CSE 544
Principles of Database Management Systems

Magdalena Balazinska

Fall 2007

Lecture 7 - Query execution

References

- **Generalized Search Trees for Database Systems**. J. M. Hellerstein, J. F. Naughton and A. Pfeffer. VLDB 1995. [To finish talking about GiST]
- **Query evaluation techniques for large databases.**
  G. Graefe. ACM Computing Survey 25(2). 1993. **Sec 1**.
- **Database management systems.**
  Ramakrishnan and Gehrke.
  Third Ed. **Chapter 12**.

Outline

- **Finish talking about GiST**

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

- **Operator implementations (part 1)**

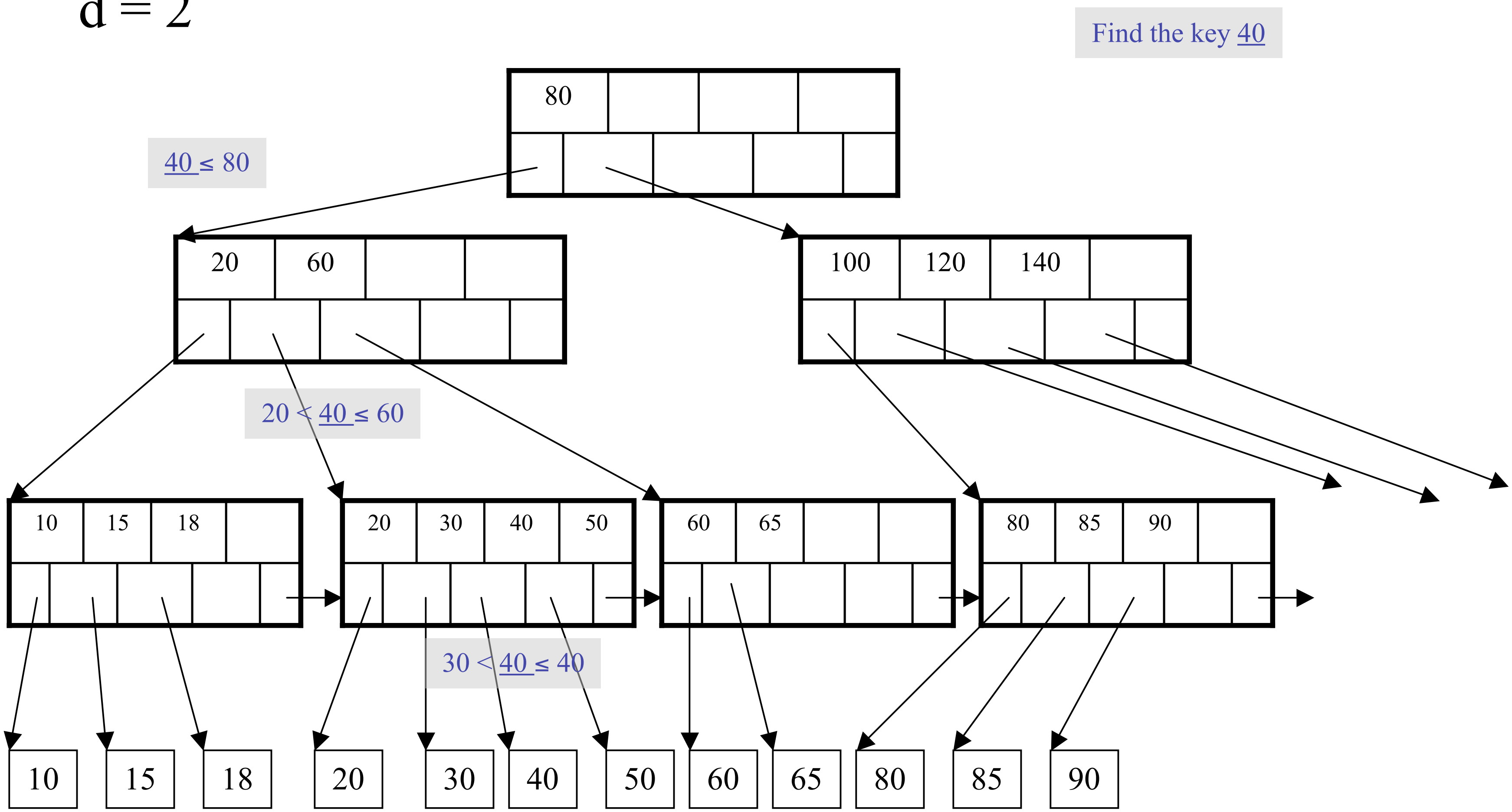
Generalized Search Tree (GiST)

- Goal: facilitate database extensibility
  - When adding a new data type
  - Want to add indexes for the data type

- Overview
  - GiST is an index structure
  - Basically, this is a template for indexes
  - Supports extensible set of queries and data types

B+ Tree Example

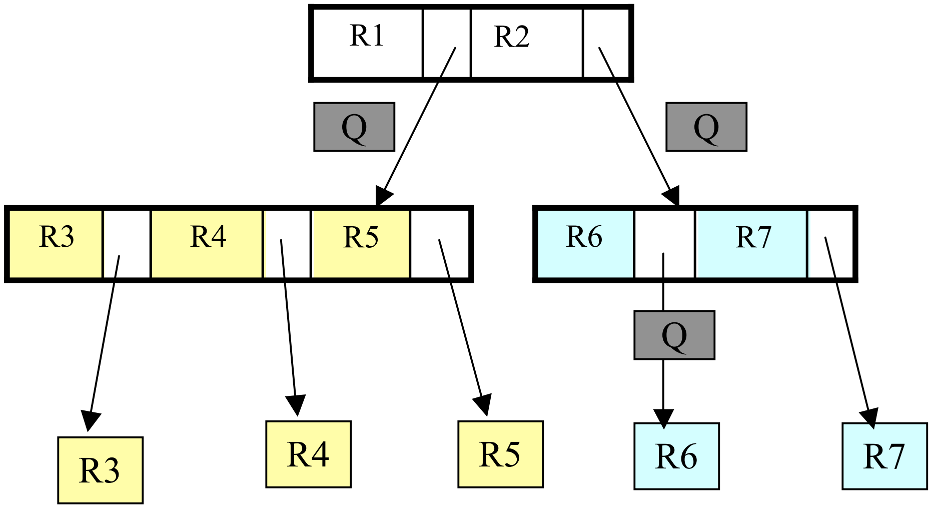
d = 2

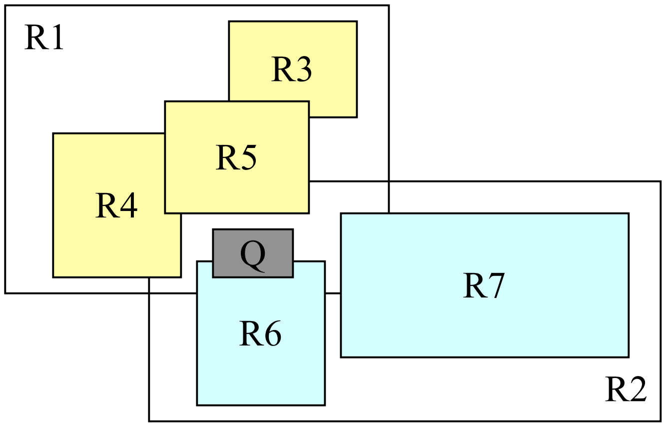


R-Tree Example

Designed for spatial data

Search key values are bounding boxes





For insertion: at each level, choose child whose bounding box needs least enlargement (in terms of area)

GiST Key Insights

Canonical database search tree

- Balanced tree with high fanout
- Leaf nodes contain pointers to actual data
- Leaf nodes stored as a linked list
- Internal nodes used as a directory
  - Contain <key,pointers> pairs
  - If key consistent with query, data may be found if we follow pointer
  - **Generalized search key**: predicate that holds for each entry below key
    - B+-tree key is pair of integers <a,b> and predicate is Contains([a,b),v)
    - R-tree key is bounding box and predicate is also containment test
  - **Generalized search tree**: hierarchy of partitions

GiST Key Methods: Consistent

- Consistent(E,q)
  - Entry E = (p, ptr) and query predicate q
  - Returns false if $p \wedge q$ can be guaranteed unsatisfiable
  - Returns true otherwise

- See Algo Search(R,q) [also FindMin(R,q) and Next(R,q,E)]

GiST Key Methods: Consistent

- In a B+-tree, query predicates q can be either
  - Contains([x,y), v) returns true if $x <= v < y$ and false otherwise
  - Equal(x,v) returns true if $x = v$ and false otherwise
- In a B+-tree, Consistent(E,q)
  - p = Contains($[x_p, y_p)$, v)
  - (1) q = Contains($[x_q, y_q)$, v) or (2) q = Equal(x_q,v)
  - For (1), return true if $(x_p < y_q) \wedge (y_p > x_q)$
  - For (2), return true if $x_p <= x_q < y_p$
- In R-tree, Consistent returns true if bounding boxes overlap

GiST Key Methods

- Penalty
  - Used during insert operations to pick subtree where to insert
  - See algorithms Insert(R,E,I) and ChooseSubtree(R,E,I)
  - B+-tree: returns zero when value to insert falls within subtree range
  - R-tree: returns change in area

- PickSplit
  - Used to split nodes during insert operations
  - See algorithm Split(R,N,E)
  - B+-tree: half the entries go into left group and half into right group
  - R-tree: e.g., minimize total area of bounding boxes after split

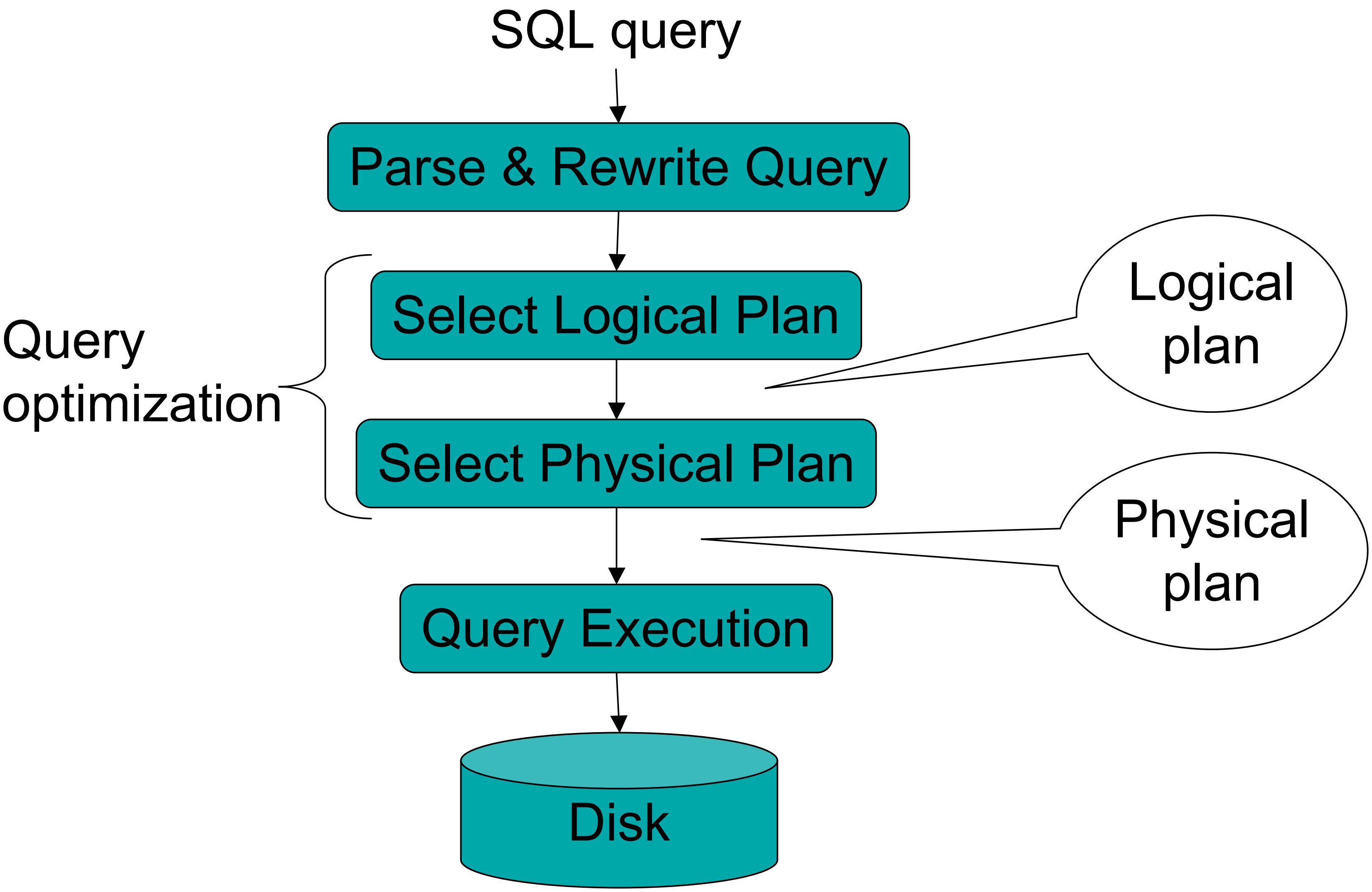
GiST Key Methods

- Union
  - Once a key is inserted, need to adjust predicates at parent nodes
  - See algorithm AdjustKeys(R,N)
  - B+-tree: computes interval that covers all given intervals
  - R-tree: computes bigger bounding box
- Compress/Decompress: for storage performance

Outline

- **Finish talking about GiST**
- **Steps involved in processing a query**
  - Logical query plan
  - Physical query plan
  - Query execution overview
- **Operator implementations (part 1)**

Query Evaluation Steps



SQL query

Parse & Rewrite Query

Query optimization

Select Logical Plan

Logical plan

Select Physical Plan

Physical plan

Query Execution

Disk

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

- **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

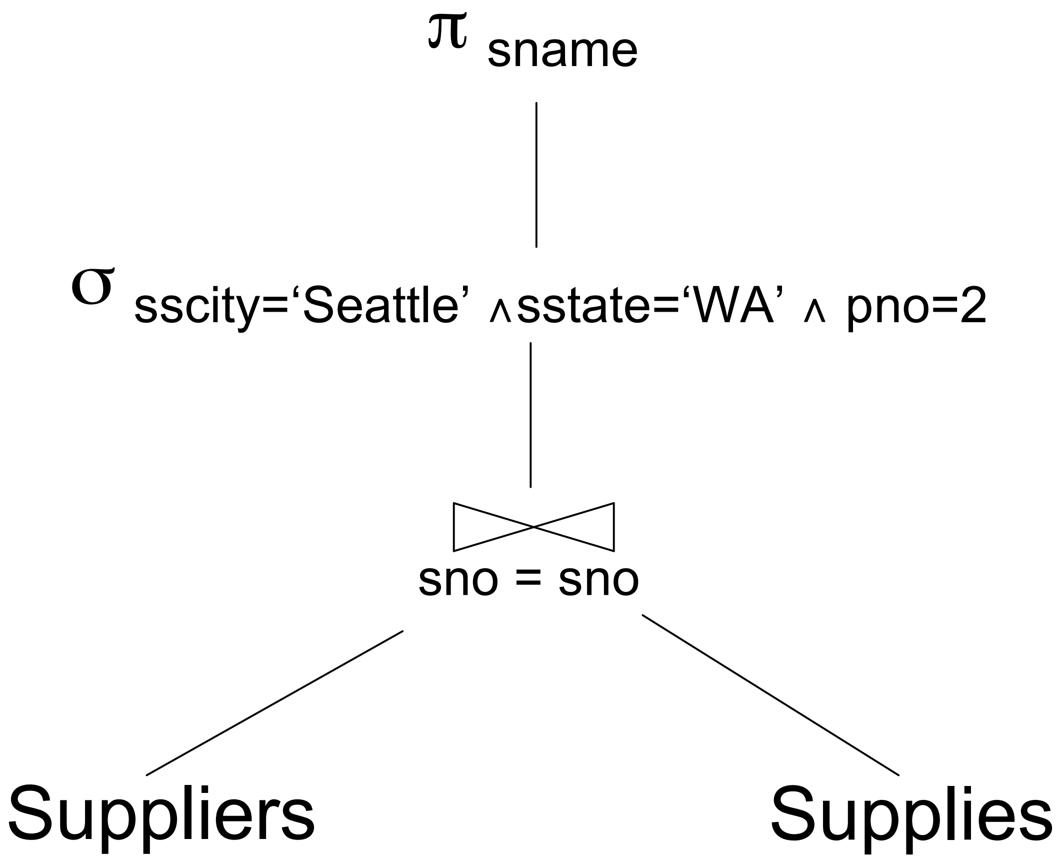
- **Step 3: Query optimization**
  - Find an efficient query plan for executing the query
  - We will spend a whole lecture on this topic

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



π_{sname}

$\sigma_{\text{sscity='Seattle'} \wedge \text{sstate='WA'} \wedge \text{pno=2}}$

$\bowtie_{\text{sno = sno}}$

Suppliers

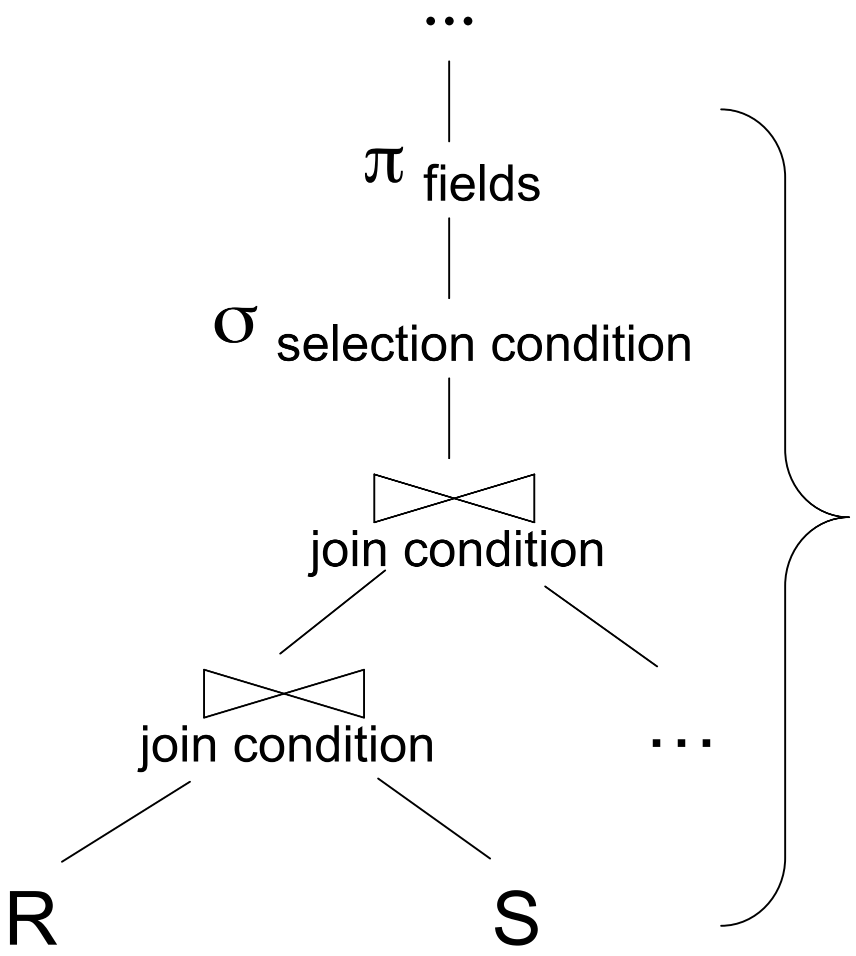
Supplies

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)



…

π_{fields}

$\sigma_{\text{selection condition}}$

$\bowtie$ join condition

$\bowtie$ join condition

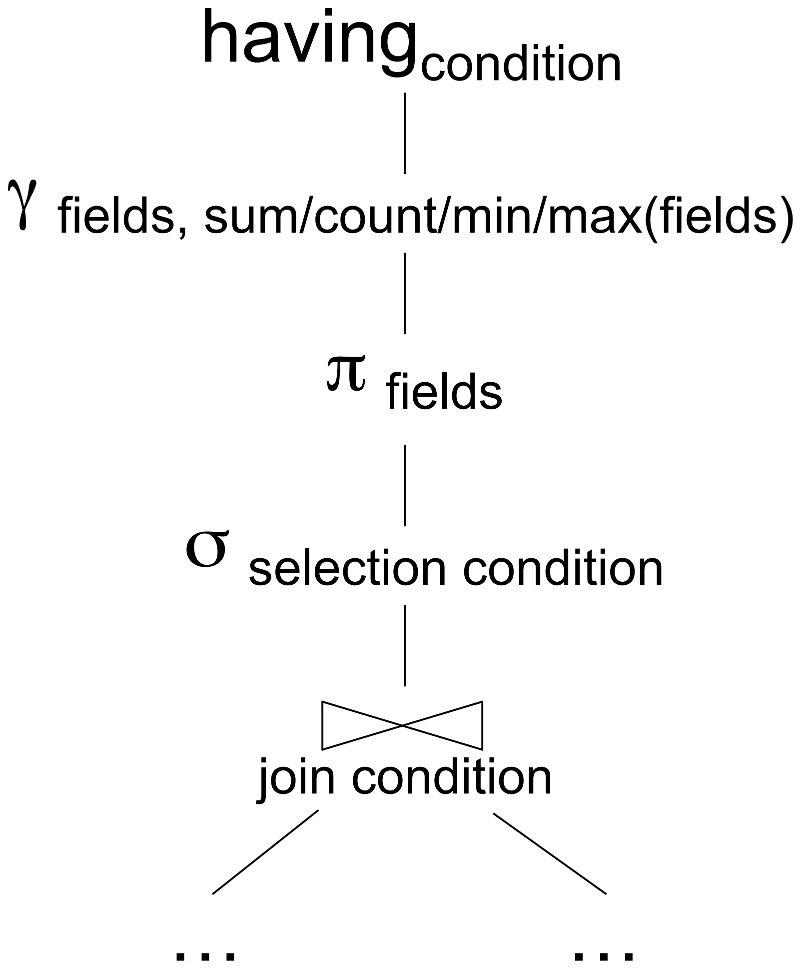
…

R

S

SELECT-PROJECT-JOIN Query

Typical Plan For Block (2/2)



$\text{having}_{\text{condition}}$

$\gamma_{\text{fields, sum/count/min/max(fields)}}$

π_{fields}

$\sigma_{\text{selection condition}}$

$\bowtie_{\text{join condition}}$

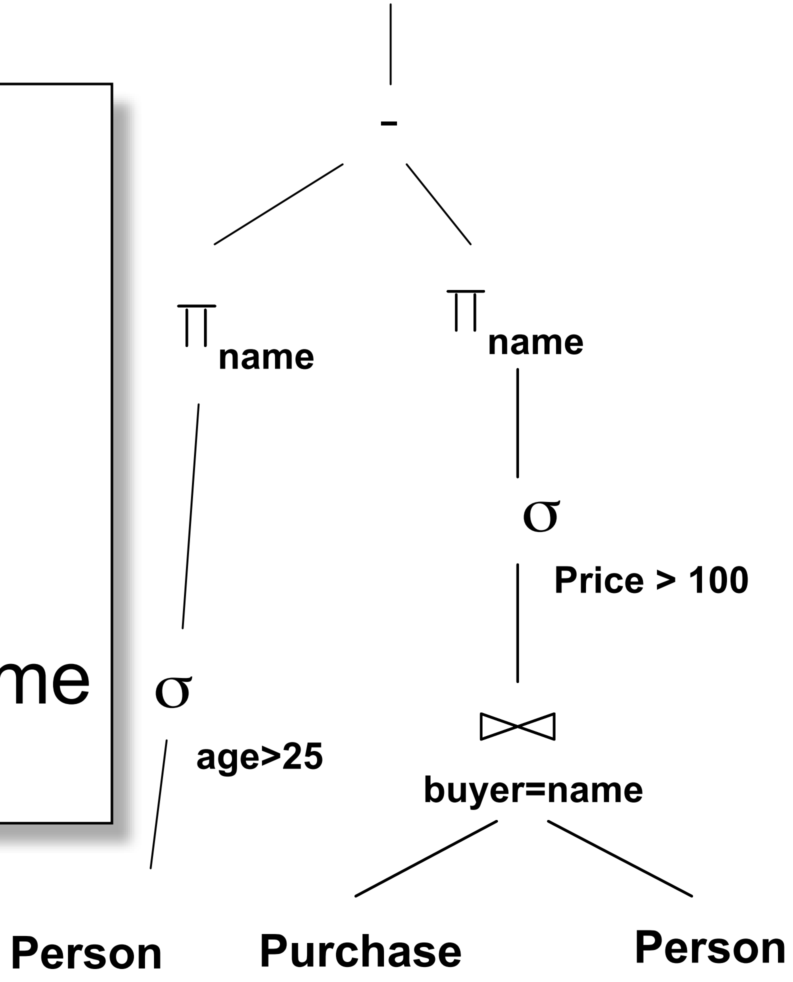
… …

How about Subqueries?

```
SELECT  Q.name
FROM Person Q
WHERE  Q.age > 25
    and not exists
       SELECT *
       FROM Purchase P
       WHERE P.buyer = Q.name
               and P.price > 100
```

How about Subqueries?

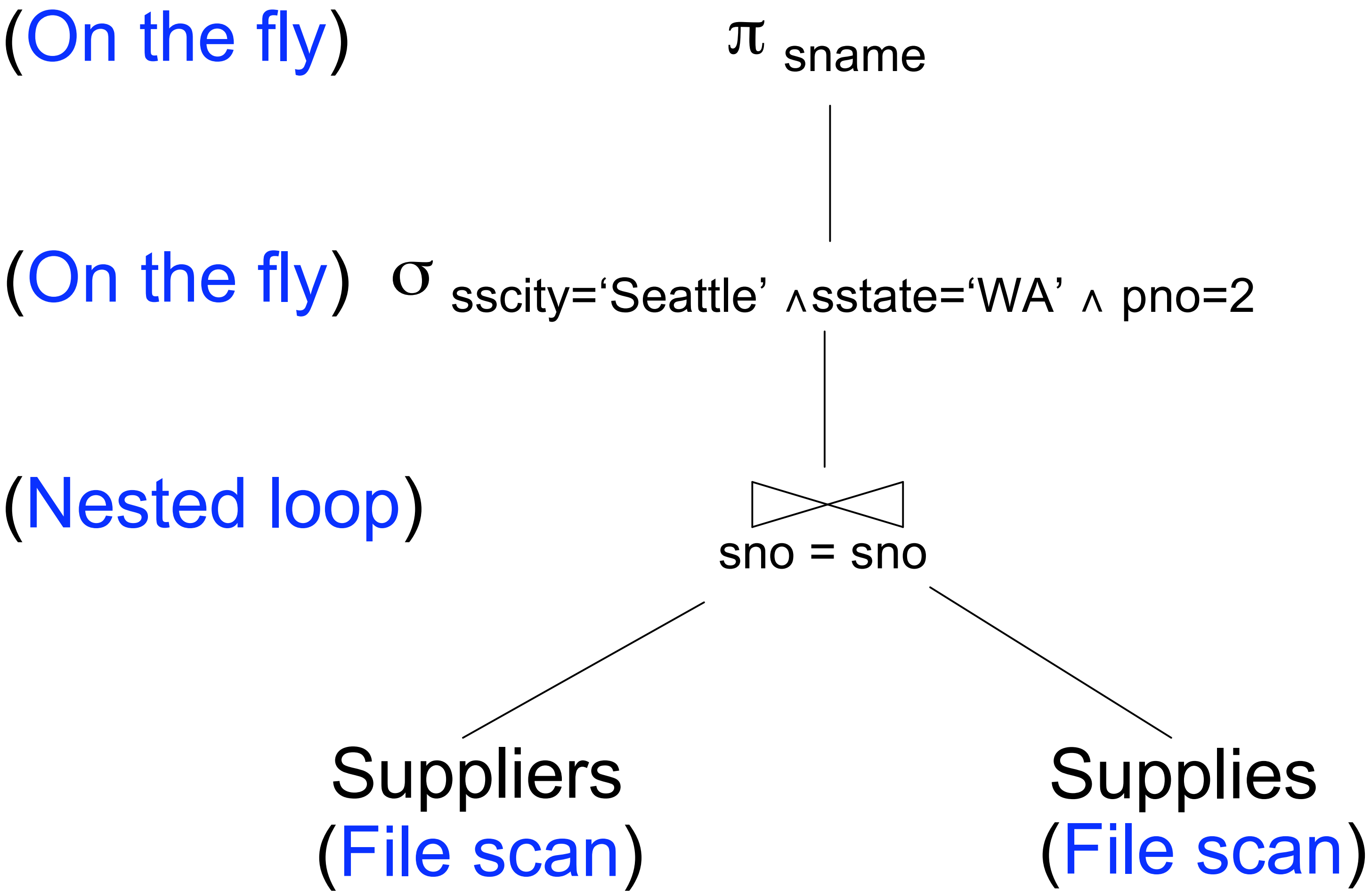
```
SELECT  Q.name
FROM Person Q
WHERE  Q.age > 25
    and not exists
       SELECT *
       FROM Purchase P
       WHERE P.buyer = Q.name
               and P.price > 100
```



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

Physical Query Plan



(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)

Final Step in Query Processing

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

- What techniques are possible (paper Sec. 1)?
  - 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()**: clean-up state
- **Examples: Table 1 in the paper**

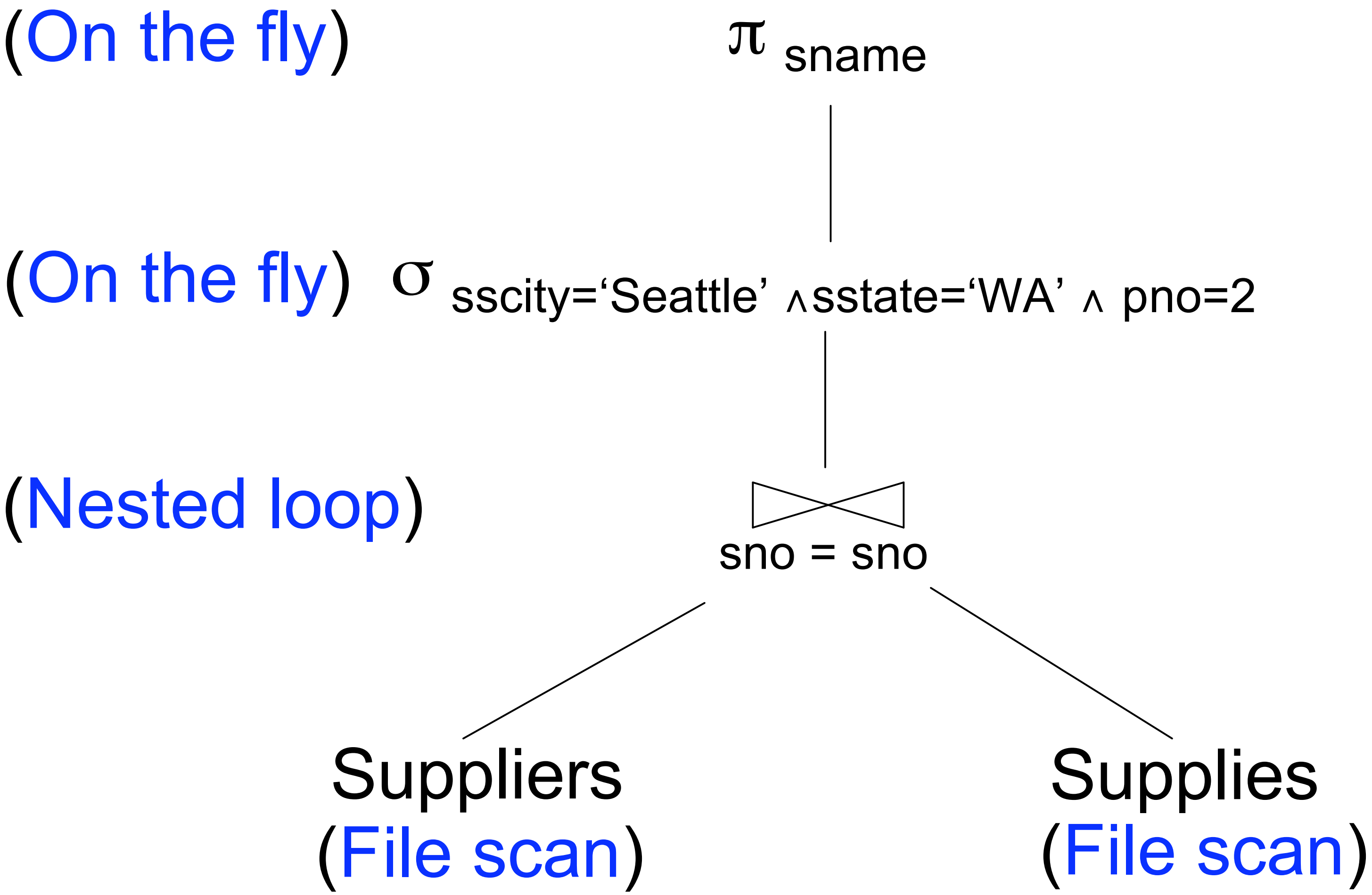
Pipelined Execution

- 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



(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)

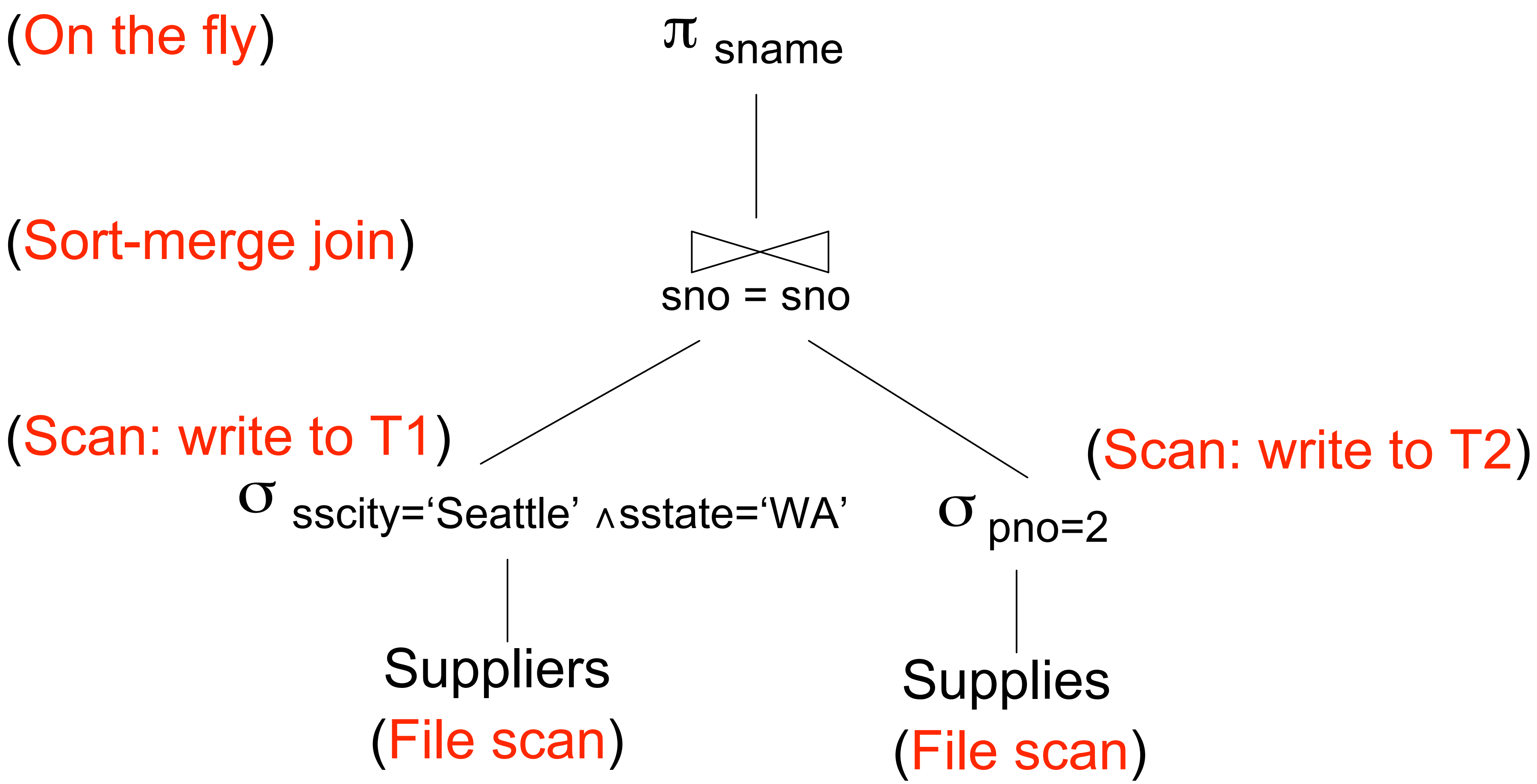
Supplies (File scan)

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

Intermediate Tuple Materialization



(On the fly) π_{sname}

(Sort-merge join) $\bowtie_{\text{sno = sno}}$

(Scan: write to T1) $\sigma_{\text{sscity='Seattle'} \wedge \text{sstate='WA'}}$

(Scan: write to T2) $\sigma_{\text{pno=2}}$

Suppliers
(File scan)

Supplies
(File scan)

Outline

- **Finish talking about GiST**
- **Steps involved in processing a query**
  - Logical query plan
  - Physical query plan
  - Query execution overview
- **Operator implementations (part 1)**

Cost Parameters

- In database systems the data is on disk
- **Cost = total number of I/Os**

- 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**

Cost

- Cost of an operation = number of disk I/Os to
  - read the operands
  - compute the result

- Cost of writing the result to disk is *not included*
  - Need to count it separately when applicable

Notions of Clustering

- **Clustered-file organization** (aka co-clustering)
  - Tuples of one relation R are placed with a tuple of another relation S with a common value
- **Clustered relation**
  - Tuples of relation are stored on blocks predominantly devoted to storing that relation
  - Sometimes also called “clustered file organization”
- **Clustered index (aka clustering index)**
  - When ordering of data records is close to the ordering of data entries in the index

Cost Parameters

- Clustered relation R:
  - Blocks consists mostly of records from this table
  - $B(R) \approx T(R)$ / blockSize
- Unclustered relation R:
  - Its records are placed on blocks with other tables
  - When R is unclustered: $B(R) \approx T(R)$
- When a is a key, $V(R,a) = T(R)$
- When a is not a key, $V(R,a)$

Cost of Scanning a Table

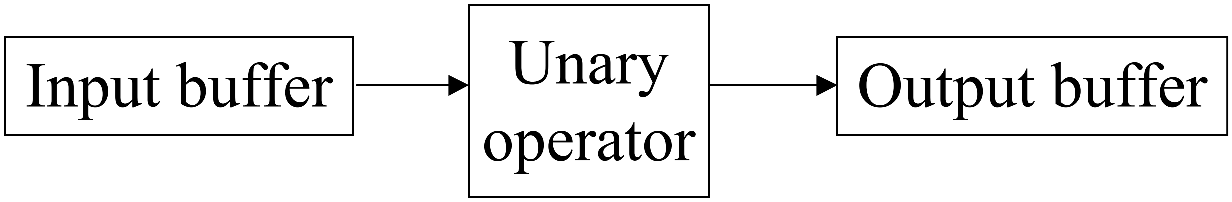
- Clustered relation:
  - Result may be unsorted: B(R)
  - Result needs to be sorted: 3B(R)

- Unclustered relation
  - Unsorted: T(R)
  - Sorted: T(R) + 2B(R)

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



Join Algorithms

- Logical operator:
  - Product(pname, cname) ⋈ Company(cname, city)

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

- 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) when S is clustered
- Cost: B(R) + T(R) T(S) when S is unclustered

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) if S is clustered
- Cost: B(R) + B(R)T(S) if S is unclustered

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

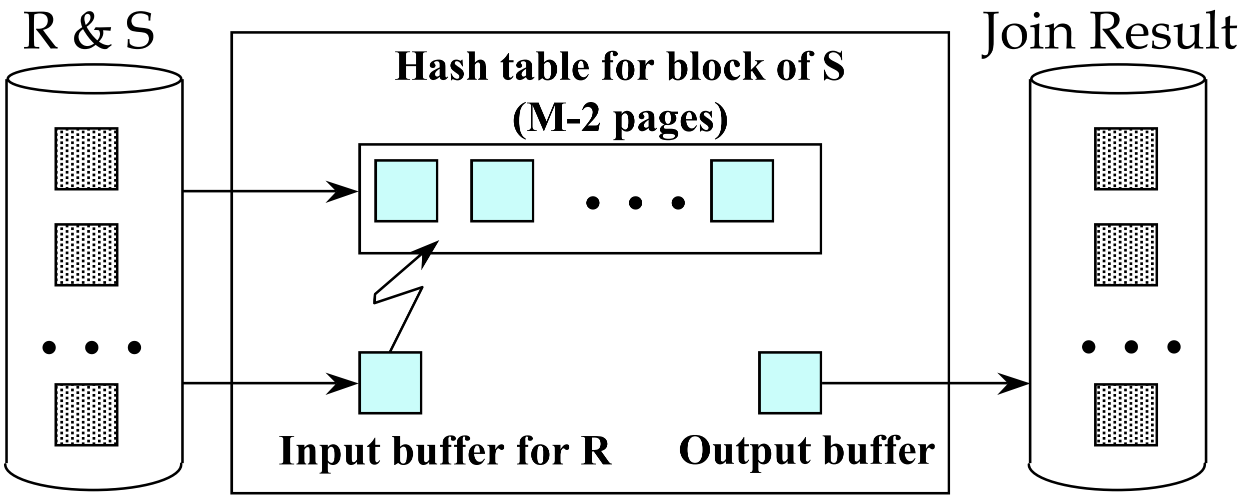
  – B(R) = 1000, B(S) = 6, M = 4

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)
```

Nested Loop Joins



Nested Loop Joins

- Cost of block-based nested loop join
  - Read S once: cost B(S)
  - Outer loop runs B(S)/(M-2) times, and each time need to read R: costs 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

One-pass Algorithms

Duplicate elimination $\delta(R)$

- Need to keep tuples in memory
- When new tuple arrives, need to compare it with previously seen tuples
- Balanced search tree or hash table
- Cost: $B(R)$
- Assumption: $B(\delta(R)) <= M$

One-pass Algorithms

Grouping:

Product(name, department, quantity)

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

How can we compute this in main memory ?

One-pass Algorithms

- Grouping: $\gamma_{\text{department, sum(quantity)}}$ (R)

- Need to store all departments in memory
- Also store the sum(quantity) for each department
- Balanced search tree or hash table
- Cost: B(R)
- Assumption: number of depts fits in memory