Database Management Systems
CSEP 544

Lecture 5: SQL++
Query Execution and Optimization

Announcements

- Please use the correct tags for your HW / RA!
  - We will start deducting points / not grade them.

- HW4 due today

- HW5 released
  - Please start early!
  - Use “hw5” / “asterixdb” tag to ask questions on Piazza

- Two lectures next week (Tues and Thurs)

- Today:
  - AsterixDB / SQL++ (wrap up)
  - RDBMS implementation and query optimization

A Case Study: AsterixDB

JSON - Overview

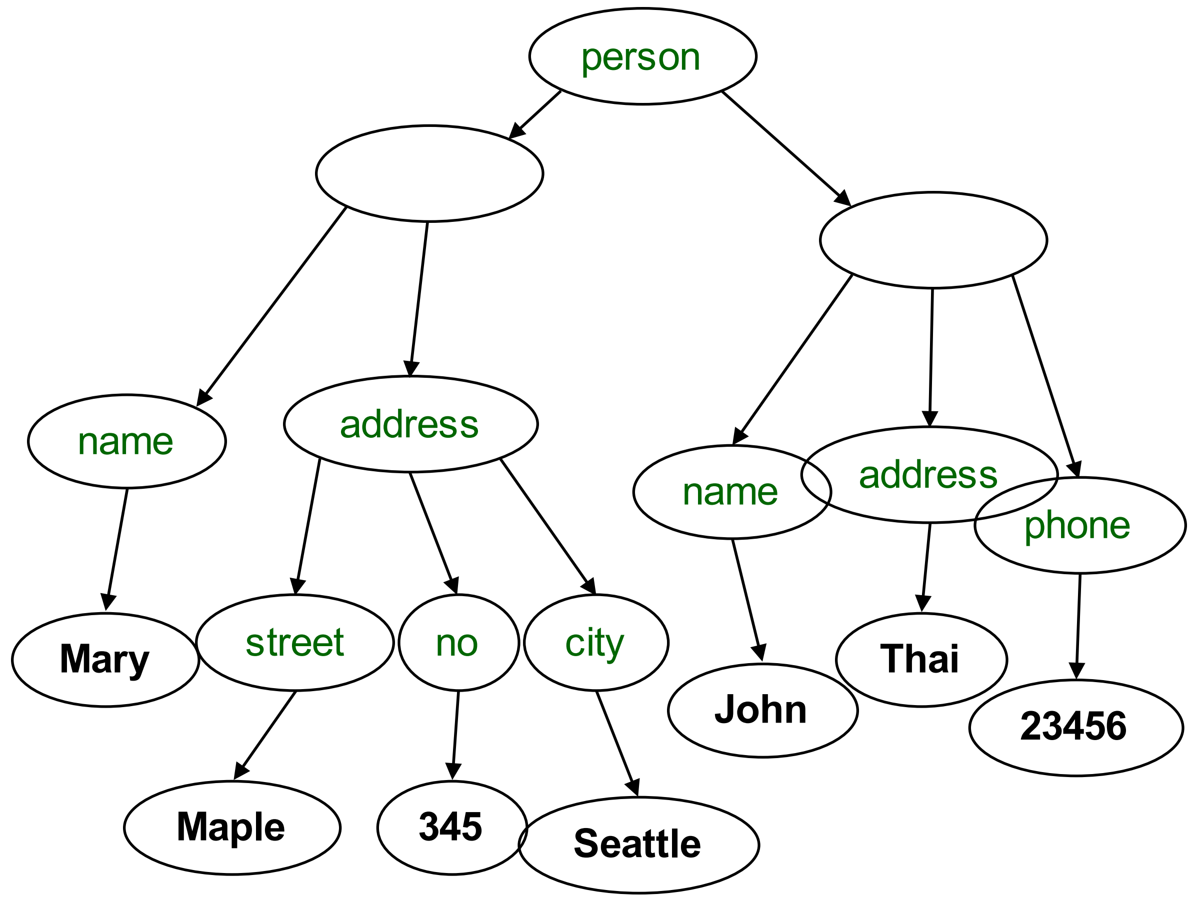
- JavaScript Object Notation = lightweight text-based open standard designed for human-readable data interchange. Interfaces in C, C++, Java, Python, Perl, etc.

- The filename extension is .json.

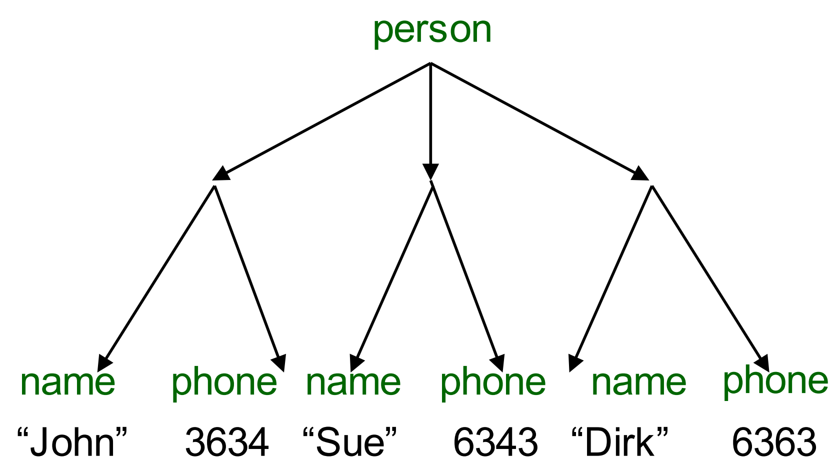
We will emphasize JSon as semi-structured data

JSon Semantics: a Tree !

```
{"person":
  [ {"name": "Mary",
     "address":
       {"street":"Maple",
        "no":345,
        "city": "Seattle"}},
    {"name": "John",
     "address": "Thailand",
     "phone":2345678}}
  ]
}
```



Mapping Relational Data to JSon



Person

name	phone
John	3634
Sue	6343
Dirk	6363

```
{"person":
    [{"name": "John", "phone":3634},
     {"name": "Sue",  "phone":6343},
     {"name": "Dirk",  "phone":6383}
    ]
}
```

Asterix Data Model (ADM)

- Objects:
  - {“Name”: “Alice”, “age”: 40}
  - Fields must be distinct:
    {“Name”: “Alice”, “age”: 40, ~~“age”:50~~}

Can’t have repeated fields

- Arrays:
  - [1, 3, “Fred”, 2, 9]
  - Note: can be heterogeneous
- Multisets:
  - {{1, 3, “Fred”, 2, 9}}

Examples

Try these queries:

```
SELECT x.age FROM [{'name': 'Alice', 'age': ['30', '50']}] x;
```

```
SELECT x.age FROM {{ {'name': 'Alice', 'age': ['30', '50']} }} x;
```

Can only select from multi-set or array

```
-- error
SELECT x.age FROM {'name': 'Alice', 'age': ['30', '50']} x;
```

SQL++ Overview

```
SELECT ... FROM ... WHERE ... [GROUP BY ...]
```

```
{"mondial":
    {"country": [ country1, country2, …],
     "continent": […],
     "organization": […],
     …
     …
}
```

← world

Retrieve Everything

```
SELECT x.mondial FROM world x;
```

Answer

```
{"mondial":
    {"country": [ country1, country2, …],
     "continent": […],
     "organization": […],
     …
     …
}
```

Retrieve countries

```
{"mondial":
   {"country": [ country1, country2, …],
    "continent": […],
    "organization": […],
    …
    …
}
```

```
SELECT x.mondial.country FROM world x;
```

Answer

```
{"country": [ country1, country2, …],
```

```
{"mondial":
   {"country": [ country1, country2, …],
    "continent": […],
    "organization": […],
    ...
    ...
}
```

Retrieve countries, one by one

```
SELECT y as country FROM world x, x.mondial.country y;
```

Answer

country1
country2
...

Heterogeneous Collections

```
{"mondial":
   {"country": [ country1, country2, …],
    "continent": […],
    "organization": […],
    ...
    ...
}
```

```
SELECT z.name as province_name, u.name as city_name
FROM  world x, x.mondial.country y, y.province z,
        (CASE  WHEN z.city is missing THEN []
               WHEN is_array(z.city) THEN z.city
               ELSE  [z.city] END)  u
WHERE  y.name='Greece';
```

The problem:

```
...
"province": [ ...
   {"name": "Attiki",
    "city" : [ {"name": "Athens"...}, {"name": "Pireus"...}, ..]
     ...},
   {"name": "Ipiros",
    "city" : {"name": "Ioannia"...}
     ...},
```

Even better

Useful Functions

- is_array
- is_boolean
- is_number
- is_object
- is_string
- is_null
- is_missing
- is_unknown = is_null or is_missing

Useful Idioms

- Unnesting
- Nesting
- Group-by / aggregate
- Join
- Multi-value join

Basic Unnesting

- An array: [a, b, c]
- A nested array: arr = [[a, b], [], [b, c, d]]
- Unnest(arr) = [a, b, b, c, d]

```
SELECT y
FROM arr x, x y
```

Unnesting Specific Field

A nested collection

```
coll =
[{A:a1, F:[{B:b1},{B:b2}], G:[{C:c1}]},
 {A:a2, F:[{B:b3},{B:b4},{B:b5}], G:[ ]},
 {A:a3, F:[{B:b6}], G:[{C:c2},{C:c3}]}]
```

Unnesting Specific Field

A nested collection

```
coll =
[{A:a1, F:[{B:b1},{B:b2}], G:[{C:c1}]},
 {A:a2, F:[{B:b3},{B:b4},{B:b5}], G:[ ]},
 {A:a3, F:[{B:b6}], G:[{C:c2},{C:c3}]}]
```

Nested Relational Algebra

$Unnest_F(coll)$ =

```
[{A:a1, {B:b1}, G:[{C:c1}]},
 {A:a1, {B:b2}, G:[{C:c1}]},
 {A:a2, {B:b3}, G:[]},
 {A:a2, {B:b4}, G:[]},
 {A:a2, {B:b5}, G:[]},
 {A:a3, {B:b6}, G:[{C:c2},{C:c3}]}]
```

Unnesting Specific Field

A nested collection

coll =
[{A:a1, F:[{B:b1},{B:b2}], G:[{C:c1}]},
 {A:a2, F:[{B:b3},{B:b4},{B:b5}], G:[ ]},
 {A:a3, F:[{B:b6}], G:[{C:c2},{C:c3}]}]

Nested Relational Algebra

$\text{Unnest}_F(\text{coll})$ =
[{A:a1, {B:b1}, G:[{C:c1}]},
 {A:a1, {B:b2}, G:[{C:c1}]},
 {A:a2, {B:b3}, G:[]},
 {A:a2, {B:b4}, G:[]},
 {A:a2, {B:b5}, G:[]},
 {A:a3, {B:b6}, G:[{C:c2},{C:c3}]}]

SQL++

```
SELECT x.A, y.B, x.G
FROM  coll x, x.F y
```

Refers to relations defined on the left

Unnesting Specific Field

A nested collection

```
coll =
[{A:a1, F:[{B:b1},{B:b2}], G:[{C:c1}]},
 {A:a2, F:[{B:b3},{B:b4},{B:b5}], G:[ ]},
 {A:a3, F:[{B:b6}], G:[{C:c2},{C:c3}]}]
```

Nested Relational Algebra

$Unnest_F(coll)$ =

```
[{A:a1, {B:b1}, G:[{C:c1}]},
 {A:a1, {B:b2}, G:[{C:c1}]},
 {A:a2, {B:b3}, G:[]},
 {A:a2, {B:b4}, G:[]},
 {A:a2, {B:b5}, G:[]},
 {A:a3, {B:b6}, G:[{C:c2},{C:c3}]}]
```

SQL++

```
SELECT x.A, y.B, x.G
FROM coll x, x.F y
```

=

```
SELECT x.A, y.B, x.G
FROM coll x
UNNEST x.F y
```

Unnesting Specific Field

A nested collection

coll =
[{A:a1, F:[{B:b1},{B:b2}], G:[{C:c1}]},
{A:a2, F:[{B:b3},{B:b4},{B:b5}], G:[ ]},
{A:a3, F:[{B:b6}], G:[{C:c2},{C:c3}]}]

Nested Relational Algebra

Unnest_F(coll) =
[{A:a1, {B:b1}, G:[{C:c1}]},
{A:a1, {B:b2}, G:[{C:c1}]},
{A:a2, {B:b3}, G:[]},
{A:a2, {B:b4}, G:[]},
{A:a2, {B:b5}, G:[]},
{A:a3, {B:b6}, G:[{C:c2},{C:c3}]}]

Unnest_G(coll) =
[{A:a1, F:[{B:b1},{B:b2}], C:c1},
{A:a3, F:[{B:b6}], C:c2},
{A:a3, F:[{B:b6}], C:c3}]

SQL++

```
SELECT x.A, y.B, x.G
FROM coll x, x.F y
```

Unnesting Specific Field

A nested collection

```
coll =
[{A:a1, F:[{B:b1},{B:b2}], G:[{C:c1}]},
 {A:a2, F:[{B:b3},{B:b4},{B:b5}], G:[ ]},
 {A:a3, F:[{B:b6}], G:[{C:c2},{C:c3}]}]
```

Nested Relational Algebra

$Unnest_F$(coll) =

```
[{A:a1, {B:b1}, G:[{C:c1}]},
 {A:a1, {B:b2}, G:[{C:c1}]},
 {A:a2, {B:b3}, G:[]},
 {A:a2, {B:b4}, G:[]},
 {A:a2, {B:b5}, G:[]},
 {A:a3, {B:b6}, G:[{C:c2},{C:c3}]}]
```

$Unnest_G$(coll) =

```
[{A:a1, F:[{B:b1},{B:b2}], C:c1},
 {A:a3, F:[{B:b6}], C:c2},
 {A:a3, F:[{B:b6}], C:c3}]
```

```
SELECT x.A, y.B, x.G
FROM coll x, x.F y
```

SQL++

```
SELECT x.A, x.F, z.C
FROM coll x, x.G z
```

Nesting (like group-by)

A flat collection

```
coll =
[{A:a1, B:b1}, {A:a1, B:b2}, {A:a2, B:b1}]
```

Nesting (like group-by)

A flat collection

coll =
[{A:a1, B:b1}, {A:a1, B:b2}, {A:a2, B:b1}]

Nested Relational Algebra

Nest$_A$(coll) =
[{A:a1, GRP:[{B:b1},{B:b2}]}
[{A:a2, GRP:[{B:b2}]}]

b1

Nesting (like group-by)

A flat collection

coll =
[{A:a1, B:b1}, {A:a1, B:b2}, {A:a2, B:b1}]

Nested Relational Algebra

$Nest_A$(coll) =
[{A:a1, GRP:[{B:b1},{B:b2}]}
[{A:a2, GRP:[{B:b2}]}]

$Nest_B$(coll) =
[{B:b1, GRP:[{A:a1},{A:a2}]},
{B:b2, GRP:[{A:a1}]}]

Nesting (like group-by)

A flat collection

coll =
[{A:a1, B:b1}, {A:a1, B:b2}, {A:a2, B:b1}]

Nested Relational Algebra

$Nest_A$(coll) =
[{A:a1, GRP:[{B:b1},{B:b2}]}
[{A:a2, GRP:[{B:b2}]}]

$Nest_B$(coll) =
[{B:b1, GRP:[{A:a1},{A:a2}]},
{B:b2, GRP:[{A:a1}]}]

```
SELECT DISTINCT x.A,
    (SELECT y.B FROM coll y WHERE x.A = y.A) as GRP
FROM coll x
```

Nesting (like group-by)

A flat collection

coll =
[{A:a1, B:b1}, {A:a1, B:b2}, {A:a2, B:b1}]

Nested Relational Algebra

$Nest_A$(coll) =
[{A:a1, GRP:[{B:b1},{B:b2}]}
[{A:a2, GRP:[{B:b2}]}]

$Nest_B$(coll) =
[{B:b1, GRP:[{A:a1},{A:a2}]},
{B:b2, GRP:[{A:a1}]}]

```
SELECT DISTINCT x.A,
    (SELECT y.B FROM coll y WHERE x.A = y.A) as GRP
FROM coll x
```

```
SELECT DISTINCT x.A, g as GRP
FROM coll x
LET g = (SELECT y.B FROM coll y WHERE x.A = y.A)
```

Group-by / Aggregate

A nested collection

```
coll =
[{A:a1, F:[{B:b1},{B:b2}], G:[{C:c1}]},
 {A:a2, F:[{B:b3},{B:b4},{B:b5}], G:[ ]},
 {A:a3, F:[{B:b6}], G:[{C:c2},{C:c3}]}]
```

Count the number of elements in the F collection

Group-by / Aggregate

A nested collection

```
coll =
[{A:a1, F:[{B:b1},{B:b2}], G:[{C:c1}]},
 {A:a2, F:[{B:b3},{B:b4},{B:b5}], G:[ ]},
 {A:a3, F:[{B:b6}], G:[{C:c2},{C:c3}]}]
```

Count the number of elements in the F collection

```
SELECT x.A, COLL_COUNT(x.F) as cnt
FROM coll x
```

Group-by / Aggregate

A nested collection

```
coll =
[{A:a1, F:[{B:b1},{B:b2}], G:[{C:c1}]},
 {A:a2, F:[{B:b3},{B:b4},{B:b5}], G:[ ]},
 {A:a3, F:[{B:b6}], G:[{C:c2},{C:c3}]}]
```

Count the number of elements in the F collection

```
SELECT x.A, COLL_COUNT(x.F) as cnt
FROM coll x
```

```
SELECT x.A, COUNT(*) as cnt
FROM coll x, x.F y
GROUP BY x.A
```

These are NOT equivalent!
(Why?)

Group-by / Aggregate

Function	NULL	MISSING	Empty Collection
COLL_COUNT	counted	counted	0
COLL_SUM	returns NULL	returns NULL	returns NULL
COLL_MAX	returns NULL	returns NULL	returns NULL
COLL_MIN	returns NULL	returns NULL	returns NULL
COLL_AVG	returns NULL	returns NULL	returns NULL
ARRAY_COUNT	not counted	not counted	0
ARRAY_SUM	ignores NULL	ignores NULL	returns NULL
ARRAY_MAX	ignores NULL	ignores NULL	returns NULL
ARRAY_MIN	ignores NULL	ignores NULL	returns NULL
ARRAY_AVG	ignores NULL	ignores NULL	returns NULL

Lesson: Read the *$@# manual!!

Join

Two flat collection

```
coll1 = [{A:a1, B:b1}, {A:a1, B:b2}, {A:a2, B:b1}]
coll2 = [{B:b1,C:c1}, {B:b1,C:c2}, {B:b3,C:c3}]
```

```
SELECT x.A, x.B, y.C
FROM coll1 x, coll2 y
WHERE x.B = y.B
```

Behind the Scences

Query Processing on NFNF data:

- Option 1: give up on query plans, use standard java/python-like execution
- Option 2: represent the data as a collection of flat tables, convert SQL++ to a standard relational query plan

Flattening SQL++ Queries

A nested collection

```
coll =
[{A:a1, F:[{B:b1},{B:b2}], G:[{C:c1}]},
 {A:a2, F:[{B:b3},{B:b4},{B:b5}], G:[ ]},
 {A:a1, F:[{B:b6}], G:[{C:c2},{C:c3}]}]
```

Flattening SQL++ Queries

A nested collection

```
coll =
[{A:a1, F:[{B:b1},{B:b2}], G:[{C:c1}]},
 {A:a2, F:[{B:b3},{B:b4},{B:b5}], G:[ ]},
 {A:a1, F:[{B:b6}], G:[{C:c2},{C:c3}]}]
```

Flat Representation

coll:

id	A
1	a1
2	a2
3	a1

F

parent	B
1	b1
1	b2
2	b3
2	b4
2	b5
3	b6

G

parent	C
1	c1
3	c2
3	c3

Flattening SQL++ Queries

A nested collection

```
coll =
[{A:a1, F:[{B:b1},{B:b2}], G:[{C:c1}]},
 {A:a2, F:[{B:b3},{B:b4},{B:b5}], G:[ ]},
 {A:a1, F:[{B:b6}], G:[{C:c2},{C:c3}]}]
```

SQL++

```
SELECT x.A, y.B
FROM coll x, x.F y
WHERE x.A = 'a1'
```

Flat Representation

coll:

id	A
1	a1
2	a2
3	a1

F

parent	B
1	b1
1	b2
2	b3
2	b4
2	b5
3	b6

G

parent	C
1	c1
3	c2
3	c3

SQL

Flattening SQL++ Queries

A nested collection

```
coll =
[{A:a1, F:[{B:b1},{B:b2}], G:[{C:c1}]},
 {A:a2, F:[{B:b3},{B:b4},{B:b5}], G:[ ]},
 {A:a1, F:[{B:b6}], G:[{C:c2},{C:c3}]}]
```

SQL++

```
SELECT x.A, y.B
FROM coll x, x.F y
WHERE x.A = 'a1'
```

Flat Representation

coll:

id	A
1	a1
2	a2
3	a1

F

parent	B
1	b1
1	b2
2	b3
2	b4
2	b5
3	b6

G

parent	C
1	c1
3	c2
3	c3

SQL

```
SELECT x.A, y.B
FROM coll x, F y
WHERE x.id = y.parent and x.A = 'a1'
```

Flattening SQL++ Queries

A nested collection

```
coll =
[{A:a1, F:[{B:b1},{B:b2}], G:[{C:c1}]},
 {A:a2, F:[{B:b3},{B:b4},{B:b5}], G:[ ]},
 {A:a1, F:[{B:b6}], G:[{C:c2},{C:c3}]}]
```

SQL++

```
SELECT x.A, y.B
FROM coll x, x.F y
WHERE x.A = 'a1'
```

```
SELECT x.A, y.B
FROM coll x, x.F y, x.G z
WHERE y.B = z.C
```

Flat Representation

coll:

id	A
1	a1
2	a2
3	a1

F

parent	B
1	b1
1	b2
2	b3
2	b4
2	b5
3	b6

G

parent	C
1	c1
3	c2
3	c3

SQL

```
SELECT x.A, y.B
FROM coll x, F y
WHERE x.id = y.parent and x.A = 'a1'
```

Flattening SQL++ Queries

A nested collection

```
coll =
[{A:a1, F:[{B:b1},{B:b2}], G:[{C:c1}]},
 {A:a2, F:[{B:b3},{B:b4},{B:b5}], G:[ ]},
 {A:a1, F:[{B:b6}], G:[{C:c2},{C:c3}]}]
```

SQL++

```
SELECT x.A, y.B
FROM coll x, x.F y
WHERE x.A = 'a1'
```

```
SELECT x.A, y.B
FROM coll x, x.F y, x.G z
WHERE y.B = z.C
```

Flat Representation

coll:

id	A
1	a1
2	a2
3	a1

F

parent	B
1	b1
1	b2
2	b3
2	b4
2	b5
3	b6

G

parent	C
1	c1
3	c2
3	c3

SQL

```
SELECT x.A, y.B
FROM coll x, F y
WHERE x.id = y.parent and x.A = 'a1'
```

```
SELECT x.A, y.B
FROM coll x, F y, G z
WHERE x.id = y.parent and x.id = z.parent
          and y.B = z.C
```

Conclusion

- Semistructured data best suited for *data exchange*
- For quick, ad-hoc data analysis, use a native query language: SQL++, or AQL, or XQuery
  - Modern, advanced query processors like AsterixDB / SQL++ can process semistructured data as efficiently as RDBMS
- For long term data analysis: spend the time and effort to normalize it, then store in a RDBMS

Query Execution and Optimization

Class overview

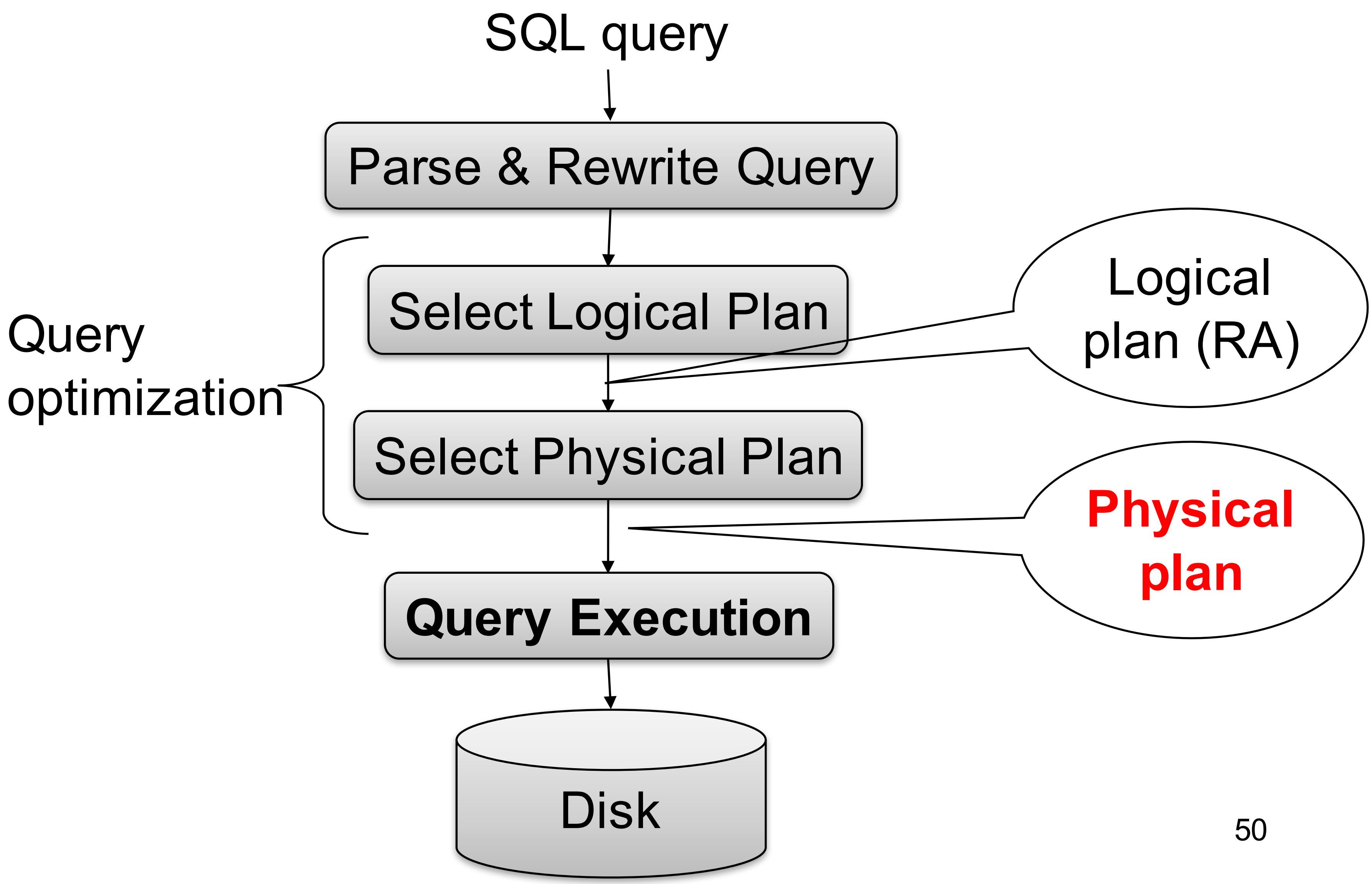
- Data models
  - Relational: SQL, RA, and Datalog
  - NoSQL: SQL++
- RDBMS internals
  - Query processing and optimization
  - Physical design
- Parallel query processing
  - Spark and Hadoop
- Conceptual design
  - E/R diagrams
  - Schema normalization
- Transactions
  - Locking and schedules
  - Writing DB applications

Data models

Query Processing

Using DBMS

Query Evaluation Steps Review



Logical vs Physical Plans

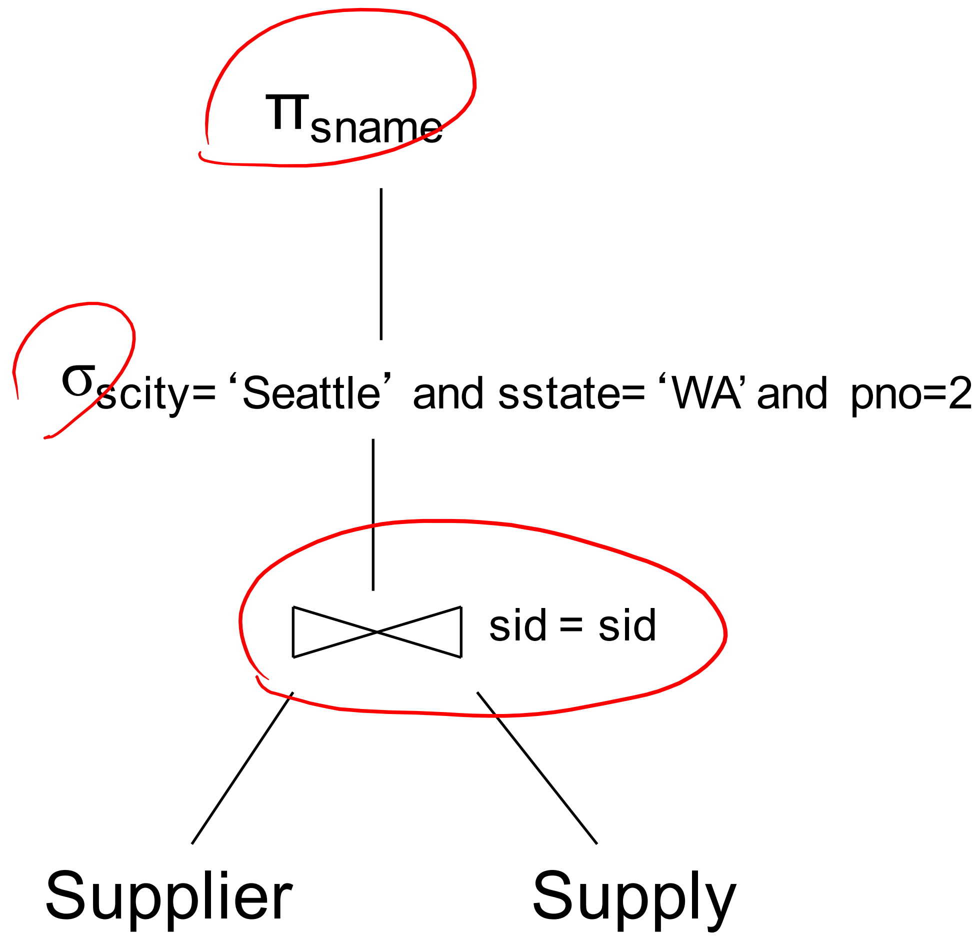
- Logical plans:
  - Created by the parser from the input SQL text
  - Expressed as a relational algebra tree
  - Each SQL query has many possible logical plans

- Physical plans:
  - Goal is to choose an efficient implementation for each operator in the RA tree
  - Each logical plan has many possible physical plans

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

Review: Relational Algebra

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

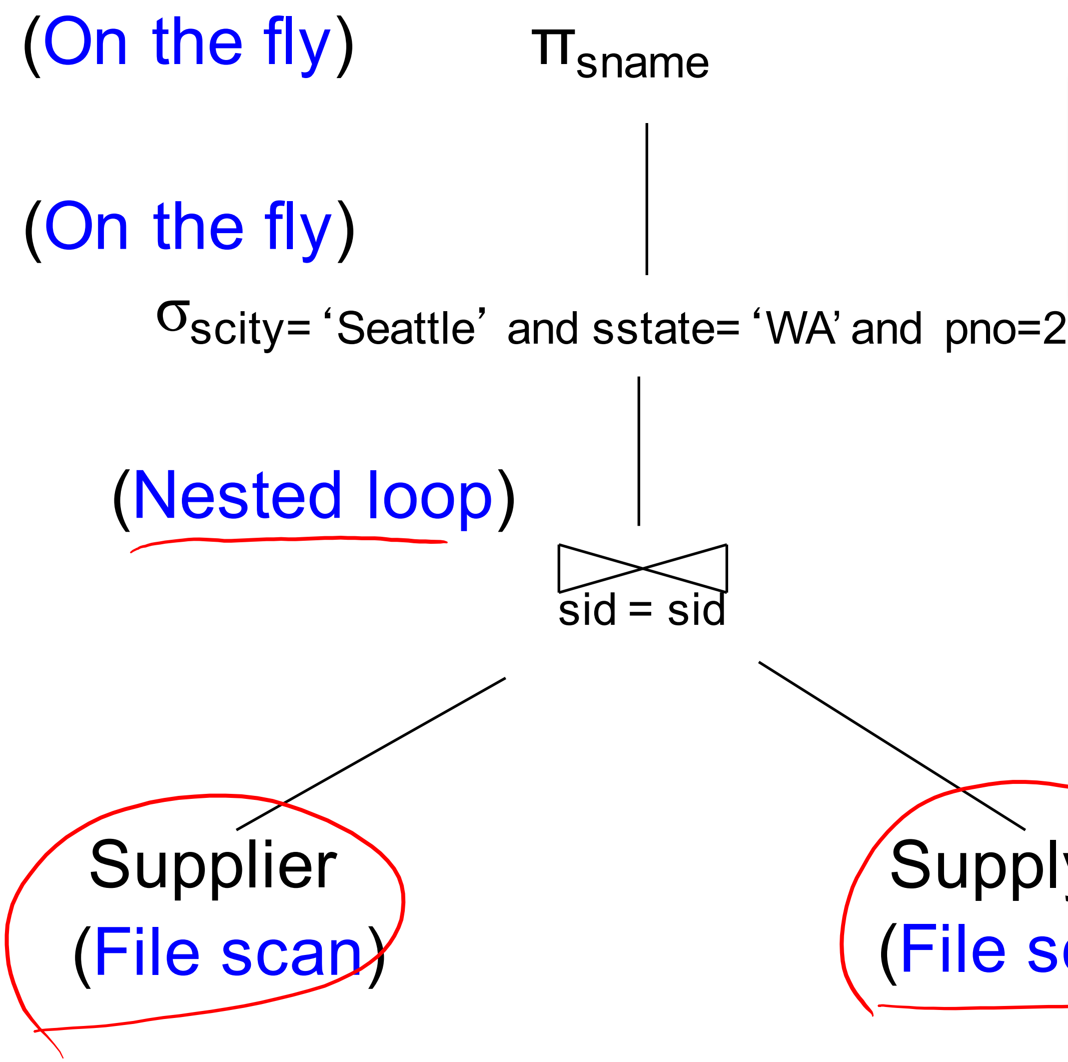


Relational algebra expression is also called the "logical query plan"

Supplier(sid, sname, scity, sstate)

Supply(sid, pno, quantity)

Physical Query Plan 1

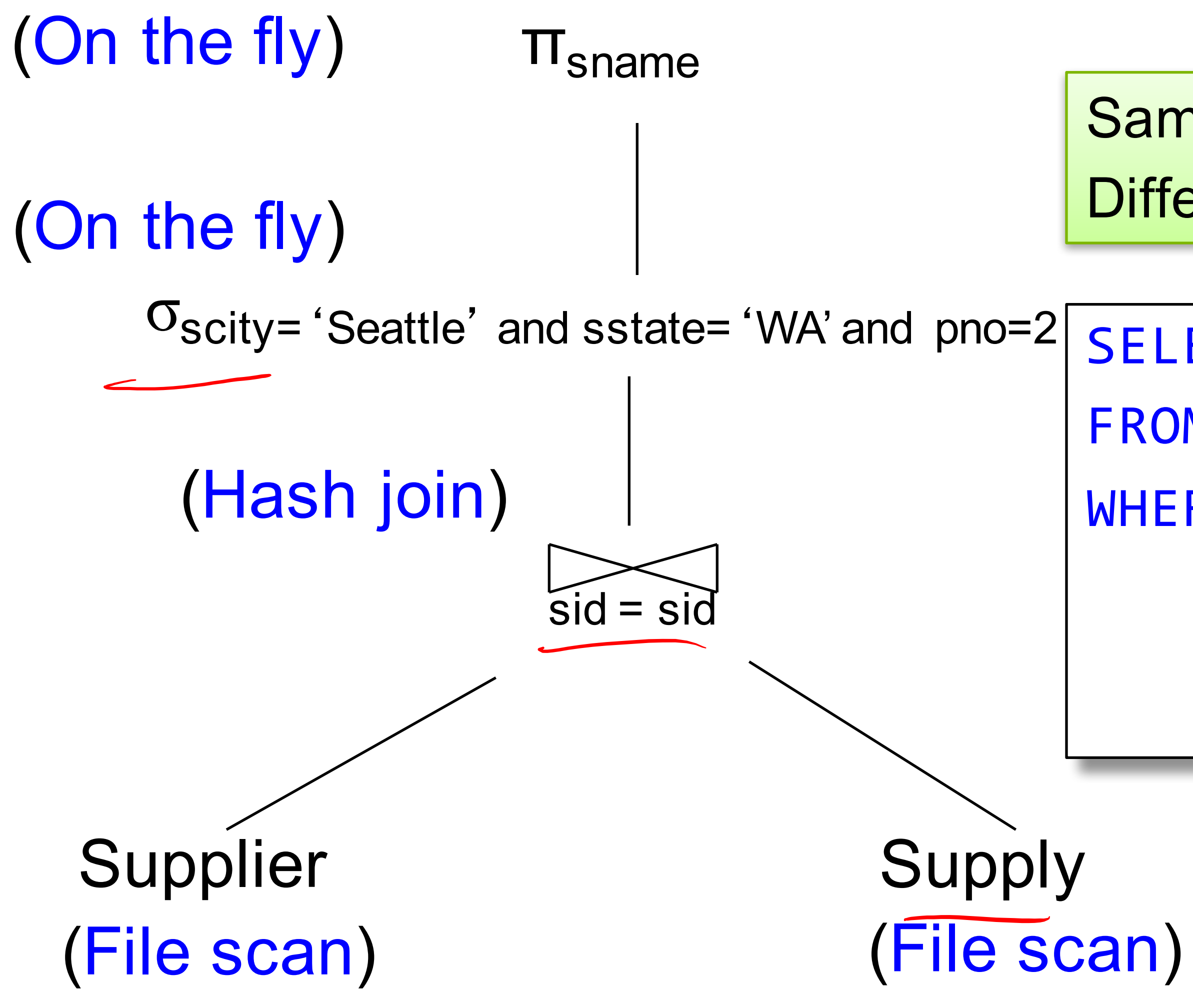


A physical query plan is a logical query plan annotated with physical implementation details

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

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

Physical Query Plan 2



Same logical query plan
Different physical plan

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

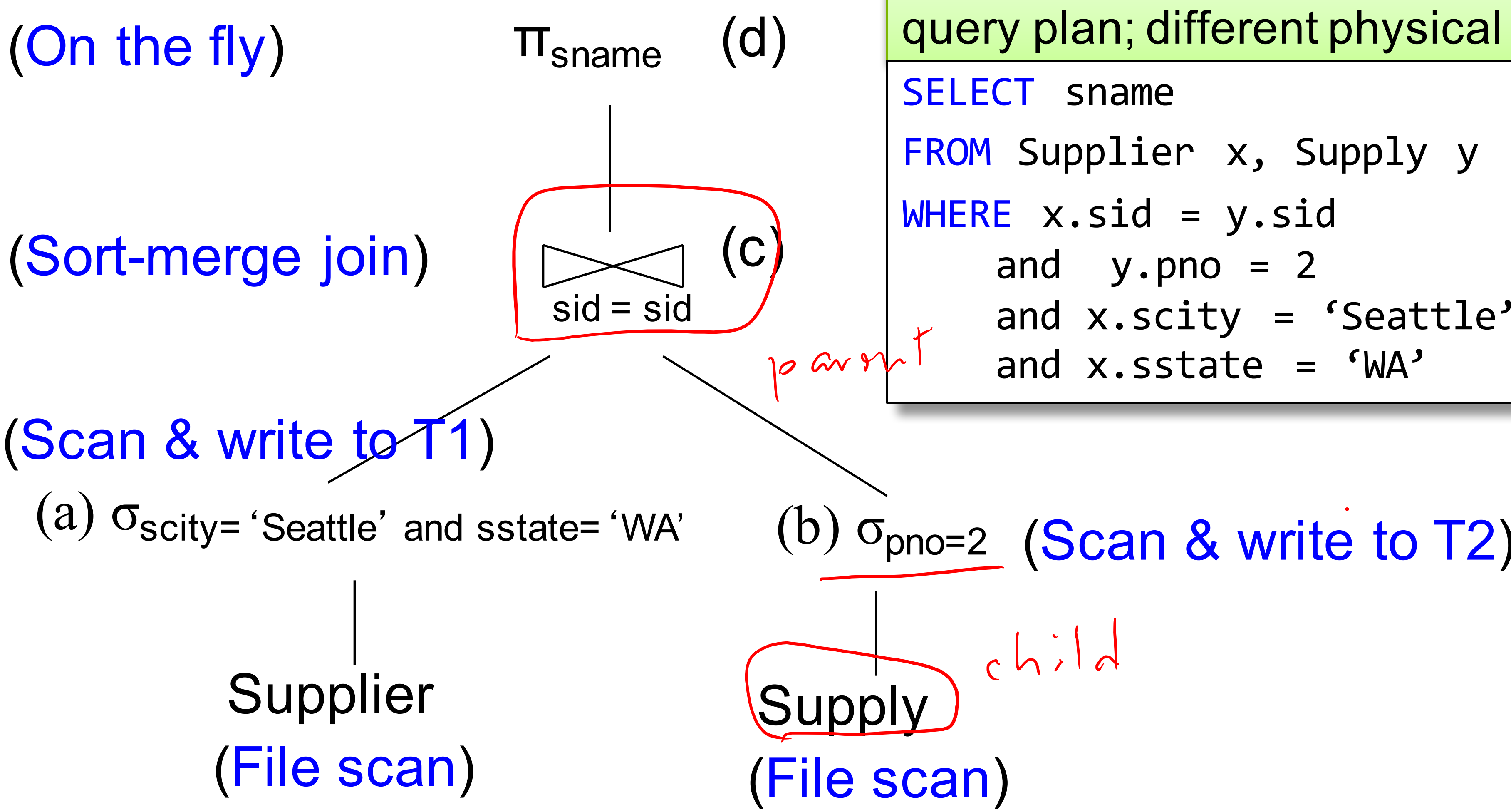
Supplier(sid, sname, scity, sstate)

Supply(sid, pno, quantity)

Physical Query Plan 3

Different but equivalent logical query plan; different physical plan

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



Query Optimization Problem

- For each SQL query… many logical plans
- For each logical plan… many physical plans
- Next: we will discuss physical operators; *how exactly are query executed?*

Query Execution

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;
  }
}
```

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;
  }
  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 child) {
    this.p = p; this.child = child;
  }
  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;
  }
  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

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;
  }
  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

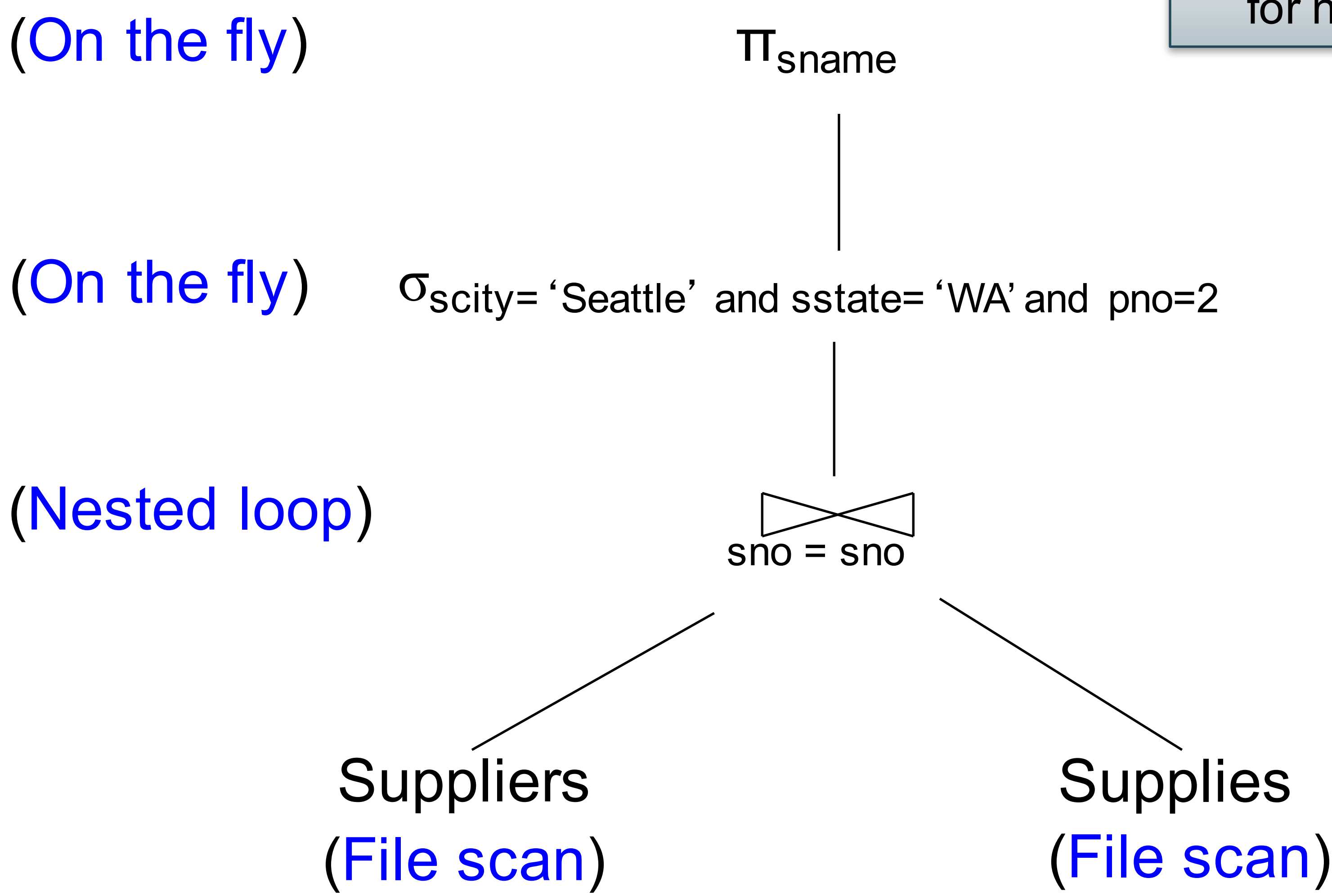
```
Operator q = parse("SELECT ...");
q = optimize(q);

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

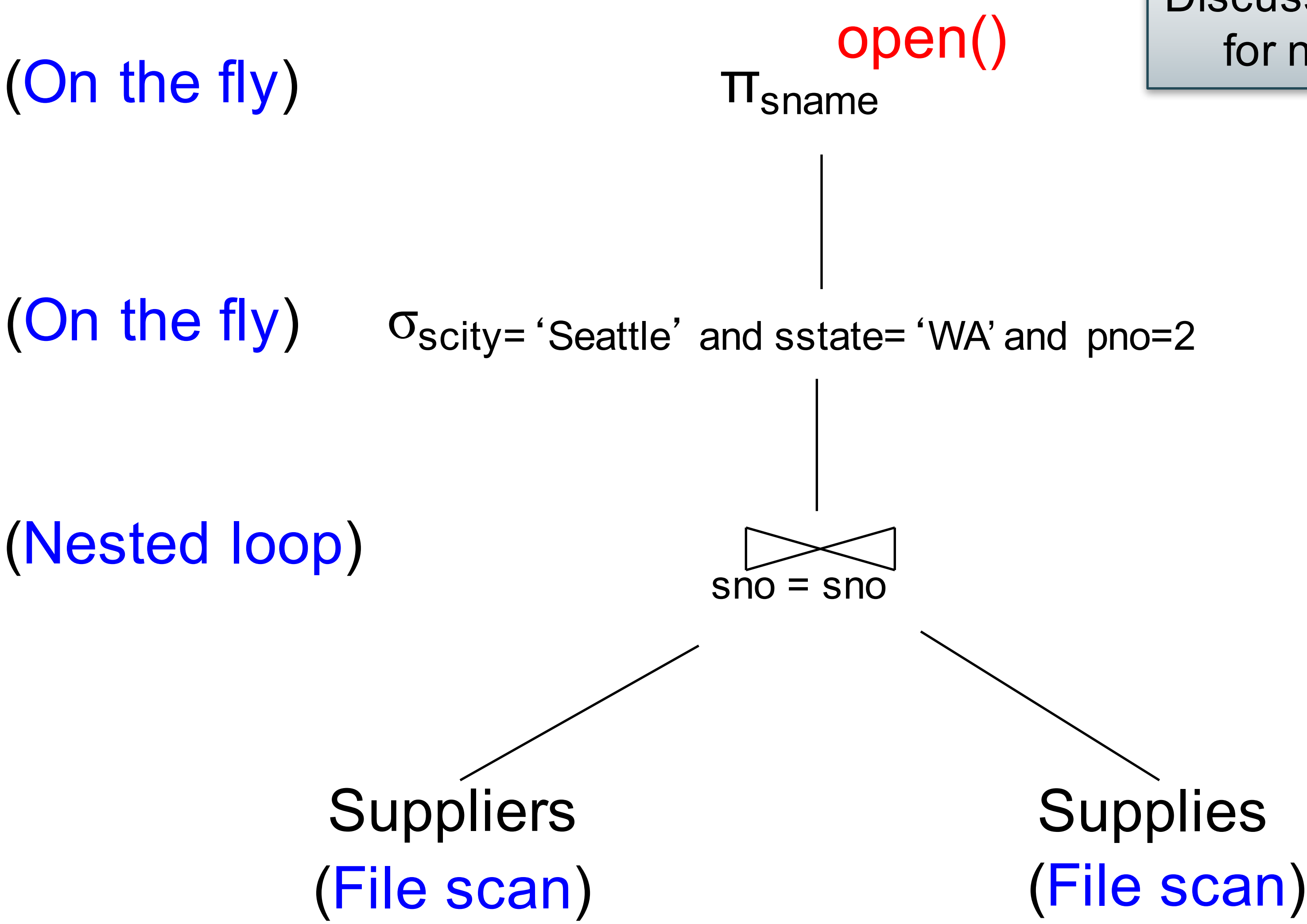
Discuss: open/next/close for nested loop join



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

Pipelining

Discuss: open/next/close for nested loop join

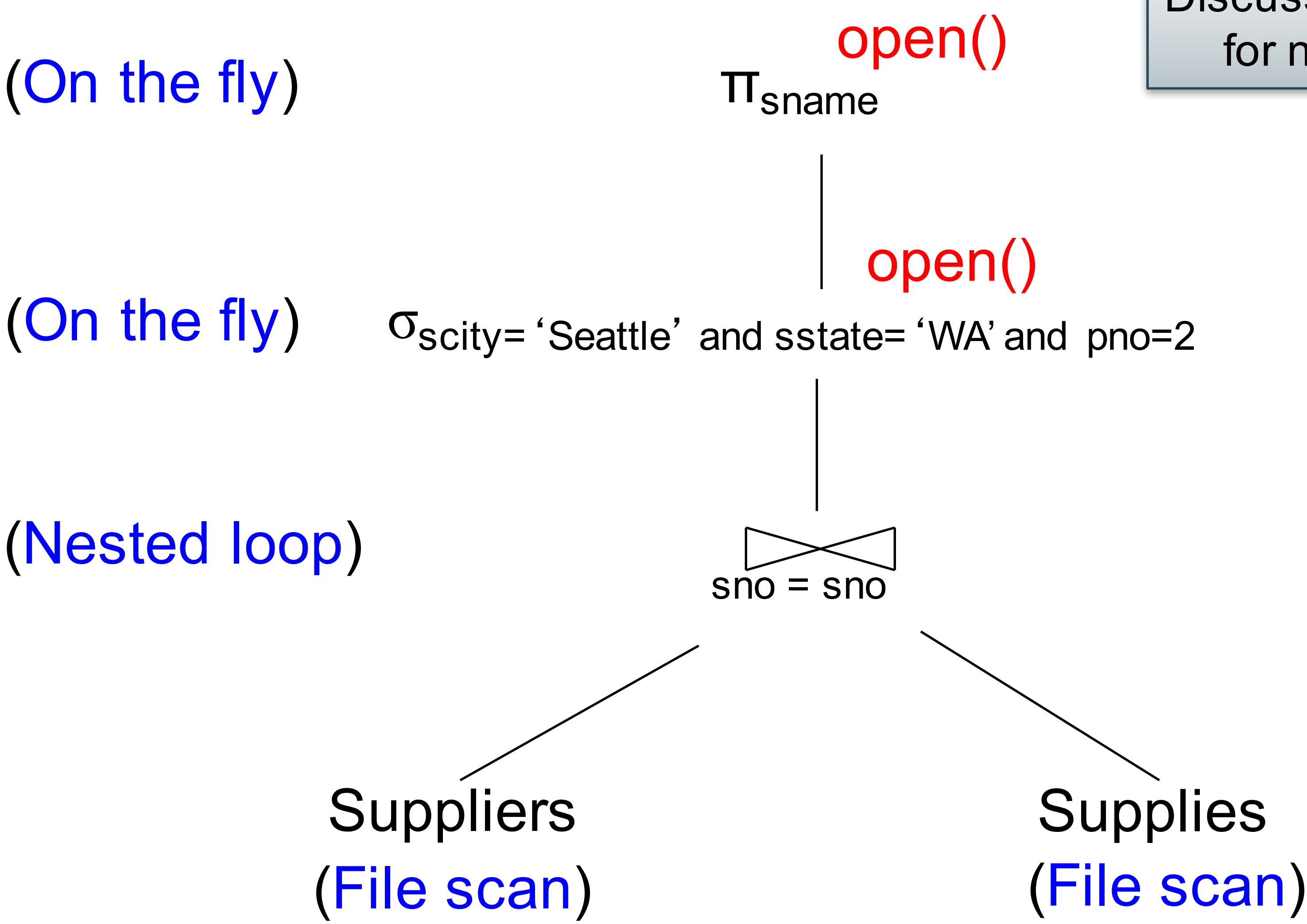


Supplier(sid, sname, scity, sstate)

Supply(sid, pno, quantity)

Pipelining

Discuss: open/next/close for nested loop join

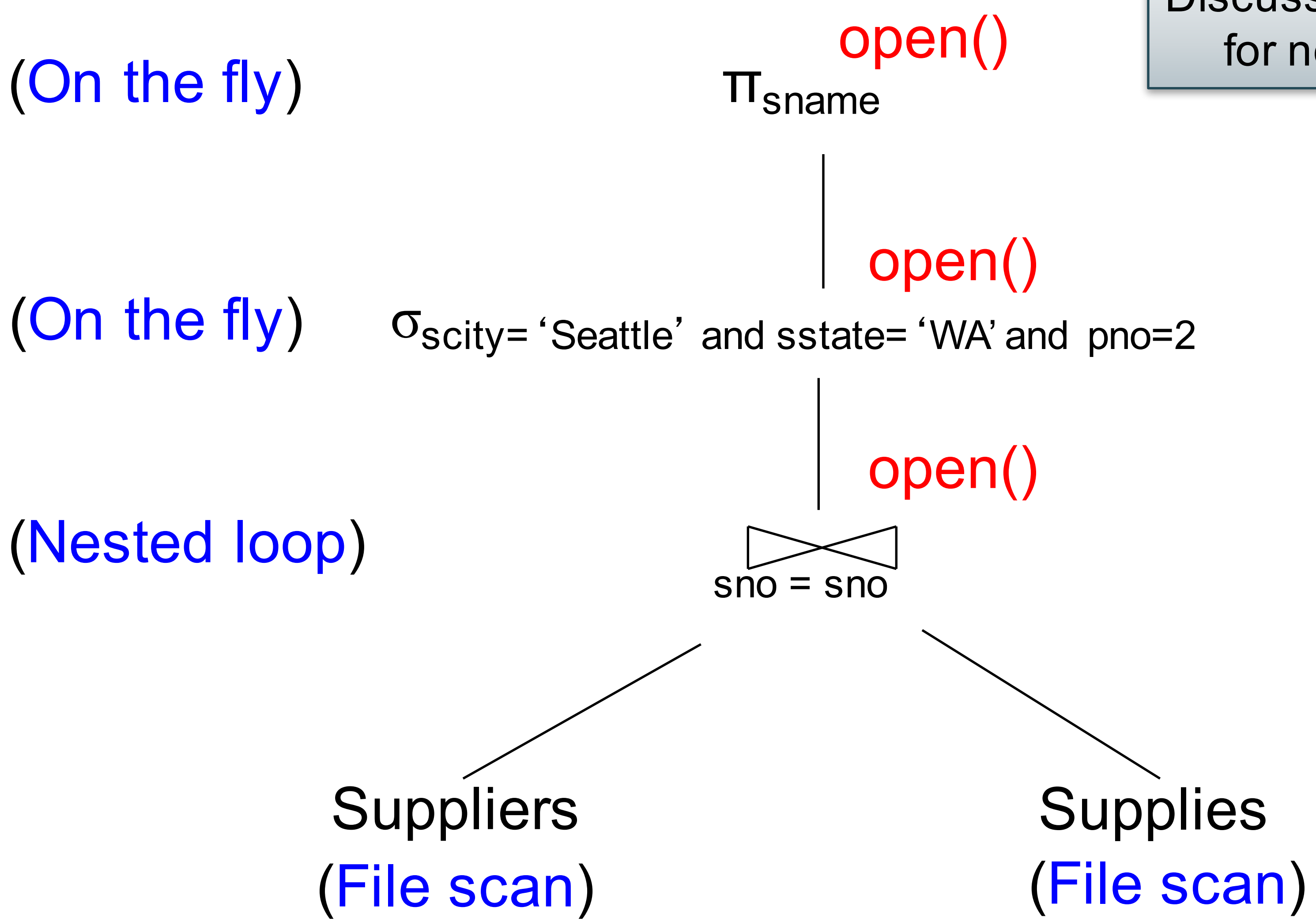


Supplier(<u>sid</u>, sname, scity, sstate)

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

Pipelining

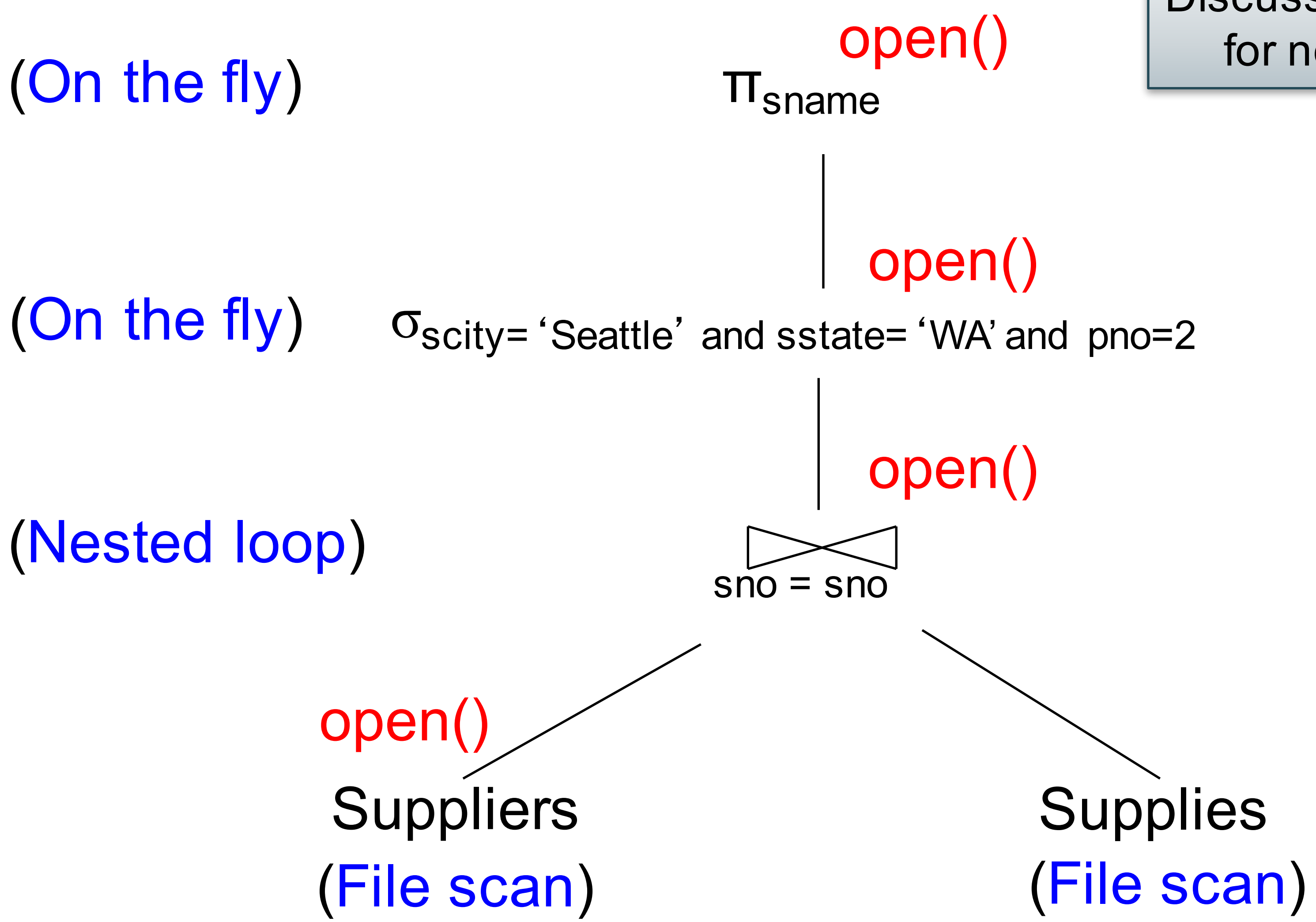
Discuss: open/next/close for nested loop join



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

Pipelining

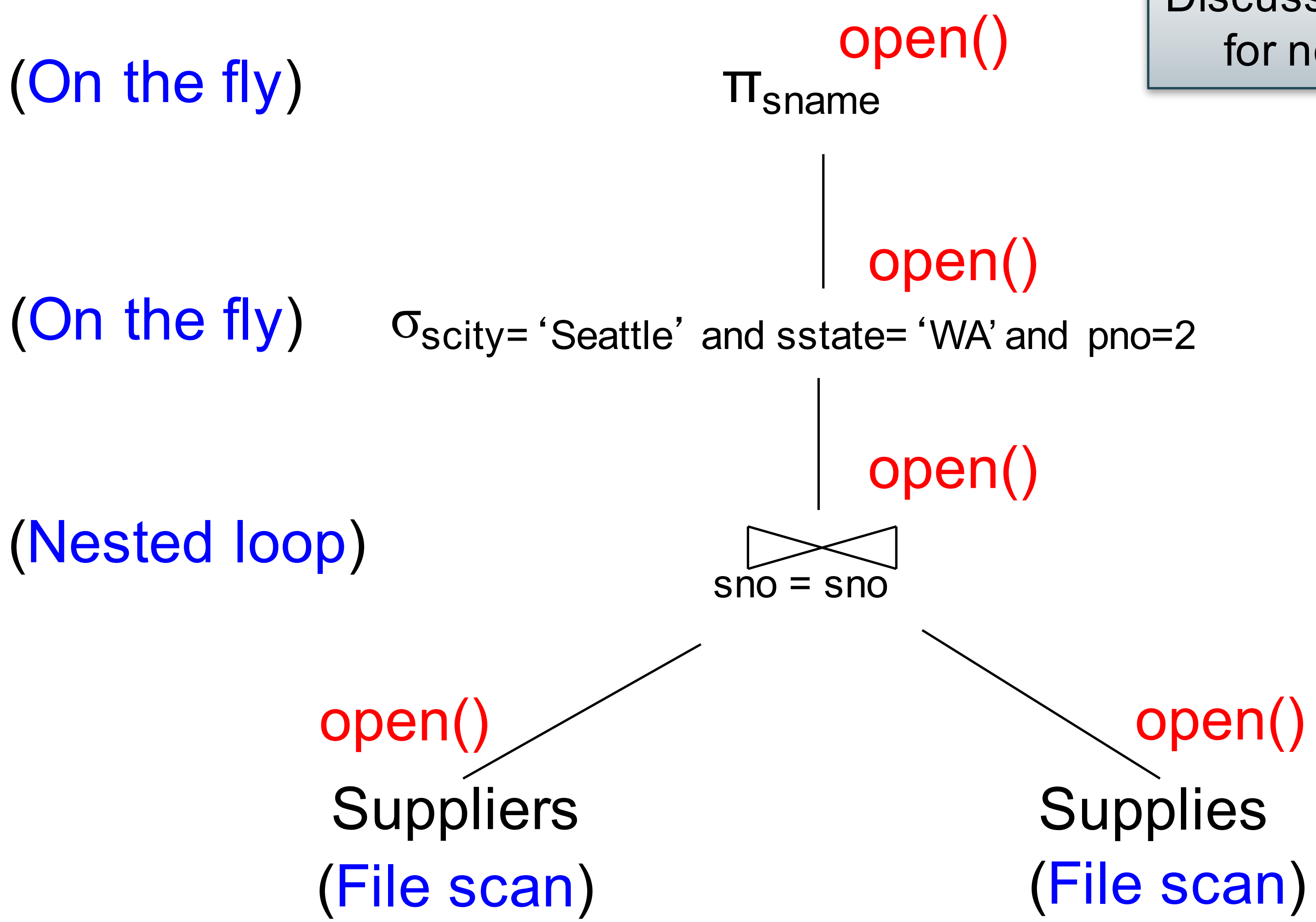
Discuss: open/next/close for nested loop join



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

Pipelining

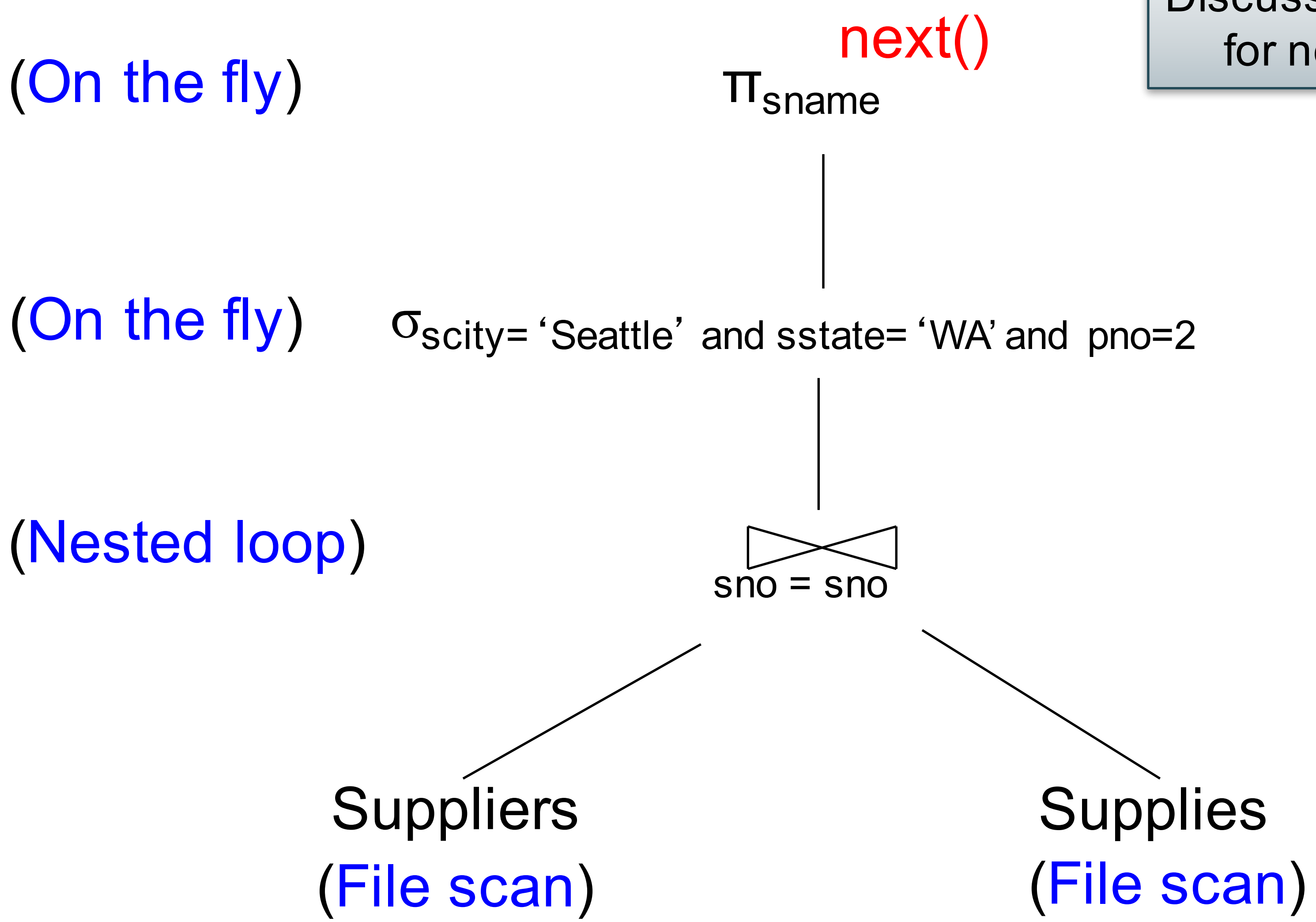
Discuss: open/next/close for nested loop join



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

Pipelining

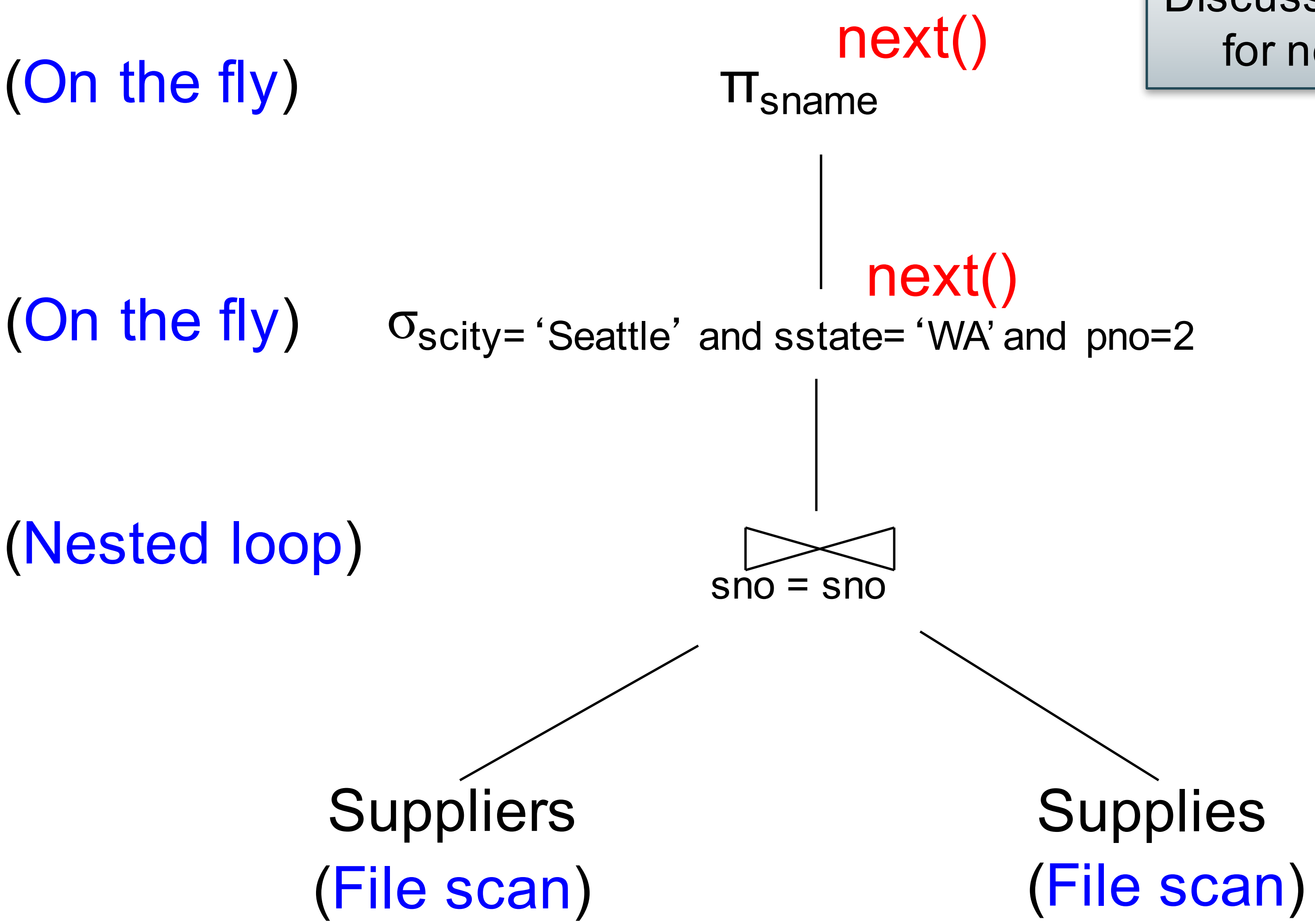
Discuss: open/next/close for nested loop join



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

Pipelining

Discuss: open/next/close for nested loop join

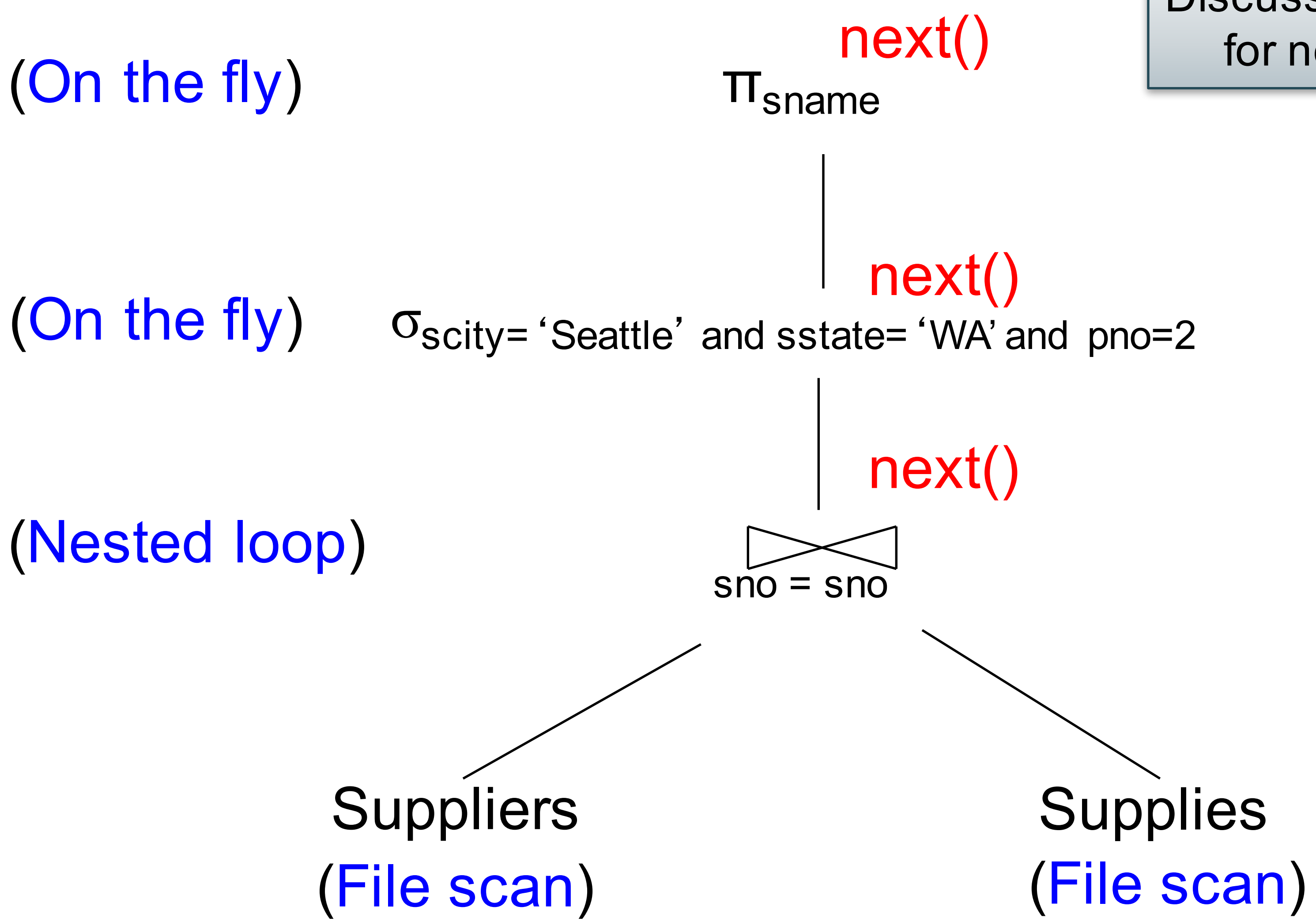


Supplier(sid, sname, scity, sstate)

Supply(sid, pno, quantity)

Pipelining

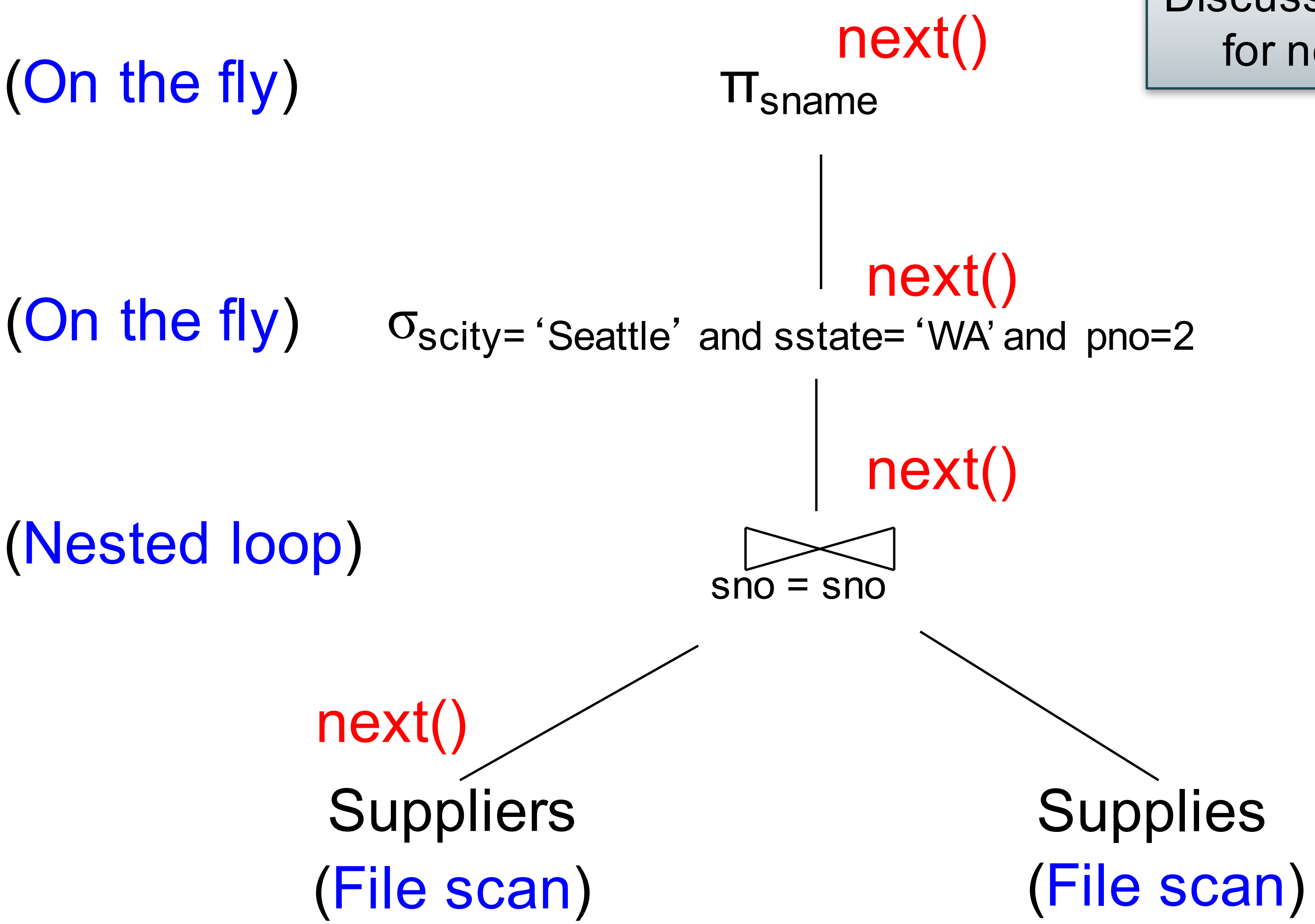
Discuss: open/next/close for nested loop join



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

Pipelining

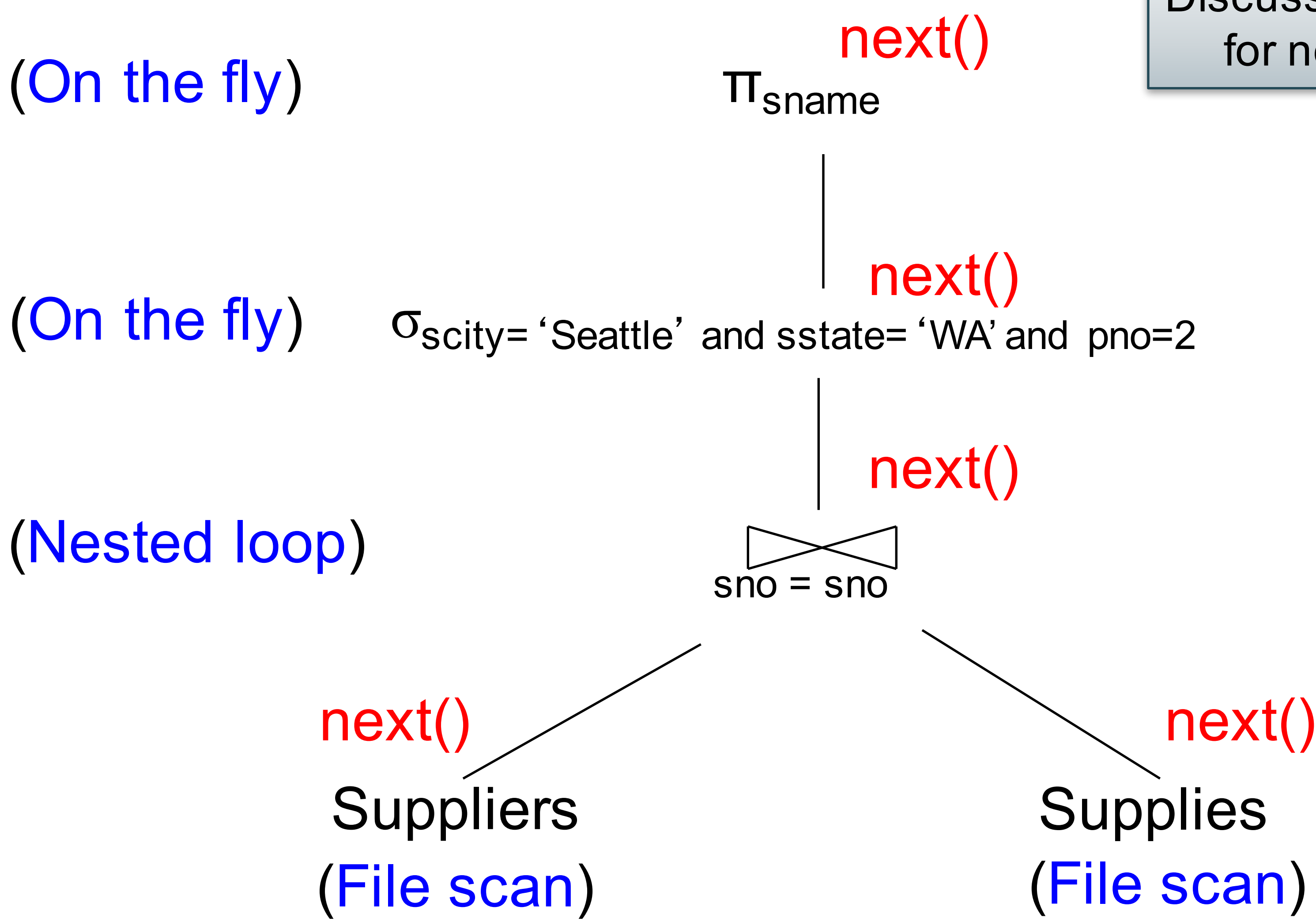
Discuss: open/next/close for nested loop join



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

Pipelining

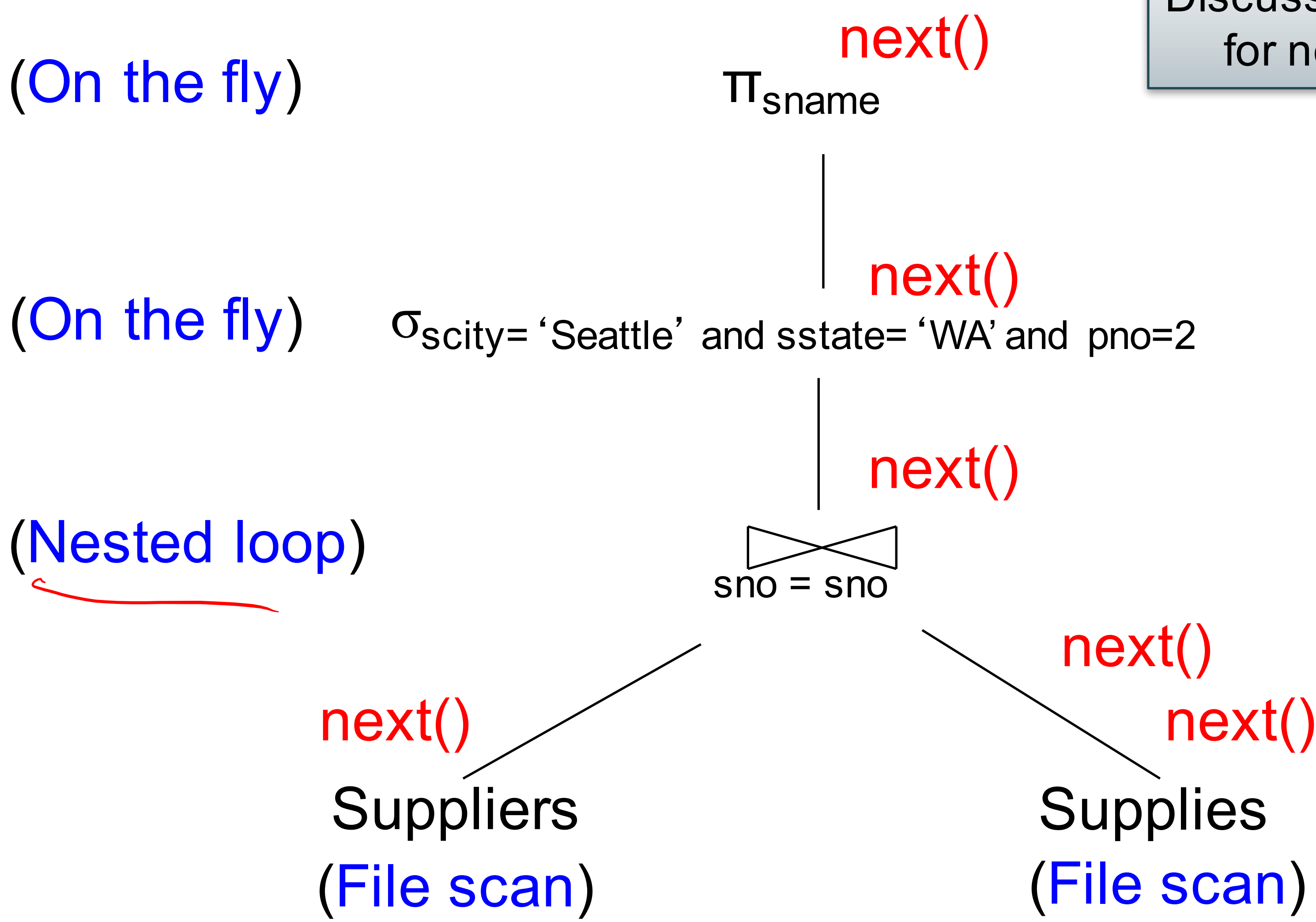
> Discuss: open/next/close for nested loop join



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

Pipelining

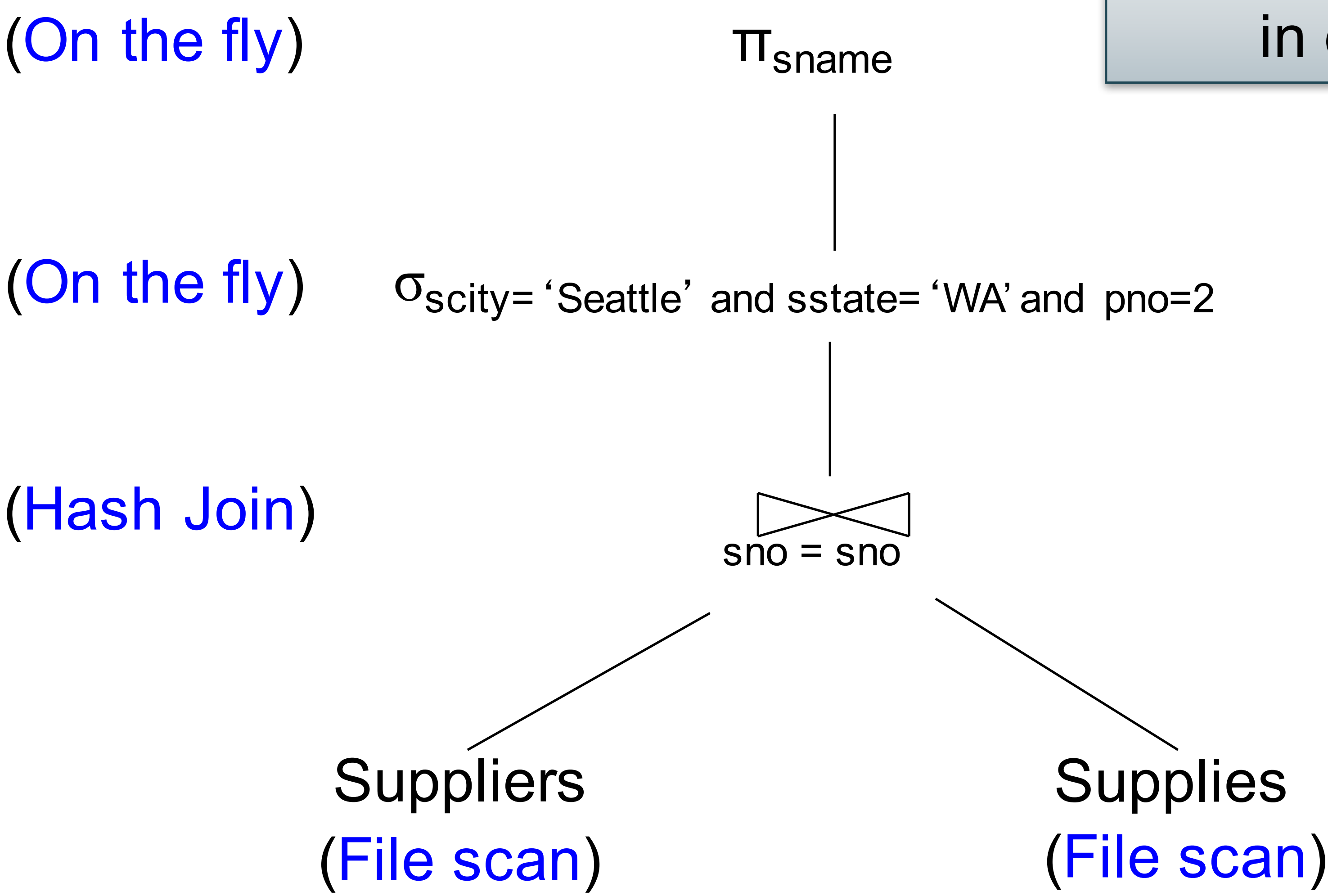
Discuss: open/next/close for nested loop join



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

Pipelining

Discuss hash-join in class

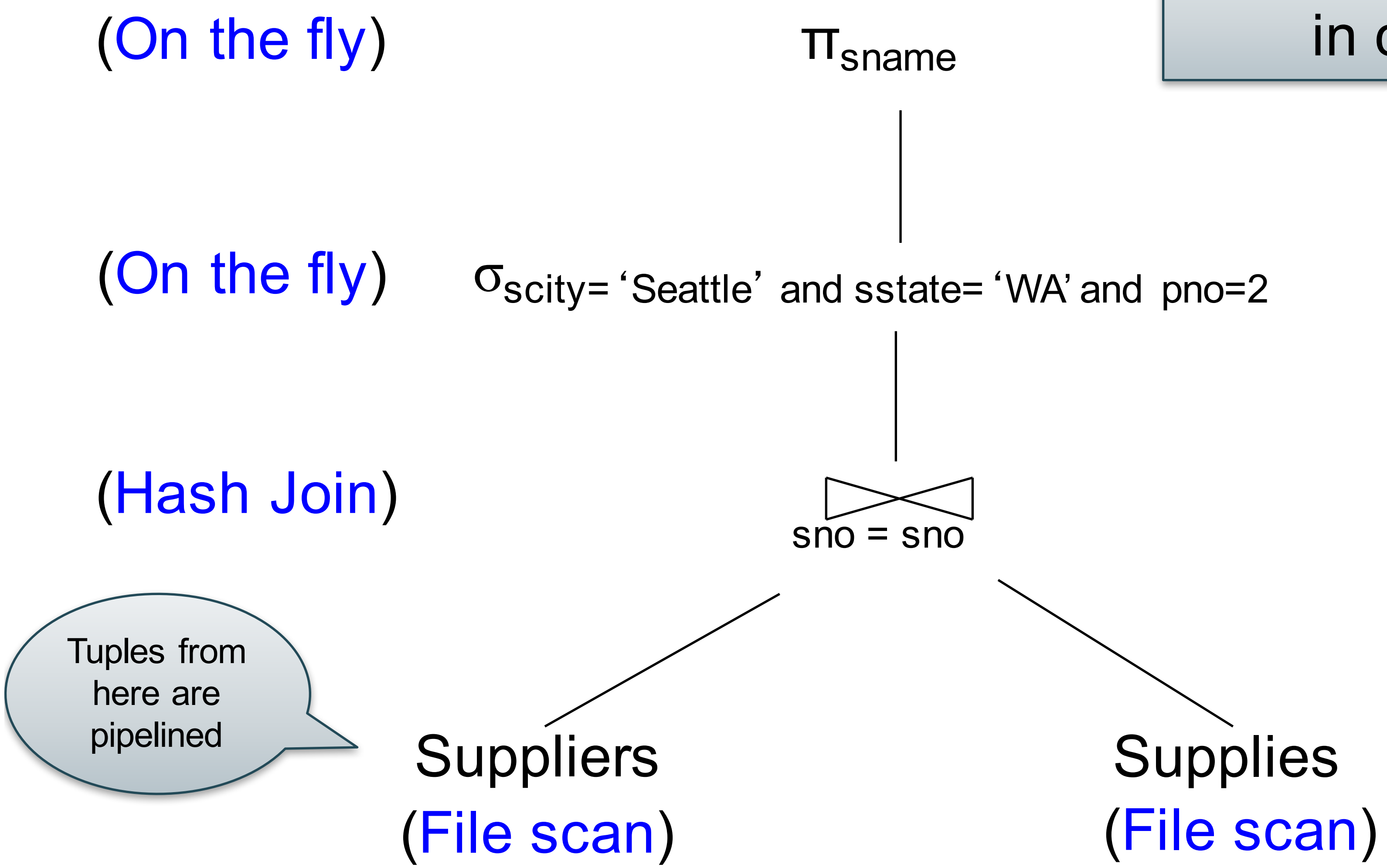


`Supplier(sid, sname, scity, sstate)`

`Supply(sid, pno, quantity)`

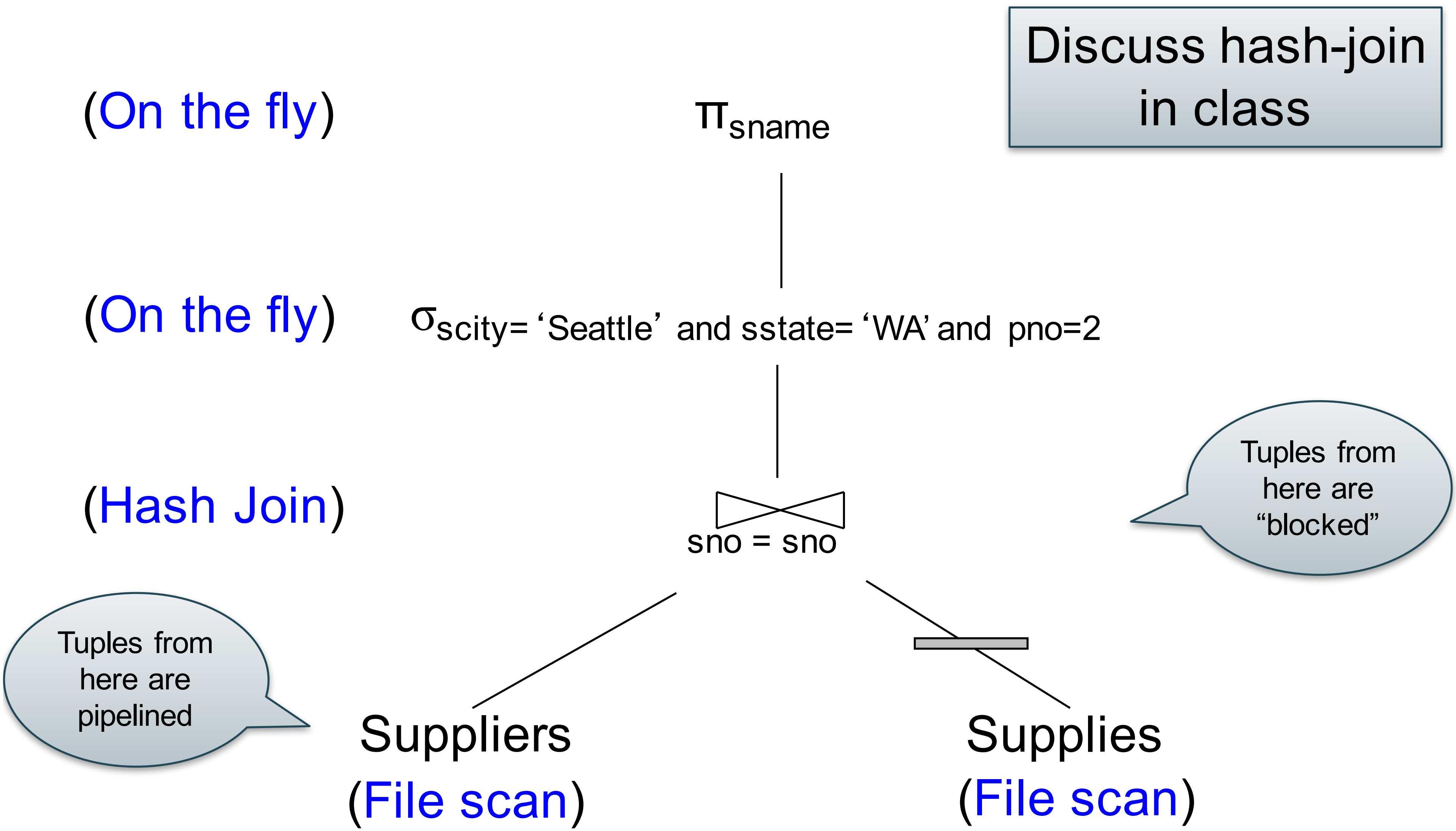
Pipelining

Discuss hash-join in class



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

Pipelining

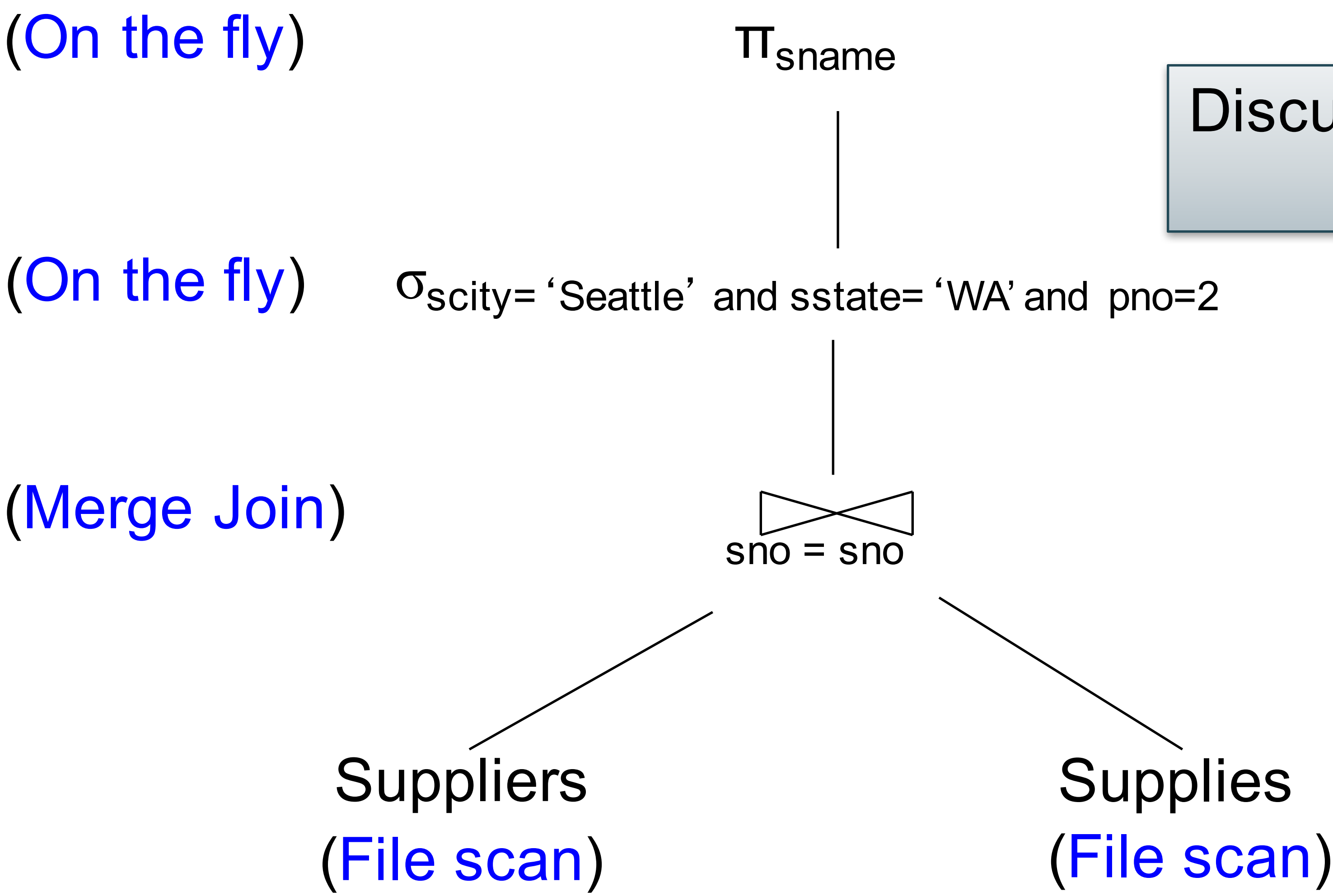


Supplier(<u>sid</u>, sname, scity, sstate)

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

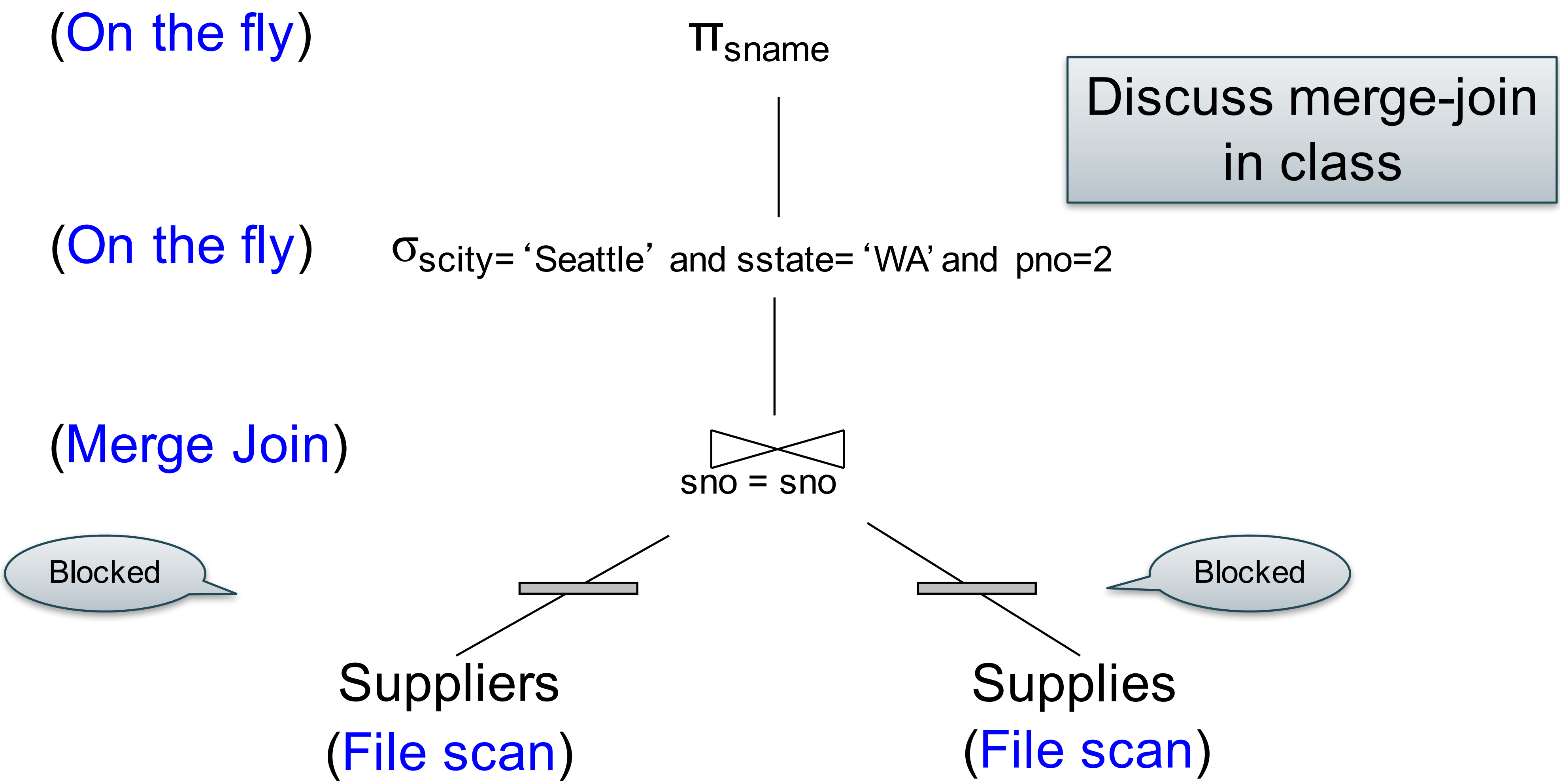
Blocked Execution

Discuss merge-join in class



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

Blocked Execution



Pipelined Execution

- Tuples generated by an operator are immediately sent to the parent
- Benefits:
  - No operator synchronization issues
  - No need to buffer tuples between operators
  - Saves cost of writing intermediate data to disk
  - Saves cost of reading intermediate data from disk
- This approach is used whenever possible

Query Execution Bottom Line

- SQL query transformed into physical plan
  - **Access path selection** for each relation
    - Scan the relation or use an index (next lecture)
  - **Implementation choice** for each operator
    - Nested loop join, hash join, etc.
  - **Scheduling decisions** for operators
    - Pipelined execution or intermediate materialization
- Pipelined execution of physical plan

Recall: Physical Data Independence

- Applications are insulated from changes in physical storage details
- SQL and relational algebra facilitate physical data independence
  - Both languages input and output relations
  - Can choose different implementations for operators

Class overview

- Data models
  - Relational: SQL, RA, and Datalog
  - NoSQL: SQL++
- RDBMS internals
  - Query processing and optimization
  - Physical design
- Parallel query processing
  - Spark and Hadoop
- Conceptual design
  - E/R diagrams
  - Schema normalization
- Transactions
  - Locking and schedules
  - Writing DB applications

Data models

Query Processing

Using DBMS

Query Performance

- My database application is too slow… why?
- One of the queries is very slow… why?

- To understand performance, we need to understand:
  - How is data organized on disk
  - How to estimate query costs

  - In this course we will focus on **disk-based** DBMSs

Student

ID	fName	lName
10	Tom	Hanks
20	Amy	Hanks
…		

Data Storage

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

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

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

Student

ID	fName	lName
10	Tom	Hanks
20	Amy	Hanks
…		

Data File Types

The data file can be one of:

- Heap file
  - Unsorted
- Sequential file
  - Sorted according to some attribute(s) called *key*

Student

ID	fName	lName
10	Tom	Hanks
20	Amy	Hanks
…		

Data File Types

The data file can be one of:

- Heap file
  - Unsorted
- Sequential file
  - Sorted according to some attribute(s) called *key*

Note: *key* here means something different from primary key: it just means that we order the file according to that attribute. In our example we ordered by **ID**. Might as well order by **fName,** if that seems a better idea for the applications running on our database.

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:
  - The key = an attribute value (e.g., student ID or name)
  - The value = a pointer to the record

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

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

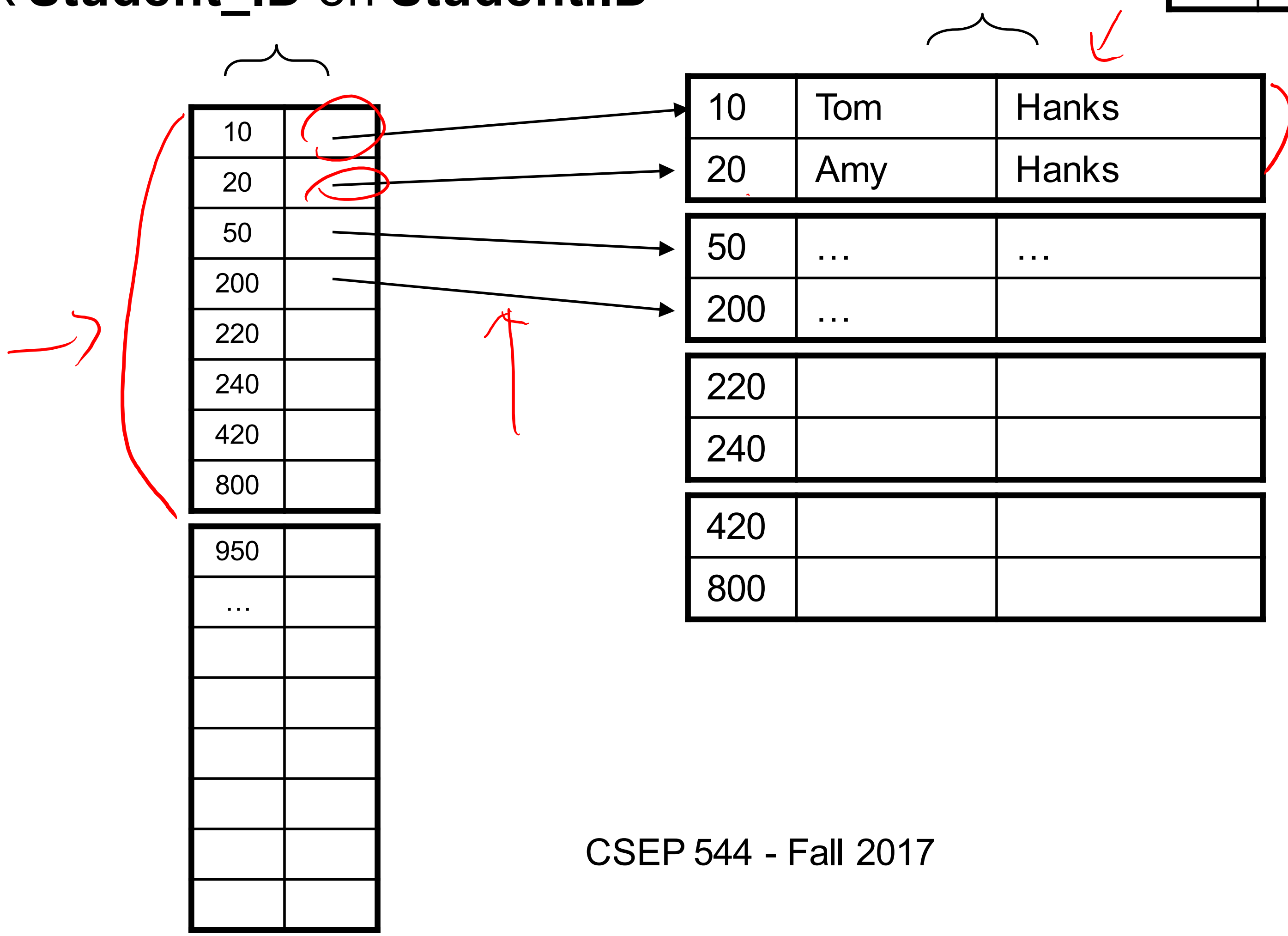
Student

ID	fName	lName
10	Tom	Hanks
20	Amy	Hanks
…		

Example 1: Index on ID

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

Data File **Student**



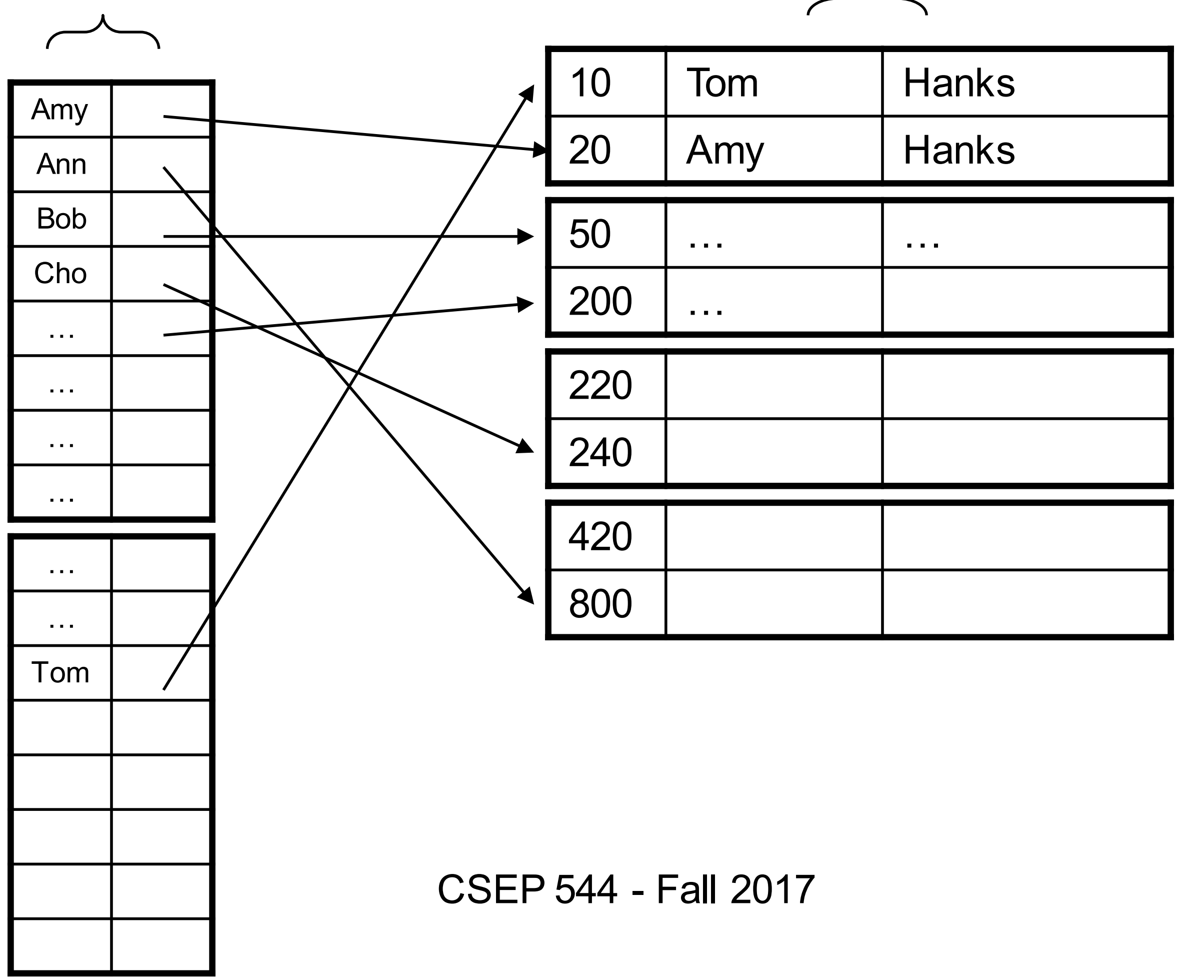
Example 2: Index on fName

Student

ID	fName	lName
10	Tom	Hanks
20	Amy	Hanks
…		

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

Data File **Student**



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

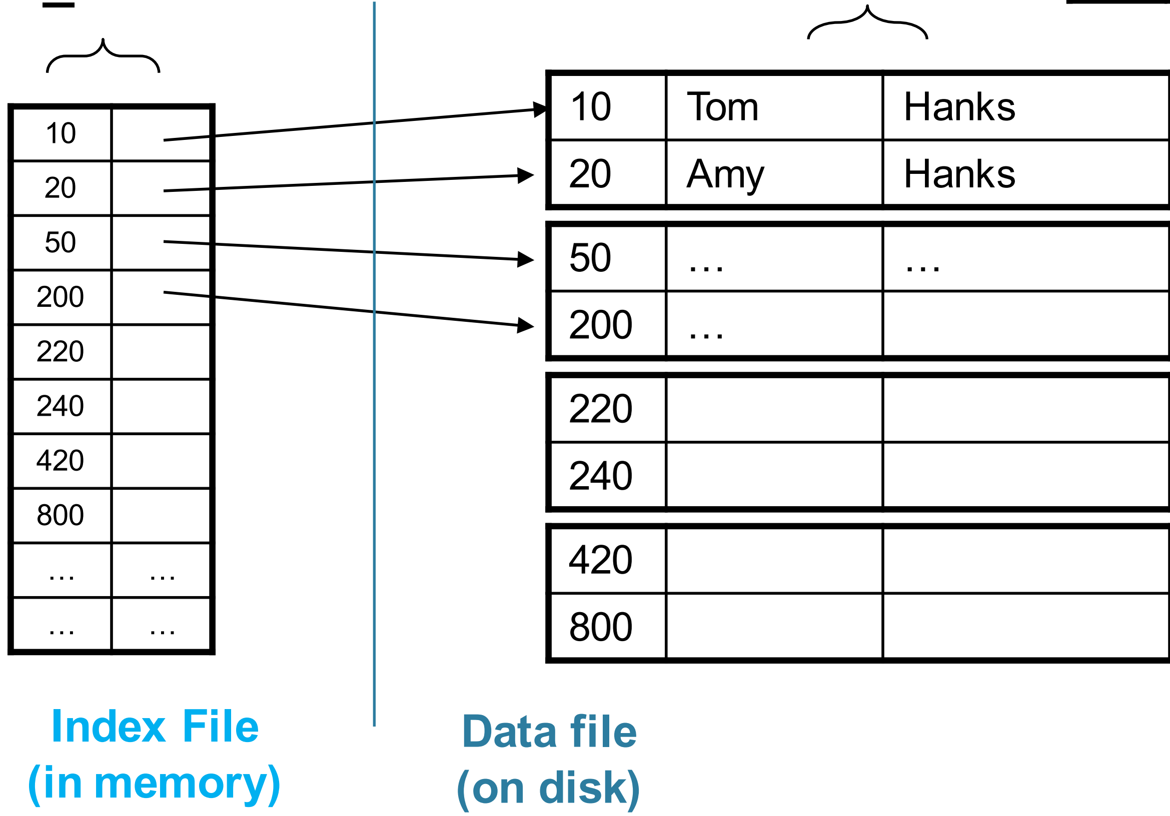
Hash table example

Student

ID	fName	lName
10	Tom	Hanks
20	Amy	Hanks
…		

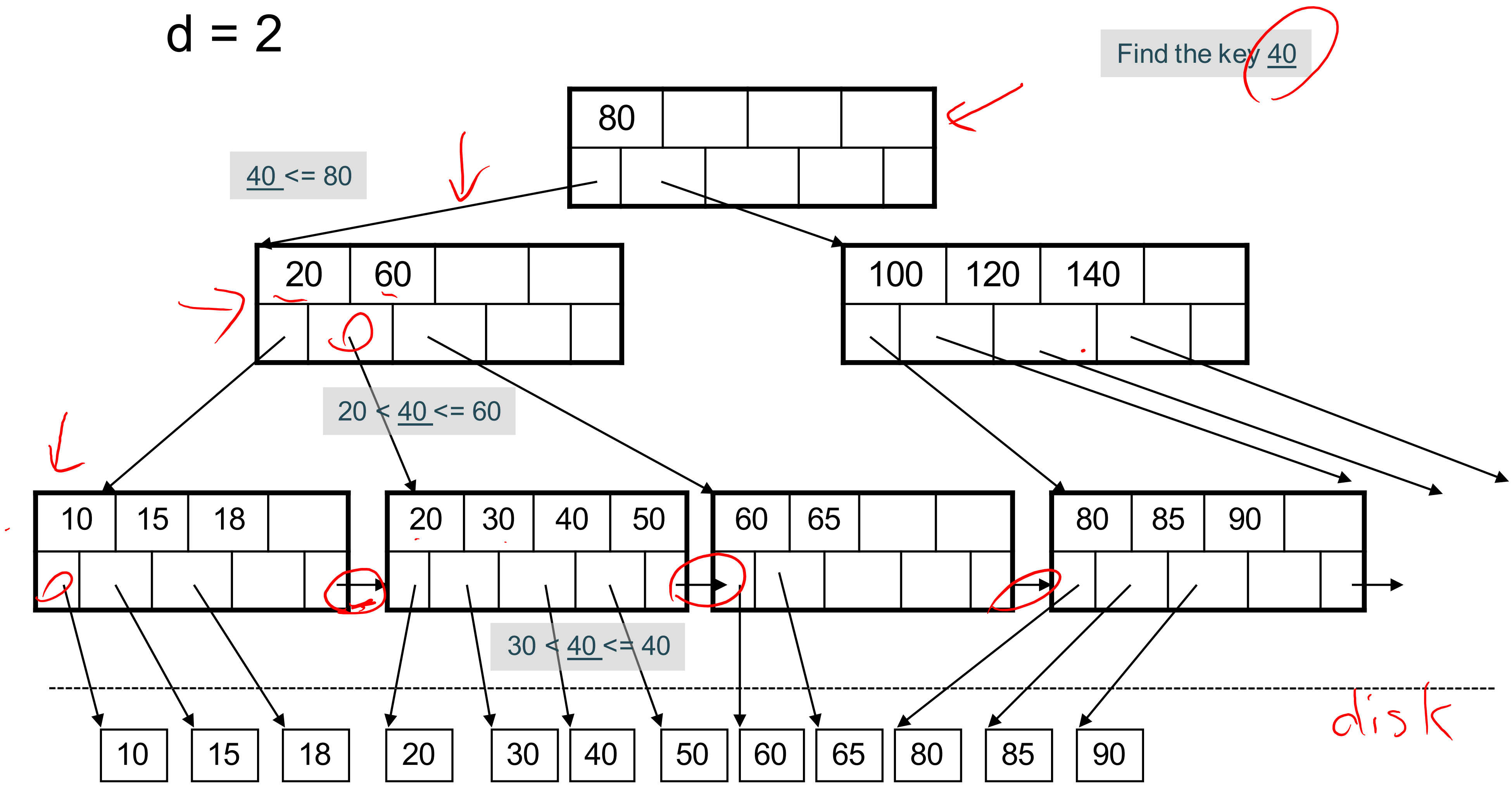
Index **Student_ID** on **Student.ID**

Data File **Student**

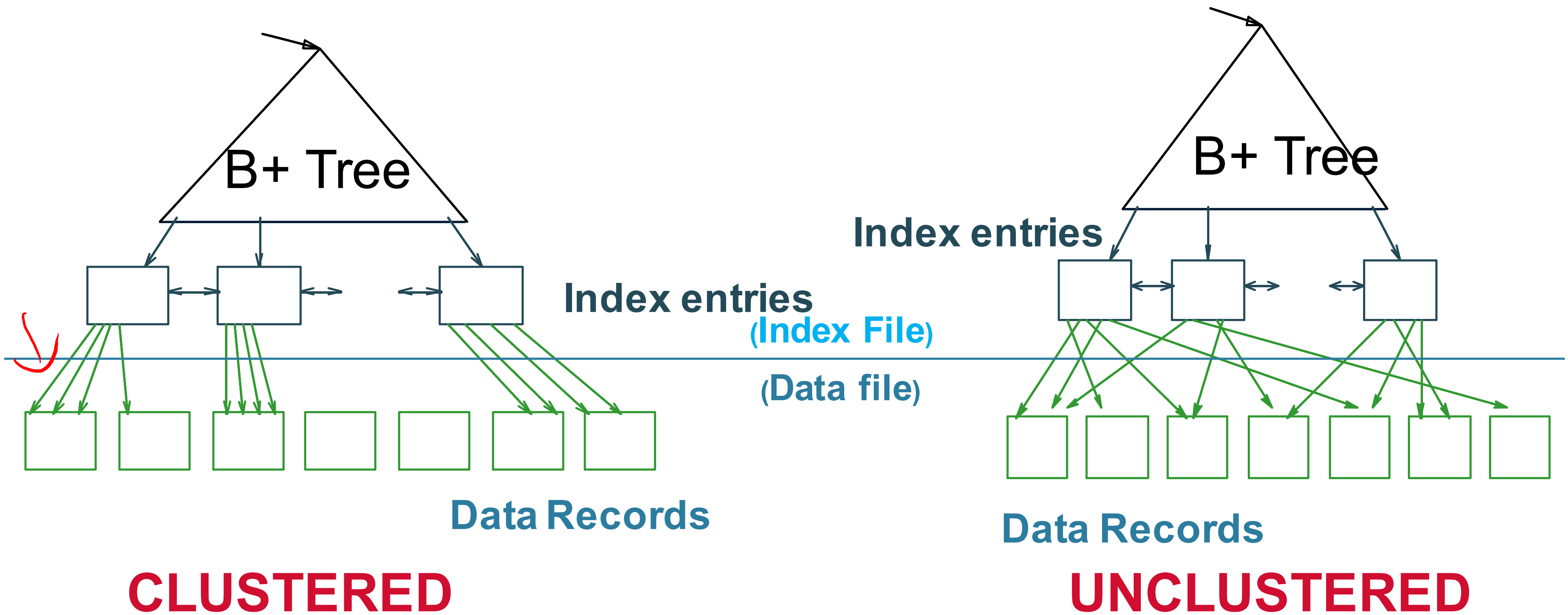


B+ Tree Index by Example

d = 2



Clustered vs Unclustered



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

Index Classification

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

Index Classification

- **Clustered/unclustered**
  - Clustered = records close in index are close in data
    - Option 1: Data inside data file is sorted on disk
    - Option 2: Store data directly inside the index (no separate files)
  - Unclustered = records close in index may be far in data
- **Primary/secondary**
  - Meaning 1:
    - Primary = is over attributes that include the primary key
    - Secondary = otherwise
  - Meaning 2: means the same as clustered/unclustered

Index Classification

- **Clustered/unclustered**
  - Clustered = records close in index are close in data
    - Option 1: Data inside data file is sorted on disk
    - Option 2: Store data directly inside the index (no separate files)
  - Unclustered = records close in index may be far in data
- **Primary/secondary**
  - Meaning 1:
    - Primary = is over attributes that include the primary key
    - Secondary = otherwise
  - Meaning 2: means the same as clustered/unclustered
- **Organization** B+ tree or Hash table

Scanning a Data File

- Disks are mechanical devices!
  - Technology from the 60s; density much higher now
- Read only at the rotation speed!
- Consequence:
  Sequential scan is MUCH FASTER than random reads
  - Good: read blocks 1,2,3,4,5,…
  - Bad: read blocks 2342, 11, 321,9, …
- Rule of thumb:
  - Random reading 1-2% of the file ≈ sequential scanning the entire file; this is decreasing over time (because of increased density of disks)
- Solid state (SSD): $$$ expensive; put indexes, other “hot” data there, not enough room for everything (NO LONGER TRUE)

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

Example

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

Assume the database has indexes on these attributes:

- **index_takes_courseID** = index on Takes.courseID
- **index_student_ID** = index on Student.ID

```
for y in index_Takes_courseID  where y.courseID > 300
  for x in Student  where x.ID = y.studentID
        output  *
```

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

Example

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

Assume the database has indexes on these attributes:

- **index_takes_courseID** = index on Takes.courseID
- **index_student_ID** = index on Student.ID

Index selection

```
for y in index_Takes_courseID  where y.courseID > 300
  for x in Student where x.ID = y.studentID
        output  *
```

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

Example

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

Assume the database has indexes on these attributes:

- **index_takes_courseID** = index on Takes.courseID
- **index_student_ID** = index on Student.ID

Index selection

```
for y in index_Takes_courseID  where y.courseID > 300
  for x in Student where x.ID = y.studentID
        output  *
```

Index join

Getting Practical: Creating Indexes in SQL

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

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

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

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

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

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

Student

ID	fName	lName
10	Tom	Hanks
20	Amy	Hanks
…		

Which Indexes?

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

Student

ID	fName	lName
10	Tom	Hanks
20	Amy	Hanks
…		

Which Indexes?

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

In general this is a very hard problem

Student

ID	fName	lName
10	Tom	Hanks
20	Amy	Hanks
…		

Which Indexes?

- The *index selection problem*
  - Given a table, and a “workload” (big Java application with lots of SQL queries), decide which indexes to create (and which ones NOT to create!)
- Who does index selection:
  - The database administrator DBA
  - Semi-automatically, using a database administration tool

Student

ID	fName	lName
10	Tom	Hanks
20	Amy	Hanks
…		

Which Indexes?

- The *index selection problem*
  - Given a table, and a “workload” (big Java application with lots of SQL queries), decide which indexes to create (and which ones NOT to create!)
- Who does index selection:
  - The database administrator DBA

  - Semi-automatically, using a database administration tool

Index Selection: Which Search Key

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

The Index Selection Problem 1

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

Your workload is this

100000 queries:

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

100 queries:

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

The Index Selection Problem 1

V(M, N, P);

Your workload is this

100000 queries:

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

100 queries:

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

What indexes ?

The Index Selection Problem 1

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

Your workload is this

100000 queries:

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

100 queries:

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

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

The Index Selection Problem 2

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

Your workload is this

100000 queries:

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

100 queries:

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

100000 queries:

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

What indexes ?

The Index Selection Problem 2

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

Your workload is this

100000 queries:

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

100 queries:

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

100000 queries:

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

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

Two typical kinds of queries

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

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

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

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

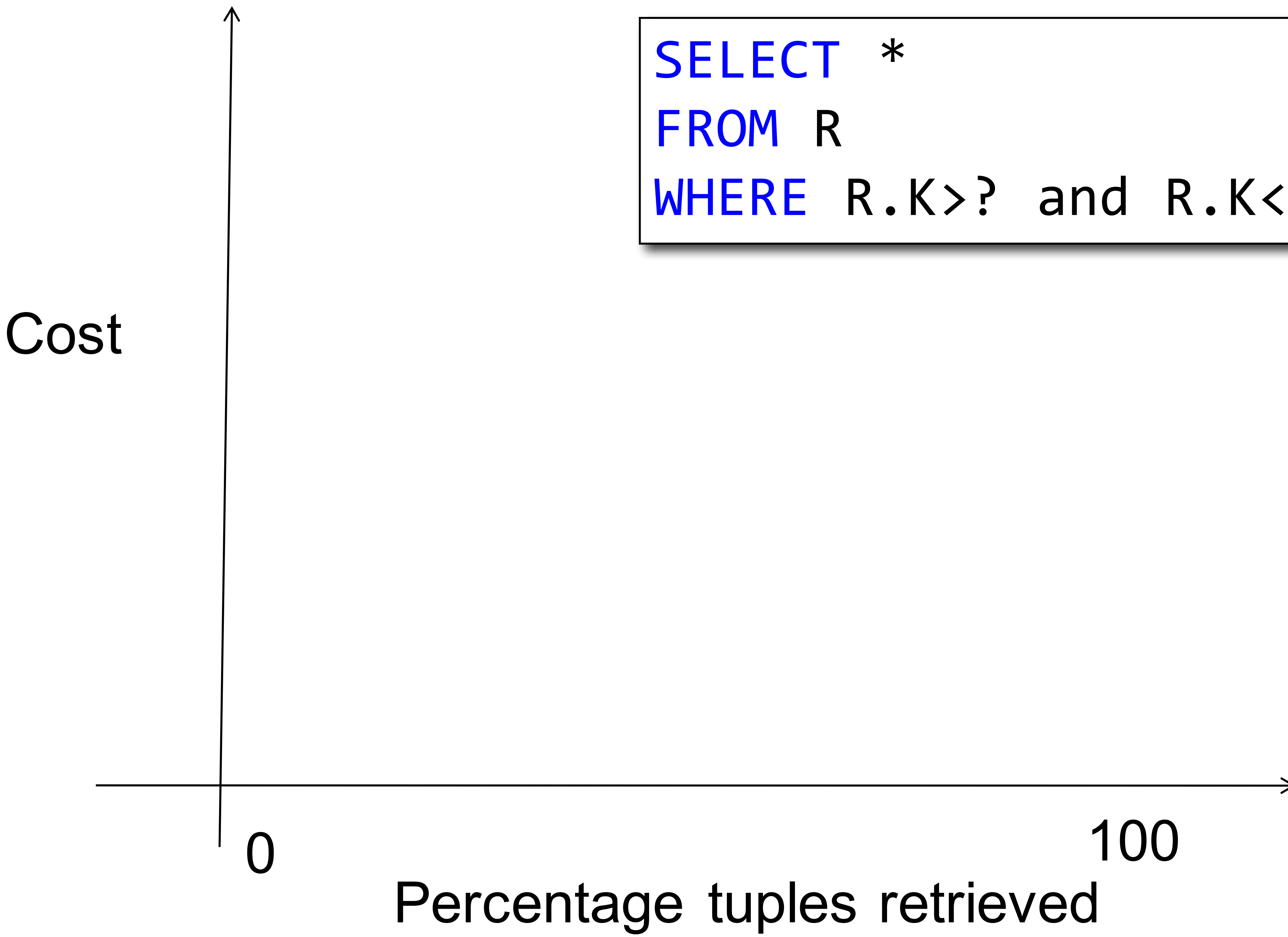
Basic Index Selection Guidelines

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

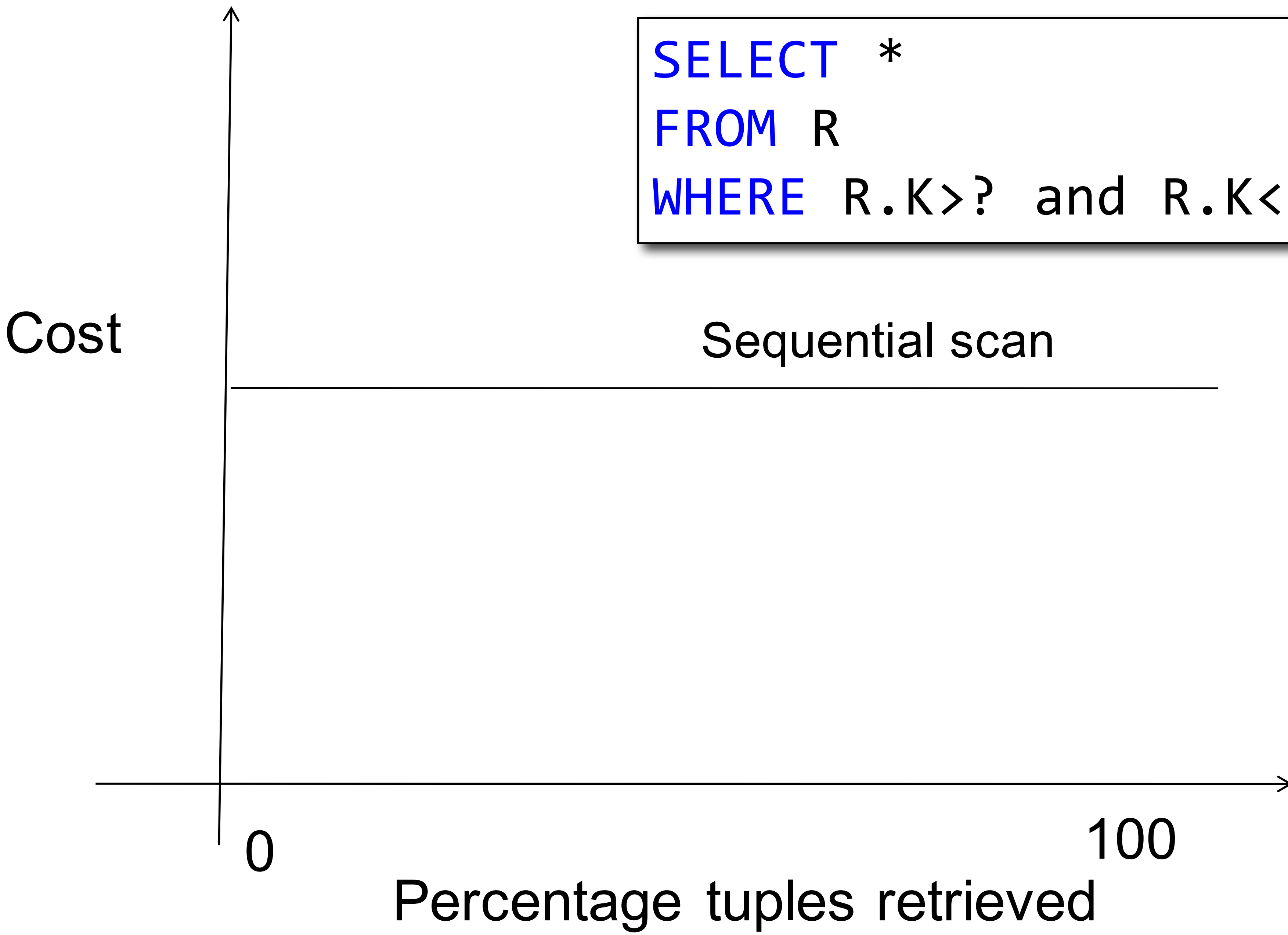
To Cluster or Not

- Range queries benefit mostly from clustering
- Covering indexes do *not* need to be clustered: they work equally well unclustered

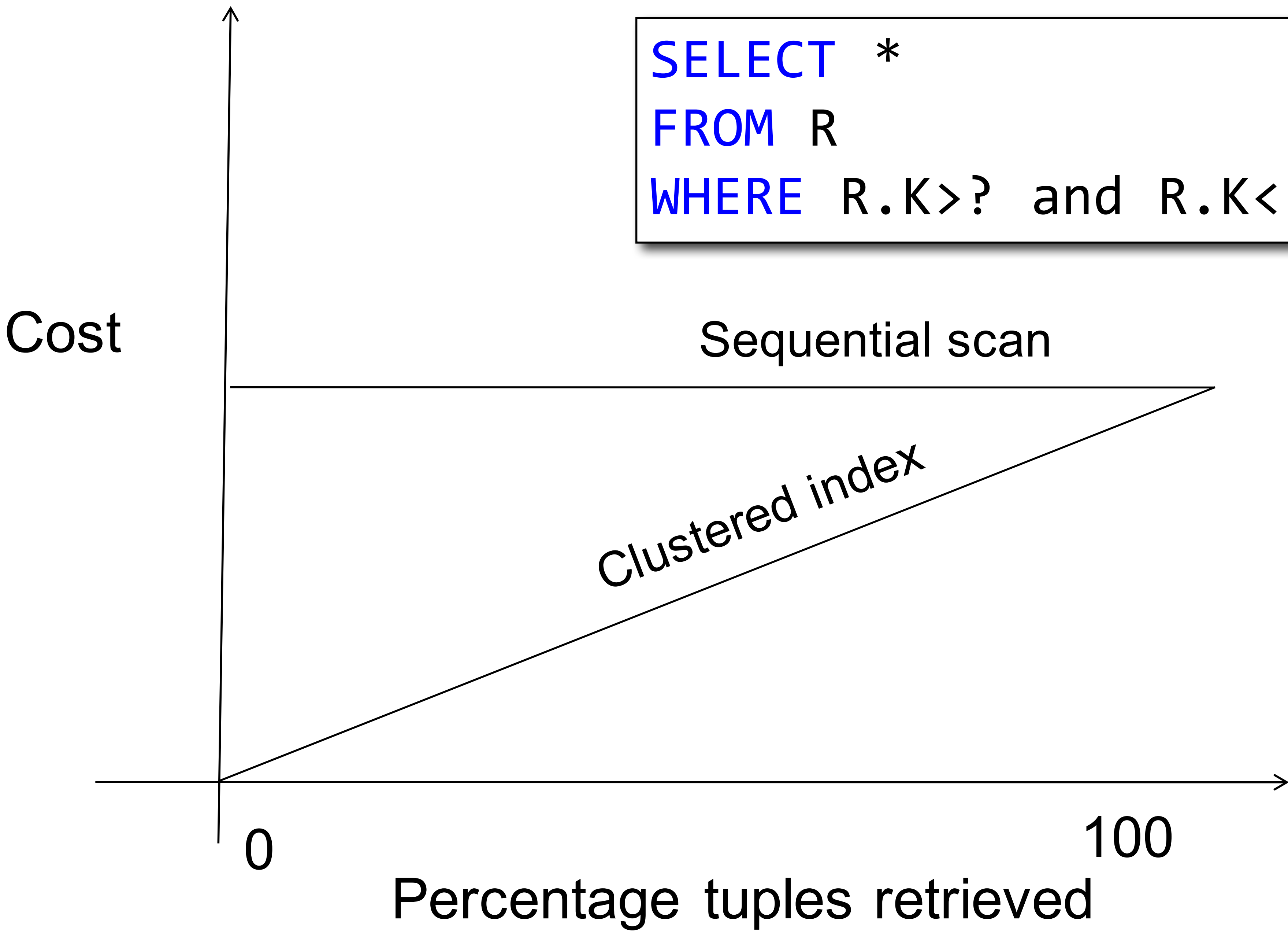
```
SELECT *
FROM R
WHERE R.K>? and R.K<?
```



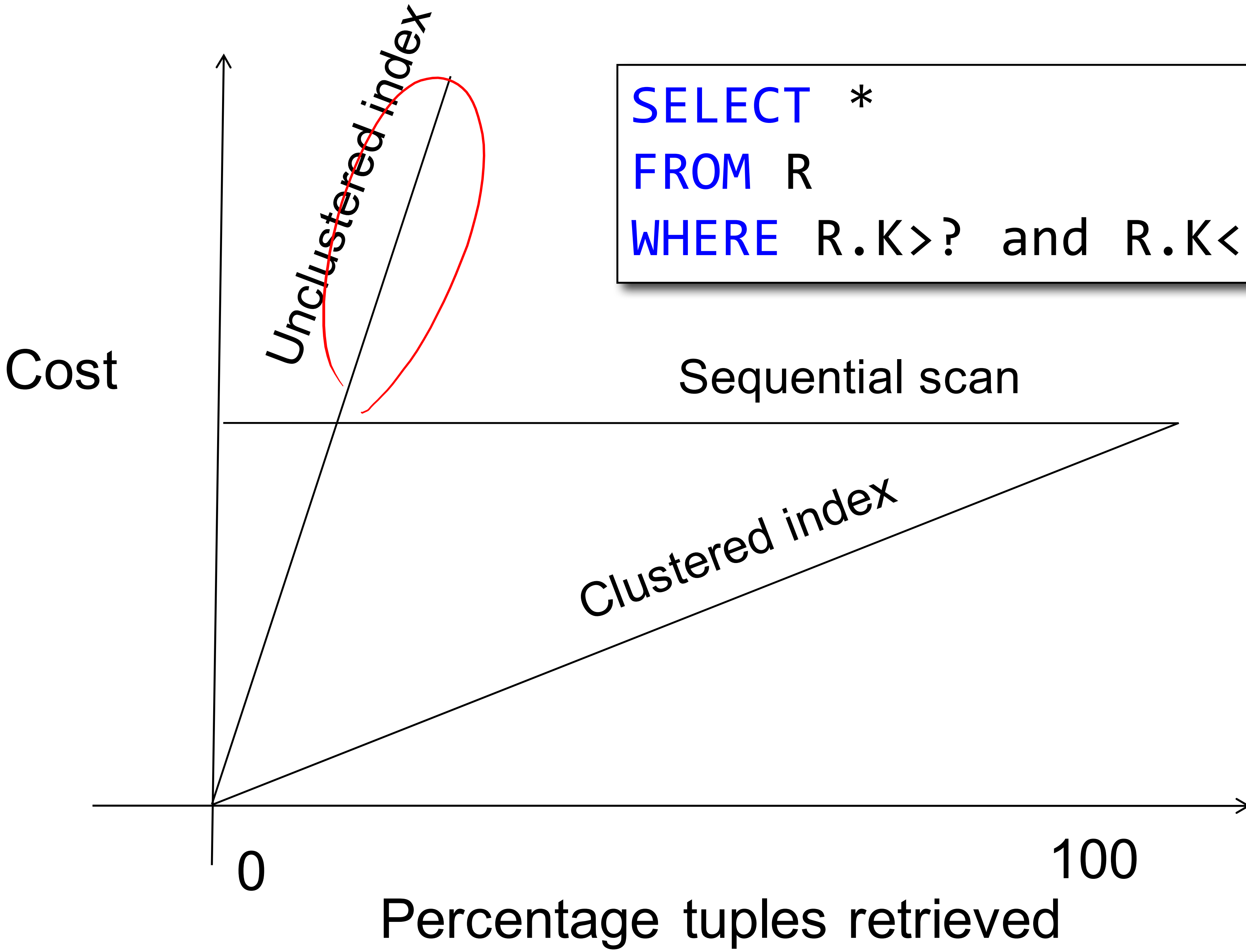
```
SELECT *
FROM R
WHERE R.K>? and R.K<?
```



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



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



Choosing Index is Not Enough

- To estimate the cost of a query plan, we still need to consider other factors:
  - How each operator is implemented
  - The cost of each operator
  - Let's start with the basics

Cost of Reading Data From Disk

Cost Parameters

- Cost = I/O + ~~CPU~~ + ~~Network~~ BW
  - We will focus on I/O in this class
- 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
    - When **a** is a key, **V(R,a) = T(R)**
    - When **a** is not a key, **V(R,a)** can be anything <= **T(R)**
- Where do these values come from?
  - DBMS collects statistics about data on disk

Selectivity Factors for Conditions

- A = c /* $\sigma_{A=c}(R)$ */
  - Selectivity = 1/V(R,A)
- A < c /* $\sigma_{A<c}(R)$*/
  - Selectivity = (c - min(R, A))/(max(R,A) - min(R,A))
- c1 < A < c2 /* $\sigma_{c1<A<c2}(R)$*/
  - Selectivity = (c2 – c1)/(max(R,A) - min(R,A))

Cost of Reading Data From Disk

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

X ≤ 1

Note: we ignore I/O cost for index pages

Index Based Selection

- Example:

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

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

- Table scan:
- Index based selection:

Index Based Selection

- Example:

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

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

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

Index Based Selection

- Example:

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

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

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

Index Based Selection

- Example:

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

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

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

Index Based Selection

- Example:

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

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

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

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

Cost of Executing Operators (Focus on Joins)

Outline

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

Join Algorithms

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

Hash Join

Hash join: $R \bowtie S$

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

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

Hash Join Example

Patient(pid, name, address)

Insurance(pid, provider, policy_nb)

Patient ⋈ Insurance

Two tuples per page

Patient

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

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

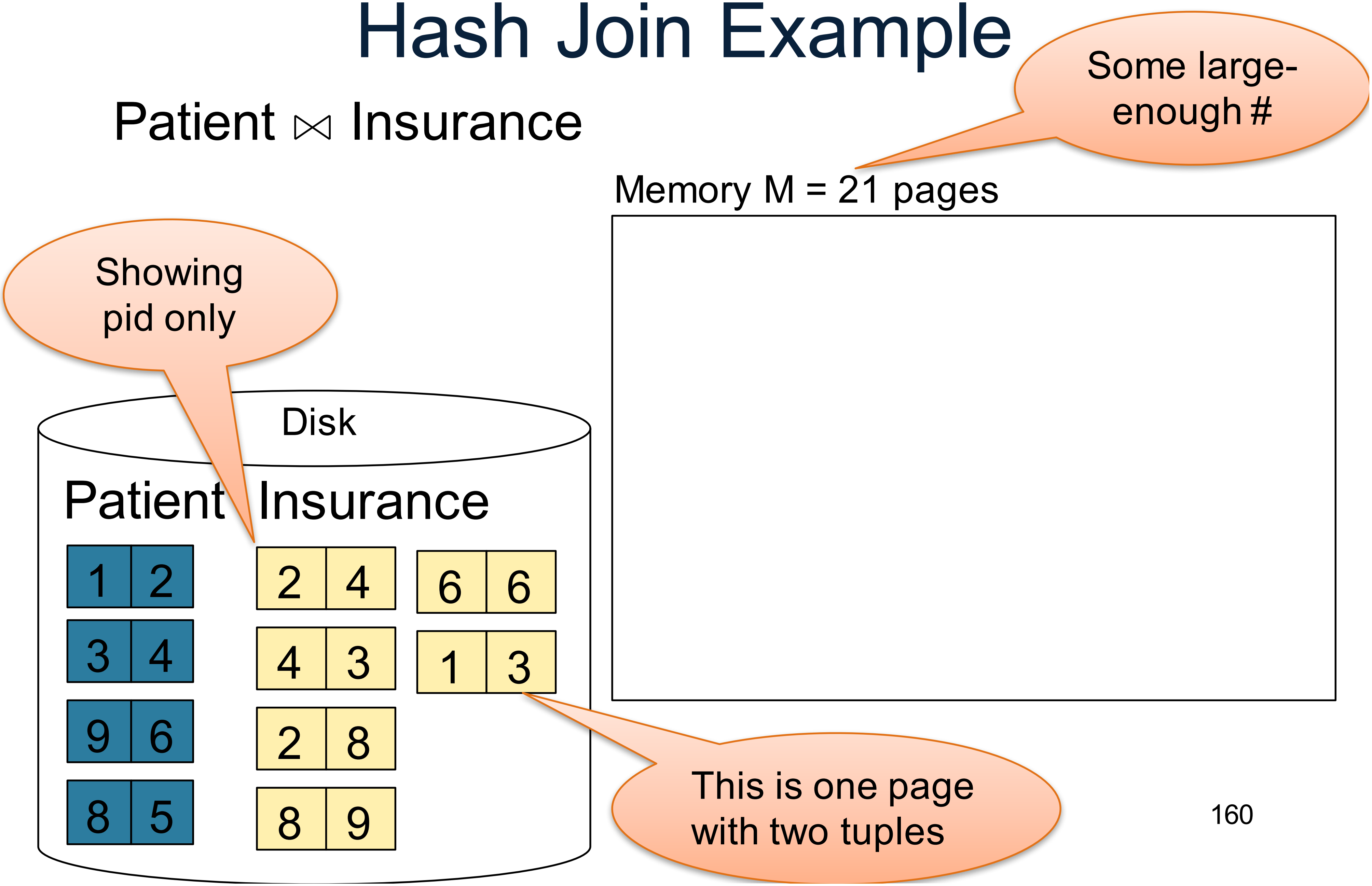
Insurance

2	'Blue'	123
4	'Prem'	432

4	'Prem'	343
3	'GrpH'	554

Hash Join Example

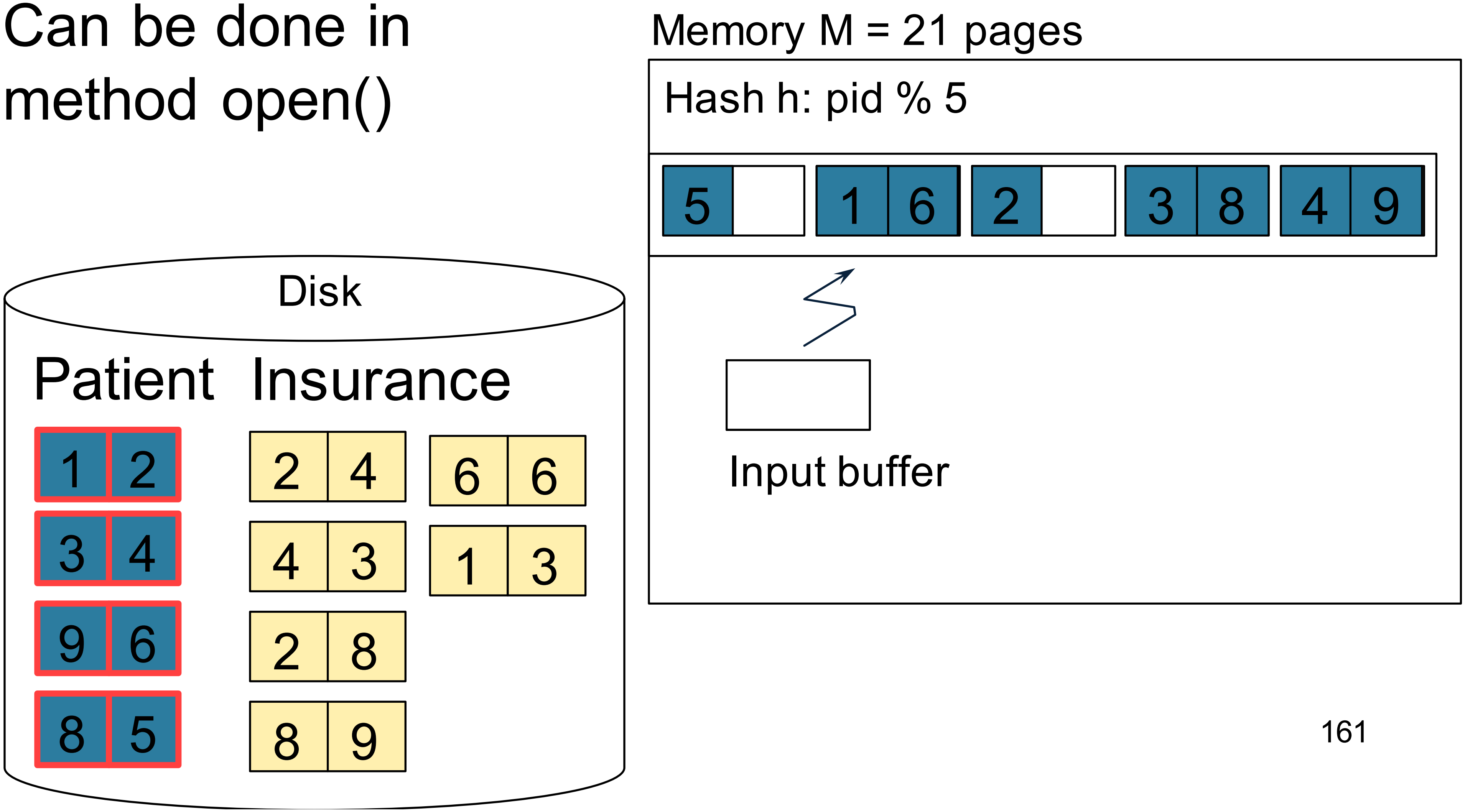
Patient $\bowtie$ Insurance



Hash Join Example

Step 1: Scan Patient and build hash table in memory

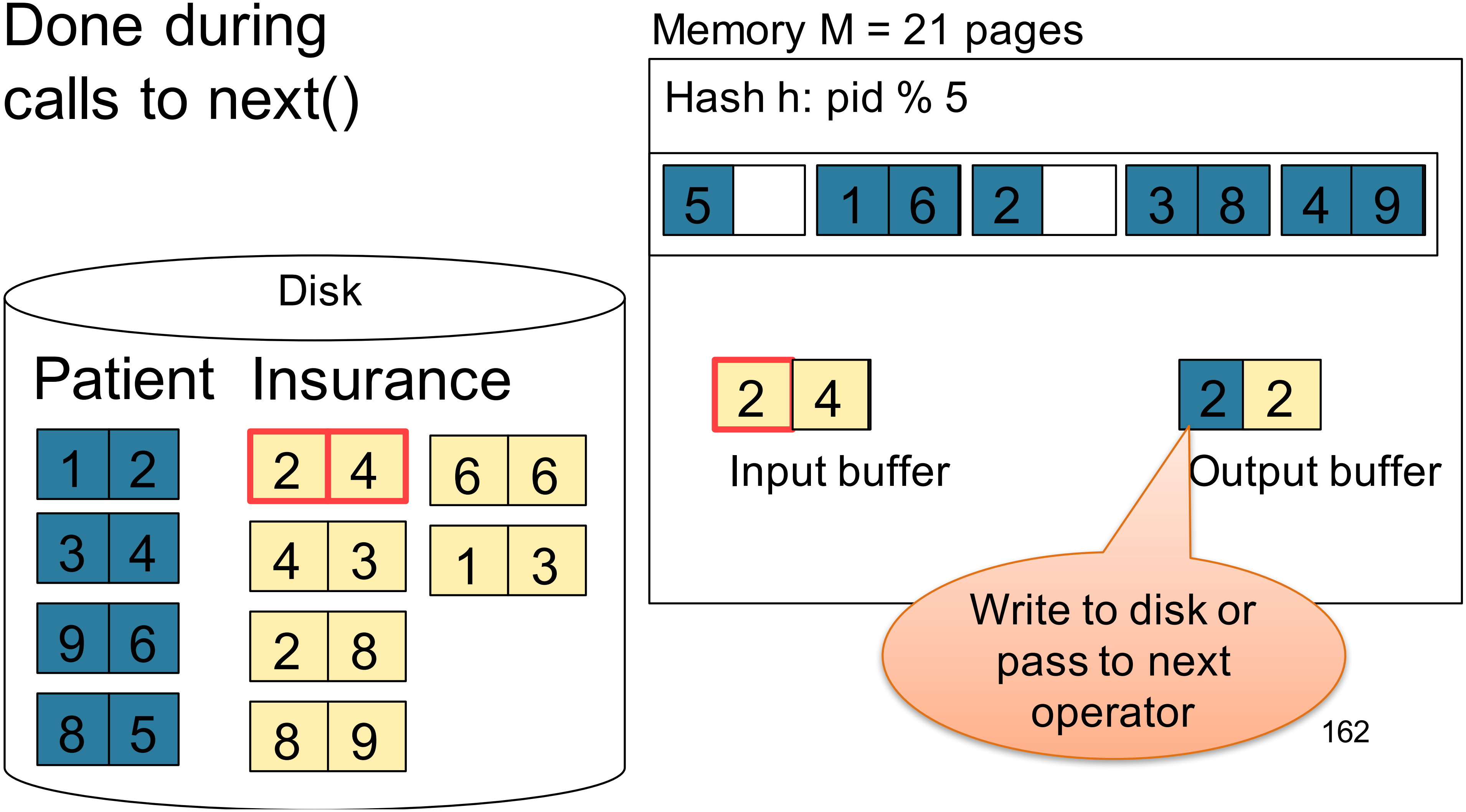
Can be done in method open()



Hash Join Example

Step 2: Scan Insurance and probe into hash table

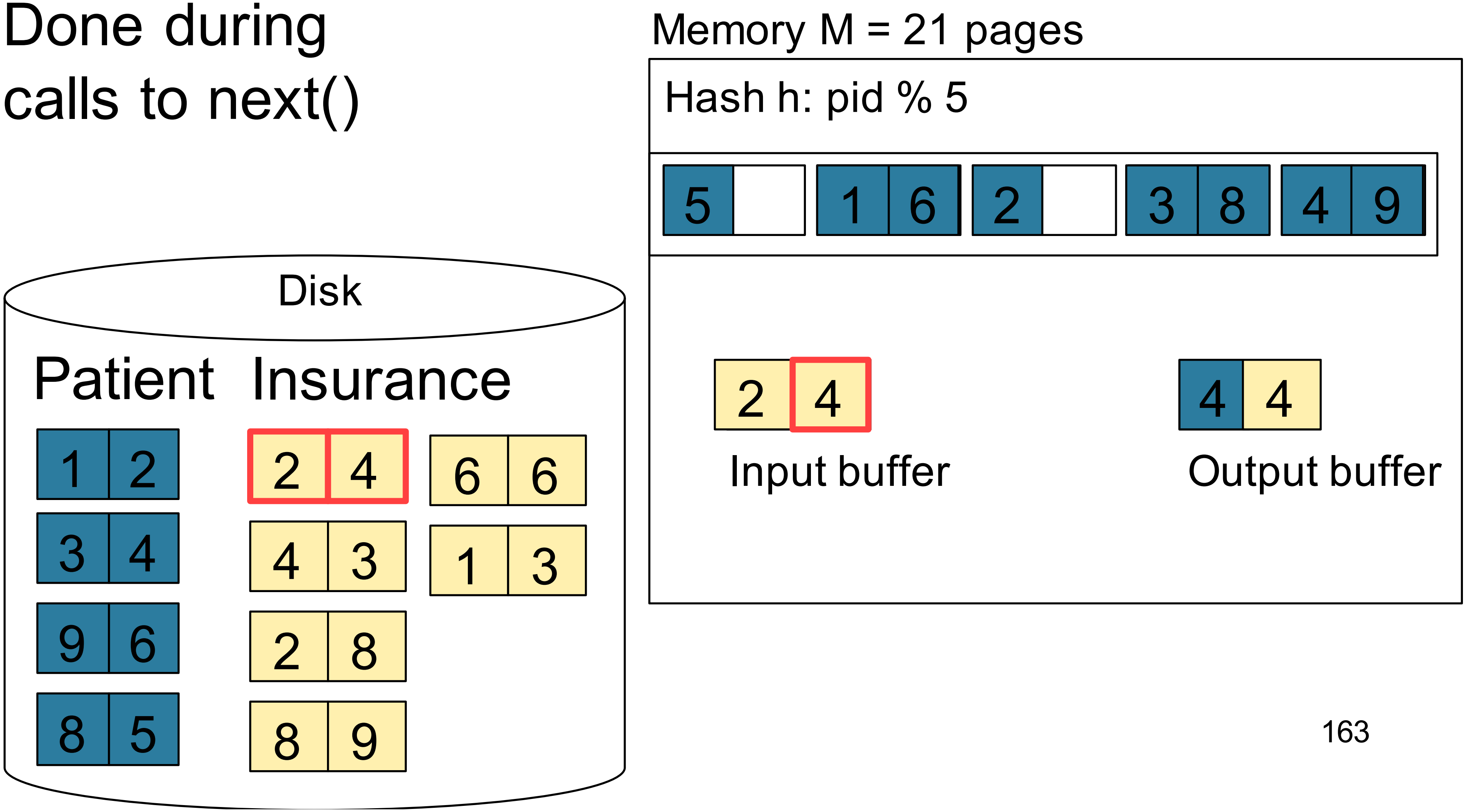
Done during calls to next()



Hash Join Example

Step 2: Scan Insurance and probe into hash table

Done during calls to next()

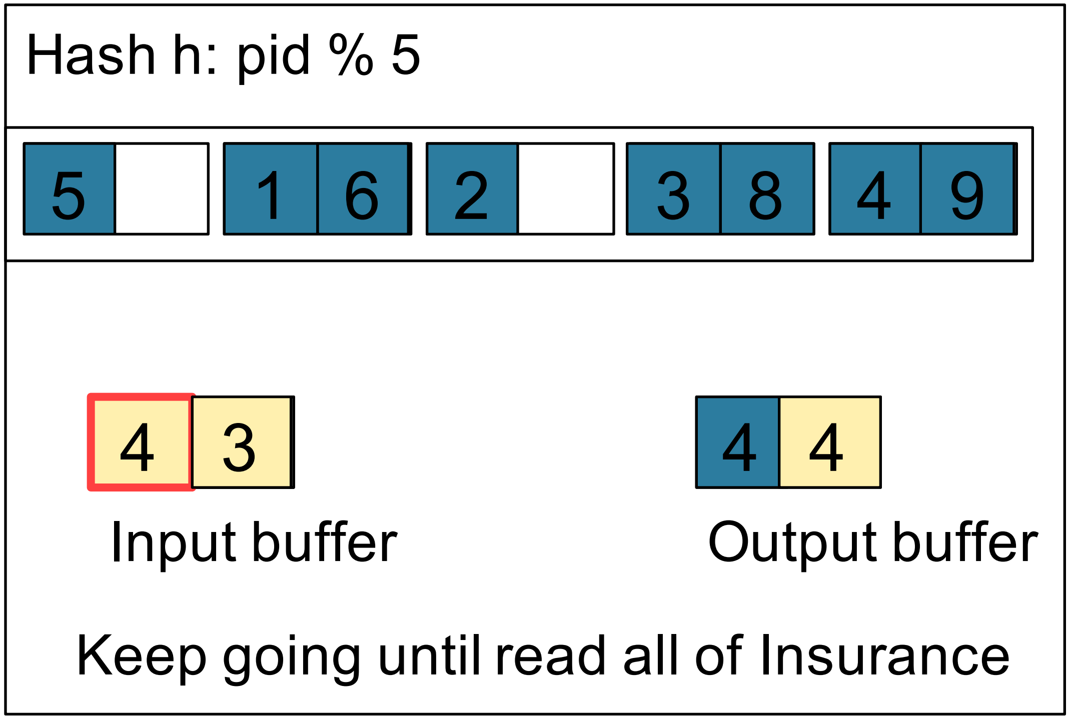


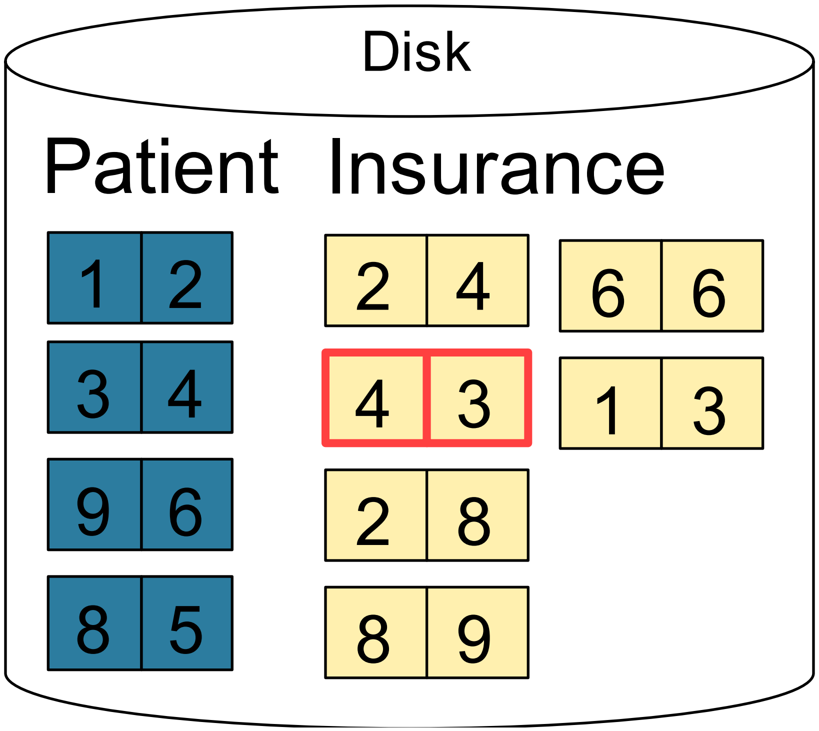
Hash Join Example

Step 2: Scan Insurance and probe into hash table

Done during calls to next()

Memory M = 21 pages





Cost: B(R) + B(S)

Nested Loop Joins

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

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

What is the Cost?

Nested Loop Joins

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

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

What is the Cost?

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

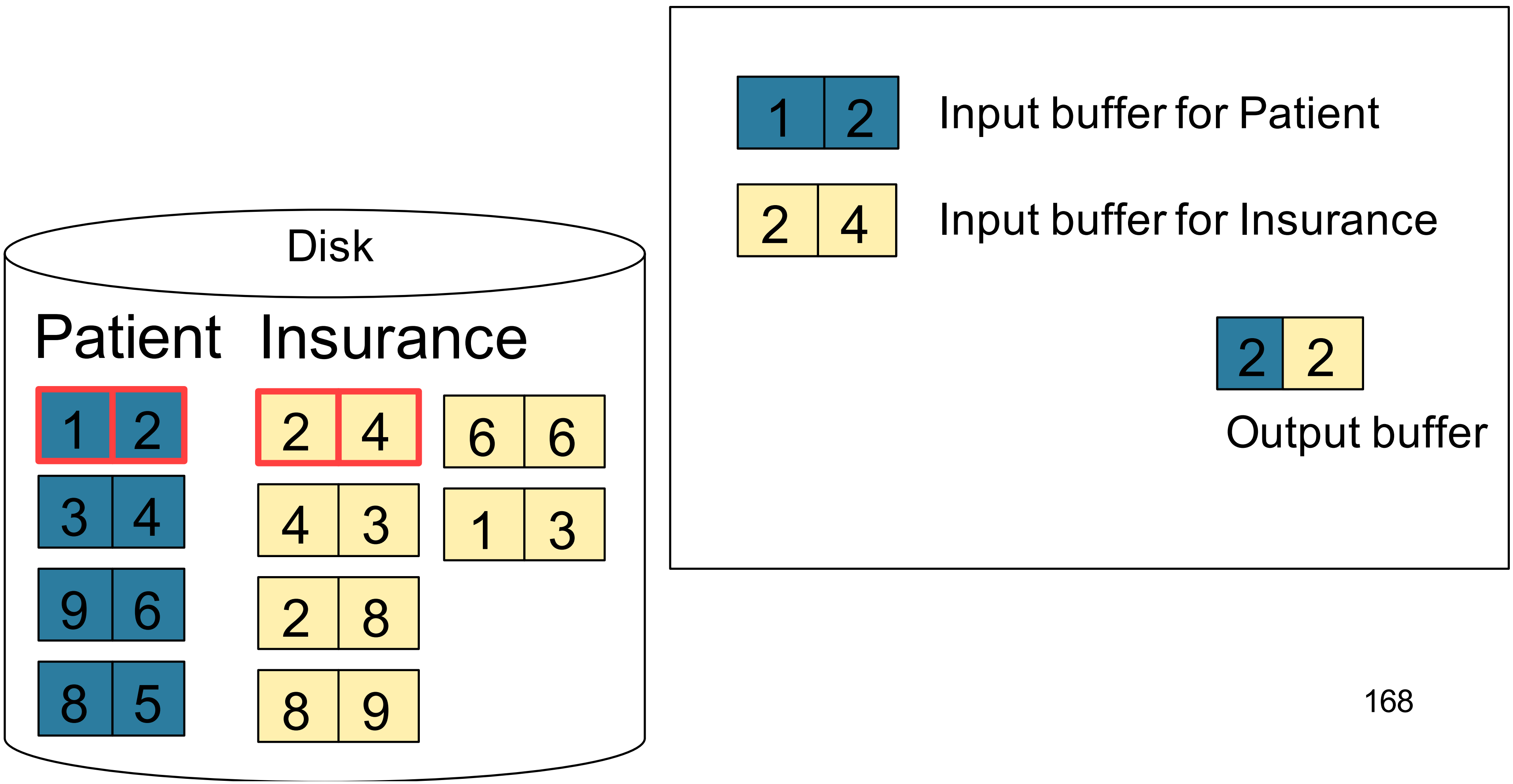
Page-at-a-time Refinement

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

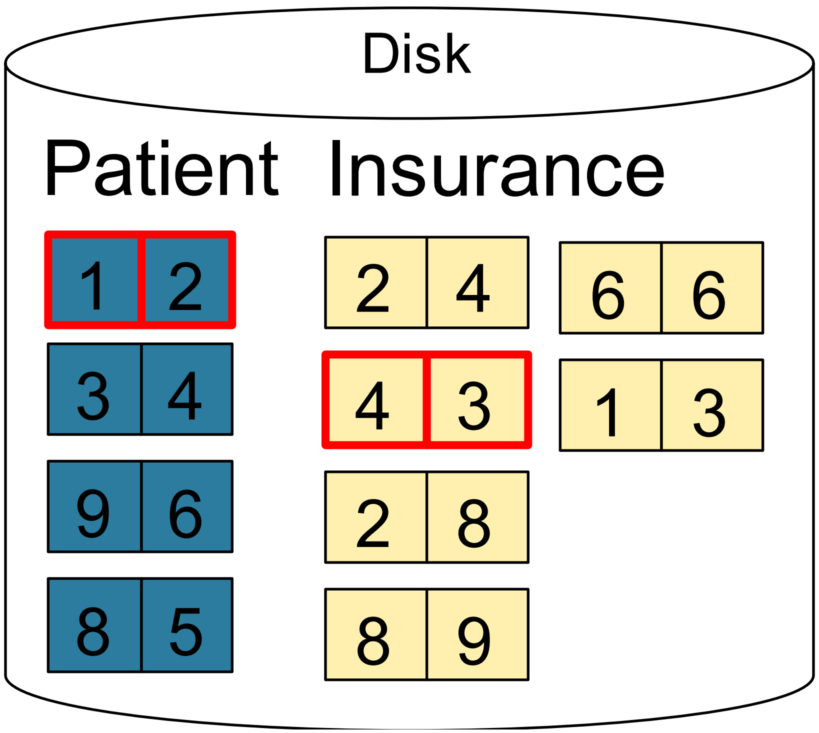
- Cost: B(R) + B(R)B(S)

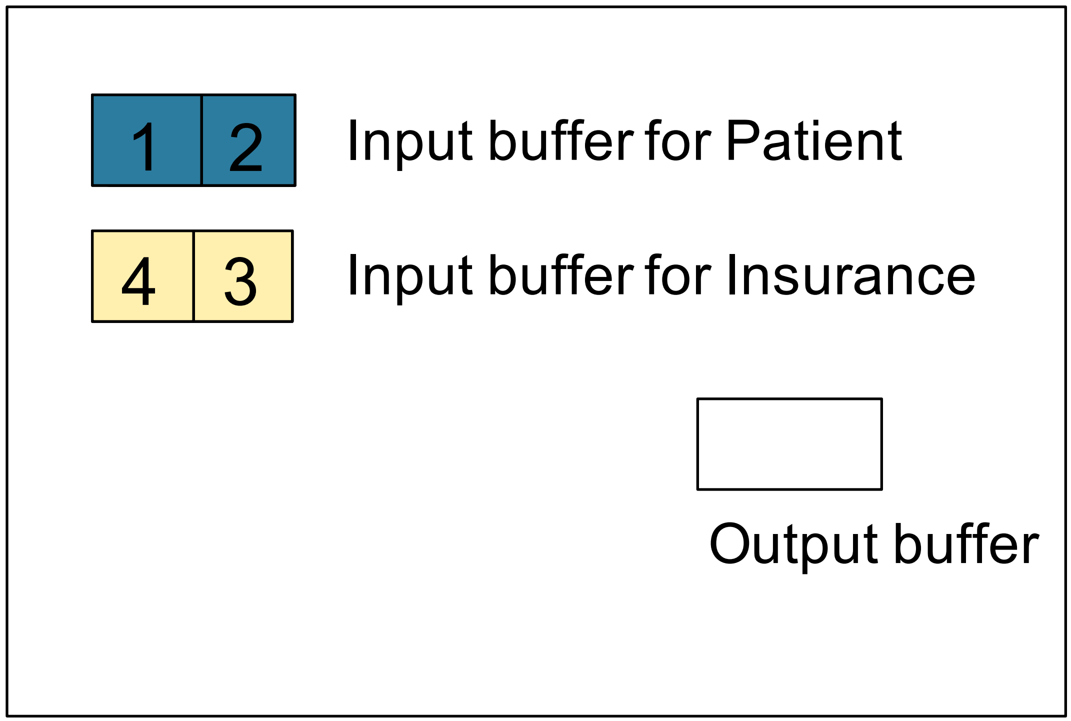
What is the Cost?

Page-at-a-time Refinement

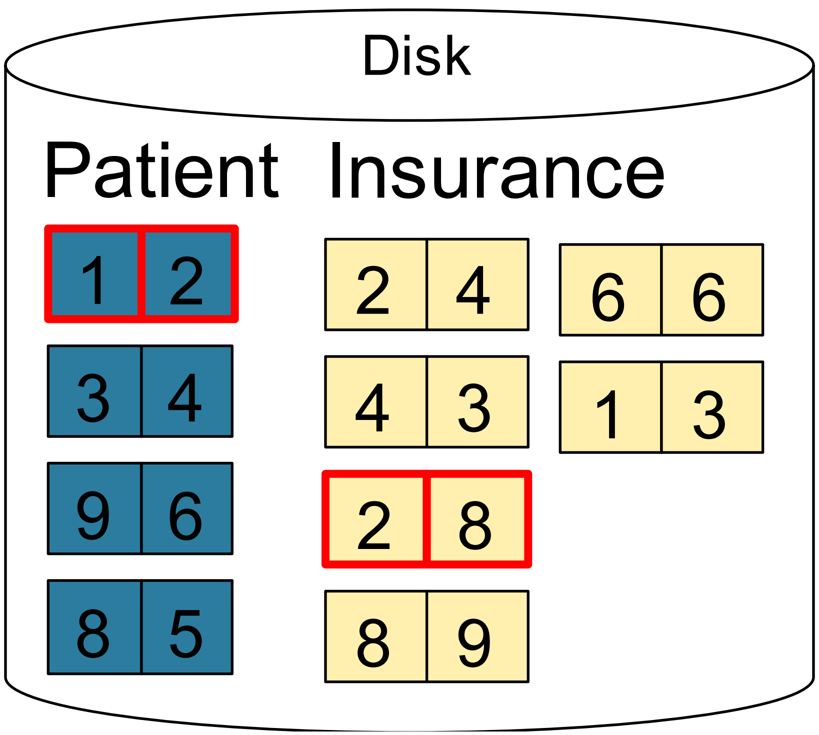


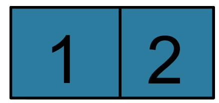
Page-at-a-time Refinement



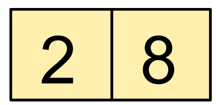


Page-at-a-time Refinement



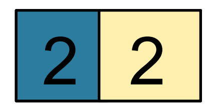


Input buffer for Patient



Input buffer for Insurance

Keep going until read all of Insurance



Output buffer

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

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

Sort-Merge Join

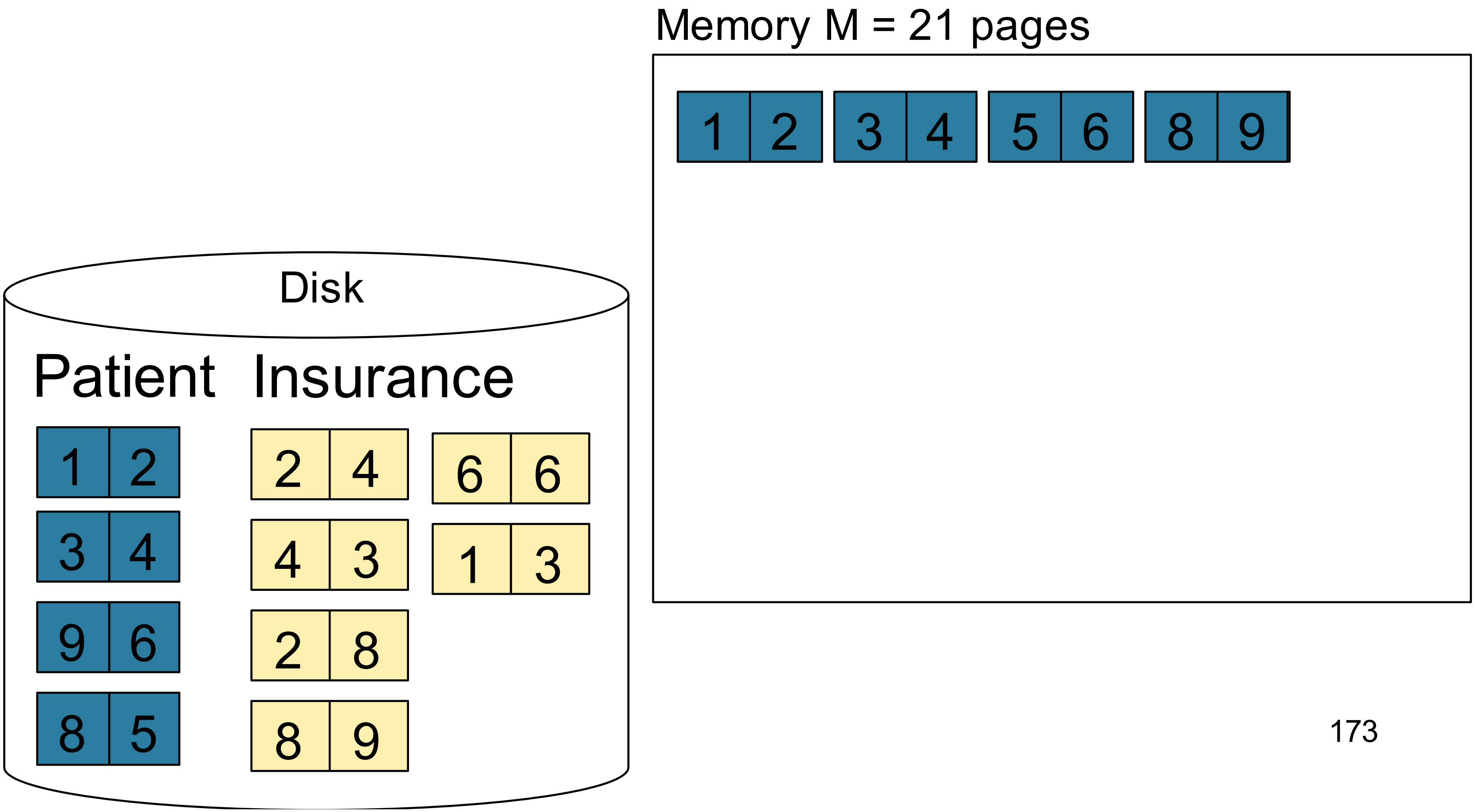
Sort-merge join: $R \bowtie S$

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

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

Sort-Merge Join Example

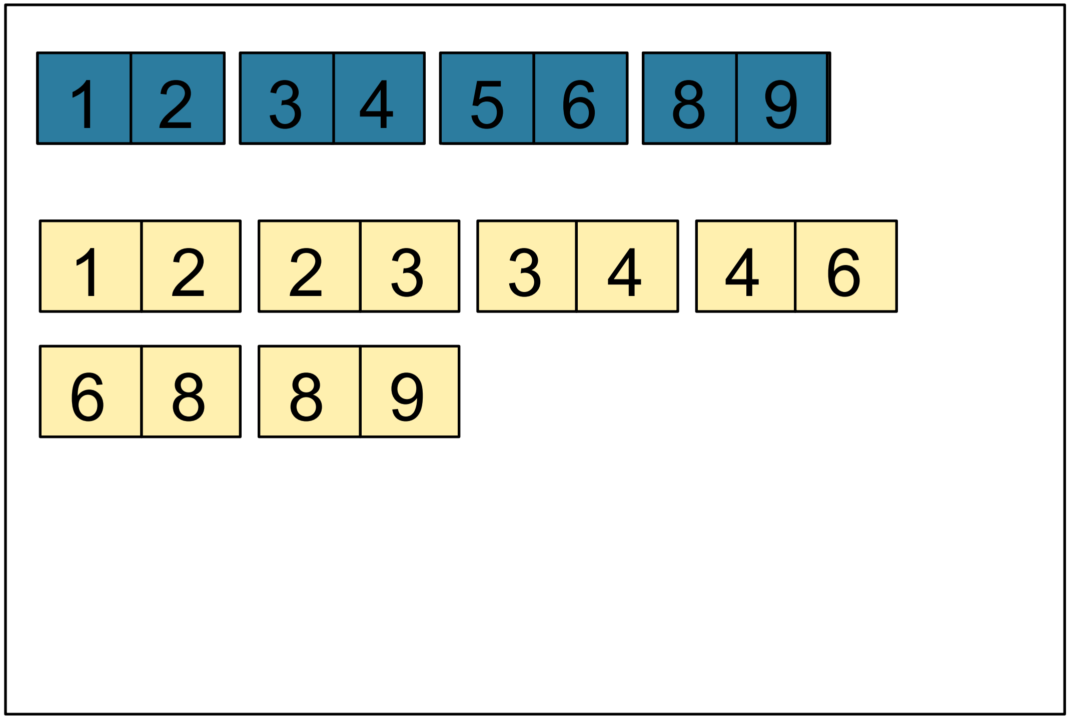
Step 1: Scan Patient and sort in memory

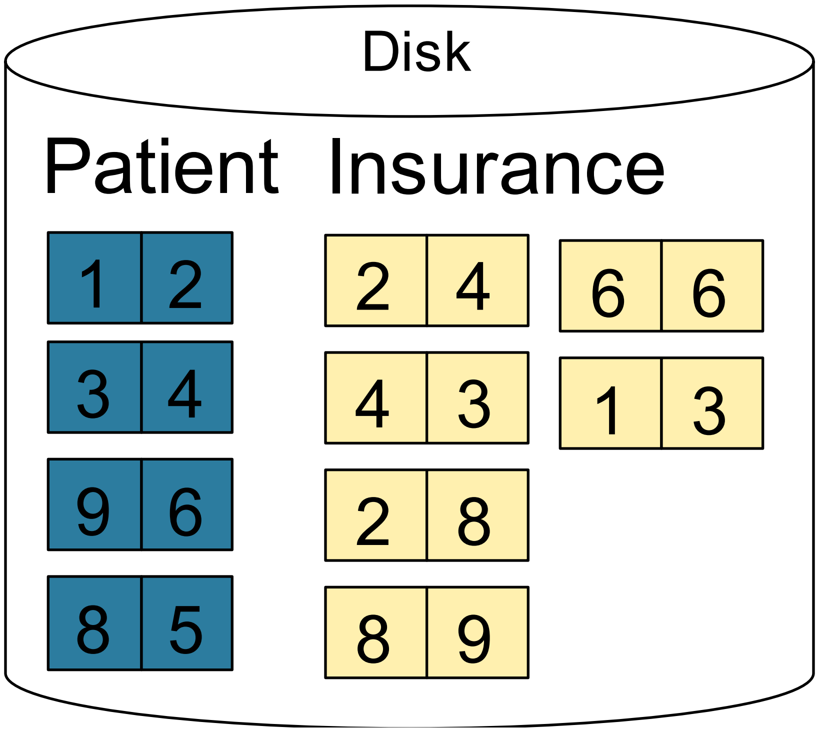


Sort-Merge Join Example

Step 2: Scan Insurance and sort in memory

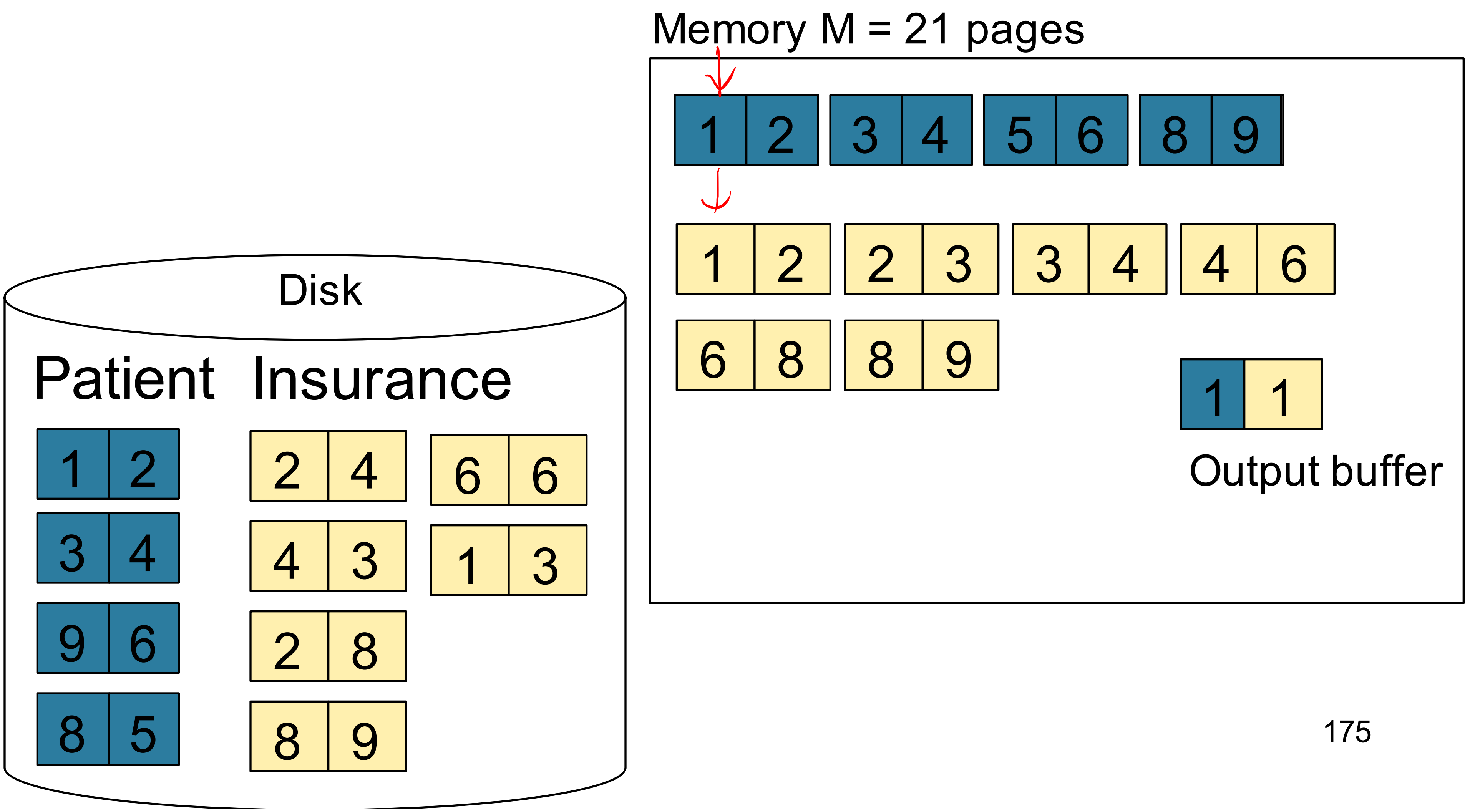
Memory M = 21 pages





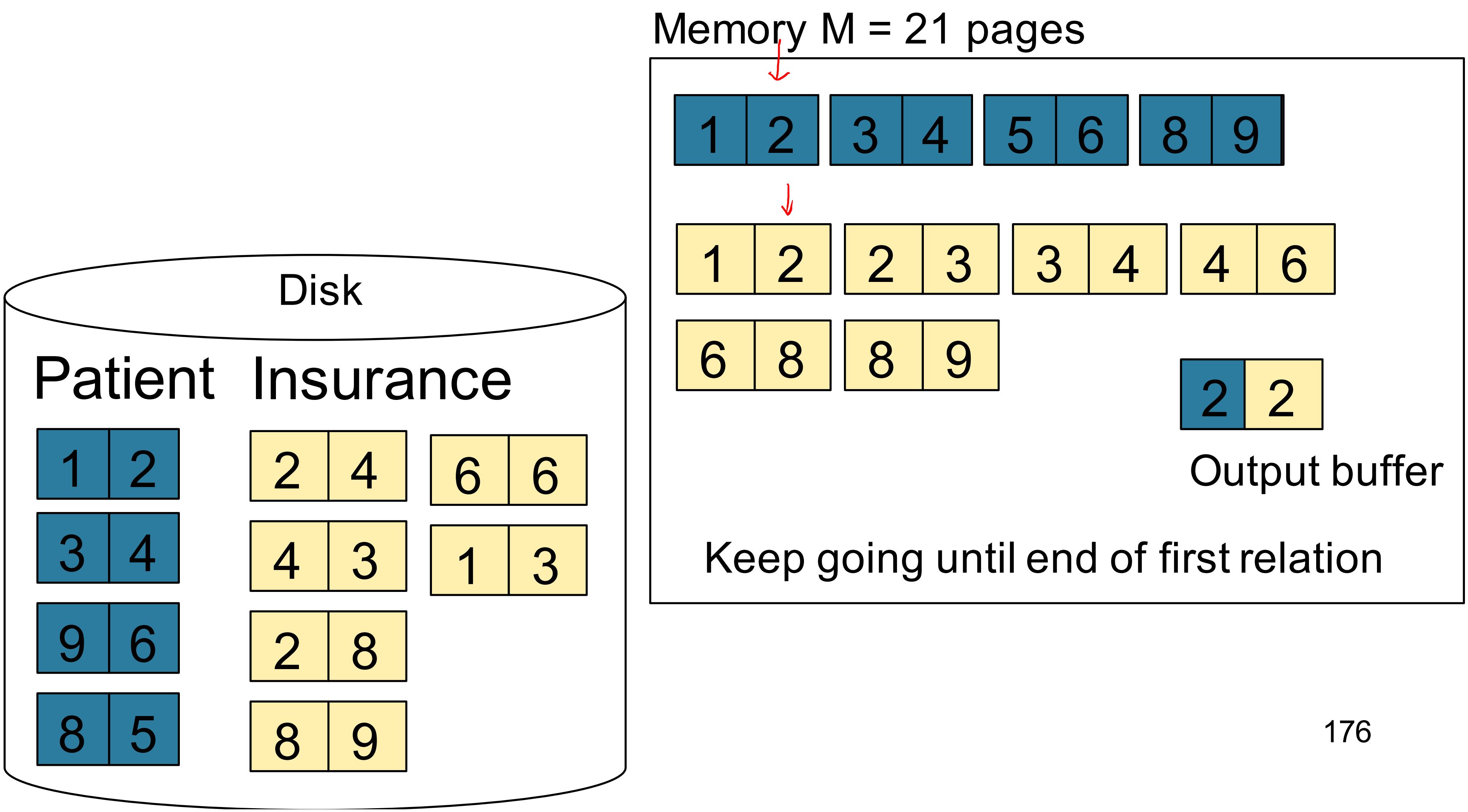
Sort-Merge Join Example

Step 3: Merge Patient and Insurance



Sort-Merge Join Example

Step 3: Merge Patient and Insurance



Index Nested Loop Join

$R \bowtie S$

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

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