SYLLABUS

DATABASE MANAGEMENT SYSTEM

SECTION A

1. Introduction

Purpose of database, data abstraction, data models, instances & schemas, data independence, data definition language, data manipulation language, database manager, database administration.

SECTION B

2. Entity Relationship Model

Entity & Entity sets, relationship sets, mapping constraints, candidate & primary key, entity relationship diagram, reducing E-R diagram to tables.

3. Relational Model

Concepts of relational model, integrity constraints, extension & intension, relational algebra, relational calculus, commercial query language, modifying the database, comments on relational model.

4. DBMS based on Relational Model

Introduction, the mapping operation, data manipulation facility, data definition facility, data control facility.

SECTION C

5. Normalization

Introduction to functional dependence, normalization-1NF, 2NF, 3NF, BCNF, 4NF, 5NF.

SECTION D

6. Oracle Ingress or Sybase

Creation of tables, modification of tables, DDL command for RDBMS, SQL command for RDBMS, command language.

SECTION A

1. Introduction of Database

CHAPTER 1 INTRODUCTION OF DATABASE

NOTES

LEARNING OBJECTIVES

- Basic Concepts
- DBMS (Database Management System)
- Purpose of Database
- Data Entry Forms—For Enter the Data or Record
- Database Environment
- Advantages of the Database Approach
- Disadvantages of DBMS
- File System vs DBMS
- Drawbacks of File-Based Systems
- Data Abstraction
- Data Models, Schemas and Instances
- Hierarchical Model
- Network Model
- Relational Model Structure
- Object-Oriented Model
- Object-Relational Model
- Deductive/Inference Model
- Comparison among the Various Database Models
- Three Schema Architecture and Data Independence
- Data Independence
- DBMS Languages
- DBMS Interfaces
- Database Users

BASIC CONCEPTS

Data Vs Information

Data

Data can be defined as raw facts and figures that have meaning like:

(1) List of customers

(2) Marks of students in different subjects

(3) Number of Units consumed.

Information

When the data is processed to achieve any meaningful result, it is known as information

Examples

(1) Invoice of Customer

(2) Mark Sheet of students

(3) Electricity Bill

Accurate, relevant, and timely information is key to good decision making.

Value of Information

NOTES

Information is used for strategic planning, to perform an action, to see the event. For effective information data must be processed in the form it is required and must be processed quickly.

File

A file is a collection of interrelated information or records. You can create different types of files your computer system, according to the usage. There can be different files for different software's and for different uses, like text files, command files and many others.

Record

Record is a group of related data items treated as a unit by an application program

Example

ID	Name	Pet Name	Pet Type
132	Jill Smith	Tuffy	Cat

What is a Database?

A database is a well-organized collection of data that are related in a meaningful way, which can be accessed in different logical orders but are stored only once. The data in the database is therefore integrated, structured and shared.

Note that not every collection of related data can be considered as a database. A database usually satisfies the following properties:

A database represents some aspect of the real word.

A database is a logically coherent collection of data with some inherent meaning.

A database is designed, built and populated with data for a specific purpose.

A database may be generated and maintained manually or it may be computerized.

Example

A simple university database maintaining information about students, courses and grade in a university environment.

DBMS (DATABASE MANAGEMENT SYSTEM)

The database management system (DBMS) is the software that:

- handles all access to the database
- is responsible for applying the authorization checks and validation procedures

The DBMS is an intermediate link between the physical database, the computer and the operating system, and on the other hand, the users. The main objective of a DBMS is to provide a convenient and efficient environment to retrieve and store database information. Database systems can support single user or multi-user environment.

NOTES

Characteristics of database approach

1. **Self-Description:** A database system includes—in addition to the data stored that is of relevance to the organization—a complete definition/description of the database's structure and constraints. This meta-data (*i.e.*, data about data) is stored in the so-called system catalog, which contains a description of the structure of each file, the type and storage format of each field, and the various constraints on the data (*i.e.*, conditions that the data must satisfy).
2. **Insulation between Programs and Data; Data Abstraction Program-Data Independence:** In traditional file processing, the structure of the data files accessed by an application is "hard-coded" in its source code. (*e.g.*, Consider a file descriptor in a COBOL program: it gives a detailed description of the layout of the records in a file.)

If, for some reason, we decide to change the structure of the data (*e.g.*, by adding the first two digits to the YEAR field!), every application in which a description of that file's structure is hard-coded must be changed!

In contrast, DBMS access programs, in most cases, do not require such changes, because the structure of the data is described (in the system catalog) separately from the programs that access it and those programs consult the catalog in order to ascertain the structure of the data (i.e., providing a means by which to determine boundaries between records and between fields within records) so that they interpret that data properly.

In other words, the DBMS provides a conceptual or logical view of the data to application programs, so that the underlying implementation may be changed without the programs being modified. (This is referred to as program-data independence.)

Also, which access paths (*e.g.*, indexes) exist are listed in the catalog, helping the DBMS to determine the most efficient way to search for items in response to a query.

3. **Multiple Views of Data:** Different users (*e.g.*, in different departments of an organization) have different "views" or perspectives on the database. For example, from the point of view of a Bursar's Office employee, student data does not include anything about which courses were taken or which grades were earned. (This is an example of a subset view.)

As another example, a Registrar's Office employee might think that GPA is a field of data in each student's record. In reality, the underlying database might calculate that value each time it is called for. This is called virtual (or derived) data.

A view designed for an academic advisor might give the appearance that the data is structured to point out the prerequisites of each course.

A good DBMS has facilities for defining multiple views. This is not only convenient for users, but also addresses security issues of data access. (*e.g.*, The Registrar's Office view should not provide any means to access financial data.)

4. **Data Sharing and Multi-user Transaction Processing:** As you learned about in the OS course, the simultaneous access of computer resources by multiple users/processes is a major source of complexity. The same is true for multi-user DBMS's.

Arising from this is the need for concurrency control, which is supposed to ensure that several users trying to update the same data do so in a "controlled" manner so

NOTES

that the results of the updates are as though they were done in some sequential order (rather than interleaved, which could result in data being incorrect).

This gives rise to the concept of a transaction, which is a process that makes one or more accesses to a database and which must have the appearance of executing in isolation from all other transactions (even ones that access the same data at the "same time") and of being atomic (in the sense that, if the system crashes in the middle of its execution, the database contents must be as though it did not execute at all).

Applications such as airline reservation systems are known as online transaction processing applications.

Basically a Database Systems consists of two parts -

Database Management System

Database Application

Database Management System is the program that organizes and maintains the information.

Database Application is the program that is used to view, retrieve and update information stored in the database.

DBMS offers the following services.

(*i*) **Data Definition**—It is a Method of Defining data and storage.

(*ii*) **Data Maintenance**—It checks whether each record has field containing all information for a particular record or not. For example in an employee table, all information about the employee like employee id, name, designation, salary, dept are recorded.

(*iii*) **Data Manipulation**—It allows data in the Database to be inserted, Updated, and Deleted and sorted.

(*iv*) **Data Display**—It helps in viewing data.

(*v*) **Data Integrity**—It ensures the accuracy of the data.

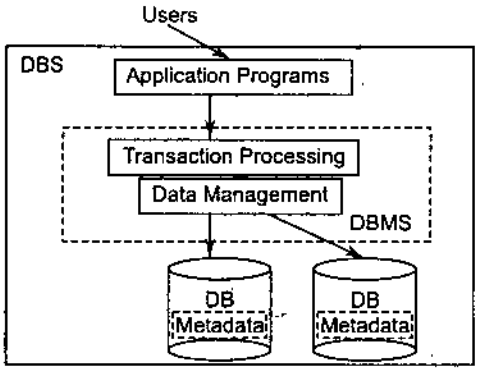


Figure 1. DBMS

An Example of Database

Customer ID	Name	Address	City	State	Acct_Number	Balance
121	Mr. Smith	123 Lexington	Smithville	KY	9987	4000
122	Mr. Jim	123 Lexington	Smithville	KY	9980	2000
123	Mrs. Jones	12 Davis Ave.	Smithville	KY	8811	1000

NOTES

Brief History of Database Systems

- 1940's, 50's Initial use of computers as calculators. Limited data, focus on algorithms. Science, military applications.
- 1960's Business uses. Organizational data, customer data, sales, inventory, accounting, etc. File system based, high emphasis on applications programs to extract and assimilate data. Larger amounts of data, relatively simple calculations.
- 1970's The relational model. Data separated into individual tables. Related by keys. Initially required heavy system resources. Examples: Oracle, Sybase, Informix, Digital RDB, IBM DB2.
- 1980's Microcomputers—the IBM PC, Apple Macintosh. Database program such as DBase (sort of), Paradox, FoxPro, MS Access. Individual user can create, maintain small databases.
- Late- 1980's Local area networks. Workgroups sharing resources such as files, printers, e-mail. Client/Server Database resides on a central server, applications programs run on client PCs attached to the server over a LAN.
- 1990's Internet and World Wide Web make databases of all kinds available from a single type of client—the Web Browser. Data warehousing and Data Mining also emerge.

Applications of a DBMS

Databases are widely used. Here are some representative applications:

- **Banking:** For customer information, accounts, and loans, and banking transactions.
- **Airlines:** For reservations and schedule information. Airlines were among the first to use databases in a geographically distributed manner—terminals situated around the world accessed the central database system through phone lines and other data networks.
- **Universities:** For student information, course registrations, and grades.
- **Credit Card Transactions:** For purchases on credit cards and generation of monthly statements.
- **Telecommunication:** For keeping records of calls made, generating monthly bills, maintaining balance on prepaid calling cards, and storing information about the communication networks.
- **Finance:** For storing information about holdings, sales and purchases of financial instruments such as stocks and bonds.
- **Sales:** For customers, product, and purchase information.
- **Manufacturing:** For management of supply chain and for tracking production of items in factories, inventories of items in warehouses / stores and orders for items.
- **Human Resources:** For information about employees, salaries, payroll taxes and benefits, and for generation of paychecks.

PURPOSE OF DATABASE

The functions performed by a typical DBMS are the following:

- **Data Definition :** The DBMS provides functions to define the structure of the data in the application. These include defining and modifying the record

NOTES

structure, the type and size of fields and the various constraints/conditions to be satisfied by the data in each field.

- **Data Manipulation:** Once the data structure is defined, data needs to be inserted, modified or deleted. The functions which perform these operations are also part of the DBMS. These function can handle planned and unplanned data manipulation needs. Planned queries are those which form part of the application. Unplanned queries are ad-hoc queries which are performed on a need basis.
- **Data Security and Integrity:** The DBMS contains functions which handle the security and integrity of data in the application. These can be easily invoked by the application and hence the application programmer need not code these functions in his/her programs.
- **Data Recovery and Concurrency:** Recovery of data after a system failure and concurrent access of records by multiple users are also handled by the DBMS.
- **Data Dictionary Maintenance:** Maintaining the Data Dictionary which contains the data definition of the application is also one of the functions of a DBMS.
- **Performance:** Optimizing the performance of the queries is one of the important functions of a DBMS. Hence the DBMS has a set of programs forming the Query Optimizer which evaluates the different implementations of a query and chooses the best among them.

Thus the DBMS provides an environment that is both convenient and efficient to use when there is a large volume of data and many transactions to be processed.

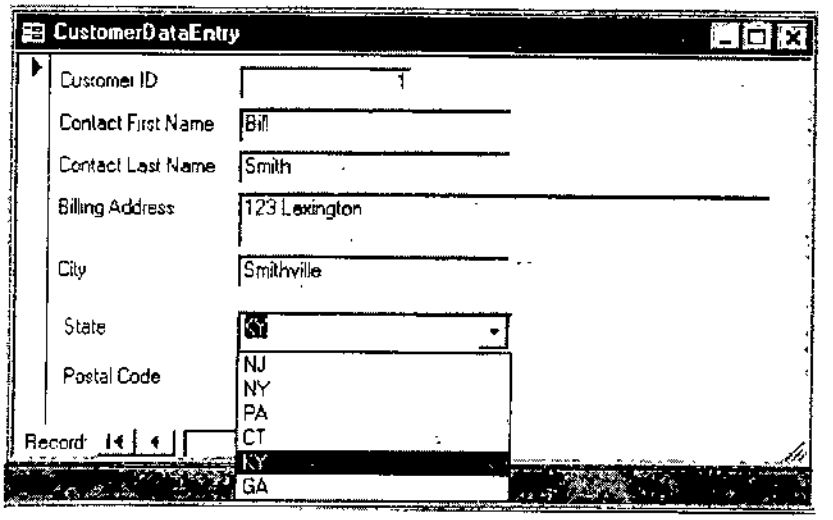
Database Application Components

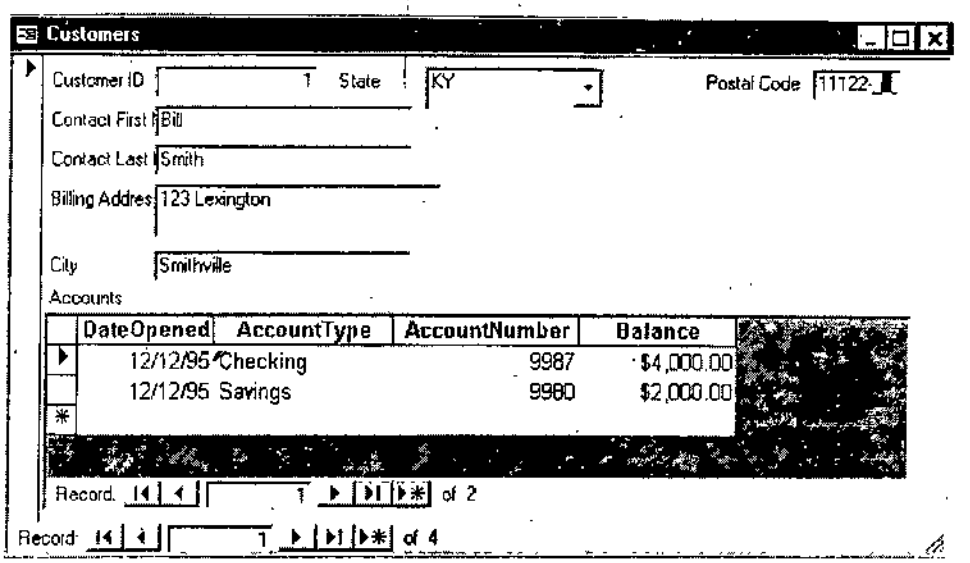
We discussed the various components of DBMS and databases. We now turn our attention to the applications or the main purpose of DBMS that are used to access databases.

DATA ENTRY FORMS—FOR ENTER THE DATA OR RECORD

- A primary means to enter data into a database and to edit existing data.
- Can also be used to query (Query By Example).
  A data entry form would have fields that correspond to each of the database columns. For example, a Customer data entry form would have fields for: Customer_Id, Name, Street, City, State, Zip.
- With Graphical User Interfaces, more efficient data entry can be affected.
  - o List boxes—provide a list of valid values for a user to choose from. Example: List of US States.
  - o Radio Buttons—Exclusive list of options. Example: Gender M/F
  - o Check Boxes—Non-Exclusive list of options.
- Other options to constrain user input:
  - o Convert input to all upper case or all lowercase
  - o Restrict the number of digits entered
  - o Check for valid numbers and other values
- A typical database application will have roughly one form for each table.
- Also information-only forms (Query-only). Not used for updating or creating new data.

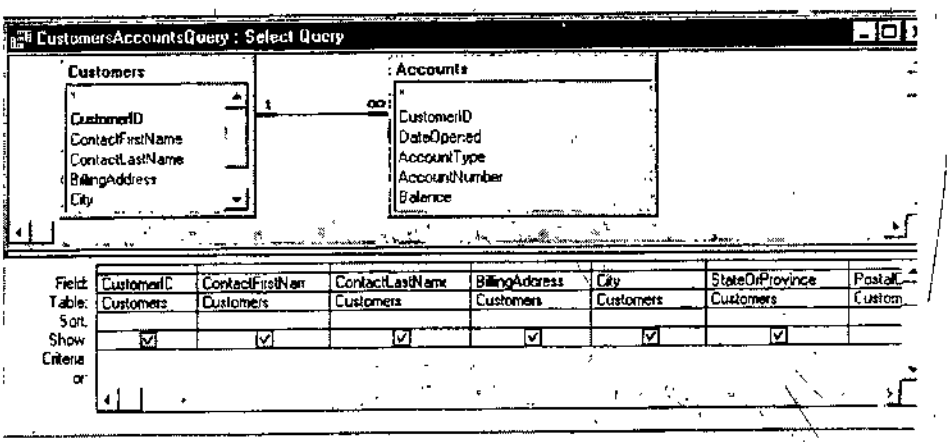
NOTES





Queries—To get the answers of database related questions

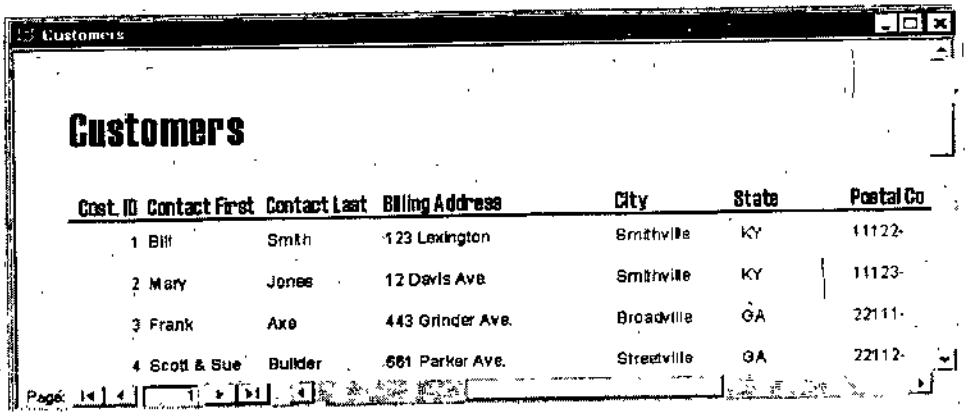
- Common Queries to the database can be formed by the database designers.
- Save queries for specific purposes.
- User supplies criteria for the query and executes the query against the tables in the database.



NOTES

Reports—To display the final output of database

- Primarily used to convey large portions of data in the database.
- Output can be specially formatted for a variety of purposes such as printing mailing labels.

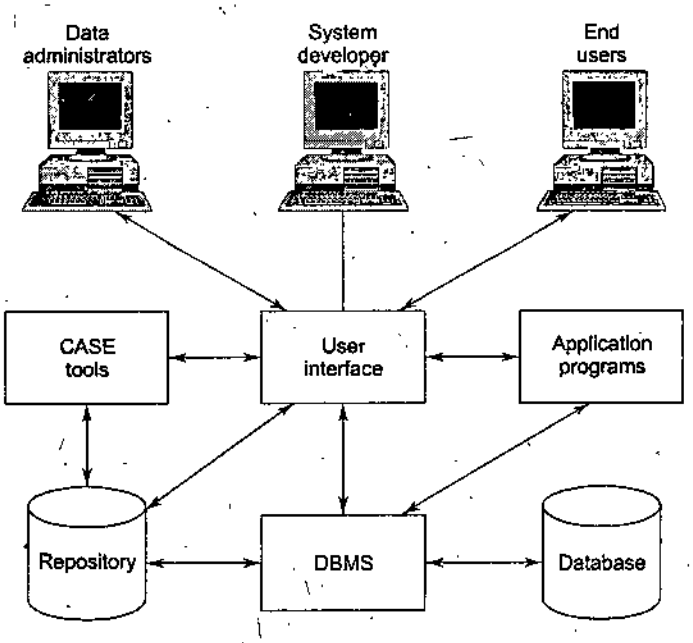


WWW Applications—To access the information on Internet

- Users access database through a WWW Browser.
- Data entry forms can be filled out and submitted to be saved in the database.
- Reports can be formatted and displayed as web pages.
- Menus are simply links to different web pages for forms and reports.
- All applications Code typically resides on the WWW server.

DATABASE ENVIRONMENT

A **database environment,** such as the one shown below, became quite sophisticated and user friendly during the last decade. It enables users to query the database, developers to create new applications and the DBA to manage the database.



The 'User Interface' represent the menus, languages and 'GUI' (Graphical User Interface).

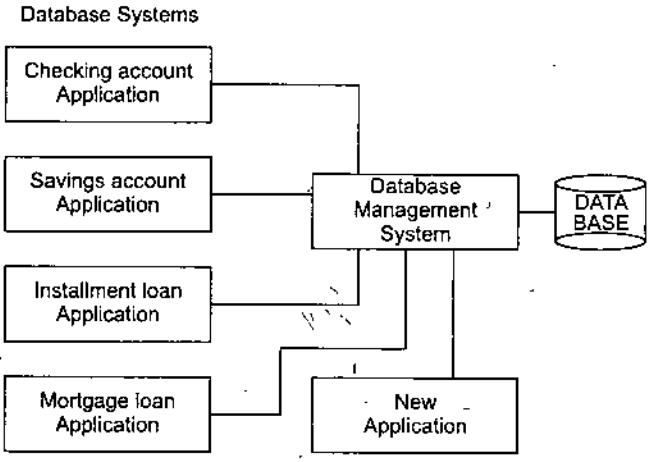
CASE (Computer Aided Software Engineering) tools are automated tools for the analysis, design and development of the database and application programs.

A database **repository** is a knowledge database for storing information about the enterprise database, such as all data definitions, relationships, screen and report formats.

NOTES

The **'DBMS'** is typically the commercial software such as Microsoft's ACCESS, Oracle's Oracle, and IBM's DB2 for managing the database. And the database symbol represents the disk storage where the entire data is stored.

ADVANTAGES OF THE DATABASE APPROACH



As shown in the figure, the DBMS is a central system which provides a common interface between the data and the various front-end programs in the application. It also provides a central location for the whole data in the application to reside.

1. Controlling Redundancy Since the whole data resides in one central database, the various programs in the application can access data in different data files. Hence data present in one file need not be duplicated in another. This reduces data redundancy. However, this does not mean all redundancy can be eliminated. There could be business or technical reasons for having some amount of redundancy. Any such redundancy should be carefully controlled and the DBMS should be aware of it.

In traditional file processing, every user group maintains its own files. Each group independently keeps files on their db *e.g.*, students. Therefore, much of the data is stored twice or more. Redundancy leads to several problems :

- Duplication of effort
- Storage space wasted when the same data is stored repeatedly
- Files that represent the same data may become inconsistent (since the updates are applied independently by each users group).

2. Restricting Unauthorized Access

A DBMS should provide a security and authorization subsystem.

- Some db users will not be authorized to access all information in the db (*e.g.*, financial data).

NOTES

- Some users are allowed only to retrieve data.
- Some users are allowed both to retrieve and to update database.

3. **Data Sharing:** Related data can be shared across programs since the data is stored in a centralized manner. Even new applications can be developed to operate against the same data.
4. **Enforcement of Standards:** Enforcing standards in the organization and structure of data files is required and also easy in a Database System, since it is one single set of programs which is always interacting with the data files.
5. **Application Development Ease :** The application programmer need not build the functions for handling issues like concurrent access, security, data integrity, etc. The programmer only needs to implement the application business rules. This brings in application development ease. Adding additional functional modules is also easier than in file-based systems.
6. **Better Controls:** Better controls can be achieved due to the centralized nature of the system.
7. **Providing Persistent Storage for Program Objects:** Object-oriented database systems make it easier for complex runtime objects (*e.g.*, lists, trees) to be saved in secondary storage so as to survive beyond program termination and to be retrievable at a later time.
8. **Providing Storage Structures for Efficient Query Processing:** The DBMS maintains indexes (typically in the form of trees and/or hash tables) that are utilized to improve the execution time of queries and updates. (The choice of which indexes to create and maintain is part of physical database design and tuning and is the responsibility of the DBA.

   The query processing and optimization module is responsible for choosing an efficient query execution plan for each query submitted to the system.
9. **Providing Backup and Recovery:** The subsystem having this responsibility ensures that recovery is possible in the case of a system crash during execution of one or more transactions.
10. **Providing Multiple User Interfaces:** For example, query languages for casual users, programming language interfaces for application programmers, forms and/or command codes for parametric users, menu-driven interfaces for stand-alone users.
11. **Representing Complex Relationships Among Data:** A DBMS should have the capability to represent such relationships and to retrieve related data quickly.
12. **Enforcing Integrity Constraints:** Most database applications are such that the semantics (*i.e.*, meaning) of the data require that it satisfy certain restrictions in order to make sense. Perhaps the most fundamental constraint on a data item is its data type, which specifies the universe of values from which its value may be drawn. (*e.g.*, a Grade field could be defined to be of type Grade_Type, which, say, we have defined as including precisely the values in the set { "A", "A-", "B+", ..., "F" }.

    Another kind of constraint is referential integrity, which says that if the database includes an entity that refers to another one, the latter entity must exist in the database. For example, if (R56547, CIL102) is a tuple in the Enrolled_In relation, indicating that a student with ID R56547 is taking a course with ID CIL102, there must be a tuple in the Student relation corresponding to a student with that ID.

13. **Permitting Inferencing and Actions Via Rules:** In a deductive database system, one may specify declarative rules that allow the database to infer new data! *e.g.*, Figure out which students are on academic probation. Such capabilities would take the place of application programs that would be used to ascertain such information otherwise.
14. **Reduced Maintenance:** Maintenance is less and easy, again, due to the centralized nature of the system.

NOTES

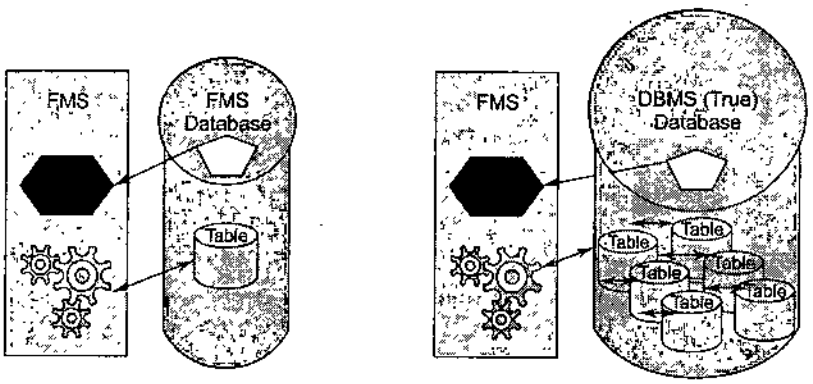
DISADVANTAGES OF DBMS

If databases technology is so powerful, why do will still find many companies using the old file processing system?

- Cost of purchasing the database software (DBMS) is high. Depending on the system platform and number of user, it could be a six digits figure
- Conversion of the old systems (referred to as legacy systems) from COBOL to SQL is expensive
- The usage of DBMS requires sophisticated computer professionals
- Organizational Conflict (politics): It required management commitment, budgets, agreements among participants

FILE SYSTEM VS DBMS

A Database Management System (DBMS) is a combination of computer software, hardware, and information designed to electronically manipulate data via computer processing. Two types of database management systems are DBMS's and FMS's. In simple terms, a File Management System (FMS) is a Database Management System that allows access to single files or tables at a time. FMS's accommodate flat files that have no relation to other files. The FMS was the predecessor for the Database Management System (DBMS), which allows access to multiple files or tables at a time



Example—Bank Example: Consider a new customer, Joe Smith, opening a savings account. The personal data and the account information are entered into the Savings file. At a later time, Joe opens a money market account with the Money Market Department. Same information is entered. Finally, same information is entered again by the Loan Department where Joe obtains a car loan. These files are shown as follows:

NOTES

Savings File in the Savings Department

Account Number	Depositor Name	SSN	Address	Phone	Deposit$ amount
S-100	Smith, Joe	111-11-1111	G-452, Chandini Estate	(215) 204-1234	$1.000
S-101	Doe, Jones	222-11-1234	G-456, Alistonia Estate	(215) 204-1237	$5.000

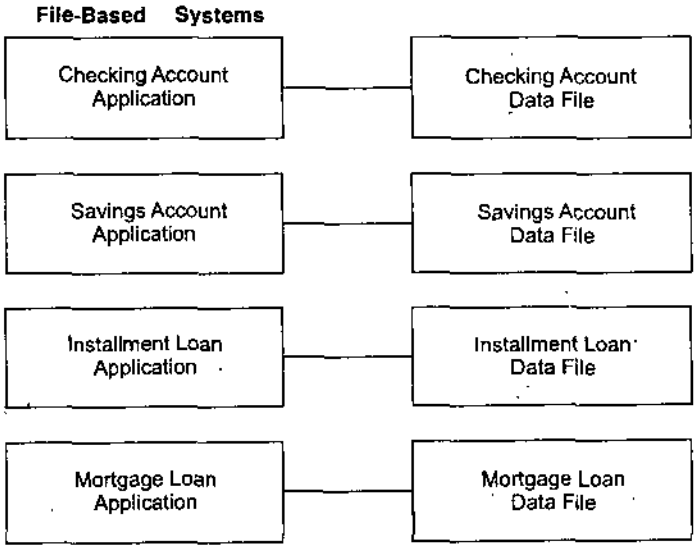
Money Market File in the MM Department

Account Number	Depositor Name	SSN	Address	Phone	Deposit$ amount
S-100	Smith, Joe	111-11-1111	G-452, Chandini Estate	(215) 204-1234	$10.000

Loan File in the Loan Department

Account Number	Depositor Name	SSN	Address	Phone	Lone amount
LD-123	Smith, Joe	111-11-1111	G-452, Chandini Estate	(215) 204-1234	$2500

DRAWBACKS OF FILE-BASED SYSTEMS



As shown in the figure, in a file-based system, different programs in the same application may be interacting with different private data files. There is no system enforcing any standardized control on the organization and structure of these data files.

- **Data Redundancy and Inconsistency:** Since data resides in different private data files, there are chances of redundancy and resulting inconsistency. For example, in the above example shown, the same customer can have a savings account as well as a mortgage loan. Here the customer details may be duplicated since the programs for the two functions store their corresponding data in two different data files. This gives rise to redundancy in the customer's data. Since the same data is stored in two files, incon-

sistency arises if a change made in the data in one file is not reflected in the other.

- **Unanticipated Queries :** In a file-based system, handling sudden/ad-hoc queries can be difficult, since it requires changes in the existing programs.
- **Data Isolation :** Though data used by different programs in the application may be related, they reside in isolated data files.
- **Concurrent Access Anomalies :** In large multi-user systems the same file or record may need to be accessed by multiple users simultaneously. Handling this in a file-based systems is difficult.
- **Security Problems :** In data-intensive applications, security of data is a major concern. Users should be given access only to required data and not the whole database. In a file-based system, this can be handled only by additional programming in each application.
- **Integrity Problems :** In any application, there will be certain data integrity rule which needs to be maintained. These could be in the form of certain conditions/constraints on the elements of the data records. In the savings bank application, one such integrity rule could be "Customer ID, which is the unique identifier for a customer record, should be non-empty". There can be several such integrity rules. In a file-based system, all these rules need to be explicitly programmed in the application program.

It may be noted that, we are not trying to say that handling the above issues like concurrent access, security, integrity problems, etc., is not possible in a file-based system. The real issue was that, though all these are common issues of concern to any data-intensive application, each application had to handle all these problems on its own. The application programmer needs to bother not only about implementing the application business rules but also about handling these common issues.

- **Difficulty in accessing data :** Suppose that one of the bank officers needs to find out the names of all customers who live within a particular postal-code area. The officer asks the data-processing department to generate such a list. Because the designers of the original system did not anticipate this request, there is no application program on hand to meet it. There is, however, an application program to generate the list of all customers. The bank officer has now two choices: either obtain the list of all customers and extract the needed information manually or ask a system programmer to write the necessary application program. Both alternatives are obviously unsatisfactory. Suppose that such a program is written, and that several days later, the same officer needs to trim that list to include only those customers who have an account balance of 10000 or more. As expected, a program to generate such a list does not exist. Again, the officer has the preceding two options, neither of which is satisfactory.

DATA ABSTRACTION

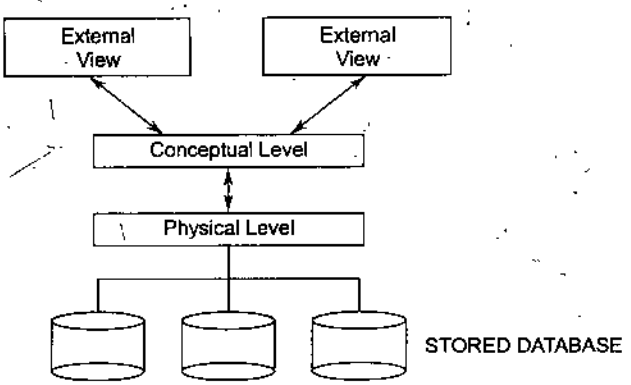
The major purpose of a database system is to provide users with an abstract view of the system.

The system hides certain details of how data is stored and created and maintained

Complexity should be hidden from database users.

There are several levels of abstraction:

NOTES



1. **Physical Level:**
   - How the data are stored.
   - *e.g.*, index, B-tree, hashing.
   - Lowest level of abstraction.
   - Complex low-level structures described in detail.
2. **Conceptual Level:**
   - Next highest level of abstraction.
   - Describes what data are stored.
   - Describes the relationships among data.
   - Database administrator level.
3. **View Level:**
   - Highest level.
   - Describes part of the database for a particular group of users.
   - Can be many different views of a database.
   - *e.g.*, tellers in a bank get a view of customer accounts, but not of payroll data.

STUDENT ACTIVITY-1

1. What are the elements of a database?

2. What do you mean by data?

3. What are relationships in a database?

4. What are database constraints?

DATA MODELS, SCHEMAS AND INSTANCES

A characteristic of the database approach is that it provides a level of data abstraction, by hiding details of data storage that are not needed by most users.

A data model is a collection of concepts that can be used to describe the structure of a database. The model provides the necessary means to achieve the abstraction.

The structure of a database is characterized by data types, relationships, and constraints that hold for the data. Models also include a set of operations for specifying retrievals and updates.

NOTES

Data models are changing to include concepts to specify the behaviour of the database application. This allows designers to specify a set of user defined operations that are allowed.

Categories of Data Models

Data models can be categorized in multiple ways.

- **High level/conceptual data models** - provide concepts close to the way users perceive the data.
- **Physical data models** - provide concepts that describe the details of how data is stored in the computer. These concepts are generally meant for the specialist, and not the end user.
- **Representational data models** - provide concepts that may be understood by the end user but not far removed from the way data is organized.

Conceptual data models use concepts such as entities, attributes and relationships.

- **Entity** - represents a real world object or concept
- **Attribute** - represents property of interest that describes an entity, such as name or salary.
- **Relationships** - among two or more entities, represents an association among two or more entities.

Representational data models are used most frequently in commercial DBMSs. They include relational data models, and legacy models such as network and hierarchical models.

Physical data models describe how data is stored in files by representing record formats, record orderings and access paths.

Object data models - a group of higher level implementation data models closer to conceptual data models.

Section (B) Q-5

HIERARCHICAL MODEL

Hierarchical Database model is one of the oldest database models, dating from late 1950s. One of the first hierarchical databases—Information Management System (IMS)—was developed jointly by North American Rockwell Company and IBM. IMS became the world's leading mainframe hierarchical database system in the 1970s and early 1980s. The hierarchical model assumes that a tree structure is the most frequently occurring relationship. This assumption is recognized today as misleading. In fact many of the limitations and shortcomings of the hierarchical model result from this overly restrictive view of relationships.

The hierarchical model organizes data elements as tabular rows, one for each instance of an entity. Consider a company's organizational structure. At the top

NOTES

we have a General Manager (CM). Under him we have several Deputy General Managers (DGMs). Each DGM looks after a couple of departments and each department will have a manager and many employees. When represented in hierarchical model, there will be separate rows for representing the GM, each DGM, each department, each Manager and each Employee. The row position implies a relationship to other rows. A given employee belongs to the department that is closest above it in the list and the department belongs to the manager that is immediately above it in the list and so on.

The hierarchical model represents relationships with the notion of 'logical adjacency' or more accurately with 'logical proximity' in a linearized tree. You can locate a set of employees working for say. Manager X by first locating Manager X and then including every employee in the list after X and before the next occurrence of a manager or the end of the list. Because linearized tree is an abstraction, the term logical proximity is more appropriate for hierarchical model.

Advantages

The hierarchical model had many advantages over the file systems it replaced. It can be said that the advantages and features of the hierarchical database systems was the reason for the development of the database models that followed it. The main advantages of this database model are:

- **Simplicity.** Since the database is based on the hierarchical structure, the relationship between the various layers is logically (conceptually) simple. Thus the design of a hierarchical database is simple.
- **Data Security.** Hierarchical model was the first database model that offered the data security that is provided and enforced by the DBMS.
- **Data Integrity.** Since the hierarchical model is based on the parent/child relationship there is always a line between the parent segment and the child segmenty under it. The child segments are always automatically referenced to its parent, this model promotes data integrity.
- **Efficiency.** The hierarchical database model is a very efficient one when the database contains a large number of l:n relationships (one-to-many relationships) and when the users require large number of transactions, using data whose relationships are fixed.

Disadvantages

The main disadvantages of the hierarchical database model are:

- **Implementation Complexity.** Although the hierarchical database model is conceptually simple and easy to design it is quite complex to implement. The database designers should have very good knowledge of the physical data storage characteristics.
- **Database Management Problems.** If you make any changes in the database structure of a hierarchical database, then you need to make the necessary changes in all the application programs that access the database. Thus maintaining the database and the applications can become very difficult.
- **Lack of structural independence.** Structural independence exists when the changes to the database structure does not affect the DBMS's ability to access data. Hierarchical database systems use physical storage paths to navigate to the different data segments. So the application programmer

should have a good knowledge of the relevant access paths to access the data. So if the physical structure is changed the applications will also have to be modified. Thus in a hierarchical database the benefits of data independence is limited by structural dependence.

- **Programming Complexity.** Due to the structural dependence and the navigational structure, the application programmers and the end users must know precisely how the data is distributed physically in the database in order to access data. This requires knowledge of complex pointer systems, which is often beyond the grasp of ordinary users (users who have little or no programming knowledge).
- **Implementation Limitation.** Many of the common relationships do not conform to the l:n format required by the hierarchical model. The many-to-many (n:n) relationships, which are more common in real life are very difficult to implement in a hierarchical model.

NETWORK MODEL

The Network Model replaces the hierarchical tree with a graph thus allowing more general connections among the nodes. The main difference of the network model from the hierarchical model is its ability to handle many-to-many (n:n) relationships. Or in other words, it allows a record to have more than one parent. Suppose an employee works for two departments. The strict hierarchical arrangement is not possible here and the tree becomes a more generalized graph—a network. Logical proximity fails because you cannot place a data item simultaneously in two locations in the list. Although it is possible to handle such situations in a hierarchical model, it becomes more complicated and difficult to comprehend. The network model was evolved to specifically handle non-hierarchical relationships.

In network database terminology, a relationship is a set. Each set is made of at least two types of records: an owner record (equivalent to the parent in the hierarchical model) and a member record (similar to the child record in the hierarchical model). The difference between the hierarchical model and the network model is that the network model allows a record to appear as a member in more than a set thus facilitating many-to-many relationships.

Advantages

The network model retains almost all the advantages of the hierarchical model while eliminating some of its shortcomings. The main advantages of the network model are:

- **Conceptual simplicity.** Just like the hierarchical model, the network model is also conceptually simple and easy to design.
- **Capability to handle more relationship types.** The network model can handle the one-to-many (l:n) and many-to-many (n:n) relationships, which is a real help in modeling the real life situations.
- **Ease of data access.** The data access is easier than and flexible than in the hierarchical model. An application can access an owner record and all the member records within a set and if one member in the set has two owners (like the employee working for two departments), then one can move from one owner to another.

NOTES

- **Data integrity.** The network model does not allow a member to exist without an owner. Thus a user must first define the owner record and then the member record. This ensures the data integrity.
- **Data independence.** The network model is better than the hierarchical model in isolating (at least partially if not fully) the programs from the complex physical storage details. This, to a certain extent, ensures that the changes in data characteristics do not require changes to the application programs.
- **Database Standards.** One of the major drawbacks of the hierarchical model was the availability of universal standards for database design and modeling. The network model is based on the standards formulated by the DBTC (Database Task Croup of CODASYL Committee) and augmented by ANSI/SPARC (American National Standards Institute/Standards Planning and Requirements Committee) in the 1970s. All the network database management systems conformed to these standards. These standards included a data definition language (DDL) and a data manipulation language (DML), thus greatly enhancing database administration and portability.

Disadvantages

Even though the network database model was significantly better than the hierarchical database model, it also had many drawbacks. Some of them are:

- **System complexity.** Like the hierarchical model, the network model also provides a navigational access to the data in which the data are accessed one record at a time. This navigational data access mechanism makes the system implementation very complex and conseguently the database administrator designers. Programmers and even the end users should be familiar with the internal data structures in order to access the data. In other words, the network database model-cannot be used to create a user-friendly database management system.
- **Absence of structural independence.** Since the data access method in the network database model is a navigational system; making structural changes to the database is very difficult in most cases and impossible in some cases. If changes are made to the database structure then all the application programs need to be modified before they can access data. Thus, even though the network database model succeeds in achieving data independence, it still fails to achieve structural independence.

Because of the disadvantages mentioned and the implementation and administration complexities, both the hierarchical and network database models were replaced by the relational database model in the 1980s. The evolution of the relational database model is considered as one of the greatest events—a major breakthrough—in the history of database management. We will have an overview of the relational model in this chapter, but will discuss it in greater detail in chapters 7.

Note: Now we will consider a book-distributor example. A book can have many attributes like ISBN, Title, Author, Publisher, Year of Publication, Distributor, Price, etc. Similarly a distributor can have attributes like Name. Contact, Discount, Lead-time, etc. We will use this example to illustrate some of the models.

RELATIONAL MODEL STRUCTURE

Relational model stores data in the form of a table. Relational database powerful because they require few assumptions about how data is related or will be extracted from the database. As a result, the same database can be v in different ways. Another feature of relational systems is that a single da can be spread across several tables. This differs from flat-file database, in each database is self-contained in a single table.

The relational model uses tables to organize data elements. Each corresponds to an application entity, and each row represents an instance entity. For example, the book entity in an application corresponds to a boo in the database. Each row in the table represents a different book. Relatio link rows from two tables by embedding row identifiers (keys) from one table as attribute values in the other table. For example, the Distributor Name, which is the row identifier in the Distributor table, is embedded as an attribute in the book table and for each book there will be a distributor name thus associating the book with a distributor. Structured Query Language (SQL) serves as a uniform interface for users providing a collection of standard expressions for storing and retrieving data.

Although the relational model is currently the most popular database model, two other models—the object-oriented and deductive database models—which claim to be more flexible in data representation, at least for specialized applications are emerging into the commercial arena. Each offers an alternative to the table for maintaining relationships between elements.

Advantages

The major advantages of the relational model are:

1. **Structural independence.** The relational model does not depend on the navigational data access system thus freeing the database designers, programmers and end users from learning the details of data storage. Changes in the database structure do not affect the data access. When it is possible to make change to the database structure without affecting the DBMS's capability to access data, we can say that structural independence have been achieved. So *relational database model has /structural independence.*
2. **Conceptual simplicity.** We have seen that both the hierarchical and the network database model were conceptually simple. But the relational database model is even simpler at the conceptual level. Since the relational data model frees the designer from the physical data storage details, the designers can concentrate on the logical view of the database.
3. **Design, implementation, maintenance and usage ease.** The relational database model achieves both data independence and structural independence making the database design, maintenance, administration and usage much easier than the other models.
4. **Ad hoc query capability.** The presence of very powerful, flexible and easy-to-use query capability is one of the main reasons for the immense popularity of the relational database model. The query language of the relational database models—structured emery language or SQL—makes ad hoc queries a reality. SQL is a fourth generation language (4GL). A 4GL allows the user to specify what must be done without specifying how it must be done.

So using SQL, the users can specify what information they want and leave the details of how to get the information to the database. The Relational database will perform the task of translating the user queries (specified as SQL statements) into the technical code required to retrieve the requested.

Disadvantages

NOTES

The relational model's disadvantages are very minor compared to the advantages and their capabilities far outweigh the shortcomings. Also the drawbacks of the relational database systems could be avoided if proper corrective measures are taken. The drawbacks are not because of the shortcomings in the database model, but in the way it is being implemented. Some of the disadvantages are:

- **Hardware overheads.** Relational database systems hides the implementation-complexities and the physical data storage details from the users. For doing this, *i.e.*, for making things easier for the users, the relational database systems need more powerful hardware—computers and data storage devices. So the RDBMS needs powerful machines to run smoothly. But as the processing power of modern computers is increasing at an exponential rate and in today's scenario, the need for more processing power is no longer a very big issue.
- **Ease of design can lead to bad design.** The relational database is an easy-to-design and use system. The users need not know the complex details of physical data storage. They need not know how the data is actually stored to access it. This ease of design and use can lead to the development and use can lead to the development and implementation of very poorly designed database management systems. Since the database is efficient, these design inefficiencies will not come to light when the database is designed and when there is only a small amount of data. As the database grows, the poorly designed database) will slow the system down and will result in performance degradation and data corruption.
- **Information island phenomenon.** As we have said before, the relational database systems are easy to implement and use. This will create a situation where too many people or departments will create their on databases and applications. These information islands will prevent the information integration that is essential for the smooth and efficient functioning of the organization. These individual databases will also create problems like data inconsistency, data duplication, data redundancy and so on.

But as we have said all these issues are minor when compared to the advantages and all these issues could be avoided if the organization has a properly designed database and has enforced good database standards.

OBJECT-ORIENTED MODEL

Relational database technology has failed to handle the needs of complex information systems. The problem with relational database systems is that they require the application developer to force an information model into tables where relationships between entities are defined by values. Relational database design is really a process of trying to figure out how to represent real-world objects within the confines of tables in such a way that good performance results and preserving data integrity is possible. Object database design is quite different. For the most part, object database design is a fundamental part of the overall application design

process. The object classes used by the programming language are the classes used by the ODBMS. Because their models are consistent, there is no need to transform the program's object model to something unique for the database manager.

A data model is a collection of mathematically well-defined concepts that help one to consider and express the static and dynamic properties of data intensive applications. A data model consists of:

1. Static properties such as objects, attributes and relationships
2. Integrity rules over objects and operations
3. Dynamic properties such as operations or rules defining new database states based on applied state changes.

NOTES

Object-oriented databases have the ability to model all three of these components directly within the database supporting a complete problem/solution modeling capability. Prior to object-oriented databases, databases were capable of directly supporting points 1 and 2 above and relied on applications for defining the dynamic properties of the model. The disadvantage of delegating the dynamic properties to applications is that these dynamic properties could not be applied uniformly in all database usage scenarios since they were defined outside the database in autonomous applications. Object-oriented databases provide a unifying paradigm that allows one to integrate all three aspects of data modeling and to apply them uniformly to all users of the database.

(Object-Oriented model represents an entity as a class. A class represents both object attributes as well as the behavior of the entity. For example a book class will have not only the book attributes such as ISBN, Title, Author, Publisher, Year of Publishing, Distributor. Price, etc. but also procedures that imitate actions expected of a book such as Update Price (updating the price). Instances of the class-object correspond to individual books. Within an object the class attributes takes specific values, which distinguish one book (object) from another. However the behavior patterns of the class is shared by all the objects belonging to the class. The object-oriented model does not restrict attribute values to the small set of native data types usually associated with databases and programming languages, such as integer, numeric, character, etc. Instead the values can be other objects. For example, one of the attributes of a book can be distributor and the value of that attribute can be a distributor object corresponding to the distributor who is distributing the book.

The object-oriented model maintains relationships through 'logical containment'. Consider the book-distributor example. You find the distributor of a particular book in the book as one of its attributes. Since distributor is an object in its own right, you can recursively examine its attributes. The distributor object can have a title attribute, which can be a book object (assuming for the moment, that the distributor has only one book). But the book is the same book in which you originally found the distributor. So book 'B' contains distributor 'BD' which contains book 'B'. This is logical containment and this is how the object-oriented model maintains relationships.

Object-oriented databases manage objects (abstract data types). An object-oriented DBMS, or OODBMS, is suited for multimedia applications as well as data with complex relationships that are difficult to model and process in a relational DBMS. Because any type of data can be stored, an OODBMS allows for fully integrated databases that hold data, text, pictures, voice and video.

NOTES

Advantages

The object-oriented database model has many advantages over the other database models:

- **Capability to handle large number of different data types.** Traditional database models—hierarchical, network and relational database—are limited in their capability to store the different types of data. For example one cannot store pictures, voices and video in these databases. But the object-oriented database can store any type of data including text, numbers, pictures, voice and video.
- **Marriage of object-oriented programming and database technology.** Perhaps the most significant characteristic of object-oriented database technology is that it combines object-oriented programming with database technology to provide an integrated application development system. There are many advantages to including the definition of operations with the definition of data. First, the defined operations apply ubiquitously and are not dependent on the particular database application running at the moment. Second, the data types can be extended to support complex data such as multimedia by defining new object classes that have operations to support the new kinds of information.
- **Object-oriented features improve productivity. Inheritance** allows one to develop solutions to complex problems incrementally by defining new objects in terms of previously defined objects. **Polymorphism** and dynamic binding allow one to define operations for one object and then to share the specification of the operation with other objects. These objects can further extend this operation to provide behaviors that are unique to those objects. **Dynamic binding** determines at runtime, which of these operations is actually executed, depending on the class of the object requested to perform the operation. Polymorphism and dynamic binding are powerful object-oriented features that allow one to compose objects to provide solutions without having to write code that is specific to each object. All of these capabilities come together synergistically to provide significant productivity advantages to database application developers.
- **Data access.** Object-oriented databases represent relationships explicitly, supporting both navigational and associative" access to information. As the complexity of interrelationships between information within the database increases, the greater the advantages of representing relationships explicitly. Another benefit of using explicit relationships is the improvement in data access performance over relational value-based relationships.

Disadvantages

The following are some of the disadvantages of object-oriented database model:

- **Difficult to maintain.** In the real world, the data model is not static and will change as organizational information needs change and as missing information is identified. Consequently, the definition of objects must be changed periodically and existing databases migrated to conform to the new object definitions. Object-oriented databases are semantically rich introducing a number of challenges when changing object definitions and migrating databases. Object-oriented databases have a greater challenge handling schema migration because it's not sufficient to simply migrate the data representation to conform to the changes in class specifications. One must

also update the behavioral code associated with each object.

- **Not suited for all applications.** Object-oriented database systems are not suited for all applications. If it is used in situations where it is not required, then it will result in performance degradation and high processing requirements. OODBMS' have established themselves in niches such as e-commerce, engineering product data management, and special purpose databases in areas such as securities and medicine. The strength of the object model is in applications where there is an underlying need to manage complex relationships among data objects.

Today, it's unlikely that OODBMS' are a threat to the stronghold that relational database vendors have in the market place. Clearly, there is a partitioning of the market into databases that are best suited for handling high volume low, complexity data and databases that are suited for high complexity, reasonable volume, with OODBMS filling the need for the latter.

OBJECT-RELATIONAL MODEL

Relational database management is one of the most successful technologies in computer science. A lot of money is spent each year on relational database systems and applications, and much of the world's business data is stored in relational form. Until recen most of the individual data items stored in relational databases were relatively small and simple. For storing these simple data items, database systems supported a set of predefined data types such as integers, real numbers, and character strings. The operations defined over these data types, such as arithmetic and comparison, were also simple and predefined.

Increasingly, modern database applications need to store and manipulate objects that are neither small nor simple, and to perform operations on these objects that are not predefined. For example, the planning department of a city might need to store maps, photographs, written documents with diagrams, and audio and video recordings. A planner might need to find all the parcels of property that intersect a proposed highway route or to find the minutes of all meetings in which construction of new schools was discussed. Multimedia applications such as this one are among the fastest-growing segments of the database industry, and because of the very large amounts of data that they require, we can expect the requirements of these applications to become increasingly important.

Clearly, the traditional data types and search capabilities of SQL are not sufficient for the new generation of multimedia database applications. But it is also clear that the requirements of these applications are so diverse that they cannot be satisfied by any set of predefined language extensions. What we need is not a collection of new data types and functions, but a facility that lets users define new data types and functions of their own.

Another requirement of modern applications is databases that not only store data but also record and enforce business rules that apply to the data. Associating rules with data makes the data more "active," enabling the database system to perform automatic validity checks and to automate many business procedures. Making data active increases its value because it enables applications to share not only the data itself but also the behavior of the data. Rules are stored in the database rather than being encoded in each application, so redundancy is eliminated and the integrity of the data is protected.

NOTES

Allowing users to define their own data types and functions, and allowing users to define rules that govern the behavior of active data, are both ways of increasing the value of stored data by increasing its semantic content. The trend toward increasing the semantic content of stored data is the most important trend in database management today. In order to accommodate and facilitate this trend, relational database systems are being enhanced in two ways:

1. By adding an "object infrastructure" to the database system itself, in the form of support for user-defined data types, functions, and rules
2. By building "relational extenders" on top of this infrastructure that support specialized applications such as image retrieval, advanced text searching, and geographic applications.

A system that includes both object infrastructure and a set of relational extenders that exploit it is called an "object-relational" database system. An object relational system is a good long-term investment, because its extenders provide the capabilities you need to manage today's specialized objects, and its object infrastructure gives you the ability to define new types, functions, and rules to deal with the evolving needs of your business. Some of the object-relational systems available in the market are IBM's DB2 Universal Servers, Oracle Corporations Oracle8, Microsoft Corporations SQL Server 7 and so on.

DEDUCTIVE/INFERENCE MODEL

Deductive model also known as inferential model, stores as little data as possible but compensates by maintaining rules that allow new data combinations to be created when needed. Suppose the database stores the facts of the form DistBook(D, B), meaning that distributor 'D' is distributing the book 'B'. For example, DistBook (MindMart, Countdown 2000) can appear in the database meaning that the book 'Countdown 2000' is being distributed by the distributor with name 'MindMart'. A distributor can distribute more than one book and one book can be distributed by many distributors. So you can form another relationship where two books distributed by the same distributor and which can be represented as SameDist(X, Y), meaning books X and Y are distributed by the same distributor. Although the database explicitly stores the DistBook(D, B) facts, it does not store the SameDist(X, Y) facts. Instead it stores a rule stating that SameDist(X,Y) fact, say SameDistfCountdown 2000, SQL Handbook), can be deduced from the existing facts say DistBookfMindMart, Countdown 2000) and DistBook(MindMart, SQL Handbook). Such inference rules indirectly capture relationship groupings. The database thus stores certain elementary facts—axioms—from which other facts can be derived as and when needed using the rules.

COMPARISON AMONG THE VARIOUS DATABASE MODELS

The Table 1 summarizes the characteristics of the five database models. The first column gives the name of the model, the second specifies how the data elements are physically organized in the database and the third column specifies how the relationships between the data elements are established.

NOTES

Table 1. Comparison between the Database Model

Model	Data Element Organization	Relationship Organization	Identity Language	Access
Hierarchical	Files, Records	Logical proximity in a Linearized tree	Record based	Procedural
Network	Files, Records	Intersecting Networks	Record based	Procedural
Relational	Tables	Indentifiers of rows in one table are embedded as attribute values in another table	Value based	Non-procedural
Object-Oriented	Objects	Logical Containment-Related objects are found within a given object by recursively examing attributes of an object that are themselves objects	Record based	Procedural
Object-Relational	Object-infrastructure to the database system itself—user-defined data types, functions, and rules	Relational extenders that support support speciallized applications such as image retrieval, advanced text searching, and geographic applications.	Value based	Non-procedural
Deductive	Facts, Rules	Inference rules that permit related facts to be generated on demand.	Value based	Non-procedural

Within the database an application object or relationship appears as a data element or grouping of data elements. For example, suppose we are describing a man and the attributes that we have included are name, age, height, weight, and eye color. This we have represented as (Ravi, 31,180, 90, Blue) which means that the man's name is Ravi, he is 31 years old and his height is 180 centimeters and weight 90 Kilograms and the color of his eyes are blue. Suppose a year later Ravi fell ill and lost his weight. So the data structure changes to (Ravi, 32,180, 70, Blue). But the real Ravi is the same person as before and therefore the data structure has not changed its identity, but only its values. But the question is how many values can change before a new person emerges. If Ravi changes his name, becomes very lean and reduces weight, does plastic surgery, undergoes a sex transition! Will he (or shall we say she) become a new person? From the database standpoint the question is simpler—if all the attributes of the data structure change, has a new entity been created? The object-oriented, network and hierarchical models assume that the object survives the changes of all its attributes. These systems are **record-based**. A record of the real world item appears in the database and even though the record contents may change completely the record itself represents the application entity. As long as the record remains in the database, the object's identity has not changed.

In contrast the relational, object-relational and deductive models are **value-based**. They assume that the real world item has not identity independent of the attribute values. So if Ravi changes his attributes sufficiently to represent a woman, then he is a she! With this representation, the content of the database record, rather than its existence, determines the identity of the object represented.

The last column of the table is the **Access Language**. Each model uses a particular style of access language to manipulate the database contents. Some models employ a procedural language, precisely a sequence of operations to compute the desired results. Other use non-procedural languages, stating only the

NOTES

desired results and leaving the specific computation to the DBMS. The relational and deductive database models use non-procedural languages while the object-oriented, hierarchical and network models use procedural languages.

Schemas, Instances and Database State

The description of a database is called the database schema. The schema is specified during database design, and is not expected to change frequently.

The Schema is the structure of data, whereas the Data are the "facts". Schema can be complex to understand to begin with, but really indicates the rules which the Data must obey.

Imagine a case where we want to store facts about employees in a company. Such facts could include their name, address, date of birth, and salary. In a database all the information on all employees would be held in a single storage "container", called a table. This table is a tabular object like a spreadsheet page, with different employees as the rows, and the facts (*e.g.* their names) as columns... Let's call this table EMP, and it could look something like:

Name	Address	Date of Birth	Salary
Jim Smith	1 Apple Lane	1/3/1991	11000
Jon Greg	5 Pear St	7/9/1992	13000
Bob Roberts	2 Plumb Road	3/2/1990	12000

From this information the schema would define that EMP has four components, "NAME","ADDRESS","DOB","SALARY". As designers we can call the columns what we like, but making them meaningful helps. In addition to the name, we want to try and make sure that people dont accidentally store a name in the DOB column, or some other silly error. Protecting the database against rubbish data is one of the most important database design steps, and is what much of this course is about. From what we know about the facts, we can say things like:

- NAME is a string, and needs to hold at least 12 characters.
- ADDRESS is a string, and needs to hold at least 12 characters.
- DOB is a date... The company forbids people over 100 years old or younger than 18 years old working for them.
- SALARY is a number. It must be greater than zero.

Such rules can be enforced by a database. During the design phase of a database schema these and more complex rules are identified and where possible implemented. The more rules the harder it is to enter poor quality data.

Each object in the schema is called a schema construct.

The data in a database may change frequently, every time records are added or updated. The data in the database at a given moment in time is called the **database state** or **snapshot.**

Database Schema vs Database State

When a database is defined, the schema is specified to the DBMS. The database state at this point is in the empty state, with no data.

The initial state of the database is when the database is first populated or loaded with the initial data. Every time data is added/removed/updated, there is a new database state.

The DBMS is responsible for ensuring every state is a **valid state**, a state that satisfies the structure and constraints specified in the schema.

The DBMS stores the descriptions of the schema constructs and constraints, called the meta data, in the DBMS catalogue.

The schema is called the intension, and the database state an extension of the schema.

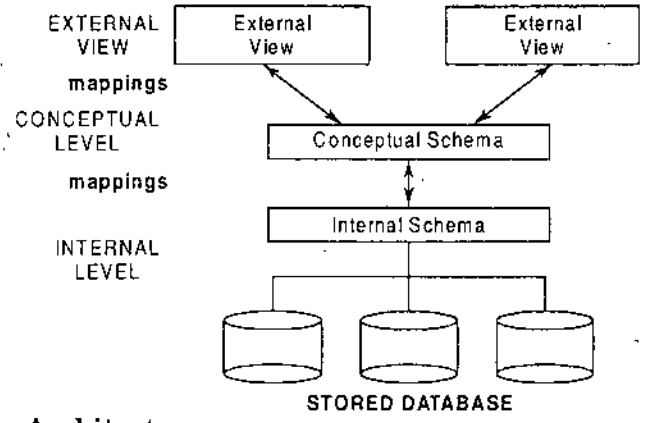
THREE SCHEMA ARCHITECTURE AND DATA INDEPENDENCE

NOTES

Three of the main characteristics of database systems, these are:

1. Insulation of programs and data
2. Support of multiple views
3. Use of a catalogue to store the database description (schema)

The three schema architecture helps to achieve these characteristics.



Three Schema Architecture

The goal of the three schema architecture is to separate the user applications and the physical database. The schemas can be defined at the following levels:

The **external level** is the view that the individual user of the database has. This view is often a restricted view of the database and the same database may provide a number of different views for different classes of users. In general, the end users and even the applications programmers are only interested in a subset of the database. For example, a department head may only be interested in the departmental finances and student enrolments but not the library information. The librarian would not be expected to have any interest in the information about academic staff. The payroll office would have no interest in student enrolments.

The **conceptual view** is the information model of the enterprise and contains the view of the whole enterprise without any concern for the physical implementation. This view is normally more stable than the other two views. In a database, it may be desirable to change the internal view to improve performance while there has been no change in the conceptual view of the database. The conceptual view is the overall community view of the database and it includes all the information that is going to be represented in the database. The conceptual view is defined by the conceptual schema which includes definitions of each of the various types of data.

The **internal view** is the view about the actual physical storage of data. It tells us what data is stored in the database and how. At least the following aspects are considered at this level:

1. Storage allocation *e.g.*, B-trees, hashing etc.
2. Access paths *e.g.*, specification of primary and secondary keys, indexes and pointers and sequencing.

NOTES

3. Miscellaneous e.g. data compression and encryption techniques, optimization of the internal structures.

Efficiency considerations are the most important at this level and the data structures are chosen to provide an efficient database. The internal view does not deal with the physical devices directly. Instead it views a physical device as a collection of physical pages and allocates space in terms of logical pages.

The three schema architecture is used to visualize the schema levels in a database. The three schemas are only descriptions of data, the data only actually exists is at the physical level.

Each user group refers only to its own external schema. The DBMS must transform a request specified on an external schema into a request against the conceptual schema, and then into a request on the internal schema for processing over the database. The process of transforming requests and results between levels is called mapping.

Mappings

- The conceptual/internal mapping:
  - defines conceptual and internal view correspondence
  - specifies mapping from conceptual records to their stored counterparts
- An external/conceptual mapping:
  - defines a particular external and conceptual view correspondence
- A change to the storage structure definition means that the conceptual/internal mapping must be changed accordingly, so that the conceptual schema may remain invariant, achieving physical data independence.
- A change to the conceptual definition means that the conceptual/external mapping must be changed accordingly, so that the external schema may remain invariant, achieving logical data independence.

DATA INDEPENDENCE

The three schema architecture further explains the concept of data independence, the capacity to change the schema at one level without having to change the schema at the next higher level.

There are two types of data independence

1. **Logical data independence**
   - the ability to change the conceptual schema without having to change the external schemas or application programs. When data is added or removed, only the view definition and the mappings need to be changed in the DBMS that support logical data independence.
   - If the conceptual schema undergoes a logical reorganization, application programs that reference the external schema constructs must work as before.
2. **Physical data independence**
   - The ability to change the internal schema without having to change the conceptual schema. By extension, the external schema should not change as well.
   - Physical file reorganization to improve performance (such as creating access structures) results in a change to the internal schema. If the same data as before remains in the database, the conceptual schema should not change.
   - For example, providing an access path to improve retrieval speed of section records by semester and year, should not require a query to be

changed, although it should become more efficient by utilizing the access path.

With a multi-level DBMS, the catalogue must be expanded to include information on how to map requests and data among the levels. The DBMS uses additional software to accomplish the mappings.

Data independence occurs because when the schema is changed at some level, the schema at the next higher level remains unchanged. Only the mapping between the levels is changed.

Database Languages and Interfaces

NOTES

- Because a database supports a number of user groups, as mentioned previously, the DBMS must have languages and interfaces that support each user group.

DBMS LANGUAGES

- DDL - the **data definition language**, used by the DBA and database designers to define the conceptual and internal schemas.
- The DBMS has a DDL compiler to process DDL statements in order to identify the schema constructs, and to store the description in the catalogue.
- In databases where there is a separation between the conceptual and internal schemas, DDL is used to specify the conceptual schema, and SDL, **storage definition language**, is used to specify the internal schema.
- For a true three-schema architecture, VDL, **view definition language**, is used to specify the user views and their mappings to the conceptual schema. But in most DBMSs, the DDL is used to specify both the conceptual schema and the external schemas.
- Once the schemas are compiled, and the database is populated with data, users need to manipulate the database. Manipulations include retrieval, insertion, deletion and modification.
- The DBMS provides operations using the DML, **data manipulation language**.
- In most DBMSs, the VDL, DML and the DML are not considered separate languages, but a comprehensive integrated language for conceptual schema definition, view definition and data manipulation. Storage definition is kept separate to fine-tune the performance, usually done by the DBA staff.
- An example of a comprehensive language: SQL, which represents a VDL, DDL, DML as well as statements for constraint specification, etc.

Data Manipulation Languages (DMLs)

Two main types:

High-level/Non procedural

- Can be used on its own to specify complex database operations.
- DMBSs allow DML statements to be entered interactively from a terminal, or to be embedded in a programming language. If the commands are embedded in a general purpose programming language, the statements must be identified so they can be extracted by a pre-compiler and processed by the DBMS.

Low Level/Procedural

- Must be embedded in a general purpose programming language.
- Typically retrieves individual records or objects from the database and processes each separately.

- Therefore it needs to use programming language constructs such as loops.
- Low-level DMLs are also called record at a time DMLS because of this.
- High-level DMLs, such as SQL can specify and retrieve many records in a single DML statement, and are called set at a time or set oriented DMLs.
- High-level languages are often called declarative, because the DML often specifies what to retrieve, rather than how to retrieve it.

DML Commands

NOTES

- When DML commands are embedded in a general purpose programming language, the programming language is called the **host language** and the DML is called the **data sub-language.**
- A high-level language used in a standalone, interactive manner is called a query language.
- Casual end users use high-level query language to specify requests, where programmers usually use embedded DML.
- Parametric end users usually interact with user-friendly interfaces, which can also be used by casual users who don't want to learn the high-level languages.

DCL - Data Control Language.

DCL statements are those which are used to control access permissions on the tables, indexes, views and other elements of the DBMS.

Granting & Revoking Privileges

Query:

```
GRANT ALL          <----------  Grants all permissions on the table customers to
ON customers                    the user who logs in as 'ashraf'.
TO ashraf;
```

Query:

```
GRANT SELECT       <----------  Grants SELECT permission on the table
ON customers                    customers to the user 'sunil'. User 'sunil' does not
TO sunil;                       have permission to insert, update, delete or
                                perform any other operation on customers table.
```

Query:

```
GRANT SELECT
ON customers
TO sunil
WITH GRANT OPTION; <----  Enables user 'sunil' to give SELECT permission
                          on customers table to other users.
```

Query:

```
REVOKE DELETE      <----------  Takes away DELETE permission on customers
ON customers FROM ashraf;      table from user 'ashraf'.
```

DBMS INTERFACES

Types of interfaces provided by the DBMS include:

Menu-Based Interfaces for Web Clients or Browsing

- Present users with list of options (menus)
- Lead user through formulation of request
- Query is composed of selection options from menu displayed by system.

Forms-Based Interfaces

- Displays a form to each user.
- User can fill out form to insert new data or fill out only certain entries.

- Designed and programmed for naïve users as interfaces to canned transactions.

Graphical User Interfaces

- Displays a schema to the user in diagram form. The user can specify a query by *manipulating the diagram. GUIs use both forms and menus.*

Natural Language Interfaces

- Accept requests in written English, or other languages and attempt to understand them.
- Interface has its own schema, and a dictionary of important words. Uses the schema and dictionary to interpret a natural language request.

Interfaces for Parametric Users

- Parametric users have small set of operations they perform.
- Analysts and programmers design and implement a special interface for each class of naïve users.
- Often a small set of commands included to minimize the number of keystrokes required. (*i.e.,* function keys)

Interfaces for the DBA

- Systems contain privileged commands only for DBA staff.
- Include commands for creating accounts, setting parameters, authorizing accounts, changing the schema, reorganizing the storage structures etc.

NOTES

DATABASE USERS

Actors on the Scene

1. **Database Administrator (DBA):** This is the chief administrator, who oversees and manages the database system (including the data and software). In large organizations, the DBA might have a support staff. Following are the duties of DBA.
   - **Defining the Schema :** The DBA defines the schema which contains the structure of the data in the application. The creation of the original database scheme involves writing a set of definitions in a DDL (data storage and definition language), compiled by the DDL compiler into a set of tables stored in the data dictionary.
   - **Liaising with Users :** The DBA needs to interact continuously with the users to understand the data in the system and its use.
   - **Scheme and physical organization modification :** Writing a set of definitions used by the DDL compiler to generate modifications to appropriate internal system tables (*e.g.* data dictionary). This is done rarely, but sometimes the database scheme or physical organization must be modified.
   - **Granting of authorization for data access :** Granting different types of authorization for data access to various users
   - **Integrity constraint specification :** Generating integrity constraints. These are consulted by the database manager module whenever updates occur.
   - **Defining Security and Integrity Checks :** The DBA finds about the access restrictions to be defined and defines security checks accordingly. Data Integrity checks are also defined by the DBA.
   - **Defining Backup / Recovery Procedures :** The DBA also defines procedures for backup and recovery. Defining backup procedures

NOTES

includes specifying what data is to backed up, the periodicity of taking backups and also the medium and storage place for the backup data.

- **Monitoring Performance :** The DBA has to continuously monitor the performance of the queries and take measures to optimize all the queries in the application.

2. **Database Designers:** They are responsible for identifying the data to be stored and for choosing an appropriate way to organize it. They also define views for different categories of users. The final design must be able to support the requirements of all the user sub-groups.
3. **End Users:** These are persons who access the database for querying, updating, and report generation. They are main reason for database's existence.
   - **Casual end users:** use database occasionally, needing different information each time; use query language to specify their requests; typically middle- or high-level managers.
   - **Naive/Parametric end users:** Typically the biggest group of users; frequently query/update the database using standard canned transactions that have been carefully programmed and tested in advance. Examples:
     - bank tellers check account balances, post withdrawals/deposits
     - reservation clerks for airlines, hotels, etc., check availability of seats/rooms and make reservations.
     - shipping clerks (*e.g.*, at UPS) who use buttons, bar code scanners, etc., to update status of in-transit packages.
   - **Sophisticated end users:** engineers, scientists, business analysts who implement their own applications to meet their complex needs.
   - **Stand-alone users:** Use "personal" databases, possibly employing a special-purpose (*e.g.*, financial) software package.
4. **System Analysts, Application Programmers, Software Engineers:**
   - **System Analysts:** determine needs of end users, especially naive and parametric users, and develop specifications for canned transactions that meet these needs.
   - **Application Programmers:** Implement, test, document, and maintain programs that satisfy the specifications mentioned above.

Workers behind the Scene

- **DBMS system designers/implementers:** provide the DBMS software that is at the foundation of all this.
- **Tool developers:** design and implement software tools facilitating database system design, performance monitoring, creation of graphical user interfaces, prototyping, ets.
- **Operators and maintenance personnel:** responsible for the day-to-day operation of the system.

STUDENT ACTIVITY-2

1. What is Data model ? Explain categories of Data model.

2. What is Hierarchical model ?

3. Explain object oriented model.

4. What is DBMS Languages ?

SUMMARY

NOTES

- In this chapter we defined a database as a collection of related data, where *data* means recorded facts. A typical database represents some aspect of the real world and is used for specific purposes by one or more groups of users. A DBMS is a generalized software package for implementing and maintaining a computerized database. The database and software together form a database system. We identified several characteristics that distinguish the database approach from traditional file-processing applications :
  - Existence of a catalog.
  - Program-data independence and program-operation independence.
  - Data abstraction.
  - Support of multiple user views.
  - Sharing of data among multiple transactions.
- We then discussed the *main categories of database users*, or the "actors on the scene".
  - Administrators.
  - Designers.
  - End users.
  - System analysts and application programmers.
- We noted that, in addition to database users, there are several categories of support personnel, or "workers behind the scene," in a database environment:
  - DBMS system designers and implementers.
  - Tool developers.
  - Operators and maintenance personnel.
- Then we presented a list of capabilities that should be provided by the DBMS software to the DBA, database designers, and users to help them design, administer, and use a database:
  - Controlling redundancy.
  - Restricting unauthorized access.
  - Providing persistent storage for program objects and data structures.
  - Permitting inferencing and actions by using rules.
  - *Providing multiple user interfaces.*
  - Representing complex relationships among data.
  - Enforcing integrity constraints.
  - Providing backup and recovery.
- We listed some additional advantages of the database approach over traditional file-processing systems:
  - Potential for enforcing standards.
  - Reduced application development time.
  - Flexibility.
  - Availability of up-to-date information to all users.
  - Economies of scale.
- *Finally, we discussed the overhead costs of using a DBMS and discussed* some situations in which it may not be advantageous to use a DBMS.
- In this chapter we introduced the main concepts used in database systems. We defined a data model, and we distinguished three main categories of data models.
  - High-level or conceptual data models (based on entities and relationships).

NOTES

- Low-level or physical data models.
- Representational or implementation data models (record-based, object-oriented).

- We distinguished the schema, or description of a database, from the database itself. The schema does not change very often, whereas the database state changes every time data is inserted, deleted, or modified. We then described the three-schema DBMS architecture, which allows three schema levels:
  - An internal schema describes the physical storage structure of the database.
  - A conceptual schema is a high-level description of the whole database.
  - External schemas describe the views of different user groups.
- A DBMS that cleanly separates the three levels must have mappings between the schemas to transform requests and results from one level to the next. Most DBMSs do not separate the three levels completely. We used the three-schema architecture to define the concepts of logical and physical data independence.
- We then discussed the main types of languages and interfaces the DBMSs support. A data definition language (DDL) is used to define the database conceptual schema. In most DBMSs, the DDL also defines user views and, sometimes, storage structures; in other DBMSs, separate languages (VDL, SDL) may exist for specifying views and storage structures. The DBMS compiles all schema definitions and stores their descriptions in the DBMS catalog. A data manipulation language (DML) is used for specifying database retrievals and updates. DMLs can be high-level (set-oriented, nonprocedural) ro low-level (record-oriented, procedural). A high-level DML can be embedded in a host programming language, or it can be used as a stand-alone language; in the latter case it is often called a query language.
- We discussed different types of interfaces provided by DBMSs, and the types of DBMS users with which each interface is associated. We then discussed the database system environment, typical DBMS software modules, and DBMS utilities for helping users and the DBA perform their tasks.
- In the final section, we classified DBMSs according to several criteria: data model, number of users, number of sites, cost, types of access paths, and generality. The main classification of DBMSs is based on the data model. We briefly discussed the main data models used in current commercial DBMSs.

TEST YOURSELF

1. What is a database schema?
2. What is the difference between internal and external schema?
3. Describe the organization of a database?
4. What is a database?

Fill in the Blanks

1. _____ represent a correspondence between the various data elements.
2. _____ are predicates that define correct database states and the schema describes the organization of data and relationships within the database.
3. The _____ defines various views of the database.
4. The _____ model defines the stored data structures in terms of the database model used.

NOTES

5. _____ is a collection of information organized in such a way that a computer program can quickly select desired pieces of data.
6. To access information from a database, you need a _____.
7. _____ neans that no two data items in a database should represent the same real-world entity.
8. A _____ is a software that provides services for accessing a database, while maintaining all the required features of the data.
9. A _____ is a sequence of database operations that represents a logical unit of work and that accesses a database and transforms it from one state to another.
10. The _____ defines how the data should be stored by the storage management mechanism and the storage interfaces with the operating system to access the physical storage.
11. A _____ provides a secure and survivable medium for the storage and retrieval of data.
12. A _____ is an organizing principle that specifies particular mechanisms for data storage and retrieval.
13. _____ was the first database model that offered the data security that is provided and enforced by the DBMS.
14. Each set is made of at least two types of records _____ and _____ records.
15. _____ are binary computer representation of stored logical entities.
16. _____ All software is divided into two general categories: _____ and _____.
17. _____ are collections of instructions for manipulating data.
18. The administrative information stored in data dictionaries is known as
19. _____ represent a correspondence between the various data elements.
20. _____ are predicates that define correct database states and the schema describes the organization of data and relationships within the database.
21. The _____ defines various views of the database.
22. The _____ model defines the stored data structures in terms of the database model used.
23. _____ is a collection of information organized in such a way that a computer program can quickly select desired pieces of data.
24. To access information from a database, you need a _____.
25. _____ means that no two data items in a database should represent the same real-world entity.

True or False

1. In database management systems, data files are the files that store the database information.
2. The external schema defines how and where data are organized in physical data storage.
3. A schema separates the physical aspects of data storage from the logical aspect of data representation.
4. The conceptual schema defines a view or views of the database for particular users. Z
5. A collection of data designed to be used by different people is called a database.
6. In a database, the data are stored in such a fashion that they are independent of the programs of people using the data.
7. Using a database redundancy can be reduced.
8. The data in a database cannot be shared.

NOTES

9. A database can avoid data inconsistency.
10. Security restrictions are impossible to be applied in a database.
11. In a database data integrity can be maintained.

Multiple Choice

1. Which of the following is a database elements?
   - ❒ Data
   - ❒ Relationships
   - ❒ Constraints and schema
   - ❒ All of the above
2. What are binary computer representations of stored logical entities?
   - ❒ Data
   - ❒ Relationships
   - ❒ Constraints
   - ❒ schema
3. What represent a correspondence between the various data elements?
   - ❒ Data
   - ❒ Relationships
   - ❒ Constraints
   - ❒ schema
4. What are predicates that define correct database states and the schema describes the organization of data and relationships within the database?
   - ❒ Data
   - ❒ Relationships
   - ❒ Constraints
   - ❒ schema
5. What separates the physical aspects of data storage from the logical aspects of data representation?
   - ❒ Data
   - ❒ Relationships
   - ❒ Constraints
   - ❒ schema
6. What defines how and where data are organized in physical data storage?
   - ❒ Internal schema
   - ❒ External schema
   - ❒ Conceptual schema
   - ❒ None of the above
7. Which of the following defines the stored data structures in terms of the database model used?
   - ❒ Internal schema
   - ❒ External schema
   - ❒ Conceptual schema
   - ❒ None of the above
8. What defines a view or views of the database for particular users?
   - ❒ Internal schema
   - ❒ External schema
   - ❒ Conceptual schema
   - ❒ None of the above
9. A collection of data designed to be used by different people is called a _____.
   - ❒ Database
   - ❒ DBMS
   - ❒ RDBMS
   - ❒ None of the above
10. To access information from a database, you need a _____.
    - ❒ EIS
    - ❒ DBMS
    - ❒ MIS
    - ❒ None of the above
11. Which of the following is an advantage of database?
    - ❒ Reduction in redundany
    - ❒ Avoidance of inconsistency
    - ❒ Security enforcement
    - ❒ All of the above
12. Which of the following is a characteristic of the data in a database?
    - ❒ Shared
    - ❒ Secure
    - ❒ Independent
    - ❒ All of the above
13. Which of the following is an example of a database application?
    - ❒ Computerized library systems
    - ❒ ATMs
    - ❒ Flight reservations systems
    - ❒ All of the above
14. DBMS stands for _____.
    - ❒ Data Blocking and Management Systems
    - ❒ Database Management Systems
    - ❒ Database Business Management Systems
    - ❒ None of the above

SECTION B

2. Entity Relational Model
3. Relational Model
4. DBMS Based on Relational Model

CHAPTER 2 ENTITY RELATIONAL MODEL

NOTES

LEARNING OBJECTIVES

- Classification of DBMSs
- Database Design Process
- Three-level Database Model
- Entity Relationship (ER) Model
- Entity Types, Entity Sets, Keys and Value Sets
- ER Diagram Notation
- Relationship Types, Relationship Sets, Roles and Structural Constraints
- Mapping Constraints
- Entity Sets
- Constructing an ER Model
- Reducing E-R Diagrams to Tables

CLASSIFICATION OF DBMSs

1. **Data Model Classification**
   - Relational data model
   - Object data model
   - Hierarchical data model
   - Network data model
   - Object relational data model
2. **Number of Users**
   - Single User systems
   - Multi User systems
3. **Number of Sites**
   - Centralized - data is stored at single site.
   - Distributes - database and DBMS software stored over many sites connected by network
   - Homogeneous - use same DBMS software at multiple sites.

DATABASE DESIGN PROCESS

The database design process consists of a number of steps listed below. We will focus mainly on step 2, the conceptual database design, and the models used during this step.

Step 1: Requirements Collection and Analysis

- Prospective users are interviewed to understand and document data requirements
- This step results in a concise set of user requirements, which should be detailed and complete.

NOTES

- The functional requirements should be specified, as well as the data requirements. Functional requirements consist of user operations that will be applied to the database, including retrievals and updates.
- Functional requirements can be documented using diagrams such as sequence diagrams, data flow diagrams, scenarios, etc.

Step 2: Conceptual Design

- Once the requirements are collected and analyzed, the designers go about creating the conceptual schema.
- Conceptual schema: concise description of data requirements of the users, and includes a detailed description of the entity types, relationships and constraints.
- The concepts do not include implementation details; therefore the end users easily understand them, and they can be used as a communication tool.
- The conceptual schema is used to ensure all user requirements are met, and they do not conflict.

Step 3: Database Implementation

- Many DBMS systems use an implementation data model, so the conceptual schema is transformed from the high-level data model into the implementation data model.
- This step is called logical design or data model mapping, which results in the implementation data model of the DBMS.

Step 4: Physical Design

- Internal storage structures, indexes, access paths and file organizations are specified.
- Application programs are designed and implemented.

Data analysis Introduction

Data analysis is concerned with the NATURE and USE of data. It involves the identification of the data elements which are needed to support the data processing system of the organization, the placing of these elements into logical groups and the definition of the relationships between the resulting groups.

Other approaches, e.g. D.F.Ds and Flowcharts, have been concerned with the flow of data-dataflow methodologies. Data analysis is one of several data structure based methodologies Jackson SP/D is another.

Systems analysts often, in practice, go directly from fact finding to implementation dependent data analysis. Their assumptions about the usage of properties of and relationships between data elements are embodied directly in record and file designs and computer procedure specifications. The introduction of Database Management Systems (DBMS) has encouraged a higher level of analysis, where the data elements are defined by a logical model or 'schema' (conceptual schema). When discussing the schema in the context of a DBMS, the effects of alternative designs on the efficiency or ease of implementation is considered, *i.e.,* the analysis is still somewhat implementation dependent. If we consider the data relationships, usages and properties that are important to the business without regard to their representation in a particular computerised system using particular software, we have what we are concerned with, implementation independent data analysis.

It is fair to ask why data analysis should be done if it is possible, in practice to go straight to a computerised system design. Data analysis is time consuming; it throws up a lot of questions. Implementation may be slowed down while the answers are sought. It is more expedient to have an experienced analyst 'get on

with the job' and come up with a design straight away. The main difference is that data analysis is more likely to result in a design which meets both present and future requirements, being more easily adapted to changes in the business or in the computing equipment. It can also be argued that it tends to ensure that policy questions concerning the organisations' data are answered by the managers of the organisation, not by the systems analysts. Data analysis may be thought of as the 'slow and careful' approach, whereas omitting this step is 'quick and dirty'.

From another viewpoint, data analysis provides useful insights for general design principals which will benefit the trainee analyst even if he finally settles for a 'quick and dirty' solution.

The development of techniques of data analysis have helped to understand the structure and meaning of data in organisations. Data analysis techniques can be used as the first step of extrapolating the complexities of the real world into a model that can be held on a computer and be accessed by many users. The data can be gathered by conventional methods such as interviewing people in the organisation and studying documents. The facts can be represented as objects of interest. There are a number of documentation tools available for data analysis, such as entity relationship diagrams. These are useful aids to communication, help to ensure that the work is carried out in a thorough manner, and ease the mapping processes that follow data analysis. Some of the documents can be used as source documents for the data dictionary.

In data analysis we analyse the data and build a systems representation in the form of a data model (conceptual). A conceptual data model specifies the structure of the data and the processes which use that data.

Data Analysis = establishing the nature of data.

Functional Analysis = establishing the use of data.

However, since Data and Functional Analysis are so intermixed, we shall use the term Data Analysis to cover both.

Building a model of an organisation is not easy. The whole organisation is too large as there will be too many things to be modelled. It takes too long and does not achieve anything concrete like an information system, and managers want tangible results fairly quickly. It is therefore the task of the data analyst to model a particular view of the organisation, one which proves reasonable and accurate for most applications and uses. Data has an intrinsic structure of its own, independent of processing, reports formats etc. The data model seeks to make explicit that structure

Data analysis was described as establishing the nature and use of data.

Database Analysis Life Cycle

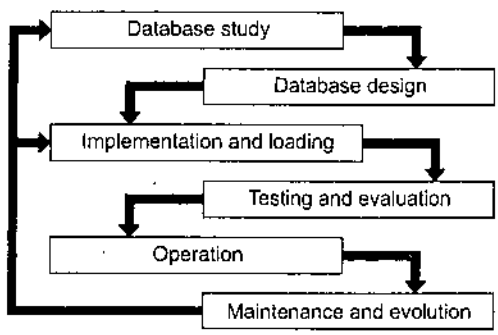


Figure 1. Database analysis life cycle

NOTES

When a database designer is approaching the problem of constructing a database system, the logical steps followed is that of the database analysis life cycle:

- **Database study** - here the designer creates a written specification in words for the database system to be built. This involves:
  - analysing the company situation - is it an expanding company, dynamic in its requirements, mature in nature, solid background in employee training for new internal products, etc. These have an impact on how the specification is to be viewed.
  - define problems and constraints - what is the situation currently? How does the company deal with the task which the new database is to perform. Any issues around the current method? What are the limits of the new system?
  - define objectives - what is the new database system going to have to do, and in what way must it be done. What information does the company want to store specifically, and what does it want to calculate. How will the data evolve.
  - define scope and boundaries - what is stored on this new database system, and what it stored elsewhere. Will it interface to another database?
- **Database Design** - conceptual, logical, and physical design steps in taking specifications to physical implementable designs. This is looked at more closely in a moment.
- **Implementation and loading** - it is quite possible that the database is to run on a machine which as yet does not have a database management system running on it at the moment. If this is the case one must be installed on that machine. Once a DBMS has been installed, the database itself must be created within the DBMS. Finally, not all databases start completely empty, and thus must be loaded with the initial data set (such as the current inventory, current staff names, current customer details, etc).
- **Testing and evaluation** - the database, once implemented, must be tested against the specification supplied by the client. It is also useful to test the database with the client using mock data, as clients do not always have a full understanding of what they thing they have specified and how it differs from what they have actually asked for! In addition, this step in the life cycle offers the chance to the designer to fine-tune the system for best performance. Finally, it is a good idea to evaluate the database in-situ, along with any linked applications.
- **Operation** - this step is where the system is actually in real usage by the company.
- **Maintenance and evolution** - designers rarely get everything perfect first time, and it may be the case that the company requests changes to fix problems with the system or to recommend enhancements or new requirements.
  - o Commonly development takes place without change to the database structure. In elderly systems the DB structure becomes fossilised.

THREE-LEVEL DATABASE MODEL

Often referred to as the three-level model, this is where the design moves from a written specification taken from the real-world requirements to a physically-implementable design for a specific DBMS. The three levels commonly referred to are 'Conceptual Design', 'Data Model Mapping', and 'Physical Design'.

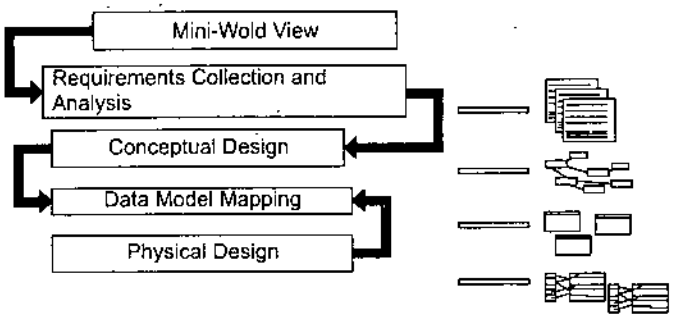


Figure 2. Logic behind the three level architecture

NOTES

The specification is usually in the form of a written document containing customer requirements, mock reports, screen drawings and the like, written by the client to indicate the requirements which the final system is to have. Often such data has to be collected together from a variety of internal sources to the company and then analysed to see if the requirements are necessary, correct, and efficient.

Once the Database requirements have been collated, the Conceptual Design phase takes the requirements and produces a high-level data model of the database structure. In this module, we use ER modelling to represent high-level data models, but there are other techniques. This model is independent of the final DBMS which the database will be installed in.

Next, the Conceptual Design phase takes the high-level data model it taken and converted into a conceptual schema, which is specific to a particular DBMS class (*e.g.*, relational). For a relational system, such as Oracle, an appropriate conceptual schema would be relations.

Finally, in the Physical Design phase the conceptual schema is converted into database internal structures. This is specific to a particular DBMS product

ENTITY RELATIONSHIP (ER) MODEL

Entity Relationship (ER) modelling

- is a design tool
- is a graphical representation of the database system
- provides a high-level conceptual data model
- supports the user's perception of the data
- is DBMS and hardware independent
- had many variants
- is composed of entities, attributes, and relationships

The diagrammatic notation associated with the ER model, is referred to as the ER diagram. ER diagrams show the basic data structures and constraints.

Entities

- An entity is any object in the system that we want to model and store information about
- Individual objects are called entities
- Groups of the same type of objects are called entity types or entity sets
- Entities are represented by rectangles (either with round or square corners)

Entities

- There are two types of entities; weak and strong entity types.

An entity is an object distinguishable from other objects and represented by a set of attributes. An entity set is a set of entities of same type. Entity sets need not be disjoint.

- Each entity is described by a set of (attribute, data value) pairs for each of its attributes in the entity set.

NOTES

ENTITY TYPES, ENTITY SETS, KEYS AND VALUE SETS

Entity Types and Entity Sets

- An entity type defines a collection of entities that have the same attributes. Each entity type in the database is described by its name and attributes. The entity share the same attributes, but each entity has its own value for each attribute.

Entity Type Example:

- Entity Type:
  Student
- Entity Attributes:
  StudentID,
  Name,
  Surname,
  Date of Birth,
  Department
- The collection of all entities of a particular entity type in the database at any point in time is called an entity set. The entity type (Student) and the entity set (Student) can be referred to using the same name.
  Entity Set Example:
- Entity Type: Student
- Entity Set:
  [123, John, Smith, 12/01/1981, Computer Technology]
  [456, Jane, Doe, 05/02/1979, Mathematics]
  [789, Semra, Aykan, 02/08/1980, Linguistics]

The entity type describes the intension, or schema for a set of entities that share the same structure. The collection of entities of a particular entity type is grouped into the entity set, called the extension.

Key Attributes of an Entity Type

- An important constraint on entities of an entity type is the uniqueness constraint.
- A key attribute is an attribute whose values are distinct for each individual entity in the entity set.
- The values of the key attribute can be used to identify each entity uniquely.
- Sometimes a key can consist of several attributes together, where the combination of attributes is unique for a given entity. This is called a composite key.
- Composite keys should be minimal, meaning that all attributes must be included to have the uniqueness property.
- Specifying that an attribute is a key of an entity type means that the uniqueness property must hold true for every entity set of the entity type.

- An entity can have more than one key attribute, and some entities may have no key attribute. Those entities with no key attribute are called weak entity types.

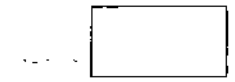
Entity	Attributes	Values
Car	Color	Red
	Make	Volkswagen
	Model	Bora
	Year	2000

Value Sets (Domains) of Attributes

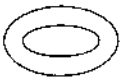
NOTES

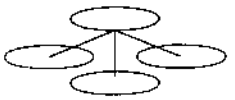
- Each simple attribute of an entity is associates with a domain of values, or value set, which specifies the set of values that may be assigned to that attribute for each entity. For example, date of birth must be before today's date, and after 01/01/1900, or the Student Name attribute must be a string of alphabetic characters.
- Value sets are not specified in ER diagrams.

ER DIAGRAM NOTATION


- Entity


- Attribute


- Multi-valued Attribute


- Composite Attributes

Key Attributes

Examples of entities and attributes:

There are several types of entities. Including:

- Simple vs. Composite
- Single-valued vs. Multi-valued
- Stored vs. Derived

Simple vs. Composite Attributes

- Composite attributes can be divided into smaller subparts, which represent more basic attributes, which have their own meanings.
- A common example of a composite attribute is Address. Address can be broken down into a number of subparts, such as Street Address, City, Postal Code. Street Address may be further broken down by Number, Street Name and Apartment/Unit number.
- Attributes that are not divisible into subparts are called simple or atomic attributes.
- Composite attributes can be used if the attribute is referred to as the whole, and the atomic attributes are not referred to. For example, if you wish to store the Company Location, unless you will use the atomic information such as Postal Code, or City separately from the other Location information (Street Address etc) then there is no need to subdivide it into its component attributes, and the whole Location can be designated as a simple attribute.

NOTES

Single-Valued vs. Multi-valued Attributes

- Most attributes have a single value for each entity, such as a car only has one model, a student has only one ID number, an employee has only one data of birth. These attributes are called single-valued attributes.
- Sometimes an attribute can have multiple values for a single entity, for example, a doctor may have more than one specialty (or may have only one specialty), a customer may have more than one mobile phone number, or they may not have one at all. These attributes are called multi-valued attributes.
- Multi-valued attributes may have a lower and upper bounds to constrain the number of values allowed. For example, a doctor must have at least one specialty, but no more than 3 specialties.

Stored vs. Derived Attributes

- If an attribute can be calculated using the value of another attribute, they are called derived attributes.
- The attribute that is used to derive the attribute is called a stored attribute.
- Derived attributes are not stored in the file, but can be derived when needed from the stored attributes.

Null Valued Attributes

- There are cases where an attribute does not have an applicable value for an attribute. For these situations, the value null is created.
- A person who does not have a mobile phone would have null stored at the value for the Mobile Phone Number attribute.
- Null can also be used in situations where the attribute value is unknown. There are two cases where this can occur, one where it is known that the attribute is valued, but the value is missing, for example hair color. Every person has a hair color, but the information may be missing. Another situation is if mobile phone number is null, it is not known if the person does not have a mobile phone or if that information is just missing.

Complex Attributes

- Complex attributes are attributes that are nested in an arbitrary way.
- For example a person can have more than one residence, and each residence can have more than one phone, therefore it is a complex attribute that can be represented as:
- {Multi-valued attributes are displayed between braces}
- (Complex Attributes are represented using parentheses)

E.g.

{**Address Phone** ({Phone (AreaCode, PhoneNumber)}, Address (Street Address (Number, Street, Apartment Number), City, State, Zip))}

Company Database Example

The company database keeps track of a company's employees, departments and projects. Suppose that after the requirements collection and analysis phase, the database designers provided the following description of the part of the company to be represented by the database.

NOTES

1. The company is organized into department. Each department has a unique name a unique number and a particular employee who manages the department. We keep track of the start date when that employee began managing the department. A department may have several locations.
2. A department controls a number of projects, each of which has a unique name, a unique number and a single location.
3. We store each employees name, ID number, address, salary, sex and birth date. An employee is assigned to one department but may work on several projects, which are not necessarily controlled by the same department. We keep track of the number of hours per week that an employee works on each project. We also keep track of the direct supervisor of each employee.
4. We want to keep track of the dependents of each employee for insurance purposes. We keep each dependent's first name, sex, birth date and relationship to the employee.

From the information given above, we can identify 4 entities.

1. **Department** - Name, Number, Locations, Manager, and Manager Start Date.
2. **Project** - Name, Number, Location and Controlling Department.
3. **Employee** - *Name, ID Number, Sex, Address, Salary, Birth Date, Department.* Both Name and Address can be a composite attribute, however it was not specified in the requirements.
4. **Dependent** - Employee, Dependent Name, Sex, Birth Date, Relationship

The information about the projects an employee works on can be represented in two ways. One, we can include a multi-valued composite, attribute, WorksOn(Project, Hours) in the Employee entity, or we can include a multi-valued composite attribute, Workers(Employee, Hours).

RELATIONSHIP TYPES, RELATIONSHIP SETS, ROLES AND STRUCTURAL CONSTRAINTS

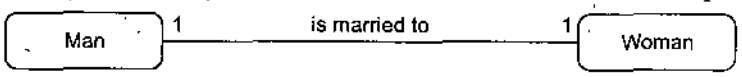
- Looking at the example above, there are several implicit relationships among the entity types.
- Whenever an attribute of one entity type refers to another entity type, some *relationship exists.*
- For example, Manager of a department refers to an employee who manages the department, Controlling Department of the project, refers to the department that controls the project. Supervisor of an employee refers to the *employee who supervises that employee.*
- In an ER diagram, these references are not represented as attributes, but as relationships.

Cardinality

- Relationships are rarely one-to-one
- For example, a manager usually manages more than one employee
- This is described by the cardinality of the relationship, for which there are four possible categories.
- One to one (1:1) relationship
- One to many (1:m) relationship
- Many to one (m:1) relationship
- Many to many (m:n) relationship
- On an ER diagram, if the end of a relationship is straight, it represents 1, while a "crow's foot" end represents many.

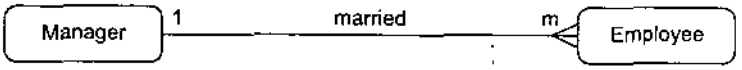
NOTES

- A one to one relationship - a man can only marry one woman, and a woman can only marry one man, so it is a one to one (1:1) relationship



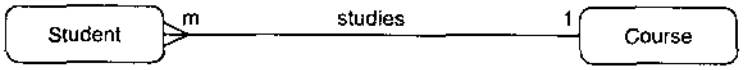
One to One relationship example

- A one to may relationship - one manager manages many employees, but each employee only has one manager, so it is a one to many (1:n) relationship



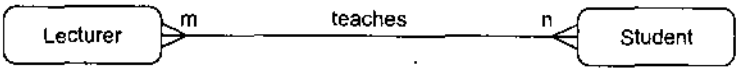
One to Many relationship example

- A many to one relationship - many students study one course. They do not study more than one course, so it is a many to one (m:1) relationship



Many to One relationship example

- A many to many relationship - One lecturer teaches many students and a student is taught by many lecturers, so it is a many to many (m:n) relationship



Many to Many relationship example

Figure 3. Relationship types

Optionality

A relationship can be optional or mandatory.

- If the relationship is mandatory
- an entity at one end of the relationship must be related to an entity at the other end.
- The optionality can be different at each end of the relationship
- For example, a student must be on a course. This is mandatory. To the relationship `student studies course' is mandatory.
- But a course can exist before any students have enrolled. Thus the relationship `course is_studied_by student' is optional.
- To show optionality, put a circle or `0' at the `optional end' of the relationship.
- As the optional relationship is `course is_studied_by student', and the optional part of this is the student, then the `O' goes at the student end of the relationship connection.

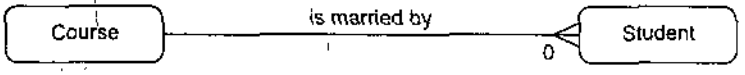


Figure 4. Optionality example

- It is important to know the optionality because you must ensure that whenever you create a new entity it has the required mandatory links.

Role Names

- Each entity type that participates in a relationship type plays a particular role in the relationship.

- The role name shows the role that an particular entity from the entity type plays in each relationship.
- Example: In the Company diagram, in the WorksFor relationship type, the employee plays the role of employee or worker, and the department entity plays the role of department or employer.
- In some cases the same entity participates more than once in a relationship type in different roles.
- For example, the Supervision relationship type relates an employee to a supervisor, where both the employee and supervisor are of the same employee entity type, therefore the employee entity participates twice in the relationship, once in the role of supervisor, and once in the role of supervisee.

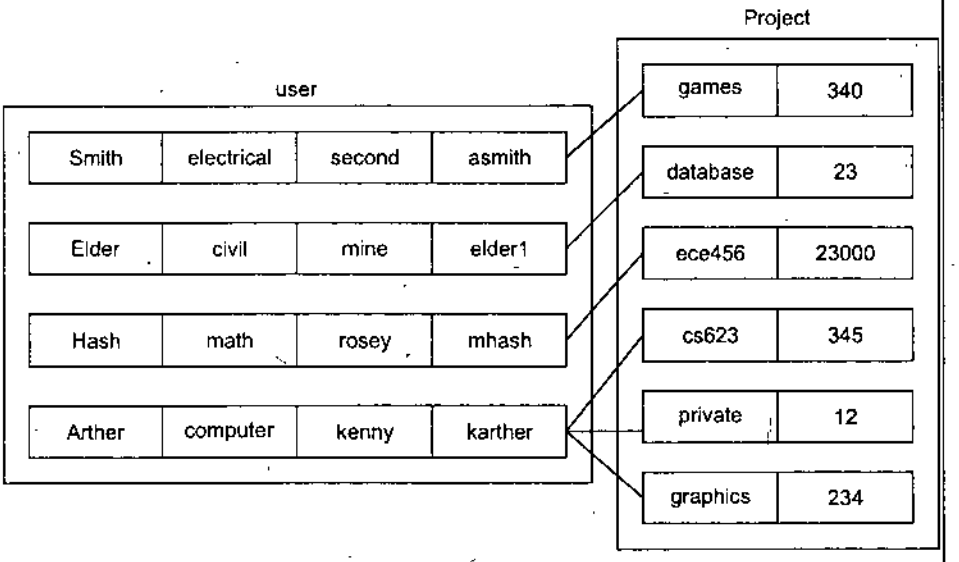
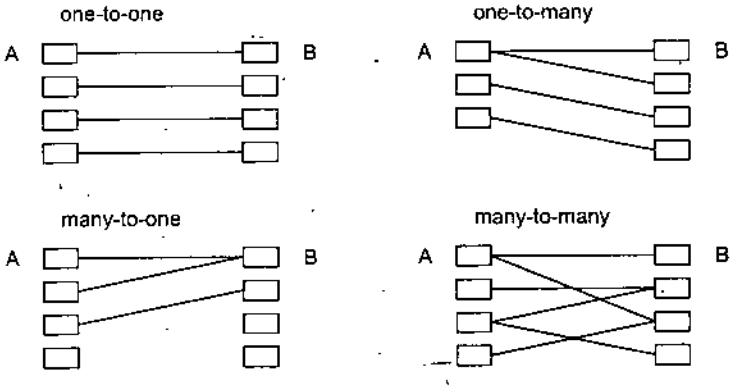


Figure 5. Example of a relationship set

MAPPING CONSTRAINTS

- Mapping Cardinalities: number of entities to which an entity can be associated with a relationship



- if existence of entity r is dependent on existence of entity s (*i.e.*, if s is deleted then so is r), then entity s is dominant and entity r is subordinate.

Relationship Types, Sets and Instances

- A relationship type, R, among entities, defines a relationship set among entities from the entity types.

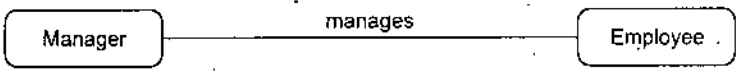
NOTES

Relationships

- A relationship type is a meaningful association between entity types.
- A relationship is an association of entities where the association includes one entity from each participating entity type.
- Relationship types are represented on the ER diagram by a series of lines.
- As always, there are many notations in use today... .
- In the original Chen notation, the relationship is placed inside a diamond, *e.g.*, managers manage employees:

Chens notation for relationships

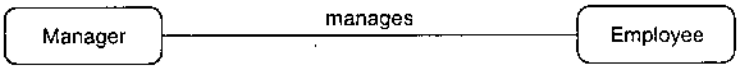
- For this module, we will use an alternative notation, where the relationship is a label on the line. The meaning is identical



Relationships used in this document

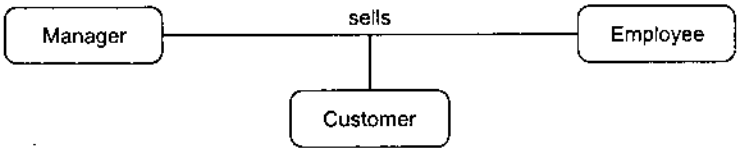
Degree of a Relationship

- The number of participating entities in a relationship is known as the degree of the relationship.
- If there are two entity types involved it is a binary relationship type



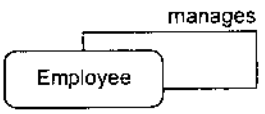
Binary Relationships

- If there are three entity types involved it is a ternary relationship type



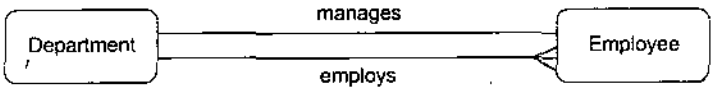
Ternary relationship

- It is possible to have a n-ary relationship (*e.g.*, quaternary or unary).
- Unary relationships are also known as a recursive relationship.



Recursive relationship

- It is a relationship where the same entity participates more than once in different roles.
- In the example above we are saying that employees are managed by employees.
- If we wanted more information about who manages whom, we could introduce a second entity type called manager.
- It is also possible to have entities associated through two or more distinct relationships.



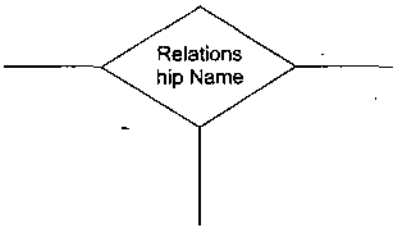
Multiple relationships

- In the representation we use it is not possible to have attributes as part of a relationship. To support this other entity types need to be developed.

N-ary relationships

NOTES

- More than 2 participating entities.



Replacing ternary relationships

When ternary relationships occurs in an ER model they should always be removed before finishing the model. Sometimes the relationships can be replaced by a series of binary relationships that link pairs of the original ternary relationship.

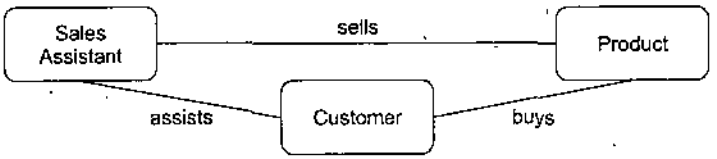


Figure 6. A ternary relationship example

- This can result in the loss of some information - It is no longer clear which sales assistant sold a customer a particular product.
- Try replacing the ternary relationship with an entity type and a set of binary relationships.

Relationships are usually verbs, so name the new entity type by the relationship verb rewritten as a noun.

- The relationship sells can become the entity type sale.

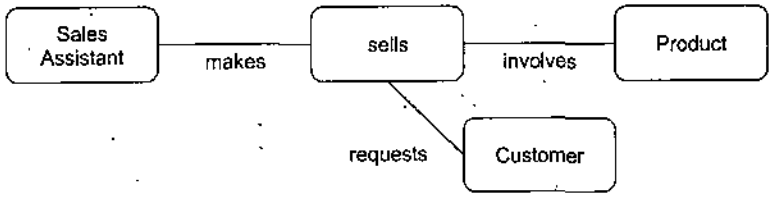
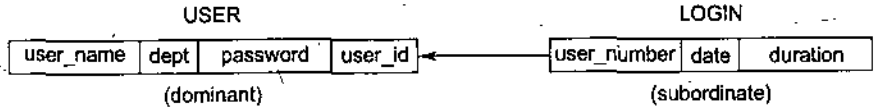


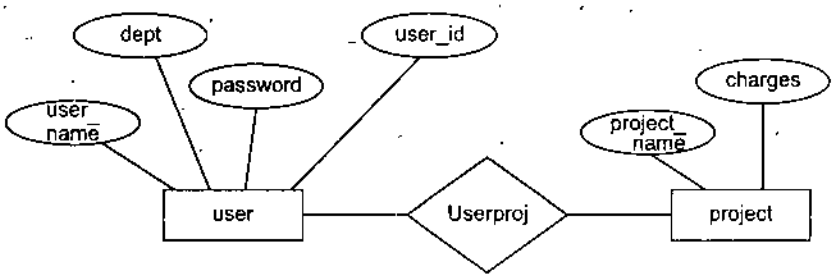
Figure 7. Replacing a ternary relationship

- So a sales assistant can be linked to a specific customer and both of them to the sale of a particular product.
- This process also works for higher order relationships.

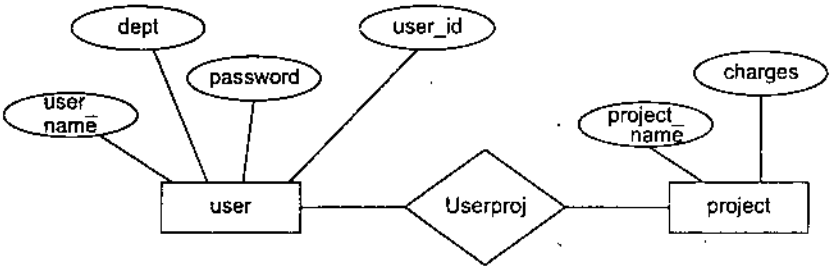


NOTES

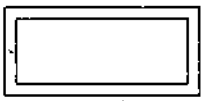
if user deleted, then corresponding logins deleted as well



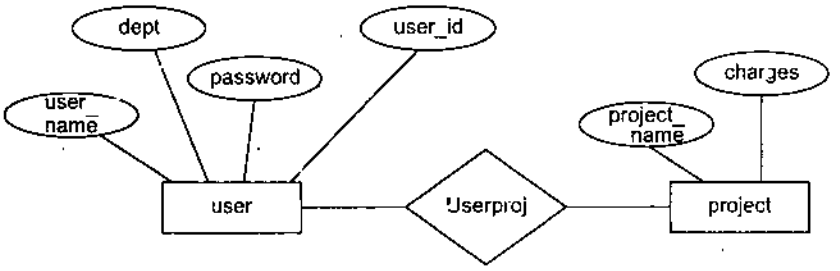
E-R diagram for many-to-many relationship



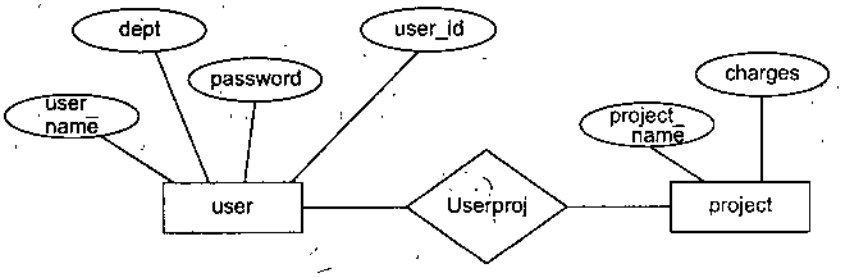
E-R diagram for one-to-many relationship



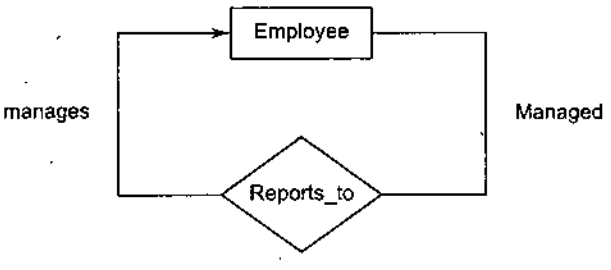
Weak entity set



E-R diagram for many-to-one relationship



E-R diagram for one-to-one relationship

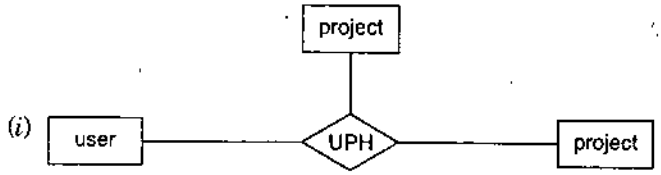


Use of Role Indicators

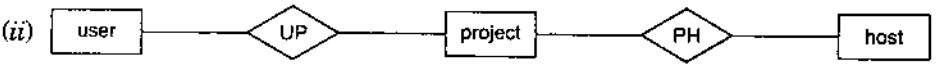
Figure 8.

Mapping Cardinalities

- consider the following ternary relationship:



can be represented as a binary relationship as:



- in *i*): one can only represent the relationship between a user and a project only if there is a corresponding host.
- in *ii*): a project can be related to a host without a corresponding user or to a user with no corresponding host.
- although, the scheme in *i*) seems more logically appropriate.

ENTITY SETS

Sometimes it is useful to try out various examples of entities from an ER model. One reason for this is to confirm the correct cardinality and optionality of a relationship. We use an `entity set diagram' to show entity examples graphically. Consider the example of `course is_studied_by student'.

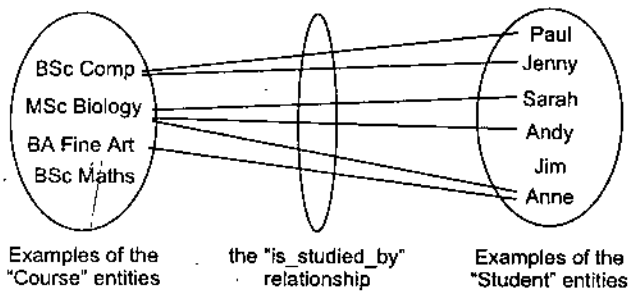


Figure 9. Entity set example

Confirming Correctness

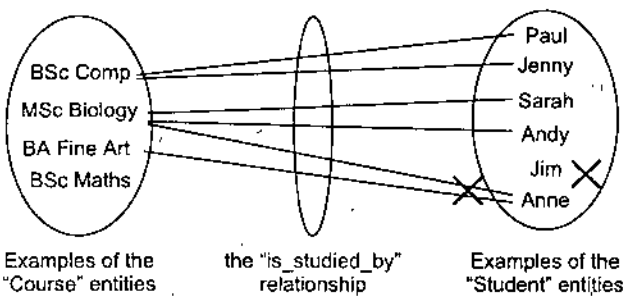


Figure 10. Entity set confirming errors

- Use the diagram to show all possible relationship scenarios.
- Go back to the requirements specification and check to see if they are allowed.
- If not, then put a cross through the forbidden relationships
- This allows you to show the cardinality and optionality of the relationship

NOTES

NOTES

Deriving the relationship parameters

To check we have the correct parameters (sometimes also known as the degree) of a relationship, ask two questions:

1. One course is studied by how many students? Answer = 'zero or more'.
   - This gives us the degree at the 'student' end.
   - The answer 'zero or more' needs to be split into two parts.
   - The 'more' part means that the cardinality is 'many'.
   - The 'zero' part means that the relationship is 'optional'.
   - If the answer was 'one or more', then the relationship would be 'mandatory'.
2. One student studies how many courses? Answer = 'One'
   - This gives us the degree at the 'course' end of the relationship.
   - The answer 'one' means that the cardinality of this relationship is 1, and is 'mandatory'
   - If the answer had been 'zero or one', then the cardinality of the relationship would have been 1, and be 'optional'.

Redundant relationships

Some ER diagrams end up with a relationship loop.

- Check to see if it is possible to break the loop without losing info
- Given three entities A, B, C, where there are relations A-B, B-C, and C-A, check if it is possible to navigate between A and C via B. If it is possible, then A-C was a redundant relationship.
- Always check carefully for ways to simplify your ER diagram. It makes it easier to read the remaining information.

Redundant relationships example

- Consider entities 'customer' (customer details), 'address' (the address of a customer) and 'distance' (distance from the company to the customer address).

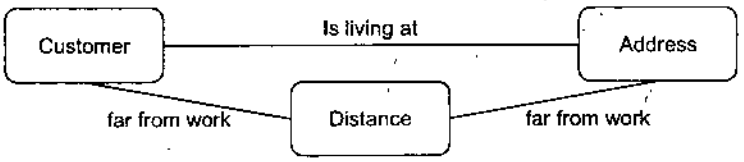


Figure 11. Redundant relationship

Splitting n:m Relationships

A many to many relationship in an ER model is not necessarily incorrect. They can be replaced using an intermediate entity. This should only be done where:

- the m:n relationship hides an entity
- the resulting ER diagram is easier to understand.

Splitting n:m Relationships - Example

Consider the case of a car hire company. Customers hire cars, one customer hires many card and a car is hired by many customers.

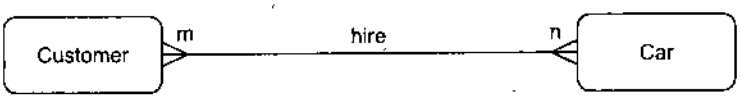


Figure 12. Many to Many example

The many to many relationship can be broken down to reveal a 'hire' entity, which contains an attribute 'date of hire'.

Figure 13. Splitting the Many to Many example

CONSTRUCTING AN ER MODEL

NOTES

Before beginning to draw the ER model, read the requirements specification carefully. Document any assumptions you need to make.

1. Identify entities - list all potential entity types. These are the object of interest in the system. It is better to put too many entities in at this stage and them discard them later if necessary.
2. Remove duplicate entities - Ensure that they really separate entity types or just two names for the same thing.
   - Also do not include the system as an entity type
   - *e.g.*, if modelling a library, the entity types might be books, borrowers, etc.
   - The library is the system, thus should not be an entity type.
3. List the attributes of each entity (all properties to describe the entity which are relevant to the application).
   - Ensure that the entity types are really needed.
   - are any of them just attributes of another entity type?
   - if so keep them as attributes and cross them off the entity list.
   - Do not have attributes of one entity as attributes of another entity!
4. Mark the primary keys.
   - Which attributes uniquely identify instances of that entity type?
   - This may not be possible for some weak entities.
5. Define the relationships
   - Examine each entity type to see its relationship to the others.
6. Describe the cardinality and optionality of the relationships
   - Examine the constraints between participating entities.
7. Remove redundant relationships
   - Examine the ER model for redundant relationships.

ER modelling is an iterative process, so draw several versions, refining each one until you are happy with it. Note that there is no one right answer to the problem, but some solutions are better than others!

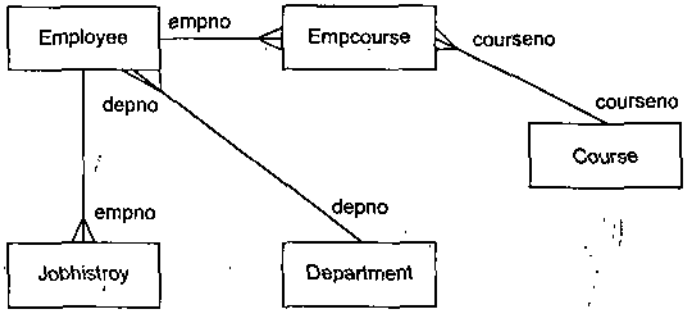


Figure 14. ER Diagram for Jobs

Country Bus Company

A Country Bus Company owns a number of busses. Each bus is allocated to a particular route, although some routes may have several busses. Each route

NOTES

passes through a number of towns. One or more drivers are allocated to each stage of a route, which corresponds to a journey through some or all of the towns on a route. Some of the towns have a garage where busses are kept and each of the busses are identified by the registration number and can carry different numbers of passengers, since the vehicles vary in size and can be single or double-decked. *Each route is identified by a route number and information is available on the* average number of passengers carried per day for each route. Drivers have an employee number, name, address, and sometimes a telephone number.

Entities

- Bus - Company owns busses and will hold information about them.
- Route - Buses travel on routes and will need described.
- Town - Buses pass through towns and need to know about them
- Driver - Company employs drivers, personnel will hold their data.
- Stage - Routes are made up of stages.
- Garage - Garage houses buses, and need to know where they are.

Relationships

- A bus is allocated to a route and a route may have several buses.
- Bus-route (m:1) is serviced by
- A route comprises of one or more stages.
- route-stage (1:m) comprises
- One or more drivers are allocated to each stage.
- driver-stage (m:1) is allocated
- A stage passes through some or all of the towns on a route.
- stage-town (m:n) passes-through
- A route passes through some or all of the towns
- *route-town (m:n) passes-through*
- Some of the towns have a garage
- garage-town (1:1) is situated
- A garage keeps buses and each bus has one 'home' garage
- garage-bus (m:1) is garaged

Draw E-R Diagram

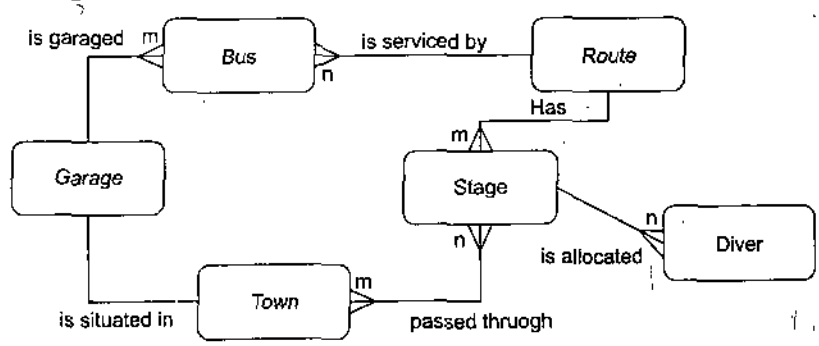


Figure 15. Bus Company

Attributes

- Bus (reg-no,make,size,deck,no-pass)
- Route (route-no,avg-pass)
- Driver (emp-no,name,address,tel-no)
- Town (name)

- Stage (stage-no)
- Garage (name,address)

Enhanced (Extended) ER Diagrams

- Contain all the basic modeling concepts of an ER Diagram
- *Adds additional concepts:*
  - Specialization/generalization
  - Subclass/super class
  - Categories
  - Attribute inheritance
- Extended ER diagrams use some object-oriented concepts such as inheritance.
- EER is used to model concepts more accurately than the ER diagram.

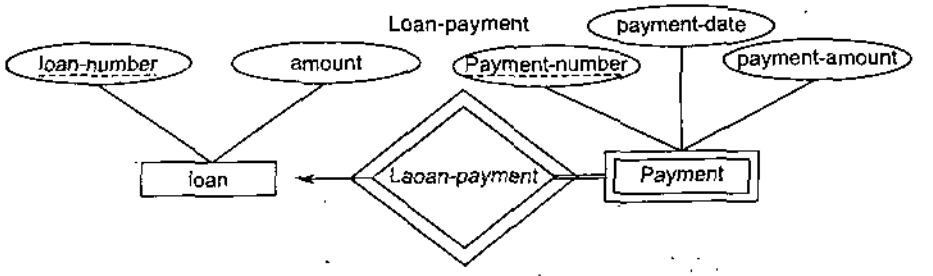
NOTES

Weak Entity

1. An entity set that does not have a primary key is referred to as a weak entity set.
2. The existence of a weak entity set depends on the existence of a identifying entity set.
3. It must relate to the identifying entity set via a total, one-to-many relationship set from the identifying to the weak entity set.
4. Identifying relationship depicted using a double diamond.
5. The discriminator (or partial key) of a weak entity set is the set of attributes that distinguishes among all the entities of a weak entity set.
6. The primary key of a weak entity set is formed by the primary key of the *strong entity set* on which the weak entity set is existence dependent, plus the weak entity set's discriminator.
7. We depict a weak entity set by double rectangles.
8. We *underline the discriminator of a weak entity set* with a dashed line.

Payment-number—discriminator of the payment entity set

Primary key for payment - (loan-number, payment-number)



Note: the primary key of the strong entity set is not explicitly stored with the weak entity set, since it is implicit in the identifying relationship.

If loan-number were explicitly stored, payment could be made a strong entity, but then the relationship between payment and loan would be duplicated by an implicit relationship defined by the attribute loan-number common to payment and loan.

More Weak Entity Set Examples

In a university, a course is a strong entity and a course-offering can be modeled as a weak entity.

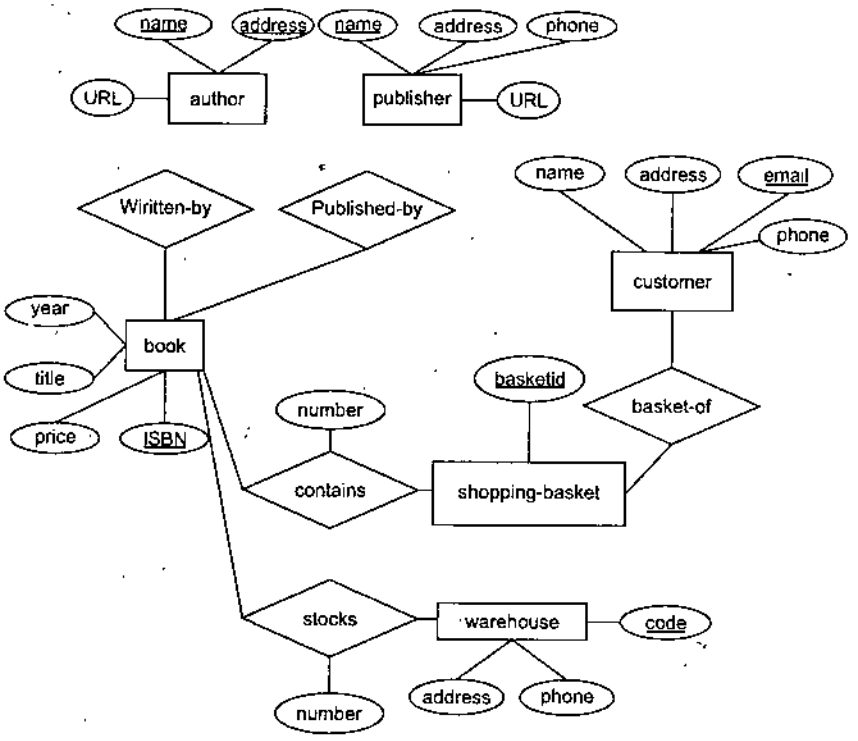
The discriminator of course-offering would be *semester (including year)* and section-number (if there is more than one section).

NOTES

E-R Diagram for a online bookstore

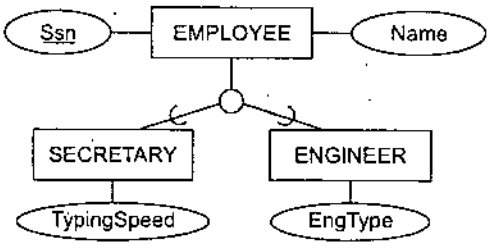
Consider the E-R diagram in figure below, which models an online bookstore.

1. List the entity sets and their primary keys.
2. Suppose the bookstore adds music cassettes and compact disks to its collection. The same music item may be present in cassette or compact disk format, with differing prices. Extend the E-R diagram to model this addition, ignoring the effect on shopping baskets.
3. Now extend the E-R diagram, using Generalization, to model the case where a shopping basket may contain any combination of books, music cassettes, or compact disks.



Sub classes and Super classes

- In some cases, and entity type has numerous sub-groupings of its entities that are meaningful, and need to be explicitly represented, because of their importance.
- For example, members of entity Employee can be grouped further into Secretary, Engineer, Manager, Technician, Salaried_Employee.
- The set listed is a subset of the entities that belong to the Employee entity, which means that every entity that belongs to one of the sub sets is also an Employee.
- Each of these sub-groupings is called a subclass, and the Employee entity is called the super-class.



- An entity cannot only be a member of a subclass; it must also be a member of the super-class.
- An entity can be included as a member of a number of sub classes, for example, a Secretary may also be a salaried employee, however not every member of the super class must be a member of a sub class.

Constraints - Participation

- Total Participation - entity X has total participation in Relationship Z, meaning that every instance of X takes part in AT LEAST one relationship. (*i.e.,* there are no members of X that do not participate in the relationship.

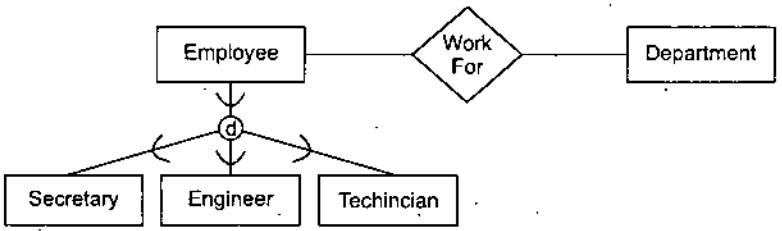
Example: X is Customer, Y is Product, and Z is a 'Purchases' relationship. The figure below indicates the requirement that every customer purchases a product.

- Partial Participation - entity Y has partial participation in Relationship Z, meaning that only some instances of Y take part in the relationship.

Example: X is Customer, Y is Product, and Z is a 'Purchases' relationship. The figure below indicates the requirement that not every product is purchases by a customer. Some products may not be purchased at all.

Type Inheritance

- The type of an entity is defined by the attributes it possesses, and the relationship types it participates in.
- Because an entity in a subclass represents the same entity from the super class, it should possess all the values for its attributes, as well as the attributes as a member of the super class.
- This means that an entity that is a member of a subclass inherits all the attributes of the entity as a member of the super class; as well, an entity inherits all the relationships in which the super class participates.



Specialization

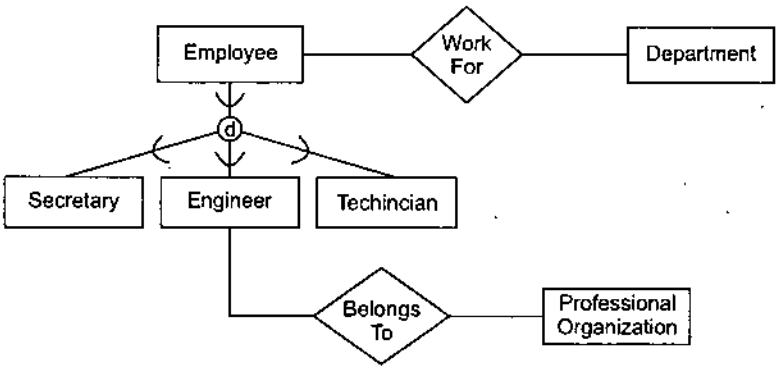
- The process of defining a set of subclasses of a super class.
- Specialization is the top-down refinement into (super) classes and subclasses.
- The set of sub classes is based on some distinguishing characteristic of the super class.
- For example, the set of sub classes for Employee, Secretary, Engineer, Technician, differentiates among employee based on job type.
- There may be several specializations of an entity type based on different distinguishing characteristics.
- Another example is the specialization, Salaried_Employee and Hourly_Employee, which distinguish employees based on their method of pay.

NOTES

NOTES

Notation for Specialization

- To represent a specialization, the subclasses that define a specialization are attached by lines to a circle that represents the specialization, and is connected to the super class.
- The subset symbol (half-circle) is shown on each line connecting a subclass to a super class, indicates the direction of the super class/subclass relationship.
- Attributes that only apply to the sub class are attached to the rectangle representing the subclass. They are called specific attributes.
- A sub class can also participate in specific relationship types.



Reasons for Specialization

- Certain attributes may apply to some but not all entities of a super class. A subclass is defined in order to group the entities to which the attributes apply.
- The second reason for using subclasses is that some relationship types may be participated in only by entities that are members of the subclass.

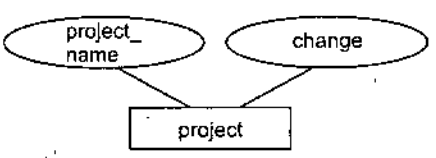
Summary of Specialization

Allows for:

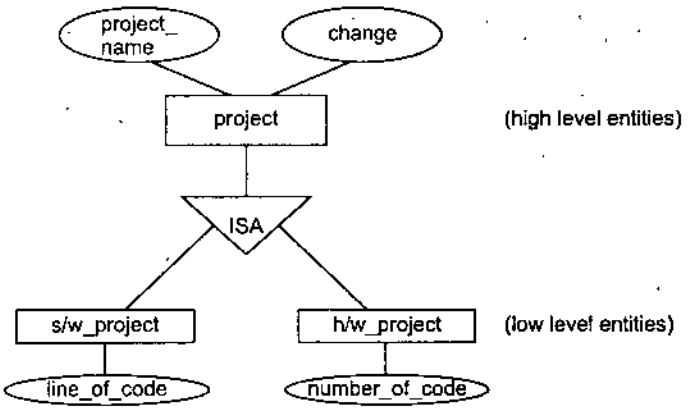
- Defining set of subclasses of entity type.
- Create additional specific attributes for each sub class.
- Create additional specific relationship types between each sub class and other entity types cr other subclasses.

Generalization

- The reverse of specialization is generalization.
- Several classes with common features are generalized into a super class.
- For example, the entity types Car and Truck share common attributes License_PlateNo, VehicleID and Price, therefore they can be generalized into the super class Vehicle.
- consider the project entity set:



- now we want to classify each project as either s/w or h/w related: create new entity sets: s/w_project h/w_project (additional attributes)*lines_of_code *number_of_ICsv



NOTES

- emphasize similarities, hide differences

Transformation to Tabular Form

Method One

- create a table for higher-level entity for each lower level entity create a table with a column for each of its attributes + a column for the primary key of the higher level entity, *e.g.,*:

(A) project with attributes {project_name, charges}
(B) s/w_project with attributes {project_name, lines_of_code}
(C) h/w_project with attributes {project_name, number_of_ICs}

Method Two

- create tables only for low level entities with full inheritance, *e.g.,*:

(A) s/w_project with attributes {project_name, charges, lines_of_code}
(B) h/w_project with attributes {project_name, charges, number_of_ICs}

Constraints on Specialization and Generalization

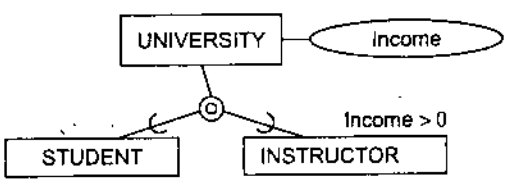
- Several specializations can be defined on an entity type.
- Entities may belong to subclasses in each of the specializations.
- The specialization may also consist of a single subclass, such as the manager specialization, in this case we don't use the circle notation.

Types of Specializations

Predicate-defined or Condition-defined specialization

- Occurs in cases where we can determine exactly the entities of each sub class by placing a condition of the value of an attribute in the super class.
- An example is where the Employee entity has an attribute, Job Type. We can specify the condition of membership in the Secretary subclass by the condition, JobType="Secretary"

Another Example:



- The condition is called the defining predicate of the sub class.
- The condition is a constraint specifying exactly those entities of the Employee entity type whose attribute value for Job Type is Secretary belong to the subclass.

NOTES

- Predicate defined subclasses are displayed by writing the predicate condition next to the line that connects the subclass to the specialization circle.

Attribute-defined specialization

- If all subclasses in a specialization have their membership condition on the same attribute of the super class, the specialization is called an attribute-defined specialization, and the attribute is called the defining attribute.
- Attribute-defined specializations are displayed by placing the defining attribute name next to the arc from the circle to the super class.

User-defined specialization

- When we do not have a condition for determining membership in a subclass the subclass is called user-defined.
- Membership to a subclass is determined by the database users when they add an entity to the subclass.

Disjointness/Overlap Constraint

- Specifies that the subclass of the specialization must be disjoint, which means that an entity can be a member of, at most, one subclass of the specialization.
- The d in the specialization circle stands for disjoint.
- If the subclasses are not constrained to be disjoint, they overlap.
- Overlap means that an entity can be a member of more than one subclass of the specialization.
- Overlap constraint is shown by placing an o in the specialization circle.

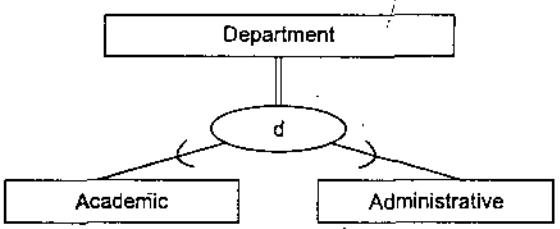
Completeness Constraint

- The completeness constraint may be either total or partial.
- A total specialization constraint specifies that every entity in the superclass must be a member of at least one subclass of the specialization.
- Total specialization is shown by using a double line to connect the super class to the circle.
- A single line is used to display a partial specialization, meaning that an entity does not have to belong to any of the subclasses.

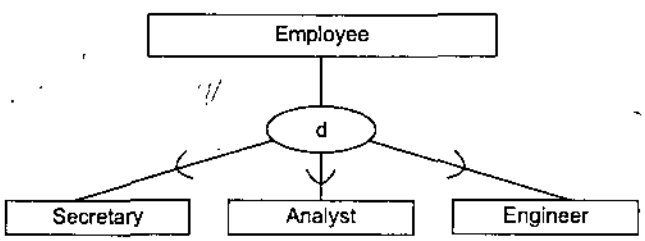
Disjointness vs. Completeness

- Disjoint constraints and completeness constraints are independent. The following possible constraints on specializations are possible:

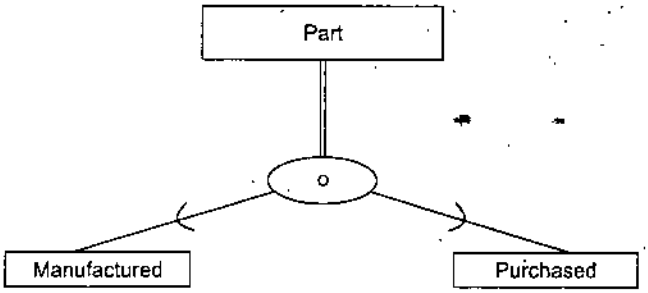
Disjoint, total



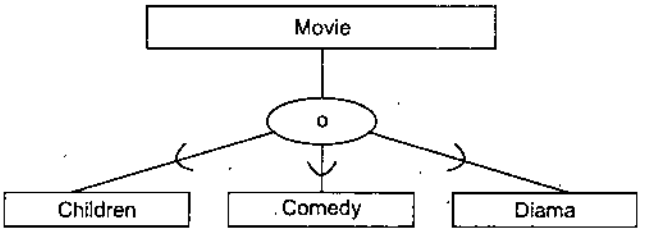
Disjoint, partial



Overlapping, total

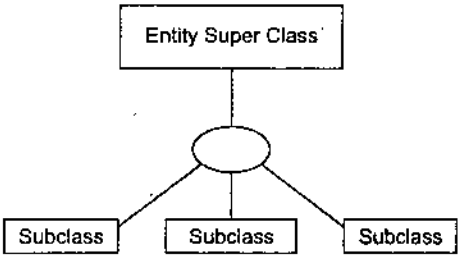


Overlapping, partial



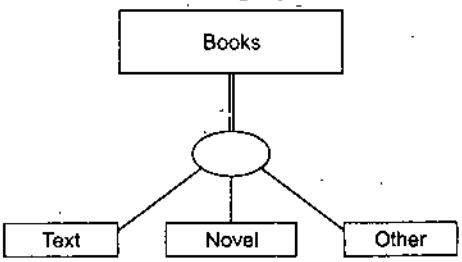
Specialization/Generalization

- Each subclass inherits all relationships and attributes from the super-class.

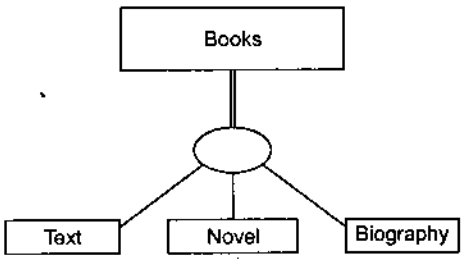


Constraints on Specialization/Generalization

- **Total Specialization** - Every member of the super-class must belong to at least one subclass. For example, any book that is not a text book, or a novel can fit into the "Other" category.



- **Partial Specialization** - each member of the super-class may not belong to one of the subclasses. For example, a book on poetry may be neither a text book, a novel or a biography.

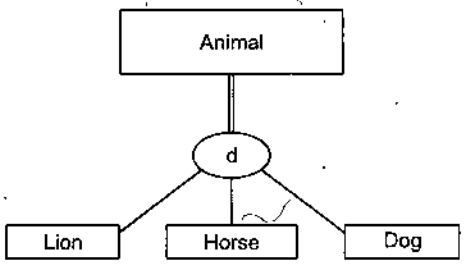


NOTES

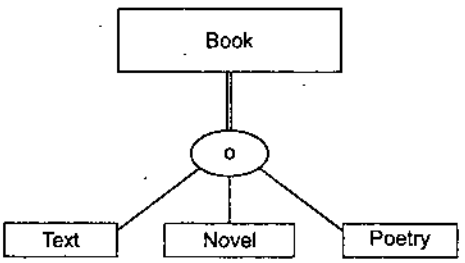
NOTES

Disjointness Constraint

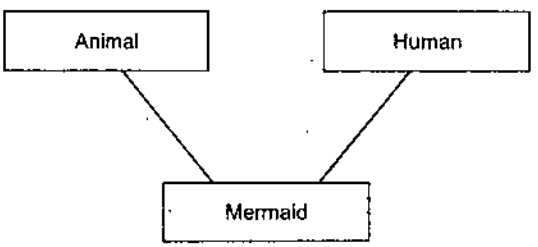
- **Disjoint** - every member of the super-class can belong to at most one of the subclasses. For example, an Animal cannot be a lion and a horse, it must be either a lion, a horse, or a dog.



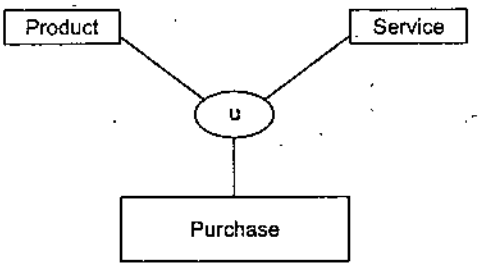
Overlapping - every member of the super-class can belong to more than one of the subclasses. For example, a book can be a text book, but also a poetry book at the same time.



Multiple Inheritance - a subclass participates in more than one subclass/super-class relationship, and inherits attributes and relationships from more than one super-class. For example, the subclass Mermaid participates in two subclass/ super-class relationships, it inherits attributes and relationships of Animals, as well as attributes and relationships of Humans.



Union - a subclass/super-class relationship can have more than one super-class, and the subclass inherits from at most one of the super-classes (*i.e.*, the subclass purchase will inherit the relationships and attributes associated with either service or product, but not both). Each super class may have different primary keys, or the same primary key. All members of the super-classes are not members of the super-class. For example, a purchase can be a product, or a service, but not both. And all products and services are not purchases.

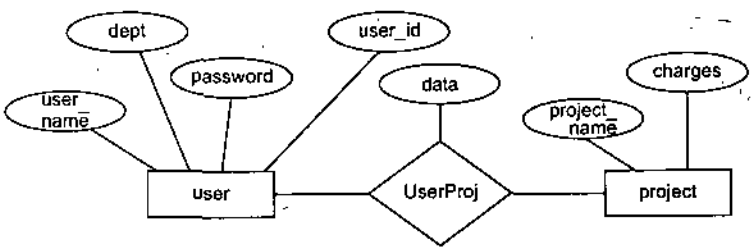


REDUCING E-R DIAGRAMS TO TABLES

- Strong entity sets:
  E with attributes $\{ a_1, a_2, ..., a_n \}$

	a_1	a_2	a_3	•	a_n
e_1					
e_2					
e_3					
•					
•					
•					
e_k					

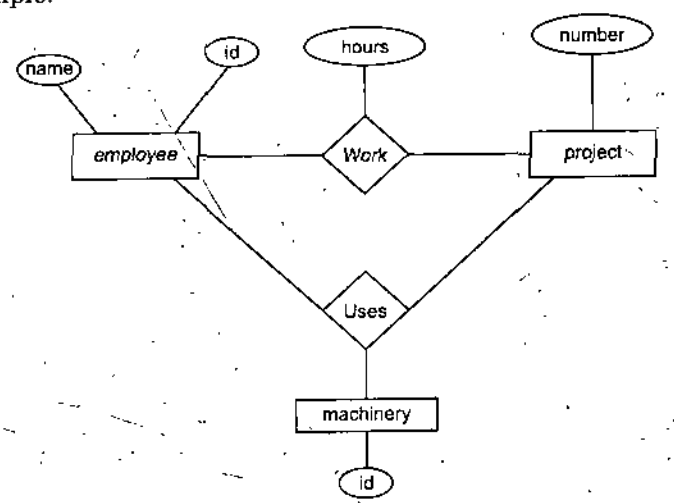
- Weak entity sets:
  - A - a weak entity set, attributes $\{a_1, ..., a_n\}$
  - B - a strong entity set with primary key $\{b_1,...., b_k\}$, where B is dominate to A
  - A represented by a table with one column for each attribute in the set $\{a_1, ..., a_n \} \cup \{b_1,...., b_k\}$
- Relationship sets:
  - Let R be a relationship set involving $E_1, E_2, ..., E_m$ and let attribute(R) consist of n attributes, the R can be represented in a table with n distinct columns.
- Consider the following example:



- Note that the primary key of user is user_id and the primary key of project is project_name and that date is the descriptive attribute of the relation UserProj.
  - ⇒ relationship set *UserProj is represented by a table with column attributes* {user_id, project_name, date}.

Aggregation

- may need to express relationships among relationships.
- for example:



NOTES

NOTES

- appears as if work and uses could be combined into one relationship. (but it would obscure the logical structure of the scheme)
- aggregation is best used as an abstraction where relationships are treated as higher level entities, *e.g.*,:
- in tabular form: need tables for: employee, project, work, machinery, uses

Problems with ER Models

There are several problems that may arise when designing a conceptual data model. These are known as connection traps.

There are two main types of connection traps:

1. fan traps
2. chasm traps

Fan traps

A fan trap occurs when a model represents a relationship between entity types, but the pathway between certain entity occurrences is ambiguous. It occurs when 1:m relationships fan out from a single entity.

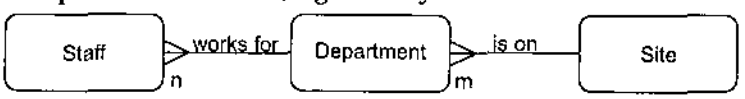


Figure 16. Fan trap

A single site contains many departments and employs many staff. However, which staff work in a particular department?

The fan trap is resolved by restructuring the original ER model to represent the correct association.

Figure 17. Resolved fan trap

Chasm traps

A chasm trap occurs when a model suggests the existence of a relationship between entity types, but the pathway does not exist between certain entity occurrences.

It occurs where there is a relationship with partial participation, which forms part of the pathway between entities that are related.

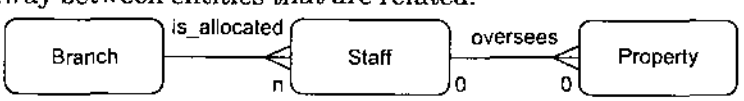


Figure 18. Chasm trap

- A single branch is allocated many staff who oversee the management of properties for rent. Not all staff oversee property and not all property is managed by a member of staff.
- What properties are available at a branch?
- The partial participation of Staff and Property in the oversees relation means that some properties cannot be associated with a branch office through a member of staff.
- We need to add the missing relationship which is called 'has' between the Branch and the Property entities.
- You need to therefore be careful when you remove relationships which you consider to be redundant.

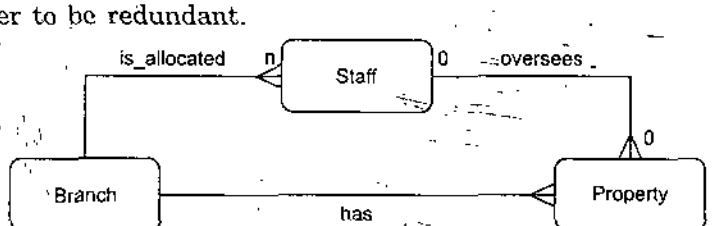


Figure 19. Resolved chasm trap

STUDENT ACTIVITY

1. What are the drawbacks of the E-R model?

2. What led to the development of the enhanced E-R model?

3. What are the additional concepts included in the EER model?

4. What are superclass entity types?

NOTES

SUMMARY

- The basic E-R model was able to handle the data modeling of the common business problems and was quite adequate for the representation of the majority of the business problems that existed in the 1970s and in the early 1980s. In the 1980s, there was a rapid increase in the development of many new database applications. The basic E-R modeling concepts were no longer sufficient to represent the requirements of these newer and complex applications. This necessitated the database designers and practitioners to search for and develop additional semantic modeling concepts. The most popular was the one that incorporated these semantic concepts into the existing E-R model. The E-R model that is supported with the additional semantic concepts is called the Enhanced Entity-Relationship model or the EER model.
- One of the most important new modeling constructs that were incorporated into the EER model was the superclass and subclass entity types. This feature allows us to model a general entity (superclass) and then subdivide it into several specialized entity types (subclass). EER diagrams are used to capture business rules such as constraints in the supertype/subtype relationships. Thus a superclass is an entity type that includes distinct subclasses that require to be represented in a data model. A subclass is an entity type that has a distinct role and is also a member of a superclass. An entity in a subclass must possess not only the values for its own attributes, but also values for its attributes as a member of the superclass. Attribute inheritance is the property by which subclass entities inherit values of all the attributes of the superclass entity. This feature makes it unnecessary to associate the superclass attributes with the subclass entities thus avoiding redundancy.
- The EER model includes all the concepts of the original E-R model together with the following additional concepts: Specialization, Generalization and Categorization. Generalization is the process of defining a more general entity type from a set of more specialized entity types. The modeling of a single subclass with a relationship that involves than one distinct superclass is called categorization.

TEST YOURSELF

1. How are superclass and subclass entity types represented in an EER diagram?
2. What are the reasons for introducing the concepts of superclasses and subclasses into and E-R model?
3. What is attribute inheritance?
4. How is attribute inheritance represented in the EER model?
5. What is relationship inheritance?
6. How is relationship inheritance represented in the EER model?
7. What is a type hierarchy?
8. What is a shared subclass?
9. What is multiple inheritance?
10. How are type hierarchies, shared subclasses and multiple inheritances represented in the EER model?
11. What do you mean by specialization? Explain with examples.
12. What is generalization? Give examples.

NOTES

13. What are specialization/generalization constraints?
14. What is a disjoint constraint? Explain with examples.
15. What is an overlapping constraint? Explain with examples.
16. What is participation constraint? Explain with examples.
17. What do you mean by selective participation?
18. What is selective inheritance?
19. How are specialization/generalization constraints represented in the EER model?
20. What is categorization?
21. What are subclass entity types?
22. How are superclass and subclass entity types related?

Fill in the Blanks

1. _____ developed the E-R model in the 1970s.
2. The E-R model that is supported with the additional semantic concepts is called the_____.
3. The _____ feature allows us to model a general entity and then subdivide it into several specialized entity types.
4. _____ is an entity type that includes distinct subclasses that require to be represented in a data model.
5. _____ is an entity type that has a distinct role and is also member of a superclass.
6. The_____ relationships add more semantic content and information to the design.
7. _____ is the property by which subclass entities inherit values of all the attributes of the superclass entity.
8. _____ makes it unnecessary to associate the superclass attributes with the subclass entities thus avoiding redundancy.
9. The subclasses inherit the relationships of the superclass. This property is_____.
10. An entity and its subclasses and their subclasses and so on are called a_____.
11. A subclass with more than one superclass is called a _____.
12. _____ is the process of maximizing the differences between members of an entity by identifying the distinguishing and unique characteristics (or attributes) of each member.
13. _____ is the process of defining a more general entity type from a set of move specialized entity types.
14. _____ is the process of minimizing the differences between the entities by identifying the common features.
15. The process of generalization can be seen as a reverse of the _____ process.
16. Specialization and generalization are valuable techniques for developing _____ relationships.
17. _____ in specialization and generalization allow us to capture some of the important business rules that apply to the relationships.
18. The constraints that are applicable to specialization and generalization are _____ and _____.
19. The _____ specifies that if the subclasses of a specialization/generalization are disjoint then an entity can be a member of only one of the subclasses of that specialization/generalization.
20. If the subclasses of a specialization/generalization are not disjoint then an entity may be a member of more than one subclass of the specialization/generalization and such a constraint is called an _____.

21. A specialization/generalization with _____ means that every entity in the superclass must be a member of a subclass in the specialization/generalization.
22. A specialization/generalization with a _____ specifies that an entity need not belong to any of the subclasses of a specialization/ generalization.
23. The modeling of a single subclass with a relationship that involves more than one distinct superclass is called _____.
24. A subclass having more than one superclass is called a _____ and the process of defining a category is called _____.
25. _____ means that the category will inherit only the attributes of one of the superclasses at a time.
26. For _____, the constraint is removed so that every occurrence of all the superclasses need not appear in the category.

NOTES

True or False

1. Enhanced Entity-Relationship model is also known as EER model.
2. The EER model does not included all the concepts of the original E-R model.
3. Each member of the subtype is also a member of the superclass.
4. A subclass need not always be a member of its superclass.
5. A superclass need not be consisted entirely of entities from the subclass alone.
6. A subclass is not an entity on its own right.
7. A subclass cannot have its own subclasses.
8. Generalization is a bottom-up approach, just opposite to the specialization approach.
9. If the subclasses of a specialization/generalization are not disjoint then an entity may be a member of more than one subclass of the specialization/ generalization.
10. The disjoint constraint is represented by placing a 'o' in the circle that connects the subclasses to the superclass.
11. A non-disjoint constraint is also called an overlapping constraint.
12. To represent an non-disjoint specialization/generalization an 'n' is placed inside the circle that connects the subclasses to the superclass.
13. The participation constraint can be total or partial.
14. To represent a total participation single (double) line is used to connect the superclass and the specialization/generalization circle.
15. A partial participation is represented using a single line between the superclass and the specialization/generalization circle.
16. Disjoint and participation constraints are not independent.
17. For total participation every occurrence of all the superclasses must appear in the category.
18. A category can be subdivided based on total or partial participation.

Multiple Choice

1. When did Chen invent the E-R model?
   - ☐ 1950
   - ☐ 1960
   - ☐ 1970
   - ☐ None of the above
2. What is the full from of CASE?
   - ☐ Computer Assisted Simulation Engine
   - ☐ Common Algorithm for Simulation and Evaluation
   - ☐ Computer Aided Software Engineering
   - ☐ None of the above

NOTES

3. EER stand for
   - ❒ Effective ER
   - ❒ Extended ER
   - ❒ Enchanced ER
   - ❒ None of the above
4. The subclasses are also connected to the circle by _____
   - ❒ Single lines
   - ❒ Double lines
   - ❒ Dotted lines
   - ❒ None of the above
5. Which is the feature that makes it unnecessary to associate the superclass attributes with the subclass entities thus avoiding redundancy?
   - ❒ Abstraction
   - ❒ Attribute inheritance
   - ❒ Categorization
   - ❒ None of the above
6. What is the principle by which the subclasses inherit the relationships of the superclass?
   - ❒ Abstraction
   - ❒ Relationship inheritance
   - ❒ Specialization
   - ❒ None of the above
7. An entity and its subclasses and their subclasses and so on are called a _____
   - ❒ Inheritance
   - ❒ Type hierarchy
   - ❒ Range hierarchy
   - ❒ None of the above
8. What is a subclass with more than one superclass called?
   - ❒ Shared subclass
   - ❒ Derived subclass
   - ❒ Differential subclass
   - ❒ None of the above

CHAPTER 3 RELATIONAL MODEL

NOTES

LEARNING OBJECTIVES

- Introduction
- RDBMS Terminology
- The Relational Data Structure
- Relational Algebra

INTRODUCTION

The foundations of Relational Database technology was laid by Dr. E.F. Codd, who in his paper 'A Relational Model of Data for Large Shared Data Banks' laid the basic principles of the RDBMS. Codd was working for IBM at their San Jose Research Lab in California. He laid down certain principles of database management, referred to as relational model. These principles were soon applied to experimental systems, and a start was made on the design of a database language that would interact with such systems.

In 1974, Chamberlain and Boyce, also from IBM San Jose, read a paper at a database workshop at the University of Ann Arbor, Michigan. This paper introduced the database language SEQUEL, which was implemented by IBM in 1974 - 75 as the prototype language SEQUEL-XRM.

The first attempt at a larger scale implementation of Codd's relational model was IBM'S System R. This system used a revised version of the SEQUEL called SEQUEL/2. In 1977, System R became operational and SEQUEL became SQL.

System R was a success and relational ideas became accessible to general computer users. This was mainly due to the writings of C.J. Date (again from IBM), who in his book 'An Introduction to Database Systems' provided a clear and readable introduction.

At present there are many implementations of the relational technology. DB2, ORACLE data, MS-SQL Server, MS-Access, Ingress etc. are some among them. Relational systems are now available in all sizes and shapes and for all sizes of computers.

RDBMS TERMINOLOGY

The relational model is an abstract theory of data that is based on the mathematical theory whose principles were laid down by Dr. E.F. Codd. The relational model of Codd used certain principles, which were not familiar in the data processing circles at that time. The terms were used to describe the database properties and functions lacked the precision necessary for formal theory that Codd was proposing. So a new set of terminology had to be evolved. The table 1 gives a list of the relational term and their corresponding informal equivalent(s).

NOTES

The Relational Database Management Systems, as said above, are based on the relational model. The relational model, in turn, is a way of looking at data. It is a prescription for how to present and manipulate data. More precisely, the relational model is concerned with three aspects of data; data structure, data integrity and data manipulation.

Table 1. RDBMS Terminology

Formal Relational term	Informal Equivalent(s)
Relation	Table
Tuple	Row, record
Cardinality	Number of rows
Attribute	Column, field
Degree	Number of columns
Primary key	Unique identifier
Domain	Set of legal values

THE RELATIONAL DATA STRUCTURE

The smallest unit of data in the relational model is the individual data value. Such values assumed to be atomic, which means that they have no internal structure as far as the model is concerned. A domain is a set of all possible data values. For example, in the supplier-parts tuple, the domain of supplier numbers is the set of all valid supplier numbers. Thus domains are sets of values, from which the actual values appearing in the attributes (columns) are drawn. The domain concept is a very important and integral part of thc Relational Model for operations such as joins, unions, etc., that directly or indirectly involves such comparisons. (Joins and Unions will be discussed in detail later, but, both of them are methods of combining data in more than one table.) If two attributes have their values from the same domain, then comparisons involving those two attributes means we are comparing like with like.

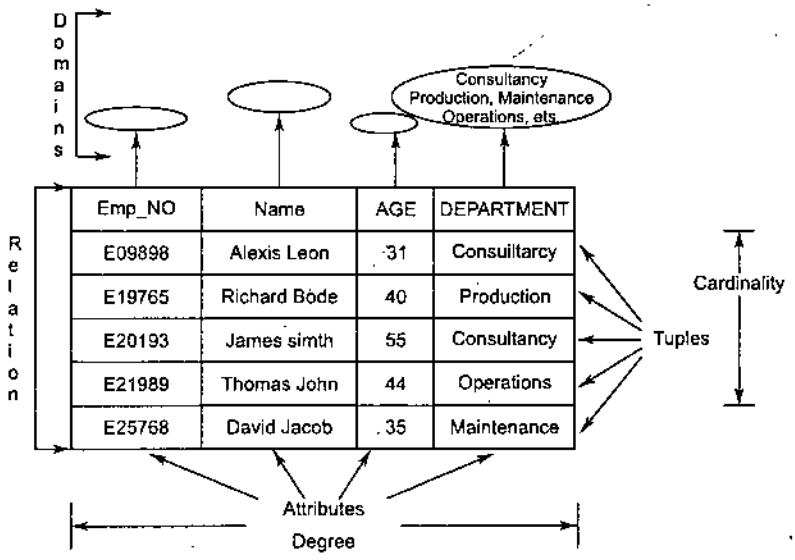
Domains are conceptual in nature. They may or may not be explicitly stored in a database sets of values. But they should be specified as part of database definition and then each definition should include a reference to the corresponding domain. Now let us take a look at the relations. A relation on domains, say D1, D2.....Dn consists of a heading and a body. The heading consists of a fixed set of distinct attributes, say A1, A2....An, such that each attribute 'Ai' corresponds to exactly one of the underlying domains 'Di'. The body consists of a time-varying set of tuples, where each tuple in turn consists of a set of attribute value pairs (Ai:Vi), one such pair for each attribute Ai in the heading. For any given attribute-value pair (Ai:vi). 'vi' is a value from the unique domain Di that is associated with the attribute Ai. We will explain this based on the following example (see Table 2):

Table 2. Employee Table

Employee Table			
Emp_No	Name	Age	Department
E09898	Alexis Leon	31	Consultancy
E19765	Richard Bode	40	Consultancy
E20193	James Smith	55	Production
E21989	Thomas John	44	Operations
E25678	David Jacob	35	Maintenance

Let us see how the employee relation (EMPLOYEE_TABLE) measures up to this definition. The underlying domains are the domain of employee numbers (say D1), the domain of employee names (say D2), domain of employee age values (D3) and the domain of employee department names (D4). The heading of the EMPLOYEE_TABLE consists of attributes EMP_NO (underlying domain D1), NAME (domain D2, AGE (domain D3) and DEPARTMENT (domain D4).

NOTES



Relational data structure

The body of EMPLOYEE_TABLE consists of a set of tuples and each tuple consists of a set of 4 attribute-value pairs, one such pair for each of the four attributes in the heading. Even though, they are used interchangeably, a table and a relation are not really the same thing. For sample the rows in a table have an ordering, that is from top to bottom, similarly the columns from left to right. But the tuples and attributes of a relation, which are mathematical sets, do not have any ordering.

The number of attributes in a relation is called the degree of the relation. A relation of degree one is unary, a relation of degree two is called binary, etc. So the employee relation has a degree of 4. The number of tuples or rows in a relation is called the cardinality of the relation. The cardinality of the relation EMPLOYEE_TABLE is 5. The cardinality of a relation changes as more and more tuples get added or deleted, but the degree does not.

Integrity Constraints

From the above discussion of the relational data structure, it is evident that most of the relations have an attribute, which can uniquely identify each tuple in the relation. In some cases there can be more than one attribute, which can uniquely identify each tuple in the relation. This attribute is called the candidate key. In other words, a candidate key is an attribute that can uniquely identify a row in a table. Consider the Table 3:

In Table 3, the attributes symbol, name and atomic number can uniquely identify each candidate key, or the ELEMENT_TABLE has three candidate keys. Since body of a relation is a set and sets by definition do not contain duplicate elements, it follows that at any given time no two tuples (or rows) of a relation can be

NOTES

duplicates of each other (or in other words no two rows can be the same). Let R be the relation with attributes A1, A2,...An. Set of attributes K = (Ai, Aj,...An) of R is said to be a candidate key of R if and only if the following two properties are satisfied:

Table 3. ELEMENT Table

ELEMENT-TABLE				
Symbol	Name	Atomic Number	Melting Point	Boiling Point
Al	Aluminum	13	933	2792
Fe	Iron	26	1811	3134
Ni	Nickel	28	1728	3186
Cu	Copper	29	1357	3200
Ag	Silver	47	1235	2435
Au	Gold	79	1337	3129

- **Uniqueness** - At any given time, no two distinct tuples (rows) of R have the same value for Ai, the same value for Aj....and the same value for An.
- **Minimality** - No proper subset of the set (Ai, Aj,...An) has the uniqueness property.

Every relation has at least one candidate key, because at least the combination of all its attributes has the uniqueness property. In the case of base relations (relations of a base table), one candidate key is designated as the primary key and the remaining candidate keys are called candidate keys. For example in the ELEMENT_TABLE, the relation has three candidate keys. We choose any one of them as the primary key. So if we choose the symbol as the primary key, the name and atomic number become alternate keys. But there are no hard and fast rules on how to choose the primary key from the list of candidate keys; it is a matter of preference and convenience of the database designer.

The terms candidate keys and primary keys should not be abbreviated to just 'keys'. The term 'key' has too many meanings in the database world. In the relational model alone, there are candidate keys, primary keys, alternate keys, foreign keys, search keys, parent keys, encryption keys, decryption keys, and so on. So it is better to qualify the word 'key' with the appropriate title to avoid confusion. Consider the following two tables. One is the ELEMENT_TABLE (Table 3) and the other the SHIPMENT_TABLE (Table 4).

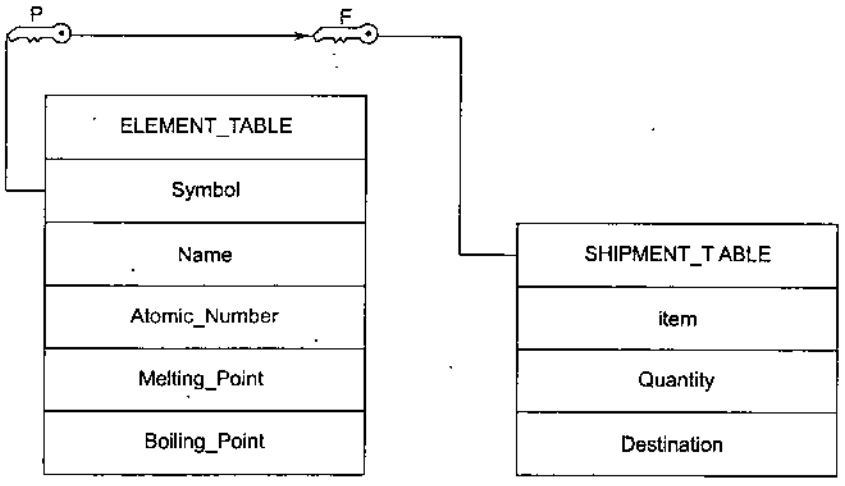
Table 4. SHIPMENT Table

Item	Quantity	Destination
Al	100	Delhi
Fe	500	Delhi
Au	5	Delhi
Al	150	Chennai
Ni	400	Kochi
Au	2	Calcutta
Ag	30	Mumbai

Let us take a look at the attribute 'Item' of relation SHIPMENT_TABLE. It is clear that a given value for that attribute, say item 'Au' should be permitted to

appear in the database only if the same value appears as a value of the primary key 'Symbol' in the relation ELEMENT_TABLE. Such an attribute is called a foreign key. Or in other words, a foreign key is an attribute or attribute combination of one relation (table) whose values are required to match those of the primary key of some other relation (table). Also the foreign key and the primary key should be defined on the same underlying domain. A pictorial representation of this is given in Figure.

NOTES



Primary Key - Foreign Key relationship

From the above discussions we are now able to identify many integrity rules (or constraints) for the relational model. Relational data model includes several types of constraints whose purpose is to maintain the accuracy and integrity of the data in the database. The major types of integrity constraints are:

- Domain Constraints
- Entity Integrity
- Referential Integrity
- Operational Constraints

Domain Constraints

All the values that appear in a column of a relation (table) must be taken from the same domain. As we have seen before, a domain is a set of values that may be assigned to an attribute. Domain definition usually consists of the following components:

- Domain name
- Meaning
- Data Type
- Size or Length
- Allowable values or Allowable range (if applicable)

For example, in the ELEMENT table, the domain for the column Symbol is a character of 2, and should be from the list of the elements. In other words, the domain of the column Symbol is a value whose maximum length is 2 characters and the first letter is in uppercase and is a value from the periodic table of elements. Similarly the domain of the column atomic number is an integer and so on.

NOTES

Entity Integrity

The entity integrity rule is designed to assure that every relation has a primary key, and the data values for that primary key are all valid. Entity integrity guarantees that every primary key attribute is non-null. No attribute participating in the primary key of a base relation is supposed to contain nulls. Primary key performs the unique identification function in a relational database. Thus a null primary key value within a base relation would be like saying that there was an entity that had no known identity. An entity that cannot be identified is a contradiction in itself hence the name entity integrity. In some cases, a particular attribute cannot be assigned a value. There are two situations where this is likely to occur:

- There is no applicable data value
- Applicable data value is not known when the values are assigned

For example, consider a situation where you are filling out your personal details. There is a column for fax number and you don't have a fax number. You will leave the field blank. This is an example of no applicable data value. In another case, suppose you are filling the ELEMENT column you do not know the melting point for Nickel. You know that Nickel has a melting point, you do not know the exact value at that point in time. So you leave that field blank since that information is not known at that point.

The relational model allows you to assign a null value to an attribute in the above-described situations. A null is a value that is assigned to an attribute when no other value applies, when the applicable value is unknown. In reality, a null is not a value, but rather the absence of value. For example, null is not the same as 0 (for numeric fields) or blank (for character fields). The inclusion of nulls in the relational model is somewhat controversial, since operations involving nulls sometimes leads to unpredictable results. On the other hand, using null for missing values is a good idea. But whatever the pros and cons of using null, it is imperative that the primary key values be non-null.

Referential Integrity

In the relational data model, associations between tables are defined using foreign keys. For example, in Figure 2, the association between the ELEMENT and the SHIPMENT tables is defined by including the Symbol attribute as a foreign key in the SHIPMENT table. This implies that before we insert a new row in the SHIPMENT table, the element for that order must already exist in the ELEMENT table. If you examine the rows in the SHIPMENT table, you will find that every item name in that table appears in the ELEMENT table.

A referential integrity constraint is a rule that maintains consistency among the rows of two tables (relations). The rule states that if there is a foreign key in one relation, either each foreign key value must match a primary key value in the other table or else the foreign key value must be null.

If base relation (table) includes a foreign key FK matching the primary key PK of some other base relation, then every value of FK in the first table must either be equal to the value of PK in some tuple (row) of the second table or be wholly null (that is each attribute value participating in that FK value must be null). Or in other words, a given foreign key value must have matching primary key value in

some tuple of the referenced relation if that foreign key value is non-null. Sometimes, it is necessary to permit foreign keys to accept nulls. Here it must be noted that the null are of the variety 'value does not exist' rather than 'value unknown'.

Operational Constraints

NOTES

These are the constraints enforced in the database by the business rules or real world limitations. For example, if the retirement age of the employees in an organization is 60, then the age column of the employee table can have a constraint "Age should be less than or equal to 60." These kinds of constraints, enforced by the business and the environment are called operational constraints.

Modifying the Database

The manipulative part of the relational model consists of a set of operators known collectively as the relational algebra together with relational assignment.

Codd's Rules

We have in the previous sections defined a relational database as a finite collection of relations and a relation in terms of domains, attributes, associations and tuples. According to the definitions, table structures are the sole building blocks for application modeling. Dr. E. F. Codd, the founder of the relational database systems, places the relational model's characteristics in three broad categories. First, structural features that support the view of the data. They include relations and their underlying components, views and queries, both mechanisms for creating virtual series. Second, integrity features such as entity and referential integrity and also application specific-constraints. Finally, data manipulation features for data retrieval, insertion, deletion and update. These features must be able to emulate any operation from relational algebra. Certain features such as outer joins and unions are also expected from a relational database system. Codd provides a set of 12 rules, which qualify a database product as relational. In other words, to merit the name 'fully relational', a database product should provide the three categories of features-structure, integrity and data manipulation—and it should abide by the Codd's rules. We will now see the Codd's rules in a little detail.

1. Information Rule

All information in a relational database including table names, column names is presented by values in tables. This simple view of data speeds design and learning process. User productivity is improved since knowledge of only one language is necessary to access all data such as description of the table and attribute definitions, integrity constraints. Action can be taken when the constraints are violated. Access to data can be restricted. All these information are also stored in tables.

2. Guaranteed Access Rule

Every piece of data in a relational database, can be accessed by using a combination of a tuple name, a primary key value that identifies the row and a column name, which identifies a cell. User productivity is improved since there is no need to resort to using physical pointers or addresses. It also provides data independence

NOTES

and makes it possible to retrieve each individual piece of data stored in a relational database by specifying the name of the table in which it is stored, the column and the primary key, which identifies the cell in which it is stored.

3. Systematic Treatment of Nulls Rule

The RDBMS handles records that have unknown or inapplicable values in a pre-defined fashion. Also, the RDBMS distinguishes between zeros, blanks and nulls in the records and handles such values in a consistent manner that produces correct answers, comparisons and calculations. Through the set of rules for handling nulls, users can distinguish results of the queries that involve nulls, zeros and blanks. Even though the rule doesn't specify what should be done in the case of nulls, it specifies that there should be a consistent policy in the treatment of nulls.

4. Active On-line Catalog Based on the Relational Model

The description of a database and its contents are database tables and therefore can be queried on-line via the data manipulation language. The Database Administrator's productivity is improved since the changes and additions to the catalog can be done with the same commands that are used to access any other table. All queries and reports can also be done as any other table.

5. Comprehensive Data Sub-language Rule

The RDBMS may support several languages. But at least one of them should allow the user to do all the following: define tables and views, query and update data, set integrity constraints, set authorizations and define transactions. User productivity is improved since there is just one approach that can be used for all database operations. In a multi-user environment the user does not have to worry about the data integrity and such things, which will be taken care of by the system. Also only users with proper authorization will be able to access data.

6. View Updating Rule

Any view that can be updated theoretically can be updated using the RDBMS. Data consistency is ensured since the changes made in the view is transmitted to the base-table and vice-versa.

7. High-Level Insert, Update and Delete

The RDBMS supports insertion, updating and deletion at a table level. The performance is improved since the commands act on a set of records rather than one record at a time.

8. Physical Data Independence

The execution of ad hoc requests and application programs is not affected by changes in the physical data access and storage methods. Database administrators can make changes to the physical access and storage method, which improve performance and do not require changes in the application programs or requests. Here the user specifies what he wants and need not worry about how the data is obtained.

NOTES

9. Logical Data Independence

Logical changes in tables and views such as adding/deleting columns or changing field lengths need not necessitate modifications in the programs or in the format of ad hoc requests. The database can change and grow to reflect changes in reality without requiring the user intervention or changes in the applications. For example adding an attribute or column to the base table should not disrupt the programs or the interactive commands that have no use for the new attribute.

10. Integrity Independence

Like table and view definitions, integrity constraints are stored in the on-line catalog and can therefore be changed without necessitating changes in the application programs. Integrity constraints specific to a particular Relational Database must be definable in the relational data sub-language and storable in the catalog. At least the entity integrity and referential integrity must be supported.

11. Distribution Independence

Application programs and ad hoc requests are not affected by changes in the distribution of data. This improves systems reliability since application programs will work even if the forms and data are moved to different sites.

12. Non-subversion Rule

If the RDBMS has a language that accesses the information of a record at a time, this language should not be used to bypass the integrity constraints. This is necessary for data integrity.

RELATIONAL ALGEBRA

A relation is a set of attributes with values for each attribute such that:

Each attribute value must be a single value only (atomic).

All values for a given attribute must be of the same type (or domain).

Each attribute name must be unique.

The order of attributes is insignificant

No two rows (tuples) in a relation can be identical.

The order of the rows (tuples) is insignificant.

Relational Algebra is a collection of operations on Relations.

Relations are operands and the result of an operation is another relation.

Two main collections of relational operators:

Set theory operations:

Union, Intersection, Difference and Cartesian product.

Specific Relational Operations:

Selection, Projection, Join, Division

STUDENT ACTIVITY-1

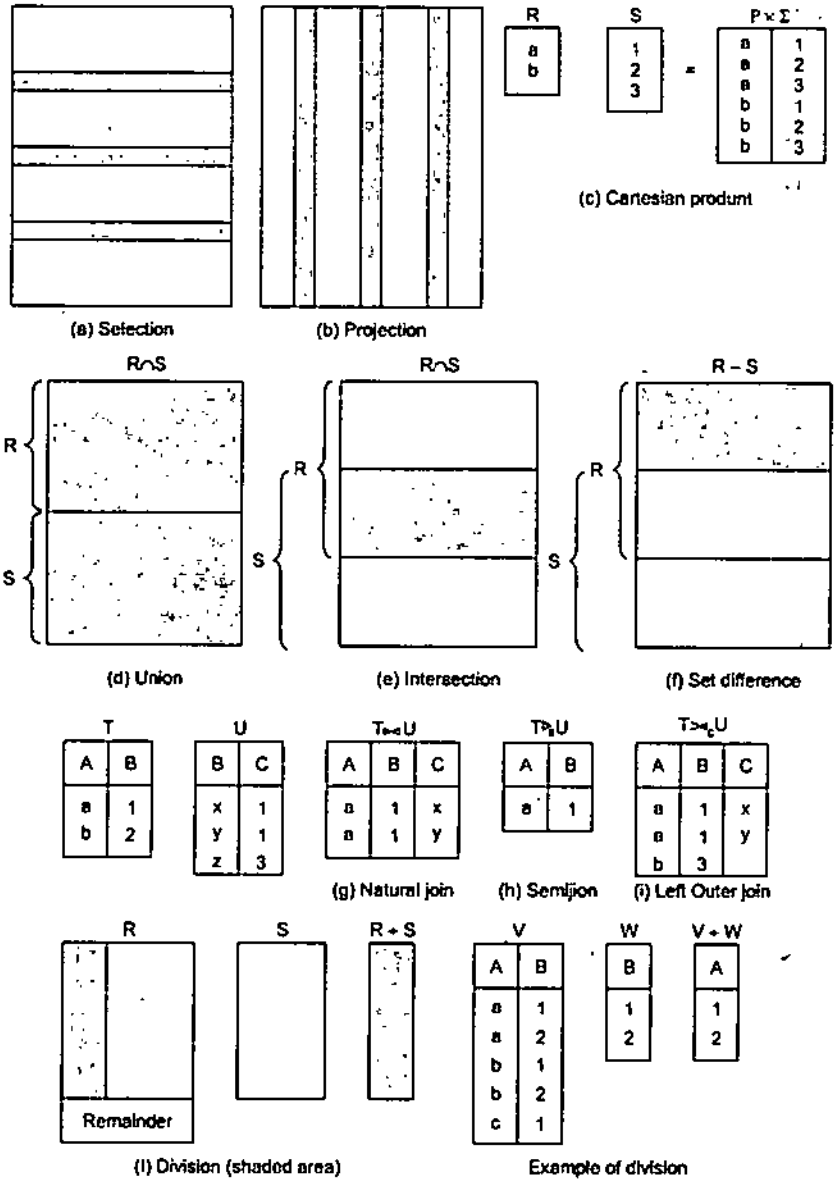
1. What is a relational database?

2. What are the advantages and disadvantages of a relational database system?

3. Describe the evolution of the relational database model.

4. Give the relational terminology for the commonly used database terms.

NOTES



Selection Operator

Selection and Projection are unary operators.

The selection operator is sigma: σ

The selection operation acts like a filter on a relation by returning only a certain number of tuples.

The resulting relation will have the same degree as the original relation.

The resulting relation may have fewer tuples than the original relation.

The tuples to be returned are dependent on a condition that is part of the selection operator.

σC (R) Returns only those tuples in R that satisfy condition C

A condition C can be made up of any combination of comparison or logical operators that operate on the attributes of R.

Comparison operators: = < > ≥ ≤ ≠

Logical operators: ∧ ∨ ¬

Use the Truth tables (memorize these) for logical expressions:

∧	T	F
T	T	F
F	F	F

∨	T	F
T	T	T
F	T	F

T	F
F	T

Selection Examples

Assume the following relation EMP has the following tuples:

Name	Office	Dept	Rank
Smith	400	CS	Assistant
Jones	220	Econ	Adjunct
Green	160	Econ	Assistant
Brown	420	CS	Associate
Smith	500	Fin	Associate

Select only those Employees in the CS department:

$$\sigma \text{Dept} = \text{'CS'} (\text{EMP})$$

Result:

Name	Office	Dept	Rank
Smith	400	CS	Assistant
Brown	420	CS	Associate

Select only those Employees with last name Smith who are assistant professors:

$$\text{Name} = \text{'Smith'} \wedge \text{Rank} = \text{'Assistant'} (\text{EMP})$$

Result:

Name	Office	Dept	Rank
Smith	400	CS	Assistant

Select only those Employees who are either Assistant Professors or in the Economics department:

$$\text{Rank} = \text{'Assistant'}\ \text{Dept} = \text{'Econ'} (\text{EMP})$$

Result:

Name	Office	Dept	Rank
Smith	400	CS	Assistant
Jones	220	Econ	Adjunct
Green	160	Econ	Assistant

Select only those Employees who are not in the CS department or Adjuncts:

$$\sigma\neg (\text{Rank} = \text{'Adjunct'} \vee \text{Dept} = \text{'CS'}) (\text{EMP})$$

Result:

Name	Office	Dept	Rank
Green	160	Econ	Assistant
Smith	500	Fin	Associate

List all staff with a salary greater than 10,000.

$$\sigma\ \text{salary} > 10000\ (\text{Staff})$$

NOTES

Staff No.	fName	lName	Position	Sex	DOB	Salary	Branch
SL1	John	White	Manager	M	1-Oct--45	30,000	B005
SG37	Ann	Beech	Assistant	F	10-Nov-60	12,000	B003
SG14	David	Ford	Supervisor	M	24-Mar-58	18,000	B003
SG5	Susan	Brand	Manager	F	3-Jun-40	24,000	B003

NOTES

Projection Operator

Projection is also a Unary operator.

The Projection operator is pi: π

Projection limits the attributes that will be returned from the original relation.

The general syntax is: $\pi_{attributes}$ R

Where attributes is the list of attributes to be displayed and R is the relation.

The resulting relation will have the same number of tuples as the original relation (unless there are duplicate tuples produced).

The degree of the resulting relation may be equal to or less than that of the original relation.

Projection Examples

Assume the same EMP relation above is used.

Project only the names and departments of the employees:

$\pi_{name, dept}$ (EMP)

Results:

Name	Dept
Smith	CS
Jones	Econ
Green	Econ
Brown	CS
Smith	Fin

Combining Selection and Projection

The selection and projection operators can be combined to perform both operations.

Show the names of all employees working in the CS department:

$$\pi_{name}(\sigma_{Dept = 'CS'}(EMP))$$

Results:

Name
Smith
Brown

Show the name and rank of those Employees who are not in the CS department or Adjuncts:

$$\pi_{name, rank}(\sigma_{\neg(Rank = 'Adjunct' \vee Dept = 'CS')}(EMP))$$

Result:

Name	Rank
Green	Assistant
Smith	Associate

- Produce a list of salaries for all staff, showing only staffNo, fName, lName, and salary details.

ΠstaffNo, fName, lName, salary(Staff)

Staff No.	fName	lName	Salary
SL21	John	White	30,000
SG37	Ann	Beech	12,000
SG14	David	Ford	18,000
SA9	Mary	Howe	9,000
SG5Susan	Brand	24,000	
SL41	Julie	Lee	9,000

NOTES

Set Theoretic Operations

Consider the following relations **R** and **S**

R

First	Last	Age
Bill	Smith	22
Sally	Green	28
Mary	Keen	23
Tony	Jones	32

S

First	Last	Age
Forrest	Gump	36
Sally	Green	28
DonJuan	DeMarco	27

Union: R∪S

Result: Relation with tuples from R and S with duplicates removed.

Difference: R - S

Result: Relation with tuples from R but not from S

Intersection: R S

Result: Relation with tuples that appear in both R and S.

R∩S

First	Last	Age
Bill	Smith	22
Sally	Green	28
Mary	Keen	23
Tony	Jones	32
Forrest	Gump	36
DonJuan	DeMarco	27

NOTES

R - S

First	Last	Age
Bill	Smith	22
Mary	Keen	23
Tony	Jones	32

First	Last	Age
Sally	Green	28

Union Compatible Relations

Attributes of relations need not be identical to perform union, intersection and difference operations.

However, they must have the same number of attributes or arity and the domains for corresponding attributes must be identical.

Domain is the datatype and size of an attribute.

The degree of relation R is the number of attributes it contains.

Definition: Two relations R and S are union compatible if and only if they have the same degree and the domains of the corresponding attributes are the same.

Some additional properties:

Union, Intersection and difference operators may only be applied to Union Compatible relations.

Union and Intersection are commutative operations

$$R \cap S = S \cap R$$

$$R \cap S = S \cap R$$

Difference operation is NOT commutative.

R - S not equal S - R

The resulting relations may not have meaningful names for the attributes. Convention is to use the attribute names from the first relation.

Cartesian Product

Produce all combinations of tuples from two relations.

R

First	Last	Age
Bill	Smith	22
Mary	Keen	23
Tony	Jones	32

S

Dinner	Dissert
Steak	Ice Cream
Lobster	Cheesecake

First	Last	Age	Dinner	Dessert
Bill	Smith	22	Steak	Ice Cream
Bill	Smith	22	Lobster	Cheesecake
Mary	Keen	23	Steak	Ice Cream
Mary	Keen	23	Lobster	Cheesecake
Tony	Jones	32	Steak	Ice Cream
Tony	Jones	32	Lobster	Cheesecake

NOTES

Example - Cartesian product and Selection

- Use selection operation to extract those tuples where Client.client No = *Viewing.clientNo.*

  sClient.clientNo = Viewing.clientNo((ÕclientNo, fName, lName(Client)) C (ÕclientNo, propertyNo, comment(Viewing)))

client.client No.	fName	lName	Viewing.clientNo	propertyNo	comment
CR76	John	Kay	CR76	PG4	too remote
CR56	Aline	Stewart	CR56	PA14	too small
CR56	Aline	Stewart	CR56	PG4	
CR56	Aline	Stewart	CR56	PG36	
CR62	Mary	Tregear	CR62	PA14	no dining room

Cartesian product and Selection can be reduced to a single operation called a Join.

Join Operation

Join operations bring together two relations and combine their attributes and tuples in a specific fashion.

Various forms of join operation

Theta join

Equijoin (a particular type of Theta join)

Natural join

Outer join

Semijoin

The generic join operator (called the Theta Join is:

It takes as arguments the attributes from the two relations that are to be joined.

For example assume we have the EMP relation as above and a separate DEPART relation with (Dept, MainOffice, Phone) :

EMP EMP.Dept = DEPART.Dept DEPART

The join condition can be

When the join condition operator is = then we call this an Equijoin

Note that the attributes in common are repeated.

Join Examples

Assume we have the EMP relation from above and the following DEPART relation:

Dept	Mainoffice	Phone
CS	404	555-1212
Econ	200	555-1234
Fin	501	555-4321
Hist	100	555-9876

NOTES

- Find all information on every employee including their department info:

EMP emp.Dept = depart.Dept DEPART

Results:

Name	Office	EMP.Dept	Salary	DEPART.Dept	MainOffice	Phone
Smith	400	CS	45000	CS	404	555-1212
Jones	220	Econ	35000	Econ	200	555-1234
Green	160	Econ	50000	Econ	200	555-1234
Brown	420	CS	65000	CS	404	555-1212
Smith	500	Fin	60000	Fin	501	555-4321

- Find all information on every employee including their department info where the employee works in an office numbered less than the department main office:

EMP (emp.office < depart.mainoffice)∧(emp.dept = depart.dept) DEPART

Results:

Name	Office	EMP.Dept	Salary	DEPART.Dept	MainOffice	Phone
Smith	400	CS	45000	CS	404	555-1212
Green	160	Econ	50000	Econ	200	555-1234
Smith	500	Fin	60000	Fin	501	555-4321

Natural Join

- Notice in the generic (Theta) join operation, any attributes in common (such as dept above) are repeated.
- The Natural Join operation removes these duplicate attributes.
- The natural join operator is: *
- We can also assume using * that the join condition will be = on the two attributes in common.
- Example: EMP * DEPART

Results:

Outer Join

In the Join operations so far, only those tuples from both relations that satisfy the join condition are included in the output relation.

Name	Office	Dept	Salary	MainOffice	Phone
Smith	400	CS	45000	404	555-1212
Jones	220	Econ	35000	200	555-1234
Green	160	Econ	50000	200	555-1234
Brown	420	CS	65000	404	555-1212
Smith	500	Fin	60000	501	555-4321

The Outer join includes other tuples as well according to a few rules.

Three types of outer joins:

Left Outer Join includes all tuples in the left hand relation and includes only those matching tuples from the right hand relation.

Right Outer Join includes all tuples in the right hand relation and includes only those matching tuples from the left hand relation.

Full Outer Join includes all tuples in the left hand relation and from the right hand relation.

NOTES

Examples:

Assume we have two relations: PEOPLE and MENU:

PEOPLE:

Name	Age	Food
Alice	21	Hamburger
Bill	24	Pizza
Carl	23	Beer
Dina	19	Shrimp

MENU:

Food	Day
Pizza	Monday
Hamburger	Tuesday
Chicken	Wednesday
Pasta	Thursday
Tacos	Friday

- PEOPLE people.food = menu.food MENU (Left Outer Join)

Name	Age	people.Food	menu.Food	Day
Alice	21	Hamburger	Hamburger	Tuesday
Bill	24	Pizza	Pizza	Monday
Carl	23	Beer	NULL	NULL
Dina	19	Shrimp	NULL	NULL

- PEOPLE people.food = menu.food MENU (Right Outer Join)

Name	Age	people.Food	menu.Food	Day
Bill	24	Pizza	Pizza	Monday
Alice	21	Hamburger	Hamburger	Tuesday
NULL	NULL	NULL	Chicken	Wednesday
NULL	NULL	NULL	Pasta	Thursday
NULL	NULL	NULL	Tacos	Friday

- PEOPLE people.food = menu.food MENU (Full Outer Join)

Name	Age	people.Food	menu.Food	Day
Alice	21	Hamburger	Hamburger	Tuesday
Bill	24	Pizza	Pizza	Monday
Carl	23	Beer	NULL	NULL
Dina	19	Shrimp	NULL	NULL
NULL	NULL	NULL	Chicken	Wednesday
NULL	NULL	NULL	Pasta	Thursday
NULL	NULL	NULL	Tacos	Friday

NOTES

Outer Union

- The Outer Union operation is applied to partially union compatible relations.
- Operator is: ∪*
- Example: PEOPLE ∪* MENU

Name	Age	Food	Day
Alice	21	Hamburger	NULL
Bill	24	Pizza	NULL
Carl	23	Beer	NULL
Dina	19	Shrimp	NULL
NULL	NULL	Hamburger	Monday
NULL	NULL	Pizza	Tuesday
NULL	NULL	Chicken	Wednesday
NULL	NULL	Pasta	Thursday
NULL	NULL	Tacos	Friday

Rename Operator

The rename operator returns an existing relation under a new name. ?A(B) is the relation B with its name changed to A. For example, find the employees in the same Department as employee 3.

$\rho_{emp2.surname,emp2.forenames}$(

$\sigma_{employee.empno = 3 \wedge employee.depno = emp2.depno}$(

employee × (ρ_{emp2}employee)

)

)

Division

- R ÷ S
  - Defines a relation over the attributes C that consists of set of tuples from R that match combination of every tuple in S.
- Expressed using basic operations:
  T1 → PC(R)
  T2 → PC((S X T1) – R)
  T → T1 – T2
- Identify all clients who have viewed all properties with three rooms.
  ($\Pi_{clientNo,propertyNo}$(Viewing)) ÷ ($\Pi_{propertyNo}$($\sigma_{rooms = 3}$ (PropertyForRent)))

client No	property No
CR56	PA14
CR76	PG4
CR56	PG4
CR62	PA14
CR56	PG36

propery No
PG4
PG36

client No
CR56

Consider the following SQL to find which departments have had employees on the 'Further Accounting' course.

SELECT DISTINCT dname

FROM department, course, empcourse, employee

WHERE cname = `Further Accounting'

AND course.courseno = empcourse.courseno

AND empcourse.empno = employee.empno

AND employee.depno = department.depno;

The equivalent relational algebra is

$PROJECT_{dname}$ (department $JOIN_{depno = depno}$ (

$PROJECT_{depno}$ (employee $JOIN_{empno = empno}$ (

$PROJECT_{empno}$ (empcourse $JOIN_{courseno = courseno}$ (

$PROJECT_{courseno}$ ($SELECT_{cname = `Further\ Accounting'}$ course)

))

))

))

Symbolic Notation

From the example, one can see that for complicated cases a large amount of the answer is formed from operator names, such as PROJECT and JOIN. It is therefore commonplace to use symbolic notation to represent the operators.

- SELECT ->σ (sigma)
- PROJECT -> π(pi)
- PRODUCT -> ×(times)
- JOIN -> |×| (bow-tie)
- UNION -> ∪ (cup)
- INTERSECTION -> ∩(cap)
- DIFFERENCE -> - (minus)
- RENAME ->ρ(rho)

Usage

The symbolic operators are used as with the verbal ones. So, to find all employees in department 1:

$$SELECT_{depno = 1}(employee)$$

becomes $\sigma_{depno = 1}(employee)$

Conditions can be combined together using ^ (AND) and v (OR). For example, all employees in department 1 called 'Smith':

$$SELECT_{depno = 1\ \wedge\ surname = `Smith'}(employee)$$

becomes $\sigma_{depno = 1\ \wedge\ surname = `Smith'}(employee)$

The use of the symbolic notation can lend itself to brevity. Even better, when the JOIN is a natural join, the JOIN condition may be omitted from |x|. The earlier example resulted in:

$PROJECT_{dname}$ (department $JOIN_{depno = depno}$ (

$PROJECT_{depno}$ (employee $JOIN_{empno = empno}$ (

$PROJECT_{empno}$ (empcourse $JOIN_{courseno = courseno}$ (

$PROJECT_{courseno}$ ($SELECT_{cname = `Further\ Accounting'}$ course)))))))

becomes

π_{dname} (department |×| (

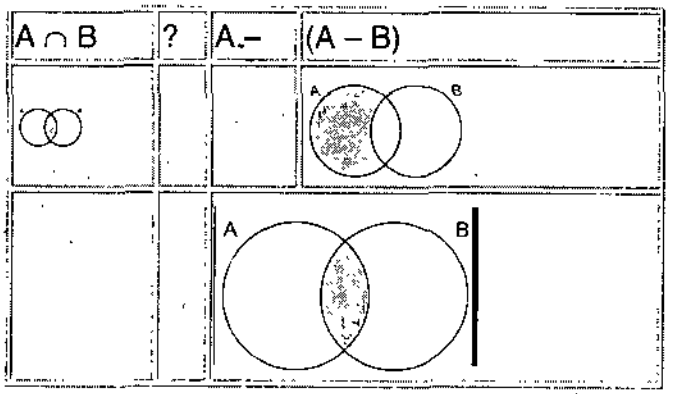
NOTES

NOTES

π_{depno} (employee |×| (
π_{empno} (empcourse |×| (
π_{courseno} ($\sigma_{\text{cname = 'Further Accounting'}}$ course)))))))

Derivable Operators

Fundamental operators:s, π, ×, ?, –, ρ
Derivable operators: |×|, ∩



Equivalence

$$A |\times|_c B ? \pi_{a1, a2, \ldots aN}(\sigma_c(A \times B))$$

- where c is the join condition (eg A.a_1 = B.a_1),
- and a_1,a_2,...aN are all the attributes of A and B without repetition.

c is called the join-condition, and is usually the comparison of primary and foreign key. Where there are N tables, there are usually N–1 join-conditions. In the case of a natural join, the conditions can be missed out, but otherwise missing out conditions results in a cartesian product (a common mistake to make).

Equivalences

The same relational algebraic expression can be written in many different ways. The order in which tuples appear in relations is never significant.

$A \times B \cup B \times A$
$A \cap B \cup B \cap A$
$A \cup B \cup B \cup A$
(A – B) is not the same as (B – A)
$\sigma c_1 (\sigma c_2(A)) ? \sigma c_2 (\sigma c_1(A)) ? \sigma c_1 \wedge c_2(A)$
$\pi_{a1}(A) ? \pi a_1(\pi a_{1,\text{etc}}(A))$

where etc represents any other attributes of A many other equivalences exist.

While equivalent expressions always give the same result, some may be much easier to evaluate that others.

When any query is submitted to the DBMS, its query optimiser tries to find the most efficient equivalent expression before evaluating it.

Aggregate Functions

We can also apply Aggregate functions to attributes and tuples:

SUM
MINIMUM
MAXIMUM

AVERAGE, MEAN, MEDIAN
COUNT

Aggregate functions are sometimes written using the Projection operator or the Script F character: $\mathfrak{F}$.

Aggregate Function Examples

Assume the relation EMP has the following tuples:

Name	Office	Dept	Salary
Smith	400	CS	45000
Jones	220	Econ	35000
Green	160	Econ	50000
Brown	420	CS	65000
Smith	500	Fin	60000

NOTES

Find the minimum Salary: $\mathfrak{F}$ MIN (salary) (EMP)

Results:

MIN(salary)
35000

Find the average Salary: $\mathfrak{F}$ AVG (salary) (EMP)

Results:

AVG(salary)
51000

Count the number of employees in the CS department: COUNT $\mathfrak{F}$ (name) ($\sigma_{Dept = 'CS'}$ (EMP))

Results:

Count (Name)
2

Find the total payroll for the Economics department: $\mathfrak{F}_{SUM\ (salary)}$ ($\sigma_{Dept\ =\ 'Econ'}$ (EMP))

Results:

SUM(salary)
85000

Comparing Relational Algebra and SQL

Relational algebra:

- is closed (the result of every expression is a relation)
- has a rigorous foundation
- has simple semantics
- is used for reasoning, query optimisation, etc.

SQL:

- is a superset of relational algebra
- has convenient formatting features, etc.

NOTES

- provides aggregate functions
- has complicated semantics
- is an end-user language.

Relational Calculus

- If predicate contains a variable (*e.g.*, '*x* is a member of staff'), there must be a range for *x*.
- When we substitute some values of this range for *x*, proposition may be true; for other values, it may be false.
- When applied to databases, relational calculus has forms: tuple and domain.

Tuple Relational Calculus

Interested in finding tuples for which a predicate is true. Based on use of **tuple variables.**

Tuple variable is a variable that 'ranges over' a named relation: *i.e.*, variable whose only permitted values are tuples of the relation.

Specify range of a tuple variable S as the Staff relation as:

Staff(S)

To find set of all tuples S such that P(S) is true:

{S | P(S)}

Tuple Relational Calculus - Example

To find details of all staff earning more than $10,000:

{S | Staff(S) ∧ S.salary > 10000}

To find a particular attribute, such as salary, write:

{S.salary | Staff(S) ∧ S.salary > 10000}

Can use two quantifiers to tell how many instances the predicate applies to:

Existential quantifier $ ('there exists')

Universal quantifier " ('for all')

Tuple variables qualified by " or $ are called bound variables, otherwise called free variables.

Existential quantifier used in formulae that must be true for at least one instance, such as:

Staff(S) ∧ ($B)(Branch(B) ∧

(B.branchNo = S.branchNo) ∧ B.city = 'London')

- Means 'There exists a Branch tuple with same branchNo as the branchNo of the current Staff tuple, S, and is located in London'.
- Universal quantifier is used in statements about every instance, such as: ("B) (B.city ≠ 'Paris')
- Means 'For all Branch tuples, the address is not in Paris'.
- Can also use ~($B) (B.city = 'Paris') which means 'There are no branches with an address in Paris'.
- Formulae should be unambiguous and make sense.
- A (well-formed) formula is made out of atoms:
- R(S*i*), where Si is a tuple variable and R is a relation
- S*i*.a_1 *q* Sj.a_2

- Si.a_1 q c
- Can recursively build up formulae from atoms:
- An atom is a formula
- If F_1 and F_2 are formulae, so are their conjunction, $F_1 \wedge F_2$; disjunction, $F_1 \wedge F_2$; *and negation*, $\sim F_1$
- If F is a formula with free variable X, then ($X)(F) and ("X)(F) are also formulae.

NOTES

Examples :

List the names of all managers who earn more than $25,000.

{S.fName, S.lName | Staff(S) ∧

S.position = 'Manager' ∧ S.salary > 25000}

List the staff who manage properties for rent in Glasgow.

{S | Staff(S) ∧ ($P) (PropertyForRent(P) ∧ (P.staffNo = S.staffNo) ∧ P.city = 'Glasgow')}

List the names of staff who currently do not manage any properties.

{S.fName, S.lName | Staff(S) ∧ (~($P) (PropertyForRent(P)∧(S.staffNo = P.staffNo)))}

Or

{S.fName, S.lName | Staff(S) ∧ ((∀P) (~PropertyForRent(P) ∨ ~(S.staffNo = P.staffNo)))}

List the names of clients who have viewed a property for rent in Glasgow.

{C.fName, C.lName | Client(C) ∧ (($V)($P)

(Viewing(V) ∧ PropertyForRent(P) ∧

(C.clientNo = V.clientNo) ∧

(V.propertyNo=P.propertyNo)∧P.city ='Glasgow'))}

Expressions can generate an infinite set. For example: {S | ~Staff(S)}

To avoid this, add restriction that all values in result must be values in the domain of the expression.

Domain Relational Calculus

Uses variables that take values from *domains* instead of tuples of relations.

If $F(d_1, d_2, \ldots, d_n)$ stands for a formula composed of atoms and $d_1, d_2, \ldots, d_n$ represent domain variables, then:

$$\{d_1, d_2, \ldots, d_n \mid F(d_1, d_2, \ldots, d_n)\}$$

is a general domain relational calculus expression.

Example - Domain Relational Calculus

Find the names of all managers who earn more than $25,000.

{fN, lN | ($sN, posn, sex, DOB, sal, bN)

(Staff (sN, fN, lN, posn, sex, DOB, sal, bN) ∧

posn = 'Manager' ∧ sal > 25000)}

List the staff who manage properties for rent in Glasgow.

{*sN, fN, lN, posn, sex*, DOB, *sal*, bN |

($sN1,cty)(Staff(sN,fN,lN,posn,sex,DOB,sal,bN) ∧

PropertyForRent(pN, st, cty, pc, typ, rms,

rnt, oN, sN1, bN1) ∧

(sN=sN1) ∧ cty='Glasgow')}

List the names of staff who currently do not manage any properties for rent.

{fN, lN | ($sN)

(Staff(sN,fN,lN,posn,sex,DOB,sal,bN) ∧

(~($sN1) (PropertyForRent(pN, st, cty, pc, typ,

rms, rnt, oN, sN1, bN1) ∧ (sN=sN1))))}

NOTES

List the names of clients who have viewed a property for rent in Glasgow.

{fN, lN | ($cN, cN1, pN, pN1, cty)

(Client(cN, fN, lN,tel, pT, mR) ∧

Viewing(cN1, pN1, dt, cmt) ∧

PropertyForRent(pN, st, cty, pc, typ,

rms, rnt,oN, sN, bN) ∧

(cN = cN1) ∧ (pN = pN1) ∧ cty = 'Glasgow')}

When restricted to safe expressions, domain relational calculus is equivalent to tuple relational calculus restricted to safe expressions, which is equivalent to relational algebra.

Means every relational algebra expression has an equivalent relational calculus expression, and vice versa.

STUDENT ACTIVITY-2

1. What is relational algebra and what are its uses?

2. What is a relational algebraic expression and what are relational algebraic operations?

3. Which are the relational algebraic operations derived from set theory?

4. Which are the relational algebraic operations developed specifically for relational databases?

SUMMARY

- The relational model has evolved through many stages since its inception. Performance of the model was an early concern. However, performance can be measured in lot of ways-performance at execution time and performance during design and implementation, first case we measure the performance when the data is actually being manipulated. In the second case we measure the performance when the database is under construction. In the second case the relational model was a clear winner from the start outperforming both hierarchical and other models by considerable margins. Although its critics attacked the slow processing speeds to the earlier relational implementation, the relational model no longer suffers from these shortcoming. The access language SQL, together with its earlier predecessors and competitors, also got a lot of criticism. Early versions of SQL were not uniform and standardized in certain concepts. But with the introduction of ANSI SQL standards, this problem standardization-also got resolved. We will see more about the relational databases SQL and other query languages in the coming chapters.

NOTES

TEST YOURSELF

1. What is a domain and how is it related to a data value?
2. What is the cardinality of a relation?
3. What is a primary key?
4. Explain the difference between the candidate keys, primary key and alternate keys?
5. What is the minimality property?
6. What are the constraints included in a relational model?
7. What is entity integrity?
8. What are operational constraints?
9. What are the situations in which we use nulls?
10. What are the three characteristics of the relational data model?
11. When can we say that a database implementation is fully relational?
12. What is guaranteed access rule?
13. What is a databse catalog?
14. What is physical data independence?
15. What is integrity independence?
16. What is non-subversion rule?
17. What is an INTERSECTION? How is it represented? Explain with an example?
18. What is a CARTESIAN PRODUCT? How is it represented? Explain with an example?
19. What do mean by the statement: UNION and INTERSECTION operations are commutative and associative operations.
20. What is a selection condition?
21. What is a RENAME operation? How is it represented? Explain with an example?
22. What is a JOIN operation? How is it represented? Explain with an example?
23. What is relational calculus?
24. What is tuple calculus?
25. How does tuple relational calculus differ from domain relational calculus?

NOTES

True or False

1. A table can have only one candidate key.
2. The foreign key and the primary key should be defined on the same underlying domain.
3. Relational data model includes several types of constraints whose purpose is to maintain the accuracy and integrity of the data in the database.
4. Referential integrity guarantees that every primary key attribute is non-null.
5. Primary key performs the unique identification function in a relational model.
6. In reality, a null is not a value, but rather the absence of a value.
7. Null is the same as 0 for numeric fields or blank for character fields.
8. Operational constraints are the constraints enforced in the database by the business rules or real world limitations.
9. The manipulative part of the relational model consists of a set of operators known collectively as the relational algebra together with relational assignment.
10. Relational database is a finite collection of relations and a relation in terms of domains, attributes, associations and tuples.
11. The RDBMS should be able to distinguish between zeros, blanks and nulls in the records and handle such values in a consistent manner that produces correct answers, comparisons and calculations.
12. The description of a database and its contents are not database tables and therefore needs special querying mechanisms to retrieve the information.
13. Relational algebraic operations enable the users to perform basic retrieval operations.
14. The result of the relational algebraic expression is not a relation.
15. Thus the relational algebraic operations produce new relations, which can be further manipulated using the same relational algebraic operations.
16. Relational algebra is not a procedural language.
17. The operations UNION, DIFFERENCE and INTERSECTION require that the tables involved be union compatible.
18. The CARTESIAN PRODUCT can be defined only on relations that are union compatible.
19. The UNION operation between relations C and O is denoted by $C \cup O$.
20. The result of the UNION (intersection) operation is a relation that includes all tuples that are in both the participating relations.
21. The difference operation between the relations C and O is denoted by C - O.
22. Both UNION and INTERSECTION operations are commutative and associative operations.
23. $A \cup B \neq B \cup A$
24. $A \cap B \neq B \cap A$
25. $A \cup (B \cup C) = (A \cup B) \cup C$ and $A \cap (B \cap C) = (A \cap B) \cap C$
26. The DIFFERENCE operation is not commutative.
27. A - B = B - A.
28. The CARTESIAN PRODUCT between relations C and O is denoted by C X O.
29. SELECT operation acts like a filter that allows only the tuples that match the specified criteria into the result set.
30. The SELECT operation is not a unary operation.
31. The number of tuples in the resulting relation is always less than or equal to the number of tuples in the original relation.
32. The symbol ρ (rho) is used to denote the PROJECT operator.
33. *It is possible to specify in relational calculus any retrieval that can be specified in the relational algebra and vice versa.*

34. The expressive powers of relational algebra and relational calculus are not identical.
35. Most commercial relational query languages are relationally complete but have more expressive power than relational algebra or relational calculus.
36. Commercial query languages include additional operations such as aggregate functions, grouping and ordering.
37. For the universal quantifier, (∀t) (F) is TRUE if every possible tuple that can be assigned to free occurrences of 't' in F is substituted for 't' and F is TRUE in every such substitutions.

NOTES

Multiple Choice

1. Who is called the father of relational database systems?
   - ❒ E. F. Codd
   - ❒ Donald Chamberlain
   - ❒ C J. Date
   - ❒ H. F. Korth
2. Who wrote the paper 'A Relational Model of Data for Large Shared Data Banks'?
   - ❒ E F. Codd
   - ❒ C J. Date
   - ❒ H. F. Korth
   - ❒ F. R. McFadden
3. Who is the author of the book 'An Introduction to Database Systems'?
   - ❒ E. F. Codd
   - ❒ C. J. Date
   - ❒ H. F. Korth
   - ❒ F. R. McFadden
4. What is the name of the database language introduced by Chamberlain and Boyce in 1974?
   - ❒ QUEL
   - ❒ QBE
   - ❒ SEQUEL
   - ❒ SQL
5. What is the name of the prototype language implemented by IBM in 1974 — 75 and which was based on SEQUEL?
   - ❒ SQLUEL/2
   - ❒ SEQUEL/75
   - ❒ SEQUEL-XRM
   - ❒ None of the above
6. The first large scale implementation of Codd's relational model was IBM's —
   - ❒ DB2
   - ❒ System R
   - ❒ DMS
   - ❒ None of the above
7. SQL was introduced in the ——year ——when become operational.
   - ❒ 1987, DB2
   - ❒ 1977, System R
   - ❒ 1977, SEQUEL-XRM
   - ❒ 1987, DB2
8. Which of the following is not a relational database management system?
   - ❒ Ingress
   - ❒ IMS
   - ❒ DB2
   - ❒ Sybase
9. What is the RDBMS terminology for a row?
   - ❒ Tuple
   - ❒ Relation
   - ❒ Attribute
   - ❒ Domain
10. What is the RDBMS terminology for column or field?
    - ❒ Tuple
    - ❒ Relation
    - ❒ Attribute
    - ❒ Domain
11. What is the RDBMS terminology for a table?
    - ❒ Tuple
    - ❒ Relation
    - ❒ Attribute
    - ❒ Domain
12. What is the RDBMS terminology for a set of legal values that an attribute can have?
    - ❒ Tuple
    - ❒ Relation
    - ❒ Attribute
    - ❒ Domain

NOTES

13. What is the RDBMS terminology for the number of tuples in a relation?
    - ☐ Cardinality
    - ☐ Relation
    - ☐ Attribute
    - ☐ Degree
14. What is the RDBMS terminology for the number of attributes in a relation?
    - ☐ Cardinality
    - ☐ Relation
    - ☐ Attribute
    - ☐ Degree
15. Which of the following aspect of data is the concern of a relational database model?
    - ☐ Data structure
    - ☐ Data integrity
    - ☐ Data manipulation
    - ☐ All of the above
16. What is the smallest unit of data in the relational model?
    - ☐ Field
    - ☐ Data value
    - ☐ Data type
    - ☐ None of the above
17. A set of possible data values is called ——.
    - ☐ Attribute
    - ☐ Degree
    - ☐ Tuple
    - ☐ Domain
18. A table with just one field is called ——table and a table with two fields is called a ——, table.
    - ☐ Single, double
    - ☐ Unary, binary
    - ☐ Single-row, double-row
    - ☐ None of the above
19. What is the cardinality of a table with 1000 rows and 10 columns?
    - ☐ 10
    - ☐ 100
    - ☐ 1000
    - ☐ None of the above
20. What is the cardinality of a table with 5000 rows and 50 columns?
    - ☐ 10
    - ☐ 50
    - ☐ 500
    - ☐ 5000
21. What is the degree of a table with 1000 rows and 10 columns?
    - ☐ 10
    - ☐ 100
    - ☐ 1000
    - ☐ None of the above
22. What is the degree of a table with 5000 rows and 50 columns?
    - ☐ 50
    - ☐ 500
    - ☐ 5000
    - ☐ None of the above
23. Which of the following keys in a table can uniquely identify a row in a table?
    - ☐ Candidate key
    - ☐ Primary key
    - ☐ Alternate key
    - ☐ All of the above
24. A table can have only one ——.
    - ☐ Candidate key
    - ☐ Primary key
    - ☐ Alternate key
    - ☐ All of the above
25. All candidate keys other than the primary keys are called ——.
    - ☐ Secondary keys
    - ☐ Alternate keys
    - ☐ Eligible keys
    - ☐ None of the above
26. What is the name of the attribute or attribute combination of one relation whose values are required to match those of the primary key of some other relation?
    - ☐ Primary key
    - ☐ Candidate key
    - ☐ Matching key
    - ☐ Foreign key
27. Which of the following is an integrity constraint?
    - ☐ Domain constraint
    - ☐ Entity integrity
    - ☐ Referential integrity
    - ☐ All of the above
28. Which of the following is part of a domain definition?
    - ☐ Domain name
    - ☐ Data type
    - ☐ Size
    - ☐ All of the above

29. Which rule guarantees that every primary key attribute is non-null?
    - ❒ Operational constraints
    - ❒ Domain constraint
    - ❒ Entity integrity constraint
    - ❒ Referential integrity
30. Which of the following rule states that if there is a foreign key in one relation either each foreign key value must match a primary key value in the other table or else the foreign key value must be null?
    - ❒ Foreign key rule
    - ❒ Foreign key matching rule
    - ❒ Referential integrity
    - ❒ Entity integrity
31. Which of the following rule states that all the information in a relational database including table names, column names is represented by values in tables?
    - ❒ Information rule
    - ❒ Guaranteed access rule
    - ❒ Non-subversion rule
    - ❒ View updating rule
32. Which of the following rule states that the RDBMS should have a language that accesses the information and this language should not be used to bypass the integrity constraints?
    - ❒ Information rule
    - ❒ Guaranteed access rule
    - ❒ Non-subversion rule
    - ❒ View updating rule
33. Which of the following rule states that any view that can be updated theoretically can be updated using the RDBMS?
    - ❒ Information rule
    - ❒ Guaranteed access rule
    - ❒ Non-subversion rule
    - ❒ View updating rule
34. Which of the following rule states that every piece of data in a relational database can be accessed by using a combination of a table name, a primary key value that identifies the row and a column name, which identifies a cell?
    - ❒ Information rule
    - ❒ Guaranteed access rule
    - ❒ Non-subversion rule
    - ❒ View updating rule
35. Which of the following rule states that the RDBMS should support insertion, update and deletion at a table level?
    - ❒ High-level update rule
    - ❒ Guaranteed access rule
    - ❒ Non-subversion rule
    - ❒ View updating rule
36. Which of the following constitutes a basic set of operations for manipulating relational data?
    - ❒ Predicate calculus
    - ❒ Relational algebra
    - ❒ Relational calculus
    - ❒ None of the above
37. Which of the following is not a relational algebraic operation that is not from the set theory?
    - ❒ UNION
    - ❒ INTERSECTION
    - ❒ CARTESIAN PRODUCT
    - ❒ SELECT
38. Which of the following is not a relational algebraic operation that is developed specifically for the relational databases?
    - ❒ *SELECT*
    - ❒ *UNION*
    - ❒ JOIN
    - ❒ PROJECT
39. Which is the symbol used to denote the SELECT operation?
    - ❒ Sigma
    - ❒ Rho
    - ❒ Pi
    - ❒ None of the above
40. Which is the symbol used to denote the PROJECT operation?
    - ❒ Sigma
    - ❒ Rho
    - ❒ Pi
    - ❒ None of the above
41. Which is the symbol used to denote the RENAME operation?
    - ❒ Sigma
    - ❒ Rho
    - ❒ Pi
    - ❒ None of the above

NOTES

NOTES

42. Which of the following operations need the participating relations to be union compatible?
    - ❒ UNION
    - ❒ INTERSECTION
    - ❒ DIFFERENCE
    - ❒ All of the above
43. What will be the number of columns of CARTESIAN PRODUCT if the participating relations have 5 and 7 columns respectively?
    - ❒ 5
    - ❒ 12
    - ❒ 35
    - ❒ None of the above
44. What will be the number of rows of CARTESIAN PRODUCT if the participating relation have 5 columns and 20 rows respectively?
    - ❒ 5
    - ❒ 20
    - ❒ 25
    - ❒ 100
45. Which of the following is the operation that is used if we are interested in only certain attributes or columns of a table?
    - ❒ SELECT
    - ❒ PROJECT
    - ❒ UNION
    - ❒ JOIN
46. Which of the following is the symbol used to represent the UNION operation?
    - ❒ X
    - ❒ $\cup$
    - ❒ -
    - ❒ $\cap$
47. Which of the following is the symbol used to represent the CARTESIAN PRODUCT?
    - ❒ X
    - ❒ $\cup$
    - ❒ -
    - ❒ $\cap$
48. Which of the following is true?
    - ❒ $A \cup B \neq B \cup A$
    - ❒ $A \cap (B \cap C) \neq (A \cap B) \cap C$
    - ❒ $A - B \neq B - A$
    - ❒ None of the above
49. If two relations have 5 and 10 tuples respectively, then what will be the number of tuples in the CARTESIAN PRODUCT?
    - ❒ 5
    - ❒ 10
    - ❒ 15
    - ❒ 50
50. Which of the following is not a procedural language?
    - ❒ Relational Algebra
    - ❒ SQL
    - ❒ Relational Calculus
    - ❒ None of the above
51. Which of the following is a logical operator?
    - ❒ AND
    - ❒ OR
    - ❒ NOT
    - ❒ All of the above
52. Which of the following is the symbol for the existential quantifier?
    - ❒ $\cap$
    - ❒ $\forall$
    - ❒ $\exists$
    - ❒ $\cup$

CHAPTER 4 DBMS BASED ON RELATIONAL MODEL

NOTES

LEARNING OBJECTIVES

- Introduction
- Structure of Relational Database
- Structures Query Language (SQL)

INTRODUCTION

While introducing a relational model to the database community in 1970, Dr. E.F. Codd stressed on the independence of the relational representation from physical computer implementation such as ordering on physical devices, indexing and using physical access path. Dr. Codd also proposed criteria for accurately structuring relational database and an implementation-independent language to operate on these databases. On the basis of his proposal, the most significant research towards three developments resulted into overwhelming interest in the relational model.

The first development was of prototype relational database system (DBMS) System *R* at IBM's San Jose Research Laboratory in California USA during the laste 1970s. System R provided the practical implementation of its data structures and operations. It also provided information about transaction management, concurrency control, recovery techniques, query optimization, data security, integrity, user interface and so on. System *R* led to the following two major developments:

- *A structured query language called* SQL, also pronounced *S-Q-L, or See-Quel.*
- Production of various commercial relational DBMS such as DB2 and SQL/DS from IBM. ORACLE from Oracle Corporation during 1970s and 1980s.

The second development was of a relational DBMS INGRESS (*Interactive Graphics Retrieval System)* at the University of California at Berkeley USA. The INGRES project involved the development of a prototype RDBMS, with the research concentrating on the same overall objectives as of the System R project.

The third development was the *Peterlee Relational Test Vehicle* at the IBM UK Scientific Centre in Peterlee. The project had more theoretical orientation than the System *R* and INGRES projects and was significant, principally for research into such issues as query processing, optimization and functional extension.

Since the introduction of the relational model, there has been many more developments in its theory and application. During the ensuing years, the relational approach to database received a great deal of publicity. Yet, only since the early 1980s have commercially viable relational database management systems (RDBMSs) have been available. Today hundreds of RDBMSs are

NOTES

commercially available for various hardware platforms both (mainframe and microcomputers). ORACLE from Oracle, INGRES System *R* from IBM, Access and FoxPro from Microsoft, Paradox from Coral Corporation, Interbase and BDE from Borland, and *R*:Base from R:BASE Technologies are some of the examples of RDBMSs that are used on Microcomputer (PC) platforms. Similarly, in addition to ORACLE and INGRES, other RDBMS available on mainframe computers are DB2, UDB, INFORMIX and so on.

The saga of RDBMSs is one of the most fascinating stories in this still young field of database. How they compare with hierarchical and network DBMSs in terms of operation, performance and overall philosophy is not only interesting but also highly instructive for a true understanding of some of the most basic concepts in database.

STRUCTURE OF RELATIONAL DATABASE

Relational database system has a simple logical structure with sound theoretical foundation. The relational model is based on the core concept of *relation*. In the relational model, all data is logically structured within relations (also called *table*). Informally a relation may be viewed as a named two-dimensional table representing an entity set. A relation has a fixed number of named columns (or *attributes*) and a variable number of rows (or *tuples*). Each tuple represents an instance of the entity set and each attribute contains a single value of some recorded property for the particular instance. All members of the entity set have the same attributes. The number of tuples is called *cardinality*, and the number of attributes is called the *degree*.

Domain

Fig. 1 shows the structure of an instance or extension, a relation called EMPLOYEE. The EMPLOYEE relation has six attributes (field items), namely EMP-NO, LAST-NAME, FIRST-NAME, DATE-OF-BIRTH, SEX, TEL-NO and SALARY. The extension has seven tuples (records). Each attribute contains values drawn from a particular *domain*. A domain is a set of *atomic* values. Atomic means that each value in the domain is indivisible to the relational model. Domain is usually specified by name, data type, format and constrained range of values. Attribute EMP-NO, is a domain whose data type is an integer with value ranging between 1,00,000 and 2,00,000. Additional information for interpreting the values of a domain can also be given for example, SALARY should have the units of measurement as Indian Rupees or US Dollar. Table 1 shows an example of seven different domains with respect to EMPLOYEE record of Fig. 1. The value of each attribute within each tuple is atomic, that means it is a single value drawn from the domain of the attribute. Multiple or repeating values are not permitted.

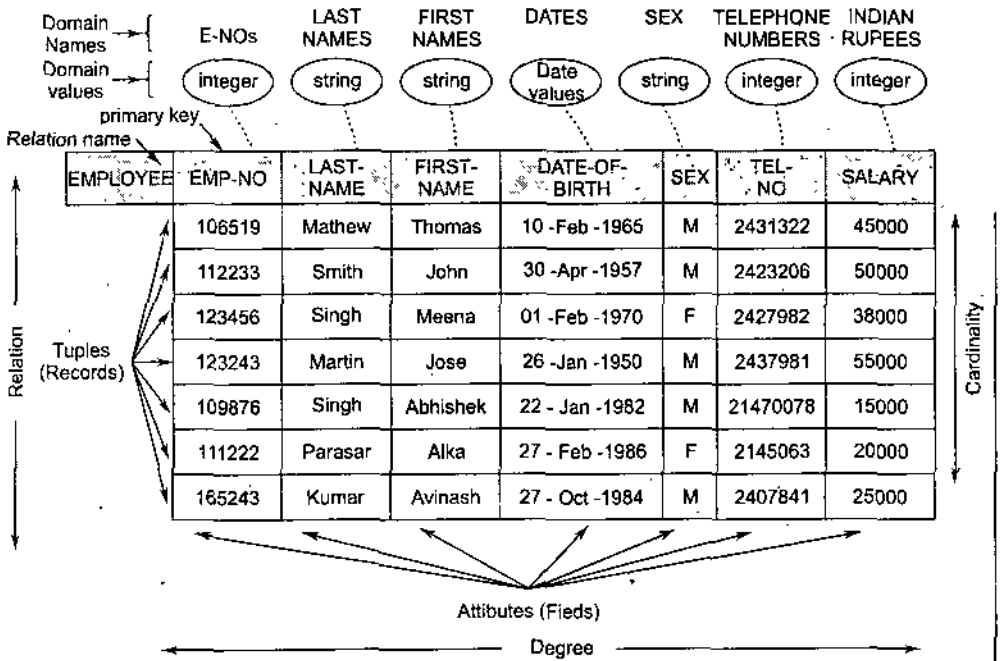


Figure 1

Table 1. Exampe of domain

Attribute	*Domain Name*	*Description*	*Domain Definition*
EMP-NO	Employee number	Unique set of all possible identification numbers	integer:size 6, range 1,00,000-2,00,000
LAST-NAME	Employee Last Name	Set of all possible last names	Character: size 10
FIRST-NAME	Employee First Name	Set of all possible first names	Character: size 20
DATE-OF-BIRTH	Dates	Set of all possible employee birth dates	Data, range from 1-Jan-1950, format dd-mm-yyyy
SEX	Sex	Sex of a person	Character: size 1, value M of F
TEL-NO	Employee contact telephone number	Set of telephone numbers valid on India	Integer: size 7
SALARY	Empolyee Salary	Possible values of employee salary	Monetary: 7 digits, range 10,000,00-90,000,00

The relationship R for a given n number of domains D ($D_1, D_2, D_3, ..., D_n$) consists of an un-ordered set of n-tuples with attributes ($A_1, A_2, A_3, ..., A_n$) where each value A_1 is drawn from the corresponding domain D_1. Thus.

$$A_1 \in D_1 A_2 \in D_2 ... A_n \in D_n$$

Each tuple is a member of the set formed by the Cartesian product (that is all possible distinct combination) of the domains $D_1 \times D_2 \times D_3 \times .... \times D_n$. Thus, each tuple is distinct from all others and any instance of the relation is a subset of the Cartesian product of its domain.

Table 2 presents a summary of structural terminologies used in the relational model. As shown in the table, the informal *equivalents* have only rough

NOTES

NOTES

(approximate) and ready definitions, while the *formal relation* terms have precise definitions. For example, a term "relation" and the term "table" are not really the same thing, although it is common in practice to pretend that they are.

Table 2. Summary of structural terminology

Formal relational term	*Informal equivalents*
relation	table
attribute	column of field
tuple	row or record
cardinality	number of rows
degree	number of columns
domain	pool of legal or atomic values
key	unique identifier

Keys of Relations

A relation always has a unique identifier, a field or group of fields (attributes) who values are unique throughout all of the tuples of the relation. Thus, each tuple is distinct, and can be identified by the values of one or more of its attributes called *key*. Keys are always minimal sequences of attributes that provide the uniqueness quality.

Superkey

Superkey is an attribute, or set of attributes, that uniquely identifies a tuple within a relation. In Fig. 1, the attribute EMP-NO is a superkey because only one row in the relation has a given value of EMP-NO. Taken together, the two attributes EMP-NO and LAST-NAME are also a supekey because only one tuple in the relation has a given value of EMP-NO and LAST-NAME. In fact, all the attributes in a relation taken together are superkey because only one row in a relation has a given value for all the relation attributes.

Relation Key

Relation key is defined as a set of one or more relation attributes concatenated. Most of the relational theory restricts the relation key to a minimum number of attributes and excludes any unnecessary one. Such restricted keys are called relation keys. Following three properties should hold for all time and for any instance of the relation:

- *Uniqueness:* A set of attributes has a unique value in the relation for each tuple.
- *Non-redundancy:* If an attribute is removed form the set of attributes, the remaining attributes will not possess the uniqueness property.
- *Validity:* No attribute value in the key may be null.

A relation key can be made up of one or many attributes. Relation keys are logical and bear no relationship to how the data are to be accessed. It only specifies that a relation have at most row with a given value of the relation key. Furthermore, the term relation key refers to all the attributes in the key as a whole, not to each one.

NOTES

Relation : ASSIGN

EMP-NO	*PROJECT*	*YRS-SPENT-BY EMP-ON-PROJECT*
106519	P1	5
112233	P3	2
106519	P2	5
123243	P4	10
106519	P3	3
111222	P1	4

Figure 2

Fig. 2 illustrates the relation ASSIGN, showing the departments in which the employees defined in the relation EMPLOYEE work. In each row, the column YRS-SPENT-BY EMP-ON-PROJECT indicates the year that an employee in the column EMP-NO spent in the department in the column PROJECT. The relation ASSIGN has a relation key with two attributes, EMP-NO and PROJECT. The values in these two columns together uniquely identify the tuples in ASSIGN. EMP-NO cannot be a relation key by itself because more than one tuple can have the same value of EMP-NO, as shown in tuple 1, 3 and 5 in Fig. 2. That means, an employee can work in more than one projects. Similarly, PROJECT cannot be relation key on its own because more than one employee can work on the same project.

Candidate Key

When more than one or group of attributes serve as a unique identifier, they are each called *candidate key*. A candidate key has more than one relation key, as shown in relation USE of Fig. 3. It contains information about project (PROJECT), project manager (PROJ-MANAGER), machine (MACHINE) used by a project and quantity of machines used (QTY-USED). It has been assumed that *each project* has one *project manager* and that each *project manager* manages *only one project*.

Relation: Use

PROJECT	*PROJ-MANAGER*	*MACHINE*	*QTY-USED*
P1	Thomas	Excavator	5
P3	John	Shovel	2
P2	Abhishek	Drilling	5
P4	Avinash	Dumper	10
P3	John	Welding	3
P1	Thomas	Drilling	4

Figure 3

The project manager of project P1 is Thomas and this project uses five excavators and four drills. There will be at most one row for a combination of a project and machine, and {PROJECT, MACHINE} is the relation key. It is to be noted that a project has only one project manager and that consequently PROJ-MANAGER can identify a project. {PROJ-MANAGER, MACHINE} is also a relation key. Thus relation USE of Fig. 3 has two relation keys. Some keys are more important than others. For example, {PROJECT, MACHINE} is considered more important than {PROJ-MANAGER, MACHINE} because PROJECT is more stable identifier of

NOTES

projects. PROJ-MANAGER is not a stable identifier because a project's manager can change during its execution. Since this is an important key, it is often known as primary key, senior to the candidate keys.

A candidate key can also be described as a superkey without the redundancies. In other words, candidate key is a superkey such that no proper subset is a superkey within the relation. There may be several candidate keys for a relation.

Primary Key

Primary key is a candidate key that is selected to identify tuples uniquely within the relation. For example, if a company assigns each employee a unique employee identification number (for example, EMP-NO in EMPLOYEE record of Fig. 1), then attribute EMP-No is a primary key which can be used to uniquely identify a particular tuple (record). On the other hand, if a company does not use employee identification number, then the LAST-NAME attribute the FIRST-NAME attribute may have to be taken as a group to provide a unique key for the relation. In this case each attribute is a candidate key.

Relation R_1: EMPLOYEE

EMP-NO	*NAME*	*DATE-OF-BIRTH*	*CITY*
106519	Thomas	10-Mar-1965	Delhi
112233	John	30-May-1957	Mumbai
106519	Abhishek	01-Feb-1970	Kolkata
123243	Avinash	26-Jan-1950	Jamshedpur

Relation R_2: ASSIGN

EMP-NO	*PROJECT*	*YRS-SPENT-BY EMP-ON-PROJECT*
106519	P1	5
112233	P3	2
106519	P2	5
123243	P4	10
106519	P3	3
111222	P1	4

Figure 4. Example of foreign key

Foreign Key

A foreign key may be defined as an attribute, or set of attributes, within one relation that matches the candidate key of some (possibly the same) relation. Thus, as shown in Fig. 4, the foreign key in relation R_1 is a set of one or more attributes that is a relation key in another relation R_2, but not a relation key of relation R_1. The foreign key is used in regard to database integrity.

Mappings

In the three-schema architecture database system, each user group refers only to its own external schema. Hence, the user's request specified at external schema level must be transformed into a request at conceptual schema level. The transformed request at conceptual schema level should be further transformed at internal schema level for final processing of data in the stored database as per

user's request. The final result from processed data as per user's request must be reformatted to satisfy the user's external view. The process of transforming requests and results between the three levels are called *mappings*. The database management system (DBMS) is responsible for this mapping between internal, conceptual and external schemas.

- Conceptual/Internal mapping
- Extern/Conceptual mapping

NOTES

Conceptual/Internal Mapping

The conceptual schema is related to the internal schema through *conceptual/internal mapping*. The conceptual internal mapping defines the correspondence between the conceptual view and the stored database. It specifies how conceptual records and fields are presented at the internal level. It enables DBMS to find the actual record or combination of records in physical storage that constitute a logical record in the conceptual schema, together with any constraints to be enforced on the operations for that logical record. It also allows any differences in entity names, attribute names, attribute orders, data types, and so on, to be resolved. In case of any change in the structure of the stored database, the conceptual/internal mapping is also changed accordingly by the DBA, so that the conceptual schema can remain invariant. Therefore, the effects of changes to the database storage structure are isolated below the conceptual level in order to preserve the physical data independence.

External/Conceptual Mapping

Each external schema is related to the conceptual schema by the *external/conceptual mapping*. The external/conceptual mapping defines the correspondence between a particular external view and the conceptual view. It gives the correspondence among the records and relationships of the external and conceptual views. It enables the DBMS to map names in the user's view on to the relevant part of the conceptual schema. Any number of external views can exist at the same time, any number of users can share a given external view and different external view can overlap.

There could be one mapping between conceptual and internal levels and several mappings between external and conceptual levels. The conceptual/internal mapping is the key to physical data independence while the external/conceptual mapping is the key to the logical data independence.

The information about the mapping requests among various schema levels are included in the system catalog of DBMS. The DBMS uses additional software to accomplish the mappings by referring to the mapping information in the system catalog. When schema is changed at some level, the schema at the next higher level remains unchanged. Only the mapping between the two levels is changed. Thus, data independence is accomplished. The two-stage mapping of ANSI-SPARC three-tier structure provides greater data independence but inefficient mapping. However, ANSI-SPARC provides efficient mapping by allowing the direct mapping of external schemas on to the internal schema (by passing the conceptual schema) but at reduced data independence (more data-dependent).

NOTES

STRUCTURES QUERY LANGUAGE (SQL)

Structured Query Language (SQL), also called Structured English Query Language (SEQUEL), is relational query language. It is the standard command set used to communicate with the relational database management system (RDBMS). It is based on the tuple relational calculus, though not as closely as QUEL. SQL resembles relational algebra in some places and tuple relational calculus in others. It is a non-procedural language in which block structured format of English key words is used. SQUEL (widely known as SQL) was the first prototype query language developed by IBM in the early-1970s. It was first implemented on a large scale in IBM prototype called System R and subsequently extended to numerous commerical products from IBM as well as other vendors. In 1986, SQL was declared a standard for relational data retrieval languages by the American National Standards Institute (ANSI) and by the International Standards Organisation (ISO) and called it SQL-86. In 1987, IBM published its own corporate SQL standard, the System Application Architecture Database Interface (SAA-SQL). ANSI published an extended standard for SQL, SQL-89 in 1989, SQL-92 in 1992 and the most recent version SQL-1999.

SQL is both data definition language and data manipulation language of a number of relational database systems such as System *R*, SQL/DS, and DB2 of IBM, ORACLE of Oracle Corporation, INGRES of Relational Technologies and so on. ORACLE was the first commercial RDBMS developed in 1979 that supported SQL. SQL is very simple to use and interactive in nature. Users with very little or no expertise in computers, can find it easy to use. SQL facilitates in executing all tasks related to RDBMS such as creating tables, querying the database for information, modifying the data in the database, deleting them, granting access to users and so on. Thus, it has various features such as query formulation, facilities for insertion, deletion and update operations. It includes statements such as RETURN, LOOP, IF, CALL, SET, LEAVE, WHILE, CASE, REPEAT and several other related features such as variables and exception handlers. It also creates new relations and controls the sets of indexes maintained on the database. SQL can be used interactively to support *ad hoc* requests, or be embedded into procedural code to support operational transactions. Different database vendors use different dialects of SQL, but the basic features of all of them are the same. They use the same base standard of the ANSI SQL standard.

SQL is essentially a free-format language, which means that parts of the statement do not have to be typed at particular locations on the screen. There are many software packages for example, SQL generators, CASE tools and application development environment, where SQL statements can automatically be generated. CASE tools such as Designer-2000, Information Engineering Facility (IEF) and so on can be used to generate the entire application including SQLs. In a Power Builder application, its DataWindow package can be used to generate SQL code. SQL codes can be generated using browser software packages like MS-Query for querying and updating data in a database. SQL is the main interface for communicating between the users and RDBMS.

SQL has the following main components:

(*a*) Data structure.
(*b*) Data type.
(*c*) SQL operators.

(*d*) Data definition language (DDL).
(*e*) Data query language (DQL).
(*f*) Data manipulation language (DML).
(*g*) Data control language (DCL).
(*h*) Data administration statements (DAS).
(*i*) Transaction control statements (TCS).

NOTES

Advantages of SQL

- SQL is the standard query language.
- It is very flexible.
- It is essentially a free-format syntax, which gives the users the ability to structure SQL statements in a way best suited to him.
- SQL is a high level language and the command structure of SQL consists of Standard English words.
- It is supported by every product in the market.
- It gives the users an ability to specify key database operation such as table view and index creation on a dynamic basic.
- It can express arithmetic operations as well as operations to aggregate data and sort data for output.
- Applications written in SQL can be easily ported across systems.

Disadvantage of SQL

- SQL is very far from being the perfect relational language and it suffers from signs of both omission and commission.
- It is not a general-purpose programming language and thus the development of an application requires the use of SQL with a programming language.

Basic SQL Data Structure

In SQL, the data appears to be stored as simple linear files or relations. These files or relations are called *'tables'* in SQL terminology. SQL is set-oriented in which the referenced data objects are always tables. SQL always produces results in tabular format. The tables are accessed either sequentially or through indexes. An index can reference one or a combination of columns of a table. A table can have several indexes built over it. when the data in a table changes, SQL automatically updates the corresponding data in any indexes that are affected by that change. In SQL, the concept of logical and physical views is implemented. A physical view is called a *'base table'*, whereas a logical view is simply called *'view'*. The logical view is derived from one or more base tables of physical view. A view may consist of a subset of the columns of a single table or of two or more joined tables.

The creation of a view in SQL does not entail the creation of a new table by physically duplicating data in a base table. Instead, information describing the nature of the view is kept in one or several system catalogs. The catalogue is a set of schemes, which when put together, constructs a description of a database. The queries can be issued to either base tables or views. When a query references a view, the information about the view in the catalogue maps it onto the base table where the required data is physically stored. The schema is that structure which contains descriptions of objects created by a user, such a base tables, views, constraints and so on as part of the database.

NOTES

SQL Data Types

Data type of every data object is required to be declared by the programmer while using programming language. Also, most database systems require the user to specify the type of each data field. The data type varies from one programming language to another and from one database application to another.

SQL data types

S.N.	Data Type	Description
1.	BIT(*n*)	Fixed-length bit string of '*n*' bits, numbered 1-*n*.
2.	BIT VARYING(*n*)	Variable-length bit string with maximum length of '*n*' bits.
3.	CHAR(*n*) or CHARACTER(*n*)	Fixed-length string of length of exactly '*n*' characters.
4.	VARCHAR (*n*) or CHAR VARYING(*n*)	Variable-length character string of maximum character length of '*n*'.
5.	DECIMAL(*p*, *s*) or DEC(*p*, *s*) or NUMERIC(*p*, *s*)	Exact decimal numeric value. The number of decimal digits or precision is given by '*p*', and the number of digits after the decimal point (the scale) by '*s*'.
6.	INTEGER or INT	Integer number.
7.	FLOAT(*p*)	Floating point number with precision equal to or greater than '*p*'.
8.	REAL	Single precision floating point number.
9.	DOUBLE PRECISION	Double precision floating point number.
10.	SMALLINT	Integer number of lower precision than INTEGER.
11.	DATE	Date expressed as YYYY-MM-DD.
12.	TIME	Time expressed as HH:MM:SS.
13.	TIME(*p*) or TIME WITH TIME ZONE or TIME(*p*) with TIME ZONE	The optional fractional seconds precision(*p*) extends the format to include fractions of seconds, for example TIME(2) HH:MM:SS WITH TIME ZONE adds six positions for a relative displacement from 12:59 to +13.00 in hours:minutes.
14.	INTERVAL	Relative time interval (positive or negative). Intervals are either year/month expressed as 'YYYY-MM' YEAR TO MONTH, or day/time, for example, 'DD HH:MM:SS' DAY TO SECOND(*p*).
15.	TIMESTAMP	Absolute time expressed as YYYY-MM-DD HH:MM:SS.
16.	TIMESTANP(*p*)	The optional fractional seconds precision(*p*) extends the format as for TIME. Timestamps are graduated to be unique and to increase monotonically.
17.	TIMESTANP WITH TIMEZONE or TIMESTAMP(*p*) WITH TIMEZONE	Same as TIME WITH TIMEZONE (Serial No. 13).

SQL Operators

SQL operators and conditions are used to perform arithmetic and comparison statements. Operators are represented by single character or reserved words, whereas conditions are the expression of several operators or expressions that evaluate to TRUE, FALSE or UNKNOWN. Two types of operators are used, namely binary and unary. The unary operator operates on only one operand, while the binary operator operates on two operands.

NOTES

S.N.	Operators	Description
Arithmetic Operators		
1.	+, -	Unary operators for denoting a positive (+ve) or negative (-ve) expression.
2.	*	Binary operator for multiplication.
3.	/	Binary operator for division.
4.	+	Binary operator for addition.
5.	-	Binary operator for subtraction.
Comparison Operators		
6.	=	Equality.
7.	! =, < >, ⌉	Inequality.
8.	<	Less than.
9.	>	Greater than.
10.	>=	Greater than or equal to.
11.	<=	Less than or equal to.
12.	IN	Equal to any member of.
13.	NOT IN	Not equal to any member of.
14.	IS NULL	Test for nulls.
15.	IS NOT NULL	Test for anything other than nulls.
16.	LIKE	Returns true when the first expression matches the pattern of the second expression.
17.	ALL	Compares a value to every value in a list.
18.	ANY, SOME	Compares a value to each value in a list.
19.	EXISTS	True if sub-query returns at least one row.
20.	BETWEEN x and y	> = x and < = y
Logical Operators		
21.	AND	Returns true if both component conditions are true, otherwise returns false.
22.	OR	Returns true if either component conditions are true, otherwise returns false.
23.	NOT	Returns true if the condition is false, otherwise returns false.
Set Operators		
24.	UNION	Returns all distinct rows from both queries.
25.	UNION ALL	Returns all rows from both queries.
26.	INTERSECT	Returns all rows selected by both queries.
27.	MINUS	Returns all distinct rows that are in the first query but not in the second one.
Aggregate Operators		
28.	AVG	Average.
29.	MIN	Minimum.
30.	MAX	Maximum.
31.	SUM	Total.
32.	COUNT	Count.

SQL Data Definition Language (DDL)

The SQL data definition language (DDL) provides commands for defining relation schemas, deleting relations and modifying relation schemas. These commands are used to create, alter and drop tables. The syntax of the commands are CREATE, ALTER and DROP. The main logical SQL data definition statements are:

CREATE TABLE
CREATE VIEW
CREATE INDEX
ALTER TABLE
DROP TABLE
DROP VIEW
DROP INDEX

NOTES

CREATE TABLE Operation

Tables are the basic building blocks of RDBMSs. Tables contain rows (called tuples) and columns (called attributes) of data in a database. CREATE TABLE operation is one of the more frequently used DDL statements. It defines the names of tables and columns, as well as specifies the type of data allowed in each columns.

Syntax for creating SQL table

```
CREATE TABLE base-table-name
(Column-1-definition
[,Column-2-definition]....
[,Column-n-definition]
[,Primary-key-defintion]
[,Alternate-key-definitions]
[,Foreign-key-definitions];
where column-definition has syntax-
Column-name data-type [NULL I NOT NULL
[WITH DEFAULT I UNIQUE]]
```

The CREATE TABLE statement specifies a logical definition of a stored table (or base table). It specifies the name of the table and lists the name and type of each column. The type of column many be standard data type or a domain name. The keywords NULL and NOT NULL are optional. A DEFAULT clause may be used to set column values automatically wherever a new row is inserted. In the absence of a specified default value, nullable columns will contain nulls. A type-dependent value, such as zero or an empty string, will be used for nun-nullable columns.

The PRIMARY KEY clause lists one or more columns that from the primary key. The FOREIGN KEY clause is used to specify referential integrity constraints and, optionally, the actions to be taken if the related tuple is deleted or the value of its primary key is updated. If the table contains other unique keys, the columns can be specified in a UNIQUE clause.

Data types with defined constraints and default values can be combined into domain definitions. A domain definition is a specialized data type, which can be defined within a schema and used as desired in columns definitions. Limited support for domains is provided by the CREATE DOMAIN statements, which associates a domain with a data type and, optionally, a default value. For example, suppose we wish to define a domain of person identifiers to be used in the column definitions of various tables. Since we will be using it over and over again in the database schema, we would like to simplify our work and thus, we create a domain as follows:

CREATE DOMAIN PERSON-IDENTIFIER NUMBER (6) DEFAULT (0)

or

CREATE DOMAIN PERSON-IDENTIFIER NUMERIC (6)
DEFAULT (0) CHECK (VALUE IS NOT NULL);

The above definition says that a domain named PERSON-IDENTIFIER has the properties such as its data type is of six-digit numeric and default value is zero. Any column defined with this domain as its data type will have all these properties. As shown in the second from above, the domain definition may also be followed by a constraint definition that limits the range of possible values by employing a CHECK clause. Here domain has the property such that it can never be null. Now we can define columns in our schema with PERSON-IDENTIFIER as their data type.

NOTES

Creating SQL table for employee health center schema

```
CREATE TABLE EMPLOYEE        (
EMPLOYEE-ID                    PERSON-IDENTIFIER    NOT NULL,
LAST-NAME                      CHAR (30)            NOT NULL,
INITIAL                        CHAR (10)            NOT NULL,
ADDRESS                        VARCHAR (40)
DATE-OF-BIRTH                  DATE,
SEX                            CHAR (1)
PRIMARY KEY                    EMPLOYEE-ID
);                                                              } Column-definition
CREATE TABLE DOCTOR          (
DOCTOR-ID                      PERSON-IDENTIFIER    NOT NULL,
PHONE-NO.                      CHAR (10)            NOT NULL,
ROOM-NO.                       CHAR (3)             NOT NULL
ADDRESS                        DATE,
DATE-APPOINTED                 (DOCTOR-ID),
PRIMARY KEY                    (ROOM-NO),
UNIQUE                                                          } Column-definition
FOREIGN KEY (DOCTOR-ID) REFERENCES EMPLOYEE )EMPLOYEE-ID)
            ON DELETE CASCADE
            ON UPDATE CASCADE
)
CREATE TABLE PATIENT (
PATIENT-ID                     PERSON-IDENTIFIER    NOT NULL
DATE-REGISTERED                DATE                 NOT NULL,
REGISTERED-WITH                ITEM-IDENTIFIER
PRIMARY KEY                    (PATIENT-ID),                    } Column-definition
FOREIGN KEY (PATIENT-ID) REFERENCES EMPLOYEE (EMPLOYEE-ID)
            ON DELETE SET NULL
            ON UPDATE CASCADE
)
CONSTRAINT PATIENT=REG
FOREIGN KEY (REGISTERED-WITH) REFERENCES DOCTOR (DOCTOR-ID)
            ON DELETE SET NULL
            ON UPDATE CASCADE
);
CREATE TABLE APPOINTMENT (
DOCTOR-ID                      PERSON-IDENTIFIER    NOT NULL,
PATIENT-ID                     PERSON-IDENTIFIER    NOT NULL,
APP-DATE                       DATE                 NOT NULL
APPT-TIME                      TIME                 NOT NULL,   } Column-definition
```

NOTES

```
APPT-DURATION          INTEGER                DEFAULT (10).
PRIMARY KEY            (DOCTOR-ID, APPT-DATE, APPT-TIME)
FOREIGN KEY (DOCTOR-ID) REFERENCES DOCTOR (DOCTOR-ID)
            ON DELETE CASCADE
            ON UPDATE CASCADE
FOREIGN KEY (PATIENT-ID) REFERENCES PATIENT (PATIENT-ID)
            ON DELETE CASCADE
            ON UPDATE CASCADE
);
CREATE TABLE TREATMENT (
DOCTOR-ID              PERSON-IDENTIFIER      NOT NULL,
PATIENT-ID             PERSON-IDENTIFIER      NOT NULL,
PATIENT-CONDITION      CHAR (30)              NOT NULL,
PATIENT-TREATMENT      CHAR (30)              NOT NULL,
START-DATE             DATE                   NOT NULL,
END-TIME               DATE                   NOT NULL,
PRIMARY KEY (DOCTOR-ID, PATIENT-ID, PATIENT-CONDITION,
            PATIENT-TREATMENT, START-DATE),
FOREIGN KEY (DOCTOR-ID) REFERENCES DOCTOR (DOCTOR-ID)
            ON DELETE CASCADE
            ON UPDATE CASCADE
FOREIGN KEY (PATIENT-ID) REFERENCES PATIENT (PATIENT-ID)
            ON DELETE CASCADE
            ON UPDATE CASCADE
);
```

Column-definition

ON UPDATE and ON DELETE clauses are used to trigger referential integrity checks and specifying their corresponding actions. The possible actions of these clauses are SET NULL, SET DEFAULT and CASCADE. Both SET NULL and SET DEFAULT remove the relationship by resetting the foreign key value to null, or to its default if it has one. The action is same for both updates and deletes. The effect of CASCADE depends on the event. With ON UPDATE, a change to the primary key value in the related tuple is reflected in the foreign key. Changing a primary key should normally be avoided but it may be necessary when a value has been entered incorrectly. Cascaded update ensures that referential integrity is maintained. With ON DELETE, if the related tuple is deleted then the tuple containing the foreign key is also deleted. Cascaded deletes are therefore appropriate for mandatory relationships such as those involving weak entity clauses.

```
CONSTRAINT PATIENT-REG
FOREIGN KEY (REGISTERED-WITH) REFERENCES DOCTOR (DOCTOR-ID)
    ON DELETE SET NULL
    ON UPDATE CASCADE
```

In the above statement, the registration of patient with a doctor is optional to enable patient details to be entered before the patient is assigned to a doctor, and to simplify the task of transferring a patient from one doctor to another. The foreign key REGISTERED-WITH will be updated to reflect any change in the primary key of the doctor table, but if the related doctor tuple is deleted, it will be set to null. By default, all constraints are immediate and not deferrable. This means that they are checked immediately after any change is made and that this behavior cannot be changed.

It is also possible to create local or global temporary tables within a transaction. They may be preserved or deleted when the transaction is committed.

Creating local or global temporary table.

```
CREATE LOCAL TEMPORARY TABLE temporary-table-name    (
                (Column-1-definition
                [,Column-2-definition]....
                [,Column-n-definition]
              ) ON COMMIT DELET ROWS    ;
                        or
CREATE GLOBAL TEMPORARY TABLE temporary-table-name?   (
                (Column-1-definition
                [,Column-2,definition]....
                [,Column-n-definition]
              ) ON COMMIT PRESERVE ROWS;
```

NOTES

DROP TABLE Operation

DROP operation is used for deleting tables from the schema. It can be used to delete all rows currently in the named table and to remove the entire definition of the table from the schema. Entire schema can be dropped. The syntax of DROP statement is given in Fig. below:

	DROP SCHEMA	(existing schema-name)
Or		(existing table-name)
Or		(existing column-name)

An example of DROP operation is given below:

DROP SCHEMA HEALTH-CENTRE

or DROP TABLE PATIENT

or DROP COLUMN CONSTRAINT

Since, simple DROP statement can be a dangerous operation, either CASCADE or RESTRICT must be specified with it as shown below:

DROP SCHEMA HEALTH-CENTRE CASCADE

or DRO TABLE PATIENT CASCADE

The above statement means to drop the schema named as well as all tables, data and other schema objects that still exist (that means removing the entire schema irrespective of its content).

DROP SCHEMA HEALTH-CENTRE RESTRICT

DROP TABLE PATIENT RESTRICT

The above statement means to drop the schema only if all other schema objects have already been deleted (that is only if schema is empty). Otherwise, an exception will be raised.

ALTER TABLE operation

ALTER operation is used for changing the definitions of tables. It is *schema evolution* command. It can be used to add one or more columns to a table, change the definition of an existing column, or drop a column from a table. The syntax of ALTER statement is as shown below.

NOTES

Syntax for ALTER operation

```
ALTER TABLE      (existing table-name)
                 ADD (column-name) data type (...)
or
ALTER TABLE (existing table-name)
                 Drop (column-name)
```

An example of ALTER operations is given below:

ALTER TABLE PATIENT ADD COLUMN ADDRESS CHAR (30)

or ALTER TABLE DOCTOR DROP COLUMN ROOM-NO

or ALTER TABLE DOCTOR DROP COLUMN ROOM-NO RESTRICT

or ALTER TABLE DOCTOR DROP COLUMN ROOM-NO CASCADE

Again, CASCADE and RESTRICT can be used in the above statements to determine the drop behavior when constraints or views depend on the affected column. Column default values may be altered or dropped, as shown below:

ALTER TABLE APPOINTMENT

ALTER COLUMN APPT-DURATION SET DEFAULT 20

or ALTER TABLE APPOINTMENT

ALTER COLUMN APPT-DURATION DROP DEFAULT

Here, the default value of 10 for the appointment duration has been changed to 20. The default even can be removed, as shown in the second statement above.

CREATE INDEX Operation

An index is a structure that provides faster access to the rows of a table based on the values of one or more columns. The index stores data values and pointers to the rows (tuples) where those data values occur. An index sorts data values and stores them in assending or descending order. Indexes are created in most RDBMSs to provide rapid random and sequential access to base table data. It can help in quickly executing a query to locate particular column and rows. The CREATE INDEX operation allows the creation of an index for an already existing relation. The column to be used in the generation of the index are also specified. The index is named and the ordering for each column used in the index can be specified as either ascending or descending. Like tables, indexes can also be created dynamically.

Syntax for creating index

```
CREATE [unique] INDEX (index-name)
      ON (existing table-name)
            (column-name) [ASCENDING or DESCENDING]
                        [,column-name [order].....]
      [CLUSTER]
```

The CLUSTER option could also be specified to indicate the records and are to be placed in physical proximity to each other. The unique option specifies that only one record could exist at any time with a given value for the column(s) specified in the statement of create the index. An example of creating index for EMPLOYEE relation is given below.

```
CREATE INDEX EMP-INDEX
        ON EMPLOYEE (LAST-NAME ASC, SEX DESC);
```

The above statement causes a creation of an index called EMP-INDEX with column LAST-NAME and SEX from the relation (table) EMPLOYEE. The entries in the index are ascending by LAST-NAME value and descending by SEX. In the above example, there are no restrictions on the number of records with the same LAST-NAME and SEX. An existing relation or index can be deleted for the database by using the DROP statement in the similar way as explained for table and schema operations.

NOTES

Create View Operation

A view is a named table that is represented by its definition in terms of other named tables. It is a virtual table, which is constructed automatically as needed by the DBMS and is nor maintained as real data. The real data are stored in base tables. The CREATE VIEW operation defines a logical table from one or more tables or views. Views may not be indexed.

```
CREATE VIEW (view-name_
        [(column-name) [,column-name [order].....]
        AS (sub-query) [WITH CHECK OPTION];
```

The sub-query cannot include either UNION or ORDER BY. The clause 'WITH CHECK OPTION' indicates that modifications (update and insert) operations against the view are to be checked to ensure that the modified row satisfies the view-defining condition. There are limitations on updating data through views. Where views can be updated, those changes can be transferred to the underlying base tables originally referenced to create the view. An example of creating view for EMPLOYEE relation is given below:

```
CREATE VIEW   PATIENT VIEW
AS SELECT DOCTOR.DOCTOR-ID, DOCTOR, PHONE-NO
        PATIENT.PATIENT-ID, PATIENT.DATE-REGISTERED,
FROM DOCTOR, PATIENT
```

The above view operation will result into creation of a PATIENT-VIEW table with listing of columns such as DOCTOR-ID and PHONE-NO from DOCTOR table and PATIENT-ID and DATE-REGISTERED from the PATIENT table.

The main purpose of a view is to simplify query commands. However, a view may also provide data security and significantly enhance programming productivity for a database. A view always contains the most recent derived values and is thus superior in terms of data currency to constructing a temporary real table from several base tables. It consumes very little storage space. However, it is costly as because its contents must be calculated each time that they are requested.

SQL Data Query Language (DQL)

SQL data query language (DQL) is one of the most commonly used SQL statements that enable the users to query one or more tables to get the information they want. DQL has only one data query statement whose syntax is SELECT. The SELECT statement is used for retrieval of data from the tables and produce reports. It is the basis for all database queries. The SELECT statement of SQL table departs from the strict definition of a relation in that unique rows are not enforced. SQL allows a table (relation) to have two or more rows (tuples) that are identical in all their attribute (column) values. Thus, a query result may

NOTES

contain duplicate rows. Hence, in general, an SQL table is not a set of tuples as is the case with relation, because a set does not allow two identical members. In face, an SQL table is a *multiset* (sometimes called *bag*) of tuples (or rows). Some SQL relations are constraints to be set because a key constraint has been declared or because the DISTINCT option has been used with the SELECT statement.

Syntax for SQL SELECT statement

SELECT	[ALL DISTINCT] column-name	
FROM	table(s)-name	
WHERE	conditional expression	Optional clause
GROUP BY	clause (column(s)-name	
HAVING	conditional expression	
ORDER BY	column(s)-name	

Database system

In the above syntax, the clauses such as WHERE, GROUP BY, HAVING and ORDER BY, are optional. They are included in the SELECT statement only when functions provided by them are required in the query. In its basic form of the SQL the SELECT statement is formed of three clauses namely, SELECT, FROM and WHERE. This basic form of SELECT statement is sometimes called a *mapping* or a *select-from-where block*. These three clauses corresponds to the relational algebra operations as follows:

- The SELECT clause corresponds to the projection operation of the relational algebra. It is used to list the attributes (columns) desired in the result of a query. SELECT * is used to get all the columns of a particular table.
- The FROM clause corresponds to the Cartesian-product operation of the relational algebra. It is used to list the relations (tables) to be scanned from where data has to be retrieved.
- The WHERE clause corresponds to the selection predicate of the relational algebra. It consists of a predicate involving attributes of the relations that appear in the FROM clause. It tells SQL to include only certain rows of data in the result set. The search criteria is specified in WHERE clause.

Examples of query using SELECT statement

Query: 1	*Result: 1*		
*SELECT * FROM ORDERS;*	*ORD-NO*	*ORD-DATE*	*CUST-NAME*
Note: This query involves only the ORDERS relation listed in the FROM clause. It selects all attributes and all rows of the relation.	ORD-1 ORD-2 ORD-3 ORD-4 ORD-5	10-May-2004 01-Jan-2004 20-Jul-2003 30-Jul-2004 15-Aug-2004	ABC Co. KLY Megapoint HCL ABC Co.

Query: 2	*Result: 2*	
SELECT CUST-NAME, FIRST-ORD-DATE FROM CUSTOMER WHERE LIVED-IN-CITY = "Kolkata"	CUST-NAME	FIRST-ORD-NAME
Note: This query involves only the CUSTOMER relation listed in the FROM clause. It selects all customer name and its first order date for those customers who are located in Kolkata.	KLY HCL	01-Jan-2000 30-July-2001

NOTES

Query: 3	*Result: 3*	
SELECT WH-ID, LOCATION-CITY FROM WAREHOUSE, STORED ITEMS WHERE ITEM-NO =25, DESC = "Electrode".	WH-ID	LOCATION-CITY
Note: This query involves WAREHOUSE, STORED and ITEMS relations listed in the FROM clause. It selects the Warehouse and its location for item no. 25 with description having "Electrode".	WH-MO1 WH-K12	Mumbai Kolkata

Query: 4	*Result: 4*	
SELECT WH-ID, LOCATION-CITY FROM WAREHOUSE, STORED, ITEMS WHERE ITEM-NO = 25, DESC = "Electrode", ORDER BY LOCATION-CITY ACS;	WH-ID	LOCATION-CITY
Note: This query involves WAREHOUSE, STORED and ITEMS relations listed in the FROM clause. It selects the Warehouse and its location for item no. 25 with description having "Electrode" and lists the result in ascending order.	WH-K12 WH-M01	Kolkata Mumbai

Query: 5	*Result: 5*		
SELECT S. WH-ID, S.QTY-HELD, LDESC FROM STORED AS S, ITEMS AS I WHERE S.ITEM-NO = LITEM-NO AND I.WT>5;	WH-ID	QTY-HELD	DESC
Note: In this query the FROM clause shows the definition of abbreviations *S* and *I* for these tables, which can be used anywhere in the query to denote their respective table names.	WH-M01 WH-K12 WH-K12	500 550 450	Bulb File Sheet

Abbreviation or Alias Name

Columns name may be qualified by the name of the table (or relation) in which they are found. But this is only necessary where queries involve two or more tables containing columns with the same name to prevent ambiguity. Dut to this reason, an abbreviations (also called correlation or alias name) *S* and *I* have been used in Query 5 to define two relations STORED and ITEMS. Instead of abbreviations, the relation names can also be directly used to qualify the attribute name, for example, STORED.ITEM-NO, ITEMS.ITEM-NO and so on. Where the column name is unique the table qualification may be omitted. Queries can also be shortened by using an abbreviation name for a table name. This abbreviation or alias is specified in the FROM clause.

Aggregate Functions and the GROUP BY Clause

SQL provides several sets, or aggregate functions using the GROUP BY clause for summarising the content of the columns. This function is usually used with aggregate functions such as AVG, SUM, MIN, MAX and so on. It used to give out common information when querying the tables of a database.

HAVING Clause

NOTES

The HAVING clause is used to include only certain groups produced by the GROUP BY clause in the query result set. It is equivalent to WHERE clause and is used to specify the search criteria or search condition when GROUP BY clause is specified. Example of HAVING clause.

Query: 1

```
SELECT ITEM-NO, ORD-NO, QTY-ORDRD
FROM   ITEMS-ORDERED
GROUP BY ORD-NO;
```

Note: In this of SQL statement, the output has been grouped by common order numbers selected from relation (table) ITEMS-ORDERED.

Result: 1

ITEM-NO	ORD-NO	QTY ORDRD
25	ORD-1	65
27	ORD-2	30
26	ORD-3	50
28	ORD-4	45
25	ORD-5	55

Query: 2

```
SELECT COUNT (ORD-NO)
AS NUM-OF-ORDS,
MIN (QTY-ORDRD) AS MIN-ORD-QTY,
MAX (QTY-ORDRD) AS MAX-ORD-QTY,
AVG (QTY-ORDRD) AS AVG-ORD-QTY,
SUM (QTY-ORDRD) AS TOT-ORD-QTY,
FROM ITEMS-ORDERED
WHERE QTY-ODRD > 45;
```

Note: This SQL query returns the count, minimum, maximum, average and sum of all values of QTY-ORDRD for the ordered quantity more than 45.

Result: 2

NUM-OF-ORDS	MIN-ORD-QTY	MAX-ORD-QTY	AVG-ORD-QTY	TOT-ORD-QTY
3	50	65	57	170

Query: 3

```
SELECT COUNT (ORD-NO)
      AS NUM-OF-ORDS,
   MIN (QTY-ORDRD) AS MIN-ORD-QTY,
   MAX (QTY-ORDRD) AS MAX-ORD-QTY,
   AVG (QTY-ORDRD) AS AVG-ORD-QTY,
   SUM (QTY-ORDRD) AS TOT-ORD-QTY,
FROM ITEMS-ORDERED
GROUP BY ORD-NO;
```

Note: This SQL query returns the count, Minimum, maximum, average and sum of all values of QTY-ORDRD, one row for each distinct value of the columns specified in the GROUP BY clause.

Result: 3

ORD-NO	NUM-OF-ORDS	MIN-ORD-QTY	MAX-ORD-QTY	AVG-ORD-QTY	TOT-ORD-QTY
ORD-1	2	30	65	47	95
ORD-2	2	45	50	47	95
ORD-3	1	55	55	55	55

Query: 1

```
SELECT COUNT (ORD-NO)
      AS NUM-OF-ORDS,
   MIN (QTY-ORDRD) AS MIN-ORD-QTY,
   MAX (QTY-ORDRD) AS MAX-ORD-QTY,
   AVG (QTY-ORDRD) AS AVG-ORD-QTY,
   SUM (QTY-ORDRD) AS TOT-ORD-QTY,
FROM ITEMS-ORDERED
GROUP BY ORD-NO
HAVING COUNT (ORD-NO) 1;
```

Note: In this SQL query, the HAVING clause determines which groups will appear in the result and in the above case lists only those orders whose count is more than one.

Result: 1

ORD-NO	NUM-OF-ORDS	MIN-ORD-QTY	MAX-ORD-QTY	AVG-ORD-QTY	TOT-ORD-QTY
ORD-1	2	30	65	47	95
ORD-2	2	45	50	47	95

ORDERED BY Clause

The ORDER BY clause is used to sort the results based on the data in one or more columns in the ascending or descending order. The defaults of ORDER BY clause is ascending (ASC) and if nothing is specified the result set will be sorted in ascending order.

Examples of ORDER BY clause

Query: 1	*Result: 1*			
SELECT * FROM WAREHOUSE WHERE NO-OF-BINS BETWEEN 200 AND 400 ORDER BY LOCATION-CITY ACS;	WH-ID	LOCATION-CITY	NO-OF-BINS	PHONE
Note: This SQL query lists all columns of table WAREHOUSE in ascending order of location and whose number of bins is between 200 and 400.	WH-J10 WH-M01 WH-D01	Jamshedpur Mumbai New Delhi	400 200 200	0657-2431322 022-2314568 011-2334456

SQL DATA Manipulation Language (DML)

The SQL data manipulation language (DML) provides query language based on both the relational algebra and the tuple relational calculus. It provides commands for updating, inserting, deleting, modifying and querying the data or tuples in the database. These commands may be issued interactively, so that a result is returned immediately following the execution of the statement. The syntax of the SQL DML commands is INSET, DELETE and UPDATE.

SQL INSERT Command

The SQL INSERT command is used to add a new tuple (row) to a relation. The relation (or table) name and list of values of the tuple must be specified. The value of each attribute (column or field) of the tuple (row or record) to be inserted is either specified by an expression or could come from selected records of existing relations. The values should be listed in the same order in which the corresponding attributes were specified in the CREATE TABLE commands or in the order of existing relation. The syntax for INSERT command is given as:

INSERT INTO (table-name) [(attributes-name)

VALUES (lists of values for row 1,

list of values for row 2,

:

:

list of values for row *n*);

In the above syntax, attribute name along with relation name is optional. An example of INSERT command.

NOTES

NOTES

Query: 2	*Result: 2*		
UPDATE STORED SET QTY-HELD = 700 WHERE ITEM-NO IN (SELECT ITEM-NO FROM ITEMS WHERE WT = >6);	WH-ID	ITEM-NO	QTY-HELD
Note: This SQL command results into modifying quantity of item numbers 25 and 28 to 700 as because their weight is more than 6 in the relation ITEMS.	WH-M01	25	700
	WH-M01	26	700
	WH-J10	27	300
	WH-K12	25	550
	WH-K12	28	700

As with other statements, an update may be performed according to the result of a search condition involving other tables, as illustrated in Query 2 in the above example.

SQL Data Control Language (DCL)

SQL data control language (DCL) provides commands to help database administrator (DBA) to control the database. It consists of the commands that control the user access to the database objects. Thus, SQL DCL is mainly related to the security issues, that is, determining who has access to the database objects and what operations they can perform on them. It includes commands to grant or revoke privileges (or authorisation) to access the database or particular objects within the database and to store or remove transactions that would affect the database. The syntax of the command is GRANT and REVOKE.

SQL GRANT Command

The SQL GRANT command is used by the DBA to grant privileges to users. The syntax of the GRANT command is given as:

```
GRANT    (privilege(s))
ON       (table-name/view-name)
TO       (user(s)-id), (group(s)-id), (public)
```

The key words for this command are GRANT, ON and TO, A privilege is typically a SQL command such as CREATE, UPDATE or DROP and so on. The user-id is the identification code of the user to whom the DBA wants to grant the specific privilege. The example of GRANT command is given below:

```
Example 1:  GRANT    CREATE
            ON       ITEMS
            TO       Abhishek
Example 2:  GRANT    DROP
            ON       ITEMS
            TO       Abshishek
Example 3:  GRANT    UPDATE
            ON       ITEMS
            TO       Abshishek
Example 4:  GRANT    CREATE, UPDATE, DROP, SELECT
            ON       ITEMS
            TO       Abhishek
            WITH     GRANT OPTION
```

In the above examples, DBA has granted a user-id named Abhishek the capability to create, update, drop and or select tables. As shown in example 4, the DBA has granted Abhishek the right to create, update, drop and select data in ITEMS table. Furthermore, Abhishek can grant these same rights to others at his descretion.

SQL REVOKE Command

The SQL REVOKE command is issued by the DBA to revoke privileges from users. It is opposite to the GRANT command. The syntax of the REVOKE command is given as:

```
REVOKE    (privilege(s))
ON        (table-name/view-name)
FROM      (user(s)-id), (group(s)-id), (public)
          Database Systems
```

NOTES

The key words for this command are REVOKE and FROM. The example of REVOKE command is given below:

```
Example 1:  INVOKE    CREATE
            ON        ITEMS
            FROM      Abhishek
Example 2:  REVOKE    DROP
            ON        ITEMS
            FROM      Abhishek
Example 3:  REVOKE    UPDATE
            ON        ITEMS
            FROM      Abhishek
Example 4:  REVOKE    CREATE, UPDATE, DROP, INSERT, SELECT
            ON        ITEMS
            FROM      Abhishek
```

In the above examples, DBA has revoked the privileges that were previously granted to user-id named Abhishek.

SQL Data Administration Statements (DAS)

The SQL data administration statement (DAS) allows the user to perform audits and analysis on operations within the database. They are also used to analyse the performance of the system. Data administration is different from database administration in the sense that database administration is the overall administration of the database whereas data administration is only a subset of that. DAS has only two statements whose syntax are START AUDIT and STOP AUDIT.

SQL Transaction Control Statements (TCS)

A transaction is a logical unit of work consisting of one or mere SQL statements that is guaranteed to be atomic with respect to recovery. It may be defined as a process that contains either read commands, write commands or both. An SQL transaction automatically begins with a transaction-initiating SQL query executed by a user or program. SQL TCS manages all the changes made by the DML statements. The main syntax of the TCS command is COMMIT and ROLLBACK.

A COMMIT statement ends the transaction successfully, making the database changes permanent. A new transaction starts after COMMIT with the next transaction-initiating statement.

A ROLLBACK statement aborbs the transaction, backing out any changes made by the transaction. A new transaction starts after ROLL BACK with the next transaction-initiating statement.

STUDENT ACTIVITY

1. What is Relational Database? Explain the domain.

2. What is SQL Data Definition Language (DDL)?

3. What is SQL data query language (DQL)?

4. Give the syntax of the UPDATE Command.

NOTES

SUMMARY

Relational database system has a simple logical structure with sound theoretical foundation. The relational model is based on the core concept of *relation*. In the relational model, all data is logically structured within relations (also called *table*). Informally a relation may be viewed as a named two-dimensional table representing an entity set. A relation has a fixed number of named columns (or *attributes*) and a variable number of rows (or *tuples*). Each tuple represents an instance of the entity set and each attribute contains a single value of some recorded property for the particular instance. All members of the entity set have the same attributes. The number of tuples is called *cardinality*, and the number of attributes is called the *degree*.

Relation key is defined as a set of one or more relation attributes concatenated. Most of the relational theory restricts the relation key to a minimum number of attributes and excludes any unnecessary one. Such restricted keys are called relation keys.

When more than one or group of attributes serve as a unique identifier, they are each called *candidate key*.

TEST YOURSELF

1. What is Structure Query Language and What are its main components?
2. Explain Advantages and Disadvantages of SQL.
3. Write note on
   Create table operations
   Alter table operations
   Drop view operations
   Drop index operations.
4. How SQL data control language (DCL) Commands are used to control the database?
5. How SQL Data Administration Statements (DAS) used to perform audits and analysis on operations within the database?

SECTION C

5. Normalisation

CHAPTER 5 NORMALISATION

LEARNING OBJECTIVES

- Introduction
- Functional Dependency
- Minimal Functional Dependencies
- Equivalent Functional Dependencies
- Multivalued Dependencies
- Closure
- Desirable Properties of Decomposition
- Dependency Preservation
- What is Normalisation?
- Boyce-Codd Normal Form (BCNF)

NOTES

INTRODUCTION

We have noted that relations that form the database must satisfy some properties, for example, relations have no duplicate tuples, tuples have no ordering associated with them, and each element in the relation is atomic. Relations that satisfy these basic requirements may still have some undesirable properties, for example, data redundancy and update anomalies. We illustrate these properties and study how relations may be transformed or decomposed (or normalised) to eliminate them. Most such undesirable properties do not arise if the database modelling has been carried out very carefully using some technique like the Entity-Relationship Model that we have discussed but it is still important to understand the techniques in this chapter to check the model that has been obtained and ensure that no mistakes have been made in modelling.

The central concept in these discussions is the notion of functional dependency which depends on understanding the semantics of the data and which deals with what information in a relation is dependent on what other information in the relation. We will define the concept of functional dependency and discuss how to reason with the dependencies. We will then show how to use the dependencies information to decompose relations whenever necessary to obtain relations that have the desirable properties that we want without loosing any of the information in the original relations.

Let us consider the following relation student.

Sno	sname	address	cno	cname	instructor	office
85001	Smith	1, Main	CP302	Database	Gupta	102
85001	Smith	1, Main	CP303	Communication	Wilson	102
85001	Smith	1, Main	CP304	Software Engg.	Williams	1024
85005	Jones	12, 7th	CP302	Database	Gupta	102

The above table satisfies the properties of a relation and is said to be in *first normal form* (or 1NF). Conceptually it is convenient to have all the information in

NOTES

one relation since it is then likely to be easier to query the database. But the above relation has the following undesirable features:

1. ***Repetition of Information :*** A lot of information is being repeated. Student name, address, course name, instructor name and office number are being repeated often. Every time we wish to insert a student enrolment, say, in CP302 we must insert the name of the course CP302 as well as the name and office number of its instructor. Also every time we insert a new enrolment for, say Smith, we must repeat his name and address. Repetition of information results in wastage of storage as well as other problems.
2. ***Update Anomalies :*** Redundant information not only wastes storage but makes updates more difficult since, for example, changing the name of the instructor of CP302 would require that all tuples containing CP302 enrolment information be updated. If for some reason, all tuples are not updated, we might have a database that gives two names of instructor for subject CP302. This difficulty is called the update anomaly.
3. ***Insertional Anomalies :*** *Inability to represent certain information :* Let the primary key of the above relation be (sno, cno). Any new tuple to be inserted in the relation must have a value for the primary key since existential integrity requires that a key may not be totally or partially NULL. However, if one wanted to insert the number and name of a new course in the database, it would not be possible until a student enrols in the course and we are able to insert values of sno and cno. Similarly information about a new student cannot be inserted in the database until the student enrols in a subject. These difficulties are called *insertion anomalies*.
4. ***Deletion Anomalies :*** Loss of Useful Information — In some instances, useful information may be lost when a tuple is deleted. For example, if we delete the tuple corresponding to student 85001 doing CP304, we will loose relevant information about course CP304 (viz. course name, instructor, office number) if the student 85001 was the only student enrolled in that course. Similarly deletion of course CP302 from the database may remove all information about the student named Jones. This is called *deletion anomalies*.

The above problems arise primarily because the relation student has information about students as well as subjects. One solution to deal with the problems is to decompose the relation into two or more smaller relations.

Decomposition may provide further benefits, for example, in a distributed database different relations may be stored at different sites if necessary. Of course, decomposition does increase the cost of query processing since the decomposed relations will need to be joined, sometime frequently.

The above relation may be easily decomposed into three relations to remove most of the above undesirable properties:

S (*sno, sname, address*)

C (*cno, cname, instructor, office*)

SC (*sno, cno*)

Such decomposition is called *normalization* and is essential if we wish to overcome undesirable anomalies. As noted earlier, normalization often has an adverse effect on performance. Data which could have been retrieved from one relation before normalization may require several relations to be joined after normalization. Normalization does however lead to more efficient updates since an update that

may have required several tuples to be updated before normalization could well need only one tuple to be updated after normalization.

Although in the above case we are able to look at the original relation and propose a suitable decomposition that eliminates the anomalies that we have discussed, in general this approach is not possible. A relation may have one hundred or more attributes and it is then almost impossible for a person to conceptualise all the information and suggest a suitable decomposition to overcome the problems. We therefore need an algorithmic approach to finding if there are problems in a proposed database design and how to eliminate them if they exist.

NOTES

FUNCTIONAL DEPENDENCY

Consider a relation R that has two attributes A and B. The attribute B of the relation is *functionally dependent* on the attribute A if and only if for each value of A no more than one value of B is associated. In other words, the value of attribute A uniquely determines the value of B and if there were several tuples that had the same value of A then all these tuples will have an identical value of attribute B. That is, if t_1 and t_2 are two tuples in the relation R and $t_1(A) = t_2(A)$ then we must have $t_1(B) = t_2(B)$.

A and B need not be single attributes. They could be any subsets of the attributes of a relation R (possibly single attributes). We may then write

R.A -> R.B

if B is functionally dependent on A (or A functionally determines B). Note that functional dependency does not imply a one-to-one relationship between A and B although a one-to-one relationship may exist between A and B.

A simple example of the above functional dependency is when A is a primary key of an entity (*e.g.*, student number) and A is some single-valued property or attribute of the entity (*e.g.*, date of birth). **A -> B** then must always hold. (why?)

Functional dependencies also arise in relationships. Let C be the primary key of an entity and D be the primary key of another entity. Let the two entities have a relationship. If the relationship is one-to-one, we must have **C -> D** and **D -> C**. If the relationship is many-to-one, we would have **C -> D** but not **D -> C**. For many-to-many relationships, no functional dependencies hold. For example, if C is student number and D is subject number, there is no functional dependency between them. If however, we were storing marks and grades in the database as well, we would have

(student_number, subject_number) -> marks

and we might have

marks -> grades

The second functional dependency above assumes that the grades are dependent only on the marks. This may sometime not be true since the instructor may decide to take other considerations into account in assigning grades, for example, the class average mark.

For example, in the student database that we have discussed earlier, we have the following functional dependencies:

sno -> sname

sno -> address

NOTES

cno -> cname

cno -> instructor

instructor -> office

These functional dependencies imply that there can be only one name for each sno, only one address for each student and only one subject name for each cno. It is of course possible that several students may have the same name and several students may live at the same address. If we consider ***cno -> instructor***, the dependency implies that no subject can have more than one instructor (perhaps this is not a very realistic assumption). Functional dependencies therefore place constraints on what information the database may store. In the above example, one may be wondering if the following FDs hold

sname -> sno

cname -> cno

Certainly there is nothing in the instance of the example database presented above that contradicts the above functional dependencies. However, whether above FDs hold or not would depend on whether the university or college whose database we are considering allows duplicate student names and subject names. If it was the enterprise policy to have unique subject names than ***cname -> cno*** holds. If duplicate student names are possible, and one would think there always is the possibility of two students having exactly the same name, then ***sname -> sno*** does not hold.

Functional dependencies arise from the nature of the real world that the database models. Often A and B are facts about an entity where A might be some identifier for the entity and B some characteristic. Functional dependencies cannot be automatically determined by studying one or more instances of a database. They can be determined only by a careful study of the real world and a clear understanding of what each attribute means.

We have noted above that the definition of functional dependency does not require that A and B be single attributes. In fact, A and B may be collections of attributes. For example

(sno, cno) -> (mark, date)

When dealing with a collection of attributes, the concept of *full functional dependence* is an important one. Let A and B be distinct collections of attributes from a relation R end let **R.A -> R.B**. B is then *fully functionally dependent* on A if B is not functionally dependent on any subset of A. The above example of students and subjects would show full functional dependence if *mark* and *date* are not functionally dependent on either student number (*sno*) or subject number (*cno*) alone. The implies that we are assuming that a student may have more than one subjects and a subject would be taken by many different students. Furthermore, it has been assumed that there is at most one enrolment of each student in the same subject.

The above example illustrates full functional dependence. However the following dependence

(sno, cno) -> instructor

is not full functional dependence because **cno -> instructor** holds.

As noted earlier, the concept of functional dependency is related to the concept of candidate key of a relation since a candidate key of a relation is an identifier which

uniquely identifies a tuple and therefore determines the values of all other attributes in the relation. Therefore any subset X of the attributes of a relation R that satisfies the property that all remaining attributes of the relation are functionally dependent on it (that is, on X), then X is candidate key as long as no attribute can be removed from X and still satisfy the property of functional dependence. In the example above, the attributes *(sno, cno)* form a candidate key (and the only one) since they functionally determine all the remaining attributes.

Functional dependence is an important concept and a large body of formal theory has been developed about it. We discuss the concept of *closure* that helps us derive all functional dependencies that are implied by a given set of dependencies. Once a complete set of functional dependencies has been obtained, we will study how these may be used to build normalised relations.

NOTES

MINIMAL FUNCTIONAL DEPENDENCIES

it is useful to define the concept of *minimal functional dependencies* or *minimal cover* which is useful in eliminating unnecessary functional dependencies so that only the minimal number of dependencies need to be enforced by the system. The concept of minimal cover of F is sometimes called *Irreducibe Set* of F. To find the minimal cover of a set of functional dependencies F, we transform F such that each FD in it that has more than one attribute in the right hand side is reduced to a set of FDs that have only one attribute on the right hand side. This can be done easily using the decomposition rule that we have already discussed. The minimal cover of F is then a set of FDs such that:

(*a*) every right hand side of each dependency is a single attribute;

(*b*) for no **X -> A** in F is the set **F - {X -> A}** equivalent to F;

(*c*) for no **X -> A** in F and proper subset Z of X is **F - {X -> A} U {Z -> A}** equivalent to F.

Requirements (*a*), as already noted, can be met easily given any set of dependencies F. Requirement (*b*) guarantees that we cannot remove any dependencies from F and still have a set of dependencies equivalent to F or no attribute on the left hand side of a dependency is redundant. Requirement (*c*) makes sure that no dependencies may be replaced by a dependency that involves a subset of the left hand side.

The concept of minimal set of dependencies is useful in normalization as we shall discuss later. The minimal set may be considered a standard or canonical form of FDs with no redundancies that is equivalent to F. Unfortunately however the minimal cover is not unique.

EQUIVALENT FUNCTIONAL DEPENDENCIES

Let **F1** and **F2** be two sets of FDs. The two FDs are called equivalent if **F+1 = F +2.** Of course, it is not always easy to test that the two sets are equivalent since each of them may consist of hundreds of FDs. One way to carry out the checking would be to take each dependency **X -> Y** in turn from **F+1** and check if it is in **F+2.**

Sometime the term **F1** covers **F2** and **F2** covers **F1** is used to denote equivalence.

NOTES

The concept of multivalued dependencies was developed to provide a basis for decomposition of relations like the one above. Therefore if a relation like *enrolment(sno, subject#)* has a relationship between sno and *subject#* in which sno uniquely determines the values of *subject#*, the dependence of *subject#* on sno is called a *trivial* MVD since the relation enrolment cannot be decomposed any further. More formally, a MVD *X* ->> *Y* is called trivial MVD if either Y is a subset of X or X and Y together form the relation R. The MVD is trivial since it results in no constraints being placed on the relation. Therefore a relation having non-trivial MVDs must have at least three attributes; two of them multivalued. Non-trivial MVDs result in the relation having some constraints on it since all possible combinations of the multivalue attributes are then required to be in the relation.

Let us now define the concept of multivalued dependency. The *multivalued dependency X ->> Y is said to hold for a relation R(X, Y, Z) if for a given set of value (set of values if X is more than one attribute) for attributes X, there is a set of (zero or more) associated values for the set of attributes Y and the Y values depend only on X values and have no dependence on the set of attributes Z.*

In the example above, if there was some dependence between the attributes *qualifications* and *language*, for example perhaps, the language was related to the qualifications (perhaps the qualification was a training certificate in a particular language), then the relation would not have MVD and could not be decomposed into two relations as abve. In the above situation whenever *X* ->> *Y* holds, so does *X* ->> *Z* since the role of the attributes Y and Z is symmetrical.

Consider two different situations.

(*a*) Z is a single valued attribute. In this situation, we deal with R(X, Y, Z) as before by entering several tuples about each entity.

(*b*) Z is multivalued.

Now, more formally, *X* ->> *Y* is said to hold for R(X, Y, Z) if *t*1 and *t*2 are two tuples in R that have the same values for attributes X and therefore with *t*1[*x*] = *t*2[*x*] then R also contains tuples *t*3 and *t*4 (not necessarily distinct) such that

$$t1[x] = t2[x] = t3[x] = t4[x]$$

$$t3[Y] = t1[Y] \text{ and } t3[Z] = t2[Z]$$

$$t4[Y] = t2[Y] \text{ and } t4[Z] = t1[Z]$$

In other words if *t*1 and *t*2 are given by

$$t1 = [X, Y1, Z1], \text{ and}$$

$$t2 = [X, Y2, Z2]$$

then there must be tuples *t*3 and *t*4 such that

$$t3 = [X, Y1, Z2], \text{ and}$$

$$t4 = [X, Y2, Z1]$$

We are therefore insisting that every value of *Y* appears with every value of *Z* to keep the relation instances consistent. In other words, the above conditions insist that *Y* and *Z* are determined by *X* alone and there is no relationship between *Y* and *Z* since *Y* and *Z* appear in every possible pair and hence these pairings present no information and are of no significance. Only if some of these pairings were not present, there would be some significance in the pairings.

Give example (instructor, quals, subjects) --- explain if subject was single valued; otherwise all combinations must occur. Discuss duplication of info in that case.

(**Note:** If *Z* is single-valued and functionally dependent on *X* then *Z1* = *Z2*. If *Z* is multivalue dependent on *X* then *Z1* <> *Z2*).

The theory of multivalued dependencies in very similar to that for functional dependencies. Given *D* a set of MVDs, we may find D^+, the closure of *D* using a set of axioms.

CLOSURE

NOTES

Let a relation R have some functional dependencies *F* specified. The closure of *F* (usually written as **F+**) is the set of all functional dependencies that may be logically derived from *F*. Often *F* is the set of most obvious and important functional dependencies and **F+**, the closure, is the set of all the functional dependencies including *F* and those that can be deduced from *F*. The closure is important and may, for example, be needed in finding one or more candidate keys of the relation.

For example, the student relation has the following functional dependencies

sno -> sname cno -> cname sno -> address cno -> instructor instructor -> office

Let these dependencies be denoted by *F*. The closure of *F*, denoted by **F+**, includes *F* and all functional dependencies that are implied by *F*.

To determine **F+**, we need rules for deriving all functional dependencies that are implied by *F*. A set of rules that may be used to infer additional dependencies was proposed by Armstrong in 1974. These rules (or axioms) are a complete set of rules in that all possible functional dependencies may be derived from them. The rules are:

1. *Reflexivity Rule* : If *X* is a set of attributes and *Y* is a subset of *X*, then ***X -> Y*** holds.
   The reflexivity rule is the most simple (almost trivial) rule. It states that each subset of *X* is functionally dependent on *X*.
2. *Augmentation Rule* : If ***X -> Y*** holds and *W* is a set of attributes, then ***WX -> WY*** holds.
   The augmentation rule is also quite simple. It states that if *Y* is determined by *X* then a set of attributes *W* and *Y* together will be determined by *W* and *X* together. Note that we use the notation *WX* to mean the collection of all attributes in *W* and *X* and write *WX* rather than the more conventional (*W*, *X*) for convenience.
3. *Transitivity Rule* : If ***X -> Y*** and ***Y -> Z*** hold, then ***X -> Z*** holds.
   The transitivity rule is perhaps the most important one. It states that if *X* functionally determines *Y* and *Y* functionally determines *Z* then *X* functionally determines *Z*.

These rules are called *Armstrong's Axioms*.

Further axioms may be derived from the above although the above three axioms are sound and complete in that they do not generate any incorrect functional dependencies (soundness) and they do generate all possible functional dependencies that can be inferred from *F* (completeness). For proof of soundness and completeness of Armstrong's Axioms, the reader is referred to Ullman (Vol 1, page 387). The most important additional axioms are:

1. *Union Rule* : If ***X -> Y*** and ***X -> Z*** hold, then ***X -> YZ*** holds.
2. *Decomposition Rule* : If ***X -> YZ*** holds, then so do ***X -> Y*** and ***X -> Z***.
3. *Pseudotransitivity Rule* : If ***X -> Y*** and ***WY -> Z*** hold then so does ***WX -> Z***.

Based on the above axioms and the functional dependencies specified for relation student, we may write a large number of functional dependencies. Some of these are:

(sno, cno) -> sno (Rule 1)

(sno, cno) -> cno (Rule 1)

(sno, cno) -> (sname, cname) (Rule 2)

cno -> office (Rule 3)

sno -> (sname, address) (Union Rule)

etc.

NOTES

Often a very large list of dependencies can be derived from a given set *F* since Rule 1 itself will lead to a large number of dependencies. Since we have seven attributes *(sno, sname, address, cno, cname, instructor, office)*, there are 128 (that is, 2^7) subsets of these attributes. These 128 subsets could form 128 values of *X* in functional dependencies of the type ***X -> Y***. Of course, each value of *X* will then be associated with a number of values for *Y* (*Y* being a subset of *X*) leading to several thousand dependencies. These large number of dependencies are not particularly helpful in achieving our aim of normalizing relations.

Although we could follow the present procedure and compute the closure of F to find all the functional dependencies, the computation requires exponential time and the list of dependencies is often very large and therefore not very useful. There are two possible approaches that can be taken to avoid dealing with the large number of dependencies in the closure. One is to deal with one attribute or a set of attributes at a time and find its closure (*i.e.*, all functional dependencies relating to them). The aim of this exercise is to find what attributes depend on a given set of attributes and therefore ought to be together. The other approach is to find the *minimal covers*. We will discuss both approaches briefly.

As noted earlier, we need not deal with the large number of dependencies that might arise in a closure since often one is only interested in determining closure of a set of attributes given a set of functional dependencies. Closure of a set of attributes *X* is all the attributes that are functionally dependent on *X* given some functional dependencies *F* while the closure of *F* was all functional dependencies that are implied by *F*. Computing the closure of a set of attributes is a much simpler task if we are dealing with a small number of attributes. We will denote the closure of a set of attributes *X* given *F* by ***X+***.

An algorithm to determine the closure:

Step 1 Let ***X^c <- X***

Step 2 Let the next dependency be ***A -> B***. If *A* is in ***X^c*** and *B* is not, ***X^c <- X^c + B***.

Step 3 Continue step 2 until no new attributes can be added to ***X^c***.

The result of the algorithm is ***X^c*** that is equal to ***X+***.

The above algorithm may also be used to remove redundant dependencies. For example, to check if ***X -> A*** is redundant, we find closure of *X* without using ***X -> A***. If *A* is in ***X^c***, we can eliminate ***X -> A*** as redundant.

Consider the following relation *student(sno, sname, cno, cname).*

We wish to determine the closure of *(sno, cno).* We have the following functional dependencies.

sno -> sname

cno -> cname

We apply the above algorithm using ***X^c*** as the place holder for all the attributes that have been found to be dependent on X so far.

Step 1 : ***X^c <- X***, that is, ***X^c <- (sno, cno)***

Step 2 : Consider ***sno -> sname***, since *sno* is in ***X^c*** and *sname* is not, we have ***X^c <- (sno, cno) + sname***

Step 3 : Consider ***cno -> cname***, since *cno* is in ***X^c*** and *cname* is not, we have ***X^c <- (sno, cno, sname) + cname***

Step 4 : Again, consider ***sno -> sname*** but this does not change ***X^c***.

Step 5 : Again, consider ***cno -> cname*** but this does not change ***X^c***.

Therefore ***X+ = X^c = (sno, cno, sname, cname)***.

This shows that all the attributes in the relation student *(sno, cno, sname, cname)* are dependent on *(sno, cno)* and therefore *(sno, cno)* is a candidate key of the present relation. In this case, it is the only candidate key.

Similarly we may wish to investigate the closure of *(sname, cname)*. We will find that the closure of *(sname, cname)* is only *(sname, cname)*. Therefore these two attributes together cannot form the key of the above relation.

NOTES

DESIRABLE PROPERTIES OF DECOMPOSITION

So far our approach has consisted of looking at individual relations and checking if they belong to 2NF, 3NF or BCNF. If a relation was not in the normal form that was being checked for and we wished the relation to be normalised to that normal form so that some of the anomalies can be eliminated, it was necessary to decompose the relation in two or more relations. The process of decomposition of a relation *R* into a set of relations *R1* , *R2* , ..., *Rn* was based on identifying different components and using that as a basis of decomposition. The decomposed relations *R1* , *R2* , ..., *Rn* are projections of *R* and are of course not disjoint otherwise the glue holding the information together would be lost. Decomposing relations in this way based on a recognise and split method is not a particularly sound approach since we do not even have a basis to determine that the original relation can be constructed if necessary from the decomposed relations. We now discuss desirable properties of good decomposition and identify difficulties that may arise if the decomposition is done without adequate care. The next section will discuss how such decomposition may be derived given the FDs.

Desirable properties of a decomposition are:

1. Attribute preservation
2. Lossless-join decomposition
3. Dependency preservation
4. Lack of redundancy

Lossless-Join Decomposition

We decomposed a relation intuitively. We need a better basis for deciding decompositions since intuition may not always be correct. We illustrate how a careless decomposition may lead to problems including loss of information.

Consider the following relation

enrol (sno, cno, date-enrolled, room-No., instructor)

Suppose we decompose the above relation into two relations enrol1 and enrol2 as follows

enrol1 (sno, cno, date-enrolled)

enrol2 (date-enrolled, room-No., instructor)

There are problems with this decomposition but we wish to focus on one aspect at the moment. Let an instance of the relation enrol be

NOTES

sno	cno	date-enrolled	room-No.	instructor
830057	CP302	1FEB1984	MP006	Gupta
830057	CP303	1FEB1984	MP006	Jones
820159	CP302	10JAN1984	MP006	Gupta
825678	CP304	1FEB1984	CE122	Wilson
826789	CP305	15JAN1984	EA123	Smith

and let the decomposed relations *enrol1* and *enrol2* be:

sno	cno	date-enrolled
830057	CP302	1FEB1984
830057	CP303	1FEB1984
820159	CP302	10JAN1984
825678	CP304	1FEB1984
826789	CP305	15JAN1984

date-enrolled	room-No.	instructor
1FEB1984	MP006	Gupta
1FEB1984	MP006	Jones
10JAN1984	MP006	Gupta
1FEB1984	CE122	Wilson
15JAN1984	EA123	Smith

All the information that was in the relation *enrol* appears to be still available in *enrol1* and *enrol2* but this is not so. Suppose, we wanted to retrieve the student numbers of all students taking a course from *Wilson*, we would need to join *enrol1* and *enrol2*. The join would have 11 tuples as follows:

sno	cno	date-enrolled	room-No.	instructor
830057	CP302	1FEB1984	MP006	Gupta
830057	CP302	1FEB1984	MP006	Jones
830057	CP303	1FEB1984	MP006	Gupta
830057	CP303	1FEB1984	MP006	Jones
830057	CP302	1FEB1984	CE122	Wilson
830057	CP303	1FEB1984	CE122	Wilson

(add further tuples ...)

The join contains a number of spurious tuples that were not in the original relation Enrol. Because of these additional tuples, we have lost the information about which students take courses from WILSON. (Yes, we have more tuples but less information because we are unable to say with certainty who is taking courses

from WILSON). Such decompositions are called *lossy* decompositions. A *nonloss* or *lossless* decomposition is that which guarantees that the join will result in exactly the same relation as was decomposed. One might think that there might be other ways of recovering the original relation from the decomposed relations but, sadly, no other operators can recover the original relation if the join does not (why?).

We need to analyse why some decompositions are lossy. The common attribute in above decompositions was Date-enrolled. The common attribute is the glue that gives us the ability to find the relationships between different relations by joining the relations together. If the common attribute is not unique, the relationship information is not preserved. If each tuple had a unique value of Date-enrolled, the problem of losing information would not have existed. The problem arises because several enrolments may take place on the same date.

NOTES

A decomposition of a relation R into relations $R_1, R_2, ..., R_n$ is called a *lossless-join* decomposition (with respect to FDs F) if the relation R is always the natural join of the relations $R_1, R_2, ..., R_n$. It should be noted that natural join is the only way to recover the relation from the decomposed relations. There is no other set of operators that can recover the relation if the join cannot. Furthermore, it should be noted when the decomposed relations $R_1, R_2, ..., R_n$ are obtained by projecting on the relation R, for example R_1 by projection $pi_1 (R)$, the relation R_1 may not always be precisely equal to the projection since the relation R_1 might have additional tuples called the *dangling* tuples. Explain ...

It is not difficult to test whether a given decomposition is lossless-join given a set of functional dependencies F. We consider the simple case of a relation R being decomposed into R_1 and R_2. If the decomposition is lossless-join, then one of the following two conditions must hold

$(R_1 \text{ intersection } R_2) \rightarrow (R_1 - R_2)$
$(R_1 \text{ intersection } R_2) \rightarrow (R_2 - R_1)$

DEPENDENCY PRESERVATION

It is clear that a decomposition must be lossless so that we do not lose any information from the relation that is decomposed. Dependency preservation is another important requirement since a dependency is a constraint on the database and if $X \rightarrow Y$ holds than we know that the two (sets) attributes are closely related and it would be useful if both attributes appeared in the same relation so that the dependency can be checked easily.

Let us consider a relation $R(A, B, C, D)$ that has the dependencies F that include the following:

$A \rightarrow B$

$A \rightarrow B$

etc

If we decompose the above relation into $R1(A, B)$ and $R2(B, C, D)$ the dependency $A \rightarrow C$ cannot be checked (or preserved) by looking at only one relation. It is desirable that decompositions be such that each dependency in F may be checked by looking at only one relation and that no joins need be computed for checking dependencies. In some cases, it may not be possible to preserve each and every dependency in F but as long as the dependencies that are preserved are equivalent to F, it should be sufficient.

Let F be the dependencies on a relation R which is decomposed in relations $R_1, R_2, ..., R_n$.

We can partition the dependencies given by F such that F_1, F_2, ..., F_n. F_n are dependencies that only involve attributes from relations R_1, R_2, ..., R_n respectively. If the union of dependencies F_i imply all the dependencies in F, then we say that the decomposition has preserved dependencies, otherwise not.

If the decomposition does not preserve the dependencies F, then the decomposed relations may contain relations that do not satisfy F or the updates to the decomposed relations may require a join to check that the constraints implied by the dependencies still hold.

NOTES

(Need an example) (Need to discuss testing for dependency preservation with an example... Ullman page 400)

Consider the following relation

sub(sno, instructor, office)

We may wish to decompose the above relation to remove the transitive dependency of *office* on *sno*. A possible decomposition is

S1(sno, instructor)

S2(sno, office)

The relations are now in 3NF but the dependency *instructor -> office* cannot be verified by looking at one relation; a join of *S1* and *S2* is needed. In the above decomposition, it is quite possible to have more than one office number for one instructor although the functional dependency *instructor -> office* does not allow it.

WHAT IS NORMALISATION?

Normalisation is the process of taking data from a problem and reducing it to a set of relations while ensuring data integrity and eliminating data redundancy

- Data integrity : all of the data in the database are consistent, and satisfy all integrity constraints.
- Data redundancy : if data in the database can be found in two different locations (direct redundancy) or if data can be calculated from other data items (indirect redundancy) then the data is said to contain redundancy.

Data should only be stored once and avoid storing data that can be calculated from other data already held in the database. During the process of normalisation redundancy must be removed, but not at the expense of breaking data integrity rules.

If redundancy exists in the database then problems can arise when the database is in normal operation:

- When data is inserted the data must be duplicated correctly in all places where there is redundancy. For instance, if two tables exist for in a database, and both tables contain the employee name, then creating a new employee entry requires that both tables be updated with the employee name.
- When data is modified in the database, if the data being changed has redundancy, then all versions of the redundant data must be updated simultaneously. So in the employee example a change to the employee name must happen in both tables simultaneously.

The removal of redundancy helps to prevent insertion, deletion, and update errors, since the data is only available in one attribute of one table in the database.

The data in the database can be considered to be in one of a number of 'normal form's'. Basically the normal form of the data indicates how much redundancy is in that data. The normal forms have a strict ordering:

1. 1st Normal Form
2. 2nd Normal Form

3. 3rd Normal Form
4. BCNF

There are other normal forms, such as 4th and 5th normal forms. They are rarely utilised in system design and are not considered further here.

To be in a particular form requires that the data meets the criteria to also be in all normal forms before that form. Thus to be in 2nd normal form the data must meet the criteria for both 2nd normal form and 1st normal form. The higher the form the more redundancy has been eliminated.

NOTES

Integrity Constraints

An integrity constraint is a rule that restricts the values that may be present in the database. The relational data model includes constraints that are used to verify the validity of the data as well as adding meaningful structure to it:

- entity integrity :

The rows (or tuples) in a relation represent entities, and each one must be uniquely identified. Hence we have the primary key that must have a unique non-null value for each row.

- referential integrity :

This constraint involves the foreign keys. Foreign keys tie the relations together, so it is vitally important that the links are correct. Every foreign key must either be null or its value must be the actual value of a key in another relation.

Understanding Data

Sometimes the starting point for understanding data is given in the form of relations and functional dependancies. This would be the case where the starting point in the process was a detailed specification of the problem. We already know what relations are. Functional dependancies are rules stating that given a certain set of attributes (the determinant) determines a second set of attributes.

The definition of a functional dependency looks like A->B. In this case B is a single attribute but it can be as many attributes as required (for instance, X->J,K,L,M). In the functional dependency, the determinant (the left hand side of the -> sign) can determine the set of attributes on the right hand side of the -> sign. This basically means that A selects a particular value for B, and that A is unique. In the second example X is unique and selects a particular set of values for J,K,L, and M. It can also be said that B is functionally dependent on A. In addition, a particular value of A ALWAYS gives you a particular value for B, but not vice-versa.

Consider this example:

R(matric_no, firstname, surname, tutor_number, tutor_name)

tutor_number -> tutor_name

Here there is a relation R, and a functional dependency that indicates that:

- instances of tutor_number are unique in the data
- from the data, given a tutor_number, it is always possible to work out the tutor_name.
- As an example tutor number 1 may be "Mr Smith", but tutor number 10 may also be "Mr Smith". Given a tutor number of 1, this is ALWAYS "Mr Smith". However, given the name "Mr Smith" it is not possible to work out if we are talking about tutor 1 or tutor 10.

There is actually a second functional dependency for this relation, which can be worked out from the relation itself. As the relation has a primary key, then given this attribute you can determine all the other attributes in R. This is an implied functional dependency and is not normally listed in the list of functional dependents.

NOTES

Extracting understanding

It is possible that the relations and the determinants have not yet been defined for a problem, and therefore must be calculated from examples of the data. Consider the following Student table.

Student - an unnormalised tablewith repeating groups

matric_no	Name	date_of_birth	subject	grade
960100	Smith, J	14/11/1977	Databases Soft_Dev ISDE	C A D
960105	White, A	10/05/1975	Soft_Dev ISDE	B B
960120	Moore, T	11/03/1970	Databases Soft_Dev Workshop	A B C
960145	Smith, J	09/01/1972	Databases	B
960150	Black, D	21/08/1973	Databases Soft_Dev ISDE Workshop	B D C D

The subject/grade pair is repeated for each student. 960145 has 1 pair while 960150 has four. Repeating groups are placed inside another set of parentheses. From the table the following relation is generated:

Student(matric_no, name, date_of_birth, (subject, grade))

The repeating group needs a key in order that the relation can be correctly defined. Looking at the data one can see that grade repeats within matric_no (for instance, for 960150, the student has 2 D grades). However, subject never seems to repeat for a single matric_no, and therefore is a candidate key in the repeating group.

Whenever keys or dependencies are extracted from example data, the information extracted is only as good as the data sample examined. It could be that another data sample disproves some of the key selections made or dependencies extracted. What is important however is that the information extracted during these exercises is correct for the data being examined.

Looking at the data itself, we can see that the same name appears more than once in the name column. The name in conjunction with the date_of_birth seems to be unique, suggesting a functional dependency of:

name, date_of_birth -> matric_no

This implies that not only is the matric_no sufficient to uniquely identify a student, the student's name combined with the date of birth is also sufficient to uniquely identify a student. It is therefore possible to have the relation Student written as:

Student(matric_no, name, date_of_birth, (subject, grade))

As guidance in cases where a variety of keys could be selected one should try to select the relation with the least number of attributes defined as primary keys.

Flattened Tables

Note that the student table shown above explicitly identifies the repeating group. It is also possible that the table presented will be what is called a flat table, where the repeating group is not explicitly shown:

Student #2 - Flattened Table

matric_no	name	date_of_birth	Subject	grade
960100	Smith, J	14/11/1977	Databases	C
960100	Smith, J	14/11/1977	Soft_Dev	A
960100	Smith, J	14/11/1977	ISDE	D
960105	White, A	10/05/1975	Soft_Dev	B
960105	White, A	10/05/1975	ISDE	B
960120	Moore, T	11/03/1970	Databases	A
960120	Moore, T	11/03/1970	Soft_Dev	B
960120	Moore, T	11/03/1970	Workshop	C
960145	Smith, J	09/01/1972	Databases	B
960150	Black, D	21/08/1973	Databases	B
960150	Black, D	21/08/1973	Soft_Dev	D
960150	Black, D	21/08/1973	ISDE	C
960150	Black, D	21/08/1973	Workshop	B

NOTES

The table still shows the same data as the previous example, but the format is different. We have removed the repeating group (which is good) but we have introduced redundancy (which is bad).

Sometimes you will miss spotting the repeating group, so you may produce something like the following relation for the Student data.

Student(matric_no, name, date_of_birth, subject, grade)

matric_no -> name, date_of_birth

name, date_of_birth -> matric_no

This data does not explicitly identify the repeating group, but as you will see the result of the normalisation process on this relation produces exactly the same relations as the normalisation of the version that explicitly does have a repeating group.

First Normal Form

- First normal form (1NF) deals with the `shape' of the record type
- A relation is in 1NF if, and only if, it contains no repeating attributes or groups of attributes.
- Example:
- The Student table with the repeating group is not in 1NF
- It has repeating groups, and it is called an `unnormalised table'.

Relational databases require that each row only has a single value per attribute, and so a repeating group in a row is not allowed.

To remove the repeating group, one of two things can be done:

- either flatten the table and extend the key, or
- decompose the relation- leading to First Normal Form

Flatten table and Extend Primary Key

The Student table with the repeating group can be written as:

Student(matric_no, name, date_of_birth, (subject, grade))

If the repeating group was flattened, as in the Student #2 data table, it would look something like:

Student(matric_no, name, date_of_birth, subject, grade)

Although this is an improvement, we still have a problem. matric_no can no longer be the primary key - it does not have an unique value for each row. So we have to find a new primary key - in this case it has to be a compound key since no single attribute can uniquely identify a row. The new primary key is a compound key (matrix_no + subject).

NOTES

We have now solved the repeating groups problem, but we have created other complications. Every repetition of the matric_no, name, and data_of_birth is redundant and liable to produce errors.

With the relation in its flattened form, strange anomalies appear in the system. Redundant data is the main cause of insertion, deletion, and updating anomalies.

Insertion anomaly:

With the primary key including subject, we cannot enter a new student until they have at least one subject to study. We are not allowed NULLs in the primary key so we must have an entry in both matric_no and subject before we can create a new record.

- This is known as the insertion anomaly. It is difficult to insert new records into the database.
- On a practical level, it also means that it is difficult to keep the data up to date.

Update anomaly

If the name of a student were changed for example Smith, J. was changed to Green, J. this would require not one change but many one for every subject that Smith, J. studied.

Deletion anomaly

If all of the records for the `Databases' subject were deleted from the table,we would inadvertently lose all of the information on the student with matric_no 960145. This would be the same for any student who was studying only one subject and the subject was deleted. Again this problem arises from the need to have a compound primary key.

Decomposing the relation

The alternative approach is to split the table into two parts, one for the repeating groups and one of the non-repeating groups.

the primary key for the original relation is included in both of the new relations

Record

matric_no	subject	grade
960100	Databases	C
960100	Soft_Dev	A
960100	ISDE	D
960105	Soft_Dev	B
960105	ISDE	B
...	...	...
960150	Workshop	B

Student

matric_no	name	date_of_birth
960100	Smith,J	14/11/1977
960105	White,A	10/05/1975
960120	Moore,T	11/03/1970
960145	Smith,J	09/01/1972
960150	Black,D	21/08/1973

NOTES

- We now have two relations, Student and Record.
- Student contains the original non-repeating groups.
- Record has the original repeating groups and the matric_no

Student(matric_no, name, date_of_birth)
Record(matric_no, subject, grade)
Matric_no remains the key to the Student relation. It cannot be the complete key to the new Record relation - we end up with a compound primary key consisting of matric_no and subject. The matric_no is the link between the two tables - it will allow us to find out which subjects a student is studying. So in the Record relation, matric_no is the foreign key.
This method has eliminated some of the anomalies. It does not always do so, it depends on the example chosen

- In this case we no longer have the insertion anomaly.
- It is now possible to enter new students without knowing the subjects that they will be studying
- They will exist only in the Student table, and will not be entered in the Record table until they are studying at least one subject.
- We have also removed the deletion anomaly
- If all of the `databases' subject records are removed, student 960145 still exists in the Student table.
- We have also removed the update anomaly

Student and Record are now in First Normal Form.

Second Normal Form

Second normal form (or 2NF) is a more stringent normal form defined as:

A relation is in 2NF if, and only if, it is in 1NF and every non-key attribute is fully functionally dependent on the whole key.

Thus the relation is in 1NF with no repeating groups, and all non-key attributes must depend on the whole key, not just some part of it. Another way of saying this is that there must be no partial key dependencies (PKDs).

The problems arise when there is a compound key, *e.g.,* the key to the Record relation - matric_no, subject. In this case it is possible for non-key attributes to depend on only part of the key - *i.e.,* on only one of the two key attributes. This is what 2NF tries to prevent.

Consider again the Student relation from the flattened Student #2 table:

Student(matric_no, name, date_of_birth, subject, grade)

- There are no repeating groups
- The relation is already in 1NF
- However, we have a compound primary key - so we must check all of the non-key attributes against each part of the key to ensure they are functionally dependent on it.
- matric_no determines name and date_of_birth, but not grade.

NOTES

- subject together with matric_no determines grade, but not name or date_of_birth.
- So there is a problem with potential redundancies

A dependency diagram is used to show how non-key attributes relate to each part or combination of parts in the primary key.

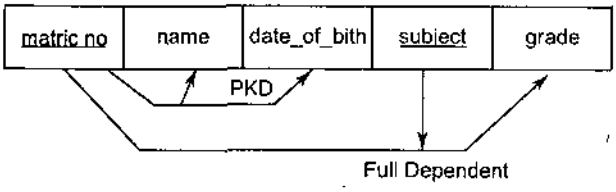
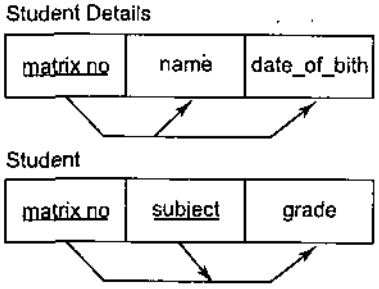


Figure 1. Dependency Diagram

- ❒ This relation is not in 2NF
- ❒ It appears to be two tables squashed into one.
- ❒ The solution is to split the relation up into its component parts.
- ❒ Separate out all the attributes that are solely dependent on matric_no
- ❒ Put them in a new Student_details relation, with matric_no as the primary key
- ❒ Separate out all the attributes that are solely dependent on subject.
- ❒ In this case no attributes are solely dependent on subject.
- ❒ Separate out all the attributes that are solely dependent on matric_no + subject
- ❒ Put them into a separate Student relation, keyed on matric_no + subject



All attributes in each relation are fully functionally dependent upon its primary key

These relations are now in 2NF

Figure 2. Dependencies after splitting

Interestingly this is the same set of relations as when we recognized that there were repeating terms in the table and directly removed the repeating terms. It should not really matter what process you followed when normalizing, as the end result should be similar relations.

Third Normal Form

3NF is an even stricter normal form and removes virtually all the redundant data :

- A relation is in 3NF if, and only if, it is in 2NF and there are no transitive functional dependencies
- Transitive functional dependencies arise:
- When one non-key attribute is functionally dependent on another non-key attribute:
- FD: non-key attribute -> non-key attribute
- And when there is redundancy in the database

By definition transitive functional dependency can only occur if there is more than one non-key field, so we can say that a relation in 2NF with zero or one non-key field must automatically be in 3NF.

project_no	manager	address
p1	Black,B	32 High Street
p2	Smith,J	11 New Street
p3	Black,B	32 High Street
p4	Black,B	32 High Street

NOTES

Project has more than one non-key field so we must check for transitive dependency:

- Address depends on the value in the manager column
- Every time B Black is listed in the manager column, the address column has the value '32 High Street'. From this the relation and functional dependency can be implied as:
  Project(project_no, manager, address)
  manager -> address
- In this case address is transitively dependent on manager. Manager is the determinant - it determines the value of address. It is transitive functional dependency only if all attributes on the left of the "->" are not in the key but are all in the relation, and all attributes to the right of the "->" are not in the key with at least one actually being in the relation.
- Data redundancy arises from this
- We duplicate address if a manager is in charge of more than one project
- Causes problems if we had to change the address- have to change several entries, and this could lead to errors.
- The solution is to eliminate transitive functional dependency by splitting the table
- Create two relations - one with the transitive dependency in it, and another for all of the remaining attributes.
- Split Project into Project and Manager.
- The determinant attribute becomes the primary key in the new relation
- Manager becomes the primary key to the Manager relation.
- The original key is the primary key to the remaining non-transitive attributes
- In this case, project_no remains the key to the new Projects table.

Project

project_no	manager
p1	Black,B
p2	Smith,J
p3	Black,B
p4	Black,B

Manager

manager	address
Black,B	32 High Street
Smith,J	11 New Street

- Now we need to store the address only once
- If we need to know a manager's address we can look it up in the Manager relation
- The manager attribute is the link between the two tables, and in the Projects table it is now a foreign key.
- These relations are now in third normal form.

STUDENT ACTIVITY

1. What is normalization?

2. What is a normal form?

3. What are different normal forms?

4. What are Keys?

NOTES

Summary: 1NF

- A relation is in 1NF if it contains no repeating groups.
- To convert an unnormalised relation to 1NF either:
- Flatten the table and change the primary key, or
- Decompose the relation into smaller relations, one for the repeating groups and one for the non-repeating groups.
- Remember to put the primary key from the original relation into both new relations.
- This option is liable to give the best results.

Summary: 2NF

- A relation is in 2NF if it contains no repeating groups and no partial key functional dependencies.
- Rule: A relation in 1NF with a single key field must be in 2NF.
- To convert a relation with partial functional dependencies to 2NF. Create a set of new relations:
- One relation for the attributes that are fully dependent upon the key.
- One relation for each part of the key that has partially dependent attributes.

Summary: 3NF

- A relation is in 3NF if it contains no repeating groups, no partial functional dependencies, and no transitive functional dependencies.
- To convert a relation with transitive functional dependencies to 3NF, remove the attributes involved in the transitive dependency and put them in a new relation.
- Rule: A relation in 2NF with only one non-key attribute must be in 3NF
- In a normalised relation a non-key field must provide a fact about the key, the whole key and nothing but the key.
- Relations in 3NF are sufficient for most practical database design problems. However, 3NF does not guarantee that all anomalies have been removed.

BOYCE-CODD NORMAL FORM (BCNF)

- When a relation has more than one candidate key, anomalies may result even though the relation is in 3NF.
- 3NF does not deal satisfactorily with the case of a relation with overlapping candidate keys.
- *i.e.,* composite candidate keys with at least one attribute in common.
- BCNF is based on the concept of a *determinant*.
- A determinant is any attribute (simple or composite) on which some other attribute is fully functionally dependent.
- A relation is in BCNF is, and only if, every determinant is a candidate key.

Consider the following relation and determinants.

R(a,b,c,d)
a,c -> b,d
a,d -> b

Here, the first determinant suggests that the primary key of R could be changed from a,b to a,c. If this change was done all of the non-key attributes present in R could still be determined, and therefore this change is legal. However, the second determinant indicates that a,d determines b, but a,d could not be the key of R as a,d does not determine all of the non key attributes of R (it does not determine c). We would say that the first determinate is a candidate key, but the second

determinant is not a candidate key, and thus this relation is not in BCNF (but is in 3rd normal form).

Normalisation to BCNF - Example 1

Patient No	Patient Name	Appointment Id	Time	Doctor
1	John	0	09:00	Zorro
2	Kerr	0	09:00	Killer
3	Adam	1	10:00	Zorro
4	Robert	0	13:00	Killer
5	Zane	1	14:00	Zorro

NOTES

Lets consider the database extract shown above. This depicts a special dieting clinic where the each patient has 4 appointments. On the first they are weighed, the second they are exercised, the third their fat is removed by surgery, and on the fourth their mouth is stitched closed... Not all patients need all four appointments! If the Patient Name begins with a letter before "P" they get a morning appointment, otherwise they get an afternoon appointment. Appointment 1 is either 09:00 or 13:00, appointment 2 10:00 or 14:00, and so on. From this (hopefully) make-believe scenario we can extract the following determinants:

DB(Patno,PatName,appNo,time,doctor)
Patno -> PatName
Patno,appNo -> Time,doctor
Time -> appNo

Now we have to decide what the primary key of DB is going to be. From the information we have, we could chose:

DB(Patno,PatName,appNo,time,doctor) (example 1a)

or

DB(Patno,PatName,appNo,time,doctor) (example 1b)

Example 1a - DB(Patno,PatName,appNo,time,doctor)

- 1NF Eliminate repeating groups.

None:

DB(Patno,PatName,appNo,time,doctor)

- 2NF Eliminate partial key dependencies

DB(Patno,appNo,time,doctor)

R1(Patno,PatName)

- 3NF Eliminate transitive dependencies

None: so just as 2NF

- BCNF Every determinant is a candidate key
  DB(Patno,appNo,time,doctor)
  R1(Patno,PatName)
- Go through all determinates where ALL of the left hand attributes are present in a relation and at least ONE of the right hand attributes are also present in the relation.
- Patno -> PatName
  Patno is present in DB, but not PatName, so not relevant.
- Patno,appNo -> Time,doctor
  All LHS present, and time and doctor also present, so relevant. Is this a candidate key? Patno,appNo IS the key, so this is a candidate key. Thus this is OK for BCNF compliance.

- Time -> appNo
  Time is present, and so is appNo, so relevant. Is this a candidate key. If it was then we could rewrite DB as:
  DB(Patno,appNo,time,doctor)
  This will not work, as you need both time and Patno together to form a unique key. Thus this determinate is not a candidate key, and therefore DB is not in BCNF. We need to fix this.
- BCNF: rewrite to
  DB(Patno,time,doctor)
  R1(Patno,PatName)
  R2(time,appNo)

NOTES

time is enough to work out the appointment number of a patient. Now BCNF is satisfied, and the final relations shown are in BCNF.

Example 1b - DB(Patno,PatName,appNo,time,doctor)

- 1NF Eliminate repeating groups.

None:

DB(Patno,PatName,appNo,time,doctor)

2NF Eliminate partial key dependencies

DB(Patno,time,doctor)

R1(Patno,PatName)

R2(time,appNo)

- 3NF Eliminate transitive dependencies

None: so just as 2NF

- BCNF Every determinant is a candidate key
- DB(Patno,time,doctor)
- R1(Patno,PatName)
- R2(time,appNo)
- Go through all determinates where ALL of the left hand attributes are present in a relation and at least ONE of the right hand attributes are also present in the relation.
- Patno -> PatName
  Patno is present in DB, but not PatName, so not relevant.
- Patno,appNo -> Time,doctor
  Not all LHS present, so not relevant.
- Time -> appNo
  Time is present, and so is appNo, so relevant. This is a candidate key. However, Time is currently the key for R2, so satisfies the rules for BCNF.
- BCNF: as 3NF
  DB(Patno,time,doctor)
  R1(Patno,PatName)
  R2(time,appNo)

Summary - Example 1

This example has demonstrated three things:

- BCNF is stronger than 3NF, relations that are in 3NF are not necessarily in BCNF.
- BCNF is needed in certain situations to obtain full understanding of the data model
- There are several routes to take to arrive at the same set of relations in BCNF.
- Unfortunately there are no rules as to which route will be the easiest one to take.

NOTES

Example 2

Grade_report(StudNo,StudName,(Major,Adviser,
(CourseNo,Ctitle,InstrucName,InstructLocn,Grade)))

- Functional dependencies

StudNo -> StudName

CourseNo -> Ctitle,InstrucName

InstrucName -> InstrucLocn

StudNo,CourseNo,Major -> Grade

StudNo,Major -> Advisor

Advisor -> Major

- Unnormalised

Grade_report(StudNo,StudName,(Major,Advisor,
(CourseNo,Ctitle,InstrucName,InstructLocn,Grade)))

- 1NF Remove repeating groups

Student(StudNo,StudName)

StudMajor(StudNo,Major,Advisor)

StudCourse(StudNo,Major,CourseNo,
Ctitle,InstrucName,InstructLocn,Grade)

- 2NF Remove partial key dependencies

Student(StudNo,StudName)

StudMajor(StudNo,Major,Advisor)

StudCourse(StudNo,Major,CourseNo,Grade)

Course(CourseNo,Ctitle,InstrucName,InstructLocn)

3NF Remove transitive dependencies

Student(StudNo,StudName)

StudMajor(StudNo,Major,Advisor)

StudCourse(StudNo,Major,CourseNo,Grade)

Course(CourseNo,Ctitle,InstrucName)

Instructor(InstructName,InstructLocn)

- BCNF Every determinant is a candidate key
- Student : only determinant is StudNo
- StudCourse: only determinant is StudNo,Major
- Course: only determinant is CourseNo
- Instructor: only determinant is InstrucName
- StudMajor: the determinants are
- StudNo,Major, or
- Adviser

Only StudNo,Major is a candidate key.

- BCNF

Student(StudNo,StudName)

StudCourse(StudNo,Major,CourseNo,Grade)

Course(CourseNo,Ctitle,InstrucName)

Instructor(InstructName,InstructLocn)

StudMajor(StudNo,Advisor)

Adviser(Adviser,Major)

NOTES

Problems BCNF overcomes

STUDENT	MAJOR	ADVISOR
123	PHYSICS	EINSTEIN
123	MUSIC	MOZART
456	BIOLOGY	DARWIN
789	PHYSICS	BOHR
999	PHYSICS	EINSTEIN

- If the record for student 456 is deleted we lose not only information on student 456 but also the fact that DARWIN advises in BIOLOGY
- We cannot record the fact that WATSON can advise on COMPUTING until we have a student majoring in COMPUTING to whom we can assign WATSON as an advisor.

In BCNF we have two tables:

STUDENT	ADVISOR
123	EINSTEIN
123	MOZART
456	DARWIN
789	BOHR
999	EINSTEIN

ADVISOR	MAJOR
EINSTEIN	PHYSICS
MOZART	MUSIC
DARWIN	BIOLOGY
BOHR	PHYSICS

Returning to the ER Model

- Now that we have reached the end of the normalisation process, you must go back and compare the resulting relations with the original ER model.
- You may need to alter it to take account of the changes that have occurred during the normalisation process Your ER diagram should always be a prefect reflection of the model you are going to implement in the database, so keep it up to date!
- The changes required depends on how good the ER model was at first!

Normalisation Example

Library

Consider the case of a simple video library. Each video has a title, director, and serial number. Customers have a name, address, and membership number. Assume only one copy of each video exists in the library. We are given:

video(title,director,serial)

customer(name,addr,memberno)

hire(memberno,serial,date)
title->director,serial
serial->title
serial->director
name,addr -> memberno
memberno -> name,addr
serial,date -> memberno

What normal form is this?

- No repeating groups, so at least 1NF
- 2NF? There is a composite key in hire. Investigate further... Can memberno in hire be found with just serial or just date. NO. Therefore relation is in at least 2NF.
- 3NF? serial->director is a non-key dependency. Therefore the relations are currently in 2NF.

Convert from 2NF to 3NF.

Rewrite
video(title,director,serial)
To
video(title,serial)
serial(serial,director)

Therefore the new relations become:
video(title,serial)
serial(serial,director)
customer(name,addr,memberno)
hire(memberno,serial,date)

In BCNF? Check if every determinant is a candidate key.

video(title,serial)
Determinants are:
title->director,serial **Candidate key**
serial->title **Candidate key**
video in BCNF

serial(serial,director)
Determinants are:
serial->director **Candidate key**
serial in BCNF

customer(name,addr,memberno)
Determinants are:
name,addr -> memberno **Candidate key**
memberno -> name,addr **Candidate key**
customer in BCNF

hire(memberno,serial,date)
Determinants are:
serial,date -> memberno **Candidate key**
hire in BCNF

Therefore the relations are also now in BCNF.

NOTES

NOTES

SUMMARY

We have noted that relations that form the database must satisfy some properties, for example, relations have no duplicate tuples, tuples have no ordering associated with them, and each element in the relation is atomic. Relations that satisfy these basic requirements may still have some undesirable properties, for example, data redundancy and update anomalies.

Functional dependence is an important concept and a large body of formal theory has been developed about it. We discuss the concept of closure that helps us derive all functional dependencies that are implied by a given set of dependencies. Once a complete set of functional dependencies has been obtained,

It is useful to define the concept of *minimal functional dependencies or minimal* cover which is useful in eliminating unnecessary functional dependencies so that only the minimal number of dependencies need to be enforced by the system.

A decomposition must be lossless so that we do not lose any information from the relation that is decomposed. Dependency preservation is another important requirement since a dependency is a constraint on the database and if $X -> Y$ holds than we know that the two (sets) attributes are closely related and it would be useful if both attributes appeared in the same relation so that the dependency can be checked easily.

Normalisation is the process of taking data from a problem and reducing it to a set of relations while ensuring data integrity and eliminating data redundancy

- Data integrity : all of the data in the database are consistent, and satisfy all integrity constraints.
- Data redundancy : if data in the database can be found in two different locations (direct redundancy) or if data can be calculated from other data items (indirect redundancy) then the data is said to contain redundancy.

TEST YOURSELF

1. What are the differences between intelligent and non-intelligent keys?
2. What is a primary key?
3. What is a foreign key?
4. What are the different types of relations between the entities in a table?
5. What is a one-to-one relationship? Give examples.
6. What is a one-to-many relationship? Give examples.
7. What is a many-to-many relationship? Give examples.
8. What is the first normal form (1NF)?
9. Explain second normal form (2NF)?
10. What is third normal form (3NF)?
11. What is an insertion anomaly?
12. What do you mean by a deletion anomaly?
13. What is a modification anomaly?
14. Explain the Boyce-Codd normal form (BCNF)?
15. What do you mean by functional dependency?
16. What is transitive dependency?
17. What is the fourth normal form (4NF)?
18. What are multi-valued dependencies or MVDs?
19. What is the fifth normal form (5NF)?
20. Explain the Domain-Key normal form (DKNF)?
21. What is denormalization?

22. Why is denormalization needed?
23. When do you perform denormalization?
24. What are the benefits of denormalization?

Fill in the Blanks

1. _____ is the formal process for deciding which attributes should be grouped together in a relation.
2. In the _____ process we analyze and decompose the complex relations and transform them into smaller, simpler and well-structured relations.
3. _____ first developed the process of normalization.
4. _____ is the process of building database structures to store data.
5. When the multi-valued attributes or repeating groups in a relation are removed then that relation is said to be in the _____.
6. A relation is said to be in the _____ when the transitive dependencies are removed or the columns not dependent on the key are eliminated.
7. A _____ uniquely identifies a row in a table.
8. There are two types of keys: _____ and _____.
9. An _____ in based upon data values such as a date, a last name or a combination of values.
10. A _____ is completely arbitrary, having no function or meaning other than identification of the row.
11. A _____ is a column in the table whose purpose is to uniquely identify records from the same table.
12. A _____ is a column in a table that uniquely identifies the records from a different table.
13. _____ are designed to work most effectively with one-to-many relationships between table, expressed using primary and foreign keys.
14. Each _____ may have any number of foreign keys using the same value, in any number of tables.
15. Each pair of primary and foreign key columns is a _____ relationship.
16. A _____ in a relation is a functional dependency between two or more non-key attributes.
17. A relation is in BCNF if and only if every determinant is a _____.
18. A relation is said to be in _____, if all possible types of dependencies that should hold on the relation can be enforced simply by enforcing the domain constraints and key constraints on the relation.
19. _____ is simply a method to analyze elements of data and their relationships and the relational model is the theoretical superstructure that supports the process.
20. _____ is the process of increasing redundancy in the database either for convenience or to improve performance.

True or False

1. Normalization serves as a tool for validating and improving the logical design, so that the logical design satisfies certain constraints and avoids unnecessary duplication of data.
2. Integrity checking (Normalization) is often performed as a series of tests on a relation to determine whether it satisfies or violates the requirements of a given normal form.
3. Normalization is a formal process of developing data structures in a manner that eliminates redundancy and promotes integrity.
4. Normalization accurately maps how humans work with data.
5. Data normalization is a corner stone of the relational theory.

NOTES

SECTION D

6. Oracle

CHAPTER 6 ORACLE

LEARNING OBJECTIVES

- Introduction
- How SQL Works
- Various Data Types of Oracle 9i
- Binary_Integer Subtypes
- Number
- Number Subtypes
- Character Types
- Char
- Varchar2
- Char Vs Varchar2
- Long and Long Raw
- Raw
- Raw(maximum_length)
- ROWID and UROWID
- NCHAR
- Boolean
- Date
- LOB Types
- BFILE
- BLOB
- CLOB
- NCLOB
- DDL , DML & DCL
- Creating a Table
- Reserved Words of SQL
- Spool
- Create Table
- Check
- Default
- Null
- Not Null
- To Add Fields Interactively
- Sequence
- Rownum
- To Delete Rows From a Table
- To Update Rows in the Table
- Like Operator
- Logical Operators
- Alter Command
- Views
- Index
- MONTHS_BETWEEN
- NEW_TIME
- NEXT_DAY
- Round

NOTES

NOTES

- Trunc
- Translate
- Upper
- CHARTOROWID
- CONVERT
- HEXTORAW
- RAWTOHEX
- ROWIDTOCHAR
- TO_DATE
- TO_NUMBER
- BFILENAME
- EMPTY_BLOB
- EMPTY_CLOB
- DUMP
- NVL
- SQLCODE
- SQLERRM
- UID
- USER
- USERENV
- VSIZE

INTRODUCTION

SQL (pronounced "ess-que-el" or "sequel") stands for Structured Query Language. SQL is used to communicate with a database. According to ANSI (American National Standards Institute), it is the standard language for relational database management systems. This language is non-procedural. SQL was introduced by IBM Corporation and standardized by ANSI and ISO (International Standards Organization) in 1986.Oracle was the first company to release a product that used SQL. SQL has structure just as English or any other language. It has rules of grammar and syntax and they are basically the normal rule of English speaking and easily understood. SQL statements are used to perform tasks such as update data on a database, or retrieve data from a database. Some common relational database management systems that use SQL are: Oracle, Sybase, Microsoft SQL Server, Access, Ingress, etc. Although most database systems use SQL, most of them also have their own additional proprietary extensions that are usually only used on their system.

The latest SQL standard was adopted in July 1999 and is often called SQL:99. The formal names of this standard are:

- ANSI X3.135-1999, "Database Language SQL", Parts 1 ("Framework"), 2 ("Foundation"), and 5 ("Bindings")
- ISO/IEC 9075:1999, "Database Language SQL", Parts 1 ("Framework"), 2 ("Foundation"), and 5 ("Bindings")

NOTES

HOW SQL WORKS

SQL is a data sublanguage. The purpose of SQL is to provide an interface to a relational database such as Oracle, and all SQL statements are instructions to the database. In this SQL differs from general-purpose programming languages like C and BASIC. Among the features of SQL are the following:

- It processes sets of data as groups rather than as individual units.
- It provides automatic navigation to the data.
- It uses statements that are complex and powerful individually, and that *therefore stand-alone*. Flow-control statements were not part of SQL originally, but they are found in the recently accepted optional part of SQL, ISO/IEC 9075-5: 1996. Flow-control statements are commonly known as "persistent stored modules" (PSM), and Oracle's PL/SQL extension to SQL is similar to PSM.

Essentially, SQL lets you work with data at the logical level. You need to be concerned with the implementation details only when you want to manipulate the data. For example, to retrieve a set of rows from a table, you define a condition used to filter the rows. All rows satisfying the condition are retrieved in a single step and can be passed as a unit to the user, to another SQL statement, or to an application. You need not deal with the rows one by one, nor do you have to worry about how they are physically stored or retrieved. All SQL statements use the **optimizer**, a part of Oracle that determines the most efficient means of accessing the specified data. Oracle also provides techniques that you can use to make the optimizer perform its job better.

VARIOUS DATA TYPES OF ORACLE 9I

SCALAR TYPES	*COMPOSITE TYPES*	*REFERENCE TYPES*	*LOB TYPES*
BINARY_INTEGER	RECORD	REF CURSOR	BFILE
DEC	TABLE	REF object_type	BLOB
DECIMAL	VARRAY		CLOB
DOUBLE PRECISION			NCLOB
FLOAT			
INT			
INTEGER			
NATURAL			
NATURAL N			
NUMBER			
NUMERIC			
REAL			
SMALLINT			
CHAR			
CHARACTER			
LONG			
NCHAR			
VARCHAR2			
RAW			
LONG RAW			
ROWID			
UROWID			
BOOLEAN (Other)			
DATE (Other)			

BINARY_INTEGER datatype is used to store signed integers. Its magnitude range is -2147483647.. 2147483647. BINARY_INTEGER values require less storage than *NUMBER* values.

NOTES

BINARY_INTEGER SUBTYPES

A base type is the datatype from which a subtype is derived. A subtype associates a base type with a constraint and so defines a subset of values.

BINARY_INTEGER subtypes:

NATURAL
NATURALN
POSITIVE
POSITIVEN
SIGNTYPE

The subtypes NATURAL and POSITIVE restrict you an integer variable to non-negative or positive values, respectively. NATURALN and POSITIVEN prevent the assigning of nulls to an integer variable. SIGNTYPE lets you restrict an integer variable to the values -1, 0, and 1, which is useful in programming tri-state logic.

NUMBER

NUMBER datatype is used to store fixed-point or floating-point numbers of virtually any size. Its magnitude range is 1E-130 .. 10E125. If the value of an expression falls outside this range, you get a numeric overflow or underflow error. You can specify precision, which is the total number of digits, and scale, which is the number of digits to the right of the decimal point. The syntax follows:

NUMBER[(precision,scale)]

To declare fixed-point numbers, for which you must specify scale, use the following form:

NUMBER(precision,scale)

To declare floating-point numbers, for which you cannot specify precision or scale because the decimal point can "float" to any position, use the following form:

NUMBER

To declare integers, which have no decimal point, use this form:

NUMBER(precision) -- same as NUMBER(precision,0)

You cannot use constants or variables to specify precision and scale; you must use integer literals. The maximum precision of a NUMBER value is 38 decimal digits. If you do not specify precision, it defaults to 38 or the maximum supported by your system, whichever is less.

Scale, which can range from -84 to 127, determines where rounding occurs. For instance, a scale of 2 rounds to the nearest hundredth (3.457 becomes 3.46). A negative scale rounds to the left of the decimal point. For example, a scale of -3 rounds to the nearest thousand (3457 becomes 3000). A scale of 0 rounds to the nearest whole number. If you do not specify scale, it defaults to 0.

NUMBER SUBTYPES

NUMBER subtypes are used for compatibility with ANSI/ISO and IBM types or when you want a more descriptive name:

DEC
DECIMAL
DOUBLE PRECISION
FLOAT

INTEGER
INT
NUMERIC
REAL
SMALLINT

Use the subtypes DEC, DECIMAL, and NUMERIC to declare fixed-point numbers with a maximum precision of 38 decimal digits.

Use the subtypes DOUBLE PRECISION and FLOAT to declare floating-point numbers with a maximum precision of 126 binary digits, which is roughly equivalent to 38 decimal digits. Or, use the subtype REAL to declare floating-point numbers with a maximum precision of 63 binary digits, which is roughly equivalent to 18 decimal digits.

NOTES

Use the subtypes INTEGER, INT, and SMALLINT to declare integers with a maximum precision of 38 decimal digits.

CHARACTER TYPES

Character types allow you to store alphanumeric data, represent words and text, and manipulate character strings.

CHAR

CHAR datatype is used to store fixed-length character data. How the data is represented internally depends on the database character set. The CHAR datatype takes an optional parameter that lets you specify a maximum length up to 32767 bytes. The syntax follows:

CHAR[(maximum_length)]

You cannot use a constant or variable to specify the maximum length; you must use an integer literal in the range 1 .. 32767.

If you do not specify a maximum length, it defaults to 1. Remember, you specify the maximum length in bytes, not characters. So, if a CHAR(n) variable stores multi-byte characters, its maximum length is less than n characters. The maximum width of a CHAR database column is 2000 bytes. So, you cannot insert CHAR values longer than 2000 bytes into a CHAR column.

You can insert any CHAR(*n*) value into a LONG database column because the maximum width of a LONG column is 2147483647 bytes or 2 gigabytes. However, you cannot retrieve a value longer than 32767 bytes from a LONG column into a CHAR(*n*) variable.

VARCHAR2

VARCHAR2 datatype is used to store variable-length character data. How the data is represented internally depends on the database character set. The VARCHAR2 datatype takes a required parameter that specifies a maximum length up to 32767 bytes. The syntax follows:

VARCHAR2(maximum_length)

You cannot use a constant or variable to specify the maximum length; you must use an integer literal in the range 1 .. 32767.

The VARCHAR2 datatype involves a trade-off between memory use and efficiency. For a VARCHAR2 (>= 2000) variable, SQL dynamically allocates only enough

memory to hold the actual value. However, for a VARCHAR2(< 2000) variable, PL/SQL preallocates enough memory to hold a maximum-size value. So, for example, if you assign the same 500-byte value to a VARCHAR2(2000) variable and to a VARCHAR2(1999) variable, the latter uses 1499 bytes more memory.

Remember, you specify the maximum length of a VARCHAR2(n) variable in bytes, not characters. So, if a VARCHAR2(n) variable stores multi-byte characters, its maximum length is less than n characters. The maximum width of a VARCHAR2 database column is 4000 bytes. Therefore, you cannot insert VARCHAR2 values longer than 4000 bytes into a VARCHAR2 column.

NOTES

You can insert any VARCHAR2(n) value into a LONG database column because the maximum width of a LONG column is 2147483647 bytes. However, you cannot retrieve a value longer than 32767 bytes from a LONG column into a VARCHAR2(n) variable.

CHAR VS VARCHAR2

When you assign a character value to a CHAR variable, if the value is shorter than the declared length of the variable, Oracle blank-pads the value to the declared length. So, information about trailing blanks is lost. In the following example, the value assigned to last_name includes six trailing blanks, not just one: last_name CHAR(10) := 'ANIL'; -- note trailing blank

If the character value is longer than the declared length of the CHAR variable, PL/SQL aborts the assignment and raises the predefined exception VALUE_ERROR Oracle neither truncates the value nor tries to trim trailing blanks. For example, given the declaration acronym CHAR(4);

the following assignment raises VALUE_ERROR:

acronym := 'ANIL '; -- note trailing blank

When you assign a character value to a VARCHAR2 variable, if the value is shorter than the declared length of the variable, PL/SQL neither blank-pads the value nor strips trailing blanks. Character values are assigned intact, so no information is lost. If the character value is longer than the declared length of the VARCHAR2 variable, PL/SQL aborts the assignment and raises VALUE_ERROR. PL/SQL neither truncates the value nor tries to trim trailing blanks.

LONG AND LONG RAW

LONG datatype is used to store variable-length character strings. The LONG datatype is like the VARCHAR2 datatype, except that the maximum length of a LONG value is 32760 bytes.

You use the LONG RAW datatype to store binary data or byte strings. LONG RAW data is like LONG data, except that LONG RAW data is not interpreted by PL/SQL. The maximum length of a LONG RAW value is 32760 bytes.

You can insert any LONG value into a LONG database column because the maximum width of a LONG column is 2147483647 bytes. However, you cannot retrieve a value longer than 32760 bytes from a LONG column into a LONG variable.

You can insert any LONG RAW value into a LONG RAW database column because the maximum width of a LONG RAW column is 2147483647 bytes. However, you cannot retrieve a value longer than 32760 bytes from a LONG RAW column into a LONG RAW variable.

LONG columns can store text, arrays of characters, or even short documents. You can reference LONG columns in UPDATE, INSERT, and (most) SELECT

statements, but not in expressions, SQL function calls, or certain SQL clauses such as WHERE, GROUP BY, and CONNECT BY.

NOTES

RAW

RAW datatype is used to store binary data or byte strings. For example, a RAW variable might store a sequence of graphics characters or a digitized picture. Raw data is like VARCHAR2 data, except that PL/SQL does not interpret raw data. Likewise, Net8 does no character set conversions when you transmit raw data from one system to another.

The RAW datatype takes a required parameter that lets you specify a maximum length up to 32767 bytes. The syntax follows:

RAW(MAXIMUM_LENGTH)

You cannot use a constant or variable to specify the maximum length; you must use an integer literal in the range 1 .. 32767.

The maximum width of a RAW database column is 2000 bytes. So, you cannot insert RAW values longer than 2000 bytes into a RAW column. You can insert any RAW value into a LONG RAW database column because the maximum width of a LONG RAW column is 2147483647 bytes. However, you cannot retrieve a value longer than 32767 bytes from a LONG RAW column into a RAW variable.

ROWID AND UROWID

Every database table has a ROWID pseudocolumn, which stores binary values called rowids. Each rowid represents the storage address of a row. A physical rowid identifies a row in an ordinary table. A logical rowid identifies a row in an index-organized table. The ROWID datatype can store only physical rowids. However, the UROWID (universal rowid) datatype can store physical, logical, or foreign (non-Oracle) rowids.

Use the ROWID datatype only for backward compatibility with old applications. For new applications, use the UROWID datatype.

When you select or fetch a rowid into a UROWID variable, you can use the built-in function ROWIDTOCHAR, which converts the binary value into an 18-byte character string. Conversely, the function CHARTOROWID converts a UROWID character string into a rowid.

NCHAR

NCHAR datatype is used to store fixed-length (blank-padded if necessary) NLS character data. How the data is represented internally depends on the national character set, which might use a fixed-width encoding such as US7ASCII or a variable-width encoding such as JA16SJIS.

The NCHAR datatype takes an optional parameter that lets you specify a maximum length up to 32767 bytes. The syntax follows: NCHAR[(maximum_length)]

You cannot use a constant or variable to specify the maximum length; you must use an integer literal in the range 1 .. 32767.

The maximum width of an NCHAR database column is 2000 bytes. So, you cannot insert NCHAR values longer than 2000 bytes into an NCHAR column. Remember,

for fixed-width, multi-byte character sets, you cannot insert NCHAR values longer than the number of characters that fit in 2000 bytes.

If the NCHAR value is shorter than the defined width of the NCHAR column, Oracle blank-pads the value to the defined width. You cannot insert CHAR values into an NCHAR column. Likewise, you cannot insert NCHAR values into a CHAR column.

Other Types

NOTES

The following types allow us to store and manipulate logical values and date/time values.

BOOLEAN

BOOLEAN datatype is used to store the logical values TRUE, FALSE, and NULL (which stands for a missing, unknown, or inapplicable value). Only logic operations are allowed on BOOLEAN variables.

The BOOLEAN datatype takes no parameters. Only the values TRUE, FALSE, and NULL can be assigned to a BOOLEAN variable. You cannot insert the values TRUE and FALSE into a database column. Also, you cannot select or fetch column values into a BOOLEAN variable.

DATE

DATE datatype is used to store fixed-length date/time values. DATE values include the time of day in seconds since midnight. The date portion defaults to the first day of the current month; the time portion defaults to midnight. The date function SYSDATE returns the current date and time.

Valid dates range from January 1, 4712 BC to December 31, 9999 AD. A Julian date is the number of days since January 1, 4712 BC. Julian dates allow continuous dating from a common reference.

LOB TYPES

The LOB (large object) datatypes BFILE, BLOB, CLOB, and NCLOB let you store blocks of unstructured data (such as text, graphic images, video clips, and sound waveforms) up to four gigabytes in size. And, they allow efficient, random, piece-wise access to the data.

The LOB types differ from the LONG and LONG RAW types in several ways. For example, LOBs can be attributes of an object type, but LONGs cannot. The maximum size of a LOB is four gigabytes, but the maximum size of a LONG is two gigabytes. Also, LOBs support random access to data, but LONG support only sequential access.

LOB types store values, called lob locators, that specify the location of large objects stored in an external file, in-line (inside the row) or out-of-line (outside the row). Database columns of type BLOB, CLOB, NCLOB, or BFILE store the locators. BLOB, CLOB, and NCLOB data is stored in the database, in or outside the row. BFILE data is stored in operating system files outside the database.

BFILE

BFILE datatype is used to store large binary objects in operating system files outside the database. Every BFILE variable stores a file locator, which points to a

large binary file on the server. The locator includes a directory alias, which specifies a full path name (logical path names are not supported).

BFILEs are read-only. You cannot modify them. The size of a BFILE is system dependent but cannot exceed four gigabytes (2**32 - 1 bytes).

BLOB

BLOB datatype is used to store large binary objects in the database in-line or out-of-line. Every BLOB variable stores a locator, which points to a large binary object. The size of a BLOB cannot exceed four gigabytes.

NOTES

CLOB

CLOB datatype is used to store large blocks of single-byte character data in the database, in-line or out-of-line. Both fixed-width and variable-width character sets are supported. Every CLOB variable stores a locator, which points to a large block of character data. The size of a CLOB cannot exceed four gigabytes.

NCLOB

NCLOB datatype is used to store large blocks of multi-byte NCHAR data in the database, in-line or out-of-line. Both fixed-width and variable-width character sets are supported. Every NCLOB variable stores a locator, which points to a large block of NCHAR data. The size of an NCLOB cannot exceed four gigabytes.

DDL , DML & DCL

DDL is Data Definition Language statements. Some examples:

- CREATE - to create objects in the database
- ALTER - alters the structure of the database
- DROP - delete objects from the database
- TRUNCATE - remove all records from a table, including all spaces allocated for the records are removed
- COMMENT - add comments to the data dictionary
- GRANT - gives user's access privileges to database
- REVOKE - withdraw access privileges given with the GRANT command

DML is Data Manipulation Language statements. Some examples:

- SELECT - retrieve data from the a database
- INSERT - insert data into a table
- UPDATE - updates existing data within a table
- DELETE - deletes all records from a table, the space for the records remain
- CALL - call a PL/SQL or Java subprogram
- EXPLAIN PLAN - explain access path to data
- LOCK TABLE - control concurrency

DCL is Data Control Language statements. Some examples:

- COMMIT - save work done
- SAVEPOINT - identify a point in a transaction to which you can later roll back
- ROLLBACK - restore database to original since the last COMMIT.
- SET TRANSACTION - Change transaction options like what rollback segment to use

NOTES

Simple Primary Key

Create a Table to store the Customer Address Record.

Create Table Customer

```
(
Cust_No Number(5)       Primary Key,
Name    Varchar2(20),
Address Varchar2(20),
City            Varchar2(20),
State           Varchar2(10),
PIN             Number(6)
);
```

In the above example Cust_No is defined as Simple Primary Key.

Composite Primary Key

Create Table Cust

```
(
order_no Number(5),
prod_no  varchar2(10),
qty             number(8),
price           number(8,2),
Primary Key (order_no, prod_no)
);
```

In the above example order_no and prod_no both are defined as Composite Primary Key.

NULL

If in a record any field that is created as nullable is not having value than Oracle will place a null value in that column. Null value is not equivalent to a blank or zero. Null value can be inserted into the columns of any data type. Null value will evaluate to null in all expressions .Example Null multiplied by 5 is null.

NOT NULL

The not null constraint ensures that the users always type the value for that column or column becomes a mandatory column. Not null constraint can be applied at column level only.

Unique

The Unique key constraint ensures that information in the column(s) is unique or value entered in unique column must not be repeated across the column(s). A table may have more than one unique key. Unique key can be null at the time of data entry. So value is not mandatory in unique column.

Note : We can define a single column as Unique and Not null both than that constraint will work like primary key.

Example:

Create table cust1

```
(
cust_no  number(5) unique not null,
```

```
name        varchar2(10)  unique
age         number(3),
Address  varchar2(20)
);
```

In the above example cust_no column is defined as both unique and not null constraints, so it will act as a primary key and name column is defined as unique. So values in column name must not be repeated.

NOTES

CHECK

Check constraint ensures that when data is entered, the data in the column is *limited to specific values. Like a, b or c.*

DEFAULT

At the time of table creation a default value can be assigned to a column. When the user is entering the values and leaves this column empty, the oracle will automatically load this column with the default value. The data type of the default value should match the data type of that column.

Example :

Create table student12

```
(
Roll_no   number(5)       Primary Key,
Enroll_no number(6)       Unique  Not  Null,
Name      varchar2(10),
Age               number(3)       Default  21,
Category  varchar2(5)     Check (category in ('SC','ST','OBC','GEN')),
Address   varchar2(10)
);
```

In the above example Roll_no is primary key and Enroll_no is defined as unique and not null so it will work like primary key. Check constraint is defined in Category column for checking the category in 'SC','ST','OBC','and GEN' at the time of data entry by the users.

To View the Structure of the table

Syntax: DESCRIBE <table name>

Example

```
SQL> DESCRIBE EMP;
OR
SQL> DESC EMP;
```

INSERTING ROWS TO THE TABLE

The INSERT INTO statement is used to insert new rows into a table.

Syntax

INSERT INTO table_name VALUES (value1, value2,....);

You can also specify the columns for which you want to insert data:

INSERT INTO table_name (column1, column2,...) VALUES (value1, value2,....);

Note :- Character and date values are typed within single quotes and numeric values are typed as they are. Date format is 'DD-MON-YEAR' as 02-APR-04'.

Example

Insert a New Row in "Persons" table where fields are given below:

LastName	FirstName	Address	City

Type this SQL statement:

NOTES

INSERT INTO Persons VALUES ('Hetland', 'Camilla', 'Hagabakka 28', 'Sandnes');

Will give this result:

LastName	FirstName	Address	City
Hetland	Camilla	Hagabakka 28	Sandnes

Insert Data in Specified Columns

This "Persons" table:

LastName	FirstName	Address	City
Hetland	Camilla	Hagabakka 28	Sandnes

Type This SQL statement:

INSERT INTO Persons (LastName, Address) VALUES ('Rasmussen', 'Storgt 68');

Will give this result:

LastName	FirstName	Address	City
Hetland	Camilla	Hagabakka 24	Sandnes
Rasmussen		Storgt 68	

TO ADD FIELDS INTERACTIVELY

INSERT INTO Persons VALUES
('&LastName','&FirstName','&Address','&City');

Note. that Names need not match the column names, the data types and order is important.

For executing the last command in SQL simply type (/) Forward slash. Than no need to type the whole command again and again. So you can enter the values frequently.

To Enter The Time portion of a date

To enter the time of date, the TO_DATE function must be used with a format mask indicating the time portion.

Syntax :- To_date (char [, format])

Example : *Insert into emp (empno,dob) values (7897,to_date('25-JUN-78 10:59 A.M.', 'DD-MON-YY HH:MI A.M.');*

SEQUENCE

A sequence is a database object that generates unique, sequential integers values in Oracle. It can be used to automatically generate primary key or unique key column. A sequence can be either in an ascending or a descending order. When you create a sequence, you can specify its initial value and an increment or decrement .

Syntax : - Create Sequence <seq_name> [increment by *n*] [start with *n*] [maxvalue *n*] [minvalue n] [cycle / nocycle] [cache/nocache];

INCREMENT BY	The interval value between sequence numbers.
START WITH	The starting value of the sequence.
MINVALUE / NOMINVALUE	The minimum sequence number.
MAXVALUE / NOMAXVALUE	The maximum sequence number.
CACHE / NOCACHE	How many values are kept in the cache - for performance reasons large databases using sequences very frequently will want a lot of values cached.
CYCLE / NOCYCLE	Specifies whether the sequence generator will cycle back to MINVALUE from MAXVALUE when MAXVALUE is met. For sequences providing key values it is probably not a good idea to cycle back.

NOTES

[ ] are optional

Increment by *n* - '*n*' is an integer that specifies the interval between sequence numbers. The default is 1. if *n* is positive, then the sequence ascends and if *n* is negative the sequence descends.

Start with *n*- Specifies the first sequence numbers to be generated.

Minvalue *n*- Specifies the minimum value of the sequence. By default, it is 1 for an ascending sequence and 10e26 – 1 for a descending sequence.

Maxvalue *n*–Specifies the maximum value that the sequence can generate. By default, it *is* -1 and 10e27 -1 for descending and ascending sequences respectively.

Cycle- Specifies that the sequence continues to generate values from the beginning after reaching either its max or min values.

No cycle - Specifies that the sequence cannot generate more values after reaching either its maxvalue or minvalue. The default value is 'no cycle'.

Cache-The cache option pre-allocates a set of sequence numbers and retains them in memory so that sequence numbers can be accessed faster. When the last of the sequence numbers in the cache has been used, Oracle reads another set of numbers into the cache.

Nocache-The default value 'nocache', does not preallocate sequence numbers for faster access.

Example :- *Create Sequence ID increment by 1*

```
start with 1
maxvalue 10
minvalue 1
cycle
cache 4;
```

After creating a sequence you can access its values with the help of currval and nextval.

```
CREATE SEQUENCE SECIDNUM INCREMENT BY 1 START WITH 1 MAXVALUE 100
NOCACHE;
```

NOTES

CURRVAL returns the current value in a specified sequence which is the value returned by the last reference to nextval.

NEXTVAL returns the initial value of the sequence, when referred to, for the first time. Later reference to nextval will increment the sequence using the increment by clause and return the new value.

Insert into emp values (Id.nextval,'John','01-Jan-99',3500,'MGR');

To obtain the current or next value in a sequence, you must use dot notation, as follows:

```
sequence_name.CURRVAL
sequence_name.NEXTVAL
```

You can use CURRVAL and NEXTVAL in a SELECT , INSERT VALUES clause, and the Update clause.

Example :- select id. currval from emp;

You may alter the INCREMENT BY, MINVALUE and MAXVALUE values of a sequence by using the ALTER command.

```
ALTER SEQUENCE SEQUENCENAME INCREMENT BY 2
ALTER SEQUENCE SEQUENCENAME MINVALUE 10
ALTER SEQUENCE SEQUENCENAME MAXVALUE 2500
```

You can remove a sequence by using the DROP command.

```
DROP SEQUENCE SEQUENCENAME
 Drop Sequence Sequence_name;
```

Syntax : Drop sequence name;

Example :- *Drop sequence id;*

```
TO VIEW THE ROWS IN THE TABLE
```

The SELECT Statement

The SELECT statement is used to view data from a table.

Syntax

SELECT column_name(s) FROM table_name ;

Select Some Columns

To select the columns named "LastName" and "FirstName", use a SELECT statement like this:

LastName	FirstName	Address	City
Hansen	Ola	Timoteivn 20	Sandnes
Svendson	Tove	Borgvn 25	Sandnes
Pettersen	Kari	Storgt 21	Stavanger

Result

LastName	FirstName
Hansen	Ola
Svendson	Tove
Pettersen	Kari

NOTES

Select All Columns

To select all columns from the "Persons" table, use a * (asterisk) symbol instead of column names, like this:

SELECT * FROM Persons;

Result

LastName	FirstName	Address	City
Hansen	Ola	Timoteivn 10	Sandnes
Svendson	Tove	Borgvn 23	Sandnes
Pettersen	Kari	Storgt 20	Stavanger

WHERE Clause

The optional WHERE condition has the general form:

WHERE boolean_expr

boolean_expr can consist of any expression which evaluates to a Boolean value. In many cases, this expression will be:

```
expr cond_op expr
```

or

```
log_op expr
```

where cond_op can be one of: =, <, <=, >, >= or <>, a conditional operator like ALL, ANY, IN, LIKE, or a locally defined operator, and log_op can be one of: AND, OR, NOT. SELECT will ignore all rows for which the WHERE condition does not return TRUE.

The SELECT Statement with where clause

The SELECT statement is used to view data from a table for a particular condition also.

Syntax

SELECT column_name(s) FROM table_name where condition ;

Multiple condition WHERE clauses

Multiple condition WHERE clauses are in the format :- WHERE keyword, 1st comparison column, 1st operator, 1st condition, AND/OR operator, 2nd comparison column, 2nd operator, 2nd condition ad infinitum. The AND operator takes precedence unless you force priority to an OR operator with brackets. An example is given below :-

SELECT * FROM BOOK WHERE SECTION_ID IN (9, 11) AND *TIMES_LENT* > 10;

The SELECT DISTINCT Statement

The DISTINCT keyword is used to return only distinct (different) values.

The SELECT statement returns information from table columns. But what if we only want to select distinct elements?

With SQL, all we need to do is to add a DISTINCT keyword to the SELECT statement:

NOTES

Syntax

SELECT DISTINCT column_name(s) FROM table_name;

Using the DISTINCT keyword

To select ALL values from the column named "Company" we use a SELECT statement like this:

SELECT Company FROM Orders ;

"Orders" table

Company	Order Number
Sega	3412
LG	2312
Trio	4678
LG	6798

Result

Company
Sega
LG
Trio
LG

Note. that "LG" is listed twice in the result-set.

To select only DIFFERENT values from the column named "Company" we use a SELECT DISTINCT statement like this:

SELECT DISTINCT Company FROM Orders;

Result:

Company
Sega
LG
Trio

ROWNUM

ROWNUM returns a number indicating the order in which a row was selected from a table. The first row selected has a ROWNUM of 1, the second row has a ROWNUM of 2, and so on. You can use ROWNUM in the WHERE clause of a SELECT statement to limit the number of rows retrieved.

TO DELETE ROWS FROM A TABLE

NOTES

The Delete Statement

The DELETE command allows you to remove rows from a table, you can include a WHERE clause in the same fashion as the SELECT statement to indicate which row(s) you want deleted - in nearly all cases you should specify a WHERE clause, running a DELETE without a WHERE clause deletes ALL rows from the table. Unlike the INSERT command the DELETE command can change multiple rows so you should take great care that you are deleting only the rows you want removed

Syntax

DELETE FROM table_name WHERE column_name = some_value;

Person:

LastName	FirstName	Address	City
Nilsen	Bond	Kirkegt 56	Stavanger
Rasmussen	John	Stien 12	Stavanger

Delete a Row

"John" is going to be deleted:

DELETE FROM Person WHERE LastName = 'John';

Result

LastName	FirstName	Address	City
Nilsen	Bond	Kirkegt 56	Stavanger

Delete All Rows

It is possible to delete all rows in a table without deleting the table. This means that the table structure, attributes, and indexes will be intact:

```
DELETE FROM table_name;
```

or

```
DELETE * FROM table_name;
```

Drop Table and Truncate

Delete a Table or Database

To delete a table (the table structure, attributes, and indexes will also be deleted):

```
DROP TABLE table_name;
```

To delete a database:

```
DROP DATABASE database_name;
```

Truncate a Table

If you only want to get rid of the data inside a table, and not the table itself.

Use the TRUNCATE TABLE command (deletes only the data inside the table):

TRUNCATE TABLE table_name;

Transaction Management

The Oracle provides a robust transaction model, as you might expect for a relational database. You determine what constitutes a transaction, the logical unit

of work that must be either saved together with a COMMIT statement or rolled back together with a ROLLBACK statement. A transaction begins implicitly with the first SQL statement issued since the last COMMIT or ROLLBACK (or with the start of a session).

PL/SQL provides the following statements for transaction management:

NOTES

COMMIT

Saves all outstanding changes since the last COMMIT or ROLLBACK and releases all locks.

ROLLBACK

Erases all outstanding changes since the last COMMIT or ROLLBACK and releases all locks.

ROLLBACK TO SAVEPOINT

Erases all changes made since the specified savepoint was established.

SAVEPOINT

Establishes a savepoint, which then allows you to perform partial ROLLBACKs.

SET TRANSACTION

Allows you to begin a read-only or read-write session, establish an isolation level, or assign the current transaction to a specified rollback segment.

LOCK TABLE

Allows you to lock an entire database table in the specified mode. This overrides the default row-level locking usually applied to a table.

Example:

```
Savepoint D1;
Delete from emp where ename = 'Scott';
Savepoint D2;
Delete from emp where ename ='King';
Rollback to D2;
Commit;
```

Will undo the second delete operation.

The SET TRANSACTION Statement

The SET TRANSACTION statement allows to begin a read-only or read-write session, establish an isolation level, or assign the current transaction to a specified rollback segment. This statement must be the first SQL statement processed in a transaction and it can appear only once. This statement comes in the following four types:

SET TRANSACTION READ ONLY;

This version defines the current transaction as read-only. In a read-only transaction, all subsequent queries only see those changes, which were committed before the transaction began (providing a read-consistent view across tables and queries). This statement is useful when you are executing long-running, multiple query reports and you want to make sure that the data used in the report is consistent:

SET TRANSACTION READ WRITE;

This version defines the current transaction as read-write:

SET TRANSACTION ISOLATION LEVEL SERIALIZABLE | READ COMMITTED;

If you specify READ COMMITTED, a DML which requires row-level locks held by another transaction will wait until those row locks are released:

SET TRANSACTION USE ROLLBACK SEGMENT rollback_segname;

This version assigns the current transaction to the specified rollback segment and establishes the transaction as read-write. This statement cannot be used in conjunction with SET TRANSACTION READ ONLY.

The LOCK TABLE Statement

This statement allows you to lock an entire database table with the specified lock mode. By doing this, you can share or deny access to that table while you perform operations against it. The syntax for this statement is:

LOCK TABLE table_reference_list IN lock_mode MODE [NOWAIT];

NOTES

where table_reference_list is a list of one or more table references (identifying either a local table/view or a remote entity through a database link), and lock_mode is the mode of the lock, which can be one of the following:

ROW SHARE
ROW EXCLUSIVE
SHARE UPDATE
SHARE
SHARE ROW EXCLUSIVE
EXCLUSIVE

If you specify the NOWAIT keyword, Oracle will not wait for the lock if the table has already been locked by another user. If you leave out the NOWAIT keyword, Oracle waits until the table is available (and there is no set limit on how long Oracle will wait). Locking a table never stops other users from querying or reading the table. The following LOCK TABLE statements show valid variations:

```
LOCK TABLE emp IN ROW EXCLUSIVE MODE;
LOCK TABLE emp, dept IN SHARE MODE NOWAIT;
LOCK TABLE scott.emp@new_york IN SHARE UPDATE MODE;
```

Now that you know the "macro" commands for managing transactions from within a PL/SQL application, let's move on to cursors; you will use cursors (in one form or another) to create transactions (*i.e.*, specify the SQL statements which make up the transaction).

TO UPDATE ROWS IN THE TABLE

The update statement is used to update or change records that match a specified criteria. This is accomplished by carefully constructing a where clause.

```
update tablename
set columnname =
newvalue
[,nextcolumn =
newvalue2...]
where columnname
OPERATOR value
[andlor column
OPERATOR value];
[] = optional
```

Examples:

```
update phone_book
set area_code = 623
where prefix = 979;
```

```
update phone_book
set last_name = 'Smith', prefix=555, suffix=9292
where last_name = 'Jons';
update employee
set age = age+1
where first_name='Pattin' and last_name='William';
```

Remember if the where clause is not given, then all the rows in the table will be updated.

NOTES

*OPERATERS in SQL *PLUS*			
Arithmetic	*Relational or Comparison*	*Logical*	*Miscellaneous*
+ (Addition) - (Subtraction) * (Multiplication) / (Divison)	= (Equal) > (Greater than) < (Less than) >= (Greater than or equal to) <= (Less than or equal to) < > or != Not equal to LIKE	AND OR NOT	IN BETWEEN NOT IN NOT LIKE || (Concatenation)

STUDENT ACTIVITY-1

1. What is a database table?

2. How do you create a table?

3. What is the syntax of the CREATE TABLE command?

4. How will you modify a table?

NOTES

LIKE OPERATOR

The LIKE is a pattern matching operator and can also be used in the conditional selection of the where clause. Like is a very powerful operator that allows you to select only rows that are "like" what you specify. The percent sign "%" (percentage) and _ (underscore) can be used as a wild card to match any possible character that might appear before or after the characters specified. The % is used to denote any number of unknown characters and _ denotes one unknown character. To denote more than one unknown character using _, the number of _ (underscore) must be equal to the number of unknown characters.

For example:

```
select empno, job, hiredate
from emp
where ename LIKE 'C%';
```

This SQL statement will match any ename that start with 'C'.

Strings must be in single quotes.

```
Or you can specify,
select empno, job, hiredate
 from emp
where ename LIKE '%K';
```

This statement will match any ename that end in a 'K'.

```
select * from emp where ename = 'SCOTT';
```

This will only select rows where the ename equals 'SCOTT' exactly.

To retrieve rows where the JOB contains the word 'NA' embedded in it.

Example : MANAGER, ANALYST.

```
SELECT * FROM EMP WHERE JOB LIKE '%NA%';
```

To retrieve rows where the ENAME starts with 'C' and end with 'K'.

```
SELECT * FROM EMP WHERE ENAME LIKE 'C%K';
```

To retrieve rows where the ENAME starts with 'C' and having four characters more after 'C'.

Example - CLERK, CLARK,CROWN,CAMIL

```
SELECT * FROM EMP WHERE ENAME LIKE 'C____';
```

LOGICAL OPERATORS

AND ensures that rows satisfying both the conditions are selected.

OR ensures that rows satisfying any one of the condition is retrieved.

NOT ensures that rows not satisfying the condition specified in the query.

IN - one of a set of values

Syntax

```
IN ( { Value [,...] } | SELECT-Command )
```

Description:

IN (Value [,...]) tells the DBMS , that one of the values must be in the specified value list.

NOTES

Example:

SELECT *

FROM order

WHERE price IN (100,200,300,400,500);

Here are all records from the table order selected, where price have one of the values in the specified list.

select * from emp where empno IN (7782,7788,7876);

Here are all records from the table emp selected, where empno have one of the values in the specified list.

SELECT * FROM EMP WHERE JOB IN ('CLERK','MANAGER');

Here are all records from the table emp selected, where JOB have one of the values in the specified list.

NOT IN

NOT IN ({ Value [,...] } | SELECT-Command)

Description:

NOT IN (Value [,...]) tells the DBMS , that values must not be in the specified value list.

SELECT * FROM EMP WHERE JOB NOT IN ('CLERK','MANAGER');

Here are all records from the table emp selected, where JOB not have values in the specified list.

BETWEEN

BETWEEN - checks when an field is between two values

Syntax

BETWEEN Value1 AND Value2

Description:

BETWEEN W_1 AND W_2 tells the Oracle, that values must be between W_1 and W_2.

Example:

SELECT num, item, price

FROM order

WHERE price BETWEEN 100 AND 500

Here are the fields num, *item and price from the* table order selected, where the field price have a value between 100 and 500.

SELECT * FROM EMP WHERE SAL BETWEEN 1500 AND 2450;

Here are the fields from EMP table selected, where the SAL value between 1500 and 2450.

You can also use **NOT BETWEEN** to exclude the values between your range.

NOT LIKE

The **NOT LIKE** is a pattern matching operator and can also be used in the conditional selection of the where clause. NOT Like operator that allows you to select only rows that are "not like" what you specify. The percent sign "%" (percentage) and _ (underscore) can be used as a wild card to match any possible character that might appear before or after the characters specified.

For example:

Select empno, job, hiredate from emp where ename NOT LIKE 'C%';

Above query will select only those rows where ename not starts with 'C' character.

ORDER BY

ORDER BY is clause will allow you to display the results of your query in a sorted order (either ascending order or descending order) based on the columns that you specify to order by.

ORDER BY clause syntax:

```
SELECT column1, SUM(column2)
FROM "list-of-tables"
ORDER BY
"column-list" ASC | DESC;
```

NOTES

This statement will select the employee_id, dept, name, age, and salary from the employee table where the results in Ascending (default) order based on their Salary.

```
ASC = Ascending Order - default
DESC = Descending Order
```

For example:

```
SELECT employee_id, dept, name, age, salary
FROM employee ORDER BY salary;
```

If you would like to order based on multiple columns, you must seperate the columns with commas. *For example:*

Select fields from tablename ORDER BY field name desc;

```
SELECT employee_id, dept, name, age, salary
FROM employee ORDER BY salary, age DESC;
```

In the above query Salary column will be arranged in ascending order and age column will be arranged in descending order.

To Rename a Table

Syntax: Rename oldtablename to newtablename;

Example: -

```
Rename emp to newemp;
```

Above query will change the table name emp to newemp.

Concatenation

There is just one string operator - | | (split bar typed twice), for string concatenation. Both operands of | | must be strings. The operator concatenates the second string to the end of the first.

For example,

'ab' | | 'cd' => 'abcd'

Example:

SELECT ENAME | |' (' | | JOB | | ') ' FROM EMP;

SELECT ENAME | | ' WORKING AS ' | | JOB | | ' IN DEPARTMENT NO ' | | DEPTNO | |' HAVING SALARY '| | SAL FROM EMP;

COLUMN ALIASES

The column of the table can be displayed with user defined heading or aliases instead of the column names appearing as the heading.

Syntax: - Select column1, column2 "Alias" from tablename;

Example :- SELECT EMPNO "NUMBER", ENAME "NAME" FROM EMP;

NOTES

COLUMN FORMAT

The column data can be formatted as per the requirement of the user.

Example

COLUMN SAL FORMAT 9,99,999.99 ;

SELECT * FROM EMP;

The SAL column will now be displayed with commas wherever applicable and two decimal places.

COLUMN JOB FORMAT A5 TRUNC;

SELECT * FROM EMP;

The JOB column is formatted to display only 5 characters. If a Job name is longer than 5 characters, it will be truncated in the display.

ALTER COMMAND

The alter command is used to modify the structure of a table and Update command is used to modify the records or field of a table. So using alter command you can add new column, modify existing columns, add or drop integrity constraints.

ALTER can be used to make the following changes to any tables:-

- Add new columns
- Add new integrity constraints
- Modify existing columns
  - Expand length
  - Change default
  - Decrease length - all values in column must be null
  - Change data type - all values in column must be null
- Modify existing columns
- Modify existing columns
- Drop integrity constraints

You cannot drop or rename columns in earlier versions. The table must be rebuilt for this. But in Oracle 9i and higher versions you can drop a column.

You must have one of the following privileges to alter a table:

- Ownership of the table.
- Alter privilege on the table.
- Alter any table (Allows altering of tables outside your schema).

To add a new column

Syntax : Alter table tablename Add(field datatype(width));

Example:

Alter table emp add (Address varchar2 (20));

To Change the width of existing field

Syntax : Alter table table modify(fieldname datatype(width))

Example:

Alter table emp modify(Address varchar2 (25));

Many column can be added in a single alter statement

Syntax : Alter table tablename add (fieldname datatype(size), fieldname datatype(size), fieldname datatype(size) default 'ABC', fieldname datatype(size));

Example:

Alter table emp add (Address varchar2 (20), City varchar2(8),D_O_B date);

To add a primary key

Syntax : Alter table tablename add primary key(field);

Example:

Alter table emp add primary key(empno);

To remove primary key

Syntax :Alter table tablename drop primary key

Example:

Alter table emp drop primary key;

To drop primary key that has dependents tables:

Syntax: Alter table tablename drop primary key cascade;

Example:

Alter table emp drop primary key cascade;

To add a foreign key constraint to an existing column

Syntax : Alter table tablename

Add (foreign key (column) references master_table on delete cascade);

Example:

Alter table emp add (foreign key (sup_code) references supplier_master on delete cascade);

To Drop a foreign key constraint

Syntax : Alter table tablename drop constraint column name;

Example:

Alter table emp drop constraint Sup_code;

To add a not null constraint

Syntax : Alter table tablename modify column name NOT NULL;

Example:

Alter table emp modify ename not null;

To Drop a NOT NULL constraint

Syntax: Alter table tablename modify column name NULL;

Example: Alter table emp modify ename null;

To add a check constraints

Example:

Alter table emp add constraint chkdeposit (check (deposit between 2000 and 15000);

To drop the check constraints

Example Alter table emp drop constraint chkdeposit;

To add a primary key constraint and name the constraint as MID

Example: Alter table emp add constraint MID primary key (ID);

To drop the primary key constraint

NOTES

Example :- Alter Table EMP drop constraint MID;

To add a Foreign key constraint to an existing column

Example: - Alter table emp add constraint supplier foreign key (su_code) references Supplier_Master on delete cascade;

To drop the foreign key constraint

Example:- Alter table emp drop constraint supplier;

To drop the column

NOTES

Example:- Alter table emp drop column ename;

Synonym

A synonym is a database object, which is used as an alias (alternative names) for a table, view or sequence. They are used to

- Simplify SQL statements.
- Hide the name and owner of an object.
- Provide location transparency for remote objects of a distributed database.
- Provide public access to an object.

Synonym can either be private or public. The former is created by normal user, which is available only to that person whereas the latter is created by the DBA, which can be availed by any database user.

Public synonyms are created by a Database Administrator to hide the identity of a base table and reduce the complexity of SQL statements.

The syntax for creating a synonym is given below.

```
Create [public] synonym <synonym_name> for <table_name>;
```

One such example of a public synonym is T AB, which we use for selecting the tables owned by the user. These public synonyms are owned by user group Public.

Example: -

SQL > Create table tab (no number (5));

You will get the message table created.

Now select the values from tab, no rows will be selected. When we have public and local objects with the same name, the local objects takes precedence. But we can use the public synonym as usual after dropping the local objects.

```
SQL > Drop table tab;
SQL> Select * from tab;
```

VIEWS

Views are logical tables of data extracted from existing tables. A view can be thought of as a 'stored query' or a "VIRTUAL TABLE". We can use views in most places where a table can be used. The tables upon which a view is based are called base tables. It can be queries just like a table, but does not require disk space.

A view is a method of organising table data to meet a specific need. Views are based on SELECT statements, which derive their data from real tables. A view allows you to reorganise the database data, you might want to do this so that you can restrict data access, reduce selection complexity, provide improved data independence or allow disparate users to view the same data in different ways. Most of the Oracle data dictionary is readable via views which interpret internal

Oracle tables. Views come in simple and complex forms. Simple views are based on a single table and data can be updated via DML commands (privilege issues aside), complex views are derived from multiple tables and DML commands cannot be used to update data.

Use of View

NOTES

- It can be used to hide sensitive columns.
- It can be used to hide complex queries involving multiple tables. For example, a single view might be defined with a join, which is a collection of related columns or rows in multiple tables. However, the view hides the fact that this information actually originates from several tables.
- Views created with a check option, prevents the updating of other rows and columns.
- Views provide an additional level of table security by restricting to a predetermined set of rows and /or columns of a table.
- Views simplify commands for the user because they allow them to select information from multiple tables without actually knowing how to perform a join.
- Views isolate applications from changes in definitions of base tables. For example, if a view's defining query references three columns of a four-column table and a fifth column is added to the table, the view's definition is not affected and all applications using the view are not affected.
- Views provide data in a different perspective than that of a base table by renaming columns without affecting the base table.

Syntax - Create [or Replace][no][force] View <view-name> [column alias name..] as <query> [with [check option][read only][constraint]];

Example *Create view emp_view as select emp_no, emp_name from emp;*

The Order By clause cannot be used in a create view statement.

To Display the view

Select * from emp_view;

To View the columns in a view

Desc emp_view

To Delete a view Drop view emp_view; VIEW cannot be updated if it contains.

(*i*) Joins
(*ii*) Set operators (Union, Intersect, Minus)
(*iii*) Group functions
(*iv*) Group by clause
(*v*) Distinct

Example: -

```
SQL> create view ven_view as select * from vendor_master;
```

The above command will create a view with name ven_view , which will have the same structure as the vendor_master table and all the rows of the base table are accessible through this view.

INDEX

Indexes are optional structures associated with tables. We can create indexes explicitly to speed up SQL statement execution on a table. Similar to the indexes

in books that helps us to locate information faster, an Oracle index provides a faster access path to table data.

The index points directly to the location of the rows containing the value. Indexes are the primary means reducing disk I/O when properly using appropriately. The absence or presence of an index does not require a change in the wording of any SQL statement. An Index is a fast access path to the data; it affects only the speed of execution.

NOTES

We create an index on a column or combinations of column using CREATE INDEX command as follows.

Assuming that the order_detail table has more tham 10000 orders and the access on the table is time consuming., So the marketing division wants to speed up the access by creating index on the frequently queried column the itemcode.

Example

```
SQL> create index oditem on order_detail(itemcode);
```

When we create an index, Oracle fetches and sorts the column to be indexed, and the stores the ROWID along with the index value for each row. Then Oracle loads the index from the bottom up. Oracle sorts the order_detail table on the itemcode column. It then loads the index with the itemcode and corresponding ROWID values in this sorted order. Using the index, Oracle does a quick search through the sorted itemcode values and then uses the associated ROWID values to locate the rows having the sought item code value.

Indexes are logically and physically independent of the data in associated table. We can create or drop an index at any time with effecting the base tables or order indexes. If we drop an index, all applications continue to work; however, access to previously indexed data might be slower. Indexes, as independent structures, requires storage space.

SQL *PLUS FUNCTIONS

SQL *Plus provides some special inbuilt functions to perform operations using the DML commands. A function takes one or more arguments and returns a value. There are mainly two types of functions.

1. Single row functions (Scalar Functions)
2. Group Functions (Aggregate Functions)

Single Row Functions

A single row or scalar function returns only one value for every row queried in the table. Single row functions can appear in a select commands and can also be included in a 'where' clause. The single row functions can be classified as:

- Date functions
- Numeric Functions
- Character Functions
- Conversion functions
- LOB and Miscellaneous functions

'DUAL' Table

Dual is a small Oracle worktable, which consist of only one row and one column, and contain the value x in that column. It supports date retrieval and its formatting and arithmetic calculations etc. You can use dual table with select statements, where clause and for retrieving the function results.

Example: -

```
SQL> Select 2*2 from dual;
SQL> Select sysdate from dual;
```

Table 1. DATE FUNCTIONS

Name	*Description*
ADD_MONTHS	Adds the specified number of months to a date.
LAST_DAY	Returns the last day in the month of the specified date.
MONTHS_ BETWEEN	Calculates the number of months between two dates.
NEW_TIME	Returns the date/time value, with the time shifted as requested by the specified time zones.
NEXT_DAY	Returns the date of the first weekday specified that is later than the date.
ROUND	Returns the date rounded by the specified format unit.
SYSDATE	Returns the current date and time in the Oracle Server.
TRUNC	Truncates the specified date of its time portion according to the format unit provided.

NOTES

SYSDATE

The SYSDATE function returns the current system date and time as recorded in the database. The time component of SYSDATE provides the current time to the nearest second. It takes no arguments. The specification for SYSDATE is:

FUNCTION SYSDATE RETURN DATE

SQL > SELECT SYSDATE FROM dual;

ADD_MONTHS

The ADD_MONTHS function returns a new date with the specified number of months added to the input date. The specification for ADD_MONTHS is as follows:

```
FUNCTION ADD_MONTHS (date_in IN DATE, month_shift NUMBER) RETURN DATE
FUNCTION ADD_MONTHS (month_shift NUMBER, date_in IN DATE) RETURN DATE
```

ADD_MONTHS is an overloaded function. You can specify the date and the number of months by which you want to shift that date, or you can list the month_shift parameter first and then the date. Both arguments are required.

Date Arithmetic

SQL allows you to perform arithmetic operations directly on date variables. You may add numbers to a date or subtract numbers from a date. To move a date one day in the future, simply add 1 to the date as shown below:

join_date + 1

You can even add a fractional value to a date. For example, adding 1/24 to a date adds an hour to the time component of that value. Adding 1/(24*60) adds a single minute to the time component, and so on.

NOTES

If the month_shift parameter is positive, ADD_MONTHS returns a date for that number of months into the future. If the number is negative, ADD_MONTHS returns a date for that number of months in the past. Here are some examples that use ADD_MONTHS:

Move ahead date by three months:

```
SQL > SELECT ADD_MONTHS ('12-JAN-1995', 3) FROM DUAL;
SQL > 12-APR-1995
```

Specify negative number of months in first position:

```
SQL > SELECT ADD_MONTHS (-12, '12-MAR-1990') FROM DUAL;
SQL > 12-MAR-1989
```

ADD_MONTHS always shifts the date by whole months. You can provide a fractional value for the month_shift parameter, but *ADD_MONTHS will always* round down to the whole number nearest zero, as shown in these examples:

```
SQL > SELECT ADD_MONTHS ('28-FEB-1989', 1.5) FROM DUAL;
same as
SQL > SELECT ADD_MONTHS ('28-FEB-1989', 1) FROM DUAL;
SQL > 31-MAR-1989
SQL > SELECT ADD_MONTHS ('28-FEB-1989', 1.9999)FROM DUAL;
same as
SQL > SELECT ADD_MONTHS ('28-FEB-1989', 1)FROM DUAL;
SQL > 31-MAR-1989
SQL > SELECT ADD_MONTHS ('28-FEB-1989', -1.9999)FROM DUAL;
same as
SQL > SELECT ADD_MONTHS ('28-FEB-1989', -1)FROM DUAL;
SQL > 31-JAN-1989
SQL > SELECT ADD_MONTHS ('28-FEB-1989', .5)FROM DUAL;
same as
SQL > SELECT ADD_MONTHS ('28-FEB-1989', 0)FROM DUAL;
SQL > 28-FEB-1989
```

If you want to shift a date by a fraction of a month, simply add to or subtract from the date the required number of days. SQL supports direct arithmetic operations between date values.

If the input date to ADD_MONTHS does not fall on the last day of the month, the date returned by ADD_MONTHS falls on the same day in the new month as in the original month. If the day number of the input date is greater than the last day of the month returned by ADD_MONTHS, the function sets the day number to the last day in the new month. For example, there is no 31st day in February, so ADD_MONTHS returns the last day in the month.

```
SQL >SELECT ADD_MONTHS ('31-JAN-1895', 1)FROM DUAL;
SQL > 28-FEB-1995
```

LAST_DAY

The LAST_DAY function returns the date of the last day of the month for a given date. The specification is:

FUNCTION LAST_DAY (date_in IN DATE) RETURN DATE

This function is useful because the number of days in a month varies throughout the year. With LAST_DAY, for example, you do not have to try to figure out if

February of this or that year has 28 or 29 days. Just let LAST_DAY figure it out for you.

Here are some examples of LAST_DAY:

Go to the last day in the month:

```
SQL >SELECT LAST_DAY ('16-JAN-99') FROM DUAL;
SQL >31-JAN-1999
```

If already on the last day, just stay on that day:

```
SQL >SELECT LAST_DAY ('31-JAN-99')FROM DUAL;
SQL > 31-JAN-1999
```

Get the last day of the month three months after being hired:

```
SQL > SELECT LAST_DAY (ADD_MONTHS (hiredate, 3)) FROM EMP;
```

Tell me the number of days until the end of the month:

```
SQL > SELECT LAST_DAY (SYSDATE)FROM DUAL;
SQL > (Last day of the current month)
```

NOTES

MONTHS_BETWEEN

The MONTHS_BETWEEN function calculates the number of months between two dates and returns that difference as a number. The specification is:

```
FUNCTION MONTHS_BETWEEN (date1 IN DATE, date2 IN DATE)
RETURN NUMBER
```

The following rules apply to MONTHS_BETWEEN:

- If date1 comes after date2, then MONTHS_BETWEEN returns a positive number.
- If date1 comes before date2, then MONTHS_BETWEEN returns a negative number.
- If date1 and date2 are in the same month, then MONTHS_BETWEEN returns a fraction (a value between -1 and +1).
- If date1 and date2 both fall on the last day of their respective months, then MONTHS_BETWEEN returns a whole number (no fractional component).
- If date1 and date2 are in different months and at least one of the dates is not a last day in the month, MONTHS_BETWEEN returns a fractional number. The fractional component is calculated on a 31-day month basis and also takes into account any differences in the time component of date1 and date2.

Here are some examples of the uses of MONTHS_BETWEEN:

Calculate two ends of month, the first earlier than the second:

```
SQL > SELECT MONTHS_BETWEEN ('31-JAN-1994', '28-FEB-1994')
FROM DUAL;
SQL > -1
```

Calculate two ends of month, the first later than the second:

```
SQL > SELECT MONTHS_BETWEEN ('31-MAR-1995', '28-FEB-1994')
 FROM DUAL;
SQL > 13
```

Calculate when both dates fall in the same month:

```
SQL > SELECT MONTHS_BETWEEN ('28-FEB-1994', '15-FEB-1994')
 FROM DUAL;
```

```
SQL > 0
```

Perform months_between calculations with a fractional component:

```
SQL >SELECT MONTHS_BETWEEN ('31-JAN-1994', '1-MAR-1994')
FROM DUAL;
SQL >-1.0322581
SQL >SELECT MONTHS_BETWEEN ('31-JAN-1994', '2-MAR-1994')
FROM DUAL;
SQL > -1.0645161
SQL >SELECT MONTHS_BETWEEN ('31-JAN-1994', '10-MAR-1994')
FROM DUAL;
SQL > -1.3225806
```

NOTES

If you detect a pattern here you are right. As we said, MONTHS_BETWEEN calculates the fractional component of the number of months by assuming that each month has 31 days. Therefore, each additional day over a complete month counts for 1/31 of a month, and:

1 divided by 31 = .032258065 --more or less!

According to this rule, the number of months between January 31, 1994 and February 28, 1994 is one -- a nice, clean integer. But to calculate the number of months between January 31, 1994 and March 1, 1994, you have to add an additional .032258065 to the difference (and make that additional number negative because in this case MONTHS_BETWEEN counts from the first date back to the second date.

NEW_TIME

This function converts dates (along with their time components) from one time zone to another. The specification for NEW_TIME is:

FUNCTION NEW_TIME (date_in DATE, zone1 VARCHAR2, zone2 VARCHAR2)

RETURN DATE

where date_in is the original date, zone1 is the starting point for the zone switch (usually, but not restricted to, your own local time zone), and zone2 is the time zone in which the date returned by NEW_TIME should be placed.

The valid time zones are shown in Table 2

The specification of time zones to NEW_TIME is not case-sensitive, as the following example shows:

```
SQL > SELECT TO_CHAR (NEW_TIME (TO_DATE ('09151994 12:30 AM', 'MMDDYYYY
HH:MI AM'),'CST', 'hdt'),'Month DD, YYYY HH:MI AM')FROM DUAL;
SQL >'September 14, 1994 09:30 PM'
```

So, when it was 12:30 in the morning of September 15, 1994 in Chicago, it was 9:30 in the evening of September 14, 1994 in Anchorage.

Note: We used TO_DATE with a format mask to make sure that a time other than the default of midnight would be used in the calculation of the new date and time. We then used TO_CHAR with another date mask (this one intended to make the output more readable) to display the date and time, because by default SQL will not include the time component unless specifically requested to do so.

Table 2. Time Zone Abbreviations and Descriptions

AST	Atlantic Standard Time
ADT	Atlantic Daylight Time
BST	Bering Standard Time
BDT	Bering Daylight Time
CST	Central Standard Time
CDT	Central Daylight Time
EST	Eastern Standard Time
EDT	Eastern Daylight Time
GMT	Greenwich Mean Time
HST	Alaska-Hawaii Standard Time
HDT	Alaska-Hawaii Daylight Time
MST	Mountain Standard Time
MDT	Mountain Daylight Time
NST	Newfoundland Standard Time
PST	Pacific Standard Time
PDT	Pacific Daylight Time
YST	Yukon Standard Time
YDT	Yukon Daylight Time

NOTES

STUDENT ACTIVITY-2

1. How we can delete tables?

2. Define index and views.

3. What are the advantages of views?

4. How are views different from base tables?

NOTES

NEXT_DAY

The NEXT_DAY function returns the date of the first day after the specified date which falls on the specified day of the week. Here is the specification for NEXT_DAY:

FUNCTION NEXT_DAY (date_in IN DATE, day_name IN VARCHAR2) RETURN DATE

The day_name must be a day of the week in your session's date language (specified by the NLS_DATE_LANGUAGE database initialization parameter). The time component of the returned date is the same as that of the input date, date_in. If the day of the week of the input date matches the specified day_name, then NEXT_DAY will return the date seven days (one full week) after date_in. NEXT_DAY does not return the input date if the day names match.

Here are some examples of the use of NEXT_DAY.

Let's figure out the date of the first Monday and Wednesday in 1997 in all of these examples.

You can use both full and abbreviated day names:

```
SQL > SELECT NEXT_DAY ('01-JAN-1997', 'MONDAY') FROM DUAL;
SQL > 06-JAN-1997
SQL > SELECT NEXT_DAY ('01-JAN-1997', 'MON') FROM DUAL;
SQL > 06-JAN-1997
```

The case of the day name doesn't matter :

```
SQL >SELECT NEXT_DAY ('01-JAN-1997', 'monday') FROM DUAL;
SQL >06-JAN-1997
```

If the date language were Spanish:

```
SQL >SELECT NEXT_DAY ('01-JAN-1997', 'LUNES') FROM DUAL;
SQL >06-JAN-1997
NEXT_DAY of Wednesday moves the date up a full week:
SQL >SELECT NEXT_DAY ('01-JAN-1997', 'WEDNESDAY') FROM DUAL;
SQL >08-JAN-1997
```

ROUND

The ROUND function rounds a date value to the nearest date as specified by a format mask. It is just like the standard numeric ROUND function, which rounds a number to the nearest number of specified precision, except that it works with dates. The specification for ROUND is as follows:

FUNCTION ROUND (date_in IN DATE [, format_mask VARCHAR2]) RETURN DATE

The ROUND function always rounds the time component of a date to midnight (12:00 A.M.). The format mask is optional. If you do not include a format mask, ROUND rounds the date to the nearest day. In other words, it checks the time component of the date. If the time is past noon, then ROUND returns the next day with a time component of midnight.

The set of format masks for ROUND is a bit different from those masks used by TO_CHAR and TO_DATE. The masks are listed in Table 4.2. These same formats are used by the TRUNC function, described later in this chapter, to perform truncation on dates.

NOTES

Table 3. Format Masks for ROUND and TRUNC

Format Mask	*Rounds or Truncates to*
CC or SSC	Century
SYYY, YYYY, YEAR, SYEAR, YYY, YY, or Y	Year (rounds up to next year on July 1)
IYYY, IYY, IY, or I	Standard ISO year
Q	Quarter (rounds up on the sixteenth day of the second month of the quarter)
MONTH, MON, MM, or RM	Month (rounds up on the sixteenth day, which is not necessarily the same as the middle of the month)
WW	Same day of the week as the first day of the year
IW	Same day of the week as the first day of the ISO year
W	Same day of the week as the first day of the month
DDD, DD, or J	Day
DAY, DY, or D	Starting day of the week
HH, HH12, HH24	Hour
MI	Minute

Here are some examples of ROUND dates:

Round up to the next century:

```
SQL > SELECT TO_CHAR (ROUND (TO_DATE ('01-MAR-1994'), 'CC'), 'DD-MON-
YYYY') FROM DUAL;
SQL > 01-JAN-2000
```

Round back to the beginning of the current century:

```
SQL >SELECT TO_CHAR (ROUND (TO_DATE ('01-MAR-1945'), 'CC'), 'DD-MON-
YYYY')FROM DUAL;
SQL >01-JAN-1900
```

Round down and up to the first of the year:

```
SQL > SELECT ROUND (TO_DATE ('01-MAR-1994'), 'YYYY')FROM DUAL;
SQL > 01-JAN-1994
SQL > SELECT ROUND (TO_DATE ('01-SEP-1994'), 'YEAR')FROM DUAL;
SQL > 01-JAN-1995
```

Round up and down to the quarter (first date in the quarter):

```
SQL >SELECT ROUND (TO_DATE ('01-MAR-1994'), 'Q')FROM DUAL;
SQL >01-APR-1994
SQL >SELECT ROUND (TO_DATE ('15-APR-1994'), 'Q')FROM DUAL;
SQL >01-APR-1994
```

Round down and up to the first of the month:

```
SQL > SELECT ROUND (TO_DATE ('12-MAR-1994'), 'MONTH')FROM DUAL;
SQL > 01-MAR-1994
SQL > SELECT ROUND (TO_DATE ('17-MAR-1994'), 'MM') FROM DUAL;
SQL > 01-APR-1994
Day of first of year is Saturday:
SQL > SELECT TO_CHAR (TO_DATE ('01-JAN-1994'), 'DAY')FROM DUAL;
SQL > 'SATURDAY'
```

NOTES

So round to date of nearest Saturday for `01-MAR-1994':

```
SQL > SELECT ROUND (TO_DATE ('01-MAR-1994'), 'WW')FROM DUAL;
SQL > 26-FEB-1994
```

First day in the month is a Friday:

```
SQL >SELECT TO_CHAR (TO_DATE ('01-APR-1994'), 'DAY')FROM DUAL;
SQL > FRIDAY
```

So round to date of nearest Friday from April 16, 1994:

```
SQL >SELECT TO_CHAR ('16-APR-1994'), 'DAY')FROM DUAL;
SQL >SATURDAY
SQL >SELECT ROUND (TO_DATE ('16-APR-1994'), 'W')FROM DUAL;
SQL > 15-APR-1994
SQL >SELECT TO_CHAR (ROUND (TO_DATE ('16-APR-1994'), 'W'), 'DAY')FROM
DUAL;
SQL >FRIDAY
```

In the rest of the examples I use TO_DATE in order to pass a time component to the ROUND function, and TO_CHAR to display the new time.

Round back to nearest day (time always midnight):

```
SQL > SELECT TO_CHAR (ROUND (TO_DATE ('11-SEP-1994 10:00 AM', 'DD-MON-
YY HH:MI AM'), 'DD'),'DD-MON-YY HH:MI AM')FROM DUAL;
SQL > 11-SEP-1994 12:00 AM
```

Round forward to the nearest day:

```
SQL >SELECT TO_CHAR (ROUND (TO_DATE ('11-SEP-1994 4:00 PM', 'DD-MON-YY
HH:MI AM'), 'DD'),'DD-MON-YY HH:MI AM')FROM DUAL;
SQL > 12-SEP-1994 12:00 AM
```

Round back to the nearest hour:

```
SQL >SELECT TO_CHAR (ROUND (TO_DATE ('11-SEP-1994 4:17 PM', 'DD-MON-YY
HH:MI AM'), 'HH'),'DD-MON-YY HH:MI AM')FROM DUAL;
SQL > 11-SEP-1994 04:00 PM
```

TRUNC

The TRUNC function truncates date values according to the specified format mask. The specification for *TRUNC* is:

```
FUNCTION TRUNC (date_in IN DATE [, format_mask VARCHAR2]) RETURN DATE
```

The TRUNC date function is similar to the numeric FLOOR function . Generally speaking, it rounds down to the beginning of the minute, hour, day, month, quarter, year, or century, as specified by the format mask.

TRUNC offers the easiest way to retrieve the first day of the month or first day of the year. It is also useful when you want to ignore the time component of dates. This is often the case when you perform comparisons with dates, such as the following:

```
IF request_date BETWEEN start_date AND end_date
THEN
  ...
```

The date component of date_entered and start_date might be the same, but if your application does not specify a time component for each of its dates, the comparison might fail. If, for example, the user enters a request_date and the screen does not include a time component, the time for request_date will be midnight or 12:00 A.M. of that day. If start_date was set from SYSDATE, however, its time component will reflect the time at which the assignment was made. Because 12:00 A.M. comes before any other time of the day, a comparison that looks to the naked eye like a match might well fail.

If you are not sure about the time components of your date fields and variables and want to make sure that your operations on dates disregard the time component, TRUNCate them:

```
IF TRUNC (request_date) BETWEEN TRUNC (start_date) AND TRUNC (end_date)
THEN
...
```

TRUNC levels the playing field with regard to the time component: all dates now have the same time of midnight (12:00 A.M.). The time will never be a reason for a comparison to fail.

Here are some examples of TRUNC for dates (all assuming a default date format mask of DD-MON-YYYY):

NOTES

Without a format mask, TRUNC sets the time to 12:00 A.M. of the same day:

```
SQL > SELECT TO_CHAR (TRUNC (TO_DATE ('11-SEP-1994 9:36 AM', 'DD-MON-
YYYY HH:MI AM'))FROM DUAL;
SQL > 11-SEP-1994 12:00 AM
```

Trunc to the beginning of the century in all cases:

```
SQL > SELECT TO_CHAR (TRUNC (TO_DATE ('01-MAR-1994'), 'CC'), 'DD-MON-
YYYY')FROM DUAL;
SQL > 01-JAN-1900
SQL > SELECT TO_CHAR (TRUNC (TO_DATE ('01-MAR-1945'), 'CC'), 'DD-MON-
YYYY')FROM DUAL;
SQL > 01-JAN-1900
```

Trunc to the first of the current year:

```
SQL >SELECT TRUNC (TO_DATE ('01-MAR-1994'), 'YYYY')FROM DUAL;
SQL > 01-JAN-1994
SQL >SELECT TRUNC (TO_DATE ('01-SEP-1994'), 'YEAR')FROM DUAL;
SQL >01-JAN-1994
```

Trunc to the first day of the quarter:

```
SQL >SELECT TRUNC (TO_DATE ('01-MAR-1994'), 'Q')FROM DUAL;
SQL > 01-JAN-1994
SQL >SELECT TRUNC (TO_DATE ('15-APR-1994'), 'Q')FROM DUAL;
SQL >01-APR-1994
```

Trunc to the first of the month:

```
SQL >SELECT TRUNC (TO_DATE ('12-MAR-1994'), 'MONTH')FROM DUAL;
SQL >01-MAR-1994
SQL >SELECT TRUNC (TO_DATE ('17-MAR-1994'), 'MM')FROM DUAL;
SQL >01-APR-1994
```

In the rest of the examples we use TO_DATE to pass a time component to the TRUNC function, and TO_CHAR to display the new time:

Trunc back to the beginning of the current day (time is always midnight):

```
SQL > SELECT TO_CHAR (TRUNC (TO_DATE ('11-SEP-1994 10:00 AM',
'DD-MON-YYYY HH:MI AM'), 'DD'),'DD-MON-YYYY HH:MI AM')FROM DUAL;
SQL > 11-SEP-1994 12:00 AM
SQL > SELECT TO_CHAR (TRUNC (TO_DATE ('11-SEP-1994 4:00 PM',
'DD-MON-YYYY HH:MI AM'), 'DD'),'DD-MON-YYYY HH:MI AM')FROM DUAL;
SQL > 11-SEP-1994 12:00 AM
```

Trunc to the beginning of the current hour:

```
SQL > SELECT TO_CHAR (TRUNC (TO_DATE ('11-SEP-1994 4:17 PM',
'DD-MON-YYYY HH:MI AM'), 'HH'),'DD-MON-YYYY HH:MI AM')FROM DUAL;
SQL > 11-SEP-1994 04:00 PM
```

NOTES

Table 4. NUMERIC FUNCTIONS

Name	*Description*
ABS	Returns the absolute value of the number.
ACOS	Returns the inverse cosine
ASIN	Returns the inverse sine.
ATAN	Returns the inverse tangent.
ATAN2	Returns the result of the tan2 inverse trigonometric function.
CEIL	Returns the smallest integer greater than or equal to the specified number.
COS	Returns the cosine.
COSH	Returns the hyperbolic cosine.
EXP (n)	Returns e raised to the nth power, where e = 2.71828183...
FLOOR	Returns the largest integer equal to or less than the specified number.
LN (a)	Returns the natural logarithm of a.
LOG (a, b)	Returns the logarithm, base a, of b.
MOD (a, b)	Returns the remainder of a divided by b.
POWER (a, b)	Returns a raised to the bth power.
ROUND (a, [b])	Returns a rounded to b decimal places.
SIGN (a)	Returns 1 if a is positive, if a is 0, and -1 if a is less than 0.
SIN	Returns the sine.
SINH	Returns the hyperbolic sine.
SQRT	Returns the square root of the number.
TAN	Returns the tangent.
TANH	Returns the hyperbolic tangent.
TRUNC (a, [b])	Returns a truncated to b decimal places.
GREATEST	Returns the greatest of the specified list of values.
LEAST	Returns the least of the specified list of values.

ABS

The ABS function returns the absolute value of the input. The specification for the ABS function is:

```
FUNCTION ABS (n NUMBER) RETURN NUMBER;
SQL > select abs(-234.56) from dual;
SQL > 234.56
```

ACOS

The ACOS function returns the inverse cosine. The specification for the ACOS function is:

FUNCTION ACOS (n NUMBER) RETURN NUMBER;

where the number n must be between -1 and 1, and the value returned by ACOS is between 0 and pi.

ASIN

The ASIN function returns the inverse sine. The specification for the ASIN function is:

FUNCTION ASIN (n NUMBER) RETURN NUMBER;

where the number n must be between -1 and 1, and the value returned by ASIN is between -pi/2 and pi/2.

NOTES

ATAN

The ATAN function returns the inverse tangent. The specification for the ATAN function is:

FUNCTION ATAN (n NUMBER) RETURN NUMBER;

where the number n must be between -infinity and infinity, and the value returned by ATAN is between -pi/2 and pi/2.

ATAN2

The ATAN2 function returns the result of the tan2 inverse trigonometric function. The specification for the ATAN2 function is:

FUNCTION ATAN (n NUMBER, m NUMBER) RETURN NUMBER;

where the numbers n and m must be between -infinity and infinity, and the value returned by ATAN is between -pi and pi.

As a result, the following holds true:

- atan2(-0.00001. -1) is approximately -pi.
- atan2(0,-1) is pi.

CEIL

The CEIL ("ceiling") function returns the smallest integer greater than or equal to the specified number. The specification for the CEIL function is:

FUNCTION CEIL (n NUMBER) RETURN NUMBER;

Here are some examples of the effect of CEIL:

```
SQL > SELECT CEIL (6) FROM DUAL ;
SQL > 6
SQL >SELECT CEIL (119.1)FROM DUAL;
SQL >120
SQL >SELECT CEIL (-17.2)FROM DUAL;
SQL > -17
```

COS

The COS trigonometric function returns the cosine of the specified angle. The specification for the COS function is:

FUNCTION COS (angle NUMBER) RETURN NUMBER;

where angle must be expressed in radians. A radian is equal to 180/pi or roughly 57.29578.

COSH

The COSH trigonometric function returns the hyperbolic cosine of the specified number. The specification for the COSH function is:

NOTES

SQRT

The SQRT function returns the square root of the input number. The specification for the SQRT function is:

```
FUNCTION SQRT (n NUMBER) RETURN NUMBER;
```

where n must be greater than or equal to 0. If n is negative, you will receive the following error:

```
ORA-01428: argument '-1' is out of range
```

TAN

The TAN trigonometric function returns the tangent of the specified angle. The specification for the TAN function is:

```
FUNCTION TAN (angle NUMBER) RETURN NUMBER;
```

where angle must be expressed in radians. A radian is equal to 180/pi or roughly 57.29578.

TANH

The TANH trigonometric function returns the hyperbolic tangent of the specified number. The specification for the TANH function is:

```
FUNCTION TANH (n NUMBER) RETURN NUMBER;
```

If n is a real number and $i = v^{-1}$ (the imaginary square root of – 1), then the relationship between TAN and TANH can be expressed as follows:

```
TAN (i * n) = i * TANH (h)
```

TRUNC

The TRUNC function truncates the first argument to the number of decimal places specified by the second argument. The specification for the TRUNC function is:

```
FUNCTION TRUNC (n NUMBER, [decimal_places NUMBER])
```

RETURN NUMBER;

The decimal_places argument is optional and defaults to 0, which means that n will be truncated to zero decimal places, a whole number. The value of decimal_places can be less than zero. A negative value for this argument directs TRUNC to truncate or zero-out digits to the left of the decimal point, rather than to the right. Here are some examples:

```
SQL > SELECT TRUNC (153.46) FROM DUAL;
SQL > 153
SQL > SELECT TRUNC (153.46, 1)FROM DUAL;
SQL > 153.4
SQL > SELECT TRUNC (-2003.16, -1)FROM DUAL;
SQL > -2000
```

GREATEST

The GREATEST function evaluates a list of values and returns the greatest value in that list. (The LEAST function, discussed below, returns the least value.) GREATEST accepts two or more arguments, and there is no upper limit on the number of values you can pass to GREATEST, which makes it especially useful. The specification for GREATEST is:

```
FUNCTION GREATEST (expr1, expr2 [, expr3 ...)
```

This example finds the greatest (most recent) of three dates:

```
SQL > SELECT GREATEST (SYSDATE, :emp.hire_date, '13-JAN-1994') FROM DUAL;
```

First expression is a call to the SYSDATE function. Second expression is an Oracle Forms item of type DATE. Third expression is a literal string. This string is converted to a date by PL/SQL with an internal call to TO_DATE. The comparison of the values then proceeds.

LEAST

The LEAST function, the opposite of the GREATEST function, evaluates a list of values and returns the least value in that list. LEAST accepts two or more arguments; there is no upper limit on the number of values you can pass to LEAST, which makes it especially useful. The specification for LEAST is as follows:

```
FUNCTION LEAST (expr1. expr2 [, expr3 ...)
```

NOTES

CHARACTER FUNCTIONA character function is a function that takes one or more character values as parameters and returns either a character value or a number value. The Oracle Server and PL/SQL provide a number of different character datatypes, including CHAR, VARCHAR, VARCHAR2, LONG, RAW, and LONG RAW. In PL/SQL, the three different datatype families for character data are: VARCHAR2 A variable-length character datatype whose data is converted by the RDBMS CHAR The fixed-length datatypeRAW A variable-length datatype whose data is not converted by the RDBMS, but instead is left in "raw" form When a character function returns a character value, that value is always of type VARCHAR2 (variable length), with the following two exceptions: UPPER and LOWER. These functions convert to upper- and lowercase, respectively, and return CHAR values (fixed length) if the strings they are called on to convert are fixed-length CHAR arguments.

PL/SQL provides a rich set of character functions that allow you to get information about strings and modify the contents of those strings in very high-level, powerful ways. Table 5 shows the character functions covered in detail in this chapter. The remaining functions (not covered in this chapter) are specific to National Language Support and Trusted Oracle.

Table 5. The Built-In Character Functions

Name	*Description*
ASCII	Returns the ASCII code of a character.
CHR	Returns the character associated with the specified collating code.
CONCAT	Concatenates two strings into one.
INITCAP	Sets the first letter of each word to uppercase. All other letters are set to lowercase.
INSTR	Returns the location in a string of the specified substring.
LENGTH	Returns the length of a string.
LOWER	Converts all letters to lowercase.
LPAD	Pads a string on the left with the specified characters.
LTRIM	Trims the left side of a string of all specified characters.
REPLACE	Replaces a character sequence in a string with a different set of characters.
RPAD	Pads a string on the right with the specified characters.
RTRIM	Trims the right side of a string of all specified characters.
SOUNDEX	Returns the "soundex" of a string.
SUBSTR	Returns the specified portion of a string.
TRANSLATE	Translates single characters in a string to different characters.
UPPER	Converts all letters in the string to uppercase.

When and why would you use INITCAP? Many Oracle shops like to store all character string data in the database, such as names and addresses, in uppercase. This makes it easier to search for records that match certain criteria.

```
SQL > INITCAP ('HAMBURGERS BY THE BILLIONS AT
MCDONALDS')
SQL > 'Hamburgers By The Billions At Mcdonalds'
```

Use INITCAP with caution when printing reports or displaying data, since the information it produces may not always be formatted correctly.

NOTES

INSTR

The INSTR function searches a string to find a match for the substring and, if found, returns the position, in the source string, of the first character of that substring. If there is no match, then INSTR returns 0. In Oracle7, if nth_appearance is not positive (*i.e.*, if it is 0 or negative), then INSTR always returns 1. In Oracle8, a value of 0 or a negative number for nth_appearance causes INSTR to raise the VALUE_ERROR exception.

The specification of the INSTR function is:

```
FUNCTION INSTR
   (string1 IN VARCHAR2,
   string2 IN VARCHAR2
   [,start_position IN NUMBER := 1
   [, nth_appearance IN NUMBER := 1]])
RETURN NUMBER
```

where string1 is the string searched by INSTR for the position in which the nth_appearance of string2 is found. The start_position parameter is the position in the string where the search will start. It is optional and defaults to 1 (the beginning of string1). The nth_appearance parameter is also optional and also defaults to 1.

Both the start_position and nth_appearance parameters can be literals like 5 or 157, variables, or complex expressions, as follows:

```
INSTR (company_name, 'INC', (last_location + 5) * 10)
```

If start_position is negative, then INSTR counts back start_position number of characters from the end of the string and then searches from that point towards the beginning of the string for the nth match.

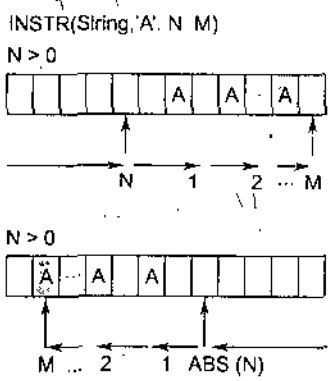


Figure 1. Forward and reverse searches with INSTR

We have found INSTR to be a very handy function -- especially when used to the fullest extent possible. Most programmers make use of (and are even only aware of) only the first two parameters.

NOTES

Let's look at some examples of INSTR. In these examples, you will see all four parameters used in all their permutations. As you write your own programs, keep in mind the different ways in which INSTR can be used to extract information from a string; it can greatly simplify the code you write to parse and analyze character data.

- Find the first occurrence of archie in "bug-or-tv-character?archie":

  SQL > INSTR ('bug-or-tv-character?archie', 'archie');

  SQL > 21The starting position and the nth appearance both defaulted to 1.
- Find the first occurrence of archie in the following string starting from position 14:

  SQL > INSTR ('bug-or-tv-character?archie', 'ar', 14);

  SQL > 21

  In this example We specified a starting position, which overrides the default of 1; the answer is still the same though. No matter where you start your search, the character position returned by INSTR is always calculated from the beginning of the string.
- Find the second occurrence of archie in the following string:

  SQL > INSTR ('bug-or-tv-character?archie', 'archie', 1, 2); SQL > 0

  There is only one archie in the string, so INSTR returns 0. Even though the starting point is the default, I cannot leave it out if I also want to specify a nondefault nth appearance (2 in this case, for "second occurrence").
- Find the second occurrence of "a" in "bug-or-tv-character?archie":

  SQL > INSTR ('bug-or-tv-character?archie', 'a', 1, 2);

  SQL > 15

  The second "a" in this string is the second "a" in "character," which is in the fifteenth position in the string.

  Find the last occurrence of "ar" in "bug-or-tv-character?archie".

  SQL > INSTR ('bug-or-tv-character?archie', 'ar', -1)

  SQL > 21

  Were you thinking that the answer might be 6? Remember that the character position returned by INSTR is always calculated from the leftmost character of the string being position 1. The easiest way to find the last of anything in a string is to specify a negative number for the starting position. We did not have to specify the nth appearance (leaving me with a default value of 1), since the last occurrence is also the first when searching backwards.
- Find the second-to-last occurrence of "a" in "bug-or-tv-character?archie":

  SQL > INSTR ('bug-or-tv-character?archie', 'a', -1, 2); SQL > 15

  No surprises here. Counting from the back of the string, INSTR passes over the "a" in archie, because that is the last occurrence, and searches for the next occurrence. Again, the character position is counted from the leftmost character, not the rightmost character, in the string.
- Find the position of the letter "t" closest to (but not past) the question mark in the following string: bug-or-tv-character?archie tophat:

  SQL >INSTR ('bug-or-tv-character?archie tophat', 't', -14); SQL > 17

We needed to find the "t" just before the question mark. The phrase "just before" indicates to me that I should search backwards from the question mark for the first occurrence. I therefore counted through the characters and determined that the question mark appears at the 20th position. I specified -14 as the starting position so that INSTR would search backwards right from the question mark.

NOTES

LTRIM

The LTRIM function is the opposite of LPAD. Whereas LPAD adds characters to the left of a string, LTRIM removes, or trims, characters from the leading portion of the string. And just as with LPAD, LTRIM offers much more flexibility than simply removing leading blanks. The specification of the LTRIM function is:

```
FUNCTION LTRIM (string1 IN VARCHAR2 [, trim_string IN
VARCHAR2])
RETURN VARCHAR2
```

LTRIM The returns string1 with all leading characters removed up to the first character not found in the trim_string. The second parameter is optional and defaults to a single space.

There is one important difference between LTRIM and LPAD. LPAD pads to the left with the specified string, and repeats that string (or pattern of characters) until there is no more room. LTRIM, on the other hand removes all leading characters which appear in the trim string, not as a pattern, but as individual candidates for trimming.

Here are some examples:

- Trim all leading blanks from ` Way Out in Right Field':

```
SQL >LTRIM (' Way Out in Right Field');
SQL > 'Way Out in Right Field'
```

  Because I did not specify a trim string, it defaults to a single space and so all leading spaces are removed.

- Trim `123' from the front of a string:

```
my_string := '123123123LotsaLuck123';
SQL >LTRIM (my_string, '123');
SQL > 'LotsaLuck123'
```

In this example, LTRIM stripped off all three leading repetitions of "123" from the specified string. Although it looks as though LTRIM trims by a specified pattern, this is not so, as the next example illustrates.

- Remove all numbers from the front of the string:

```
SQL > my_string := '70756234LotsaLuck';
SQL >LTRIM (my_string, '0987612345');
SQL > 'LotsaLuck
```

'By specifying every possible digit in my trim string, I ensured that any and all numbers would be trimmed, regardless of the order in which they occurred (and the order in which I place them in the trim string).-

- Remove all a's, b's, and c's from the front of the string: `abcabccccI LOVE CHILI':

```
SQL >LTRIM ('abcabccccI LOVE CHILI', 'abc')
SQL > 'I LOVE CHILI'
```

LTRIM removed the patterns of "abc", but also removed the individual instances of the letter "c". This worked out fine since the request was to remove any and all of those three letters. What if I wanted to remove only any instance of "abc" as a pattern from the front of the string? We couldn't use LTRIM since it trims off any matching individual characters. To remove a pattern from a string -- or to replace one pattern with another pattern -- you will want to make use of the REPLACE function, which is discussed next.

NOTES

REPLACE

The REPLACE function returns a string in which all occurrences of a specified match string are replaced with a replacement string. REPLACE is useful for searching out patterns of characters and then changing all instances of that pattern in a single function call. The specification of the REPLACE function is:

```
FUNCTION REPLACE (string1 IN VARCHAR2, match_string IN
VARCHAR2
        [, replace_string IN VARCHAR2])
RETURN VARCHAR2
```

If you do not specify the replacement string, then REPLACE simply removes all occurrences of the match_string in string1. If you specify neither a match string nor a replacement string, REPLACE returns NULL.

Here are several examples using REPLACE:

- Remove all instances of the letter "C" in the string "CAT CALL":

  ```
  SQL > REPLACE ('CAT CALL', 'C');
  SQL > 'AT ALL'
  ```

  Because we did not specify a replacement string, REPLACE changed all occurrences of "C" to NULL.
- Replace all occurrences of "99" with "100" in the following string:

  ```
  SQL > REPLACE ('Zero defects in period 99 reached 99%!', '99', '100');
  SQL > 'Zero defects in period 100 reached 100%!'
  ```
- Replace all occurrences of "th" with the letter "z":

  ```
  SQL > REPLACE ('this that and the other', 'th', 'z');
  SQL > 'zis zat and ze ozer'
  ```
- Handle occurrences of a single quote (') within a query criteria string. The single quote is a string terminator symbol. It indicates the start and/or end of the literal string. I ran into this requirement when building query-by-example strings in Oracle Forms. If the user enters a string with a single quote in it, such as:

Customer didn't have change and then we concatenate that string into a larger string, the resulting SQL statement (created dynamically by Oracle Forms in Query Mode) fails, because there are unbalanced single quotes in the string. You can resolve this problem in one of three ways:

1. Simply strip out the single quote before you execute the query. This is workable only if there really aren't any single quotes in the data in the database.
2. If you do allow single quotes, you can then either replace the single quote with a single character wildcard (_) or:
3. Embed that single quote inside other single quotes so that the SQL layer can properly parse the statement.

To replace the single quote with a wild card, you would code:

```
criteria_string := REPLACE (criteria_string, '''', '_');
```

That's right! Four single quotes in sequence are required for SQL to understand that you want to search for one single quote in the criteria string. The first quote indicates the start of a literal. The fourth quote indicates the end of the string literal. The two inner single quotes parse into one single quote. That is, whenever you want to embed a single quote inside a literal, you must place another single quote before it.

This principle comes in handy for the final resolution of the single quote in query criteria problem: change the single quote to two single quotes and then execute the query.

```
criteria_string := REPLACE (criteria_string, '''', '''''');
```

as in:

RTRIM returns string1 with all trailing characters removed up to the first character not found in the trim_string. The second parameter is optional and defaults to a single space.

Here are some examples of RTRIM:

- Trim all trailing blanks from a string:

```
SQL > RTRIM ('Way Out in Right Field  ');
SQL > 'Way Out in Right Field'
```

NOTES

Since we did not specify a trim string, it defaults to a single space and so all trailing spaces are removed.

- Trim all the characters in "BAM! ARGH!" from the end of a string:
- my_string := 'Sound effects: BAM!ARGH!BAM!HAM';

```
SQL > RTRIM (my_string, 'BAM! ARGH!');
SQL > 'Sound effects:
```

'This use of RTRIM stripped off all the letters at the end of the string which are found in "BAM!ARGH!." This includes "BAM" and "HAM," so those words too are removed from the string even though "HAM" is not listed explicitly as a "word" in the trim string. Also, the inclusion of two exclamation marks in the trim string is unnecessary, because RTRIM is not looking for the word "ARGH!", but each of the letters in "ARGH!".

SOUNDEX

The SOUNDEX function allows you to perform string comparisons based on phonetics (the way a word sounds), as opposed to semantics (the way a word is spelled).[1] SOUNDEX returns a character string which is the "phonetic representation" of the argument. The specification of the SOUNDEX function is as follows:

[1] Oracle Corporation used the algorithm in The Art of Computer Programming, Volume 3, by Donald Knuth, to generate the phonetic representation.

```
FUNCTION SOUNDEX (string1 IN VARCHAR2) RETURN VARCHAR2
```

Here are some of the values SOUNDEX generated and how they vary according to the input string:

```
SQL > SOUNDEX ('smith');
SQL > 'S530'
SQL > SOUNDEX ('SMYTHE');
SQL > "S530"
SQL > SOUNDEX ('smith smith');
SQL > 'S532'
SQL > SOUNDEX ('smith z');
SQl > 'S532'
SQL > SOUNDEX ('feuerstein');
SQL > 'F623'
SQL > SOUNDEX ('feuerst');
SQL 'F623'
```

Example.

```
SQL > Select City,temp from weather where Soundex(City) = Soundex('Sidny');
```

Keep the following SOUNDEX rules in mind when using this function:

- The SOUNDEX value always begins with the first letter in the input string.
- SOUNDEX only uses the first five consonants in the string to generate the

return value.

- Only consonants are used to compute the numeric portion of the SOUNDEX value. Except for a possible leading vowel, all vowels are ignored.
- SOUNDEX is not case-sensitive. Upper- and lowercase letters return the same SOUNDEX value.

The SOUNDEX function is useful for ad hoc queries, and any other kinds of searches where the exact spelling of a database value is not known or easily determined.

NOTES

SUBSTR

The SUBSTR function is one of the most useful and commonly used character functions. It allows you to extract a portion or subset of contiguous (connected) characters from a string. The substring is specified by starting position and a length.

The specification for the SUBSTR function is:

```
FUNCTION SUBSTR
  (string_in IN VARCHAR2,
  start_position_in IN NUMBER
  [, substr_length_in IN NUMBER])
RETURN VARCHAR2
```

where the arguments are used as follows:

string_in
  The source string

start_position_in
  The starting position of the substring in string_in

substr_length_in
  The length of the substring desired (the number of characters to be returned in the substring)

The last parameter, substr_length_in, is optional. If you do not specify a substring length, then SUBSTR returns all the characters to the end of string_in (from the starting position specified).

The start position cannot be zero. If the start position is less than zero, then the substring is retrieved from the back of the string. SUBSTR counts backwards substr_length_in number of characters from the end of string_in. In this case, however, the characters which are extracted are still to the right of the starting position. See Figure 2 for an illustration of how the different arguments are used by SUBSTR.

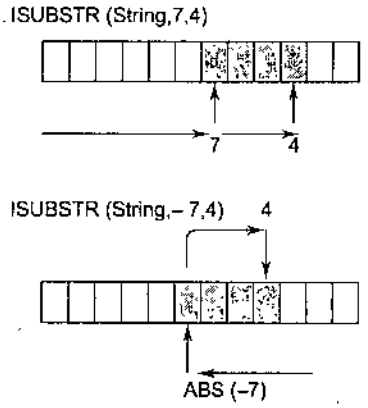


Figure 2. How arguments are used by SUBSTR

```
SQL > UPPER ('123abc');
SQL > '123ABC'
```

SQL is not a case-sensitive language as concerns its own syntax and names of identifiers. It is sensitive to case, however, in character strings, whether found in named constants, literals, or variables. The string "ABC" is not the same as "abc", and this can cause problems in your programs if you are not careful and consistent in your handling of such values.

NOTES

CONVERSION FUNCTIONS

Table 6

Name	*Description*
CHARTOROWID	Converts a string to a ROWID.
CONVERT	Converts a string from one character set to another.
HEXTORAW	Converts from hexadecimal to raw format.
RAWTOHEX	Converts from raw value to hexadecimal.
ROWIDTOCHAR	Converts a binary ROWID value to a character string.
TO_CHAR	Converts a number or date to a string.
TO_DATE	Converts a string to a date.
TO_NUMBER	Converts a string to a number.

CHARTOROWID

The CHARTOROWID function converts a string of either type CHAR or VARCHAR2 to a value of type ROWID. The specification of the CHARTOROWID function is:

```
FUNCTION CHARTOROWID (string_in IN CHAR) RETURN
ROWID
FUNCTION CHARTOROWID (string_in IN VARCHAR2) RETURN
ROWID
```

In order for CHARTOROWID to successfully convert the string, it must be of the format:

```
BBBBBBBB.RRRR.FFFF
```

where BBBBBBBB is the number of the block in the database file, RRRR is the number of the row in the block, and FFFF is the number of the database file. All three numbers must be in hexadecimal format.

Example.

```
SELECT ename FROM emp
WHERE ROWID = CHARTOROWID('AAAAfZAABAAACp8AAO');
ENAME
----------
LEWIS
```

If the input string does not conform to the above format, SQL raises the VALUE_ERROR exception.

NOTES

CONVERT

The CONVERT function converts strings from one character set to another character set. The specification of the CONVERT function is:

```
FUNCTION CONVERT
(string_in IN VARCHAR2,
new_char_set VARCHAR2
[, old_char_set VARCHAR2])
RETURN VARCHAR2
```

The old_char_set is an optional argument. If this third argument is not specified, then the default character set for the database instance is used.

The CONVERT function does not translate words or phrases from one language to another! CONVERT simply substitutes the letter or symbol in one character set with the corresponding letter or symbol in another character set. (A character set is not the same thing as a human language.)

Two commonly used character sets are US7ASCII (U.S. 7-bit ASCII character set) and F7DEC (DEC French 7-bit character set).

Example.

```
SELECT CONVERT('Groß', 'US7ASCII', 'WE8HP')
"Conversion" FROM DUAL;
Conversion
----------
Gross
```

Common character sets include:

US7ASCII	US 7-bit ASCII character set
WE8DEC	DEC West European 8-bit character set
WE8HP	HP West European Laserjet 8-bit character set
F7DEC	DEC French 7-bit character set
WE8EBCDIC500	IBM West European EBCDIC Code Page 500
WE8PC850	IBM PC Code Page 850
WE8ISO8859P1	ISO 8859-1 West European 8-bit character set

HEXTORAW

The HEXTORAW function converts a hexadecimal string from type CHAR or VARCHAR2 to type RAW. The specification of the HEXTORAW function is:

```
FUNCTION HEXTORAW (string_in IN CHAR) RETURN RAW
FUNCTION HEXTORAW (string_in IN VARCHAR2) RETURN
RAW
```

RAWTOHEX

The RAWTOHEX function converts a value from type RAW to a hexadecimal string of type VARCHAR2. The specification of the RAWTOHEX function is:

```
FUNCTION RAWTOHEX (binary_value_in IN RAW) RETURN
VARCHAR2
```

January 1, 4712 B.C. Only in this use of TO_DATE can a number be passed as the first parameter of TO_DATE.

For all other cases, string_in is the string variable, literal, named constant, or expression to be converted, format_mask is the format mask TO_DATE will use to convert the string, and nls_language is a string which specifies the language which is to be used to interpret the names and abbreviations of both months and days in the string. The format of nls_language is as follows:

NOTES

```
'NLS_DATE_LANGUAGE=<language>'
```

where <language> is a language recognized by your instance of the database. You can usually determine the acceptable languages by checking your installation guide.

Here are some examples of the TO_DATE function:

- Convert the string `123188' to a date:

```
SQL >TO_DATE ('123188', 'MMDDYY');
SQL > 31-DEC-1988
```

- Convert a date using the Spanish language:

```
SQL >TO_DATE ('Abril 12 1991', 'Month DD YYYY',
'NLS_DATE_LANGUAGE=Spanish');
SQL >12-APR-1991
```

Any Oracle errors between ORA-01800 and ORA-01899 are related to the internal Oracle date function and can arise when you encounter date conversion errors. You can learn additional nuances of date conversion rules by perusing the different errors and reading about the documented causes of these errors. Some of these rules are:

- A date literal passed to TO_CHAR for conversion to a date cannot be longer than 220 characters.
- You cannot include both a Julian date element (J) and the day of year element (DDD) in a single format mask.
- You cannot include multiple elements for the same component of the date/time in the mask. For example, the format mask YYYY-YYY-DD-MM is illegal because it includes two year elements, YYYY and YYY.
- You cannot use the 24-hour time format (HH24) and a meridian element (*e.g.*, AM) in the same mask.

TO_NUMBER

The TO_NUMBER function converts both fixed- and variable-length strings to numbers using the associated format mask. The specification of the TO_NUMBER function is as follows:

```
FUNCTION TO_NUMBER
(string_in IN CHAR
[, format_mask VARCHAR2 [, nls_language VARCHAR2 ]])
RETURN NUMBER;
FUNCTION TO_NUMBER
(string_in IN VARCHAR2
[, format_mask VARCHAR2 [, nls_language VARCHAR2 ]])
RETURN NUMBER;
```

where string_in is the string containing a sequence of characters to be converted to a number, format_mask is the optional string directing TO_NUMBER how to

convert the character bytes to a number, and nls_language is a string containing up to three specifications of National Language Support parameters, as follows:

NLS_NUMERIC_CHARACTERS

The characters used to specify the decimal point and the group separator in a number. The decimal point character for the American language is a dot (.) while the group separator is a comma (,).

NLS_CURRENCY

The character(s) used to specify the local currency symbol. The currency character for the American language is a dollar sign ($).

NLS_ISO_CURRENCY

The character(s) used to specify the international currency symbol in the string.

The format for nls_language in the call to TO_NUMBER is as follows:

'NLS_NUMERIC_CHARACTERS = "string"'

'NLS_CURRENCY = "string"'

'NLS_ISO_CURRENCY = "string"'

Two contiguous single quotes are needed before and after the values for each string value so that SQL will parse the entire parameter and leave behind a single quote around each value.

NOTES

LOB AND MISCELLANEOUS FUNCTIONS

Table 7. The Built-in LOB Functions

Name	*Description*
BFILENAME	Initializes a BFILE column in an INSERT statement by associating it with a file in the server's filesystem.
EMPTY_BLOB	Returns an empty locator of type BLOB (binary large object).
EMPTY_CLOB	Returns an empty locator of type CLOB (character large object).

Table 8. The Built-in MiscellaneousFunctions

Name	*Description*
DUMP	Returns a string containing a "dump" of the specified expression. This dump includes the datatype, length in bytes, and internal representation.
NVL	Returns a substitution value if the argument is NULL.
SQLCODE	Returns the number of the Oracle error for the most recent internal exception.
SQLERRM	Returns the error message associated with the error number returned by SQLCODE.
UID	Returns the User ID (a unique integer) of the current Oracle session.
USER	Returns the name of the current Oracle user.
USERENV	Returns a string containing information about the current session.
VSIZE	Returns the number of bytes in the internal representation of the specified value.

NOTES

Value, then return X." The NVL function is massively overloaded because any type of data can also have a NULL value. Here is the specification:

```
FUNCTION NVL (string_in IN CHAR, replace_with_in IN CHAR)
RETURN CHAR
FUNCTION NVL (string_in IN VARCHAR2, replace_with_in IN
VARCHAR2)
RETURN VARCHAR2
FUNCTION NVL (date_in IN DATE, replace_with_in IN DATE)
RETURN DATE
FUNCTION NVL (date_in IN NUMBER, replace_with_in IN DATE)
RETURN NUMBER
FUNCTION NVL (date_in IN CHAR, replace_with_in IN DATE)
RETURN BOOLEAN
```

NVL simply provides a much cleaner and more concise way of coding this functionality. And since it is a function, you can call it inline to provide substitution of NULL values where such a state of data would disrupt your program. For example, if you calculate the total compensation of an employee as salary plus compensation, then the expression:

```
salary + commission
```

will be NULL when commission is NULL. With NVL, however, you can be sure that the calculated value will make sense:

```
salary + NVL (commission, 0)
```

SQLCODE

The SQLCODE function returns the number of the exception raised by SQL. The specification for this function is:

FUNCTION SQLCODE RETURN INTEGER

SQLCODE returns values as follows:

- If you reference SQLCODE outside of an exception section, it always returns 0, which means normal, successful completion.
- If you explicitly raise your own user-defined exception, then SQLCODE returns a value of + 1.
- If PL/SQL raises the NO_DATA_FOUND exception, then SQLCODE returns a value of +100.
- In all other cases, SQLCODE returns a negative value. In other words, if you try to convert a date to a string with TO_CHAR and use the wrong format mask, you might encounter the following error message:

  ORA-01830: date format picture ends before converting entire input string

  In this case, SQLCODE returns a value of -1830.

You will find SQLCODE and its sibling function, SQLERRM, most useful in the WHEN OTHERS exception handler. If an error is trapped by WHEN OTHERS, you do not know which exception was raised or which error was encountered. You can, however, use SQLCODE to find out, as shown in this example:

SQLERRM

The SQLERRM returns the error message associated with the specified code. The specification for this function is:

```
FUNCTION SQLERRM (code_in IN INTEGER := SQLCODE)
RETURN VARCHAR2
```

If you do not provide an error code when you call SQLERRM, it uses the value returned by SQLCODE (see the preceding section). If SQLCODE returns 0, then SQLERRM returns the following message:

```
ORA-0000; normal, successful completion
```

If PL/SQL has raised an internal Oracle error or you pass a negative value to SQLERRM, then the function returns the error message provided by Oracle Corporation. If you pass (or allow SQLCODE to pass) a value of +100 to SQLERRM, it returns this message:

NOTES

```
ORA-01403: no data found
```

Any other positive value passed to SQLERRM will result in this message:

```
User-Defined Exception
```

The maximum length of a message returned by SQLERRM is 512 bytes. This length includes the error code and all nested messages that may have been flagged by the compiler.

UID

The UID function returns an integer that uniquely identifies the current user. This integer is generated by the Oracle RDBMS when a user connects to the database. The specification for UID is as follows:

```
FUNCTION UID RETURN NUMBER
```

When called inline, the UID function looks like a variable since it has no arguments. Remember that when you call UID you will actually issue a SQL call to the RDBMS to extract the UID information for the user. Furthermore, in a distributed SQL statement, the UID always returns the value identifying the user on the local database. You cannot obtain the UID for connections to other, remote databases.

USER

The USER function returns the name of the current account. The specification for USER is as follows:

```
FUNCTION USER RETURN VARCHAR2
```

Like UID, when called inline, the USER function looks like a variable since it has no arguments. Remember that when you call USER you actually issue a SQL call to the RDBMS to extract the account name for the user. Furthermore, in a distributed SQL statement, the USER always returns the value identifying the user on the local database. You cannot obtain the USER for connections to other, remote databases.

The most common use for the USER function is to initialize an application session with configuration for a user.

Most of the applications we build have a system configuration table (with one row for each system or application) and a separate user configuration table (with one row for each user in each application). This user configuration table might have the following columns:

NOTES

RDBMS account name

The name of the Oracle account, which matches the value returned by USER

User name

The actual name of the user, as in: STEVEN FEUERSTEIN.

Business data

Information about the user that relates to the business of the application, such as the user's department and default printer.

Preference data

Information about the preferences of the user, such as "Display Toolbar" or "Automatically pop up a list of values boxes."

Assuming an Oracle Forms-based set of screens, each screen the user is able to enter from a Windows icon will contain a When-New-Form-Instance trigger. This trigger calls a procedure to transfer the information from the user configuration table to GLOBAL variables that are then available to all screens for the duration of the session.

USERENV

The USERENV function returns information about the current user session or environment. The specification for USERENV is as follows:

```
FUNCTION USERENV (info_type_in IN VARCHAR2) RETURN VARCHAR2
```

where info_type_in can be one of these values (specified in a named constant or a literal string in single quotes). The following list gives options and descriptions of what they return:

ENTRYID

An auditing entry identifier

LANGUAGE

The language, territory, and character set used by your session. The value is returned in this format: language_territory.characterset

SESSIONID

An auditing session identifier

TERMINAL

The operating system identifier for your current session's terminal. The format of this information will clearly be dependent on your underlying operating system.

VSIZE

The VSIZE function returns the number of bytes used by the internal representation of the input expression. The specification for VSIZE is:

```
FUNCTION VSIZE (expr_in IN DATE) RETURN NUMBER
FUNCTION VSIZE (expr_in IN VARCHAR2) RETURN NUMBER
```

```
FUNCTION VSIZE (expr_in IN NUMBER) RETURN NUMBER
FUNCTION VSIZE (expr_in IN CHAR) RETURN NUMBER
```

If the expression is NULL, then VSIZE returns NULL.

```
SQL> SELECT VSIZE (hiredate) FROM emp WHERE ROWNUM=1;

VSIZE(HIREDATE)
---------------
              7
```

We included the WHERE clause for ROWNUM equal to 1 so that we would receive the answer (7 bytes) only once. Otherwise, we would have had the "opportunity" to learn that the VSIZE of hiredate is 7 -- for every record in the emp table.

NOTES

Interestingly, if you apply VSIZE to the SYSDATE function, you get a slightly different answer:

```
SQL> SELECT VSIZE (SYSDATE) FROM dual;
VSIZE(SYSDATE)
--------------
             8
```

The Decode Function

Unlike the translate function which performs a character by character replacement the DECODE function does a a value by value replacement.

DECODE (s, search1, result1, search2, result2) - Compares s with search1, search2, etc. and returns the corresponding result when there is a match.

Or

```
DECODE(value,if1,then1,if2,then2,if3,then3,.....,else)
```

SQL > Select vencode, decode (venname, 'frances','francis')name, tel_no from vendor_master where vencode = 'v001';

Venco	Name	Tel_no
v001	Francis	611892

EXAMPLE

SQL > Select feature, section , decode(page,'1','Welcome','Go To || page) from newspaper;

In the above command page number is decoded. If the page number is 1 then the word 'Welcome' are substituted. If the page number is anything else then word 'Go To ' are concated with page number.

AGGREGATE FUNCTIONS (Group Function)

Function	*Usage*
AVG(expression)	Computes the average value of a column by the expression
COUNT(expression)	Counts the rows defined by the expression
COUNT(*)	Counts all rows in the specified table or view
MIN(expression)	Finds the minimum value in a column by the expression
MAX(expression)	Finds the maximum value in a column by the expression
SUM(expression)	Computes the sum of column values by the expression
STDDEV(expression)	Standard devition of all values for group of rows
VARIANCE(expression)	Variance of all values for group of rows

The general syntax of an aggregate function is:

aggregate_function_name ([ALL | DISTINCT] expression)

The aggregate function name may be *AVG, COUNT, MAX, MIN,* or *SUM*. The *ALL* clause, which is the default behavior and does not actually need to be specified, evaluates all rows when aggregating the value of the function. The *DISTINCT* clause uses only distinct values when evaluating the function.

NOTES

AVG h

The AVG Function returns the average value for the column when applied to a column containing numeric data. The following is the syntax for the AVG Function.

- AVG (column_name)

Example

SELECT AVG(commission_rate) FROM sales;

COUNT

The COUNT function has three variations.

COUNT(*) counts all the rows in the target table whether they include nulls or not.

COUNT(expression) computes the number of rows with non-NULL values in a specific column or expression.

COUNT(DISTINCT expression) computes the number of distinct non-NULL values in a column or expression.

Examples

This query counts all rows in a table:

SELECT COUNT(*) FROM publishers;

The following query finds the number of different countries where publishers are located:

SELECT COUNT(DISTINCT country) "Count of Countries"

FROM publishers

MIN

The MIN Function returns the data item with the lowest value for a column when applied to a column containing numeric data. If you apply the MIN Function to a CHARACTER value, it returns the first value in the sorted values for that column. The following syntax is for the MIN Function.

MIN(column_name)

Example

SELECT MIN(commission_rate) FROM sales;

MAX

The MAX Function returns the data item with the highest value for a column when applied to a column containing numeric data. If you apply the MAX Function to a CHARACTER value, it returns the last value in the sorted values for that column. The following syntax is for the MAX Function.

MAX(column_name)

Example

SELECT MAX(commission_rate) FROM sales;

SUM

The SUM Function returns the sum of all values in the specified column. The result of the SUM Function has the same precision as the column on which it is operating. The following syntax is for the SUM Function.

NOTES

SUM(column_name)

Example

SELECT SUM(ytd_sales) FROM sales;

STDDEV

The STDDEV Function returns the standard deviation of values in the specified column. The result of the STDDEV Function has the same precision as the column on which it is operating. The following syntax is for the STDDEV Function.

STDDEV(column_name)

Example

SELECT STDDEV(Basic) FROM salary;

VARIANCE

The VARIANCE Function returns the variance of values in the specified column. The result of the Variance Function has the same precision as the column on which it is operating. The following syntax is for the Variance Function.

VARIANCE(column_name)

Example

SELECT VARIANCE(Basic) FROM salary;

RELATIONS in SQL

Foreign Key

Foreign key represent relationships between two or more tables. A foreign key is a column or a group of columns whose value are derived from the primary key or unique key of any other table.

The table in which the foreign key is defined is called a foreign table. The table that defines the primary or unique key and is referenced by the foreign key is called Master table.

The master table can be referenced in the foreign key definition by using the ***REFERENCES****. If the name of the table is not specified, by default, Oracle references the primary key in the master table.*

Insert, Update and Delete operation in Foreign Key

The existence of a foreign key implies that the table with the foreign key is related to the master table from which the foreign key is derived. A foreign key must have a corresponding primary key or unique key value in the master table.

NOTES

If the user tries to delete a record in the master table when corresponding.records exists in the foreign table oracle will display an error message.

ON DELETE CASCADE

On delete cascade constraint can change the default bahavior of the foreign key. When the on delete cascade constraint is specified in the foreign key definition, if the user deletes a record in the master table, all corresponding records in the foreign table along with the record in the master table will be deleted.

Principles of Foreign Key / References constraint

a) *Rejects an Insert or Update of a value, if a corresponding value does not currently exist in the master table.*
b) *Rejects a Delete for the master table if corresponding records in the foreign table exist.*
c) *Foreign key must reference a primary key or unique column in primary table.*
d) *Foreign key requires that the foreign key column and the constraint column have same data types.*

Example :-

For maintaining the record of students in a college we can create two tables as :

(i) *Master Table Student_Address will contain only one record for every student like Roll_No, Name, Address, Place, Pin.*
(ii) *Marks Table will store more than one record for every student i.e., every time the student appears for an exam, a record will be added to the marks table like Roll_No, Subject, Exam_Date, Marks. So marks table will considered as transaction file.*

```
SQL > Create Table Student_Address
        (Roll_No        Number(4) Primary Key,
        Name            Varchar2(20),
        Address         Varchar2(20),
        Place           Varchar2(10),
        PIN             Number(6) );
SQL > Create Table Marks
        (Roll_No        Number(4) References Student_address on delete cascade,
        Subject         Varchar2(10),
        Exam_Date       Date,
        Marks           Number (3) );
```

Master Table :

Student_Address

Roll_No	*Name*	*Address*	*Place*	*Pin*
1.	*Don Mario*	*1, Alistonia*	*Tokio*	*21398*
2.	*Khaliona*	*2, Amaltash*	*Japan*	*23456*
3.	*Lee Vinn*	*5, King Estate*	*England*	*67864*
4.	*Uccharal*	*9, Queen Street*	*Singapore*	*89765*
5.	*Breet Lee*	*90, Alistonia*	*Kohima*	*98765*

Foreign Table :

Marks

Roll_No	Subject	Exam_Date	Marks
1	Oracle	12-Jan-2004	90
2	C++	12-Feb-2004	95
2.	OOPs	12-March-2004	85
3	Windows-NT	12-March-2004	80

NOTES

So from the above table can the marks file contain a row for roll number 10 ?

No it cannot and that is because roll number 10 does not exist in the student master table or there is no reference for that number in the Parent table. This is referential integrity which ensures that there are no rows in the transaction table if there is no reference for it in the master table.

FOREIGN KEY CONSTRAINT AT THE COLUMN LEVEL:

Syntax *:- columnname datatype(size) REFERENCES tablename [(columnname)]*

[ON DELETE CASCADE]

Example *:- Create a table sales_order_detail table with its primary key as d_order_no and prod_no.The foreign key is d_order_no, referencing column order_no in the sale_order table.*

```
SQL > Create table sale_order
      ( order_no     varchar2 (8) PRIMARY KEY,
      order_date date,
      client_no number (6),
      order_status varchar2(10) );
SQL > Create table sales_order_detail
      (d_order_no varchar2 (8) REFERENCES sale_order,
      prod_no varchar2 (8),
      qty_ordered number (9),
      prod_price number (6, 2),
      PRIMARY KEY (d_order_no,prod_no) );
```

In the above example the reference key word points to the table sale_order. The table sale_order has the column order_no as its primary key column.So when no column is specified in the foreign key definition, Oracle applies an default link to the primary key column like order_no of the table sale_order.

Foreign key definition definition will be specified as given below :

(d_order_no varchar2(8) REFERENCES sale_order (order_no))

FOREIGN KEY AT THE TABLE LEVEL :

Syntax *:-*

Foreign Key (column name) References tablename [(column name)]

EXAMPLE :

```
SQL > Create table sales_order_detail
      (d_order_no varchar2 (8) REFERENCES sale_order,
      prod_no varchar2 (8),
```

qty_ordered number (9),
prod_price number (6, 2),
PRIMARY KEY (d_order_no,prod_no),
Foreign Key (d_order_no) REFERENCES sale_order);

JOIN

NOTES

All of the queries up until this point have been useful with the exception of one major limitation - that is, we can select only one table at a time with SELECT statement. There is one of the most beneficial features of SQL & relational database systems - the **"Join"**. "Join" makes relational database systems "relational".

Joins allow to link data from two or more tables together into a single query result from one single SELECT statement.

The FROM clause allows more than 1 table in its list, however simply listing more than one table will very rarely produce the expected results. The rows from one table must be correlated with the rows of the others. This correlation is known as joining.

pno	descr	color
P1	Wid	Blue
P2	Wid	Red
P3	Dongle	Green

P Table (parts)

sno	name	city
S1	Pierre	Paris
S2	John	London
S3	Mario	Rome

s Table (suppliers)

sno	pno	qty
S1	P1	NULL
S2	P1	200
S3	P1	1000
S3	P2	200

Sp Table

An example can best illustrate the rationale behind joins. The following query:

SELECT * FROM sp, p

Produces:

sno	pno	qty	pno	Descry	color
S1	P1	NULL	P1	Wid	Blue
S1	P1	NULL	P2	Wid	Red
S1	P1	NULL	P3	Dongle	Green
S2	P1	200	P1	Wid	Blue
S2	P1	200	P2	Wid	Red
S2	P1	200	P3	Dongle	Green
S3	P1	1000	P1	Wid	Blue
S3	P1	1000	P2	Wid	Red
S3	P1	1000	P3	Dongle	Green
S3	P2	200	P1	Wid	Blue
S3	P2	200	P2	Wid	Red
S3	P2	200	P3	Dongle	Green

Each row in Sp is arbitrarily combined with each row in P, giving 12 result rows (4 rows in SP X 3 rows in P.) This is known as a **cartesian product.**

EQUI JOIN or INNER JOIN

A join, which is based on equalities, is called an equi join.The equi join combining rows that have equivalent values for the specified columns.

A more usable query would correlate the rows from Sp with rows from P, for instance matching on the common column - pno:

```
SELECT *
FROM sp, p
WHERE sp.pno = p.pno
```

This produces:

NOTES

sno	pno	qty	pno	Descry	color
S1	P1	NULL	P1	Wid	Blue
S2	P1	200	P1	Wid	Blue
S3	P1	1000	P1	Wid	Blue
S3	P2	200	P2	Wid	Red

Rows for each part in P are combined with rows in Sp for the same part by matching on part number (pno). In this query, the WHERE Clause provides the join predicate, matching pno from p with pno from sp.

The join in this example is known as an **inner equi-join**. equi meaning that the join predicate uses = (equals) to match the join columns. Other types of joins use different comparison operators. For example, a query might use a *greater-than* join.

The term *inner* means only rows that match are included. Rows in the first table that have no matching rows in the second table are excluded and vice versa (in the above join, the row in *p* with pno P3 is not included in the result.) An outer join includes unmatched rows in the result.

More than 2 tables can participate in a join. This is basically just an extension of a 2 table join. 3 tables -- *a*, *b*, c, might be joined in various ways:

- *a* joins *b* which joins *c*
- *a* joins *b* and the join of *a* and *b* joins *c*
- *a* joins *b* and *a* joins *c*

Plus several other variations. With *inner* joins, this structure is not explicit. It is implicit in the nature of the join predicates. With outer joins, it is explicit.

This query performs a 3 table join:

```
SELECT name, qty, descr, color
FROM s, sp, p
WHERE s.sno = sp.sno
AND sp.pno = p.pno
```

It joins S to Sp and Sp to P, producing:

name	qty	descr	color
Pierre	NULL	Wid	Blue
John	200	Wid	Blue
Mario	1000	Wid	Blue
Mario	200	Wid	Red

Note that the order of tables listed in the FROM clause should have no significance, nor does the order of join predicates in the WHERE clause.

Another example:

```
SELECT employee_info.employeeid, employee_info.lastname,
employee_sales.comission
FROM employee_info, employee_sales
WHERE employee_info.employeeid = employee_sales.employeeid;
```

This statement will select the employee id, lastname (from the employee_info table), and the comission value (from the employee_sales table) for all of the rows where the employeeid in the employee_info table matches the employeeid in the employee_sales table.

NOTES

Non equi-join

A non equi join specifies the relationship between columns belonging to different tables by making use of the relational operators (>,<,<=,>=,<>) other than = (equal).

SQL > Select item, qty_order from item, order where ((item.order < order. qty_order) ;

Outer Joins

An *inner* join excludes rows from either table that don't have a matching row in the other table. An outer join provides the ability to include unmatched rows in the query results. The outer join combines the unmatched row in one of the tables with an artificial row for the other table. This artificial row has all columns set to *null*.

The outer join is specified in the FROM clause and has the following general format:

table-1 { LEFT | RIGHT | FULL } OUTER JOIN table-2 ON predicate-1

predicate-1 is a join predicate for the outer join. It can only reference columns from the joined tables. The LEFT, RIGHT or FULL specifiers give the type of join:

- LEFT : only unmatched rows from the left side table (*table-1*) are retained
- RIGHT : only unmatched rows from the right side table (*table-2*) are retained
- FULL : unmatched rows from both tables (*table-1* and *table-2*) are retained

Outer join example:

```
SELECT pno, descr, color, sno, qty
FROM P LEFT OUTER JOIN Sp ON P.pno = Sp.pno
```

pno	descr	color	sno	qty
P1	Wid	Blue	S1	NULL
P1	Wid	Blue	S2	200
P1	Wid	Blue	S3	1000
P2	Wid	Red	S3	200
P3	Dongle	Green	NULL	NULL

We can use outer join using (+) plus symbol also:

Example

Table :- Player (ID_NO is the primary key)

ID_No	NAME
10	Sachin Tandulkar
20	Leander Paes
30	Mahesh Bhupathi
40	Saurav Ganguly

MATCH (MatchNO is the primary key, ID_NO is the foreign key)

MATCHNO	ID_NO	MATCH_DATE	OPPONENT
1	20	10-Aug-2003	Washington
2	30	12-Aug-2003	Vishwanathan
3	20	13-Aug-2003	Sampras
4	30	20-Mar-2004	London

SQL > Select player.id_no, name,match_date,opponent from player,match

Where player.id_no = match.id_no (+);

Above query will return following result :

NOTES

ID_NO	NAME	MATCH_DATE	OPPONENT
10	Sachin Tandulkar	NULL	NULL
20	Leander Paes	10-Aug-2003	Washington
20	Leander Paes	13-Aug-2003	Sampras
30	Mahesh Bhupathi	12-Aug-2003	Vishwanathan
30	Mahesh Bhupathi	20-Mar-2004	London
40	Saurav Ganguly	NULL	NULL

If matching records are not present in the second file, certain names from the first file are not listed at all.To retrieve these records also we have to perform an outer join operation using the outer join operator(+).The data, which is not available in the second file, will be presented as null values.

TABLE ALIAS

The name of the table can be coded with an alias making the query easier to code. Table alias are used to make multiple table queries shorter and more readable. *The alias can be used instead of the table name throughout the query.*

Example :

SQL > Select PLAYER.RoLLNO, NAME, MATCH_DATE,OPPONENT FROM PLAYER, MATCH WHERE PLAYER.ROLLNO = MATCH.ROLLNO;

We can write the above query using alias as given below :

SQL > Select P.ROLLNO, NAME, MATCH_DATE, OPPONENT FROM *PLAYER P, MATCH M WHERE P.ROLLNO = M.ROLLNO;*

In the above example alias 'P' and "M' is used for table names player and match. To retrieve all the columns from both the table we can write P.* and M.* also.

Self Joins

A query can join a table to itself. Self joins have a number of real world uses. For example, a self join can determine which parts have more than one supplier:

```
SELECT DISTINCT a.pno
FROM Sp a, Sp b
WHERE a.pno = b.pno
AND a.sno < > b.sno
```

pno
P1

As illustrated in the above example, self joins use *correlation* names to distinguish columns in the select list and where predicate. In this case, the references to the same table are renamed - *a* and *b*.

NOTES

SET OPERATORS

The SQL UNION operator combines the results of two queries into a *composite* result. The following set operators are used by SQL in joining queries to retrieve rows :-

- UNION
- UNION ALL
- INTERSECT
- MINUS

The columns in the select statement joined using the set operators should strictly follow the rules given below.

- The queries , which are related by a set operator should have the same number of columns and the corresponding columns, must be of the same data type.
- Such a query should not contain any column of type long.
- The label under which the rows are displayed are those from the first select statement.

UNION	**Returns all rows returned by either query does not eliminates duplicates.**
UNION ALL	**Returns all rows returned by either query eliminates including duplicates rows.**
INTERSECT	**Returns all distinct rows or common rows returned by both queries. A row must exist in both query outputs to be returned in the final result.**
MINUS	**Produce rows returned by the first query but not the second.**

UNION

The union operator returns all distinct rows selected by both queries.

SQL > SELECT SECTION_ID FROM BOOK

UNION

SELECT SECTION_ID FROM SECTION ;

section_id
10
5
6
7
9
11

Note that the returned values have been grouped because only rows distinct in both tables are returned.

UNION ALL

Union All returns all rows returned by either query eliminates including duplicates rows.

SQL >SELECT SECTION_ID FROM BOOK

UNION ALL

SELECT SECTION_ID FROM SECTION;

NOTES

Section_id
9
9
10
5
9
6
10
9
11
10
5
6
7
9
11

Now all the rows have been returned (including duplicates).

INTERSECT

Intersect returns all distinct rows or common rows returned by both queries. A row must exist in both query outputs to be returned in the final result.

If we want to see only the rows that that match in both queries than we use INTERSECT.

SQL > SELECT SECTION_ID FROM BOOK

INTERSECT
SELECT SECTION_ID FROM SECTION;

section_id
10
5
6
9
11

MINUS

Produce rows returned by the first query but not by the second query.

SQL > SELECT SECTION_ID FROM SECTION

MINUS
SELECT SECTION_ID FROM BOOK ;

section_id
7

Section_id 7 is returned as the only row that doesn't match in both result sets.

GROUP BY clause

The GROUP BY clause will gather all of the rows together that contain data in the specified column(s) and will allow aggregate functions to be performed on the one or more columns.

NOTES

GROUP BY operates on the rows from the FROM clause as filtered by the WHERE clause. It collects the rows into groups based on common values in the grouping columns. Except nulls, rows with the same set of values for the grouping columns are placed in the same group. If any grouping column for a row contains a null, the row is given its own group.

GROUP BY was added to SQL because aggregate functions (like SUM) return the aggregate of all column values every time they are called, and without the GROUP BY function it was impossible to find the sum for each individual group of column values.

The syntax for the GROUP BY function is:

GROUP BY clause syntax:

SELECT column1,

SUM(column2),

FROM "list-of-tables"

GROUP BY "column-list";

Retrieve a list of the highest paid salaries in each dept:

SELECT max(salary), dept

FROM employee

GROUP BY dept;

This statement will select the maximum salary for the people in each unique department. Basically, the salary for the person who makes the most in each department will be displayed. Their, salary and their department will be returned.

Example :-

This "Sales" Table:

Company	Amount
NIIT	5500
IBM	4500
NIIT	7100

And This SQL:

SELECT Company, SUM(Amount) FROM Sales;

Returns this result:

Company	SUM(Amount)
NIIT	17100
IBM	17100
NIIT	17100

The above code is invalid because the column returned is not part of an aggregate. A GROUP BY clause will solve this problem:

SELECT Company,SUM(Amount) FROM Sales GROUP BY Company;

Returns this result:

Company	SUM(Amount)
NIIT	12600
IBM	4500

NOTES

For example, take a look at the items_ordered table. Let's say you want to group everything of quantity 1 together, everything of quantity 2 together, everything of quantity 3 together, etc. If you would like to determine what the largest cost item is for each grouped quantity (all quantity 1's, all quantity 2's, all quantity 3's, etc.), you would enter:

SQL > SELECT quantity, max(price) FROM items_ordered

GROUP BY quantity;

Example using the SUM function

For example, you could also use the SUM function to return the name of the department and the total sales (in the associated department).

SELECT department, SUM (sales) as "Total sales"
FROM order_details
GROUP BY department;

Because you have listed one column in your SELECT statement that is not encapsulated in the SUM function, you must use a GROUP BY clause. The department field must, therefore, be listed in the GROUP BY section.

Using the grouping the *sp* table on the *pno* column:

Sno	pno	Qty	
S1	P1	NULL	'P1' Group
S2	P1	200	
S3	P1	1000	
S3	P2	200	'P2' Group

A Set Function can compute the total quantities for each group:

Sno	pno	qty		qty total
S1	P1	NULL	'P1' Group	1200
S2	P1	200		
S3	P1	1000		
S3	P2	200	'P2' Group	200

Null columns are ignored in computing the summary. The Set Function -- SUM, computes the arithmetic sum of a numeric column in a set of grouped/aggregate rows. For example,

SELECT pno, SUM(qty)
FROM sp
GROUP BY pno;

Pno	
P1	1200
P2	200

Set Functions have the following general format:

set-function ([DISTINCT|ALL] column-1)

NOTES

Example using the COUNT function

For example, you could use the COUNT function to return the name of the department and the number of employees (in the associated department) that make over $25,000 / year.

```
SELECT department, COUNT (*) as "Number of employees"
FROM employees
WHERE salary > 25000
GROUP BY department;
```

Example using the MIN function

For example, you could also use the MIN function to return the name of each department and the minimum salary in the department.

```
SELECT department, MIN (salary) as "Lowest salary"
FROM employees
GROUP BY department;
```

Example using the MAX function

For example, you could also use the MAX function to return the name of each department and the maximum salary in the department.

```
SELECT department, MAX (salary) as "Highest salary"
FROM employees
GROUP BY department;
```

- COUNT : count of rows
- SUM : arithmetic sum of numeric column
- AVG : arithmetic average of numeric column; should be SUM()/COUNT().
- MIN : minimum value found in column
- MAX : maximum value found in column

The result of the COUNT function is always integer. The result of all other Set Functions is the same data type as the argument.

The Set Functions skip columns with *nulls*, summarizing *non-null* values. COUNT counts rows with non-null values, AVG averages non-null values, and so on. COUNT returns 0 when no non-null column values are found; the other functions return *null* when there are no values to summarize.

A Set Function argument can be a column or an scalar expression.

The DISTINCT and ALL specifiers are optional. ALL specifies that *all* non-null values are summarized; it is the default. DISTINCT specifies that *distinct* column values are summarized; duplicate values are skipped. Note: DISTINCT has no effect on MIN and MAX results.

COUNT also has an alternate format:

COUNT(*)

Which counts the underlying rows regardless of column contents.

Set Function examples:

SQL > SELECT pno, MIN(sno), MAX(qty), AVG(qty), COUNT(DISTINCT sno)FROM sp GROUP BY pno;

pno				
P1	S1	1000	600	3
P2	S3	200	200	1

SQL > SELECT sno, COUNT(*) parts FROM sp GROUP BY sno;

NOTES

sno	parts
S1	1
S2	1
S3	2

HAVING clause

The HAVING clause allows to specify conditions on the rows for each group - in other words, which rows should be selected will be based on the specified conditions. The HAVING clause should follow the GROUP BY clause.

HAVING was added to SQL because the WHERE keyword is used to specify conditions to retrieve rows of a table but where clause could not be used against aggregate functions (like SUM, AVG), and without HAVING it would be impossible to test for result conditions.

HAVING clause syntax:

SELECT column1, SUM(column2) FROM "list-of-tables"GROUP BY "column-list"

HAVING "condition";

HAVING can best be described by example. Let's say you have an employee table containing the employee's name, department, salary, and age. If you would like to select the average salary for each employee in each department, you could enter:

SELECT dept, avg(salary)

FROM employee

GROUP BY dept;

But, let's say that you want to ONLY calculate & display the average if their salary is over 20000:

SELECT dept, avg(salary)

FROM employee

GROUP BY dept

HAVING avg(salary) > 20000;

SQL > SELECT sno, COUNT(*) parts

FROM sp

GROUP BY sno

HAVING COUNT(*) > 1

sno	parts
S3	2

NOTES

SUBQUERY

A subquery is a query within a query. A select statement is used as part of another SQL statement. The outer statement is called the parent and the nested query passes a value to the outer query is called child.The nested query executes first.

Important Rules about Subqueries

- The nested query must return a single column.
- The result can only contain columns from the tables referenced in the outermost query.
- The nested query must return a single row when a standard operator such as =, <, > is used.
- The between operator cannot be used with a subquery.
- Subquery can also be used in INSERT, UPDATE and DELETE statement.

Example :-

To list the employees who earn less than the average salary in the organization, a group function AVG must be used to calculate the average salary, however, the group function cannot be used to calculate the average salary.In such case a subquery may be used.

SQL > Select * from salary where basic < (Select avg(basic) from salary);

There are 3 basic types of subqueries in SQL:

- **Predicate Subqueries :** extended logical constructs in the WHERE and HAVING clause.
- **Scalar Subqueries :** standalone queries that return a single value; they can be used anywhere a scalar value is used.
- **Table Subqueries :** queries nested in the FROM clause.

All subqueries must be enclosed in parentheses.

Predicate Subqueries

Predicate subqueries are used in the WHERE and HAVING clause. Each is a special logical construct. Except for EXISTS, predicate subqueries must retrieve one column (in their select list.)

- IN Subquery

  The IN Subquery tests whether a scalar value matches the single query column value in any subquery result row. It has the following general format:

  value-1 [NOT] IN (query-1)

  Using NOT is equivalent to:

  NOT value-1 IN (query-1)

  For example, to list parts that have suppliers:

  ```
  SELECT *
  FROM p
  WHERE pno IN (SELECT pno FROM sp);
  ```

pno	descr	color
P1	Wid	Blue
P2	Wid	Red

  The Self Join example in the previous subsection can be expressed with an IN Subquery:

  ```
  SELECT DISTINCT pno
  FROM sp a
  ```

```
WHERE pno IN (SELECT pno FROM sp b WHERE a.sno <>
b.sno);
```

pno
P1

Note that the subquery where clause references a column in the outer query (*a.sno*). This is known as an *outer reference*. Subqueries with outer references are sometimes known as *correlated subqueries*.

NOTES

- **Quantified Subqueries**

A quantified subquery allows several types of tests and can use the full set of comparison operators. It has the following general format:

value-1 {=|>|<|>=|<=|<>} {ANY|ALL|SOME} (query-1)

The comparison operator specifies how to compare *value-1* to the single query column value from each subquery result row. The ANY, ALL, SOME specifiers give the type of match expected. ANY and SOME must match at least one row in the subquery. ALL must match all rows in the subquery.

For example, to list all parts that have suppliers:

```
SELECT *
FROM p
WHERE pno =ANY (SELECT pno FROM sp);
```

Pno	descr	color
P1	Wid	Blue
P2	Wid	Red

A self join is used to list the supplier with the highest quantity of each part (ignoring null quantities):

```
SELECT *
FROM sp a
WHERE qty >ALL (SELECT qty FROM sp b
 WHERE a.pno = b.pno
 AND a.sno <> b.sno
 AND qty IS NOT NULL);
```

sno	pno	qty
S3	P1	1000
S3	P2	200

Example : Select all the fields of the table SALARY and the name from EMP for those employees who had BASIC less than the average salary.

```
SQL > SELECT SALARY.*,
 EMP_NAME from SALARY, EMP
 WHERE EMP.EMP_NO = SALARY.EMP_NO and
 BASIC < (SELECT AVG(BASIC) FROM SALARY);
```

- EXISTS Subqueries

The EXISTS Subquery tests whether a subquery retrieves at least one row, that is, whether a qualifying row *exists*. It has the following general format

EXISTS(query-1)

Any valid EXISTS subquery must contain an *outer reference*. It must be a *correlated subquery*.

Note: the select list in the EXISTS subquery is not actually used in evaluating the EXISTS, so it can contain any valid select list (though * is normally used).

To list parts that have suppliers:

```
SELECT *
FROM p
WHERE EXISTS(SELECT * FROM sp WHERE p.pno = sp.pno);
```

pno	descr	Color
P1	Widget	Blue
P2	Widget	Red

NOTES

Example : To display the names of the employees who belong to SLS department.

```
SELECT EMP_NAME FROM EMP
WHERE EXISTS (SELECT * FROM SALARY WHERE
EMP.EMP_NO = SALARY.EMP_NO AND DEPARTMENT = 'SLS');
```

Scalar Subqueries

The Scalar Subquery can be used anywhere a value can be used. The subquery must reference just one column in the select list. It must also retrieve no more than one row.

When the subquery returns a single row, the value of the single select list column becomes the value of the Scalar Subquery. When the subquery returns no rows, a database *null* is used as the result of the subquery. Should the subquery retreive more than one row, it is a *run-time* error and aborts query execution.

A Scalar Subquery can appear as a scalar value in the select list and where predicate of an another query. The following query on the *sp* table uses a Scalar Subquery in the select list to retrieve the supplier city associated with the supplier number (*sno* column in *sp*):

```
SELECT pno, qty, (SELECT city FROM s WHERE s.sno = sp.sno)
FROM sp;
```

pno	qty	city
P1	NULL	Paris
P1	200	London
P1	1000	Rome
P2	200	Rome

The next query on the *sp* table uses a Scalar Subquery in the where clause to match parts on the color associated with the part number (*pno* column in *sp*):

```
SELECT *
FROM sp
WHERE 'Blue' = (SELECT color FROM p WHERE p.pno = sp.pno);
```

sno	Pno	qty
S1	P1	NULL
S2	P1	200
S3	P1	1000

Table Subqueries

Table Subqueries are queries used in the FROM clause, replacing a table name. Basically, the result set of the Table Subquery acts like a base table in the from

list. Table Subqueries can have a correlation name in the from list. They can also be in outer joins.

The following two queries produce the same result:

```
SELECT p.*, qty
FROM p, sp
WHERE p.pno = sp.pno
AND sno = 'S3';
```

pno	Descr	color	qty
P1	Wid	Blue	1000
P2	Wid	Red	200

```
SELECT p.*, qty
FROM p, (SELECT pno, qty FROM sp WHERE sno = 'S3')
WHERE p.pno = sp.pno;
```

pno	Descr	color	qty
P1	Wid	Blue	1000
P2	Wid	Red	200

For example,

```
SELECT *
FROM p
WHERE (SELECT COUNT(*) FROM sp WHERE sp.pno=p.pno) > 0;
```

pno	descr	color
P1	Wid	Blue
P2	Wid	Red

Parts with multiple suppliers:

```
SELECT *
FROM p
WHERE (SELECT COUNT(DISTINCT sno) FROM sp WHERE
sp.pno=p.pno) > 1;
```

pno	descr	color
P1	Wid	Blue

CREATE TABLE FROM A TABLE

Example :

SQL > Create table saldup as select * from salary;

A new table named, Saldup is created which will be a duplicate of the table named salary.

Example :

SQL > Create table empdup as select empno, ename, job from emp;

A new table empdup will created with three columns of emp table like empno, ename, job

Subquery to Copy Structure of a table

SQL > Create table saldup as select * from salary where 1>2 ;

NOTES

A new table named saldup is created which will have the structure of the table named, salary. The rows will not be copied because of the condition 1 > 2 that is always false. So you can copy the structure of any table using any false condition like 2 > 3 also.

Subquery to Insert data

SQL > Insert into saldup (select * from salary where department = 'SLS');

NOTES

Rows from the table salary where Department is SLS will be inserted into the table saldup

SQL > Insert into saldup (emp_no, basic, department)(select emp_no, basic, department from salary where department = 'SLS');

Three columns from the table SALARY where Department is SLS will be inserted into the table SALDUP.

Subquery to DELETE data

Syntax :

Delete from <table> where <column_name> in (Select <column_name> from <table> where <query>);

Example

SQL > Delete from emp where emp_no in (select emp_no from salary where deduction = 150);

Update Data using Subquery

SQL > Update saldup set
(Basic, Commission) = (select sum(basic),sum (commission)from salary
where emp_no = 1001) where department ='SLS';

SUMMARY

- In this chapter we saw tables, views and index. Tables are the basic building block of any relational database management system. They contain the rows and columns of your data table in a relational system consists of a row of column headings, together with zero or more of data values. A table is created using the CREATE TABLE statement. An existing base table can be modified using the ALTER TABLE statement. A base table can be deleted at any time by using DROP TABLE statement.
- A view is named table that is represented, not by its own physically separate stored data but by its definition in terms of other named tables (base tables or views). Base table is a parent table that is not defined in terms of other tables or in other words, it is autonomous and have its own right. Views are not autonomous and so not exist by their own right. A view is created defined using the CREATE VIEW statement. All views are not updateable. That is INSERT UPDATE and DELETE operations cannot be performed on all views. Non-updateable views are also called 'read-only views'. Most RDBMSs add more restrictions on the views that be updated, if you want to delete or remove an existing view you can do so by using the DBMS VIEW statement.
- An index is a structure that provides faster access to the rows of a table based on the values of one or more columns. The index stores data values and pointers to the rows when their data values occur. There are different types of indexes-composite, unique, clustered and table indexes are created using the CREATE INDEX statement. Indexes can be dropped explicitly with the DROP INDEX command.
- When you want to find the total, average, maximum, minimum, etc., of a column or columns, SQL finds the answers to these kinds of questions using aggregate functions and the GROUP BY and HAVING clauses of the SELECT statement. The aggregate functions greatly enhance the power of the SQL statements. They let you summarize the data from the tables. An aggregate function takes an entire column of data as its argument and produces a single data item that summarizes the column. The aggregate functions provided by SQL are: COUNT (), COUNT(*), SUM(), AVG(), MAX() and MIN(). COUNT() is used to count the number of values in a column. COUNT(*) is used to count the number of rows of the query results. SUM() is used to find the sum of the values in a column. AVG() is used to find the average of the values in a column. MAX() is used to find the maximum value in a column. MIN() is for finding the minimum value in a column.

TEST YOURSELF

1. What are the uses of views?
2. How are data retrieved using views?
3. How are views updated? What are the restrictions in updating views?
4. What is the use of the WITH CHECK OPTION clause?
5. What are updateable and non-updateable views?
6. What are the advantages of views?
7. What are the situations in which views are best suited?
8. How are views deleted?
9. What are indexes?

NOTES

10. What is the different between a book index and a table index?
11. Why do we use an index?
12. How are indexes created?
13. What are the different types of indexes?
14. What are the composite indexes?
15. What are unique indexes?
16. What are clustered indexes?
17. How are indexes deleted?
18. What are the uses of indexes?
19. What are aggregate functions or column functions?
20. What are agregate functions used for?
21. Name the most commonly used aggregate functions.
22. What are the rules to be followed when using aggregate functions?
23. What is the difference between COUNT () and COUNT (*)?
24. What is the use of SUM () and AVG ()?
25. What is the purpose of MIN () and MAX ()?

Fill in the Blanks

1. A ____ in a relational system consists of a row of column headings, together with zero or more rows of data values.
2. The ____ of a table is responsible for granting others access to his/her tables.
3. You create a table using the ____ statement.
4. If you specify the ____ option in the column definition, then the RDBMS will insert a null in that column if the user odes not specify a value.
5. When you use the NOT NULL option in the column definition, you can specify either the ____ or ____ option.
6. If you specify the ____ option in the column definition, then the RDBMS will substitute the default values for the columns.
7. An existing base table can be modified by using the ____ statement.
8. New columns can be added to a table using the ____ statement.
9. Primary and foreign key specifications can be added or removed to or from a table using the ____ statement.
10. An existing base table can be deleted at any time by using the ____ statement.
11. A ____ is a named table that is represented, not by its own physically separate stored date. But by its definition in terms of other named tables.
12. A view is created or defined using the ____ statement.
13. If the clause ____ is included in the view-definition, then all INSERTs and UPDATEs on that view will be cgeck4ed to ensure that the newly INSERTed or UPDATED rows do not violate the view-defining conditions.
14. Non-updateable views are called ____
15. If you want to delete or remove an existing view you can do so by using the ____ statement.
16. If ____ is specified in the DROP VIEW statement, then if the view is referenced in any other view definition or in an integrity constraint the DROP VIEW will fail.
17. If ____ is specified in the DROP VIEW statement, the DROP VIEW will always succeed and any referencing views and integrity constraints will automatically be dropped too.
18. An ____ is a structure that provides faster access to the rows of a table based on the values of one or more columns.
19. Indexes are created using the ____ statement.
20. When an index is made up of more than one column it is called a ____ .

21. A ____ is one in which no two rows are permitted or in which no duplicate values are allowed for the same index value.
22. When you create ____ , it means that the system will sort the rows of a table when there is a change made to the index.
23. In a ____ , the physical order of the rows is not the same as their indexed order.
24. Indexes can be dropped explicitly using the ____ command.
25. _____ let you summarize the data from the tables.
26. The aggregate functions provided by SQL are : _____, _____, _____, _____, _____, _____.
27. Except for COUNT (*), the argument may be preceded by the keyword _____, to eliminate the duplicate rows before the function is applied to a column.
28. The keyword DISTINCT is not allowed for _____ function.
29. Any _____ in the column is eliminated before the function is applied.
30. In the case of _____ nulls are handled like normal values.
31. In computers using _____ character set, digits come before letters in the sorting sequence and all uppercase characters come before the lowercase characters.
32. On machines that use the _____ character set, the order is lower case characters, uppercase characters and then digits.
33. _____ is used to count the number of values in a column.
34. _____ is used to count the number of rows of the query results.
35. _____ is used to find the sum of the values in a column.
36. _____ is used to find the average of the values in a column.
37. _____ is used to find the maximum value in a column.
38. _____ is for finding the minimum value in a column.
39. _____ is the standard command set used to communicate with the management systems.
40. The first commercial RDBMS was _____ from relational software Inc.
41. In 1978, the _____ approved the SQL databae language project, which let to the formulation of the initial SQL standard language.
42. A _____ is a database that can maintain information such as video, images, sound as well as in the traditional forms.
43. A _____ is a collection of tables, where a table is an unordered collection of rows.
44. SQL as a language is _____ of the way it is implemented internally.
45. All SQL operations are performed at a _____ level.
46. Data types CHARACTER and Character VARYING are known collectively as _____.
47. Data types BIT and BIT VARYING are known as _____.
48. Character and bit string data types are together known as _____.
49. Data types NUMERIC, DECIMAL, INTEGER and SMALLINT are known as _____.
50. The Data type FLOAT (or REAL or DOUBLE PRECISION) is known as _____.
51. Exact and approximate numeric data types are collectively known as _____.
52. _____ and _____ are used to perform operations such as addition or subtraction or comparison on the data items in an SQL statement.
53. There are two types of operatiors: _____ and _____.
54. _____ operators are used in SQL expressions to add, subtract, multiply, divide and negate data values.
55. _____ operators are used in SQL expressions to add, subtract, multiply, divide and negate data values.

NOTES

56. _____ operators are used to compare one expression with another.
57. A _____ operator is used to produce a single result from combining the two separate conditions.
58. _____ operators combine the results of two separate queries into a single result.
59. UNION, INTERSECT and MINUS are _____ operators.

NOTES

True or False

1. You can create, modify and delete tables using the Date Definition Language (DDL) commands.
2. In most SQL implementations, the user who creates the table is usually its owner.
3. The CREATE TABLE statement creates new base table.
4. A base table is not an autonomous named table.
5. If you specify the NOT NULL option it means that the column should have a value.
6. If you don't specify a value for that column which has a NOT NULL option, the system will substitute a value for the column.
7. The statement "CREATE TABLE CATALOG LIKE BOOK", will create a table called CATALOG with the same structure as BOOK.
8. If NOT NULL UNIQUE is specified, the RDBMS will ensure that the values for that column are unique or in other words, duplicates will be allowed.
9. When a table is created from an existing table, only the structure is copied; the primary, alternate and foreign key definitions are not inherited.
10. ALTRE TABLE enables us to delete columns from a table.
11. ALTER TABLE statement does not support any kind of change to the width or data type of an existing column neither does it support the deletion of an existing column.
12. Alternate key specifications can be changed using the ALTER TABLE statement.
13. The specified base table is removed from the system. All indexes and views defined for the table are also automatically dropped.
14. Views are autonomous and exist by their own right.
15. If column names are not specified explicity in the view definition statement then the view inherits the column names of the source of the view.
16. A view maintains the characteristics of a table.
17. All views are updateable.
18. A views defined on a non-updateable view is not updateable.
19. If the definition of the view involves either a GROUP BY or a HAVING clause at the outermost level, then the view is updateable.
20. If RESTRICT is specified in the DROP VIEW statement and if there aren't any integrity constraints the DROP VIEW will succeed and the view will be deleted.
21. The view stores data values and pointers to the rows where those data values occur.
22. In the index the data values are sorted and stored in the ascending or descending order.
23. The decision of the RDBMS to choose the access path is definitely based on the presence of an index.
24. Unique indexes are usually created on the primary key or any of the candidate key of a table.
25. When you use the clustered index, the rows in the table will not be in the same order as that of the index.

26. Since clustered index controls the physical location of data, there can be only one clustered index per table.
27. There can be only one non-clustered indexes per table.
28. A clustered index is usually very advantageous when many rows with contiguous values are being retrieved.
29. Clustered indexes are much faster than non-clustered ones.
30. Whenever the base table is dropped the indexes for that table is not automatically dropped.
31. Aggregate functions are also known as column functions.
32. An aggregate function takes an entire column of data as its argument and produces a single data item that summarizes the column.
33. For SUM and AVG the argument must be of type numeric.
34. The DISTINCT keyword is illegal for MAX and MIN.
35. The special function COUNT (*) is used to all rows without any duplicate elimination.
36. When using an aggregate function, the argument can involve not more than two aggregate function references or table expressions at any level of nesting.
37. What are bit string data types? Explain with examples.
38. What are numeric data types? Explain with examples.
39. What are exact numeric data types? Explain with examples.
40. What are approximate numeric data types? Explain with examples.
41. What are the different categories of SQL commands?
42. What are data definition language (DDL) commands? Explain with examples.
43. What are data manifulation language (DML) commands? Explain with examples.
44. What are data control languare (DCL) commands? Explain with examples.
45. What are data query language (DQL) commands? Explain with examples.
46. What are data administration statements (DAS)? Explain with examples.
47. What are transaction control statements (TCS)? Explain with examples.
48. What are SQL operators?
49. What are the different arithmetic operators in SQL? Explain with examples.
50. What is the difference between unary and binary arithmetic operators?
51. What are comparison operators in SQL? Explain with examples.
52. What are logical operators in SQL? Explain with examples.
53. What is a truth table? Give the truth table for conditional expressions.
54. What are set operators in SQL? Explain with examples.
55. What is the operator precedence in SQL?
56. All tasks related to relational data management cannot be done using SQL.
57. The basic features of all the different SQL implementations are the same-they have the same base, the ANSI SQL standard.
58. The entire field of relational database management system has its origins in Dr. Codd's paper.
59. The first commercial RDBMS was IBM's DB2.
60. As a result of the experience with SEQUEL-XRM, a revised and refined version of SEQUEL was released in 1976-77 called SEQUEL/2.
61. The SQL-3 language was broken into four levels for conformance testing and development-basic, entry, intermediate and full.
62. The SQL-3 language was released as a draft language in 1999, resulting in elevation to standard level later the same year.
63. The SQL-3 standard consists of eight parts-Framework, foundation, Call-level interface (CLI), Persistent Stored Modules (PSM), Bindings, Transactions, Temporal and Object.

NOTES

NOTES

64. SQL statements can be invoked either interactively in a terminal session but cannot be embedded in application programs.
65. Each SQL request is parsed by the RDBMS before execution, to check for proper syntax and to optimize the request.
66. SQL is coded with embedded data-navigational instructions.
67. Applications witten in SQL can be easily ported across systems.
68. Data types are classification of a particular type of information.
69. Data Definition language is used to create, after and delete database objects.
70. The data control language commands let users insert data into the database, modify and delete the data in the database.
71. Data Query Language enables the users to query one or more table to get the information they want.
72. The data manipulation language consists of commands that control the user access to the database objects.
73. Data administration commands allow the user to perform audits and analysis on operations within the database.
74. The unary operator operates on two operands.
75. Transaction control statements manage all the changes made by the DML statement.
76. The binary operator operates on two operands.
77. The result of a comparison is True, False or Unknown.
78. IN, NOT IN, IS NULL, IS NOT NULL, ALL, ANY, SOME, EXISTS, NOT EXISTS, BETWEEN and NOT BETWEEN are set operators.
79. AND, OR and NOT are logical operators.

Multiple Choice

1. Which of the following consists of a row of column headings, together with zero or more rows of data values?
   - ❒ COMPOSITE INDEX
   - ❒ UNIQUE INDEX
   - ❒ TABLE
   - ❒ None of above
2. Which of the following command is used to create a table
   - ❒ MAKE TABLE
   - ❒ CONSTRUCT TABLE
   - ❒ CREATE TABLE
   - ❒ None of the above
3. In which of the following cases will the RDBMS specify a default value for the column if there are no values for it?
   - ❒ NOT NULL WITH DEFAULT
   - ❒ COLUMN DEFAULT
   - ❒ NOT NULL UNIQUE
   - ❒ None of the above
4. Which of the following statements will create a table called CATALOG with the same structure as BOOK?
   - ❒ CREATE TABLE BOOK LIKE CATALOG;
   - ❒ CREATE TABLE BOOK FROM CATALOG;
   - ❒ CREATE TABLE CATALOG LIKE BOOK;
   - ❒ None of the above
5. Which of the following statements is used to modify a table?
   - ❒ MODIFY TABLE
   - ❒ ALTER TABLE
   - ❒ UPDATE TABLE
   - ❒ All of the above
6. Which of the following database object does not physically exist?
   - ❒ Base table
   - ❒ Index
   - ❒ View
   - ❒ None of the above
7. Which of the following can generate SQL statements?
   - ❒ CASE Tools
   - ❒ SQL Generators
   - ❒ PowerBuilder's DataWindow
   - ❒ All of the above

8. Who developed the Sructured English Query Language?
   - ☐ E.F. Codd
   - ☐ D. Chamberlain
   - ☐ Chris Date
   - ☐ None of the above
9. System R was based on which product?
   - ☐ SQL
   - ☐ SEQUEL
   - ☐ SQL-XRM
   - ☐ SEQUEL/2
10. Which of the following is the first commercial RDBMS?
    - ☐ DB2
    - ☐ INGRESS
    - ☐ ORACLE
    - ☐ None of the above
11. Which of the following is IBM's first RDBMS?
    - ☐ DB2
    - ☐ IMS
    - ☐ SQL/DS
    - ☐ None of the above
12. Which of the following is the company now know as the Oracle Corporation?
    - ☐ Stepware Inc.
    - ☐ Relational Software Inc.
    - ☐ Rational Inc.
    - ☐ Oracle Software Inc.
13. Which of the following is the latest SQL standard?
    - ☐ SQL-82
    - ☐ SQL-92
    - ☐ SQL-3
    - ☐ SQL-8
14. In which year did SQL-3 become an official SQL standard?
    - ☐ 1993
    - ☐ 1995
    - ☐ 1999
    - ☐ 2000
15. Which of the following is a conformance level of SQL-92?
    - ☐ Entry
    - ☐ Intermediate
    - ☐ Full
    - ☐ All of the above
16. SQL-3 standard consists of how many parts?
    - ☐ 3
    - ☐ 5
    - ☐ 8
    - ☐ None of the above
17. Which of the following is the expansion of CLI?
    - ☐ Call-level Interface
    - ☐ Cell-level interface
    - ☐ Command-level Interface
    - ☐ None of the above
18. Which of the following is a database that can maintain information such as video, images, and sound as well as the information in the traditional forms?
    - ☐ Universal server
    - ☐ RDBMS
    - ☐ DBMS
    - ☐ None of the above
19. What is the process that is done to SQL before execution, to check for proper syntax and to optimize the request called?
    - ☐ Syntax checking
    - ☐ Performance tuning
    - ☐ Parsing
    - ☐ Optimizing
20. Which of the following is a valid SQL data type?
    - ☐ CHARACTER
    - ☐ NUMERIC
    - ☐ FLOAT
    - ☐ All of the above
21. Which of the following is not a character string?
    - ☐ 'Alexis Leon'
    - ☐ 1234
    - ☐ 'Don't do'
    - ☐ 'QWERTY'
22. Which of the following is a bit character string?
    - ☐ B'1'
    - ☐ B'0'
    - ☐ X'A5'
    - ☐ All of the above
23. Which of the following is not an exact numeric?
    - ☐ 9E9
    - ☐ 99
    - ☐ -99E-9
    - ☐ 9
24. Which of the following is an approximate numeric?
    - ☐ +999E-9
    - ☐ 0.99E9
    - ☐ - 9.99E-9
    - ☐ All of the above
25. Which of the following is an approximate numberic?
    - ☐ CREATE
    - ☐ ALTER
    - ☐ DROP
    - ☐ SELECT

NOTES