Database Management System (MDS 505)

Jagdish Bhatta

Unit – 4

SQL & Query Optimization

SQL Standards ,Data types

- We have already covered these in unit 2.

Database Objects- DDL-DML-DCL-TCL

- We have already covered these in unit 2

Embedded SQL

- **Embedded SQL** is a method of combining the computing power of a programming language and the database manipulation capabilities of SQL. Embedded SQL statements are SQL statements written inline with the program source code, of the host language. The embedded SQL statements are parsed by an embedded SQL preprocessor and replaced by host-language calls to a code library. The output from the preprocessor is then compiled by the host compiler. This allows programmers to embed SQL statements in programs written in any number of languages such as C/C++. E.g.: Proc*C is Oracle's embedded SQL environment.

Embedded SQL

- **E.g: Embedding SQL into C.**

```
{
  int a;
 /* ... */
EXEC SQL SELECT salary INTO :a
                                  FROM Employee
                                   WHERE SSN=876543210;
/* ... */
  printf("The salary is %d\n", a);
   /* ... */
 }
```

All SQL statements need to start with EXEC SQL and end with a semicolon ";". You can place the SQL statements anywhere within a C/C++ block, with the restriction that the declarative statements do not come after the executable statements.

Static Vs Dynamic SQL

- In the previous examples, the embedded SQL queries were written as part of the host program source code. Hence, anytime we want to write a different query, we must modify the program code and go through all the steps involved (compiling, debugging, testing, and so on). In some cases, it is convenient to write a program that can execute different SQL queries or updates (or other operations) *dynamically at runtime*. For example, we may want to write a program that accepts an SQLquery typed from the monitor, executes it, and displays its result, such as the interactive interfaces available for most relational DBMSs. Another example is when a user-friendly interface generates SQL queries dynamically for the user based on user input through a Web interface or mobile App. **Dynamic SQL**, which is one technique for writing this type of database program.

Query Processing

- **Query processing refers to the range of activities involved in extracting data from** a database. The activities include translation of queries in high-level database languages into expressions that can be used at the physical level of the file system, a variety of query-optimizing transformations, and actual evaluation of queries.
- The basic steps are:
  - **Parsing and translation.**
  - **Optimization.**
  - **Evaluation.**

Query Processing

- Before query processing can begin, the system must translate the query into a usable form. A language such as SQL is suitable for human use, but is ill suited to be the system's internal representation of a query. A more useful internal representation is one based on the extended relational algebra.

Query Processing

- Thus, the first action the system must take in query processing is to translate a given query into its internal form. This translation process is similar to the work performed by the parser of a compiler. In generating the internal form of the query, the parser checks the syntax of the user’s query, verifies that the relation names appearing in the query are names of the relations in the database, and so on. The system constructs a parse-tree representation of the query, which it then translates into a relational-algebra expression. If the query was expressed in terms of a view, the translation phase also replaces all uses of the view by the relational-algebra expression that defines the view.

Query Processing

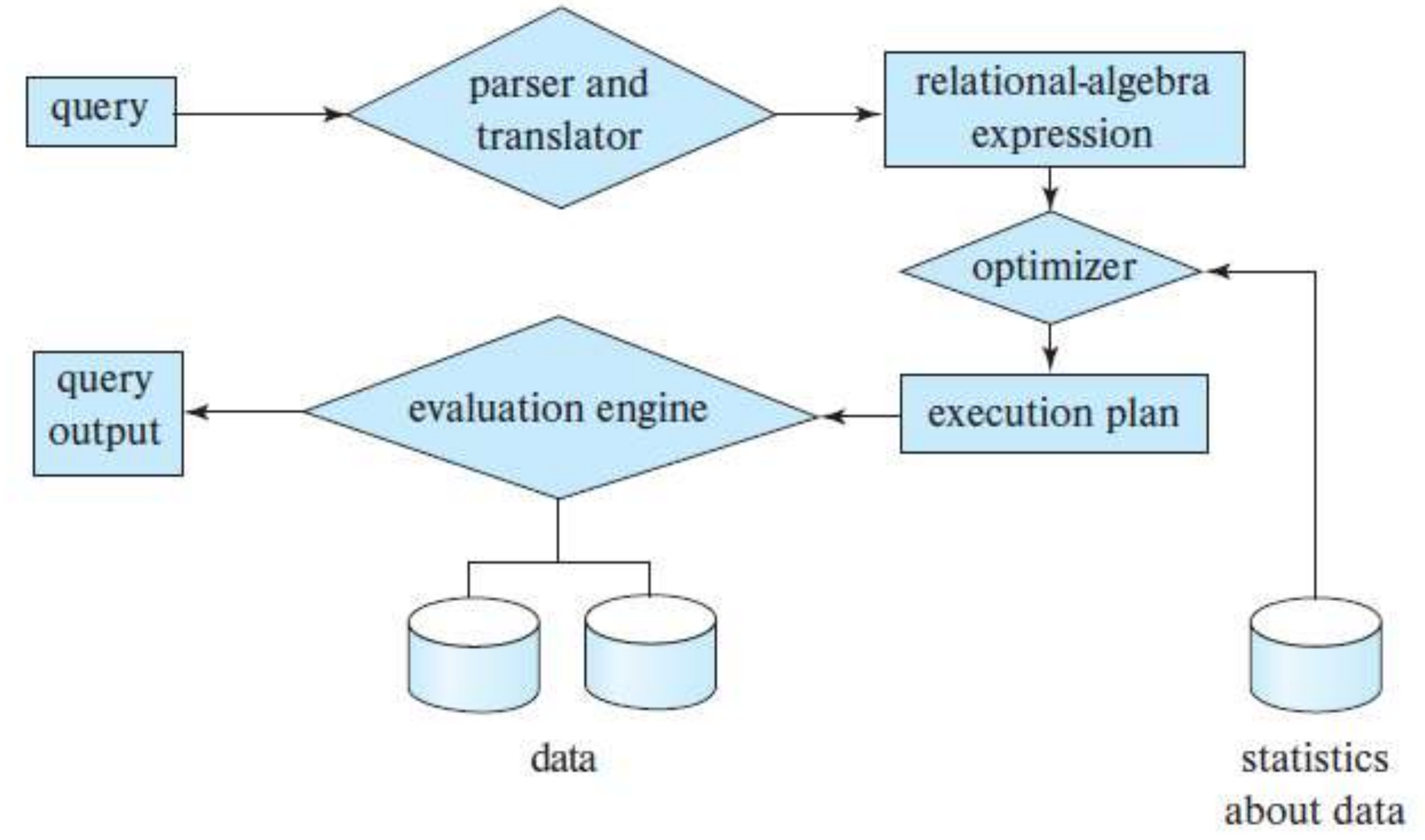


Figure 15.1 Steps in query processing.

Query Processing

- Given a query, there are generally a variety of methods for computing the answer. For example, we have seen that, in SQL, a query could be expressed in several different ways. Each SQL query can itself be translated into a relational algebra expression in one of several ways. Furthermore, the relational-algebra representation of a query specifies only partially how to evaluate a query; there are usually several ways to evaluate relational-algebra expressions.

Query Processing

- Consider the query;

select *salary*

from *instructor*

where *salary* < *75000;*

- This query can be translated into either of the following relational-algebra expressions:

$$\sigma_{salary<75000}\,(\Pi_{salary}\,(instructor))$$

$$\Pi_{salary}\,(\sigma_{salary<75000}\,(instructor))$$

Query Processing

- Further, we can execute each relational-algebra operation by one of several different algorithms.
- To specify fully how to evaluate a query, we need not only to provide the relational-algebra expression, but also to annotate it with instructions specifying how to evaluate each operation. Annotations may state the algorithm to be used for a specific operation, or the particular index or indices to use.
- A relational algebra operation annotated with instructions on how to evaluate it is called an **evaluation primitive. A sequence of primitive operations that can be used to** evaluate a query is a **query-execution plan or query-evaluation plan.**
- The **query-execution engine takes a query-evaluation plan, executes that plan, and** returns the answers to the query.

Query Processing

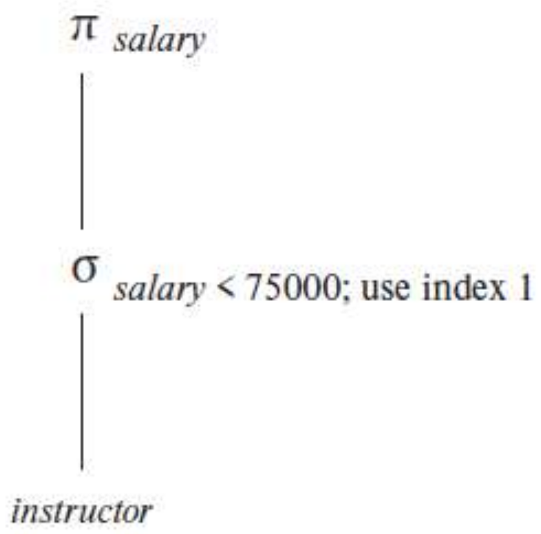


Figure 15.2 A query-evaluation plan.

Query Processing

- The different evaluation plans for a given query can have different costs. We do not expect users to write their queries in a way that suggests the most efficient evaluation plan. Rather, it is the responsibility of the system to construct a query evaluation plan that minimizes the cost of query evaluation; this task is called ***query optimization.***
- Once the query plan is chosen, the query is evaluated with that plan, and the result of the query is output.
- The sequence of steps already described for processing a query is representative; not all databases exactly follow those steps. For instance, instead of using the relational-algebra representation, several databases use an annotated parse tree representation based on the structure of the given SQL query

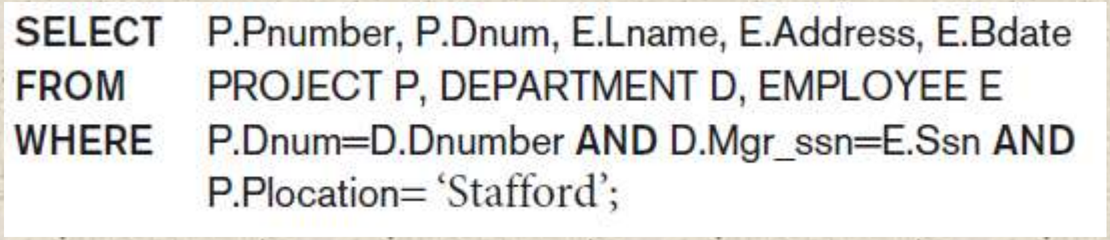
Query Processing

- In order to optimize a query, a query optimizer must know the cost of each operation. Although the exact cost is hard to compute, since it depends on many parameters such as actual memory available to the operation, it is possible to get a rough estimate of execution cost for each operation.

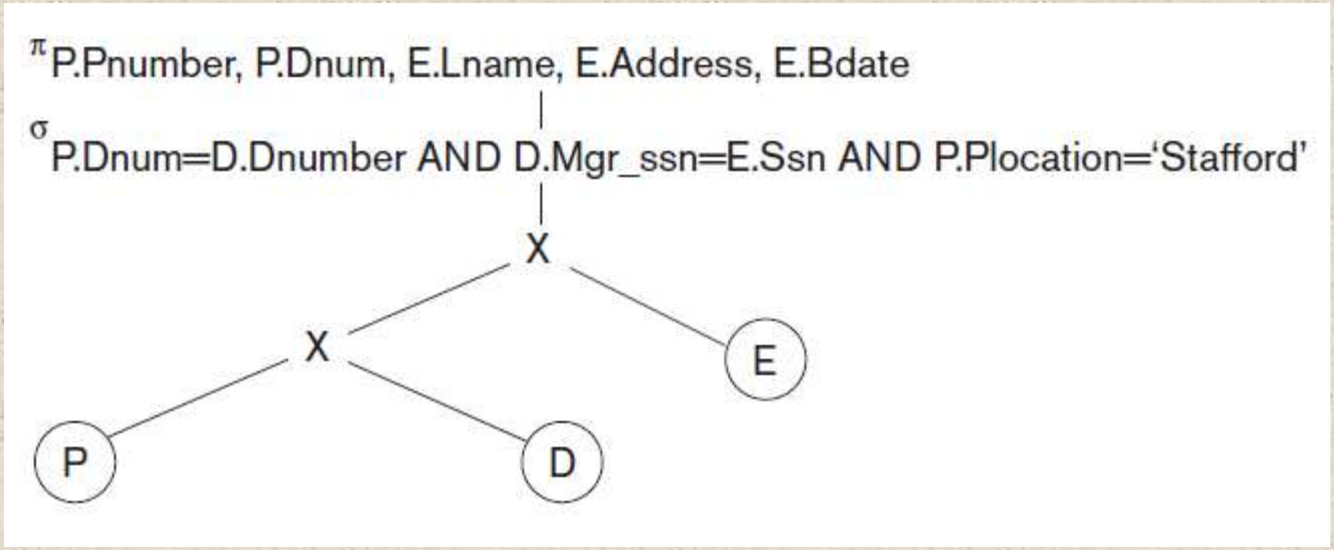
Query Tree

- A **query tree** is a tree data structure that corresponds to an extended relational algebra expression. It represents the input relations of the query as *leaf nodes* of the tree, and it 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. The order of execution of operations *starts at the leaf nodes*, which represents the input database relations for the query, and *ends at the root node*, which represents the final operation of the query. The execution terminates when the root node operation is executed and produces the result relation for the query

Query Tree

- 

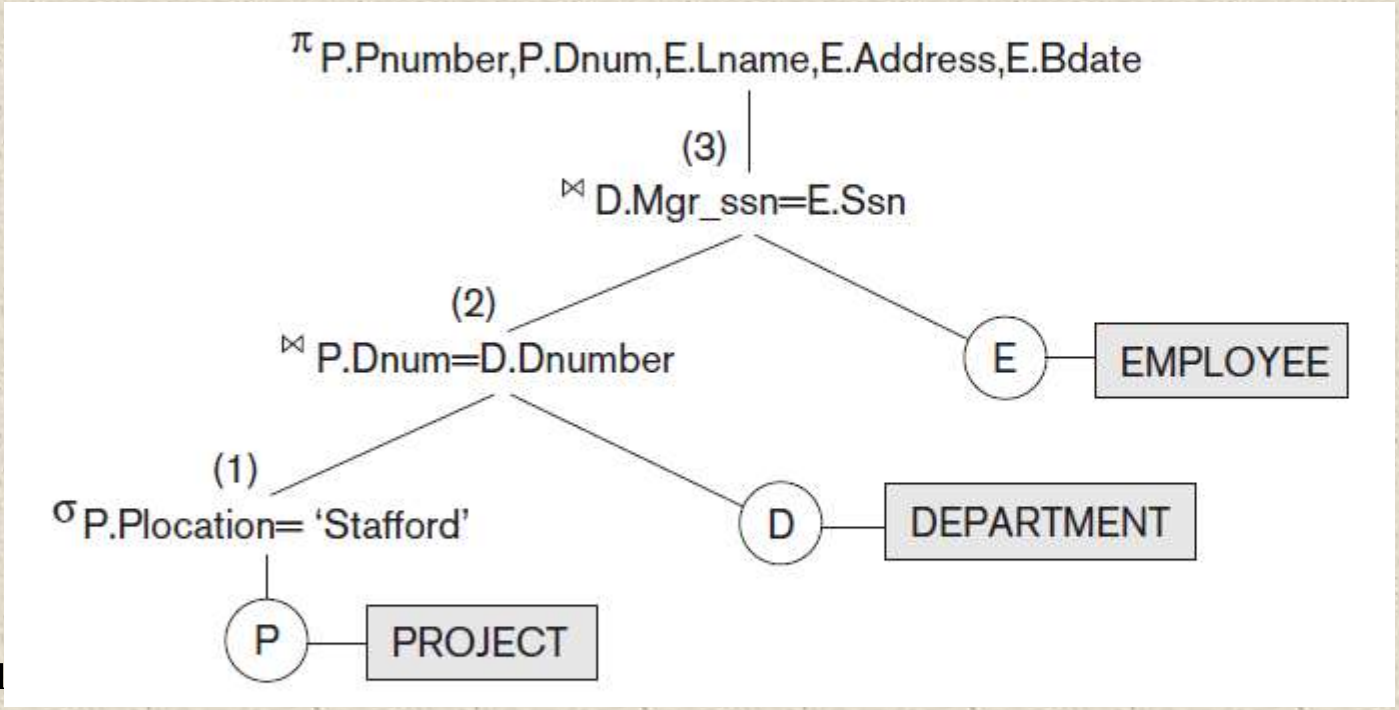
```
SELECT   P.Pnumber, P.Dnum, E.Lname, E.Address, E.Bdate
FROM     PROJECT P, DEPARTMENT D, EMPLOYEE E
WHERE    P.Dnum=D.Dnumber AND D.Mgr_ssn=E.Ssn AND
         P.Plocation= 'Stafford';
```


π P.Pnumber, P.Dnum, E.Lname, E.Address, E.Bdate
σ P.Dnum=D.Dnumber AND D.Mgr_ssn=E.Ssn AND P.Plocation='Stafford'
X
X
E
P
D

Query Tree

-
  ```
  SELECT   P.Pnumber, P.Dnum, E.Lname, E.Address, E.Bdate
  FROM     PROJECT P, DEPARTMENT D, EMPLOYEE E
  WHERE    P.Dnum=D.Dnumber AND D.Mgr_ssn=E.Ssn AND
           P.Plocation= 'Stafford';
  ```

$$\pi_{\text{Pnumber, Dnum, Lname, Address, Bdate}}(((\sigma_{\text{Plocation='Stafford'}}(\text{PROJECT})) \bowtie_{\text{Dnum=Dnumber}}(\text{DEPARTMENT})) \bowtie_{\text{Mgr\_ssn=Ssn}}(\text{EMPLOYEE}))$$



Measures of Query Cost

- There are multiple possible evaluation plans for a query, and it is important to be able to compare the alternatives in terms of their (estimated) cost, and choose the best plan. To do so, we must estimate the cost of individual operations, and combine them to get the cost of a query evaluation plan.
- The cost of query evaluation can be measured in terms of a number of different resources, including disk accesses, CPU time to execute a query, and, in a distributed or parallel database system, the cost of communication.
- In large database systems, the cost to access data from disk is usually the most important cost, since disk accesses are slow compared to in-memory operations.

Measures of Query Cost

- The *number of block transfers from disk and the number of disk seeks* can be used to estimate the cost of a query-evaluation plan. If the disk subsystem takes an average of *tT seconds to transfer a block of data, and has an average block-access* time (disk seek time plus rotational latency) of *tS seconds, then an operation that* transfers *b blocks and performs S seeks would take* $b * tT + S * tS$ *seconds.*
- *The* values of *tT and tS must be calibrated for the disk system used, but typical values* for high-end disks today would be *tS = 4 milliseconds and tT = 0.1 milliseconds*, assuming a 4-kilobyte block size and a transfer rate of 40 megabytes per second.
- One can refine the cost estimates further by distinguishing block reads from block writes, since block writes are typically about twice as expensive as reads (this is because disk systems read sectors back after they are written to verify that the write was successful).

Measures of Query Cost

- The **response time for a query-evaluation plan (that is, the wall-clock time** required to execute the plan), assuming no other activity is going on in the computer, would account for all these costs, and could be used as a measure of the cost of the plan.
- Unfortunately, the response time of a plan is very hard to estimate without actually executing the plan, for the following reasons:
  - **The response time depends on the contents of the buffer when the query** begins execution; this information is not available when the query is optimized, and is hard to account for even if it were available.
  - **In a system with multiple disks, the response time depends on how accesses** are distributed among disks, which is hard to estimate without detailed knowledge of data layout on disk.

Measures of Query Cost

- Optimizers often try to minimize the total **resource** consumption of a query plan. Where resource consumption may be extra disk reads, parallel reads across multiple disks etc.

Evaluation of Expressions

- The obvious way to evaluate an expression is simply to evaluate one operation at a time, in an appropriate order. The result of each evaluation is **materialized** in a temporary relation for subsequent use. A disadvantage to this approach is the need to construct the temporary relations, which (unless they are small) must be written to disk. An alternative approach is to evaluate several operations simultaneously in a **pipeline, with the results of one operation passed on to the** next, without the need to store a temporary relation.

Evaluation of Expressions

- **Materialization:**
- **Consider the expression;** $\Pi_{name}(\sigma_{building = \text{"Watson"}}(department) \bowtie instructor)$

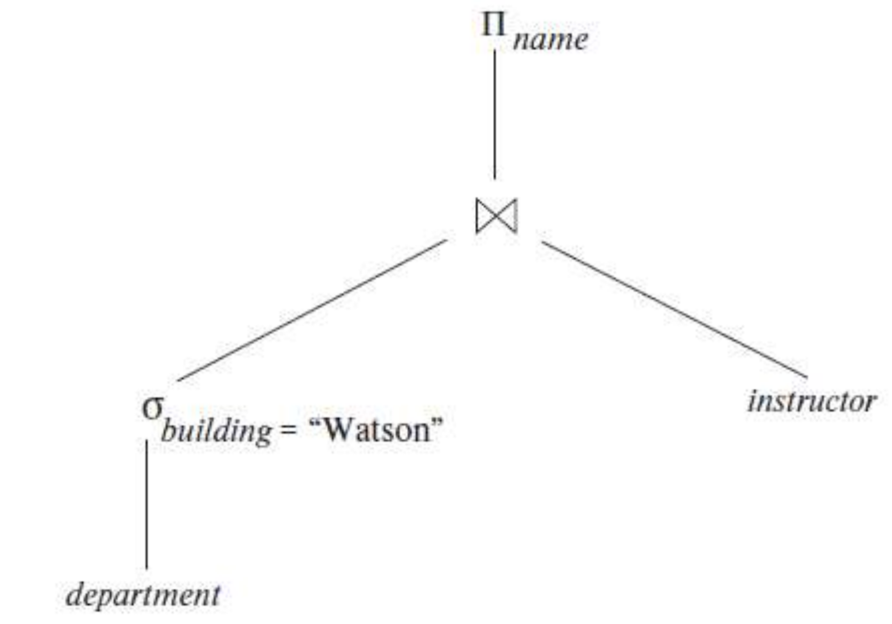


Figure 15.11 Pictorial representation of an expression.

Evaluation of Expressions

- **Materialization:**

- **If we apply the materialization approach, we start from the lowest-level operations in the expression (at the bottom of the tree).** In our example, there is only one such operation: *the selection operation on department. The inputs to the* lowest-level operations are relations in the database. We execute these operations by the algorithms that we studied earlier, like selection, join etc, and we store the results in temporary relations. We can use these temporary relations to execute the operations at the next level up in the tree, where the inputs now are either temporary relations or relations stored in the database. In our example, the inputs to the join are the *instructor* relation and the temporary relation created by the *selection on department*. The join can now be evaluated, creating another temporary relation.

Evaluation of Expressions

- **Materialization:**
- By repeating the process, we will eventually evaluate the operation at the root of the tree, giving the final result of the expression. In the earlier example, we get the final result by executing the projection operation at the root of the tree, using as input the temporary relation created by the join.

- This is known as **materialized evaluation,** since the results of each intermediate operation are created (materialized) and then are used for evaluation of the next-level operations.

- To compute the cost of evaluating an expression as done here, we have to add the costs of all the operations, as well as the cost of writing the intermediate results to disk.

Evaluation of Expressions

- **Pipelining:**
- We can improve query-evaluation efficiency by reducing the number of temporary files that are produced. We achieve this reduction by combining several relational operations into a *pipeline of operations, in which the results of one operation* are passed along to the next operation in the pipeline. This is called **pipelined evaluation.**
- For example, consider the expression $(\Pi_{a1,a2}(r \bowtie s))$. *If materialization were applied, evaluation would involve creating a temporary relation to hold the result of the join, and then reading back in the result to perform the projection.* These operations can be combined: When the join operation generates a tuple of its result, it passes that tuple immediately to the project operation for processing. By combining the join and the projection, we avoid creating the intermediate result, and instead create the final result directly.

Evaluation of Expressions

- **Pipelining:**
- Creating a pipeline of operations can provide two benefits:
  - **It eliminates the cost of reading and writing temporary relations, reducing** the cost of query evaluation.
  - **It can start generating query results quickly, if the root operator of a query evaluation** plan is combined in a pipeline with its inputs. This can be quite useful if the results are displayed to a user as they are generated, since otherwise there may be a long delay before the user sees any query results.

Query Optimization

- **Query optimization is the process of selecting the most efficient query-evaluation** plan from among the many strategies usually possible for processing a given query, especially if the query is complex. We do not expect users to write their queries so that they can be processed efficiently. Rather, we expect the system to construct a query-evaluation plan that minimizes the cost of query evaluation. This is where query optimization comes into play.
- Query optimization is the process of executing a SQL query in relational databases to determine the most efficient way to execute a given query by considering the possible query plans. The goal of query optimization is to optimize the given query for the sake of efficiency.

Query Optimization

- Query optimization in database has gained significant importance as it helps to reduce the size, memory usage and time required for any query to be processed.
- The main objective of any query optimization is to determine the best strategy for executing each query.

Query Optimization

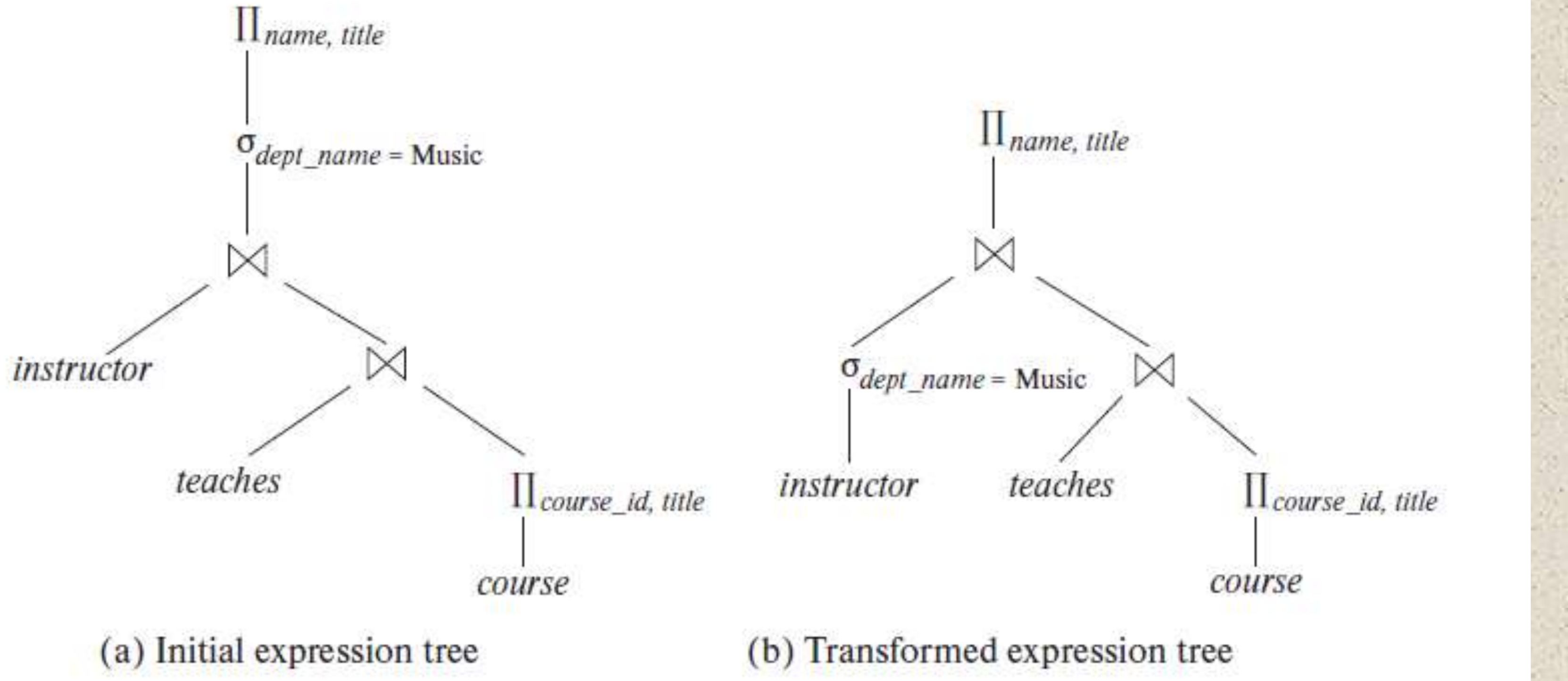
- Consider the following relational-algebra expression, for the query “Find the names of all instructors in the Music department together with the course title of all the courses that the instructors teach.”

$$\Pi_{name, title}\,(\sigma_{dept\_name = \text{“Music”}}\,(instructor \bowtie (teaches \bowtie \Pi_{course\_id, title}(course))))$$

- Since we are concerned with only those tuples in the *instructor relation that pertain to the Music department, we* do not need to consider those tuples that do not have *dept name = “Music”. By* reducing the number of tuples of the *instructor relation that we need to access,* we reduce the size of the intermediate result. Our query is now represented by the relational-algebra expression:

$$\Pi_{name, title}\,((\sigma_{dept\_name = \text{“Music”}}\,(instructor)) \bowtie (teaches \bowtie \Pi_{course\_id, title}(course)))$$

Query Optimization



$\Pi_{name,\ title}$
$\sigma_{dept\_name\ =\ \text{Music}}$
$\bowtie$
instructor
$\bowtie$
teaches
$\Pi_{course\_id,\ title}$
course
(a) Initial expression tree
$\Pi_{name,\ title}$
$\bowtie$
$\sigma_{dept\_name\ =\ \text{Music}}$
$\bowtie$
instructor
teaches
$\Pi_{course\_id,\ title}$
course
(b) Transformed expression tree

Figure 16.1 Equivalent expressions.

Query Optimization

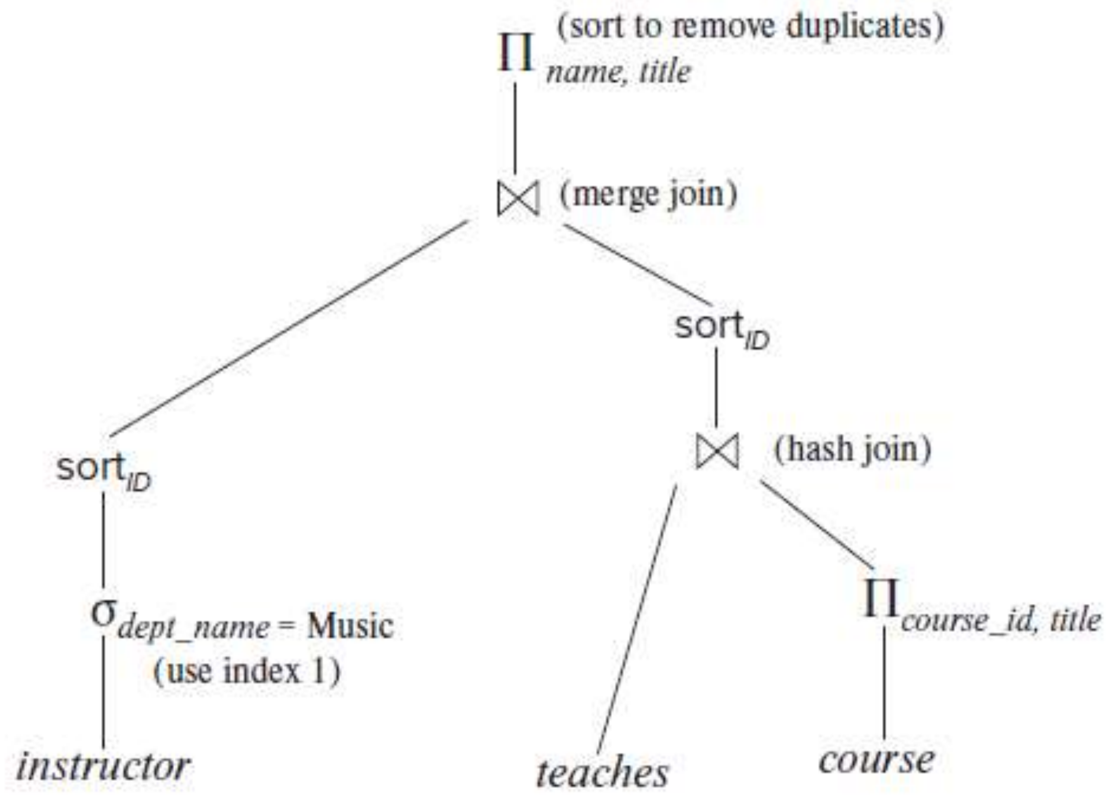


Figure 16.2 An evaluation plan.

Query Optimization

- An evaluation plan defines exactly what algorithm should be used for each operation and how the execution of the operations should be coordinated. Figure 16.2 illustrates one possible evaluation plan for the expression from Figure 16.1(b).
- As we have seen, several different algorithms can be used for each relational operation, giving rise to alternative evaluation plans. In the figure, hash join has been chosen for one of the join operations, while the other uses merge join, after sorting the relations on the join attribute, which is ID. All edges are assumed to be pipelined, unless marked as materialized. With pipelined edges the output of the producer is sent directly to the consumer, without being written out to disk.

Query Optimization

- Given a relational-algebra expression, it is the job of the query optimizer to come up with a query-evaluation plan that computes the same result as the given expression, and is the least costly way of generating the result (or, at least, is not much costlier than the least costly way).

Query Optimization

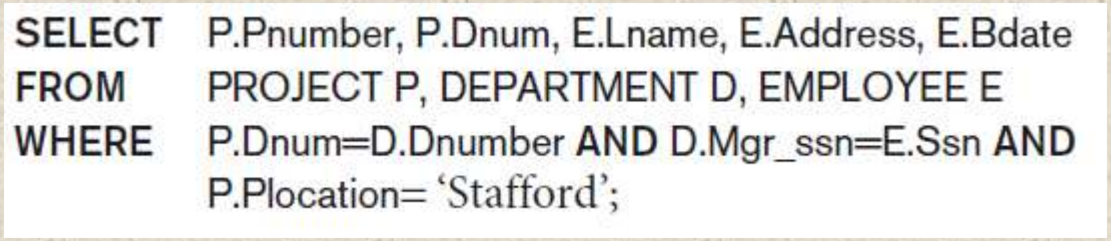
- To find the least-costly query-evaluation plan, the optimizer needs to generate alternative plans that produce the same result as the given expression, and to choose the least-costly one. Generation of query-evaluation plans involves three steps:
  - generating expressions that are logically equivalent to the given expression,
  - annotating the resultant expressions in alternative ways to generate alternative query-evaluation plans, and
  - estimating the cost of each evaluation plan, and choosing the one whose estimated cost is the least

Heuristic Optimization of Query Trees

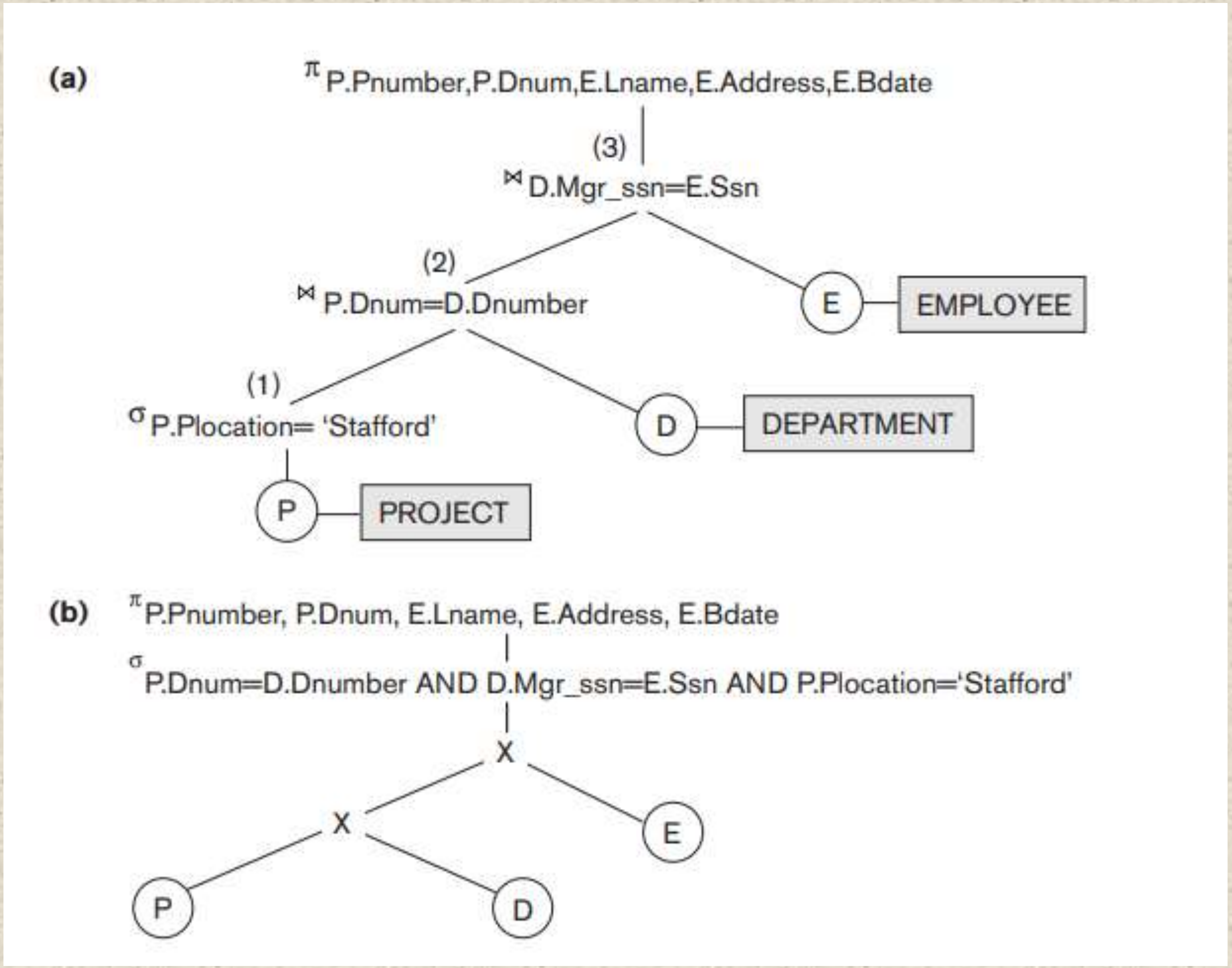
- Many different query trees—can be semantically equivalent; that is, they can represent the same query and produce the same results.

- The query parser will typically generate a standard initial query tree to correspond to an SQL query, without doing any optimization. For example, for a SELECTPROJECT-JOIN query, in next slide, the initial tree is shown in Figure (b). The CARTESIAN PRODUCT of the relations specified in the FROM clause is first applied; then the selection and join conditions of the WHERE clause are applied, followed by the projection on the SELECT clause attributes. Such a canonical query tree represents a relational algebra expression that is very inefficient if executed directly, because of the CARTESIAN PRODUCT (×) operations.

Heuristic Optimization of Query Trees



```
SELECT   P.Pnumber, P.Dnum, E.Lname, E.Address, E.Bdate
FROM     PROJECT P, DEPARTMENT D, EMPLOYEE E
WHERE    P.Dnum=D.Dnumber AND D.Mgr_ssn=E.Ssn AND
         P.Plocation= 'Stafford';
```


(a)
π P.Pnumber,P.Dnum,E.Lname,E.Address,E.Bdate
(3)
$\bowtie$ D.Mgr_ssn=E.Ssn
(2)
$\bowtie$ P.Dnum=D.Dnumber
E
EMPLOYEE
(1)
σ P.Plocation= 'Stafford'
D
DEPARTMENT
P
PROJECT
(b)
π P.Pnumber, P.Dnum, E.Lname, E.Address, E.Bdate
σ P.Dnum=D.Dnumber AND D.Mgr_ssn=E.Ssn AND P.Plocation='Stafford'
X
X
E
P
D

Heuristic Optimization of Query Trees

- For example, if the PROJECT, DEPARTMENT, and EMPLOYEE relations had record sizes of 100, 50, and 150 bytes and contained 100, 20, and 5,000 tuples, respectively, the result of the CARTESIAN PRODUCT would contain 10 million tuples of record size 300 bytes each.
- However, this canonical query tree in Figure b is in a simple standard form that can be easily created from the SQL query. It will never be executed. The heuristic query optimizer will transform this initial query tree into an equivalent final query tree that is efficient to execute.

Heuristic Optimization of Query Trees

- The optimizer must include rules for equivalence among extended relational algebra expressions that can be applied to transform the initial tree into the final, optimized query tree.
- Consider the following query Q on the database: Find the last names of employees born after 1957 who work on a project named ‘Aquarius’. This query can be specified in SQL as follows:

```
Q: SELECT E.Lname
    FROM EMPLOYEE AS E, WORKS_ON AS W, PROJECT AS P
        WHERE      P.Pname=‘Aquarius’ AND P.Pnumber = W.Pno
AND     E.Essn=W.Ssn AND E.Bdate > ‘1957-12-31’;
```

Heuristic Optimization of Query Trees

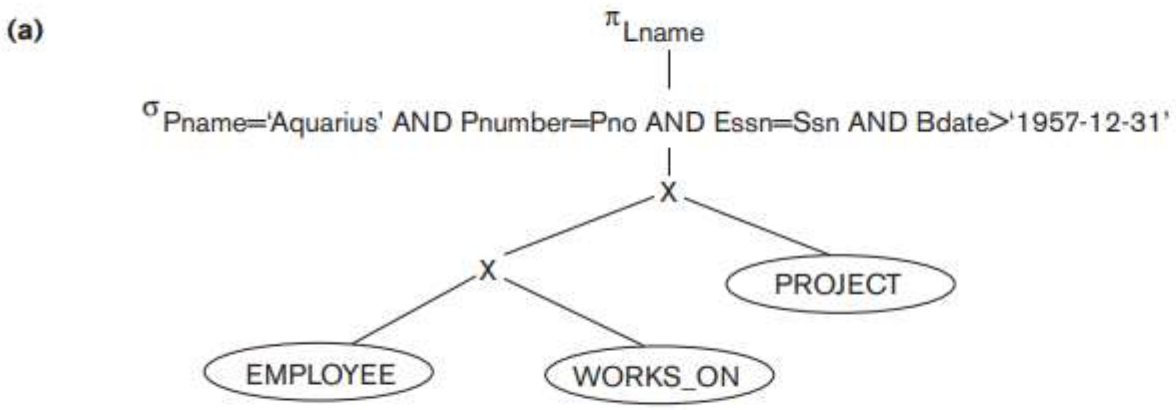
- The initial query tree for Q is shown in Figure 19.2(a). Executing this tree directly first creates a very large file containing the CARTESIAN PRODUCT of the entire EMPLOYEE, WORKS_ON, and PROJECT files. That is why the initial query tree is never executed, but is transformed into another equivalent tree that is efficient to execute. This particular query needs only one record from the PROJECT relation—for the 'Aquarius' project—and only the EMPLOYEE records for those whose date of birth is after '1957-12-31'.
- Figure 19.2(b) shows an improved query tree that first applies the SELECT operations to reduce the number of tuples that appear in the CARTESIAN PRODUCT.

Heuristic Optimization of Query Trees

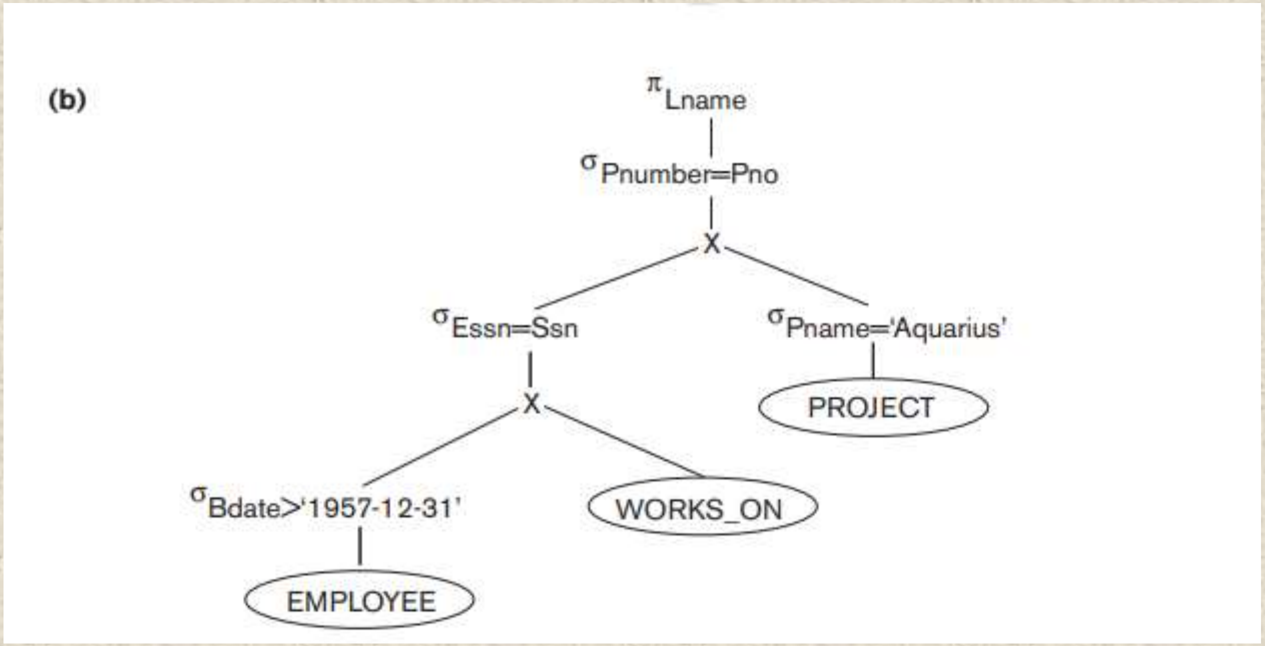
- A further improvement is achieved by switching the positions of the EMPLOYEE and PROJECT relations in the tree, as shown in Figure 19.2(c). This uses the information that Pnumber is a key attribute and Pname is uniqueof the PROJECT relation, and hence the SELECT operation on the PROJECT relation will retrieve a single record only. We can further improve the query tree by replacing any CARTESIAN PRODUCT operation that is followed by a join condition as a selection with a JOIN operation, as shown in Figure 19.2(d). Another improvement is to keep only the attributes needed by subsequent operations in the intermediate relations, by including PROJECT (π) operations as early as possible in the query tree, as shown in Figure 19.2(e). This reduces the attributes (columns) of the intermediate relations, whereas the SELECT operations reduce the number of tuples (records).

Heuristic Optimization of Query Trees

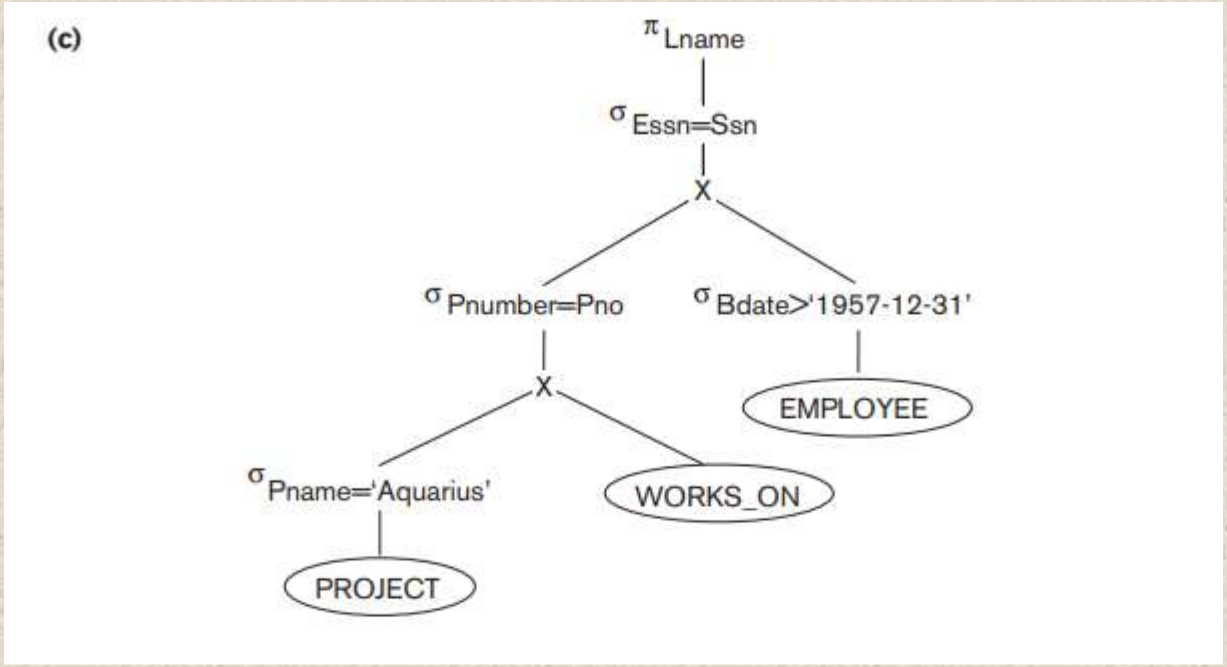
Figure 19.2
Steps in converting a query tree during heuristic optimization. (a) Initial (canonical) query tree for SQL query Q. (b) Moving SELECT operations down the query tree. (c) Applying the more restrictive SELECT operation first.


(a)
π_{Lname}
$\sigma_{\text{Pname='Aquarius' AND Pnumber=Pno AND Essn=Ssn AND Bdate>'1957-12-31'}}$
X
X
PROJECT
EMPLOYEE
WORKS_ON

Heuristic Optimization of Query Trees



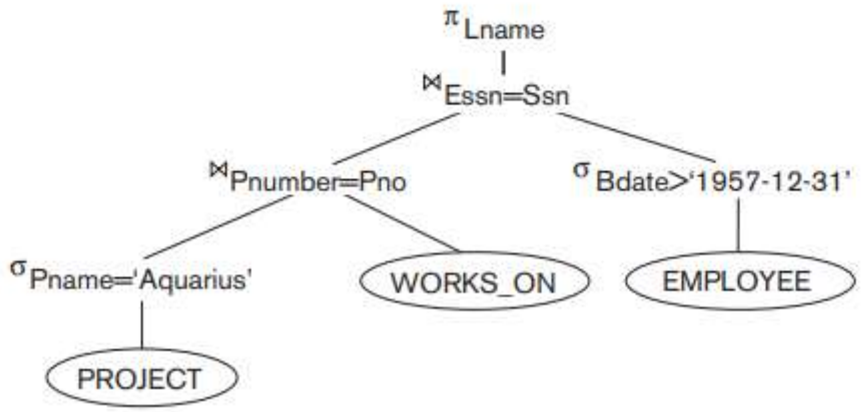
Heuristic Optimization of Query Trees


(c)
π_{Lname}
$\sigma_{\text{Essn=Ssn}}$
X
$\sigma_{\text{Pnumber=Pno}}$
$\sigma_{\text{Bdate>'1957-12-31'}}$
X
EMPLOYEE
$\sigma_{\text{Pname='Aquarius'}}$
WORKS_ON
PROJECT

Heuristic Optimization of Query Trees

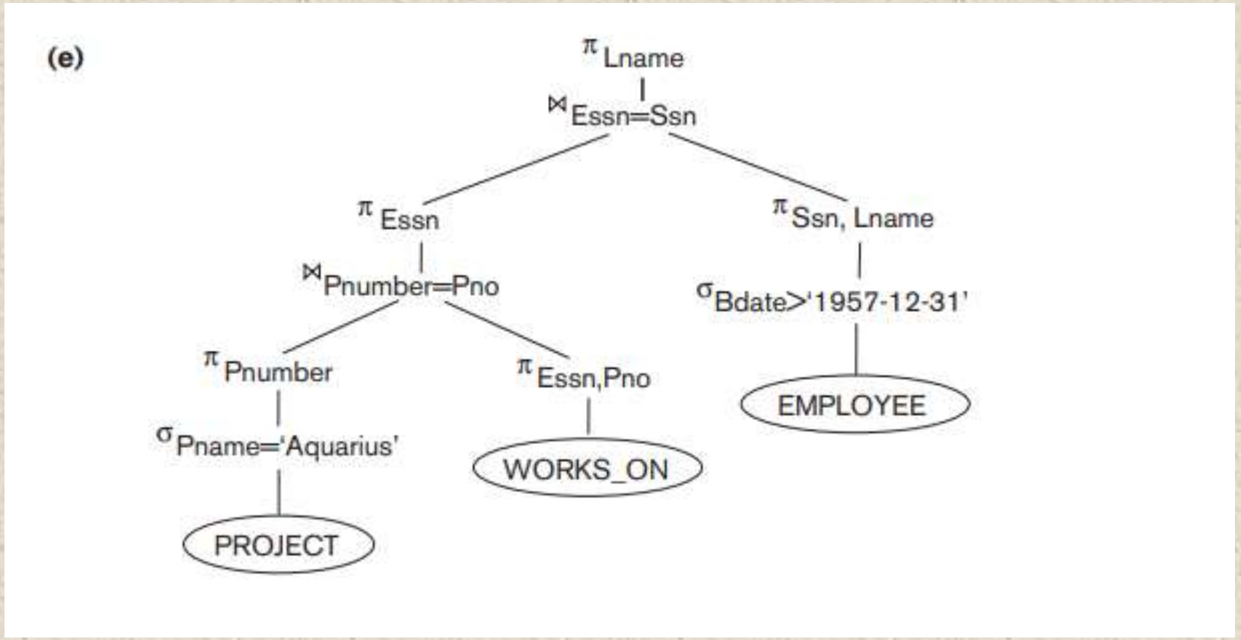
Figure 19.2 (continued)
Steps in converting a query tree during heuristic optimization. (d) Replacing CARTESIAN PRODUCT and SELECT with JOIN operations. (e) Moving PROJECT operations down the query tree.

(d)



π_{Lname}
$\bowtie_{\text{Essn=Ssn}}$
$\bowtie_{\text{Pnumber=Pno}}$
$\sigma_{\text{Bdate>'1957-12-31'}}$
$\sigma_{\text{Pname='Aquarius'}}$
WORKS_ON
EMPLOYEE
PROJECT

Heuristic Optimization of Query Trees


(e)
π_{Lname}
$\bowtie_{\text{Essn=Ssn}}$
π_{Essn}
$\pi_{\text{Ssn, Lname}}$
$\bowtie_{\text{Pnumber=Pno}}$
$\sigma_{\text{Bdate>'1957-12-31'}}$
π_{Pnumber}
$\pi_{\text{Essn,Pno}}$
EMPLOYEE
$\sigma_{\text{Pname='Aquarius'}}$
WORKS_ON
PROJECT

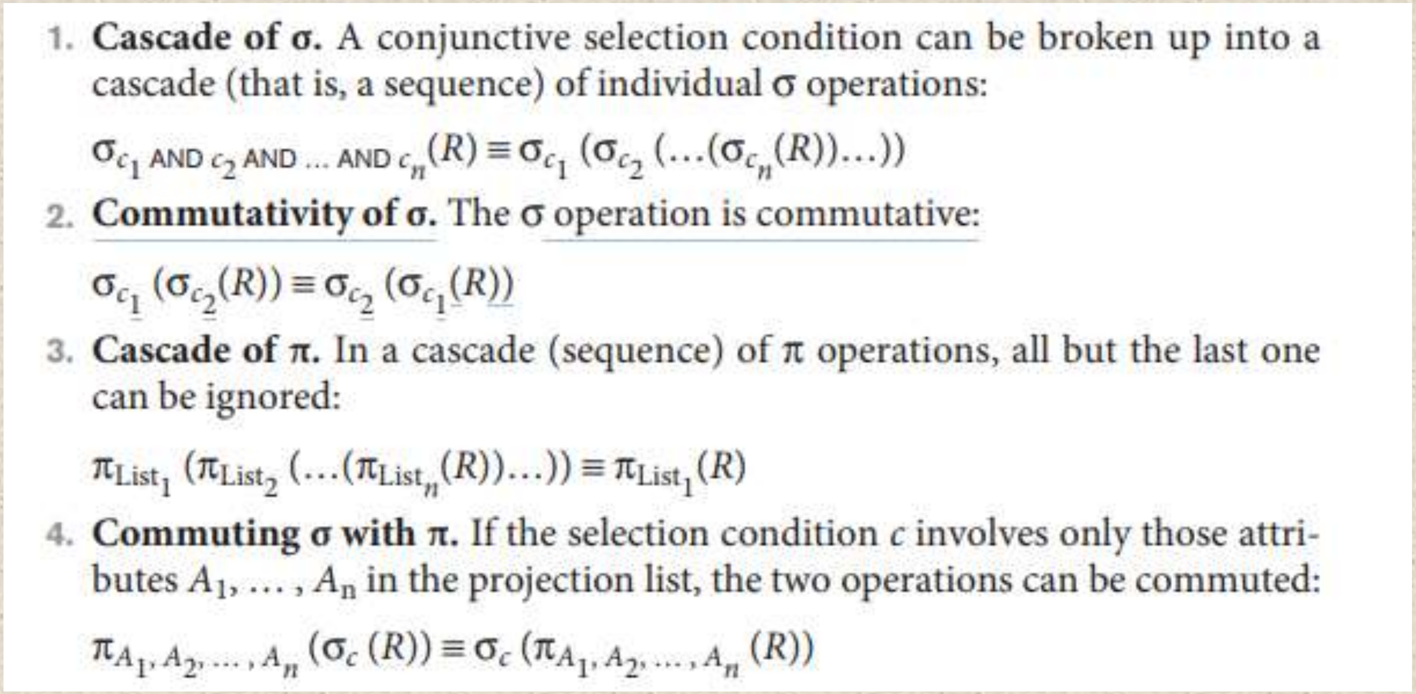
Heuristic Optimization of Query Trees

- As the preceding example demonstrates, a query tree can be transformed step by step into an equivalent query tree that is more efficient to execute. However, we must make sure that the transformation steps always lead to an equivalent query tree. To do this, the query optimizer must know which transformation rules preserve this equivalence.

General Transformation Rules for Relational Algebra Operations

- There are many rules for transforming relational algebra operations into equivalent ones. For query optimization purposes, we are interested in the meaning of the operations and the resulting relations. Hence, if two relations have the same set of attributes in a different order but the two relations represent the same information, we consider the relations to be equivalent.

General Transformation Rules for Relational Algebra Operations



1. **Cascade of σ.** A conjunctive selection condition can be broken up into a cascade (that is, a sequence) of individual σ operations:

   $\sigma_{c_1 \text{ AND } c_2 \text{ AND } \ldots \text{ AND } c_n}(R) \equiv \sigma_{c_1}(\sigma_{c_2}(\ldots(\sigma_{c_n}(R))\ldots))$

2. **Commutativity of σ.** The σ operation is commutative:

   $\sigma_{c_1}(\sigma_{c_2}(R)) \equiv \sigma_{c_2}(\sigma_{c_1}(R))$

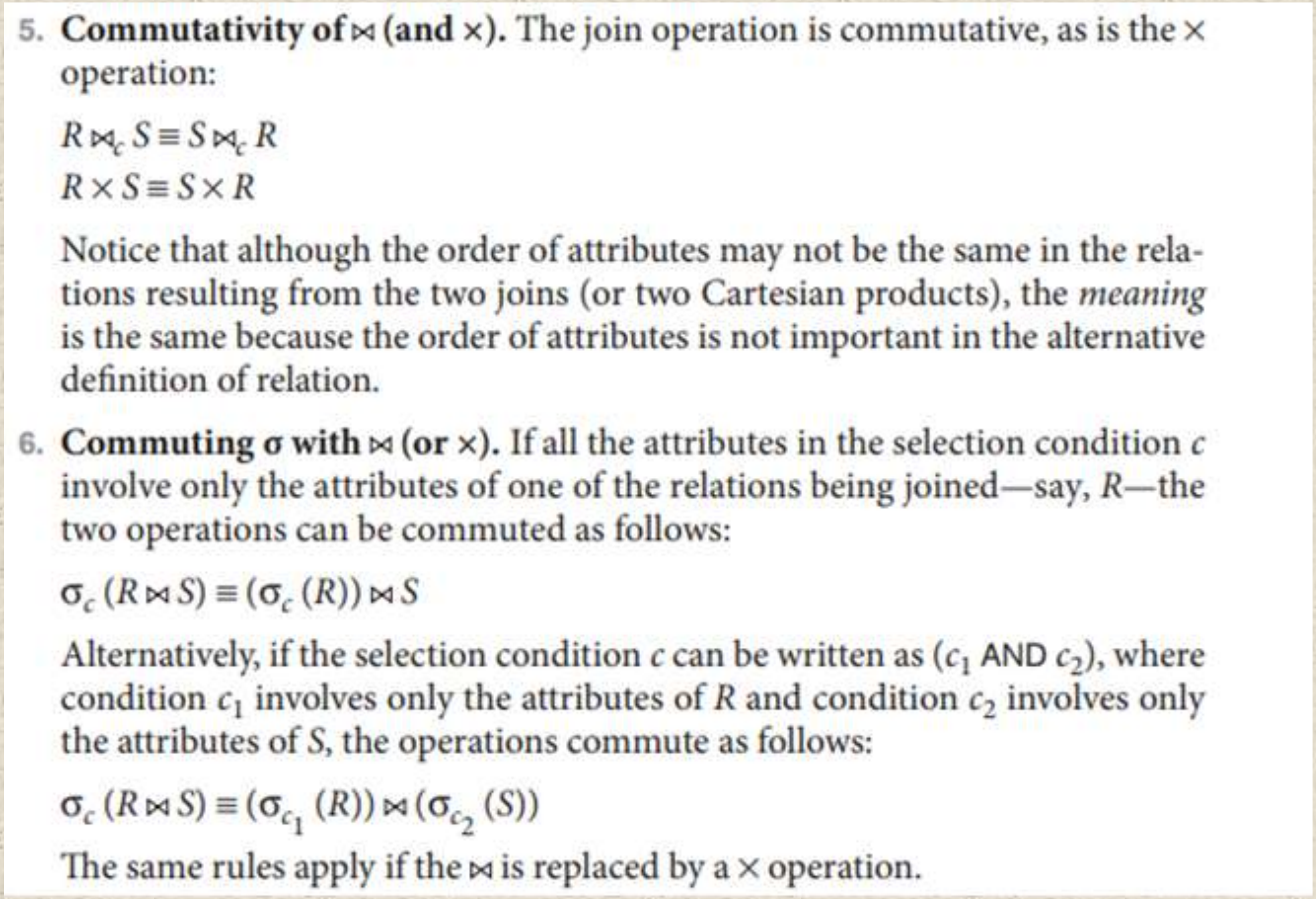
3. **Cascade of π.** In a cascade (sequence) of π operations, all but the last one can be ignored:

   $\pi_{\text{List}_1}(\pi_{\text{List}_2}(\ldots(\pi_{\text{List}_n}(R))\ldots)) \equiv \pi_{\text{List}_1}(R)$

4. **Commuting σ with π.** If the selection condition c involves only those attributes $A_1, \ldots, A_n$ in the projection list, the two operations can be commuted:

   $\pi_{A_1, A_2, \ldots, A_n}(\sigma_c(R)) \equiv \sigma_c(\pi_{A_1, A_2, \ldots, A_n}(R))$

General Transformation Rules for Relational Algebra Operations



5. **Commutativity of $\bowtie$ (and $\times$).** The join operation is commutative, as is the $\times$ operation:

   $R \bowtie_c S \equiv S \bowtie_c R$
   $R \times S \equiv S \times R$

   Notice that although the order of attributes may not be the same in the relations resulting from the two joins (or two Cartesian products), the *meaning* is the same because the order of attributes is not important in the alternative definition of relation.

6. **Commuting σ with $\bowtie$ (or $\times$).** If all the attributes in the selection condition c involve only the attributes of one of the relations being joined—say, R—the two operations can be commuted as follows:

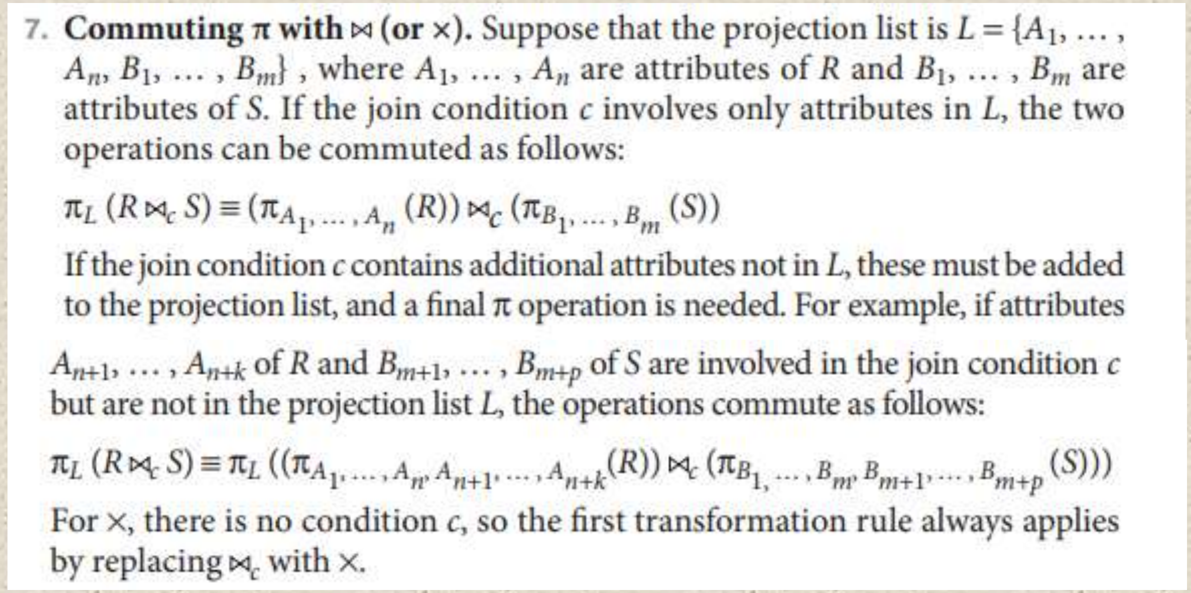
   $\sigma_c (R \bowtie S) \equiv (\sigma_c (R)) \bowtie S$

   Alternatively, if the selection condition c can be written as (c_1 AND c_2), where condition c_1 involves only the attributes of R and condition c_2 involves only the attributes of S, the operations commute as follows:

   $\sigma_c (R \bowtie S) \equiv (\sigma_{c_1} (R)) \bowtie (\sigma_{c_2} (S))$

   The same rules apply if the $\bowtie$ is replaced by a $\times$ operation.

General Transformation Rules for Relational Algebra Operations



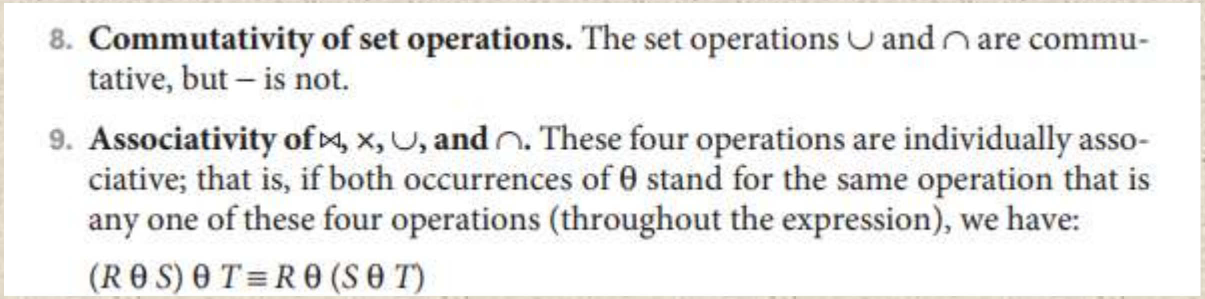
7. **Commuting π with $\bowtie$ (or $\times$).** Suppose that the projection list is $L = \{A_1, \ldots, A_n, B_1, \ldots, B_m\}$, where $A_1, \ldots, A_n$ are attributes of R and $B_1, \ldots, B_m$ are attributes of S. If the join condition c involves only attributes in L, the two operations can be commuted as follows:

$$\pi_L (R \bowtie_c S) \equiv (\pi_{A_1, \ldots, A_n} (R)) \bowtie_c (\pi_{B_1, \ldots, B_m} (S))$$

If the join condition c contains additional attributes not in L, these must be added to the projection list, and a final π operation is needed. For example, if attributes $A_{n+1}, \ldots, A_{n+k}$ of R and $B_{m+1}, \ldots, B_{m+p}$ of S are involved in the join condition c but are not in the projection list L, the operations commute as follows:

$$\pi_L (R \bowtie_c S) \equiv \pi_L ((\pi_{A_1, \ldots, A_n, A_{n+1}, \ldots, A_{n+k}}(R)) \bowtie_c (\pi_{B_1, \ldots, B_m, B_{m+1}, \ldots, B_{m+p}} (S)))$$

For $\times$, there is no condition c, so the first transformation rule always applies by replacing $\bowtie_c$ with $\times$.

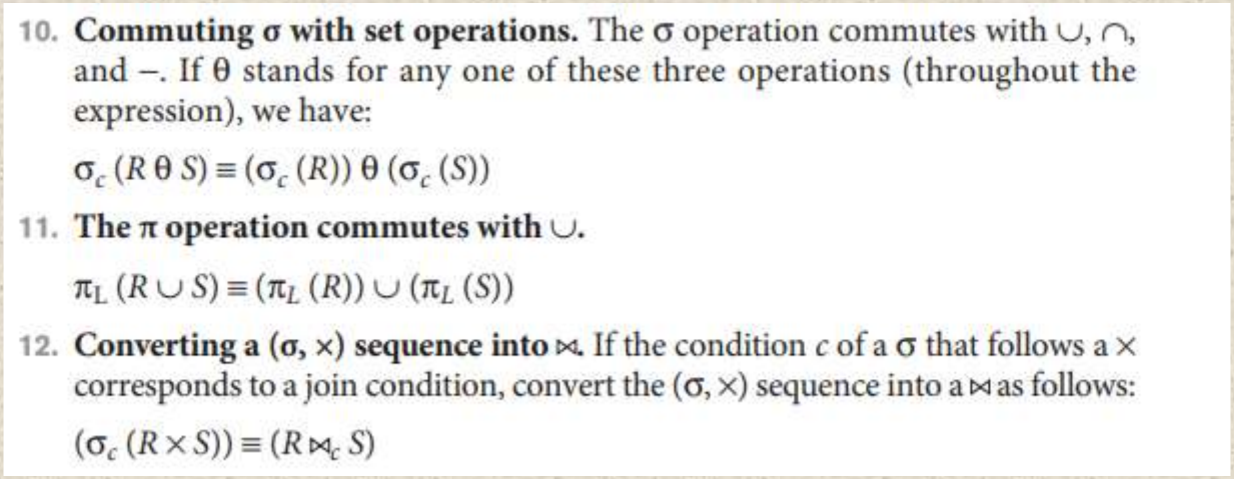


8. **Commutativity of set operations.** The set operations $\cup$ and $\cap$ are commutative, but $-$ is not.

9. **Associativity of $\bowtie$, $\times$, $\cup$, and $\cap$.** These four operations are individually associative; that is, if both occurrences of θ stand for the same operation that is any one of these four operations (throughout the expression), we have:

$$(R \,\theta\, S) \,\theta\, T \equiv R \,\theta\, (S \,\theta\, T)$$

General Transformation Rules for Relational Algebra Operations



10. **Commuting σ with set operations.** The σ operation commutes with ∪, ∩, and −. If θ stands for any one of these three operations (throughout the expression), we have:

$\sigma_c (R \; \theta \; S) \equiv (\sigma_c (R)) \; \theta \; (\sigma_c (S))$

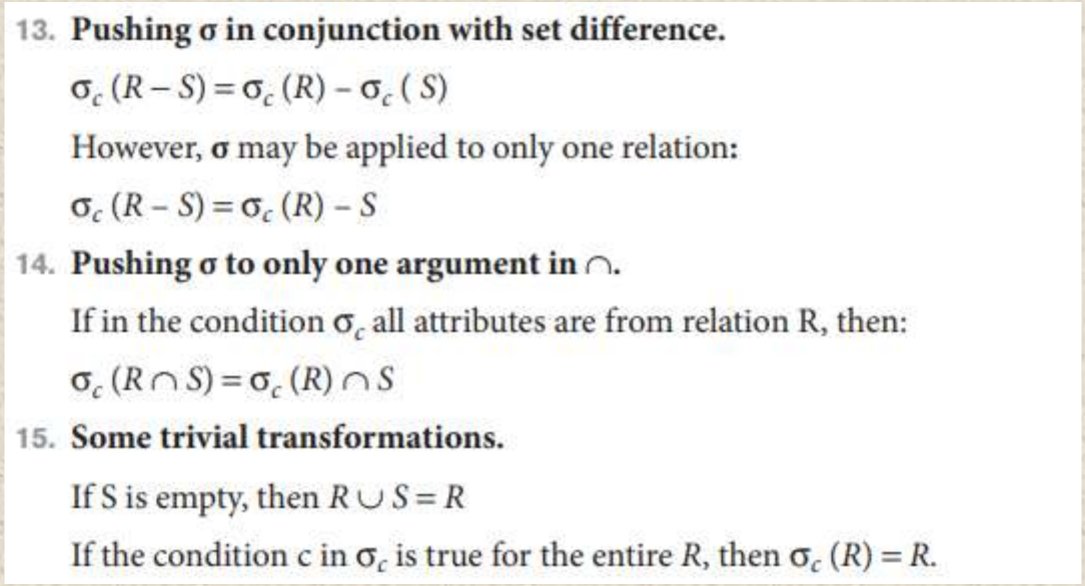
11. **The π operation commutes with ∪.**

$\pi_L (R \cup S) \equiv (\pi_L (R)) \cup (\pi_L (S))$

12. **Converting a (σ, ×) sequence into ⋈.** If the condition *c* of a σ that follows a × corresponds to a join condition, convert the (σ, ×) sequence into a ⋈ as follows:

$(\sigma_c (R \times S)) \equiv (R \bowtie_c S)$

General Transformation Rules for Relational Algebra Operations



13. **Pushing σ in conjunction with set difference.**

    $\sigma_c (R - S) = \sigma_c (R) - \sigma_c (S)$

    However, σ may be applied to only one relation:

    $\sigma_c (R - S) = \sigma_c (R) - S$

14. **Pushing σ to only one argument in ∩.**

    If in the condition σ_c all attributes are from relation R, then:

    $\sigma_c (R \cap S) = \sigma_c (R) \cap S$

15. **Some trivial transformations.**

    If S is empty, then $R \cup S = R$

    If the condition c in σ_c is true for the entire R, then $\sigma_c (R) = R$.

Outline of a Heuristic Algebraic Optimization Algorithm

- We can now outline the steps of an algorithm that utilizes some of the above rules to transform an initial query tree into a final tree that is more efficient to execute (in most cases).
  - Step 1: Using Rule 1, break up any SELECT operations with conjunctive conditions into a cascade of SELECT operations. This permits a greater degree of freedom in moving SELECT operations down different branches of the tree.
  - Step 2: Using Rules 2, 4, 6, and 10, 13, 14 concerning the commutativity of SELECT with other operations, move each SELECT operation as far down the query tree as is permitted by the attributes involved in the select condition. If the condition involves attributes from only one table, which means that it represents a selection condition, the operation is moved all the way to the leaf node that represents this table. If the condition involves attributes from two tables, which means that it represents a join condition, the condition is moved to a location down the tree after the two tables are combined

Outline of a Heuristic Algebraic Optimization Algorithm

- We can now outline the steps of an algorithm that utilizes some of the above rules to transform an initial query tree into a final tree that is more efficient to execute (in most cases).
  - Step 3: Using Rules 5 and 9 concerning commutativity and associativity of binary operations, rearrange the leaf nodes of the tree using the following criteria. First, position the leaf node relations with the most restrictive SELECT operations so they are executed first in the query tree representation. The definition of most restrictive SELECT can mean either the ones that produce a relation with the fewest tuples or with the smallest absolute size.
  - Step 4: Another possibility is to define the most restrictive SELECT as the one with the smallest selectivity; this is more practical because estimates of selectivities are often available in the DBMS catalog. Second, make sure that the ordering of leaf nodes does not cause CARTESIAN PRODUCT operations; for example, if the two relations with the most restrictive SELECT do not have a direct join condition between them, it may be desirable to change the order of leaf nodes to avoid Cartesian products.

Outline of a Heuristic Algebraic Optimization Algorithm

- We can now outline the steps of an algorithm that utilizes some of the above rules to transform an initial query tree into a final tree that is more efficient to execute (in most cases).
  - Step 5: Using Rule 12, combine a CARTESIAN PRODUCT operation with a subsequent SELECT operation in the tree into a JOIN operation, if the condition represents a join condition.
  - Step 6: Using Rules 3, 4, 7, and 11 concerning the cascading of PROJECT and the commuting of PROJECT with other operations, break down and move lists of projection attributes down the tree as far as possible by creating new PROJECT operations as needed. Only those attributes needed in the query result and in subsequent operations in the query tree should be kept after each PROJECT operation.

Outline of a Heuristic Algebraic Optimization Algorithm

- We can now outline the steps of an algorithm that utilizes some of the above rules to transform an initial query tree into a final tree that is more efficient to execute (in most cases).
  - Identify subtrees that represent groups of operations that can be executed by a single algorithm.

Outline of a Heuristic Algebraic Optimization Algorithm

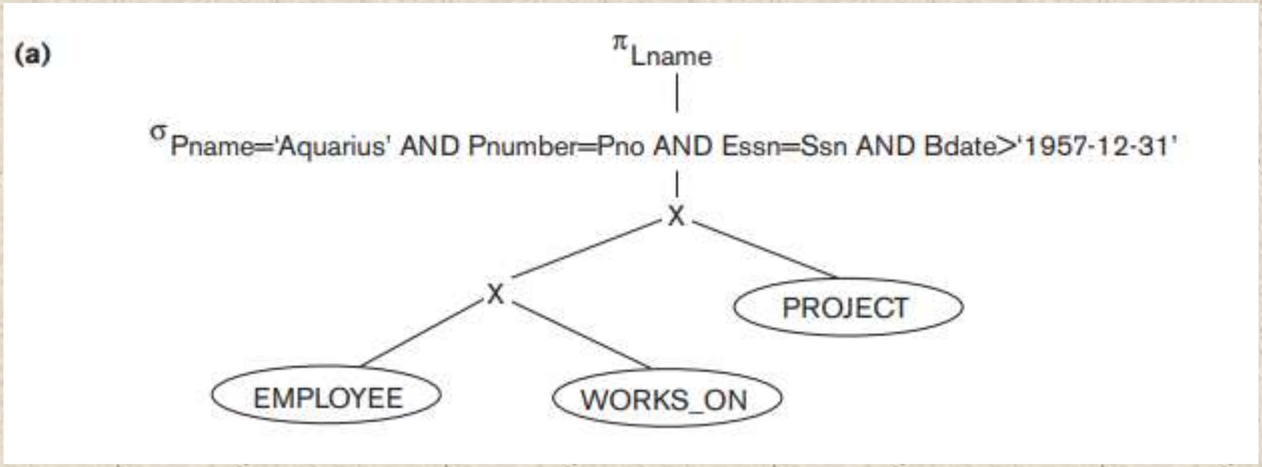
- In the example, Figure 19.2(b) shows the tree in Figure 19.2(a) after applying steps 1 and 2 of the algorithm;
- Figure 19.2(c) shows the tree after step 3; Figure 19.2(d) after step 4; and
- Figure 19.2(e) after step 5.
- In step 6, we may group together the operations in the subtree whose root is the operation π_{Essn} into a single algorithm. We may also group the remaining operations into another subtree, where the tuples resulting from the first algorithm replace the subtree whose root is the operation π_{Essn} , because the first grouping means that this subtree is executed first.

Outline of a Heuristic Algebraic Optimization Algorithm

SELECT E.Lname

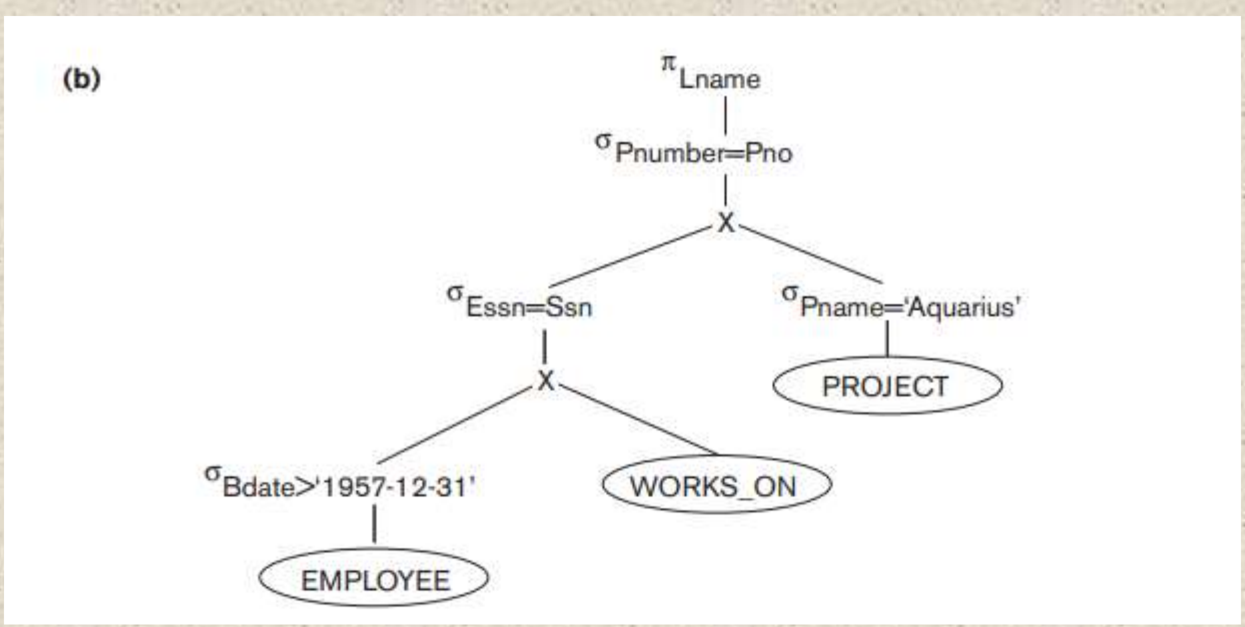
FROM EMPLOYEE AS E, WORKS_ON AS W, PROJECT AS P

WHERE P.Pname='Aquarius' AND P.Pnumber = W.Pno AND E.Essn=W.Ssn AND E.Bdate > '1957-12-31';


(a)
π_{Lname}
$\sigma_{Pname='Aquarius' \text{ AND } Pnumber=Pno \text{ AND } Essn=Ssn \text{ AND } Bdate>'1957-12-31'}$
X
X
PROJECT
EMPLOYEE
WORKS_ON

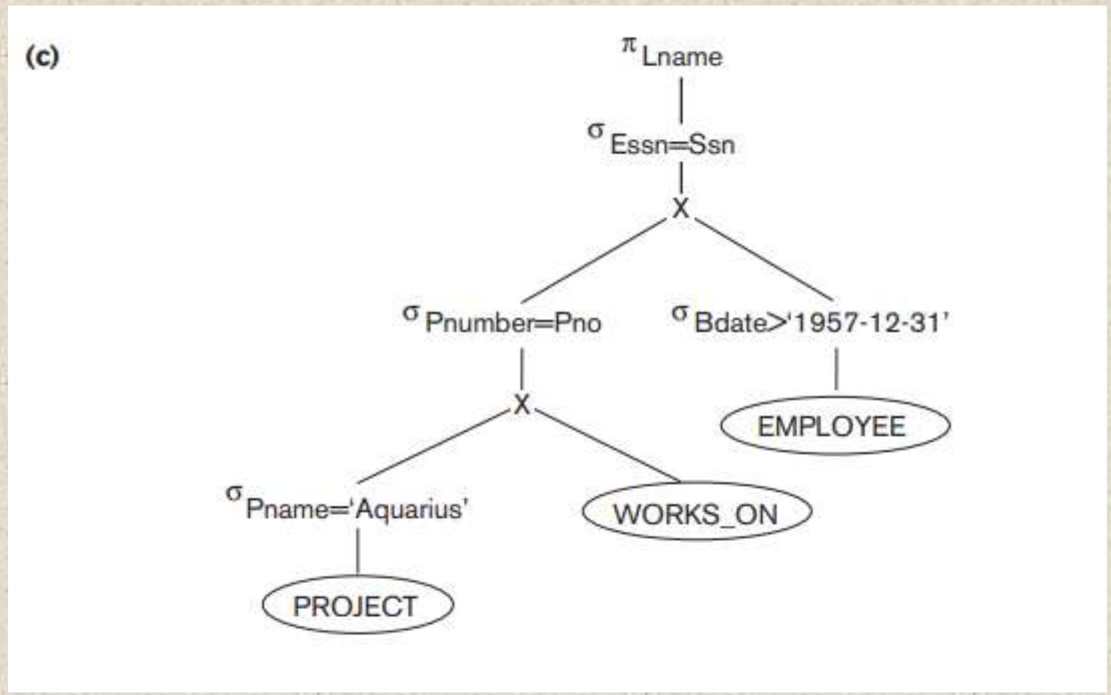
Outline of a Heuristic Algebraic Optimization Algorithm

- In the example, Figure 19.2(b) shows the tree in Figure 19.2(a) after applying steps 1 and 2 of the algorithm;



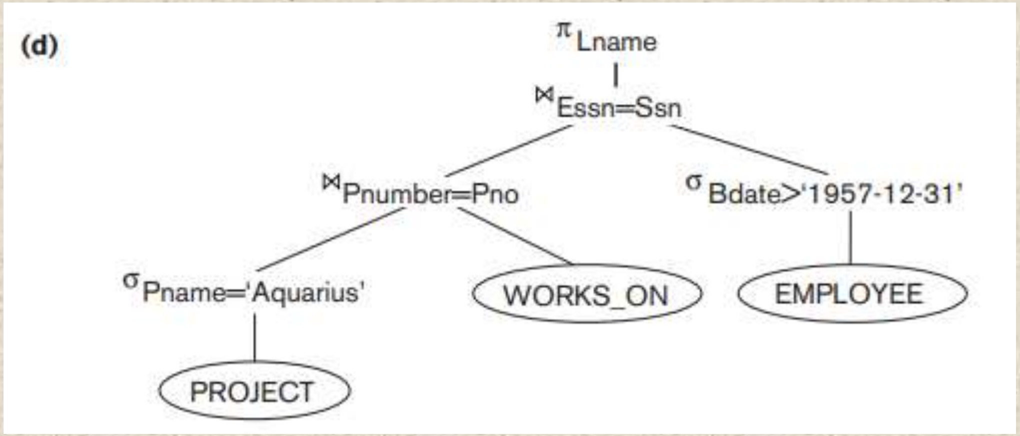
Outline of a Heuristic Algebraic Optimization Algorithm

- Figure 19.2(c) shows the tree after step 3;



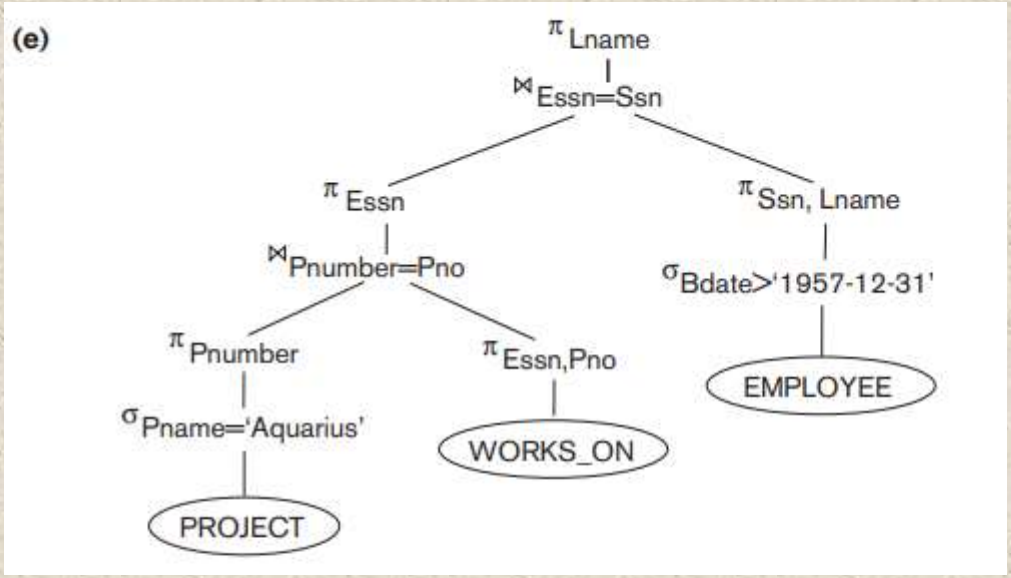
Outline of a Heuristic Algebraic Optimization Algorithm

- Figure 19.2(d) after step 4;


(d)
π_{Lname}
$\bowtie_{\text{Essn=Ssn}}$
$\bowtie_{\text{Pnumber=Pno}}$
$\sigma_{\text{Bdate>'1957-12-31'}}$
$\sigma_{\text{Pname='Aquarius'}}$
WORKS_ON
EMPLOYEE
PROJECT

Outline of a Heuristic Algebraic Optimization Algorithm

- Figure 19.2(e) after step 5.



Outline of a Heuristic Algebraic Optimization Algorithm

- In step 6, we may group together the operations in the subtree whose root is the operation π_{Essn} into a single algorithm. We may also group the remaining operations into another subtree, where the tuples resulting from the first algorithm replace the subtree whose root is the operation π_{Essn} , because the first grouping means that this subtree is executed first.

Outline of a Heuristic Algebraic Optimization Algorithm

- The main heuristic is to apply first the operations that reduce the size of intermediate results.
- This includes performing as early as possible SELECT operations to reduce the number of tuples and PROJECT operations to reduce the number of attributes—by moving SELECT and PROJECT operations as far down the tree as possible.
- Additionally, the SELECT and JOIN operations that are most restrictive—that is, result in relations with the fewest tuples or with the smallest absolute size—should be executed before other similar operations. The latter rule is accomplished through reordering the leaf nodes of the tree among themselves while avoiding Cartesian products, and adjusting the rest of the tree appropriately

Cost-Based Optimization

- A query optimizer does not depend solely on heuristic rules or query transformations; it also estimates and compares the costs of executing a query using different execution strategies and algorithms, and it then chooses the strategy with the lowest cost estimate. . For this approach to work, accurate cost estimates are required so that different strategies can be compared fairly and realistically.
- It uses traditional optimization techniques that search the solution space to a problem for a solution that minimizes an objective (cost) function. The cost functions used in query optimization are estimates and not exact cost functions, so the optimization may select a query execution strategy that is not the optimal (absolute best) one.

Cost-Based Optimization

- Cost-based query optimization is an overall process of choosing the most efficient means of executing a SQL statement based on overall cost of the query. The efficient execution is the execution with minimum cost. To find the cost of query execution plan, the optimization technique uses database statistics.

Cost Components for Query Execution

- **Access cost to secondary storage**. This is the cost of transferring (reading and writing) data blocks between secondary disk storage and main memory buffers. This is also known as disk I/O (input/output) cost. The cost of searching for records in a disk file depends on the type of access structures on that file, such as ordering, hashing, and primary or secondary indexes. In addition, factors such as whether the file blocks are allocated contiguously on the same disk cylinder or scattered on the disk affect the access cost.

Cost Components for Query Execution

- **Disk storage cost.** This is the cost of storing on disk any intermediate files that are generated by an execution strategy for the query.
- **Computation cost.** This is the cost of performing in-memory operations on the records within the data buffers during query execution. Such operations include searching for and sorting records, merging records for a join or a sort operation, and performing computations on field values. This is also known as CPU (central processing unit) cost.

Cost Components for Query Execution

- **Memory usage cost.** This is the cost pertaining to the number of main memory buffers needed during query execution.
- **Communication cost.** This is the cost of shipping the query and its results from the database site to the site or terminal where the query originated. In distributed databases, it would also include the cost of transferring tables and results among various computers during query evaluation.

Cost Components for Query Execution

- For large databases, the main emphasis is often on minimizing the access cost to secondary storage. Simple cost functions ignore other factors and compare different query execution strategies in terms of the number of block transfers between disk and main memory buffers.
- For smaller databases, where most of the data in the files involved in the query can be completely stored in memory, the emphasis is on minimizing computation cost.
- In distributed databases, where many sites are involved, communication cost must be minimized. It is difficult to include all the cost components in a (weighted) cost function because of the difficulty of assigning suitable weights to the cost components.
- This is why some cost functions consider a single factor only—disk access.

Cost Based Optimization

- Consider two relations
- employee (emp_no, emp_name, emp_address, position, salary, branch_no)
- (branch_no, branch_city, branch_address, city) with a member of employee can only work at one branch.
- Consider an SQL query as given below.

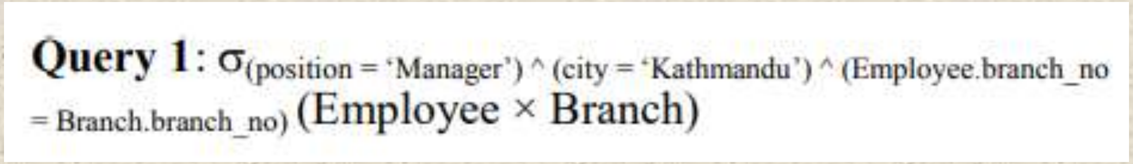
SELECT * FROM Employee, Branch

WHERE Employee.branch_no = Branch.branch_no AND Employee.position = ‘Manager’ AND Branch.city = ‘Kathmandu’;

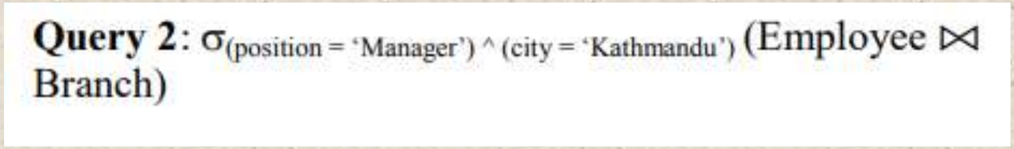
Cost Based Optimization

- SELECT * FROM Employee, Branch

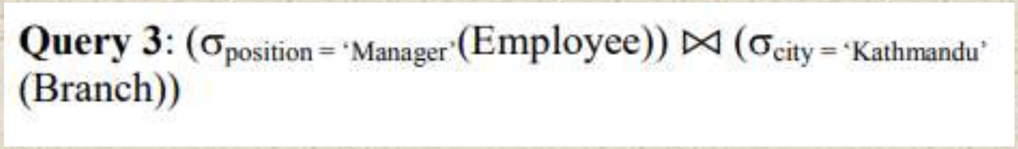
WHERE Employee.branch_no = Branch.branch_no AND Employee.position = ‘Manager’ AND Branch.city = ‘Kathmandu’;



Query 1: $\sigma_{(position = \text{‘Manager’}) \wedge (city = \text{‘Kathmandu’}) \wedge (Employee.branch\_no = Branch.branch\_no)}$ (Employee × Branch)



Query 2: $\sigma_{(position = \text{‘Manager’}) \wedge (city = \text{‘Kathmandu’})}$ (Employee $\bowtie$ Branch)



Query 3: ($\sigma_{position = \text{‘Manager’}}$(Employee)) $\bowtie$ ($\sigma_{city = \text{‘Kathmandu’}}$ (Branch))

Cost Based Optimization

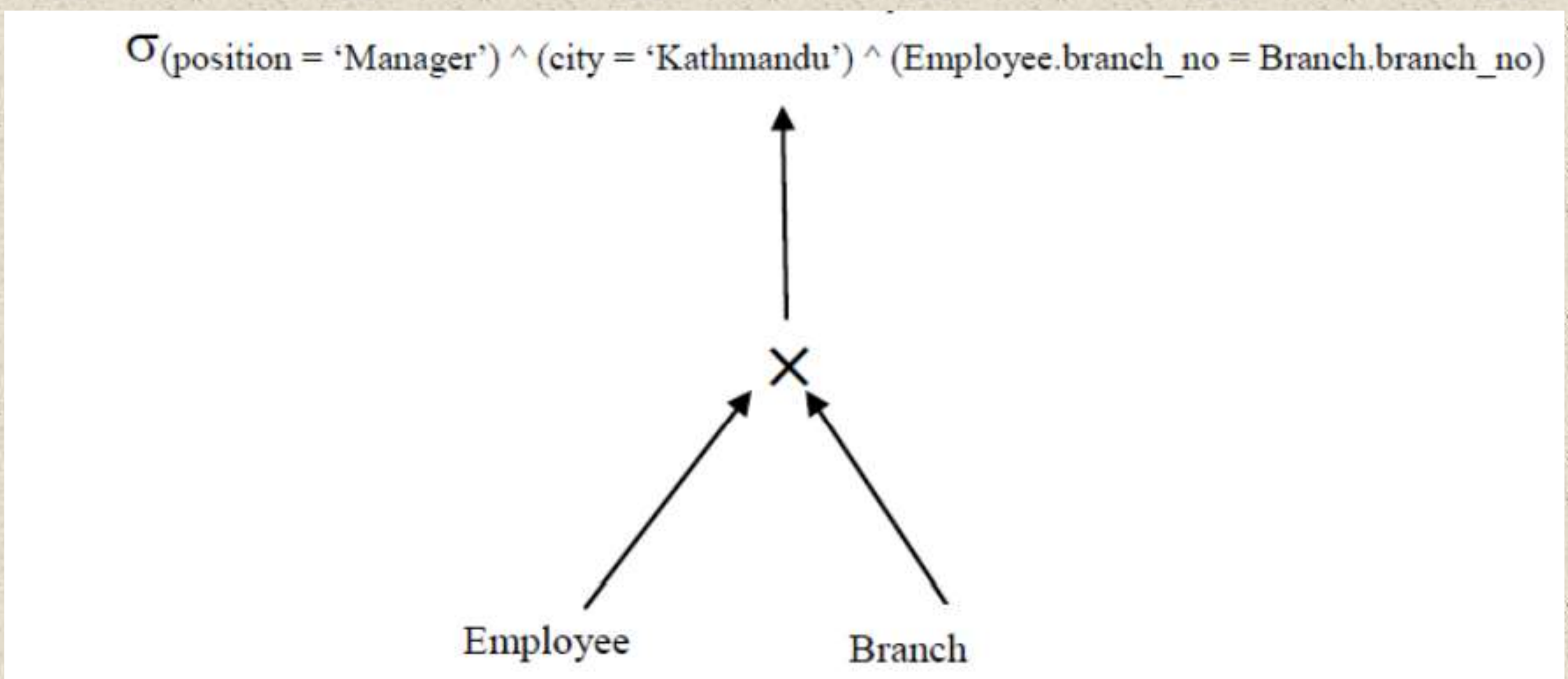

σ(position = 'Manager') ^ (city = 'Kathmandu') ^ (Employee.branch_no = Branch.branch_no)
×
Employee
Branch

Fig. 2. Relational algebra tree for query 1

Cost Based Optimization

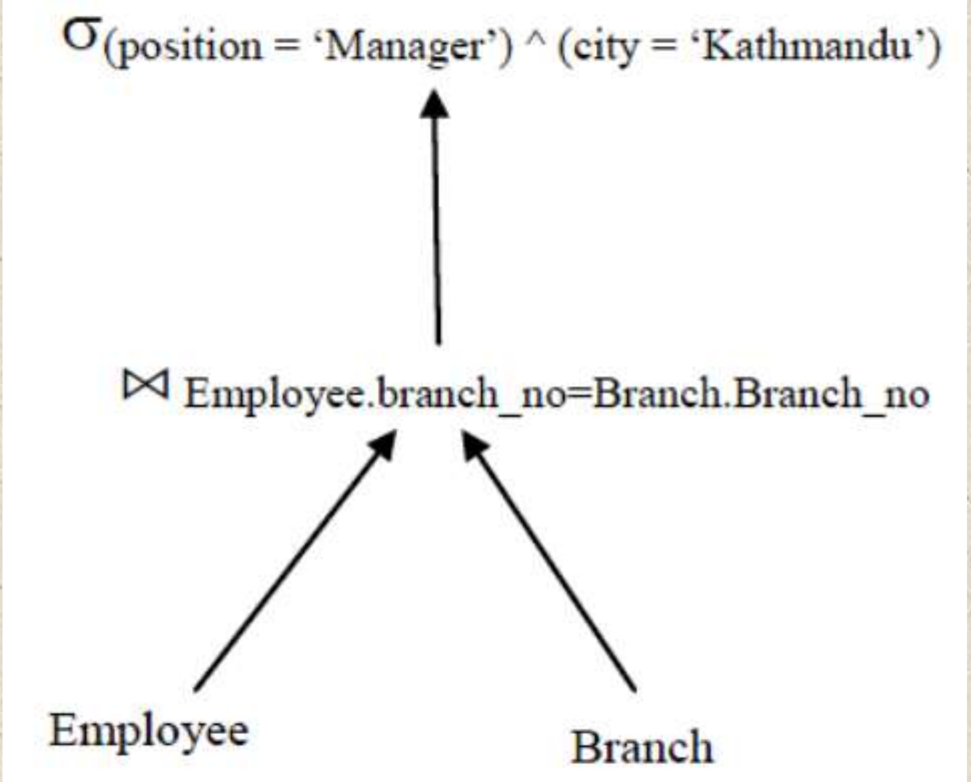

σ(position = 'Manager') ^ (city = 'Kathmandu')
⋈ Employee.branch_no=Branch.Branch_no
Employee
Branch

Fig. 3. Relational algebra tree for query 2

Cost Based Optimization

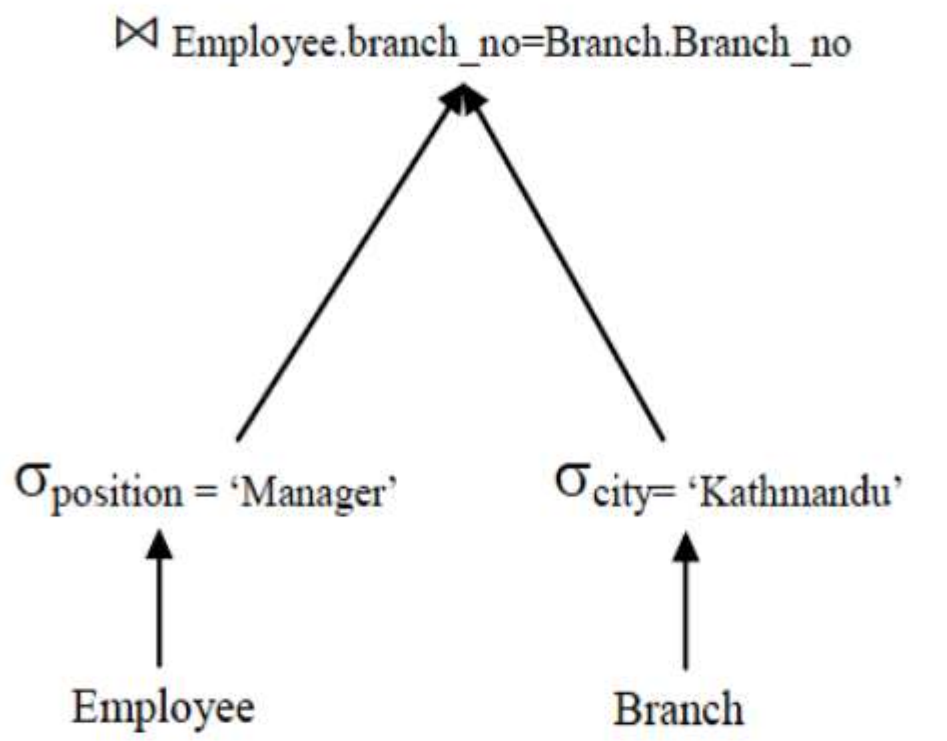


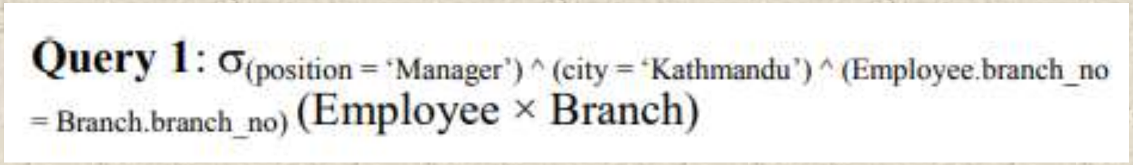
Fig. 4. Relational algebra tree for query 3

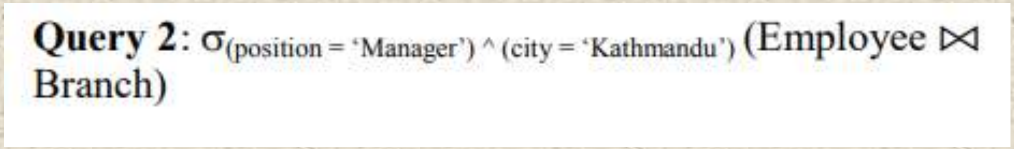
Cost Based Optimization

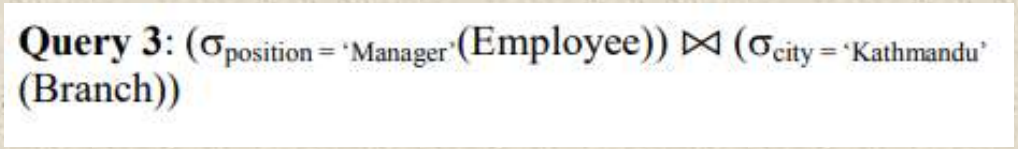
- Suppose there are 2000 tuples in Employee, 20 tuples in Branch, 20 Managers (one for each branch), and 10 Kathmandu branches.
- To compare these three queries, we assume number of disk accesses. We also assume that there are no indexes or sort keys on either relation.
- The results of any intermediate operations are stored on disk.
- The cost of the final write is ignored because it is the same in each query.
- We further assume that tuples are accessed one at a time (although in practice disk accesses would be based on blocks, which would typically contain several tuples), and main memory is large enough to process entire relations for each relational algebra operation.

Cost Based Optimization

- The query1 calculates the Cartesian product of Employee and Branch, which requires (2000 + 20) disk accesses to read these two relations, and creates a relation with (2000 × 20) tuples. We then have to read each of these tuples again to test them against the selection predicate at a cost of another (2000 × 20) disk accesses, giving a total cost of (2000 + 20) + 2 × (2000 × 20) = 82020 disk accesses.

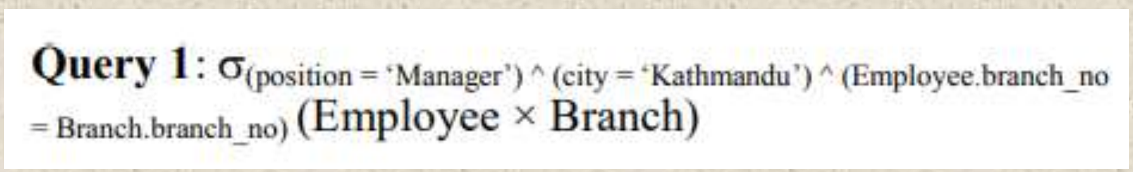

Query 1: $\sigma_{(\text{position} = \text{'Manager'}) \wedge (\text{city} = \text{'Kathmandu'}) \wedge (\text{Employee.branch\_no} = \text{Branch.branch\_no})}(\text{Employee} \times \text{Branch})$


Query 2: $\sigma_{(\text{position} = \text{'Manager'}) \wedge (\text{city} = \text{'Kathmandu'})}(\text{Employee} \bowtie \text{Branch})$

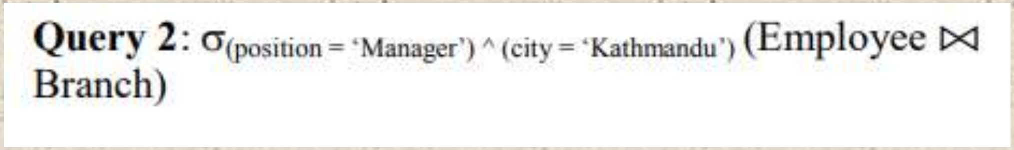

Query 3: $(\sigma_{\text{position} = \text{'Manager'}}(\text{Employee})) \bowtie (\sigma_{\text{city} = \text{'Kathmandu'}}(\text{Branch}))$

Cost Based Optimization

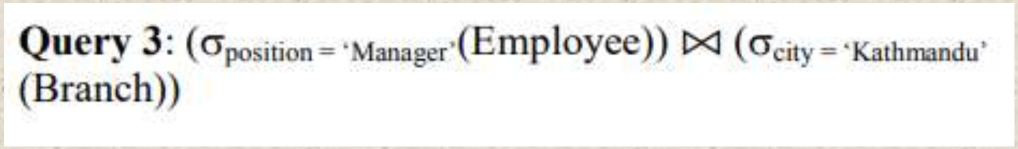
- The query 2 joins Employee and Branch which again requires (2000 + 20) disk accesses to read each of the relations. The Join of these two relations has 2000 tuples, one for each member of Employee. Consequently, the Selection operation requires 2000 disk accesses to read the result of the join, giving a total cost of (2000 + 20) + 2 × (2000) = 6020 disk accesses.



Query 1: $\sigma_{(position = 'Manager') \wedge (city = 'Kathmandu') \wedge (Employee.branch\_no = Branch.branch\_no)}$ (Employee × Branch)



Query 2: $\sigma_{(position = 'Manager') \wedge (city = 'Kathmandu')}$ (Employee $\bowtie$ Branch)



Query 3: ($\sigma_{position = 'Manager'}$(Employee)) $\bowtie$ ($\sigma_{city = 'Kathmandu'}$ (Branch))

Cost Based Optimization

- The query 3 first reads each Employee tuple to determine the Manager tuples, which requires 2000 disk accesses and produces a relation with 20 tuples. Similarly, the second Selection operation reads each Branch tuple to determine the Kathmandu branches, which requires 20 disk accesses and produces a relation with 10 tuples. The final operation is the join of the reduced Employee and Branch relations, which requires (20 + 10) disk accesses, giving a total cost of (2000 + 20) + (20 + 10) + (20 + 10) = 2080 disk accesses.

Cost Based Optimization

- From the calculations above, it is clear that query 3 is the most efficient query and is 2.89 times faster than query 2 and 39.43 times faster than the query1.