CSE 544
Principles of Database Management Systems

Lectures 7 and 8
DBMS Architecture and Query Execution

Announcements

- Project proposals: please sign up for a 15' meeting on Friday
  - You will present your proposal (5')
  - We discuss it (5')
  - Additional questions/comments (5')
- Homework 2 is due on Friday
- Homework 3 is posted

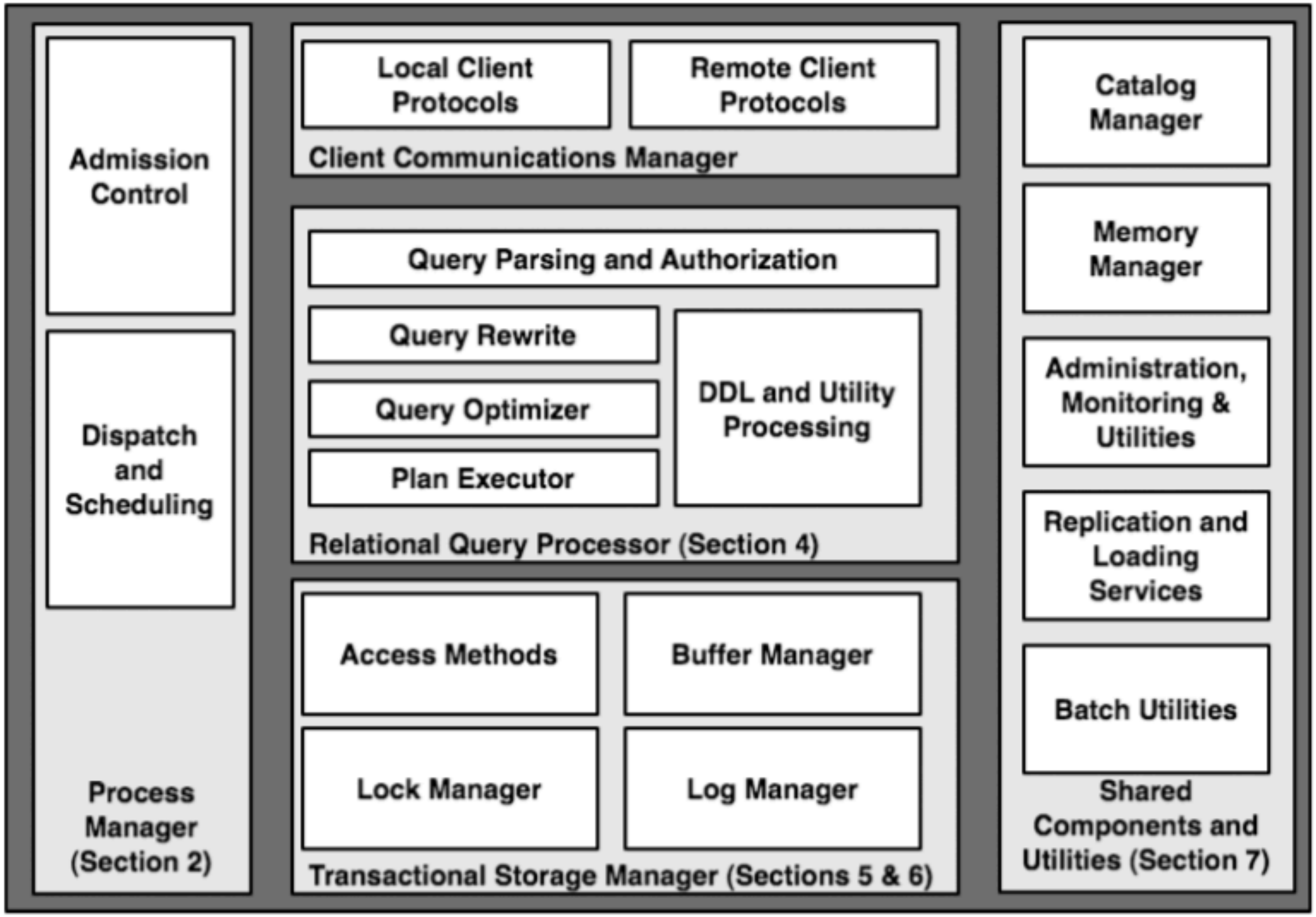
Outline

- Architecture of a DBMS
- Steps involved in processing a query
- Operator implementations

Architecture of DBMS

- Reading:
  Architecture of a DBMS, chap. 1 and 2

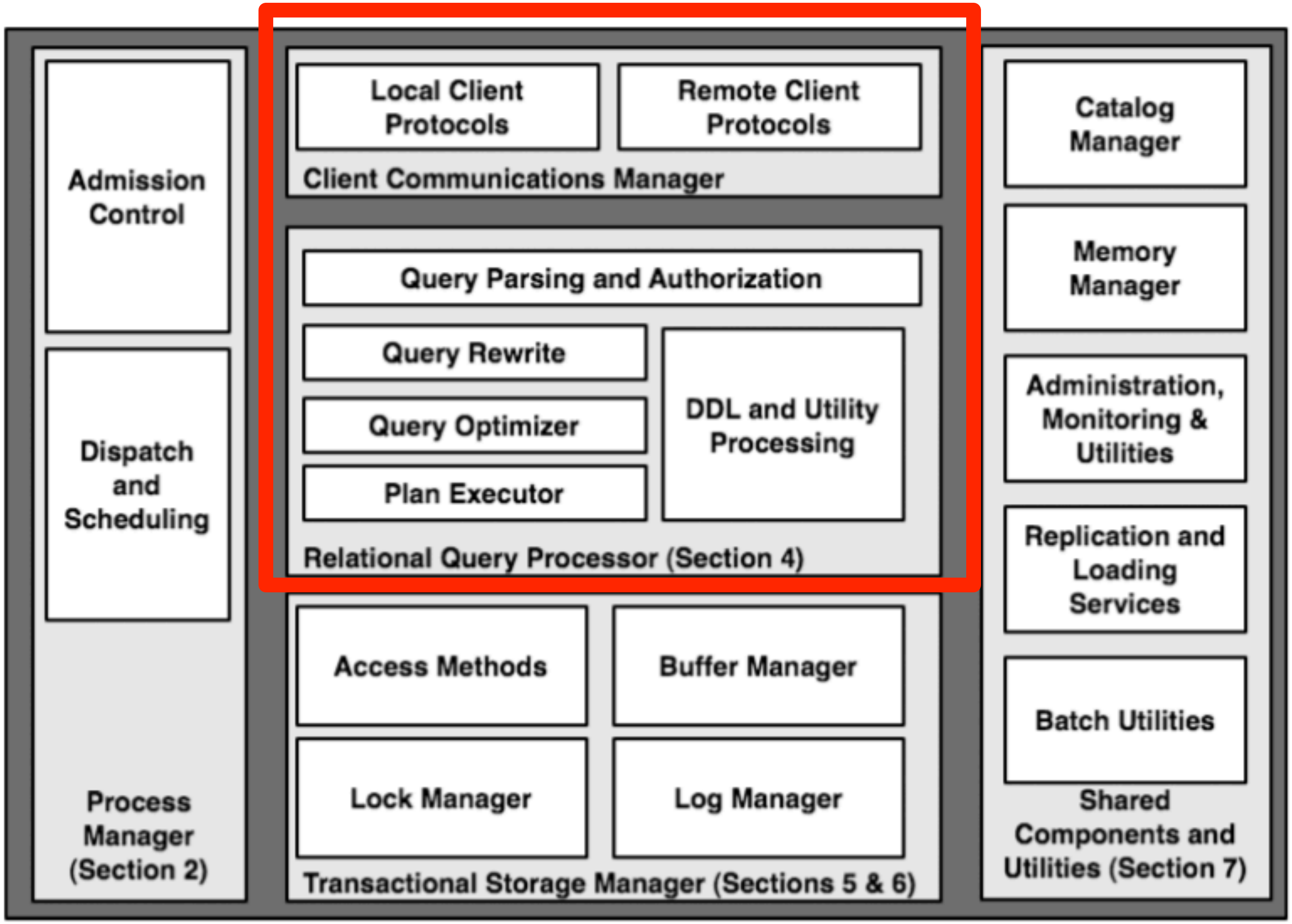
Architecture of DBMS



Why Multiple Processes

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

Multiple Processes

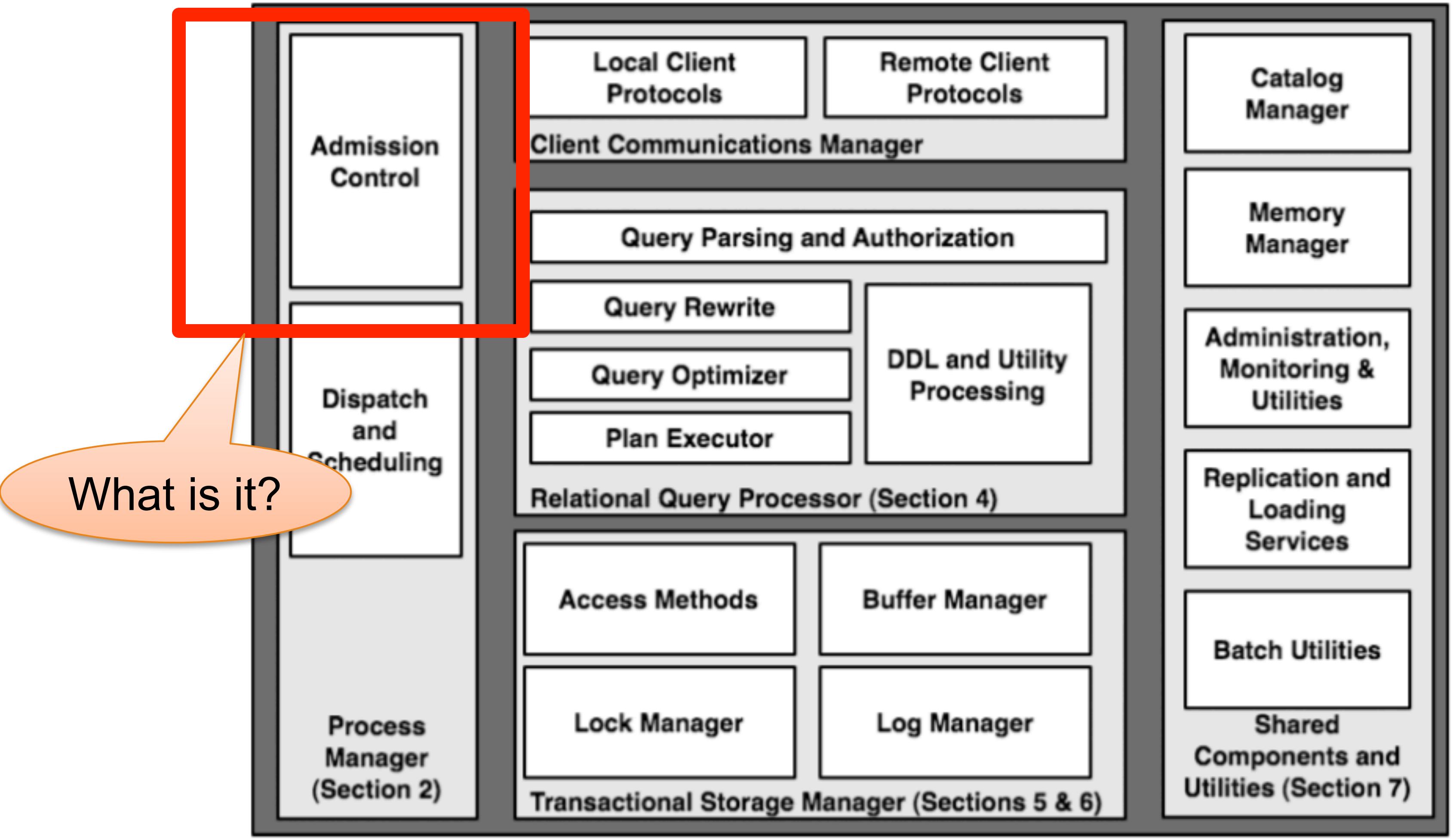


Process Models

Discuss pro/cons for each model

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

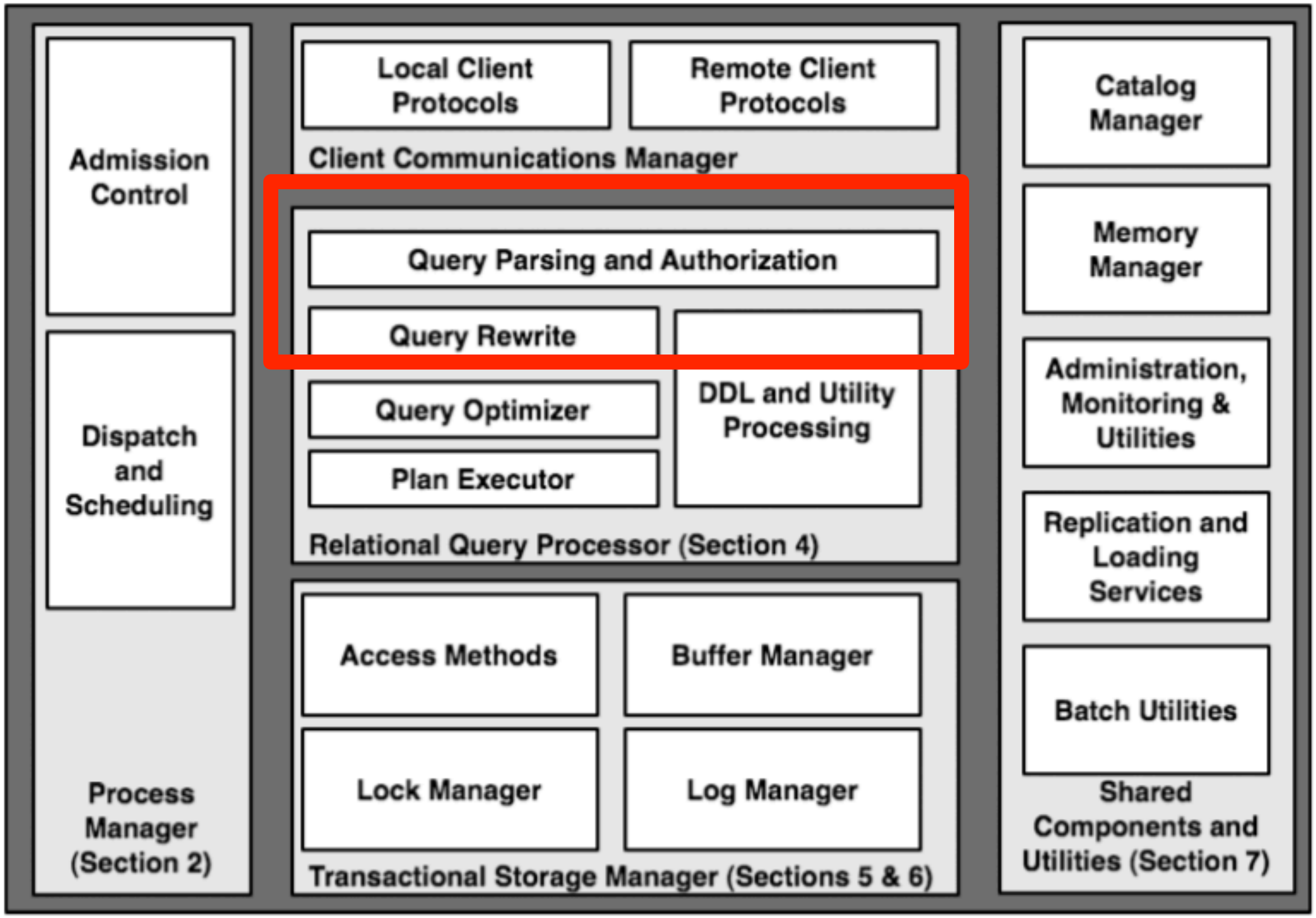
Admission Control



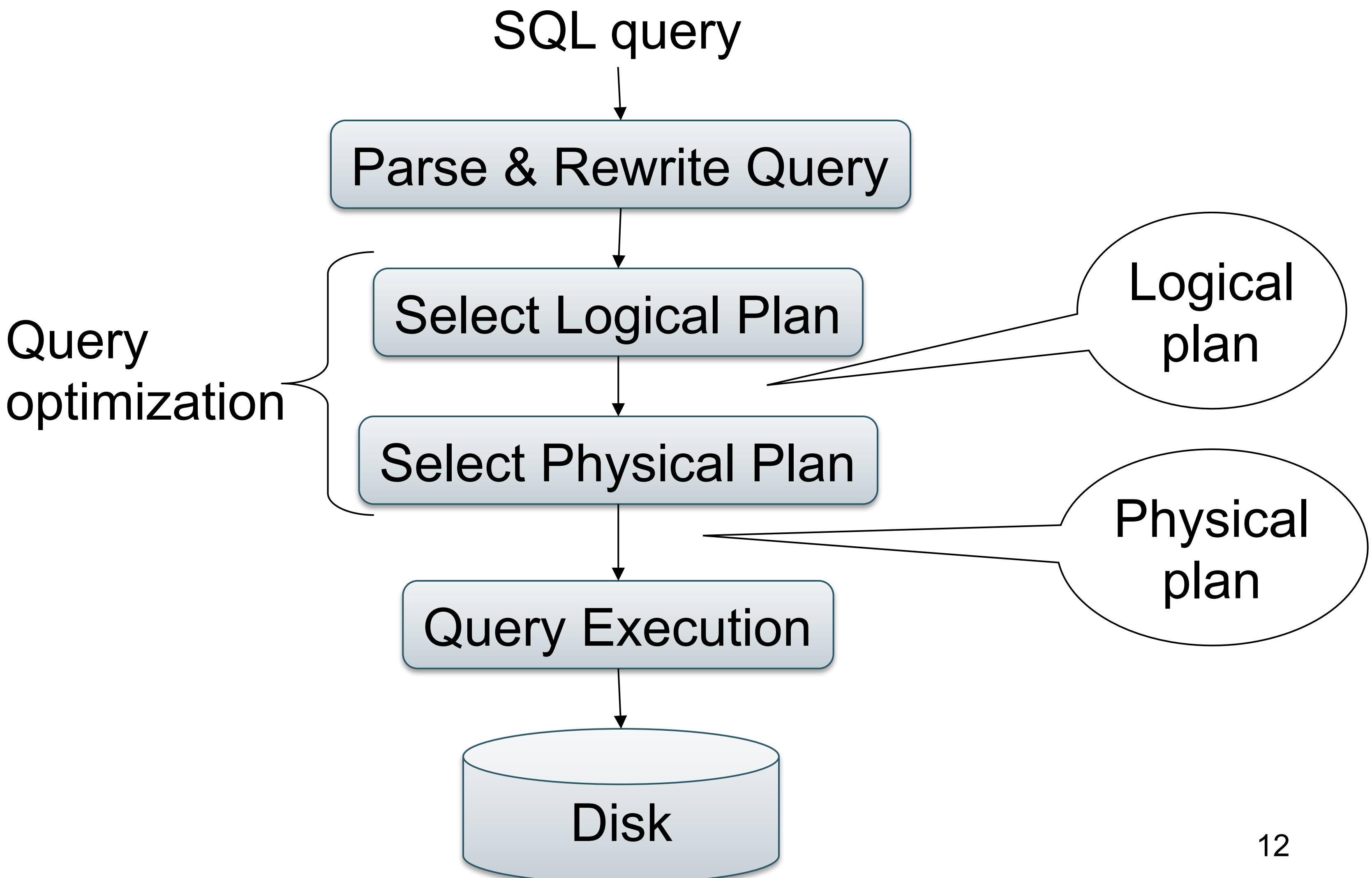
Outline

- Architecture of a DBMS
- Steps involved in processing a query
- Operator implementations

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

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)
```

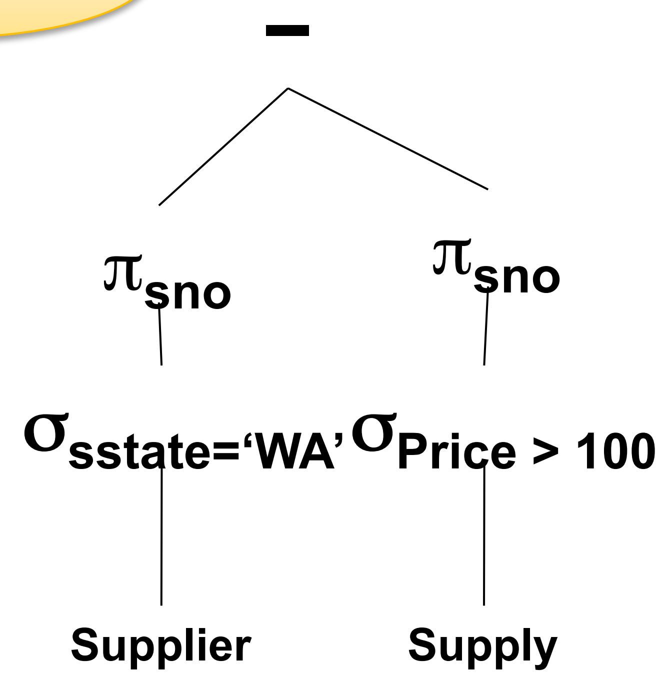
EXCEPT = set difference

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 next lecture on this topic

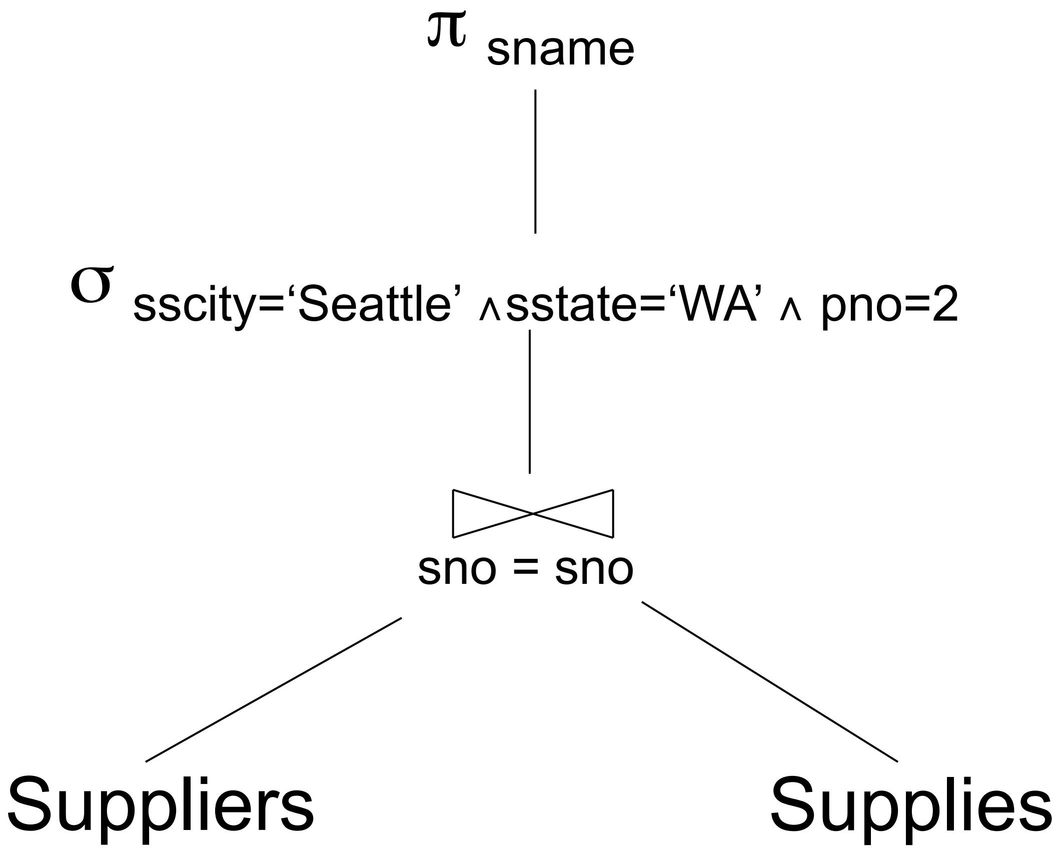
- A **query plan** is
  - **Logical query plan**: an extended relational algebra tree
  - **Physical query plan**: with additional annotations at each node

Extended Algebra Operators

- Union $\cup$, intersection $\cap$, difference -
- **S**election σ
- **P**rojection π
- Join $\bowtie$
- **D**uplicate elimination δ
- **G**rouping and aggregation γ
- **S**orting τ
- **R**ename ρ

Bag semantics!

Logical Query Plan

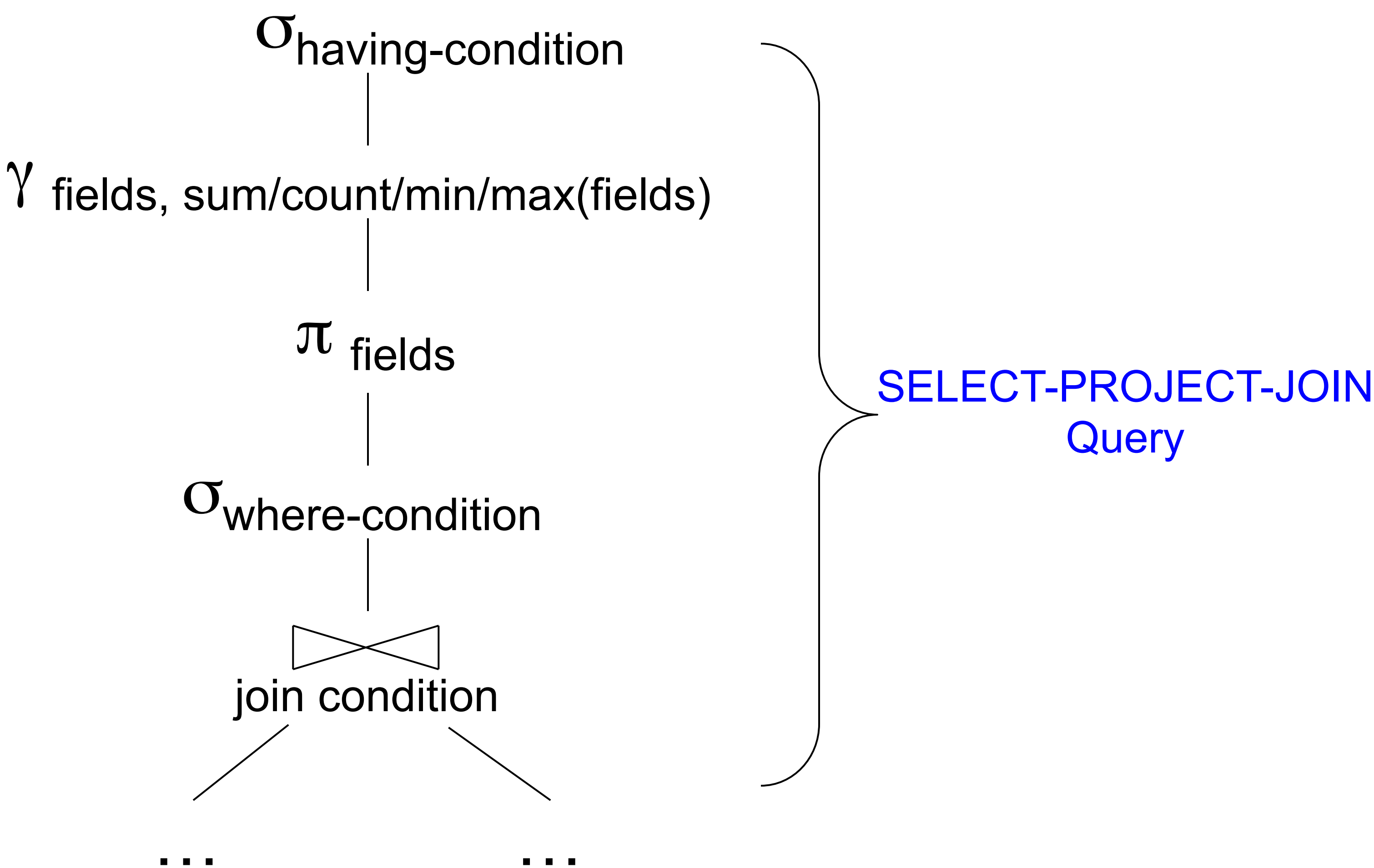


Query Block

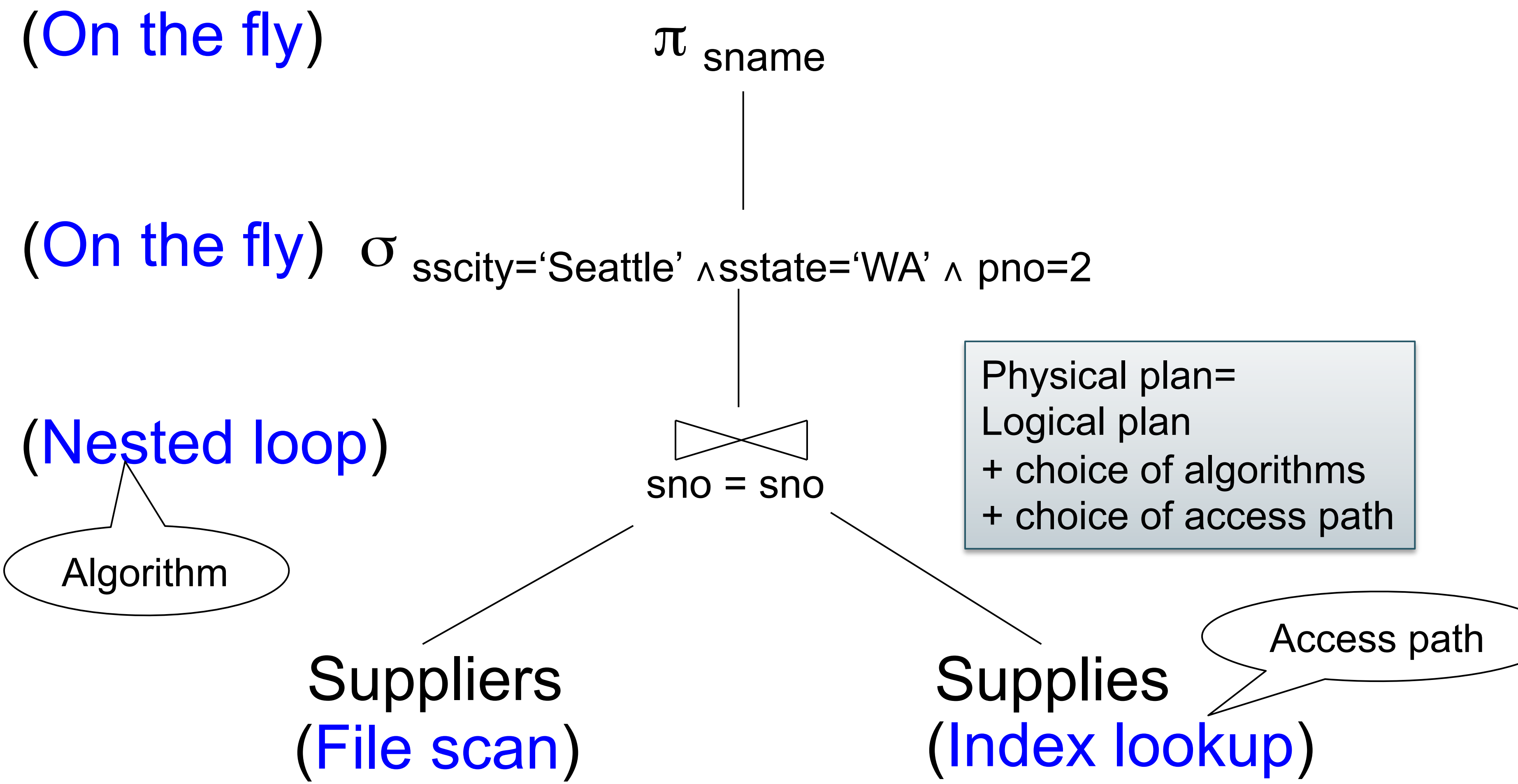
- Most optimizers operate on individual query blocks

- 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

Typical Plan For Block



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

Implementing Query Operators with the Iterator Interface

Each operator implements three methods:

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

Implementing Query Operators with the Iterator Interface

Example "on the fly" selection operator

```
interface Operator {

}
```

Implementing Query Operators with the Iterator Interface

Example “on the fly” selection operator

```
interface Operator {

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

}
```

Implementing Query Operators with the Iterator Interface

Example "on the fly" selection operator

```
interface Operator {

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


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



}
```

Implementing Query Operators with the Iterator Interface

Example "on the fly" selection operator

```
interface Operator {

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


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


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

Implementing Query Operators with the Iterator Interface

Example "on the fly" selection operator

```
interface Operator {

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


  // 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
                child) {this.p = p;
      this.child=child; child.open();
  }
}
```

Implementing Query Operators with the Iterator Interface

Example "on the fly" selection operator

```
interface Operator {

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


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


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

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




  }

}
```

Implementing Query Operators with the Iterator Interface

```
interface Operator {

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


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


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

Example "on the fly" selection operator

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

Implementing Query Operators with the Iterator Interface

```
interface Operator {

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


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


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

Query plan execution

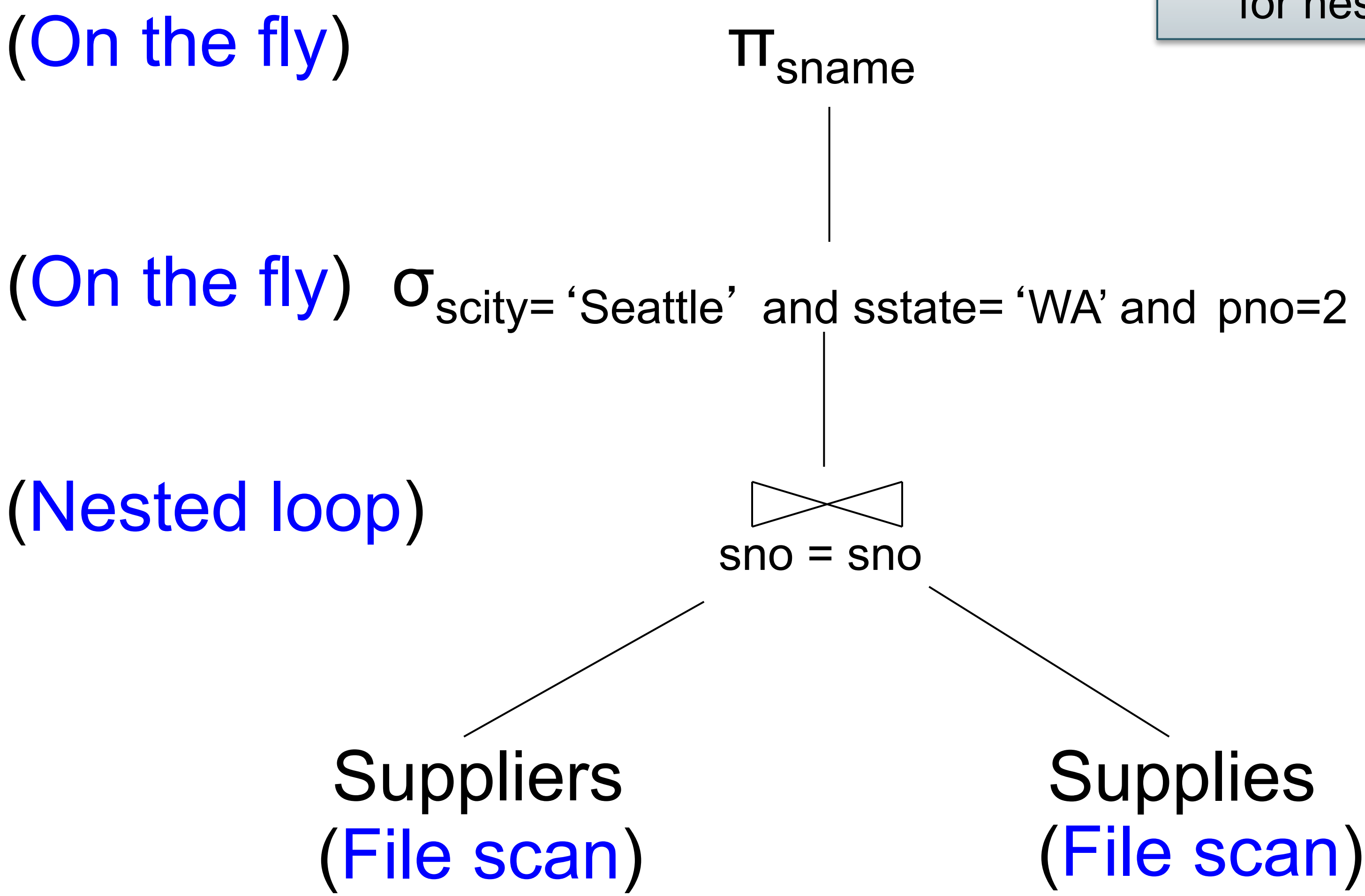
```
Operator q = parse(“SELECT ...”);
q = optimize(q);

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

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

Pipelining

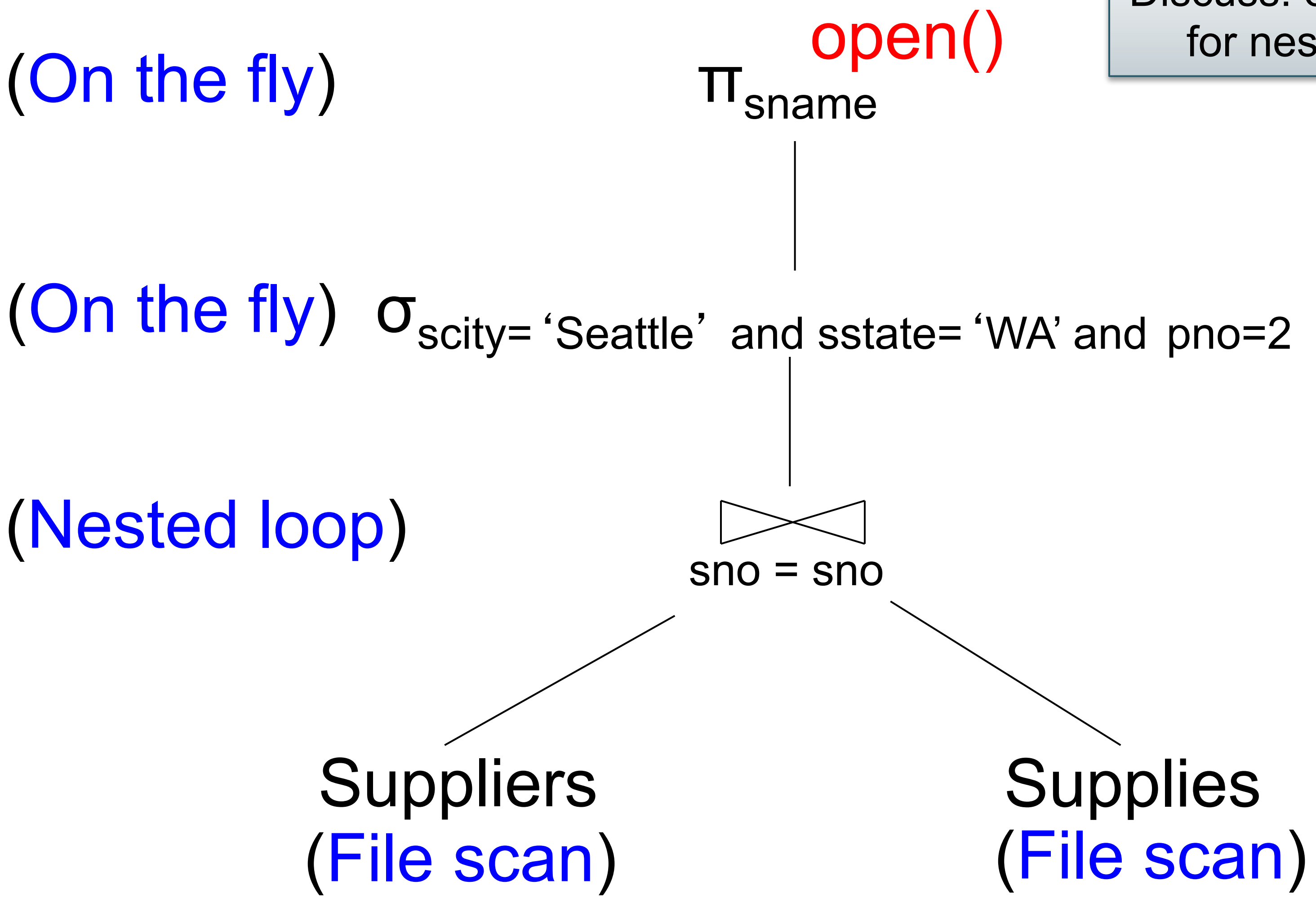
Discuss: open/next/close for nested loop join



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

Pipelining

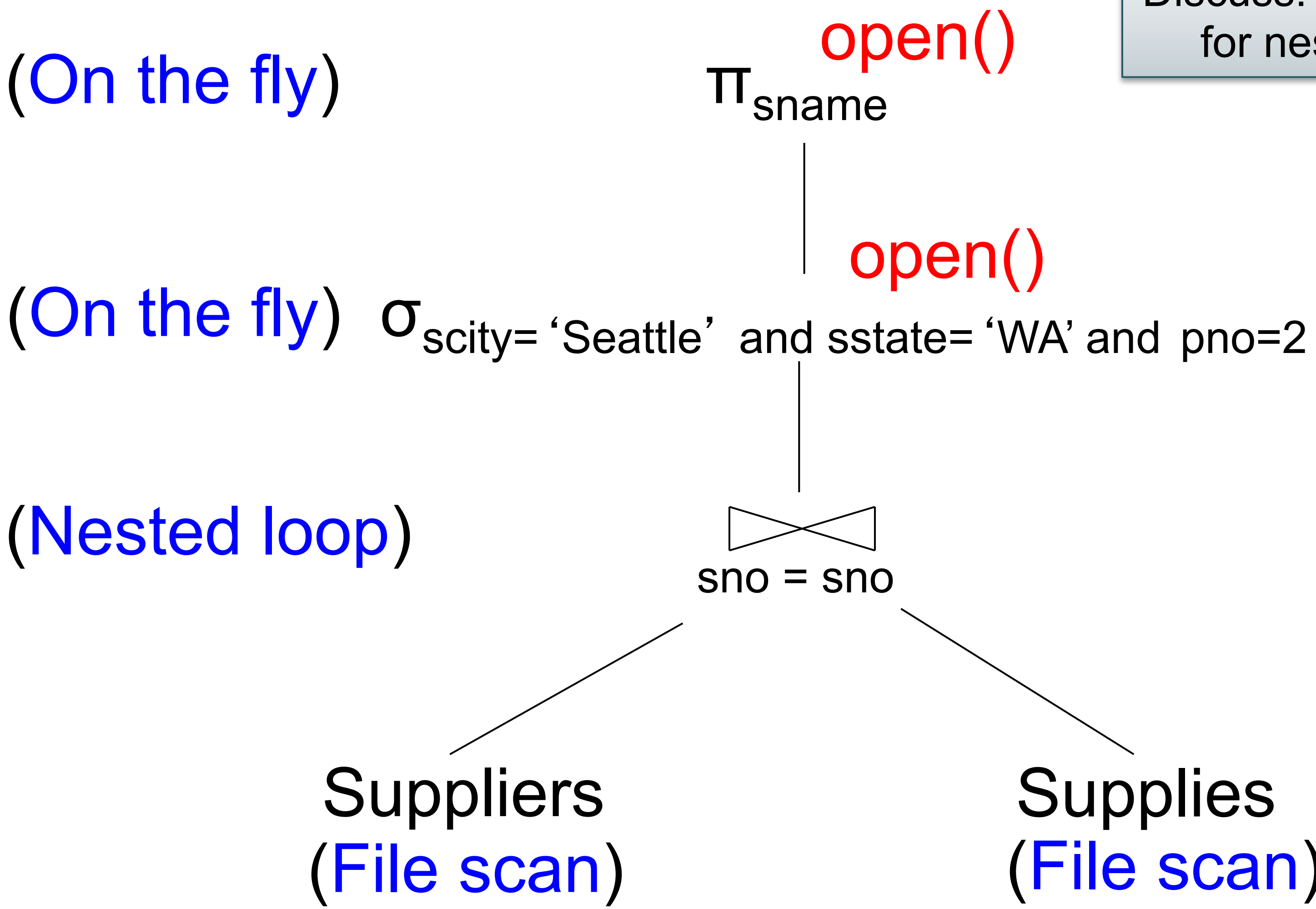
Discuss: open/next/close for nested loop join



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

Pipelining

Discuss: open/next/close for nested loop join



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

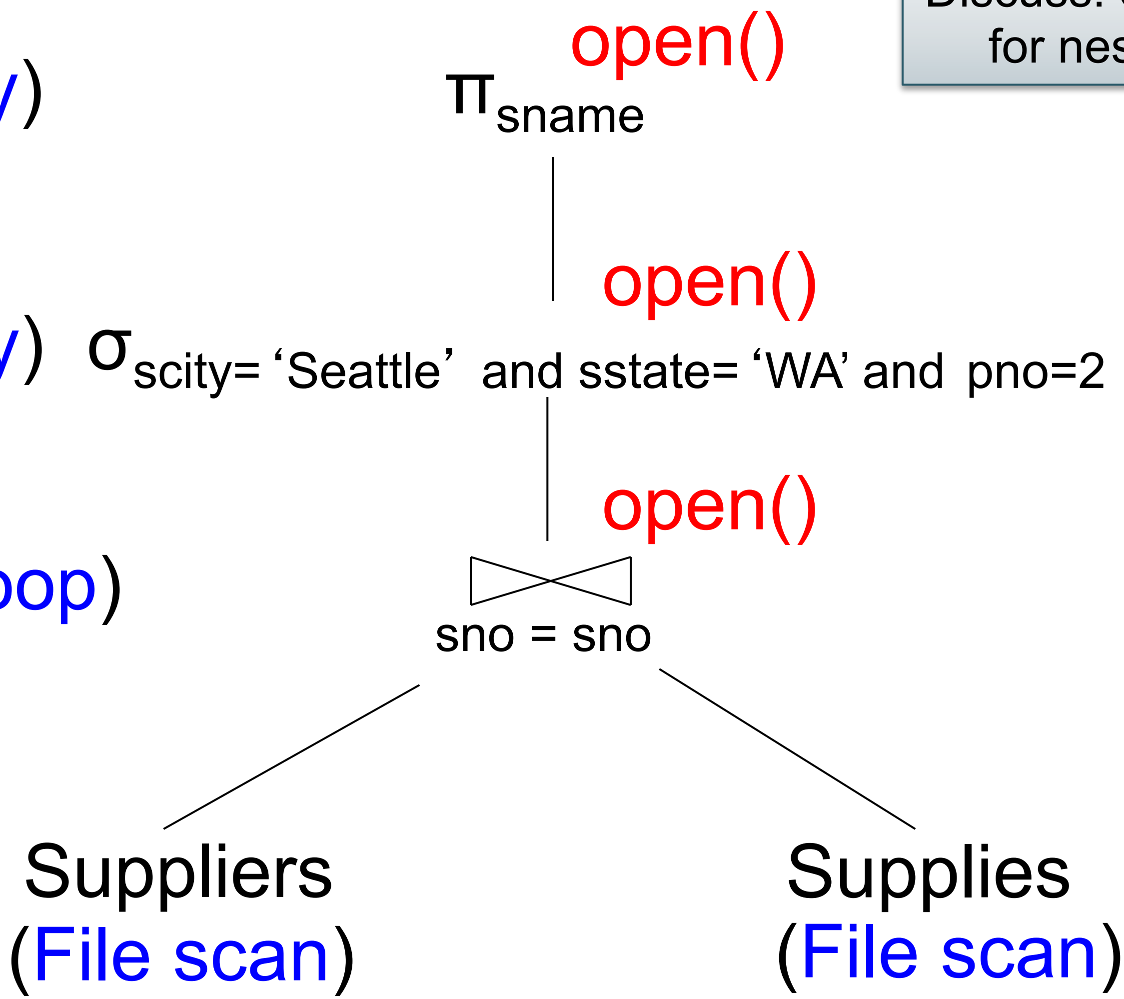
Pipelining

Discuss: open/next/close for nested loop join

(On the fly)

(On the fly)

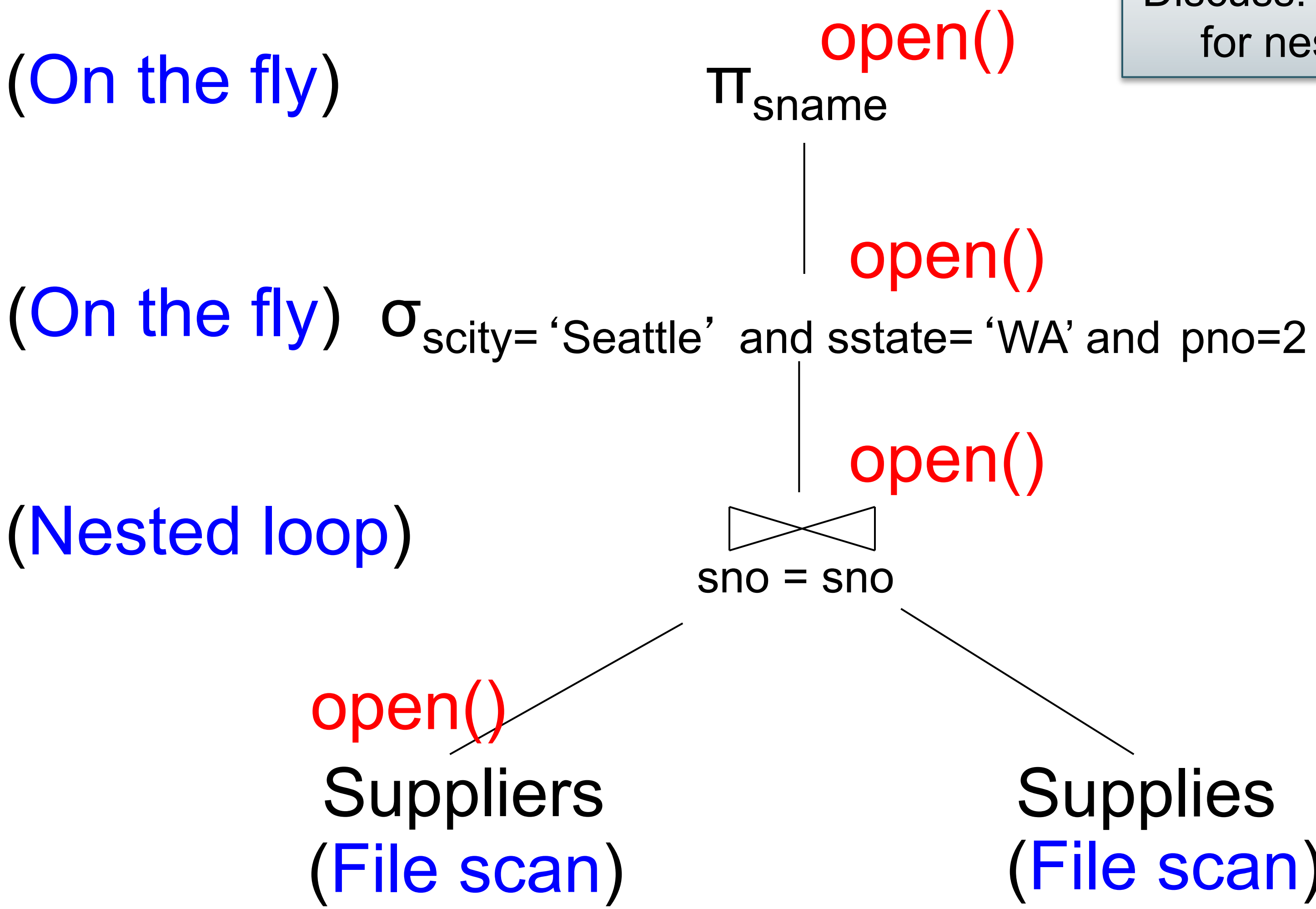
(Nested loop)



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

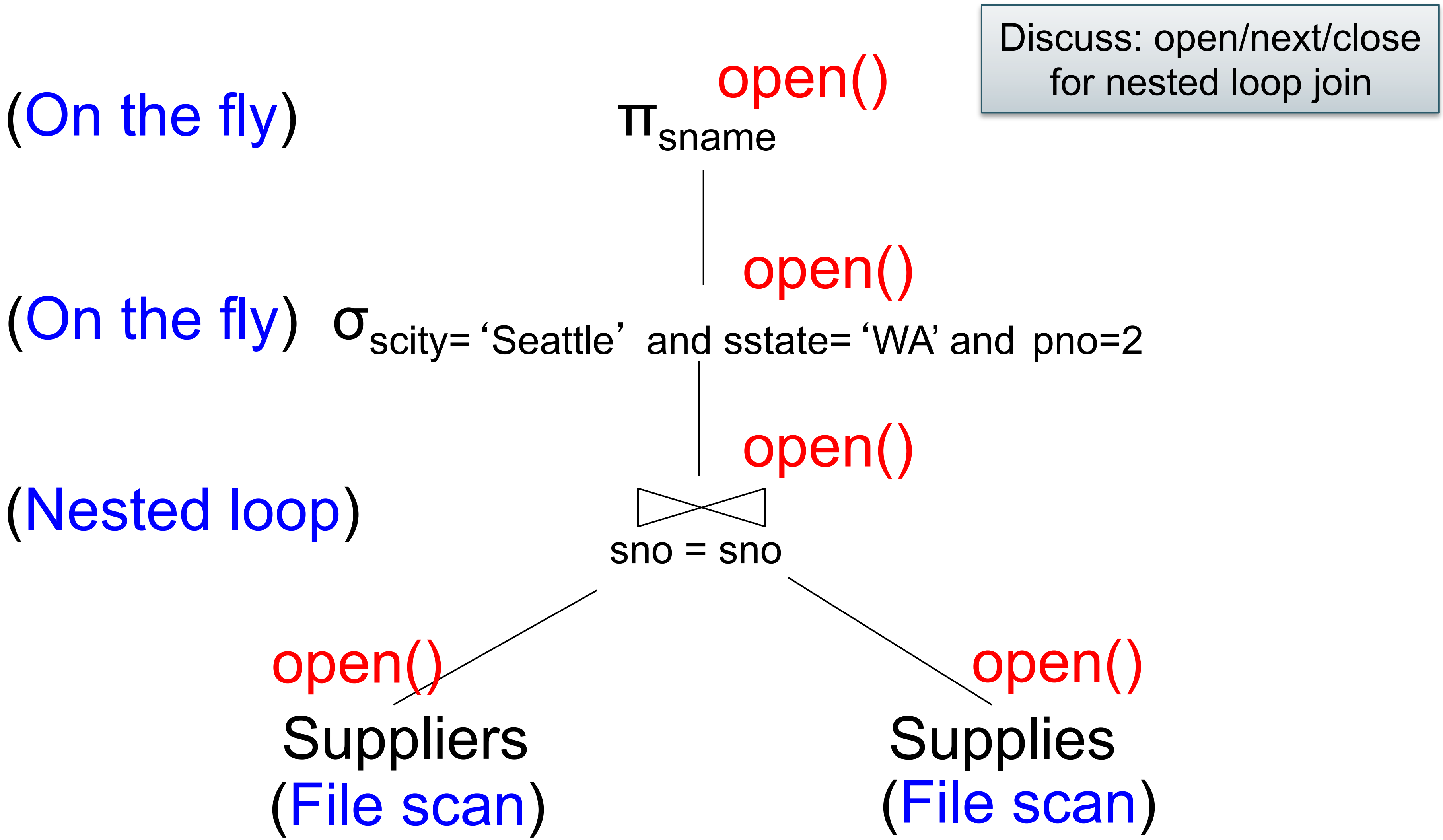
Pipelining

Discuss: open/next/close for nested loop join



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

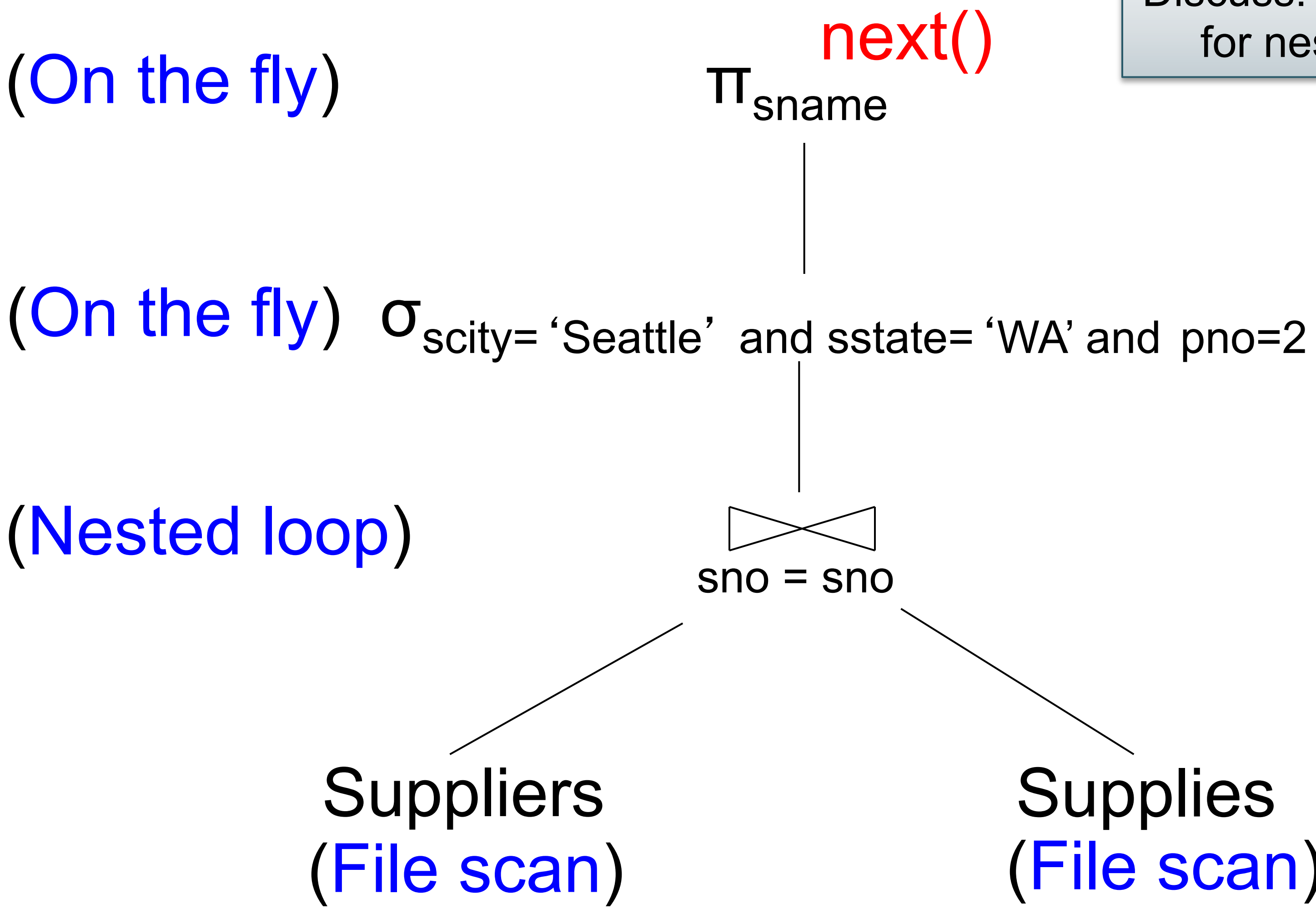
Pipelining



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

Pipelining

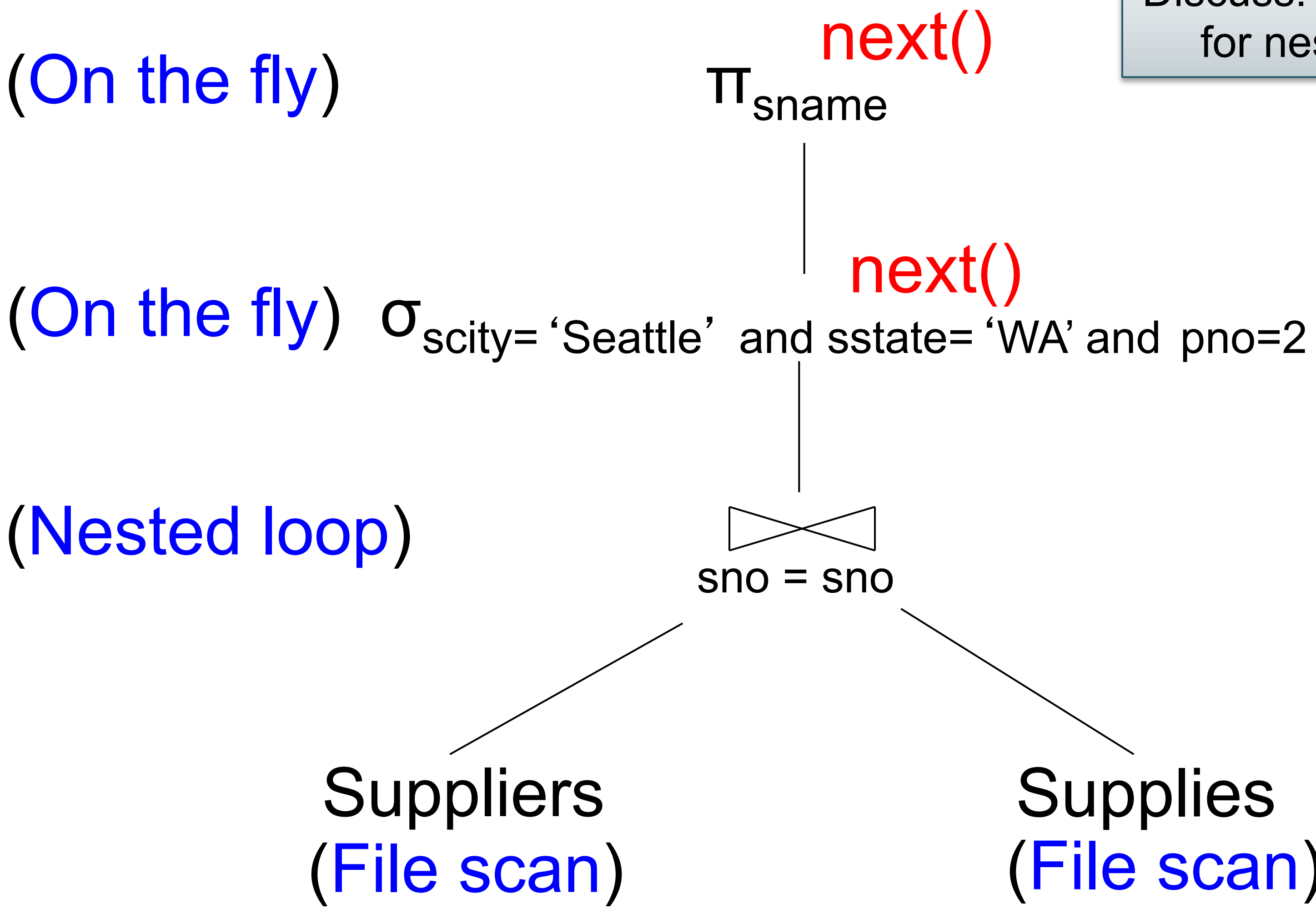
Discuss: open/next/close for nested loop join



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

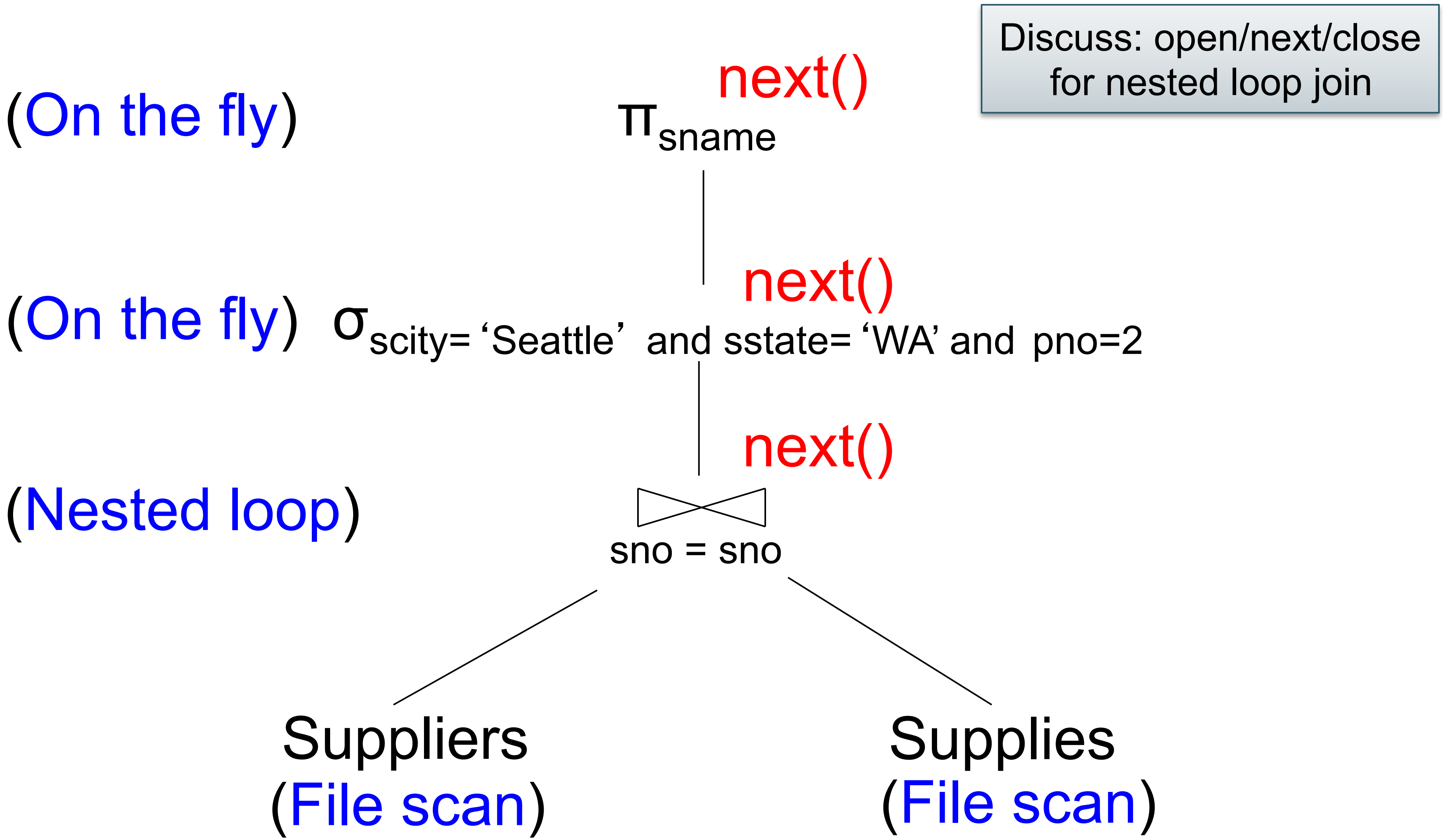
Pipelining

Discuss: open/next/close for nested loop join



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

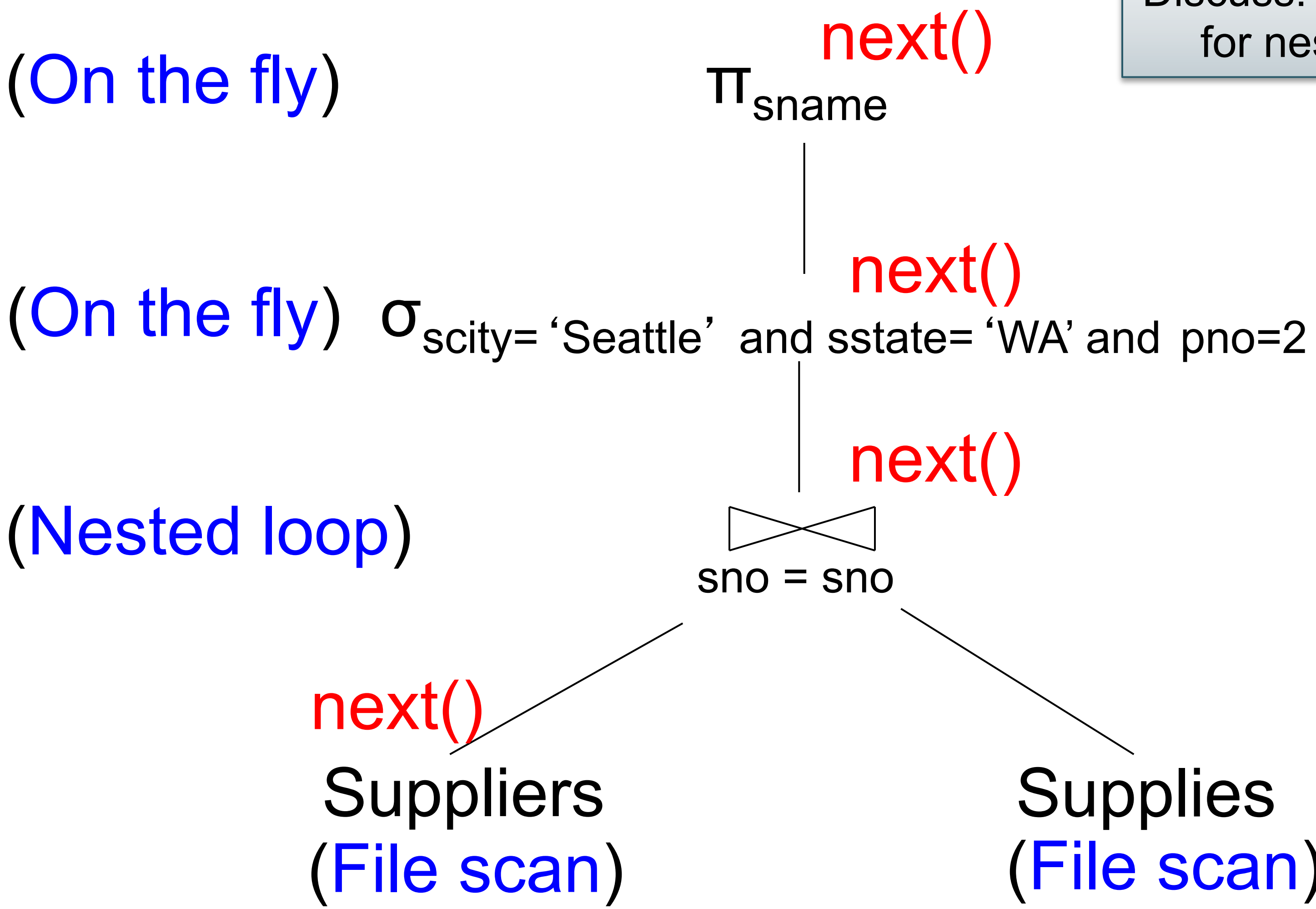
Pipelining



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

Pipelining

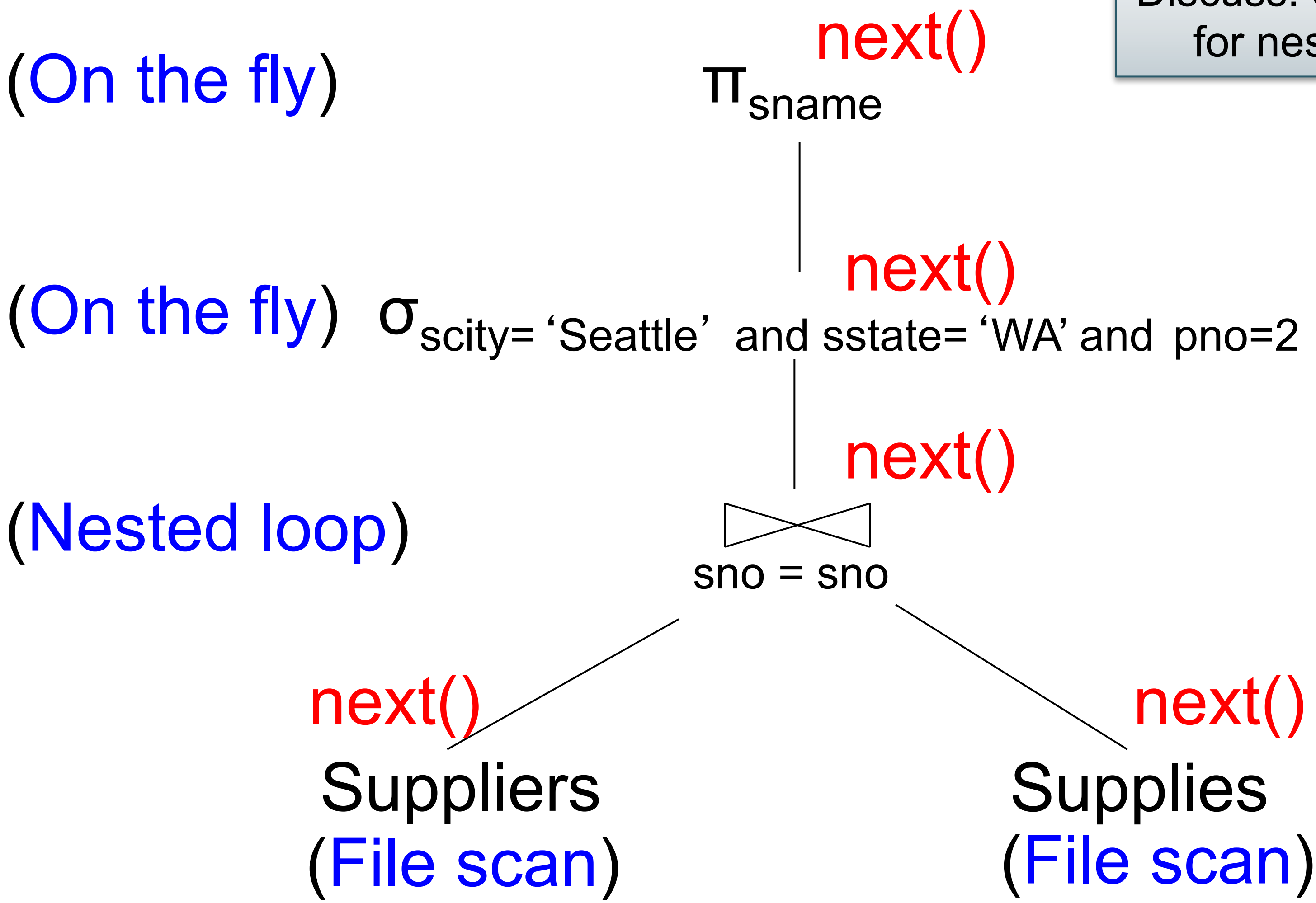
Discuss: open/next/close for nested loop join



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

Pipelining

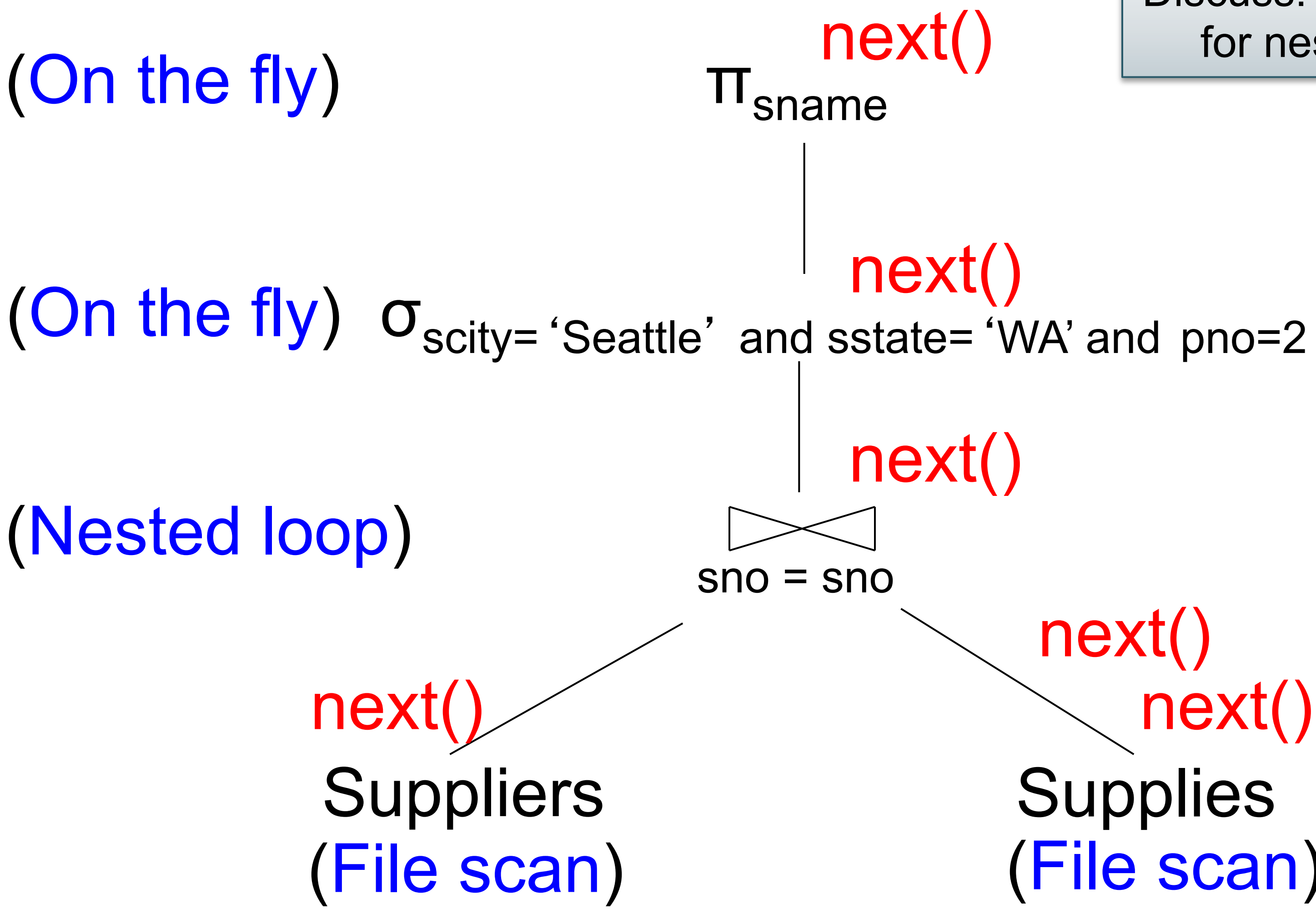
Discuss: open/next/close for nested loop join



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

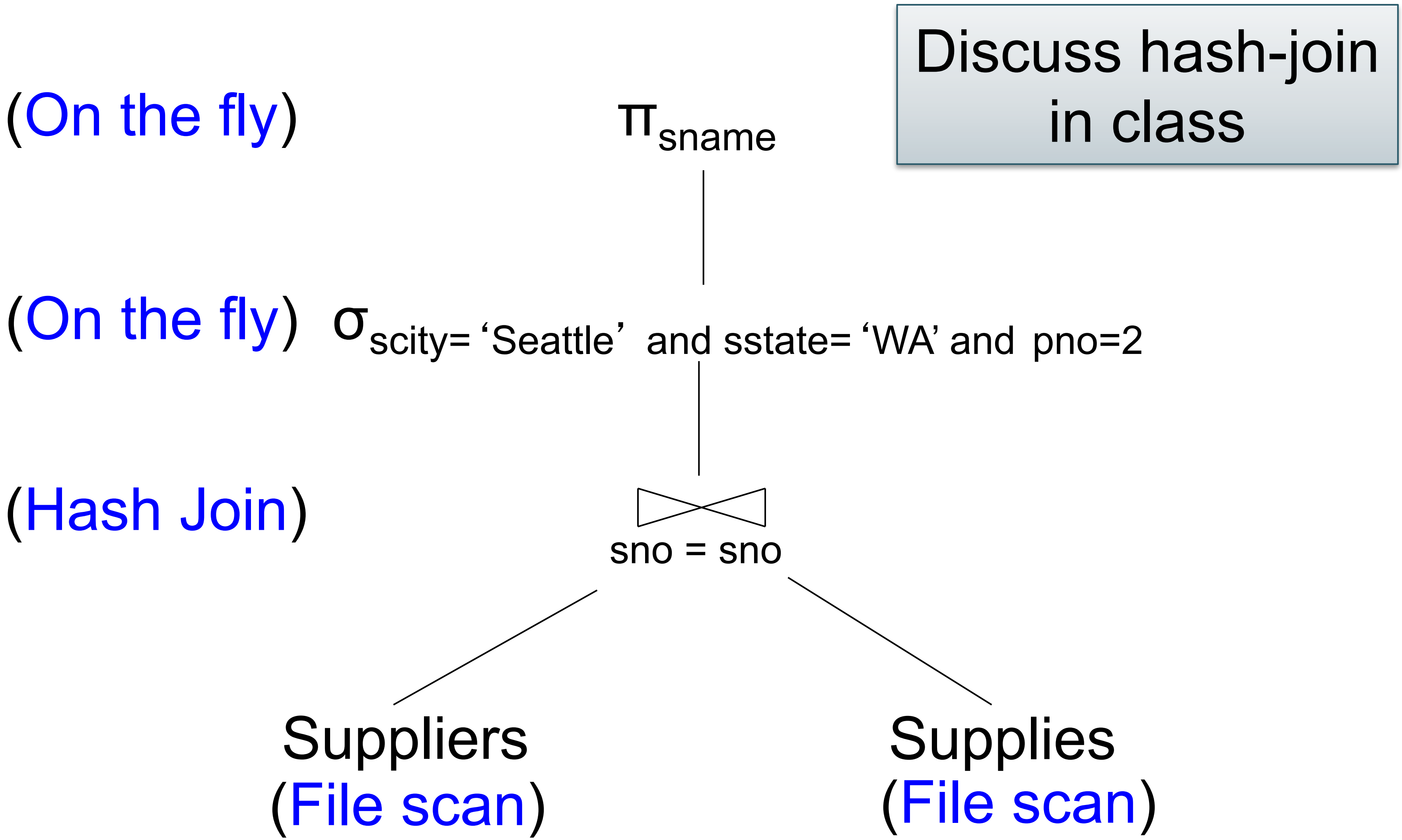
Pipelining

Discuss: open/next/close for nested loop join



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

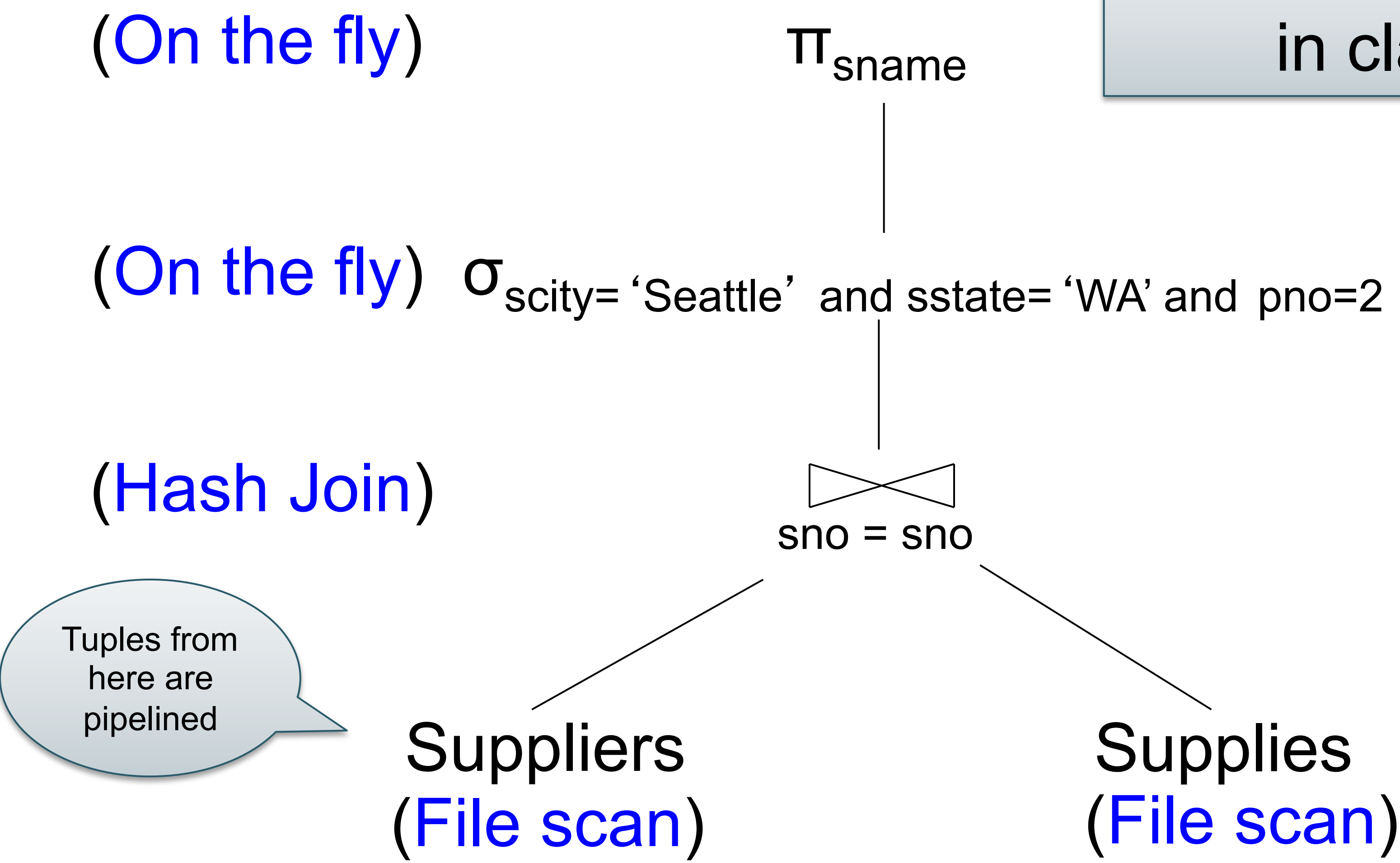
Pipelining



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

(On the fly) π_{sname}

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

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

Tuples from here are "blocked"

Tuples from here are pipelined

Suppliers (File scan)

Supplies (File scan)

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

Blocked Execution

(On the fly) π_{sname}

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

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

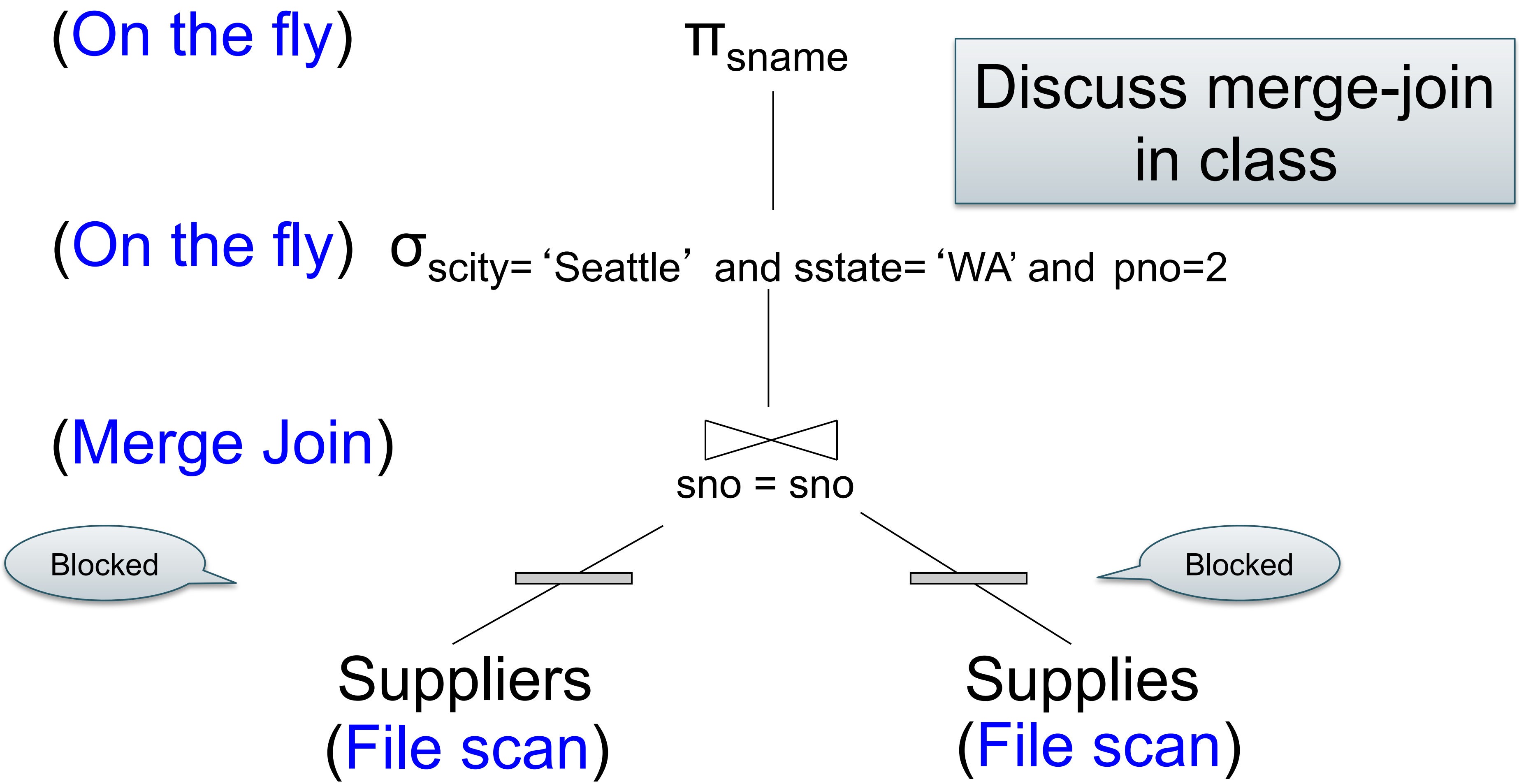
Suppliers (File scan)

Supplies (File scan)

Discuss merge-join in class

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

Blocked Execution



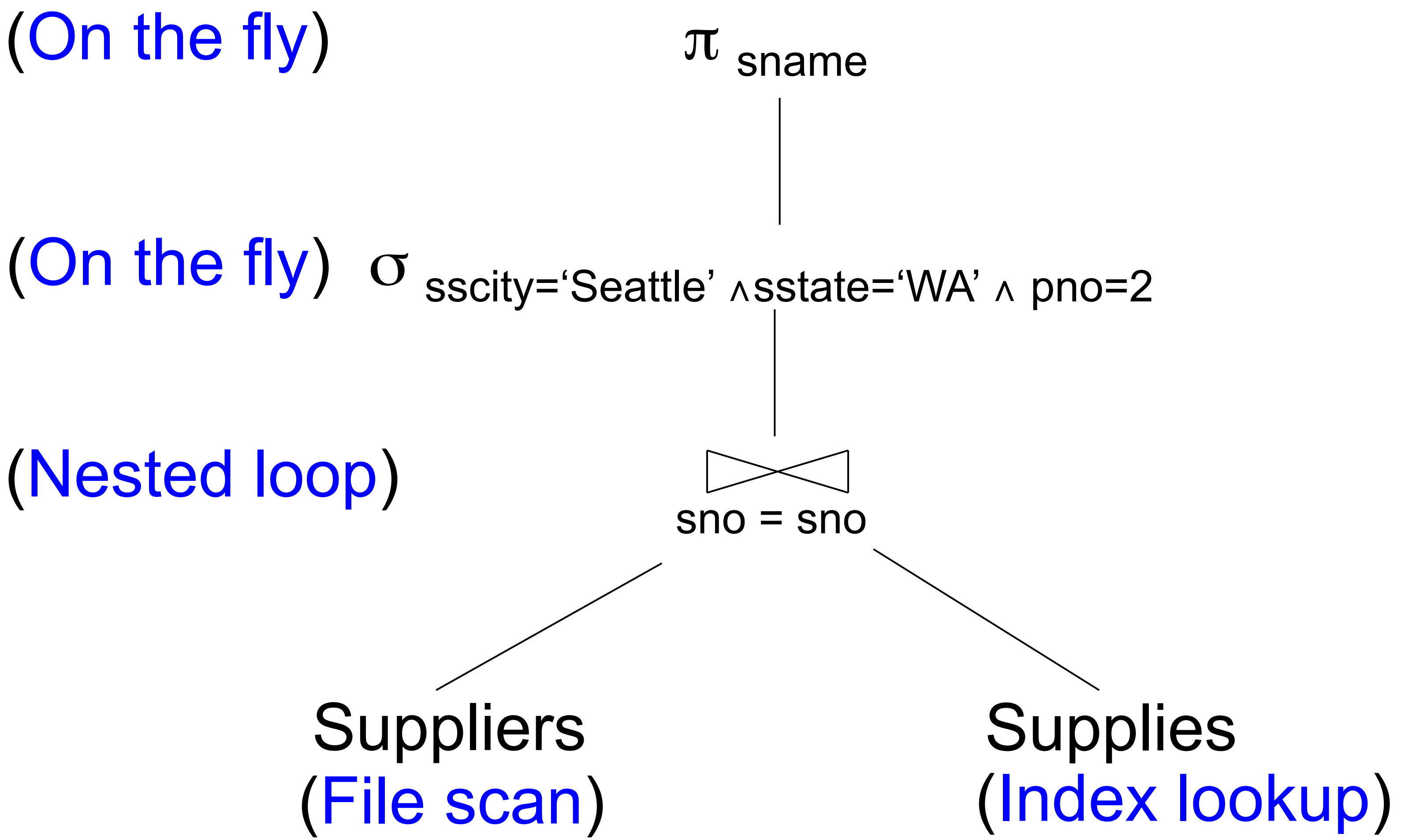
Pipelined Execution

- Applies parent operator to tuples directly as they are produced by child operators

- Benefits
  - No operator synchronization issues
  - Saves cost of writing intermediate data to disk
  - Saves cost of reading intermediate data from disk
  - Good resource utilizations on single processor

- This approach is used whenever possible

Pipelined Execution

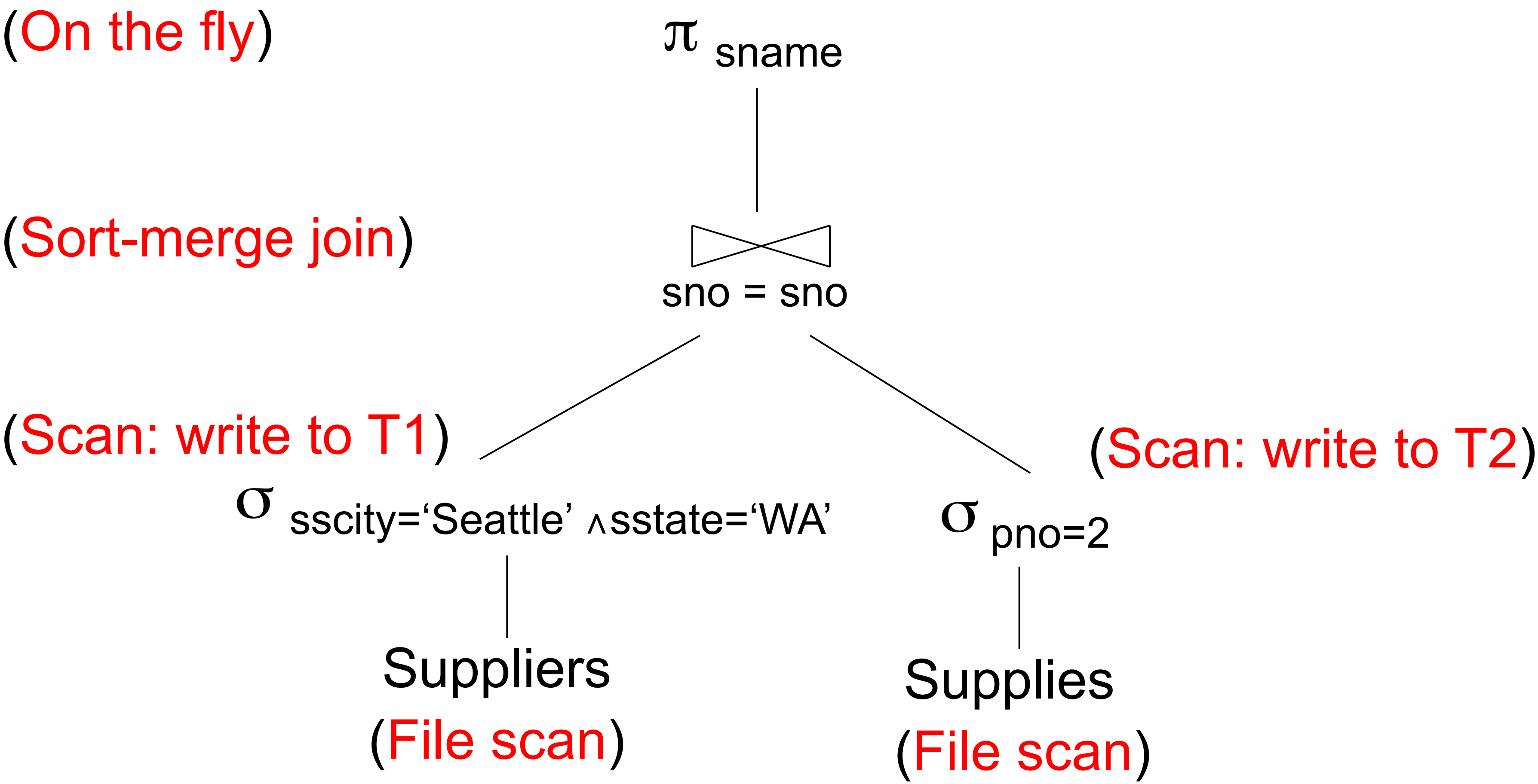


Intermediate Tuple Materialization

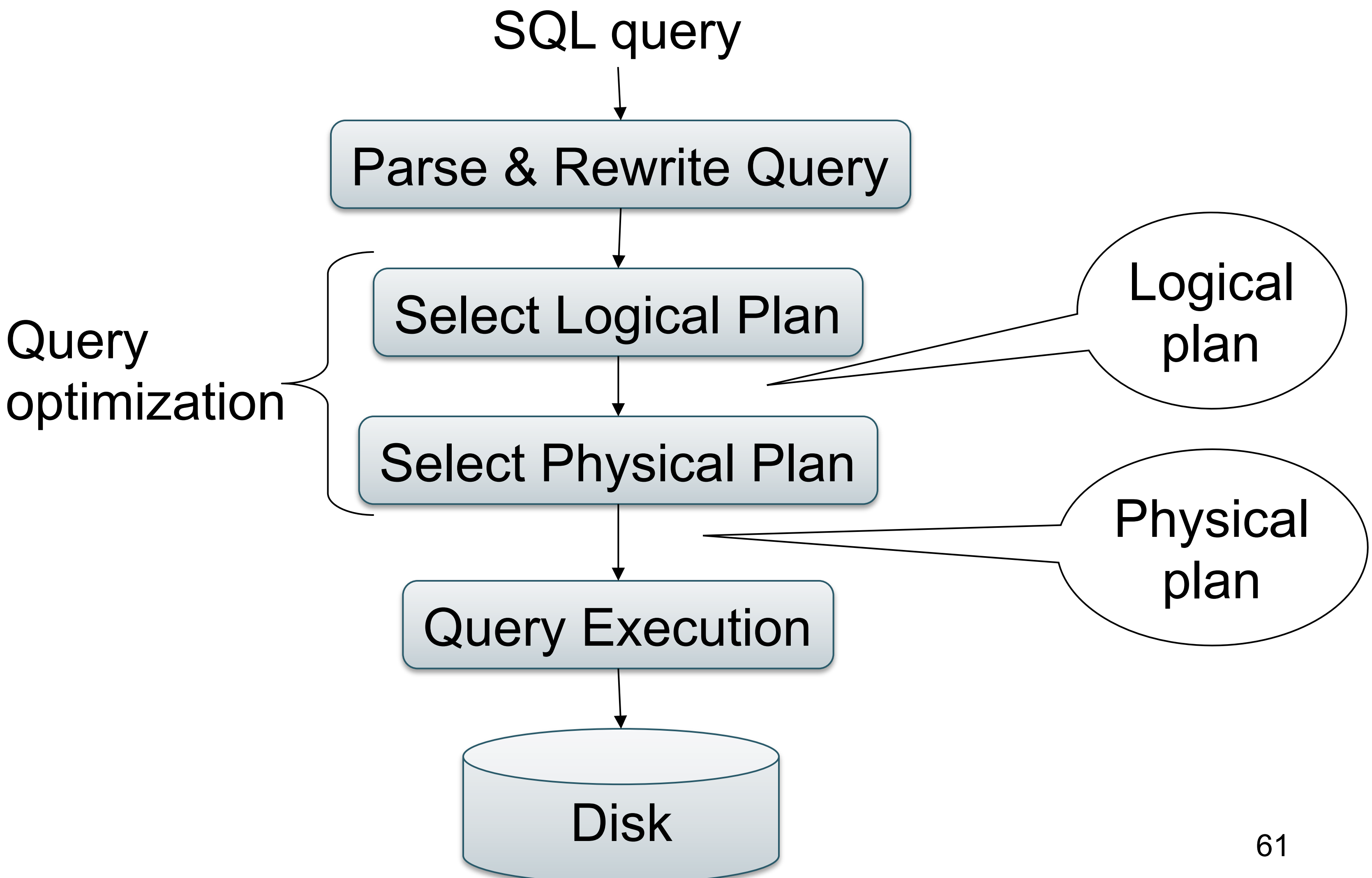
- Writes the results of an operator to an intermediate table on disk

- Necessary for some operator implementations
- When operator needs to examine the same tuples multiple times

Intermediate Tuple Materialization



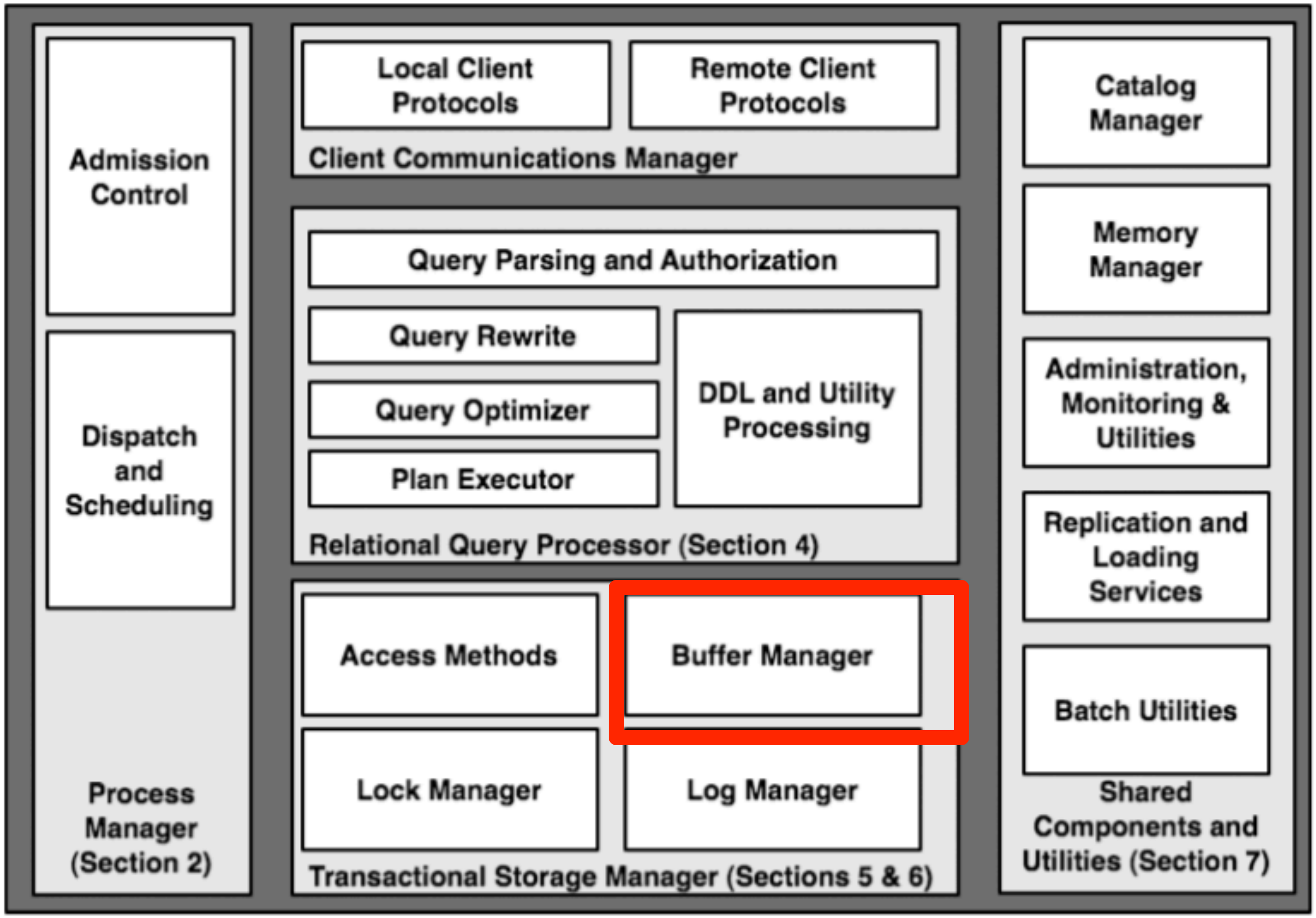
Lifecycle of a Query



Outline

- Architecture of a DBMS
- Steps involved in processing a query
- Operator implementations

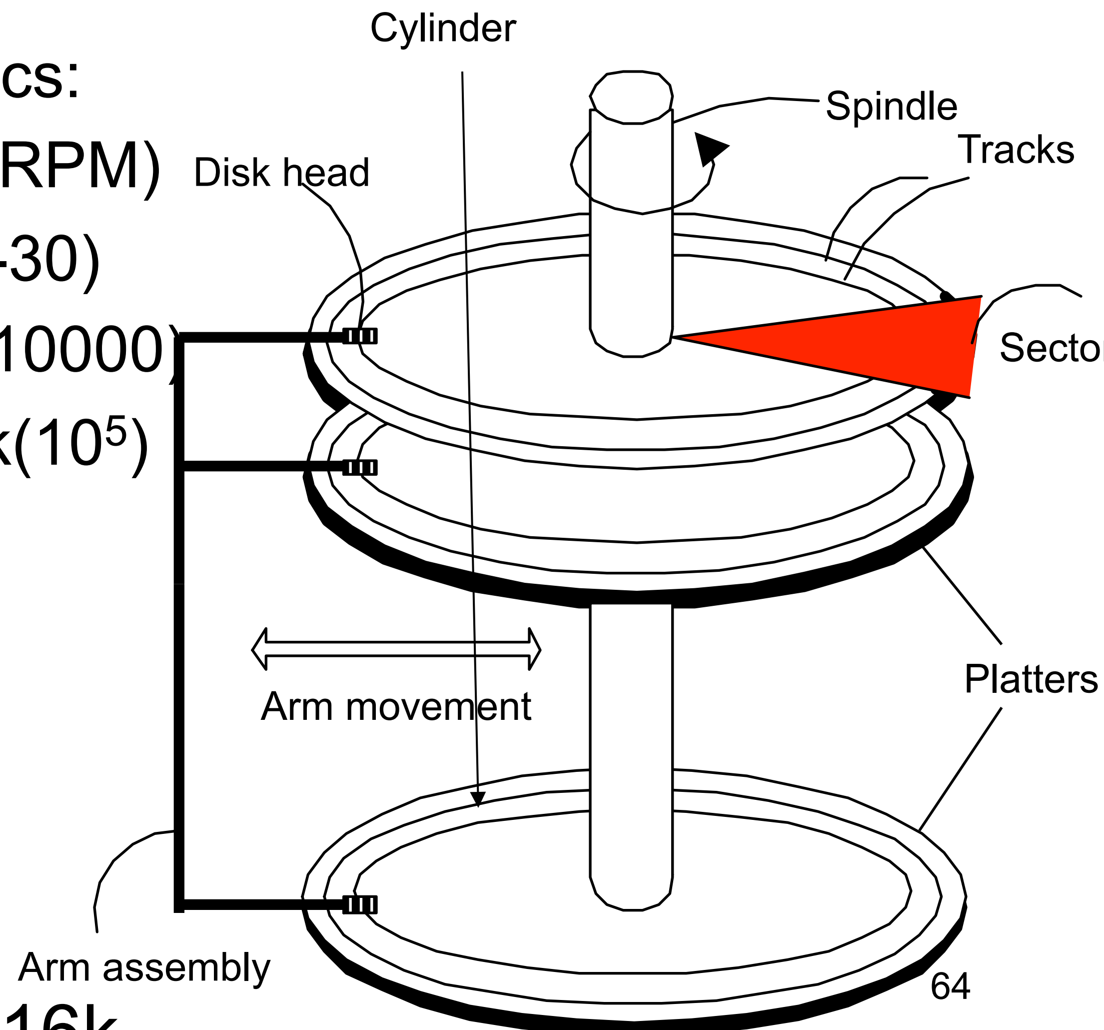
Multiple Processes



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

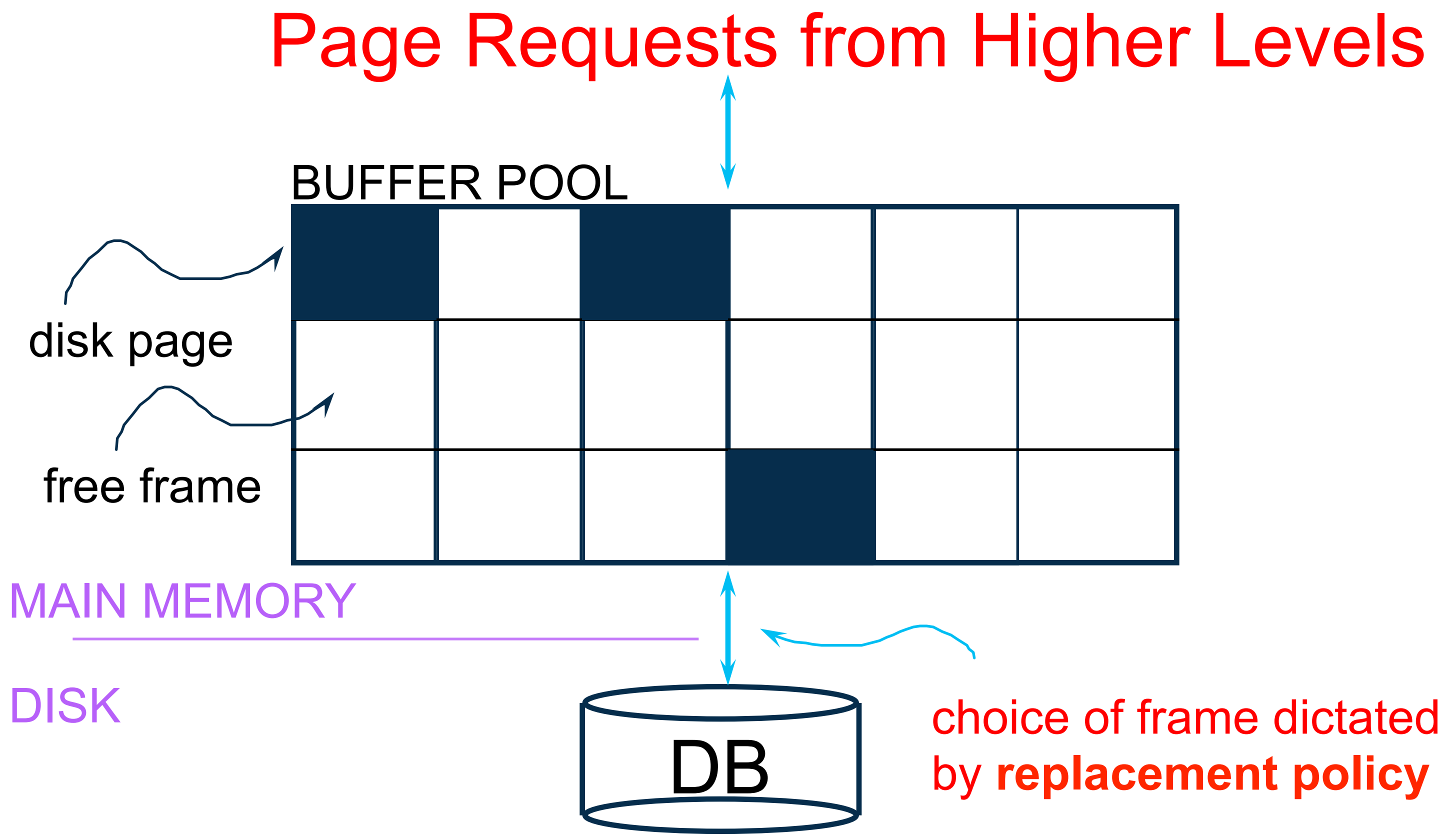
Typically: 4k or 8k or 16k

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

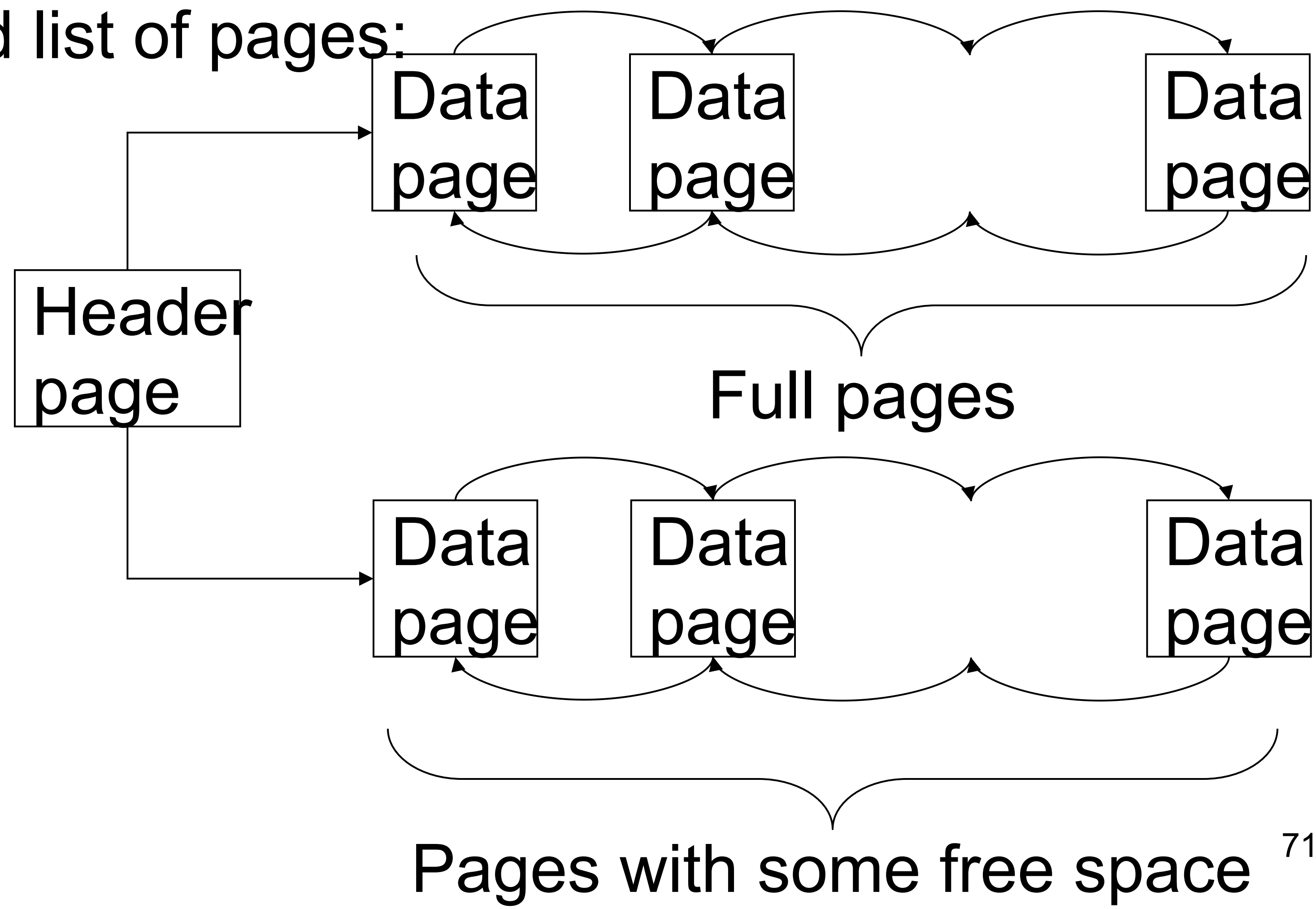
- Managing free blocks
- File Organization
- Represent the records inside the blocks
- Represent attributes inside the records

Managing Free Blocks

- Linked list of free blocks
- Or bit map

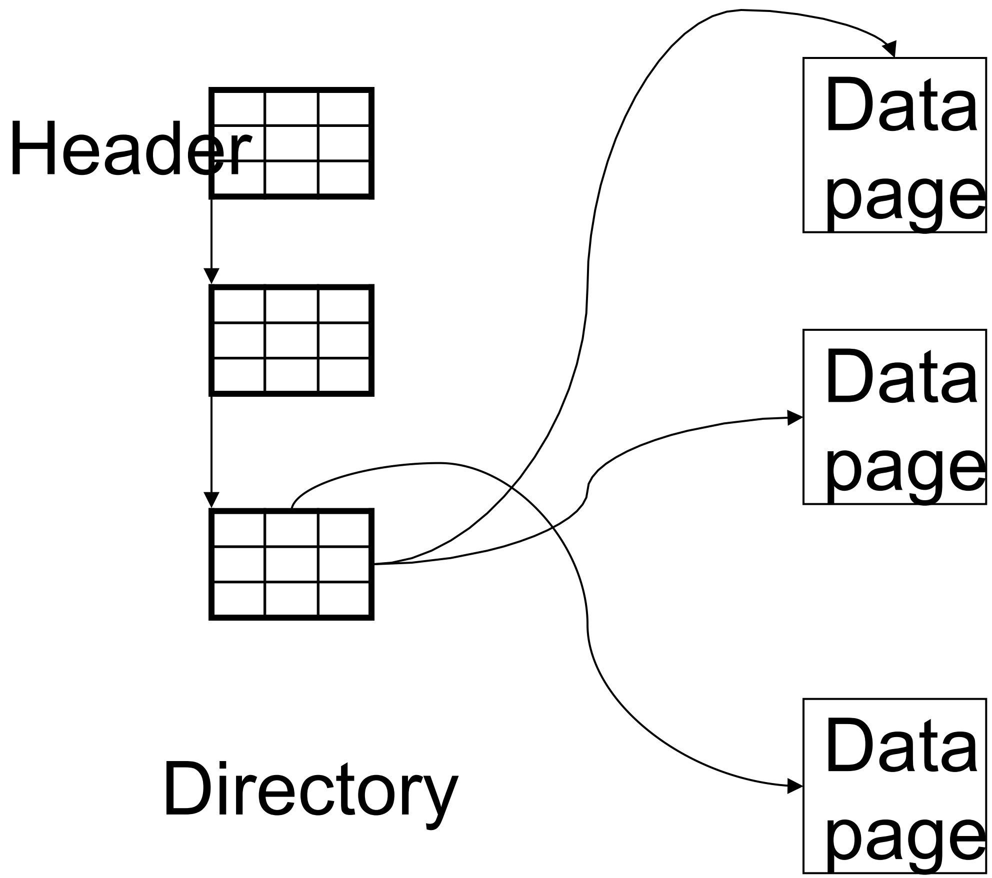
File Organization

Linked list of pages:



File Organization

Better: directory of pages



Page Formats

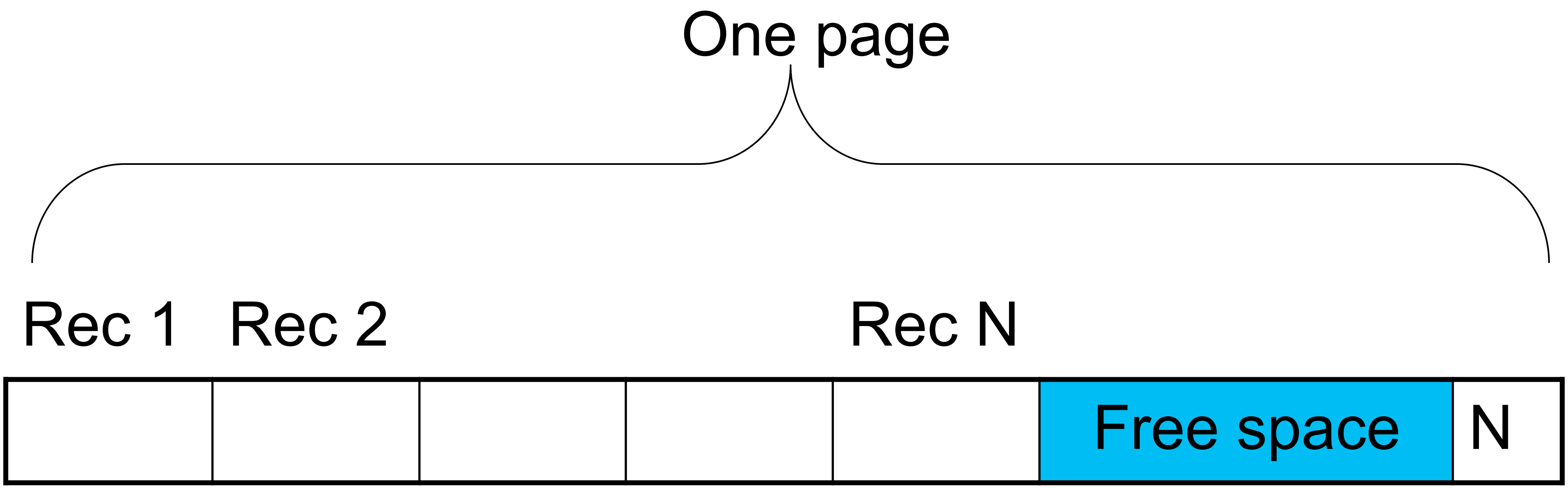
Issues to consider

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

Why do we need RID's in a relational DBMS ?

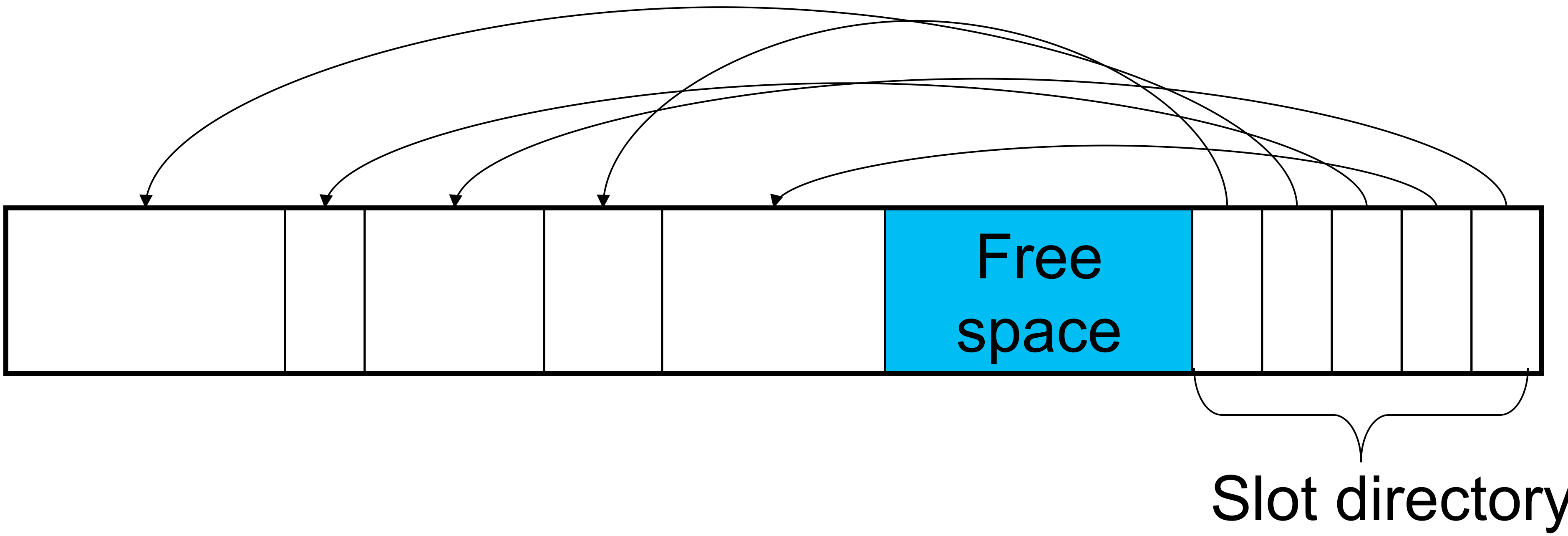
Page Formats

Fixed-length records: packed representation



Problems ?

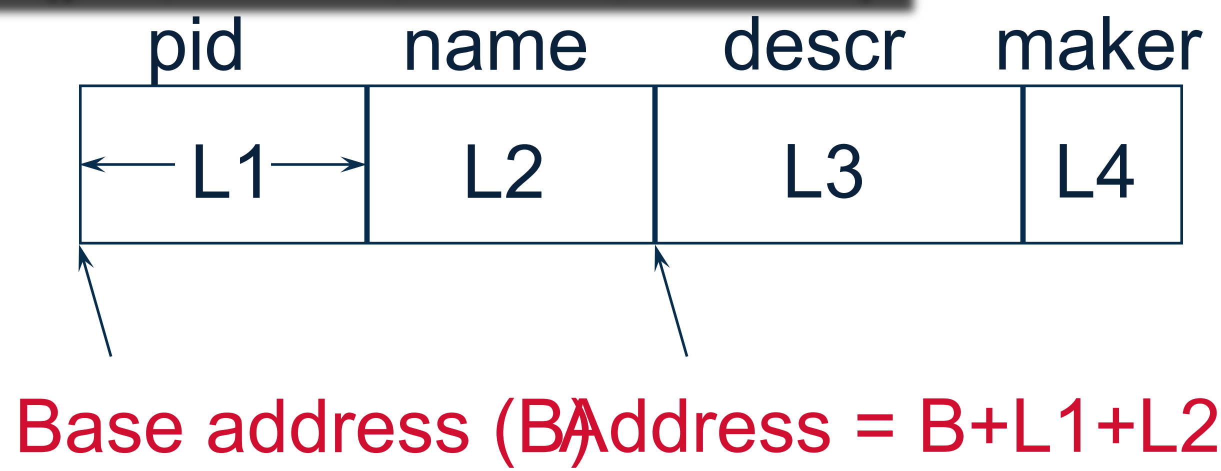
Page Formats



Variable-length records

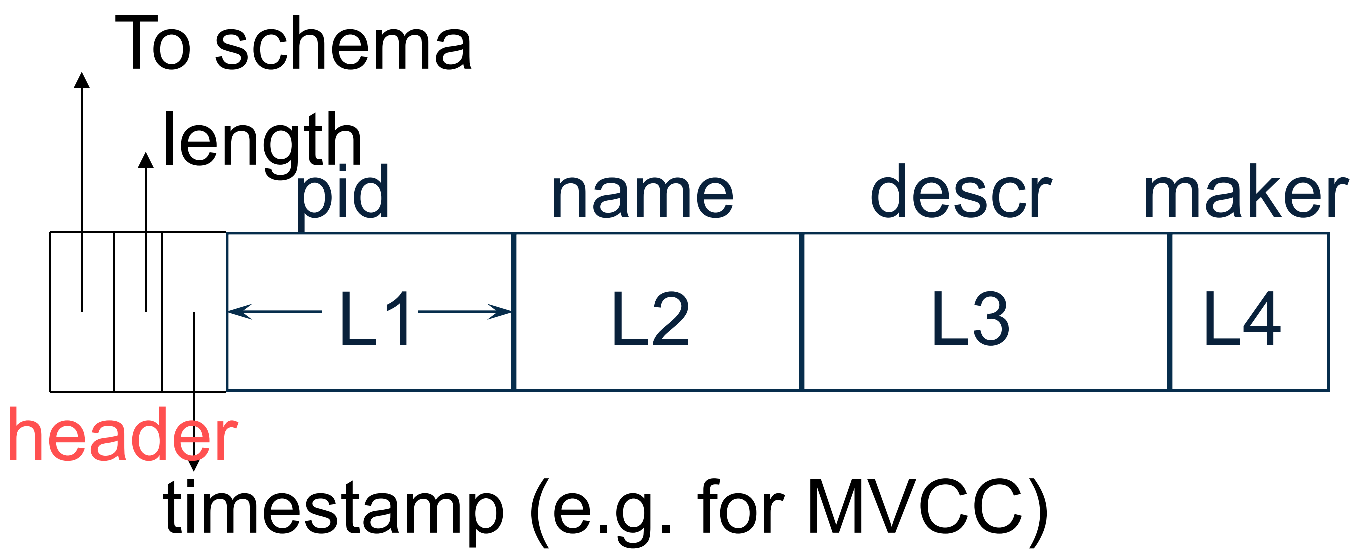
Record Formats: Fixed Length

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!

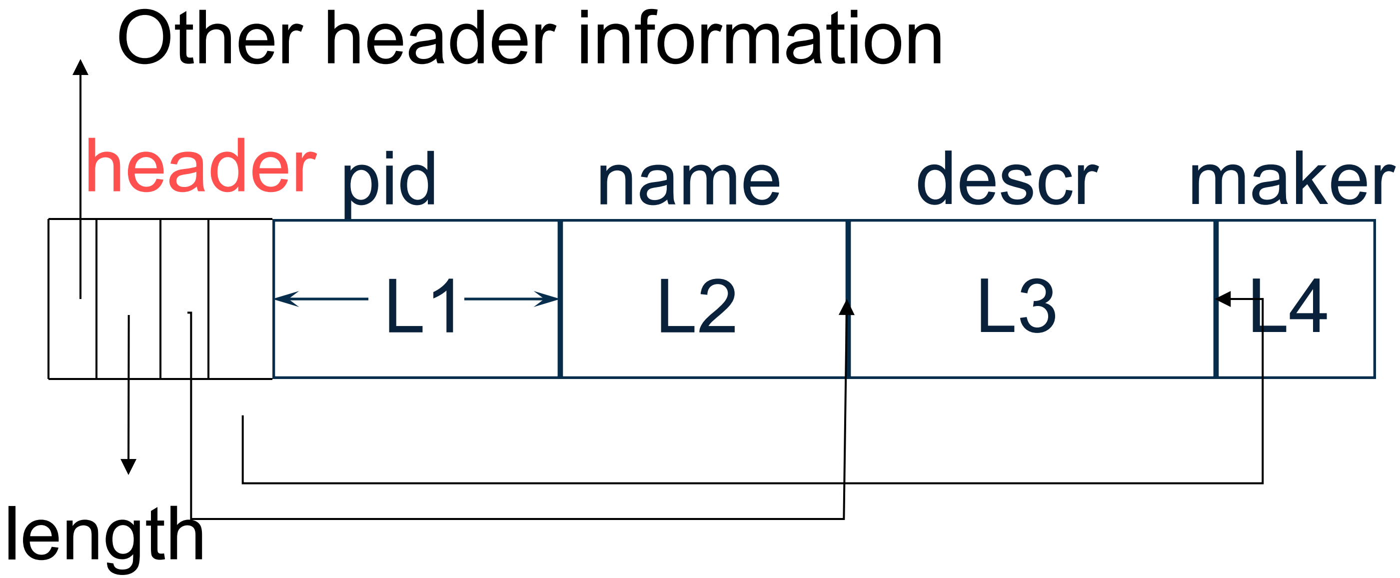
Record Header



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

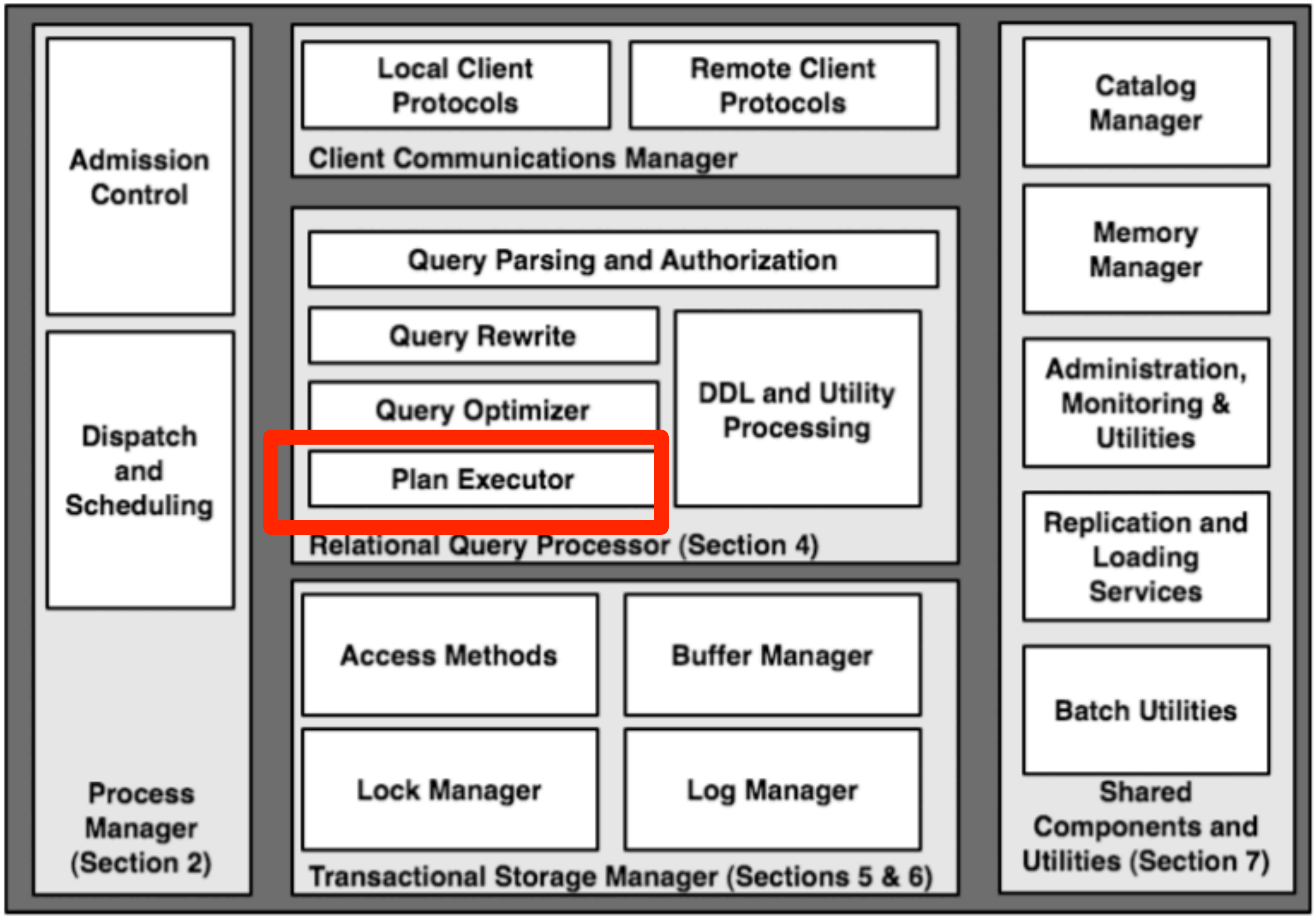
CLOB = character large object

- Supports only restricted operations

File Organizations

- Heap (random order) files: Suitable when typical access is a file scan retrieving all records.
- Sorted Files Best if records must be retrieved in some order, or only a `range' of records is needed.
- Indexes Data structures to organize records via trees or hashing.
  - Like sorted files, they speed up searches for a subset of records, based on values in certain ("search key") fields
  - Updates are much faster than in sorted files.

Multiple Processes



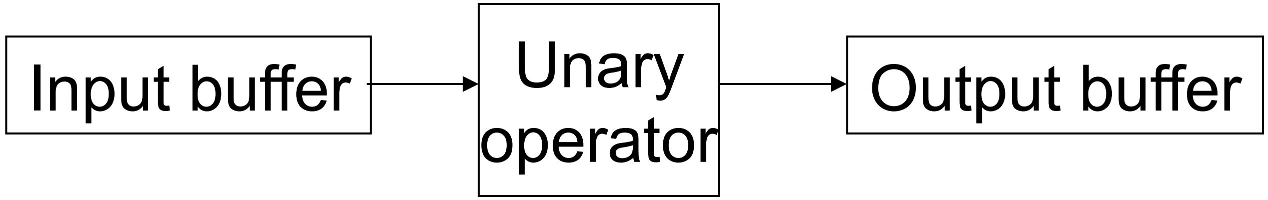
Cost Parameters

- 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*

One-pass Algorithms

Selection $\sigma(R)$, projection $\Pi(R)$

- Both are ***tuple-at-a-time*** algorithms
- Cost: B(R), the cost of scanning the relation



Main Memory Join Algorithms

Three standard main memory algorithms:

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

Review in class

One Pass Hash Join

Hash join: R ⋈ 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 $\bowtie$ 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

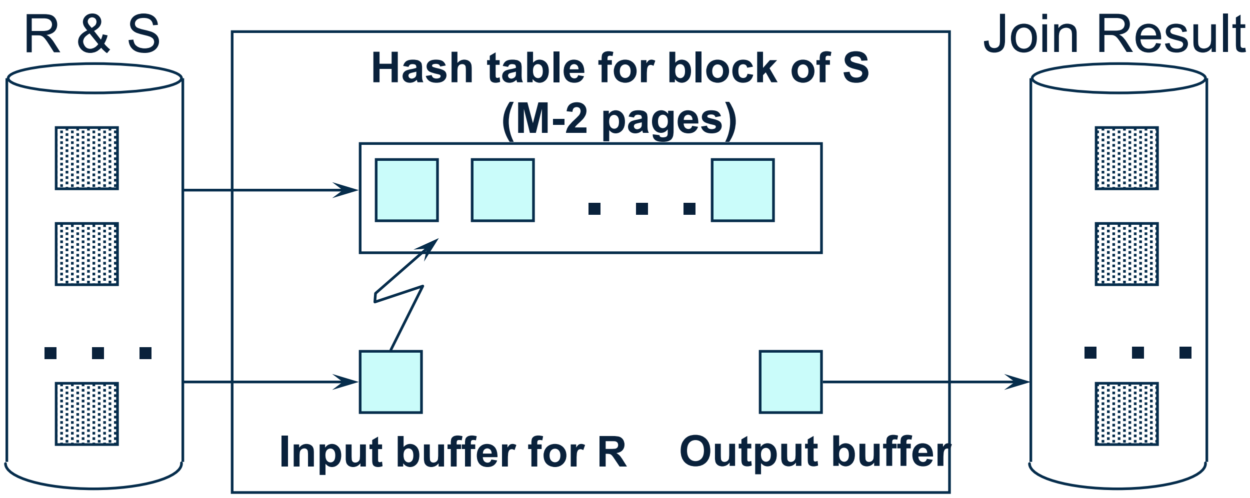
Nested Loop Joins

- 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 $ps \bowtie pr$

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: $B(S) + B(S)B(R)/(M-2)$

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

Example

Grouping:

Product(name, department, quantity)

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

In class: describe a one-pass algorithms. Cost=?

Outline

- **Steps involved in processing a query**
  - Logical query plan
  - Physical query plan
  - Query execution overview

- **Operator implementations**
  - One pass algorithms
  - Two-pass algorithms
  - Index-based algorithms

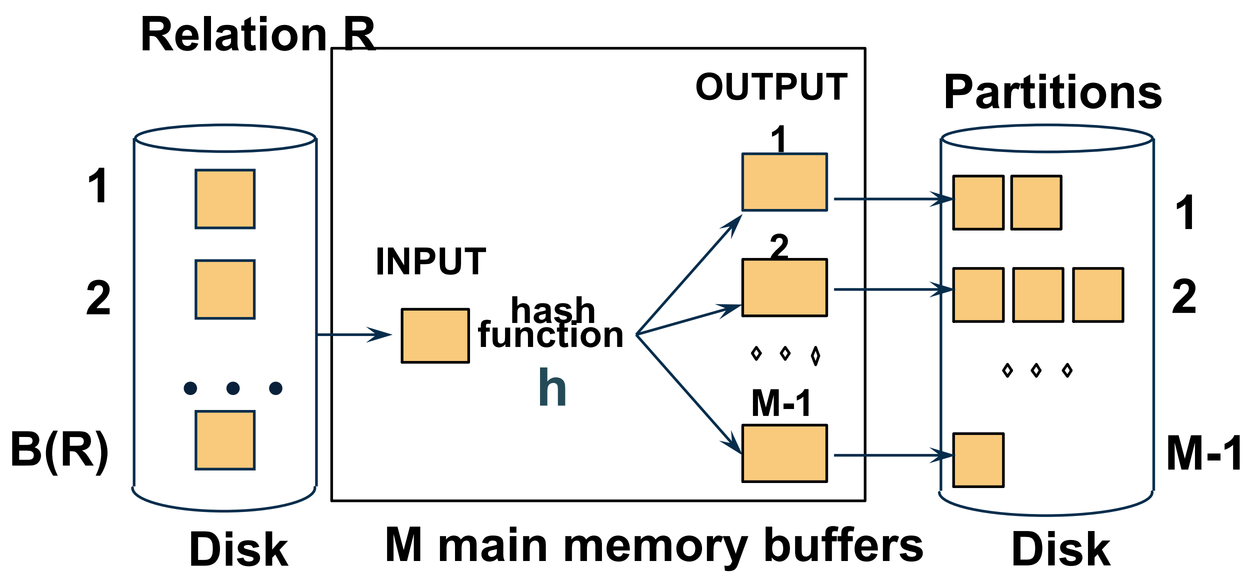
Two-Pass Algorithms

- 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
- Each bucket has size approx. B(R)/M



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

Hash Based Algorithms for γ

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

- Step 1. Partition R into buckets
- Step 2. Apply γ to each bucket

- Cost: 3B(R)
- Assumption: $B(R) <= M^2$

Partitioned (Grace) Hash Join

$R \bowtie S$

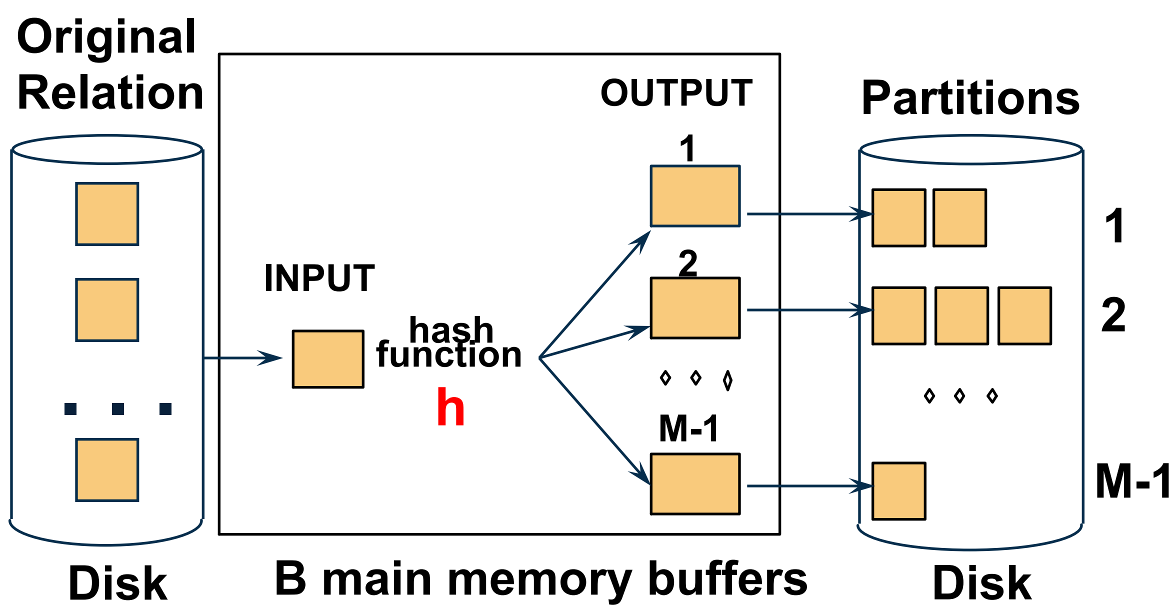
- Step 1:
  - Hash S into M-1 buckets
  - Send all buckets to disk

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

- Step 3
  - Join every pair of buckets

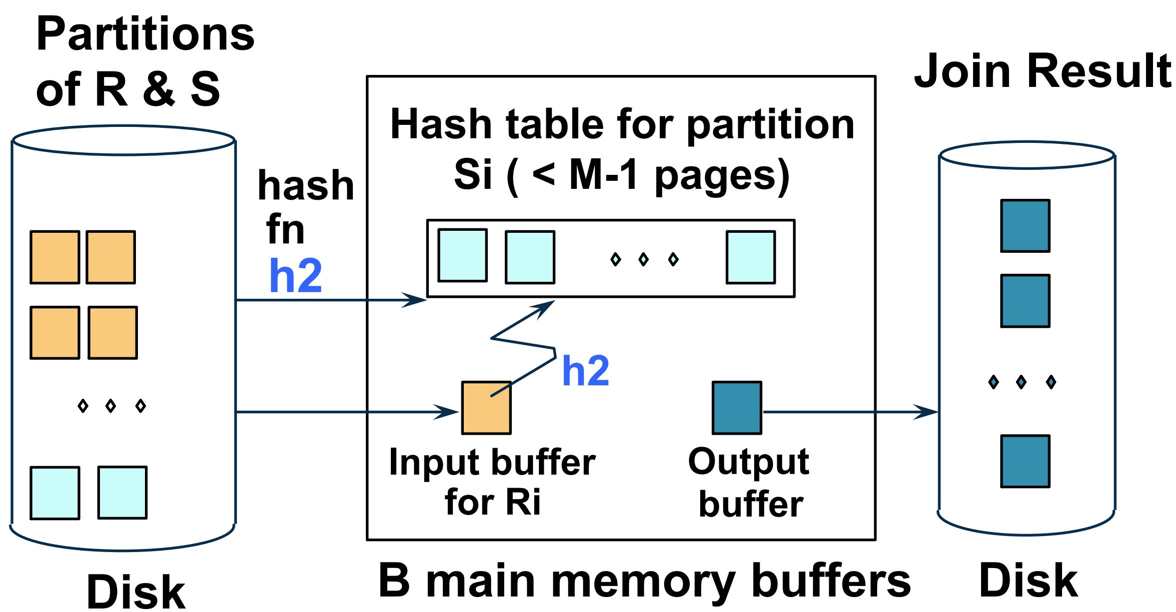
Partitioned Hash Join

- Partition both relations using hash fn **h**
- R tuples in partition i will only match S tuples in partition i.



Partitioned Hash Join

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



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: $t/k * B(S) \leq M$
  - 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

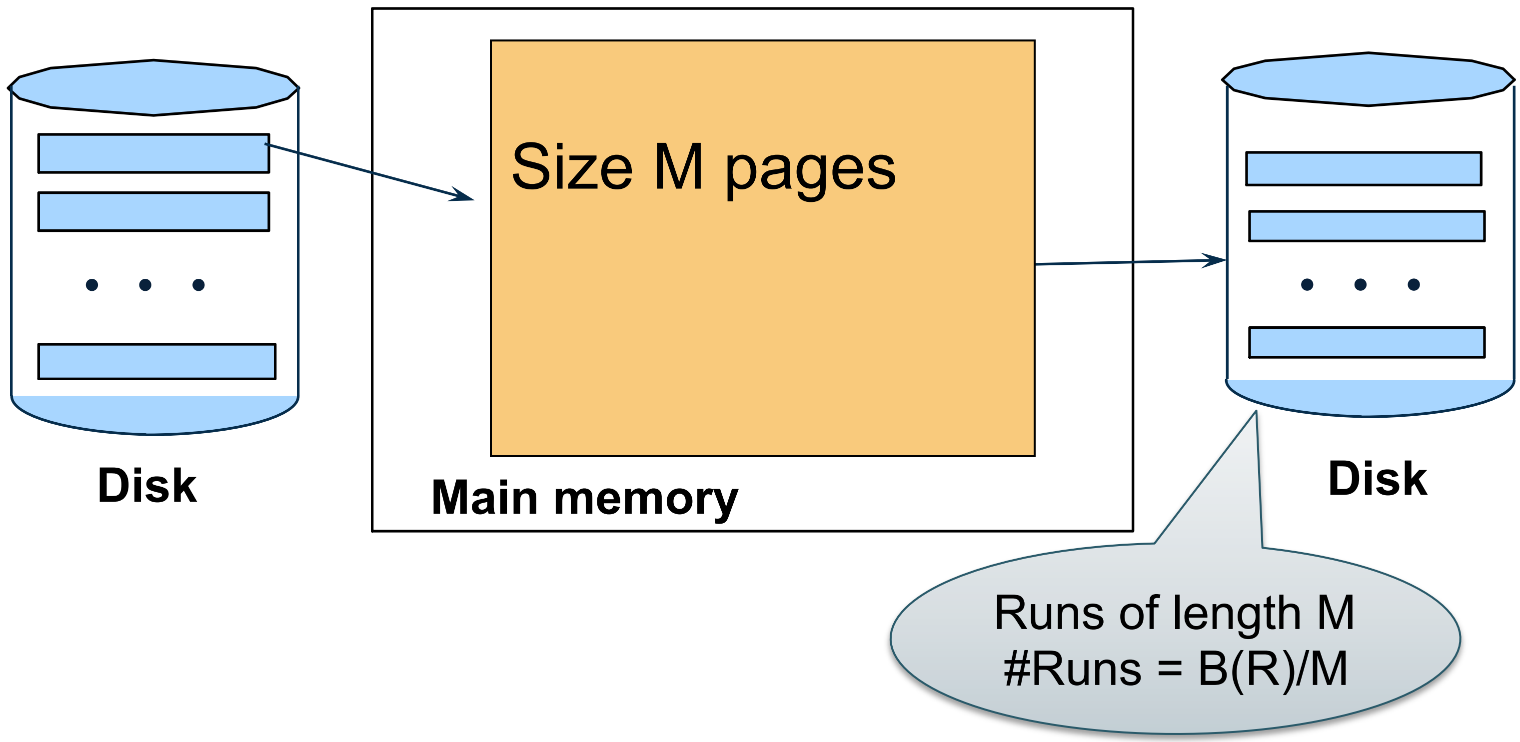
- Cost: (3-2t/k)(B(R) + B(S)) = (3-2M/B(S))(B(R) + B(S))

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 $B < M^2$

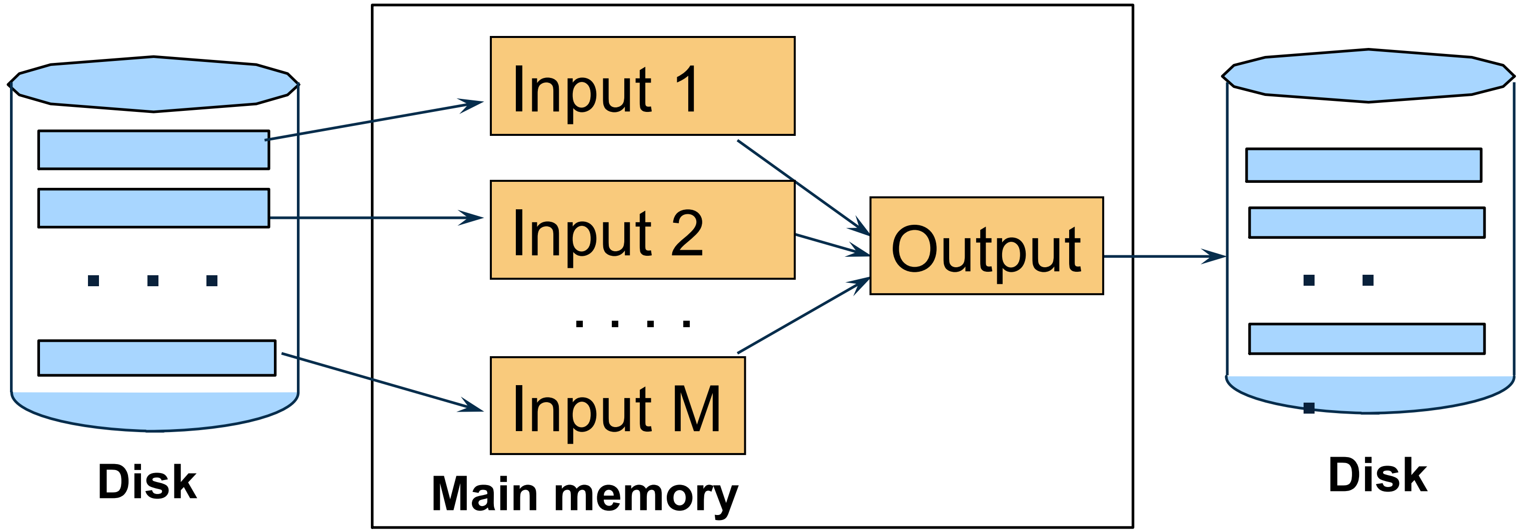
External Merge-Sort: Step 1

- 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



If B <= M^2 then 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) <= 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: $B(R)+B(S) \leq M^2$

Index

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

Key = means here search key

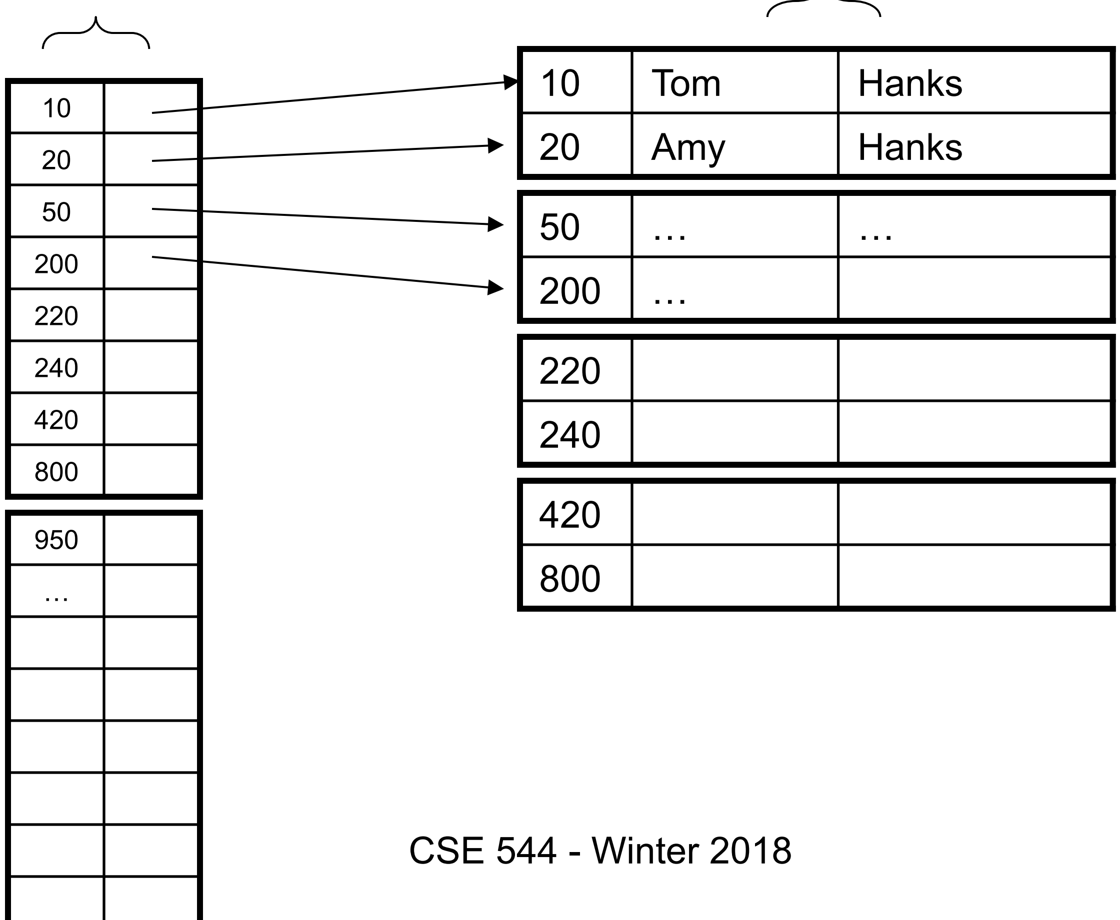
Example 1: Index on ID

Student

ID	fName	lName
10	Tom	Hanks
20	Amy	Hanks
…		

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

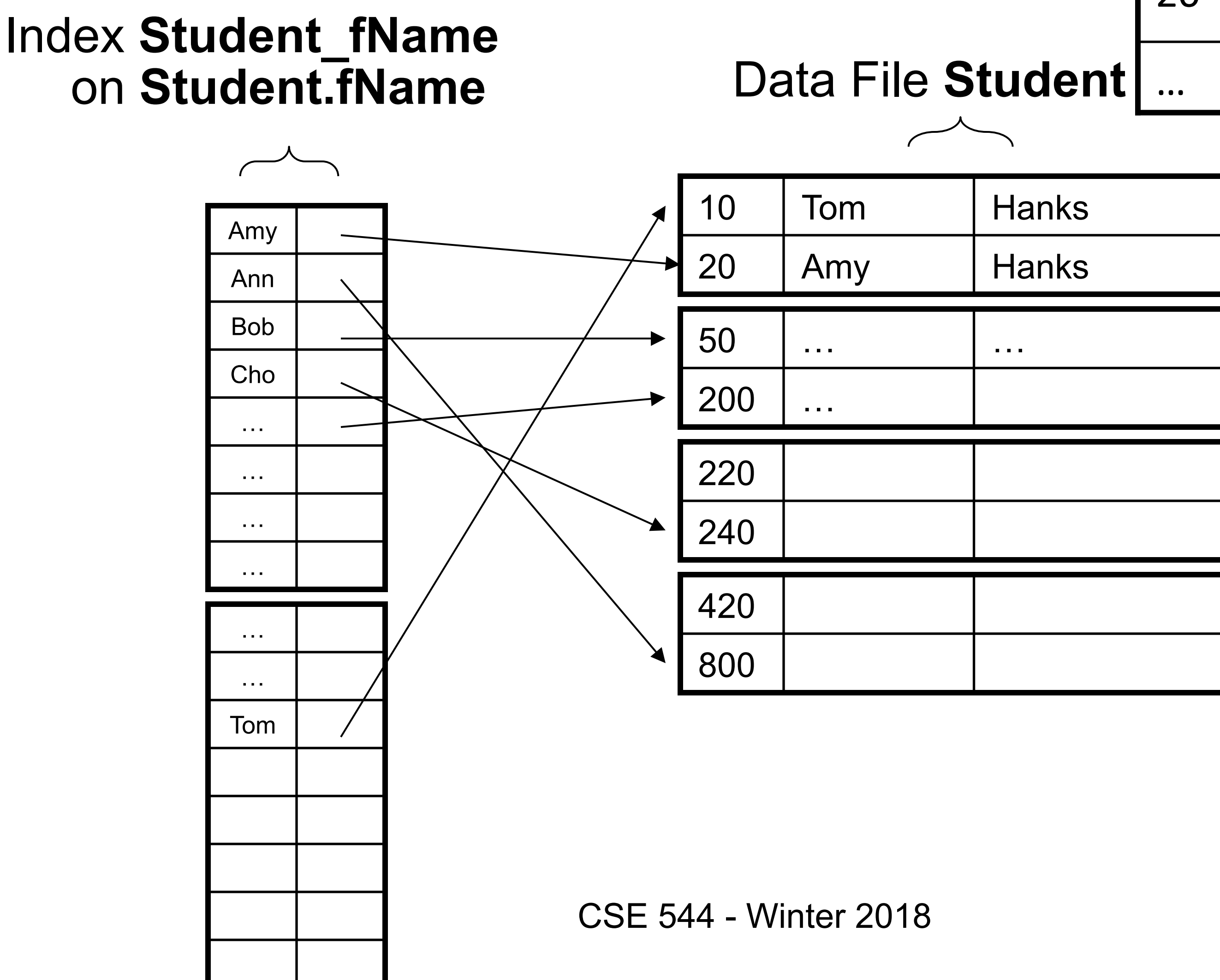
Data File **Student**



Example 2: Index on fName

Student

ID	fName	lName
10	Tom	Hanks
20	Amy	Hanks
…		



Index Organization

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

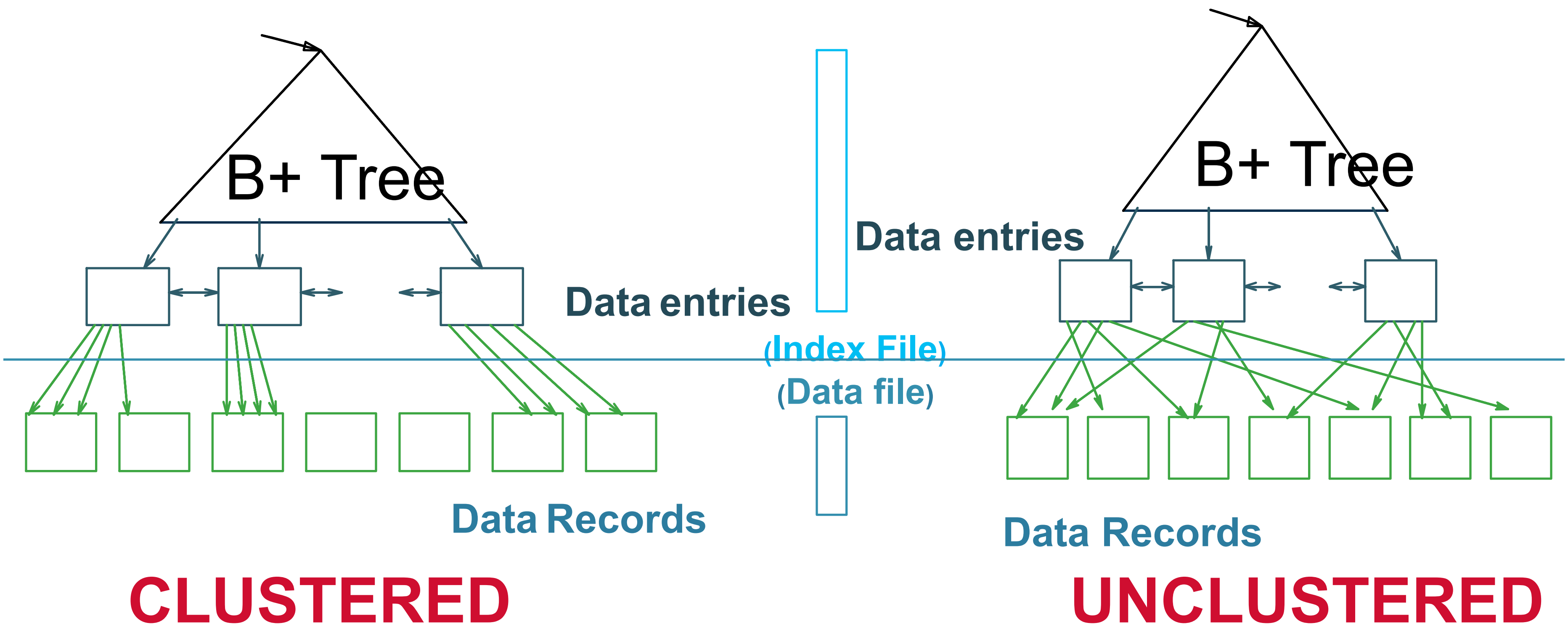
Several ways to do this:

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

Review: 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

Clustered vs Unclustered



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

Index Based Selection

- Selection on equality: $\sigma_{a=v}(R)$
- V(R, a) = # of distinct values of attribute a
- Clustered index on a: cost B(R)/V(R,a)
- Unclustered index on a: cost T(R)/V(R,a)
- Note: we ignored the I/O cost for the index pages (why?)

Index Based Selection

- Example:

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

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

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

Index Based Selection

- Example:

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

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

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

The 2% rule!

Note: the "2" in 2% decreases yearly (why?)

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: $B(R) + T(R)T(S)/V(S,a)$

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