Programme M.Sc. in Computer Science

Course – Advanced DBMS

Module - Query processing

BIBEK RANJAN GHOSH

(Assistant Professor, Department of Computer Science)

Steps of query processing

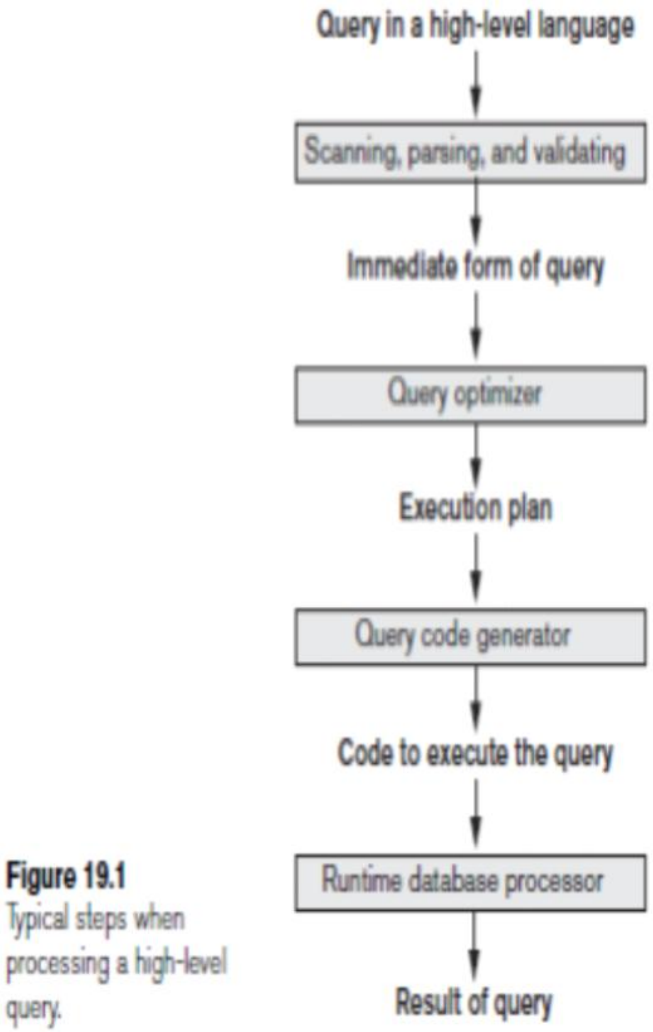


Figure 19.1
Typical steps when processing a high-level query.

Code can be:

Executed directly (interpreted mode)

Stored and executed later whenever needed (compiled mode)

1.The **scanner** identifies the query tokens—such as SQL keywords, attribute names, and relation names—that appear in the text of the query.

2.The **parser** checks the query syntax to determine whether it is formulated according to the syntax rules (rules of grammar) of the query language.

3. The query must also be **validated (semantically)** by checking that all attribute and relation names are valid and semantically meaningful names in the schema of the particular database being queried.

4. An internal representation of the query is then created, usually as a tree data structure called a **query tree**. It is also possible to represent the query using a graph data structure called a **query graph**.

5. The DBMS must then devise an **execution strategy** or **query plan** for retrieving the results of the query from the database files.

6. A query typically has many possible execution strategies, and the process of choosing a suitable one for processing a query is known as **query optimization**.

7.The **query optimizer** module has the task of producing a good execution plan, and the **code generator** generates the code to execute that plan.

8.The **runtime database processor** has the task of running (executing) the query code, whether in compiled or interpreted mode, to produce the query result.

The term *optimization* is actually a misnomer because in some cases the chosen execution plan is not the optimal (or absolute best) strategy—it is just a *reasonably efficient strategy* for executing the query.

There are two main techniques that are employed during query optimization.

The **first** technique is based on **heuristic rules** for ordering the operations in a query execution strategy. A heuristic is a rule that works well in most cases but is not guaranteed to work well in every case. The rules typically reorder the operations in a query tree.

The **second** technique involves **systematically estimating** the cost of different execution strategies and choosing the execution plan with the lowest cost estimate.

These techniques are usually combined in a query optimizer.

1) Translating SQL Queries into Relational Algebra

In practice, SQL is the query language that is used in most commercial RDBMSs.

An SQL query is first translated into an equivalent extended relational algebra expression—represented as a query tree data structure—that is then optimized.

Typically, SQL queries are decomposed into ***query blocks,*** which form the basic units 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 clauses if these are part of the block.

Hence, nested queries within a query are identified as separate query blocks.

Example:

Consider the following SQL query on the EMPLOYEE relation

SELECT Lname, Fname **FROM** EMPLOYEE

WHERE Salary > (**SELECT MAX** (Salary) **FROM** EMPLOYEE **WHERE** Dno=5);

The query includes a nested subquery and hence would be decomposed into two blocks.

The **inner block** is:

(**SELECT MAX** (Salary) **FROM** EMPLOYEE **WHERE** Dno=5) This retrieves the highest salary in department 5.

The **outer query** block is:

SELECT Lname, Fname **FROM** EMPLOYEE **WHERE** Salary > c

where c represents the result returned from the inner block.

The inner block could be translated into the following extended relational algebra expression: $\Im_{\text{MAX Salary}}(\sigma_{\text{Dno=5}}(\text{EMPLOYEE}))$

and the outer block into the expression: $\pi_{\text{Lname,Fname}}(\sigma_{\text{Salary}>c}(\text{EMPLOYEE}))$

The *query optimizer* would then choose an execution plan for each query block.

2) Algorithms for External Sorting

Sorting is one of the primary algorithms used in query processing. **External sorting** refers to sorting algorithms that are suitable for large files of records stored on disk that do not fit entirely in main memory, such as most database files.

The typical external sorting algorithm uses a **sort-merge strategy.**

The buffer space in main memory an area in the computer's main memory that is controlled by the DBMS. The buffer space is divided into individual buffers. One buffer can hold the contents of exactly *one disk block*.

The basic algorithm consists of two phases: the **sorting phase** and the **merging phase.**

i) Sorting phase Portions of the file that can fit in the available buffer space are read into main memory, sorted using an *internal* sorting algorithm, and written back to disk as temporary sorted subfiles (or runs). The size of each run and the number of initial runs (nR) are dictated by the number of file blocks (b) and the available buffer space (nB).

For example, if the number of available main memory buffers $nB = 5$ disk blocks and the size of the file $b = 1024$ disk blocks, then $nR = \lceil (b/nB) \rceil$ or 205 initial runs each of size 5 blocks (except the last run which will have only 4 blocks). Hence, after the sorting phase, 205 sorted runs (or 205 sorted subfiles of the original file) are stored as temporary subfiles on disk.

ii) **Merging phase**, The sorted runs are merged during one or more **merge passes**. Each merge pass can have one or more merge steps. The **degree of merging (*dM*)** is the number of sorted subfiles that can be merged in each merge step.

During each merge step, one buffer block is needed to hold one disk block from each of the sorted subfiles being merged, and one additional buffer is needed for containing one disk block of the merge result, which will produce a larger sorted file that is the result of merging several smaller sorted subfiles. Hence, *dM* is the smaller of ($nB - 1$) and nR, and the number of merge passes is $\lceil (\log_{dM}(nR)) \rceil$.

In our example where $nB = 5$, $dM = 4$ (four-way merging), so the 205 initial sorted runs would be merged 4 at a time in each step into 52 larger sorted subfiles at the end of the first merge pass.

These 52 sorted files are then merged 4 at a time into 13 sorted files, which are then merged into 4 sorted files, and then finally into 1 fully sorted file, which means that *four passes* are needed.

Sort Merge Algorithm:

```
set     i ← 1;
        j ← b;          {size of the file in blocks}
        k ← nB;         {size of buffer in blocks}
        m ← ⌈(j/k)⌉;

{Sorting Phase}
while (i ≤ m)
do {
        read next k blocks of the file into the buffer or if there are less than k blocks
           remaining, then read in the remaining blocks;
        sort the records in the buffer and write as a temporary subfile;
        i ← i + 1;
}

{Merging Phase: merge subfiles until only 1 remains}
set     i ← 1;
        p ← ⌈log(k−1) m⌉   {p is the number of passes for the merging phase}
        j ← m;
while (i ≤ p)
do {
        n ← 1;
        q ← ⌈j/(k−1)⌉;  {number of subfiles to write in this pass}
        while (n ≤ q)
        do {
           read next k−1 subfiles or remaining subfiles (from previous pass)
                one block at a time;
           merge and write as new subfile one block at a time;
           n ← n + 1;
        }
        j ← q;
        i ← i + 1;
}
```

Figure 19.2
Outline of the sort-merge algorithm for external sorting.

Performance

The performance of the sort-merge algorithm can be measured in the number of

disk block reads and writes (between the disk and main memory) before the sorting

of the whole file is completed.

The following formula approximates this cost:

$(2 * b) + (2 * b * (\log_{dM} (nR))$

The first term $(2 * b)$ represents the number of block accesses for the sorting phase, since each file block is accessed twice: once for reading into a main memory buffer and once for writing the sorted records back to disk into one of the sorted subfiles.

The second term represents the number of block accesses for the merging phase. During each merge pass, a number of disk blocks approximately equal to the original file blocks b is read and written. Since the number of merge passes is $(\log_{dM} (nR))$we get the total merge cost of $(2 * b * (\log dM\ nR))$.

3) Implementation of SELECT

Algorithms for executing a SELECT operation is basically a search operation to locate the records in a disk file that satisfy a certain condition.

Search algorithms depend on the physical file organization (primary and secondary) and selection conditions. We shall use the following queries for discussion.

OP1: $\sigma_{Ssn = '123456789'}(EMPLOYEE)$

OP2: $\sigma_{Dnumber > 5}(DEPARTMENT)$

OP3: $\sigma_{Dno = 5}(EMPLOYEE)$

OP4: $\sigma_{Dno = 5 \text{ AND } Salary > 30000 \text{ AND } Sex = 'F'}(EMPLOYEE)$

OP5: $\sigma_{Essn='123456789' \text{ AND } Pno = 10}(WORKS\_ON)$

Search methods for simple selection

a) S1—Linear search (brute force algorithm). Each disk block is read into a main memory buffer, and then a search through the records within the disk block is conducted in main memory.

b) S2—Binary search. If the selection condition involves an equality comparison

on a key attribute on which the file is ordered, binary search can be used.

An example is OP1 if Ssn is the ordering attribute for the EMPLOYEE file.

c) **S3a—Using a primary index.** If the selection condition involves an equality comparison on a **key attribute** with a primary index—for example, Ssn ='123456789' in OP1—use the primary index to retrieve the record. Note that this condition retrieves a single record (at most).

d) **S3b—Using a hash key.** If the selection condition involves an equality comparison on a **key attribute** with a hash key—for example, Ssn = '123456789' in OP1—use the hash key to retrieve the record. Note that this condition retrieves a single record (at most).

e)**S4—Using a primary index to retrieve multiple records.** If the comparison condition is >, >=, <, or <= on a key field with a primary index—for example, Dnumber > 5 in OP2—use the index to find the record satisfying the corresponding equality condition (Dnumber = 5), then retrieve all subsequent records in the (ordered) file. For the condition Dnumber < 5, retrieve all the preceding records.

f) **S5—Using a clustering index to retrieve multiple records.** If the selection condition involves an equality comparison on a **nonkey attribute** with a clustering index—for example, Dno = 5 in OP3—use the index to retrieve all the records satisfying the condition.

g) **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 is a **key** (has unique values) or to retrieve multiple records if the indexing field is **not a key**. This can also be used for comparisons involving >, >=, <, or <=.

Search Methods for Complex Selection

If a condition of a SELECT operation is a **conjunctive condition**—that is, if it is made up of several simple conditions connected with the AND logical connective such as OP4 above—the DBMS can use the following additional methods to implement the operation:

h) S7—Conjunctive selection using an individual index. If an attribute involved in any **single simple condition** in the conjunctive select 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 select condition.

i) S8—Conjunctive selection using a composite index. If two or more attributes are involved in equality conditions in the conjunctive select condition and a composite index (or hash structure) exists on the combined fields— for example, if an index has been created on the composite key (Essn, Pno) of the WORKS_ON file for OP5—we can use the index directly.

j) **S9—Conjunctive selection by intersection of record pointers.** If secondary indexes (or other access paths) are available on more than one of the fields involved in simple conditions in the conjunctive select condition, and if the indexes include record pointers (rather than block pointers), then each index can be used to retrieve the **set of record pointers** that satisfy the individual condition. The **intersection** of these sets of record pointers gives the record pointers that satisfy the conjunctive select condition, which are then used to retrieve those records directly.

When the optimizer is choosing between multiple simple conditions in a conjunctive select condition, it typically considers the ***selectivity*** of each condition.

The **selectivity (*sl*)** is defined as the ratio of the number of records (tuples) that satisfy the condition to the total number of records (tuples) in the file (relation), and thus is a number between zero and one.

Estimates of selectivities are often kept in the DBMS catalog and are used by the optimizer.

For example, for an equality condition on a key attribute of relation $r(R)$, $sl = 1/|r(R)|$, where $|r(R)|$ is the number of tuples in relation $r(R)$.

For an equality condition on a nonkey attribute with *i distinct values, s* can be estimated by sl=($|r(R)|/i$)/$|r(R)|$ or $1/i$. $|r(R)|/i$ records will satisfy an equality condition on this attribute.

In general, the number of records satisfying a selection condition with selectivity *sl* is estimated to be $|r(R)| * sl$.

The smaller this estimate is, the higher the desirability of using that condition first to retrieve records

Disjunctive Selection Conditions.

Compared to a conjunctive selection condition, a **disjunctive condition** (where simple conditions are connected by the OR logical connective) is much harder to process and optimize.

A DBMS will have available many of the methods discussed above, and typically many additional methods. The query optimizer must choose the appropriate one for executing each SELECT operation in a query. This optimization uses formulas that estimate the costs for each available access method. The optimizer chooses the access method with the lowest estimated cost.

4) Implementation of JOIN

The JOIN operation is one of the most time-consuming operations in query processing. Many of the join operations encountered in queries are of the EQUIJOIN and NATURAL JOIN varieties.

There are many possible ways to implement a **two-way join**, which is a join on two files. Joins involving more than two files are called **multiway joins.**

The algorithms we discuss next are for a join operation of the form

$$R \bowtie_{A=B} S$$

where *A* and *B* are the **join attributes.**

We illustrate four of the most common techniques for performing such a join, using the following sample operations:

OP6: $\text{EMPLOYEE} \bowtie_{\text{Dno=Dnumber}} \text{DEPARTMENT}$

OP7: $\text{DEPARTMENT} \bowtie_{\text{Mgr\_ssn=Ssn}} \text{EMPLOYEE}$

a) J1—Nested-loop join (or nested-block join). This is the default (brute force) algorithm, as it does not require any special access paths on either file in the join. For each record *t* in *R* (outer loop), retrieve every record *s* from *S* (inner loop) and test whether the two records satisfy the join condition $t[A] = s[B]$.

b) J2—Single-loop join (using an access structure to retrieve the matching records). If an index (or hash key) exists for one of the two join attributes— say, attribute *B* of file *S*—retrieve each record *t* in *R* (loop over file *R*), and then use the access structure (such as an index or a hash key) to retrieve directly all matching records *s* from *S* that satisfy $s[B] = t[A]$.

c) J3—Sort-merge join. If the records of *R* and *S* are *physically sorted* (ordered) by value of the join attributes *A* and *B*, respectively, we can implement the join in the most efficient way possible.

Both files are scanned concurrently in order of the join attributes, matching the records that have the same values for *A* and *B*. If the files are not sorted, they may be sorted first by using external sorting.

In this method, pairs of file blocks are copied into memory buffers in order and the records of each file are scanned only once each for matching with the other file.

Sort and merge join

(a)

```
sort the tuples in R on attribute A;                (* assume R has n tuples (records) *)
sort the tuples in S on attribute B;                (* assume S has m tuples (records) *)
set i ← 1, j ← 1;
while (i ≤ n) and (j ≤ m)
do {  if R(i)[A] > S(j)[B]
        then  set j ← j + 1
      elseif R(i)[A] < S(j)[B]
        then  set i ← i + 1
      else  {  (* R(i)[A] = S(j)[B], so we output a matched tuple *)
               output the combined tuple <R(i), S(j)> to T;

               (* output other tuples that match R(i), if any *)
               set l ← j + 1;
               while (l ≤ m) and (R(i)[A] = S(l)[B])
               do {  output the combined tuple <R(i), S(l)> to T;
                     set l ← l + 1
               }

            (* output other tuples that match S(j), if any *)
            set k ← i + 1;
            while (k ≤ n) and (R(k)[A] = S(j)[B])
            do {  output the combined tuple <R(k), S(j)> to T;
                  set k ← k + 1
            }
            set i ← k, j ← l
      }
}
```

d) J4—Partition-hash join. The partitioning of each file is done using the same hashing function *h* on the join attribute *A* of *R* and *B* of *S*.

First, a single pass through the file with fewer records (say,*R*) hashes its records to the various partitions of R; this is called the **partitioning phase**, since the records of *R* are partitioned into the hash buckets.

In the simplest case, we assume that the smaller file can fit entirely in main memory after it is partitioned.

In the second phase, called the **probing phase**, a single pass through the other file (*S*) then hashes each of its records using the same hash function *h*(*B*) to *probe* the appropriate bucket, and that record is combined with all matching records from *R* in that bucket.

The buffer space available has an important effect on some of the join algorithms.

Another factor that affects the performance of a join, particularly the single-loop method J2, is the fraction of records in one file that will be joined with records in the other file. We call this the **join selection factor**11 of a file with respect to an equijoin condition with another file.

5) Implementation Algorithms for PROJECT and Set Operations

A PROJECT operation π<attribute list>(*R*) is straightforward to implement if <attribute list> includes a key of relation *R*, because in this case the result of the operation will have the same number of tuples as *R*, but with only the values for the attributes in<attribute list> in each tuple. If <attribute list> does not include a key of *R*, *duplicate tuples must be eliminated*. This can be done by sorting the result of the operation and then eliminating duplicate tuples, which appear consecutively after sorting.

(b) create a tuple *t*[<attribute list>] in T' for each tuple *t* in R;
(* T' contains the projection results *before* duplicate elimination *)
if <attribute list> includes a key of *R*
then $T \leftarrow T'$
else { sort the tuples in T';
set $i \leftarrow 1$, $j \leftarrow 2$;
while $i \leq n$
do { output the tuple $T'[i]$ to T;
while $T'[i] = T'[j]$ and $j \leq n$ do $j \leftarrow j + 1$; (* eliminate duplicates *)
$i \leftarrow j$; $j \leftarrow i + 1$
}
}
(* T contains the projection result after duplicate elimination *)

Set operations—UNION, INTERSECTION, SET DIFFERENCE, and CARTESIAN PRODUCT—are sometimes expensive to implement. In particular, the CARTESIAN PRODUCT operation $R \times S$ is quite expensive because its result includes a record for each combination of records from R and S. Also, each record in the result includes all attributes of R and S. If R has n records and j attributes, and S has m records and k attributes, the result relation for $R \times S$ will have $n * m$ records and each record will have $j + k$ attributes. Hence, it is important to avoid the CARTESIAN PRODUCT operation and to substitute other operations such as join during query optimization.

(c) sort the tuples in R and S using the same unique sort attributes;
set $i \leftarrow 1$, $j \leftarrow 1$;
while $(i \leq n)$ and $(j \leq m)$
do { if $R(i) > S(j)$
then { output $S(j)$ to T;
set $j \leftarrow j + 1$
}
elseif $R(i) < S(j)$
then { output $R(i)$ to T;
set $i \leftarrow i + 1$
}
else set $j \leftarrow j + 1$ (* $R(i)=S(j)$, so we skip one of the duplicate tuples *)
}
if $(i \leq n)$ then add tuples $R(i)$ to $R(n)$ to T;
if $(j \leq m)$ then add tuples $S(j)$ to $S(m)$ to T;

(d) sort the tuples in R and S using the same unique sort attributes;
set $i \leftarrow 1, j \leftarrow 1$;
while $(i \leq n)$ and $(j \leq m)$
do { if $R(i) > S(j)$
then set $j \leftarrow j + 1$
elseif $R(i) < S(j)$
then set $i \leftarrow i + 1$
else { output $R(j)$ to T; (* $R(i)$=$S(j)$, so we output the tuple *)
set $i \leftarrow i + 1, j \leftarrow j + 1$
}
}

```
(e)  sort the tuples in R and S using the same unique sort attributes;
     set i ← 1, j ← 1;
     while (i ≤ n) and (j ≤ m)
     do {  if R(i) > S(j)
              then  set j ← j + 1
           elseif R(i) < S(j)
              then  {  output R(i) to T;        (* R(i) has no matching S(j), so output R(i) *)
                       set i ← i + 1
                    }
           else  set i ← i + 1, j ← j + 1
     }
     if (i ≤ n) then add tuples R(i) to R(n) to T;
```

Hashing can also be used to implement UNION, INTERSECTION, and SET DIFFERENCE

COMBINATION OF OPERATIONS USING PIPELINING

- A query specified in SQL will typically be translated into a relational algebra expression that is a sequence of relational operations.
- If we execute a single operation at a time, we must generate temporary files on disk to hold the results of these temporary operations, creating excessive overhead
- To reduce the number of temporary files, it is common to generate query execution code that corresponds to algorithms for combinations of operations in a query.
- Heuristic relational algebra optimization can group operations together for execution. This is called pipelining or stream-based processing.

- Query execution code is created dynamically to implement multiple operations.
- The generated code for producing the query combines several algorithms that correspond to individual operations.
- As the result tuples from one operation are produced, they are provided as input for subsequent operations and work as a stream or pipeline.

USING HEURISTICS IN QUERY OPTIMIZATION

- Some optimization techniques apply heuristic rules to modify the internal representation of a query—which is usually in the form of a query tree or a query graph data structure—to improve its expected performance.
- The scanner and parser of an SQL query first generate a data structure that corresponds to an initial query representation, which is then optimized according to heuristic rules.
- This leads to an optimized query representation, which corresponds to the query execution strategy.
- Following that, a query execution plan is generated to execute groups of operations based on the access paths available on the files involved in the query

QUERY TREES AND QUERY GRAPHS

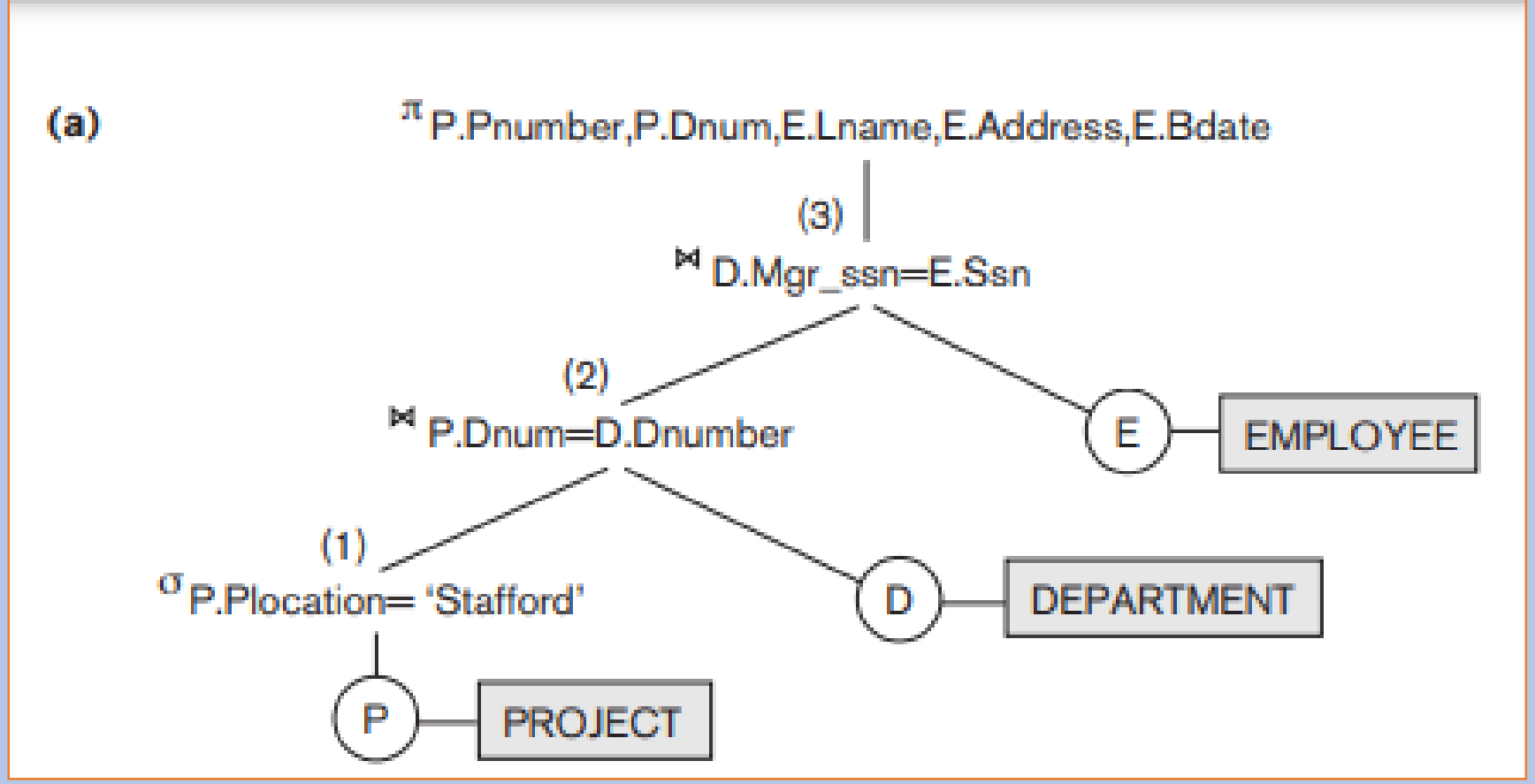
- A query tree is a tree data structure that corresponds to a relational algebra expression.
- It represents the input relations of the query as leaf nodes of the tree, and represents the relational algebra operations as internal nodes
- An execution of the query tree consists of executing an internal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

Figure 19.4a shows a query tree (the same as shown in Figure 6.9) for query Q2 in Chapters 4 to 6: For every project located in 'Stafford', retrieve the project number, the controlling department number, and the department manager's last name, address, and birthdate. This query is specified on the COMPANY relational schema in Figure 3.5 and corresponds to the following relational algebra expression:

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

This corresponds to the following SQL query:

```
Q2:  SELECT  P.Pnumber, P.Dnum, E.Lname, E.Address, E.Bdate
     FROM    PROJECT AS P, DEPARTMENT AS D, EMPLOYEE AS E
     WHERE   P.Dnum=D.Dnumber AND D.Mgr_ssn=E.Ssn AND
             P.Plocation= 'Stafford';
```

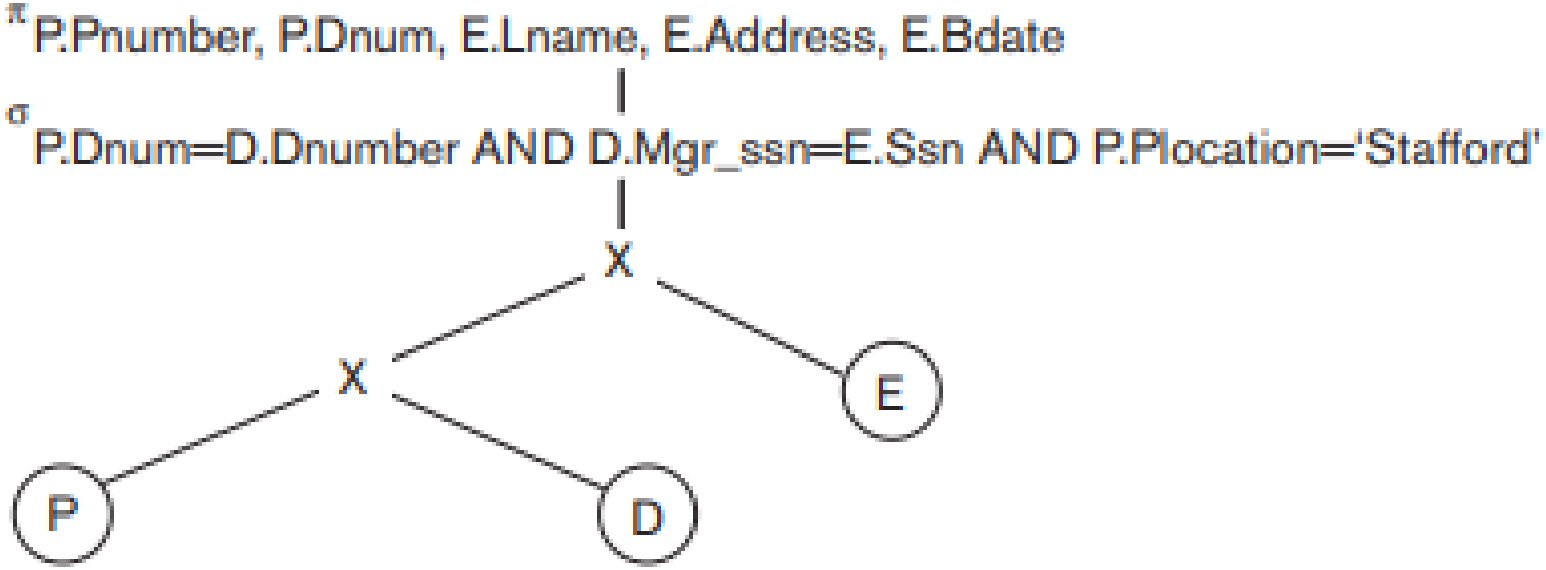


➢When this query tree is executed, the node marked (1must begin execution before node (2) because some resulting tuples of operation (1) must be available before we can begin executing operation (2). Similarly, node (2) must begin executing and producing results before node (3) can start execution, and so on. The query tree represents a specific order of operations for executing a query

This corresponds to the following SQL query:

```
Q2:  SELECT  P.Pnumber, P.Dnum, E.Lname, E.Address, E.Bdate
     FROM    PROJECT AS P, DEPARTMENT AS D, EMPLOYEE AS E
     WHERE   P.Dnum=D.Dnumber AND D.Mgr_ssn=E.Ssn AND
             P.Plocation= 'Stafford';
```

(b)



- A more neutral data structure for representation of a query is the **query graph**.
- **Relations i**n the query are represented by relation nodes, which **are displayed as single circles.**
- **Constant values**, typically from the query selection conditions, are represented by constant nodes, which are displayed as **double circles or ovals.**
- **Selection and join conditions** are represented by the graph edges.
- Finally, the **attributes to be retrieved** from each relation are **displayed in square brackets** above each relation.

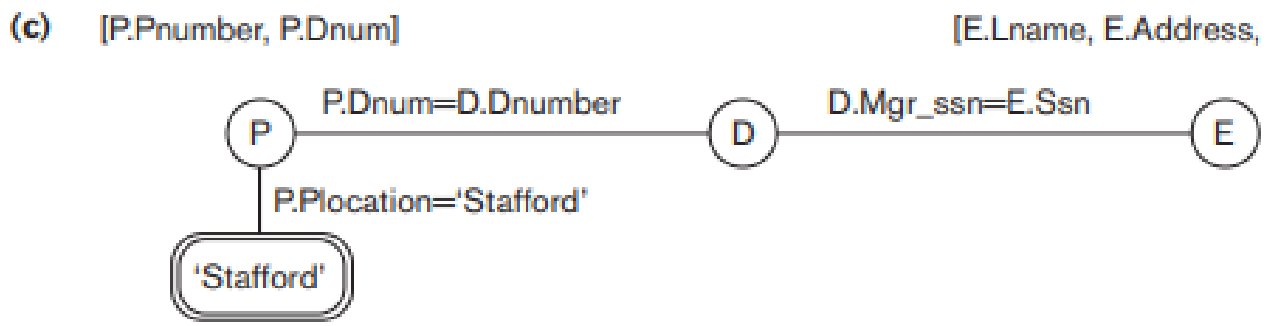


Figure 19.4
Two query trees for the query Q2. (a) Query tree corresponding to the relational algebra expression for Q2. (b) Initial (canonical) query tree for SQL query Q2. (c) Query graph for Q2.

- The query graph representation does not indicate an order on which operations to perform first.
- There is only a single graph corresponding to each query.
- Query trees are preferable w.r.t query graph because, in practice, the query optimizer needs to show the order of operations for query execution, which is not possible in query graphs

HEURISTIC OPTIMIZATION OF QUERY TREES

- Same query may have different equivalent relational algebraic expressions--- hence many different query trees.
- The query parser will typically generate a standard initial query tree to correspond to an SQL query, without doing any optimization.
- The heuristic query optimizer will transform this initial query tree into an equivalent final query tree that is efficient to execute.
- The optimizer must include rules for equivalence among relational algebra expressions that can be applied to transform the initial tree into the final, optimized query tree.

INFORMAL EXAMPLE OF HEURISTIC OPTIMIZATION

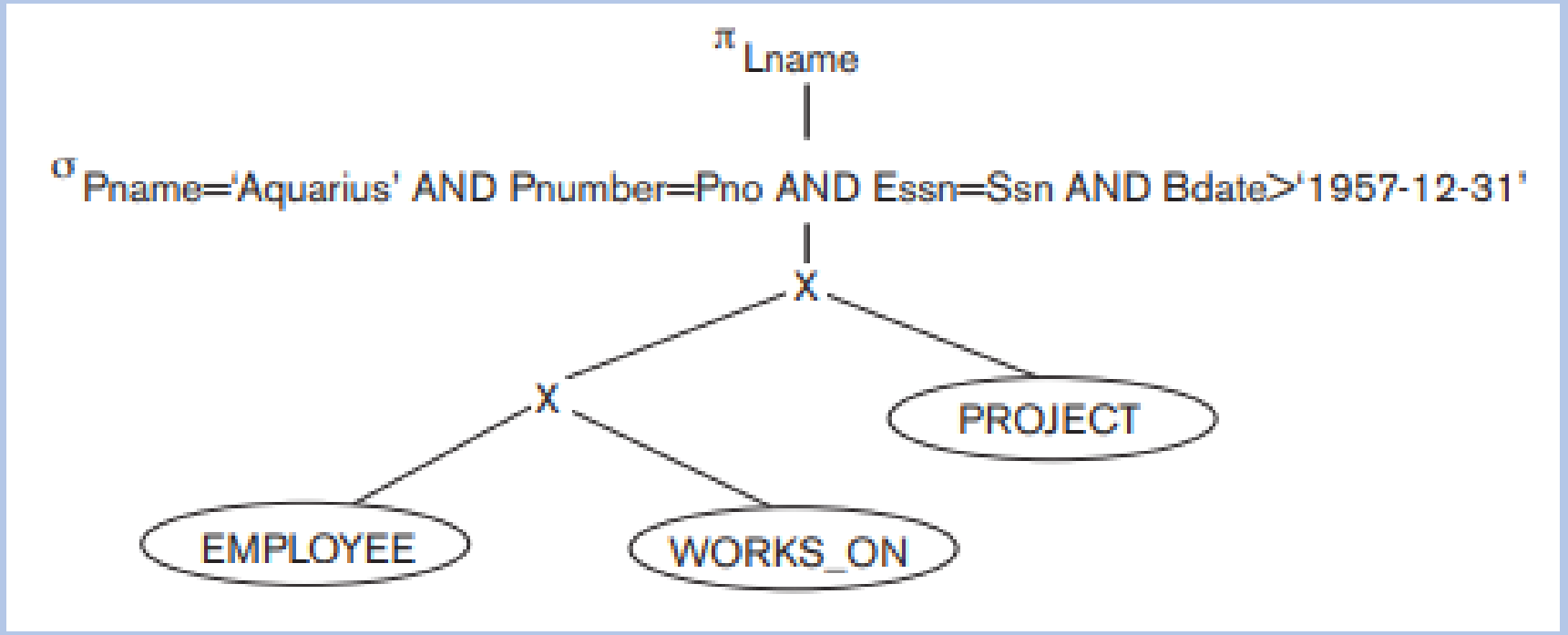
Query: Find the last names of employees born after 1957 who work on a project named ‘Aquarius’.

SQL:

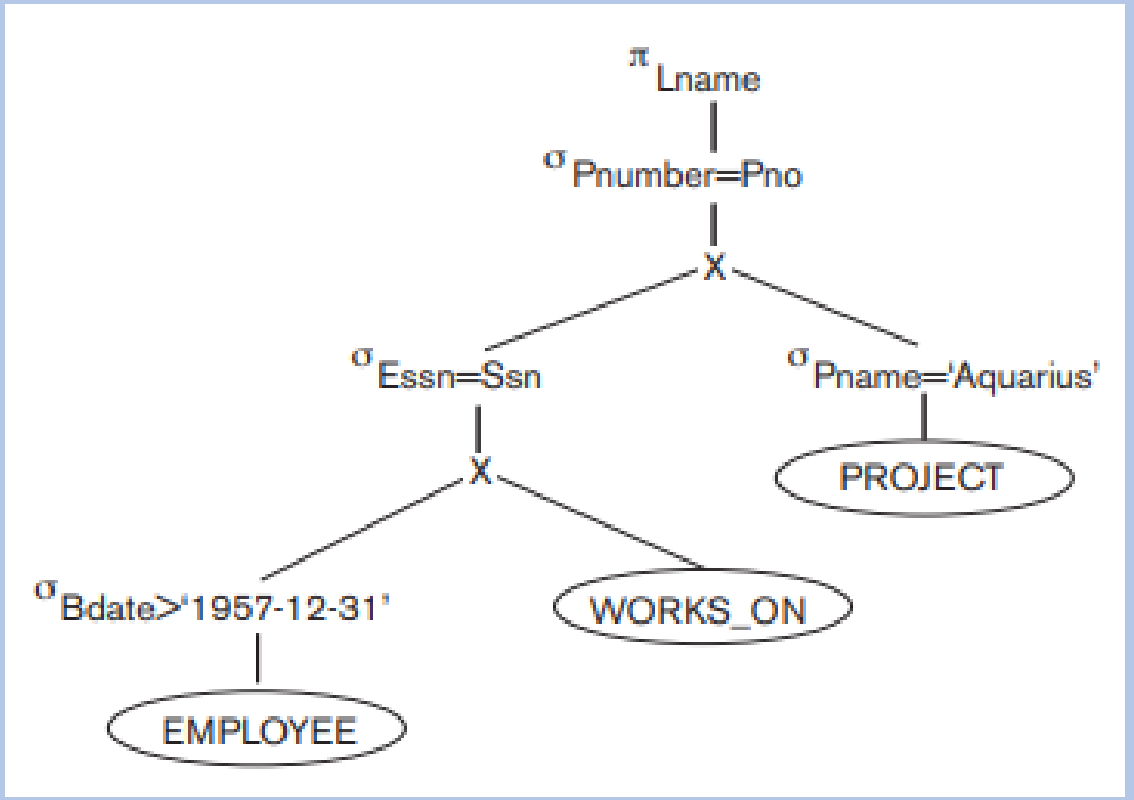
```
SELECT LNAME
        FROM    EMPLOYEE, WORKS_ON, PROJECT
        WHERE   PNAME = ‘AQUARIUS’ AND PNMUBER=PNO
                AND ESSN=SSN AND BDATE > ‘1957-12-31’;
```

(a) Initial (canonical) query tree for SQL query Q.

```
SELECT LNAME
FROM       EMPLOYEE, WORKS_ON, PROJECT
WHERE  PNAME = 'AQUARIUS' AND PNMUBER=PNO
AND ESSN=SSN AND BDATE > '1957-12-31';
```

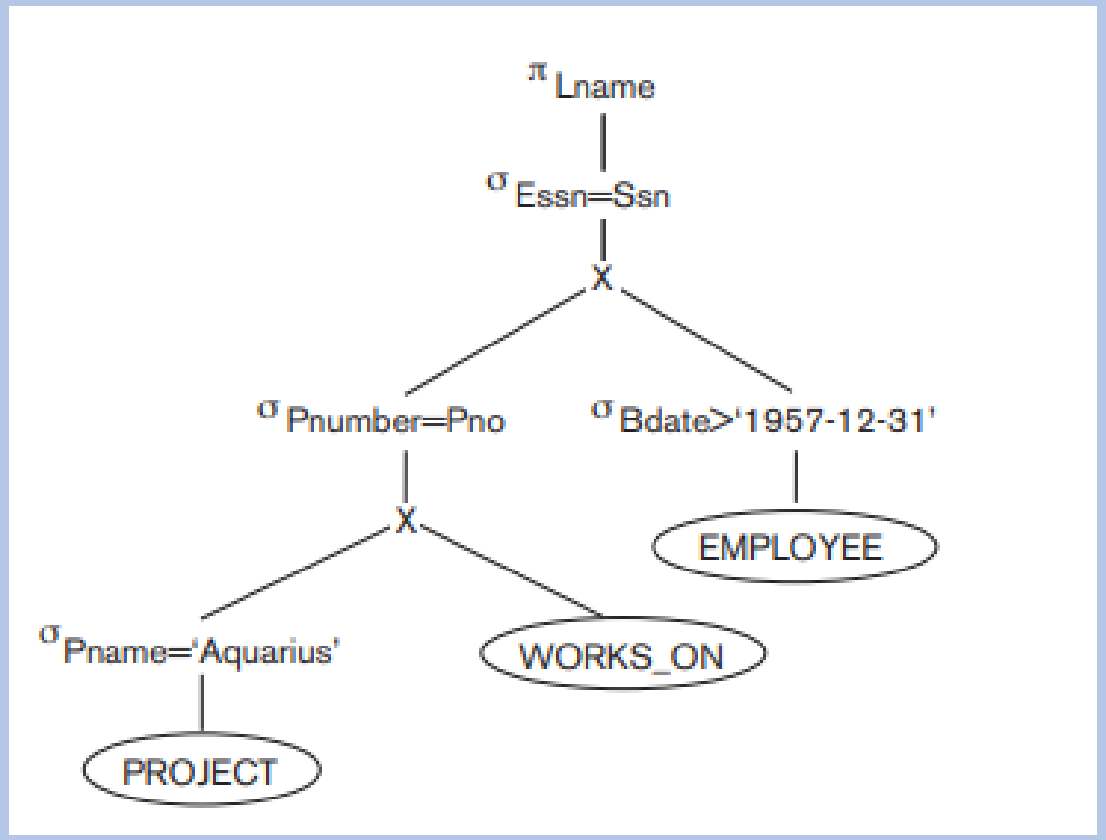


(b) Moving SELECT operations down the query tree



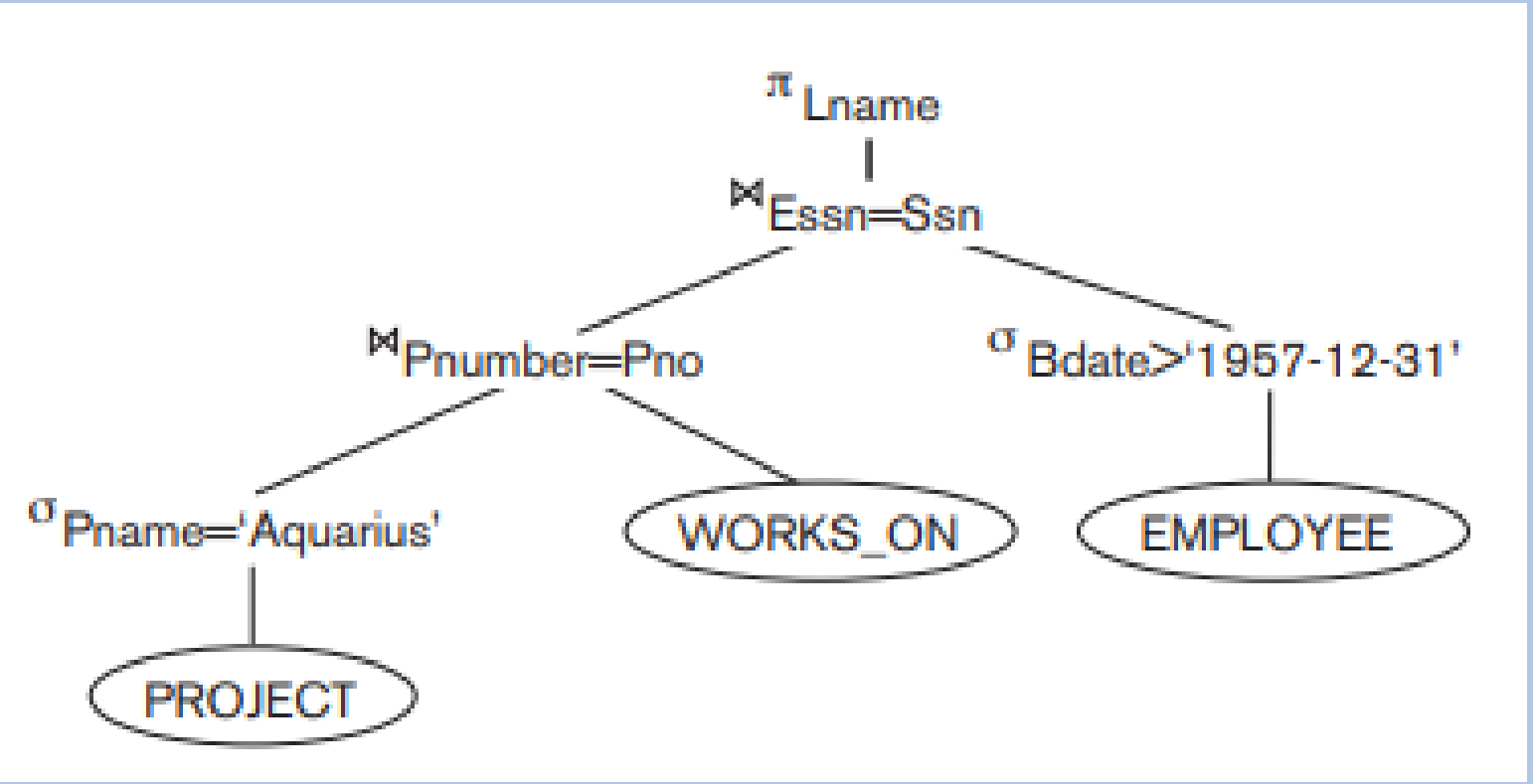
- Improved query tree that first applies the SELECT operations to reduce the number of tuples that appear in the CARTESIAN PRODUCT.

(c) Applying the more restrictive SELECT operation first.



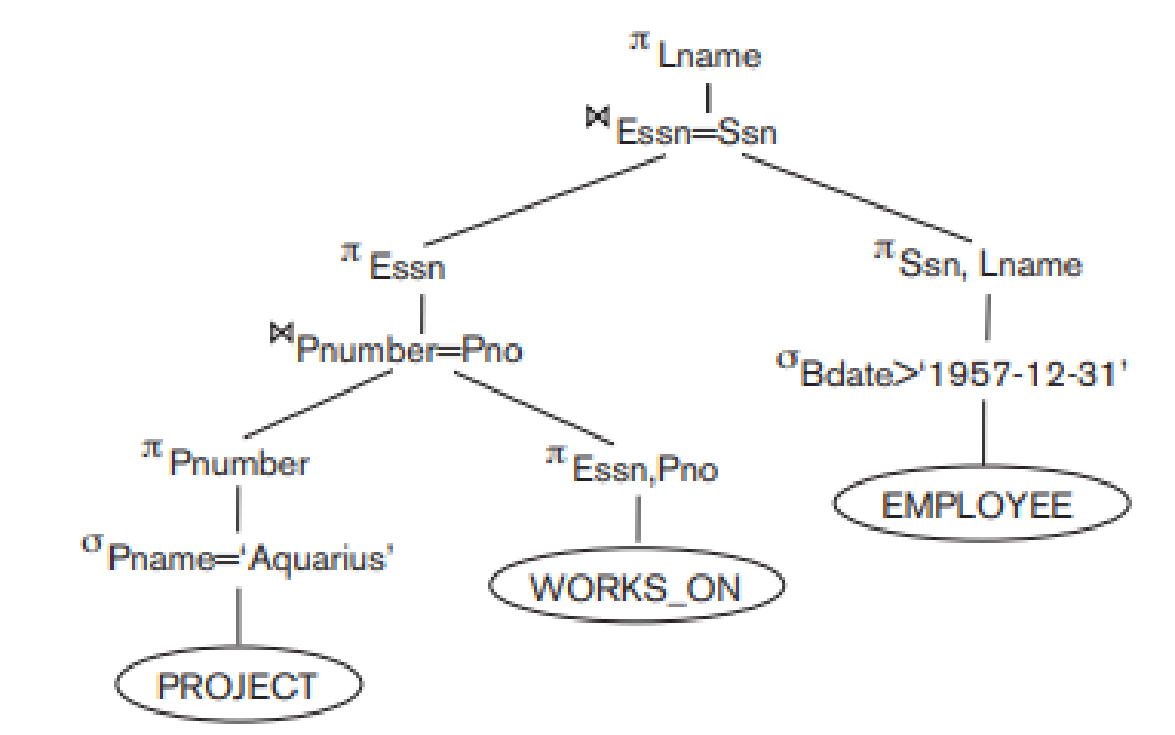
A further improvement is achieved by switching the positions of the EMPLOYEE and PROJECT relations in the tree). This uses the information that Pnumber is a key attribute of the PROJECT relation, and hence the SELECT operation on the PROJECT relation will retrieve a single record only.

(d) Replacing CARTESIAN PRODUCT and SELECT with JOIN operations



Improve the query tree by replacing any CARTESIAN PRODUCT operation that is followed by a join condition with a JOIN operation.

(e) Moving PROJECT operations down the query tree.



- 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.
- This reduces the attributes (columns) of the intermediate relations, whereas the SELECT operations reduce the number of tuples (records).

General Transformation Rules for Relational Algebra Operations.

- 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.
- 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
- 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.

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:

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

5. **Commutativity of ⋈ (and ×).** The join operation is commutative, as is the × 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 ⋈ (or ×).** 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 $(c_1 \text{ AND } c_2)$, 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 ⋈ is replaced by a × operation.

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

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

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 $A_{n+1}, ..., A_{n+k}$ of R and $B_{m+1}, ..., B_{m+p}$ of S are involved in the join condition c but are not in the projection list L, the operations commute as follows:

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

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

8. **Commutativity of set operations.** The set operations ∪ and ∩ are commutative but − is not.

9. **Associativity of ⋈, ×, ∪, and ∩.** These four operations are individually associative; that is, if θ stands for any one of these four operations (throughout the expression), we have:

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

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 $\cup$.**

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

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

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

There are other possible transformations. For example, a selection or join condition c can be converted into an equivalent condition by using the following standard rules from Boolean algebra (DeMorgan's laws):

$$\textbf{NOT } (c_1 \textbf{ AND } c_2) \equiv (\textbf{NOT } c_1) \textbf{ OR } (\textbf{NOT } c_2)$$
$$\textbf{NOT } (c_1 \textbf{ OR } c_2) \equiv (\textbf{NOT } c_1) \textbf{ AND } (\textbf{NOT } c_2)$$

Outline of a Heuristic Algebraic Optimization Algorithm.

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).

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
2. **Using Rules 2, 4, 6, and 10 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.

3. Using **Rules 5 and 9** concerning **commutativity** and associativity o**f binary operations, rearrange the leaf nodes** of the tree using the following criteria.

   i) First, position the leaf node relations with the most restrictive SELECT operations so they are executed first in the query tree representation.

   most restrictive SELECT can mean

   a) either the ones that produce a relation with the fewest tuples or with the smallest absolute size.

   b)or one with the smallest selectivity; this is more practical because estimates of selectivities are often available in the DBMS catalog.

   ii)Second, make sure that the ordering of leaf nodes does not cause CARTESIAN PRODUCT operations.

4. 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.

5. **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.
6. Identify subtrees that represent groups of operations that can be executed by a single algorithm.

Summary of Heuristics for Algebraic Optimization.

- 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.

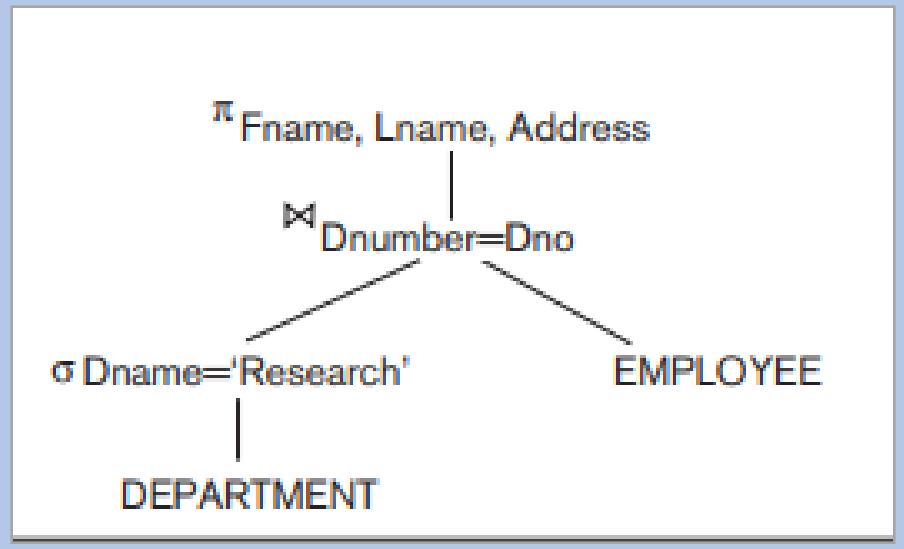
Converting Query Trees into Query Execution Plans

▪**An execution plan** for a **relational algebra expression represented as a query tree includes information** about

1. **The access methods available for each relation.**
2. **The algorithms to be used in computing the relational operators represented in the tree.**

An execution plan for a relational algebra query consists of a combination of the relational algebra query tree and information about the access methods to be used for each relation as well as the methods to be used in computing the relational operators stored in the tree.

$$\pi_{Fname,\ Lname,\ Address}(\sigma_{Dname='Research'}(DEPARTMENT) \bowtie_{Dnumber=Dno} EMPLOYEE)$$



- To convert this into an execution plan, the optimizer might choose an index search for the SELECT operation on DEPARTMENT.
- A single-loop join algorithm that loops over the records in the result of the SELECT operation on DEPARTMENT for the join operation (assuming an index exists on the Dno attribute of EMPLOYEE).
- a scan of the JOIN result for input to the PROJECT operator.

Executing the query may specify

1. a materialized
2. or a pipelined evaluation, although in general a pipelined evaluation is preferred whenever feasible

With materialized evaluation, the **result of an operation is stored as a temporary relation** (that is, the result is physically materialized).

For instance, the JOIN operation can be computed and the entire result stored as a temporary relation, which is then read as input by the algorithm that computes the PROJECT operation, which would produce the query result table.

with pipelined evaluation, as the resulting tuples of an operation are produced, they are forwarded directly to the next operation in the query sequence.

For example, as the selected tuples from DEPARTMENT are produced by the SELECT operation, they are placed in a buffer; the JOIN operation algorithm would then consume the tuples from the buffer, and those tuples that result from the JOIN operation are pipelined to the projection operation algorithm.

The advantage of pipelining is the cost savings in not having to write the intermediate results to disk and not having to read them back for the next operation.

COST ESTIMATION IN QUERY OPTIMIZATION

•A query optimizer does not depend solely on heuristic rules.

• It also **estimates and compares** the **costs of executing a query using different execution strategies and algorithms.**

•It then **choose**s the strategy with **the lowest cost estimate**.

•So **accurate cost estimates are required.**

•The **optimizer must limit the number of execution strategies** to be considered; otherwise, too much time will be spent making cost estimates.

• Hence, this approach is **more suitable** for **compiled queries** where the **optimization is done at compile time** and the **resulting execution strategy code is stored and executed directly at runtime.**

• This approach is generally referred to as **cost-based query optimization.**

• 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 Components for Query Execution

1. **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.

2. **Disk storage cost.** This is the cost **of storing on disk any intermediate files** that are generated by an execution strategy for the query.

3. **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.

4. **Memory usage cost**. This is the cost pertaining to the number of main memory buffers needed during query execution.
5. **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

Catalog Information Used in Cost Functions

- **First**, we must know the **size of each file.**
- For a file whose records are all of the same type, **the number of records (tuples) (r), the (average) record size (R), and the number of file blocks (b)** (or close estimates of them) are needed.
- The **blocking factor (bfr)** for the file may also be needed.
- We must also keep track of the **primary file organization for each file**. The primary file organization records may be unordered, ordered by an attribute with or without a primary or clustering index, or hashed (static hashing or one of the dynamic hashing methods) on a key attribute.
- **Information is also kept on all primary, secondary, or clustering indexes** and their indexing attributes. The number of levels (x) of each multilevel index (primary, secondary, or clustering) is needed.
- In some cost functions the **number of first-level index blocks (bI1) is needed**.

- Another important parameter is the **number of distinct values (d)** of an attribute and **the attribute selectivity (sl), which is the fraction of records satisfying an equality condition on the attribute**.
- This allows **estimation of the selection cardinality (s = sl*r) of an attribute**, which is the **average number of records that will satisfy an equality selection** condition on that attribute.
- For a key attribute, d = r, sl = 1/r and s = 1.
- For a nonkey attribute, by making an assumption that the d distinct values are uniformly distributed among the records, we estimate sl = (1/d) and so s = (r/d).

For a nonkey attribute with d distinct values, it is often the case that the records are not uniformly distributed among these values.

In such cases, the optimizer can store a histogram that reflects the distribution.

Catalog Information Used in Cost Functions

- Information about the size of a file
  - **number of records (tuples) (r),**
  - **record size (R),**
  - **number of blocks (b)**
  - **blocking factor (bfr)**
- Information about indexes and indexing attributes of a file
  - **Number of levels (x)** of each multilevel index
  - **Number of first-level index blocks (b_{I1})**
  - **Number of distinct values (d)** of an attribute
  - **Selectivity (sl)** of an attribute
  - **Selection cardinality (s)** of an attribute. (s = sl * r)

Examples of Cost Functions for SELECT

- **S1. Linear search (brute force) approach**

  $C_{S1a} = b$;

  For an equality condition on a key, $C_{S1a} = (b/2)$ if the record is found; otherwise $C_{S1a} = b$.

- **S2. Binary search**:

  $C_{S2} = \log_2 b + \lceil (s/bfr) \rceil - 1$

  For an equality condition on a unique (key) attribute,

  $C_{S2} = \log_2 b$

- **S3. Using a primary index (S3a) or hash key (S3b) to retrieve a single record**

  $C_{S3a} = x + 1$; $C_{S3b} = 1$ for static or linear hashing;

  $C_{S3b} = 1$ for extendible hashing;

- **S4. Using an ordering index to retrieve multiple records:**

  For the comparison condition on a key field with an ordering index, $C_{S4} = x + (b/2)$

- **S5. Using a clustering index to retrieve multiple records**:

  $C_{S5} = x + \lceil (s/bfr) \rceil$

- **S6. Using a secondary (B^+-tree) index:**

  For an equality comparison, $C_{S6a} = x + s$;

  For an comparison condition such as >, <, >=, or <=,

  $C_{S6a} = x + (b_{I1}/2) + (r/2)$

- **S7. Conjunctive selection:**

  Use either S1 or one of the methods S2 to S6 to solve.

  For the latter case, use one condition to retrieve the records and then check in the memory buffer whether each retrieved record satisfies the remaining conditions in the conjunction.

- **S8. Conjunctive selection using a composite index**:

  Same as S3a, S5 or S6a, depending on the type of index.

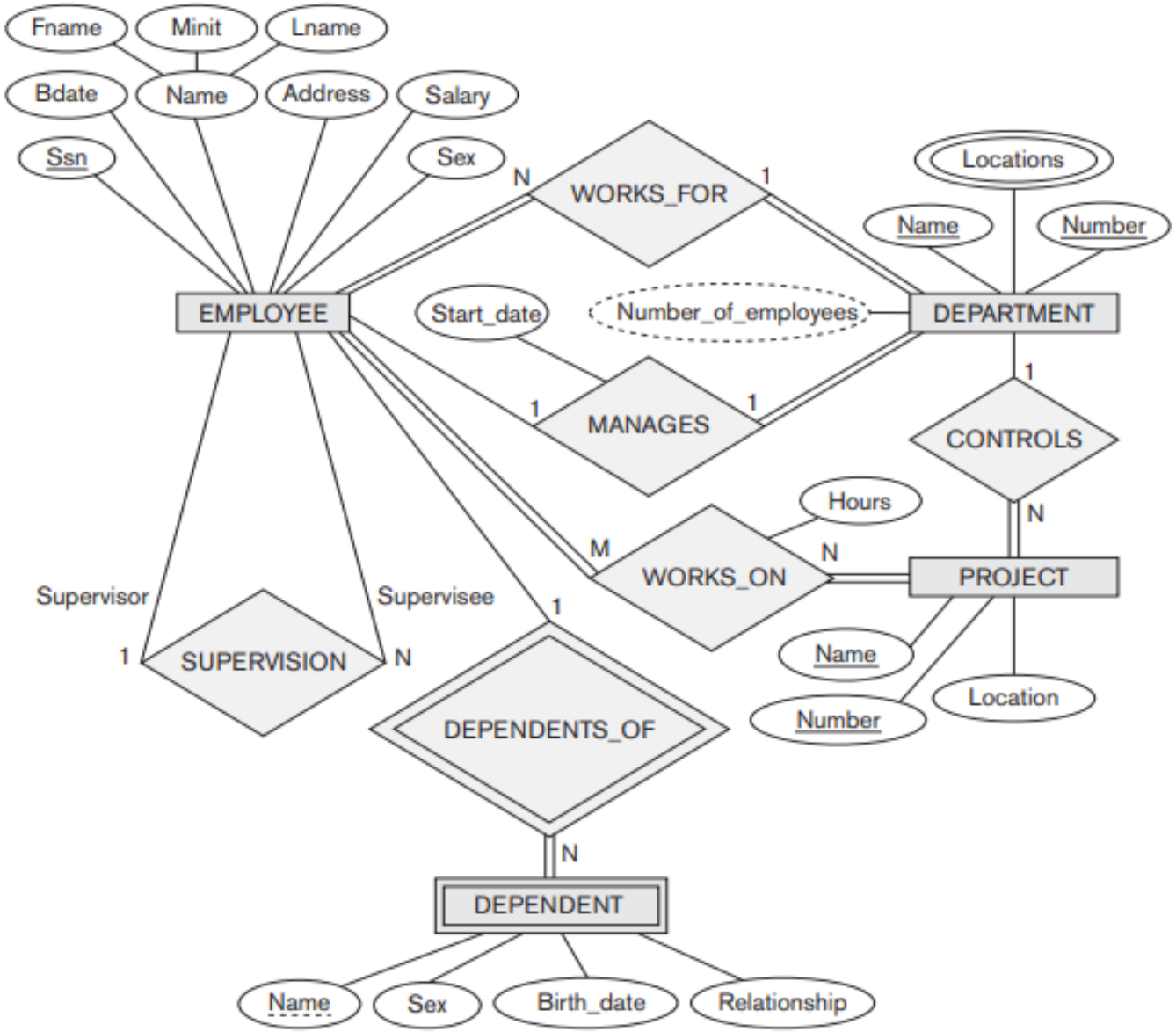


Figure 7.2
An ER schema diagram for the COMPANY database. The diagrammatic notation is introduced gradually throughout this chapter and is summarized in Figure 7.14.

EMPLOYEE

Fname	Minit	Lname	Ssn	Bdate	Address	Sex	Salary	Super_ssn	Dno

DEPARTMENT

Dname	Dnumber	Mgr_ssn	Mgr_start_date

DEPT_LOCATIONS

Dnumber	Dlocation

PROJECT

Pname	Pnumber	Plocation	Dnum

WORKS_ON

Essn	Pno	Hours

DEPENDENT

Essn	Dependent_name	Sex	Bdate	Relationship

Figure 3.5
Schema diagram for the COMPANY relational database schema.

Example of Using the Cost Functions

Suppose that the EMPLOYEE file has rE = 10,000 records stored in bE = 2000 disk blocks with blocking factor bfrE = 5 records/block and the following access paths:

1. A clustering index on Salary, with levels $x_{\text{Salary}} = 3$ and average selection cardinality $s_{\text{Salary}} = 20$. (This corresponds to a selectivity of $sl_{\text{Salary}} = 0.002$).
2. A secondary index on the key attribute Ssn, with $x_{\text{Ssn}} = 4$ ($s_{\text{Ssn}} = 1$, $sl_{\text{Ssn}} = 0.0001$).
3. A secondary index on the nonkey attribute Dno, with $x_{\text{Dno}} = 2$ and first-level index blocks $b_{I1\text{Dno}} = 4$. There are $d_{\text{Dno}} = 125$ distinct values for Dno, so the selectivity of Dno is $sl_{\text{Dno}} = (1/d_{\text{Dno}}) = 0.008$, and the selection cardinality is $s_{\text{Dno}} = (r_E * sl_{\text{Dno}}) = (r_E/d_{\text{Dno}}) = 80$.
4. A secondary index on Sex, with $x_{\text{Sex}} = 1$. There are $d_{\text{Sex}} = 2$ values for the Sex attribute, so the average selection cardinality is $s_{\text{Sex}} = (r_E/d_{\text{Sex}}) = 5000$. (Note that in this case, a histogram giving the percentage of male and female employees may be useful, unless they are approximately equal.)

We illustrate the use of cost functions with the following examples:

OP1: $\sigma_{Ssn='123456789'}(EMPLOYEE)$

OP2: $\sigma_{Dno>5}(EMPLOYEE)$

OP3: $\sigma_{Dno=5}(EMPLOYEE)$

1. **The cost of the brute force** (linear search or file scan) option S1 will be estimated as CS1a = bE = 2000 (for a selection on a nonkey attribute) or CS1b = (bE/2) = 1000 (average cost for a selection on a key attribute).
2. **For OP1 we** can use either **method S1 or method S6a**; the cost estimate for S6a is CS6a = xSsn + 1 = 4 + 1 = 5, and it is chosen over method S1, whose average cost is CS1b = 1000.
3. **For OP2** we can use either **method S1 (with estimated cost CS1a = 2000) or method S6b (**with estimated cost CS6b = xDno + (bI1Dno/2) + (rE /2) = 2 + (4/2) + (10,000/2) = 5004), so we choose the linear search approach for OP2.
4. **For OP3** we can use either method S1 (with estimated cost CS1a = 2000) or method S6a (with estimated cost CS6a = xDno + sDno = 2 + 80 = 82), so we choose method S6a.

OP4: $\sigma_{Dno=5\ AND\ SALARY>30000\ AND\ Sex='F'}(EMPLOYEE)$

4. **Consider OP4**, which has a conjunctive selection condition. We need to estimate the cost of using any one of the three components of the selection condition to retrieve the records, plus the linear search approach.

- The latter gives cost estimate CS1a = 2000
- Using the condition (Dno = 5) first gives the cost estimate CS6a = 82.
- Using the condition (Salary > 30,000) first gives a cost estimate CS4 = xSalary + (bE/2) = 3 + (2000/2) = 1003.
- Using the condition (Sex = 'F') first gives a cost estimate CS6a = xSex + sSex = 1 + 5000 = 5001

The optimizer would then choose method S6a on the secondary index on Dno because it has the lowest cost estimate. The condition (Dno = 5) is used to retrieve the records, and the remaining part of the conjunctive condition (Salary > 30,000 AND Sex = 'F') is checked for each selected record after it is retrieved into memory

Examples of Cost Functions for JOIN

- **Join selectivity (js)**

  $js = |(R \bowtie_C S)| / |R \times S| = |(R \bowtie_C S)| / (|R| * |S|)$

  If condition C does not exist, js = 1;

  If no tuples from the relations satisfy condition C, js = 0;

  Usually, **0 <= js <= 1**;

- **Size of the result file after join operation**

  $|(R \bowtie_C S)| = js * |R| * |S|$

For a join where the condition c is an equality comparison R.A = S.B, we get the following two special cases:

1. If A is a key of R, then $|(R \bowtie_c S)| \leq |S|$, so $js \leq (1/|R|)$. This is because each record in file S will be joined with at most one record in file R, since A is a key of R. A special case of this condition is when attribute B is a *foreign key* of S that references the *primary key* A of R. In addition, if the foreign key B has the NOT NULL constraint, then $js = (1/|R|)$, and the result file of the join will contain $|S|$ records.
2. If B is a key of S, then $|(R \bowtie_c S)| \leq |R|$, so $js \leq (1/|S|)$.

Assume that R has bR blocks and that S has bS blocks:

- **J1.** Nested-loop join:

  $C_{J1} = b_R + (b_R * b_S) + ((js * |R| * |S|)/bfr_{RS})$

  (Use R for outer loop)

  If nB main memory buffers are available to perform the join, the cost formula becomes:

  $CJ1 = bR + (\lceil bR/(nB - 2) \rceil * bS) + ((js * |R| * |S|)/bfrRS)$

- **J2. Single-loop join** (using an access structure to retrieve the matching record(s))

  If an index exists for the join attribute B of S with index levels x_B, we can retrieve each record s in R and then use the index to retrieve all the matching records t from S that satisfy t[B] = s[A].

  The cost depends on the type of index.

- **J2. Single-loop join (cont.)**

  For a secondary index,

  $C_{J2a} = b_R + (|R| * (x_B + s_B)) + ((js * |R| * |S|)/bfr_{RS})$;

  For a clustering index,

  $C_{J2b} = b_R + (|R| * (x_B + (s_B/bfr_B))) + ((js * |R| * |S|)/bfr_{RS})$;

  For a primary index,

  $C_{J2c} = b_R + (|R| * (x_B + 1)) + ((js * |R| * |S|)/bfr_{RS})$;

  If a hash key exists for one of the two join attributes — B of S

  $C_{J2d} = b_R + (|R| * h) + ((js * |R| * |S|)/bfr_{RS})$;

- **J3. Sort-merge join:**

  $C_{J3a} = C_S + b_R + b_S + ((js * |R| * |S|)/bfr_{RS})$; ($C_S$: Cost for sorting files)

Example of Using the Cost Functions. Suppose that we have the EMPLOYEE file described in the example in the previous section, and assume that the DEPARTMENT file in Figure 3.5 consists of $r_D = 125$ records stored in $b_D = 13$ disk blocks. Consider the following two join operations:

OP6: EMPLOYEE $\bowtie_{\text{Dno=Dnumber}}$ DEPARTMENT
OP7: DEPARTMENT $\bowtie_{\text{Mgr\_ssn=Ssn}}$ EMPLOYEE

Suppose that we have a primary index on Dnumber of DEPARTMENT with $x_{\text{Dnumber}} = 1$ level and a secondary index on Mgr_ssn of DEPARTMENT with selection cardinality $s_{\text{Mgr\_ssn}} = 1$ and levels $x_{\text{Mgr\_ssn}} = 2$. Assume that the join selectivity for OP6 is $js_{\text{OP6}} = (1/|\text{DEPARTMENT}|) = 1/125$ because Dnumber is a key of DEPARTMENT. Also assume that the blocking factor for the resulting join file is $bfr_{\text{ED}} = 4$ records per block. We can estimate the worst-case costs for the JOIN operation OP6 using the applicable methods J1 and J2 as follows:

1. Using method J1 with EMPLOYEE as outer loop:

$$C_{J1} = b_E + (b_E * b_D) + ((js_{OP6} * r_E * r_D)/bfr_{ED})$$
$$= 2000 + (2000 * 13) + (((1/125) * 10{,}000 * 125)/4) = 30{,}500$$

2. Using method J1 with DEPARTMENT as outer loop:

$$C_{J1} = b_D + (b_E * b_D) + ((js_{OP6} * r_E * r_D)/bfr_{ED})$$
$$= 13 + (13 * 2000) + (((1/125) * 10{,}000 * 125/4) = 28{,}513$$

3. Using method J2 with EMPLOYEE as outer loop:

$$C_{J2c} = b_E + (r_E * (x_{Dnumber} + 1)) + ((js_{OP6} * r_E * r_D)/bfr_{ED}$$
$$= 2000 + (10{,}000 * 2) + (((1/125) * 10{,}000 * 125/4) = 24{,}500$$

4. Using method J2 with DEPARTMENT as outer loop:

$$C_{J2a} = b_D + (r_D * (x_{Dno} + s_{Dno})) + ((js_{OP6} * r_E * r_D)/bfr_{ED})$$
$$= 13 + (125 * (2 + 80)) + (((1/125) * 10{,}000 * 125/4) = 12{,}763$$

Case 4 has the lowest cost estimate and will be chosen. Notice that in case 2 above, if 15 memory buffers (or more) were available for executing the join instead of just 3, 13 of them could be used to hold the entire DEPARTMENT relation (outer loop

relation) in memory, one could be used as buffer for the result, and one would be used to hold one block at a time of the EMPLOYEE file (inner loop file), and the cost for case 2 could be drastically reduced to just $b_E + b_D + ((\ js_{OP6} * r_E * r_D)/bfr_{ED})$ or 4,513, as discussed in Section 19.3.2. If some other number of main memory buffers was available, say $n_B = 10$, then the cost for case 2 would be calculated as follows, which would also give better performance than case 4:

$$\begin{aligned} C_{J1} &= b_D + (\lceil b_D/(n_B - 2) \rceil * b_E) + ((js * |R| * |S|)/bfr_{RS}) \\ &= 13 + (\lceil 13/8 \rceil * 2000) + (((1/125) * 10{,}000 * 125/4) = 28{,}513 \\ &= 13 + (2 * 2000) + 2500 = 6{,}513 \end{aligned}$$

As an exercise, the reader should perform a similar analysis for OP7.

Multiple Relation Queries and Join Ordering

- A query joining n relations will have n-1 join operations, and hence can have a large number of different join orders when we apply the algebraic transformation rules.
- Estimating the cost of every possible join tree for a query with a large number of joins will require a substantial amount of time by the query optimizer. Hence, some pruning of the possible query trees is needed.
- Current query optimizers typically limit the structure of a (join) query tree to that of left-deep (or right-deep) trees.
- **Left-deep tree**: a binary tree where the right child of each non-leaf node is always a base relation. The optimizer would choose the particular left-deep tree with the lowest estimated cost
  - Amenable to pipelining
  - Could utilize any access paths on the base relation (the right child) when executing the join.

Figure 19.7
Two left-deep (JOIN) query trees.