A P P E N D I X B

Hierarchical Model

In the network model, the data are represented by collections of *records* and relationships between data are represented by *links*. This structure holds for the hierarchical model as well. The only difference is that, in the hierarchical model, records are organized as collections of trees, rather than as arbitrary graphs.

B.1 Basic Concepts

A hierarchical database consists of a collection of *records* that are connected to each other through *links*. A record is similar to a record in the network model. Each record is a collection of fields (attributes), each of which contains only one data value. A link is an association between precisely two records. Thus, a link here is similar to a link in the network model.

Consider a database that represents a *customer-account* relationship in a banking system. There are two record types: *customer* and *account*. The *customer* record type can be defined in the same manner as in Appendix A. It consists of three fields: *customer-name*, *customer-street*, and *customer-city*. Similarly, the *account* record consists of two fields: *account-number* and *balance*.

A sample database appears in Figure B.1. It shows that customer Hayes has account A-305, customer Johnson has accounts A-101 and A-201, and customer Turner has account A-305.

Note that the set of all customer and account records is organized in the form of a rooted tree, where the root of the tree is a dummy node. As we shall see, a hierarchical database is a collection of such rooted trees, and hence forms a forest. We shall refer to each such rooted tree as a *database tree*.

The content of a particular record may have to be replicated in several different locations. For example, in our customer-account banking system, an account may belong to several customers. The information pertaining to that account, or the information pertaining to the various customers to which that account may belong, will

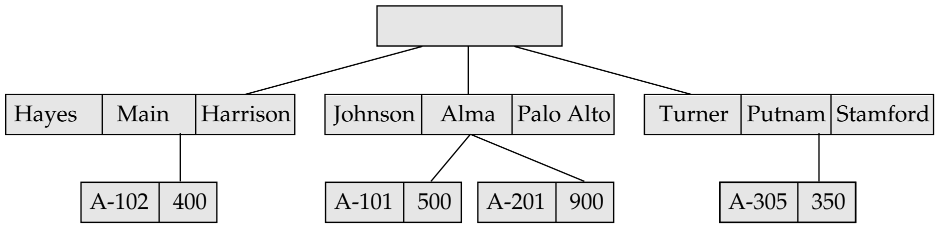


Figure B.1 Sample database.

have to be replicated. This replication may occur either in the same database tree or in several different trees. Record replication has two major drawbacks:

1. Data inconsistency may result when updating takes place.
2. Waste of space is unavoidable.

We shall deal with this issue in Section B.5 by introducing the concept of a *virtual record*.

B.2 Tree-Structure Diagrams

A *tree-structure diagram* is the schema for a hierarchical database. Such a diagram consists of two basic components:

1. *Boxes*, which correspond to record types
2. *Lines*, which correspond to links

A tree-structure diagram serves the same purpose as an entity–relationship (E-R) diagram; namely, it specifies the overall logical structure of the database. A tree-structure diagram is similar to a data-structure diagram in the network model. The main difference is that, in the latter, record types are organized in the form of an arbitrary graph, whereas in the former, record types are organized in the form of a *rooted tree*.

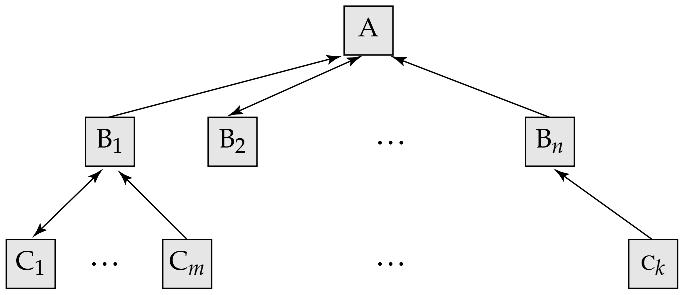


Figure B.2 General structure of a tree-structure diagram.

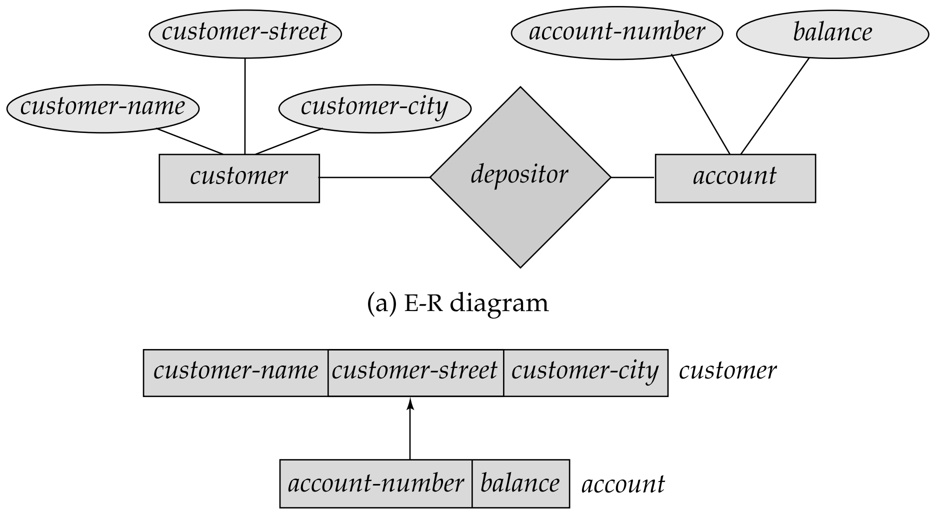
We have to be more precise about what a *rooted tree* is. First, there can be no cycles in the underlying graph. Second, there is a record type that is designated as the root of the tree. The relationships formed in the tree structure diagram must be such that only one-to-many or one-to-one relationships exist between a parent and a child. The general form of a tree-structure diagram appears in Figure B.2. Note that the arrows are pointing from children to parents. A parent *may* have an arrow pointing to a child, but a child *must* have an arrow pointing to its parent.

The database schema is represented as a collection of tree-structure diagrams. For each such diagram, there exists one *single* instance of a database tree. The root of this tree is a dummy node. The children of the dummy node are instances of the root record type in the tree structure diagram. Each record instance may, in turn, have several children, which are instances of various record types, as specified in the corresponding tree-structure diagram.

To understand how tree-structure diagrams are formed, we shall show how to transform E-R diagrams to their corresponding tree-structure diagrams. We first show how to apply such transformations to single relationships. We then explain how to ensure that the resulting diagrams are in the form of rooted trees.

B.2.1 Single Relationships

Consider the E-R diagram of Figure B.3a; it consists of the two entity sets *customer* and *account* related through a binary, one-to-many relationship *depositor*, with no descriptive attributes. This diagram specifies that a customer can have several accounts, but an account can belong to only one customer. The corresponding tree-structure diagram appears in Figure B.3b. The record type *customer* corresponds to the entity



(a) E-R diagram

(b) Tree-structure diagram

Figure B.3 E-R diagram and its corresponding tree-structure diagram.

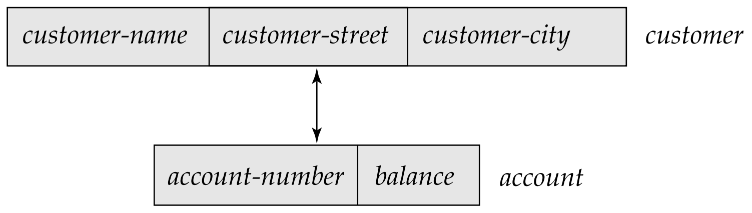


Figure B.4 Tree-structure diagram with one-to-one relationship.

set *customer*. It includes three fields: *customer-name*, *customer-street*, and *customer-city*. Similarly, *account* is the record type corresponding to the entity set *account*. It includes two fields: *account-number* and *balance*. Finally, the relationship *depositor* has been replaced with the link *depositor*, with an arrow pointing to *customer* record type.

An instance of a database corresponding to the described schema may thus contain a number of *customer* records linked to a number of *account* records, as in Figure B.1. Since the relationship is one to many from *customer* to *account*, a customer can have more than one account, as does Johnson, who has both accounts A-101 and A-201. An account, however, cannot belong to more than one customer; none do in the sample database.

If the relationship *depositor* is one to one, then the link *depositor* has two arrows: one pointing to *account* record type, and one pointing to *customer* record type (Figure B.4). A sample database corresponding to this schema appears in Figure B.5. Since the relationship is one to one, an account can be owned by precisely one customer, and a customer can have only one account, as is indeed the case in the sample database.

If the relationship *depositor* is many to many (see Figure B.6a), then the transformation from an E-R diagram to a tree-structure diagram is more complicated. Only one-to-many and one-to-one relationships can be directly represented in the hierarchical model.

There are many different ways to transform this E-R diagram to a tree-structure diagram. All these diagrams, however, share the property that the underlying database tree (or trees) will have replicated records.

The decision regarding which transformation should be used depends on many factors, including

- The type of queries expected on the database

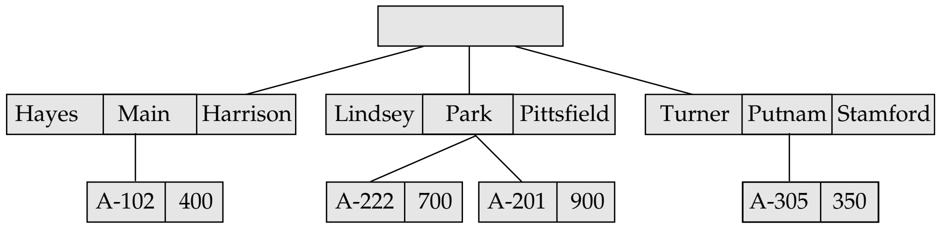


Figure B.5 Sample database corresponding to diagram of Figure B.4.

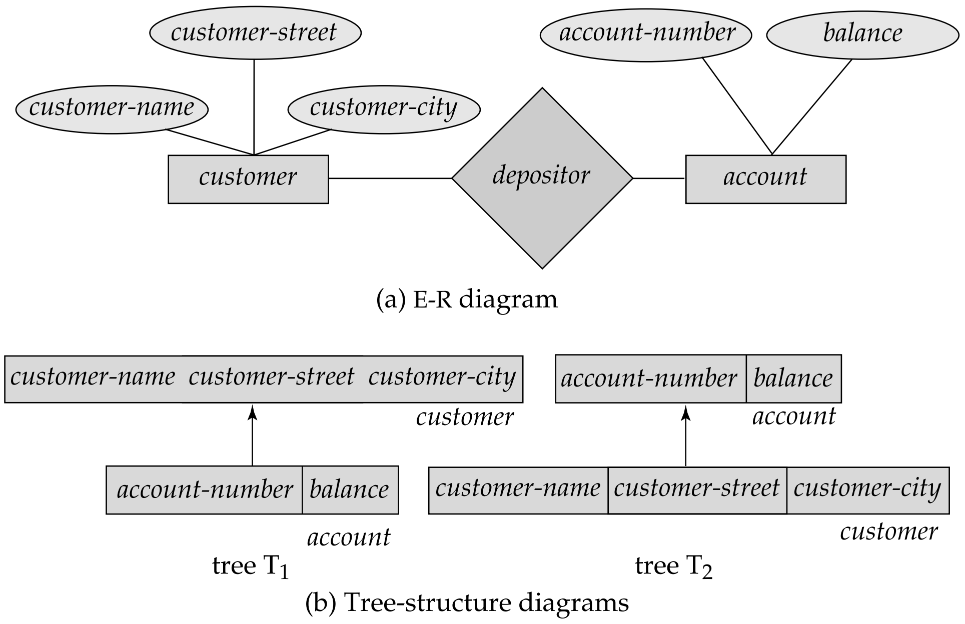


Figure B.6 E-R diagram and its corresponding tree-structure diagrams.

- The degree to which the overall database schema being modeled fits the given E-R diagram

We shall present a transformation that is as general as possible. That is, all other possible transformations are a special case of this one transformation.

To transform the E-R diagram of Figure B.6a into a tree-structure diagram, we take these steps:

1. Create two separate tree-structure diagrams, T_1 and T_2, each of which has the *customer* and *account* record types. In tree T_1, *customer* is the root; in tree T_2, *account* is the root.

2. Create the following two links:
   - *depositor*, a many-to-one link from *account* record type to *customer* record type, in T_1
   - *account-customer*, a many-to-one link from *customer* record type to *account* record type, in T_2

The resulting tree-structure diagrams appear in Figure B.6b. The presence of two diagrams (1) permits customers who do not participate in the *depositor* relationship as well as accounts that do not participate in the *depositor* relationship, and (2) permits efficient access to account information for a given customer as well as customer information for a given account.

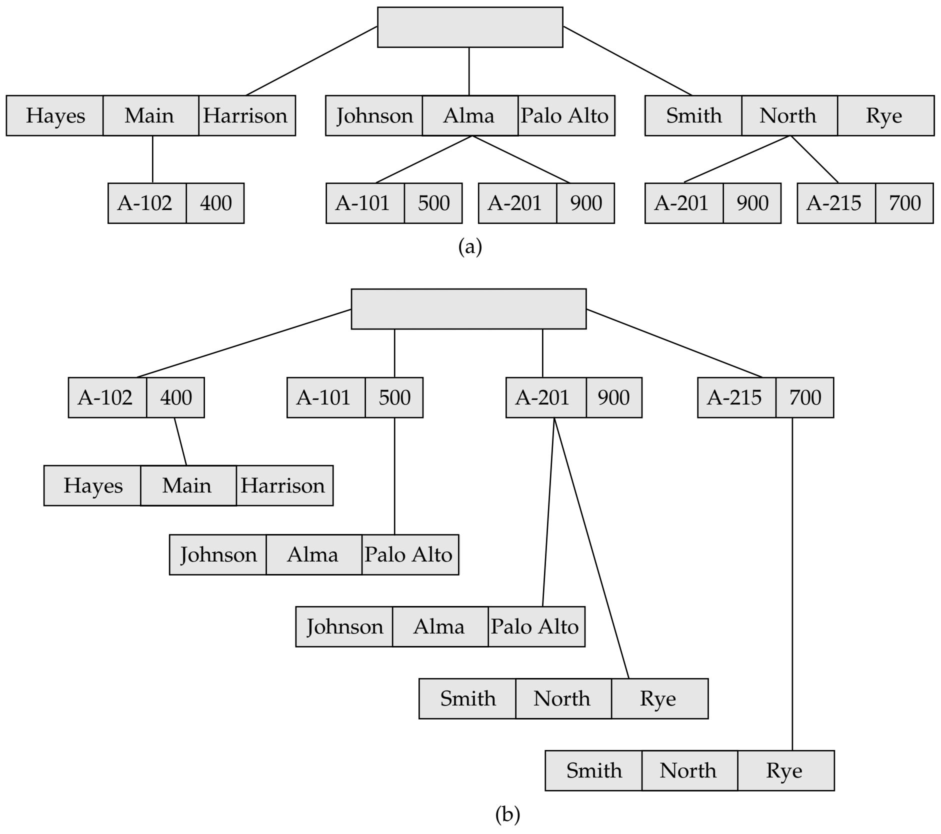
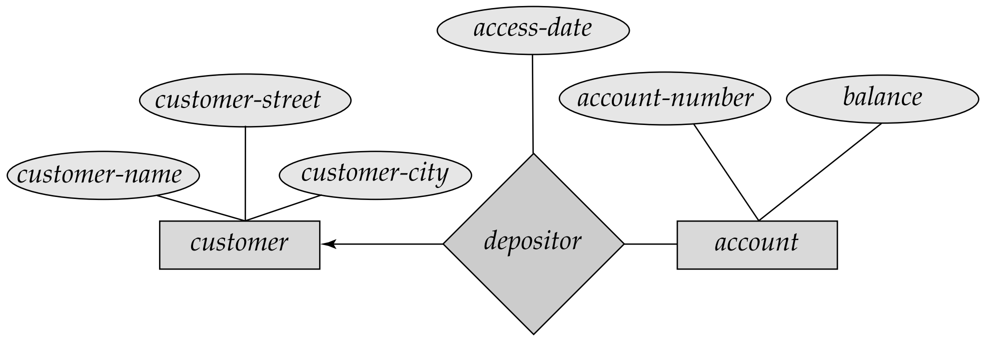


Figure B.7 Sample database corresponding to diagram of Figure B.6b.

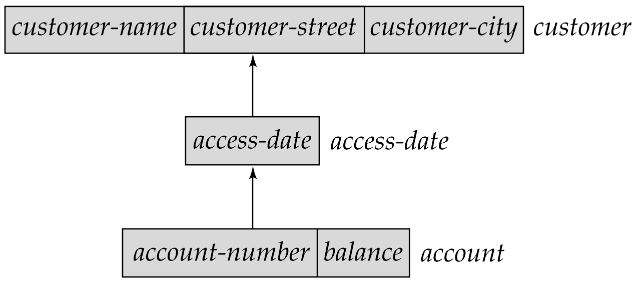
A sample database corresponding to the tree-structure diagram of Figure B.6b appears in Figure B.7. There are two database trees. The first tree (Figure B.7a) corresponds to the tree-structure diagram T_1; the second tree (Figure B.7b) corresponds to the tree-structure diagram T_2. As we can see, all *customer* and *account* records are replicated in both database trees. In addition, *account* record A-201 appears twice in the first tree, whereas *customer* records Johnson and Smith appear twice in the second tree.

If a relationship also includes a descriptive attribute, the transformation from an E-R diagram to a tree-structure diagram is more complicated. A link cannot contain any data value. In this case, a new record type needs to be created, and the appropriate links need to be established. The manner in which links are formed depends on the way the relationship *depositor* is defined.

Consider the E-R diagram of Figure B.3a. Suppose that we add the attribute *access-date* to the relationship *depositor*, to denote the most recent date on which a customer accessed the account. This newly derived E-R diagram appears in Figure B.8a. To transform this diagram into a tree-structure diagram, we must



(a) E-R diagram



(b) Tree-structure diagram

Figure B.8 E-R diagram and its corresponding tree-structure diagram.

1. Create a new record type *access-date* with a single field.
2. Create the following two links:
   - *customer-date*, a many-to-one link from *access-date* record type to *customer* record type
   - *date-account*, a many-to-one link from *account* record type to *access-date* record type

The resulting tree-structure diagram is illustrated in Figure B.8b.

An instance corresponding to the described schema appears in Figure B.9. It shows that:

- Hayes has account A-102, which was last accessed on 10 June 1996.
- Johnson has two accounts: A-101, which was last accessed on 24 May 1996, and A-201, which was last accessed on 17 June 1996.
- Turner has account A-305, which was last accessed on 10 June 1996.

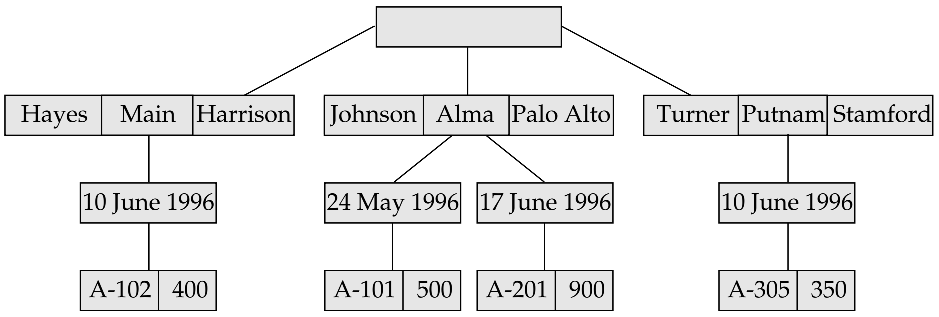


Figure B.9 Sample database corresponding to diagram of Figure B.8b.

Note that two different accounts can be accessed on the same date, as were accounts A-102 and A-305. These accounts belong to two different customers, so the *access-date* record must be replicated to preserve the hierarchy.

If the relationship *depositor* were one to one with the attribute *date*, then the transformation algorithm would be similar to the one described. The only difference would be that the two links *customer-date* and *date-account* would be one-to-one links.

Assume that the relationship *depositor* is many to many with the attribute *access-date*; here again, we can choose among a number of alternative transformations. We shall use the most general transformation; it is similar to the one applied to the case where the relationship *depositor* has no descriptive attribute. The record types *customer*, *account*, and *access-date* need to be replicated, and two separate tree-structure diagrams must be created, as in Figure B.10. A sample database corresponding to this schema is in Figure B.11.

Until now, we have considered only binary relationships. We shift our attention here to general relationships. The transformation of E-R diagrams corresponding to general relationships into tree-structure diagrams is complicated. Rather than present a general transformation algorithm, we present a single example to illustrate the overall strategy that you can apply to deal with such a transformation.

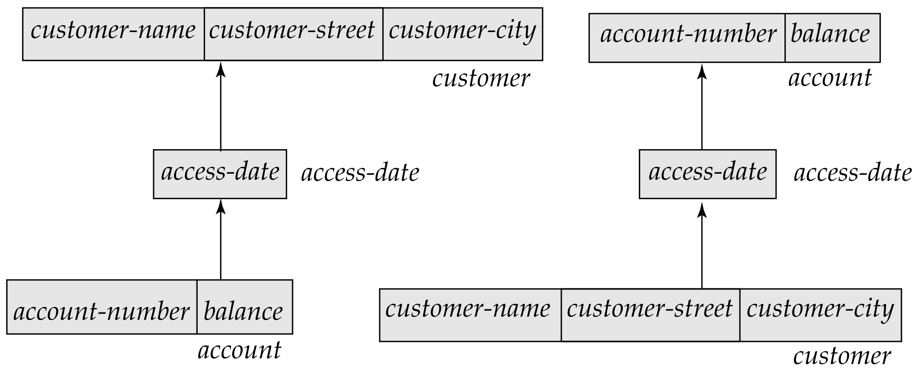


Figure B.10 Tree-structure diagram with many-to-many relationships.

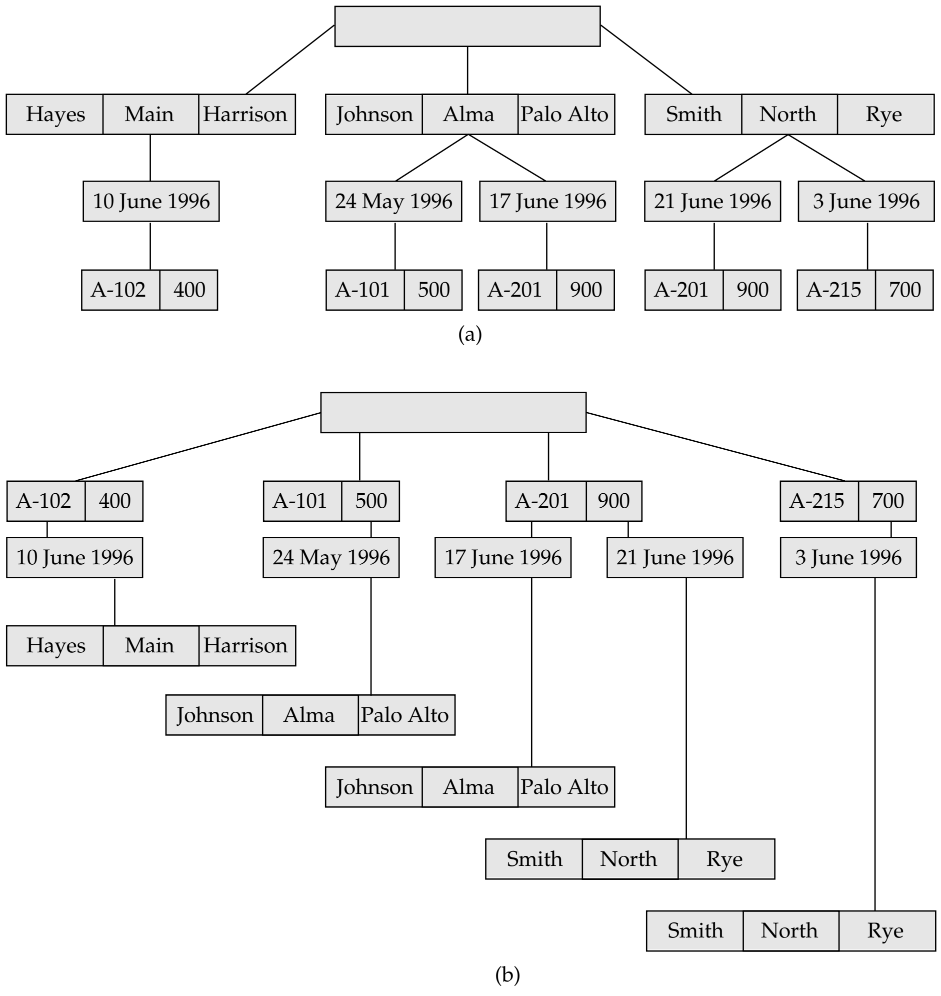
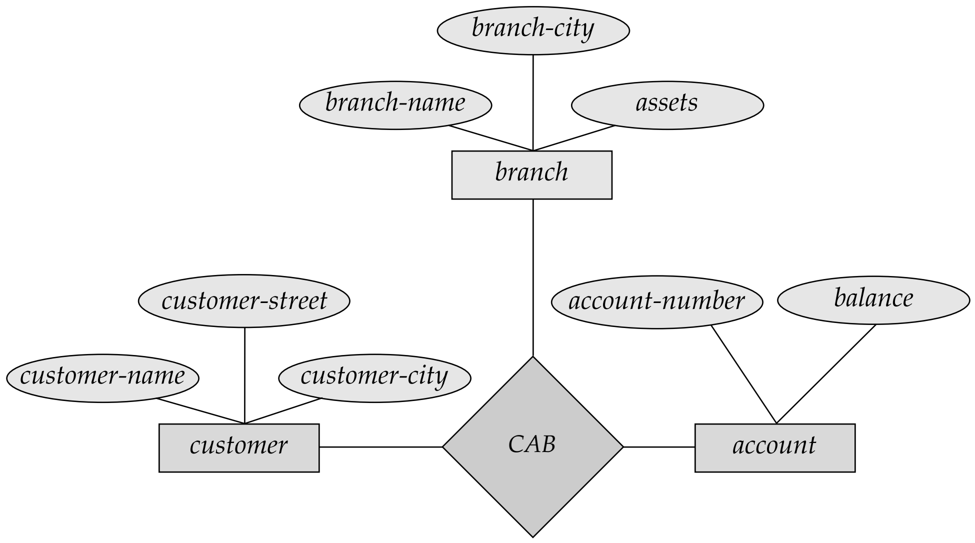


Figure B.11 Sample database corresponding to diagram of Figure B.10.

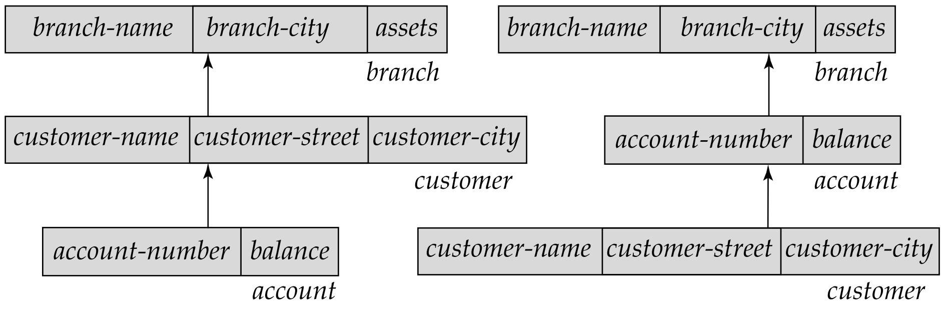
Consider the E-R diagram of Figure B.12a, which consists of the three entity sets *customer*, *account*, and *branch*, related through the general relationship set *CAB* with no descriptive attribute.

There are many different ways to transform this E-R diagram into a tree-structure diagram. Again, all share the property that the underlying database tree (or trees) will have replicated records. The most straightforward transformation is to create two tree-structure diagrams, as shown in Figure B.12b.

An instance of the database corresponding to this schema is illustrated in Figure B.13. It shows that Hayes has account A-102 in the Perryridge branch; Johnson has accounts A-101 and A-201 in the Downtown and Perryridge branches, respectively; and Smith has accounts A-201 and A-215 in the Perryridge and Mianus branches, respectively.



(a) E-R diagram



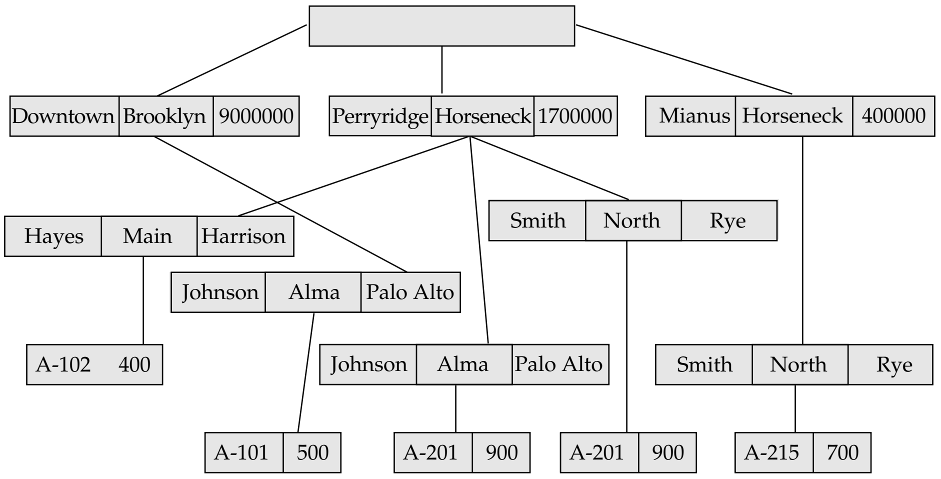
(b) Tree-structure diagrams

Figure B.12 E-R diagram and its corresponding tree-structure diagrams.

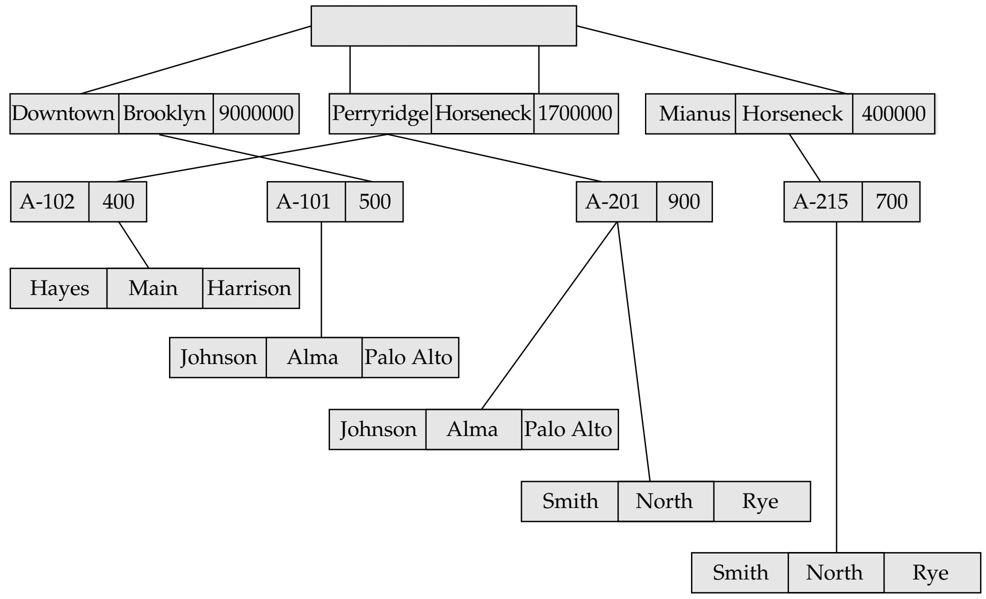
We can extend the preceding transformation algorithm in a straightforward manner to deal with relationships that span more than three entity sets. We simply replicate the various record types, and generate as many tree-structure diagrams as necessary. We can extend this approach, in turn, to deal with a general relationship that has descriptive attributes. We need only to create a new record type with one field for each descriptive attribute, and then to insert that record type in the appropriate location in the tree-structure diagram.

B.2.2 Several Relationships

The scheme that we have described to transform an E-R diagram to a tree-structure diagram ensures that, for each single relationship, the transformation will result in



(a)

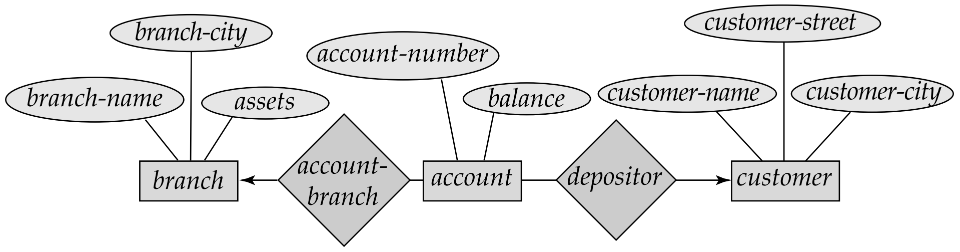


(b)

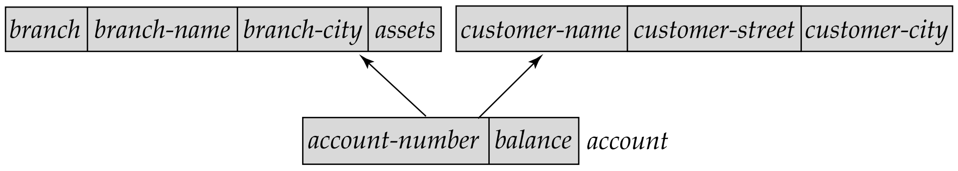
Figure B.13 Sample database corresponding to diagram of Figure B.12b.

diagrams that are of the form of rooted trees. Unfortunately, application of such a transformation individually to each relationship in an E-R diagram does not necessarily result in diagrams that are rooted trees.

Next, we shall discuss means for resolving the problem. The technique is to split the diagrams in question into several diagrams, each of which is a rooted tree. We present here two examples to illustrate the overall strategy that you can apply to



(a) E-R diagram



(b) transformation of E-R diagram

Figure B.14 E-R diagram and its transformation.

deal with such transformations. (The large number of different possibilities would make it cumbersome to present a general transformation algorithm.)

Consider the E-R diagram of Figure B.14a. By applying the transformation algorithm in Section B.2.1 separately to the relationships *account-branch* and *depositor*, we obtain the diagram of Figure B.14b. This diagram is not a rooted tree, since the only possible root can be the record type *account*, but this record type has many-to-one relationships with both its children, and that violates our definition of a rooted tree (see Section B.2). To transform this diagram into one that is in the form of a rooted tree, we replicate the *account* record type, and create two separate trees, as in Figure B.15. Note that each such tree is indeed a rooted tree. Thus, in general, we can split such a diagram into several diagrams, each of which is a rooted tree.

Now consider the E-R diagram of Figure B.16a. By applying the transformation algorithm described in Section B.2.1, we obtain the diagram in Figure B.16b. This diagram is not in the form of a rooted tree, since it contains a cycle. To transform the diagram to a tree-structure diagram, we replicate all three record types, and create

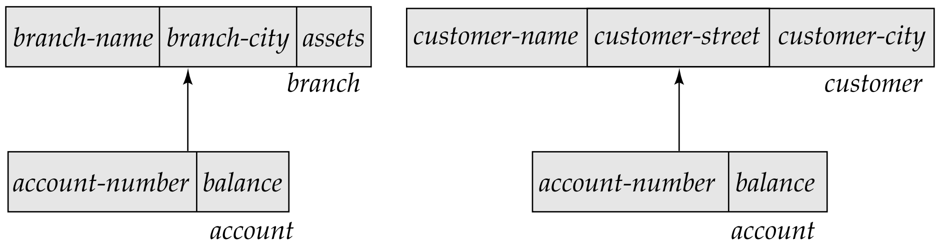


Figure B.15 Tree-structure diagram corresponding to Figure B.14a.

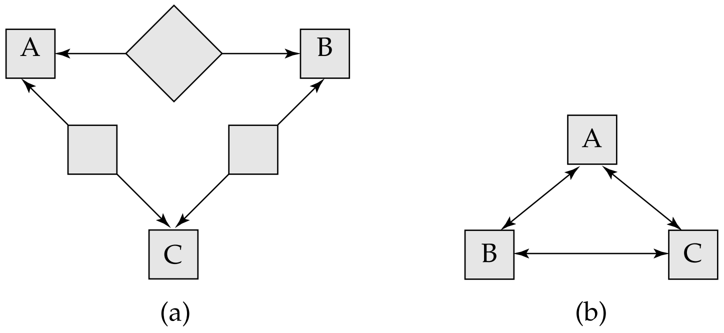


Figure B.16 E-R diagram and its transformation.

two separate diagrams, as in Figure B.17. Note that each such diagram is indeed a rooted tree. Thus, in general, we can split such a diagram into several diagrams, each of which is a rooted tree.

B.3 Data-Retrieval Facility

In this section, we present a query language for hierarchical databases that is derived from DL/I, the data-manipulation language of IMS. To simplify the presentation, we shall deviate from the DL/I syntax, and shall use a simplified notation. Our language consists of commands that are embedded in a host language, Pascal. We shall use a simple example of a *customer-account-branch* schema. The tree-structure diagram corresponding to this schema appears in Figure B.18. It specifies that a branch can have several customers, each of which can have several accounts. An account, however, may belong to only one customer, and a customer can belong to only one branch. An instance corresponding to this schema appears in Figure B.19.

B.3.1 Program Work Area

Each application program executing in the system consists of a sequence of statements. Some of these statements are in Pascal; others are data-manipulation-language

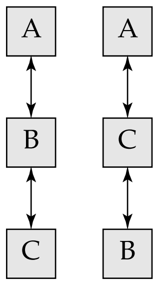


Figure B.17 Tree-structure diagram corresponding to Figure B.16a.

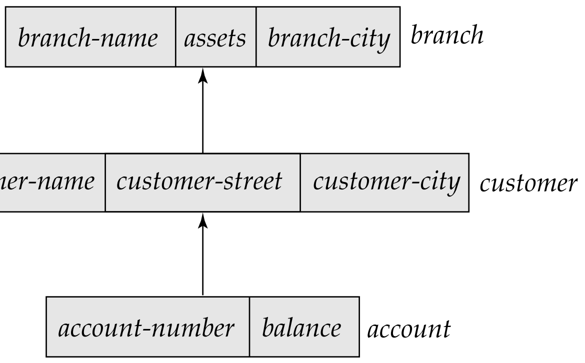


Figure B.18 Tree-structure diagram.

command statements. These statements access and manipulate database items, as well as locally declared variables. For each such application program, the system maintains a *program work area*, which is a buffer storage area that contains the following variables:

- **Record templates.** A record (in the Pascal sense) for each record type accessed by the application program
- **Currency pointers.** A set of pointers, one for each database tree, containing the *address* of the record in that particular tree (regardless of type) most recently accessed by the application program

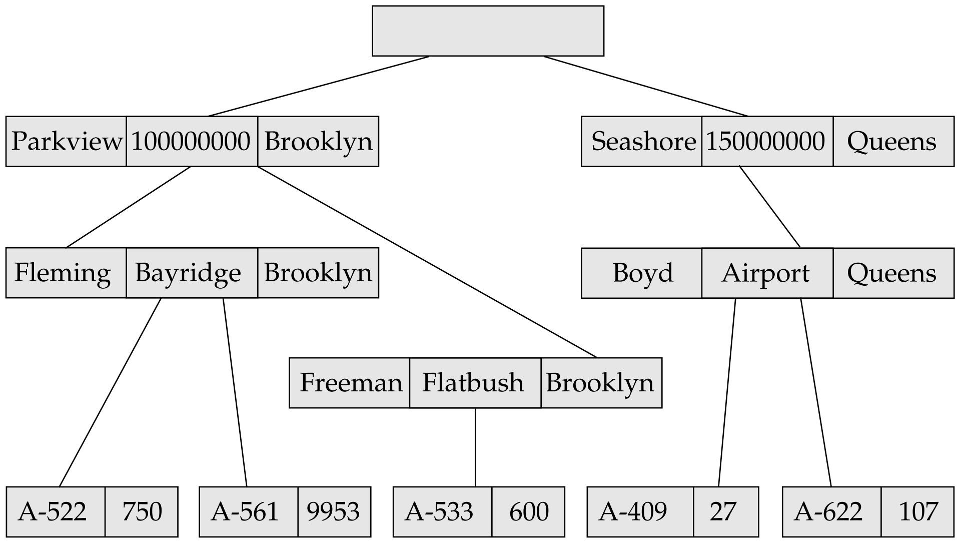

Parkview 100000000 Brooklyn
Seashore 150000000 Queens
Fleming Bayridge Brooklyn
Boyd Airport Queens
Freeman Flatbush Brooklyn
A-522 750
A-561 9953
A-533 600
A-409 27
A-622 107

Figure B.19 Sample database corresponding to Figure B.18.

- **Status flag.** A variable set by the system to indicate to the application program the outcome of the most recent database operation; we call this flag *DB-status* and use the same convention as in the DBTG model to denote failure—namely, if *DB-status* = 0, then the most recent operation succeeded.

We reemphasize that a particular program work area is associated with precisely one application program.

For our *branch-customer-account* example, a particular program work area contains the following:

- **Templates.** One record for each of three record types:
  - ☐ *branch* record
  - ☐ *customer* record
  - ☐ *account* record
- **Currency pointer.** A pointer to the most recently accessed record of *branch*, *customer*, or *account* type
- **Status.** One status variable

B.3.2 The get Command

Data are retrieved through the **get** command. The actions taken in response to a **get** are as follows:

1. Locate a record in the database and set the currency pointer to it
2. Copy that record from the database to the appropriate program area template

The **get** command must specify which of the database trees is to be searched. For our example, we assume that the only database tree to be searched is the sample database of Figure B.19; thus, we omit this specification in our queries.

As an illustration of the general effect that the **get** command has on the program work area, consider the sample database of Figure B.19. Suppose that a **get** command is issued to locate the *customer* record belonging to Freeman. Once this command executes successfully, these changes occur in the state of the program work area:

- The currency pointer points now to the record of Freeman.
- The information pertaining to Freeman is copied into the *customer* record work-area template.
- *DB-status* is set to the value 0.

To scan all records in a consistent manner, we must impose an ordering on the records. The one commonly used is *preorder*. A preorder search starts at the root, and then searches the subtrees of the root from left to right, recursively. Thus, we start at the root, visit the leftmost child, visit its leftmost child, and so on, until we reach a leaf (childless) node. We then move back to the parent of the leaf and visit the leftmost unvisited child. We proceed in this manner until we have visited the entire tree. For example, the preordered listing of the records in the database tree of Figure B.19 is:

```
Parkview, Fleming, A-522, A-561, Freeman, A-533,
Seashore, Boyd, A-409, A-622
```

B.3.3 Access within a Database Tree

There are two different **get** commands for locating records in a database tree. The simplest command has the form

```
get first <record type>
    where <condition>
```

The **where** clause is optional. The attached <condition> is a predicate that may involve any record type that is either an ancestor of <record type> or the <record type> itself.

The **get** command locates the first record (in preorder) of type <record type> in the database that satisfies the <condition> of the **where** clause. If the **where** clause is omitted, then the command locates the first record of type <record-type>. Once such a record is found, the currency pointer is set to point to that record, and the content of the record is copied into the appropriate work-area template. If no such record exists in the database tree, then the search fails, and the variable *DB-status* is set to an appropriate error message.

As an illustration, we construct the database query that prints the address of customer Fleming:

```
get first customer
    where customer.customer-name = "Fleming";
print (customer.customer-address);
```

As another example, consider the query that prints an account belonging to Fleming that has a balance greater than $10,000 (if one such exists).

```
get first account
    where customer.customer-name = "Fleming" and account.balance > 10000;
if DB-status = 0 then print (account.account-number);
```

There may be several similar records in the database that we wish to retrieve. The **get first** command locates one of these. To locate the other database records, we can use the following command:

```
get next <record type>
    where <condition>
```

This command locates the next record (in preorder) that satisfies <condition>. If the **where** clause is omitted, then the command locates the next record of type <record

type>. Note that the system uses the currency pointer to determine where to resume the search. As before, the currency pointer, the work-area template of type <record-type>, and *DB-status* are affected.

As an illustration, we construct the database query that prints the account number of all the accounts that have a balance greater than $500.

```
get first account
      where account.balance > 500;
while DB-status = 0 do
     begin
         print (account.account-number);
         get next account
               where account.balance > 500;
     end
```

We have enclosed part of the query in a **while** loop, since we do not know in advance how many such accounts exist. We exit from the loop when *DB-status* $\neq 0$. This value indicates that the last **get next** operation failed, implying that we have exhausted all account records with *account.balance* > 500.

The two previous **get** commands locate a database record of type <record type> within a particular database tree. There are, however, many circumstances in which we wish to locate such a record within a particular subtree. That is, we want to limit the search to one specific subtree, rather than search the entire database tree. The root of the subtree in question is the most recent record that was located with either a **get first** or **get next** command. This record is known as the *current parent*. There is only one current parent record per database tree. The **get** command to locate a record within the subtree rooted at the current parent has the form

```
get next within parent <record type>
      where <condition>
```

It locates the next record (in preorder) of type <record type> that satisfies <condition> and is in the subtree rooted at the current parent. If the **where** clause is omitted, then the command locates the next record of type <record type> within the designated subtree. The system uses the currency pointer to determine where to resume the search. As before, the currency pointer and the work-area template of type <record type> are affected. In this case, however, the *DB-status* is set to a nonzero value if no such record exists in the designated subtree, rather than if none exists in the entire tree. Note that a **get next within parent** command will not modify the pointer to the current parent.

To illustrate how this **get** command executes, we shall construct the query that prints the total balance of all accounts belonging to Boyd:

```
sum := 0;
get first customer
    where customer.customer-name = "Boyd";
get next within parent account;
while DB-status = 0 do
    begin
        sum := sum + account.balance;
        get next within parent account;
    end
print (sum);
```

Note that we exit from the **while** loop and print out the value of *sum* only when the *DB-status* is set to a value not equal to 0. Such a value exists after the **get next within parent** operation fails, indicating that we have exhausted all the accounts whose owner is customer Boyd.

B.4 Update Facility

Section B.3 described commands for querying the database. In this section, we describe the mechanisms available for updating information in the database. They allow insertion and deletion of records, as well as modification of the content of existing records.

B.4.1 Creation of New Records

To insert a record of type <record type> into the database, we must first set the appropriate values in the corresponding <record type> work-area template. Once we set them, we add the new record to the database tree by executing

```
insert <record type>
    where <condition>
```

If the **where** clause is included, the system searches the database tree (in preorder) for a record that satisfies the <condition> in the **where** clause. Once it finds such a record—say, *X*—it inserts the newly created record into the tree as the leftmost child of *X*. If the **where** clause is omitted, the system inserts the record in the first position (in preorder) in the database tree where a record type <record type> can be inserted in accordance with the schema specified by the corresponding tree-structure diagram.

Consider the program for adding a new customer, Jackson, to the Seashore branch:

```
customer.customer-name := "Jackson";
customer.customer-street := "Old Road";
customer.customer-city := "Queens";
insert customer
    where branch.branch-name = "Seashore";
```

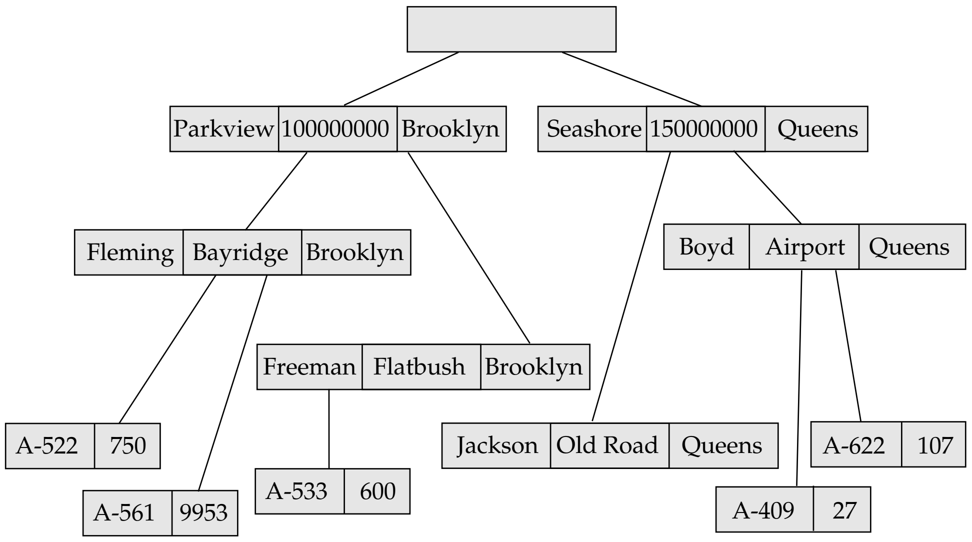

Parkview | 100000000 | Brooklyn
Seashore | 150000000 | Queens
Fleming | Bayridge | Brooklyn
Boyd | Airport | Queens
Freeman | Flatbush | Brooklyn
A-522 | 750
Jackson | Old Road | Queens
A-622 | 107
A-533 | 600
A-561 | 9953
A-409 | 27

Figure B.20 New database tree.

The result of executing this program is the database tree of Figure B.20.

As another example, consider the program for creating a new account numbered A-655 that belongs to customer "Jackson":

```
account.account-number := "A-655";
account.balance := 100;
insert account
   where customer.customer-name = "Jackson";
```

The result of executing this program is the database tree of Figure B.21.

B.4.2 Modification of an Existing Record

To modify an existing record of type <record type>, we must get that record into the work-area template for <record type>, and change the desired fields in that template. Then, we reflect the changes in the database by executing

replace

Note that the **replace** command does not have <record type> as an argument. The record that is affected is the one to which the currency pointer points, which must be the desired record.

The DL/I language requires that, before a record can be modified, the **get** command must have the additional clause **hold**, so that the system is aware that a record is to be modified.

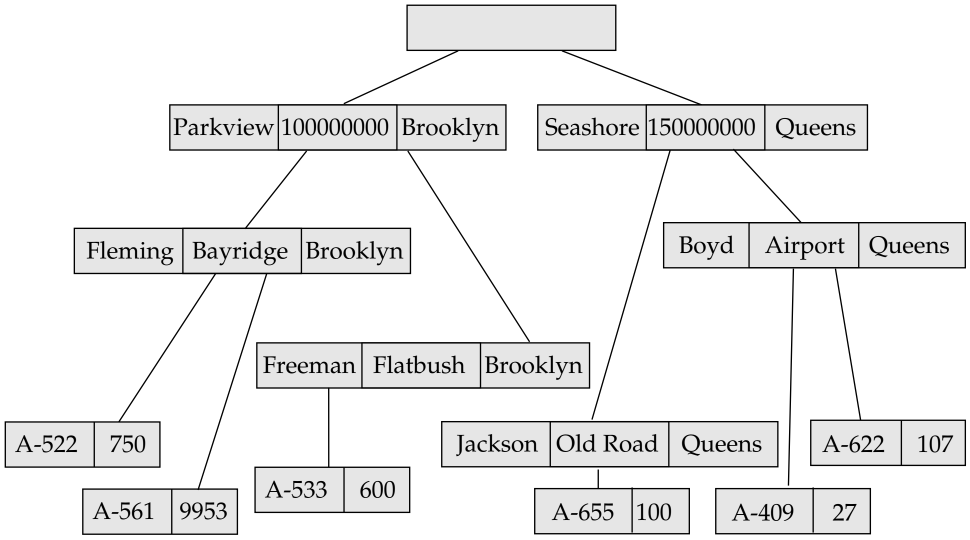

Parkview 100000000 Brooklyn
Seashore 150000000 Queens
Fleming Bayridge Brooklyn
Boyd Airport Queens
Freeman Flatbush Brooklyn
A-522 750
Jackson Old Road Queens
A-622 107
A-561 9953
A-533 600
A-655 100
A-409 27

Figure B.21 New database tree.

As an example, consider the program to change the street address of Boyd to Northview:

```
get hold first customer
    where customer.customer-name = "Boyd";
customer.customer-street := "Northview";
replace;
```

Note that, in our example, we have only one record containing the address of Boyd. If that were not the case, our program would have included a loop to search all Boyd records.

B.4.3 Deletion of a Record

To delete a record of type <record type>, we must set the currency pointer to point to that record. Then, we can delete that record by executing

delete

Note that, as in record modification, the **get** command must have the attribute **hold** attached to it.

As an illustration, consider the program to delete account A-561:

```
get hold first account
    where account.account-number = "A-561";
delete;
```

A **delete** operation deletes not only the record in question, but also the entire subtree rooted by that record. Thus, to delete customer Boyd and all his accounts, we write

```
get hold first customer
    where customer.customer-name = "Boyd";
delete;
```

B.5 Virtual Records

We have seen that, in the case of many-to-many relationships, record replication is necessary to preserve the tree-structure organization of the database. Record replication has two major drawbacks:

1. Data inconsistency may result when updating takes place.
2. Waste of space is unavoidable.

There are several ways to eliminate these drawbacks.

To eliminate record replication, we need to relax our requirement that the logical organization of data be constrained to a tree structure. We need to do that cautiously, however, since otherwise we will end up with the network model.

The solution is to introduce the concept of a *virtual record*. Such a record contains no data value; it does contain a logical pointer to a particular physical record. Instead of replication, we keep a single copy of the physical record, and everywhere else we keep virtual records containing a pointer to that physical record.

More specifically, we let *R* be a record type that is replicated in several tree-structure diagrams—say, $T_1, T_2, \cdots, T_n$. To eliminate replication, we create a new virtual record type *virtual-R*, and replace *R* in each of the $n - 1$ trees with a record of type *virtual-R*.

As an example, consider the E-R diagram of Figure B.6a and its corresponding tree-structure diagram, which comprises two separate trees, each consisting of both *customer* and *account* record types (Figure B.6b).

To eliminate data replication, we create two virtual record types: *virtual-customer* and *virtual-account*. We then replace record type *account* with record type *virtual-account* in the first tree, and replace record type *customer* with record type *virtual-customer* in the second tree. We also add a dashed line from *virtual-customer* record to *customer* record, and a dashed line from *virtual-account* record to *account* record,

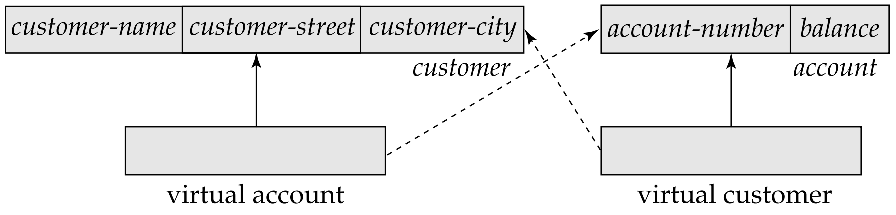


Figure B.22 Tree-structure diagram with virtual records.

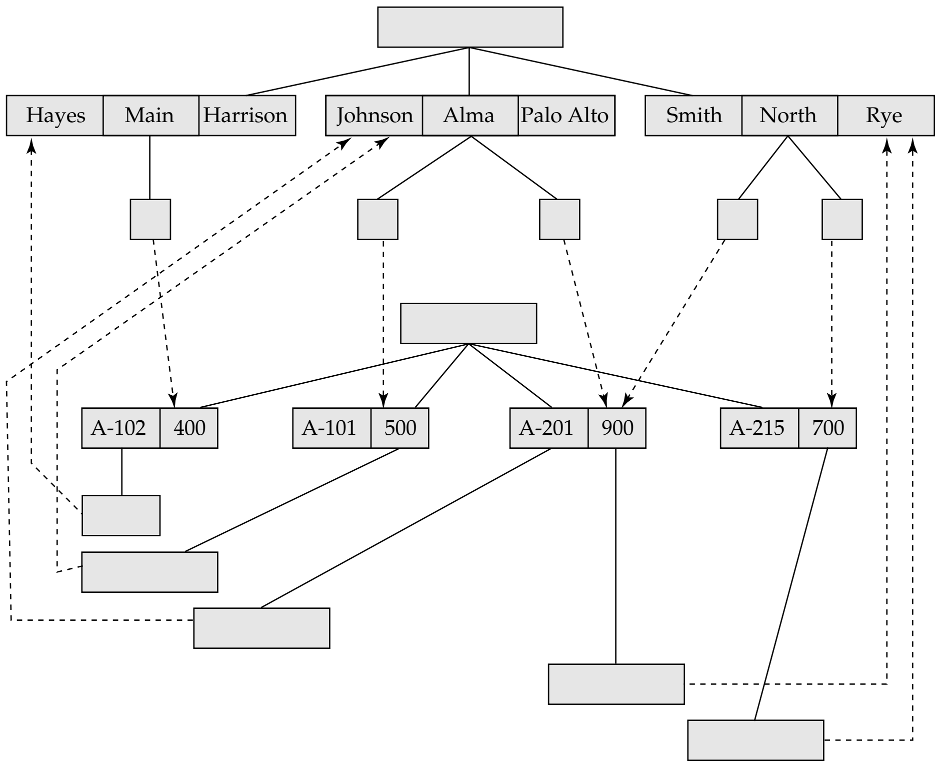


Figure B.23 Sample database corresponding to diagram of Figure B.22.

to specify the association between a virtual record and its corresponding physical record. The resulting tree-structure diagram appears in Figure B.22.

A sample database corresponding to the diagram of Figure B.22 appears in Figure B.23. Note that only a single copy of the information for each customer and each account exists. Contrast this database with the same information depicted in Figure B.7, where replication is allowed.

The data-manipulation language for this new configuration remains the same as in the case where record replication is allowed. Thus, a user does not need to be aware of these changes. Only the internal implementation is affected.

B.6 Mapping of Hierarchies to Files

A straightforward technique for implementing the instance of a tree-structure diagram is to associate one pointer with a record for each child that the record has. Consider the database tree of Figure B.1. Figure B.24 shows an implementation of this database using parent-to-child pointers. Parent–child pointers, however, are not an ideal structure for the implementation of hierarchical databases, since a parent record may have an arbitrary number of children. Thus, fixed-length records become variable-length records once the parent–child pointers are added.

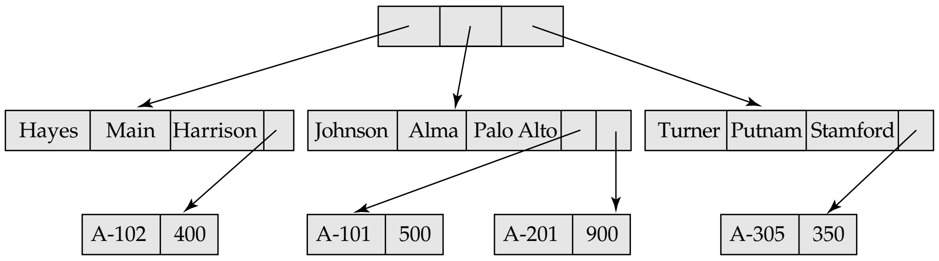


Figure B.24 Implementation with parent-child pointers.

Instead of parent–child pointers, we can use *leftmost-child* and *next-sibling* pointers. Figure B.25 shows this structure for the database tree of Figure B.1. Under this structure, every record has exactly two pointer fields. Thus, fixed-length records retain their fixed length when we add the necessary pointers. Note that the leftmost-child pointers for the *account* record are null, since *account* is a leaf of the tree.

Observe that several pointer fields are unused in Figure B.25. In general, the final child of a parent has no next sibling; thus, its next-sibling field is set to null. Rather than place nulls in such fields, we can place pointers there to facilitate the preorder traversal required to process queries on hierarchical databases. Thus, for each record that is a rightmost sibling, we place a pointer in the next-sibling field to the next record in preorder after traversing its subtree. Figure B.26 shows this modification to the structure of Figure B.25. These pointers allow us to process a tree instance in preorder simply by following pointers. For this reason, the pointers are sometimes referred to as *preorder threads*.

A parent pointer is often added to records in an implementation of a hierarchical database. This pointer facilitates the processing of queries that give a value for a child record and request a value from the corresponding parent record. If we include parent pointers, a total of exactly three pointer fields is added to each record.

To see how best to locate records of a hierarchical database physically on disk, we draw an analogy between the parent–child relationship within a hierarchy and the owner–member relationship within a DBTG set. In both cases, a one-to-many relationship is being represented. We want to store together the members and the owners of a set occurrence. Similarly, we want to store physically close on disk the child records

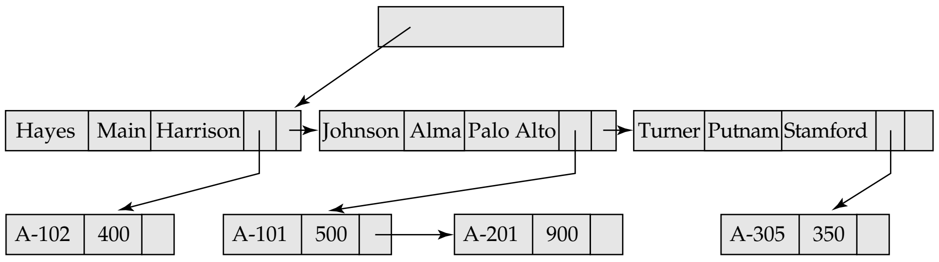


Figure B.25 Implementation with leftmost-child and next-sibling pointers.

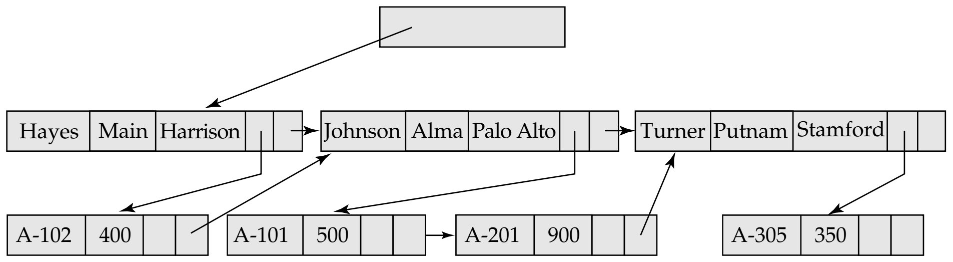


Figure B.26 Implementation using preorder threads.

and their parent. This form of storage allows a sequence of **get first**, **get next**, and **get next within parent** statements to be executed with a minimal number of block accesses.

B.7 The IMS Database System

The hierarchical model is significant primarily because of the importance of IBM's IMS database system.

The IBM Information Management System (IMS) is one of the oldest and most widely used database systems. Since IMS databases have historically been among the largest, the IMS developers were among the first to deal with such issues as concurrency, recovery, integrity, and efficient query processing. Through several releases, IMS acquired a large number of features and options. As a result, IMS is a highly complex system. We shall consider only a few features of IMS here.

Queries on IMS databases are issued through embedded calls in a host language. The embedded calls are part of the IMS database language DL/I. (The language used in this appendix is a simplified form of DL/I.)

Since performance is critically important in large databases, IMS allows the database designer a broad number of options in the data-definition language. The database designer defines a physical hierarchy as the database schema. She can define several subschemas (or views) by constructing a logical hierarchy from the record types constituting the schema. The various options available in the data-definition language (block sizes, special pointer fields, and so on) allow the database administrator to tune the system for improved performance.

Several record access schemes are available in IMS:

- The hierarchical sequential-access method (HSAM) is used for physically sequential files (such as tape files). Records are stored physically in preorder.

- The hierarchical indexed-sequential-access method (HISAM) is an index-sequential organization at the root level of the hierarchy. Records are stored physically in preorder.

- The hierarchical indexed-direct-access method (HIDAM) is an ordered index organization at the root level with pointers to child records.

- The hierarchical direct-access method (HDAM) is similar to HIDAM, but with hashed access at the root level.

The original version of IMS predated the development of concurrency-control theory. Early versions of IMS had a simple form of concurrency control. Only one update application program could run at a time. However, any number of read-only applications could run concurrently with an update application. This feature permitted applications to read uncommitted updates and allowed nonserializable executions. Exclusive access to the database was the only option available to applications that demanded a greater degree of isolation from the anomalies of concurrent processing.

Later versions of IMS included a more sophisticated *program-isolation feature* that allowed for both improved concurrency control and more sophisticated transaction-recovery techniques (such as logging). These features increased in importance as more IMS users began to use online transactions, as opposed to the batch transactions that were originally the norm.

The need for high-performance transaction processing led to the introduction of *IMS Fast Path*. Fast Path uses an alternative physical data organization designed to allow the most active parts of the database to reside in main memory. Instead of forcing updates to disk at the end of a transaction (as standard IMS does), Fast Path defers update until a checkpoint or synchronization point. In the event of a crash, the recovery subsystem must redo all committed transactions whose updates were not forced to disk. These tricks and others allow for extremely high rates of transaction throughput. IMS Fast Path is a forerunner of much of the work on developing main-memory database systems that has emerged as main memory has become larger and less expensive.

B.8 Summary

A hierarchical database consists of a collection of *records* that are connected to each other through *links*. A record is a collection of fields, each of which contains only one data value. A link is an association between precisely two records. The hierarchical model is thus similar to the network model in the sense that data and relationships between data are also represented by records and links, respectively. The hierarchical model differs from the network model in that the record types are organized as collections of trees, rather than as arbitrary graphs.

A *tree-structure diagram* is a schema for a hierarchical database. Such a diagram consists of two basic components: boxes, which correspond to record types, and lines, which correspond to links. A tree-structure diagram serves the same purpose as an E-R diagram; it specifies the overall logical structure of the database. A tree-structure diagram is similar to a data-structure diagram in the network model. The main difference is that, in the former, record types are organized in the form of an arbitrary graph, whereas in the latter, record types are organized in the form of a *rooted tree*. For every E-R diagram, there is a corresponding tree-structure diagram.

The database schema is thus represented as a collection of tree-structure diagrams. For each such diagram, there exists a *single* instance of a database tree. The root of this tree is a dummy node. The children of the dummy node are instances of the root record type in the tree structure diagram. Each record instance may, in turn, have several children, which are instances of various record types, as specified in the corresponding tree-structure diagram.

The data-manipulation language discussed in this appendix consists of commands that are embedded in a host language. These commands access and manipulate database items, as well as locally declared variables. For each application program, the system maintains a *program work area* that contains *record templates*, *currency pointers*, and a *status flag*.

Data items are retrieved through the **get** command, which locates a record in the database, sets the currency pointer to point to that record, and then copies the record from the database to the appropriate program work-area template. There are various forms of the **get** command. The main distinction among them is where in the database tree the search starts and whether the search continues until the end of the entire database tree or restricts itself to a particular subtree.

Various mechanisms are available for updating information in the database. They allow the creation and deletion of records (via the **insert** and **delete** operations), and the modification (via the **replace** operation) of the content of existing records.

In the case of many-to-many relationships, record replication is necessary to preserve the tree-structure organization of the database. Record replication has two major drawbacks: (1) data inconsistency may result when updating takes place and (2) waste of space is unavoidable. The solution is to introduce the concept of a *virtual record*. Such a record contains no data value; it does contains a logical pointer to a particular physical record. When a record needs to be replicated, a single copy of the actual record is retained, and all other records are replaced with a virtual record containing a pointer to that physical record. The data-manipulation language for this new configuration remains the same as in the case where record replication is allowed. Thus, a user does not need to be aware of these changes. Only the internal implementation is affected.

Implementations of hierarchical databases do not use parent-to-child pointers, since that would require the use of variable-length records. Instead, they use preorder threads. This technique allows each record to contain exactly two pointers. Optionally, a third child-to-parent pointer may be added.

Exercises

B.1 Transform the E-R diagram of Figure B.27 to a tree-structure diagram.

B.2 Construct a sample database for the tree-structure diagrams of Exercise B.1, with three students and three different classes.

B.3 Show the preorder order of the sample database of Exercise B.2.

B.4 Show the set of variables in a program work area for the tree-structure diagram corresponding to the E-R diagram of Figure B.27.

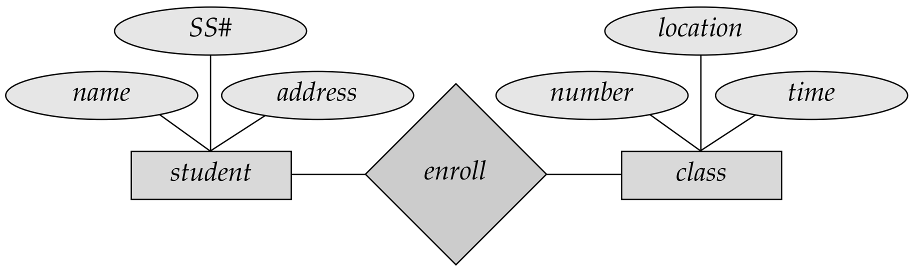


Figure B.27 Class-enrollment E-R diagram.

B.5 Suppose that we add the attribute "grade" to the relationship *enroll* of Figure B.27. Show the corresponding tree-structure diagram.

B.6 Transform the E-R diagram of Figure B.28 into a tree-structure diagram.

B.7 Compare the hierarchical model with the relational model in terms of ease of learning and ease of use.

B.8 Are certain applications easier to code in the hierarchical model than in the relational model? If you answer yes, give an example of one; if you answer no, explain your answer.

B.9 Transform the E-R diagram of Figure B.29 into a tree-structure diagram.

B.10 For the tree-structure diagram corresponding to the E-R diagram of Figure B.29, construct the following queries:

a. Find the total number of people whose car was involved in an accident in 1993.
b. Find the total number of accidents in which the cars belonging to "John Smith" were involved.
c. Add a new customer to the database.
d. Delete the car "Mazda" belonging to "John Smith."
e. Add a new accident record for the Toyota belonging to "Jones."

B.11 The addition of virtual records to the hierarchical model results in a structure that is no longer tree-like. In effect, the underlying structure is quite similar to the network model. What are the differences between the hierarchical model with virtual records and the network model?

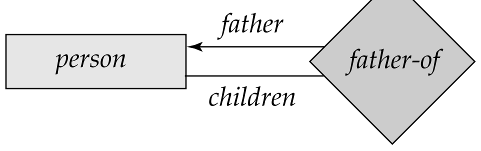


Figure B.28 Parent–child E-R diagram.

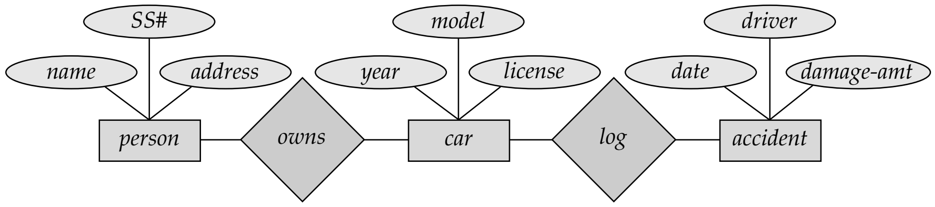


Figure B.29 Car-insurance E-R diagram.

B.12 Give an appropriate tree-structure diagram for the following relational database:

employee (*person-name*, *street*, *city*)
works (*person-name*, *company-name*, *salary*)
company (*company-name*, *city*)
manages (*person-name*, *manager-name*)

B.13 Consider the database schema corresponding to the tree-structure diagram that you obtained as a solution to Exercise B.12. For each of the following queries, construct the appropriate program:

a. Find the names of all employees who work for First Bank Corporation.

b. Find the names and cities of residence of all employees who work for First Bank Corporation.

c. Find the names, streets, and cities of residence of all employees who work for First Bank Corporation and earn more than $10,000.

d. Find all employees who live in the city where the company for which they work is located.

e. Find all employees who live in the same city and on the same street as their managers.

f. Find all employees in the database who do not work for First Bank Corporation.

g. Find all employees in the database who earn more than every employee of Small Bank Corporation.

h. Assume that the companies can be located in several cities. Find all companies located in every city in which Small Bank Corporation is located.

i. Find all employees who earn more than the average salary of employees who work in their company.

j. Find the company that employs the most people.

k. Find the company that has the smallest payroll.

l. Find those companies that pay higher salaries, on average, than the average salary at First Bank Corporation.

m. Modify the database such that Jones now lives in Newtown.

n. Give all employees of First Bank Corporation a 10 percent raise.

o. Give all managers in the database a 10 percent raise.

p. Give all managers in the database a 10 percent raise, unless the resulting salary would be greater than \$100,000; if it would be, give only a 3 percent raise.

q. Delete all employees of Small Bank Corporation.

B.14 Give a tree-structure diagram for the following relational database:

course (*course-name, room, instructor*)
enrollment (*course-name, student-name, grade*)

Also give an example implementation of an instance of this database.

Bibliographical Notes

Two influential database systems that rely on the hierarchical model are IBM's Information Management System (IMS) [IBM 1978a, McGee 1977] and MRI's System 2000 [MRI 1974, 1979]. The first IMS version was developed in the late 1960s by IBM and by North American Aviation (Rockwell International) for the Apollo moon-landing program.

A survey paper on the hierarchical data model is presented by Tsichritzis and Lochovsky [1976]. The simplified version of DL/I used in this appendix is similar to the one presented by Ullman [1988]. With the current dominance of relational database systems, there is often a need to query data in legacy hierarchical databases by using a relational language. Meng et al. [1995] discusses translation of relational queries into hierarchical queries.

Obermarck [1980] discusses the IMS program-isolation feature and gives a brief history of the concurrency-control component of IMS. Bjorner and Lovengren [1982] presents a formal definition of IMS.