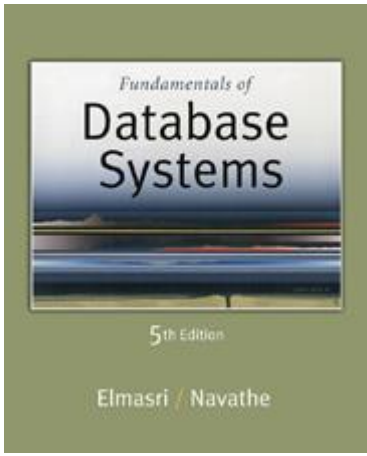
Lecture 5

Algorithms for Query Processing and Optimization



Copyright © 2004 Pearson Education, Inc.

Lecture Outline:

- Introduction to Query Processing

1. Translating SQL Queries into Relational Algebra
2. Algorithms for External Sorting
3. Algorithms for SELECT and JOIN Operations
4. Algorithms for PROJECT and SET Operations
5. Implementing Aggregate Operations and Outer Joins
6. Combining Operations using Pipelining
7. Using Heuristics in Query Optimization
8. Using Selectivity and Cost Estimates in Query Optimization
9. Overview of Query Optimization in Oracle
10. Semantic Query Optimization

Copyright © 2004 Ramez Elmasri and Shamkant B. Navathe

Literature

- These slides are based on
  - Chapters 15 in:

    El Masri, R and Navathe, S., "*Fundamentals of Database Systems*", 5th Edition, Pearson Int. Edition, Addison Wesley, ISBN: 0-321-4150-X (2007).

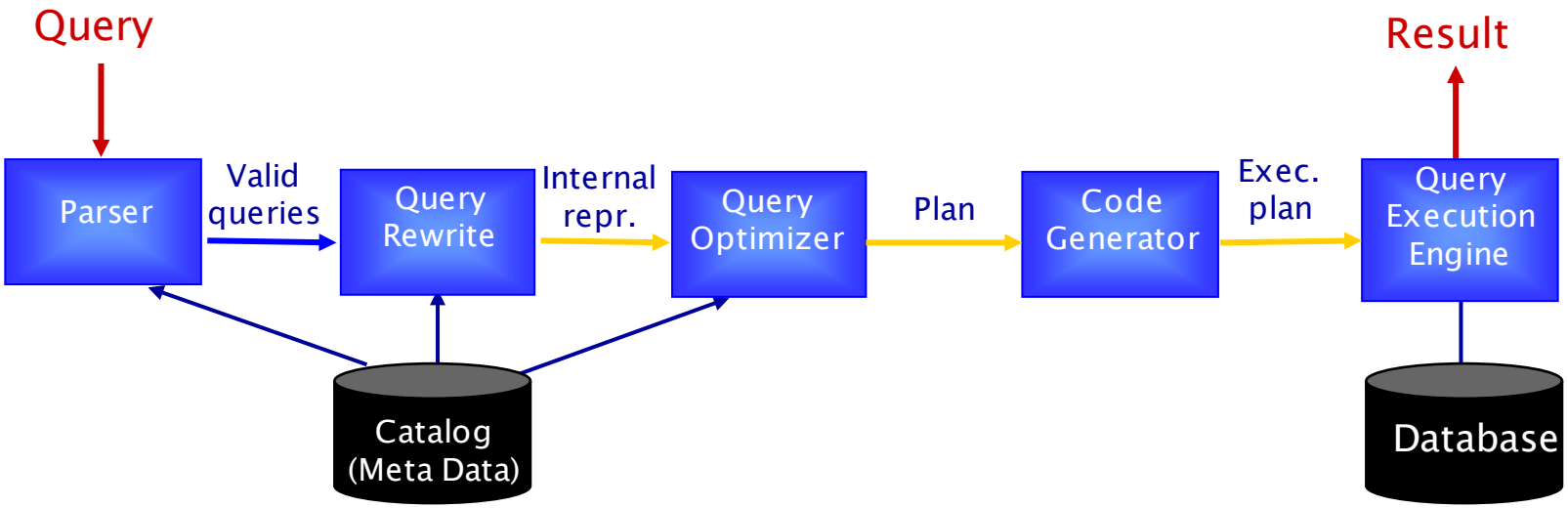
  - Chapter s 13 and 14 in:

    Henry F. Korth, Abraham Silberschatz, *Database System Concepts*, *(5th Ed.).* McGraw-Hill, 2006

Copyright © 2007 Ramez Elmasri and Shamkant B. Navathe

Basic Steps in Query Processing

- Deals with developing algorithms that analyze queries to generate a *good* execution plan that defines a sequence of steps for query evaluation, each step corresponds to one relational operation and its evaluation method.



Phases of Query Processing

Copyright © 2007 Ramez Elmasri and Shamkant B. Navathe

Basic Steps in Query Processing

- Parsing and translation
  - Parser checks syntax, verifies relations
  - translate the query into its internal form.
    - Query Graph
    - Query Tree

- Query optimization

  No single way to do things:
  - • Many ways to express the same query
  - • Many ways to translate into relational algebra
  - • Many ways to evaluate the relational algebra expression (e.g., order of operations)
  - • Many algorithms to execute an individual relational algebra operation

  Each of these choices has a cost — the process of query optimization **seeks to minimize this cost**
  - .

Copyright © 2007 Ramez Elmasri and Shamkant B. Navathe

Basic Steps in Query Processing

- Execution
  - The query-execution engine takes a query-evaluation plan, executes that plan, and returns the query answers to the user

Copyright © 2007 Ramez Elmasri and Shamkant B. Navathe

Translating SQL Queries into Relational Algebra

- **Query block:** the basic unit that can be translated into the algebraic operators and optimized.
- A query block contains a single SELECT-FROM-WHERE expression, as well as GROUP BY and HAVING clause if these are part of the block.
- **Nested queries** within a query are identified as separate query blocks.
- Because SQL includes aggregate operators—such as MAX, MIN, these operators also be included in the extended algebra

Copyright © 2007 Ramez Elmasri and Shamkant B. Navathe

Translating SQL Queries into Relational Algebra

- Select Operation (σ) It selects tuples that satisfy the given predicate from a relation.

$$\sigma_{subject = "database"}(Books)$$

Selects tuples from books where subject is 'database'.

- Project Operation (∏)

It projects column(s) that satisfy a given predicate.

$$\Pi_{subject,\ author}\ (Books)$$

Selects and projects columns named as subject and author from the relation Books

Copyright © 2007 Ramez Elmasri and Shamkant B. Navathe

Translating SQL Queries into Relational Algebra

- Union Operation (∪) It performs binary union between two given relations.
  - Two relations must have the same number of attributes.
  - Attribute domains must be compatible.
  - Duplicate tuples are automatically eliminated.

$$\Pi_{\text{author}}(\text{Books}) \cup \Pi_{\text{author}}(\text{Articles})$$

Projects the names of the authors who have either written a book or an article or both

Copyright © 2007 Ramez Elmasri and Shamkant B. Navathe

Translating SQL Queries into Relational Algebra

- Set Difference (−)

The result of set difference operation is tuples, which are present in one relation but are not in the second relation.

$$\prod_{author}(Books) - \prod_{author}(Articles)$$

Provides the name of authors who have written books but not articles.

- Cartesian Product (X)

Combines information of two different relations into one.

$$\sigma_{author = 'tutorialspoint'}(Books \times Articles)$$

Yields a relation, which shows all the books and articles written by tutorials point. This operation is very expensive and should be avoided if possible.

Copyright © 2007 Ramez Elmasri and Shamkant B. Navathe

Translating SQL Queries into Relational Algebra

- Intersection
  - Sort the two relations on the same attributes.
  - Scan and merge both sorted files concurrently, keep in the merged results only those tuples that appear in both relations.

Copyright © 2007 Ramez Elmasri and Shamkant B. Navathe

Translating SQL Queries into Relational Algebra (2)

```
SELECT    LNAME, FNAME
FROM      EMPLOYEE
WHERE     SALARY > (    SELECT MAX (SALARY)
                        FROM   EMPLOYEE
                        WHERE  DNO = 5);
```

```
SELECT    LNAME, FNAME
FROM      EMPLOYEE
WHERE     SALARY > C
```

```
SELECT MAX (SALARY)
FROM   EMPLOYEE
WHERE  DNO = 5
```

$\pi_{LNAME,\ FNAME}\ (\sigma_{SALARY>C}(EMPLOYEE))$

$\mathcal{F}_{MAX\ SALARY}\ (\sigma_{DNO=5}\ (EMPLOYEE))$

Copyright © 2007 Ramez Elmasri and Shamkant B. Navathe

Basic Steps in Query Processing : Optimization

- A relational algebra expression may have many equivalent expressions
  - E.g., $\sigma_{balance<2500}(\Pi_{balance}(account))$

    is equivalent to

    $\Pi_{balance}(\sigma_{balance<2500}(account))$
- Each relational algebra operation can be evaluated using one of several different algorithms
  - Correspondingly, a relational-algebra expression can be evaluated in many ways.
- Annotated expression specifying detailed evaluation strategy is called an **evaluation-plan**.

Copyright © 2007 Ramez Elmasri and Shamkant B. Navathe

Basic Steps: Optimization (2)

- **Query Optimization**: Amongst all equivalent evaluation plans choose the one with lowest cost.
  - Cost is estimated using statistical information from the database catalog
    - e.g. number of tuples in each relation, size of tuples, etc.
- In this lecture we will study:
  - How to measure query costs
  - Algorithms for evaluating relational algebra operations
  - How to combine algorithms for individual operations in order to evaluate a complete expression
  - How to optimize queries, that is, how to find an evaluation plan with lowest estimated cost

Copyright © 2007 Ramez Elmasri and Shamkant B. Navathe

1st Way: Measures of Query Cost

- Cost is generally measured as total elapsed time for answering query
  - Many factors contribute to time cost
    - *disk accesses, CPU*, or even network *communication*
- Typically, disk access is the predominant cost and is also relatively easy to estimate.
- Measured by taking into account
  - Number of seeks (N) calculated using → N* average-seek-cost
  - Number of blocks read calculated using → N* average-block-read-cost
  - Number of blocks written calculated using → N* average-block-write-cost
    - Cost to write a block is greater than cost to read a block
      - data is read back after being written to ensure that the write was successful

Copyright © 2007 Ramez Elmasri and Shamkant B. Navathe

Query Optimization Problem

- For each query there is a number of equivalent execution plans that can produce the same result.
  - ❖ Search Space

  For a query with *m* operations, each operation can be evaluated in *k* different ways, the search space can be as many as (*m*!) . *k* different execution plans.

- The goal of query optimization is to select an execution plan such that the total cost of executing the query is minimum.

  block transfers (data flow to/from disk) and disk seeks (finding data on the disk)

Copyright © 2007 Ramez Elmasri and Shamkant B. Navathe

Disk Cost

- If a disk has an average block transfer time of t_T and an average seek time of t_S, then reading a set of b blocks that requires S seeks would take:

$$\mathbf{bt_T + St_S \text{ seconds}}$$

- Many databases run disk tests upon installation to determine t_T and t_S on their current server
- Block writes are usually twice as expensive as block reads, since they often do a read after the write in order to verify a successful write
- Cost estimation often occurs on the conservative side, so actual time may be better than estimated

Copyright © 2007 Ramez Elmasri and Shamkant B. Navathe

Questions about the Workload

For each query in the workload:

– Which relations does it access?

– Which attributes are retrieved?

– Which attributes are involved in selection/join conditions?

– How selective are these conditions likely to be?

Copyright © 2007 Ramez Elmasri and Shamkant B. Navathe

Questions about the Workload

For each update in the workload:

– Which attributes are involved in selection/join conditions?

– How selective are these conditions likely to be?

– The type of update (INSERT/DELETE/UPDATE), and the attributes that are affected.

Copyright © 2007 Ramez Elmasri and Shamkant B. Navathe

Choices of Indexes

What indexes should we create?

– Which relations should have indexes?

– What field(s) should be the search key?

– Should we build several indexes?

• For each index, what kind of index?
– Clustered? Hash/tree?

Copyright © 2007 Ramez Elmasri and Shamkant B. Navathe

Choices of Indexes

One approach:

Consider the most important queries in turn.

• Key Question: Adding an index improve the plan? Yes, create it. Does it hurt update rates?

Copyright © 2007 Ramez Elmasri and Shamkant B. Navathe

3. Algorithms for SELECT and JOIN Operations (1)

- Implementing the SELECT Operation

- Examples:
  - (OP1): $\sigma_{SSN='123456789'}(EMPLOYEE)$
  - (OP2): $\sigma_{DNUMBER>5}(DEPARTMENT)$
  - (OP3): $\sigma_{DNO=5}(EMPLOYEE)$
  - (OP4): $\sigma_{DNO=5\ AND\ SALARY>30000\ AND\ SEX=F}(EMPLOYEE)$
  - (OP5): $\sigma_{ESSN=123456789\ AND\ PNO=10}(WORKS\_ON)$

Copyright © 2007 Ramez Elmasri and Shamkant B. Navathe

Algorithms for SELECT and JOIN Operations (2)

- Implementing the SELECT Operation (contd.):
- Search Methods for Simple Selection:
  - S1 **Linear search** (brute force):
    - Retrieve every record in the file, and test whether its attribute values satisfy the selection condition.
  - S2 **Binary search**:
    - If the selection condition involves an equality comparison on a key attribute on which the file is ordered, binary search (which is more efficient than linear search) can be used. (See OP1).
  - S3 **Using a primary index or hash key to retrieve a single record**:
    - If the selection condition involves an equality comparison on a key attribute with a primary index (or a hash key), use the primary index (or the hash key) to retrieve the record.

Copyright © 2007 Ramez Elmasri and Shamkant B. Navathe

Algorithms for SELECT and JOIN Operations (3)

- Implementing the SELECT Operation (contd.):
- Search Methods for Simple Selection:
  - S4 **Using a primary index to retrieve multiple records**:
    - If the comparison condition is >, ≥, <, or ≤ on a **key** field with a primary index, use the index to find the record satisfying the corresponding equality condition, then retrieve all subsequent records in the (ordered) file.
  - S5 **Using a clustering index to retrieve multiple records**:
    - If the selection condition involves an equality comparison on a **non-key** attribute with a clustering index, use the clustering index to retrieve all the records satisfying the selection condition.
  - S6 **Using a secondary (B+-tree) index**:
    - On an equality comparison, this search method can be used to retrieve a single record if the indexing field has unique values (is a key) or to retrieve multiple records if the indexing field is not a key.
    - In addition, it can be used to retrieve records on conditions involving >,>=, <, or <=. (FOR RANGE QUERIES)

Copyright © 2007 Ramez Elmasri and Shamkant B. Navathe

Algorithms for SELECT and JOIN Operations (4)

- Implementing the SELECT Operation (contd.):
- Search Methods for Complex Selection:
  - S7 **Conjunctive selection**:
    - If an attribute involved in any single simple condition in the conjunctive condition has an access path that permits the use of one of the methods S2 to S6, use that condition to retrieve the records and then check whether each retrieved record satisfies the remaining simple conditions in the conjunctive condition.
  - S8 **Conjunctive selection using a composite index**
    - If two or more attributes are involved in equality conditions in the conjunctive condition and a composite index (or hash structure) exists on the combined field, we can use the index directly.

Copyright © 2007 Ramez Elmasri and Shamkant B. Navathe

Algorithms for SELECT and JOIN Operations (5)

- Implementing the SELECT Operation (contd.):
- Search Methods for Complex Selection:
  - S9 **Conjunctive selection by intersection of record pointers**:
    - This method is possible if secondary indexes are available on all (or some of) the fields involved in equality comparison conditions in the conjunctive condition and if the indexes include record pointers (rather than block pointers).
    - Each index can be used to retrieve the record pointers that satisfy the individual condition.
    - The intersection of these sets of record pointers gives the record pointers that satisfy the conjunctive condition, which are then used to retrieve those records directly.
    - If only some of the conditions have secondary indexes, each retrieved record is further tested to determine whether it satisfies the remaining conditions.

Copyright © 2007 Ramez Elmasri and Shamkant B. Navathe

Algorithms for SELECT and JOIN Operations (7)

- Implementing the SELECT Operation (contd.):
  - Whenever a **single condition** specifies the selection, we can only check whether an access path exists on the attribute involved in that condition.
    - If an access path exists, the method corresponding to that access path is used; otherwise, the "brute force" linear search approach of method S1 is used. (See OP1, OP2 and OP3)
  - For **conjunctive selection conditions**, whenever *more than one* of the attributes involved in the conditions have an access path, query optimization should be done to choose the access path that *retrieves the fewest records* in the most efficient way.
  - **Disjunctive selection conditions**

Copyright © 2007 Ramez Elmasri and Shamkant B. Navathe

Selection Operation

File scan: search algorithms that locate and retrieve records that fulfill a selection condition.

Algorithm **A1** (*linear search*). Scan each file block and test all records to see whether they satisfy the selection condition.

- Cost estimate (worse scenario)= **b_r block transfers + 1 seek**
  - b_r denotes number of blocks containing records from relation *r*
- (best scenario) If selection is on a key attribute, can stop on finding record **cost = (b_r/2) block transfers + 1 seek**
- Linear search can be applied regardless of selection condition or ordering of records in the file, or availability of indices

Copyright © 2007 Ramez Elmasri and Shamkant B. Navathe

Selection Operation (2)

- Algorithm **A2** *(binary search).* Applicable if selection is an equality comparison on the attribute on which file is ordered.
  - Assume that the blocks of a relation are stored contiguously
  - Cost estimate (number of disk blocks to be scanned):
    - cost of locating the first tuple by a binary search on the blocks

      $\lceil \log_2(b_r) \rceil * (t_T + t_S)$
    - If there are multiple records satisfying selection
      - *Add transfer cost of the* number of blocks containing records that satisfy selection condition

Copyright © 2007 Ramez Elmasri and Shamkant B. Navathe

Selections Using Indices

Index scan: Search algorithms that use an index

- selection condition must be on search-key of index.

- **A3** (*primary index on candidate key, equality*). Retrieve a single record that satisfies the corresponding equality condition

  $Cost = (h_i + 1) * (t_T + t_S)$

- **A4** (*primary index on nonkey, equality)* Retrieve multiple records.
  - Records will be on consecutive blocks
    - Let b = number of blocks containing matching records
  - $Cost = h_i * (t_T + t_S) + t_S + t_T * b$
- **A5** (*equality on search-key of secondary index).*
  - Retrieve a single record if the search-key is a candidate key

    $Cost = (h_i + 1) * (t_T + t_S)$
  - Retrieve multiple records if search-key is not a candidate key
    - each of *n* matching records may be on a different block
    - **Cost =** $(h_i + n) * (t_T + t_S)$
      - Can be very expensive!

Copyright © 2007 Ramez Elmasri and Shamkant B. Navathe

Index Selection Guidelines

- Avoid using the Select (*)
- Attributes in WHERE clause are candidates for index keys.

– Exact match condition suggests hash index.

– Range query suggests tree index.

• Clustering is especially useful for range queries; can also help on equality queries if there are many duplicates.

Copyright © 2007 Ramez Elmasri and Shamkant B. Navathe

Index Selection Guidelines

- • Multi attribute search keys should be considered when a WHERE clause contains several conditions.

– Order of attributes is important for range queries. – Such indexes can sometimes enable index-only strategies for important queries.

• For **index-only strategies**, clustering is not important!

• Try to choose indexes that benefit as many queries as possible. Since only one index can be clustered per relation, choose it based on important queries that would benefit the most from clustering

Copyright © 2007 Ramez Elmasri and Shamkant B. Navathe

Algorithms for SELECT and JOIN Operations (8)

- Implementing the JOIN Operation:
  - Join (EQUIJOIN, NATURAL JOIN)
    - two–way join: a join on two files
    - e.g. $R \bowtie_{A=B} S$
    - multi-way joins: joins involving more than two files.
    - e.g. $R \bowtie_{A=B} S \bowtie_{C=D} T$
- Examples
  - (OP6): EMPLOYEE $\bowtie_{DNO=DNUMBER}$ DEPARTMENT
  - (OP7): DEPARTMENT $\bowtie_{MGRSSN=SSN}$ EMPLOYEE

Copyright © 2007 Ramez Elmasri and Shamkant B. Navathe

Join Operation

- The most time-consuming operations
- Several different algorithms to implement joins
  - Nested-loop join
  - Block nested-loop join
  - Indexed nested-loop join
  - Merge-join
  - Hash-join
- Choice is based on cost estimate

Copyright © 2007 Ramez Elmasri and Shamkant B. Navathe

Nested-Loop Join

- To compute the theta join $r \bowtie_\theta s$

  for each tuple t_r **in** r **do begin**
  for each tuple t_s **in** s **do begin**
  test pair (t_r, t_s) to see if they satisfy the join condition θ
  if they do, add $t_r \bullet t_s$ to the result.
  end
  end

- r is called the **outer relation** and s the **inner relation** of the join.
- Requires no indices and can be used with any kind of join condition.
- Expensive since it examines every pair of tuples in the two relations.

Copyright © 2007 Ramez Elmasri and Shamkant B. Navathe

Nested Loop Join

```
For each tuple r in R do
   For each tuple s in S do
      If r and s satisfy the join condition
         Then output the tuple <r,s>
```

```
for each row R1 in the outer table
    for each row R2 in the inner table
        if R1 joins with R2
            return (R1, R2)
```

Copyright © 2007 Ramez Elmasri and Shamkant B. Navathe

Block Nested-Loop Join

- In the worst case, if there is enough memory only to hold one block of each relation, If the smaller relation fits entirely in memory, use that as the inner relation.
- Block nested-loops algorithm is preferable.
- Variant of nested-loop join in which every block of inner relation is paired with every block of outer relation.

```
for each block B_r of r do begin
    for each block B_s of s do begin
        for each tuple t_r in B_r do begin
            for each tuple t_s in B_s do begin
                Check if (t_r,t_s) satisfy the join condition
                if they do, add t_r • t_s to the result.
            end
        end
    end
end
```

Copyright © 2007 Ramez Elmasri and Shamkant B. Navathe

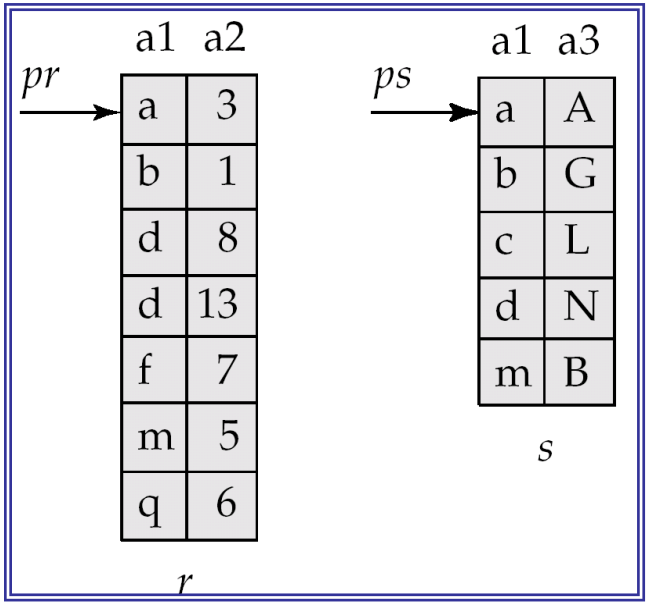
Indexed Nested-Loop Join

- Index lookups can replace file scans if
  - join is an equi-join or natural join and
  - an index is available on the inner relation's join attribute
    - Can construct an index just to compute a join.
- For each tuple t_r in the outer relation *r,* use the index to look up tuples in *s* that satisfy the join condition with tuple t_r.
- Worst case: buffer has space for only one page of *r*, and, for each tuple in *r*, we perform an index lookup on *s.*
- Cost of the join: $b_r\ (t_T + t_S) + n_r * c$
  - Where *c* is the cost of traversing index and fetching all matching *s* tuples for one tuple or *r*
  - *c* can be estimated as cost of a single selection on *s* using the join condition.
- If indices are available on join attributes of both *r* and *s,* use the relation with fewer tuples as the outer relation.

Copyright © 2007 Ramez Elmasri and Shamkant B. Navathe

Merge-Join

- Sort both relations on their join attribute (if not already sorted on the join attributes).
- Merge the sorted relations to join them
  - Join step is similar to the merge stage of the sort-merge algorithm.
  - Main difference is handling of duplicate values in join attribute — every pair with same value on join attribute must be matched
  - Detailed algorithm in book



Copyright © 2007 Ramez Elmasri and Shamkant B. Navathe

Merge-Join (2)

- Can be used only for equi-joins and natural joins
- Each block needs to be read only once (assuming all tuples for any given value of the join attributes fit in memory
- Thus the cost of merge join is:

$$b_r + b_s \text{ block transfers } + \lceil b_r / b_b \rceil + \lceil b_s / b_b \rceil \text{ seeks}$$

  - + the cost of sorting if relations are unsorted.

Copyright © 2007 Ramez Elmasri and Shamkant B. Navathe

Hash-Join

- Applicable for equi-joins and natural joins.
- A hash function *h* is used to partition tuples of both relations
- *h* maps *JoinAttrs* values to {0, 1, ..., *n*}, where *JoinAttrs* denotes the common attributes of *r* and *s* used in the natural join.
  - $r_0, r_1, \ldots, r_n$ denote partitions of *r* tuples
    - Each tuple $t_r \in r$ is put in partition r_i where $i = h(t_r[JoinAttrs])$.
  - $s_0, s_1 \ldots, s_n$ denotes partitions of *s* tuples
    - Each tuple $t_s \in s$ is put in partition s_i, where $i = h(t_s[JoinAttrs])$.

Copyright © 2007 Ramez Elmasri and Shamkant B. Navathe

Hash-Join (2)

- *r* tuples in r_i need only to be compared with *s* tuples in s_i Need not be compared with *s* tuples in any other partition, since:
  - an *r* tuple and an *s* tuple that satisfy the join condition will have the same value for the join attributes.
  - If that value is hashed to some value *i*, the *r* tuple has to be in r_i and the *s* tuple in s_i.

Copyright © 2007 Ramez Elmasri and Shamkant B. Navathe

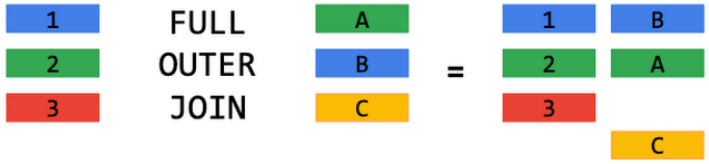
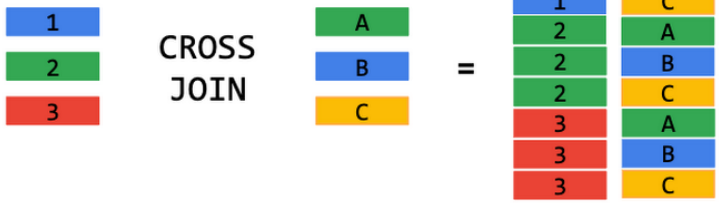
Hash-Join

- Hash Join is preferred by the execution plan to join unsorted large dataset but it consume more memory than the outer joins.

- Merge join is faster than the hash join

Copyright © 2007 Ramez Elmasri and Shamkant B. Navathe

Types of join

INNER JOIN		Only returns rows that meet the join condition
RIGHT OUTER JOIN		Returns all rows from the table on the right side of JOIN and matched rows from the left side of the JOIN
LEFT OUTER JOIN		Returns all rows from the table on the left side of JOIN and matched rows from the right side of the JOIN
FULL OUTER JOIN		Returns all rows from both sides even if join condition is not met
CROSS JOIN		Cartesian product between the two sides is a join but without a join condition. Returns all rows joined from both sides

Copyright © 2007 Ramez Elmasri and Shamkant B. Navathe

Algorithms for SELECT and JOIN Operations (14)

- Implementing the JOIN Operation (contd.):
- Factors affecting JOIN performance
  - Available buffer space
  - Join selection factor
  - Choice of inner VS outer relation

Copyright © 2007 Ramez Elmasri and Shamkant B. Navathe

2nd Way: Using Heuristics in Query Optimization

- **Query tree**: a tree data structure that corresponds to a relational algebra expression. It represents the input relations of the query as *leaf nodes* of the tree, and represents the relational algebra operations as *internal nodes*.
- An execution of the query tree consists of executing an internal node operation whenever its operands are available and then replacing that internal node by the relation that results from executing the operation.

Copyright © 2007 Ramez Elmasri and Shamkant B. Navathe

Using Heuristics in Query Optimization

- **Example**:

  Q: SELECT LNAME

  FROM EMPLOYEE, WORKS_ON, PROJECT

  WHERE PNAME = ‘AQUARIUS’

  AND PNMUBER=PNO

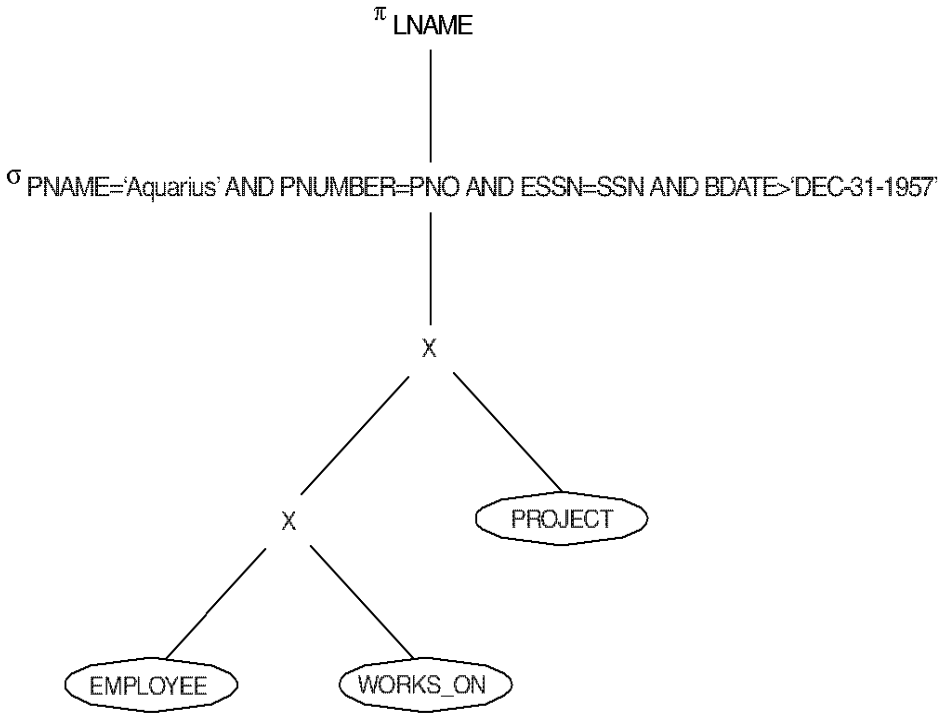
  AND ESSN=SSN

  AND BDATE > ‘1957-12-31’;

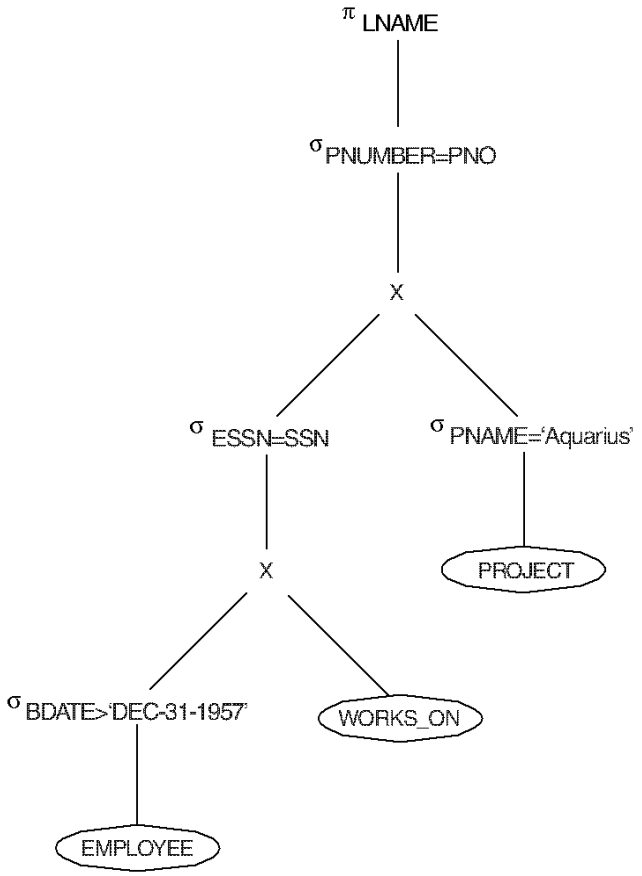
Copyright © 2007 Ramez Elmasri and Shamkant B. Navathe

Using Heuristics in Query Optimization (10)

Steps in converting a query tree using heuristic optimization



(a) Initial (canonical) query tree

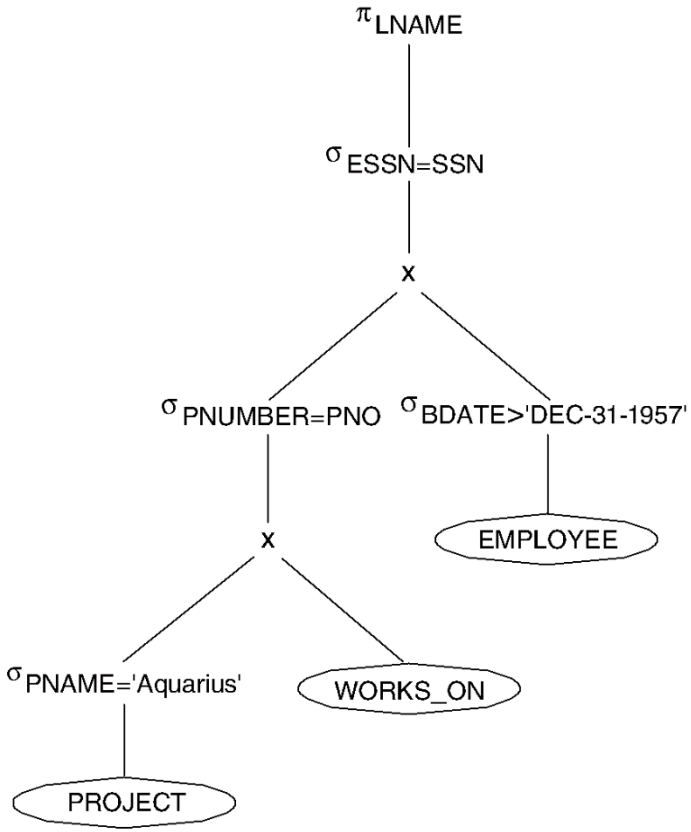


(b) Moving selection down the tree

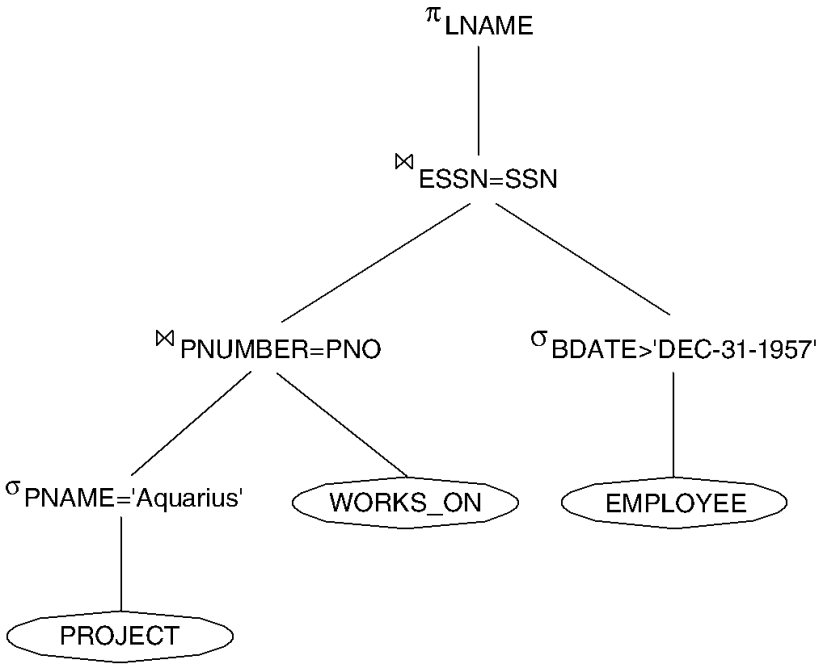
Copyright © 2007 Ramez Elmasri and Shamkant B. Navathe

Using Heuristics in Query Optimization (11)

Steps in converting a **query tree** using heuristic optimization



(c) Apply the more restrictive SELECT operations first.

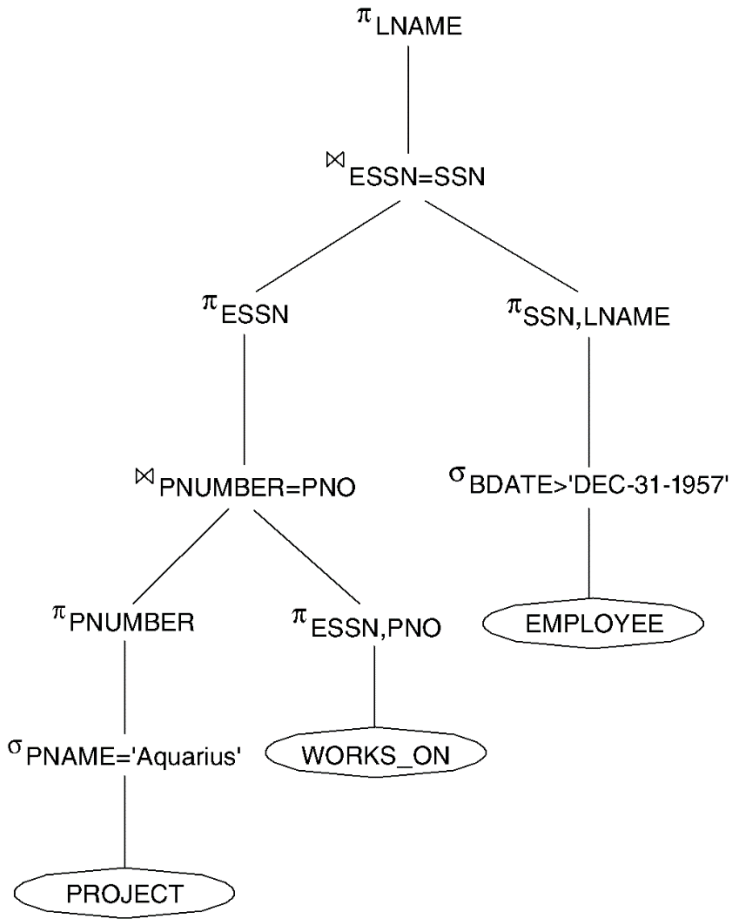


(d) Replace CARTESIEN Product and SELECT with JOIN operations

Copyright © 2007 Ramez Elmasri and Shamkant B. Navathe

Using Heuristics in Query Optimization (12)

Steps in converting a query tree using heuristic optimization



(e) Move PROJECT operations down the query tree

Copyright © 2007 Ramez Elmasri and Shamkant B. Navathe

Questions?

Copyright © 2007 Ramez Elmasri and Shamkant B. Navathe