The hierarchical model of data was developed to model the many types of hierarchical organizations that exist in the real world. Humans have long used hierarchical organization of information to help them better understand the world. There are many examples, such as classification schemes for species in the plant and animal worlds and classification schemes for human languages. Humans also adopted hierarchical structures and naming schemes to deal with the structures they created, such as corporate organization charts, library classification schemes, and governmental hierarchies. The hierarchical data model represents hierarchical organizations in a direct and natural way and may be the best choice in some situations, but it has problems when representing situations with nonhierarchical relationships.

There is no original document that describes the hierarchical model, as there are for the relational and network models. Rather, several early computer information management systems were developed using hierarchical storage structures. Recent examples of these systems include Control Data Corporation's Multi-Access Retrieval System (MARS VI), IBM's Information Management System (IMS), and MRI's System-2000 (now sold by SAS Institute).

In this chapter we present the principles behind the hierarchical model independent of any specific system and relate them to IMS, which is the dominant hierarchical system in use today. Section 11.1 discusses hierarchical schemas and instances. In Section 11.2 the concept of a virtual parent-child relationship, which is used to overcome the limitations of pure hierarchies, is discussed. Section 11.3 discusses constraints on the hierarchical model. Section 11.5 addresses the mapping of schemas from the ER model into the hierarchical model. Sections 11.4 and 11.6 discuss data definition and data manipulation languages for the hierarchical data model as defined here. A representative hierarchical system (namely, IMS) is discussed in Section 11.7. Some features of another
hierarchical system called System 2000 are also mentioned. Section 11.8 summarizes this chapter.
For readers seeking only a brief overview of the hierarchical model, some or all of Sections 11.4 through 11.7 may be skipped.

11.1 Hierarchical Database Structures

In this section, the data structuring concepts of the hierarchical model are discussed. We first discuss parent-child relationships and how they can be used to form a hierarchical schema (in Section 11.1.1). Then we discuss the properties of a hierarchical schema (in Section 11.1.2). Hierarchical occurrence trees are discussed in Section 11.1.3 and a common method for storing these trees—called the hierarchical sequence—is discussed (in Section 11.1.4).

11.1.1 Parent-Child Relationships and Hierarchical Schemas

The hierarchical model employs two main data structuring concepts: records and parent-child relationships. A **record** is a collection of **field values** that provide information on an entity or a relationship instance. Records of the same type are grouped into **record types**. A record type is given a name, and its structure is defined by a collection of named **fields** or **data items**. Each field has a certain data type, such as integer, real, or string.

A parent-child relationship type (PCR type) is a 1:N relationship between two record types. The record type on the 1-side is called the **parent record type**, and the one on the N-side is called the **child record type** of the PCR type. An **occurrence** (or instance) of the PCR type consists of one record of the parent record type and a number of records (zero or more) of the child record type.

A **hierarchical database schema** consists of a number of hierarchical schemas. Each **hierarchical schema** (or hierarchy) consists of a number of record types and PCR types.

A hierarchical schema is displayed as a **hierarchical diagram**, in which record type names are displayed in rectangular boxes and PCR types are displayed as lines connecting the parent record type to the child record type. Figure 11.1 shows a simple hierarchical diagram for a hierarchical schema with three record types and two PCR types. The record

![Figure 11.1](image-url)
types are `DEPARTMENT`, `EMPLOYEE`, and `PROJECT`. Field names can be displayed under each record type name, as shown in Figure 11.1. In some diagrams, for brevity, we display only the record type names.

We refer to a PCR type in a hierarchical schema by listing the pair (parent record type, child record type) between parentheses. The two PCR types in Figure 11.1 are `(DEPARTMENT, EMPLOYEE)` and `(DEPARTMENT, PROJECT)`. Notice that PCR types do not have a name in the hierarchical model. However, a certain meaning is associated with each PCR type by the database designer. In Figure 11.1 each occurrence of the `(DEPARTMENT, EMPLOYEE)` PCR type relates one department record to the records of the many (zero or more) employees who work in that department. An occurrence of the `(DEPARTMENT, PROJECT)` PCR type relates a department record to the records of projects controlled by that department. Figure 11.2 shows two PCR occurrences (or instances) for each of these two PCR types.

### 11.1.2 Properties of a Hierarchical Schema

A hierarchical schema of record types and PCR types must have the following properties:

1. One record type, called the root of the hierarchical schema, does not participate as a child record type in any PCR type.
2. Every record type except the root participates as a child record type in exactly one PCR type.
3. A record type can participate as parent record type in any number (zero or more) of PCR types.
4. A record type that does not participate as parent record type in any PCR type is called a leaf of the hierarchical schema.
5. If a record type participates as parent in more than one PCR type, then its child

![Diagram](attachment:image.png)

**Figure 11.2** Occurrences of PCR types. (a) Two occurrences of the PCR type `(DEPARTMENT, EMPLOYEE)`. (b) Two occurrences of the PCR type `(DEPARTMENT, PROJECT)`. 
record types are ordered. The order is displayed, by convention, from left to right in a hierarchical diagram.

The definition of a hierarchical schema defines a tree data structure. In the terminology of tree data structures, a record type corresponds to a node of the tree, and a PCR type corresponds to an edge (or arc) of the tree. We will use the terms node and record type, and edge and PCR type, interchangeably. The usual convention of displaying a tree is slightly different from that used in hierarchical diagrams, in that each tree edge is shown separately from other edges (Figure 11.3). In hierarchical diagrams the convention is that all edges emanating from the same parent node are joined together (Figure 11.1). We will use this latter hierarchical diagram convention.

The preceding properties of a hierarchical schema mean that every node except the root has exactly one parent node. However, a node can have several child nodes, and in this case they are ordered from left to right. In Figure 11.1 EMPLOYEE is the first child of DEPARTMENT, and PROJECT is the second child. The previously identified properties also limit the types of relationships that can be represented in a hierarchical schema. In particular, M:N relationships between record types cannot be directly represented, because parent-child relationships are 1:N relationships, and a record type cannot participate as child in two or more distinct parent-child relationships.

An M:N relationship may be handled in the hierarchical model by allowing duplication of child record instances. For example, consider an M:N relationship between EMPLOYEE and PROJECT, where a project can have several employees working on it, and an employee can work on several projects. We can represent the relationship as a (PROJECT, EMPLOYEE) PCR type as shown in Figure 11.4(a). In this case a record describing the same employee can be duplicated by appearing once under each project that the employee works for. Alternatively, we can represent the relationship as an (EMPLOYEE, PROJECT) PCR type as shown in Figure 11.4(b), in which case project records may be duplicated.

EXAMPLE 1: Consider the following instances of the EMPLOYEE:PROJECT relationship:

<table>
<thead>
<tr>
<th>Project</th>
<th>Employees Working on the Project</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>E1, E3, E5</td>
</tr>
<tr>
<td>B</td>
<td>E2, E4, E6</td>
</tr>
<tr>
<td>C</td>
<td>E1, E4</td>
</tr>
<tr>
<td>D</td>
<td>E2, E3, E4, E5</td>
</tr>
</tbody>
</table>

Figure 11.3 Tree representation of the hierarchical schema in Figure 11.1.
Figure 11.4 Representing an M:N relationship. (a) One representation of the M:N relationship. (b) Alternative representation of the M:N relationship.

If these instances are stored using the hierarchical schema of Figure 11.4(a), there will be four occurrences of the (PROJECT, EMPLOYEE) PCR type—one for each project. The employee records for E1, E2, E3, and E5 will appear twice each as child records, however, because each of these employees works on two projects. The employee record for E4 will appear three times—once under each of projects B, C, and D. Some data field values in the employee records may be context-dependent; that is, these field values depend on both EMPLOYEE and PROJECT. Such data may differ in each occurrence of a duplicated employee record because it also depends on the parent project record. An example is a field that gives the number of hours per week that an employee works on the project. However, the majority of the field values in the employee records, such as employee name, social security number, and salary, would certainly be duplicated under each project that the employee works for.

To avoid such duplication, a technique is used whereby several hierarchical schemas can be specified in the same hierarchical database schema. Relationships like the preceding PCR type can now be defined across hierarchical schemas, allowing us to circumvent the problem of duplication discussed earlier. This technique, called "virtual" relationships, causes a departure from the "strict" hierarchical model. We discuss this technique in Section 11.2.

11.1.3 Hierarchical Occurrence Trees

Corresponding to a hierarchical schema, many hierarchical occurrences exist in the database. Each hierarchical occurrence, also called an occurrence tree, is a tree structure whose root is a single record from the root record type. The occurrence tree also contains all the children record occurrences of the root record, all children record occurrences within the PCRs of each of the child records of the root record, and so on, all the way to records of the leaf record types.

For example, consider the hierarchical diagram shown in Figure 11.5, which represents part of the COMPANY database introduced in Chapter 3 and also used in Chapters 6 to 10. Figure 11.6 shows one hierarchical occurrence tree of this hierarchical schema. In the occurrence tree, each node is a record occurrence, and each arc represents a parent-child relationship between two records. In both Figures 11.5 and 11.6, we use the characters D, E, P, T, S, and W to represent type indicators for the record types DEPART-
MENT, EMPLOYEE, PROJECT, DEPENDENT, SUPERVISEE, and WORKER, respectively. We shall see the significance of these type indicators when we discuss hierarchical sequences in the next section.

We can define occurrence trees more formally by using the terminology for tree structures, which we need in our subsequent discussion. In a tree structure, the root is said to have level zero. The level of a nonroot node is one more than the level of its parent node, as shown in Figures 11.5 and 11.6. A descendant D of a node N is a node connected to N via one or more arcs such that the level of D is greater than the level of N. A node N and all its descendent nodes form a subtree of node N. An occurrence tree can now be defined as the subtree of a record whose type is of the root record type.
The root of an occurrence tree is a single record occurrence of the root record type. There may be a varying number of occurrences of each nonroot record type, and each such occurrence must have a parent record in the occurrence tree; that is, each such occurrence must participate in a PCR occurrence. Notice that each nonroot node, together with all its descendent nodes, forms a subtree, which, taken alone, satisfies the structure of an occurrence tree for a portion of the hierarchical diagram. Notice, too, that the level of a record in an occurrence tree is the same as the level of its record type in the hierarchical diagram.

11.1.4 Linearized Form of a Hierarchical Occurrence

A hierarchical occurrence tree can be represented in storage by using any of a variety of data structures. However, a particularly simple storage structure that can be used is the hierarchical record, which is a linear ordering of the records in an occurrence tree in the preorder traversal of the tree. This order produces a sequence of record occurrences known as the hierarchical sequence (or hierarchical record sequence) of the occurrence tree; it can be obtained by applying the following recursive procedure to the root of an occurrence tree:

```
procedure Pre_order_traverse ( root record );
   begin
      output ( root record );
      for each child record of root record in left to right order do
         Pre_order_traverse ( child record )
   end;
```

The above procedure, when applied to the occurrence tree in Figure 11.6, gives the hierarchical sequence shown in Figure 11.7. If we use the hierarchical sequence to implement occurrence trees, we need to store a record type indicator with each record because of the different record types and the variable number of child records in each parent-child relationship. The system needs to examine the type of each record as it goes

![Hierarchical sequence](image-url)

Figure 11.7 Hierarchical sequence for the occurrence tree in Figure 11.6.
sequentially through the records. Notice that these record type indicators are implementation structures and are not seen by the hierarchical DBMS user.

The hierarchical sequence is often desirable because the child nodes follow their parent node in storage. Hence, given a parent record, all descendent records in its subtree follow it in the hierarchical sequence and can be retrieved efficiently. However, child records are collectively placed after their parent record only if the child records are leaf nodes in the occurrence tree. Otherwise, whole subtrees of each child node are placed after their parent record in left-to-right order.

The hierarchical sequence is also important because some hierarchical data-manipulation languages, such as that used in IMS, use it as a basis for defining hierarchical database operations. The HDML language we discuss in Section 11.4 is based on the hierarchical sequence.

Next, we define two additional terms that are used by some hierarchical languages. A **hierarchical path** is a sequence of nodes $N_1, N_2, \ldots, N_i$, where $N_1$ is the root of a tree and $N_j$ is a child of $N_{j-1}$ for $j = 2, 3, \ldots, i$. A hierarchical path can be defined either on a hierarchical schema or on an occurrence tree. A hierarchical path is **complete** if $N_i$ is a leaf of the tree. A **broom** is a set of hierarchical paths resulting from the hierarchical path $N_1, N_2, \ldots, N_i$ along with all the hierarchical paths in the subtree of $N_i$. For example, \{(DEPARTMENT, EMPLOYEE, SUPERVISEE)\} is a complete path in the hierarchical schema of Figure 11.5. In the occurrence tree of Figure 11.6, (Administration, Wallace) is a path, and (Administration, Wallace, \{Abner, Zelaya, Jabbar\}) is a broom.

We can now define a **hierarchical database occurrence** as a sequence of all the occurrence trees that are occurrences of a hierarchical schema. This is similar to the definition of a **forest** of trees in data structures. For example, a hierarchical database occurrence of the hierarchical schema shown in Figure 11.5 would consist of a number of occurrence trees similar to the one shown in Figure 11.6. There would be one occurrence tree for each DEPARTMENT record, and they would be ordered as the first, second, \ldots, last occurrence tree.

### 11.2 Virtual Parent-Child Relationships

The hierarchical model has problems when modeling certain types of relationships. These include the following relationships and situations:

1. M:N relationships.
2. The case where a record type participates as child in more than one PCR type.
3. N-ary relationships with more than two participating record types.

As we saw in Section 11.1.2, case 1 can be represented as a PCR type at the expense of duplicating record occurrences of the child record type. Case 2 can be represented in a similar fashion, with more duplication of records. Case 3 presents a problem because the PCR is a binary relationship.

Record duplication, in addition to wasting storage space, causes problems with maintaining consistent duplicate copies of the same record. The concept of a virtual (or pointer) record type is used in the IMS system to deal with all three problem cases iden-
tified earlier. The idea is to include more than one hierarchical schema in the hierarchical database schema and to use pointers from nodes of one hierarchical schema to the other to represent the relationships. We do not follow IMS terminology, but instead develop the concepts more generally.

A virtual (or pointer) record type VC is a record type with the property that each of its records contains a pointer to a record of another record type VP. VC plays the role of “virtual child” and VP of “virtual parent” in a “virtual parent-child relationship.” Each record occurrence c of VC points to exactly one record occurrence p of VP. Rather than duplicating the record p itself in an occurrence tree, we include the virtual record c that contains a pointer to p. Several virtual records may point to p, but only a single copy of p itself is stored in the database.

Figure 11.8 shows the M:N relationship between EMPLOYEE and PROJECT represented with virtual records EPOINTER and PPOINTER. Compare this with Figure 11.4, where the same relationship was represented without virtual records. Figure 11.9 shows the occurrence trees and pointers for the data instances given in Example 1 when the hierarchical schema shown in Figure 11.8(a) is used. In Figure 11.9 there is only a single copy of each EMPLOYEE record; however, several virtual records may point to the same EMPLOYEE record. Hence, the information stored in an EMPLOYEE record is not duplicated. Information that depends on both parent and child records—such as hours per week that an employee works on a project—is included in the virtual pointer record; such data is popularly known among hierarchical database users as intersection data.

Notice that the relationship between EMPLOYEE and EPOINTER in Figure 11.8(a) is a 1:N relationship and hence qualifies as a PCR type. Such a relationship is called a virtual parent-child relationship (VPCR) type. EMPLOYEE is called the virtual parent of EPOINTER; and conversely, EPOINTER is called a virtual child of EMPLOYEE. Conceptually, PCR types and VPCR types are similar. The main difference between the two lies in the way they are implemented. A PCR type is usually implemented by using the hierarchical sequence, whereas a VPCR type is usually implemented by establishing a pointer (a phys-

\textbf{Figure 11.8} Representing an M:N relationship by using VPCR types. (a) One representation of the M:N relationship, with virtual parent EMPLOYEE. (b) Alternative representation of the M:N relationship, with virtual parent PROJECT.
Figure 11.9  The occurrences of Example 1 corresponding to the hierarchical schema in Figure 11.8(a).

Figure 11.10 shows a hierarchical database schema of the COMPANY database that uses some VPCRs and has no redundancy in its record occurrences. The hierarchical database schema is made up of two hierarchical schemas—one with root DEPARTMENT, and the other with root EMPLOYEE. Four VPCRs, all with virtual parent EMPLOYEE, are included to represent the relationships without redundancy. Notice that IMS may not allow this because an implementation constraint in IMS limits a record to being virtual parent of at most one VPCR; to get around this constraint, one can create dummy children record types of EMPLOYEE in Hierarchy 2 so that each VPCR points to a distinct virtual parent record type.
In general, there are many feasible methods of designing a database using the hierarchical model. In many cases, performance considerations are the most important factor in choosing one hierarchical database schema over another. Performance depends on the implementation options available on each specific system, as well as on specific limits set by the DBA at a particular installation—for example, whether certain types of pointers are provided by the system and whether certain limits on number of levels are imposed by the DBA.

One thing to consider about VPCRs is that they can be implemented in different ways. One option is just to have a pointer in the virtual child to the virtual parent, as we discussed earlier. A second option is to have, in addition to the child-to-parent pointer, a backward link from the virtual parent to a linked list of virtual child records. The pointer from the virtual parent to the first virtual child record is called a virtual child pointer, whereas a pointer from one virtual child to the next is called a virtual twin pointer. In this case, the hierarchical model becomes very similar to the network model, which was discussed in Chapter 10. This backward link makes it easy to retrieve all the virtual child records of a particular virtual parent record.
11.3 Integrity Constraints in the Hierarchical Model

A number of built-in inherent constraints exist in the hierarchical model whenever we specify a hierarchical schema. These include the following constraints:

1. No record occurrences except root records can exist without being related to a parent record occurrence. This has the following implications:
   a. A child record cannot be inserted unless it is linked to a parent record.
   b. A child record may be deleted independently of its parent; however, deletion of a parent record automatically results in deletion of all its child and descendant records.
   c. The above rules do not apply to virtual child records and virtual parent records. The rule here is that a pointer in a virtual child record must point to an actual occurrence of a virtual parent record. Deletion of a record should not be allowed while pointers exist to it from virtual child records, making it a virtual parent.

2. If a child record has two or more parent records from the same record type, the child record must be duplicated once under each parent record.

3. A child record having two or more parent records of different record types can do so only by having at most one real parent, with all the others represented as virtual parents.

In addition, each hierarchical DBMS may have its own additional integrity rules that are unique to its own implementation. For example, in IMS a record type can be the virtual parent in only one VPCR type. This implies that the schema of Figure 11.10 is not allowed by IMS, because the EMPLOYEE record is virtual parent in four distinct VPCRs. Another rule in IMS is that a root record type cannot be a virtual child record type in a VPCR type.

Any other constraints that are not implicit in a hierarchical schema must be enforced explicitly by the programmers in the database update programs. For example, if a duplicated record is updated, it is the responsibility of the update program to ensure that all copies are updated in the same way.

11.4 Data Definition in the Hierarchical Model*

In this section we give an example of a hierarchical data definition language (HDDL), which is not the language of any specific hierarchical DBMS but is used to illustrate the language concepts for a hierarchical database. The HDDL demonstrates how a hierarchical database schema can be defined. Some of the terminology used here is different from that of IMS and other hierarchical DBMSs. To define a hierarchical database schema, we must define the fields of each record type, the data type of each field, and any key constraints on fields. In addition, we must specify a root record type as such; and for every nonroot record type, we must specify its (real) parent in a PCR type. Any VPCR types must also be specified.
Figure 11.11 shows the HDDL specification of the database schema shown in Figure 11.10. Most of the statements are self-explanatory. In actual hierarchical DBMSs, the syntax is usually more complicated, and (as we mentioned earlier) the terminology may be different. Notice also that some of the structures, such as the representation of EMPLOYEE as a virtual parent of more than one VPCR type, may not be allowed in some hierarchical DBMSs such as IMS.

In Figure 11.11, either each record type is declared to be of type root or a single (real) parent record type is declared for the record type. The data items of the record are then listed along with their data types. We must specify a virtual parent for data items that are of type pointer. Data items declared under the key clause are constrained to have unique values for each record. Each key clause specifies a separate key; and if a single key clause lists more than one field, the combination of these field values must be unique in each record.

The child number clause specifies the left-to-right order of a child record type under its (real) parent record type. In Figure 11.11 these correspond to the left-to-right order shown in Figure 11.10 and are needed to specify the order of the child subtrees of different child record types under a parent record in the hierarchical sequence. For example, under an EMPLOYEE record we first have all subtrees of its dependent child records (child number = 1), followed by all subtrees of its ESUPERVISEES child records (child number = 2) in the hierarchical sequence.

The order by clause specifies the order of individual records of the same record type in the hierarchical sequence. For a root record type, this specifies the order of the occurrence trees. For example, EMPLOYEE records are ordered alphabetically by LNAME, FNAME, so the occurrence trees of these records are ordered alphabetically by these fields. For nonroot record types, the order by clause specifies how the records should be ordered within each parent record, by specifying a field called a sequence key. For example, PROJECT records controlled by a particular DEPARTMENT have their subtrees ordered alphabetically within the same parent DEPARTMENT record by PNAME, according to Figure 11.11.

11.5 Hierarchical Database Design Using ER-to-Hierarchical Mapping*

In the hierarchical model, only 1:N relationship types can be represented in a particular hierarchy as parent-child relationship (PCR) types. In addition, a record type can have at most one (real) parent record type; hence, M:N relationship types are difficult to represent. Possible ways to represent M:N relationship types in a hierarchical database include the following:

- Represent the M:N relationship type as though it were a 1:N relationship type. In this case, record instances at the N-side are duplicated because each record may be related to several parents. This representation keeps all record types in a single hierarchy at the cost of duplicating record instances. The application programs that update the database must maintain the consistency of duplicate copies.
SCHEMA NAME = COMPANY

HIERARCHIES = HIERARCHY1, HIERARCHY2

RECORD
  NAME = EMPLOYEE
  TYPE = ROOT OF HIERARCHY2
  DATA ITEMS =
    FNAME   CHARACTER 15
    MIN'T   CHARACTER 1
    LNAME   CHARACTER 15
    SSN     CHARACTER 9
    BDATE   CHARACTER 9
    ADDRESS CHARACTER 30
    SEX     CHARACTER 1
    SALARY  CHARACTER 10
  KEY = SSN
  ORDER BY LNAME, FNAME

RECORD
  NAME = DEPARTMENT
  TYPE = ROOT OF HIERARCHY1
  DATA ITEMS =
    DNAME   CHARACTER 15
    DNUMBER INTEGER
  KEY = DNAME
  KEY = DNUMBER
  ORDER BY DNAME

RECORD
  NAME = DLOCATIONS
  PARENT = DEPARTMENT
  CHILD NUMBER = 1
  DATA ITEMS =
    LOCATION CHARACTER 15

RECORD
  NAME = DMANAGER
  PARENT = DEPARTMENT
  CHILD NUMBER = 3
  DATA ITEMS =
    MGRSTARTDATE CHARACTER 9
    MPTR        POINTER WITH VIRTUAL PARENT = EMPLOYEE

RECORD
  NAME = PROJECT
  PARENT = DEPARTMENT
  CHILD NUMBER = 4
  DATA ITEMS =
    PNAME   CHARACTER 15
    PNUMBER INTEGER
    PLOCATION CHARACTER 15
  KEY = PNAME
  KEY = PNUMBER
  ORDER BY PNAME

Figure 11.11  Declarations for the hierarchical schema in Figure 11.10. (continued on next page)
CREATE more than one hierarchy and have virtual parent-child relationship (VPCR) types (logical pointers) from a record type that appears in one hierarchy to the root record type of another hierarchy. These pointers can be used to represent the M:N relationship type in a manner similar to the one used in the network model. Even then, a constraint, adopted from the IMS DBMS model, restricts each record type to having at most one virtual child.

Because multiple options may be considered, there is no standard method for mapping an ER schema into a hierarchical schema. We will illustrate the two possibilities discussed above with two hierarchical schemas, in Figures 11.12(a) and (b), that can be used to represent the ER schema shown in Figure 3.2. A third alternative is shown in Figure 11.10.

Figure 11.12(a) shows a single hierarchy that can be used to represent the ER schema of Figure 3.2. We choose DEPARTMENT as the root record type of the hierarchy. The 1:N relationship types WORKS_FOR and CONTROLS and the 1:1 relationship type MANAGES are represented at the first level of the hierarchy by the record types EMPLOYEE, PROJECT, and DEPT_MANAGER, respectively. However, to limit redundancy, we keep only some of the attributes of an employee who is a manager in the DEPT_MANAGER record type. The EMPLOYEE records in a hierarchical tree owned by a particular DEPARTMENT record will
Figure 11.12  Mapping the ER schema of Figure 3.2 to the hierarchical model. (a) A hierarchical database schema for the COMPANY database with a single hierarchy. (b) Another hierarchical database schema for the same database with two hierarchies and four VPCRs.
represent the employees who work for that department. Similarly, the PROJECT records will represent the projects controlled by that department, and the DEPTMANAGER record will represent the employee who manages the department. An employee who is a manager is thus represented twice—once as an instance of EMPLOYEE, and the second time as an instance of DEPTMANAGER. The application programs are responsible for maintaining these copies in consistent form.

The 1:N relationship type DEPENDENTS_OF is represented by the record type DEPENDENT as a subordinate of EMPLOYEE (we did not include the RELATIONSHIP attribute of DEPENDENT here). The M:N relationship type WORKS_ON is represented as a subordinate of PROJECT, but only an employee’s ENAME and ESSN are included in WORKS_ON, along with the relationship attribute HOURS. An employee who works on several different projects will be stored in as many copies of WORKS_ON record instances with identical ENAME, ESSN values. The rest of the information on employees is not replicated in WORKS_ON, in order to limit redundancy. Notice that each WORKS_ON record represents one of the employees working on a particular project. Alternatively, we could represent WORKS_ON as a subordinate record type of EMPLOYEE; in this case, each WORKS_ON record would represent one of the projects that an EMPLOYEE works on, and its fields would be PNAME, PNUMBER, and HOURS, as shown by the dotted WORKS_ON box in Figure 11.12(a). In the latter case, PROJECT information would be duplicated in multiple copies of WORKS_ON records.

Finally, the SUPERVISION relationship type is represented as a subordinate of EMPLOYEE. We could choose to represent it in either the role of a supervisor or that of a supervisee. In Figure 11.12(a) each SUPERVISOR record represents the supervisor of the owner EMPLOYEE record in the hierarchy, so the hierarchical relationship represents the supervisor role; each employee has a single child record SUPERVISOR representing his or her direct supervisor. Alternatively, we could represent the role of a supervisee in the hierarchical relationship; then every EMPLOYEE record that represents a supervisory employee would be related to (potentially) many direct supervisees as child records. This is shown in dotted lines in the box SUPERVISEE in Figure 11.12(a). In either case, EMPLOYEE information is replicated in the SUPERVISOR or SUPERVISEE records, so we include only a few of the attributes of EMPLOYEE.

There is excessive replication in Figure 11.12(a) of employee information in the record types EMPLOYEE, DEPTMANAGER, WORKS_ON, and SUPERVISOR, because they all represent employees in various roles. We can somewhat limit the replication by representing the ER schema of Figure 3.2 with two or more hierarchies. In Figure 11.12(b) two hierarchies are used. The first hierarchy has DEPARTMENT as root record type and represents the relationship types CONTROLS, MANAGES, and WORKS_ON. The second hierarchy has EMPLOYEE as root record type and represents the relationship types DEPENDENTS_OF and SUPERVISION. By using virtual parent-child relationships and virtual pointers in the WORKS_ON, DEPTMANAGER, and SUPERVISOR record types, we do not replicate any employee information. Each pointer will point to an EMPLOYEE record, but EMPLOYEE information is stored only once as a root record of the second hierarchy. The WORKS_FOR relationship type is represented by a child of the EMPLOYEE record called WORKS_FOR with virtual parent DEPARTMENT.*

*This is done because the root record EMPLOYEE is not allowed to have a virtual parent in IMS.
Figure 11.13  Mapping an n-ary (n = 3) relationship type SUPPLY from ER to hierarchical. (a) One option for representing a ternary relationship. (b) Representing a ternary relationship using three VPCR types.

Finally, consider the mapping of n-ary relationship types, n > 2. Figure 11.13 shows two options for mapping the SUPPLY ternary relationship type of Figure 3.16(a). Because of the constraint, derived from IMS, that a record type can have at most one virtual parent, we cannot place SUPPLY under PART, say, and include two pointers to two virtual parents PROJECT and SUPPLIER. The option in Figure 11.13(a) creates two pointer record types under SUPPLY with virtual parents PROJECT and SUPPLIER. Another option, shown in Figure 11.13(b), is to have SUPPLY as root of a hierarchy and create three pointer record types under it to point to the participating record types as virtual parents. This option is the most flexible one.
Clearly, the hierarchical model offers many options for representing the same ER schema. Many other representations, besides those discussed above, could have been used. The issues of efficient data access and of limiting redundancy versus facilitating retrieval are important in choosing a particular representation. The hierarchical model is generally considered inferior in its modeling capability to both the relational model and the network model, for the following reasons:

- M:N relationship types can be represented only by adding redundant records or by using virtual parent-child relationships and pointer records.
- All 1:N relationship types in a hierarchy must be maintained in the same direction.
- A record type in a hierarchy can have at most one (real) owner record type.
- A record type can have at most two parents—one real and one virtual. (This limitation is specific to IMS.)

### 11.6 Data Manipulation Language for the Hierarchical Model

We now discuss HDML (Hierarchical Data Manipulation Language), which is a record-at-a-time language for manipulating hierarchical databases. The commands of the language must be embedded in a general-purpose programming language, called the host language. Although it is more common to have COBOL or PL/I as the host language, we use PASCAL in our examples to maintain consistency with the rest of this book. Notice that HDML is not a language for a particular hierarchical DBMS; rather, it is introduced to illustrate the concepts of a hierarchical database manipulation language. We begin by introducing the general concept of a user work area for communication with the system, together with some currency concepts.

#### 11.6.1 User Work Area (UWA) and Currency Concepts for Using HDML Commands

In a record-at-a-time language, a database retrieval operation retrieves database records into program variables. In our examples, database records are retrieved into PASCAL program variables. The program can then refer to the program variables to access the field values of the database records. We assume that a PASCAL record type has been declared for each record type in the schema of Figure 11.10. The PASCAL program variables, shown in Figure 11.14, use the same field names as those in the database schema of Figure 11.10, whereas record names are prefixed with a P_. These program variables exist in what is often called the user work area. Notice that it is possible to have these variables declared automatically by referring to the database schema declared in Figure 11.11. Initially, the values of these record variables are undefined. Whenever a data retrieval operation retrieves a database record of a particular type, it is placed in the corresponding UWA program variable.
var P_EMPLOYEE : record
  FNAME: packed array [1..15] of char;
  MINIT: char;
  LNAME: packed array [1..15] of char;
  SSN: packed array [1..9] of char;
  BDATE: packed array [1..9] of char;
  ADDRESS: packed array [1..30] of char;
  SEX: char;
  SALARY : packed array [1..10] of char
end;

P_DEPARTMENT : record
  DNAME: packed array [1..15] of char;
  DNUMBER: integer
end;

P_DLOCATIONS : record
  LOCATION: packed array [1..15] of char
end;

P_DMANAGER : record
  MGRSTARTDATE: packed array [1..9] of char;
  MPTR: database pointer to EMPLOYEE
end;

P_PROJECT : record
  PNAME: packed array [1..15] of char;
  PNUMBER: integer;
  PLOCATION: packed array [1..15] of char
end;

P_PWORKER : record
  HOURS: packed array [1..4] of char;
  WPTR: database pointer to EMPLOYEE
end;

P_DEMPLOYEES : record
  EPTR: database pointer to EMPLOYEE
end;

P_ESUPERVISEE : record
  SPTR: database pointer to EMPLOYEE
end;

P_DEPENDENT : record
  DEPNAME: packed array [1..15] of char;
  SEX: char;
  BIRTHDATE: packed array [1..9] of char;
  RELATIONSHIP: packed array [1..10] of char
end;

Figure 11.14  PASCAL program variables in the UWA corresponding to part of the hierarchical schema in Figure 11.10.

The HDM is based on the concept of hierarchical sequence defined in Section 11.1. Following each database command, the last record accessed by the command is called the current database record. The DBMS maintains a pointer to the current record. Subsequent database commands proceed from the current record and may define a new current record, depending on the type of command. Hence, HDM commands traverse through a hierarchical database retrieving the records required by the query. Originally, the current database record is an "imaginary record" located just before the root record of the first occurrence tree in the database.

If a database has more than one hierarchical schema in it, and these hierarchies are processed together, the IMS system allows the definition of a user view to create a tailora-
made hierarchical schema that includes the desired record types connected by VPCR types.* Such a view is treated as a single hierarchical schema and has its own current database record. Since we do not intend to go into the details of defining and processing these views, we will assume for convenience that each hierarchical schema has its own current of hierarchy record. IMS also has provisions for remembering the last record accessed of each record type, so we assume that the system keeps track of the current of record type for each record type—this is a pointer to the last record accessed from the record type. The HDML commands implicitly refer to these three types of currency indicators:

- Current of database.
- Current of hierarchy for each hierarchical schema.
- Current of record type for each record type.

Record-at-a-time programming requires continuous interaction between the user program and the DBMS. Status information at the end of each database command must be communicated back to the program. This is accomplished by a variable called DB_STATUS, whose value is set by the DBMS software after each database command is executed. We will assume that a value of DB_STATUS = 0 specifies that the last database command was successfully executed.

HDML commands can be categorized as retrieval commands, update commands, and currency retention commands. Retrieval commands retrieve one or more database records into the corresponding program variables and may change some currency indicators. Update commands are used to insert, delete, and modify database records. Currency retention commands are used to mark the current record so that it can be updated or deleted by a subsequent command. The HDML commands are summarized in Table 11.1.

We now discuss each of these commands and illustrate our discussion with examples based on the schema shown in Figure 11.10. In the program segments, HDML commands are prefixed with a $-sign to distinguish them from the PASCAL language statements. PASCAL language key words—such as if, then, while, and for—are written in lowercase.

### 11.6.2 The GET Command

The HDML command for retrieving a record is the GET command. There are many variations of GET; the structure of two of these variations is as follows, with optional parts shown between brackets [ .. ]:

- GET FIRST** `<record type name>` [ WHERE `<condition>` ]
- GET NEXT `<record type name>` [ WHERE `<condition>` ]

The simplest variation is the GET FIRST command, which always starts searching the database from the beginning of the hierarchical sequence until it finds the first record occurrence of `<record type name>` that satisfies `<condition>`. This record also becomes

---

* These “view” schemas are called logical databases in IMS and will be discussed briefly in Section 11.7.3.

** This is similar to the GET UNIQUE (GU) command of IMS.
Table 11.1  Summary of HDML Commands

RETRIEVAL
GET  RETRIEVE A RECORD INTO THE CORRESPONDING
     PROGRAM VARIABLE AND MAKE IT THE CURRENT
     RECORD. VARIATIONS INCLUDE GET FIRST, GET NEXT,
     GET NEXT WITHIN PARENT, AND GET PATH.

RECORD UPDATE
INSERT STORE A NEW RECORD IN THE DATABASE AND MAKE
IT THE CURRENT RECORD
DELETE DELETE THE CURRENT RECORD (AND ITS SUBTREE)
FROM THE DATABASE
REPLACE MODIFY SOME FIELDS OF THE CURRENT RECORD

CURRENCY
RETENTION
GET HOLD RETRIEVE A RECORD AND HOLD IT AS THE CURRENT
RECORD SO IT CAN SUBSEQUENTLY BE DELETED OR
REPLACED

the current of database, current of hierarchy, and current of record type and is retrieved
into the corresponding UWA program variable. For example, to retrieve the “first”
EMPLOYEE record in the hierarchical sequence whose name is John Smith, we write EX1:

EX1:  $GET FIRST EMPLOYEE WHERE FNAME='John' AND LNAME='Smith';

The DBMS uses the condition following WHERE to search for the first record in order
of the hierarchical sequence that satisfies the condition and is of the specified record
type. The value of DB_STATUS is set to 0 (zero) if the record is found successfully; other-
wise, DB_STATUS is set to some other value—1, say—that indicates not found. Other
errors or exceptions are indicated by different values for DB_STATUS.

If more than one record in the database satisfies the WHERE condition and we want
to retrieve all of them, we must write a looping construct in the host program and use
the GET NEXT command. We assume that the GET NEXT starts its search from the current
record of the record type specified in GET NEXT* and searches forward in the hierarchi-
ical sequence to find another record of the specified type satisfying the WHERE condition. For
example, to retrieve records of all EMPLOYEES whose salary is less than $20,000 and obtain
a printout of their names, we can write the program segment shown in EX2:

EX2:  $GET FIRST EMPLOYEE WHERE SALARY < '20000.00';
while DB_STATUS = 0 do
      begin
      writeln( P_EMPLOYEE.FNAME, P_EMPLOYEE.LNAME );
      $GET NEXT EMPLOYEE WHERE SALARY < '20000.00'
      end;

* IMS commands generally proceed forward from the current of database, rather than from the current of spec-
ified record type as HDML commands do.
In EX2, the while loop continues until no more EMPLOYEE records in the database satisfy the WHERE condition; hence, the search goes through to the last record in the database (hierarchical sequence). When no more records are found, DB_STATUS becomes nonzero, with a code indicating "end of database reached," and the while loop terminates. Notice that the WHERE condition in the GET commands is optional. If there is no condition, the very next record in the hierarchical sequence of the specified record type is retrieved. For example, to retrieve all EMPLOYEE records in the database, we can use EX3:

EX3:  **$GET FIRST** EMPLOYEE;
      while DB_STATUS = 0 do
         begin
            writeln ( P_EMPLOYEE.FNAME, P_EMPLOYEE.LNAME);
            **$GET NEXT** EMPLOYEE
         end;

11.6.3 The GET PATH and GET NEXT WITHIN PARENT Commands

So far we have considered retrieving single records by using the GET command. But when we have to locate a record deep in the hierarchy, the retrieval may be based on a series of conditions on records along the entire hierarchical path. To accommodate this, we introduce the GET PATH command:

GET ( FIRST | NEXT ) PATH <hierarchical path> [ WHERE <condition> ]

Here, <hierarchical path> is a list of record types that starts from the root along a path in the hierarchical schema, and <condition> is a Boolean expression specifying conditions on the individual record types along the path. Because several record types may be specified, the field names are prefixed by the record type names in <condition>. For example, consider the following query: "List the lastname and birthdates of all employee-dependent pairs, where both have the first name John." This is shown in EX4:

EX4:  **$GET FIRST** PATH EMPLOYEE, DEPENDENT
      WHERE EMPLOYEE.FNAME='John' AND DEPENDENT.DEPNAME='John';
      while DB_STATUS = 0 do
         begin
            writeln ( P_EMPLOYEE.LNAME, P_EMPLOYEE.BDATE,
                      P_DEPENDENT.BIRTHDATE);
            **$GET NEXT PATH** EMPLOYEE, DEPENDENT
            WHERE EMPLOYEE.FNAME='John' AND
            DEPENDENT.DEPNAME='John'
         end;

We assume that a GET PATH command retrieves all records along the specified path into the UWA variables,* and the last record along the path becomes the current database record. In addition, all records along the path become the current records of their respective record types.

*IMS provides the capability of specifying that only some of the records along the path are to be retrieved.
Another common type of query is to find all records of a given type that have the same parent record. In this case we need the GET NEXT WITHIN PARENT command, which can be used to loop through the child records of a parent record and has the following format:

\[
\text{GET NEXT } \langle\text{child record type name}\rangle \\
\text{WITHIN [ VIRTUAL ] PARENT [ } \langle\text{parent record type name}\rangle \text{ ]} \\
[ \text{WHERE } \langle\text{condition}\rangle ]
\]

This command retrieves the next record of the child record type by searching forward from the current of the child record type for the next child record owned by the current parent record. If no more child records are found, DB_STATUS is set to a nonzero value to indicate that "there are no more records of the specified child record type that have the same parent as the current parent record." The \langle\text{parent record type name}\rangle is optional, and the default is the immediate (real) parent record type of \langle\text{child record type name}\rangle. For example, to retrieve the names of all projects controlled by the 'Research' department, we can write the program segment shown in EX5:

\[
\text{EX5: } \quad \$\text{GET FIRST PATH DEPARTMENT, PROJECT} \\
\text{WHERE DNAME='Research';} \\
(\text{the above establishes the 'Research' DEPARTMENT record as} \\
\text{current parent of type DEPARTMENT, and retrieves the first child} \\
\text{PROJECT record under that DEPARTMENT record })
\]

\[
\text{while DB\_STATUS } = 0 \text{ do} \\
\text{begin} \\
\text{writeln ( P\_PROJECT.PNAME );} \\
\text{\$GET NEXT PROJECT WITHIN PARENT} \\
\text{end;}
\]

In EX5, we can write "WITHIN PARENT DEPARTMENT" rather than just "WITHIN PARENT" in the GET NEXT command with the same effect, because DEPARTMENT is the immediate parent record type of PROJECT. However, if we want to retrieve all records owned by a parent that is not the immediate parent—for example, all the PWORKER records owned by the same DEPARTMENT record—then we must specify DEPARTMENT as the parent record type in the "WITHIN PARENT" clause.

Notice that there are two main methods for explicitly establishing a parent record as the current record:

- If we use GET FIRST or GET NEXT, the record retrieved becomes the current parent record.
- If we use a GET PATH command, a hierarchical path of current parent records of the respective record types is established. This can also retrieve the first child record, as demonstrated in EX5, so that subsequent GET NEXT WITHIN PARENT commands can be issued.

*There is no provision for retrieving all children of a virtual parent in IMS in this way without defining a view of the database.*
We can rewrite EX4 without the GET PATH command by using one loop to find EMPLOYEES with FNAME = ‘John’ and a nested loop using GET NEXT WITHIN PARENT to find any DEPENDENTS of each such EMPLOYEE with DEPNAME = ‘John’. However, the GET PATH command allows us to do this more directly and with a smaller number of calls to the DBMS.

Another variation of the GET command can be used to locate the real or virtual parent record of the current record of a specified child record type.*

GET [ VIRTUAL ] PARENT <parent record type name> OF <child record type name>

For example, to retrieve the names and hours per week for each employee who works on ‘ProjectX’, we can use the GET PARENT command, as in EX6.

EX6:  $GET FIRST PATH DEPARTMENT PROJECT, PWORKER
      WHERE PNAME=’ProjectX’; (* establish parent record and retrieve first child *)
      while DB_STATUS = 0 do
        begin
          $GET VIRTUAL PARENT EMPLOYEE OF PWORKER;
          if DB_STATUS=0 then
            writeln (P_EMPLYEE.LNAME, P_EMPLYEE.FNAME, P_PWORKER.HOURS)
          else writeln (’error--has no EMPLOYEE virtual parent’);
          $GET NEXT PWORKER WITHIN PARENT PROJECT
        end;

Notice that we can use a WHERE condition with the GET NEXT WITHIN PARENT command. For example, to retrieve the names of employees who work more than 5 hours per week on ‘ProjectX’, we can modify the GET NEXT PWORKER WITHIN PARENT command in EX6 to:

$GET NEXT PWORKER WITHIN PARENT PROJECT WHERE HOURS > ’5.0’;

We must also modify the GET FIRST PATH command appropriately. Just as we permitted traversing a VFCR from child to parent, as illustrated in EX6, we can also traverse from parent to child, using the following command modification:

GET NEXT <virtual child record type name> WITHIN PARENT <virtual parent record type name>

11.6.4 Calculating Aggregate Functions

Aggregate functions such as COUNT and AVERAGE must be explicitly implemented by the programmer, using the facilities of the host programming language. For example, to

* IMS does not have a counterpart for this command without using what IMS calls a logical database, which “hides” this operation by thinking of a virtual parent and a virtual child as a single record.
calculate the number of employees who work in each department and their average salary, we can write EX7:

```
EX7:  $GET FIRST PATH DEPARTMENT, DEPLOYEES;
     while DB_STATUS = 0 do
         begin
             total_sal:= 0; no_of_emps:= 0; writeln (P_DEPARTMENT.DNAME);
             (* department name *)
             while DB_STATUS = 0 do
                 begin
                     $GET VIRTUAL PARENT EMPLOYEE;
                     total_sal:= total_sal + conv_sal (P_EMPLOYEE.SALARY);
                     no_of_emps:= no_of_emps + 1;
                     $GET NEXT DEPLOYEES WITHIN PARENT DEPARTMENT
                     end;
                     writeln( 'no of emps =', no_of_emps,'avg sal of emps =',
                     total_sal/no_of_emps);
                     $GET NEXT PATH DEPARTMENT, DEPLOYEES
                 end;
```

To accumulate the total salary and number of employees, we must have program variables declared, so we assume that the program variables total_sal:real and no_of_emps:integer are declared in the program header and that a PASCAL function conv_sal:real is also declared that converts a salary value from string to real number.

### 11.6.5 HDML Commands for Update

The HDML commands for updating a hierarchical database are shown in Table 11.1. The INSERT command is used to insert a new record. Before inserting a record of a particular record type, we must first place the field values of the new record in the appropriate user work area program variable. For example, suppose that we want to insert a new EMPLOYEE record for John F. Smith; we can use the program segment in EX8:

```
EX8:  P_EMPLOYEE.FNAME := 'John';
      P_EMPLOYEE.LNAME := 'Smith';
      P_EMPLOYEE.MINIT := 'F';
      P_EMPLOYEE.SSN := '567342739';
      P_EMPLOYEE.ADDRESS := '40 N.W. 80th Blvd., Gainesville, Florida, 32607'
      P_EMPLOYEE.BDATE := '10-JAN-55';
      P_EMPLOYEE.SEX := 'M';
      P_EMPLOYEE.SALARY := '30000.00'
      $INSERT EMPLOYEE FROM P_EMPLOYEE;
```

The INSERT command inserts a record into the database. The newly inserted record also becomes the current record for the database, its hierarchical schema, and its record type. If it is a root record, as in EX8, it creates a new hierarchical occurrence tree with the new record as root. The record is inserted in the hierarchical sequence in the order specified by any ORDER BY fields in the schema definition. For example, the new
EMPLOYEE record in EX8 is inserted in alphabetic order of its LNAME, FNAME combined value, according to the schema definition in Figure 11.11. If no ordering fields are specified in the definition of the root record of a hierarchical schema, a new root record is inserted following the occurrence tree that contained the current database record before the insertion.

To insert a child record, we should make its parent, or one of its sibling records, the current record of the hierarchical schema before issuing the INSERT command. We should also set any virtual parent pointers before inserting the record. To do that, we need a command SET VIRTUAL PARENT, which sets the pointer field in the program variable to the current record of the virtual parent record type.* The record is inserted after finding an appropriate place for it in the hierarchical sequence past the current record. For example, suppose that we want to relate the EMPLOYEE record inserted in EX8 as a 40-hour-per-week worker on the project whose project number is 55; we can use EX9:

EX9:  \$GET FIRST EMPLOYEE WHERE SSN='567342739'; (* find virtual parent *)
     if DB_STATUS=0 then
       begin
         P_PWORDER.WPTR := SET VIRTUAL PARENT; (* virtual parent pointer to current record *)
         P_PWORDER.HOURS := '40.0';
         \$GET FIRST PROJECT WHERE PNUMBER=55; (* make (real) parent the current record *)
         if DB_STATUS=0 then \$INSERT PWORDER FROM P_PWORDER;
       end;

To delete a record from the database, we first make it the current record and then issue the DELETE command. The GET HOLD is used to make the record the current record, where the HOLD keyword indicates to the DBMS that the program will delete or update the record just retrieved. For example, to delete all male EMPLOYEES, we can use EX10, which also lists the deleted employee names before deleting their records:

EX10:  \$GET HOLD FIRST EMPLOYEE WHERE SEX='M';
     while DB_STATUS=0 do
       begin
         writeln (P_EMPLOYEE.LNAME, P_EMPLOYEE.FNAME);
         \$DELETE EMPLOYEE;
         \$GET HOLD NEXT EMPLOYEE WHERE SEX='M';
       end;

Notice that deleting a record means automatically deleting all its descendent records—all records in its subtree. However, virtual child records in other hierarchies are not deleted. In fact, before deleting a record, the DBMS should make sure that no virtual child records point to it. Following a successful DELETE command, the current record becomes an “empty position” in the hierarchical sequence corresponding to the record just deleted. Subsequent operations continue from that position.

*The SET VIRTUAL PARENT action is done implicitly in IMS when a logical record that includes the virtual parent in its definition is inserted.
To modify field values of a record, we take the following steps:

1. Make the record to be modified the current record, and retrieve it into the corresponding UWA program variable by using the GET HOLD command.
2. Modify the desired fields in the UWA program variable.
3. Issue the REPLACE command.

For example, to give all employees in the ‘Research’ department a 10% raise, we can use the program shown in EX11:

```sql
EX11: $GET FIRST PATH DEPARTMENT, DEMPLOYEES
    WHERE DNAME='Research';
while DB_STATUS = 0 do
    begin
        $GET HOLD VIRTUAL PARENT EMPLOYEE OF DEMPLOYEES;
        P_EMPLOYEE.SALARY := P_EMPLOYEE.SALARY * 1.1;
        $REPLACE EMPLOYEE FROM P_EMPLOYEE;
        $GET NEXT DEMPLOYEES WITHIN PARENT DEPARTMENT
    end;
```

### 11.7 Overview of the IMS Hierarchical Database System

#### 11.7.1 Introduction

In this section we survey a major hierarchical system—Information Management System, or IMS. Although IMS essentially implements the hypothetical hierarchical data model described earlier in this chapter, many features are peculiar to this complex system. We highlight the architecture and special types of view processing and storage structures in IMS, and we compare DL/1, IMS’s data language, with the HDI and HDML discussed earlier.

IMS is one of the earliest DBMSs, and it ranks as the dominant system in the commercial market for support of large-scale accounting and inventory systems. IBM manuals refer to the full product as IMS/VS (Virtual Storage); and typically, the full product is installed under the MVS operating system. IMS DB/DC is the term used for installations that utilize the product’s own subsystems to support the physical database (DB) and to provide data communications (DC).

However, other important versions exist that support only the IMS data language (DL/1). Such DL/1-only configurations can be implemented under MVS, but they may also use the DOS/VSE operating system. These systems issue their calls to VSAM files and use IBM’s Customer Information Control System (CICS) for data communications. The trade-off is a sacrifice of support features for the sake of simplicity and improved throughput.

The original IMS/360 Version 1 product was introduced by IBM in 1968 following a joint development project with North American Rockwell. A number of major revisions to IMS have followed. These have incorporated or accommodated major technological advances: modern communications networking, direct record access (augmented “fast
path"), and secondary indexes, among others. IMS development now embodies several thousand man-years of effort, much of that driven by the needs of an articulate user community.

IMS has no built-in query language, which can be seen as a major shortcoming. Partial responses to this situation appeared early on, with IBM's IQF (interactive query facility) and other add-on products sold by vendors or developed by users. One common and high-flexibility solution today is to download information from the typically enormous IMS database to a separate relational system. Then, with relevant summary data moved to a microcomputer or to the mainframe's SQL/DS or DB2 system, individual corporate entities can carry out their own information system functions.

A number of versions of IMS have been marketed to work with various IBM operating systems, including (among the recent systems) OS/VS1, OS/VS2, MVS, MVS/SD and ESA. The system comes with various options. IMS runs under different versions on the IBM 370 and 30XX family of computers. The data definition and manipulation language of IMS is Data Language One, or DL/I. Application programs written in COBOL, PL/1, FORTRAN, and BAL (Basic Assembly Language) interface with DL/I.

System 2000 (S2K) is another popular hierarchical system that follows a different version of the hierarchical data model. It operates on a wide range of systems, including IBM 360/370, 43XX, and 30XX models, as well as UNIVAC, CDC, and CYBER hardware. System S2K can be configured with options such as a nonprocedural query language for nonprogrammers, a procedural language interface to COBOL, PL/1, and FORTRAN, a sequential file processing capability, and a teleprocessing monitor. In the rest of this section, we describe various aspects of IMS.

### 11.7.2 Basic Architecture of IMS

The internal organization of IMS can be described in terms of various layers of definitions and mappings. IMS uses its own terminology, which is sometimes confusing or misleading. A stored hierarchy in IMS is called a physical database (PDB). For a given installation, the data in the database comprises several physical databases. Each physical database has a data definition or a schema written in DL/I. IMS calls this definition a DBD, for database description. The compiled form of a DBD is stored internally; it includes information on how the database definition is mapped into storage and what access methods are applicable.

IMS provides a view facility that is fairly complex. A view can be defined by choosing part of a physical database or by choosing parts of a number of physical databases and interlinking them into a new hierarchy. We shall refer to these as type 1 and type 2 views, respectively. (The type nomenclature is our own.)

A type 1 view is a subhierarchy and is defined by means of a Program Communication Block or PCB. A type 2 view must be defined in DL/I in terms of a logical DBD. The resulting structure is called a logical database (LDB). Physical and logical databases are discussed in Section 11.7.3.

A user application program needs to access data from several isolated physical databases or from type 1 or type 2 views. In high-volume on-line transaction systems, reentrant data access modules are used. All the data descriptions needed by an application are packaged in a Program Specification Block or PSB. A PSB contains different chunks
of description, corresponding to type 1 or type 2 view definitions. These chunks are stored as Program Communication Blocks. Each application must have a distinct PSB, even though it may be identical to another PSB. The application program in COBOL, PL/1, FORTRAN, or BAL invokes DL/I via a call to get IMS to service a retrieval or update operation. The IMS system in turn communicates with the user via the PCB, which is (defined in the application program as) an area addressable via a pointer passed to the program. Current status information is posted to the PCB. IMS mainly supports five access methods, HSAM, HISAM, HDAM, HIDAM, and MSDB, which in turn use the built-in access methods of the operating system to manage various files.

Figure 11.15 shows how two applications called SALES and GENERAL LEDGER may actually share underlying physical databases in IMS via DBDs, PCBs, and PSBs.

11.7.3 **Logical Organization of Data in IMS**

In IMS, records are called segments, and relationships are distinguished into physical and logical (instead of real and virtual). Our terminology is cross-referenced with that of IMS and System 2000 in Table 11.2.
### Table 11.2  Hierarchical Data Model Terminology

<table>
<thead>
<tr>
<th>Hierarchical Data Model</th>
<th>IMS Term</th>
<th>System 2000 Term</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Record type</td>
<td>Segment type</td>
<td>Repeating group record</td>
</tr>
<tr>
<td>2. Record occurrence</td>
<td>Segment occurrence</td>
<td>Record occurrence</td>
</tr>
<tr>
<td>3. Field or data item</td>
<td>Field</td>
<td>Data item</td>
</tr>
<tr>
<td>4. Sequence field as key</td>
<td>Sequence field</td>
<td>Key</td>
</tr>
<tr>
<td>5. Parent-child relationship type</td>
<td>Physical parent-child relationship type</td>
<td>Hierarchical schema relationship</td>
</tr>
<tr>
<td>6. Virtual parent-child relationship type</td>
<td>Logical parent-child relationship type</td>
<td>Not provided except at run time</td>
</tr>
<tr>
<td>7. Hierarchical database schema</td>
<td>Physical or logical database definition (done in DBD)</td>
<td>Schema tree</td>
</tr>
<tr>
<td>8. Root of hierarchy</td>
<td>Root segment</td>
<td>Root record</td>
</tr>
<tr>
<td>9. Occurrence tree of a hierarchy</td>
<td>Physical database record</td>
<td>Data tree</td>
</tr>
<tr>
<td>10. Hierarchical record sequence</td>
<td>Hierarchical sequence</td>
<td>No special term</td>
</tr>
<tr>
<td>11. Pointer record type</td>
<td>Pointer segment type</td>
<td>No similar concept</td>
</tr>
</tbody>
</table>

**Physical Databases.** An IMS physical database (PDB) refers to the hierarchy that is actually stored. It is defined in the form of a physical DBD using the DL/I language. Figure 11.16 shows the definition of a physical database that corresponds to the hierarchy shown in Figure 11.5. It contains six segment types, each of which can have an arbitrary number of occurrences in the database. For the schema of Figure 11.10, we would need to use two physical database definitions. Later, appropriate logical databases would be defined based on these physical databases. The definition of the virtual parent-child relationships shown in Figure 11.10 is included in both of these physical DBDs and is quite complicated.

We make the following important points about the database definition:

- The database description is written in terms of the macros DBD, SEGM, FIELD, DBDGEM, FINISH, and END. The SEGM macro defines a segment, and the FIELD macro defines a field. Other macro names are self-explanatory.
- Each macro uses certain keywords. The hierarchical logical structure of the database is defined by virtue of the “PARENT =” specifications of the segments.
- The order of occurrence of the SEGM statements is the means of ordering segments within the logical schema. This top-to-bottom left-to-right ordering is significant. Changing this ordering yields a different physical database.
- A sequence field (optionally) designates a field within a segment type by which its occurrences may be ordered. The specific value of a sequence field is called the key of that segment occurrence.
1 DBD NAME = COMPANY
2 SEG M NAME = DEPARTMENT, BYTES = 28
3 FIELD NAME = DNAME, BYTES = 10, START = 1
4 FIELD NAME = DNUMBER, SEQ, BYTES = 6, START = 11
5 FIELD NAME = MGRNAME, BYTES = 3, START = 17
6 FIELD NAME = MGRSTARTDATE, BYTES = 9, START = 20

7 SEG M NAME = EMPLOYEE, PARENT = DEPARTMENT, BYTES = 79
8 FIELD NAME = NAME, BYTES = 31, START = 1
9 FIELD NAME = (SSN, SEQ), BYTES = 9, START = 32
10 FIELD NAME = BDATE, BYTES = 9, START = 41
11 FIELD NAME = ADDRESS, BYTES = 30, START = 50

12 SEG M NAME = DEPENDENT, PARENT = EMPLOYEE, BYTES = 25
13 FIELD NAME = (DEP_NAME, SEQ), BYTES = 15, START = 1
14 FIELD NAME = SEX, BYTES = 1, START = 16
15 FIELD NAME = BIRTHDATE, BYTES = 9, START = 17

16 SEG M NAME = SUPERVISEE, PARENT = EMPLOYEE, BYTES = 24
17 FIELD NAME = NAME, BYTES = 15, START = 1
18 FIELD NAME = SSN, BYTES = 9, START = 16

19 SEG M NAME = PROJECT, PARENT = DEPARTMENT, BYTES = 16
20 FIELD NAME = PNAME, BYTES = 10, START = 1
21 FIELD NAME = (PNUMBER, SEQ), BYTES = 6, START = 11

22 SEG M NAME = WORKER, PARENT = PROJECT, BYTES = 26
23 FIELD NAME = NAME, BYTES = 15, START = 1
24 FIELD NAME = (SSN, SEQ), BYTES = 9, START = 16
25 FIELD NAME = HOURS, BYTES = 2, START = 25
26 DBDGEN
27 FINISH
28 END

Figure 11.16 Physical database definition corresponding to the hierarchy of Figure 11.5.

• Sequence fields may be unique (which is the default) or nonunique. To designate a nonunique sequence field, one uses M (for multiple) in the FIELD definition, as:

  FIELD NAME = (PARTNAME, SEQ.M), . . . .

• A unique sequence field is required for the root segment if the database is stored by using HISAM and HIDAM (see Section 11.7.5), because this provides an index key for the primary index.

• Combinations of two or more fields are recognized as new fields. This allows a combination of fields to be treated as a composite key. For example, STATENAME, CITYNAME may together be given a new field name, say SCNAME, which can be defined as a sequence field.

An occurrence tree in our terminology is called a physical database record in IMS. The linearized form of an occurrence tree within a physical database record is produced by a preorder traversal of the segment occurrences (see Section 11.1.4). The sequence of segments from any segment up to the root—obtained by going through a series of successive parent segments—is called the segment’s hierarchical path. A concatenation of
keys (including segment type codes) along this path is called the **hierarchical sequence key** of that segment. The hierarchical sequence key of a **dependent** occurrence in a physical database record for the database of Figure 11.5 may be as follows:

1 | '000005' | 2 | '369278157' | 4 | 'JOHN...

Here, 1, 2, and 4 are, respectively, the segment type codes of **DEPARTMENT**, **EMPLOYEE**, and **DEPENDENT**, assigned automatically by **IMS**, and '000005', '369278157', and 'JOHN...' are sequence keys along the hierarchical path up to that segment occurrence for **JOHN**.

The physical database records in an **IMS** database occur in sequence by the key of its root segment. Within a physical database record, the segments occur in ascending order of their hierarchical sequence key.

**Type 1 Views in **IMS**—Subsets of Physical Databases.** **IMS** allows two kinds of views, or "logical databases"* (an **IMS** term), to be constructed from physical databases. For easy reference we call them type 1 and type 2 logical databases, or type 1 and type 2 views, which is our own nomenclature. Incidentally, additional confusion arises in **IMS** because only type 2 views are actually defined with a logical database definition; type 1 views are defined by using a **PCB**.

A type 1 logical database schema defines a **subhierarchy** of a physical database schema by observing these rules:

1. The root segment type must be part of the view.
2. Nonroot segment types may be omitted.
3. If a segment type is omitted, all its children segment types must be omitted.
4. From the segments that are included, any field type may be omitted.

For example, a large number of type 1 views can be defined for the database of Figure 11.5. Two valid type 1 views are shown in Figure 11.17(a). They are meaningful for two different applications. One deals with dependents, and the other deals with workers who work on projects.

Two other subhierarchies are shown in Figure 11.17(b). In the first, the **EMPLOYEE** segment is omitted but its child segment is included. This violates rule 3 of the preceding list. In the second subhierarchy, the root segment **DEPARTMENT** is omitted, which violates rule 1. Hence both of these subhierarchies are invalid views in **IMS**. Every type 1 hierarchy is defined by means of a **PCB**. We will not describe the **PCB** syntax here.

This view facility accomplishes the usual objectives of permitting selective access to only the relevant portion of a database, and it provides a certain amount of security. When the **PROCOPT** specification allows updates, the corresponding change allows updates to the "base" physical record. This is governed by a complicated set of rules in **IMS** and may lead to inconsistencies if not done properly. One **PSB** for a given application may include several **PCBs** that correspond to several type 1 views.

---

*IMS uses the term **logical database** loosely, with two meanings corresponding to the two types of views. A logical database or **LDB** is, however, defined only for the virtual hierarchy or type 2 view.
Type 2 Views in IMS over Multiple Physical Databases. This facility in IMS is a true view facility in that it allows one to create views that are virtual hierarchies. A virtual hierarchy consists of segments, some of which are connected by logical parent-child relationships (an IMS term)—we called them virtual parent-child relationships (VPCRs) in Section 11.2. By setting up logical relationships among segments from different physical databases, one can create a complex network. The type 2 view facility allows us to carve out any hierarchies from such a network. In Figure 11.18 we show virtual hierarchies based on the hierarchies of Figure 11.10. Type 2 views must be defined explicitly in IMS as logical databases (LDBs) by using the DBD macro and with ACCESS = LOGICAL.

Various rules govern logical parent-child relationships and the construction of logical databases from physical databases. Among the more important ones are:

1. The root of an LDB must be the root of some PDB.
2. A logical child segment must have one physical and only one logical parent. As a consequence, a root segment cannot be a logical child segment.
3. A physical child of a logical parent may appear as a dependent of a concatenated (logical child/logical parent) segment in the LDB. By virtue of this facility, we can create a logical database from Figure 11.10 as shown in Figure 11.18, which shows the use of an EMPLOYEE segment type twice (once as a supervisor and once as a supervisee) in the same logical database. This makes tracking currencies of segments even more difficult than we indicated in Section 11.6.1. This feature is a very powerful one in IMS and expands the scope of generating new hierarchies a great deal.
4. A logical parent segment type may have multiple logical child segment types. This is already seen for the EMPLOYEE segment in Figure 11.10.

Views in IMS versus Views in Relational Systems. We saw two types of view definitions or external schema facilities in IMS. Type 1 allowed views over single hierarchies; type
Figure 11.18 Type 2 views in IMS on the database in Figure 11.10. (a) Manager_View. (b) Project_View. (c) Dependent_View.

2 allowed views over multiple hierarchies. Let us compare this view facility to the views in relational systems:

1. A relational view does not have to be contemplated at the time of defining a conceptual schema or a set of base relations. In contrast, the definition of PDBs is determined by what LDBs need to use them. The IMS type 2 views are therefore not purely external schemas; they influence and determine the definition of the conceptual schema. The spirit of the three-schema architecture (see Section 2.2.1) is thus not fully maintained.

2. Relational views do not assume any physical access structure to support the views. IMS type 2 views, in contrast, require explicit definitions of pointers to link segments from multiple PDBs. The feasible LDBs are restricted by the types of pointers declared in the physical database(s).

3. The definition of a type 1 view is compulsory for an application to access a physical (or a logical) database. Even if the entire physical database is accessed by one application, a PCB (type 1 view) must be defined for it. In fact, a PCB is still required on top of an LDB to access it. There is no corresponding requirement in relational systems.

The type 2 view facility is a useful feature that extends the capabilities of IMS as follows:

- It allows a limited network facility by allowing two segments to have an M:N relationship via a common child pointer segment. N-ary relationships with N > 2 are not possible, unlike the situation in the network model.
- It reduces redundant storage of data. Correspondingly, updates may also be saved.
- Most important, it allows users to view the data in a variety of hierarchical ways besides the rigidly defined physical database hierarchies. This is done by combining segments from multiple existing hierarchies.
Unfortunately, the physical and logical database definitions, the different types of pairing of segments—physical and virtual—in "bidirectional relationships," and the complicated loading procedures for logical databases make type 2 views a very complex feature of IMS. We have left out a number of details in this discussion. It appears that some IMS installations do without the use of logical databases and are satisfied to rely entirely on physical databases.

11.7.4 Data Manipulation in IMS

Data manipulation operations in IMS closely parallel the HDML operations of Section 11.6. DL/1 includes the data manipulation language (DML) of IMS as well as the data definition language (DDL). We will not describe the detailed language syntax here. Instead we will show how DL/1 applications interface with IMS and will point out a few special features of DL/1. Calls to DL/1 are embedded in an IMS application program written in COBOL, PL/1, System 360/370 basic assembly language, or FORTRAN. This call has the following syntax:

\[
\text{CALL} \ <\text{procedure name}> \ (<\text{parameter list}>).
\]

The procedure name to be called varies depending on the language in which the application program is written. A PL/1 program must use the name PL1TDL1 (for PL/1 to DL/1), which is fixed. Consider the following query: "Obtain a list of dependents born after JAN-01-1980 for the employees in the departments with DNUMBER = 4". A call to DL/1 would be coded as

\[
\text{CALL PL1TDL1 (SIX, GU, PCB_1, DEPND_IO_AREA, DEPT_D_SSA, EMPL_SSA, DEPND_SSA)}
\]

This call appears in the application program that is assumed to be in PL/1. The parameter list is interpreted as follows:

- **SIX** refers to a variable containing the string 'SIX'. It indicates the number of remaining parameters in the list. Different queries can have different numbers of parameters.

- **GU** represents a variable containing the string 'GU', which stands for the operation to be performed—in this case, "get unique."

- **PCB_1** is the name of the structure defined in the PL/1 program that acts as a mask to address an area called the program communication block (PCB). It is the common area through which information is passed back and forth between IMS and the application program. Among other things, it includes a hierarchy level indicator, the processing options in effect, the current segment name, the current hierarchical sequence key; and the number of sensitive segments for the corresponding PCB definition.

- **DEPND_IO_AREA** is a 25-byte input-output area reserved in the program to receive the entire DEPENDENT segment.

- **D_SSA, EMPL_SSA, and DEPND_SSA** are segment search arguments, one per query. They stand for variable-containing strings that specify the search conditions for that statement. In our example the three strings would contain 'DEPARTMENT
(DNUMBER = '000004'), 'EMPLOYEE', and 'DEPENDENT(BDATE > 'JAN-01-1980')', respectively.

In the HDML language of Section 11.6, the effect of the above CALL would be to do the following query:

GET FIRST PATH DEPARTMENT, EMPLOYEE, DEPENDENT
WHERE DNUMBER = '000004' AND DEPENDENT.BDATE > 'JAN-01-1980'

The only difference is that in HDML we assumed that this would retrieve data for each of the segments; in IMS, with the Get Unique command, only the DEPENDENT segment (the terminal node of the path) is fetched into memory.

**CALL Interface versus Embedded Query Language Interface.** The above scenario demonstrates the use of a parameterized 'CALL interface' between a high-level programming language such as PL/1 and the DBMS. This should be contrasted with the embedded query language interface exemplified by the use of SQL in relational systems. The advantages of a CALL interface are:

1. The compiler for the host language remains unchanged; hence no precompilation is necessary.
2. The application program looks homogeneous; there is no intervention of any foreign syntax.

The main disadvantages are:

1. The positional parameters in a CALL can easily be interchanged or omitted.
2. By looking at a CALL statement, it is impossible to judge the embedded data retrieval update operations. Hence application programs become difficult to read.
3. There are no semantic checks on the parameters in a CALL. Hence errors may surface later at execution time without being detected at compile time.

In our opinion the CALL interface may be convenient for the implementer but it is not desirable for application development.

Table 11.3 summarizes the correspondence between the operations proposed in HDML and those that exist in DL/1. The DL/1 operations are invoked using the CALL facility described above. In Section 11.6 we used a different notation, where the HDML commands were embedded in PASCAL programs and the search conditions were written with a WHERE clause. This distinction must be kept in mind while reading this table. We offer next a few additional explanations corresponding to the notes in Table 11.3.

- **Note 1:** Each time a hierarchy is processed, in general, IMS requires that the first command be a Get Unique (GU), which must address a hierarchical path in the hierarchy, starting with the root. Accessing segments within the hierarchy directly is not possible. (Exceptions exist but are beyond the scope of this discussion.)

- **Note 2:** Get Unique in DL/1 is used to account for GET FIRST as well as GET FIRST PATH of HDML. For Examples 1, 2, and 3 of Section 11.6.2, GU of DL/1 would work exactly like GET FIRST of HDML. Example 4 of Section 11.6.3 is shown in DL/1 plus a host language below (in a pseudosyntax without coding as an exact CALL).
<table>
<thead>
<tr>
<th>HDML Operation</th>
<th>DL/1 Operation</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Get First (GF)</td>
<td>Get Unique (GU)</td>
<td>Get the first occurrence of a specified record (see Note 1)</td>
</tr>
<tr>
<td>Get Next (GN)</td>
<td>Get Next (GN)</td>
<td>Get the next occurrence of a specified record</td>
</tr>
<tr>
<td>Get First Path</td>
<td>Get Unique (GU) plus command code *D</td>
<td>Get the first occurrence of all records along a hierarchical path (see Note 2)</td>
</tr>
<tr>
<td>Get Next Path</td>
<td>Get Next within Parent (GNP) or Get Next (GN)</td>
<td>Get the next occurrence of a specified hierarchical path (see Note 3)</td>
</tr>
<tr>
<td>Get Next within Parent</td>
<td>Get Next within Parent (GNP)</td>
<td>Get the next child occurrence for the current parent occurrence</td>
</tr>
<tr>
<td>Get Next within Virtual Parent</td>
<td>No special operation</td>
<td>Get the next child occurrence for the current virtual parent occurrence (see Note 3)</td>
</tr>
<tr>
<td>INSERT</td>
<td>INSERT (ISRT)</td>
<td>Insert new record occurrence</td>
</tr>
<tr>
<td>DELETE</td>
<td>DELETE (DELT)</td>
<td>Delete old record occurrence</td>
</tr>
<tr>
<td>REPLACE</td>
<td>REPLACE (REPL)</td>
<td>Replace current record occurrence with a new occurrence</td>
</tr>
<tr>
<td>Get with Hold</td>
<td>Get Unique with Hold (GHU)</td>
<td>Perform a corresponding GET operation with a hold on the record so that it can be subsequently replaced or deleted</td>
</tr>
<tr>
<td></td>
<td>Get Next with Hold (GHN)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Get Next within Parent with Hold (GHNP)</td>
<td></td>
</tr>
</tbody>
</table>
This query retrieves employee-dependent pairs where both have the first name John. The segment search arguments (e.g., "EMPLOYEE*D (FNAME = 'JOHN')") are shown next to the operation for notational convenience.

**GU** EMPLOYEE *D (FNAME = 'JOHN')
DEPENDENT (DEPNAME = 'JOHN')
while DB_STATUS = 'segment found' do
  begin
    WRITE EMPLOYEE NAME, EMPLOYEE BIRTHDATE, DEPENDENT BIRTHDATE
    **GN** EMPLOYEE *D (FNAME = 'JOHN')
    DEPENDENT (DEPNAME = 'JOHN')
  end;

The *D for data in the preceding example is called a **command code**. Notice that Get Next with a *D produces the same effect as Get Next Path in HDML. It would be possible in DL/I to code this example without using *D. In that case, instead of the entire path, only the terminal segment (DEPENDENT, in the above example) would be retrieved.

- **Note 3**: There are no special commands in IMS for processing virtual ("logical" in IMS) parent-child relationships, because virtual relationships **cannot be processed directly without defining a logical database**.

A few additional observations comparing DL/I with HDML are in order. The hold option was demonstrated for HDML in Example 10 of Section 11.6.5. It applies to all forms of the GET operation in IMS. The while ... do ... end loops shown in the PASCAL plus HDML examples in Section 11.6 were controlled by a DB_STATUS code. Loops must be explicitly coded in the same way in the host language for IMS. Status code is available as part of the PCB area.

IMS provides a command code *F that allows an application to search in the hierarchy to determine whether a condition is satisfied and then move back to a previous named segment within the same physical database record and retrieve data. The *V command code is used to position the retrieval processing in the current of a particular segment type (see Section 11.6.2). With these command codes, it is possible to change the normal forward direction of processing through the linearized hierarchical sequences.

An IMS application program can open several PCBs or type 1 views and process them concurrently. Currency of segment types within each PCB is maintained by the system. Each CALL includes one parameter referring to a specific PCB.

### 11.7.5 Storage of Databases in **IMS**

We shall review the different types of file organizations available to store physical databases in **IMS** without going into details. In **IMS** these are referred to as access methods. Compared to most DBMSs, **IMS** provides a far wider array of access methods. We mention all of them here:

- Each physical database in **IMS** is a stored database. The logical databases are virtual hierarchical databases that may be viewed as such, but in storage they do not
represent separate data. Logical databases consist of the physical databases plus linkages provided by pointer structures.

- Each stored segment contains stored data fields plus a prefix (not visible to user programs) containing a segment type code, pointers, a delete flag, and other control information.

- Regardless of the access method, a stored database is always stored as a sequence of occurrence trees, called **physical database records**, where each occurrence tree contains a preorder sequence of segments (see Sections 11.1.3 and 11.1.4). It is owned by a specific root segment. For brevity, we shall sometimes refer to an occurrence tree as a **tree**.

- Various IMS access methods differ as to how the sequence of segments is "tied together" within a physical database record and what type of access structure is provided to locate the physical database record or individual segment occurrences within it.

- Based on the type of access provided to the physical database records, two types of structures are provided in IMS: **hierarchical sequential** (HS) and **hierarchical direct** (HD). They are further divided into HSAM, HISAM, HDAM, and HIDAM, as shown in Figure 11.19.

The IMS access methods may be regarded as high-level ones. IMS provides routines for HISAM, HDAM, and so on, that in turn use the lower-level access methods called SAM (sequential access method), OSAM (overflow sequential access method), and ISAM (indexed sequential access method). A combination of files at the lower-level access method is used by the higher-level access method (see Figure 11.19).

**HSAM.** HSAM ties together the segments within a tree by **physical contiguity**. The trees themselves are placed in physical sequence in storage. Each physical database record represents one occurrence tree. HSAM "strings out" the physical records sequentially, in order of the sequence key of the root segment, with a fixed block size. This organization is

![Figure 11.19](image-url)  
**Figure 11.19** An overview of IMS access methods.
“tapelike” and is of academic importance only, since it supports no updating. However, it can be used for dumping and transporting databases. Once a database is loaded (by using ISRT commands), only the GET operations (excluding GET HOLD) are allowed. To do modification/deletion/insertion of data, the old database is read in and an entire new copy is written out. This organization is used to process data that remains fixed over an extended period of time.

**HISAM.** In the HISAM organization, the database consists of two files or storage areas or data sets: one file (prime area) contains root segments (plus some additional segments that fit within the record in that file); another file (overflow area) contains the remaining tail portion of each linearized tree. Figure 11.20 shows how the physical database record corresponding to Figure 11.7 would be stored in IMS using HISAM. It shows how a record is split into two files and how fixed-length blocks are linked together within the overflow.

Both files have fixed-length records. Hence, because of the uneven lengths of segments, some space is wasted at the end of records. The first file is either an ISAM or a prime VSAM area that is accessible via an index on the sequence field of the root segment. This is a popular access method with IMS.

---

**Figure 11.20** HISAM file organization in IMS.
HDAM. An HDAM database consists of a single OSAM or VSAM file. In both HD structures, the root segment is accessed by a hash on the sequence field; segments are stored independently and linked by means of two types of pointers:

- **Hierarchical pointers**: With hierarchical pointers, each segment points to the next in hierarchical sequence, except for the last dependent segment in the hierarchy, which does not bear a pointer. This is essentially the preorder threading of the tree.

- **Child/twin pointers**: Figure 11.21 shows the use of child/twin pointers. Each segment type has a designated number of child pointers equal to the number of children segment types defined in the database definition and has one twin pointer. For a given tree, the child pointer in a segment can be (a) a null value if it has no child segment of the corresponding type (for example, no dependent for Zelaya or Jabbar in Figure 11.21) or (b) the location of the first child segment of the corresponding type. Children of the same parent are linked by using a twin pointer. The twin pointer of the last child is a null.

Note that the HDAM organization provides no sequential access on the root segment. It can be provided by means of a secondary index. Backward pointers can be created in addition to forward pointers. Declaration of pointers is done as part of the database definition.

![Diagram](image-url)  
**Figure 11.21** HDAM with child/twin pointers in IMS.
From a performance standpoint, hierarchical pointers are favored when requirements dictate direct access to the root coupled with sequential access on dependent segments, as in producing reports, where a variety of different segments must be listed. Child/twin pointers are favored when quick access to lower levels in the hierarchy or to the right bottom part of the tree is desired. HDAM differs from the other three access methods in that the initial loading of the database can be done in any (random) sequence, a tree at a time.

**HIDAM.** A HIDAM database consists of two parts: an index database and a “data” database. The data database is a single OSAM or VSAM (entry-sequenced data set) file consisting of fixed-length records that are initially loaded in hierarchical sequence. It is treated as an HDAM database by itself. Either hierarchical or child/twin pointer schemes are used to link segments.

The index database is a HISAM database in which the hierarchy consists of only one type of segment, an index segment. Each index segment contains a root sequence field value as its key and the pointer to that root segment in the data database as its data field. The index database is much smaller than the data database. If VSAM is used, a single key-sequenced data set suffices as the index database. Space utilization of the index database is more efficient with VSAM than with ISAM/OSAM. HIDAM databases are popular because they combine the advantages of HISAM and HDAM. Direct access to the root segments in the database is provided by using the root sequence key as a hash key; indexed access is available by going through the index database.

**Other IMS Storage Structures.** Besides the access methods described previously, IMS provides the following additional storage structures:

1. **Simple HSAM (SHSAM), and simple HISAM (SHISAM)** are variants of HSAM and HISAM, respectively, where a database contains only one segment type (the root segment).

2. **The fast path feature** in IMS is designed for on-line transaction systems with high transaction rates and relatively simple processing. It provides data communication facilities and two special database structures:
   a. **Main storage databases (MSDBs):** An MSDB is a root-only database. It is kept in primary memory throughout system operation. Small reference tables such as conversion tables and timetables are good candidates for MSDBs.
   b. **Data entry databases (DEDBs):** A DEDB is a special form of HDAM database designed for better availability and performance. It is a restricted form of hierarchy with only two levels, and it may be partitioned into up to 240 areas. The leftmost segment at the second level, called the *sequential dependent segment type*, is given special treatment. Each area is a separate VSAM data set, and each database record (root plus all dependents) is wholly contained in the area. The partitioning is not visible to the application.

3. **Secondary data set groups (DSGs):** A HISAM, HDAM, or HIDAM database can be partitioned into groups of segment types. One primary data set group and nine secondary data set groups can be created. The primary DSG contains the root
segment. Each secondary DSG is a separate database containing all segment occurrences for the type of segments that belong to it.

**Secondary Indexing in IMS.** In Chapter 5 we saw the importance of secondary indexes for improving access times in various types of files. IMS provides only the following two types of secondary indexes:

1. An index that provides access to a root or a dependent segment based on the value of any field in it.
2. An index that indexes a given segment on the basis of a field in some segment at a lower level. For our database of Figure 11.5, some of the possible secondary indexes are:
   
   A. An index to DEPARTMENT by department name (DNAME).
   B. An index on DEPENDENT by birthdate (BIRTHDATE).
   C. An index on DEPARTMENT by location of project in that department (LOCATION in PROJECT).

There are two shortcomings with secondary index processing in IMS:

1. The DL/1 code must refer explicitly to an index if it is to be used; otherwise processing is done without the index.
2. When a field in a lower-level segment in the hierarchy is used for indexing, the hierarchy is visualized as if it were restructured with that segment as the root.

Both of these characteristics directly violate the objective of data independence (see Section 2.2) whereby a user’s external view is insulated from the internal organization of the database.

### 11.8 Summary

In this chapter we discussed the hierarchical model, which represents data by emphasizing hierarchical relationships. The presentation was general, although some aspects were patterned after the major hierarchical system—IBM’s IMS. Departures from IMS were mostly pointed out (without detailed discussion) in the text and in footnotes. The main structures used by the hierarchical model are record types and parent-child relationship (PCR) types. Each PCR type defines a hierarchical 1:N relationship between a parent record type and a child record type. Relationships are strictly hierarchical in that a record type can participate as child in at most one PCR type. This restriction makes it difficult to represent a database where numerous relationships exist.

We then saw how hierarchical database schemas can be defined as a number of hierarchical schemas of record types. A hierarchical schema is basically a tree data structure. Corresponding to a hierarchical schema, a number of occurrence trees will exist in the database. The hierarchical sequence of storing database records from an occurrence tree is a preorder traversal of the records in an occurrence tree. The type of each record is stored with the record so that the DBMS can identify the records while searching through records of a hierarchical sequence.
We then discussed the limitations of hierarchical representation when we try to represent M:N relationships, or relationships in which more than two record types participate. It is possible to represent some of these cases by allowing redundant records to exist in the database. The concept of virtual parent-child relationship (VPCR) types is used to permit a record type to have two parents—a real parent and a virtual parent. This VPCR type can also be used to represent M:N relationships without redundancy of database records. We also discussed the types of implicit integrity constraints in hierarchies.

In Section 11.5, we discussed hierarchical database design from an ER conceptual schema. In general, the hierarchical model works well for database applications that are naturally hierarchical. However, when there are many nonhierarchical relationships, trying to fit those relationships into a hierarchical form is difficult, and the resulting representations are often unsatisfactory. We then presented the commands of a hypothetical hierarchical data definition language (HDDL) and of a record-at-a-time hierarchical data manipulation language (HDML). The HDML is based on the hierarchical sequence. We saw how to write programs with embedded HDML commands to retrieve information from a hierarchical database and to update the database.

Finally, we gave the basic architecture, language features, and storage organization of the popular hierarchical database system, IMS.

Although the relational model and relational DBMSs have recently become quite popular, the hierarchical model will be with us for several years to come because a big investment has been made in hierarchical DBMSs in the commercial world. In addition, the hierarchical model is quite suitable for situations where the majority of relationships are hierarchical and where database access mainly uses these hierarchical relationships.

Review Questions

11.1. Define the following terms: parent-child relationship (PCR) type, root of a hierarchy, leaf of a hierarchy.

11.2. Discuss the main properties of a hierarchy.

11.3. Discuss the problems with using a PCR type to represent an M:N relationship.

11.4. What is an occurrence tree of a hierarchy?

11.5. What is the hierarchical sequence? Why is it necessary to assign a record type field to each record when the hierarchical sequence is used to represent an occurrence tree?

11.6. Define the following terms: hierarchical path, broom, forest of trees, hierarchical database.

11.7. What are virtual parent-child relationship (VPCR) types? How do they enhance the modeling power of the hierarchical model?

11.8. Discuss different techniques that may be used for implementing VPCR types in a hierarchical database.

11.9. Discuss the inherent integrity constraints of the hierarchical model.
11.10. Show how each of the following types of relationships is represented in the hierarchical model: (a) M:N relationships; (b) n-ary relationships, with n > 2; (c) 1:1 relationships. Discuss how an ER schema can be mapped to a hierarchical schema.

11.11. Why is it necessary to embed the HDML commands in a host programming language such as PASCAL?

11.12. Discuss the following concepts, and identify what each is used for when writing an HDML database program: (a) the user work area (UWA); (b) currency indicators; (c) database status indicator.

11.13. Discuss the different types of GET commands of the HDML, and tell how each affects the currency indicators.

11.14. Discuss how parent records are established as a result of a retrieval command. Why does the GET NEXT WITHIN PARENT command not establish a new parent?

11.15. Discuss the update commands of the HDML.

Exercises

11.16. Specify the queries of Exercise 6.19 in HDML embedded in PASCAL on the hierarchical database schema of Figure 11.10. Use the PASCAL program variables declared in Figure 11.11, and declare any additional variables you may need.

11.17. Consider the hierarchical database schema shown in Figure 11.22, which corresponds to the relational schema of Figure 2.1. Write appropriate HDDL statements to define the record types and set types of the schema.

11.18. There is some redundancy in the schema of Figure 11.22; what data items are repeated redundantly? Can you specify a hierarchical database schema for this database without redundancy by using VFCRs?

---

**Figure 11.22** Hierarchical schema for a university database.
11.19. Write PASCAL program segments with embedded HDML commands to specify the queries of Exercise 7.16 on the schema of Figure 11.22. Repeat the same queries for your schema of Exercise 11.18.

11.20. Write PASCAL program segments with embedded HDML commands to do the updates and tasks of Exercises 7.17 and 7.18 on the hierarchical database schema of Figure 11.22. Specify any program variables that you need. Repeat the same queries for your schema of Exercise 11.18.

11.21. Choose some database application that you are familiar with or interested in.
   a. Design a hierarchical database schema for your database application.
   b. Declare your record types, PCR types, and VPCR types, using the HDDL.
   c. Specify a number of queries and updates that are needed by your database application, and write a PASCAL program segment with embedded HDML commands for each of your queries.
   d. Implement your database if you have a hierarchical DBMS system available.

11.22. Map the following ER schemas into hierarchical schemas. Specify for each ER schema one or more hierarchical schemas, and state which have redundancies and which do not.
   a. The AIRLINES ER schema of Figure 3.19.
   b. The BANK ER schema of Figure 3.20.
   c. The SHIP_TRACKING ER schema of Figure 6.21.

Selected Bibliography


The Time-shared Data Management System (TDMS) of System Development Corporation (now Burroughs) (Vorhaus and Mills 1967; Bleier and Vorhaus 1968) and the Remote File Management System (RFMS) developed at the University of Texas at Austin (Everett et al. 1971) are precursors of another major commercial hierarchical system called System 2000, which is now marketed by SAS Inc. Hardgrave (1974, 1980) describes a language, BOLT, for the hierarchical model.