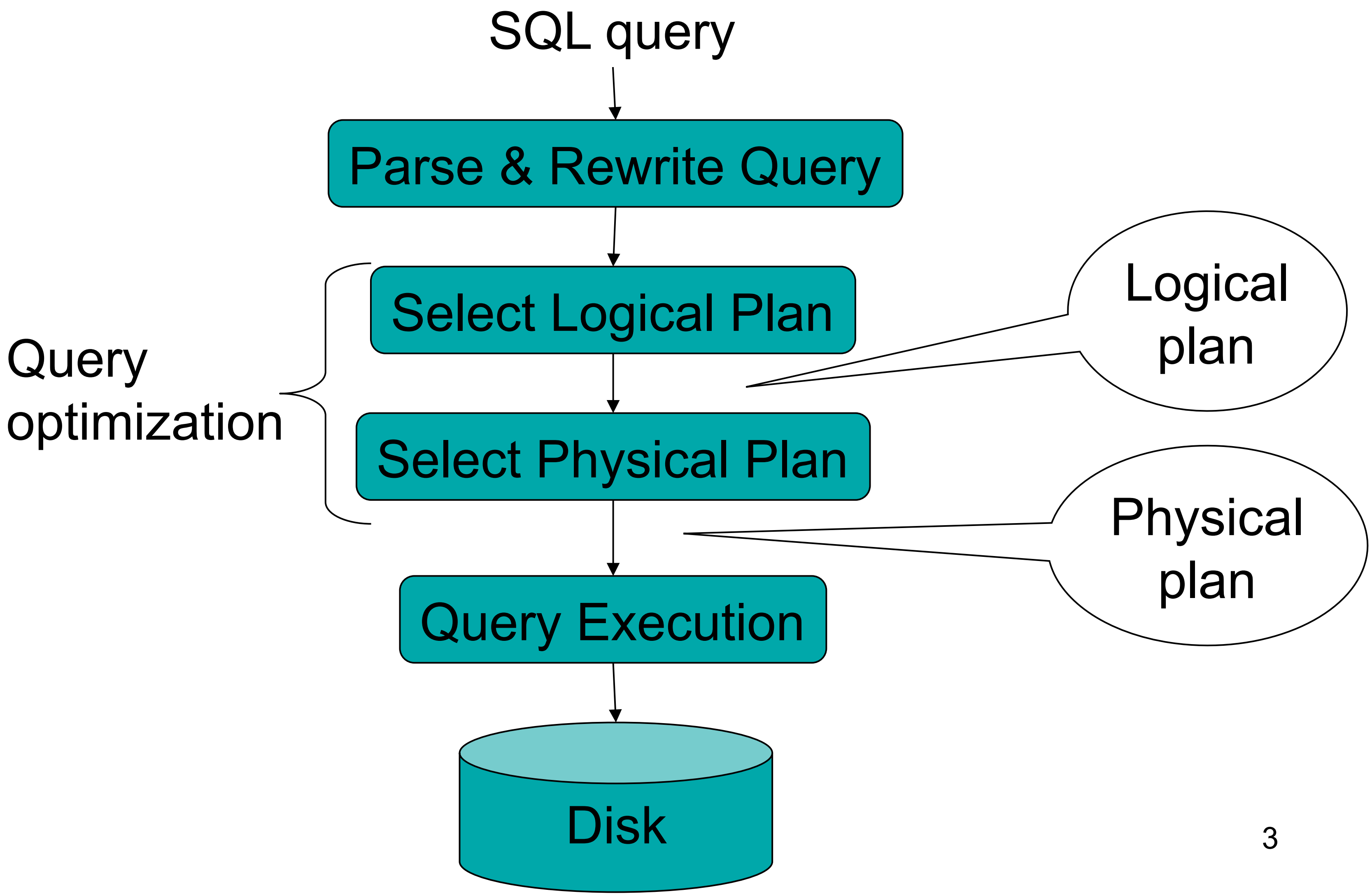
Lecture 17: Query execution

Wednesday, May 12, 2010

Outline of Next Few Lectures

- Query execution
- Query optimization

Steps of the Query Processor



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 )
```

Steps in Query Evaluation

- **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, etc.

Rewritten Version of Our Query

Original query:

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

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;
```

Continue with Query Evaluation

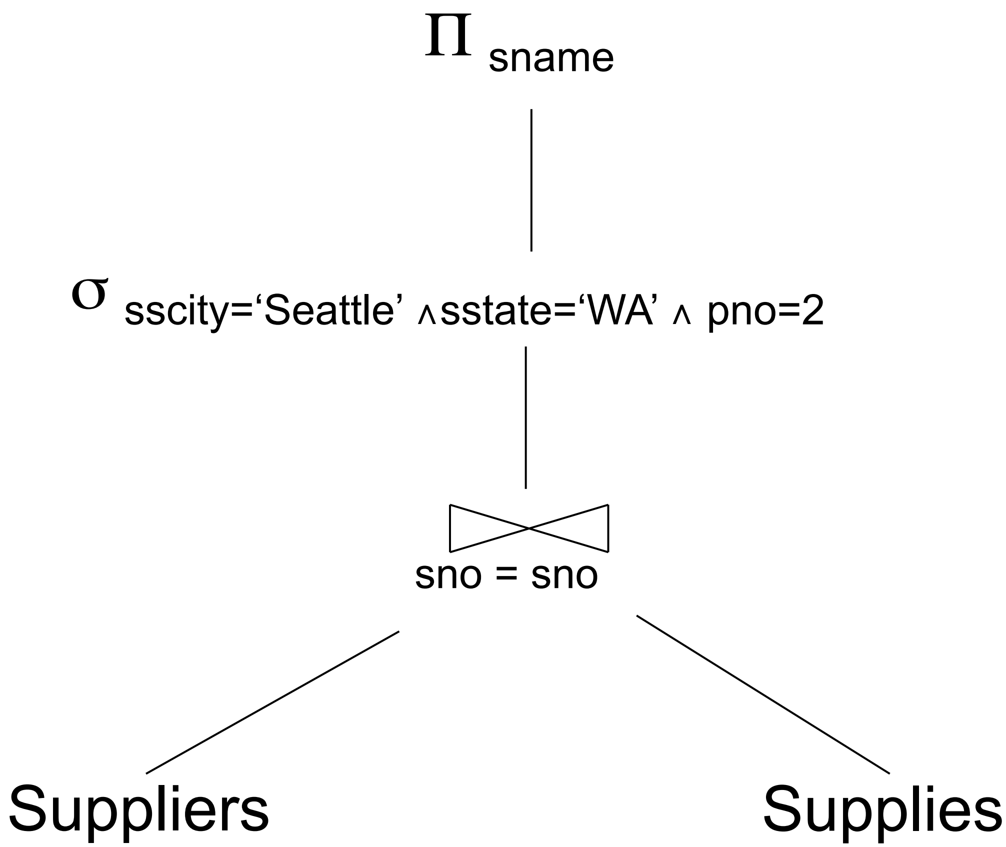
- **Step 3: Query optimization**
  - Find an efficient query plan for executing the query

- A **query plan** is
  - **Logical query plan**: an extended relational algebra tree
  - **Physical query plan**: with additional annotations at each node
    - Access method to use for each relation
    - Implementation to use for each relational operator

Extended Algebra Operators

- Union $\cup$, intersection $\cap$, difference $-$
- Selection σ
- Projection π
- Join $\bowtie$
- Duplicate elimination δ
- Grouping and aggregation γ
- Sorting τ
- Rename ρ

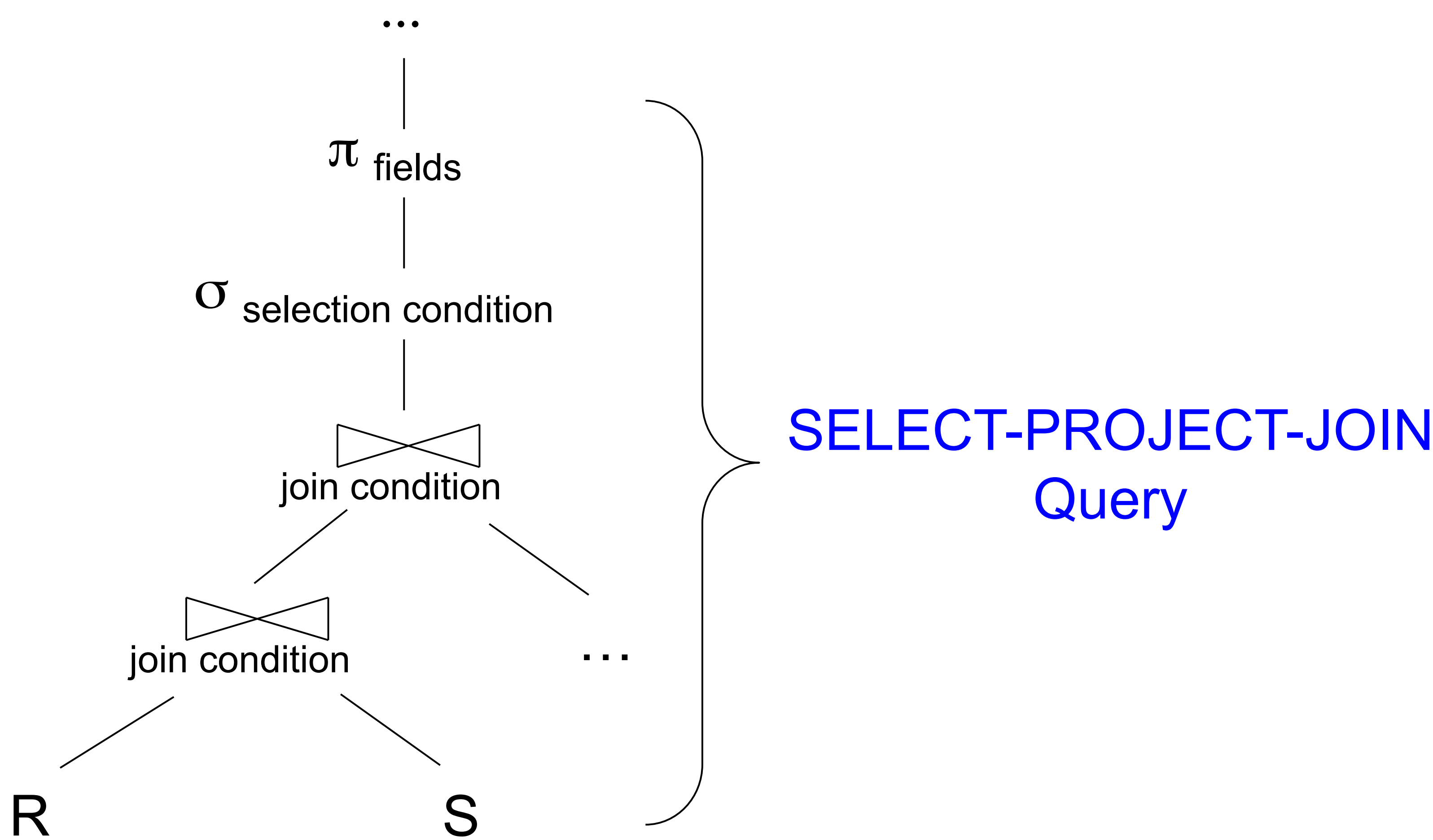
Logical Query Plan



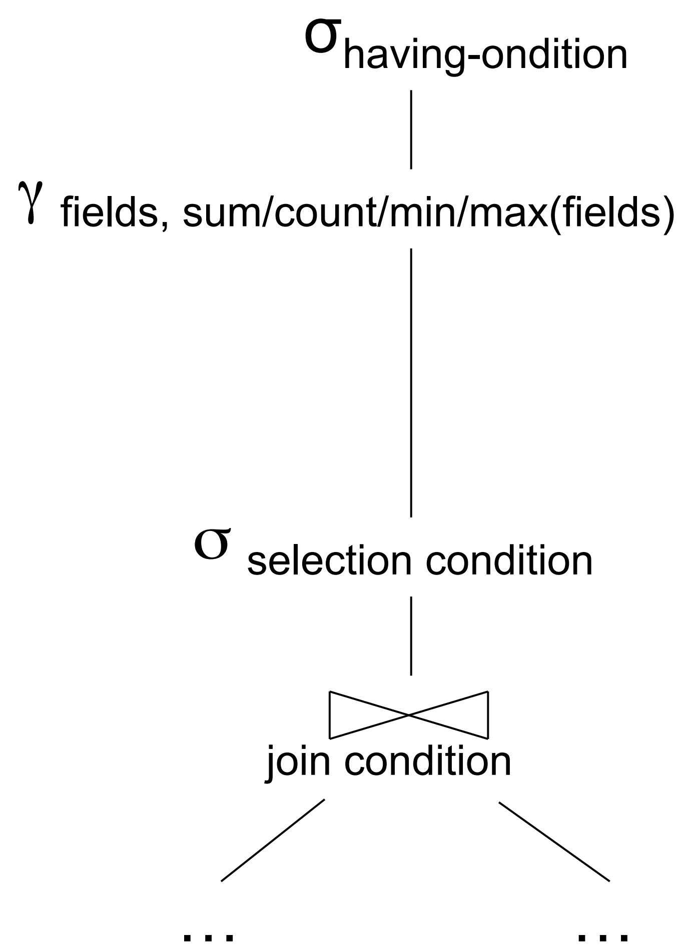
Query Block

- 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 (1/2)



Typical Plan For Block (2/2)



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

How about Subqueries?

```
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)

How about Subqueries?

```
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)

How about Subqueries?

```
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)

How about Subqueries?

Un-nesting

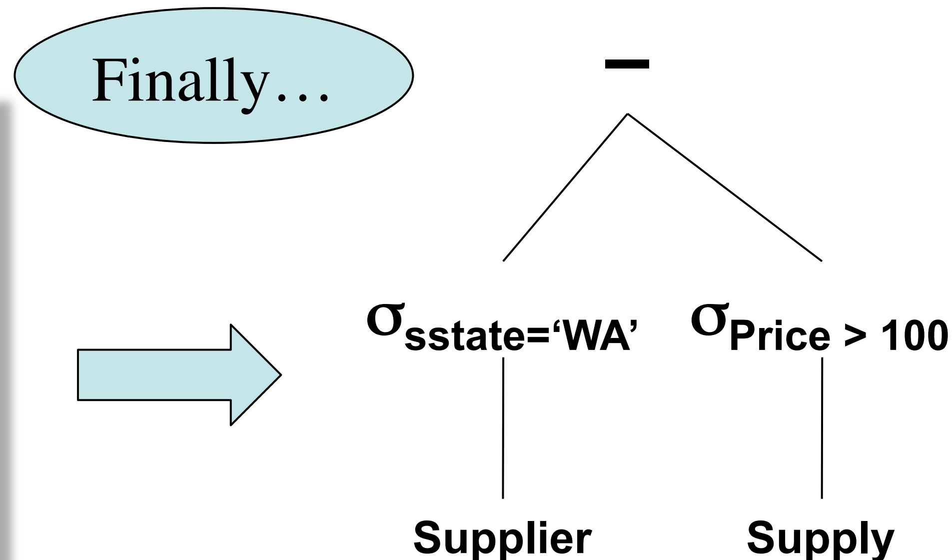
```
(SELECT  Q.sno
 FROM Supplier Q
 WHERE  Q.sstate = 'WA')
   EXCEPT
(SELECT P.sno
  FROM Supply P
  WHERE P.price > 100)
```

```
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
```

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

How about Subqueries?

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

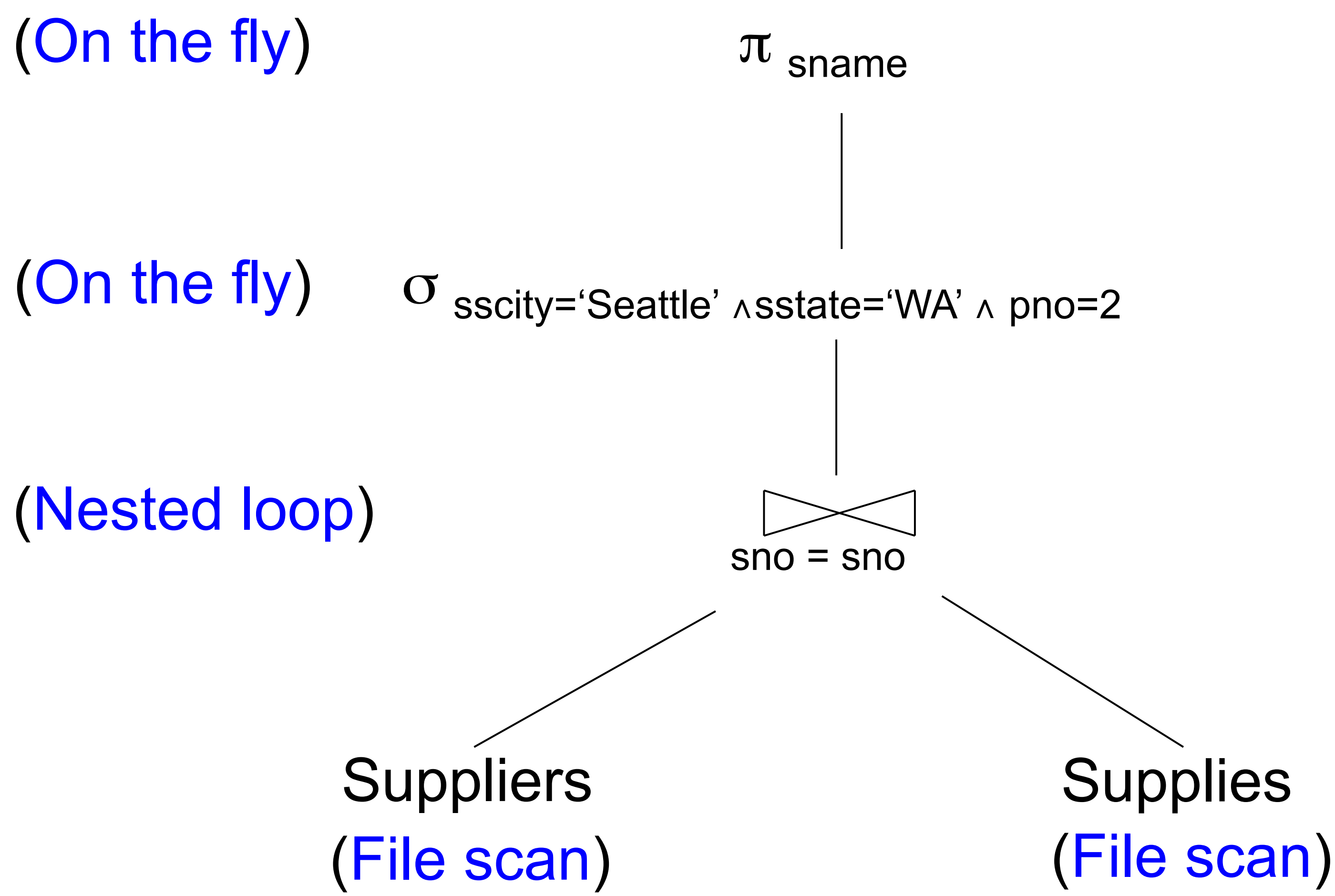


Physical Query Plan

- Logical query plan with extra annotations
- **Access path selection** for each relation
  - Use a file scan or use an index
- **Implementation choice** for each operator
- **Scheduling decisions** for operators

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

Physical Query Plan



Final Step in Query Processing

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

- What techniques are possible?
  - One thread per process
  - Iterator interface
  - Pipelined execution
  - Intermediate result materialization

Iterator Interface

- Each **operator implements this interface**
- Interface has only three methods
- **open()**
  - Initializes operator state
  - Sets parameters such as selection condition
- **get_next()**
  - Operator invokes get_next() recursively on its inputs
  - Performs processing and produces an output tuple
- **close()**: cleans-up state

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

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

Pipelined Execution

(On the fly) π_{sname}

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

(Nested loop) $\bowtie_{\text{sno = sno}}$

Suppliers (File scan)

Supplies (File scan)

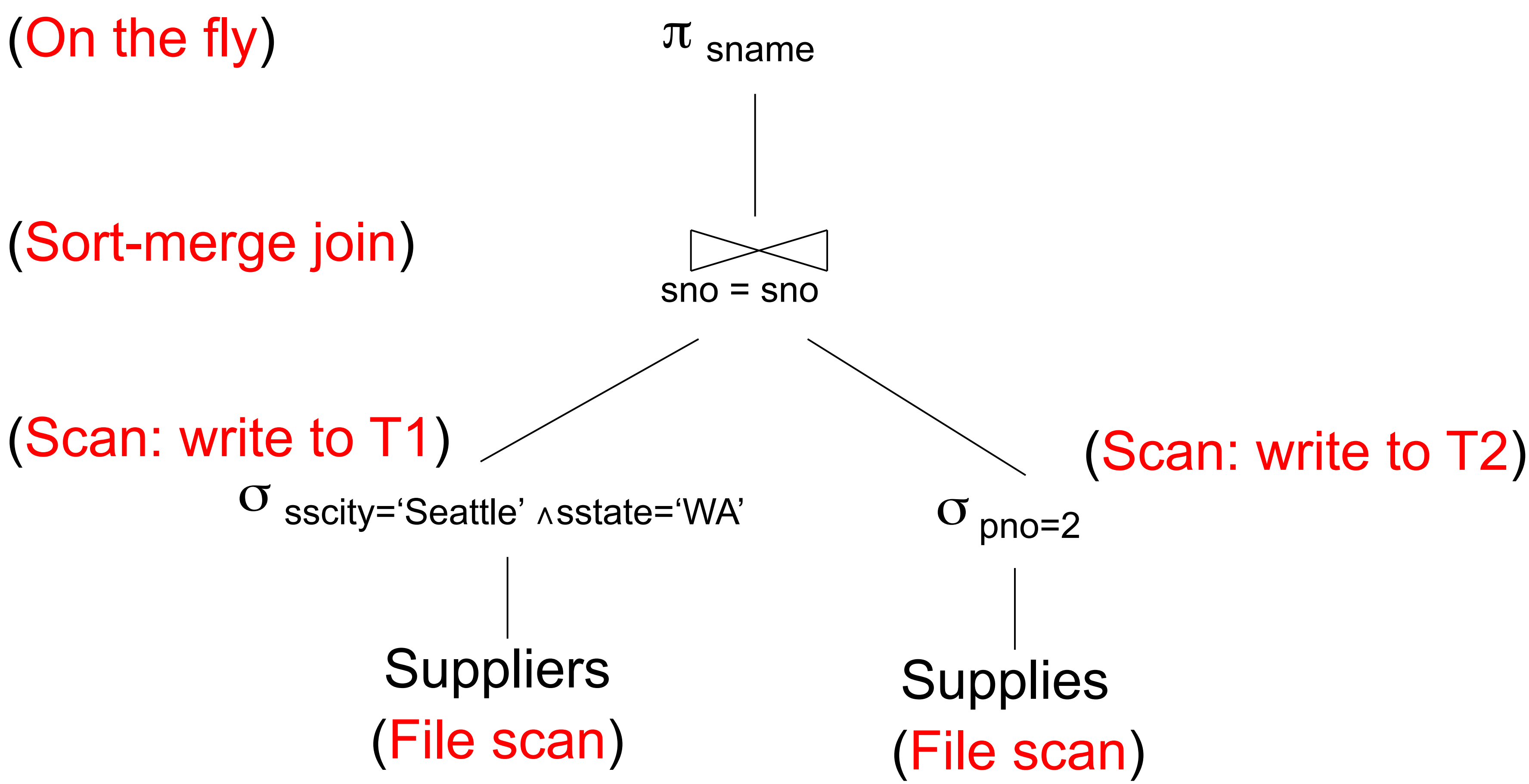
Intermediate Tuple Materialization

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

- No direct benefit but
- Necessary for some operator implementations
- When operator needs to examine the same tuples multiple times

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

Intermediate Tuple Materialization



Physical Operators

Each of the logical operators may have one or more implementations = physical operators

Will discuss several basic physical operators, with a focus on join

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

Question in Class

Logical operator:

Supply(sno,pno,price) $\bowtie_{pno=pno}$ **Part(pno,pname,psize,pcolor)**

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

1.
2.
3.

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

Question in Class

Logical operator:

Supply(sno,pno,price) $\bowtie_{pno=pno}$ **Part(pno,pname,psize,pcolor)**

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

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

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

1. Nested Loop Join

```
for S in Supply do {
    for  P in Part do {
        if (S.pno == P.pno) output(S,P);
    }
}
```

Supply = *outer relation*
Part = *inner relation*
Note: sometimes terminology is switched

Would it be more efficient to choose Part=inner, Supply=outer ?
What if we had an index on Part.pno ?

It’s more complicated…

- Each **operator implements this interface**
- **open()**
- **get_next()**
- **close()**

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

Nested Loop Join Revisited

```
open ( ) {
  Supply.open( );
  Part.open( );
  S = Supply.get_next( );
}
```

```
close ( ) {
  Supply.close ( );
  Part.close ( );
}
```

```
get_next( ) {
  repeat {
    P= Part.get_next( );
    if (P== NULL)
      { Part.close();
        S= Supply.get_next( );
        if (S== NULL) return NULL;
        Part.open( );
        P= Part.get_next( );
      }
  until (S.pno == P.pno);
  return (S, P)
}
```

ALL operators need to be implemented this way !

BRIEF Review of Hash Tables

A (naïve) hash function:

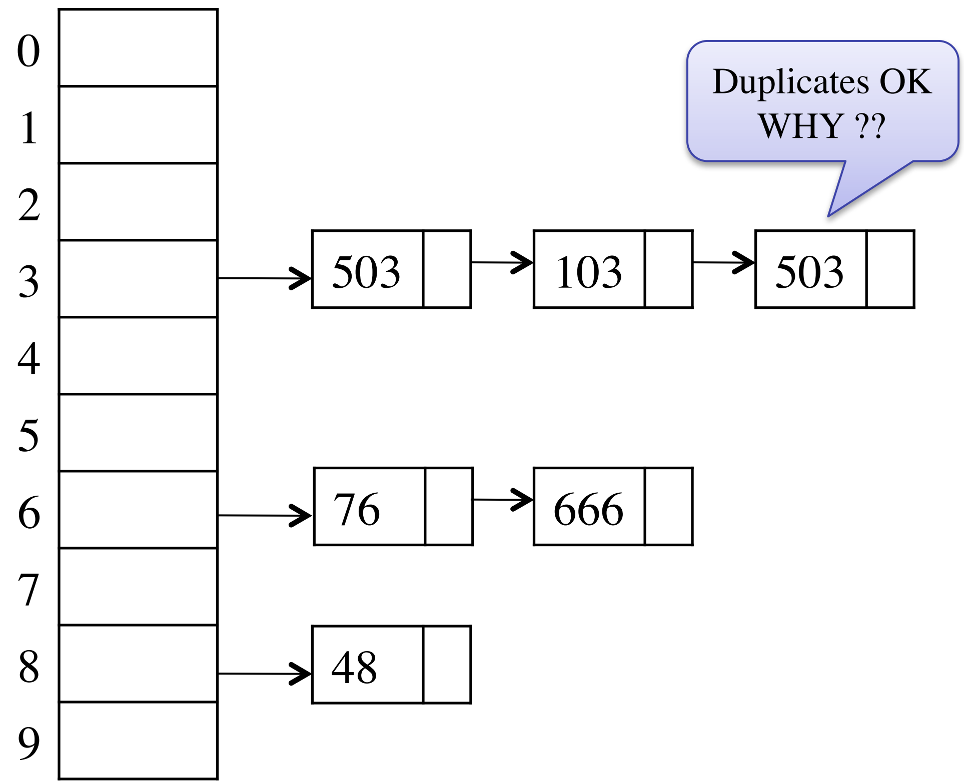
$$h(x) = x \bmod 10$$

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

2. Hash Join (main memory)

Build phase

```
for S in Supply do  insert(S.pno, S);

for P in Part do {
   LS = find(P.pno);
   for S in LS do { output(S, P); }
}
```

Probing

Supply=outer
Part=inner

Recall: need to rewrite as open, get_next, close

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

3. Merge Join (main memory)

```
Part1 = sort(Part, pno);
Supply1 = sort(Supply,pno);
P=Part1.get_next(); S=Supply1.get_next();

While (P!=NULL and S!=NULL) {
   case:
      P.pno > S.pno:    P = Part1.get_next( );
      P.pno < S.pno:    S = Supply1.get_next();
      P.pno == S.pno { output(P,S);
                        S = Supply1.get_next();
                      }
}
```

Why ???