# RELATIONAL DATABASE MANAGEMENT SYSTEM…

# CHAPTER 1. INTRODUCTION

**Data:-**

- Data are raw facts and figures that on their own have no meaning
- These can be any alphanumeric characters i.e. text, numbers, symbols
- Example Yes, Yes, No, Yes, No, Yes, No, Yes
- 42, 63, 96, 74, 56, 86
- None of the above data sets have any meaning until they are given a **CONTEXT** and **PROCESSED** into a useable form

**Data into Information**

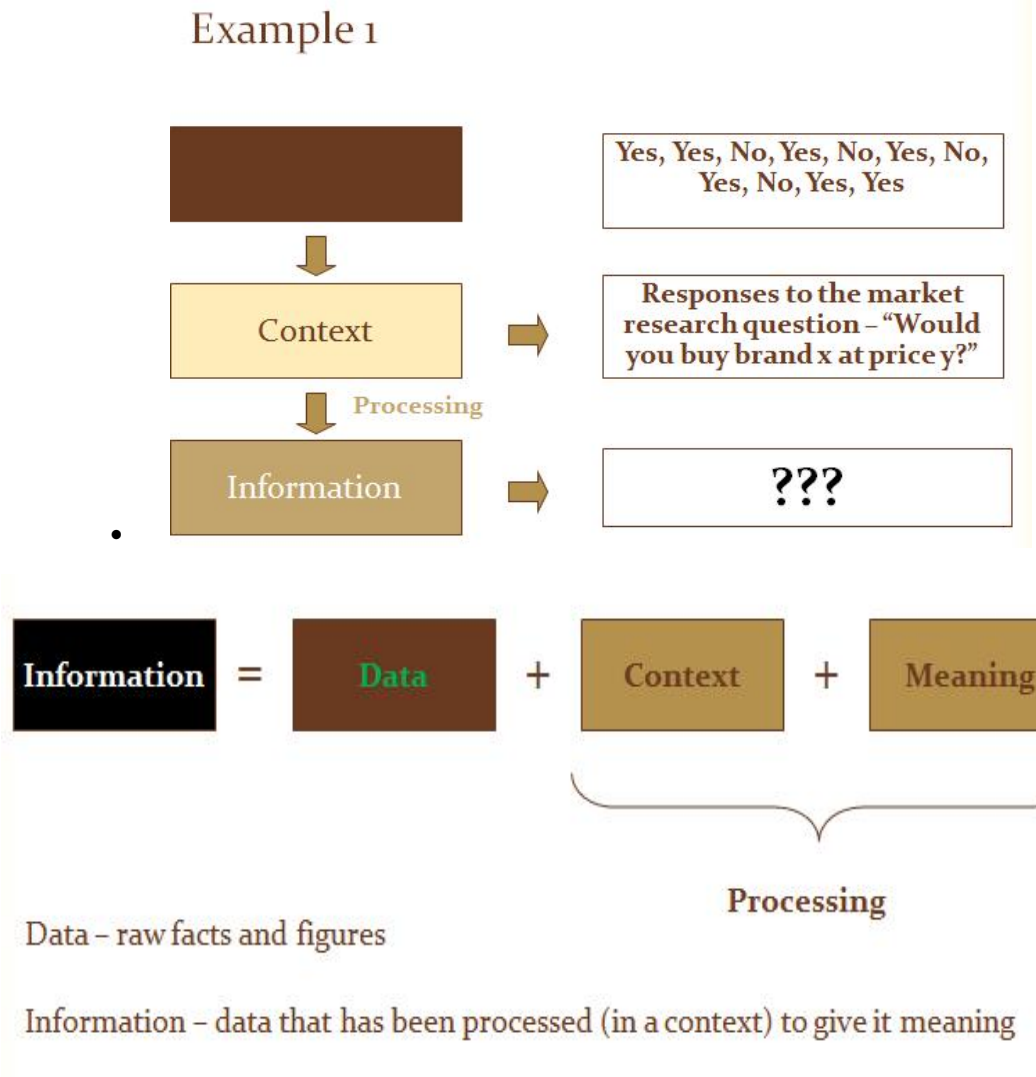
- To achieve its aims the organisation will need to process data into information.
- Data needs to be turned into meaningful information and presented in its most useful format
- Data must be processed in a context in order to give it meaning

**Information:-**

- Data that has been processed within a context to give it meaning

OR

- Data that has been processed into a form that gives it meaning
- **Example**
- In the next 3 examples explain how the data could be processed to give it meaning
- What information can then be derived from the data?




**Knowledge:-**

- Knowledge is the understanding of rules needed to interpret information
- Using the previous examples:
    - A Marketing Manager could use this information to decide whether or not to raise or lower price y

## 1.1 An Introduction to database.

**Definition Of Database**:- A **database** is a collection of data, typically describing the activities of one or more related organizations. For example, a university database might contain information about the following:

*Entities* such as students, faculty, courses, and classrooms.

*Relationships* between entities, such as students' enrollment in courses, faculty teaching courses, and the use of rooms for courses

**Definition Of DBMS** :-A **database-management system** (DBMS) is a collection of interrelated data and a set of programs to access those data

Goal Of DBMS :- The primary goal of a DBMS is to provide a way to store and retrieve database information that is both *convenient* and *efficient*.

**Database System Applications**

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.

• ***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 balances 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 customer, 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.

## 1.2 Disadvantages of File Processing System:-

- **DATA REDUNDANCY AND INCONSISTENCY.**
  - Different programmers create the files and application programs over a long period, the various files are likely to have different formats and the programs may be written in several programming languages. Moreover, the same information may be duplicated in several places

(files). For example, the address and telephone number of a particular customer may appear in a file that consists of savings-account records and in a file that consists of checking-account records. This redundancy leads to higher storage and access cost.

- **DATA INCONSISTENCY**:-
  - That is, the various copies of the same data may no longer agree. For example, a changed customer address may be reflected in savings-account records but not elsewhere in the system.
- **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. The point here is that conventional file-processing environments do not allow needed data to be retrieved in a convenient and efficient manner. More responsive data-retrieval systems are required for general use.
- **DATA ISOLATION.**
  - Because data are scattered in various files, and files may be in different formats, writing new application programs to retrieve the appropriate data is difficult.
- **INTEGRITY PROBLEMS**.
  - The data values stored in the database must satisfy certain types of **consistency constraints.**
- **ATOMICITY PROBLEMS**.
  - A computer system, like any other mechanical or electrical device, is subject to failure.
- **CONCURRENT-ACCESS ANOMALIES**.
  - For the sake of overall performance of the system and faster response, many systems allow multiple users to update the data simultaneously. In such an environment, **interaction of concurrent updates may result in inconsistent data.**
- **SECURITY PROBLEMS**.
  - Not every user of the database system should be able to access all the data. For example, in a banking system, payroll personnel need to see only that part of the database that has information about the various bank employees. They do not need access to information about customer accounts. But, since application programs are added to the system in an ad hoc manner, enforcing such security constraints is difficult.

### 1.3 ADVANTAGES OF A DBMS over File Processing System.

Using a DBMS to manage data has many advantages:

- **DATA INDEPENDENCE:** Application programs should be as independent as possible from details of data representation and storage. The DBMS can provide an abstract view of the data to insulate application code from such details.
- **E_CIENT DATA ACCESs:** A DBMS utilizes a variety of sophisticated techniques to store and retrieve data efficiently. This feature is especially important if the data is stored on external storage devices.
- **DATA INTEGRITY AND SECURITY:** If data is always accessed through the DBMS, the
- DBMS can enforce integrity constraints on the data. For example, before inserting salary information for an employee, the DBMS can check that the department budget is not exceeded. Also, the DBMS can enforce *access controls* that govern what data is visible to different classes of users.
- **DATA ADMINISTRATION:** When several users share the data, centralizing the administration of data c`an offer significant improvements. Experienced professionals who understand the nature of the data being managed, and how deferent groups of users use it, can be responsible for organizing the data representation to minimize redundancy and for fine tuning the storage of the data to make retrieval efficient.
- **CONCURRENT ACCESS AND CRASH RECOVERY:** A DBMS schedules concurrent accesses to the data in such a manner that users can think of the data as being accessed by only one user at a time. Further, the DBMS protects users from the effects of system failures.
- **REDUCED APPLICATION DEVELOPMENT TIME:** Clearly, the DBMS supports many Important functions that are common to many applications accessing data stored in the DBMS.

### 1.4 View of Data

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

**Data Abstraction:**

• **Physical level**. The lowest level of abstraction describes *how* the data are actually stored. The physical level describes complex low-level data structures in detail.

• **Logical level**. The next-higher level of abstraction describes *what* data are stored in the database, and what relationships exist among those data. The logical level thus describes the entire database in terms of a small number of relatively simple structures. Database administrators, who must decide what information to keep in the database, use the logical level of abstraction.

• **View level**. The highest level of abstraction describes only part of the entire database .The view level of abstraction exists to simplify their interaction with the system. The system may provide many views for the same database.

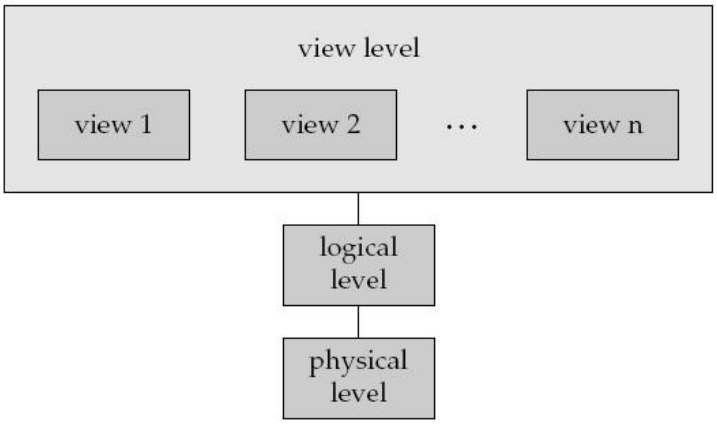



**Figure 1.1** The three levels of data abstraction.

**Def of RDBMS:-**The database which stores data in the tables that have relationships with other tables in the database is called RDBMS or Relational Database Management System.

## 1.5 DIFFERENCE BETWEEN DBMS AND

## RDBMSDBMS vs RDBMS

The software application that enables the users to store the data is known as a database. In database architecture, there are different implementations and theories in order to store physical data. The database which stores data in the tables that have relationships with other tables in the database is called RDBMS or Relational Database Management System. However, in DBMS or Database Management System, there are no relationships among tables.

### DBMS

- DBMS is defined as the software program that is used to manage all the databases that are stored on the network or system hard disk. There are different types of database management systems and some of them are configured for specific purposes.
- DBMS is available in different forms as a tool that is used to manage databases. Some popular DBMS solutions include DB2, Oracle, FileMaker and Microsoft Access. Using these products, privileges or rights can be created that can be specific to particular users. It means that the administrators of the database can grant specific rights to some users or assign different levels of administration.
- Every DBMS has some fundamental elements. First is the implementation of the modeling language which defines the language used for each database. Second, DBMS also administers the data structures. Data query language is the third element of a DBMS. Data structures work with data query language in order to make sure that irrelevant data cannot be entered into the database used in the system.

### RDBMS

- The database system in which the relationships among different tables are maintained is called Relational Database Management System. Both RDBMS and DBMS are used to store information in physical database.
- RDBMS solution is required when large amounts of data are to be stored as well as maintained. A relational data model consists of indexes, keys, foreign keys, tables and their relationships with other tables. Relational DBMS enforces the rules even though foreign keys are support by both RDBMS and DBMS.

- In 1970s, Edgar Frank Codd introduced the theory of relational database. Thirteen rules were defined by Codd for this relational theory or model. Relationships among different types of data Is the main requirement of the relational model.
- RDMS can be termed as the next generation of database management system. DBMS is used as a base model in order to store data in a relational database system. However, complex business applications use RDBMS rather than DBMS.

DBMS vs. RDBMS

• Relationship among tables is maintained in a RDBMS whereas this not the case DBMS as it is used to manage the database.

• DBMS accepts the 'flat file' data that means there is no relation among different data whereas RDBMS does not accepts this type of design.

• DBMS is used for simpler business applications whereas RDBMS is used for more complex applications.

• Although the foreign key concept is supported by both DBMS and RDBMS but its only RDBMS that enforces the rules.

• RDBMS solution is required by large sets of data whereas small sets of data can be managed by DBMS.

| Sr. No. | DBMS | RDBMS |
|---|---|---|
| 1 | Old version of software to handle the databases. | Latest version of software for handling databases. |
| 2. | Can relate one table to another table. | RDBMS can relate one database to another database. |
| 3. | Data security is low as compare to RDBMS. | Level of data security is very high as compare to DBMS. |
| 4. | Data storage capacity is less as compare to RDBMS. | Data storage capacity is very high. |
| 5. | Not easy to maintain data integrity. | Data integrity is one of the most important features of RDBMS. It can be maintained easily in RDBMS. |
| 6. | Works better in single user or few user systems. | Works very efficiently and give good performance over the network. |
| 7. | All Codd's 12 rules are not followed. | All Codd's 12 rules are followed. |

## 1.6 Data Independence:-

**Data independence** is the type of data transparency that matters for a centralized DBMS. It refers to the immunity of user applications to make changes in the definition and organization of data.

- Physical data independence is the ability to modify the physical schema without causing application programs to be rewritten. Modifications at the physical level are occasionally necessary to improve performance. It means we change the physical storage/level without affecting the conceptual or external view of the data. The new changes are absorbed by mapping techniques.
- Logical data independence is the ability to modify the logical schema without causing application program to be rewritten. Modifications at the logical level are necessary whenever the logical structure of the database is altered (for example, when money-market accounts are added to banking system). Logical Data independence means if we add some new columns or remove some columns from table then the user view and programs should not changes. It is called the logical independence. For example: consider two users A & B. Both are selecting the empno and ename. If user B adds a new column salary in

his view/table then it will not affect the external view user; user A, but internal view of database has been changed for both users A & B. Now user A can also print the salary. It means if we change in view then program which use this view need not to be changed.

Logical data independence is more difficult to achieve than physical data independence, since application programs are heavily dependent on the logical structure of the data that they access.

Physical data independence means we change the physical storage/level without affecting the conceptual or external view of the data. Mapping techniques absorbs the new changes.

## 1.7 Database Users:-

There are four different types of database-system users, differentiated by the way they expect to interact with the system.

**1. Naive users;-**

- Are unsophisticated users who interact with the system by invoking one of the application programs that have been written previously. For example, a bank teller who needs to transfer $50 from account *A* to account *B* invokes a program called *transfer*.
- The typical user interface for naive users is a forms interface, where the user can fill in appropriate fields of the form. Naive users may also simply read *reports* generated from the database.

**2. Application programmers;-**

- Are computer professionals who write application programs. Application programmers can choose from many tools to develop user interfaces. **Rapid application development (RAD)** tools are tools that enable an application programmer to construct forms and reports without writing a program.
- Most major commercial database systems include a fourth generation language. e.g programming languages that combine imperative control structures (for example, for loops, while loops and if-then-else statements) with statements of the data manipulation language.

**3. Sophisticated users;-**

- Interact with the system without writing programs. Instead, they form their requests in a database query language.
- Class of tools for analysts is **data mining** tools, which help them find certain kinds of patterns in data.

**4. Specialized users;-**

- are sophisticated users who write specialized database applications that do not fit into the traditional data-processing framework.
- Among these applications are computer-aided design systems, knowledge base and expert systems, systems that store data with complex data types (for example, graphics data and audio data), and environment-modeling systems.

## 1.8 Database Administrators;-

A person who has such central control over the system is called a **database administrator** (**DBA**). The **functions of a DBA include:**

- **Schema definition**. The DBA creates the original database schema by executing set of data definition statements in the DDL.
- **Storage structure and access-method definition**.
- **Schema and physical-organization modification**. The DBA carries out changes to the schema and physical organization to reflect the changing needs of the organization, or to alter the physical organization to improve performance.
- **Granting of authorization for data access**.
  - By granting different types of authorization, the database administrator can regulate which parts of the database various users can access. The authorization information is kept in a special system structure that the database system consults whenever someone attempts to access the data in the system.
- **Routine maintenance**. Examples of the database administrator's routine maintenance activities are:
  - Periodically backing up the database, either onto tapes or onto remote servers, to prevent loss of data in case of disasters such as flooding.
  - Ensuring that enough free disk space is available for normal operations, and upgrading disk space as required.
  - Monitoring jobs running on the database and ensuring that performance is not degraded by very expensive tasks submitted by some users.

## 1.9 Components of DBMS and overall Structure of DBMS

A database system is partitioned into modules that deal with each of the responsibilities of the overall system. The functional components of a database

system can be broadly divided into the storage manager and the query processor components.

## 1. Storage Manager:-

A *storage manager* is a program module that provides the interface between the low level data stored in the database and the application programs and queries submitted to the system. The storage manager is responsible for the interaction with the file manager. The raw data are stored on the disk using the file system, which is usually provided by a conventional operating system. The storage manager translates the various DML statements into low-level file-system commands. Thus, the storage manager is responsible for storing, retrieving, and updating data in the database.

The storage manager components include:

- **Authorization and integrity manager**, which tests for the satisfaction of integrity constraints and checks the authority of users to access data.
- **Transaction manager**, which ensures that the database remains in a consistent(correct) state despite system failures, and that concurrent transaction executions proceed without conflicting.
- **File manager**, which manages the allocation of space on disk storage and the data structures used to represent information stored on disk.
- **Buffer manager**, which is responsible for fetching data from disk storage into main memory, and deciding what data to cache in main memory. The buffer manager is a critical part of the database system, since it enables the database to handle data sizes that are much larger than the size of main memory.

- **The storage manager implements several data structures as part of the physical system implementation:**
  - **Data files**, which store the database itself.
  - **Data dictionary**, which stores metadata about the structure of the database, in particular the schema of the database.
  - **Indices**, which provide fast access to data items that hold particular values.

## 2. The Query Processor

The query processor components include

- **DDL interpreter**, which interprets DDL statements and records the definitions in the data dictionary.
- **DML** compiler, which translates DML statements in a query language into an evaluation plan consisting of low-level instructions that the query evaluation engine understands. A query can usually be translated into any of a number of alternative evaluation plans that all give the same result. The DML compiler also performs **query optimization**, that is, it picks the lowest cost evaluation plan from among the alternatives.
- **Query evaluation engine**, which executes low-level instructions generated by the DML compiler.

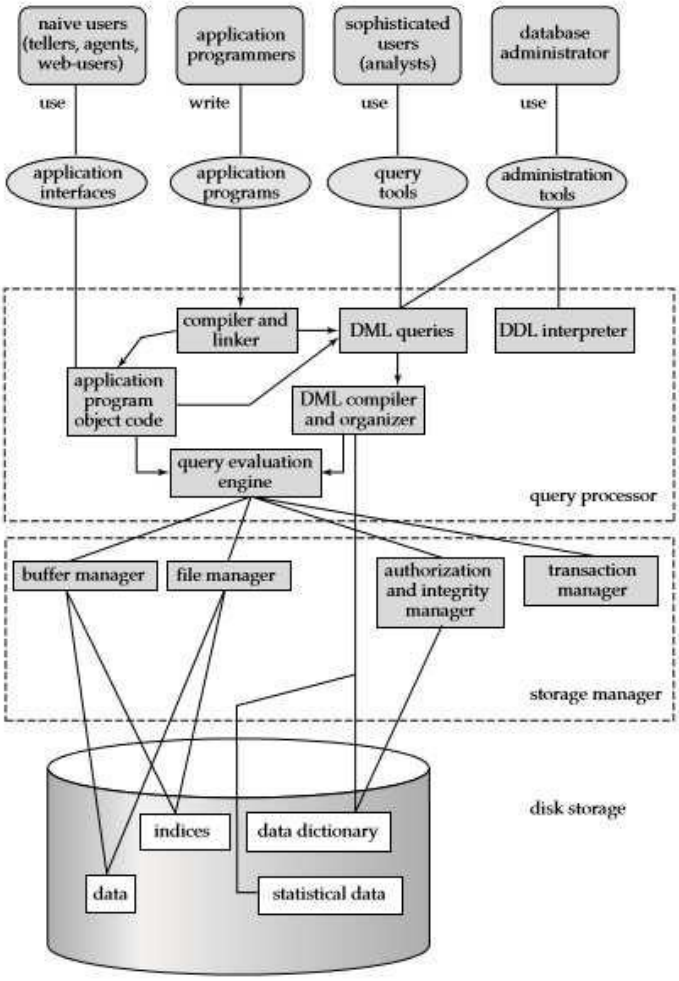



**Figure 1.4** System structure.

## 1.10 Introduction to client server architecture and Two/Three tier Architecture.

Most users of a database system today are not present at the site of the database system, but connect to it through a network. We can therefore differentiate between client machines, on which remote database user's work, and server machines, on which the database system runs. Database applications are usually partitioned into two or three parts, as in Fig- ure 1.5. In a two-tier architecture, the application is partitioned into a component that t resides at the client machine, which invokes database system functionality at the server machine

through query language statements. Application program interface standards like ODBC and JDBC are used for interaction between the client and the server. In contrast, in a three-tier architecture, the client machine acts as merely a front end and does not contain any direct database calls. Instead, the client end communicates with an application server, usually through a forms interface. The application server in turn communicates with a database system to access data. The business logic of the application, which says what actions to carry out under what conditions, is embedded in the application server, instead of being distributed across multiple clients. Three-tier applications are more appropriate for large applications, and for applications that run on the World Wide Web

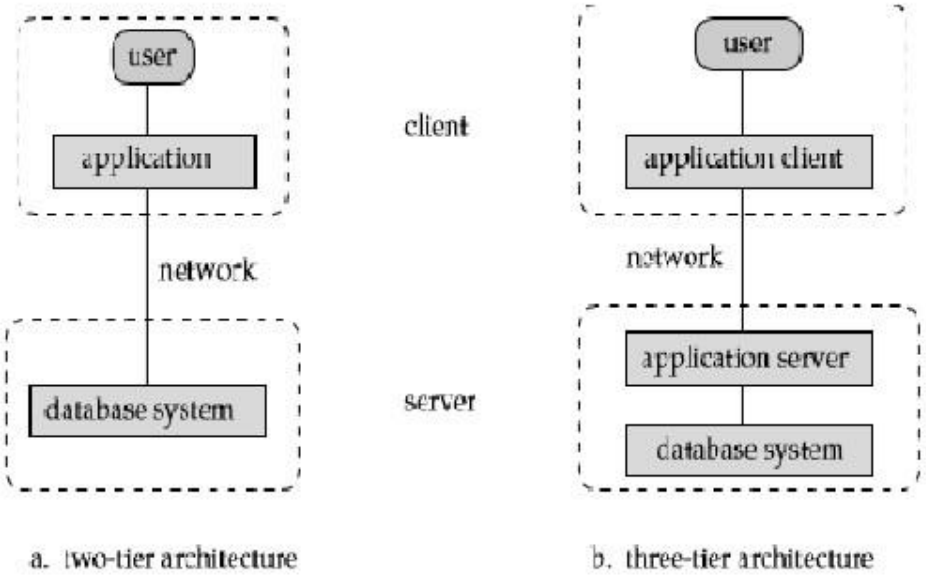



**Figure 1.5** Two-tier and three-tier architectures.

## Codd's 12 rules

**Rule 0:** The Foundation rule:

A relational database management system must manage its stored data using only its relational capabilities. The system must qualify as relational, as a database, and as a management system. For a system to qualify as a relational database management system (RDBMS), that system must use its relational facilities (exclusively) to manage the database.

**Rule 1:** The information rule:

All information in a relational database (including table and column names) is represented in only one way, namely as a value in a table.

**Rule 2:** The guaranteed access rule:

All data must be accessible. This rule is essentially a restatement of the fundamental requirement for primary keys. It says that every individual scalar value in the database must be logically addressable by specifying the name of the containing table, the name of the containing column and the primary key value of the containing row.

**Rule 3:** Systematic treatment of null values:

The DBMS must allow each field to remain null (or empty). Specifically, it must support a representation of "missing information and inapplicable information" that is systematic, distinct from all regular values (for example, "distinct from zero or any other number", in the case of numeric values), and independent of data type. It is also implied that such representations must be manipulated by the DBMS in a systematic way.

**Rule 4:** Active online catalog based on the relational model:

The system must support an online, inline, relational catalog that is accessible to authorized users by means of their regular query language. That is, users must be able to access the database's structure (catalog) using the same query language that they use to access the database's data.

**Rule 5:** The comprehensive data sublanguage rule:

The system must support at least one relational language that

- Has a linear syntax

- Can be used both interactively and within application programs,

Supports data definition operations (including view definitions), data manipulation operations (update as well as retrieval), security and integrity constraints, and transaction management operations (begin, commit, and rollback).

**Rule 6:** The view updating rule:

All views that are theoretically updatable must be updatable by the system.

**Rule 7**: High-level insert, update, and delete:

The system must support set-at-a-time insert, update, and delete operators. This means that data can be retrieved from a relational database in sets constructed of data from multiple rows and/or multiple tables. This rule states that insert, update, and delete operations should be supported for any retrievable set rather than just for a single row in a single table.

**Rule 8:** Physical data independence:

Changes to the physical level (how the data is stored, whether in arrays or linked lists etc.) must not require a change to an application based on the structure.

**Rule 9:** Logical data independence:

Changes to the logical level (tables, columns, rows, and so on) must not require a change to an application based on the structure. Logical data independence is more difficult to achieve than physical data independence.

**Rule 10**: Integrity independence:

Integrity constraints must be specified separately from application programs and stored in the catalog. It must be possible to change such constraints as and when appropriate without unnecessarily affecting existing applications.

**Rule 11:** Distribution independence:

The distribution of portions of the database to various locations should be invisible to users of the database. Existing applications should continue to operate successfully:

- when a distributed version of the DBMS is first introduced; and
- When existing distributed data are redistributed around the system.

**Rule 12:** The no subversion rule:

If the system provides a low-level (record-at-a-time) interface, then that interface cannot be used to subvert the system, for example, bypassing a relational security or integrity constraint.

## Data Warehouses

A database consists of one or more files that need to be stored on a computer. In large organizations, databases are typically not stored on the individual computers of employees but in a central system. This central system typically consists of one or more **computer servers**. A server is a computer system that provides a service over a network. The server is often located in a room with controlled access, so only authorized personnel can get physical access to the server.

In a typical setting, the database files reside on the server, but they can be accessed from many different computers in the organization. As the number and complexity of databases grows, we start referring to them together as a **data warehouse**.

A data warehouse is a collection of databases that work together. A data warehouse makes it possible to integrate data from multiple databases, which can give new insights into the data. The ultimate goal of a database is not just to store data, but to help businesses make decisions based on that data. A data warehouse supports this goal by providing architecture and tools to systematically organize and understand data from multiple databases.

## Distributed DBMS

As databases get larger, it becomes increasingly difficult to keep the entire database in a single physical location. Not only does storage capacity become an issue, there are also security and performance considerations. Consider a company with several offices around the world.

It is possible to create one large, single database at the main office and have all other offices connect to this database. However, every single time an employee needs to work with the database, this employee needs to create a connection over thousands of miles, through numerous network nodes. As long as you are moving relatively small amounts of data around, this does not present a major challenge.

But, what if the database is huge? It is not very efficient to move large amounts of data back and forth over the network. It may be more efficient to have

a **distributed database**. This means that the database consists of multiple, interrelated databases stored at different computer network sites.

To a typical user, the distributed database appears as a centralized database. Behind the scenes, however, parts of that database are located in different places. The typical characteristics of a distributed database management system, or DBMS, are:

- *Multiple computer network sites are connected by a communication system*
- *Data at any site are available to users at other sites*
- *Data at each site are under control of the DBMS*

You have probably used a distributed database without realizing it. For example, you may be using an e-mail account from one of the major service providers. Where exactly do your e-mails reside? Most likely, the company hosting the e-mail service uses several different locations without you knowing it.

The major advantage of distributed databases is that data access and processing is much faster. The major disadvantage is that the database is much more complex to manage. Setting up a distributed database is typically the task of a database administrator with very specialized database skills.

**Data Mining**

**O**nce all the data is stored and organized in databases, what's next? Many day-to-day operations are supported by databases. Queries based on SQL, a database programming language, are used to answer basic questions about data. But, as the collection of data grows in a database, the amount of data can easily become overwhelming. How does an organization get the most out of its data, without getting lost in the details? That's where **data mining** comes in.

Data mining is the process of analyzing data and summarizing it to produce useful information. Data mining uses sophisticated data analysis tools to discover patterns and relationships in large datasets. These tools are much more than basic summaries or queries and use much more complicated algorithms. When data mining is used in business applications, it is also referred to as **business analytics** or **business intelligence**.

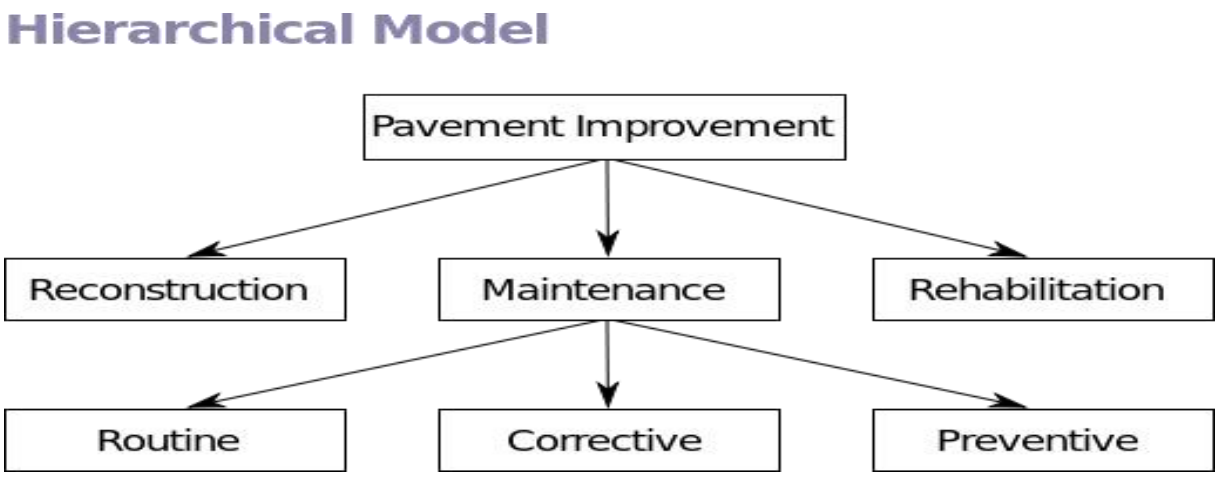
Consider an online retailer that sells a wide variety of products. In a typical day, it may sell thousands of different products to tens of thousands of different customers. How does the company leverage all this data to improve its business? One strategy is to discover which products are often bought together.

# Chapter 2.Relational Data Model and Security and Integrity Specification

**Data Model:-**Underlying the structure of a database is the **data model**: a collection of conceptual tools for describing data, data relationships, data semantics, and consistency constraints.

## 1. Hierarchical database model:-

- This model is recognized as the first database model created by IBM in the 1960s.
- A hierarchical database model is a data model in which the data is organized into a tree-like structure.
- The structure allows representing information using parent/child relationships: each parent can have many children, but each child has only one parent (also known as a 1-to-many relationship).
- In a database an entity type is the equivalent of a table. Each individual record is represented as a row, and each attribute as a column. Entity types are related to each other using 1:N mappings, also known as one-to-many relationships
- Currently the most widely used hierarchical databases are IMS developed by IBM and Windows Registry by Microsoft.
- The hierarchical database model structures data as a tree of records, with each record having one parent record and many children,
- This structure is simple but inflexible because the relationship is confined to a one-to-many relationship




**2. Network model:-**

- The network model is a database model conceived as a flexible way of representing objects and their relationships
- Its distinguishing feature is that the schema, viewed as a graph in which object types are nodes and relationship types are arcs, is not restricted to being a hierarchy
- The network model allows each record to have multiple parent and child records, forming a generalized graph structure
- This property applies at two levels: the schema is a generalized graph of record types connected by relationship types (called "set types" in CODASYL), and the database itself is a generalized graph of record occurrences connected by relationships (CODASYL "sets"). Cycles are permitted at both levels
- It allowed a more natural modeling of relationships between entities
- It failed to become dominant for two main reasons. Firstly, IBM chose to stick to the hierarchical model with semi-network extensions in their established products such as IMS and DL/I. Secondly, it was eventually displaced by the relational model, which offered a higher-level, more declarative interface

**Relational Model: - Basic Concepts Attributes and Domains.**
**Key Concepts: - Candidate key, Primary key, Foreign key and Super key. E-R model, Components of ER Model, Types of attributes, role indicator, weak & strong entity set.**

**Entity;-**

- An **entity** is a "thing" or "object" in the real world that is distinguishable from all other objects.
- For example, each person in an enterprise is an entity
- An entity has a set of properties, and the values for some set of properties may uniquely identify an entity.

**Entity set:-**

- An entity set is a set of entities of the same type that share the same properties, or attributes.
- .The set of all persons who are customers at a given bank, for example ,can be defined as the entity set customer
- Entity sets do not need to be disjoint.
- A person entity may be an employee entity, a customer entity, both, or neither.

**Attribute:-**

- An entity is represented by a set of attributes
- Attributes are descriptive properties possessed by each member of an entity set
- The designation of an attribute for an entity set expresses that the database stores similar information concerning each entity in the entity set; however, each entity may have its own value for each attribute. Possible attributes of the customer entity set are customer-id, customer-name, customer- Street, and customer-city.
- For each attribute, there is a set of permitted values, called the **Domain**, or value set, of that attribute.
- An attribute takes a null value when an entity does not have a value for it. The null value may indicate "not applicable"- that is, that the value does not exist for the entity. For example, one may have no middle name. Null can also designate that an attribute value is unknown. An unknown value may be either missing (the value does

exist, but we do not have that information) or not known (we do not know whether or not the value actually exists).

**An attribute, as used in the E-R model, can be characterized by the following attribute types**

**1. Simple and Composite:-**

- The attributes have been **simple** that is, they are not divided into subparts. e.g. custmore_id cannot divide in to other part.
- **Composite attributes**, on the other hand, can be divided into subparts (that is, other attributes). For example, an attribute name could be structured as a composite attribute consisting of first-name, middle-initial, and last-name.
- **Composite attributes** help us to group together related attributes, making the modeling cleaner.
- That **a composite attribute** may appear as a hierarchy.

**2. Single-valued and multivalued attributes:-**

- The attributes in our examples all have a single value for a particular entity. For instance, the loan-number attribute for a specific loan entity refers to only one loan number. Such attributes are said to be **single valued.**
- An employee may have zero, one, or several phone numbers, and different employees may have different numbers of phones. This type of attribute is said to be **multivalued.**

**3. Derived Attribute:-**

- The value for this type of attribute can be derived from the values of other related attributes or entities.
- Example, suppose that the customer entity set has an attribute age, which indicates the customer's age.
- **Age is a derived attribute**. In this case, date-of-birth may be referred to as a base attribute, or a stored attribute. The value of a derived attribute is not stored, but is computed when required.

**Relation and relationship Set:-**

- **A relationship** is an association among several entities.
- For example, we can define a relationship that associates customer Hayes with loan L-15. This relationship specifies that Hayes is a customer with loan number L-15.

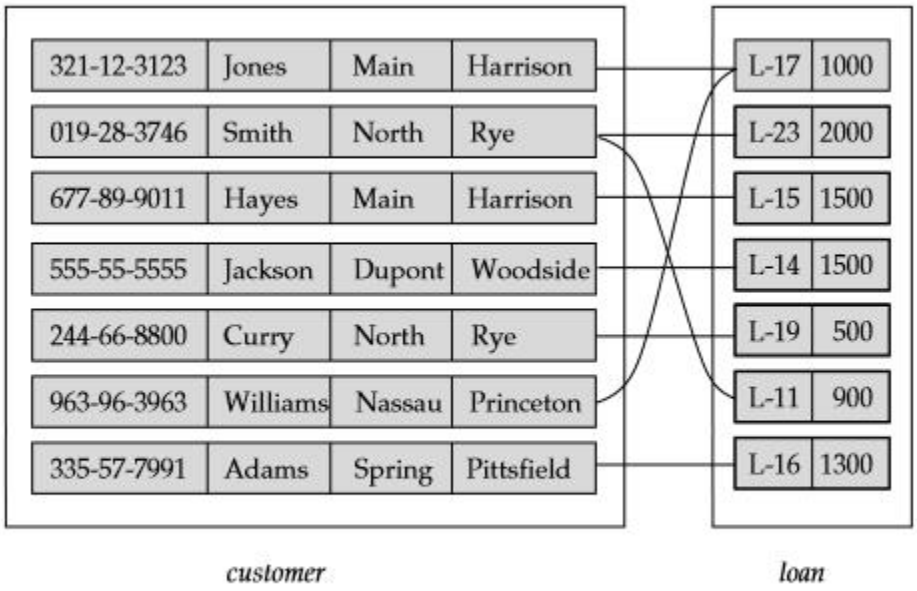



**Figure 2.3** Relationship set *borrower*.

- **A relationship set** is a set of relationships of the same type.
- Formally, it is a mathematical relation on $n \geq 2$ (possibly non distinct) entity sets. If E1,E 2,...,En are entity sets, then a relationship set R is a subset of {(e1,e2,...,en)|e1 ∈ E1,e2 ∈ E2,...,en ∈ En} where (e1,e2,...,en) is a relationship.
- The two entity sets customer and loan in Figure 2.1. We define the relationship set borrower to denote the association between customers and the bank loans that the customers have. Figure 2.3 depicts this association.
- The function that an entity plays in a relationship is called that entity's role
- Since entity sets participating in a relationship set are generally distinct, roles are implicit and are not usually specified

- A relationship may also have attributes called **descriptive attributes.**

**Mapping Cardinalities**

- Mapping cardinalities, or cardinality ratios, express the number of entities to which another entity can be associated via a relationship set.
- Mapping cardinalities are most useful in describing **binary relationship** sets, although they can contribute to the description of relationship sets that involve more than two entity sets. For a binary relationship set R between entity sets A and B, the mapping cardinality must be one of the following:

• One to one. An entity in A is associated with at most one entity in B, and an entity in B is associated with at most one entity in A. (See Figure 2.4a.)

• One to many. An entity in A is associated with any number (zero or more) of entities in B. An entity in B, however, can be associated with at most one entity in A. (See Figure 2.4b.)

• Many to one. An entity in A is associated with at most one entity in B. An entity in B, however, can be associated with any number (zero or more) of entities in A. (See Figure 2.5a.)

• Many to many. An entity in A is associated with any number (zero or more) of entities in B, and an entity in B is associated with any number (zero or more) of entities in A. (See Figure 2.5b.)

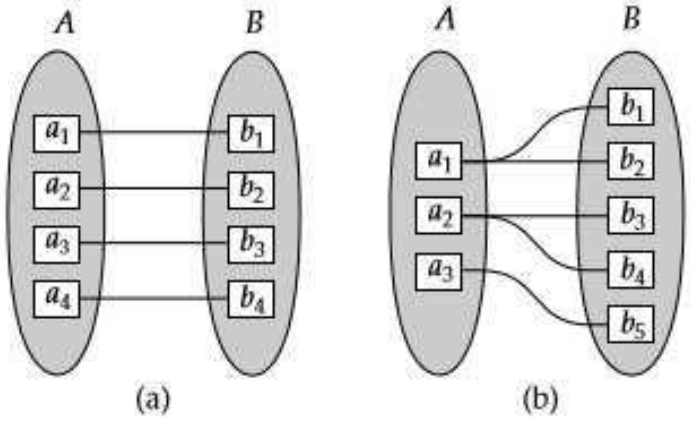



**Figure 2.4** Mapping cardinalities. (a) One to one. (b) One to many.

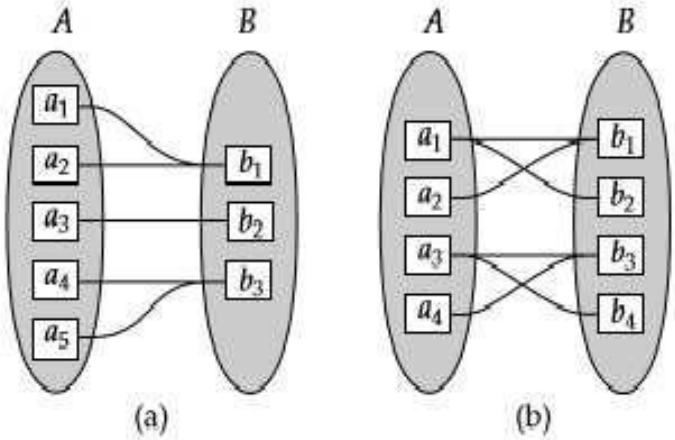



**Figure 2.5** Mapping cardinalities. (a) Many to one. (b) Many to many.

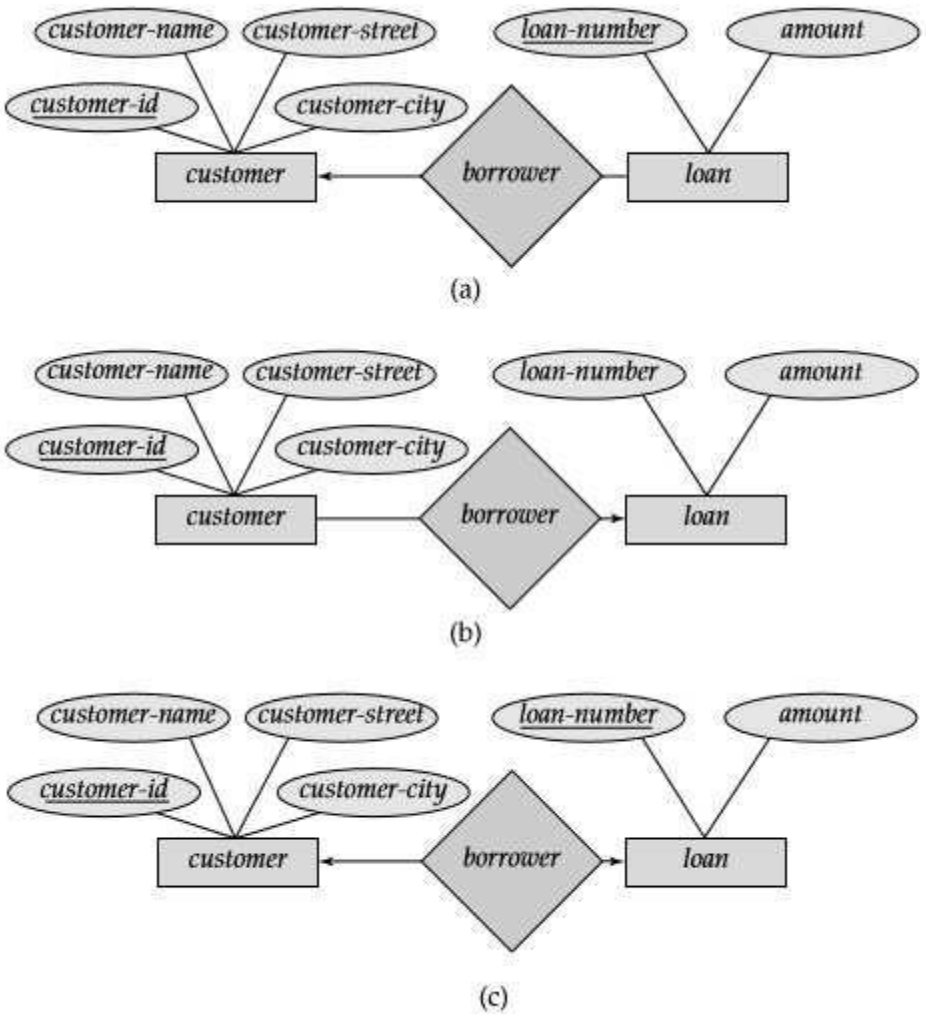



**Figure 2.9** Relationships. (a) one to many. (b) many to one. (c) one-to-one.

## Participation Constraints

The participation of an entity set E in a relationship set R is said to be total if every entity in E participates in at least one relationship in R. If only some entities in E participate in relationships in R, the participation of entity set E in relationship R is said to be partial. For example, we expect every loan entity to be related to at least one customer through the borrower relationship .Therefore the participation of loan in the relationship set borrower is total. In contrast ,an individual can be a bank customer whether or not she has a loan with the bank. Hence, it is possible that only some of the customer entities are related to the loan entity set through the borrower relationship, and the participation of customer in the borrower relationship set is therefore partial.

**Keys: -** A key allows us to identify a set of attributes that suffice to distinguish entities from each other. Keys also help uniquely identify relationships, and thus distinguish relationships from each other.

**Super key:-** A super key is a set of one or more attributes that, taken collectively, allow us to identify uniquely an entity in the entity set.

The combination of customer-name and customer-id is a super key for the entity set customer. The customer-name attribute of customer is not a super key, because several people might have the same name.

**Candidate Key: -** If K is a supper key, then so is any superset of K. We are often interested in super keys for which no proper subset is a super key. Such minimal super keys are called candidate keys.

Both {customer-id} and {customer-name, customer-street}are candidate keys. Although the attributes customer- id and customer-name together can distinguish customer entities, their combination does not form a candidate key, since the attribute customer-id alone is a candidate key.

**Primary key** :- Primary key to denote a candidate key that is chosen by the database designer as the principal means of identifying entities within an entity set.

The primary key should be chosen such that its attributes are never, or very rarely, changed.

Candidate keys must be chosen with care.

E.g. Customer_id.

**Foreign Key:- A** relation schema, say r1, derived from an E-R schema may include among its attributes the primary key of another relation schema, say r2. This attribute is called a **foreign key** from r1, referencing r2. The relation **r1** is also called the **referencing relation** of the foreign key dependency, and r2 is called the **referenced relation** of the foreign key.

**Weak and Strong Entity Sets:-**

- An entity set may not have sufficient attributes to form a primary key .Such an entity set is termed a **weak entity set.** An entity set that has a primary key is termed a **strong entity set.**
- Consider the entity set payment, which has the three attributes: payment-number, payment-date, and payment-amount. Payment numbers are typically sequential numbers, starting from 1, generated separately for each loan. Thus, although each payment entity is distinct, payments for different loans may

share the same payment number. Thus, this entity set does not have a primary key; it is a weak entity set.

- For a weak entity set to be meaningful, it must be associated with another entity set, called the **identifying or owner entity set.**
- Every weak entity must be associated with an identifying entity; that is, the weak entity set is said to be **existence dependent** on the identifying entity set.
- The relationship associating the weak entity set with the identifying entity set is called the **identifying relationship.**
- The discriminator of a weak entity set is a set of attributes that allows this distinction to be made. For example, the **discriminator** of the weak entity set payment is the attribute **payment-number**, since, for each loan, a payment number uniquely identifies one single payment for that loan.
- .The discriminator of a weak entity set is also called the partial key of the entity set.
- The primary key of a weak entity set is formed by the primary key of the identifying entity set, plus the weak entity set's discriminator.

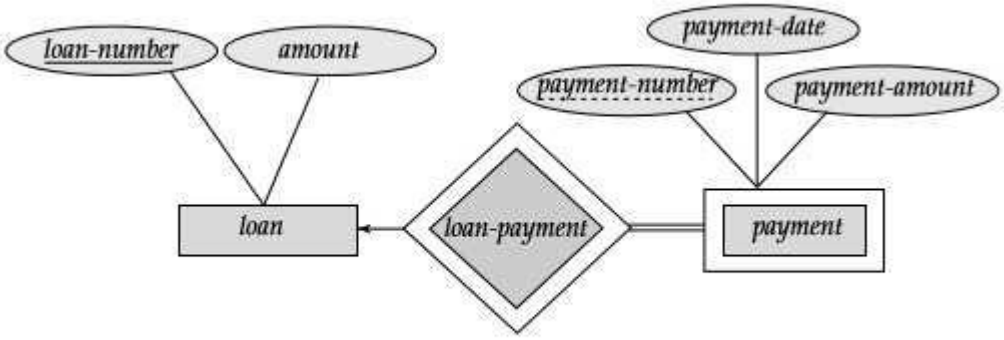



**Figure 2.16** E-R diagram with a weak entity set.

**Specialization:-**

- An entity set may include subgroupings of entities that are distinct in some way from other entities in the set.
- Each of these person types is described by a set of attributes that includes all the attributes of entity set person plus possibly additional attributes
- The process of designating subgroupings within an entity set is called specialization.

- The specialization of person allows us to distinguish among persons according to whether they are employees or customers.
- For example, customer entities may be described further by the attribute customer-id, whereas employee entities may be described further by the attributes employee-id and salary.
- Specialization is depicted by a triangle component labeled ISA.
- The label ISA stands for "is a".
- .The ISA relationship may also be referred to as a **superclass-subclass** relationship.

**Generalization:-**

- The refinement from an initial entity set into successive levels of entity subgroupings represents a top-down design process in which distinctions are made explicit. The design process may also proceed in a bottom-up manner, in which multiple entity sets are synthesized into a higher-level entity set on the basis of common features.
- The database designer may have first identified a customer entity set with the attributes name, street, city, and customer-id, and an employee entity set with the attributes name, street, city, employee-id, and salary.
- There are similarities between the customer entity set and the employee entity set in the sense that they have several attributes in common.
- This commonality can be expressed by **generalization**, which is a containment relationship that exists between a higher-level entity set and one or more lower-level entity sets.
- Higher- and lower-level entity sets also may be designated by the terms superclass and subclass, respectively.
- **The person entity set is the superclass of the customer and employee subclasses.**

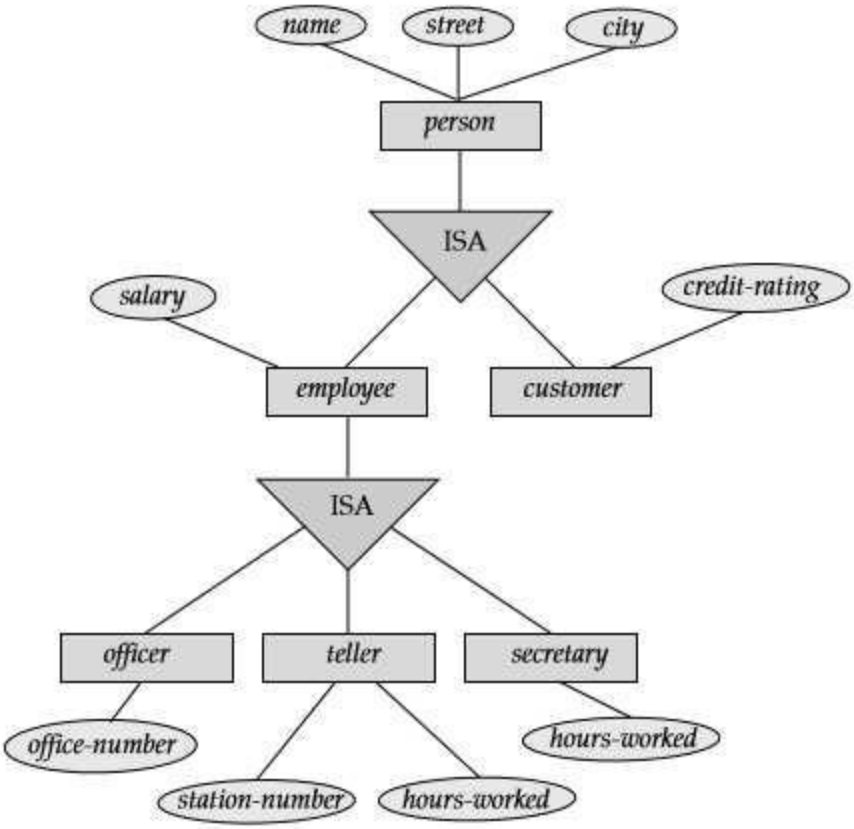



**Figure 2.17** Specialization and generalization.

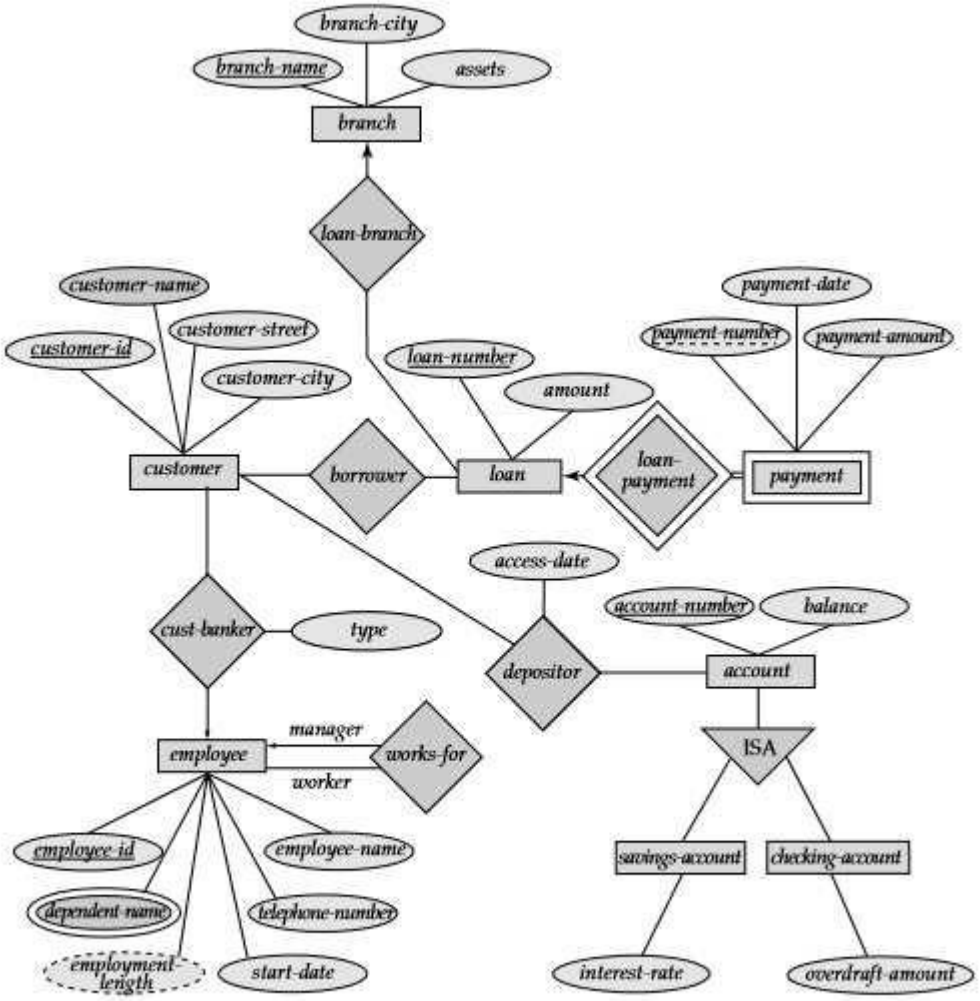
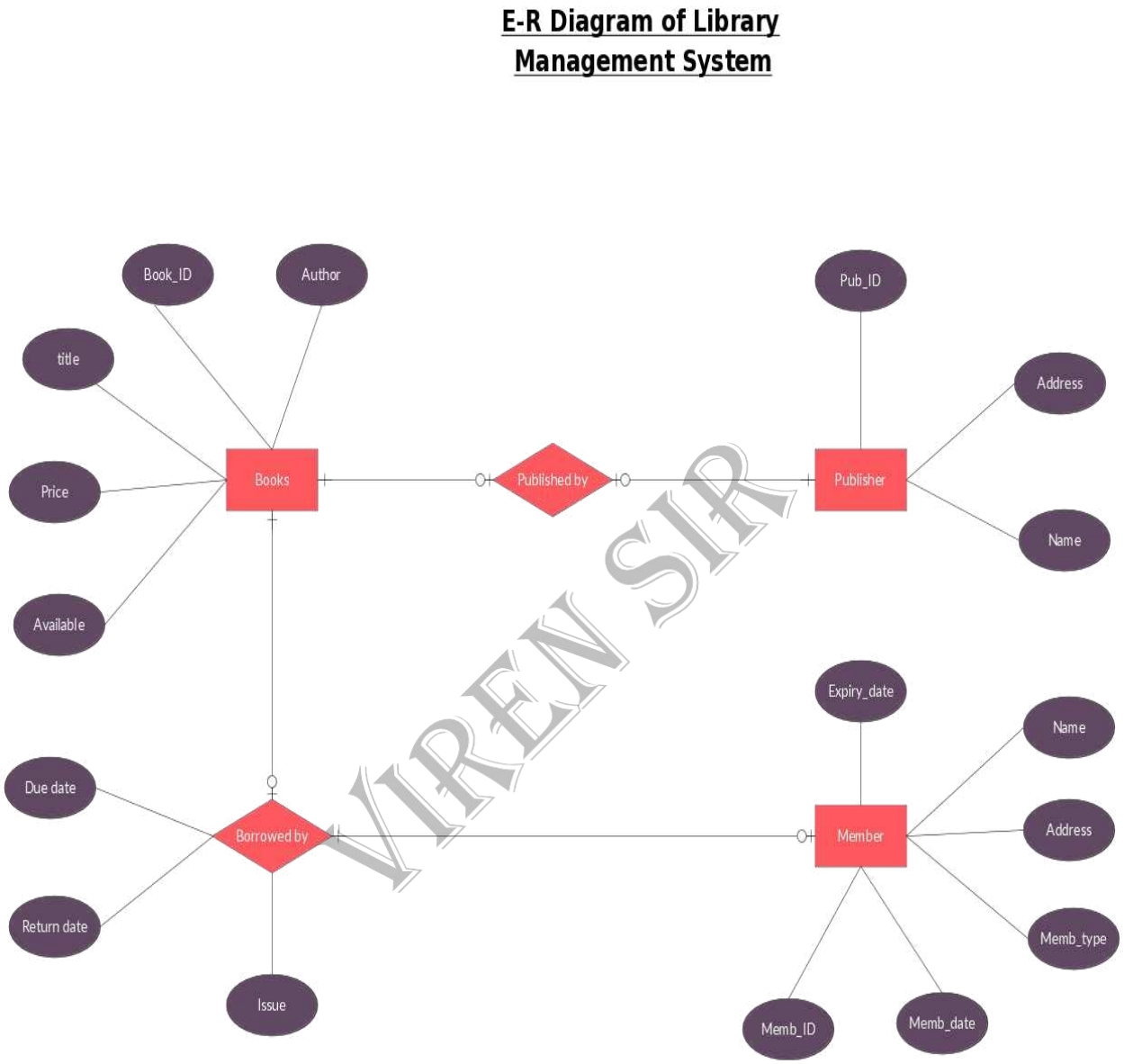
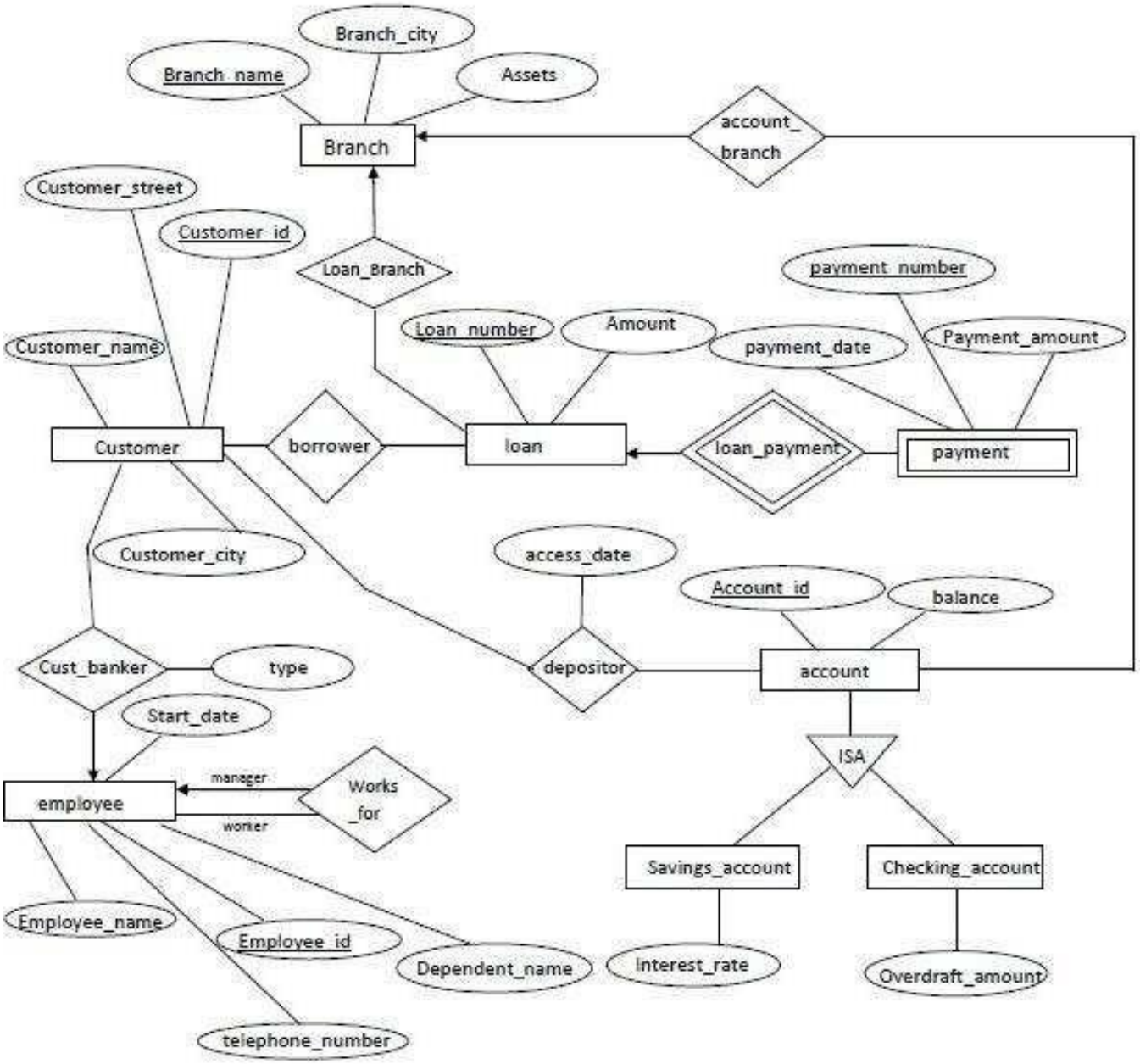



**Figure 2.22** E-R diagram for a banking enterprise.

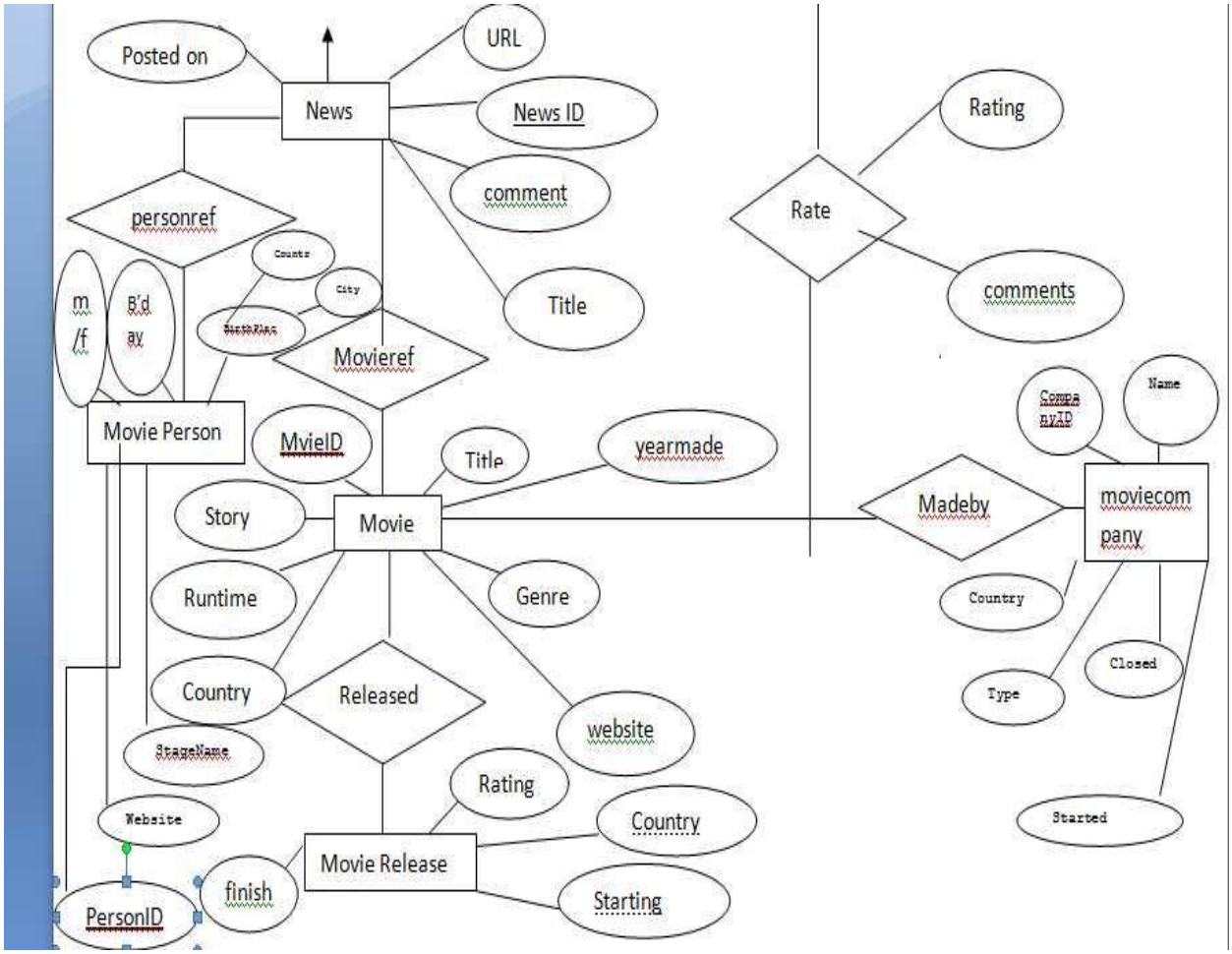
## Some E-R Diagram



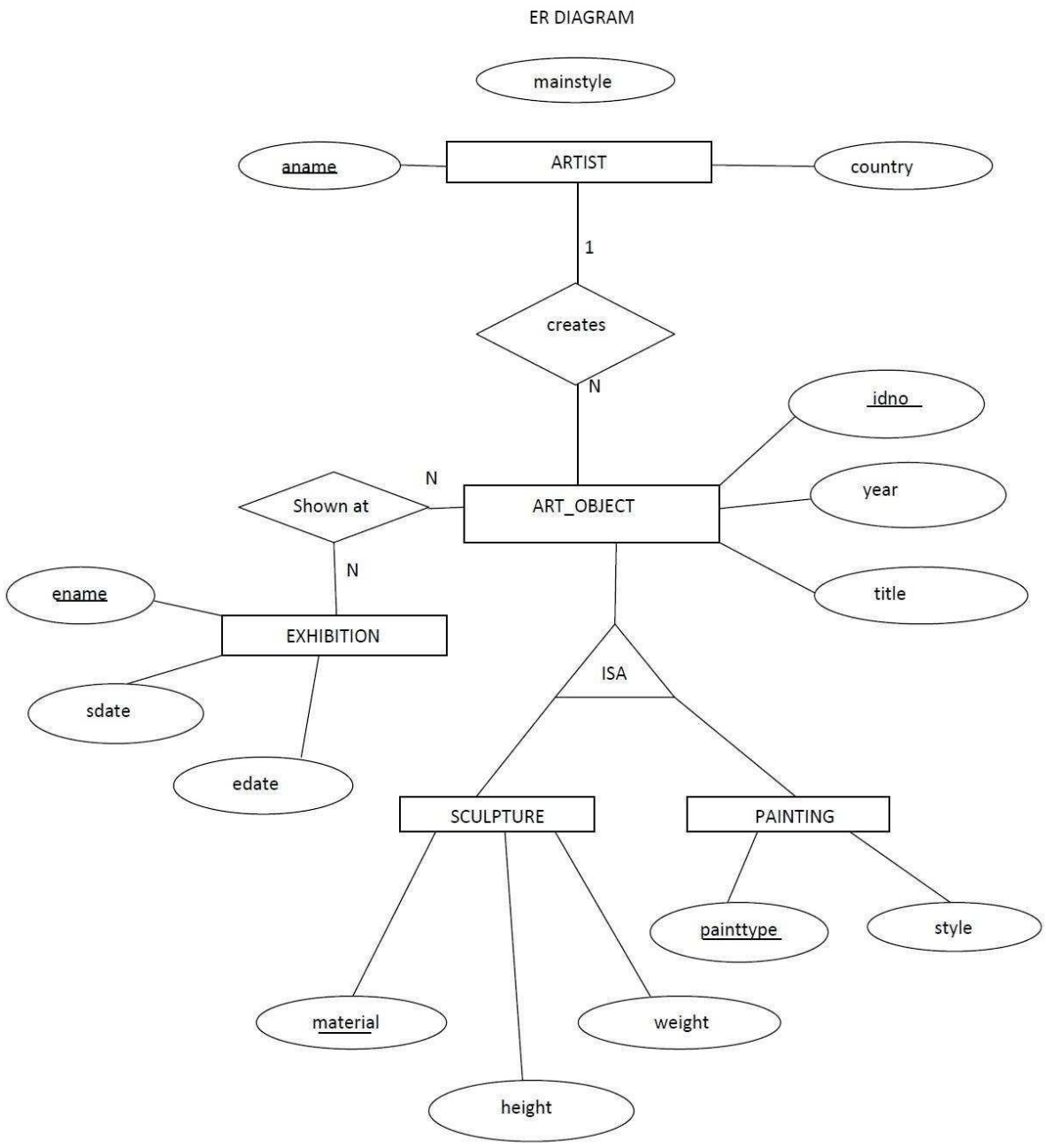
## E-R Diagram Bank Enterprise



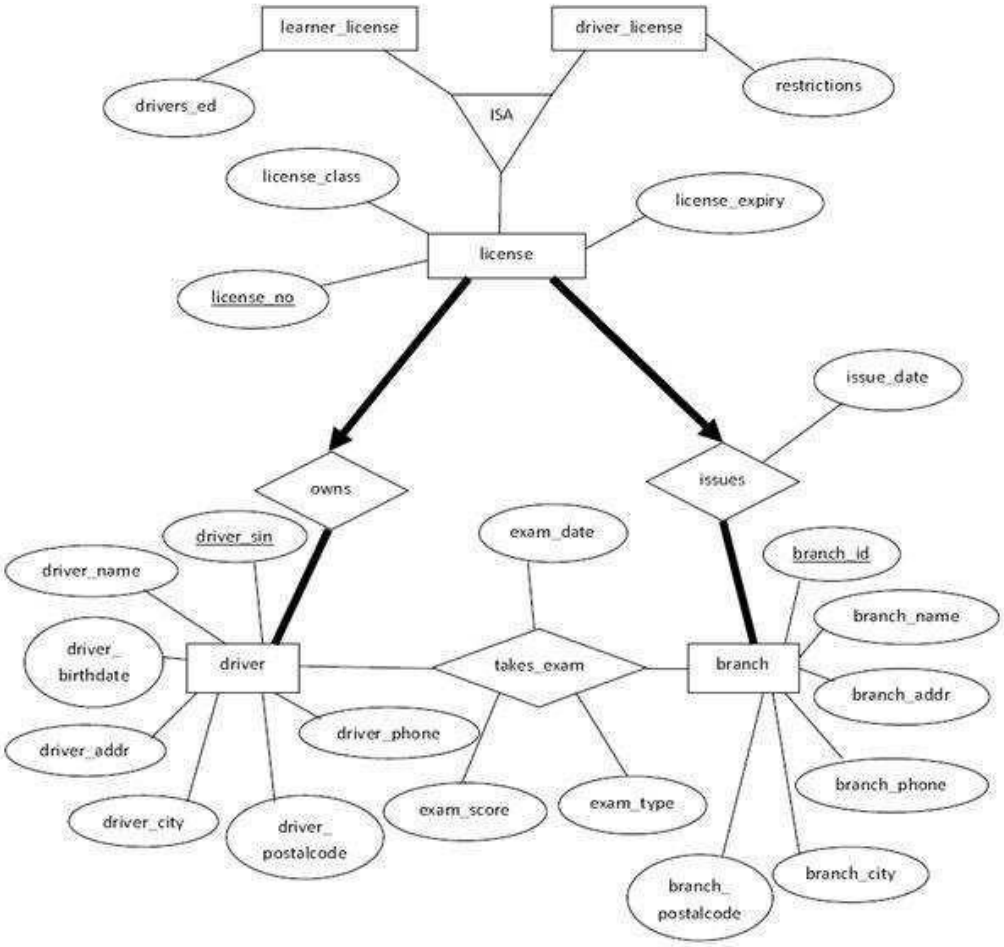
## E-R diagram of Movie Management System



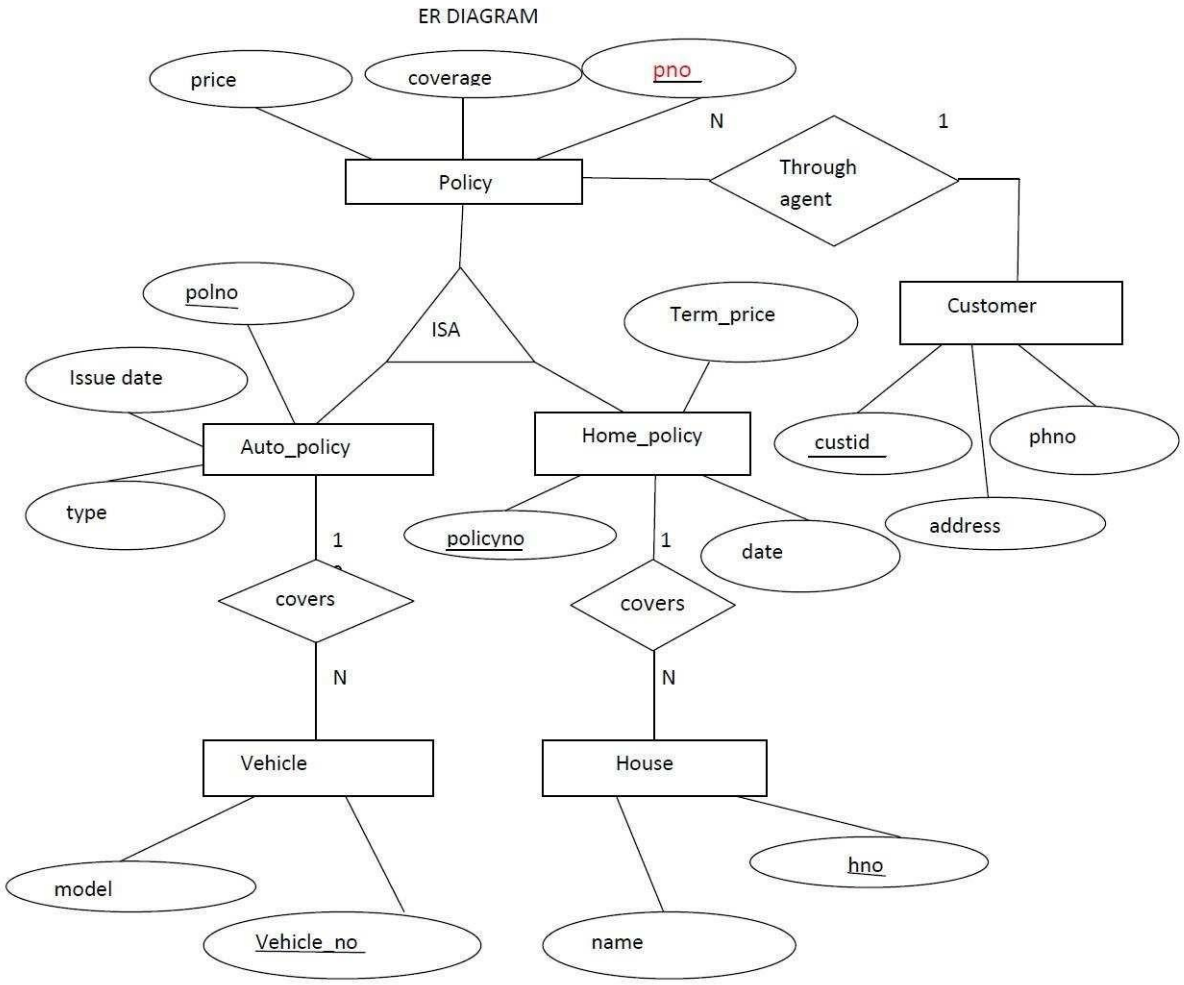
# E-R diagram of Museum Management System



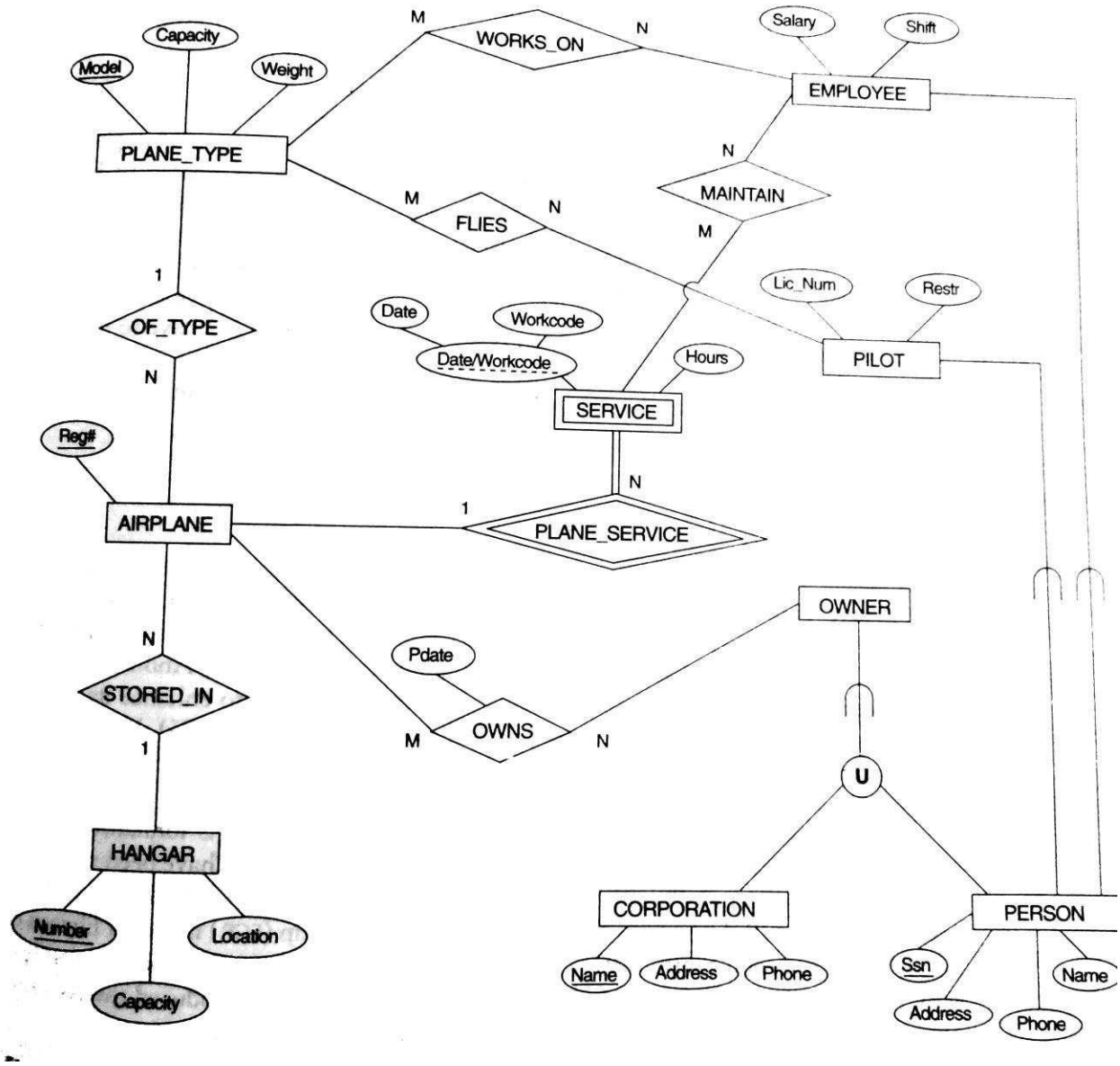
## E-R diagram of License Management System



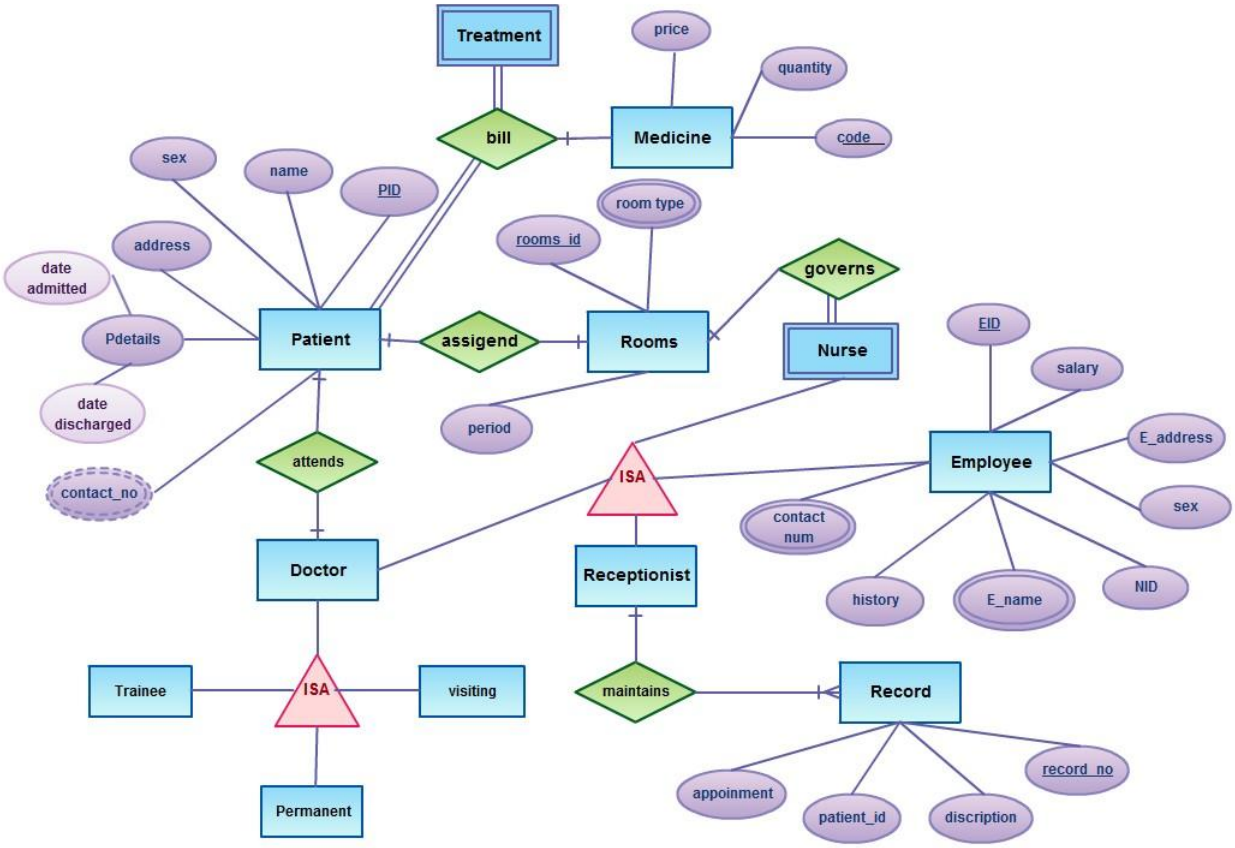
# E-R diagram of Insurance Management System



## E-R diagram of Air Port Management System



E-R Diagram for Hospital Management System




## Relational Algebra and Relational Calculus:-

## Relational Algebra:-

The relational algebra is a procedural query language. It consists of a set of operations that take one or two relations as input and produce a new relation as their result .The fundamental operations in the relational algebra are select, project, union, set difference, Cartesian product, and rename.

### 1. The Select Operation:-

- The select operation selects tuples that satisfy a given predicate.
- The lowercase Greek letter sigma (σ) to denote selection
- The argument relation is in parentheses after the σ.
- Example:- select those tuples of the loan relation where the branch is "Perryridge,"
- σ branch-name ="Perryridge" (loan)

find those tuples pertaining to loans of more than $1200 made by the Perryridge branch, we write

σ branch-name =“Perryridge”A amount>1200 (loan)

| loan-number | branch-name | amount |
|---|---|---|
| L-15 | Perryridge | 1500 |
| L-16 | Perryridge | 1300 |

**Figure 3.10** Result of $\sigma_{branch\text{-}name = \text{"Perryridge"}}(loan)$.

2. **The Project Operation;-**
   - Suppose we want to list all loan numbers and the amount of the loans, but do not care about the branch name.
   - The project operation allows us to produce this relation.
   - The project operation is a unary operation that returns its argument relation, with certain attributes left out.
   - Since a relation is a set, any duplicate rows are eliminated. Projection is denoted by the upper case Greek letter pi (Π).
   - Π loan-number, amount(loan)

**Composition of Relational Operations:-**

**“Find those customers who live in Harrison.” We write:**

**Π customer-name (σ customer-city =“Harrison” (customer))**

Notice that, instead of giving the name of a relation as the argument of the projection operation, we give an expression that evaluates to a relation.

| loan-number | amount |
|---|---|
| L-11 | 900 |
| L-14 | 1500 |
| L-15 | 1500 |
| L-16 | 1300 |
| L-17 | 1000 |
| L-23 | 2000 |
| L-93 | 500 |

**Figure 3.11** Loan number and the amount of the loan.

3. **The Union Operation:-**
   - Find the names of all bank customers who have either an account or a loan or both.

- Note that the customer relation does not contain the information, since a customer does not need to have either an account or a loan at the bank.
- To answer this query, we need the information in the depositor relation and in the borrower relation (Figure 3.7).We know how to find the names of all customers with a loan in the bank:
    - Π customer-name (borrower)
- We also know how to find the names of all customers with an account in the bank:
    - Π customer-name (depositor)
- To answer the query, we need the union of these two sets; that is, we need all customer names that appear in either or both of the two relations. We find these data by the binary operation union, denoted, as in set theory, by ∪.
- So the expression needed is

  Π customer-name (borrower) ∪ Π customer-name (depositor)

| *customer-name* |
|---|
| Adams |
| Curry |
| Hayes |
| Jackson |
| Jones |
| Smith |
| Williams |
| Lindsay |
| Johnson |
| Turner |

**Figure 3.12** Names of all customers who have either a loan or an account.

Therefore, for a union operation r s to be valid, we require that two conditions hold:

1. The relations r and s must be of the same arity. That is, they must have the same number of attributes.

2. The domains of the $i^{th}$ attribute of r and the $i^{th}$ attribute of s must be the same, for all i.

Note that r and s can be, in general, temporary relations that are the result of relational- algebra expressions.

## 4. The Set Difference Operation

- The set-difference operation, denoted by −, allows us to find tuples that are in one relation but are not in another.
- The expression r −s produces a relation containing those tuples in r but not in s.
- We can find all customers of the bank who have an account but not a loan by writing

  Π customer-name (depositor)−Π customer-name (borrower)

## 5. The Cartesian-Product Operation:-

- The Cartesian-product operation, denoted by a cross (×), allows us to combine information from any two relations. We write the Cartesian product of relations r1 and r2 as r1 × r2.
- The same attribute name may appear in both r1 and r2, we need to devise a naming schema to distinguish between these attributes.
- For example, the relation schema for r = borrower ×loan is (borrower. Customer-name,borrower.loan-number,loan.loan-number, loan.branch-name, loan.amount)
- For those attributes that appear in only one of the two schemas, we shall usually drop the relation-name prefix. This simplification does not lead to any ambiguity. We can then write the relation schema for r as (customer-name, borrower. Loan-number,loan.loan-number, branch-name, amount)
- Suppose that we want to find the names of all customers who have a loan at the Perryridge branch.

  σ branch-name =“Perryridge”(borrower × loan)

| customer-name | borrower. loan-number | loan. loan-number | branch-name | amount |
|---|---|---|---|---|
| Adams | L-16 | L-15 | Perryridge | 1500 |
| Adams | L-16 | L-16 | Perryridge | 1300 |
| Curry | L-93 | L-15 | Perryridge | 1500 |
| Curry | L-93 | L-16 | Perryridge | 1300 |
| Hayes | L-15 | L-15 | Perryridge | 1500 |
| Hayes | L-15 | L-16 | Perryridge | 1300 |
| Jackson | L-14 | L-15 | Perryridge | 1500 |
| Jackson | L-14 | L-16 | Perryridge | 1300 |
| Jones | L-17 | L-15 | Perryridge | 1500 |
| Jones | L-17 | L-16 | Perryridge | 1300 |
| Smith | L-11 | L-15 | Perryridge | 1500 |
| Smith | L-11 | L-16 | Perryridge | 1300 |
| Smith | L-23 | L-15 | Perryridge | 1500 |
| Smith | L-23 | L-16 | Perryridge | 1300 |
| Williams | L-17 | L-15 | Perryridge | 1500 |
| Williams | L-17 | L-16 | Perryridge | 1300 |

**Figure 3.15** Result of $\sigma_{branch\text{-}name = \text{"Perryridge"}}(borrower \times loan)$.

## 6.The Rename Operation:-

- The rename operator, denoted by the lower case Greek letter rho (ρ).
- Given a relational-algebra expression E, the expression
  - ρx (E)
- Returns the result of expression E under the name x.

## 7. The Set-Intersection Operation:-

- The first additional-relational algebra operation that we shall define is set intersection (∩).
- Suppose that we wish to find all customers who have both a loan and an account.
- Using set intersection, we can write

  Π customer-name (borrower) ∩ Π customer-name (depositor)

## 8. The Natural join:-

- It is often desirable to simplify certain queries that require a Cartesian product.

- Usually, a query that involves a Cartesian product includes a selection operation on the result of the Cartesian product.
- Consider the query "Find the names of all customers who have a loan at the bank, along with the loan number and the loan amount
- Π customer-name, loan.loan-number, amount (σ borrower.loan-number =loan.loan-number (borrower × loan))
- It is denoted by the "join" |X| symbol.
- By using the natural join as follows:

  Π customer-name, loan-number, amount (borrower |X| loan)

**Database Design: Relational database Design, Functional dependencies, Normalization based on functional dependencies, Normal forms: 1NF, 2NF, 3NF, BCNF. Normalization based on multivalued dependencies, Normalization based on Join dependencies.**

**Functional Dependencies**

- Functional dependencies play a key role in differentiating good database designs from bad database designs. A **functional dependency** is a type of constraint that is a generalization of the notion of *key*.

**Basic Concepts**

- Functional dependencies are constraints on the set of legal relations. They allow us to express facts about the enterprise that we are modeling with our database.
- We defined the notion of a *superkey* as follows. Let $R$ be a relation schema. A subset $K$ of $R$ is a **superkey** of $R$ if, in any legal relation $r(R)$, for all pairs $t1$ and $t2$ of tuples in $r$ such that $t1 \_= t2$, then $t1[K] \_= t2[K]$. That is, no two tuples in any legal relation $r(R)$ may have the same value on attribute set $K$.
- The notion of functional dependency generalizes the notion of super key. Consider a relation schema $R$, and let $\alpha \subseteq R$ and $\beta \subseteq R$. The **functional dependency**

$$\alpha \rightarrow \beta$$

Holds on schema $R$ if, in any legal relation $r(R)$, for all pairs of tuples $t1$ and $t2$ in $r$ such that $t1[\alpha] = t2[\alpha]$, it is also the case that $t1[\beta] = t2[\beta]$. Using the functional-dependency notation, we say that $K$ is a superkey of $R$ if $K \rightarrow R$. That is, $K$ is a superkey if, whenever $t1[K] = t2[K]$, it is also the case that $t1[R] = t2[R]$ (that is, $t1 = t2$).

Functional dependencies allow us to express constraints that we cannot express with super keys. Consider the schema

*Loan-info-schema* = (*loan-number, branch-name, customer-name, amount*)

which is simplification of the *Lending-schema* that we saw earlier. The set of functional dependencies that we expect to hold on this relation schema is

*loan-number* →*amount*

*loan-number* →*branch-name*

We would not, however, expect the functional dependency

*loan-number* →*customer-name*

to hold, since, in general, a given loan can be made to more than one customer (for example, to both members of a husband–wife pair).

We shall use functional dependencies in two ways:

**1.** To test relations to see whether they are legal under a given set of functional dependencies. If a relation *r* is legal under a set *F* of functional dependencies, we say that *r* **satisfies** *F*.

**2.** To specify constraints on the set of legal relations. We shall thus concern ourselves with *only* those relations that satisfy a given set of functional dependencies.

If we wish to constrain ourselves to relations on schema *R* that satisfy a set *F* of functional dependencies, we say that *F* **holds** on *R*.

## NEED OF NORMALIZATION

- **Normalization** can be defined as process of decomposition of database tables to avoid the data redundancy.
- The normalization is an important process that removes the repeating data from database and also used to remove the inconsistency. How to avoid the data redundancy and update anomalies.
- **Insert anomaly** due to lack of that is all the data available for insertion such that null values in keys should be avoided. This kind anomaly and seriously damage a database.
- **Update anomaly** due to data redundancy that is multiple occurrences of same values in column.
- **Deletion anomaly** it leads to loss of data for rows that are not stored else square it could result in loss of vital data.

**First Normal Form (1NF):** A relation R is said to be in first normal form (1NF) if the domain of all attributes of R are atomic.

**OR**

A table is in the first normal form if it contains no repeating elements groups.

Example: Supplier(sno,sname,location,pno,qty)

| SNO | SNAME | LOCATION | PNO | QTY |
|---|---|---|---|---|
| S1 | Abc | Mumbai | P1 | 200 |
| S2 | Pqr | Pune | P2 | 300 |
| S3 | Lmn | Delhi | P1 | 400 |

The above relation is in 1NF as all the domains are having atomic value. But it is not in 2NF.

**Second Normal Form (2NF):** A relation is said to be in the second normal form if it is in first normal form and all the non key attributes are fully functionally dependent on the primary key.

Example: In the above relation NAME, LOCATION depends on SNO and QTY on (SNO, PNO) so the table can be split up into two tables as Supplier(SNO,SNAME,LOCATION) and SP(SNO,PNO,QTY) and now both the tables are in second normal form.

Supplier

| SNO | SNAME | LOCATION |
|---|---|---|
| S1 | Abc | Mumbai |
| S2 | Pqr | Pune |
| S3 | Lmn | Delhi |

SP

| SNO | PNO | QTY |
|---|---|---|
| S1 | P1 | 200 |
| S2 | P2 | 300 |
| S3 | P1 | 400 |

**BCNF:-**

**Definition**

One of the more desirable normal forms that we can obtain is **Boyce-Codd normal form** (**BCNF**). A relation schema *R* is in BCNF with respect

to a set $F$ of functional dependencies if, for all functional dependencies in $F^+$ of the form $\alpha \rightarrow \beta$, where $\alpha \subseteq R$ and $\beta \subseteq R$, at least one of the following holds:

- $\alpha \rightarrow \beta$ is a trivial functional dependency (that is, $\beta \subseteq \alpha$).
- $\alpha$ is a superkey for schema $R$.

A database design is in BCNF if each member of the set of relation schemas that constitutes the design is in BCNF.

As an illustration, consider the following relation schemas and their respective functional dependencies:

- *Customer-schema* = (*customer-name*, *customer-street*, *customer-city*)
  *Customer-name* → *customer-street customer-city*
- *Branch-schema* = (*branch-name*, *assets*, *branch-city*)
  *Branch-name* → *assets branch-city*
- *Loan-info-schema* = (*branch-name*, *customer-name*, *loan-number*, *amount*)
  *loan-number* → *amount branch-name*

We claim that *Customer-schema* is in BCNF. We note that a candidate key for the schema is *customer-name*. The only nontrivial functional dependencies that hold on *Customer-schema* have *customer-name* on the left side of the arrow. Since *customer-name* is a candidate key, functional dependencies with *customer-name* on the left side do not violate the definition of BCNF. Similarly, it can be shown easily that the relation schema *Branch-schema* is in BCNF.

The schema *Loan-info-schema*, however, **is *not* in BCNF**.

## Third Normal Form:-

### Definition

BCNF requires that all nontrivial dependencies be of the form $\alpha \rightarrow \beta$, where $\alpha$ is a Super key. 3NF relaxes this constraint slightly by allowing nontrivial functional dependencies whose left side is not a super key.

A relation schema $R$ is in **third normal form (3NF)** with respect to a set $F$ of functional dependencies if, for all functional dependencies in $F^+$ of the form $\alpha \rightarrow \beta$, where $\alpha \subseteq R$ and $\beta \subseteq R$, at least one of the following holds:

- $\alpha \rightarrow \beta$ is a trivial functional dependency.
- $\alpha$ is a super key for $R$.
- Each attribute $A$ in $\beta - \alpha$ is contained in a candidate key for $R$.

Note that the third condition above does not say that a single candidate key should contain all the attributes in $\beta - \alpha$; each attribute $A$ in $\beta - \alpha$ may be contained in a *different* candidate key.

The first two alternatives are the same as the two alternatives in the definition of BCNF. The third alternative of the 3NF definition seems rather unintuitive

Observe that any schema that satisfies BCNF also satisfies 3NF, since each of its functional dependencies would satisfy one of the first two alternatives. BCNF is therefore a more restrictive constraint than is 3NF.

The definition of 3NF allows certain functional dependencies that are not allowed in BCNF. A dependency $\alpha \rightarrow \beta$ that satisfies only the third alternative of the 3NF definition is not allowed in BCNF, but is allowed in 3NF.

**Multivalued dependencies** occur when the presence of one or more rows in a table implies the presence of one or more other rows in that same table.

**OR**

**A multivalued dependency (MVD)** $X$ —>> $Y$ specified on relation schema $R$, where $X$ and $Y$ are both subsets of $R$, specifies the following constraint on any relation state $r$ of $R$: If two tuples $t1$ and $t2$ exist in $r$ such that $t1[X] = t2[X]$, then two tuples $t3$ and $t4$ should also exist in $r$ with the following properties, where we use $Z$ to denote ($R$ 2 ($X$ υ $Y$)):

- $t3[X] = t4[X] = t1[X] = t2[X]$.
- $t3[Y] = t1[Y]$ and $t4[Y] = t2[Y]$.
- $t3[Z] = t2[Z]$ and $t4[Z] = t1[Z]$.

**Example 1:** For example, imagine a car company that manufactures many models of car, but always makes both red and blue colors of each model. If you have a table that contains the model name, color and year of each car the company manufactures, there is a multivalued dependency in that table. If there is a row for a certain model name and year in blue, there must also be a similar row corresponding to the red version of that same car.

**Example 2:**

| Subject | Text | Teacher |
|---|---|---|
| MATHEMATICS | ALGEBRA | Mr. SINGH |
| MATHEMATICS | GEOMETRY | Mr. PATIL |
| MATHEMATICS | ALGEBRA | Mr. PATIL |
| MATHEMATICS | GEOMETRY | Mr. SINGH |
| COMPUTER | DATABASE MANAGEMENT | Mrs. DIXIT |
| COMPUTER | VB.NET | Mrs. DIXIT |

In the above relation Text and Teacher are multivalued dependent on Subject. There are two multivalued dependencies in this.

{Subject} {Text} and {Subject} {Teacher}

**Integrity Constraints:-**

1. **Domain Integrity Constraints:-**It is used to maintain the value according to the user specification.
   a. **Not Null:-**By default all columns in table allow null values. By enforcing Not Null constraint on column, it not allows any null values.
   **Syntax:**
   Create table<table name>(column_name1 datatype(size),column_name2 datatype(size) not null,.....,column_namen datatype(size));
   **Example:**
   Create table emp (eno number(10),ename varchar2(20) not null);
   b. **Check: -** A common use of the **check** constraint is to ensure that attribute values satisfy specified conditions, in effect creating a powerful type system.
   **Syntax:**

**Create table** <*TableName*> (column_name1 datatype(size) ,column_name2 datatype(size) not null,.....,column_namen datatype(size **) check** (*Column2*>= 0))

**Example:-**
**Create table** *branch*(*branch-name* **char**(15),*branch-city* **char**(30),*assets* **integer**, **primary key** (*branch-name*),**check** (*assets* >= 0))

2. **Entity Integrity Constraints :-**
   a. **Primary key** (*Aj*1,*Aj*2, . . .,*Ajm*): The **primary key** specification says that attributes *Aj*1,*Aj*2, . . .,*Ajm* form the primary key for the relation. The primary key attributes are required to **be *non-null* and *unique***; that is, no tuple can have a null value for a primary key attribute, and no two tuples in the relation can be equal on all the primary-key attributes.1 Although the primary key specification is optional, it is generally a good idea to specify a primary key for each relation.
   **Example:-**

**Create table** *customer* (*customer-name* **char** (20) **primary key**, *customer-street* **char** (30),*Customer-city* **char** (30))

**b. Unique Key:-** **unique** (*Aj*1,*Aj*2, . . . , *Ajm*)

The **unique** specification says that attributes *Aj*1,*Aj*2, . . .,*Ajm* form a candidate key; that is, no two tuples in the relation can be equal on all the primary-key attributes. However, candidate key attributes are permitted to be null unless they have explicitly been declared to be **not null**. Recall that a null value does not equal any other value.

**Example:-**

**Create table** *customer* (*customer-name* **char** (20) **not null**, cust-id varchar2(30) **primary key**, *customer-street* **char** (30),Mobile-no number(13) **Unique),***Customer-city* **char** (30))

*3* . **Referential Integrity**

That a value that appears in one relation for a given set of attributes also appears for a certain set of attributes in another relation. This condition is called **referential integrity**.

**create table** *customer*
(*customer-name* **char**(20),
*customer-street* **char**(30),
*customer-city* **char**(30),
**primary key** (*customer-name*))

**create table** *branch*
(*branch-name* **char**(15),
*branch-city* **char**(30),
*assets* **integer**,
**primary key** (*branch-name*),
**check** (*assets* >= 0))

**create table** *account*
(*account-number* **char**(10),
*branch-name* **char**(15),
*balance* **integer**,
**primary key** (*account-number*),
**foreign key** (*branch-name*) **references** *branch*,
**check** (*balance* >= 0))

**create table** *depositor*
(*customer-name* **char**(20),
*account-number* **char**(10),
**primary key** (*customer-name*, *account-number*),
**foreign key** (*customer-name*) **references** *customer*,
**foreign key** (*account-number*) **references** *account*)

4 **On delete Cascade:-**

**create table** *account*
( . . .
**foreign key** (*branch-name*) **references** *branch*
**on delete cascade**
**on update cascade**,
. . . )

Because of the clause **on delete cascade** associated with the foreign-keydeclaration, if a delete of a tuple in *branch* results in this referential-integrity constraint being violated,

the system does not reject the delete. Instead, the delete "cascades" to the *account* relation, deleting the tuple that refers to the branch that was deleted

Use the ON DELETE CASCADE option if you want rows deleted in the child table when corresponding rows are deleted in the parent table. If you do not specify cascading deletes, the default behavior of the database server prevents you from deleting data in a table if other tables reference it.

If you specify this option, when you delete a row in the parent table, the database server also deletes any rows associated with that row (foreign keys) in a child table. The advantage of the ON DELETE CASCADE option is that it allows you to reduce the quantity of SQL statements needed to perform delete actions.

**Data Securities**

**Database security** concerns the use of a broad range of information security controls to protect databases (potentially including the data, the database applications or stored functions, the database systems, the database servers and the

associated network links) against compromises of their confidentiality, integrity and availability. It involves various types or categories of controls, such as technical, procedural/administrative and physical. *Database security* is a specialist topic within the broader realms of computer security, information security and risk management.

Security risks to database systems include, for example:

- Unauthorized or unintended activity or misuse by authorized database users, database administrators, or network/systems managers, or by unauthorized users or hackers (e.g. inappropriate access to sensitive data, metadata or functions within databases, or inappropriate changes to the database programs, structures or security configurations);
- Malware infections causing incidents such as unauthorized access, leakage or disclosure of personal or proprietary data, deletion of or damage to the data or programs, interruption or denial of authorized access to the database, attacks on other systems and the unanticipated failure of database services;
- Overloads, performance constraints and capacity issues resulting in the inability of authorized users to use databases as intended;
- Physical damage to database servers caused by computer room fires or floods, overheating, lightning, accidental liquid spills, static discharge, electronic breakdowns/equipment failures and obsolescence;
- Design flaws and programming bugs in databases and the associated programs and systems, creating various security vulnerabilities (e.g. unauthorized privilege escalation), data loss/corruption, performance degradation etc.;
- Data corruption and/or loss caused by the entry of invalid data or commands, mistakes in database or system administration processes, sabotage/criminal damage etc.

# CHAPTER 3 INTERACTIVE SQL

**Introduction to SQL:-**

SQL, the most influential commercially marketed query language, SQL. SQL uses a combination of relational-algebra and relational-calculus constructs. Although we refer to the SQL language as a "query language," it can do much more than just query a database. It can define the structure of the data, modify data in the database, and specify security constraints

IBM developed the original version of SQL at its San Jose Research Laboratory (now **the Almaden Research Center**). IBM implemented the language, originally called Sequel, as part of the System R project in the early 1970s. The **Sequel** language has evolved since then, and its name has changed to **SQL** (**Structured Query Language**). Many products now support the SQL language. SQL has clearly established itself as *the* standard relational-database language.

In 1986, the **American National Standards Institute (ANSI**) and the **International Organization for Standardization (ISO) published an SQL standard;** called SQL-86.IBM published its own corporate SQL standard, the **Systems Application Architecture Database Interface (SAA-SQL**) in 1987. ANSI published an extended standard for SQL, SQL-89, in 1989. The next version of the standard was SQL-92 standard, and the most recent version is SQL.

**The SQL language has several parts:**

• **Data-definition language** (DDL). The SQL DDL provides commands for defining relation schemas, deleting relations, and modifying relation schemas.

• **Interactive data-manipulation language** (DML). The SQL DML includes a query language based on both the relational algebra and the tuple relational calculus. It includes also commands to insert tuples into, delete tuples from, and modify tuples in the database.

• **View definition**. The SQL DDL includes commands for defining views.

• **Transaction control**. SQL includes commands for specifying the beginning and ending of transactions.

• **Embedded SQL** and **dynamic SQL**. Embedded and dynamic SQL define how SQL statements can be embedded within general-purpose programming languages, such as C, C++, Java, PL/I, Cobol, Pascal, and Fortran.

• **Integrity**. The SQL DDL includes commands for specifying integrity constraints that the data stored in the database must satisfy. Updates that violate integrity constraints are disallowed.

• **Authorization**. The SQL DDL includes commands for specifying access rights to relations and views.

**Data Types of SQL;**

| | | |
|---|---|---|
| Varchar2 | The varchar2 data type supports a variable length character string. It also stores alphanumeric values. The size for this data type ranges from 1 - 4000 bytes. Using varchar2 saves disk space when compared to char. This statement can be justified with the help of an example. Considering a column assigned with varchar2 datatype of size 30 bytes, if the user enters 10 bytes of character, then the length in that row would only be 10 bytes and not 30 bytes. In the case of char, it would still occupy 30 bytes because the remaining would be blank padded by Oracle. | Name Varchar2 (10) |
| Long | This data type is used to store variable character length. Maximum size is 2GB.Long data type has several characters similar to varchar2 data type.<br>Restrictions:<br>Only one column in a table can have long data type. This should not contain unique or primary key constraints. The column cannot be indexed. | Doc LONG; |
| Number | The Number data type can store positive numbers, negative number, zeroes, fixed point number, and floating point numbers with a precision of 38.<br>Column-name number {p=38, s=0}<br>Column-name number(p) {fixed point}<br>Column-name (p, s) {floating point}<br>Where p is the precision which refers to the total number of digits, it varies between 1 to 38, s is the scale which refers to number of digits to the right of the decimal point, which varies between -84 to 127. | Empno number(10);<br>or<br>Empno number(10,2); |
| Date | Date data type is used to store date and time in a table. Oracle database makes use of its own format to store date in a fixed length of 7 bytes each for century, month, day, year, hour, minute and second. Default date data type is "dd-mm-yy".<br>To view system's date and time we can use the SQL function called sysdate( ).Valid date is from Jan 1, 4712 BC to Dec 31, 4712 AD. | Dob Date;<br>04-APR-86 |

| Data Type | Description | Example |
|---|---|---|
| Char | The char data type is used when fixed length character string is required. It can store alphanumeric values. The column length of such a data type can vary between 1 - 2000 bytes. By default it is one byte. If the user enters a value shorter than the specified length then the database blank-pads to the fixed length. | column_1 CHAR(10) |

**Data Definition Language (DDL)**

The **Data Definition Language (DDL)** manages table and index structure. The most basic items of DDL are the CREATE, ALTER, RENAME,TRUNCATE and DROP statements:

• **CREATE -**Creates an object (a table, for example) in the database.
• **DROP-** deletes an object in the database, usually irretrievably.
• **ALTER-** modifies the structure an existing object in various ways—for example, adding a column to an existing table.

- **RENAME- Rename** the table name.
- **TRUNCATE-** Delete the all rows but the table structure remain in database.

**Data Manipulation Language (DML)**

The **Data Manipulation Language (DML)** is the subset of SQL used to add, update and delete data.

The acronym **CRUD** refers to all of the major functions that need to be implemented in a relational database application to consider it complete. Each letter in the acronym can be mapped to a standard SQL statement:

| Operation | SQL | Description |
|---|---|---|
| **C**reate | INSERT INTO | inserts new data into a database |
| **R**ead (Retrieve) | SELECT | extracts data from a database |
| **U**pdate | UPDATE | updates data in a database |
| **D**elete (Destroy) | DELETE | deletes data from a database |

**DDL COMMONDS:-**

1. **CREATE**;- The **CREATE TABLE** statement is used to create a table in a database.

Syntax:
CREATE TABLE table_name
(
column_name1 data_type,
column_name2 data_type,
column_name3 data_type,
....)
The data type specifies what type of data the column can hold.

You are special data types for numbers, text dates, etc.
Examples:
Numbers: **int**, **float**
• Text/Stings: **varchar(X)**–where X is the length of the string
• Dates: **datetime**

**Example;-**

CREATE TABLE **CUSTOMER**
(
**CustomerId int** IDENTITY(1,1) PRIMARY KEY,
**CustomerNumber int** NOT NULL UNIQUE,
**LastName varchar(50)** NOT NULL,
**FirstName varchar(50)** NOT NULL,
**AreaCode int** NULL,
**Address varchar(50)** NULL,
**Phone varchar(50)** NULL,
)

2. **DROP TABLE**

Just like there is a ‘create table’ there is also a ‘drop table’, which simply removes the table.

Note that it doesn’t ask you for confirmation, and once you remove a table, it is gone forever.

Syntax:-
DROP TABLE <table-name>

Example:-

Drop Table Customer

3. **ALTER TABLE**

There is a command to 'alter' tables after you create them. This is usually only useful if the table already has data, and you don't want to drop it and recreate it (which is generally much simpler). Also, most databases have varying restrictions on what 'alter table' is allowed to do. For example, Oracle allows you do add a column, but not remove a column.

**The general syntax to add a field is**:

ALTER TABLE <table-name>
ADD <field-name> <data-type>

**The general syntax to modify a field is;**

ALTER TABLE <table-name>
MODIFY <field-name> <new-field-declaration>

**To delete a column in a table, use the following syntax**
**ALTER TABLE table_name**
**DROP COLUMN column_name**

Example:-

1. Alter Table Customer Add(Cust_City Char(20) not null)
2. Alter Table Customer Modify(Cust_Address Char(60) Not Null)
3. Alter Table Customer Drop Column Cust_City

4. **TRUNCATE** - remove all records from a table, including all spaces allocated for the records are removed. But the structures of table remain as it is.

**Syntax;**

TRUNCATE TABLE <TABLE NAME>;

**Example;-**

TRUNCATE TABLE CUSTOMER

5. **RENAME-** Rename the table name and structure remain same.

6. **Get the description of table.**

SQL>desc dept;

**DML - Data Manipulation Language**

This is a standard subset of SQL that is used for data manipulation. Intuitively, we need to first inset data into the database. Once it's there, we can retrieve it, modify it, and delete it. These directly correspond to: INSERT, SELECT, UPDATE, and DELETE statements.

**1 INSERT Statement**

To get data into a database, we need to use the 'insert' statement.

**The general syntax is:**

INSERT INTO <table-name> (<column1>,<column2>,<column3>,...)
VALUES (<column-value1>,<column-value2>,<column-value3>);

The column names (i.e.: column1, etc.) must correspond to column values (i.e.: column-value1, etc.). There is a short-hand for the statement:

INSERT INTO <table-name>
VALUES (<column-value1>,<column-value2>,<column-value3>);

In which the column values must correspond exactly to the order columns appear in the 'create table' declaration. It must be noted, that this sort of statement should (or rather, must) be avoided! If someone changes the table, moves columns around in the table declaration, the code using the shorthand insert statement will fail.

**A typical example, of inserting the 'person' record we've created earlier would be:**

INSERT INTO PERSON(PERSONID,LNAME,FNAME,DOB)
VALUES(1,'DOE','JOHN','1956-11-23');

OR

INSERT INTO PERSON VALUES(1,'DOE','JOHN','1965-11-23')

**2 SELECT Statements**

Probably the most used statement in all of SQL is the SELECT statement. The select statement has the general format of:

SELECT <column-list>
FROM <table-list>
WHERE <search-condition>

The column-list indicates what columns you're interested in (the ones which you want
to appear in the result), the table-list is the list of tables to be used in the query, and search-condition specifies what criteria you're looking for.

An example of a short-hand version to retrieve all 'person' records we've been using:

**SELECT * FROM PERSON;**

**3 The WHERE Clause**

The WHERE clause is used in UPDATE, DELETE, and SELECT statements, and has the same format in all these cases. It has to be evaluated to either true or false.

Lists some of the common operators.

= equals to
> greater than
< less than
>= greater than or equal to
<= less than or equal to
<> not equal to

**4 UPDATE Statements**

The update statement is used for changing records. The general syntax is:

UPDATE <table-name>
SET <column1> = <value1>, <column2> = <value2>, ...
WHERE <criteria>

The criteria are what select the records for update. The 'set' portion indicates which columns should be updated and to what values. An example of the use would be:

UPDATE PERSON
SET FNAME='Clark', LNAME='Kent'
WHERE FNAME='Superman';
Usually, the AND operator has higher precedence than OR.The function TO DATE is available on Oracle.

**5 DELETE Statements**

The 'delete' is used to remove elements from the database. The syntax is very similar to update and select statements:

DELETE FROM <table-name>
WHERE <criteria>
Basically we select which records we want to delete using the where clause. An example use would be:

DELETE FROM PERSON
WHERE FNAME='Superman';

**6. CALL** - call a PL/SQL or Java subprogram

**DCL - Data Control Language**

This is a standard subset of SQL that is used for security management. Most databases have their own flavored syntax, but generally, there exist two commands: GRANT, and REVOKE, these 'grant' and 'remove' privileges.

**1 GRANT Statement**

The general format is something like this:
GRANT <privilege> ON <object> TO <who> ;

Basically, a privilege can be something like 'update' or 'select', etc., or it can be 'all' when granting 'all' privileges. An 'object' can be pretty much anything, but is often a database, or database table. The 'who' is generally a database login/user. Some databases (like MySQL) will actually create the user if they don't already exist.

In some cases, you also have the option of specifying the user password, via: 'identified by' parameter.

**2 REVOKE Statements**

The revoke is the opposite of GRANT. The general format is:

REVOKE <privilege> ON <object> FROM <who> ;

The individual elements are the same as for GRANT statement.

**For Example:_**

(a) GRANT SELECT ON employee TO user1

This command grants a SELECT permission on employee table to user1.

(b) REVOKE SELECT ON employee FROM user1

This command will revoke a SELECT privilege on employee table from user1.

**Privileges and Roles**

• Privileges define the access rights provided to a user on a database objects. There are two types of privileges:

(a) **System Privileges**: This indicate user to CREATE, ALTER, or DROP database elements._

(b) **Object Privileges**: This allows user to EXECUTE, SELECT,"INSERT, or DELETE data from database objects to which the privileges apply.

• Roles are the collection of privileges or access rights. When there are many users in a database it becomes difficult to grant or revoke privileges to the users.

• So, if we define roles we can automatically grant/revoke privileges.

**Transaction Control (TCL)**

Statements are used to manage the changes made by DML statements. It allows statements to be grouped together into logical transactions.

1. **COMMIT - save work done**

   **Save changes (transactional).**

**Syntax:**

COMMIT [WORK] [COMMENT '*comment_text*']

COMMIT [WORK] [FORCE '*force_text*' [,*int*] ]

**FORCE - will manually commit an in-doubt *distributed* transaction force_text - transaction identifier*)* int - sets a specific.**

**If a network or machine failure prevents a distributed transaction from committing properly, Oracle will store any commit comment in the data dictionary along with the transaction ID.**

**INPUT:**
**SQL>commit;**
**RESULT:** Commit complete

2. **SAVEPOINT - identify a point in a transaction to which you can later roll back**
   **Save changes to a point (transactional).**

**Syntax:**

SAVEPOINT *text_identifier*

**Example**:

```
UPDATE employees
SET salary = 95000
WHERE last_name = 'Smith';
SAVEPOINT justsmith;
UPDATE employees
SET salary = 1000000;
SAVEPOINT everyone;
SELECT SUM(salary) FROM employees;
ROLLBACK TO SAVEPOINT justsmith;
COMMIT
```

3. **ROLLBACK - restore database to original since the last COMMIT**
   **Undo work done (transactional).**

**Syntax:**

ROLLBACK [WORK] [TO [SAVEPOINT]'*savepoint_text_identifier*'];
ROLLBACK [WORK] [FORCE '*force_text*'];

FORCE - will manually rollback an in-doubt *distributed* transaction

**INPUT:**
SQL>rollback;

**RESULT:** Rollback complete.

4. **SET TRANSACTION - Change transaction options like isolation level and what rollback segment to use**

**OPERATOR**

**What is an Operator in SQL?**

An operator is a reserved word or a character used primarily in an SQL statement's WHERE clause to perform operation(s), such as comparisons and arithmetic operations. Operators are used to specify conditions in an SQL statement and to serve as conjunctions for multiple conditions in a statement.

**Arithmetic operators**
**Comparison operators**
**Logical operators**
**Operators used to negate conditions**

**SQL Arithmetic Operators:**
Assume variable a holds 10 and variable b holds 20, then:
Show Examples

| **Operator** | **Description** | **Example** |
|---|---|---|
| + Addition | Adds values on either side of the operator | a + b will give 30 |
| - Subtraction | Subtracts right hand operand from left hand operand | a - b will give -10 |
| * Multiplication - | Multiplies values on either side of the operator | a * b will give 200 |

/ Division - Divides left hand operand by right hand operand b / a will give 2

% Modulus - Divides left hand operand by right hand operand and returns remainder b % a will give 0

**SQL Comparison Operators:**

Assume variable a holds 10 and variable b holds 20, then:

Show Examples

| Operator | Description | Example |
|---|---|---|
| = | Checks if the values of two operands are equal or not, if yes then condition becomes true. | (a = b) is not true. |
| != | Checks if the values of two operands are equal or not, if values are not equal then condition becomes true. | (a != b) is true |
| <> | Checks if the values of two operands are equal or not, if values are not equal then condition becomes true. | (a <> b) is true. |
| > | Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true. | (a > b) is not true |
| < | Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true. | (a < b) is true. |
| >= | Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true. | (a >= b) is not true. |
| <= | Checks if the value of left operand is less than or equal to the value of right operand, | (a <= b) is true. |

if yes then condition becomes true.

!< Checks if the value of left operand is not less than the value of right operand, if yes then condition becomes true. (a !< b) is false.

!> Checks if the value of left operand is not greater than the value of right operand, if yes then condition becomes true. (a !> b) is true.

**SQL Logical Operators:**

Here is a list of all the logical operators available in SQL.

Show Examples

| Operator | Description |
|---|---|
| ALL | The ALL operator is used to compare a value to all values in another value set. |
| AND | The AND operator allows the existence of multiple conditions in an SQL statement's WHERE clause. |
| ANY | The ANY operator is used to compare a value to any applicable value in the list according to the condition. |
| BETWEEN | The BETWEEN operator is used to search for values that are within a set of values, given the minimum value and the maximum value. |
| EXISTS | The EXISTS operator is used to search for the presence of a row in a specified table that meets certain criteria. |
| IN | The IN operator is used to compare a value to a list of literal values that have been specified. |
| LIKE | The LIKE operator is used to compare a value to similar values using wildcard operators. |
| NOT | The NOT operator reverses the meaning of the logical operator with which it is used. Eg: NOT EXISTS, NOT BETWEEN, NOT IN, etc. **This is a negate operator.** |

| | |
|---|---|
| OR | The OR operator is used to combine multiple conditions in an SQL statement's WHERE clause. |
| IS NULL | The NULL operator is used to compare a value with a NULL value. |
| UNIQUE uniqueness | The UNIQUE operator searches every row of a specified table for (no duplicates). |

**STRING FUNCTION:-**

| Name | Description |
|---|---|
| ASCII | Returns numeric value of left-most character |
| BIN | Returns a string representation of the argument |
| BIT_LENGTH | Returns length of argument in bits |
| CHAR_LENGTH | Returns number of characters in argument |
| CHAR | Returns the character for each integer passed |
| CHARACTER_LENGTH | A synonym for CHAR_LENGTH |
| CONCAT_WS | Returns concatenate with separator |
| CONCAT | Returns concatenated string |
| CONV | Converts numbers between different number bases |
| ELT | Returns string at index number |
| EXPORT_SET | Returns a string such that for every bit set in the value bits, you get an on string and for every unset bit, you get an off string |

| | |
|---|---|
| FIELD | Returns the index *position* of the first argument in the subsequent arguments |
| FIND_IN_SET | Returns the index position of the first argument within the second argument |
| FORMAT | Returns a number formatted to specified number of decimal places. |
| HEX<br>value | Returns a string representation of a hex |
| INSERT | Inserts a substring at the specified position up to the specified number of characters. |
| INSTR | Returns the index of the first occurrence of substring. |
| LCASE | Synonym for LOWER |
| LEFT | Returns the leftmost number of characters as specified. |
| LENGTH | Returns the length of a string in bytes |
| LOAD_FILE | Loads the named file |
| LOCATE | Returns the position of the first occurrence of substring |
| LOWER | Returns the argument in lowercase |
| LPAD | Returns the string argument, left-padded with the specified string. |
| LTRIM | Removes leading spaces |
| MAKE_SET | Returns a set of comma-separated strings that have the corresponding bit in bits set |

| | |
|---|---|
| MID | Returns a substring starting from the specified position |
| OCT | Returns a string representation of the octal argument |
| OCTET_LENGTH | A synonym for LENGTH |
| ORD | If the leftmost character of the argument is a multi-byte character, returns the code for that character |
| POSITION | A synonym for LOCATE |
| QUOTE | Escapes the argument for use in an SQL statement |
| REGEXP | Pattern matching using regular expressions |
| REPEAT | Repeats a string the specified number of times |
| REPLACE | Replaces occurrences of a specified string |
| REVERSE | Reverses the characters in a string |
| RIGHT | Returns the specified rightmost number of characters |
| RPAD | Appends string the specified number of times |
| RTRIM | Removes trailing spaces |
| SOUNDEX | Returns a soundex string |
| SOUNDS LIKE | Compares sounds |
| SPACE | Returns a string of the specified number of spaces |

| | |
|---|---|
| STRCMP | Compares two strings |
| SUBSTRING_INDEX | Returns a substring from a string before the specified number of occurrences of the delimiter |
| SUBSTRING, SUBSTR | Returns the substring as specified |
| TRIM | Removes leading and trailing spaces |
| UCASE | Synonym for UPPER |
| UNHEX | Converts each pair of hexadecimal digits to a character |
| UPPER | Converts to uppercase |

**ASCII*str***

Returns the numeric value of the leftmost character of the string str. Returns 0 if str is the empty string. Returns NULL if str is NULL. ASCII works for characters with numeric values from 0 to 255.

```
SQL> SELECT ASCII('2');
+____________+
| ASCII('2') |
+____________+
| 50 |
+____________+
```

```
SQL> SELECT ASCII('dx');
+_____________+
| ASCII('dx') |
+_____________+
| 100 |
+_____________+
```

**BIN*N***
Returns a string representation of the binary value of N, where N is a longlong *BIGINT* number. This is equivalent to CONV*N*, 10, 2. Returns NULL if N is NULL.

```
SQL> SELECT BIN(12);
+_____________________________________+
| BIN(12) |
+_____________________________________+
| 1100 |
+_____________________________________+
```

**BIT_LENGTH*str***
Returns the length of the string str in bits.

```
SQL> SELECT BIT_LENGTH('text');
+_____________________________________+
| BIT_LENGTH('text') |
+_____________________________________+
| 32 |
+_____________________________________+
```

**CHAR*N*, . . . [*USINGcharsetname*]**
CHAR interprets each argument N as an integer and returns a string consisting of the characters given by the code values of those integers. NULL values are skipped.

```
SQL> SELECT CHAR(77,121,83,81,'76');
+_____________________________________+
| CHAR(77,121,83,81,'76') |
+_____________________________________+
| SQL |
+_____________________________________+
```

**CHAR_LENGTH*str***

Returns the length of the string str measured in characters. A multi-byte character counts as a single character. This means that for a string containing five two-byte characters, LENGTH returns 10, whereas CHAR_LENGTH returns 5.

```
SQL> SELECT CHAR_LENGTH("text");
+_____________________________+
| CHAR_LENGTH("text") |
+_____________________________+
| 4 |
+_____________________________+
```

**CHARACTER_LENGTH*str***
CHARACTER_LENGTH is a synonym for CHAR_LENGTH.

**CONCAT*str*1, *str*2, . . .**
Returns the string that results from concatenating the arguments. May have one or more arguments. If all arguments are non-binary strings, the result is a non-binary string. If the arguments include any binary strings, the result is a binary string. A numeric argument is converted to its equivalent binary string form; if you want to avoid that, you can use an explicit
type cast, as in this example:

```
SQL> SELECT CONCAT('My', 'S', 'QL');
+_____________________________+
| CONCAT('My', 'S', 'QL') |
+_____________________________+
| SQL |
+_____________________________+
```

**CONCAT_WS*separator*, *str*1, *str*2, . . .**
CONCAT_WS stands for Concatenate With Separator and is a special form of CONCAT. The first argument is the separator for the rest of the arguments. The separator is added between the strings to be concatenated. The separator can be a string, as can the rest of the arguments. If the separator is NULL, the result is NULL.

```
SQL> SELECT CONCAT_WS(',','First nam e','Last Nam e' );
+_____________________________+
```

```
| CONCAT_WS(',','First nam e','Last Nam e' ) |
+_____________________________________+
| First nam e, Last Nam e |
+_____________________________________+
```

**CONV*N*, *frombase*, *tobase***

Converts numbers between different number bases. Returns a string representation of the number N, converted from base from_base to to_base. Returns NULL if any argument is NULL. The argument N is interpreted as an integer, but may be specified as an integer or a string. The minimum base is 2 and the maximum base is 36. If to_base is a negative number, N is regarded as a signed number. Otherwise, N is treated as unsigned. CONV works with 64-bit precision.

```
SQL> SELECT CONV('a',16,2);
+_____________________________________+
| CONV('a',16,2) |
+_____________________________________+
| 1010 |
+_____________________________________+
```

**ELT*N*, *str*1, *str*2, *str*3, . . .**

Returns str1 if N = 1, str2 if N = 2, and so on. Returns NULL if N is less than 1 or greater than the number of arguments. ELT is the complement of FIELD.

```
SQL> SELECT ELT(1, 'ej', 'Heja', 'hej', 'foo');
+_____________________________________+
| ELT(1, 'ej', 'Heja', 'hej', 'foo') |
+_____________________________________+
| ej |
+_____________________________________+
```

**EXPORT_SET*bits*, *on*, *off*[, *separator*[, *numberofbits*]]**

Returns a string such that for every bit set in the value bits, you get an on string and for every bit
not set in the value, you get an off string. Bits in bits are examined from right to left *from low – order to high – order bits*. Strings are added to the result from left to right, separated by the

Separator string *the default being the comma character*. , . . The number of bits examined is given by number_of_bits *defaults to* 64.

```
SQL> SELECT EXPORT_SET(5,'Y','N',',',4);
+_____________________________________+
| EXPORT_SET(5,'Y','N',',',4) |
+_____________________________________+
| Y,N,Y,N |
+_____________________________________+
```

**FIELD*str*, *str*1, *str*2, *str*3, . . .**
Returns the index *positionstartingwith*1 of str in the str1, str2, str3, ... list. Returns 0 if str is not found.

```
SQL> SELECT FIELD('ej', 'Hej', 'ej', 'Heja', 'hej', 'foo');
+_____________________________________+
| FIELD('ej', 'Hej', 'ej', 'Heja', 'hej', 'foo') |
+_____________________________________+
| 2 |
+_____________________________________+
```

**FIND_IN_SET*str*, *strlist***
Returns a value in the range of 1 to N if the string str is in the string list strlist consisting of N
substrings.

```
SQL> SELECT FIND_IN_SET('b','a,b,c,d');
+_____________________________________+
| SELECT FIND_IN_SET('b','a,b,c,d') |
+_____________________________________+
| 2 |
+_____________________________________+
```

**FORMAT*X*, *D***
Formats the number X to a format like '#,###,###.##', rounded to D decimal places, and returns the result as a string. If D is 0, the result has no decimal point or fractional part.

```
SQL> SELECT FORMAT(12332.123456, 4);
```

```
+-----------------------------------------+
| FORMAT(12332.123456, 4) |
+-----------------------------------------+
| 12,332.1235 |
+-----------------------------------------+
```

**HEX*NorS***

If N_or_S is a number, returns a string representation of the hexadecimal value of N, where N is a long *BIGINT* number. This is equivalent to CONV*N*, 10, 16. If N_or_S is a string, returns a hexadecimal string representation of N_or_S where each character in N_or_S is converted to two hexadecimal digits.

```
SQL> SELECT HEX(255);
+-----------------------------------------+
| HEX(255) |
+-----------------------------------------+
| FF |
+-----------------------------------------+
```

```
SQL> SELECT 0x616263;
+-----------------------------------------+
| 0x616263 |
+-----------------------------------------+
| abc |
+-----------------------------------------+
```

**INSERT*str*, *pos*, *len*, *newstr***

Returns the string str, with the substring beginning at position pos and len characters long replaced by the string new str. Returns the original string if pos is not within the length of the string. Replaces the rest of the string from position pos if len is not within the length of the rest of the string. Returns NULL if any argument is NULL.

```
SQL> SELECT INSERT('Quadratic', 3, 4, 'What');
+-----------------------------------------+
| INSERT('Quadratic', 3, 4, 'What') |
+-----------------------------------------+
```

```
| QuWhattic |
+_____________________________________+
```

**INSTR*str*, *substr***
Returns the position of the first occurrence of substring substr in string str. This is the same as the two-argument form of LOCATE, except that the order of the arguments is reversed.

```
SQL> SELECT INSTR('foobarbar', 'bar');
+_____________________________________+
| INSTR('foobarbar', 'bar') |
+_____________________________________+
| 4 |
+_____________________________________+
```

**LCASE*str***
LCASE is a synonym for LOWER.

**LEFT*str*, *len***
Returns the leftmost len characters from the string str, or NULL if any argument is NULL.

```
SQL> SELECT LEFT('foobarbar', 5);
+_____________________________________+
| LEFT('foobarbar', 5) |
+_____________________________________+
| fooba |
+_____________________________________+
```

**LENGTH*str***
Returns the length of the string str, measured in bytes. A multi-byte character counts as multiple
bytes. This means that for a string containing five two-byte characters, LENGTH returns 10,
whereas CHAR_LENGTH returns 5.

```
SQL> SELECT LENGTH('text');
```

```
+_____________________________________+
| LENGTH('text') |
+_____________________________________+
| 4 |
+_____________________________________+
```

**LOAD_FILE*filename***

Reads the file and returns the file contents as a string. To use this function, the file must be located on the server host, you must specify the full pathname to the file, and you must have the FILE privilege. The file must be readable by all and its size less than max_allowed_packet bytes.

If the file does not exist or cannot be read because one of the preceding conditions is not satisfied, the function returns NULL. As of SQL 5.0.19, the character_set_filesystem system variable controls interpretation of filenames that are given as literal strings.

```
SQL> UPDATE table_test
-> SET blob_col=LOAD_FILE('/tm p/picture')
-> WHERE id=1;
```

.....

**LOCATE*substr*, *str*, LOCATE*substr*, *str*, *pos***

The first syntax returns the position of the first occurrence of substring substr in string str. The second syntax returns the position of the first occurrence of substring substr in string str, starting at position pos. Returns 0 if substr is not in str.

```
SQL> SELECT LOCATE('bar', 'foobarbar');
+_____________________________________+
| LOCATE('bar', 'foobarbar') |
+_____________________________________+
| 4 |
+_____________________________________+
```

**LOWER*str***

Returns the string str with all characters changed to lowercase according to the current character
set mapping.

```
SQL> SELECT LOWER('QUADRATICALLY');
+-----------------------------------------+
| LOWER('QUADRATICALLY') |
+-----------------------------------------+
| quadratically |
+-----------------------------------------+
```

**LPAD*str*, *len*, *padstr***

Returns the string str, left-padded with the string pad str to a length of len characters. If str is longer than len, the return value is shortened to len characters.

```
SQL> SELECT LPAD('hi',4,'??');
+------------------------------------+
| LPAD('hi',4,'??') |
+------------------------------------+
| ??hi |
+------------------------------------+
```

**LTRIM*str***

Returns the string str with leading space characters removed.

```
SQL> SELECT LTRIM(' barbar');
+-------------------------------------+
| LTRIM(' barbar') |
+-------------------------------------+
| barbar |
+-------------------------------------+
```

**MAKE_SET*bits*, *str*1, *str*2, . . .**

Returns a set value *a string containing substrings separated by. , . characters* consisting of the strings that have the corresponding bit in bits set. str1 corresponds to bit 0, str2 to bit 1, and so on. NULL values in str1, str2, ... are not appended to the result.

```
SQL> SELECT MAKE_SET(1,'a','b','c');
+-------------------------------------+
| MAKE_SET(1,'a','b','c') |
```

```
+_____+
| a |
+_____+
```

**MID*str*, *pos*, *len***
MID*str*, *pos*, *len* is a synonym for SUBSTRING*str*, *pos*, *len*.

**OCT*N***
Returns a string representation of the octal value of N, where N is a longlong *BIGINT* number. This is equivalent to CONV*N*, 10, 8. Returns NULL if N is NULL.

```
SQL> SELECT OCT(12);
+_____+
| OCT(12) |
+_____+
| 14 |
+_____+
```

**OCTET_LENGTH*str***
OCTET_LENGTH is a synonym for LENGTH.

**ORD*str***
If the leftmost character of the string str is a multi-byte character, returns the code for that character, calculated from the numeric values of its constituent bytes using this formula:

```
(1st byte code)
+ (2nd byte code . 256)
+ (3rd byte code . 2562) ...
```
If the leftmost character is not a multi-byte character, ORD returns the same value as the ASCII
function.

```
SQL> SELECT ORD('2');
+_____+
| ORD('2') |
+_____+
```

```
| 50 |
+____________________________________+
```

**POSITION*substrINstr***
POSITION*substrINstr* is a synonym for LOCATE*substr*, *str*.

**QUOTE*str***
Quotes a string to produce a result that can be used as a properly escaped data value in an SQL
statement. The string is returned enclosed by single quotes and with each instance of single quote
', backslash '\', ASCII NUL, and Control-Z preceded by a backslash. If the argument is NULL, the return value is the word 'NULL' without enclosing single quotes.

```
SQL> SELECT QUOTE('Don\'t!');
+____________________________________+
| QUOTE('Don\'t!') |
+____________________________________+
| 'Don\'t!' |
+____________________________________+
```

**REPEAT*str*, *count***
Returns a string consisting of the string str repeated count times. If count is less than 1, returns an
empty string. Returns NULL if str or count are NULL.

```
SQL> SELECT REPEAT('SQL', 3);
+____________________________________+
| REPEAT('SQL', 3) |
+____________________________________+
| SQLSQLSQL |
+____________________________________+
```

**REPLACE*str*, *fromstr*, *tostr***
Returns the string str with all occurrences of the string from_str replaced by the string to_str.
REPLACE performs a case-sensitive match when searching for from_str.

```
SQL> SELECT REPLACE('www.m ysql.com ', 'w', 'Ww');
+____________________________________+
| REPLACE('www.m ysql.com ', 'w', 'Ww') |
```

```
+_____________________________________+
| WwWwWw.m ysql.com |
+_____________________________________+
```

**REVERSE*str***
Returns the string str with the order of the characters reversed.

```
SQL> SELECT REVERSE('abcd');
+_____________________________________+
| REVERSE('abcd') |
+_____________________________________+
| dcba |
+_____________________________________+
```

**RIGHT*str*, *len***
Returns the rightmost len characters from the string str, or NULL if any argument is NULL.

```
SQL> SELECT RIGHT('foobarbar', 4);
+_____________________________________+
| RIGHT('foobarbar', 4) |
+_____________________________________+
| rbar |
+_____________________________________+
```

**RPAD*str*, *len*, *padstr***
Returns the string str, right-padded with the string padstr to a length of len characters. If str is longer than len, the return value is shortened to len characters.

```
SQL> SELECT RPAD('hi',5,'?');
+_____________________________________+
| RPAD('hi',5,'?') |
+_____________________________________+
| hi??? |
+_____________________________________+
```

**RTRIM*str***
Returns the string str with trailing space characters removed.

```
SQL> SELECT RTRIM('barbar ');
+_____________________________________+
| RTRIM('barbar ') |
+_____________________________________+
| barbar |
+_____________________________________+
```

**SOUNDEX*str***
Returns a soundex string from str. Two strings that sound almost the same should have identical
soundex strings. A standard soundex string is four characters long, but the SOUNDEX function
returns an arbitrarily long string. You can use SUBSTRING on the result to get a standard soundex string. All non-alphabetic characters in str are ignored. All international alphabetic characters outside the A-Z range are treated as vowels.

```
SQL> SELECT SOUNDEX('Hello');
+_____________________________________+
| SOUNDEX('Hello') |
+_____________________________________+
| H400 |
+_____________________________________+
```

**expr1 SOUNDS LIKE expr2**
This is the same as SOUNDEX*expr*1 = SOUNDEX*expr*2.
**SPACE*N***
Returns a string consisting of N space characters.

```
SQL> SELECT SPACE(6);
+_____________________________________+
| SELECT SPACE(6) |
+_____________________________________+
| ' ' |
+_____________________________________+
```

**STRCMP*str*1, *str*2**
Compares two strings and returns 0 if both strings are equal, it returns -1 if the first argument is smaller than the second according to the current sort order otherwise it returns 1.

```
SQL> SELECT STRCMP('MOHD', 'MOHD');
+_____________________________________+
| STRCMP('MOHD', 'MOHD') |
+_____________________________________+
| 0 |
+_____________________________________+
```

Another example is:

```
SQL> SELECT STRCMP('AMOHD', 'MOHD');
+_____________________________________+
| STRCMP('AMOHD', 'MOHD') |
+_____________________________________+
| -1 |
+_____________________________________+
```

Let's see one more example:

```
SQL> SELECT STRCMP('MOHD', 'AMOHD');
+_____________________________________+
| STRCMP('MOHD', 'AMOHD') |
+_____________________________________+
| 1 |
+_____________________________________+
```

**SUBSTRING*str*, *pos***
**SUBSTRING*str FROM pos***
**SUBSTRING*str*, *pos*, *len***
**SUBSTRING*str FROM pos FOR len***

The forms without a len argument return a substring from string str starting at position pos. The
forms with a len argument return a substring len characters long from string str, starting at position pos. The forms that use FROM are standard SQL syntax. It is also possible to use a negative value for pos. In this case, the beginning of the substring is pos characters from the end of the string, rather than the beginning. A negative value may be used for pos in any of the forms of this function.

```
SQL> SELECT SUBSTRING('Quadratically',5);
+_____________________________________+
| SSUBSTRING('Quadratically',5) |
+_____________________________________+
```

```
| ratically |
+____________________________________+

SQL> SELECT SUBSTRING('foobarbar' FROM 4);
+____________________________________+
| SUBSTRING('foobarbar' FROM 4) |
+____________________________________+
| barbar |
+____________________________________+

SQL> SELECT SUBSTRING('Quadratically',5,6);
+____________________________________+
| SUBSTRING('Quadratically',5,6) |
+____________________________________+
| ratica |
+____________________________________+
```

**SUBSTRING_INDEX*str*, *delim*, *count***

Returns the substring from string str before count occurrences of the delimiter delim. If count is
positive, everything to the left of the final delimiter *counting from the left* is returned. If count is negative, everything to the right of the final delimiter *countingfromtheright* is returned. SUBSTRING_INDEX performs a case-sensitive match when searching for delim.

```
SQL> SELECT SUBSTRING_INDEX('www.m ysql.com ', '.', 2);
+____________________________________+
| SUBSTRING_INDEX('www.m ysql.com ', '.', 2) |
+____________________________________+
| www.m ysql |
+____________________________________+
```

**TRIM[*BOTH* | *LEADING* | *TRAILING*[*remstr*]*FROM*]*str***
**TRIM[*remstrFROM*]*str***

Returns the string str with all remstr prefixes or suffixes removed. If none of the specifiers BOTH,
LEADING, or TRAILING is given, BOTH is assumed. remstr is optional and, if not specified, spaces are removed.

```
SQL> SELECT TRIM(' bar ');
```

```
+-------------------+
| TRIM(' bar ') |
+-------------------+
| bar |
+-------------------+

SQL> SELECT TRIM(LEADING 'x' FROM 'xxxbarxxx');
+-----------------------------------+
| TRIM(LEADING 'x' FROM 'xxxbarxxx') |
+-----------------------------------+
| barxxx |
+-----------------------------------+

SQL> SELECT TRIM(BOTH 'x' FROM 'xxxbarxxx');
+---------------------------------+
| TRIM(BOTH 'x' FROM 'xxxbarxxx') |
+---------------------------------+
| bar |
+---------------------------------+

SQL> SELECT TRIM(TRAILING 'xyz' FROM 'barxxyz');
+------------------------------------+
| TRIM(TRAILING 'xyz' FROM 'barxxyz') |
+------------------------------------+
| barx |
+------------------------------------+
```

**UCASE*str***

UCASE is a synonym for UPPER.

**UNHEX*str***

Performs the inverse operation of HEX*str*. That is, it interprets each pair of hexadecimal digits in the argument as a number and converts it to the character represented by the number. Then resulting characters are returned as a binary string.

```
SQL> SELECT UNHEX('4D7953514C');
+---------------------------+
| UNHEX('4D7953514C') |
```

```
+_____+
| SQL |
+_____+
```

The characters in the argument string must be legal hexadecimal digits: '0' .. '9', 'A' .. 'F', 'a' .. 'f'. If UNHEX encounters any non-hexadecimal digits in the argument, it returns NULL.

**UPPER*str*** Returns the string str with all characters changed to uppercase according to the current character set mapping.

```
SQL> SELECT UPPER('Allah-hus-sam ad');
+_____+
| UPPER('Allah-hus-sam ad') |
+_____+
| ALLAH-HUS-SAMAD |
+_____+
```

**Queries using Group by, having, and Order by clause, Joins, Types of Joins, Sub queries.**

***The JOIN clause***

When a query includes more than one table, you need to specify how records from the tables are to be matched up to produce the result set. This is called *joining* the tables. The JOIN sub-clause of SQL-SELECT's FROM clause is used for this purpose.

A join has two parts: the pair of tables to be joined, and the condition for matching records. The syntax looks like Listing 1.

Listing 1. The JOIN clause lists the tables to be joined and the expression that matches up records in the two tables.

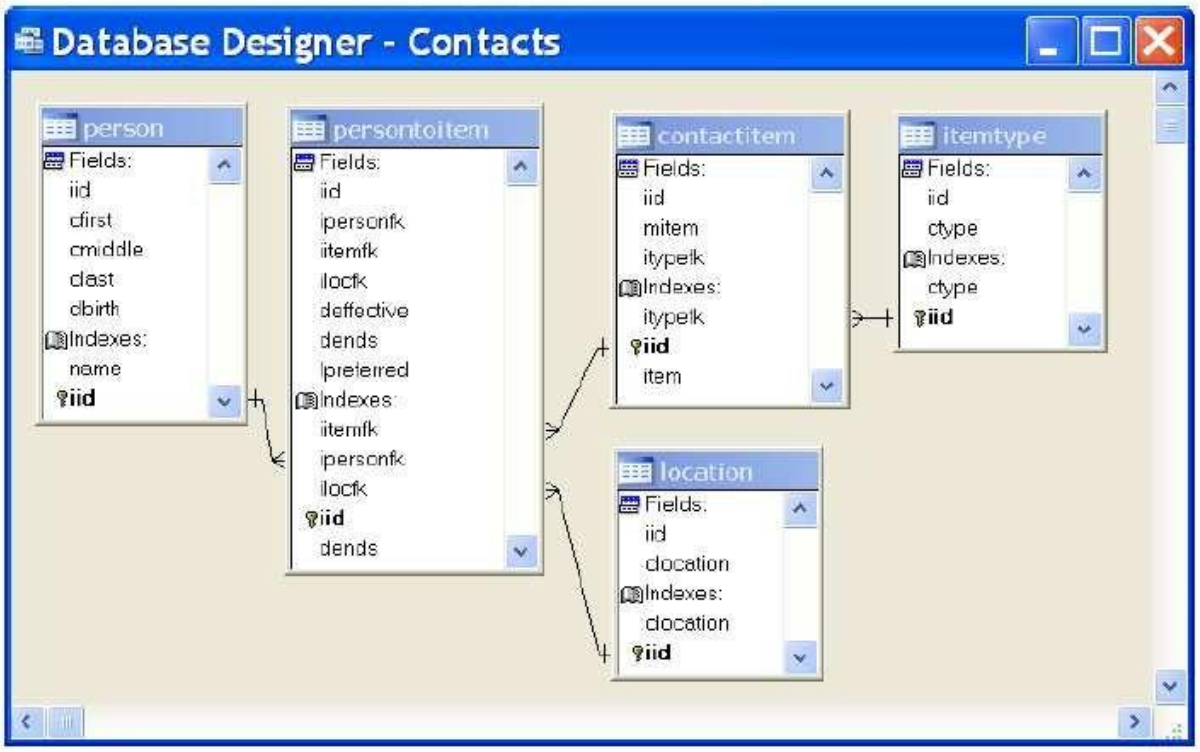
<Table1> JOIN <Table2> ON <expression>

Most often, the expression following the ON keyword (called the *join condition*) matches the primary key (PK) of one table with a foreign key (FK) in another table. For example, using the sample Northwind database, the query in Listing 2 produces a list of products by supplier. To do so, it matches the SupplierID field of Suppliers (the PK) to the SupplierID of Products.

Listing 2. The simplest joins match the primary key of one table with a foreign key in another.

```
SELECT CompanyName, ProductName ;
FROM Products ;
JOIN Suppliers ;
ON Suppliers.SupplierID = Products.SupplierID ;
ORDER BY CompanyName, ProductName ;
INTO CURSOR csrProductsBySupplier
```

However, the expression can be more complex than a simple comparison. For example, if the criteria for joining two tables involve multiple fields, you can use AND and OR to combine multiple comparisons. (The example databases, North wind and Contacts, don't include any tables where you need multiple fields to join.)




***Outer joins***

All of the multi-table queries in the preceding section include records from one table only if a match is found in the other tables. For example, only customers who placed orders in September, 1996, appear in the results of Listing 5. Similarly, only those products that were ordered in that month are included in the results. Such a join, which filters out records without matches, is called an *inner join*. Unless you specify otherwise, all joins are inner; you can explicitly specify that you want an inner join in VFP by including the keyword INNER before JOIN.

There are situations where you want to include the unmatched records, as well. A join that includes all the records from one or both of its tables, regardless of

matches in the other table, is called an *outer join*. There are three types of outer joins: *left joins*, *right joins* and *full joins*. In a left join, all the records from the table listed before the JOIN keyword are included, along with whichever records they match from the table after the JOIN keyword; use LEFT JOIN or LEFT OUTER JOIN to specify a left join. A right join, indicated by RIGHT JOIN or RIGHT OUTER JOIN, is the opposite; the results include all the records from the table listed after the JOIN keyword, along with whichever records they match from the table listed before the JOIN keyword. A full join includes all the records from both tables, making matches where possible; for a full join, use FULL JOIN or FULL OUTER JOIN.

**INNER JOIN**

The INNER JOIN keyword returns rows when there is at least one match in both tables.

**SYNTAX**

SELECT column_name(s) FROM table_name1 INNER JOIN table_name2 ON table_name1.column_name=table_name2.column_name

**PROGRAM**

SELECT Persons.LastName, Persons.FirstName, Orders.OrderNo FROM Persons INNER JOIN Orders ON Persons.P_Id=Orders.P_Id ORDER BY Persons.LastName

**LEFT JOIN**

The LEFT JOIN keyword returns all rows from the left table (table_name1), even if there are no matches in the right table (table_name2).

**SYNTAX**

SELECT column_name(s) FROM table_name1 LEFT JOIN table_name2 ON table_name1.column_name=table_name2.column_name

**PROGRAM**

SELECT Persons.LastName, Persons.FirstName, Orders.OrderNo FROM Persons LEFT JOIN Orders ON Persons.P_Id=Orders.P_Id ORDER BY Persons.LastName

**RIGHT JOIN**

The RIGHT JOIN keyword returns all the rows from the right table (table_name2), even if there are no matches in the left table (table_name1).

**SYNTAX**

SELECT column_name(s) FROM table_name1 RIGHT JOIN table_name2 ON table_name1.column_name=table_name2.column_name

**PROGRAM**

SELECT Persons.LastName, Persons.FirstName, Orders.OrderNo FROM Persons RIGHT JOIN Orders ON Persons.P_Id=Orders.P_Id ORDER BY Persons.LastName

**FULL JOIN**

The FULL JOIN keyword return rows when there is a match in one of the tables.

**SYNTAX**

SELECT column_name(s) FROM table_name1 FULL JOIN table_name2 ON table_name1.column_name=table_name2.column_name

**PROGRAM**

SELECT Persons.LastName, Persons.FirstName, Orders.OrderNo FROM Persons FULL JOIN Orders ON Persons.P_Id=Orders.P_Id ORDER BY Persons.LastName

**Equijoins**

To determine the name of an employee's department, you compare the value in the DEPTNO column in the EMP table with the DEPTNO values in the DEPT table.

The relationship between the EMP and DEPT table is an equijoin - that is, values in the DEPTNO column on both tables must be equal.

Frequently, this type of join involves primary and foreign key complements.

**Note: Equijoins are also called simple joins or inner joins.**

**Obtaining Data from Multiple Tables**

```
SELECT e.empno, e.deptno, d.loc
FROM emp e, dept d
WHERE e.deptno = d.deptno;
```

Data from Multiple Tables

Sometimes you need to use data from more than one table. In the slide example, the report displays data from two separate tables.

· EMPNO exists in the EMP table
· DEPTNO exists in both the EMP and DEPT the Tables.
· LOC exists in the DEPT table.

To produce the report. You need to link EMP and DEPT tables and access data from both of them.

```
SELECT e.empno, e.deptno, d.loc
FROM emp e inner join dept d
on e.deptno = d.deptno;
EQUIJOIN
SELECT emp.empno, emp.ename, emp.deptno,
dept.deptno, dept.loc
FROM emp, dept
WHERE emp.deptno = Dept.deptno;
```

**output**

| EMPNO | ENAME | DEPTNO | DEPTNO | LOC |
|---|---|---|---|---|
| 7698 | BLAKE | 30 | 30 | CHICAGO |
| 7369 | SMITH | 20 | 20 | DALLAS |

**Non-Equijoins**

The relationship between the EMP table and the SALGRADE table is a non-equijoin, meaning that no column in the EMP table corresponds directly to a column in the SALGRADE table.

The relationship between the two tables is that the SAL column in the EMP table is between the LOSAL and HISAL column of the SALGRADE table.

The relationship is obtained using an operator other than equal (=).

The slide example creates a non-equijoin to evaluate an employee's salary grade. The salary must be *between* any pair of the low and high salary ranges.

It is important to note that all employees appear exactly once when this query is executed. No employee is repeated in the list. There are two reasons for this:
None of the rows in the salary grade table contain grades that overlap. That is, the salary value for an employee can only lay between the low salary and high salary values of one of the rows in the salary grade table.
All of the employees' salaries lie within the limits provided *by* the salary grade table.
That is, no employee earns less than the lowest value contained in the LOSAL column or more than the highest value contained in the HISAL column.
**Note:** Other operators such as <= and >= could be used, but BETWEEN is the simplest. Remember to specify the low value first and the high value last when using BETWEEN. Table aliases have been specified for performance reasons, not because of possible ambiguity.

```
SELECT e.ename, e.sal, s.grade
FROM EMP e INNER JOIN SALGRADE s
ON e.sal +e.comm > s.hisal ;
```

Output:-

| ENAME | SAL | GRADE |
|---|---|---|
| ALLEN | 1600 | 1 |
| WARD | 1250 | 1 |
| MARTIN | 1250 | 1 |
| TURNER | 1500 | 1 |
| ALLEN | 1600 | 2 |
| WARD | 1250 | 2 |
| MARTIN | 1250 | 2 |
| TURNER | 1500 | 2 |
| MARTIN | 1250 | 3 |

**Self Joins**

**MGR in the WORKER table is equal to EMPNO in the MANAGER table.**

```
SELECT WORKER.ename, WORKER.empno ,
MANAGER.ename, MANAGER.empno
FROM emp WORKER, emp MANAGER
WHERE WORKER.mgr = MANAGER.empno ;
```

| ENAME | EMPNO | ENAME | EMPNO |
|---|---|---|---|
| JAMES | 7900 | BLAKE | 7698 |
| TURNER | 7844 | BLAKE | 7698 |
| MARTIN | 7654 | BLAKE | 7698 |
| WARD | 7521 | BLAKE | 7698 |
| ALLEN | 7499 | BLAKE | 7698 |
| FORD | 7902 | JONES | 7566 |
| SCOTT | 7788 | JONES | 7566 |
| MILLER | 7934 | CLARK | 7782 |
| ADAMS | 7876 | SCOTT | 7788 |
| CLARK | 7782 | KING | 7839 |
| JONES | 7566 | KING | 7839 |
| BLAKE | 7698 | KING | 7839 |
| SMITH | 7369 | FORD | 7902 |

**Order by**

SQL offers the user some control over the order in which tuples in a relation are displayed.

The **order by** clause causes the tuples in the result of a query to appear in sorted order.

To list in alphabetic order all customers who have a loan at the Perryridge branch, we write

**select distinct** *customer-name*
**from** *borrower, loan*
**where** *borrower.loan-number = loan.loan-number* **and**
*branch-name* = 'Perryridge' **order by** *customer-name*

By default, the **order by** clause lists items in ascending order. To specify the sort order, we may specify **desc** for descending order or **asc** for ascending order. Furthermore, ordering can be performed on multiple attributes. Suppose that we wish to list the entire *loan* relation in descending order of *amount*. If several loans have the same amount, we order them in ascending order by loan number.
We express this query in

SQL as follows:

**select** *
**from** *loan*
**order by** *amount* **desc**, *loan-number* **asc**

To fulfill an **order by** request, SQL must perform a sort. Since sorting a large number of tuples may be costly, it should be done only when necessary.

## The GROUP BY Statement

Aggregate functions often need an added GROUP BY statement. The GROUP BY statement is used in conjunction with the aggregate functions to group the result---set by one or more columns.

### Syntax

SELECT column_name, aggregate_function(column_name)

FROM table_name

WHERE column_name operator value

GROUP BY column_name

Example:

We use the CUSTOMER table as an example:

select FirstName, MAX(AreaCode) from CUSTOMER

**group by** FirstName

| | FirstName | (No column name) |
|---|---|---|
| 1 | John | 32 |
| 2 | Smith | 45 |

## The HAVING Clause

The HAVING clause was added to SQL because the WHERE keyword could not used with aggregate functions.

**Syntax:**

SELECT column_name, aggregate_function(column_name)

FROM table_name

WHERE column_name operator value

GROUP BY column_name

HAVING aggregate_function(column_name) operator value

**The following query:**

select CourseId, AVG(Grade) from GRADE

group by CourseId

**having AVG(Grade)>3**

| | CourseId | (No column name) |
|---|---|---|
| 1 | 1 | 4,5 |

**Transaction Concept**

A **transaction** is a **unit** of program execution that accesses and possibly updates various data items. To ensure integrity of the data, we require that the database system maintain the following properties of the transactions:

- **Atomicity**. Either all operations of the transaction are reflected properly in the database, or none are.

• **Consistency**. Execution of a transaction in isolation (that is, with no other transaction executing concurrently) preserves the consistency of the database.

• **Isolation**. Even though multiple transactions may execute concurrently, the system guarantees that, for every pair of transactions *Ti* and *Tj* , it appears to *Ti* that either *Tj* finished execution before *Ti* started, or *Tj* started execution after *Ti*

finished. Thus, each transaction is unaware of other transactions executing concurrently in the system.

• **Durability**. After a transaction completes successfully, the changes it has made to the database persist, even if there are system failures.

These properties are often called the **ACID properties**; the acronym is derived from the first letter of each of the four properties.

**Transaction State**

In the absence of failures, all transactions complete successfully. However, as we noted earlier, a transaction may not always complete its execution successfully. Such a transaction is termed **aborted**. If we are to ensure the atomicity property, an aborted transaction must have no effect on the state of the database. Thus, any changes that the aborted transaction made to the database must be undone. Once the changes caused by an aborted transaction have been undone, we say that the transaction has been **rolled back**. It is part of the responsibility of the recovery scheme to manage transaction aborts.

A transaction that completes its execution successfully is said to be **committed**.

A committed transaction that has performed updates transforms the database into a new consistent state, which must persist even if there is a system failure.

Once a transaction has committed, we cannot undo its effects by aborting it. The only way to undo the effects of a committed transaction is to execute a **compensating transaction**. For instance, if a transaction added $20 to an account, the compensating transaction would subtract $20 from the account. However, it is not always possible to create such a compensating transaction. Therefore, the responsibility of writing and executing a compensating transaction is left to the user, and is not handled by the database system.

A transaction must be in one of the following states:

• **Active**, the initial state; the transaction stays in this state while it is executing
• **Partially committed**, after the final statement has been executed
• **Failed**, after the discovery that normal execution can no longer proceed
• **Aborted**, after the transaction has been rolled back and the database has been restored to its state prior to the start of the transaction
• **Committed**, after successful completion

A transaction has committed only if it has entered the committed state. Similarly, we say that a transaction has aborted only if it has entered the aborted state. A transaction is said to have **terminated** if has either committed or aborted.

A transaction starts in the active state. When it finishes its final statement, it enters the partially committed state. At this point, the transaction has completed its execution, but it is still possible that it may have to be aborted, since the actual output may still be temporarily residing in main memory, and thus a hardware failure may preclude its successful completion.
The database system then writes out enough information to disk that, even in the event of a failure, the updates performed by the transaction can be re-created when the system restarts after the failure. When the last of this information is written out, the transaction enters the committed state.
A transaction enters the failed state after the system determines that the transaction can no longer proceed with its normal execution (for example, because of hardware or logical errors). Such a transaction must be rolled back. Then, it enters the aborted state.

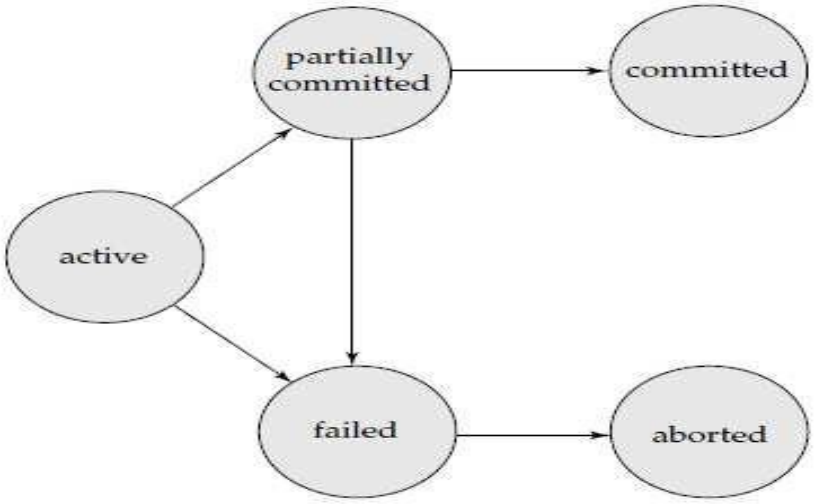



**Figure 15.1** State diagram of a transaction.

### Concurrent Executions

Transaction-processing systems usually allow multiple transactions to run concurrently. Allowing multiple transactions to update data concurrently causes several complications with consistency of the data, as we saw earlier. Ensuring consistency in spite of concurrent execution of transactions requires extra work; it is far easier to insist that transactions run **serially**—that is, one at a time, each starting only after the previous one has completed. However, there are two good reasons for allowing concurrency:

• **Improved throughput and resource utilization**. A transaction consists of many steps. Some involve I/O activity; others involve CPU activity. The CPU and the disks in a computer system can operate in parallel. Therefore, I/O activity can be done in parallel with processing at the CPU. The parallelism of the CPU and the I/O system can therefore be exploited to run multiple transactions in parallel. While a read or write on behalf of one transaction is in progress on one disk, another transaction can be running in the CPU, while another disk may be executing a read or write on behalf of a third transaction. All of this increases the **throughput** of the system—that is, the number of transactions executed in a given amount of time. Correspondingly, the processor and disk **utilization** also increase; in other words, the processor and disk spend less time idle, or not performing any useful work.

• **Reduced waiting time**. There may be a mix of transactions running on a system, some short and some long. If transactions run serially, a short transaction may have to wait for a preceding long transaction to complete, which can lead to unpredictable delays in running a transaction. If the transactions are operating on different parts of the database, it is better to let them run concurrently, sharing the CPU cycles and disk accesses among them. Concurrent execution reduces the unpredictable delays in running transactions. Moreover, it also reduces the **average response time**: the average time for a transaction to be completed after it has been submitted.

a set of transactions that access and update those accounts. Let $T1$ and $T2$ be two transactions that transfer funds from one account to another. Transaction $T1$ transfers \$50 from account $A$ to account $B$. It is defined as

$T1$: read($A$);

$A := A - 50$;

write($A$);

read($B$);

$B := B + 50$;

write($B$).

Transaction $T2$ transfers 10 percent of the balance from account $A$ to account $B$. It is

defined as

$T2$: read($A$);

$temp := A * 0.1$;

$A := A - temp$;

write($A$);

read($B$);

$B := B + temp$;

write($B$).

Suppose the current values of accounts $A$ and $B$ are \$1000 and \$2000, respectively.

Suppose also that the two transactions are executed one at a time in the order $T1$ followed by $T2$. This execution sequence appears in Figure 15.3. In the figure, the sequence of instruction steps is in chronological order from top to bottom, with instructions of $T1$ appearing in the left column and instructions of $T2$ are appearing in the right column. The final values of accounts $A$ and $B$, after the execution in takes place, are \$855 and \$2145, respectively.

| $T1$ | $T2$ |
|---|---|
| read($A$) | |
| $A := A - 50$ | |
| write ($A$) | |
| read($B$) | |
| $B := B + 50$ | |
| write($B$) | |
| | read($A$) |
| | $temp := A * 0.1$ |
| | $A := A - temp$ |
| | write($A$) |
| | read($B$) |
| | $B := B + temp$ |
| | write($B$) |

Schedule 1—a serial schedule in which $T_1$ is followed by $T_2$.

| $T_1$ | $T_2$ |
|---|---|
| read($A$) | |
| $A := A - 50$ | |
| write($A$) | |
| | read($A$) |
| | $temp := A * 0.1$ |
| | $A := A - temp$ |
| | write($A$) |
| read($B$) | |
| $B := B + 50$ | |
| write($B$) | |
| | read($B$) |
| | $B := B + temp$ |
| | write($B$) |

Schedule 3—a concurrent schedule equivalent to schedule 1.

## SERIALIZABILITY

The database system must control concurrent execution of transactions, to ensure that the database state remains consistent. . Serializability of schedules generated by concurrently executing transactions can be ensured through one of a variety of mechanisms called *concurrency control* schemes.

### 1. Conflict Serializability

Let us consider a schedule *S* in which there are two consecutive instructions *Ii* and *Ij*, of transactions *Ti* and *Tj* , respectively (*i* _= *j*). If *Ii* and *Ij* refer to different data items, then we can swap *Ii* and *Ij* without affecting the results of any instruction in the schedule. However, if *Ii* and *Ij* refer to the same data item *Q*, then the order of the two steps may matter. Since we are dealing with only read and write instructions, there are four cases that we need to consider:

1. *Ii* = read(*Q*), *Ij* = read(*Q*). The order of *Ii* and *Ij* does not matter, since the same value of *Q* is read by *Ti* and *Tj* , regardless of the order.

2. *Ii* = read(*Q*), *Ij* = write(*Q*). If *Ii* comes before *Ij*, then *Ti* does not read the value of *Q* that is written by *Tj* in instruction *Ij*. If *Ij* comes before *Ii*, then *Ti* reads the value of *Q* that is written by *Tj*. Thus, the order of *Ii* and *Ij* matters.

3. *Ii* = write(*Q*), *Ij* = read(*Q*). The order of *Ii* and *Ij* matters for reasons similar to those of the previous case.

4. *Ii* = write(*Q*), *Ij* = write(*Q*). Since both instructions are write operations, the order of these instructions does not affect either *Ti* or *Tj* . However, the value obtained by the next read(*Q*) instruction of *S* is affected, since the result of only the latter of the two write instructions is preserved in the database. If there is no other write(*Q*) instruction after *Ii* and *Ij* in *S*, then the order of *Ii* and *Ij* directly affects the final value of *Q* in the database state that results from schedule *S*.

**View Serializability**

A form of equivalence that is less stringent than conflict equivalence, but that, like conflict equivalence, is based on only the read and write operations of transactions.

| $T_1$ | $T_5$ |
|---|---|
| read($A$) | |
| $A := A - 50$ | |
| write($A$) | |
| | read($B$) |
| | $B := B - 10$ |
| | write($B$) |
| read($B$) | |
| $B := B + 50$ | |
| write($B$) | |
| | read($A$) |
| | $A := A + 10$ |
| | write($A$) |

Consider two schedules S and S', where the same set of transactions participates in both schedules. The schedules S and S' are said to be view equivalent if three conditions are met:

1. For each data item Q, if transaction Ti reads the initial value of Q in schedule S, then transaction Ti must, in schedule S', also read the initial value of Q.
2. For each data item Q, if transaction Ti executes read(Q) in schedule S, and if that value was produced by a write(Q) operation executed by transaction Tj , then the read(Q) operation of transaction Ti must, in schedule S', also read the value of Q that was produced by the same write(Q) operation of transaction Tj .
3. For each data item Q, the transaction (if any) that performs the final write(Q) operation in schedule S must perform the final write(Q) operation in schedule S'.

| $T_3$ | $T_4$ | $T_6$ |
|---|---|---|
| read($Q$) | | |
| | write($Q$) | |
| write($Q$) | | |
| | | write($Q$) |

12 Schedule 9—a view-serializable schedule.

# CHAPTER 4 ADVANCE SQL

## About Views

A view is a logical representation of another table or combination of tables. A view derives its data from the tables on which it is based. These tables are called base tables. Base tables might in turn be actual tables or might be views themselves. All operations performed on a view actually affect the base table of the view. You can use views in almost the same way as tables. You can query, update, insert into, and delete from views, just as you can standard tables.

Views can provide a different representation (such as subsets or supersets) of the data that resides within other tables and views. Views are very powerful because they allow you to tailor the presentation of data to different types of users.

## Creating Views

To create a view, you must meet the following requirements:

- To create a view in your schema, you must have the CREATE VIEW privilege. To create a view in another user's schema, you must have the CREATE ANY VIEW system privilege. You can acquire these privileges explicitly or through a role.
- The owner of the view (whether it is you or another user) must have been explicitly granted privileges to access all objects referenced in the view definition. The owner cannot have obtained these privileges through roles. Also, the functionality of the view depends on the privileges of the view owner. For example, if the owner of the view has only the INSERT privilege for Scott's emp table, then the view can be used only to insert new rows into the emp table, not to SELECT, UPDATE, or DELETE rows.
- If the owner of the view intends to grant access to the view to other users, the owner must have received the object privileges to the base objects with the GRANT OPTION or the system privileges with the ADMIN OPTION.

You can create views using the CREATE VIEW statement. Each view is defined by a query that references tables, materialized views, or other views. As with all subqueries, the query that defines a view cannot contain the FOR UPDATE clause.

The following statement creates a view on a subset of data in the emp table:

```
CREATE VIEW sales_staff AS
    SELECT empno, ename, deptno
    FROM emp
    WHERE deptno = 10
  WITH CHECK OPTION CONSTRAINT sales_staff_cnst;
```

The query that defines the sales_staff view references only rows in department 10. Furthermore, the CHECK OPTION creates the view with the constraint (named sales_staff_cnst) that INSERT and UPDATE statements issued against the view cannot result in rows that the view cannot select. For example, the following INSERT statement successfully inserts a row into the emp table by means of the sales_staff view, which contains all rows with department number 10:

```
INSERT INTO sales_staff VALUES (7584, 'OSTER', 10);
```

However, the following INSERT statement returns an error because it attempts to insert a row for department number 30, which cannot be selected using the sales_staff view:

```
INSERT INTO sales_staff VALUES (7591, 'WILLIAMS', 30);
```

The view could have been constructed specifying the WITH READ ONLY clause, which prevents any updates, inserts, or deletes from being done to the base table through the view. If no WITH clause is specified, the view, with some restrictions, is inherently updatable.

**Join Views**

You can also create views that specify more than one base table or view in the FROM clause. These are called join views. The following statement creates the division1_staff view that joins data from the emp and dept tables:

```
CREATE VIEW division1_staff AS

    SELECT ename, empno, job, dname

    FROM emp, dept
```

```
WHERE emp.deptno IN (10, 30)
  AND emp.deptno = dept.deptno;
```

Restrictions on DML operations for views use the following criteria in the order listed:

1. If a view is defined by a query that contains SET or DISTINCT operators, a GROUP BY clause, or a group function, then rows cannot be inserted into, updated in, or deleted from the base tables using the view.
2. If a view is defined with WITH CHECK OPTION, a row cannot be inserted into, or updated in, the base table (using the view), if the view cannot select the row from the base table.
3. If a NOT NULL column that does not have a DEFAULT clause is omitted from the view, then a row cannot be inserted into the base table using the view.
4. If the view was created by using an expression, such as DECODE(deptno, 10, "SALES", ...), then rows cannot be inserted into or updated in the base table using the view.

The constraint created by WITH CHECK OPTION of the sales_staff view only allows rows that have a department number of 30 to be inserted into, or updated in, the emp table. Alternatively, assume that the sales_staff view is defined by the following statement (that is, excluding the deptno column):

```
CREATE VIEW sales_staff AS
  SELECT empno, ename
  FROM emp
  WHERE deptno = 10
  WITH CHECK OPTION CONSTRAINT sales_staff_cnst;
```

**Altering Views**

You use the ALTER VIEW statement only to explicitly recompile a view that is invalid. If you want to change the definition of a view,

The ALTER VIEW statement lets you locate recompilation errors before run time. To ensure that the alteration does not affect the view or other objects that depend on it, you can explicitly recompile a view after altering one of its base tables.

To use the ALTER VIEW statement, the view must be in your schema, or you must have the ALTER ANY TABLE system privilege.

**Dropping Views**

You can drop any view contained in your schema. To drop a view in another user's schema, you must have the DROP ANY VIEW system privilege. Drop a view using the DROP VIEW statement. For example, the following statement drops the emp_dept view:

DROP VIEW emp_dept;

**Sequences**

Sequences are database objects from which multiple users can generate unique integers. The sequence generator generates sequential numbers, which can help to generate unique primary keys automatically, and to coordinate keys across multiple rows or tables.

Without sequences, sequential values can only be produced programmatically. A new primary key value can be obtained by selecting the most recently produced value and incrementing it. This method requires a lock during the transaction and causes multiple users to wait for the next value of the primary key; this waiting is known as serialization. If developers have such constructs in applications, then you should encourage the developers to replace them with access to sequences. Sequences eliminate serialization and improve the concurrency of an application.

**Creating Sequences**

To create a sequence in your schema, you must have the CREATE SEQUENCE system privilege. To create a sequence in another user's schema, you must have the CREATE ANY SEQUENCE privilege.

Create a sequence using the CREATE SEQUENCE statement. For example, the following statement creates a sequence used to generate employee numbers for the empno column of the emp table:

```
CREATE SEQUENCE emp_sequence
    INCREMENT BY 1
    START WITH 1
    NOMAXVALUE
    NOCYCLE
    CACHE 10;
```

Notice that several parameters can be specified to control the function of sequences. You can use these parameters to indicate whether the sequence is ascending or descending, the starting point of the sequence, the minimum and maximum values, and the interval between sequence values. The NOCYCLE option indicates that the sequence cannot generate more values after reaching its maximum or minimum value.

The CACHE clause preallocates a set of sequence numbers and keeps them in memory so that sequence numbers can be accessed faster. When the last of the sequence numbers in the cache has been used, the database reads another set of numbers into the cache.

The database might skip sequence numbers if you choose to cache a set of sequence numbers. For example, when an instance abnormally shuts down (for example, when an instance failure occurs or a SHUTDOWN ABORT statement is issued), sequence numbers that have been cached but not used are lost. Also, sequence numbers that have been used but not saved are lost as well. The database might also skip cached sequence numbers after an export and import.

| **OPTION** | **MEANING** |
|---|---|
| START WITH | Specifies the values at which sequence must start. Default is 1. |

MAXVALUE Maximum value the sequence can generate. Default is 10e27-1.

MINVALUE Minimum value the sequence can generate. Default is 1.

INCREMENT BY Specifies by how much the value of the sequence is to be incremented .If you want numbers in the descending order give negative value. Default is 1.

CYCLE Restarts numbers from MINVALUE after reaching MAXVALUE.

| **PSEUDO COLUMN** | **MEANING** |
|---|---|
| CURRVAL | Returns the current value of a sequence. |
| NEXTVAL | Returns the next value of a sequence. |
| NULL | Return a null value. |
| ROWID | Returns the ROWID of a row. See ROWID section below. |
| ROWNUM | Returns the number indicating in which order Oracle selects rows. |
| ROWNUM | First row selected will be ROWNUM of 1 and second row of 2 and so on. |
| SYSDATE | Returns current date and time. |
| USER | Returns the name of the current user. |
| UID | Returns the unique number assigned to the current user. |

**Altering Sequences**

To alter a sequence, your schema must contain the sequence, or you must have the ALTER ANY SEQUENCE system privilege. You can alter a sequence to change any of the parameters that define how it generates sequence numbers

except the sequence starting number. To change the starting point of a sequence, drop the sequence and then re-create it.

**Syntax**

ALTER SEQUENCE sequencename

[INCREMENT BY integer]

[MAXVALUE integer | NOMAXVALUE]

[MINVALUE integer | NOMINVALUE]

Alter a sequence using the ALTER SEQUENCE statement. For example, the following statement alters the emp_sequence:

ALTER SEQUENCE emp_sequence

INCREMENT BY 10

MAXVALUE 10000

CYCLE

CACHE 20;

**Using Sequences**

To use a sequence, your schema must contain the sequence or you must have been granted the SELECT object privilege for another user's sequence. Once a sequence is defined, it can be accessed and incremented by multiple users (who have SELECT object privilege for the sequence containing the sequence) with no waiting. The database does not wait for a transaction that has incremented a sequence to complete before that sequence can be incremented again.

The examples outlined in the following sections show how sequences can be used in master/detail table relationships. Assume an order entry system is partially comprised of two tables, orders_tab (master table) and line_items_tab (detail table), that hold information about customer orders. A sequence named order_seq is defined by the following statement:

```
CREATE SEQUENCE Order_seq
   START WITH 1
   INCREMENT BY 1
   NOMAXVALUE
   NOCYCLE
   CACHE 20;
```

**Caching Sequence Numbers**

Sequence numbers can be kept in the sequence cache in the System Global Area (SGA). Sequence numbers can be accessed more quickly in the sequence cache than they can be read from disk.

The sequence cache consists of entries. Each entry can hold many sequence numbers for a single sequence.

Follow these guidelines for fast access to all sequence numbers:

- Be sure the sequence cache can hold all the sequences used concurrently by your applications.
- Increase the number of values for each sequence held in the sequence cache.

**The Number of Entries in the Sequence Cache**

When an application accesses a sequence in the sequence cache, the sequence numbers are read quickly. However, if an application accesses a sequence that is not in the cache, then the sequence must be read from disk to the cache before the sequence numbers are used.

If your applications use many sequences concurrently, then your sequence cache might not be large enough to hold all the sequences. In this case, access to sequence numbers might often require disk reads. For fast access to all sequences, be sure your cache has enough entries to hold all the sequences used concurrently by your applications.

**The Number of Values in Each Sequence Cache Entry**

When a sequence is read into the sequence cache, sequence values are generated and stored in a cache entry. These values can then be accessed quickly. The number of sequence values stored in the cache is determined by the CACHE parameter in the CREATE SEQUENCE statement. The default value for this parameter is 20.

This CREATE SEQUENCE statement creates the seq2 sequence so that 50 values of the sequence are stored in the SEQUENCE cache:

```
CREATE SEQUENCE seq2
  CACHE 50;
```

Choosing a high value for CACHE lets you access more successive sequence numbers with fewer reads from disk to the sequence cache. However, if there is an instance failure, then all sequence values in the cache are lost. Cached sequence numbers also could be skipped after an export and import if transactions continue to access the sequence numbers while the export is running.

If you use the NOCACHE option in the CREATE SEQUENCE statement, then the values of the sequence are not stored in the sequence cache. In this case, every access to the sequence requires a disk read. Such disk reads slow access to the sequence. This CREATE SEQUENCE statement creates the SEQ3 sequence so that its values are never stored in the cache:

```
CREATE SEQUENCE seq3
  NOCACHE;
```

**Dropping Sequences**

You can drop any sequence in your schema. To drop a sequence in another schema, you must have the DROP ANY SEQUENCE system privilege. If a sequence is no longer required, you can drop the sequence using the DROP SEQUENCE statement. For example, the following statement drops the order_seq sequence:

```
DROP SEQUENCE order_seq;
```

When a sequence is dropped, its definition is removed from the data dictionary

**Synonyms**

A synonym is an alias for a schema object. Synonyms can provide a level of security by masking the name and owner of an object and by providing location transparency for remote objects of a distributed database. Also, they are convenient to use and reduce the complexity of SQL statements for database users.

Synonyms allow underlying objects to be renamed or moved, where only the synonym needs to be redefined and applications based on the synonym continue to function without modification.

You can create both public and private synonyms. A public synonym is owned by the special user group named PUBLIC and is accessible to every user in a database. A private synonym is contained in the schema of a specific user and available only to the user and the user's grantees.

**Creating Synonyms**

To create a private synonym in your own schema, you must have the CREATE SYNONYM privilege. To create a private synonym in another user's schema, you must have the CREATE ANY SYNONYM privilege. To create a public synonym, you must have the CREATE PUBLIC SYNONYM system privilege.

Create a synonym using the CREATE SYNONYM statement. The underlying schema object need not exist, nor do you need privileges to access the object. The following statement creates a public synonym named public_emp on the emp table contained in the schema of jward:

CREATE PUBLIC SYNONYM public_emp FOR jward.emp

When you create a synonym for a remote procedure or function, you must qualify the remote object with its schema name. Alternatively, you can create a local public synonym on the database where the remote object resides, in which

case the database link must be included in all subsequent calls to the procedure or function.

**Using Synonyms in DML Statements**

You can successfully use any private synonym contained in your schema or any public synonym, assuming that you have the necessary privileges to access the underlying object, either explicitly, from an enabled role, or from PUBLIC. You can also reference any private synonym contained in another schema if you have been granted the necessary object privileges for the private synonym.

You can only reference another user's synonym using the object privileges that you have been granted. For example, if you have the SELECT privilege for the jward.emp synonym, then you can query the jward.emp synonym, but you cannot insert rows using the jward.emp synonym.

A synonym can be referenced in a DML statement the same way that the underlying object of the synonym can be referenced. For example, if a synonym named emp refers to a table or view, then the following statement is valid:

```
INSERT INTO emp (empno, ename, job)
   VALUES (emp_sequence.NEXTVAL, 'SMITH', 'CLERK');
```

If the synonym named fire_emp refers to a standalone procedure or package procedure, then you could execute it with the command

```
EXECUTE Fire_emp(7344);
```

**Dropping Synonyms**

You can drop any private synonym in your own schema. To drop a private synonym in another user's schema, you must have the DROP ANY SYNONYM system privilege. To drop a public synonym, you must have the DROP PUBLIC SYNONYM system privilege.

Drop a synonym that is no longer required using DROP SYNONYM statement. To drop a private synonym, omit the PUBLIC keyword. To drop a public synonym, include the PUBLIC keyword.

For example, the following statement drops the private synonym named emp:

DROP SYNONYM emp;

The following statement drops the public synonym named public_emp:

DROP PUBLIC SYNONYM public_emp;

**Index**

All the characteristics of an index in a textbook will be found in an index of Oracle. The following are the characteristics of an index in Oracle.

- Index is used to search for required rows quickly.
- Index occupies extra space. Index is stored separately from table.
- Index contains the values of key – column on which index is created – in the ascending order.
- Just like the page number in book index, Oracle index stores ROWID – a unique value to internally identify each row of the table. For each entry in the index a key and corresponding ROWID are stored.
- Oracle uses index only when it feels the index is going to improve performance of the query.

**Why To Use An INDEX**

An index in Oracle is used for two purposes.

- To speed up searching for a particular value thereby improving performance of query.
- To enforce uniqueness

**Using index to improving performance**

Just like how you can quickly locate a particular topic in the book by using index at the end of the book, Oracle uses index to quickly locate the row with the

given value in the indexed column. Indexed column is the one on which index is created.

**Enforcing uniqueness with index**

An index may also be used to enforce uniqueness in the column(s) given in the index. Once a UNIQUE index is created, Oracle makes sure values in the indexed column(s) are unique.

**Creating an Index**

DDL command CREATE INDEX is used to create an index. The following is the syntax of this command.

CREATE [UNIQUE] INDEX index_name

ON table (column-1 [, column-2]...);

UNIQUE keyword is used to create a unique index. Unique index makes sure that the indexed column(s) is always unique.

To create an index on NAME column of STUDENTS table, enter:

**create index students_name_idx on students (name);**

If you want to create a unique index on NAME column of STUDENTS table,enter:

**create unique index students_name_idx on students(name);**

**Dropping an index**

You can drop an index using DROP INDEX command. It removes the named index. Removing an index will effect the performance of existing applications but not the functionality in any way.

Using and not using an index is transparent to users. Oracle decides whether to use or not on its own. However, it is possible for users to control the usage of index to certain extent using hints, which are directive to Oracle regarding how to execute a command. But hints are too heavy in a book meant for beginners.

**DROP INDEX indexname;**

The following example drops the index created on NAME of STUDENTS table.

```
drop index student_name_idx;
```

# CHAPTER 5----PL/SQL

The PL/SQL programming language was developed by Oracle Corporation in the late 1980s as procedural extension language for SQL and the Oracle relational database. Following are notable facts about PL/SQL:

- PL/SQL is a completely portable, high-performance transaction-processing language.
- PL/SQL provides a built-in interpreted and OS independent programming environment.
- PL/SQL can also directly be called from the command-line SQL*Plus interface.
- Direct call can also be made from external programming language calls to database.
- PL/SQL's general syntax is based on that of ADA and Pascal programming language.
- Apart from Oracle, PL/SQL is available in TimesTen in-memory database and IBM DB2.

**Features of PL/SQL**

PL/SQL has the following features:

- PL/SQL is tightly integrated with SQL.
- It offers extensive error checking.
- It offers numerous data types.
- It offers a variety of programming structures.
- It supports structured programming through functions and procedures.
- It supports object-oriented programming.
- It supports developing web applications and server pages.

## Advantages of PL/SQL

PL/SQL has the following advantages:

- SQL is the standard database language and PL/SQL is strongly integrated with SQL. PL/SQL supports both static and dynamic SQL. Static SQL supports DML operations and transaction control from PL/SQL block. Dynamic SQL is SQL allows embedding DDL statements in PL/SQL blocks.
- PL/SQL allows sending an entire block of statements to the database at one time. This reduces network traffic and provides high performance for the applications.
- PL/SQL gives high productivity to programmers as it can query, transform, and update data in a database.
- PL/SQL saves time on design and debugging by strong features, such as exception handling, encapsulation, data hiding, and object-oriented data types.
- Applications written in PL/SQL are fully portable.
- PL/SQL provides high security level.
- PL/SQL provides access to predefined SQL packages.
- PL/SQL provides support for Object-Oriented Programming.
- PL/SQL provides support for Developing Web Applications and Server Pages.

## Basic Syntax

PL/SQL is a block-structured language, meaning that PL/SQL programs are divided and written in logical blocks of code. Each block consists of three sub-parts:

| S.N. | Sections & Description |
|---|---|
| 1 | **Declarations**<br>This section starts with the keyword **DECLARE**. It is an optional section and defines all variables, cursors, subprograms, and other elements to be used in the program. |
| 2 | **Executable Commands**<br>This section is enclosed between the keywords **BEGIN** and **END** and it is a mandatory section. It consists of the executable PL/SQL statements of the program. It should have at least one executable line of code, which may be just a NULL command to indicate that nothing should be executed. |
| 3 | **Exception Handling**<br>This section starts with the keyword **EXCEPTION**. This section is again optional and contains exception(s) that handle errors in the program. |

Every PL/SQL statement ends with a semicolon **(;)**. PL/SQL blocks can be nested within other PL/SQL blocks using **BEGIN** and **END**. Here is the basic structure of a PL/SQL block:

**DECLARE**

**<declarations section>**

**BEGIN**

**<executable command(s)>**

**EXCEPTION**

**<exception handling>**

**END;**

**The 'Hello World' Example:**

```
DECLARE
-- variable declaration
message varchar2(20):= 'Hello, World!';
BEGIN
/*
* PL/SQL executable statement(s)
*/
dbms_output.put_line(message);
END;
/
```

**Data Types:-**

PL/SQL variables, constants and parameters must have a valid data type which specifies a storage format, constraints, and valid range of values. This tutorial will take you through SCALAR and LOB data types available in PL/SQL and other two data types.

| Category | Description |
| --- | --- |
| Scalar | Single values with no internal components, such as a NUMBER, DATE, or BOOLEAN. |
| Large Object (LOB) | Pointers to large objects that are stored separately from other data items, such as text, graphic images, video clips, and sound waveforms. |
| Composite | Data items that have internal components that can be accessed individually. For example, collections and records. |
| Reference | Pointers to other data items. |

| Date Type | Description |
| --- | --- |
| Numeric | Numeric values on which arithmetic operations are performed. |
| Character | Alphanumeric values that represent single characters or strings of characters. |
| Boolean | Logical values on which logical operations are performed. |
| Datetime | Dates and times. |

## PL/SQL Character Data Types and Subtypes

| Data Type | Description |
|---|---|
| CHAR | Fixed-length character string with maximum size of 32,767 bytes |
| VARCHAR2 | Variable-length character string with maximum size of 32,767 bytes |
| RAW | Variable-length binary or byte string with maximum size of 32,767 bytes, not interpreted by PL/SQL |
| NCHAR | Fixed-length national character string with maximum size of 32,767 bytes |
| NVARCHAR2 | Variable-length national character string with maximum size of 32,767 bytes |
| LONG | Variable-length character string with maximum size of 32,760 bytes |
| LONG RAW | Variable-length binary or byte string with maximum size of 32,760 bytes, not interpreted by PL/SQL |
| ROWID | Physical row identifier, the address of a row in an ordinary table |
| UROWID | Universal row identifier (physical, logical, or foreign row identifier) |

**Variable:-** A variable is nothing but a name given to a storage area that our programs can manipulate. Each variable in PL/SQL has a specific data type, which determines the size and layout of the variable's memory; the range of values that can be stored within that memory and the set of operations that can be applied to the variable.

### Variable Declaration in PL/SQL

PL/SQL variables must be declared in the declaration section or in a package as a global variable. When you declare a variable, PL/SQL allocates memory for the variable's value and the storage location is identified by the variable name.

The syntax for declaring a variable is:

**variable_name [CONSTANT] datatype [NOT NULL] [:= | DEFAULT initial_value]**

**The following example which makes use of various types of variables:**

```
DECLARE
a integer := 10;
b integer := 20;
c integer;
f  real;
BEGIN
c := a + b;
dbms_output.put_line('Value of c: ' || c);
f := 70.0/3.0;
dbms_output.put_line('Value of f: ' || f);END;
/
```

When the above code is executed, it produces the following result:

```
Value of c: 30
Value of f: 23.333333333333333
PL/SQL procedure successfully completed.
```

**Variable Scope in PL/SQL**

PL/SQL allows the nesting of Blocks, i.e., each program block may contain another inner block. If a variable is declared within an inner block, it is not accessible to the outer block. However, if a variable is declared and accessible to an outer Block, it is also accessible to all nested inner Blocks. There are two types of variable scope:

- **Local variables** - variables declared in an inner block and not accessible to outer blocks.
- **Global variables** - variables declared in the outermost block or a package.

**Following example shows the usage of Local and Global variables in its simple form:**

```
DECLARE
-- Global variables
num1 number := 95;
num2 number := 85;
BEGIN
dbms_output.put_line('Outer Variable num1: ' || num1);
dbms_output.put_line('Outer Variable num2: ' || num2);
DECLARE
-- Local variables
num1 number := 195;
num2 number := 185;
BEGIN
dbms_output.put_line('Inner Variable num1: ' || num1);
dbms_output.put_line('Inner Variable num2: ' || num2);
END;
END;
/
```

**When the above code is executed, it produces the following result:**

```
Outer Variable num1: 95
Outer Variable num2: 85
Inner Variable num1: 195
Inner Variable num2: 185
PL/SQL procedure successfully completed.
```

**Constants**

A constant holds a value that once declared, does not change in the program. A constant declaration specifies its name, data type, and value, and allocates storage for it. The declaration can also impose the NOT NULL constraint.

**Declaring a Constant**

A constant is declared using the CONSTANT keyword. It requires an initial value and does not allow that value to be changed.

**For example**:

```
PI CONSTANT NUMBER := 3.141592654;

DECLARE

-- constant declaration

pi constant number := 3.141592654;

-- other declarations

radius  number(5,2);

dia number(5,2);

circumference number(7, 2);

area number (10, 2);

BEGIN

-- processing

radius  :=  9.5;

dia := radius * 2;

circumference := 2.0 * pi * radius;

area := pi * radius * radius;

-- output

dbms_output.put_line('Radius: ' || radius);

dbms_output.put_line('Diameter: ' || dia);
```

```
dbms_output.put_line('Circumference: ' || circumference);
dbms_output.put_line('Area: ' || area);
END;
/
```

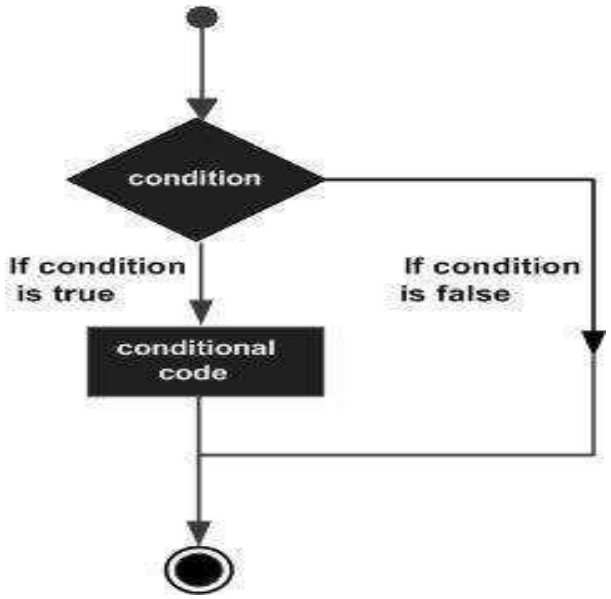
**When the above code is executed at SQL prompt, it produces the following result:**

```
Radius: 9.5
Diameter: 19
Circumference: 59.69
Area: 283.53
Pl/SQL procedure successfully completed.
```

## Conditions

Decision-making structures require that the programmer specify one or more conditions to be evaluated or tested by the program, along with a statement or statements to be executed if the condition is determined to be true, and optionally, other statements to be executed if the condition is determined to be false.

Following is the general from of a typical conditional (i.e., decision making) structure found in most of the programming languages:




| Statement | Description |
|---|---|
| IF - THEN statement | The **IF statement** associates a condition with a sequence of statements enclosed by the keywords **THEN** and **END IF**. If the condition is true, the statements get executed and if the condition is false or NULL then the IF statement does nothing. |
| IF-THEN-ELSE statement | **IF statement** adds the keyword **ELSE** followed by an alternative sequence of statement. If the condition is false or NULL , then only the alternative sequence of statements get executed. It ensures that either of the sequence of statements is executed. |
| IF-THEN-ELSIF statement | It allows you to choose between several alternatives. |
| Case statement | Like the IF statement, the **CASE statement** selects one sequence of statements to execute. However, to select the sequence, the CASE statement uses a selector rather than multiple Boolean expressions. A selector is an expression whose value is used to select one of several alternatives. |
| Searched CASE statement | The searched CASE statement **has no selector**, and it's WHEN clauses contain search conditions that yield Boolean values. |
| nested IF-THEN-ELSE | You can use one **IF-THEN** or **IF-THEN-ELSIF** statement inside another **IF-THEN** or **IF-THEN-ELSIF** statement(s). |

## IF - THEN statement

It is the simplest form of **IF** control statement, frequently used in decision making and changing the control flow of the program execution.

The **IF statement** associates a condition with a sequence of statements enclosed by the keywords **THEN** and **END IF**. If the condition is **TRUE**, the statements get executed, and if the condition is **FALSE** or **NULL,** then the **IF** statement does nothing.

**Syntax:**

Syntax for IF-THEN statement is:

IF condition THEN

S;

END IF;

**Where *condition* is a Boolean or relational condition and *S* is a simple or compound statement. Example of an IF-THEN statement is:**

IF (a <= 20) THEN

c:= c+1;

END IF;

If the Boolean expression *condition* evaluates to true, then the block of code inside the if statement will be executed. If Boolean expression evaluates to false, then the first set of code after the end of the if statement (after the closing end if) will be executed.

**Example 1:**

Let us try a complete example that would illustrate the concept:

DECLARE

a number(2) := 10;

BEGIN

a:= 10;

-- check the boolean condition using if statement

IF( a < 20 ) THEN

-- if condition is true then print the following

dbms_output.put_line('a is less than 20 ' );

END IF;

dbms_output.put_line('value of a is : ' || a);

END; /

**When the above code is executed at SQL prompt, it produces the following result:**

a is less than 20

value of a is : 10

PL/SQL procedure successfully completed.

**IF-THEN-ELSE statement**

A sequence of **IF-THEN** statements can be followed by an optional sequence of **ELSE** statements, which execute when the condition is **FALSE**.

**Syntax:**

Syntax for the IF-THEN-ELSE statement is:

IF condition THEN

S1;

ELSE

S2;

END IF;

Where, *S1* and *S2* are different sequence of statements. In the IF-THEN-ELSE statements, when the test *condition* is TRUE, the statement *S1* is executed and *S2* is skipped; when the test *condition* is FALSE, then *S1* is bypassed and statement *S2* is executed.

**For example:**

IF color = red THEN

dbms_output.put_line('You have chosen a red car')

ELSE

dbms_output.put_line('Please choose a color for your car');

END IF;

If the Boolean expression *condition* evaluates to true, then the if-then block of code will be executed, otherwise the else block of code will be executed.

**Example:**

Let us try a complete example that would illustrate the concept:

```
DECLARE
a number(3) := 100;
BEGIN
-- check the boolean condition using if statement
IF( a < 20 ) THEN
-- if condition is true then print the following
dbms_output.put_line('a is less than 20 ' );
ELSE
dbms_output.put_line('a is not less than 20 ' );
END IF;
dbms_output.put_line('value of a is : ' || a);
END;
/
```

**When the above code is executed at SQL prompt, it produces the following result:**

```
a is not less than 20
value of a is : 100
PL/SQL procedure successfully completed.
```

**IF-THEN-ELSIF statement**

The **IF-THEN-ELSIF** statement allows you to choose between several alternatives. An **IF-THEN** statement can be followed by an optional **ELSIF...ELSE** statement. The **ELSIF** clause lets you add additional conditions.

When using **IF-THEN-ELSIF** statements, there are few points to keep in mind.

- It's ELSIF, not ELSEIF

- An IF-THEN statement can have zero or one ELSE's and it must come after any ELSIF's.
- An IF-THEN statement can have zero to many ELSIF's and they must come before the
- ELSE.
- Once an ELSIF succeeds, none of the remaining ELSIF's or ELSE's will be tested.

**Syntax:**

The syntax of an IF-THEN-ELSIF Statement in PL/SQL programming language is:

IF(boolean_expression 1)THEN

S1; -- Executes when the boolean expression 1 is true

ELSIF( boolean_expression 2) THEN

S2; -- Executes when the boolean expression 2 is true

ELSIF( boolean_expression 3) THEN

S3; -- Executes when the boolean expression 3 is true

ELSE

S4; -- executes when the none of the above condition is true

END IF;

**Example:**

```
DECLARE
a number(3) := 100;
BEGIN
IF ( a = 10 ) THEN
dbms_output.put_line('Value of a is 10' );
ELSIF ( a = 20 ) THEN
dbms_output.put_line('Value of a is 20' );
ELSIF ( a = 30 ) THEN
dbms_output.put_line('Value of a is 30' );
ELSE
dbms_output.put_line('None of the values is matching');
END IF;
dbms_output.put_line('Exact value of a is: '|| a );
END;
/
```

**When the above code is executed at SQL prompt, it produces the following result:**

```
None of the values is matching
Exact value of a is: 100
PL/SQL procedure successfully completed.
```

**Case statement**

Like the **IF** statement, the **CASE statement** selects one sequence of statements to execute. However, to select the sequence, the **CASE** statement uses a selector rather than multiple Boolean expressions. A selector is an expression, whose value is used to select one of several alternatives.

**Syntax:**

The syntax for case statement in PL/SQL is:

```
CASE selector
WHEN 'value1' THEN S1;
WHEN 'value2' THEN S2;
WHEN 'value3' THEN S3;
...
ELSE Sn; -- default case
END CASE;
```

**Example:**

```
DECLARE
grade char(1) := 'A';
BEGIN
CASE grade
when 'A' then dbms_output.put_line('Excellent');
when 'B' then dbms_output.put_line('Very good');
when 'C' then dbms_output.put_line('Well done');
when 'D' then dbms_output.put_line('You passed');
when 'F' then dbms_output.put_line('Better try again');
else dbms_output.put_line('No such grade');
END CASE;
END;
/
```

**When the above code is executed at SQL prompt, it produces the following result:**

Excellent

PL/SQL procedure successfully completed.

**Loops**

There may be a situation when you need to execute a block of code several number of times. In general, statements are executed sequentially: The first statement in a function is executed first, followed by the second, and so on.

Programming languages provide various control structures that allow for more complicated execution paths.

A loop statement allows us to execute a statement or group of statements multiple times and following is the general form of a loop statement in most of the programming languages:

PL/SQL provides the following types

| Loop Type | Description |
|---|---|
| PL/SQL Basic LOOP | In this loop structure, sequence of statements is enclosed between the LOOP and END LOOP statements. At each iteration, the sequence of statements is executed and then control resumes at the top of the loop. |
| PL/SQL WHILE LOOP | Repeats a statement or group of statements until a given condition is true. It tests the condition before executing the loop body. |
| PL/SQL FOR LOOP | Execute a sequence of statements multiple times and abbreviates the code that manages the loop variable. |
| Nested loops in PL/SQL | You can use one or more loop inside any another basic loop, while or for loop. |

## PL/SQL Basic LOOP

Basic loop structure encloses sequence of statements in between the **LOOP** and **END LOOP** statements. With each iteration, the sequence of statements is executed and then control resumes at the top of the loop.

**Syntax:**

The syntax of a basic loop in PL/SQL programming language is:

```
LOOP

Sequence of statements;

END LOOP;
```

Here, sequence of statement(s) may be a single statement or a block of statements. An EXIT statement or an EXIT WHEN statement is required to break the loop.

**Example:**

```
DECLARE

x number := 10;

BEGIN

LOOP

dbms_output.put_line(x);

x := x + 10;

IF x > 50 THEN

exit;

END IF;

END LOOP;

-- after exit, control resumes here

dbms_output.put_line('After Exit x is: ' || x);
```

```
END;
/
```

**When the above code is executed at SQL prompt, it produces the following result:**

```
10
20
30
40
50
After Exit x is: 60
```

**You can use the EXIT WHEN statement instead of the EXIT statement:**

```
DECLARE
x number := 10;
BEGIN
LOOP
dbms_output.put_line(x);
x := x + 10;
exit WHEN x > 50;
END LOOP;
-- after exit, control resumes here
dbms_output.put_line('After Exit x is: ' || x);
END;
/
```

When the above code is executed at SQL prompt, it produces the following result:

10

20

30

40

50

After Exit x is: 60

PL/SQL procedure successfully completed.

**PL/SQL WHILE LOOP**

A **WHILE LOOP** statement in PL/SQL programming language repeatedly executes a target statement as long as a given condition is true.

**Syntax:**

WHILE condition LOOP

sequence_of_statements

END LOOP;

Example:

DECLARE

a number(2) := 10;

BEGIN

WHILE a < 20 LOOP

dbms_output.put_line('value of a: ' || a);

a := a + 1;

END LOOP;

END; /

**When the above code is executed at SQL prompt, it produces the following result:**

value of a: 10

value of a: 11

value of a: 12

value of a: 13

value of a: 14

value of a: 15

value of a: 16

value of a: 17

value of a: 18

value of a: 19

PL/SQL procedure successfully completed.

## PL/SQL FOR LOOP

A **FOR LOOP** is a repetition control structure that allows you to efficiently write a loop that needs to execute a specific number of times.

**Syntax:**

FOR counter IN initial_value .. final_value LOOP

sequence_of_statements;

END LOOP;

Here is the flow of control in a for loop:

- The initial step is executed first, and only once. This step allows you to declare and initialize any loop control variables.

- Next, the condition ,i.e., *initial_value .. final_value* is evaluated. If it is TRUE, the body of the loop is executed. If it is FALSE, the body of the loop does not execute and flow of control jumps to the next statement just after the for loop.
- After the body of the for loop executes, the value of the *counter* variable is increased or decreased.
- The condition is now evaluated again. If it is TRUE, the loop executes and the process repeats itself (body of loop, then increment step, and then again condition). After the condition becomes FALSE, the FOR-LOOP terminates.
- Following are some special characteristics of PL/SQL for loop:
- The *initial_value* and *final_value* of the loop variable or *counter* can be literals, variables, or expressions but must evaluate to numbers. Otherwise, PL/SQL raises the predefined exception VALUE_ERROR.
- The *initial_value* need not to be 1; however, the **loop counter increment (or decrement) must be 1**.
- PL/SQL allows determine the loop range dynamically at run time.

**Example:**

```
DECLARE
a number(2);
BEGIN
FOR a in 10 .. 20 LOOP
dbms_output.put_line('value of a: ' || a);
END LOOP;
END;
/
```

**When the above code is executed at SQL prompt, it produces the following result:**

value of a: 10

value of a: 11

value of a: 12

value of a: 13

value of a: 14

value of a: 15

value of a: 16

value of a: 17

value of a: 18

value of a: 19

value of a: 20

PL/SQL procedure successfully completed.

**Reverse FOR LOOP Statement**

By default, iteration proceeds from the initial value to the final value, generally upward from the lower bound to the higher bound. You can reverse this order by using the **REVERSE** keyword. In such case, iteration proceeds the other way. After each iteration, the loop counter is decremented.

However, you must write the range bounds in ascending (not descending) order. The following program illustrates this:

DECLARE

a number(2) ;

```
BEGIN
FOR a IN REVERSE 10 .. 20 LOOP
dbms_output.put_line('value of a: ' || a);
END LOOP;
END;
/
```

**When the above code is executed at SQL prompt, it produces the following result:**

```
value of a: 20
value of a: 19
value of a: 18
value of a: 17
value of a: 16
value of a: 15
value of a: 14
value of a: 13
value of a: 12
value of a: 11
value of a: 10
PL/SQL procedure successfully completed.
```

## The Loop Control Statements

Loop control statements change execution from its normal sequence. When execution leaves a scope, all automatic objects that were created in that scope are destroyed.

PL/SQL supports the following control statements. Labeling loops also helps in taking the control outside a loop. Click the following links to check their detail.

| Control Statement | Description |
|---|---|
| EXIT statement | The Exit statement completes the loop and control passes to the statement immediately after END LOOP |
| CONTINUE statement | Causes the loop to skip the remainder of its body and immediately retest its condition prior to reiterating. |
| GOTO statement | Transfers control to the labeled statement. Though it is not advised to use GOTO statement in your program. |

## EXIT statement

The **EXIT** statement in PL/SQL programming language has following two usages:

- When the EXIT statement is encountered inside a loop, the loop is immediately terminated and program control resumes at the next statement following the loop.
- If you are using nested loops (i.e. one loop inside another loop), the EXIT statement will stop the execution of the innermost loop and start executing the next line of code after the block.

**Syntax:**

The syntax for an EXIT statement in PL/SQL is as follows:

```
EXIT;
```

**Example:**

```
DECLARE
a number(2) := 10;
BEGIN
-- while loop execution
WHILE a < 20 LOOP
dbms_output.put_line ('value of a: ' || a);
a := a + 1;
IF a > 15 THEN
-- terminate the loop using the exit statement
EXIT;
END IF;
END LOOP;
END; /
```

**When the above code is executed at SQL prompt, it produces the following result:**

```
value of a: 10
value of a: 11
value of a: 12
value of a: 13
value of a: 14
value of a: 15
PL/SQL procedure successfully completed.
```

**The EXIT WHEN Statement**

The **EXIT-WHEN** statement allows the condition in the WHEN clause to be evaluated. If the condition is true, the loop completes and control passes to the statement immediately after END LOOP.

Following are two important aspects for the EXIT WHEN statement:

- Until the condition is true, the EXIT-WHEN statement acts like a NULL statement, except for evaluating the condition, and does not terminate the loop.
- A statement inside the loop must change the value of the condition.

**Syntax:**

The syntax for an EXIT WHEN statement in PL/SQL is as follows:

EXIT WHEN condition;

The EXIT WHEN statement **replaces a conditional statement like if-then** used with the EXIT statement.

**Example:**

```
DECLARE
a number(2) := 10;
BEGIN
-- while loop execution
WHILE a < 20 LOOP
dbms_output.put_line ('value of a: ' || a);
a := a + 1;
-- terminate the loop using the exit when statement
EXIT WHEN a > 15;
```

```
END LOOP;
END;
/
```

**When the above code is executed at SQL prompt, it produces the following result:**

```
value of a: 10
value of a: 11
value of a: 12
value of a: 13
value of a: 14
value of a: 15
PL/SQL procedure successfully completed.
```

**CONTINUE statement**

The **CONTINUE** statement causes the loop to skip the remainder of its body and immediately retest its condition prior to reiterating. In other words, it forces the next iteration of the loop to take place, skipping any code in between.

**Syntax:**

The syntax for a CONTINUE statement is as follows:

```
CONTINUE;
```

**Example**

```
DECLARE
a number(2) := 10;
BEGIN
-- while loop execution
```

```
WHILE a < 20 LOOP
dbms_output.put_line ('value of a: ' || a);
a := a + 1;
IF a = 15 THEN
-- skip the loop using the CONTINUE statement
a := a + 1;
CONTINUE;
END IF;
END LOOP;
END;
/
```

**When the above code is executed at SQL prompt, it produces the following result:**

```
value of a: 10
value of a: 11
value of a: 12
value of a: 13
value of a: 14
value of a: 16
value of a: 17
value of a: 18
value of a: 19
PL/SQL procedure successfully completed.
```

**GOTO statement**

A **GOTO** statement in PL/SQL programming language provides an unconditional jump from the GOTO to a labeled statement in the same subprogram.

**NOTE:** Use of GOTO statement is highly discouraged in any programming language because it makes difficult to trace the control flow of a program, making the program hard to understand and hard to modify. Any program that uses a GOTO can be rewritten so that it doesn't need the GOTO.

**Syntax:**

The syntax for a GOTO statement in PL/SQL is as follows:

```
GOTO label;

..

..

<< label >>

statement;
```

**Example:**

```
DECLARE

a number(2) := 10;

BEGIN

<<loopstart>>

-- while loop execution

WHILE a < 20 LOOP

dbms_output.put_line ('value of a: ' || a);

a := a + 1;

IF a = 15 THEN
```

```
a := a + 1;
GOTO loopstart;
END IF;
END LOOP;
END;
/
```

**When the above code is executed at SQL prompt, it produces the following result:**

```
value of a: 10
value of a: 11
value of a: 12
value of a: 13
value of a: 14
value of a: 16
value of a: 17
value of a: 18
value of a: 19
PL/SQL procedure successfully completed.
```

**Restrictions with GOTO Statement**

GOTO Statement in PL/SQL imposes the following restrictions:

- A GOTO statement cannot branch into an IF statement, CASE statement, LOOP statement or sub-block.
- A GOTO statement cannot branch from one IF statement clause to another or from one CASE statement WHEN clause to another.

- A GOTO statement cannot branch from an outer block into a sub-block (that is, an inner BEGIN-END block).
- A GOTO statement cannot branch out of a subprogram. To end a subprogram early, either use the RETURN statement or have GOTO branch to a place right before the end of the subprogram.
- A GOTO statement cannot branch from an exception handler back into the current BEGIN-END block. However, a GOTO statement can branch from an exception handler into an enclosing block.

**Procedures**

A **subprogram** is a program unit/module that performs a particular task. These subprograms are combined to form larger programs. This is basically called the 'Modular design'. A subprogram can be invoked by another subprogram or program which is called the calling program.

A subprogram can be created:

- At schema level
- Inside a package
- Inside a PL/SQL block

A schema level subprogram is a **standalone subprogram**. It is created with the CREATE PROCEDURE or CREATE FUNCTION statement. It is stored in the database and can be deleted with the DROP PROCEDURE or DROP FUNCTION statement.

A subprogram created inside a package is a **packaged subprogram**. It is stored in the database and can be deleted only when the package is deleted with the DROP PACKAGE statement.

PL/SQL subprograms are named PL/SQL blocks that can be invoked with a set of parameters. PL/SQL provides two kinds of subprograms:

 **Functions**: these subprograms return a single value, mainly used to compute and return a value.

 **Procedures**: these subprograms do not return a value directly, mainly used to perform an action.

| S.N. | Parts & Description |
|---|---|
| 1 | **Declarative Part**<br>It is an optional part. However, the declarative part for a subprogram does not start with the DECLARE keyword. It contains declarations of types, cursors, constants, variables, exceptions, and nested subprograms. These items are local to the subprogram and cease to exist when the subprogram completes execution. |
| 2 | **Executable Part**<br>This is a mandatory part and contains statements that perform the designated action. |
| 3 | **Exception-handling**<br>This is again an optional part. It contains the code that handles run-time errors. |

**Creating a Procedure**

A procedure is created with the CREATE OR REPLACE PROCEDURE statement. The simplified syntax for the CREATE OR REPLACE PROCEDURE statement is as follows: **Syntax**

CREATE [OR REPLACE] PROCEDURE procedure_name

[(parameter_name [IN | OUT | IN OUT] type [, ...])]

{IS | AS}

BEGIN

< procedure_body >

END procedure_name;

Where,

- *procedure-name* specifies the name of the procedure.
- [OR REPLACE] option allows modifying an existing procedure.
- The optional parameter list contains name, mode and types of the parameters. IN represents that value will be passed from outside and OUT represents that this parameter will be used to return a value outside of the procedure.
- *procedure-body* contains the executable part.
- The AS keyword is used instead of the IS keyword for creating a standalone procedure.

**Example:**

**The following example creates a simple procedure that displays the string 'Hello World!' on the screen when executed.**

```
CREATE OR REPLACE PROCEDURE greetings

AS

BEGIN

dbms_output.put_line('Hello World!');

END;

/
```

**When above code is executed using SQL prompt, it will produce the following result:**

```
Procedure created.
```

**Executing a Standalone Procedure**

A standalone procedure can be called in two ways:

- Using the EXECUTE keyword
- Calling the name of the procedure from a PL/SQL block

**The above procedure named 'greetings' can be called with the EXECUTE keyword as**:

EXECUTE greetings;

**The above call would display:**

Hello World

PL/SQL procedure successfully completed.

**The procedure can also be called from another PL/SQL block:**

BEGIN

greetings;

END;

/

**The above call would display:**

Hello World

PL/SQL procedure successfully completed.

**Deleting a Standalone Procedure**

A standalone procedure is deleted with the DROP PROCEDURE statement. **Syntax** for deleting a procedure is:

DROP PROCEDURE procedure-name;

**So you can drop *greetings* procedure by using the following statement:**

BEGIN

DROP PROCEDURE greetings;

END;

/

**IN & OUT Mode Example 1**

**This program finds the minimum of two values, here procedure takes two numbers using IN mode and returns their minimum using OUT parameters.**

```
DECLARE
a number;
b number;
c number;
PROCEDURE findMin(x IN number, y IN number, z OUT number) IS
BEGIN
IF x < y THEN
z:= x;
ELSE
z:= y;
END IF;
END;
BEGIN
a:= 23;
b:= 45;
findMin(a, b, c);
dbms_output.put_line(' Minimum of (23, 45) : ' || c);
END;
/
```

**When the above code is executed at SQL prompt, it produces the following result:**

Minimum of (23, 45) : 23

PL/SQL procedure successfully completed.

**Functions**

APL/SQL function is same as a procedure except that it returns a value. Therefore, all the discussions of the previous chapter are true for functions too.

Creating a Function

A standalone function is created using the CREATE FUNCTION statement. The simplified syntax for the CREATE OR REPLACE PROCEDURE statement is as follows:

CREATE [OR REPLACE] FUNCTION function_name

[(parameter_name [IN | OUT | IN OUT] type [, ...])]

RETURN return_datatype

{IS | AS}

BEGIN

< function_body >

END [function_name];

Where,

- *Function-name* specifies the name of the function.
- [OR REPLACE] option allows modifying an existing function.
- The optional parameter list contains name, mode and types of the parameters. IN represents that value will be passed from outside and OUT represents that this parameter will be used to return a value outside of the procedure.
- The function must contain a **return** statement.

- *RETURN* clause specifies that data type you are going to return from the function.
- *Function-body* contains the executable part.
- The AS keyword is used instead of the IS keyword for creating a standalone function.

**Example:**

The following example illustrates creating and calling a standalone function. This function returns the total number of CUSTOMERS in the customers table.

```
Select * from customers;

+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
|  1 | Ramesh   |  32 | Ahmedabad |  2000.00 |
|  2 | Khilan   |  25 | Delhi     |  1500.00 |
|  3 | kaushik  |  23 | Kota      |  2000.00 |
|  4 | Chaitali |  25 | Mumbai    |  6500.00 |
|  5 | Hardik   |  27 | Bhopal    |  8500.00 |
|  6 | Komal    |  22 | MP        |  4500.00 |
+----+----------+-----+-----------+----------+
```

```
CREATE OR REPLACE FUNCTION totalCustomers

RETURN number IS

total number(2) := 0;

BEGIN

SELECT count(*) into total

FROM customers;

RETURN total;

END;
```

/

**When above code is executed using SQL prompt, it will produce the following result:**

Function created.

**Calling a Function**

While creating a function, you give a definition of what the function has to do. To use a function, you will have to call that function to perform the defined task. When a program calls a function, program control is transferred to the called function.

A called function performs defined task and when its return statement is executed or when it last end statement is reached, it returns program control back to the main program.

To call a function you simply need to pass the required parameters along with function name and if function returns a value then you can store returned value. Following program calls the function total Customers from an anonymous block:

```
DECLARE

c number(2);

BEGIN

c := totalCustomers();

dbms_output.put_line('Total no. of Customers: ' || c);

END; /
```

**When the above code is executed at SQL prompt, it produces the following result:**

Total no. of Customers: 6

PL/SQL procedure successfully completed.

**Example:**

The following is one more example which demonstrates Declaring, Defining, and Invoking a Simple PL/SQL Function that computes and returns the maximum of two values.

```
DECLARE

a number;

b number;

c number;

FUNCTION findMax(x IN number, y IN number)

RETURN number

IS

z number;

BEGIN

IF x > y THEN

z:= x;

ELSE

Z:= y;

END IF;

RETURN z;

END;

BEGIN

a:= 23;

b:= 45;

c := findMax(a, b);
```

```
dbms_output.put_line(' Maximum of (23,45): ' || c);

END;

/
```

**When the above code is executed at SQL prompt, it produces the following result:**

```
Maximum of (23,45): 78

PL/SQL procedure successfully completed.
```

**PL/SQL Recursive Functions**

We have seen that a program or subprogram may call another subprogram. When a subprogram calls itself, it is referred to as a recursive call and the process is known as recursion.

To illustrate the concept, let us calculate the factorial of a number. Factorial of a number n is defined as:

```
n! = n*(n-1)!

= n*(n-1)*(n-2)!

...

= n*(n-1)*(n-2)*(n-3)... 1
```

**The following program calculates the factorial of a given number by calling itself recursively:**

```
DECLARE

num number;

factorial number;

FUNCTION fact(x number)

RETURN number
```

```
IS
f number;
BEGIN
IF x=0 THEN
f := 1;
ELSE
f := x * fact(x-1);
END IF;
RETURN f;
END;
BEGIN
num:= 6;
factorial := fact(num);
dbms_output.put_line(' Factorial '|| num || ' is ' || factorial);
END; /
```

**When the above code is executed at SQL prompt, it produces the following result:**

```
Factorial 6 is 720
PL/SQL procedure successfully completed.
```

**Cursors**

Oracle creates a memory area, known as context area, for processing an SQL statement, which contains all information needed for processing the statement, for example, number of rows processed, etc.

A cursor is a pointer to this context area. PL/SQL controls the context area through a cursor. A cursor holds the rows (one or more) returned by a SQL statement. The set of rows the cursor holds is referred to as the **active set**.

You can name a cursor so that it could be referred to in a program to fetch and process the rows returned by the SQL statement, one at a time. There are two types of cursors:

- Implicit cursors
- Explicit cursors

## Implicit Cursors

Implicit cursors are automatically created by Oracle whenever an SQL statement is executed, when there is no explicit cursor for the statement. Programmers cannot control the implicit cursors and the information in it.

Whenever a DML statement (INSERT, UPDATE and DELETE) is issued, an implicit cursor is associated with this statement. For INSERT operations, the cursor holds the data that needs to be inserted. For UPDATE and DELETE operations, the cursor identifies the rows that would be affected.

In PL/SQL, you can refer to the most recent implicit cursor as the **SQL cursor**, which always has the attributes like %FOUND, %ISOPEN, %NOTFOUND, and %ROWCOUNT. The SQL cursor has additional attributes, %BULK_ROWCOUNT and %BULK_EXCEPTIONS, designed for use with the FORALL statement. The following table provides the description of the most used attributes:

| Attribute | Description |
|---|---|
| %FOUND | Returns TRUE if an INSERT, UPDATE, or DELETE statement affected |

| | |
|---|---|
| | one or more rows or a SELECT INTO statement returned one or more rows. Otherwise, it returns FALSE. |
| %NOTFOUND | The logical opposite of %FOUND. It returns TRUE if an INSERT, UPDATE, or DELETE statement affected no rows, or a SELECT INTO statement returned no rows. Otherwise, it returns FALSE. |
| %ISOPEN | Always returns FALSE for implicit cursors, because Oracle closes the SQL cursor automatically after executing its associated SQL statement. |
| %ROWCOUNT | Returns the number of rows affected by an INSERT, UPDATE, or DELETE statement, or returned by a SELECT INTO statement. |

Example:

We will be using the CUSTOMERS table we had created and used in the previous chapters.

```
Select * from customers;

+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
|  1 | Ramesh   |  32 | Ahmedabad |  2000.00 |
|  2 | Khilan   |  25 | Delhi     |  1500.00 |
|  3 | kaushik  |  23 | Kota      |  2000.00 |
|  4 | Chaitali |  25 | Mumbai    |  6500.00 |
|  5 | Hardik   |  27 | Bhopal    |  8500.00 |
|  6 | Komal    |  22 | MP        |  4500.00 |
+----+----------+-----+-----------+----------+
```

**The following program would update the table and increase salary of each customer by 500 and use the SQL%ROWCOUNT attribute to determine the number of rows affected:**

DECLARE

total_rows number(2);

```
BEGIN
UPDATE customers
SET salary = salary + 500;
IF sql%notfound THEN
dbms_output.put_line('no customers selected');
ELSIF sql%found THEN
total_rows := sql%rowcount;
dbms_output.put_line( total_rows || ' customers selected ');
END IF;
END; /
```

**When the above code is executed at SQL prompt, it produces the following result:**

```
6 customers selected
PL/SQL procedure successfully completed.
```

**If you check the records in customers table, you will find that the rows have been updated:**

```
Select * from customers;
+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
|  1 | Ramesh   |  32 | Ahmedabad |  2500.00 |
|  2 | Khilan   |  25 | Delhi     |  2000.00 |
|  3 | kaushik  |  23 | Kota      |  2500.00 |
|  4 | Chaitali |  25 | Mumbai    |  7000.00 |
|  5 | Hardik   |  27 | Bhopal    |  9000.00 |
|  6 | Komal    |  22 | MP        |  5000.00 |
+----+----------+-----+-----------+----------+
```

**Explicit Cursors**

Explicit cursors are programmer defined cursors for gaining more control over the **context area**. An explicit cursor should be defined in the declaration section of the PL/SQL Block. It is created on a SELECT Statement which returns more than one row.

**The syntax for creating an explicit cursor is**

CURSOR cursor_name IS select_statement;

**Working with an explicit cursor involves four steps:**

- Declaring the cursor for initializing in the memory
- Opening the cursor for allocating memory
- Fetching the cursor for retrieving data
- Closing the cursor to release allocated memory

**Declaring the Cursor**

Declaring the cursor defines the cursor with a name and the associated SELECT statement. For example:

CURSOR c_customers IS

SELECT id, name, address FROM customers;

**Opening the Cursor**

Opening the cursor allocates memory for the cursor and makes it ready for fetching the rows returned by the SQL statement into it. For example, we will open above-defined cursor as follows:

OPEN c_customers;

**Fetching the Cursor**

Fetching the cursor involves accessing one row at a time. For example we will fetch rows from the above-opened cursor as follows:

FETCH c_customers INTO c_id, c_name, c_addr;

**Closing the Cursor**

Closing the cursor means releasing the allocated memory. For example, we will close above-opened cursor as follows:

```
CLOSE c_customers;
```

**Example:**

Following is a complete example to illustrate the concepts of explicit cursors:

```
DECLARE
c_id  customers.id%type;
c_name customers.name%type;
c_addr customers.address%type;
CURSOR c_customers is
SELECT id, name, address FROM customers;
BEGIN
OPEN c_customers;
LOOP
FETCH c_customers into c_id, c_name, c_addr;
dbms_output.put_line(c_id || ' ' || c_name || ' ' || c_addr);
EXIT WHEN c_customers%notfound;
END LOOP;
CLOSE c_customers;
END;
/
```

**When the above code is executed at SQL prompt, it produces the following result:**

1 Ramesh Ahmedabad

2 Khilan Delhi

3 kaushik Kota

4 Chaitali Mumbai

5 Hardik Bhopal

6 Komal MP

PL/SQL procedure successfully completed.

## Exceptions

An error condition during a program execution is called an exception in PL/SQL. PL/SQL supports programmers to catch such conditions using **EXCEPTION** block in the program and an appropriate action is taken against the error condition. There are two types of exceptions:

- System-defined exceptions
- User-defined exceptions

## Syntax for Exception Handling

The General Syntax for exception handling is as follows. Here, you can list down as many as exceptions you want to handle. The default exception will be handled using *WHEN others THEN*:

DECLARE

<declarations section>

BEGIN

<executable command(s)>

EXCEPTION

<exception handling goes here >

WHEN exception1 THEN

exception1-handling-statements

WHEN exception2 THEN

exception2-handling-statements

WHEN exception3 THEN

exception3-handling-statements

.....

WHEN others THEN

exception3-handling-statements

END;

**Example**

Let us write some simple code to illustrate the concept. We will be using the CUSTOMERS table we had created and used in the previous chapters:

DECLARE

c_id customers.id%type := 8;

c_name customers.name%type;

c_addr customers.address%type;

BEGIN

SELECT name, address INTO c_name, c_addr

FROM customers

```
WHERE id = c_id;

DBMS_OUTPUT.PUT_LINE ('Name: '|| c_name);

DBMS_OUTPUT.PUT_LINE ('Address: ' || c_addr);

EXCEPTION

WHEN no_data_found THEN

dbms_output.put_line('No such customer!');

WHEN others THEN

dbms_output.put_line('Error!');

END;

/
```

**When the above code is executed at SQL prompt, it produces the following result:**

```
No such customer!

PL/SQL procedure successfully completed.
```

The above program displays the name and address of a customer whose ID is given. Since there is no customer with ID value 8 in our database, the program raises the run-time exception **NO_DATA_FOUND,** which is captured in **EXCEPTION** block.

**Raising Exceptions**

Exceptions are raised by the database server automatically whenever there is any internal database error, but exceptions can be raised explicitly by the programmer by using the command **RAISE**. Following is the simple syntax of raising an exception:

```
DECLARE

exception_name EXCEPTION;
```

```
BEGIN

IF condition THEN

RAISE exception_name;

END IF;

EXCEPTION

WHEN exception_name THEN

statement;

END;
```

You can use above syntax in raising Oracle standard exception or any user-defined exception. Next section will give you an example on raising user-defined exception, similar way you can raise Oracle standard exceptions as well.

**User-defined Exceptions**

PL/SQL allows you to define your own exceptions according to the need of your program. A user-defined exception must be declared and then raised explicitly, using either a RAISE statement or the procedure DBMS_STANDARD.RAISE_APPLICATION_ERROR.

The syntax for declaring an exception is:

```
DECLARE

my-exception EXCEPTION;
```

**Example:**

The following example illustrates the concept. This program asks for a customer ID, when the user enters an invalid ID, the exception invalid_id is raised.

```
DECLARE

c_id customers.id%type := &cc_id;

c_name customers.name%type;
```

```
c_addr customers.address%type;
-- user defined exception
ex_invalid_id EXCEPTION;
BEGIN
IF c_id <= 0 THEN
RAISE ex_invalid_id;
ELSE
SELECT name, address INTO c_name, c_addr
FROM customers
WHERE id = c_id;
DBMS_OUTPUT.PUT_LINE ('Name: '|| c_name);
DBMS_OUTPUT.PUT_LINE ('Address: ' || c_addr);
END IF;
EXCEPTION
WHEN ex_invalid_id THEN
dbms_output.put_line('ID must be greater than zero!');
WHEN no_data_found THEN
dbms_output.put_line('No such customer!');
WHEN others THEN
dbms_output.put_line('Error!');
END;
/
```

**When the above code is executed at SQL prompt, it produces the following result:**

Enter value for cc_id: -6 (let's enter a value -6)

old 2: c_id customers.id%type := &cc_id;

new 2: c_id customers.id%type := -6;

ID must be greater than zero!

PL/SQL procedure successfully completed.

**Pre-defined Exceptions**

PL/SQL provides many pre-defined exceptions, which are executed when any database rule is violated by a program. For example, the predefined exception NO_DATA_FOUND is raised when a SELECT INTO st

| Exception | Oracle Error | SQLCODE | Description |
|---|---|---|---|
| ACCESS_INTO_NULL | 06530 | -6530 | It is raised when a null object is automatically assigned a value. |
| CASE_NOT_FOUND | 06592 | -6592 | It is raised when none of the choices in the WHEN clauses of a CASE statement is selected, and there is no ELSE clause. |
| COLLECTION_IS_NULL | 06531 | -6531 | It is raised when a program attempts to apply collection methods other than EXISTS to an uninitialized nested table or varray, or the program attempts to assign values to the elements of an uninitialized nested table or varray. |
| DUP_VAL_ON_INDEX | 00001 | -1 | It is raised when duplicate values are attempted to be stored in a column with unique index. |
| INVALID_CURSOR | 01001 | -1001 | It is raised when attempts are made to make a cursor operation that is not allowed, such as closing an unopened cursor. |

| | | | |
|---|---|---|---|
| INVALID_NUMBER | 01722 | -1722 | It is raised when the conversion of a character string into a number fails because the string does not represent a valid number. |
| LOGIN_DENIED | 01017 | -1017 | It is raised when s program attempts to log on to the database with an invalid username or password. |
| NO_DATA_FOUND | 01403 | +100 | It is raised when a SELECT INTO statement returns no rows. |
| NOT_LOGGED_ON | 01012 | -1012 | It is raised when a database call is issued without being connected to the database. |
| PROGRAM_ERROR | 06501 | -6501 | It is raised when PL/SQL has an internal problem. |
| ROWTYPE_MISMATCH | 06504 | -6504 | It is raised when a cursor fetches value in a variable having incompatible data type. |
| SELF_IS_NULL | 30625 | -30625 | It is raised when a member method is invoked, but the instance of the object type was not initialized. |
| STORAGE_ERROR | 06500 | -6500 | It is raised when PL/SQL ran out of memory or memory was corrupted. |
| TOO_MANY_ROWS | 01422 | -1422 | It is raised when s SELECT INTO statement returns more than one row. |
| VALUE_ERROR | 06502 | -6502 | It is raised when an arithmetic, conversion, truncation, or size-constraint error occurs. |
| ZERO_DIVIDE | 01476 | 1476 | It is raised when an attempt is made to divide a number by zero. |

## Triggers

Triggers are stored programs, which are automatically executed or fired when some events occur. Triggers are, in fact, written to be executed in response to any of the following events:

- A database manipulation (DML) statement (DELETE, INSERT, or UPDATE).
- A database definition (DDL) statement (CREATE, ALTER, or DROP).
- A database operation (SERVERERROR, LOGON, LOGOFF, STARTUP, or SHUTDOWN).

Triggers could be defined on the table, view, schema, or database with which the event is associated.

## Benefits of Triggers

Triggers can be written for the following purposes:

- Generating some derived column values automatically
- Enforcing referential integrity
- Event logging and storing information on table access
- Auditing
- Synchronous replication of tables
- Imposing security authorizations
- Preventing invalid transactions

## Creating Triggers

The syntax for creating a trigger is:

CREATE [OR REPLACE ] TRIGGER trigger_name

{BEFORE | AFTER | INSTEAD OF }

{INSERT [OR] | UPDATE [OR] | DELETE}

[OF col_name]

ON table_name

[REFERENCING OLD AS o NEW AS n]

[FOR EACH ROW]

WHEN (condition)

DECLARE

Declaration-statements

BEGIN

Executable-statements

EXCEPTION

Exception-handling-statements

END;

Where,

- CREATE [OR REPLACE] TRIGGER trigger_name: Creates or replaces an existing trigger with the *trigger_name*.
- {BEFORE | AFTER | INSTEAD OF}: This specifies when the trigger would be executed. The INSTEAD OF clause is used for creating trigger on a view.
- {INSERT [OR] | UPDATE [OR] | DELETE}: This specifies the DML operation.
- [OF col_name]: This specifies the column name that would be updated.
- [ON table_name]: This specifies the name of the table associated with the trigger.
- [REFERENCING OLD AS o NEW AS n]: This allows you to refer new and old values for various DML statements, like INSERT, UPDATE, and DELETE.

- [FOR EACH ROW]: This specifies a row level trigger, i.e., the trigger would be executed for each row being affected. Otherwise the trigger will execute just once when the SQL statement is executed, which is called a table level trigger.
- WHEN (condition): This provides a condition for rows for which the trigger would fire. This clause is valid only for row level triggers.

**Example:**

To start with, we will be using the CUSTOMERS table we had created and used in the previous chapters:

```
Select * from customers;

+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
|  1 | Ramesh   |  32 | Ahmedabad |  2500.00 |
|  2 | Khilan   |  25 | Delhi     |  2000.00 |
|  3 | kaushik  |  23 | Kota      |  2500.00 |
|  4 | Chaitali |  25 | Mumbai    |  7000.00 |
|  5 | Hardik   |  27 | Bhopal    |  9000.00 |
|  6 | Komal    |  22 | MP        |  5000.00 |
+----+----------+-----+-----------+----------+
```

The following program creates a row level trigger for the customers table that would fire for INSERT or UPDATE or DELETE operations performed on the CUSTOMERS table. This trigger will display the salary difference between the old values and new values:

```
CREATE OR REPLACE TRIGGER display_salary_changes

BEFORE DELETE OR INSERT OR UPDATE ON customers

FOR EACH ROW

WHEN (NEW.ID > 0)

DECLARE

sal_diff number;

BEGIN
```

```
sal_diff := :NEW.salary - :OLD.salary;

dbms_output.put_line('Old salary: ' || :OLD.salary);

dbms_output.put_line('New salary: ' || :NEW.salary);

dbms_output.put_line('Salary difference: ' || sal_diff);END;

/
```

**When the above code is executed at SQL prompt, it produces the following result:**

Trigger created.

Here following two points are important and should be noted carefully:

- OLD and NEW references are not available for table level triggers, rather you can use them for record level triggers.
- If you want to query the table in the same trigger, then you should use the AFTER keyword, because triggers can query the table or change it again only after the initial changes are applied and the table is back in a consistent state.
- Above trigger has been written in such a way that it will fire before any DELETE or INSERT or UPDATE operation on the table, but you can write your trigger on a single or multiple operations, for example BEFORE DELETE, which will fire whenever a record will be deleted using DELETE operation on the table.

**Triggering a Trigger**

Let us perform some DML operations on the CUSTOMERS table. Here is one INSERT statement, which will create a new record in the table:

```
INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY)

VALUES (7, 'Kriti', 22, 'HP', 7500.00 );
```

**When a record is created in CUSTOMERS table, above create trigger display_salary_changes will be fired and it will display the following result:**

Old salary:

New salary: 7500

Salary difference:

**Because this is a new record so old salary is not available and above result is coming as null. Now, let us perform one more DML operation on the CUSTOMERS table. Here is one UPDATE statement, which will update an existing record in the table:**

```
UPDATE customers
SET salary = salary + 500
WHERE id = 2;
```

**When a record is updated in CUSTOMERS table, above create trigger display_salary_changes will be fired and it will display the following result:**

Old salary: 1500

New salary: 2000

Salary difference: 500

**LOCKS**

- *Locks* are mechanisms that prevent destructive interaction between transactions accessing the same resource.

General Object Type Affected By Locks:

- User objects, such as tables and rows (structures and data)
- System objects not visible to users, such as shared data structures in the memory and data dictionary rows

- All locks acquired by statements within a transaction are held for the duration of the transaction.
- Oracle releases all locks acquired by the statements within a transaction when an explict or implied commit or roll back is executed. Oracle also releases locks acquired after a savepoint when rolling back to the savepoint.

* Note: Only transactions not waiting for the previously locked resources can acquire locks on now available resources. Waiting transactions continue to wait until after the original transaction commits or completely rolls back.

**Oracle Lock Modes**

- **Exclusive Lock Mode**
- **Share Lock Mode**

**Exclusive Lock Mode**

Prevents the associates resource from being shared. This lock mode is obtained to modify data. The first transaction to lock a resource exclusively is the only transaction that can alter the resource until the exclusive lock is released.

**Share Lock Mode**

Allows the associated resource to be shared, depending on the operations involved . Multiple users reading data can share the data, holding share locks to prevent concurrent access by a writer (who needs an exclusive lock). Several transactions can acquire share locks on the same resource.

**Define lock? Explain shared and exclusive locks.**

***(Definition - 1 Mark, Explanation of each lock -1 ½ Marks)***

**Ans: Lock** can be defined as the mechanism that avoids the destructive interaction between two concurrent transactions or sessions that are attempting to access the same database object.

**Types of Locks-shared &Exclusive:**

- Exclusive locks avoid the resource from being shared. Exclusive lock mode is used to modify the data. Using exclusive lock single transaction can perform both read and write operation at a time.
- Shared lock allows the resource to be shared. This sharing of resource depends on the operations that are involved. Many users at a time can read the data. Share locks are used to avoid concurrent access to the writers.

**Write a PL/SQL program which handles divide by zero exception.**

***(Correct Program - 4 Marks)***

**Ans:**

```
DECLARE
A number:=20;
B number:=0;
C number;
BEGIN
dbms_output.put_line(„First Num : “||A);
dbms_output.put_line(„Second Num : “||B);
C:= A / B; --Raise Exception
dbms_output.put_line(„ Result “ || C); -- Result will not be displayed
EXCEPTION
WHEN ZERO_DIVIDE THEN
dbms_output.put_line(„ Trying to Divide by zero :: Error “);
END;
/
```

**Explain Lock Compatibility Table. What is Two-phase locking protocol?**

***(Lock compatibility table – 2 Marks, two phase locking protocol – 2Marks)***

**Ans:**

The two commonly used locks are 1) shared and 2) Exclusive

1) Shared lock : It can lock the transaction only for reading.

2) Exclusive Lock : It can lock the transaction for reading as well as writing.

The compatibility table for these two locks can be given as:

| | Shared | Exclusive |
|---|---|---|
| Shared | Compatible | Not compatible |
| Exclusive | Not Compatible | Not Compatible |

Two phase locking protocol:

This protocol is for ensuring serializability.

It requires the locking and unlocking of the transaction to be done in two phases.

1) **Growing phase:** In this phase, a transaction may obtain locks, but may not release locks.

2) **Shrinking phase:** In this phase, a transaction may release locks, but may not acquire any new locks.

**Explain the difference between DROP and TRUNCATE with example**

| DROP | TRUNCATE |
|---|---|
| 1 .It is a DDL Statement which deals with the structure of the table along with data. | 1.It is DDL statement which deals only with the data from the table. |
| 2. It deletes entire table at once from the disk. | 2.It deletes all records from table at once. |
| 3.Column Structure of table does not remain on the disk. | 3.Empty column structure of the table remains on the disk. |
| 4.Syntax :<br><br>Drop <tablename>; | 4.Syntax :<br><br>Truncate table <tablename>; |
| 5.Example :<br><br>Drop employee; | 5.Example :<br><br>Truncate table employee; |

**Write PL/SQL procedure to calculate factorial of a give number.**

*(Correct program – 4Marks)*

**[Note: Any other relevant logic shall be considered]**

**Ans:**

```
SQL> create or replace procedure Factorial(n in number)
is
v number :=1;
begin
for i in 1..n
loop
v :=v * i;
end loop;
dbms_output.put_line(v);
end;
Procedure created
SQL> Begin
Factorial(5);
End;
```

**Differentiate between function and procedure.**

| | function | procedure |
|---|---|---|
| 1. | Syntax:<br>CREATE [OR REPLACE]<br>PROCEDURE proc_name [list of parameters]<br>IS <Local Declaration section><br>BEGIN<br>Execution section<br>EXCEPTION<br>Exception section<br>END; | Syntax:<br>CREATE OR REPLACE FUNCTION function_name (argument IN data type ,…)<br>RETURN datatype {IS,AS}<br>Variable declaration;<br>Constant declaration;<br>BEGIN<br>PL/SQL statements;<br>EXCEPTION<br>Exception PL/SQL block<br>END; |
| 2. | Return type is must | Return type is not must |
| 3. | It can return any scalar or table | It can directly return only integers |
| 4. | Will compiled at run time | Stored in database in compiled format. |
| 5. | It won't support out parameters | It can also return more than one values (of any data type) indirectly with the help of out parameters |

**EXTRA**

**HOW TO DISPLAY MESSAGES ON SCREEN ---**

**DBMS_OUTPUT** : is a package that includes a number of procedure and functions that accumulate information in a buffer so that it can be retrieved later. These functions can also be used to display messages to the user.

**PUT_LINE** : Put a piece of information in the package buffer followed by an end-of-line marker. It can also be used to display message to the user. Put_line expects a single parameter of character data type. If used to display a message, it is the message 'string'.

EG: dbms_output.put_line(x);

**REMEMBER**: To display messages to the user the SERVEROUTPUT should be set to ON. SERVEROUTPUT is a sql*plus environment parameter that displays the information passed as a parameter to the PUT_LINE function.

EG: SET SERVEROUTPUT ON

A bit about comments. A comment can have 2 forms i.e.

-- The comment line begins with a double hyphen (--). The entire line will be treated as a comment.

-- The C style comment such as /* i am a comment */

**CONDITIONAL CONTROL AND ITERATIVE CONTROL AND SEQUENTIAL CONTROL**

IF and else.....

IF --Condition THEN

--Action

ELSEIF --Condition THEN

--Action

ELSE

--Action

END IF;

**SIMPLE LOOP**

loop

-- Sequence of statements;

end loop;

the loop ends when u use EXIT WHEN statement --condition

WHILE LOOP

While --condition

loop

--sequence of statements

end loop;

**FOR LOOP**

FOR i in 1..10

loop

--sequence of statements

end loop;

GOTO (sequential control)

GOTO X;

<< X >>

EXAMPLES

**--ADDITION**

declare

a number;

b number;

c number;

begin

a:=&a;

b:=&b;

c:=a+b;

dbms_output.put_line('Sum of ' || a || ' and ' || b || ' is ' || c);

Here & is used to take user input at runtime.....

**--SUM OF 100 NUMBERS**

```
Declare
a number;
s1 number default 0;
Begin
a:=1;
loop
s1:=s1+a;
exit when (a=100);
a:=a+1;
end loop;
dbms_output.put_line('Sum between 1 to 100 is '||s1);
End;
```

**--SUM OF odd NUMBERS USING USER INPUT...for loop**

```
declare
n number;
sum1 number default 0;
endvalue  number;
begin
endvalue:=&endvalue;
n:=1;
for n in 1.. endvalue
loop
```

```
if mod(n,2)=1
then
sum1:=sum1+n;
end if
end loop;
dbms_output.put_line('sum = ' || sum1);
end;
```

**--SUM OF 100 ODD NUMBER .. WHILE LOOP**

```
declare
n number;
endvalue number;
sum1 number default 0;
begin
endvalue:=&endvalue;
n:=1;
while (n < endvalue)
loop sum1:=sum1+n;
n:=n+2;
end loop;
dbms_output.put_line('Sum of odd numbers between 1 and ' || endvalue || ' is ' ||
sum1);
end;
```

**--CALCULATION OF NET SALARY**

```
declare
ename varchar2(15);
basic number;
da number;
hra number;
pf number;
netsalary number;
begin
ename:=&ename;
basic:=&basic;

da:=basic * (41/100);
hra:=basic * (15/100);

if (basic < 3000)
then
pf:=basic * (5/100);
elsif (basic >= 3000 and basic <= 5000)
then
pf:=basic * (7/100);
elsif (basic >= 5000 and basic <= 8000)
then
```

```
pf:=basic * (8/100);
else
pf:=basic * (10/100);
end if;
netsalary:=basic + da + hra -pf;
dbms_output.put_line('Employee name : ' || ename);
dbms_output.put_line('Providend Fund : ' || pf);
dbms_output.put_line('Net salary : ' || netsalary);
end;
```

**--MAXIMUM OF 3 NUMBERS**

```
Declare
a number;
b number;
c number;
d number;
Begin
dbms_output.put_line('Enter a:');
a:=&a;
dbms_output.put_line('Enter b:');
b:=&b;
dbms_output.put_line('Enter c:');
c:=&b;
if (a>b) and (a>c) then
```

```
dbms_output.putline('A is Maximum');
elsif (b>a) and (b>c) then
dbms_output.putline('B is Maximum');
else
dbms_output.putline('C is Maximum');
end if;
End;
```

**--QUERY EXAMPLE--IS SMITH EARNING ENOUGH**

```
declare
s1 emp.sal %type;
begin
select sal into s1 from emp
where ename = 'SMITH';
if(no_data_found)
then
raise_application_error
(20001,'smith is not present');
end if;
if(s1 > 10000)
then
raise_application_error
(20002,'smith is earning enough');
end if;
```

```
update emp set sal=sal + 500
where ename='SMITH';
end;
```

**--PRIME NO OR NOT**

```
DECLARE
no NUMBER (3) := &no;
a NUMBER (4);
b NUMBER (2);
BEGIN
FOR i IN 2..no - 1
LOOP
a := no MOD i;
IF a = 0
THEN
GOTO out;
END IF;
END LOOP;
<>
IF a = 1
THEN
DBMS_OUTPUT.PUT_LINE (no || ' is a prime number');
ELSE
DBMS_OUTPUT.PUT_LINE (no || ' is not a prime number');
```

```
END IF;
END;
```

**--SIMPLE EXAMPLE OF LOOP STATEMENT I.E. EXIT WHEN**

```
Declare
a number:= 100;
begin
loop
a := a+25;
exit when a=250;
end loop;
dbms_output.put_line (to_Char(a));
end;
```

**--EXAMPLE OF WHILE LOOP**

```
Declare
i  number:=0;
j number:= 0;
begin
while i <=100 loop
j := j+1;
i := i +2;
end loop;
dbms_output.put_line(to_char(i));
end;
```

**--EXAMPLE OF FOR LOOP**

```
Declare
begin
for i in 1..10
loop
dbms_output.put_line(to_char(i));
end loop;
end;
```

**--SEQUENTIAL CONTROL GOTO**

```
declare
--takes the default datatype of the column of the table price
cost price.minprice%type;
begin
select stdprice into cost from price where prodial in (Select prodid from product where prodese = "shampoo");
if cost > 7000 then
goto Upd;
end if;
<< Upd >>
Update price set minprice = 6999 where prodid=111;end;
```

**--CALCULATE THE AREA OF A CIRCLE FOR A VALUE OF RADIUS VARYING FROM 3 TO 7. STORE THE RADIUS AND THE CORRESPONDING VALUES OF CALCULATED AREA IN A TABLE AREAS.**

```
Declare
pi constant number(4,2) := 3.14;
radius number(5);
area number(14,2);

Begin
radius := 3;
While radius <=7
Loop
area := pi* power(radius,2);
Insert into areas values (radius, area);
radius:= radius+1;
end loop;
end;
```

--REVERSING A NUMBER 5639 TO 9365

```
Declare
given_number varchar(5) := '5639';
str_length number(2);
```

```
inverted_number varchar(5);


Begin
str_length := length(given_number);
For cntr in reverse 1..str_length
loop
inverted_number := inverted_number || substr(given_number, cntr, 1);
end loop;
dbms_output.put_line('The Given no is ' || given_number);
dbms_output.put_line('The inverted number is ' || inverted_number);
end;
```

**EXCEPTION HANDLING IN PLSQL**

Errors in pl/sql block can be handled...error handling refers to the way we handle the errors in pl/sql block so that no crashing stuff of code takes place...This is exactly the same as we do in C++ or java..right!!

There are two type:

===> predefined exceptions

===> user defined exceptions

The above 2 terms are self explanatory

**Predefined exceptions:**

No-data-found == when no rows are returned

Cursor-already-open == when a cursor is opened in advance

Dup-val-On-index == for duplicate entry of index..

Storage-error == if memory is damaged

Program-error == internal problem in pl/sql

Zero-divide == divide by zero

invalid-cursor == if a cursor is not open and u r trying to close it

Login-denied == invalid user name or password

Invalid-number == if u r inserting a string datatype for a number datatype which is already declared

Too-many-rows == if more rows r returned by select statement

**SYNTAX**

begin

sequence of statements;

exception

when --exception name then

sequence of statements; end;

**EXAMPLES**

--When there is no data returned by row

declare

price item.actualprice%type;

begin

```
Select actual price into price from item where qty=888;
when no-data-found then
dbms_output.put_line('item missing');
end;
```

**--EXAMPLE OF USER DEFINED EXCEPTION**

```
DECLARE
e_rec emp%ROWTYPE;
e1 EXCEPTION;
sal1 emp.sal%TYPE;
BEGIN
SELECT sal INTO sal1 FROM emp WHERE deptno = 30 AND ename = 'John';
IF sal1 < 5000 THEN
RAISE e1;
sal1 := 8500;
UPDATE emp SET sal = sal1 WHERE deptno = 30 AND ename = 'John';
END IF;
EXCEPTION
WHEN no_data_found THEN
RAISE_APPLICATION_ERROR (-20001, 'John is not there.');
WHEN e1 THEN
RAISE_APPLICATION_ERROR (-20002, 'Less Salary.');
END;
```

**--EXAMPLE OF RAISE-APPLICATION-ERROR... THIS IS YOUR OWN ERROR STATEMENT...YOU RAISE YOUR OWN ERROR**

```
Declare
s1 emp.sal %type;
begin
select sal into s1 from emp where ename='SOMDUTT';
if(no-data-found) then
raise_application_error(20001, 'somdutt is not there');
end if;
if(s1 > 10000) then
raise_application_error(20002, 'somdutt is earing a lot');
end if;
update emp set sal=sal+500 where ename='SOMDUTT';
end;
```

**--INTERESTING EG OF USER DEFINED EXCEPTIONS**

```
Declare
zero-price exception;
price number(8);
begin
select actualprice into price from item where ordid =400;
if price=0 or price is null then
raise zero-price;
end if;
```

```
exception
when zero-price then
dbms_output.put_line('raised xero-price exception');
end;
```

# Relational Database Management System important questions for Exam

- **CHAPTER 1**

**2 Marks Question**

1. Definition of DBMS
2. Definition of RDBMS
3. State Application of DBMS
4. State any 2 difference in RDBMS & DBMS
5. Definition of instance & schema
6. State 4 Codd's Rule
7. What is Data Independence?
8. What is Physical Data Independence?
9. What is Logical Data Independence?
10. State Component of DBMS
11. State different User of DMBS
12. State DBA functions.
13. What is data mining?
14. What is data warehouse?
15. Various names of DBMS & RDBMS software

**4 Marks Questions**

1. State & explain disadvantages of file processing system over DBMS.
2. Differentiate between RDBMS & DBMS.
3. State Data Abstraction. Explain different level of data abstraction
4. Explain Data Independence
5. Draw neat diagram of overall structure of database
6. State component of DBMS. Explain them.
7. List Database Users, Explain types of database user
8. List function of DBA. Explain DBA Function in details.
9. What is client-server Architecture? What is 2 Tier & 3 Tier architecture?
10. List all 12 Codd's Rule.
11. Explain any 4 rules in 12 Codd's Rule.

# Relational Database Management System important questions for External Oral

## CHAPTER 1

1. Definition of DBMS
2. Definition of RDBMS
3. State Application of DBMS
4. State difference in RDBMS & DBMS
5. Definition of instance & schema
6. State Codd's Rule
7. What is Data Independence?
8. What is Physical Data Independence?
9. What is Logical Data Independence?
10. State Component of DBMS
11. State different User of DMBS
12. State DBA functions.
13. What is data mining?
14. What is data warehouse?
15. Various names of DBMS & RDBMS software
16. State disadvantages of file processing system over DBMS.
17. State Data Abstraction. Explain different level of data abstraction
18. List Database Users.
19. What is client-server Architecture? What is 2 Tier & 3 Tier architecture?

# Relational Database Management System important questions for Exam

- **CHAPTER 2**

**2 Marks Question**

1. What is Data Model?
2. Types of Data Model
3. Define Primary key, Candidate key
4. Define Primary key, Foreign key
5. What is Relational Algebra?
6. What is Discriminator?
7. What is Weak & Strong Entity set?
8. Define Normalization & define its Goals.
9. What is Need of Normalization?
10. Explain Integrity Constraints.
11. What is the type of Integrity Constraints?
12. What is select operation in relational algebra? Give example.
13. List out 4 symbol used in E-R diagram with their names.
14. What is Data security?
15. Define Rules for 1NF.
16. Define Rules for 2NF.
17. Define Rules for BCNF.
18. Define Rules for 3NF.
19. Define Rules for 4NF.

**4 Marks Questions**

1. What is data Model? Explain Network & Hierarchical model.
2. Differentiate between Network & Hierarchical model.
3. Differentiate between Strong & Weak entity set.
4. What is E-R? Draw with name of symbol E-R.
5. What is attribute ? Explain types of attribute.
6. Draw an E-R Diagram of Customer account relationship.
7. Draw an E-R Diagram of Library management.
8. Draw an E-R Diagram of Hospital management.

9. Draw an E-R Diagram of School management.
10. Draw an E-R Diagram of Department management.
11. Draw an E-R Diagram of Bus management.
12. Explain Enhanced E-R model.
13. Explain specialization & generalization.
14. Explain 'ISA' relation.
15. Explain 1NF with example.
16. Explain 2NF with example.
17. Explain BCNF with example.
18. Explain 3NF with example.
19. Explain 4NF (Multivalued Dependency) with example.
20. Explain 5NF (Join Dependency) with example.
21. What are data security & its requirement?
22. Write a short note on Integrity Constraints.
23. Explain in brief network model with example.
24. Explain in brief hierarchical model with example.
25. Consider following schema algebra

    Student (Roll_no, Name, DOB, percentage, Course)

    Department (Dept_no, Dept_Name, Head)

    Write Relational Algebra Expression for:

    1. Find Student name & percentage from Computer Dept.
    2. Get the student name who has percentages greater than 70.

# Relational Database Management System important questions for External Oral

## CHAPTER 2

1. What is Data Model?
2. Types of Data Model
3. Define Primary key, Candidate key
4. Define Primary key, Foreign key
5. What is Relational Algebra?
6. What is Discriminator?
7. What is Weak & Strong Entity set?
8. Define Normalization & define its Goals.
9. What is Need of Normalization?
10. Explain Integrity Constraints.
11. What is the type of Integrity Constraints?
12. What is select operation in relational algebra? Give example.
13. List out 4 symbol used in E-R diagram with their names.
14. What is Data security?
15. Define Rules for 1NF.
16. Define Rules for 2NF.
17. Define Rules for BCNF.
18. Define Rules for 3NF.
19. Define Rules for 4NF.
20. Explain 'ISA' relation.
21. What is attribute ? Explain types of attribute.

# Relational Database Management System important questions for Exam

## CHAPTER 3

### 2 Marks Question

1. Define DDL
2. Define DML
3. Explain Data types of SQL
4. Write syntax for Create Table.
5. Write syntax for Alter Table.
6. Write syntax for Truncate Table.
7. Write syntax for DESC.
8. Write syntax for Drop Table.
9. Write syntax for Rename Table.
10. Write syntax for Update Table.
11. Write syntax for Create.
12. Write syntax for Update.
13. Write syntax for Delete.
14. Write syntax for Insert.
15. State Arithmetic Operators.
16. State Comparison Operators.
17. State Logical Operators.
18. State Set Operators.
19. State Range Searching operators.
20. State Pattern matching operators.
21. State String Function.
22. State Arithmetic function.
23. State Date and time function.
24. State Aggregate function.
25. State Date formats using To_char(), To_date function.
26. Write syntax for Group by
27. Write syntax for Having Clause
28. Write syntax for Order by clause
29. Write syntax for GRANT
30. Write syntax for REVOKE

**4 Marks Question.**

1. Explain group by having & order by clause in SQL.
2. Give the syntax of create & RENAME command with example.
3. Consider following schema.
   EMPLOYEE-DETAILS (empname, empId, DOB, salary, job)
   Create a table EMPLOYEE-DETAILS
4. Consider the following schema.
   STUDENT (Name, Mark, Age, Place, Phone, Birthdate)
   Write SQL queries for following:
   i) To list name of student who do not have phone number.
   ii) To list student from Nashik & Pune.
   iii) To change mark of Monika to 88 instead of 80.
   iv) To list the student from Amit's age group.
5. Explain DROP & DELETE commands with syntax. State the difference between them.
6. Explain the set operators of SQL.
7. List the SQL operators & explain range searching operator - "BETWEEN" & pattern matching operator - "LIKE".
8. Explain the database security with its requirements.
9. Explain any four aggregate functions with example
10. Explain any four string functions.
11. What is the use of GRANT and REVOKE.
12. What are the four ways to insert a record in a table ?
13. Given – Employee(EMP_ID, FIRST_NAME,LAST_NAME, SALARY, JOINING_DATE,DEPARTMENT )
    Write SQL queries for –
    i) Get FIRST_NAME, LAST_NAME from employee table.
    ii) Get unique DEPARTMENT from employee table.
    iii) Get FIRST_NAME from employee table using alias name "Employee Name"
    iv) Get FIRST_NAME from employee table after removing white spaces from left side.
14. Explain the difference between DROP and TRUNCATE with example.
15. What is schedule? Explain conflict Serializability.
16. Explain types of JOINs
17. How to use COMMIT, SAVE POINT, ROLLBACK commands.
18. Explain ACID properties
19. Explain GROUP BY, ORDER BY clause of SQL with example.

20. Consider following database and solve queries
    emp (empno, ename, ph, sal, dept_no, comm)
    (i) Change employee name 'Rahul' to 'Ramesh'.
    (ii) Give increment of 20% in salary to all employees
21. Consider following schema:
22. Employee (emp_no, emp_name, dept, designation, salary, Dept_location) Solve following queries:
    (i) List all Managers in Mumbai location.
    (ii) Set salary of all 'project leaders' to 70000/-.
    (iii) List employees with having alphabet 'A' as second letter in their name.
    (iv) Display details of those employees who work in Mumbai or Chennai.
23. Explain Alter command. Give syntax of add and modify option.
24. Consider following schema:
    depositor (cust_name, acc_no)
    Borrower (cust_name, loan_no)
    Solve following queries:
    (i) Find customer name having savings account as well as loan account.
    (ii) Find customer names having loan account but not the saving account.
25. Explain:
    (i) order by clause (ii) grant command (iii) commit command (iv) savepoint command.
26. Explain word comparison operators:
    (i) IN and NOT IN (ii) BETWEEN and NOT BETWEEN
27. Give SQL DDL defination of the databse scheme given below and solve the following queries.

    Emp (Emp no., Ename, Job, mgr, Joindate, Salary, Comm., Dept no., Address)

    (i) Change the joining date of employee having emp no. - 1000 to 12/11/2009.

    (ii) Write a query to display the employee details whose emp no = 500.

    (iii) Write a query to display the name and salary of employee who earn more than Rs. 5000 and are in department 10 or 30.

    (iv) Find the list of employee whose salary is between 5000 to 20,000.

28. Draw & Explain State of Transaction.

# Relational Database Management System important questions for Oral Exam

1. Define DDL
2. Define DML
3. Explain Data types of SQL
4. Write syntax for Create Table.
5. Write syntax for Alter Table.
6. Write syntax for Truncate Table.
7. Write syntax for DESC.
8. Write syntax for Drop Table.
9. Write syntax for Rename Table.
10. Write syntax for Update Table.
11. Write syntax for Create.
12. Write syntax for Update.
13. Write syntax for Delete.
14. Write syntax for Insert.
15. State Arithmetic Operators.
16. State Comparison Operators.
17. State Logical Operators.
18. State Set Operators.
19. State Range Searching operators.
20. State Pattern matching operators.
21. State String Function.
22. State Arithmetic function.
23. State Date and time function.
24. State Aggregate function.
25. State Date formats using To_char(), To_date function.
26. Write syntax for Group by
27. Write syntax for Having Clause
28. Write syntax for Order by clause
29. Write syntax for GRANT
30. Write syntax for REVOKE