Chapter B: Hierarchical Model

- Basic Concepts
- Tree-Structure Diagrams
- Data-Retrieval Facility
- Update Facility
- Virtual Records
- Mapping of Hierarchies to Files
- The IMS Database System

Basic Concepts

- A hierarchical database consists of a collection of records which are connected to one another 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 differs from the network model in that the records are organized as collections of trees rather than as arbitrary graphs.
Tree-Structure Diagrams

- The schema for a hierarchical database consists of
  - boxes, which correspond to record types
  - lines, which correspond to links
- Record types are organized in the form of a rooted tree.
  - No cycles in the underlying graph.
  - Relationships formed in the graph must be such that only
    one-to-many or one-to-one relationships exist between a parent and
    a child.

General Structure

- A parent *may* have an arrow pointing to a child, but a child *must*
  have an arrow pointing to its parent.
**Tree-Structure Diagrams (Cont.)**

- Database schema is represented as a collection of tree-structure diagrams.
  - *single* instance of a database tree
  - The root of this tree is a dummy node
  - The children of that node are actual instances of the appropriate record type
- When transforming E-R diagrams to corresponding tree-structure diagrams, we must ensure that the resulting diagrams are in the form of rooted trees.

**Single Relationships**

![Diagram of E-R and tree-structure diagrams]

(a) E-R diagram

(b) Tree-structure diagram
Single relationships (Cont.)

- Example E-R diagram with two entity sets, `customer` and `account`, related through a binary, one-to-many relationship `depositor`.

- Corresponding tree-structure diagram has:
  - the record type `customer` with three fields: `customer-name`, `customer-street`, and `customer-city`.
  - the record type `account` with two fields: `account-number` and `balance`.
  - the link `depositor`, with an arrow pointing to `customer`.

- If the relationship `depositor` is one to one, then the link `depositor` has two arrows.

- Only one-to-many and one-to-one relationships can be directly represented in the hierarchical mode.
Transforming Many-To-Many Relationships

- Must consider the type of queries expected and the degree to which the database schema fits the given E-R diagram.

- In all versions of this transformation, the underlying database tree (or trees) will have replicated records.

Many-To Many Relationships (Cont.)

(a) E-R diagram

(b) Tree-structure diagrams
Many-To-Many Relationships (Cont.)

- Create two tree-structure diagrams, $T_1$, with the root customer, and $T_2$, with the root account.

- In $T_1$, create depositor, a many-to-one link from account to customer.

- In $T_2$, create account-customer, a many-to-one link from customer to account.

Sample Database
General Relationships

Example ternary E-R diagram and corresponding tree-structure diagrams are shown on the following page.

Sample Ternary Databases. (a) T₁ (b) T₂
Several Relationships

To correctly transform an E-R diagram with several relationships, split the unrooted tree structure diagrams into several diagrams, each of which is a rooted tree.

Example E-R diagram and transformation leading to diagram that is not a rooted tree:

Several Relationships (Cont.)

(a) E-R diagram

(b) transformation of E-R diagram
Several Relationships (Cont.)

- Corresponding diagrams in the form of rooted trees.

```
branch-name | branch-city | assets
branch
account-number | balance
account
```

```
customer-name | customer-street | customer-city
customer
account-number | balance
account
```

Several Relationships (2nd Example)

- Diagram (b) contains a cycle.
- Replicate all three record types, and create two separate diagrams.
Several Relationships (2nd Example)

- Each diagram is now a rooted tree.

Data Retrieval Facility

- We present querying of hierarchical databases via a simplified version of DL/I, the data-manipulation language of IMS.
- Example schema: `customer-account-branch`
- A branch can have several customers, each of which can have several accounts.
- An account may belong to only one customer, and a customer can belong to only one branch.
**Program Work Area**

- A buffer storage area that contains these variables
  - Record templates
  - Currency pointers
  - Status flag

- A particular program work area is associated with precisely one application program.

- Example program work area:
  - Templates for three record types: **customer**, **account**, and **branch**.
  - Currency pointer to the most recently accessed record of **branch**, **customer**, or **account** type.
  - One status variable.
### The get Command

- Data items are retrieved through the **get** command
  - locates a record in the database and sets the currency pointer to point to it
  - copies that record from the database to the appropriate program work-area template
- The **get** command must specify which of the database trees is to be searched.
- State of the program work area after executing **get** command to locate the *customer* record belonging to Freeman
  - 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.

---

### The get Command (Cont.)

- To scan all records in a consistent manner, we must impose an ordering on the records.
- **Preorder** search starts at the root, and then searches the subtrees of the root from left to right, recursively.
  - Starts at the root, visits the leftmost child, visits its leftmost child, and so on, until a leaf (childless) node is reached.
  - Move back to the parent of the leaf and visit the leftmost unvisited child.
  - Proceed in this manner until the entire tree is visited.
- Preordered listing of the records in the example database three:
Access Within A Database Tree

- Locates the first record (in preorder), of type `<record type>` that satisfies the `<condition>` of the `where` clause.
- The `where` clause is optional `<condition>` is a predicate that involves either an ancestor of `<record type>` or the `<record type>` itself.
- If `where` is omitted, locate the first record of type `<record-type>`
  - Set currency pointer to that record
  - Copy its contents into the appropriate work-area template.
- If no such record exists in the tree, then the search fails, and `DB-status` is set to an appropriate error message.

Example Queries

- Print the address of customer Fleming:
  ```
  get first customer
  where customer.customer-name = “Fleming”;
  print (customer.customer-address);
  ```
- Print an account belonging to Fleming that has a balance greater than $10,000.
  ```
  get first account
  where customer.customer-name = “Fleming”;
  and account.balance > 10000;
  if DB-status = 0 then print (account.account-number);
  ```
Access Within a Database Tree (Cont.)

get next <record type>
   where <condition>

- Locates the next record (in preorder) that satisfies <condition>.
- If the where clause is omitted, then the next record of type <record type> is located.
- The currency pointer is used by the system 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.

Example Query

- Print 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

- When while loop returns DB-status ≠ 0, we exhausted all account records with account.balance > 500.
**Access Within a Database Tree (Cont.)**

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

- Searches only the specific subtree whose root is the most recent record that was located with either `get first` or `get next`.
- Locates the next record (in preorder) that satisfies `<condition>` in the subtree whose root is the parent of current of `<record type>`.
- If the `where` clause is omitted, then the next record of type `<record type>` within the designated subtree to resume search.
- Use currency pointer to determine where to resume search.
- `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).

**Example Query**

- Print 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);
  
  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. This value exists after the `get next within parent` operation fails.
  ```
Update Facility

- Various mechanisms are available for updating information in the database.
- Creation and deletion of records (via the `insert` and `delete` operations).
- Modification (via the `replace` operation) of the content of existing records.

Creation of New Records

- To insert `<record type>` into the database, first set the appropriate values in the corresponding `<record type>` work-area template. Then execute
  
  ```
  insert <record type>
  where <condition>
  ```
  
- If the `where` clause is included, the system searches the database three (in preorder) for a record that satisfies the `<condition>` in the `where` clause.
- Once such a record — say, $X$ — is found, the newly created record is inserted in the tree as the leftmost child of $X$.
- If `where` is omitted, the record is inserted in the first position (in preorder) in the tree where `<record type>` can be inserted in accordance with the specified schema.
**Example Queries**

- Add a new customer, Jackson, to the Seashore branch:
  
  ```sql
  customer.customer-name := “Jackson”;
customer.customer-street := “Old Road”;
customer.customer-city := “Queens”;
insert customer
  where branch.branch-name = “Seashore”;
```

- Create a new account numbered A-655 that belongs to customer “Jackson”:
  
  ```sql
  account.account-number := “A-655”;
account.balance := 100;
insert account
  where customer.customer-name = “Jackson”;
```

**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.

- Reflect the changes in the database by executing `replace`.

- `replace` does not have `<record type>` as an argument; the record that is affected is the one to which the currency pointer points.

- DL/I requires that, prior to a record being modified, the `get` command must have the additional clause `hold`, so that the system is aware that a record is to be modified.
Example Query

- Change the street address of Boyd to Northview:
  
  ```
  get hold first customer
  where customer.customer-name = "Boyd";
  customer.customer-street := "Northview";
  replace;
  ```

- If there were more than one record containing Boyd's address, the program would have included a loop to search all Boyd records.

Deletion of a Record

- To delete a record of type `<record type>`, set the currency pointer to point to that record and execute `delete`.

- As a record modification, the `get` command must have the attribute `hold` attached to it. Example: 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 gold first customer
  where customer.customer-name = "Boyd";
  delete;
  ```
Virtual Records

- For many-to-many relationships, record replication is necessary to preserve the tree-structure organization of the database.
  - Data inconsistency may result when updating takes place
  - Waste of space is unavoidable
- Virtual record — contains no data value, only a logical pointer to a particular physical record.
- When a record is to be replicated in several database trees, a single copy of that record is kept in one of the trees and all other records are replaced with a virtual record.
- Let $R$ be a record type that is replicated in $T_1, T_2, \ldots, T_n$. Create a new virtual record type virtual-$R$ and replace $R$ in each of the $n - 1$ trees with a record of type virtual-$R$.

Virtual Records (Cont.)

- Eliminate data replication in the diagram shown on page B.11; create virtual-customer and virtual-account.
- Replace account with virtual-account in the first tree, and replace customer with virtual-customer in the second tree.
- Add a dashed line from virtual-customer to customer, and from virtual-account to account, to specify the association between a virtual record and its corresponding physical record.
Implementations of hierarchical databases do not use parent-to-child pointers, since these would require the use of variable-length records.

Can use leftmost-child and next-sibling pointers which allow each record to contain exactly two pointers.

- The leftmost-child pointer points to one child.
- The next-sibling pointer points to another child of the same parent.
Mapping Hierarchies to Files (Cont.)

- Implementation with parent-child pointers.

- Implementation with leftmost child and next-sibling pointers.

In general, the final child of a parent has no next sibling; rather than setting the next-sibling filed to null, place a pointer (or preorder thread) that points to the next record in preorder.

Using preorder threads allows us to process a tree instance in preorder simply by following pointers.
Mapping Hierarchies to Files (Cont.)

- May add a third child-to-parent pointer which facilitates the processing of queries that give a value for a child record and request a value from the corresponding parent record.
- The parent-child relationship within a hierarchy is analogous to the owner-member relationship within a DBTG set.
  - A one-to-many relationship is being represented.
  - Store together the members and the owners of a set occurrence.
  - Store physically close on disk the child records and their parent.
  - Such 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.

The IMS Database System

- IBM Information Management System — first developed in the late 1960s; historically among the largest databases.
- Issue queries through embedded calls which are part of the IMS database language DL/I.
- Allows the database designer a broad number of options in the data-definition language.
  - Designer defines a physically hierarchy as the database schema.
  - Can define several subschemas (or view) by constructing a logical hierarchy from the record types constituting the schema.
  - Options such as block sizes, special pointer fields, and so on, allow the database administrator to tune the system.
Record Access Schemes

- Hierarchical sequential-access method (HSAM) — used for physically sequential files (such as tape files). Records are stored physically in preorder.
- Hierarchical indexed-sequential-access method (HISAM) — an index-sequential organization at the root level of the hierarchy.
- Hierarchical indexed-direct-access method (HIDAM) — index organization at the root level with pointers to child records.
- Hierarchical direct-access method (HDAM) — similar to HIDAM, but with hashed access at the root level.

IMS Concurrency Control

- Early versions handled concurrency control by permitting only one update application program to run at a time. Read-only applications could run concurrent with updates.
- Later versions included a program-isolation feature
  - Allowed for improved concurrency control
  - Offered more sophisticated transaction-recovery techniques (such as logging); important to online transactions.
- The need for high-performance transaction processing led to the introduction of IMS Fast Path.
IMS Fast Path

- Uses an alternative physical data organization that allows the most active parts of the database to reside in main memory.
- Instead of updates to disk being forced at the end of a transaction, update is deferred 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.
- Allows for extremely high rates of transaction throughput.
- Forerunner of main-memory database systems.

Sample Database
Sample Database Corresponding to Diagram of Figure B.4

Sample Database Corresponding To Diagram of Figure B.8b
Tree-Structure Diagram With Many-To-Many Relationships

E-R Diagram and Its Corresponding Tree-Structure Diagrams
Sample Database Corresponding To Diagram of Figure B.12b

New Database Tree
New Database Tree

Class-enrollment E-R Diagram
Parent–Child E-R Diagram

Car-insurance E-R Diagram