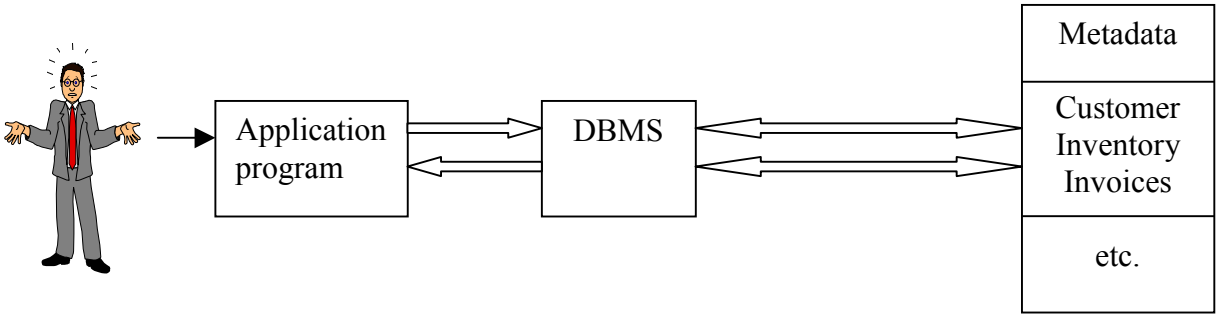
Introducing the Database

Efficient data management requires the use of a **computer database**, which is managed by a **Data Base Management System (DBMS).**

Database structure



10 pav. The DBMS Manages the Interaction Between the End User and the Database.

A DBMS is a **collection** of programs that manages the database structure and controls access to the data stored in the database. The DBMS serves as **the intermediary** between the end user and the database by translating user requests into the complex computer code. The end user interacts with the DBMS through an **application** program. The application program may be written by a programmer, using a programming language such as COBOL, or it may be created through the use of DBMS **utility programs**.

To speak of **database design**, *it means the design of the database structure that will be used to store and manage data.*

The historical roots of the database: Files and File Systems

Historically, the first computer applications focused on clerical tasks: **order/entry processing, payroll, work scheduling, and so on**.

There are **several good reasons for studying file systems**:

- File systems provide a useful historical perspective on how we handle data;
- Some of the problems of file systems may be duplicated in database software;
- The greater design complexity of database software may be easier to understand once the relatively simple file system's characteristics are understood;
- If you intend to convert an obsolete file system to a database system, knowledge of the file system's basic characteristics turns out to be useful;
- A manager of almost any small organization was (is) able to keep track of all necessary data with the help of a *manual* file system.

However, to keep track of data in the manual file system is difficult:

- by finding and using data in the growing collection of file folders;
- having to make long and detailed quarterly reports whose contents required meticulous record keeping.

The conversion from a manual file system to a matching computer file system tended to be technically complex and a new kind of professional, known as a **data processing (DP) specialist,** is emerged.

The DP specialist created the necessary computer file structures and often wrote the software that managed the data within those structures, and computerized **file systems** were born.

Basic File Terminology

Data	"Raw" facts that have little meaning unless they have been organized in some logical manner. The smallest piece of data is a single character, it requires one *byte* of computer storage
Field	A character or group of characters (alphabetic or numeric) that has a specific meaning (it may define a telephone number, a birth date, a customer name, and so on)
Record	A logically connected set of one or more fields that describes something like a person, place, or thing, and so on
File	A collection of related records

As time passed, a file success (maintained by one department) was so obvious that other departments demanded access to the DP specialist in order to automate their tasks. So a small file system evolved. Each of the files in the system used its own application programs to store, retrieve, and modify data. And each file was *owned* by the individual or the department that commissioned its creation.

As the file system grew, the demand for the DP specialist's programming skills grew even faster and additional programmers were hired. The file system's size also required the use of a larger, more complex computer. The DP specialist's job evolved into that of a DP manager. However the DP department's primary activity remained programming, and the DP manager inevitably spent much time as a supervising "super programmer" and program troubleshooter.

From a general management point of view the file system is composed of five parts:

1.	*Hardware:*	The computer
2.	*Software:*	The operating system, the utilities, the files, file management programs, and applications programs that generate reports from the data stored in the files
3.	*People:*	DP managers, DP specialists, programmers, and end users
4.	*Procedures:*	The instructions and rules that govern the design and use of the software component
5.	*Data:*	The collection of facts

A file system critique

The critique serves two major purposes:

- understanding the shortcomings of the file system enables us to understand the reasons for the development of the database;
- many of the problems of file systems are not unique to them - failure to understand such problems is likely to lead to their duplication in a database environment, even when database technology makes it relatively easy to avoid them.

File System Data Management

Even the simplest data retrieval task required extensive programming in some third-generation language (3GL). A 3GL requires the programmer both to specify *what* must be done and *how* it is to be done.

Programming in a 3GL tends to be a time-consuming, high-skill activity. Because a file is quite different from the way the computer physically stores the data on disk, the programmer must

be familiar with the physical file structure. As file systems become more complex, the access paths become tortuous and tend to precipitate system malfunctions.

The need to write 3GL programs to produce even the simplest reports makes the spur-of-the-moment questions known as ***ad hoc queries*** impossible.

As the number of files in the system expands, the system's administration becomes difficult, too. Each file must have its own file-management system, composed of programs that allow the user to:

- Create the file structure;
- Add data to the file;
- Delete data from the file;
- Modify the data contained in the file;
- List the file contents.

Even a simple file system composed of only twenty files thus requires 5 x 20 = 1OO file-management programs. And, the number of files also tends to multiply rapidly.

Planning the file structures carefully is especially important to the DP manager because making changes in an existing structure tends to be difficult in a file- system environment. For example, changing just one field in the original file requires the use of a program that

- Puts the new file structure into a **buffer;**
- Opens the original file, using a different buffer;
- Reads a record from the original file;
- Transforms the original data to conform to the new structure; using complex manipulations;
- Writes the transformed data into the new file structure;
- The original file is deleted;
- All the programs that made use of the file must be modified to fit the revised file structure.

Modifications are likely to yield errors, so additional time may be spent to find the errors (bugs) in a **debugging** process.

Because all data-access programs are subject to change when the file structure changes, the file system is said to exhibit **structural dependence.**

Security features such as effective password protection, locking out parts of files or parts of the system itself, and other measures designed to safeguard data confidentiality are difficult to implement. The file system's structure and lack of security make it difficult to pool data.

Structural and Data Dependence

- the file system exhibits structural dependence; that is, access to a file is dependent on its structure.
- all data-access programs are subject to change when any of the file's data characteristics change, the file system is said to exhibit **data dependence.**

As a consequence there exists a difference between the **data logical format** (how the human being views the data) and the **data physical format** (how the computer "sees" the data). To access a file system's file it must not only tell the computer *what* to do, but also *how* to do it. *Any change in the data characteristics or the file structures, no matter how minor, requires changes in all the programs that use a modified file.*

Field Definitions

Unique identification of records is turn out to be handy. To get a listing for records by some field turns out to be inefficient.

Data Duplication

If the file system environment makes it difficult to pool data, the same data are to be stored in different locations:

1. ***Data redundancy.*** The system contains duplicate fields in two or more files. Such duplication costs extra data-entry time and storage space. Worse, data redundancy is likely to lead to data inconsistency;
2. ***Data inconsistency*** (lack of data integrity). The change a content of field in one record and to forget make similar changes in another location yield inconsistent results. Also data entry errors are much more likely. In fact, a data-entry error such as an incorrectly spelled name or an incorrect phone number will yield the same kinds of **data integrity** problems;
3. ***Data anomalies.*** Data redundancy creates an abnormal condition by forcing field-value changes in many different locations. Data anomalies exist because *any* change in any field value must be correctly made in *many* places to maintain data consistency. Data anomalies thus can create inconsistencies by:
   - Forgetting to make changes (update and delete anomalies);
   - Data entry errors due to miskeying (615 rather than 651);
   - Data entry errors due to entry of invalid data.

Database systems

The problems in the use of computer file systems make the database system very desirable. The current generation DBMS provides the following functions:

- Stores the definitions of data relationships (metadata) in a data dictionary. In turn, all programs that access the database work through the DBMS. The DBMS uses the data in the data dictionary to look up the required data-component structures and relationships. Any changes made in a database file are automatically recorded in the data dictionary, thus freeing us from having to modify all the programs that access a changed file. The DBMS **removes structural and data dependency** from the system;
- Creates the complex structures required for data storage, thus relieving us of the difficult task of defining and programming the physical data characteristics;
- Transforms entered data to conform to the data structures and relieves us of the chore of making a distinction between the data logical format and the data physical format;
- Creates a security system and enforces security and privacy within the database;
- Creates complex structures that allow multiple-user access to the data;
- Provides backup and data recovery procedures to ensure data safety and integrity;
- Promotes and enforces integrity rules to eliminate data integrity problems, thus allowing us to minimize data redundancy and maximize data consistency;
- Provides data access *via* a **query language** (a nonprocedural language) and *via* procedural (3GL) languages.

Managing the Database System: A Shift in Focus

The role of the DP specialist or the DP manager changes from emphasizing programming to focusing on the broader management aspects of the organization's data resource and on the administration of the complex database software itself. So DP manager may be promoted to systems **administrator** or to a **database administrator.**

From a management point of view, the database system is composed of five major parts:

1.	*Hardware:*	The computer
2.	*Software:*	The operating system, DBMS. applications and utilities
3.	*People:*	Systems administrator, database administrators, programmers, end users
4.	*Procedures:*	The instructions and rules that govern the design and use of the database
5.	*Data:*	The collection of facts stored in the database, includes metadata

Database Design and Modeling

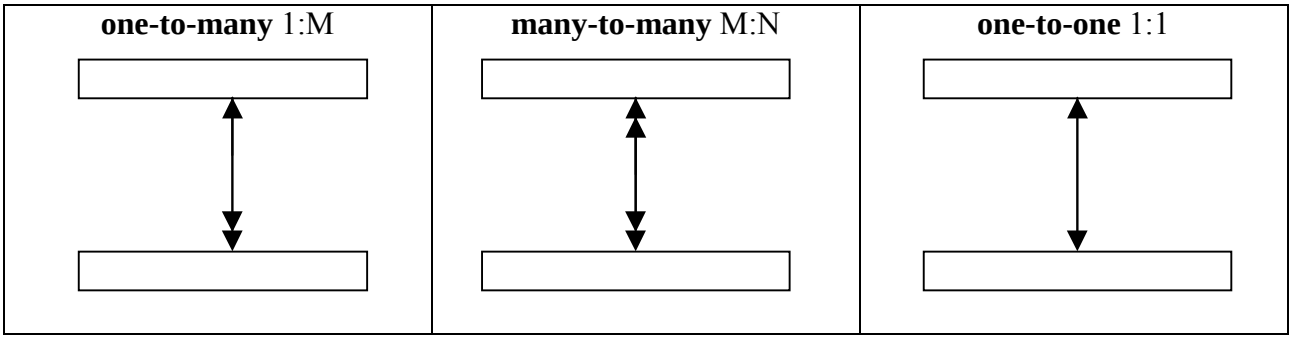
The DBMS makes it possible to tackle far more sophisticated uses of the data resource and the kinds of data structures created and the extent of the relationships among them play a powerful role, therefore, database *design* becomes a crucial activity.

Database design is made much simpler by using models. Models are simplified abstractions of real-world events or conditions and will enable us to explore the characteristics of entities and their relationships.

A database model is a collection of *logical* constructs used to *represent* the data structure and the data relationships found within the database. Database models may be grouped into two categories:

- ***conceptual* model** focuses on the logical nature of the data representation and is concerned with *what* is represented in the database;
- ***implementation* model** emphases on *how* information is represented in the database or on how the data structures are implemented to represent what is modeled. Implementation models include the **hierarchical database model, the network database model, and the relational database model.**

Conceptual models include the Entity Relationship (E-R) model and the Object-Oriented model. They use three types of relationships to describe associations among data:



Independence: The Comparison of Databases and File systems

	Data Independence	Structural Independence
File system	No	No
Hierarchical Database	Yes	No
Network Database	Yes	No
Relational Database	Yes	Yes

The Relational Database Model

Background:

- Earlier database models (hierarchical, network) demanded to know the physical details of database structures and it made good database design difficult at best;
- The lack of *ad hoe* query capability put heavy pressure on programmers;
- And any structural change in the database still had to produce havoc in all the application programs;
- Many database old-timers can recall the interminable information delays.

The relational model, first developed by E. **F. Codd** (of IBM) in 1970, represents a major breakthrough for both users and designers. Codd's work was considered ingenious but impractical in 1970; the model's conceptual simplicity was bought at the expense of computer overhead, computers lacked the power to implement the relational model.

The relational database is *perceived by the user* to be a collection of **tables in which data are stored**.

Each of the tables is a matrix consisting of a series of row/column intersections. Tables, also called **relations,** are related to each other by **sharing a common entity** characteristic.

Although the **tables are completely independent** of one another, it is easy to connect the data between tables. The relational model provides **a minimum level of controlled redundancy** to eliminate most of the redundancies commonly found in file systems.

A relational table stores a collection of related entities. In this respect, the relational database table resembles a file. But there is **one crucial difference between a table and a file**: The former yields complete data and structural independence because it is a purely *logical* structure. How the data are *physically* stored' in the database is of no concern to the user or the designer; and this property of the relational model turns out to be the source of a real revolution.

Advantages:

- the relational database model achieves both data independence and structural independence.
- the relational database model has very powerful and flexible query capability, like **Structured Query Language (SQL)**; a so-called **fourth-generation language** (4GL). A 4GL allows the user to specify *what* must be done *without* specifying *how* it must be done (it means, the relational database tends to require less programming than any other database);

History: *SQL is based on work done by IBM. In 1974, IBM first used Structured English Query Language, or SEQUEL, for use on its VS/2 mainframes. In 1976, a much- refined version (SEQUELS emerged. SEQUEL/2 became the SQL used in IBM's mainframe relational database products, such as SQL/OS and the popular DB2. Given IBM's size and marketing ability, SQL is an important product. The American National Standards Institute (ANSI) adopted standards in 1986 that further legitimized SQL as the standard relational database language. In 1987, IBM incorporated SQL into OS/2 as the standard database query language for its Database Manager. Other vendors have now incorporated SQL into their database wares, and SQL-based database applications now run on micros, minis, and mainframes.*

Any **SQL-based** relational database application involves **three parts**:

- a user interface (menus, query operations, report generators, etc., it’s a software vendor's idea of meaningful interaction with the data);
- a set of tables within the database,

- the SQL "engine." (it does the tough database jobs, like to create table structures, maintain the data dictionary and the system catalog, channel database table access and maintenance, translate user requests into a format that the computer can handle).

Disadvantages:

- the RDBMS needs the substantial hardware and operating system overhead, it tends to be slower than the other database systems;
- relatively untrained people find easy to generate reports, queries, a proper database, and if the database grows, lack of proper design slows the database and produce data anomalies.

A logical View of Relational Database's Model

The relational database model's basic data components are **entities** and their **attributes**. These basic data components fit into **a logical construct** known as **a table**. Tables may be treated as **independent** units. But the tables within the database may be **related** to one another.

A database stores and manages both data and metadata. Physically, the data within a database do *not* reside in separate files that store related records; instead, data are stored in a single structure. The relational model views data ***logically*** rather than ***physically***.

Entities and Attributes

Relational design begins by defining the required entities: a person, place, event, or thing, for which data are collected.

Each entity has certain characteristics known as attributes: number, name, grade point average (GPA), date of enrollment, date of birth, home address, phone number, major, and so on.

A grouping of related entities becomes an **entity set.** Each entity set is named.

Tables and Their Characteristics

The logical view of the relational database is facilitated by the creation of data relationships based on a (logical) construct known as ***a table.*** *which contains a group of related entities - that is,* ***an entity set.*** A table is also called ***a relation.***

Because of tables, data in relational model are considered to be **homogeneous**. Every row has the same format. It is a question of **data modeling** how real entities or data could be reflected in the homogeneous format of relational tables.

Note: The word *relation* is based on the mathematical set theory. The relation in mathematics is any set R:

$$R \subset A \times A$$

in a Cartesian product of a set A, together with some logical properties (which for instance, are like symmetry, antisymmetry, etc.).

There are logical operations on relations, like

$$R \cup S, R \cap S, R \subset S, R \not\subset S, R \triangleright\!\triangleleft S, R - S \text{ , etc.}$$

In order to implement mathematical notion of relations into computer software, it is necessary to develop **a schema of relation**. This was done by Codd.

The **mathematical theory of relational algebra** is presented for instance in ***[D. Maier. The Theory of Relational Databases,*** *Oregon Graduate Center, Computer Science Press, 1985****]***

or ***[C.J.Date. An Introduction to Database Systems, 6th Edition,*** *Addison-Wesley Publishing Company, 1996]***.**

The table is composed of intersecting rows and columns. Each of the table's rows - ***(tuples),*** represents a single entity, and each of that entity's attributes is represented by a column in such a table.

Note: Relational database terminology is very **precise**. Really, there is almost a mistake to name ***rows as records***, and ***columns as fields.*** Occasionally, ***tables*** are labeled ***files.*** The database table is **a logical** rather than **a physical** concept, and the terms *file, record,* and *field* describe physical concepts. In fact, some database software vendors still use the familiar file terminology.

Limits on Table and Columns

Software usually places limits on the names that may be used to label tables and their components. There are the following general rules:

1. Table (relation) names are limited to eight characters;
2. Column (attribute) names are limited to ten characters;
3. Column names cannot begin with a digit.

A table may contain many rows and columns. The table's size is limited (or unlimited) by the relational software being used.

1. A table exists for each entity set.
2. Each of the entity's attributes is represented by a column in the table.
3. Each of the attribute's values is referenced by row.
4. Each row/column intersection contains only a single data value. Data must be classified according to their format and function. Although various DBMSes may support different data types, most support at least the following:
   - *Numeric.* Numeric data are composed of data with meaningful arithmetic procedures;
   - *Character.* Character data, also known as *string data* or *strings,* are composed of any character or symbol not intended for any sort of mathematical manipulation;
   - *Date.* Date attributes contain calendar dates stored in a special format known as a *Julian date format.* Although the physical storage of the Julian date is immaterial to the user and designer, the Julian date format allows us to perform a special kind of arithmetic known as *lulian date arithmetic,* (to determine the number of days that elapsed between 05/12/90 and 07/25/92 is simply to subtract one date from another);
   - *Logical.* Logical data can have only a true or false condition;
   - *BLOB or so*. Such data may need some other operations.
5. Each row must have a **primary** key (an attribute (or a combination of attributes) that uniquely identifies any given entity (row)); this is the table to exhibit **entity integrity.**
6. Each column represents an attribute, and each column has a distinct name.
7. All the values in a column match the entity's row characteristics.
8. The column's range of permissible values is known as its **domain** (the *database* use of the term domain is slightly different from that used by mathematicians).
9. Each row carries information describing each one of the entity's attributes.
10. The order of the rows and columns is immaterial to the user.

Keys: Links between Tables

What makes the relational database work is *controlled redundancy.* Tables within the database share common attributes that enables to hook the tables together.

A key is a device that helps to define **entity relationships**.

The **link** is created by having two tables **share a common attribute** (the primary key of one table appears again as the link (or **foreign key**) in another table). The foreign key **must** contain **values** that match the other table's primary key value, or it must contain a *null* "value" in order for the table to exhibit **referential integrity.**

The primary key plays **an important role** in the relational environment and is to examine more carefully. There are several other kinds of keys that warrant attention, like **superkeys, candidate keys, and secondary keys**.

The key's role is made possible because one or more attributes within a table display a relationship known as **functional dependence:**

- The attribute *(B)* is functionally dependent on attribute (A) if each value in column (A) *determines* one and only one value in column (B).

In relation to the concept of functional dependence, a key is an attribute that determines the values of other attributes within the entity and it may take more than a single attribute to define functional dependence; that is, a key may be composed of more than one attribute. Such a multi-attribute key is known as a **composite** key.

Given the possible existence of a composite key, it is further to refine the notion of functional dependence by specifying **full functional dependence:**

- If the attribute (B) is functionally dependent on a composite key (A) but not on any subset of that composite key, the attribute (B) *is fully functionally dependent* on (A).

A superkey is any key that identifies each entity uniquely (the superkey functionally determines all of the entity's attributes).

A candidate key may be described as a superkey without the redundancies (it is a superkey that does ***not*** contain a subset of attributes that contain a superkey).

A secondary key is defined as a key that is used strictly for data- retrieval purposes (the primary key is the customer number and the secondary key is the customer last name and the street name, for instance).

Relational Database Keys

Superkey	An attribute (or combination of attributes) that uniquely identifies each entity in a table.
Candidate Key	A minimal superkey. A superkey that does not contain a subset of attributes that is itself a superkey.
Primary Key	A candidate key selected to uniquely identify all other attribute values in any given row. Cannot contain null entries.
Secondary Key	An attribute (or combination of attributes) used strictly for data- retrieval purposes.
Foreign Key	An attribute (or combination of attributes) in one table whose values must either match the primary key in another table or be null.

A Summary of the Integrity Rules

ENTITY INTEGRITY

Requirement	No null entries in a primary key.
Purpose	Guarantees that each entity will have a unique identity.

REFERENTIAL INTEGRITY

Requirement	Foreign key must have either a null entry or a matching entry.
Purpose	Makes it possible for an attribute NOT to have a corresponding attribute, but it will still be impossible to have an invalid entry.
Example	A customer may not (yet) have an assigned sales representative (number), but it will be impossible to have an invalid sales representative (number).

Relational Database Operators

Relational algebra defines the theoretical way of manipulating table contents. It is using the eight relational functions:

SELECT, PROJECT, JOIN, INTERSECT, UNION,
DIFFERENCE, PRODUCT, DIVIDE.

Theoretically relational-algebra functions on existing tables produces new tables.

The degree of **relational completeness** may be defined by the extent to which relational algebra is supported. To be considered minimally relational, the DBMS must support the key relational functions SELECT, PROJECT, and JOIN. Very few DBMSes are capable of supporting all eight relational operators.

The use of the relational-algebra functions may be illustrated as follows:

- SELECT yields values for *all attributes* found in a table. SELECT may be used to list *all of the values* for each attribute, or it may yield *selected values* for each attribute (SELECT yields a *horizontal subset* of a table);
- PROJECT produces a list of *all values* for *selected attributes* (PROJECT yields a *vertical subset* of a table);
- JOIN allows to *combine* information from two or more tables. *JOIN is the real power behind the relational database scene,* allowing the use of independent tables linked by common attributes. There are a few forms of JOIN:
  - **a natural JOIN** (contains *both* copies of the tables joined on the selected column),
  - an **equiJOIN (**yields a table that does not include unmatched pairs and provides only the copies of the matches),
  - **an outer JOIN (**the unmatched pairs would be retained, and the values for the unmatched other table would be left vacant or null).
- INTERSECT produces a table that contains all rows that appear in both tables. The attribute characteristics must be identical for the intersecting attributes to yield valid results.
- UNION combines all rows from two tables. Attribute characteristics must be identical.
- DIFFERENCE yields all rows in one table that are not found in the other table; that is, it "subtracts" one table from the other. Attribute characteristics must be identical for the attributes to be used in the DIFFERENCE.
- PRODUCT produces a list of all possible pairs of rows from two tables. Therefore, if one table has ten rows and the other table contains twenty rows, the PRODUCT yields a list composed of 10 X 20 = 2OO rows.
- DIVIDE requires the use of one single-column table and one two-column table. Both tables contain the common column. The resulting table consist of "unshared" column with values associated with *every* value in two-column table.

Relational Database Software Classification

The three main building blocks of the relational model are: logical structure, integrity constraints, and operators. However, not the all DBMSes support all three building blocks. Some DBMSes sold as "relational" support only the minimum three relational operators SELECT, PROJECT and JOIN, without supporting the logical model and without enforcing the integrity constraints. Some others support only the integrity constraints and the minimum three relational operators.

Relational Database Classification

CLASSIFICATION	DESCRIPTION OF FEATURES
Fully relational	Supports all eight relational-algebra functions and also enforces both entity and referential integrity rules
Relationally complete	Supports all eight relational-algebra functions but not the integrity rules
Minimally relational	Supports only SELECT, PROJECT and JOIN
Tabular	Supports only SELECT, PROJECT and JOIN and requires that an access paths be defined by the user

The distinction between various grades of relational-database software implementations is not trivial. It is quite a bit of extra design and programming work to make applications conform to the integrity rules if the software is minimally relational. If the software is tabular, only a few of the relational model's benefits could be claimed.

The data dictionary and the system catalog

The data dictionary is used to provide a detailed accounting of all tables found within the user/designer-created database: it yields (at least) all the attribute names and characteristics for each of the tables in the system. The data dictionary contains metadata-data about data.

The system catalog also contains metadata, like data dictionary. The system catalog may be described as a very detailed **system data dictionary** that describes *all* objects within the database, including data about table names, who created the tables and when, the number of columns in each table, the data type corresponding to each column, index file names, index creators, authorized users, access privileges, and so on. Because the system catalog contains all of the required data-dictionary information, the terms *system catalog* and *data dictionary* are often used as synonyms. In fact, current relational database software generally provides only a system catalog, from which the data-dictionary information may be derived. The system-catalog tables may be queried just like any user/designer-created tables.

The system catalog automatically produces database **documentation,** and it also makes possible for the RDBMS to check for and eliminate **homonyms** (similarly sounding words with different meanings) and **synonyms** (different names to describe the same attribute).

The synonyms and homonyms should be avoided. For instance, if the same attribute is found in different tables, that attribute should carry the same name in each table. Ideally, a foreign key should carry the same label as the primary key of the originating table.

Relationships within the Relational Database

The **one-to-many** (l:M) and **many-to-many** (M:N) relationships are just as important in the relational environment. The third relationship, the **one-to-one** (1:1), may also be encountered.

The l:M relationship is the **relational ideal** and constitutes the relational database's main building block. The existence of a 1:1 relationship often means that the entity components were not defined properly, and it may indicate that the two entities actually belong in the same table!

There is a close ties between relational database design and an **Entity Relationship (E-R) model. A**n E-R model provides a simplified picture of the relationship among entities.

Note: A relational table is equivalent to an entity set. However, the creators and subsequent users of E-R models use *entity* in lieu of *entity set.*

The one-to-many (l:M) relationship is easily implemented in the relational model: Just make sure that the primary key of the one is included in the table of the *many.*

A many-to-many (M:N) relationship is a much more troublesome proposition in the relational environment. The following E-R model shows a many-to-many relationship:

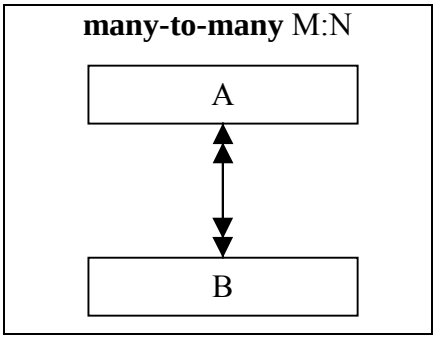


Table A:

A-num	A-name	B-num

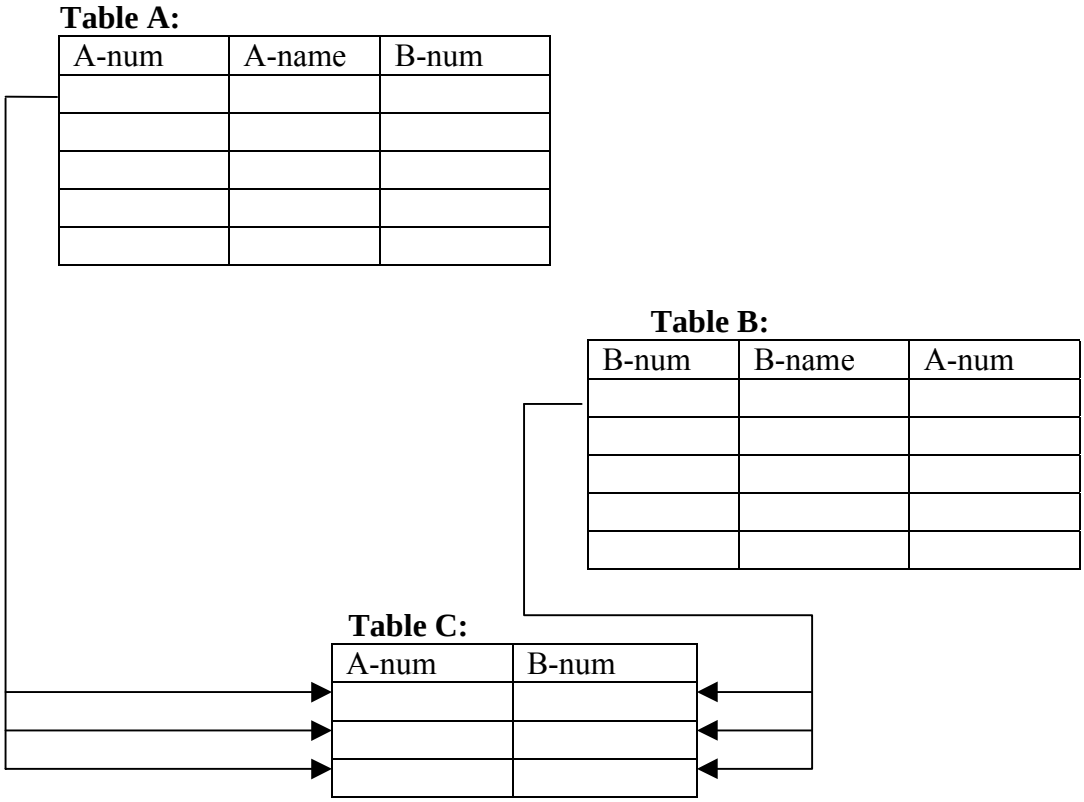
Table B:

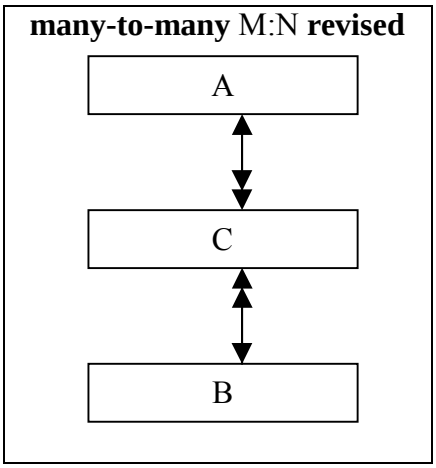
B-num	B-name	A-num

Although the M:N relationship depicted makes sense, it *should not be implemented as shown,* for a couple of good reasons:

- the tables contain much data duplication, and the resulting data redundancy is wasteful and leads to the troublesome data anomalies;
- the pointer movement becomes very complex and is likely to lead to errors and system inefficiency.

Fortunately, it is easy to circumvent the problems inherent in the many-to- many (M:N) relationship by creating a so-called **composite entity or bridge entity.** Such an entity is one whose primary key is composed of the combination of the primary keys of the entities that must be linked. This entity leads to a **linking table:**





Note: In addition to the linking attributes, the (composite) table may also contain some relevant attributes. Also the creation of the linking table produces the desired l:M relationships within the database. *And especially note that the tables A and B now contain only one row for entity.* Only the linking table contain redundancies.

Index key	**Pointers to table(s) entries**
values of entities in index key (search key)	referencies
	referencies
	referencies
	referencies

Indexes

Indexes in the relational database environment work for search and retrieval purposes. Data retrieval speed may be increased dramatically by using indexes. From a conceptual point of view, an **index** is composed of an **index key** and a set of pointers. The index key is the index's reference point. Indexes may be created very easily with the help of SQL commands.

Earlier Database Models

Database notions were developed to address the file system's inherent weaknesses. Rather than depositing data within different files, data are kept within a single data repository, thus enabling the DBMS to maintain tight control over the database activities. Three major database models:

hierarchical, network, and relational

became commercial successes. All three use a DBMS to help data managers perform their tasks more capably than was possible with the file system. However, the extent of the DBMS activities and its user-friendliness differs rather markedly among the three database systems.

The Hierarchical Database Model

This model is based on a tree structure that is composed of a root segment, parent segments, and child segments. The **segment** is the equivalent of a **file's record** type. The hierarchical database model depicts a set at one-to-many (l:M) relationships between a parent and its children. The hierarchical model uses a hierarchic sequence or **preorder traversal** to navigate through its structures, always starting at the left side of the tree.

Given its parent/child structure, the hierarchical model yields database integrity and consistency: There cannot be a child record without a parent. In addition, **a well-designed** hierarchical database tends to be **very efficient** when large amounts of data or many transactions are to be handled.

Unfortunately, the hierarchical model yields only **some of the benefits** of data independence. Although changes in data characteristics **do not require** changes in programs that use such changed data, any **change in the database structure still mandates extensive rework**.

For example, **a segment-location change** triggers a requirement for changes in all the programs that use such a **segment within their hierarchical path**. Moreover, **changes in segments or their location** require DP managers to **perform complex system-management tasks**. Unless special care is taken, **segment deletion may lead to the unintended deletion of all segments below the deleted segment.**

Because the hierarchical model **does not include *ad hoe*** querying capability, applications **programming** tends to be **extensive**.

Hierarchical design also tends to be difficult; the DBMS efficiency is thus hard to capture. In addition, **a multiparent relationship is difficult to implement**, as is a **many-to-many** (M:N) relationship.

Finally, since there is **no standard** hierarchical model, it lacks **portability**.

The hierarchical model's implementation is cumbersome, and it lacks the **ability to work** well within the constraints of **complex logical relationships**. Although **IMS** still has a large installed base, the hierarchical model has lost its former dominance in the database marketplace.

History and Background.

North American Rockwell was the prime contractor for the Apollo project, which culminated in a moon landing in 1969. Bringing such a complex project **to** a successful conclusion required the management of millions of parts. Information concerning the parts was generated by a complex computer file system.

When North American Rockwell began to develop its own database system, an **audit** of the collection of computer tapes revealed that more **than 60 percent** of the data were **redundant**.

This forced North American Rockwell to develop an alternative strategy for managing such huge data quantities and software known as **GUAM (Generalized Update Access Method)** was developed. GUAM was based on the recognition that the many smaller parts would come together as components of still larger components, and so on, until all the components came together in the final unit.

The GUAM's ordered arrangement conformed to the '**upside-down tree**" structure or as a **hierarchical structure.**

In the mid-sixties, **IBM** joined North American Rockwell to expand GUAM's capabilities, replacing the **computer-tape** medium with more **up-to-date disk computer storage**, which allowed the introduction of complex **pointer** systems.

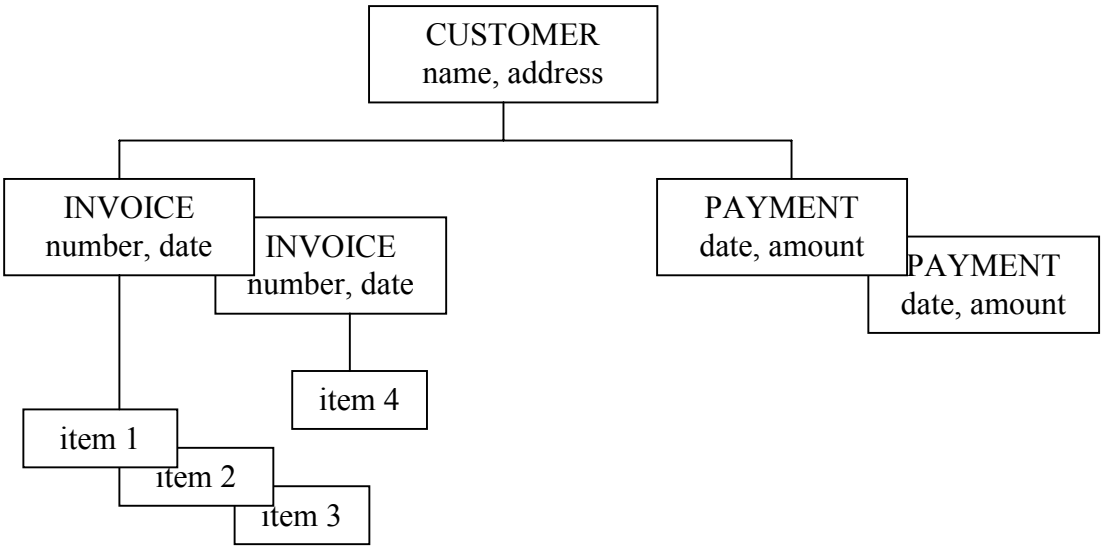
The results of the joint Rockwell-IBM effort later became known as the **Information Management System (IMS).** Given IBM's development and marketing clout, IMS became the world's leading mainframe hierarchical database system in the seventies and early eighties. However, although IMS still retains a large installed base, it is rapidly being replaced by IBM's relational database software known as DB2.

The hierarchical model turned out to be the **first major commercial** implementation of a growing pool of database concepts that were developed to counter the computer file system's inherent shortcomings. Although the hierarchical model no longer serves as a leading-edge database standard, it is **worthwhile to understand** at least a few of its characteristics, for these reasons:

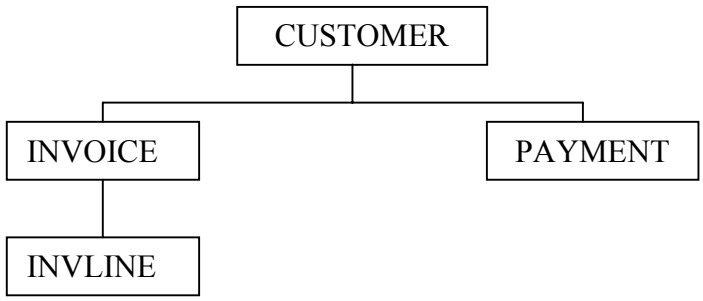
- Its basic concepts form the basis for subsequent database development.
- Its limitations lead to a different way of looking at database design.

Basic Hierarchical Model Concepts

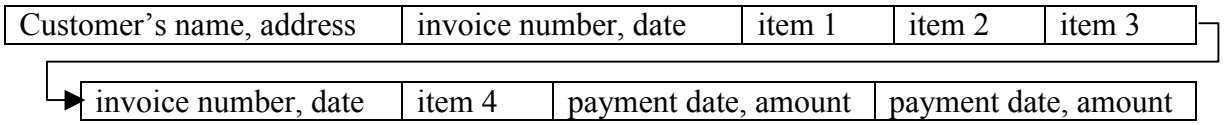
A hierarchical database model has its basic concepts and components. From a hierarchical point of view, entities and relations between them can be represented by **a hierarchy** based on **segments** and their types. Each **segment type** represents a specific **entity set** and contains several **segment occurrences.** This is used to represent a **1:M relationship.** The segment can be **a parent** to some another segment, which is a **child segment** to the parent. The segment in a root level of the tree is called a **root segment.** Each child segment occurrence is related to only one parent segment occurrence. Each match of a root segment occurrence with its child segment occurrences represents a **hierarchical database record** occurrence. Below is an illustration of several relationships produced by the root segment occurrence of the CUSTOMER. There are two INVOICE segment occurrences and two PAYMENT segment occurrences, etc.



This occurrence of a hierarchical database record corresponds to the hierarchical structure (or model):



The trace of the path of hierarchical record:



The tree structure depicted in figure cannot be duplicated on the computer's storage media. Instead, the tree is defined by the path that traces the parents to their children, beginning from the left. This **ordered** sequencing of segments to represent the hierarchical structure is known as the **hierarchical path**.

Given the structure, the hierarchical path for the record composed of the segments may be traced from the root, starting at the leftmost segment. The **left-list** path is known as the **preorder traversal** or the hierarchic sequence. Given such a path, designers must make sure that the **most frequently accessed** segments and their components are located closest to the **tree's leftmost** branches.

Contrasting File Systems with the Hierarchical Model

There are similarities as well as differencies between file structures and the hierarchical database.

The records in each of files are physically isolated from the records in the other files.

In sharp contrast to the file system, the hierarchical model merges the separate physical files into a single structure known as a database. Therefore, there is **no equivalent** of a file in the hierarchical model. The **fields** encountered in the file system are **simply segment components** in the hierarchical database.

Given the contrasting structures the file system's user must **maintain physical** control of the indexes and pointers that validate data integrity. However, in the hierarchical model the DBMS takes care of such complex chores, and the **pointer movement is transparent** to the user. (The word transparent indicates that the user is unaware of the system's operation.).

Defining A Hierarchical Database

IBM's Information Management System (IMS) uses a language named **Data Language One (DL/1).** At the **conceptual** level, IMS may control **several** databases. **Each database** is composed of a **collection of physical records** (segments) that are occurrences of a single tree structure. Therefore, **each tree requires its own database**. Each of the physical databases is defined by a database description (**DBD**) statement when the database is created.

The hierarchical model's **segment relationships** are determined explicitly by the user when the database is defined, using the data definition language (**DDL**). These segment relationships **do not depend** on the contents of a field in the **child** record (as was true in the relational model.) Therefore, the relationships between the segments cannot be derived *via* each segment's components or fields.

DL/I is used to describe the conceptual and logical views of the database. The **conceptual view** encompasses the entire database as seen by the database administrator; the **logical view** describes the programmer's and user's perception of the database. The logical view is thus much more **restrictive**, limiting the programmer/user to the portion of the database that is currently in use. The existence of logical views constitutes a **security** measure that helps avoid the unauthorized use of the database. Both the conceptual and logical views are **necessary** in order to work with a hierarchical database.

The Conceptual View Definition

The **tree** structure is defined starting from the **left**. Therefore, the sequence shown in the DDL conforms to the path CUSTOMER → INVOICE → INVLINE → PAYMENT. Based on this structure definition, the DL/I statements used to define the conceptual view of the database as seen by the database administrator:

```
 1    DBD     NAME=CUSREC, ACCESS=HISAM
 2    SEGM    NAME=CUSTOMER,BYTES=60
 3    FIELD   NAME=(CUS-NUMBER,SEQ,U),BYrES=5,START=l
 4    FIELD   NAME=CUS-NAME,BY]FES=25,START=6
 5    FIELD   NAME=CUS-ADDRESS,BYTES=30,START=31
 6    SEGM    NAME=INVOICE,PARENT=CUSTOMER,BYrES=21
 7    FIELD   NAME= (IN-NUMBER,SEQ,U),BYrES=6,START= 1
 8    FIELD   NAME=IN-DATE,BYTES=8,START=7
 9    FIELD   NAME=IN-AMOUNTBYFES=7,STARr=16
10    SEGM    NAME=INVLINE,PARENT=INVOICE,BYTES=35
11    FIELD   NAME=(INV-PRODUCTSEQ,M),BYTES=25,START=l
12    FIELD   NAME=INV-PRICE,BYTES=7,START=26
13    FIELD   NAME=INV-QUANTBYTES=3,START=33
14    SEGM    NAME=PAYMENTPARENT=CUSTOMER,BYrES=21
```

```
15      FIELD    NAME=(PAY-NUMBER,SEQ,U),BYTES=6,START=l
16      FIELD    NAME=PAY-DATE,BYTES=8,START=7
17      FIELD    NAME=PAY-AMOUNTBYITS=7,START=14
18      DBGEN
19   FINISH
20   END
```

The hierarchical model's definition of the database must conform to its physical characteristics. Even given the simplified DL/I syntax, the details make the hierarchical model sufficiently complex to be described as a system designed by programmers for programmers. For instance, the physical storage details may require the definition of complex storage schemes, such as:

1. HSAM (Hierarchical Sequential Access Method)
2. SHSAM (Simple Hierarchical Sequential Access Method)
3. HISAM (Hierarchical Indexed Sequential Access Method)
4. SHISAM (Simple Hierarchical Indexed Sequential Access Method)
5. HDAM (Hierarchical Direct Access Method)
6. HIDAM (Hierarchical Indexed Direct Access Method)
7. MSDB (Main Storage DataBase)
8. DEDB (Data Entry DataBase)
9. GSAM (Generalized Sequential Access Method)

Specific access methods are best suited to particular kinds of applications. HSAM, SHSAM, HISAM, and SHISAM are particularly well suited for storing and retrieving data in hierarchic sequence, putting parent and children records in contiguous disk locations. (GSAM is a special case of the sequential access method.) On the other hand, if direct-access pointers are required to keep track of the hierarchy of segments, HDAM, HIDAM, MSDB, or DEBD are preferred. This latter series of access methods is generally more valuable when many (and frequent) changes are made to the database. Generally, the IMS manuals suggest:

1. Use HSAM when relatively small databases with relatively few access requirements are used.
2. Use HISAM with databases that require direct segment access, especially when
   a. Fixed record lengths are used.
   b. All segments are the same size.
   c. Few root segments and many child segments exist.
   d. Few deletions are made.
3. Use HDAM with databases designed for fast direct access.
4. Use HIDAM with databases whose users require both random (direct) and sequential access.
5. Use MSDB with databases that use fixed-length segments and that require very fast processing. MSDB will reside in virtual storage during execution.
6. Use DEBD with databases that are characterized by high data volume.
7. Use SHSAM, SHISAM, and GSAM when you frequently import and export data between database and nondatabase applications.

It may happen database definition requires each segment to be identified by a so-called **sequence field.** The identifier is also known as a **key**. Working with sequence fields requires:

1. Sequence fields allow direct access to segments when working with HISAM, HDAM, or HIDAM access methods. Such access methods make it possible to address segments directly, without having to search the entire database. Direct access increases performance substantially.
2. Sequence fields do not have to be defined for every segment.
3. Sequence fields may be either unique (U) or duplicate (M).

An IMS database is (structurally) rather limited:

1. Each database can have a maximum of 255 different segment types.
2. Each segment can have a maximum of 255 segment fields.
3. Each database can have a maximum of 1,OO0 different fields.

The Logical View Definition

The logical view depicts the application program's view. Application programs use embedded DL/I statements to manipulate the data in the database. Each application that accesses an IMS database requires the creation of a **program specification block (PSB).** The PSB defines the database(s), segments, and types of operations that can be performed by the application. In other words, the PSB represents a logical view of a selected portion of the database. The use of PSBs yields better data security as well as improved program efficiency by allowing access to only the portion of the database that is required to perform a given function.

Defining a Program Specification Block for an Application

```
1  PCB      DBNAME = CUSREC
1  SENSEG   NAME = CUSTOMER, PROCOPT = G
2  SENSEG   NAME = PAYMENT, PARENT = CUSTOMER, PROCOPT = G
3  SENFLD   NAME = PAY-DATE, START = 8
4  SENFLD   NAME = PAY-AMOUNT, START = 15
5  PSBGEN   LANG = COBOL, PSBNAME = ROBPROG
```

The application program and the database system communicate through a common storage area in primary memory known as the **program communication block** (PCB). The PSB contains one or more PCBs, one for each database that is accessed by the application program.

Then the DL/I may be used to define the type of access or processing option (PROCOPT) granted to the program. (The access types are (**G**)et, (**I**)nsert, (**R**)eplace, and (**D**)elete).

The SENSEG (SENsitive SEGment) declares the segments that will be available, starting with the root segment. The SENFLD indicates which fields are available to the program.

The creation of the database structure and the PSBs is not based on interactive operations. Instead, an independent utility programs must be run from the operating-system prompt. Therefore, it must be recreated (recompiled) the database definitions and reloaded **and recreated and validated all the user views to make any changes to the database.**

The order of the SEGM statements indicates the physical order of the records in the database. In other words, the **physical order** represents the **hierarchical path** that has to be followed to access any segment.

IMS provides support for several different data-access methods:

- some are very efficient in terms of sequential file processing;
- others work well in an indexed file environment;
- others work best in a direct-access environment.

Loading IMS Databases

An IMS database **must be loaded** before any program can access it. It is not allowed to load a database from an interactive application program. Instead, a batch program must be used to

perform the loading, and this batch program must be run in 'load' mode (PROCOPT=L in the PCB).

The database must be loaded in the **proper** hierarchic sequence; that is, **the segment order is crucial.** (Load the parent segments before loading the child segments!) If sequence fields are defined, the segment order must conform to the sequence field order. Failure to maintain the proper segment order will cause the subsequent applications programs to fail.

Accessing The Database

Hierarchical databases are so-called **record-at-a-time** databases. Basically, the term *record at a* time indicates that the database commands **affect** a single record only. Other database types, such as the relational database, allow a command to affect several (many) records at a time.

The record-at-a-time structure implies that each record is accessed independently when database operations are performed. Therefore, to access a specific record it must follow the tree's hierarchical path, starting at the root and following the appropriate branches of the tree, using preorder traversal.

After the database and its characteristics have been defined, it's possible to navigate through the database by using the data manipulation language (DML) invoked from some host language such as COBOL, PL/1, or assembler. Keep in mind that some lines of code must be written by an **experienced** programmer before we can access the database. End users are usually not able to cope with such a complex environment.

IMS requires the use of a (3GL) host language to access the database. In order to correctly communicate with the application program, IMS assumes the use of certain parameters. Therefore, each application must declare:

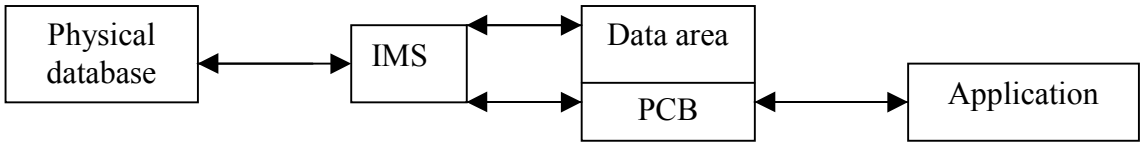
1. An input area (program record area) reserved to
   a. Receive the data retrieved from the database.
   b. Temporarily store data to be written into the database.
2. A PCB to store a return code for every operation that is executed. (The program must check this area to see if the requested operation was completed successfully.)

A COBOL application communicates with the IMS DBMS through **call** statements in its procedure division and uses the PCB. When the application program calls IMS, several so-called *flow parameters* are needed:

1. The function code; that is, the operation to be executed on the database.
2. The PCB name to be used.
3. The input area address.
4. The (optional) segment search argument (SSA). The SSA parameter identifies the key of the segment being retrieved.

After the completion of a call to the database, the program must check the status of the return code in the PCB to ensure that the operation was executed correctly.

The usage of PCB:



Data Retrieval: Get Unique

The IMS statement Get Unique (GU) is used to retrieve a database segment into the application program input area or record area. The syntax for the Get Unique statement is
GU (segment) (SSA)

Sequential Retrieval: Get Next

The Get Next (GN) statement is used to retrieve segments sequentially. (Naturally, the retrieval sequence is based on the preorder traversal requirements!) The GN syntax conforms to the format GN (segment) SSA

Get Next Within Parent

Get Next Within Parent (GNP) will return all of the segments within the current parent.

Data Deletion and Replacement

The **Get Hold** (GH) statement is used to hold a segment for delete or replace operations. There are three different Get Hold statements:

Statement	**Meaning**
GHU	Get Hold Unique
GHN	Get Hold Next
GHNP	Get Hold Next Within Parent

Used in combination with the GH statement, DLET deletes a segment occurrence from the database. If a **root segment** is deleted, all the dependent segments are deleted, too.

The REPL statement allows us to change (update) the contents of a field within a segment. REPL, too, requires the GH operation before it can be invoked. The REPL function cannot be used to update a key field. Instead, **first delete** the record and **then insert** the updated version. The application program should use the input area to store the necessary fields that are to be updated and the new values. The operation sequence thus becomes:

1. Retrieve the data and put it in the input area.
2. Make the changes in the input area.
3. Invoke REPL to move the changed values into the physical database.

Adding a New Segment to the Database.

The ISRT (Insert) statement is used to add a segment to the database. The parent segment must already exist if a child segment is to be inserted. The segment will be inserted in the database in the sequence field order specified for the segment. The input area in the applications program must contain the data to be stored in the segment.

Logical Relationships

There is a problem with a scenario, when a segment has two parents, a condition that cannot be easily supported by the hierarchical model. The multiple-parent problem can be solved by creating a **logical relationship** between **one segment** and **another segment**. **One segment** becomes the **logical child** of **another segment** and P **another segment** becomes the **logical parent** of **one segment**. Unfortunately, this solution has some drawbacks:

- Implementing such a solution yields an even more complex applications environment.

- Creating logical parent/child relationships is very complex and requires the services of an experienced programmer. To accomplish the task, referential rules must be defined for each of the operations (Insert, Replace, and Delete) for each logical segment involved in the two physical databases. The rules may be unidirectional or bidirectional, depending on which way we want to access the database.

Nonetheless, using logical relationships, it is possible to link two independent physical databases and treat them as though they were one. Thus, logical relationships allow to reduce data redundancy. In addition, IMS can manage all the data required to link the databases in logical relationships; it is always better to have the DBMS software do the delicate work of keeping track of such data rather than trusting the applications software to do such chores.

IMS supports three different types of logical relationships:

- *Unidirectional logical relationships* are established by linking a logical child with a logical parent in a one-way arrangement, that is, a pointer in the logical child pointing to the logical parent.

  The two segments can be in the same database, or they may be located in different databases. If the two segments of the unidirectional relationship are located in different databases, the segments are treated independently of one another. Therefore, if a parent segment is deleted, the logical children are not deleted because the logical parent does not point to the logical child.
- *Bidirectional physically paired logical relationships* link a logical child with its logical parent in two directions. IMS creates a duplicate of the child segment in the logical parent's database and manages all operations (Insert, Delete, Replace) applied to the segments. IMS uses pointers in the logical-child segments pointing to their logical parents. The segments can be in one database, or they can be in different (physical) databases. Although the process creates data redundancy, IMS manages such redundancies transparently.
- *Bidirectional virtually paired logical relationships* are created when a logical child segment is linked to its logical parent in two directions. The virtually paired relationship is different from the physically paired relationship in that no duplicates are created; IMS stores a pointer in the logical parent to point to the logical child's database and another in the logical child to point to the logical parent. The virtually paired method thus reduces data duplication and overhead in the management of both hierarchical paths.

  The creation of bidirectional virtually paired logical relationships is a very delicate and cumbersome task that requires a skilled designer with extensive knowledge of the physical details required by this task.

IMS Rules for Logical Relationships

Rules for defining logical reiationships in physical databases

Logical Child

1. A logical child must have a physical and a logical parent.
2. A logical child can have only one physical and one logical parent.
3. A logical child is defined as a physical child in the physical database of its physical parent.
4. A logical child is always a dependent segment in a physical database and can, therefore, be defined at any level except the first level of the database.
5. In its physical database, a logical child cannot have a physical child defined at the next lower level in the database that is also a logical child.
6. A logical child can have a physical child. However, if the logical child is physically paired with another logical child, only one of the paired segments can have physical children.

Logical Parent

1. A logical parent can be defined at any level in the physical database, including the root level.
2. A logical parent can have one or more logical children. Each logical child related to the same logical parent defines a logical relationship.
3. A segment in a physical database cannot be defined as both a logical parent and a logical child.
4. A logical parent can be defined in the same physical database as its logical child, or in a different database.

Physical Parent

1. A physical parent of a logical child cannot also be a logical child.

The use of logical parents is rather limited: One of DL/1's restrictions is that any given segment can have only one logical parent. Such a restriction severely limits the ability to deal with complex structures. In fact, the two-parent problem became one of the reasons for the development of the network model.

Altering The Hierarchical Database Structure

Finally, the hierarchical model's database structure modifications are cumbersome. A simple request, just to add a field in the segment., often turns out to be something that is not naturally supported by the hierarchical system.

Database modifications require the performance of the following tasks in sequence:

1. Unload the database.
2. Define the new database structure.
3. Load the old database into the new structure.
4. Delete the old database.

Since these four tasks are time-consuming and potentially dangerous from a database point of view, database structure modifications require very careful planning, excellent system coordination skills, and a high level of technical understanding of the DBMS.

The Network Database Model

The network model represents an attempt to deal with many of the hierarchical model's limitations. The DataBase Task Group (DBTG) of the COnference on DAta SYstems Languages (**CODASYL**) produced standard network specifications for a network schema, subschemas, and a data management language. The data management language contained three basic components: a Data Definition Language (DDL) used to define schema components, a subschema DDL, and a Data Manipulation Language (DMLj designed to manipulate the database contents.

Although the network model resembles the hierarchical model, its structure easily accommodates the multiparent child. Even the basic model components are similar; the network member is the equivalent of the hierarchical child, and the network owner is the equivalent of the hierarchical parent.

The network model shares some of the hierarchical model's efficiencies and greatly improves others. For example, the model's data integrity is assured by the fact that a member record cannot exist without its owner. The network set defines the relationship between owner and member; its existence allows a program to access an owner record and all of tile member records

within a set, thus yielding greater data access flexibility than was possible with the hierarchical model.

As efficient as the network database model is, its structural complexity often limits its effectiveness: Designers must be very familiar with the database structure to capture its efficiency. Like the hierarchical model that preceded it, data independence is not accompanied by structural independence: Any change in the database structure requires that all subschema definitions be revalidated before any applications programs can access the database. In other words, the network database is sometimes difficult to manage well, and its complexity tends to cause database design problems.

A network database model implementation, like the hierarchical database model, may be represented by a tree structure in which 1:M relationships are maintained. However, the network model easily handles complex multiparent relationships without resorting to the creation of logical (as opposed to physical) database links.

The network model conforms to a database standard created by the COnference on DAta SYstem Languages (CODASYL) group. Because there are CODASYL standards governing the database concept and language, the network model's constructs are independent of the software produced by the various vendors. In fact, the only real differences exist in the implementation details and the selection of the system interfaces.

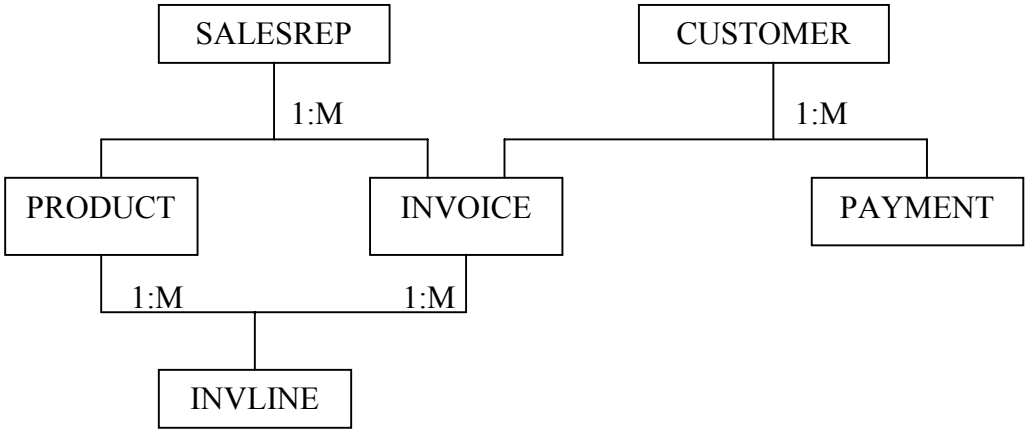
A close kinship exists between hierarchical and network models. For example, the network model's owner corresponds to the hierarchical model's parent, and the network model's member corresponds to the hierarchical model's child. However, the network model places a set between the owner and the member, using this set to describe the relationship between the two. The set's existence makes it possible to describe more complex relationships between owners and members than was feasible in the hierarchical model's parent-child structure.

Although the pointer movement is more complex in the network model than in its hierarchical counterpart, the DBMS creates and maintains the pointer system, making it transparent to the user and even to the applications programmer. However, such pointer-system transparency is bought at the cost of greater system complexity. For example, you will learn that the schema requires the **careful** delineation of the model's components.

The network model requires the database administrator to pay close attention to the model's physical environment. In turn, application programmers must take note of the model's physical details.

Basic Network Model Concepts

The network model's end user *perceives* the network database to be **a collection** of records in one-to-many (**1:M**) relationships. But unlike the hierarchical database model, a record in the network database model can have **more than one** parent:



This simple network database model illustrates the basic concepts:

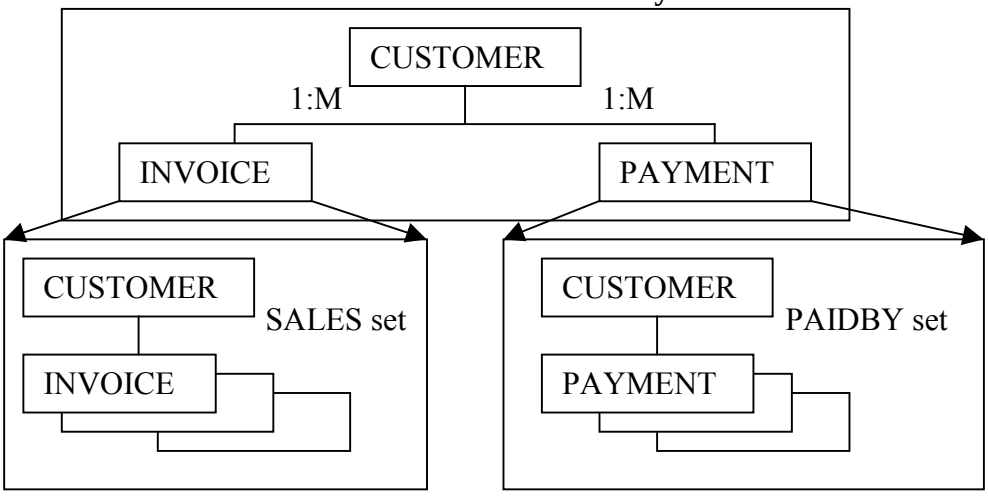
- CUSTOMER, SALESREP, INVOICE, INVLINE, PRODUCT, and PAYMENT represent record types;
- INVOICE is owned by both SALESREP and CUSTOMER, and INVLINE has two owners, PRODUCT and INVOICE.
- The network database model may still include hierarchical one-owner relationships (like CUSTOMER owns PAYMENT).

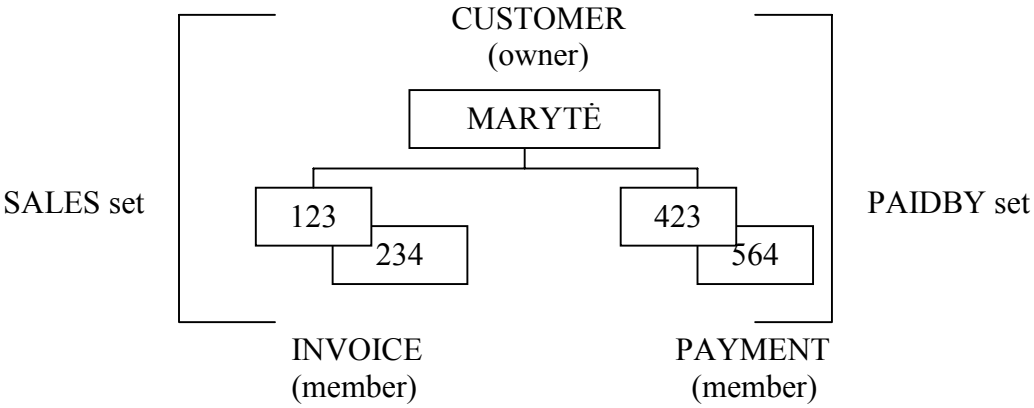
The relationships among records must be decomposed into a series of *sets* before a network database model can be implemented:

- The **SALES** set includes all the INVOICE tickets that belong to a specific CUSTOMER.
- The **PAIDBY** set defines the relationship between CUSTOMER (the owner of the set) and PAYMENT (the member of the set).

In fact, before the network database model can be implemented, all of its data structures must be **decomposed** into sets of 1:M relationships. Each of the sets represents a relationship between owners and members. When implementing a network database design, every set must be **given a name**, and all **owner** and **member** record types must be defined. Given such a structure, the user may navigate through any one of these two sets using the **data manipulation language** (DML) provided by the DBMS.

The INVOICE and PAYMENT records are related to the **specific** CUSTOMER only. When the user accesses another CUSTOMER record, a **different** series of INVOICE and PAYMENT records will be available for that customer. Therefore, network database designers must also be aware of **currency.** The word **currency** is used to indicate the position of the record pointer within the database and **refers to the most recently accessed record.**



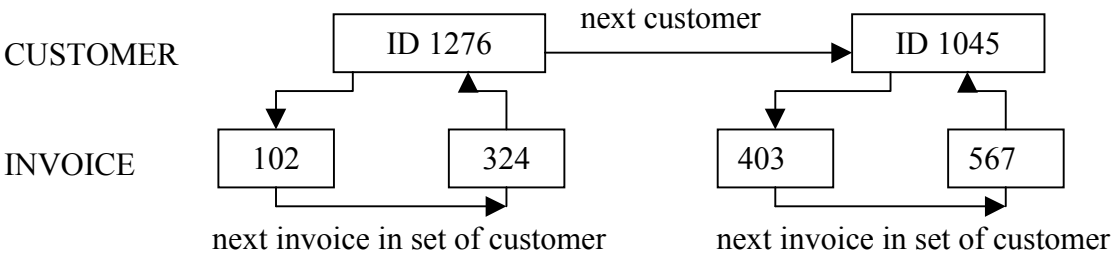


The DBMS automatically updates the pointers after the execution of each operation. A pointer exists for each record type in a set. A pointer's current value refers to a **current record.** Actually, two types of currency pointers exist:

a **record pointer** and a **set pointer.**

Record pointers, also known as **record type pointers,** exist for each record type within the database, **and they always point to the current record within each record type**. Because a set must contain two record types, an owner and a member, the set pointer points to either an owner record or a member record.

There are two occurrences of the SALES set. The first occurrence corresponds to CUSTOMER number 1276. The second occurrence corresponds to CUSTOMER number 1045. Note that the occurrences are determined by the owner of the set: Every time of move to a new CUSTOMER record, a new group of INVOICE member records is made available.



The Database Definition Language (DDL)

The database standardization efforts of the **Data Base Task Group** (DBTG) led to the development of **standard** *data definition language (**DDL**)* specifications. These specifications include DDL instructions that are used to define a network database.

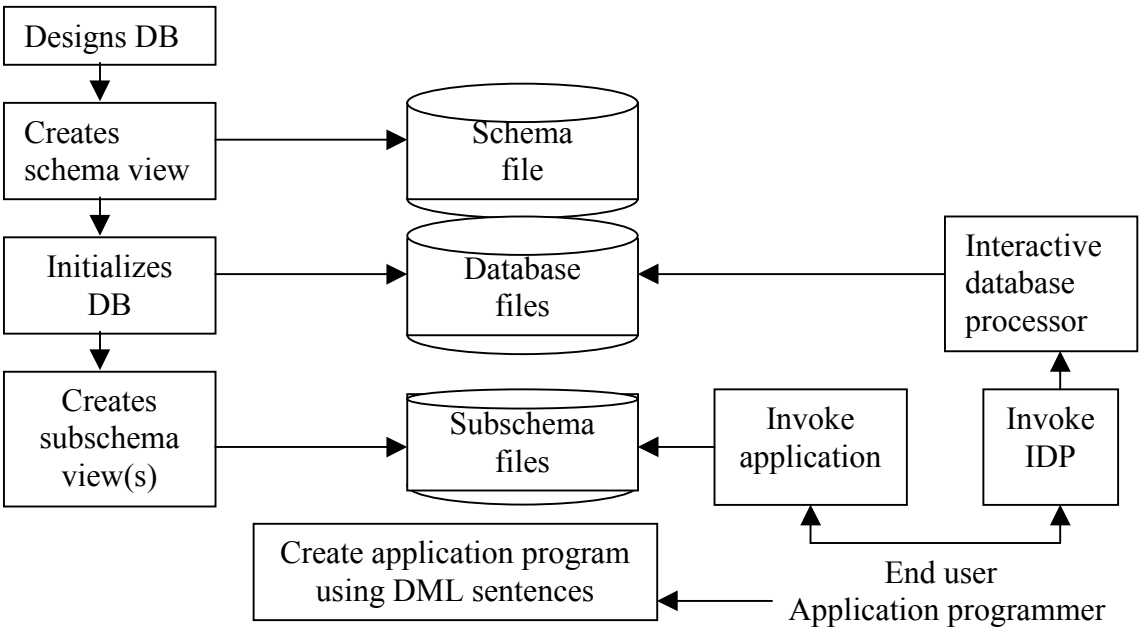
Network database **definition** and **creation** are not interactive processes. Therefore, they must be done through the use of DBMS utility programs at the system prompt level. Creating an **I-D-S/11** (a DBMS of Honeywell Corp.) database requires three steps:

- a **logical definition** of the database using the DDL,
- a physical definition of the database through the use of the DMCL **(device media control language),**
- a **physical creation** of the database storage files on secondary storage.

The network database **schema view** or *schema* describes the entire database as seen by the database administrator. The schema defines the database name, the record type for each of the records, and the field, set, owner, and member records. The database **subschema view** or *subschema* describes the portion of the database used by each of the **application** programs.

The schema view and subschema view(s) are normal text files. These schema and subschema text files may be created with any text-processor program. The files contain DDL and DMCL instructions, which describe the database and application views of the database. Database utility programs must be invoked to validate and create the database structure. Subschema views must be defined and validated for each application that makes use of the database.

I-D-S/11 has an **Interactive Database Processor (IDP)** facility, which allows users to manipulate databases. The IDP front-end is intended for users with some programming knowledge and is not well suited for most end users. The flow diagram for the creation of a network database:



The Schema Definition

The first step in the network database implementation process is the definition of the entire database *as seen by the database administrator* (DBA).

A network database is basically a system driven by pointers. To understand the DDL sequence it is necessary to think of a network database as a system composed of two components: **the data and the pointer structures**. The data (records with fields) are the raw facts kept in permanent storage devices for processing and retrieval. The pointers represent the structure that models the data and describes the required relationships (sets).

More precisely, the pointers define the way relationships are represented among entities. When an application program stores data in the network database, two different **structures** are **updated**: the data and the sets (pointers).

An Explanation Of The Schema Definition

Main schema definition components:

- **Database Administrator Control System (DBACS).** The DBACS is the database definition processor that reads the database definition and validates the schema (the DBACS works like a compiler.)
- **AREA clause**. An AREA is a section of physical storage space that is reserved to store the database. The AREA clause allows the (physical) storage of a database in more than one location, thereby improving access speed. The area name must be unique, and there must be at least one area defined for the database.
- **Record Definitions**
  - **The *RECORD NAME* clause** initiates the record's definition by assigning it a unique name. A valid schema must contain at least one record type.
  - **The *LOCATION MODE* clause** determines where the record will be (physically) stored in the database and how the record will be retrieved. Four location modes are supported:
    - *DIRECT* is the fastest location mode and requires the application program to assign a unique numeric key for each record. Using the DIRECT approach, the user exercises direct control over the arrangement of the records in the database. The DIRECT location mode allows the application program (programmer) to exercise the greatest degree of control over the location and retrieval of records from the database.

- CALC mode uses a hashing algorithm over a record's field to generate the database key for each record. The same algorithm is used to retrieve the records. The DBA indicates the field over which DBMS will apply the algorithm in the schema definition text. This method yields a uniform dispersion of the records across the database.
- VIA *SET* places member records together and near the owner record occurrence in a set. The VIA SET approach is particularly useful when member records will be accessed sequentially.
- INDEXED defines an independent storage structure. Indexed records do not participate in any sets. Instead, indexes are stored in an independent file. A unique primary index is created over a record's field. The index represents the order in which the records are stored in the database. There can be one primary key and several secondary keys for each record.

- **Set Definitions**
  - name the set;
  - identify the OWNER record type;
  - state the SET IS PRIOR PROCESSABLE clause to include a pointer to the previous record, thereby allowing efficient backward processing;
  - state the ORDER clause to specify where the record will be inserted within the set (the *INSERTION* clause can be: *FIRST, LAST, NEXT,* or *PRIOR*);
  - state the MEMBER clause to identify the set's member record type;
  - state the INSERTION and RETENTION clauses to define the way how to associate the member records with their respective owner records.

The network model uses several pointers to create the database's logical structure. The database schema includes pointers to the next record, pointers to the prior record, pointers to the owner record, and so on. The degree of physical detail involved in the definition of the database is also very clear. As a result, learning the intricacies of such a database environment takes a considerable amount of time and effort.

Network database programmers must also be quite familiar with the available storage structures at the physical level. The DBACS not only validates and translates the schema specifications, it also defines and validates the database's physical storage characteristics (the physical characteristics are defined through the use of a device media control language.)

The Device Media Control Language

After defining the database schema, the database administrator (**DBA**) must define the **physical storage** characteristics. The system must 'know' how the database will be stored on disk, what the area name is, and what records and sets belong to the specified area.

The DMCL file contains five components:

1. The schema name.
2. The area name and physical characteristics of the area.
3. The record definitions.
4. The set entry.
5. The key entries, used to name all the record keys found in the area.

Database Initialization

After the schema DDL and DMCL have been validated by the DBACS, the database must be initialized through the use of a utility program. The database initialization process creates the physical files that will contain the database. The database files will be located in the physical storage devices identified in the AREA clause specified in the DDL and DMCL shema files.

Subschema Definition

All applications programs view databases through a *subschema.* The subschema contains all the fields, records, and sets that are available to the application. In effect, the subschema is a 'window' that the **DBCS (Data Base Control System)** opens to the application. The application uses this window to communicate with the database. The subschema is contained within the database's (total) conceptual schema for a database. The DBCS validates all subschema entries against the schema.

When an application program invokes a subschema, a **User Work Area (UWA)** is created by the DBCS. The UWA is a specific area of memory that contains several fields used to access and inform on the status of the database. The UWA also contains space for each record type defined in the subschema.

The LIWA allows the application to communicate with the DBCS. Application programs read from and write to the UWA when the database is accessed or updated. Application programs can also check the database status after each operation to see if the operation was performed properly.

When an application retrieves a database record, the DBCS reads that record and places it in the space reserved for it by the application program's LRWA. The DBCS also updates all the required LTWA status fields. The application can also check and validate the database status after its last operation.

Subschemas are created manually by the DBA. In this case, the DBA must assure that all the subschema's definitions are correct and valid to the schema. A better way to create subschemas is to use the DBACS to create a full subschema from the main schema. Such a subschema will allow an unconstrained manipulation of the entire database. This all-encompassing subschema can then be modified by erasing all the fields, records, and relations not required by the application program. The DBMS can generate subschemas for APL, BASIC, COBOL, and FORTRAN.

The subschema contains three main components:

1. The *Title Division,* containing schema and subschema names.
2. The *Mapping Division,* containing all aliases used in the subschema.
3. The *Structure Division,* in which the area, sets, keys, and records used by the application are defined.

An Introduction To The Data Manipulation Language

Application programs navigate in a database by using a *data manipulation language* (DML). The DBMS provides interfaces to four languages: APL, BASIC, COBOL, and FORTRAN.

The UWA (User Work Area) has eight special status registers. These registers are used by the DBCS and the application program to share information about the status of the database:

- DB-STATUS. Returns the status of the DML statement after its execution.
- DB-REALM-NAME. Returns the name of the realm at the conclusion of DML sentences.
- DB-SET-NAME. Returns the set name after an unsuccessful DML statement.
- DB-RECORD. Returns the record name at the conclusion of DML statements
- DB-PRIVACY-KEY. The DBCS places the value of the PRIVACY key in this register during the schema translation. The PRIVACY key is a keyword used to restrict access to the database's authorized users.

- DIRECT-REFERENCE. This register is used to pass on a record key for DIRECT access. The DBCS updates this register with the value of the key for the current record in the last DML statement.
- DB-DATA-NAME. If the subschema was translated with the INCLUDE DATA NAMES option, the DBCS returns the DATA-ITEM name when an invaliddatatype problem occurs.
- DB-KEY-NAME. The DBCS returns the name of the record key at the conclusion of an unsuccessful DML statement.

In order to keep track of the record and set'pointers, **I-D-S/11** keeps five currency register records in the LIWA:

1. *Current record of the run* unit. The DBCS updates this pointer after certain DML statements. This is the pointer to the last valid record accessed by the application.
2. *Current record of a set type.* A pointer for each set defined in the subschema. Such pointers specify the last record in each set that was accessed by the application.
3. *Current record of a* realm. A pointer for each realm specified in the subschema. A database can be stored in one or more areas or realms (physical files).
4. *Current record of a record type.* A pointer for each of the subschema's record types.
5. *Current record of a record key type.* A pointer for each record key type defined in the subschema. Each record key type points to a specific record. DBCS keeps a pointer to the last record accessed for each of the defined record key types.

Data manipulation language commands:

- **Opening Realms.** An application program must invoke the READY command in order to access a database. The READY command makes the database available to the program. Three usage modes are available: UPDATE-Read/write to the database; LOAD-Initial load of the database; RETRIEVAL-Read from the database.
- **Closing Realms.** When the database's use is no longer required, close the realm using the syntax FINISH (realm list). The realm list refers to the realm names that make up a database.
- **STORE**. The STORE command saves a database record and updates the current record of the run unit in the UWA.
- **FIND.** The FIND command is used to locate records in the database and works with the LOCATION MODE used in the schema definition. The FIND command updates the currency values of the UWA. The syntax for the FIND command varies according to the access type.
  - A Direct Access Mode.
  - Calc Access Mode.
  - Navigating Within Sets.
  - Locating Owner Records.
- **CONNECT.** The purpose of the CONNECT command is to insert an existing record as a set member. Both the member and the owner records must already be stored in the database. The CONNECT command is used when the INSERTION IS MANUAL and the OWNER IDENTIFIED BY APPLICATION clauses were specified for the member record in the schema definition. The user has to manually CONNECT the record with the appropriate owner record in each of the sets that the record belongs to.
- **DISCONNECT.** The DISCONNECT command removes a record from a set. The command is used only when records were declared as AUTOMATIC OPTIONAL or MANUAL OPTIONAL members of a set in the schema definition.
- **GET.** The GET command reads a record from the database, making the record's fields available to the program. Only the fields defined in the subschema are available to the program.
- **MODIFY.** The MODIFY command changes the current record's field contents.
- **ERASE**. The ERASE command removes the current record from the database and automatically removes all member records associated with it.

Network Models Contribution To Database Systems

The network database model provided several advantages over its file-system and hierarchical database predecessors. In fact, the network database model paved the way for subsequent database developments through CODASYL's attempt to standardize basic database concepts such as schema, subschema, and DML. The network database model also set the stage for more complex and better data modeling by providing support for relations in which a record could be related to more than one owner or parent record.

Yet, despite its many contributions and subsequent enhancements, network database implementations did not enjoy longevity. The network database model failed to gain dominance for several reasons:

1. It was still tied to the physical level, thus continuing the requirement for complicated design and implementation details.
2. Its adherents could not overcome the marketing clout and the large installed base of IBM's IMS hierarchical DBMS.
3. Its most important appeal, its ability to provide support for more complex relations, lost out to the much 'friendlier' relational database model.

Despite the fact that the network database model has largely yielded to the relational database model, quite a few network installations continue to exist in big corporations. Its survival is at least partly due to the fact that COBOL is still the dominant business-oriented programming language in many of those corporations. The coexistence of COBOL and network databases is not surprising if to remember that the CODASYL group was responsible for both the creation of a standard COBOL language and the creation of the network model's database standards.