

DJA3D: RDBMS AND ORACLE

UNIT I

DATABASE CONCEPTS:

Database - purpose of Database systems - View of Data - Relational Databases - Database Architecture - the relational Database model - Integrity rules - Relational Algebra.

UNIT II

DATABASE DESIGN:

Data modeling – functional dependency – Database design – Normal forms - Personal databases – Client/ server databases – The SQL*PLUS Environment – SQL*PLUS commands.

UNIT III

Oracle Tables: Naming rules and conventions – Data types – Constraints – Creating Oracle table – Displaying table information – Altering an existing table- Dropping a table – Renaming a table – Truncating a table.

UNIT IV

Working with tables: DML statements – Arithmetic operations – Where clause sorting – DEFINE command – Built in functions – Grouping data
Multiple tables: Joins – Set operators – sub query.

UNIT V

PL/SQL: Fundamentals – Block structure – Comments – Data types – Variable declaration – Anchored declaration – Assignment operation – Bind variables – Substitution Variables – Arithmetic operators- control structures - PL/SQL Cursors & Exceptions.

TEXT BOOK:

1. Database Systems Using Oracle – Second edition – Nilesh Shah – PHI 2007

Reference Books:

1. Database system concepts –Henry F.Korth.
2. Oracle 9i Complete reference – Loney Koch – Tata Mc Graw Hill 2005.

LESSON-1

DATABASE CONCEPTS

DATABASE AN INTRODUCTION

A database is an electronic store of data. It is a repository that stores information about different "things" and also contains relationships among those different "things." Let us examine some of the basic terms used to describe the structure of a database:

- A person, place, event, or item is called an **entity**.
- The facts describing an entity are known as **data**. For example, if you were a registrar in a college, you would like to have all the information about the students. Each student is an entity in such a scenario.
- Each entity can be described by its characteristics, which are known as attributes. For example, some of the likely attributes for a college student are student identification number, last name, first name, phone number, Social Security number, gender, birthdate, and so on.
- All the related entities are collected together to form an entity set. An entity set is given a singular name. For example, the STUDENT entity set contains data about students only. All related entities in the STUDENT entity set are students. Similarly, a company keeps track of all its employees in an entity set called EMPLOYEE. The EMPLOYEE entity set does not contain information about the company's customers, because it wouldn't make any sense.
- A database is a collection of entity sets. For example, a college's database may include information about entities such as student, faculty, course, term, course section, building, registration information, and so on.

- The entities in a database are likely to interact with other entities. The interactions between the entity sets are called relationships. The interactions are described using active verbs. For example, a student takes a course section (CRSSECTION), so the relationship between STUDENT and CRSSECTION is takes. A faculty member teaches in a building, so the relationship between FACULTY and BUILDING is teaches.

RELATIONSHIPS

The database design requires you to create entity sets, each describing a set of related entities. The design also requires you to establish all the relationships between the entity sets within the database. The different database management software packages handle the creation and use of relationships in different manners. Depending on the type of interaction, the relationships are classified into three categories:

1.One-to-one relationship: A one-to-one relationship is written as **1:1** in short form. It exists between two entity sets, X and Y, if an entity in entity set X has only one matching entity in entity set Y, and vice versa. For example, a department in a college has one chairperson, and a chairperson chairs one department in a college. An employee manages one department in a company, and only one employee manages a department.

2.One-to-many relationship: A one-to-many relationship is written as **1:M**. It exists between two entity sets, X and Y, if an entity in entity set X has many matching entities in entity set Y but an entity in entity set Y has only one matching entity in entity set X. In such a situation, a 1:M relationship exists between entity sets X and Y. For example, a faculty teaches for one division in a college, but a division has many faculty members. The relationship between DIVISION and FACULTY is 1:M. An employee works in a department, but a department has many employees. The relationship between DEPARTMENT and EMPLOYEE is 1:M.

3. Many-to-many relationship: A many-to-many relationship is written as **M:N** or **M:M**. It exists between two entity sets, X and Y, if an entity in entity set X has many matching entities in entity set Y and an entity in entity set Y has many matching entities in entity set X. For example, a student takes many courses, and many students take a course. An employee works on many projects, and a project has many employees.

DATABASE MANAGEMENT SYSTEM (DBMS)

The database system consists of the following components:

- A database management System (DBMS) software package such as Microsoft Access, Visual Fox Pro, Microsoft SQL-Server, or Oracle.
- A user-developed and implemented database or databases that include tables, a data dictionary, and other database objects.
- Custom applications such as data-entry forms, reports, queries, blocks, and programs.
- Computer hardware personal computers, minicomputers, and mainframes in a network environment.
- Software—an operating system and a network operating system.
- Personnel a database administrator, a database designer/analyst, a programmer, and end users.

Data are the raw materials. Information is processed, manipulated, collected, or organized data. The information is produced when a user uses the applications to

<i>User</i>

<i>Applications</i>	DBMS	<i>Database</i>
<i>OS Software</i>		
<i>Hardware</i>		

Database system

transform data managed by the DBMS. The database system is utilized as a decision-making system and is also referred to as an information system (IS).

A DBMS based on the relational model is also known as a Relational Database Management System (RDBMS). An RDBMS not only manages data but is also responsible for other important functions:

- It manages the data and relationships stored in the database. It creates a Data Dictionary as a user creates a database. The Data Dictionary is a system structure that stores Metadata (data about data). The Metadata include table names, attribute names, data types, physical space, relationships, and so on.
- It manages all day-to-day transactions.
- It performs bookkeeping duties, so the user has data independence at the application level. The applications do not have information about data characteristics.
- It transforms logical data requests to match physical data structures. When a user requests data, the RDBMS searches through the Data Dictionary, filters out unnecessary data, and displays the results in a readable and understandable form.

- It allows users to specify validation rules. For example, if only M and F are possible values for the attribute gender, users can set validation rules to keep incorrect values from being accepted.
- It secures access through passwords, encryption, and restricted user rights.
- It provides backup and recovery procedures for physical security of data.
- It allows users to share data with data-locking capabilities.
- It provides import and export utilities to use data created in other database or spreadsheet software or to use data in other software.
- It enables users to join tables to view information stored in different tables within the database. The user is able to design a database with less redundancy, which means fewer data-entry errors, fewer data corrections, better data integrity, and a more efficient database.

RELATIONAL DATABASE MODEL

The need for data is always present. In the computer age, the need to represent data in an easy-to-understand, logical form has led to many different models, such as the relational model, the hierarchical model, the network model, and the object model. Because of its simplicity in design and ease in retrieval of data, the relational database model has been very popular, especially in the personal computer environment.

E. F. Codd developed the relational database model in 1970. The model is based on mathematical set theory, and it uses a **relation** as the building block of the database. The relation is represented by a two-dimensional, flat structure known as a table. The user does not have to know the mathematical details or the physical aspects of the data, but the user views the data in a logical, two-dimensional structure. The

database system that manages a relational database environment is known as a Relational Database Management System (RDBMS). Some of the popular relational database systems are Oracle9i by Oracle Corporation, Microsoft Access 2000, and Microsoft Visual Fox Pro 6.0.

A table is a matrix of rows and columns in which each row represents an entity and each column represents an attribute. In other words, a table represents an entity set as per database theory, and it represents a relation as per relational database theory. In daily practice, the terms table, relation, and entity set are used interchangeably.

INTEGRITY RULES

In any database managed by an RDBMS, it is very important that the data in the underlying tables be consistent. If consistency is compromised, the data are not usable. This need led the pioneers of database field to formulate two integrity rules:

1. Entity integrity: No column in a primary key may be null. The primary key provides the means of uniquely identifying a row or an entity. A null value means a value that is not known, not entered, not defined, or not applicable. A zero or a space is not considered to be a null value. If the primary key value is a null value in a row, we do not have enough information about the row to uniquely identify it. The RDBMS software strictly follows the entity integrity rule and does not allow users to enter a row without a unique value in the primary key column.

2. Referential integrity: A foreign key value may be a null value, or it must exist as a value of a primary key in the referenced table.

Referential integrity is not fully supported by all commercially available systems, but Oracle supports it religiously! Oracle does not allow you to declare a foreign key if it does not exist as a primary key

in another table. It allows you to leave the foreign key column value as a null. If a user enters a value in the foreign key column, Oracle cross-references the referenced primary key column in the other table to confirm the existence of such a value.

It is not a good practice to use null values in any non-primary key columns, because this results in extra overhead on the system's part in search operations. The programmers or query users have to add extra measures to include or exclude rows with null values. In certain cases, it is not possible to avoid null values. For example, an employee does not have a middle initial, an employee is hired but does not have an assigned department, or a student's major is undefined. In Oracle, a default value can be assigned to a column, and a user does not have to enter a value for that column.

THEORETICAL RELATIONAL LANGUAGES

E. F. Codd suggested two theoretical relational languages to use with the relational model:

- 1. Relational algebra**, a procedural language.
- 2. Relational calculus**, a nonprocedural language.

Third-generation high-level compiler languages can be used to manipulate data in a table, but they can only work with one row at a time. In contrast, the relational languages can work on the entire table or on a group of rows. The multiple-row

manipulation does not even need a looping structure! The relational languages provide more power with a very little coding. Codd proposed these languages to embed them in other host languages for more processing capability and more sophisticated application development. In the database systems available today, nonprocedural Structured Query Language (SQL) is used as a data-manipulation

sublanguage. The theoretical languages have provided the basis for SQL.

Relational Algebra

Relational algebra is a procedural language, because the user accomplishes desired results by using a set of operations in a sequence. It uses set operations on tables to produce new resulting tables. These resulting tables are then used for subsequent sequential operations. In Oracle, all operation names are not actually used as programming terms, and most of these operations do not create a new resulting table, as shown in the following examples using relational algebra.

The nine operations used by relational algebra are:

1. Union.
2. Intersection.
3. Difference.
4. Projection.
5. Selection.
6. Product.
7. Assignment.
8. Join.
9. Division.

Union. The union of two tables results in retrieval of all rows that are in one or both tables. The duplicate rows are eliminated from the resulting table. The resulting table does not contain two rows with identical data values. There is a basic requirement to perform a union operation on two tables:

- Both tables must have the same degree.
- The domains of the corresponding columns in two tables must be same.

Such tables are said to be union compatible. In mathematical set theory, a union can be performed on any two sets, but in relational algebra, a union can be performed only on union-compatible tables.

Intersection. The intersection of two tables produces a table with rows that are in both tables. The two tables must be union compatible to perform an intersection on them.

Difference. The difference of two tables produces a table with rows that are present in the first table but not in the second table. The difference can be performed on union-compatible tables only.

Projection. The projection operation allows us to create a table based on desirable columns from all existing columns in a table. The undesired columns are ignored. The projection operation returns the "vertical slices" of a table. The projection is indicated by including the table name and a list of desired columns:

Selection. The selection operation selects rows from a table based on a condition or conditions. The conditional operators ($=$, $<$, $>$, \geq , \leq , $<=$) and the logical operators (AND, OR, NOT) are used along with columns and values to create conditions. The selection operation returns "horizontal slices" from a table.

Product. A product of two tables is a combination everything in both tables. It is also known as a Cartesian product. It can cause huge results with big tables. If the first table has x rows and the second table has y rows, the resulting product has $x \cdot y$ rows. If the first table has m columns and the second table has n columns, the resulting product has $m + n$ columns.

Assignment. This operation creates a new table from existing tables. We have been doing it throughout all the other operations. Assignment ($=$) gives us an ability to name new tables that are based on other tables. Note that assignment is not an Oracle term.

For example,

TABLE_A = PROJ2002 U PROJ2003

TABLE_C = PROJ2002 - PROJ2003

Join. The join is one of the most important operations because of its ability to get related data from a number of tables. The join is based on common set of values, which does not have to have the same name in both tables but does have to have the same domain in both tables. When a join is based on equality of value, it is known as a **natural join**. In Oracle, you will learn about the natural join, or **equijoin**, and also about other types of joins, such as **outer join**, **non equijoin**, and **self-join**, that are based on the operators other than the equality operator.

Division. The division operation is the most difficult operation to comprehend. It is not as simple as division in mathematics. In relational algebra, it identifies rows in one table that have a certain relationship to all rows in another table. Let us consider the following two tables.

LESSON -2
DATABASE DESIGN
DATA MODELING

A model is a simplified version of real-life, complex objects. Databases are complex, and data modeling is a tool to represent the various components and their relation-ships. The entity-relationship (E-R) model is a very popular modeling tool among many such tools available today. Many tools are available for data modeling with E-R. All tools have some variations in representation of components. The E-R model provides:

- An excellent communication tool.
- A simple graphical representation of data.

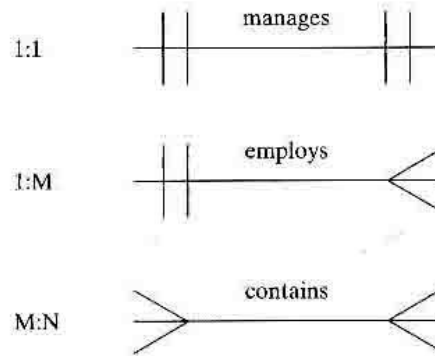
The E-R model uses E-R diagrams (ERD) for graphical representation of the database components. An entity (or an entity set) is represented by a rectangle. The name of the entity (set) is written within the rectangle. Some tools prefer to use uppercase letters only for entities. The name of an entity set is a singular noun. For example, EMPLOYEE, CUSTOMER, and DEPARTMENT are singular entity set names.

A line represents relationship between the two entities. The name of the relationship is an active verb in lowercase letters. For example, works; manages, and employs are active verbs. Passive verbs can be used, but active verbs are preferable

.



Entity representation in an E-R diagram.



Representation of relationship in an E-R diagram.

The types of relationships (1:1, 1:M, and M:N) between entities are called **connectivity or multiplicity**. The connectivity is shown with vertical or angled lines next to each entity. For example, an EMPLOYEE supervises a DEPARTMENT, and a DEPARTMENT has one EMPLOYEE supervisor. A DIVISION contains many FACULTY members, but a FACULTY works for one DIVISION. An INVOICE contains many ITEMS, and an ITEM can be in more than one INVOICE.

Let us put everything together and represent these scenarios with the E-R diagram that shows entities, relationships, and connectivity.

The relationship between two entities can be given using the lower and upper limits. This information is called the **cardinality**. The cardinality is written next to each entity in the form (n, m) , where n is the minimum number and m is the maximum number. For example, $(1,1)$ next to EMPLOYEE means that an employee can supervise a minimum of one and a maximum of one department. Similarly, $(1,1)$ next to DEPARTMENT says that one and only one employee supervises the department. The value $(1,N)$ means a minimum of one and a maximum equal to any number. Some modern tools do not show cardinality in an E-R diagram.

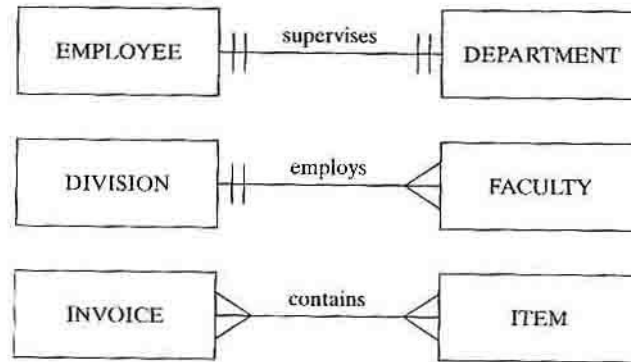
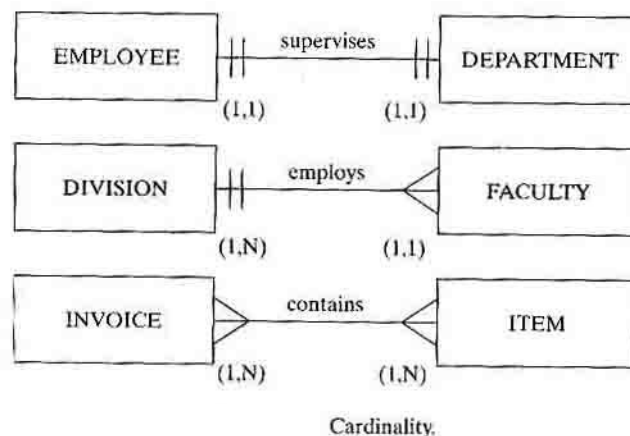


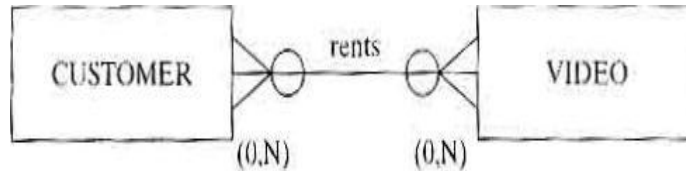
Figure 2-3 Entity, relationship, and connectivity.



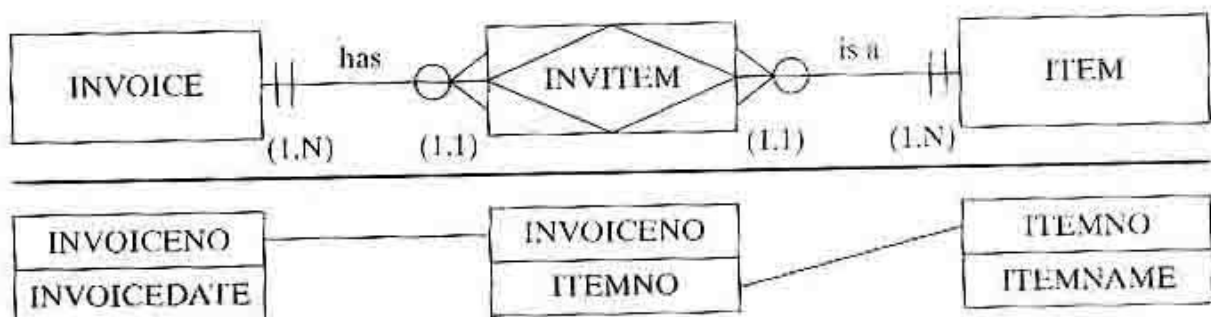
In reality, corporations set rules for the minimum and maximum values for cardinality. A corporation may decide that a department must have a minimum of 10 employees and a maximum of 25 employees, which results in cardinality of (10,25). A college decides that a computer-science course section must have at minimum 5 students to recover the cost incurred and at maximum 35 students, because the computer lab contains only 35 terminals. An employee can be part of zero or more than one department, and an item may not be in any invoice! These types of decisions are known as business rules.

the above E-R diagram with added cardinality. In real life, it is possible to have an entity that is not related to another entity at all times. The relationship becomes optional in such a case. In the example of a video rental store. a customer can rent video movies. In this case. there are times when the customer has not rented any movie, and there are times when the customer has rented one or more movies. Similarly, there can be a movie in the database that is or is not rented

at a particular time. These are called **optional relationships** and are shown with a small circle next to the optional entity. The optional relationship can occur in 1:1, 1:M, or M:N relationships, and it can occur on one or both sides of the relationship.



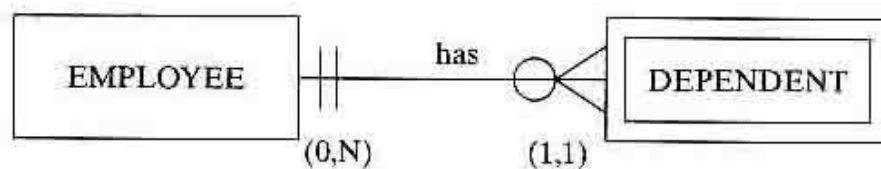
In relational databases, many-to-many (M:N) relationships are allowed, but they are not easy to implement. For example, an invoice has many items, and an item can be in many invoices. Refer to the INVOICE and ITEM relationship. At this point, you will be introduced to the relational schema, a graphical representation of tables, their column names, key components, and relations between the primary key in one table and the foreign key in another. You will also see the decomposition of an M:N relationship into two 1:M relationships. The decomposition from M:N to 1:M involves a third entity, known as a **composite entity** or an associative entity. The composite entity is created with the primary key from both tables with M:N relationships. The *new* entity has a composite key, which is a combination of primary keys from the original two entities. In the E-R diagram, a composite entity is drawn as a diamond within a rectangle. The composite



Composite entity and relational schema.

entity has a composite primary key with two columns, each of them being foreign keys referencing the other two entities in the database. For example, the foreign key INVOICENO in the INVITEM table references the INVOICENO column in the INVOICE table, and the foreign key ITEMNO in the INVITEM table references the ITEMNO column in the ITEM table.

In a database, there are entities that cannot exist by themselves. Such entities are known as weak entities. you will be introduced to two different sample databases. In the employee database of that chapter, there is an entity called EMPLOYEE with employees' demographic information and another entity called DEPENDENT with information about each employee's dependents. The DEPENDENT entity cannot exist by itself. There are no dependents for an employee who does not exist. In other words, you need the existence of an employee for his or her dependent to exist in the database. The weak entities are shown by double-lined rectangles



Weak Entity

Some of the other elements considered in the database design are:

- **Simple attributes**—attributes that cannot be subdivided; for example, last name, city, or gender.
- **Composite attributes**—attributes that can be subdivided, into atomic form; for example, a full name can be subdivided into the last name, first name, and middle initial.

- **Single-valued attributes**—attributes with a single value; for example, Employee ID, Social Security number, or date of birth.
- **Multivalued attributes**—attributes with multiple values; for example, degree codes or course registration. The multivalued attributes have to be given special consideration. They can be entered into one attribute with a value separator mark, or they can be entered in separate attributes with names like Course1, Course2, Course3, and so on. Alternatively, a separate, composite entity can be created.

DEPENDENCY

In Chapter 1, you learned that the primary key in a table identifies an entity. Every table in the database should have a primary key, which uniquely identifies an entity. For example, PartNo is a primary key in the PARTS table, and DeptNo is a primary key in the DEPARTMENT table. In Oracle, if you create a table and do not define its primary key. Oracle does not consider it to be an error. You should define a primary key for all tables for integrity of data. Each table has other columns that do not make up the primary key for the table. Such columns are called the nonkey columns. The nonkey columns are functionally dependent on the primary key column. For example, PartDesc and Cost in the PARTS table are dependent on the primary key PartNo, and DeptName is dependent on the primary key DeptNo in the DEPARTMENT table.

Now, let us take a scenario as shown below. The INVOICE table in does not have any single column that can uniquely identify an entity. The first choice would be InvNo. It is not a unique value in the table, however, because an invoice may contain more than one item and there may be more than one entry for an invoice. CustNo cannot be the primary key, because there can be many invoices for a customer and CustNo does not identify an invoice. ItemNo cannot be the primary key either, because an item may appear in more than one invoice

and ItemNo does not describe an invoice. The table has a composite primary key, which consists of InvNo and ItemNo. InvNo and ItemNo together make up unique values for each row. All other columns that do not constitute the primary key are nonkey columns, and they are dependent on the primary key.

INVOICE

Inv No	InvDate	CustN	ItemNo	CustName	ItemName	ItemPrice	Qty
1001	04/14/03	212	1	Starks	Screw	\$2.25	5
1001	04/14/03	212	3	Starks	Bolt	\$3.99	5
1001	04/14/03	212	5	Starks	Washer	\$1.99	9
1002	04/17/03	225	1	Connors	Screw	\$2.25	2
1002	04/17/03	225	2	Connors	Nut	\$5.00	3
1003	04/17/03	239	1	Kapur	Screw	\$2.25	7
1003	04/17/03	239	2	Kapur	Nut	\$5.00	1
1004	04/18/03	211	4	Garcia	Hammer	\$9.99	5

INVOICE table and its columns.

There are three types of dependencies in a table:

1. **Total or full dependency:** A nonkey column dependent on all primary key columns shows total dependency.
2. **Partial dependency:** In partial dependency, a nonkey column is dependent on part of the primary key.
3. **Transitive dependency:** In transitive dependency, a nonkey column is dependent on another nonkey column.

For example, in the INVOICE table, ItemName and ItemPrice are nonkey columns that are dependent only on a part of the primary key column ItemNo. They are not dependent on the InvNo column. Similarly, the nonkey

column InvDate is dependent only on InvNo. They are partially dependent on the primary key columns. The nonkey column CustName is not dependent on any primary key column but is dependent on another nonkey column, CustNo. It is said to have transitive dependency. The nonkey column Qty is dependent on both InvNo and ItemNo, so it is said to have full dependency.

DATABASE DESIGN

Relational database design involves an attempt to synthesize the database structure to get the "first draft." The initial draft goes through an analysis phase to improve the structure. More formal techniques are available for the analysis and improvement of the structure. In the synthesis phase, entities and their relationships are identified. The characteristics or the columns of all entities are also identified, and the designer defines the domains for each column. The candidate keys are picked, and primary keys are selected from them. The minimal set of columns is used as a primary key. If one column is sufficient to uniquely identify an entity, there is no need to select two columns to create a composite key. Avoid using names as primary keys, and break down composite columns into separate columns. For example, a name should be split into last name and first name. Once entities, columns, domains, and keys are defined, each entity is synthesized by creating a table for it. A process called **normalization** analyzes tables created by the synthesis process.

NORMAL FORMS

data are repeated from row to row. For example, InvDate, CustNo, and CustName are repeated for same InvNo. The ItemName is entered repeatedly from invoice to invoice. There is a large amount of redundant data in a table with just eight rows! **Redundant data** can pose a huge problem in databases. First of all, someone has to enter the same data repeatedly. Second, if a change is made in one piece of the data, the change has to be made in many places. For example, if customer Starks changes his or her name to Starks-Johnson, you would go to the

individual row in INVOICE and make that change. The redundancy may also lead to **anomalies**.

Anomalies

A deletion anomaly results when the deletion of information about one entity leads to the deletion of information about another entity. For example, if an invoice for customer Garcia is removed, information about item number 4 is also deleted. An insertion anomaly occurs when the information about an entity cannot be inserted unless the information about another entity is known. For example, if the company buys a new item, this information cannot be entered unless an invoice is created for a customer with that new item. An update anomaly can occur if the item price changes to a new price. The price change is valid after the change date, but not before the change date.

Unnecessary and unwanted redundancy and anomalies are not appropriate in databases. Such tables are in lower normal form. Normalization is a technique to reduce redundancy. It is a decomposition process to split tables. The splitting is performed carefully so that no information is lost. The higher the normal form is, the lower the redundancy. The table in is in first normal form (1NF).

First Normal Form (1NF)

A table is said to be in first normal form, or can be labeled 1NF, if the following conditions exist:

- The primary key is defined. This includes a composite key if a single column cannot be used as a primary key. In our INVOICE table, InvNo and ItemId are defined as the composite primary key components.
- All nonkey columns show functional dependency on the primary key components. If you know the invoice number and the item number, you can find out the invoice date, customer number and name, item name and price, and quantity ordered. For example, if InvNo = 1001 and ItemNo = 5 are

known, then InvDate = 04114/03. ItemName = Washer, ItemPrice = \$1.99, CustNo = 212. and CustName = Starks.

- The table contains no multivalued columns. In a single-valued column, the intersection of a row and a column returns only one value. In a normalized table, the intersection of a row and a column is a single value. Some database packages, such as Unidata and Prime Information, allow multiple values in a column in a row, but Oracle does not. Fthe INVOICE table of in unnormalized form. The ItemNo, ItemName, ItemPrice, and Qty columns are multivalued.

A table that is in 1NF may have redundant data. A table in 1NF does not show data consistency and integrity in the long run. The normalization technique is used to control and reduce redundancy and to bring the table to a higher normal form.

Second Normal Form (2NF)

A table is said to be in second normal form, or 2NF, if the following requirements are satisfied:

- All 1NF requirements are fulfilled.
- There is no partial dependency.

As you already know, partial dependency exists in a table in which nonkey columns are partially dependent on part of a composite key. Suppose a table is in 1NF and does not have a composite key. Is it in the second normal form also? Yes, it is in 2NF, because there is no partial dependency. Partial dependency only exists in a table with a composite key.

Third Normal Form (3NF)

A table is said to be in third normal form, or 3NF, if the following requirements are satisfied:

- All 2NF requirements are fulfilled.
- There is no transitive dependency.

A table that has transitive dependency is not in 3NF, but it needs to be decomposed further to achieve 3NF. However, a table in 2NF that does not contain any transitive dependency does not need any further decomposition and is automatically in 3NF.

Other, higher normal forms are defined in some database texts. Boyce-Codd normal form (BCNF), fourth normal form (4NF), fifth normal form (5NF), and do-main key normal form (DKNF) are not covered in this text. In the following section, you will learn the normalization process by using dependency diagrams.

LESSON-3

SQL

Structured Query Language (SQL)

SQL is a Structured Query Language and is the industry standard language to define and manipulate the data in. Relational Database Management" System. In a database environment, the interactions between the Client and the Server are only through SQL. This one-point communication language.in Client-Server architecture facilitates the data base, connectivity and processing

Structured Query Language is a simple English-like language SQL is also pronounced as . "sequel" and consists of layers of increasing complexity and capability. End-users with little orno experience in data processing can learn SQL features very quickly. It is a Fourth Generation Language.

SQL was first introduced by IBM Research and was introduced into the commercial market first by Oracle Corporation in 1979. A committee at the American National,StandardsInstitute hasendorsed SQL as the standard language for RDBMS.

SQL Provides the following functionalities :

- Creation of tables.
- Querying the exact data.
- Change the data structure and the data.
- Combine and calculate the data to get required information.

Non Procedural Language

SQL is a non-procedural language and is free of logic and,procedural constructs. In SQL, all we need to say is what we want and not how to go about it. It aces not require,the.user to specify the methodology for accessing the data.SQL processes of records rather than one atatime. SQL language can be used by Database Administrators application programmers decision support personnel and management.

SQL facilitates interaction by embedding SQL Standard programming languages such as COBOL, FORTRAN, C etc. through a variety of RDBMS tools like SQL * Plus, Report Generators, Duplication Generators, Form Generators.

Database Access through SQL

SQL operates over database tables. Tables constitute tabular representation of data with data residing in the form of a spreadsheet or rows and columns. Each row has a set of data items and the kerns are called fields.

SQL * Plus

SQL Queries are sent to the Oracle RDBMS using the tool called SQL * Plus. This is the principal CLIENT tool for ORACLE. It is an environment through which any interaction with the database is done using SQL commands.

SQL * Plus program can be used in conjunction with the SQL database language and its procedural language extension FL/SQL. SQL * Plus enables the user to manipulate SQL commands and PL/SQL statements and to perform additional tasks such as to

- Enter, edit, store, retrieve and run SQL commands
- Format and print calculations, query results in the form of reports.
- List column definitions for any table
- Access and copy data between SQL databases.

Logging to Oracle

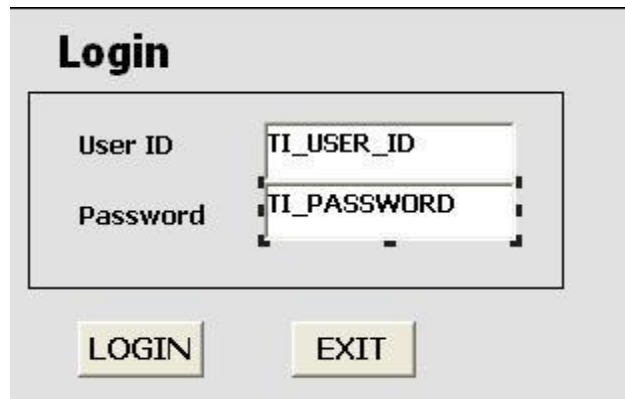
TO enter into Oracle and interact with the database using SQL * Plus, a username and a password must be given. Logging to oracle can be done by using either the menu option or by entering.

Plus 80w (From Oracle 8) in the Start - Run Option.

It prompts the user to enter a username and a password. If the system is connected to multi-user environment, the database alias name must be provided in the "Host String".

The Database Administrator configures this database alias name.

The figure explains this :



The image shows a 'Login' dialog box with a title bar. Inside the dialog, there are two input fields. The first is labeled 'User ID' and contains the text 'TI_USER_ID'. The second is labeled 'Password' and contains the text 'TI_PASSWORD'. Below these fields are two buttons: 'LOGIN' and 'EXIT'.

Shortcuts to starting SQL * Plus

While starting SQL * Plus, the username along with the password and the database alias (if required) can be given. The username and the password must be separated by a slash (/). For example consider a user called SCOTT and the password called TIGER.

If the user is connected to the personal database, SQL * Plus can be started by giving.

`PLUS SOW SCOTT/TIGER (or) SQL PLUS SCOTT / TIGER`

If the user is connected to a network, SQL * Plus can be started by giving `PLUS SOW SCOTT/TIGER @ ORACLE`

After this, the SQL prompt appears from where the commands can be entered and executed.

SQL Buffer

The area where SQL * Plus stores the most recently typed SQL commands or PL/SQL commands is called SQL Buffer. The command remains in the buffer until another command is entered. Thus, if the same command or block has to be re-executed or edited, it can be done without retyping the same. A detailed list of SQL * Plus commands are dealt later.

Note : SQL * Plus commands are not stored in the SQL Buffer and hence may not be re-run.

SQL Command Classification

Based on the type of action that each command performs, SQL commands can be broadly classified as follows:

Classifications	Description	Commands
DDL (Data Definition Language)	Is used to define the structure of a table, or modify the structure. Is used to manipulate with the data	CREATE ALTER DROP, TRUNCATE, RENAME
DML (Data Manipulation Language)	Is used to restrict or grant access to tables	INSERT, UPDATE, DELETE
DCL (Data Control Language)	Is used to restrict or grant access to tables	GRANT, REVOKE
TCL (Transaction Control Language)	Is used to complete fully or undo the transactions	COMMIT, SAVEPOINT, ROLLBACK
Queries	Is used to select records from the tables or other objects	SELECT

Before discussing about creating tables, a detail description about data types is dealt with

Data types

Each literal or column value manipulated by Oracle has a datatype. A value's datatype associates a fixed set of properties with the Value. Broadly classifying the datatypes, they can be of two types :

- BUILT-IN
- USER — DEFINED (dealt in a later Chapter)

Built-in datatypes are predefined set of data types set in Oracle. Based on the type of data that can be stored, built-in datatypes can be classified as

- Character Datatypes
- Numeric Datatype
- Date Datatype
- ORaw Datatype
- Long Raw Datatype
- Lob Datatype

Character Datatype

Char (n)

Char datatype is a fixed length character data of length n bytes.

Default size is 1 byte and it can hold a maximum of 2000 bytes. Character datatypes pad blank spaces to the fixed length if the user enters a value lesser than the specified length.

Syntax

Char (n)

Example :

X char (4) stores upto 4 characters of data in the column X.

Varchar 2 (size)

Varchar 2 datatypes are variable length character strings. They can store alpha-numeric values and the size must be specified. The maximum length of varchar 2 datatype is 4000 bytes. Unlike char datatype, blank spaces are not padded to the length of the string. So, this is more preferred than character datatypes since it does not store the maximum length.

Syntax

Varchar 2 (size)

Example :

X varchar2 (10) stores upto 10 characters of data in the column X.

Numeric datatypes

Number

The number datatypes can store numeric values where p stands for the precision and s stands for the scale. The precision can range between 1 to 38 and the scale ranges from - 84 to 127.

Syntax

Number (p, s)

Example :

Sal number — Here the scale is 0 and the precision is 38.

Sal number(7) — Here the scale is. 0 and the number is a fixed point number of 7 digits

Sal number (7,3) — Stores 5 digits followed by 2 decimal points.

DATA datatype

Date datatype is used to store date and time values. The default format is DD-MON-YY. The valid data for a date ranges from January 1,4712 BC to December 31,4712 AD. Date datatype stores 7 bytes one each for century, year, month, day, hour, minute and second.

RAW Datatype

RAW (n)

RAW datatype stores binary data of length n bytes. The maximum size is 255 bytes. Specifying the size is a must for this datatype.

Syntax

Raw (n)

LONG Datatype

Stores character data of variable length upto 2 Gigabytes (GB) or $2^{31}-1$

LONG RAW Datatype

stores upto 2 Gigabytes (GB) of raw binary data. The use of **LONG** Values are restricted.

The restrictions are :

- A Table cannot contain more than one LONG column.

- LONG columns cannot appear in Integrity constraints (dealt later)
- They cannot appear in WHERE, ORDER BY clauses of SELECT statements
- Cannot be a part of expressions or **conditions**.
- Cannot appear in the SELECT list of CREATE TABLE as SELECT.

LOB Datatypes

In addition to the above datatypes, Oracle8 supports LOB datatypes. LOB is the acronym for LARGE OBJECTS. The LOB datatypes stores upto 4 GB of data. This datatype is used for storing video clippings, large images, history documents etc..

Create, View, Manipulate data

Data Creation through SQL

This section deals with creation of tables, altering its structure, inserting and retrieving records and querying complex data using SQL

Using the CREATE TABLE- Command

Oracle Database is made up of tables that contain rows (horizontal) and contains (vertical). Each column contains a data value at the intersection of A row and a column the table definition contains the name of the attribute (property of field) **and the type** of the data that the column **contains** To create a table, use CREATE TABLE command. CREATE Command is used to define the structure of a table or any object.

Syntax:

```
CREATE TABLE <table name.> (column 1 datatype, column 2 datatype....c.);
```

Here, tablename refers. to the name of the table or entity, column Hi the name the-first column, column2 the name of the second column and so on. For each column there must be an appropriate data type which describes the type of data it can hold: The statement terminated by a semi-colon.

The following example illustrates the Creation of a table:

Example

```
Create table EMPLOYEE ( Empno    NUMBER
                        Empname CHAR (10),
                        Doj       DATE
```

This would display

Table created

In the above example, an entity called EMPLOYEE is **created**. It contains columns *Emp* to that can hold numeric data, *Empname* that contains character data and *Doj* that contains the type of data. Table names are case-insensitive.

The structure of the data would look like.

Name	
EMPNO	NUMBER
EMPNAME	CHAR(10)
DOJ	DATE

While creating tables, consider the following points

- Tablename must start with alphabet
- Tablename length must not exceed 30 characters
- No two tables can have the, same name
- Reserved words of Oracle are not allowed.

Viewing the Table structure

After creating the table, viewing the structure can be done using **DESCRIBE** followed by the name of the table.

Syntax:

```
DESC [rile] <tablename>
```

Example:

```
DESC EMPLOYEE
```

The output would look like

Name	Type
EMPNO	NUMBER
EMPNAME	CHAR (10)
DOJ	DATE

Using the ALTER TABLE Command

A table's structure can be altered using the **ALTER** Command. The Command allows the structure of the existing table to be altered by adding new columns or fields dynamically and modifying the existing fields datatypes. Using this command, one or more columns can be added.

Syntax:

Alter table <tablename> add (column1 datatype, Column 2 datatype

Alter table<tablename>modify (column 1 datatype. column 2 datatype)

Example :

Consider the previous example where a new column called Salary is to be added. **ALTER TABLE** employee **ADD** (salary **NUMBER**).

Using INSERT command

Insert command is used to add one or more rows to a table. The values are separated commas and the values are entered in the same ORDER as specified by the structure of the table. Inserting records into tables can be done in different ways:

- Inserting records into all fields
- Inserting records into selective fields
- . Continuous Insertions.
- Inserting records using SELECT statement.

Case 1:

Consider inserting records onto all the fields in the table.

Syntax:

Insert into <tablename> Values (value1, value 2, value 3..);

Here, the number of values must correspond to the number of columns in the table.

Example :

```
INSERT INTO Employee VALUES (1237,'Kalai',10 MAR-2000, 5000);
```

The message displayed will be:

1 row created.

This command inserts the record Where the employee number is 1000, his name is Jack and so on. Always character data must be entered within single quotes.

In the above example, the column called *doj* contains a date datatype while inserting data values, the values must be enclosed within quotes. The standard format of entering the date values is DD-MON-YY'.

Note : Any value other than mere number must be placed within single quotes. Otherwise, it treats the value as a column name.

Case 2:

Consider inserting values into selective fields.

Syntax:

```
Insert into <tablename> (Selective column1, selective column2 ) values (value 1, value 2)
```

Example:

```
INSERT INTO Employee (empno, empname) VALUES (1330, 'Saravanan');
```

Displays the feedback as

1 row created:

Case 3:

Consider continuous insertion of records. In order to insert continuously, use "&" (ampersand).

Syntax :

Insert into <tablename> Values (&Coll, &Co12, &Co13...);

Oracle prompts the user to insert values onto all the columns of the table. The following example illustrates this..

Example

```
INSERT INTO employee VALUES (&eno, &name, &doj, &sal);
```

Output will be:

Enter value: for eno: 1247

Enter valuefor name: Mena

Enter value for doj name:10-jan-2000

Enter value for sal : 5000

old I: insert into employee values (&eno, '&name', &doj, &sal)

new I : insert into employee values (1247, 'meena', 10-jan-2000', 5000)

I row created.

Here, instead of enclosing the character values. in Quotes each time. it is enough *if* it is given while using the Insert command. Now the same command can be re-executed by giving /(slash) as long as this. is the latest command that is there in the buffer.

Now, consider inserting_ records continuously for selective fields. This is similar to case 2 Insert into <tablename> (selective column 1, selective column2) Values (Coll, &co12);

The following example inserts records into the *empno and empname* columns.

Example :

```
INSERT INTO Employee(empno, empname) VALUES (&eno, '&name');
```

Output

Enter value for eno : 1440

Enter value for name : Diana

old I: insert into employee values (&eno, '&name');

new I: insert into employee values (1440,'Diana');

I row created.

Case 4:

Multiple Records can be inserted using a single Insert command along with Select statement. This case is dealt after the section on Select Statement.

Note : Using Insert and Values combination, only one record can be inserted at a time.

Retrieving Records — Using the. SELECT Statement.

Retrieving data from the database is the most common SQL operation. A database retrieval is called a *query* and is performed using SELECT statement. A basic SELECT statement contains two clauses or parts

Select some data (columnname (s)) FROM a table or More tables (table name(s)) Retrieval of records can be done in various ways:

- Selecting all records from a table
- Retrieving selective columns for all records from a table
- Selecting records based on conditions
- Selecting records in a sorted order

Consider the first case of selecting all the records from the table.

Example

```
SELECT empno, empname, doj, salary FROM employee;
```

Here, all the column names are given in the SELECT clause. This can be further simplified by giving '*' as follows:

Example

```
SELECT * FROM employee;
```

This would display:

EMPNO	EMPNAME	DOJ	SALARY
1237	Kalai	10-MAR-2000	5000

1330	Saravanan		
1247	Meena	10-JAN-2000	5000
1440	Diana		

‘*’ INDICATED ALL THE COLUMN NAMES.

Note in the above display, there are no values entered in DOJ and Salary Column for the employee 1001 and 1004. Here the values in these columns are considered to have NULL values. Anytime it can be updated using the Update Command.

In the second case, the column names must be specified in the SELECT statement.

Syntax :

SELECT Col1, Col2.,

FROM <tablename>; **Example:**

SELECT empname, salary FROM employee;

The records would be displayed as follows :

EMPNAME	SALARY
Kalai	5000
Saravanan	
Meena	5000
Diana	

This statement retrieves the column values of empname and salary.

Conditional Retrieval

Conditional retrieval enables selective rows to be selected. While selecting rows, restriction can be applied through a condition that governs the selection. An additional clause called WHERE must be given along with the SELECT statement to apply the condition to select a specific set of rows! The order of precedence first goes to the WHERE clause and the records that match the condition are alone selected.

Syntax:

Select (column name (s)) FROM (table name(s)) WHERE condition(s)

Consider selecting employee records whose salary is equal to or greater than 3000. The query can be -written as.

Example:

```
SELECT empno, empname, salary FROM employee where salary >=3000;
```

The records selected Will be,

EMPNO	EMPNAME	SALARY
1237	Kalai	5000
1247	Meena	5000

Example :

```
SELECT * FROM employee WHERE salary = 1000;
```

The display would be no rows selected since there are no records matching, the condition specified in the WHERE clause.

Retrieving Records in a sorted order

Records selected can be displayed either in ascending order or in descending order based on the column specified. ORDER BY clause is used to perform this operation,

Syntax :

```
Select (column name(s));  
FROM (table name(s)) WHERE condition(s)  
ORDER BY <column name(s)>
```

Example :

```
SELECT * FROM employee ORDER By empname;
```

This would display

EMPNO	EMPNAME	DOJ	SALARY
1440	Diana		
1237	Kalai	10-MAR-2000	5000
1247	Meena	10-JAN-2000	5000
1330	Saravanan		

There is a difference in the display. This query displays all the records in the employee.

table sorted in ascending order of the employee name. By default ascending is the order in which the records are displayed. If the records need to be displayed in descending, use DESC along with the Column name. For example,

```
SELECT * FROM employee ORDER BY empname DESC; .
```

Here sorting is done. In descending order. So the display will be totally, different as shown below

EMPNO	EMPNAME	DOJ	
1440	Saravanan		
1247	Meena	10-JAN-2000	5000
1237	Kalai	16-MAR-2000	5000
1330	Diana		

Sorting records can be done on more than one column. In this case, sorting is done on the first column and then within that sorting is done on the second Column. The following example illustrates this.

suppose that sorting is to be done on the salary column in ascending order and then on the empname column in descending order.

```
SELECT empno, empname, salary FROM employee ORDER BY salary, empname DESC;
```

This stores first the salary and within that the employee name which would look like:

EMP NO	EMPNAME	SALARY
1247	Meena	5000
1237	KALAI	5000.
1440	Saravanan	
1330	Diana	

Note : DESC denotes descending order and this is not the same as DESC in SQL*

Plus. Copying the Structure with Records.

If the Structure of one table has to be copied on to another table along with the records, the Create table statement is used in combination with the select statement. The structure along with the records is copied on to the second table.

Syntax:

CREATE TABLE <tablename2>AS SELECT <columnlist> FROM <tablename1> where <conditions>]

Example:

Consider creating a table called employee1 whose structure is the same as employee table. To create this use.

CREATE TABLE employee1 AS SELECT * FROM EMPLOYEE,

Displays the feedback;

Table created.

The structure of the employee 1 would be the same as the structure of employee table. **The records will also be the same.** To view the records of employer table, use

SELECT * FROM employee1;

This would display

EMPNO	EMPNAME	DOJ	SALARY
1237	Kalai	10-Mar-2000	5000
1330	Saravanan		
1247	Meena	10-JAN-2000	5000
1440	Diana		

If only specific columns need to be copied use;

CREATE TALE employee2 AS SELECT empno, empname **FROM EMPLOYEE;**

The statements given above create new tables called employee 1 and employees2 respectively. In the case of employee' table, the structure, which exists in the employeetable, is copied and the records are inserted. In the case of employee2 table, two columns are copied fromthe employee name with the records and me structure. The above Statements can alternately written as.

. CREATE TABLE <tablename>

and

INSERT INTO <tablename>SELECT <columnlist> FROM <tablename>

As(explained in the section on Inserting records using SELECT Statement).

Copying the Structure.

The Structure of one table can be copied on to another table without the records being copied. In order to do this, along with the SELECT statement, add a WHERE clause which yields to any FALSE condition. The following example explain this

```
CREATE TABLE employee 3 AS SELECT * FROM employee WHERE 1=2;
```

This statement creates a table called Employee3 whose structure is the same as Employee but the records are not copied since WHERE clause evaluates to FALSE.

LESSON-4

Inserting records using SELECT statement.

Records can be selected and inserted from one table to another table using INSERT! SELECT statement.

Syntax:

```
INSERT INTO <tablename> SELECT <columnlist> FROM <tablename>
[WHERE<conditions>],
```

Example

```
INSERT INTO employee3 SELECT * FROM employee WHERE
salary>12000;-
```

The above example copies records from employee table to employee3 table where the records meet the criteria specified in the WHERE clause.

Modifying data — Update Command

The UPDATE command is used to Modify one or a set of rows at a time. An UPDATE statement consists of 2 clauses—UPDATE clause followed by a SET clause and an optional 3rd clause - the WHERE clause. The WHERE clause specifies a condition, which is optional.

UPDATE statement allows the user to specify the table name for which the rows are to be Modified. The SET clause sets the values of one or more columns as specified by the UPDATE statement. Without a WHERE clause updates all the records in the table.

Syntax:

```
UPDATE <tablename>
SET<col1>=value<col2>=value,...
WHERE<condition>
```

Example :

Consider the table employee2, add a column called deptno to this table as shown below

ALTER TABLE employee 2 ADD deptno NUMBER;

Before the update command is issued on issuing.

SELECT * FROM employee2;

This display will be:

EMPNO	EMPNAME	DEPT NO
1237	Kalai	
1330	Saravanan	
1247	Meena	
1440	Diana	

There are no values in the deptno column. Since INSERT is used to insert new records, UPDATE performs the value updation, Now, consider the following UPDATE statement.

UPDATE employee2

SET deptno=10;

This statement updates all the records of employee table with deptno as 10 and displays number of records that were modified. Here the display will be,

4rows updated.

Example:

UPDATE employee2

SET deptno=20 WHERE empno>1003;

The deptno column is updated for all the records that satisfy the condition specified in the WHERE clause. After this command is issued, the select statement for this table would yield.

SELECT * FROM employee2;

ENO	EMPNAME	DEPT NO
1217	Kalai	10
1330	Saravanan	10
1247	Meena	10
1440	Diana	20

To update more than one column, separate the columns with a ',' as illustrated below. .

```
UPDATE employee1
```

```
SET salary = 6000, doj='09-SEP-2000' WHERE doj is NULL;
```

Here, the salary is Updated to 6000 and the doj is changed to 09-SEP-2000 for employees whose the date of joining is NULL. After updation the selection from the table would yield.

EMPNO	EMPNAME	DOJ	SALARY
1237	Kalai	10-MAR-2000	5000
1330	Saravanan	09-SEP-2000	6000
1247	Meena	10-JAN-2000	5000
1440	Diana	09-SEP-2000	6000

Removing Data-DELETE Statement

Delete statement removes the rows from the table. The command has a DELETE from clause and an optional WHERE clause. The DELETE FROM statement names the table on which the deletion operation is to be performed and the WHERE clause specifies the condition.

Syntax:

```
DELETE FROM <tablename> [WHERE <condition>]
```

Example

```
DELETE FROM employee2 WHERE deptno=20;
```

This statement removes all the records from the employee 2 table for which the deptno is 20.

The following example removes all the records from the table.

Example:

```
DELETE FROM employee 2;
```

Working with Transactions

Transaction means a logical unit of work that would make sense only on the completion of all the operations as a whole. This means that, either all the operations in the unit are completed fully or none of them are completed.- In Oracle, can be any Data

Manipulation Language Statements. SQL offers two commands to simulate real-life transactions in the database. The commands are:

- COMMIT
- ROLLBACK

COMMIT completes the transaction done by making changes permanently to the database and initiates a new transaction. ROLLBACK reverts back the changes made and completes the transaction. The following example illustrates this in a better way :

```
INSERT INTO <tablename> Values(...);  
INSERT INTO <tablename> Values (...);  
UPDATE <tablename> SET .....  
COMMIT  
Displays  
Commit complete
```

All the changes are made permanently in the database. The two Insert statements and an Update statement effect the changes and the commit completes the transaction.

Now, consider the following :

```
DELETE FROM <tablename>,
```

When this statement is issued, the records are deleted from the table. But if this is done accidentally, records can be retrieved immediately after the DELETE statement by issuing the statement.

```
ROLL BACK;
```

This statement reverts back the changes made to the table and the records are not deleted.

Note : All DDL, DCL, statements perform an Implicit Commit.

Temporary Markers

Now, assume that after the DELETE statement is one INSERT statement is issued as

given below:

```
DELETE FROM <tablename> WHERE....  
INSERT INTO <tablename> VALUES(.....);  
ROLLBACK;
```

Here both the INSERT and the DELETE operations are un-done. In order to avoid this kind of situations SAVEPOINTS are used. SAVEPOINTS are temporary markers that are used to divide a lengthy transaction into a smaller ones. They are used along with ROLLBACK.

Syntax:

```
SAVEPOINT<savepoint id>;
```

Where savepoint_id is the name of the savepoint. Multiple savepoints can be created within a transaction. Re-pharsing the previous example,

```
DELETE FROM <tablename> WHERE<condition>;  
SAVEPOINT S1;  
INSERT INTO <tablename> VALUES<...>;  
ROLLBACK To s1;  
.....
```

In this case, only the deleted statements are rolled back as savepoint has been used after the Delete statements.

Note : SAVEPOINTS can be used with in transaction. After a transaction is completed, the same save points cannot be used in the next transaction.

DCL statements are dealt in the chapter on database objects.

Differences between Delete and Truncate

The following table illustrates between Truncate and Delete.

Truncate

Delete

Deletes all the records	Can delete all the records or selective rows alone
This is a DDL statement	This is a DML statement
Commits Implicitly	Explicit commit is required and
Hence ROLLBACK	
Hence Not Possible	can be used to rollback
SQL*PLUS Commands DESC	Used to describe the structure of an object
SET PAGE SIZE N	Sets the pagesize to n length
set feedback[ON/OFF]	SQL displays the feedback for every statement executed. By turning it off, the feedback is not displayed on the screen.
SET HEADING[on/off]	In query displays, the heading of the column name is displayed. By turning it off, the heading is not shown on the screen. it can be turned on again.
SET LINE SIZE x	Set 'x' number of lines to be displayed on the screen
SET PAUSE on/off	Pauses the display till the user presses <Enter> key Executes the latest command in the buffer.

Short Summary

- SQL is a Structured Query Language used for all RDBMS.
- SOL * Plus is a tool that is used to Work With and the commands are executed.
- SQL* Plus statements are not stored in the buffer.
- DDL — Data Definition Languages that defines the structure of an object.
- They are CREATE, ALTER, DROP, TRUNCATE, RENAME

Constraints :

Data security and Data Integrity are the most important factors in deciding the success of a system. Constraints are a mechanism, used by Oracle to restrict invalid data from being entered into the table and thereby maintain the integrity of the data. They are otherwise called a Business Rules. These constraints can be broadly classified into 3 types:

- Entity Integrity Constraints
- Domain Integrity Constraints
- Referential Integrity Constraint's

Entity Integrity Constraint

Entity Integrity constraints can be Classified as

- PRIMARY KEY
- UNIQUE KEY

Choosing a table's Primary Key

A primary key allows each row in a table to be uniquely identified and ensures that no duplicate rows exist and no null value are entered. Selecting a primary key needs the following: guideline:

- Choose a column whose data values are unique
- Choose a column whose data values never change.

A primary key value is used to identify a row in the table. Therefore, primary key values must not contain any data that is Used for any other purpose. Primary key can contain one or more columns of the same table. Together they form a composite Primary Key.

Using UNIQUE Key Constraints

Unique Key constraint is used to prevent the duplication of key values within the rows of a table. If values are entered into a column defined with a unique key, repeating the same data for that column is not possible but it can contain any number of null values. According to Oracle one null is not equal to another null.

Domain Integrity Constraints

Domain Integrity constraints are based on the column values and any deviations or violations: are prevented. The two types of Domain Integrity Constraints are

- Not Null Constraints
- Check Constraints

Choosing NULL Constraint

By Default all columns can contain null values. NOT NULL constraints are used for columns that absolutely require values at all times. NOT NULL constraints are often combined with other types of constraints to further restrict the values that can exist in specific columns of a table

Choosing Check Constraints.

Check Constraints are used to check whether the values in the table satisfy the criteria specified for that column. They contain conditions. The conditions have the following limitations.

Conditions must be a Boolean expression that can be evaluated using the values in the record being inserted or updated

Conditions must not contain sub-queries.

Conditions cannot contain any SQL functions.

Conditions cannot contain pseudo columns.

Referential integrity Constraint

This constraint establishes the relationship between tables. A single or combination of columns, which can be related to the other tables, is used to

perform this operation. Foreign key is used to establish the relationship. This kind of relationship can be referred to as a Parent-Child relationship.

The table containing the referenced Key from while other tables refer for values is called the Parent table and the table containing the foreign key is called the child table.

Adding Constraints

Constraints can be added in two different ways. There are

- Adding at the time of creating tables
- Adding after creating tables

The next section deals with the adding constraints at the time of creating tables. Every constraints contains a name followed by the type of the constraint

Adding Entity Integrity Constraints

Both Primary Key and Unique Key constraints can be added at the time of creation of tables. Let us consider creating a table called item master, which contain item code, item_name and unit_price.

Example

```
Create table Item master (item _code number primary key,  
                          Item name varchar2 (20) unique,  
                          Unit Price number (9,2)  
                          );
```

The table is created along with the constraints. If a constraint is given without specifying the name of the constraint, Oracle by default assigns a name to the constraint that is unique. The constraint starts with 'SYS_C' followed by some numbers.

After creation, records are inserted into the table as follows:

```
INSERT INTO item_master VALUES (1, 'Pencils', 2.50);
```

If the same command is executed again, it raises an error.

```
INSERT INTO item master VALUES (1, 'Pencils', 2.50);
```


*

ERROR at line 1.

ORA-00001: unique constraint (HEMA.SYS_C00769) violated

By looking at this error message, the user may not understand which value is violated. In order to avoid this situation, a name has to be provided for every constraint that is easy to read. Refining the above example

```
Create table Item_master (item_code number CONSTRAINT  
pkitem_code primary key.
```

```
Item name varchar2(20) CONSTRAINT unque name unique,  
Unit_Price number (9,2)  
).
```

Naming the constraints always provide better readability.

Adding Domain Integrity Constraints

The user has to necessarily provide values for the columns containing NOT NULL values. NOT NULL constraint is ideal in cases where the value for the column must exist. Consider an organization that needs to store all the employee information. In this case, the employee name column cannot be left blank.

What is NULL?

NULL means some un-known value. NULL values can always be updated later with some values. Remember it is not equal to Zero.

Example

```
CREATE TABLE employee master(empno NUMBER, empname  
varchar2 (20)CONSTRAINT NN_ENAME NOT NULL);
```

While inserting records to this table,

```
INSERT INTO employee Master VALUES (1, null);
```

```
INSERT INTO employee_master VALUES (1, null)
```

ERROR at line 1

```
ORA01400 : cannot insert NULL into ("HEMA",  
"EMPLOYEE_MASTER", EMPNAME")
```

In the above situation, since empname column has a constraint, it ensures that sore values are entered and hence the error is raised,

Check constraints are used to perform the checking of condition used on the value entered. The CHECK constraint is used to enforce business rules as shown an the followingexample

Example.

```
CREATE TABLE employee_master (empno NUMBER, empname
VARCHAR2 (20) sex CHAR(1) CONSTRAINT chk_sex CHECK (SEX IN('M',
'F', 'm', 'f')));
```

In the sex column, valid values are 'M', 'F', 'm', 'f'. If values entered do not match the condition, error message appears on the screen. As shown below:

```
INSERT INTO employee_masterr VALUES (2, 'Aditya', 'b');
```

```
INSERT INTO envloyee_master VALUES (2, 'Aditya', 'b');
```

*

ERROR. at line 1:

ORA - 02290 : Check constraint (HEMA.CHK_SEX) violated

Another example for using CHECK Constraint is given below. This example checks for constraint on salary that an organization provides. The minimum salary that the organization provides 1000 and the maximum is 7000. To check for this type of constraint use,

Example

```
CREATE TABLE employee_ master (Empcode NUMBER, empname
VARCHAR 2(20), Salary NUMBER CONSTRAINT CHIK_SAL CHECK
(salary BETWEEN 1000 and 7000));
```

In the above example, BETWEEN operator is used to check if the salary lies within the range of 1000 to 7000.

Relationships between Parent and Child tables

In order to establish a parent-child or master.-detail relationship, referential integrity constraints are used. The relationship can be from one parent to one or more than one child. The establishment of a Parent-Child relationship has some restrictions.

Parent table must be created before creating the child table.

The parent table column, which is referenced in the child table must contain either Primary Key or Unique Key Constraints.

Consider a table called Customer which contains the details of the customers who. have ordered for products and an Order table that maintains the information about the orders placed. The structures of the tables look like :

CUST_MASTER

CCODE	NUMBER(5)
NAME	VARCHAR2(40)
ADDRESS	VARCHAR2(100)

ORDER_MASTER

OCODE	NUMBER(4)
CCODE	NUMBER(5)
ODATE	DATE
OVAL	NUMBER

Here the Cust_master is the master table and order_master is the child table.

```
CREATE TABLE cust_master (ccode NUMBER (5) CONSTRAINT  
Pk codePRIMARY KEY, name VARCHAR 2(40); Address VARCHAR2  
(100));  
CREATE TABLE order_master (ocode NUMBER(4),ccode  
NUMBER (5) CONSTRAINT fk_code REFERENCES cust_master (ccode).  
odate DATE, oval NUMBER);
```

In the above example, now that the master table must be created first and Only then the child table is created. To establish the relationship. add REFERENCES clause which refers to the master table.

If the value in the child table refers to a non-existing record in the parent table, an error is raised

Example

```
INSERT INTO cust_master VALUES (1, 'Arun', 'Chennai');
```

```
INSERT INTO cust_master VALUES (2, 'Ramesh', 'Bangalore');
```

Inserting records into child table.

```
INSERT INTO order_master VALUES (1, 1, '01-JAN-2000', 3000"
```

The record is inserted in to the child table since the value 1 or customer code column refers to the parent record in Cust_master table

```
INSERT INTO order_master VALUES (2,4 '01-Jul-2000', 1000);
```

```
INSERT INTO order master VALUES (2,4 '01-3111-2000', 1000 * ERROR  
at line 1:
```

```
ORA - 02291: integrity constraint (HEMA.FK_CCODE) violated parent key  
not found. This raises to an error since there is no reference record in the  
master table for Customer.
```

code 4.

The above - discussed types of constraints are called column level constraints because they are defined along with the table definition. Another type of constraint which is shown below is called the Table Level Constraint which is shown below .

```
CREATE TABLE employee_master (Empcode NUMBER, embname  
VARCHAR2(20); Salary NUMBER, Sex NUMBER,  
CONSTRAINT pk empno PRIMARY KEY (empcode), CONSTRAINT  
chk_sal CHECK (salary BETWEEN 1000 and 7000) CONSTRAINT  
chk_sex CHECK (sex IN ('M', 'F', 'm', 'T'));
```

The table - level constraint for adding a referential integrity constraint is shown below. CREAM TABLE order_master (ocode NUMBER (4), ccode NUMBER (5), odate DATE, oval NUMBER,

```
CONSTRAINT fk_ccode FOREIGN KEY (ccode) REFERENCES
cust master (ccode));
```

While creating Table-level constraints for referential Integrity constraint, FOREIGN KEY along with REFERENCES must be used.

Note: NOT NULL constraints cannot be given for Table-level constraints

Using ALTER Statement

Constraints can be created for existing tables using ALTER statement. The following example illustrates this. Assume that the tables order master, employee master, item master are created without constraints.

To add a primary key constraint,

Syntax:

```
ALTER TABLE <tablename> ADD [CONSTRAINT <constraint name>]
PRIMARY KEY (colname);
```

Example

```
ALTER TABLE employee master ADD CONSTRAINT pk_empno PRIMARY
KEY (empcode);
```

To, add a unique constraint,

Syntax:

```
ALTER TABLE <tablename> ADD [CONSTRAINT <constraint name>]
UNIQUE (colname);
```

Example

```
ALTER TABLE item_master ADD [CONSTRAINT unq_name UNIQUE (item
name);
```

Check constraints are added using,

Syntax:

```
ALTER TABLE <tablename> ADD [CONSTRAINT .<constraint name>]  
CHECK (colname<condition>);
```

Example

```
ALTER TABLE employee master ADD CONSTRAINT ck_sal CHECK  
(salary BETWEEN 1000 AND 7000);
```

In order to add a referential integrity constraint use FOREIGN KEY and REFERENCES keyword.

Syntax:

```
ALTER. TABLE <tablename> ADD [<CONSTRAINT <constraint name>]  
FOREIGN KEY (colname)REFERENCES <mastertable>(colname);
```

Example:

```
ALTER TABLE order_master ADD CONSTRAINT fk_custcode FOREIGN  
KEY (ccode) REFERNCES oust master (ccode);
```

Inserting a NOT NULL constraint using ALTER statement cannot be done directly. Either MODIFY or CHECK is required to do this functionality.

Example

```
ALTER TABLE employee_rmaster MODIFY (empname varchar2 (20) not null)
```

Enabling Constraints

The following CREATE TABLE and ALTER TABLE statements both define and enable integrity constraints:

```
CREATE TABLE emp (  
    Empno NUMBER (5) PRIMARY KEY,...);  
  
ALTER TABLE emp 1  
    ADD PRIMARY KEY (empno);
```

The above two statements add the primary key constraint. The first addsthe primary key constraint at the time of creation of the table. The next statement tries to add the constraintalter the table is created. Here if the records are not available,

the constraint is added. If records are already present, the ALTER TABLE statement will fail.

Disabling Constraints

The following CREATE TABLE and ALTER TABLE statements both define and disable integrity constraints:

```
CREATE TABLE emp (  
    empno NUMBER(5) PRIMARY KEY DISABLE, . . .);  
  
ALTER TABLE emp  
    ADD PRIMARY KEY (empno) DISABLE
```

The above two statements describe the employee number to be primarykey but initially it is disabled.

Enabling and Disabling Defined Integrity Constraints

Use the ALTER TABLE command to

Enable a disabled constraint, Using the ENABLE clause

Disable an enabled constraint, using the DISABLE clause

Enabling Disabled Constraints

The following statements are examples of statements that enable disabled integrity constraints:

```
ALTER TABLE dept  
    ENABLE CONSTRAINT dname_ulcey;
```

```
ALTER TABLE dept  
    ENABLE PRIMARY KEY.
```

```
ENABLE UNIQUE (dname, loc);
```

The above 2 statements triesto enable the constraints:

AnALTERTABLE statement that attempts toenable an integrity constraint falls When the rowsof the table violate the integrity constraint.

Disabling Enabled Constraints

The following statements are examples of statements that disable enabled integrity constraints;

```
ALTERTABLE dept
    DISABLE CONSTRAINT dname_ukey;
TABLE dept
    DISABLE PRIMARY KEY,
    DISBLE UNIQUE (dname, loc);
```

The reversal of enable is disable. These two statements disables the enabled constraints. Unlike ENABLE, this does not perform any checking while disabling.

Dropping Integrity Constraints

Drop an integrity constraint if the rule that it enforces is no longer true or if the constraint is no longer needed. Drop an integrity constraint using- the ALTER TABLE command and the DROP clause. For example the following statements drop integrity, constraints:

```
ALTER TABLE dept
    DROP UNIQUE (dname, LOC);
ALTER TABLE emp
    DROP PRIMARY KEY,
    DROP CONSTRAINT dept_fkey;
DROPTABLE emp CASCADE CONSTRAINTS;
```

The two Alter statements drops the unique, primacy keys. The third statement drops the EMP table along with all the constraints.

This chapter gives an insight into the SQL operators, expressions in Oracle and all the important functions that can be used in Oracle applications through SQL. This chapter also deals with the Aggregate Functions that are used to group values and display as a singular value. The usage of Group By and Having clause are also dealt with.

- ❖ Expressions
- ❖ Logical Operators
- ❖ Arithmetic Operators
- ❖ Comparison Operators
- ❖ Built-in Scalar Functions
- ❖ Numeric Functions
- ❖ Character Functions
- ❖ Date Functions
- ❖ Conversion Functions
- ❖ Aggregate Functions

Operators

Expressions are a combination of formulae, constants and/or variables using operators. They are used with operators to perform some action.

Operators

An operator is used to manipulate individual data items and return a result. These sets of data items are called *operands*. Operators are represented by special characters or by keywords. For example a "multiplication operator is represented by '*'. The following section of this chapter deals with various operators. The various kinds of operators are:

- ❖ Arithmetic
- ❖ Character
- ❖ Comparison/Relational
- ❖ Logical
- ❖ Set

Arithmetic Operators

Arithmetic operators are used to perform operations on numeric values. The result of the operation is a numeric value. Some operations can be used for calculating date arithmetic values. The following table lists out the arithmetic operators.

Operator	Purpose	Example
+	Adds values	Select salary + 1000 from employee
-	Subtract values	Select qoh-qty_ord from order_tab
*	Multiplies the values	Select salary, salary*. 01 from employee;
/	Divides the values	Select marks/ 100 from student.

These operators can be given in conjunction with each other. In these situations, multiple operators must be used according to the precedence rules within parentheses. If the parentheses are not provided the order in which arithmetic operators take place changes.

Character Operators

Character operators are used in expressions to manipulate character strings. Concatenation operator(||) is used to perform this operation.

Example:

```
SELECT This is an '||' Example for '||' Concatenation operator'
```

```
FROM DUAL;
```

```
Displays.
```

```
THIS IS AN EXAMPLE FOR CONCATENATION OPERATOR
```

```
This is an Example for Concatenation operator.
```

In this example, a system table called 'DUAL' is used. This table contains one column.

The result of concatenating three character strings is another character string. If all the strings are of character datatype, the resultant also contains the character datatype and is restricted to 255 characters.

Relational Operators:

Certain operators are used to perform comparisons between values. The result of the comparison or relational operator containing an expression will either be TRUE, FALSE or unknown. The following are the list of relational operators.

Operator	Purpose	Example
=	Equality test	SELECT empno, salary FROM employee WHERE deptno=10
!=, <>	Un-equality test	SELECT * FROM employee WHERE salary!=4000
>	Greater	SELECT * FROM employee than specified value WHERE salary>2000
<	Lesser than specified value	SELECT * FROM students WHERE marks<40
>=	Greater than or equal to	SELECT * FROM product WHERE unit_price<=4
<=	Lesser than or equal to	SELECT * FROM product WHERE unit_price<=4
Between	The value lies within the range	UPDATE student SET grade='A' WHERE marks BETWEEN 80 AND 90
IN	Displays records that satisfy the list of values	SELECT * FROM employee WHERE deptno IN (10,20,30) (Records where the department number is either 10 or 20 or 30 are displayed)
IS[NOT]	Is used to check for NULL Values	SELECT * FROM employee WHERE empname
IS NULL	The only operator checks NULL	That Value.

LIKE Matches a specified pattern. To match a single character, Use '_' and for multiple Characters use '%'

SELECT empname FROM employee WHERE ename LIKE '_A%'
(Displays employee names Where the second letter of the name is 'A')

Logical Operators

A Logical Operator is used to combine the results Of two or more conditions a produce a single result based on them. Following are the set of Logical Operators.

- ❖ AND
- ❖ OR
- ❖ NOT

AND Operator

Returns TRUE if both the condition are TRUE. Otherwise, it returns FALSE

Example:

```
SELECT * FROM emp WHERE deptno=10 AND sal >=4000;
```

This would display.

EMPNO	ENAME	JOBMGR	HIREDATE	SAL	COMM	DEPTNO
1383	Saran	PRESIDENT	7-NOV-81	5000	10	

This example checks for both the conditions and the records that match both the conditions are alone displayed. All the records with deptno 10 and sal greater than or equal to 4000 are displayed.

OR Operator

- ❖ Returns TRUE if either of the conditions evaluates to TRUE. Otherwise it returns FALSE.

Example

```
SELECT * FROM emp WHERE deptno=10 OR deptno=20;
```

This display would be as, follows:

EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM	DEPTNO
7839	KING	PRESIDENT		17-NOV-81	5000	10	
7782	CLARK	MANAGER	7839	09-JUN-81	2450	10	
7566	JONES	MANAGER	7839	02-APR-81	2975	20	
7902	FORD	ANALYST	7566	03-DEC-81	3000	20	
7369	SMITH	CLERK	7902	17-DEC-80	800	20	

7788	SCOTT ANALYST	7566	09-DEC-82	3000	20
7876	ADAMS CLERK	7788	12-JAN-83	1100	20
7934	MILLER CLERK	7782	23-JAN-82	1300	10

The example displays records of all the Employees where the deptno is either 10 or 20.

NOT Operator

NOT operator returns TRUE if the enclosed condition evaluates to FALSE and FALSE if the condition evaluates to TRUE.

Example

```
SELECT * FROM Student WHERE NOT (sname is NULL);
```

The example displays all the rows where the student name is NOT NULL.

SET operators

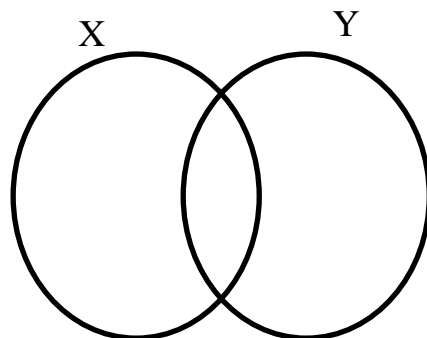
Set operators combine the results of two component queries into a single result. Queries containing set operators are called Compound Queries.

NOTE: The SET operators can compare two sets of data only if the data are of the same datatype.

The SET operators are discussed below.

Union

Union operator is used to display all the records selected by either query. Consider two tables X and Y with the same structure. A UNION operation performed on these two tables yields records from both the tables without the values being repeated.



Data in the tables

Table X Table Y

Item code	Name	Item code	Name
1	Poivders	2	Soaps
2	Soaps	3	Creamt
4	Pens	4	Pencils

SELECT* FROM X

UNION

SELECT * FROM Y

The output of this query looks like:

ITEMCODE	NAME
1	Powers
2	Soaps
3	Creams
4	Pencils
4	pens

UNION ALL OPERATOR

UNION ALL operator works the same way as the UNION Operator . But the difference is that this operator displays duplicate values also.

Example

SELECT * FROM X

UNION ALL

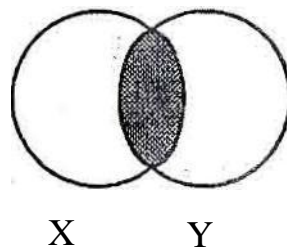
SELECT * FROM Y

The output of this query looks like:

ITEMCODE	NAME
1	Powders
2	Soaps
2	Soaps
3	creams
4	Pencils
4	Pens

INTERSECT operator

All the common distinct values are alone selected by the INTERSECT operator



Example

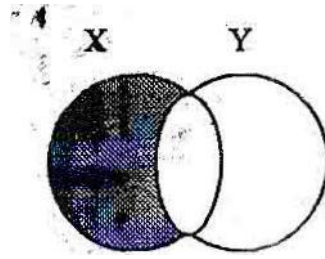
```
SELECT * FROM X
INTERSECT
SELECT * FROM Y
```

This would return

ITEMCO	NAME
3	Soaps

MINUS Operator

All the distinct rows selected by the first query but which are not in the second query are listed



Example

```
SELECT * FROM x
MINUS
SELECT * FROM y
```

Records available in table X that are not available in table Y are displayed.

The output will look like

ITEMCODE	NAME
1	Powders
4	pens

All set operators have equal precedence. If a SQL statement contains multiple sets of operators, ORACLE evaluates them from left to right. If the order is to be changed, parentheses

has to be used to explicitly specify another order. These querying tables can contain different structures but the type of the data in the tables must match,

Functions

Functions are similar to an operator. A function manipulates data items and returns a result. Functions differ from operators in the format in which they appear with arguments. Functions can be broadly classified into two types.

- Built - In Functions
- User — Defined Functions (dealt in later chapters) Built — in functions are predefined functions that perform a specific task. Built — in functions based on the values that they take to perform a task, can be classified into two type's.
- Scalar or Single — Row Functions

➤ Aggregate or Group Functions

A single row function returns a single result row for every row-of a queried table or view, while a group or aggregate. function works on a group of rows.. This-section deals briefly with the various types of Single -, Row functions.

SCALAR FUNCTIONS

Scalar Functions can be classified as follows:

- Number Functions
- Character Functions
- Returning Number Values
- Returning Character Values
- Date Functions
- Conversion Functions
- Other Functions

Number Functions

Number functions accept numeric input return numeric values. This section deals with numeric functions.

ABS

Syntax

ABS (n)

Returns the absolute value of n

Example

```
SELECT ABS (-15) " Absolute " FROM DUAL
```

Absolute

15

FLOOR Syntax

Floor (n)

Returns the largest integer equal to or less than n.

Example

SELECT FLOOR (15.7) " FLOOR" FROM DUAL

Floor

15

CEIL Syntax

CEIL (n)

Returns the smallest integer greater than or equal to n

Example

SELECT CEIL (15.7) " Ceiling " FROM DUAL

Ceiling

16

EXP Syntax

Exp (n)

Returns e raised to the nth power ; e = 2.71828183....

Example

SELECT EXP (4) " e to the 4th power " FROM DUAL

e to the 4th power

54.59815

LN Syntax

LN(n)

Returns the natural logarithm of n, where n is greater than 0

Example

SELECT LN(95) "Natural log of 95" FROM DUAL

Natural log of 95

4.5538769

LOG Syntax

LOG (m,n)

Returns the logarithm , base m, of n. The base m can be any positive number other than 0 or 1 and n can be any positive number.

Example

```
SELECT LOG (10,100) " Log base 10 of 100" FROM DUAL
```

Log base 10 of 100

2

MOD Syntax

MOD (m,u)

Returns the remainder of m divided by n. Returns m if n is 0. Example

```
SELECT MOD (11,4) " Modulus " FROM DUAL
```

Modulus

3

POWER Syntax

POWER (m,n)

Returns m raised to the nth power . The base m and-the exponent n can be any numbers, but if m is negative , n must be an integer

Example

```
SELECT POWER (3,2) " Raised " FROM DUAL
```

Raised

9

ROUND Syntax

ROUND (n{ ,m})

Returns n rounded to m places right of the decimal point; if m is omitted, n is rounded to 0 places. m can be negative to round off digits left of the decimal point. m must be an integer.

Example

```
SELECT ROUND (15.193,1) " Round " FROM DUAL
```

Round

15.2

SIGN Syntax

SIGN (n)

If $n < 0$, the function returns -1; if $n = 0$, the function returns 0, if $n > 0$ the function returns 1.

Example

```
SELECT SIGN (-15) " sign " FROM DUAL
```

Sign

-1

SQRT Syntax

SQRT (n)

Returns square root of n. The value of n cannot be negative. SQRT returns a " real " result

Example

```
SELECT SQRT (26) " Square root " FROM DUAL
```

Square root

5.0990195

TRUNC Syntax.

TRUNC (n{ ,m})

Returns n truncated to m - 1 places; if n is omitted, n is truncated to 0 places. m can be negative to delete (make zero) m digits left of the decimal point

Example

```
SELECT TRUNC (15.79,1) " Truncate " FROM DUAL
```

Truncate

15.7,

```
SELECT TRUNC (15,79,-1) " Truncatee" FROM DUAL
```

Truncate

10

Character Functions

Character functions can return both numeric and character values. The following functions are character functions that return numeric values.

ASCII

Syntax

ASCII(char)

Returns the decimal representation in the database character set of the first byte of char. If the database character set is 7-bit ASCII, the function returns an

ASCII Value

Example

```
SELECT ASCII('Q') FROM DUAL
```

ASCII ('Q')

81

INSTR

Syntax

`INSTR (char 1, char2, [n[,m]])`

Searches char 1 beginning with its nth character for rth occurrence of char 2 and returns the position of the character in char 1 that is the first character of this occurrence. If n is negative, ORACLE counts and searches backward from the end of char 1. The value of m must be positive. The default values of both n and m are 1, meaning ORACLE begins searching at the first character of char 1 for the first occurrence of char2, The return value is relative to the beginning of char 1, regardless of the value of n, and is expressed in characters. If the search is unsuccessful (if char 2 does not appear m times after the nth character of char 1) the return value is 0.

Examples

```
SELECT INSTR('CORPORATE FLOOR', 'OR', 3, 2 " Instring " FROM  
DUAL  
Instring
```

14

```
SELECT INSTR ( 'CORPORATE FLOOR', 'OR', -3, 2 "Reversed Instring" FROM  
DUAL
```

```
Reversed Instring
```

INSTRB

Syntax

`INSTRB (Char 1 , Char 2 [,n [,m]])`

This is the same as INSTR , except that n and the return value are expressed in bytes rather than in characters. For a single — byte database character set, INSTRB is equivalent is

Example

```
SELECT INSTRB ('CORPORATE FLOOR ', ' OR', 5,2) " Instring in bytes"  
FROM DUAL
```

Instring in bytes

27

LENGTH

Syntax

LENGTH (Char).

Returns the length of characters in char . If Char has data type CHAR ,the length includes all trailing blanks . If char is null , this function returns null.

Example

```
SELECT LENGTH ( ' RADIANT ) " Length in characters" 'FROM DUAL  
Length in characters
```

7

LENGTHB

Syntax

LENGTHB (char)

Return the length at char in bytes . If char is null, this function returns null,
For a single byte database character set, LENGTHB is equivalent to LENGTH

Example

Assume a double - byte database character set:

```
SELECT LENGTHB ( ' CANDIDE ) " Length in bytes " FROM DUAL  
Length in bytes
```

14

Character Functions Returning character values.

CHR

Syntax

CHR(n)

Returns the character which is the binary equivalent of n in the database character set.

Example

```
SELECT CHR(75) "character" FROM DUAL
```

Character

K

CONCAT

Syntax

CONCAT (char 1, char 2)

Returns char1 concatenated with char 2. This Function is equivalent to the concatenation (||)

Example

This example uses nesting to concatenate three character strings ;

```
SELECT CONCATE ( CONCAT ( ename , ' is a ' ) , job ) " job " FROM  
emp WHERE empno = 7900
```

Job

JAMES is a CLERK

INIT CAP

Syntax

INITCAP(char)

Returns char, with the first letter of each word in uppercase and all other letters in lowercase. Words are delimited by white spaces or characters that are not alphanumeric.

Example

```
SELECT INITCAP('the soap') Capitalized " FROM DUAL
```

Capitalized

The Soap

LPAD

Syntax

```
LPAD( char 1, n [,char 2])
```

Returns char 1, Left- padded to length n with the sequence of characters in char,2; char 2 defaults to " , a single blank . If char 1 is longer than n, this function returns the portion of char 1 that fits in n.

The argument n is the total length or the return value as it is displayed on the screen. In most character sets, this is also the number of characters in the return value. However, in C52111 multi- byte character sets, the display length of a character string can differ from the number of characters in the string.

Example

```
SELECT LPAD (' Page 1', 15, * .') LPAD example " FROM DUAL
```

LPAD example

***** Page 1

LTRIM

Syntax

```
LTRIM ( char [ , Set] )
```

Removes characters that appear in set from the left of char , set defaults to " , a single blank

Example

```
SELECT LTRIM ( ' xyxXxy LAST WORD ', 'xy ' ) " Left trim example "  
FROM DUAL
```

Left trim example

X xy LAST WORD

REPLACE

Syntax

```
REPLACE ( char , search_string [ , replacement_string ] )
```

Returns char with every occurrence of search_string replaced with replacement_string. If search replacement_string is omitted or null, all occurrences of search_string are removed . search string is null, char is returned. TES function provides a superset of the functionality provided by the TRANSLATE function. TRANSLATE provides single character , one to of): substitution . REPLACE allows you to substitute one string for another as well as to real character strings.

Example

```
SELECT REPLACE ( ' JACK and JUE', 'J', 'BL' ) " Changes " FROM DUAL  
Changes
```

BLACK and BLUE

RPAD

Syntax

```
RPAD ( char1, n [ , char 27])
```

Returns char1, right — padded to- length n with char2, replicated as many times as necessary. The default padding is “”, a single blank . If char 1 is longer than n, function returns the portion of char 1 that fits in n.

The argument is the total length of the return value as it is displayed on the terminal screen. In most character sets, this is also the number of characters in the

return value. However, in certain multi-byte character sets, the display length of a character string can differ from the number of characters in the string.

Example

```
SELECT RPAD ( ename , 11, ' ab ' ) " RPAD example" FROM emp
WHERE ename = ' TURNER '
```

```
RPAD example
```

```
TURNER ababa
```

RTRIM

Syntax

RTRIM (char [, set])

Returns char, with all the right-most characters that appear in set removed; set defaults to ' ', single blank . RTRIM works similar to LTRIM

Example

```
SELECT RTRIM ( ' TURNER yxXx',' xy' ) " Right trim example" FROM
DUAL
```

```
Right trim example
```

```
TURNER yxX
```

SOUNDEX

Syntax

SOUNDEX (char)

Returns a character string containing the phonetic representation of char. This function allows words that are spelled differently, but sound alike in English to be compared.

Example

```
SELECT ename FROM emp WHERE SOUNDEX (ename=)
SOUNDEX('SMYTHE')
```

```
ENAME
```

SMITH

SUBSTR

Syntax

SUBSTR (char,m [,n])

Returns a Portion of char , beginning at character m,n characters long . If m is positive , ORACLE counts from the beginning of char to find the first character. If m is negative ORACLE counts backwards from the end of char. The value of m cannot be 0. If n is omitted, ORACLE returns all characters till the end of char. The value of n cannot be less than 1.

Example

```
SELECT SUBSTR (' ABCDEFG ' , 3,2) " substring " FROM DUAL
```

substring

CD

```
SELECT SUBSTR (' ABCDEFG ',-3,2 ) " substring " FROM DUAL.
```

substring

EF

SUBSTRB

Syntax

SUBSTRB (char,m[,n])

The same as SUBSTR , except that the arguments m and n expressed in bytes, rather than in characters. Fiat a Single - byte database character set, SUBSTRB ,equivalent to SUBSTR

Example

Assume a double - byte database character set:

```
SELECT SUBSTRB ('ABCDEFGF' , 5,4) " Substring with bytes" FROM DUAL
```

Substring with bytes

CD

TRANSLATE

Syntax

TRANSLATE (char, from , to)

Returns char with all occurrences of from character replaced by its corresponding to character . Characters in char that are not in from character are not replaced. The argument from can contain more character is than to. In this case, the extra characters at the end of from have no corresponding characters in to If these extra characters appear in char, they are removed from the return value. We cannot use empty string for to in order to remove all characters in from the return value . ORACLE interprets the empty string as null, and if this function has null argument , it returns null.

Example

This statement translates the given word called ' Miles' to ' Tiles'

```
SELECT TRANSLATE('Miles','M','T')TRANSLATE example"
```

```
FROM DUAL
```

```
Translate example
```

Tiles

This statement returns a license number with the characters ,removed and the digits remaining.

```
SELECT TRANSLATE ('2 KRW 229', '0123456789  
ABCDEFGHIJKLMNOPQRSTUVWXYZ ','0123456789')
```

```
"Translate example " FROM DUAL
```

Translate example

2229

UPPER

Syntax

UPPER (char)

Returns char, with all letters uppercase . The return value has the same datatype as the argument char

Example

```
SELECT UPPER (' Large') " Uppercase " FROM DUAL  
uppcase
```

LARGE

LOWER

Syntax

LOWER (char)

Returns char, with all letters in lowercase. The return value has the same datatype as the argument char (CHAR or VARCHAR 2)

Example

```
SELECT LOWER ('MR . SAMUEL HILLHOUSE ') "Lowercase "  
FROM DUAL
```

Lowercase

mr. samuel hillhouse

Date Functions

Date functions operate on values of the Date datatype . All Date functions return a value of a DATE data type, except the MONTHS_BETWEEN function that returns a number.

ADD_MONTHS

Syntax

ADD_MONTHS (d,n)

Return the date d plus n months, The argument n can be any integer. If d is the last of the month or if the resulting month has fewer days than the day component of d, thenif result is the last day of the resulting month. Otherwise, the result has the same day component as d.

Example

```
SELECT ADD_MONTHS (sysdate, 1 ) "Next month " FROM DUAL ; Next  
month
```

04- JAN-01

LASTDAY

Syntax

LAST -DAY (d)

Returns the date of the last day of the month that contains d. This function is determine how many days are left in the current month.

Example

```
SELECT SYSDATE, LAST_DAY(SYSDATE ) " Last " , LAST DAY  
(SYSDATE)- SYSDATE." Days Left " FROM DUAL
```

SYSDATE Last Days Left

04-DEC-00 31 – DEC-00 27

```
'SELECT ADD_MONTHS ( LAST DAY ( sysdate ) , 5 ) " Five months "  
FROM dual ;
```

Five month

31- MAY -01

MONTHS_BETWEEN

Syntax

MONTHS_BETWEEN (d1,d2)

Returns number of months between dates d1 and d2 . If d1 is later than d2, the result positive; if d 1 is earlier than d2, the result is negative. If d1 and d2 are either the same days o the month or both last days of Months, the result is always an integer, otherwise ORACLIT calculates the fractional portion of the result based on a 31 - day month and also considers the difference in time components of d1 and d2.

EXAMPLE

```
SELECT MONTHS_BETWEEN (sysdate , ' 10 -JAN-00') "Months"  
FROM DUAL;
```

Months

10.829904

NEXT_DAY

Syntax

NEXT_DAY (d,char)

Returns the date of the first weekday named by char that is later than the date d. The argument char must be a day of the week in the session's date language. The return value has the same hours, minutes,and seconds component as the argument d.

Example

```
SEIECT NEXT_DAY('06-DEC-00', TUESDAY) "NEXTDAY"
```



```
FROM DUAL;  
NEXT DAY
```

```
12-DEC-00
```

ROUND

Syntax

```
ROUND(d [fmt])
```

Returns d rounded to the unit specified by the format model fmt. If fmt is omitted d is rounded to the nearest day.

Example

```
SELECT ROUND(SYSDATE,'YEAR') "FIRST OF THE YEAR"  
FROM DUAL  
FIRST OF  


---

  
01-JAN-01
```

TRUNC

Syntax

```
TRUNC (d, [fmt])
```

Returns d with the time portion of the day truncated to the unit specified by the format model fmt. If we omit fmt, d is truncated to the nearest day.

Example

```
SELECT TRUNC (SYSDATE, 'MM') "First of the Month"  
FROM DUAL  
First of  


---

  
01-DEC-00
```

Conversion Functions

Conversion Function converts a value from one datatype to another. Generally, the form of the function names follows the convention datatype TO datatype. The first datatype is the input datatype; the latter datatype is the output datatype. This section lists the SQL conversion functions.

TO_CHAR, date conversion

Syntax

TO_CHAR(d[,fmt])

Converts d of DATE datatype- to value of VARCHAR2 datatype in the form specified by the date format fmt. If fmt is not specified, d is converted to a VARCHAR2 value in the default date format:

Example

```
SELECT TO_CHAR(SYSDATE,'Month DD, YYYY') "New date format"
FROM dual;
New date format
_____
December 04,2000
```

TO_CHAR, number conversion

Syntax

TO_CHAR(n[,fmt])

Converts n, or NUMBER datatype a Value of VARCHAR2 datatype, using the optional number format fmt. If no fmt is provided n is converted to a VARCHAR2 value exactly long enough to hold its significant digits.

Example

```
SELECT TO_CHAR(17145, '99G999') "Char" FROM DUAL Char
_____
017,145
```

TO_DATE

Syntax

TO_DATE(Char[,fmt])

Converts char of CHAR or VARCHAR2 datatype to a value of DATE datatype. The fmt is a date format specifying the format of char. If the fmt is not specified, char, must-be in the default date format. If fmt is 'J' for Julian, then char must be a number.

Do not use the TO DATE function With a DATE value for the char argument. The returned DATE value can have a different Century value than the original char, depending-on fat or the default date format.

Example

```
SELECT TO_DATE ('September 25,2000,11:00 A.M' 'Month dd, YYYY, HH:MI  
A.M.')
```

 FROM DUAL

Output

```
TO_DATE('
```

25-SEP-00

The following Table is lists of all the functions:

FUNCTIONS	NAME
Numeric functions	ABS(), EXP(), FLOOR(), CEIL(),LN(), LOG(),MOD(), POWER(), ROUND(), SIGN(), SQRT(), TRUNC()
Character functions Returning numeric values	ASCII(), INSTR(), INSTRB(), LENGTH()
Character functions returning character values	CHR(), CONCATO, INITCAP(), LPAD(),LTRIM()REPLACE(), RPAD(), RTRIM(), SOUNDEX(), SUBSTR(), SUBSTRB(), TRANSLATE(), UPPER(), LOWER(),

Date functions	ADD_MONTH(),LAST DAY(),MONTHS_BETWEEN(), NEXT_DAY(),ROUND(), TRUNC(),
Date functions	ADD_MONTHS(),LAST_DAY(),MONTHS_BETWEEN(), NEXT DAY(),ROUND(), TRUNC(),
Conversion functions	TO CHAR (), TO_DATE(), TO NUMBER(),
Other functions	DUMP (), UID(), GREATEST(), LEAST(), NVL(), UID, USER, SYSDATE

Aggregate Functions

Aggregate Functions are used to perform queries based on groups of rows rather than on a single row. Also, group functions allow users to select summary information's from groups of rows. Following are the list of group functions.

Function name	Description
AVG()	Computes the average of values
MAX()	Finds the maximum of all values
MIN()	Calculates the minimum value
COUNT()	Counts the total number of values/records
SUM()	Calculates the sum or total of all values

CREATION OF. SYNONYMS

Database objects

Database Objects are the schema objects. The various data base objects are listed below:

- Tables
- Views
- Synonyms
- Sequences
- Indexes
- Cluster

Creating a simple view

Views are created using the Create command. The definition of the view will contain the column names given in the query. Syntax for Creating a view.

Syntax:

```
CREATE OR REPLACE VIEW <viewname> AS< query>]
```

Example:

This creates a simple view with records selected from the Employee Table.

```
CREATE OR REPLACE VIEW empview AS SELECT ename, depno, sal FROM emp;
```

This displays the feed back.

View created.

Dropping Views

Dropping Views can be done using the statement;

```
DROPVIEW <viewname>
```

Synonyms

A synonym is a database object, which is used as an alias name for any object.

The main advantages of using Synonyms are:

Simplify SQL statements

Hide the real identity of an object

Use in database link

Synonyms can be public or private. Synonyms created as a Public Synonyms are accessible to all users, The public synonyms are owned by the user group PUBLIC and can be dropped only by a DBA.

Syntax for creating a synonym name:

Syntax:

```
CREATE SYNONYM <synonymname>FOR<objectname>;
```

Example:

```
CREATE SYNONYM synemp FOR EMP
```

SELECT * FROM-synemp will display all the employee records in the emp table.

Dropping Synonyms

Dropping of Synonyms can be done Using theDropcommand:

```
DROP SYNONYM synemp;
```

Sequences

Sequences are a set of database objects which can generate unique or Sequential integer value. They are used for automatically generating primary key or .unique key values. A sequence can be created in ascending or descending order.

Creating Sequences

CREATE SEQUENCE command is used to create sequence. The syntax for creating Sequences are:

Syntax:

```
CREATE SEQUENCE <SEQUENCE NAME>  
[<INCREMENT BY><VALUE>  
<START WITH><VALUE>  
<MIN VALUE><VALUE>  
<MAX VALUE.><VALUE>  
<CACITE><V AWE>  
<CYCLE><VALUE>
```

where,

INCREMENT BY : specifies the interval between 2 integers. Can be a positive or negative value but not zero.

STARTWITH : Specifies the first sequence number to be generated.

MINVALUE : Indicates the minimum Value in the sequence. It must be less than or equal to STARTWITH and less than MAXVALUE.

MAXVALUE : Specifies the maximum value the sequence can generate.

CYCLE: Indicates that sequence continues to generate values after reaching maximum values .

CACHE : Specifies how many values must be kept in memory for faster access.

For Example

```
CREATE SEQUENCE s1
START WITH
1 MINVALUE 1
INCREMENT BY 1
MAX VALUE 20
CYCLE
CACHE 15;
```

Displays,

Sequence created Index

When an index is present and will help the performance of an application request , Oracle automatically uses the index; otherwise; Oracle ignores the index

Oracle automatically updates an index to keep it in synch with its table.

. Although indexes can dramatically improve the performance of application request Unwise to index every column in a table. Indexes are meaningful only for the key columns application requests specifically use to find rows of interest. Furthermore, index maintenance generates overhead-unnecessary indexes can actually slow down your system rather than improve its performance.

Oracle 8 supports several different types of indexes to satisfy many types of application requirements. The following sections explain more about the various types of indexes that can be created for a table's columns:

When an index is created, Oracle fetches and sorts the columns to be indexed and stores the ROWID along with index value for row. Oracle loads the index from

the bottom They are logically and physically independent of the data associated with the tables.

How Indexes are

When you Create an index, Oracle automatically allocates an index segment to hold index's data in a table space

The tablespace of an index's segment is either the owner's default tablespace or a tablespace specifically named in the CREATE INDEX statement. You do not have to place index in the same table space as its associated table.

For a unique - index, there is ROWID per data value. For a non-Unique index; ROWID is included, in the key in sorted order, the index key and ROWID sort so non-unique indexes. Key values containing nulls are not indexed, except for cluster indexes. Two rows can both contain all nulls and not violate a unique index.

Unique indexes

Unique indexes are indexes that are defined on columns, which ensure that no two can hold the same values... In other words, it avoids duplication of values.

For example, Primary key and unique constraints are by default create unique index.

Composite Indexes

Composite Indexes are indexes that get created, on more than one column based on the leading column and, then on the subsequent column given inside the index. They are also called as Concatenation Index.

Creating Indexes:

Indexes can be created in the following ways:

Create index <indexname> ON <tablename> (<columnname>)

Example:

```
CREATE INDEX i1 ON emp (JOB)
```

Dropping indexes

Index can be dropped by using Drop command

Syntax:

Drop

index<indexname>Example:

```
DROP INDEX 11
```

LESSON-5

Introduction to PL/SQL

Oracle's PL/SQL is a procedural language extension of SQL. It is the standard programming language for Oracle RDBMS and follows the procedural approach. It also provides conditional constructs like IF... THEN..., WHILE LOOP Which are available in other languages like C, Pascal, COBOL, PL/SQL is very powerful as it combines the flexibility available in SQL with the procedural constructs. Oracle's PL/SQL adds a lot of additional capabilities to Oracle programming in order to do validations, customization, include user interlaces and handling errors effectively.

What is PL/SQL

PL/SQL is a block-structured language. The basic unit of PL/SQL is called a BLOCK, which contains declarative statements, executable statements and error handling statements. These blocks can be nested into one or more blocks.

Features of PL/SQL

- ❖ Allows us to embed one or more SQL statements together and execute as a single SQL unit.
- ❖ Allows declaration of Variables.
- ❖ Allows usage of conditions! co acts
- ❖ Allows programm4 of error-handling using exceptions.
- ❖ Allows row-by--row processing of data using cursors
- ❖ Allows triggers to created arid fired

Advantages of PL/SQL

SQL Support

SQL has become the standard database language because it is flexible, powerful and easy to learn. Since it is a non-procedural language, a user need not know how the statements are processed but just needs to indicate the requirement.

allows the usage of all kinds of SQL data manipulation, cursor control and transaction control statements as well as the SQL functions, operators and pseudo columns.

Better performance

Without PL/SQL Oracle would process SQL statements one at a time. Each SQL statement results in another Oracle call to Oracle and higher performance overhead. Since PL/SQL can contain SQL statements, all the SQL statements can be placed inside PL/SQL thereby reducing the time taken for communication between the server and the application. This reduces network traffic and results in better performance.

Support for Object Oriented Programming

Oracle implements the concept of object oriented programming. This allows the creation of software components that are modular, maintainable and reusable. Using encapsulation operations with the data object types lets users to move data-maintenance codes out of scripts and PL/SQL blocks into methods.

Portability

Applications written in PL/SQL provides portability to the operating system and platform on which they run. These applications can run wherever Oracle can run.

Higher Productivity

PL/SQL adds functionality to Oracle's non procedural tools like Forms and Reports. These tools can help to build applications using familiar procedural constructs. PL/SQL does not differ in environments. It works the same way as it works for other built-in tools. Also, scripts written using one tool can be used by another tool.

Integration with Oracle

Oracle and PL/SQL are based on SQL statements. It supports all the SQL datatypes. These datatypes integrate PL/SQL with the Oracle Data dictionary.

PL/SQL Architecture

Before dismissing the architecture of PL/SQL block the basic unit of a PLSQL which is a block needs to be understood. A PLSQL block contains three parts.

1. Declarative part
2. Executable part
3. Error handling part.

Declarative part

This is the first section of PL/SQL block. This section is used to declare variables, constants, Every declaration or definition has a memory allocated in the session's memory. The keyword DECLARE represents this part.

Syntax

DECLARE

Set of variables.

Executable part

This part contains all the SQL and PUSQL statements, which are used for querying processing the data. BEGIN is used to mark the beginning of the block and END endblock. There must be at least one executable statement within a set of BEGIN and END statements.

Error handling part

This part is called Exception. This part contains the error handling statements. Oracle takes the control from the execution part of this part whenever any error is raised in the program and checks for the exception.

Architecture of PL/SQL

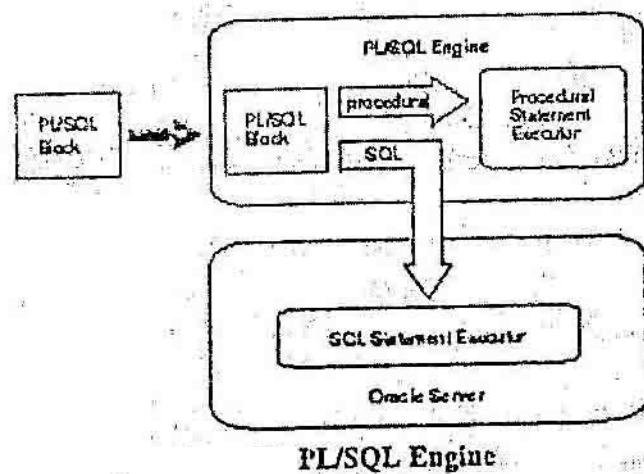
PL/SQL runtime system is a technology, which has an engine that executes PL/SQL blocks and subprograms. The engine can be installed in the Oracle server or

in an application development tool such as Forms and Reports. The engine can reside in either of the two environment

Oracle server

Oracle tools

These two environments are independent. PL/SQL might be available in Oracle Server t unavailable in tools or the other way around The PL/SQL engine processes the procedural Statements and sends the SQL statements to the SQL statement Executor in the Oracle Server the following figure illustrates this:



PL/SQL Engine in the Oracle Server

Applications development tools that lack a local PL/SQL engine must rely on oracle to process PL/SQL blocks and subprograms. When it contains the PL/SQL engine, an oracle server can process PL/SQL blocks and subprograms as well as single SQL statements. The Oracle server passes the blocks and subprograms to its local PL/SQL engine. Basically there are 4 types of block

Anonymous Blocks

Named blocks

Stored Subprograms

Triggers.

Anonymous Blocks

Anonymous PL/SQL blocks can be embedded in an Oracle precompiler or OCI program. At run time, the program, lacking a local PL/SQL engine, sends these blocks to the Oracle Server, where they are compiled and executed. Like wise, interactive tools such as SQL* plus and Enterprise Manager lack a local PL/SQL engine, must send anonymous blocks to Oracle.

Named Blocks

Named blocks act in the same way as anonymous blocks except that they can have names in the form of labels, which are valid only for that program.

Stored Sub programs

Subprograms can be compiled separately and stored permanently in an Oracle database names in the form of labels, which are valid only for that program.

Stored Subprograms

Subprograms can be compiled separately and stored permanently in an Oracle database ready to be executed. A subprogram explicitly created using an Oracle tool is called a stored subprogram. Once compiled and stored in the data dictionary, it is a scheme object, which can be referenced by any number of applications connected to that database.

Subprograms defined within another subprogram or within a PL/SQL block are called subprograms. They cannot be referenced by other applications and exist only for the convenient the enclosing block.

Stored subprograms offer higher productivity, better performance, memory savings application integrity, and tighter security. For example, by designing applications around a library stored procedures and functions you can avoid redundant coding and increase productivity.

Each stored subprogram has a name and is called or manipulated using the name in an PL/SQL structure and the associated rules are the same when a

subprogram is constructed using them. Procedures and functions are known as subprograms. They can be called inside triggers also

Oracle has a concept by which all the related subprograms along with the variables and an can be packaged for defining and deploying an application. These are called packages.

Database Triggers

A database trigger is a stored subprogram associated with a table, you can have On automatically fire the database trigger before or after an INSERT, UPDATE or DELETE statement affects the table. One of the many uses of database triggers is to audit data modifications.

A database trigger has a typical PL/SQL structure and can call subprograms.

PL/SQL engine in Oracle Wools

When it contains the PL/SQL engine, an application development tool can process blocks. The tool passes the blocks to its local PL/SQL engine. The engine executesprocedural statements at the application site and sends only SQL statements to Oracle. Thus, most of the work is done at the application site, not at the server site.

Furthermore, if the block contains no SQL statements,. The engine executes the entire block at the application site. This is useful if the application cart benefit from conditional and iterative control.

Frequently, Oracle Forms applications use SQL statements merely to test the value of field entries or to do simple computations. By using PL/SQL instead, calls to the Oracle Server can be avoided. Moreover, PL/SQL functions can be used to manipulate field entries.

PL/SQL Delimiters

Before writing a simple PL/SQL block first we need to know the basic delimiters

Delimiter	Name	Description
;	Statement Terminator	Each statement terminates
:=	Assignment Operator	Used to assign values to (A:=10) variables
<<ZZ>>	Label Indicator	Used to defined the labels inside a block
_____	Single Line	Commenting a single line comment
/*..*/	Multiline comment	Many statements can be placed inside this and they
..	Range indicator	This is used in FOR Loops which indicate the minimum range and
&	Substitution variable	Used to substitute values to variables.
:	Bind or Session Variable accessed	Global variable can be
%	Attribute Indicator	They are used to assign the datatype of a column to a variable.
	DBMS_OUTPUT .PUT LINE	Used to print messages on the screen

Data types

Every constant and variable has a datatype that specifies a storage format and valid range of values. A part from the datatypes that are available in SQL, PL/SQL has its own set of datatypes that can be used in PL/SQL blocks.

NUMBER Datatypes

Numeric Datatypes are used to store numeric data and represent quantities, calculations can be performed on this datatypes.

Binary Integer

Binary `_integer` datatype is used to store signed integers. Its magnitude ranges from `-2147483647...2147483647`. They are used to store array type or data. This type requires storage compared to number values.

Subtypes:

A base type is the datatype from which a datatype is derived. A subtype associates a base type with a constraint and so defines a subset of values. A following are the list of sub types available.

- `NATURAL`.
- `NATURALN`
- `POSITIVE`
- `POSITIVEN`
- `SIGNTYPE`

Of these `NATURAL` and `POSITIVE` datatypes hold all positive numeric values. To prevent `NULLS` from being entered, `NATURALN` and `POSITIVEN` are used. The `SIGNTYPE` restricts an integer variables to the value `1,-1` or `0` depending on the type or value entered.

Collection Datatypes

A collection is in ordered group of elements, all of the same type. Each element has a unique subscript that determines the position of the collection. We shall see more about Collections in the later chapters.

Boolean Datatypes

Logical values `TRUE`, `FALSE`, or `NULL` can be stored using the `BOOLEAN` datatype. The datatype has no parameters.

Exception Datatype

This is a datatype that is used to define exception handlers, which is defined by the user. This is dealt in the chapter Exceptions.

Writing a simple program

The following example is used to display a message called 'WELCOME TO THE WORLD OF PL/SQL'.

```
BEGIN
  DBMS_OUTPUT.PUT_LINE("WELCOME TO THE WORLD OF
PL/SQL");
END;
```

Note that the program has no DECLARE part since the program does not use any variable declaration. Declarative section is required only when variables are to be used like the following example:

Example

```
DECLARE
  x NUMBER;
BEGIN
  x:=56;
  DBMS_OUTPUT.PUT_LINE (X);
END;
```

The above example, displays the value 56 which is stored in the variable x. In order to accept the value at run-time the assignment operator has to be used.

Note: Although DBMS_OUTPUT.PUT_LINE is used to display messages, the messages will not be displayed unless the environment setting called.SET SERVEROUTPUT is turned on. This is a SQL Plus statement .

Example

```
DECLARE
  x NUMBER;
BEGIN
  x :=&numb;
  DBMS_OUTPUT.PUT_LINE (X);
END;
```

The assignment operator (&) is used to accept the value in the numb variable and this variable assigned to x. It is similar to the continuous insert Statements ,

Using the SELECT statement inside PL/SQL

As discussed earlier, PL/SQL allows embedding of one or more SQL statements inside the block. To display a condition based record, the select statement requires an 'INTO' clause. A SELECT statement must hold an INTO clause where the- output of the query is assigned to the variables given in the INTO clause. The following example illustrates this

Example

```
DECLARE
    s NUMBER;
BEGIN
    SELECT sal INTO s FROM emp WHERE empno=7369;
    DBMS_OUTPUT.PUT_LINE ('the salary is '||s);
END;
```

The variables is assigned the salary of the employee' number 7369 and the result would be displayed.

Declaring Variables

Variables can be declared in various ways. The two examples given above have shown how to declare variables and assign values to them. Certain variables can carry default Values which can be used to initialize values and certain other variables can take constant values. Tb following example illustrates the usage of the keywords DEFAULT and CONSTANT.

```
Using DEFAULT DECLARE
S NUMBER DEFAULT 10;
BEGIN
    DBMS_OUTPUT.PUTLINE (S);
END
```

In this example, the default value of S i.e. 10 is printed. The value in the variable s ,can b altered. It can also be re-assigned with any other numerical value.

Using CONSTANT

Consider a situation where variables have to contain constant values; like for example p1 has to be assigned 3.14

Example

```
DECLARE
```

```
    Pi CONSTANT REAL :=3.14;
```

```
    Area NUMBER ;
```

```
    R NUMBER:=&R;
```

```
BEGIN
```

```
    Area :=pi*r**2;
```

```
    DBMS_OUPUT_LINE (' THE AREA OF CIRCLE WITH RADIUS.  
'||r||'IS'||area);
```

in the above example, me variable pi is declared as, a constant variable whose value cannot change anywhere inside me program. Reassigning me value for the same variable leads to an error.

Using NOT NULL

Besides assigning an initial value, declarations can impose the NOT NULL constraint, as the following example shows;

Example

```
DECLARE.
```

```
    s VARCHAR2(10) NOT NULL :='RADIANT'
```

```
BEGIN
```

```
    DBMS_OUTPUT.PUT_LINE (s);
```

```
END
```

The-difference between using NOT NULL and DEFAULT is that default . can be assigned to NULL but NOT NULL as the name suggests cannot hold NULL values.

Control Structures

Often it is necessary to take alternative actions depending on circumstances. The IF statement allows the execution of a sequence of statements conditionally. The execution purely depends on the value of condition specified. The working is similar to the IF conditions in other programming languages. There are three forms of IF statements.

```
IF ....THEN
IF....THEN....ELSE
IF....THEN....ELSE IF
IF....THEN
```

This is the simplest form of the IF statement . This associates a condition with a sequence of statements enclosed by the key words THEN and END IF . The following example illustrates. this;

Syntax

```
IF condition THEN
Sequence_of_statements
End if;
```

The sequence of statements is executed only if the condition yields TRUE. If the condition yields FALSE or NULL , the IF statement does nothing In either case, control passes to The next statement following END IF. An example follows.

Example

```
DECLARE
    s number;
BEGIN
    S:=&a,
    If s>=10 THEN
```

```

        DBMS_OUTPUT.PUT_LINE('s greater than or equal to 10');
    ENDIF
END;

```

The above example displays the value "S" 'greater than or equal to 10. only if the variable s holds the Value 10 or greater than that;

IF - THEN - ELSE

The second form of IF statement adds the key word ELSE followed by an alternative sequence of statements. The general syntax is as follows:

```

IF condition THEN
    Sequence_of_statements 1;
ELSE
    Sequence_of_statements2;
ENDIF;

```

The sequence of statements in the ELSE clause is executed only if the condition yields FALSE or NULLS. Thus the ELSE clause ensures that a sequence of statements is executed.

Modifying the above example;

Example

```

DECLARE
    S number;
BEGIN
    S:=&a;
    IF S >=10 THEN
        DBMS_OUTPUT.PUT_LINE('S GREATER THAN OR EQUAL 10')
    ELSE
        DBMS_OUTPUT.PUT_LINE ('S IS LESSER THAN 10')
    ENDIF ;
END;

```

IF-THEN-ELSEIF

The third form of IF Statement uses the key word ELSIF (not ELSIF) to introduce additional conditions. Multiple IF conditions are clubbed and written using the ELSIF clause.

Syntax:

```
IF CONDITION 1 THEN
    SEQUENCE OF STATEMENTS 1;
ELSIF CONDITION 2 THEN
    SEQUENCE _ OF _ STATEMENTS2;
ELSE
    SEQUENCE _ OF _ STATEMENTS3;
ENDIF;
```

If the first condition yields FALSE or NULL . the ELSIF clause tests another condition. An IF **statement** can have any number of ELSIF clause; the final ELSE clause is optional. Conditions are evaluated one by one from top to bottom. If any condition yields TRUE, its associated sequence of statements is executed and control passes to the next statement following ENDIF. If all the conditions yield FALSE or NULL, the sequence in the ELSE clause is executed. The following example finds the greatest or two numbers using IF-THEN-ELSIF

Example:

```
DECLARE
A NUMBER := &A
B NUMBER := &B
BEGIN
IF A > B THEN
DBMS_OUTPUT.PUT_LINE ('THE GREATEST NUMBER IS' || A);
ELSIF B > A THEN
```

```

DBMS_OUTPUT.PUT_LINE ( ' THE GREATEST NUMBER IS' || B);
ELSE
DBMS OUTPUT.PUT_LINE ( ' BOTH ARE EQUAL ');
ENDIF;
END;

```

The above example accepts two numbers and the conditions are checked and depending on the condition that evaluates to TRUE, the message under that condition is displayed .

When possible, use the ELSIF clause instead of nested IF Statements: This will make the code easier to read and understand. Compare the following the IF statements;

<pre> IF conditional 1 THEN .Statement 1 ELSE IF condition2 THEN Statement2; ELSE IF condition 3 THEN Statement 3; END IF; END IF; END IF; </pre>	<pre> IF. condition1 THEN statement 1; ELSIF condition2 THEN statement 2; ELSIF condition3 THEN: statement3; END IF; </pre>
---	---

These statements are logically equivalent. While the statement on the left obscures the flow of logic, the statement on the right reveals it.

Iterative control

Iterative statements are a set of statements that are performed a number of times depending on the value in the LOOP statement. There are three basic forms of LOOP statements.

- ❖ LOOP
- ❖ WHILE LOOP
- ❖ FOR LOOP

LOOP

The simplest form of LOOP statement is the basic (or infinite) loop, which encloses a sequence of statements between the keywords LOOP and ENDLOOP. The syntax is as follows.

Syntax

```
LOOP
    Sequence _ of _ statements,
End LOOP
```

With each Iteration of the loop, the sequence of the statements is executed and the control resumes at the top of the loop. The following example shows the execution of the LOOP statement.

Example 7.9

```
DECLARE
a NUMBER :=10;
BEGIN
LOOP
DBMS_OUTPUT.PUT_LINE (a);
END LOOP;
END.
```

The output leads to an error. Because the number of times the loop must be

executed is not specified and it does for an infinite loop. In order to terminate the loop some form of termination statement is required. The EXIT statement is used to perform the operation

Example

```
DECLARE
A NUMBER :=& A;
BEGIN
LOOP
DBMS_OUTPUT.PUT_LINE(A);
a :=a+1;
IF a >20 THEN
EXIT;
End if :
END LOOP;
END.
```

Till the value crosses 20, the loop is executed and once the value for the variable reach 20, the loop is terminated. using the EXIT statement. This EXIT statement can be further simplified by the usage of EXIT_WHEN statement.

EXIT WHEN

The EXIT_WHEN statement allows a loop to complete conditionally. When the EXIT statement is encountered, the condition in the WHEN clause is evaluated. If the condition yields TRUE, the loop completes and control passes to the next statement after the loop. The syntax follows;

```
BEGIN
EXIT WHEN <CONDITION>
END;
```

The following example shows the usage of EXIT WHEN statement. This example is a modification of the previous example with the EXIT statement

Example

```
DECLARE  
  
a NUMBER:=&A  
  
BEGIN  
  
LOOP  
  
DBMS_OUTPUT.PUT_LINE (a);  
  
a:=a+1;  
  
EXIT WHEN a>20;  
  
END LOOP;  
  
END;
```

WHILE LOOP

The while loop statement associates a condition with a sequence of Statements enclosed by the keywords LOOP and ENDLOOP. The syntax of the WHILE LOOP

Syntax

```
WHILE condition loop  
  
Sequence_of_ statements;  
  
END LOOP
```

Before each iteration of the loop, the condition is evaluated. If the condition yields TRUE, the sequence of statements is executed and the control at the top of the loop. If the condition yields FALSE or NULL, the loop is bypassed and control passes to the next statement after end loop: The following example displays the total of the first 20 numbers using the WHILE loop statement.

Example

```
DECLARE
X integer :=1;
Y integer :=0;
BEGIN
WHILE x<=20
LOOP
Y:=Y+x;
X:=X+1;
END LOOP;
DBMS_ OUTPUT.PUT_LINE (Y);
END;
```

In the above example, the loop is executed only when the while condition is satisfied.

FOR LOOP

While the number of iterations for a WHILE loop is unknown until the loop completes the number of iterations for a FOR loop is known before the loop is entered. FOR loops iterate over a specified range of integers. The range is part of an iteration scheme, which is enclosed within the keywords FOR and LOOP.

The syntax follows;

```
FOR counter IN[REVERSE] lower_bound..higher_bound LOOP
Sequence_of_statements;
END LOOP;
```

The range is evaluated when the FOR loop is first entered and is never re-evaluated the next example shows the sequence of statements is executed once for

each integer in the range. After the each iteration, the loop counter is incremented. The following example displays the reversal of a string.

Example:

```
DECLARE
    s VARCHAR2 (20):='&s';
    s1 VARCHAR2 (20);
BEGIN
    FOR I IN 1...LENGTH (S)
LOOP
    s1:=s1||SUBSTR(s,-I,1);
END LOOP
    DBMS_OUTPUT.PUT_LINE(s1);
END;
```

In the above example a string is accepted. The number of iterations is fixed by the length of the strings. Each character in the string is extracted and assigned to the variable SI. The variable is called a counter variable and it cannot be assigned or declared.

Example

```
DECLARE
    s VARCHAR2 (20):='&s';
    s1 VARCHAR2 (20);
BEGIN
    FOR I IN 1...LENGTH (S)
LOOP
    s1:=s1||SUBSTR(s,-I,1);
    DBMS_OUTPUT.PUT_LINE('The counter value is' ||i).
```

```

END LOOP
DBMS_OUTPUT.PUT_LINE(s1);
END:

```

This displays the value of the counter variable.

By default iteration proceeds upward from the lower bound to the higher bound. However, the keyword REVERSE is used, iteration proceeds downward from the higher bound to the lower bound, as the example given below shows. After each iteration, the loop counter is decremented.

```

FOR i IN 1..3 LOOP _____ assign the values 1,2,3 to i.
Sequence_of statements; _____ executes three times
END LOOP;

```

Nevertheless, the range bounds are to be written in ascending (not descending) order. Inside a FOR loop, the loop counter can be referenced like a constant. So, the loop counter can

appear in expressions but it cannot be assigned values, at the following example shows;

```

FOR ctr IN 1 .. 10 LOOP
.....
IF NOT finished THEN
INSERT INTO .... VALUES(ctr,...); _____ legal
Factor := ctr * 2; _____ legal
ELSE
Ctr := 10; _____ illegal
ENDIF;
END LOOP;

```

ITERATION SCHEMES

The bounds of a loop range can be literals, variables, or expressions, but they must evaluate to integers. For example, the following iteration schemes are legal.

```
j IN 5..5
```

```
k IN REVERSE first..last
```

```
Step IN 0...TRUNC(high/low)*2
```

```
Code IN ASCII('A')... ASCII('J')
```

Loop labels

Like PL/SQL blocks loop can be labeled. The label, an undeclared identifier enclosed by double angle brackets, must appear at the beginning of the loop statement as follows

```
<<label_name>>
```

```
LOOP
```

```
Sequence_of_statements
```

```
END LOOP.
```

Optionally the label name can also appear at the end of the loop statement, as the following example shows;

```
<<my_loop>>
```

```
LOOP
```

```
END LOOP my_loop
```

When labeled loops are nested ending label names can be used to improve readability. With either form of EXIT statements, not only the current loop but also any enclosing loops can be completed. This can be done by labeling the enclosing loop that is to be completed. The label can then be used in an EXIT statement as follows,

```
< outer>>
```

LOOP

.....

LOOP

.....

Exit outer WHENExit both loops

End loop;

.....

End loop outer;

Every enclosing loop up to and including the labeled loop is exited:

EXCEPTION MANAGEMENT:

Exception

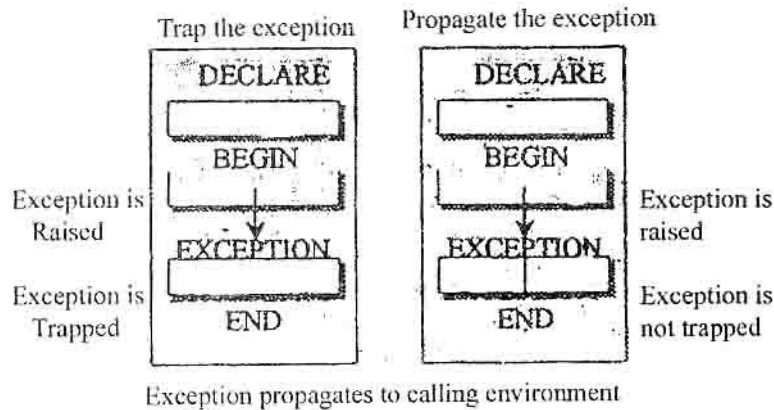
An exception is an identifier in PL/SQL raised during the execution of a block that terminates its main body of actions. A block always terminates when PL/SQL raises an exception, but you specify an exception handler to perform final actions.

Two methods for raising an exceptions

1. An oracle error occurs and the associated exception is raised automatically. For example, if the error ORA -04103 occurs with no rows are retrieved from the database in a select statement, then PL/SQL raises the exception NO_DATA_FOUND.

2. You raise an exception explicitly by issuing the RAISE statement within the block. The exception being raised may be either user defined or predefined.

Handling exceptions



ERRORS

Runtime errors arise from design faults, mistakes, hardware failures, and many other sources. Although these errors are not anticipated, these errors can be handled meaningfully. To capture the errors raised, exceptions are used. In PL/SQL warnings or error condition is called Exceptions. When an error occurs, an exception is raised. That is normal execution is stopped and control transferred to the exception – handling part of the PL/SQL block. They are designed for runtime rather than compile time errors. They are handled in the Exception section of the PL/SQL block. Errors can be classified as

- ❖ Runtime Errors
- ❖ Compile-time Errors

Exceptions can be broadly classified into

- ❖ Pre- defined Exceptions
- ❖ User defined Exceptions
- ❖ Un defined Exceptions

Pre defined exceptions

Pre-defined exceptions are exceptions that are already defined by Oracle. The following list gives the set of predefined exception.

Advantages of exceptions

Using exceptions for error handling has several advantages.

Without exceptions handling, everytime a command is issued, execution errors must be checked , as follows,

```
BEGIN
SELECT....
____check for "no data found " error
SELECT.....
____check for "no data found" error
SELECT.....
_____check for "no data found" error
```

Error processing is not clearly separated from normal processing; nor is it robust. If you neglect to code a check, the error goes undetected and is likely to cause other, seemingly unrelated errors.

With exceptions, errors can be conveniently handled with out the need to code multiple checks as follows;

```
BEGIN
SELECT....
SELECT....
SELECT...
....
```

Exception

WHEN NO_DATA _FOUND THEN- catches all "no data found" errors.

Exception improve, readability by letting error handling routines to be isolated Error recovery algorithms do not obscure the primary algorithm. Exceptions also improve reliability. You need not worry about checking for an error at every point it might occur. Just add an exception handler to your PL/SQL block. If the exception is ever raised in that block (or any sub block) , you can be sure it will be handled.

The following examples illustrates the usage of pre denned exceptions.

Zero - Divide

When a number is divided by zero, this exception is raised. The following example briefs this exceptions.

Example

```
DECLARE
X NUMBER :=& X;
V NUMBER := 8iY;
BEGIN
DBMS_OUTPUT.PUT LINE ( ' RESULT IS' || XY):
END:
```

In this example, two values are accepted for x and y respectively. If I both contains non zero values the result is printed. If 'y' contains zero , it leads to the exception which is displayed as:

```
ORA — 01476: divisor is equal to zero
```

In order to handle the exception, exception clause must contain the Exception
Zen Divide Refining the above example,

```
DECLARE
X NUMBER :=& X
Y NUMBER :=& Y
BEGIN
DBMS_OUTPUT.PUT_LINE ("RESULT IS " || XY);
EXCEPTION
WHEN ZERO DIVIDE THEN
DBMS_OUTPUT.PUT_LINE ("VALUE IS DIVIDED BY ZERO");
END;
```

The second example illustrates the usage of NO_DATA_FOUND

Example

```
DECLARE
MYENAME VARCHAR2 (20);
```

```

BEGIN
SELECT ENAME INTO MYENAME FROM EMP WHERE EMPNO=&X;
  DBMS_OUTPUT.PUT_LINE ('THE NAME OF THE EMPLOYEE IS'
  || MYENAME)
EXCEPTION
WHEN NO_DATA_FOUND THEN('NO RECORD FOUND')
END;
/

```

A SELECT statement returns not more than one row. If a SELECT statement is made to return more than one record, a cursor is required. The following example shows this:

Example

```

DECLARE
  MYSAL NUMBER;
  SELECT SAL INTO MYSAL FROM BM? WHERE DEPTNO=10;
DBMS_OUTPUT.PUT_LINE (MYSAL);
END;

```

This would display.

ORA - 01422 exact fetch returns more than requested number of rows.

Raises the pre-defined exception too many rows. To avoid this handle the exception in the Exception clause.

```

DECLARE
  MYSAL NUMBER;

```

```

BEGIN
SELECT SAL INTO MYSAL FROMEMP WHERE
DBMS_OUTPUT.PUT_LINE(MYSAL);
EXCEPTION
WHEN TOO_MANY_ROWS THEN
DEMS_OUTPUT.PUT_LINE ('Use cursors')
END;

```

The given above example illustrated the usage of handling pre-defined exceptions in the exception section of any PL/SQL block. Likewise, all the other user-defined exceptions can be handled in the following fashion.

Using Others

To handle any kind of an exception use OTHERS. This is a handler that handles any exception. The example follows

Unhandled Exceptions

Remember, if it cannot find a handler for a raised exception, PL/SQL returns an unhandled exception error to the host environment which determines the outcome. For example,

in the Oracle Precompiles environment, any database changes made by a failed SQL statement or PL/SQL blocks are rolled back.

Unhandled exceptions can also affect subprograms. If you exit a subprogram successfully, PL/SQL assigns values to OUT parameters. However, if you exit with an unhandled exception, PL/SQL does not assign values to OUT parameters. Also, if a stored sub program falls with an unhandled exception, PL/SQL does not roll back database work done by the sub program

Unhandled exceptions can be avoided by coding an OTHERS handler at the top most level of every PL/SQL block and sub program

How exceptions propagate

When an exception is raised, if PL/SQL cannot find a handler for it in the current block or subprogram, the exception propagates. That is, the exception reproduces itself in successive enclosing blocks until a handler is found or there are no more blocks to search. In the later part, PL/SQL returns an unhandled exceptions error to the host environment.

Cursor management

Why cursors?

When a query inside a PL/SQL block returns more than one record or one set of data. Oracle requires a place holder to place the values. The variables provided in the INTO clause can contain only one value at a time. In order to process Multiple records, CURSORS are used.

Types of Cursors

PL/SQL implicitly declares a cursor for all SQL data manipulation. Statements, including queries that return only one row. Cursors can be of 2 types:

- ❖ Implicit cursors
- ❖ Explicit cursors

Implicit cursors

Whenever a SQL statement is issued the Database server opens an area of memory in which the command is parsed and executed. This area is called a cursor. When the executable part of PL/SQL block issues a SQL command. PL/SQL creates an implicit cursor, which has the identifier SQL, PL/SQL manages this cursor.

Explicit cursor:

SELECT statements that occurs within PL/SQL blocks are known as embedded. They must return one row and may only return one row. To get

around this a SELECT Statement is defined as a cursor (an area or memory), the query is executed and the returned rows are manipulated within the cursors. Explicit cursors can be of two types

- ❖ Static cursors
- ❖ Dynamic cursors

Static cursors

Static cursors are a type of cursors where the SELECT statement is given at compile time itself. That is, the table from which the data are coming and the records that are going to be selected are predetermined at compile time itself. The definition of the cursor is done in the declarative part.

Dynamic cursors

Dynamic cursors, as the name suggests, are a set of cursors where the records from the tables are selected at runtime rather than at compile time. Dynamic cursors are dealt in the latter part of this chapter.

Each cursor has four attributes;

% ROW COUNT	Returns the number of rows processed by a SQL Statement.
%FOUND	Holds TRUE if at least one row is processed.
% NOT FOUND	Holds TRUE if no rows are processed.
% ISOPEN	Holds TRUE if a cursor is open or FALSE if cursor has not been opened or has been closed. They are used only in connection with explicit cursor.

%ROW COUNT:

The % ROW COUNT Attribute is used to return the number of records fetched. This is in accordance with the number of FETCHES done.

% FOUND

This attribute contains TRUE if the FETCH statement fetches any records. The attribute will hold FALSE if there are no records to be fetched.

% NOT FOUND

it is the logical opposite of % FOUND . If records are fetched, the % NOT FOUND is FALSE and TRUE,if there are no more records to be processed. Using this we can terminate from the loop.

%ISOPEN

This attribute checks for the status of if the status of the cursor is open or not. If the cursor is opened , it holds TRUE and FALSE, if the cursor is not opened.

The tabular structure illustrates this better.

Attribute	Is TRUE	Is FALSE
% ISOPEN	If the cursor is opened	if the cursor is not opened
% FOUND	When records are fetched using FETCH statement	If there are no more records to be fetched and processed
%NOT FOUND	When there are no records available to be fetched	When records are fetched by the FETCH statement
%ROW COUNT	Hold the number of records	-----

Explicit Cursors

Explicit cursor manipulation is performed using Four commands

- ❖ DECLARE
- ❖ OETCH PEN
- ❖ FETCH
- ❖ CLOSE

Declaring a cursor

Cursor declaration defines the name and structure of the cursor together with the SELECT statement that will populate the cursor with data. The query is validated but not executed. The keyword CURSOR is used to declare a cursor.

The syntax to declare a cursor is:

Syntax

```
CURSOR <cursor name > IS <query
```

Example

```
CURSOR CI is SELECT ename, job FROM emp
```

Opening a cursor

After declaring the cursor, it needs to be opened for manipulation in the executable section. Opening a cursor means that the query is executed and the rows are populated in the cursor.

Note that the declaration of the cursor does not mean that the query is executed and records are selected. Only opening a cursor performs this operation.

Syntax

```
OPEN <cursor name>
```

Example

```
OPEN c1;
```

FETCHING RECORDS FROM THE CURSOR

FETCH statement loads the row addressed by the cursor pointer into variables, moves the cursor pointer on to the next row ready for the next fetch. After each fetch, the cursor pointer moves to the next row in the result set.

Syntax

```
FETCH < cursorname> INTO <variable list>
```

Example

```
FETCH C1 INTO x,
```

Here assume the variables, x and y are already declared. For each column that is used in the SELECT list a corresponding variable in the INTO list Must appear. Other wise it leads to an error.

Closing a cursor

CLOSE statement releases the data within the cursor and closes it. The cursor can be reopened to refresh its data.

Using Table type

Tabletype is a collection datatype that stores the table values and it can be accessed. It consists of various collection methods.

Syntax

Type < typename> is table of<datatype> index by binary integer.

Here type name indicates the name of the type and the datatype is the type a can hold and values by binary_integer specifies that it can hold a dynamic range of values

CURSOR FOR UPDATE

Very often, the processing done in a fetch loop modifies the rows that have been retrieved by the cursor. PL/SQL provides a convenient syntax for doing this. This method consists of two parts.

The FOR UPDATE clause in the cursor declaration

The WHERE CURRENT OF clause in an UPDATE or DELETE statement

FOR UPDATE

The FOR UPDATE clause identifies the rows that will be updated or deleted and then locks the rows in the result set.

The syntax is

```
SELECT...FROM...FOR UPDATE[OF column_reference]
[NOWAIT]
```

Where `column_reference` is a column in the table against which the query is performed. A list of columns can also be used.

Cursor Variables

All of the `explicitcursor` examples we have seen so far are examples of static cursors. The cursor is associated with one SQL statement and this statement is determined when the compilation of PL/SQL takes place.

A cursor variable, on the other hand, can be associated with different statements at run time. Cursor variables are analogous to PL/SQL variables, which can hold different values at run time. Static cursors are analogous to PL/SQL constants, since they can only be associated with one run-time query.

In order to use a cursor variable, it must first be declared. Storage for it must then be allocated at run time, since a cursor variable is a REF type. This means that, it is a reference type to a cursor. From this point opening; fetching of records and closing are similar to those of Static Cursors

Declaring a cursor variable

SUBPROGRAMS

Subprograms are named PL/SQL blocks that can: take parameters and be invoked. PL/SQL has two types of sub programs called procedures and functions. Generally, a procedure is used to perform an action and a function to compute a value

Like unnamed or anonymous PL/SQL blocks, subprograms have a declarative part, an executable part, and an optional exception —handling part. The declarative part contains declarations of types, cursors, constants, variables, exceptions and nested subprograms. Items are local and cease to exist when you exit the subprogram. The executable part contains statements, that is: assign values, control execution and manipulate Oracle data. The exception handling part contains exception handlers, which deal with exceptions raised during execution.

Advantages of subprograms

Subprograms provide extensibility, that is, they let you tailor the PL/SQL language to suit your needs. Subprograms also provide modularity that is, they let you break a program down into manageable, well-defined logic modules. This supports top-down design and the stepwise refinement approach to problem solving

Also subprograms promote reusability and maintainability. Once validated, a subprogram can be used with confidence in any number of applications. Furthermore, only the subprogram is affected if its definition changes. This simplifies maintenance and enhancement

Finally, subprograms and abstraction the mental separation from particulars. To use subprograms you must know what they do, not how they work. Therefore, you can design applications from the top down without worrying about implementation details. Dummy subprograms - (stubs) allow you, to defer the definitions of procedures and functions until you test and debug the main program.

Procedures to Parameters

A procedure is a subprogram that performs a specific action. The syntax for creating procedure :

```
.CREATE OR REPLACE PROCEDURE
```

```
<Procedure name>[parameter list] As
```

```
PL/SQL statements
```

A Procedure contains two parts

- Specifications
- Body

The specification contains the procedure name, the parameter and the body contains the executable statements.

Parameters

Parameters are the values that are passed to the subprogram for computing values or performing specific actions. In-turn the subprograms can return some values which can be the output of the subprogram. Based on this, the parameters are classified as

- **ACTUAL**
- **FORMAL**

ACTUAL PARAMETERS

They are the variables defined in the parameter list of a subprogram call.

FORMAL PARAMETERS

They are the variables given while declaring subprograms,

Parameter modes

Parameter modes define the behavior of the formal parameters. There are three parameter modes

- ❖ IN
- ❖ OUT
- ❖ INOUT

IN MODE

IN mode is used to read values from the subprogram. It is a read only variable where reading alone is done and no assignment is performed. Consider the following example that illustrates the usage of the IN mode.

Example

```
CREATE OR REPLACE PROCEDURE DISP_ENAME
(ENO IN NUMBER) IS MNAME VARCHAR2 (20);
BEGIN
SELECT ENAME INTO MNAME FROM EMP WHERE EMPNO
= ENO;
```

```
DBMS_OUTPUT.PUTLINE ('THE NAME IS' || MNAME);  
END;
```

The procedure gets created. In order to execute the procedure, the statement EXEC is used as shown below:

```
EXEC disp_ename(7902)
```

Displays ,

```
EXEC DISP _ ENAME (7902);
```

The name is FORD

PL/SQL procedure successfully completed.

This procedure uses the IN mode and displays the name of the employee.

OUT MODE

The OUT parameter is used to return values to the program. Rather it is a write - Only variable. The value can be assigned to this variable. Consider a situation where the employee name and the salary are to be displayed. The following example shows the usage of the OUT parameter.

```
CREATE OR REPLACE PROCEDURE DISP_SAL (ENO IN NUMBER,  
SALARY OUT NUMBER) AS
```

```
  MNAME VARCHAR2(20);
```

```
  BEGIN
```

```
    SELECT SAL, ENAME INTO SALARY, MNAME FROM EMP  
    WHERE EMPNO=ENO;
```

```
    DBMS_OUTPUT.PUT LINE ('THE EMPLOYEE NAME IS' ||  
    MNAME);
```

Execution can not be done like the execution of the previous example. Since there is an OUT parameter, a place holder, must be provided for the out variable value to be assigned. The two steps in execution of this program at SQL prompt are:

First declare a session variable or local variable using the keyword VAR as shown below:

VAR <variable name><data type>

Next, execute the procedure with the arguments specified and replace the out parameter with the variable defined

```
EXEC <procedure name> (value ,;<variable name>);
```

Since, the variable is a session variable, session variables can be called using colon delimiter (Refer PL / SQL delimiters)

Finally, print the value placed on the variable using PRINT statement

```
PRINT <variable name>
```

The example and the output is shown below:

```
VAR n NUMBER
```

```
EXEC disp_sal(7902,:n);
```

The employee name is FORD

PL /SQL procedure successfully completed.

```
PRINT n
```

```
N
```

```
-----
```

```
3000
```

Alternatively this procedure can be executed inside a PL/SQL block as shown below:

```
DECLARE
```

```
MSAL NUMBER;
```

```
MENONUMBER:=&N;
```

```
BEGIN
```

```

DISP SAL(MENO, MSAL);
DBMS_OUTPUT.PUT_LINE ('THE SALARY IS ' || MSAL);
END;

```

When this block gets executed, it displays

Entervalue for n:7902

old 3Meno number := &n

new 3Meno number :=7902;

the employee name is FORD

the salary is 3000

PL/SQL procedure successfully completed.

IN OUT Mode

The INOUT parameter is used to pass initial values to the subprogram. It is a read- write variable and can be used for both reading and writing values. Inside the sub program the IN out parameter acts like an un-initialized variable. An example for displaying the product name and the price is shown below.

```

CREATE OR REPLACE PROCEDURE disp_prod
(no IN OUT NUMBER, name OUT VARCHAR2) AS
BEGIN
SELECT pname, ucSt INTO name, no FROM product WHERE pcode=no;
END;

```

The output of the above program is,

Procedure created.

The procedure, which contains an IN OUT parameter, cannot be executed as SQL prompt. A subprogram that contains an IN OUT parameter must be

executed only inside a PL/ SQL block. The below program shows how to execute an IN OUT parameter.

Example :

```
DECLARE.  
  
Name VARCHAR 2(20)  
Var_x NUMBER; = & X;  
  
BEGIN  
DISP_PROD (VAR_X, NAME);  
DBMS_OUTPUT.PUT LINE ('name is'||name||'rate is'||var_x);  
END;
```

The Program accepts the number and the rate is assigned to the same variable and the name of the product is assigned to the variable name.

The output would look like ;

Enter value for x : 102

old 3 : Var x NUMBER:=&X;

new 3: Var x NUMBER:=102;

name is storewell rate is 10000

Passing Default Values

As the example below shows, you can initialize IN parameters to default values. That way, different numbers of actual parameters can be passed to a subprogram, accepting or overriding the default values.

```
PROCEDURE create_dcpt  
new_dname CHAR DEFAULT 'TEMP',  
new_loc CHAR DEFAULT 'TEMP') IS  
BEGIN  
  
INSERT INTO dept  
VALUES (deptno_seq.NEXTVAL, new-dname, new_loc);
```

If an actual parameter is not passed, the default value at its corresponding formal parameter is used. Consider the following calls to `Create_dept`:

```
Exec create_dept;
```

```
Exec create_dept ('MARKETING'),
```

```
Exec create_dept ('MARKETING','NEW YORK')
```

The first call passes no actual parameters, so both default values are used. The Second call passes one actual parameter, so the default value for `new_loc` is used. The third call passes two actual parameters, so neither default value is used.

Forward Declarations

PL /SQL requires a declaration an identifier before using it. For example, the following declaration of procedure `award_bonus` is illegal because `award_bonus` calls procedure `calc_rating`, which is not yet declared when the call is made.

```
DECLARE
```

```
PROCEDURE award_bonus (...) IS
```

```
BEGIN
```

```
calc_rating(...); - - undeclared identifier
```

```
...
```

```
END;
```

```
PROCEDURE calc_rating (...) IS
```

```
BEGIN
```

```
....
```

```
END;
```

In this case, this problem can be solved easily by placing procedure `calc_rating` before procedure `award_bonus`. However, the easy solution does not always work. For example, suppose the procedures are mutually recursive (call each other) or they are defined in alphabetical order.

PL/SQL solves this problem by providing a special subprogram declaration called a forward declaration. You can use forward declarations to

- define subprograms in logical or alphabetical order .
- define mutually recursive sub programs
- group sub programs in a package

A forward declaration consists of a subprogram specification .terminated by a semicolon. the following example, the forward declaration advises PL/SQL that the body of procedure `calc_rating` can be found later in the block:

```
DECLARE
PROCEDURE    calc_rating (...); -- forward declaration
....
/* Define subprograms in alphabetical order*/
PROCEDURE award bonus (...) IS
BEGIN
calc _rating(...);
....
END;
PROCEDURE calc_rating (...) IS
BEGIN
....
END;
```

Although the formal parameter list appears in the forward declaration. it must also appear the subprogram body. The sub program body can be placed anywhere after the forward declaration, butthey must appear in the sameprogram unit.

Functions of Notations

A function is a program that computes a value. Functions and procedures are structured e, except that functions have a RETURN clause. Functions are created by using the following syntax:

CREATE OR REPLACE FUNCTIONS

<FUNCTIONNAME>[parameter list] RETURN datatype IS

PL/SQL Statements

Return <value/variable>;

END;

Procedures Vs Functions

The following table lists out the differences between procedures and functions.

PROCEDURES	FUNCTIONS
Used to perform a specific task	Used to Calculate Values
Does not return values (can be explicitly returned using OUT mode)	Must return 1 value using Return Statement (can be made to return more than 1 value using OUT mode)

Viewing Procedures and Functions :

Procedures and Functions are viewed using the Data dictionary Views:

SELECT * FROM USER_SOURCE;

Dropping Subprograms

Subprograms when not required can be dropped using Drop command. **Syntax :**

DROP FUNCTION <function name>;

DROP PROCEDURE <procedure name>;