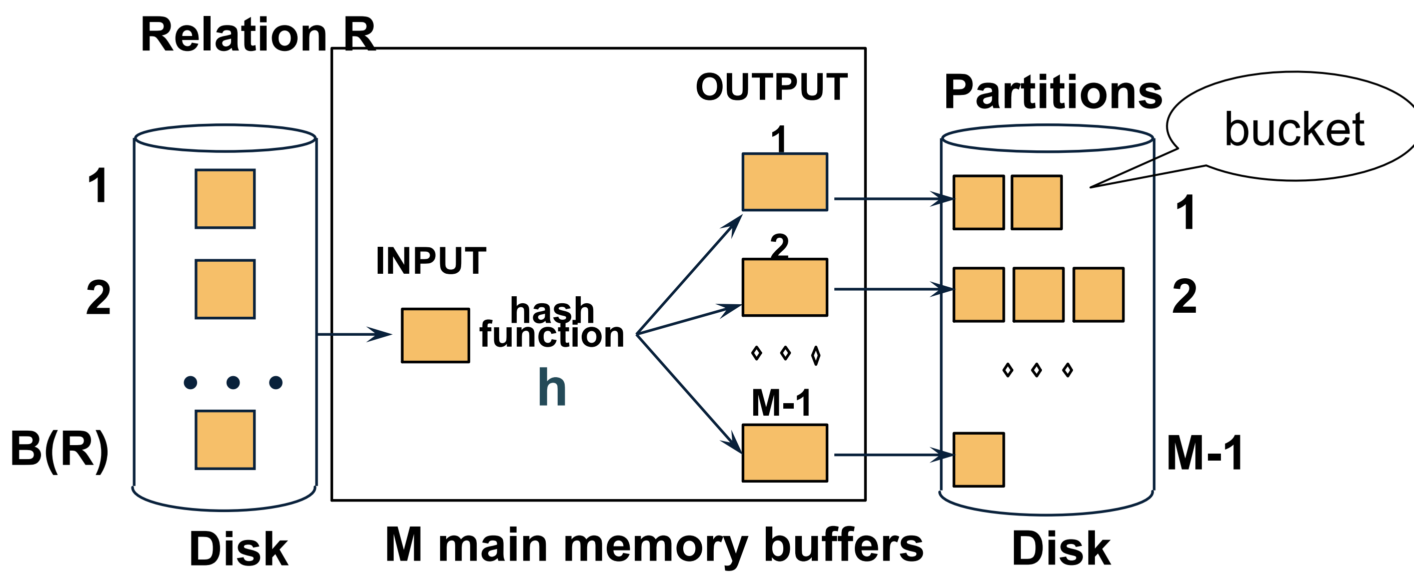
# Two Pass Algorithms Based on Hashing

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



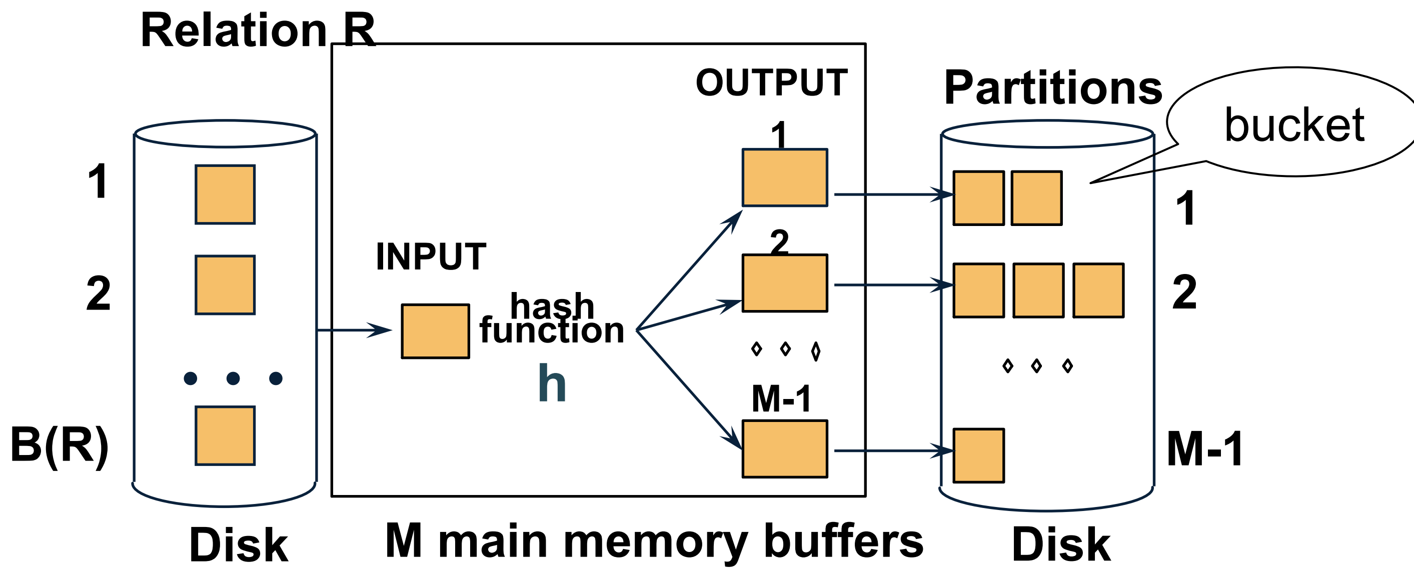
# Two Pass Algorithms Based on Hashing

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



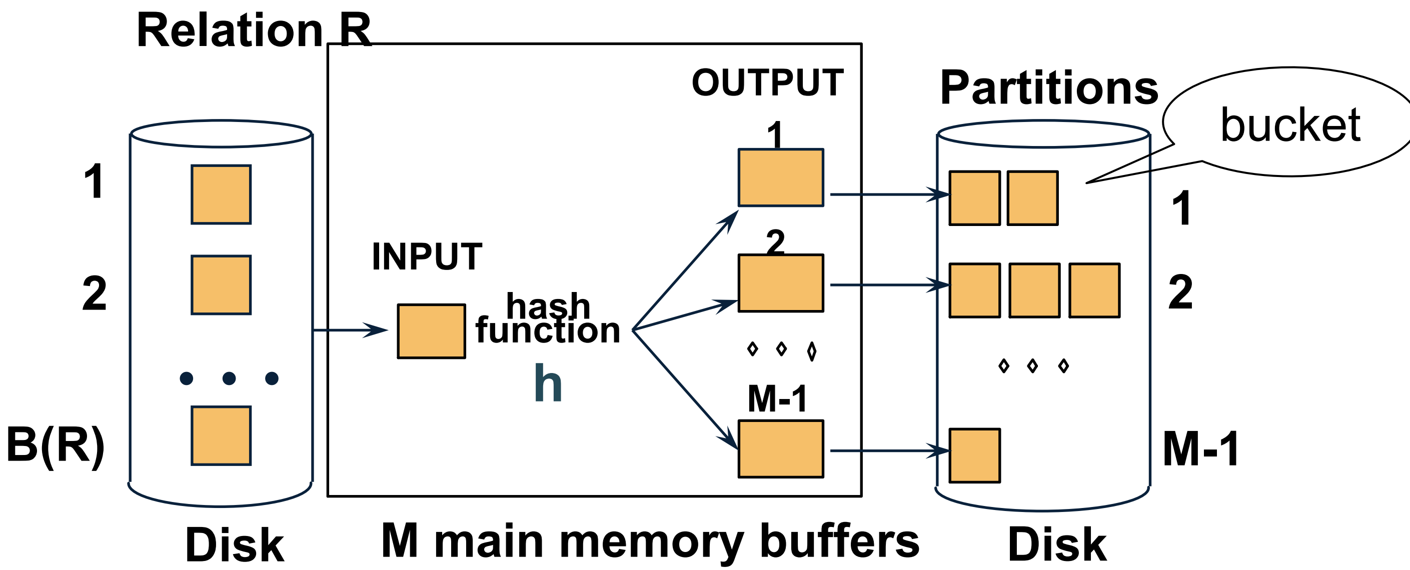
# Two Pass Algorithms Based on Hashing

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



# Two Pass Algorithms Based on Hashing

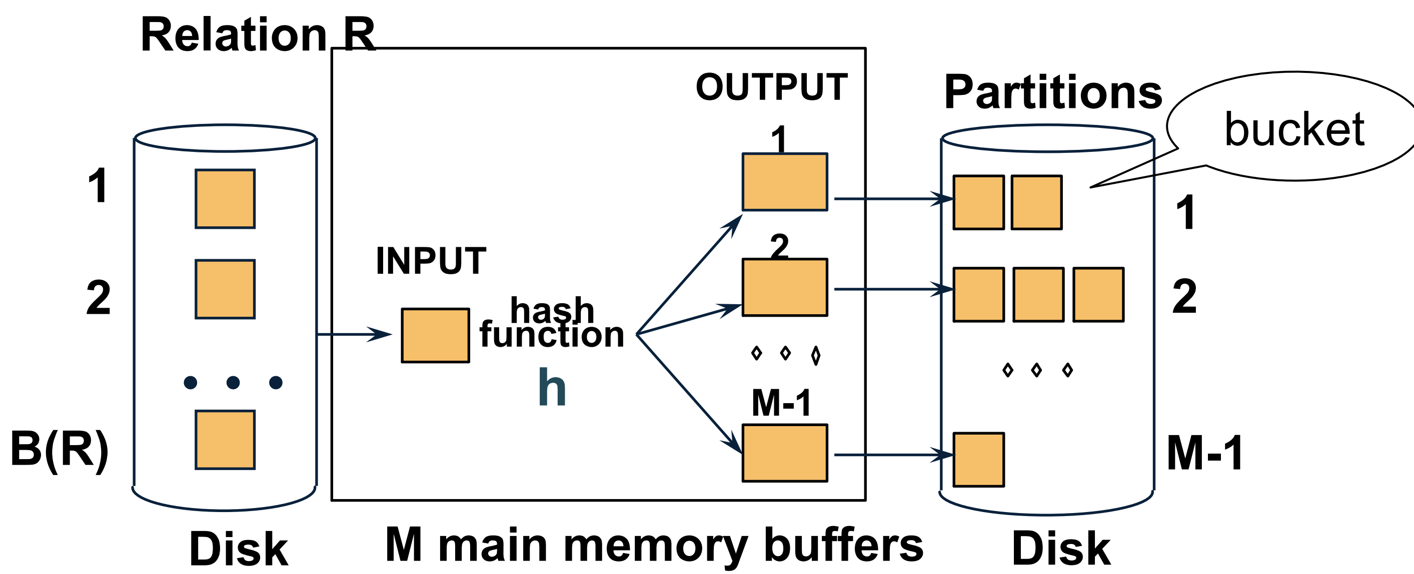
- Idea: partition a relation R into buckets, on disk
- Each bucket has size approx. B(R)/M




- Does each bucket fit in main memory ?

# Two Pass Algorithms Based on Hashing

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




- Does each bucket fit in main memory ?
- Yes when: $B(R)/M \leq M$, i.e. $B(R) \leq M^2$

# Hash Based Algorithms for $\gamma$

- Recall: $\gamma(R)$ = grouping and aggregation

- Step 1. Partition R into buckets
- Step 2. Apply $\gamma$ to each bucket

- Cost: $3B(R)$
- Assumption: $B(R) \leq M^2$

# Partitioned (Grace) Hash Join

$R \bowtie S$

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

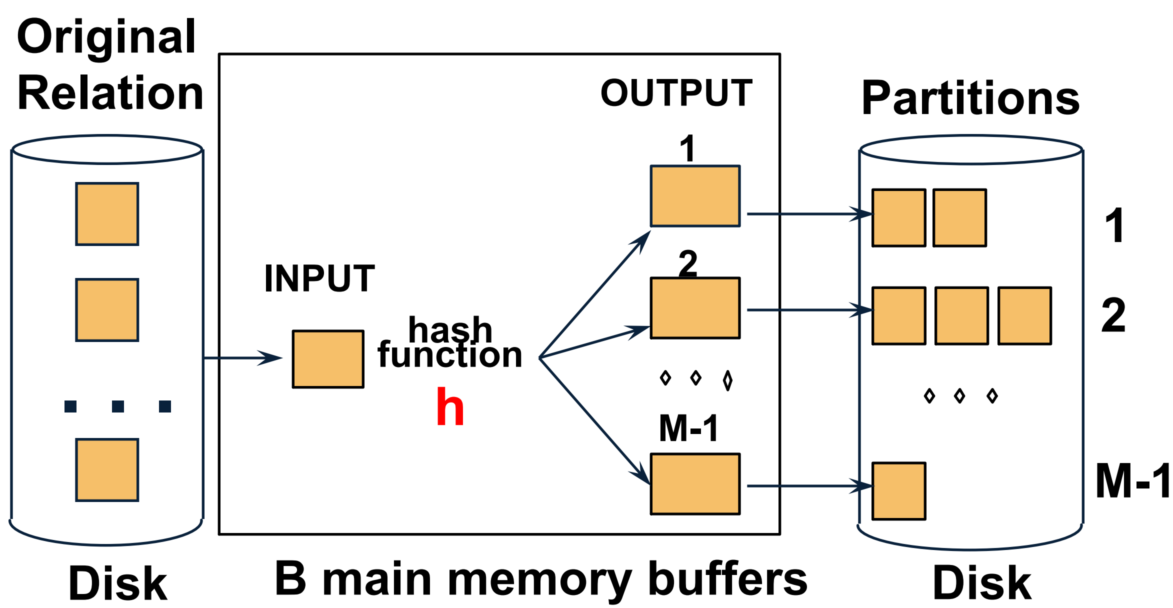
- Step 2
  - Hash R into M-1 buckets
  - Send all buckets to disk

- Step 3
  - Join every pair of buckets

# Partitioned Hash Join R

R ⋈ S

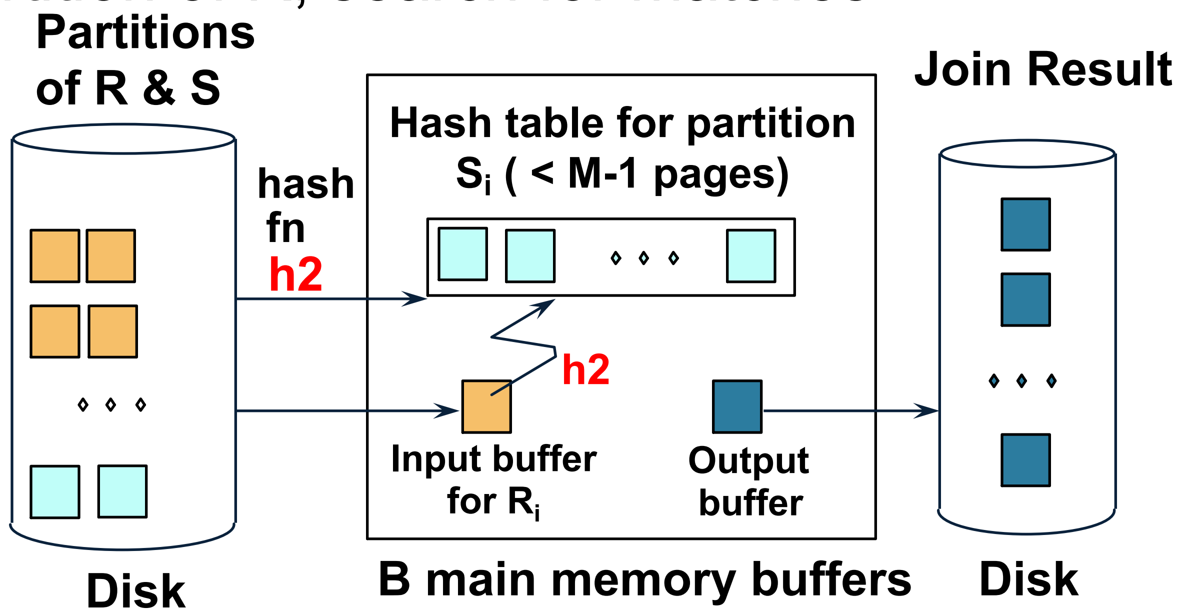
- Partition both relations using hash fn **h**



# Partitioned Hash Join

R ⋈ S

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



# Partitioned Hash Join

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

# 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: $k\text{-}t \leq M$
- Thus: $t/k * B(S) + k\text{-}t \leq M$

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

# 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

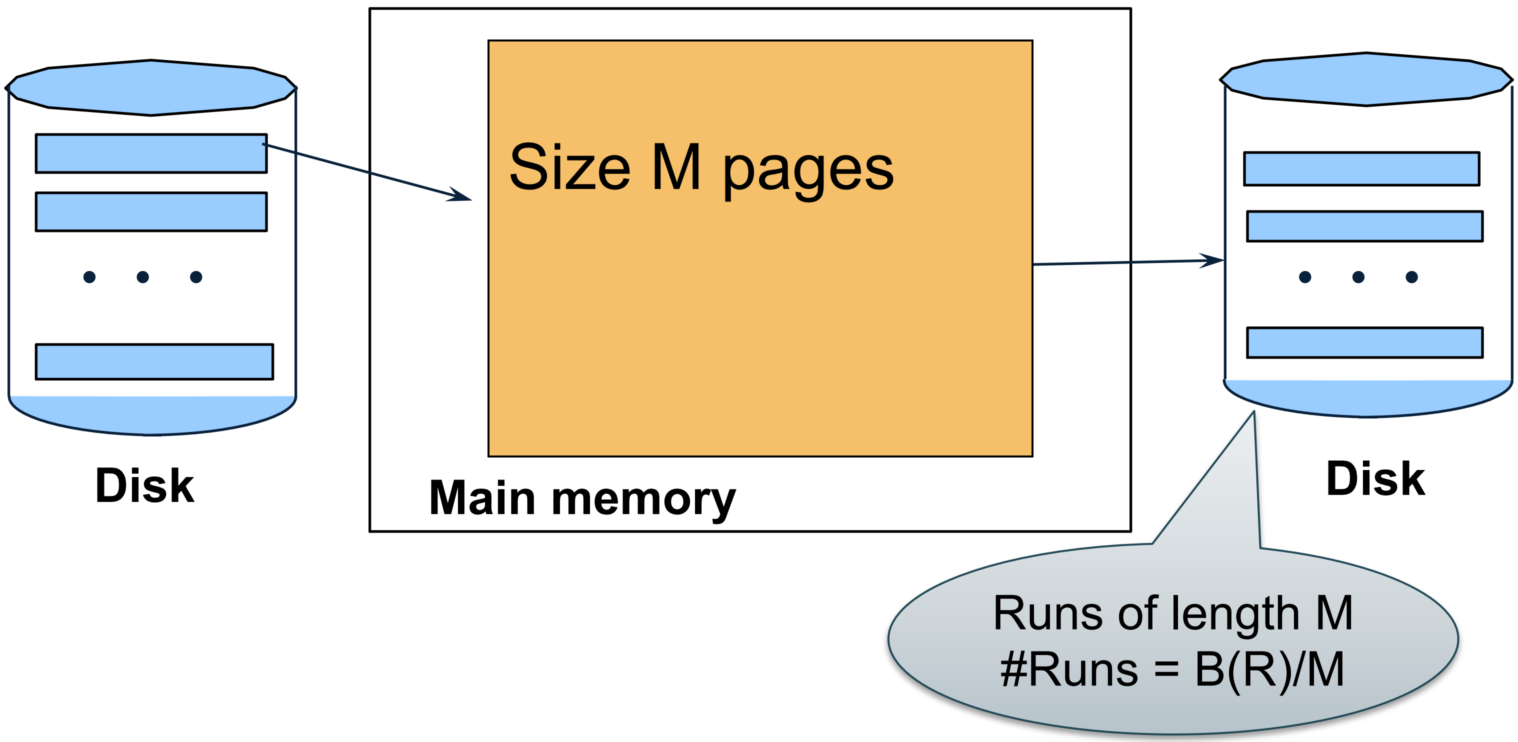
Cost: (3-2t/k)(B(R) + B(S)) = (3-2M/B(S))(B(R) + B(S))

# 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 $B \leq M^2$

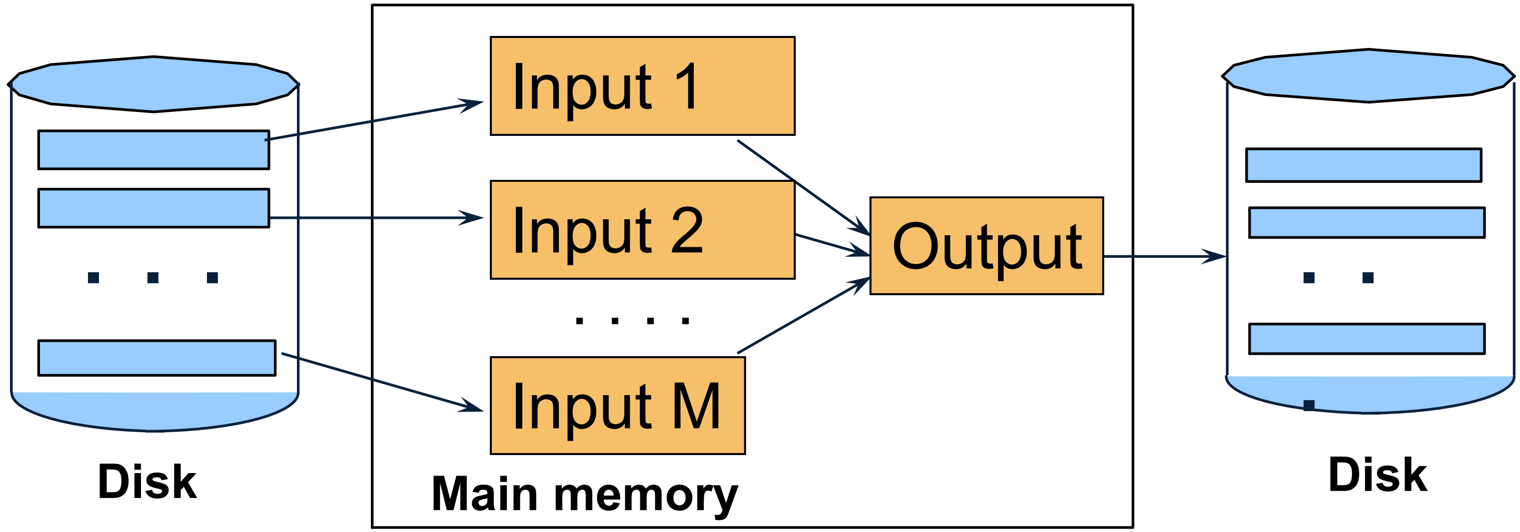
# External Merge-Sort: Step 1

- 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) ≈ $M^2$




Assuming B ≤ $M^2$, 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
- Hybrid Hash Join: (3-2M/B(S))(B(R) + B(S))
  Assuming t/k * B(S) >> k-t
- Sort-Merge Join: 3B(R)+3B(S)
  Assuming B(R)+B(S) ≤ $M^2$
- Index Nested Loop Join: B(R) + T(R)B(S)/V(S,a)
  Assuming R is clustered and S has clustered index on a