This page is intentionally left blank.

Chapter 9

Database Implementation

Traveler, there is no path,
paths are made by walking.

Antonio Machado

Chapter 7 has presented models of databases; Chap. 8 discussed means to describe databases, and now we can finally look at actual database implementation problems. In this chapter we will proceed in a top-down fashion. We begin with methods derived from formal models and continue on to systems developed over time and regular use. The discussion in this chapter proceeds in the direction of increased binding, which causes loss of flexibility but increases the performance of systems. Such important implementation issues such as reliability, access protection, integrity, and data representation are not covered in this chapter; they are separately discussed in Chaps. 11, 12, 13, and 14.

We concentrate on concepts, and do not present complete system descriptions, although many actual systems will be referenced to allow further study. Appendix B can be consulted for references about the systems named in this chapter. The exact syntax of examples based on these systems has been modified at times in order to provide continuity. The fact that a certain system is cited here as a commendable example does not imply an endorsement of this implementation for a given application but only reflects on the values of the concepts being discussed.

Database Systems versus File Systems To review the necessity for a database system, its benefits and costs can be compared with the benefits and costs incurred when a file structure is superimposed on a hardware system.

A *file system* organizes the data-storage capability that is provided by the hardware. The hardware is partitioned into files, which are associated with a particular user. These users are now able to work in apparent isolation.

A *database system* organizes the file storage capability that is provided by the file systems. The relationships between the elements of the relations are made accessible. The data can be shared by cooperating users.

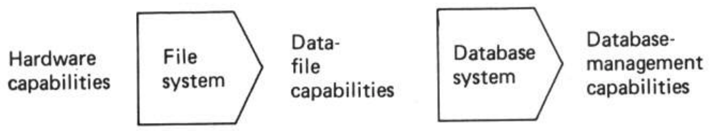


Figure 9-1 File and database system.

It is not essential that all secondary storage in a computer be managed through a file system. When multiple users share the computer, however, the more formal and complex approach has to be supported in order to enable these users to do productive work in isolation. The benefits of a well-organized file structure may already be a boon to a specific user; on the other hand, many respectable data files exist without any file-system support, specifically in small single-application computers.

It is not essential that storage of databases in a computer be controlled by means of a database system. When large amounts of interrelated data are stored that are of interest to diverse groups of users, a database system becomes necessary.

Objectives We have implied earlier a number of objectives for a database system design:

1 The ability to refer to data items without having knowledge of record or file structure and as a corollary:

2 The ability to change record or file content and structure without affecting existing database programs We also desire

3 The ability to handle related files within one general framework, so that the data in separate files can remain consistent and so that excessive redundancy in updating and storage can be avoided and

4 A description of the database integrating diverse points of view, so that this description can become a communication medium between data generators and information seekers

In order to achieve these lofty goals, we will consider how to implement systems that use schemas to present high-level services while using file-based, record-oriented structures.

9-1 ISSUES IN DATABASE IMPLEMENTATION

In this introductory section we will introduce briefly some concepts that recur throughout this chapter.

9-1-1 Functionality and Generality

Database-management systems can be built with a wide range of generality. A categorization of these approaches into three levels distinguishes systems which support a single application, several applications of the same type, or multiple types of applications. Some systems have developed through these three levels; others have been designed consciously to attack problems at one specific level.

Single-Application Database Systems An organization establishes a database operation using available file system facilities, and designs application programs that interface to the database using a centrally maintained package which implements the required degree of data and structure description.

The original airline reservation system at American Airlines, SABRE, many large information systems, such as MEDLARS (a system to query the medical literature), and military command and control systems are examples of this approach.

Single-Application-Type Database Systems A group of users working in some type of application area recognizes the commonality of their needs. They or their vendor design a system to match their needs. User differences are incorporated into tables and schemas specific to the user. This step often follows success with a more single-minded system.

Examples of this approach are the generalized airline reservation systems (PARS), clinical information systems (TOD, GEMISCH), and bills-of-materials systems (BOMP).

Multiple-Application-Type Database Systems A vendor or academic group designs a system with the intent to serve the general database needs in a better fashion. An effort is made to provide a complete set of services. There will, of course, be a tendency to emphasize aspects relating to the experience of the designers, so that in practice a great deal of difference is found among the generalized systems. Another source for generalized systems is a continued evolution from single application or application-type services. An understanding of the history of generalized systems helps to explain features of their design. The development of the CODASYL specification and of the relational model for databases has provided a basis for generalized systems that are relatively independent of past history. Generalized systems developed independently encountered so far in the text are PRTV, RETRIEVE, SOCRATE, and SQL/DS. The IDS and IMS systems owe much to the BOMP applications.

This chapter will consider mainly *generalized database management systems*. This is not intended to imply that more specific approaches are not valid. An orientation toward a specific application or type of application allows the recognition of semantic relationships which are difficult to exploit in a generalized system. A generalized system, however, presents a better balance of the problems of database system implementation. The term *database management system* (DBMS) is used here to refer to any of these approaches, and the adjective *generalized* is added only when the emphasis is required.

9-1-2 Models and Implementation Style

A database implementation carries a strong implication of the style of model the user is expected to have of the database. Since different classes of users will have different problems and different approaches to their use of the database, no single style of representation may be adequate for all users.

A major distinction can be made between information-retrieval usage and data-processing. Another, related, distinction is the predominant type of operational use of the system. We distinguish especially operations that retrieve simple items of data versus operations that require retrieval of data collections for further processing. We will present these issues in Chap. 10, but consider now styles of models and their effect on the implementation of the database.

The structural model defined in Sec. 7-3 used relations and connections. This model implies that data values can be obtainable by naming relations and attributes, and defining operations between them. Joins along connections have a predictable behavior.

We distinguish styles based on their stress on *relations versus connections*. We note first that a fully decomposed normalized model may contain many more relations than the user's personal model needs. The use of a submodel, which defines a specific and perhaps nonnormalized view, can present the database in a way that is more natural to some users.

A user's model which ignores the connections is called a *relational model*. The user will define explicitly all connections to be used in queries or updates.

To find a `child` of an `employee_name`, both the `Children` and the `Employee` relations have to be referenced, and the connection has to be defined in the query.

A *universal relation scheme* provides a view to the user of only a single relation. All the component attributes will have unique names. All connections are deduced from specification of the dependencies among attributes. The complexities of a normalized model are now hidden from the user, and the underlying database relations can be freely rearranged to support optimal processing strategies. Any processing that relates attributes by paths other than defined dependencies will still require explicit definitions, as the pure relational model does. If multiple paths between the attributes are possible, either user interaction or a set of predefined rules can be used to determine the best path.

The user's model can be simplified by exploiting the known connections, or relationships as defined in the structural model and in the general class of *entity-relationship* models. Now related attributes in connected relations can be found by following the connections. Intermediate relations can be ignored.

If the implementable or actual implemented connections are restricted to a tree, we can talk about a *hierarchical model*. These models tend to be easy to follow if they match the user's model, and the paths between attributes can be determined without ambiguity.

The parts in a subassembly (Fig. 4-23) were found via the connections defined by the hierarchy. A name with hierarchical qualifiers implies how the data element is accessed:

`machine.assembly.subassembly.part`.

Some problems remain for retrieval involving elements from more than two relations. We will present these with the hierarchical implementations in Sec. 9-4.

In a network with associations and multiply referenced entity relations in the model the connections become more complex. In those models the accessing style tends to require procedural descriptions. Two attributes in a network may be connected by more than one path. If there are multiple connections in a database model the appropriate path has to be found. The user can avoid ambiguity in a network by providing the full path definition in a query. An example of two paths based on Figs. 7-16 and 7-17 follows.

Example 9-1 Alternate Paths in a Network

We have the owner relations

```
Suppliers: RELATION                 Parts : RELATION
     s_id :⟩ name, etc.  ;              p_id :⟩name, size, weight, etc.  ;
```

and two associations owned by them

```
Supply: RELATION                    Possible_supplier : RELATION
     s_id, p_id :⟩ quantity ;           s_id, p_id :⟩ ;
```

Two possible query fragments to "Locate Source of Heavy Parts" are:

```
GET Parts.weight, Supplier.address      FOR Supply
GET Parts.weight, Supplier.address      FOR Possible_supplier
```

The two paths will lead to different result tuples.

A number of alternative techniques to retrieve data from a database are feasible. We list them in order of increased sophistication.

1 No path selection is attempted, and the user will move explicitly from one relation to the next, generally using the familiar and defined connections.

2 A database submodel is predefined to provide the desired view for the user, and that submodel selects one connection, determining the path.

3 All possible pathways are computed and presented to the user for selection.

4 Rules for selecting the *best* from the set of all possible pathways are applied by the system.

An implementation often does not define all the known connections, so that implemented networks are often simpler than the database model indicated.

The dual problem also exists in networks, when one data item may be found by more than one name. The two queries below, using a different path, will locate the same data item.

```
Location("Atlanta").dept("Assembly").job("Welder").name("Mike").age

Healthrecordno(13436).age
```

Alternate names for the same item can lead to problems in integrity protection. This issue is addressed in Chap. 13.

9-1-3 Self-Contained versus Host Language Services

The obvious role of a database-management system is retrieval from and update of the database. In the implementation of a complete system it is also necessary to support analysis capability on the retrieved data. There are two basic approaches to the incorporation of analysis capability into database systems, as illustrated in Fig. 9-2, namely, *self-contained* and *host-based*.

Self-Contained Systems The database system provides all the required services. There has to be at least some capability to execute computations on data retrieved from the database system. The RETRIEVE system (Figs. 8-5, 8-6, 8-7) and SOCRATE (Fig. 8-10) provided examples of self-contained systems.

Host-Based Systems The database system carries out the retrieval and update functions only, and delivers the data on request to programs written in a *host system language*. The IDS system (Fig. 8-12) and the DBTG specifications (Fig. 8-9) shown in Chap. 8 provide examples of the host-based approach.

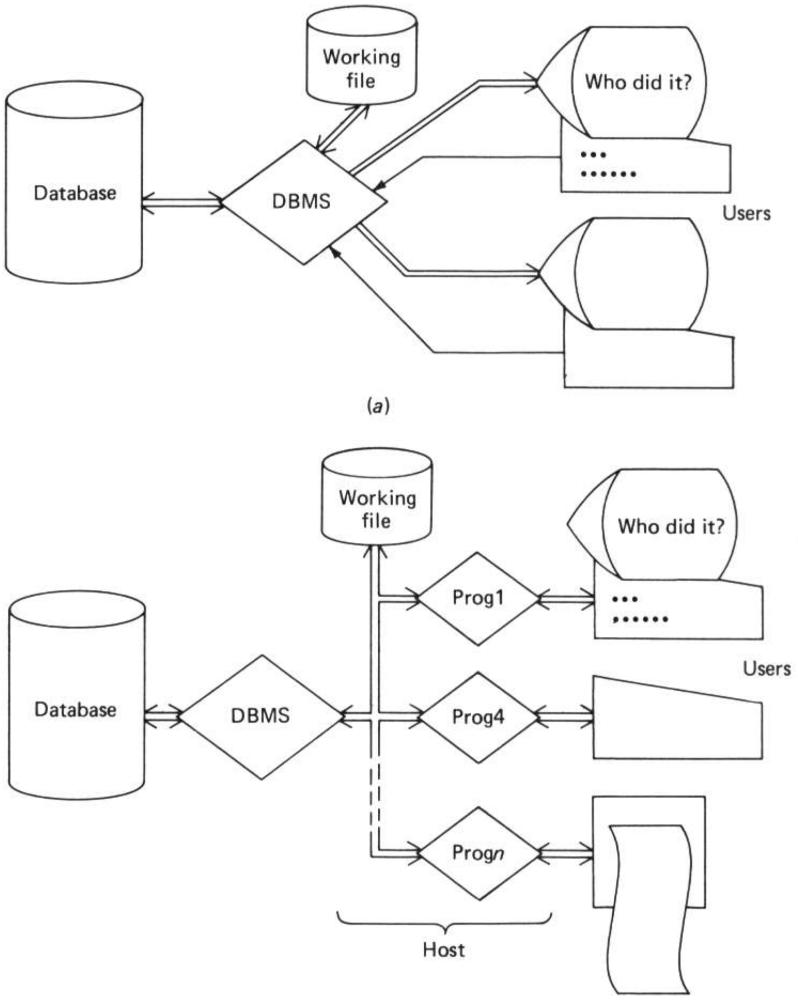


Figure 9-2 Database system alternatives.

It is possible to provide systems that combine approaches. In the TOD approach (Fig. 8-11) a host-based DBMS also provides programs which allow the use of preprogrammed search requests (i.e., Prog1 in Fig. 9-2). The user is prompted for parameters. Functions such as file updating (Prog$n$), which require more general processing capabilities, are written by programmers in the PL/1 language.

Having a host available for processing does not remove all concerns about processing from the DBMS. In any database system it is desirable to retrieve data in a form that matches the user's view of the data. The selection algorithms for data may become quite complex if a user's requirements for a relation involve selection from multiple files. Selection requests addressed to the DBMS may retrieve many tuples, but few programming languages can deal with more than one record at a time. The DBMS serving a host language has to be able to provide a set of tuples, one record at a time, in a form the language can understand.

The interface of a host system and DBMS can lead to problems and inefficiency. If the DBMS cannot select tuples based on powerful selection criteria, the DBMS will deliver many tuples to the host programs that the programs will reject, so that retrieval and communication effort will be wasted. The host programs do not know the internal schema of the DBMS, so the requests will not be submitted in an optimal sequence. Effective sharing of capabilities requires a very complex interface between host system and DBMS.

In a self-contained DBMS closer interaction between processing and database is possible, this allows greater efficiency. Queries can be rearranged to take advantage of the relative accessing effectiveness of the query attributes. Indexes and other auxiliary information available in the system can be used beneficially. It is difficult, however, to provide in a self-contained DBMS all the processing facilities available through conventional computing systems, such as auxiliary files, statistical library programs, simulation systems, and communication facilities to other users and their computers.

The expectations which the users bring to generalized DBMSs are increasing, so that fewer pure self-contained systems are being implemented. The availability of a competent host system provides access to many processing facilities. The hybrid approach, where the DBMS provides some packages for self-contained operations as well as access for user programs to carry out operations that are not provided as part of the DBMS, is becoming prevalent.

9-1-4 Information Hiding and Secrecy

The database content will be easier to adapt to changing conditions if the user's external schema can remain unchanged. For that reason, it may seem desirable that the actual database structure remains hidden from the user. This implies that the internal database schema is to be kept secret from the user of the external schema. Elements of the internal schema that may be of interest to a user are the data representation and access path information. The data representation indicates the permissible range of domain values in those systems where domains are not explicit. Knowledge of access paths can affect the processing of transactions where more than one computational sequence is feasible. The assumption that favors hiding is that lack of knowledge will prevent users from taking advantage of coding or access structure in order to optimize their programs or retrieval operations, and that hiding will avoid problems when the internal structure has to be changed.

Some problems, however, will develop when there is excessive emphasis on keeping the structure of the database secret. The mere fact that information is being kept from users will make them suspicious and will have a negative effect on the relations between users and the service organization. Such secrecy then becomes very costly in terms of human productivity. Exceptions to secrecy become necessary when some applications do not receive adequate performance levels, and this can create further conflicts between users who are more and those who are less privileged.

Differences in response time also reveal to the user differences in the access paths, and any human will take advantage of alternatives which have shown themselves to be profitable. A typical example is seen in a medical clinic where the search for the patient's record by number takes less than 3 seconds and a search by name requires nearly a minute. The clerks will demand the number from the patient and the system will be regarded as a typical example of dehumanization. The fact that the system designer told the users that the patient's name is just as good a retrieval key will be ignored.

A more comfortable solution is reached if the database system has a clean structure, provides means to extract any combination of data available in the system with an effort proportional to the result, and allows the users to determine if certain patterns of usage are advantageous or not. This implies some education about the systems, especially for frequent and intensive users of the services. The benefits of education are always two-sided, since the feedback from educated users provides the system designer with criteria for decision making. The effort expended by the user in the demystification process is probably less than the effort spent in ferreting secrets and half truths out of the black systems box. Information hiding remains a valid concept if it is used to design clean interfaces among system components. The approaches favored here use clean models, with understandable semantics and documented trade-offs in performance issues.

9-2 RELATIONAL CALCULUS IMPLEMENTATION

An elegant approach to manipulation of a database is to use the concepts of relations and transformations of relations presented when modeling databases in Chap. 7. In those implementations the relations are implemented by files, the schema is used to name the relations and their attributes, and the operations are specified implicitly by defining result relations.

A result relation is specified by a formula which expresses the desired result in terms of the relations in the database. A formula in the relational calculus has the general form

$$\textit{Result relation} = \{\ \textit{tuple definition} \parallel \textit{conditions}\ \}$$

The attributes used to define tuples and test conditions come from the database relations. We note that the process of arriving at a result is not specified. We have here a language that is not procedural, like a programming language, but is result- or problem-oriented.

9-2-1 A Relational Calculus System

We will begin by presenting aspects of a language, SQL, used by several implementations of the relational calculus, and will then discuss some features of similar systems as well as some implementation issues. We focus on retrieval and present the principal command in Table 9-1, followed by examples of its use.

The SELECT command presents a result table based on attributes from the database tables listed in the FROM clause. The WHERE clause restricts the result to rows meeting certain conditions. Aggregation during selection and other commands will be touched on later. The notation follows Fig. 8-10.

Table 9-1 The Retrieval Command of SQL/DS

```
                    *                                   /* * means all attributes */
  SELECT {  attribute_ac [ ,... ] }
     FROM table_name [tuple_variable] [, ... [  ... ]]
     [ WHERE selection_expression ]
     [ GROUP BY attribute [HAVING selection_expression], ... ]
     [ ORDER BY attribute [DESCending], ...]
/* The principal primitive component of the Select command is: */
       attribute        /* must be listed in the schema of some FROM table */
       attribute_ac    /* is a simple attribute or
                      an arithmetic expression of attributes and constants. */
       /* tuple_variables are presented in Sec. 9-2-2. */
  Selection_expression::= /* a boolean expression using attribute_expressions:*/
       [NOT] selection_expression [[NOT]{AND / OR} selection_expression]
       ( selection_expression )
       attribute = USER             /* user id, good for checking a VIEW  */
       attribute_expression
  Attribute_expression::=  /* a simple or a complex conditional expression: */
       attribute_ac ⊖ constant
                 /* ⊖ is one of the set {=  ¬=  >  >=  <  <=} */
       attribute_ac ⊖ attribute_ac
       attribute_ac BETWEEN low_attribute_ac AND high_attribute_ac
       attribute_ac [NOT] IN (constant_1, ..., constant_n)
       attribute ⊖ {ANY / ALL} (constant_1, ..., constant_n)
       attribute IS [NOT] NULL
       attribute [NOT] LIKE "search_string"   /* See note in Table 9-3 */
/* Attribute_expressions can include other SELECT substatements: */
     /* If the subquery leads to a single value: */
       attribute_ac ⊖ ( SELECT ... FROM ... ... )
     /* If the subquery can lead to a set of values: */
       attribute_ac [NOT] IN ( SELECT ... FROM ... ... )
       attribute_ac ⊖ { ANY / ALL } ( SELECT ... FROM ... ... )
     /* If the result of the subquery is to be quantified (see Sec. 9-2-3): */
       [NOT] EXISTS ( SELECT ... FROM ... ... )
```

The expressions found in the formulas are translated into sequences of operations to be carried out on the database. The translator we refer to is the IBM SQL/DS system. A translator will rearrange the execution steps of the expression in order to minimize the total time required for CPU usage and disk accesses. Since block accesses are the principal cost factor, the optimization of the query will concentrate on this aspect. If indexes are available, these will be used whenever it seems profitable.

In the SQL documentation the term for a file is `table` and for a record is `row`. A table may contain duplicate identical rows, and is hence not a relation according to the stricter definition used in modeling. Example 9-3 clarifies this distinction. The available data types are two sizes of integers, decimal, floating point, and three types of strings: fixed, limited varying ($n < 255$), and long varying($< 32\,768$).

Any `attribute` used in a SQL expression must have been defined in the schema of a `FROM` table. If multiple rows of the same table appear in a statement, a *tuple variable* has to be specified. Tuple variables are presented in Sec. 9-2-2.

Example 9-2 Relational Calculus Statement

A simple example is a query for rows of one table based on one attribute value. Given

```
Employee:  RELATION
     name :⟩ birthdate, height, weight, job, dep_no, health_no ;
```

the statement to create a table `Welders` with their names and birthdates

```
     SELECT name, birthdate  FROM Employee  WHERE job = "Welder";
```

when applied to the relation instance shown in Fig. 7-6, defines the result table

```
Welders:  RELATION
     name,  birthdate;
     Gerbil 1938
     Havic  1938
```

Joins in the relational calculus are implied. When more than one relation is specified in the `FROM table_name, ...` list, the attributes in the `SELECT attribute` list may come from any of the relations specified, requiring a join. A clause such as `WHERE one_table.attribute_1 = another_table.attribute_2` specifies the join condition; without such a condition a Cartesian product of the relations in the `attribute_r` list is implied. The "=" here implies execution of an equijoin during the processing of the query. Other joins are possible depending on $\ominus$. The subquery (`SELECT ...`) expression may have its own `WHERE` clauses, to restrict the rows which participate in the join.

Other commands for retrieval in SQL/DS are `PRINT` and `FORMAT` commands for report generation. A `GROUP` clause causes a result table to be broken into groups. For instance, `GROUP BY school_name` would list the `Children` school by school. In `PRINT`ed reports `FORMAT` options can be used to generate a summary row with each group, and this row can contain `COUNT`s, `SUBTOTAL`s, and the like.

Tables versus Relations In the description of SQL we already found a number of instances where the basic concepts of a calculus have been extended to provide more suitable or effective facilities. An important feature is that SELECTed TABLEs are not treated as pure sets; they are permitted to include duplicate tuples. We follow SQL and use the term *table* to refer to such relations; sets of this type are also called *multi-sets* or *bags*. The alternatives, table versus relation, lead to the existence in SQL of two types of SELECT. The default version considers ALL the rows of a table independently and the other permits treatment of a table as a set of DISTINCT rows, with any identical rows removed.

Example 9-3 Tables versus Relations

A simple example is a query for rows applied to the stored table shown in Fig. 7-6, the WHERE clause is based on one attribute value:

```
Age_of_all_welders:
     SELECT birthdate, job  FROM Employee  WHERE job LIKE "%Welder%"
                   /* Note: the % symbol stands for an arbitrary substring
                          to ensure that all kinds of 'Welders' will be found. */
                   /* In a SQL search_string: a _ denotes any single character,
                                            a % denotes any substring.    */
```

This statement defines the table:

```
 All_welders:  RELATION
      birthdate, job ;
     1938       Welder
     1947       Asst. Welder
     1938       Welder
```

The alternative type of query states:

```
Age_of_welders:
    SELECT DISTINCT birthdate, job FROM Employee
                                  WHERE job LIKE "%Welder%"
```

and, when applied to the same data, defines a true relation with

```
 Distinct_welders:  RELATION
       birthdate, job ;
     1938       Welder
     1947       Asst. Welder
```

Aggregation Instead of producing a table to be displayed, it is also possible to apply functions to summarize the resulting table. The final result is a single row or several rows each representing a group.

In this form of a SELECT, if a GROUP BY clause is used, the functions will be applied group by group. A group is defined as the set of rows having the same value in the specified attribute column. For instance, GROUP BY school_name would apply the functions to the Children rows school by school. The HAVING clause can be used to eliminate summary rows and it can use the functions as well. Only functions and the grouping attributes, here school_name, can appear in the function lists of Table 9-2.

The SELECT command as used for summarization is shown in Table 9-2. This type of SELECT can also be used to define subqueries within another SELECT statement.

Table 9-2 Use of Aggregation Functions

```
   SELECT function [, ...] FROM ...
              [GROUP BY ... ] [HAVING function ⊖ ... [, ...]  ]
              ...
available functions include:
        COUNT (*)                          /* count the table rows */
        COUNT (DISTINCT attributes)        /* project and count the rows */
        MAX (attribute)
        MIN (attribute)
        SUM (DISTINCT attribute)     /* SUM over projected result rows */
        SUM (ALL attribute)          /* SUM over all result rows */
        AVG (DISTINCT attribute)     /* AVeraGe over projected result rows */
        AVG (ALL attribute)          /* AVeraGe over all result rows */
     /* AVG excludes NULLs. There is unfortunately no Standard_Deviation */
```

To COUNT rows in SQL a table as All_welders has first to be defined in a WHERE SELECT clause. Then the count of all table instances or the distinct types can be obtained. For the All_welders table of Example 9-3 the two alternatives are:

```
COUNT (*)                    FROM All_welders    = 3
COUNT (DISTINCT birthdate)   FROM All_welders    = 2
```

The alternative of tables versus relations supports the *token versus type* choice often required in data analysis, e.g. Sec. 14-3-2, and query design. The average number of token instances of a given type for some attribute Rel.att is

```
token_type_ratio = COUNT (*) FROM Rel / COUNT (Distinct Rel.att)...
```

A high token_type_ratio for an attribute indicates a poor partitioning effectiveness. The ratio is 1 for unique attributes.

Updating The update commands are similar in syntax to the SELECT command, but affect the stored tables. More care is required here to assure that the database is correctly maintained. The update commands include: INSERT, UPDATE, DELETE.

An INSERT INTO table VALUES (list) command only inserts one row, with the listed attributes values, so that they match the position of the columns in the TABLE or VIEW definition given for the relation. Fields of the row missing from the list are set to NULL. Several clauses shown with the SELECT command are also available here.

An UPDATE table SET attribute = constant,...[WHERE ...] command affects all rows satisfying its WHERE clause; it reports the number of rows changed. A DELETE FROM table ... WHERE ... statement deletes all rows satisfying the WHERE clause.

All types of update have to be severely restricted on views (see Sec. 8-5-1). A view which was created with a join in its `SELECT` clause cannot be updated. The view definition does provide a convenient way to define transactions, since a view can incorporate an arbitrarily complex `SELECT` clause, exclusive of any `ORDER` or `GROUP BY` phrases. A privileged user, having DBA status, can `GRANT` or withdraw updating and other privileges to any user for any table, view, or specific attribute.

An `EXTRACT` statement permits the insertion of many rows from an existing DL/1 file, as defined in Sec. 8-3-2. The extracted data is redundant, however, and no identity connection (Sec. 7-5-1) is maintained. Tables created this way are typically only queried, but not updated.

9-2-2 Tuple Variables

The attributes used in the statements belong to a specific table named in the `FROM` clause. Specific data elements used, especially during comparisons within `WHERE` clauses, also belong to specific row instances. In many situations the proper instance in the proper table can be found without any ambiguity. There are cases, however, where a precise definition is needed, for instance, if a relation is joined with itself.

The use of *tuple variables* with the attributes provides an unambiguous notation for attributes and value instances. A tuple variable is a free variable with a domain which is the identification of the tuples of a relation. Some relational languages, for instance, INGRES' QUEL, consistently demand that all attributes be qualified with a tuple variable to designate a specific instance in a specific relation.

Example 9-4 Use of Tuple Variables to Distinguish Attributes

We wish to produce a result table with names and departments for all employees with school age children. Given tables based on the employees and their children from Figs. 7-6 and 7-7

```
Employee:  RELATION                 Children :  RELATION
     name :⟩ age, height, ...  ;        father, child :⟩ age;
```

but with an identical attribute name `age`. We let one tuple variable `kid` range over the `Children` relation and another one, `emp`, identify tuples in the `Employee` relation.

```
SELECT emp.name, emp.age
   FROM  Children kid, Employee emp
   WHERE kid.father = emp.name  ∧kid.age >= 6
```

For each `emp.name` all `kid.father` values will be investigated. When there is any match (here once for `Gerbil`, twice for `Hare`), the second condition, `kid.age > 6` is tested. Only the tuple

```
Hare 38
```

fulfills both requirements. Since two of `Hare`'s children are older, the tuple will appear twice in an SQL result table.

In SQL the case shown in Example 9-4 does not actually require an explicit specification of tuple variables; the `table_name` can be used as the tuple variable so that `Employee.age` and `Children.age` can be used to distinguish between attributes having identical names.

In SQL the use of explicitly defined tuple variables is needed only to avoid ambiguity. This requirement often occurs with subqueries. For instance, when a `SELECT` subquery in a `WHERE` clause has to refer to the same row as another `SELECT` clause, the `tuple_variables` must be defined with their table names in the `FROM ...` clause; the syntax is indicated in Table 9-1. The `attribute` is then written as `tuple_variable.attribute`.

Queries with two or more distinct rows instances of one relation cannot be formulated without the use of tuple variables. Example 9-5 provides an example with a query where the `boss` and the `subordinate` are distinct entries in the same `Employee_2` relation.

Example 9-5 Use of Tuple Variables to Distinguish Tuples

In order to form a table which contains all the employees who supervise other employees it is necessary to look at the Employee_2 row for the boss and the Employee_2 row for the subordinate at the same time.

The result table is to contain both names and the years of supervision. Two tuple variables range independently over the employee table (see Fig. 7-10 for the instance), so that given

```
Employee_2:  RELATION               Supervision :  RELATION
     name :⟩ age, ..., experience;     super, sub :⟩ years;

SELECT boss.name, subord.name, superv.years
        FROM  Employee_2 boss, Employee_2 subord, Supervision superv
        WHERE boss.name  = superv.super ∧
                 superv.sub = subord.name;
```

In SQL the tuple variable `superv` is not required.

9-2-3 Quantification

In the examples up to now the `WHERE` clause contained an `attribute_expression`. The resulting boolean `True` or `False` value is used to control the result of `SELECT` expressions. The boolean for the `WHERE` clause can also be obtained by asking about the result of a subquery in a quantitative sense. We distinguish two types, referred to as *existential* and *universal quantification*.

Existential quantification is obtained through use of the `EXISTS` clause in SQL. The `EXISTS ( SELECT * ...)` clause is satisfied whenever there is *any* match in the `SELECT * ...` subquery. Universal quantification is true only if *all* instances are true. Universal quantification is not directly available in SQL but in most cases can be obtained by using `NOT EXISTS` applied to the complement of the condition.

The use in both types of quantification is shown in Example 9-6. Note that use of `EXISTS` produces a `True-or-False` condition for a row identified with a tuple variable, while there was a join implied in Example 9-4.

Example 9-6 Use of Quantification

We use again the employees and their children from Figs. 7-6 and 7-7. We first find employees with at least one child of school age and then employees where all children are of school age.

```
 Employee:  RELATION                      Children :  RELATION
      name :⟩ age, height, ...  ;          father, child :⟩ age ;
```

```
  SELECT emp.name FROM Employee emp
     WHERE EXISTS ( SELECT *  FROM Children kids
                    WHERE emp.name = kids.father AND kids.age >= 6 );
```

To find employees that do not have school age children use `NOT EXISTS`:

```
  SELECT emp.name FROM Employee emp
     WHERE NOT EXISTS ( SELECT *  FROM Children kids
                        WHERE emp.name = kids.father AND kids.age >= 6 );
```

To find employees where all the children are of school age, we transform the query from universal to existential quantification by using the complement:

```
  SELECT emp.name FROM Employee emp
    WHERE NOT EXISTS ( SELECT *  FROM Children kids
                       WHERE emp.name = kids.father AND kids.age >= 6 );
```

There are queries involving universal quantification which cannot be reasonably transformed into existential form. Such cases occur when a condition has to hold for all members of a group, and that condition is to be computed within the group. An example is a query to locate suppliers capable of manufacturing subassemblies using only their own parts, formulated later using the relational algebra in Examples 9-12 and 9-13.

Requests of this form can generate a collection of tuples which describe subsets of a relation, and hence provide a link to the hierarchical presentation of data.

The composition of statements with quantification can become quite difficult, and expressions with universal quantification are avoided in most current systems. Some uses for universal quantification are satisfied by the `GROUP BY` statement. Programmed access may be used to collect the required information and perform further analysis on the retrieved data, including the checking for some condition over all instances in a subgroup.

9-2-4 Management of the Database

In order to use `SQL`, a database has to be designed and described in a schema, space for the database has to be acquired, and access structures have to be specified to satisfy performance demands. We only list some of the relevant commands. Usage of `SQL/DS` beyond simple on-line retrieval and update requires knowledge of these statements.

SQL includes, of course, the commands to define the schema: `CREATE TABLE`, `DROP TABLE`, and `ALTER TABLE`. You can `CREATE INDEX` or `DROP INDEX`. To manage external views use `CREATE VIEW` with subschema definition commands as shown in Example 8-11 or `DROP VIEW`. All these can be issued at any time, and SQL/DS will recompile or rebind any routines which have been invalidated.

Repeated use of a sequence of SQL statements is made possible by collecting the statements themselves into a `TABLE` for recall and reuse. To manage an interactive session, some additional commands are available; they are shown in Table 9-3.

Table 9-3 Controlling a Transaction

CONNECT	identify and assign the privileges available to the user; disconnects a prior user.
COMMIT WORK	indicate the end of a logical transaction
ROLLBACK WORK	remove all changes made since the last COMMIT or CONNECT

Programming Access We have presented SQL in the preceding sections as seen by a user executing on-line requests. If more general computations are needed than single SQL commands conveniently provide, or if there is a large volume of updates, then access by a program is needed.

A type of computation not possible within the relational calculus is a *t*ransitive search through an undetermined number of relations. An example of such a query is `"Find the top-level manager for an employee"`. Here the iteration of joins of the attributes `Employee.name` and `Employee.manager` terminates only when `Employee.manager = null`.

Five steps provide programmed access to relational calculus operations:

1 Variables to match the attributes used are declared in the programs.

2 SQL `SELECT` commands, augmented with `INTO variable` clauses, are inserted into the program.

3 A cursor is declared for every `SELECT` which might retrieve more than one row.

4 A parameter area is defined to receive condition and error codes; we encountered such areas with VSAM programming in Sec. 4-3-5.

5 The program is completed with computational and control statements.

A user program containing `SQL` declarations and `INTO` commands is processed through a preprocessor, which translates them into `CALL` statements to `SQL` routines kept on a library and declarations acceptable to the host language, similar to the sequence shown in Fig. 8-12. After compilation and loading, the programs can be executed and access the `SQL` database.

The programs in the SQL/DS library are associated with the version and status of the database. A program which tries to access a database whose access structure is changed, will be automatically recompiled. Since these programs are independent of access paths, no programmer intervention is needed.

The cursor used to identify individual rows from a `SELECT` table is manipulated by `OPEN, FETCH, DELETE, UPDATE,` and `CLOSE` commands. Cursors perform functions similar to tuple variables on the result table of the `SELECT` commands in the programs.

The `OPEN` statement initializes the execution of the `SELECT` command and identifies a cursor. A subsequent `FETCH` retrieves one row as specified, places the values into the designated variables, and sets the cursor. If no row is fetched a condition code is set; otherwise another `FETCH` can be executed to attempt to retrieve data from a row not yet fetched via the cursor. `DELETE` and `UPDATE` commands can use the same cursor in a `WHERE` clause. The `CLOSE` command makes the cursor unavailable.

9-2-5 Extensions of the Relational Calculus

We will present two facilities which can improve the usability of these systems. *Workspaces* provide the capability to use temporary relations for intermediate results and *integrity assertion statements* permit the definition of constraints.

Workspaces A user of the relational calculus may find it easier to define intermediate relations using simple statements, and then define further conditions on relations previously defined. In order to allow manipulations to be specified as a number of steps, systems which support the relational calculus as a user language provide for the definition of intermediate relations. These relations are kept in *workspaces*. Workspaces have the following features:

1 They are not part of the database itself, so that there will be no inconsistency due to the creation of redundant data.

2 The workspaces belong to one individual user or process, so that the manipulations of data within such workspaces do not conflict with other users.

3 A naming scheme is employed to distinguish relations and their attributes within the workspace from the relations and attributes in the database.

4 Workspaces may also be used to collect input data prior to update of the database relations.

In order to illustrate the use of workspaces we will augment the SQL statement described with a statement from SEQUEL, a predecessor of SQL. The request for supervisors from Example 9-5 can now be written as shown in Example 9-7.

Since SQL permits dynamic definition and creation of tables, specific workspace statements are not part of the language. In a host-based DBMS general array and file facilties may serve as workspaces. In self-contained implementations of a relational calculus, for example, INGRES, workspaces are important to satisfy complex general processing needs. Workspaces may also be used to store derived attributes and tuples.

Example 9-7 Use of Workspaces

We use again the employee relations from Fig. 7-10:

```
Employee_2:  RELATION                 Supervision :  RELATION
     name :⟩ age, ..., experience;      super, sub :⟩ years;
```

```
Bosses(b_name, b_age)  = SELECT  name, age FROM Employee_2, Supervision
                         WHERE name = super;
Workers(w_super, w_age) = SELECT  super, age FROM Employee_2, Supervision
                         WHERE  name = sub;
SELECT b_name FROM Bosses, Workers
               WHERE b_name = w_super;
```

Example 9-8 Derived Values in Workspaces

An operation which can create a new attribute would be the calculation of the weight in stock for the various parts, given the Supply and Parts relations of Fig. 7-16. We have to use tuple variables to distinguish the attributes.

```
Parts: RELATION                          Supply : RELATION
    p_id :⟩ name,size,weight,...;            s_id, p_id :⟩ quantity ;
```

```
Stockweight(p_id, pounds) = SELECT part.p_id, sup.quantity * part.weight
                            FROM  Parts part, Supply sup
                            WHERE part.p_id = sup.p_id;
```

The relation Stockweight contains as many tuples as the Supply relation (7) since the join implied here of Parts and Supply uses an ownership connection from Parts to Supply in the model. Some part numbers are repeated since they came from different Suppliers.

Workspaces are useful when the result relations are further processed. For instance, to obtain a commercial-looking output a "Total" tuple should be appended to the list created above. Example 9-9 uses a SEQUEL statement, since SQL does not permit the same table to be used as the destination and source of an INSERT.

Example 9-9 Inserting a SUM into the Workspace

We will augment this workspace with a single tuple for the total weight. Using the function SUM and the INSERT INTO operation from SEQUEL:

```
INSERT INTO Stockweight(p_id, pounds)
            : ("Total", SUM( SELECT pounds FROM Stockweight) );
```

Supplementary capabilities have been proposed for the relational sublanguage SQUARE. SQUARE as defined uses a two-dimensional notation to avoid workspaces and tuple variables. To demonstrate the features of SQUARE in the notation used here, a relation is created which contains the total weight of each part type.

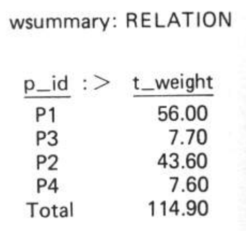
Example 9-10 Inserting SUMs of Groups into the Workspace

```
Stockweight(p_id, pounds) <-
  SELECT DISTINCT part.p_id, SUM ( SELECT sup.quantity * part_grp.weight
                                   WHERE part.p_id = sup.p_id      )
     FROM  Parts as part, Supply as sup ;
```

Now only one tuple per p_id is left. We can still insert a total tuple; the result relation is shown in Fig. 9-3.

Stockweight: RELATION

p_id,	pounds;
P1	58.00
P2	7.70
P3	43.00
P4	7.60
Total	114.90



wsummary: RELATION

p_id :>	t_weight
P1	56.00
P3	7.70
P2	43.60
P4	7.60
Total	114.90

Figure 9-3 Subtotal relation.

Integrity Constraints In order to maintain a database correctly, the constraints implied by the connections defined in the database model still have to be imposed.

For instance, without the maintenance of the ownership connection among the relations we might not be sure that all `p_ids` in `Supply` were listed in `Parts`. If any `p_ids` are missing from `Parts`, the join caused by the `WHERE` clause of Example 9-8 will fail to include those parts in the result and in the final `pounds` calculations.

In most relational systems such constraints are the responsibility of the user; a basic relational schema does not provide for specification of update constraints. Two alternatives have to be considered, maintenance of integrity during update or consideration of lack of integrity during retrieval. Even if most update commands are executed with care, and include `WHERE` clauses to assure that constraints are obeyed, no guarantee can be given by the system that there are no inconsistencies. This means that queries have to formulated with great care. There is, for instance, no guarantee that attributes along a connection will match, and hence a chance that a join will fail to retrieve expected data.

In Example 9-3 we wrote the `WHERE` clause permitting arbitrary substrings:

```
        job LIKE "%Welder%"
```

to ensure that the query will also retrieve `Hare`, the `Asst. Welder`. We found that the prior query in Example 9-2, using `LIKE "Welder"` for an exact match along the presumed reference connection, did not retrieve that record. In this case the reference connection to the table defining the domain of `job`s was not maintained for this attribute.

Such maintenance constraints have to be added explicitly to update commands or queries have to be written to take care of approximate matches to reduce the chance of missing records that should be connected.

Integrity Constraint Assertions The INGRES system, from the University of California, Berkeley, allows constraints to be specified with the database description. Integrity constraints are provided as assertions and will be kept with the schema. They do not affect the structure of the database. At execution time the assertion statements are merged, if relevant, with the queries or update statements and interpreted as if they were additional `WHERE` clauses. Exploitation of the constraints to simplify queries may be done by a user who is sure that they have been applied throughout.

Some constraints applicable to Example 9-8 are shown in Example 9-11. For the `COUNT` functions introduced in Table 9-2 we use a token (`_ALL`) and type (`_DISTINCT`) notation which is symmetric and closer to INGRES.

Example 9-11 Integrity Constraint Statements

```
/* Field limit */
    INTEGRITY Parts.weight > 0;
/* Reference */
    INTEGRITY Employee.dept = Departments.name
/* Unique key: tokens = types */
    INTEGRITY COUNT_ALL(Parts.p_id) = COUNT_DISTINCT(Parts.pid);
/* Complete ownership: owned types = owner tokens */
    INTEGRITY COUNT_DISTINCT(Supply.p_id) = COUNT_ALL(Parts.p_id);
```

9-2-6 File Support

Many relational implementations use simple file structures. When the mapping of relational tables to files is one-to-one, the file records will have a fixed number of fields. If the fields are also restricted to be of fixed length, the records will also be of fixed length. If data values can be variable, or if more complex mapping from relations to files is permitted, the demands on the supporting files increase.

For instance, one SQL row is stored as one record, but SQL/DS does permit variable-length strings, up to 32 767 characters, and hence needs variable-length-record support. This support is provided by VSAM (Sec. 4-3), and the index structure of VSAM also provides the maintenance for indexed attributes.

Statistical descriptive information is maintained by SQL, so that a decision can be made whether to use indexes or sorting to perform the join. Estimates about the expected partitioning effectiveness of selection clauses on attributes, or *selectivity*, provide information used to reorder the primitive operations into which a complex query may be decomposed. We consider this aspect in Sec. 9-2-7.

To provide the capability for recovery of a transaction, log files may be specified to collect all inputs and affected records. These logs are also VSAM files. The principles and the use of logs are presented in Secs. 11-3 and 11-4. The existence of a log permits the `ROLLBACK` operation mentioned in Table 9-3.

The allocation and extension of space for an SQL database is handled by a set of commands which direct VSAM. Multiple tables can share a single file space. The information is itself collected in SQL TABLES and used during operation of the database, so that the user does not have to be concerned about the underlying file structure.

The INGRES implementation cited in Secs. 9-2-2 and 9-2-5 provides four alternative file types to the system implementor. All records are of fixed length. For each relation to be implemented the file type may be either a `HEAP` – similar to a pile, a `HEAPSORT` – similar to sequential file, `ISAM` – an indexed sequential file, or `HASH` – a direct file. The file implementation choice determines if the stored data is kept as a table (`HEAP`) or as a relation (`HEAPSORT, ISAM, HASH`), according to the definitions shown in Example 9-3.

The ORACLE system, which uses SEQUEL, the query language used in the development system which preceded SQL/DS, implements its files using a hierarchical organization, similar to DL/1 files.

9-2-7 The Execution of Calculus Statements

In order to execute the statements of the relational calculus, a transformation to a sequence of relational operations is required. The operations are essentially those described in Sec. 7-4, perhaps modified to deal with tables instead of relations. The parsing of the calculus statements is similar to the problem faced by compilers, with the additional consideration that reordering of clauses is possible and sometimes necessary. Reordering is also part of the compiling process for other nonprocedural languages which describe relationships, and is seen in simulation languages.

The execution of the statements requires a capable and sophisticated DBMS if adequate efficiency is to be obtained. A high degree of optimization is possible in the relational calculus by rearranging the clauses into an optimal sequence. The general rule is to first reduce the number of records to be accessed by exploiting indexes for attributes appearing in restricting `WHERE` clauses. When multiple *restriction* clauses are available and have indexes, the clause with the greatest expected partitioning effectiveness should be chosen first. The partitioning effectiveness of an attribute could be kept in a schema entry using a suitable encoding. A practical problem is that different values within one attribute have greatly differing power.

If, for instance, we wish to find personnel in a relation identified by `p` for a construction task in Alaska,

```
    (p.name,p.dep_no) WHERE (p.sex="Male" ∧ p.experience>10)
```

then the predicate `experience>10` should be evaluated first, since we can expect that this will leave a smaller intermediate relation than predicate `sex="Male"`.

In the case above, the value `Female` applied to construction workers will in all probability retrieve a much smaller intermediate result than the `WHERE` clause does now.

Intermediate results are often not materialized into intermediate relations. It is often more effective to process result tuples further, and so avoid repetitive storing and retrieval of intermediate values. Whenever possible the TIDs of the database records will be manipulated, since a large number of TIDs can be kept in memory.

When a record containing the tuple is located, all required attributes are projected to reduce the tuple size but avoid any further record access. Cross-product and join operations, which have the potential to generate large intermediate results, are typically scheduled last. If the number of tuples satisfying the join condition or *join selectivity* can be estimated, however, and appears low, joins may be performed earlier. Along a connection a join result will never be greater than the larger relation.

The eventual choice of execution sequence for a query is determined by minimization of the total cost of all operations required to carry out the query. In a transaction with multiple embedded queries yet better performance is possible by an overall optimization. Within a transaction the queries may refer to different aspects of a similar subset of data, so that a single retrieval from the database files can serve multiple queries.

An airline transaction may first retrieve the times of the flights between two points, and then, conditionally, the seat availability and the cost of the flight. A single retrieval can obtain all potentially relevant data at a low incremental cost and can greatly reduce the average transaction cost.

The overall effect of query optimization in a relational system will depend greatly on the variability of the access patterns. It is difficult to beat a system with defined connections along a query path, as we encountered in Sec. 8-3, when the query matches the defined physical structure and the database is reasonably large. For unexpected queries, however, the optimization will provide much better performance than a system designed to address a different pattern will provide.

The cost of the relational operations ($\bowtie$, $\prod$, etc.) can vary a great deal from simplistic approaches to methods that exploit optimal algorithms and locality. We will discuss these in the next section with the relational algebras.

9-3 RELATIONAL ALGEBRA IMPLEMENTATION

We encountered in Sec. 7-4 the basic operations of the relational algebra:

Union, intersection, and difference of matching relations, ($\bigcup, \bigcap, -$)
Projection by domains ($\prod$)
Selection of tuples ($\underline{\subseteq}$)
Join of arbitrary relations ($\bowtie$)

These operations, together with the comparison and boolean operators used in the qualification statements of the relational calculus, provide the tools for systems based on the relational algebra.

These systems may be used to support a relational calculus or can be made directly available to users. Their nature is inherently procedural and these systems are comparable with systems using multiple unlinked nonhierarchical files and conventional data-processing operations. An important early implementation based on a relational algebra is the Prototype Relational Test Vehicle (PRTV). We encountered it in Sec. 7-4-1 when discussing the relational difference operation. The systems seen today range from small to large, from microcomputers to large systems for multiple users. An early APL implementation of a relational algebra was limited to numeric data values and had no schema; attributes were referenced using column indexes. Some systems based on the relational algebra, RDMS, which will be used for our first examples, are in routine data-processing use at MIT.

9-3-1 Relational Manipulations

In the relational algebraic systems the computations are specified and carried out statement by statement. There will be a greater use of workspaces and there is no need for tuple variables. Selection is generally made available through `WHERE` clauses, but these will apply only to attributes of the referenced relation.

The syntax differs greatly among these systems; the functions are nearly the same. Some systems implement relations only as sets with distinct tuples; others permit more general tables. Some of the more competent systems permit complex expressions of relational primitives, and then may try to minimize record accesses.

The query for employees of Example 9-4 to locate parents with children of school age can be presented to RDMS as follows:

```
Fathers(name) = PROJECT(Children WHERE age_c ¿= 6 BY father);
Output(name,dep_no) = COMPOSE(PROJECT(Employee BY(name,dep_no)),Fathers)
```

`COMPOSE(r1,r2)`	is a natural join operation using matching attribute names	
`PROJECT(r BY a)`	specifies projection	$\prod$ ␣`r.a`

Additional relational and aggregation operators are available. The statements can be part of programs to be executed together or can be used as commands entered on a terminal and executed immediately. RDMS also provides facilities for self-contained use of the database through the use of inquiry packages and report generators.

Other operations seen in relational algebras are also provided. We need union, intersection, difference, and cartesian product:

```
Union_result         =  Rel_1 UNION Rel_2
Intersection_result  =  Rel_1 INTER Rel_2
Difference_result    =  Rel_1 DIFF  Rel_2
Cross_product_result =  Rel_1 XPROD Rel_2
```

and a generalized join, of which the equi-join is one form,

```
Join_result = JOIN[(⊖)] Rel_1 ON attr_1 WITH Rel_2 ON attr_2
```

‡‡‡‡‡‡ A final operation, *divide* performs function which in the relational calculus normally requires use of universal quantification. Division generates a quotient relation with one tuple for every group of tuples in the dividend which completely matches tuples of the divisor. We will write division

```
Quotient_result = DIVIDE Dividend ON attr_1 BY Divisor ON attr_2
```

where attr_1 and attr_2 define the matching attribute domains of dividend and divisor that control the division. The attributes in the quotient consist of the complement of the matching attributes of the divisor, so that

```
quot_attr = ¬(Dividend.attr_1 )
```

Hence, this quotient is a projection of the dividend by quot_attr, containing only the tuple types whose associated tuple tokens have all instances in dividend and divisor matched for the controlling attr_1 and attr_2 domains. An example, effectively phrasing a problem requiring universal quantification, can demonstrate the use of a relational division:

Example 9-12 Use of DIVIDE

Given the Possible_supplier and Parts_skill required relations of Figs. 7-17 and 7-23, we wish to determine which Possible_supplier.s_id can deliver ALL parts for which (assembly, type). The relation schemas are:

```
 Possible_supplier:  RELATION       Parts_skill_required :  RELATION
     s_id, p_id :⟩  ;                  assembly, type, p_id :⟩ no_req, ...;
```

We first select the tuples for each assembly,type:

```
  Parts_req_for_body = SELECT(*) FROM Parts_skill_required
                       WHERE assembly = "750381" AND type = "Body";
  Parts_req_for_fender = SELECT(*) FROM Parts_skill_required
                       WHERE assembly = "750381" AND type = "Fender";
```

and then divide

```
   Good_guys_body =   DIVIDE Possible_supplier ON parts
                      BY Parts_req_for_body ON parts;
   Good_guys_fender = DIVIDE Possible_supplier ON parts
                      BY Parts_req_for_fender ON parts;
```

The quotient relations can have only one attribute here.
Since Possible_supplier had only two attributes, s_id and p_id, and p_id controls the division, only s_id is left.

```
  Good_guys_body: RELATION        Good_guys_fender: RELATION
                 s_id;                          s_id;
                 null                           s1
```

--- ‡‡‡‡‡‡

‡‡‡‡‡‡ **DIVISION IN TERMS OF RELATIONAL PRIMITIVES** Division can be written using primitive functions. Given two relations composed of multiple attributes grouped for division,

```
Dividend:  RELATION
     quot_attr, attr_1;
Divisor :  RELATION
     any_attr, attr_2;
```

where `attr_1` and `attr_2` are sets of attributes have the same domains, but not the same attribute values, the operations defined above allow the rewriting of the division,

```
Quotient = DIVIDE Dividend ON attr_1 WITH Divisor ON attr_2
```

into an equivalent sequence,

```
Wanted_segm    = PROJ Dividend BY quot_attr;
Divisor_match  = PROJ Divisor BY attr_2;
Full_set       = Wanted_segm XPROD Divisor_match;
Remainder      = Full_set DIFF Dividend;
Remainder_segm = PROJ Remainder BY quot_attr;
Quotient       = Wanted_segm DIFF Remainder_segm.
```

The cost of the cartesian product in the third statement of this definition makes a direct implementation of division desirable. Without division, sequences of boolean operations and counting operations are required to provide the power of universal quantifiers when sets of subsets are needed. ‡‡‡‡‡‡

Group Summarization An industrial implementation, REGIS, solves the problem of constructing sets which summarize subsets with a special statement:

```
Sub_rel = SUMMARY Rel ONKEY quot_attr COUNT count_attr
```

Example 9-13 Use of SUMMARY

The query to determine which `Possible_supplier.s_id` can deliver all parts for which `(assembly, type)`, applied to:

```
Possible_supplier:  RELATION        Parts_skill_required :  RELATION
     s_id, p_id :⟩ ;                     assembly, type, p_id :⟩ no_req, ...;
```

can now be constructed as follows:

To find the suppliers who can supply all the parts create a relation by joining `Parts_skill_required` and `Possible_supplier` which contains all types to be supplied. The natural join of the two summaries over two domains provides the result.

```
Supplier_explosion = JOIN
          (PROJECT Parts_skill_required BY type, p_id)   ON p_id
                  WITH Possible_supplier                   ON p_id;
Supplier_capability = SUMMARY Supplier_explosion
                    ONKEY(type,s_id)  COUNT p_id;
Type_needs = SUMMARY (PROJECT Parts_skill_required BY type, p_id)
                    ONKEY type  COUNT p_id;
Result = JOIN Supplier_capability ON type WITH Type_needs ON type.
```

The relation `Sub_rel` will have one tuple for each value type in `quot_attr`. There will be as many domains in `Sub_rel` as there were in `Rel`, and all domains other than `quot_attr` will contain the totals of the matching source domains, except for one arbitrary domain specified by `count_attr`, which will be replaced with an attribute column containing the number of value-tokens for each type-token which defined a subset tuple in `Sub_rel`. In the example we will name the new domain with a prefix "`#`". This count can be used for testing or for the computation of averages, and for operations on subsets, as Example 9-13 shows.

`Supplier_explosion:`

type,	p_id,	s_id;
Body	P1	S1
Body	P2	S1
Body	P2	S2
Body	P4	S2
Body	P4	S4
Fender	P1	S1
Fender	P3	S1
Fender	P3	S2
Fender	P3	S3
Fender	P3	S4
Fender	P3	S5

`Supplier_capability:`

type,	s_id,	#p_id;
Body	S1	2
Body	S2	2
Body	S4	1
Fender	S1	2
Fender	S2	1
Fender	S3	1
Fender	S4	1
Fender	S5	1

`Type_needs:`

type,	#p_id;
Body	3
Fender	2

`Result:`

type,	s_id,	#p_id;
Fender	S1	2

Figure 9-4 Relation summarized to provide universal quantification.

It can be seen that the `SUMMARY` operation in a relational environment provides a facility similar to the subtotal capability used in conventional data processing. The domain-naming convention was added for clarity. REGIS supports about 40 operation types, including the capability to produce graphical output.

9-3-2 File Support

RDMS as well as several other systems built at MIT use the virtual storage facilities of the MULTICS system for file support. The sequential nature of the set operations used in systems which are based on relations increases the locality over random file access. In order to work with fixed-length records, which is important for dense utilization of a sequential file space, all string data in RDMS are coded, and only reference numbers are kept in the database records. All strings are hence retrieved indirectly. String arrays are assigned reference numbers which define the collating sequence. Data elements in the domain `DATE` (see Sec. 8-1-3) are also replaced by numbers allowing sequencing. The coding scheme will be described in Chap. 14.

The use of *indirection* to manage variable-length data is a feature of many systems, including TDMS and TOD. Indirect access to data elements is expensive, but coding can avoid excessive use of indirection. In many production systems arbitrary strings are not used in data-processing analysis so that the cost of indirection is incurred only when reports are generated. If the strings have limited domains, they are best not kept as strings, but coded via a lexicon. The codes are kept in directly accessible storage and can be used for comparison and similar data-processing purposes.

PRTV is based on an extension of PL/1. There is a user language which, when interpreted, manages the user variables and provides them as parameters to PL/1 subroutines. The basic relations are kept as compressed variable-length sequential files, with indexes for fast retrieval. They are updated only by periodic rewriting in off-line mode. Derived relations for the workspace are kept in *on-access* form: the operation sequences specified to create working relations are cataloged to be used for regeneration of the tuples as needed. This keeps the database free of redundancy and the apparent workspace fully current. A cataloged collection of potential derived relations provides also another view of the database. A cataloged sequence of operations is analyzed for optimal execution when the relation is regenerated.

Optimization of Relational Expressions Since PRTV can collect relational processing steps it has the same capability to optimize the execution of requests which exists in the relational calculus systems.

The following optimization steps are considered by PRTV:

1 Restriction due to selection is done as early as possible to reduce tuple volume.

2 Projections of projections are combined into a single pass.

3 Projections by unsorted attributes are deferred.

4 Projections by sorted attributes are done early.

5 Expressions are rearranged according to the estimated partitioning efficiency (selectivity) of their terms.

6 Intermediate relations appearing in separate processing sequences are shared when shared use is profitable.

7 Attributes which control joins are presorted if this reduces execution effort.

8 Comparison or merging of files which are being presorted can be carried out as the tuples are being emitted in sorted order, so that sorted files do not have to be actually generated.

Relational systems which provide indexes will also treat indexed attributes early, similar to sorted attributes in PRTV.

9-3-3 The Execution Cost of Relational Operations

The simplicity of relational operations can hide potentially high execution costs. Effective use of access structures, as encountered in Chaps. 3 and 4, can reduce these costs greatly. A good database-management system hence has to match the operations to the available access structures and determine how to execute the operations. If the queries can be rearranged, as described for PRTV and relational calculus systems, there will be an interaction between the way operations can be processed and the optimal arrangement of the operations from a query. Here we consider only the basic operations, and specifically the join.

The brute force approach to obtain the result relation specified by statements using multiple relations is to compare each tuple of one with each tuple of each of the other relations. More specifically this is done by executing nested loops fetching tuples from the relations, one loop for each explicit or implicit tuple variable. This technique is called an *inner-outer loop join*. The number of steps $S(\bowtie)$ required for a join of two relations is the product of the relation sizes `#(Rel_i)`:

$$S(\bowtie) = \texttt{\#(Rel\_1)} \times \texttt{\#(Rel\_2)} \qquad \langle\text{Inner-outer loop}\rangle\ 9\text{-}1$$

In each loop step a tuple has to be retrieved and analyzed. To excute Example 9-4, finding employees with older children, the number of operations would be on the order of `#(Employee)`$\times$ `#(Children)`. Example 9-5, finding bosses, would require `#(Employee_2)`$^2\times$ `#(Supervision)` operations.

If, in either or both files, the relations are already in some sequence for the join attribute, the attributes can be processed in order and no loop is required on that file. If both files are in sequence only a merge is required. A merge alone requires only

$$S(\bowtie) = \texttt{\#(Rel\_1)} + \texttt{\#(Rel\_2)} \qquad \langle\text{Merge for Join of Sequenced Files}\rangle\ 9\text{-}2$$

A file can only be in one sequence, and maintenance of sequentiality can be costly, but we need not limit this approach to files which are initially sequential on the join attribute. Previous operations within a relational expression may be able to leave a workspace in sorted order, or the files may be accessed to obtain tuples in join-attribute sequence.

In order to obtain tuples in sequence, if the files are not, two techniques are possible: sorting, or fetching the tuples via an index or by hashing. The cost of sorting was estimated in Eq. 3-11 and will involve on the order of

$$S(sort) = 2\,\texttt{\#(Rel)}/Bfr \times (1 + \log_2(\texttt{\#(Rel)}/Bfr)) \qquad 9\text{-}3$$

steps for each file. The initial factor 2 accounts for the need to read and rewrite tuples, and the factor *Bfr* accounts for the advantage obtained because sorting is performed on sequential blocks. Retrieval of `#(Rel)` tuples via an index has a cost of similar magnitude

$$S(indexed\ fetching) = \texttt{\#(Rel)} \times (\log_y + 1)(\texttt{\#(Rel)}) \qquad 9\text{-}4$$

Of course, an index has to be available. Since many index accesses on the join attribute will be made, it can be profitable to attempt to keep much of the index in memory, effectively reducing the $\log_y$ term. If a file holding a relation provides direct access via the join attribute, `#(Rel)` tuples can be retrieved at a cost of

$$S(hashed\ fetching) = (1 + p) \times \texttt{\#(Rel)} \qquad 9\text{-}5$$

where the overflow cost p can be kept quite low and constant, as discussed with Eq. 3-71. The constraint is, of course, that only one hashed access attribute is possible per file; a database designer will allocate it to a known frequent join attribute, typically along an important connection.

Separability An aspect of designing databases is to determine access structures, i.e., assign indexes, clustering, and hashing to record attributes. It is important to note that the techniques based on merging (Eq. 9-2 with 9-3 and 9-4) have the property of *separability*; that is, the file implementation and access structures chosen for one relation do not affect the processing method of the other relation. The designer can hence choose the best implementation scheme on a relation-by-relation basis. The existence of indexes is assumed in the design phase. After the design is complete, no indexes need be assigned to attributes which do not use them for join or restriction purposes.

The exception to separability is when merging occurs with hashed access (Eq. 9-5). Hashed access can be used on only one relation and only one attribute. Here the join is carried out using the join attributes obtained from sequentially retrieved tuples from the other relation (`Rel_2`) to fetch matching tuples by hashing in the hashed relation (`Rel_1`). If the join attribute in `Rel_2` is unique, no sequencing at all is required in `Rel_2`, since any sequence will do for hashed access to `Rel_1`.

Any restriction is easily applied to the merging methods prior to the join. Only selected records need be submitted to a sort, and when indexes are used the TID list of records to be joined can be restricted based on matching TIDs from selection attributes, as shown in Sec. 4-2-3.

If the inner-outer-loop join method is chosen, the decision which file is to be assigned to the outer loop and which one to the inner loop can be deferred to query-processing time, so that some separability exists within this choice. If one of the relations will fit into memory, perhaps after restriction, this relation will be used for the inner loop. If restrictions will not reduce one of the relations sufficiently, the inner-outer-loop method will also be hampered by the inability to incorporate restrictions prior to looping without making a copy of the file.

A further design step is the decision on which attribute to select for clustering, so that sequentiality can be exploited. The ability to cluster exists for only one attribute for each relation. The attribute chosen will be the one involved in most retrievals and joins, with consideration of the update cost of clustered attributes. Chapter 3 provides the basis for optimization of database performance.

Projection Relational projection requires also that each tuple be compared with every other tuple in the relation to eliminate duplicates, a computation which takes about $\frac{1}{2}\texttt{\#(Rel)}^2$ operations when implemented simply by looping. Some systems avoid this cost by generating tables with redundant entries unless explicitly instructed as shown in Example 9-3.

If the relation collapses materially in the process, because many tuples are identical, fewer steps will be used. A merge-sorting process, where the merge eliminates one of two identical tuples, allows the projection process to be done in $S(\prod)$ steps, where

$$\texttt{\#}_b \log(\texttt{\#}_b) \geq S(\prod) > \texttt{\#}_f \log(\texttt{\#}_f) \qquad \text{9-6}$$

where $\texttt{\#}_b$ and $\texttt{\#}_f$ are the beginning and final sizes of the projected relation `Rel`. Projection is also often combined with selection; each accessed tuple is inspected for relevance and immediately discarded if not wanted.

9-4 HIERARCHICAL DATABASES

In the development of view models hierarchical concepts as nests play an important role. A model based on hierarchies can be directly implemented using nest-oriented file structures and appropriate schema facilities. Such a model is then restricted to one entity relation and its nest relations. In order to accommodate data that do not fit within a single tree, a hierarchical database system will allow the existence of multiple trees or a *forest*. The trees of the forest will have different heights; there are often many single-level trees, equivalent to entity relations in first normal form. These contain data structures which do not fit into the hierarchical structure. A small forest based on the example relations used in Chap. 7 is shown in Fig. 9-5, the structure is again visually simplified through the use of Bachman arrows.

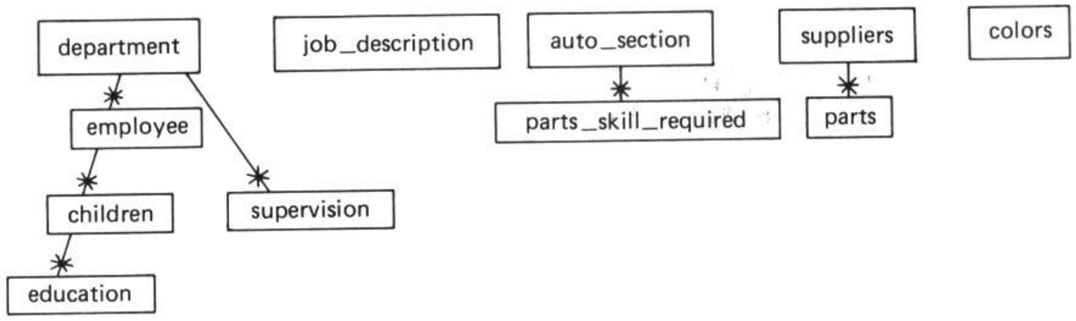


Figure 9-5 Our relations in a forest.

A database model which is bound to any predefined structure will lose considerably in flexibility. On the other hand, the definition of access paths, implied by the structure, means that these paths do not have to be created during query or update processing, but that they already exist when needed. The advantage of such early binding is often a considerable gain in processing speed and a simplification of query formulation when the data model relevant to the query is within the bound database structure. Database structures can be implemented to match almost *any* model but will never be able to match *all* models satisfactorily. One person's hierarchy is often another one's bureaucracy. The hierarchical model is relatively simple and satisfies the conceptual needs of database users in many cases.

9-4-1 Manipulating Trees

A *tree* is by definition not in first normal form. We are dealing therefore with a basic structure whose elements cannot be described by the triple: relation name, tuple name, attribute name. Relational operations, such as projection or join, are difficult to define and rarely implemented in these systems. It is, however, possible in most cases to reformulate relational queries into search strategies on trees, or to transform unambiguous hierarchical queries into equivalent relational queries.

Relations which will often participate in the equivalent of joins are best kept at the top level of the hierarchy; this includes primary and referenced entity relations as well as lexicons. Tuples in the lower-level nests of a tree provide only record segments, since ancestor data are a logical part of such tuples.

Brooms If a segment in a hierarchical structure has been selected, for instance, `employee="Hare"` in Fig. 9-6, we can identify with this segment the structural sequence of all segments which are its owners or ancestors, namely,

(`department="Assembly", file="Personnel"`),

as well as the subtrees for this segment, namely, nests of

(`children, education,` and `supervision`).

This construct has been named a *broom*, and the segments which make up the broom of `employee="Hare"` are indicated in the figure. A broom is often the unit to be manipulated here, rather than a single segment. Segments in hierarchical structures depend greatly on their ancestors and hence cannot be freely moved about.

The operations *union*, *intersection*, and *complement* of two brooms remain definable in terms of the operations $\vee$, $\wedge$, and $\neg$on their members.

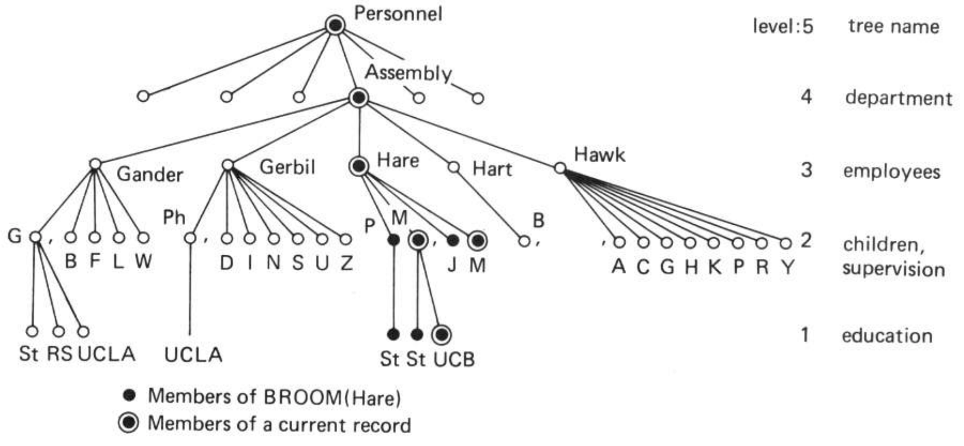


Figure 9-6 Personnel tree in a hierarchical database.

⧻⧻⧻ Operations with brooms require some care. The broom based on `"Hare"` includes `"Assembly"`, as does a broom based on `"Gander"`. The *union* of the two brooms includes all elements in both brooms. The *intersection* of the two brooms contains only `"Assembly"` and `"Personnel"`. The *complement* of the broom of `"Hare"` contains all employees except `"Hare"`, but not `"Assembly"` or `"Personnel"`.

The results of *intersection* and *complement* are hence incomplete brooms, and subsequent operations may not be obvious, as will be shown in Example 9-14.

Retrieval of records based on selected segments requires definition of the record based on a qualifying segment. To identify a segment in a nest qualifying terms as

```
FIRST, LAST, NUMBER = ns, ANY, ALL
```

can be used. To find now the current school for the oldest child, we can request

```
LAST.children.LAST.education
```

or to find who sent all his children to St. Mary's,

```
ALL.children.ANY.education = "St. Mary"
```

An understanding of the structure plays an important role in these systems. A search for a specific record is constrained by the nesting structure. ⧻⧻⧻

In order to simplify the construction of queries, we introduce the operation `BROOM` which defines all segments of the broom for a segment. Membership of a segment in a BROOM is tested by the operator IN, analogous to the set membership operator . In order to constrain a search to a specific broom, say the assembly department, we can state

```
    employees IN BROOM(department="Assembly")
```

Brooms can also define segments based on selected segments lower in the hierarchy.

```
    employees IN BROOM(education.schoolname="St. Mary")
```

The use of broom in such a case implies ANY.education.

We show some examples below. These are not based on any specific implementation but represent a generalization of available facilities in query languages and database-analysis programs, as well as some ideas from a proposal for a database retrieval language for tree-structured system, BOLTS [Hardgrave80]. Most hierarchical query languages do provide for correct expression of arbitrary queries, but their syntax may actually be awkward and not show the issue in an obvious manner.

Example 9-14 Use of BROOMS

Let us assume that there is a celebration at St. Mary's school on Thursday. To find which employees will be affected, we can ask

```
    GET employees.name |
        employees.children.LAST.education.schoolname="St. Mary"
```

Employees with more than one child at St. Mary's will be listed multiple times. We really want

```
    GET employees.name |
        employee IN BROOM(LAST.education,schoolname="St. Mary")
```

Now employees is the controlling variable and the descendants or ancestors can be tested for the validity of the predicate. When multiple predicates have to be combined, brooms can provide more flexibility, since the access path does not have to be specified segment by segment. We want to locate the employees who have children at St. Mary's and at UC Berkeley. The simple question

```
    GET employees.name |
        LAST education.schoolname="St. Mary" ∧
        LAST education.schoolname="UC Berkeley"
```

does not make sense; if one predicate is true, the other one is false, and the result of this query will always be NULL. Using a BROOM clause allows precision:

```
    GET employees
        IN BROOM(LAST education.schoolname="St. Mary") ∧
        IN BROOM(LAST education.schoolname="UC Berkeley")
```

using a simplified form of the query retrieval specifications to obtain complete segments.

The qualification is now brought to the level of the employee. Similar action is required if we are interested only in employees of the assembly department.

The attribute `department.name` is not defined in the `employees` segments, so that we will ask

```
GET employees IN BROOM(department.name="Assembly") ∧
    IN BROOM(LAST education.schoolname="St. Mary") ∧
    IN BROOM(LAST education.schoolname="UC Berkeley").
```

The qualification can be at a different level than the object being retrieved. We can ask which employees have children who went to St. Mary's and reform school.

```
GET employees IN BROOM(children
    IN BROOM(education.schoolname="St. Mary") ∧
    IN BROOM(education.schoolname="Reform school") ).
```

Failing to specify `children` would also retrieve employees with one child in Reform school and another child in St. Mary's. These expressions may combine segments from different levels. The question above, "Which employee has children at UC Berkeley who are younger than 18 years old?" is stated as

```
GET employees IN BROOM(children ∧
    IN BROOM(LAST.education.schoolname="UC Berkeley") ∧
             children.age_c < 18 )
```

Only within the `BROOM(children)` are the predicates related; wrong answers would again be obtained if the data satisfying the predicates are aggregated for the `employees` broom.

In complex hierarchical structures, especially if some levels do not have a strong semantic identity, the formulation and analysis of queries requires care and insight. Levels will sometimes not be obvious if a hierarchical database is constructed out of hierarchical data models which differ in level. A financial model may view employees within departments, but an organizational model may consider employees to be within sections which are within departments.

Currency An interface with a conventional programming language which does not have capabilities to manipulate repeating segments or nests requires establishing a correspondence of one data segment or item per name. The programming interface will obtain a record with one segment instance for each type of segment.

In order to provide data in such a flat presentation from a hierarchical file, a *currency* indicator for each segment type identifies the segments of the total record which are to be used. This means that for the entity relation at the highest level, one segment is identified and one member is identified out of each nest below. An example of currency is given in Fig. 9-6.

```
Personnel_record: Department, Employee, Children, School, Supervision.
                           Assembly    Hare      Mary  UC Berkeley   Mike
```

A record is a catenation of all current segments. When the result of a query identifies a segment, say `Mary`, all the ancestral owners, the handle of the broom, should also be presented, because the element by itself is not sufficiently identified. Records are well defined, but incomplete, if some lower-level segments have not been chosen, say `UC Berkeley` and `Mike`.

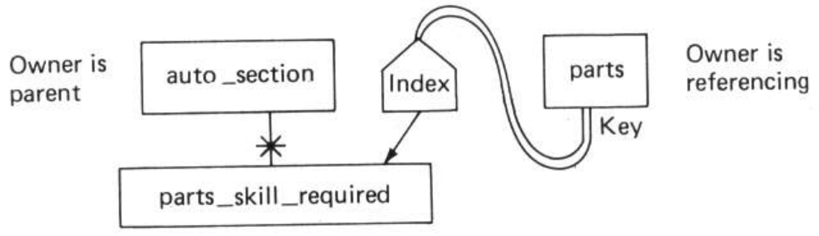


Figure 9-7 Association in a hierarchy.

Nonhierarchical Relations An associative relation, if required, has to be made a nest of either one of the owners as shown in Fig. 9-7. Access from outside of the implemented hierarchy requires a search, through an index if needed for speed, as it does in a relational implementation. Since the situation is not symmetric, the user has to be aware of the design decisions and their implementation. Efficiency considerations create the temptation to put the data which is functionally dependent on the nonhierarchical owner (here `Parts`) into the nest (`Parts_skill_required`). Two considerations prohibit this:

Orphans Members in a hierarchy cannot be kept around without their owner. If a part is not required at one point on the production cycle by any of the `auto_sections`, information about this part kept in the nest would be lost. The lack of a distinct `Part` relation also makes it impossible to store data about a new part not yet used by any `auto_section`.

Redundancy There will be multiple `Parts_skill_required` records for each `Parts` record. Inclusion of a part description with several dependent fields in each instance of the `Parts_skill_required` increases update efforts. If updating of the descriptions of one part is required, many records of `Parts_skill_required` have to be modified. Also, if the part description is at all voluminous, much space is wasted due to the redundancy.

The design problems shown here are common to all bound databases. In a system which is strictly hierarchical the binding of associations is not completed until the time that the query is being processed.

Example 9-15 SYSTEM 2000 Schema Entries

The schema entries specify

```
fieldnumber,fieldname (data_type, format(length) IN owner_segment)
    ...
 1*  name  (NAME X(16))
 2*  birthdate (DATE)
 3*  height (DECIMAL NUMBER 9.99)
 4*  weight (NON-KEY INTEGER NUMBER 999)
 5*  children (REPEATING GROUP)
 6*  child (NAME X IN 5)
 7*  age_c (INTEGER NUMBER 99 IN 5)
 8*  education (REPEATING GROUP)
 9*  school_type (NAME X(10) IN 8)
    ...
```

9-4-2 Hierarchical Database Systems

A commercial system which provides a hierarchical structure and is available for many large computer systems is the INTEL/MRI SYSTEM 2000. It provides self-contained system services and also allows database access from COBOL, PL/1, FORTRAN, or assembly language programs. The database definition is independent of the processing programs.

The definition for the `Employee` entry in the schema for S2000 might be as shown in Example 9-15. Default values are assigned as needed. One database may have multiple entity relations plus their owned nest relations, up to 32 levels deep. A relation can be the owner of multiple nest relation types, as is required to implement in our example the `Children` and `Supervision` of `Employee`.

A subschema and data-manipulation statements are combined with host-language statements when programmed access is required. In order to process such a hybrid program a technique similar to the one illustrated for IDS in Fig. 8-13 is used. The subschema and data-manipulation statement translator is specific to the source language and computer type, but the SYSTEM 2000 statements are the same for any host language. The subschema for the personnel tree is shown in Example 9-16 as it would appear in a COBOL† environment. This host schema description has to be a proper subset of the schema used when the database was defined.

Example 9-16 SYSTEM 2000 Subschema

```
COMMBLOCK OF Personnel
  01  Personnel
    02  /*  Area to communicate system status, error, and control data */
  SCHEMA Employee OF Personnel
  01  Employee
    02  name         PICTURE IS X(16).
    02  birthdate    PICTURE IS X(8).
    02  height       PICTURE IS 9(6).
    02  weight       PICTURE IS 9(8).
  SCHEMA Children OF Employee
  01  Children
    02  child        PICTURE IS X(6).
    02  age_c        PICTURE IS 99.
  SCHEMA Education OF Children
  01  Education
    02  school_type  PICTURE IS X(10).
    02  school_name  PICTURE IS X(12).
    02  subject      PICTURE IS X(8).
  SCHEMA  Supervision OF Employee
  01  Supervision
    02  subordinate  PICTURE IS X(16).
    02  year         PICTURE IS 99.
  END SCHEMAS.
```

† We keep variable name usage consistent; COBOL actually uses "–" for "_".

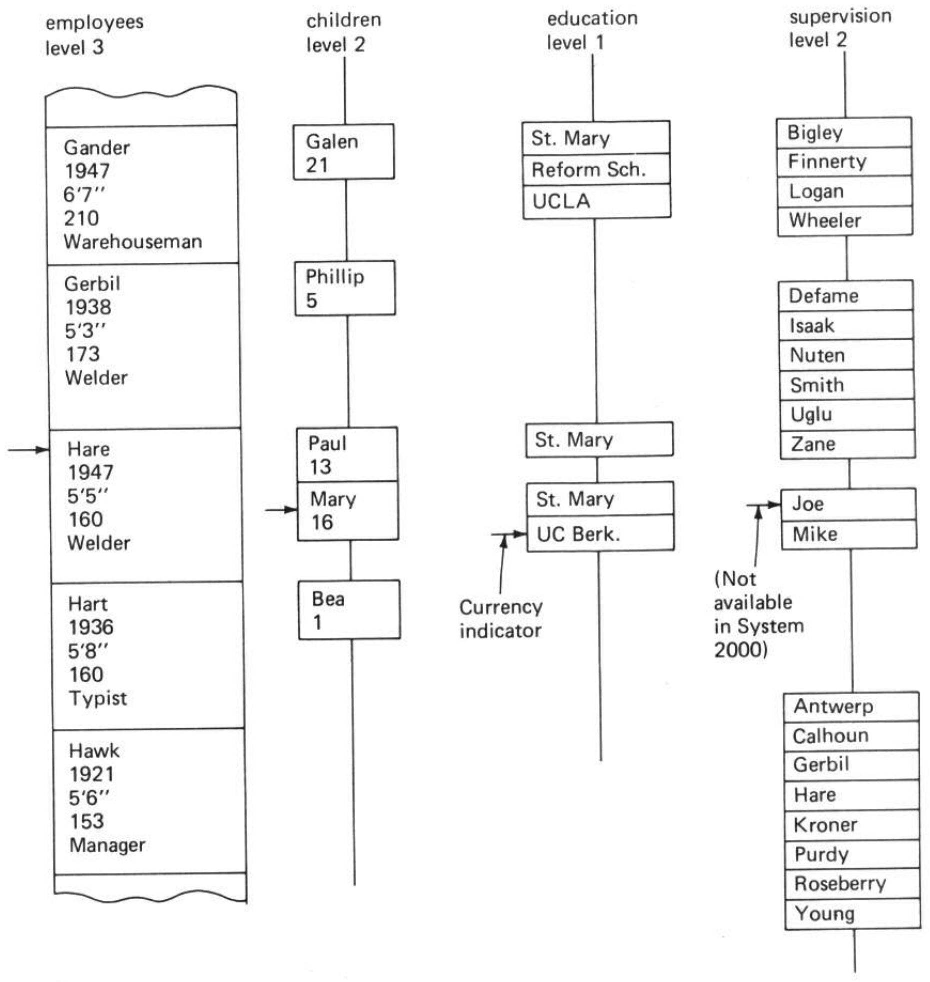


Figure 9-8 Hierarchy implemented through nest files.

Figure 9-8 shows the relationships of the four files comprising the single hierarchy. Each relation, entity or nest, is implemented with a separate file. The level `01` entries in the `SCHEMA` for each file provide space for the owner ruling part in the nest relations. All cross references between the files are maintained by the system, so that the user is not aware of any structure outside of the declared schema.

All attributes are indexed unless `NON-KEY` appears in the schema definitions. If a retrieval request involves an attribute which is not indexed, or requires an exhaustive scan of the database because of the form of the selection clause, i.e.

```
GET employee WHERE weight > 150
```

then long processing times can be encountered.

One currency indicator for each level is maintained by SYSTEM 2000, as indicated in Fig. 9-8. The collection of current segments defines a record, which may be incomplete. A `GET1` command with a key establishes a new currency indicator position for the matched segment, and also sets ancestor currency indicators, while descendant currency indicators are reset to `NULL`. `GET1 NEXT` commands which refer to a `NULL` level move the currency indicator to the first record while again resetting yet descendant currency indicators to `NULL`. From the first record onward

GET1 NEXT will obtain the other records sequentially in the nest file, moving from nest to nest. Currency is redefined as the retrieval proceeds. GETA and GETD commands obtain ancestors and descendants.

If one nest at a time is to be obtained, it is necessary to recognize when the current owner is about to change. SYSTEM 2000 provides the combination LOCATE and GET NEXT for this purpose. An END_OF_DATA message is generated as a result when the NEXT entry is outside of the broom of the current ancestor. The ancestor is not restricted to the immediate owner. To obtain, for instance, the schools attended by the children of Hare, the sequence could be

Example 9-17 Navigating in a Hierarchy

```
    LOCATE education WHERE name = "Hare"
    GET NEXT education -> "St. Mary"
    GET NEXT education -> "St. Mary"
    GET NEXT education -> "UC Berkeley"
    GET NEXT education -> END_OF_DATA
```

In addition the program will contain OPEN, GET, INSERT, MODIFY, REMOVE, and CLOSE statements of various flavors which will allow accessing the various files comprising the tree. A WHERE clause adds selection capability, and ORDER BY provides sequencing. A HAS clause allows specification of the descendants of a segment, so that a predicate can be restricted to that part of a broom. Aggregation operations available in the self-contained language are SUM, COUNT, MINIMUM, MAXIMUM, AVERAGE, and the equally important SIGMA to obtain the standard deviation.

Completeness Earlier versions of SYSTEM 2000 as well as some other DBMSs will not permit retrieval requests to be executed which require exhaustive searches of the database. Such systems are therefore not *functionally complete* for retrieval. Most relational systems discussed earlier were functionally complete, since they implemented a mathematical approach which covers all possibilities. The fact that a system is complete does not imply that all retrievals are carried out with equal dispatch. The documentation for OASIS, a hierarchical database system for university administrative needs, provides detailed guidelines for programmers, so that they can formulate queries with optimal partitioning and indexing efficiency. The system still allows arbitrary queries, so that management can use the database completely, without regard to efficiency.

9-4-3 File Support

Database systems such as SYSTEM 2000 or ADABAS do not actually implement the database by distinct files for each relation. The segments of the hierarchical record are placed as they are entered into one data file. All hierarchical relationships in SYSTEM 2000 are expressed by a second file which contains pointer chains that are similar to ring structures, but all data fields are referenced by pointers to the data file. The pointer file has high locality because of its small size, but getting the next record requires file accesses. Indexes are kept in other files.

The ADABAS system permits these access structures to be created either at schema definition time or at access time. Selecting when and which connections are to be bound becomes partially a decision of the user. A good understanding of the underlying database structure can make much difference in performance.

A completely different approach is followed by OASIS. An instance of a tuple in an entity relation, together with all its descendants, is placed into a single compact variable-length record. The entries appear within the record in a compact and fixed order, beginning with the entity relation segments. Figure 9-9 sketches the record content where `employees` is the top level entity file.

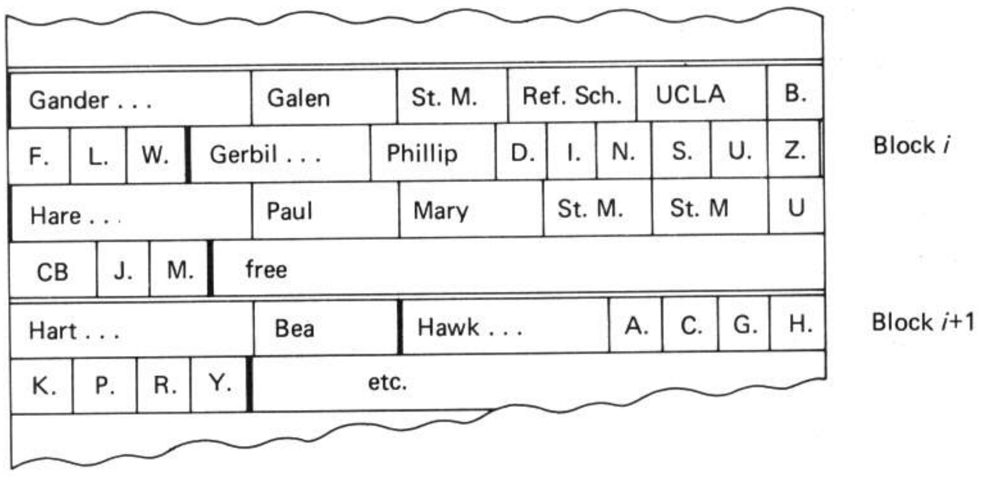


Figure 9-9 Compacted hierarchy.

A fetch of a record provides access to an entire broom at a time, but the design limits the size of a hierarchy, since a record cannot span blocks.

A block will contain several records, and since records can grow over time, the original space allocation, even if a low loading density was originally specified, may not be adequate. This problem is solved through the use of *indirect references*. A record number index is maintained which allows placement of a record into any block in the file having sufficient free space. All indexes based on attribute values in the records refer to the record number, so that indexes are not bound to relative block addresses.

Other alternatives used to implement hierarchical systems have included the use of ring structures. We will find samples of this design among systems with network capability.

9-5 DATABASES WITH NETWORK CAPABILITY

A *network* is created when structures more complex than hierarchies are bound. A hierarchy is as complex a structure as can be built within a single file using ordering conventions for the segments. Even then reference pointers are found in many implementations. In a network references are an inherent part of the structure. We will refer to these structural references as *links*.

Links can be implemented by any of the reference structures discussed in Sec. 8-1-2 (pointer, symbolic, indirect). Symbolic references are not of practical interest in this section since they defer the binding and hence the existence of a network to query processing time. Direct pointer references can be used only if records are not moved within the database during their lifetime, since otherwise the pointers lose validity. On the other hand, indirect pointers can be changed by simply changing the pointer index when records are moved. Indirect pointers are hence the common means to implement linkages.

Loss or invalidation of a link implies loss of information. Pointer or indirect references may describe the structure redundantly because of the continued existence of symbolic references. If such pointer or indirect references are not redundant, however, because the symbolic reference has been omitted, we call them *essential links*. Maintenance of files using essential links requires carefully worked out procedures to avoid invalidation of these links. In many network systems, links are essential, although an application designer may decide to maintain redundancy of links by keeping symbolic references within the data records as well.

9-5-1 A Simple Network Implementation

A DBMS available on many small and large computers is TOTAL. It provides a two-level network hierarchy. The top level contains entity relations, and the bottom level contains either nest or associative relations. Figure 9-10 shows the placement of relations in such a structure. The top-level relations are implemented as direct files, and the bottom-level relations as chains. These chains are rings without the final link back to the top-level record.

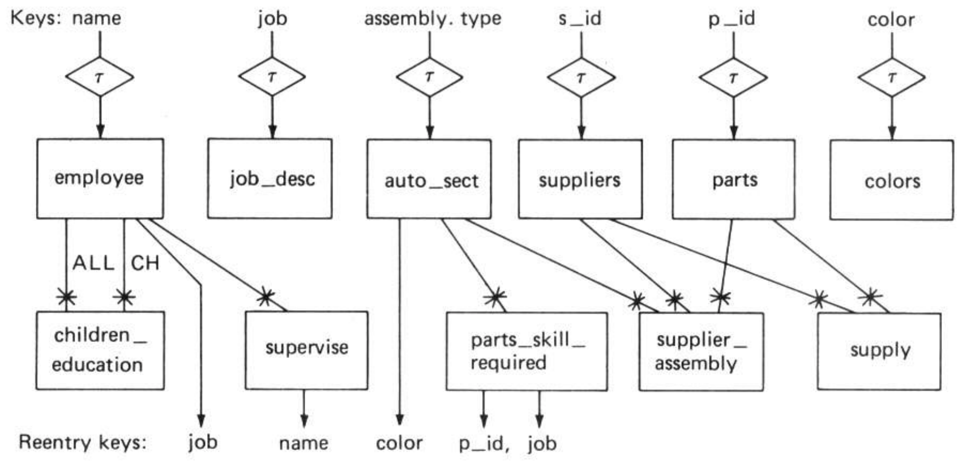


Figure 9-10 TOTAL network structure.

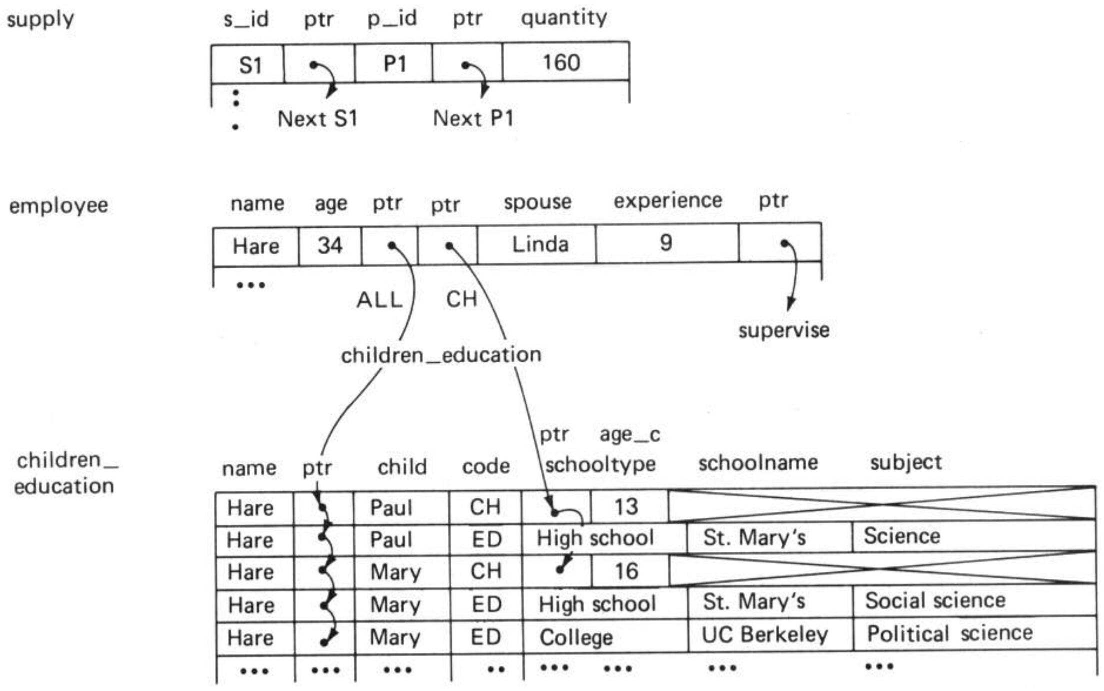

supply
s_id ptr p_id ptr quantity
S1 P1 160
Next S1
Next P1
employee
name age ptr ptr spouse experience ptr
Hare 34 Linda 9
ALL
CH
supervise
children_education
children_ education
name ptr child code ptr age_c schooltype schoolname subject
Hare Paul CH 13
Hare Paul ED High school St. Mary's Science
Hare Mary CH 16
Hare Mary ED High school St. Mary's Social science
Hare Mary ED College UC Berkeley Political science

Figure 9-11 TOTAL network records.

A top-level entity record can start multiple chains. A bottom-level, owned record may be a member of multiple chains. At the bottom level we find then nest relations and associations of entity relations, as shown by the `supply` record in Fig. 9-11. There may be multiple chains from an entity record which link different record subtypes within one owned relation. Such connections allow an approximation of more complex structures, say, the required three-level hierarchy of `Employee—* Children—* Education` by `Employee—* Children_education`.

Linkages are created and modified automatically when a record is inserted. The linkages are not *essential*: a complete ruling part is kept in all the bottom-level records. Since the chains do not return to the owner records, the key in the ruling part provides a symbolic reference argument, as the only means to locate the owner of a bottom-level record. This is indicated by reference arrows.

Deletion of an entity record is not permitted while it has chain members. In the example member records in `Children_education` must be deleted before the `Employee` record can be deleted. This implements a reference connection.

Referenced relations and lexicons are hence best placed with the entity relations at the top level. When they refer in turn to relations on the bottom level, they can use linkages. Linkages to implement additional connections have to be handled by the application programs, using symbolic references. Because of the use of direct file access the top-level relations cannot be processed serially; only unordered sequential access is possible. This may force some relations to the bottom level which otherwise would fit better on top.

All linkages maintained by TOTAL are defined in the schema. No new relations or new linkage types can be dynamically created. An associative relation, such as `supplier_assembly`, has to be created at the time the schema is defined and will be updated as its owners change.

9-5-2 The DBTG Access Specifications

Links and sets of links are an important aspect of the database architecture defined in the schema language of the Data Description Language Committee (DDLC) of CODASYL. Although the original Database Task Group (DBTG) reports provided only the language specifications and functional requirements, the specifications had so much detail that many implementation choices were implied. Ring structured files (Sec. 3-6) and hashing are common implementation techniques; we will review their use here in Sec. 9-5-4. Changes issued in 1978 by DDLC are directed toward increased implementation independence. The desire to obtain machine-independent software, achieved by CODASYL to a large extent with the COBOL language, is the reason for the degree of detail in the specification. We describe here the recent specifications. The examples shown also consider earlier implementations, since most available systems are based on the 1974 specifications.

Network Structure The CODASYL schema permits the construction of nearly arbitrary networks of connected relations. We encountered in the DDL schema specification shown in Fig. 8-9 the two principal components used to construct a CODASYL database:

RECORDs: Records implement the relation concept of tuples, and contain the data fields. A collection of records of the same name implements either a relation or a table, since there is a clause which indicates whether duplicate tuples are not or are allowed.

SETs: Records may be linked to each other, thereby implementing the concept of a connection. Each *link-set* specifies the owner and member record names. The owner of a linkage is often the owner relation in a hierarchy, or one of the owners in an associative relationship. Link-sets may also be maintained in such a way that they implement reference connections. Specifying that the SYSTEM is the owner provides an initial entry point to a collection of record.

To illustrate the use of some of the alternatives we will define in Example 9-18 a simple schema, similar to the hierarchical schema given in Example 9-16. To show the network capabilities, we implement the Department as an entity and add an association of Employees and Departments as a record type Worked_in. The ownership connections from Fig. 7-29 are implemented as link-sets.

Example 9-18 CODASYL Schema

```
SCHEMA NAME IS Personnel.
  AREA NAME IS personal_data.
  AREA NAME IS company_data.
    RECORD-NAME is Employee WITHIN personal_data
       KEY empl_key IS ASCENDING ssn  DUPLICATES ARE NOT ALLOWED.
        02  name            PICTURE IS X(16).
        02  ssn             PICTURE IS 9(6) CHECK IS NONULL.
        02  birthdate       PICTURE IS X(8) CHECK IS PROCEDURE date_verify.
        02  supervisor      PIC X(16). /* Reference to another Employee */
        02  job             PIC X(8).  /* Reference to a Job file */
   RECORD-NAME IS Children WITHIN personal_data.
       02  child           PICTURE IS X(6).
       02  age_c           PICTURE IS 99.
```

Continuation of Example 9-18.

```
RECORD-NAME is Education WITHIN personal_data.
   02  school_type       PICTURE IS X(10).
   02  school_year       PICTURE IS 9(4) CHECK IS VALUE 1900 THRU 2000.
   02  school_name       PICTURE IS X(12).
   02  subject           PICTURE IS X(8).
RECORD-NAME Supervision WITHIN company_data.
   02  subordinate       PICTURE IS X(16).
   02  year              PICTURE IS 99.
RECORD-NAME Department WITHIN company_data
  KEY department_key IS ASCENDING dep_name  DUPLICATES ARE NOT ALLOWED.
   02  dep_name          PICTURE IS X(10).
   02  year_established PICTURE IS 99.
   02  year_dismantled  PICTURE IS 99.
RECORD-NAME Worked_in WITHIN company_data.
   02  w_emp_name        PICTURE IS X(16).
   02  w_dep_name        PICTURE IS X(10).
   02  w_year            PICTURE IS 99.

SET Our_employees.
   OWNER IS SYSTEM.
   ORDER IS PERMANENT     INSERTION IS SORTED BY DEFINED KEYS.
  MEMBER IS Employee
   INSERTION IS MANUAL    RETENTION IS OPTIONAL.
SET Parenthood.
   OWNER IS Employee.
   ORDER IS PERMANENT     INSERTION IS SORTED BY DEFINED KEYS.
  MEMBER IS Children
   INSERTION IS AUTOMATIC    RETENTION IS FIXED.
SET Children_education.
   OWNER IS Children.
   ORDER IS PERMANENT     INSERTION IS LAST.
  MEMBER IS Education
   INSERTION IS AUTOMATIC    RETENTION IS FIXED
   RANGE KEY edu_key IS ASCENDING school_year DUPLICATES LAST.
SET Employee_ties.        /* defines both ownership of supervisees   */
   OWNER IS Employee.   /*      and association with departments */
   ORDER IS PERMANENT
     INSERTION IS SORTED WITHIN RECORD-TYPE BY DEFINED KEYS.
  MEMBER IS Supervision   /*  One record type owned by this set   */
   INSERTION IS MANUAL     RETENTION IS FIXED
   RANGE KEY IS ASCENDING year  DUPLICATES ARE LAST.
  MEMBER IS Worked_in      /*  and another one owned by it    */
   INSERTION IS MANUAL     RETENTION IS MANDATORY
   RANGE KEY IS ASCENDING year  DUPLICATES ARE LAST.
SET Department_assignments.
   OWNER IS Department.
   ORDER IS PERMANENT  INSERTION IS LAST.
  MEMBER IS Worked_in  INSERTION IS MANUAL  RETENTION IS MANDATORY.
```

A single link-set, Employee_ties, is used for both the Supervision and the Worked_in connection, perhaps since for most Employees the total number of members will be small. A better reason for a link-set to have several types of members occurs if different record types have to be managed similarly, such as Infants, School_age_children, and Other_dependents. All members of a link-set must specify keys of the same format to allow an overall sorted sequence to be maintained.

To simplify the presentation we skip the COBOL subschema, which is the extenal schema actually made available to the data manipulation programs. We also leave out the header statements used by COBOL to separate sections of a program.

Not all connections specified in a conceptual schema as developed in Chap. 7 need to be implemented as link-sets. There is a cost to maintaining link-sets, and the benefits are obtained only when link-sets are used to retrieve information. Design decisions similar to those made for index selection in Sec. 9-3-3 are made for selecting link-sets, but accessing relations without link-sets is quite awkward in many CODASYL-based systems. To reach a connected record for which no link-set has been defined a program has to search through all the records which implement the destination relation.

In order to obtain a certain record an iterative process is performed:

1 Enter the database in one of two ways: either use a SYSTEM owned link-set and continue according to step **2**, or select a record by giving a record name and its key or position and continue according to step **3**.

2 Select a link-set; determine which entry to follow to a member record.

3 From the member record found continue either to successor members of this set and continue according to this step **3**, or select a link-set owned by this member record, if any, and continue with step **2**.

It is obvious that a programmer has many choices available in a schema which specifies extensive connections. The repeated process of moving from a current record to a goal record was termed "*navigation*" in a Turing award lecture given by the major contributor to this database approach [Bachman^{73}]. At each embarkation point one record can be retrieved, modified, or stored. The manipulation of the single records obtained at each point is easily achieved using host programming statements. The map available during the voyages is the external schema allocated to the programmer. Figure 9-12 sketches the entire structure specified by the schema designed for one of the first applications of a system based on the DBTG report, with an indication of the external subschema used by one application group.

An important aspect of CODASYL databases is the ability to implement nonhierarchical connections. Associative relations can be implemented since a record may be a member of more than one link-set. The relation Worked_in is linked in Example 9-18 both to Employee and to Department. The traditional relation Supply would be owned both by Suppliers and by Parts. A network resembling the semantic connections shown in Fig. 7-29 can be constructed.

Reference connections are not as conveniently established with link-sets. To assure correct manipulation, an accessible key for the referenced relation must be defined. Then the connections can be maintained using MANUAL INSERTION, as shown in Example 9-24.

Networks are typical for "bills-of-materials" problems. We can access through the links all the parts needed for an Auto_section, and, if we have a shortage, go via the link-set to Parts, and obtain the list of other equipment using the same part. Then the other relevant Auto_sections can be consulted for their requirements and a possible surplus.

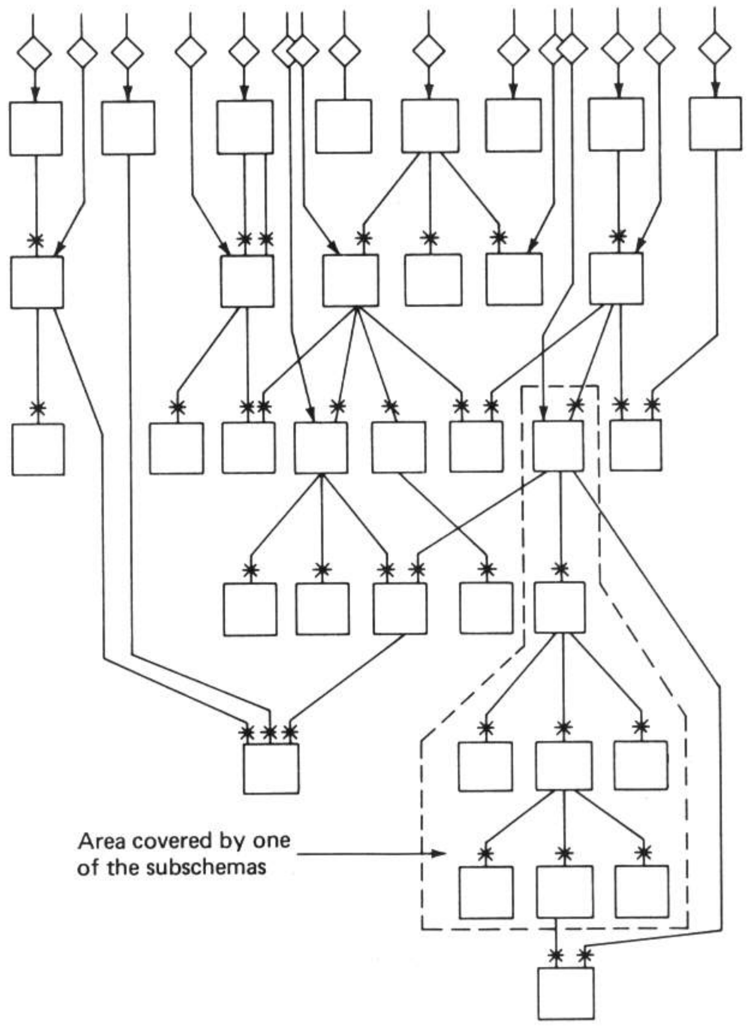


Figure 9-12 Sketch of a commercial database.

Currency Pointers Whenever a link-set entry or member record is reached a *currency* pointer is defined, so that it is always possible to return to an earlier link-set entry or member record. The NEXT and PRIOR options permit forward and backward tracking relative to these currency pointers. A large number of currency pointers can be active at any time: one for each link-set and each record type in the schema. Currency pointers are also available for the last reference in an AREA, and there is one procedure currency pointer which always identifies the last record referenced by the processing program. We use the symbol db_cp for a currency pointer reference; the actual variable name may be a record_name, a link_set_name, an area_name or the name of the current procedure.

Database Manipulation The statement types to be available for manipulation of a 1978 CODASYL database are given in Table 9-4. Specific formats for the COBOL language are given in the documentation for COBOL of the CODASYL committee, and FORTRAN versions have also been specified.

Each executable statement has a numeric code. The number is used when the manipulation functions of a DBMS are invoked by one of the different host languages.

Table 9-4 Declarations and Manipulation Commands for a CODASYL Database

/* *Obtain access to the relevant portions of a schema, its storage schema, and the contents of the database defined by it by invoking a subschema, using* COBOL *

DB sub_schema_name WITHIN schema_name [; ACCESS CONTROL KEY = xxxx].

LD keep_list_name LIMIT IS integer. /* *to keep currency indicators* */

/* *Transaction control* */		
13	`READY`	lock areas as specified in the `AREA` clause of the schema.
01	`COMMIT`	release record locks and reset all currency indicators and `keep_lists`. The statements coded `02, 03, 04, 11, 12, 15,` and `16` lock the records and link-set entries accessed.
06	`FINISH`	release locked areas.
09	`IF ...`	test database status and error condition codes.
14	`ROLLBACK`	remove all database changes since `READY` or `COMMIT.`
/* *Finding and manipulating records* */		
05	`FIND`	locate a record.
08	`GET`	obtain specified data items or all of the current record.
15	`STORE`	insert a record according to the schema specifications.
11	`MODIFY`	update the current record.
04	`ERASE`	delete the current record.
10	`KEEP db_cp`	obtain a currency indicator and place it into a `keep_list.`
/* *Manipulation of link-sets* */		
02	`CONNECT`	establish `MANUAL` link-set membership.
03	`DISCONNECT`	remove a record from link-set membership.
16	`RECONNECT`	move a record from one link-set to another link-set.
/* *Other* */		
07	`FREE db_cp`	release currency indicator, including any entries kept for currency indicator `db_cp` in a keep_list.
12	`ORDER`	sort the members of the current set logically so that they can be retrieved in a certain order.
	`USE ...`	declaration to identify procedure to be executed when an exception or error condition occurs and to identify access control procedures.

Program Flow Control The standard COBOL statements that permit conditional expressions (`IF, EVALUATE, PERFORM, SEARCH`) can also refer to database conditions. The three condition codes which permit testing the database are named:

`TENANCY` Is the current record an owner or a member, or either (a tenant) of a specified link-set?

`MEMBERS` Is the specified link-set empty, i.e., are there any member records?

`DB-KEY` Is the currency indicator of a found record the same as a previously obtained currency indicator, or does it match one kept within a given `keep_list`?

end ninepoint

There are five predefined registers to provide information when an error occurs: `DB-STATUS` gives a numeric code defining the error, as shown in Example 9-19; while `DB-SET-NAME`, `DB-RECORD-NAME`, `DB-ACCESS-CONTROL-KEY`, and `DB-DATA-NAME` contain character strings defining the current position in the database.

9-5-3 Finding Records

The `FIND` command is the principal means of *navigation* through the network. A `FIND` command does not cause the actual record to be put into the user's program area but does change relevant currency pointers. After a `FIND` command locates a record which contains desired data, some or all of its datafields may be retrieved by the execution of a `GET` command. The `GET` command specifies which data fields are required or if the entire record is to desired. The data is copied from the record indicated by the currency pointer into the program area associated with the `GET` data. To obtain another record a new `FIND` statement has to be executed.

Example 9-19 Finding Records with Link-Sets

```
 READY personal_data.                /* Start transaction, lock an AREA */
   FIND FIRST WITHIN Our_employees.  /* Use link-set order based on ssn */
 next_parent.                        /* 02100 indicates end of link-set */
    IF DB-STATUS EQUALS 02100, THEN GO TO no_more_employees.
      GET name.                      /* of potential parent */
      MOVE name TO name_field IN print_line.
  next_child.                        /* 02300 indicates empty link-set */
      FIND NEXT Children WITHIN Parenthood.   /* if any or if any left)*/
      IF DB-STATUS EQUALS 02100 OR EQUALS 02300, THEN GO TO done_parent.
         GET age_c.                  /* There is one or another child */
         MOVE age_c TO age_field IN print_line.
         WRITE output FROM print_line AFTER ADVANCING LINE.
         GO TO next_child.
  done_parent.
      FIND NEXT WITHIN Our_employees. /* Use link-set order based on ssn */
      GO TO next_parent.
 no_more_employees.
* End of parent, child age listing.
```

COBOL statements to retrieve ages of the children for all employees are shown in Example 9-19. We assume a subschema corresponding to Example 9-18 has been invoked. The program finds all records by traversal in link-set order. For `Employees` without children, no link-set will be found and their `name` will be ignored.

‡‡‡‡‡ Navigation through a CODASYL structure can become much more complex than the simple hierarchical traversal shown in Example 9-19. There are in fact 10 conceptually distinct ways to locate a record, as shown in Table 9-5. Some of the choices have similar syntax, and given specific options may in fact carry out an identical process. Once programs use certain ways to retrieve records, it becomes difficult to change the database structure. It is hence very important to design the initial database with the utmost care.

Table 9-5 How to Find a Record

1 Locate a record using the link-set order. Requires `KEY` and `ORDER`:

```
FIND{FIRST, NEXT}record.
```

2 Locate a record with a matching key using a search argument value. Requires `KEY` and `ORDER`. The search argument is placed into the declared record area. The `NEXT` record is the next record with a matching key. Next is based on the link-set order and relative to the currency pointer.

```
MOVE value TO data_item IN record_name.
FIND{FIRST, NEXT}record USING data_item.
```

3 Using access by a key with a prespecified search argument value. The storage schema typically indicates a direct storage (`CALC`) organization:

```
MOVE value TO key_name IN record_name.
FIND ANY record.                   for the first of any duplicates and
FIND DUPLICATE record.             for successors
```

4 Using a link-set of the current owner to find any, or a specific type, member record, `NEXT` may be the first member of the link-set:

```
 FIND{FIRST, NEXT, PRIOR, LAST, integer, variable}[record]
     WITHIN link_set.
```

5 Using an argument to locate a member of the current link-set by attribute value:

```
FIND record WITHIN link_set CURRENT USING search_arg_value.
```

6 Using an argument to locate further link-set members having the same attribute value:

```
FIND DUPLICATE WITHIN link_set USING search_argument_value.
```

7 Using a `SET SELECTION` clause from the schema to determine the link-set instance for the record type and an argument to locate the link-set member by attribute value (see Example 9-22):

```
FIND record WITHIN link_set USING search_argument_value.
```

8 Using the pointer-value (TID) sequence for references within the storage area, for all or for specific record types:

```
FIND{NEXT, PRIOR, FIRST, LAST, integer, variable}[record]
     WITHIN area_name.
```

9 Using a link-set to locate the owner of a current member record:

```
FIND OWNER WITHIN link_set.
```

10 Using a currency pointer as defined in Sec. 9-5-2:

```
FIND db_cp.
```

We will now illustrate some specific cases of navigation with the `FIND` command. Programs are written to access data according to the structure specified in the schema. The use of a database system as specified by the DDLC does not eliminate programming in this respect, although it greatly simplifies the task of dealing with the detailed complexity of network structures.

Using Keys If a key has been defined in the record definition of the schema, a record may be found by using a search argument to match the key. Precisely how the record is found depends on the available physical structure defined in the storage schema; the basic choices are sequential, indexed, or direct. Section 9-5-4 will deal with these implementation structures. To retrieve ages of the children for a particular `Employee`, the COBOL statements would be as shown in Example 9-20.

Example 9-20 Use of a Key

```
To find the age of the children of Employee having the key in parent_ssn:
  find_employee.
       MOVE parent_ssn TO empl_key IN Employee.
       FIND Employee.              /* Find matching social security number */
       GET name.
* Continue to MOVE name and next_child. to FIND Children as in Example 9-19.
```

Link-Set Usage for Networks The various `FIND` operations using link-sets provide the capability to traverse the network defined in the schema. All link-sets can be used to find the the `FIRST` or `LAST` entries. If a specific entry of a link-set is frequently needed, say, the most recent `Department_assignment` of an `Employee`, the link-set may be ordered `INSERTION LAST`. Now the `LAST` entry will correspond to the current and most needed one. The corresponding storage schema (see Fig. 9-14) should provide a `POINTER FOR PRIOR` and a ring where the `POINTER FOR LAST` member links to the owner.

The order of link-set entries can also be used to limit the search to part of the ring, for example to all young `Children`. The option `RANGE KEY` provides a warning that retrieval may occur by range. This means that not only exact matches for a given key, but also `GREATER` or `LESS THAN` matches have to be supported by the links.

Any link-set can also be used to find an `OWNER` when given a `MEMBER`. One important application of this function is to find the other owner of a member in an association. One link-set is used to find the shared member, and then the other is used to find the other owner. In a relational approach the equivalent operation will use two joins. Both owner relations are joined with their association relation, on different attributes. If finding owners is done frequently, a `POINTER FOR OWNER` specification will maintain a direct pointer from each entry to the owner record (see Fig. 4-41). This pointer would be advantageous in moving from an arbitrary member to its owner but is of no benefit if the `LAST` member or an entire subset is processed.

Link-set features are illustrated in Example 9-21, where one key and four link-sets of the given schema are traversed in order to provide the required listing. The output includes fields from three of the record types touched during the traversal. You may want to sketch a diagram to help you find your way during the journey.

Example 9-21 Traversal of a Network

We wish to list `Children` younger than 19, who now attend `"Harvard", of Employees` now working for the `"Foundry" Department`.
Of the two potential search arguments: `"Harvard"` and `"Foundry"`, only the second one can be used, since only the `Department` record has a `KEY`, so we enter the database there.

```
find_department.
  MOVE "Foundry" TO department_key IN Department.
  FIND Department.                                          /* by KEY */
 another_employee.
  FIND NEXT Worked_in WITHIN Department_assignments.
  IF DB-STATUS EQUALS 02100, THEN GO TO list_done.       /* No more workers */
* Test if this is the last Worked_in entry for this Employee
  FIND NEXT Worked_in WITHIN Employee_ties.       /* Ignore if record not  */
  IF DB-STATUS NOT EQUALS 02100, THEN GO TO another_employee. /* LAST */
 FIND OWNER WITHIN Employee_ties.          /* Employee currently in dept. */
    GET name.                          /* and potential Harvard parent */
      MOVE name TO name_field IN print_line.
next_child.
    FIND NEXT Children WITHIN Parenthood.
    IF DB-STATUS EQUALS 02100 OR EQUALS 02300, THEN GO TO another_employee.
      GET age_c.                   /* Parenthood is SORTED BY edu_key */
      IF age_c GREATER THAN 18, THEN GO TO another_employee.
        GET child.                 /* a potential Harvard student */
        MOVE child TO child_field IN print_line.
      FIND LAST Education WITHIN Children_education.  /* look only at last  */
      IF DB-STATUS EQUALS 02300, THEN GO TO next_child.  /* entry, if any  */
       GET Education.                /* entire last Education record for child */
      IF school_name NOT EQUALS "Harvard", THEN GO TO next_child.
* found one!
        MOVE school_year TO year_field IN print_line.
        MOVE subject TO subject_field IN print_line.
        WRITE output FROM print_line AFTER ADVANCING LINE.
    GO TO next_child.
*
 list_done.
```

Set Selection When the CODASYL database schema language was summarized in Sec. 8-3-5; the `SET SELECTION` clause was not described since it relates to the database manipulation process rather than to the database structure. Access choices to records specified under this keyword and other clauses related to link-set manipulation are now shown in Fig. 9-13. The statement `FIND` can use the `SELECTION` path specified in the schema to locate a record without specification of the path within the program.

`SET SELECTION` permits predefinition of the initial link-set or record key, and can specify a path through multiple levels of the structure. Again, a subsequent `GET` is needed to read the record, once it is found, from the file into core storage. Set selection also permits a subschema to ignore some intermediate level.

```
/* The connections to be implemented in a CODASYL schema are defined as follows */
  SET NAME IS link_set_name.
     OWNER IS {record_name_o / SYSTEM }.
     ORDER ⎧PERMANENT⎫ INSERTION ⎧FIRST / LAST / NEXT / PRIOR / SYSTEM DEFAULT       ⎫
       IS  ⎩TEMPORARY⎭     IS    ⎨SORTED ⎧WITHIN RECORD-TYPE                  ⎫   ⎪
                                 ⎪       ⎨BY DEFINED KEYS                     ⎬   ⎬
                                 ⎪       ⎪   [RECORD-TYPE SEQUENCE            ⎪   ⎪
                                 ⎪       ⎪       IS record_name, ... ]        ⎪   ⎪
                                 ⎩       ⎩   [DUPLICATES ARE  ... ]           ⎭   ⎭.
     MEMBER IS record_name_m  ...  as shown in Fig. 9-14.
   SET       ⎧THRU set_name_owner OWNER IDENTIFIED BY                          ⎫
   SELECTION ⎪   ⎧SYSTEM / APPLICATION                                       ⎫ ⎪
        IS   ⎨   ⎨KEY owner_key_name [o_key EQUAL TO parameter_1[, ...]]     ⎬ ⎬
             ⎪   ⎪SELECTION DEFINED FOR record_name_1                        ⎪ ⎪
             ⎪   ⎩   [ THEN THRU ... /*  other sets down the hierarchy  */ ] ⎭ ⎪
             ⎪BY PROCEDURE procedure_name_s                                    ⎪
             ⎩BY STRUCTURAL CONSTRAINT                                         ⎭.
/* Omitted are access procedures and escape calls. */
```

Figure 9-13 The DDL clauses related to data manipulation.

Use of set selection is especially useful if a member record is many hierarchical levels below the level of the owner of the link-set, since it allows members to be found without having to find intermediate-level records. Example 9-22 provides a simple case.

Currency variables of all implied records and link-sets are, however, defined as a result of set selection traversal, and this permits references to data along the selection path, as will be shown in Example 9-23.

Example 9-22 Use of Set Selection

If we had specified for the link-set `Parenthood` that the `Children` are to be found implicitly by stating in the schema with the `Parenthood SET`:

```
SELECTION IS THRU Employee OWNER IDENTIFIED BY KEY empl_key.
```

then the explicit `FIND` for the `Employee` in Example 9-20 is not necessary when searching for the `Children`.

```
what_age.
   MOVE parent_ssn TO Emp_key IN Employee.
 next_child.
   FIND NEXT Children RECORD OF Parenthood SET.
     IF DB-STATUS ...
     GET age_c.
     MOVE age_c TO age_field IN print_line.
* The print_line does not yet contain the parent's name; see Example 9-23.
     ...
```

Using Currency The use of the currency pointer associated with a database structural element `db_cp` implies a reference to a previously found position in storage. We continue Example 9-22 to illustrate a use of currency.

Example 9-23 Use of Currency Pointers

```
      ...
*  A Children record was found, and the parent's name is needed.
      FIND Employee.   /* Uses the currency pointer defined earlier */
      GET name.
      MOVE name TO name_field IN print_line.
      WRITE output FROM print_line AFTER ADVANCING LINE.
      GO TO next_child.
```

The `FIND` statement will return the navigator to the record defined by the currency pointer `Employee`, so that the data items in the record become available for a subsequent `GET` operation.

In order to deal with more than one record of a given record type, a currency indicator may be placed into a `keep_list` by the `KEEP` statement and used later for comparison or to reset the actual currency indicator.

The representation format of such a currency indicator is not specified, and the currency indicators will lose their validity when the program `COMMIT`s or terminates. During the execution of a transaction the currency pointers are to remain valid, even during update operations. A `keep_list` can be used to compare and manipulate TIDs as defined in Sec. 4-2-3.

Earlier versions of the CODASYL specifications defined a special, manipulable data type, called `DB-KEY`, to handle currency pointers. It was possible, but dangerous, to use these `DB-KEY`s to implement semipermanent reference lists for records.

Manipulation of Link-Sets In the description of the `MEMBER` clause of a link-set appear some parameters which specify the manner in which link-sets will be maintained. Figure 9-14 reiterates that clause; `INSERTION` of a `MEMBER` record may be `AUTOMATIC` or `MANUAL`, while deletion is constrained by a `RETENTION` clause to be either `FIXED`, `MANDATORY`, or `OPTIONAL`.

```
/* The connections to be implemented in a CODASYL schema are defined as follows */
  SET NAME IS link_set_name.
         ...
    MEMBER IS record_name_m
      INSERTION IS{AUTOMATIC / MANUAL}
      RETENTION IS{FIXED / MANDATORY / OPTIONAL}
      [DUPLICATES ARE NOT ALLOWED FOR key_attribute [, ...] ]
      [[RANGE] KEY IS{ASCENDING / DESCENDING}{key_at / RECORD-TYPE[, ...]} ]
      [         [DUPLICATES ARE{FIRST / LAST / NOT ALLOWED / SYSTEM DEFAULT}]]
      [         [NULL IS [NOT] ALLOWED]                                      ]
      [ STRUCTURAL CONSTRAINT IS variable_a EQUAL TO variable_b [, ...]].
```

Figure 9-14 The DDL MEMBER clause of a link-set.

Automatic insertion means that when the member record is STOREd the currency pointer for the current OWNER will be used to create an entry into the link-set. For instance, the link-set Parenthood to the Children relation was specified AUTOMATIC. A sequence of FIND Employee and STORE Children will establish the linkage.

Manual means that the user transaction program will have to execute CONNECT or DISCONNECT statements to manage the entries. The schema specified MANUAL insertion for the Supervision records in the link-set Employee_ties. Example 9-24 presents the statements needed to add a supervisee to Employee_ties. A MANUAL connection is appropriate here to avoid possible confusion since both owner and member can have currency Employee. To link boss and subordinate the program stores the record for the subordinate, establishes a currency indicator for the boss in the Employee relation, and then connects the two by inserting an entry into the link-set. The ORDER of the link-set was defined in the schema to depend on the KEY year.

Example 9-24 Adding a Supervisee

```
  Establish a linkage from supervisor identified by boss_ssn to subordinate
having  sub_name:
 new_subordinate.
    MOVE sub_name TO subordinate IN Supervision.        /* Create record */
    MOVE 1982 TO year IN Supervision.           /* Determines ORDER KEY */
    STORE Supervision.                   /* Place the record into the file */
    MOVE boss_ssn TO emp_key IN Employee.
    FIND Employee.
    CONNECT Supervision TO Employee_ties. /* OWNER determined by currency*/
```

The RETENTION clause FIXED indicates an ownership dependence of the member record on the owner of this link-set. When an owner record is deleted, its link-set disappears and the link-set's members alsodisappear. This is the expected constraint for the Children of the Employees and their Education.

A DISCONNECT statement will also cause the FIXED record to disappear, say, if a child becomes independent. To avoid losing the member record if a member has to change owners, a RECONNECT statement may be used. In our schema this might be necessary if Children had to be switched to another Employee parent. A more frequent case would occur if the schema would include a link-set Current_department from Department to a FIXED Employee member. This link-set would require the RECONNECT statement whenever the Employee switches Departments.

The option MANDATORY indicates an ownership dependence on the union of the owners of all link-sets that this record is a member of. This means a record is not necessarily deleted when the owner of one link-set disappears, only when there are no more owners at all. This choice is like a reference connection with garbage collection. We made membership of the Worked_in records MANDATORY for both owners. Now, if an Employee record disappears, the name of such an Employee can still be found from the Department record and vice versa. A search from that record to find employee data will fail, however. The `Worked_in` records will be deleted if the `Employee` disappears and the `Department` is dismantled.

The choice `OPTIONAL RETENTION` means that the member records exist independently of their link-set owners. This clause is appropriate for entity and referenced entity relations. It is wise to have a `KEY` field in such a record to avoid making the record inaccessible while manipulating link-sets.

9-5-4 File Organization in a CODASYL system

The storage schema, following the definitions given in Fig. 8-10, specifies alternatives for the implementation of records and link-sets. It is not intended to be restrictive; other implementation techniques which provide identical functionality are permitted. The writer of a storage schema has to be very careful that the model schema is appropriately supported. We can expect that software tools will be developed to help create efficient storage schemas for model schemas and estimated file sizes and access frequencies.

There are quite a number of restrictions in the model schema definition to prevent requiring implementations which are awkward or inefficient. There are also some restrictions on data manipulation which are implicitly derived from the storage schema. The intent of the storage schema is to make the model schema user independent of physical placement considerations. The restrictions will make sense to a user who understands file storage alternatives, but will appear capricious to someone without such insight.

For the `PLACEMENT` of records five alternatives are specified in the storage schema: direct, close to link-set members, close to owner records, close to some other records, and sequential. The implementation of these methods was covered in Chap. 3, and only a quick summary is given here.

Direct (`CALC`) The records will be placed into the `AREA` for direct access according to a hashing algorithm operating on the `KEY` of the record. The `KEY` in the model schema lists the attribute fields. A lexicon typically uses direct access for the most frequent access path. If access according to either attribute is required, a redundant copy can be kept, or the other attribute of the lexicon can be accessed using a link-set. Here duplicates would not be allowed.

Direct access is common for the records at a top level of some hierarchy. For entity relations serial access is generally needed as well. If an `ASCENDING` or `DESCENDING` sequence is specified in the model, further records can be obtained serially. This may require the additional availability of a link-set with an `ORDER` clause, probably owned by the `SYSTEM`.

Clustered `VIA SET` The records that are members of the same link-set (`SET`) will be placed in the same or adjoining blocks. Finding the next member of a link-set will then be fast.

Clustered `NEAR OWNER` The records that are members of the same `SET` instance will be placed in the block or in blocks adjoining the block which contains the specified type `OWNER` record. This implies, of course, that the `OWNER` records themselves will have poor locality.

Clustered `VIA SET ... WITH ... .` The records that are members of the same link-set (`SET ...`) will be placed in the block or in blocks adjoining the block which contains the specified record type (`WITH ...`).

Placement is `SEQUENTIAL` The records of this type will be placed so that they can be efficiently retrieved for serial access. The ordering will be according to the named `identifier`.

The organization of the link-set structure is also specified in the storage schema; the `ORDER`ing of the link-set members is specified in the model definition. The essentials of the 1978 storage schema definition for link-sets are given in Fig. 9-15. In earlier DDLC documents similar specifications appeared as the `MODE` of the link-set. In the 1978 schema the choices are basically limited to two alternatives:

Access members using an `INDEX` Use of an in a link-set index is appropriate if the number of member records for a link-set is large. It is used predominantly for `SYSTEM`-owned records.

Access members using a chain This choice defines the common ring structure. Options include either all or some of forward, backward, and owner pointers, as well as a choice of chains or rings. The `POINTER FOR` clause may be repeated as often as necessary.

Within each record the actual data fields and the required linkage fields have to be declared. The pointers in the linkage fields can be specified to be `DIRECT` or `INDIRECT`, implementing the two nonsymbolic alternatives shown in Fig. 8-2.

```
     The SET clause describes a link_set connection in the storage schema:
SET SCHEMA_SET   [ALLOCATION IS {STATIC / DYNAMIC }]
    POINTER {   INDEX index_name                                        }
      FOR   { { [NEXT] [PRIOR]  [OWNER]                     [ RECORD   } }
            { { [LAST] [FIRST] /* connects to owner  */                } }
            { {  schema_record_name] {IS [DIRECT][INDIRECT]            } }
            {                        { TO storage_record_name_memb, ...} } .

/* In each storage record type each member and owner pointer has to be defined */
STORAGE RECORD NAME IS storage_record_name_1
    [LINK TO storage_record_name_2 [ IS{DIRECT / INDIRECT}]] [, ... [...] ]
     [RESERVE integer_5 POINTERS]     /* spares for later record expansion */
```

Figure 9-15 DDLC-DSDL linkage implementation specification.

9-5-5 Design Considerations

The many options available for CODASYL databases can make the design process complex. The opportunities for restructuring of a database are quite limited once a design is implemented and programs have been written to use the database.

One consideration in database design is whether linkages in the records should replace attributes from the conceptual tuples. If attributes are omitted, certain access choices are no longer available.

Structural linkages to the `Children` relation of Example 9-18 are *essential* since the `parent_name` is not kept in the record. Link-sets using `OWNER` or `NEXT` and `LAST` pointers are required to find the parent. If the reference value `parent_name` is also put into the `Children` record, retrieval of the parent can also be carried out by a fetch to `Employee`. The linkage is redundant with the value. To preserve integrity of the databases links and values must match. The maintenance of such integrity constraints is left to the user's update programs.

Major reorganizations of data relationships (e.g., reversal of an owner-member connection) will cause problems even when structure independence has been carefully considered. An example of such a reversal would be a decision to maintain lists of patients of a clinic according to the problems they have presented (maybe to improve medical management), whereas these patients were traditionally seen as individual entries in an entity relation, and their problems were handled as a nest relation.

It is possible within the CODASYL specification to design service programs and applications that maintain databases according to rigorous standards. This can be achieved if management is willing to support a well-defined conceptual model. This means setting and enforcing design and programming standards. Use of the access procedures can help provide some enforcement, although many DBMSs do not yet have complete implementations. Some storage and update costs will be incurred if increased redundancy is required.

The description of the DBTG proposal given above is intended only to convey the flavor of the proposal. The full set of specifications exceeds 400 pages but is easily read if the objectives, structure, and underlying fundamentals are understood, although some ambiguities remain. We will now discuss some of the actual implementation issues.

9-5-6 Current Implementations

Many CODASYL systems are now available, although no system has implemented all the features. The richness of the specification makes it difficult to provide a full system on a small computer, but implementations for large minicomputers are also in progress. As indicated earlier, most current CODASYL DBMSs are based on earlier specifications. They also may vary, as implementors became aware of problems with the initial specifications and devised their own solutions.

Using Subschemas An external schema, to be included with the compilation of programs accessing the database, includes only the file-organization specifications required for data manipulation. A CODASYL subschema, for instance, does not include information from the storage schema. Optionally, further elements (data attributes, records, link-sets, and areas) may be omitted to force programs to be independent of certain information. Such programs will be less affected if a portion of the database is reorganized. In order to provide subschemas for the programmers to use, an `INVOKE` command is available for COBOL which will copy the selected subschema from a file maintained by the database administrator:

```
INVOKE SUBSCHEMA production_schedule OF SCHEMA order_entry.
```

A single program can use only one subschema, so that a large number of separate subschemas may be developed if access privileges are to be controlled using subschemas. Programs which are under direct management control can be given access to more features via a more comprehensive subschema.

Data-Manipulation Statements One system, Cullinane's IDMS, originally developed by B.F.Goodrich, implements the CODASYL architecture on a variety of computers. Table 9-6 summarizes the statements available to a COBOL user for manipulation of the database. We note with the statements variances with the 1978 specifications when the difference is confusing or critical.

Table 9-6 IDMS Statements

```
Control Operations:
   INVOKE SUBSCHEMA ...      .
   OPEN AREA realm_name USAGE MODE IS protection_clause.
   CLOSE ALL AREAS.
   IF link_set SET EMPTY GO TO label.          /* check DB-STATUS */
   IF RECORD MEMBER OF link_set SET GO TO label. /* compare db_cp */
   MOVE STATUS FOR ... TO variable.              /* keep currency */
   CALL data_base_procedure ON operation.
 Operations on Records and Link Sets:
   STORE record_name RECORD.
   FIND ...     .                      /* as described in Table 9-5 */
   GET record_name RECORD.             /* always the entire record  */
   OBTAIN ...   .                      /* combines FIND and GET     */
   MODIFY record_name RECORD.
   DELETE record_name RECORD ...    .
   INSERT record_name INTO link_set.   /* CONNECT    */
   REMOVE record_name FROM link_set.   /* DISCONNECT */
```

For IDMS users who are programming in FORTRAN, BASIC, PL/1, or assembly language `CALL` statements are available to carry out the functions provided.

The Storage Schema IDMS is based on the 1972 DBTG specification, but it already defined a schema language augmented with statements which control the assignment of the database to the storage devices. IDMS translates all database keys, both the internal DBMS pointers and the users' `DB-KEY`s, into indirect references to the blocks of the direct access files. One file can contain multiple `AREA`s and each `AREA` provides blocks which are packed with multiple record types as shown in Fig. 4-33. To avoid block overflow, a new block will be assigned when a record does not fit. To provide the control mechanism for the physical storage assignment, a device and media control language (DMCL) is defined which interacts with the stored schema. The process of schema translation is as shown in Fig. 9-16, which may be compared with the processes described in Sec. 8-4-2 (Figs. 8-10 to 8-13).

During execution the database handler can be dedicated to one user or can be shared among multiple users. Users will have a copy of the subschema table relevant to them in their own section of core storage, as well as a copy of the IDMS modules which accept the `CALL` statements created during the process of translation of the database-manipulation statements to COBOL statements (A). UNIVAC has an implementation based on the earlier 1969 DBTG report, DMS1100, which adds two statements to make and break the subroutine linkages between the user's executing program and the database handler: `IMPART` and `DEPART` (B).

An example of statements used for storage assignment by IDMS is given in Table 9-7. The allocation of the physical devices is completed by using operating system control language statements referring to `sys009, sys010, sys011`. The defined ranges describe the high-order digits of the database pointers which will be used. The numbers assigned within the files are the relative block addresses to be used for each `AREA`. The size of a block is described in the `PAGE` statement.

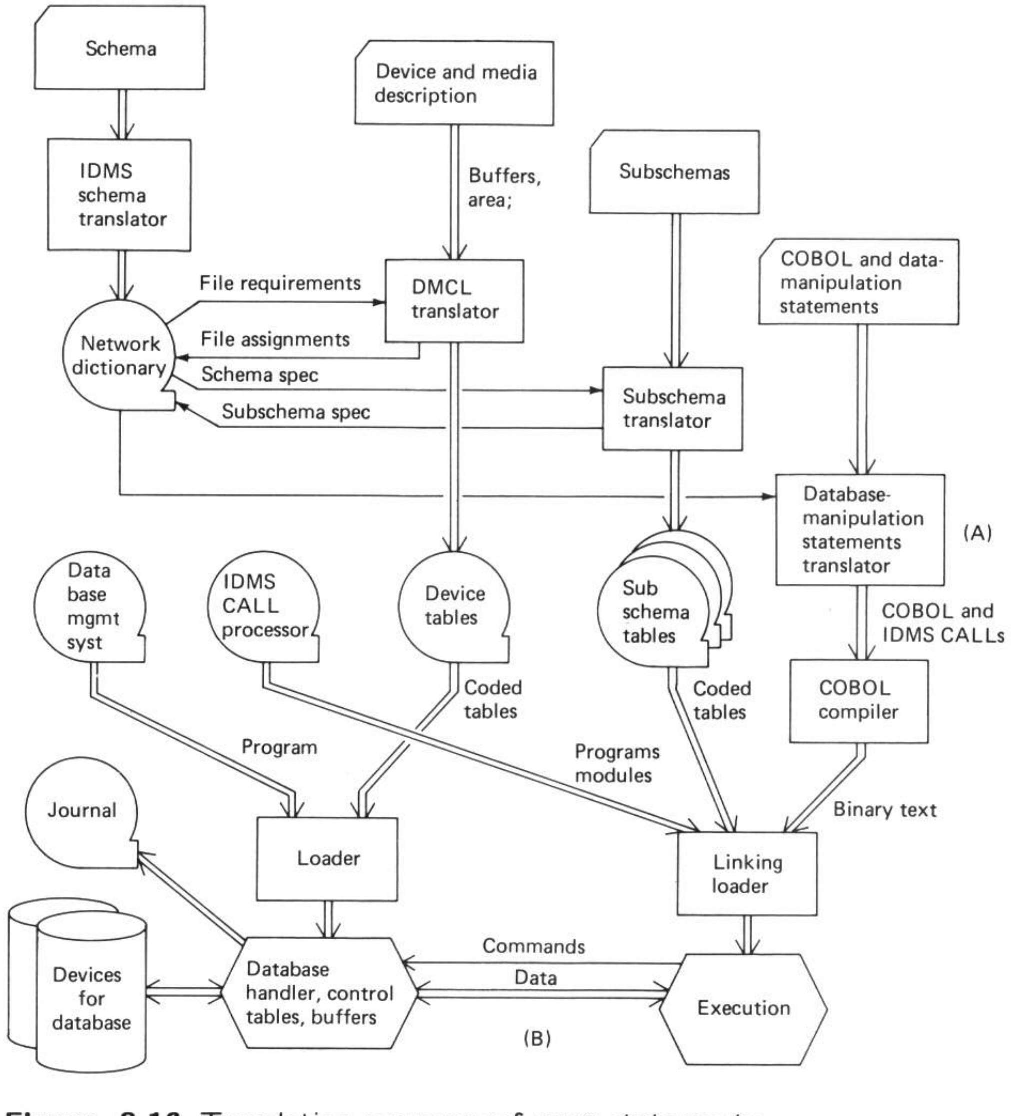


Figure 9-16 Translation sequence of IDMS statements.

Table 9-7 Storage Assignment Statements in IDMS

```
/* In the Schema */
 FILE DESCRIPTION.
   FILE NAME IS IDMS_file_1 ASSIGN TO sys010 DEVICE TYPE 2314.
   FILE NAME IS IDMS_file-2 ASSIGN TO sys011 DEVICE TYPE 2314.
   FILE NAME IS journal ASSIGN TO sys009.
 AREA DESCRIPTION.
   AREA NAME IS customer_area  RANGE IS 1002 THRU 1100
                     WITHIN FILE IDMS_file_1 FROM 1 TO 99.
   AREA NAME IS order_area    RANGE IS 1101 THRU 1300
                     WITHIN FILE IDMS_file_1 FROM 100 THRU 199
                     WITHIN FILE IDMS_file_2 FROM 111 thru 120.
   AREA NAME IS product_area  RANGE IS 301 THRU 1310
                     WITHIN FILE IDMS_file_2 FROM 1 THRU 10.
/* In the Device and Media Description */
 BUFFER SECTION.     BUFFER NAME IS IDMS_buffer
               PAGE CONTAINS 1508 CHARACTERS
               BUFFER CONTAINS 5 PAGES.
 AREA SECTION.       COPY customer-area AREA
               COPY order_area AREA
               COPY product_area AREA.
```

9-6 INTERLINKED HIERARCHIES

The final database implementation to be described is one of the earliest to be used for commercial or industrial use. IMS was originally conceived at North American Aviation in order to solve the bills-of-materials problem in a large company with strong central management. The system has been extensively developed by IBM, and IMS/360 and IMS/VS are now principal database products of that company. IMS includes also a scheduler and communication interface, so that it can be used on-line within batch-oriented systems. Database schema and manipulation support is mainly provided through the DL/1 subsystem.

The IMS system is relatively unique in its approach. In its storage representation it is strictly hierarchical, so that a database schema will again be composed of a forest of schema trees. Various storage alternatives are available for the entity and nest relations. The schema trees, however, can also be linked to each other, and alternate schema tree definitions can be formed using the stored segments. The aggregate conceptual structure can become more complex than DBTG networks.

The programmer, however, does not have direct access to these links. The links are used to construct alternate hierarchies. Access to an alternate external or *logical* hierarchy is made possible through the interposition of DBMS procedures which use the available links to obtain the appropriate segments. In order to obtain data the programmer specifies some hierarchical view and uses only hierarchical operations to navigate through the database. The actual database structure may remain hidden.

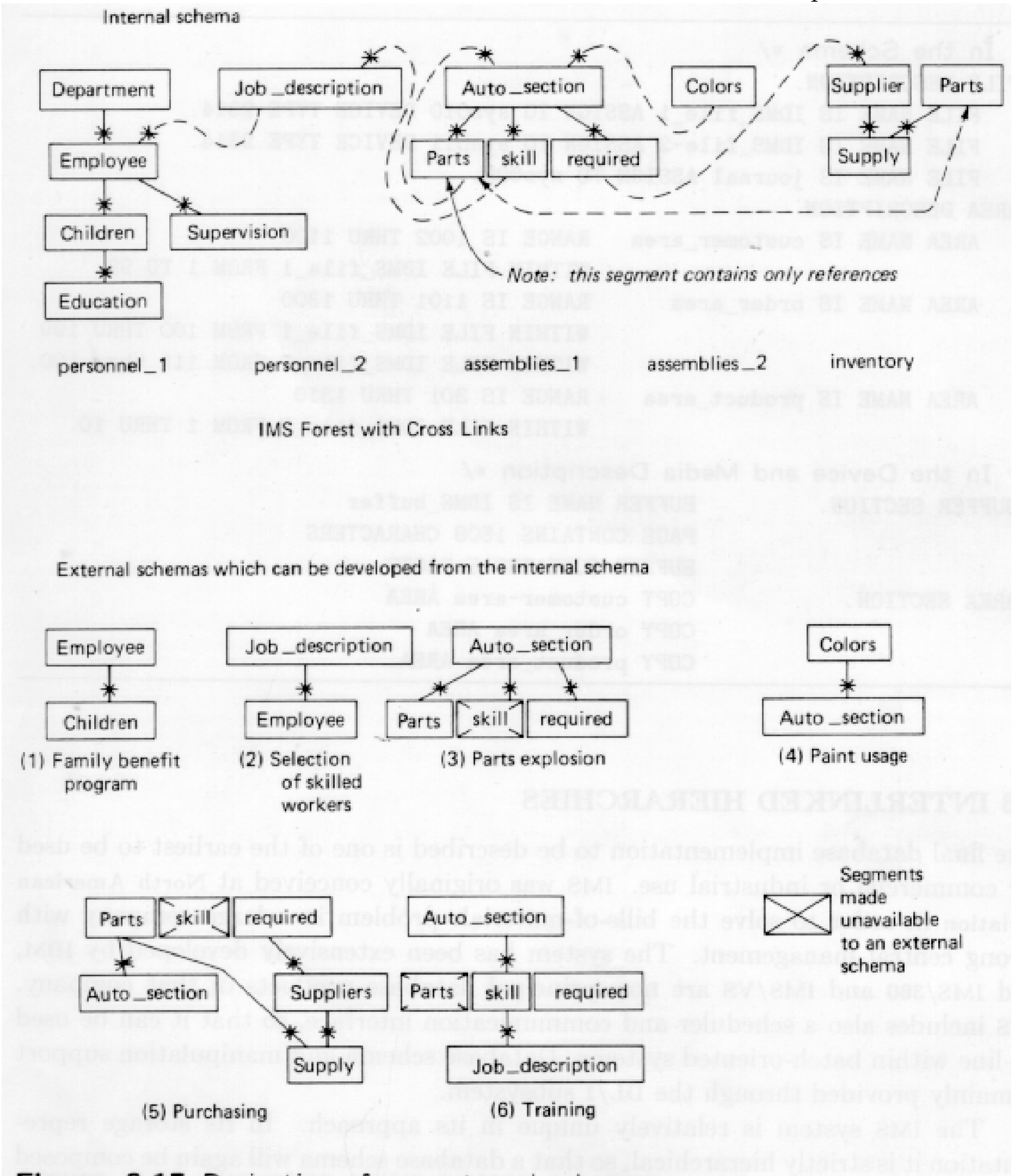


Figure 9-17 Derivation of IMS external schemas.

The hierarchy of the external schemas means that in IMS the accent is still on the retrieval of a composite record. A record consists of current segments at each level of the selected hierarchy. A database administrator is required to maintain the overall structural specifications and their implementation.

An organizational separation of application and database system staff categories is assumed. A third category consists of the users of the programs, who enter and retrieve data. The administrator has to be able to maintain an integrated database which allows concurrent existence of the various logical hierarchies. The complexity of this task is such that considerable programming and management resources are required when IMS is used.

9-6-1 The IMS Schema

The schema language for an IMS database is DL/1 as described in Sec. 8-2. DL/1 is used in another mode to provide external logical schemas for the users. Each external schema can depict a hierarchical structure appropriate to the user's data model, subject to the existence of relations and linkages in the internal database schema. Procedures within IMS use the linkages in the schema to transform the data from the database relations into the data presentation specified by a DL/1 subschema. Numerous structural restrictions exist in regard to the permissible transformation, but as IMS developed these restrictions have become less severe. The *logical* view of the database, described by the user's external schema, is neither a conceptual view of the database or a database subset, nor a view of physical reality, but rather an imitation of a physical view of a database subset.

In the joint schema all structures depicted form a forest of entity and nest relations. A hierarchical structure for the database used earlier (see Figs. 7-27, 9-5, and 9-10), suitable for a DL/1 internal schema is shown in Fig. 9-17, together with some possibilities for external schemas.

Each external subschema is in itself logically hierarchical. The choices of subschema structures are limited by the predefined linkages in the internal schema. Linkages between trees can implement *owner, member, or sibling* relationships, and can be specified to be *next*, or *next and prior*. In addition, pointers are used to link nests into rings or to link overflow areas to records as required by the file organization chosen. All the linkages to be implemented require, of course, specifications in the DL/1 internal schema, storage space in the files, and updating effort during database use. The trade-off between linkage availability and excess redundancy has to be made carefully. When frequency of usage does not warrant the maintenance of linkages, symbolic references can still provide access to other files.

An external schema is defined using the same DL/1 mechanism used for the internal schema. `SEGMENT` definitions obtain position and size information from the database and the segment names of the internal schema. `FIELD` definitions are completely ignored here. For the `parts_explosion` external schema the definition might read as shown in Example 9-25.

Example 9-25 DL/1 Logical Hierarchy for the External Schema

```
DBD      NAME=explosion,ACCESS=LOGICAL
DATASET  LOGICAL
SEGM     NAME=section,SOURCE=((auto_section,,assemblies_1))
SEGM     NAME=all_about_parts,PARENT=section,
    SOURCE=((parts_skill_required,,assemblies_1),(parts,,inventory))
DBDGEN
FINISH
```

The segment `all_about_parts` consists of the catenation of two source segments, so that this associative relation is formed without actual duplication of entries from the `parts` owner. This is possible because of the existence of *logical sibling* pointers in the `parts_skill_required` segments.

A lexicon can also be composed of two logical sibling segments, so that access to the relation can use either of the two attributes, but the access is not symmetric in performance. Access procedures, however, using either one of the two attributes, given that the lexicon is properly specified in two external schemas, can use the same operation formats.

9-6-2 Operations on the Logical Hierarchy

IMS procedures are initiated by executing `CALL` statements. The argument of a `CALL` is a parameter area containing the specifications to be communicated to IMS. Three operands are specified with each operation:

1 The type of *operation* to be performed

2 The *object* of the manipulation

3 The user's *data area* for the retrieval results or update values

Operations The permissible operations are given in Table 9-8.

Table 9-8 IMS Data Manipulation operations

`Get_Unique`	for a fetch according to a key adequate to identify segments up to the desired level in the hierarchy
`Get_Next`	for a sequential read to a successor segment or segment sequence
`Get_Next_with_same_Parent`	for a read within a nest
`InSeRT`	to add a segment
`DeLETe`	to delete a segment
`REPLace`	to replace a segment

Locking options for these statements are discussed in Sec. 13-1-1.

Objects The objects to be manipulated are described by reference to another DL/1 construct, the *program communication block* or PCB. This table, generated also through an assembly process, specifies the segments from the program point of view, references the logical hierarchical structure given in the external schema, and controls the access privileges. A PCB which uses the external schema `Training` (see Fig. 9-19) could read as shown in Example 9-26. The parameter `PROCOPT` specifies that segments may be retrieved (`G`), inserted (`I`), replaced (`R`), or deleted (`D`).

Example 9-26 DL/1 Schema Selection

```
PCB     TYPE=DB,DBDNAME=Training,KEYLEN=34
SENSEG  NAME=section,PROCOPT=G
SENSEG  NAME=job_descr,PROCOPT=GR
```

It is also possible for the program's PCB to refer to the internal schema directly if no (or no appropriate), external schema was defined. Since not all segments of the external schema have to be included in a PCB, the PCB provides a further subsetting capability for the database: in Fig. 9-19 `skill` data are omitted from the `Parts_explosion`. A program may use multiple PCBs if it wishes to use more than one external schema simultaneously. This allows the use of multiple records, since each external schema defines one record, constructed of current segments in a hierarchy. A currency indicator is associated with every PCB in use. The individual data fields are not subject to DL/1 control; the units *retrieved, inserted, deleted, or replaced* are single segments or a string of multiple segments, each segment at a different hierarchical level. We will call the segment at the highest level the *root segment*.

In order to reference a lower-level segment in a hierarchical structure, a sequence of keys is required. A key for a segment to be fetched is the catenation of the keys of all ancestor segments beginning with the root segment and ending with the key of the goal segment. The maximum total key length is given as `KEYLEN` in the PCB. To get data regarding the supervised employee `Mike`, the fully qualified search key would be

```
    (dep="Assembly").(employee="Hare").(supervision="Mike")
```

These keys are presented to DL/1 in a compact form. When the computation retrieves a successor segment using the `GET-NEXT` operation, DL/1 provides the key.

Data Areas The data input or output area for the program is a conventional PL/1, COBOL, or assembly language data structure, and the names given to the fields by the programmer are the actual data element names used in computational manipulations. The type, unit, length, etc. of the elements remain fully under control of the program which has access to the segments containing the data. Segments not included in the PCBs, however, do not appear to exist at all. The programmer has hence to construct data areas where the size of the segments is determined by the internal schema, but which segments will appear in the data area is determined by the external schema and by the entries in the PCB. The order of the segments is determined by the external schema. The programmer ignores the linkages; these will not appear in the program data area. The size and content of key fields must be calculated in a similar manner from the field specifications. If alternate segments are possible, such as `children.education` or `supervision`, the record and key fields have to be dimensioned according to the maximum length, and overlapping data structures will be used using the `REDEFINES` clause in COBOL and `BASED` variables in PL/1.

9-6-3 Storage and Linkage

In order to support the hierarchical structures, IMS uses a mapping procedure, so that all data for an entity relation and all its nests appear in a single file. There are hence as many files as there are trees in the IMS forest. Each top-level entry in a file, for instance, each `department` of `Personnel_1` in Fig. 9-17, is the root segment for a *tree instance*. Each tree instance, with all its subsidiary segments, is treated as if it were a long and complex record. Such IMS trees can span many blocks.

The file structures used by IMS are extensions of standard file organization methods. Extensions were needed in order to serve the hierarchical nature and potentially large size of the tree instances.

The four file choices for IMS are

HSAM	Sequential
HISAM	Indexed sequential
HDAM	Direct access
HIDAM	Indexed direct access

We will first present how entire trees instances are placed into blocks, and then see how segments and blocks are manipulated within these files.

Mapping of the Hierarchy The mapping of the segments comprising the hierarchical tree is

top-to-bottom, then left-to-right

or *preorder* using the terminology used by Knuth[73F]. This order was also used by OASIS; see Fig. 9-9. One tree instance, when completed, will be followed by the next tree instance, until all entity tuples and their nest members are stored. For the `Personnel_1` tree of Fig. 9-19, using the same data as the SYSTEM 2000 files of Fig. 9-8, the sequence would be as given in Example 9-27.

Example 9-27 Segment Storage for One DL/1 Record

```
department(1),employee(1,1),child(1,1,1),education(1,1,
1,1),education(1,1,1,2),education(1,1,1,3),supervision(
1,1,1),supervision(1,1,2),supervision(1,1,3),supervisio
n(1,1,4),employee(1,2),child(1,2,1),supervision(1,2,1),
supervision(1,2,2),...,supervision(1,2,6),employee(1,3)
,child(1,3,1),education(1,3,1,1),child(1,3,2),education
(1,3,2,1),education(1,3,2,2),supervision(1,3,1),supervi
sion(1,3,2),employee(1,4),...,supervision(1,5,8)
```

A new record begins with the next root segment: `department(2), ...` .

In Example 9-27 the first sequence of four segments provides one complete flat record. A successor record according to the lowest level hierarchy will use `education(1,1,1,2)` instead of `education(1,1,1,1)`. An alternate successor record can be built from `dep(1), employee(1,1), supervision(1,1,1)`. Successor records are constructed by replacing low-order segments with their successors, using an algorithm reminiscent of reconstitution of an index key after rear compression (Sec. 4-2-2). Sequential processing allows any complete record to be available without ever backing up.

Linkages The links which provide the capability for the derivation of subschemas do not come free. Every nested segment which can logically exist in some secondary external schema requires in the corresponding actual file an entry at its apparent position containing the reference pointers to the real segment. A segment which implements an association will contain the dependent data and links to the logical second owner.

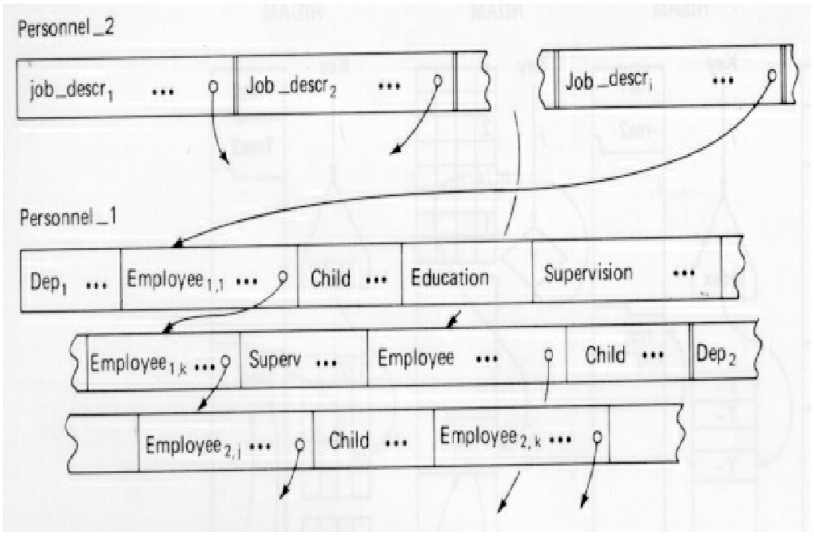


Figure 9-18 Logical pointers between segments.

For the personnel trees in Fig. 9-17 the files will contain segments with pointers as shown in Fig. 9-18 given that `employees(1,1),(1,k),(2,j),...` all have `job_descr(i)`.

Files with many logical linkages will hence be in practice much larger than they appear to one user. The requirements for linkages are specified in the DL/1 internal schema through segment entries for logical members (`LCHILD`) and segment options for siblings (`LTWIN`), owners (`LPARNT`), and catenations of two segments only (`PAIRED`) for associative relations. It is also possible to specify prior linkages allowing, for instance, owners to be found from members segments.

It should be noted that these linkages allow the reversal of hierarchical relationships in different internal schemas for the same database. This is a facility not commonly available in database systems.

File Structures The file structures used by IMS are not identical to the standard methods provided by IBM for the 360 systems, since these were insufficient. This is only one of many instances where inadequate file facilities have added considerably to the cost, complexity, and inadequate structuring of database systems. Figure 9-19 sketches the four approaches used by IMS. The storage linkages shown in the sketches are supplementary to the linkages used for the data models.

The basic unit to be stored, a DL/1 *record*, is a tree instance with one root segment and all its nested segments in *preorder* sequence. Such a tree instance can be very large and may span many blocks. Segments are not spanned. The entire tree instance does not have to be read into core storage at one time but is accessed segment by segment, as was shown above. Since access is based on the key of the root it is often wise to shorten the hierarchy by decaptation. This creates more and smaller trees. In the database laid out in Example 9-27 we may move the `department` level to a referenced relation, and use `employee` as the root.

HSAM The sequential file organization is available in IMS to allow tape or disk storage for files where mainly sequential access is expected. The fetch of a tree instance based on an arbitrary search key will be very time-consuming. The only

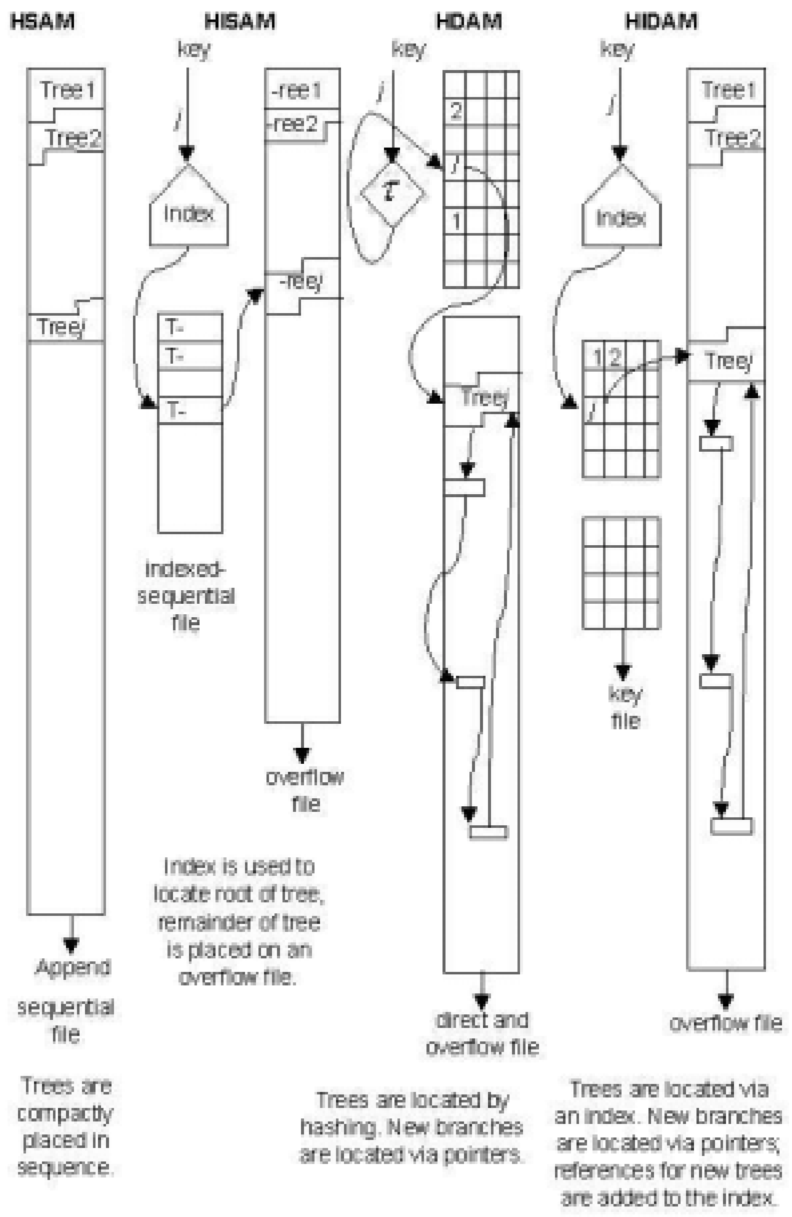

HSAM
HISAM
HDAM
HIDAM
key
key
key
Tree1
Tree2
Treej
Index
indexed-sequential file
overflow file
Append sequential file
Trees are compactly placed in sequence.
Index is used to locate root of tree, remainder of tree is placed on an overflow file.
direct and overflow file
Trees are located by hashing. New branches are located via pointers.
key file
overflow file
Trees are located via an index. New branches are located via pointers; references for new trees are added to the index.

Figure 9-19 IMS file methods.

dynamic updating possible is the addition of trees to the end. All other updating is to be performed by batch programs which copy the file.

HISAM The indexed-sequential ISAM or VSAM have been extended with an overflow file to permit the management of the large and variable-length tree instances. The HISAM method is suitable when sequential access, indexed access, and some updating are required. The initial loading is done by a batch process, using an input file which contains the segments in the desired *preorder* sequence.

Only the initial block of a tree instance is kept on the indexed-sequential file. Successor blocks for a big tree are placed on an overflow file. This overflow file is a disk-resident sequential file, augmented so that blocks can be appended, marked deleted, or replaced. In IMS VS another VSAM file without an index is used for this purpose.

In IMS/360 the insertion of a new tree instance is accommodated by use of an IMS-provided link chain to the overflow file, ignoring the ISAM overflow mechanism. Insertion of new segments is accomplished by placing the segment in its proper physical position and moving the successor segments of this tree instance out of the way. Overflow blocks will be acquired at the end of the overflow file, to accommodate any final segments of a tree that do not fit within the old blocks.

HDAM The direct-access organization has also been extended with an overflow file. The HDAM method is useful if sequentiality is not required and updates are frequent. Collisions are resolved by chaining in key sequence within a bucket.

Only the key of a root segment and a pointer to the remainder of the tree instance is stored in the direct-access file; the remainder of the root segment and all other segments of the tree are kept on the overflow file. The direct file and the overflow file are actually separate areas of the same operating system file. New segments are placed in any free area of the overflow file. The segments of a tree instance are related to each other using pointers and will not be physically moved during updating. Segments belonging to different tree instances may share blocks. Two linkage choices for segments of a tree are available:

1 A chain which links the segments in the *preorder* sequence

2 A hierarchical ring structure using member and sibling pointers

The first choice is sketched in Fig. 9-19.

The insertion algorithm attempts to place the segment according to best locality and minimal space fragmentation. A bit map which indicates blocks that have free space adequate for segments and a free space list within each block are used by this algorithm to achieve efficient placement and allow recovery of space from deleted segments.

HIDAM A combination of indexed-sequential access to those tree segments which are loaded initially and linked access to new segments placed into the file allows sequential processing, indexed access, and frequent updating. Only the key fields of the top segment of a tree instance are kept in the indexed-sequential file; the complete tree instances are kept in the overflow file. A separate indexed-overflow file is used to hold keys for new tree instances. The initial generation of the file is again a batch process, receiving segments in the desired *preorder* sequence. Updating will disturb this sequence. Sequential processing will be slow if updating has scattered the segments of a tree instance over many blocks, or if many new tree instances have been inserted at the end of the file. These are also retrieved via the key file, unless two trees have identical keys.

Other Options When tree instances become excessively large, collections of subtree instances can be kept on a different file using as root segments only copies of the keys of the original root segments. IMS provides this option for HISAM using schema specifications without requiring programmer intervention.

If any indexing, outside of the primary index to the tree instances, is required, lexicons have to be defined by the user. These can be kept using HDAM or HISAM file organization. As a dependent part they will use *logical* segments, which are connected by next and prior pointers to the real segment instances to be indexed.

A plethora of additional options to control storage and linkage is available. Traditional database education does little to prepare a programmer to make decisions which deal with systems of this complexity while attempting to provide reliable, responsive, and cost-effective service to the folks who require the information hidden in the database. The database administrator may need some automatic design tools.

9-7 ADVANCES IN IMPLEMENTATION

The systems we presented in this chapter have a long history. Many of their design concepts were developed in the late 1960s or early 1970s. They are important now because of their maturity. They can handle nontrivial databases, have the backup and reliability mechanisms necessary in a multiuser environment, and are known to a wide range of data-processing professionals. At the same time they tend to lack features we would like to see in more advanced systems. We will discuss such features now without citing specific systems. It is hard to predict commercial success of any new system; much depends on marketing and financing. No new system we investigated includes all or even most of these features.

9-7-1 Distribution of Databases

If data are distributed, schemas at each site have to carry information about remote as well as about local data. A local subschema has only to keep track of remote elements which may be requested by queries originating at the site of the subschema. Such subschemas will contain definitional entries for the data elements at other sites, but only the names and sites of attributes at remote sites have to be kept in the storage schema section. Even then the distributed schemas may become large and difficult to maintain, since schema information is replicated over multiple sites.

Some of the data may be replicated onto more than one site. A retrieval request needs only to be directed to the site which is easiest to reach or has the lowest load. An update has to be directed to all copies of the data. To execute a transaction which involves multiple sites, those portions of the transaction which cannot be executed locally will be transformed into subtransactions to be transmitted over the communication links for execution at the remote sites.

For optimization of transactions on distributed databases the following points need to be considered:

Sources Which sites contain the requested data elements?

Data-segment sizes What is the expected partial result or data-segment size for the subtransaction? This question is repeated after each processing step.

Retrieval capability What are the speed and cost for retrieval at those sites?

Communication capability What are the data transmission capabilities, i.e., the available data transfer rates, between the sites?

Processing capability What is the capability of various sites to carry out any required processing? Especially joins or partial joins (see Sec. 9-3-5) are important.

Result site Is the final output required at the site of the request, at another site, or anywhere in the net? The last case applies to subtransactions creating data segments for further processing.

If one node has all this information, the query can be preplanned at one site, and all the subtransactions can be generated and scheduled there. Linear programming algorithms have been used to solve such problems, at least for the case without replicated data.

It can, however, be impractical to provide all other nodes with procedures for estimation of remote data-segment sizes, since the data volume and key distribution at a site will fluctuate. The global performance of these schemes is unfortunately critically dependent on data-segment sizes, since the transfer rates across communication links tend to be an order of magnitude slower than those of storage devices. In that case only primitive retrieval subtransactions will be spawned, and optimization will proceed step by step, as the data-segment sizes become known. Current systems tend to rely on programmer decision to handle such problems.

9-7-2 Multimodel Capability

We find instances where programmed, navigational access appears to be appropriate to look for specific instances in a database. For a single user a hierarchical view is often clear and adequate. Relational queries provide the most generality and manipulability. Relational formulations may also be appropriate as an intermediate interface for queries stated in a natural language such as English or in an interactive manner on on-line terminals, perhaps with graphics. Translation of navigational queries on a relational implementation is equally feasible, but efficient execution may be difficult. We discuss such issues in Chap. 10.

9-7-3 Choice of Access Structures

A database system should be able to support a wide variety of query types and the corresponding updates. Most relational systems will process a query using indexes if indexes are available for the search arguments. Similarly other access constructs which use locality, pointers, etc., can be used to permit general and efficient retrieval capability on any database structure. We are aware that locality is central to good retrieval performance.

In order to increase locality for critical data, it is often desirable to replicate data elements. *Replication* does increase update and storage costs, but its benefits can outweigh these costs, especially for data which are read much more frequently than updated. The database systems should deal formally with replicated data, so that updates will be correctly and completely executed. Some experiments have been made on relational systems; the process was called *denormalization*, and demonstrated its feasibility. In current systems this technique is used only in an ad hoc fashion during database design, and the correctness of updates has to be assured by user programs.

9-7-4 Binding Alternatives

There are query situations and data collections where efficiency is important, and situations where flexibility is paramount. The alternatives of compiled and interpretive access are appropriate for some and not for other cases. It is hence desirable to be able to provide a mix of translation schemes.

In a system which provides a range of binding choices it may become feasible to add relations to the model, add attributes to existing relations, and define and implement new connections. Then it may become possible to change the database model without an extensive reorganization.

In current systems a reorganization tends to be traumatic. It takes many hours, and a failure of the process will also require many hours of restoration to the earlier state.

Very Large Databases To allow growth of the database, the 1978 DSDL storage schema can specify an initial loading `DENSITY` of fewer records than fit into a block. Equivalent facilities were actually provided in earlier implementations; for instance, the UNIVAC DBMS-1100 provides an `INTERVAL` statement for this function.

When extra space is exhausted, a database reorganization may be required. Reorganization of a database is apt to be a major effort. Data for TOTAL indicates, for instance, that on a large IBM 360 computer, it takes about 1 minute to reload 1000 records. Reorganization of the database at Equitable Life Assurance takes 72 hours of computer time [Gosden in Jardine74]. This leads to the following definition of a *very large database*.

> A very large database is a database whose reorganization by reloading takes a longer time than the users can afford to have the database unavailable.

Problems of very large databases are now being addressed prior to standardization efforts [Steel75]. To avoid frequent reorganizations, the records of the database may be extended with spare fields, so that attributes can be added later. A sophisticated method to cope with database reorganization will be presented in Sec. 11-4-3.

9-7-5 Tools for the Design and Testing of a Database

The complexity of databases is such that it is difficult for a single individual to maintain control. The structural model provides conceptual assistance. Database dictionary systems provide the capability to create catalogs of the data in the database, and many can be used for automatic schema generation. They will typically support only one type of DBMS, however.

Testing has its own risks. The danger of a misprogrammed transaction destroying the corporate database is real. Keeping a test copy of the database is expensive in storage costs and maintenance.

It should be possible to run transactions during checkout on the real database for read accesses and on test blocks when writing or reading data created during the test.

9-7-6 Schemas with High Level Definitions of Connections

Recent developments in query languages do recognize relationships among entities. They need to be supported by data-definition facilities of equivalent power. An important issue in schema management is integrity and constraint specifications. In most systems today such specifications are closely linked to implementation techniques. It is desirable that logical constraints be specified first, and the implementation choice be left to something akin to a storage schema.

Rigorous domain definitions can do much to increase the integrity of databases. Just stating the computer representation (i.e., `integer, floating point, char, ...`) does not indicate the constraints appropriate to connections and joins in a database. Character type variables are appropriate for connections or joins only if defined by a common referenced entity relation or lexicon; otherwise many matches will be missed. Real values are rarely appropriate for connections or equijoins. Joins with integer-type attributes can easily attempt to match domains which are inappropriate, say, `age` and `number-of-children`.

9-7-7 Modularity

The requirements listed above could imply that a future DBMS will be even more massive than the current ones. To avoid this trap we look forward to having database systems composed out of manageable modules. Candidate modules are:

File Systems Current file systems are often awkward for database system use because of lack of symmetry and formal interfaces. Most DBMSs today build their own file support out of primitive facilities. Our understanding of file alternatives is such that this should not be necessary.

Schema Management Translation and manipulation of the tables for a data definition language is a general function which can be handled separately.

Query Interface The translation of queries from a user-friendly language is a major and specialized task. Use of well-defined schema tables and a general data manipulation language will provide interfaces to isolate this function into a module.

Backup and Recovery Protection of the content of a database, presented in Secs. 11-3 and 11-4, is an important task in many but not all database applications. The level of protection required may be specified in the schema, and actions which may require protection can be triggered when the database is manipulated. In a transaction oriented database system the tasks are well-defined (see Sec. 11-3-1) but exceed often the services provided by the operating system. The module can provide a consistent interface while size of this module will depend on the capability of the operating system.

BACKGROUND AND REFERENCES

Descriptions about the the implementation of databases tend to be scattered through user manuals, company reports, academic research reports, and internal documents. Manufacturers' manuals describe only what the system will do and not what it will not do; a number of systems are referenced in Appendix B. Manuals do, of course, provide much more detail than can be found in scholarly publications. The reference material on IMS alone exceeds by far the size of this book. The terminology used is often specific to the system and may even differ from the terminology used to describe other systems of the same manufacturer. Appendix A is intended to be helpful here. Descriptions of new systems are found in the popular computing magazines.

A number of textbooks as Cardenas[79], Kroenke[78], and Tsichritzis[77] stress description and comparison of database-management systems. Specialized database systems are typically described in application-oriented publications; see the references in Chap. 1.

Early descriptions in the computer literature included Bachman[64], Dodd[66], Bleier[68], and Hsiao[71]. Specialized systems include PARS (Siwiec[77]) and TOD (Weyl[75]).

Improvement of binding choices is considered by Stemple[76]. Nunamaker[73] considers the database system as a unit in a larger user support system.

Some early work in set-oriented databases (Childs[68]) led to RDMS (Steuert in Rustin[74]), and related systems as MACAIMS (Strnad[71]), SAM (Symonds[68]), and DAMAS (Rothnie in Rustin[74]). Papers by E.F. Codd (in Codd[71], Rustin[72], and Klimbie[75]) and his colleagues (Boyce[75]) have provided impetus for relational implementations. After initial experimentation using APL (Palermo[75]) came several developmental systems, as IS/1 or PRTV (Todd in Kerr[75]), MORIS (Bracchi in Klimbie[75]), INGRES (Held[75], Stonebraker[76]), and SYSTEM-R (Chamberlin[76], King[80]).Some systems try to integrate the programming language with database manipulation (Schmidt[77], Shopiro[79], and vandeRiet[81])

Some of these systems are now moving into commerce (Chamberlin[81G]). King[80] assesses their status. Brodie[82] surveys the large number of relational systems are now on the market; Appendix B lists many. Not all systems going by the name "relational" provide the same range of functions and their performance differs greatly. Some systems appear even to be limited to handling one relation at a time.

The process of translation of relational calculus languages is detailed by Codd in Rustin[72] and has received much attention. Gotlieb in King[75] analyzed the cost of joins. Merrett[81] tries to reduce their costs. Problems of operational efficiency are being attacked by Rothnie in Rustin[74], Wong[76], Härder[78], Selinger in Bernstein[79], Yao[79], Aho[79S], and Katz in Chen[80]. Sequences of queries are optimized by Finkelstein[82]. Hall[76] and Smith[75] consider relational algebras.

Major joint efforts by users and manufacturers produced the first large commercial database systems, IDS (Bachman[66]) and IMS (Lutz[71], McGee[77]).

Techniques for optimal design, considering both retrieval and update, are due to Yao[77,79] and Whang[81] Design of CODASYL databases was considered by Gambino[77], and Whang[82] provides separability rules.

Methods to aid database design decisions has been published (Mehl in Rustin[74], Smith in King[75]). Lusk in Chen[80], Hubbard[81], and Gerritsen in Yao[82] They all tend to apply to specific system styles.

At the same time a number of relatively simple but clean implementations of databases have become available commercially: SYSTEM 2000, ADABAS, TOTAL. Commercially available systems have been compared in CODASYL[71A] (see Olle[71]), Cohen[75], and Palmer[75]. Many comparisons describe features and suffer from lack of an underlying model. Comparisons of systems are often part of a company's software selection process and, if they can be obtained will provide good study material. Problems faced in hierarchical query design are described by Hardgrave[80]. Lefkovitz[74] describes implementation issues in a general sense.

The work of the CODASYL group has become the basis for the implementation or augmentation of a number of commercial database systems: DBMS-1100 (Emerson in Jardine[74]), IDMS and Boeing IPAD (Swanson[80]).

Issues of CODASYL implementation (Olle[78], Douque[76], and Parsons[74]) and appropriate usage (Stacey[74], Taylor in Rustin[74] and in Douque[76]) have been presented. Buneman[82] implemented a functional query language within a CODASYL structure.

Some worked-out example programs have become available to aid in the evaluation of the DBTG design: Frank[73] (Codd in Rustin[74] uses the same example), and Robinson in Douque[76]. In Sibley[76] a sample database about U.S. presidents is used for three approaches. The work of CODASYL is continuing and changes in syntax and semantics continue to be made. These changes are published regularly as *change pages* to the Journal of Development (CODASYL[73,...]).

The relational approach is compared with the implementation of IMS and the specification of CODASYL by Bachman[75], Martin[77], Date[81], Olle in Benci[75], and Nijssen in Neuhold[76]. The significance of the relational approach is stated by Codd[82]. Analyses of the effects of implementation choices are given by Bachman in Jardine[74], Stonebraker[80], Kay in Douque[76], and Engles in Neuhold[76].

The issues of optimal distribution of data and queries in a network have been analyzed by Eswaran[74], Chu[79], Baldissera[79], Hevner[79], and Ceri[81]. When replication of fragments of the database is permitted the optimal allocation problem becomes very hard, although designers have been able to deal in an ad hoc manner with practical cases. Morgan[77] considers the constraints due to the dependencies between programs and data. Bernstein[81] analyzes a method to minimize join costs between sites. The topics have been surveyed by Epstein[80] and Hevner in Wiederhold[82].

Recent developments in implementation include Tandem's ENFORM system. At CCA investigations were based on the design of SDD-1 (Rothnie[80]). Proposals for distribution have made by Stonebraker[79] and several systems in development are presented in Wiederhold[82].

EXERCISES

1 What are the result relations from the examples in Sec. 9-1-2 given the data in the relations from Chap. 7?

2 How would the query for employees supervised by younger empoyees shown in Example 9-3 be formulated if there were no necessity to specify the relation names?

3 Write the statements required in the relational calculus to obtain the names of departments which have employees with the skill to assemble a given `Assembly` from the `Automobile_section_b` relation using the examples summarized in Fig. 7-29.

4 Work out the division in Example 9-4, using the definition of division given below the example.

5 Formulate for the hierarchy of Fig. 9-8 the query, "Who has children at UC Berkeley and UCLA?"

6 Estimate the retrieval time of the query in Exercise 5 in terms of `fetch` and `get-next` for a hierarchical and a relational database structure.

7 Determine the answer for the query: "Which department has employees with children at UC Berkeley who supervise more than two people?" which is formulated as follows:

```
GET department | schoolname='UC Berkeley' ∧ COUNT(supervision) > 2
```

Then fix the query to represent the stated intent.

8 For which attributes would you expect to find the specification `KEY` or `NON-KEY` in SYSTEM 2000? Who in an organization using such a database should know about the existence of the specification, and who should not?

9 Sketch the layout of the relations
`auto_section, suppliers, parts, supply, possible_supplier` and
`parts_skill_required` in a hierarchical and in a network-type database.

10 Program at a high level the process of locating the suppliers of any `part` in an `auto_section` in a relational algebra database, a hierarchy, and a network.

11 Estimate the retrieval times of each of the above processes. State all assumptions, and use the same assumptions in all three cases.

12 Compare the above process given the SYSTEM 2000 file structure and the OASIS file structure.

13 What attributes are stored redundantly in the TOTAL network database design given in Fig. 9-10?

14 Sketch the TOTAL records for `supplier` and `supplier_assembly` of Fig. 9-10.

15 What is the difference between `KEY` in SYSTEM 2000 and `LOCATION MODE EQUALS CALC` in CODASYL.

16 What CODASYL access choice would you use to locate an employee given his name? Why?

17 To avoid problems with reference pointers, an implementor decides to use symbolic references. What will happen to each of the seven performance parameters used in Chap. 3 for such a system?

18 Could a programmer effectively navigate through a DBTG structure for the relations shown in Fig. 9-19? What about a manager?

19 What is the difference in the implementation of multiple member types in a TOTAL nest relation and multiple member types of a DBTG link set?

20 Compare the storage of IMS tree instances with the storage of MUMPS tree instances in terms of storage density, and fetch and get-next speed.

21 Find an example of a database system other than the ones mentioned in the text which incorporates features belonging more properly to a file system. Explain why the database system was designed in this manner. Suggest an alternate approach.