DUE an occurrence of its logical parent is inserted in the physical database (assume that BOOK_COPY is a root node of a physical tree).

Consider the following statements to insert in the database information to indicate that Cook has borrowed copy 3 of a book with *Call_No* 1235 (entitled *Anne of Green Gables* by Montgomery) from the Lynn branch:

```
BOOK_DUE.Call_No = '123';
BOOK_DUE.Copy_No = 3;
BOOK_DUE.Branch_Id = 'Lynn';
BOOK_DUE.Status = 'Lent';
BOOK_DUE.Due_Date = 12/28;
insert (BOOK_DUE)
    where (CLIENT.Client_No = '237');
```

The last statement will succeed if an occurrence of the BOOK_COPY exists, and in this case the BOOK_COPY.*Status* is updated to lent. If no occurrence of the record BOOK_COPY exists, then depending on the rules specified for BOOK_COPY, the operation will succeed or fail. In the former case, an occurrence for BOOK_COPY will be inserted in the database. The information for this occurrence is available in the record BOOK_DUE.

Deleting a CLIENT may or may not succeed depending on the rules specified for CLIENT and whether there are any volumes outstanding with the CLIENT. If the rule specified for CLIENT is physical and if the client has a number of books on loan, the attempt to delete a client will fail. If the rule specified is either logical or virtual, the occurrence is made inaccessible as a CLIENT record occurrence. However, it remains accessible via VIR_CLIENT.

Finally, modification of certain fields in the records are not allowed. For example, the *Call_No* and the *Client_No* fields, which are used to establish the logical parent/child record occurrence association, cannot be modified.

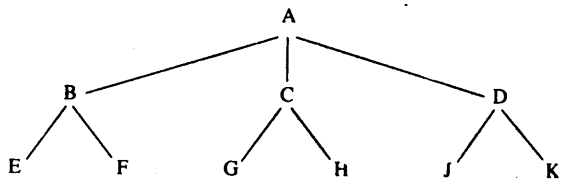
Replacement of the other fields of CLIENT can always be done. However, replacement of a field of BOOK_DUE could affect the logical parent BOOK_COPY and would succeed if the option specified for BOOK_COPY is virtual.

9.6 Implementation of the Hierarchical Database

Each occurrence of a hierarchical tree can be stored as a variable length physical record, the nodes of the hierarchy being stored in preorder. In addition, the stored record contains a prefix field. This field contains control information including pointers, flags, locks, and counters, which are used by the DBMS to allow concurrent usage and enforce data integrity.

A number of methods could be used to store the hierarchical database system. The storage of the hierarchical trees in the physical medium affects not only the performance of the system but also the operations that can be performed on the database. For example, if each occurrence of the hierarchical tree is stored as a variable length record on a magnetic tape like device, the DBMS will allow only sequential retrieval and insertion or modification may be disallowed or performed only by recreating the entire database with the insertion and modification. Storage of

Figure 9.16 Hierarchical definition tree.



the database on a direct access device allows an index structure to be supported for the root nodes and allows direct access to an occurrence of a hierarchical tree.

The storage of one occurrence of the hierarchical definition tree of Figure 9.16 using the variable length record approach is given in Figure 9.17.

The hierarchy can also be represented using pointers of either preorder hierarchical type or child/sibling type. In the hierarchical type of pointer, each record occurrence has a pointer that points to the next record in the preorder sequence. In the child/sibling scheme each record has two types of pointers. The **sibling** pointer points to its right sibling (or twin). The **child** pointer points to its leftmost child record occurrence. A record has one sibling pointer and as many child pointers as the number of child types associated with the node corresponding to the record. Figure 9.16 gives the hierarchical definition tree and the one occurrence of this hierarchical definition tree is given in Figures 9.18 and 9.19. In Figure 9.18 the preorder hierarchical pointers are shown, whereas in Figure 9.19 we present the same database using the child/sibling pointers.

Figure 9.17 Sequential storage of hierarchical database.

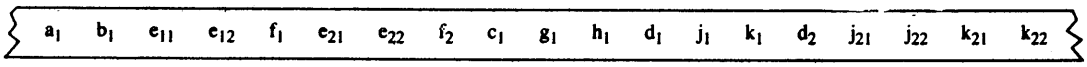


Figure 9.18 Preorder hierarchical pointers.

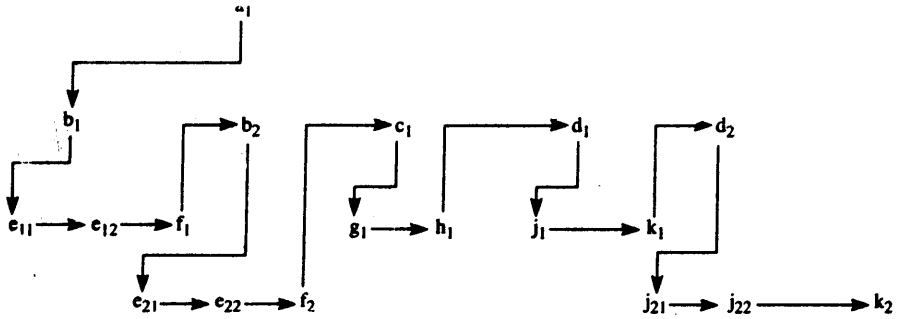
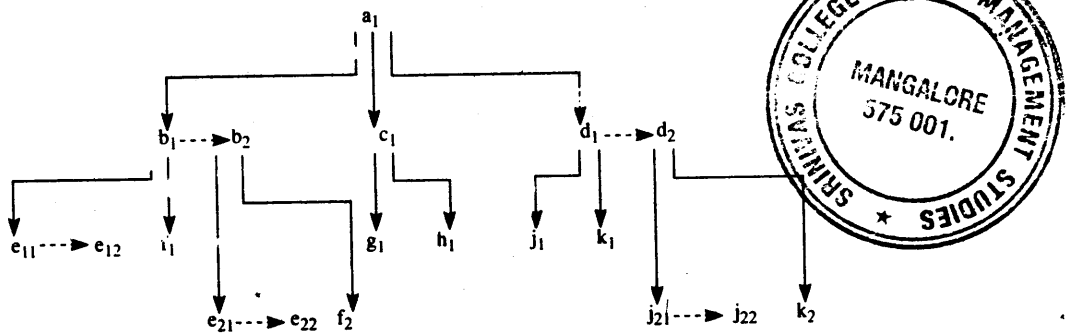


Figure 9.19 Child/sibling pointers.



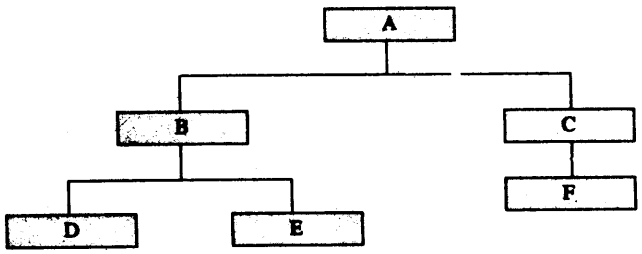
9.7 Additional Features of the Hierarchical DML

Consider the hierarchical definition tree of Figure 9.20. Access to a dependent record type is via a path beginning at the root node and after traversing through intermediate nodes, ending at the required record type. Such paths are called **hierarchical paths.** Access to record type E, in the hierarchical definition tree of Figure 9.20, requires a traversal through nodes A and B.

In addition to the data manipulation statement discussed earlier, the hierarchical data manipulation language needs a number of functions for better control of navigating through the database. This saves both processing and program development time.

One such feature is the use of control codes associated with the get statements. We will not give the exact syntax of these statements or describe them in detail, but we will highlight their usefulness. **Control codes** are associated with the get statement to perform additional functions. These include retrieving all records in a hierarchical path, locating first occurrence, locating last occurrence, and maintaining the currency indicators at a given level of the hierarchy or for the hierarchical path to this level.

Figure 9.20 A sample hierarchical definition tree.



The need to retrieve all records in a hierarchical path can be illustrated by the following example. Suppose we need to find an occurrence of a record type E and also list its parentage. Instead of successively retrieving the correct occurrence of record type A, then record type B, and subsequently record type E, we can combine these operations in one statement as given below:

get next *D where A = A_1, ***D where** B = B_{11}, **where** E = E_{112}

Here the control code is specified by ***D** and it indicates that the corresponding occurrence of the record types in the hierarchical path are also to be retrieved and placed in the UWA in the appropriate record template.

If we wanted the last occurrence of record type D in a hierarchical path, the following version of the get statement could be used. Here the last sibling in the D record type is indicated by the ***L** control code.

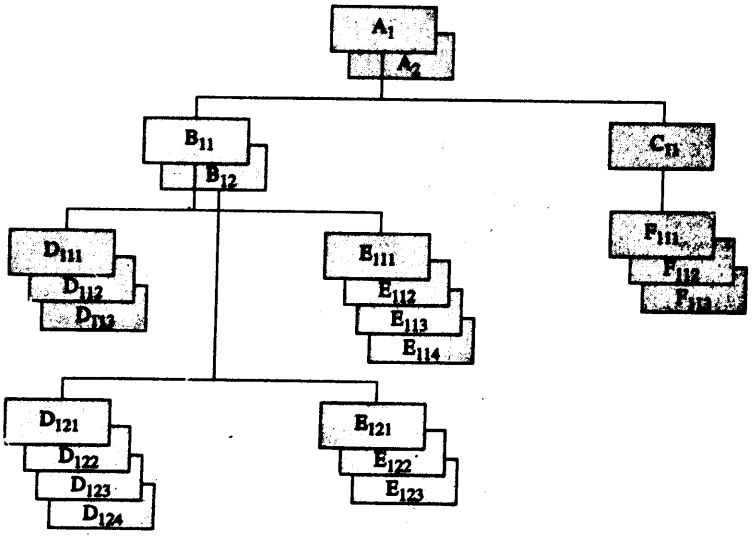
get unique *L D **within parent** A = A_1 **and** B = B_{11}

A similar command to back up to the first sibling in a record type, while performing a sequential retrieval using the **get next within parent** statement, is provided by the ***F** command code.

Another feature of the hierarchical DML is the possibility of maintaining and navigating through multiple dependent record types at each level of a hierarchical path. To understand this facility, consider the database shown in Figure 9.21. Suppose we want to list the dependent record types of B_{11} in the order of D_{111}, E_{111}, D_{112}, E_{112}, and so on. The following statements would cause a problem:

get next where A = A_1, **where** B = B_{11}
get next within parent D
get next within parent E
get next within parent D

Figure 9.21 Data corresponding to the hierarchical definition tree of Figure 9.20.



This is so because the database uses hierarchical pointers and for the second get request for record type D, it would access either the record occurrence D_{121} or return an error condition indicating that there are no more record types.

If we use multiple positioning, the position within the record type D would be maintained. Consequently, this would give us the correct occurrence, D_{112}, of the record type D.

9.8 Concluding Remarks

In the hierarchical model, we have to select the order of entities involved in the application into a hierarchy. This involves choosing the root node at each level. The ordering of the nodes at each level is also significant. Because we are unaware, at database design time, of the users' intent and range of needs, the number of different possible hierarchies with a sizable number of record types is enormous. Choosing among these hierarchies would be a formidable task. As a case in point, with two record types, the number of different hierarchies is two; with three record types, the number of different hierarchies is 12, and so on. Only some of these hierachies are suitable and the optimum for one application could turn out to be far from satisfactory for another application.

We face another problem in converting cyclic relationships into hierarchies. A cycle of relationship, for example,

BRANCH ⟷ DEPT_SECTION ⟷ EMPLOYEE ⟷ BRANCH

in the E-R diagram of Figure 9.2 cannot be expressed directly in the form of an ordered tree. However, we have resolved this cycle by the hierarchies:

BRANCH → DEPT_SECTION → EMPLOYEE
and
BRANCH → EMPLOYEE → DEPT_SECTION.

This resolution requires the use of replication or virtual records for the record types DEPT_SECTION and EMPLOYEE at the lowest level of the above hierarchies. In general, any set of relationships in E-R diagrams that forms a cycle can be converted to a number of rooted trees, using either replication or virtual records.

The hierarchical model inherently requires that the data in the database be structured in the form of a tree. However, some records in the database represent entities involved in more than one relationship. Furthermore, some of these relationships are of the many-to-many type. The implementation of these relationships using the hierarchical data model leads to a number of hierarchical trees that are unconnected except via a DBMS-supplied dummy root record. Such a collection of hierarchical trees is sometimes called a set of spanning trees. Hierarchical data manipulation facilities do not provide an easy means of accessing several hierarchical trees simultaneously. The virtual record facility allows the hierarchical data manipulation language to access data belonging to separate hierarchical trees. The virtual record facility also allow a record type to be included in several hierarchies without actual replication.

The paired bidirectional logical relationship, with its associated symmetrical virtual records, is one way to implement a many-to-many relationship. The database

system is aware of the replication of the common data in such virtual records and the need to maintain consistency.

However, the virtual record scheme, even without any intersection data fields, requires physical support in the form of related pointers to and from the logical parent. Virtual records cannot be defined dynamically but require some database reorganization to be defined and implemented in conjunction with the DBA.

Performance considerations may require the hierarchical database to have an index not only on the key field of the root node of the hierarchical tree but also on other fields of the root node of a hierarchical tree or subtree. This type of index, called a **secondary index,** is particularly useful for logical parent records.

The hierarchical model is considered to have a built-in bias that is physically implemented. This bias may not be good for all applications. Consequently, a logical structure using secondary indexes is useful. The use of virtual records avoids replication, and provides a logical view of the database. Unfortunately, the implementation of this is not as straight forward as a view in the relation data model. The virtual record facility requires support of the underlying physical database and hence preplanning and involvement of the DBA at database design time. Consequently, new virtual records may not be defined. The update operations on the database and the records that are associated with a virtual record are much more complex than the operations on DBTG sets.

The hierarchical model, through one of its major implementations in the IMS system from IBM, has the lion's share of the current corporate databases. IMS has matured over the years and the applications have been tuned to an optimum level of performance. The results of attempts to move some of these applications to a relational model have been mixed. However, a number of companies are marketing products to provide a relational user front end, that interfaces with the existing hierarchical DBMS.

9.9 Summary

The hierarchical data model consists of a set of record types. The relationship between two record types is of the parent/child form, expressed using links or pointers. The records thus connected form an ordered tree, the so-called hierarchical definition tree.

The hierarchical model provides a straightforward and natural method of implementing a one-to-many relationship. However, a many-to-many relationship between record types cannot be expressed directly in the hierarchical model. Such a relationship can be expressed by using data replication or virtual records.

The disadvantages of data replication are waste of storage space and the problem of maintaining data consistencies. A virtual record is a mechanism to point to an occurrence of a physical record. Thus, instead of replicating a record occurrence, a single record occurrence is stored and a virtual record points to this record wherever the record is required. The virtual record can contain some data that is common to a relationship; such data is called the intersection data. The virtual record is the logical child of the physical record that it points to, which is its logical parent.

The database using the hierarchical model results in a number of hierarchical

structure diagrams, each of which represents a hierarchical tree. These trees can be interrelated via the logical parent/child relationship to form a set of spanning trees. However, one can assume that the DBMS provides a single occurrence of a dummy record type and all the hierarchical trees can then be attached to this single dummy parent record. The roots of these trees can be treated as children of this dummy record.

The data manipulation facility of the hierarchical model provides functions similar to the network approach; however, the navigation to be provided is based on the hierarchical model. The get command is used to retrieve an occurrence of a specified record type that satisfies the specified conditions. The get next command is used for sequential processing and the get next within parent is used for sequential processing within a preselected hierarchy.

The database can be modified using the insert, replace, and delete operations. When records to be modified are virtual records, detailed rules have to be specified so that the modification, if allowed, leaves the database in a consistent state.

Key Terms

ordered tree
preorder traversal
child pointer
sibling pointer
hierarchical data model (HDM)
tree structure diagram
definition tree
hierarchical definition tree
virtual record
replication
logical parent
logical child
intersection data
paired bidirectional logical relationship
DB-Status
get
get first
get unique
get leftmost
where
get next
get next within parent
insert
get hold
replace
delete
hierarchical path
control codes
secondary index

Exercises

9.1 Write an algorithm to convert a network diagram into a hierarchical diagram.

9.2 Write an algorithm to convert a hierarchical diagram into a network diagram.

9.3 Consider the record types BOOK and CLIENT. Implement the relationship to model the waiting list of clients waiting to borrow a given BOOK.

9.4 Consider the record types BOOK_COPY and CLIENT. Implement the relationship to model the waiting list of clients waiting to borrow a given BOOK_COPY.

9.5 Comment on the statement that the HDM has limited network capabilities. Give an example of a network that cannot be represented in an HDM.

9.6 Why does the association between parent and child record type in the hierarchical data model *not* need the foreign key concept of the relational data model?

9.7 Figure A represents a hierarchical tree structure diagram for the hospitals in a certain area. Write the data description statements to define the structure.

Chapter

10

Query Processing

Contents

In this chapter we focus on different aspects of converting a user's query into a standard form and thence into a plan to be executed to generate a response

10.1 Introduction

Query processing is the procedure of selecting the best plan or strategy to be used in responding to a database request. The plan is then executed to generate a response. The component of the DBMS responsible for generating this strategy is called a **query processor.**

Query processing is also referred to in database literature as **query optimization.** However, bear in mind that optimization here is mostly in the form of improvement in light of the inexact knowledge of the status of the database. The optimization done in practical systems is not necessarily the best. The optimal strategy may be too difficult to evaluate and could require much more computing to improve on it which on average may not be dramatically different from the one afforded through a heuristic strategy.

Query processing is a stepwise process. The first step is to transform the query into a standard form. For instance, a query expressed in QBE is translated into SQL and subsequently into a relational algebraic expression. During this transformation process, the **parser** portion of the query processor checks the syntax and verifies if the relations and attributes used in the query are defined in the database. Having translated the query into a given form such as a relational algebraic expression, the optimization is performed by substituting equivalent expressions for those in the query. Such equivalent expressions, which we focus on in Section 10.4, are more efficiently evaluated than the ones in the transformed query. Substitution of such expressions also depends on factors such as the existence of certain database structures, whether or not a given file is sorted, the presence of different indexes, and so on. In the next step a number of strategies called **access plans** are generated for evaluating the transformed query. The physical characteristics of the data and any supporting access methods are taken into account in generating the alternate access plans. The cost of each access plan is estimated and the optimal one is chosen and executed.

We concentrate in this chapter on query processing for interactive usage on a relational database management system (RDBMS). A compiler would process database requests from batch programs. Techniques similar to the one to be discussed here could also be applied to compiled queries. The overhead involved in the query processing of an interactive query that is unlikely to be repeated should not be too high. Contrast this with the compilation of a batch query. A batch program is likely to be executed many times. Thus, a more intensive search for an optimal plan could be justified. However, the optimization of compiled queries is not guaranteed to remain optimal since the status of the database changes over time.

In the hierarchical and network models, the user specifies navigation though the database by indicating the low-level path to be followed through records. This path, for instance, leads from the parents to the children record types in a hierarchical database, or from the owners to the members record types (or from the members to the owners) of sets in the network database. Since these paths are already indicated, the onus of optimization is on the user. Nonetheless, even in these systems some

represents course offered and, ignoring the multiple sections of certain courses, represents 5,000 courses.

A given request can be expressed in a number of different ways in any language. Consider the query: "List the names of courses higher than COMP300 and all students registered in them."

The following are some different ways of stating this query in SQL and relational algebra. In SQL:

```
select Std_Name,Course_Name
from STUDENT, REGISTRATION, COURSE
where STUDENT.Std# = REGISTRATION.Std# and
      COURSE.Course# = REGISTRATION.Course# and
      REGISTRATION.Course# > COMP300
```

or

```
select Std_Name,c1.Course_Name
from STUDENT, REGISTRATION, COURSE c1
where STUDENT.Std# = REGISTRATION.Std# and
      REGISTRATION.Course# in
          (select c2.Course#
           from COURSE c2
           where c2.Course# > COMP300 and
                 c1.Course# = c2.Course# )
```

or

```
select Std_Name,c1.Course_Name
from STUDENT, COURSE c1
where STUDENT.Std# in
     (select REGISTRATION.Std#
      from REGISTRATION
      where REGISTRATION.Course# in
          (select   c2.Course#
           from COURSE c2
           where c2.Course# > COMP300 and
              c1.Course# = c2.Course# ))
```

In relational algebra:

$$\pi_{Std\_Name,Course\_Name}(\sigma_{Course\#>COMP300}\,(STUDENT \underset{Std\#}{\bowtie} REGISTRATION \underset{Course\#}{\bowtie} COURSE))$$

or

$$\pi_{Std\_Name,Course\_Name}(STUDENT \underset{Std\#}{\bowtie} (\sigma_{Course\#>COMP300}\,(REGISTRATION \underset{Course\#}{\bowtie} COURSE))$$

or

$$\pi_{Std\_Name,Course\_Name}(STUDENT \underset{Std\#}{\bowtie} (\sigma_{Course\#>COMP300}\,REGISTRATION) \underset{Course\#}{\bowtie} (\sigma_{Course\#>COMP300}\,COURSE)$$

Some of these illustrated forms may be better than others as far as the use of computing resources is concerned. The DBMS must perform a transformation to convert a query from an undesirable form into an equivalent one that uses less resources and is therefore deemed better.

For the sample database, we get the following query processing costs for the different relational algebraic forms of the same query. Here, to simplify discussion, we compare costs in terms of the number of tuples processed. In an actual system, the cost would be given in terms of the processing cost and the I/O cost measured in terms of the number of block accesses required. This I/O cost depends, too, on the size of the relation and block.

Let us examine the cost for the first relational algebraic expression tabulated in Figure 10.2a. It involves a join of the relation STUDENT, containing 40,000 tuples, with REGISTRATION, having 400,000 tuples. In this case, the referential integrity constraint indicates that a tuple in REGISTRATION cannot exist unless there is a tuple in STUDENT with the same *Std#*. Therefore, the result would be equal to the number of tuples in REGISTRATION. If we use the brute force method of comparing each tuple of STUDENT with each tuple of REGISTRATION, this join is obtained by processing 40,000 * 400,000 tuples.

If the STUDENT and REGISTRATION relations are sorted on the joining attribute *Std#*, then the join can be obtained by processing 40,000 + 400,000 tuples. If indexes exist on the joining attribute, one per relation, then access to the tuples is not required unless the indexes indicate that there is a tuple in both relations with a common value for the joining attribute. We discuss these aspects in Section 9.8.

The second join is between the result of the first join and the tuples of COURSE involving a processing of 5,000 * 400,000 tuples. The result of this, again, would be 400,000 tuples. This is followed by a selection for *Course* > COMP300. If we assume that there are 500 courses whose course number is higher than COMP300, the result would involve, let us say, 40,000 tuples. The final result of the query is obtained by projecting these tuples on the attributes *Std_Name* and *Course_Name* and involves processing 40,000 tuples.

For the second relational algebraic form of the same query, the first join is between the relations REGISTRATION and COURSE. This entails the processing of 5,000 * 400,000 tuples for unsorted relations. If both these relations were sorted the join would involve processing 5,000 + 400,000 tuples. The result of this join is 400,000 tuples. We then select from the joined tuples those wherein the *Course#* is greater than COMP300, requiring the processing of 400,000 tuples to produce a result consisting of 40,000 tuples. This is subsequently joined with the tuples of the STUDENT relation, requiring processing 40,000 * 40,000 tuples or 40,000 + 40,000 tuples for unsorted and sorted cases, respectively. The final projection operation involves 40,000 tuples. These costs are tabulated in Figure 10.2b.

Let us now consider the third form of the relational algebraic query. The selection is done before each of the joins. The selection on COURSE entails the processing of 5,000 tuples to generate 500 tuples with *Course#* > COMP300. Similarly, the selection on REGISTRATION involves processing 400,000 tuples to select 40,000 tuples. The join of the STUDENT with the selected tuples of REGISTRATION involves processing 40,000 * 40,000 tuples to arrive at 40,000 resulting tuples. This result is joined with 500 tuples selected from COURSE and entails a processing of 500 * 40,000 tuples. The result is, as before, 40,000 tuples. We notice, however, that the amount of processing is considerably reduced. These costs are tabulated in Figure 10.2c.

the verification of the existence of a relation (or attribute) has to be performed at the time of the initial analysis of the query. For internal use, it is convenient to represent queries using a procedural format. This rules out relational calculus and algebra for internal representation, even though these formats have been used in a number of query processors. We use operator graphs for internal representation of queries in this text.

Operator Graphs

An **operator graph** depicts how a sequence of operations can be performed. In operator graphs, operations are represented by nodes and the flow of data is shown by directed edges. The graph visually represents the query and is easily understood. Consider the query: "List the names of students registered in the Database course." One possible algebraic formulation is:

$$\pi_{Std\_Name}(\sigma_{Course\_Name = 'Database'}(\text{STUDENT} \bowtie \text{REGISTRATION} \bowtie \text{COURSE})$$

An operator graph for the above sample query is shown in Figure 10.3.

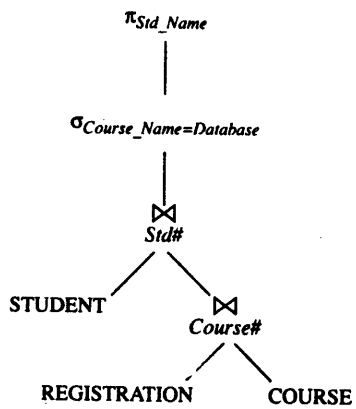
Equivalence transformations such as the earlier application of the selection operation can be used to modify the graph. The graph clearly shows what the effect of such a transformation would be. For most simple queries, the graph resembles a tree. Later on we demonstrate how the graph can be used to discover redundancies in query expressions.

Steps in Query Processing

The steps involved in query processing are as follows:

1. **Convert to a standard starting point.** We would use a relational algebraic form and the operator graph as the starting point. We would also assume that the query expression is in **conjunctive normal form,** that is, the query is of the form $p_1 \vee p_2 \vee \ldots$, where each disjunct p_i is a conjunction of terms $t_{11} \wedge t_{12} \wedge \ldots$.

Figure 10.3 Example of an operator graph.



2. **Transform the query.** The query is transformed by replacing expressions in the query with those that are likely to enhance performance. Note that the choice of an equivalent form may be influenced by the existence of an index or the fact that a relation is sorted.
3. **Simplify the query.** The query is simplified by removing redundant and useless operations. We discuss query improvement in Section 10.7.
4. **Prepare alternate access plans.** The alternate access plans indicates the order in which the various operations will be performed and the cost of each such plan. The cost depends on whether or not the relations are sorted and the presence or absence of indexes. The optimal access plan is chosen.

Steps 2, 3, and 4 are usually done in conjunction with each other and use statistical information to derive the best possible form of the query and the associated access plan. The query transformations are carried out by applying standard processing strategies. We discuss some of these strategies for processing a query below and discuss some equivalent forms in Section 10.4.

10.3.2 General Processing Strategies

Recall Example 4.3, in which we illustrated the decrease in the size of join when a selection operation on one of the relations participating in the join was performed before the actual join. Since selection reduces the cardinality of a relation, the join would involve a relation with a smaller number of tuples and could be executed faster. There are a number of similar general strategies used in query processing to reduce the size of the intermediate and final results as well as processing costs. They are described below.

1. **Perform selection as early as possible.** Selection reduces the cardinality of the relation and, as a result, reduces the subsequent processing time.
2. **Combine a number of unary operations.** Consider the evaluation of $\pi_X(\sigma_Y(R))$, where $X, Y \subseteq \mathbf{R}$. Both the selection and projection operations can be done on the tuples of R simultaneously, requiring a single pass over these tuples and singular access to them. Similarly,

$$\sigma_{C1}(\sigma_{C2}(R)) \equiv \sigma_{C1 \wedge C2}(R), \qquad \pi_X(\pi_Y(R)) \equiv \pi_{X \cap Y}(R)$$

If $X \subseteq Y$, then $\pi_X(\pi_Y(R)) \equiv \pi_X(R)$

3. **Convert the cartesian product with a certain subsequent selection into a join.** Consider the evaluation of $\sigma_Y(R * S)$, where Y is, let us say, $A\ \theta\ B$ and $A \in R$, $B \in S$. In this case, the cartesian product can be replaced by a theta join as follows:

$$R \underset{A\theta B}{\bowtie} S$$

4. **Compute common expressions once.** A common expression that appears more than once in a query may be computed once, stored, and then reused. This is advantageous only if the size of the relation resulting from the common expression is small enough to be either stored in main memory or accessed from secondary storage at a total cost less than that of recomputing it. Bear in

4. **Use associative and commutative rules for joins and cartesian products.**

$R \bowtie S \equiv S \bowtie R$

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

$R * S \equiv S * R$

$R * S * T \equiv R * (S * T) \equiv (R * S) * T \equiv (R * T) * S \equiv \ldots$

The order of the join and product is very important as it can substantially affect the size of the intermediate relations and, therefore, the total cost of generating the result relation.

Example 10.3

In Example 10.1, the expression

$$\sigma_{Course\#>COMP300}(STUDENT \underset{Std\#}{\bowtie} REGISTRATION) \underset{Course\#}{\bowtie} COURSE)$$

can be replaced by the more efficient expression:

$$(STUDENT \underset{Std\#}{\bowtie} (\sigma_{Course\#>COMP300} REGISTRATION) \underset{Course\#}{\bowtie} (\sigma_{Course\#>COMP300} COURSE)$$

The above expression is equivalent to the following:

$$((\sigma_{Course\#>COMP300} REGISTRATION) \underset{Course\#}{\bowtie} (\sigma_{Course\#>COMP300} COURSE) \underset{Std\#}{\bowtie} STUDENT) \blacksquare$$

5. **Perform selection before a join or cartesian product.** Consider $\sigma_C(R \bowtie S)$. If the attributes involved in the condition C are in the scheme of R and not in S, that is, attr(C) $\in$ **R** and attr(C) $\notin$ **S**, then

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

If the attributes involved in the condition C are in the scheme of S but not in R, i.e., attr(C) $\in$ **S** and attr(C), $\notin$ **R**, then

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

If the attributes involved in the condition C are in the scheme of **R** and **S**, i.e., attr(C) $\in$ **R** and attr(C) $\in$ **S**, then

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

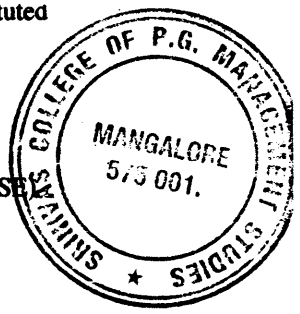
If C = C1 $\wedge$ C2 and the attributes involved in the condition C1 are from **R**, i.e., attr(C1) $\in$ **R**, and the attributes involved in the condition C2 are from **S**, i.e., attr(C2) $\in$ **S**, then

$$\sigma_C(R \bowtie S) \equiv \sigma_{C1}(R) \bowtie \sigma_{C2}(S)$$

If C = C1 $\wedge$ C2 $\wedge$ C3 and the attributes involved in the condition C2 are only in **R**, i.e., attr(C2) $\in$ **R** $\wedge$ attr(C2) $\notin$ **S**, the attributes involved in the condition C3 are only in S, i.e., attr(C3) $\in$ **S** $\wedge$ attr(C3), $\notin$ **R**, and the attributes involved in the condition C1 are in **R** and **S**, then

$$\sigma_C(R \bowtie S) \equiv \sigma_{C1}(\sigma_{C2}(R) \bowtie \sigma_{C3}(S))$$

The above equivalences also apply when the cartesian product operation is substituted for the join.



Example 10.4

Consider the expression:

$\sigma_{Std\# > 1234567 \wedge Course\# = COMP353 \wedge Course\_Name = \text{'Database'}}$(GRADE ⋈ COURSE)

It is equivalent to:

$\sigma_{Course\# = COMP353}((\sigma_{Std\# > 1234567}(GRADE)) \bowtie (\sigma_{Course\_Name = \text{'Database'}}(COURSE)))$ ■

It is possible to combine projections with a binary operation that precedes or follows it. Only the attribute values specified in the projection need to be retained. The remaining ones can be eliminated as we evaluate the binary operation.

6. **Perform a modified projection before a join.** Note that when a projection operation is preceded by a join, it is possible to push the projection down before the join, but the projection acquires new attributes. This necessitates performing the original projection after the join. However, unless the cardinalites of intermediate relations are reduced, which would reduce the cost of the join operation and the subsequent size of the joined relation, the usefulness of pushing a projection before a join is questionable.

$$\pi_X(R \bowtie S) \equiv \pi_X(\pi_{R'}(R) \bowtie \pi_{S'}(S))$$

where $R' = \mathbf{R} \cap (X \cup \mathbf{S})$ and $S' = \mathbf{S} \cap (X \cup \mathbf{R})$, and **R, S** represent the set of attributes in these relation schemes. When $X \equiv \mathbf{R} \cup \mathbf{S} - \mathbf{R} \cap \mathbf{S}$, there is no improvement because $R' \equiv R$ and $S' \equiv S$.

Example 10.5

Consider the relations GRADE *(Std#, Course#, Grade)* and COURSE *(Course#, Course_Name, Instructor)*. The expression

$$\pi_{Std\#, Course\_Name}(GRADE \bowtie COURSE)$$

is equivalent to:

$$\pi_{Std\#, Course\_Name}(\pi_{Std\#, Course\#}(GRADE) \bowtie \pi_{Course\#, Course\_Name}(COURSE))$$

However, consider the relations STUDENT *(Std#, Std_Name)* and REGISTRATION (Std#,Course#). The expression

$$\pi_{Std\_Name, Course\#}(STUDENT \bowtie REGISTRATION)$$

is equivalent to:

$$\pi_{Std\_Name, Course\#}(\pi_{Std\#, Std\_Name}(STUDENT) \bowtie \pi_{Std\#, Course\#}(REGISTRATION))$$

which is equivalent to the original query:

$$\pi_{Std\_Name, Course\#}(STUDENT \bowtie REGISTRATION))$$ ■

7. **Commuting projection with a cartesian product.** Consider the expression $\pi_X(R * S)$. This expression can be replaced by the following equivalent one under

In this chapter we restrict ourselves to centralized database systems, for which the communication cost would be zero. We return to distributed query processing in Chapter 15.

Selection, projections, and joins affect the sizes of the resulting relations. The effect of projection is simple to calculate if the sizes of the attribute values are known. The effect of selections and joins is more involved.

We are interested in the size of the result relation for several reasons. First, the result relations could be intermediate relations and their size would be required to determine the cost of the succeeding part of the query expression. Second, the result relation may be too large to be stored in primary memory and would have to be written to secondary storage. We may want to compare the cost of this access with alternate equivalent query expressions.

Let us assume that the values of an attribute are uniformly distributed over its domain and that the distribution is independent of values in the other attributes. These assumptions are usually made for simplifying cost calculations, and it should be noted that these assumptions cannot be justified on any other grounds. In practice both uniform distribution and independence are unlikely to occur. In that case, the expressions become complicated and are beyond the scope of this text.

10.5.1 Selection

Let $T = \sigma_C (R)$ represent the selection of relation R on condition C, and let C be a simple clause of the form R[*A*] = constant. Before we can estimate the size of the resultant relation we must possess some knowledge about the value distributions, that is, the number of times an attribute takes a particular value. We can simply assume that each value occurs with equal probability. Then the expected number of tuples in relation T is given by

$$|T| = \frac{1 * |R|}{|R[A]|}$$

where |R[*A*]| is the number of distinct values for attribute A of relation R. The factor 1/|R[*A*]| is known as the **selectivity factor** and is usually represented by the symbol ρ (rho). As illustrated in Example 10.10, the nature of the data may allow an estimation of some selectivity factors.

Example 10.10

Recall that in the university database example, the assumption that each student is registered in 10 courses is a reasonable assumption. Therefore, we expect that

$$\sigma_{Std\# = 1234567}(\text{REGISTRATION})$$

will have ten tuples and

$$\sigma_{Course\# = \text{COMP453}}(\text{REGISTRATION})$$

will have 80 tuples if there are 5000 courses. We recognize that in reality, there will be considerable variations on these values. However, we can use them as estimates. ■

As discussed in Chapter 3, it is unfortunately not reasonable to assume uniform distribution of values in all cases. Uniform distribution assumption is widely used nonetheless for estimating costs in choosing a query processing strategy. We should therefore bear in mind that this is just an estimate.

Having generated the relation T (consisting of the tuples of relation R, satisfying the predicate C, involving the attribute A), suppose we need to estimate the number of distinct values for the attribute B in T. Note that $B \neq A$ and the number of distinct values for B in the relation T is given by $|T[B]|$.

We assume that the occurrence of a value in attribute B is unaffected by the values in A. In other words, the distributions are independent. Under these assumptions, it can be shown that this problem is equivalent to the so-called colored balls problem. In this problem we have n balls of m different colors. (Apart from color, all balls are identical.) Each color is represented by the same number of balls. We must determine the expected number of different colors represented by a random selection of t of these n balls.

It can be shown that the expected number of colors in these t balls is given by the following expression:

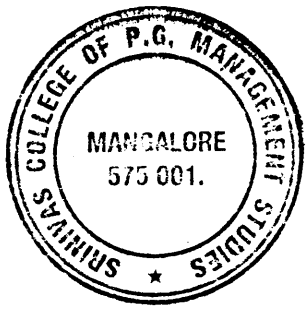
$$\text{expected number of colors} = m * \left[1 - \prod_{i=1}^{t} \frac{n((m-1)/m - i + 1}{n - i + 1}\right]$$

We can estimate $|T[B]|$, the expected number of different values for the attribute B in T, by the following substitution in the above expression: $n = |R|$, $m = |R[B]|$, and $t = |T|$.

However, the computation involved in evaluating this expression is considerable. As a result, a number of different approximations to the above expression have been proposed. We present below one of the more widely used approximations. This approximation is given by the following formula for different sizes of the relation T:

$$|T[B]| = \begin{bmatrix} |T| & \text{if } |T| < \frac{|R[B]|}{2} \\ \frac{(|T| + |R[B]|)}{3} & \text{if } \frac{|R[B]|}{2} \leq |T| \leq 2 * |R[B]| \\ |R[B]| & \text{if } |T| > 2 * |R[B]| \end{bmatrix}$$

The size of each tuple in relation T is the same as in relation R.



10.5.2 Projection

The cardinality of the resulting relation could be affected by a projection because duplicates would be deleted; however, most commercial database systems only delete duplicates as a result of explicit commands.

$$T = \pi_X (R)$$

where X is a set of attributes, $X \subseteq R$.

When X is a single attribute, or contains the key attribute of R, and we represent the single or key attribute by A, then

$$|T| = |R[A]|$$

If A is a key attribute of R then $|T| = |R|$.
When X is a set of attributes, then

$$|T| = \prod_{A_i \in X} |R[A_i]|$$

In the above estimation of the result we are assuming that the relation is a cartesian product of the values of its attributes. Such an assumption is rarely justified. We can take this as the worse case estimate. The upper limit in the above expression is given as:

$$|T| \leq |R|$$

The size of the tuples of T is the sum of the size of the attributes in X.

10.5.3 Join

The join operation is very common in relational database systems. The size estimation for the result of a join is somewhat more complicated than that of selection because the cardinality of the result relation depends on the distribution of values in the joining attribute. Furthermore, the cost of evaluating a join is not reflected in the size of the result. The cost depends on the size of the relations being joined. We are, however, interested in estimating the size of the result, since it could be used in subsequent operations in evaluating a query.

Since the size of the result depends on the values of the joining attributes and the distribution of these values, we shall consider a number of special cases.

Let

$$T = R \underset{R.A = S.B}{\bowtie} S$$

Estimating the cardinality of T is complex because it is difficult to estimate correctly the number of tuples of each relation that join with tuples of the other relation. In the worse case the join is equivalent to a cartesian product; this occurs when the operand relations do not share attributes defined on common domains. In such cases, the cardinality of the result relation is given by:

$$|T| \leq |R| * |S|$$

This value of cardinality is much too large for most practical databases. We consider a number of special cases below, assuming a uniform distribution of values.

1. Let $\{A\}$ represent the set of values that the attribute A takes in the relation R. The number of distinct values for attribute A is given by $|R[A]|$. We assume uniform distribution of these values and further assume that these values will also be in relation S. In this case, we could conclude that there are $|S|/|R[A]|$ tuples in S for each value for attribute A. Therefore, each tuple in R joins with $|S|/|R[A]|$ tuples in S and the number of tuples in T is given by:

$$|T| = \frac{|R| * |S|}{|R[A]|}$$

Let $\{B\}$ represent the set of values that attribute B takes in relation S. The number of distinct values for attribute B is given by $|S[B]|$. Again, using uniform distribution and further assuming that these values would also be in relation R, we could conclude that there are $|R|/|S[B]|$ tuples in R for each value for attribute B. This means that each tuple in S joins with $|R|/|S[B]|$ tuples in R, and it follows that the number of tuples in T is given by:

$$|T| = \frac{|R| * |S|}{|S[B]|}$$

If $\{A\} \neq \{B\}$, then $|R[A]| \neq |S[B]|$ and the values for $|T|$, obtained by the expressions $(|R| * |S|)/|R[A]|$ and $(|R| * |S|)/|S[B]|$, would be different. This indicates that there are tuples in R and S that do not participate in the join. Such tuples are called dangling tuples.

The greater, average, or the lesser of $(|R| * |S|)/|R[A]|$ and $(|R| * |S|)/|S[B]|$ could be taken as the estimate of the size of T.

2. If A is the key of R, then every tuple of S can only join with one tuple of R, i.e., the cardinality of the resultant relation cannot be greater than the cardinality of S:

$$|T| \leq |S|$$

3. Another possible derivation of an estimate, which takes into account the size of the domain and which estimates a much smaller value for the cardinality of the join, is as follows. The number of distinct values of A in R and B in S is $|R[A]|$ and $|S[B]|$, respectively. Assuming uniform distribution as before, each value of A in R (B in S) is associated with $|R|/|R[A]|$ tuples $(|S|/|S[B]|)$. Thus, for each value of A (or B) in the join, we could derive the upper limit on the number of tuples in the join as given below:

$$\frac{|R| * |S|}{|R[A]| * |S[B]|} \text{ tuples}$$

The above will hold if the same set of values are in both R and S. Since the same set of values is unlikely to be in the two relations, the expected number of common domain values is much lower. This expected number depends on the probability of any value appearing in both the relations. The expected number of distinct values of A in R (or B in S) that takes part in the join is given by:

$$\frac{|R[A]| * |S[B]|}{|D|}$$

where $|D|$ is the cardinality of the domain of A and B. Therefore, the expected actual size of the join is given by:

$$|T| = \frac{|R[A]| * |S[B]|}{|D|} * \frac{|R| * |S|}{|R[A]| * |S[B]|}$$
$$= \frac{|R| * |S|}{|D|}$$

The size of tuples of T equals the sum of the sizes of tuples of R and S, minus the size of the joining attribute A (or B).

As we discussed under rule 6 in Section 10.4, a projection cannot be simply moved down. Given relations R and S defined on the relation schemes **R**(X,Y,Z) and S(X,Y,W), where W. X, Y, Z are sets of attributes, then

$$\pi_X(R \underset{Y}{\bowtie} S) \equiv \pi_X(\pi_{X,Y}(R) \underset{Y}{\bowtie} \pi_{X,Y}(S))$$

In other words, as the projection is pushed down, it acquires additional attributes. These additional attributes finally have to be eliminated by the original projection. This is illustrated in the following example.

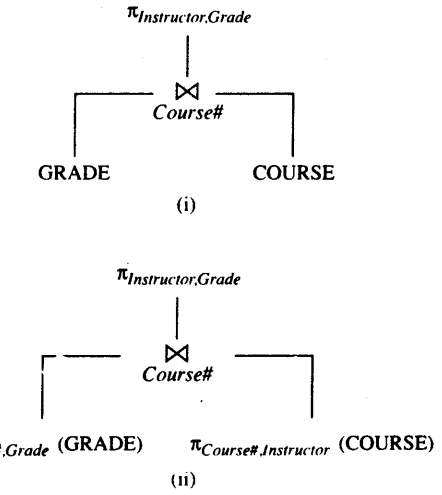
Example 10.12

Consider the query: "Compile a list of instructors and the grades they assign." The relational algebraic expression for this query is given below:

$$\pi_{Instructor,Grade}(\text{GRADE} \bowtie \text{COURSE})$$

The corresponding query tree is given in Figure Bi. To push the projection down the tree, we would have to include the common attribute *Course#* of GRADE and COURSE in both branches of the join operation as indicated in Figure Bii.

Figure B Pushing projection down the query tree.



Example 10.13 illustrates the effect of pushing the projection operation down the query tree.

Example 10.13

Consider the query: "List the names of the students in the Database course." The relational algebraic expression for this query is given below:

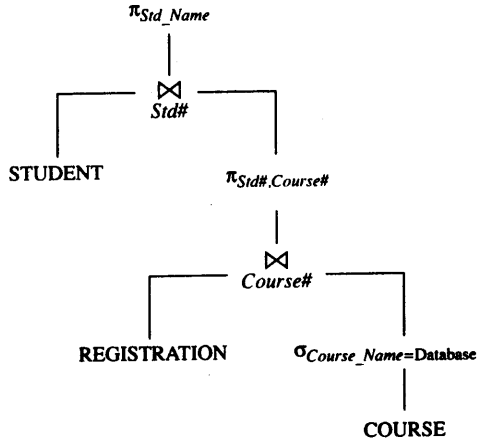
$$\pi_{Std\_Name}(\text{STUDENT} \underset{Std\#}{\bowtie} \pi_{Std\#,Course\#}(\text{REGISTRATION} \underset{Course\#}{\bowtie} (\sigma_{Course\_Name = \text{'Database'}}(\text{COURSE}))))$$

This expression can be simplified by moving the second projection further to the right in the expression, before the join on *Course#*. In the case of the relation REGISTRATION the projection is the entire relation and for COURSE the projection is on the attribute *Course#*. The modified expression is shown below:

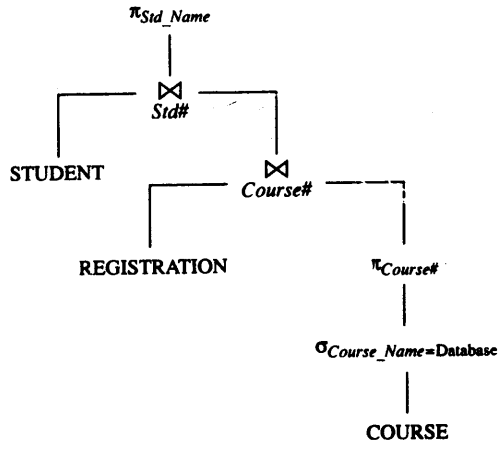
$$\pi_{Std\_Name}(\text{STUDENT} \underset{Std\#}{\bowtie} (\text{REGISTRATION} \underset{Course\#}{\bowtie} \pi_{Course\#}(\sigma_{Course\_Name = \text{'Database'}}(\text{COURSE}))))$$

The effect of pushing the projection operation down the query tree is illustrated in Figures Ci and Cii. Since the projection on the attributes *Std#* and *Course#* of the relation REGISTRATION is the entire relation, the operation is redundant and dropped. *Course#* is the only attribute appearing

Figure C Effect of pushing down projection operator.

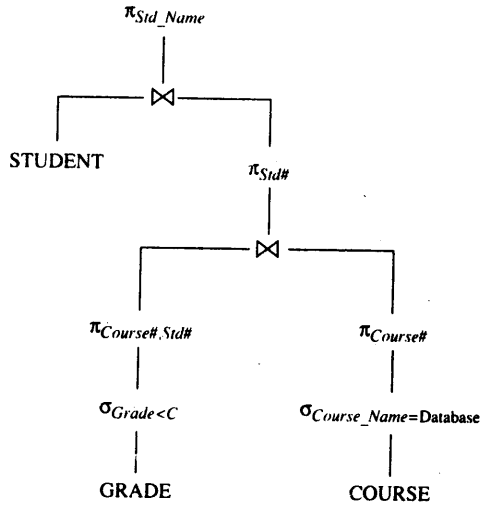


(i)



(ii)

Figure G Final optimization for Example 10.14.



Finally, we can push down the selections and projections to give us the tree of Figure G. ■

10.8 Query Evaluation

We have presented a sampling of the many different query improvement strategies. Having found the best equivalent form of a query, the next step is to evaluate it. We classify the query evaluation approaches according to the number of relations involved in the query expression. Thus, we distinguish between the approach to be used when the query expression involves one, two, or many relations. These are known as one-variable, two-variable, and N-variable expressions, respectively. The last stage of query processing deals with the execution of access plans. A number of different query evaluation strategies have been proposed. Here we look at some commonly implemented techniques.

10.8.1 One-Variable Expressions

A **one-variable expression** involves the selection of tuples from a single relation. Let us consider the SQL query:

select al, . . ., ak
from R
where p

The simplest approach would involve reading in each tuple of the relation and testing it to ascertain if it satisfies the required predicates. This is illustrated below.

Sequential Access

Use **sequential access** to read in every tuple of the relation. If the tuple satisfies the qualification conditions, include the projection of the tuple on the target list attributes in the result relation. The algorithm is given below:

```
result := ∅ {empty}
for every r in R do
    if satisfies (p, r)
        then result := result + <r.a1 . . . r.ak>
```

where <r.a1 . . . r.ak> represents the tuple obtained by concatenating the projections of r onto the attributes in the target list.

If the relation has n tuples that are blocked as b tuples/block, then for sequential access to the tuples, the number of block accesses is ⌈n/b⌉. In dealing with large relations, this is an inefficient approach, as illustrated in Example 10.15.

Example 10.15

Consider the REGISTRATION relation to evaluate the query: "Generate the list of students (*Std#* only) enrolled in COMP353." The SQL version of this query is:

select *Std#*
from REGISTRATION
where *Course#* = COMP353

We use sequential access to the tuples of REGISTRATION. Suppose there are 400 tuples per block of secondary storage devices. Reading in all tuples of REGISTRATION would involve access to 400,000/400 = 1,000 block accesses. ■

Access Aid

The number of tuples needing to be accessed could be reduced if the relation is sorted with respect to one or more attributes. In such cases, if the predicates involve one or more attributes on which the relation is sorted, then only some of the tuples need be accessed. Use of indexes can provide faster access to the required tuples.

Example 10.16

Let us reconsider the previous example of generating the list of students enrolled in COMP353. If the tuples of REGISTRATION are sorted in order based on *Course#* and the records are clustered with 400 tuples per block, we could do a binary search on these blocks. Locating the block containing the required course would limit access to about 10 blocks. This will be followed by access to at most one additional block. The last block accessed would be needed only if some 80 tuples with the required course number were not in the same block. This gives us a total of approximately 11 block accesses. ■

number of tuples per block, bf_S, for the STUDENT relation is 200, the bf_R for REGISTRATION is 400, and up to 5 blocks of the STUDENT relation can be kept in main memory. The nested loop using block access with STUDENT, the smaller relation in the outer loop, would involve a total of 40,200 disk accesses. If the smaller relation in the outer loop could be kept entirely in memory, then the number of disk accesses would be 1200. Note that this method requires sorting the result relation on the attribute *Course#* to obtain class lists. ■

Sort and Merge Method

Relations are assumed to be sorted in the **sort and merge method.** If they are not sorted, a preprocessing step in the query evaluation sorts them. These sorted relations can be scanned in ascending or descending order of the values of the join attributes. Tuples that satisfy the join predicate are merged. The process can be terminated as indicated in Algorithm 10.1 on page 491.

In the algorithm, we join the relation R with relation S and the join predicate is $R.A = S.B$. We assume that the relations have been sorted in ascending order with respect to the attributes A and B and that sufficient space for an appropriate number of buffers in available. The tuples are placed in the buffers by the file manager and the algorithm reads the tuples from these buffers. R ↑ and S ↑ are pointers that point to the corresponding tuples in the buffers. We assume that once the last tuple in a buffer has been read, the buffer is refilled. If the joining attributes are not the primary key of the relations, a many-to-many relationship could exist via the joining attributes. We use an array U where pointers to tuples of relation S that have the same attribute value as the current tuple of R are stored. These tuples join with the current tuple of the relation R and allow a single pass over the tuples of both the relations. A tuple whose pointer has been stored in this array locks the tuple so that the buffer containing it is not released. An attempt to read past the last tuple in the relation would raise the *eof* (end-of-file) condition. The algorithm could be easily modified to include cases where the join involves more than one attribute.

The number of accesses for Algorithm 10.1 is given by:

$$\lceil |R|/bf_R \rceil + \lceil |S|/bf_S \rceil + R_{CS} + S_{CS}$$

where R_{CS} and S_{CS} are the costs of sorting the relations, assumed to be equal to the number of accesses required during the sorting of the relations R and S, respectively. The sort costs depend on memory availability and the number of runs produced in the initial sort stage. For example, if we have enough memory to perform a max(N,M)-way merge, where the number of runs produced for R and S are N and M, respectively, then the number of accesses required for the join is as follows:

Initial read: $\lceil |R|/bf_R \rceil + \lceil |S|/bf_S \rceil$ blocks
Writes of the sorted runs: $\lceil |R|/bf_R \rceil + \lceil |S|/bf_S \rceil$ blocks
Read in merge phase: $\lceil |R|/bf_R \rceil + \lceil |S|/bf_S \rceil$ blocks
Writes of the join: $\lceil |T|/bf_T \rceil$ blocks

Note that T is the result relation and bf_T is the blocking factor for it. Similar calculations can be done for other memory sizes.

Algorithm

10.1 Sort-Merge to Include a Many-to-Many Relationship

Input: R, S, the two relations to be joined on attributes A and B, respectively.

Output: T, the relation that is the join of R and S (concatenation of the attributes of R and S, including the attributes A and B).

```
begin {sort-merge}
T := empty
sort R by A values and S by B values in ascending order
read (R)
read (S)
while not (eof(R) or eof(S)) do (* main while loop *)
begin
  while not(eof(R) or eof(S) or R↑.A ≠ S↑.B) do
                    (* find a join value *)
  if R↑.A < S↑.B
    then read(R)
    else read(S)
  if not (eof(R) or eof(S))
     then
       begin (*join a R tuple with one or more S tuples*)
         n := 0
         R_current.A := R↑.A
         while S↑.B = R_current.A and not (eof(S)) do
           begin
             n := n+1
             U[n] := S↑
             read (S)
           end
         while R↑.A = R_current.A and not (eof(R)) do
           begin
             for i = 1 to n do
               T := T + R↑ || U[i]↑
             read(R) (*does another tuple of R join with
                          the tuples whose pointers are in
                                        array U?*)
           end
       end
  end (*main while loop*)
end (*sort-merge*)
```

10.8.3 N-Variable Expressions

An **n-variable expression** involves more than two variables. The strategy used here is to try to avoid accessing the same data more than once. One method of implementing such expressions is to simultaneously evaluate all terms of the query. Therefore, if a number of terms in the query require unary operations on the data accessed, these could be done in parallel. If the data accessed participates in binary operations, these binary operations are partially evaluated.

General n-variable queries can be reduced for evaluation by either tuple substitution or decomposition.

Tuple Substitution

In the **tuple substitution method** we substitute the tuples for one of the variables. Consequently, we reduce the query to $K_1 * (n-1)$-variable queries, where K_1 is the cardinality of the substituted variable. The process is repeated until we get a set of one-variable queries. This process is an extension of the nested loop approach and requires the processing of tuples equal to the cartesian product of all relations participating in the query.

Example 10.22

Consider the query: "Compile a list giving the *Std#*s and *Std_Names* of students who, having failed the Database course, are taking it again." Note that we assume that the GRADE relation contains the best grade a student received in a given course. For a student who failed a course and subsequently passed it, the only tuple in the GRADE relation would be the one involving the second attempt!

The SQL and relational algebraic forms of this query are:

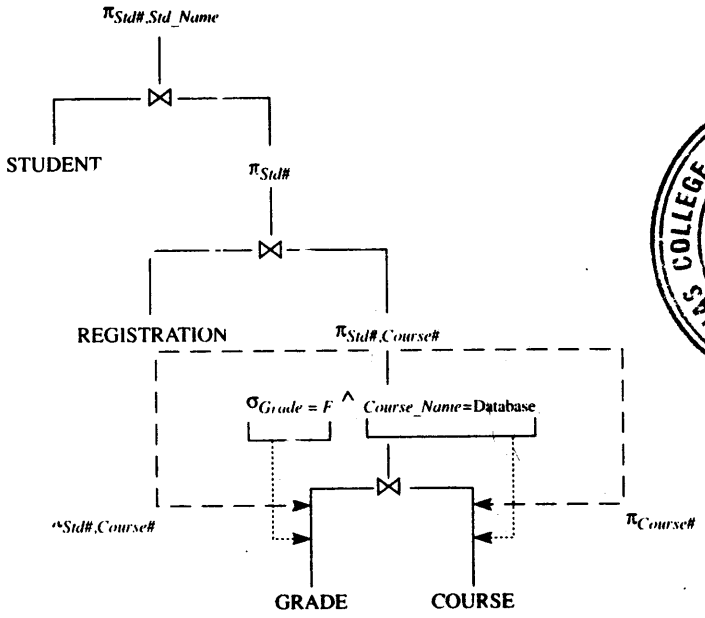
```
select Std#, Std_Name
from STUDENT s, REGISTRATION r, GRADE g, COURSE c
where s.Std# = r.Std# and
      c.Course_Name = 'Database' and
      g.Std# = s.Std# and
      g.Course# = c.Course# and
      g.Grade = F and
      r.Std# = g.Std# and
      r.Course# = c.Course#
```

$$\pi_{Std\#,Std\_Name}(\text{STUDENT} \bowtie \pi_{Std\#}(\text{REGISTRATION} \bowtie \pi_{Std\#,Course\#}(\sigma_{Grade=\text{F} \wedge Course\_Name=\text{'Database'}}(\text{GRADE} \bowtie \text{COURSE}))))$$

This query can be evaluated by substituting the value of each tuple of the four relations involved in the query. The number of tuples to be processed is approximately equal to 40,000 * 400,000 * 600,000 * 1,000. ■

Even though the substitution method will always work, it should be avoided because of the exponential increase in the number of tuples to be processed.

Figure 10.4 Moving selection and projection down the query tree.



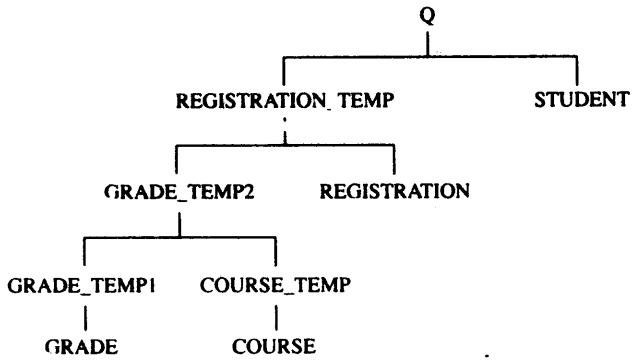
Note that we could use the optimization strategies discussed earlier to reduce the cost. One such operation involves moving the selection operations, as indicated in the query tree of Figure 10.4. This optimization scheme leads us to modify the tuple substitution scheme. In this modified scheme, the cardinality of one or more of the participating relations is reduced by selection or projection. For instance, instead of substituting all tuples of GRADE and COURSE, these relations could be scanned once and their cardinality restricted to those tuples that satisfy the query predicates.

Similar query modifications could be achieved in SQL or QUEL by a nested select statement or by using temporary relations, as illustrated below.

Using nested select in SQL:

```
select Std#, Std_Name
from STUDENT s
where s.Std# in
      (select r.Std#
      from REGISTRATION r
      where r.Course# =
            (select c.Course#
             from COURSE c
             where c.Course_Name = 'Database') and
             r.Std# =
            (select g.Std#
             from GRADE g
             where g.Grade = F and
                   g.Course# =
```

Figure 10.5 Decomposition of query of Example 10.22.



GRADE_TEMP2 and REGISTRATION to evaluate REGISTRATION_TEMP. The latter is used in the final stage of the query to compile the required list. In this decomposition, evaluation of GRADE_TEMP1 and COURSE_TEMP involves a one-variable query. GRADE_TEMP2 is a two-variable query, as are REGISTRATION TEMP and Q. Suppose there are 60,000 tuples in GRADE_TEMP1 with a grade of F (obtained after processing the 600,000 tuples of GRADE) and one tuple with the course name of Database (obtained after processing 5,000 tuples of COURSE). The number of tuples in GRADE_TEMP2 would be, let us say, 6. If only two of these students are reregistered, the tuple substitution at the point of evaluating Q involves finding only the names of these two students who have failed the Database course and are reregistered in the course. This tuple substitution results in the following:

retrieve *Std#*, *Stud_Name* **where** *Std#* = 1234567
retrieve *Std#*, *Stud_Name* **where** *Std#* = 7654321

In the decomposition approach, an n-variable query is replaced by a sequence of single variable queries. If this is impossible or undesirable, the query is split into two subqueries with a single common variable between them. Such subqueries could be recursively decomposed until they become single variable queries or irreducible. A query is reducible if it can be separated into two subqueries with a common variable, each of the subqueries having at least two variables. An irreducible subquery cannot be reduced and must be evaluated.

Some of the relations involved in the subqueries obtained by the reduction process can be reduced in cardinality by projection or selection. In this manner, the original query is replaced by a sequence of smaller queries. Figure 10.6 illustrates the decomposition of a query in the form of a tree.

The decomposition algorithm (Wong 76) consists of four subalgorithms refered to as reduction, subquery sequencing, tuple substitution, and variable selection. In the reduction subalgorithm, the query is separated into irreducible components. These are evaluated in an order determined by the subsequency subalgorithm. Each subquery is evaluated in order and the result of the evaluation is used in tuple substitution. Optimization is attempted by determining the sequence in which the subqueries are to be evaluated and selecting the variables for which the tuple substi-